JavaScript 基础教程 V1.01

目录

1 JavaScript 简介

JavaScript 是因特网上最流行的脚本语言，它存在于全世界所有 Web 浏览器中，能够增强用户与 Web 站点和 Web 应用程序之间的交互。

JavaScript 的历史

为了发挥 JavaScript 的全部潜力，了解它的本质、历史及局限性是十分重要的。

本节为您讲解 JavaScript 和客户端脚本的起源。
Nombas 和 ScriptEase
大概在 1992 年，一家称作 Nombas 的公司开发了一种叫做 C 减减（C-minus-minus，简称 Cmm）的嵌入式脚本语言。Cmm 背后的理念很简单：一个足够强大可以替代宏操作（macro）的脚本语言，同时保持与 C （和 C ++）足够的相似性，以便开发人员能很快学会。这个脚本语言捆绑在一个叫做 CEnvi 的共享软件中，它首次向开发人员展示了这种语言的威力。

Nombas 最终把 Cmm 的名字改成了 ScriptEase，原因是后面的部分（mm）听起来过于消极，同时字母 C “令人害怕”。

现在 ScriptEase 已经成为了 Nombas 产品背后的主要驱动力。
Netscape 发明了 JavaScript
当 Netscape Navigator 崭露头角时，Nombas 开发了一个可以嵌入网页中的 CEnvi 的版本。这些早期的试验被称为 Espresso Page（浓咖啡般的页面），它们代表了第一个在万维网上使用的客户端语言。而 Nombas 丝毫没有料到它的理念将会成为万维网的一块重要基石。

当网上冲浪越来越流行时，对于开发客户端脚本的需求也逐渐增大。此时，大部分因特网用户还仅仅通过 28.8 kbit/s 的调制解调器连接到网络，即便这时网页已经不断地变得更大和更复杂。而更加加剧用户痛苦的是，仅仅为了简单的表单有效性验证，就要与服务器进行多次地往返交互。设想一下，用户填完一个表单，点击提交按钮，等待了 30 秒的处理后，看到的却是一条告诉你忘记填写一个必要的字段。

那时正处于技术革新最前沿的 Netscape，开始认真考虑开发一种客户端脚本语言来解决简单的处理问题。

当时工作于 Netscape 的 Brendan Eich，开始着手为即将在 1995 年发行的 Netscape Navigator 2.0 开发一个称之为 LiveScript 的脚本语言，当时的目的是在浏览器和服务器（本来要叫它 LiveWire）端使用它。Netscape 与 Sun 及时完成 LiveScript 实现。

就在 Netscape Navigator 2.0 即将正式发布前，Netscape 将其更名为 JavaScript，目的是为了利用 Java 这个因特网时髦词汇。Netscape 的赌注最终得到回报，JavaScript 从此变成了

因特网的必备组件。

三足鼎立

因为 JavaScript 1.0 如此成功，Netscape 在 Netscape Navigator 3.0 中发布了 1.1 版。恰巧那个时候，微软决定进军浏览器，发布了 IE 3.0 并搭载了一个 JavaScript 的克隆版，叫做 JScript（这样命名是为了避免与 Netscape 潜在的许可纠纷）。微软步入 Web 浏览器领域的这重要一步虽然令其声名狼藉，但也成为 JavaScript 语言发展过程中的重要一步。

在微软进入后，有 3 种不同的 JavaScript 版本同时存在：Netscape Navigator 3.0 中的 JavaScript、IE 中的 JScript 以及 CEnvi 中的 ScriptEase。与 C 和其他编程语言不同的是，JavaScript 并没有一个标准来统一其语法或特性，而这 3 中不同的版本恰恰突出了这个问题。随着业界担心的增加，这个语言的标准化显然已经势在必行。

标准化

1997 年，JavaScript 1.1 作为一个草案提交给欧洲计算机制造商协会（ECMA）。第 39 技术委员会（TC39）被委派来“标准化一个通用、跨平台、中立于厂商的脚本语言的语法和语义”(http://www.ecma-international.org/memento/TC39.htm)。由来自 Netscape、Sun、微软、Borland 和其他一些对脚本编程感兴趣的公司的程序员组成的 TC39 锤炼出了 ECMA-262，该标准定义了名为 ECMAScript 的全新脚本语言。

在接下来的几年里，国际标准化组织及国际电工委员会（ISO/IEC）也采纳 ECMAScript 作为标准（ISO/IEC-16262）。从此，Web 浏览器就开始努力（虽然有着不同的程度的成功和失败）将 ECMAScript 作为 JavaScript 实现的基础。

JavaScript 实现

JavaScript 的核心 ECMAScript 描述了该语言的语法和基本对象；

DOM 描述了处理网页内容的方法和接口；

BOM 描述了与浏览器进行交互的方法和接口。

ECMAScript、DOM 和 BOM

尽管 ECMAScript 是一个重要的标准，但它并不是 JavaScript 唯一的部分，当然，也不是唯一被标准化的部分。实际上，一个完整的 JavaScript 实现是由以下 3 个不同部分组成的：

核心（ECMAScript）
文档对象模型（DOM）
浏览器对象模型（BOM）

注意：大器智诚 PS-LCD 产品只支持核心（ECMAScript） 部分，不支持 DOM 和 BOM 部分！

ECMAScript 并不与任何具体浏览器相绑定，实际上，它也没有提到用于任何用户输入输出的方法（这点与 C 这类语言不同，它需要依赖外部的库来完成这类任务）。那么什么才是 ECMAScript 呢？ECMA-262 标准（第 2 段）的描述如下：

“ECMAScript 可以为不同种类的宿主环境提供核心的脚本编程能力，因此核心的脚本语言是与任何特定的宿主环境分开进行规定的... ...”
Web 浏览器对于 ECMAScript 来说是一个宿主环境，但它并不是唯一的宿主环境。事实上，还有不计其数的其他各种环境（例如 Nombas 的 ScriptEase，以及 Macromedia 同时用在 Flash 和 Director MX 中的 ActionScript）可以容纳 ECMAScript 实现。那么 ECMAScript 在浏览器之外规定了些什么呢？

简单地说，ECMAScript 描述了以下内容：

- 语法
- 类型
- 语句
- 关键字
- 保留字
- 运算符
- 对象

ECMAScript 仅仅是一个描述，定义了脚本语言的所有属性、方法和对象。其他语言可以实现 ECMAScript 来作为功能的基准，JavaScript 就是这样：

每个浏览器都有它自己的 ECMAScript 接口的实现，然后这个实现又被扩展，包含了 DOM 和 BOM（在以下几节中再探讨）。当然还有其他实现并扩展了 ECMAScript 的语言，例如 Windows 脚本宿主（Windows Scripting Host, WSH）、Macromedia 在 Flash 和 Director MX 中的 ActionScript，以及 Nombas ScriptEase。

1. ECMAScript 的版本

ECMAScript 分成几个不同的版本，它是在一个叫做 ECMA-262 的标准中定义的。和其他标准一样，ECMA-262 会被编辑和更新。当有了主要更新时，就会发布一个标准的新版。最新 ECMA-262 的版本是第三版，于 1999 年 12 月发布。

ECMA-262 的第一版在本质上与 Netscape 的 JavaScript 1.1 是一样，只是把所有与浏览器相关的代码删除了，此外还有一些小的调整。首先，ECMA-262 要求对 Unicode 标准的支持（以便支持多语言）。第二，它要求对象是平台无关的（Netscape 的 JavaScript 1.1 事实上有不同的对象实现，例如 Date 对象，是依赖于平台）。这是 JavaScript 1.1 和 1.2 为什么不符合 ECMA-262 规范第一版的主要原因。

ECMA-262 的第二版大部分更新本质上是编辑性的。这次标准的更新是为了与 ISO/IEC-16262 的严格一致，也并没有特别添加、更改和删除内容。ECMAScript 一般不会遵守第二版。

ECMA-262 第三版是该标准第一次真正的更新。它提供了对字符串处理、错误定义和数值输出的更新。同时，它还增加了正则表达式、新的控制语句、try...catch 异常处理的支持，以及一些为使标准国际化而做的小改动。一般来说，它标志着 ECMAScript 成为一种真正的编

程语言。

2. 何谓 ECMAScript 符合性

在 ECMA-262 中，ECMAScript 符合性（conformance）有明确的定义。一个脚本语言必须满足以下四项基本原则：

符合的实现必须按照 ECMA-262 中所描述的支持所有的“类型、值、对象、属性、函数和程序语言及语义”（ECMA-262，第一页）
符合的实现必须支持 Unicode 字符标准（UCS）
符合的实现可以增加没有在 ECMA-262 中指定的“额外类型、值、对象、属性和函数”。ECMA-262 将这些增加描述为规范中未给定的新对象或对象的新属性
符合的实现可以支持没有在 ECMA-262 中定义的“程序和正则表达式语法”（意思是可以替换或者扩展内建的正则表达式支持）
所有 ECMAScript 实现必须符合以上标准。

3. Web 浏览器中的 ECMAScript 支持

含有 JavaScript 1.1 的 Netscape Navigator 3.0 在 1996 年发布。然后，JavaScript 1.1 规范被作为一个新标准的草案被提交给 EMCA。有了 JavaScript 轰动性的流行，Netscape 十分高兴地开始开发 1.2 版。但有一个问题，ECMA 并未接受 Netscape 的草案。在 Netscape Navigator 3.0 发布后不久，微软就发布了 IE 3.0。该版本的 IE 含有 JScript 1.0（微软自己的 JavaScript 实现的名称），原本计划可以与 JavaScript 1.1 相提并论。然后，由于文档不全以及一些不当的重复特性，JScript 1.0 远远没有达到 JavaScript 1.1 的水平。

在 ECMA-262 第一版定稿之前，发布含有 JavaScript 1.2 的 Netscape Navigator 4.0 是在 1997 年，在那年晚些时候，ECMA-262 标准被接受并标准化。因此，JavaScript 1.2 并不和 ECMAScript 的第一版兼容，虽然 ECMAScript 应该基于 JavaScript 1.1。

JScript 的下一步是 IE 4.0 中加入的 JScript 3.0（2.0 版是随 IIS 3.0 一起发布的，但并未包含在浏览器中）。微软大力宣传 JScript 3.0 是世界上第一个真正符合 ECMA 标准的脚本语言。而那时，ECMA-262 还没有最终定稿，所以 JScript 3.0 也遭受了和 JavaScript 1.2 同样的命运 - 它还是没能符合最终的 ECMAScript 标准。

Netscape 选择在 Netscape Navigator 4.06 中升级它的 JavaScript 实现。JavaScript 1.3 使 Netscape 终于完全符合了 ECMAScript 第一版。Netscape 加入了对 Unicode 标准的支持，并让所有的对象保留了在 JavaScript 1.2 中引入的新特性的同时实现了平台独立。

当 Netscape 将它的源代码作为 Mozilla 项目公布于众时，本来计划 JavaScript 1.4 将会嵌入到 Netscape Navigator 5.0 中。然而，一个冒进的决定 - 要完全从头重新设计 Netscape 的代码，破坏了这个工作。JavaScript 1.4 仅仅作为一个 Netscape Enterprise Server 的服务器端脚本语言发布，以后也没有被放入浏览器中。

如今，所有主流的 Web 浏览器都遵守 ECMA-262 第三版。

2 ECMAScript 基础

语法

熟悉 Java、C 和 Perl 这些语言的开发者会发现 ECMAScript 的语法很容易掌握，因为它借用了这些语言的语法。

Java 和 ECMAScript 有一些关键的语法特性相同，也有一些完全不同。
区分大小写
与 Java 一样，变量、函数名、运算符以及其他一切东西都是区分大小写的。

比如：变量 test 与变量 TEST 是不同的。
变量是弱类型的
与 Java 和 C 不同，ECMAScript 中的变量无特定的类型，定义变量时只用 var 运算符，可以将它初始化为任意值。

因此，可以随时改变变量所存数据的类型（尽量避免这样做）。

例子

```
var color = "red";
var num = 25;
var visible = true;
```

每行结尾的分号可有可无
Java、C 和 Perl 都要求每行代码以分号（;）结束才符合语法。

ECMAScript 则允许开发者自行决定是否以分号结束一行代码。如果没有分号，ECMAScript 就把折行代码的结尾看做该语句的结尾（与 Visual Basic 和 VBScript 相似），前提是这样没有破坏代码的语义。

最好的代码编写习惯是总加入分号，因为没有分号，有些浏览器就不能正确运行，不过根据 ECMAScript 标准，下面两行代码都是正确的：

```
var test1 = "red"
var test2 = "blue";
```

注释与 Java、C 和 PHP 语言的注释相同
ECMAScript 借用了这些语言的注释语法。

有两种类型的注释：

单行注释以双斜杠开头（//）
多行注释以单斜杠和星号开头（/*），以星号和单斜杠结尾（*/）

```
//this is a single-line comment

/*this is a multi-
line comment*/
```

括号表示代码块
从 Java 中借鉴的另一个概念是代码块。

代码块表示一系列应该按顺序执行的语句，这些语句被封装在左括号（{）和右括号（}）之间。

例如：

```
if (test1 == "red") {
        test1 = "blue";
        alert(test1);
}
```

变量

请使用 var 运算符声明变量。

变量名需要遵守一些简单的规则。
声明变量
在上一节中我们讲解过，ECMAScript 中的变量是用 var 运算符（variable 的缩写）加变量名定义的。例如：

var test = "hi";在这个例子中，声明了变量 test，并把它的值初始化为 "hi"（字符串）。由于 ECMAScript 是弱类型的，所以解释程序会为 test 自动创建一个字符串值，无需明确的类型声明。

还可以用一个 var 语句定义两个或多个变量：

var test1 = "hi", test2 = "hello";前面的代码定义了变量 test1，初始值为 "hi"，还定义了变量 test2，初始值为 "hello"。

不过用同一个 var 语句定义的变量不必具有相同的类型，如下所示：

var test = "hi", age = 25;这个例子除了（再次）定义 test 外，还定义了 age，并把它初始化为 25。即使 test 和 age 属于两种不同的数据类型，在 ECMAScript 中这样定义也是完全合法的。

与 Java 不同，ECMAScript 中的变量并不一定要初始化（它们是在幕后初始化的，将在后面讨论这一点）。因此，下面这一行代码也是有效的：

var test;此外，与 Java 不同的还有变量可以存放不同类型的值。这是弱类型变量的优势。例如，可以把变量初始化为字符串类型的值，之后把它设置为数字值，如下所示：

```
var test = "hi";
alert(test);
test = 55;
alert(test);
```

这段代码将毫无问题地输出字符串值和数字值。但是，如前所述，使用变量时，好的编码习惯是始终存放相同类型的值。

命名变量

变量名需要遵守两条简单的规则：

第一个字符必须是字母、下划线（_）或美元符号（$）

余下的字符可以是下划线、美元符号或任何字母或数字字符

下面的变量都是合法的：

```
var test;
var $test;
var $1;
var _$te$t2;
```

著名的变量命名规则

只是因为变量名的语法正确，并不意味着就该使用它们。变量还应遵守以下某条著名的命名规则：

Camel 标记法

首字母是小写的，接下来的字母都以大写字符开头。例如：

```
var myTestValue = 0, mySecondValue = "hi";
```

Pascal 标记法

首字母是大写的，接下来的字母都以大写字符开头。例如：

```
var MyTestValue = 0, MySecondValue = "hi";
```

匈牙利类型标记法

在以 Pascal 标记法命名的变量前附加一个小写字母（或小写字母序列），说明该变量的类型。例如，i 表示整数，s 表示字符串，如下所示 “

```
var iMyTestValue = 0, sMySecondValue = "hi";
```

本教程采用了这些前缀，以使示例代码更易阅读：

类型	前缀	示例
数组	a	aValues
布尔型	b	bFound
浮点型（数字）	f	fValue
函数	fn	fnMethod
整型（数字）	i	iValue
对象	o	oType
正则表达式	re	rePattern
字符串	s	sValue
变型（可以是任何类型）	v	vValue

变量声明不是必须的

ECMAScript 另一个有趣的方面（也是与大多数程序设计语言的主要区别），是在使用变量之前不必声明。例如：

```
var sTest = "hello ";
sTest2 = sTest + "world";
alert(sTest2);
```

在上面的代码中，首先，sTest 被声明为字符串类型的值 "hello"。接下来的一行，用变量 sTest2 把 sTest 与字符串 "world" 连在一起。变量 sTest2 并没有用 var 运算符定义，这里只是插入了它，就像已经声明过它一样。

ECMAScript 的解释程序遇到未声明过的标识符时，用该变量名创建一个全局变量，并将其初始化为指定的值。

这是该语言的便利之处，不过如果不能紧密跟踪变量，这样做也很危险。最好的习惯是像使用其他程序设计语言一样，总是声明所有变量。

关键字

本节提供完整的 ECMAScript 关键字列表。

ECMAScript 关键字

ECMA-262 定义了 ECMAScript 支持的一套关键字（keyword）。

这些关键字标识了 ECMAScript 语句的开头和/或结尾。根据规定，关键字是保留的，不能用作变量名或函数名。

下面是 ECMAScript 关键字的完整列表：

break
case
catch

continue
default
delete
do
else
finally
for
function
if
in
instanceof
new
return
switch
this
throw
try
typeof
var
void
while
with

注意：如果把关键字用作变量名或函数名，可能得到诸如 "Identifier Expected"（应该有标识符、期望标识符）这样的错误消息。

保留字

本节提供完整的 ECMAScript 保留字列表。

ECMAScript 保留字

ECMA-262 定义了 ECMAScript 支持的一套保留字（reserved word）。

保留字在某种意思上是为将来的关键字而保留的单词。因此保留字不能被用作变量名或函数名。

ECMA-262 第三版中保留字的完整列表如下：

abstract
boolean
byte
char
class
const

debugger
double
enum
export
extends
final
float
goto
implements
import
int
interface
long
native
package
private
protected
public
short
static
super
synchronized
throws
transient
volatile

注意：如果将保留字用作变量名或函数名，那么除非将来的浏览器实现了该保留字，否则很可能收不到任何错误消息。当浏览器将其实现后，该单词将被看做关键字，如此将出现关键字错误。

