CHAPTER

第 2 章

# JavaScript 的语法

当然，我们可以从很多个方面来阐述“JavaScript 是怎样的一种语言”这个话题，但对于开发人员来说，最直接的感受总是来自语言的语法与约定。

以对 JavaScript 的语法叙述来说，《JavaScript 权威指南》是最好的一本参考书。但我不能期望用户要读完那本厚厚的书才能阅读本书，因此还是在这里讲述一下语法。

很多人、很多书会把浏览器、DOM 等作为 JavaScript 的一部分进行讲述。在我看来，JavaScript 只是一种语言，由标识符、值、语句等要素构成。而且本章（包括本书）面向的都是有一定开发经验的程序员，所以仅讲述语法中的关键部分，并不打算讨论除此之外的一些细节。

由于本章是概述性质的，因此请留意每小节之前对内容的概括和汇总性的表格。它们可能是从另一个角度概括、汇总了相关的知识，因此或许出现与你正在阅读的书籍（或既有的知识）不一致的情况。但这些不一致，却是我们后面进一步讨论语言的基础。

## 2.1 语法综述

一般来说，语言中的标识符可以分为两类，一类用于命名语法（以及类型），一类用于命名值（的存储位置）。前者被称为“语法关键字”，后者则被称为“变量”和“常量”。

由此引入了一个概念：绑定。从标识符的角度来说，绑定分为语法关键字与语法（语

义）逻辑的绑定，以及变量与它所存储值的位置的绑定。语法关键字对逻辑的绑定的结果，是作用域的限定；变量对位置的绑定的结果，则是变量生存周期的限定。

### 2.1.1 标识符所绑定的语义

我们看到了程序语言中“声明”的意义（这里强调是意义，而非定义）：所谓声明，即是约定变量的生存周期和逻辑的作用域。由于这里的“声明”已经涵盖了逻辑与数据（这相当于“程序”的全部），因此整个编程的过程，其实被解释成了“说明逻辑和数据”的过程。

- 纯粹陈述“值”的过程，被称为变量和类型声明。
- 纯粹陈述“逻辑”的过程，被称为语句（含流程控制子句）。
- 陈述“值与（算法的）逻辑”的关系的过程，被称为表达式。

表 2-1 阐述了主要标识符与其语义关系。

**表 2-1 标识符语义关系的基本分类**

| | 标识符 | 子分类 | JavaScript 示例（部分） |
|---|---|---|---|
| 与值相关 | **类型** | | （无显式类型声明） |
| | **变量** | 直接量<br>对象 | `null`<br>`undefined`<br>`new Object()` |
| 与逻辑和值都相关 | **表达式（*注 1）** | 值运算<br>对象存取 | `v = 'this is a string.'`<br>`obj.constructor` |
| | **逻辑语句（*注 2）** | 顺序<br>分支<br>循环 | `v = 'this is a string.';`<br>`if (false) {`<br>`  // ...`<br>`}` |
| 与逻辑相关 | **流程控制语句** | 标签声明<br>一般流程控制子句<br>异常 | `break;`<br>`continue;`<br>`return;`<br>`try {`<br>`  // ...`<br>`}`<br>`catch (e) {`<br>`  // ...`<br>`}` |
| | **其他** | 注释 | （略） |

*注 1：表达式首先是与值相关的，但因为存在运算的先后顺序，因此它也有与逻辑相关的含义。

*注 2：在 JavaScript 中，逻辑语句是有值的，因此它也是与值相关的。这一点与多数语言不一样。

### 2.1.2 识别语法错误与运行错误

一般来说，JavaScript 引擎会在代码装入时先进行语法分析，如果语法分析通不过，整个脚本代码块都不执行；当语法分析通过时，才会执行这段脚本代码。若在执行过程中出错，那么在同一代码上下文中、出错点之后的代码将不再执行。

不同引擎处理这两类错误的提示的策略并不相同。例如，在 IE 的 JScript 脚本引擎环境中，两种错误的提示大多数时候看起来是一样的。要在不同的脚本引擎中简单地区别两种错误，较为通行的方法是在代码片断的最前面加上一行输出，例如，使用 `alert()` 来显示一个信息[1]。脚本引擎的出错提示在该行之前，则是语法分析期错误。例如：

```
var head = 'alert("loaded.");';

// 示例 1：声明函数的语法错误
var code = 'funcion func(){};';
eval(head + code);
```

如果在该行之后，则是执行期错误。例如：

```
var head = 'alert("loaded.");';

// 示例 2：执行时发现未定义 value 变量，触发运行期错误
var code = 'value++';
eval(head + code);
```

我们在此强调这两种错误提示，是因为本章主要讨论语法问题，而在特定脚本引擎中，一段代码是执行异常还是语法分析错误，是需要通过上述的方法来区分的。但如果有该引擎下的调试器或脚本宿主环境，允许加载用户的错误处理代码（例如，浏览器中可以通过 `window.onerror` 响应错误处理），也还是有其他方法的。

## 2.2 JavaScript 的语法：变量声明

JavaScript 是弱类型语言。但所谓弱类型语言，只表明该语言在表达式运算中不强制校验运算元的数据类型，而并不表明该语言是否具有类型系统。所以有些书在讲述 JavaScript 时说它是“无类型语言（untype language）”，其实是不正确的。本小节将对 JavaScript 变量的数据类型做一个概述。

一般来说，JavaScript 的变量可以从作用域的角度分为全局变量与局部变量。为了便

[1] 事实上，更好的做法是使用集成调试环境在这里设置一个断点。这些集成调试或集成开发环境，包括在 Microsoft JScript 中的 Script Debugger 或 Script Editor、Netscape 的 JavaScript Debugger、Mozilla Firefox 的 Firebug 插件、Adobe 的 Adobe ExtendScript Toolkit 等。自 1998 年以来，我用过上述所有的调试环境在不同的宿主或引擎中开发脚本，然而直到现在都还有开发人员抱怨“JavaScript 无法单步调试”。这实在是一件令人无可奈何的事情，因为本书的重点并不在于 step by step 地教会开发人员找到及使用某种开发工具。

于本小节的叙述，读者可以简单地认为：所谓全局变量是指在函数外声明的变量，局部变量则是在函数或子函数内声明的变量。更详细的变量作用域问题，将在“3.2.4 模块化的效果：变量作用域”中讲述。

## 2.2.1 变量的数据类型

相对于 Pascal、C 等语言，JavaScript 没有明确的类型声明过程——事实上在 JavaScript 约定的保留字列表中，根本就没有 `type` 或 `define` 关键字。总的来说，JavaScript 识别 6 种数据类型，并在运算过程中自动处理语言类型的转换。

### 2.2.1.1 基本数据类型

我们称 JavaScript 识别的这 6 种类型为基本类型、基础类型或元类型（meta types），具体内容见表 2-2。

表 2-2 JavaScript 的 6 种基本数据类型

| 类型 | 含义 | 说明 |
|---|---|---|
| `undefined` | 未定义 | 未声明的变量，或者声明过但未赋值的变量的值，会是 `undefined`。也可以显式或隐式地给一个变量赋值为 `undefined` |
| `number` | 数值 | 除赋值操作之外，只有数值与数值的运算结果是数值；一些函数/方法的返回值是数值 |
| `string` | 字符串 | 不能直接读取或修改字符串中的单一字符 |
| `boolean` | 布尔值 | `true/false` |
| `function` | 函数(*注 1) | JavaScript 中的函数存在多重含义 |
| `object` | 对象(*注 2) | 基于原型继承的面向对象 |

*注 1：在 JavaScript 中，函数的多重含义包括：函数、方法、构造器、类以及函数对象等。

*注 2：因为不具备对象系统的全部特性，因此 JavaScript 通常被称为基于对象而非面向对象的语言。但这并不是其“基于原型继承”带来的问题——我的意思是说，基于原型继承也可以构造“完全面向对象”的系统。

任何一个变量或值的类型都可以（而且应当首先）使用 `typeof` 运算得到。`typeof` 是 JavaScript 内部保留的一个关键字，与其他语言不一样的是，`typeof` 关键字可以像函数调用一样，在后面跟上一对括号。例如：

```
// 示例 1：取变量的类型
var str = 'this is a test.';
alert(typeof str);
// or
// alert(typeof(str));

// 示例 2：对直接量取类型值
alert(typeof 'test!');
```

typeof()中的括号，只是产生了一种“使 typeof 看起来像一个函数”的假象。关于这个假象的由来，我随后会在“2.7 运算符的二义性”再讲。现在，你只需知道它的确可以这样使用就足够了。

typeof 运算总是以字符串形式返回上述 6 种类型值之一。如果不考虑 JavaScript 中的面向对象编程[2]，那么这个类型系统的确是足够简单的。

#### 2.2.1.2 值类型与引用类型

变量不但有数据类型之别，而且还有值类型与引用类型之别，这种分类方式主要约定了变量的使用方法。JavaScript 中的 6 种值类型与引用类型见表 2-3。

表 2-3 JavaScript 中的值类型与引用类型

| 数据类型 | 值/引用类型 | 备注 |
|---|---|---|
| undefined | 值类型 | 无值 |
| number | 值类型 | |
| boolean | 值类型 | |
| string | 值类型 | 字符串在赋值运算中会按引用类型的方式来处理 |
| function | 引用类型 | |
| object | 引用类型 | |

在 JavaScript 中，“全等（===）运算符”用来对值类型/引用类型的实际数据进行比较和检查。按照约定，基于上述类型系统的运算中（以下所谓“值”，也包括 undefined）：

- 一般表达式运算的结果总是值。
- 函数/方法调用的结果可以返回值或者引用。
- 值与引用、值与值之间即使等值（==），也不一定全等（===）。
- 两个引用之间如果等值（==），则一定全等（===）。

从表面上来看，一个值应该与其自身“等值/全等”。但事实上，在 JavaScript 中存在一个例外：一个 NaN 值，与自身并不等值，也不全等。

JavaScript 中的值类型与引用类型，同其他通用高级语言（或像汇编语言一样的低级语言）一样，表达的含义是数据在运算时的使用方式：参与运算的是其值亦或其引用。因此，在下面的示例中，当两次调用函数 func()时，各自传入的数据采用了不同的方式：

```
var str = 'abcde';
var obj = new String(str);
```

[2] JavaScript 存在两套类型系统，其“对象类型系统”可以视为“基本数据类型”中的 object 数据类型的一个分支。关于这一点，可以参考：3.4.5 JavaScript 中的对象（构造器）。

```
function newToString() {
  return 'hello, world!';
}
function func(val) {
  val.toString = newToString;
}

// 示例 1: 传入值
func(str);
alert(str);

// 示例 2: 传入引用
func(obj);
alert(obj);
```

从语义上来说，由于在示例 1 中实际只传入了 str 的值，因此对它的 toString 属性的修改是无意义的；而在示例 2 中传入了一个对象的引用，因此对它的 toString 属性修改将会影响到后来的运算。所以我们看到这两个示例返回的结果不一致。

关于“为什么可以修改一个值类型数据的属性”这个问题，以及 str 在传入 func()后发生的一些细节，我们在“5.5.3 包装值类型数据的必要性与问题”中会进一步说明。

## 2.2.2 变量声明

JavaScript 中的变量声明有两种方法：

- 显式声明。
- 隐式声明（即用即声明）。

所谓显式声明，一般是指用关键字 var 进行的声明语句[3]。例如：

```
// 声明变量 str 和 num
var str = 'test';
var num = 3 + 2 - 5;
```

显式声明也包括一些语句中使用 var 进行的显式声明，例如 for 语句：

```
// 声明变量 n
for (var n in Object) {
  // ...
}
// 声明变量 i,j,k
for (var i,j,k=0; k<100; k++) {
  // ...
}
```

最后两种情况是函数中的函数名声明，以及异常捕获子句中声明的异常对象。例如：

```
// 声明函数 foo
```

[3] 参见：2.4.1.3（显式的）变量声明语句。

```
function foo() {
  str = 'test';
}

//声明异常对象 ex
try {
  // ...
}
catch (ex) {
  // ...
}
```

而隐式声明则发生在不使用关键字 var 的赋值语句中[4]。例如：

```
// 当 aVar 未被声明时，以下语句将隐式地声明它
aVar = 100;
```

解释器总是将显式声明理解为“变量声明”，而对隐式声明则不一定：

- 如果变量未被声明，则先声明该变量并立即赋给值。
- 如果变量已经声明过，则该语句是赋值语句。

隐式声明的语法效果如图 2-1 所示。

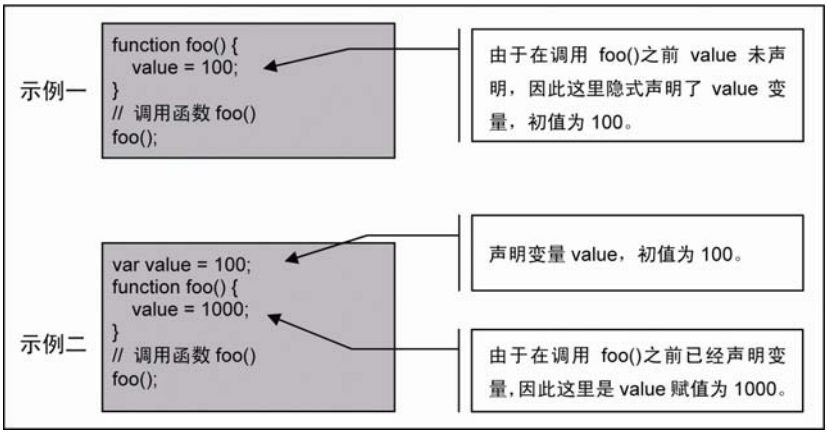


图 2-1 隐式声明的语法效果

## 2.2.3 变量与直接量

变量声明通常具有两个方面的特性，一是声明类型，二是声明初值。但 JavaScript 中没有类型声明的概念，因此变量声明仅限于说明一个变量的初值。在声明中，等号右边既可以是表达式——这意味着将表达式运算的结果作为该变量的初值，也可以是更为强大和灵活的直接量声明。例如：

[4] 参见：2.4.1.2 赋值语句与隐式的变量声明。

```
var str = 'test';       // 'test'是直接量
var num = 3 + 2 - 5;    // 3+2-5 是一个表达式
```

直接量类似于汇编语言中的立即值，是无须声明就可以立即使用的常值。本小节中主要讲述直接量声明的方法——当然直接量也可以作为表达式运算的运算元使用，这与其他语言中是一致的。表 2-4 简要说明了 JavaScript 中能进行直接量声明的数据类型和对象。

表 2-4 JavaScript 中的 6 种基本类型的直接量声明

| 类型 | 直接量声明 | 包装对象 | 说明 |
|---|---|---|---|
| undefined | v = undefined | 无 | |
| string | v = '……'<br>v = "……" | String | 2.2.3.1 字符串直接量、转义符 |
| number | v = 1234 | Number | 2.2.3.2 数值直接量 |
| boolean | v = true<br>v = false | Boolean | |
| object | v = null | 无 | |
| | v = { …… } | Object | 2.5.1 对象直接量声明与实例创建 |
| | v = [ …… ] | Array | |
| | v = /.../... | RegExp | |
| function | v = function() { …… } | Function | 2.2.4 函数声明 |

本小节的部分问题参见：

- 《JavaScript 权威指南》
- 2.5.1 对象直接量声明与实例创建
- 5.4.2.1 语法声明与语句含义不一致的问题
- 5.7.4 值类型之间的转换
- 5.5 包装类：面向对象的妥协

### 2.2.3.1 字符串直接量、转义符

你总是可以用一对双引号或一对单引号来表示字符串直接量。在早期 Netscape 的 JavaScript 中允许出现非 Unicode 的字符[5]，但现在 ECMA 标准统一要求 JavaScript 中的字符串必须是 Unicode 字符序列。

转义符主要用于在字符串中包含控制字符，以及当前操作系统语言（以及字符集）

[5] 在早期 Netscape 的 JavaScript 中编程时，使用 escape()可能会解出单字节的编码，而使用 unescape()时则可能在字符串中包含相应的 ASCII 字符。即使是在 IE 的早期版本（for Windows 9x）中，也可能由于 cookies 存取而出现在字符串中存在 ASCII 字符序列的情况。

不能直接输入的字符，也可以用于字符串中嵌套引号（不过你总是可以在单引号声明的字符串中直接使用双引号，或者反过来在双引号中使用单引号）。转义符总是用一个反斜线字符“\”引导，包括表 2-5 中的转义序列。

表 2-5 JavaScript 中的字符串转义序列

| 转义符 | 含义 | 转义符 | 含义 |
|---|---|---|---|
| \b | 退格符 | \' | 单引号 |
| \t | 水平制表符 | \" | 双引号 |
| \v | 垂直制表符 | \\ | 反斜线字符 |
| \n | 换行符 | \0 | 字符 NUL（编码为 0 的字符） |
| \r | 回车符 | \xnn | ASCII 字符编码为 nn 的字符（*注 1） |
| \f | 换页符 | \unnnn | Unicode 字符编码为 nnnn 的字符 |

*注 1：将被转换为 Unicode 字符存储。

除了定义转义符之外，当反斜线字符“\”位于一行的末尾（其后立即是代码文本中的换行）时，也用于表示连续的字符串声明，这在声明大段文本块时很有用。例如：

```
1 var aTextBlock = '\
2 abcdefghijklmnopqrstuvwxyz\
3  \
4 123456789\
5  \
6 +-*/';
```

注意第 3 行与第 5 行中各包括一个空格，因此输出时第 2、4、6 行将用一个空格分开。显示为：

```
abcdefghijklmnopqrstuvwxyz 123456789 +-*/
```

在这种字符串表示法中也可以使用其他转义符，只要它们出现在文本行最后的这个“\”字符之前即可。而且与一般习惯不同的是，不能在这种表示的文本行末使用注释。

另外一个需要特别说明的是“\0”，它表示 NUL 字符。在某些语言中，NUL 被用于说明一种“以#0 字符结束的字符串”（这也是 Windows 操作系统所直接支持的一种字符串形式），这种情况下字符串是不能包括该 NUL 字符串的。但在 JavaScript 中，这是允许存在的。这时，NUL 字符是一个真实存在于字符序列中的字符。下例说明 NUL 字符在 JavaScript 中的真实性：

```
// 或
// str = String.fromCharCode(0, 0, 0, 0, 0);
str = '\0\0\0\0\0';

// 显示字符串长"5", 表明 NUL 字符在字符串中是真实存在的
alert(str.length);
```

在 JavaScript 中也可以用一对不包含任意字符的单引号与双引号来表示一个空字符串（Null String），其长度值总是为 0。比较容易被忽视的是，空字符串与其他字符串一样也可以用做对象成员名。例如：

```
obj = {
  '': 100
}
// 显示该成员的值: 100
alert(obj['']);
```

### 2.2.3.2 数值直接量

数值直接量总是以一个数字字符或一个点字符“.”，以及不多于一个的正值符号“＋”或负值符号“－”开始。当以数字字符开始时，它有三条规则：

- 如果以 0x 或 0X 开始，则表明是一个十六进制数值。
- 如果仅以 0 开始，则表明是一个八进制数值。
- 其他情况下，表明是一个十进制整数或浮点数。

当以点字符“.”开始时，它总是表明一个十进制浮点数。正值符号“＋”、负值符号“－”总是可以出现在上述两种表示法的最前面。例如：

```
1234       // 十进制整数
01234      // 八进制整数
0x1234     // 十六进制整数
-0X1234    // 负值的十六进制整数
+100       // 正值的十进制整数
```

当一个直接量声明被识别为十六进制数值时，该直接量由 0~9 和 A~F 字符构成；被识别为八进制数值时，该直接量由 0~7 字符构成，如果在语法分析中发现此外其他字符，则出现语法分析错误。

当一个直接量被识别为十进制整型数时，它内部的存放格式可能是浮点数，也可能是整型数，这取决于不同引擎的实现。因此不能指望 JavaScript 中的整型数会有较高的运算性能[6]。但是你可以用位运算来替代算术运算，这时引擎总是以整型数的形式来运算的——即使运算元是一个浮点数。

当一个直接量被识别为十进制时，它可以由 0~9，以及（不多于一个的）点字符“.”或字符 e、E 组成。当包括点字符“.”、字符 e、E 时，该直接量总被识别为浮点数（注意某些引擎会优化一些直接量的内部存储形式）。例如：

[6] 在 JScript 中，引擎对直接量会进行特别的分析并以最优的形式来存放它。例如，值 `100` 与值 `100.0` 在 JScript 中其实都是以一个 `LongInt` 类型的整数值来存放的，而 `100.1` 则以 `Double` 类型的浮点数来存放。

```
3.1415926
12.345
.1234
.0e8
1.02E30
```

当使用带字符 e、E 的指数法表示时，也可以使用正、负符号来表示正、负整数指数。例如：

```
1.555E+30
1.555E-30
```

## 2.2.4 函数声明

函数直接声明的语法是：

```
function functionName()
  // ……
}
```

当 functionName 是一个有效标识符时，表示声明一个具名函数；如果省略，则表示声明一个匿名（anonymous）函数。functionName 后使用一对不能省略的“( )”来表示形式参数列表。所谓形式参数，是指可以在函数体内部使用的、有效的标识符名。你可以声明零个至任意多个形式参数，即使你在函数体内部并不使用它。你也可以不声明形式参数，这时也可以在函数体内使用一个名为 arguments 的内部对象来存取调用中传入的实际参数。

