第7章

数组

数组是值的有序集合。每个值叫做一个元素，而每个元素在数组中有一个位置，以数字表示，称为索引。JavaScript数组是无类型的：数组元素可以是任意类型，并且同一个数组中的不同元素也可能有不同的类型。数组的元素甚至也可能是对象或其他数组，这允许创建复杂的数据结构，如对象的数组和数组的数组。JavaScript数组的索引是基于零的32位数值：第一个元素的索引为0，最大可能的索引为4 294 967 294（$2^{32}-2$），数组最大能容纳4 294 967 295个元素。JavaScript数组是动态的：根据需要它们会增长或缩减，并且在创建数组时无须声明一个固定的大小或者在数组大小变化时无须重新分配空间。JavaScript数组可能是稀疏的：数组元素的索引不一定要连续的，它们之间可以有空缺。每个JavaScript数组都有一个`length`属性。针对非稀疏数组，该属性就是数组元素的个数。针对稀疏数组，`length`比所有元素的索引要大。

JavaScript数组是JavaScript对象的特殊形式，数组索引实际上和碰巧是整数的属性名差不多。我们将在本章的其他地方更多地讨论特殊化的数组。通常，数组的实现是经过优化的，用数字索引来访问数组元素一般来说比访问常规的对象属性要快很多。

数组继承自`Array.prototype`中的属性，它定义了一套丰富的数组操作方法，7.8节和7.9节涵盖这方面内容。大多数这些方法是通用的，这意味着它们不仅对真正的数组有效，而且对“类数组对象”同样有效。7.11节讨论类数组对象。在ECMAScript 5中，字符串的行为与字符数组类似，我们将在7.12节讨论。

7.1 创建数组

使用数组直接量是创建数组最简单的方法，在方括号中将数组元素用逗号隔开即可。例如：

```
var empty = [];                 // 没有元素的数组
var primes = [2, 3, 5, 7, 11];  // 有5个数值的数组
var misc = [ 1.1, true, "a", ]; // 3个不同类型的元素和结尾的逗号
```

数组直接量中的值不一定要是常量；它们可以是任意的表达式：

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

它可以包含对象直接量或其他数组直接量：

```
var b = [[1,{x:1, y:2}], [2, {x:3, y:4}]];
```

如果省略数组直接量中的某个值，省略的元素将被赋予undefined值：

```
var count = [1,,3]; // 数组有3个元素，中间的那个元素值为undefined
var undefs = [,,];  // 数组有2个元素，都是undefined
```

数组直接量的语法允许有可选的结尾的逗号，故[,,]只有两个元素而非三个。

调用构造函数Array()是创建数组的另一种方法。可以用三种方式调用构造函数。

- 调用时没有参数：

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

该方法创建一个没有任何元素的空数组，等同于数组直接量[ ]。

- 调用时有一个数值参数，它指定长度：

```
var a = new Array(10);
```

该技术创建指定长度的数组。当预先知道所需元素个数时，这种形式的Array()构造函数可以用来预分配一个数组空间。注意，数组中没有存储值，甚至数组的索引属性“0”、“1”等还未定义。

- 显式指定两个或多个数组元素或者数组的一个非数值元素：

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

以这种形式，构造函数的参数将会成为新数组的元素。使用数组字面量比这样使用Array()构造函数要简单多了。

7.2 数组元素的读和写

使用[ ]操作符来访问数组中的一个元素。数组的引用位于方括号的左边。方括号中是一

个返回非负整数值的任意表达式。使用该语法既可以读又可以写数组的一个元素。因此，如下代码都是合法的JavaScript语句：

```
var a = ["world"];      // 从一个元素的数组开始
var value = a[0];       // 读第0个元素
a[1] = 3.14;            // 写第1个元素
i = 2;
a[i] = 3;               // 写第2个元素
a[i + 1] = "hello";     // 写第3个元素
a[a[i]] = a[0];         // 读第0个和第2个元素，写第3个元素
```

请记住，数组是对象的特殊形式。使用方括号访问数组元素就像用方括号访问对象的属性一样。JavaScript将指定的数字索引值转换成字符串——索引值1变成“1”——然后将其作为属性名来使用。关于索引值从数字转换为字符串没什么特别之处：对常规对象也可以这么做：

```
o = {};         // 创建一个普通的对象
o[1] = "one";   // 用一个整数来索引它
```

数组的特别之处在于，当使用小于2^{32}的非负整数作为属性名时数组会自动维护其length属性值。如上，创建仅有一个元素的数组。然后在索引1、2和3处分别进行赋值。当我们这么做时数组的length属性值变为：

```
a.length        // => 4
```

清晰地区分数组的索引和对象的属性名是非常有用的。所有的索引都是属性名，但只有在$0 \sim 2^{32}-2$之间的整数属性名才是索引。所有的数组都是对象，可以为其创建任意名字的属性。但如果使用的属性是数组的索引，数组的特殊行为就是将根据需要更新它们的length属性值。

注意，可以使用负数或非整数来索引数组。这种情况下，数值转换为字符串，字符串作为属性名来用。既然名字不是非负整数，它就只能当做常规的对象属性，而非数组的索引。同样，如果凑巧使用了是非负整数的字符串，它就当做数组索引，而非对象属性。当使用的一个浮点数和一个整数相等时情况也是一样的：

```
a[-1.23] = true;  // 这将创建一个名为"-1.23"的属性
a["1000"] = 0;    // 这是数组的第1001个元素
a[1.000]          // 和a[1]相等
```

