

# js闭包:Javascript闭包

疯狂代码 <http://CrazyCoder.cn/>     <http://CrazyCoder.cn/Javascript/Article3851.html>

简介

Closure

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。闭包是 ECMAScript（JavaScript）最强大的特性之一，但用好闭包的前提是必须理解闭包。闭包的创建相对容易，人们甚至会在不经意间创建闭包，但这些无意创建的闭包却存在潜在的危害，尤其是在比较常见的浏览器环境下。如果想要扬长避短地使用闭包这一特性，则必须了解它们的工作机制。而闭包工作机制的实现很大程度上有赖于标识符（或者说对象属性）解析过程中作用域的角色。关于闭包，最简单的描述就是 ECMAScript 允许使用内部函数——即函数定义和函数表达式位于另一个函数的函数体内。而且，这些内部函数可以访问它们所在的外部函数中声明的所有局部变量、参数和声明的其他内部函数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。也就是说，内部函数会在外部函数返回后被执行。而当这个内部函数执行时，它仍然必需访问其外部函数的局部变量、参数以及其他内部函数。这些局部变量、参数和函数声明（最初时）的值是外部函数返回时的值，但也会受到内部函数的影响。遗憾的是，要适当地理解闭包就必须理解闭包背后运行的机制，以及许多相关的技术细节。虽然本文的前半部分并没有涉及 ECMA 262 规范指定的某些算法，但仍然有许多无法回避或简化的内容。对于个别熟悉对象属性名解析的人来说，可以跳过相关的内容，但是除非你对闭包也非常熟悉，否则最好是不要跳过下面几节。对象属性名解析

ECMAScript 认可两类对象：原生（Native）对象和宿主（Host）对象，其中宿主对象包含一个被称为内置对象的原生对象的子类（ECMA 262 3rd Ed Section 4.3）。原生对象属于语言，而宿主对象由环境提供，比如说可能是文档对象、DOM 等类似的对象。原生对象具有松散和动态的命名属性（对于某些实现的内置对象子类而言，动态性是受限的——但这不是太大的问题）。对象的命名属性用于保存值，该值可以是指向另一个对象（Objects）的引用（在这个意义上说，函数也是对象），也可以是一些基本的数据类型，比如：String、Number、Boolean、Null 或 Undefined。其中比较特殊的是 Undefined 类型，因为可以给对象的属性指定一个 Undefined 类型的值，而不会删除对象的相应属性。而且，该属性只是保存着 undefined 值。下面简要介绍一下如何设置和读取对象的属性值，并最大程度地体现相应的内部细节。值的赋予

对象的命名属性可以通过为该命名属性赋值来创建，或重新赋值。即，对于：`var objectRef = new Object();`  
`//创建一个普通的 javascript 对象。`

可以通过下面语句来创建名为“testNumber”的属性：`objectRef.testNumber = 5;`

`/* - 或- */`

`objectRef["testNumber"] = 5;`

在赋值之前，对象中没有“testNumber”属性，但在赋值后，则创建一个属性。之后的任何赋值语句都不需要再创建这个属性，而只会重新设置它的值：`objectRef.testNumber = 8;`

`/* - 或- */`

`objectRef["testNumber"] = 8;`

稍后我们会介绍，Javascript 对象都有原型（prototypes）属性，而这些原型本身也是对象，因而也可以带有命名的属性。但是，原型对象命名属性的作用并不体现在赋值阶段。同样，在将值赋给其命名属性时，如果对

象没有该属性则会创建该命名属性，否则会重设该属性的值。值的读取

当读取对象的属性值时，原型对象的作用便体现出来。如果对象的原型中包含属性访问器（property accessor）所使用的属性名，那么该属性的值就会返回：/\* 为命名属性赋值。如果在赋值前对象没有相应的属性，那么赋值后就会得到一个：\*/

```
objectRef.testNumber = 8; /* 从属性中读取值 */ var val = objectRef.testNumber;
```

```
/* 现在， - val - 中保存着刚赋给对象命名属性的值 8*/
```

