附注：重点看5.5数组方法　大致看5.6 Iterable object（可迭代对象）

简介

JavaScript 最初的目的是为了“**赋予网页生命**”。

这种编程语言我们称之为 **脚本**。它们可以写在 HTML 中，在页面加载的时候会自动执行。

脚本作为纯文本存在和执行。它们不需要特殊的准备或编译即可运行。

- 和 HTML/CSS 完全的集成。
- 使用简单的工具完成简单的任务。
- 被所有的主流浏览器支持，并且默认开启。

满足这三条的浏览器技术也只有 JavaScript 了。

这就是为什么 JavaScript 与众不同！这也是为什么它是创建浏览器界面的最普遍的工具。

此外，JavaScript 还支持创建服务器，移动端应用程序等。

总结

- JavaScript 最开始是为浏览器设计的一门语言，但是现在也被用于很多其他的环境。
- 现在，JavaScript 是一门在浏览器中使用最广、并且能够很好集成 HTML/CSS 的语言。
- 有很多其他的语言可以被编译成 JavaScript，这些语言还提供了更多的功能。最好还是了解一下这些语言，至少在掌握了 JavaScript 之后简单地看一下。

手册规范

MDN　各大小总结网站案例

使用

一般来说，只有最简单的脚本才嵌入到 HTML 中。更复杂的脚本存放在单独的文件中。

使用独立文件的好处是浏览器会下载它，然后将它保存到浏览器的 缓存 中。

之后，其他页面想要相同的脚本就会从缓存中获取，而不是下载它。所以文件实际上只会下载一次。

这可以节省流量，并使得页面（加载）更快。

- 我们可以使用一个 `<script>` 标签将 JavaScript 代码添加到页面中。
- `type` 和 `language` 特性（attribute）不是必需的。
- 外部的脚本可以通过 `<script src="path/to/script.js"></script>` 的方式插入。

基础运算符，数学

let a = 1;

let b = 2;

let c = 3 - (a = b + 1);

alert(a); // 3

alert(c); // 0

另一个有趣的特性是链式赋值：

let a, b, c;

a = b = c = 2 + 2;

alert(a); // 4

alert(b); // 4

alert(c); // 4

链式赋值从右到左进行计算。首先，对最右边的表达式 `2 + 2` 求值，然后将其赋给左边的变量：`c`、`b` 和 `a`。最后，所有的变量共享一个值。

同样，出于可读性，最好将这种代码分成几行：

```
c = 2 + 2;
b = c;
a = c;
```

这样可读性更强，尤其是在快速浏览代码的时候。

- 当运算符置于变量后，被称为“后置形式”：`counter++`。
- 当运算符置于变量前，被称为“前置形式”：`++counter`。

let counter = 1;

let a = ++counter; // (*)

alert(a); // 2

`(*)` 所在的行是前置形式 `++counter`，对 `counter` 做自增运算，返回的是新的值 `2`。因此 `alert` 显示的是 `2`。

let counter = 1;

let a = counter++; // (*) 将 ++counter 改为 counter++

alert(a); // 1

`(*)` 所在的行是后置形式 `counter++`，它同样对 `counter` 做加法，但是返回的是 **旧值**（做加法之前的值）。因此 `alert` 显示的是 `1`。

总结：

- 如果自增/自减的值不会被使用，那么两者形式没有区别
- 如果我们想要对变量进行自增操作，**并且** 需要立刻使用自增后的值，那么我们需要使用前置形式
- 如果我们想要将一个数加一，但是我们想使用其自增之前的值，那么我们需要使用后置形式

let a = 1, b = 1;

alert(++a); // 2，前置运算符返回最新值

alert(b++); // 1，后置运算符返回旧值

alert(a); // 2，自增一次

alert(b); // 2，自增一次

```
"" + 1 + 0 = "10" // (1)
"" - 1 + 0 = -1 // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6
4 + 5 + "px" = "9px"
"$" + 4 + 5 = "$45"
"4" - 2 = 2
"4px" - 2 = NaN
7 / 0 = Infinity
"  -9  " + 5 = "  -9  5" // (3)
"  -9  " - 5 = -14 // (4)
null + 1 = 1 // (5)
undefined + 1 = NaN // (6)
" \t \n" - 2 = -2 // (7)
```

1. 有字符串的加法 `"" + 1`，首先会将数字 `1` 转换为一个字符串：`"" + 1 = "1"`，然后我们得到 `"1" + 0`，再次应用同样的规则得到最终的结果。
2. 减法 `-`（像大多数数学运算一样）只能用于数字，它会使空字符串 `""` 转换为 `0`。
3. 带字符串的加法会将数字 `5` 加到字符串之后。
4. 减法始终将字符串转换为数字，因此它会使 `" -9 "` 转换为数字 `-9`（忽略了字符串首尾的空格）。
5. `null` 经过数字转换之后会变为 `0`。
6. `undefined` 经过数字转换之后会变为 `NaN`。
7. 字符串转换为数字时，会忽略字符串的首尾处的空格字符。在这里，整个字符串由空格字符组成，包括 `\t`、`\n` 以及它们之间的“常规”空格。因此，类似于空字符串，所以会变为 `0`。

值的比较

```
5 > 4 → true
"apple" > "pineapple" → false
"2" > "12" → true
undefined == null → true
undefined === null → false
null == "\n0\n" → false
null === +"\n0\n" → false
```

结果的原因：

1. 数字间比较大小，显然得 true。
2. 按词典顺序比较，得 false。`"a"` 比 `"p"` 小。
3. 与第 2 题同理，首位字符 `"2"` 大于 `"1"`。
4. `null` 只与 `undefined` 互等。
5. 严格相等模式下，类型不同得 false。
6. 与第 4 题同理，`null` 只与 `undefined` 相等。
7. 不同类型严格不相等。

条件分支：if 和 '?'

使用一系列问号 `?` 运算符可以返回一个取决于多个条件的值。

例如：

```
let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Greetings!' :
  'What an unusual age!';

alert( message );
```

可能很难一下子看出发生了什么。但经过仔细观察，我们可以看到它只是一个普通的检查序列。

1. 第一个问号检查 `age < 3`。
2. 如果为真 — 返回 `'Hi, baby!'`。否则，会继续执行冒号 `":"` 后的表达式，检查 `age < 18`。
3. 如果为真 — 返回 `'Hello!'`。否则，会继续执行下一个冒号 `":"` 后的表达式，检查 `age < 100`。
4. 如果为真 — 返回 `'Greetings!'`。否则，会继续执行最后一个冒号 `":"` 后面的表达式，返回 `'What an unusual age!'`。

使用 `if..else` 语句，编写代码实现通过 `prompt` 获取一个数字并用 `alert` 显示结果：

- 如果这个数字大于 0，就显示 `1`，
- 如果这个数字小于 0，就显示 `-1`，
- 如果这个数字等于 0，就显示 `0`。

在这个任务中，我们假设输入永远是一个数字。

```
let value = prompt('Type a number', 0);

if (value > 0) {
 alert( 1 );
} else if (value < 0) {
 alert( -1 );
} else {
 alert( 0 );
}
```

箭头函数，基础知识

创建函数还有另外一种非常简单的语法，并且这种方法通常比函数表达式更好。

它被称为“箭头函数”，因为它看起来像这样：

```
let func = (arg1, arg2, ...argN) => expression
```

让我们来看一个具体的例子：

```
let sum = (a, b) => a + b;
/* 这个箭头函数是下面这个函数的更短的版本：
let sum = function(a, b) {
  return a + b;
};
*/
alert( sum(1, 2) ); // 3
```

如果没有参数，括号将是空的（但括号应该保留）：

```
let sayHi = () => alert("Hello!");
sayHi();
```

总结

对于一行代码的函数来说，箭头函数是相当方便的。它具体有两种：

1. 不带花括号：`(...args) => expression` — 右侧是一个表达式：函数计算表达式并返回其结果。
2. 带花括号：`(...args) => { body }` — 花括号允许我们在函数中编写多个语句，但是我们需要显式地 `return` 来返回一些内容

用箭头函数重写

用箭头函数重写下面的函数表达式：

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

解决方案

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  () => alert("You agreed."),
  () => alert("You canceled the execution.")
);
```

JavaScript 特性

变量

可以使用以下方式声明变量：

- `let`
- `const`（不变的，不能被改变）
- `var`（旧式的，稍后会看到）

一个变量名可以由以下组成：

- 字母和数字，但是第一个字符不能是数字。
- 字符 `$` 和 `_` 是允许的，用法同字母。
- 非拉丁字母和象形文字也是允许的，但通常不会使用。

有 8 种数据类型：

- `number` — 可以是浮点数，也可以是整数，
- `bigint` — 用于任意长度的整数，
- `string` — 字符串类型，
- `boolean` — 逻辑值：`true/false`，
- `null` — 具有单个值 `null` 的类型，表示“空”或“不存在”，
- `undefined` — 具有单个值 `undefined` 的类型，表示“未分配（未定义）”，
- `object` 和 `symbol` — 对于复杂的数据结构和唯一标识符.

函数

我们介绍了三种在 JavaScript 中创建函数的方式：

1. 函数声明：主代码流中的函数

```
function sum(a, b) {
  let result = a + b;

  return result;
}
```

2. 函数表达式：表达式上下文中的函数

```
let sum = function(a, b) {
  let result = a + b;
  return result;
}
```

3. 箭头函数：

```
// 表达式在右侧
let sum = (a, b) => a + b;
// 或带 {...} 的多行语法，此处需要 return：
let sum = (a, b) => {
  // ...
  return a + b;
}

// 没有参数
let sayHi = () => alert("Hello");

// 有一个参数
let double = n => n * 2;
```

- 函数可能具有局部变量：在函数内部声明的变量。这类变量只在函数内部可见。
- 参数可以有默认值：`function sum(a = 1, b = 2) {...}`。
- 函数总是返回一些东西。如果没有 `return` 语句，那么返回的结果是 `undefined`。

代码运行和调试重点看：代码质量

注意到以下几点：

```
function pow(x,n)  // <- 参数之间没有空格
{  // <- 花括号独占了一行
  let result=1;   // <- = 号两边没有空格
  for(let i=0;i<n;i++) {result*=x;}   // <- 没有空格
  // { ... } 里面的内容应该在新的一行上
  return result;
}

let x=prompt("x?",''), n=prompt("n?",'') // <-- 从技术的角度来看是可以的，
// 但是拆分成 2 行会更好，并且这里也缺了空格和分号 ;
```

```
if (n<=0)  // <- (n <= 0) 里面没有空格，并且应该在本行上面加一个空行
{   // <- 花括号独占了一行
  // 下面的一行代码太长了，可以将其拆分成 2 行以提高可读性
  alert(`Power ${n} is not supported, please enter an integer number greater
than zero`);
}
else // <- 可以像 "} else {" 这样写在一行上
{
  alert(pow(x,n))  // 缺失了空格和分号 ;
}
```

修改后的版本：

```
function pow(x, n) {
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
let x = prompt("x?", "");
let n = prompt("n?", "");
if (n <= 0) {
  alert(`Power ${n} is not supported,
    please enter an integer number greater than zero`);
} else {
  alert( pow(x, n) );
}
```

Object(对象)基础知识

对象

我们可以用下面两种语法中的任一种来创建一个空的对象（“空柜子”）：

```
let user = new Object(); // “构造函数” 的语法
let user = {};  // “字面量” 的语法
```

文本和属性

我们可以在创建对象的时候，立即将一些属性以键值对的形式放到 `{...}` 中。