原始值和引用值

在 ECMAScript 中，变量可以存在两种类型的值，即原始值和引用值。

原始值和引用值

在 ECMAScript 中，变量可以存在两种类型的值，即原始值和引用值。

原始值

存储在栈（stack）中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

引用值

存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存处。

为变量赋值时，ECMAScript 的解释程序必须判断该值是原始类型，还是引用类型。要实现这一点，解释程序则需尝试判断该值是否为 ECMAScript 的原始类型之一，即 Undefined、 Null、Boolean、Number 和 String 型。由于这些原始类型占据的空间是固定的，所以可将他们存储在较小的内存区域 - 栈中。这样存储便于迅速查寻变量的值。

在许多语言中，字符串都被看作引用类型，而非原始类型，因为字符串的长度是可变的。ECMAScript 打破了这一传统。

如果一个值是引用类型的，那么它的存储空间将从堆中分配。由于引用值的大小会改变，所以不能把它放在栈中，否则会降低变量查寻的速度。相反，放在变量的栈空间中的值是该对象存储在堆中的地址。地址的大小是固定的，所以把它存储在栈中对变量性能无任何负面影响。

原始类型
如前所述，ECMAScript 有 5 种原始类型（primitive type），即 Undefined、Null、Boolean、Number 和 String。EMCA-262 把术语类型（type）定义为值的一个集合，每种原始类型定义了它包含的值的范围及其字面量表示形式。

ECMAScript 提供了 typeof 运算符来判断一个值是否在某种类型的范围内。可以用这种运算符判断一个值是否表示一种原始类型：如果它是原始类型，还可以判断它表示哪种原始类型。

在稍后的章节，我们将为您深入讲解 ECMAScript 的原始类型和引用类型。

原始类型

ECMAScript 有 5 种原始类型（primitive type），即 Undefined、Null、Boolean、Number 和 String。
typeof 运算符
typeof 运算符有一个参数，即要检查的变量或值。例如：

```
var sTemp = "test string";
alert (typeof sTemp);      //输出 "string"
alert (typeof 86);      //输出 "number"
```

对变量或值调用 typeof 运算符将返回下列值之一：

undefined - 如果变量是 Undefined 类型的
boolean - 如果变量是 Boolean 类型的
number - 如果变量是 Number 类型的
string - 如果变量是 String 类型的
object - 如果变量是一种引用类型或 Null 类型的
注释：您也许会问，为什么 typeof 运算符对于 null 值会返回 "Object"。这实际上是 JavaScript 最初实现中的一个错误，然后被 ECMAScript 沿用了。现在，null 被认为是对象的占位符，从而解释了这一矛盾，但从技术上来说，它仍然是原始值。
Undefined 类型

如前所述，Undefined 类型只有一个值，即 undefined。当声明的变量未初始化时，该变量的默认值是 undefined。

var oTemp;前面一行代码声明变量 oTemp，没有初始值。该变量将被赋予值 undefined，即 undefined 类型的字面量。可以用下面的代码段测试该变量的值是否等于 undefined：

```
var oTemp;
alert(oTemp == undefined);
```

这段代码将显示 "true"，说明这两个值确实相等。还可以用 typeof 运算符显示该变量的值是 undefined：

```
var oTemp;
alert(typeof oTemp); //输出 "undefined"
```

提示：值 undefined 并不同于未定义的值。但是，typeof 运算符并不真正区分这两种值。考虑下面的代码：

```
var oTemp;

alert(typeof oTemp);   //输出 "undefined"
alert(typeof oTemp2);   //输出 "undefined"
```

前面的代码对两个变量输出的都是 "undefined"，即使只有变量 oTemp2 从未被声明过。如果对 oTemp2 使用除 typeof 之外的其他运算符的话，会引起错误，因为其他运算符只能用于已声明的变量上。

例如，下面的代码将引发错误：

```
var oTemp;
alert(oTemp2 == undefined);
```

当函数无明确返回值时，返回的也是值 "undefined"，如下所示：

```
function testFunc() {
}

alert(testFunc() == undefined);   //输出 "true"
```

Null 类型

另一种只有一个值的类型是 Null，它只有一个专用值 null，即它的字面量。值 undefined 实际上是从值 null 派生来的，因此 ECMAScript 把它们定义为相等的。

```
alert(null == undefined);   //输出 "true"
```

尽管这两个值相等，但它们的含义不同。undefined 是声明了变量但未对其初始化时赋予该变量的值，null 则用于表示尚未存在的对象（在讨论 typeof 运算符时，简单地介绍过这一点）。如果函数或方法要返回的是对象，那么找不到该对象时，返回的通常是 null。

Boolean 类型

Boolean 类型是 ECMAScript 中最常用的类型之一。它有两个值 true 和 false （即两个 Boolean 字面量）。

即使 false 不等于 0，0 也可以在必要时被转换成 false，这样在 Boolean 语句中使用两者都是安全的。

var bFound = true;
var bLost = false;
Number 类型
ECMA-262 中定义的最特殊的类型是 Number 类型。这种类型既可以表示 32 位的整数，还可以表示 64 位的浮点数。

直接输入的（而不是从另一个变量访问的）任何数字都被看做 Number 类型的字面量。例如，下面的代码声明了存放整数值的变量，它的值由字面量 86 定义：

var iNum = 86;八进制数和十六进制数
整数也可以被表示为八进制（以 8 为底）或十六进制（以 16 为底）的字面量。八进制字面量的首数字必须是 0，其后的数字可以是任何八进制数字（0-7），如下面的代码所示：

var iNum = 070; //070 等于十进制的 56 要创建十六进制的字面量，首位数字必须为 0，后面接字母 x，然后是任意的十六进制数字（0 到 9 和 A 到 F）。这些字母可以是大写的，也可以是小写的。例如：

var iNum = 0x1f; //0x1f 等于十进制的 31
var iNum = 0xAB; //0xAB 等于十进制的 171
提示：尽管所有整数都可以表示为八进制或十六进制的字面量，但所有数学运算返回的都是十进制结果。

浮点数
要定义浮点值，必须包括小数点和小数点后的一位数字（例如，用 1.0 而不是 1）。这被看作浮点数字面量。例如：

var fNum = 5.0;对于浮点字面量的有趣之处在于，用它进行计算前，真正存储的是字符串。

科学计数法
对于非常大或非常小的数，可以用科学计数法表示浮点数，可以把一个数表示为数字（包括十进制数字）加 e（或 E），后面加乘以 10 的倍数。例如：

var fNum = 5.618e7 该符号表示的是数 56180000。把科学计数法转化成计算式就可以得到该值：5.618 x 107。

也可以用科学计数法表示非常小的数，例如 0.00000000000000008 可以表示为 8-e17（这里，10 被升到 -17 次冥，意味着需要被 10 除 17 次）。ECMAScript 默认把具有 6 个或 6

个以上前导 0 的浮点数转换成科学计数法。

提示：也可用 64 位 IEEE 754 形式存储浮点值，这意味着十进制值最多可以有 17 个十进制位。17 位之后的值将被裁去，从而造成一些小的数学误差。

特殊的 Number 值

几个特殊值也被定义为 Number 类型。前两个是 Number.MAX_VALUE 和 Number.MIN_VALUE，它们定义了 Number 值集合的外边界。所有 ECMAScript 数都必须在这两个值之间。不过计算生成的数值结果可以不落在这两个值之间。

当计算生成的数大于 Number.MAX_VALUE 时，它将被赋予值 Number.POSITIVE_INFINITY，意味着不再有数字值。同样，生成的数值小于 Number.MIN_VALUE 的计算也会被赋予值 Number.NEGATIVE_INFINITY，也意味着不再有数字值。如果计算返回的是无穷大值，那么生成的结果不能再用于其他计算。

事实上，有专门的值表示无穷大，（如你猜到的）即 Infinity。Number.POSITIVE_INFINITY 的值为 Infinity。Number.NEGATIVE_INFINITY 的值为 -Infinity。

由于无穷大数可以是正数也可以是负数，所以可用一个方法判断一个数是否是有穷的（而不是单独测试每个无穷数）。可以对任何数调用 isFinite() 方法，以确保该数不是无穷大。例如：

```
var iResult = iNum * some_really_large_number;

if (isFinite(iResult)) {
    alert("finite");
}

else {
    alert("infinite");
}
```

最后一个特殊值是 NaN，表示非数（Not a Number）。NaN 是个奇怪的特殊值。一般说来，这种情况发生在类型（String、Boolean 等）转换失败时。例如，要把单词 blue 转换成数值就会失败，因为没有与之等价的数值。与无穷大一样，NaN 也不能用于算术计算。NaN 的另一个奇特之处在于，它与自身不相等，这意味着下面的代码将返回 false：

alert(NaN == NaN);   //输出 "false"出于这个原因，不推荐使用 NaN 值本身。函数 isNaN() 会做得相当好：

```
alert(isNaN("blue"));   //输出 "true"
alert(isNaN("666"));   //输出 "false"
```

String 类型

String 类型的独特之处在于，它是唯一没有固定大小的原始类型。可以用字符串存储 0 或更多的 Unicode 字符，有 16 位整数表示（Unicode 是一种国际字符集，本教程后面将讨

论它）。

字符串中每个字符都有特定的位置，首字符从位置 0 开始，第二个字符在位置 1，依此类推。这意味着字符串中的最后一个字符的位置一定是字符串的长度减 1：

字符串字面量是由双引号（"）或单引号（'）声明的。而 Java 则是用双引号声明字符串，用单引号声明字符。但是由于 ECMAScript 没有字符类型，所以可使用这两种表示法中的任何一种。例如，下面的两行代码都有效：

```
var sColor1 = "red";
var sColor2 = 'red';
```

String 类型还包括几种字符字面量，Java、C 和 Perl 的开发者应该对此非常熟悉。

下面列出了 ECMAScript 的字符字面量：

字面量	含义
\n	换行
\t	制表符
\b	空格
\r	回车
\f	换页符
\\	反斜杠
\'	单引号
\"	双引号
\0nnn	八进制代码 nnn 表示的字符（n 是 0 到 7 中的一个八进制数字）
\xnn	十六进制代码 nn 表示的字符（n 是 0 到 F 中的一个十六进制数字）
\unnnn	十六进制代码 nnnn 表示的 Unicode 字符（n 是 0 到 F 中的一个十六进制数字）

类型转换

所有程序设计语言最重要的特征之一是具有进行类型转换的能力。

ECMAScript 给开发者提供了大量简单的类型转换方法。

大部分类型具有进行简单转换的方法，还有几个全局方法可以用于更复杂的转换。无论哪种情况，在 ECMAScript 中，类型转换都是简短的一步操作。

转换成字符串

ECMAScript 的 Boolean 值、数字和字符串的原始值的有趣之处在于它们是伪对象，这意味着它们实际上具有属性和方法。

例如，要获得字符串的长度，可以采用下面的代码：

```
var sColor = "red";
alert(sColor.length);      //输出 "3"
```

尽管 "red" 是原始类型的字符串，它仍然具有属性 length，用于存放字符串的大小。

总而言之，3 种主要的原始类型 Boolean 值、数字和字符串都有 toString() 方法，可以把它们的值转换成字符串。

提示：您也许会问，“字符串还有 toString() 方法吗，这不是多余吗？”是的，的确如此，不过 ECMAScript 定义所有对象都有 toString() 方法，无论它是伪对象，还是真对象。因为 String 类型属于伪对象，所以它一定有 toString() 方法。

Boolean 类型的 toString() 方法只是输出 "true" 或 "false"，结果由变量的值决定：

```
var bFound = false;
alert(bFound.toString());//输出 "false"
```

Number 类型的 toString() 方法比较特殊，它有两种模式，即默认模式和基模式。采用默认模式，toString() 方法只是用相应的字符串输出数字值（无论是整数、浮点数还是科学计数法），如下所示：

```
var iNum1 = 10;
var iNum2 = 10.0;
alert(iNum1.toString()); //输出 "10"
alert(iNum2.toString()); //输出 "10"
```

注释：在默认模式中，无论最初采用什么表示法声明数字，Number 类型的 toString() 方法返回的都是数字的十进制表示。因此，以八进制或十六进制字面量形式声明的数字输出的都是十进制形式的。

采用 Number 类型的 toString() 方法的基模式，可以用不同的基输出数字，例如二进制的基是 2，八进制的基是 8，十六进制的基是 16。

基只是要转换成的基数的另一种加法而已，它是 toString() 方法的参数：

```
var iNum = 10;
alert(iNum1.toString(2));      //输出 "1010"
alert(iNum1.toString(8));      //输出 "12"
alert(iNum1.toString(16));     //输出 "A"
```

在前面的示例中，以 3 种不同的形式输出了数字 10，即二进制形式、八进制形式和十六进制形式。HTML 采用十六进制表示每种颜色，在 HTML 中处理数字时这种功能非常有用。

注释：对数字调用 toString(10) 与调用 toString() 相同，它们返回的都是该数字的十进制形式。

参阅：

请参阅 JavaScript 参考手册提供的有关 toString() 方法的详细信息：

arrayObject.toString()
booleanObject.toString()
dateObject.toString()
NumberObject.toString()
stringObject.toString()

转换成数字

ECMAScript 提供了两种把非数字的原始值转换成数字的方法,即 parseInt() 和 parseFloat()。

正如您可能想到的，前者把值转换成整数，后者把值转换成浮点数。只有对 String 类型调用这些方法，它们才能正确运行；对其他类型返回的都是 NaN。

parseInt()

在判断字符串是否是数字值前，parseInt() 和 parseFloat() 都会仔细分析该字符串。

parseInt() 方法首先查看位置 0 处的字符，判断它是否是个有效数字；如果不是，该方法将返回 NaN，不再继续执行其他操作。但如果该字符是有效数字，该方法将查看位置 1 处的字符，进行同样的测试。这一过程将持续到发现非有效数字的字符为止，此时 parseInt() 将把该字符之前的字符串转换成数字。

例如，如果要把字符串 "12345red" 转换成整数，那么 parseInt() 将返回 12345，因为当它检查到字符 r 时，就会停止检测过程。

字符串中包含的数字字面量会被正确转换为数字，比如 "0xA" 会被正确转换为数字 10。不过，字符串 "22.5" 将被转换成 22，因为对于整数来说，小数点是无效字符。

一些示例如下：

```
var iNum1 = parseInt("12345red");      //返回 12345
var iNum1 = parseInt("0xA"); //返回 10
var iNum1 = parseInt("56.9");//返回 56
var iNum1 = parseInt("red"); //返回 NaN
```

parseInt() 方法还有基模式，可以把二进制、八进制、十六进制或其他任何进制的字符串转换成整数。基是由 parseInt() 方法的第二个参数指定的，所以要解析十六进制的值，需如下调用 parseInt() 方法：

var iNum1 = parseInt("AF", 16);   //返回 175 当然,对二进制、八进制甚至十进制(默认模式),都可以这样调用 parseInt() 方法：

```
var iNum1 = parseInt("10", 2);     //返回 2
var iNum2 = parseInt("10", 8);     //返回 8
```

```
var iNum3 = parseInt("10", 10);     //返回 10
```

如果十进制数包含前导 0，那么最好采用基数 10，这样才不会意外地得到八进制的值。例如：

```
var iNum1 = parseInt("010"); //返回 8
var iNum2 = parseInt("010", 8);     //返回 8
var iNum3 = parseInt("010", 10);   //返回 10
```

在这段代码中，两行代码都把字符 "010" 解析成一个数字。第一行代码把这个字符串看作八进制的值，解析它的方式与第二行代码（声明基数为 8）相同。最后一行代码声明基数为 10，所以 iNum3 最后等于 10。

参阅

请参阅 JavaScript 参考手册提供的有关 parseInt() 方法的详细信息：parseInt()。

parseFloat()

parseFloat() 方法与 parseInt() 方法的处理方式相似，从位置 0 开始查看每个字符，直到找到第一个非有效的字符为止，然后把该字符之前的字符串转换成整数。

不过，对于这个方法来说，第一个出现的小数点是有效字符。如果有两个小数点，第二个小数点将被看作无效的。parseFloat() 会把这个小数点之前的字符转换成数字。这意味着字符串 "11.22.33" 将被解析成 11.22。

使用 parseFloat() 方法的另一不同之处在于，字符串必须以十进制形式表示浮点数，而不是用八进制或十六进制。该方法会忽略前导 0，所以八进制数 0102 将被解析为 102。对于十六进制数 0xA，该方法将返回 NaN，因为在浮点数中，x 不是有效字符。

此外，parseFloat() 方法也没有基模式。

下面是使用 parseFloat() 方法的一些示例：

```
var fNum1 = parseFloat("12345red");   //返回 12345
var fNum2 = parseFloat("0xA");     //返回 NaN
var fNum3 = parseFloat("11.2");    //返回 11.2
var fNum4 = parseFloat("11.22.33");    //返回 11.22
var fNum5 = parseFloat("0102");   //返回 102
var fNum1 = parseFloat("red");      //返回 NaN
```

参阅

请参阅 JavaScript 参考手册提供的有关 parseFloat() 方法的详细信息：parseFloat()。

强制类型转换

您还可以使用强制类型转换（type casting）来处理转换值的类型。使用强制类型转换可以访问特定的值，即使它是另一种类型的。

编者注：cast 有“铸造”之意，很贴合“强制转换”的意思。

ECMAScript 中可用的 3 种强制类型转换如下：

Boolean(value) - 把给定的值转换成 Boolean 型；
Number(value) - 把给定的值转换成数字（可以是整数或浮点数）；
String(value) - 把给定的值转换成字符串；
用这三个函数之一转换值，将创建一个新值，存放由原始值直接转换成的值。这会造成意想不到的后果。

Boolean() 函数
当要转换的值是至少有一个字符的字符串、非 0 数字或对象时，Boolean() 函数将返回 true。如果该值是空字符串、数字 0、undefined 或 null，它将返回 false。