函数体中可以有零至任意多行代码或内嵌的（子级的）函数。如果在函数体内出现显式变量声明，则视为函数体内部的局部变量；该函数的内嵌函数的名字也作为局部变量。

在 JScript 中，所有在代码内出现的具名函数（直接量）声明，都将视为所在的语法作用域中的一个变量标识符。这在 SpiderMonkey JavaScript 中存在一项限制：在当前作用域中，表达式中具名函数只识别为匿名函数而忽略它的函数名。而在 JScript 中将有一个具名函数声明[7]：

```
// 使用一对括号来强制表达式运算
(function foo()
  // ……
});
//在 SpiderMonkey 中，上述声明在当前作用域中是匿名的，所以下一行代码导致异常
alert(foo);
```

[7] 这个问题将在“5.4.2.1 语法声明与语句含义不一致的问题”中予以更详细的讨论。

# 2.3 JavaScript 的语法：表达式运算

相较于其他语言，JavaScript 在运算符上有一种特殊性：许多语句/语法分隔符也同时是运算符——它们的含义当然是不同的，我的意思只是强调它们使用了相同的符号，例如括号“()”既是语法分隔符也是运算符。

在 JavaScript 中，运算符大多数是特殊符号，但也有少量单词——我们在前面用来取数据类型的 typeof，其实就是一个运算符。表 2-6 列举了这些单词形式的运算符，应当避免把它们误解成语句。

表 2-6 JavaScript 中单词形式的运算符

| 运算符/符号 | | 运算符含义 | 备注 |
|---|---|---|---|
| 单词形式的运算符 | typeof | 取变量或值的类型 | 参见:<br>2.3.7 特殊作用的运算符 |
| | void | 运算表达式并忽略值 | |
| | new | 创建指定类的对象实例 | 与面向对象相关。参见:<br>2.5 面向对象编程的语法概要 |
| | in | 检查对象属性 | |
| | instanceof | 检查变量是否指定类的实例 | |
| | delete | 删除实例属性 | |

表达式由运算符与运算元构成。运算元除了包括变量，还包括函数（或方法）的返回值，此外也包括直接量。但 JavaScript 中也可以存在没有运算符的表达式，这称为“单值表达式”。单值表达式有值的含义，表达式的结果即是该值，主要包括：

- this 引用。
- 变量引用，即一个已声明的标识符。
- 直接量，包括 null、undefined、字符串、布尔值、数值、正则表达式。

在 ES5 中，将下面几种表达式与单值表达式一起称为基本表达式（Primary Expression）——正则表达式已经包含在单值表达式的“直接量”中：

- 数组初始器/直接量，即[ … ]。
- 对象初始器/直接量，即{ … }。
- 表达式分组运算，即( … )。

除此之外，一个 JavaScript 表达式中必然存在至少一个运算符。运算符可以有 1～3 个运算元，没有运算元的、孤立于代码上下文的运算符是不符合语法的。

通过对表达式的考察，我们发现 JavaScript 的表达式总有结果值——一个值类型或引用类型的数据，或者 undefined。其中，单值表达式的结果是值本身，其他表达式的结

果是运算的结果值（也因此必然有运算符）。

复合表达式由多个表达式连接构成。如前所述，由于每一个独立的表达式都总有结果值，因此该值能作为“邻近”的表达式的运算元参与运算。有了这样的关系，我们就总可以将无限个表达式“邻近”地连接成复合表达式，该复合表达式的运算结果也与其他普通的表达式一样：值类型或引用类型的数据，或者 undefined。

如同数学含义上的“运算”存在优先性（例如乘除法优先于加减法），复合表达式在“按从左至右的”邻近关系运算的同时，也受到运算符的优先级的影响。为了让这种运算次序可控，就有了强制优先级运算符。出于有了默认次序、优先级次序和强制优先级，则表达式就存在算法的逻辑上的含义——这就是本章“2.1 语法综述”中说表达式既有值的含义，也有逻辑的含义的原因[8]。

通过对值的含义的考查，我们会注意到，所有 JavaScript 表达式的运算结果，要么产生一个“基本类型的值”，要么产生“对一个对象实例的引用”。根据运算元和结果值的不同，我们对运算符做一个简单分类，见表 2-7。

表 2-7 JavaScript 中的运算符按结果值的分类

| 分类 | 说明 | 运算符示例 | 运算元 | 目标类型 | 章节 |
|---|---|---|---|---|---|
| 数值运算 | 一般性的数值运算 | + - * / % | number | number | 2.3.1 |
| 位运算 | 数值的位运算 | ~ & \| ^ << 等 | | | |
| 逻辑运算 | 布尔值运算 | ! && \|\| | boolean | boolean | |
| | 值逻辑运算 | && \|\| | （运算元） | （运算元） | 2.3.2 |
| 字符运算 | （仅有）字符串连接 | + | string | string | 2.3.3 |
| 等值检测 | 检测两个值是否相等 | == != === 等 | * | boolean | 2.3.4 |
| 赋值运算 | 一般赋值和复合赋值 | = += 等 | * | * | 2.3.5 |
| 函数调用 | （仅有）函数调用 | () | function | * | 2.3.6 |
| 对象 | 对象创建、存取、检查等 | . [] new | object | * | （*注 1） |
| 其他 | 表达式运算、typeof 运算等 | void typeof 等 | （表达式等） | * | 2.3.7 |

*注 1：有关对象的运算、语法等，我们将在“2.5 面向对象编程的语法概要”章节中讲述。

## 2.3.1 一般表达式运算

无论在哪种语言中，一般表达式运算总是一个很大的分类。今后我们会讲到这种状况的成因，但现在，我们只需概要地讲述一般表达式。

JavaScript 中的一般表达式运算只操作两种运算数：数值和布尔值。这些运算的运算

[8] 但表达式运算的本质目的还是在于求值，而非表达逻辑。这将在“4.3 从运算式语言到函数式语言”中更详细地讨论。

元与结果值总是同一类型的。对数据结构、存储系统或计算原理有些基本了解的人都应该知道，这两种数据通常是可以被存储在基本的存储单元中，并参与 CPU 指令运算的[9]。

我们强调这里说的“一般表达式运算”的逻辑运算仅指“布尔值运算”，因此它的运算元和结果值都必然是布尔值。当然在使用中，这可能由编译器进行了类型转换，例如下面的表达式中，无论 aVar 是其他何种类型，都将被“逻辑否（!）”运算符转换为 bool 值参与运算：

```
!aVar
```

除了“加减乘除”这类一般性的数值运算之外，这里说的“一般表达式运算”也包括数值的位运算。在位运算操作中，JavaScript 强制运算目标为一个有符号的 32 位整型数[10]：如果目标是非数值，那么会被强制转换为数值；如果目标是浮点数，那么会被取整；否则，将目标识别为有符号整型数。

JavaScript 中的类型转换是一种动态语言特性，因此有关它的细节请参考“5.7 类型转换”。

对于不同的硬件系统，数值和布尔值的表示法、表示范围都可能不同。但对于 JavaScript 这种运行在解释系统中的语言来说，它会通过一些约定来清除硬件差异的影响，如在“2.2.3.2 数值直接量”中所讨论到的一些规则。

## 2.3.2 逻辑运算

一般语言中，逻辑运算与布尔运算是等义的，其运算元与目标类型都是布尔值（true/false）。JavaScript 当然支持这种纯布尔运算，上一小节已经对此有过叙述。不但如此，JavaScript 还包括另外一种逻辑运算，它的表达式结果类型是不确定的。

只有“逻辑或（||）”和“逻辑与（&&）”两种运算能做这样的事。它们的使用方法与运算逻辑都与基本的布尔运算一致，例如：

```
var str = 'hello';
var obj = {};
x = str || obj;
y = str && obj;
```

这种运算的特别之处在于：运算符“||”与“&&”既不改变运算元的数据类型，也不强制运算结果的数据类型。除此之外，还有以下两条特性：

[9] 这并不表明 JavaScript 采用这种方式存储或在运算中传送这些运算数，但这的确是 JavaScript 保留这两种值类型的原因之一。

[10] 这里的转换、识别规则是非常复杂且与 JavaScript 版本相关的，因此本书只是（并不准确地）概述了几种可能性。有关转换的细节，请参考《JavaScript 权威指南》一书中的“5.8 逐位运算符”。

- 运算符会将运算元理解为布尔值，以进行布尔运算。
- 运算过程（与普通布尔运算一样）是支持布尔短路的。

由于支持布尔短路，因此在上例中“`str || obj`”表达式只处理第一个运算元就可以有结果，其结果值是 `str`——转换为布尔值时为 `true`，不过由于前面所述的“不强制运算结果的数据类型”，所以表达式的结果值仍是“`str`”。同样，若以“`str && obj`”为例，其返回结果值就会是“`obj`”了。

这种逻辑运算的结果一样可以用在任何需要判断布尔条件的地方，包括 `if` 或 `while` 语句，以及复合的布尔表达式中。例如：

```
(续上例)

// 用于语句
if (str || obj) {
  ...
}

// 用于复杂的布尔表达式
z = !str && !(str || obj);
```

由于表达式的运算元可以是值或其他表达式（包括函数调用等），因此连续的逻辑运算也可以用来替代语句。这也是一种被经常提及的方法，关于这一点，请参考如下章节：

- 4.3.3.1 通过表达式消灭分支语句
- 7.14 使用更复杂的表达式来消减 if 语句

## 2.3.3 字符串运算

JavaScript 中的字符串有且只有一种“字符串运算”，就是字符串连接，该运算相应的运算符是加号“+”，但的确还有其他几种运算可以作用于字符串。符号“+”还可以被用在其他两个地方：一元正值运算符和数值加法运算符。因此一个带有符号“+”的表达式是不是“字符串连接”运算，取决于它在运算时存在几个操作数，以及每个操作数的类型。

字符串连接运算总是产生一个新的字符串，它在运算效果（结果值）上完全等同于调用字符串对象的 `concat()`方法。

你不可以直接修改字符串中的指定字符。

**字符串其实还可以参与其他运算，例如比较、等值、赋值等运算。但在这里，我们把它放在相应的其他分类中讲述。**

## 2.3.4 比较运算

### 2.3.4.1 等值检测

等值检测的目的是判断两个变量是否相同或相等。我们说相同与不相同，是指运算符“===”和“!==”的运算效果；说相等与不相等，是指运算符“==”和“!=”的运算效果。

具体来说，等值检测是指如表 2-8 所示的运算符的运算效果。

表 2-8 比较运算中的等值检测

| 名称 | 运算符 | 说明 |
| --- | --- | --- |
| 相等 | == | 比较两个表达式，看是否相等 |
| 不等 | != | 比较两个表达式，看是否不相等 |
| 严格相等 | === | 比较两个表达式，看值是否相等并具有相同的数据类型 |
| 不严格相等 | !== | 比较两个表达式，看是否具有不相等的值或数据类型不同 |

对于等值检测来说，最简单和有效率的方法当然是比较两个变量引用（所指向的内存地址）。但这并不准确，因为我们显然会在两个不同的内存地址上存放同样的数据，例如两个相同的字符串。因此，比较引用虽然高效，但很多时候，我们却需要比较两个变量的值。在下面的讨论中，我们会先忽略“比较引用”时的问题，侧重讲述值的比较。

我们先讨论等值检测中“相等”的问题。它遵循表 2-9 的运算规则。

表 2-9 等值检测中“相等”运算规则

| 类型 | 运算规则 |
| --- | --- |
| 两个值类型进行比较 | 转换成相同数据类型的值进行“数据等值”比较 |
| 值类型与引用类型比较 | 将引用类型的数据转换为与值类型数据相同的数据，再进行“数据等值”比较 |
| 两个引用类型比较 | 比较引用（的地址） |

运算规则中所谓“数据等值”，是仅指针对“值类型”的比较而言，表明比较变量所指向的存储单元中的数据（通常指“内存数据”）。

在三种值类型（数值、布尔值和字符串）中，数值和布尔值的“数据等值”检测开销都很小，但对字符串检测时就会存在非常大的开销。因为必须对字符串中的每一个字符进行比较，才能判断两个字符串是否相等。

下面的代码说明两个值类型的字符串检测：

```
// 示例 1: 对两个值类型字符串进行“值相等”的检测
var str1 = 'abc' + 'def';
alert(typeof str1);   // 显示'string'

var str2 = 'abcd' + 'ef';
alert(typeof str2);   // 显示'string'

// 下面的运算需要进行六次字符比较，才能得到结果值 true
alert(str1 == str2);
```

接下来我们讨论等值检测中“相同”的问题。它遵循表 2-10 的运算规则。

表 2-10 等值检测中“相同”运算规则

| 类型 | 运算规则 |
| --- | --- |
| 两个值类型进行比较 | 数据类型不同，则必然“不相同”；<br>数据类型相同时，进行“数值等值”比较 |
| 值类型与引用类型比较 | 必然“不相同” |
| 两个引用类型比较 | 比较引用（的地址） |

所以“相同与否”的检测，仅对两个相同数据类型、值类型的数据有意义（其他情况下的比较值都是“不同”的），这时所比较的方法，也是完全的“数据等值”的比较。换而言之，下面的代码与上一个示例所发生的运算，以及运算效果是完全一致的：

```
// 示例 2: 对两个值类型字符串进行“值相同”的检测
var str1 = 'abc' + 'def';
var str2 = 'abcd' + 'ef';

// 运算过程和结果完全等同于上一个示例
alert(str1 === str2);
```

通过上面的示例，我们成功地否定了一个习惯性的说法：“===”运算是比较引用的，而“==”是比较值的。因为在示例 2 中，我们看到，str1 与 str2 是两个不同的值类型的变量，但它们是完全相等的。我们事实上也发现，在对两个引用类型的比较运算过程中，“==”与“===”并没有任何的不同。

引用类型的等值比较，将直接“比较引用（的地址）”。这听起来比较拗口，但实际的意义不过是说：如果不是同一个变量或其引用，则两个变量不相等，也不相同。

下面的例子说明这种情况：

```
var str = 'abcdef';
var obj1 = new String(str);
var obj2 = new String(str);

// 返回 false
alert(obj1 == obj2);
alert(obj1 === obj2);
```

我们看到，obj1 与 obj2 是类型相同的，且值都是通过同一个直接量来创建的，但是，由于 String()对象是引用类型，所以它们既不“相等”，也不“相同”。

#### 2.3.4.2 序列检测

从数学概念来说，实数数轴上可比较的数字是无限的（正无穷到负无穷）。该数轴上的有序类型，只是该无限区间上的一些点。但对于具体的语言来说，由于数值的表达范围有限，所以数值的比较也是有限的。

如果一个数据的值能投射（例如通过类型转换）到该轴上的一点，则它可以参与在该轴所表达范围内的序列检测：亦即是比较其序列值的大小。在 JavaScript 中，这包括表 2-11 中的数据类型（其中，`Number` 类型是实数数轴的抽象）。

表 2-11 JavaScript 中可进行序列检测的数据类型

| 可比较序列的类型 | 序列值 |
| --- | --- |
| `boolean` | 0~1 |
| `string` | （*注 1） |
| `number` | `NEGATIVE_INFINITY ~ POSITIVE_INFINITY`（*注 2） |

*注 1：在 JavaScript 中，“字符串”是有序类型的一种特例。一般语言中，“字符（char）”这种数据类型是有序的（字符`#0`~`#255`）。虽然 JavaScript 不存在“字符”类型，但它的字符串的每一个字符，都被作为单一字符来参与序列检测。

*注 2：负无穷~正无穷。值 `NaN` 没有序列值，任何值与 `NaN` 进行序列检测将得到 `false`。

不要以其他高级语言数据类型的分类中的“有序类型”来理解这里的序列。这种所谓的“有序类型”是指该类型的有限集合存在一种有序的排布。例如“字节”这种数据类型即存在序数 `0~255`，而“布尔类型”则是“`0~1`”。这些高级语言中的“有序类型”并不包括实数。

序列检测的含义在于比较变量在序列中的大小，亦即是数学概念中的数轴上点的位置先后。所以运算符见表 2-12。

表 2-12 比较运算中的序列检测

| 名称 | 运算符 | 说明 |
| --- | --- | --- |
| 大于 | `>` | 比较两个表达式，看一个是否大于另一个 |
| 大于等于 | `>=` | 比较两个表达式，看一个是否大于等于另一个 |
| 小于 | `<` | 比较两个表达式，看是否一个小于另一个 |
| 小于等于 | `<=` | 比较两个表达式，看是否一个小于等于另一个 |

运算符同时遵循如表 2-13 所示的运算规则。

表 2-13 序列检测的运算规则

| 类型 | 运算规则 |
| --- | --- |
| 两个值类型进行比较 | 直接比较数据在序列中的大小 |
| 值类型与引用类型比较 | 将引用类型的数据转换为与值类型数据相同的数据，再进行“序列大小”比较 |

续表

| 类型 | 运算规则 |
| --- | --- |
| 两个引用类型比较 | 无意义，总是返回 false（*注 1） |

*注 1：其实，对引用类型进行序列检测运算仍然是可能的，但这与 valueOf()运算的效果有关。关于这一点，我们将在“5.7.5 从引用到值：深入探究 valueOf()方法”中详细讲述。

下面的代码说明了这个运算规则：

```
var o1 = {};
var o2 = {};
var str = '123';
var num = 1;
var b0 = false;
var b1 = true;
var ref = new String();

// 示例 1：值类型的比较，考查布尔值与数值在序列中的大小
alert(b1 < num);    // 显示 false
alert(b1 <= num);   // 显示 true，表明 b1==num
alert(b1 > b0);     // 显示 true

// 示例 2：值类型与引用类型的比较
// ( 空字符串被转换为 0 值 )
alert(num > ref);   // 显示 true

// 示例 3：两个对象（引用类型）比较时总是返回 false
alert(o1 > o2 || o1 < o2 || o1 == o2);
```

下面补充说明字符串的序列检测含义。

只有两个运算元都是字符串时，表 2-12 中所列的 4 个运算符才表示字符串序列检测。任意一个运算元为非字符串时，将按数值来进行比较（也就是将字符串转换为数值参与运算）。下例说明这一点：

```
var s1 = 'abc';
var s2 = 'ab';
var s3 = '101';

var b = true;
var i = 100;

// 示例 1：两个运算元为字符串，将比较每个字符的序列值。所以显示为 true。
alert( s1 > s2 );

// 示例 2：当字符串与其他类型值比较时，将字符串转换为数值比较。所以显示为 true。
alert( s3 > i );
// 示例 3：在将字符串转换为数值时得到 NaN，所以下面的三个比较都为 false。
// (注：变量 b 中的布尔值 true，转换为数值 1 参与运算)
alert( s1 > b || s1 < b || s1 == b );

// 示例 4：两个 NaN 的比较。NaN 不等值也不大于或小于自身，所以下面的三个比较都为 false。
alert( s1 > NaN || s1 < NaN || s1 == NaN );
```

## 2.3.5 赋值运算

JavaScript 里有两种赋值运算符，见表 2-14（v: variant，e: expression）。

表 2-14 JavaScript 中的赋值运算

| 类型 | 示例 | 等价等式 | 操作数类型 |
|---|---|---|---|
| （一般）赋值运算符 | v = e | | |
| 带操作的赋值运算符（复合赋值运算符） | v += e | v = v + e | 字符串，数值 |
| | v -= e | v = v - e | 数值 |
| | v *= e | v = v * e | |
| | v /= e | v = v / e | |
| | v %= e | v = v % e | |
| | v <<= e | v = v << e | 位运算 |
| | v >>= e | v = v >> e | |
| | v >>>= e | v = v >>> e | |
| | v &= e | v = v & e | |
| | v \|= e | v = v \| e | |
| | v ^= e | v = v ^ e | |

在 JavaScript 中，赋值是一个运算，而不是一个语句（如何将它变成语句，是下一小节的话题）。所以，在赋值表达式中，运算符左右都是运算元。当然，按照“表达式”的概念，表达式的运算元既可以是值（也包括立即值），也可以是引用。因此从语法上来说，下面的代码是成立的：

```
// 下面的代码是两个运算元都是立即值的“赋值运算”表达式
100 = 1000;
```

在 JavaScript 中，上面这行代码的确能通过语法检测，但是它会触发一个执行期错误。在 IE 的 JScript 引擎中，错误信息是“不能给 '[number]' 赋值”；在 Firefox 的 SpiderMonkey 引擎中，错误信息则是“左侧无效赋值”。这是由于左侧的运算元是直接量，其存储单元是不可写的，因此在表达式运算过程中，将因为赋值的效果（修改存储单元中的值）无法完成，而导致出错。

这里已经提及了“赋值的效果是修改存储单元中的值”。而这其实就是“赋值运算”的本质。所谓“存储单元”，对于值类型数据来说是存放**值数据**的内存，对于引用类型数据来说则是存放**引用地址**的内存（即指向**引用数据**的指针）。所以赋值运算对值类型来讲是复制数据，而对于引用类型来讲，则只是复制一个地址。