```
let user = {     // 一个对象
  name: "John",  // 键 "name"，值 "John"
  age: 30        // 键 "age"，值 30
};
```

检查属性是否存在的操作符 `"in"`。

语法是：

```
"key" in object
```

例如：

```
let user = { name: "John", age: 30 };

alert( "age" in user ); // true，user.age 存在
alert( "blabla" in user ); // false，user.blabla 不存在。
```

请注意，`in` 的左边必须是 **属性名**。通常是一个带引号的字符串。

如果我们省略引号，就意味着左边是一个变量，它应该包含要判断的实际属性名。例如：

```
let user = { age: 30 };

let key = "age";
alert( key in user ); // true，属性 "age" 存在
```

让我们列出 `user` 所有的属性：

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // keys
  alert( key );  // name, age, isAdmin
  // 属性键的值
  alert( user[key] ); // John, 30, true
}
```

总结

对象是具有一些特殊特性的关联数组。

它们存储属性（键值对），其中：

- 属性的键必须是字符串或者 symbol（通常是字符串）。
- 值可以是任何类型。

我们可以用下面的方法访问属性：

- 点符号: `obj.property`。
- 方括号 `obj["property"]`，方括号允许从变量中获取键，例如 `obj[varWithKey]`。

其他操作：

- 删除属性：`delete obj.prop`。
- 检查是否存在给定键的属性：`"key" in obj`。
- 遍历对象：`for(let key in obj)` 循环。

JavaScript 中还有很多其他类型的对象：

- `Array` 用于存储有序数据集合，
- `Date` 用于存储时间日期，
- `Error` 用于存储错误信息。
- ……等等。

按下面的要求写代码，一条对应一行代码：

1. 创建一个空的对象 `user`。
2. 为这个对象增加一个属性，键是 `name`，值是 `John`。
3. 再增加一个属性，键是 `surname`，值是 `Smith`。
4. 把键为 `name` 的属性的值改成 `Pete`。
5. 删除这个对象中键为 `name` 的属性。

解决方案

```
let user = {};
user.name = "John";
user.surname = "Smith";
user.name = "Pete";
delete user.name;
```

写一个 `isEmpty(obj)` 函数，当对象没有属性的时候返回 `true`，否则返回 `false`。

应该像这样：

```
let schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "get up";

alert( isEmpty(schedule) ); // false
```

我们有一个保存着团队成员工资的对象：

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
}
```

写一段代码求出我们的工资总和，将计算结果保存到变量 `sum`。从所给的信息来看，结果应该是 `390`。

如果 `salaries` 是一个空对象，那结果就为 `0`。

解决方案

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
};

let sum = 0;
for (let key in salaries) {
  sum += salaries[key];
}

alert(sum); // 390
```

创建一个 `multiplyNumeric(obj)` 函数，把 `obj` 所有的数值属性都乘以 `2`。

例如：

```
// 在调用之前
let menu = {
  width: 200,
  height: 300,
  title: "My menu"
};

multiplyNumeric(menu);

// 调用函数之后
menu = {
  width: 400,
  height: 600,
  title: "My menu"
};
```

注意 `multiplyNumeric` 函数不需要返回任何值，它应该就地修改对象。

P.S. 用 `typeof` 检查值类型。

解决方案

```
function multiplyNumeric(obj) {
  for (let key in obj) {
    if (typeof obj[key] == 'number') {
      obj[key] *= 2;
    }
  }
}
```

对象拷贝，引用

变量存储的不是对象自身，而是该对象的“内存地址”，换句话说就是一个对该对象的“引用”。

当一个对象变量被拷贝 —— 引用则被拷贝，而该对象并没有被复制。

例如：

```
let user = { name: "John" };

let admin = user; // 拷贝引用
```

现在我们有了两个变量，它们保存的都是对同一个对象的引用。

我们可以用任何变量来访问该对象并修改它的内容：

```
let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // 通过 "admin" 引用来修改

alert(user.name); // 'Pete'，修改能通过 "user" 引用看到
```

上面的例子说明这里只有一个对象。就像我们有个带两把钥匙的锁柜，并使用其中一把钥匙（`admin`）来打开它。那么，我们如果之后用另外一把钥匙（`user`），就也能看到所作的改变。

总结

对象通过引用被赋值和拷贝。换句话说，一个变量存储的不是“对象的值”，而是一个对值的“引用”（内存地址）。因此，拷贝此类变量或将其作为函数参数传递时，所拷贝的是引用，而不是对象本身。

所有通过被拷贝的引用的操作（如添加、删除属性）都作用在同一个对象上。

垃圾回收

JavaScript 中主要的内存管理概念是 **可达性**。

简而言之，“可达”值是那些以某种方式可访问或可用的值。它们一定是存储在内存中的。

如果一个值可以通过引用或引用链从根访问任何其他值，则认为该值是可达的。

总结

主要需要掌握的内容：

- 垃圾回收是自动完成的，我们不能强制执行或是阻止执行。
- 当对象是可达状态时，它一定是存在于内存中的。
- 被引用与可访问（从一个根）不同：一组相互连接的对象可能整体都不可达。

对象方法，"this"

方法中的 “this”

通常，对象方法需要访问对象中存储的信息才能完成其工作。

例如，`user.sayHi()` 中的代码可能需要用到 `user` 的 name 属性。

为了访问该对象，方法中可以使用 `this` 关键字。

`this` 的值就是在点之前的这个对象，即调用该方法的对象。

“this” 不受限制

在 JavaScript 中，`this` 关键字与其他大多数编程语言中的不同。JavaScript 中的 `this` 可以用于任何函数。

解除 `this` 绑定的后果

如果你经常使用其他的编程语言，那么你可能已经习惯了“绑定 `this`”的概念，即在对象中定义的方法总是有指向该对象的 `this`。

在 JavaScript 中，`this` 是“自由”的，它的值是在调用时计算出来的，它的值并不取决于方法声明的位置，而是取决于在“点符号前”的是什么对象。

在运行时对 `this` 求值的这个概念既有优点也有缺点。一方面，函数可以被重用于不同的对象。另一方面，更大的灵活性造成了更大的出错的可能。

这里我们的立场并不是要评判编程语言的这个设计是好是坏。而是要了解怎样使用它，如何趋利避害。

箭头函数没有自己的 “this”

箭头函数有些特别：它们没有自己的 `this`。如果我们在这样的函数中引用 `this`，`this` 值取决于外部“正常的”函数。

举个例子，这里的 `arrow()` 使用的 `this` 来自于外部的 `user.sayHi()` 方法：

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

user.sayHi(); // Ilya
```

这是箭头函数的一个特性，当我们并不想要一个独立的 `this`，反而想从外部上下文中获取时，它很有用。

总结

- 存储在对象属性中的函数被称为“方法”。
- 方法允许对象进行像 `object.doSomething()` 这样的“操作”。
- 方法可以将对象引用为 `this`。

`this` 的值是在程序运行时得到的。

- 一个函数在声明时，可能就使用了 `this`，但是这个 `this` 只有在函数被调用时才会有值。
- 可以在对象之间复制函数。
- 以“方法”的语法调用函数时：`object.method()`，调用过程中的 `this` 值是 `object`。

请注意箭头函数有些特别：它们没有 `this`。在箭头函数内部访问到的 `this` 都是从外部获取的。

这里 `makeUser` 函数返回了一个对象。

访问 `ref` 的结果是什么？为什么？

```
function makeUser() {
  return {
    name: "John",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // 结果是什么？
```

解决方案

答案：一个错误。

试一下：

```
function makeUser() {
  return {
    name: "John",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // Error: Cannot read property 'name' of undefined
```

这是因为设置 `this` 的规则不考虑对象定义。只有调用那一刻才重要。

这里 `makeUser()` 中的 `this` 的值是 `undefined`，因为它是被作为函数调用的，而不是通过点符号被作为方法调用。

`this` 的值是对于整个函数的，代码段和对象字面量对它都没有影响。

所以 `ref: this` 实际上取的是当前函数的 `this`。

我们可以重写这个函数，并返回和上面相同的值为 `undefined` 的 `this`：

```
function makeUser(){
  return this; // 这次这里没有对象字面量
}

alert( makeUser().name ); // Error: Cannot read property 'name' of undefined
```

我们可以看到 `alert( makeUser().name )` 的结果和前面那个例子中 `alert( user.ref.name )` 的结果相同。

这里有个反例：

```
function makeUser() {
  return {
    name: "John",
    ref() {
      return this;
    }
  };
};

let user = makeUser();

alert( user.ref().name ); // John
```

现在正常了，因为 `user.ref()` 是一个方法。`this` 的值为点符号 `.` 前的这个对象。

构造器和操作符 "new"

构造函数

构造函数在技术上是常规函数。不过有两个约定：

1. 它们的命名以大写字母开头。
2. 它们只能由 `"new"` 操作符来执行。

是否可以创建像 `new A()==new B()` 这样的函数 `A` 和 `B`？

```
function A() { ... }
function B() { ... }

let a = new A;
let b = new B;

alert( a == b ); // true
```

如果可以，请提供一个它们的代码示例。

解决方案

是的，这是可以的。

如果一个函数返回一个对象，那么 `new` 返回那个对象而不是 `this`。

所以它们可以，例如，返回相同的外部定义的对象 `obj`：

```
let obj = {};

function A() { return obj; }
function B() { return obj; }

alert( new A() == new B() ); // true
```

总结

- 构造函数，或简称构造器，就是常规函数，但大家对于构造器有个共同的约定，就是其命名首字母要大写。
- 构造函数只能使用 `new` 来调用。这样的调用意味着在开始时创建了空的 `this`，并在最后返回填充了值的 `this`。

我们可以使用构造函数来创建多个类似的对象。

JavaScript 为许多内置的对象提供了构造函数：比如日期 `Date`、集合 `Set` 以及其他我们计划学习的内容。

创建一个构造函数 `Calculator`，它创建的对象中有三个方法：

- `read()` 使用 `prompt` 请求两个值并把它们记录在对象的属性中。
- `sum()` 返回这些属性的总和。
- `mul()` 返回这些属性的乘积。

例如：

```
let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

解决方案

```
function Calculator() {

  this.read = function() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  };

  this.sum = function() {
    return this.a + this.b;
  };

  this.mul = function() {
    return this.a * this.b;
  };
}

let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

创建一个构造函数 `Accumulator(startingValue)`。

它创建的对象应该：

- 将“当前 value”存储在属性 `value` 中。起始值被设置到构造器 `startingValue` 的参数。
- `read()` 方法应该使用 `prompt` 来读取一个新的数字，并将其添加到 `value` 中。

换句话说，`value` 属性是所有用户输入值与初始值 `startingValue` 的总和。

下面是示例代码：

```
let accumulator = new Accumulator(1); // 初始值 1

accumulator.read(); // 添加用户输入的 value
accumulator.read(); // 添加用户输入的 value

alert(accumulator.value); // 显示这些值的总和
```

解决方案

```
function Accumulator(startingValue) {
  this.value = startingValue;

  this.read = function() {
    this.value += +prompt('How much to add?', 0);
  };

}

let accumulator = new Accumulator(1);
accumulator.read();
accumulator.read();
alert(accumulator.value);
```

Symbol 类型

根据规范，对象的属性键只能是字符串类型或者 Symbol 类型。不是 Number，也不是 Boolean，只有字符串或 Symbol 这两种类型。

“Symbol” 值表示唯一的标识符。

可以使用 `Symbol()` 来创建这种类型的值：

```
// id 是 symbol 的一个实例化对象
let id = Symbol();
```

创建时，我们可以给 Symbol 一个描述（也称为 Symbol 名），这在代码调试时非常有用：

