第4章

表达式和运算符

表达式（expression）JavaScript中的一个短语，JavaScript解释器会将其计算（evaluate）出一个结果。程序中的常量是最简单的一类表达式。变量名也是一种简单的表达式，它的值就是赋值给变量的值。复杂表达式是由简单表达式组成的。比如，数组访问表达式是由一个表示数组的表达式、左方括号、一个整数表达式和右方括号构成。它们所组成的新的表达式的运算结果是该数组的特定位置的元素值。同样的，函数调用表达式由一个表示函数对象的表达式和0个或多个参数表达式构成。

将简单表达式组合成复杂表达式最常用的方法就是使用运算符（operator）。运算符按照特定的运算规则对操作数（通常是两个）进行运算，并计算出新值。乘法运算符“*”是比较简单的例子。表达式x*y是对两个变量表达式x和y进行运算并得出结果。有时我们更愿意说运算符返回了一个值而不是“计算”出了一个值。

本章将讲解所有的JavaScript运算符，同时也讲解不涉及运算符的表达式（比如访问数组元素和函数调用）。如果你熟悉C语法风格的其他编程语言，你会发现大多数JavaScript表达式和运算符都似曾相识。

4.1 原始表达式

最简单的表达式是“原始表达式”（primary expression）。原始表达式是表达式的最小单位——它们不再包含其他表达式。JavaScript中的原始表达式包含常量或直接量、关键字和变量。

直接量是直接在程序中出现的常数值。它们看起来像：

```
1.23    // 数字直接量
```

```
"hello"      // 字符串直接量
/pattern/    // 正则表达式直接量
```

JavaScript数字直接量的语法在3.1节已经做了讲解。字符串直接量在3.2节做了讲解。正则表达式直接量语法在3.2.4节做了简单介绍，在第10章将做专门讲解。

JavaScript中的一些保留字构成了原始表达式：

```
true    // 返回一个布尔值：真
false   // 返回一个布尔值：假
null    // 返回一个值：空
this    // 返回"当前"对象
```

我们在3.3节和3.4节中学习了true、false和null。和其他关键字不同，this并不是一个常量，它在程序的不同地方返回的值也不相同。this关键字经常在面向对象编程中出现。在一个方法体内，this返回调用这个方法的对象。参照 4.5节、第8章（8.2.2节）和第9章来获取关于this的详细信息。

最后，第三种原始表达式是变量：

```
i           // 返回变量i的值
sum         //返回sum的值
undefined   // undefined是全局变量，和null不同，它不是一个关键字
```

当JavaScript代码中出现了标识符，JavaScript会将其当做变量而去查找它的值。如果变量名不存在，表达式运算结果为undefined。然而，在ECMAScript 5的严格模式中，对不存在的变量进行求值会抛出一个引用错误异常。

4.2 对象和数组的初始化表达式

对象和数组初始化表达式实际上是一个新创建的对象和数组。这些初始化表达式有时称做“对象直接量”和“数组直接量”。然而和布尔直接量不同，它们不是原始表达式，因为它们所包含的成员或者元素都是子表达式。数组初始化表达式语法非常简单，我们以此开始。

数组初始化表达式是通过一对方括号和其内由逗号隔开的列表构成的。初始化的结果是一个新创建的数组。数组的元素是逗号分隔的表达式的值：

```
[]          // 一个空数组：[]内留空即表示该数组没有任何元素
[1+2,3+4]   // 拥有两个元素的数组，第一个是3，第二个是7
```

数组初始化表达式中的元素初始化表达式也可以是数组初始化表达式。也就是说，这些表达式是可以嵌套的：

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

JavaScript对数组初始化表达式进行求值的时候，数组初始化表达式中的元素表达式也都会各自计算一次。也就是说，数组初始化表达式每次计算的值有可能是不同的。

数组直接量中的列表逗号之间的元素可以省略，这时省略的空位会填充值undefined。例如，下面这个数组包含5个元素，其中三个元素是undefined：

```
var sparseArray = [1,,,,5];
```

数组直接量的元素列表结尾处可以留下单个逗号，这时并不会创建一个新的值为undefined的元素。

对象初始化表达式和数组初始化表达式非常类似，只是方括号被花括号代替，并且每个子表达式都包含一个属性名和一个冒号作为前缀：

```
var p = { x:2.3, y:-1.2 };     // 一个拥有两个属性成员的对象
var q = {};                    // 一个空对象
q.x = 2.3; q.y = -1.2;         // q的属性成员和p的一样
```

对象直接量也可以嵌套，比如：

```
var rectangle = { upperLeft: { x: 2, y: 2 },
                  lowerRight: { x: 4, y: 5 } };
```

JavaScript求对象初始化表达式的值的时候，对象表达式也都会各自计算一次，并且它们不必包含常数值：它们可以是任意JavaScript表达式。同样，对象直接量中的属性名称可以是字符串而不是标识符（这在那些只能使用保留字或一些非法标识符作为属性名的地方非常有用）：

```
var side = 1;
var square = { "upperLeft": { x: p.x, y: p.y },
               'lowerRight': { x: p.x + side, y: p.y + side}};
```

第6章和第7章会再次讨论对象和数组的初始化表达式。

4.3 函数定义表达式

函数定义表达式定义一个JavaScript函数。表达式的值是这个新定义的函数。从某种意义上讲，函数定义表达式可称为“函数直接量”，毕竟对象初始化表达式也称为“对象直

接量”。一个典型的函数定义表达式包含关键字function，跟随其后的是一对圆括号，括号内是一个以逗号分割的列表，列表含有0个或多个标识符（参数名），然后再跟随一个由花括号包裹的JavaScript代码段（函数体），例如：

```
// 这个函数返回传入参数值的平方
var square = function(x) { return x * x; }
```

函数定义表达式同样可以包含函数的名字。函数也可以通过函数语句来定义，而不是函数表达式。更多详情会在第8章中讨论。

4.4 属性访问表达式

属性访问表达式运算得到一个对象属性或一个数组元素的值。JavaScript为属性访问定义了两种语法：

```
expression . identifier
expression [ expression ]
```

第一种写法是一个表达式后跟随一个句点和标识符。表达式指定对象，标识符则指定需要访问的属性的名称。第二种写法是使用方括号，方括号内是另外一个表达式（这种方法适用于对象和数组）。第二个表达式指定要访问的属性的名称或者代表要访问数组元素的索引。这里有一些具体的例子：

```
var o = {x:1,y:{z:3}};          // 一个示例对象
var a = [o,4,[5,6]];            // 一个包含这个对象的示例数组
o.x                             // => 1: 表达式o的x属性
o.y.z                           // => 3: 表达式o.y的z属性
o["x"]                          // => 1: 对象o的x属性
a[1]                            // => 4: 表达式a中索引为1的元素
a[2]["1"]                       // => 6: 表达式a[2]中索引为1的元素
a[0].x                          // => 1: 表达式a[0]的x属性
```

不管使用哪种形式的属性访问表达式，在“.”和“[”之前的表达式总是会首先计算。如果计算结果是null或者undefined，表达式会抛出一个类型错误异常，因为这两个值都不能包含任意属性。如果运算结果不是对象（或者数组），JavaScript会将其转换为对象（参考3.6节）。如果对象表达式后跟随句点和标识符，则会查找由这个标识符所指定的属性的值，并将其作为整个表达式的值返回。如果对象表达式后跟随一对方括号，则会计算方括号内的表达式的值并将它转换为字符串。不论哪种情况，如果命名的属性不存在，那么整个属性访问表达式的值就是undefined。

显然.identifier的写法更加简单，但需要注意的是，这种方式只适用于要访问的属性名称是合法的标识符，并且需要知道要访问的属性的名字。如果属性名称是一个保留字或

者包含空格和标点符号，或是一个数字（对于数组来说），则必须使用方括号的写法。当属性名是通过运算得出的值而不是固定的值的时候，这时必须使用方括号写法（具体示例参照6.2.1节）。