这里存在一个特例：值类型的字符串是一个大的、不确定长度的连续数据块，这导致复制数据的开销很大，所以 JavaScript 中将字符串的赋值也变成了复制（连续数据块起始处的）地址，即该字符串的地址引用。由此引入了三条字符串处理的限制：

- 不能直接修改字符串中的字符。
- 字符串连接运算必然导致写复制，这将产生新的字符串。
- 不能改变字符串的长短，例如修改 `length` 属性是无意义的。

赋值运算符除了等号“=”之外，还有一类“复合赋值运算符”。这类运算符由一个一般表达式运算符与一个赋值运算符复合构成。由于字符串只支持“字符串连接（+）”运算的缘故，在表 2-14 所列的复合运算符中，除第一个“+=”能用于字符串之外，其他的都只能用于数值类型，如果将它们使用于非数值类型，则运算中会出现隐式的类型转换。

### 2.3.6 函数调用

JavaScript 中只有一种方法来完成函数调用，即在函数后紧临函数调用运算符“( )”。这个运算符被解释成两个含义：

- 使函数得以执行，并且，当函数执行时。
- 从左至右运算并传入“( )”内的参数序列。

这里所说的“函数”，包括普通的、类型值（即 `typeof` 值）为 `function` 的函数，也包括创建自 `Function` 类的函数对象。也就是说，函数调用运算符作用于以下两个变量的效果是一致的：

```
var func_1 = function() { };
var func_2 = new Function('');

// 调用函数 1
func_1();

// 调用函数 2
func_2();
```

如果该运算符之前既非上述两种之一（函数直接量与 `func_1` 是类同的），也非它们的引用，则函数调用运算将会出错，这会触发一个运行期异常[11]。

### 2.3.7 特殊作用的运算符

有些运算符不直接产生运算效果，而是用于影响运算效果，这一类运算符的操作对象通常是“表达式”，而非“表达式的值”。另外的一些运算符不直接针对变量的值运算，而是针对变量运算（如 `typeof` 等）。这些特殊作用的运算符见表 2-15。

[11] 不过这还与具体的宿主系统或引擎有关系，例如，WSH 或 IE 中可以在某些 ActiveX 对象后面使用函数调用运算符，而在 SpiderMonkey JavaScript 中，也可以在正则表达式对象后面使用它。

表 2-15 特殊作用的运算符

| 目标 | 运算符 | 作用 | 备注 |
|---|---|---|---|
| 运算元 | typeof | 返回表示数据类型的字符串 | |
| | instanceof | 返回继承关系 | 参见:<br>2.5 面向对象编程的语法概要 |
| | in | 返回成员关系 | |
| | delete | 删除成员 | |
| 表达式 | void | 避免表达式返回值 | 使表达式总是返回值 undefined |
| | ? : | 按条件执行两个表达式之一 | 也称“三元（三目）条件运算符” |
| | () | 表达式分组和调整运算次序 | 也称“优先级运算符” |
| | , | 表达式顺序地连续执行 | 也称“多重求值”或“逗号运算符” |

注意这里的 typeof 是运算符，而不是语句的语法关键字。之所以说它是特殊作用的运算符，是因为在其他运算符中，变量是以其值参与运算的。例如，下面的表达式中，变量 N 是以值 13 参与运算的：

```
var N = 13;
alert( N * 2 );
```

而 typeof 运算符并不访问变量的值，而是取值的类型信息。例如，下面的代码显示值的类型，而非变量 N 所在的存储单元中的值。

```
// 续上例
alert( typeof N );
```

其他 4 个运算符则面向表达式进行运算。其特殊性在于：普通运算符对表达式求值以获得结果，而这 4 个运算符并不直接对表达式进行求值运算，而是通过操作表达式去影响运算结果。例如，void 运算符，它对其后的表达式的影响就是：避免产生结果。

尽管它们是以“表达式”为目标，但语义上它们仍然是运算符。因此它们与运算对象（表达式）结合的结果仍然是表达式（而非语句）。例如，图 2-2 中的代码是表达式。

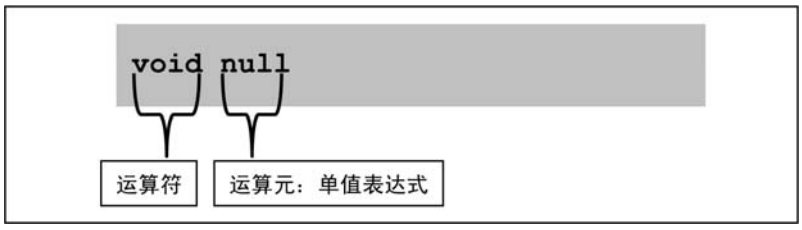


图 2-2 运算与运算元组合的结果总是表达式

但运算符的后面不能是语句（很显然，这是语法规则），所以下面的代码是不合法的：

```
// 用{}表示的复合语句不能作为 void 的运算对象
void {
  // ...
}
```

### 2.3.8 运算优先级

上面的例子存在一个问题。既然 void 运算的对象是表达式，且 JavaScript 允许单值表达式，那么这样的代码中：

```
void 1+2
```

void 运算的对象到底是“1”这个单值表达式，还是“1+2”这个算术表达式呢？

如果是第一种情况，那么结果如图 2-3 所示。

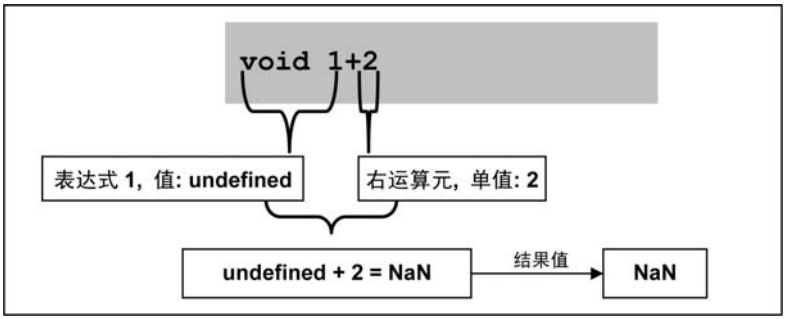


图 2-3 上述表达式的第一种解析

如果是第二种情况，则结果如图 2-4 所示。

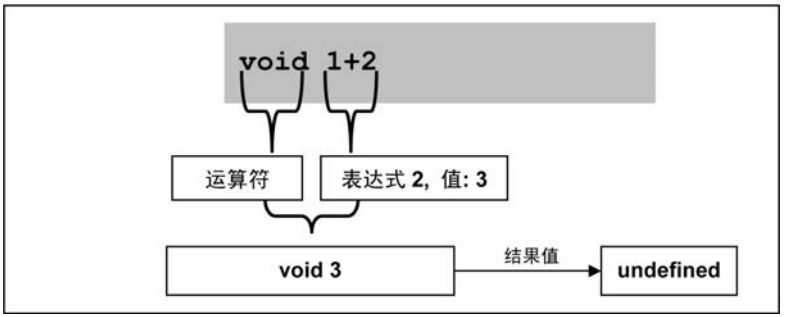


图 2-4 上述表达式的第二种解析

这就涉及两个例子中表达式 1 的运算符“void”和表达式 2 中的运算符“+”的优先级的问题了：谁的运算优先级更高，则先以该运算符来构成表达式并完成运算。在 JavaScript 中，该优先级顺序见表 2-16（序数越小，越优先运算）。

表 2-16 JavaScript 中运算符的优先级

| 序号 | 运算符 | 描述 |
|---|---|---|
| 1 | . [] () | 对象成员存取、数组下标、函数调用等 |
| 2 | ++ — - ~ ! delete new typeof void | 一元运算符等 |
| 3 | * / % | 乘法、除法、取模 |

续表

| 序号 | 运算符 | 描述 |
|---|---|---|
| 4 | + - + | 加法、减法、字符串连接 |
| 5 | << >> >>> | 移位 |
| 6 | < <= > >= instanceof | 序列检测、instanceof |
| 7 | == != === !== | 等值检测 |
| 8 | & | 按位与 |
| 9 | ^ | 按位异或 |
| 10 | \| | 按位或 |
| 11 | && | 逻辑与 |
| 12 | \|\| | 逻辑或 |
| 13 | ?: | 条件 |
| 14 | = oP= | 赋值、运算赋值 |
| 15 | , | 多重求值 |

通过该优先级表可知：由于“void”运算符高于“+”运算符，因此应该以上述的第一种情况进行运算，其结果值为 NaN。

因此如果希望以第二种情况进行运算（事实上在我写下这个例子之前，我认为是以这种情况运算的），就需要使用强制运算符“( )”来改变执行（优先级别的）顺序。下面的代码实现第二种情况的运算效果：

```
void (1 + 2)
```

正如我们留意到的：在“void”与“( )”参与的这几个运算中，运算符并不对值进行运算，而是对表达式（包括单值表达式）的运算效果进行影响。所以我们一再强调，“void”与“( )”等这些运算符的运算对象，是“表达式”而非变量/值运算元。

类同的，我们看到下面两个运算符（“? :”和“,”）也都用于影响表达式运算的效果（“2.7.5 逗号‘,’的二义性”一节将对示例 2 做更多的说明）。

```
/**
 * 示例 1: 运算符"?:"用于条件化地运算表达式
 */

// 显示表达式 100+20 的值
alert( true ? 100+20 : 100-20 )

/**
 * 示例 2: 三个表达式连续运算求值，返回最后一个表达式的值
 */

// 显示最后表达式的值"value: 240"
var i = 100;
alert( (i+=20, i*=2, 'value: '+i) );
```

# 2.4 JavaScript 的语法：语句

整个的 JavaScript 代码都是由语句构成的。语句表明执行过程中的流程、限定与约定，形式上可以是单行语句，或者由一对大括号“{ }”括起来的复合语句。在语法描述中，复合语句可以整体作为一个单行语句处理。

下面两个原则，有助于了解 JavaScript 的“语句”的定义：

- 语句由语法分隔符“;（分号）”来分隔（注意，它不是运算符）。
- （除空语句、声明语句，以及控制子句之外）语句存在返回值，该值由执行中的最后一个子句/表达式的值决定。

前面的章节中，我们已经讲述过的语句有：

- 赋值语句，使用“等号（=）”赋值运算符。
- 变量声明语句，使用 var 关键字开始一个变量声明。
- 标签声明语句，使用“identifier: statement”的语法开始一个标签声明。

当语句位于以下几种情况之一时，也可以省略分号。

（1）一个文本行或整个文本文件的末尾。

（2）在语法分隔符之前（如复合语句的大括号“}”）。

（3）在复合语句的大括号“}”之后。

关于这三点，JavaScript 的创始者的原始意图是为了更好地容错。当然，一部分原因也在于这符合其他一些语言的惯例。

除了前面讲述过的标签声明之外，下面我们还将补充更多的有关赋值与变量声明语句的内容。本章节关注的其他语句见表 2-17。

表 2-17 JavaScript 中的语句

| 类型 | 子类型 | 语法 |
|---|---|---|
| 声明语句 | 变量声明语句 | **Var**(*注 1) *variable1* [= *v1*] [, *variable2* [=*v2*],…]; |
| | 标签声明语句 | *labelname* : *statements*; |
| | 函数声明语句 | **function** *functionName() { … }* |
| 表达式语句 | 变量赋值语句 | *variable* = *value*; |
| | 函数调用语句 | *foo*(); |
| | 属性赋值语句 | *object.property* = *value*; |
| | 方法调用语句 | *object.method*(); |

续表

| 类型 | 子类型 | 语法 |
| --- | --- | --- |
| 分支语句 | 条件分支语句 | **if** (*condition*)<br>  *statement1*<br>[**else**<br>  *statement2*]; |
| | 多重分支语句 | **switch** (*expression*) {<br>  case *label* :<br>  *statementlist*<br>  case *label* :<br>  *statementlist*<br>  ...<br>  default :<br>  *statementlist*<br>}; |
| 循环语句 | for | **for** ([**var** ]*initialization*; *test*; *increment*)<br>  *statements*; |
| | For...in | **for** ([**var** ]*variable* **in** <*object* \| *Array*>)<br>  *statements*; |
| | while | **while** (*expression*)<br>  *statements*; |
| | Do...while | **do**<br>  *statement*<br>**while** (*expression*); |
| 控制结构 | 继续执行子句 | **continue** [*label*]; |
| | 中断执行子句 | **break** [*label*]; |
| | 函数返回子句 | **return** [*expression*]; |
| | 异常触发语句 | **throw** *exception*; |
| | 异常捕获与处理 | **try** {<br>  *tryStatements*<br>}<br>**catch** (*exception*) {<br>  *catchStatements*<br>}<br>**finally** {<br>  *finallyStatements*<br>}; |
| 其他 | 空语句 | ; |
| | with 语句 | **with** (*object*)<br>  *statements*; |

*注 1：语法描述中加粗的为语法标识符/关键字，加方括号的为可选的语法部分，加尖括号的为必选的语法部分，“|”表示所列项中选一。

## 2.4.1 表达式语句

为什么会存在“表达式语句”这样的概念呢？因为我们对编程语言术语的“表达式”的约定是：由运算数和操作符构成，并运算产生结果的语法结构。那么很显然，下面的代码就是一个表达式：

```
1+2+3
```

接下来的另外一项约定是：程序是由语句构成的，语句是则由“；（分号）”分隔的句子或命令。那么如果在表达式后面加上一个“；”分隔符，JavaScript 又如何理解呢？

```
1+2+3;
```

这就被称为“表达式语句”。它表明“只有表达式，而没有其他语法元素的语句”。

在 JavaScript 中，许多语法与语义其实最终都是由“表达式语句”来实现的。例如赋值、函数调用，以及我们在其他高级语言中常见的“调用对象方法”。

我们前面说过，语句的返回值由最后的一个子句或表达式的值决定。因此，“表达式语句”的值，就是该表达式运算的结果——即使不返回值（严格说来并没有“不返回值”，而是返回 `undefined` 值），也可以参与后续运算。

#### 2.4.1.1 一般表达式语句

对于一个表达式，你可以“计算值，并参与运算”，例如：

```
v * (1+2+3)
```

由于“`1+2+3`”是表达式，而括号“`( )`”作为运算符时，是指强制运算并求值。那么“`(1+2+3)`”就是计算值，而后参与和变量“`v`”的求乘积运算。

或者，你也可以计算但不返回值，例如，下面的表达式就经过了 4 次运算（先强制运算，再两次求和，最后忽略返回值）并返回 undefined：

```
void (1+2+3)
```

再或者，你也可以用 `eval()`函数来执行一行字符串（如果它是表达式），例如：

```
eval('1+2+3')
```

这种用法在一些语法分析，或者动态执行语句时会有实用价值。由于 `eval()`实际上也是一个运算，所以这项表达式其实完成了三次运算（`eval` 函数调用、两次求和）并返回值。

所以上述几种情况，其实都是在运算表达式。但你也可以在表达式末尾加一个“；”，表明这里完成了一个表达式语句：运算这个表达式语句，并且返回语句的值。例如：

```
v * (1+2+3);
void (1+2+3);
1+2+3;
```

事实上，JavaScript 承认单值表达式。尽管这个表达式也许不能表明任何确定的含义，但某些时候它的确有用，所以同样也存在单值语句。例如：

```
2;
```

由于“`2`”是一个单值，也可以理解为一个单值表达式，因此“`2;`”就变成了单值表

达式语句。这个语句在远程读取数据时可能会有用，因为你不能确保读到本机的是一个单独的数还是一个数组，而下面的代码可能给你提供了处理的机会：

```
// 远程数据
// returnValue = '2;';

// 取远程数据到本地
var remoteMetaData = ajax.Get(your_url);
var remoteData = eval(remoteMetaData);

switch (typeof remoteData) {
  // …
}
```

我们发现上面的代码中，即使远程数据是“'2;'”（亦即是上例中的 returnValue），程序仍然能够正常地处理。这是因为“2;”被理解为语句，并有效地执行了。

我们也顺道讲一下空语句。上面这个例子中，如果单值表达式“2”也没有了，只剩下了一个分号“;”又会如何呢？在 JavaScript 中，这就被理解为空语句。空语句、变量声明、函数声明以及控制子句都不产生返回值，因此当我们试图用 eval()执行一批语句并返回结果值时，这些语句都将被忽略，然后返回“最后执行到的、有返回值的”那条语句的值。例如下面的代码将返回结果值“3”：

```
eval('1+2;var x=5;;;;;function f() {};')
```

void 运算并没有类似的效果。因此下面的代码将返回 undefined：

```
eval('1+2;var x=5;;function f() {};void 0')
```

空语句的另一种应用情况是写空循环，或者空分支。例如：

```
var value = 100;

// 使用空语句的空循环
while (value--);  // <- 这里存在一个空的循环

if (value > 0);   // <- 这里有一个空的 then 分支
else {
  // ...
}
```

在代码中使用空语句时，一定要添加准确的注释，否则代码回顾（review）时将无法清晰地理解使用该技术的意图。

最后要强调一点：eval()函数总是执行语句[12]。即使传入下面这样的代码：

```
eval('1+2+3')    // 是试图“运算表达式”并返回表达式结果吗?
```

在语法逻辑上，它也是被当成语句处理的。因为还存在另一条规则：换行符和文本

[12] eval()事实上是将字符串作为语句处理的——它也是获得“语句”的结果值并使之可以参与后续运算的唯一方法。关于这些内容，将在第 5 章进一步讲述。

结束符的前面可以没有分号“;”，所以上面这行代码仍然是执行语句并返回语句的结果值。

#### 2.4.1.2 赋值语句与隐式的变量声明

赋值语句也是典型的表达式语句，例如，图 2-5 的赋值表达式：

```
str = 'test string'
```

它一方面可以继续参与运算，例如：

```
str2 = 'this is a ' + (str = 'test string')
```

另一方面，也可以直接加上一个语句结束符“;”，以表明这是一个“表达式语句”，如图 2-5 所示。

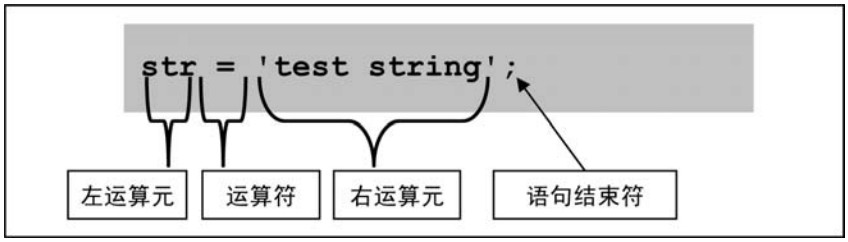


**图 2-5 赋值表达式语句**

所以在 JavaScript 中，“赋值语句”其实是“赋值表达式运算”的一种效果。这与其他语言对“赋值语句”的理解并不一致。

赋值表达式（以及赋值语句）具有声明一个变量的隐式效果：一个变量（标识符）在赋值前未被声明，则脚本会首先隐式地声明该变量，然后完成赋值运算。这种情况下，隐式声明的变量总是全局变量——因此它也被视为局部变量“泄露”到了全局。

#### 2.4.1.3 （显式的）变量声明语句

JavaScript 中可以使用 `var` 关键字显式地声明变量。显式声明时可以为变量赋一个初值；如果不赋初值，则该变量默认值为 `undefined`。

然而需要注意的是，显式声明语句中的 `var` 是语句语法符号，而不是运算符。该语句的语法是：

```
var variable1 [ = value1 ] [, variable2 [ = value2], …]
```

所以下面这个语句中的“=（等号）”（见图 2-6），其实是语法分隔符，而非运算符。例如：

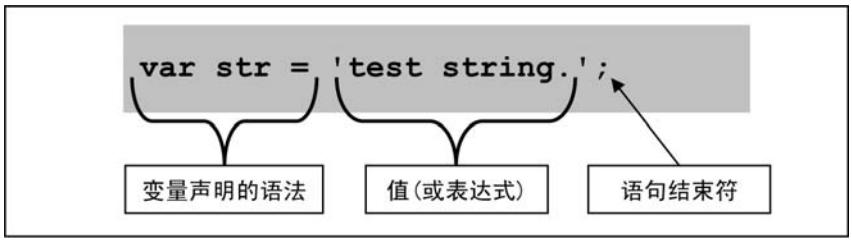


图 2-6 变量声明语句中的“=”号是语法分隔符

出于语法的设定，我们显然也可以不使用“=”这个语法分隔符（或标识符）来指明初始赋值。那么此时该变量声明语句就如图 2-7 所示。

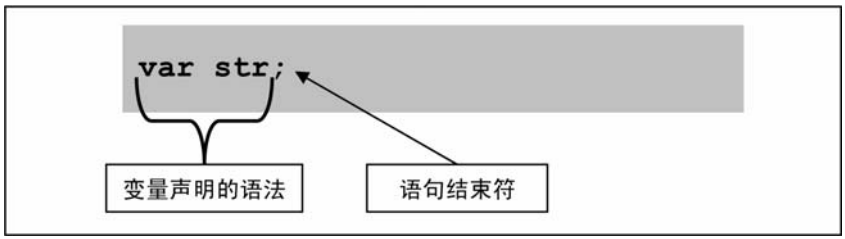


图 2-7 省略变量声明语句中的“=”号的情况

赋值表达式与此相比就有明显的区别：它至少要包括一个赋值运算符而不能省略。赋值运算符左侧一定是一个变量或对象属性，右侧可以是值或求值的表达式，语法表达如下：

```
语法: <变量>  赋值运算符 <值(或求值表达式)>
示例 1: str = 'this is ' + 'sample.'
示例 2: str += 'this is ' + 'sample.'
```