可以用下面的代码测试 Boolean 型的强制类型转换：

```
var b1 = Boolean("");          //false - 空字符串
var b2 = Boolean("hello");          //true - 非空字符串
var b1 = Boolean(50);          //true - 非零数字
var b1 = Boolean(null);        //false - null
var b1 = Boolean(0);           //false - 零
var b1 = Boolean(new object());     //true - 对象
```

Number() 函数
Number() 函数的强制类型转换与 parseInt() 和 parseFloat() 方法的处理方式相似，只是它转换的是整个值，而不是部分值。

还记得吗，parseInt() 和 parseFloat() 方法只转换第一个无效字符之前的字符串，因此 "1.2.3" 将分别被转换为 "1" 和 "1.2"。

用 Number() 进行强制类型转换，"1.2.3" 将返回 NaN，因为整个字符串值不能转换成数字。如果字符串值能被完整地转换，Number() 将判断是调用 parseInt() 方法还是 parseFloat() 方法。

下表说明了对不同的值调用 Number() 方法会发生的情况：

用法	结果
Number(false)	0
Number(true)	1
Number(undefined)	NaN
Number(null)	0
Number("1.2")	1.2
Number("12")	12
Number("1.2.3")	NaN
Number(new object())	NaN

Number(50) 50

String() 函数
最后一种强制类型转换方法 String() 是最简单的，因为它可把任何值转换成字符串。

要执行这种强制类型转换，只需要调用作为参数传递进来的值的 toString() 方法，即把 12 转换成 "12"，把 true 转换成 "true"，把 false 转换成 "false"，以此类推。

强制转换成字符串和调用 toString() 方法的唯一不同之处在于，对 null 和 undefined 值强制类型转换可以生成字符串而不引发错误：

```
var s1 = String(null);      //"null"
var oNull = null;
var s2 = oNull.toString(); //会引发错误
```

在处理 ECMAScript 这样的弱类型语言时，强制类型转换非常有用，不过应该确保使用值的正确。

引用类型

引用类型通常叫做类（class）。

本教程会讨论大量的 ECMAScript 预定义引用类型。
引用类型
引用类型通常叫做类（class），也就是说，遇到引用值，所处理的就是对象。

本教程会讨论大量的 ECMAScript 预定义引用类型。

从现在起，将重点讨论与已经讨论过的原始类型紧密相关的引用类型。

注意：从传统意义上来说，ECMAScript 并不真正具有类。事实上，除了说明不存在类，在 ECMA-262 中根本没有出现“类”这个词。ECMAScript 定义了“对象定义”，逻辑上等价于其他程序设计语言中的类。

提示：本教程将使用术语“对象”。

对象是由 new 运算符加上要实例化的对象的名字创建的。例如，下面的代码创建 Object 对象的实例：

var o = new Object();这种语法与 Java 语言的相似，不过当有不止一个参数时，ECMAScript 要求使用括号。如果没有参数，如以下代码所示，括号可以省略：

var o = new Object;注意：尽管括号不是必需的，但是为了避免混乱，最好使用括号。

提示：我们会在对象基础这一章中更深入地探讨对象及其行为。

这一节的重点是具有等价的原始类型的引用类型。

Object 对象

Object 对象自身用处不大，不过在了解其他类之前，还是应该了解它。因为 ECMAScript 中的 Object 对象与 Java 中的 java.lang.Object 相似，ECMAScript 中的所有对象都由这个对象继承而来，Object 对象中的所有属性和方法都会出现在其他对象中，所以理解了 Object 对象，就可以更好地理解其他对象。

Object 对象具有下列属性：
constructor
对创建对象的函数的引用（指针）。对于 Object 对象，该指针指向原始的 Object() 函数。
Prototype
对该对象的对象原型的引用。对于所有的对象，它默认返回 Object 对象的一个实例。
Object 对象还具有几个方法：
hasOwnProperty(property)
判断对象是否有某个特定的属性。必须用字符串指定该属性。（例如，o.hasOwnProperty("name")）
IsPrototypeOf(object)
判断该对象是否为另一个对象的原型。
PropertyIsEnumerable
判断给定的属性是否可以用 for...in 语句进行枚举。
ToString()
返回对象的原始字符串表示。对于 Object 对象，ECMA-262 没有定义这个值，所以不同的 ECMAScript 实现具有不同的值。
ValueOf()
返回最适合该对象的原始值。对于许多对象，该方法返回的值都与 ToString() 的返回值相同。
注释：上面列出的每种属性和方法都会被其他对象覆盖。

Boolean 对象

Boolean 对象是 Boolean 原始类型的引用类型。

要创建 Boolean 对象，只需要传递 Boolean 值作为参数：

var oBooleanObject = new Boolean(true);Boolean 对象将覆盖 Object 对象的 ValueOf() 方法，返回原始值，即 true 和 false。ToString() 方法也会被覆盖，返回字符串 "true" 或 "false"。

遗憾的是，在 ECMAScript 中很少使用 Boolean 对象，即使使用，也不易理解。

问题通常出现在 Boolean 表达式中使用 Boolean 对象时。例如：

```
var oFalseObject = new Boolean(false);
var bResult = oFalseObject && true;      //输出 true
```

在这段代码中，用 false 值创建 Boolean 对象。然后用这个值与原始值 true 进行 AND 操作。在 Boolean 运算中，false 和 true 进行 AND 操作的结果是 false。不过，在这行代码中，计算的是 oFalseObject，而不是它的值 false。

正如前面讨论过的，在 Boolean 表达式中，所有对象都会被自动转换为 true，所以 oFalseObject 的值是 true。然后 true 再与 true 进行 AND 操作，结果为 true。

注意：虽然你应该了解 Boolean 对象的可用性，不过最好还是使用 Boolean 原始值，避免发生这一节提到的问题。

Number 对象

Number 对象是原始数值的包装对象。

创建 Number 对象的语法：

```
var myNum=new Number(value);
var myNum=Number(value);
```

参数

参数 value 是要创建的 Number 对象的数值，或是要转换成数字的值。

返回值

当 Number() 和运算符 new 一起作为构造函数使用时，它返回一个新创建的 Number 对象。如果不用 new 运算符，把 Number() 作为一个函数来调用，它将把自己的参数转换成一个原始的数值，并且返回这个值（如果转换失败，则返回 NaN）。

Number 对象属性

属性	描述
constructor	返回对创建此对象的 Number 函数的引用。
MAX_VALUE	可表示的最大的数。
MIN_VALUE	可表示的最小的数。
NaN	非数字值。
NEGATIVE_INFINITY	负无穷大，溢出时返回该值。
POSITIVE_INFINITY	正无穷大，溢出时返回该值。
prototype	使您有能力向对象添加属性和方法。

Number 对象方法

方法	描述
toString	把数字转换为字符串，使用指定的基数。

toLocaleString	把数字转换为字符串，使用本地数字格式顺序。
toFixed	把数字转换为字符串，结果的小数点后有指定位数的数字。
toExponential	把对象的值转换为指数计数法。
toPrecision	把数字格式化为指定的长度。
valueOf	返回一个 Number 对象的基本数字值。

toFixed() 方法

toFixed() 方法返回的是具有指定位数小数的数字的字符串表示。例如：

```
var oNumberObject = new Number(68);
alert(oNumberObject.toFixed(2));  //输出 "68.00"
```

在这里，toFixed() 方法的参数是 2，说明应该显示两位小数。该方法返回 "68.00"，空的字符串位由 0 来补充。对于处理货币的应用程序，该方法非常有用。toFixed() 方法能表示具有 0 到 20 位小数的数字，超过这个范围的值会引发错误。

toExponential() 方法

与格式化数字相关的另一个方法是 toExponential()，它返回的是用科学计数法表示的数字的字符串形式。

与 toFixed() 方法相似，toExponential() 方法也有一个参数，指定要输出的小数的位数。例如：

```
var oNumberObject = new Number(68);
alert(oNumberObject.toExponential(1));  //输出 "6.8e+1"
```

这段代码的结果是 "6.8e+1"，前面解释过，它表示 6.8x101。问题是，如果不知道要用哪种形式（预定形式或指数形式）表示数字怎么办？可以用 toPrecision() 方法。

toPrecision() 方法

toPrecision() 方法根据最有意义的形式来返回数字的预定形式或指数形式。它有一个参数，即用于表示数的数字总数（不包括指数）。例如，var oNumberObject = new Number(68);
alert(oNumberObject.toPrecision(1)); //输出 "7e+1"
这段代码的任务是用一位数字表示数字 68，结果为 "7e+1"，以另外的形式表示即 70。的确，toPrecision() 方法会对数进行舍入。不过，如果用 2 位数字表示 68，就容易多了：var oNumberObject = new Number(68);
alert(oNumberObject.toPrecision(2)); //输出 "68"
当然，输出的是 "68"，因为这正是该数的准确表示。不过，如果指定的位数多于需要的位

数又如何呢？var oNumberObject = new Number(68);
alert(oNumberObject.toPrecision(3)); //输出 "68.0"
在这种情况下，toPrecision(3) 等价于 toFixed(1)，输出的是 "68.0"。toFixed()、toExponential() 和 toPrecision() 方法都会进行舍入操作，以便用正确的小数位数正确地表示一个数。提示：与 Boolean 对象相似，Number 对象也很重要，不过应该少用这种对象，以避免潜在的问题。只要可能，都使用数字的原始表示法。参阅如需更多有关 Number 对象的信息，请访问 JavaScript Number 对象参考手册。

String 对象

String 对象用于处理文本（字符串）。

创建 String 对象的语法：
new String(s);
String(s);
参数
参数 s 是要存储在 String 对象中或转换成原始字符串的值。
返回值
当 String() 和运算符 new 一起作为构造函数使用时，它返回一个新创建的 String 对象，存放的是字符串 s 或 s 的字符串表示。当不用 new 运算符调用 String() 时，它只把 s 转换成原始的字符串，并返回转换后的值。
String 对象属性

属性	描述
constructor	对创建该对象的函数的引用
length	字符串的长度
prototype	允许您向对象添加属性和方法

String 对象方法

方法	描述
anchor()	创建 HTML 锚。
big()	用大号字体显示字符串。
blink()	显示闪动字符串。
bold()	使用粗体显示字符串。
charAt()	返回在指定位置的字符。
charCodeAt()	返回在指定的位置的字符的 Unicode 编码。
concat()	连接字符串。
fixed()	以打字机文本显示字符串。
fontcolor()	使用指定的颜色来显示字符串。
fontsize()	使用指定的尺寸来显示字符串。

fromCharCode()	从字符编码创建一个字符串。
indexOf()	检索字符串。
italics()	使用斜体显示字符串。
lastIndexOf()	从后向前搜索字符串。
link()	将字符串显示为链接。
localeCompare()	用本地特定的顺序来比较两个字符串。
match()	找到一个或多个正在表达式的匹配。
replace()	替换与正则表达式匹配的子串。
search()	检索与正则表达式相匹配的值。
slice()	提取字符串的片断，并在新的字符串中返回被提取的部分。
small()	使用小字号来显示字符串。
split()	把字符串分割为字符串数组。
strike()	使用删除线来显示字符串。
sub()	把字符串显示为下标。
substr()	从起始索引号提取字符串中指定数目的字符。
substring()	提取字符串中两个指定的索引号之间的字符。
sup()	把字符串显示为上标。
toLocaleLowerCase()	把字符串转换为小写。
toLocaleUpperCase()	把字符串转换为大写。
toLowerCase()	把字符串转换为小写。
toUpperCase()	把字符串转换为大写。
toSource()	代表对象的源代码。
toString()	返回字符串。
valueOf()	返回某个字符串对象的原始值。

length 属性

String 对象具有属性 length，它是字符串中的字符个数：var oStringObject = new String("hello world");

alert(oStringObject.length); //输出 "11"

这个例子输出的是 "11"，即 "hello world" 中的字符个数。注意，即使字符串包含双字节的字符（与 ASCII 字符相对，ASCII 字符只占用一个字节），每个字符也只算一个字符。

charAt() 和 charCodeAt()

方法 String 对象还拥有大量的方法。首先，两个方法 charAt() 和 charCodeAt() 访问的是字

符串中的单个字符。这两个方法都有一个参数，即要操作的字符的位置。charAt() 方法返回的是包含指定位置处的字符的字符串：var oStringObject = new String("hello world");
alert(oStringObject.charAt(1)); //输出 "e"
在字符串 "hello world" 中，位置 1 处的字符是 "e"。在“ECMAScript 原始类型”这一节中我们讲过，第一个字符的位置是 0，第二个字符的位置是 1，依此类推。因此，调用 charAt(1) 返回的是 "e"。如果想得到的不是字符，而是字符代码，那么可以调用 charCodeAt() 方法：
var oStringObject = new String("hello world");
alert(oStringObject.charCodeAt(1)); //输出 "101"
这个例子输出 "101"，即小写字母 "e" 的字符代码。

concat() 方法

接下来是 concat() 方法，用于把一个或多个字符串连接到 String 对象的原始值上。该方法返回的是 String 原始值，保持原始的 String 对象不变：var oStringObject = new String("hello ");
var sResult = oStringObject.concat("world");
alert(sResult); //输出 "hello world"
alert(oStringObject); //输出 "hello "
在上面这段代码中，调用 concat() 方法返回的是 "hello world"，而 String 对象存放的仍然是 "hello "。出于这种原因，较常见的是用加号（+）连接字符串，因为这种形式从逻辑上表明了真正的行为：var oStringObject = new String("hello ");
var sResult = oStringObject + "world";
alert(sResult); //输出 "hello world"
alert(oStringObject); //输出 "hello "

indexOf() 和 lastIndexOf() 方法

迄今为止，已讨论过连接字符串的方法，访问字符串中的单个字符的方法。不过如果无法确定在某个字符串中是否确实存在一个字符，应该调用什么方法呢？这时，可调用 indexOf() 和 lastIndexOf() 方法。indexOf() 和 lastIndexOf() 方法返回的都是指定的子串在另一个字符串中的位置，如果没有找不到子串，则返回 -1。这两个方法的不同之处在于，indexOf() 方法是从字符串的开头（位置 0）开始检索字符串，而 lastIndexOf() 方法则是从字符串的结尾开始检索子串。例如：var oStringObject = new String("hello world!");
alert(oStringObject.indexOf("o")); 输出 "4"
alert(oStringObject.lastIndexOf("o")); 输出 "7"
在这里，第一个 "o" 字符串出现在位置 4，即 "hello" 中的 "o"；最后一个 "o" 出现在位置 7，即 "world" 中的 "o"。如果该字符串中只有一个 "o" 字符串，那么 indexOf() 和 lastIndexOf() 方法返回的位置相同。

localeCompare() 方法

下一个方法是 localeCompare()，对字符串进行排序。该方法有一个参数 - 要进行比较的字符串，返回的是下列三个值之一：如果 String 对象按照字母顺序排在参数中的字符串之前，返回负数。 如果 String 对象等于参数中的字符串，返回 0 如果 String 对象按照字母顺序排在参数中的字符串之后，返回正数。 注释：如果返回负数，那么最常见的是 -1，不过真

正返回的是由实现决定的。如果返回正数，那么同样的，最常见的是 1，不过真正返回的是由实现决定的。示例如下：

```
var oStringObject = new String("yellow");
alert(oStringObject.localeCompare("brick"));		//输出 "1"
alert(oStringObject.localeCompare("yellow"));		//输出 "0"
alert(oStringObject.localeCompare("zoo"));		//输出 "-1"
```

在这段代码中，字符串 "yellow" 与 3 个值进行了对比，即 "brick"、"yellow" 和 "zoo"。由于按照字母顺序排列，"yellow" 位于 "brick" 之后，所以 localeCompare() 返回 1；"yellow" 等于 "yellow"，所以 localeCompare() 返回 0；"zoo" 位于 "yellow" 之后，localeCompare() 返回 -1。再强调一次，由于返回的值是由实现决定的，所以最好以下面的方式调用 localeCompare() 方法：

```
var oStringObject1 = new String("yellow");
var oStringObject2 = new String("brick");

var iResult = sTestString.localeCompare("brick");

if(iResult < 0) {
  alert(oStringObject1 + " comes before " + oStringObject2);
} else if (iResult > 0) {
  alert(oStringObject1 + " comes after " + oStringObject2);
} else {
  alert("The two strings are equal");
}
```

采用这种结构，可以确保这段代码在所有实现中都能正确运行。localeCompare() 方法的独特之处在于，实现所处的区域（locale，兼指国家/地区和语言）确切说明了这种方法运行的方式。在美国，英语是 ECMAScript 实现的标准语言，localeCompare() 是区分大小写的，大写字母在字母顺序上排在小写字母之后。不过，在其他区域，情况可能并非如此。

slice() 和 substring()

ECMAScript 提供了两种方法从子串创建字符串值，即 slice() 和 substring()。这两种方法返回的都是要处理的字符串的子串，都接受一个或两个参数。第一个参数是要获取的子串的起始位置，第二个参数（如果使用的话）是要获取子串终止前的位置（也就是说，获取终止位置处的字符不包括在返回的值内）。如果省略第二个参数，终止位就默认为字符串的长度。与 concat() 方法一样，slice() 和 substring() 方法都不改变 String 对象自身的值。它们只返回原始的 String 值，保持 String 对象不变。

```
var oStringObject = new String("hello world");
alert(oStringObject.slice("3"));		//输出 "lo world"
alert(oStringObject.substring("3"));		//输出 "lo world"
alert(oStringObject.slice("3, 7"));		//输出 "lo w"
alert(oStringObject.substring("3, 7"));	//输出 "lo w"
```

在这个例子中，slice() 和 substring() 的用法相同，返回值也一样。当只有参数 3 时，两个方法返回的都是 "lo world"，因为 "hello" 中的第二个 "l" 位于位置 3 上。当有两个参数 "3" 和 "7" 时，两个方法返回的值都是 "lo w"（"world" 中的字母 "o" 位于位置 7 上，所以它不包括在结果中）。为什么有两个功能完全相同的方法呢？事实上，这两个方法并不完全相同，不过只在参数为负数时，它们处理参数的方式才稍有不同。对于负数参数，slice() 方