对象和其属性的细节会在第6章涵盖。数组及其元素会在第7章讲述。

4.5 调用表达式

JavaScript中的调用表达式（invocation expression）是一种调用（或者执行）函数或方法的语法表示。它以一个函数表达式开始，这个函数表达式指代了要调用的函数。函数表达式后跟随一对圆括号，括号内是一个以逗号隔开的参数列表，参数可以有0个也可有多个，例如：

```
f(0)                // f是一个函数表达式；0是一个参数表达式
Math.max(x,y,z)     // Math.max 是一个函数；x，y 和 z 是参数
a.sort()            // a.sort是一个函数，它没有参数
```

当对调用表达式进行求值的时候，首先计算函数表达式，然后计算参数表达式，得到一组参数值。如果函数表达式的值不是一个可调用的对象，则抛出一个类型错误异常（所有的函数都是可调用的，即使宿主对象不是函数它也有可能被调用，这里的区别将在8.7.7节讲述）。然后，实参的值被依次赋值给形参，这些形参是定义函数时指定的，接下来开始执行函数体。如果函数使用`return`语句给出一个返回值，那么这个返回值就是整个调用表达式的值。否则，调用表达式的值就是`undefined`。函数调用——包括当形参表达式的个数和函数定义中实参的个数不匹配的时候的运行情况——的细节将会在第8章详细讨论。

任何一个调用表达式都包含一对圆括号和左圆括号之前的表达式。如果这个表达式是一个属性访问表达式，那么这个调用称做“方法调用”（method invocation）。在方法调用中，执行函数体的时候，作为属性访问主题的对象和数组便是其调用方法内`this`的指向。这种特性使得在面向对象编程范例中，函数（其OO名称为“方法”）可以调用其宿主对象。参照第9章以获取更相信的信息。

并不是方法调用的调用表达式通常使用全局对象作为`this`关键字的值。然而在ECMAScript 5中，那些通过严格模式定义的函数在调用时将使用`undefined`作为`this`的值，`this`不会指向全局对象。参照5.7.3节以获得更多关于严格模式的信息。

4.6 对象创建表达式

对象创建表达式（object creation expression）创建一个对象并调用一个函数（这个函数

称做构造函数）初始化新对象的属性。对象创建表达式和函数调用表达式非常类似，只是对象创建表达式之前多了一个关键字new：

```
new Object()
new Point(2,3)
```

如果一个对象创建表达式不需要传入任何参数给构造函数的话，那么这对空圆括号是可以省略掉的：

```
new Object
new Date
```

当计算一个对象创建表达式的值时，和对象初始化表达式通过{}创建对象的做法一样，JavaScript首先创建一个新的空对象，然后，JavaScript通过传入指定的参数并将这个新对象当做this的值来调用一个指定的函数。这个函数可以使用this来初始化这个新创建对象的属性。那些被当成构造函数的函数不会返回一个值，并且这个新创建并被初始化后的对象就是整个对象创建表达式的值。如果一个构造函数确实返回了一个对象值，那么这个对象就作为整个对象创建表达式的值，而新创建的对象就废弃了。

构造函数的细节将在第9章讲述。

4.7 运算符概述

JavaScript中的运算符用于算术表达式、比较表达式、逻辑表达式、赋值表达式等。表4-1简单列出了JavaScript中的运算符，作为一个方便的参照。

需要注意的是，大多数运算符都是由标点符号表示的，比如“+”和“=”。而另外一些运算符则是由关键字表示的，比如delete和instanceof。关键字运算符和标点符号所表示的运算符一样都是正规的运算符，它们的语法都非常言简意赅。

表4-1是按照运算符的优先级排序的，前面的运算符优先级要高于后面的运算符优先级。被水平分割线分隔开来的运算符具有不同的优先级。标题为A的列表示运算符的结合性，L（从左至右）或R（从右至左），标题为N的列表示操作数的个数。标题为“类型”的列表示期望的操作数类型，以及运算符的结果类型（在“→”符号之后）。表4-1之后的段落会解释优先级、结合性和操作数类型的概念。表4-1只对运算符做单独讨论。

表4-1：JavaScript 运算符

运算符	操作	A	N	类型
++	前/后增量	R	1	lval[①]→num
--	前/后减量	R	1	lval→num
-	求反	R	1	num→num
+	转换为数字	R	1	num→num
~	按位求反	R	1	int→int
!	逻辑非	R	1	bool→bool
delete	删除属性	R	1	lval→bool
typeof	检测操作数类型	R	1	any→str
void	返回undefined值	R	1	any→undef
*、/、%	乘，除，求余	L	2	num,num→num
+、-	加，减	L	2	num,num→num
+	字符串连接	L	2	str,str→str
<<	左移位	L	2	int,int→int
>>	有符号右移	L	2	int,int→int
>>>	无符号右移	L	2	int,int→int
<、<=、>、>=	比较数字顺序	L	2	num,num→bool
<、<=、>、>=	比较在字母表中的顺序	L	2	str,str→bool
instanceof	测试对象类	L	2	obj,func→bool
in	测试属性是否存在	L	2	str,obj→bool
==	判断相等	L	2	any,any→bool
!=	判断不等	L	2	any,any→bool
===	判断恒等	L	2	any,any→bool
!==	判断非恒等	L	2	any,any→bool
&	按位与	L	2	int,int→int
^	按位异或	L	2	int,int→int
|	按位或	L	2	int,int→int
&&	逻辑与	L	2	any,any→any
||	逻辑或	L	2	any,any→any
?:	条件运算符	R	3	bool,any,any→any
=	变量赋值或对象属性赋值	R	2	lval,any→any
*=、/=、%=、+=、-=、&=、^=、|=、<<=、>>=、>>>=	运算且赋值	R	2	lval,any→any
,	忽略第一个操作数，返回第二个操作数	L	2	any,any→any

① lval是left-value的简写，意思是“左值”。

4.7.1 操作数的个数

运算符可以根据其操作数的个数进行分类。JavaScript中的大多数运算符（比如“*”乘法运算符）是一个二元运算符（binary operator），将两个表达式合并成一个稍复杂的表达式。换言之，它们的操作数均是两个。JavaScript同样支持一些一元运算符（unary operator），它们将一个表达式转换为另一个稍复杂的表达式。表达式-x中的“-”运算符就是一个一元运算符，是将操作数x求负值。最后，JavaScript支持一个三元运算符（ternary operator），条件判断运算符“?:”，它将三个表达式合并成一个表达式。

4.7.2 操作数类型和结果类型

一些运算符可以作用于任何数据类型，但仍然希望它们的操作数是指定类型的数据，并且大多数运算符返回（或计算出）一个特定类型的值。在表4-1标题为“类型”的列中列出了运算符操作数的类型（箭头前）和运算结果的类型（箭头后）。

JavaScript运算符通常会根据需要对操作数进行类型转换（参照3.8节）。乘法运算符“*”希望操作数为数字，但表达式"3"*"5"却是合法的，因为JavaScript会将操作数转换为数字。这个表达式的值是数字15，而不是字符串“15”。之前也提到过，JavaScript中的所有值不是真值就是假值，因此对于那些希望操作数是布尔类型的操作符来说，它们的操作数可以是任意类型。

有一些运算符对操作数类型有着不同程度的依赖。最明显的例子是加法运算符，“+”运算符可以对数字进行加法运算，也可以对字符串作连接。同样，比如“<”比较运算符可以根据操作数类型的不同对数字进行大小值的比较，也可以比较字符在字母表中的次序先后。单个运算符的描述充分解释了它们对类型有着怎样的依赖以及对操作数进行怎样的类型转换。

4.7.3 左值

你可能会注意到，表4-1中的赋值运算符和其他少数运算符期望它们的操作数是lval类型。左值（lvalue）是一个古老的术语，它是指“表达式只能出现在赋值运算符的左侧”。在JavaScript中，变量、对象属性和数组元素均是左值。ECMAScript规范允许内置函数返回一个左值，但自定义的函数则不能返回左值。

4.7.4 运算符的副作用

计算一个简单的表达式（比如2*3）不会对程序的运行状态造成任何影响，程序后续执行的计算也不会受到该计算的影响。而有一些表达式则具有很多副作用，前后的表达式运算会相互影响。赋值运算符是最明显的一个例子：如果给一个变量或属性赋值，那么

