JavaScript还定义了另一种特殊对象——函数。函数是具有与它相关联的可执行代码的对象，通过调用函数来运行可执行代码，并返回运算结果。和数组一样，函数的行为特征和其他对象都不一样。JavaScript为使用函数定义了专用语法。对于JavaScript函数来讲，最重要的是，它们都是真值，并且JavaScript可以将它们当做普通对象来对待。第8章会专门讲述函数。

如果函数用来初始化（使用new运算符）一个新建的对象，我们称之为构造函数（constructor）。每个构造函数定义了一类（class）对象——由构造函数初始化的对象组成的集合。类可以看做是对象类型的子类型。除了数组（Array）类和函数（Function）类之外，JavaScript语言核心定义了其他三种有用的类。日期（Date）类定义了代表日期的对象。正则（RegExp）类定义了表示正则表达式（一种强大的模式匹配工具，在第10章会讲到）的对象。错误（Error）类定义了那些表示JavaScript程序中运行时错误和语法错误的对象。可以通过定义自己的构造函数来定义需要的类。这会在第9章讲述。

JavaScript解释器有自己的内存管理机制，可以自动对内存进行垃圾回收（garbage collection）。这意味着程序可以按需创建对象，程序员则不必担心这些对象的销毁和内存回收。当不再有任何引用指向一个对象，解释器就会知道这个对象没用了，然后自动回收它所占用的内存资源。

JavaScript是一种面向对象的语言。不严格地讲，这意味着我们不用全局的定义函数去操作不同类型的值，数据类型本身可以定义方法（method）来使用值。例如，要对数组a中的元素进行排序，不必要将a传入sort()函数，而是调用a的一个方法sort()：

```
a.sort(); // sort(a)的面向对象的版本
```

第9章将会讲述方法的定义。从技术上讲，只有JavaScript对象才能拥有方法。然而，数字、字符串和布尔值也可以拥有自己的方法（3.6节解释其工作机制）。在JavaScript中，只有null和undefined是无法拥有方法的值。

JavaScript的类型可以分为原始类型和对象类型，也可分为可以拥有方法的类型和不能拥有方法的类型，同样可分为可变（mutable）类型和不可变（immutable）类型。可变类型的值是可修改的。对象和数组属于可变类型：JavaScript程序可以更改对象属性值和数组元素的值。数字、布尔值、null和undefined属于不可变类型——比如，修改一个数值的内容本身就说不通。字符串可以看成由字符组成的数组，你可能会认为它是可变的。然而在JavaScript中，字符串是不可变的：可以访问字符串任意位置的文本，但JavaScript并未提供修改已知字符串的文本内容的方法。3.7节会详细讲解可变类型和不可变类型的不同之处。

JavaScript可以自由地进行数据类型转换。比如，如果在程序期望使用字符串的地方使用了数字，JavaScript会自动将数字转换为字符串。如果在期望使用布尔值的地方使用了非布尔值，JavaScript也会进行相应的转换。类型转换规则将在3.8节讲述。JavaScript中灵活的类型转换规则对“判断相等”（equality）的定义亦有影响。等号运算符“==”所进行的类型转换细节将在3.8.1节详细描述。

JavaScript变量是无类型的(untyped)，变量可以被赋予任何类型的值，同样一个变量也可以重新赋予不同类型的值。使用var关键字来声明（declare）变量。JavaScript采用词法作用域（lexical scoping）。不在任何函数内声明的变量称做全局变量（global variable），它在JavaScript程序中的任何地方都是可见的。在函数内声明的变量具有函数作用域（function scope），并且只在函数内可见。变量声明和作用域将会在3.9节和3.10节详细讲解。

3.1 数字

和其他编程语言[译注1]不同，JavaScript不区分整数值和浮点数值。JavaScript中的所有数字均用浮点数值表示。JavaScript采用IEEE 754标准[注1]定义的64位浮点格式表示数字，这意味着它能表示的最大值是 $\pm 1.7976931348623157 \times 10^{308}$，最小值是 $\pm 5 \times 10^{-324}$。

按照JavaScript中的数字格式，能够表示的整数范围是从 −9 007 199 254 740 992～9 007 199 254 740 992 (即 $-2^{53} \sim 2^{53}$)，包含边界值。如果使用了超过此范围的整数，则无法保证低位数字的精度。然而需要注意的是，JavaScript中实际的操作（比如数组索引，以及第4章讲到的位操作符）则是基于32位整数。

当一个数字直接出现在JavaScript程序中，我们称之为数字直接量（numeric literal）。JavaScript支持多种格式的数字直接量，在接下来的小节中会有讨论。注意，在任何数字直接量前添加负号（−）可以得到它们的负值。但负号是一元求反运算符（参见第4章），并不是数字直接量语法的组成部分。

3.1.1 整型直接量

在JavaScript程序中，用一个数字序列表示一个十进制整数。例如：

```
0
3
10000000
```

译注1：例如C和Java。

注1：Java程序员应该很熟悉这种格式，就像他们熟悉双精度（double）类型一样。在C和C++的所有现代实现中也都用到了双精度类型。

除了十进制的整型直接量，JavaScript同样能识别十六进制（以16为基数）值。所谓十六进制的直接量是指以“0x”或“0X”为前缀，其后跟随十六进制数串的直接量。十六进制值是0～9之间的数字和a（A）～f（F）之间的字母构成，a～f的字母对应的表示数字10～15。下面是十六进制整型直接量的例子：

```
0xff // 15*16 + 15 = 255 (十进制)
0xCAFE911
```

尽管ECMAScript标准不支持八进制直接量，但JavaScript的某些实现可以允许采用八进制（基数为8）形式表示整数。八进制直接量以数字0开始，其后跟随一个由0～7（包括0和7）之间的数字组成的序列，例如：

```
0377 // 3*64 + 7*8 + 7 = 255 (十进制)
```

由于某些JavaScript的实现支持八进制直接量，而有些不支持，因此最好不要使用以0为前缀的整型直接量，毕竟我们也无法得知当前JavaScript的实现是否支持八进制的解析。在ECMAScript 6（见5.7.3节）的严格模式下，八进制直接量是明令禁止的。

3.1.2 浮点型直接量

浮点型直接量可以含有小数点，它们采用的是传统的实数写法。一个实数由整数部分、小数点和小数部分组成。

此外，还可以使用指数记数法表示浮点型直接量，即在实数后跟字母e或E，后面再跟正负号，其后再加一个整型的指数。这种记数方法表示的数值，是由前面的实数乘以10的指数次幂。

可以使用更简洁的语法表示：

```
[digits][.digits][(E|e)[(+|-)]digits]
```

例如：

```
3.14
2345.789
.333333333333333333
6.02e23       // 6.02 × 10^23
1.4738223E-32 // 1.4738223 × 10^-32
```

3.1.3 JavaScript中的算术运算

JavaScript程序是使用语言本身提供的算术运算符来进行数字运算的。这些运算符包括加法运算符（+）、减法运算符（－）、乘法运算符（*）、除法运算符（/）和求余（求整除后的余数）运算符（%）。第4章将详细介绍这些以及更多的运算符。

除了基本的运算符外，JavaScript还支持更加复杂的算术运算，这些复杂运算通过作为Math对象的属性定义的函数和常量来实现：

```
Math.pow(2,53)              // => 9007199254740992: 2 的 53次幂
Math.round(.6)              // => 1.0: 四舍五入
Math.ceil(.6)               // => 1.0: 向上求整
Math.floor(.6)              // => 0.0: 向下求整
Math.abs(-5)                // => 5: 求绝对值
Math.max(x,y,z)             // 返回最大值
Math.min(x,y,z)             // 返回最小值
Math.random()               // 生成一个大于等于0小于1.0的伪随机数
Math.PI                     // π: 圆周率
Math.E                      // e: 自然对数的底数
Math.sqrt(3)                // 3的平方根
Math.pow(3, 1/3)            // 3的立方根
Math.sin(0)                 // 三角函数: 还有Math.cos, Math.atan等
Math.log(10)                // 10的自然对数
Math.log(100)/Math.LN10     // 以10为底100的对数
Math.log(512)/Math.LN2      // 以2为底512的对数
Math.exp(3)                 // e的三次幂
```

