为"cond"，而字符串的剩余部分保持不变。通过将第一个参数改为带全局标记的正则表达式，字符串中的所有"at"都被替换成了"ond"。

第二个参数是字符串的情况下，有几个特殊的字符序列，可以用来插入正则表达式操作的值。ECMA-262 中规定了下表中的值。

字符序列	替换文本
$$	$
$&	匹配整个模式的子字符串。与 RegExp.lastMatch 相同
$'	匹配的子字符串之前的字符串。与 RegExp.rightContext 相同
$`	匹配的子字符串之后的字符串。与 RegExp.leftContext 相同
$*n*	匹配第 *n* 个捕获组的字符串，其中 *n* 是 0~9。比如，$1 是匹配第一个捕获组的字符串，$2 是匹配第二个捕获组的字符串，以此类推。如果没有捕获组，则值为空字符串
$*nn*	匹配第 *nn* 个捕获组字符串，其中 *nn* 是 01~99。比如，$01 是匹配第一个捕获组的字符串，$02 是匹配第二个捕获组的字符串，以此类推。如果没有捕获组，则值为空字符串

使用这些特殊的序列，可以在替换文本中使用之前匹配的内容，如下面的例子所示：

```
let text = "cat, bat, sat, fat";
result = text.replace(/(.at)/g, "word ($1)");
console.log(result);  // word (cat), word (bat), word (sat), word (fat)
```

这里，每个以"at"结尾的词都会被替换成"word"后跟一对小括号，其中包含捕获组匹配的内容$1。

replace()的第二个参数可以是一个函数。在只有一个匹配项时，这个函数会收到 3 个参数：与整个模式匹配的字符串、匹配项在字符串中的开始位置，以及整个字符串。在有多个捕获组的情况下，每个匹配捕获组的字符串也会作为参数传给这个函数，但最后两个参数还是与整个模式匹配的开始位置和原始字符串。这个函数应该返回一个字符串，表示应该把匹配项替换成什么。使用函数作为第二个参数可以更细致地控制替换过程，如下所示：

```
function htmlEscape(text) {
  return text.replace(/[<>"&]/g, function(match, pos, originalText) {
    switch(match) {
      case "<":
        return "&lt;";
      case ">":
        return "&gt;";
      case "&":
        return "&amp;";
      case "\"":
        return "&quot;";
    }
  });
}

console.log(htmlEscape("<p class=\"greeting\">Hello world!</p>"));
// "&lt;p class=&quot;greeting&quot;&gt;Hello world!</p>"
```

这里，函数 htmlEscape()用于将一段 HTML 中的 4 个字符替换成对应的实体：小于号、大于号、和号，还有双引号（都必须经过转义）。实现这个任务最简单的办法就是用一个正则表达式查找这些字符，然后定义一个函数，根据匹配的每个字符分别返回特定的 HTML 实体。

最后一个与模式匹配相关的字符串方法是 split()。这个方法会根据传入的分隔符将字符串拆分成

数组。作为分隔符的参数可以是字符串，也可以是 RegExp 对象。（字符串分隔符不会被这个方法当成正则表达式。）还可以传入第二个参数，即数组大小，确保返回的数组不会超过指定大小。来看下面的例子：

```
let colorText = "red,blue,green,yellow";
let colors1 = colorText.split(",");       // ["red", "blue", "green", "yellow"]
let colors2 = colorText.split(",", 2);    // ["red", "blue"]
let colors3 = colorText.split(/[^,]+/);   // ["", ",", ",", ",", ""]
```

在这里，字符串 colorText 是一个逗号分隔的颜色名称符串。调用 split(",")会得到包含这些颜色名的数组，基于逗号进行拆分。要把数组元素限制为 2 个，传入第二个参数 2 即可。最后，使用正则表达式可以得到一个包含逗号的数组。注意在最后一次调用 split()时，返回的数组前后包含两个空字符串。这是因为正则表达式指定的分隔符出现在了字符串开头（"red"）和末尾（"yellow"）。

