let 和 const 命令

1. let 命令

1.1. 基本用法

```
var a = [];
for (var i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 10
```

上面代码中，变量 `i` 是 `var` 命令声明的，在全局范围内都有效，所以全局只有一个变量 `i`。每一次循环，变量 `i` 的值都会发生改变，而循环内被赋给数组 `a` 的函数内部的 `console.log(i)`，里面的 `i` 指向的就是全局的 `i`。也就是说，**所有数组`a` 的成员里面的 `i`，指向的都是同一个 `i`**，导致运行时输出的是最后一轮的 `i` 的值，也就是 10。

如果使用let，声明的变量仅在块级作用域内有效，最后输出的是 6。

```
var a = [];
for (let i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 6
```

上面代码中，变量 `i` 是 `let` 声明的，**当前的 `i` 只在本轮循环有效，所以每一次循环的 `i` 其实都是一个新的变量**，所以最后输出的是 6。你可能会问，如果每一轮循环的变量 `i` 都是重新声明的，那它怎么知道上一轮循环的值，从而计算出本轮循环的值？这是因为 JavaScript 引擎内部会记住上一轮循环的值，初始化本轮的变量 `i` 时，就在上一轮循环的基础上进行计算。

`for` 循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

```
for (let i = 0; i < 3; i++) {
  let i = 'abc';
  console.log(i);
}

// abc
// abc
// abc
```

上面代码正确运行，输出了 3 次 `abc`。**这表明函数内部的变量 `i` 与循环变量 `i` 不在同一个作用域，有各自单独的作用域（同一个作用域不可使用 let 重复声明同一个变量）。**

1.2. 暂时性死区

如果区块中存在 `let` 和 `const` 命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

“暂时性死区”也意味着 `typeof` 不再是一个百分之百安全的操作。

```
typeof x; // ReferenceError
let x;
```

上面代码中，变量 `x` 使用 `let` 命令声明，所以在声明之前，都属于 `x` 的“死区”，只要用到该变量就会报错。因此，`typeof` 运行时就会抛出一个 `ReferenceError`。

作为比较，如果一个变量根本没有被声明，使用typeof反而不会报错。

```
typeof undeclared_variable // "undefined"
```

上面代码中，`undeclared_variable` 是一个不存在的变量名，结果返回 “undefined”。所以，在没有 `let` 之前，`typeof` 运算符是百分之百安全的，永远不会报错。现在这一点不成立了。这样的设计是为了让大家养成良好的编程习惯，变量一定要在声明之后使用，否则就报错。

1.3. 不允许重复申明

`let` 不允许在相同作用域内，重复声明同一个变量。

```
// 报错
function func() {
  let a = 10;
  var a = 1;
}

// 报错
function func() {
  let a = 10;
  let a = 1;
}
```

不能在函数内部重新声明参数。

```
function func(arg) {
  let arg;
}
func() // 报错

function func(arg) {
  {
    let arg;
  }
}
func() // 不报错
```

2. 块级作用域

块级作用域的出现，使得匿名立即执行函数表达式（匿名 IIFE）不再必要了。

```
// IIFE 写法
(function () {
  var tmp = 3;
  console.log(tmp);
}());
// 3

// 块级作用域写法
{
  let tmp = 4;
  console.log(tmp);
}
// 4
```

考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。 如果确实需要，也应该写成函数表达式，而不是函数声明语句。

```
// 块级作用域内部的函数声明语句，建议不要使用
{
  let a = 'secret';
  function f() {
    return a;
  }
}

// 块级作用域内部，优先使用函数表达式
{
  let a = 'secret';
  let f = function () {
    return a;
  };
}
```

ES6 的块级作用域必须有大括号，如果没有大括号，JavaScript 引擎就认为不存在块级作用域。

```
// 第一种写法，报错
if (true) let x = 1;
// SyntaxError: Lexical declaration cannot appear in a single-statement context 词法声明不能出现在单语句上下文中

// 第二种写法，不报错
if (true) {
  let x = 1;
}
```

上面代码中，第一种写法没有大括号，所以不存在块级作用域，而 `let` 只能出现在当前作用域的顶层，所以报错。第二种写法有大括号，所以块级作用域成立。

3. const 命令

`const` 声明一个只读的常量。一旦声明，常量的值就不能改变。

```
const PI = 3.1415;
PI // 3.1415

PI = 3; // TypeError: Assignment to constant variable.
```

`const` 声明的变量不得改变值，这意味着，`const` 一旦声明变量，就必须立即初始化，不能留到以后赋值。

```
const foo;
// SyntaxError: Missing initializer in const declaration
```

3.1. ES6 声明变量的方法

ES6 声明变量的方法有六种：`var`、`function`、`let`、`const`、`import`、`class`。

（1）. 使用 `let` 关键字：`let` 是块级作用域，只在声明它的块或子块中可见。它允许你在同一作用域中多次声明相同的变量。

```
let x = 10;
let x = 20; // 重新声明 x
```

（2）. 使用 `const` 关键字：`const` 也是块级作用域，但一旦赋值就不能改变。这使得 `const` 很适合用于你不想改变的值。

```
const PI = 3.14159;
```

（3）. 使用 `var` 关键字：这是在 ES6 之前的主要变量声明方法，但在 ES6 中，`var` 的使用已经被 `let` 和 `const` 取代。`var` 的作用域是函数级的，而不是块级的。

```
var x = 10;
```

（4）. 使用 function 声明来定义变量。这种方式是在函数的参数列表中使用变量名，并在函数体内部使用该变量。

例如，下面的代码使用函数声明定义了一个名为 `myVariable` 的变量：

```
function myFunction(myVariable) {
  // 在函数体内部使用 myVariable
  console.log(myVariable);
}
```

然后，可以通过调用 `myFunction` 函数并将参数传递给它来设置 `myVariable` 的值：

```
myFunction("Hello, world!");
```

在上面的示例中，`myVariable` 的值将被设置为字符串 `"Hello, world!"`，并且该值将被打印到控制台中。

（5）. 使用 `class` 关键字声明类，并在类中声明变量：这是在 ES6 中引入的新特性，允许你使用面向对象编程。

```
class MyClass {
  constructor() {
    this.x = 10;
  }
}
```

（6）. 使用 `import` 和 `export` 进行模块化的变量声明和导出：这也是在 ES6 中引入的新特性，允许你创建独立的模块并在需要时导入。

```
// moduleA.js
export const x = 10;

// moduleB.js
import { x } from './moduleA';
```

4. globalThis 对象

JavaScript 语言存在一个顶层对象，它提供全局环境（即全局作用域），所有代码都是在这个环境中运行。但是，顶层对象在各种实现里面是不统一的。

- 浏览器环境，顶层对象是 `window` 和 `self`。
- `Web Worker` 环境，顶层对象是 `self`。
- `Node` 环境，顶层对象是 `global`。

ES2020 在语言标准的层面，引入 `globalThis` 作为顶层对象。也就是说，任何环境下，`globalThis` 都是存在的，都可以从它拿到顶层对象，指向全局环境下的 `this`。

变量的解构赋值

（1）交换变量的值

```
let x = 1, y = 2;
[x, y] = [y, x];
x // 2
y // 1
```

当没有解构赋值的语法时，只能下面这样写：

```
let  x= 1, y = 2, z; // 引入第三个变量 z
z = x;
x = y;
y = z;
x // 2
y // 1
```

第一种写法，交换变量 x 和 y 的值，这样的写法不仅简洁，而且易读，语义非常清晰。

（2）从函数返回多个值

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

```
// 返回一个数组
function example() {
  return [1, 2, 3];
}
let [a, b, c] = example();

// 返回一个对象
function example() {
  return {
    foo: 1,
    bar: 2
  };
}
let { foo, bar } = example();
```

（3）函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有次序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

// 参数是一组无次序的值
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

（4）提取 JSON 数据

解构赋值对提取 JSON 对象中的数据，尤其有用。

```
let jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};
let { id, status, data: number } = jsonData;
console.log(id, status, number); // 42, "OK", [867, 5309]
```

（5）函数参数的默认值

```
jQuery.ajax = function (url, {
  async = true,
  beforeSend = function () {},
  cache = true,
  complete = function () {},
  crossDomain = false,
  global = true,
  // ... more config
} = {}) {
  // ... do stuff
};
```

指定参数的默认值，就避免了在函数体内部再写 `var foo = config.foo || 'default foo';` 这样的语句。

（6）遍历 Map 结构

任何部署了 Iterator 接口的对象，都可以用 `for...of` 循环遍历。

```
const map = new Map();
map.set('first', 'hello');
map.set('second', 'world');

for (let [key, value] of map) {
  console.log(key + " is " + value);
}
```

```
// first is hello
// second is world
```

如果只想获取键名，或者只想获取键值，可以写成下面这样。

```
// 获取键名
for (let [key] of map) {
  // ...
}

// 获取键值
for (let [,value] of map) {
  // ...
}
```

（7）输入模块的指定方法

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

```
const { SourceMapConsumer, SourceNode } = require("source-map");
```

字符串的扩展

1. 字符串的遍历器接口

ES6 为字符串添加了遍历器接口，使得字符串可以被 `for...of` 循环遍历。

```
for (let codePoint of 'foo') {
  console.log(codePoint)
}
// "f"
// "o"
// "o"
```

2. 模板字符串

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

```
// 普通字符串
`In JavaScript '\n' is a line-feed.`

// 多行字符串
`In JavaScript this is
 not legal.`

console.log(`string text line 1
string text line 2`);

// 字符串中嵌入变量
let name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`
```

如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。

```
let greeting = `\`Yo\` World!`;
```

字符串的新增方法

1. 实例方法：includes(), startsWith(), endsWith()

传统上，JavaScript 只有 `indexOf()` 方法，可以用来确定一个字符串是否包含在另一个字符串中。ES6 又提供了三种新方法。

- `includes()`：返回布尔值，表示是否找到了参数字符串。
- `startsWith()`：返回布尔值，表示参数字符串是否在原字符串的头部。
- `endsWith()`：返回布尔值，表示参数字符串是否在原字符串的尾部。

```
let s = 'Hello world!';

s.startsWith('Hello') // true
s.endsWith('!') // true
s.includes('o') // true
```

这三个方法都支持第二个参数，表示开始搜索的位置。

```
let s = 'Hello world!';

s.startsWith('world', 6) // true
s.endsWith('Hello', 5) // true
s.includes('Hello', 6) // false
```

使用第二个参数 `n` 时，`endsWith` 的行为与其他两个方法有所不同。它针对前 `n` 个字符，而其他两个方法针对从第 `n` 个位置直到字符串结束。

2. 实例方法：repeat()

`repeat` 方法返回一个新字符串，表示将原字符串重复 `n` 次。

```
'x'.repeat(3); // "xxx"
'hello'.repeat(2); // "hellohello"
'na'.repeat(0); // ""
```

参数如果是小数，会被取整。

```
'na'.repeat(2.9) // "nana"
```

如果 `repeat` 的参数是负数或者 `Infinity`，会报错。

```
'na'.repeat(Infinity); // RangeError
'na'.repeat(-1); // RangeError
```

但是，如果参数是 0 到 -1 之间的小数，则等同于 0，这是因为会先进行取整运算。0 到 -1 之间的小数，取整以后等于 `-0`，`repeat` 视同为 0。

```
'na'.repeat(-0.9); // ""
```

参数 `NaN` 等同于 0。

```
'na'.repeat(NaN); // ""
```

如果 `repeat` 的参数是字符串，则会先转换成数字。

```
'na'.repeat('na'); // ""
'na'.repeat('3'); // "nanana"
```

3. 实例方法：padStart()，padEnd()

如果某个字符串不够指定长度，会在头部或尾部补全。`padStart()` 用于头部补全，`padEnd()` 用于尾部补全。

```
'x'.padStart(5, 'ab') // 'ababx'
'x'.padStart(4, 'ab') // 'abax'

'x'.padEnd(5, 'ab') // 'xabab'
'x'.padEnd(4, 'ab') // 'xaba'
```

`padStart()` 和 `padEnd()` 一共接受两个参数，第一个参数是字符串补全生效的最大长度，第二个参数是用来补全的字符串。

如果原字符串的长度，等于或大于最大长度，则字符串补全不生效，返回原字符串。

```
'xxx'.padStart(2, 'ab') // 'xxx'
'xxx'.padEnd(2, 'ab') // 'xxx'
```

如果用来补全的字符串与原字符串，两者的长度之和超过了最大长度，则会截去超出位数的补全字符串。

```
'abc'.padStart(10, '0123456789'); // '0123456abc'
```

如果省略第二个参数，默认使用空格补全长度。

```
'x'.padStart(4) // '   x'
'x'.padEnd(4) // 'x   '
```

`padStart()` 的常见用途是为数值补全指定位数。下面代码生成 10 位的数值字符串。

```
'1'.padStart(10, '0') // "0000000001"
'12'.padStart(10, '0') // "0000000012"
'123456'.padStart(10, '0') // "0000123456"
```

另一个用途是提示字符串格式。

```
'12'.padStart(10, 'YYYY-MM-DD') // "YYYY-MM-12"
'09-12'.padStart(10, 'YYYY-MM-DD') // "YYYY-09-12"
```

4. 实例方法：trimStart()，trimEnd()

`trimStart()` 和 `trimEnd()` 这两个方法。它们的行为与 `trim()` 一致，`trimStart()` 消除字符串头部的空格，`trimEnd()` 消除尾部的空格。**它们返回的都是新字符串，不会修改原始字符串。**

```
const s = '  abc  ';

s.trim(); // "abc"
s.trimStart(); // "abc  "
s.trimEnd(); // "  abc"
s; // '  abc  '
```

除了空格键，这两个方法对字符串头部（或尾部）的 tab 键、换行符等不可见的空白符号也有效。

浏览器还部署了额外的两个方法，`trimLeft()` 是 `trimStart()` 的别名，`trimRight()` 是 `trimEnd()` 的别名。

5. 实例方法：replaceAll()

历史上，字符串的实例方法 `replace()` 只能替换第一个匹配。

```
'aabbcc'.replace('b', '_')
// 'aa_bcc'
```

如果要替换所有的匹配，不得不使用正则表达式的g修饰符。

```
'aabbcc'.replace(/b/g, '_')
// 'aa__cc'
```

`replaceAll()` 方法，可以一次性替换所有匹配。

```
'aabbcc'.replaceAll('b', '_')
// 'aa__cc'
```

6. 实例方法：at()

`at()` 方法接受一个整数作为参数，返回参数指定位置的字符，支持负索引（即倒数的位置）。

```
const str = 'hello';
str.at(1) // "e"
str.at(-1) // "o"
str.at(0) // "h"
str.at() // "h" ，不传参数取第一位
str.at(9) // undefined
str.at(-9) // undefined
```

如果参数位置超出了字符串范围，`at()` 返回 `undefined`。

数值的扩展

1. 二进制和八进制表示法

二进制（binary）和八进制（octonary）数值的新的写法，分别用前缀 `0b`（或 `0B`）和 `0o`（或 `0O`）表示。十六进制用前缀 `0x` 表示：

```
0b10011 === 19 // true, (1 * 2 ** 4) + (1 * 2 ** 1) + (1 * 2 ** 0 )
0o23 === 19 // true, (2 * 8 ** 1) + (3 * 8 ** 0)
0b10011 === 0o23 // true

0x13 === 19 // 前缀 0x 表示十六进制
```