那些使用这个变量或属性的表达式的值都会发生改变。“++”和“--”递增和递减运算符与此类似，因为它们包含隐式的赋值。delete运算符同样有副作用：删除一个属性就像（但不完全一样）给这个属性赋值undefined。

其他的JavaScript运算符都没有副作用，但函数调用表达式和对象创建表达式有些特别，在函数体或者构造函数内部运用了这些运算符并产生了副作用的时候，我们说函数调用表达式和对象创建表达式是有副作用的。

4.7.5 运算符优先级

表4-1中所示的运算符是按照优先级从高到低排序的，每个水平分割线内的一组运算符具有相同的优先级。运算符优先级控制着运算符的执行顺序。优先级高的运算符（表格的顶部）的执行总是先于优先级低（表格的底部）的运算符。

看一下下面这个表达式：

```
w = x + y*z;
```

乘法运算符“*”比加法运算符“+”具有更高的优先级，所以乘法先执行，加法后执行。然后，由于赋值运算符“=”具有最低的优先级，因此赋值操作是在右侧的表达式计算出结果后进行的。

运算符的优先级可以通过显式使用圆括号来重写。为了让加法先执行，乘法后执行，可以这样写：

```
w = (x + y)*z;
```

需要注意的是，属性访问表达式和调用表达式的优先级要比表4-1中列出的所有运算符都要高。看一下这个例子：

```
typeof my.functions[x](y)
```

尽管typeof是优先级最高的运算符之一，但typeof也是在两次属性访问和函数调用之后执行的。

实际上，如果你真的不确定你所使用的运算符的优先级，最简单的方法就是使用圆括号来强行指定运算次序。有些重要规则需要熟记：乘法和除法的优先级高于加法和减法，赋值运算的优先级非常低，通常总是最后执行的。

4.7.6 运算符的结合性

在表4-1中标题为A的列说明了运算符的结合性。L指从左至右结合，R指从右至左结合。

结合性指定了在多个具有同样优先级的运算符表达式中的运算顺序。从左至右是指运算的执行是按照由左到右的顺序进行。例如，减法运算符具有从左至右的结合性，因此：

```
w = x - y - z;
```

和这段代码一模一样：

```
w = ((x - y) - z);
```

反过来讲，下面这个表达式：

```
x = ~-y;
w = x = y = z;
q = a?b:c?d:e?f:g;
```

和这段代码一模一样：

```
x = ~(-y); w = (x = (y = z)); q =
a?b:(c?d:(e?f:g));
```

因为一元操作符、赋值和三元条件运算符都具有从右至左的结合性。

4.7.7 运算顺序

运算符的优先级和结合性规定了它们在复杂的表达式中的运算顺序，但并没有规定子表达式的计算过程中的运算顺序。JavaScript总是严格按照从左至右的顺序来计算表达式。例如，在表达式w=x+y*z中，将首先计算子表达式w，然后计算x、y和z，然后，y的值和z的值相乘，再加上x的值，最后将其赋值给表达式w所指代的变量或属性。给表达式添加圆括号将会改变乘法、加法和赋值运算的关系，但从左至右的顺序是不会改变的。

只有在任何一个表达式具有副作用而影响到其他表达式的时候，其求值顺序才会和看上去有所不同。如果表达式x中的一个变量自增1，这个变量在表达式z中使用，那么实际上是先计算出了x的值再计算z的值，这一点非常重要[译注1]。

4.8 算术表达式

本节涵盖了那些进行算术计算的运算符，以及对操作数的算术操作。乘法、除法和减法

译注1：作者在这里揭示了一种很容易忽略的现象。假设存在a=1，那么“b=(a++)+a;”将如何计算结果呢？按照正文所述，顺序应该是，1）计算b，2）计算a++（假设值为c），3）计算a，4）计算c+a，5）将c+a的结果赋值给b。按照“++”的定义，第2）步中a++的结果依然是1，即c为1，随后a立即增1，因此在执行第3）步时，a的值已经是2。所以b的结果为3。很多初学者会误认为a增1的操作是在表达式计算完毕后执行的。

运算符非常简单，我们首先讲解它们。加法运算符单独占一节，因为加法同样可以做字符串连接操作，并且其类型转换有些特殊。一元运算符和位运算符同样在单独的两节中会讲到。

基本的算术运算符是*（乘法）、/（除法）、%（求余）、+（加法）和-（减法）。我们会在随后有专门一节讲述“+”运算符。剩下的4个运算符非常简单，只是在必要的时候将操作数转换为数字而已，然后求积、商、余数和差。所有那些无法转换为数字的操作数都转换为NaN值。如果操作数（或者转换结果）是NaN值，算术运算的结果也是NaN。

运算符“/”用第二个操作数来除第一个操作数，如果你使用过那些区分整型和浮点型数字的编程语言，那么当用一个整数除以另一个整数时，则希望得到的结果也是整数。但在JavaScript中，所有的数字都是浮点型的，除法运算的结果也是浮点型，比如，5/2的结果是2.5，而不是2。除数为0的运算结果为正无穷大或负无穷大，而0/0的结果是NaN，所有这些运算均不会报错。

运算符“%”计算的是第一个操作数对第二个操作数的模[译注2]。换句话说，就是第一个操作数除以第二个操作数的余数。结果的符号和第一个操作数（被除数）的符号保持一致。例如，5%2结果是1，-5%2的结果是-1。

求余运算符的操作数通常都是整数，但也适用于浮点数，比如，6.5%2.1结果是0.2。

4.8.1 “+”运算符

二元加法运算符“+”可以对两个数字做加法，也可以做字符串连接操作：

```
1 + 2 // => 3
"hello" + " " + "there" // => "hello there"
"1" + "2" // => "12"
```

当两个操作数都是数字或都是字符串的时候，计算结果是显而易见的。然而对于其他情况来说，则要进行一些必要的类型转换，并且运算符的行为依赖于类型转换的结果。加号的转换规则优先考虑字符串连接，如果其中一个操作数是字符串或者转换为字符串的对象，另外一个操作数将会转换为字符串，加法将进行字符串的连接操作。如果两个操作数都不是类字符串（string-like）的，那么都将进行算术加法运算。

从技术上讲，加法操作符的行为表现为：

- 如果其中一个操作数是对象，则对象会遵循对象到原始值的转换规则转换为原始类值（参照 3.8.3节）：日期对象通过toString()方法执行转换，其他对象则通过

译注2： 求余运算也叫做模运算，模就是余数。

valueOf()方法执行转换（如果valueOf()方法返回一个原始值的话）。由于多数对象都不具备可用的valueOf()方法，因此它们会通过toString()方法来执行转换。

- 在进行了对象到原始值的转换后，如果其中一个操作数是字符串的话，另一个操作数也会转换为字符串，然后进行字符串连接。
- 否则，两个操作数都将转换为数字（或者NaN），然后进行加法操作。

这里有一些例子：

```
1 + 2         // => 3: 加法
"1" + "2"     // => "12": 字符串连接
"1" + 2       // => "12": 数字转换为字符串后进行字符串连接
1 + {}        // => "1[object Object]": 对象转换为字符串后进行字符串连接
true + true   // => 2: 布尔值转换为数字后做加法

2 + null      // => 2: null转换为0后做加法
2 + undefined // => NaN: undefined转换为NaN后做加法
```

最后，需要特别注意的是，当加号运算符和字符串和数字一起使用时，需要考虑加法的结合性的对运算顺序的影响。也就是说，运算结果是依赖于运算符的运算顺序的，比如：

```
1 + 2 + " blind mice";       // => "3 blind mice"
1 + (2 + " blind mice");     // => "12 blind mice"
```

第一行没有圆括号，“+”运算符具有从左至右的结合性，因此两个数字首先进行加法计算，计算结果和字符串进行连接。在第二行中，圆括号改变了运算顺序：数字2和字符串连接，生成一个新字符串，然后数字1和这个新字符串再次连接，生成了最终结果。

4.8.2 一元算术运算符

一元运算符作用于一个单独的操作数，并产生一个新值。在JavaScript中，一元运算符具有很高的优先级，而且都是右结合（right-associative）。本节将讲述一元算术运算符（+、－、++和－－），必要时，它们会将操作数转换为数字。需要注意的是，“+”和“－”是一元运算符，也是二元运算符。

