
Broadview
ECMAScript 6 Primer
ECMAScript 6
入门
编制：vueres.cn Vue.js研发群165862199
阮一峰 著

ECMAScript 6简介

ECMAScript 6（以下简称ES6）是JavaScript语言的下一代标准，已经在2015年6月正式发布了。Mozilla公司将在这个标准的基础上，推出JavaScript 2.0。

ES6的目标，是使得JavaScript语言可以用来编写大型的复杂的应用程序，成为企业级开发语言。

标准的制定者有计划，以后每年发布一次标准，使用年份作为标准的版本。因为当前版本的ES6是在2015年发布的，所以又称ECMAScript 2015。

ECMAScript和JavaScript的关系

很多初学者感到困惑：ECMAScript和JavaScript到底是什么关系？

简单说，ECMAScript是JavaScript语言的国际标准，JavaScript是ECMAScript的实现。

要讲清楚这个问题，需要回顾历史。1996年11月，JavaScript的创造者Netscape公司，决定将JavaScript提交给国际标准化组织ECMA，希望这种语言能够成为国际标准。次年，ECMA发布262号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为ECMAScript。这个版本就是ECMAScript 1.0版。

之所以不叫JavaScript，有两个原因。一是商标，Java是Sun公司的商标，根据授权协议，只有Netscape公司可以合法地使用JavaScript这个名字，且JavaScript本身也已经被Netscape公司注册为商标。二是想体现这门语言的制定者是ECMA，不是Netscape，这样有利于保证这门语言的开放性和中立性。因此，ECMAScript和JavaScript的关系是，前者是后者的规格，后者是前者的一种实现。在日常场合，这两个词是可以互换的。

ECMAScript的历史

1998年6月，ECMAScript 2.0版发布。

1999年12月，ECMAScript 3.0版发布，成为JavaScript的通行标准，得到了广泛支持。

2007年10月，ECMAScript 4.0版草案发布，对3.0版做了大幅升级，预计次年8月发布正式版本。草案发布后，由于4.0版的目标过于激进，各方对于是否通过这个标准，发生了严重分歧。以Yahoo、Microsoft、Google为首的大公司，反对JavaScript的大幅升级，主张小幅改动；以JavaScript创造者Brendan Eich为首的Mozilla公司，则坚持当前的草案。

2008年7月，由于对于下一个版本应该包括哪些功能，各方分歧太大，争论过于激进，ECMA开会决定，中止ECMAScript 4.0的开发，将其中涉及现有功能改善的一小部分，发布为ECMAScript 3.1，而将其他激进的设想扩大范围，放入以后的版本，由于会议的气氛，该版本的项目代号起名为Harmony（和谐）。会后不久，ECMAScript 3.1就改名为ECMAScript 5。

2009年12月，ECMAScript 5.0版正式发布。Harmony项目则一分为二，一些较为可行的设想定名为JavaScript.next继续开发，后来演变成ECMAScript 6；一些不是很成熟的设想，则被视为JavaScript.next.next，在更远的将来再考虑推出。

2011年6月，ECMAscript 5.1版发布，并且成为ISO国际标准（ISO/IEC 16262:2011）。

2013年3月，ECMAScript 6草案冻结，不再添加新功能。新的功能设想将被放到ECMAScript 7。

2013年12月，ECMAScript 6草案发布。然后是12个月的讨论期，听取各方反馈。

2015年6月，ECMAScript 6正式通过，成为国际标准。

ECMA的第39号技术专家委员会（Technical Committee 39，简称TC39）负责制订ECMAScript标准，成员包括Microsoft、Mozilla、Google等大公司。TC39的总体考虑是，ES5与ES3基本保持兼容，较大的语法修正和新功能加入，将由JavaScript.next完成。当时，JavaScript.next指的是ES6，第六版发布以后，就指ES7。TC39的判断是，ES5会在2013年的年中成为JavaScript开发的主流标准，并在此后五年中一直保持这个位置。

部署进度

各大浏览器的最新版本，对ES6的支持可以查看kangax.github.io/es5-compat-table/es6/。随着时间的推移，支持度已经越来越高了，ES6的大部分特性都实现了。

Node.js和io.js（一个部署新功能更快的Node分支）是JavaScript语言的服务器运行环境。它们对ES6的支持度，比浏览器更高。通过它们，可以体验更多ES6的特性。

建议使用版本管理工具nvm，来安装Node.js和io.js。不过，nvm不支持Windows系统，下面的操作可以改用nvmw或nvm-windows代替。

安装nvm需要打开命令行窗口，运行下面的命令。

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/<version number>/install.sh | bash
```

上面命令的version number处，需要用版本号替换。本书写作时的版本号是v0.25.4。

该命令运行后，nvm会默认安装在用户主目录的`.nvm`子目录。然后，激活nvm。

