### 闭包的解释

闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包的常见方式，就是在一个函数内部创建另一个函数，createComparisonFunction() 函数为例。

|  |
| --- |
| function createComparisonFunction(propertyName) {  return function(object1, object2){  **var value1 = object1[propertyName];**  **var value2 = object2[propertyName];**  if (value1 < value2){  return -1;  } else if (value1 > value2){  return 1;  } else {  return 0;  }  };  } |

在这个例子中，突出的那两行代码是内部函数（一个匿名函数）中的代码，这两行代码访问了外部函数中的变量 propertyName 。即使这个内部函数被返回了，而且是在其他地方被调用了，但它仍然可以访问变量 propertyName 。之所以还能够访问这个变量，是因为内部函数的作用域链中包含createComparisonFunction() 的作用域。要彻底搞清楚其中的细节，必须从理解函数被调用的时候都会发生什么入手。

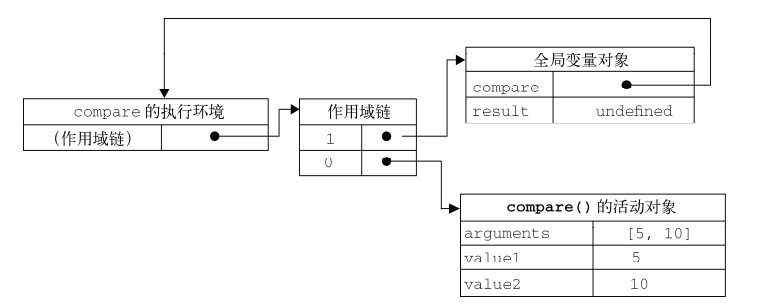
当某个函数被调用时，会创建一个执行环境（execution context）及相应的作用域链。

然后，使用 arguments 和其他命名参数的值来初始化函数的活动对象（activation object）。但在作用域链中，外部函数的活动对象始终处于第二位，外部函数的外部函数的活动对象处于第三位，……直至作为作用域链终点的全局执行环境。

**在函数执行过程中，为读取和写入变量的值，就需要在作用域链中查找变量。**

|  |
| --- |
| function compare(value1, value2){  if (value1 < value2){  return -1;  } else if (value1 > value2){  return 1;  } else {  return 0;  }  }  var result = compare(5, 10); |

以上代码先定义了 compare() 函数，然后又在全局作用域中调用了它。当调用 compare() 时，会创建一个包含 arguments 、 value1 和 value2 的活动对象。全局执行环境的变量对象（包含 result和 compare ）在 compare() 执行环境的作用域链中则处于第二位。图展示了包含上述关系的compare() 函数执行时的作用域链。



后台的每个执行环境都有一个表示变量的对象——变量对象。全局环境的变量对象始终存在，而像compare() 函数这样的局部环境的变量对象，则只在函数执行的过程中存在。在创建 compare() 函数时，会创建一个预先包含全局变量对象的作用域链，这个作用域链被保存在内部的 [[Scope]] 属性中。

当调用 compare() 函数时，会为函数创建一个执行环境，然后通过复制函数的 [[Scope]] 属性中的对象构建起执行环境的作用域链。此后，又有一个活动对象（在此作为变量对象使用）被创建并被推入执行环境作用域链的前端。对于这个例子中 compare() 函数的执行环境而言，其作用域链中包含两个变量对象：本地活动对象和全局变量对象。显然，作用域链本质上是一个指向变量对象的指针列表，它只引用但不实际包含变量对象。

无论什么时候在函数中访问一个变量时，就会从作用域链中搜索具有相应名字的变量。一般来讲，当函数执行完毕后，局部活动对象就会被销毁，内存中仅保存全局作用域（全局执行环境的变量对象）。

但是，闭包的情况又有所不同。

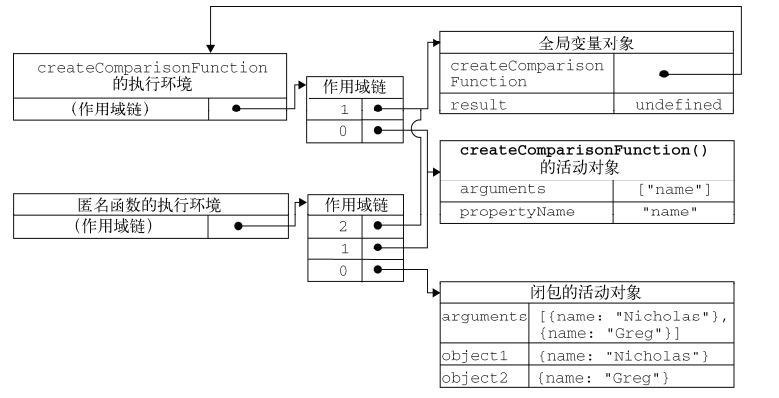
在另一个函数内部定义的函数会将包含函数（即外部函数）的活动对象添加到它的作用域链中。因此，在 createComparisonFunction() 函数内部定义的匿名函数的作用域链中，实际上将会包含外部函数 createComparisonFunction() 的活动对象。下图展示了当下列代码执行时，包含函数与内部匿名函数的作用域链。

