第5章

语句

第4章提到，表达式在JavaScript中是短语，那么语句（statement）就是JavaScript整句或命令。正如英文是用句号作结尾来分隔语句，JavaScript语句是以分号结束（见2.5节）。表达式计算出一个值，但语句用来执行以使某件事发生。

“使某件事发生”的一个方法是计算带有副作用的表达式。诸如赋值和函数调用这些有副作用的表达式，是可以作为单独的语句的，这种把表达式当做语句的用法也称做表达式语句（expression statement）。类似的语句还有声明语句（declaration statement），声明语句用来声明新变量或定义新函数。

JavaScript程序无非就是一系列可执行语句的集合。默认情况下，JavaScript解释器依照语句的编写顺序依次执行。另一种“使某件事发生”的方法是改变语句的默认执行顺序。JavaScript中有很多语句和控制结构（control structure）来改变语句的默认执行顺序：

- 条件（conditional）语句，JavaScript解释器可以根据一个表达式的值来判断是执行还是跳过这些语句，如`if`语句和`switch`语句。
- 循环（loop）语句，可以重复执行语句，如`while`和`for`语句。
- 跳转（jump）语句，可以让解释器跳转至程序的其他部分继续执行，如`break`、`return`和`throw`语句。

接下来的几节将介绍JavaScript中各式各样的语句及其语法。本章最后的表5-1对这些语句作了总结。一个JavaScript程序无非是一个以分号分隔的语句集合，所以一旦掌握了JavaScript语句，就可以开始编写JavaScript程序了。

5.1 表达式语句

具有副作用的表达式是JavaScript中最简单的语句（5.7.3节介绍了一种重要的无副作用的表达式语句）。这类语句已经在第4章讲述了。赋值语句是一类比较重要的表达式语句，例如：

```
greeting = "Hello " + name;
i *= 3;
```

递增运算符（++）和递减运算符（--）和赋值语句有关。它们的作用是改变一个变量的值，就像执行一条赋值语句一样：

```
counter++;
```

delete运算符的重要作用是删除一个对象的属性，所以，它一般作为语句使用，而不是作为复杂表达式的一部分：

```
delete o.x;
```

函数调用是表达式语句的另一个大类，例如：

```
alert(greeting);
window.close();
```

虽然这些客户端函数调用都是表达式，但它们都对Web浏览器造成了一些影响，所以我们认为它们也是语句。调用一个没有任何副作用的函数是没有意义的，除非它是复杂表达式或赋值语句的一部分，例如，不可能计算了一个余弦值随即把它丢弃：

```
Math.cos(x);
```

相反，得出了余弦值就得把它赋值给一个变量，以便将来才能使用这个值：

```
cx = Math.cos(x);
```

再次提醒读者，这些示例中的每行代码都是以分号结束的。

5.2 复合语句和空语句

可以用逗号运算符（见4.13.5节）将几个表达式连接在一起，形成一个表达式，同样，JavaScript中还可以将多条语句联合在一起，形成一条复合语句（compound statement）。只须用花括号将多条语句括起来即可。因此，下面几行代码就可以当成一条单独的语句，使用在JavaScript中任何希望使用一条语句的地方：

```
{
    x = Math.PI;
    cx = Math.cos(x);
    console.log("cos(π) = " + cx);
}
```

关于语句块有几点需要注意，第一，语句块的结尾不需要分号。块中的原始语句必须以分号结束，但语句块不需要。第二，语句块中的行都有缩进，这不是必需的，但整齐的缩进能让代码可读性更强，更容易理解。最后，需要注意，JavaScript中没有块级作用域，在语句块中声明的变量并不是语句块私有的（参照3.10.1节）。

将多条语句合并成一个大语句块的做法在JavaScript编程中非常常见。类似表达式通常包含子表达式一样，很多JavaScript语句包含其他子语句。从形式上讲，JavaScript语法通常允许一个语句块只包含一条子语句。例如，while循环的循环体就可以只包含一条语句。使用语句块，可以将任意数量的语句放到这个块中，这个语句块可以作为一条语句使用。

在JavaScript中，当希望多条语句被当做一条语句使用时，使用复合语句来替代。空语句（empty statement）则恰好相反，它允许包含0条语句的语句。空语句如下所示：

```
;
```

JavaScript解释器执行空语句时它显然不会执行任何动作。但实践证明，当创建一个具有空循环体的循环时，空语句有时是很有用的。例如下面的for循环（for循环详见5.5.3节）：

```
//初始化一个数组a
for(i = 0; i < a.length; a[i++] = 0) ;
```

在这个循环中，所有的操作都在表达式a[i++]=0中完成，这里并不需要任何循环体。然而JavaScript需要循环体中至少包含一条语句，因此，这里只使用了一个单独的分号来表示一条空语句。

注意，在for循环、while循环或if语句的右圆括号后的分号很不起眼，这很可能造成一些致命bug，而这些bug很难定位到。例如，下面的代码的执行结果可能就不是程序作者想要的效果：

```
if ((a == 0) || (b == 0));    //糟糕！这一行代码什么都没做...
o = null;                     //这一行代码总是会执行
```

如果有特殊的目的需要使用空语句，最好在代码中添加注释，这样可以更清楚地说明这条空语句是有用的，例如：

```
for(i = 0; i < a.length; a[i++] = 0) /* empty */ ;
```

5.3 声明语句

var和function都是声明语句，它们声明或定义变量或函数。这些语句定义标识符（变量名和函数名）并给其赋值，这些标识符可以在程序中任意地方使用。声明语句本身什么也不做，但它有一个重要的意义，通过创建变量和函数，可以更好地组织代码的语义。

接下来的几节将会讲述var语句和functuion语句，但并不包含变量和函数的全部内容，更多关于变量的内容请参照3.9节和3.10节。更多关于函数的内容请参照第8章。

5.3.1 var

var语句用来声明一个或者多个变量，它的语法如下：

```
var name_1 [ = value_1] [ ,..., name_n [= value_n]]
```

关键字var之后跟随的是要声明的变量列表，列表中的每一个变量都可以带有初始化表达式，用于指定它的初始值，例如：

```
var i;                                          //一个简单的变量
var j = 0;                                      //一个带有初始值的变量
var p, q;                                       //两个变量
var greeting = "hello" + name;                  //更复杂的初始化表达式
var x = 2.34, y = Math.cos(0.75), r, theta;     //很多变量
var x = 2, y = x*x;                             //第二个变量使用了第一个变量
var x = 2,                                      //更多变量
    f = function(x) { return x*x },             //每一个变量都独占一行
    y = f(x);
```

