

第 7 章

函数表达式


本章内容

- 函数表达式的特征
- 使用函数实现递归
- 使用闭包定义私有变量

函数表达式是 JavaScript 中的一个既强大又容易令人困惑的特性。第 5 章曾介绍过，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。函数声明的语法是这样的。

```
function functionName(arg0, arg1, arg2) {  
    //函数体  
}
```


首先是 `function` 关键字，然后是函数的名字，这就是指定函数名的方式。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。这个属性的值永远等于跟在 `function` 关键字后面的标识符。



```
//只在 Firefox、Safari、Chrome 和 Opera 有效  
alert(functionName.name); // "functionName"
```

FunctionNameExample01.htm

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。这就意味着可以把函数声明放在调用它的语句后面。



```
sayHi();  
function sayHi(){  
    alert("Hi!");  
}
```

FunctionDeclarationHoisting01.htm

这个例子不会抛出错误，因为在代码执行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  
};
```

这种形式看起来好像是常规的变量赋值语句，即创建一个函数并将它赋值给变量 `functionName`。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。以下代码会导致错误。

```
sayHi();    //错误：函数还不存在
var sayHi = function(){
    alert("Hi!");
};
```

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！
if(condition){
    function sayHi(){
        alert("Hi!");
    }
} else {
    function sayHi(){
        alert("Yo!");
    }
}
```

FunctionDeclarationsErrorExample01.htm

表面上看，以上代码表示在 `condition` 为 `true` 时，使用一个 `sayHi()` 的定义；否则，就使用另一个定义。实际上，这在 ECMAScript 中属于无效语法，JavaScript 引擎会尝试修正错误，将其转换为合理的状态。但问题是浏览器尝试修正错误的做法并不一致。大多数浏览器会返回第二个声明，忽略 `condition`；Firefox 会在 `condition` 为 `true` 时返回第一个声明。因此这种使用方式很危险，不应该出现在你的代码中。不过，如果是使用函数表达式，那就没有什么问题了。

```
//可以这样做
var sayHi;

if(condition){
    sayHi = function(){
        alert("Hi!");
    };
} else {
    sayHi = function(){
        alert("Yo!");
    };
}
```

这个例子不会有什么意外，不同的函数会根据 `condition` 被赋值给 `sayHi`。

能够创建函数再赋值给变量，也就能够把函数作为其他函数的值返回。还记得第 5 章中的那个 `createComparisonFunction()` 函数吗：

```
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;
        }
    };
}

```

createComparisonFunction() 就返回了一个匿名函数。返回的函数可能会被赋值给一个变量，或者以其他方式被调用；不过，在 createComparisonFunction() 函数内部，它是匿名的。在把函数当成值来使用的情况下，都可以使用匿名函数。不过，这并不是匿名函数唯一的用途。

7.1 递归

递归函数是在一个函数通过名字调用自身的情况下构成的，如下所示。

```

function factorial(num){
    if (num <= 1){
        return 1;
    } else {
        return num * factorial(num-1);
    }
}

```

[RecursionExample01.htm](#)

这是一个经典的递归阶乘函数。虽然这个函数表面看来没什么问题，但下面的代码却可能导致它出错。

```

var anotherFactorial = factorial;
factorial = null;
alert(anotherFactorial(4)); // 出错!

```

[RecursionExample01.htm](#)

以上代码先把 factorial() 函数保存在变量 anotherFactorial 中，然后将 factorial 变量设置为 null，结果指向原始函数的引用只剩下一个。但在接下来调用 anotherFactorial() 时，由于必须执行 factorial()，而 factorial 已经不再是函数，所以就会导致错误。在这种情况下，使用 arguments.callee 可以解决这个问题。

我们知道，arguments.callee 是一个指向正在执行的函数的指针，因此可以用它来实现对函数的递归调用，例如：

```

function factorial(num){
    if (num <= 1){
        return 1;
    } else {
        return num * arguments.callee(num-1);
    }
}

```

[RecursionExample02.htm](#)

加粗的代码显示，通过使用 `arguments.callee` 代替函数名，可以确保无论怎样调用函数都不会出问题。因此，在编写递归函数时，使用 `arguments.callee` 总比使用函数名更保险。

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。例如：

```
var factorial = (function f(num){
    if (num <= 1){
        return 1;
    } else {
        return num * f(num-1);
    }
})();
```