12. localeCompare()方法

最后一个方法是 localeCompare()，这个方法比较两个字符串，返回如下 3 个值中的一个。

- 如果按照字母表顺序，字符串应该排在字符串参数前头，则返回负值。（通常是-1，具体还要看与实际值相关的实现。）
- 如果字符串与字符串参数相等，则返回 0。
- 如果按照字母表顺序，字符串应该排在字符串参数后头，则返回正值。（通常是 1，具体还要看与实际值相关的实现。）

下面是一个例子：

```
let stringValue = "yellow";
console.log(stringValue.localeCompare("brick"));  // 1
console.log(stringValue.localeCompare("yellow")); // 0
console.log(stringValue.localeCompare("zoo"));    // -1
```

在这里，字符串"yellow"与 3 个不同的值进行了比较："brick"、"yellow"和"zoo"。"brick"按字母表顺序应该排在"yellow"前头，因此 localeCompare()返回 1。"yellow"等于"yellow"，因此"localeCompare()"返回 0。最后，"zoo"在"yellow"后面，因此 localeCompare()返回-1。强调一下，因为返回的具体值可能因具体实现而异，所以最好像下面的示例中一样使用 localeCompare()：

```
function determineOrder(value) {
  let result = stringValue.localeCompare(value);
  if (result < 0) {
    console.log(`The string 'yellow' comes before the string '${value}'.`);
  } else if (result > 0) {
    console.log(`The string 'yellow' comes after the string '${value}'.`);
  } else {
    console.log(`The string 'yellow' is equal to the string '${value}'.`);
  }
}

determineOrder("brick");
determineOrder("yellow");
determineOrder("zoo");
```

这样一来，就可以保证在所有实现中都能正确判断字符串的顺序了。

localeCompare()的独特之处在于，实现所在的地区（国家和语言）决定了这个方法如何比较字符串。在美国，英语是 ECMAScript 实现的标准语言，localeCompare()区分大小写，大写字母排在小

写字母前面。但其他地区未必是这种情况。

13. HTML 方法

早期的浏览器开发商认为使用 JavaScript 动态生成 HTML 标签是一个需求。因此，早期浏览器扩展了规范，增加了辅助生成 HTML 标签的方法。下表总结了这些 HTML 方法。不过，这些方法基本上已经没有人使用了，因为结果通常不是语义化的标记。

方　法	输　出
anchor(*name*)	*string*
big()	<big>*string*</big>
bold()	*string*
fixed()	<tt>*string*</tt>
fontcolor(*color*)	*string*
fontsize(*size*)	*string*
italics()	*string*
link(url)	*string*
small()	*string*
strike()	<strike>*string*</strike>
sub()	_{*string*}
sup()	^{*string*}

5.4　单例内置对象

ECMA-262 对内置对象的定义是“任何由 ECMAScript 实现提供、与宿主环境无关，并在 ECMAScript 程序开始执行时就存在的对象”。这就意味着，开发者不用显式地实例化内置对象，因为它们已经实例化好了。前面我们已经接触了大部分内置对象，包括 Object、Array 和 String。本节介绍 ECMA-262 定义的另外两个单例内置对象：Global 和 Math。

5.4.1　Global

Global 对象是 ECMAScript 中最特别的对象，因为代码不会显式地访问它。ECMA-262 规定 Global 对象为一种兜底对象，它所针对的是不属于任何对象的属性和方法。事实上，不存在全局变量或全局函数这种东西。在全局作用域中定义的变量和函数都会变成 Global 对象的属性 。本书前面介绍的函数，包括 isNaN()、isFinite()、parseInt()和 parseFloat()，实际上都是 Global 对象的方法。除了这些，Global 对象上还有另外一些方法。