下面介绍一元算术运算符：

一元加法(+)

一元加法运算符把操作数转换为数字（或者NaN），并返回这个转换后的数字。如果操作数本身就是数字，则直接返回这个数字。

一元减法（-）

当“-”用做一元运算符时，它会根据需要把操作数转换为数字，然后改变运算结果的符号。

递增（++）

递增“++”运算符对其操作数进行增量（加一）操作，操作数是一个左值（lvalue）（变量、数组元素或对象属性）。运算符将操作数转换为数字，然后给数字加1，并将加1后的数值重新赋值给变量、数组元素或者对象属性。

递增“++”运算符的返回值依赖于它相对于操作数的位置。当运算符在操作数之前，称为“前增量”（pre-increment）运算符，它对操作数进行增量计算，并返回计算后的值。当运算符在操作数之后，称为“后增量”（post-increment）运算符，它对操作数进行增量计算，但返回未做增量计算的（unincremented）值。思考一下如下两行代码之间的区别：

```
var i = 1, j = ++i;     // i和j的值都是2
var i = 1, j = i++;     // i是2，j是1
```

需要注意的是，表达式++x并不总和x=x+1完全一样，“++”运算符从不进行字符串连接操作，它总是会将操作数转换为数字并增1。如果x是字符串“1”，++x的结果就是数字2，而x+1是字符串“11”。

同样需要注意的是，由于JavaScript会自动进行分号补全，因此不能在后增量运算符和操作数之间插入换行符。如果插入了换行符，JavaScript将会把操作数当做一条单独的语句，并在其之前补上一个分号。

不管是前增量还是后增量，这个运算符通常用在for循环中，用于控制循环内的计数器（见5.5.3节）。

递减（--）

递减“-”运算符的操作数也是一个左值。它把操作数转换为数字，然后减1，并将计算后的值重新赋值给操作数。和“++”运算符一样，递减“--”运算符的返回值依赖于它相对操作数的位置，当递减运算符在操作数之前，操作数减1并返回减1之后的值。当递减运算符在操作数之后，操作数减1并返回减1之前的值。当递减运算符在操作符的右侧时，运算符和操作数之间不能有换行符。

4.8.3 位运算符

位运算符可以对由数字表示的二进制数据进行更低层级的按位运算。尽管它们并不是传统的数学运算，但这里也将其归类为算术运算符，因为它们作用于数值类型的操作数并返回数字。这些运算符在JavaScript编程中并不常用，如果你对十进制整数的二进制表示

并不熟悉的话，你可以跳过本节内容。这里的4个运算符都是对操作数的每个位进行布尔运算，这里将操作数的每个位当做布尔值（1=true，0=false），其他三个位运算符用来进行左移位和右移位。

位运算符要求它的操作数是整数，这些整数表示为32位整型而不是64位浮点型。必要时，位运算符首先将操作数转换为数字，并将数字强制表示为32位整型，这会忽略原格式中的小数部分和任何超过32位的二进制位。移位运算符要求右操作数在0～31之间。在将其操作数转换为无符号32位整数后，它们将舍弃第5位之后的二进制位，以便生成一个位数正确的数字。需要注意的是，位运算符会将NaN、Infinity和-Infinity都转换为0。

按位与（&）

位运算符“&”对它的整型操作数逐位执行布尔与（AND）操作。只有两个操作数中相对应的位都是1，结果中的这一位才是1。例如，`0x1234 & 0x00FF = 0x0034`。

按位或（|）

位运算符“|”对它的整型操作数逐位执行布尔或（OR）操作。如果其中一个操作数相应的位为1，或者两个操作数相应位都是1，那么结果中的这一位就为1。例如：`0x1234 | 0x00FF = 0x12FF`。

按位异或（^）

位运算符“|”对它的整型操作数逐位执行布尔异或（XOR）操作。异或是指第一个操作数为`true`或第二个操作数为`true`，但两者不能同时为`true`。如果两个操作数中只有一个相应位为1（不能同时为1），那么结果中的这一位就是1。例如，`0xFF00 ^ 0xF0F0 = 0x0FF0`。

按位非（~）

运算符“～”是一元运算符，位于一个整型参数之前，它将操作数的所有位取反。根据JavaScript中带符号的整数的表示方法，对一个值使用“～”运算符相当于改变它的符号并减1。例如，`~0x0F = 0xFFFFFFF0`或－16。

左移（<<）

将第一个操作数的所有二进制位进行左移操作，移动的位数由第二个操作数指定，移动的位数是0～31之间的一个整数。例如，在表达式`a<<1`中，a的第一位变成了第二位，a的第二位变成了它的第三位，以此类推。新的第一位用0来补充，舍弃第32位。将一个值左移1位相当于它乘以2，左移两位相当于乘以4，以此类推。例如，`7<<2=28`。

带符号右移（>>）

运算符“>>”将第一个操作数的所有位进行右移操作，移动的位数由第二个操作数指定，移动的位数是0～31之间的一个整数。右边溢出的位将忽略。填补在左边

的位由原操作数的符号决定，以便保持结果的符号与原操作数一致。如果第一个操作数是正数，移位后用0填补最高位；如果第一个操作数是负的，移位后就用1填补高位。将一个值右移1位，相当于用它除以2（忽略余数），右移两位，相当于它除以4，以此类推，例如，7>>1=3，-7>>1=-4。

无符号右移（>>>）

运算符“>>>”和运算符“>>”一样，只是左边的高位总是填补0，与原来的操作数符号无关，例如，-1>>4=-1，但是-1>>>4=0x0FFFFFFF。

4.9 关系表达式

本节介绍JavaScript的关系运算符。关系运算符用于测试两个值之间的关系（比如“相等”，“小于”，或“是...的属性”），根据关系是否存在而返回true或false。关系表达式总是返回一个布尔值，通常在if、while或者for语句（参照第5章）中使用关系表达式，用以控制程序的执行流程。接下来的几节将会讲述相等和不等运算符、比较运算符和JavaScript中其他两个关系运算符in和instanceof。

4.9.1 相等和不等运算符

“==”和“===”运算符用于比较两个值是否相等，当然它们对相等的定义不尽相同。两个运算符允许任意类型的操作数，如果操作数相等则返回true，否则返回false。“===”也称为严格相等运算符（strict equality）（有时也称做恒等运算符（identity operator）），它用来检测两个操作数是否严格相等。“==”运算符称做相等运算符（equality operator），它用来检测两个操作数是否相等，这里“相等”的定义非常宽松，可以允许进行类型转换。

JavaScript支持“=”、“==”和“===”运算符。你应当理解这些（赋值、相等、恒等）运算符之间的区别，并在编码过程中小心使用。尽管它们都可以称做“相等”，但为了减少概念混淆，应该把“=”称做“得到或赋值”，把“==”称做“相等”，把“===”称做“严格相等”。

“!=”和“!==”运算符的检测规则是“==”和“===”运算符的求反。如果两个值通过“==”的比较结果为true，那么通过“!=”的比较结果则为false。如果两值通过“===”的比较结果为true，那么通过“!==”的比较结果则为false。4.10节会提到，“！”运算符是布尔非运算符。我们只要记住“!=”称做“不相等”、“!==”称做“不严格相等”就可以了。

在3.7节已经提到，JavaScript对象的比较是引用的比较，而不是值的比较。对象和其本

身是相等的，但和其他任何对象都不相等。如果两个不同的对象具有相同数量的属性，相同的属性名和值，它们依然是不相等的。相应位置的数组元素是相等的两个数组也是不相等的。

严格相等运算符“===”首先计算其操作数的值，然后比较这两个值，比较过程没有任何类型转换：

- 如果两个值类型不相同，则它们不相等。
- 如果两个值都是null或者都是undefined，则它们不相等。
- 如果两个值都是布尔值true或都是布尔值false，则它们相等。
- 如果其中一个值是NaN，或者两个值都是NaN，则它们不相等。NaN和其他任何值都是不相等的，包括它本身！通过x!==x来判断x是否为NaN，只有在x为NaN的时候，这个表达式的值才为true。
- 如果两个值为数字且数值相等，则它们相等。如果一个值为0，另一个值为-0，则它们同样相等。
- 如果两个值为字符串，且所含的对应位上的16位数（参照3.2节）完全相等，则它们相等。如果它们的长度或内容不同，则它们不等。两个字符串可能含义完全一样且所显示出的字符也一样，但具有不同编码的16位值。JavaScript并不对Unicode进行标准化的转换，因此像这样的字符串通过“===”和“==”运算符的比较结果也不相等。第三部分的String.localeCompare()提供了另外一种比较字符串的方法。
- 如果两个引用值指向同一个对象、数组或函数，则它们是相等的。如果指向不同的对象，则它们是不等的，尽管两个对象具有完全一样的属性。