以上代码创建了一个名为 `f()` 的命名函数表达式，然后将它赋值给变量 `factorial`。即便把函数赋值给了另一个变量，函数的名字 `f` 仍然有效，所以递归调用照样能正确完成。这种方式在严格模式和非严格模式下都行得通。

7.2 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。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，仍以前面的 `createComparisonFunction()` 函数为例，注意加粗的代码。

```
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，必须从理解函数第一次被调用的时候都会发生什么入手。

第4章介绍了作用域链的概念。而有关如何创建作用域链以及作用域链有什么作用的细节，对彻底理解闭包至关重要。当某个函数第一次被调用时，会创建一个执行环境（`execution context`）及相应的作用域链，并把作用域链赋值给一个特殊的内部属性（即 `[[Scope]]`）。然后，使用 `this`、`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。来看下面的例子。

```
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。当第一次调用 `compare()` 时，会创建一个包含 `this`、`arguments`、`value1` 和 `value2` 的活动对象。全局执行环境的变量对象（包含 `this`、`result` 和 `compare`）在 `compare()` 执行环境的作用域链中则处于第二位。图 7-1 展示了包含上述关系的 `compare()` 函数执行时的作用域链。

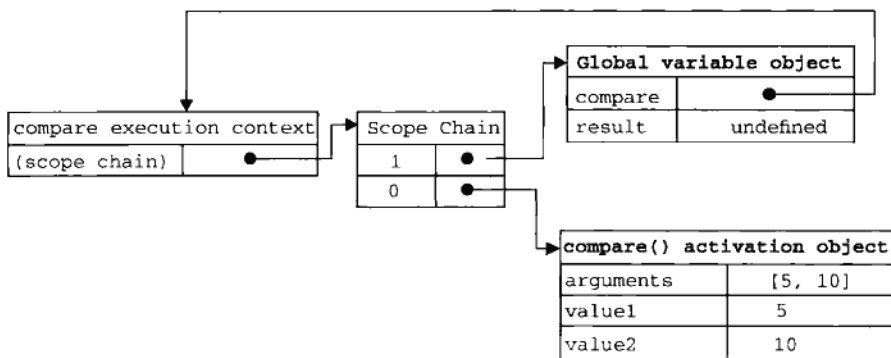


图 7-1

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。图 7-2 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。图 7-2 展示了调用 `compareNames()` 的过程中产生的作用域链之间的关系。

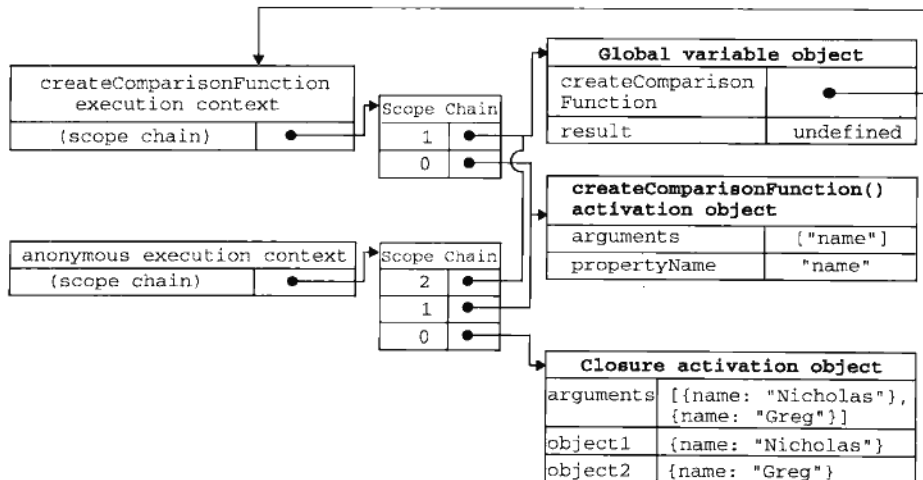


图 7-2



由于闭包会携带包含它的函数的作用域，因此会比其他函数占用更多的内存。过度使用闭包可能会导致内存占用过多，我们建议读者只在必要时再考虑使用闭包。虽然像 V8 等优化后的 JavaScript 引擎会尝试回收被闭包占用的内存，但请大家还是要慎重使用闭包。