1. URL 编码方法

encodeURI()和 encodeURIComponent()方法用于编码统一资源标识符(URI)，以便传给浏览器。有效的 URI 不能包含某些字符，比如空格。使用 URI 编码方法来编码 URI 可以让浏览器能够理解它们，同时又以特殊的 UTF-8 编码替换掉所有无效字符。

ecnodeURI()方法用于对整个 URI 进行编码，比如"www.wrox.com/illegal value.js"。而 encodeURIComponent()方法用于编码 URI 中单独的组件，比如前面 URL 中的"illegal value.js"。

这两个方法的主要区别是，encodeURI()不会编码属于URL组件的特殊字符，比如冒号、斜杠、问号、井号，而encodeURIComponent()会编码它发现的所有非标准字符。来看下面的例子：

```
let uri = "http://www.wrox.com/illegal value.js#start";

// "http://www.wrox.com/illegal%20value.js#start"
console.log(encodeURI(uri));

// "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start"
console.log(encodeURIComponent(uri));
```

这里使用 encodeURI()编码后，除空格被替换为%20 之外，没有任何变化。而 encodeURI-Component()方法将所有非字母字符都替换成了相应的编码形式。这就是使用encodeURI()编码整个URI，但只使用encodeURIComponent()编码那些会追加到已有URI后面的字符串的原因。

> **注意** 一般来说，使用encodeURIComponent()应该比使用encodeURI()的频率更高，这是因为编码查询字符串参数比编码基准URI的次数更多。

与encodeURI()和encodeURIComponent()相对的是decodeURI()和decodeURIComponent()。decodeURI()只对使用encodeURI()编码过的字符解码。例如，%20会被替换为空格，但%23不会被替换为井号（#），因为井号不是由encodeURI()替换的。类似地，decodeURIComponent()解码所有被encodeURIComponent()编码的字符，基本上就是解码所有特殊值。来看下面的例子：

```
let uri = "http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.js%23start";

// http%3A%2F%2Fwww.wrox.com%2Fillegal value.js%23start
console.log(decodeURI(uri));

// http:// www.wrox.com/illegal value.js#start
console.log(decodeURIComponent(uri));
```

这里，uri 变量中包含一个使用 encodeURIComponent()编码过的字符串。首先输出的是使用decodeURI()解码的结果，可以看到只用空格替换了%20。然后是使用 decodeURIComponent()解码的结果，其中替换了所有特殊字符，并输出了没有包含任何转义的字符串。（这个字符串不是有效的URL。）

> **注意** URI方法encodeURI()、encodeURIComponent()、decodeURI()和decodeURI-Component()取代了escape()和unescape()方法，后者在ECMA-262第3版中就已经废弃了。URI方法始终是首选方法，因为它们对所有Unicode字符进行编码，而原来的方法只能正确编码ASCII字符。不要在生产环境中使用escape()和unescape()。

2. eval()方法

最后一个方法可能是整个ECMAScript语言中最强大的了，它就是eval()。这个方法就是一个完整的ECMAScript解释器，它接收一个参数，即一个要执行的ECMAScript（JavaScript）字符串。来看一个例子：

```
eval("console.log('hi')");
```

上面这行代码的功能与下一行等价：

```
console.log("hi");
```

当解释器发现 eval()调用时，会将参数解释为实际的 ECMAScript 语句，然后将其插入到该位置。通过 eval()执行的代码属于该调用所在上下文，被执行的代码与该上下文拥有相同的作用域链。这意味着定义在包含上下文中的变量可以在 eval()调用内部被引用，比如下面这个例子：

```
let msg = "hello world";
eval("console.log(msg)");  // "hello world"
```

这里，变量 msg 是在 eval()调用的外部上下文中定义的，而 console.log()显示了文本"hello world"。这是因为第二行代码会被替换成一行真正的函数调用代码。类似地，可以在 eval()内部定义一个函数或变量，然后在外部代码中引用，如下所示：

```
eval("function sayHi() { console.log('hi'); }");
sayHi();
```

这里，函数 sayHi()是在 eval()内部定义的。因为该调用会被替换为真正的函数定义，所以才可能在下一行代码中调用 sayHi()。对于变量也是一样的：

```
eval("let msg = 'hello world';");
console.log(msg);  // Reference Error: msg is not defined
```