由于赋值表达式也可以使用“+=”这一类运算符，因此在上面的语法说明中，没有直接使用“等号（=）”，而是使用中文的“赋值运算符”。接下来，如果我们将“+=”放到显式变量声明中，就会是错误的语法了：

```
// 错误的语法
var str += 'test!';
```

由此可见，变量声明语句与赋值表达式存在根本的不同：这里的“=”是语句的语法分隔符，而不是“赋值运算符”，因此并不能替代成“+=”（或其他复合赋值运算符）。

最后的一点说明，是使用 var 来声明变量的语法，还可以在如下两种情况下使用：

```
// 1. 在 for 循环中声明变量
for (var i=0; i<10; i++) {
  // ...
}

// 2. 在 for…in 循环中声明变量
```

```
for (var prop in Object.prototype) {
  // ...
}
```

这两种情况下它相当于一个语句的子句。由于 JavaScript 的变量作用域只能达到函数一级（而非语句一级），因此这里声明变量，与在 `for` 或 `for…in` 语句之外声明变量没有什么区别。

#### 2.4.1.4 函数调用语句

JavaScript 中函数本身是一个变量/值，因此函数调用其实是一个表达式，如图 2-8 所示。

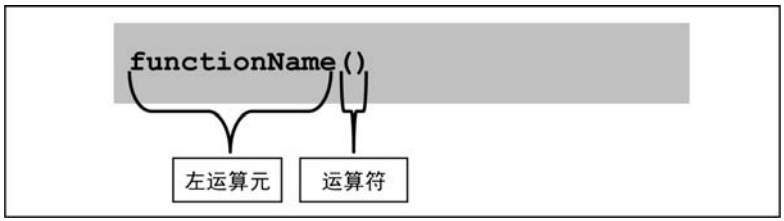


**图 2-8 函数调用其实是一个表达式**

所以，下面的代码就成了函数调用语句，它也是一个表达式语句：

```
 functionName();
```

在 JavaScript 中，具名函数可以使用上述方法直接调用，匿名函数可以通过引用变量调用，但没有引用的匿名函数怎么调用呢？下面的例子说明这三种情况：

```
// 示例 1: 具名函数直接调用
function foo() {
}
foo();

// 示例 2: 匿名函数通过引用来调用
fooRef = function() {
}
fooRef();

// 示例 3: 没有引用的匿名函数的调用方法(1)
(function() {
   // …
 }());

// 示例 4: 没有引用的匿名函数的调用方法(2)
(function() {
   // …
 })();

// 示例 5: 没有引用的匿名函数的调用方法(3)
void function() {
  // …
}();
```

示例 1、2 的用法比较常见。而示例 3、4、5 虽不太多见，但各有其用。

其中，示例 3 与示例 4 都用于“调用函数并返回值”。两种表达式都有三对括号，但含义各不相同，如图 2-9 所示（示例 3）。

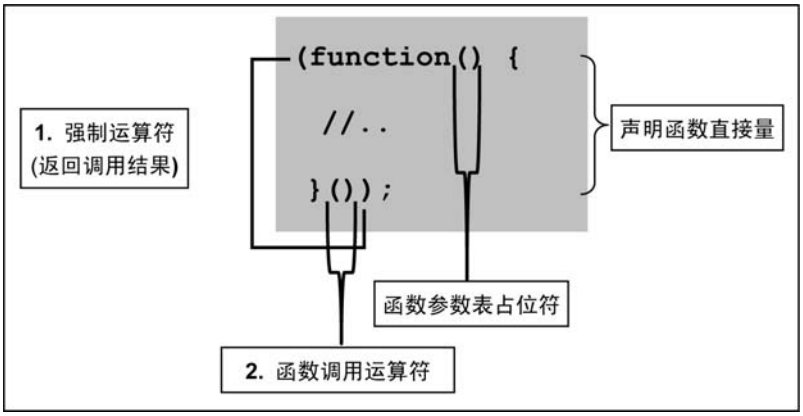


图 2-9 代码示例 3 的语法解析

图 2-10 是对示例 4 的说明。

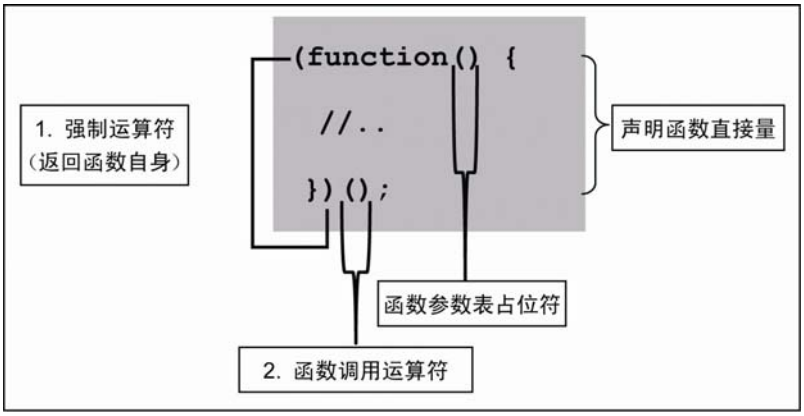


图 2-10 代码示例 4 的语法解析

我们看到示例 3 与示例 4 基本一致。但事实上两种表达式的运算过程略有不同：示例 3 中是用强制运算符使函数调用运算得以执行，示例 4 中则用强制运算符运算“函数直接量声明”这个表达式，并返回一个函数自身的引用，然后通过函数调用运算符“( )”来操作这个函数引用。

换而言之，“函数调用运算符( )”在示例 3 中作用于匿名函数本身，在示例 4 中却作用于一个运算的结果值。

最后的示例 5，则用于“调用函数并忽略返回值”。运算符 void 用于使其后的函数表达式执行运算。然而由此带来的问题是：如果不使用 void 与“( )”这两个运算符，而直接使用下面的代码，能否使函数表达式语句得到执行呢？

```
// 示例 6. 直接使用函数调用运算符"()"调用
function() {
  // …
}()

// 示例 7. 使用语句结束符";"来执行语句
function() {
  // …
}();
```

示例 6、7 看起来是正确的，起码用我们以前提到的所有知识来看，这两个示例中的代码均能被理解。但是事实上它们都不可执行。究其原因，则是因为它们无法通过脚本引擎的语法检测。在语法检测阶段，脚本引擎会认为下面的代码：

```
function() {
}

// 或
function foo() {
}
```

结果是函数声明，因此示例 6、7 中使用具名函数也是通不过语法检测的。正因为这里是函数声明，所以示例 6、7 的代码中位于函数后面的一对括号没有语法意义。这样一来，它们的代码无疑被语法解析成了：

```
// 示意：对示例 6 的语法解释
function() {
  // …
};
();

// 示意：对示例 7 的语法解释
// (略)
```

既然“function () {}”被作为完整的语法结构（函数声明语句）来解释，那么也就相当于其后已经存在语句结束符。因而“();”被独立成一行进行语法解释，而这显然是错误的语法。由此，我们将看到如图 2-11 所示的错误提示。

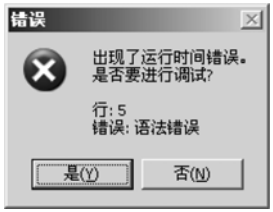


图 2-11 示例 6 和示例 7 在引擎的语法分析期中导致的错误提示

可见，这个“语法错误”事实上是针对于“( );”，而不是针对于前面的函数声明的。为了证明这一点，我们改写代码如下：

```
// 改写示例 6 的代码以通过语法解释
function() {
  // …
}(1,2)
```

这样一来你会发现语法检测通过了。因为语句被语法解释成了：

```
function() {
  // …
};
(1,2);
```

而最后这行代码被解释成如图 2-12 所示的内容。

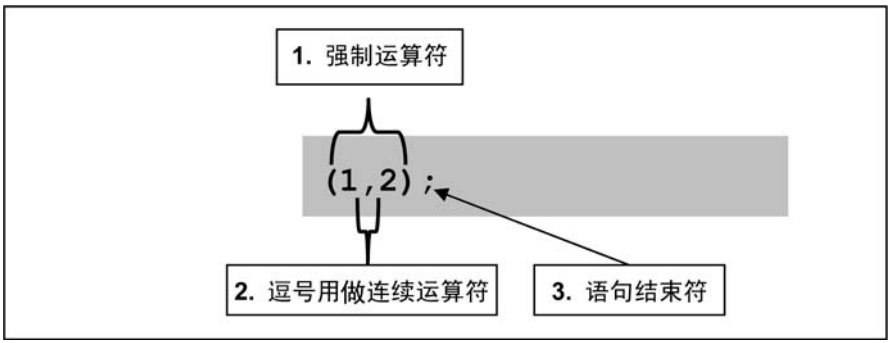


图 2-12 改写示例 6 的代码以通过语法解释

图 2-12 中的“1”和“2”被解释成了两个单值表达式——当然也可以是“(1)”这样的一个单值表达式，因此语法上就合法了。但重要的是，由于这段代码被解释成了一个函数直接量声明和一个表达式语句，因此它事实上并不能起到“执行函数并传入参数”的作用。简单地说，你不能指望用下面的代码来声明，并同时执行函数：

```
function foo() {
  // …
}(1,2);
```

如果你真的想在声明的时候执行一下该函数，那么请参考本小节开始的示例 3、4、5，用一个括号“( )”或 void 运算将函数声明变成“（直接量的）单值表达式”：

```
// 示例：声明时立即执行该函数(也可以用于匿名函数声明)
void function foo() {
  // …
}(1,2);
```

当引擎在解释这样的代码时，由于先识别到运算符 void，于是（默认地）将后面的匿名函数识别为操作数，因此需要讨论的仅仅是函数调用运算符“()”与 void 运算符之间的优先级问题。这样一来就避免了与“函数声明语句”混淆而发生歧义。

## 2.4.2 分支语句

基本上来说，JavaScript 的 `if` 语句跟 C 风格下的 `if` 语句并没有什么不同，所以事实上 `if` 分支语句并没有什么可多说的地方（同样的原因，《JavaScript 权威指南》一书只花了两页的篇幅来教导读者如何排布 `if` 语句的缩进格式）。与此相同的是，JavaScript 的 `switch` 分支语句也同于其他 C 风格的语言，但是，不幸的是，C 语言与 Pascal 等其他一些语言的多重分支语句之间，却存在着较大的差异。

因此在下面的小节中，我会略微阐述一下 `if` 和 `switch` 语句的基本语法。但对于 `switch` 语句，我会详细地叙述它的一些独特之处。

### 2.4.2.1 条件分支语句（if 语句）

`if` 语句语法描述如下：

```
if (condition)
  statement1
[else
  statement2];
```

当 `condition` 条件成立（值为 `true`）时执行语句 `statement1`；否则执行 `statement2`。语法描述中的方括号表明 `else` 子句可以省略，这时，如果 `condition` 条件不成立（值为 `false`），则什么也不做。

由于 `statement1`、`statement2` 在语法上表明是“语句”，因此事实上它们既可以是单行语句，也可以是复合语句。在本小节开始的部分，我们已经说过“在语法描述中，复合语句可以整体作为一个单行语句处理”。这表明下列代码中的大括号“`{ }`”是复合语句的语法符号，而并非（像一些人想的那样）是 `if` 语句的语法元素：

```
// 代码风格 1: if 语句中使用复合语句带来的效果
if (condition) {
  // …
}
else {
  // …
}
```

同样，我们也应该了解，`if...else if...`这样的格式，并非是“一种语法的变种”。只不过 `else` 子句中的“`statement2`”是一个新的、单行的 `if` 语句而已：

```
// 代码风格 2: 在 else 子句中，使用单行 if 语句带来的效果
if (condition1) {
  // ...
}
else if (condition2) {
  // ...
}
```

### 2.4.2.2 多重分支语句（switch 语句）

无论是解释 switch 语句，还是使用 switch 语句，都非常容易令人迷惑。其中的主要原因之一在于 switch 语句中的 break 子句的使用。

在 Pascal 风格的语言系统中，switch 是没有 break 这样的子句的。这虽然导致有些控制逻辑的代码会变得复杂，但也使得代码的结构化程度得以提高。而在 C 风格的语言系统中，由于存在了 break 子句，使得多重分支的流程出现了“例外”。因此虽然提供了灵活性，却也产生了类似于 GOTO 语句的副作用。

我们比较一下两种风格的 switch 语句（在 Pascal 中称为 case）：

```
(**
 * Pascal语言风格的多重分支语句
 *)
var i,j : integer;
// … (略去有 i,j 初值或相关运算的代码)

case ( i ) of
  100: begin

  end;

  200: begin

  end;

  else begin

  end;
end;
```

```
/**
 * C 语言风格(JavaScript)的多重分支语句
 */
var i, j;
// …(略去有 i,j 初值或相关运算的代码)

switch ( i ) {
  case 100: {
    break;
  }

  case 200: {
    break;
  }

  default: {
    break;
  }
}
```

可见除了一些关键字和语法符号的差异之外，二者并没有不同。但是因为 Pascal 中没有 break 语句，所以 begin…end 之间总是一个完整的语法块——关键字 end 起到了“结束该语句块（及其逻辑流程）”的作用。而对应的，在 C 语言中，你必须使用 break 子句来中止这个语句块的逻辑流程。

case 分支中的大括号“{ }”在这里只起到了标识一个复合语句的作用。而 switch 语句的“一批语句”是由该语句的语法：

```
switch (condition) {
  statements
}
```

来标识的（case 只用于标识这批语句的某个入口点），因此这时就不必用大括号来局部地开始和结束了（这样做没有逻辑意义）。所以在一般情况下，可以省却大括号而直接书写多行代码。例如：

```
switch ( i ) {
```

```
  case 100:
    j++;
    i += j;
    break;

  case 200:
    // …
}
```

但是，在 Pascal 风格的代码中，由于 `begin...end` 必须是一个完整的语法块，因此下面的代码必然会出现：

```
(**
 * Pascal 语法风格中没有 break 语句导致的问题
 *)
case ( i ) of
  100: begin
    i := i + 1;
    j := j + 1;
  end;

  200: begin
    j := j + 1;
  end;

  // …
end;
```

也就是说，在 `i` 值为 `100` 和 `200` 的处理中，可能有一些相同的代码行需要重复使用，然而由于 `begin...end` 决定了完整的代码块，因此只能出现冗余的代码。除非在这个多重分支语句前面/后面写条件分支代码或新的多重分支。例如：

```
(**
 * 解决方案一
 *)
case ( i ) of
  100, 200: begin
    if (i = 100) i := i + 1;
    j := j + 1;
  end;

  // ...
end;

(**
 * 解决方案二
 *)
case ( i ) of
  100, 200: begin
    j := j + 1;
  end;

  // ...
end;
if (i = 100) i := i + 1;
```

但 C 语言在代码风格上解决这个问题，使得两个分支复用同一个代码块成为可能。

上面的代码在 C 风格的语言中的解决方案如下：

```
/**
 * C 风格语言(JavaScript)的解决方案
 */
switch ( i ) {
  case 100 : i++;
  case 200 : j++;
}
```

也就是说，我们只需要不在语句“i++”后面写 break 子句，那么逻辑流程就会“漏”到下一行的“case 200”这个分支，从而达到了“复用多个分支的代码”的效果。

因为 break 改变了控制的流程，对使用一对大括号“{}”表示的结构化的代码块造成了语法伤害。因此包括在经典的 C 或 C++语法材料中，这种“漏掉 break 子句”技巧都是被有争议地、谨慎地进行描述的。更为严格的要求是：不要省略最后一个分支后的 break，以避免将来加入新的分支时遗忘掉一个 break 子句。

另一方面，在使用这种技巧的代码中也被明确要求：“对算法或省却 break 的原因做出备注”。所以，仍以上面的代码为例，一个较为良好的风格应当是：

```
/**
 * C 风格语言(JavaScript)的解决方案
 */
switch ( i ) {
  case 100 : i++;   // defer break;
  case 200 : j++;   // break omitted for end.
}
```

其中，第一个备注明确指出在这里的算法要求延迟（defer）进行 break；第二个注释则说明由于结束而省略 break。

最后，由于 break 是一个在循环和多重分支中都可以使用的子句，所以还存在更多的细节没有描述。关于这些细节，请阅读“2.4.4.2 break 子句”。

## 2.4.3 循环语句

JavaScript 的循环语句相对丰富——尽管您使用其中一个就可以替代其他循环语句。一般来讲，以下三种循环结构是开发人员所熟知的：

```
// for 循环(增量循环)
for ([var ]initialization; test; increment)
   statements;

// while 循环
while (expression)
   statements;

// do…while 循环
do
  statement
while (expression);
```

通常 while 与 do...while 中的循环条件（expression）都应当是有意义的，仅有在少数情况下，它被置为 true 以表示无限循环（当然在循环体内应使用 break 子句来中断）。例如：

```
while (true) …

//或
do
  …
while (true);
```

在很多情况下，上述用法仍可称为良好的结构化设计[13]，但是在循环体中的空语句却是不宜使用的。例如：

```
var i = 10;
while (alert(i), i--);
```

这除了展示 while 条件中可以使用连续运算（的技巧）之外毫无用处，它完全等价于：

```
// 方法一
var i = 10;
while (i) {
  alert(i);
  i--;
}

// 方法二
var i = 10;
do {
  alert(i);
}
while (--i)
```

与此类同的问题也会出现在 for 语句中：很多人习惯于将循环体塞在 for 语句的表达式中。例如：

```
for (var i = 10; i<10; alert(i), i--);
```

这样的用法对读代码的人来说会是一种灾难。因此，建议 for、while 与 do...while 等循环语句中只放置与循环条件相关的表达式运算（例如，同时处理多数组时使用的多个下标控制变量）。

除了上述三种循环之外，JavaScript 也支持一种用于对象成员列举的循环语句 for...in：

```
for ([var ]variable in <object | Array>)
   statements;
```

需要注意的是，从语法上来看它能够处理数组中的元素，但其实这不过是 JavaScript

[13] 但我不建议使用“for (;;) ...”来表示无限循环，这种用法大概只是出于炫耀的目的而被创造出来的。

将数组作为对象处理时的一种表面现象[14]。关于 `for...in` 语句的使用细节，我们将在“2.5.2.1 对象成员列举、存取和删除”中专门讲述。

## 2.4.4 流程控制：一般子句

### 2.4.4.1 标签声明

JavaScript 中的标签就是一个标识符。标签可以与变量重名而互不影响，因为它是另一种独立的语法元素（既不是变量，也不是类型），其作用是指示“标签化语句（labeled statement）”。

标签的声明很简单：一个标识符后面跟一个冒号“:”。你可以在任何一个语句前面加上这样的标签，以使得该语句被“标签化（labeled）”。例如：

```
this_is_a_label:
  myFunc();
```

除了单一的语句之外，标签也可以作用于由大括号表示的复合语句。例如：

```
label_statements: {
  var i = 1;
  while (i < 10) i++;
}
```

但是标签不能作用于注释语句。因此在下例中，标签实际作用于注释语句后面的一个语句（即 `if` 条件语句）：

```
my_label_2:
/*
  hello, world;
*/
if (true) {
  alert('hello, is a test!');
}
```

标签表示一个具有“语句块”含义的范围（但不是后文中提到的上下文环境、闭包，或者变量作用域等概念上的范围）。这个语句块的范围是指单一语句的开始到结束位置（分号或回车），或者成批语句的开始到结束位置（一对大括号）。图 2-13 可说明该范围。

[14] Core JavaScript 1.5 中约定了一种 `forEach` 语句语法，用以从一个对象中列举所有属性的值，它直到 SpiderMonkey JavaScript 1.6 以上版本才开始支持。但这不仅不包含在 ECMA 规范中，也不为 JScript 等其他 JavaScript 引擎所支持。（仅对数组的成员列举来说，）相比之下更值得称赞的一项扩展是 JavaScript 1.6 中的设计：为 `Array` 对象原型扩充 `forEach()`方法。最后，在 SpiderMonkey JavaScript 1.7 中，由于提出了“destructuring assignment”的概念，因此允许在 `for...in` 中同时列举对象的属性和属性的值。

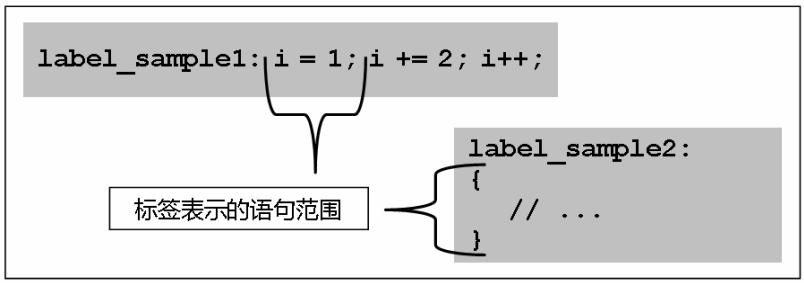


图 2-13 标签表示的语句范围

在 JavaScript 中，标签只能被 break 语句与 continue 语句所引用。前者表明停止语句执行，并跳转到 break 所指示的范围之外；后者表明停止当前循环，并跳转到 break 所指示的范围的开始。