如果var语句出现在函数体内，那么它定义的是一个局部变量，其作用域就是这个函数。如果在顶层代码中使用var语句，它声明的是全局变量，在整个JavaScript程序中都是可见的。正如在3.10.2节提到的，全局变量是全局对象的属性。然而和其他全局对象属性不同的是，var声明的变量是无法通过delete删除的。

如果var语句中的变量没有指定初始化表达式，那么这个变量的值初始为undefined。3.10.1节已经提到，变量在声明它们的脚本或函数中都是有定义的，变量声明语句会被“提前”至脚本或者函数的顶部。但是初始化的操作则还在原来var语句的位置执行，在声明语句之前变量的值是undefined。

需要注意的是，var语句同样可以作为for循环或者for/in循环的组成部分（和在循环之外声明的变量声明一样，这里声明的变量也会“提前”）。这里重复一下3.9节的例子：

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var i in o) console.log(i);
```

注意，多次声明同一个变量是无所谓的。

5.3.2 function

关键字function用来定义函数。在4.3节中我们已经见过函数定义表达式。函数定义也可以写成语句的形式。例如，下面示例代码中的两种定义写法：

```
var f = function(x) { return x+1; }    //将表达式赋值给一个变量
function f(x) { return x+1; }          //含有变量名的语句
```

函数声明语句的语法如下：

```
function funcname([arg1 [, arg2 [..., argn]]]) {
    statements
}
```

funcname是要声明的函数的名称的标识符。函数名之后的圆括号中是参数列表，参数之间使用逗号分隔。当调用函数时，这些标识符则指代传入函数的实参。

函数体是由JavaScript语句组成的，语句的数量不限，且用花括号括起来。在定义函数时，并不执行函数体内的语句，它和调用函数时待执行的新函数对象相关联。注意，function语句里的花括号是必需的，这和while循环和其他一些语句所使用的语句块是不同的，即使函数体只包含一条语句，仍然必须使用花括号将其括起来。

下面是一些函数声明的例子：

```
function hypotenuse(x, y) {
    return Math.sqrt(x*x + y*y);       //下一节会讲到return
}

function factorial(n) {                //一个递归函数
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

函数声明语句通常出现在JavaScript代码的最顶层，也可以嵌套在其他函数体内。但在嵌套时，函数声明只能出现在所嵌套函数的顶部。也就是说，函数定义不能出现在if语句、while循环或其他任何语句中，正是由于函数声明位置的这种限制，ECMAScript标准规范并没有将函数声明归类为真正的语句。有一些JavaScript实现的确允许在出现语句的地方都可以进行函数声明，但是不同的实现在细节处理方式上有很大差别，因此将函数声明放在其他的语句内的做法并不具备可移植性。

尽管函数声明语句和函数定义表达式包含相同的函数名，但二者仍然不同。两种方式都创建了新的函数对象，但函数声明语句中的函数名是一个变量名，变量指向函数对象。

和通过var声明变量一样，函数定义语句中的函数被显式地“提前”到了脚本或函数的顶部。因此它们在整个脚本和函数内都是可见的。使用var的话，只有变量声明提前了——变量的初始化代码仍然在原来的位置。然而使用函数声明语句的话，函数名称和函数体均提前：脚本中的所有函数和函数中所有嵌套的函数都会在当前上下文中其他代码之前声明。也就是说，可以在声明一个JavaScript函数之前调用它。

和var语句一样，函数声明语句创建的变量也是无法删除的。但是这些变量不是只读的，变量值可以重写。

5.4 条件语句

条件语句是通过判断指定表达式的值来决定执行还是跳过某些语句。这些语句是代码的“决策点”，有时称为“分支”。如果说JavaScript解释器是按照代码的“路径”执行的，条件语句就是这条路径上的分叉点，程序执行到这里时必须选择其中一条路径继续执行。

下面几节将会讲述JavaScript中基本的条件语句，如if/else语句和switch语句，switch语句是一种更复杂的多分支条件语句。

5.4.1 if

if语句是一种基本的控制语句，它让JavaScript程序可以选择执行路径，更准确地说，就是有条件地执行语句，这种语句有两种形式，第一种是：

```
if (expression)
    statement
```

在这种形式中，需要计算*expression*的值，如果计算结果是真值，那么就执行*statement*。如果*expression*的值是假值，那么就不执行*statement*。例如：

```
if (username == null)          //如果username是null或者undefined
username = "John Doe";         // 对其进行定义
```

同样地：

```
//如果username是null、undefined、false、0、""或者NaN，那么给它赋一个新值
if (!username) username = "John Doe";
```

需要注意的是，if语句中括住*expression*的圆括号在语法上是必需的。

JavaScript语法规定，if关键字和带圆括号的表达式之后必须跟随一条语句，但可以使用语句块将多条语句合并成一条。因此，if语句的形式如下所示：

```
if (!address) {
    address = "";
    message = "Please specify a mailing address.";
}
```

if语句的第二种形式引入了else从句，当*expression*的值是false的时候执行else中的逻辑。其语法如下：

```
if (expression)
        statement1
else
        statement2
```

在这段代码中，当*expression*为真值时执行statement1，当*expression*为假值时执行statement2，例如：

```
if (n == 1)
    console.log("You have 1 new message.");
else
    console.log("You have " + n + " new messages.");
```

当在if/else语句中嵌套使用if语句时，必须注意确保else语句匹配正确的if语句。考虑如下代码：

```
i = j = 1;
k = 2;
if (i == j)
    if (j == k)
        console.log("i equals k");
else
    console.log("i doesn't equal j"); // 错误!!
```

在这个示例中，内层if语句构成了外层if语句所需要的子句。但是，if和else的匹配关系并不清晰（只有缩进给出了一些暗示），而且在这个例子中，缩进给出的暗示是错误的，因为JavaScript解释器将上述代码实际解释为：

```
if (i == j) {
    if (j == k)
        console.log("i equals k");
    else
        console.log("i doesn't equal j"); //错误!
}
```

和大多数编程语言一样，JavaScript中的if、else匹配规则是，else总是和就近的if语句匹配。为了让这个例子可读性更强、更易理解、更方便维护和调试，应当适当地使用花括号：

```
if (i == j) {
    if (j == k) {
        console.log("i equals k");
    }
}
else {  //花括号让代码结构更加清晰
    console.log("i doesn't equal j");
}
```

虽然这并不是本书中所使用的编码风格，但许多程序员都有将if和else语句主体用花括号括起来的习惯（就像在类似while循环这样的复合语句中一样），即便每条分支只有一条语句，但坚持这样做可以避免刚才这种程序歧义的问题。