而且，由于所有对象都有原型，而原型本身也是对象，所以原型也可能有原型，这样就构成了所谓的原型链。

原型链终止于链中原型为 null 的对象。Object 构造函数的默认原型就有一个 null 原型，因此：var

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

创建了一个原型为 Object.prototype 的对象，而该原型自身则拥有一个值为 null 的原型。也就是说

，objectRef 的原型链中只包含一个对象 - - Object.prototype。但对于下面的代码而言：/\* 创建 -

```
MyObject1 - 类型对象的函数*/
```

```
function MyObject1(formalParameter){
```

```
/* 给创建的对象添加一个名为 - testNumber -
```

```
的属性并将传递给构造函数的第一个参数指定为该属性的值：*/
```

```
this.testNumber = formalParameter;
```

```
}/ /* 创建 - MyObject2 - 类型对象的函数*/
```

```
function MyObject2(formalParameter){
```

```
/* 给创建的对象添加一个名为 - testString -
```

```
的属性并将传递给构造函数的第一个参数指定为该属性的值：*/
```

```
this.testString = formalParameter;
```

```
}
```

```
/* 接下来的操作 MyObject1 类的实例替换了所有与 MyObject2
```

```
类的实例相关联的原型。而且，为 MyObject1 构造函数传递了参数
```

```
- 8 - ，因而其 - testNumber - 属性被赋予该值：*/
```

```
MyObject2.prototype = new MyObject1( 8 );
```

```
/* 最后，将一个字符串作为构造函数的第一个参数，
```

```
创建一个 - MyObject2 - 的实例，并将指向该对象的
```

```
引用赋给变量 - objectRef - : */ var objectRef = new MyObject2( "String_Value" );
```

被变量 objectRef 所引用的 MyObject2 的实例拥有一个原型链。该链中的第一个对象是在创建后被指定给

MyObject2 构造函数的 prototype 属性的 MyObject1 的一个实例。MyObject1 的实例也有一个原型，即与

Object.prototype 所引用的对象对应的默认的 Object 对象的原型。最后，Object.prototype 有一个值为

null 的原型，因此这条原型链到此结束。当某个属性访问器尝试读取由 objectRef 所引用的对象的属性值时

，整个原型链都会被搜索。在下面这种简单的情况下：var val = objectRef.testString;

因为 objectRef 所引用的 MyObject2 的实例有一个名为 “testString” 的属性，因此被设置为

“String\_Value” 的该属性的值被赋给了变量 val。但是：var val = objectRef.testNumber;

则不能从 MyObject2 实例自身中读取到相应的命名属性值，因为该实例没有这个属性。然而，变量 val 的值仍然被设置为 8，而不是未定义 - - 这是因为在该实例中查找相应的命名属性失败后，解释程序会继续检查其原型对象。而该实例的原型对象是 MyObject1 的实例，这个实例有一个名为“testNumber”的属性并且值为 8，所以这个属性访问器最后会取得值 8。而且，虽然 MyObject1 和 MyObject2 都没有定义 toString 方法，但是当属性访问器通过 objectRef 读取 toString 属性的值时：`var val = objectRef.toString;` 变量 val 也会被赋予一个函数的引用。这个函数就是在 Object.prototype 的 toString 属性中所保存的函数。之所以会返回这个函数，是因为发生了搜索 objectRef 原型链的过程。当在作为对象的 objectRef 中发现没有“toString”属性存在时，会搜索其原型对象，而当原型对象中不存在该属性时，则会继续搜索原型的原型。而原型链中最终的原型是 Object.prototype，这个对象确实有一个 toString 方法，因此该方法的引用被返回。最后：`var val = objectRef.madeUpProperty;`

返回 undefined，因为在搜索原型链的过程中，直至 Object.prototype 的原型 - - null，都没有找到任何对象有名为“madeUpProperty”的属性，因此最终返回 undefined。不论是在对象或对象的原型中，读取命名属性值的时候只返回首先找到的属性值。而当为对象的命名属性赋值时，如果对象自身不存在该属性则创建相应的属性。这意味着，如果执行像 `objectRef.testNumber = 3` 这样一条赋值语句，那么这个 MyObject2 的实例自身也会创建一个名为“testNumber”的属性，而之后任何读取该命名属性的尝试都将获得相同的新值。这时候，属性访问器不会再进一步搜索原型链，但 MyObject1 实例值为 8 的“testNumber”属性并没有被修改。给 objectRef 对象的赋值只是遮挡了其原型链中相应的属性。注意：ECMAScript 为 Object 类型定义了一个内部 `[[prototype]]` 属性。这个属性不能通过脚本直接访问，但在属性访问器解析过程中，则需要用到这个内部 `[[prototype]]` 属性所引用的对象链 - - 即原型链。可以通过一个公共的 prototype 属性，来对与内部的 `[[prototype]]` 属性对应的原型对象进行赋值或定义。这两者之间的关系在 ECMA 262 (3rd edition) 中有详细描述，但超出了本文要讨论的范畴。标识符解析、执行环境和作用域链