相等运算符“==”和恒等运算符相似，但相等运算符的比较并不严格。如果两个操作数不是同一类型，那么相等运算符会尝试进行一些类型转换，然后进行比较：

- 如果两个操作数的类型相同，则和上文所述的严格相等的比较规则一样。如果严格相等，那么比较结果为相等。如果它们不严格相等，则比较结果为不相等。
- 如果两个操作数类型不同，“==”相等操作符也可能会认为它们相等。检测相等将会遵守如下规则和类型转换：
  - 如果一个值是null，另一个是undefined，则它们相等。
  - 如果一个值是数字，另一个是字符串，先将字符串转换为数字，然后使用转换后的值进行比较。
  - 如果其中一个值是true，则将其转换为1再进行比较。如果其中一个值是false，则将其转换为0再进行比较。

— 如果一个值是对象，另一个值是数字或字符串，则使用3.8.3节所提到的转换规则将对象转换为原始值，然后再进行比较。对象通过toString()方法或者valueOf()方法转换为原始值。JavaScript语言核心的内置类首先尝试使用valueOf()，再尝试使用toString()，除了日期类，日期类只使用toString()转换。那些不是JavaScript语言核心中的对象则通过各自的实现中定义的方法转换为原始值。

— 其他不同类型之间的比较均不相等。

这里有一个判断相等的小例子：

```
"1"==true
```

这个表达式的结果是true，这表明完全不同类型的值比较结果为相等。布尔值true首先转换为数字1，然后再执行比较。接下来，字符串“1”也转换为了数字1，因为两个数字的值相等，因此比较结果为true。

4.9.2 比较运算符

比较运算符用来检测两个操作数的大小关系（数值大小或者字母表的顺序）：

小于（<）

如果第一个操作数小于第二个操作数，则“<”运算符的计算结果为true，否则为false。

大于（>）

如果第一个操作数大于第二个操作数，则“>”运算符的计算结果为true，否则为false。

小于等于(<=)

如果第一个操作数小于或者等于第二个操作数，则“<=”运算符的计算结果为true，否则为false。

大于等于(>=)

如果第一个操作数大于或者等于第二个操作数，则“>=”运算符的计算结果为false，否则为false。

比较操作符的操作数可能是任意类型。然而，只有数字和字符串才能真正执行比较操作，因此那些不是数字和字符串的操作数都将进行类型转换，类型转换规则如下：

- 如果操作数为对象，那么这个对象将依照3.8.3节结尾处所描述的转换规则转换为

原始值：如果valueOf()返回一个原始值，那么直接使用这个原始值。否则，使用toString()的转换结果进行比较操作。

- 在对象转换为原始值之后，如果两个操作数都是字符串，那么将依照字母表的顺序对两个字符串进行比较，这里提到的“字母表顺序”是指组成这个字符串的16位Unicode字符的索引顺序。
- 在对象转换为原始值之后，如果至少有一个操作数不是字符串，那么两个操作数都将转换为数字进行数值比较。0和-0是相等的。Infinity比其他任何数字都大（除了Infinity本身），-Infinity比其他任何数字都小（除了它自身）。如果其中一个操作数是（或转换后是）NaN，那么比较操作符总是返回false。

需要注意的是，JavaScript字符串是一个由16位整数值组成的序列，字符串的比较也只是两个字符串中的字符的数值比较。由Unicode定义的字符编码顺序和任何特定语言或者本地语言字符集中的传统字符编码顺序不尽相同。注意，字符串比较是区分大小写的，所有的大写的ASCII字母都“小于”小写的ASCII字母。如果不注意这条不起眼的规则的话会造成一些小麻烦。比如，使用“<”小于运算符比较“Zoo”和“aardvark”，结果为true。

参照String.localCompare()方法来获取更多字符串比较的相关信息，String.localCompare()方法更加健壮可靠，这个方法参照本地语言的字母表定义的字符次序。对于那些不区分字母大小写的比较来说，则需要首先将字符串转全部换为小写字母或者大写字母，通过String.toLowerCase()和String.toUpperCase()做大小写的转换。

对于数字和字符串操作符来说，加号运算符和比较运算符的行为都有所不同，前者更偏爱字符串，如果它的其中一个操作数是字符串的话，则进行字符串连接操作。而比较运算符则更偏爱数字，只有在两个操作数都是字符串的时候，才会进行字符串的比较：

```
1 + 2         // 加法，结果是3
"1" + "2"     // 字符串连接，结果是"12"
"1" + 2       // 字符串链接，2转换为"2"，结果是"12"
11 < 3        // 数字的比较，结果为false
"11" < "3"    // 字符串比较，结果为true
"11" < 3      // 数字的比较，"11"转换为11，结果为false
"one" < 3     // 数字的比较，"one"转换为NaN，结果为false
```

最后，需要注意的是，“<=”（小于等于）和“>=”（大于等于）运算符在判断相等的时候，并不依赖于相等运算符和严格相等运算符的比较规则。相反，小于等于运算符只是简单的“不大于”，大于等于运算符也只是“不小于”。只有一个例外，那就是当其一个操作数是（或者转换后是）NaN的时候，所有4个比较运算符均返回false。

4.9.3 in运算符

in运算符希望它的左操作数是一个字符串或可以转换为字符串，希望它的右操作数是一个对象。如果右侧的对象拥有一个名为左操作数值的属性名，那么表达式返回true，例如：

```
var point = { x:1, y:1 };      // 定义一个对象
"x" in point                   // => true:对象有一个名为"x"的属性
"z" in point                   // => false:对象中不存在名为"z"的属性
"toString" in point            // => true:对象继承了toString()方法

var data = [7,8,9];            // 拥有三个元素的数组
"0" in data                    // => true:数组包含元素"0"
1 in data                      // => true:数字转换为字符串
3 in data                      // => false:没有索引为3的元素
```

4.9.4 instanceof 运算符

instanceof运算符希望左操作数是一个对象，右操作数标识对象的类。如果左侧的对象是右侧类的实例，则表达式返回true；否则返回false。第9章将会讲到，JavaScript中对象的类是通过初始化它们的构造函数来定义的。这样的话，instanceof的右操作数应当是一个函数。比如：

```
var d = new Date();   // 通过Date()构造函数来创建一个新对象
d instanceof Date;    // 计算结果为true，d是由Date()创建的
d instanceof Object;  // 计算结果为true，所有的对象都是Object的实例
d instanceof Number;  // 计算结果为false，d不是一个Number对象
var a = [1, 2, 3];    // 通过数组直接量的写法创建一个数组
a instanceof Array;   // 计算结果为true，a是一个数组
a instanceof Object;  // 计算结果为true，所有的数组都是对象
a instanceof RegExp;  // 计算结果为false，数组不是正则表达式
```

需要注意的是，所有的对象都是Object的实例。当通过instanceof判断一个对象是否是一个类的实例的时候，这个判断也会包含对“父类”（superclass）的检测。如果instanceof的左操作数不是对象的话，instanceof返回false。如果右操作数不是函数，则抛出一个类型错误异常。

为了理解instanceof运算符是如何工作的，必须首先理解“原型链”（prototype chain）。原型链作为JavaScript的继承机制，将在6.2.2节详细讲述。为了计算表达式o instanceof f，JavaScript首先计算f.prototype，然后在原型链中查找o，如果找到，那么o是f（或者f的父类）的一个实例，表达式返回true。如果f.prototype不在o的原型链中的话[译注3]，那么o就不是f的实例，instanceof返回false。