事实上数组索引仅仅是对象属性名的一种特殊类型，这意味着JavaScript数组没有“越界”错误的概念。当试图查询任何对象中不存在的属性时，不会报错，只会得到undefined值。类似于对象，对于对象同样存在这种情况。

既然数组是对象，那么它们可以从原型中继承元素。在ECMAScript 5中，数组可以定

义元素的getter和setter方法（见6.6节）。如果一个数组确实继承了元素或使用了元素的getter和setter方法，你应该期望它使用非优化的代码路径：访问这种数组的元素的时间会与常规对象属性的查找时间相近。

7.3 稀疏数组

稀疏数组就是包含从0开始的不连续索引的数组。通常，数组的length属性值代表数组中元素的个数。如果数组是稀疏的，length属性值大于元素的个数。可以用Array()构造函数或简单地指定数组的索引值大于当前的数组长度来创建稀疏数组。

```
a = new Array(5);   // 数组没有元素，但是a.length是5
a = [];             // 创建一个空数组，length = 0
a[1000] = 0;        // 赋值添加一个元素，但是设置length为1001
```

后面会看到你也可以用delete操作符来生产稀疏数组。

足够稀疏的数组通常在实现上比稠密的数组更慢、内存利用率更高，在这样的数组中查找元素的时间与常规对象属性的查找时间一样长。

注意，当在数组直接量中省略值时不会创建稀疏数组。省略的元素在数组中是存在的，其值为undefined。这和数组元素根本不存在是有一些微妙的区别的。可以用in操作符检测两者之间的区别：

```
var a1 = [,,,];         // 数组是[undefined, undefined, undefined]
var a2 = new Array(3);  // 该数组根本没有元素
0 in a1                 // => true: a1在索引0处有一个元素
0 in a2                 // => false: a2在索引0处没有元素
```

当使用for/in循环时，a1和a2之间的区别也很明显（见7.6节）。

需要注意的是，当省略数组直接量中的值时（使用连续的逗号，比如[1,,3]），这时所得到的数组也是稀疏数组，省略掉的值是不存在的：

```
var a1 = [,];           // 此数组没有元素，长度是1
var a2 = [undefined];   // 此数组包含一个值为undefined的元素
0 in a1                 // => false: a1在索引0处没有元素
0 in a2                 // => true: a2在索引0处有一个值为undefined的元素
```

在一些旧版本的实现中（比如Firefox 3），在存在连续逗号的情况下，插入undefined值的操作则与此不同，在这些实现中，[1,,3]和[1,undefined,3]是一模一样的。

了解稀疏数组是了解JavaScript数组的真实本质的一部分。尽管如此，实际上你所碰到的

绝大多数JavaScript数组不是稀疏数组。并且，如果你确实碰到了稀疏数组，你的代码很可能像对待非稀疏数组一样来对待它们，只不过它们包含一些undefined值。

7.4 数组长度

每个数组有一个length属性，就是这个属性使其区别于常规的JavaScript对象。针对稠密（也就是非稀疏）数组，length属性值代表数组中元素的个数。其值比数组中最大的索引大1：

```
[].length            // => 0: 数组没有元素
['a','b','c'].length  // => 3: 最大的索引为2, length为3
```

当数组是稀疏的时，length属性值大于元素的个数。而且关于此我们可以说的一切也就是数组长度保证大于它每个元素的索引值。或者，换一种说法，在数组中（无论稀疏与否）肯定找不到一个元素的索引值大于或等于它的长度。为了维持此规则不变化，数组有两个特殊的行为。第一个如同上面的描述：如果为一个数组元素赋值，它的索引i大于或等于现有数组的长度时，length属性的值将设置为i+1。

第二个特殊的行为就是设置length属性为一个小于当前长度的非负整数n时，当前数组中那些索引值大于或等于n的元素将从中删除：

```
a = [1,2,3,4,5];     // 从5个元素的数组开始
a.length = 3;        // 现在a为[1,2,3]
a.length = 0;        // 删除所有的元素。a为[ ]
a.length = 5;        // 长度为5，但是没有元素，就像new Array(5)
```

还可以将数组的length属性值设置为大于其当前的长度。实际上这不会向数组中添加新的元素，它只是在数组尾部创建一个空的区域。

在ECMAScript 5中，可以用Object.defineProperty()让数组的length属性变成只读的（见6.7节）：

```
a = [1,2,3];                                    // 从3个元素的数组开始
Object.defineProperty(a, "length",              // 让length属性只读
                      {writable: false});
a.length = 0;                                   // a不会改变
```

类似地，如果让一个数组元素不能配置，就不能删除它。如果不能删除它，length属性不能设置为小于不可配置元素的索引值。（见6.7节和6.8.3节的Object.seal()和Object.freeze()方法。）

7.5 数组元素的添加和删除

我们已经见过添加数组元素最简单的方法：为新索引赋值：

```
a = []           // 开始是一个空数组
a[0] = "zero";   // 然后向其中添加元素
a[1] = "one";
```

也可以使用push()方法在数组末尾增加一个或多个元素：

```
a = [];              // 开始是一个空数组
a.push("zero")       // 在末尾添加一个元素。a = ["zero"]
a.push("one", "two") // 再添加两个元素。a = ["zero", "one", "two"]
```

在数组尾部压入一个元素与给数组a[a.length]赋值是一样的。可以使用unshift()方法（在7.8节有描述）在数组的首部插入一个元素，并且将其他元素依次移到更高的索引处。