#### 执行环境

执行环境是 ECMAScript 规范 (ECMA 262 第 3 版) 用于定义 ECMAScript 实现必要行为的一个抽象的概念。对如何实现执行环境，规范没有作规定。但由于执行环境中包含引用规范所定义结构的相关属性，因此执行环境中应该保有 (甚至实现) 带有属性的对象 - - 即使属性不是公共属性。所有 JavaScript 代码都是在一个执行环境中被执行的。全局代码 (作为内置的 JS 文件执行的代码，或者 HTML 页面加载的代码) 是在我将称之为“全局执行环境”的执行环境中执行的，而对函数的每次调用 (有可能是作为构造函数) 同样有关联的执行环境。通过 eval 函数执行的代码也有截然不同的执行环境，但因为 JavaScript 程序员在正常情况下一般不会使用 eval，所以这里不作讨论。有关执行环境的详细说明请参阅 ECMA 262 (第 3 版) 第 10.2 节。当调用一个 JavaScript 函数时，该函数就会进入相应的执行环境。如果又调用了另外一个函数 (或者递归地调用同一个函数)，则又会创建一个新的执行环境，并且在函数调用期间执行过程都处于该环境中。当调用的函数返回后，执行过程会返回原始执行环境。因而，运行中的 JavaScript 代码就构成了一个执行环境栈。在创建执行环境的过程中，会按照定义的先后顺序完成一系列操作。首先，在一个函数的执行环境中，会创建一个“活动”对象。活动对象是规范中规定的另外一种机制。之所以称之为对象，是因为它拥有可访问的命名属性，但是它又不像正常对象那样具有原型 (至少没有预定义的原型)，而且不能通过 JavaScript 代码直接引用活动对象。为函数调用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，它以

整数索引的数组成员——对应地保存着调用函数时所传递的参数。这个对象也有 `length` 和 `callee` 属性（这两个属性与我们讨论的内容无关，详见规范）。然后，会为活动对象创建一个名为“arguments”的属性，该属性引用前面创建的 arguments 对象。接着，为执行环境分配作用域。作用域由对象列表（链）组成。每个函数对象都有一个内部的 `[[scope]]` 属性（该属性我们稍后会详细介绍），这个属性也由对象列表（链）组成。指定给一个函数调用执行环境的作用域，由该函数对象的 `[[scope]]` 属性所引用的对象列表（链）组成，同时，活动对象被添加到该对象列表的顶部（链的前端）。之后会发生由 ECMA 262 中所谓“可变”对象完成的“变量实例化”的过程。只不过此时使用活动对象作为可变对象（这里很重要，请注意：它们是同一个对象）。此时会将函数的形式参数创建为可变对象命名属性，如果调用函数时传递的参数与形式参数一致，则将相应参数的值赋给这些命名属性（否则，会给命名属性赋 `undefined` 值）。对于定义的内部函数，会以其声明时所用名称为可变对象创建同名属性，而相应的内部函数则被创建为函数对象并指定给该属性。变量实例化的最后一步是将在函数内部声明的所有局部变量创建为可变对象的命名属性。根据声明的局部变量创建的可变对象的属性在变量实例化过程会被赋予 `undefined` 值。在执行函数体内的代码、并计算相应的赋值表达式之前不会对局部变量执行真正的实例化。事实上，拥有 arguments 属性的活动对象和拥有与函数局部变量对应的命名属性的可变对象是同一个对象。因此，可以将标识符 arguments 作为函数的局部变量来看待。

回到顶部最后，在 `this` 可以被使用之前，还必须先对其赋值。如果赋的值是一个对象的引用，则 `this.m` 访问的便是该对象上的 `m`。如果（内部）赋的值是 `null`，则 `this` 就指向全局对象。（此段由 pangba 刘未鹏 翻译）