参阅第三部分中关于Math对象的介绍，那里列出了JavaScript所支持的所有数学函数。

JavaScript中的算术运算在溢出（overflow）、下溢（underflow）或被零整除时不会报错。当数字运算结果超过了JavaScript所能表示的数字上限（溢出），结果为一个特殊的无穷大（infinity）值，在JavaScript中以Infinity表示。同样地，当负数的值超过了JavaScript所能表示的负数范围，结果为负无穷大，在JavaScript中以-Infinity表示。无穷大值的行为特性和我们所期望的是一致的：基于它们的加、减、乘和除运算结果还是无穷大值（当然还保留它们的正负号）。

下溢（underflow）是当运算结果无限接近于零并比JavaScript能表示的最小值还小的时候发生的一种情形。这种情况下，JavaScript将会返回0。当一个负数发生下溢时，JavaScript返回一个特殊的值“负零”。这个值（负零）几乎和正常的零完全一样，JavaScript程序员很少用到负零。

被零整除在JavaScript并不报错：它只是简单的返回无穷大（Infinity）或负无穷大（-Infinity）。但有一个例外，零除以零是没有意义的，这种整除运算结果也是一个非数字（not-a-number）值，用NaN表示。无穷大除以无穷大、给任意负数作开方运算或者算术运算符与不是数字或无法转换为数字的操作数一起使用时都将返回NaN。

JavaScript预定义了全局变量Infinity和NaN，用来表示正无穷大和非数字值。在ECMAScript 3中，这两个值是可读/写的，并可修改。ECMAScript 5修正了这个错误，将它们定义为只读的。在ECMAScript 3中Number对象定义的属性值也是只读的。这里有一些例子：

```
Infinity                        // 将一个可读/写的变量初始化为infinity
Number.POSITIVE_INFINITY        //同样的值，只读
1/0                             // 这也是同样的值
Number.MAX_VALUE + 1            //计算结果还是Infinity
Number.NEGATIVE_INFINITY        // 该表达式表示了负无穷大
-Infinity
-1/0
-Number.MAX_VALUE - 1
NaN                             // 将一个可读/写的变量初始化为NaN
Number.NaN                      //同样的值，但是只读
0/0                             // 计算结果是NaN
Number.MIN_VALUE/2              // 发生下溢：计算结果为0
-Number.MIN_VALUE/2             // 负零
-1/Infinity                     // 同样是负零
-0
```

js特有的吧

JavaScript中的非数字值有一点特殊：它和任何值都不相等，包括自身。也就是说，没办法通过x==NaN来判断变量x是否是NaN。相反，应当使用x!=x来判断，当且仅当x为NaN的时候，表达式的结果才为true。函数isNaN()的作用与此类似，如果参数是NaN或者是一个非数字值（比如字符串和对象），则返回true。JavaScript中有一个类似的函数isFinite()，在参数不是NaN、Infinity或-Infinity的时候返回true。

负零值同样有些特殊，它和正零值是相等的（甚至使用JavaScript的严格相等测试来判断）。这意味着这两个值几乎一模一样，除了作为除数之外：

```
var zero = 0;                   // 正常的零值
var negz = -0;                  // 负零值
zero === negz                   // => true: 正零值和负零值相等
1/zero === 1/negz               // => false: 正无穷大和负无穷大不等
```

3.1.4 二进制浮点数和四舍五入错误

实数有无数个，但JavaScript通过浮点数的形式只能表示其中有限的个数（确切地说是18 437 736 874 454 810 627个）。也就是说，当在JavaScript中使用实数的时候，常常只是真实值的一个近似表示。

JavaScript采用了IEEE-754浮点数表示法（几乎所有现代编程语言所采用），这是一种二进制表示法，可以精确地表示分数，比如1/2、1/8和1/1024。遗憾的是，我们常用的分数（特别是在金融计算方面）都是十进制分数1/10、1/100等。二进制浮点数表示法并不能精确表示类似0.1这样简单的数字。

JavaScript中的数字具有足够的精度，并可以极其近似于0.1。但事实是，数字不能精确表述的确带来了一些问题。看下这段代码：

```
var x = .3 - .2;                // 30美分减去20美分
var y = .2 - .1;                // 20美分减去10美分
x == y                          // => false:两值不相等!
```

```
x == .1                    // => false: .3-.2 不等于 .1
y == .1                    // => true: .2-.1 等于 .1
```

测试C/C++, Py， Delphi

由于舍入误差，0.3和0.2之间的近似差值实际上并不等于0.2和0.1之间的近似差值[译注2]。这个问题并不只在JavaScript中才会出现，理解这一点非常重要：在任何使用二进制浮点数的编程语言中都会有这个问题。同样需要注意的是，上述代码中x和y的值非常接近彼此和最终的正确值。这种计算结果可以胜任大多数的计算任务：这个问题也只有在比较两个值是否相等的时候才会出现。

JavaScript的未来版本或许会支持十进制数字类型以避免这些舍入问题。在这之前你可能更愿意使用大整数进行重要的金融计算，例如，要使用整数“分”而不要使用小数“元”进行基于货币单位的运算。

3.1.5 日期和时间

JavaScript语言核心包括Date()构造函数，用来创建表示日期和时间的对象。这些日期对象的方法为日期计算提供了简单的API。日期对象不像数字那样是基本数据类型。本节给出了使用日期对象的一个简单教程。在第三部分可以查阅更多细节：

```
var then = new Date(2011, 0, 1); // 2011年1月1日
var later = new Date(2011, 0, 1, 17, 10, 30);// 同一天，当地时间5:10:30pm,
var now = new Date(); // 当前日期和时间
var elapsed = now - then; // 日期减法：计算时间间隔的毫秒数
later.getFullYear() // => 2011
later.getMonth() // => 0: 从0开始计数的月份
later.getDate() // => 1: 从1开始计数的天数
later.getDay() // => 5: 得到星期几， 0代表星期日，5代表星期一
later.getHours() // => 当地时间17: 5pm
later.getUTCHours() // 使用UTC表示小时的时间，基于时区
```

3.2 文本

两个字节，Unicode？

字符串（string）是一组由16位值组成的不可变的有序序列，每个字符通常来自于Unicode字符集。JavaScript通过字符串类型来表示文本。字符串的长度（length）是其所含16位值的个数。JavaScript字符串（和其数组）的索引从零开始：第一个字符的位置是0，第二个字符的位置是1，以此类推。空字符串（empty string）长度为0，JavaScript中并没有表示单个字符的“字符型”。要表示一个16位值，只需将其赋值给字符串变量即可，这个字符串长度为1。

译注2： 在JavaScript的真实运行环境中，0.3－0.2=0.099 999 999 999 999 98。

字符集，内码和JavaScript字符串

JavaScript采用UTF-16编码的Unicode字符集，JavaScript字符串是由一组无符号的16位值组成的序列。最常用的Unicode字符（这些字符属于“基本多语种平面”译注3）都是通过16位的内码表示，并代表字符串中的单个字符，那些不能表示为16位的Unicode字符则遵循UTF-16编码规则——用两个16位值组成的一个序列（亦称做“代理项对”）表示。这意味着一个长度为2的JavaScript字符串(两个16位值)有可能表示一个Unicode字符：

兼容ANSI的话，1个字节不就可以吗?

```
var p = "π";          //π由16位内码表示0x03c0
var e = "e";          // e由17位内码表示0x1d452
p.length              // => 1: p包含一个16位值
e.length              // => 2: e通过UTF-16编码后包含两个16位值："\ud835\udc52"
```