可以像删除对象属性一样使用delete运算符来删除数组元素：

```
a = [1,2,3];
delete a[1];   // a在索引1的位置不再有元素
1 in a         // => false: 数组索引1并未在数组中定义
a.length       // => 3: delete操作并不影响数组长度
```

删除数组元素与为其赋undefined值是类似的（但有一些微妙的区别）。注意，对一个数组元素使用delete不会修改数组的length属性，也不会将元素从高索引处移下来填充已删除属性留下的空白。如果从数组中删除一个元素，它就变成稀疏数组。

上面我们看到，也可以简单地设置length属性为一个新的期望长度来删除数组尾部的元素。数组有pop()方法（它和push()一起使用），后者一次使减少长度1并返回被删除元素的值。还有一个shift()方法（它和unshift()一起使用），从数组头部删除一个元素。和delete不同的是shift()方法将所有元素下移到比当前索引低1的地方。7.8节和第三部分涵盖pop()和shift()的内容。

最后，splice()是一个通用的方法来插入、删除或替换数组元素。它会根据需要修改length属性并移动元素到更高或较低的索引处。详细内容见7.8节。

7.6 数组遍历

使用for循环（见5.5.3节）是遍历数组元素最常见的方法：

```
var keys = Object.keys(o);   // 获得o对象属性名组成的数组
var values = []              // 在数组中存储匹配属性的值
for(var i = 0; i < keys.length; i++) {  // 对于数组中每个索引
```

```
    var key = keys[i];                  // 获得索引处的键值
    values[i] = o[key];                 // 在values数组中保存属性值
}
```

在嵌套循环或其他性能非常重要的上下文中，可以看到这种基本的数组遍历需要优化，数组的长度应该只查询一次而非每次循环都要查询：

```
for(var i = 0, len = keys.length; i < len; i++) {
   // 循环体仍然不变
}
```

这些例子假设数组是稠密的，并且所有的元素都是合法数据。否则，使用数组元素之前应该先检测它们。如果想要排除null、undefined和不存在的元素，代码如下：

```
for(var i = 0; i < a.length; i++) {
    if (!a[i]) continue;  // 跳过null、undefined和不存在的元素
    // 循环体
}
```

如果只想跳过undefined和不存的元素，代码如下：

```
for(var i = 0; i < a.length; i++) {
    if (a[i] === undefined) continue; // 跳过undefined+不存在的元素
    // 循环体
}
```

最后，如果只想跳过不存在的元素而仍然要处理存在的undefined元素，代码如下：

```
for(var i = 0; i < a.length; i++) {
    if (!(i in a)) continue ; // 跳过不存在的元素
    // 循环体
}
```

还可以使用for/in循环（见5.5.4节）处理稀疏数组。循环每次将一个可枚举的属性名（包括数组索引）赋值给循环变量。不存在的索引将不会遍历到：

```
for(var index in sparseArray) {
   var value = sparseArray[index];
   // 此处可以使用索引和值做一些事情
}
```

在6.5节已经注意到for/in循环能够枚举继承的属性名，如添加到Array.prototype中的方法。由于这个原因，在数组上不应该使用for/in循环，除非使用额外的检测方法来过滤不想要的属性。如下检测代码取其一即可：

```
for(var i in a) {
    if (!a.hasOwnProperty(i)) continue;  // 跳过继承的属性
    // 循环体
```

```
}
for(var i in a) {
    // 跳过不是非负整数的i
    if (String(Math.floor(Math.abs(Number(i)))) !== i) continue;
}
```

ECMAScript规范允许for/in循环以不同的顺序遍历对象的属性。通常数组元素的遍历实现是升序的，但不能保证一定是这样的。特别地，如果数组同时拥有对象属性和数组元素，返回的属性名很可能是按照创建的顺序而非数值的大小顺序。如何处理这个问题的实现各不相同，如果算法依赖于遍历的顺序，那么最好不要使用for/in而用常规的for循环。

ECMAScript 5定义了一些遍历数组元素的新方法，按照索引的顺序按个传递给定义的一个函数。这些方法中最常用的就是forEach()方法：

```
var data = [1,2,3,4,5];     // 这是需要遍历的数组
var sumOfSquares = 0;       // 要得到数据的平方和
data.forEach(function(x) {  // 把每个元素传递给此函数
                sumOfSquares += x*x;  // 平方相加
             });
sumOfSquares                // =>55 : 1+4+9+16+25
```

forEach()和相关的遍历方法使得数组拥有简单而强大的函数式编程风格。它们涵盖在7.9节中，当涉及函数式编程时，还将在8.8节再次碰到它们。

7.7 多维数组

JavaScript不支持真正的多维数组，但可以用数组的数组来近似。访问数组的数组中的元素，只要简单地使用两次[]操作符即可。例如，假设变量matrix是一个数组的数组，它的基本元素是数值，那么matrix[x]的每个元素是包含一个数值数组，访问数组中特定数值的代码为matrix[x][y]。这里有一个具体的例子，它使用二维数组作为一个九九乘法表：

```
// 创建一个多维数组
var table = new Array(10);                  // 表格有10行
for(var i = 0; i < table.length; i++)
    table[i] = new Array(10);               // 每行有10列
// 初始化数组
for(var row = 0; row < table.length; row++) {
    for(col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}
// 使用多维数组来计算（查询）5*7
var product = table[5][7];  // 35
```

7.8 数组方法

ECMAScript 3在Array.prototype中定义了一些很有用的操作数组的函数，这意味着这些函数作为任何数组的方法都是可用的。下面几节介绍ECMAScript 3中的这些方法。像通常一样，完整的细节参见第四部分关于数组的内容。ECMAScript 5中新增了一些新的数组遍历方法；它们涵盖在7.9节中。