如果要将 `0b`、`0o` 和 `0x` 前缀的字符串数值转为十进制，要使用 `Number()` 方法：

```
Number('0b111');  // 或 Number('0b111', 10),  7
Number('0o10');  // 或 Number('0o10', 10),  8
Number('0x18'); // 或 Number('0x18', 10), 24
```

如果要将十进制数转换为二进制、八进制、十六进制，需要使用 `toString()` 方法：

```
(19).toString(2)  // '10010'
(19).toString(8)  // '23'
(19).toString(16)  // '13'
```

2. 数值分隔符

数值使用下划线（_）作为分隔符。

```
1_000 === 1000; // true
```

数值分隔符没有指定间隔的位数，可以每三位添加一个分隔符，也可以每一位、每两位、每四位添加一个...甚至可以随意位数间隔：

```
1_2_3_4 === 1234; // true, 每一位添加分隔符
12_34_56 === 123456; // true, 每二位添加分隔符
123_456_789 === 123456789; // true, 每三位添加分隔符
1234_5678_9000 === 123456789000; // true, 每四位添加分隔符
1_23_456_789 === 123456789; // true, 随意位数间隔添加分隔符
```

数值分隔符不允许的写法：

> - 不能放在数值的最前面（leading）或最后面（trailing）。
> - 不能两个或两个以上的分隔符连在一起。
> - 小数点的前后不能有分隔符。
> - 科学计数法里面，表示指数的e或E前后不能有分隔符。

```
// 下面写法全部报错
_1464301 // 分隔符不能放在数值的最前面
1464301_ // 分隔符不能放在数值的最后面
123__456 // Only one underscore is allowed as numeric separator
3_.141 // 小数点前面不能有分隔符
3._141 // 小数点后面不能有分隔符
1_e12 // e 前面不能有分隔符
1e_12 // e 后面不能有分隔符
```

数值分隔符只是一种书写便利，对于 JavaScript 内部数值的存储和输出，并没有影响。

数值分隔符主要是为了编码时书写数值的方便，而不是为了处理外部输入的数据。 下面三个将字符串转成数值的函数，不支持数值分隔符：

> - Number()
> - parseInt()
> - parseFloat()

```
Number('123_456') // NaN
parseInt('123_456') // 123
```

3. 安全整数和 Number.isSafeInteger()

JavaScript 能够准确表示的整数范围在 `-2^53` 到 `2^53` 之间（不含两个端点），超过这个范围，无法精确表示这个值。

```
Math.pow(2, 53) // 9007199254740992
Math.pow(2, 53) === Math.pow(2, 53) + 1; // true

9007199254740992 === 9007199254740993;  // true
```

上面代码中，超出 2 的 53 次方之后，一个数就不精确了。

`Number.MAX_SAFE_INTEGER` 和 `Number.MIN_SAFE_INTEGER` 这两个常量，用来表示这个范围的上下限。

```
Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1; // true
Number.MAX_SAFE_INTEGER === 9007199254740991; // true
```

```
Number.MIN_SAFE_INTEGER === -Math.pow(2, 53) + 1; // true
Number.MIN_SAFE_INTEGER === -9007199254740991; // true

Number.MIN_SAFE_INTEGER === -Number.MAX_SAFE_INTEGER; // true
```

`Number.isSafeInteger()` 则是用来判断一个整数是否落在这个范围之内。

```
Number.isSafeInteger('a') // false
Number.isSafeInteger(null) // false
Number.isSafeInteger(NaN) // false
Number.isSafeInteger(Infinity) // false
Number.isSafeInteger(-Infinity) // false

Number.isSafeInteger(3) // true
Number.isSafeInteger(1.2) // false
Number.isSafeInteger(9007199254740990) // true
Number.isSafeInteger(9007199254740992) // false

Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1) // false
Number.isSafeInteger(Number.MIN_SAFE_INTEGER) // true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER) // true
Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1) // false
```

这个函数的实现很简单，就是跟安全整数的两个边界值比较一下。

```
Number.isSafeInteger = function (n) {
  return (typeof n === 'number' && Math.round(n) === n && Number.MIN_SAFE_INTEGER
<= n && n <= Number.MAX_SAFE_INTEGER);
}
```

实际使用这个函数时，需要注意。验证运算结果是否落在安全整数的范围内，不要只验证运算结果，而要同时验证参与运算的每个值。

```
Number.isSafeInteger(9007199254740993); // false
Number.isSafeInteger(990); // true
Number.isSafeInteger(9007199254740993 - 990); // true

9007199254740993 - 990; // 返回结果 9007199254740002, // 正确答案应该是
9007199254740003
```

`9007199254740993` 不是一个安全整数，但是 `Number.isSafeInteger` 会返回结果，显示计算结果是安全的。这是因为，这个数超出了精度范围，导致在计算机内部，以 `9007199254740992` 的形式储存。`9007199254740993 === 9007199254740992`。所以，如果只验证运算结果是否为安全整数，很可能得到错误结果。下面的函数可以同时验证两个运算数和运算结果：

```
function trusty (left, right, result) {
  if (Number.isSafeInteger(left) && Number.isSafeInteger(right) &&
Number.isSafeInteger(result)) {
    return result;
  }
  throw new RangeError('Operation cannot be trusted!');
}

trusty(9007199254740993, 990, 9007199254740993 - 990); // RangeError: Operation
cannot be trusted!
trusty(1, 2, 3); // 3
```

BigInt 数据类型

1. 简介

JavaScript 的数值表示有两大限制：

> - 数值的精度只能到 53 个二进制位（相当于 16 个十进制位），大于这个范围的整数，JavaScript 是无法精确表示。
> - 大于或等于 2 的 1024 次方的数值，JavaScript 无法表示，会返回 `Infinity`。

```
Math.pow(2, 53) === Math.pow(2, 53) + 1 // true, 超过 53 个二进制位的数值，无法保持精度
Math.pow(2, 1024) // Infinity, // 超过 2 的 1024 次方的数值，无法表示
```

`BigInt`（大整数）是第八种数据类型，用来表示整数，没有位数的限制，任何位数的整数都可以精确表示。

```
const a = 2172141653n;
const b = 15346349309n;

a * b // 33334444555566667777n, BigInt 可以保持精度
Number(a) * Number(b) // 33334444555566670000, 普通整数无法保持精度
2n ** 1024n // 179...216n, BigInt 可以表示 2 的 1024 次方的数值
```

为了与 Number 类型区别，BigInt 类型的数据必须添加后缀 `n`。

```
1234 // 普通整数
1234n // BigInt
1n + 2n // 3n, // BigInt 的运算
```

BigInt 同样可以使用各种进制表示，都要加上后缀n。

```
0b1101n // 二进制，输出十进制：13n
0o777n // 八进制，输出十进制：511n
0xFFn // 十六进制，输出十进制：255n
```

BigInt 与普通整数是两种值，它们之间并不相等。

```
42n === 42 // false
```

typeof 运算符对于 BigInt 类型的数据返回 `bigint`。

```
typeof 123n // 'bigint'
```

BigInt 可以使用负号（-），但是不能使用正号（+），因为会与 asm.js 冲突。

```
-42n // 正确
+42n // Cannot convert a BigInt value to a number
```

JavaScript 以前不能计算70的阶乘（即70!），因为超出了可以表示的精度。

```
let p = 1;
for (let i = 1; i <= 70; i++) {
  p *= i;
}
console.log(p); // 1.197857166996989e+100
```

现在支持大整数了，就可以算了，浏览器的开发者工具运行下面代码，就OK。

```
let p = 1n;
for (let i = 1n; i <= 70n; i++) {
  p *= i;
}
console.log(p); // 11978571...00000000n
```

2. BigInt 函数

`BigInt()` 函数，可以用它生成 BigInt 类型的数值。转换规则基本与 `Number()` 一致，将其他类型的值转为 `BigInt`。

```
BigInt(123) // 123n
BigInt('123') // 123n
BigInt(false) // 0n
BigInt(true) // 1n
```

`BigInt()` 函数必须有参数，而且参数必须可以正常转为数值：

```
// 下面的用法都会报错
new BigInt() // TypeError: BigInt is not a constructor
BigInt(undefined) //TypeError: Cannot convert undefined to a BigInt
BigInt(null) // TypeError: Cannot convert null to a BigInt
BigInt('123n') // SyntaxError: Cannot convert 123n to a BigInt
BigInt('abc') // SyntaxError: Cannot convert abc to a BigInt
```

参数如果是小数，也会报错。

```
BigInt(1.5) // RangeError
BigInt('1.5') // SyntaxError
```

BigInt 继承了 Object 对象的两个实例方法。`BigInt.prototype.toString()`，`BigInt.prototype.valueOf()`。

```
// Number.parseInt() 与 BigInt.parseInt() 的对比
Number.parseInt('9007199254740993', 10); // 9007199254740992
BigInt.parseInt('9007199254740993', 10); // 9007199254740993n
```

`Number.parseInt()` 方法返回的结果是不精确的，由于有效数字超出了最大限度。而 `BigInt.parseInt()` 方法正确返回了对应的 `BigInt`。

3. 转换规则

可以使用 `Boolean()`、`Number()` 和 `String()`，将 `BigInt` 可以转为布尔值、数值和字符串类型。

```
Boolean(0n) // false
Boolean(1n) // true
Number(1n)  // 1
String(1n)  // "1"，转为字符串时后缀n会消失。
```

取反运算符（!）也可以将 BigInt 转为布尔值。

```
!0n // true
!1n // false
```

4. 数学运算

数学运算方面，BigInt 类型的 `+`、`-`、`*` 和 `**` 这四个二元运算符，与 Number 类型的行为一致。除法运算 `/` 会舍去小数部分，返回一个整数：

```
9n / 5n // 1n
```

BigInt 不能与普通数值进行混合运算。

```
1n + 1.3 // 报错
```

上面代码报错是因为无论返回的是 BigInt 或 Number，都会导致丢失精度信息。比如 `(2n**53n + 1n) + 0.5` 这个表达式，如果返回 BigInt 类型，`0.5` 这个小数部分会丢失；如果返回 Number 类型，有效精度只能保持 53 位，导致精度下降。

同样的原因，如果一个标准库函数的参数预期是 Number 类型，但是得到的是一个 BigInt，就会报错。

```
Math.sqrt(4n) // Cannot convert a BigInt value to a number
Math.sqrt(Number(4n)) // 2，正确的写法
```

`Math.sqrt` 的参数预期是 Number 类型，如果是 BigInt 就会报错，必须先用 `Number()` 方法转一下类型，才能进行计算。

asm.js 里面，`|0` 跟在一个数值的后面会返回一个 32 位整数。根据不能与 Number 类型混合运算的规则，**BigInt 如果与 `|0` 进行运算会报错。**

```
1n | 0 // 报错
```

5. 其他运算

BigInt 对应的布尔值，与 Number 类型一致，即 `0n` 会转为 false，其他值转为 true。

```
if (0n) {
  console.log('if');
} else {
  console.log('else');
}
// else
```

比较运算符（比如 `>`）和相等运算符（`==`）允许 BigInt 与其他类型的值混合计算，因为这样做不会损失精度：

```
0n < 1 // true
0n < true // true
0n == 0 // true
0n == false // true
0n === 0 // false
```

BigInt 与字符串混合运算时，会先转为字符串，再进行运算。

```
'' + 123n // "123"
```

函数的扩展

1. 函数参数默认值

1.1. 基本用法

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

```
function log(x, y) {
  y = y || 'World';
  console.log(x, y);
}
log('Hello') // Hello World, y 无值是 undefined 会取 'World'
log('Hello', 'China') // Hello China
log('Hello', '') // Hello World, y 有值是空字符串，还是会取 'World'
```

如果参数 `y` 赋值了 `falsely`，则该赋值不起作用，结果被改为默认值。

为了避免这个问题，通常需要先判断一下参数 `y` 是否被赋值，如果没有，再等于默认值。

```
if (typeof y === 'undefined') {
  y = 'World';
}
```

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

```
function log(x, y = 'World') {
  console.log(x, y);
}
log('Hello') // Hello World
log('Hello', 'China') // Hello China
log('Hello', '') // Hello
```

ES6 的写法除了简洁，还有两个好处：

> - 首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；
> - 其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

默认值的参数变量是默认声明的，所以不能用 `let` 或 `const` 再次声明。

```
function foo(x = 5) {
  let x = 1; // Identifier 'x' has already been declared
```

```
  const x = 2; // Identifier 'x' has already been declared
}
```

参数默认值不是传值的，而是每次都重新计算默认值表达式的值。也就是说，参数默认值是惰性求值的。

```
let x = 99;
function foo(p = x + 1) {
  console.log(p);
}
// p = x + 1; x 值改变后，p 会重新求值
foo() // 100，x 是 99 时，p 是 100

x = 100;
foo() // 101，x 是 100 时，p 是 101
```

每次调用函数 `foo()`，都会重新计算 `x + 1`。

1.2. 函数的 length 属性

指定了默认值以后，函数的 `length` 属性，将返回没有指定默认值的参数个数。这是因为 `length` 属性的含义是，该函数预期传入的参数个数。某个参数指定默认值以后，预期传入的参数个数就不包括这个参数了。

```
(function (a) {}).length // 1
(function (a = 5) {}).length // 0
(function (a, b, c = 5) {}).length // 2
```

如果设置了默认值的参数不是尾参数，那么length属性也不再计入后面的参数了。

```
(function (a = 0, b, c) {}).length // 0
(function (a, b = 1, c) {}).length // 1
```

1.3. 作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域（context）。等到初始化结束，这个作用域就会消失。

```
let x = 1;
function f(x, y = x) {
  console.log(y);
}
f(2) // 2
```

参数 y 的默认值等于变量 x。调用函数 f 时，参数形成一个单独的作用域。在这个作用域里面，默认值变量 x 指向第一个参数 x，而不是全局变量 x，所以输出是 2。

```
let x = 1;
function f(y = x) {
  let x = 2; // 此时定义的 x 不影响默认变量 x
  console.log(y);
}
f() // 1
```

函数 f 调用时，参数 y = x 形成一个单独的作用域。这个作用域里面，变量 x 本身没有定义，所以指向外层的全局变量 x。函数调用时，函数体内部的局部变量 x 影响不到默认值变量 x。

如果此时，全局变量 x 不存在，就会报错。

```
function f(y = x) {
  let x = 2;
  console.log(y);
}
f() // ReferenceError: x is not defined
```

2. rest 参数

rest 参数（形式为 ...变量名），用于获取函数的多余参数，这样就不需要使用 arguments 对象了。**rest 参数搭配的变量是一个数组**，该变量将多余的参数放入数组中。

```
function add(...values) {
  let sum = 0;
  for (let val of values) {
    sum += val;
  }
  return sum;
}
add(2, 5, 3) // 10
```

使用 rest 参数代替 arguments 变量的例子。

```
// arguments 变量的写法
function sortNumbers() {
  // arguments 是类似数组的对象，需要先转换为数值才能使用 sort() 方法
  return Array.from(arguments).sort();
}

// rest 参数的写法
const sortNumbers = (...numbers) => numbers.sort();
```

rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。

```
function f(a, ...b, c) {
  // ...
}
// Rest parameter must be last formal parameter
```

函数的 length 属性，不包括 rest 参数。

```
(function(a) {}).length  // 1
(function(...a) {}).length  // 0
(function(a, ...b) {}).length  // 1
```