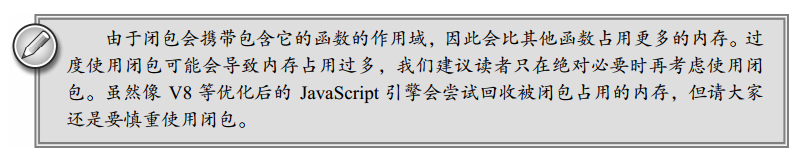
|  |
| --- |
| var compare = createComparisonFunction("name");  var result = compare({ name: "Nicholas" }, { name: "Greg" }); |

在匿名函数从 createComparisonFunction() 中被返回后，它的作用域链被初始化为包含createComparisonFunction() 函数的活动对象和全局变量对象。这样，匿名函数就可以访问在createComparisonFunction() 中定义的所有变量。更为重要的是， createComparisonFunction()函数在执行完毕后，其活动对象也不会被销毁，因为匿名函数的作用域链仍然在引用这个活动对象。换句话说，当 createComparisonFunction() 函数返回后，其执行环境的作用域链会被销毁，但它的活动对象仍然会留在内存中；直到匿名函数被销毁后， createComparisonFunction() 的活动对象才会被销毁，例如：

|  |
| --- |
| //创建函数  var compareNames = createComparisonFunction("name");  //调用函数  var result = compareNames({ name: "Nicholas" }, { name: "Greg" });  //解除对匿名函数的引用（以便释放内存）  compareNames = null; |

首先，创建的比较函数被保存在变量 compareNames 中。而通过将 compareNames 设置为等于 null解除该函数的引用，就等于通知垃圾回收例程将其清除。随着匿名函数的作用域链被销毁，其他作用域（除了全局作用域）也都可以安全地销毁了。下图展示了调用 compareNames() 的过程中产生的作用域链之间的关系。





事实上，每个函数都可以被认为是一个闭包。因为每个函数都在其所在域（即该函数的作

用域）中维护了某种私有联系。但在大多数时候，该作用域在函数体执行完之后就自行销毁了—除非发生一些有趣的事，**导致作用域被保持**。

根据目前的讨论，**我们可以说，如果一个函数会在其父级函数返回之后留住对父级作**

**用域的链接的话，相关闭包就会被创建起来**。但其实每个函数本身就是一个闭包，因为每

个函数至少都有访问全局作用域的权限，而全局作用域是不会被破坏的。

### 循环中的闭包

接下来，让我们来看看在闭包问题上会犯哪些典型的错误。毕竟由闭包所导致的 bug 往往很难被发现，因为它们总是表面上看起来一切正常。

让我们来看一个三次的循环操作，它在每次迭代中都会创建一个返回当前循环序号的

新函数。该新函数会被添加到一个数组中，并最终返回。具体代码如下：

|  |
| --- |
| function F() {  var arr = [], i;  for (i = 0; i < 3; i++) {  arr[i] = function () {  return i;  };  }  return arr;  } |

下面，我们来运行一下函数，并将结果赋值给数组 arr。

|  |
| --- |
| > var arr = F(); |

现在，我们拥有了一个包含三个函数的数组。您可以通过在每个数组元素后面加一对括号来调用它们。按通常的估计，它们应该会依照循环顺序分别输出 0、 1 和 2，下面就让我们来试试：

|  |
| --- |
| > arr[0]();  3  > arr[1]();  3  > arr[2]();  3 |

显然，这并不是我们想要的结果。究竟是怎么回事呢？原来我们在这里创建了三个闭包，而它们都指向了一个共同的局部变量 i。但是，闭包并不会记录它们的值，它们所拥有的只是相关域在创建时的一个连接（即引用）。在这个例子中，变量 i 恰巧存在于定义这三个函数域中。对这三个函数中的任何一个而言，当它要去获取某个变量时，它会从其所在的域开始逐级寻找那个距离最近的 i 值。由于循环结束时 i 的值为 3，所以这三个函数都指向了这一共同值。

那么，应该如何纠正这种行为呢？答案是换一种闭包形式：

|  |
| --- |
| function F() {  var arr = [], i;  for(i = 0; i < 3; i++) {  arr[i] = (function (x){  return function () {  return x;  }  }(i));  }  return arr;  } |

这样就能获得我们预期的结果了：

|  |
| --- |
| > var arr = F();  > arr[0]();  0  > arr[1]();  1  > arr[2]();  2 |

在这里，我们不再直接创建一个返回 i 的函数了，而是将 i 传递给了另一个即时函数。在该函数中， i 就被赋值给了局部变量 x，这样一来，每次迭代中的 x 就会拥有各自不同的值了。

或者，我们也可以定义一个“正常点的”内部函数（不使用即时函数）来实现相同的

功能。要点是在每次迭代操作中，我们要在中间函数内将 i 的值“本地化”。

|  |
| --- |
| function F() {  function binder(x) {  return function(){  return x;  };  }  var arr = [], i;  for(i = 0; i < 3; i++) {  arr[i] = binder(i);  }  return arr;  } |