5.4.2 else if

if/else语句通过判断一个表达式的计算结果来选择执行两条分支中的一条。但当代码中有多条分支的时候该怎么办呢？一种解决办法是使用else if语句。else if语句并不是真正的JavaScript语句，它只不过是多条if/else语句连在一起时的一种惯用写法。

```
if (n == 1) {
    // 执行代码块 1
}
else if (n == 2) {
    // 执行代码块 2
}
else if (n == 3) {
    // 执行代码块 3
}
else {
    // 之前的条件都为false，则执行这里的代码块 4
}
```

这种代码并没有什么特别之处，它由多条if语句组成，每条if语句的else从句又包含另外一条if语句。可以用if语句的嵌套形式来完成在语法上等价的代码，但与之相比，显然使用else if写法更清晰也更可取：

```
if (n == 1) {
    // 执行代码块 1
}
else {
    if (n == 2) {
     // 执行代码块 2
    }
    else {
     if (n == 3) {
         // 执行代码块 3
     }
     else {
```

```
        // 如果所有的判断都是false，执行代码块 4
    }
  }
}
```

5.4.3 switch

if语句在程序执行过程中创建一条分支，并且可以使用else if来处理多条分支。然而，当所有的分支都依赖于同一个表达式的值时，else if并不是最佳解决方案。在这种情况下，重复计算多条if语句中的条件表达式是非常浪费的做法。

switch语句正适合处理这种情况。关键字switch之后紧跟着圆括号括起来的一个表达式，随后是一对花括号括起来的代码块：

```
switch(expression) {
     statements
}
```

然而，switch语句的完整语法要比这复杂一些。代码块中可以使用多个由case关键字标识的代码片段，case之后是一个表达式和一个冒号，case和标记语句很类似，只是这个标记语句并没有名字，它只和它后面的表达式关联在一起。当执行这条switch语句的时候，它首先计算*expression*的值，然后查找case子句中的表达式是否和*expression*的值相同（这里的“相同”是按照“===”运算符进行比较的）。如果找到匹配的case，那么将会执行这个case对应的代码块。如果找不到匹配的case，那么将会执行“default:”标签中的代码块。如果没有“default:”标签，switch语句将跳过它的所有代码块。

switch语句是非常容易引起混淆的。用例子来解释会比较清晰一些，下面的switch语句和方才展示的if/else语句是等价的：

```
switch(n) {
    case 1:                  // 如果 n === 1 ，从这里开始执行
     // 执行代码块 1
    break;                   // 停止执行switch语句
    case 2:                  // 如果 n === 2，从这里执行
     // 执行代码块 2
     break;                  // 在这里停止执行switch语句
    case 3:                  // 如果 n === 3，从这里执行
     // 执行代码块 3
     break;                  // 在这里停止执行switch语句
    default:                 // 如果所有的条件都不匹配
     // 执行代码块 4
     break;                  // 在这里停止执行switch语句
}
```

需要注意的是，在上面的代码中，在每一个case语句块的结尾处都使用了关键字break。

我们将在后面介绍break语句，break语句可以使解释器跳出switch语句或循环语句。在switch语句中，case只是指明了要执行的代码起点，但并没有指明终点。如果没有break语句，那么switch语句就会从与*expression*的值相匹配的case标签处的代码块开始执行，依次执行后续的语句，一直到整个switch代码块的结尾。这种由一个case标签执行到下一个case标签的代码逻辑是很少使用的，在大多数情况下，应该使用break语句来终止每个case语句块。当然，如果在函数中使用switch语句，可以使用return来代替break，return和break都用于终止switch语句，也会防止一个case语句块执行完后继续执行下一个case语句块。

下面的switch语句的例子更加贴近实战，它根据值的类型将该值转换为字符串：

```
function convert(x) {
    switch(typeof x) {
     case 'number':            // 将数字转换为十六进制数
         return x.toString(16);
     case 'string':            // 返回两端带双引号的字符串
         return '"' + x + '"';
     default:                  // 使用普通的方法转换其他类型
         return String(x);
    }
}
```

注意，在上面两个例子中，case关键字后跟随的是数字和字符串直接量，在实际中这是switch语句最常见的用法，但是ECMAScript标准允许每个case关键字跟随任意的表达式。

switch语句首先计算switch关键字后的表达式，然后按照从上到下的顺序计算每个case后的表达式，直到执行到case的表达式的值与switch的表达式的值相等时为止[注1]。由于对每个case的匹配操作实际上是“===”恒等运算符比较，而不是“==”相等运算符比较，因此，表达式和case的匹配并不会做任何类型转换。

由于每次执行switch语句的时候，并不是所有的case表达式都能执行到，因此，应当避免使用带有副作用的case表达式，比如函数调用表达式和赋值表达式。最安全的做法就是在case表达式中使用常量表达式。

前面提到过，如果switch表达式与所有case表达式都不匹配，则执行标记为“default:”的语句块；如果没有“default:”标签，则switch的整个语句块都将跳

注1：由于JavaScript中的case表达式的值是在运行时（run-time）计算的，这一点使得JavaScript的switch语句和C、C++和Java中的switch语句有很大区别（并且效率也很低）。在C、C++和Java中，case表达式必须为同类型的编译时（compile-time）常量，而且switch语句通常会编译成一个跳转表（jump table），这让switch语句的执行非常高效。

过。我们注意到，在之前的例子中，“default:”标签都出现在switch的末尾，位于所有case标签之后。当然这是最合理也是最常用的写法，实际上，“default:”标签可以放置在switch语句内的任何地方。

5.5 循环

为了理解条件语句，可以将在JavaScript中的代码想象成一条条的分支路径。循环语句（looping statement）就是程序路径的一个回路，可以让一部分代码重复执行。JavaScript中有4种循环语句：while、do/while、for和for/in。下面几节将会依次讲解它们。其中最常用的循环就是对数组元素的遍历，7.6节详细讨论这种循环和使用数组类定义的特殊循环方法。

5.5.1 while

if语句是一种基本的控制语句，用来选择执行程序的分支语句。和if一样，while语句也是一个基本循环语句，它的语法如下：

```
while (expression)
       statement
```

在执行while语句之前，JavaScript解释器首先计算*expression*的值，如果它的值是假值，那么程序将跳过循环体中的逻辑*statement*转而执行程序中的下一条语句。反之，如果表达式*expression*是真值，JavaScript解释器将执行循环体内的逻辑，然后再次计算表达式*expression*的值，这种循环会一直继续下去，直到*expression*的值为假值为止。换一种说法就是当表达式*expression*是真值时则循环执行*statement*，注意，使用while(true)则会创建一个死循环。