7.2.1 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。别忘了闭包所保存的是整个变量对象，而不是某个特殊的变量。下面这个例子可以清晰地说明这个问题。



```
function createFunctions(){
    var result = new Array();

    for (var i=0; i < 10; i++){
        result[i] = function(){
            return i;
        };
    }

    return result;
}
```

ClosureExample01.htm

这个函数会返回一个函数数组。表面上看，似乎每个函数都应该返回自己的索引值，即位置 0 的函数返回 0，位置 1 的函数返回 1，以此类推。但实际上，每个函数都返回 10。因为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。当 createFunctions() 函数返回后，变量 i 的值是 10，此时每个函数都引用着保存变量 i 的同一个变量对象，所以在每个函数内部 i 的值都是 10。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，如下所示。

```
function createFunctions(){
    var result = new Array();

    for (var i=0; i < 10; i++){
        result[i] = function(num){
            return function(){
                return num;
            };
        }(i);
    }

    return result;
}
```

ClosureExample02.htm

在重写了前面的 `createFunctions()` 函数后，每个函数就会返回各自不同的索引值了。在这个版本中，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。这里的匿名函数有一个参数 `num`，也就是最终的函数要返回的值。在调用每个匿名函数时，我们传入了变量 `i`。由于函数参数是按值传递的，所以就会将变量 `i` 的当前值复制给参数 `num`。而在这个匿名函数内部，又创建并返回了一个访问 `num` 的闭包。这样一来，`result` 数组中的每个函数都有自己 `num` 变量的一个副本，因此就可以返回各自不同的数值了。

7.2.2 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：在全局函数中，`this` 等于 `window`，而当函数被作为某个对象的方法调用时，`this` 等于那个对象。不过，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`^①。但有时候由于编写闭包的方式不同，这一点可能不会那么明显。下面来看一个例子。

```
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)
```

[ThisObjectExample01.htm](#)

以上代码先创建了一个全局变量 `name`，又创建了一个包含 `name` 属性的对象。这个对象还包含一个方法——`getNameFunc()`，它返回一个匿名函数，而匿名函数又返回 `this.name`。由于 `getNameFunc()` 返回一个函数，因此调用 `object.getNameFunc()()` 就会立即调用它返回的函数，结果就是返回一个字符串。然而，这个例子返回的字符串是 "The Window"，即全局 `name` 变量的值。为什么匿名函数没有取得其包含作用域（或外部作用域）的 `this` 对象呢？

前面曾经提到过，每个函数在被调用时，其活动对象都会自动取得两个特殊变量：`this` 和 `arguments`。内部函数在搜索这两个变量时，只会搜索到其活动对象为止，因此永远不可能直接访问外部函数中的这两个变量（这一点通过图 7-2 可以看得更清楚）。不过，把外部作用域中的 `this` 对象保存在一个闭包能够访问到的变量里，就可以让闭包访问该对象了，如下所示。

```
var name = "The Window";

var object = {
  name : "My Object",

  getNameFunc : function(){
```

① 当然，在通过 `call()` 或 `apply()` 改变函数执行环境的情况下，`this` 就会指向其他对象。


```

        var that = this;
        return function(){
            return that.name;
        };
    };

    alert(object.getNameFunc()()); // "My Object"

```

ThisObjectExample02.htm

代码中突出的行展示了这个例子与前一个例子之间的不同之处。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。而在定义了闭包之后，闭包也可以访问这个变量，因为它是在包含函数中特意声明的一个变量。即使在函数返回之后，`that` 也仍然引用着 `object`，所以调用 `object.getNameFunc()()` 就返回了 "My Object"。



`this` 和 `arguments` 也存在同样的问题。如果想访问作用域中的 `arguments` 对象，必须将该对象的引用保存到另一个闭包能够访问的变量中。

在几种特殊情况下，`this` 的值可能会意外地改变。比如，下面的代码是修改前面例子的结果。

```

var name = "The Window";

var object = {
    name: "My Object",

    getName: function(){
        return this.name;
    }
};

```

这里的 `getName()` 方法只简单地返回 `this.name` 的值。以下是几种调用 `object.getName()` 的方式以及各自的结果。

```

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

```

ThisObjectExample03.htm

第一行代码跟平常一样调用了 `object.getName()`，返回的是 "My Object"，因为 `this.name` 就是 `object.name`。第二行代码在调用这个方法前先给它加上了括号。虽然加上括号之后，就好像只是在引用一个函数，但 `this` 的值得到了维持，因为 `object.getName` 和 `(object.getName)` 的定义是相同的。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"。

