所有字符串组合起来，以逗号分隔。最后一行代码直接用 alert()显示数组，因为 alert()期待字符串，所以会在后台调用数组的 toString()方法，从而得到跟前面一样的结果。

toLocaleString()方法也可能返回跟 toString()和 valueOf()相同的结果，但也不一定。在调用数组的 toLocaleString()方法时，会得到一个逗号分隔的数组值的字符串。它与另外两个方法唯一的区别是，为了得到最终的字符串，会调用数组每个值的 toLocaleString()方法，而不是 toString()方法。看下面的例子：

```
let person1 = {
  toLocaleString() {
    return "Nikolaos";
  },

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

let person2 = {
  toLocaleString() {
    return "Grigorios";
  },

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

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

这里定义了两个对象 person1 和 person2，它们都定义了 toString()和 toLocaleString()方法，而且返回不同的值。然后又创建了一个包含这两个对象的数组 people。在将数组传给 alert()时，输出的是"Nicholas,Greg"，这是因为会在数组每一项上调用 toString()方法（与下一行显式调用 toString()方法结果一样）。而在调用数组的 toLocaleString()方法时，结果变成了"Nikolaos,Grigorios"，这是因为调用了数组每一项的 toLocaleString()方法。

继承的方法 toLocaleString()以及 toString()都返回数组值的逗号分隔的字符串。如果想使用不同的分隔符，则可以使用 join()方法。join()方法接收一个参数，即字符串分隔符，返回包含所有项的字符串。来看下面的例子：

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

这里在 colors 数组上调用了 join()方法，得到了与调用 toString()方法相同的结果。传入逗号，结果就是逗号分隔的字符串。最后一行给 join()传入了双竖线，得到了字符串"red||green||blue"。如果不给 join()传入任何参数，或者传入 undefined，则仍然使用逗号作为分隔符。

> **注意** 如果数组中某一项是 null 或 undefined，则在 join()、toLocaleString()、toString()和 valueOf()返回的结果中会以空字符串表示。

6.2.8 栈方法

ECMAScript 给数组提供几个方法，让它看起来像是另外一种数据结构。数组对象可以像栈一样，也就是一种限制插入和删除项的数据结构。栈是一种后进先出（LIFO，Last-In-First-Out）的结构，也就是最近添加的项先被删除。数据项的插入（称为**推入**，push）和删除（称为**弹出**，pop）只在栈的一个地方发生，即栈顶。ECMAScript 数组提供了 push()和 pop()方法，以实现类似栈的行为。

push()方法接收任意数量的参数，并将它们添加到数组末尾，返回数组的最新长度。pop()方法则用于删除数组的最后一项，同时减少数组的 length 值，返回被删除的项。来看下面的例子：

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

count = colors.push("black");  // 再推入一项
alert(count);                  // 3

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

这里创建了一个当作栈来使用的数组（注意不需要任何额外的代码，push()和 pop()都是数组的默认方法）。首先，使用 push()方法把两个字符串推入数组末尾，将结果保存在变量 count 中（结果为 2）。

然后，再推入另一个值，再把结果保存在 count 中。因为现在数组中有 3 个元素，所以 push()返回 3。在调用 pop()时，会返回数组的最后一项，即字符串"black"。此时数组还有两个元素。

栈方法可以与数组的其他任何方法一起使用，如下例所示：

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

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

这里先初始化了包含两个字符串的数组，然后通过 push()添加了第三个值，第四个值是通过直接在位置 3 上赋值添加的。调用 pop()时，返回了字符串"black"，也就是最后添加到数组的字符串。

6.2.9 队列方法

就像栈是以 LIFO 形式限制访问的数据结构一样，队列以先进先出（FIFO，First-In-First-Out）形式限制访问。队列在列表末尾添加数据，但从列表开头获取数据。因为有了在数据末尾添加数据的 push()方法，所以要模拟队列就差一个从数组开头取得数据的方法了。这个数组方法叫 shift()，它会删除数组的第一项并返回它，然后数组长度减 1。使用 shift()和 push()，可以把数组当成队列来使用：

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

count = colors.push("black"); // 再推入一项
alert(count);                 // 3

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

这个例子创建了一个数组并用 push()方法推入三个值。加粗的那行代码使用 shift()方法取得了数组的第一项，即"red"。删除这一项之后，"green"成为第一个元素，"black"成为第二个元素，数组此时就包含两项。

ECMAScript 也为数组提供了 unshift()方法。顾名思义，unshift()就是执行跟 shift()相反的操作：在数组开头添加任意多个值，然后返回新的数组长度。通过使用 unshift()和 pop()，可以在相反方向上模拟队列，即在数组开头添加新数据，在数组末尾取得数据，如下例所示：

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

count = colors.unshift("black");  // 再推入一项
alert(count);                     // 3

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

这里，先创建一个数组，再通过 unshift()填充数组。首先，给数组添加"red"和"green"，再添加"black"，得到["black","red","green"]。调用 pop()时，删除最后一项"green"并返回它。

6.2.10　排序方法

数组有两个方法可以用来对元素重新排序：reverse()和 sort()。顾名思义，reverse()方法就是将数组元素反向排列。比如：

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

这里，数组 values 的初始状态为[1,2,3,4,5]。通过调用 reverse()反向排序，得到了[5,4,3,2,1]。这个方法很直观，但不够灵活，所以才有了 sort()方法。

默认情况下，sort()会按照升序重新排列数组元素，即最小的值在前面，最大的值在后面。为此，sort()会在每一项上调用 String()转型函数，然后比较字符串来决定顺序。即使数组的元素都是数值，也会先把数组转换为字符串再比较、排序。比如：

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

一开始数组中数值的顺序是正确的，但调用 sort()会按照这些数值的字符串形式重新排序。因此，即使 5 小于 10，但字符串"10"在字符串"5"的前头，所以 10 还是会排到 5 前面。很明显，这在多数情况下都不是最合适的。为此，sort()方法可以接收一个**比较函数**，用于判断哪个值应该排在前面。

比较函数接收两个参数，如果第一个参数应该排在第二个参数前面，就返回负值；如果两个参数相等，就返回 0；如果第一个参数应该排在第二个参数后面，就返回正值。下面是使用简单比较函数的一个例子：

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

这个比较函数可以适用于大多数数据类型，可以把它当作参数传给 sort()方法，如下所示：

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

在给 sort()方法传入比较函数后，数组中的数值在排序后保持了正确的顺序。当然，比较函数也可以产生降序效果，只要把返回值交换一下即可：

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

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

此外，这个比较函数还可简写为一个箭头函数：