通常来说，我们并不想让JavaScript反复执行同一操作。在几乎每一次循环中，都会有一个或多个变量随着循环的迭代而改变。正是由于改变了这些变量，因此每次循环执行的*statement*的操作也不尽相同。而且，如果改变的变量在*expression*中用到，那么每次循环表达式的值也不同。这一点非常重要，否则一个初始值为真值的表达式的值永远都是真值，循环也不会结束，下面这个示例所示的while循环输出0～9之间的值：

```
var count = 0;
while (count < 10) {
    console.log(count);
    count++;
}
```

可以发现，在这个例子中，变量count的初始值是0，在循环执行过程中，它的值每次都

递增1。当循环执行了10次，表达式的值就变成了false（即，变量count的值不再小于10），这时while就会结束，JavaScript解释器将执行程序中的下一条语句。大多数循环都会有一个像count这样的计数器变量。尽管循环计数器常用i、j、k这样的变量名，但如果想要让代码可读性更强，就应当使用更具语义的变量名。

5.5.2 do/while

do/while循环和while循环非常相似，只不过它是在循环的尾部而不是顶部检测循环表达式，这就意味着循环体至少会执行一次。do/while循环的语法如下：

```
do
    statement
while (expression);
```

do/while循环并不像while循环那么常用。这是因为在实践中那种想要循环至少一次的情况并不常见，下面是一个do/while循环的例子：

```
function printArray(a) {
    var len = a.length, i = 0;
    if (len == 0)
        console.log("Empty Array");
    else {
        do {
            console.log(a[i]);
        } while (++i < len);
    }
}
```

在do/while循环和普通的while循环之间有两点语法方面的不同之处。首先，do循环要求必须使用关键字do来标识循环的开始，用while来标识循环的结尾并进入循环条件判断；其次，和while循环不同，do循环是用分号结尾的。如果while的循环体使用花括号括起来的话，则while循环也不用使用分号做结尾。

5.5.3 for

for语句提供了一种比while语句更加方便的循环控制结构。for语句对常用的循环模式做了一些简化。大部分的循环都具有特定的计数器变量。在循环开始之前要初始化这个变量，然后在每次循环执行之前都检测一下它的值。最后，计数器变量做自增操作，否则就在循环结束后、下一次判断循环条件前做修改。在这一类循环中，计数器的三个关键操作是初始化、检测和更新。for语句就将这三步操作明确声明为循环语法的一部分，各自使用一个表达式来表示。for语句的语法如下：

```
for(initialize ; test ; increment)
```

```
    statement
```

initialize、test和increment三个表达式之间用分号分隔，它们分别负责初始化操作、循环条件判断和计数器变量的更新。将它们放在循环的第一行会更容易理解for循环正在做什么，而且也可以防止忘记初始化或者递增计数器变量。

要解释for循环是如何工作的，最简单的方法莫过于列出一个与之等价的while循环[注2]。

```
initialize;
  while(test) {
      statement
      increment;
  }
```

换句话说，initialize表达式只在循环开始之前执行一次。初始化表达式应当具有副作用（通常是一条赋值语句）。JavaScript同样允许初始化表达式中带有var变量声明语句，这样的话就可以同时声明并初始化一个计数变量。每次循环执行之前会执行test表达式，并判断表达式的结果来决定是否执行循环体，如果test计算结果为真值，则执行循环体中的statement。最后，执行increment表达式。同样，为了有用起见，这里的increment表达式也必须具有副作用。通常来讲，它不是一个赋值表达式就是一个由“++”或“--”运算符构成的表达式。

在上文中的while循环的例子可以使用for循环来重写，这个循环同样输出数字0～9：

```
for(var count = 0; count < 10; count++)
    console.log(count);
```

当然，有些循环会比这些例子更加复杂，而且循环中的一次迭代会改变多个变量。在JavaScript中，这种情况则必须用到逗号运算符，它将初始化表达式和自增表达式合并入一个表达式中以用于for循环：

```
var i,j;
for(i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

到目前为止，在示例代码中的循环变量都是数字。当然数字是最常用的，但不是必需的。下面这段代码就使用for循环来遍历链表数据结构，并返回链表中的最后一个对象（也就是第一个不包含next属性的对象）：

```
function tail(o) {                          // 返回链表的最后一个节点对象
    for(; o.next; o = o.next) /* empty */ ;  // 根据判断o.next是不是真值来执行遍历
    return o;
}
```

注2：5.6.3节会讲到，在使用continue语句时，while循环和for循环并不等价。

需要注意的是，这段代码不包含initialize表达式，for循环中那三个表达式中的任何一个都可以忽略，但是两个分号必不可少。如果省略test表达式，那么这将是一个死循环，同样，和while(true)类似，死循环的另外一种写法是for(;;)。

5.5.4 for/in

for/in语句也使用for关键字，但它是和常规的for循环完全不同的一类循环。for/in循环语句的语法如下：

```
for (variable in object)
     statement
```

variable通常是一个变量名，也可以是一个可以产生左值的表达式或者一个通过var语句声明的变量，总之必须是一个适用于赋值表达式左侧的值。object是一个表达式，这个表达式的计算结果是一个对象。同样，statement是一个语句或语句块，它构成了循环的主体。

使用for循环来遍历数组元素是非常简单的：

```
for(var i = 0; i < a.length; i++)      //i代表了数组元素的索引
    console.log(a[i]);                 //输出数组中的每个元素
```

而for/in循环则是用来更方便地遍历对象属性成员：

```
for(var p in o)                 // 将属性名字赋值给变量p
    console.log(o[p]);          //输出每一个属性的值
```

在执行for/in语句的过程中，JavaScript解释器首先计算*object*表达式。如果表达式为null或者undefined，JavaScirpt解释器将会跳过循环并执行后续的代码[注3]。如果表达式等于一个原始值，这个原始值将会转换为与之对应的包装对象（wrapper object）（见3.6节）。否则，expression本身已经是对象了。JavaScript会依次枚举对象的属性来执行循环。然而在每次循环之前，JavaScript都会先计算*variable*表达式的值，并将属性名（一个字符串）赋值给它 。

需要注意的是，只要for/in循环中*variable*的值可以当做赋值表达式的左值，它可以是任意表达式。每次循环都会计算这个表达式，也就是说每次循环它计算的值有可能不同。例如，可以使用下面这段代码将所有对象属性复制至一个数组中：

```
var o = {x:1, y:2, z:3};
var a = [], i = 0;
for(a[i++] in o) /* empty */;
```

注3：　在这种情况下，ECMAScript 3的实现可能会抛出一个类型错误异常。

JavaScript数组不过是一种特殊的对象，因此，for/in循环可以像枚举对象属性一样枚举数组索引。例如，在上面的代码之后加上这段代码就可以枚举数组的索引0、1、2：