当然，你不大可能会像第二行和第三行代码一样调用这个方法。不过，这个例子有助于说明即使是语法的细微变化，都有可能意外地改变 `this` 的值。

7.2.3 内存泄漏

由于 IE9 之前的版本对 JScript 对象和 COM 对象使用不同的垃圾收集例程（第 4 章曾经讨论过），

因此闭包在 IE 的这些版本中会导致一些特殊的问题。具体来说，如果闭包的作用域链中保存着一个 HTML 元素，那么就意味着该元素将无法被销毁。来看下面的例子。

```
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。只要匿名函数存在，element 的引用数至少也是 1，因此它所占用的内存就永远不会被回收。不过，这个问题可以通过稍微改写一下代码来解决，如下所示。

```
function assignHandler(){
    var element = document.getElementById("someElement");
    var id = element.id;


    element.onclick = function(){
        alert(id);
    };

    element = null;
}
```

在上面的代码中，通过把 element.id 的一个副本保存在一个变量中，并且在闭包中引用该变量消除了循环引用。但仅仅做到这一步，还是不能解决内存泄漏的问题。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。这样就能够解除对 DOM 对象的引用，顺利地减少其引用数，确保正常回收其占用的内存。

7.3 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，来看下面的例子。



```
function outputNumbers(count){
    for (var i=0; i < count; i++){
        alert(i);
    }
    alert(i);    //计数
}
```

BlockScopeExample01.htm

这个函数中定义了一个 for 循环，而变量 i 的初始值被设置为 0。在 Java、C++ 等语言中，变量 i 只会在 for 循环的语句块中有定义，循环一旦结束，变量 i 就会被销毁。可是在 JavaScript 中，变量 i 是定义在 outputNumbers() 的活动对象中的，因此从它有定义开始，就可以在函数内部随处访问它。即使像下面这样错误地重新声明同一个变量，也不会改变它的值。

```
function outputNumbers(count){
    for (var i=0; i < count; i++){
        alert(i);
    }

    var i;          //重新声明变量
    alert(i);        //计数
}
```

BlockScopeExample02.htm

JavaScript 从来不会告诉你是否多次声明了同一个变量；遇到这种情况，它只会对后续的声明视而不见（不过，它会执行后续声明中的变量初始化）。匿名函数可以用来模仿块级作用域并避免这个问题。用作块级作用域（通常称为私有作用域）的匿名函数的语法如下所示。

```
(function(){
    //这里是块级作用域
})();
```

以上代码定义并立即调用了匿名函数。将函数声明包含在一对圆括号中，表示它实际上是一个函数表达式。而紧随其后的另一对圆括号会立即调用这个函数。如果有读者感觉这种语法不太好理解，可以再看看下面这个例子。

```
var count = 5;
outputNumbers(count);
```

这里初始化了变量 `count`，将其值设置为 5。当然，这里的变量是没有必要的，因为可以把值直接传给函数。为了让代码更简洁，我们在调用函数时用 5 来代替变量 `count`，如下所示。

```
outputNumbers(5);
```

这样做之所以可行，是因为变量只不过是值的另一种表现形式，因此用实际的值替换变量没有问题。再看下面的例子。

```
var someFunction = function(){
    //这里是块级作用域
};
someFunction();
```

这个例子先定义了一个函数，然后立即调用了它。定义函数的方式是创建一个匿名函数，并把匿名函数赋值给变量 `someFunction`。而调用函数的方式是在函数名称后面添加一对圆括号，即 `someFunction()`。通过前面的例子我们知道，可以使用实际的值来取代变量 `count`，那在这里是不是也可以用函数的值直接取代函数名呢？然而，下面的代码却会导致错误。

```
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(){
    //这里是块级作用域
})();
```

无论在什么地方，只要临时需要一些变量，就可以使用私有作用域，例如：

```
function outputNumbers(count){
    (function () {
        for (var i=0; i < count; i++){
            alert(i);
        }
    })();

    alert(i);    //导致一个错误!
}
```

BlockScopeExample03.htm

在这个重写后的 outputNumbers() 函数中，我们在 for 循环外部插入了一个私有作用域。在匿名函数中定义的任何变量，都会在执行结束时被销毁。因此，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，它能够访问包含作用域中的所有变量。

这种技术经常在全局作用域中被用在函数外部，从而限制向全局作用域中添加过多的变量和函数。一般来说，我们都应该尽量少向全局作用域中添加变量和函数。在一个由很多开发人员共同参与的大型应用程序中，过多的全局变量和函数很容易导致命名冲突。而通过创建私有作用域，每个开发人员既可以使用自己的变量，又不必担心搞乱全局作用域。例如：