法会用字符串的长度加上参数，substring() 方法则将其作为 0 处理（也就是说将忽略它）。例如：var oStringObject = new String("hello world");

```
alert(oStringObject.slice("-3"));          //输出 "rld"
alert(oStringObject.substring("-3"));      //输出 "hello world"
alert(oStringObject.slice("3, -4"));       //输出 "lo w"
alert(oStringObject.substring("3, -4")); //输出 "hel"
```

这样即可看出 slice() 和 substring() 方法的主要不同。当只有参数 -3 时，slice() 返回 "rld"，substring() 则返回 "hello world"。这是因为对于字符串 "hello world"，slice("-3") 将被转换成 slice("8")，而 substring("-3") 将被转换成 substring("0")。同样，使用参数 3 和 -4 时，差别也很明显。slice() 将被转换成 slice(3, 7)，与前面的例子相同，返回 "lo w"。而 substring() 方法则将两个参数解释为 substring(3, 0)，实际上即 substring(0, 3)，因为 substring() 总把较小的数字作为起始位，较大的数字作为终止位。因此，substring("3, -4") 返回的是 "hel"。这里的最后一行代码用来说明如何使用这些方法。

toLowerCase() toLocaleLowerCase() toUpperCase() 和 toLocaleUpperCase()

最后一套要讨论的方法涉及大小写转换。有 4 种方法用于执行大小写转换，即 toLowerCase() toLocaleLowerCase() toUpperCase() toLocaleUpperCase() 从名字上可以看出它们的用途，前两种方法用于把字符串转换成全小写的，后两种方法用于把字符串转换成全大写的。toLowerCase() 和 toUpperCase() 方法是原始的，是以 java.lang.String 中相同方法为原型实现的。toLocaleLowerCase() 和 toLocaleUpperCase() 方法是基于特定的区域实现的（与 localeCompare() 方法相同）。在许多区域中，区域特定的方法都与通用的方法完全相同。不过，有几种语言对 Unicode 大小写转换应用了特定的规则（例如土耳其语），因此必须使用区域特定的方法才能进行正确的转换。var oStringObject = new String("Hello World");

```
alert(oStringObject.toLocaleUpperCase());  //输出 "HELLO WORLD"
alert(oStringObject.toUpperCase());          //输出 "HELLO WORLD"
alert(oStringObject.toLocaleLowerCase());  //输出 "hello world"
alert(oStringObject.toLowerCase());          //输出 "hello world"
```

这段代码中，toUpperCase() 和 toLocaleUpperCase() 输出的都是 "HELLO WORLD"，toLowerCase() 和 toLocaleLowerCase() 输出的都是 "hello world"。一般来说，如果不知道在以哪种编码运行一种语言，则使用区域特定的方法比较安全。提示：记住，String 对象的所有属性和方法都可应用于 String 原始值上，因为它们是伪对象。instanceof 运算符在使用 typeof 运算符时采用引用类型存储值会出现一个问题，无论引用的是什么类型的对象，它都返回 "object"。ECMAScript 引入了另一个 Java 运算符 instanceof 来解决这个问题。instanceof 运算符与 typeof 运算符相似，用于识别正在处理的对象的类型。与 typeof 方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。例如：var oStringObject = new String("hello world");

```
alert(oStringObject instanceof String); //输出 "true"
```

这段代码问的是“变量 oStringObject 是否为 String 对象的实例？”oStringObject 的确是 String 对象的实例，因此结果是 "true"。尽管不像 typeof 方法那样灵活，但是在 typeof 方法返回 "object" 的情况下，instanceof 方法还是很有用的。

Math 对象

Math 对象用于执行数学任务。使用 Math 的属性和方法的语法：

```
var pi_value=Math.PI;

var sqrt_value=Math.sqrt(15);
```

注释：Math 对象并不像 Date 和 String 那样是对象的类，因此没有构造函数 Math()，像 Math.sin() 这样的函数只是函数，不是某个对象的方法。您无需创建它，通过把 Math 作为对象使用就可以调用其所有属性和方法。

Math 对象属性

属性	描述
E	返回算术常量 e，即自然对数的底数（约等于 2.718）。
LN2	返回 2 的自然对数（约等于 0.693）。
LN10	返回 10 的自然对数（约等于 2.302）。
LOG2E	返回以 2 为底的 e 的对数（约等于 1.414）。
LOG10E	返回以 10 为底的 e 的对数（约等于 0.434）。
PI	返回圆周率（约等于 3.14159）。
SQRT1_2	返回返回 2 的平方根的倒数（约等于 0.707）。
SQRT2	返回 2 的平方根（约等于 1.414）。

Math 对象方法

方法	描述
abs(x)	返回数的绝对值。
acos(x)	返回数的反余弦值。
asin(x)	返回数的反正弦值。

atan(x)	以介于 -PI/2 与 PI/2 弧度之间的数值来返回 x 的反正切值。
atan2(y,x)	返回从 x 轴到点 (x,y) 的角度（介于 -PI/2 与 PI/2 弧度之间）。
ceil(x)	对数进行上舍入。
cos(x)	返回数的余弦。
exp(x)	返回 e 的指数。
floor(x)	对数进行下舍入。
log(x)	返回数的自然对数（底为 e）。
max(x,y)	返回 x 和 y 中的最高值。
min(x,y)	返回 x 和 y 中的最低值。
pow(x,y)	返回 x 的 y 次幂。
random()	返回 0 ~ 1 之间的随机数。
round(x)	把数四舍五入为最接近的整数。
sin(x)	返回数的正弦。
sqrt(x)	返回数的平方根。
tan(x)	返回角的正切。
toSource()	返回该对象的源代码。
valueOf()	返回 Math 对象的原始值。

全局对象

全局属性和函数可用于所有内建的 JavaScript 对象。

顶层函数（全局函数）

函数	描述

decodeURI()	解码某个编码的 URI。
decodeURIComponent()	解码一个编码的 URI 组件。
encodeURI()	把字符串编码为 URI。
encodeURIComponent()	把字符串编码为 URI 组件。
escape()	对字符串进行编码。
eval()	计算 JavaScript 字符串，并把它作为脚本代码来执行。
getClass()	返回一个 JavaObject 的 JavaClass。
isFinite()	检查某个值是否为有穷大的数。
isNaN()	检查某个值是否是数字。
Number()	把对象的值转换为数字。
parseFloat()	解析一个字符串并返回一个浮点数。
parseInt()	解析一个字符串并返回一个整数。
String()	把对象的值转换为字符串。
unescape()	对由 escape() 编码的字符串进行解码。

顶层属性（全局属性）

方法	描述
Infinity	代表正的无穷大的数值。
java	代表 java.* 包层级的一个 JavaPackage。
NaN	指示某个值是不是数字值。
Packages	根 JavaPackage 对象。
undefined	指示未定义的值。

3 ECMAScript 运算符

一元运算符

一元运算符只有一个参数，即要操作的对象或值。它们是 ECMAScript 中最简单的运算符。

delete

delete 运算符删除对以前定义的对象属性或方法的引用。例如：

```
var o = new Object;
o.name = "David";
alert(o.name); //输出 "David"
delete o.name;
alert(o.name); //输出 "undefined"
```

在这个例子中，删除了 name 属性，意味着强制解除对它的引用，将其设置为 undefined（即创建的未初始化的变量的值）。

delete 运算符不能删除开发者未定义的属性和方法。例如，下面的代码将引发错误：

delete o.toString;即使 toString 是有效的方法名，这行代码也会引发错误，因为 toString() 方法是原始的 ECMAScript 方法，不是开发者定义的。

void

void 运算符对任何值返回 undefined。该运算符通常用于避免输出不应该输出的值，例如，从 HTML 的 <a> 元素调用 JavaScript 函数时。要正确做到这一点，函数不能返回有效值，否则浏览器将清空页面，只显示函数的结果。例如：

Click me如果把这行代码放入 HTML 页面，点击其中的链接，即可看到屏幕上显示 "[object]"。TIY

这是因为 window.open() 方法返回了新打开的窗口的引用。然后该对象将被转换成要显示的字符串。

要避免这种效果，可以用 void 运算符调用 window.open() 函数：

Click me这使 window.open() 调用返回 undefined，它不是有效值，不会显示在浏览器窗口中。

提示：请记住，没有返回值的函数真正返回的都是 undefined。

前增量/前减量运算符

直接从 C（和 Java）借用的两个运算符是前增量运算符和前减量运算符。

所谓前增量运算符，就是数值上加 1，形式是在变量前放两个加号（++）：

```
var iNum = 10;
++iNum;
```

第二行代码把 iNum 增加到了 11，它实质上等价于：

```
var iNum = 10;
iNum = iNum + 1；
```

同样，前减量运算符是从数值上减 1，形式是在变量前放两个减号（--）：

```
var iNum = 10;
--iNum;
```

在这个例子中，第二行代码把 iNum 的值减到 9。

在使用前缀式运算符时，注意增量和减量运算符都发生在计算表达式之前。考虑下面的例子：

```
var iNum = 10;
--iNum;
alert(iNum);    //输出 "9"
alert(--iNum); //输出 "8"
alert(iNum);    //输出 "8"
```

第二行代码对 iNum 进行减量运算，第三行代码显示的结果是（"9"）。第四行代码又对 iNum 进行减量运算，不过这次前减量运算和输出操作出现在同一个语句中，显示的结果是 "8"。为了证明已实现了所有的减量操作，第五行代码又输出一次"8"。

在算术表达式中，前增量和前减量运算符的优先级是相同的，因此要按照从左到右的顺序计算之。例如：

```
var iNum1 = 2;
var iNum2 = 20;
var iNum3 = --iNum1 + ++iNum2;  //等于 "22"
var iNum4 = iNum1 + iNum2;        //等于 "22"
```

在前面的代码中，iNum3 等于 22，因为表达式要计算的是 1 + 21。变量 iNum4 也等于 22，也是 1 + 21。

后增量/后减量运算符

还有两个直接从 C（和 Java）借用的运算符，即后增量运算符和后减量运算符。

后增量运算符也是给数值上加 1，形式是在变量后放两个加号（++）：

```
var iNum = 10;
iNum++;
```

不出所料，后减量运算符也是从数值上减 1，形式为在变量后加两个减号（--）：

```
var iNum = 10;
iNum--;
```

第二行代码把 iNum 的 值减到 9。

与前缀式运算符不同的是，后缀式运算符是在计算过包含它们的表达式后才进行增量或减量运算的。考虑以下的例子：

```
var iNum = 10;
iNum--;
alert(iNum);    //输出 "9"
alert(iNum--); //输出 "9"
alert(iNum);    //输出 "8"
```

与前缀式运算符的例子相似，第二行代码对 iNum 进行减量运算，第三行代码显示结果（"9"）。第四行代码继续显示 iNum 的值，不过这次是在同一语句中应用减量运算符。由于减量运算发生在计算过表达式之后，所以这条语句显示的数是 "9"。执行了第五行代码后，alert 函数显示的是 "8"，因为在执行第四行代码之后和执行第五行代码之前，执行了后减量运算。

在算术表达式中，后增量和后减量运算符的优先级是相同的，因此要按照从左到右的顺序计算之。例如：

```
var iNum1 = 2;
var iNum2 = 20;
var iNum3 = iNum1-- + iNum2++;  //等于 "22"
var iNum4 = iNum1 + iNum2;        //等于 "22"
```

在前面的代码中，iNum3 等于 22，因为表达式要计算的是 2 + 20。变量 iNum4 也等于 22，不过它计算的是 1 + 21，因为增量和减量运算都在给 iNum3 赋值后才发生。

一元加法和一元减法

大多数人都熟悉一元加法和一元减法，它们在 ECMAScript 中的用法与您高中数学中学到的用法相同。

一元加法本质上对数字无任何影响：

```
var iNum = 20;
iNum = +iNum;
alert(iNum);    //输出 "20"
```

这段代码对数字 20 应用了一元加法，返回的还是 20。

尽管一元加法对数字无作用，但对字符串却有有趣的效果，会把字符串转换成数字。

```
var sNum = "20";
alert(typeof sNum);//输出 "string"
```

```
var iNum = +sNum;
alert(typeof iNum); //输出 "number"
```

这段代码把字符串 "20" 转换成真正的数字。当一元加法运算符对字符串进行操作时，它计算字符串的方式与 parseInt() 相似，主要的不同是只有对以 "0x" 开头的字符串（表示十六进制数字），一元运算符才能把它转换成十进制的值。因此，用一元加法转换 "010"，得到的总是 10，而 "0xB" 将被转换成 11。

另一方面，一元减法就是对数值求负（例如把 20 转换成 -20）：

```
var iNum = 20;
iNum = -iNum;
alert(iNum);    //输出 "-20"
```

与一元加法运算符相似，一元减法运算符也会把字符串转换成近似的数字，此外还会对该值求负。例如：

```
var sNum = "20";
alert(typeof sNum);//输出 "string"
var iNum = -sNum;
alert(iNum);          //输出 "-20"
alert(typeof iNum); //输出 "number"
```

在上面的代码中，一元减法运算符将把字符串 "-20" 转换成 -20（一元减法运算符对十六进制和十进制的处理方式与一元加法运算符相似，只是它还会对该值求负）。

位运算符

位运算符是在数字底层（即表示数字的 32 个数位）进行操作的。

重温整数

ECMAScript 整数有两种类型，即有符号整数（允许用正数和负数）和无符号整数（只允许用正数）。在 ECMAScript 中，所有整数字面量默认都是有符号整数，这意味着什么呢？

有符号整数使用 31 位表示整数的数值，用第 32 位表示整数的符号，0 表示正数，1 表示负数。数值范围从 -2147483648 到 2147483647。

可以以两种不同的方式存储二进制形式的有符号整数，一种用于存储正数，一种用于存储负数。正数是以真二进制形式存储的，前 31 位中的每一位都表示 2 的幂，从第 1 位（位 0）开始，表示 20，第 2 位（位 1）表示 21。没用到的位用 0 填充，即忽略不计。例如，下图展示的是数 18 的表示法。

18 的二进制版本只用了前 5 位，它们是这个数字的有效位。把数字转换成二进制字符串，就能看到有效位：

```
var iNum = 18;
alert(iNum.toString(2));  //输出 "10010"
```

这段代码只输出 "10010"，而不是 18 的 32 位表示。其他的数位并不重要，因为仅使用前 5 位即可确定这个十进制数值。如下图所示：

负数也存储为二进制代码，不过采用的形式是二进制补码。计算数字二进制补码的步骤有三步：

确定该数字的非负版本的二进制表示（例如，要计算 -18 的二进制补码，首先要确定 18 的二进制表示）

求得二进制反码，即要把 0 替换为 1，把 1 替换为 0

在二进制反码上加 1

要确定 -18 的二进制表示，首先必须得到 18 的二进制表示，如下所示：

0000 0000 0000 0000 0000 0000 0001 0010 接下来，计算二进制反码，如下所示：

1111 1111 1111 1111 1111 1111 1110 1101 最后，在二进制反码上加 1，如下所示：

```
1111 1111 1111 1111 1111 1111 1110 1101
                                         1
---------------------------------------
1111 1111 1111 1111 1111 1111 1110 1110
```

因此，-18 的二进制表示即 1111 1111 1111 1111 1111 1111 1110 1110。记住，在处理有符号整数时，开发者不能访问 31 位。

有趣的是，把负整数转换成二进制字符串后，ECMAScript 并不以二进制补码的形式显示，而是用数字绝对值的标准二进制代码前面加负号的形式输出。例如：

```
var iNum = -18;
alert(iNum.toString(2));  //输出 "-10010"
```

这段代码输出的是 "-10010"，而非二进制补码，这是为避免访问位 31。为了简便，ECMAScript 用一种简单的方式处理整数，使得开发者不必关心它们的用法。

另一方面，无符号整数把最后一位作为另一个数位处理。在这种模式中，第 32 位不表示数字的符号，而是值 231。由于这个额外的位，无符号整数的数值范围为 0 到 4294967295。对于小于 2147483647 的整数来说，无符号整数看来与有符号整数一样，而大于 2147483647 的整数则要使用位 31（在有符号整数中，这一位总是 0）。

把无符号整数转换成字符串后，只返回它们的有效位。

注意：所有整数字面量都默认存储为有符号整数。只有 ECMAScript 的位运算符才能创建无符号整数。

位运算 NOT

位运算 NOT 由否定号（~）表示，它是 ECMAScript 中为数不多的与二进制算术有关的运算符之一。

位运算 NOT 是三步的处理过程：

把运算数转换成 32 位数字
把二进制数转换成它的二进制反码
把二进制数转换成浮点数
例如：

```
var iNum1 = 25;           //25 等于 00000000000000000000000000011001
var iNum2 = ~iNum1;    //转换为 11111111111111111111111111100110
alert(iNum2);          //输出 "-26"
```

位运算 NOT 实质上是对数字求负，然后减 1，因此 25 变 -26。用下面的方法也可以得到同样的方法：

```
var iNum1 = 25;
var iNum2 = -iNum1 -1;
alert(iNum2); //输出 -26
```

位运算 AND

位运算 AND 由和号（&）表示，直接对数字的二进制形式进行运算。它把每个数字中的数位对齐，然后用下面的规则对同一位置上的两个数位进行 AND 运算：

第一个数字中的数位	第二个数字中的数位	结果
1	1	1
1	0	0
0	1	0
0	0	0

例如，要对数字 25 和 3 进行 AND 运算，代码如下所示：

```
var iResult = 25 & 3;
alert(iResult); //输出 "1"
```

25 和 3 进行 AND 运算的结果是 1。为什么？分析如下：

```
 25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
---------------------------------------------
AND = 0000 0000 0000 0000 0000 0000 0000 0001
```

可以看出，在 25 和 3 中，只有一个数位（位 0）存放的都是 1，因此，其他数位生成的都是 0，所以结果为 1。

位运算 OR

位运算 OR 由符号（|）表示，也是直接对数字的二进制形式进行运算。在计算每位时，OR 运算符采用下列规则：

第一个数字中的数位 第二个数字中的数位 结果
1 1 1
1 0 1
0 1 1
0 0 0

仍然使用 AND 运算符所用的例子，对 25 和 3 进行 OR 运算，代码如下：

```
var iResult = 25 | 3;
alert(iResult); //输出 "27"
```