（原文备考：Finally a value is assigned for use with the `this` keyword. If the value assigned refers to an object then property accessors prefixed with the `this` keyword reference properties of that object. If the value assigned (internally) is `null` then the `this` keyword will refer to the global object.）创建全局执行环境的过程会稍有不同，因为它没有参数，所以不需要通过定义的活动对象来引用这些参数。但全局执行环境也需要一个作用域，而它的作用域链实际上只由一个对象——全局对象——组成。全局执行环境也会有变量实例化的过程，它的内部函数就是涉及大部分 JavaScript 代码的、常规的顶级函数声明。而且，在变量实例化过程中全局对象就是可变对象，这就是为什么全局性声明

的函数是全局对象属性的原因。全局性声明的变量同样如此。全局执行环境也会使用 `this` 对象来引用全局对象。作用域链与 `[[scope]]`

调用函数时创建的执行环境会包含一个作用域链，这个作用域链是通过将该执行环境的活动（可变）对象添加到保存于所调用函数对象的 `[[scope]]` 属性中的作用域链前端而构成的。所以，理解函数对象内部的 `[[scope]]` 属性的定义过程至关重要。在 ECMAScript 中，函数也是对象。函数对象在变量实例化过程中会根据函数声明来创建，或者是在计算函数表达式或调用 `Function` 构造函数时创建。通过调用 `Function` 构造函数创建的函数对象，其内部的 `[[scope]]` 属性引用的作用域链中始终只包含全局对象。通过函数声明或函数表达式创建的函数对象，其内部的 `[[scope]]` 属性引用的则是创建它们的执行环境的作用域链。在最简单的情况下，比如声明如下全局函数：

```
- function exampleFunction(formalParameter){
... // 函数体内的代码
}
```

当为创建全局执行环境而进行变量实例化时，会根据上面的函数声明创建相应的函数对象。因为全局执行环境的作用域链中只包含全局对象，所以它就给自己创建的、并以名为“exampleFunction”的属性引用的这个函

数对象的内部 `[[scope]]` 属性，赋予了只包含全局对象的作用域链。当在全局环境中计算函数表达式时，也会发生类似的指定作用域链的过程：`- var exampleFuncRef = function(){`

```
... // 函数体代码
```

```
}
```

在这种情况下，不同的是在全局执行环境的变量实例化过程中，会先为全局对象创建一个命名属性。而在计算赋值语句之前，暂时不会创建函数对象，也不会将该函数对象的引用指定给全局对象的命名属性。但是，最终还是会在全局执行环境中创建这个函数对象（当计算函数表达式时。译者注），而为此创建的函数对象的 `[[scope]]` 属性指定的作用域链中仍然只包含全局对象。内部的函数声明或表达式会导致在包含它们的外部函数的执行环境中创建相应的函数对象，因此这些函数对象的作用域链会稍微复杂一些。在下面的代码中，先定义了一个带有内部函数声明的外部函数，然后调用外部函数：`function`

```
exampleOuterFunction(formalParameter){
```

```
function exampleInnerFuncitonDec(){
```

```
... // 内部函数体代码
```

```
}
```

```
... // 其余的外部函数体代码
```

```
}exampleOuterFunction( 5 );
```

与外部函数声明对应的函数对象会在全局执行环境的变量实例化过程中被创建。因此，外部函数对象的

`[[scope]]` 属性中会包含一个只有全局对象的“单项目”作用域链。当在全局执行环境中调用

`exampleOuterFunction` 函数时，会为该函数调用创建一个新的执行环境和一个活动（可变）对象。这个新执行环境的作用域就由新的活动对象后跟外部函数对象的 `[[scope]]` 属性所引用的作用域链（只有全局对象）构成。在新执行环境的变量实例化过程中，会创建一个与内部函数声明对应的函数对象，而同时会给这个函数对象的

`[[scope]]` 属性指定创建该函数对象的执行环境（即新执行环境。译者注）的作用域值 - - 即一个包含活动对象后跟全局对象的作用域链。到目前为止，所有过程都是自动、或者由源代码的结构所控制的。但我们发现，

执行环境的作用域链定义了执行环境所创建的函数对象的 `[[scope]]` 属性，而函数对象的 `[[scope]]` 属性则定义了它的执行环境的作用域（包括相应的活动对象）。不过，ECMAScript 也提供了用于修改作用域链 `with` 语句。

`with` 语句会计算一个表达式，如果该表达式是一个对象，那么就将这个对象添加到当前执行环境的作用域链中（在活动<可变>对象之前）。然后，执行 `with` 语句（它自身也可能是一个语句块）中的其他语句。之后，又恢复到调用它之前的执行环境的作用域链中。`with` 语句不会影响在变量实例化过程中根据函数声明创建