```
(function(){
    var now = new Date();
    if (now.getMonth() == 0 && now.getDate() == 1){
        alert("Happy new year!");
    }
})();
```

把上面这段代码放在全局作用域中，可以用来确定哪一天是1月1日；如果到了这一天，就会向用户显示一条祝贺新年的消息。其中的变量 now 现在是匿名函数中的局部变量，而我们不必在全局作用域中创建它。



这种做法可以减少闭包占用的内存问题，因为没有指向匿名函数的引用。只要函数执行完毕，就可以立即销毁其作用域链了。

7.4 私有变量

严格来讲，JavaScript 中没有私有成员的概念；所有对象属性都是公有的。不过，倒是有一个私有变量的概念。任何在函数中定义的变量，都可以认为是私有变量，因为不能在函数的外部访问这些变量。私有变量包括函数的参数、局部变量和在函数内部定义的其他函数。来看下面的例子：

```
function add(num1, num2){
    var sum = num1 + num2;
    return sum;
}
```

在这个函数内部，有 3 个私有变量：num1、num2 和 sum。在函数内部可以访问这几个变量，但在函数外部则不能访问它们。如果在这个函数内部创建一个闭包，那么闭包通过自己的作用域链也可以访问这些变量。而利用这一点，就可以创建用于访问私有变量的公有方法。

我们把有权访问私有变量和私有函数的公有方法称为特权方法（privileged method）。有两种在对象上创建特权方法的方式。第一种是在构造函数中定义特权方法，基本模式如下。

```
function MyObject(){


    //私有变量和私有函数
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }

    //特权方法
    this.publicMethod = function (){
        privateVariable++;
        return privateFunction();
    };
}
```

这个模式在构造函数内部定义了所有私有变量和函数。然后，又继续创建了能够访问这些私有成员的特权方法。能够在构造函数中定义特权方法，是因为特权方法作为闭包有权访问在构造函数中定义的所有变量和函数。对这个例子而言，变量 privateVariable 和函数 privateFunction() 只能通过特权方法 publicMethod() 来访问。在创建 MyObject 的实例后，除了使用 publicMethod() 这一个途径外，没有任何办法可以直接访问 privateVariable 和 privateFunction()。

利用私有和特权成员，可以隐藏那些不应该被直接修改的数据，例如：



```
function Person(name){

    this.getName = function(){
        return name;
    };

    this.setName = function (value) {
        name = value;
    };
}

var person = new Person("Nicholas");
alert(person.getName());    //"Nicholas"
person.setName("Greg");
alert(person.getName());    //"Greg"
```

PrivilegedMethodExample01.htm

以上代码的构造函数中定义了两个特权方法：getName() 和 setName()。这两个方法都可以在构造函数外部使用，而且都有权访问私有变量 name。但在 Person 构造函数外部，没有任何办法访问 name。由于这两个方法是在构造函数内部定义的，它们作为闭包能够通过作用域链访问 name。私有变量 name 在 Person 的每一个实例中都不相同，因为每次调用构造函数都会重新创建这两个方法。不过，在构造

函数中定义特权方法也有一个缺点，那就是你必须使用构造函数模式来达到这个目的。第 6 章曾经讨论过，构造函数模式的缺点是针对每个实例都会创建同样一组新方法，而使用静态私有变量来实现特权方法就可以避免这个问题。

7.4.1 静态私有变量

通过在私有作用域中定义私有变量或函数，同样也可以创建特权方法，其基本模式如下所示。

```
(function(){  
  
    //私有变量和私有函数  
    var privateVariable = 10;  
  
    function privateFunction(){  
        return false;  
    }  
  
    //构造函数  
    MyObject = function(){  
    };  
  
    //公有/特权方法  
    MyObject.prototype.publicMethod = function(){  
        privateVariable++;  
        return privateFunction();  
    };  
  