3. name 属性

函数的 name 属性，返回该函数的函数名，ES6 将其写入了标准。

```
function foo() {}
foo.name // "foo"
```

如果将一个匿名函数赋值给一个变量，ES6 的 name 属性会返回变量名作为实际的函数名。

```
var f = function () {};
f.name // "f"
```

如果将一个具名函数赋值给一个变量，ES6 的name属性都返回这个具名函数原本的名字。

```
const bar = function baz() {};
bar.name // "baz"
```

Function 构造函数返回的函数实例，name属性的值为anonymous。

```
(new Function).name // "anonymous"（匿名的，不知名的）
```

bind 返回的函数，name 属性值会加上 bound 前缀。

```
function foo() {};
foo.bind({}).name // "bound foo"
```

```
(function(){}).bind({}).name // "bound "
```

4. Function.prototype.toString()

ES2019 对函数实例的 `toString()` 方法做出了修改。修改后的 `toString()` 方法，明确要求返回一模一样的原始代码。

```
function /* foo comment */ foo () {}
foo.toString() // "function /* foo comment */ foo () {}"

function a() {/**/}
a.toString() // 'function a() {/**/}'
```

5. catch 命令的参数省略

以前明确要求 `try...catch` 结构中，`catch` 命令后面必须跟参数，接受 `try` 代码块抛出的错误对象。

```
try {
  // ...
} catch (err) {
  // 处理错误
}
```

很多时候，`catch` 代码块可能用不到这个参数。但为了保证语法正确，还是必须写。ES2019 做出了改变，允许 `catch` 语句省略参数。

```
try {
  // ...
} catch {
  // ...
}
```

数组的扩展

1. 扩展运算符

扩展运算符（spread）是三个点（...）。好比 rest 的逆运算，**将一个数组转为用逗号分隔的参数序列。**

```
console.log(...[1, 2, 3]); // 1 2 3
console.log(1, ...[2, 3, 4], 5); // 1 2 3 4 5
[...document.querySelectorAll('div')]; // [<div>, <div>, <div>]
```

该运算符主要用于函数调用。

```
function push(array, ...items) {
  array.push(...items); // 数组 push 方法可以一次添加多个值，用逗号隔开
}
function add(x, y) {
  return x + y;
}
const numbers = [4, 38];
add(...numbers) // 42，相当于：add(4, 38)
```

扩展运算符与正常的函数参数可以结合使用：

```
function f(v, w, x, y, z) { }
const args = [0, 1];
f(-1, ...args, 2, ...[3]); // 相当于：f(-1, 0, 1, 2, 3)
```

如果扩展运算符后面是一个空数组，则不产生任何效果。

```
[...[], 1]; // [1]
```

只有函数调用时，扩展运算符才可以放在圆括号中，否则会报错。

```
(...[1, 2]) // Uncaught SyntaxError: Unexpected token '...'
console.log((...[1, 2])) // Uncaught SyntaxError: Unexpected token '...'
console.log(...[1, 2]) // 1 2
```

1.1. 替代函数的 apply() 方法

由于扩展运算符可以展开数组，所以不再需要 apply() 方法将数组转为函数的参数了。

```
// ES5 的写法
function f(x, y, z) {
  // ...
}
var args = [0, 1, 2];
f.apply(null, args);

// ES6 的写法
function f(x, y, z) {
  // ...
}
let args = [0, 1, 2];
f(...args);
```

应用 `Math.max()` 方法，简化求出一个数组最大元素：

```
Math.max.apply(null, [14, 3, 77]) // ES5 的写法
Math.max(...[14, 3, 77])  // ES6 的写法
// 等同于
Math.max(14, 3, 77); // Math.max() 不能直接求数组的最大值，只能求一组序列数字的最大值
```

将一个数组添加到另一个数组的尾部：

```
// ES5 的写法
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
Array.prototype.push.apply(arr1, arr2);

// ES6 的写法
let arr1 = [0, 1, 2];
let arr2 = [3, 4, 5];
arr1.push(...arr2);
```

1.2. 扩展运算符的应用

（1）复制数组

数组是复合的数据类型，直接复制的话，只是复制了指向底层数据结构的指针，而不是克隆一个全新的数组。

```
const a1 = [1, 2];
const a2 = a1;

a2[0] = 2;
a1 // [2, 2]
```

a2 和 a1 指向同一份数据的另一个指针。修改 a2，会直接导致 a1 的变化。

ES5 只能用变通方法来复制数组。

```
const a1 = [1, 2];
const a2 = a1.concat(); // concat 返回一个新数组，a2 是一个新数组，和 a1 无关

a2[0] = 2;
a1 // [1, 2]
```

a1 会返回原数组的克隆，再修改 a2 就不会对 a1 产生影响。

扩展运算符提供了复制数组的简便写法。

```
const a1 = [1, 2];
// 下面的两种写法，`a2` 都是 `a1` 的克隆。
const a2 = [...a1]; // 写法一
const [...a2] = a1; // 写法二
```

（2）合并数组

扩展运算符提供了数组合并的新写法。

```
const arr1 = ['a', 'b'];
const arr2 = ['c'];
const arr3 = ['d', 'e'];

// ES5 的合并数组
arr1.concat(arr2, arr3); // [ 'a', 'b', 'c', 'd', 'e' ]

// ES6 的合并数组
[...arr1, ...arr2, ...arr3]; // [ 'a', 'b', 'c', 'd', 'e' ]
```

（3）与解构赋值结合

扩展运算符可以与解构赋值结合起来，用于生成数组。

```
a = list[0], rest = list.slice(1) // ES5

[a, ...rest] = list  // ES6
```

（4）字符串

扩展运算符还可以将字符串转为真正的数组。

```
[...'hello'] // [ "h", "e", "l", "l", "o" ]
```

（5）实现了 Iterator 接口的对象

任何定义了遍历器（Iterator）接口的对象，都可以用扩展运算符转为真正的数组。

```
let nodeList = document.querySelectorAll('div');
let array = [...nodeList];
```

`querySelectorAll()` 方法返回的是一个 `NodeList` 对象，是一个类似数组的对象。扩展运算符可以将其转为真正的数组，原因就在于NodeList对象实现了 Iterator。

对于那些没有部署 Iterator 接口的类似数组的对象，扩展运算符就无法将其转为真正的数组。

```
let arrayLike = {
  '0': 'a',
  '1': 'b',
  '2': 'c',
  length: 3
};
let arr = [...arrayLike]; // arrayLike is not iterable
```

arrayLike 是一个类似数组的对象，但是没有部署 Iterator 接口，扩展运算符就会报错。这时，可以改为使用 `Array.from` 方法将 `arrayLike` 转为真正的数组。

（6）Map 和 Set 结构

扩展运算符内部调用的是数据结构的 Iterator 接口，因此只要具有 Iterator 接口的对象，都可以使用扩展运算符，比如 Map 结构。

let map = new Map([[1, 'one'], [2, 'two'], [3, 'three'],]);

如果对没有 Iterator 接口的对象，使用扩展运算符，将会报错。

```
const obj = {a: 1, b: 2};
let arr = [...obj]; // TypeError: Cannot spread non-iterable
```

Array.from()

`Array.from()` 方法用于将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括 ES6 新增的数据结构 Set 和 Map）。

下面是一个类似数组的对象，Array.from()将它转为真正的数组。

```
let arrayLike = {
    '0': 'a',
    '1': 'b',
    '2': 'c',
    length: 3
};

// ES5 的写法
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']
var arr1 = Array.prototype.slice.call(arrayLike); // ['a', 'b', 'c']
Array.prototype === [].__proto__ // true, 类的显示原型指向实例的隐士原型

// ES6 的写法
let arr2 = Array.from(arrayLike); // ['a', 'b', 'c']
```

字符串、NodeList、arguments、定义了 `length` 的对象，Array.from()都可以将它们转为真正的数组。

只要是部署了 Iterator 接口的数据结构，`Array.from()` 都能将其转为数组。

```
Array.from('hello') // ['h', 'e', 'l', 'l', 'o']

let namesSet = new Set(['a', 'b'])
Array.from(namesSet) // ['a', 'b']
```

字符串和 Set 结构都具有 Iterator 接口，因此可以被Array.from()转为真正的数组。

如果参数是一个真正的数组，Array.from()会返回一个一模一样的新数组。

```
Array.from([1, 2, 3]) // [1, 2, 3]
```

扩展运算符背后调用的是遍历器接口（Symbol.iterator），如果一个对象没有部署这个接口，就无法转换。Array.from() 方法还支持类似数组的对象。

运算符的扩展

1. 指数运算符

指数运算符 **，这个运算符的一个特点是右结合，而不是常见的左结合。多个指数运算符连用时，是从最右边开始计算的。

```
2 ** 3; // 8
2 ** 3 ** 2; // <=> 2 ** (3 ** 2), <=> 2 ** 9 // 512
```

指数运算符可以与等号结合，形成一个新的赋值运算符（**=）。

```
let a = 1.5;
a **= 2; // 等同于 a = a * a;

let b = 4;
b **= 3; // 等同于 b = b ** 3;
```

2. 链判断运算符

以往如果读取对象内部的某个属性，往往需要判断一下，属性的上层对象是否存在。比如，读取 message.body.user.firstName 这个属性，安全的写法是写成下面这样。

```
// 错误的写法
const  firstName = message.body.user.firstName || 'default';

// 正确的写法
const firstName = (message && message.body && message.body.user &&
message.body.user.firstName) || 'default';
```

firstName 属性在对象的第四层，所以需要判断四次，每一层是否有值。

三元运算符 ?: 也常用于判断对象是否存在。

```
const fooInput = myForm.querySelector('input[name=foo]')
const fooValue = fooInput ? fooInput.value : undefined
```

这样的层层判断非常麻烦，“链判断运算符”（optional chaining operator）?. 可以简化上面的写法：

```
const firstName = message?.body?.user?.firstName || 'default';
const fooValue = myForm.querySelector('input[name=foo]')?.value
```

使用链判断运算符 `?.` 运算符判断左侧的对象是否为 `null` 或 `undefined`。如果是的，就不再往下运算，而是返回 `undefined`。

判断对象方法是否存在，如果存在就立即执行：

```
iterator.return?.()
```

`iterator.return` 如果有定义，就会调用该方法，否则 `iterator.return` 直接返回 `undefined`，不再执行 `?.` 后面的部分。

对于那些可能没有实现的方法，这个运算符尤其有用：

```
if (myForm.checkValidity?.() === false) { // 表单校验失败
  return;
}
```

老式浏览器的表单对象可能没有 `checkValidity()` 这个方法，这时 `?.` 运算符就会返回 `undefined`，判断语句就变成了 `undefined === false`，所以就会跳过下面的代码。

3. `Null` 判断运算符

以往读取对象属性的时候，如果某个属性的值是 `null` 或 `undefined`，有时候需要通过 `||` 运算符指定默认值。

```
const headerText = response.settings.headerText || 'Hello, world!';
const animationDuration = response.settings.animationDuration || 300;
const showSplashScreen = response.settings.showSplashScreen || true;
```

开发者的原意是，只要属性的值为 `null` 或 `undefined`，默认值就会生效，但是属性的值如果为空字符串或 `false` 或 `0`，默认值也会生效。

`Null` 判断运算符 `??`，它的行为类似 `||`，但是只有运算符左侧的值为 `null` 或 `undefined` 时，才会返回右侧的值。

```
const headerText = response.settings.headerText ?? 'Hello, world!';
const animationDuration = response.settings.animationDuration ?? 300;
const showSplashScreen = response.settings.showSplashScreen ?? true;
```

这个运算符的一个目的，就是跟链判断运算符 `?.` 配合使用，为 `null` 或 `undefined` 的值设置默认值。

```
const animationDuration = response.settings?.animationDuration ?? 300;
```

如果 response.settings 是 null 或 undefined，或者 response.settings.animationDuration 是 null 或 undefined，就会返回默认值 300。也就是说，这一行代码包括了两级属性的判断。

?? 本质上是逻辑运算，它与其他两个逻辑运算符 && 和 || 有一个优先级问题，它们之间的优先级到底孰高孰低。优先级的不同，往往会导致逻辑运算的结果不同。

现在的规则是，如果多个逻辑运算符一起使用，必须用括号表明优先级，否则会报错。

```
// 下面四个表达式都会报错
lhs && middle ?? rhs
lhs ?? middle && rhs
lhs || middle ?? rhs
lhs ?? middle || rhs
```

必须加入表明优先级的括号，才不会报错：

```
(lhs && middle) ?? rhs; // 或 lhs && (middle ?? rhs);

(lhs ?? middle) && rhs; // 或 lhs ?? (middle && rhs);

(lhs || middle) ?? rhs; // 或 lhs || (middle ?? rhs);

(lhs ?? middle) || rhs; // 或 lhs ?? (middle || rhs);
```

4. 逻辑赋值运算符

逻辑赋值运算符（logical assignment operators），将逻辑运算符与赋值运算符进行结合。

```
// 或赋值运算符
x ||= y; // 等同于 x || (x = y)

// 与赋值运算符
x &&= y; // 等同于 x && (x = y)

// Null 赋值运算符
x ??= y; // 等同于 x ?? (x = y)
```

这三个运算符 ||=、&&=、??= 相当于先进行逻辑运算，然后根据运算结果，再视情况进行赋值运算。

它们的一个用途是，为变量或属性设置默认值。

```
user.id = user.id || 1; // 老的写法

user.id ||= 1; // 新的写法
```

`user.id` 属性如果不存在，则设为1，新的写法比老的写法更紧凑一些。

参数对象opts如果不存在属性foo和属性baz，则为这两个属性设置默认值。

```
function example(opts) {
  opts.foo = opts.foo ?? 'bar';
  opts.baz ?? (opts.baz = 'qux');
}
```

有了“`Null` 赋值运算符”以后，就可以统一写成下面这样：

```
function example(opts) {
  opts.foo ??= 'bar';
  opts.baz ??= 'qux';
}
```

5. #! 命令

JavaScript 脚本引入了 `#!` 命令，写在脚本文件或者模块文件的第一行。对于 JavaScript 引擎来说，会把 `#!` 理解成注释，忽略掉这一行。

```
// index.js
#! /usr/bin/env node // 写在脚本文件第一行
console.log(1);

#!/usr/bin/env node // 写在模块文件第一行
export {};
console.log(1);
```

有了这一行以后，Unix 命令行就可以直接执行脚本。

```
# 以前执行脚本的方式
$ node index.js

# hashbang 的方式
$ ./index.js
```

对象的扩展

1. 属性的简介表示法

如果属性值是一个变量，且变量名和属性名一致，则对象键值对可以缩写成属性名：

```
const foo = 'bar';
const baz = {foo};
// 等同于
const baz = {foo: foo};
baz // {foo: "bar"}
```

除了属性简写，方法也可以简写：

```
const o = {
  method() { return "Hello!"; }
};
// 等同于
const o = {
  method: function() { return "Hello!"; }
};
```

简写的对象方法不能用作构造函数，会报错。

```
const obj = {
  f() {
    this.foo = 'bar';
  }
};
new obj.f() // obj.f is not a constructor

const obj2 = {
  f: function() {
    this.foo = 'bar';
  }
};
new obj2.f() // f {foo: 'bar'}
```

2. 属性名表达式

ES6 允许字面量定义对象时，用表达式作为对象的属性名，即把表达式放在方括号内。

```
let lastWord = 'last word';

const a = {
  'first word': 'hello',
  [lastWord]: 'world' // 用表达式作为对象属性名
};

a['first word'] // "hello"
a[lastWord] // "world"
a['last word'] // "world"
```