函数对象。但是，可以在一个 `with` 语句内部对函数表达式求值：`- /* 创建全局变量 - y - 它引用一个对象： -`

```
*/
```

```
var y = {x:5}; // 带有一个属性 - x - 的对象直接量
```

```
function exampleFuncWith(){
```

```
var z;
```

```
/* 将全局对象 - y - 引用的对象添加到作用域链的前端： - */
```

```
with(y){
```

```
/* 对函数表达式求值以创建函数对象并将该函数对象的引用指定给局部变量 - z - :- */
```

```

z = function(){
... // 内部函数表达式中的代码;
}
}
...
}/* 执行 - exampleFuncWith - 函数:- */
exampleFuncWith();

```

在调用 exampleFuncWith 函数所创建的执行环境中包含一个由其活动对象后跟全局对象构成的作用域链。而在执行 with 语句时，又会把全局变量 y 引用的对象添加到这个作用域链的前端。在对其中的函数表达式求值的过程中，所创建函数对象的 [[scope]] 属性与创建它的执行环境的作用域保持一致 - - 即，该属性会引用一个由对象 y 后跟调用外部函数时所创建执行环境的活动对象，后跟全局对象的作用域链。当与 with 语句相关的语句块执行结束时，执行环境的作用域得以恢复（y 会被移除），但是已经创建的函数对象（z。译者注）的 [[scope]] 属性所引用的作用域链中位于最前面的仍然是对象 y。标识符解析

标识符是沿作用域链逆向解析的。ECMA 262 将 this 归类为关键字而不是标识符，并非不合理。因为解析 this 值时始终要根据使用它的执行环境来判断，而与作用域链无关。标识符解析从作用域链中的第一个对象开始。检查该对象中是否包含与标识符对应的属性名。因为作用域链是一条对象链，所以这个检查过程也会包含相应对象的原型链（如果有）。如果没有在作用域链的第一个对象中发现相应的值，解析过程会继续搜索下一个对象。这样依次类推直至找到作用域链中包含以标识符为属性名的对象为止，也有可能作用域链的所有对象中都没有发现该标识符。当基于对象使用属性访问器时，也会发生与上面相同的标识符解析过程。当属性访问器中有相应的属性可以替换某个对象时，这个属性就成为表示该对象的标识符，该对象在作用域链中的位置进而被确定。全局对象始终都位于作用域链的尾端。因为与函数调用相关的执行环境将会把活动（可变）对象添加到作用域链的前端，所以在函数体内使用的标识符会首先检查自己是否与形式参数、内部函数声明的名称或局部变量一致。这些都可以由活动（可变）对象的命名属性来确定。闭包