7.8.1 join()

Array.join()方法将数组中所有元素都转化为字符串并连接在一起，返回最后生成的字符串。可以指定一个可选的字符串在生成的字符串中来分隔数组的各个元素。如果不指定分隔符，默认使用逗号。如以下代码所示：

```
var a = [1, 2, 3];     // 创建一个包含三个元素的数组
a.join();              // => "1,2,3"
a.join(" ");           // => "1 2 3"
a.join("");            // => "123"
var b = new Array(10); // 长度为10的空数组
b.join('-')            // => '---------': 9个连字号组成的字符串
```

Array.join()方法是String.split()方法的逆向操作，后者是将字符串分割成若干块来创建一个数组。

7.8.2 reverse()

Array.reverse()方法将数组中的元素颠倒顺序，返回逆序的数组。它采取了替换；换句话说，它不通过重新排列的元素创建新的数组，而是在原先的数组中重新排列它们。例如，下面的代码使用reverse()和join()方法生成字符串“3，2，1”：

```
var a = [1,2,3];
a.reverse().join()  // => "3,2,1",并且现在的a是[3,2,1]
```

7.8.3 sort()

Array.sort()方法将数组中的元素排序并返回排序后的数组。当不带参数调用sort()时，数组元素以字母表顺序排序（如有必要将临时转化为字符串进行比较）：

```
var a = new Array("banana", "cherry", "apple");
a.sort();
var s = a.join(", ");  // s == "apple, banana, cherry"
```

如果数组包含undefined元素，它们会被排到数组的尾部。

为了按照其他方式而非字母表顺序进行数组排序，必须给sort()方法传递一个比较函数。该函数决定了它的两个参数在排好序的数组中的先后顺序。假设第一个参数应该在前，比较函数应该返回一个小于0的数值。反之，假设第一个参数应该在后，函数应该返回一个大于0的数值。并且，假设两个值相等（也就是说，它们的顺序无关紧要），函数应该返回0。因此，例如，用数值大小而非字母表顺序进行数组排序，代码如下：

```
var a = [33, 4, 1111, 222];
a.sort();                 // 字母表顺序:  1111, 222, 33, 4
a.sort(function(a,b) {    // 数值顺序: 4, 33, 222, 1111
          return a-b;     // 根据顺序，返回负数、0、正数
       });
a.sort(function(a,b) {return b-a});   // 数值大小相反的顺序
```

注意，这里使用匿名函数表达式非常方便。既然比较函数只使用一次，就没必要给它们命名了。

另外一个数组元素排序的例子，也许需要对一个字符串数组执行不区分大小写的字母表排序，比较函数首先将参数都转化为小写字符串（使用toLowerCase()方法），再开始比较：

```
a = ['ant', 'Bug', 'cat', 'Dog']
a.sort();                 // 区分大小写的排序：['Bug','Dog','ant',cat']
a.sort(function(s,t) {    // 不区分大小写的排序
          var a = s.toLowerCase();
          var b = t.toLowerCase();
          if (a < b) return -1;
          if (a > b) return 1;
          return 0;
       });                // => ['ant','Bug','cat','Dog']
```

7.8.4 concat()

Array.concat()方法创建并返回一个新数组，它的元素包括调用concat()的原始数组的元素和concat()的每个参数。如果这些参数中的任何一个自身是数组，则连接的是数组的元素，而非数组本身。但要注意，concat()不会递归扁平化数组的数组。concat()也不会修改调用的数组。下面有一些示例：

```
var a = [1,2,3];
a.concat(4, 5)          // 返回[1,2,3,4,5]
a.concat([4,5]);        // 返回[1,2,3,4,5]
a.concat([4,5],[6,7])   // 返回[1,2,3,4,5,6,7]
a.concat(4, [5,[6,7]])  // 返回[1,2,3,4,5,[6,7]]
```

7.8.5 slice()

Array.slice()方法返回指定数组的一个片段或子数组。它的两个参数分别指定了片段

的开始和结束的位置。返回的数组包含第一个参数指定的位置和所有到但不含第二个参数指定的位置之间的所有数组元素。如果只指定一个参数，返回的数组将包含从开始位置到数组结尾的所有元素。如参数中出现负数，它表示相对于数组中最后一个元素的位置。例如，参数-1指定了最后一个元素，而-3指定了倒数第三个元素。注意，slice()不会修改调用的数组。下面有一些示例：

```
var a = [1,2,3,4,5];
a.slice(0,3);    // 返回[1,2,3]
a.slice(3);      // 返回[4,5]
a.slice(1,-1);   // 返回[2,3,4]
a.slice(-3,-2);  // 返回[3]
```

7.8.6 splice()

