第5章 引用类型

本章内容
- 使用对象
- 创建并操作数组
- 理解基本的 JavaScript 类型
- 使用基本类型和基本包装类型

引用类型的值（对象）是引用类型的一个实例。在 ECMAScript 中，引用类型是一种数据结构，用于将数据和功能组织在一起。它也常被称为类，但这种称呼并不妥当。尽管 ECMAScript 从技术上讲是一门面向对象的语言，但它不具备传统的面向对象语言所支持的类和接口等基本结构。引用类型有时候也被称为对象定义，因为它们描述的是一类对象所具有的属性和方法。

虽然引用类型与类看起来相似，但它们并不是相同的概念。为避免混淆，本书将不使用类这个概念。

如前所述，对象是某个特定引用类型的实例。新对象是使用 new 操作符后跟一个构造函数来创建的。构造函数本身就是一个函数，只不过该函数是出于创建新对象的目的而定义的。请看下面这行代码：

```
var person = new Object();
```

这行代码创建了 Object 引用类型的一个新实例，然后把该实例保存在了变量 person 中。使用的构造函数是 Object，它只为新对象定义了默认的属性和方法。ECMAScript 提供了很多原生引用类型（例如 Object），以便开发人员用以实现常见的计算任务。

5.1 Object 类型

到目前为止，我们看到的大多数引用类型值都是 Object 类型的实例；而且，Object 也是 ECMAScript 中使用最多的一个类型。虽然 Object 的实例不具备多少功能，但对于在应用程序中存储和传输数据而言，它们确实是非常理想的选择。

创建 Object 实例的方式有两种。第一种是使用 new 操作符后跟 Object 构造函数，如下所示：

```
var person = new Object();
person.name = "Nicholas";
person.age = 29;
```

ObjectTypeExample01.htm

另一种方式是使用**对象字面量**表示法。对象字面量是对象定义的一种简写形式，目的在于简化创建包含大量属性的对象的过程。下面这个例子就使用了对象字面量语法定义了与前面那个例子中相同的person对象：

```
var person = {
    name : "Nicholas",
    age : 29
};
```

ObjectTypeExample02.htm

在这个例子中，左边的花括号（{）表示对象字面量的开始，因为它出现在了表达式上下文（expression context）中。ECMAScript中的表达式上下文指的是能够返回一个值（表达式）。赋值操作符表示后面是一个值，所以左花括号在这里表示一个表达式的开始。同样的花括号，如果出现在一个语句上下文（statement context）中，例如跟在if语句条件的后面，则表示一个语句块的开始。

然后，我们定义了name属性，之后是一个冒号，再后面是这个属性的值。在对象字面量中，使用逗号来分隔不同的属性，因此"Nicholas"后面是一个逗号。但是，在age属性的值29的后面不能添加逗号，因为age是这个对象的最后一个属性。在最后一个属性后面添加逗号，会在IE7及更早版本和Opera中导致错误。

在使用对象字面量语法时，属性名也可以使用字符串，如下面这个例子所示。

```
var person = {
    "name" : "Nicholas",
    "age" : 29,
    5 : true
};
```

这个例子会创建一个对象，包含三个属性：name、age和5。但这里的数值属性名会自动转换为字符串。

另外，使用对象字面量语法时，如果留空其花括号，则可以定义只包含默认属性和方法的对象，如下所示：

```
var person = {};          //与 new Object()相同
person.name = "Nicholas";
person.age = 29;
```

这个例子与本节前面的例子是等价的，只不过看起来似乎有点奇怪。关于对象字面量语法，我们推荐只在考虑对象属性名的可读性时使用。

在通过对象字面量定义对象时，实际上不会调用Object构造函数（Firefox 2及更早版本会调用Object构造函数；但Firefox 3之后就不会了）。

虽然可以使用前面介绍的任何一种方法来定义对象，但开发人员更青睐对象字面量语法，因为这种语法要求的代码量少，而且能够给人封装数据的感觉。实际上，对象字面量也是向函数传递大量可选参数的首选方式，例如：

```
function displayInfo(args) {
    var output = "";

    if (typeof args.name == "string"){
```

```
        output += "Name: " + args.name + "\n";
    }

    if (typeof args.age == "number") {
        output += "Age: " + args.age + "\n";
    }

    alert(output);
}

displayInfo({
    name: "Nicholas",
    age: 29
});

displayInfo({
    name: "Greg"
});
```

ObjectTypeExample04.htm

在这个例子中，函数 displayInfo()接受一个名为 args 的参数。这个参数可能带有一个名为 name 或 age 的属性，也可能这两个属性都有或者都没有。在这个函数内部，我们通过 typeof 操作符来检测每个属性是否存在，然后再基于相应的属性来构建一条要显示的消息。然后，我们调用了两次这个函数，每次都使用一个对象字面量来指定不同的数据。这两次调用传递的参数虽然不同，但函数都能正常执行。

这种传递参数的模式最适合需要向函数传入大量可选参数的情形。一般来讲，命名参数虽然容易处理，但在有多个可选参数的情况下就会显示不够灵活。最好的做法是对那些必需值使用命名参数，而使用对象字面量来封装多个可选参数。

一般来说，访问对象属性时使用的都是点表示法，这也是很多面向对象语言中通用的语法。不过，在 JavaScript 也可以使用方括号表示法来访问对象的属性。在使用方括号语法时，应该将要访问的属性以字符串的形式放在方括号中，如下面的例子所示。

```
alert(person["name"]);          //"Nicholas"
alert(person.name);             //"Nicholas"
```

从功能上看，这两种访问对象属性的方法没有任何区别。但方括号语法的主要优点是可以通过变量来访问属性，例如：

```
var propertyName = "name";
alert(person[propertyName]);    //"Nicholas"
```

如果属性名中包含会导致语法错误的字符，或者属性名使用的是关键字或保留字，也可以使用方括号表示法。例如：

```
person["first name"] = "Nicholas";
```

由于"first name"中包含一个空格，所以不能使用点表示法来访问它。然而，属性名中是可以包含非字母非数字的，这时候就可以使用方括号表示法来访问它们。

通常，除非必须使用变量来访问属性，否则我们建议使用点表示法。

5.2 Array 类型

除了 Object 之外，Array 类型恐怕是 ECMAScript 中最常用的类型了。而且，ECMAScript 中的数组与其他多数语言中的数组有着相当大的区别。虽然 ECMAScript 数组与其他语言中的数组都是数据的有序列表，但与其他语言不同的是，ECMAScript 数组的每一项可以保存任何类型的数据。也就是说，可以用数组的第一个位置来保存字符串，用第二位置来保存数值，用第三个位置来保存对象，以此类推。而且，ECMAScript 数组的大小是可以动态调整的，即可以随着数据的添加自动增长以容纳新增数据。

创建数组的基本方式有两种。第一种是使用 Array 构造函数，如下面的代码所示。

```
var colors = new Array();
```

如果预先知道数组要保存的项目数量，也可以给构造函数传递该数量，而该数量会自动变成 length 属性的值。例如，下面的代码将创建 length 值为 20 的数组。

```
var colors = new Array(20);
```

也可以向 Array 构造函数传递数组中应该包含的项。以下代码创建了一个包含 3 个字符串值的数组：

```
var colors = new Array("red", "blue", "green");
```

当然，给构造函数传递一个值也可以创建数组。但这时候问题就复杂一点了，因为如果传递的是数值，则会按照该数值创建包含给定项数的数组；而如果传递的是其他类型的参数，则会创建包含那个值的只有一项的数组。下面就两个例子：

```
var colors = new Array(3);          // 创建一个包含 3 项的数组
var names = new Array("Greg");      // 创建一个包含 1 项，即字符串"Greg"的数组
```

ArrayTypeExample01.htm

另外，在使用 Array 构造函数时也可以省略 new 操作符。如下面的例子所示，省略 new 操作符的结果相同：

```
var colors = Array(3);              // 创建一个包含 3 项的数组
var names = Array("Greg");          // 创建一个包含 1 项，即字符串"Greg"的数组
```

创建数组的第二种基本方式是使用数组字面量表示法。数组字面量由一对包含数组项的方括号表示，多个数组项之间以逗号隔开，如下所示：

```
var colors = ["red", "blue", "green"];  // 创建一个包含 3 个字符串的数组
var names = [];                         // 创建一个空数组
var values = [1,2,];                    // 不要这样！这样会创建一个包含 2 或 3 项的数组
var options = [,,,,,];                 // 不要这样！这样会创建一个包含 5 或 6 项的数组
```

ArrayTypeExample02.htm

以上代码的第一行创建了一个包含 3 个字符串的数组。第二行使用一对空方括号创建了一个空数组。第三行展示了在数组字面量的最后一项添加逗号的结果：在 IE 中，values 会成为一个包含 3 个项且每项的值分别为 1、2 和 undefined 的数组；在其他浏览器中，values 会成为一个包含 2 项且值分别为 1 和 2 的数组。原因是 IE8 及之前版本中的 ECMAScript 实现在数组字面量方面存在 bug。由于这个 bug

导致的另一种情况如最后一行代码所示，该行代码可能会创建包含 5 项的数组（在 IE9+、Firefox、Opera、Safari 和 Chrome 中），也可能会创建包含 6 项的数组（在 IE8 及更早版本中）。在像这种省略值的情况下，每一项都将获得 undefined 值；这个结果与调用 Array 构造函数时传递项数在逻辑上是相同的。但是由于 IE 的实现与其他浏览器不一致，因此我们强烈建议不要使用这种语法。

与对象一样，在使用数组字面量表示法时，也不会调用 Array 构造函数（Firefox 3 及更早版本除外）。

在读取和设置数组的值时，要使用方括号并提供相应值的基于 0 的数字索引，如下所示：

```
var colors = ["red", "blue", "green"];  // 定义一个字符串数组
alert(colors[0]);                       // 显示第一项
colors[2] = "black";                    // 修改第三项
colors[3] = "brown";                    // 新增第四项
```

方括号中的索引表示要访问的值。如果索引小于数组中的项数，则返回对应项的值，就像这个例子中的 colors[0]会显示"red"一样。设置数组的值也使用相同的语法，但会替换指定位置的值。如果设置某个值的索引超过了数组现有项数，如这个例子中的 colors[3]所示，数组就会自动增加到该索引值加 1 的长度（就这个例子而言，索引是 3，因此数组长度就是 4）。

数组的项数保存在其 length 属性中，这个属性始终会返回 0 或更大的值，如下面这个例子所示：

```
var colors = ["red", "blue", "green"];      // 创建一个包含 3 个字符串的数组
var names = [];                             // 创建一个空数组

alert(colors.length);               //3
alert(names.length);                //0
```

数组的 length 属性很有特点——它不是只读的。因此，通过设置这个属性，可以从数组的末尾移除项或向数组中添加新项。请看下面的例子：

```
var colors = ["red", "blue", "green"];      // 创建一个包含 3 个字符串的数组
colors.length = 2;
alert(colors[2]);                   //undefined
```

ArrayTypeExample03.htm

这个例子中的数组 colors 一开始有 3 个值。将其 length 属性设置为 2 会移除最后一项（位置为 2 的那一项），结果再访问 colors[2]就会显示 undefined 了。如果将其 length 属性设置为大于数组项数的值，则新增的每一项都会取得 undefined 值，如下所示：

```
var colors = ["red", "blue", "green"];    // 创建一个包含 3 个字符串的数组
colors.length = 4;
alert(colors[3]);                   //undefined
```

ArrayTypeExample04.htm

在此，虽然 colors 数组包含 3 个项，但把它的 length 属性设置成了 4。这个数组不存在位置 3，所以访问这个位置的值就得到了特殊值 undefined。

利用 length 属性也可以方便地在数组末尾添加新项，如下所示：

```
var colors = ["red", "blue", "green"];      // 创建一个包含 3 个字符串的数组
colors[colors.length] = "black";            //（在位置 3）添加一种颜色
colors[colors.length] = "brown";            //（在位置 4）再添加一种颜色
```

ArrayTypeExample05.htm

由于数组最后一项的索引始终是 length-1，因此下一个新项的位置就是 length。每当在数组末尾添加一项后，其 length 属性都会自动更新以反应这一变化。换句话说，上面例子第二行中的 colors[colors.length]为位置 3 添加了一个值，最后一行的 colors[colors.length]则为位置 4 添加了一个值。当把一个值放在超出当前数组大小的位置上时，数组就会重新计算其长度值，即长度值等于最后一项的索引加 1，如下面的例子所示：

```
var colors = ["red", "blue", "green"];      // 创建一个包含 3 个字符串的数组
colors[99] = "black";                       // （在位置 99）添加一种颜色
alert(colors.length); // 100
```

ArrayTypeExample06.htm

在这个例子中，我们向 colors 数组的位置 99 插入了一个值，结果数组新长度（length）就是 100（99+1）。而位置 3 到位置 98 实际上都是不存在的，所以访问它们都将返回 undefined。

数组最多可以包含 4 294 967 295 个项，这几乎已经能够满足任何编程需求了。如果想添加的项数超过这个上限值，就会发生异常。而创建一个初始大小与这个上限值接近的数组，则可能会导致运行时间超长的脚本错误。

5.2.1　检测数组

自从 ECMAScript 3 做出规定以后，就出现了确定某个对象是不是数组的经典问题。对于一个网页，或者一个全局作用域而言，使用 instanceof 操作符就能得到满意的结果：

```
if (value instanceof Array){
    //对数组执行某些操作
}
```

instanceof 操作符的问题在于，它假定单一的全局执行环境。如果网页中包含多个框架，那实际上就存在两个以上不同的全局执行环境，从而存在两个以上不同版本的 Array 构造函数。如果你从一个框架向另一个框架传入一个数组，那么传入的数组与在第二个框架中原生创建的数组分别具有各自不同的构造函数。

为了解决这个问题，ECMAScript 5 新增了 Array.isArray()方法。这个方法的目的是最终确定某个值到底是不是数组，而不管它是在哪个全局执行环境中创建的。这个方法的用法如下。

```
if (Array.isArray(value)){
    //对数组执行某些操作
}
```

支持 Array.isArray()方法的浏览器有 IE9+、Firefox 4+、Safari 5+、Opera 10.5+和 Chrome。要在尚未实现这个方法中的浏览器中准确检测数组，请参考 22.1.1 节。

5.2.2 转换方法

如前所述，所有对象都具有 toLocaleString()、toString()和 valueOf()方法。其中，调用数组的 toString()方法会返回由数组中每个值的字符串形式拼接而成的一个以逗号分隔的字符串。而调用 valueOf()返回的还是数组。实际上，为了创建这个字符串会调用数组每一项的 toString()方法。来看下面这个例子。

```
var colors = ["red", "blue", "green"];      // 创建一个包含 3 个字符串的数组
alert(colors.toString());      // red,blue,green
alert(colors.valueOf());       // red,blue,green
alert(colors);                 // red,blue,green
```

ArrayTypeExample07.htm

在这里，我们首先显式地调用了 toString()和 valueOf()方法，以便返回数组的字符串表示，每个值的字符串表示拼接成了一个字符串，中间以逗号分隔。最后一行代码直接将数组传递给了 alert()。由于 alert()要接收字符串参数，所以它会在后台调用 toString()方法，由此会得到与直接调用 toString()方法相同的结果。

另外，toLocaleString()方法经常也会返回与 toString()和 valueOf()方法相同的值，但也不总是如此。当调用数组的 toLocaleString()方法时，它也会创建一个数组值的以逗号分隔的字符串。而与前两个方法唯一的不同之处在于，这一次为了取得每一项的值，调用的是每一项的 toLocaleString()方法，而不是 toString()方法。请看下面这个例子。

```
var person1 = {
    toLocaleString : function () {
        return "Nikolaos";
    },

    toString : function() {
        return "Nicholas";
    }
};

var person2 = {
    toLocaleString : function () {
        return "Grigorios";
    },

    toString : function() {
        return "Greg";
    }
};

var people = [person1, person2];
alert(people);                          //Nicholas,Greg
alert(people.toString());               //Nicholas,Greg
alert(people.toLocaleString());         //Nikolaos,Grigorios
```