```
$ source ~/.nvm/nvm.sh
```

激活以后，安装Node或io.js的最新版。

```
$ nvm install node
# 或
$ nvm install iojs
```

安装完成后，就可以在各种版本的node之间自由切换。

```
# 切换到node
$ nvm use node

# 切换到iojs
$ nvm use iojs
```

需要注意的是，Node.js对ES6的支持，需要打开harmony参数，iojs不需要。

```
$ node --harmony
# iojs不需要打开harmony参数
$ node
```

上面命令执行后，就会进入REPL环境，该环境支持所有已经实现的ES6特性。

使用下面的命令，可以查看Node.js所有已经实现的ES6特性。

```
$ node --v8-options | grep harmony

  --harmony_typeof
  --harmony_scoping
  --harmony_modules
  --harmony_symbols
  --harmony_proxies
  --harmony_collections
  --harmony_observation
  --harmony_generators
  --harmony_iteration
  --harmony_numeric_literals
  --harmony_strings
  --harmony_arrays
  --harmony_maths
  --harmony
```

上面命令的输出结果，会因为版本的不同而有所不同。

我写了一个ES-Checker模块，用来检查各种运行环境对ES6的支持情况。访问ruanyf.github.io/es-checker，可以看到您的浏览器支持ES6的程度。运行下面的命令，可以查看本机支持ES6的程度。

```
$ npm install -g es-checker
$ es-checker
```

Babel转码器

Babel是一个广泛使用的ES6转码器，可以ES6代码转为ES5代码，从而在浏览器或其他环境执行。这意味着，你可以用ES6的方式编写程序，又不用担心现有环境是否支持。下面是一个例子。

```
// 转码前
input.map(item => item + 1);

// 转码后
input.map(function (item) {
  return item + 1;
});
```

上面的原始代码用了箭头函数，这个特性还没有得到广泛支持，Babel将其转为普通函数，就能在现有的JavaScript环境执行了。

它的安装命令如下。

```
$ npm install --global babel
```

Babel自带一个`babel-node`命令，提供支持ES6的REPL环境。它支持Node的REPL环境的所有功能，而且可以直接运行ES6代码。

```
$ babel-node
>
> console.log([1,2,3].map(x => x * x))
  [ 1, 4, 9 ]
>
```

`babel-node`命令也可以直接运行ES6脚本。假定将上面的代码放入脚本文件`es6.js`。

```
$ babel-node es6.js
[1, 4, 9]
```

babel命令可以将ES6代码转为ES5代码。

```
$ babel es6.js
"use strict";

console.log([1, 2, 3].map(function (x) {
  return x * x;
}));
```

`-o`参数将转换后的代码，从标准输出导入文件。

```
$ babel es6.js -o es5.js
# 或者
$ babel es6.js --out-file es5.js
```

`-d`参数用于转换整个目录。

```
$ babel -d build-dir source-dir
```

注意，`-d`参数后面跟的是输出目录。

如果希望生成source map文件，则要加上`-s`参数。

```
$ babel -d build-dir source-dir -s
```

Babel也可以用于浏览器。

```
<script src="node_modules/babel-core/browser.js">

// Your ES6 code
```

上面代码中，`browser.js`是Babel提供的转换器脚本，可以在浏览器运行。用户的ES6脚本放在script标签之中，但是要注明`type="text/babel"`。

Babel配合Browserify一起使用，可以生成浏览器能够直接加载的脚本。

```
$ browserify script.js -t babelify --outfile bundle.js
```

Traceur转码器

Google公司的Traceur转码器，也可以将ES6代码转为ES5代码。

直接插入网页

Traceur允许将ES6代码直接插入网页。首先，必须在网页头部加载Traceur库文件。

```
traceur.options.experimental = true;
```

接下来，就可以把ES6代码放入上面这些代码的下方。

```
class Calc {
  constructor(){
    console.log('Calc constructor');
  }
  add(a, b){
    return a + b;
  }
}

var c = new Calc();
console.log(c.add(4,5));
```

正常情况下，上面代码会在控制台打印出9。

注意，`script`标签的`type`属性的值是`module`，而不是`text/javascript`。这是Traceur编译器识别ES6代码的标识，编译器会自动将所有`type=module`的代码编译为ES5，然后再交给浏览器执行。

如果ES6代码是一个外部文件，也可以用`script`标签插入网页。