Array.splice()方法是在数组中插入或删除元素的通用方法。不同于slice()和concat()，splice()会修改调用的数组。注意，splice()和slice()拥有非常相似的名字，但它们的功能却有本质的区别。

splice()能够从数组中删除元素、插入元素到数组中或者同时完成这两种操作。在插入或删除点之后的数组元素会根据需要增加或减小它们的索引值，因此数组的其他部分仍然保持连续的。splice()的第一个参数指定了插入和（或）删除的起始位置。第二个参数指定了应该从数组中删除的元素的个数。如果省略第二个参数，从起始点开始到数组结尾的所有元素都将被删除。splice()返回一个由删除元素组成的数组，或者如果没有删除元素就返回一个空数组。例如：

```
var a = [1,2,3,4,5,6,7,8];
a.splice(4);    // 返回[5,6,7,8]; a 是[1,2,3,4]
a.splice(1,2);  // 返回[2,3]; a 是[1,4]
a.splice(1,1);  // 返回[4]; a 是[1]
```

splice()的前两个参数指定了需要删除的数组元素。紧随其后的任意个数的参数指定了需要插入到数组中的元素，从第一个参数指定的位置开始插入。例如：

```
var a = [1,2,3,4,5];
a.splice(2,0,'a','b');  // 返回[]; a是[1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3);  // 返回['a','b']; a是[1,2,[1,2],3,3,4,5]
```

注意，区别于concat()，splice()会插入数组本身而非数组的元素。

7.8.7 push()和pop()

push()和pop()方法允许将数组当做栈来使用。push()方法在数组的尾部添加一个或多个元素，并返回数组新的长度。pop()方法则相反：它删除数组的最后一个元素，减小数组

长度并返回它删除的值。注意，两个方法都修改并替换原始数组而非生成一个修改版的新数组。组合使用push()和pop()能够用JavaScript数组实现先进后出的栈。例如：

```
var stack = [];       // stack: []
stack.push(1,2);      // stack: [1,2]      返回2
stack.pop();          // stack: [1]        返回2
stack.push(3);        // stack: [1,3]      返回2
stack.pop();          // stack: [1]        返回3
stack.push([4,5]);    // stack: [1,[4,5]]  返回2
stack.pop()           // stack: [1]        返回[4,5]
stack.pop();          // stack: []         返回1
```

7.8.8 unshift()和shift()

unshift()和shift()方法的行为非常类似于push()和pop()，不一样的是前者是在数组的头部而非尾部进行元素的插入和删除操作。unshift()在数组的头部添加一个或多个元素，并将已存在的元素移动到更高索引的位置来获得足够的空间，最后返回数组新的长度。shift()删除数组的第一个元素并将其返回，然后把所有随后的元素下移一个位置来填补数组头部的空缺。例如：

```
var a = [];            // a:[]
a.unshift(1);          // a:[1]         返回: 1
a.unshift(22);         // a:[22,1]      返回: 2
a.shift();             // a:[1]         返回: 22
a.unshift(3,[4,5]);    // a:[3,[4,5],1] 返回: 3
a.shift();             // a:[[4,5],1]   返回: 3
a.shift();             // a:[1]         返回: [4,5]
a.shift();             // a:[]          返回: 1
```

注意，当使用多个参数调用unshift()时它的行为令人惊讶。参数是一次性插入的（就像splice()方法）而非一次一个地插入。这意味着最终的数组中插入的元素的顺序和它们在参数列表中的顺序一致。而假如元素是一次一个地插入，它们的顺序应该是反过来的。

7.8.9 toString()和toLocaleString()

数组和其他JavaScript对象一样拥有toString()方法。针对数组，该方法将其每个元素转化为字符串（如有必要将调用元素的toString()方法）并且输出用逗号分隔的字符串列表。注意，输出不包括方括号或其他任何形式的包裹数组值的分隔符。例如：

```
[1,2,3].toString()          // 生成'1,2,3'
["a", "b", "c"].toString()  // 生成'a,b,c'
[1, [2,'c']].toString()     // 生成'1,2,c'
```

注意，这里与不使用任何参数调用join()方法返回的字符串是一样的。

toLocaleString()是toString()方法的本地化版本。它调用元素的toLocaleString()方法将每个数组元素转化为字符串，并且使用本地化（和自定义实现的）分隔符将这些字符串连接起来生成最终的字符串。

7.9 ECMAScript 5中的数组方法

ECMAScript 5定义了9个新的数组方法来遍历、映射、过滤、检测、简化和搜索数组。下面几节描述了这些方法。

但在开始详细介绍之前，很有必要对ECMAScript 5中的数组方法做一个概述。首先，大多数方法的第一个参数接收一个函数，并且对数组的每个元素（或一些元素）调用一次该函数。如果是稀疏数组，对不存在的元素不调用传递的函数。在大多数情况下，调用提供的函数使用三个参数：数组元素、元素的索引和数组本身。通常，只需要第一个参数值，可以忽略后两个参数。大多数ECMAScript 5数组方法的第一个参数是一个函数，第二个参数是可选的。如果有第二个参数，则调用的函数被看做是第二个参数的方法。也就是说，在调用函数时传递进去的第二个参数作为它的this关键字的值来使用。被调用的函数的返回值非常重要， 但是不同的方法处理返回值的方式也不一样。ECMAScript 5中的数组方法都不会修改它们调用的原始数组。当然，传递给这些方法的函数是可以修改这些数组的。

7.9.1 forEach()

forEach()方法从头至尾遍历数组，为每个元素调用指定的函数。如上所述，传递的函数作为forEach()的第一个参数。然后forEach()使用三个参数调用该函数：数组元素、元素的索引和数组本身。如果只关心数组元素的值，可以编写只有一个参数的函数——额外的参数将忽略：

```
var data = [1,2,3,4,5];                          // 要求和的数组
// 计算数组元素的和值
var sum = 0;                                     // 初始为0
data.forEach(function(value) { sum += value; }); // 将每个值累加到sum上
sum                                              // => 15
// 每个数组元素的值自加1
data.forEach(function(v, i, a) { a[i] = v + 1; });
data                                             // => [2,3,4,5,6]
```