ArrayTypeExample08.htm

我们在这里定义了两个对象:person1 和 person2。而且还分别为每个对象定义了一个 toString()方法和一个 toLocaleString()方法，这两个方法返回不同的值。然后，创建一个包含前面定义的两个对象的数组。在将数组传递给 alert()时，输出结果是"Nicholas,Greg"，因为调用了数组每一项的 toString()方法（同样，这与下一行显式调用 toString()方法得到的结果相同）。而当调用数组的 toLocaleString()方法时，输出结果是"Nikolaos,Grigorios"，原因是调用了数组每一项的 toLocaleString()方法。

数组继承的 toLocaleString()、toString()和 valueOf()方法，在默认情况下都会以逗号分隔的字符串的形式返回数组项。而如果使用 join()方法，则可以使用不同的分隔符来构建这个字符串。join()方法只接收一个参数，即用作分隔符的字符串，然后返回包含所有数组项的字符串。请看下面的例子：

```
var colors = ["red", "green", "blue"];
alert(colors.join(","));        //red,green,blue
alert(colors.join("||"));       //red||green||blue
```

ArrayTypeJoinExample01.htm

在这里，我们使用 join()方法重现了 toString()方法的输出。在传递逗号的情况下，得到了以逗号分隔的数组值。而在最后一行代码中，我们传递了双竖线符号，结果就得到了字符串"red||green||blue"。如果不给 join()方法传入任何值，或者给它传入 undefined，则使用逗号作为分隔符。IE7 及更早版本会错误的使用字符串"undefined"作为分隔符。

如果数组中的某一项的值是 null 或者 undefined，那么该值在 join()、toLocale- String()、toString()和 valueOf()方法返回的结果中以空字符串表示。

5.2.3 栈方法

ECMAScript 数组也提供了一种让数组的行为类似于其他数据结构的方法。具体说来，数组可以表现得就像栈一样，后者是一种可以限制插入和删除项的数据结构。栈是一种 LIFO（Last-In-First-Out，后进先出）的数据结构，也就是最新添加的项最早被移除。而栈中项的插入（叫做**推入**）和移除（叫做**弹出**），只发生在一个位置——栈的顶部。ECMAScript 为数组专门提供了 push()和 pop()方法，以便实现类似栈的行为。

push()方法可以接收任意数量的参数，把它们逐个添加到数组末尾，并返回修改后数组的长度。而 pop()方法则从数组末尾移除最后一项，减少数组的 length 值，然后返回移除的项。请看下面的例子：

```
var colors = new Array();                    // 创建一个数组
var count = colors.push("red", "green");     // 推入两项
alert(count);    //2

count = colors.push("black");                // 推入另一项
alert(count);    //3

var item = colors.pop();                     // 取得最后一项
alert(item);     //"black"
alert(colors.length);   //2
```

ArrayTypeExample09.htm

以上代码中的数组可以看成是栈（代码本身没有任何区别，而 push()和 pop()都是数组默认的方法）。首先，我们使用 push()将两个字符串推入数组的末尾，并将返回的结果保存在变量 count 中（值为 2）。然后，再推入一个值，而结果仍然保存在 count 中。因为此时数组中包含 3 项，所以 push()返回 3。在调用 pop()时，它会返回数组的最后一项，即字符串"black"。此后，数组中仅剩两项。

可以将栈方法与其他数组方法连用，像下面这个例子一样。

```
var colors = ["red", "blue"];
colors.push("brown");                    // 添加另一项
colors[3] = "black";                     // 添加一项
alert(colors.length);       // 4

var item = colors.pop();                 // 取得最后一项
alert(item);  //"black"
```

ArrayTypeExample10.htm

在此，我们首先用两个值来初始化一个数组。然后，使用 push()添加第三个值，再通过直接在位置 3 上赋值来添加第四个值。而在调用 pop()时，该方法返回了字符串"black"，即最后一个添加到数组的值。

5.2.4 队列方法

栈数据结构的访问规则是 LIFO（后进先出），而队列数据结构的访问规则是 FIFO（First-In-First-Out，先进先出）。队列在列表的末端添加项，从列表的前端移除项。由于 push()是向数组末端添加项的方法，因此要模拟队列只需一个从数组前端取得项的方法。实现这一操作的数组方法就是 shift()，它能够移除数组中的第一个项并返回该项，同时将数组长度减 1。结合使用 shift()和 push()方法，可以像使用队列一样使用数组。

```
var colors = new Array();                    //创建一个数组
var count = colors.push("red", "green");     //推入两项
alert(count);     //2

count = colors.push("black");                //推入另一项
alert(count);      //3

var item = colors.shift();                   //取得第一项
alert(item);       //"red"
alert(colors.length); //2
```

ArrayTypeExample11.htm

这个例子首先使用 push()方法创建了一个包含 3 种颜色名称的数组。代码中加粗的那一行使用 shift()方法从数组中取得了第一项，即"red"。在移除第一项之后，"green"就变成了第一项，而"black"则变成了第二项，数组也只包含两项了。

ECMAScript 还为数组提供了一个 unshift()方法。顾名思义，unshift()与 shift()的用途相反：它能在数组前端添加任意个项并返回新数组的长度。因此，同时使用 unshift()和 pop()方法，可以从相反的方向来模拟队列，即在数组的前端添加项，从数组末端移除项，如下面的例子所示。

```
var colors = new Array();                    //创建一个数组
var count = colors.unshift("red", "green");  //推入两项
alert(count);     //2
```

```
count = colors.unshift("black");                    //推入另一项
alert(count);    //3

var item = colors.pop();                            //取得最后一项
alert(item);     //"green"
alert(colors.length); //2
```

ArrayTypeExample12.htm

这个例子创建了一个数组并使用 unshift()方法先后推入了 3 个值。首先是"red"和"green"，然后是"black"，数组中各项的顺序为"black"、"red"、"green"。在调用 pop()方法时，移除并返回的是最后一项，即"green"。

IE7 及更早版本对 JavaScript 的实现中存在一个偏差，其 unshift()方法总是返回 undefined 而不是数组的新长度。IE8 在非兼容模式下会返回正确的长度值。

5.2.5　重排序方法

数组中已经存在两个可以直接用来重排序的方法：reverse()和 sort()。有读者可能猜到了，reverse()方法会对反转数组项的顺序。请看下面这个例子。

```
var values = [1, 2, 3, 4, 5];
values.reverse();
alert(values);          //5,4,3,2,1
```

ArrayTypeExample13.htm

这里数组的初始值及顺序是 1、2、3、4、5。而调用数组的 reverse()方法后，其值的顺序变成了 5、4、3、2、1。这个方法的作用相当直观明了，但不够灵活，因此才有了 sort()方法。

在默认情况下，sort()方法按升序排列数组项——即最小的值位于最前面，最大的值排在最后面。为了实现排序，sort()方法会调用每个数组项的 toString()转型方法，然后比较得到的字符串，以确定如何排序。即使数组中的每一项都是数值，sort()方法比较的也是字符串，如下所示。

```
var values = [0, 1, 5, 10, 15];
values.sort();
alert(values);      //0,1,10,15,5
```

ArrayTypeExample14.htm

可见，即使例子中值的顺序没有问题，但 sort()方法也会根据测试字符串的结果改变原来的顺序。因为数值 5 虽然小于 10，但在进行字符串比较时，"10"则位于"5"的前面，于是数组的顺序就被修改了。不用说，这种排序方式在很多情况下都不是最佳方案。因此 sort()方法可以接收一个比较函数作为参数，以便我们指定哪个值位于哪个值的前面。

比较函数接收两个参数，如果第一个参数应该位于第二个之前则返回一个负数，如果两个参数相等则返回 0，如果第一个参数应该位于第二个之后则返回一个正数。以下就是一个简单的比较函数：

```
function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
}
```

ArrayTypeExample15.htm

这个比较函数可以适用于大多数数据类型，只要将其作为参数传递给 sort()方法即可，如下面这个例子所示。

```
var values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);   //0,1,5,10,15
```

在将比较函数传递到 sort()方法之后，数值仍然保持了正确的升序。当然，也可以通过比较函数产生降序排序的结果，只要交换比较函数返回的值即可。


```
function compare(value1, value2) {
    if (value1 < value2) {
        return 1;
    } else if (value1 > value2) {
        return -1;
    } else {
        return 0;
    }
}

var values = [0, 1, 5, 10, 15];
values.sort(compare);
alert(values);    // 15,10,5,1,0
```

ArrayTypeExample16.htm

在这个修改后的例子中，比较函数在第一个值应该位于第二个之后的情况下返回 1，而在第一个值应该在第二个之前的情况下返回-1。交换返回值的意思是让更大的值排位更靠前，也就是对数组按照降序排序。当然，如果只想反转数组原来的顺序，使用 reverse()方法要更快一些。

reverse()和 sort()方法的返回值是经过排序之后的数组。

对于数值类型或者其 valueOf()方法会返回数值类型的对象类型，可以使用一个更简单的比较函数。这个函数只要用第二个值减第一个值即可[①]。

```
function compare(value1, value2){
    return value2 - value1;
}
```

① 如果想要按照升级排序，则 compare()函数中的 return 语句应该返回 value2-value1。

由于比较函数通过返回一个小于零、等于零或大于零的值来影响排序结果，因此减法操作就可以适当地处理所有这些情况。

5.2.6　操作方法

ECMAScript 为操作已经包含在数组中的项提供了很多方法。其中，concat()方法可以基于当前数组中的所有项创建一个新数组。具体来说，这个方法会先创建当前数组一个副本，然后将接收到的参数添加到这个副本的末尾，最后返回新构建的数组。在没有给 concat()方法传递参数的情况下，它只是复制当前数组并返回副本。如果传递给 concat()方法的是一或多个数组，则该方法会将这些数组中的每一项都添加到结果数组中。如果传递的值不是数组，这些值就会被简单地添加到结果数组的末尾。下面来看一个例子。

```
var colors = ["red", "green", "blue"];
var colors2 = colors.concat("yellow", ["black", "brown"]);

alert(colors);      //red,green,blue
alert(colors2);     //red,green,blue,yellow,black,brown
```

ArrayTypeConcatExample01.htm

以上代码开始定义了一个包含 3 个值的数组 colors。然后，基于 colors 调用了 concat()方法，并传入字符串"yellow"和一个包含"black"和"brown"的数组。最终，结果数组 colors2 中包含了"red"、"green"、"blue"、"yellow"、"black"和"brown"。至于原来的数组 colors，其值仍然保持不变。

下一个方法是 slice()，它能够基于当前数组中的一或多个项创建一个新数组。slice()方法可以接受一或两个参数，即要返回项的起始和结束位置。在只有一个参数的情况下，slice()方法返回从该参数指定位置开始到当前数组末尾的所有项。如果有两个参数，该方法返回起始和结束位置之间的项——但不包括结束位置的项。注意，slice()方法不会影响原始数组。请看下面的例子。

```
var colors = ["red", "green", "blue", "yellow", "purple"];
var colors2 = colors.slice(1);
var colors3 = colors.slice(1,4);

alert(colors2);    //green,blue,yellow,purple
alert(colors3);    //green,blue,yellow
```

ArrayTypeSliceExample01.htm

在这个例子中，开始定义的数组 colors 包含 5 项。调用 slice()并传入 1 会得到一个包含 4 项的新数组；因为是从位置 1 开始复制，所以会包含"green"而不会包含"red"。这个新数组 colors2 中包含的是"green"、"blue"、"yellow"和"purple"。接着，我们再次调用 slice()并传入了 1 和 4，表示复制从位置 1 开始，到位置 3 结束。结果数组 colors3 中包含了"green"、"blue"和"yellow"。

如果 slice()方法的参数中有一个负数，则用数组长度加上该数来确定相应的位置。例如，在一个包含 5 项的数组上调用 slice(-2,-1)与调用 slice(3,4)得到的结果相同。如果结束位置小于起始位置，则返回空数组。

下面我们来介绍 splice()方法，这个方法恐怕要算是最强大的数组方法了，它有很多种用法。splice()的主要用途是向数组的中部插入项，但使用这种方法的方式则有如下 3 种。

- **删除**：可以删除任意数量的项，只需指定 2 个参数：要删除的第一项的位置和要删除的项数。例如，splice(0,2)会删除数组中的前两项。
- **插入**：可以向指定位置插入任意数量的项，只需提供 3 个参数：起始位置、0（要删除的项数）和要插入的项。如果要插入多个项，可以再传入第四、第五，以至任意多个项。例如，splice(2,0,"red","green")会从当前数组的位置 2 开始插入字符串"red"和"green"。
- **替换**：可以向指定位置插入任意数量的项，且同时删除任意数量的项，只需指定 3 个参数：起始位置、要删除的项数和要插入的任意数量的项。插入的项数不必与删除的项数相等。例如，splice (2,1,"red","green")会删除当前数组位置 2 的项，然后再从位置 2 开始插入字符串"red"和"green"。

splice()方法始终都会返回一个数组，该数组中包含从原始数组中删除的项（如果没有删除任何项，则返回一个空数组）。下面的代码展示了上述 3 种使用 splice()方法的方式。


```
var colors = ["red", "green", "blue"];
var removed = colors.splice(0,1);                    // 删除第一项
alert(colors);      // green,blue
alert(removed);     // red, 返回的数组中只包含一项

removed = colors.splice(1, 0, "yellow", "orange");   // 从位置 1 开始插入两项
alert(colors);      // green,yellow,orange,blue
alert(removed);     // 返回的是一个空数组

removed = colors.splice(1, 1, "red", "purple");      // 插入两项，删除一项
alert(colors);      // green,red,purple,orange,blue
alert(removed);     // yellow, 返回的数组中只包含一项
```

ArrayTypeSpliceExample01.htm

上面的例子首先定义了一个包含 3 项的数组 colors。第一次调用 splice()方法只是删除了这个数组的第一项，之后 colors 还包含"green"和"blue"两项。第二次调用 splice()方法时在位置 1 插入了两项，结果 colors 中包含"green"、"yellow"、"orange"和"blue"。这一次操作没有删除项，因此返回了一个空数组。最后一次调用 splice()方法删除了位置 1 处的一项，然后又插入了"red"和"purple"。在完成以上操作之后，数组 colors 中包含的是"green"、"red"、"purple"、"orange"和"blue"。

5.2.7 位置方法

ECMAScript 5 为数组实例添加了两个位置方法：indexOf()和 lastIndexOf()。这两个方法都接收两个参数：要查找的项和（可选的）表示查找起点位置的索引。其中，indexOf()方法从数组的开头（位置 0）开始向后查找，lastIndexOf()方法则从数组的末尾开始向前查找。

这两个方法都返回要查找的项在数组中的位置，或者在没找到的情况下返回-1。在比较第一个参数与数组中的每一项时，会使用全等操作符；也就是说，要求查找的项必须严格相等（就像使用===一样）。以下是几个例子。

```
var numbers = [1,2,3,4,5,4,3,2,1];

alert(numbers.indexOf(4));          //3
```

```
alert(numbers.lastIndexOf(4));     //5

alert(numbers.indexOf(4, 4));      //5
alert(numbers.lastIndexOf(4, 4)); //3

var person = { name: "Nicholas" };
var people = [{ name: "Nicholas" }];

var morePeople = [person];

alert(people.indexOf(person));      //-1
alert(morePeople.indexOf(person)); //0
```

ArrayIndexOfExample01.htm

使用 indexOf()和 lastIndexOf()方法查找特定项在数组中的位置非常简单，支持它们的浏览器包括 IE9+、Firefox 2+、Safari 3+、Opera 9.5+和 Chrome。

5.2.8　迭代方法

ECMAScript 5 为数组定义了 5 个迭代方法。每个方法都接收两个参数：要在每一项上运行的函数和（可选的）运行该函数的作用域对象——影响 this 的值。传入这些方法中的函数会接收三个参数：数组项的值、该项在数组中的位置和数组对象本身。根据使用的方法不同，这个函数执行后的返回值可能会也可能不会影响访问的返回值。以下是这 5 个迭代方法的作用。

- every()：对数组中的每一项运行给定函数，如果该函数对每一项都返回 true，则返回 true。
- filter()：对数组中的每一项运行给定函数，返回该函数会返回 true 的项组成的数组。
- forEach()：对数组中的每一项运行给定函数。这个方法没有返回值。
- map()：对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组。
- some()：对数组中的每一项运行给定函数，如果该函数对任一项返回 true，则返回 true。