属性名表达式与简洁表示法，不能同时使用，会报错。

```
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] }; // Unexpected token '['

const foo = 'bar';
const baz = { [foo]: 'abc'};
baz // {bar: 'abc'}
```

3. 方法的属性名

函数的 name 属性，返回函数名。对象方法也是函数，因此也有 name 属性。

```
const person = {
  sayName() {
    console.log('hello!');
  },
};
person.sayName.name   // "sayName"
```

如果对象的方法使用了取值函数（getter）和存值函数（setter），则 `name` 属性不是在该方法上面，而是该方法的属性的描述对象的 `get` 和 `set` 属性上面，返回值是方法名前加上 `get` 和 `set`。

```
const obj = {
  get foo() {},
  set foo(x) {}
};
obj.foo.name; // TypeError: Cannot read properties of undefined (reading 'name')

const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');
descriptor.get.name // "get foo"
descriptor.set.name // "set foo"
```

4. super 关键字

this 关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字 super，指向当前对象的原型对象：

```
const proto = {
  foo: 'hello'
};

const obj = {
  foo: 'world',
  find() {
    return super.foo;
  }
};

Object.setPrototypeOf(obj, proto); // 设置 obj 的原型是 proto
obj.find() // "hello", 相当于 proto.foo
```

super.foo 等同于 Object.getPrototypeOf(this).foo。

super 关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。

```
 // 下面三种情况均报错: 'super' keyword unexpected here
const obj = { foo: super.foo };

const obj = { foo: () => super.foo };

const obj = {
  foo: function () {
    return super.foo
  }
}
```

5. 对象的扩展运算符

5.1. 解构赋值

对象的解构赋值用于从一个对象取值，相当于将目标对象自身的所有可遍历的（enumerable）、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。

```
let { x, ...z } = { x: 1, a: 3, b: 4 };
x // 1
z // { a: 3, b: 4 }
```

由于解构赋值要求等号右边是一个对象，所以如果等号右边是 `undefined` 或 `null`，就会报错，因为它们无法转为对象。

```
let { ...z } = null; // 运行时错误
let { ...z } = undefined; // 运行时错误
```

解构赋值必须是最后一个参数，否则会报错。

```
let { ...x, y } = { a: 1, b: 2, y: 3 }; // Rest element must be last element
let { x, ...y, ...z } = { x: 1, a: 1, b: 2 };; // Rest element must be last
element
```

解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本。

```
let obj = { a: { b: 1 } };
let { ...x } = obj;
obj.a.b = 2;
x.a.b // 2
```

5.2. 扩展运算符

对象的扩展运算符（`...`）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

```
let z = { a: 3, b: 4 };
let n = { ...z };
n // { a: 3, b: 4 }
```

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

```
let foo = { ...['a', 'b', 'c'] };
foo // {0: "a", 1: "b", 2: "c"}
```

```
// 等同于 {...Object(true)}
{...true} // {}

// 等同于 {...Object(undefined)}
{...undefined} // {}

// 等同于 {...Object(null)}
{...null} // {}
```

如果扩展运算符后面是字符串，它会自动转成一个类似数组的对象，因此返回的不是空对象。

```
{...'hello'} // {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}
```

对象的扩展运算符，只会返回参数对象自身的、可枚举的属性，尤其是用于类的实例对象时。

```
class C {
  p = 12;
  m() {}
}

let c = new C();
let clone = { ...c };

clone.p; // 12
clone.m(); // clone.m is not a function
```

不会返回 `c` 的方法 `c.m()`，因为这个方法定义在 `C` 的原型对象上。

对象的扩展运算符等同于使用 `Object.assign()` 方法。

```
let aClone = { ...a };
// 等同于
let aClone = Object.assign({}, a);
```

上面的例子只是拷贝了对象实例的属性。

扩展运算符可以用于合并两个对象。

```
let ab = { ...a, ...b };
// 等同于
let ab = Object.assign({}, a, b);
```

如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

```
let aWithOverrides = { ...a, x: 1, y: 2 };
// 等同于
let aWithOverrides = { ...a, ...{ x: 1, y: 2 } };
// 等同于
let x = 1, y = 2, aWithOverrides = { ...a, x, y };
// 等同于
let aWithOverrides = Object.assign({}, a, { x: 1, y: 2 });
```

这用来修改现有对象部分的属性就很方便了。

```
let newVersion = {
  ...previousVersion,
  name: 'New Name' // Override the name property
};
```

对象新增方法

1. Object.is()

`Object.is()` 用来比较两个值是否严格相等，与严格比较运算符（`===`）的行为基本一致，只有两点不同：

> - `+0` 不等于 `-0`
> - `NaN` 等于自身

```
+0 === -0 //true
NaN === NaN // false

Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

2. Object.assign()

`Object.assign()` 方法用于对象的合并，第一个参数是目标对象，后面的参数都是源对象。将源对象（`source`）的所有可枚举属性，复制到目标对象（`target`）：

```
const target = { a: 1 };

const source1 = { b: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

```
const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

如果只有一个参数，Object.assign() 会直接返回该参数。

```
const obj = {a: 1};
Object.assign(obj) === obj // true
```

`Object.assign()` 拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（`enumerable: false`）。

```
Object.assign({b: 'c'},
  Object.defineProperty({}, 'invisible', {
    enumerable: false,
    value: 'hello'
  })
)
// { b: 'c' }
```

上面代码中，`Object.assign()` 要拷贝的对象只有一个不可枚举属性invisible，这个属性并没有被拷贝进去。

属性名为 Symbol 值的属性，也会被Object.assign()拷贝。

```
Object.assign({ a: 'b' }, { [Symbol('c')]: 'd' })
// { a: 'b', Symbol(c): 'd' }
```

Symbol

1. 概述

ES6 引入 Symbol 的原因是为对象添加独一无二的属性或方法。Symbol 是一种原始数据类型，它属于 JavaScript 语言的原生数据类型之一。

Symbol 值通过 `Symbol()` 函数生成。对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

```
let s = Symbol();
typeof s; // "symbol"
```

`Symbol()` 函数前不能使用 `new` 命令，这是因为生成的 Symbol 是一个原始类型的值，不是对象，所以不能使用 `new` 命令来调用。 另外，由于 Symbol 值不是对象，所以也不能添加属性。基本上，它是一种类似于字符串的数据类型。

```
new Symbol(); // TypeError: Symbol is not a constructor
```

`Symbol()` 函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述。 这主要是为了在控制台显示，或者转为字符串时，比较容易区分。

```
let s1 = Symbol('foo');
let s2 = Symbol('bar');

s1 // Symbol(foo)
s2 // Symbol(bar)

s1.toString() // "Symbol(foo)"
s2.toString() // "Symbol(bar)"
```

上例中，s1 和 s2 是两个 Symbol 值。如果不加参数，它们在控制台的输出都是 `Symbol()`，不利于区分。有了参数以后，就等于为它们加上了描述，输出的时候就能够分清，到底是哪一个值。

如果 Symbol 的参数是一个对象，就会调用该对象的 `toString()` 方法，将其转为字符串，然后才生成一个 Symbol 值。

```
const obj = {
  toString() {
    return 'abc';
  }
};
```

```
const sym = Symbol(obj);
sym // Symbol(abc)
```

`Symbol()` 函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的 Symbol 函数的返回值是不相等的。

```
// 没有参数的情况
let s1 = Symbol();
let s2 = Symbol();
s1 === s2; // false

// 有参数的情况
let s1 = Symbol('foo');
let s2 = Symbol('foo');
s1 === s2; // false
```

Symbol 值不能与其他类型的值进行运算，会报错。

```
let sym = Symbol('My symbol');
"your symbol is " + sym; // TypeError: can't convert symbol to string
`your symbol is ${sym}`; // TypeError: can't convert symbol to string
```

但是，Symbol 值可以显式转为字符串。

```
let sym = Symbol('My symbol');
String(sym); // 'Symbol(My symbol)'
sym.toString(); // 'Symbol(My symbol)'
```

Symbol 值也可以转为布尔值，但是不能转为数值。

```
let sym = Symbol();
Boolean(sym) // true
!sym  // false
!!sym  // true
if (sym) {
  console.log("a"); // a
}

Number(sym) // TypeError: Cannot convert a Symbol value to a number
sym + 2 // TypeError: Cannot convert a Symbol value to a number
```

2. Symbol.prototype.description

Symbol 值的实例属性 `description` 返回 Symbol 值的描述。

```
const sym = Symbol('foo');
sym.description; // "foo"
```

3. Symbol 作为属性名

由于每一个 Symbol 值都是不相等的，这意味着只要 Symbol 值作为标识符，用于对象的属性名，就能保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止某一个键被不小心改写或覆盖。

```
let s = Symbol();

// 第一种写法
let a = {};
a[s] = 'Hello!';

// 第二种写法
let a = {
  [s]: 'Hello!'
};

// 第三种写法
let a = {};
Object.defineProperty(a, s, { value: 'Hello!' });

// 以上写法都得到同样结果
a[s] // "Hello!"
```

Symbol 值作为对象属性名时，不能用点运算符。

```
const mySymbol = Symbol();
const a = {};

a.mySymbol = 'Hello!'; // 此处 mySymbol 是字符串
a[mySymbol] // undefined, 此处 mySymbol 是 Symbol 类型
a['mySymbol'] // "Hello!", 获取字符串 mySymbol 属性
```

4. 属性名的遍历

Symbol 值作为属性名，遍历对象的时候，该属性不会出现在 `for...in`、`for...of` 循环中，也不会被 `Object.keys()`、`Object.getOwnPropertyNames()`、`JSON.stringify()` 返回。

但是，它也不是私有属性，有一个 `Object.getOwnPropertySymbols()` 方法，可以获取指定对象的所有 Symbol 属性名。该方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。

```
const obj = {};
const foo = Symbol('foo');
obj[foo] = 'bar';

for (let i in obj) {
  console.log(i); // 无输出
}
Object.keys(obj); // []
JSON.stringify(obj); // {}
Object.getOwnPropertyNames(obj) // []
Object.getOwnPropertySymbols(obj) // [Symbol(foo)]
```

上例中，需要使用 `Object.getOwnPropertySymbols()` 方法才能得到对象的所有 Symbol 属性名。

另一个新的 API，`Reflect.ownKeys()` **方法可以返回所有类型的键名，包括常规键名和 Symbol 键名。**

```
let obj = {
  [Symbol('my_key')]: 1,
  enum: 2,
  nonEnum: 3
};

Reflect.ownKeys(obj); //  ["enum", "nonEnum", Symbol(my_key)]
```

5. Symbol.for()，Symbol.keyFor()

如果希望重新使用同一个 Symbol 值，`Symbol.for()` 方法可以做到这一点。它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建一个以该字符串为名称的 Symbol 值，并将其注册到全局。

```
let s1 = Symbol.for('foo');
let s2 = Symbol.for('foo');

typeof s1; // 'symbol'
typeof s2; // 'symbol'

s1; // Symbol(foo)
s2; // Symbol(foo)

s1 === s2 // true
```

Symbol.for()与Symbol()这两种写法，都会生成新的 Symbol。它们的区别是，前者会被登记在全局环境中供搜索，后者不会。`Symbol.for()` 不会每次调用就返回一个新的 Symbol 类型的值，而是会先检查给定的`key` 是否已经存在，如果不存在才会新建一个值。比如，如果调用 `Symbol.for("cat")` 30 次，每次都会返回同一个 Symbol 值，但是调用 `Symbol("cat")` 30 次，会返回 30 个不同的 Symbol 值。

```
Symbol.for("bar") === Symbol.for("bar"); // true
Symbol("bar") === Symbol("bar"); // false
```

上面代码中，由于Symbol()写法没有登记机制，所以每次调用都会返回一个不同的值。

`Symbol.keyFor()` 方法返回一个已登记的 Symbol 类型值的 key。

```
let s1 = Symbol.for("foo");
Symbol.keyFor(s1); // "foo"

let s2 = Symbol("foo");
Symbol.keyFor(s2); // undefined，变量 s2 属于未登记的 Symbol 值，所以返回 undefined。
s2.description; // "foo"
```

注意，Symbol.for()为 Symbol 值登记的名字，是全局环境的，不管有没有在全局环境运行。

```
function foo() {
  return Symbol.for('bar');
}

const x = foo();
const y = Symbol.for('bar');
console.log(x === y); // true
```

`Symbol.for()` 的这个全局登记特性，可以用在不同的 iframe 或 service worker 中取到同一个值。

```
iframe = document.createElement('iframe');
iframe.src = String(window.location);
document.body.appendChild(iframe);

iframe.contentWindow.Symbol.for('foo') === Symbol.for('foo'); // true
```

Set 和 Map 数据结构

1. Set

1.1. 基本用法

`Set` 本身是一个构造函数，用来生成 Set 数据结构。它类似于数组，但是成员的值都是唯一的，没有重复的值。

```
const s = new Set([2, 3, 5, 4, 5, 2, 2]);
s; // Set(4) {2, 3, 5, 4}
```

Set 原始参数是 `[2, 3, 5, 4, 5, 2, 2]`，最终 s 只有 `2 3 5 4` 四个没有重复的值，后面的 `5 2 2` 因为与前面的成员重复，没有被加入到 s。证明 **Set 数据结构成员的值都是唯一的，没有重复。**

`Set` 函数可以接受一个数组（或者具有 iterable 接口的其他数据结构，如字符串、NodeList）作为参数，用来初始化。

```
// 例一，数组作为 Set 的参数
const set1 = new Set([1, 2, 3, 3]);
set1; // Set(3) {1, 2, 3}

// 例二，字符串作为 Set 的参数
const set2 = new Set("abcbcdcde");
set2; // Set(5) {'a', 'b', 'c', 'd', 'e'}

// 例三
const set3 = new Set(document.querySelectorAll('div'));
set3.size // 31，本页面 div 标签的个数
```

根据 Set 数据结构的成员是唯一的，没有重复这一特点，可以用于数组去除重复成员：

```
[...new Set(array)];
// 或者
Array.from(new Set(array));
```

也可以用于去除字符串里面的重复字符：

```
// Set 接受字符串作为参数，将其转换为数组，再转换为字符串，达到“去除字符串重复字符”的目的
[...new Set('abcbcdcde')].join(''); // 'abcde'
```

在 Set 内部，两个 `NaN` 是相等的。

```
let set = new Set();
set.add(NaN);
set.add(NaN);
set; // Set {NaN}
// 添加了两个 NaN，但是只保留了一个，由于 Set 数据结构中值是唯一的，证明 Set 内部将两个 NaN 认为是重复的
```

在 Set 内部，两个对象总是不相等的。

```
let set = new Set();
set.add({});
set.add({});
set.size; // 2
// 添加了两个空对象，成员是两个，证明将两个空对象被视为两个值。
```

1.2. Set 实例的属性

1.2.1. Set.prototype.constructor

`Set.prototype.constructor` 构造函数，默认就是 Set 函数。

```
Set.prototype.constructor === Set; // true
```

1.2.2. Set.prototype.size

`Set.prototype.size` 返回 Set 实例的成员总数。

```
let s = new Set([1, 2, 3]);
s.size; // 3
```

1.3. Set 的操作方法

- Set.prototype.add(value)：添加某个值，返回 Set 结构本身。
- Set.prototype.delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
- Set.prototype.has(value)：返回一个布尔值，表示该值是否为Set的成员。
- Set.prototype.clear()：清除所有成员，没有返回值。