```
let values = [0, 1, 5, 10, 15];
values.sort((a, b) => a < b ? 1 : a > b ? -1 : 0);
alert(values); // 15,10,5,1,0
```

在这个修改版函数中，如果第一个值应该排在第二个值后面则返回 1，如果第一个值应该排在第二个值前面则返回–1。交换这两个返回值之后，较大的值就会排在前头，数组就会按照降序排序。当然，如果只是想反转数组的顺序，reverse()更简单也更快。

> **注意** reverse()和 sort()都返回调用它们的数组的引用。

如果数组的元素是数值，或者是其 valueOf()方法返回数值的对象（如 Date 对象），这个比较函数还可以写得更简单，因为这时可以直接用第二个值减去第一个值：

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

比较函数就是要返回小于 0、0 和大于 0 的数值，因此减法操作完全可以满足要求。

6.2.11　操作方法

对于数组中的元素，我们有很多操作方法。比如，concat()方法可以在现有数组全部元素基础上创建一个新数组。它首先会创建一个当前数组的副本，然后再把它的参数添加到副本末尾，最后返回这个新构建的数组。如果传入一个或多个数组，则 concat()会把这些数组的每一项都添加到结果数组。如果参数不是数组，则直接把它们添加到结果数组末尾。来看下面的例子：

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

console.log(colors);   // ["red", "green","blue"]
console.log(colors2);  // ["red", "green", "blue", "yellow", "black", "brown"]
```

这里先创建一个包含 3 个值的数组 colors。然后 colors 调用 concat()方法，传入字符串"yellow"和一个包含"black"和"brown"的数组。保存在 colors2 中的结果就是["red", "green", "blue", "yellow", "black", "brown"]。原始数组 colors 保持不变。

打平数组参数的行为可以重写，方法是在参数数组上指定一个特殊的符号：Symbol.isConcatSpreadable。这个符号能够阻止 concat()打平参数数组。相反，把这个值设置为 true 可以强制打平类数组对象：

```
let colors = ["red", "green", "blue"];
let newColors = ["black", "brown"];
let moreNewColors = {
  [Symbol.isConcatSpreadable]: true,
  length: 2,
  0: "pink",
  1: "cyan"
};

newColors[Symbol.isConcatSpreadable] = false;

// 强制不打平数组
let colors2 = colors.concat("yellow", newColors);

// 强制打平类数组对象
let colors3 = colors.concat(moreNewColors);

console.log(colors);   // ["red", "green", "blue"]
console.log(colors2);  // ["red", "green", "blue", "yellow", ["black", "brown"]]
console.log(colors3);  // ["red", "green", "blue", "pink", "cyan"]
```