```
for(i in a) console.log(i);
```

其实，for/in循环并不会遍历对象的所有属性，只有“可枚举”（enumerable）的属性才会遍历到（参照6.7节）。由JavaScript语言核心所定义的内置方法就不是“可枚举的”。比如，所有的对象都有方法toString()，但for/in循环并不枚举toString这个属性。除了内置方法之外，还有很多内置对象的属性也是“不可枚举的”（nonenumerable）。而代码中定义的所有属性和方法都是可枚举的（6.7节会讲到，但在ECMAScript 5中可以通过特殊手段让可枚举属性变为不可枚举）。对象可以继承其他对象的属性，那些继承的自定义属性（参照6.2.2节）也可以使用for/in枚举出来。

如果for/in的循环体删除了还未枚举的属性，那么这个属性将不会再枚举到。如果循环体定义了对象的新属性，这些属性通常也不会枚举到（然而，JavaScript的有些实现是可以枚举那些在循环体中增加的继承属性的）。

属性枚举的顺序

ECMAScript规范并没有指定for/in循环按照何种顺序来枚举对象属性。但实际上，主流浏览器厂商的JavaScript实现是按照属性定义的先后顺序来枚举简单对象的属性，先定义的属性先枚举。如果使用对象直接量的形式创建对象，则将按照直接量中属性的出现顺序枚举。有一些网站和JavaScript库是依赖于这种枚举顺序的，浏览器厂商不大可能会修改这个顺序。

上一段讨论了JavaScript解释器枚举“简单”对象一种交互的属性枚举顺序。在下列情况下，枚举的顺序取决于具体的实现（并且是非交互的）：

- 对象继承了可枚举属性；
- 对象具有整数数组索引的属性；
- 使用delete删除了对象已有的属性；
- 使用Object.defineProperty()（见6.7节）或者类似的方法改变了对象的属性。

除了所有非继承的“自有”属性以外的继承属性（参照6.2.2节）都往往（但并不是所有的JavaScript实现都是如此）都是可枚举的，而且可以按照它们定义的顺序进行枚举。如果对象属性继承自多个“原型”（prototype）（参照6.1.3节），也就是说它的原型链上有多个对象，那么链上面的每一个原型对象的属性的遍历也是依照特定顺序执行的。JavaScript的一些（但不是全部）实现依照数字顺序来枚举数组属性，而不是某种特定的

顺序。但当数组元素的索引是非数字或数组是稀疏数组（数组索引是不连续的）时它们则按照特定顺序枚举。

5.6 跳转

JavaScript中另一类语句是跳转语句（jump statement）。从名称就可以看出，它使得JavaScript的执行可以从一个位置跳转到另一个位置。break语句是跳转到循环或者其他语句的结束。continue语句是终止本次循环的执行并开始下一次循环的执行。JavaScript中的语句可以命名或带有标签，break和continue可以标识目标循环或者其他语句标签。

return语句让解释器跳出函数体的执行，并提供本次调用的返回值。throw语句触发或者“抛出”一个异常，它是与try/catch/finally语句一同使用的，这些语句指定了处理异常的代码逻辑。这是一种复杂的跳转语句，当抛出一个异常的时候，程序将跳转至最近的闭合异常处理程序，这个异常处理程序可以是在同一个函数中或者在更高层的调用栈中。

接下来会详细讲述每一种跳转语句。

5.6.1 标签语句

语句是可以添加标签的，标签是由语句前的标识符和冒号组成：

```
identifier: statement
```

通过给语句定义标签，就可以在程序的任何地方通过标签名引用这条语句。也可以对多条语句定义标签，尽管只有在给语句块定义标签时它才更有用，比如循环和条件判断语句。通过给循环定义一个标签名，可以在循环体内部使用break和continue来退出循环或者直接跳转到下一个循环的开始。break和continue是JavaScript中唯一可以使用语句标签的语句。本章接下来会有讲述。这里有一个例子，其中while循环定义了一个标签，continue语句使用了这个标签：

```
mainloop: while(token != null) {
    // 忽略这里的代码...
    continue mainloop; // 跳转到下一次循环
    // 忽略这里的代码...
}
```

这里用做标签的identifier必须是一个合法的JavaScript标识符，而不能是一个保留字。标签的命名空间和变量或函数的命名空间是不同的，因此可以使用同一个标识符作为语句标签和作为变量名或函数名。语句标签只有在它所起作用的语句（当然也可以在它的子句中）内是有定义的。一个语句标签不能和它内部的语句标签重名，但在两个代码段

不相互嵌套的情况下是可以出现同名的语句标签的。带有标签的语句还可以带有标签，也就是说，任何语句可以有很多个标签。

5.6.2 break 语句

单独使用break语句的作用是立即退出最内层的循环或switch语句。它的语法如下：

```
break;
```

由于它能够使循环和switch语句退出，因此这种形式的break只有出现在这类语句中才是合法的。

我们在switch语句的例子中已经见到过break语句。在循环中，不论出于什么原因，只要不想继续执行整个循环，就可以用break来提前退出。当循环终止条件非常复杂时，在函数体内使用break语句实现这些条件判断的做法要比直接在循环表达式中写出这个复杂终止条件的做法简单很多。下面的例子中的循环遍历整个数组元素来查找某个特定的值，当整个数组遍历完成后会正常退出循环，如果找到了需要查找的数组元素，则使用break语句退出循环：

```
for(var i = 0; i < a.length; i++) {
    if (a[i] == target) break;
}
```

JavaScript中同样允许break关键字后面跟随一个语句标签（只有标识符，没有冒号）：

```
break labelname;
```

当break和标签一块使用时，程序将跳转到这个标签所标识的语句块的结束，或者直接终止这个闭合语句块的执行。当没有任何闭合语句块指定了break所用的标签，这时会产生一个语法错误。当使用这种形式的break语句时，带标签的语句不应该是循环或者switch语句，因为break可以“跳出”任何闭合的语句块。这里的语句可以是由花括号括起来的一组语句，使用同一个标签来标识这一组语句。

在break关键字和*labelname*之间不能换行。因为JavaScript可以给语句自动补全省略掉的分号，如果break关键字和标签之间有换行，JavaScript解释器会认为你在使用break不带标签的最简形式，因此会在break后补充分号（参照2.5节）。

当你希望通过break来跳出非就近的循环体或者switch语句时，就会用到带标签的break语句。下面是示例代码：