JavaScript定义的各式字符串操作方法均作用于16位值，而非字符，且不会对代理项对做单独处理，同样JavaScript不会对字符串做标准化的加工，甚至不能保证字符串是合法的UTF-16格式。

3.2.1 字符串直接量

在JavaScript程序中的字符串直接量，是由单引号或双引号括起来的字符序列。由单引号定界的字符串中可以包含双引号，由双引号定界的字符串中也可以包含单引号。这里有几个字符串直接量的例子：

```
"" //空字符串：它包含0个字符
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"This string\nhas two lines"
"π is the ratio of a circle's circumference to its diameter"
```

在ECMAScript 3中，字符串直接量必须写在一行中，而在ECMAScript 5中，字符串直接量可以拆分成数行，每行必须以反斜线（\）结束，反斜线和行结束符都不算是字符串直接量的内容。如果希望在字符串直接量中另起一行，可以使用转义字符\n（后续会有介绍）：

续行用\

```
"two\nlines" // 这里定义了一个显示为两行的字符串
"one\ // 用三行代码定义了显示为单行的字符串，只在ECMAScript 5中可用
long\
```

译注3：“基本多语种平面”(Basic Multilingual Plane，BMP)，也称“零断面”(Plan 0)，是Unicode中的一个编码区段。编码介于U+0000~U+FFFF之间。

```
line"
```

需要注意的是，当使用单引号来定界字符串时，需要格外小心英文中的缩写和所有格写法，比如can't和O'Reilly's。因为撇号和单引号是同一个字符，所以必须使用反斜线(\)来转义（转义符将在下一章讲解）所有的撇号。

在客户端JavaScript程序设计中，JavaScript代码会夹杂HTML代码的字符串，HTML代码也会夹杂JavaScript代码。和JavaScript一样，HTML也使用单引号或者双引号来定界字符串，因此，当JavaScript代码和HTML代码混杂在一起的时候，最好在JavaScript和HTML代码中各自使用独立的引号风格。例如，在JavaScript表达式中使用单引号表示字符串“Thank you”，而在HTML事件处理程序属性中则使用双引号表示字符串：

```
<button onclick="alert('Thank you')">Click Me</button>
```

3.2.2 转义字符

在JavaScript字符串中，反斜线(\)有着特殊的用途，反斜线符号后加一个字符，就不再表示它们的字面含义了，比如，\n就是一个转义字符(escape sequence)[译注4]，它表示的是一个换行符。

另一个例子是上节中提到的转义字符\'，表示单引号（或撇号）。当需要在一个单引号定界的字符串内使用撇号的时候，它就显得非常有用。现在你就会明白我们为什么把它们叫做转义字符了，因为反斜线可以使我们避免使用常规方式解释单引号，当单引号不是用来标记字符串结尾时，它只是一个撇号：

```
'You\'re right, it can\'t be a quote'
```

表格3-1列出了JavaScript中的转义字符以及它们所代表的含义。其中有两个是通用的，通过十六进制数表示Latin-1或Unicode中的任意字码。例如，\xA9表示版权符号，版权符号的Latin-1编码是十六进制数A9。同样，\u表示由4个十六进制数指定的任意Unicode字符，比如，\u03c0表示字符π。

表3-1：JavaScript转义字符

转义字符	含义
\0	NUL字符(\u0000)
\b	退格符(\u0008)
\t	水平制表符(\u0009)

译注4：escape sequence译为“转义序列”，有时也译成“转义字符”和“逃逸符”，本节中统一译为“转义字符”。

表3-1：JavaScript转义字符（续）

转义字符	含义
\n	换行符(\u000A)
\v	垂直制表符(\u000B)
\f	换页符(\u000C)
\r	回车符(\u000D)
\"	双引号(\u0022)
\'	撇号或单引号(\u0027)
\\	反斜线(\u005C)
\x*XX*	由两位十六进制数*XX*指定的Latin-1字符
\u*XXXX*	由4位十六进制数*XXXX*指定的Unicode字符

如果“\”字符位于没有在表3-1中列出的字符前，则忽略“\”（当然，JavaScript语言将来的版本可能定义新的转义符）。比如，“\#”和“#”等价。最后，上文提到过，在ECMAScript 5中，允许在一个多行字符串直接量里的每行结束处使用反斜线。

3.2.3 字符串的使用

JavaScript的内置功能之一就是字符串连接。如果将加号(+)运算符用于数字，表示两数相加。但将它作用于字符串，则表示字符串连接，将第二个字符串拼接在第一个之后，例如：

```
msg = "Hello, " + "world"; // 生成字符串 "Hello, world"
greeting = "Welcome to my blog," + " " + name;
```

要确定一个字符串的长度——其所包含的16位值的个数——可以使用字符串的length属性。比如，要得到字符串s的长度：

```
s.length
```

除了length属性，字符串还提供许多可以调用的方法（可以在第三部分查到详细信息）：

```
var s = "hello, world"        // 定义一个字符串
s.charAt(0)                   // => "h": 第一个字符
s.charAt(s.length-1)          // => "d": 最后一个字符
s.substring(1,4)              // => "ell":第2～4个字符
s.slice(1,4)                  // => "ell": 同上
s.slice(-3)                   // => "rld": 最后三个字符
s.indexOf("l")                // => 2: 字符l首次出现的位置
s.lastIndexOf("l")            // => 10:字符l最后一次出现的位置
s.indexOf("l", 3)             // => 3:在位置3及之后首次出现字符l的位置
s.split(", ")                 // => ["hello", "world"] 分割成子串
```

```
s.replace("h", "H")          // => "Hello, world": 全文字符替换
s.toUpperCase()              // => "HELLO, WORLD"
```

记住，在JavaScript中字符串是固定不变的，类似replace()和toUpperCase()的方法都返回新字符串，原字符串本身并没有发生改变。

在ECMAScript 5中，字符串可以当做只读数组，除了使用charAt()方法，也可以使用方括号来访问字符串中的单个字符（16位值）：

```
s = "hello, world";
s[0]                 // => "h"
s[s.length-1]        // => "d"
```

基于Mozilla的Web浏览器（比如Firefox）很久之前就支持这种方式的字符串索引，多数现代浏览器(IE除外)也紧跟Mozilla的脚步，在ECMAScript 5成型之前就支持了这一特性。

3.2.4 模式匹配

JavaScript定义了RegExp()构造函数，用来创建表示文本匹配模式的对象。这些模式称为“正则表达式”（regular expression），JavaScript采用Perl中的正则表达式语法。String和RegExp对象均定义了利用正则表达式进行模式匹配和查找与替换的函数。

RegExp并不是JavaScript的基本类型。和Date一样，它只是一种具有实用API的特殊对象。正则表达式的语法很复杂，API也很丰富。在第10章有详尽的文档介绍。RgeExp是一种强大和常用的文本处理工具，本节只是一个概述。

尽管RegExp并不是语言中的基本数据类型，但是它们依然具有直接量写法，可以直接在JavaScript程序中使用。在两条斜线之间的文本构成了一个正则表达式直接量。第二条斜线之后也可以跟随一个或多个字母，用来修饰匹配模式的含义，例如：

```
/^HTML/ //匹配以HTML开始的字符串
/[1-9][0-9]*/ // 匹配一个非零数字，后面是任意个数字
/\bjavascript\b/i // 匹配单词"javascript"，忽略大小写
```

RegExp对象定义了很多有用的方法，字符串同样具有可以接收RegExp参数的方法，例如：

```
var text = "testing: 1, 2, 3"; // 文本示例
var pattern = /\d+/g // 匹配所有包含一个或多个数字的实例
pattern.test(text) // => true: 匹配成功
text.search(pattern) // => 9: 首次匹配成功的位置
text.match(pattern) // => ["1", "2", "3"]: 所有匹配组成的数组
text.replace(pattern, "#"); // => "testing: #, #, #"
text.split(/\D+/); // => ["","1","2","3"]: 用非数字字符截取字符串
```