接下来，方法 slice()用于创建一个包含原有数组中一个或多个元素的新数组。slice()方法可以接收一个或两个参数：返回元素的开始索引和结束索引。如果只有一个参数，则 slice()会返回该索引到数组末尾的所有元素。如果有两个参数，则 slice()返回从开始索引到结束索引对应的所有元素，其中不包含结束索引对应的元素。记住，这个操作不影响原始数组。来看下面的例子：

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

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

这里，colors 数组一开始有 5 个元素。调用 slice()传入 1 会得到包含 4 个元素的新数组。其中

不包括"red"，这是因为拆分操作要从位置 1 开始，即从"green"开始。得到的 colors2 数组包含"green"、"blue"、"yellow"和"purple"。colors3 数组是通过调用 slice()并传入 1 和 4 得到的，即从位置 1 开始复制到位置 3。因此 colors3 包含"green"、"blue"和"yellow"。

> **注意** 如果 slice()的参数有负值，那么就以数值长度加上这个负值的结果确定位置。比如，在包含 5 个元素的数组上调用 slice(-2,-1)，就相当于调用 slice(3,4)。如果结束位置小于开始位置，则返回空数组。

或许最强大的数组方法就属 splice()了，使用它的方式可以有很多种。splice()的主要目的是在数组中间插入元素，但有 3 种不同的方式使用这个方法。

- **删除**。需要给 splice()传 2 个参数：要删除的第一个元素的位置和要删除的元素数量。可以从数组中删除任意多个元素，比如 splice(0, 2)会删除前两个元素。
- **插入**。需要给 splice()传 3 个参数：开始位置、0（要删除的元素数量）和要插入的元素，可以在数组中指定的位置插入元素。第三个参数之后还可以传第四个、第五个参数，乃至任意多个要插入的元素。比如，splice(2, 0, "red", "green")会从数组位置 2 开始插入字符串"red"和"green"。
- **替换**。splice()在删除元素的同时可以在指定位置插入新元素，同样要传入 3 个参数：开始位置、要删除元素的数量和要插入的任意多个元素。要插入的元素数量不一定跟删除的元素数量一致。比如，splice(2, 1, "red", "green")会在位置 2 删除一个元素，然后从该位置开始向数组中插入"red"和"green"。

splice()方法始终返回这样一个数组，它包含从数组中被删除的元素（如果没有删除元素，则返回空数组）。以下示例展示了上述 3 种使用方式。

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(0,1);  // 删除第一项
alert(colors);                     // green,blue
alert(removed);                    // red，只有一个元素的数组

removed = colors.splice(1, 0, "yellow", "orange");    // 在位置 1 插入两个元素
alert(colors);                                        // green,yellow,orange,blue
alert(removed);                                       // 空数组

removed = colors.splice(1, 1, "red", "purple");  // 插入两个值，删除一个元素
alert(colors);                                   // green,red,purple,orange,blue
alert(removed);                                  // yellow，只有一个元素的数组
```

这个例子中，colors 数组一开始包含 3 个元素。第一次调用 splice()时，只删除了第一项，colors 中还有"green"和"blue"。第二次调用 slice()时，在位置 1 插入两项，然后 colors 包含"green"、"yellow"、"orange"和"blue"。这次没删除任何项，因此返回空数组。最后一次调用 splice()时删除了位置 1 上的一项，同时又插入了"red"和"purple"。最后，colors 数组包含"green"、"red"、"purple"、"orange"和"blue"。

6.2.12 搜索和位置方法

ECMAScript 提供两类搜索数组的方法：按严格相等搜索和按断言函数搜索。

1. 严格相等

ECMAScript 提供了 3 个严格相等的搜索方法：indexOf()、lastIndexOf()和 includes()。其中，前两个方法在所有版本中都可用，而第三个方法是 ECMAScript 7 新增的。这些方法都接收两个参数：要查找的元素和一个可选的起始搜索位置。indexOf()和 includes()方法从数组前头（第一项）开始向后搜索，而 lastIndexOf()从数组末尾（最后一项）开始向前搜索。

indexOf()和 lastIndexOf()都返回要查找的元素在数组中的位置，如果没找到则返回–1。includes()返回布尔值，表示是否至少找到一个与指定元素匹配的项。在比较第一个参数跟数组每一项时，会使用全等（===）比较，也就是说两项必须严格相等。下面来看一些例子：

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

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

alert(numbers.indexOf(4, 4));       // 5
alert(numbers.lastIndexOf(4, 4));   // 3
alert(numbers.includes(4, 7));      // false

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

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

2. 断言函数

ECMAScript 也允许按照定义的断言函数搜索数组，每个索引都会调用这个函数。断言函数的返回值决定了相应索引的元素是否被认为匹配。

断言函数接收 3 个参数：元素、索引和数组本身。其中元素是数组中当前搜索的元素，索引是当前元素的索引，而数组就是正在搜索的数组。断言函数返回真值，表示是否匹配。

find()和 findIndex()方法使用了断言函数。这两个方法都从数组的最小索引开始。find()返回第一个匹配的元素，findIndex()返回第一个匹配元素的索引。这两个方法也都接收第二个可选的参数，用于指定断言函数内部 this 的值。