#### 自动垃圾收集

ECMAScript 要求使用自动垃圾收集机制。但规范中并没有详细说明相关的细节，而是留给了实现来决定。但据了解，相当一部分实现对它们的垃圾收集操作只赋予了很低的优先级。但是，大致的思想都是相同的，即如果对象不再“可引用（由于不存在对它的引用，使执行代码无法再访问到它）”时，该对象就成为垃圾收集的目标。因而，在将来的某个时刻会将这个对象销毁并将它所占用的一切资源释放，以便操作系统重新利用。正常情况下，当退出一个执行环境时就会满足类似的条件。此时，作用域链结构中的活动（可变）对象以及在该执行环境中创建的任何对象 - - 包括函数对象，都不再“可引用”，因此将成为垃圾收集的目标。构成闭包闭包是通过在对一个函数调用的执行环境中返回一个函数对象构成的。比如，在对函数调用的过程中，将一个对内部函数对象的引用指定给另一个对象的属性。或者，直接将这样一个（内部）函数对象的引用指定给一个全局变量、或者一个全局性对象的属性，或者一个作为参数以引用方式传递给外部函数的对象。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 的内部函数的引用 :- */
return exampleReturned;
}var globalVar = exampleClosureForm(2, 4);

```

这种情况下，在调用外部函数 exampleClosureForm 的执行环境中所创建的函数对象就不会被当作垃圾收集，因为该函数对象被一个全局变量所引用，而且仍然是可以访问的，甚至可以通过 globalVar(n) 来执行。的确，情况比正常的时候要复杂一些。因为现在这个被变量 globalVar 引用的内部函数对象的 [[scope]] 属性所引用的作用域链中，包含着属于创建该内部函数对象的执行环境的活动对象（和全局对象）。由于在执行被 globalVar 引用的函数对象时，每次都要把该函数对象的 [[scope]] 属性所引用的整个作用域链添加到创建的（内部函数的）执行环境的作用域中（即此时的作用域中包括：内部执行环境的活动对象、外部执行环境的活动对象、全局对象。译者注），所以这个（外部执行环境的）活动对象不会被当作垃圾收集。闭包因此而构成。此时，内部函数对象拥有自由的变量，而位于该函数作用域链中的活动（可变）对象则成为与变量绑定的环境。由于活动（可变）对象受限于内部函数对象（现在被 globalVar 变量引用）的 [[scope]] 属性中作用域链的引用，所以活动对象连同它的变量声明 - - 即属性的值，都会被保留。而在对内部函数调用的执行环境中进行作用域解析时，将会把与活动（可变）对象的命名属性一致的标识符作为该对象的属性来解析。活动对象的这些属性值即使是在创建它的执行环境退出后，仍然可以被读取和设置。在上面的例子中，当外部函数返回（退出它的执行环境）时，其活动（可变）对象的变量声明中记录了形式参数、内部函数定义以及局部变量的值。arg1 属性的值为 2，而 arg2 属性的值为 4，localVar 的值是 8，还有一个 exampleReturned 属性，它引用由外部函数返回的内部函数对象。（为方便起见，我们将在后面的讨论中，称这个活动<可变>对象为“ActOuter1”）。如果再次调用 exampleClosureForm 函数，如：- var secondGlobalVar = exampleClosureForm(12, 3);

- 则会创建一个新的执行环境和一个新的活动对象。而且，会返回一个新的函数对象，该函数对象的 [[scope]] 属性引用的作用域链与前一次不同，因为这一次的作用域链中包含着第二个执行环境的活动对象，而这个活动对象的属性 arg1 值为 12 而属性 arg2 值为 3。（为方便起见，我们将在后面的讨论中，称这个活动<可变>对象为“ActOuter2”）。通过第二次执行 exampleClosureForm 函数，第二个、也是截然不同的闭包诞生了。通过执行 exampleClosureForm 创建的两个函数对象分别被指定给了全局变量 globalVar 和 secondGlobalVar，并返回了表达式 ((arg1 + arg2)/(innerArg + localVar))。该表达式对其中的四个标识符应用了不同的操作符。如何确定这些标识符的值是体现闭包价值的关键所在。我们来看一看，在执行由 globalVar 引用的函数对象 - - 如 globalVar(2) - - 时的情形。此时，会创建一个新的执行环境和相应的活动对象（我们将称之为“ActInner1”），并把该活动对象添加到执行的函数对象的 [[scope]] 属性所引用的作用域链的前端。ActInner1 会带有一个属性 innerArg，根据传递的形式参数，其值被指定为 2。这个新执行环境的作用域链变成：ActInner1->ActOuter1->全局对象。为了返回表达式 ((arg1 + arg2)/(innerArg + localVar)) 的值，要沿着作用域链进行标识符解析。表达式中标识符的值将通过依次查找作用域链中每个对象（与标识符名称一致）的属性来确定。作用域链中的第一个对象是 ActInner1，它有一个名为 innerArg 的属性，值是 2。所有其他三个标识符在 ActOuter1 中都有对应的属性：arg1 是 2，arg2 是 4 而 localVar 是 8。

最后，函数调用返回  $((2 + 2)/(2 + 8))$ 。现在再来看一看由 `secondGlobalVar` 引用的同一个函数对象的执行情况，比如 `secondGlobalVar(5)`。我们把这次创建的新执行环境的活动对象称为“ActInner2”，相应的作用域链就变成了：ActInner2->ActOuter2->全局对象。ActInner2 返回 `innerArg` 的值 5，而 ActOuter2 分别返回 `arg1`、`arg2` 和 `localVar` 的值 12、3 和 8。函数调用返回的值就是  $((12 + 3)/(5 + 8))$ 。如果再执行一次 `secondGlobalVar`，则又会有一个新活动对象被添加到作用域链的前端，但 ActOuter2 仍然是链中的第二个对象，而他的命名属性会再次用于完成标识符 `arg1`、`arg2` 和 `localVar` 的解析。这就是 ECMAScript 的内部函数获取、维持和访问创建他们的执行环境的形式参数、声明的内部函数以及局部变量的过程。这个过程说明了构成闭包以后，内部的函数对象在其存续过程中，如何维持对这些值的引用、如何对这些值进行读取的机制。即，创建内部函数对象的执行环境的活动（可变）对象，会保留在该函数对象的 `[[scope]]` 属性所引用的作用域链中。直到所有对这个内部函数的引用被释放，这个函数对象才会成为垃圾收集的目标（连同它的作用域链中任何不再需要的对象）。内部函数自身也可能有内部函数。在通过函数执行返回内部函数构成闭包以后，相应的闭包自身也可能会返回内部函数从而构成它们自己的闭包。每次作用域链嵌套，都会增加由创建内部函数对象的执行环境引发的新活动对象。ECMAScript 规范要求作用域链是临时性的，但对作用域链的长度却没有加以限制。在具体实现中，可能会存在实际的限制，但还没有发现有具体限制数量的报告。目前来看，嵌套的内部函数所拥有的潜能，仍然超出了使用它们的人的想像能力。通过闭包可以做什么？

对这个问题的回答可能会令你惊讶 - - 闭包什么都可以做。据我所知，闭包使得 ECMAScript 能够模仿任何事物，因此局限性在于设计和实现要模仿事物的能力。只是从字面上看可能会觉得这么说很深奥，下面我们就来看一些更有实际意义的例子。例 1：为函数引用设置延时

闭包的一个常见用法是在执行函数之前为要执行的函数提供参数。例如：将函数作为 `setTimeout` 函数的第一个参数，这在 Web 浏览器的环境下是非常常见的一种应用。`setTimeout` 用于有计划地执行一个函数（或者一串 JavaScript 代码，不是在本例中），要执行的函数是其第一个参数，其第二个参数是以毫秒表示的执行间隔。也就是说，当在一段代码中使用 `setTimeout` 时，要将一个函数的引用作为它的第一个参数，而将以毫秒表示的时间值作为第二个参数。但是，传递函数引用的同时无法为计划执行的函数提供参数。然而，可以在代码中调用另外一个函数，由它返回一个对内部函数的引用，再把这个对内部函数对象的引用传递给 `setTimeout` 函数。执行这个内部函数时要使用的参数在调用返回它的外部函数时传递。这样，`setTimeout` 在执行这个内部函数时，不用传递参数，但该内部函数仍然能够访问在调用返回它的外部函数时传递的参数：