3.3 布尔值

布尔值指代真或假、开或关、是或否。这个类型只有两个值，保留字true和false。

JavaScript程序中的比较语句的结果通常都是布尔值，例如：

```
a==4
```

这段代码用来检测变量a的值是否等于4。如果等于，比较结果的布尔值就是true；如果不等，比较结果则为false。

布尔值通常用于JavaScript中的控制结构中。例如，JavaScript中的if/else语句，如果布尔值为true执行第一段逻辑，如果为false执行另一段逻辑。通常将一个创建布尔值的比较直接与使用这个比较的语句结合在一起，结果如下所示：

```
if (a == 4)
b = b + 1;
else
a = a + 1;
```

这段代码检测变量a是否等于4。如果等于，则b加1；否则，a加1。我们同样会在3.8节讨论到，任意JavaScript的值都可以转换为布尔值。下面这些值会被转换成false：

```
undefined
null
0
-0
NaN
"" // 空字符串
```

所有其他值，包括所有对象（数组）都会转换成true。false和上面6个可以转换成false的值有时称做“假值”(falsy value)，其他值称做“真值”(truthy value)。JavaScript期望使用一个布尔值的时候，假值会被当成false，真值会被当成true。

来看一个例子，假设变量o是一个对象或是null，可以通过一条if语句来显式地检测o是否是非null值：

```
if (o !== null) ...
```

不等操作符“!==”将o和null比较，并得出结果为true或false。可以先忽略这里的比较语句，null是一个假值，对象是一个真值：

```
if (o) ...
```

null / undefined

对于第一种情况，只有当o不是null时才会执行if后的代码，第二种情况的限制没那么严

格：只有o不是false或任何假值（比如null或undefined）时它才会执行这个if。到底选用哪条语句取决于期望赋给o的值是什么。如果需要将null与0或""区分开来，则需要使用一个显式的比较。

布尔值包含toString()方法，因此可以使用这个方法将字符串转换为“true”或“false”，但它并不包含其他有用的方法。除了这个不重要的API，还有三个重要的布尔运算符。

“&&”运算符执行了逻辑与（AND）操作。当且仅当两个操作数都是真值时它才返回true；否则返回false。“||”运算符是布尔或（OR）操作，如果两个操作数其中之一为真值它就返回true，如果两个操作数都是假值则返回false。最后，一元操作符“!”执行了布尔非（NOT）操作：如果操作数是真值则返回false；如果是假值，则返回true。比如：

```
if ((x == 0 && y == 0) || !(z == 0)) {
    // x 和 y 都是零或 z 是非零
}
```

关于操作数的完整的细节可以参照4.10节。

3.4 null和undefined

null是JavaScript语言的关键字，它表示一个特殊值，常用来描述“空值”。对null执行typeof预算，结果返回字符串“object”，也就是说，可以将null认为是一个特殊的对象值，含义是“非对象”。但实际上，通常认为null是它自有类型的唯一一个成员，它可以表示数字、字符串和对象是“无值”的。大多数编程语言和JavaScript一样含有null：你可能对null或nil很眼熟。

JavaScript还有第二个值来表示值的空缺。用未定义的值表示更深层次的“空值”。它是变量的一种取值，表明变量没有初始化，如果要查询对象属性或数组元素的值时返回undefined则说明这个属性或元素不存在。如果函数没有返回任何值，则返回undefined。引用没有提供实参的函数形参的值也只会得到undefined。undefined是预定义的全局变量（它和null不一样，它不是关键字），它的值就是“未定义”。在ECMAScript 3中，undefined是可读/写的变量，可以给它赋任意值。这个错误在ECMAScript 5中做了修正，undefined在该版本中是只读的。如果使用typeof运算符得到undefined的类型，则返回“undefined”，表明这个值是这个类型的唯一成员。

尽管null和undefined是不同的，但它们都表示“值的空缺”，两者往往可以互换。判断相等运算符“==”认为两者是相等的（要使用严格相等运算符“===”来区分它们）。在希望值是布尔类型的地方它们的值都是假值，和false类似。null和undefined都不包含

任何属性和方法。实际上，使用“.”和“[]”来存取这两个值的成员或方法都会产生一个类型错误。

你或许认为undefined是表示系统级的、出乎意料的或类似错误的值的空缺，而null是表示程序级的、正常的或在意料之中的值的空缺。如果你想将它们赋值给变量或者属性，或将它们作为参数传入函数，最佳选择是使用null。

3.5 全局对象

前几节讨论了JavaScript的原始类型和原始值。对象类型——对象、数组和函数——在本书中均会有独立章节来讲述。但有一类非常重要的对象，我们不得不现在就把它们讲清楚——全局对象。全局对象（global object）在JavaScript中有着重要的用途：全局对象的属性是全局定义的符号，JavaScript程序可以直接使用。当JavaScript解释器启动时（或者任何Web浏览器加载新页面的时候），它将创建一个新的全局对象，并给它一组定义的初始属性：

- 全局属性，比如undefined、Infinity和NaN
- 全局函数，比如isNaN()、parseInt()（见3.8.2节）和eval()（见4.12节）
- 构造函数，比如Date()、RegExp()、String()、Object()和Array()（见3.8.2节）
- 全局对象，比如Math和JSON(见6.9节)

全局对象的初始属性并不是保留字，但它们应该当做保留字来对待。2.4.1节列出了所有这些属性。本章对一部分全局属性也有描述。其他属性在其他章节也会讲述。可以在第三部分中通过名称查找到，或者通过别名“Global”来找到这些全局对象。对于客户端JavaScript来讲，Window对象定义了一些额外的全局属性，可以在第四部分中查看它们。

在代码的最顶级——不在任何函数内的JavaScript代码——可以使用JavaScript关键字this来引用全局对象：

```
var global = this; // 定义一个引用全局对象的全局变量
```

在客户端JavaScript中，在其表示的浏览器窗口中的所有JavaScript代码中，Window对象充当了全局对象。这个全局Window对象有一个属性window引用其自身，它可以代替this来引用全局对象。Window对象定义了核心全局属性，但它也针对Web浏览器和客户端JavaScript定义了一少部分其他全局属性。

当初次创建的时候，全局对象定义了JavaScript中所有的预定义全局值。这个特殊对象同

样包含了为程序定义的全局值。如果代码声明了一个全局变量，这个全局变量就是全局对象的一个属性，3.10.2节有关于此的详尽解释。

3.6 包装对象

JavaScript对象是一种复合值：它是属性或已命名值的集合。通过“.”符号来引用属性值。当属性值是一个函数的时候，称其为方法。通过o.m()来调用对象o中的方法。

我们看到字符串也同样具有属性和方法：

```
var s = "hello world!";                                    // 一个字符串
var word = s.substring(s.indexOf(" ")+1, s.length);        //使用字符串的属性
```

字符串既然不是对象，为什么它会有属性呢？只要引用了字符串s的属性，JavaScript就会将字符串值通过调用new String(s)的方式转换成对象，这个对象继承了字符串的方法（见6.2.2节），并被用来处理属性的引用。一旦属性引用结束，这个新创建的对象就会销毁（其实在实现上并不一定创建或销毁这个临时对象，然而整个过程看起来是这样）。

同字符串一样，数字和布尔值也具有各自的方法：通过Number()和Boolean()构造函数创建一个临时对象，这些方法的调用均是来自于这个临时对象。null和undefined没有包装对象：访问它们的属性会造成一个类型错误。

看如下代码，思考它们的执行结果：

```
var s = "test";        //创建一个字符串
s.len = 4;             // 给它设置一个属性
var t = s.len;         // 查询这个属性
```