})();
```

这个模式创建了一个私有作用域，并在其中封装了一个构造函数及相应的方法。在私有作用域中，首先定义了私有变量和私有函数，然后又定义了构造函数及其公有方法。公有方法是在原型上定义的，这一点体现了典型的原型模式。需要注意的是，这个模式在定义构造函数时并没有使用函数声明，而是使用了函数表达式。函数声明只能创建局部函数，但那并不是我们想要的。出于同样的原因，我们也没有在声明 `MyObject` 时使用 `var` 关键字。记住：初始化未经声明的变量，总是会创建一个全局变量。因此，`MyObject` 就成了一个全局变量，能够在私有作用域之外被访问到。但也要知道，在严格模式下给未经声明的变量赋值会导致错误。

这个模式与在构造函数中定义特权方法的主要区别，就在于私有变量和函数是由实例共享的。由于特权方法是在原型上定义的，因此所有实例都使用同一个函数。而这个特权方法，作为一个闭包，总是保存着对包含作用域的引用。来看一看下面的代码。

```
(function(){  
  
    var name = "";  
  
    Person = function(value){  
        name = value;  
    };  
  
    Person.prototype.getName = function(){  
        return name;  
    };  
  
    Person.prototype.setName = function (value){
```

```

        name = value;
    };
})();

var person1 = new Person("Nicholas");
alert(person1.getName()); // "Nicholas"
person1.setName("Greg");
alert(person1.getName()); // "Greg"

var person2 = new Person("Michael");
alert(person1.getName()); // "Michael"
alert(person2.getName()); // "Michael"

```

PrivilegedMethodExample02.htm

这个例子中的 `Person` 构造函数与 `getName()` 和 `setName()` 方法一样，都有权访问私有变量 `name`。在这种模式下，变量 `name` 就变成了一个静态的、由所有实例共享的属性。也就是说，在一个实例上调用 `setName()` 会影响所有实例。而调用 `setName()` 或新建一个 `Person` 实例都会赋予 `name` 属性一个新值。结果就是所有实例都会返回相同的值。

以这种方式创建静态私有变量会因为使用原型而增进代码复用，但每个实例都没有自己的私有变量。到底是使用实例变量，还是静态私有变量，最终还是要视你的具体需求而定。



多查找作用域链中的一个层次，就会在一定程度上影响查找速度。而这正是使用闭包和私有变量的一个显明的不足之处。

7.4.2 模块模式

前面的模式是用于为自定义类型创建私有变量和特权方法的。而道格拉斯所说的模块模式（`module pattern`）则是为单例创建私有变量和特权方法。所谓单例（`singleton`），指的就是只有一个实例的对象。按照惯例，JavaScript 是以对象字面量的方式来创建单例对象的。

```

var singleton = {
    name : value,
    method : function () {
        //这里是方法的代码
    }
};

```

模块模式通过为单例添加私有变量和特权方法能够使其得到增强，其语法形式如下：

```

var singleton = function(){

    //私有变量和私有函数
    var privateVariable = 10;

    function privateFunction(){
        return false;
    }
}

```

//特权/公有方法和属性

return {

publicProperty: true,

publicMethod : function(){

privateVariable++;

return privateFunction();

}

};

}();

这个模块模式使用了一个返回对象的匿名函数。在这个匿名函数内部,首先定义了私有变量和函数。然后,将一个对象字面量作为函数的值返回。返回的对象字面量中只包含可以公开的属性和方法。由于这个对象是在匿名函数内部定义的,因此它的公有方法有权访问私有变量和函数。从本质上来讲,这个对象字面量定义的是单例的公共接口。这种模式在需要对单例进行某些初始化,同时又需要维护其私有变量时是非常有用的,例如:

var application = function(){

//私有变量和函数

var components = new Array();

//初始化

components.push(new BaseComponent());

//公共

return {

getComponentCount : function(){

return components.length;

},

registerComponent : function(component){

if (typeof component == "object"){

components.push(component);

}

}

};

}();

ModulePatternExample01.htm

在 Web 应用程序中,经常需要使用一个单例来管理应用程序级的信息。这个简单的例子创建了一个用于管理组件的 application 对象。在创建这个对象的过程中,首先声明了一个私有的 components 数组,并向数组中添加了一个 BaseComponent 的新实例(在这里不需要关心 BaseComponent 的代码,我们只是用它来展示初始化操作)。而返回对象的 getComponentCount() 和 registerComponent() 方法,都是有权访问数组 components 的特权方法。前者只是返回已注册的组件数目,后者用于注册新组件。

简言之,如果必须创建一个对象并以某些数据对其进行初始化,同时还要公开一些能够访问这些私有数据的方法,那么就可以使用模块模式。以这种模式创建的每个单例都是 Object 的实例,因为最终要通过一个对象字面量来表示它。事实上,这也没有什么;毕竟,单例通常都是作为全局对象存在的,我们不会将它传递给一个函数。因此,也就没有什么必要使用 instanceof 操作符来检查其对象类型了。

7.4.3 增强的模块模式

有人进一步改进了模块模式，即在返回对象之前加入对其增强的代码。这种增强的模块模式适合那些单例必须是某种类型的实例，同时还必须添加某些属性和（或）方法对其加以增强的情况。来看下面的例子。