25 和 3 进行 OR 运算的结果是 27：

```
25 = 0000 0000 0000 0000 0000 0000 0001 1001
  3 = 0000 0000 0000 0000 0000 0000 0000 0011
---------------------------------------------
OR = 0000 0000 0000 0000 0000 0000 0001 1011
```

可以看出，在两个数字中，共有 4 个数位存放的是 1，这些数位被传递给结果。二进制代码 11011 等于 27。

位运算 XOR

位运算 XOR 由符号（^）表示，当然，也是直接对二进制形式进行运算。XOR 不同于 OR，当只有一个数位存放的是 1 时，它才返回 1。真值表如下：

第一个数字中的数位 第二个数字中的数位 结果
1 1 0
1 0 1
0 1 1
0 0 0

对 25 和 3 进行 XOR 运算，代码如下：

```
var iResult = 25 ^ 3;
alert(iResult); //输出 "26"
```

25 和 3 进行 XOR 运算的结果是 26：

```
  25 = 0000 0000 0000 0000 0000 0000 0001 1001
    3 = 0000 0000 0000 0000 0000 0000 0000 0011
---------------------------------------------
XOR = 0000 0000 0000 0000 0000 0000 0001 1010
```

可以看出，在两个数字中，共有 4 个数位存放的是 1，这些数位被传递给结果。二进制代码 11010 等于 26。

左移运算

左移运算由两个小于号表示（<<）。它把数字中的所有数位向左移动指定的数量。例如，把数字 2（等于二进制中的 10）左移 5 位，结果为 64（等于二进制中的 1000000）：

```
var iOld = 2;           //等于二进制 10
var iNew = iOld << 5;      //等于二进制 1000000 十进制 64
```

注意：在左移数位时，数字右边多出 5 个空位。左移运算用 0 填充这些空位，使结果成为完整的 32 位数字。

注意：左移运算保留数字的符号位。例如，如果把 -2 左移 5 位，得到的是 -64，而不是 64。“符号仍然存储在第 32 位中吗？”是的，不过这在 ECMAScript 后台进行，开发者不能直接访问第 32 个数位。即使输出二进制字符串形式的负数，显示的也是负号形式（例如，-2 将显示 -10。）

有符号右移运算

有符号右移运算符由两个大于号表示（>>）。它把 32 位数字中的所有数位整体右移，同时保留该数的符号（正号或负号）。有符号右移运算符恰好与左移运算相反。例如，把 64 右移 5 位，将变为 2：

```
var iOld = 64;          //等于二进制 1000000
var iNew = iOld >> 5;      //等于二进制 10 十进制 2
```

同样，移动数位后会造成空位。这次，空位位于数字的左侧，但位于符号位之后。ECMAScript 用符号位的值填充这些空位，创建完整的数字，如下图所示：

无符号右移运算

无符号右移运算符由三个大于号（>>>）表示，它将无符号 32 位数的所有数位整体右移。对于正数，无符号右移运算的结果与有符号右移运算一样。

用有符号右移运算中的例子，把 64 右移 5 位，将变为 2：

```
var iOld = 64;          //等于二进制 1000000
var iNew = iOld >>> 5;     //等于二进制 10 十进制 2
```

对于负数，情况就不同了。

无符号右移运算用 0 填充所有空位。对于正数，这与有符号右移运算的操作一样，而负数则被作为正数来处理。

由于无符号右移运算的结果是一个 32 位的正数，所以负数的无符号右移运算得到的总是一个非常大的数字。例如，如果把 -64 右移 5 位，将得到 134217726。如果得到这种结果的呢？

要实现这一点，需要把这个数字转换成无符号的等价形式（尽管该数字本身还是有符号的），可以通过以下代码获得这种形式：

var iUnsigned64 = -64 >>> 0;然后，用 Number 类型的 toString() 获取它的真正的位表示，采用的基为 2：

alert(iUnsigned64.toString(2));这将生成 11111111111111111111111111000000，即有符号整数 -64 的二进制补码表示，不过它等于无符号整数 4294967232。

出于这种原因，使用无符号右移运算符要小心。

Boolean 运算符

Boolean 运算符非常重要，它使得程序语言得以正常运行。

Boolean 运算符有三种：NOT、AND 和 OR。
ToBoolean 操作
在学习各种逻辑运算符之前，让我们先了解一下 ECMAScript-262 v5 规范中描述的 ToBoolean 操作。

抽象操作 ToBoolean 将其参数按照下表中的规则转换为逻辑值：

参数类型	结果
Undefined	false
Null	false
Boolean	结果等于输入的参数（不转换）
Number	如果参数为 +0, -0 或 NaN，则结果为 false；否则为 true。
String	如果参数为空字符串，则结果为 false；否则为 true。
Object	true

逻辑 NOT 运算符
在 ECMAScript 中，逻辑 NOT 运算符与 C 和 Java 中的逻辑 NOT 运算符相同，都由感叹号（!）表示。

与逻辑 OR 和逻辑 AND 运算符不同的是，逻辑 NOT 运算符返回的一定是 Boolean 值。

逻辑 NOT 运算符的行为如下：

如果运算数是对象，返回 false
如果运算数是数字 0，返回 true
如果运算数是 0 以外的任何数字，返回 false
如果运算数是 null，返回 true
如果运算数是 NaN，返回 true
如果运算数是 undefined，发生错误
通常，该运算符用于控制循环：

```
var bFound = false;
var i = 0;

while (!bFound) {
   if (aValue[i] == vSearchValues) {
      bFound = true;
   } else {
      i++;
   }
}
```

在这个例子中，Boolean 变量（bFound）用于记录检索是否成功。找到问题中的数据项时，bFound 将被设置为 true，!bFound 将等于 false，意味着运行将跳出 while 循环。

判断 ECMAScript 变量的 Boolean 值时，也可以使用逻辑 NOT 运算符。这样做需要在一行代码中使用两个 NOT 运算符。无论运算数是什么类型，第一个 NOT 运算符返回 Boolean 值。第二个 NOT 将对该 Boolean 值求负，从而给出变量真正的 Boolean 值。

```
var bFalse = false;
var sRed = "red";
var iZero = 0;
var iThreeFourFive = 345;
var oObject = new Object;

document.write("bFalse 的逻辑值是 " + (!!bFalse));
document.write("sRed 的逻辑值是 " + (!!sRed));
document.write("iZero 的逻辑值是 " + (!!iZero));
document.write("iThreeFourFive 的逻辑值是 " + (!!iThreeFourFive));
document.write("oObject 的逻辑值是 " + (!!oObject));
```

结果：

```
bFalse 的逻辑值是 false
sRed 的逻辑值是 true
iZero 的逻辑值是 false
iThreeFourFive 的逻辑值是 true
oObject 的逻辑值是 true
```

逻辑 AND 运算符

在 ECMAScript 中，逻辑 AND 运算符用双和号（&&）表示：

例如：

```
var bTrue = true;
var bFalse = false;
```

```
var bResult = bTrue && bFalse;
```

下面的真值表描述了逻辑 AND 运算符的行为：

运算数 1	运算数 2	结果
true	true	true
true	false	false
false	true	false
false	false	false

逻辑 AND 运算的运算数可以是任何类型的，不止是 Boolean 值。

如果某个运算数不是原始的 Boolean 型值，逻辑 AND 运算并不一定返回 Boolean 值：

如果一个运算数是对象，另一个是 Boolean 值，返回该对象。
如果两个运算数都是对象，返回第二个对象。
如果某个运算数是 null，返回 null。
如果某个运算数是 NaN，返回 NaN。
如果某个运算数是 undefined，发生错误。
与 Java 中的逻辑 AND 运算相似，ECMAScript 中的逻辑 AND 运算也是简便运算，即如果第一个运算数决定了结果，就不再计算第二个运算数。对于逻辑 AND 运算来说，如果第一个运算数是 false，那么无论第二个运算数的值是什么，结果都不可能等于 true。

考虑下面的例子：

```
var bTrue = true;
var bResult = (bTrue && bUnknown);    //发生错误
alert(bResult);                //这一行不会执行
```

这段代码在进行逻辑 AND 运算时将引发错误，因为变量 bUnknown 是未定义的。变量 bTrue 的值为 true，因为逻辑 AND 运算将继续计算变量 bUnknown。这样做就会引发错误，因为 bUnknown 的值是 undefined，不能用于逻辑 AND 运算。

如果修改这个例子，把第一个数设为 false，那么就不会发生错误：

```
var bFalse = false;
var bResult = (bFalse && bUnknown);
alert(bResult);                //输出 "false"
```

在这段代码中，脚本将输出逻辑 AND 运算返回的值，即字符串 "false"。即使变量 bUnknown 的值为 undefined，它也不会被计算，因为第一个运算数的值是 false。

提示：在使用逻辑 AND 运算符时，必须记住它的这种简便计算特性。

逻辑 OR 运算符

ECMAScript 中的逻辑 OR 运算符与 Java 中的相同，都由双竖线（||）表示：

```
var bTrue = true;
var bFalse = false;
var bResult = bTrue || bFalse;
```

下面的真值表描述了逻辑 OR 运算符的行为：

运算数 1	运算数 2	结果
true	true	true
true	false	true
false	true	true
false	false	false

与逻辑 AND 运算符相似，如果某个运算数不是 Boolean 值，逻辑 OR 运算并不一定返回 Boolean 值：

如果一个运算数是对象，并且该对象左边的运算数值均为 false，则返回该对象。
如果两个运算数都是对象，返回第一个对象。
如果最后一个运算数是 null，并且其他运算数值均为 false，则返回 null。
如果最后一个运算数是 NaN，并且其他运算数值均为 false，则返回 NaN。
如果某个运算数是 undefined，发生错误。
与逻辑 AND 运算符一样，逻辑 OR 运算也是简便运算。对于逻辑 OR 运算符来说，如果第一个运算数值为 true，就不再计算第二个运算数。

例如：

```
var bTrue = true;
var bResult = (bTrue || bUnknown);
alert(bResult);             //输出 "true"
```

与前面的例子相同，变量 bUnknown 是未定义的。不过，由于变量 bTrue 的值为 true，bUnknown 不会被计算，因此输出的是 "true"。

如果把 bTrue 改为 false，将发生错误：

```
var bFalse = false;
var bResult = (bFalse || bUnknown);    //发生错误
alert(bResult);             //不会执行这一行
```

乘性运算符

ECMAScript 的乘性运算符与 Java、C、Perl 等于语言中的同类运算符的运算方式相似。

需要注意的是，乘性运算符还具有一些自动转换功能。
乘法运算符

乘法运算符由星号（*）表示，用于两数相乘。

ECMAScript 中的乘法语法与 C 语言中的相同：

var iResult = 12 * 34 不过，在处理特殊值时，ECMAScript 中的乘法还有一些特殊行为：

如果结果太大或太小，那么生成的结果是 Infinity 或 -Infinity。
如果某个运算数是 NaN，结果为 NaN。
Infinity 乘以 0，结果为 NaN。
Infinity 乘以 0 以外的任何数字，结果为 Infinity 或 -Infinity。
Infinity 乘以 Infinity，结果为 Infinity。
注释：如果运算数是数字，那么执行常规的乘法运算，即两个正数或两个负数为正数，两个运算数符号不同，结果为负数。
除法运算符
除法运算符由斜杠（/）表示，用第二个运算数除第一个运算数：

var iResult = 88 /11;
与乘法运算符相似，在处理特殊值时，除法运算符也有一些特殊行为：

如果结果太大或太小，那么生成的结果是 Infinity 或 -Infinity。
如果某个运算数是 NaN，结果为 NaN。
Infinity 被 Infinity 除，结果为 NaN。
Infinity 被任何数字除，结果为 Infinity。
0 除一个任何非无穷大的数字，结果为 NaN。
Infinity 被 0 以外的任何数字除，结果为 Infinity 或 -Infinity。
注释：如果运算数是数字，那么执行常规的除法运算，即两个正数或两个负数为正数，两个运算数符号不同，结果为负数。
取模运算符
除法（余数）运算符由百分号（%）表示，使用方法如下：

var iResult = 26%5; //等于 1 与其他乘性运算符相似，对于特殊值，取模运算符也有特殊的行为：

如果被除数是 Infinity，或除数是 0，结果为 NaN。
Infinity 被 Infinity 除，结果为 NaN。
如果除数是无穷大的数，结果为被除数。
如果被除数为 0，结果为 0。
注释：如果运算数是数字，那么执行常规的算术除法运算，返回除法运算得到的余数。

加性运算符

在多数程序设计语言中，加性运算符（即加号或减号）通常是最简单的数学运算符。

在 ECMAScript 中，加性运算符有大量的特殊行为。

加法运算符

法运算符由加号（+）表示：

var iResult = 1 + 2 与乘性运算符一样，在处理特殊值时，ECMAScript 中的加法也有一些特殊行为：

某个运算数是 NaN，那么结果为 NaN。

-Infinity 加 -Infinity，结果为 -Infinity。

Infinity 加 -Infinity，结果为 NaN。

+0 加 +0，结果为 +0。

-0 加 +0，结果为 +0。

-0 加 -0，结果为 -0。

不过，如果某个运算数是字符串，那么采用下列规则：

如果两个运算数都是字符串，把第二个字符串连接到第一个上。

如果只有一个运算数是字符串，把另一个运算数转换成字符串，结果是两个字符串连接成的字符串。

例如：

```
var result = 5 + 5;    //两个数字
alert(result);         //输出 "10"
var result2 = 5 + "5";    //一个数字和一个字符串
alert(result);         //输出 "55"
```

这段代码说明了加法运算符的两种模式之间的差别。正常情况下，5+5 等于 10（原始数值），如上述代码中前两行所示。不过，如果把一个运算数改为字符串 "5"，那么结果将变为 "55"（原始的字符串值），因为另一个运算数也会被转换为字符串。

注意：为了避免 JavaScript 中的一种常见错误，在使用加法运算符时，一定要仔细检查运算数的数据类型。

减法运算符

减法运算符（-），也是一个常用的运算符：

var iResult = 2 - 1;与加法运算符一样，在处理特殊值时，减法运算符也有一些特殊行为：

某个运算数是 NaN，那么结果为 NaN。

Infinity 减 Infinity，结果为 NaN。

-Infinity 减 -Infinity，结果为 NaN。

Infinity 减 -Infinity，结果为 Infinity。

-Infinity 减 Infinity，结果为 -Infinity。

+0 减 +0，结果为 +0。

-0 减 -0，结果为 -0。

+0 减 -0，结果为 +0。
某个运算符不是数字，那么结果为 NaN。
注释：如果运算数都是数字，那么执行常规的减法运算，并返回结果。

关系运算符

关系运算符执行的是比较运算。每个关系运算符都返回一个布尔值。
常规比较方式
关系运算符小于、大于、小于等于和大于等于执行的是两个数的比较运算，比较方式与算术比较运算相同。

每个关系运算符都返回一个布尔值：

```
var bResult1 = 2 > 1//true
var bResult2 = 2 < 1//false
```

不过，对两个字符串应用关系运算符，它们的行为则不同。许多人认为小于表示“在字母顺序上靠前”，大于表示“在字母顺序上靠后”，但事实并非如此。对于字符串，第一个字符串中每个字符的代码都与会第二个字符串中对应位置的字符的代码进行数值比较。完成这种比较操作后，返回一个 Boolean 值。问题在于大写字母的代码都小于小写字母的代码，这意味这着可能会遇到下列情况：

```
var bResult = "Blue" < "alpha";
alert(bResult); //输出 true
```

在上面的例子中，字符串 "Blue" 小于 "alpha"，因为字母 B 的字符代码是 66，字母 a 的字符代码是 97。要强制性得到按照真正的字母顺序比较的结果，必须把两个数转换成相同的大小写形式（全大写或全小写的），然后再进行比较：

```
var bResult = "Blue".toLowerCase() < "alpha".toLowerCase();
alert(bResult); //输出 false
```

把两个运算数都转换成小写，确保了正确识别出 "alpha" 在字母顺序上位于 "Blue" 之前。
比较数字和字符串
另一种棘手的状况发生在比较两个字符串形式的数字时，比如：

```
var bResult = "25" < "3";
alert(bResult); //输出 "true"
```

上面这段代码比较的是字符串 "25" 和 "3"。两个运算数都是字符串，所以比较的是它们的字符代码（"2" 的字符代码是 50，"3" 的字符代码是 51）。

不过，如果把某个运算数该为数字，那么结果就有趣了：

```
var bResult = "25" < 3;
```

```
alert(bResult); //输出 "false"
```

这里，字符串 "25" 将被转换成数字 25，然后与数字 3 进行比较，结果不出所料。

无论何时比较一个数字和一个字符串，ECMAScript 都会把字符串转换成数字，然后按照数字顺序比较它们。

不过，如果字符串不能转换成数字又该如何呢？考虑下面的例子：

```
var bResult = "a" < 3;
alert(bResult);
```

你能预料到这段代码输出什么吗？字母 "a" 不能转换成有意义的数字。不过，如果对它调用 parseInt() 方法，返回的是 NaN。根据规则，任何包含 NaN 的关系运算符都要返回 false，因此这段代码也输出 false：

```
var bResult = "a" >= 3;
alert(bResult);
```

通常，如果小于运算的两个值返回 false，那么大于等于运算必须返回 true，不过如果某个数字是 NaN，情况则非如此。

等性运算符

判断两个变量是否相等是程序设计中非常重要的运算。在处理原始值时，这种运算相当简单，但涉及对象，任务就稍有点复杂。

ECMAScript 提供了两套等性运算符：等号和非等号用于处理原始值，全等号和非全等号用于处理对象。

等号和非等号

在 ECMAScript 中，等号由双等号（==）表示，当且仅当两个运算数相等时，它返回 true。非等号由感叹号加等号（!=）表示，当且仅当两个运算数不相等时，它返回 true。为确定两个运算数是否相等，这两个运算符都会进行类型转换。