注意，forEach()无法在所有元素都传递给调用的函数之前终止遍历。也就是说，没有像for循环中使用的相应的break语句。如果要提前终止，必须把forEach()方法放在一个try块中，并能抛出一个异常。如果forEach()调用的函数抛出foreach.break异常，循环会提前终止：

```
function foreach(a,f,t) {
    try { a.forEach(f,t); }
    catch(e) {
        if (e === foreach.break) return;
        else throw e;
    }
}
foreach.break = new Error("StopIteration");
```

7.9.2 map()

map()方法将调用的数组的每个元素传递给指定的函数，并返回一个数组，它包含该函数的返回值。例如：

```
a = [1, 2, 3];
b = a.map(function(x) { return x*x; });  // b是[1, 4, 9]
```

传递给map()的函数的调用方式和传递给forEach()的函数的调用方式一样。但传递给map()的函数应该有返回值。注意，map()返回的是新数组：它不修改调用的数组。如果是稀疏数组，返回的也是相同方式的稀疏数组：它具有相同的长度，相同的缺失元素。

7.9.3 filter()

fliter()方法返回的数组元素是调用的数组的一个子集。传递的函数是用来逻辑判定的：该函数返回true或false。调用判定函数就像调用forEach()和map()一样。如果返回值为true或能转化为true的值，那么传递给判定函数的元素就是这个子集的成员，它将被添加到一个作为返回值的数组中。例如：

```
a = [5, 4, 3, 2, 1];
smallvalues = a.filter(function(x) { return x < 3 });   // [2, 1]
everyother = a.filter(function(x,i) { return i%2==0 }); // [5, 3, 1]
```

注意，filter()会跳过稀疏数组中缺少的元素，它的返回数组总是稠密的。为了压缩稀疏数组的空缺，代码如下：

```
var dense = sparse.filter(function() { return true; });
```

甚至，压缩空缺并删除undefined和null元素，可以这样使用filter()：

```
a = a.filter(function(x) { return x !== undefined && x != null; });
```

7.9.4 every()和some()

every()和some()方法是数组的逻辑判定：它们对数组元素应用指定的函数进行判定，返回true或false。

every()方法就像数学中的“针对所有”的量词∀：当且仅当针对数组中的所有元素调用判定函数都返回true，它才返回true：

```
a = [1,2,3,4,5];
a.every(function(x) { return x < 10; })      // => true: 所有的值<10
a.every(function(x) { return x % 2 === 0; }) // => false: 不是所有的值都是偶数
```

some()方法就像数学中的“存在”的量词∃：当数组中至少有一个元素调用判定函数返回true，它就返回true；并且当且仅当数值中的所有元素调用判定函数都返回false，它才返回false：

```
a = [1,2,3,4,5];
a.some(function(x) { return x%2===0; })  // => true:  a含有偶数值
a.some(isNaN)                            // => false: a不包含非数值元素
```

注意，一旦every()和some()确认该返回什么值它们就会停止遍历数组元素。some()在判定函数第一次返回true后就返回true，但如果判定函数一直返回false，它将会遍历整个数组。every()恰好相反：它在判定函数第一次返回false后就返回false，但如果判定函数一直返回true，它将会遍历整个数组。注意，根据数学上的惯例，在空数组上调用时，every()返回true，some()返回false。

7.9.5 reduce()和reduceRight()

reduce()和reduceRight()方法使用指定的函数将数组元素进行组合，生成单个值。这在函数式编程中是常见的操作，也可以称为“注入”和“折叠”。举例说明它是如何工作的：

```
var a = [1,2,3,4,5]
var sum = a.reduce(function(x,y) { return x+y }, 0);     // 数组求和
var product = a.reduce(function(x,y) { return x*y }, 1); // 数组求积
var max = a.reduce(function(x,y) { return (x>y)?x:y; }); // 求最大值
```

reduce()需要两个参数。第一个是执行化简操作的函数。化简函数的任务就是用某种方法把两个值组合或化简为一个值，并返回化简后的值。在上述例子中，函数通过加法、乘法或取最大值的方法组合两个值。第二个（可选）的参数是一个传递给函数的初始值。

reduce()使用的函数与forEach()和map()使用的函数不同。比较熟悉的是，数组元素、元素的索引和数组本身将作为第2~4个参数传递给函数。第一个参数是到目前为止的化简操作累积的结果。第一次调用函数时，第一个参数是一个初始值，它就是传递给reduce()的第二个参数。在接下来的调用中，这个值就是上一次化简函数的返回值。在上面的第一个例子中，第一次调用化简函数时的参数是0和1。将两者相加并返回1。再次调用时的参数是1和2，它返回3。然后它计算3+3=6、6+4=10，最后计算10+5=15。最后的值是15，reduce()返回这个值。

可能已经注意到了，上面第三次调用reduce()时只有一个参数：没有指定初始值。当不指定初始值调用reduce()时，它将使用数组的第一个元素作为其初始值。这意味着第一次调用化简函数就使用了第一个和第二个数组元素作为其第一个和第二个参数。在上面求和与求积的例子中，可以省略初始值参数。

在空数组上，不带初始值参数调用reduce()将导致类型错误异常。如果调用它的时候只有一个值——数组只有一个元素并且没有指定初始值，或者有一个空数组并且指定一个初始值——reduce()只是简单地返回那个值而不会调用化简函数。

reduceRight()的工作原理和reduce()一样，不同的是它按照数组索引从高到低（从右到左）处理数组，而不是从低到高。如果化简操作的优先顺序是从右到左，你可能想使用它，例如：

```
var a = [2, 3, 4]
// 计算2^(3^4)。乘方操作的优先顺序是从右到左
var big = a.reduceRight(function(accumulator,value) {
                            return Math.pow(value,accumulator);
                        });
```