没有len属性

当运行这段代码时，t的值是undefined。第二行代码创建一个临时字符串对象，并给其len属性赋值为4，随即销毁这个对象。第三行通过原始的（没有被修改过）字符串值创建一个新字符串对象，尝试读取其len属性，这个属性自然不存在，表达式求值结果为undefined。这段代码说明了在读取字符串、数字和布尔值的属性值（或方法）的时候，表现的像对象一样。但如果你试图给其属性赋值，则会忽略这个操作：修改只是发生在临时对象身上，而这个临时对象并未继续保留下来。

存取字符串、数字或布尔值的属性时创建的临时对象称做包装对象，它只是偶尔用来区分字符串值和字符串对象、数字和数值对象以及布尔值和布尔对象。通常，包装对象只是被看做是一种实现细节，而不用特别关注。由于字符串、数字和布尔值的属性都是只读的，并且不能给它们定义新属性，因此你需要明白它们是有别于对象的。

需要注意的是，可通过String()，Number()或Boolean()构造函数来显式创建包装对象：

```
var s = "test", n = 1, b = true;      // 一个字符串、数字和布尔值
var S = new String(s);                // 一个字符串对象
var N = new Number(n);                // 一个数值对象
var B = new Boolean(b);               // 一个布尔对象
```

JavaScript会在必要时将包装对象转换成原始值，因此上段代码中的对象S、N和B常常——但不总是——表现的和值s、n和b一样。“==”等于运算符将原始值和其包装对象视为相等，但“===”全等运算符将它们视为不等。通过typeof运算符可以看到原始值和其包装对象的不同。

3.7 不可变的原始值和可变的对象引用

JavaScript中的原始值（undefined、null、布尔值、数字和字符串）与对象（包括数组和函数）有着根本区别。原始值是不可更改的：任何方法都无法更改（或“突变”）一个原始值。对数字和布尔值来说显然如此——改变数字的值本身就说不通，而对字符串来说就不那么明显了，因为字符串看起来像由字符组成的数组，我们期望可以通过指定索引来修改字符串中的字符。实际上，JavaScript是禁止这样做的。字符串中所有的方法看上去返回了一个修改后的字符串，实际上返回的是一个新的字符串值。例如：

```
var s = "hello";      // 定义一个由小写字母组成的文本
s.toUpperCase();      // 返回"HELLO"，但并没有改变s的值
s                     // => "hello": 原始字符串的值并未改变
```

原始值的比较是值的比较：只有在它们的值相等时它们才相等。这对数字、布尔值、null和undefined来说听起来有点儿难懂，并没有其他办法来比较它们。同样，对于字符串来说则并不明显：如果比较两个单独的字符串，当且仅当它们的长度相等且每个索引的字符都相等时，JavaScript才认为它们相等。

对象和原始值不同，首先，它们是可变的——它们的值是可修改的：

```
var o = { x:1 };      // 定义一个对象
o.x = 2;              // 通过修改对象属性值来更改对象
o.y = 3;              // 再次更改这个对象，给它增加一个新属性

var a = [1,2,3]       // 数组也是可修改的
a[0] = 0;             // 更改数组的一个元素
a[3] = 4;             // 给数组增加一个新元素
```

对象的比较并非值的比较：即使两个对象包含同样的属性及相同的值，它们也是不相等的。各个索引元素完全相等的两个数组也不相等。

```
var o = {x:1}, p = {x:1};     // 具有相同属性的两个对象
o === p                       // => false: 两个单独的对象永不相等
var a = [], b = [];           // 两个单独的空数组
a === b                       // => false: 两个单独的数组永不相等
```

我们通常将对象称为引用类型（reference type），以此来和JavaScript的基本类型区分开来。依照术语的叫法，对象值都是引用（reference），对象的比较均是引用的比较：当且仅当它们引用同一个基对象时，它们才相等。

```
var a = [];  // 定义一个引用空数组的变量a
var b = a;   // 变量b引用同一个数组
b[0] = 1;    // 通过变量b来修改引用的数组
a[0]         // => 1: 变量a也会修改
a === b      // => true:a和b引用同一个数组，因此它们相等
```

就像你刚看到的如上代码，将对象（或数组）赋值给一个变量，仅仅是赋值的引用值：对象本身并没有复制一次。如果你想得到一个对象或数组的副本，则必须显式复制对象的每个属性或数组的每个元素。下面这个例子则是通过循环来完成数组复制（见5.5.3节）：

```
var a = ['a','b','c'];                  // 待复制的数组
var b = [];                             // 复制到的目标空数组
for(var i = 0; i < a.length; i++) {     // 遍历a[]中的每个元素
   b[i] = a[i];                         // 将元素值复制到b中
}
```

同样的，如果我们想比较两个单独的对象或者数组，则必须比较它们的属性或元素。下面这段代码定义了一个比较两个数组的函数：

```
function equalArrays(a,b) {
    if (a.length != b.length) return false;     // 两个长度不同的数组不相等
    for(var i = 0; i < a.length; i++)           // 循环遍历所有元素
        if (a[i] !== b[i]) return false;        // 如果有任意元素不等，则数组不相等
    return true;                                // 否则它们相等
}
```

3.8 类型转换

JavaScript中的取值类型非常灵活，我们已经从布尔值看到了这一点：当JavaScript期望使用一个布尔值的时候，你可以提供任意类型值，JavaScript将根据需要自行转换类型。一些值（真值）转换为true，其他值（假值）转换为false。这在其他类型中同样适用：如果JavaScript期望使用一个字符串，它把给定的值将转换为字符串。如果JavaScript期望使用一个数字，它把给定的值将转换为数字（如果转换结果无意义的话将返回NaN），一些例子如下：

```
10 + " objects"       // => "10 objects". 数字10转换成字符串
"7" * "4"             // => 28: 两个字符串均转换为数字
var n = 1 - "x";      // => NaN: 字符串"x"无法转换为数字
n + " objects"        // => "NaN objects": NaN转换为字符串"NaN"
```

表3-2简要说明了在JavaScript中如何进行类型转换。表3-2中的粗体部分突出显示了那些让你倍感意外的类型转换。空单元格表示不必要也没有执行转换。

表3-2：JavaScript类型转换

值	转换为：			
	字符串	数字	布尔值	对象
undefined	"undefined"	NaN	false	throws TypeError
null	"null"	0	false	throws TypeError
true	"true"	1		new Boolean(true)
false	"false"	0		new Boolean(false)
""(空字符串)		0	false	new String("")
"1.2"(非空,数字)		1.2	true	new String("1.2")
"one"(非空,非数字)		NaN	true	new String("one")
0	"0"		false	new Number(0)
-0	"0"		false	new Number(-0)
NaN	"NaN"		false	new Number(NaN)
Infinity	"Infinity"		true	new Number(Infinity)
-Infinity	"-Infinity"		true	new Number(-Infinity)
1(无穷大,非零)	"1"		true	new Number(1)
{}(任意对象)	参考3.8.3节	参考3.8.3节	true	
[]任意数组)	""	0	true	
[9](1个数字元素)	"9"	9	true	
['a'](其他数组)	使用join()方法	NaN	true	
function(){}(任意函数)	参考3.8.3节	NaN	true	

表3-2中提到的原始值到原始值的转换相对简单，我们已经在3.3节讨论过转换为布尔值的情况了。所有原始值转换为字符串的情形也已经明确定义。转换为数字的情形比较微妙。那些以数字表示的字符串可以直接转换为数字，也允许在开始和结尾处带有空格。但在开始和结尾处的任意非空格字符都不会被当成数字直接量的一部分，进而造成字符串转换为数字的结果为NaN。有一些数字转换看起来让人奇怪：true转换为1，false、空字符串""转换为0。

原始值到对象的转换也非常简单，原始值通过调用String()、Number()或Boolean()构造函数，转换为它们各自的包装对象（见3.6节）。

null和undefined属于例外，当将它们用在期望是一个对象的地方都会造成一个类型错误（TypeError）异常，而不会执行正常的转换。

对象到原始值的转换多少有些复杂，3.8.3 节将以此为专题专门讲述。

3.8.1 转换和相等性

由于JavaScript可以做灵活的类型转换，因此其“==”相等运算符也随相等的含义灵活多变。例如，如下这些比较结果均是true：

```
null == undefined      // 这两值被认为相等
"0" == 0               // 在比较之前字符串转换成数字
0 == false             // 在比较之前布尔值转换成数字
"0" == false           // 在比较之前字符串和布尔值都转换成数字
```