```
```

在线转换

Traceur提供一个在线编译器，可以在线将ES6代码转为ES5代码。转换后的代码，可以直接作为ES5代码插入网页运行。

上面的例子转为ES5代码运行，就是下面这个样子。

```
        traceur.options.experimental = true;


$traceurRuntime.ModuleStore.getAnonymousModule(function() {
  "use strict";

  var Calc = function Calc() {
    console.log('Calc constructor');
  };

  ($traceurRuntime.createClass)(Calc, {add: function(a, b) {
    return a + b;
  }}, {});

  var c = new Calc();
  console.log(c.add(4, 5));
  return {};
});
```

命令行转换

作为命令行工具使用时，Traceur是一个Node.js的模块，首先需要用npm安装。

```
$ npm install -g traceur
```

安装成功后，就可以在命令行下使用traceur了。

traceur直接运行es6脚本文件，会在标准输出显示运行结果，以前面的calc.js为例。

```
$ traceur calc.js
Calc constructor
9
```

如果要将ES6脚本转为ES5保存，要采用下面的写法

```
$ traceur --script calc.es6.js --out calc.es5.js
```

上面代码的`--script`选项表示指定输入文件，`--out`选项表示指定输出文件。

为了防止有些特性编译不成功，最好加上`--experimental`选项。

```
$ traceur --script calc.es6.js --out calc.es5.js --experimental
```

命令行下转换的文件，就可以放到浏览器中运行。

Node.js环境的用法

Traceur的Node.js用法如下（假定已安装traceur模块）。

```
var traceur = require('traceur');
var fs = require('fs');

// 将ES6脚本转为字符串
var contents = fs.readFileSync('es6-file.js').toString();

var result = traceur.compile(contents, {
  filename: 'es6-file.js',
  sourceMap: true,
  // 其他设置
  modules: 'commonjs'
});

if (result.error)
  throw result.error;

// result对象的js属性就是转换后的ES5代码
fs.writeFileSync('out.js', result.js);
// sourceMap属性对应map文件
fs.writeFileSync('out.js.map', result.sourceMap);
```

ECMAScript 7

2013年3月，ES6的草案封闭，不再接受新功能了。新的功能将被加入ES7。

ES7可能包括的功能有：

（1）**Object.observe**：用来监听对象（以及数组）的变化。一旦监听对象发生变化，就会触发回调函数。

（2）**Async函数**：在Promise和Generator函数基础上，提出的异步操作解决方案。

（3）**Multi-Threading**：多线程支持。目前，Intel和Mozilla有一个共同的研究项目RiverTrail，致力于让JavaScript多线程运行。预计这个项目的研究成果会被纳入ECMAScript标准。

（4）**Traits**：它将是“类”功能（class）的一个替代。通过它，不同的对象可以分享同样的特性。

其他可能包括的功能还有：更精确的数值计算、改善的内存回收、增强的跨站点安全、类型化的更贴近硬件的低级别操作、国际化支持（Internationalization Support）、更多的数据结构等等。

本书对于那些明确的、或者很有希望列入ES7的功能，尤其是那些Babel已经支持的功能，都将予以介绍。

let和const命令

let命令

基本用法

ES6新增了let命令，用来声明变量。它的用法类似于var，但是所声明的变量，只在let命令所在的代码块内有效。

```
{
  let a = 10;
  var b = 1;
}

a // ReferenceError: a is not defined.
b // 1
```

上面代码在代码块之中，分别用let和var声明了两个变量。然后在代码块之外调用这两个变量，结果let声明的变量报错，var声明的变量返回了正确的值。这表明，let声明的变量只在它所在的代码块有效。

for循环的计数器，就很合适使用let命令。

```
for(let i = 0; i < arr.length; i++){}

console.log(i)
//ReferenceError: i is not defined
```

上面代码的计数器i，只在for循环体内有效。

下面的代码如果使用var，最后输出的是10。

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

不存在变量提升

let不像var那样，会发生“变量提升”现象。

```
function do_something() {
  console.log(foo); // ReferenceError
  let foo = 2;
}
```

上面代码在声明foo之前，就使用这个变量，结果会抛出一个错误。

这也意味着typeof不再是一个百分之百安全的操作。

```
if (1) {
  typeof x; // ReferenceError
  let x;
}
```

上面代码中，由于块级作用域内typeof运行时，x还没有值，所以会抛出一个ReferenceError。

只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

```
var tmp = 123;

if (true) {
  tmp = 'abc'; // ReferenceError
  let tmp;
}
```

上面代码中，存在全局变量tmp，但是块级作用域内let又声明了一个局部变量tmp，导致后者绑定这个块级作用域，所以在let声明变量前，对tmp赋值会报错。

ES6明确规定，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命令，就会报错。

总之，在代码块内，使用let命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死

区”（temporal dead zone，简称TDZ）。