以上方法都不会修改数组中的包含的值。

在这些方法中，最相似的是 every()和 some()，它们都用于查询数组中的项是否满足某个条件。对 every()来说，传入的函数必须对每一项都返回 true，这个方法才返回 true；否则，它就返回 false。而 some()方法则是只要传入的函数对数组中的某一项返回 true，就会返回 true。请看以下例子。

```
var numbers = [1,2,3,4,5,4,3,2,1];

var everyResult = numbers.every(function(item, index, array){
    return (item > 2);
});

alert(everyResult);      //false

var someResult = numbers.some(function(item, index, array){
    return (item > 2);
});

alert(someResult);       //true
```

ArrayEveryAndSomeExample01.htm

以上代码调用了 every()和 some(),传入的函数只要给定项大于 2 就会返回 true。对于 every(),它返回的是 false，因为只有部分数组项符合条件。对于 some()，结果就是 true，因为至少有一项是大于 2 的。

下面再看一看 filter()函数，它利用指定的函数确定是否在返回的数组中包含的某一项。例如，要返回一个所有数值都大于 2 的数组，可以使用以下代码。

```
var numbers = [1,2,3,4,5,4,3,2,1];

var filterResult = numbers.filter(function(item, index, array){
    return (item > 2);
});

alert(filterResult);        //[3,4,5,4,3]
```

ArrayFilterExample01.htm

这里，通过调用 filter()方法创建并返回了包含 3、4、5、4、3 的数组，因为传入的函数对它们每一项都返回 true。这个方法对查询符合某些条件的所有数组项非常有用。

map()也返回一个数组，而这个数组的每一项都是在原始数组中的对应项上运行传入函数的结果。例如，可以给数组中的每一项乘以 2，然后返回这些乘积组成的数组，如下所示。

```
var numbers = [1,2,3,4,5,4,3,2,1];

var mapResult = numbers.map(function(item, index, array){
    return item * 2;
});

alert(mapResult);  //[2,4,6,8,10,8,6,4,2]
```

ArrayMapExample01.htm

以上代码返回的数组中包含给每个数乘以 2 之后的结果。这个方法适合创建包含的项与另一个数组一一对应的数组。

最后一个方法是 forEach()，它只是对数组中的每一项运行传入的函数。这个方法没有返回值，本质上与使用 for 循环迭代数组一样。来看一个例子。

```
var numbers = [1,2,3,4,5,4,3,2,1];

numbers.forEach(function(item, index, array){
    //执行某些操作
});
```

这些数组方法通过执行不同的操作，可以大大方便处理数组的任务。支持这些迭代方法的浏览器有 IE9+、Firefox 2+、Safari 3+、Opera 9.5+和 Chrome。

5.2.9 缩小方法

ECMAScript 5 还新增了两个缩小数组的方法：reduce()和 reduceRight()。这两个方法都会迭代数组的所有项，然后构建一个最终返回的值。其中，reduce()方法从数组的第一项开始，逐个遍历到最后。而 reduceRight()则从数组的最后一项开始，向前遍历到第一项。

这两个方法都接收两个参数：一个在每一项上调用的函数和（可选的）作为缩小基础的初始值。传

给 reduce()和 reduceRight()的函数接收 4 个参数：前一个值、当前值、项的索引和数组对象。这个函数返回的任何值都会作为第一个参数自动传给下一项。第一次迭代发生在数组的第二项上，因此第一个参数是数组的第一项，第二个参数就是数组的第二项。

使用 reduce()方法可以执行求数组中所有值之和的操作，比如：

```
var values = [1,2,3,4,5];
var sum = values.reduce(function(prev, cur, index, array){
    return prev + cur;
});
alert(sum); //15
```

ArrayReductionExample01.htm

第一次执行回调函数，prev 是 1，cur 是 2。第二次，prev 是 3（1 加 2 的结果），cur 是 3（数组的第三项）。这个过程会持续到把数组中的每一项都访问一遍，最后返回结果。

reduceRight()的作用类似，只不过方向相反而已。来看下面这个例子。

```
var values = [1,2,3,4,5];
var sum = values.reduceRight(function(prev, cur, index, array){
    return prev + cur;
});
alert(sum); //15
```

在这个例子中，第一次执行回调函数，prev 是 5，cur 是 4。当然，最终结果相同，因为执行的都是简单相加的操作。

使用 reduce()还是 reduceRight()，主要取决于要从哪头开始遍历数组。除此之外，它们完全相同。

支持这两个缩小函数的浏览器有 IE9+、Firefox 3+、Safari 4+、Opera 10.5 和 Chrome。

5.3　Date 类型

ECMAScript 中的 Date 类型是在早期 Java 中的 java.util.Date 类基础上构建的。为此，Date 类型使用自 UTC（Coordinated Universal Time，国际协调时间）1970 年 1 月 1 日午夜（零时）开始经过的毫秒数来保存日期。在使用这种数据存储格式的条件下，Date 类型保存的日期能够精确到 1970 年 1 月 1 日之前或之后的 285 616 年。

要创建一个日期对象，使用 new 操作符和 Date 构造函数即可，如下所示。

```
var now = new Date();
```

DateTypeExample01.htm

在调用 Date 构造函数而不传递参数的情况下，新创建的对象自动获得当前日期和时间。如果想根据特定的日期和时间创建日期对象，必须传入表示该日期的毫秒数（即从 UTC 时间 1970 年 1 月 1 日午夜起至该日期止经过的毫秒数）。为了简化这一计算过程，ECMAScript 提供了两个方法：Date.parse()和 Date.UTC()。

其中，Date.parse()方法接收一个表示日期的字符串参数，然后尝试根据这个字符串返回相应日期的毫秒数。ECMA-262 没有定义 Date.parse()应该支持哪种日期格式，因此这个方法的行为因实现而异，而且通常是因地区而异。将地区设置为美国的浏览器通常都接受下列日期格式：

- “月/日/年”，如 6/13/2004；
- “英文月名 日,年”，如 January 12,2004；
- “英文星期几 英文月名 日 年 时:分:秒 时区”，如 Tue May 25 2004 00:00:00 GMT-0700。
- ISO 8601 扩展格式 YYYY-MM-DDTHH:mm:ss.sssZ（例如 2004-05-25T00:00:00）。只有兼容 ECMAScript 5 的实现支持这种格式。

例如，要为 2004 年 5 月 25 日创建一个日期对象，可以使用下面的代码：

```
var someDate = new Date(Date.parse("May 25, 2004"));
```

DateTypeExample01.htm

如果传入 Date.parse()方法的字符串不能表示日期，那么它会返回 NaN。实际上，如果直接将表示日期的字符串传递给 Date 构造函数，也会在后台调用 Date.parse()。换句话说，下面的代码与前面的例子是等价的：

```
var someDate = new Date("May 25, 2004");
```

这行代码将会得到与前面相同的日期对象。

日期对象及其在不同浏览器中的实现有许多奇怪的行为。其中有一种倾向是将超出范围的值替换成当前的值，以便生成输出。例如，在解析"January 32, 2007"时，有的浏览器会将其解释为"February 1, 2007"。而 Opera 则倾向于插入当前月份的当前日期，返回"January 当前日期, 2007"。也就是说，如果在 2007 年 9 月 21 日运行前面的代码，将会得到"January 21, 2007"（都是 21 日）。

Date.UTC()方法同样也返回表示日期的毫秒数，但它与 Date.parse()在构建值时使用不同的信息。Date.UTC()的参数分别是年份、基于 0 的月份（一月是 0，二月是 1，以此类推）、月中的哪一天（1 到 31）、小时数（0 到 23）、分钟、秒以及毫秒数。在这些参数中，只有前两个参数（年和月）是必需的。如果没有提供月中的天数，则假设天数为 1；如果省略其他参数，则统统假设为 0。以下是两个使用 Date.UTC()方法的例子：

```
// GMT 时间 2000 年 1 月 1 日午夜零时
var y2k = new Date(Date.UTC(2000, 0));

// GMT 时间 2005 年 5 月 5 日下午 5:55:55
var allFives = new Date(Date.UTC(2005, 4, 5, 17, 55, 55));
```

DateTypeUTCExample01.htm

这个例子创建了两个日期对象。第一个对象表示 GMT 时间 2000 年 1 月 1 日午夜零时，传入的值一个是表示年份的 2000，一个是表示月份的 0（即一月份）。因为其他参数是自动填充的（即月中的天数为 1，其他所有参数均为 0），所以结果就是该月第一天的午夜零时。第二个对象表示 GMT 时间 2005 年 5 月 5 日下午 5:55:55，即使日期和时间中只包含 5，也需要传入不一样的参数：月份必须是 4（因为月份是基于 0 的）、小时必须设置为 17（因为小时以 0 到 23 表示），剩下的参数就很直观了。

如同模仿 Date.parse()一样，Date 构造函数也会模仿 Date.UTC()，但有一点明显不同：日期和时间都基于本地时区而非 GMT 来创建。不过，Date 构造函数接收的参数仍然与 Date.UTC()相同。

因此，如果第一个参数是数值，Date 构造函数就会假设该值是日期中的年份，而第二个参数是月份，以此类推。据此，可以将前面的例子重写如下。

```
// 本地时间 2000 年 1 月 1 日午夜零时
var y2k = new Date(2000, 0);

// 本地时间 2005 年 5 月 5 日下午 5:55:55
var allFives = new Date(2005, 4, 5, 17, 55, 55);
```

DateTypeConstructorExample01.htm

以上代码创建了与前面例子中相同的两个日期对象，只不过这次的日期都是基于系统设置的本地时区创建的。

ECMAScript 5 添加了 Data.now()方法，返回表示调用这个方法时的日期和时间的毫秒数。这个方法简化了使用 Data 对象分析代码的工作。例如：

```
//取得开始时间
var start = Date.now();

//调用函数
doSomething();

//取得停止时间
var stop = Date.now(),
    result = stop - start;
```

支持 Data.now()方法的浏览器包括 IE9+、Firefox 3+、Safari 3+、Opera 10.5 和 Chrome。在不支持它的浏览器中，使用+操作符把 Data 对象转换成字符串，也可以达到同样的目的。

```
//取得开始时间
var start = +new Date();

//调用函数
doSomething();
//取得停止时间
var stop = +new Date(),
    result = stop    start;
```

5.3.1　继承的方法

与其他引用类型一样，Date 类型也重写了 toLocaleString()、toString()和 valueOf()方法；但这些方法返回的值与其他类型中的方法不同。Date 类型的 toLocaleString()方法会按照与浏览器设置的地区相适应的格式返回日期和时间。这大致意味着时间格式中会包含 AM 或 PM，但不会包含时区信息（当然，具体的格式会因浏览器而异）。而 toString()方法则通常返回带有时区信息的日期和时间，其中时间一般以军用时间（即小时的范围是 0 到 23）表示。下面给出了在不同浏览器中调用 toLocaleString()和 toString()方法，输出 PST（Pacific Standard Time，太平洋标准时间）时间 2007 年 2 月 1 日午夜零时的结果。

Internet Explorer 8

```
toLocaleString() — Thursday, February 01, 2007 12:00:00 AM
toString() — Thu Feb 1 00:00:00 PST 2007
```

Firefox 3.5

```
toLocaleString() — Thursday, February 01, 2007 12:00:00 AM
toString() — Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

Safari 4

```
toLocaleString() — Thursday, February 01, 2007 00:00:00
toString() — Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

Chrome 4

```
toLocaleString() — Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
toString() — Thu Feb 01 2007 00:00:00 GMT-0800 (Pacific Standard Time)
```

Opera 10

```
toLocaleString() — 2/1/2007 12:00:00 AM
toString() — Thu, 01 Feb 2007 00:00:00 GMT-0800
```

显然，这两个方法在不同的浏览器中返回的日期和时间格式可谓大相径庭。事实上，toLocaleString()和toString()的这一差别仅在调试代码时比较有用，而在显示日期和时间时没有什么价值。

至于Date类型的valueOf()方法，则根本不返回字符串，而是返回日期的毫秒表示。因此，可以方便使用比较操作符（小于或大于）来比较日期值。请看下面的例子。

```
var date1 = new Date(2007, 0, 1);          //"January 1, 2007"
var date2 = new Date(2007, 1, 1);          //"February 1, 2007"

alert(date1 < date2); //true
alert(date1 > date2); //false
```

DateTypeValueOfExample01.htm

从逻辑上讲，2007年1月1日要早于2007年2月1日，此时如果我们说前者小于后者比较符合常理。而表示2007年1月1日的毫秒值小于表示2007年2月1日的毫秒值，因此在首先使用小于操作符比较日期时，返回的结果是true。这样，就为我们比较日期提供了极大方便。

5.3.2 日期格式化方法

Date类型还有一些专门用于将日期格式化为字符串的方法，这些方法如下。

- toDateString()——以特定于实现的格式显示星期几、月、日和年；
- toTimeString()——以特定于实现的格式显示时、分、秒和时区；
- toLocaleDateString()——以特定于地区的格式显示星期几、月、日和年；
- toLocaleTimeString()——以特定于实现的格式显示时、分、秒；
- toUTCString()——以特定于实现的格式完整的UTC日期。

与toLocaleString()和toString()方法一样，以上这些字符串格式方法的输出也是因浏览器而异的，因此没有哪一个方法能够用来在用户界面中显示一致的日期信息。

除了前面介绍的方法之外，还有一个名叫toGMTString()的方法，这是一个与toUTCString()等价的方法，其存在目的在于确保向后兼容。不过，ECMAScript推荐现在编写的代码一律使用toUTCString()方法。

5.3.3　日期/时间组件方法

到目前为止，剩下还未介绍的 Date 类型的方法（如下表所示），都是直接取得和设置日期值中特定部分的方法了。需要注意的是，UTC 日期指的是在没有时区偏差的情况下（将日期转换为 GMT 时间）的日期值。

方　法	说　明
getTime()	返回表示日期的毫秒数；与valueOf()方法返回的值相同
setTime(毫秒)	以毫秒数设置日期，会改变整个日期
getFullYear()	取得4位数的年份（如2007而非仅07）
getUTCFullYear()	返回UTC日期的4位数年份
setFullYear(年)	设置日期的年份。传入的年份值必须是4位数字（如2007而非仅07）
setUTCFullYear(年)	设置UTC日期的年份。传入的年份值必须是4位数字（如2007而非仅07）
getMonth()	返回日期中的月份，其中0表示一月，11表示十二月
getUTCMonth()	返回UTC日期中的月份，其中0表示一月，11表示十二月
setMonth(月)	设置日期的月份。传入的月份值必须大于0，超过11则增加年份
setUTCMonth(月)	设置UTC日期的月份。传入的月份值必须大于0，超过11则增加年份
getDate()	返回日期月份中的天数（1到31）
getUTCDate()	返回UTC日期月份中的天数（1到31）
setDate(日)	设置日期月份中的天数。如果传入的值超过了该月中应有的天数，则增加月份
setUTCDate(日)	设置UTC日期月份中的天数。如果传入的值超过了该月中应有的天数，则增加月份
getDay()	返回日期中星期的星期几（其中0表示星期日，6表示星期六）
getUTCDay()	返回UTC日期中星期的星期几（其中0表示星期日，6表示星期六）
getHours()	返回日期中的小时数（0到23）
getUTCHours()	返回UTC日期中的小时数（0到23）
setHours(时)	设置日期中的小时数。传入的值超过了23则增加月份中的天数
setUTCHours(时)	设置UTC日期中的小时数。传入的值超过了23则增加月份中的天数
getMinutes()	返回日期中的分钟数（0到59）
getUTCMinutes()	返回UTC日期中的分钟数（0到59）
setMinutes(分)	设置日期中的分钟数。传入的值超过59则增加小时数
setUTCMinutes(分)	设置UTC日期中的分钟数。传入的值超过59则增加小时数
getSeconds()	返回日期中的秒数（0到59）
getUTCSeconds()	返回UTC日期中的秒数（0到59）
setSeconds(秒)	设置日期中的秒数。传入的值超过了59会增加分钟数
setUTCSeconds(秒)	设置UTC日期中的秒数。传入的值超过了59会增加分钟数
getMilliseconds()	返回日期中的毫秒数
getUTCMilliseconds()	返回UTC日期中的毫秒数
setMilliseconds(毫秒)	设置日期中的毫秒数