4.9.1节详细讲解了“==”等于运算符在判断两个值是否相等时做了哪些类型转换，并同样介绍了“===”恒等运算符在判断相等时并未做任何类型转换。

需要特别注意的是，一个值转换为另一个值并不意味着两个值相等。比如，如果在期望使用布尔值的地方使用了undefined，它将会转换为false，但这并不表明undefined == false。JavaScript运算符和语句期望使用多样化的数据类型，并可以相互转换。if语句将undefined转换为false，但“==”运算符从不试图将其操作数转换为布尔值。

3.8.2 显式类型转换

尽管JavaScript可以自动做许多类型转换，但有时仍需要做显式转换，或者为了使代码变得清晰易读而做显式转换。

做显式类型转换最简单的方法就是使用Boolean()、Number()、String()或Object()函数。我们在3.6节已经介绍过了。当不通过new运算符调用这些函数时，它们会作为类型转换函数并按照表3-2所描述的规则做类型转换：

```
Number("3")            // => 3
String(false)          // => "false" 或使用 false.toString()
Boolean([])            // => true
Object(3)              // => new Number(3)
```

需要注意的是，除了null或undefined之外的任何值都具有toString()方法，这个方法的执行结果通常和String()方法的返回结果一致。同样需要注意的是，如果试图把null或undefined转换为对象，则会像表3-2所描述的那样抛出一个类型错误（TypeError）。Object()函数在这种情况下不会抛出异常：它仅简单地返回一个新创建的空对象。

JavaScript中的某些运算符会做隐式的类型转换，有时用于类型转换。如果“+”运算符的一个操作数是字符串，它将会把另外一个操作数转换为字符串。一元“+”运算符将其操作数转换为数字。同样，一元“！”运算符将其操作数转换为布尔值并取反。在代码中会经常见到这种类型转换的惯用法：

```
x + ""  // 等价于String(x)
```

```
+x      // 等价于 Number(x).也可以写成 x-0
!!x     // 等价于 Boolean(x). 注意是双叹号
```

在计算机程序中数字的解析和格式化是非常普通的工作，JavaScript中提供了专门的函数和方法用来做更加精确的数字到字符串（number-to-string）和字符串到数字（string-to-number）的转换。

Number类定义的toString()方法可以接收表示转换基数（radix）[译注5]的可选参数，如果不指定此参数，转换规则将是基于十进制。同样，亦可以将数字转换为其他进制数（范围在2～36之间），例如：

```
var n = 17;
binary_string = n.toString(2);        // 转换为 "10001"
octal_string = "0" + n.toString(8);   // 转换为 "021"
hex_string = "0x" + n.toString(16);   // 转换为 "0x11"
```

当处理财务或科学数据的时候，在做数字到字符串的转换过程中，你期望自己控制输出中小数点位置和有效数字位数，或者决定是否需要指数记数法。Number类为这种数字到字符串的类型转换场景定义了三个方法。toFixed()根据小数点后的指定位数将数字转换为字符串，它从不使用指数记数法。toExponential()使用指数记数法将数字转换为指数形式的字符串，其中小数点前只有一位，小数点后的位数则由参数指定（也就是说有效数字位数比指定的位数要多一位）[译注6]，toPrecision()根据指定的有效数字位数将数字转换成字符串。如果有效数字的位数少于数字整数部分的位数，则转换成指数形式。我们注意到，所有三个方法都会适当地进行四舍五入或填充0。看一下下面几个例子：

```
var n = 123456.789;
n.toFixed(0); // "123457"
n.toFixed(2); // "123456.79"
n.toFixed(5); // "123456.78900"
n.toExponential(1); // "1.2e+5"
n.toExponential(3); // "1.235e+5"
n.toPrecision(4); // "1.235e+5"
n.toPrecision(7); // "123456.8"
n.toPrecision(10); // "123456.7890"
```

如果通过Number()转换函数传入一个字符串，它会试图将其转换为一个整数或浮点数直接量，这个方法只能基于十进制数进行转换，并且不能出现非法的尾随字符。parseInt()函数和parseFloat()函数（它们是全局函数，不从属于任何类的方法）更加灵活。parseInt()只解析整数，而parseFloat()则可以解析整数和浮点数。如果字

译注5：这里的转换基数是指二进制、八进制、十六进制等。

译注6：如果指定的参数为3，有效数字位数为4位。

符串前缀是“0x”或者“0X”，parseInt()将其解释为十六进制数[注2]，parseInt()和parseFloat()都会跳过任意数量的前导空格，尽可能解析更多数值字符，并忽略后面的内容。如果第一个非空格字符是非法的数字直接量，将最终返回NaN：

```
parseInt("3 blind mice")      // => 3
parseFloat(" 3.14 meters")    // => 3.14
parseInt("-12.34")            // => -12
parseInt("0xFF")              // => 255
parseInt("0xff")              // => 255
parseInt("-0XFF")             // => -255
parseFloat(".1")              // => 0.1
parseInt("0.1")               // => 0
parseInt(".1")                // => NaN: 整数不能以"."开始
parseFloat("$72.47");         // => NaN: 数字不能以"$"开始
```

parseInt()可以接收第二个可选参数，这个参数指定数字转换的基数，合法的取值范围是2～36，例如：

```
parseInt("11", 2);    // => 3 (1*2 + 1)
parseInt("ff", 16);   // => 255 (15*16 + 15)
parseInt("zz", 36);   // => 1295 (35*36 + 35)
parseInt("077", 8);   // => 63 (7*8 + 7)
parseInt("077", 10);  // => 77 (7*10 + 7)
```

3.8.3 对象转换为原始值

对象到布尔值的转换非常简单：所有的对象（包括数组和函数）都转换为true。对于包装对象亦是如此：new Boolean(false)是一个对象而不是原始值，它将转换为true。

对象到字符串（object-to-string）和对象到数字（object-to-number）的转换是通过调用待转换对象的一个方法来完成的。一个麻烦的事实是，JavaScript对象有两个不同的方法来执行转换，并且接下来要讨论的一些特殊场景更加复杂。值得注意的是，这里提到的字符串和数字的转换规则只适用于本地对象（native object）。宿主对象（例如，由Web浏览器定义的对象）根据各自的算法可以转换成字符串和数字。

所有的对象继承了两个转换方法。第一个是toString()，它的作用是返回一个反映这个对象的字符串。默认的toString()方法并不会返回一个有趣的值（在例6-4中我们会发现它非常有用）：

注2：在ECMAScript 3中，parseInt()可以对前缀为“0”（不能是“0x”或“0X”）的数字做八进制转换。由于其细节没有详细说明，你并无法直接使用parseInt()来对前缀为0的值进行解析，除非你明确指出所使用的转换基数！在ECMAScript 5中，parseInt()只有在明确传入第二个参数8时才会解析八进制数。

```
({x:1, y:2}).toString() // => "[object Object]"
```

很多类定义了更多特定版本的toString()方法。例如，数组类（Array class）的toString()方法将每个数组元素转换为一个字符串，并在元素之间添加逗号后合并成结果字符串。函数类（Function class）的toString()方法返回这个函数的实现定义的表示方式。实际上，这里的实现方式是通常是将用户定义的函数转换为JavaScript源代码字符串。日期类（Date class）定义的toString()方法返回了一个可读的（可被JavaScript解析的[译注7]）日期和时间字符串。RegExp类（RegExp class）定义的toString()方法将RegExp对象转换为表示正则表达式直接量的字符串：

```
[1,2,3].toString()                    // => "1,2,3"
(function(x) { f(x); }).toString()    // => "function(x) {\n f(x);\n}"
/\d+/g.toString()                     // => "/\\d+/g"
new Date(2010,0,1).toString()         // => "Fri Jan 01 2010 00:00:00 GMT-0800 (PST)"
```