注意，reduce()和reduceRight()都能接收一个可选的参数，它指定了化简函数调用时的this关键字的值。可选的初始值参数仍然需要占一个位置。如果想让化简函数作为一个特殊对象的方法调用，请参看Function.bind()方法。

值得注意的是，上面描述的every()和some()方法是一种类型的数组化简操作。但是不同的是，它们会尽早终止遍历而不总是访问每一个数组元素。

为了简单起见，到目前位置所展示的例子都是数值的，但数学计算不是reduce()和reduceRight()的唯一意图。考虑一下例6-2中的union()函数。它计算两个对象的“并集”，并返回另一个新对象，新对象具有二者的属性。该函数期待两个对象并返回另一个对象，所以它的工作原理和一个化简函数一样，并且可以使用reduce()来把它一般化，计算任意数目的对象的“并集”。

```
var objects = [{x:1}, {y:2}, {z:3}];
var merged = objects.reduce(union);    // => {x:1, y:2, z:3}
```

回想一下，当两个对象拥有同名的属性时，union()函数使用第一个参数的属性值。这样，reduce()和reduceRight()在使用union()时给出了不同的结果：

```
var objects = [{x:1,a:1}, {y:2,a:2}, {z:3,a:3}];
var leftunion = objects.reduce(union);       // {x:1, y:2, z:3, a:1}
var rightunion = objects.reduceRight(union); // {x:1, y:2, z:3, a:3}
```

7.9.6 indexOf()和lastIndexOf()

indexOf()和lastIndexOf()搜索整个数组中具有给定值的元素，返回找到的第一个元素的索引或者如果没有找到就返回-1。indexOf()从头至尾搜索，而lastIndexOf()则反向搜索。

```
a = [0,1,2,1,0];
a.indexOf(1)       // => 1: a[1]是1
a.lastIndexOf(1)   // => 3: a[3]是1
a.indexOf(3)       // => -1: 没有值为3的元素
```

不同于本节描述的其他方法，indexOf()和lastIndexOf()方法不接收一个函数作为其参数。第一个参数是需要搜索的值，第二个参数是可选的：它指定数组中的一个索引，从那里开始搜索。如果省略该参数，indexOf()从头开始搜索，而lastIndexOf()从末尾开始搜索。第二个参数也可以是负数，它代表相对数组末尾的偏移量，对于splice()方法：例如，-1指定数组的最后一个元素。

如下函数在一个数组中搜索指定的值并返回包含所有匹配的数组索引的一个数组。它展示了如何运用indexOf()的第二个参数来查找除了第一个以外匹配的值。

```
// 在数组中查找所有出现的x，并返回一个包含匹配索引的数组
function findall(a, x) {
    var results = [],            // 将会返回的数组
        len = a.length,          // 待搜索数组的长度
        pos = 0;                 // 开始搜索的位置
    while(pos < len) {           // 循环搜索多个元素...
        pos = a.indexOf(x, pos); // 搜索
        if (pos === -1) break;   // 未找到，就完成搜索
        results.push(pos);       // 否则，在数组中存储索引
        pos = pos + 1;           // 并从下一个位置开始搜索
    }
    return results;              // 返回包含索引的数组
}
```

注意，字符串也有indexOf()和lastIndexOf()方法，它们和数组方法的功能类似。

7.10 数组类型

我们在本章中到处都可以看见数组是具有特殊行为的对象。给定一个未知的对象，判定它是否为数组通常非常有用。在ECMAScript 5中，可以使用Array.isArray()函数来做这件事情：

```
Array.isArray([])     // => true
Array.isArray({})     // => false
```

但是，在ECMAScript 5以前，要区分数组和非数组对象却令人惊讶地困难。typeof操作

符在这里帮不上忙：对数组它返回“对象”（并且对于除了函数以外的所有对象都是如此）。instanceof操作符只能用于简单的情形：

```
[] instanceof Array      // => true
({}) instanceof Array    // => false
```

使用instanceof的问题是在Web浏览器中有可能有多个窗口或窗体（frame）存在。每个窗口都有自己的JavaScript环境，有自己的全局对象。并且，每个全局对象有自己的一组构造函数。因此一个窗体中的对象将不可能是另外窗体中的构造函数的实例。窗体之间的混淆不常发生，但这个问题足已证明instanceof操作符不能视为一个可靠的数组检测方法。

解决方案是检查对象的类属性（见6.8.2节）。对数组而言该属性的值总是“Array”，因此在ECMAScript 3中isArray()函数的代码可以这样书写：

```
var isArray = Function.isArray || function(o) {
    return typeof o === "object" &&
    Object.prototype.toString.call(o) === "[object Array]";
};
```

实际上，此处类属性的检测就是ECMAScript 5中Array.isArray()函数所做的事情。获得对象类属性的技术使用了6.8.2节和例6-4中展示的Object.prototype.toString()方法。

7.11 类数组对象

我们已经看到，JavaScript数组的有一些特性是其他对象所没有的：

- 当有新的元素添加到列表中时，自动更新length属性。
- 设置length为一个较小值将截断数组。
- 从Array.prototype中继承一些有用的方法。
- 其类属性为“Array”。