```
// id 是描述为 "id" 的 Symbol
let id = Symbol("id");
```

Symbol 保证是唯一的。即使我们创建了许多具有相同描述的 Symbol，它们的值也是不同。描述只是一个标签，不影响任何东西。

例如，这里有两个描述相同的 Symbol —— 它们不相等：

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

Symbol 不会被自动转换为字符串

JavaScript 中的大多数值都支持字符串的隐式转换。例如，我们可以 `alert` 任何值，都可以生效。Symbol 比较特殊，它不会被自动转换。

例如，这个 `alert` 将会提示出错：

```
let id = Symbol("id");
alert(id); // 类型错误：无法将 Symbol 值转换为字符串。
```

这是一种防止混乱的“语言保护”，因为字符串和 Symbol 有本质上的不同，不应该意外地将它们转换成另一个。

如果我们真的想显示一个 Symbol，我们需要在它上面调用 `.toString()`，如下所示：

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id)，现在它有效了
```

或者获取 `symbol.description` 属性，只显示描述（description）：

```
let id = Symbol("id");
alert(id.description); // id
```

数字类型

字符串

在 JavaScript 中，文本数据被以字符串形式存储，单个字符没有单独的类型。

字符串的内部格式始终是 UTF-16，它不依赖于页面编码。

引号

让我们回忆一下引号的种类。

字符串可以包含在单引号、双引号或反引号中：

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

单引号和双引号基本相同。但是，反引号允许我们通过 `${…}` 将任何表达式嵌入到字符串中：

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

字符串长度

`length` 属性表示字符串长度：

```
alert( `My\n`.length ); // 3
```

注意 `\n` 是一个单独的“特殊”字符，所以长度确实是 `3`。

`length` 是一个属性

掌握其他编程语言的人，有时会错误地调用 `str.length()` 而不是 `str.length`。这是行不通的。

请注意 `str.length` 是一个数字属性，而不是函数。后面不需要添加括号。

访问字符

要获取在 `pos` 位置的一个字符，可以使用方括号 `[pos]` 或者调用 str.charAt(pos) 方法。第一个字符从零位置开始：

```
let str = `Hello`;

// 第一个字符
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// 最后一个字符
alert( str[str.length - 1] ); // o
```

方括号是获取字符的一种现代化方法，而 `charAt` 是历史原因才存在的。

它们之间的唯一区别是，如果没有找到字符，`[]` 返回 `undefined`，而 `charAt` 返回一个空字符串：

```
let str = `Hello`;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // ''（空字符串）
```

我们也可以使用 `for..of` 遍历字符：

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o（char 变为 "H"，然后是 "e"，然后是 "l" 等）
}
```

改变大小写

toLowerCase() 和 toUpperCase() 方法可以改变大小写：

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

查找子字符串

在字符串中查找子字符串有很多种方法。

str.indexOf

第一个方法是 str.indexOf(substr, pos)。

它从给定位置 `pos` 开始，在 `str` 中查找 `substr`，如果没有找到，则返回 `-1`，否则返回匹配成功的位置。

例如：

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0，因为 'Widget' 一开始就被找到
alert( str.indexOf('widget') ); // -1，没有找到，检索是大小写敏感的

alert( str.indexOf("id") ); // 1，"id" 在位置 1 处（……idget 和 id）
```

可选的第二个参数允许我们从给定的起始位置开始检索。

例如，`"id"` 第一次出现的位置是 `1`。查询下一个存在位置时，我们从 `2` 开始检索：

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

如果我们对所有存在位置都感兴趣，可以在一个循环中使用 `indexOf`。每一次新的调用都发生在上一匹配位置之后：

```
let str = 'As sly as a fox, as strong as an ox';

let target = 'as'; // 这是我们要查找的目标

let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Found at ${foundPos}` );
  pos = foundPos + 1; // 继续从下一个位置查找
}
```

相同的算法可以简写：

```
let str = "As sly as a fox, as strong as an ox";
let target = "as";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

在 `if` 测试中 `indexOf` 有一点不方便。我们不能像这样把它放在 `if` 中：

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
    alert("We found it"); // 不工作!
}
```

上述示例中的 `alert` 不会显示，因为 `str.indexOf("Widget")` 返回 `0`（意思是它在起始位置就查找到了匹配项）。是的，但是 `if` 认为 `0` 表示 `false`。

因此我们应该检查 `-1`，像这样：

```
let str = "Widget with id";

if (str.indexOf("Widget") != -1) {
    alert("We found it"); // 现在工作了!
}
```

获取子字符串

JavaScript 中有三种获取字符串的方法：`substring`、`substr` 和 `slice`。

- `str.slice(start [, end])`

  返回字符串从 `start` 到（但不包括）`end` 的部分。

例如：

```
let str = "stringify";
alert( str.slice(0, 5) ); // 'strin'，从 0 到 5 的子字符串（不包括 5）
alert( str.slice(0, 1) ); // 's'，从 0 到 1，但不包括 1，所以只有在 0 处的字符
```

如果没有第二个参数，`slice` 会一直运行到字符串末尾：

```
let str = "stringify";
alert( str.slice(2) ); // 从第二个位置直到结束
```

`start/end` 也有可能是负值。它们的意思是起始位置从字符串结尾计算：

```
let str = "stringify";

// 从右边的第四个位置开始，在右边的第一个位置结束
alert( str.slice(-4, -1) ); // 'gif'
```

```
str.substring(start [, end])
```

返回字符串在 `start` 和 `end` **之间** 的部分。

这与 `slice` 几乎相同，但它允许 `start` 大于 `end`。

例如：

```
let str = "stringify";

// 这些对于 substring 是相同的
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// ……但对 slice 是不同的：
alert( str.slice(2, 6) ); // "ring"（一样）
alert( str.slice(6, 2) ); // ""（空字符串）
```

不支持负参数（不像 slice），它们被视为 `0`。

```
str.substr(start [, length])
```

返回字符串从 `start` 开始的给定 `length` 的部分。

与以前的方法相比，这个允许我们指定 `length` 而不是结束位置：

```
let str = "stringify";
alert( str.substr(2, 4) ); // 'ring'，从位置 2 开始，获取 4 个字符
```

第一个参数可能是负数，从结尾算起：

```
let str = "stringify";
alert( str.substr(-4, 2) ); // 'gi'，从第 4 位获取 2 个字符
```

我们回顾一下这些方法，以免混淆：

方法	选择方式……	负值参数
`slice(start, end)`	从 `start` 到 `end`（不含 `end`）	允许
`substring(start, end)`	`start` 与 `end` 之间（包括 `start`，但不包括 `end`）	负值代表 `0`
`substr(start, length)`	从 `start` 开始获取长为 `length` 的字符串	允许 `start` 为负数

使用哪一个？

它们可以完成这项工作。形式上，`substr` 有一个小缺点：它不是在 JavaScript 核心规范中描述的，而是在附录 B 中，它涵盖了主要由于历史原因而存在的仅浏览器特性。因此，非浏览器环境可能无法支持它。但实际上它在任何地方都有效。

相较于其他两个变体，`slice` 稍微灵活一些，它允许以负值作为参数并且写法更简短。因此仅仅记住这三种方法中的 `slice` 就足够了。

比较字符串

字符串按字母顺序逐字比较。

不过，也有一些奇怪的地方。

1. 小写字母总是大于大写字母：

```
alert( 'a' > 'Z' ); // true
```

2. 带变音符号的字母存在“乱序”的情况：

```
alert( 'Österreich' > 'Zealand' ); // true
```

总结

- 有 3 种类型的引号。反引号允许字符串跨越多行并可以使用 `${…}` 在字符串中嵌入表达式。
- JavaScript 中的字符串使用的是 UTF-16 编码。
- 我们可以使用像 `\n` 这样的特殊字符或通过使用 `\u...` 来操作它们的 unicode 进行字符插入。
- 获取字符时，使用 `[]`。
- 获取子字符串，使用 `slice` 或 `substring`。
- 字符串的大/小写转换，使用：`toLowerCase/toUpperCase`。
- 查找子字符串时，使用 `indexOf` 或 `includes/startsWith/endsWith` 进行简单检查。
- 根据语言比较字符串时使用 `localeCompare`，否则将按字符代码进行比较。

还有其他几种有用的字符串方法：

- `str.trim()` —— 删除字符串前后的空格 (“trims”)。
- `str.repeat(n)` —— 重复字符串 `n` 次。

写一个函数 `ucFirst(str)`，并返回首字母大写的字符串 `str`，例如：

```
ucFirst("john") == "John";
```

解决方案

我们不能“替换”第一个字符，因为在 JavaScript 中字符串是不可变的。

但是我们可以根据已有字符串创建一个首字母大写的新字符串：

```
let newStr = str[0].toUpperCase() + str.slice(1);
```

这里存在一个小问题。如果 `str` 是空的，那么 `str[0]` 就是 `undefined`，但由于 `undefined` 并没有 `toUpperCase()` 方法，因此我们会得到一个错误。

存在如下两种变体：

1. 使用 `str.charAt(0)`，因为它总是会返回一个字符串（可能为空）。
2. 为空字符添加测试。

这是第二种变体：

```
function ucFirst(str) {
  if (!str) return str;

  return str[0].toUpperCase() + str.slice(1);
}

alert( ucFirst("john") ); // John
```

写一个函数 `checkSpam(str)`，如果 `str` 包含 `viagra` 或 `XXX` 就返回 `true`，否则返回 `false`。

函数必须不区分大小写：

```
checkSpam('buy ViAgRA now') == true
checkSpam('free xxxxx') == true
checkSpam("innocent rabbit") == false
```

解决方案

为了使搜索不区分大小写，我们将字符串改为小写，然后搜索：

```
function checkSpam(str) {
  let lowerStr = str.toLowerCase();

  return lowerStr.includes('viagra') || lowerStr.includes('xxx');
}

alert( checkSpam('buy ViAgRA now') );
alert( checkSpam('free xxxxx') );
alert( checkSpam("innocent rabbit") );
```

我们有以 `"$120"` 这样的格式表示的花销。意味着：先是美元符号，然后才是数值。

创建函数 `extractCurrencyValue(str)` 从字符串中提取数值并返回。

例如：

```
alert( extractCurrencyValue('$120') === 120 ); // true
```

解决方案

```
function extractCurrencyValue(str) {
  return +str.slice(1);
}
```

数组

声明

创建一个空数组有两种语法：

```
let arr = new Array();
let arr = [];
```

绝大多数情况下使用的都是第二种语法。我们可以在方括号中添加初始元素：

```
let fruits = ["Apple", "Orange", "Plum"];
```

数组可以存储任何类型的元素。

例如:

```
// 混合值
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];

// 获取索引为 1 的对象然后显示它的 name
alert( arr[1].name ); // John

// 获取索引为 3 的函数并执行
arr[3](); // hello
```

数组就像对象一样，可以以逗号结尾：

```
let fruits = [
  "Apple",
  "Orange",
  "Plum",
];
```

因为每一行都是相似的，所以这种以“逗号结尾”的方式使得插入/移除项变得更加简单。

pop/push, shift/unshift 方法

队列（queue）是最常见的使用数组的方法之一。在计算机科学中，这表示支持两个操作的一个有序元素的集合：

- `push` 在末端添加一个元素.
- `shift` 取出队列首端的一个元素，整个队列往前移，这样原先排第二的元素现在排在了第一。

这两种操作数组都支持。

队列的应用在实践中经常会碰到。例如需要在屏幕上显示消息队列。

数组还有另一个用例，就是数据结构 栈。

它支持两种操作：

- `push` 在末端添加一个元素.
- `pop` 从末端取出一个元素.