我们已经被无数次地忠告“不要使用 GOTO 语句”，然而还是有一些语言保留了 GOTO 语句。事实上 JavaScript 语言也将这个关键字作为保留的语法关键字，但至今仍未启用。在除去了 GOTO 语句之外，程序系统中仍然会存在一些语句来改变程序的执行流程。例如本小节中将提及的 break 和 continue，以及 return 语句。

但是这些在语言中顽强地存活下来的语句，都不再像 GOTO 语句一样可以“无条件地任意跳转”。事实上它们总是受限于语句的上下文环境。例如 break 只能用于 for、while 等循环语句，switch 分支语句和标签化语句的内部，而 return 只能用于函数内部。因此，一些讲述 JavaScript 语言语法的书籍中会称它们为“子句”。本书在此也采用这种说法，表明它们“是上下文受限的”这一事实。

### 2.4.4.2 break 子句

通常，我们在 for、while 等循环中使用 break，表明停止一个最内层的循环；或在 switch 语句中跳出 switch 语句。例如：

```
/**
 * 在 for 循环中使用 break 的简单示例
 * ( 使 i=50,j=50 不被处理 )
 */for (var i=0; i<100; i++) {
  //…
  for (var j=0; j<100; j++) {
    if (i==50 && j==50) break;
    // …
  }
}

/**
 * 在 switch 中使用 break 的简单示例
```

```
 */
var chr = 'A';  // or 'B' and other...
switch (chr) {
  case null: break;
  case 'A':
  case 'B': break;
  default:
    chr = 'X';
    break;
}
alert(chr);
```

一部分开发人员并不理解 `default` 分支中的 `break` 有什么价值，因为在他们看来，这里使用 `break` 跳出 `switch`，与 `default` 分支直接运行到结束代码并没有什么区别。下例则说明 `break` 在 `default` 分支中的使用价值：

```
  // (参见上例, …)
  default:
    if (!isNaN(parseInt(chr))) break;
    chr = chr.toUpperCase();
}
```

默认情况下，`break` 子句作用于循环语句的最内层，或者整个 `switch` 语句，因此它不必特别地指定中断语句的范围。但 `break` 子句也具有一种扩展的语法，以指示它所作用的范围。该范围用声明过的标签来表示。例如：

```
break my_label;
```

这使得 `break` 子句不但可以使用在循环与条件分支内部，也可使用在标签化语句（labeled statement）的内部。例如：

```
/**
 * 显示输入字符串的末 10 个字符
 */
var str = prompt('please input a string', '12345678910');

my_label: {
  if (str && str.length < 10) {
    break my_label;
  }
  str = str.substr(str.length-10);
}

// other process…
alert(str);
```

这种情况下，`break` 子句后的 `my_label` 不能省略——尽管 `break` 位于 `my_label` 所表示的语句范围之内。因此，以下三种用法都将触发语法编译期的脚本异常：

```
my_label: {
  if (str && str.length < 10) {
    // 错误 1: 在标签化语句中使用 break 而不带 label
    break;
  }
  str = str.substr(str.length-10);
```

```
}

if (true) {
  // 错误 2: 在标签化语句的范围之外引用该标签
  break my_label;
}
else {
  // 错误 3: 在有效的范围(标签化语句、循环和 switch 分支)之外使用 break;
  break;
}
```

三种错误情况下的异常信息分别如图 2-14 所示。


（a） （b） （c）

图 2-14 三种 break 使用错误的异常信息

应该注意到：图 2-14（a）的错误与图 2-14（c）的错误的异常信息是一样的。由于语法上我们是可以在“循环之外使用 break 子句的”，因此事实上该提示信息所指的是“un-labeled break”——即在脚本引擎内部，会认为“不带标签的 break”仅能用于循环（和 switch 分支）的内部。

### 2.4.4.3 continue 子句

continue 仅对循环语句有意义，因此它只能作用于 for、for...in、while 和 do...while 这些语句的内部。在默认情况下，它表明停止当前循环并跳转到下一次循环迭代开始处运行。例如：

```
// 声明一个 "工人(Worker)"类
function Worker() {
  this.headSize = 10;
  this.lossHat = false;
  this.hat = null;
  this.name = 'anonymous';
}

// 声明工人使用的"帽子(Hat)"类
function Hat() {
  this.size = 12;
}

// 有三个工位(或更多)
works = new Array(3);
```

```
// 这里有一段业务逻辑，例如给每个工人发帽子，或者工作中丢掉帽子
// works[2] = new Worker();

// 现在检查每个工人的帽子，给没帽子的工人补发一个
for (var i=0; i<works.length; i++) {
  if (!works[i]) continue;
  if (!works[i].lossHat) continue;

  works[i].hat = new Hat(works[i].name);
  works[i].lossHat = flase;
}
```

我们看到 `continue` 使代码的结构变得简单了。我们可以用 `continue` 尽早地清理掉一些分支，这样在循环体的尾部，就只剩下符合条件的数据了。这样我们就可以在这里很轻松地写出“干净”的业务代码。

`continue` 后面也可以带一个标签，这时它表明从循环体内部中止，并继续到标签指示处开始执行。

如果这个标签指示的语句不是一个循环语句——即使是一个包括循环的复合语句，JavaScript 引擎也认为是一个语法错误。也就是说，`continue` 后面的标签，只能是对单个的循环语句有意义。例如：

```
// (..., 接上面的代码)

// 建立一个仓库
library = {};
library.hats = function(sex) {
  return [];  //<-- 这里应当返回仓库中的读性别适用的全部帽子的列表
}

breakToHere:
  for (var i=0; i < works.length; i++) {
    if (!works[i]) continue;
    if (!works[i].lossHat) continue;

    // 在仓库中为该工人挑选一顶大小合适的帽子
    var oldHats = library.hats(works.sex);
    for (var j=0; j < oldHats.length; j++) {
      if (oldHats[j] && (oldHats[j].size > works[i].headSize)) {
         works[i].hat = oldHats[j];
         works[i].lossHat = false;
         delete oldHats[j];
        // 挑选成功，跳到外层循环处理下一个工人
        continue breakToHere;
      }
    }
  }
```

这段代码是可以执行的，但是如果你在 `breakToHere` 后面加一对大括号（如下面的代码那样），脚本解释引擎就会认为出错了：

```
breakToHere : {
  for (var i=0; i < works.length; i++) {
     // ...
  }
}
```

因为 continue 不允许跳转到“当前/外层的单个循环语句的起始”之外的其他任何地方。

#### 2.4.4.4 return 子句

return 子句只能用在一个函数之内，且同一个函数之内允许存在多个 return。当函数被调用时，代码执行到第一个 return 子句则退出该函数，并返回 return 子句所指定的值；当 return 子句没有指定返回值时，该函数返回 undefined。例如：

```
1  function test(tag) {
2    if (!tag) {
3      return;
4    }
5
6    return tag.toString();
7  }
8  var v1 = test();
9  var v2 = test(1234);
```

这里的“第一个 return 子句”是指逻辑含义上的、第一个被执行到的 return 子句，而不是物理位置上的。例如在上面的代码中，第 8 行没有传入 tag 变量的值，因此将从第 3 行的 return 返回；而第 9 行调用 test()时，则从第 6 行的 return 返回。

当使用 void 运算调用函数时，return 子句指定的返回值将被忽略。这使得下面的代码返回 undefined：

```
function test2(tag) {
  return true;
}
var v3 = void test2();
```

最后一行代码使得 v3 赋值为 undefined。但这是 void 运算的结果，而并不是 return 子句没有返回值。两者看起来效果一致，但本质却不同。

当执行函数的逻辑过程中没有遇到 return 子句时，函数将会执行到最后一条语句（末尾的大括号处），并返回 undefined 值。

### 2.4.5 流程控制：异常

异常是比前面所提到的其他子句复杂许多的一种流程控制逻辑。异常与一般子句存在着本质的不同：一般流程控制子句作用于语句块的内部，并且是编程人员可预知、可

控制的一种流程控制逻辑；而异常正好反过来，它作用于一个语句块的全局，处理该语句块中不可以预知、不可控制的流程逻辑。

结构化异常处理的语法结构如下：

```
try {
  tryStatements
}
catch (exception) {
  catchStatements
}
finally {
  finallyStatements
};
```

该处理机制被分为三个部分（上述语法只说明了其中两个部分），包括：

- 触发异常，使用 `throw` 语句可以在任意位置触发异常或由引擎内部在执行过程中触发异常。
- 捕获异常，使用 `try…catch` 语句可以（在形式上表明）捕获一个代码块中可能发生的异常。
- 结束处理[15]，使用 `try…finally` 语句可以无视指定代码块中发生的异常，确保 `finally` 语句块中的代码总是被执行。

在上述语法中，`finally{…}`块是可选的，但如果存在同一级别的 `catch() {…}`块，则 `finally` 块必须位于 `catch` 块之后。在执行上，`catch` 块是先于 `finally` 块的。但如果在 `finally` 执行中存在一个未被处理的异常——例如在 `finally` 之前没有 `catch` 处理块，或者在 `catch`、`finally` 块处理中又触发了异常，那么这个异常会被抛出到更外一层的 `try…catch/finally` 中处理。

`finally{…}`语句块的一个重要之处在于它“总是在 `try/catch` 块退出之前被执行”。这一过程中常常被忽略的情况包括[16]：

```
// 在(函数内部的)try 块中使用 return 时, finally 块中的代码仍是在 return 子句前执行的
try {
  // …
  return;
}
finally {
  …
}

// 在(标签化语句的)try 块中使用 break 时, finally 块中的代码仍是在 break 子句前执行的
aLabel:
try {
  // …
  break aLabel;
```

[15] 很难给 `finally{…}`语句块一个合适的命名，这里的命名主要强调 `finally` 块的一般性作用。
[16] 类似情况还包括 `continue` 子句。在《JavaScript 权威指南》中包含了一个这样的例子，读者请自行参考。

```
}
finally {
  …
}
```

最后我们讨论 throw 语句。这个语句既可以作用于上述语法的 try {…}块，也可以作用于 catch {…}与 finally {…}块。无论是在哪个位置使用，它总是表明触发一个异常并终止其后的代码执行。throw 语句后“应当是”一个错误对象：Error()构造器的实例或通过“catch(exception)”子句中捕获到的异常 exception。之所以说“应当是”，是因为 throw 语句其实可以将任何对象 / 值作为异常抛出。

有趣的是，如果 throw 语句位于一个 finally{…}语句块中，那么在它之后的语句也不能被执行——这意味着 finally{…}语句中的代码“不一定”能被完整地执行。同样的道理，即使不是使用 throw 语句显式地触发异常，在 finally{…}块中出现的任何执行期异常也能中止其后的代码执行。

因此，对于开发者来说，应尽可能保证 finally{…}语句块中的代码都能安全、无异常地执行。如果不能确信这一点，那么应当将那些不安全的代码移入到 try{…}块中。

## 2.5 面向对象编程的语法概要

JavaScript 中，面向对象框架看起来有一套自己的语法规则，但其实很多规则都是演化自此前所述的“声明、（表达式）运算、语句”这一基本体系。例如对象属性的存取与方法调用，就是一种简单的表达式语句。

在本书的“第 3 章 JavaScript 的非函数式语言特性”中我们还将更加详细地讲述面向对象编程，因此在接下来几个小节中，我们将仅仅概述在 JavaScript 中编程的基本语法。这些为面向对象设计的语法元素见表 2-18。

表 2-18 JavaScript 中为面向对象设计的语法元素

| 类型 | 语法元素 | 语法 | 含义 | 注 |
|---|---|---|---|---|
| （直接量） | { … } | **{** *propertyName_1: expression_1,* … *propertyName_n*: expression_n **}** | （一般）对象直接量 | （*注 1） |
| | […] | **[** *element_1,* … *element_n* **]** | 数组对象直接量 | |
| | / … / … | */expression pattern/flags* | 正则表达式对象直接量 | （*注 2） |

续表

| 类型 | 语法元素 | 语法 | 含义 | 注 |
|---|---|---|---|---|
| 运算符 | new | **new** *constructor*[**(***arguments***)**] | 创建指定类的对象实例 | |
| | in | *propertyName* **in** *object* | 检查对象属性 | |
| | instanceof | *objectInstance* **instanceof** *constructor* | 检查变量是否指定类的实例 | |
| | delete | **delete** *expression* | 删除实例属性 | |
| | . | *object.Identifier* | 存取对象成员（属性、方法） | |
| | [ ] | *object***[***string_expression***]** | | |
| 语句 | for … in | **for** ([**var** ]*variable* **in** <*object* \| *Array*>) *statements*; | 列举对象成员 | |
| | with | **with** (*object*) *statements*; | 设定语句默认对象 | |

*注 1：各表达式和语法元素可以写在同一行，或者不同的行。此处使用这样的代码格式只为了清晰地展示语法。

*注 2：正则表达式的立即值必须写在同一行。

## 2.5.1 对象直接量声明与实例创建

一般情况下，你可以使用直接量声明对象，或者用 new 关键字创建新的对象实例。此外，你也可以通过宿主程序来添加自己的构造器，并用 new 关键字来创建它的对象实例。更加特殊的方法是，你可以直接在宿主环境中创建对象实例，并让某些引擎的脚本代码中去持有并使用它[17]。

### 2.5.1.1 使用构造器创建对象实例

构造器是“创建和初始化”对象的一般性方法。但总的来说，在 JavaScript 中有三种方法会涉及“初始化对象实例”的问题：

- 其一，是通过在构造器中利用 this 引用来初始化。
- 其二，是通过构造原型实例来初始化。
- 其三，是通过 Object.create()并使用属性描述符的方式来构建对象并初始化。

[17] 在 JScript 中可以通过 `ActiveXObject()`来创建宿主或操作系统环境中的对象，也可以在宿主或操作系统中创建对象并注册到 ROT（Running Object Table），最后由 JavaScript 通过 `GetObject()`来取用。不过这些已经超出了本书讨论话题的范围。

在这里我们只讲述第一种方法[18]。第二、三种方法涉及原型继承的问题，并且第三种方法是在 ES5 中被加入规范的，因此它们被安排到“第 3 章 JavaScript 的非函数式语言特性”中去讨论。

第一种方法将借助 new 运算让构造器产生对象实例。new 运算的语法规则为：

```
obj = new constructor[(arguments)];
```

其中“构造器（constructor）”其实就是一个普通的函数，或者是 JavaScript 内置的或宿主程序扩展的构造器。下面的示例简要说明由用户声明的构造器创建实例的方法：

```
// 可以被对象方法引用的外部函数
function getValue() {
  return this.value;
}

// 构造器(函数)
function MyObject() {
  this.name = 'Object1';
  this.value = 123;
  this.getName = function() {
    return this.name;
  }
  this.getValue = getValue;
}

// 使用 new 运算符，实现实例创建
var aObject = new MyObject();
```

在构造器函数执行过程中，JavaScript 将传入 new 运算所产生的实例，并以该实例作为函数上下文环境中的 this 对象引用。这样一来，在构造器函数内部，就可以通过“修改或添加 this 对象引用的成员”来完成对象构造阶段的“初始化对象实例”——就像我们在上例中声明的构造器 MyObject( )一样。

语法中，当参数表为空时与没有参数表是一致的。因此下面两行代码是等义的：

```
// 示例 1：下面两行代码等义
obj = new constructor;
obj = new constructor();
```

但是，我们不能认为 constructor 后面的括号是函数调用的括号。因为如果这是函数调用的括号，那么下面的代码就应该是合理的了（但事实上，这行代码对于构造器来说是错误的用法）：

```
// 示例 2：错误的代码
obj = new (constructor());
```

所以，不能错误地认为：new 运算是“产生对象实例，并调用 constructor 函数”

[18] 这是一种沿袭自 JavaScript 1.0 的、相对有些“过时的”对象构建方法。尽管在 ECMAScript 规范中中仍旧明确地支持，但在开发实践中却并不太“时兴”。但它并非无用。相反的，它在框架的实现代码中常常出现，用于构建基本的对象模型。

——尽管看起来很像是这样。

但是，如果我们的确不打算让 new 后面的函数作为构造器，而只是作为函数使用，那么可以使用下面的代码来实现一些特殊的效果：

```
// 示例 3: 将 foo()视为普通函数
function foo() {
  var data = this;  // <-- 这里暂存了 this,你当然也可以不暂存它
  return {};
  // or
  // return new Object();
}
obj = new foo();
```

在这个例子中，最终 obj 也会被赋值为一个对象实例。但这个实例并不是 new 运算产生的，而是 foo()函数中返回的。注意，使用这种方法的时候，只能通过 return 返回一个引用类型的直接量或对象实例，但不能是值类型的直接量——例如不能是 true、'abc'之类，当用户试图返回这种值类型数据时，脚本引擎会忽略掉它们，仍然使用原来的 this 引用。

此外，如上例所示的，foo()函数内部保留的 new 运算产生的实例，也可以用做其他的用途，例如保存私有数据。

将 foo()视为普通函数的方法，虽然带来了一些特殊效果，但是也破坏了对象的继承链。关于这种技术对继承链产生的影响，在“3.3 JavaScript 中的原型继承”中讲述。

构造器中“暂存的 this 实例的一个引用”是一个有用的技巧——尽管你也可以不暂存它。

最后，我们回到“将 new 右边的运算元视为构造器处理”的情况上来。由于“构造器”在 JavaScript 中是由函数（直接量或对象）来承担的，因此如果改变运算顺序，事实上也可以使用“返回函数的运算”来得到构造器。例如：

```
// 示例 4: 用表达式(运算)作为 new 的运算元
function getClass(name) {
  switch (name) {
    case 'string': return String;
    case 'array': return Array;
    default: return Object;
  }
}
obj = new (getClass());
```

在示例 3、4 中，我们没有将 foo()和 getClass()作为构造器来使用，但我们却用到了 new 运算的特性——并且，令人惊奇地产生了一些效果。

这里已经涉及 JavaScript 的函数式特性。因此，更多的内容我们将放到“第 4 章 JavaScript 的函数式语言特性”中讲述。

#### 2.5.1.2 对象直接量声明

对象直接量声明比用构造函数来得简单方便。它的基本语法是：

```
obj = { propertyName: expression[, ...] }
```

示例如下：

```
// 示例：一些已声明过的变量或标识符
function getValue() {
  // ...
}
// 对象直接量声明
var aObject = {
  name: 'Object1',
  value: 123,
  getName: function() {
    return this.name;
  },
  getValue: getValue
}
```

这里的名字（propertyName）可以用字符串来表示，也可以只是一个标识符。因为这是语句语法，所以我们通常都用标识符，只有在特殊的情况下，才使用它的字符串格式。这些“特殊的情况”通常是指：

- 使用的标识符不满足 JavaScript 对标识符的规则。
- 特殊的、强调的属性名。

例如你可以用“abc.def”来做属性名，但这并不是一个合法的标识符；也可以用数字“1”来做属性名，同样它也不是合法的标识符。这时，就可以像下面这样声明：

```
obj = {
  'abc.def': 123,
  '1': 456
};
```

对于“名字：值”对的右边，可以是任何类型的立即值，也可以是任何表达式运算的结果。因此下面的声明也是合法的：

```
// 示例 1：嵌套的对象立即值声明
obj = {
  'obj2': {
    name: 'MyObject2',
    value: 1234
  }
};

// 示例 2：使用函数(表达式)的返回值
function getValue() {
  return 100;
}

obj = {
  name: 'MyObject3',
  value: getValue()
}
```

如果一对大括号“{ }”中间没有任何“名称：值”对，那么将得到一个“空的对象”。在某些时候，我会称其为“干净的对象”，关于这一点，请参阅“5.6.3 干净的对象”。

空字符串和点号也可以作为属性名——尽管可能没有什么实用性。此外，JavaScript 对数字属性名还有一些特殊使用的情况。在后面的章节中会独立出来再讲。

某些类的对象实例也可以使用直接量的方式声明。具体来说，这包括数组（Array）与正则表达式（RegExp）。另外，空对象也以直接量“null”的形式存在（当然，你也可以把它看成常量，或者语法关键字）。下面的代码简要地说明了这三种直接量的声明：

```
// 示例 1: 数组对象
var arrayObject = [1, 'abcd', true, undefined];

// 示例 2: 正则表达式对象
var regexpObject = /^a?/gi;

// 示例 3: 空对象
var nullObject = null;
```

为什么不把字符串、布尔值、数字的直接量也放在本小节中来说明呢？因为这三种直接量声明的变量，其类型（typeof）是基本类型。它们表现得像一个对应类的实例，是因为使用了“包装类”的技术。关于这一点，请参阅“5.5 包装类：面向对象的妥协”。