这些特性让JavaScript数组和常规的对象有明显的区别。但是它们不是定义数组的本质特性。一种常常完全合理的看法把拥有一个数值length属性和对应非负整数属性的对象看做一种类型的数组。

实践中这些“类数组”对象实际上偶尔出现，虽然不能在它们之上直接调用数组方法或者期望length属性有什么特殊的行为，但是仍然可以用针对真正数组遍历的代码来遍历它们。结论就是很多数组算法针对类数组对象工作得很好，就像针对真正的数组一样。如果算法把数组看成只读的或者如果它们至少保持数组长度不变，也尤其是这种情况。

以下代码为一个常规对象增加了一些属性使其变成类数组对象，然后遍历生成的伪数组的“元素”：

```
var a = {};  // 从一个常规空对象开始
// 添加一些属性，称为"类数组"
var i = 0;
while(i < 10) {
    a[i] = i * i;
    i++;
}
a.length = i;

// 现在，当做真正的数组遍历它
var total = 0;
for(var j = 0; j < a.length; j++)
    total += a[j];
```

8.3.2节描述的Arguments对象就是一个类数组对象。在客户端JavaScript中，一些DOM方法（如document.getElementsByTagName()）也返回类数组对象。下面有一个函数可以用来检测类数组对象：

```
// 判定o是否是一个类数组对象
// 字符串和函数有length属性，但是它们
// 可以用typeof检测将其排除。在客户端JavaScript中，DOM文本节点
// 也有length属性，需要用额外判断o.nodeType != 3将其排除
function isArrayLike(o) {
    if (o &&                                    // o非null、undefined等
        typeof o === "object" &&                // o是对象
        isFinite(o.length) &&                   // o.length是有限数值
        o.length >= 0 &&                        // o.length为非负值
        o.length===Math.floor(o.length) &&      // o.length是整数
        o.length < 4294967296)                  // o.length < 2^32
        return true;                            // o是类数组对象
    else
        return false;                           // 否则它不是
}
```

将在7.12节中看到在ECMAScript 5中字符串的行为与数组类似（并且有些浏览器在ECMAScript 5之前已经让字符串变成可索引的了）。然而，类似上述的类数组对象的检测方法针对字符串常常返回false——它们通常最好当做字符串处理，而非数组。

JavaScript数组方法是特意定义为通用的，因此它们不仅应用在真正的数组而且在类数组对象上都能正确工作。在ECMAScript 5中，所有的数组方法都是通用的。在ECMAScript 3中，除了toString()和toLocaleString()以外的所有方法也是通用的。（concat()方法是一个特例：虽然可以用在类数组对象上，但它没有将那个对象扩充进返回的数组中。）既然类数组对象没有继承自Array.prototype，那就不能在它们上面直接调用数组方法。尽管如此，可以间接地使用Function.call方法调用：

```
var a = {"0":"a", "1":"b", "2":"c", length:3};  // 类数组对象
Array.prototype.join.call(a, "+")  // => "a+b+c"
Array.prototype.slice.call(a, 0)   // => ["a","b","c"]: 真正数组的副本
Array.prototype.map.call(a, function(x) {
    return x.toUpperCase();
})                                 // => ["A","B","C"]:
```

在7.10节的isArray()方法之前我们就已经见过call()技术。8.7.3节涵盖关于Function对象的call()方法的更多内容。

ECMAScript 5数组方法是在Firefox 1.5中引入的。由于它们的写法的一般性，Firefox还将这些方法的版本在Array构造函数上直接定义为函数。使用这些方法定义的版本，上述例子就可以这样重写：

```
var a = {"0":"a", "1":"b", "2":"c", length:3};  // 类数组对象
Array.join(a, "+")
Array.slice(a, 0)
Array.map(a, function(x) { return x.toUpperCase(); })
```

当用在类数组对象上时，数组方法的静态函数版本非常有用。但既然它们不是标准的，不能期望它们在所有的浏览器中都有定义。可以这样书写代码来保证使用它们之前是存在的：

```
Array.join = Array.join || function(a,sep) {
    return Array.prototype.join.call(a,sep);
};
Array.slice = Array.slice || function(a,from,to) {
    return Array.prototype.slice.call(a,from,to);
};
Array.map = Array.map || function(a, f, thisArg) {
    return Array.prototype.map.call(a, f, thisArg);
}
```

7.12 作为数组的字符串

在ECMAScript 5（在众多最近的浏览器实现——包括IE8——早于ECMAScript 5）中，字符串的行为类似于只读的数组。除了用charAt()方法来访问单个的字符以外，还可以使用方括号：

```
var s = test;
s.charAt(0)    // => "t"
s[1]           // => "e"
```

当然，针对字符串的typeof操作符仍然返回“string”，但是如果给Array.isArray()传递字符串，它将返回false。

可索引的字符串的最大的好处就是简单，用方括号代替了charAt()调用，这样更加简洁、可读并且可能更高效。不仅如此，字符串的行为类似于数组的事实使得通用的数组方法可以应用到字符串上。例如：

```
s = "JavaScript"
Array.prototype.join.call(s, " ")      // => "J a v a S c r i p t"
Array.prototype.filter.call(s,         // 过滤字符串中的字符
    function(x) {
        return x.match(/[^aeiou]/);    // 只匹配非元音字母
    }).join("")                        // => "JvScrpt"
```

请记住，字符串是不可变值，故当把它们作为数组看待时，它们是只读的。如push()、sort()、reverse()和splice()等数组方法会修改数组，它们在字符串上是无效的。不仅如此，使用数组方法来修改字符串会导致错误：出错的时候没有提示。