（续）

方　法	说　明
setUTCMilliseconds(毫秒)	设置UTC日期中的毫秒数
getTimezoneOffset()	返回本地时间与UTC时间相差的分钟数。例如，美国东部标准时间返回300。在某地进入夏令时的情况下，这个值会有所变化

5.4 RegExp 类型

ECMAScript 通过 RegExp 类型来支持正则表达式。使用下面类似 Perl 的语法，就可以创建一个正则表达式。

```
var expression = / pattern / flags ;
```

其中的模式（pattern）部分可以是任何简单或复杂的正则表达式，可以包含字符类、限定符、分组、向前查找以及反向引用。每个正则表达式都可带有一或多个标志（flags），用以标明正则表达式的行为。正则表达式的匹配模式支持下列 3 个标志。

- g：表示全局（global）模式，即模式将被应用于所有字符串，而非在发现第一个匹配项时立即停止；
- i：表示不区分大小写（case-insensitive）模式，即在确定匹配项时忽略模式与字符串的大小写；
- m：表示多行（multiline）模式，即在到达一行文本末尾时还会继续查找下一行中是否存在与模式匹配的项。

因此，一个正则表达式就是一个模式与上述 3 个标志的组合体。不同组合产生不同结果，如下面的例子所示。

```
/*
 * 匹配字符串中所有"at"的实例
 */
var pattern1 = /at/g;

/*
 * 匹配第一个"bat"或"cat"，不区分大小写
 */
var pattern2 = /[bc]at/i;

/*
 * 匹配所有以"at"结尾的 3 个字符的组合，不区分大小写
 */
var pattern3 = /.at/gi;
```

与其他语言中的正则表达式类似，模式中使用的所有元字符都必须转义。正则表达式中的元字符包括：

```
( [ { \ ^ $ | ) ? * + .]}
```

这些元字符在正则表达式中都有一或多种特殊用途，因此如果想要匹配字符串中包含的这些字符，就必须对它们进行转义。下面给出几个例子。

```
/*
 * 匹配第一个"bat"或"cat"，不区分大小写
 */
```

```
var pattern1 = /[bc]at/i;

/*
 * 匹配第一个" [bc]at", 不区分大小写
 */
var pattern2 = /\[bc\]at/i;

/*
 * 匹配所有以"at"结尾的 3 个字符的组合, 不区分大小写
 */
var pattern3 = /.at/gi;

/*
 * 匹配所有".at", 不区分大小写
 */
var pattern4 = /\.at/gi;
```

在上面的例子中，pattern1 匹配第一个"bat"或"cat"，不区分大小写。而要想直接匹配"[bc]at"的话，就需要像定义 pattern2 一样，对其中的两个方括号进行转义。对于 pattern3 来说，句点表示位于"at"之前的任意一个可以构成匹配项的字符。但如果想匹配".at"，则必须对句点本身进行转义，如 pattern4 所示。

前面举的这些例子都是以字面量形式来定义的正则表达式。另一种创建正则表达式的方式是使用 RegExp 构造函数，它接收两个参数：一个是要匹配的字符串模式，另一个是可选的标志字符串。可以使用字面量定义的任何表达式，都可以使用构造函数来定义，如下面的例子所示。

```
/*
 * 匹配第一个"bat"或"cat", 不区分大小写
 */
var pattern1 = /[bc]at/i;

/*
 * 与 pattern1 相同, 只不过是使用构造函数创建的
 */
var pattern2 = new RegExp("[bc]at", "i");
```

在此，pattern1 和 pattern2 是两个完全等价的正则表达式。要注意的是，传递给 RegExp 构造函数的两个参数都是字符串（不能把正则表达式字面量传递给 RegExp 构造函数）。由于 RegExp 构造函数的模式参数是字符串，所以在某些情况下要对字符进行双重转义。所有元字符都必须双重转义，那些已经转义过的字符也是如此，例如\n（字符\在字符串中通常被转义为\\，而在正则表达式字符串中就会变成\\\\）。下表给出了一些模式，左边是这些模式的字面量形式，右边是使用 RegExp 构造函数定义相同模式时使用的字符串。

字面量模式	等价的字符串
/\[bc\]at/	"\\[bc\\]at"
/\.at/	"\\.at"
/name\/age/	"name\\/age"
/\d.\d{1,2}/	"\\d.\\d{1,2}"
/\w\\hello\\123/	"\\w\\\\hello\\\\123"

使用正则表达式字面量和使用 RegExp 构造函数创建的正则表达式不一样。在 ECMAScript 3 中，正则表达式字面量始终会共享同一个 RegExp 实例，而使用构造函数创建的每一个新 RegExp 实例都是

一个新实例。来看下面的例子。

```
var re = null,
    i;

for (i=0; i < 10; i++){
    re = /cat/g;
    re.test("catastrophe");
}

for (i=0; i < 10; i++){
    re = new RegExp("cat", "g");
    re.test("catastrophe");
}
```

在第一个循环中，即使是循环体中指定的，但实际上只为/cat/创建了一个 RegExp 实例。由于实例属性（下一节介绍实例属性）不会重置，所以在循环中再次调用 test()方法会失败。这是因为第一次调用 test()找到了"cat"，但第二次调用是从索引为 3 的字符（上一次匹配的末尾）开始的，所以就找不到它了。由于会测试到字符串末尾，所以下一次再调用 test()就又从开头开始了。

第二个循环使用 RegExp 构造函数在每次循环中创建正则表达式。因为每次迭代都会创建一个新的 RegExp 实例，所以每次调用 test()都会返回 true。

ECMAScript 5 明确规定，使用正则表达式字面量必须像直接调用 RegExp 构造函数一样，每次都创建新的 RegExp 实例。IE9+、Firefox 4+和 Chrome 都据此做出了修改。

5.4.1 RegExp 实例属性

RegExp 的每个实例都具有下列属性，通过这些属性可以取得有关模式的各种信息。

- **global**：布尔值，表示是否设置了 g 标志。
- **ignoreCase**：布尔值，表示是否设置了 i 标志。
- **lastIndex**：整数，表示开始搜索下一个匹配项的字符位置，从 0 算起。
- **multiline**：布尔值，表示是否设置了 m 标志。
- **source**：正则表达式的字符串表示，按照字面量形式而非传入构造函数中的字符串模式返回。

通过这些属性可以获知一个正则表达式的各方面信息，但却没有多大用处，因为这些信息全都包含在模式声明中。例如：

```
var pattern1 = /\[bc\]at/i;

alert(pattern1.global);         //false
alert(pattern1.ignoreCase);     //true
alert(pattern1.multiline);      //false
alert(pattern1.lastIndex);      //0
alert(pattern1.source);         //"\[bc\]at"

var pattern2 = new RegExp("\\[bc\\]at", "i");

alert(pattern2.global);         //false
alert(pattern2.ignoreCase);     //true
alert(pattern2.multiline);      //false
alert(pattern2.lastIndex);      //0
alert(pattern2.source);         //"\[bc\]at"
```

RegExpInstancePropertiesExample01.htm

我们注意到，尽管第一个模式使用的是字面量，第二个模式使用了 RegExp 构造函数，但它们的 source 属性是相同的。可见，source 属性保存的是规范形式的字符串，即字面量形式所用的字符串。

5.4.2 RegExp 实例方法

RegExp 对象的主要方法是 exec()，该方法是专门为捕获组而设计的。exec()接受一个参数，即要应用模式的字符串，然后返回包含第一个匹配项信息的数组；或者在没有匹配项的情况下返回 null。返回的数组虽然是 Array 的实例，但包含两个额外的属性：index 和 input。其中，index 表示匹配项在字符串中的位置，而 input 表示应用正则表达式的字符串。在数组中，第一项是与整个模式匹配的字符串，其他项是与模式中的捕获组匹配的字符串（如果模式中没有捕获组，则该数组只包含一项）。请看下面的例子。

```
var text = "mom and dad and baby";
var pattern = /mom( and dad( and baby)?)?/gi;

var matches = pattern.exec(text);
alert(matches.index);      // 0
alert(matches.input);      // "mom and dad and baby"
alert(matches[0]);         // "mom and dad and baby"
alert(matches[1]);         // " and dad and baby"
alert(matches[2]);         // " and baby"
```

RegExpExecExample01.htm

这个例子中的模式包含两个捕获组。最内部的捕获组匹配"and baby"，而包含它的捕获组匹配"and dad"或者"and dad and baby"。当把字符串传入 exec()方法中之后，发现了一个匹配项。因为整个字符串本身与模式匹配，所以返回的数组 matchs 的 index 属性值为 0。数组中的第一项是匹配的整个字符串，第二项包含与第一个捕获组匹配的内容，第三项包含与第二个捕获组匹配的内容。

对于 exec()方法而言，即使在模式中设置了全局标志（g），它每次也只会返回一个匹配项。在不设置全局标志的情况下，在同一个字符串上多次调用 exec()将始终返回第一个匹配项的信息。而在设置全局标志的情况下，每次调用 exec()则都会在字符串中继续查找新匹配项，如下面的例子所示。

```
var text = "cat, bat, sat, fat";
var pattern1 = /.at/;

var matches = pattern1.exec(text);
alert(matches.index);         //0
alert(matches[0]);            //cat
alert(pattern1.lastIndex);    //0

matches = pattern1.exec(text);
alert(matches.index);         //0
alert(matches[0]);            //cat
alert(pattern1.lastIndex);    //0

var pattern2 = /.at/g;

var matches = pattern2.exec(text);
alert(matches.index);         //0
alert(matches[0]);            //cat
alert(pattern2.lastIndex);    //0
```

```
matches = pattern2.exec(text);
alert(matches.index);          //5
alert(matches[0]);             //bat
alert(pattern2.lastIndex);     //8
```

RegExpExecExample02.htm

这个例子中的第一个模式 pattern1 不是全局模式，因此每次调用 exec()返回的都是第一个匹配项（"cat"）。而第二个模式 pattern2 是全局模式，因此每次调用 exec()都会返回字符串中的下一个匹配项，直至搜索到字符串末尾为止。此外，还应该注意模式的 lastIndex 属性的变化情况。在全局匹配模式下，lastIndex 的值在每次调用 exec()后都会增加，而在非全局模式下则始终保持不变。

IE 的 JavaScript 实现在 lastIndex 属性上存在偏差，即使在非全局模式下，lastIndex 属性每次也会变化。

正则表达式的第二个方法是 test()，它接受一个字符串参数。在模式与该参数匹配的情况下返回 true；否则，返回 false。在只想知道目标字符串与某个模式是否匹配，但不需要知道其文本内容的情况下，使用这个方法非常方便。因此，test()方法经常被用在 if 语句中，如下面的例子所示。

```
var text = "000-00-0000";
var pattern = /\d{3}-\d{2}-\d{4}/;

if (pattern.test(text)){
    alert("The pattern was matched.");
}
```

在这个例子中，我们使用正则表达式来测试了一个数字序列。如果输入的文本与模式匹配，则显示一条消息。这种用法经常出现在验证用户输入的情况下，因为我们只想知道输入是不是有效，至于它为什么无效就无关紧要了。

RegExp 实例继承的 toLocaleString()和 toString()方法都会返回正则表达式的字面量，与创建正则表达式的方式无关。例如：

```
var pattern = new RegExp("\\[bc\\]at", "gi");
alert(pattern.toString());             // /\[bc\]at/gi
alert(pattern.toLocaleString());       // /\[bc\]at/gi
```

RegExpToStringExample01.htm

即使上例中的模式是通过调用 RegExp 构造函数创建的，但 toLocaleString()和 toString()方法仍然会像它是以字面量形式创建的一样显示其字符串表示。

正则表达式的 valueOf()方法返回正则表达式本身。

5.4.3 RegExp 构造函数属性

RegExp 构造函数包含一些属性（这些属性在其他语言中被看成是静态属性）。这些属性适用于作用

域中的所有正则表达式，并且基于所执行的最近一次正则表达式操作而变化。关于这些属性的另一个独特之处，就是可以通过两种方式访问它们。换句话说，这些属性分别有一个长属性名和一个短属性名（Opera是例外，它不支持短属性名）。下表列出了RegExp构造函数的属性。

长属性名	短属性名	说　明
input	$_	最近一次要匹配的字符串。Opera未实现此属性
lastMatch	$&	最近一次的匹配项。Opera未实现此属性
lastParen	$+	最近一次匹配的捕获组。Opera未实现此属性
leftContext	$`	input字符串中lastMatch之前的文本
multiline	$*	布尔值，表示是否所有表达式都使用多行模式。IE和Opera未实现此属性
rightContext	$'	Input字符串中lastMatch之后的文本

使用这些属性可以从exec()或test()执行的操作中提取出更具体的信息。请看下面的例子。

```
var text = "this has been a short summer";
var pattern = /(.)hort/g;

/*
 * 注意：Opera 不支持 input、lastMatch、lastParen 和 multiline 属性
 * Internet Explorer 不支持 multiline 属性
 */
if (pattern.test(text)){
    alert(RegExp.input);            // this has been a short summer
    alert(RegExp.leftContext);      // this has been a
    alert(RegExp.rightContext);     // summer
    alert(RegExp.lastMatch);        // short
    alert(RegExp.lastParen);        // s
    alert(RegExp.multiline);        // false
}
```

RegExpConstructorPropertiesExample01.htm

以上代码创建了一个模式，匹配任何一个字符后跟hort，而且把第一个字符放在了一个捕获组中。RegExp构造函数的各个属性返回了下列值：

- input属性返回了原始字符串；
- leftContext属性返回了单词short之前的字符串，而rightContext属性则返回了short之后的字符串；
- lastMatch属性返回最近一次与整个正则表达式匹配的字符串，即short；
- lastParen属性返回最近一次匹配的捕获组，即例子中的s。

如前所述，例子使用的长属性名都可以用相应的短属性名来代替。只不过，由于这些短属性名大都不是有效的ECMAScript标识符，因此必须通过方括号语法来访问它们，如下所示。

```
var text = "this has been a short summer";
var pattern = /(.)hort/g;

/*
 * 注意：Opera 不支持 input、lastMatch、lastParen 和 multiline 属性
 * Internet Explorer 不支持 multiline 属性
 */
```

```
if (pattern.test(text)){
    alert(RegExp.$_);                    // this has been a short summer
    alert(RegExp["$`"]);                 // this has been a
    alert(RegExp["$'"]);                 // summer
    alert(RegExp["$&"]);                 // short
    alert(RegExp["$+"]);                 // s
    alert(RegExp["$*"]);                 // false
}
```

RegExpConstructorPropertiesExample02.htm

除了上面介绍的几个属性之外，还有多达 9 个用于存储捕获组的构造函数属性。访问这些属性的语法是 RegExp.$1、RegExp.$2…RegExp.$9，分别用于存储第一、第二……第九个匹配的捕获组。在调用 exec()或 test()方法时，这些属性会被自动填充。然后，我们就可以像下面这样来使用它们。

```
var text = "this has been a short summer";
var pattern = /(..)or(.)/g;

if (pattern.test(text)){
    alert(RegExp.$1);        //sh
    alert(RegExp.$2);        //t
}
```

RegExpConstructorPropertiesExample03.htm

这里创建了一个包含两个捕获组的模式，并用该模式测试了一个字符串。即使 test()方法只返回一个布尔值，但 RegExp 构造函数的属性$1 和$2 也会被匹配相应捕获组的字符串自动填充。

5.4.4 模式的局限性

尽管 ECMAScript 中的正则表达式功能还是比较完备的，但仍然缺少某些语言（特别是 Perl）所支持的高级正则表达式特性。下面列出了 ECMAScript 正则表达式不支持的特性（要了解更多相关信息，请访问 www.regular-expressions.info）。

- 匹配字符串开始和结尾的\A 和\Z 锚[①]
- 向后查找（lookbehind）[②]
- 并集和交集类
- 原子组（atomic grouping）
- Unicode 支持（单个字符除外，如\uFFFF）
- 命名的捕获组[③]
- s（single，单行）和 x（free-spacing，无间隔）匹配模式
- 条件匹配
- 正则表达式注释

