**第七章函数表达式（193）**

定义函数的方式有两种：一种是函数声明，另一种就是函数表达式。用函数声明定义函数的时候，Firefox、Safari、Chrome和 Opera 都给函数定义了一个非标准的 name 属性，通过这个属性可以访问到给函数指定的名字。

//只在 Firefox、Safari、Chrome 和 Opera 有效

alert(functionName.name); //"functionName"

关于函数声明，它的一个重要特征就是函数声明提升（function declaration hoisting），意思是在执行代码之前会先读取函数声明。

第二种创建函数的方式是使用函数表达式。函数表达式有几种不同的语法形式。下面是最常见的一种形式。

var functionName = function(arg0, arg1, arg2){

//函数体

};

这种情况下创建的函数叫做匿名函数（anonymous function），因为 function 关键字后面没有标识符。（匿名函数有时候也叫拉姆达函数。）匿名函数的 name 属性是空字符串。函数表达式与其他表达式一样，在使用前必须先赋值。

理解函数提升的关键，就是理解函数声明与函数表达式之间的区别。例如，执行以下代码的结果可能会让人意想不到。

//不要这样做！请使用函数表达式

if(condition){

function sayHi(){

alert("Hi!");

}

} else {

function sayHi(){

alert("Yo!");

}

}

1. 递归

function factorial(num){

if (num <= 1){

return 1;

} else {

return num \* arguments.callee(num-1);

}

}

但在严格模式下，不能通过脚本访问 arguments.callee，访问这个属性会导致错误。不过，可以使用命名函数表达式来达成相同的结果。

var factorial = (function f(num){

if (num <= 1){

return 1;

} else {

return num \* f(num-1);

}

});

1. 闭包

有不少开发人员总是搞不清匿名函数和闭包这两个概念，因此经常混用。闭包是指有权访问另一个函数作用域中的变量的函数。

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 {

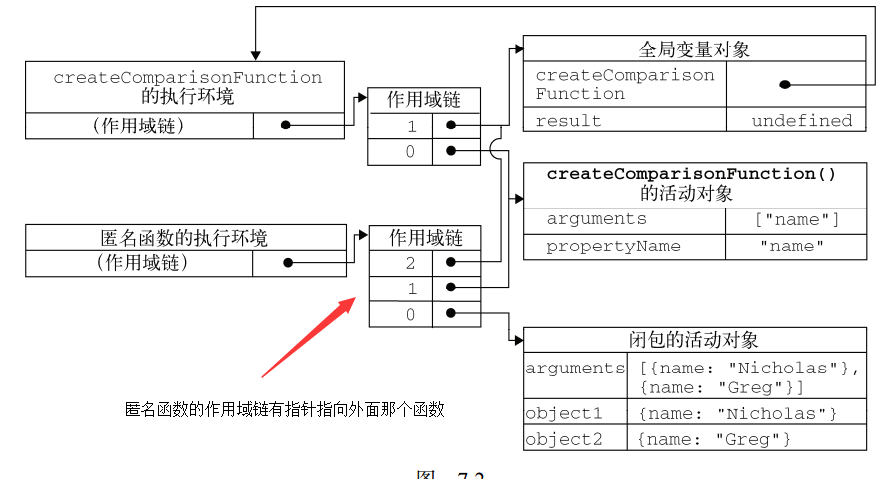
return 0;

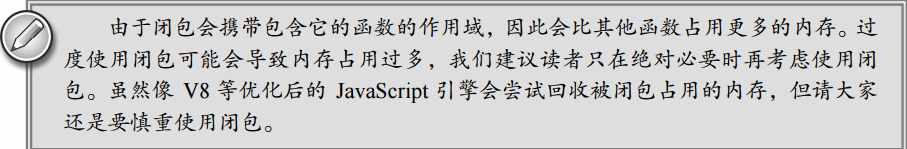
}

};

}

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。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。





**闭包与变量**

作用域链的这种配置机制引出了一个值得注意的副作用，即闭包只能取得包含函数中任何变量的最后一个值。（for循环经典问题）

function createFunctions(){

var result = new Array();

for (var i=0; i < 10; i++){

result[i] = function(){

return i;

};

}

return result;

}

每个函数的作用域链中都保存着 createFunctions() 函数的活动对象，所以它们引用的都是同一个变量 i 。但是，我们可以通过创建另一个匿名函数强制让闭包的行为符合预期。

function createFunctions(){

var result = new Array();

for (var i=0; i < 10; i++){

result[i] = function(num){

return function(){

return num;

};

}(i);

}

return result;

}

我们没有直接把闭包赋值给数组，而是定义了一个匿名函数，并将立即执行该匿名函数的结果赋给数组。

**关于this对象**

在闭包中使用 this 对象也可能会导致一些问题。我们知道，this 对象是在运行时基于函数的执行环境绑定的。，匿名函数的执行环境具有全局性，因此其 this 对象通常指向 window。但有时由于编写闭包的方式不同，这一点可能不会那么明显。

var name = "The Window";

var object = {

name : "My Object",

getNameFunc : function(){

return function(){

return this.name;

};

}

};

alert(object.getNameFunc()()); //"The Window"（在非严格模式下）

在定义匿名函数之前，我们把 this对象**赋值给了一个名叫 that** 的变量，就能解决问题。在几种特殊情况下，this 的值可能会意外地改变。

var name = "The Window";

var object = {

name : "My Object",

getName: function(){

return this.name;

}

};

object.getName(); //"My Object"

(object.getName)(); //"My Object"

(object.getName = object.getName)(); //"The Window"，在非严格模式下

第三行代码先执行了一条赋值语句，然后再调用赋值后的结果。因为这个赋值表达式的值是函数本身，所以 this 的值不能得到维持，结果就返回了"The Window"。

**内存泄漏**

function assignHandler(){

var element = document.getElementById("someElement");

element.onclick = function(){

alert(element.id);

};

}

以上代码创建了一个作为 element 元素事件处理程序的闭包，而这个闭包则又创建了一个循环引用（事件将在第 13 章讨论）。由于匿名函数保存了一个对 assignHandler()的活动对象的引用，因此就会导致无法减少 element 的引用数。

var element = document.getElementById("someElement");

var id = element.id;

element.onclick = function(){

alert(id);

};

element = null;

}

必须要记住：闭包会引用包含函数的整个活动对象，而其中包含着 element。即使闭包不直接引用 element，包含函数的活动对象中也仍然会保存一个引用。因此，**有必要把 element 变量设置为 null。**

1. 模仿块级作用域

如前所述，JavaScript 没有块级作用域的概念。这意味着在块语句中定义的变量，实际上是在包含函数中而非语句中创建的。

function outputNumbers(count){

for (var i=0; i < count; i++){

alert(i);

}

**var i; //重新声明变量**

alert(i); //计数，10，i没有在for循环的块级作用域结束后销毁

}

遇到这种情况，它只会对后续的声明视而不见。匿名函数可以用来模仿块级作用域并避免这个问题。

function(){

//这里是块级作用域

}(); //出错！

这段代码会导致语法错误，是因为 JavaScript 将 function 关键字当作一个**函数声明的开始**，而**函数声明后面不能跟圆括号**。然而，函数表达式的后面可以跟圆括号。**要将函数声明转换成函数表达式**，只要像下面这样给它加上一对圆括号即可。

function outputNumbers(count){

(function () {

for (var i=0; i < count; i++){

alert(i);

}

})();

alert(i); //导致一个错误！

}

变量 i 只能在循环中使用，使用后即被销毁。而在私有作用域中能够访问变量 count，是因为这个匿名函数是一个闭包。