```
var matrix = getData(); // 从某处得到一个二维数组
// 将矩阵中所有元素进行求和
var sum = 0, success = false;
```

```
//从标签名开始，以便在报错时退出程序
compute_sum: if (matrix) {
    for(var x = 0; x < matrix.length; x++) {
     var row = matrix[x];
     if (!row) break compute_sum;
        for(var y = 0; y < row.length; y++) {
            var cell = row[y];
            if (isNaN(cell)) break compute_sum;
            sum += cell;
        }
    }
    success = true;
}
//break语句跳转至此
//如果在success == false的条件下到达这里，说明我们给出的矩阵中有错误
//否则将矩阵中所有的元素进行求和
```

最后，需要注意的是，不管`break`语句带不带标签，它的控制权都无法越过函数的边界。比如，对于一条带标签的函数定义语句来说，不能从函数内部通过这个标签来跳转到函数外部。

5.6.3 continue 语句

`continue`语句和`break`语句非常类似，但它不是退出循环，而是转而执行下一次循环。continue语句的语法和break语句语法一样简单：

```
continue;
```

continue语句同样可以带有标签：

```
continue labelname;
```

不管`continue`语句带不带标签，它只能在循环体内使用。在其他地方使用将会报语法错误。

当执行到`continue`语句的时候，当前的循环逻辑就终止了，随即执行下一次循环，在不同类型的循环中，continue的行为也有所区别：

- 在`while`循环中，在循环开始处指定的*expression*会重复检测，如果检测结果为true，循环体会从头开始执行。
- 在do/while循环中，程序的执行直接跳到循环结尾处，这时会重新判断循环条件，之后才会继续下一次循环。
- 在`for`循环中，首先计算自增表达式，然后再次检测`test`表达式，用以判断是否执行循环体。

- 在for/in循环中，循环开始遍历下一个属性名，这个属性名赋给了指定的变量。

需要注意continue语句在while和for循环中的区别，while循环直接进入下一轮的循环条件判断，但for循环首先计算其*increment*表达式，然后判断循环条件。之前的章节讨论了和while循环“等价”的for循环的行为。但由于continue在这两种循环中的行为表现不同，因此使用while循环不可能完美地模拟等价的for循环。

下面这段代码展示了不带标签的continue语句，当产生一个错误的时候跳过当前循环的后续逻辑：

```
for(i = 0; i < data.length; i++) {
    if (!data[i]) continue; // 不能处理undefined数据
    total += data[i];
}
```

和break语句类似，带标签的continue语句可以用在嵌套的循环中，用以跳出多层次嵌套的循环体逻辑。同样和break语句类似，在continue语句和labelname之间不能有换行。

5.6.4 return 语句

回想一下，函数调用是一种表达式，而所有表达式都有值。函数中的return语句既是指定函数调用后的返回值。这里是return语句的语法：

```
return expression;
```

return语句只能在函数体内出现，如果不是的话会报语法错误。当执行到return语句的时候，函数终止执行，并返回*expression*的值给调用程序。例如：

```
function square(x) { return x*x; }     //一个包含return语句的函数
square(2)                              //调用结果为4
```

如果没有return语句，则函数调用仅依次执行函数体内的每一条语句直到函数结束，最后返回调用程序。这种情况下，调用表达式的结果是undefined。return语句经常作为函数内的最后一条语句出现，但并不是说要一定放在函数最后，即使在执行return语句的时候还有很多后续代码没有执行到，这时函数也还会返回调用程序。

return语句可以单独使用而不必带有*expression*，这样的话函数也会向调用程序返回undefined。例如：

```
function display_object(o) {
    // 如果参数是null或者undefined则立即返回
    if (!o) return;
    // 其他的逻辑
```

```
}
```

由于JavaScript可以自动插入分号（见2.5节），因此在return关键字和它后面的表达式之间不能有换行。

5.6.5 throw 语句

所谓异常（exception）是当发生了某种异常情况或错误时产生的一个信号。抛出异常，就是用信号通知发生了错误或异常状况。捕获异常是指处理这个信号，即采取必要的手段从异常中恢复。在JavaScript中，当产生运行时错误或者程序使用throw语句时就会显式地抛出异常。使用try/catch/finally语句可以捕获异常，下一节会对它作详细介绍。

throw语句的语法如下：

```
throw expression;
```

*expression*的值可以是任意类型的。可以抛出一个代表错误码的数字，或者包含可读的错误消息的字符串。当JavaScript解释器抛出异常的时候通常采用Error类型和其子类型，当然也可以使用它们。一个Error对象有一个name属性表示错误类型，一个message属性用来存放传递给构造函数的字符串（参照第三部分的Error类），在下面的例子中，当使用非法参数调用函数时就抛出一个Error对象：

```
function factorial(x) {
    // 如果输入参数是非法的，则抛出一个异常
    if (x < 0) throw new Error("x不能是负数");
    // 否则，计算出一个值，并正常地返回它
    for(var f = 1; x > 1; f *= x, x--) /* empty */ ;
    return f;
}
```

当抛出异常时，JavaScript解释器会立即停止当前正在执行的逻辑，并跳转至就近的异常处理程序。异常处理程序是用try/catch/finally语句的catch从句编写的，下一节会介绍它。如果抛出异常的代码块没有一条相关联的catch从句，解释器会检查更高层的闭合代码块，看它是否有相关联的异常处理程序。以此类推，直到找到一个异常处理程序为止。如果抛出异常的函数没有处理它的try/catch/finally语句，异常将向上传播到调用该函数的代码。这样的话，异常就会沿着JavaScript方法的词法结构和调用栈向上传播。如果没有找到任何异常处理程序，JavaScript将把异常当成程序错误来处理，并报告给用户。

5.6.6 try/catch/finally语句

try/catch/finally语句是JavaScript的异常处理机制。其中try从句定义了需要处理的异常所在的代码块。catch从句跟随在try从句之后，当try块内某处发生了异常时，调用

catch内的代码逻辑。catch从句后跟随finally块，后者中放置清理代码，不管try块中是否产生异常，finally块内的逻辑总是会执行。尽管catch和finally都是可选的，但try从句需要至少二者之一与之组成完整的语句。try、catch和finally语句块都需要使用花括号括起来，这里的花括号是必需的，即使从句中只有一条语句也不能省略花括号。

下面的代码说明了try/catch/finally的语法和使用目的：