即使存在这些限制，ECMAScript 正则表达式仍然是非常强大的，能够帮我们完成绝大多数模式匹配任务。

① 但支持以插入符号（^）和美元符号（$）来匹配字符串的开始和结尾。
② 但完全支持向前查找（lookahead）。
③ 但支持编号的捕获组。

5.5 Function 类型

说起来 ECMAScript 中什么最有意思，我想那莫过于函数了——而有意思的根源，则在于函数实际上是对象。每个函数都是 Function 类型的实例，而且都与其他引用类型一样具有属性和方法。由于函数是对象，因此函数名实际上也是一个指向函数对象的指针，不会与某个函数绑定。函数通常是使用函数声明语法定义的，如下面的例子所示。

```
function sum (num1, num2) {
    return num1 + num2;
}
```

这与下面使用函数表达式定义函数的方式几乎相差无几。

```
var sum = function(num1, num2){
    return num1 + num2;
};
```

以上代码定义了变量 sum 并将其初始化为一个函数。有读者可能会注意到，function 关键字后面没有函数名。这是因为在使用函数表达式定义函数时，没有必要使用函数名——通过变量 sum 即可以引用函数。另外，还要注意函数末尾有一个分号，就像声明其他变量时一样。

最后一种定义函数的方式是使用 Function 构造函数。Function 构造函数可以接收任意数量的参数，但最后一个参数始终都被看成是函数体，而前面的参数则枚举出了新函数的参数。来看下面的例子：

```
var sum = new Function("num1", "num2", "return num1 + num2"); // 不推荐
```

从技术角度讲，这是一个函数表达式。但是，我们不推荐读者使用这种方法定义函数，因为这种语法会导致解析两次代码（第一次是解析常规 ECMAScript 代码，第二次是解析传入构造函数中的字符串），从而影响性能。不过，这种语法对于理解“函数是对象，函数名是指针”的概念倒是非常直观的。

由于函数名仅仅是指向函数的指针，因此函数名与包含对象指针的其他变量没有什么不同。换句话说，一个函数可能会有多个名字，如下面的例子所示。

```
function sum(num1, num2){
    return num1 + num2;
}
alert(sum(10,10));          //20

var anotherSum = sum;
alert(anotherSum(10,10)); //20

sum = null;
alert(anotherSum(10,10)); //20
```

FunctionTypeExample01.htm

以上代码首先定义了一个名为 sum() 的函数，用于求两个值的和。然后，又声明了变量 anotherSum，并将其设置为与 sum 相等（将 sum 的值赋给 anotherSum）。注意，使用不带圆括号的函数名是访问函数指针，而非调用函数。此时，anotherSum 和 sum 就都指向了同一个函数，因此 anotherSum() 也可以被调用并返回结果。即使将 sum 设置为 null，让它与函数“断绝关系”，但仍然可以正常调用 anotherSum()。

5.5.1 没有重载（深入理解）

将函数名想象为指针，也有助于理解为什么 ECMAScript 中没有函数重载的概念。以下是曾在第 3 章使用过的例子。

```
function addSomeNumber(num){
    return num + 100;
}

function addSomeNumber(num) {
    return num + 200;
}

var result = addSomeNumber(100); //300
```

显然，这个例子中声明了两个同名函数，而结果则是后面的函数覆盖了前面的函数。以上代码实际上与下面的代码没有什么区别。

```
var addSomeNumber = function (num){
    return num + 100;
};

addSomeNumber = function (num) {
    return num + 200;
};

var result = addSomeNumber(100); //300
```

通过观察重写之后的代码，很容易看清楚到底是怎么回事儿——在创建第二个函数时，实际上覆盖了引用第一个函数的变量 addSomeNumber。

5.5.2 函数声明与函数表达式

本节到目前为止，我们一直没有对函数声明和函数表达式加以区别。而实际上，解析器在向执行环境中加载数据时，对函数声明和函数表达式并非一视同仁。解析器会率先读取函数声明，并使其在执行任何代码之前可用（可以访问）；至于函数表达式，则必须等到解析器执行到它所在的代码行，才会真正被解释执行。请看下面的例子。

```
alert(sum(10,10));
function sum(num1, num2){
    return num1 + num2;
}
```

FunctionDeclarationExample01.htm

以上代码完全可以正常运行。因为在代码开始执行之前，解析器就已经通过一个名为函数声明提升（function declaration hoisting）的过程，读取并将函数声明添加到执行环境中。对代码求值时，JavaScript 引擎在第一遍会声明函数并将它们放到源代码树的顶部。所以，即使声明函数的代码在调用它的代码后面，JavaScript 引擎也能把函数声明提升到顶部。如果像下面例子所示的，把上面的函数声明改为等价的函数表达式，就会在执行期间导致错误。

```
alert(sum(10,10));
var sum = function(num1, num2){
    return num1 + num2;
};
```

FunctionInitializationExample01.htm

以上代码之所以会在运行期间产生错误，原因在于函数位于一个初始化语句中，而不是一个函数声明。换句话说，在执行到函数所在的语句之前，变量 sum 中不会保存有对函数的引用；而且，由于第一行代码就会导致“unexpected identifier”（意外标识符）错误，实际上也不会执行到下一行。

除了什么时候可以通过变量访问函数这一点区别之外，函数声明与函数表达式的语法其实是等价的。

也可以同时使用函数声明和函数表达式，例如 var sum = function sum(){}。不过，这种语法在 Safari 中会导致错误。

5.5.3　作为值的函数

因为 ECMAScript 中的函数名本身就是变量，所以函数也可以作为值来使用。也就是说，不仅可以像传递参数一样把一个函数传递给另一个函数，而且可以将一个函数作为另一个函数的结果返回。来看一看下面的函数。

```
function callSomeFunction(someFunction, someArgument){
    return someFunction(someArgument);
}
```

这个函数接受两个参数。第一个参数应该是一个函数，第二个参数应该是要传递给该函数的一个值。然后，就可以像下面的例子一样传递函数了。

```
function add10(num){
    return num + 10;
}

var result1 = callSomeFunction(add10, 10);
alert(result1);    //20

function getGreeting(name){
    return "Hello, " + name;
}

var result2 = callSomeFunction(getGreeting, "Nicholas");
alert(result2);    //"Hello, Nicholas"
```

FunctionAsAnArgumentExample01.htm

这里的 callSomeFunction()函数是通用的，即无论第一个参数中传递进来的是什么函数，它都会返回执行第一个参数后的结果。还记得吧，要访问函数的指针而不执行函数的话，必须去掉函数名后面的那对圆括号。因此上面例子中传递给 callSomeFunction()的是 add10 和 getGreeting，而不是执行它们之后的结果。

当然，可以从一个函数中返回另一个函数，而且这也是极为有用的一种技术。例如，假设有一个对象数组，我们想要根据某个对象属性对数组进行排序。而传递给数组 sort()方法的比较函数要接收两个参数，即要比较的值。可是，我们需要一种方式来指明按照哪个属性来排序。要解决这个问题，可以定义一个函数，它接收一个属性名，然后根据这个属性名来创建一个比较函数，下面就是这个函数的定义。

```
function createComparisonFunction(propertyName) {

    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2){
            return -1;
        } else if (value1 > value2){
            return 1;
        } else {
            return 0;
        }
    };
}
```

FunctionReturningFunctionExample01.htm

这个函数定义看起来有点复杂，但实际上无非就是在一个函数中嵌套了另一个函数，而且内部函数前面加了一个 return 操作符。在内部函数接收到 propertyName 参数后，它会使用方括号表示法来取得给定属性的值。取得了想要的属性值之后，定义比较函数就非常简单了。上面这个函数可以像在下面例子中这样使用。

```
var data = [{name: "Zachary", age: 28}, {name: "Nicholas", age: 29}];

data.sort(createComparisonFunction("name"));
alert(data[0].name);  //Nicholas

data.sort(createComparisonFunction("age"));
alert(data[0].name);  //Zachary
```

这里，我们创建了一个包含两个对象的数组 data。其中，每个对象都包含一个 name 属性和一个 age 属性。在默认情况下，sort()方法会调用每个对象的 toString()方法以确定它们的次序；但得到的结果往往并不符合人类的思维习惯。因此，我们调用 createComparisonFunction("name")方法创建了一个比较函数，以便按照每个对象的 name 属性值进行排序。而结果排在前面的第一项是 name 为"Nicholas"，age 是 29 的对象。然后，我们又使用了 createComparisonFunction("age")返回的比较函数，这次是按照对象的 age 属性排序。得到的结果是 name 值为"Zachary"，age 值是 28 的对象排在了第一位。

5.5.4 函数内部属性

在函数内部，有两个特殊的对象：arguments 和 this。其中，arguments 在第 3 章曾经介绍过，它是一个类数组对象，包含着传入函数中的所有参数。虽然 arguments 的主要用途是保存函数参数，但这个对象还有一个名叫 callee 的属性，该属性是一个指针，指向拥有这个 arguments 对象的函数。请看下面这个非常经典的阶乘函数。

```
function factorial(num){
    if (num <=1) {
        return 1;
    } else {
        return num * factorial(num-1)
    }
}
```

定义阶乘函数一般都要用到递归算法；如上面的代码所示，在函数有名字，而且名字以后也不会变的情况下，这样定义没有问题。但问题是这个函数的执行与函数名 factorial 紧紧耦合在了一起。为了消除这种紧密耦合的现象，可以像下面这样使用 arguments.callee。

```
function factorial(num){
    if (num <=1) {
        return 1;
    } else {
        return num * arguments.callee(num-1)
    }
}
```

FunctionTypeArgumentsExample01.htm

在这个重写后的 factorial()函数的函数体内，没有再引用函数名 factorial。这样，无论引用函数时使用的是什么名字，都可以保证正常完成递归调用。例如：

```
var trueFactorial = factorial;

factorial = function(){
    return 0;
};

alert(trueFactorial(5));      //120
alert(factorial(5));          //0
```

在此，变量 trueFactorial 获得了 factorial 的值，实际上是在另一个位置上保存了一个函数的指针。然后，我们又将一个简单地返回 0 的函数赋值给 factorial 变量。如果像原来的 factorial()那样不使用 arguments.callee，调用 trueFactorial()就会返回 0。可是，在解除了函数体内的代码与函数名的耦合状态之后，trueFactorial()仍然能够正常地计算阶乘；至于 factorial()，它现在只是一个返回 0 的函数。

函数内部的另一个特殊对象是 this，其行为与 Java 和 C#中的 this 大致类似。换句话说，this 引用的是函数据以执行的环境对象——或者也可以说是 this 值(当在网页的全局作用域中调用函数时，this 对象引用的就是 window)。来看下面的例子。

```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}

sayColor();      //"red"

o.sayColor = sayColor;
o.sayColor();    //"blue"
```

FunctionTypeThisExample01.htm

上面这个函数 sayColor()是在全局作用域中定义的，它引用了 this 对象。由于在调用函数之前，this 的值并不确定，因此 this 可能会在代码执行过程中引用不同的对象。当在全局作用域中调用 sayColor()时，this 引用的是全局对象 window；换句话说，对 this.color 求值会转换成对 window.color 求值，于是结果就返回了"red"。而当把这个函数赋给对象 o 并调用 o.sayColor()时，this 引用的是对象 o，因此对 this.color 求值会转换成对 o.color 求值，结果就返回了"blue"。

请读者一定要牢记，函数的名字仅仅是一个包含指针的变量而已。因此，即使是在不同的环境中执行，全局的 sayColor()函数与 o.sayColor()指向的仍然是同一个函数。

ECMAScript 5 也规范化了另一个函数对象的属性：caller。除了 Opera 的早期版本不支持，其他浏览器都支持这个 ECMAScript 3 并没有定义的属性。这个属性中保存着调用当前函数的函数的引用，如果是在全局作用域中调用当前函数，它的值为 null。例如：

```
function outer(){
    inner();
}

function inner(){
    alert(inner.caller);
}

outer();
```

FunctionTypeArgumentsCallerExample01.htm

以上代码会导致警告框中显示 outer()函数的源代码。因为 outer()调用了 inter()，所以 inner.caller 就指向 outer()。为了实现更松散的耦合，也可以通过 arguments.callee.caller 来访问相同的信息。

```
function outer(){
    inner();
}

function inner(){
    alert(arguments.callee.caller);
}

outer();
```

FunctionTypeArgumentsCallerExample02.htm

IE、Firefox、Chrome 和 Safari 的所有版本以及 Opera 9.6 都支持 caller 属性。

当函数在严格模式下运行时，访问 arguments.callee 会导致错误。ECMAScript 5 还定义了 arguments.caller 属性，但在严格模式下访问它也会导致错误，而在非严格模式下这个属性始终是 undefined。定义这个属性是为了分清 arguments.caller 和函数的 caller 属性。以上变化都是为了加强这门语言的安全性，这样第三方代码就不能在相同的环境里窥视其他代码了。

严格模式还有一个限制：不能为函数的 caller 属性赋值，否则会导致错误。

5.5.5　函数属性和方法

前面曾经提到过，ECMAScript 中的函数是对象，因此函数也有属性和方法。每个函数都包含两个属性：length 和 prototype。其中，length 属性表示函数希望接收的命名参数的个数，如下面的例子所示。

```
function sayName(name){
    alert(name);
}

function sum(num1, num2){
    return num1 + num2;
}

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

alert(sayName.length);      //1
alert(sum.length);          //2
alert(sayHi.length);        //0
```

FunctionTypeLengthPropertyExample01.htm

以上代码定义了 3 个函数，但每个函数接收的命名参数个数不同。首先，sayName()函数定义了一个参数，因此其 length 属性的值为 1。类似地，sum()函数定义了两个参数，结果其 length 属性中保存的值为 2。而 sayHi()没有命名参数，所以其 length 值为 0。

在 ECMAScript 核心所定义的全部属性中，最耐人寻味的就要数 prototype 属性了。对于 ECMAScript 中的引用类型而言，prototype 是保存它们所有实例方法的真正所在。换句话说，诸如 toString()和 valueOf()等方法实际上都保存在 prototype 名下，只不过是通过各自对象的实例访问罢了。在创建自定义引用类型以及实现继承时，prototype 属性的作用是极为重要的（第 6 章将详细介绍）。在 ECMAScript 5 中，prototype 属性是不可枚举的，因此使用 for-in 无法发现。

每个函数都包含两个非继承而来的方法：apply()和 call()。这两个方法的用途都是在特定的作用域中调用函数，实际上等于设置函数体内 this 对象的值。首先，apply()方法接收两个参数：一个是在其中运行函数的作用域，另一个是参数数组。其中，第二个参数可以是 Array 的实例，也可以是 arguments 对象。例如：

```
function sum(num1, num2){
    return num1 + num2;
}

function callSum1(num1, num2){
    return sum.apply(this, arguments);          // 传入 arguments 对象
}

function callSum2(num1, num2){
    return sum.apply(this, [num1, num2]);       // 传入数组
}

alert(callSum1(10,10));    //20
alert(callSum2(10,10));    //20
```

FunctionTypeApplyMethodExample01.htm

在上面这个例子中，callSum1()在执行 sum()函数时传入了 this 作为 this 值（因为是在全局作用域中调用的，所以传入的就是 window 对象）和 arguments 对象。而 callSum2 同样也调用了 sum()函数，但它传入的则是 this 和一个参数数组。这两个函数都会正常执行并返回正确的结果。

在严格模式下，未指定环境对象而调用函数，则 this 值不会转型为 window。除非明确把函数添加到某个对象或者调用 apply()或 call()，否则 this 值将是 undefined。

call()方法与 apply()方法的作用相同，它们的区别仅在于接收参数的方式不同。对于 call()方法而言，第一个参数是 this 值没有变化，变化的是其余参数都直接传递给函数。换句话说，在使用 call()方法时，传递给函数的参数必须逐个列举出来，如下面的例子所示。

```
function sum(num1, num2){
    return num1 + num2;
}

function callSum(num1, num2){
    return sum.call(this, num1, num2);
}

alert(callSum(10,10));    //20
```

FunctionTypeCallMethodExample01.htm

在使用 call()方法的情况下，callSum()必须明确地传入每一个参数。结果与使用 apply()没有什么不同。至于是使用 apply()还是 call()，完全取决于你采取哪种给函数传递参数的方式最方便。如果你打算直接传入 arguments 对象，或者包含函数中先接收到的也是一个数组，那么使用 apply()肯定更方便；否则，选择 call()可能更合适。（在不给函数传递参数的情况下，使用哪个方法都无所谓。）