```
let s = new Set();
s.add(1).add(2).add(2); // add 方法返回 Set 结构本身，所以可以链式操作
s.size // 2

s.has(1) // true
```

```
s.has(2) // true
s.has(3) // false

s.delete(2) // true
s.has(2) // false

s.delete(2) // false，删除失败，因为前面已经删除过该值了，s 中已经不存在该成员了

s.clear();
s.size; // 0
```

1.4. Set 的遍历方法

Set的遍历顺序就是插入顺序，使用 Set 保存一个回调函数列表，调用时就能保证按照添加顺序调用。

1.4.1. keys()，values()，entries()

`keys()`、`values()`、`entries()` 返回的都是遍历器对象。由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以 `keys()` 方法和 `values()` 方法的行为完全一致。

```
let set = new Set(['red', 'green', 'blue']);

// keys() 和 values() 都返回 SetIterator {'red', 'green', 'blue'}
for (let item of set.keys()) { console.log(item); }
for (let item of set.values()) { console.log(item); }
// red
// green
// blue

for (let item of set.entries()) { console.log(item); }
// ["red", "red"]
// ["green", "green"]
// ["blue", "blue"]
```

上例中，`entries()` 返回的遍历器，同时包括键名和键值，每次输出一个数组，它的两个成员完全相等。

Set 结构的实例默认可遍历，它的默认遍历器生成函数就是它的 `values()` 方法。

```
Set.prototype[Symbol.iterator] === Set.prototype.values; // true
Set[Symbol.iterator] === Set.values; // true
```

这意味着，**可以省略 `values()` 方法，直接用 `for...of` 循环遍历 Set。**

```
let set = new Set(['red', 'green', 'blue']);
for (let x of set) { console.log(x); }
// red
```

```
// green
// blue
```

1.4.2. forEach()

Set 结构的实例与数组一样，也拥有 `forEach` 方法，用于对每个成员执行某种操作，没有返回值。

```
let set = new Set([1, 4, 9]);
set.forEach((value, key) => console.log(key + ' : ' + value))
// 1 : 1
// 4 : 4
// 9 : 9
```

上例 `forEach` 方法的参数就是一个处理函数，该函数的参数与数组的 `forEach` 一致，依次为键值、键名、集合本身（上例省略了该参数）。**Set 结构的键名就是键值（两者是同一个值），因此第一个参数与第二个参数的值永远都是一样的。**

1.4.3. 遍历的应用

扩展运算符（`...`）内部使用 `for...of` 循环，所以也可以用于 Set 结构。

```
let set = new Set(['red', 'green', 'blue']);
[...set]; // ['red', 'green', 'blue']
```

数组的 `map` 和 `filter` 方法也可以间接用于 Set 了。

```
let s1 = new Set([1, 2, 3]);
s1 = new Set([...s1].map(x => x * 2));
// s1: Set(3) {2, 4, 6}

let s2 = new Set([1, 2, 3, 4, 5]);
s2 = new Set([...s2].filter(x => (x % 2) == 0));
// s2: Set(2) {2, 4}
```

使用 Set 可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。

```
// 先构造需要求并集/交集/差集的 Set 数据结构 a 和 b
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);

// 并集。将 a 和 b 转换成数组并合并起来
let union = new Set([...a, ...b]);
// union: Set {1, 2, 3, 4}
```

```
// 交集。将 a 转换成数组，并求 b 中“包含 a 的成员”的成员
let intersect1 = new Set([...a].filter(x => b.has(x)));
// intersect1: set {2, 3}
// 交集。将 b 转换成数组，并求 a 中“包含 b 的成员”的成员
let intersect2 = new Set([...b].filter(x => a.has(x)));
// intersect2: set {3, 2}

//  (a 相对于 b 的) 差集。将 a 转换成数组，并求 b 中“不包含 a 的成员”的成员
let difference1 = new Set([...a].filter(x => !b.has(x)));
// difference1: Set {1}
//  (b 相对于 a 的) 差集。将 b 转换成数组，并求 a 中“不包含 b 的成员”的成员
let difference2 = new Set([...b].filter(x => !a.has(x)));
// difference2: Set {4}
```

2. Map

2.1. 含义和基本用法

JavaScript 的对象（Object）只能用字符串当作键。Map 数据结构类似于对象，但是各种类型的值（包括对象）都可以当作键，是一种更完善的 Hash 结构实现。

```
const m = new Map();
const o = {p: 'Hello World'};

m.set(o, 'content'); // 给 m 加成员，该成员键是对象
m.get(o); // "content" ，读取成员

m.has(o); // true，判断是否有该成员
m.delete(o); // true，删除成员
m.has(o); // false
```

Map 作为构造函数，参数是一个二维数组，该数组每一个成员数组（又包含两个成员）是表示键值对的数组。

```
const map = new Map([['name', '张三'], ['title', 'Author']]);
map.size; // 2
map.has('name'); // true
map.get('name'); // "张三"
map.has('title'); // true
map.get('title'); // "Author"
```

`Map` 构造函数接受数组作为参数，实际上执行的是下面的算法：

```
const items = [['name', '张三'], ['title', 'Author']];
const map = new Map();
items.forEach(([key, value]) => map.set(key, value));
```

事实上，不仅仅是数组，任何具有 Iterator 接口、且每个成员都是一个双元素的数组的数据结构都可以当作 Map 构造函数的参数。这就是说，Set 和 Map 都可以用来生成新的 Map。

```
const set = new Set([['foo', 1], ['bar', 2]]);
const m1 = new Map(set);
m1.get('foo') // 1

const m2 = new Map([['baz', 3]]);
const m3 = new Map(m2);
m3.get('baz') // 3
```

如果对同一个键多次赋值，后面的值将覆盖前面的值。

```
const map = new Map();
map.set(1, 'aaa').set(1, 'bbb'); // 'bbb' 将覆盖前面的
map.get(1) // "bbb"
```

如果读取一个未知的键，则返回 `undefined`。

```
new Map().get('asfddfsasadf'); // undefined
```

只有对同一个对象的引用，Map 结构才将其视为同一个键。

```
const map = new Map();
map.set(['a'], 555);
map.get(['a']) // undefined
```

上例的 set 和 get 方法，表面是针对同一个键，但实际上这是两个不同的数组实例，内存地址是不一样的，因此 get 方法无法读取该键，返回 undefined。

同样的值的两个实例，在 Map 结构中被视为两个键。

```
const map = new Map();
const k1 = ['a'];
const k2 = ['a'];
map.set(k1, 111).set(k2, 222);
map.get(k1) // 111
map.get(k2) // 222
```

Map 的键实际上是跟内存地址绑定的，只要内存地址不一样，就视为两个键。这就解决了同名属性碰撞（clash）的问题，我们扩展别人的库的时候，如果使用对象作为键名，就不用担心自己的属性与原作者的属性同名。

如果 Map 的键是一个简单类型的值（数字、字符串、布尔值），则只要两个值严格相等，Map 将其视为一个键：

```
let map = new Map();

// 0 和 -0 是一个键
map.set(-0, 123);
map.get(+0); // 123

// 布尔值 true 和字符串 'true' 是两个不同的键
map.set(true, 1);
map.set('true', 2);
map.get(true); // 1

// undefined 和 null 是两个不同的键
map.set(undefined, 3);
map.set(null, 4);
map.get(undefined); // 3

// NaN 是一个键
map.set(NaN, 123);
map.get(NaN); // 123
```

2.2. Map 实例的属性

2.2.1. Map.prototype.constructor

`Map.prototype.constructor` 构造函数，默认就是 Map 函数。

```
Map.prototype.constructor === Map; // true
```

2.2.2. Map.prototype.size

`Map.prototype.size` 返回 Map 实例的成员总数。

```
let m = new Map(["foo", true], ["bar", false]);
m.size; // 2
```

2.3. Map 的操作方法

2.3.1. Map.prototype.set()

`set` 方法设置键名 `key` 对应的键值为 `value`，然后返回整个 Map 结构。如果 `key` 已经有值，则键值会被更新，否则就新生成该键。

```
const m = new Map();
// set() 方法返回的是当前的 Map 对象，因此可以采用链式写法
m.set('edition', 6).set(262, 'standard').set(undefined, 'nah');
```

2.3.2. Map.prototype.get()

`get()` 方法读取 `key` 对应的键值，如果找不到 `key`，返回 `undefined`。

```
const m = new Map();
const hello = function() {console.log('hello');};

m.set(hello, 'Hello ES6!'); // 键是函数
m.get(hello);  // Hello ES6!

m.set(hello, 'Hello ES2015!'); // 更新 'hello' 键
m.get(hello);  // Hello ES2015!

m.get('world'); // undefined
```

2.3.3. Map.prototype.has()

`has()` 方法返回一个布尔值，表示某个键是否在当前 Map 对象之中。

```
const m = new Map();
m.set('edition', 6).set(262, 'standard').set(undefined, 'nah');

m.has('edition'); // true
m.has(262);       // true
m.has(undefined); // true
m.has('years');   // false
```

2.3.4. Map.prototype.delete()

`delete()` 方法删除某个键，返回 `true`。如果删除失败，返回 `false`。

```
const m = new Map();
m.set(undefined, 'nah'); // 新增一个 undefined 键
m.has(undefined);        // true

m.delete(undefined);     // true，删除成功
m.has(undefined);        // false

m.delete(undefined);     // false，删除失败，因为上面已经删除过了，该键不存在了
```

2.3.5. Map.prototype.clear()

clear()方法清除所有成员，没有返回值。

```
let map = new Map();
map.set('foo', true).set('bar', false);
map.size // 2

map.clear()
map.size // 0
```

2.4. Map 的遍历方法

Map 的遍历顺序就是插入顺序。

2.4.1. keys()，values()，entries()

```
const map = new Map([['F', 'no'], ['T',  'yes']]);

for (let key of map.keys()) {
  console.log(key);
}
// "F"
// "T"

for (let value of map.values()) {
  console.log(value);
}
// "no"
// "yes"

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
// "F" "no"
// "T" "yes"

// 或者
for (let [key, value] of map.entries()) {
  console.log(key, value);
}
// "F" "no"
// "T" "yes"
```

Map 结构的默认遍历器接口（Symbol.iterator 属性），就是 `entries()` 方法。

```
Map.prototype[Symbol.iterator] === Map.prototype.entries; // true
Map[Symbol.iterator] === Map.entries; // true
```

这意味着，**可以省略 `entries()` 方法，直接用 `for...of` 循环遍历 Map。**

```
const map = new Map([['F', 'no'], ['T',  'yes']]);
for (let [key, value] of map) {
  console.log(key, value);
}
// "F" "no"
// "T" "yes"
```

2.4.2. forEach()

Map 还有一个 `forEach` 方法，与数组的 `forEach` 方法类似，也可以实现遍历。

```
const map = new Map([[1, 'one'], [2, 'two'], [3, 'three']]);
const reporter = {
  report: function(key, value) {
    console.log("Key: %s, Value: %s", key, value);
  }
};
map.forEach(function(value, key, map) {
  this.report(key, value); // this 指向 reporter
}, reporter); // forEach 第二个参数绑定 this
// Key: 1, Value: one
// Key: 2, Value: two
// Key: 3, Value: three
```

2.4.3. 遍历的应用

Map 结构转为数组结构，比较快速的方法是使用扩展运算符（`...`）。

```
const map = new Map([[1, 'one'], [2, 'two'], [3, 'three']]);
[...map.keys()];    // [1, 2, 3]
[...map.values()];  // ['one', 'two', 'three']
[...map.entries()]; // [[1,'one'], [2, 'two'], [3, 'three']]
[...map];           // [[1,'one'], [2, 'two'], [3, 'three']]
```

结合数组的 `map`、`filter` 方法，可以实现 Map 的遍历和过滤。

```
const m = new Map().set(1, 'a').set(2, 'b').set(3, 'c');
const map1 = new Map([...m].filter(([k, v]) => k < 3));
// 产生 Map 结构 {1 => 'a', 2 => 'b'}
```

```
const map2 = new Map([...m].map(([k, v]) => [k * 2, '_' + v]));
// 产生 Map 结构 {2 => '_a', 4 => '_b', 6 => '_c'}
```

2.5. 与其他数据结构的相互转换

2.5.1. Map 转为数组

Map 转为数组最方便的方法，就是使用扩展运算符（...）。

```
const myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);
[...myMap]; // [[true, 7], [{foo: 3}, ['abc']]]
```

2.5.2. 数组 转为 Map

将数组传入 Map 构造函数，就可以转为 Map。

```
let array = [[true, 7], [{foo: 3}, ['abc']]];
let m = new Map(array);
// Map(2) {true => 7, Object {foo: 3} => ['abc']}
```

2.5.3. Map 转为对象

如果所有 Map 的键都是字符串，它可以无损地转为对象。

```
function strMapToObj(strMap) {
  let obj = Object.create(null);
  for (let [k,v] of strMap) {
    obj[k] = v;
  }
  return obj;
}
const myMap = new Map().set('yes', true).set('no', false);
strMapToObj(myMap); // { yes: true, no: false }
```

如果有非字符串的键名，那么这个键名会被转成字符串，再作为对象的键名。

2.5.4. 对象转为 Map

对象转为 Map 可以通过 `Object.entries()`。

```
let obj = {"a":1, "b":2};
let map = new Map(Object.entries(obj));
// map: Map(2) {'a' => 1, 'b' => 2}
```

2.5.5. Map 转为 JSON

Map 转为 JSON 要区分两种情况。一种情况是，Map 的键名都是字符串，这时可以选择转为对象 JSON。

```
function strMapToJson(strMap) {
  return JSON.stringify(strMapToObj(strMap));
}
let myMap = new Map().set('yes', true).set('no', false);
strMapToJson(myMap); // '{"yes":true,"no":false}'
```

另一种情况是，Map 的键名有非字符串，这时可以选择转为数组 JSON。

```
function mapToArrayJson(map) {
  return JSON.stringify([...map]);
}
let myMap = new Map().set(true, 7).set({foo: 3}, ['abc']);
mapToArrayJson(myMap); // '[[true,7],[{"foo":3},["abc"]]]'
```

2.5.6. JSON 转为 Map

JSON 转为 Map，正常情况下，所有键名都是字符串。

```
function jsonToStrMap(jsonStr) {
  return objToStrMap(JSON.parse(jsonStr));
}
jsonToStrMap('{"yes": true, "no": false}');
// Map {'yes' => true, 'no' => false}
```

但是，有一种特殊情况，整个 JSON 就是一个数组，且每个数组成员本身，又是一个有两个成员的数组。这时，它可以一一对应地转为 Map。这往往是 Map 转为数组 JSON 的逆操作。