#### 2.5.1.3 数组直接量声明

JavaScript 可以用直接量声明一个数组，而且这个数组可以是异质的（数组元素的类型可以不同）、交错的（数组元素可以是不同维度的数组）。数组的交错性使它看起来像是“多维的”，但事实上不过是“数组的数组”这种嵌套特性。换成最直接的表达式方式，我们并不能用类似于：

```
arr = new Array(10, 10);
```

这样的方式来得到一个每维 10 个分量的二维数组（该语法在 JavaScript 中会得到包含两个元素的一个一维数组），但可以用：

```
arr = [[1,2],[3,4]];
```

这样的方法来得到一个交错的数组——数组的数组，尽管它的大小（以及表达的数学含义）也是 2*2 的，但在数据结构的本质上并不具备某些多维数组的特性[19]。

从表面上来看，你也可以在 JavaScript 的数组中使用如下语法：

```
arr[1,2,3]
```

但这种语法并不返回某个多维数组指定为“1,2,3”的下标元素，而只是返回 arr[3]

[19] 《程序设计语言——实践之路》P374：从技术上说，连续布局才是真正的多维数组，而“行指针”只是指向数组的指针数组，亦即数组分量是一个其他数组的引用。在本书中这被称为数组的数组、交错数组。

这个元素。因为 JavaScript 将“1,2,3”解释为连续运算，并返回最后一个表达式的值“3”（表达式运算的细节可以参考“2.3 JavaScript 的语法：表达式运算”和“2.7.5 逗号‘,’的二义性”）。如果你试图访问交错数组的某个下标分量，应该用类似如下的语法：

```
arr[1][2][3]
```

使用数组下标时也可以使用数值字符串，但这时在语义上却并不是对索引数组的下标存取，而是对关联数组中的“名-值”存取。这是因为 JavaScript 中的一个数组，既是一个以数组下标顺序存取的索引数组，也是一个可存取属性的关联数组。为了减少二者之间的差异，在将数组视为普通对象并用 for...in 语句列举时，可以列举到那些数值的索引下标。有关数组的这一特性，请参见如下章节：

- 2.5.2.1 对象成员列举、存取和删除
- 5.6 关联数组：对象与数组的动态特性
- 7.9 实现二叉树

#### 2.5.1.4 正则表达式直接量声明

正则表达式由普通字符（字符 a~z、0~9 等）和元字符构成。JavaScript 实现了正则表达式的一个子集，这个子集包括表 2-19 中的元字符[20]。

表 2-19 JavaScript 支持的正则表达式元字符

| 匹配对象 | 元字符 | 备注 |
| --- | --- | --- |
| 字符子集 | \d, \D, \s, \S, \w, \W | 每个元字符只能匹配一个字符。若该元字符表示一个字符子集，则匹配子集中的任一字符。<br>在使用“自定义匹配字符子集”时，可使用'a-z'格式来指定连续子集 |
| 单个字符 | \f, \n, \r, \t, \v, . | |
| 一般性转义字符 | \\, \(, \{ 等 | |
| 位置 | ^, $, \b, \B | |
| 控制字符 | \cx | |
| 十六进制 ASCII 字符 | \xn | |
| 八进制 ASCII 字符 | \n, \nm, \nml | |
| 十六进制 Unicode 字符 | \unnnn | |
| 自定义匹配字符子集 | [xyz], [^xyz] | |

[20] 该表仅是 JavaScript 中正则表达式语法的一个概要，详细说明请参见 Microsoft JScript 手册或《JavaScript 权威指南》，以及《精通正则表达式》等资料或书籍。

续表

| 匹配方式 | 元字符或语法 | |
|---|---|---|
| 匹配分组 | (), (?:),(?=), (?!) | 获取、非获取，以及正、负向预查等 |
| 匹配 x“或”匹配 y | x\|y | |
| 匹配次数设定 | *, +, ?, {n}, {n,}, {n,m} | 如果“?”紧临“匹配次数设定”之后，则表明非贪婪模式。默认为贪婪模式 |
| 非贪婪模式设定 | ? | |
| 引用匹配 | 元字符 | |
| 引用一个已获取的匹配 | \n | （*注 1） |

*注 1：是指在一个正则表达式中复用已通过“匹配分组”获取的、文本中的子字符序列。它的指定格式与“八进制 ASCII 字符”是冲突的。当发生歧义时，优先理解为“获取匹配”；若找不到足够的匹配个数，则理解为“八进制 ASCII 字符”。

除了上表备注中说明的一些常见问题之外，不太常见的用法是在正则表达式内的“引用匹配”——我们可能会在 `String.replace()`等方法中使用`$xx` 来引用某个已获取的匹配，但那是在正则表达式之外进行的。

所谓“在正则表达式内的引用匹配”是指在如下这种情形：

```
// 有字符串如下格式
'abcd1234cdef.............1234........1234....'
'abcd182349cdef.............182349........182349....'
```

若试图表达上述重复出现的子匹配，那么可以使用这样的表达式，如图 2-15 所示。

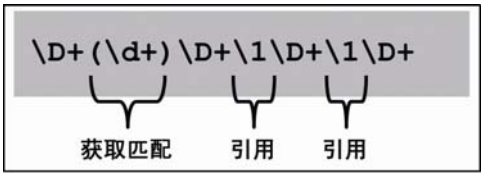


图 2-15 正则表达式中“引用匹配”的使用示例

这样就可以匹配上面列举的两个字符串，而不匹配下面这种：

```
// 不匹配的格式(注意数字不重复)
'abcd1234cdef.............1111........11111111....'
```

另外一种情况是一种经常犯的错误。该错误源于正则表达式具有单独语法格式，而非一个字符串，因此当开发人员试图直接将正则表达式用在字符串中——例如用这样的一个字符串来创建对象时，就会出现一些意料之外的问题。最常见的情况是这样：

```
// 有正则表达式直接量如下
rx = /abcd\n\r/gi
```

开发人员试图将它修改成一个正则表达式对象的创建，于是直接复制了上述代码，修改如下：

```
// 使用字符串创建的正则表达式对象
rx = new RegExp('abcd\n\r', 'gi');
```

在开发人员的预期中，这两个正则应该是一样的。然而这正好忽略了一个问题：在字符串中“\”也是转义符，因此当使用转义后的字符串'abcd\n\r'来创建正则表达式对象时，就出现了错误。而在很多情况下，该错误既非语法错，也不导致运行错——它只是与开发人员的预期不一致而已，所以是被忽略的。而这，可能在代码中留下巨大的隐患。

解决该问题的方法是为字符串中的'\'增加转义，因此上例应被修改为：

```
rx = new RegExp('abcd\\n\\r', 'gi');
```

#### 2.5.1.5 [ES5]在对象直接量中使用属性读写器

除了上述的基础语法之外，在 ES5 中允许在直接量声明中使用属性的**读写器（get/setter）**。读写器的语法规则以及功能都类似于函数，但不使用 function 作为关键字，而是使用 get/set。例如：

```
get propName() {
  //…
}

set propName(newValue) {
  //…
}
```

在对象直接量声明中，允许出现名字或存取器之一，但不允许同时出现。例如：

```
// 正确的用法
obj1 = {
  aName: 'a value.'
}

// 不正确的用法：同时出现“名/值”声明的基本语法与存取器
obj2 = {
  aName: 'a value.',
  get aName() {
    …
  }
}
```

当使用 get/setter 时，所用的名称 *propertyName* 将被认为是一个新声明的属性。因此当它与基本语法同时存在时，解析器将会认为声明了两个相同名字的属性，并抛出一个“无效属性（Invalid property）”的异常。

同时使用 get/setter 是合法的，它表明属性是可读写的；如果只使用其中的一个，则表明该属性是只读/只写的——尽管后者较少使用。

#### 2.5.1.6 讨论：初始器与直接量的区别

ECMAScript 对 JavaScript 语言的规范是一个渐进认识的过程，这一点在变量相关问题的解释上是有充分体现的。我们下面讨论三种表达形式：

```
// 对象、数组与正则表达式的声明
{ … }
[ … ]
/…/…
```

对象与数组在 ES2 中就已经提出，但这时他们并没有直接量形式，而仅仅是一个普通对象；到了 ES3 时，它们如上例中的形式第一次出现即被称为初始器（Initialiser）。然而 ES3 对于 JavaScript 来讲总是一个“迟来的规范”，因为这些表示方法在自 JavaScript 1.2 以来的、历史中的、传统的称谓原本就是“直接量”。也就是说，直接量其实是一个用得更久、更俗成的称谓。

我们的第一个问题是：为什么 ES3 没有沿用习俗称这两种形式为直接量，而是使用了一个新名词，叫做初始器呢？

另一方面，在 ES3 中正则表达式也没有被正式确指为“直接量”， 但到了 ES5，正则表达式就被归类到直接量的范畴中了。因此，我们提第二个问题：相对于数组和一般对象，为什么正则表达式的声明形式又能称为“直接量”呢？

这两个问题的答案是一致的，即：因为直接量的声明中不包括运算过程，而初始器的声明中是可以包括运算过程的。

下面举例来说明这个问题。

```
// 例 1
var x = 100;
var y = 100 + 100;
```

例 1 中，x 的初值被赋以一个直接量 100，而 y 的初值则是一个表达式运算的结果。在例 1 这样的情况下，y 的值是可以在编译期就得到结果的，但在另一些情况下就未必能得到值了：

```
// 例 2
var y = z + 100;
```

显而易见的：例 1 与例 2 中的 y 值都取决于一个运算过程的动态结果，而这一结果可能是无法预知的。

所以 ES5 中就只明确了 5 种直接量声明——在本书中，将 undefined 也归入其中，所以称为 6 种。这意味着这些**直接量（Literals）**的值是可以在声明的同时——在引擎看来即是编译期就确知的：undefined、null、true/false、数值、字符串和正则表达式。

而数组与对象的声明形式就不具备这一特点，反而与例 1 、例 2 中的 y 值类似。例如它们可以声明成：

```
// 例 3
var a1 = [100, 200];
var a2 = [x, 200];
var o1 = {x: 100, y: 1000};
var o2 = {x: zz, y: 1000};
```

这样的声明中：对象与数组的成员，都是可以通过一些表达式来赋以初值的，因此它们本身也“可能”是无法预知的。所以在 ES3 与 ES5 中，都将对象、数组的这两种声明形式称为**“初始器（Initialiser）”**，并归类在“**表达式（Expressions）**”之中。

《JavaScript 权威指南》第四、五版均采用习惯性的称谓，仍将数组与对象的声明形式称为直接量，但对它们的具体叙述却放到了表达式章节之中。在本书的第 1 版中，将包括函数在内的几种声明形式均称为直接量，并进一步地引申出相应的直接量表达式。但在第 2 版中开始依据 ECMAScript 标准，不再将函数、对象、数组的声明形式归为直接量的范畴之内。

本书中一般仍习惯性地将初始器称为直接量，这种情况下主要是强调它可以用于值运算；若有特殊需要，例如，在强调它的表达式特性时，也会使用初始器这一称谓。

## 2.5.2 对象成员

### 2.5.2.1 对象成员列举、存取和删除

可以用下面的方法列举成员的显式成员列表：

```
var obj = {
  // …
}

// 示例：列举成员的显式成员
for (var n in obj) {
  // 在变量 n 中保存有成员名
  alert('name: '+ n + ', value: ' + obj[n]);
}
```

你可以列举到每一个显式的成员名，但并不能确知该对象的设计者是打算让这个成员作为一个方法（method），或者属性（property），亦或者是事件句柄（event）。在 JavaScript 中，任何类型的值都可以成为对象属性，因此这也包括“函数类型的值”；而对象方法和事件句柄的类型信息，也都是“函数类型（或者函数对象实例）”。所以你根本没有办法来辨识它们——也就是说，在 JavaScript 中，你不可能从成员的类型上准确了解对象/类的设计者的原始意图。

我们强调可以列举的是“显式的成员名”，其原因在于 JavaScript 约定了一些隐含的成员名称[21]，例如所有对象都应该具有的“toString”方法。这些名称在 for...in 语句中不会被列举出来。对于一些引擎来说，这些隐含的成员名称总是不被列举（例如，在 JScript 引擎中）；而对于另一些引擎中，只要覆盖、重写这个名称，它就可以被列举出来了（例如，在 SpiderMonkey JavaScript 引擎中）。因此在跨引擎的设计中，我们并不能依赖于 for...in 语句来获取对象成员名称。

现在，我们通过一些方法得到了成员的名字（或者我们在设计代码时，便已经自知了）。有了名字，我们就可以存取它。这有两种方法：

```
var obj = {
  value: 1234,
  method: function() { }
}

// 方法 1: 使用对象成员存取运算符 “.”
var aValue = obj.value;

// 方法 2: 使用对象成员存取运算符 “[ ]”
var aValue = obj['value'];
```

“.”和“[]”都是对象成员存取运算符，所不同的是：前者右边的运算元必须是一个标识符，后者中间的运算元可以是变量、直接量或表达式。

由于“.”号要求运算元是标识符，因此对一些不满足标识符命名规则的属性，就不可以使用“.”号。例如我们前面提到过的“abcd.def”、“1”和“.”这些属性名，这种情况下就只能使用“[]”运算符，例如：

```
var obj = {
  'abcd.def': 1234,
  '1': 4567,
 '.': 7890
}

// 示例: 需要使用[]运算符的一些情况
alert(obj['abcd.def']);
alert(obj['1']);
alert(obj['.']);
```

接下来，在某些情况下你可能需要删除一个对象的属性，以使它不能被 for...in 语句列举。这时你可以使用 delete 运算符，例如：

```
var obj = {
  method: function() { },
  prop: 1234
}
golbal_value = 'abcd';
array_value = [0,1,2,3,4];
function testFunc() {
  value2 = 1234;
```

[21] 在一些书籍中，“隐含的成员名称”被称为“内置成员”或“预定义成员（方法或属性）”。

```
  delete value2;
}// 调用 testFunc()函数，函数内部的 delete 用法也是正确的
testFunc();

// 以下四种用法都是正确的
delete obj.method;
delete obj['prop'];
delete array_value[2];
delete golbal_value;
```

大多数情况下成员都能够被删除，包括对象成员和数组元素，甚至也包括全局对象 Global 的成员，例如：

```
alert( delete isNaN );
```

但是它不能删除：

- 用 var 声明的变量。
- 直接继承自原型的成员。

其中，delete 不能删除继承自原型的成员，但如果修改了这个成员的值，你仍然可以删除它——这将使它恢复到原型的值。关于这一点的真相是：delete 运算事实上是删除了实例的成员表中的值与描述符，可以通过阅读“3.3 JavaScript 中的原型继承”的内容来了解成员继承与值的获取。下面的例子只描述这种现象的表面：

```
function MyObject() {
  this.name = "instance's name";
}
MyObject.prototype.name = "prototype's name";

// 创建后，在构造器中 name 成员被置为值"instanc's name"
var obj = new MyObject();
alert( obj.name );

// 删除该成员
delete obj.name;
// 显示 true，成员名仍然存在
alert( 'name' in obj );
// 恢复到原型的值"prototype's name'
alert( obj.name );
```

这个例子也说明：delete 不能通过实例来删除原型的及父类原型的成员（本例中的 name 成员是在原型中定义的）。如果真的需要删除该属性，你只能对原型实例进行操作，当然，由于这是原型，所以它会直接影响到这个类构造的所有实例。下面的例子说明这一点：

```
function MyObject() {
  // …
}
// 在原型中声明属性
MyObject.prototype.value = 100;
```

```
// 创建实例
var obj1 = new MyObject();
var obj2 = new MyObject();

// 示例 1: 下面的代码并不会使 obj1.value 被删除掉
delete obj1.value;
alert(obj1.value);

// 示例 2:
// 下面的代码可以删除掉 obj1.value. 但是,
// 由于是对原型进行操作, 所以也会使 obj2.value 被删除
delete obj1.constructor.prototype.value;
alert(obj1.value);
alert(obj2.value);
```

JavaScript 的一些官方文档中提及 `delete` 仅在删除一个不能删除的成员时，才会返回 `false`。而其他情况下——例如删除不存在的成员，或者继承自父类 / 原型的成员，都应当返回 `true`。但是请注意下面的示例：

```
function MyObject() {
}
MyObject.prototype.value = 100;

// 该成员继承自原型, 且未被重写, 删除返回 true
// 由于 delete 操作不对原型产生影响, 因此 obj1.value 的值未变化
var obj1 = new MyObject();
alert( delete obj1.value );
alert( obj1.value );

// 尝试删除 Object.prototype, 该成员禁止删除, 返回 false
alert( delete Object.prototype );
```

这种因“不能删除成员”而返回 `false` 的情况并不多。我所知道的这种不能删除的成员大概有 `Function` 对象的 `length`、`prototype`、`arguments` 等极少数的几个，而且这还依赖不同的脚本引擎，例如在 Mozilla 中 `arguments` 就是可以删除的。

除此之外，用 `delete` 操作来删除宿主对象的成员时，也可能存在其他的问题。例如下面这个例子中，属性 `aValue` 就删除不掉，而在取值时又触发异常[22]：

```
// code in Internet Explorer 5~7.x
aValue = 3;

// 显示 true
alert('aValue' in window);
delete aValue;

// 条件仍然为真, 然而用 alert()显示值时却出现异常
if ('aValue' in window) {
  alert(aValue);
}
```

[22] 这里疑为 JScript 的一个 Bug。

尽管并没有资料提及 delete 运算会导致异常（上例中是取值而非删除操作导致异常），但这种情况的确会发现。如果你试图删除宿主（例如，浏览器中的 window）的成员，那么这种可怕的灾难就会发生了：

```
// code in Internet Explorer 5~7.x
window.prop = 'my custom property';
delete window.prop;
```

这时发生的异常可能是“对象不支持该操作”，表明宿主不提供删除成员的能力，不过不同的宿主的处理方案也不一致，例如，Firefox 浏览器就可以正常删除。

#### 2.5.2.2 属性存取与方法调用

很多语言中的“面向对象系统”是由编译器，或者指定的语句/语法来实现的。例如 Delphi 从 Pascal 过渡而来，C++则源自 C 语言，为了实现“面向对象”，它们都各自扩展了自己的语法。然而，JavaScript 并不这样，它是通过“运算”来实现面向对象特性的典型。例如，我们在这一小节要说到的属性存取与方法调用。

如果我们已经创建了一个对象实例 obj，那么可以用上一节所提到的两种方法之一来存取对象属性：

```
// 方法 1: 使用对象成员存取运算符 “.”
var aValue = obj.value;

// 方法 2: 使用对象成员存取运算符 “[ ]”
var avalue = obj['value'];
```

我们已经说过，“.”与“[ ]”是两个运算符。在此基础上，JavaScript 并没有做任何的语法扩展就实现了方法调用：

```
// 方法 1: 基于运算符 “.”
obj.method();

// 方法 2: 基于运算符 “[ ]”
obj['method']();
```

所以，事实上 JavaScript 中的方法调用，就是指“取得对象的成员，并执行函数调用运算”。当然，在这个过程中还要传入 this 对象引用。具体到语法的实现方法上，只需要使“.”和“[ ]”运算的优先级高于“( )”即可。

在 JScript 环境中还存在一种特例：如果一个 ActiveObject 对象实例，那么该对象的方法在一些情况下能不使用“( )”即得到调用。如下例所示：

```
// 示例: Exit 是方法调用，但没有使用 “( )” 运算符
var excel = new ActiveXObject("Excel.Application");
alert('enter to close excel…');
excel.Exit;
```

### 2.5.2.3 对象及其成员的检查

我们可以用“for...in”语句来列举对象成员，但如果要检查一个对象是否具有某个成员，就不用这么麻烦了。JavaScript 使用 in 运算来得到这个结果，其语法为：

```
// 语法
propertyName in object
```

例如：

```
// 示例：用 in 运算检测属性
'value' in obj
```

这种运算最经常出现在引擎环境兼容的代码中，例如：

```
// 示例：为不同的脚本引擎初始化一个 XMLHttpRequest 对象
if ('XMLHttpRequest' in window) {
  // for ie7.0+, or mozilla and compat browser
  return new XMLHttpRequest();
}
else if ('ActiveXObject' in window) {
  // for ie 4.0 ... 6.0
  return new ActiveXObject('Microsoft.XMLHTTP');
}
else {
  throw new Error('can''t init ajax system... ')
}
```

但这样的检查并不一定可靠。因为更早版本的 JavaScript 引擎可能根本没有实现“in”运算。因此在 JavaScript v1.3 之前，我们被建议用下面的代码来做相同的事：

```
// 示例：兼容更早版本的对象检查代码
if (window.XMLHttpRequest) {
  // for ie7.0+, or mozilla and compat browser
  return new XMLHttpRequest();
}
else if (window.ActiveXObject) {
  // for ie 4.0 ... 6.0
  return new ActiveXObject('Microsoft.XMLHTTP');
}
else {
  throw new Error('can\'t init ajax system... ')
}
```

按照前面讲述过的知识，“window.XMLHttpRequest”这个运算用于取 XMLHttpRequest 成员。如果该属性存在，则返回该属性的值。接下来，这个值被类型转换为 if 语句所需的布尔值“true”。同样，按照 JavaScript 语言的约定，取一个不存在的属性的值并不会导致异常，而是返回“undefined”，而“undefined”可以被类型转换为 if 语句所需的布尔值“false”。所以当对“存在的属性”做取值运算时，在 if 语句中的效果正好相当于使用 in 运算；反之对不存在的属性做取值运算时，在 if 语句中的效果也正好相当于使用 in 运算。