```
function callLater(paramA, paramB, paramC){
  /* 返回一个由函数表达式创建的匿名内部函数的引用:- */ return (function(){
  /* 这个内部函数将通过 - setTimeout - 执行，
  而且当它执行时它会读取并按照传递给
  外部函数的参数行事：
  */
  paramA[paramB] = paramC;
  });
}.../* 调用这个函数将返回一个在其执行环境中创建的内部函数对象的引用。
```

传递的参数最终将作为外部函数的参数被内部函数使用。

返回的对内部函数的引用被赋给一个全局变量:-

```
*/var functRef = callLater(elStyle, "display", "none");
```

```
/* 调用 setTimeout 函数，将赋给变量 - functRef -
```

```
的内部函数的引用作为传递的第一个参数:- */hideMenu=setTimeout(functRef, 500);
```

例 2: 通过对象实例方法关联函数回到顶部许多时候我们需要将一个函数对象暂时挂到一个引用上留待后面执行，因为不等到执行的时候是很难知道其具体参数的，而先前将它赋给那个引用的时候更是压根不知道的。（此段由 pangba 刘未鹏 翻译）（luyy朋友的翻译\_2008-7-7更新）很多时候需要将一个函数引用进行赋值，以便在将来某个时候执行该函数，在执行这些函数时给函数提供参数将会是有用处的，但这些参数在执行时不容易获得，他们只有在上面赋值给时才能确定。（原文备考：There are many other circumstances when a reference to a function object is assigned so that it would be executed at some future time where it is useful to provide parameters for the execution of that function that would not be easily available at the time of execution but cannot be known until the moment of assignment.）一个相关的例子是，用 JavaScript 对象来封装与特定 DOM 元素的交互。这个 JavaScript 对象具有 doOnClick、doMouseOver 和 doMouseOut 方法，并且当用户在该特定的 DOM 元素中触发了相应的事件时要执行这些方法。不过，可能会创建与不同的 DOM 元素关联的任意数量的 JavaScript 对象，而且每个对象实例并不知道实例化它们的代码将会如何操纵它们（即注册事件处理函数与定义相应的事件处理函数分离。译者注）。这些对象实例并不知道如何在全局环境中引用它们自身，因为它们不知道将会指定哪个全局变量（如果有）引用它们的实例。因而问题可以归结为执行一个与特定的 JavaScript 对象关联的事件处理函数，并且要知道调用该对象的哪个方法。下面这个例子使用了一个基于闭包构建的一般化的函数（此句多谢未鹏指点），该函数会将对象实例与 DOM 元素事件关联起来，安排执行事件处理程序时调用对象实例的指定方法，给象的指定方法传递的参数是事件对象和与元素关联的引用，该函数返回执行相应方法后的返回值。/\* 一个关联对象实例和事件处理器的函数。

它返回的内部函数被用作事件处理器。对象实例以 - obj - 参数表示，

而在该对象实例中调用的方法名则以 - methodName - （字符串）参数表示。