所以新元素的添加和取出都是从“末端”开始的。

作用于数组末端的方法：

- `pop`

  取出并返回数组的最后一个元素：

  ```
  let fruits = ["Apple", "Orange", "Pear"];

  alert( fruits.pop() ); // 移除 "Pear" 然后 alert 显示出来

  alert( fruits ); // Apple, Orange
  ```

- push

在数组末端添加元素：

```
let fruits = ["Apple", "Orange"];

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

调用 `fruits.push(...)` 与 `fruits[fruits.length] = ...` 是一样的。

作用于数组首端的方法：

- `shift`

  取出数组的第一个元素并返回它：

  ```
  let fruits = ["Apple", "Orange", "Pear"];

  alert( fruits.shift() ); // 移除 Apple 然后 alert 显示出来

  alert( fruits ); // Orange, Pear
  ```

- `unshift`

  在数组的首端添加元素：

  ```
  let fruits = ["Orange", "Pear"];

  fruits.unshift('Apple');

  alert( fruits ); // Apple, Orange, Pear
  ```

`push` 和 `unshift` 方法都可以一次添加多个元素：

```
let fruits = ["Apple"];

fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");

// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

内部

数组是一种特殊的对象。使用方括号来访问属性 `arr[0]` 实际上是来自于对象的语法。它其实与 `obj[key]` 相同，其中 `arr` 是对象，而数字用作键（key）。

例如，它是通过引用来复制的：

```
let fruits = ["Banana"]

let arr = fruits; // 通过引用复制 (两个变量引用的是相同的数组)

alert( arr === fruits ); // true

arr.push("Pear"); // 通过引用修改数组

alert( fruits ); // Banana, Pear - 现在有 2 项了
```

数组误用的几种方式：

- 添加一个非数字的属性，比如 `arr.test = 5`。
- 制造空洞，比如：添加 `arr[0]`，然后添加 `arr[1000]` (它们中间什么都没有)。
- 以倒序填充数组，比如 `arr[1000]`，`arr[999]` 等等。

请将数组视为作用于 **有序数据** 的特殊结构。它们为此提供了特殊的方法。数组在 JavaScript 引擎内部是经过特殊调整的，使得更好地作用于连续的有序数据，所以请以正确的方式使用数组。如果你需要任意键值，那很有可能实际上你需要的是常规对象 `{}`。

性能

`push/pop` 方法运行的比较快，而 `shift/unshift` 比较慢。

为什么作用于数组的末端会比首端快呢？让我们看看在执行期间都发生了什么：

```
fruits.shift(); // 从首端取出一个元素
```

只获取并移除数字 `0` 对应的元素是不够的。其它元素也需要被重新编号。

`shift` 操作必须做三件事：

1. 移除索引为 `0` 的元素。
2. 把所有的元素向左移动，把索引 `1` 改成 `0`，`2` 改成 `1` 以此类推，对其重新编号。
3. 更新 `length` 属性。

数组里的元素越多，移动它们就要花越多的时间，也就意味着越多的内存操作。

`unshift` 也是一样：为了在数组的首端添加元素，我们首先需要将现有的元素向右移动，增加它们的索引值。

那 `push/pop` 是什么样的呢？它们不需要移动任何东西。如果从末端移除一个元素，`pop` 方法只需要清理索引值并缩短 `length` 就可以了。

`pop` 操作的行为：

```
fruits.pop(); // 从末端取走一个元素
```

对于数组来说还有另一种循环方式，`for..of`：

```
let fruits = ["Apple", "Orange", "Plum"];

// 遍历数组元素
for (let fruit of fruits) {
  alert( fruit );
}
```

关于 “length”

当我们修改数组的时候，`length` 属性会自动更新。准确来说，它实际上不是数组里元素的个数，而是最大的数字索引值加一。

例如，一个数组只有一个元素，但是这个元素的索引值很大，那么这个数组的 `length` 也会很大：

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

`length` 属性的另一个有意思的点是它是可写的。

如果我们手动增加它，则不会发生任何有趣的事儿。但是如果我们减少它，数组就会被截断。该过程是不可逆的，下面是例子：

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

arr.length = 2; // 截断到只剩 2 个元素
alert( arr ); // [1, 2]

arr.length = 5; // 又把 length 加回来
alert( arr[3] ); // undefined：被截断的那些数值并没有回来
```

所以，清空数组最简单的方法就是：`arr.length = 0;`。

toString

数组有自己的 `toString` 方法的实现，会返回以逗号隔开的元素列表。

例如：

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

此外，我们试试运行一下这个：

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

数组没有 `Symbol.toPrimitive`，也没有 `valueOf`，它们只能执行 `toString` 进行转换，所以这里 `[]` 就变成了一个空字符串，`[1]` 变成了 `"1"`，`[1,2]` 变成了 `"1,2"`。

当 `"+"` 运算符把一些项加到字符串后面时，加号后面的项也会被转换成字符串，所以下一步就会是这样：

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

下面的代码将会显示什么？

```
let fruits = ["Apples", "Pear", "Orange"];

// 在“副本”里 push 了一个新的值
let shoppingCart = fruits;
shoppingCart.push("Banana");

// fruits 里面是什么？
alert( fruits.length ); // ?
```

解决方案

结果是 `4`：

```
let fruits = ["Apples", "Pear", "Orange"];

let shoppingCart = fruits;

shoppingCart.push("Banana");

alert( fruits.length ); // 4
```

这是因为数组是对象。所以 `shoppingCart` 和 `fruits` 是同一数组的引用。

我们试试下面的 5 个数组操作。

1. 创建一个数组 `styles`，里面存储有 “Jazz” 和 “Blues”。
2. 将 “Rock-n-Roll” 从数组末端添加进去。
3. 用 “Classics” 替换掉数组最中间的元素。查找数组最中间的元素的代码应该适用于任何奇数长度的数组。
4. 去掉数组的第一个值并显示它。
5. 在数组前面添加 `Rap` 和 `Reggae`。

过程中的数组：

```
Jazz, Blues
Jazz, Bues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

解决方案

```
let styles = ["Jazz", "Blues"];
styles.push("Rock-n-Roll");
styles[Math.floor((styles.length - 1) / 2)] = "Classics";
alert( styles.shift() );
styles.unshift("Rap", "Reggae");
```

输入是以数字组成的数组，例如 `arr = [1, -2, 3, 4, -9, 6]`.

任务是：找出所有项的和最大的 `arr` 数组的连续子数组。

写出函数 `getMaxSubSum(arr)`，用其找出并返回最大和。

例如：

```
getMaxSubSum([-1, 2, 3, -9]) == 5（高亮项的加和）
getMaxSubSum([2, -1, 2, 3, -9]) == 6
getMaxSubSum([-1, 2, 3, -9, 11]) == 11
getMaxSubSum([-2, -1, 1, 2]) == 3
getMaxSubSum([100, -9, 2, -3, 5]) == 100
getMaxSubSum([1, 2, 3]) == 6（所有项的和）
```

如果所有项都是负数，那就一个项也不取（子数组是空的），所以返回的是 0：

```
getMaxSubSum([-1, -2, -3]) = 0
```

请尝试想出一个快速的解决方案：复杂度可以是 O(n2)，有能力达到 O(n) 则更好。

慢的解决方案

我们可以计算所有可能的子集的和。

最简单的方法就是获取每个元素然后计算从它开始所有子数组的和。

以 `[-1, 2, 3, -9, 11]` 为例：

```
// 从 -1 开始：
-1
-1 + 2
-1 + 2 + 3
-1 + 2 + 3 + (-9)
-1 + 2 + 3 + (-9) + 11

// 从 2 开始：
2
2 + 3
2 + 3 + (-9)
2 + 3 + (-9) + 11

// 从 3 开始：
3
3 + (-9)
3 + (-9) + 11

// 从 -9 开始：
-9
```

```
-9 + 11

// 从 -11 开始：
-11
```

这样写出来的代码实际上是一个嵌套循环：外部循环遍历数组所有元素，内部循环计算从当前元素开始的所有子数组各自的和。

```
function getMaxSubSum(arr) {
  let maxSum = 0; // 如果没有取到任何元素，就返回 0

  for (let i = 0; i < arr.length; i++) {
    let sumFixedStart = 0;
    for (let j = i; j < arr.length; j++) {
      sumFixedStart += arr[j];
      maxSum = Math.max(maxSum, sumFixedStart);
    }
  }

  return maxSum;
}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
```

该方案的时间复杂度是 O(n2)。也就是说，如果我们把数组大小增加 2 倍，那么算法的运行时间将会延长4倍。

对于大型数组（1000，10000 或者更多项）这种算法会导致严重的时间消耗。

快的解决方案

让我们遍历数组，将当前局部元素的和保存在变量 `s` 中。如果 `s` 在某一点变成负数了，就重新分配 `s=0`。所有 `s` 中的最大值就是答案。

如果文字描述不太好理解，就直接看下面的代码吧，真的很短：

```
function getMaxSubSum(arr) {
  let maxSum = 0;
  let partialSum = 0;

  for (let item of arr) { // arr 中的每个 item
    partialSum += item; // 将其加到 partialSum
    maxSum = Math.max(maxSum, partialSum); // 记住最大值
    if (partialSum < 0) partialSum = 0; // 如果是负数就置为 0
  }

  return maxSum;
}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
```

```
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([-1, -2, -3]) ); // 0
```

该算法只需要遍历 1 轮数组，所以时间复杂度是 O(n)。

你也可以在这获取更多该算法的细节信息：最大子数组问题。如果还是不明白，那就调试上面的例子，观察它是怎样工作的，说得再多也没有自己去调试好使。

```
function getMaxSubSum(arr) {
  let maxSum = 0;
  let partialSum = 0;

  for (let item of arr) {
    partialSum += item;
    maxSum = Math.max(maxSum, partialSum);
    if (partialSum < 0) partialSum = 0;
  }
  return maxSum;
}
```

数组方法

添加/移除数组元素

我们已经学了从数组的首端或尾端添加和删除元素的方法：

- `arr.push(...items)` —— 从尾端添加元素，
- `arr.pop()` —— 从尾端提取元素，
- `arr.shift()` —— 从首端提取元素，
- `arr.unshift(...items)` —— 从首端添加元素。

splice

如何从数组中删除元素？

数组是对象，所以我们可以尝试使用 `delete`：

```
let arr = ["I", "go", "home"];

delete arr[1]; // remove "go"

alert( arr[1] ); // undefined

// now arr = ["I",  , "home"];
alert( arr.length ); // 3
```

元素被删除了，但数组仍然有 3 个元素，我们可以看到 `arr.length == 3`。

这很正常，因为 `delete obj.key` 是通过 `key` 来移除对应的值。对于对象来说是可以的。但是对于数组来说，我们通常希望剩下的元素能够移动并占据被释放的位置。我们希望得到一个更短的数组。

所以应该使用特殊的方法。

arr.splice(str) 方法可以说是处理数组的瑞士军刀。它可以做所有事情：添加，删除和插入元素。

语法是：

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

从 `index` 开始：删除 `deleteCount` 个元素并在当前位置插入 `elem1, ..., elemN`。最后返回已删除元素的数组。

通过例子我们可以很容易地掌握这个方法。

让我们从删除开始：

```
let arr = ["I", "study", "JavaScript"];

arr.splice(1, 1); // 从索引 1 开始删除 1 个元素

alert( arr ); // ["I", "JavaScript"]
```

简单，对吧？从索引 `1` 开始删除 `1` 个元素。

在下一个例子中，我们删除了 3 个元素，并用另外两个元素替换它们：

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");

alert( arr ) // now ["Let's", "dance", "right", "now"]
```

在这里我们可以看到 `splice` 返回了已删除元素的数组：

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// 删除前两个元素
let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- 被从数组中删除了的元素
```