译注3：对象o中存在一个隐藏的成员，这个成员指向其父类的原型，如果父类的原型是另外一个类的实例的话，则这个原型对象中也存在一个隐藏成员指向另外一个类的原型，这种链条将许多对象或类串接起来，既是原型链。原文所讲f.prototype不在o的原型链中也就是说f和o没有派生关系，更多细节请参照6.2.2节。

4.10 逻辑表达式

逻辑运算符“&&”、“||”和“!”是对操作数进行布尔算术运算，经常和关系运算符一起配合使用，逻辑运算符将多个关系表达式组合起来组成一个更复杂的表达式。这些运算符在下面几节中会一一讲述，为了更好地理解它们，应当首先回顾一下3.3节提到的“真值”和“假值”的概念。

4.10.1 逻辑与(&&)

“&&”运算符可以从三个不同的层次进行理解。最简单的第一层理解是，当操作数都是布尔值的时候，“&&”对两个值执行布尔与（AND）操作，只有在第一个操作数和第二个操作数都是true的时候，它才返回true。如果其中一个操作数是false，它返回false。

“&&”常用来连接两个关系表达式：

```
x == 0 && y == 0 // 只有在x和y都是0的时候，才返回true
```

关系表达式的运算结果总是为true或false，因此当这样使用的时候，“&&”运算符本身也返回true或false。关系运算符的优先级比“&&”（和“||”）要高，因此类似这种表达式可以放心地书写，而不用补充圆括号。

但是“&&”的操作数并不一定是布尔值，回想一下，有些值可以当做“真值”和“假值”（参照3.3节，假值是false、null、undefined、0、-0、NaN和""，所有其他的值包括所有对象都是真值）。对“&&”的第二层理解是，“&&”可以对真值和假值进行布尔与（AND）操作。如果两个操作数都是真值，那么返回一个真值；否则，至少一个操作数是假值的话，则返回一个假值。在JavaScript中任何希望使用布尔值的地方，表达式和语句都会将其当做真值或假值来对待，因此实际上“&&”并不总是返回true和false，但也并无大碍。

需要注意的是，上文提到了运算符返回一个“真值”或者“假值”，但并没有说明这个“真值”或者“假值”到底是什么值。为此，我们深入讨论对“&&”的第三层（也是最后一层）理解。运算符首先计算左操作数的值，即首先计算“&&”左侧的表达式。如果计算结果是假值，那么整个表达式的结果一定也是假值，因此“&&”这时简单地返回左操作数的值，而并不会对右操作数进行计算。

反过来讲，如果左操作数是真值，那么整个表达式的结果则依赖于右操作数的值。如果右操作数是真值，那么整个表达式的值一定是真值；如果右操作数是假值，那么整个表达式的值一定是假值。因此，当左操作数是真值时，“&&”运算符将计算右操作数的值并将其返回作为整个表达式的计算结果：

```
var o = { x : 1 };
var p = null;
o && o.x     // =>1:o 是真值，因此返回值为o.x
p && p.x     //=>null: p是假值，因此将其返回，而并不去计算p.x
```

这对于理解“&&”可能不会去计算右操作数的情况至关重要，在上述示例代码中，变量p的值是null，而如果计算表达式p.x的话则会抛出一个类型错误异常。但是示例代码使用了“&&”的一种符合语言习惯的用法，因此只有在p为真值（不能是null或者undefined）的情况下才会计算p.x。

“&&”的行为有时称做“短路”（short circuiting），我们也会经常看到很多代码利用了这一特性来有条件地执行代码。例如，下面两行JavaScript代码是完全等价的：

```
if (a == b) stop();   //只有在a==b的时候才调用stop()
(a == b) && stop();   //同上
```

一般来讲，当“&&”右侧的表达式具有副作用的时候（赋值、递增、递减和函数调用表达式）要格外小心。因为这些带有副作用的表达式的执行依赖于左操作数的计算结果。

尽管“&&”可以按照第二层和第三层的理解进行一些复杂表达式运算，但大多数情况下，“&&”仅用来对真值和假值做布尔计算。

4.10.2 逻辑或（||）

“||”运算符对两个操作数做布尔或（OR）运算。如果其中一个或者两个操作数是真值，它返回一个真值。如果两个操作数都是假值，它返回一个假值。

尽管“||”运算符大多数情况下只是做简单布尔或（OR）运算，和“&&”一样，它也具有一些更复杂的行为。它会首先计算第一个操作数的值，也就是说会首先计算左侧的表达式。如果计算结果为真值，那么返回这个真值。否则，再计算第二个操作数的值，即计算右侧的表达式，并返回这个表达式的计算结果。

和“&&”运算符一样，同样应当避免右操作数包含一些具有副作用的表达式，除非你目地明确地在右侧使用带副作用的表达式，而有可能不会计算右侧的表达式。

这个运算符最常用的方式是用来从一组备选表达式中选出第一个真值表达式：

```
// 如果max_width已经定义了，直接使用它；否则在preferences对象中查找max_width
// 如果没有定义它，则使用一个写死的常量
var max = max_width || preferences.max_width || 500;
```

这种惯用法通常用在函数体内，用来给参数提供默认值：

```
// 将o的成员属性复制到p中，并返回p
function copy(o, p) {
    p = p || {}; //如果向参数p没有传入任何对象，则使用一个新创建的对象
    // 函数体内的主逻辑
}
```

4.10.3 逻辑非（!）

“!”运算符是一元运算符。它放置在一个单独的操作数之前。它的目的是将操作数的布尔值进行求反。例如，如果x是真值，则!x返回false；如果x是假值，则!x返回true。

和“&&”与“||”运算符不同，“!”运算符首先将其操作数转换为布尔值（参照第3章讲述的转换规则），然后再对布尔值求反。也就是说“!”总是返回true或者false，并且，可以通过使用两次逻辑非运算来得到一个值的等价布尔值：!!x（参照3.8.2节）。

作为一个一元运算符，“!”具有很高的优先级，并且和操作数紧密绑定在一起。如果你希望对类似p && q的表达式做求反操作，则需要使用圆括号：!(p && q)。布尔计算的更多原理性知识不必要做过多的解释，这里仅用JavaScript代码做简单说明：

```
// 对于p和q取任意值，这两个等式都永远成立
!(p && q) === !p || !q
!(p || q) === !p && !q
```

4.11 赋值表达式

JavaScript使用“=”运算符来给变量或者属性赋值。例如：

```
i = 0        // 将变量i设置为0
o.x = 1      //将对象o的属性x设置为1
```

“=”运算符希望它的左操作数是一个左值：一个变量或者对象属性（或数组元素）。它的右操作数可以是任意类型的任意值。赋值表达式的值就是右操作数的值。赋值表达式的副作用是，右操作数的值赋值给左侧的变量或对象属性，这样的话，后续对这个变量和对象属性的引用都将得到这个值。

尽管赋值表达式通常非常简单，但有时仍会看到一些复杂表达式包含赋值表达式的情况。例如，可以将赋值和检测操作放在一个表达式中，就像这样：

```
(a = b) == 0
```

如果这样做的话，应当清楚地知道“=”和“==”运算符之间的区别！需要注意的是，

“=”具有非常低的优先级，通常在一个较长的表达式中用到了一条赋值语句的值的时候，需要补充圆括号以保证正确的运算顺序。

赋值操作符的结合性是从右至左，也就是说，如果一个表达式中出现了多个赋值运算符，运算顺序是从右到左。因此，可以通过如下的方式来对多个变量赋值：

```
i=j=k=0;      //把三个变量初始化为0
```

带操作的赋值运算

除了常规的赋值运算“=”之外，JavaScript还支持许多其他的赋值运算符，这些运算符将赋值运算符和其他运算符连接起来，提供一种更为快捷的运算方式。例如，运算符“+=”执行的是加法运算和赋值操作，下面的表达式：

```
total += sales_tax
```

和接下来的表达式是等价的：

```
total = total + sales_tax
```

运算符“+=”可以作用于数字或字符串，如果其操作数是数字，它将执行加法运算和赋值操作；如果操作数是字符串，它就执行字符串连接操作和赋值操作。

这类运算符还包括“-=”、“*=”、“&=”等。表4-2列出了这一类的所有运算符。

表4-2：赋值运算符

运算符	示例	等价于
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b
<<=	a<<=b	a=a<<b
>>=	a>>=b	a=a>>b
>>>=	a>>>=b	a=a>>>b
&=	a&=b	a=a&b
|=	a|=b	a=a|b
^=	a^=b	a=a^b