执行类型转换的规则如下：

如果一个运算数是 Boolean 值，在检查相等性之前，把它转换成数字值。false 转换成 0，true 为 1。
如果一个运算数是字符串，另一个是数字，在检查相等性之前，要尝试把字符串转换成数字。
如果一个运算数是对象，另一个是字符串，在检查相等性之前，要尝试把对象转换成字符串。
如果一个运算数是对象，另一个是数字，在检查相等性之前，要尝试把对象转换成数字。
在比较时，该运算符还遵守下列规则：

值 null 和 undefined 相等。
在检查相等性时，不能把 null 和 undefined 转换成其他值。

如果某个运算数是 NaN，等号将返回 false，非等号将返回 true。
如果两个运算数都是对象，那么比较的是它们的引用值。如果两个运算数指向同一对象，那么等号返回 true，否则两个运算数不等。
重要提示：即使两个数都是 NaN，等号仍然返回 false，因为根据规则，NaN 不等于 NaN。

下表列出了一些特殊情况，以及它们的结果：

表达式	值
null == undefined	true
"NaN" == NaN	false
5 == NaN	false
NaN == NaN	false
NaN != NaN	true
false == 0	true
true == 1	true
true == 2	false
undefined == 0	false
null == 0	false
"5" == 5	true

全等号和非全等号
等号和非等号的同类运算符是全等号和非全等号。这两个运算符所做的与等号和非等号相同，只是它们在检查相等性前，不执行类型转换。

全等号由三个等号表示（===），只有在无需类型转换运算数就相等的情况下，才返回 true。

例如：

```
var sNum = "66";
var iNum = 66;
alert(sNum == iNum);    //输出 "true"
alert(sNum === iNum);  //输出 "false"
```

在这段代码中，第一个 alert 使用等号来比较字符串 "66" 和数字 66，输出 "true"。如前所述，这是因为字符串 "66" 将被转换成数字 66,，然后才与另一个数字 66 进行比较。第二个 alert 使用全等号在没有类型转换的情况下比较字符串和数字，当然，字符串不等于数字，所以输出 "false"。

非全等号由感叹号加两个等号（!==）表示，只有在无需类型转换运算数不相等的情况下，才返回 true。

例如：

```
var sNum = "66";
var iNum = 66;
```

alert(sNum != iNum);　//输出 "false"
alert(sNum !== iNum);　//输出 "true"
这里，第一个 alert 使用非等号，把字符串 "66" 转换成数字 66，使得它与第二个运算数 66 相等。因此，计算结果为 "false"，因为两个运算数是相等的。第二个 alert 使用的非全等号。该运算是在问："sNum" 与 "iNum" 不同吗？这个问题的答案是：是的（true），因为 sNum 是字符串，而 iNum 是数字，它们当然不同。

条件运算符

条件运算符
条件运算符是 ECMAScript 中功能最多的运算符，它的形式与 Java 中的相同。

variable = boolean_expression ? true_value : false_value; 该表达式主要是根据 boolean_expression 的计算结果有条件地为变量赋值。如果 Boolean_expression 为 true，就把 true_value 赋给变量；如果它是 false，就把 false_value 赋给变量。

例如：

var iMax = (iNum1 > iNum2) ? iNum1 : iNum2;在这里例子中，iMax 将被赋予数字中的最大值。表达式声明如果 iNum1 大于 iNum2，则把 iNum1 赋予 iMax。但如果表达式为 false（即 iNum2 大于或等于 iNum1），则把 iNum2 赋予 iMax。

赋值运算符

简单的赋值运算符由等号（=）实现，只是把等号右边的值赋予等号左边的变量。
例如：
var iNum = 10;复合赋值运算是由乘性运算符、加性运算符或位移运算符加等号（=）实现的。这些赋值运算符是下列这些常见情况的缩写形式：

var iNum = 10;
iNum = iNum + 10；
可以用一个复合赋值运算符改写第二行代码：

var iNum = 10;
iNum += 10；
每种主要的算术运算以及其他几个运算都有复合赋值运算符：

乘法/赋值（*=）
除法/赋值（/=）
取模/赋值（%=）
加法/赋值（+=）

减法/赋值（-=）
左移/赋值（<<=）
有符号右移/赋值（>>=）
无符号右移/赋值（>>>=）

逗号运算符

逗号运算符
用逗号运算符可以在一条语句中执行多个运算。

例如：
var iNum1 = 1, iNum = 2, iNum3 = 3;逗号运算符常用变量声明中。

4 ECMAScript 语句

if 语句

if 语句是 ECMAScript 中最常用的语句之一。
ECMAScript 语句
ECMA - 262 描述了 ECMAScript 的几种语句（statement）。

语句主要定义了 ECMAScript 的大部分语句，通常采用一个或多个关键字，完成给定的任务。

语句可以非常简单，例如通知函数退出，也可以非常复杂，如声明一组要反复执行的命令。

在《ECMAScript 语句》这一章，我们介绍了所有标准的 ECMAScript 语句。
if 语句
if 语句是 ECMAScript 中最常用的语句之一，事实上在许多计算机语言中都是如此。

if 语句的语法：

if (condition) statement1 else statement2 其中 condition 可以是任何表达式，计算的结果甚至不必是真正的 boolean 值，ECMAScript 会把它转换成 boolean 值。

如果条件计算结果为 true，则执行 statement1；如果条件计算结果为 false，则执行 statement2。

每个语句都可以是单行代码，也可以是代码块。

例如：

```
if (i > 30)
   {alert("大于 30");}
else
   {alert("小于等于 30");}
```

提示：使用代码块被认为是一种最佳的编程实践，即使要执行的代码只有一行。这样做可以使每个条件要执行什么一目了然。

还可以串联多个 if 语句。就像这样：

if (condition1) statement1 else if (condition2) statement2 else statement3 例如：

```
if (i > 30) {
   alert("大于 30");
} else if (i < 0) {
   alert("小于 0");
} else {
   alert("在 0 到 30 之间");
}
```

迭代语句

迭代语句又叫循环语句，声明一组要反复执行的命令，直到满足某些条件为止。

循环通常用于迭代数组的值（因此而得名），或者执行重复的算术任务。

本节为您介绍 ECMAScript 提供的四种迭代语句。

do-while 语句

do-while 语句是后测试循环，即退出条件在执行循环内部的代码之后计算。这意味着在计算表达式之前，至少会执行循环主体一次。

它的语法如下：

do {statement} while (expression);例子：

```
var i = 0;
do {i += 2;} while (i < 10);
```

while 语句

while 语句是前测试循环。这意味着退出条件是在执行循环内部的代码之前计算的。因此，循环主体可能根本不被执行。

它的语法如下：

while (expression) statement 例子：

```
var i = 0;
while (i < 10) {
   i += 2;
}
```

for 语句
for 语句是前测试循环，而且在进入循环之前，能够初始化变量，并定义循环后要执行的代码。

它的语法如下：

for (initialization; expression; post-loop-expression) statement 注意：post-loop-expression 之后不能写分号，否则无法运行。

例子：

```
iCount = 6;
for (var i = 0; i < iCount; i++) {
   alert(i);
}
```

这段代码定义了初始值为 0 的变量 i。只有当条件表达式（i < iCount）的值为 true 时，才进入 for 循环，这样循环主体可能不被执行。如果执行了循环主体，那么将执行循环后表达式，并迭代变量 i。
for-in 语句
for 语句是严格的迭代语句，用于枚举对象的属性。

它的语法如下：

for (property in expression) statement 例子：

```
for (sProp in window) {
   alert(sProp);
}
```

这里，for-in 语句用于显示 window 对象的所有属性。

前面讨论过的 PropertyIsEnumerable() 是 ECMAScript 中专门用于说明属性是否可以用 for-in 语句访问的方法。

标签语句

有标签的语句
可以用下列语句给语句加标签，以便以后调用：

label : statement 例如：

start : i = 5;在这个例子中，标签 start 可以被之后的 break 或 continue 语句引用。

提示：在下面的章节，我们将为您介绍 break 和 continue 语句。

break 和 continue 语句

break 和 continue 语句对循环中的代码执行提供了更严格的控制。
break 和 continue 语句的不同之处
break 语句可以立即退出循环，阻止再次反复执行任何代码。

而 continue 语句只是退出当前循环，根据控制表达式还允许继续进行下一次循环。

例如：

```
var iNum = 0;

for (var i=1; i<10; i++) {
   if (i % 5 == 0) {
      break;
   }
   iNum++;
}
alert(iNum);    //输出 "4"
```

在以上代码中，for 循环从 1 到 10 迭代变量 i。在循环主体中，if 语句将（使用取模运算符）检查 i 的值是否能被 5 整除。如果能被 5 整除，将执行 break 语句。alert 显示 "4"，即退出循环前执行循环的次数。

如果用 continue 语句代替这个例子中的 break 语句，结果将不同：

```
var iNum = 0;

for (var i=1; i<10; i++) {
   if (i % 5 == 0) {
```

```
    continue;
  }
  iNum++;
}
alert(iNum);   //输出 "8"
```

这里，alert 将显示 "8"，即执行循环的次数。可能执行的循环总数为 9，不过当 i 的值为 5 时，将执行 continue 语句，会使循环跳过表达式 iNum++，返回循环开头。

与有标签的语句一起使用

break 语句和 continue 语句都可以与有标签的语句联合使用，返回代码中的特定位置。

通常，当循环内部还有循环时，会这样做，例如：

```
var iNum = 0;

outermost:
for (var i=0; i<10; i++) {
  for (var j=0; j<10; j++) {
    if (i == 5 && j == 5) {
    break outermost;
  }
  iNum++;
  }
}

alert(iNum);   //输出 "55"
```

在上面的例子中，标签 outermost 表示的是第一个 for 语句。正常情况下，每个 for 语句执行 10 次代码块，这意味着 iNum++ 正常情况下将被执行 100 次，在执行完成时，iNum 应该等于 100。这里的 break 语句有一个参数，即停止循环后要跳转到的语句的标签。这样 break 语句不止能跳出内部 for 语句（即使用变量 j 的语句），还能跳出外部 for 语句（即使用变量 i 的语句）。因此，iNum 最后的值是 55，因为当 i 和 j 的值都等于 5 时，循环将终止。

可以以相同的方式使用 continue 语句：

```
var iNum = 0;

outermost:
for (var i=0; i<10; i++) {
  for (var j=0; j<10; j++) {
    if (i == 5 && j == 5) {
    continue outermost;
  }
```

```
    iNum++;
    }
}

alert(iNum);    //输出 "95"
```

在上例中，continue 语句会迫使循环继续，不止是内部循环，外部循环也如此。当 j 等于 5 时出现这种情况，意味着内部循环将减少 5 次迭代，致使 iNum 的值为 95。

提示：可以看出，与 break 和 continue 联合使用的有标签语句非常强大，不过过度使用它们会给调试代码带来麻烦。要确保使用的标签具有说明性，同时不要嵌套太多层循环。

提示：想了解什么是有标签语句，请阅读 ECMAScript 标签语句 这一节。

with 语句

有标签的语句
with 语句用于设置代码在特定对象中的作用域。

它的语法：

with (expression) statement 例如：

```
var sMessage = "hello";
with(sMessage) {
    alert(toUpperCase()); //输出 "HELLO"
}
```

在这个例子中，with 语句用于字符串，所以在调用 toUpperCase() 方法时，解释程序将检查该方法是否是本地函数。如果不是，它将检查伪对象 sMessage，看它是否为该对象的方法。然后，alert 输出 "HELLO"，因为解释程序找到了字符串 "hello" 的 toUpperCase() 方法。

提示：with 语句是运行缓慢的代码块，尤其是在已设置了属性值时。大多数情况下，如果可能，最好避免使用它。

switch 语句

switch 语句
switch 语句是 if 语句的兄弟语句。

开发者可以用 switch 语句为表达式提供一系列的情况（case）。

switch 语句的语法：

```
switch (expression)
  case value: statement;
    break;
  case value: statement;
    break;
  case value: statement;
    break;
  case value: statement;
    break;
...
  case value: statement;
    break;
  default: statement;
```

每个情况（case）都是表示“如果 expression 等于 value，就执行 statement”。

关键字 break 会使代码跳出 switch 语句。如果没有关键字 break，代码执行就会继续进入下一个 case。

关键字 default 说明了表达式的结果不等于任何一种情况时的操作（事实上，它相对于 else 从句）。

switch 语句主要是为避免让开发者编写下面的代码：

```
if (i == 20)
  alert("20");
else if (i == 30)
  alert("30");
else if (i == 40)
  alert("40");
else
  alert("other");
```

等价的 switch 语句是这样的：

```
switch (i) {
  case 20: alert("20");
    break;
  case 30: alert("30");
    break;
  case 40: alert("40");
    break;
  default: alert("other");
}
```

ECMAScript 和 Java 中的 switch 语句

ECMAScript 和 Java 中的 switch 语句有两点不同。在 ECMAScript 中，switch 语句可以用于字符串，而且能用不是常量的值说明情况：

```
var BLUE = "blue", RED = "red", GREEN  = "green";

switch (sColor) {
  case BLUE: alert("Blue");
    break;
  case RED: alert("Red");
    break;
  case GREEN: alert("Green");
    break;
  default: alert("Other");
}
```

这里，switch 语句用于字符串 sColor，声明 case 使用的是变量 BLUE、RED 和 GREEN，这在 ECMAScript 中是完全有效的。

5 ECMAScript 函数

函数概述

什么是函数？
函数是一组可以随时随地运行的语句。

函数是 ECMAScript 的核心。

函数是由这样的方式进行声明的：关键字 function、函数名、一组参数，以及置于括号中的待执行代码。

函数的基本语法是这样的：

```
function functionName(arg0, arg1, ... argN) {
  statements
}
```

例如：

```
function sayHi(sName, sMessage) {
  alert("Hello " + sName + sMessage);
}
```

如何调用函数？
函数可以通过其名字加上括号中的参数进行调用，如果有多个参数。

如果您想调用上例中的那个函数，可以使用如下的代码：

sayHi("David", " Nice to meet you!")调用上面的函数 sayHi() 会生成一个警告窗口。您可以亲自试一试这个例子。
函数如何返回值？
函数 sayHi() 未返回值，不过不必专门声明它（像在 Java 中使用 void 那样）。

即使函数确实有值，也不必明确地声明它。该函数只需要使用 return 运算符后跟要返回的值即可。

```
function sum(iNum1, iNum2) {
  return iNum1 + iNum2;
}
```

下面的代码把 sum 函数返回的值赋予一个变量：

```
var iResult = sum(1,1);
alert(iResult);  //输出 "2"
```

另一个重要概念是，与在 Java 中一样，函数在执行过 return 语句后立即停止代码。因此，return 语句后的代码都不会被执行。

例如，在下面的代码中，alert 窗口就不会显示出来：

```
function sum(iNum1, iNum2) {
   return iNum1 + iNum2;
   alert(iNum1 + iNum2);
}
```

一个函数中可以有多个 return 语句，如下所示：

```
function diff(iNum1, iNum2) {
   if (iNum1 > iNum2) {
      return iNum1 - iNum2;
   } else {
      return iNum2 - iNum1;
   }
}
```

上面的函数用于返回两个数的差。要实现这一点，必须用较大的数减去较小的数，因此用 if 语句决定执行哪个 return 语句。

如果函数无返回值，那么可以调用没有参数的 return 运算符，随时退出函数。

例如：

```
function sayHi(sMessage) {
   if (sMessage == "bye") {
      return;
   }

   alert(sMessage);
}
```

这段代码中，如果 sMessage 等于 "bye"，就永远不显示警告框。

注释：如果函数无明确的返回值，或调用了没有参数的 return 语句，那么它真正返回的值是 undefined。

arguments 对象

arguments 对象
在函数代码中，使用特殊对象 arguments，开发者无需明确指出参数名，就能访问它们。

例如，在函数 sayHi() 中，第一个参数是 message。用 arguments[0] 也可以访问这个值，即第一个参数的值（第一个参数位于位置 0，第二个参数位于位置 1，依此类推）。

因此，无需明确命名参数，就可以重写函数：

```
function sayHi() {
   if (arguments[0] == "bye") {
      return;
   }

   alert(arguments[0]);
}
```

检测参数个数
还可以用 arguments 对象检测函数的参数个数，引用属性 arguments.length 即可。

下面的代码将输出每次调用函数使用的参数个数：

```
function howManyArgs() {
   alert(arguments.length);
}

howManyArgs("string", 45);
howManyArgs();
howManyArgs(12);
```

上面这段代码将依次显示 "2"、"0" 和 "1"。

注释：与其他程序设计语言不同，ECMAScript 不会验证传递给函数的参数个数是否等于函数定义的参数个数。开发者定义的函数都可以接受任意个数的参数（根据 Netscape 的文档，最多可接受 25 个），而不会引发任何错误。任何遗漏的参数都会以 undefined 传递给函数，多余的函数将忽略。
模拟函数重载
用 arguments 对象判断传递给函数的参数个数，即可模拟函数重载：

```
function doAdd() {
   if(arguments.length == 1) {
      alert(arguments[0] + 5);
   } else if(arguments.length == 2) {
```

```
        alert(arguments[0] + arguments[1]);
    }
}

doAdd(10);      //输出 "15"
doAdd(40, 20);      //输出 "60"
```

当只有一个参数时，doAdd() 函数给参数加 5。如果有两个参数，则会把两个参数相加，返回它们的和。所以，doAdd(10) 输出的是 "15"，而 doAdd(40, 20) 输出的是 "60"。

虽然不如重载那么好，不过已足以避开 ECMAScript 的这种限制。

Function 对象（类）

ECMAScript 的函数实际上是功能完整的对象。
Function 对象（类）
ECMAScript 最令人感兴趣的可能莫过于函数实际上是功能完整的对象。

Function 类可以表示开发者定义的任何函数。

用 Function 类直接创建函数的语法如下：

var function_name = new function(arg1, arg2, ..., argN, function_body)在上面的形式中，每个 arg 都是一个参数，最后一个参数是函数主体（要执行的代码）。这些参数必须是字符串。

记得下面这个函数吗？

```
function sayHi(sName, sMessage) {
    alert("Hello " + sName + sMessage);
}
```

还可以这样定义它：

```
var sayHi
=
new Function("sName", "sMessage", "alert(\"Hello \" + sName + sMessage);");
```