我们可以将 `deleteCount` 设置为 `0`，`splice` 方法就能够插入元素而不用删除任何元素：

```
let arr = ["I", "study", "JavaScript"];

// 从索引 2 开始
// 删除 0 个元素
// 然后插入 "complex" 和 "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

在这里和其他数组方法中，负向索引都是被允许的。它们从数组末尾计算位置，如下所示：

```
let arr = [1, 2, 5];

// 从索引 -1（尾端前一位）
// 删除 0 个元素，
// 然后插入 3 和 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

slice

arr.slice 方法比 `arr.splice` 简单得多。

语法是：

```
arr.slice([start], [end])
```

它会返回一个新数组，将所有从索引 `start` 到 `end`（不包括 `end`）的数组项复制到一个新的数组。`start` 和 `end` 都可以是负数，在这种情况下，从末尾计算索引。

它和字符串的 `str.slice` 方法有点像，就是把子字符串替换成子数组。

例如：

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s（复制从位置 1 到位置 3 的元素）

alert( arr.slice(-2) ); // s,t（复制从位置 -2 到尾端的元素）
```

我们也可以不带参数地调用它：`arr.slice()` 会创建一个 `arr` 的副本。其通常用于获取副本，以进行不影响原始数组的进一步转换。

concat

arr.concat 创建一个新数组，其中包含来自于其他数组和其他项的值。

语法：

```
arr.concat(arg1, arg2...)
```

它接受任意数量的参数 —— 数组或值都可以。

结果是一个包含来自于 `arr`，然后是 `arg1`，`arg2` 的元素的新数组。

如果参数 `argN` 是一个数组，那么其中的所有元素都会被复制。否则，将复制参数本身。

例如：

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

通常，它只复制数组中的元素。其他对象，即使它们看起来像数组一样，但仍然会被作为一个整体添加：

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

……但是，如果类似数组的对象具有 `Symbol.isConcatSpreadable` 属性，那么它就会被 `concat` 当作一个数组来处理：此对象中的元素将被添加：

```
let arr = [1, 2];

let arrayLike = {
  0: "something",
  1: "else",
  [Symbol.isConcatSpreadable]: true,
  length: 2
};

alert( arr.concat(arrayLike) ); // 1,2,something,else
```

遍历：forEach

arr.forEach 方法允许为数组的每个元素都运行一个函数。

语法：

```
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

例如，下面这个程序显示了数组的每个元素：

```
// 对每个元素调用 alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

而这段代码更详细地介绍了它们在目标数组中的位置：

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} is at index ${index} in ${array}`);
});
```

该函数的结果（如果它有返回）会被抛弃和忽略。

在数组中搜索

现在，让我们介绍在数组中进行搜索的方法。

indexOf/lastIndexOf 和 includes

arr.indexOf、arr.lastIndexOf 和 arr.includes 方法与字符串操作具有相同的语法，并且作用基本上也与字符串的方法相同，只不过这里是对数组元素而不是字符进行操作：

- `arr.indexOf(item, from)` 从索引 `from` 开始搜索 `item`，如果找到则返回索引，否则返回 `-1`。
- `arr.lastIndexOf(item, from)` —— 和上面相同，只是从右向左搜索。
- `arr.includes(item, from)` —— 从索引 `from` 开始搜索 `item`，如果找到则返回 `true`（译注：如果没找到，则返回 `false`）。

例如：

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

请注意，这些方法使用的是严格相等 `===` 比较。所以如果我们搜索 `false`，会精确到的确是 `false` 而不是数字 `0`。

如果我们想检查是否包含某个元素，并且不想知道确切的索引，那么 `arr.includes` 是首选。

此外，`includes` 的一个非常小的差别是它能正确处理 `NaN`，而不像 `indexOf/lastIndexOf`：

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1（应该为 0，但是严格相等 === equality 对 NaN 无效）
alert( arr.includes(NaN) );// true（这个结果是对的）
```

find 和 findIndex

想象一下，我们有一个对象数组。我们如何找到具有特定条件的对象？

这时可以用 arr.find 方法。

语法如下：

```
let result = arr.find(function(item, index, array) {
  // 如果返回 true，则返回 item 并停止迭代
  // 对于 falsy 则返回 undefined
});
```

依次对数组中的每个元素调用该函数：

- `item` 是元素。
- `index` 是它的索引。
- `array` 是数组本身。

如果它返回 `true`，则搜索停止，并返回 `item`。如果没有搜索到，则返回 `undefined`。

例如，我们有一个存储用户的数组，每个用户都有 `id` 和 `name` 字段。让我们找到 `id == 1` 的那个用户：

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);

alert(user.name); // John
```

在现实生活中，对象数组是很常见的，所以 `find` 方法非常有用。

注意在这个例子中，我们传给了 `find` 一个单参数函数 `item => item.id == 1`。这很典型，并且 `find` 方法的其他参数很少使用。

arr.findIndex 方法（与 `arr.find` 方法）基本上是一样的，但它返回找到元素的索引，而不是元素本身。并且在未找到任何内容时返回 `-1`。

filter

`find` 方法搜索的是使函数返回 `true` 的第一个（单个）元素。

如果需要匹配的有很多，我们可以使用 arr.filter(fn)。

语法与 `find` 大致相同，但是 `filter` 返回的是所有匹配元素组成的数组：

```
let results = arr.filter(function(item, index, array) {
  // 如果 true item 被 push 到 results，迭代继续
  // 如果什么都没找到，则返回空数组
});
```

例如：

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// 返回前两个用户的数组
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

转换数组

让我们继续学习进行数组转换和重新排序的方法。

map

arr.map 方法是最有用和经常使用的方法之一。

它对数组的每个元素都调用函数，并返回结果数组。

语法：

```
let result = arr.map(function(item, index, array) {
  // 返回新值而不是当前元素
})
```

例如，在这里我们将每个元素转换为它的字符串长度：

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

sort(fn)

arr.sort 方法对数组进行 **原位（in-place）** 排序，更改元素的顺序。(译注：原位是指在此数组内，而非生成一个新数组。)

它还返回排序后的数组，但是返回值通常会被忽略，因为修改了 `arr` 本身。

语法：

```
let arr = [ 1, 2, 15 ];

// 该方法重新排列 arr 的内容
arr.sort();

alert( arr );  // 1, 15, 2
```

你有没有注意到结果有什么奇怪的地方？

顺序变成了 `1, 15, 2`。不对，但为什么呢？

这些元素默认情况下被按字符串进行排序。

从字面上看，所有元素都被转换为字符串，然后进行比较。对于字符串，按照词典顺序进行排序，实际上应该是 `"2" > "15"`。

要使用我们自己的排序顺序，我们需要提供一个函数作为 `arr.sort()` 的参数。

该函数应该比较两个任意值并返回：

```
function compare(a, b) {
  if (a > b) return 1; // 如果第一个值比第二个值大
  if (a == b) return 0; // 如果两个值相等
  if (a < b) return -1; // 如果第一个值比第二个值小
}
```

例如，按数字进行排序：

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr);  // 1, 2, 15
```

现在结果符合预期了。

我们思考一下这儿发生了什么。`arr` 可以是由任何内容组成的数组，对吗？它可能包含数字、字符串、对象或其他任何内容。我们有一组 **一些元素**。要对其进行排序，我们需要一个 **排序函数** 来确认如何比较这些元素。默认是按字符串进行排序的。

`arr.sort(fn)` 方法实现了通用的排序算法。我们不需要关心它的内部工作原理（大多数情况下都是经过 快速排序 算法优化的）。它将遍历数组，使用提供的函数比较其元素并对其重新排序，我们所需要的就是提供执行比较的函数 `fn`。

顺便说一句，如果我们想知道要比较哪些元素 —— 那么什么都不会阻止 alert 它们：

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {
  alert( a + " <> " + b );
});
```

该算法可以在此过程中，将一个元素与多个其他元素进行比较，但是它会尝试进行尽可能少的比较。

reverse

arr.reverse 方法用于颠倒 `arr` 中元素的顺序。

例如：

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

alert( arr ); // 5,4,3,2,1
```

它也会返回颠倒后的数组 `arr`。

split 和 join

举一个现实生活场景的例子。我们正在编写一个消息应用程序，并且该人员输入以逗号分隔的接收者列表：`John, Pete, Mary`。但对我们来说，名字数组比单个字符串舒适得多。怎么做才能获得这样的数组呢？

str.split(delim) 方法可以做到。它通过给定的分隔符 `delim` 将字符串分割成一个数组。

在下面的例子中，我们用“逗号后跟着一个空格”作为分隔符：

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');

for (let name of arr) {
  alert( `A message to ${name}.` ); // A message to Bilbo（和其他名字）
}
```

`split` 方法有一个可选的第二个数字参数 —— 对数组长度的限制。如果提供了，那么额外的元素会被忽略。但实际上它很少使用：

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);

alert(arr); // Bilbo, Gandalf
```

调用带有空参数 `s` 的 `split(s)`，会将字符串拆分为字母数组：

```
let str = "test";

alert( str.split('') ); // t,e,s,t
```

arr.join(glue) 与 `split` 相反。它会在它们之间创建一串由 `glue` 粘合的 `arr` 项。

例如：

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';'); // 使用分号 ; 将数组粘合成字符串

alert( str ); // Bilbo;Gandalf;Nazgul
```

reduce/reduceRight

当我们需要遍历一个数组时 —— 我们可以使用 `forEach`，`for` 或 `for..of`。

当我们需要遍历并返回每个元素的数据时 —— 我们可以使用 `map`。

arr.reduce 方法和 arr.reduceRight 方法和上面的种类差不多，但稍微复杂一点。它们用于根据数组计算单个值。

语法是：

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

该函数一个接一个地应用于所有数组元素，并将其结果“搬运（carry on）”到下一个调用。

参数：

- `accumulator` —— 是上一个函数调用的结果，第一次等于 `initial`（如果提供了 `initial` 的话）。
- `item` —— 当前的数组元素。
- `index` —— 当前索引。
- `arr` —— 数组本身。

应用函数时，上一个函数调用的结果将作为第一个参数传递给下一个函数。

因此，第一个参数本质上是累加器，用于存储所有先前执行的组合结果。最后，它成为 `reduce` 的结果。

听起来复杂吗？

掌握这个知识点的最简单的方法就是通过示例。

在这里，我们通过一行代码得到一个数组的总和：

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

let result = arr.reduce((sum, current) => sum + current, 0);

alert(result); // 15
```

传递给 `reduce` 的函数仅使用了 2 个参数，通常这就足够了。

让我们看看细节，到底发生了什么。

1. 在第一次运行时，`sum` 的值为初始值 `initial`（`reduce` 的最后一个参数），等于 0，`current` 是第一个数组元素，等于 `1`。所以函数运行的结果是 `1`。
2. 在第二次运行时，`sum = 1`，我们将第二个数组元素（`2`）与其相加并返回。
3. 在第三次运行中，`sum = 3`，我们继续把下一个元素与其相加，以此类推……

以表格的形式表示，每一行代表的是对下一个数组元素的函数调用：

	`sum`	`current`	`result`
第 1 次调用	`0`	`1`	`1`
第 2 次调用	`1`	`2`	`3`
第 3 次调用	`3`	`3`	`6`
第 4 次调用	`6`	`4`	`10`
第 5 次调用	`10`	`5`	`15`

在这里，我们可以清楚地看到上一个调用的结果如何成为下一个调用的第一个参数。

我们也可以省略初始值：

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

// 删除 reduce 的初始值（没有 0）
let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15
```