```
if (true) {
  // TDZ开始
  tmp = 'abc'; // ReferenceError
  console.log(tmp); // ReferenceError

  let tmp; // TDZ结束
  console.log(tmp); // undefined

  tmp = 123;
  console.log(tmp); // 123
}
```

上面代码中，在let命令声明变量tmp之前，都属于变量tmp的“死区”。

有些“死区”比较隐蔽，不太容易发现。

```
function bar(x=y, y=2) {
  return [x, y];
}

bar(); // 报错
```

上面代码中，调用bar函数之所以报错，是因为参数x默认值等于另一个参数y，而此时y还没有声明，属于”死区“。

需要注意的是，函数的作用域是其声明时所在的作用域。如果函数A的参数是函数B，那么函数B的作用域不是函数A。

```
let foo = 'outer';

function bar(func = x => foo) {
  let foo = 'inner';
  console.log(func()); // outer
}

bar();
```

上面代码中，函数bar的参数func，默认是一个匿名函数，返回值为变量foo。这个匿名函数的作用域就不是bar。这个匿名函数声明时，是处在外层作用域，所以内部的foo指向函数体外的声明，输出outer。它实际上等同于下面的代码。

```
let foo = 'outer';
let f = x => foo;

function bar(func = f) {
  let foo = 'inner';
  console.log(func()); // outer
}

bar();
```

不允许重复声明

let不允许在相同作用域内，重复声明同一个变量。

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

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

因此，不能在函数内部重新声明参数。

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

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

块级作用域

let实际上为JavaScript新增了块级作用域。

```
function f1() {
  let n = 5;
  if (true) {
     let n = 10;
  }
  console.log(n); // 5
}
```

上面的函数有两个代码块，都声明了变量n，运行后输出5。这表示外层代码块不受内层代码块的影响。如果使用var定义变量n，最后输出的值就是10。

块级作用域的出现，实际上使得获得广泛应用的立即执行匿名函数（IIFE）不再必要了。

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

// 块级作用域写法
{
  let tmp = ...;
  ...
}
```

另外，ES6也规定，函数本身的作用域，在其所在的块级作用域之内。

```
function f() { console.log('I am outside!'); }
(function () {
  if(false) {
    // 重复声明一次函数f
    function f() { console.log('I am inside!'); }
  }

  f();
}());
```

上面代码在ES5中运行，会得到“I am inside!”，但是在ES6中运行，会得到“I am outside!”。这是因为ES5存在函数提升，不管会不会进入if代码块，函数声明都会提升到当前作用域的顶部，得到执行；而ES6支持块级作用域，不管会不会进入if代码块，其内部声明的函数皆不会影响到作用域的外部。

需要注意的是，如果在严格模式下，函数只能在顶层作用域和函数内声明，其他情况（比如if代码块、循环代码块）的声明都会报错。

const命令

const也用来声明变量，但是声明的是常量。一旦声明，常量的值就不能改变。

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

PI = 3;
PI // 3.1415

const PI = 3.1;
PI // 3.1415
```

上面代码表明改变常量的值是不起作用的。需要注意的是，对常量重新赋值不会报错，只会默默地失败。

const的作用域与let命令相同：只在声明所在的块级作用域内有效。

```
if (true) {
  const MAX = 5;
}

// 常量MAX在此处不可得
```

const命令也不存在提升，只能在声明的位置后面使用。

```
if (true) {
  console.log(MAX); // ReferenceError
  const MAX = 5;
}
```

上面代码在常量MAX声明之前就调用，结果报错。

const声明的常量，也与let一样不可重复声明。

```
var message = "Hello!";
let age = 25;

// 以下两行都会报错
const message = "Goodbye!";
const age = 30;
```

由于const命令只是指向变量所在的地址，所以将一个对象声明为常量必须非常小心。

```
const foo = {};
foo.prop = 123;

foo.prop
// 123

foo = {} // 不起作用
```

上面代码中，常量foo储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把foo指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

下面是另一个例子。

```
const a = [];
a.push("Hello"); // 可执行
a.length = 0;    // 可执行
a = ["Dave"];    // 报错
```

上面代码中，常量a是一个数组，这个数组本身是可写的，但是如果将另一个数组赋值给a，就会报错。

如果真的想将对象冻结，应该使用Object.freeze方法。