通过 eval()定义的任何变量和函数都不会被提升，这是因为在解析代码的时候，它们是被包含在一个字符串中的。它们只是在 eval()执行的时候才会被创建。

在严格模式下，在 eval()内部创建的变量和函数无法被外部访问。换句话说，最后两个例子会报错。同样，在严格模式下，赋值给 eval 也会导致错误：

```
"use strict";
eval = "hi";  // 导致错误
```

> **注意**　解释代码字符串的能力是非常强大的，但也非常危险。在使用 eval()的时候必须极为慎重，特别是在解释用户输入的内容时。因为这个方法会对 XSS 利用暴露出很大的攻击面。恶意用户可能插入会导致你网站或应用崩溃的代码。

3. Global 对象属性

Global 对象有很多属性，其中一些前面已经提到过了。像 undefined、NaN 和 Infinity 等特殊值都是 Global 对象的属性。此外，所有原生引用类型构造函数，比如 Object 和 Function，也都是 Global 对象的属性。下表列出了所有这些属性。

属　性	说　明
undefined	特殊值 undefined
NaN	特殊值 NaN
Infinity	特殊值 Infinity
Object	Object 的构造函数
Array	Array 的构造函数
Function	Function 的构造函数
Boolean	Boolean 的构造函数
String	String 的构造函数

（续）

属　性	说　明
Number	Number 的构造函数
Date	Date 的构造函数
RegExp	RegExp 的构造函数
Symbol	Symbol 的伪构造函数
Error	Error 的构造函数
EvalError	EvalError 的构造函数
RangeError	RangeError 的构造函数
ReferenceError	ReferenceError 的构造函数
SyntaxError	SyntaxError 的构造函数
TypeError	TypeError 的构造函数
URIError	URIError 的构造函数

4. window 对象

虽然 ECMA-262 没有规定直接访问 Global 对象的方式，但浏览器将 window 对象实现为 Global 对象的代理。因此，所有全局作用域中声明的变量和函数都变成了 window 的属性。来看下面的例子：

```
var color = "red";

function sayColor() {
  console.log(window.color);
}

window.sayColor(); // "red"
```

这里定义了一个名为 color 的全局变量和一个名为 sayColor()的全局函数。在 sayColor()内部，通过 window.color 访问了 color 变量，说明全局变量变成了 window 的属性。接着，又通过 window 对象直接调用了 window.sayColor()函数，从而输出字符串。

> **注意** window 对象在 JavaScript 中远不止实现了 ECMAScript 的 Global 对象那么简单。关于 window 对象的更多介绍，请参考第 12 章。

另一种获取 Global 对象的方式是使用如下的代码：

```
let global = function() {
  return this;
}();
```

这段代码创建一个立即调用的函数表达式，返回了 this 的值。如前所述，当一个函数在没有明确（通过成为某个对象的方法，或者通过 call()/apply()）指定 this 值的情况下执行时，this 值等于 Global 对象。因此，调用一个简单返回 this 的函数是在任何执行上下文中获取 Global 对象的通用方式。

5.4.2 Math

ECMAScript 提供了 Math 对象作为保存数学公式、信息和计算的地方。Math 对象提供了一些辅助计算的属性和方法。

> **注意**　Math 对象上提供的计算要比直接在 JavaScript 实现的快得多，因为 Math 对象上的计算使用了 JavaScript 引擎中更高效的实现和处理器指令。但使用 Math 计算的问题是精度会因浏览器、操作系统、指令集和硬件而异。

1. Math 对象属性

Math 对象有一些属性，主要用于保存数学中的一些特殊值。下表列出了这些属性。

属　性	说　明
Math.E	自然对数的基数 e 的值
Math.LN10	10 为底的自然对数
Math.LN2	2 为底的自然对数
Math.LOG2E	以 2 为底 e 的对数
Math.LOG10E	以 10 为底 e 的对数
Math.PI	π 的值
Math.SQRT1_2	1/2 的平方根
Math.SQRT2	2 的平方根