```
var singleton = function(){  
  
    //私有变量和私有函数  
    var privateVariable = 10;  
  
    function privateFunction(){  
        return false;  
    }  
  
    //创建对象  
    var object = new CustomType();  
  
    //添加特权/公有属性和方法  
    object.publicProperty = true;  
  
    object.publicMethod = function(){  
        privateVariable++;  
        return privateFunction();  
    };  
  
    //返回这个对象  
    return object;  
}();
```

如果前面演示模块模式的例子中的 application 对象必须是 BaseComponent 的实例，那么就可以使用以下代码。



```
var application = function(){  
  
    //私有变量和函数  
    var components = new Array();  
  
    //初始化  
    components.push(new BaseComponent());  
  
    //创建 application 的一个局部副本  
    var app = new BaseComponent();  
  
    //公共接口  
    app.getComponentCount = function(){  
        return components.length;  
    };  
  
    app.registerComponent = function(component){  
        if (typeof component == "object"){  
            components.push(component);  
        }  
    };  
  
    //返回这个副本
```

```

    return app;
  }());

```

ModuleAugmentationPatternExample01.htm

在这个重写后的应用程序（application）单例中，首先也是像前面例子中一样定义了私有变量。主要的不同之处在于命名变量 `app` 的创建过程，因为它必须是 `BaseComponent` 的实例。这个实例实际上是 `application` 对象的局部变量版。此后，我们又为 `app` 对象添加了能够访问私有变量的公有方法。最后一步是返回 `app` 对象，结果仍然是将它赋值给全局变量 `application`。

7.5 小结

在 JavaScript 编程中，函数表达式是一种非常有用的技术。使用函数表达式可以无须对函数命名，从而实现动态编程。匿名函数，也称为拉姆达函数，是一种使用 JavaScript 函数的强大方式。以下总结了函数表达式的特点。

- 函数表达式不同于函数声明。函数声明要求有名字，但函数表达式不需要。没有名字的函数表达式也叫做匿名函数。
- 在无法确定如何引用函数的情况下，递归函数就会变得比较复杂；
- 递归函数应该始终使用 `arguments.callee` 来递归地调用自身，不要使用函数名——函数名可能会发生变化。

当在函数内部定义其他函数时，就创建了闭包。闭包有权访问包含函数内部的所有变量，原理如下。

- 在后台执行环境中，闭包的作用域链包含着它自己的作用域、包含函数的作用域和全局作用域。
- 通常，函数的作用域及其所有变量都会在函数执行结束后被销毁。
- 但是，当函数返回了一个闭包时，这个函数的作用域将会一直在内存中保存到闭包不存在为止。

使用闭包可以在 JavaScript 中模仿块级作用域（JavaScript 本身没有块级作用域的概念），要点如下。

- 创建并立即调用一个函数，这样既可以执行其中的代码，又不会在内存中留下对该函数的引用。
- 结果就是函数内部的所有变量都会被立即销毁——除非将某些变量赋值给了包含作用域（即外部作用域）中的变量。

闭包还可以用于在对象中创建私有变量，相关概念和要点如下。

- 即使 JavaScript 中没有正式的私有对象属性的概念，但可以使用闭包来实现公有方法，而通过公有方法可以访问在包含作用域中定义的变量。
- 有权访问私有变量的公有方法叫做特权方法。
- 可以使用构造函数模式、原型模式来实现自定义类型的特权方法，也可以使用模块模式、增强的模块模式来实现单例的特权方法。

JavaScript 中的函数表达式和闭包都是极其有用的特性，利用它们可以实现很多功能。不过，因为创建闭包必须维护额外的作用域，所以过度使用它们可能会占用大量内存。