但是，应该注意到这里有一个“隐含的类型强制转换”。正是因为这个强制转换，所以下面所有情况都可能导致表达式结果为 false：

```
var obj = {};

function _in(obj, prop) {
  if (obj[prop]) return true;
  return false;
}

// 检测不存在的属性
alert( _in(obj, 'myProp') );

// 检测某些有值的属性，仍会返回 false
var propertyNames = [0, '', [], false, undefined, null];
for (var i=0; i<propertyNames.length; i++) {
  alert( _in(obj, propertyNames[i]) );
}
```

我们看到，一个属性即使存在，如果它的值为 propertyNames 中任意的一个，都将导致_in()检测返回 false。这显然没有达到我们的目的。因此，JavaScript 另外推荐一种方案，以在旧版本中检测属性是否存在：

```
// 示例：兼容更早版本的对象检查代码
if (typeof(window.XMLHttpRequest) != 'undefined') {
  …
}
else if (typeof(window.ActiveXObject) != 'undefined') {
  …
}
else …
```

由于前面说过取不存在的属性将返回 undefined，因此用 typeof 运算来检测该值，一定是 undefined 字符串。这样看起来是达到目的了，但事实上还是存在问题。例如：

```
var obj = {
  'aValue': undefined
};
// 示例：使用 typeof 运算存在的问题
if (typeof(obj.aValue) != 'undefined') {
  ...
}
```

这种情况下，aValue 属性是存在的，但我们不能通过 typeof 运算来检测它是否存在。正是由于这个缘故，在 Web 浏览器中，DOM 的约定是“如果一个属性没有初值，则应该置为 null”。可见，早期的 JavaScript 不能有效地通过 undefined 来检测属性是否存在。同样的道理，JavaScript 规范在较后期的版本中，便要求引擎实现 in 运算，以更有效地检测属性。

上面说如果我们需要检测“对象是否具有某个属性”，这应当使用 in 运算符，而另一些情况下，还需要检测“对象是不是一个类的实例”。这时，就应该使用 instanceof 运算符了。如下例所示：

```
// 示例：用 instanceof 运算检测实例类别
obj instanceof MyObject
```

在 JavaScript 中没有严格意义上的类类型，因此只能通过构造器来检测实例。也就是说，上面的代码会在下面的条件下成立：

```
// 声明构造器
function MyObject() {
  // …
}
// 实例创建
var obj = new MyObject();
// 显示 true
alert(obj instanceof MyObject);
```

instanceof 运算符将会检测类的继承关系。因此一个子类的实例，在对祖先类做 instanceof 运算时，仍会得到 true。如下例：

```
function MyObjectEx() {
  // …
}
MyObjectEx.prototype = new MyObject();

// 实例创建
var obj2 = new MyObjectEx();

// 检测构造类，显示 true
alert(obj2 instanceof MyObjectEx);

// 检测祖先类，显示 true
alert(obj2 instanceof MyObject);
```

#### 2.5.2.4 可列举性

对象成员是否能被列举，称为成员的可列举性。由于 JavaScript 的成员可以被概括为属性（方法调用被实现为“存取对象属性并作为函数执行‘( )’运算”），因此“成员是否能被列举”也就可以表述为“属性是否能被列举”。因此在 JavaScript 中可以看到 propertyIsEnumerable()方法，而不会有（也不必有）检测方法是否可列举的操作。

当某个对象成员不存在或它不可列举时，则对该成员调用 propertyIsEnumerable() 方法将返回 false。比较常见的情况是：JavaScript 对象的内置成员不能被列举。例如：

```
var obj = new Object();

// 不存在'aCustomMember'，显示 false
alert( obj.propertyIsEnumerable('aCustomMember') );

// 内置成员不能被列举
alert( obj.propertyIsEnumerable('constructor') );
```

这种情况下，你可以用 in 运算检测到该成员，但不能用“for…in”语句来列举它。

在 JScript 与 JavaScript 中，对可列举性中的“继承自……”这个描述的理解也是不同的。这导致 propertyIsEnumerable()的行为，以及（包括内置成员在内的）可见性的处理并不一致。关于这一点，我们在“3.3.8.4 如何理解‘继承来的成员’？”中会再次讲到。

《JavaScript 权威指南》指出了 ECMAScript 对 propertyIsEnumerable()的一个规范错误。按照规范，该方法是不检测对象的原型链的，但如果规范更合理一些，那么该方法是应该检测原型链的。为什么？因为事实上原型链上的（父类的）成员也是可以被“for...in”语句列举的，但它的 propertyIsEnumerable()却是 false。如下例所示：

```
// 定义原型链
function MyObject() {
}
function MyObjectEx() {
}
MyObjectEx.prototype = new MyObject();

// 'aCustomMember'是原型链上的（父类的）成员
MyObject.prototype.aCustomMember = 'MyObject';

// 显示 false，因为“继承来的成员”不能被列举
var obj1 = new MyObjectEx();
alert( obj1.propertyIsEnumerable('aCustomMember') );

// 列举 obj1 时，将包括'aCustomMember'
for (var propName in obj1) {
  alert( propName );
}
```

但是，既然规范是这样要求的，那么也必须按照错误的方法来实现，这即是所谓标准的强制性。结果是，在脚本引擎中 propertyIsEnumerable()被实现为“只检测对象的非（自原型链继承而来的）继承属性”。

## 2.5.3 默认对象的指定

JavaScript 提供 with 语句，以便开发人员能在一个代码块中显式地指定默认对象。如果不指定默认对象，则默认使用宿主程序在全局指定的默认对象（在浏览器环境中的默认对象是 window）。

一个代码块中，用户代码要么是在访问一个对象成员，要么是在访问对象（或值）。二者有非常明显的区别：是否使用对象成员存取运算符。因此，脚本引擎可以容易地区分下面的代码的含义：

```
1  // 示例 1: 存取对象成员
2  var obj = new Object();
3  obj.value = 100;
```

```
4
5   // 示例 2: 访问(全局的)对象或值
6   value = 1000;
```

with 语句可以改变上述第 6 行代码的语义，让它存取 obj 对象的成员（而不是全局变量）。例如：

```
7   // (续上例)
8   with (obj) {
9     value *= 2;
10  }
11
12  // 显示 200
13  alert(obj.value)
14  // 显示 1000
15  alert(value);
```

对于对象及其闭包系统来说，with 语句具有某些特别的语法效果。关于这一部分信息，请参见“4.6.4.2 对象闭包带来的可见性效果”。

## 2.6 [ES5]严格模式下的语法限制

严格模式需要使用字符串序列：

```
"use strict"
```

来开启，注意它是包括双引号的（也可以是一对单引号）。换而言之，在代码中它是一个字符串直接量，被用在段代码文本的最前面，作为“指示前缀（Directive Prologue）”。

由于 JavaScript 中的代码块是按语句来解析的，因此这样的指示前缀就可以被解释成“直接量表达式语句”。当然，这也意味着在这个字符串的后面，应该用一个分号或回车符来表示它与后续代码分别是两行语句。

在如下位置加入上述指示前缀，可以开启相应代码块中的严格模式：

- 在全局代码的开始处加入。
- 在 eval 代码开始处加入。
- 在函数声明代码开始处加入。
- 在 new Function()所传入的 body 参数块开始处加入。

例如：

```
// 下面的函数声明表明它是一个运行在严格模式下的函数
function foo() {
  "use strict";
```

```
  return true;
}
```

在本小节中，我们所说的“语法限制”是指如果在代码文本中出现了违例，则在语法分析期该段代码文本就是无效的，所在代码块完全不能装载执行或函数直接量未能成功声明，以及函数对象创建或 eval()执行返回“语法错误”异常。

- 本小节仅讨论严格模式限制中与语法相关的部分，即这些限制会导致语法错误。其他一些将导致运行期错误的限制将在“4.7.1 严格模式下的执行限制”中讨论。

## 2.6.1 语法限制

总的来说，有 7 种语法在严格模式中被禁用了，在旧的 ECMAScript 版本中，它们是合法的[23]。

其一，是在对象直接量声明中，存在相同的属性名。例如：

```
// 禁例 1：对象直接量的相同属性名
var obj = {
  'name': 1,
  'name': 'aName'
}
```

在非严格模式中，上述的声明将使用最后一个有效的声明项。

其二，在函数声明中，参数表中带有相同的参数名。例如：

```
// 禁例 2：函数参数表的相同参数名
func = new Function('x', 'x', 'z', 'return x+z');

// 或
function foo(x,x,z) {
  return x+z
}
```

在非严格模式中，将会传入参数表应该是指定个数的参数（即与形式参数对应），但在代码中访问同名的参数时，只有最后一个声明是有效的。例如：

```
// 在非严格模式中，下面的函数声明是有效的
function foo(m,m,n,m,a,n) {
  return m+n
}

// 显示 6，表明函数声明中有 6 个形式参数
alert(foo.length);

// 显示 10，表明 m,n 的取值分别为第 4 和第 6 个参数
alert(foo(1,2,3,4,5,6));
```

[23] 读者需要自行将代码中的禁例运行在严格模式中以进行测试。

```
// 显示 NaN，因为 m+n 的实际运算行为是"4 + undefined"
alert(foo(1,2,3,4));
```

其三，不能声明或重写 eval 和 arguments 这两个标识符，亦即是说，它们不能出现在赋值运算的左边，也不能使用 var 语句来声明。另外，由于 catch 子句以及具名函数都会隐式地声明变量名，因此在它们的语法中也不允许用 eval 和 arguments 作为标识符。最后要强调的是，arguments 或 eval 也不能使用 delete 去删除。例如：

```
// 禁例 3.1：向 eval 或 arguments 赋值
eval = function() { }

// 禁例 3.2：重新声明 eval 或 arguments
var arguments;

// 禁例 3.3：将 eval 或 arguments 用做 catch 子句的异常对象名
try {
  // …
}
catch (eval) {
  // …
}

// 禁例 3.4：将 eval 或 arguments 用做函数名
function arguments() { }

// 禁例 3.5：删除 arguments，或形式参数名
function foo() {
  delete arguments;
}
```

在非严格模式中，上述语法都是有效的，但在一些引擎中，重写 eval 将导致运行期异常。

其四，使用 0 前缀声明的 8 进制直接量。例如：

```
// 禁例 4：8 进制直接量
var num = 012;
alert(num);
```

在非严格模式中，上述代码运行将显示 10。

其五，用 delete 删除显式声明的标识符、名称或具名函数。例如：

```
// 禁例 5.1：删除变量名
var x;
delete x;

// 禁例 5.2：删除具名函数
function foo() {}
delete foo;

// 禁例 5.3：删除 arguments，或形式参数名
function foo(x) {
```

```
  delete x;
}

// 禁例 5.4 删除 catch 子句中声明的异常对象
try{} catch(e) { delete e }
```

在非严格模式中，通常这些操作只是“无效的”，并不会抛出异常。此外，用 delete 操作其他一些不能被删除的对象属性、标识符时将导致执行期异常[24]。

其六，在代码中使用一些扩展的保留字，这些保留字包括：implements, interface, let, package, private, protected, public, static，以及 yield。例如：

```
// 禁例 6: 使用扩展保留字
var yield;
function let() { }
```

这些保留字并不存在于旧 ECMAScript 版本的保留字列表中，因此在非严格模式中它们是可用的。

其七，在代码中包括 with 语句。例如：

```
// 禁例 4: with 语句
foo = new Function() { with (arguments) return length }
```

注意，在严格模式中，with 语句直接被禁止了。这是一个非常大的变化。

## 2.6.2 严格模式的范围

如果一段代码被标志为“严格模式”，则其中运行的所有代码都必然是严格模式下的。其一，如果在语法检测时发现语法问题，则整个代码块失效，并导致一个语法异常；其二，如果在运行期出现了违反严格模式的代码，则抛出执行异常。

举例来说：

```
1  "use strict";
2
3  function foo() {
4    var arguments;
5  }
```

如果上述代码是引擎初始化完成之后装载的第一段用户代码——例如浏览器中的第一个<script>块，那么由于第一行的“指示前缀”已经表明当前引擎运行在严格模式中，因此 foo()函数在语法解释期就会导致异常，尽管它的 *functionBody* 区并没有这样的指示前缀。

[24] 这些内容属于“执行限制”的范畴，请参见：4.7.1 严格模式下的执行限制。

反过来，如果在某个函数内部加入“指示前缀”，它却不会影响到外面的代码。例如：

```
1  function foo() {
2    "use strict";
3
4    //…
5  }
6
7  var eval;
```

在这段代码中，foo()函数是运行在严格模式下的，而它外面的（全局的）代码却运行在非严格模式下，所以第 7 行是合法的，整段代码也是合法的。

但是，一个函数如果需要运行在严格模式下，则它的名字和参数的违例情况总会被检测——这里反复重申这一点，是因为下面的代码“在形式上”会被误解。例如：

```
1  function eval(x,x) {
2    "use strict";
3
4    //…
5  }
6
7  var arguments;
```

在这个例子中，“指示前缀”在形式上位于第二行，或许会被误认为它不能“向前地”影响到第 1 行。而事实上该指示前缀表明从第 1 行到第 5 行所声明的整个函数都运行在严格模式下——亦即是说，该代码的第 1 行有两个违例（参考上一小节的禁例 3.4 和禁例 2）。但是，第 7 行的代码是合法的。

最后，指示前缀如果出现在代码中间——作为一个直接量表达式语句，那么它将被忽略，也不会导致当前代码块或函数进入严格模式。例如下面整段代码其实没有运行在严格模式中，因此将是合法的：

```
1
2  "use strict";
3  var eval;
4  function foo(m,m) { ;"use strict"; return m }
```

由于第 1 行的空行被理解为空语句，因此第 2 行的字符串不再触发严格模式。类似的，第 4 行的函数体的最前面也是一个空语句，因此第 4 行的函数 foo()也不会进入严格模式。

## 2.7 运算符的二义性

在 JavaScript 中，很多运算符是具有二义性的。因为这些二义性的存在，一方面使

JavaScript 显得更灵活，另一方面也使一些 JavaScript 的代码显得更难理解。

严格来说，我们这里说的二义性并不是学术含义上的。JavaScript 会通过一套语法规则、优先级算法以及默认的系统机制来处理这些“（看起来）存在二义的代码”，使代码在运行时有某一确定的含义。但我在这里要说的是，这些代码在开发人员的理解中是存有二义的，或者是不能那么清晰、直观地理解代码的意图。

我们接下来讲述这些由运算符二义性带来的不清晰的语法。这些语法之所以存在，不是因为我在这里玩弄技巧，而（在某些时候）只是出于想使语法更清晰。但如果原本只是修改代码过程中的误操作，而 JavaScript 引擎未能检出错误，继而使得系统中容存了“不可知”的隐患，那就很可怕了。

本书不打算讨论正则表达式中的符号与运算符之间的二义性问题。在这个前提下，表 2-20 基于对运算符的考查，列出存在二义性的语法元素。

**表 2-20 二义性的语法元素**

| | 运算符/符号 | 运算符含义 | 其他含义 | 章节 |
|---|---|---|---|---|
| 具有二义性的运算符 | , | 连续运算符 | 参数分隔符<br>对象/数组声明分隔符 | 2.7.5 |
| | + | 增值运算符<br>正值运算符<br>连接运算符 | 数字直接量声明<br>（正、负或指数形式） | 2.7.1<br>（*注 1） |
| | () | 函数调用运算符 | 强制运算（优先级）<br>形式参数表 | 2.7.2 |
| | ?: | 条件运算符 | :号有声明标签的含义<br>:号有声明 switch 分支的含义<br>:号有声明对象成员的含义 | 2.7.3 |
| | [] | 数组下标<br>对象成员存取 | 数组直接量声明 | 2.7.6 |
| 其他 | {} | | 函数直接量（代码部分的）声明<br>对象直接量声明<br>复合语句 | 2.7.4 |
| | ; | | 空语句<br>语句分隔符 | （*注 2） |

*注 1：有关加号“+”的二义性问题，部分内容同样适用于运算符“+=”。

*注 2：空语句可以视做语句分隔符使用中的特例，因此本书不讨论这个二义性的细节。

### 2.7.1 加号“+”的二义性

单个的加号作为运算符在 JavaScript 中有三种作用。它可以表示字符串连接，例如：

```
var str = 'hello ' + 'world!';
```

或表示数字取正值的一元运算符，例如：

```
var n = 10;
var n2 = +n;
```

或表示数值表达式的求和运算，例如：

```
var n = 100;
var n2 = n + 1;
```

三种表示法里，字符串连接与数字求和是容易出现二义性的。因为 JavaScript 中对这两种运算的处理将依赖于数据类型，而无法从运算符上进行判读。我们单独地看一个表达式：

```
a = a + b;
```

是根本无法知道它真实的含义是在求和，亦或是在做字符串连接。这在 JavaScript 引擎做语法分析时，也是无法确知的。

加号“+”带来的主要问题与另一条规则有关。这条规则是“如果表达式中存在字符串，则优先按字符串连接进行运算”。例如：

```
var v1 = '123';
var v2 = 456;

// 显示结果值为字符串'123456'
alert( v1 + v2 );
```

这会在一些宿主中出现问题。例如浏览器中，由于 DOM 模型的许多值看起来是数字，但实际上却是字符串。因此试图做“和”运算，却变成了“字符串连接”运算。下面的例子说明了这个问题：

```
<img id="testPic" style="border: 1 solid red">
```

我们看到这个 id 为 testPic 的 IMG 元素（element）有一个宽度为 1 的边框——省略了默认的单位 px（pixel，像素点）。但是如果你试图用下面的代码来加宽它的边框，就会导致错误（一些浏览器忽略该值，另一些则弹出异常，还有一些浏览器则可能崩溃）：

```
var el = document.getElementById('testPic');
el.style.borderWidth += 10;
```

因为事实上在 DOM 模型里，borderWidth 是有单位的字符串值，因此这里的值会是 "1px"。JavaScript 本身并不会出错，它会完成类似下面的运算，并将值赋给 borderWidth：

```
el.style.borderWidth = '1px' + 10;   // 值为 '1px10'
```

这时，浏览器的 DOM 模型无法解释“1px10”的含义，因此出错了。当你再次读

borderWidth 值时，它将仍是值 1px。那么，怎么证明上述的运算过程呢？下面的代码将表明 JavaScript 运算的结果是 1px10，但赋值到 borderWidth 时，是由于 DOM 忽略掉这个错误的值，因此 borderWidth 没有发生实际的修改：

```
alert( el.style.borderWidth = '1px' + 10 );   // 值为 '1px10'
```

这个问题追其根源，一方面在于我们允许了省略单位的样式表写法，另一方面也在于脚本引擎不能根据运算符来确定这里的操作是数值运算还是字符串连接。

后来 W3C 推动 XHTML 规范，试图从第一个方面来避免这个问题，但对开发界的影响仍旧有限。因此，在浏览器的开发商提供的手册中，都会尽可能地写明每一个属性的数据类型，以避免开发人员写出上面这样的代码。在这种情况下，最正确的写法是：

```
var el = document.getElementById('testPic');
// 1. 取原有的单位
var value = parseInt(el.style.borderWidth);
var unit = el.style.borderWidth.substr(value.toString().length);
// 2. 运算结果并附加单位
el.style.borderWidth = value + 10 + unit;

// 如果你确知属性采用了默认单位 px，并试图仍然省略单位值，
// 那么你可以用下面这种方法(我并不推荐这样):
// el.style.borderWidth = parseInt(el.style.borderWidth) + 10;
```

## 2.7.2 括号“()”的二义性

我们来看下面语句中的括号“( )”应该是什么含义呢？

```
var str = typeof(123);
var str = ('please input a string', 1000);
```

第一个 typeof 看起来好像被当成了函数使用——这是否说明 typeof 为一个“内部函数”呢？而第二行代码，你相信它会成功执行并使 str“意外地”赋值为 1000 吗？

括号首先可以作为语句中的词法元素。例如函数声明中的“虚拟参数表”：

```
// 声明函数时，括号用做参数表
function foo(v1, v2) {
  // …
}
```

第二种情况，就是括号只作为“传值参数表(这有别于函数声明中的‘虚拟参数表’)”，注意这里它并不表达函数调用的含义。目前为止，它只出现在 new 关键字的使用中：new 关键字用于创建一个对象实例并负责调用该构造器函数，如果存在一对括号“( )”指示的参数表，则在调用构造器函数时传入该参数表，对此的具体分析，我们已经在“2.5.1.1 使用构造器创建对象实例”中讲述过了。例如：

```
// 构造对象时，用于传入初始化参数
var myArray = new Array('abc', 1, true);
```

接下来，它也可以在 with、for、if、while 和 do...while 等语句，以及 catch()

等子句中用来作为限定表达式的词法元素：

```
// for 语句(for…in 语句类同)
for ( initialize; test; increment )
  statement

// if 语句
if ( expression )
  statement

// while 语句
while ( expression )
  statement

// do…while 语句
do
  statement
while ( expression )
```