这些值的含义和用法超出了本书的范畴，但都是 ECMAScript 规范定义的，并可以在你需要时使用。

2. min()和 max()方法

Math 对象也提供了很多辅助执行简单或复杂数学计算的方法。

min()和 max()方法用于确定一组数值中的最小值和最大值。这两个方法都接收任意多个参数，如下面的例子所示：

```
let max = Math.max(3, 54, 32, 16);
console.log(max);  // 54

let min = Math.min(3, 54, 32, 16);
console.log(min);  // 3
```

在 3、54、32 和 16 中，Math.max()返回 54，Math.min()返回 3。使用这两个方法可以避免使用额外的循环和 if 语句来确定一组数值的最大最小值。

要知道数组中的最大值和最小值，可以像下面这样使用扩展操作符：

```
let values = [1, 2, 3, 4, 5, 6, 7, 8];
let max = Math.max(...val);
```

3. 舍入方法

接下来是用于把小数值舍入为整数的 4 个方法：Math.ceil()、Math.floor()、Math.round()和 Math.fround()。这几个方法处理舍入的方式如下所述。

- Math.ceil()方法始终向上舍入为最接近的整数。

- Math.floor()方法始终向下舍入为最接近的整数。
- Math.round()方法执行四舍五入。
- Math.fround()方法返回数值最接近的单精度（32 位）浮点值表示。

以下示例展示了这些方法的用法：

```
console.log(Math.ceil(25.9));   // 26
console.log(Math.ceil(25.5));   // 26
console.log(Math.ceil(25.1));   // 26

console.log(Math.round(25.9));  // 26
console.log(Math.round(25.5));  // 26
console.log(Math.round(25.1));  // 25

console.log(Math.fround(0.4));  // 0.4000000059604645
console.log(Math.fround(0.5));  // 0.5
console.log(Math.fround(25.9)); // 25.899999618530273

console.log(Math.floor(25.9));  // 25
console.log(Math.floor(25.5));  // 25
console.log(Math.floor(25.1));  // 25
```

对于 25 和 26（不包含）之间的所有值，Math.ceil()都会返回 26，因为它始终向上舍入。Math.round()只在数值大于等于 25.5 时返回 26，否则返回 25。最后，Math.floor()对所有 25 和 26（不包含）之间的值都返回 25。

4. random()方法

Math.random()方法返回一个 0~1 范围内的随机数，其中包含 0 但不包含 1。对于希望显示随机名言或随机新闻的网页，这个方法是非常方便的。可以基于如下公式使用 Math.random()从一组整数中随机选择一个数：

```
number = Math.floor(Math.random() * total_number_of_choices + first_possible_value)
```

这里使用了 Math.floor()方法，因为 Math.random()始终返回小数，即便乘以一个数再加上一个数也是小数。因此，如果想从 1~10 范围内随机选择一个数，代码就是这样的：

```
let num = Math.floor(Math.random() * 10 + 1);
```

这样就有 10 个可能的值（1~10），其中最小的值是 1。如果想选择一个 2~10 范围内的值，则代码就要写成这样：

```
let num = Math.floor(Math.random() * 9 + 2);
```

2~10 只有 9 个数，所以可选总数（total_number_of_choices）是 9，而最小可能的值（first_possible_value）是 2。很多时候，通过函数来算出可选总数和最小可能的值可能更方便，比如：

```
function selectFrom(lowerValue, upperValue) {
  let choices = upperValue - lowerValue + 1;
  return Math.floor(Math.random() * choices + lowerValue);
}

let num = selectFrom(2,10);
console.log(num);  // 2~10 范围内的值，其中包含 2 和 10
```