另一个转换对象的函数是valueOf()。这个方法的任务并未详细定义：如果存在任意原始值，它就默认将对象转换为表示它的原始值。对象是复合值，而且大多数对象无法真正表示为一个原始值，因此默认的valueOf()方法简单地返回对象本身，而不是返回一个原始值。数组、函数和正则表达式简单地继承了这个默认方法，调用这些类型的实例的valueOf()方法只是简单返回对象本身。日期类定义的valueOf()方法会返回它的一个内部表示：1970年1月1日以来的毫秒数。

```
var d = new Date(2010, 0, 1); // 2010年1月1日(太平洋时间)
d.valueOf()                   // => 1262332800000
```

通过使用我们刚刚讲解过的toString()和valueOf()方法，就可以做到对象到字符串和对象到数字的转换了。但需要注意的是，在某些特殊的场景中，JavaScript执行了完全不同的对象到原始值的转换。这些特殊场景在本节的最后会讲到。

JavaScript中对象到字符串的转换经过了如下这些步骤：

- 如果对象具有toString()方法，则调用这个方法。如果它返回一个原始值，JavaScript将这个值转换为字符串(如果本身不是字符串的话)，并返回这个字符串结果。需要注意的是，原始值到字符串的转换在表3-2中已经有了详尽的说明。
- 如果对象没有toString()方法，或者这个方法并不返回一个原始值，那么JavaScript会调用valueOf()方法。如果存在这个方法，则JavaScript调用它。如果返回值是原始值，JavaScript将这个值转换为字符串（如果本身不是字符串的话），并返回这个字符串结果。

译注7： 这里的原文是JavaScript-parsable，意指可以通过JavaScript的方法过滤并再做封装。

- 否则，JavaScript无法从toString()或valueOf()获得一个原始值，因此这时它将抛出一个类型错误异常。

在对象到数字的转换过程中，JavaScript做了同样的事情，只是它会首先尝试使用valueOf()方法：

- 如果对象具有valueOf()方法，后者返回一个原始值，则JavaScript将这个原始值转换为数字（如果需要的话）并返回这个数字。
- 否则，如果对象具有toString()方法，后者返回一个原始值，则JavaScript将其转换并返回[译注8]。
- 否则，JavaScript抛出一个类型错误异常。

对象转换为数字的细节解释了为什么空数组会被转换为数字0以及为什么具有单个元素的数组同样会转换成一个数字。数组继承了默认的valueOf()方法，这个方法返回一个对象而不是一个原始值，因此，数组到数字的转换则调用toString()方法。空数组转换成为空字符串，空字符串转换成为数字0。含有一个元素的数组转换为字符串的结果和这个元素转换字符串的结果一样。如果数组只包含一个数字元素，这个数字转换为字符串，再转换回数字。

JavaScript中的“+”运算符可以进行数学加法和字符串连接操作。如果它的其中一个操作数是对象，则JavaScript将使用特殊的方法将对象转换为原始值，而不是使用其他算术运算符的方法执行对象到数字的转换，“==”相等运算符与此类似。如果将对象和一个原始值比较，则转换将会遵照对象到原始值的转换方式进行。

“+”和“==”应用的对象到原始值的转换包含日期对象的一种特殊情形。日期类是JavaScript语言核心中唯一的预先定义类型，它定义了有意义的向字符串和数字类型的转换。对于所有非日期的对象来说，对象到原始值的转换基本上是对象到数字的转换（首先调用valueOf()），日期对象则使用对象到字符串的转换模式，然而，这里的转换和上文讲述的并不完全一致：通过valueOf或toString()返回的原始值将被直接使用，而不会被强制转换为数字或字符串。

和“==”一样，“<”运算符以及其他关系运算符也会做对象到原始值的转换，但要除去日期对象的特殊情形：任何对象都会首先尝试调用valueOf()，然后调用toString()。不管得到的原始值是否直接使用，它都不会进一步被转换为数字或字符串。

“+”、“==”、“!=”和关系运算符是唯一执行这种特殊的字符串到原始值的转换方

译注8：对象的toString()方法返回一个字符串直接量（作者所说的原始值），JavaScript将这个字符串转换为数字类型，并返回这个数字。

式的运算符。其他运算符到特定类型的转换都很明确，而且对日期对象来讲也没有特殊情况。例如“－”（减号）运算符把它的两个操作数都转换为数字。下面的代码展示了日期对象和“+”、“－”、“==”以及“>”的运行结果：

```
var now = new Date();          // 创建一个日期对象
typeof (now + 1)               // => "string": "+"将日期转换为字符串
typeof (now - 1)               // => "number": "-"使用对象到数字的转换
now == now.toString()          // => true: 隐式的和显式的字符串转换
now > (now -1)                 // => true: ">"将日期转换为数字
```

3.9 变量声明

在JavaScript程序中，使用一个变量之前应当先声明。变量是使用关键字var来声明的，如下所示：

```
var i;
var sum;
```

也可以通过一个var关键字来声明多个变量：

```
var i,sum;
```

而且还可以将变量的初始赋值和变量声明合写在一起：

```
var message = "hello";
var i = 0, j = 0, k = 0;
```

如果未在var声明语句中给变量指定初始值，那么虽然声明了这个变量，但在给它存入一个值之前，它的初始值就是undefined。

我们注意到，在for和for/in循环（在第5章会讲到）中同样可以使用var语句，这样可以更简洁地声明在循环体语法内中使用的循环变量。例如：

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

如果你之前编写过诸如C或Java的静态语言[译注9]，你会注意到在JavaScript的变量声明中

译注9：编程语言分为动态（类型）语言和静态（类型）语言，动态类型语言是指在运行期间才去做数据类型检查的语言，也就是说，在用动态类型的语言编程时，永远也不用给任何变量指定数据类型，该语言会在第一次赋值给变量时，在内部将数据类型记录下来。Python、Ruby和JavaScript就是典型的动态类型语言。静态类型语言与动态类型语言刚好相反，它的数据类型是在编译其间检查的，也就是说在写程序时要声明所有变量的数据类型，C/C++是静态类型语言的典型代表，其他的静态类型语言还有C#、JAVA等。

并没有指定变量的数据类型。JavaScript变量可以是任意数据类型。例如，在JavaScript中首先将数字赋值给一个变量，随后再将字符串赋值给这个变量，这是完全合法的：

```
var i = 10;
i = "ten";
```

重复的声明和遗漏的声明

使用var语句重复声明变量是合法且无害的。如果重复声明带有初始化器，那么这就和一条简单的赋值语句没什么两样。

如果你试图读取一个没有声明的变量的值，JavaScript会报错。在ECMAScript 5严格模式（见5.7.3节）中，给一个没有声明的变量赋值也会报错。然而从历史上讲，在非严格模式下，如果给一个未声明的变量赋值，JavaScript实际上会给全局对象创建一个同名属性，并且它工作起来像（但并不完全一样，查看3.10.2节）一个正确声明的全局变量。这意味着你可以侥幸不声明全局变量。但这是一个不好的习惯并会造成很多bug，因此，你应当始终使用var来声明变量。

3.10 变量作用域

一个变量的作用域（scope）是程序源代码中定义这个变量的区域。全局变量拥有全局作用域，在JavaScript代码中的任何地方都是有定义的。然而在函数内声明的变量只在函数体内有定义。它们是局部变量，作用域是局部性的。函数参数也是局部变量，它们只在函数体内有定义。