事实上，传递参数并非 apply()和 call()真正的用武之地；它们真正强大的地方是能够扩充函数赖以运行的作用域。下面来看一个例子。

```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}

sayColor();                 //red

sayColor.call(this);        //red
sayColor.call(window);      //red
sayColor.call(o);           //blue
```

FunctionTypeCallExample01.htm

这个例子是在前面说明 this 对象的示例基础上修改而成的。这一次，sayColor()也是作为全局函数定义的，而且当在全局作用域中调用它时，它确实会显示"red"——因为对 this.color 的求值会

转换成对 window.color 的求值。而 sayColor.call(this)和 sayColor.call(window)，则是两种显式地在全局作用域中调用函数的方式，结果当然都会显示"red"。但是，当运行 sayColor.call(o)时，函数的执行环境就不一样了，因为此时函数体内的 this 对象指向了 o，于是结果显示的是"blue"。

使用 call()（或 apply()）来扩充作用域的最大好处，就是对象不需要与方法有任何耦合关系。在前面例子的第一个版本中，我们是先将 sayColor()函数放到了对象 o 中，然后再通过 o 来调用它的；而在这里重写的例子中，就不需要先前那个多余的步骤了。

ECMAScript 5 还定义了一个方法：bind()。这个方法会创建一个函数的实例，其 this 值会被绑定到传给 bind()函数的值。例如：

```
window.color = "red";
var o = { color: "blue" };

function sayColor(){
    alert(this.color);
}
var objectSayColor = sayColor.bind(o);
objectSayColor();     //blue
```

FunctionTypeBindMethodExample01.htm

在这里，sayColor()调用 bind()并传入对象 o，创建了 objectSayColor()函数。objectSayColor()函数的 this 值等于 o，因此即使是在全局作用域中调用这个函数，也会看到"blue"。这种技巧的优点请参考第 22 章。

支持 bind()方法的浏览器有 IE9+、Firefox 4+、Safari 5.1+、Opera 12+和 Chrome。

每个函数继承的 toLocaleString()和 toString()方法始终都返回函数的代码。返回代码的格式则因浏览器而异——有的返回的代码与源代码中的函数代码一样，而有的则返回函数代码的内部表示，即由解析器删除了注释并对某些代码作了改动后的代码。由于存在这些差异，我们无法根据这两个方法返回的结果来实现任何重要功能；不过，这些信息在调试代码时倒是很有用。另外一个继承的 valueOf()方法同样也只返回函数代码。

5.6　基本包装类型

为了便于操作基本类型值，ECMAScript 还提供了 3 个特殊的引用类型：Boolean、Number 和 String。这些类型与本章介绍的其他引用类型相似，但同时也具有与各自的基本类型相应的特殊行为。实际上，每当读取一个基本类型值的时候，后台就会创建一个对应的基本包装类型的对象，从而让我们能够调用一些方法来操作这些数据。来看下面的例子。

```
var s1 = "some text";
var s2 = s1.substring(2);
```

这个例子中的变量 s1 包含一个字符串，字符串当然是基本类型值。而下一行调用了 s1 的 substring()方法，并将返回的结果保存在了 s2 中。我们知道，基本类型值不是对象，因而从逻辑上讲它们不应该有方法（尽管如我们所愿，它们确实有方法）。其实，为了让我们实现这种直观的操作，后台已经自动完成了一系列的处理。当第二行代码访问 s1 时，访问过程处于一种读取模式，也就是要从内存中读取这个字符串的值。而在读取模式中访问字符串时，后台都会自动完成下列处理。

(1) 创建 String 类型的一个实例；
(2) 在实例上调用指定的方法；
(3) 销毁这个实例。
可以将以上三个步骤想象成是执行了下列 ECMAScript 代码。

```
var s1 = new String("some text");
var s2 = s1.substring(2);
s1 = null;
```

经过此番处理，基本的字符串值就变得跟对象一样了。而且，上面这三个步骤也分别适用于 Boolean 和 Number 类型对应的布尔值和数字值。

引用类型与基本包装类型的主要区别就是对象的生存期。使用 new 操作符创建的引用类型的实例，在执行流离开当前作用域之前都一直保存在内存中。而自动创建的基本包装类型的对象，则只存在于一行代码的执行瞬间，然后立即被销毁。这意味着我们不能在运行时为基本类型值添加属性和方法。来看下面的例子：

```
var s1 = "some text";
s1.color = "red";
alert(s1.color);    //undefined
```

在此，第二行代码试图为字符串 s1 添加一个 color 属性。但是，当第三行代码再次访问 s1 时，其 color 属性不见了。问题的原因就是第二行创建的 String 对象在执行第三行代码时已经被销毁了。第三行代码又创建自己的 String 对象，而该对象没有 color 属性。

当然，可以显式地调用 Boolean、Number 和 String 来创建基本包装类型的对象。不过，应该在绝对必要的情况下再这样做，因为这种做法很容易让人分不清自己是在处理基本类型还是引用类型的值。对基本包装类型的实例调用 typeof 会返回"object"，而且所有基本包装类型的对象都会被转换为布尔值 true。

Object 构造函数也会像工厂方法一样，根据传入值的类型返回相应基本包装类型的实例。例如：

```
var obj = new Object("some text");
alert(obj instanceof String);    //true
```

把字符串传给 Object 构造函数，就会创建 String 的实例；而传入数值参数会得到 Number 的实例，传入布尔值参数就会得到 Boolean 的实例。

要注意的是，使用 new 调用基本包装类型的构造函数，与直接调用同名的转型函数是不一样的。例如：

```
var value = "25";
var number = Number(value);   //转型函数
alert(typeof number);         //"number"

var obj = new Number(value);  //构造函数
alert(typeof obj);            //"object"
```

在这个例子中，变量 number 中保存的是基本类型的值 25，而变量 obj 中保存的是 Number 的实例。要了解有关转型函数的更多信息，请参考第 3 章。

尽管我们不建议显式地创建基本包装类型的对象，但它们操作基本类型值的能力还是相当重要的。而每个基本包装类型都提供了操作相应值的便捷方法。

5.6.1 Boolean 类型

Boolean 类型是与布尔值对应的引用类型。要创建 Boolean 对象，可以像下面这样调用 Boolean 构造函数并传入 true 或 false 值。

```
var booleanObject = new Boolean(true);
```

Boolean 类型的实例重写了 valueOf()方法，返回基本类型值 true 或 false；重写了 toString()方法，返回字符串"true"和"false"。可是，Boolean 对象在 ECMAScript 中的用处不大，因为它经常会造成人们的误解。其中最常见的问题就是在布尔表达式中使用 Boolean 对象，例如：

```
var falseObject = new Boolean(false);
var result = falseObject && true;
alert(result);  //true

var falseValue = false;
result = falseValue && true;
alert(result);  //false
```

BooleanTypeExample01.htm

在这个例子中，我们使用 false 值创建了一个 Boolean 对象。然后，将这个对象与基本类型值 true 构成了逻辑与表达式。在布尔运算中，false && true 等于 false。可是，示例中的这行代码是对 falseObject 而不是对它的值（false）进行求值。前面讨论过，布尔表达式中的所有对象都会被转换为 true，因此 falseObject 对象在布尔表达式中代表的是 true。结果，true && true 当然就等于 true 了。

基本类型与引用类型的布尔值还有两个区别。首先，typeof 操作符对基本类型返回"boolean"，而对引用类型返回"object"。其次，由于 Boolean 对象是 Boolean 类型的实例，所以使用 instanceof 操作符测试 Boolean 对象会返回 true，而测试基本类型的布尔值则返回 false。例如：

```
alert(typeof falseObject);   //object
alert(typeof falseValue);    //boolean
alert(falseObject instanceof Boolean);  //true
alert(falseValue instanceof Boolean);   //false
```

理解基本类型的布尔值与 Boolean 对象之间的区别非常重要——当然，我们的建议是永远不要使用 Boolean 对象。

5.6.2 Number 类型

Number 是与数字值对应的引用类型。要创建 Number 对象，可以在调用 Number 构造函数时向其中传递相应的数值。下面是一个例子。

```
var numberObject = new Number(10);
```

NumberTypeExample01.htm

与 Boolean 类型一样，Number 类型也重写了 valueOf()、toLocaleString()和 toString()方法。重写后的 valueOf()方法返回对象表示的基本类型的数值，另外两个方法则返回字符串形式的

数值。我们在第 3 章还介绍过，可以为 toString()方法传递一个表示基数的参数，告诉它返回几进制数值的字符串形式，如下面的例子所示。

```
var num = 10;
alert(num.toString());      //"10"
alert(num.toString(2));     //"1010"
alert(num.toString(8));     //"12"
alert(num.toString(10));    //"10"
alert(num.toString(16));    //"a"
```

NumberTypeExample01.htm

除了继承的方法之外，Number 类型还提供了一些用于将数值格式化为字符串的方法。其中，toFixed()方法会按照指定的小数位返回数值的字符串表示，例如：

```
var num = 10;
alert(num.toFixed(2));      //"10.00"
```

NumberTypeExample01.htm

这里给 toFixed()方法传入了数值 2，意思是显示几位小数。于是，这个方法返回了"10.00"，即以 0 填补了必要的小数位。如果数值本身包含的小数位比指定的还多，那么接近指定的最大小数位的值就会舍入，如下面的例子所示。

```
var num = 10.005;
alert(num.toFixed(2));      //"10.01"
```

能够自动舍入的特性，使得 toFixed()方法很适合处理货币值。但需要注意的是，不同浏览器给这个方法设定的舍入规则可能会有所不同。在给 toFixed()传入 0 的情况下，IE8 及之前版本不能正确舍入范围在{(–0.94,–0.5],[0.5,0.94)}之间的值。对于这个范围内的值，IE 会返回 0，而不是–1 或 1；其他浏览器都能返回正确的值。IE9 修复了这个问题。

toFixed()方法可以表示带有 0 到 20 个小数位的数值。但这只是标准实现的范围，有些浏览器也可能支持更多位数。

另外可用于格式化数值的方法是 toExponential()，该方法返回以指数表示法（也称 e 表示法）表示的数值的字符串形式。与 toFixed()一样，toExponential()也接收一个参数，而且该参数同样也是指定输出结果中的小数位数。看下面的例子。

```
var num = 10;
alert(num.toExponential(1));    //"1.0e+1"
```

以上代码输出了"1.0e+1"；不过，这么小的数值一般不必使用 e 表示法。如果你想得到表示某个数值的最合适的格式，就应该使用 toPrecision()方法。

对于一个数值来说，toPrecision()方法可能会返回固定大小（fixed）格式，也可能返回指数（exponential）格式；具体规则是看哪种格式最合适。这个方法接收一个参数，即表示数值的所有数字的位数（不包括指数部分）。请看下面的例子。

```
var num = 99;
alert(num.toPrecision(1));      //"1e+2"
alert(num.toPrecision(2));      //"99"
alert(num.toPrecision(3));      //"99.0"
```

NumberTypeExample01.htm

以上代码首先完成的任务是以一位数来表示 99，结果是"1e+2"，即 100。因为一位数无法准确地表示 99，因此 toPrecision()就将它向上舍入为 100，这样就可以使用一位数来表示它了。而接下来的用两位数表示 99，当然还是"99"。最后，在想以三位数表示 99 时，toPrecision()方法返回了"99.0"。实际上，toPrecision()会根据要处理的数值决定到底是调用 toFixed()还是调用 toExponential()。而这三个方法都可以通过向上或向下舍入，做到以最准确的形式来表示带有正确小数位的值。

toPrecision()方法可以表现 1 到 21 位小数。某些浏览器支持的范围更大，但这是典型实现的范围。

与 Boolean 对象类似，Number 对象也以后台方式为数值提供了重要的功能。但与此同时，我们仍然不建议直接实例化 Number 类型，而原因与显式创建 Boolean 对象一样。具体来讲，就是在使用 typeof 和 instanceof 操作符测试基本类型数值与引用类型数值时，得到的结果完全不同，如下面的例子所示。

```
var numberObject = new Number(10);
var numberValue = 10;
alert(typeof numberObject);    //"object"
alert(typeof numberValue);     //"number"
alert(numberObject instanceof Number);  //true
alert(numberValue instanceof Number);   //false
```

在使用 typeof 操作符测试基本类型数值时，始终会返回"number"，而在测试 Number 对象时，则会返回"object"。类似地，Number 对象是 Number 类型的实例，而基本类型的数值则不是。

5.6.3 String 类型

String 类型是字符串的对象包装类型，可以像下面这样使用 String 构造函数来创建。

```
var stringObject = new String("hello world");
```

StringTypeExample01.htm

String 对象的方法也可以在所有基本的字符串值中访问到。其中，继承的 valueOf()、toLocaleString()和 toString()方法，都返回对象所表示的基本字符串值。

String 类型的每个实例都有一个 length 属性，表示字符串中包含多个字符。来看下面的例子。

```
var stringValue = "hello world";
alert(stringValue.length);      //"11"
```

这个例子输出了字符串"hello world"中的字符数量，即"11"。应该注意的是，即使字符串中包含双字节字符（不是占一个字节的 ASCII 字符），每个字符也仍然算一个字符。

String 类型提供了很多方法，用于辅助完成对 ECMAScript 中字符串的解析和操作。

1. 字符方法

两个用于访问字符串中特定字符的方法是：charAt()和 charCodeAt()。这两个方法都接收一个参数，即基于 0 的字符位置。其中，charAt()方法以单字符字符串的形式返回给定位置的那个字符（ECMAScript 中没有字符类型）。例如：

```
var stringValue = "hello world";
alert(stringValue.charAt(1));   //"e"
```

字符串"hello world"位置 1 处的字符是"e"，因此调用 charAt(1)就返回了"e"。如果你想得到的不是字符而是字符编码，那么就要像下面这样使用 charCodeAt()了。

```
var stringValue = "hello world";
alert(stringValue.charCodeAt(1));   //输出"101"
```

这个例子输出的是"101"，也就是小写字母"e"的字符编码。

ECMAScript 5 还定义了另一个访问个别字符的方法。在支持浏览器中，可以使用方括号加数字索引来访问字符串中的特定字符，如下面的例子所示。

```
var stringValue = "hello world";
alert(stringValue[1]);   //"e"
```

使用方括号表示法访问个别字符的语法得到了 IE8 及 Firefox、Safari、Chrome 和 Opera 所有版本的支持。如果是在 IE7 及更早版本中使用这种语法，会返回 undefined 值（尽管根本不是特殊的 undefined 值）。

2. 字符串操作方法

下面介绍与操作字符串有关的几个方法。第一个就是 concat()，用于将一或多个字符串拼接起来，返回拼接得到的新字符串。先来看一个例子。

```
var stringValue = "hello ";
var result = stringValue.concat("world");
alert(result);              //"hello world"
alert(stringValue);         //"hello"
```

在这个例子中，通过 stringValue 调用 concat()方法返回的结果是"hello world"——但 stringValue 的值则保持不变。实际上，concat()方法可以接受任意多个参数，也就是说可以通过它拼接任意多个字符串。再看一个例子：

```
var stringValue = "hello ";
var result = stringValue.concat("world", "!");

alert(result);              //"hello world!"
alert(stringValue);         //"hello"
```

这个例子将"world"和"!"拼接到了"hello"的末尾。虽然 concat()是专门用来拼接字符串的方法，但实践中使用更多的还是加号操作符(+)。而且，使用加号操作符在大多数情况下都比使用 concat()方法要简便易行（特别是在拼接多个字符串的情况下）。

ECMAScript 还提供了三个基于子字符串创建新字符串的方法：slice()、substr()和 substring()。这三个方法都会返回被操作字符串的一个子字符串，而且也都接受一或两个参数。第一个参数指定子字符串的开始位置，第二个参数（在指定的情况下）表示子字符串到哪里结束。具体来说，slice()和 substring()的第二个参数指定的是子字符串最后一个字符后面的位置。而 substr()的第二个参数指