这里的函数 selectFrom()接收两个参数：应该返回的最小值和最大值。通过将这两个值相减再

加 1 得到可选总数，然后再套用上面的公式。于是，调用 selectFrom(2,10)就可以从 2~10（包含）范围内选择一个值了。使用这个函数，从一个数组中随机选择一个元素就很容易，比如：

```
let colors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
let color = colors[selectFrom(0, colors.length-1)];
```

在这个例子中，传给 selecFrom()的第二个参数是数组长度减 1，即数组最大的索引值。

> **注意**　Math.random()方法在这里出于演示目的是没有问题的。如果是为了加密而需要生成随机数（传给生成器的输入需要较高的不确定性），那么建议使用 window.crypto.getRandomValues()。

5. 其他方法

Math 对象还有很多涉及各种简单或高阶数运算的方法。讨论每种方法的具体细节或者它们的适用场景超出了本书的范畴。不过，下表还是总结了 Math 对象的其他方法。

方　法	说　明
Math.abs(*x*)	返回 *x* 的绝对值
Math.exp(*x*)	返回 Math.E 的 *x* 次幂
Math.expm1(*x*)	等于 Math.exp(*x*) - 1
Math.log(*x*)	返回 *x* 的自然对数
Math.log1p(*x*)	等于 1 + Math.log(*x*)
Math.pow(*x*, *power*)	返回 *x* 的 *power* 次幂
Math.hypot(...*nums*)	返回 *nums* 中每个数平方和的平方根
Math.clz32(*x*)	返回 32 位整数 *x* 的前置零的数量
Math.sign(*x*)	返回表示 *x* 符号的 1、0、-0 或-1
Math.trunc(*x*)	返回 *x* 的整数部分，删除所有小数
Math.sqrt(*x*)	返回 *x* 的平方根
Math.cbrt(*x*)	返回 *x* 的立方根
Math.acos(*x*)	返回 *x* 的反余弦
Math.acosh(*x*)	返回 *x* 的反双曲余弦
Math.asin(*x*)	返回 *x* 的反正弦
Math.asinh(*x*)	返回 *x* 的反双曲正弦
Math.atan(*x*)	返回 *x* 的反正切
Math.atanh(*x*)	返回 *x* 的反双曲正切
Math.atan2(*y*, *x*)	返回 *y*/*x* 的反正切
Math.cos(*x*)	返回 *x* 的余弦
Math.sin(*x*)	返回 *x* 的正弦
Math.tan(*x*)	返回 *x* 的正切

即便这些方法都是由 ECMA-262 定义的，对正弦、余弦、正切等计算的实现仍然取决于浏览器，因为计算这些值的方式有很多种。结果，这些方法的精度可能因实现而异。

5.5 小结

JavaScript 中的对象称为引用值，几种内置的引用类型可用于创建特定类型的对象。

- 引用值与传统面向对象编程语言中的类相似，但实现不同。
- `Date` 类型提供关于日期和时间的信息，包括当前日期、时间及相关计算。
- `RegExp` 类型是 ECMAScript 支持正则表达式的接口，提供了大多数基础的和部分高级的正则表达式功能。

JavaScript 比较独特的一点是，函数实际上是 `Function` 类型的实例，也就是说函数也是对象。因为函数也是对象，所以函数也有方法，可以用于增强其能力。

由于原始值包装类型的存在，JavaScript 中的原始值可以被当成对象来使用。有 3 种原始值包装类型：`Boolean`、`Number` 和 `String`。它们都具备如下特点。

- 每种包装类型都映射到同名的原始类型。
- 以读模式访问原始值时，后台会实例化一个原始值包装类型的对象，借助这个对象可以操作相应的数据。
- 涉及原始值的语句执行完毕后，包装对象就会被销毁。

当代码开始执行时，全局上下文中会存在两个内置对象：`Global` 和 `Math`。其中，`Global` 对象在大多数 ECMAScript 实现中无法直接访问。不过，浏览器将其实现为 `window` 对象。所有全局变量和函数都是 `Global` 对象的属性。`Math` 对象包含辅助完成复杂计算的属性和方法。