在函数体内，局部变量的优先级高于同名的全局变量。如果在函数内声明的一个局部变量或者函数参数中带有的变量和全局变量重名，那么全局变量就被局部变量所遮盖。

```
var scope = "global";          // 声明一个全局变量
function checkscope() {
    var scope = "local";       // 声明一个同名的局部变量
    return scope;              // 返回局部变量的值，而不是全局变量的值
}
checkscope()                   // => "local"
```

尽管在全局作用域编写代码时可以不写var语句，但声明局部变量时则必须使用var语句。思考一下如果不这样做会怎样：

```
scope = "global";              // 声明一个全局变量，甚至不用var来声明
function checkscope2() {
    scope = "local";           // 糟糕！我们刚修改了全局变量
    myscope = "local";         // 这里显式地声明了一个新的全局变量
    return [scope, myscope];   // 返回两个值
}
```

```
checkscope2()                   // => ["local", "local"]: 产生了副作用
scope                           // => "local": 全局变量修改了
myscope                         // => "local":全局命名空间搞乱了
```

函数定义是可以嵌套的。由于每个函数都有它自己的作用域，因此会出现几个局部作用域嵌套的情况，例如：

```
var scope = "global scope";    // 全局变量
function checkscope() {
var scope = "local scope";     //局部变量
function nested() {
var scope = "nested scope";    // 嵌套作用域内的局部变量
return scope;                  // 返回当前作用域内的值
}
return nested();
}
checkscope()                   // => "嵌套作用域"
```

3.10.1 函数作用域和声明提前

在一些类似C语言的编程语言中，花括号内的每一段代码都具有各自的作用域，而且变量在声明它们的代码段之外是不可见的，我们称为块级作用域（block scope），而JavaScript中没有块级作用域。JavaScript取而代之地使用了函数作用域（function scope）：变量在声明它们的函数体以及这个函数体嵌套的任意函数体内都是有定义的。

在如下所示的代码中，在不同位置定义了变量i、j和k，它们都在同一个作用域内——这三个变量在函数体内均是有定义的。

```
function test(o) {
    var i = 0;                          // i 在整个函数体内均是有定义的
    if (typeof o == "object") {
        var j = 0;                      // j在函数体内是有定义的，不仅仅是在这个代码段内
        for(var k=0; k < 10; k++) {     // k在函数体内是有定义的，不仅仅是在循环内
            console.log(k);             // 输出数字0～9
        }
        console.log(k);                 // k已经定义了，输出10
    }
    console.log(j);                     // j已经定义了，但可能没有初始化
}
```

JavaScript的函数作用域是指在函数内声明的所有变量在函数体内始终是可见的。有意思的是，这意味着变量在声明之前甚至已经可用。JavaScript的这个特性被非正式地称为声明提前（hoisting），即JavaScript函数里声明的所有变量（但不涉及赋值）都被“提前”至函数体的顶部[译注10]，看一下如下代码：

译注10：“声明提前”这步操作是在JavaScript引擎的“预编译”时进行的，是在代码开始运行之前，更多细节请阅读相关ppt：*http://www.slideshare.net/lijing00333/javascript-engine*。

```
var scope = "global";
function f() {
    console.log(scope);  // 输出"undefined"，而不是"global"
    var scope = "local"; // 变量在这里赋初始值，但变量本身在函数体内任何地方均是有定义的
    console.log(scope);  // 输出"local"
}
```

你可能会误以为函数中的第一行会输出“global”，因为代码还没有执行到var语句声明局部变量的地方。其实不然，由于函数作用域的特性，局部变量在整个函数体始终是有定义的，也就是说，在函数体内局部变量遮盖了同名全局变量。尽管如此，只有在程序执行到var语句的时候，局部变量才会被真正赋值。因此，上述过程等价于：将函数内的变量声明“提前”至函数体顶部，同时变量初始化留在原来的位置：

```
function f() {
    var scope;                // 在函数顶部声明了局部变量
    console.log(scope);       // 变量存在，但其值是"undefined"
    scope = "local";          // 这里将其初始化并赋值
    console.log(scope);       // 这里它具有了我们所期望的值
}
```

在具有块级作用域的编程语言中，在狭小的作用域里让变量声明和使用变量的代码尽可能靠近彼此，通常来讲，这是一个非常不错的编程习惯。由于JavaScript没有块级作用域，因此一些程序员特意将变量声明放在函数体顶部，而不是将声明靠近放在使用变量之处。这种做法使得他们的源代码非常清晰地反映了真实的变量作用域。

3.10.2 作为属性的变量

当声明一个JavaScript全局变量时，实际上是定义了全局对象的一个属性（见3.5节）。当使用var声明一个变量时，创建的这个属性是不可配置的（见6.7节），也就是说这个变量无法通过delete运算符删除。可能你已经注意到了，如果你没有使用严格模式并给一个未声明的变量赋值的话，JavaScript会自动创建一个全局变量。以这种方式创建的变量是全局对象的正常的可配值属性，并可以删除它们：

```
var truevar = 1;      // 声明一个不可删除的全局变量
fakevar = 2;          // 创建全局对象的一个可删除的属性
this.fakevar2 = 3;    // 同上
delete truevar        // => false: 变量并没有被删除
delete fakevar        // => true: 变量被删除
delete this.fakevar2  // => true: 变量被删除
```

JavaScript全局变量是全局对象的属性，这是在ECMAScript规范中强制规定的。对于局部变量则没有如此规定，但我们可以想象得到，局部变量当做跟函数调用相关的某个对象的属性。ECMAScript 3规范称该对象为“调用对象”(call object)，ECMAScript 5规范称为“声明上下文对象”（declarative environment record）。JavaScript可以允许使用

this关键字来引用全局对象，却没有方法可以引用局部变量中存放的对象。这种存放局部变量的对象的特有性质，是一种对我们不可见的内部实现。然而，这些局部变量对象存在的观念是非常重要的，我们会在下一节展开讲述。

3.10.3 作用域链

JavaScript是基于词法作用域的语言：通过阅读包含变量定义在内的数行源码就能知道变量的作用域。全局变量在程序中始终都是有定义的。局部变量在声明它的函数体内以及其所嵌套的函数内始终是有定义的。

如果将一个局部变量看做是自定义实现的对象的属性的话，那么可以换个角度来解读变量作用域。每一段JavaScript代码（全局代码或函数）都有一个与之关联的作用域链（scope chain）。这个作用域链是一个对象列表或者链表，这组对象定义了这段代码“作用域中”的变量。当JavaScript需要查找变量x的值的时候（这个过程称做“变量解析”（variable resolution）），它会从链中的第一个对象开始查找，如果这个对象有一个名为x的属性，则会直接使用这个属性的值，如果第一个对象中不存在名为x的属性，JavaScript会继续查找链上的下一个对象。如果第二个对象依然没有名为x的属性，则会继续查找下一个对象，以此类推。如果作用域链上没有任何一个对象含有属性x，那么就认为这段代码的作用域链上不存在x，并最终抛出一个引用错误（ReferenceError）异常。

在JavaScript的最顶层代码中（也就是不包含在任何函数定义内的代码），作用域链由一个全局对象组成。在不包含嵌套的函数体内，作用域链上有两个对象，第一个是定义函数参数和局部变量的对象，第二个是全局对象。在一个嵌套的函数体内，作用域链上至少有三个对象。理解对象链的创建规则是非常重要的。当定义一个函数时，它实际上保存一个作用域链。当调用这个函数时，它创建一个新的对象来存储它的局部变量，并将这个对象添加至保存的那个作用域链上，同时创建一个新的更长的表示函数调用作用域的“链”。对于嵌套函数来讲，事情变得更加有趣，每次调用外部函数时，内部函数又会重新定义一遍。因为每次调用外部函数的时候，作用域链都是不同的。内部函数在每次定义的时候都有微妙的差别——在每次调用外部函数时，内部函数的代码都是相同的，而且关联这段代码的作用域链也不相同。

作用域链的概念对于理解with语句（见5.7.1节）是非常有帮助的，同样对理解闭包（见8.6节）的概念也至关重要。