结果是一样的。这是因为如果没有初始值，那么 `reduce` 会将数组的第一个元素作为初始值，并从第二个元素开始迭代。

计算表与上面相同，只是去掉第一行。

但是这种使用需要非常小心。如果数组为空，那么在没有初始值的情况下调用 `reduce` 会导致错误。

例如：

```
let arr = [];

// Error: Reduce of empty array with no initial value
// 如果初始值存在，则 reduce 将为空 arr 返回它（即这个初始值）。
arr.reduce((sum, current) => sum + current);
```

所以建议始终指定初始值。

arr.reduceRight 和 arr.reduce 方法的功能一样，只是遍历为从右到左。

Array.isArray

数组是基于对象的，不构成单独的语言类型。

所以 `typeof` 不能帮助从数组中区分出普通对象：

```
alert(typeof {}); // object
alert(typeof []); // same
```

……但是数组经常被使用，因此有一种特殊的方法用于判断：Array.isArray(value)。如果 `value` 是一个数组，则返回 `true`；否则返回 `false`。

```
alert(Array.isArray({})); // false

alert(Array.isArray([])); // true
```

总结

数组方法备忘单：

- 添加/删除元素：
  - `push(...items)` —— 向尾端添加元素，
  - `pop()` —— 从尾端提取一个元素，
  - `shift()` —— 从首端提取一个元素，
  - `unshift(...items)` —— 向首端添加元素，
  - `splice(pos, deleteCount, ...items)` —— 从 `pos` 开始删除 `deleteCount` 个元素，并插入 `items`。
  - `slice(start, end)` —— 创建一个新数组，将从位置 `start` 到位置 `end`（但不包括 `end`）的元素复制进去。
  - `concat(...items)` —— 返回一个新数组：复制当前数组的所有元素，并向其中添加 `items`。如果 `items` 中的任意一项是一个数组，那么就取其元素。
- 搜索元素：
  - `indexOf/lastIndexOf(item, pos)` —— 从位置 `pos` 开始搜索 `item`，搜索到则返回该项的索引，否则返回 `-1`。
  - `includes(value)` —— 如果数组有 `value`，则返回 `true`，否则返回 `false`。
  - `find/filter(func)` —— 通过 `func` 过滤元素，返回使 `func` 返回 `true` 的第一个值/所有值。
  - `findIndex` 和 `find` 类似，但返回索引而不是值。
- 遍历元素：

- `forEach(func)` —— 对每个元素都调用 `func`，不返回任何内容。
- 转换数组：
  - `map(func)` —— 根据对每个元素调用 `func` 的结果创建一个新数组。
  - `sort(func)` —— 对数组进行原位（in-place）排序，然后返回它。
  - `reverse()` —— 原位（in-place）反转数组，然后返回它。
  - `split/join` —— 将字符串转换为数组并返回。
  - `reduce(func, initial)` —— 通过对每个元素调用 `func` 计算数组上的单个值，并在调用之间传递中间结果。
- 其他：
  - `Array.isArray(arr)` 检查 `arr` 是否是一个数组。

请注意，`sort`，`reverse` 和 `splice` 方法修改的是数组本身。

将 border-left-width 转换成 borderLeftWidth

重要程度: 5

编写函数 `camelize(str)` 将诸如 “my-short-string” 之类的由短划线分隔的单词变成骆驼式的 “myShortString”。

即：删除所有短横线，并将短横线后的每一个单词的首字母变为大写。

示例：

```
camelize("background-color") == 'backgroundColor';
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

提示：使用 `split` 将字符串拆分成数组，对其进行转换之后再 `join` 回来。

解决方案

```
function camelize(str) {
  return str
    .split('-') // splits 'my-long-word' into array ['my', 'long', 'word']
    .map(
      // capitalizes first letters of all array items except the first one
      // converts ['my', 'long', 'word'] into ['my', 'Long', 'Word']
      (word, index) => index == 0 ? word : word[0].toUpperCase() + word.slice(1)
    )
    .join(''); // joins ['my', 'Long', 'Word'] into 'myLongWord'
}
```

写一个函数 `filterRange(arr, a, b)`，该函数获取一个数组 `arr`，在其中查找数值大小在 `a` 和 `b` 之间的元素，并返回它们的数组。

该函数不应该修改原数组。它应该返回新的数组。

例如：

```
let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1（匹配值）

alert( arr ); // 5,3,8,1（未修改）
```

写一个函数 `filterRangeInPlace(arr, a, b)`，该函数获取一个数组 `arr`，并删除其中介于 `a` 和 `b` 区间以外的所有值。检查：`a ≤ arr[i] ≤ b`。

该函数应该只修改数组。它不应该返回任何东西。

例如：

```
let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // 删除了范围在 1 到 4 之外的所有值

alert( arr ); // [3, 1]
```

解决方案

```
function filterRangeInPlace(arr, a, b) {

  for (let i = 0; i < arr.length; i++) {
    let val = arr[i];

    // 如果超出范围，则删除
    if (val < a || val > b) {
      arr.splice(i, 1);
      i--;
    }
  }
}

let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // 删除 1 到 4 范围之外的值

alert( arr ); // [3, 1]
```

降序排列

重要程度: 4

```
let arr = [5, 2, 1, -10, 8];

// ……你的代码以降序对其进行排序

alert( arr ); // 8, 5, 2, 1, -10
```

解决方案

```
let arr = [5, 2, 1, -10, 8];

arr.sort((a, b) => b - a);

alert( arr );
```

复制和排序数组

重要程度: 5

我们有一个字符串数组 `arr`。我们希望有一个排序过的副本，但保持 `arr` 不变。

创建一个函数 `copySorted(arr)` 返回这样一个副本。

```
let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted ); // CSS, HTML, JavaScript
alert( arr ); // HTML, JavaScript, CSS (no changes)
```

解决方案

我们可以使用 `slice()` 来创建一个副本并对其进行排序：

```
function copySorted(arr) {
  return arr.slice().sort();
}

let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted );
alert( arr );
```

创建一个可扩展的 calculator

重要程度: 5

创建一个构造函数 `Calculator`，以创建“可扩展”的 calculator 对象。

该任务由两部分组成。

1. 首先，实现 `calculate(str)` 方法，该方法接受像 `"1 + 2"` 这样格式为“数字 运算符 数字”（以空格分隔）的字符串，并返回结果。该方法需要能够理解加号 `+` 和减号 `-`。

   用法示例：

   ```
   let calc = new Calculator;

   alert( calc.calculate("3 + 7") ); // 10
   ```

2. 然后添加方法 `addMethod(name, func)`，该方法教 calculator 进行新操作。它需要运算符 `name` 和实现它的双参数函数 `func(a,b)`。

   例如，我们添加乘法 `*`，除法 `/` 和求幂 `**`：

   ```
   let powerCalc = new Calculator;
   powerCalc.addMethod("*", (a, b) => a * b);
   powerCalc.addMethod("/", (a, b) => a / b);
   powerCalc.addMethod("**", (a, b) => a ** b);

   let result = powerCalc.calculate("2 ** 3");
   alert( result ); // 8
   ```

- 此任务中没有括号或复杂的表达式。
- 数字和运算符之间只有一个空格。
- 你可以自行选择是否添加错误处理功能。

解决方案

- 请注意方法的存储方式。它们只是被添加到 `this.methods` 属性中。
- 所有检测和数字转换都通过 `calculate` 方法完成。将来可能会扩展它以支持更复杂的表达式。

```
function Calculator() {

  this.methods = {
    "-": (a, b) => a - b,
    "+": (a, b) => a + b
  };

  this.calculate = function(str) {

    let split = str.split(' '),
      a = +split[0],
      op = split[1],
      b = +split[2]

    if (!this.methods[op] || isNaN(a) || isNaN(b)) {
      return NaN;
    }

    return this.methods[op](a, b);
  }

  this.addMethod = function(name, func) {
    this.methods[name] = func;
  };
}
```

映射到 names

重要程度: 5

你有一个 `user` 对象数组，每个对象都有 `user.name`。编写将其转换为 names 数组的代码。

例如：

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = /* ... your code */

alert( names ); // John, Pete, Mary
```

解决方案

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = users.map(item => item.name);

alert( names ); // John, Pete, Mary
```

映射到对象

重要程度: 5

你有一个 `user` 对象数组，每个对象都有 `name`，`surname` 和 `id`。

编写代码以该数组为基础，创建另一个具有 `id` 和 `fullName` 的对象数组，其中 `fullName` 由 `name` 和 `surname` 生成。

例如：

```
let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [ john, pete, mary ];

let usersMapped = /* ... your code ... */

/*
usersMapped = [
  { fullName: "John Smith", id: 1 },
  { fullName: "Pete Hunt", id: 2 },
  { fullName: "Mary Key", id: 3 }
]
*/

alert( usersMapped[0].id ) // 1
alert( usersMapped[0].fullName ) // John Smith
```

所以，实际上你需要将一个对象数组映射到另一个对象数组。在这儿尝试使用箭头函数 `=>` 来编写。

解决方案

```
let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [ john, pete, mary ];

let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));

/*
usersMapped = [
  { fullName: "John Smith", id: 1 },
  { fullName: "Pete Hunt", id: 2 },
  { fullName: "Mary Key", id: 3 }
]
*/

alert( usersMapped[0].id ); // 1
alert( usersMapped[0].fullName ); // John Smith
```

请注意，在箭头函数中，我们需要使用额外的括号。

我们不能这样写：

```
let usersMapped = users.map(user => {
  fullName: `${user.name} ${user.surname}`,
  id: user.id
});
```

我们记得，有两种箭头函数的写法：直接返回值 `value => expr` 和带主体的 `value => {...}`。

JavaScript 在这里会把 `{` 视为函数体的开始，而不是对象的开始。解决方法是将它们包装在普通括号 `()` 中：

```
let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));
```

这样就可以了。

按年龄对用户排序

重要程度: 5

编写函数 `sortByAge(users)` 获得对象数组的 `age` 属性，并根据 `age` 对这些对象数组进行排序。

例如：

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// now: [john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

解决方案

```
function sortByAge(arr) {
  arr.sort((a, b) => a.age > b.age ? 1 : -1);
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let arr = [ pete, john, mary ];

sortByAge(arr);

// 排序后的数组为：[john, mary, pete]
alert(arr[0].name); // John
alert(arr[1].name); // Mary
alert(arr[2].name); // Pete
```

译注：解决方案的代码还可以更短一些

```
function sortByAge(arr) {
  arr.sort((a, b) => a.age - b.age);
}
```

因为 `sort()` 方法的语法为 `arr.sort([compareFunction])`，如果没有指明 `compareFunction`，那么元素会被按照转换为的字符串的诸个字符的 Unicode 编码进行排序，如果指明了 `compareFunction`，那么数组会按照调用该函数的返回值排序。即 `a` 和 `b` 是两个将要被比较的元素：

- 如果 `compareFunction(a, b)` 小于 `0`，那么 `a` 会被排列到 `b` 之前；
- 如果 `compareFunction(a, b)` 等于 `0`，那么 `a` 和 `b` 的相对位置不变。备注：ECMAScript 标准并不保证这一行为，而且也不是所有浏览器都会遵守（例如 Mozilla 在 2003 年之前的版本）；
- 如果 `compareFunction(a, b)` 大于 `0`，那么 `b` 会被排列到 `a` 之前。

因此，升序排列的函数可以简写为：`(a, b) => a.age - b.age`。

随机排列数组

重要程度: 3

编写函数 `shuffle(array)` 来随机排列数组的元素。

多次运行 `shuffle` 可能导致元素顺序的不同。例如：