```
try {
    //通常来讲，这里的代码会从头执行到尾而不会产生任何问题，
    //但有时会抛出一个异常，要么是由throw语句直接抛出异常，
    //要么是通过调用一个方法间接抛出异常
}
catch(e){
    //当且仅当try语句块抛出了异常，才会执行这里的代码
    //这里可以通过局部变量e来获得对Error对象或者抛出的其他值的引用
    //这里的代码块可以基于某种原因处理这个异常，也可以忽略这个异常，
    //还可以通过throw语句重新抛出异常
}
finally {
    //不管try语句块是否抛出了异常，这里的逻辑总是会执行，终止try语句块的方式有：
    //  1）正常终止，执行完语句块的最后一条语句
    //  2）通过break、continue或return语句终止
    //  3）抛出一个异常，异常被catch从句捕获
    //  4）抛出一个异常，异常未被捕获，继续向上传播
}
```

我们注意到，关键字catch后跟随了一对圆括号，圆括号内是一个标识符。这个标识符和函数参数很像。当捕获一个异常时，把和这个异常相关的值（比如Error对象）赋值给这个参数。和普通的变量不同，这条catch子句中的标识符具有块级作用域，它只在catch语句块内有定义。

这里有一个关于try/catch语句更实际的例子，这里使用了前面章节中提到的factorial()方法，并使用客户端JavaScript方法prompt()和alert()来输入和输出：

```
try {
    // 要求用户输入一个数字
    var n = Number(prompt("请输入一个正整数", ""));
    // 假设输入是合法的，计算这个数的阶乘
    var f = factorial(n);
    // 显示结果
    alert(n + "! = " + f);
}
catch (ex) {
    //如果输入不合法，将执行这里的逻辑
    alert(ex); // 告诉用户产生了什么错误
}
```

这里的try/catch语句并不包含finally从句。尽管finally不像catch那样经常使用，但有时候它还是非常有用。然而，我们需要更详尽地解释它的行为。不管try语句块中的代

码执行完成了多少，只要try语句中有一部分代码执行了，finally从句就会执行。它通常在try从句的代码后用于清理工作。

通常状况下，解释器执行到try块的尾部，然后开始执行finally中的逻辑，以便进行必要的清理工作。当由于return、continue或break语句使得解释器跳出try语句块时，解释器在执行新的目标代码之前先执行finally块中的逻辑。

如果在try中产生了异常，而且存在一条与之相关的catch从句来处理这个异常，解释器会首先执行catch中的逻辑，然后执行finally中的逻辑。如果不存在处理异常的局部catch从句，解释器会首先执行finally中的逻辑，然后向上传播这个异常，直到找到能处理这个异常的catch从句。

如果finally块使用了return、continue、break或者throw语句使程序发生跳转，或者通过调用了抛出异常的方法改变了程序执行流程，不管这个跳转使程序挂起还是继续执行，解释器都会将其忽略。例如，如果finally从句抛出一个异常，这个异常将替代正在抛出的异常。如果finally从句运行到了return语句，尽管已经抛出了异常且这个抛出的异常还没有处理，这个方法依然会正常返回[译注1]。

在没有catch从句的情况下try从句可以和finally从句一起使用。在这种情况下，finally块只包含清理代码，不管try块中是否有break、continue或return语句，这里的代码一定会执行，回想一下，我们无法完全精确地使用while循环来模拟for循环，因为continue语句在两个循环中的行为表现不一致。如果使用try/finally语句，就能使用while循环来正确模拟包含continue的for循环：

```
//模拟for(initialize;test;increment)body;
initialize ;
while( test ) {
    try { body ; }
    finally { increment ; }
}
```

然而需要注意的是，当body包含break语句时，while循环和for循环便有了更微妙的区别

译注1：按照原文的描述，这段代码将正常执行，并foo()函数会有正常的返回值，参照这段代码：

```
var foo = function(){
    try{
    //抛出一个异常
    }
    finally {
        return 1;//未处理异常直接返回，这里将正常返回
    }
    };
foo();
```

（造成了一次额外的自增运算），因此即便使用了finally从句，使用while来完全模拟for循环依然是不可能的。

5.7 其他语句类型

本节讨论剩余的三种JavaScript语句——width、debugger和use strict。

5.7.1 with语句

3.10.3节讨论了作用域链（scope chain），一个可以按序检索的对象列表，通过它可以进行变量名解析。with语句用于临时扩展作用域链，它具有如下的语法：

```
with (object)
statement
```

这条语句将*object*添加到作用域链的头部，然后执行*statement*，最后把作用域链恢复到原始状态。

在严格模式中（参照 5.7.3节）是禁止使用with语句的，并且在非严格模式里也是不推荐使用with语句的，尽可能避免使用with语句。那些使用with语句的JavaScript代码非常难于优化，并且同没有使用with语句的代码相比，它运行得更慢。

在对象嵌套层次很深的时候通常会使用with语句来简化代码编写。例如，在客户端JavaScript中，可能会使用类似下面这种表达式来访问一个HTML表单中的元素：

```
document.forms[0].address.value
```

如果这种表达式在代码中多次出现，则可以使用with语句将form对象添加至作用域链的顶层：

```
with(document.forms[0]) {
    // 直接访问表单元素，例如:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

这种方法减少了大量的输入，不用再为每个属性名添加document.forms[0]前缀。这个对象临时挂载在作用域链上，当JavaScript需要解析诸如address的标识符时，就会自动在这个对象中查找。当然，不使用with语句的等价代码可以写成这样：

```
var f = document.forms[0];
f.name.value = "";
```

```
f.address.value = "";
f.email.value = "";
```

不要忘记，只有在查找标识符的时候才会用到作用域链，创建新的变量的时候不使用它，看一下下面这行代码：

```
with(o) x = 1;
```

如果对象o有一个属性x，那么这行代码给这个属性赋值为1。但如果o中没有定义属性x，这段代码和不使用with语句的代码 x=1 是一模一样的。它给一个局部变量或者全局变量x赋值，或者创建全局对象的一个新属性。with语句提供了一种读取o的属性的快捷方式，但它并不能创建o的属性。

5.7.2 debugger语句

debugger语句通常什么也不做。然而，当调试程序可用并运行的时候，JavaScript解释器将会（非必需）以调式模式运行。实际上，这条语句用来产生一个断点（breakpoint），JavaScript代码的执行会停止在断点的位置，这时可以使用调试器输出变量的值、检查调用栈等。例如，假设由于调用函数f()的时候使用了未定义的参数，因此f()抛出一个异常，但无法定位到底是哪里抛出了异常。为了有助于调试这个问题，需要修改函数f()：

```
function f(o) {
if (o === undefined) debugger; //这一行代码只是用于临时调试
...                            //函数的其他部分
}
```