在充当 if、while 和 do...while 语句中的词法元素时，括号会有“将表达式结果转换为布尔值”的副作用（参见“5.7.3.2 语句（语义）导致的类型转换”）。在很多时候，语句中的括号会产生类似于“运算”这样的附加效果。

第四种情况，是括号“( )”用于强制表达式运算。这种情况下，基本含义是我们通常说的强制运算优先级。但事实上，不管有没有优先级的问题，括号总是会强制其内部的代码作为表达式运算。例如我们在这一小节开始列举的例子：

```
var str1 = typeof(123);
var str2 = ('please input a string', 1000);
```

在第 1 行代码中，“( )”强制 123 作为单值表达式运算，当然运算结果还是 123，于是再进行 typeof 运算。所以这里的一对括号起到了强制运算的作用。同样的道理，第 2 行代码里的一对括号也起到相同的作用，它强制两个单值表达式做连续运算，由于连续运算符“,”的返回值是最后一个表达式的值，因此这个结果返回了 1000。所以我们要意识到，上面的第 1 行代码并没有调用函数的意思，而第 2 行代码将使 str2 被赋值为 1000。

最后一种情况最为常见：作为函数/方法调用运算符。例如：

```
// 有(), 表明函数调用
foo();
// 没有(), 则该语句只是一个变量的返回。
foo;
```

我们一再强调：函数调用过程中的括号“( )”是运算符。也因此得出推论，当“( )”作为运算符时，它只作用于表达式运算，而不可能作用于语句。所以你只能将位于：

```
function foo() {
  return( 1 + 2);
}
```

这个函数内的、return 之后的括号理解成“强制表达式优先级”，而不是理解成“把 return

当成函数或运算符使用”。所以从代码格式化的角度上来说，在下面两种书写方法中，第二种才是正确的：

```
// 第一种，像函数调用一样，return 后无空格
  return(1 + 2);

// 第二种，return 后置一空格
  return (1 + 2);
```

基于同样的理由，无论“break (my_label)”看起来如何合理，也会被引擎识别为语法错误。因为 my_label 只是一个标签，而不是可以交给“( )”运算符处理的运算元，标签与运算元属于两个各自独立的、可重复（而不发生覆盖）的标识符系统。

## 2.7.3 冒号“:”与标签的二义性

冒号有三种语法作用：声明直接量对象的成员和声明标签，以及在 switch 语句中声明一个分支。冒号还具有一个运算符的含义：在“?:”三元表达式中，表示条件为 false 时的表达式分支。下面的例子说明这几种情况：

```
// 示例 1: 用于声明直接量对象的成员
var obj = {
  value: 100,
  func: function() {
    //…
  }
}

// 示例 2: 用于声明标签
myLabel: {
  // …
}

// 示例 3: 声明 case 分支
switch (obj) {
  case X : break;
  default: {
  }
}

// 示例 4: 用于条件(三元)表达式
X ? 'yes' : 'no'
```

其中，由于三元表达式中的问号“?”没有二义性，因此冒号“:”作为运算符的情况是比较容易识别的；case 和 default 分支能被作为标签语句的特例来解释（参见《JavaScript 权威指南》）。因此冒号的二义性问题集中在标签声明与对象成员的问题上。在下一小节中，我们将给出一个实例来详细说明。

## 2.7.4 大括号“{}”的二义性

大括号有 4 种作用，4 种都是语言的语法符号。第一种比较常见，表示“复合语句”。例如：

```
// 示例 1: 表示标签后的复合语句
myLabel : {
  //...
}
// 示例 2: 在其他语句中表示复合语句
if ( condition ) {
  // ...
}
else {
  // ...
}
```

我们在前面曾经说过，在复合语句末尾的大括号之前，语句的“;”号可以省略。我们又说过，有一类“表达式语句”。因此，下面这行代码，就值得回味了：

```
// 示例 3: 复合语句中的表达式语句
{
  1,2,3
}
```

这其实是一个只有一条语句的复合语句。这条语句是：

```
  1,2,3;
```

语句中的逗号“,”是连续运算符。由于外面有一对大括号，所以我们省略了语句末尾的一个分号。

示例 3 中的格式与大括号的另一种用法就很接近了，这就是对象直接量声明。例如：

```
// 示例 4: 声明对象直接量
var obj = {
  v1: 1,
  v2: 2,
  v3: 3
}
```

由于该例整体上是一条赋值语句，所以其中的直接量声明部分其实是一个表达式：

```
{
  v1: 1,
  v2: 2,
  v3: 3
}
```

注意这几行代码，它们的整体才是一个表达式——一个直接量的单值表达式。

按照前面的解释，我们可以有两种方法将这个单值表达式变成一个语句。如：

```
// 方法 1: 使用分号的表示法
{
  v1: 1,
```

```
  v2: 2,
  v3: 3
};
// 方法 2: 使用复合语句的表示法
{
 {
   v1: 1,
   v2: 2,
   v3: 3
 }
}
```

在第二种方法里，两对大括号的含义就完全不同。从语法解析的角度上讲，二者的区别在于，对象立即值声明时，大括号的内部会有一个：

```
propertyName: expression
```

的语法格式，而大括号用做复合语句时没有这项限制。

好了，接下来我们要问的问题，就会混淆这两种大括号了。看下面这段代码：

```
// 示例大括号的语法歧义
if (true) {
  entry: 1
}
```

在这段代码中，由于在 `if` 后面的语句可能是以下三者之一：

- 一个单行语句。
- 一个表达式（语句）。
- 一个由大括号包含起来的复合语句。

那我们这里是应该把这一对大括号理解为一个对象声明的语法符号呢，还是表示复合语句的语法符号呢？对照一下图 2-16 的两种解释：

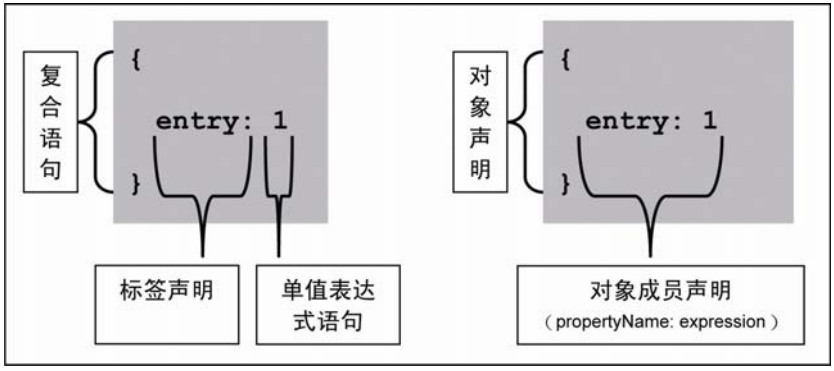


图 2-16 大括号产生的两种语法歧义

在左边这种情况中，`if` 语句后面是一个复合语句；右边这种情况中，`if` 语句后面被

理解成了一个由“对象直接量”构成的单值表达式语句。所以仅从语法上来看，我们无法正确地识别这两种情况。

JavaScript 对此做出的解释是“语句优先”——因为语句的语法分析在时序上先于代码执行。所以表达式被作为次级元素来理解。因此我们输出语句的值，会是按左边的方式解析执行的结果值“1”：

```
// 显示上面的示例语句的值
var code = 'if (true) { entry: 1 }';
var value = eval(code);
alert( value ); // 显示值 1
```

如果用户代码需要用右边的方式来解析执行，那么应该用一对括号“( )”来强制表明“表达式运算”。如：

```
if (true) ({
  entry: 1
});
```

这样一来，返回的结果值就是一个对象直接量。下面的代码用于验证这一结果：

```
// 使用括号“( )”强制表达式运算的结果
var code = 'if (true) ({ entry: 1 })';
var value = eval(code);
alert( value ); // 显示值"[object Object]"
```

大括号的第三种用法，是被用于函数直接量声明时的语法符号：

```
foo = function() {
  // …
}
```

但由于存在“function ()”这样的语法关键字作为前缀，因此基本上是不会混淆的。当然，你得留意阅读一下“2.4.1.4 函数调用语句”，理解一下下面这段导致语法分析失败的代码。需要强调的是，这里的语法歧义是括号“( )”运算符导致的，而不是大括号“{ }”的问题：

```
function foo() {
  // …
}(1,2);
```

最后，大括号也是结构化异常处理的语法符号：

```
try {
  // 代码块 1
}
catch (exception) {
  // 代码块 2
}
finally {
  // 代码块 3
}
```

需要强调的是，这里的大括号并不是复合语句的语法符号，而是结构化异常处理中用来分隔代码块的符号。因为如果它是复合语句的语法标识符，那么它必然是可以用单

行语句来替代的。然而你会发现下面的语句将出现语法分析错误：

```
// 示例：遗漏了 try 后面的语法符号，因此下面的代码导致语法分析错
try
  i=100;
catch (e) { /* 略 */ }
```

## 2.7.5 逗号“,”的二义性

我们先看看下面两行语句的不同：

```
// 示例 1
a = (1, 2, 3);

// 示例 2
a = 1, 2, 3;
```

接下来，你能预想下面这个语句的结果吗？

```
// 示例 3
a = [1, 2,(3,4,5),6]
```

这种用法产生的混乱，是因为逗号“,”既可以是语法分隔符，又可以是运算符所导致的。在上面的示例 1、示例 2 中，逗号都被作为“连续运算符”在使用。示例 1 中的括号是强制运算符，因此它的效果是运算如下表达式：

```
(1, 2, 3)
```

并将结果值赋值给变量 a。由于该表达式是三个（直接量的）单值表达式连续运算，其结果值是最后一个表达式，亦即是数值 3。因此示例 1 的效果是“变量 a 赋值为 3”。而示例 2 则因为没有括号来强制优先级，因此按默认优先级会先完成赋值运算，效果是“变量 a 赋值为 1”。

我们在前面提到过“语句是有值的”。对于示例 1、示例 2 来说，尽管示例 1、示例 2 在变量赋值的效果上并不一样，但语句的值却都是 3。这是因为示例 2 在完成赋值运算“a = 1”之后，还仍将继续执行连续运算，而最后一个表达式就是直接量 3。下面的代码考查这两行代码的语句返回值：

```
// 显示数值 3
alert( eval('a = (1, 2, 3);') );
// 显示数值 3
alert( eval('a = 1, 2, 3;') );
```

同样的，我们也可以知道示例 3 的结果是使变量 a 赋值为：

```
[1, 2, 5, 6]
```

这是因为表达式“(3,4,5)”将会被先运算并返回结果值 5，作为数组声明时的一个元素。示例 3 中要强调的是，逗号在这里分别有“数组声明时的语法分隔符”和“连续运算符”两种作用。

我们再回顾一下示例 2。该示例所展示的问题容易出现在变量声明中。例如我们可能会写出下面这样的代码：

```
// 示例 4：语法解释期就提示出错
var a = 1, 2, 3;
```

然而这个代码却并不会像示例 2 一样正常地执行。因为示例 4 中，逗号被解释成了语句 var 声明时用来分隔多个变量的语法分隔符，而不是连续运算符。而语句 var 要求用“,”号来分隔的是多个标识符，而数值 2 和数值 3 显然都不是合法的标识符，因此示例 4 的代码会在语法解释期就提示出错。

示例 4 的代码出自某个真实的项目。最早这个项目使用如下的代码并可以正常运行：

```
function loadFromRemote() {
  // 获取服务器端的数据片断并返回成字符串，例如返回：
  // "a = 1, 2, 3;"
}

function getRemoteData() {
  var ctx = loadFromRemote();
  return eval(ctx);
}
```

但在一次改写后，结果出错了。修改后的代码是这样：

```
function getRemoteData_2(name) {
  var ctx = loadFromRemote();
  eval( 'var ' + ctx);
  return eval(name);
}
```

这次改写的原因在于想获取某个指定名称 name 的变量值，但又不想因为 eval()执行而破坏全局变量，因此在代码前加“var”，使语句变成局部变量声明和赋值。然而正如示例 4 所示范的，在某些情况下，这行代码会失效，在语法解释期就失败。而这个错误并不总会出现，因此浪费了大量的时间来调试。

根源在于我们将原来的代码从“表达式执行”变成了“变量声明语句”。由于代码已经从语义上发生了根本性的改变，因此不能按预期的效果来执行。更合理的改写方法是：

```
function getRemoteData_2(name) {
  var ctx = loadFromRemote();
  eval('var ' + name');
  eval(ctx);
  return eval(name);
}
```

这与前面的区别在于：我们将变量声明与变量赋值分开，这样保证了“eval(ctx)”这行代码与原来的语义完全相同，而只是限定了它的变量作用域。这不但与我们改写该函数的本意一致，而且也避免了 var 声明语句带来的“语法检测”的副作用。

由此所得的教训是：var 声明会使连续运算表达式变为连续声明语句。

存在同样混乱问题的，还有在“2.3.7 特殊作用的运算符”中列举过的示例：

```
// 显示最后表达式的值"value: 240"
var i = 100;
alert( (i+=20, i*=2, 'value: '+i) );
```

在这个示例的 alert()函数调用中，还存在一对括号，用来表示强制运算。因为alert()是函数，所以它会把下面的代码：

```
alert( i+=20, i*=2, 'value: '+i );
```

理解为三个参数传入。而函数参数表的传值顺序是从左至右的，因此这行代码被理解为：

```
alert( 120, 240, 'value: 240' );
```

但 alert()这个函数自身只会处理一个参数，因此最终显示的结果会是值“120”——需要补充的是，这行代码执行完毕之后，变量“i”的值已经变成 240 了。为了让 JavaScript 理解这里的逗号“,”是连续运算符，而不是函数参数表的语法分隔符，我们在这里加入了一对强制（表达式）运算的括号：

```
alert( (i+=20, i*=2, 'value: '+i) );
```

这样，由于外层的括号是函数调用符，因此它将内部的

```
(i+=20, i*=2, 'value: '+i)
```

理解为一个参数，并且是一个需要求值的表达式。因此这里的括号就顺理成章地理解成了“强制运算符”，下面的代码

```
i+=20
i*=2
'value: '+i
```

也就被作为这个强制运算符的一个运算元：由“,”连接起来的一个表达式。直到绕了这样大的一个圈子，逗号才被合理地解释为“连续运算符”。

## 2.7.6 方括号“[ ]”的二义性

下面的代码会有语法错误吗？

```
/**
 * 示例 1: 方括号的二义性
 */
a = [ [1] [1] ];
```

很奇怪，这个语句并没有语法错误。尽管我们几乎不理解这行代码的含义，但 JavaScript 解释器可以理解，它会使得 a 被赋值为[undefined]。也就是说，右边部分作为表达式，可以被运算出一个结果：只有一个元素的数组，该元素为 undefined。

这个例子其实是在工程中有实际意义的。它最早出现在我的一个项目中，因为我用

如下的方式来声明一个二维表——这种用数组来实现本地数据表的方法其实很常见：

```
var table = [
 [ … ],
 [ … ],
 [ … ]
];
```

但是我在一次“copy/paste”时，漏掉了一个逗号。因此变成了下面这个样子：

```
/**
 * 示例 2: 方括号的二义性
 */
var table = [
 ['A', 1, 2, 3]     // <-- 这里漏掉了一个逗号
 ['B', 3, 4, 5],
 ['C', 5, 6, 7]
];
```

而这在 JavaScript 看来却是正常的语法，所以没有在语法解释时给出警告，而是在运行时产生了意想不到的效果——上面的代码被 JavaScript 引擎解释成了如下的数组声明：

```
var table = [
 undefined,
 ['C', 5, 6, 7]
];
```

出现这个问题的原因，首先在于方括号既可以用于声明数组的直接量，又可以是存取数组下标的运算符。因此对于示例 1：

```
a = [ [1] [1] ];
```

来说，它相当于在执行下面的代码：

```
arr = [1];
a = [ arr[1] ];
```

由于 JavaScript 中直接量可以参与运算，因此第一个“[1]”被理解成了一个数组的直接量，它只有一个元素，即“arr[0] = 1”。接下来，由于它是对象，所以 arr[1]就被理解为取下标为 1 的元素——很显然，这个元素还没有声明。因此“[1][1]”的运算结果就是 undefined，而 a = [ [1][1] ]就变成了：

```
a = [ undefined ];
```

根据这个分析，我们可以推导出下面的一些结论：

```
a = [ [][100] ];    // 第一个数组为空数组，第二个数为任意数值，都将得到 [ undefined ]
a = [ [1,2,3][2] ];// 第一个数组有三个元素，因此 arr[2]是存在的，故而得到 [ 3 ]
a = [ [][] ];    // 第一个数组为空，是正常的，但第二个作为下标运算时缺少索引，故语法错误。
```

我们也知道方括号不但可以作为数组下标运算符，也可以作为对象成员的存取运算符。因此从此前的分析来看，下面的代码也能够得以运行：

```
a = [ []['length'] ]; // 第一个数组为空数组，因此将返回它的长度。结果得到 [ 0 ]
```

这看起来可能只是个小麻烦，但下面的例子也许真的就会出现在你的代码中了：

```
array_properties = [
 ['pop'],
 ['push']
 ['length']
];
```

无论出于什么原因，你可能忘掉了“`['push']`”后面的那个逗号，然而你将得到如下的一个数组——而它居然还能继续参与运算：

```
arr_properties = [
 ['pop'], 1
];
```

不过这样的二义性仍然不够复杂，因为我们还是无法解释在示例 2 中为何会出现一个 `undefined`。而示例 2 之复杂就在于它集中呈现了下面三个语法二义性带来的恶果：

- 方括号可以被理解为数组声明或下标存取。
- 方括号还可以被理解为对象成员存取。
- 逗号可以被理解为语法分隔符或连续运算符。

我们再来看这个例子：

```
/**
 * 示例 2: 方括号的二义性
 */
var table = [
 ['A', 1, 2, 3]     // <-- 这里漏掉了一个逗号
 ['B', 3, 4, 5],
 ['C', 5, 6, 7]
];
```

它的第 2 行并没有被理解成一个数组，也没有直接理解成数组元素的存取。相反，它理解成了 4 个表达式连续运算。因为从语法上来说，由于`['A', 1, 2, 3]`是一个数组对象，因此后面的方括号“`[ ]`”会被理解为“对象属性存取运算符”。那么规则就变成了这样：

- 如果其中运算的结果是整数，则用于做下标存取。
- 如果其，中运算的结果是字符串，则用于对象成员存取。

在这里，`['B', 3, 4, 5]`的作用是运算取值，所以“`'B', 3, 4, 5`”被当成了 4 个“各由一个直接量构成的”表达式。“`,`”号也就不再是数组声明时的语法分隔符，而是连续运算符了。而上一小节中我们说过，“`,`”号作为连续运算符时，是返回最后一个表达式的值。于是，表达式“`'B', 3, 4, 5`”就得到了 `5` 这个值。然后，JavaScript 会据此把下面的代码：

```
var table = [
```

```
 ['A', 1, 2, 3]     // <-- 这里漏掉了一个逗号
 ['B', 3, 4, 5],
 ['C', 5, 6, 7]
];
```

理解成：

```
var table = [
 ['A', 1, 2, 3][5],
 ['C', 5, 6, 7]
];
```

而['A', 1, 2, 3]这个数组没有第五个元素，于是这里的声明结果变成了：

```
var table = [
 undefined,
 ['C', 5, 6, 7]
];
```

图 2-17 更加清晰地展示了这个声明与运算交叠在一起的过程：

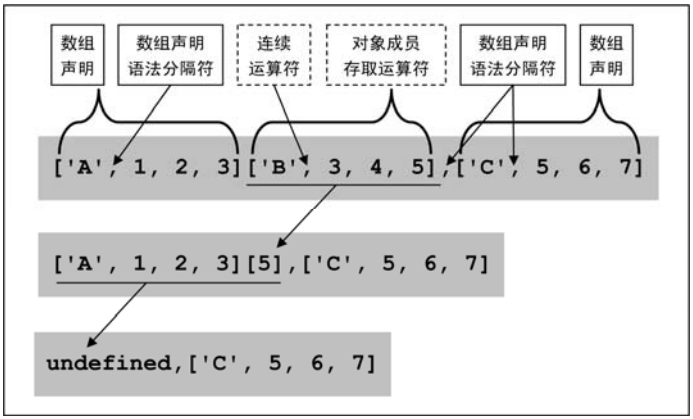


图 2-17 示例 2 运算过程的详细解析

用同样的方法，我们就不难解释表 2-21 中的代码了。

表 2-21 一些其他类似代码的分析

| | 用户声明 | 引擎的理解 |
|---|---|---|
| 示例 1 | var table = [<br>['A', 1, 2, 3] // <-- 这里漏掉了一个逗号<br>['B', 3, 4, 0], // 理解为取下标 0<br>['C', 5, 6, 7]<br>]; | var table = [<br>'A',<br>['C', 5, 6, 7]<br>]; |
| 示例 2 | var table = [<br>['A', 1, 2, 3] // <-- 这里漏掉了一个逗号<br>['B','length'], // 理解为取属性'length'<br>['C', 5, 6, 7]<br>]; | var table = [<br>4,<br>['C', 5, 6, 7]<br>]; |