```
let arr = [1, 2, 3];

shuffle(arr);
// arr = [3, 2, 1]

shuffle(arr);
// arr = [2, 1, 3]

shuffle(arr);
// arr = [3, 1, 2]
// ...
```

所有元素顺序应该具有相等的概率。例如，可以将 `[1,2,3]` 重新排序为 `[1,2,3]` 或 `[1,3,2]` 或 `[3,1,2]` 等，每种情况的概率相等。

解决方案

简单的解决方案可以是：

```
function shuffle(array) {
  array.sort(() => Math.random() - 0.5);
}

let arr = [1, 2, 3];
shuffle(arr);
alert(arr);
```

这样是可以的，因为 `Math.random() - 0.5` 是一个可能是正数或负数的随机数，因此排序函数会随机地对数组中的元素进行重新排序。

但是，由于排序函数并非旨在以这种方式使用，因此并非所有的排列都具有相同的概率。

例如，请考虑下面的代码。它运行 100 万次 `shuffle` 并计算所有可能结果的出现次数：

```
function shuffle(array) {
  array.sort(() => Math.random() - 0.5);
}

// 所有可能排列的出现次数
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
  '312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}
```

```
// 显示所有可能排列的出现次数
for (let key in count) {
  alert(`${key}: ${count[key]}`);
}
```

示例结果（取决于 Javascript 引擎）：

```
123: 250706
132: 124425
213: 249618
231: 124880
312: 125148
321: 125223
```

我们可以清楚地看到这种倾斜：`123` 和 `213` 的出现频率比其他情况高得多。

使用不同的 JavaScript 引擎运行这个示例代码得到的结果可能会有所不同，但是我们已经可以看到这种方法是不可靠的。

为什么它不起作用？一般来说，`sort` 是一个“黑匣子”：我们将一个数组和一个比较函数放入其中，并期望其对数组进行排序。但是由于比较的完全随机性，这个黑匣子疯了，它发疯地确切程度取决于引擎中的具体实现方法。

还有其他很好的方法可以完成这项任务。例如，有一个很棒的算法叫作 Fisher-Yates shuffle。其思路是：逆向遍历数组，并将每个元素与其前面的随机的一个元素互换位置：

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1)); // 从 0 到 i 的随机索引

    // 交换元素 array[i] 和 array[j]
    // 我们使用“解构分配（destructuring assignment）”语法来实现它
    // 你将在后面的章节中找到有关该语法的更多详细信息
    // 可以写成：
    // let t = array[i]; array[i] = array[j]; array[j] = t
    [array[i], array[j]] = [array[j], array[i]];
  }
}
```

让我们以相同的方式测试一下：

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1));
    [array[i], array[j]] = [array[j], array[i]];
  }
}

// 所有可能排列的出现次数
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
```

```
  '312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}

// 显示所有可能排列的出现次数
for (let key in count) {
  alert(`${key}: ${count[key]}`);
}
```

示例输出：

```
123: 166693
132: 166647
213: 166628
231: 167517
312: 166199
321: 166316
```

现在看起来不错：所有排列都以相同的概率出现。

另外，在性能方面，Fisher — Yates 算法要好得多，没有“排序”开销。

获取平均年龄

重要程度: 4

编写 `getAverageAge(users)` 函数，该函数获取一个具有 `age` 属性的对象数组，并返回平均年龄。

平均值的计算公式是 `(age1 + age2 + ... + ageN) / N`。

例如：

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

解决方案

```
function getAverageAge(users) {
  return users.reduce((prev, user) => prev + user.age, 0) / users.length;
}

let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // 28
```

数组去重

重要程度: 4

`arr` 是一个数组。

创建一个函数 `unique(arr)`，返回去除重复元素后的数组 `arr`。

例如：

```
function unique(arr) {
  /* your code */
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(strings) ); // Hare, Krishna, :-O
```

解决方案

让我们先遍历数字：

- 对于每个元素，我们将检查结果数组是否已经有该元素。
- 如果有，则忽略，否则将其添加到结果中。

```
function unique(arr) {
  let result = [];

  for (let str of arr) {
    if (!result.includes(str)) {
      result.push(str);
    }
  }

  return result;
}

let strings = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];
```

```
alert( unique(strings) ); // Hare, Krishna, :-O
```

代码有效，但其中存在潜在的性能问题。

方法 `result.includes(str)` 在内部遍历数组 `result`，并将每个元素与 `str` 进行比较以找到匹配项。

所以如果 `result` 中有 `100` 个元素，并且没有任何一项与 `str` 匹配，那么它将遍历整个 `result` 并进行 `100` 次比较。如果 `result` 很大，比如 `10000`，那么就会有 `10000` 次的比较。

这本身并不是问题，因为 JavaScript 引擎速度非常快，所以遍历一个有 `10000` 个元素的数组只需要几微秒。

但是我们在 `for` 循环中对 `arr` 的每个元素都进行了一次检测。

因此，如果 `arr.length` 是 `10000`，我们会有 `10000 * 10000` = 1 亿次的比较。那真的太多了。

所以该解决方案仅适用于小型数组。

从数组创建键（值）对象

重要程度: 4

假设我们收到了一个用户数组，形式为：`{id:..., name:..., age... }`。

创建一个函数 `groupById(arr)` 从该数组创建对象，以 `id` 为键（key），数组项为值。

例如:

```
let users = [
  {id: 'john', name: "John Smith", age: 20},
  {id: 'ann', name: "Ann Smith", age: 24},
  {id: 'pete', name: "Pete Peterson", age: 31},
];

let usersById = groupById(users);

/*
// 调用函数后，我们应该得到：

usersById = {
  john: {id: 'john', name: "John Smith", age: 20},
  ann: {id: 'ann', name: "Ann Smith", age: 24},
  pete: {id: 'pete', name: "Pete Peterson", age: 31},
}
*/
```

处理服务端数据时，这个函数很有用。

在这个任务里我们假设 `id` 是唯一的。没有两个具有相同 `id` 的数组项。

请在解决方案中使用数组的 `.reduce` 方法。

解决方案

```
function groupById(array) {
  return array.reduce((obj, value) => {
    obj[value.id] = value;
    return obj;
  }, {})
}
```

Map and Set（映射和集合）

Map

Map 是一个带键的数据项的集合，就像一个 `Object` 一样。 但是它们最大的差别是 `Map` 允许任何类型的键（key）。

它的方法和属性如下：

- `new Map()` —— 创建 map。
- `map.set(key, value)` —— 根据键存储值。
- `map.get(key)` —— 根据键来返回值，如果 `map` 中不存在对应的 `key`，则返回 `undefined`。
- `map.has(key)` —— 如果 `key` 存在则返回 `true`，否则返回 `false`。
- `map.delete(key)` —— 删除指定键的值。
- `map.clear()` —— 清空 map。
- `map.size` —— 返回当前元素个数。

举个例子：

```
let map = new Map();

map.set('1', 'str1');   // 字符串键
map.set(1, 'num1');     // 数字键
map.set(true, 'bool1'); // 布尔值键

// 还记得普通的 Object 吗? 它会将键转化为字符串
// Map 则会保留键的类型，所以下面这两个结果不同:
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

如我们所见，与对象不同，键不会被转换成字符串。键可以是任何类型。

Map 还可以使用对象作为键。

例如：

```
let john = { name: "John" };

// 存储每个用户的来访次数
let visitsCountMap = new Map();

// john 是 Map 中的键
visitsCountMap.set(john, 123);

alert( visitsCountMap.get(john) ); // 123
```

使用对象作为键是 `Map` 最值得注意和重要的功能之一。对于字符串键，`Object`（普通对象）也能正常使用，但对于对象键则不行。

Map 迭代

如果要在 `map` 里使用循环，可以使用以下三个方法：

- `map.keys()` —— 遍历并返回所有的键（returns an iterable for keys），
- `map.values()` —— 遍历并返回所有的值（returns an iterable for values），
- `map.entries()` —— 遍历并返回所有的实体（returns an iterable for entries）`[key, value]`，`for..of` 在默认情况下使用的就是这个。

例如：

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);

// 遍历所有的键（vegetables）
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // cucumber, tomatoes, onion
}

// 遍历所有的值（amounts）
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

// 遍历所有的实体 [key, value]
for (let entry of recipeMap) { // 与 recipeMap.entries() 相同
  alert(entry); // cucumber,500 (and so on)
}
```

使用插入顺序

迭代的顺序与插入值的顺序相同。与普通的 `Object` 不同，`Map` 保留了此顺序。

除此之外，`Map` 有内置的 `forEach` 方法，与 `Array` 类似：

```
// 对每个键值对 (key, value) 运行 forEach 函数
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // cucumber: 500 etc
});
```

解构赋值

JavaScript 中最常用的两种数据结构是 `Object` 和 `Array`。

对象让我们能够创建通过键来存储数据项的单个实体，数组则让我们能够将数据收集到一个有序的集合中。

但是，当我们把它们传递给函数时，它可能不需要一个整体的对象/数组，而是需要单个块。

解构赋值 是一种特殊的语法，它使我们可以将数组或对象“拆包”为到一系列变量中，因为有时候使用变量更加方便。解构操作对那些具有很多参数和默认值等的函数也很奏效。

数组解构

下面是一个将数组解构到变量中的例子：

```
// 我们有一个存放了名字和姓氏的数组
let arr = ["Ilya", "Kantor"]

// 解构赋值
// sets firstName = arr[0]
// and surname = arr[1]
let [firstName, surname] = arr;

alert(firstName); // Ilya
alert(surname);  // Kantor
```

忽略使用逗号的元素

数组中不想要的元素也可以通过添加额外的逗号来把它丢弃：

```
// 不需要第二个元素
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman
Republic"];

alert( title ); // Consul
```

在上面的代码中，数组的第二个元素被跳过了，第三个元素被赋值给了 `title` 变量，数组中剩下的元素也都被跳过了（因为在这没有对应给它们的变量）。

赋值给等号左侧的任何内容

我们可以在等号左侧使用任何“可以被赋值的”东西。

例如，一个对象的属性：

```
let user = {};
[user.name, user.surname] = "Ilya Kantor".split(' ');

alert(user.name); // Ilya
```

交换变量值的技巧

一个用于交换变量值的典型技巧：

```
let guest = "Jane";
let admin = "Pete";

// 交换值：让 guest=Pete, admin=Jane
[guest, admin] = [admin, guest];

alert(`${guest} ${admin}`); // Pete Jane（成功交换！）
```

这里我们创建了一个由两个变量组成的临时数组，并且立即以交换了的顺序对其进行了解构。

我们可以用这种方式交换两个以上的变量。

剩余的 '...'

如果我们不只是要获得第一个值，还要将后续的所有元素都收集起来 — 我们可以使用三个点 `"..."` 来再加一个参数来接收"剩余的"元素：

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius
alert(name2); // Caesar

// 请注意，`rest` 的类型是数组
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

`rest` 的值就是数组中剩下的元素组成的数组。不一定要使用变量名 `rest`，我们也可以使用其他的变量名，只要确保它前面有三个点，并且在解构赋值的最后一个参数位置上就行了。

日期和时间