在大多数情况下，表达式为：

```
a op= b
```

这里op代表一个运算符，这个表达式和下面的表达式等价：

```
a=a op b
```

在第一行中，表达式a计算了一次，在第二行中，表达式a计算了两次。只有在a包含具有副作用的表达式（比如函数调用和赋值操作）的时候，两者才不等价。比如，下面两个表达式就不等价：

```
data[i++] *= 2;
data[i++] = data[i++] * 2;
```

4.12 表达式计算

和其他很多解释性语言一样，JavaScript同样可以解释运行由JavaScript源代码组成的字符串，并产生一个值。JavaScript通过全局函数eval()来完成这个工作：

```
eval("3+2")   //=> 5
```

动态判断源代码中的字符串是一种强大的语言特性，几乎没有必要在实际中应用。如果你使用了eval()，你应当仔细考虑是否真的需要使用它。

下面讲解eval()的基础用法，并且介绍严格使用它的两种方法，从代码优化的角度讲，这两种方法对于原有代码造成的影响是最小的。

eval()是一个函数还是一个运算符

eval()是一个函数，但由于它已经被当成运算符来对待了，因此将它放在本章来讲述。JavaScript语言的早期版本定义了eval()函数，从那时起，该语言的设计者和解释器的作者对其实施了更多限制，使其看起来更像运算符。现代JavaScript解释器进行了大量的代码分析和优化。而eval()的问题在于，用于动态执行的代码通常来讲是不能分析。一般来讲，如果一个函数调用了eval()，那么解释器将无法对这个函数做进一步优化。而将eval()定义为函数的另一个问题是，它可以被赋予其他的名字：

```
var f = eval;
var g = f;
```

如果允许这种情况的话，那么解释器将无法放心地优化任何调用g()的函数。而当eval是一个运算符（并作为一个保留字）的时候，这种问题就可以避免掉。接下来的4.12.2节 和4.12.3节将会介绍如何对eval()实施更多的限制，以便让它的行为更接近运算符。

4.12.1 eval()

eval()只有一个参数。如果传入的参数不是字符串，它直接返回这个参数。如果参数是字符串，它会把字符串当成JavaScript代码进行编译（parse）[译注4]，如果编译失败则抛出一个语法错误（SyntaxError）异常。如果编译成功，则开始执行这段代码，并返回字符串中的最后一个表达式或语句的值，如果最后一个表达式或语句没有值，则最终返回undefined。如果字符串抛出一个异常，这个异常将把该调用传递给eval()[译注5]。

关于eval()最重要的是，它使用了调用它的变量作用域环境。也就是说，它查找变量的值和定义新变量和函数的操作和局部作用域中的代码完全一样。如果一个函数定义了一个局部变量x，然后调用eval("x")，它会返回局部变量的值。如果它调用eval("x=1")，它会改变局部变量的值。如果函数调用了eval("var y = 3; ")，它声明一个新的局部变量y。同样地，一个函数可以通过如下代码声明一个局部函数：

```
eval("function f() { return x+1; }");
```

如果在最顶层代码中调用eval()，当然，它会作用于全局变量和全局函数。

需要注意的是，传递给eval()的字符串必须在语法上讲的通——不能通过eval()往函数中任意粘贴代码片段，比如，eval("return; ")是没有意义的，因为return只有在函数中才起作用，并且事实上，eval的字符串执行时的上下文环境和调用函数的上下文环境是一样的，这不能使其作为函数的一部分来运行[译注6]。如果字符串作为一个单独的脚本

译注4：这里的原文是parse，意思是“解析”这段字符串，更精确地讲，应该是“编译”这段字符串，编译不包括代码的执行。

译注5：原文有误，已修改。

译注6：比如这段代码：

```
var foo = function(a){
    eval(a);
};
foo("return; ");
```

按照原文的意思，这段代码中执行eval(a)的上下文是全局的，在全局上下文中使用return会抛出语法错误:return not in function。

是有语义的（就像诸如x=0的短代码），那么将其传递给eval()作参数是完全没有问题的，否则，eval()将抛出语法错误异常[译注7]。

4.12.2 全局eval()

eval()具有更改局部变量的能力，这对于JavaScript优化器来说是一个很大的问题。然而作为一种权宜之计，JavaScript解释器针对那些调用了eval()的函数所做的优化并不多。但当脚本定义了eval()的一个别名，且用另一个名称调用它，JavaScript解释器又会如何工作呢？为了让JavaScript解释器的实现更加简化，ECMAScript 3标准规定了任何解释器都不允许对eval()赋予别名。如果eval()函数通过别名调用的话，则会抛出一个EvalError异常。

实际上，大多数的实现并不是这么做的。当通过别名调用时，eval()会将其字符串当成顶层的全局代码来执行。执行的代码可能会定义新的全局变量和全局函数，或者给全局变量赋值，但却不能使用或修改主调函数中的局部变量，因此，这不会影响到函数内的代码优化。

ECMAScript 5是反对使用EvalError的，并且规范了eval()的行为。“直接的eval”，当直接使用非限定的“eval”名称（eval看起来像是一个保留字）来调用eval()函数时，通常称为“直接eval”（direct eval）。直接调用eval()时，它总是在调用它的上下文作用域内执行。其他的间接调用则使用全局对象作为其上下文作用域，并且无法读、写、定义局部变量和函数。下面有一段示例代码：

```
var geval = eval;                       // 使用别名调用eval将是全局eval
var x = "global", y = "global";         // 两个全局变量
function f() {                          // 函数内执行的是局部eval
    var x = "local";                    // 定义局部变量
    eval("x += 'changed';");            // 直接eval更改了局部变量的值
    return x;                           // 返回更改后的局部变量
}
function g() {                          // 这个函数内执行了全局eval
    var y = "local";                    // 定义局部变量
    geval("y += 'changed';");           // 间接调用改变了全局变量的值
    return y;                           // 返回未更改的局部变量
}
console.log(f(), x);                    // 更改了局部变量：输出"local changed global":
console.log(g(), y);                    // 更改了全局变量：输出"local globalchanged":
```

我们注意到，全局eval的这些行为不仅仅是出于代码优化器的需要而做出的一种折中方案，它实际上是一种非常有用的特性，它允许我们执行那些对上下文没有任何依赖的

译注7：这里是指那些没有语义的代码片段通过eval()执行都会抛出语法错误异常。

全局脚本代码段。我们在本节开始处也提到，真正需要eval来执行代码段的场景并不多见。但当你真的意识到它的必要性时，你更可能会使用全局eval而不是局部eval。

IE 9之前的早期版本IE和其他浏览器有所不同，当通过别名调用`eval()`时并不是全局`eval()`（它也不会抛出一个EvalError异常，仅仅将其当做局部eval来调用）。但IE的确定义了一个名叫`execScript()`的全局函数来完成全局eval的功能（但和`eval()`稍有不同，`execScript()`总是会返回`null`）。

4.12.3 严格eval()

ECMAScript 5严格模式（参照5.7.3节）对`eval()`函数的行为施加了更多的限制，甚至对标识符eval的使用也施加了限制。当在严格模式下调用`eval()`时，或者`eval()`执行的代码段以“use strict”指令开始，这里的`eval()`是私有上下文环境中的局部eval。也就是说，在严格模式下，eval执行的代码段可以查询或更改局部变量，但不能在局部作用域中定义新的变量或函数。

此外，严格模式将“eval”列为保留字，这让`eval()`更像一个运算符。不能用一个别名覆盖`eval()`函数。并且变量名、函数名、函数参数或者异常捕获的参数都不能取名为“eval”。

4.13 其他运算符

JavaScript支持很多其他各种各样的运算符，后续几节详细讨论它们：

4.13.1 条件运算符(?:)

条件运算符是JavaScript中唯一的一个三元运算符（三个操作数），有时直接称做“三元运算符”。通常这个运算符写成“?:”，当然在代码中往往不会这么简写，因为这个运算符拥有三个操作数，第一个操作数在“?”之前，第二个操作数在“?”和“:”之间，第三个操作数在“:”之后，例如：

```
x > 0 ? x : -x // 求x的绝对值
```