```
*/function associateObjWithEvent(obj, methodName){
```

```
/* 下面这个返回的内部函数将作为一个 DOM 元素的事件处理器*/ return (function(e){
```

```
/* 在支持标准 DOM 规范的浏览器中，事件对象会被解析为参数 - e - ，
```

```
若没有正常解析，则使用 IE 的事件对象来规范化事件对象。
```

```
*/ e = e||window.event;
```

```
/* 事件处理器通过保存在字符串 - methodName - 中的方法名调用了对象
```

```
- obj - 的一个方法。并传递已经规范化的事件对象和触发事件处理器的元素
```

```
的引用 - this - （之所以 this 有效是因为这个内部函数是作为该方法执行的）
```

```
*/ return obj[methodName](e, this);
```

```
});
```

```
/* 这个构造函数用于创建将自身与 DOM 元素关联的对象，
```

```
DOM 元素的 ID 作为构造函数的字符串参数。
```

```
所创建的对象会在相应的元素触发 _disiblevent= function(){
```

... // 方法体。

};

### Internet Explorer 的内存泄漏问题

Internet Explorer Web 浏览器（在 IE 4 到 IE 6 中核实）的垃圾收集系统中存在一个问题，即如果 ECMAScript 和某些宿主对象构成了“循环引用”，那么这些对象将不会被当作垃圾收集。此时所谓的宿主对象指的是任何 DOM 节点（包括 document 对象及其后代元素）和 ActiveX 对象。如果在一个循环引用中包含了一或多个这样的对象，那么这些对象直到浏览器关闭都不会被释放，而它们所占用的内存同样在浏览器关闭之前都不会交回系统重用。当两个或多个对象以首尾相连的方式相互引用时，就构成了循环引用。比如对象 1 的一个属性引用了对象 2，对象 2 的一个属性引用了对象 3，而对象 3 的一个属性又引用了对象 1。对于纯粹的 ECMAScript 对象而言，只要没有其他对象引用对象 1、2、3，也就是说它们只是相互之间的引用，那么仍然会被垃圾收集系统识别并处理。但是，在 Internet Explorer 中，如果循环引用中的任何对象是 DOM 节点或者 ActiveX 对象，垃圾收集系统则不会发现它们之间的循环关系与系统中的其他对象是隔离的并释放它们。最终它们将被保留在内存中，直到浏览器关闭。闭包非常容易构成循环引用。如果一个构成闭包的函数对象被指定给，比如一个 DOM 节点的事件处理器，而对该节点的引用又被指定给函数对象作用域中的一个活动（或可变）对象，那么就存在一个循环引用。DOM\_Node.onevent -> function\_object.[[scope]] -> scope\_chain -> Activation\_object.nodeRef -> DOM\_Node。形成这样一个循环引用是轻而易举的，而且稍微浏览一下包含类似循环引用代码的网站（通常会出现在网站的每个页面中），就会消耗大量（甚至全部）系统内存。多加注意可以避免形成循环引用，而在无法避免时，也可以使用补偿的方法，比如使用 IE 的 onunload 事件来来清空（null）事件处理函数的引用。时刻意识到这个问题并理解闭包的工作机制是在 IE 中避免此类问题的关键。

2008-8-21 21:10:53

疯狂代码 <http://CrazyCoder.cn/>