```
const people = [
  {
    name: "Matt",
    age: 27
  },
  {
    name: "Nicholas",
    age: 29
  }
];

alert(people.find((element, index, array) => element.age < 28));
// {name: "Matt", age: 27}

alert(people.findIndex((element, index, array) => element.age < 28));
// 0
```

找到匹配项后，这两个方法都不再继续搜索。

```
const evens = [2, 4, 6];

// 找到匹配后，永远不会检查数组的最后一个元素
evens.find((element, index, array) => {
  console.log(element);
  console.log(index);
  console.log(array);
  return element === 4;
});
// 2
// 0
// [2, 4, 6]
// 4
// 1
// [2, 4, 6]
```

6.2.13 迭代方法

ECMAScript 为数组定义了 5 个迭代方法。每个方法接收两个参数：以每一项为参数运行的函数，以及可选的作为函数运行上下文的作用域对象（影响函数中 this 的值）。传给每个方法的函数接收 3 个参数：数组元素、元素索引和数组本身。因具体方法而异，这个函数的执行结果可能会也可能不会影响方法的返回值。数组的 5 个迭代方法如下。

- every()：对数组每一项都运行传入的函数，如果对每一项函数都返回 true，则这个方法返回 true。
- filter()：对数组每一项都运行传入的函数，函数返回 true 的项会组成数组之后返回。
- forEach()：对数组每一项都运行传入的函数，没有返回值。
- map()：对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组。
- some()：对数组每一项都运行传入的函数，如果有一项函数返回 true，则这个方法返回 true。

这些方法都不改变调用它们的数组。

在这些方法中，every()和 some()是最相似的，都是从数组中搜索符合某个条件的元素。对 every()来说，传入的函数必须对每一项都返回 true，它才会返回 true；否则，它就返回 false。而对 some()来说，只要有一项让传入的函数返回 true，它就会返回 true。下面是一个例子：

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

let everyResult = numbers.every((item, index, array) => item > 2);
alert(everyResult);  // false

let someResult = numbers.some((item, index, array) => item > 2);
alert(someResult);   // true
```

以上代码调用了 every()和 some()，传入的函数都是在给定项大于 2 时返回 true。every()返回 false 是因为并不是每一项都能达到要求。而 some()返回 true 是因为至少有一项满足条件。

下面再看一看 filter()方法。这个方法基于给定的函数来决定某一项是否应该包含在它返回的数组中。比如，要返回一个所有数值都大于 2 的数组，可以使用如下代码：