```
function jsonToMap(jsonStr) {
  return new Map(JSON.parse(jsonStr));
}
jsonToMap('[[true,7],[{"foo":3},["abc"]]]');
// Map {true => 7, Object {foo: 3} => ['abc']}
```

Promise 对象

1. Promise 的含义

`Promise` 是一个对象，从它可以获取异步操作的消息，它提供统一的接口，使得控制异步操作更加容易，避免了层层嵌套的回调函数（回调地狱）。

`Promise` 对象有以下两个特点：

> - 对象的状态不受外界影响。
>   `Promise` 对象代表一个异步操作，有三种状态：`pending`（进行中）、`fulfilled`（已成功）和 `rejected`（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。
> - 一旦状态改变，就不会再变，任何时候都可以得到这个结果。
>   `Promise` 对象的状态改变，只有两种可能：从 `pending` 变为 `fulfilled` 和从 `pending` 变为 `rejected`。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 resolved（已定型）。如果改变已经发生了，再对 `Promise` 对象添加回调函数，也会立即得到这个结果。

`Promise` 也有一些缺点：

> - 无法取消 `Promise`，一旦新建它就会立即执行，无法中途取消。
> - 如果不设置回调函数，`Promise` 内部抛出的错误，不会反应到外部。
> - 当处于 `pending` 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

2. 基本用法

`Promise` 对象是一个构造函数，用来生成 `Promise` 实例。它接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。`resolve` 函数的作用是将 `Promise` 对象的状态从“未完成”变为“成功”（即从 `pending` 变为 `resolved`），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；`reject` 函数的作用是将 `Promise` 对象的状态从“未完成”变为“失败”（即从 `pending` 变为 `rejected`），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

`Promise` 新建后就会立即执行。

```
let promise = new Promise(function(resolve, reject) {
  console.log('Promise'); // 新建 Promise 实例后就会立即执行，甚至在同步代码之前
  resolve();
});
promise.then(function() {
  console.log('resolved.'); // 在所有同步任务执行完才会执行
});
console.log('Hi!');
// Promise
// Hi!
// resolved
```

调用 `resolve` 或 `reject` 并不会终结 `Promise` 的参数函数的执行。

```
new Promise((resolve, reject) => {
  resolve(1);
  console.log(2);
}).then(r => {
  console.log(r);
});
// 2
// 1
```

立即 `resolved` 的 `Promise` 是在本轮事件循环的末尾执行，总是晚于本轮循环的同步任务。

一般来说，调用 `resolve` 或 `reject` 以后，`Promise` 的使命就完成了，后继操作应该放到 `then` 方法里面，而不应该直接写在 `resolve` 或 `reject` 的后面。所以，最好在它们前面加上 `return` 语句，这样就不会有意外。

```
new Promise((resolve, reject) => {
  return resolve(1);
  console.log(2); // 该条语句不会执行
})
```

Promise.prototype.then()

`Promise` 实例具有 `then` 方法，它的作用是为 `Promise` 实例添加状态改变时的回调函数，`then` 方法的第一个参数是 `resolved` 状态的回调函数，第二个参数是 `rejected` 状态的回调函数，它们都是可选的。

`then` 方法返回的是一个新的 `Promise` 实例，因此可以采用链式写法，即 `then` 方法后面再调用另一个 `then` 方法。采用链式的 `then`，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个 `Promise` 对象（即有异步操作），这时后一个回调函数，就会等待该 `Promise` 对象的状态发生变化，才会被调用。

```
getJSON("/post/1.json").then(
  post => getJSON(post.commentURL)
).then(
  comments => console.log("resolved: ", comments),
  err => console.log("rejected: ", err)
);
```

上例中，第一个 `then` 方法指定的回调函数，返回的是另一个 `Promise` 对象。这时，第二个 `then` 方法指定的回调函数，就会等待这个新的 `Promise` 对象状态发生变化。如果变为 `resolved`，就调用第一个回调函数，如果状态变为 `rejected`，就调用第二个回调函数。

4. Promise.prototype.catch()

`Promise.prototype.catch()` 方法是 `.then(null, rejection)` 或 `.then(undefined, rejection)` 的别名，用于指定发生错误时的回调函数。

```
p.then((val) => console.log('fulfilled:', val)).catch((err) =>
console.log('rejected', err));
// 等同于
p.then((val) => console.log('fulfilled:', val)).then(null, (err) =>
console.log("rejected:", err));
```

如果 Promise 状态已经变成 resolved，再抛出错误是无效的。

```
const promise = new Promise(function(resolve, reject) {
  resolve('ok');
  throw new Error('test');
});
promise
  .then(function(value) { console.log(value) })
  .catch(function(error) { console.log(error) });
// ok
```

Promise 在 resolve 语句后面，再抛出错误，不会被捕获，等于没有抛出。**因为 Promise 的状态一旦改变，就永久保持该状态，不会再变了。**

不要在 then() 方法里面定义 reject 状态的回调函数（即 then 的第二个参数），总是使用 catch 方法，它捕获前面 then 方法执行中的错误，也更接近同步的写法（try/catch）。

```
promise.then(function(data) { /* success */ }, function(err) { /* error */ }); //
bad
promise.then(function(data) { /* success */ }).catch(function(err) { /* error */
}); // good
```

catch() 方法之中，还能再抛出错误。

```
Promise.resolve().then(function() {
  return x + 2; // 该行会报错，因为 x 没有声明
}).catch(function(error) {
  console.log('oh no', error);
  y + 2; // 该行会报错，因为 y 没有声明
}).catch(function(error) { // 第二个 catch 方法用来捕获前一个 catch 方法抛出的错误。
  console.log('carry on', error);
});
// oh no ReferenceError: x is not defined
// carry on ReferenceError: y is not defined
```

5. Promise.prototype.finally()

`finally()` 方法用于指定不管 `Promise` 对象最后状态如何，都会执行的操作。`finally()` 方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 `Promise` 状态到底是 `fulfilled` 还是`rejected`。**`finally()` 方法里面的操作，应该是与状态无关的，不依赖于 `Promise` 的执行结果。**

```
promise.then(result => { /* success */ }).catch(error => { /* error */
}).finally(() => { /* finally */ });
```

`finally` 本质上是 `then` 方法的特例。

```
promise.finally(() => { /* 语句 */ });
// 等同于
promise.then(result => { /* 语句 */ return result; }, error => { /* 语句 */ throw
error; });
```

6. Promise.all()

`Promise.all()` 方法用于将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。**记忆同数组的 every 方法，rejected 对应 false，fulfilled 对应 true。**

> - 只有参数实例的状态都变成 `fulfilled`，包装实例的状态才会变成 `fulfilled`，此时参数实例的返回值组成一个数组，传递给包装实例的回调函数。
> - 只要参数实例之中有一个被 `rejected`，包装实例的状态就变成 `rejected`，此时第一个被 `reject` 的实例的返回值，会传递给包装实例的回调函数。

```
// 生成一个 Promise 对象的数组
const promises = [2, 3, 5, 7, 11, 13].map(function (id) {
  return getJSON('/post/' + id + ".json");
});
Promise.all(promises).then(function (posts) { /* */ }).catch(function(reason){ /*
*/ });
```

`promises` 是包含 6 个 `Promise` 实例的数组，只有这 6 个实例的状态都变成 `fulfilled`，或者其中有一个变为 `rejected`，才会调用 `Promise.all` 方法后面的回调函数。

如果作为参数的 Promise 实例，自己定义了 `catch` 方法，那么它一旦被 `rejected`，并不会触发 `Promise.all()` 的 `catch` 方法。

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
}).then(result => result)
.catch(e => e);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
}).then(result => result)
```

```
  .catch(e => e);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// ["hello", Error: 报错了]
```

如果 p2 没有自己的 `catch` 方法，就会调用 `Promise.all()` 的 `catch` 方法。

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
}).then(result => result);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
}).then(result => result);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// Error: 报错了
```

7. Promise.race()

`Promise.race()` 方法同样是将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。

> - 只要参数实例之中有一个实例率先改变状态，包装实例的状态就跟着改变。那个率先改变的 `Promise` 实例的返回值，就传递给包装实例的回调函数。

```
const p = Promise.race([
  fetch('/resource-that-may-take-a-while'),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
]);
p.then(console.log).catch(console.error);
```

如果 5 秒之内 `fetch` 方法无法返回结果，变量 `p` 的状态就会变为 `rejected`，从而触发 `catch` 方法指定的回调函数。

8. Promise.allSettled()

`Promise.race()` 方法同样是将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。用来确定一组异步操作是否都结束了（不管成功或失败）。

> - 只有等到参数数组的所有 `Promise` 对象都发生状态变更（不管是 `fulfilled` 还是 `rejected`），返回的 Promise 对象才会发生状态变更。

一旦发生状态变更，状态总是 `fulfilled`，不会变成 `rejected`。状态变成 `fulfilled` 后，它的回调函数会接收到一个数组作为参数，该数组的每个成员对应前面数组的每个 `Promise` 对象。

```
const resolved = Promise.resolve(42);
const rejected = Promise.reject(-1);

const allSettledPromise = Promise.allSettled([resolved, rejected]);
allSettledPromise.then(function (results) {
  console.log(results);
});
// [
//    { status: 'fulfilled', value: 42 },
//    { status: 'rejected', reason: -1 }
// ]
```

成员对象的 `status` 属性的值只可能是字符串 `fulfilled` 或 `rejected`，用来区分异步操作是成功还是失败。如果是成功（`fulfilled`），对象会有 `value` 属性，如果是失败（`rejected`），会有 `reason` 属性，对应两种状态时前面异步操作的返回值。

9. Promise.any()

`Promise.race()` 方法同样是将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。**记忆同数组的 some 方法，rejected 对应 false，fulfilled 对应 true。**

> - 只要参数实例有一个变成 `fulfilled` 状态，包装实例就会变成 `fulfilled` 状态。
> - 如果所有参数实例都变成 `rejected` 状态，包装实例就会变成 `rejected` 状态。

```
Promise.any([
  fetch('https://v8.dev/').then(() => 'home'),
  fetch('https://v8.dev/blog').then(() => 'blog'),
  fetch('https://v8.dev/docs').then(() => 'docs')
]).then((first) => {  // 只要有一个 fetch() 请求成功
  console.log(first);
}).catch((error) => { // 所有三个 fetch() 全部请求失败
  console.log(error); // AggregateError: All promises were rejected
});
```

`Promise.any()` 抛出的错误是一个 `AggregateError` 实例。

```
let resolved = Promise.resolve(42);
let rejected = Promise.reject(-1);
let alsoRejected = Promise.reject(Infinity);

Promise.any([resolved, rejected, alsoRejected]).then(function (result) {
  console.log(result); // 42
});
```

```
Promise.any([rejected, alsoRejected]).catch(function (results) {
  console.log(results instanceof AggregateError); // true
  console.log(results.errors); // [-1, Infinity]
});
```

10. Promise.resolve()

有时需要将现有对象转为 Promise 对象，`Promise.resolve()` 方法就起到这个作用。

```
Promise.resolve('foo');
// 等价于
new Promise(resolve => resolve('foo'));
```

立即 `resolve()` 的 `Promise` 对象，是在本轮“事件循环”（`event loop`）的结束时执行，而不是在下一轮“事件循环”的开始时。

```
setTimeout(function () {
  console.log('three'); // 在下一轮“事件循环”开始时执行
}, 0);
Promise.resolve().then(function () {
  console.log('two'); // 在本轮“事件循环”结束时执行
});
console.log('one'); // 立即执行
// one
// two
// three
```

`Promise.resolve()` 方法的参数分成四种情况。

（1）参数是一个 Promise 实例

如果参数是 Promise 实例，那么 `Promise.resolve()` 将不做任何修改、原封不动地返回这个实例。

（2）参数是一个 `thenable` 对象

`thenable` 对象指的是具有 `then` 方法的对象。`Promise.resolve()` 方法会将这个对象转为 `Promise` 对象，然后就立即执行 `thenable` 对象的 `then()` 方法。

```
let thenable = {
  then: function(resolve, reject) { resolve(42); }
};
let p1 = Promise.resolve(thenable);
p1.then(function (value) {
  console.log(value);  // 42
});
```

`thenable` 对象的 `then()` 方法执行后，对象 `p1` 的状态就变为 `resolved`，从而立即执行最后那个 `then()` 方法指定的回调函数，输出 `42`。

（3）参数不是具有 `then()` 方法的对象，或根本就不是对象

如果参数是一个原始值，或者是一个不具有 `then()` 方法的对象，则 `Promise.resolve()` 方法返回一个新的 `Promise` 对象，状态为 `resolved`。

```
const p = Promise.resolve('Hello');
p.then(function (s) { console.log(s) }); // Hello
```

上例生成一个新的 `Promise` 对象的实例 `p`。由于字符串 `Hello` 不属于异步操作（判断方法是字符串对象不具有 `then` 方法），返回 `Promise` 实例的状态从一生成就是 `resolved`，所以回调函数会立即执行。`Promise.resolve()` 方法的参数，会同时传给回调函数。

（4）不带有任何参数

`Promise.resolve()` 方法允许调用时不带参数，直接返回一个 `resolved` 状态的 `Promise` 对象。

如果希望得到一个 `Promise` 对象，比较方便的方法就是直接调用 `Promise.resolve()` 方法。

```
const p = Promise.resolve(); // p 就是一个 Promise 对象
p.then(function () {   /* success */ });
```

11. Promise.reject()

`Promise.reject(reason)` 方法也会返回一个新的 `Promise` 实例，该实例的状态为 `rejected`。

```
const p = Promise.reject('出错了');
// 等同于
const p = new Promise((resolve, reject) => reject('出错了'));
p.then(null, function (s) { console.log(s); }); // 出错了
// 实例 p 状态为 `rejected`，回调函数会立即执行。
```

`Promise.reject()` 方法的参数，会原封不动地作为 `reject` 的理由，变成后续方法的参数。

```
Promise.reject('出错了').catch(e => { console.log(e === '出错了'); }); // true
// `Promise.reject()` 方法的参数是一个字符串，后面 `catch()` 方法的参数 `e` 就是这个字符串。
```

async 函数

1. 基本用法

async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。当函数执行的时候，一旦遇到 await 就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}
async function asyncPrint(value, ms) {
  await timeout(ms);
  console.log(value);
}
asyncPrint('hello world', 500);
// 500ms 后输出 hello world
```

async 函数有多种使用形式。

```
async function foo() {} // 函数声明

const foo = async () => {}; // 箭头函数

const foo = async function () {}; // 函数表达式

// 对象的方法
let obj = { async foo() {} };
obj.foo().then(...)

// Class 的方法
class Storage {
  constructor() {
    this.cachePromise = caches.open('avatars');
  }
  async getAvatar(name) {
    const cache = await this.cachePromise;
    return cache.match(`/avatars/${name}.jpg`);
  }
}
const storage = new Storage();
storage.getAvatar('jake').then(…);
```

2. 语法

2.1. 返回 Promise 对象

async 函数返回一个 Promise 对象。**async 函数内部 return 语句返回的值，会成为 then 方法回调函数的参数。**

```
async function f() {
  return 'hello world'; // 这里返回的值会被 f 函数 then 方法接收
}
f().then(v => console.log(v)); // "hello world"
```

async 函数内部抛出错误，会导致返回的 Promise 对象变为 reject 状态。**抛出的错误对象会被 catch 方法回调函数接收到。**

```
async function f() {
  throw new Error('出错了'); // 这里抛出的错误会被 f 函数 catch 方法（then 方法第二个参数）接收
}
f().then(v => console.log('resolve', v), e => console.log('reject', e)); // reject Error: 出错了
```

2.2. Promise 对象的状态变化

async 函数返回的 Promise 对象，必须等到内部所有 await 命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到 return 语句或者抛出错误。也就是说，只有 async 函数内部的异步操作执行完，才会执行 then 方法指定的回调函数。

```
async function getTitle(url) {
  let response = await fetch(url);
  let html = await response.text();
  return html.match(/<title>([\s\S]+)<\/title>/i)[1];
}
getTitle('https://tc39.github.io/ecma262/').then(console.log);
// "ECMAScript 2017 Language Specification"
```

上面代码中，函数 getTitle 内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行 then 方法里面的 console.log。