这时，当调用f()的时候没有传入参数，程序将停止执行，这时可以通过调试器检测调用栈并找出错误产生的原因。

在ECMAScript 5中，debugger语句正式加入到这门语言里。但在相当长的一段时间里，主流浏览器厂商已经将其实现了。注意，可用的调试器是远远不够的，debugger语句不会启动调试器。但如果调试器已经在运行中，这条语句才会真正产生一个断点。例如，如果使用Firefox的调试扩展插件Firebug，则必须首先为待调试的网页启用Friebug，这样debugger语句才能正常工作。

5.7.3 “use strict”

“use strict”是ECMAScript 5引入的一条指令。指令不是语句（但非常接近于语句）。“use strict”指令和普通的语句之间有两个重要的区别：

- 它不包含任何语言的关键字，指令仅仅是一个包含一个特殊字符串直接量的表达

式（可以是使用单引号也可以使用双引号），对于那些没有实现ECMAScript 5的JavaScript解释器来说，它只是一条没有副作用的表达式语句，它什么也没做。将来的ECMAScript标准希望将use用做关键字，这样就可以省略引号了。

- 它只能出现在脚本代码的开始或者函数体的开始、任何实体语句之前。但它不必一定出现在脚本的首行或函数体内的首行，因为“use strict”指令之后或之前都可能有其他字符串直接量表达式语句，并且JavaScript的具体实现可能将它们解析为解释器自有的指令。在脚本或者函数体内第一条常规语句之后字符串直接量表达式语句只当做普通的表达式语句对待；它们不会当做指令解析，它们也没有任何副作用。

使用“use strict”指令的目的是说明（脚本或函数中）后续的代码将会解析为严格代码（strict code）。如果顶层（不在任何函数内的）代码使用了“use strict”指令，那么它们就是严格代码。如果函数体定义所处的代码是严格代码或者函数体使用了“use strict”指令，那么函数体的代码也是严格代码。如果eval()调用时所处的代码是严格代码或者eval()要执行的字符串中使用了“scrict code”指令，则eval()内的代码是严格代码。

严格代码以严格模式执行。ECMAScript 5中的严格模式是该语言的一个受限制的子集，它修正了语言的重要缺陷，并提供健壮的查错功能和增强的安全机制。严格模式和非严格模式之间的区别如下（前三条尤为重要）：

- 在严格模式中禁止使用with语句。
- 在严格模式中，所有的变量都要先声明，如果给一个未声明的变量、函数、函数参数、catch从句参数或全局对象的属性赋值，将会抛出一个引用错误异常（在非严格模式中，这种隐式声明的全局变量的方法是给全局对象新添加一个新属性）。
- 在严格模式中，调用的函数（不是方法）中的一个this值是undefined。（在非严格模式中，调用的函数中的this值总是全局对象）。可以利用这种特性来判断JavaScript实现是否支持严格模式：

```
var hasStrictMode = (function() { "use strict"; return this===undefined}());
```

- 同样，在严格模式中，当通过call()或apply()来调用函数时，其中的this值就是通过call()或apply()传入的第一个参数（在非严格模式中，null和undefined值被全局对象和转换为对象的非对象值所代替）。
- 在严格模式中，给只读属性赋值和给不可扩展的对象创建新成员都将抛出一个类型错误异常（在非严格模式中，这些操作只是简单地操作失败，不会报错）。
- 在严格模式中，传入eval()的代码不能在调用程序所在的上下文中声明变量或定义函数，而在非严格模式中是可以这样做的。相反，变量和函数的定义是在eval()创建的新作用域中，这个作用域在eval()返回时就弃用了。

- 在严格模式中，函数里的arguments对象（见8.3.2节）拥有 传入函数值的静态副本。在非严格模式中，arguments对象具有“魔术般”的行为，arguments里的数组元素和函数参数都是指向同一个值的引用。
- 在严格模式中，当delete运算符后跟随非法的标识符（比如变量、函数、函数参数）时，将会抛出一个语法错误异常（在非严格模式中，这种delete表达式什么也没做，并返回false）。
- 在严格模式中，试图删除一个不可配置的属性将抛出一个类型错误异常（在非严格模式中，delete表达式操作失败，并返回false）。
- 在严格模式中，在一个对象直接量中定义两个或多个同名属性将产生一个语法错误（在非严格模式中不会报错）。
- 在严格模式中，函数声明中存在两个或多个同名的参数将产生一个语法错误（在非严格模式中不会报错）。
- 在严格模式中是不允许使用八进制整数直接量（以0为前缀，而不是0x为前缀）的（在非严格模式中某些实现是允许八进制整数直接量的）。
- 在严格模式中，标识符eval和arguments当做关键字，它们的值是不能更改的。不能给这些标识符赋值，也不能把它们声明为变量、用做函数名、用做函数参数或用做catch块的标识符。
- 在严格模式中限制了对调用栈的检测能力，在严格模式的函数中，arguments.caller和arguments.callee都会抛出一个类型错误异常。严格模式的函数同样具有caller和arguments属性，当访问这两个属性时将抛出类型错误异常（有一些JavaScript的实现在非严格模式里定义了这些非标准的属性）。

5.8 JavaScript语句小结

本章介绍了JavaScript语言中的每种语句。表5-1是本章的总结，列出了每种语句的语法和用途：

表5-1：JavaScript语句语法

语句	语法	用途
break	break [*label*];	退出最内层循环或者退出switch语句，又或者退出label指定的语句
case	case *expression*:	在switch语句中标记一条语句
continue	continue [*label*];	重新开始最内层的循环或重新开始label指定的循环

表5-1：JavaScript语句语法（续）

语句	语法	用途
debugger	debugger;	断点器调试
default	default;	在switch中标记默认的语句
do/while	do *statement* while(*expression*);	while循环的一种替代形式
empty	;	什么都不做
for	for(*init;test;incr*)*statement*	一种简写的循环
for/in	for(*var in object*)*statement*	遍历一个对象的属性
function	function *name*([*param*[],...]){*body*}	声明一个函数
if/else	if(*expr*)*statement*1 [else *statement*2]	执行*statement*1或者*statement*2
label	*label:statement*	给*statement*指定一个名字：*label*
return	return [*expression*];	从函数返回一个值
switch	switch(*expression*){*statements*}	用case或者“default:”语句标记的多分支语句
throw	throw *expression*;	抛出异常
try	try {*statements*} [catch {*handler statements*}] [finally {*cleanup statements*}]	捕获异常
use strict	"use strict"	对脚本和函数应用严格模式
var	var *name*=[=*expr*][,...];	声明并初始化一个或多个变量
while	while(*expression*) *statement*	基本的循环结构
with	with(*object*) *statement*	扩展作用域链（不赞成使用）