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

let filterResult = numbers.filter((item, index, array) => item > 2);
alert(filterResult);  // 3,4,5,4,3
```

这里，调用 filter()返回的数组包含 3、4、5、4、3，因为只有对这些项传入的函数才返回 true。这个方法非常适合从数组中筛选满足给定条件的元素。

接下来 map()方法也会返回一个数组。这个数组的每一项都是对原始数组中同样位置的元素运行传入函数而返回的结果。例如，可以将一个数组中的每一项都乘以 2，并返回包含所有结果的数组，如下所示：

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

let mapResult = numbers.map((item, index, array) => item * 2);

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

以上代码返回了一个数组，包含原始数组中每个值乘以 2 的结果。这个方法非常适合创建一个与原始数组元素一一对应的新数组。

最后，再来看一看 forEach()方法。这个方法只会对每一项运行传入的函数，没有返回值。本质上，forEach()方法相当于使用 for 循环遍历数组。比如：

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

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

数组的这些迭代方法通过执行不同操作方便了对数组的处理。

6.2.14　归并方法

ECMAScript 为数组提供了两个归并方法：reduce()和 reduceRight()。这两个方法都会迭代数组的所有项，并在此基础上构建一个最终返回值。reduce()方法从数组第一项开始遍历到最后一项。而 reduceRight()从最后一项开始遍历至第一项。

这两个方法都接收两个参数：对每一项都会运行的归并函数，以及可选的以之为归并起点的初始值。传给 reduce()和 reduceRight()的函数接收 4 个参数：上一个归并值、当前项、当前项的索引和数组本身。这个函数返回的任何值都会作为下一次调用同一个函数的第一个参数。如果没有给这两个方法传入可选的第二个参数（作为归并起点值），则第一次迭代将从数组的第二项开始，因此传给归并函数的第一个参数是数组的第一项，第二个参数是数组的第二项。

可以使用 reduce()函数执行累加数组中所有数值的操作，比如：

```
let values = [1, 2, 3, 4, 5];
let sum = values.reduce((prev, cur, index, array) => prev + cur);

alert(sum);  // 15
```

第一次执行归并函数时，prev 是 1，cur 是 2。第二次执行时，prev 是 3（1 + 2），cur 是 3（数组第三项）。如此递进，直到把所有项都遍历一次，最后返回归并结果。

reduceRight()方法与之类似，只是方向相反。来看下面的例子：

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

在这里，第一次调用归并函数时 prev 是 5，而 cur 是 4。当然，最终结果相同，因为归并操作都是简单的加法。

究竟是使用 reduce()还是 reduceRight()，只取决于遍历数组元素的方向。除此之外，这两个方法没什么区别。

6.3 定型数组

定型数组（typed array）是 ECMAScript 新增的结构，目的是提升向原生库传输数据的效率。实际上，JavaScript 并没有“TypedArray”类型，它所指的其实是一种特殊的包含数值类型的数组。为理解如何使用定型数组，有必要先了解一下它的用途。

6.3.1 历史

随着浏览器的流行，不难想象人们会满怀期待地通过它来运行复杂的 3D 应用程序。早在 2006 年，Mozilla、Opera 等浏览器提供商就实验性地在浏览器中增加了用于渲染复杂图形应用程序的编程平台，无须安装任何插件。其目标是开发一套 JavaScript API，从而充分利用 3D 图形 API 和 GPU 加速，以便在<canvas>元素上渲染复杂的图形。

1. WebGL

最后的 JavaScript API 是基于 OpenGL ES（OpenGL for Embedded Systems）2.0 规范的。OpenGL ES 是 OpenGL 专注于 2D 和 3D 计算机图形的子集。这个新 API 被命名为 WebGL（Web Graphics Library），于 2011 年发布 1.0 版。有了它，开发者就能够编写涉及复杂图形的应用程序，它会被兼容 WebGL 的浏览器原生解释执行。

在 WebGL 的早期版本中，因为 JavaScript 数组与原生数组之间不匹配，所以出现了性能问题。图形驱动程序 API 通常不需要以 JavaScript 默认双精度浮点格式传递给它们的数值，而这恰恰是 JavaScript 数组在内存中的格式。因此，每次 WebGL 与 JavaScript 运行时之间传递数组时，WebGL 绑定都需要在目标环境分配新数组，以其当前格式迭代数组，然后将数值转型为新数组中的适当格式，而这些要花费很多时间。

2. 定型数组

这当然是难以接受的，Mozilla 为解决这个问题而实现了 CanvasFloatArray。这是一个提供 JavaScript 接口的、C 语言风格的浮点值数组。JavaScript 运行时使用这个类型可以分配、读取和写入数组。这个数组可以直接传给底层图形驱动程序 API，也可以直接从底层获取到。最终，CanvasFloatArray 变成了 Float32Array，也就是今天定型数组中可用的第一个“类型”。

6.3.2 ArrayBuffer

Float32Array 实际上是一种“视图”，可以允许 JavaScript 运行时访问一块名为 ArrayBuffer 的预分配内存。ArrayBuffer 是所有定型数组及视图引用的基本单位。

> **注意** SharedArrayBuffer 是 ArrayBuffer 的一个变体，可以无须复制就在执行上下文间传递它。关于这种类型，请参考第 27 章。