虽然由于字符串的关系，这种形式写起来有些困难，但有助于理解函数只不过是一种引用类型，它们的行为与用 Function 类明确创建的函数行为是相同的。

请看下面这个例子：

```
function doAdd(iNum) {
    alert(iNum + 20);
```

```
}

function doAdd(iNum) {
    alert(iNum + 10);
}

doAdd(10);     //输出 "20"
```

如你所知，第二个函数重载了第一个函数，使 doAdd(10) 输出了 "20"，而不是 "30"。

如果以下面的形式重写该代码块，这个概念就清楚了：

```
var doAdd = new Function("iNum", "alert(iNum + 20)");
var doAdd = new Function("iNum", "alert(iNum + 10)");
doAdd(10);
```

请观察这段代码，很显然，doAdd 的值被改成了指向不同对象的指针。函数名只是指向函数对象的引用值，行为就像其他对象一样。甚至可以使两个变量指向同一个函数：

```
var doAdd = new Function("iNum", "alert(iNum + 10)");
var alsodoAdd = doAdd;
doAdd(10);     //输出 "20"
alsodoAdd(10);     //输出 "20"
```

在这里，变量 doAdd 被定义为函数，然后 alsodoAdd 被声明为指向同一个函数的指针。用这两个变量都可以执行该函数的代码，并输出相同的结果 - "20"。因此，如果函数名只是指向函数的变量，那么可以把函数作为参数传递给另一个函数吗？回答是肯定的！

```
function callAnotherFunc(fnFunction, vArgument) {
    fnFunction(vArgument);
}

var doAdd = new Function("iNum", "alert(iNum + 10)");

callAnotherFunc(doAdd, 10); //输出 "20"
```

在上面的例子中，callAnotherFunc() 有两个参数 - 要调用的函数和传递给该函数的参数。这段代码把 doAdd() 传递给 callAnotherFunc() 函数，参数是 10，输出 "20"。

注意：尽管可以使用 Function 构造函数创建函数，但最好不要使用它，因为用它定义函数比用传统方式要慢得多。不过，所有函数都应看作 Function 类的实例。

Function 对象的 length 属性

如前所述，函数属于引用类型，所以它们也有属性和方法。

ECMAScript 定义的属性 length 声明了函数期望的参数个数。例如：

```
function doAdd(iNum) {
```

```
   alert(iNum + 10);
}

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

alert(doAdd.length);      //输出 "1"
alert(sayHi.length); //输出 "0"
```

函数 doAdd() 定义了一个参数，因此它的 length 是 1；sayHi() 没有定义参数，所以 length 是 0。

记住，无论定义了几个参数，ECMAScript 可以接受任意多个参数（最多 25 个），这一点在《函数概述》这一章中讲解过。属性 length 只是为查看默认情况下预期的参数个数提供了一种简便方式。

Function 对象的方法

Function 对象也有与所有对象共享的 valueOf() 方法和 toString() 方法。这两个方法返回的都是函数的源代码，在调试时尤其有用。例如：

```
function doAdd(iNum) {
   alert(iNum + 10);
}

document.write(doAdd.toString());
```

上面这段代码输出了 doAdd() 函数的文本。亲自试一试！

闭包（closure）

ECMAScript 最易让人误解的一点是，它支持闭包（closure）。

闭包，指的是词法表示包括不被计算的变量的函数，也就是说，函数可以使用函数之外定义的变量。

简单的闭包实例

在 ECMAScript 中使用全局变量是一个简单的闭包实例。请思考下面这段代码：

```
var sMessage = "hello world";

function sayHelloWorld() {
   alert(sMessage);
}

sayHelloWorld();
```

在上面这段代码中，脚本被载入内存后，并没有为函数 sayHelloWorld() 计算变量 sMessage 的值。该函数捕获 sMessage 的值只是为了以后的使用，也就是说，解释程序知道在调用该函数时要检查 sMessage 的值。sMessage 将在函数调用 sayHelloWorld() 时（最后一行）被赋值，显示消息 "hello world"。

复杂的闭包实例

在一个函数中定义另一个会使闭包变得更加复杂。例如：

```
var iBaseNum = 10;

function addNum(iNum1, iNum2) {
    function doAdd() {
        return iNum1 + iNum2 + iBaseNum;
    }
    return doAdd();
}
```

这里，函数 addNum() 包括函数 doAdd() （闭包）。内部函数是一个闭包，因为它将获取外部函数的参数 iNum1 和 iNum2 以及全局变量 iBaseNum 的值。 addNum() 的最后一步调用了 doAdd()，把两个参数和全局变量相加，并返回它们的和。

这里要掌握的重要概念是，doAdd() 函数根本不接受参数，它使用的值是从执行环境中获取的。

可以看到，闭包是 ECMAScript 中非常强大多用的一部分，可用于执行复杂的计算。

提示：就像使用任何高级函数一样，使用闭包要小心，因为它们可能会变得非常复杂。

6 ECMAScript 对象

面向对象技术

面向对象术语
对象
ECMA-262 把对象（object）定义为“属性的无序集合，每个属性存放一个原始值、对象或函数”。严格来说，这意味着对象是无特定顺序的值的数组。

尽管 ECMAScript 如此定义对象，但它更通用的定义是基于代码的名词（人、地点或事物）的表示。

类
每个对象都由类定义，可以把类看做对象的配方。类不仅要定义对象的接口（interface）（开发者访问的属性和方法），还要定义对象的内部工作（使属性和方法发挥作用的代码）。编译器和解释程序都根据类的说明构建对象。

实例
程序使用类创建对象时，生成的对象叫作类的实例（instance）。对类生成的对象的个数的唯一限制来自于运行代码的机器的物理内存。每个实例的行为相同，但实例处理一组独立的数据。由类创建对象实例的过程叫做实例化（instantiation）。

在前面的章节我们提到过，ECMAScript 并没有正式的类。相反，ECMA-262 把对象定义描述为对象的配方。这是 ECMAScript 逻辑上的一种折中方案，因为对象定义实际上是对象自身。即使类并不真正存在，我们也把对象定义叫做类，因为大多数开发者对此术语更熟悉，而且从功能上说，两者是等价的。
面向对象语言的要求
一种面向对象语言需要向开发者提供四种基本能力：

封装 - 把相关的信息（无论数据或方法）存储在对象中的能力
聚集 - 把一个对象存储在另一个对象内的能力
继承 - 由另一个类（或多个类）得来类的属性和方法的能力
多态 - 编写能以多种方法运行的函数或方法的能力
ECMAScript 支持这些要求，因此可被是看做面向对象的。
对象的构成
在 ECMAScript 中，对象由特性（attribute）构成，特性可以是原始值，也可以是引用值。如果特性存放的是函数，它将被看作对象的方法（method），否则该特性被看作对象的属性（property）。

对象应用

对象的创建和销毁都在 JavaScript 执行过程中发生，理解这种范式的含义对理解整个语言至关重要。

声明和实例化

对象的创建方式是用关键字 new 后面跟上实例化的类的名字：

```
var oObject = new Object();
var oStringObject = new String();
```

第一行代码创建了 Object 类的一个实例，并把它存储到变量 oObject 中。第二行代码创建了 String 类的一个实例，把它存储在变量 oStringObject 中。如果构造函数无参数，括号则不是必需的。因此可以采用下面的形式重写上面的两行代码：

```
var oObject = new Object;
var oStringObject = new String;
```

对象引用

在前面的章节中，我们介绍了引用类型的概念。在 ECMAScript 中，不能访问对象的物理表示，只能访问对象的引用。每次创建对象，存储在变量中的都是该对象的引用，而不是对象本身。

对象废除

ECMAScript 拥有无用存储单元收集程序（garbage collection routine），意味着不必专门销毁对象来释放内存。当再没有对对象的引用时，称该对象被废除（dereference）了。运行无用存储单元收集程序时，所有废除的对象都被销毁。每当函数执行完它的代码，无用存储单元收集程序都会运行，释放所有的局部变量，还有在一些其他不可预知的情况下，无用存储单元收集程序也会运行。

把对象的所有引用都设置为 null，可以强制性地废除对象。例如：

```
var oObject = new Object;
// do something with the object here
oObject = null;
```

当变量 oObject 设置为 null 后，对第一个创建的对象的引用就不存在了。这意味着下次运行无用存储单元收集程序时，该对象将被销毁。

每用完一个对象后，就将其废除，来释放内存，这是个好习惯。这样还确保不再使用已经不能访问的对象，从而防止程序设计错误的出现。此外，旧的浏览器（如 IE/MAC）没有全面的无用存储单元收集程序，所以在卸载页面时，对象可能不能被正确销毁。废除对象和它的所有特性是确保内存使用正确的最好方法。

注意：废除对象的所有引用时要当心。如果一个对象有两个或更多引用，则要正确废除该对象，必须将其所有引用都设置为 null。

早绑定和晚绑定

所谓绑定（binding），即把对象的接口与对象实例结合在一起的方法。

早绑定（early binding）是指在实例化对象之前定义它的属性和方法，这样编译器或解释程序就能够提前转换机器代码。在 Java 和 Visual Basic 这样的语言中，有了早绑定，就可以在开发环境中使用 IntelliSense（即给开发者提供对象中属性和方法列表的功能）。ECMAScript 不是强类型语言，所以不支持早绑定。

另一方面，晚绑定（late binding）指的是编译器或解释程序在运行前，不知道对象的类型。使用晚绑定，无需检查对象的类型，只需检查对象是否支持属性和方法即可。ECMAScript 中的所有变量都采用晚绑定方法。这样就允许执行大量的对象操作，而无任何惩罚。

对象类型

在 ECMAScript 中，所有对象并非同等创建的。

一般来说，可以创建并使用的对象有三种：本地对象、内置对象和宿主对象。
本地对象
ECMA-262 把本地对象（native object）定义为“独立于宿主环境的 ECMAScript 实现提供的对象”。简单来说，本地对象就是 ECMA-262 定义的类（引用类型）。它们包括：

Object
Function
Array
String
Boolean
Number
Date
RegExp
Error
EvalError
RangeError
ReferenceError
SyntaxError
TypeError
URIError

内置对象

ECMA-262 把内置对象（built-in object）定义为“由 ECMAScript 实现提供的、独立于宿主环境的所有对象，在 ECMAScript 程序开始执行时出现”。这意味着开发者不必明确实例化内置对象，它已被实例化了。ECMA-262 只定义了两个内置对象，即 Global 和 Math （它们也是本地对象，根据定义，每个内置对象都是本地对象）。

宿主对象

所有非本地对象都是宿主对象（host object），即由 ECMAScript 实现的宿主环境提供的对象。

所有 BOM 和 DOM 对象都是宿主对象。

对象作用域

作用域指的是变量的适用范围。
公用、私有和受保护作用域
概念
在传统的面向对象程序设计中，主要关注于公用和私有作用域。公用作用域中的对象属性可以从对象外部访问，即开发者创建对象的实例后，就可使用它的公用属性。而私有作用域中的属性只能在对象内部访问，即对于外部世界来说，这些属性并不存在。这意味着如果类定义了私有属性和方法，则它的子类也不能访问这些属性和方法。

受保护作用域也是用于定义私有的属性和方法，只是这些属性和方法还能被其子类访问。

ECMAScript 只有公用作用域
对 ECMAScript 讨论上面这些作用域几乎毫无意义，因为 ECMAScript 中只存在一种作用域 - 公用作用域。ECMAScript 中的所有对象的所有属性和方法都是公用的。因此，定义自己的类和对象时，必须格外小心。记住，所有属性和方法默认都是公用的！

建议性的解决方法
许多开发者都在网上提出了有效的属性作用域模式，解决了 ECMAScript 的这种问题。

由于缺少私有作用域，开发者确定了一个规约，说明哪些属性和方法应该被看做私有的。这种规约规定在属性前后加下划线：

obj._color_ = "blue";这段代码中，属性 color 是私有的。注意，下划线并不改变属性是公用属性的事实，它只是告诉其他开发者，应该把该属性看作私有的。

有些开发者还喜欢用单下划线说明私有成员，例如：obj._color。
静态作用域
静态作用域定义的属性和方法任何时候都能从同一位置访问。在 Java 中，类可具有属性和方法，无需实例化该类的对象，即可访问这些属性和方法，例如 java.net.URLEncoder 类，它的函数 encode() 就是静态方法。

ECMAScript 没有静态作用域
严格来说，ECMAScript 并没有静态作用域。不过，它可以给构造函数提供属性和方法。还记得吗，构造函数只是函数。函数是对象，对象可以有属性和方法。例如：

```
function sayHello() {
```

```
    alert("hello");
}

sayHello.alternate = function() {
    alert("hi");
}

sayHello();              //输出 "hello"
sayHello.alternate();     //输出 "hi"
```

TIY

这里，方法 alternate() 实际上是函数 sayHello 的方法。可以像调用常规函数一样调用 sayHello() 输出 "hello"，也可以调用 sayHello.alternate() 输出 "hi"。即使如此，alternate() 也是 sayHello() 公用作用域中的方法，而不是静态方法。

关键字 this

this 的功能

在 ECMAScript 中，要掌握的最重要的概念之一是关键字 this 的用法，它用在对象的方法中。关键字 this 总是指向调用该方法的对象，例如：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function() {
    alert(this.color);
};

oCar.showColor();         //输出 "red"
```

TIY

在上面的代码中，关键字 this 用在对象的 showColor() 方法中。在此环境中，this 等于 oCar。下面的代码与上面的代码的功能相同：

```
var oCar = new Object;
oCar.color = "red";
oCar.showColor = function() {
    alert(oCar.color);
};

oCar.showColor();         //输出 "red"
```

TIY

使用 this 的原因

为什么使用 this 呢？因为在实例化对象时，总是不能确定开发者会使用什么样的变量名。使用 this，即可在任何多个地方重用同一个函数。请思考下面的例子：

```
function showColor() {
  alert(this.color);
};

var oCar1 = new Object;
oCar1.color = "red";
oCar1.showColor = showColor;

var oCar2 = new Object;
oCar2.color = "blue";
oCar2.showColor = showColor;

oCar1.showColor();		//输出 "red"
oCar2.showColor();		//输出 "blue"
```

TIY

在上面的代码中，首先用 this 定义函数 showColor()，然后创建两个对象（oCar1 和 oCar2），一个对象的 color 属性被设置为 "red"，另一个对象的 color 属性被设置为 "blue"。两个对象都被赋予了属性 showColor，指向原始的 showColor () 函数（注意这里不存在命名问题，因为一个是全局函数，而另一个是对象的属性）。调用每个对象的 showColor()，oCar1 输出是 "red"，而 oCar2 的输出是 "blue"。这是因为调用 oCar1.showColor() 时，函数中的 this 关键字等于 oCar1。调用 oCar2.showColor() 时，函数中的 this 关键字等于 oCar2。

注意，引用对象的属性时，必须使用 this 关键字。例如，如果采用下面的代码，showColor() 方法不能运行：

```
function showColor() {
  alert(color);
};
```

如果不用对象或 this 关键字引用变量，ECMAScript 就会把它看作局部变量或全局变量。然后该函数将查找名为 color 的局部或全局变量，但是不会找到。结果如何呢？该函数将在警告中显示 "null"。

定义类或对象

使用预定义对象只是面向对象语言的能力的一部分，它真正强大之处在于能够创建自己专用的类和对象。

ECMAScript 拥有很多创建对象或类的方法。

工厂方式

原始的方式

因为对象的属性可以在对象创建后动态定义，所有许多开发者都在 JavaScript 最初引入时编写类似下面的代码：

```
var oCar = new Object;
oCar.color = "blue";
oCar.doors = 4;
oCar.mpg = 25;
oCar.showColor = function() {
  alert(this.color);
};
```

TIY

在上面的代码中，创建对象 car。然后给它设置几个属性：它的颜色是蓝色，有四个门，每加仑油可以跑 25 英里。最后一个属性实际上是指向函数的指针，意味着该属性是个方法。执行这段代码后，就可以使用对象 car。

不过这里有一个问题，就是可能需要创建多个 car 的实例。

解决方案：工厂方式

要解决该问题，开发者创造了能创建并返回特定类型的对象的工厂函数（factory function）。

例如，函数 createCar() 可用于封装前面列出的创建 car 对象的操作：

```
function createCar() {
  var oTempCar = new Object;
  oTempCar.color = "blue";
  oTempCar.doors = 4;
  oTempCar.mpg = 25;
  oTempCar.showColor = function() {
    alert(this.color);
  };
  return oTempCar;
}

var oCar1 = createCar();
var oCar2 = createCar();
```

TIY

在这里，第一个例子中的所有代码都包含在 createCar() 函数中。此外，还有一行额外的代码，返回 car 对象（oTempCar）作为函数值。调用此函数，将创建新对象，并赋予它所有必要的属性，复制出一个我们在前面说明过的 car 对象。因此，通过这种方法，我们可以很容易地创建 car 对象的两个版本（oCar1 和 oCar2），它们的属性完全一样。

为函数传递参数

我们还可以修改 createCar() 函数，给它传递各个属性的默认值，而不是简单地赋予属性默认值：

```
function createCar(sColor,iDoors,iMpg) {
  var oTempCar = new Object;
  oTempCar.color = sColor;
  oTempCar.doors = iDoors;
  oTempCar.mpg = iMpg;
  oTempCar.showColor = function() {
    alert(this.color);
  };
  return oTempCar;
}

var oCar1 = createCar("red",4,23);
var oCar2 = createCar("blue",3,25);

oCar1.showColor();      //输出 "red"
oCar2.showColor();      //输出 "blue"
```

TIY

给 createCar() 函数加上参数，即可为要创建的 car 对象的 color、doors 和 mpg 属性赋值。这使两个对象具有相同的属性，却有不同的属性值。

在工厂函数外定义对象的方法

虽然 ECMAScript 越来越正式化，但创建对象的方法却被置之不理，且其规范化至今还遭人反对。一部分是语义上的原因（它看起来不像使用带有构造函数 new 运算符那么正规），一部分是功能上的原因。功能原因在于用这种方式必须创建对象的方法。前面的例子中，每次调用函数 createCar()，都要创建新函数 showColor()，意味着每个对象都有自己的 showColor() 版本。而事实上，每个对象都共享同一个函数。

有些开发者在工厂函数外定义对象的方法，然后通过属性指向该方法，从而避免这个问题：