定的则是返回的字符个数。如果没有给这些方法传递第二个参数，则将字符串的长度作为结束位置。与concat()方法一样，slice()、substr()和substring()也不会修改字符串本身的值——它们只是返回一个基本类型的字符串值，对原始字符串没有任何影响。请看下面的例子。

```
var stringValue = "hello world";
alert(stringValue.slice(3));            //"lo world"
alert(stringValue.substring(3));        //"lo world"
alert(stringValue.substr(3));           //"lo world"
alert(stringValue.slice(3, 7));         //"lo w"
alert(stringValue.substring(3,7));      //"lo w"
alert(stringValue.substr(3, 7));        //"lo worl"
```

StringTypeManipulationMethodsExample01.htm

这个例子比较了以相同方式调用slice()、substr()和substring()得到的结果，而且多数情况下的结果是相同的。在只指定一个参数3的情况下，这三个方法都返回"lo world"，因为"hello"中的第二个"l"处于位置3。而在指定两个参数3和7的情况下，slice()和substring()返回"lo w"（"world"中的"o"处于位置7，因此结果中不包含"o"），但substr()返回"lo worl"，因为它的第二个参数指定的是要返回的字符个数。

在传递给这些方法的参数是负值的情况下，它们的行为就不尽相同了。其中，slice()方法会将传入的负值与字符串的长度相加，substr()方法将负的第一个参数加上字符串的长度，而将负的第二个参数转换为0。最后，substring()方法会把所有负值参数都转换为0。下面来看例子。

```
var stringValue = "hello world";
alert(stringValue.slice(-3));           //"rld"
alert(stringValue.substring(-3));       //"hello world"
alert(stringValue.substr(-3));          //"rld"
alert(stringValue.slice(3, -4));        //"lo w"
alert(stringValue.substring(3, -4));    //"hel"
alert(stringValue.substr(3, -4));       //"" (空字符串)
```

StringTypeManipulationMethodsExample01.htm

这个例子清晰地展示了上述三个方法之间的不同行为。在给slice()和substr()传递一个负值参数时，它们的行为相同。这是因为-3会被转换为8（字符串长度加参数11+(-3)=8），实际上相当于调用了slice(8)和substr(8)。但substring()方法则返回了全部字符串，因为它将-3转换成了0。

IE的JavaScript实现在处理向substr()方法传递负值的情况时存在问题，它会返回原始的字符串。IE9修复了这个问题。

当第二个参数是负值时，这三个方法的行为各不相同。slice()方法会把第二个参数转换为7，这就相当于调用了slice(3,7)，因此返回"lo w"。substring()方法会把第二个参数转换为0，使调用变成了substring(3,0)，而由于这个方法会将较小的数作为开始位置，将较大的数作为结束位置，因此最终相当于调用了substring(0,3)。substr()也会将第二个参数转换为0，这也就意味着返回包含零个字符的字符串，也就是一个空字符串。

3. 字符串位置方法

有两个可以从字符串中查找子字符串的方法：indexOf()和 lastIndexOf()。这两个方法都是从一个字符串中搜索给定的子字符串，然后返子字符串的位置（如果没有找到该子字符串，则返回-1）。这两个方法的区别在于：indexOf()方法从字符串的开头向后搜索子字符串，而 lastIndexOf()方法是从字符串的末尾向前搜索子字符串。还是来看一个例子吧。

```
var stringValue = "hello world";
alert(stringValue.indexOf("o"));          //4
alert(stringValue.lastIndexOf("o"));      //7
```

StringTypeLocationMethodsExample01.htm

子字符串"o"第一次出现的位置是 4，即"hello"中的"o"；最后一次出现的位置是 7，即"world"中的"o"。如果"o"在这个字符串中仅出现了一次，那么 indexOf()和 lastIndexOf()会返回相同的位置值。

这两个方法都可以接收可选的第二个参数，表示从字符串中的哪个位置开始搜索。换句话说，indexOf()会从该参数指定的位置向后搜索，忽略该位置之前的所有字符；而 lastIndexOf()则会从指定的位置向前搜索，忽略该位置之后的所有字符。看下面的例子。


```
var stringValue = "hello world";
alert(stringValue.indexOf("o", 6));       //7
alert(stringValue.lastIndexOf("o", 6));   //4
```

在将第二个参数 6 传递给这两个方法之后，得到了与前面例子相反的结果。这一次，由于 indexOf()是从位置 6（字母"w"）开始向后搜索，结果在位置 7 找到了"o"，因此它返回 7。而 lastIndexOf()是从位置 6 开始向前搜索。结果找到了"hello"中的"o"，因此它返回 4。在使用第二个参数的情况下，可以通过循环调用 indexOf()或 lastIndexOf()来找到所有匹配的子字符串，如下面的例子所示：

```
var stringValue = "Lorem ipsum dolor sit amet, consectetur adipisicing elit";
var positions = new Array();
var pos = stringValue.indexOf("e");

while(pos > -1){
    positions.push(pos);
    pos = stringValue.indexOf("e", pos + 1);
}

alert(positions);     //"3,24,32,35,52"
```

StringTypeLocationMethodsExample02.htm

这个例子通过不断增加 indexOf()方法开始查找的位置，遍历了一个长字符串。在循环之外，首先找到了"e"在字符串中的初始位置；而进入循环后，则每次都给 indexOf()传递上一次的位置加 1。这样，就确保了每次新搜索都从上一次找到的子字符串的后面开始。每次搜索返回的位置依次被保存在数组 positions 中，以便将来使用。

4. trim()方法

ECMAScript 5 为所有字符串定义了 trim()方法。这个方法会创建一个字符串的副本，删除前置及后缀的所有空格，然后返回结果。例如：

```
var stringValue = "   hello world   ";
var trimmedStringValue = stringValue.trim();
alert(stringValue);             //"   hello world   "
alert(trimmedStringValue);      //"hello world"
```

由于 trim()返回的是字符串的副本，所以原始字符串中的前置及后缀空格会保持不变。支持这个方法的浏览器有 IE9+、Firefox 3.5+、Safari 5+、Opera 10.5+和 Chrome。此外，Firefox 3.5+、Safari 5+和 Chrome 8+还支持非标准的 trimLeft()和 trimRight()方法，分别用于删除字符串开头和末尾的空格。

5. 字符串大小写转换方法

接下来我们要介绍的是一组与大小写转换有关的方法。ECMAScript 中涉及字符串大小写转换的方法有 4 个：toLowerCase()、toLocaleLowerCase()、toUpperCase()和 toLocaleUpperCase()。其中，toLowerCase()和 toUpperCase()是两个经典的方法，借鉴自 java.lang.String 中的同名方法。而 toLocaleLowerCase()和 toLocaleUpperCase()方法则是针对特定地区的实现。对有些地区来说，针对地区的方法与其通用方法得到的结果相同，但少数语言（如土耳其语）会为 Unicode 大小写转换应用特殊的规则，这时候就必须使用针对地区的方法来保证实现正确的转换。以下是几个例子。

```
var stringValue = "hello world";
alert(stringValue.toLocaleUpperCase());  //"HELLO WORLD"
alert(stringValue.toUpperCase());        //"HELLO WORLD"
alert(stringValue.toLocaleLowerCase());  //"hello world"
alert(stringValue.toLowerCase());        //"hello world"
```

StringTypeCaseMethodExample01.htm

以上代码调用的 toLocaleUpperCase()和 toUpperCase()都返回了"HELLO WORLD"，就像调用 toLocaleLowerCase()和 toLowerCase()都返回"hello world"一样。一般来说，在不知道自己的代码将在哪种语言环境中运行的情况下，还是使用针对地区的方法更稳妥一些。

6. 字符串的模式匹配方法

String 类型定义了几个用于在字符串中匹配模式的方法。第一个方法就是 match()，在字符串上调用这个方法，本质上与调用 RegExp 的 exec()方法相同。match()方法只接受一个参数，要么是一个正则表达式，要么是一个 RegExp 对象。来看下面的例子。

```
var text = "cat, bat, sat, fat";
var pattern = /.at/;

//与 pattern.exec(text)相同
var matches = text.match(pattern);
alert(matches.index);                //0
alert(matches[0]);                   //"cat"
alert(pattern.lastIndex);            //0
```

StringTypePatternMatchingExample01.htm

本例中的 match()方法返回了一个数组；如果是调用 RegExp 对象的 exec()方法并传递本例中的字符串作为参数，那么也会得到与此相同的数组：数组的第一项是与整个模式匹配的字符串，之后的每一项（如果有）保存着与正则表达式中的捕获组匹配的字符串。

另一个用于查找模式的方法是 search()。这个方法的唯一参数与 match()方法的参数相同：由字符串或 RegExp 对象指定的一个正则表达式。search()方法返回字符串中第一个匹配项的索引；如果没有找到匹配项，则返回-1。而且，search()方法始终是从字符串开头向后查找模式。看下面的例子。

```
var text = "cat, bat, sat, fat";
var pos = text.search(/at/);
alert(pos);   //1
```

StringTypePatternMatchingExample01.htm

这个例子中的 search()方法返回 1，即"at"在字符串中第一次出现的位置。

为了简化替换子字符串的操作，ECMAScript 提供了 replace()方法。这个方法接受两个参数：第一个参数可以是一个 RegExp 对象或者一个字符串（这个字符串不会被转换成正则表达式），第二个参数可以是一个字符串或者一个函数。如果第一个参数是字符串，那么只会替换第一个子字符串。要想替换所有子字符串，唯一的办法就是提供一个正则表达式，而且要指定全局（g）标志，如下所示。

```
var text = "cat, bat, sat, fat";
var result = text.replace("at", "ond");
alert(result);    //"cond, bat, sat, fat"

result = text.replace(/at/g, "ond");
alert(result);    //"cond, bond, sond, fond"
```

StringTypePatternMatchingExample01.htm

在这个例子中，首先传入 replace()方法的是字符串"at"和替换用的字符串"ond"。替换的结果是把"cat"变成了"cond"，但字符串中的其他字符并没有受到影响。然后，通过将第一个参数修改为带有全局标志的正则表达式，就将全部"at"都替换成了"ond"。

如果第二个参数是字符串，那么还可以使用一些特殊的字符序列，将正则表达式操作得到的值插入到结果字符串中。下表列出了 ECMAScript 提供的这些特殊的字符序列。

字符序列	替换文本
$$	$
$&	匹配整个模式的子字符串。与RegExp.lastMatch的值相同
$'	匹配的子字符串之前的子字符串。与RegExp.leftContext的值相同
$`	匹配的子字符串之后的子字符串。与RegExp.rightContext的值相同
$n	匹配第*n*个捕获组的子字符串，其中*n*等于0～9。例如，$1是匹配第一个捕获组的子字符串，$2是匹配第二个捕获组的子字符串，以此类推。如果正则表达式中没有定义捕获组，则使用空字符串
$nn	匹配第*nn*个捕获组的子字符串，其中*nn*等于01～99。例如，$01是匹配第一个捕获组的子字符串，$02是匹配第二个捕获组的子字符串，以此类推。如果正则表达式中没有定义捕获组，则使用空字符串

通过这些特殊的字符序列，可以使用最近一次匹配结果中的内容，如下面的例子所示。

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

StringTypePatternMatchingExample01.htm

在此，每个以"at"结尾的单词都被替换了，替换结果是"word"后跟一对圆括号，而圆括号中是被字符序列$1 所替换的单词。

replace()方法的第二个参数也可以是一个函数。在只有一个匹配项（即与模式匹配的字符串）的情况下，会向这个函数传递 3 个参数：模式的匹配项、模式匹配项在字符串中的位置和原始字符串。在正则表达式中定义了多个捕获组的情况下，传递给函数的参数依次是模式的匹配项、第一个捕获组的匹配项、第二个捕获组的匹配项……，但最后两个参数仍然分别是模式的匹配项在字符串中的位置和原始字符串。这个函数应该返回一个字符串，表示应该被替换的匹配项使用函数作为 replace()方法的第二个参数可以实现更加精细的替换操作，请看下面这个例子。

```
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;";
        }
    });
}

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

StringTypePatternMatchingExample01.htm

这里，我们为插入 HTML 代码定义了函数 htmlEscape()，这个函数能够转义 4 个字符：小于号、大于号、和号以及双引号。实现这种转义的最简单方式，就是使用正则表达式查找这几个字符，然后定义一个能够针对每个匹配的字符返回特定 HTML 实体的函数。

最后一个与模式匹配有关的方法是 split()，这个方法可以基于指定的分隔符将一个字符串分割成多个子字符串，并将结果放在一个数组中。分隔符可以是字符串，也可以是一个 RegExp 对象（这个方法不会将字符串看成正则表达式）。split()方法可以接受可选的第二个参数，用于指定数组的大小，以便确保返回的数组不会超过既定大小。请看下面的例子。

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

StringTypePatternMatchingExample01.htm

在这个例子中，colorText 是逗号分隔的颜色名字符串。基于该字符串调用 split(",")会得到一个包含其中颜色名的数组，用于分割字符串的分隔符是逗号。为了将数组截短，让它只包含两项，可以为 split()方法传递第二个参数 2。最后，通过使用正则表达式，还可以取得包含逗号字符的数组。需要注意的是，在最后一次调用 split()返回的数组中，第一项和最后一项是两个空字符串。之所以会

这样，是因为通过正则表达式指定的分隔符出现在了字符串的开头（即子字符串"red"）和末尾（即子字符串"yellow"）。

对 split()中正则表达式的支持因浏览器而异。尽管对于简单的模式没有什么差别，但对于未发现匹配项以及带有捕获组的模式，匹配的行为就不大相同了。以下是几种常见的差别。

- IE8 及之前版本会忽略捕获组。ECMA-262 规定应该把捕获组拼接到结果数组中。IE9 能正确地在结果中包含捕获组。
- Firefox 3.6 及之前版本在捕获组未找到匹配项时，会在结果数组中包含空字符串；ECMA-262 规定没有匹配项的捕获组在结果数组中应该用 undefined 表示。

在正则表达式中使用捕获组时还有其他微妙的差别。在使用这种正则表达式时，一定要在各种浏览器下多做一些测试。

要了解关于 split()方法以及捕获组的跨浏览器问题的更多讨论，请参考 Steven Levithan 的文章“JavaScript split bugs: Fixed!”（http://blog.stevenlevithan.com/archives/cross-browser-split）。

7. localeCompare()方法

与操作字符串有关的最后一个方法是 localeCompare()，这个方法比较两个字符串，并返回下列值中的一个：

- 如果字符串在字母表中应该排在字符串参数之前，则返回一个负数（大多数情况下是-1，具体的值要视实现而定）；
- 如果字符串等于字符串参数，则返回 0；
- 如果字符串在字母表中应该排在字符串参数之后，则返回一个正数（大多数情况下是 1，具体的值同样要视实现而定）。

下面是几个例子。

```
var stringValue = "yellow";
alert(stringValue.localeCompare("brick"));      //1
alert(stringValue.localeCompare("yellow"));     //0
alert(stringValue.localeCompare("zoo"));        //-1
```

StringTypeLocaleCompareExample01.htm

这个例子比较了字符串"yellow"和另外几个值："brick"、"yellow"和"zoo"。因为"brick"在字母表中排在"yellow"之前，所以 localeCompare()返回了 1；而"yellow"等于"yellow"，所以 localeCompare()返回了 0；最后，"zoo"在字母表中排在"yellow"后面，所以 localeCompare()返回了-1。再强调一次，因为 localeCompare()返回的数值取决于实现，所以最好是像下面例子所示的这样使用这个方法。

```
function determineOrder(value) {
    var result = stringValue.localeCompare(value);
    if (result < 0){
        alert("The string 'yellow' comes before the string '" + value + "'.");
    } else if (result > 0) {
        alert("The string 'yellow' comes after the string '" + value + "'.");
    } else {
```

```
            alert("The string 'yellow' is equal to the string '" + value + "'.");
        }
    }

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

StringTypeLocaleCompareExample01.htm

使用这种结构，就可以确保自己的代码在任何实现中都可以正确地运行了。

localeCompare()方法比较与众不同的地方，就是实现所支持的地区（国家和语言）决定了这个方法的行为。比如，美国以英语作为 ECMAScript 实现的标准语言，因此 localeCompare()就是区分大小写的，于是大写字母在字母表中排在小写字母前头就成为了一项决定性的比较规则。不过，在其他地区恐怕就不是这种情况了。