```
new Date(year, month, date, hours, minutes, seconds, ms)
```

使用当前时区中的给定组件创建日期。只有前两个参数是必须的。

- `year` 必须是四位数：`2013` 是合法的，`98` 是不合法的。
- `month` 计数从 `0`（一月）开始，到 `11`（十二月）结束。
- `date` 是当月的具体某一天，如果缺失，则为默认值 `1`。
- 如果 `hours/minutes/seconds/ms` 缺失，则均为默认值 `0`。

Date.now()

如果我们仅仅想要测量时间间隔，我们不需要 `Date` 对象。

有一个特殊的方法 `Date.now()`，它会返回当前的时间戳。

它相当于 `new Date().getTime()`，但它不会创建中间的 `Date` 对象。因此它更快，而且不会对垃圾处理造成额外的压力。

这种方法很多时候因为方便，又或是因性能方面的考虑而被采用，例如使用 JavaScript 编写游戏或其他的特殊应用场景。

创建日期

重要程度: 5

创建一个 `Date` 对象，日期是：Feb 20, 2012, 3:12am。时区是当地时区。

使用 `alert` 显示结果。

解决方案

`new Date` 构造函数默认使用本地时区。所以唯一需要牢记的就是月份从 0 开始计数。

所以二月对应的数值是 1。

```
let d = new Date(2012, 1, 20, 3, 12);
alert( d );
```

许多天之前是哪个月几号?

重要程度: 4

写一个函数 `getDateAgo(date, days)`，返回特定日期 `date` 往前 `days` 天是哪个月的哪一天。

例如，假设今天是 20 号，那么 `getDateAgo(new Date(), 1)` 的结果应该是 19 号，`getDateAgo(new Date(), 2)` 的结果应该是 18 号。

跨月、年也应该是正确输出：

```
let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

P.S. 函数不应该修改给定的 `date` 值。

解决方案

思路很简单：从 `date` 中减去给定的天数：

```
function getDateAgo(date, days) {
  date.setDate(date.getDate() - days);
  return date.getDate();
}
```

……但是函数不能修改 `date`。这一点很重要，因为我们提供日期的外部代码不希望它被修改。

要实现这一点，我们可以复制这个日期，就像这样：

```
function getDateAgo(date, days) {
  let dateCopy = new Date(date);

  dateCopy.setDate(date.getDate() - days);
  return dateCopy.getDate();
}

let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

某月的最后一天？

重要程度: 5

写一个函数 `getLastDayOfMonth(year, month)` 返回 month 月的最后一天。有时候是 30，有时是 31，甚至在二月的时候会是 28/29。

参数：

- `year` —— 四位数的年份，比如 2012。
- `month` —— 月份，从 0 到 11。

举个例子，`getLastDayOfMonth(2012, 1) = 29`（闰年，二月）

解决方案

让我们使用下个月创建日期，但将零作为天数（day）传递：

```
function getLastDayOfMonth(year, month) {
  let date = new Date(year, month + 1, 0);
  return date.getDate();
}

alert( getLastDayOfMonth(2012, 0) ); // 31
alert( getLastDayOfMonth(2012, 1) ); // 29
alert( getLastDayOfMonth(2013, 1) ); // 28
```

通常，日期从 1 开始，但从技术上讲，我们可以传递任何数字，日期会自动进行调整。因此，当我们传递 0 时，它的意思是“一个月的第一天的前一天”，换句话说：“上个月的最后一天”。

JSON 方法，toJSON

假设我们有一个复杂的对象，我们希望将其转换为字符串，以通过网络发送，或者只是为了在日志中输出它。

当然，这样的字符串应该包含所有重要的属性。

我们可以像这样实现转换：

```
let user = {
  name: "John",
  age: 30,

  toString() {
    return `{name: "${this.name}", age: ${this.age}}`;
  }
};

alert(user); // {name: "John", age: 30}
```

闭包 是指内部函数总是可以访问其所在的外部函数中声明的变量和参数，即使在其外部函数被返回（寿命终结）了之后。在某些编程语言中，这是不可能的，或者应该以特殊的方式编写函数来实现。但是如上所述，在 JavaScript 中，所有函数都是天生闭包的

函数会选择最新的内容吗？

重要程度: 5

函数 sayHi 使用外部变量。当函数运行时，将使用哪个值？

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi(); // 会显示什么："John" 还是 "Pete"？
```

这种情况在浏览器和服务器端开发中都很常见。一个函数可能被计划在创建之后一段时间后才执行，例如在用户行为或网络请求之后。

因此，问题是：它会接收最新的修改吗？

解决方案

答案：**Pete**。

函数将从内到外依次在对应的词法环境中寻找目标变量，它使用最新的值。

旧变量值不会保存在任何地方。当一个函数想要一个变量时，它会从自己的词法环境或外部词法环境中获取当前值。

if 内的函数

看看下面这个代码。最后一行代码的执行结果是什么？

```
let phrase = "Hello";

if (true) {
  let user = "John";

  function sayHi() {
    alert(`${phrase}, ${user}`);
  }
}

sayHi();
```

解决方案

答案：**error**。

函数 `sayHi` 是在 `if` 内声明的，所以它只存在于 `if` 中。外部是没有 `sayHi` 的。

闭包 sum

重要程度: 4

编写一个像 `sum(a)(b) = a+b` 这样工作的 `sum` 函数。

是的，就是这种通过双括号的方式（并不是错误）。

举个例子：

```
sum(1)(2) = 3
sum(5)(-1) = 4
```

解决方案

为了使第二个括号有效，第一个（括号）必须返回一个函数。

就像这样：

```
function sum(a) {

  return function(b) {
    return a + b; // 从外部词法环境获得 "a"
  };

}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1) ); // 4
```

变量可见吗?

重要程度: 4

下面这段代码的结果会是什么?

```
let x = 1;

function func() {
  console.log(x); // ?

  let x = 2;
}

func();
```

P.S. 这个任务有一个陷阱。解决方案并不明显。

解决方案

答案：**error**。

你运行一下试试：

```
let x = 1;

function func() {
  console.log(x); // ReferenceError: Cannot access 'x' before initialization
  let x = 2;
}

func();
```

在这个例子中，我们可以观察到“不存在”的变量和“未初始化”的变量之间的特殊差异。

函数进阶内容

有一个名为 `arguments` 的特殊的类数组对象，该对象按参数索引包含所有参数。

例如：

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // 它是可遍历的
  // for(let arg of arguments) alert(arg);
}

// 依次显示：2，Julius，Caesar
showName("Julius", "Caesar");
```

```
// 依次显示：1，Ilya，undefined（没有第二个参数）
showName("Ilya");
```

箭头函数是没有 "arguments"

如果我们在箭头函数中访问 `arguments`，访问到的 `arguments` 并不属于箭头函数，而是属于箭头函数外部的“普通”函数。

举个例子：

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg();
}

f(1); // 1
```

我们已经知道，箭头函数没有自身的 `this`。现在我们知道了它们也没有特殊的 `arguments` 对象。

Spread 语法

例如，内建函数 Math.max 会返回参数中最大的值：

```
alert( Math.max(3, 5, 1) ); // 5
```

假如我们有一个数组 `[3, 5, 1]`，我们该如何用它调用 `Math.max` 呢？

直接把数组“原样”传入是不会奏效的，因为 `Math.max` 希望你传入一个列表形式的数值型参数，而不是一个数组：

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

以 `Math.max` 为例：

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5（spread 语法把数组转换为参数列表）
```

我们还可以通过这种方式传递多个可迭代对象：

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8
```

并且，我们还可以使用 spread 语法来合并数组：

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15（0，然后是 arr，然后是 2，然后是 arr2）
```

全局对象

全局对象提供可在任何地方使用的变量和函数。默认情况下，这些全局变量内置于语言或环境中。

在浏览器中，它的名字是 “window”，对 Node.js 而言，它的名字是 “global”，其它环境可能用的是别的名字。

全局对象的所有属性都可以被直接访问：

```
alert("Hello");
// 等同于
window.alert("Hello");
```

在浏览器中，使用 `var`（而不是 `let/const`！）声明的全局函数和变量会成为全局对象的属性。

```
var gVar = 5;

alert(window.gVar); // 5（成为了全局对象的属性）
```

如果我们使用 `let`，就不会发生这种情况：

```
let gLet = 5;

alert(window.gLet); // undefined（不会成为全局对象的属性）
```

如果一个值非常重要，以至于你想使它在全局范围内可用，那么可以直接将其作为属性写入：

```
// 将当前用户信息全局化，以允许所有脚本访问它
window.currentUser = {
  name: "John"
};

// 代码中的另一个位置
alert(currentUser.name);  // John

// 或者，如果我们有一个名为 "currentUser" 的局部变量
// 从 window 显示地获取它（这是安全的！）
alert(window.currentUser.name); // John
```

也就是说，一般不建议使用全局变量。全局变量应尽可能的少。与使用外部变量或全局变量相比，函数获取“输入”变量并产生特定“输出”的代码设计更加清晰，不易出错且更易于测试。

调度：setTimeout 和 setInterval

有时我们并不想立即执行一个函数，而是等待特定一段时间之后再执行。这就是所谓的“计划调用（scheduling a call）”。

目前有两种方式可以实现：

- `setTimeout` 允许我们将函数推迟到一段时间间隔之后再执行。
- `setInterval` 允许我们重复运行一个函数，从一段时间间隔之后开始运行，之后以该时间间隔连续重复运行该函数。

这两个方法并不在 JavaScript 的规范中。但是大多数运行环境都有内建的调度程序，并且提供了这些方法。目前来讲，所有浏览器以及 Node.js 都支持这两个方法。

setTimeout

语法：

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

参数说明：

- `func|code`

  想要执行的函数或代码字符串。 一般传入的都是函数。由于某些历史原因，支持传入代码字符串，但是不建议这样做。

- `delay`

  执行前的延时，以毫秒为单位（1000 毫秒 = 1 秒），默认值是 0；

- `arg1`，`arg2`…

  要传入被执行函数（或代码字符串）的参数列表（IE9 以下不支持）

例如，在下面这个示例中，`sayHi()` 方法会在 1 秒后执行：

```
function sayHi() {
  alert('Hello');
}

setTimeout(sayHi, 1000);
```

带参数的情况：

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

不建议使用字符串，我们可以使用箭头函数代替它们，如下所示：

```
setTimeout(() => alert('Hello'), 1000);
```

深入理解箭头函数

JavaScript 充满了我们需要编写在其他地方执行的小函数的情况。

例如：

- `arr.forEach(func)` —— `forEach` 对每个数组元素都执行 `func`。
- `setTimeout(func)` —— `func` 由内建调度器执行。
- ……还有更多。

JavaScript 的精髓在于创建一个函数并将其传递到某个地方。

在这样的函数中，我们通常不想离开当前上下文。这就是箭头函数的主战场啦。

箭头函数没有 “this”

箭头函数没有 `this`。如果访问 `this`，则会从外部获取。

例如，我们可以使用它在对象方法内部进行迭代：

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

这里 `forEach` 中使用了箭头函数，所以其中的 `this.title` 其实和外部方法 `showList` 的完全一样。那就是：`group.title`。

如果我们使用正常的函数，则会出现错误：

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title' of undefined
      alert(this.title + ': ' + student)
    });
  }
};

group.showList();
```

报错是因为 `forEach` 运行它里面的这个函数，但是这个函数的 `this` 为默认值 `this=undefined`，因此就出现了尝试访问 `undefined.title` 的情况。

但箭头函数就没事，因为它们没有 `this`。

箭头函数 VS bind

箭头函数 `=>` 和使用 `.bind(this)` 调用的常规函数之间有细微的差别：

- `.bind(this)` 创建了一个该函数的“绑定版本”。
- 箭头函数 `=>` 没有创建任何绑定。箭头函数只是没有 `this`。`this` 的查找与常规变量的搜索方式完全相同：在外部词法环境中查找。

总结

箭头函数：

- 没有 `this`
- 没有 `arguments`
- 不能使用 `new` 进行调用