```
function showColor() {
  alert(this.color);
}

function createCar(sColor,iDoors,iMpg) {
  var oTempCar = new Object;
  oTempCar.color = sColor;
  oTempCar.doors = iDoors;
```

```
    oTempCar.mpg = iMpg;
    oTempCar.showColor = showColor;
    return oTempCar;
}

var oCar1 = createCar("red",4,23);
var oCar2 = createCar("blue",3,25);

oCar1.showColor();        //输出 "red"
oCar2.showColor();        //输出 "blue"
```

TIY

在上面这段重写的代码中，在函数 createCar() 之前定义了函数 showColor()。在 createCar() 内部，赋予对象一个指向已经存在的 showColor() 函数的指针。从功能上讲，这样解决了重复创建函数对象的问题；但是从语义上讲，该函数不太像是对象的方法。

所有这些问题都引发了开发者定义的构造函数的出现。

构造函数方式

创建构造函数就像创建工厂函数一样容易。第一步选择类名，即构造函数的名字。根据惯例，这个名字的首字母大写，以使它与首字母通常是小写的变量名分开。除了这点不同，构造函数看起来很像工厂函数。请考虑下面的例子：

```
function Car(sColor,iDoors,iMpg) {
    this.color = sColor;
    this.doors = iDoors;
    this.mpg = iMpg;
    this.showColor = function() {
        alert(this.color);
    };
}

var oCar1 = new Car("red",4,23);
var oCar2 = new Car("blue",3,25);
```

TIY

下面为您解释上面的代码与工厂方式的差别。首先在构造函数内没有创建对象，而是使用 this 关键字。使用 new 运算符构造函数时，在执行第一行代码前先创建一个对象，只有用 this 才能访问该对象。然后可以直接赋予 this 属性，默认情况下是构造函数的返回值（不必明确使用 return 运算符）。

现在，用 new 运算符和类名 Car 创建对象，就更像 ECMAScript 中一般对象的创建方式了。

你也许会问，这种方式在管理函数方面是否存在于前一种方式相同的问题呢？是的。

就像工厂函数，构造函数会重复生成函数，为每个对象都创建独立的函数版本。不过，与工厂函数相似，也可以用外部函数重写构造函数，同样地，这么做语义上无任何意义。这正是下面要讲的原型方式的优势所在。

原型方式

该方式利用了对象的 prototype 属性，可以把它看成创建新对象所依赖的原型。

这里，首先用空构造函数来设置类名。然后所有的属性和方法都被直接赋予 prototype 属性。我们重写了前面的例子，代码如下：

```
function Car() {
}

Car.prototype.color = "blue";
Car.prototype.doors = 4;
Car.prototype.mpg = 25;
Car.prototype.showColor = function() {
  alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();
```

TIY

在这段代码中，首先定义构造函数（Car），其中无任何代码。接下来的几行代码，通过给 Car 的 prototype 属性添加属性去定义 Car 对象的属性。调用 new Car() 时，原型的所有属性都被立即赋予要创建的对象，意味着所有 Car 实例存放的都是指向 showColor() 函数的指针。从语义上讲，所有属性看起来都属于一个对象，因此解决了前面两种方式存在的问题。

此外，使用这种方式，还能用 instanceof 运算符检查给定变量指向的对象的类型。因此，下面的代码将输出 TRUE：

```
alert(oCar1 instanceof Car);	//输出 "true"
```

原型方式的问题

原型方式看起来是个不错的解决方案。遗憾的是，它并不尽如人意。

首先，这个构造函数没有参数。使用原型方式，不能通过给构造函数传递参数来初始化属性的值，因为 Car1 和 Car2 的 color 属性都等于 "blue"，doors 属性都等于 4，mpg 属性都等于 25。这意味着必须在对象创建后才能改变属性的默认值，这点很令人讨厌，但还没完。真正的问题出现在属性指向的是对象，而不是函数时。函数共享不会造成问题，但对象却很少被多个实例共享。请思考下面的例子：

```
function Car() {
}
```

```
Car.prototype.color = "blue";
Car.prototype.doors = 4;
Car.prototype.mpg = 25;
Car.prototype.drivers = new Array("Mike","John");
Car.prototype.showColor = function() {
   alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();

oCar1.drivers.push("Bill");

alert(oCar1.drivers);      //输出 "Mike,John,Bill"
alert(oCar2.drivers);      //输出 "Mike,John,Bill"
```

TIY

上面的代码中，属性 drivers 是指向 Array 对象的指针，该数组中包含两个名字 "Mike" 和 "John"。由于 drivers 是引用值，Car 的两个实例都指向同一个数组。这意味着给 oCar1.drivers 添加值 "Bill"，在 oCar2.drivers 中也能看到。输出这两个指针中的任何一个，结果都是显示字符串 "Mike,John,Bill"。

由于创建对象时有这么多问题，你一定会想，是否有种合理的创建对象的方法呢？答案是有，需要联合使用构造函数和原型方式。

混合的构造函数/原型方式

联合使用构造函数和原型方式，就可像用其他程序设计语言一样创建对象。这种概念非常简单，即用构造函数定义对象的所有非函数属性，用原型方式定义对象的函数属性（方法）。结果是，所有函数都只创建一次，而每个对象都具有自己的对象属性实例。

我们重写了前面的例子，代码如下：

```
function Car(sColor,iDoors,iMpg) {
   this.color = sColor;
   this.doors = iDoors;
   this.mpg = iMpg;
   this.drivers = new Array("Mike","John");
}

Car.prototype.showColor = function() {
   alert(this.color);
};
```

```
var oCar1 = new Car("red",4,23);
var oCar2 = new Car("blue",3,25);

oCar1.drivers.push("Bill");

alert(oCar1.drivers);      //输出 "Mike,John,Bill"
alert(oCar2.drivers);      //输出 "Mike,John"
```

TIY

现在就更像创建一般对象了。所有的非函数属性都在构造函数中创建，意味着又能够用构造函数的参数赋予属性默认值了。因为只创建 showColor() 函数的一个实例，所以没有内存浪费。此外，给 oCar1 的 drivers 数组添加 "Bill" 值，不会影响到 oCar2 的数组，所以输出这些数组的值时，oCar1.drivers 显示的是 "Mike,John,Bill"，而 oCar2.drivers 显示的是 "Mike,John"。因为使用了原型方式，所以仍然能利用 instanceof 运算符来判断对象的类型。

这种方式是 ECMAScript 采用的主要方式，它具有其他方式的特性，却没有他们的副作用。不过，有些开发者仍觉得这种方法不够完美。

动态原型方法

对于习惯使用其他语言的开发者来说，使用混合的构造函数/原型方式感觉不那么和谐。毕竟，定义类时，大多数面向对象语言都对属性和方法进行了视觉上的封装。请考虑下面的 Java 类：

```
class Car {
  public String color = "blue";
  public int doors = 4;
  public int mpg = 25;

  public Car(String color, int doors, int mpg) {
    this.color = color;
    this.doors = doors;
    this.mpg = mpg;
  }

  public void showColor() {
    System.out.println(color);
  }
}
```

Java 很好地打包了 Car 类的所有属性和方法，因此看见这段代码就知道它要实现什么功能，它定义了一个对象的信息。批评混合的构造函数/原型方式的人认为，在构造函数内部找属性，在其外部找方法的做法不合逻辑。因此，他们设计了动态原型方法，以提供更友好的编码风格。

动态原型方法的基本想法与混合的构造函数/原型方式相同，即在构造函数内定义非函数属

性，而函数属性则利用原型属性定义。唯一的区别是赋予对象方法的位置。下面是用动态原型方法重写的 Car 类：

```
function Car(sColor,iDoors,iMpg) {
  this.color = sColor;
  this.doors = iDoors;
  this.mpg = iMpg;
  this.drivers = new Array("Mike","John");

  if (typeof Car._initialized == "undefined") {
    Car.prototype.showColor = function() {
      alert(this.color);
    };

    Car._initialized = true;
  }
}
```

TIY

直到检查 typeof Car._initialized 是否等于 "undefined" 之前，这个构造函数都未发生变化。这行代码是动态原型方法中最重要的部分。如果这个值未定义，构造函数将用原型方式继续定义对象的方法，然后把 Car._initialized 设置为 true。如果这个值定义了（它的值为 true 时，typeof 的值为 Boolean），那么就不再创建该方法。简而言之，该方法使用标志（_initialized）来判断是否已给原型赋予了任何方法。该方法只创建并赋值一次，传统的 OOP 开发者会高兴地发现，这段代码看起来更像其他语言中的类定义了。

混合工厂方式

这种方式通常是在不能应用前一种方式时的变通方法。它的目的是创建假构造函数，只返回另一种对象的新实例。

这段代码看起来与工厂函数非常相似：

```
function Car() {
  var oTempCar = new Object;
  oTempCar.color = "blue";
  oTempCar.doors = 4;
  oTempCar.mpg = 25;
  oTempCar.showColor = function() {
    alert(this.color);
  };

  return oTempCar;
}
```

TIY

与经典方式不同，这种方式使用 new 运算符，使它看起来像真正的构造函数：

var car = new Car();由于在 Car() 构造函数内部调用了 new 运算符，所以将忽略第二个 new 运算符（位于构造函数之外），在构造函数内部创建的对象被传递回变量 car。

这种方式在对象方法的内部管理方面与经典方式有着相同的问题。强烈建议：除非万不得已，还是避免使用这种方式。

采用哪种方式

如前所述，目前使用最广泛的是混合的构造函数/原型方式。此外，动态原始方法也很流行，在功能上与构造函数/原型方式等价。可以采用这两种方式中的任何一种。不过不要单独使用经典的构造函数或原型方式，因为这样会给代码引入问题。

实例

对象令人感兴趣的一点是用它们解决问题的方式。ECMAScript 中最常见的一个问题是字符串连接的性能。与其他语言类似，ECMAScript 的字符串是不可变的，即它们的值不能改变。请考虑下面的代码：

```
var str = "hello ";
str += "world";
```

实际上，这段代码在幕后执行的步骤如下：

创建存储 "hello " 的字符串。
创建存储 "world" 的字符串。
创建存储连接结果的字符串。
把 str 的当前内容复制到结果中。
把 "world" 复制到结果中。
更新 str，使它指向结果。

每次完成字符串连接都会执行步骤 2 到 6，使得这种操作非常消耗资源。如果重复这一过程几百次，甚至几千次，就会造成性能问题。解决方法是用 Array 对象存储字符串，然后用 join() 方法（参数是空字符串）创建最后的字符串。想象用下面的代码代替前面的代码：

```
var arr = new Array();
arr[0] = "hello ";
arr[1] = "world";
var str = arr.join("");
```

这样，无论数组中引入多少字符串都不成问题，因为只在调用 join() 方法时才会发生连接操作。此时，执行的步骤如下：

创建存储结果的字符串
把每个字符串复制到结果中的合适位置

虽然这种解决方案很好，但还有更好的方法。问题是，这段代码不能确切反映出它的意图。要使它更容易理解，可以用 StringBuffer 类打包该功能：

```
function StringBuffer () {
   this._strings_ = new Array();
}

StringBuffer.prototype.append = function(str) {
   this._strings_.push(str);
};

StringBuffer.prototype.toString = function() {
   return this._strings_.join("");
};
```

这段代码首先要注意的是 strings 属性，本意是私有属性。它只有两个方法，即 append() 和 toString() 方法。append() 方法有一个参数，它把该参数附加到字符串数组中，toString() 方法调用数组的 join 方法，返回真正连接成的字符串。要用 StringBuffer 对象连接一组字符串，可以用下面的代码：

```
var buffer = new StringBuffer ();
buffer.append("hello ");
buffer.append("world");
var result = buffer.toString();
```

TIY

可用下面的代码测试 StringBuffer 对象和传统的字符串连接方法的性能：

```
var d1 = new Date();
var str = "";
for (var i=0; i < 10000; i++) {
      str += "text";
}
var d2 = new Date();

document.write("Concatenation with plus: "
 + (d2.getTime() - d1.getTime()) + " milliseconds");

var buffer = new StringBuffer();
d1 = new Date();
for (var i=0; i < 10000; i++) {
      buffer.append("text");
}
var result = buffer.toString();
d2 = new Date();

document.write("<br />Concatenation with StringBuffer: "
```

```
 + (d2.getTime() - d1.getTime()) + " milliseconds");
```

TIY

这段代码对字符串连接进行两个测试，第一个使用加号，第二个使用 StringBuffer 类。每个操作都连接 10000 个字符串。日期值 d1 和 d2 用于判断完成操作需要的时间。请注意，创建 Date 对象时，如果没有参数，赋予对象的是当前的日期和时间。要计算连接操作历经多少时间，把日期的毫秒表示（用 getTime() 方法的返回值）相减即可。这是衡量 JavaScript 性能的常见方法。该测试的结果可以帮助您比较使用 StringBuffer 类与使用加号的效率差异。

修改对象

通过使用 ECMAScript，不仅可以创建对象，还可以修改已有对象的行为。

prototype 属性不仅可以定义构造函数的属性和方法，还可以为本地对象添加属性和方法。

创建新方法

通过已有的方法创建新方法

可以用 prototype 属性为任何已有的类定义新方法，就像处理自己的类一样。例如，还记得 Number 类的 toString() 方法吗？如果给它传递参数 16，它将输出十六进制的字符串。如果这个方法的参数是 2，那么它将输出二进制的字符串。我们可以创建一个方法，可以把数字对象直接转换为十六进制字符串。创建这个方法非常简单：

```
Number.prototype.toHexString = function() {
  return this.toString(16);
};
```

在此环境中，关键字 this 指向 Number 的实例，因此可完全访问 Number 的所有方法。有了这段代码，可实现下面的操作：

```
var iNum = 15;
alert(iNum.toHexString());		//输出 "F"
```

TIY

由于数字 15 等于十六进制中的 F，因此警告将显示 "F"。

重命名已有方法

我们还可以为已有的方法命名更易懂的名称。例如，可以给 Array 类添加两个方法 enqueue() 和 dequeue()，只让它们反复调用已有的 push() 和 shift() 方法即可：

```
Array.prototype.enqueue = function(vItem) {
  this.push(vItem);
};
```

```
Array.prototype.dequeue = function() {
    return this.shift();
};
```

TIY

添加与已有方法无关的方法

当然，还可以添加与已有方法无关的方法。例如，假设要判断某个项在数组中的位置，没有本地方法可以做这种事情。我们可以轻松地创建下面的方法：

```
Array.prototype.indexOf = function (vItem) {
    for (var i=0; i<this.length; i++) {
        if (vItem == this[i]) {
            return i;
        }
    }

    return -1;
}
```

该方法 indexOf() 与 String 类的同名方法保持一致，在数组中检索每个项，直到发现与传进来的项相同的项目为止。如果找到相同的项，则返回该项的位置，否则，返回 -1。有了这种定义，我们可以编写下面的代码：

```
var aColors = new Array("red","green","blue");
alert(aColors.indexOf("green"));    //输出 "1"
```

TIY

为本地对象添加新方法

最后，如果想给 ECMAScript 中每个本地对象添加新方法，必须在 Object 对象的 prototype 属性上定义它。前面的章节我们讲过，所有本地对象都继承了 Object 对象，所以对 Object 对象做任何改变，都会反应在所有本地对象上。例如，如果想添加一个用警告输出对象的当前值的方法，可以采用下面的代码：

```
Object.prototype.showValue = function () {
    alert(this.valueOf());
};

var str = "hello";
var iNum = 25;
str.showValue();          //输出 "hello"
iNum.showValue();         //输出 "25"
```

TIY

这里，String 和 Number 对象都从 Object 对象继承了 showValue() 方法，分别在它们的对

象上调用该方法，将显示 "hello" 和 "25"。

重定义已有方法

就像能给已有的类定义新方法一样，也可重定义已有的方法。如前面的章节所述，函数名只是指向函数的指针，因此可以轻松地指向其他函数。如果修改了本地方法，如 toString()，会出现什么情况呢？

```
Function.prototype.toString = function() {
  return "Function code hidden";
}
```

前面的代码完全合法，运行结果完全符合预期：

```
function sayHi() {
  alert("hi");
}

alert(sayHi.toString());   //输出 "Function code hidden"
```

TIY

也许你还记得，Function 对象这一章中介绍过 Function 的 toString() 方法通常输出的是函数的源代码。覆盖该方法，可以返回另一个字符串（在这个例子中，可以返回 "Function code hidden"）。不过，toString() 指向的原始函数怎么了呢？它将被无用存储单元回收程序回收，因为它被完全废弃了。没有能够恢复原始函数的方法，所以在覆盖原始方法前，比较安全的做法是存储它的指针，以便以后的使用。有时你甚至可能在新方法中调用原始方法：

```
Function.prototype.originalToString = Function.prototype.toString;

Function.prototype.toString = function() {
  if (this.originalToString().length > 100) {
    return "Function too long to display.";
  } else {
    return this.originalToString();
  }
};
```

TIY

在这段代码中，第一行代码把对当前 toString() 方法的引用保存在属性 originalToString 中。然后用定制的方法覆盖了 toString() 方法。新方法将检查该函数源代码的长度是否大于 100。如果是，就返回错误信息，说明该函数代码太长，否则调用 originalToString() 方法，返回函数的源代码。

极晚绑定（Very Late Binding）

从技术上讲，根本不存在极晚绑定。本书采用该术语描述 ECMAScript 中的一种现象，即能够在对象实例化后再定义它的方法。例如：

```
var o = new Object();

Object.prototype.sayHi = function () {
   alert("hi");
};

o.sayHi();
```

TIY

在大多数程序设计语言中，必须在实例化对象之前定义对象的方法。这里，方法 sayHi() 是在创建 Object 类的一个实例之后来添加进来的。在传统语言中不仅没听说过这种操作，也没听说过该方法还会自动赋予 Object 对象的实例并能立即使用（接下来的一行）。