2.3. await 命令

如果 await 命令后面是一个 Promise 对象，返回该对象的结果。如果不是 Promise 对象，就直接返回对应的值。

```
async function f() {
  return await 123; // 等同于 return 123;
```

```
}
f().then(v => console.log(v)); // 123
```

如果 await 命令后面是一个 thenable 对象（即定义了 then 方法的对象），那么 await 会将其等同于 Promise 对象。

```
class Sleep {
  constructor(timeout) {
    this.timeout = timeout;
  }
  then(resolve, reject) {
    const startTime = Date.now();
    setTimeout(
      () => resolve(Date.now() - startTime),
      this.timeout
    );
  }
}
(async () => {
  const sleepTime = await new Sleep(1000);
  console.log(sleepTime); // 1000
})();
```

上例中，await 命令后面是一个 Sleep 对象的实例。这个实例不是 Promise 对象，但是因为定义了 then 方法，await 会将其视为 Promise 处理。

JavaScript 一直没有休眠的语法，但是借助 await 命令就可以让程序停顿指定的时间。简化的 sleep 实现：

```
function sleep(interval) {
  return new Promise(resolve => {
    setTimeout(resolve, interval);
  })
}
// 用法
async function one2FiveInAsync() {
  for(let i = 1; i <= 3; i++) {
    console.log(i);
    await sleep(1000);
  }
}
one2FiveInAsync(); // 首先输出 1，1s 后输出 2，再 1s 后输出 3
```

await 命令后面的 Promise 对象如果变为 reject 状态，则 reject 的参数会被 catch 方法的回调函数接收到。

```
async function f() {
  await Promise.reject('出错了');
```

```
}
f().then(v => console.log("v", v)).catch(e => console.log("e", e)); // e 出错了
```

任何一个 `await` 语句后面的 `Promise` 对象变为 `reject` 状态，那么整个 `async` 函数都会中断执行。

```
async function f() {
  await Promise.reject('出错了');
  await Promise.resolve('hello world'); // 不会执行
}
f().then(v => console.log("v", v)).catch(e => console.log("e", e)); // e 出错了
```

上例中，第二个 `await` 语句是不会执行的，因为第一个 `await` 语句状态变成了 `reject`。

如果希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个 `await` 放在 `try...catch` 结构里面，这样不管这个异步操作是否成功，第二个 `await` 都会执行。

```
async function f() {
  try {
    await Promise.reject('出错了');
  } catch(e) {
  }
  return await Promise.resolve('hello world');
}

f().then(v => console.log(v)); // hello world
```

另一种方法是 `await` 后面的 `Promise` 对象再跟一个 `catch` 方法，处理前面可能出现的错误。

```
async function f() {
  await Promise.reject('出错了').catch(e => console.log("e", e));
  return await Promise.resolve('hello world');
}
f().then(v => console.log("v", v));
// e 出错了
// v hello world
```

2.4. 错误处理

如果 `await` 后面的异步操作出错，那么等同于 `async` 函数返回的 `Promise` 对象被 `reject`。

```
async function f() {
  await new Promise(function (v, e) {
    throw new Error('出错了');
  });
}
```

```
f().then(v => console.log("v", v)).catch(e => console.log("e", e)); // e Error: 出
错了
```

防止出错的方法，也是将其放在 `try...catch` 代码块之中。

```
async function f() {
  try {
    await new Promise(function (resolve, reject) {
      throw new Error('出错了');
    });
  } catch(e) {
  }
  return await('hello world');
}
```

如果有多个 `await` 命令，可以统一放在 `try...catch` 结构中。

```
async function main() {
  try {
    const val1 = await firstStep();
    const val2 = await secondStep(val1);
    const val3 = await thirdStep(val1, val2);
    console.log('Final: ', val3);
  }
  catch (err) {
    console.error(err);
  }
}
```

下面的例子使用 `try...catch` 结构，实现多次重复尝试。

```
const superagent = require('superagent');
const NUM_RETRIES = 3;

async function test() {
  let i;
  for (i = 0; i < NUM_RETRIES; ++i) {
    try {
      await superagent.get('http://google.com/this-throws-an-error');
      break;
    } catch(err) {}
  }
  console.log(i); // 3
}

test();
```

上例中，如果 await 操作成功，就会使用 break 语句退出循环；如果失败，会被 catch 语句捕捉，然后进入下一轮循环。

2.5. 使用注意点

> 1. await 命令后面的 Promise 对象，运行结果可能是 rejected，所以最好把 await 命令放在 try...catch 代码块中。

```
async function myFunction() {
  try {
    await somethingThatReturnsAPromise();
  } catch (err) {
    console.log(err);
  }
}
// 另一种写法
async function myFunction() {
  await somethingThatReturnsAPromise().catch(function (err) {
    console.log(err);
  });
}
```

> 2. 多个 await 命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}
(async function asyncPrint() {
  await timeout(3000);
  await timeout(5000);
  console.log("a"); // 8s 后输出 a
})();
```

上例中，asyncPrint 函数中 timeout(3000) 和 timeout(5000) 是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有 timeout(3000) 完成以后，才会执行 timeout(5000)，timeout(5000) 完成以后才会输出 a，完全可以让它们同时触发。

```
// 写法一
await Promise.all([timeout(3000), timeout(5000)]);

// 写法二
let p1 = timeout(3000);
let p2 = timeout(5000);
await p1;
await p2;
```

上面两种写法，`timeout(3000)` 和 `timeout(5000)` 都是同时触发，这样就会缩短程序的执行时间。5s 后会输出 `a`。

> 3. `await` 命令只能用在 `async` 函数之中，如果用在普通函数，就会报错。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];

  // 报错
  docs.forEach(function (doc) {
    await db.post(doc);
  });
}
```

如果确实希望多个请求并发执行，可以使用 `Promise.all` 方法。当三个请求都会 `resolved` 时，下面两种写法效果相同。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];
  let promises = docs.map((doc) => db.post(doc));

  let results = await Promise.all(promises);
  console.log(results);
}
// 或者使用下面的写法
async function dbFuc(db) {
  let docs = [{}, {}, {}];
  let promises = docs.map((doc) => db.post(doc));

  let results = [];
  for (let promise of promises) {
    results.push(await promise);
  }
  console.log(results);
}
```

> 4. async 函数可以保留运行堆栈。

```
const a = () => {
  b().then(() => c());
};
```

上例中，函数 `a` 内部运行了一个异步任务 `b()`。当 `b()` 运行的时候，函数 `a()` 不会中断，而是继续执行。等到 `b()` 运行结束，可能 `a()` 早就运行结束了，`b()` 所在的上下文环境已经消失了。如果 `b()` 或 `c()` 报错，错误堆栈将不包括 `a()`。

现在将这个例子改成 `async` 函数。

```
const a = async () => {
  await b();
  c();
};
```

上例中，`b()` 运行的时候，`a()` 是暂停执行，上下文环境都保存着。一旦 `b()` 或 `c()` 报错，错误堆栈将包括 `a()`。

3. 顶层 await

允许在模块的顶层独立使用 `await` 命令，解决模块异步加载的问题。

```
// import() 方法加载
const strings = await import(`/i18n/${navigator.language}`);

// 数据库操作
const connection = await dbConnector();

// 依赖回滚
let jQuery;
try {
  jQuery = await import('https://cdn-a.com/jQuery');
} catch {
  jQuery = await import('https://cdn-b.com/jQuery');
}
```

Class

1. 类的由来

JavaScript 语言中，生成实例对象的传统方法是通过构造函数：

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.toString = function () {
  return '(' + this.x + ', ' + this.y + ')';
};

let p = new Point(1, 2);
```

ES6 提供了更接近传统语言的写法，引入了 `Class`（类）这个概念，作为对象的模板。通过 `class` 关键字，可以定义类。

上面的代码用 ES6 的 class 改写，就是下面这样：

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}
```

上面 `constructor()` 方法，这就是构造方法，而 `this` 关键字则代表实例对象。这种新的 `Class` 写法，本质上与 ES5 的构造函数 `Point` 是一致的。

`Point` 类除了构造方法，还定义了一个 `toString()` 方法。**定义 `toString()` 方法的时候，前面不需要加上 `function` 这个关键字，直接把函数定义放进去了就可以了。方法与方法之间不需要逗号分隔，加了会报错。**

ES6 的类，完全可以看作构造函数的另一种写法。

```
class Point {
  // ...
}

typeof Point // "function"
Point === Point.prototype.constructor // true
```

类的数据类型就是函数，类本身就指向构造函数。

构造函数的 prototype 属性，在 ES6 的"类"上面继续存在。事实上，类的所有方法都定义在类的 prototype 属性上面。

```
class Point {
  constructor() { /* xxx */ }
  toString() { /* xxx */ }
  toValue() { /* xxx */ }
}

// 等同于
Point.prototype = {
  constructor() {},
  toString() {},
  toValue() {},
};
```

上面代码中，constructor()、toString()、toValue() 这三个方法，其实都是定义在 Point.prototype 上面。

因此，在类的实例上面调用方法，其实就是调用原型上的方法。

```
class B {}
const b = new B();

b.constructor === B.prototype.constructor // true
```

上面代码中，b 是 B 类的实例，它的 constructor() 方法就是 B 类原型的 constructor() 方法。

由于类的方法都定义在 prototype 对象上面，所以类的新方法可以添加在 prototype 对象上面。Object.assign() 方法可以很方便地一次向类添加多个方法。

```
class Point {
  constructor(){ /* xxx */ }
}
Object.assign(Point.prototype, {
  toString(){},
  toValue(){}
});
```

prototype 对象的 constructor 属性，直接指向"类"的本身，这与 ES5 的行为是一致的。

```
Point.prototype.constructor === Point // true
```

类的内部所有定义的方法，都是不可枚举的（non-enumerable）。

```
class Point {
  constructor(x, y) { /* xxx */ }
  toString() { /* xxx */ }
}

Object.keys(Point.prototype); // []
Object.getOwnPropertyNames(Point.prototype); // ["constructor","toString"]
```

上面代码中，toString()方法是Point类内部定义的方法，它是不可枚举的。这一点与 ES5 的行为不一致。

```
let Point = function (x, y) { /* xxx */ };
Point.prototype.toString = function () { /* xxx */ };
Object.keys(Point.prototype); // ["toString"]
Object.getOwnPropertyNames(Point.prototype); // ["constructor","toString"]
```

上面代码采用 ES5 的写法，`toString()` 方法就是可枚举的。

2. constructor() 方法

`constructor()` 方法是类的默认方法，通过 `new` 命令生成对象实例时，自动调用该方法。 一个类必须有 `constructor()` 方法，如果没有显式定义，一个空的 `constructor()` 方法会被默认添加。

```
class Point {
}
// 等同于
class Point {
  constructor() {}
}
```

上面代码中，定义了一个空的类 `Point`，JavaScript 引擎会自动为它添加一个空的 `constructor()` 方法。

constructor()方法默认返回实例对象（即this），完全可以指定返回另外一个对象。

```
class Foo {
  constructor() {
    return Object.create(null);
  }
}
new Foo() instanceof Foo
// false
```

上面代码中，constructor()函数返回一个全新的对象，结果导致实例对象不是Foo类的实例。

类必须使用 new 调用，否则会报错。 这是它跟普通构造函数的一个主要区别，后者不用 new 也可以执行。

```
class Foo {
  constructor() {
    return Object.create(null);
  }
}
Foo()
// TypeError: Class constructor Foo cannot be invoked without 'new'
```

3. 类的实例

类的属性和方法，除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}

let point = new Point(2, 3);

point.toString() // (2, 3)

point.hasOwnProperty('x') // true
point.hasOwnProperty('y') // true
point.hasOwnProperty('toString') // false
point.__proto__.hasOwnProperty('toString') // true
```

上面代码中，`x` 和 `y` 都是实例对象 `point` 自身的属性（因为定义在 `this` 对象上），所以`hasOwnProperty()` 方法返回 `true`，而 `toString()` 是原型对象的属性（因为定义在 `Point` 类上），所以 `hasOwnProperty()` 方法返回 `false`。

类的所有实例共享一个原型对象。

```
var p1 = new Point(2,3);
var p2 = new Point(3,2);

p1.__proto__ === p2.__proto__; // true
p1.__proto__ === Point.prototype; // true
```

4. 实例属性的新写法

实例属性现在除了可以定义在 `constructor()` 方法里面的 `this` 上面，也可以定义在类内部的最顶层。

```
// 原来的写法
class IncreasingCounter {
  constructor() {
    this._count = 0;
  }
  get value() {
    console.log('Getting the current value!');
    return this._count;
  }
  increment() {
    this._count++;
  }
}
```

上面示例中，实例属性 `_count` 定义在 `constructor()` 方法里面的 `this` 上面。

现在的新写法是，这个属性也可以定义在类的最顶层，其他都不变。

```
class IncreasingCounter {
  _count = 0;
  get value() {
    console.log('Getting the current value!');
    return this._count;
  }
  increment() {
    this._count++;
  }
}
```

上面代码中，实例属性 `_count` 与取值函数 `value()` 和 `increment()` 方法，处于同一个层级。这时，不需要在实例属性前面加上 `this`。

新写法定义的属性是实例对象自身的属性，而不是定义在实例对象的原型上面。

这种新写法的好处是，所有实例对象自身的属性都定义在类的头部，看上去比较整齐，一眼就能看出这个类有哪些实例属性。

5. 取值函数（getter）和存值函数（setter）

```
class MyClass {
  constructor() {
    // ...
  }
```

```
  get prop() {
    return 'getter';
  }
  set prop(value) {
    console.log('setter: '+value);
  }
}

let inst = new MyClass();

inst.prop = 123;
// setter: 123

inst.prop
// 'getter'
```

6. 属性表达式

```
let methodName = 'getArea';
class Square {
  constructor(length) {
    // ...
  }
  [methodName]() {
    // ...
  }
}
```

`Square` 类的方法名 `getArea`，是从表达式得到的。

7. Class 表达式

采用 Class 表达式，可以写出立即执行的 Class。

```
let person = new class {
  constructor(name) {
    this.name = name;
  }
  sayName() {
    console.log(this.name);
  }
}('张三');

person.sayName(); // "张三"
```

8. 静态方法

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上 `static` 关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}
Foo.classMethod(); // 'hello'
let foo = new Foo();
foo.classMethod(); // TypeError: foo.classMethod is not a function
```

如果静态方法包含this关键字，这个 `this` 指的是类，而不是实例。

```
class Foo {
  static bar() {
    this.baz();
  }
  static baz() {
    console.log('hello');
  }
  baz() {
    console.log('world');
  }
}
Foo.bar(); // hello
let f = new Foo();
f.baz(); // world
```

父类的静态方法，可以被子类继承。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}
class Bar extends Foo {
}
Bar.classMethod() // 'hello'
```

9. 静态属性

在实例属性的前面，加上 `static` 关键字。

```
class Foo {
  static prop = 1;
```

```
}
Foo.prop; // 1
let f = new Foo();
f.prop; // undefined
```

10. 私有属性

在属性名之前使用 # 表示。

```
class IncreasingCounter {
  #count = 0;
  get value() {
    console.log('Getting the current value!');
    return this.#count;
  }
  increment() {
    this.#count++;
  }
}
```

#count 就是私有属性，只能在类的内部使用（this.#count）。如果在类的外部使用，就会报错。

```
const counter = new IncreasingCounter();
counter.#count // 报错，从 Chrome 111 开始，开发者工具里面可以读写私有属性，不会报错，
原因是 Chrome 团队认为这样方便调试。其他浏览器和 Chrome 111 以下会报错。
counter.#count = 42 // 报错
```

上面示例中，在类的外部，读取或写入私有属性 #count，都会报错。

11. in 运算符

in 运算符，使它也可以用来判断私有属性。也可以跟this一起配合使用。

```
class A {
  #foo = 0;
  m() {
    console.log(#foo in this);
  }
}
let a = new A();
a.m(); // true
```

判断私有属性时，in 只能用在类的内部。另外，判断所针对的私有属性，一定要先声明，否则会报错。

```
class A {
  m() {
    console.log(#foo in this); // Private field '#foo' must be declared in an
enclosing class。私有字段“#foo”必须在封闭类中声明
  }
}
```

Class 的继承

1. 简介

Class 可以通过 `extends` 关键字实现继承，让子类继承父类的属性和方法。`extends` 的写法比 ES5 的原型链继承，要清晰和方便很多。

```
class Point { /* ... */ }
class ColorPoint extends Point {
  constructor(x, y, color) {
    super(x, y); // 调用父类的constructor(x, y)
    this.color = color;
  }
  toString() {
    return this.color + ' ' + super.toString(); // 调用父类的toString()
  }
}
```

`constructor()` 方法和 `toString()` 方法内部，都出现了 `super` 关键字。`super` 在这里表示父类的构造函数，用来新建一个父类的实例对象。

子类必须在 `constructor()` 方法中调用 `super()`，否则就会报错。因为**子类自己的 `this` 对象，必须先通过父类的构造函数完成塑造，得到与父类同样的实例属性和方法，然后再对其进行加工，添加子类自己的实例属性和方法。如果不调用 `super()` 方法，子类就得不到自己的 `this` 对象。**

```
class Point { /* ... */ }
class ColorPoint extends Point {
  constructor() {
  }
}
let cp = new ColorPoint(); // ReferenceError: Must call super constructor in
derived class before accessing 'this' or returning from derived constructor
```

`ColorPoint` 继承了父类 `Point`，但是它的构造函数没有调用 `super()`，导致新建实例时报错。

为什么子类的构造函数，一定要调用super()?