条件运算符的操作数可以是任意类型。第一个操作数当成布尔值，如果它是真值，那么将计算第二个操作数，并返回其计算结果。否则，如果第一个操作数是假值，那么将计算第三个操作数，并返回其计算结果。第二个和第三个操作数总是会计算其中之一，不可能两者同时执行。

其实使用`if`语句也会带来同样的效果（参照5.4.1节），“?:”运算符只是提供了一种简

写形式。这里是一个“?:”的典型应用场景，判断一个变量是否有定义（并拥有一个有意义的真值[译注8]），如果有定义则使用它，如果无定义则使用一个默认值：

```
greeting = "hello " + (username ? username : "there");
```

这和下面使用if语句的代码是等价的，但显然上面的代码更加简洁：

```
greeting = "hello ";
if (username)
    greeting += username;
else
    greeting += "there";
```

4.13.2 typeof运算符

typeof是一元运算符，放在其单个操作数的前面，操作数可以是任意类型。返回值为表示操作数类型的一个字符串。表4-3列出了任意值在typeof运算后的返回值：

表4-3：任意值在typeof运算后的返回值

x	typeof x
undefined	"undefined"
null	"object"
true或false	"boolean"
任意数字或NaN	"number"
任意字符串	"string"
任意函数	"function"
任意内置对象（非函数）	"object"
任意宿主对象	由编译器各自实现的字符串，但不是"undefined"、"boolean"、"number"或"string"

typeof最常用的用法是写在表达式中，就像这样：

```
(typeof value == "string") ? "'" + value + "'" : value
```

typeof运算符同样在switch语句（见5.4.3节）中非常有用，需要注意的是，typeof运算符可以带上圆括号，这让typeof看起来像一个函数名，而不是一个运算符关键字：

```
typeof(i)
```

我们注意到，当操作数是null的时候，typeof将返回"object"。如果想将null和对象

译注8：这里的场景其实不包括如果变量已经定义且值为false的情况。

区分开，则必须针对特殊值显式检测。对于宿主对象来说，typeof有可能并不返回“object”，而返回字符串。但实际上客户端JavaScript中的大多数宿主对象都是“object”类型。

由于所有对象和数组的typeof运算结果是“object”而不是“function”，因此它对于区分对象和其他原始值来说是很有帮助的。如果想区分对象的类，则需要使用其他的手段，比如使用instanceof运算符（参照4.9.4节）、*class*特性（参照6.8.2节）以及constructor属性（参照6.8.1 节和§9.2.2节）。

尽管JavaScript中的函数是对象的一种，但typeof运算符还是将函数特殊对待，对函数做typeof运算有着特殊的返回值。在JavaScript中，函数和“可执行的对象”（callable object）有着微妙的区别。所有的函数都是可执行的（callable），但是对象也有可能是可执行的，可以像调用函数一样调用它，但它并不是一个真正的函数。根据ECMAScript 3规范，对于所有内置可执行对象，typeof运算符一律返回“function”。ECMAScript 5规范则扩充至所有可执行对象，包括内置对象（native object）和宿主对象（host object），所有可执行对象进行typeof运算都将返回“function”。大多数浏览器厂商也将JavaScript的原生函数对象（native function object）当成它们的宿主对象的方法来使用。但微软却一直将非原生可执行对象（non-native callable object）当成其客户端的方法来使用，在IE 9之前的版本中，非原生可执行对象的typeof运算将返回“object”，尽管它们的行为和函数非常相似。而在IE 9中，这些客户端方法是真正的内置函数对象（native function object）。要了解真正的函数和可执行对象之间的详细差别请参照8.7.7节。

4.13.3 delete运算符

delete是一元操作符，它用来删除对象属性或者数组元素[注1]。就像赋值、递增、递减运算符一样，delete也是具有副作用的，它是用来做删除操作的，不是用来返回一个值的，例如：

```
var o = { x: 1, y: 2};          // 定义一个对象
delete o.x;                     // 删除一个属性
"x" in o                        // => false:这个属性在对象中不再存在

var a = [1,2,3];                // 定义一个数组
delete a[2];                    // 删除最后一个数组元素
2 in a;                         // => false:元素2在数组中已经不存在了
```

注1：如果你是C++程序员，请注意JavaScript中的delete和C++中的delete是完全不同的。在JavaScript中，内存的回收是通过垃圾回收自动回收的，你不用担心内存的显式释放问题，这样则完全不用像C++那样通过delete来删除整个对象。

```
a.length                    // => 3:注意，数组长度并没有改变，尽管上一行
                               代码删除了这个元素，但删除操作留下了一个“洞”，实际
                               上并没有修改数组的长度，因此a数组的长度仍然是3
```

需要注意的是，删除属性或者删除数组元素不仅仅是设置了一个undefined的值。当删除一个属性时，这个属性将不再存在。读取一个不存在的属性将返回undefined，但是可以通过in运算符（见4.9.3节）来检测这个属性是否在对象中存在。

delete希望他的操作数是一个左值，如果它不是左值，那么delete将不进行任何操作同时返回true。否则，delete将试图删除这个指定的左值。如果删除成功，delete将返回true。然而并不是所有的属性都可删除，一些内置核心和客户端属性是不能删除的，用户通过var语句声明的变量不能删除。同样，通过function语句定义的函数和函数参数也不能删除。

在ECMAScript 5严格模式中，如果delete的操作数是非法的，比如变量、函数或函数参数，delete操作将抛出一个语法错误（SyntaxError）异常，只有操作数是一个属性访问表达式（见4.4节）的时候它才会正常工作。在严格模式下，delete删除不可配置的属性（参照 6.7节）时会抛出一个类型错误异常。在非严格模式下，这些delete操作都不会报错，只是简单地返回false，以表明操作数不能执行删除操作。

这里有一些关于delete运算符的例子：

```
var o = {x:1, y:2};   //定义一个变量，初始化为对象
delete o.x;           //删除一个对象属性，返回true
typeof o.x;           //属性不存在，返回"undefined"
delete o.x;           //删除不存在的属性，返回true
delete o;             //不能删除通过var声明的变量，返回false
                      //在严格模式下，将抛出一个异常
delete 1;             //参数不是一个左值，返回true
this.x = 1;           //给全局对象定义一个属性，这里没有使用var
delete x;             //试图删除它，在非严格模式下返回true
                      //在严格模式下会抛出异常，这时使用"delete this.x"来代替
    x;                //运行时错误，没有定义x
```

6.3节还会有关于delete操作符的讨论。

4.13.4 void运算符

void是一元运算符，它出现在操作数之前，操作数可以是任意类型。这个运算符并不是经常使用：操作数会照常计算，但忽略计算结果并返回undefined。由于void会忽略操作数的值，因此在操作数具有副作用的时候使用void来让程序更具语义。

这个运算符最常用在客户端的URL——javascript:URL中，在URL中可以写带有副作用的

表达式，而void则让浏览器不必显示这个表达式的计算结果。例如，经常在HTML代码中的<a>标签里使用void运算符：

```
<a href="javascript:void window.open();">打开一个新窗口</a>
```

通过给<a>的onclick绑定一个事件处理程序要比在href中写“javascript:URL”要更加清晰，当然，这样的话void操作符就可有可无了。

4.13.5 逗号运算符(,)

逗号运算符是二元运算符，它的操作数可以是任意类型。它首先计算左操作数，然后计算右操作数，最后返回右操作数的值，看下面的示例代码：

```
i=0, j=1, k=2;
```

计算结果是2，它和下面的代码基本上是等价的：

```
i = 0; j = 1; k = 2;
```

总是会计算左侧的表达式，但计算结果忽略掉，也就是说，只有左侧表达式具有副作用，才会使用逗号运算符让代码变得更通顺。逗号运算符最常用的场景是在for循环中（见5.5.3节），这个for循环通常具有多个循环变量：

```
//for循环中的第一个逗号是var语句的一部分
//第二个逗号是逗号运算符
//它将两个表达式（i++和j--）放在一条（for循环中的）语句中
for(var i=0,j=10; i < j; i++,j--)
    console.log(i+j);
```