8. fromCharCode()方法

另外，String 构造函数本身还有一个静态方法：fromCharCode()。这个方法的任务是接收一或多个字符编码，然后将它们转换成一个字符串。从本质上来看，这个方法与实例方法 charCodeAt()执行的是相反的操作。来看一个例子：

```
alert(String.fromCharCode(104, 101, 108, 108, 111)); //"hello"
```

StringTypeFromCharCodeExample01.htm

在这里，我们给 fromCharCode()传递的是字符串"hello"中每个字母的字符编码。

9. HTML 方法

早期的 Web 浏览器提供商觉察到了使用 JavaScript 动态格式化 HTML 的需求。于是，这些提供商就扩展了标准，实现了一些专门用于简化常见 HTML 格式化任务的方法。下表列出了这些 HTML 方法。不过，需要请读者注意的是，应该尽量不使用这些方法，因为它们创建的标记通常无法表达语义。

方　法	输出结果
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*}

5.7　单体内置对象

ECMA-262 对内置对象的定义是："由 ECMAScript 实现提供的、不依赖于宿主环境的对象，这些对象在 ECMAScript 程序执行之前就已经存在了。"意思就是说，开发人员不必显式地实例化内置对象，因

为它们已经实例化了。前面我们已经介绍了大多数内置对象，例如 Object、Array 和 String。ECMA-262 还定义了两个单体内置对象：Global 和 Math。

5.7.1 Global 对象

Global（全局）对象可以说是 ECMAScript 中最特别的一个对象了，因为不管你从什么角度上看，这个对象都是不存在的。ECMAScript 中的 Global 对象在某种意义上是作为一个终极的“兜底儿对象”来定义的。换句话说，不属于任何其他对象的属性和方法，最终都是它的属性和方法。事实上，没有全局变量或全局函数；所有在全局作用域中定义的属性和函数，都是 Global 对象的属性。本书前面介绍过的那些函数，诸如 isNaN()、isFinite()、parseInt()以及 parseFloat()，实际上全都是 Global 对象的方法。除此之外，Global 对象还包含其他一些方法。

1. URI 编码方法

Global 对象的 encodeURI()和 encodeURIComponent()方法可以对 URI（Uniform Resource Identifiers，通用资源标识符）进行编码，以便发送给浏览器。有效的 URI 中不能包含某些字符，例如空格。而这两个 URI 编码方法就可以对 URI 进行编码，它们用特殊的 UTF-8 编码替换所有无效的字符，从而让浏览器能够接受和理解。

其中，encodeURI()主要用于整个 URI（例如，http://www.wrox.com/illegal value.htm），而 encodeURIComponent()主要用于对 URI 中的某一段（例如前面 URI 中的 illegal value.htm）进行编码。它们的主要区别在于，encodeURI()不会对本身属于 URI 的特殊字符进行编码，例如冒号、正斜杠、问号和井字号；而 encodeURIComponent()则会对它发现的任何非标准字符进行编码。来看下面的例子。

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

//"http://www.wrox.com/illegal%20value.htm#start"
alert(encodeURI(uri));

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

GlobalObjectURIEncodingExample01.htm

使用 encodeURI()编码后的结果是除了空格之外的其他字符都原封不动，只有空格被替换成了%20。而 encodeURIComponent()方法则会使用对应的编码替换所有非字母数字字符。这也正是可以对整个 URI 使用 encodeURI()，而只能对附加在现有 URI 后面的字符串使用 encodeURIComponent()的原因所在。

> 一般来说，我们使用 encodeURIComponent()方法的时候要比使用 encodeURI()更多，因为在实践中更常见的是对查询字符串参数而不是对基础 URI 进行编码。

与 encodeURI()和 encodeURIComponent()方法对应的两个方法分别是 decodeURI()和 decodeURIComponent()。其中，decodeURI()只能对使用 encodeURI()替换的字符进行解码。例如，它可将%20 替换成一个空格，但不会对%23 作任何处理，因为%23 表示井字号（#），而井字号不是使用 encodeURI()替换的。同样地，decodeURIComponent()能够解码使用 encodeURIComponent()编码

的所有字符，即它可以解码任何特殊字符的编码。来看下面的例子：

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

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

//http://www.wrox.com/illegal value.htm#start
alert(decodeURIComponent(uri));
```

GlobalObjectURIDecodingExample01.htm

这里，变量 uri 包含着一个由 encodeURIComponent()编码的字符串。在第一次调用 decodeURI()输出的结果中，只有%20 被替换成了空格。而在第二次调用 decodeURIComponent()输出的结果中，所有特殊字符的编码都被替换成了原来的字符，得到了一个未经转义的字符串（但这个字符串并不是一个有效的 URI）。

> URI 方法 encodeURI()、encodeURIComponent()、decodeURI()和 decodeURIComponent()用于替代已经被 ECMA-262 第 3 版废弃的 escape()和 unescape()方法。URI 方法能够编码所有 Unicode 字符，而原来的方法只能正确地编码 ASCII 字符。因此在开发实践中，特别是在产品级的代码中，一定要使用 URI 方法，不要使用 escape()和 unescape()方法。

2. eval()方法

现在，我们介绍最后一个——大概也是整个 ECMAScript 语言中最强大的一个方法：eval()。eval()方法就像是一个完整的 ECMAScript 解析器，它只接受一个参数，即要执行的 ECMAScript（或 JavaScript）字符串。看下面的例子：

```
eval("alert('hi')");
```

这行代码的作用等价于下面这行代码：

```
alert("hi");
```

当解析器发现代码中调用 eval()方法时，它会将传入的参数当作实际的 ECMAScript 语句来解析，然后把执行结果插入到原位置。通过 eval()执行的代码被认为是包含该次调用的执行环境的一部分，因此被执行的代码具有与该执行环境相同的作用域链。这意味着通过 eval()执行的代码可以引用在包含环境中定义的变量，举个例子：

```
var msg = "hello world";
eval("alert(msg)");     //"hello world"
```

可见，变量 msg 是在 eval()调用的环境之外定义的，但其中调用的 alert()仍然能够显示"hello world"。这是因为上面第二行代码最终被替换成了一行真正的代码。同样地，我们也可以在 eval()调用中定义一个函数，然后再在该调用的外部代码中引用这个函数：

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

显然，函数 sayHi()是在 eval()内部定义的。但由于对 eval()的调用最终会被替换成定义函数的实际代码，因此可以在下一行调用 sayHi()。对于变量也一样：

```
eval("var msg = 'hello world'; ");
alert(msg);     //"hello world"
```

在 eval()中创建的任何变量或函数都不会被提升，因为在解析代码的时候，它们被包含在一个字符串中；它们只在 eval()执行的时候创建。

严格模式下，在外部访问不到 eval()中创建的任何变量或函数，因此前面两个例子都会导致错误。同样，在严格模式下，为 eval 赋值也会导致错误：

```
"use strict";
eval = "hi";    //causes error
```

能够解释代码字符串的能力非常强大，但也非常危险。因此在使用 eval()时必须极为谨慎，特别是在用它执行用户输入数据的情况下。否则，可能会有恶意用户输入威胁你的站点或应用程序安全的代码（即所谓的代码注入）。

3. Global 对象的属性

Global 对象还包含一些属性，其中一部分属性已经在本书前面介绍过了。例如，特殊的值 undefined、NaN 以及 Infinity 都是 Global 对象的属性。此外，所有原生引用类型的构造函数，像 Object 和 Function，也都是 Global 对象的属性。下表列出了 Global 对象的所有属性。

属　性	说　明	属　性	说　明
undefined	特殊值undefined	Date	构造函数Date
NaN	特殊值NaN	RegExp	构造函数RegExp
Infinity	特殊值Infinity	Error	构造函数Error
Object	构造函数Object	EvalError	构造函数EvalError
Array	构造函数Array	RangeError	构造函数RangeError
Function	构造函数Function	ReferenceError	构造函数ReferenceError
Boolean	构造函数Boolean	SyntaxError	构造函数SyntaxError
String	构造函数String	TypeError	构造函数TypeError
Number	构造函数Number	URIError	构造函数URIError

ECMAScript 5 明确禁止给 undefined、NaN 和 Infinity 赋值，这样做即使在非严格模式下也会导致错误。

4. window 对象

ECMAScript 虽然没有指出如何直接访问 Global 对象，但 Web 浏览器都是将这个全局对象作为 window 对象的一部分加以实现的。因此，在全局作用域中声明的所有变量和函数，就都成为了 window 对象的属性。来看下面的例子。

```
var color = "red";

function sayColor(){
    alert(window.color);
}

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

GlobalObjectWindowExample01.htm

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，我们通过window.color来访问color变量，以说明全局变量是window对象的属性。然后，又使用window.sayColor()来直接通过window对象调用这个函数，结果显示在了警告框中。

JavaScript中的window对象除了扮演ECMAScript规定的Global对象的角色外，还承担了很多别的任务。第8章在讨论浏览器对象模型时将详细介绍window对象。

另一种取得Global对象的方法是使用以下代码：

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

以上代码创建了一个立即调用的函数表达式，返回this的值。如前所述，在没有给函数明确指定this值的情况下（无论是通过将函数添加为对象的方法，还是通过调用call()或apply()），this值等于Global对象。而像这样通过简单地返回this来取得Global对象，在任何执行环境下都是可行的。第7章将深入讨论函数表达式。

5.7.2 Math对象

ECMAScript还为保存数学公式和信息提供了一个公共位置，即Math对象。与我们在JavaScript直接编写的计算功能相比，Math对象提供的计算功能执行起来要快得多。Math对象中还提供了辅助完成这些计算的属性和方法。

1. Math对象的属性

Math对象包含的属性大都是数学计算中可能会用到的一些特殊值。下表列出了这些属性。

属　性	说　明
Math.E	自然对数的底数，即常量e的值
Math.LN10	10的自然对数
Math.LN2	2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（即2的平方根的倒数）
Math.SQRT2	2的平方根

虽然讨论这些值的含义和用途超出了本书范围，但你确实可以随时使用它们。

2. min()和max()方法

Math对象还包含许多方法，用于辅助完成简单和复杂的数学计算。

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

```
var max = Math.max(3, 54, 32, 16);
alert(max);     //54

var min = Math.min(3, 54, 32, 16);
alert(min);     //3
```

MathObjectMinMaxExample01.htm

对于 3、54、32 和 16，Math.max()返回 54，而 Math.min()返回 3。这两个方法经常用于避免多余的循环和在 if 语句中确定一组数的最大值。

要找到数组中的最大或最小值，可以像下面这样使用 apply()方法。

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

这个技巧的关键是把 Math 对象作为 apply()的第一个参数，从而正确地设置 this 值。然后，可以将任何数组作为第二个参数。

3. 舍入方法

下面来介绍将小数值舍入为整数的几个方法：Math.ceil()、Math.floor()和 Math.round()。这三个方法分别遵循下列舍入规则：

- Math.ceil()执行向上舍入，即它总是将数值向上舍入为最接近的整数；
- Math.floor()执行向下舍入，即它总是将数值向下舍入为最接近的整数；
- Math.round()执行标准舍入，即它总是将数值四舍五入为最接近的整数（这也是我们在数学课上学到的舍入规则）。

下面是使用这些方法的示例：

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

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

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

MathObjectRoundingExample01.htm

对于所有介于 25 和 26（不包括 26）之间的数值，Math.ceil()始终返回 26，因为它执行的是向上舍入。Math.round()方法只在数值大于等于 25.5 时返回 26；否则返回 25。最后，Math.floor()对所有介于 25 和 26（不包括 26）之间的数值都返回 25。

4. random()方法

Math.random()方法返回介于 0 和 1 之间一个随机数，不包括 0 和 1。对于某些站点来说，这个方法非常实用，因为可以利用它来随机显示一些名人名言和新闻事件。套用下面的公式，就可以利用 Math.random()从某个整数范围内随机选择一个值。

```
值 = Math.floor(Math.random() * 可能值的总数 + 第一个可能的值)
```

公式中用到了 Math.floor()方法，这是因为 Math.random()总返回一个小数值。而用这个小数值乘以一个整数，然后再加上一个整数，最终结果仍然还是一个小数。举例来说，如果你想选择一个 1 到 10 之间的数值，可以像下面这样编写代码：

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

MathObjectRandomExample01.htm

总共有 10 个可能的值（1 到 10），而第一个可能的值是 1。而如果想要选择一个介于 2 到 10 之间的值，就应该将上面的代码改成这样：

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

MathObjectRandomExample02.htm

从 2 数到 10 要数 9 个数，因此可能值的总数就是 9，而第一个可能的值就是 2。多数情况下，其实都可以通过一个函数来计算可能值的总数和第一个可能的值，例如：

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

var num = selectFrom(2, 10);
alert(num);    // 介于 2 和 10 之间（包括 2 和 10）的一个数值
```

MathObjectRandomExample03.htm

函数 selectFrom()接受两个参数：应该返回的最小值和最大值。而用最大值减最小值再加 1 得到了可能值的总数，然后它又把这些数值套用到了前面的公式中。这样，通过调用 selectFrom(2,10)就可以得到一个介于 2 和 10 之间（包括 2 和 10）的数值了。利用这个函数，可以方便地从数组中随机取出一项，例如：

```
var colors = ["red", "green", "blue", "yellow", "black", "purple", "brown"];
var color = colors[selectFrom(0, colors.length-1)];
alert(color);  // 可能是数组中包含的任何一个字符串
```

MathObjectRandomExample03.htm

在这个例子中，传递给 selectFrom()的第二个参数是数组的长度减 1，也就是数组中最后一项的位置。

5. 其他方法

Math 对象中还包含其他一些与完成各种简单或复杂计算有关的方法，但详细讨论其中每一个方法的细节及适用情形超出了本书的范围。下面我们就给出一个表格，其中列出了这些没有介绍到的 Math 对象的方法。

方　法	说　明	方　法	说　明
Math.abs(*num*)	返回*num*的绝对值	Math.asin(*x*)	返回*x*的反正弦值
Math.exp(*num*)	返回*Math.E*的*num*次幂	Math.atan(*x*)	返回*x*的反正切值
Math.log(*num*)	返回*num*的自然对数	Math.atan2(*y*,*x*)	返回*y*/*x*的反正切值
Math.pow(*num*,*power*)	返回*num*的*power*次幂	Math.cos(*x*)	返回*x*的余弦值
Math.sqrt(*num*)	返回*num*的平方根	Math.sin(*x*)	返回*x*的正弦值
Math.acos(*x*)	返回*x*的反余弦值	Math.tan(*x*)	返回*x*的正切值

虽然 ECMA-262 规定了这些方法，但不同实现可能会对这些方法采用不同的算法。毕竟，计算某个值的正弦、余弦和正切的方式多种多样。也正因为如此，这些方法在不同的实现中可能会有不同的精度。

5.8 小结

对象在 JavaScript 中被称为引用类型的值，而且有一些内置的引用类型可以用来创建特定的对象，现简要总结如下：

- 引用类型与传统面向对象程序设计中的类相似，但实现不同；
- `Object` 是一个基础类型，其他所有类型都从 `Object` 继承了基本的行为；
- `Array` 类型是一组值的有序列表，同时还提供了操作和转换这些值的功能；
- `Date` 类型提供了有关日期和时间的信息，包括当前日期和时间以及相关的计算功能；
- `RegExp` 类型是 ECMAScript 支持正则表达式的一个接口，提供了最基本的和一些高级的正则表达式功能。

函数实际上是 `Function` 类型的实例，因此函数也是对象；而这一点正是 JavaScript 最有特色的地方。由于函数是对象，所以函数也拥有方法，可以用来增强其行为。

因为有了基本包装类型，所以 JavaScript 中的基本类型值可以被当作对象来访问。三种基本包装类型分别是：`Boolean`、`Number` 和 `String`。以下是它们共同的特征：

- 每个包装类型都映射到同名的基本类型；
- 在读取模式下访问基本类型值时，就会创建对应的基本包装类型的一个对象，从而方便了数据操作；
- 操作基本类型值的语句一经执行完毕，就会立即销毁新创建的包装对象。

在所有代码执行之前，作用域中就已经存在两个内置对象：`Global` 和 `Math`。在大多数 ECMAScript 实现中都不能直接访问 `Global` 对象；不过，Web 浏览器实现了承担该角色的 `window` 对象。全局变量和函数都是 `Global` 对象的属性。`Math` 对象提供了很多属性和方法，用于辅助完成复杂的数学计算任务。