> - ES6 的继承机制，与 ES5 完全不同。
> - ES5 的继承机制，是先创造一个独立的子类的实例对象，然后再将父类的方法添加到这个对象上面，即“实例在前，继承在后”。
> - ES6 的继承机制，则是先将父类的属性和方法，加到一个空的对象上面，然后再将该对象作为子类的实例，即“继承在前，实例在后”。这就是为什么 ES6 的继承必须先调用 `super()` 方法，因为这一步会生成一个继承父类的 `this` 对象，没有这一步就无法继承父类。

这意味着新建子类实例时，父类的构造函数必定会先运行一次。

```
class Foo {
  constructor() {
    console.log(1);
  }
}
class Bar extends Foo {
  constructor() {
    super(); // 调用一次父类 Foo 的构造函数，输出 1
    console.log(2);
  }
}
const bar = new Bar();
// 1
// 2
```

在子类的构造函数中，只有调用 `super()` 之后，才可以使用 this 关键字，否则会报错。

这是因为子类实例的构建，必须先完成父类的继承，只有 `super()` 方法才能让子类实例继承父类。

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
class ColorPoint extends Point {
  constructor(x, y, color) {
    this.color = color; // ReferenceError
    super(x, y);
    this.color = color; // 正确
  }
}
```

上面代码中，子类的 `constructor()` 方法没有调用 `super()` 之前，就使用 `this` 关键字，结果报错，而放在 `super()` 之后就是正确的。

如果子类没有定义 `constructor()` 方法，这个方法会默认添加，并且里面会调用 `super()`。也就是说，不管有没有显式定义，任何一个子类都有 `constructor()` 方法。

```
class ColorPoint extends Point {}
// 等同于
class ColorPoint extends Point {
  constructor(...args) {
    super(...args);
  }
}
```

有了子类的定义，就可以生成子类的实例了。

```
let cp = new ColorPoint(25, 8, 'green');

cp instanceof ColorPoint // true
cp instanceof Point // true
```

2. 私有属性和私有方法的继承

父类所有的属性和方法，都会被子类继承，除了私有的属性和方法。子类无法继承父类的私有属性，私有属性只能在定义它的 class 里面使用。

```
class Foo {
  #p = 1;
  #m() {
    console.log('hello');
  }
}
class Bar extends Foo {
  constructor() {
    super();
    console.log(this.#p); // Private field '#p' must be declared in an enclosing
class
    this.#m(); // Private field '#m' must be declared in an enclosing class
  }
}
```

上例中，子类 Bar 调用父类 Foo 的私有属性或私有方法，都会报错。

如果父类定义了私有属性的读写方法，子类就可以通过这些方法，读写私有属性。

```
class Foo {
  #p = 1;
  getP() {
    return this.#p;
  }
}
class Bar extends Foo {
  constructor() {
    super();
    console.log(this.getP()); // 1
  }
}
```

上例中，`getP()` 是父类用来读取私有属性的方法，通过该方法，子类就可以读到父类的私有属性。

3. 静态属性和静态方法的继承

父类的静态属性和静态方法，也会被子类继承。

```
class A {
  static hello() {
    console.log('hello world');
  }
}
class B extends A {}

B.hello()  // hello world
```

静态属性是通过浅拷贝实现继承的。

```
class A { static foo = 100; }
class B extends A {
  constructor() {
    super();
    B.foo--;
  }
}

const b = new B();
B.foo // 99
A.foo // 100
b.foo // undefined
```

上例中，foo 是 A 类的静态属性，B 类继承了 A 类，因此也继承了这个属性。但是，在 B 类内部操作 `B.foo` 这个静态属性，影响不到 `A.foo`，原因就是 B 类继承静态属性时，会采用浅拷贝，拷贝父类静态属性的值，因此A.foo和B.foo是两个彼此独立的属性。

但是，由于这种拷贝是浅拷贝，如果父类的静态属性的值是一个对象，那么子类的静态属性也会指向这个对象，因为浅拷贝只会拷贝对象的内存地址。

```
class A {
  static foo = { n: 100 };
}

class B extends A {
  constructor() {
    super();
    B.foo.n--;
  }
}

const b = new B();
B.foo.n // 99
A.foo.n // 99
```

上例中，`A.foo` 的值是一个对象，浅拷贝导致 `B.foo` 和 `A.foo` 指向同一个对象。所以，子类 B 修改这个对象的属性值，会影响到父类 A。

4. Object.getPrototypeOf()

`Object.getPrototypeOf()` 方法可以用来从子类上获取父类

```
class Point { /*...*/ }
class ColorPoint extends Point { /*...*/ }
Object.getPrototypeOf(ColorPoint) === Point; // true
```

可以使用这个方法判断，一个类是否继承了另一个类。

module

历史上，JavaScript 一直没有模块（module）体系，无法将一个大程序拆分成互相依赖的小文件，再用简单的方法拼装起来。ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

ES6 模块的设计思想是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。

由于 ES6 模块是编译时加载，使得静态分析成为可能。

- 不再需要 `UMD` 模块格式了，将来服务器和浏览器都会支持 ES6 模块格式。目前，通过各种工具库，其实已经做到了这一点。
- 将来浏览器的新 API 就能用模块格式提供，不再必须做成全局变量或者 `navigator` 对象的属性。
- 不再需要对象作为命名空间（比如 `Math` 对象），未来这些功能可以通过模块提供。

ES6 的模块自动采用严格模式，不管有没有在模块头部加上 `"use strict";` 。

一个模块就是一个独立的文件。 模块功能主要由两个命令构成：`export` 和 `import`。`export` 命令用于规定模块的对外接口，`import` 命令用于输入其他模块提供的功能。

export

输出变量：

```
// profile.js
export let firstName = 'Michael';
export let lastName = 'Jackson';
export let year = 1958;
```

```
// profile.js
let firstName = 'Michael';
let lastName = 'Jackson';
let year = 1958;

export { firstName, lastName, year };
```

输出函数或类：

```
export function multiply(x, y) {
  return x * y;
};
```

可以使用 `as` 关键字重命名：

```
function v1() { ... }
function v2() { ... }

export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
```

重命名后，v2 用不同的名字输出了两次。

export 命令能够对外输出的就是三种接口：函数（Functions），类（Classes），var、let、const 声明的变量（Variables）。

export 命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错。 这是因为处于条件代码块之中，就没法做静态优化了，违背了 ES6 模块的设计初衷。

2. import

使用 export 命令定义了模块的对外接口以后，其他 JS 文件就可以通过 import 命令加载这个模块。

```
// main.js
import { firstName, lastName, year } from './profile.js';

function setName(element) {
  element.textContent = firstName + ' ' + lastName;
}
```

上面代码的 import 命令，用于加载 profile.js 文件，并从中输入变量。import 命令接受一对大括号，里面指定要从其他模块导入的变量名。**大括号里面的变量名，必须与导出模块（profile.js）对外接口的名称相同。**

将输入的变量重命名，import 命令要使用 as 关键字：

```
import { lastName as surname } from './profile.js';
```

import命令输入的变量都是只读的，因为它的本质是输入接口。也就是说，不允许在加载模块的脚本里面，改写接口。

```
import {a} from './xxx.js';

a = {}; // Syntax Error : 'a' is read-only;
```

如果a是一个对象，改写a的属性是允许的。不过，这种写法很难查错，建议凡是输入的变量，都当作完全只读，不要轻易改变它的属性。

```
import {a} from './xxx.js';

a.foo = 'hello'; // 合法操作
```

import命令具有提升效果，会提升到整个模块的头部，首先执行。

```
foo();

import { foo } from 'my_module';
```

上面的代码不会报错，因为 `import` 的执行早于 `foo` 的调用。这种行为的本质是，`import` 命令是编译阶段执行的，在代码运行之前。

由于 `import` 是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。在静态分析阶段，这些语法都是没法得到值的。

```
// 报错
import { 'f' + 'oo' } from 'my_module';

// 报错
let module = 'my_module';
import { foo } from module;

// 报错
if (x === 1) {
  import { foo } from 'module1';
} else {
  import { foo } from 'module2';
}
```

3. 模块的整体加载

除了指定加载某个输出值，还可以使用整体加载，即用星号（*）指定一个对象，所有输出值都加载在这个对象上面。

```
// circle.js
export function area(radius) {
  return Math.PI * radius * radius;
}
export function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

```
// main.js
import { area, circumference } from './circle';
console.log('圆面积：' + area(4));
console.log('圆周长：' + circumference(14));
```

上面写法是逐一指定要加载的方法，整体加载的写法如下：

```
import * as circle from './circle';
console.log('圆面积：' + circle.area(4));
console.log('圆周长：' + circle.circumference(14));
```

注意，模块整体加载所在的那个对象（上例是circle），应该是可以静态分析的，所以不允许运行时改变。下面的写法都是不允许的。

```
import * as circle from './circle';

// 下面两行都是不允许的
circle.foo = 'hello';
circle.area = function () {};
```

4. export default

`export default` 命令，为模块指定默认输出。

```
// export-default.js
export default function () {
  console.log('foo');
}
```

其他模块加载该模块时，import命令可以为该匿名函数指定任意名字。

```
// import-default.js
import customName from './export-default';
customName(); // 'foo'
```

上面代码的import命令，可以用任意名称指向export-default.js输出的方法，这时就不需要知道原模块输出的函数名。**这时import命令后面，不使用大括号。**

```
// 第一组
export default function crc32() { // 输出
  // ...
```

```
}

import crc32 from 'crc32'; // 输入

// 第二组
export function crc32() { // 输出
  // ...
};

import {crc32} from 'crc32'; // 输入
```

第一组是使用 `export default` 时，对应的 `import` 语句不需要使用大括号；第二组是不使用 `export default` 时，对应的 `import` 语句需要使用大括号。

```
// modules.js
function add(x, y) {
  return x * y;
}
export {add as default};
// 等同于
// export default add;

// app.js
import { default as foo } from 'modules';
// 等同于
// import foo from 'modules';
```

因为 `export default` 命令其实只是输出一个叫做 `default` 的变量，所以它后面不能跟变量声明语句。

```
// 正确
export var a = 1;

// 正确
var a = 1;
export default a;

// 错误
export default var a = 1;
```

有了 `export default` 命令，输入模块时就非常直观了，以输入 lodash 模块为例：

```
import _ from 'lodash';
```

如果想在一条 `import` 语句中，同时输入默认方法和其他接口，可以写成下面这样：

```
import _, { each, forEach } from 'lodash';
```

5. export 与 import 的复合写法

如果在一个模块之中，先输入后输出同一个模块，`import` 语句可以与 `export` 语句写在一起。

```
export { foo, bar } from 'my_module';

// 可以简单理解为
import { foo, bar } from 'my_module';
export { foo, bar };
```

`export` 和 `import` 语句可以结合在一起，写成一行。但需要注意的是，写成一行以后，`foo` 和 `bar` 实际上并没有被导入当前模块，只是相当于对外转发了这两个接口，导致当前模块不能直接使用 `foo` 和 `bar`。

模块的接口改名和整体输出，也可以采用这种写法。

```
// 接口改名
export { foo as myFoo } from 'my_module';

// 整体输出
export * from 'my_module';
```

```
export * as ns from "mod";

// 等同于
import * as ns from "mod";
export {ns};
```

7. import() 函数

import()返回一个 Promise 对象。

7.1. 按需加载

```
button.addEventListener('click', event => {
  import('./dialogBox.js')
  .then(dialogBox => {
    dialogBox.open();
  })
  .catch(error => {
    /* Error handling */
```

```
  })
});
```

只有用户点击了按钮，才会加载这个模块。

7.2. 条件加载

```
if (condition) {
  import('moduleA').then(...);
} else {
  import('moduleB').then(...);
}
```

放在if代码块，根据不同的情况，加载不同的模块。

7.3. 动态的模块路径

```
import(f()).then(...);
```

根据函数 `f` 的返回结果，加载不同的模块。

8. 加载规则

浏览器加载 ES6 模块，也使用 `<script>` 标签，但是要加入type="module"属性。

```
<script type="module" src="./foo.js"></script>
```

浏览器对于带有 `type="module"` 的 `<script>`，都是异步加载，不会造成堵塞浏览器，即等到整个页面渲染完，再执行模块脚本，等同于打开了 `<script>` 标签的 `defer` 属性。

```
<script type="module" src="./foo.js"></script>
<!-- 等同于 -->
<script type="module" src="./foo.js" defer></script>
```

9. package.json 的 main 字段

`package.json` 文件有两个字段可以指定模块的入口文件：`main` 和 `exports`。比较简单的模块，可以只使用 `main` 字段，指定模块加载的入口文件。

```
// ./node_modules/es-module-package/package.json
{
  "type": "module",
```

```
  "main": "./src/index.js"
}
```

上面代码指定项目的入口脚本为 `./src/index.js`，它的格式为 ES6 模块。如果没有 `type` 字段，`index.js` 就会被解释为 `CommonJS` 模块。

然后，`import` 命令就可以加载这个模块。

```
// ./my-app.mjs
import { something } from 'es-module-package';
// 实际加载的是 ./node_modules/es-module-package/src/index.js
```

上面代码中，运行该脚本以后，`Node.js` 就会到 `./node_modules` 目录下面，寻找 `es-module-package` 模块，然后根据该模块 `package.json` 的 `main` 字段去执行入口文件。