```
const foo = Object.freeze({});
foo.prop = 123; // 不起作用
```

上面代码中，常量foo指向一个冻结的对象，所以添加新属性不起作用。

除了将对象本身冻结，对象的属性也应该冻结。下面是一个将对象彻底冻结的函数。

```
var constantize = (obj) => {
  Object.freeze(obj);
  Object.keys(obj).forEach( (key, value) => {
    if ( typeof obj[key] === 'object' ) {
      constantize( obj[key] );
    }
  });
};
```

跨模块常量

上面说过，const声明的常量只在当前代码块有效。如果想设置跨模块的常量，可以采用下面的写法。

```
// constants.js 模块
export const A = 1;
export const B = 3;
export const C = 4;

// test1.js 模块
import * as constants from './constants';
console.log(constants.A); // 1
console.log(constants.B); // 3

// test2.js 模块
import {A, B} from './constants';
console.log(A); // 1
console.log(B); // 3
```

全局对象的属性

全局对象是最顶层的对象，在浏览器环境指的是window对象，在Node.js指的是global对象。在JavaScript语言中，所有全局变量都是全局对象的属性。

ES6规定，var命令和function命令声明的全局变量，属于全局对象的属性；let命令、const命令、class命令声明的全局变量，不属于全局对象的属性。

```
var a = 1;
// 如果在node环境，可以写成global.a
// 或者采用通用方法，写成this.a
window.a // 1

let b = 1;
window.b // undefined
```

上面代码中，全局变量a由var命令声明，所以它是全局对象的属性；全局变量b由let命令声明，所以它不是全局对象的属性，返回undefined。

变量的解构赋值

数组的解构赋值

ES6允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

以前，为变量赋值，只能直接指定值。

```
var a = 1;
var b = 2;
var c = 3;
```

ES6允许写成下面这样。

```
var [a, b, c] = [1, 2, 3];
```

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”，只要等号两边的模式相同，左边的变量就会被赋予对应的值。下面是一些使用嵌套数组进行解构的例子。

```
let [foo, [[bar], baz]] = [1, [[2], 3]];
foo // 1
bar // 2
baz // 3

let [,,third] = ["foo", "bar", "baz"];
third // "baz"

let [x, , y] = [1, 2, 3];
x // 1
y // 3

let [head, ...tail] = [1, 2, 3, 4];
head // 1
tail // [2, 3, 4]
```

如果解构不成功，变量的值就等于undefined。

```
var [foo] = [];
var [foo] = 1;
var [foo] = false;
var [foo] = NaN;
var [bar, foo] = [1];
```

以上几种情况都属于解构不成功，foo的值都会等于undefined。这是因为原始类型的值，会自动转为对象，比如数值1转为 `new Number(1)` ，从而导致foo取到undefined。

另一种情况是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。这种情况下，解构依然可以成功。

```
let [x, y] = [1, 2, 3];
x // 1
y // 2

let [a, [b], d] = [1, [2, 3], 4];
a // 1
b // 2
d // 4
```

上面代码的两个例子，都属于不完全解构，但是可以成功。

如果对undefined或null进行解构，会报错。

```
// 报错
let [foo] = undefined;
let [foo] = null;
```

这是因为解构只能用于数组或对象。其他原始类型的值都可以转为相应的对象，但是，undefined和null不能转为对象，因此报错。

解构赋值允许指定默认值。

```
var [foo = true] = [];
foo // true

[x, y='b'] = ['a'] // x='a', y='b'
[x, y='b'] = ['a', undefined] // x='a', y='b'
```

注意，ES6内部使用严格相等运算符（===），判断一个位置是否有值。所以，如果一个数组成员不严格等于undefined，默认值是不会生效的。

```
var [x = 1] = [undefined];
x // 1

var [x = 1] = [null];
x // null
```

上面代码中，如果一个数组成员是null，默认值就不会生效，因为null不严格等于undefined。

解构赋值不仅适用于var命令，也适用于let和const命令。

```
var [v1, v2, ..., vN ] = array;
let [v1, v2, ..., vN ] = array;
const [v1, v2, ..., vN ] = array;
```

对于Set结构，也可以使用数组的解构赋值。

```
[a, b, c] = new Set(["a", "b", "c"])
a // "a"
```

事实上，只要某种数据结构具有Iterator接口，都可以采用数组形式的解构赋值。

```
function* fibs() {
  var a = 0;
  var b = 1;
  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

var [first, second, third, fourth, fifth, sixth] = fibs();
sixth // 5
```

上面代码中，fibs是一个Generator函数，原生具有Iterator接口。解构赋值会依次从这个接口获取值。

对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

```
var { foo, bar } = { foo: "aaa", bar: "bbb" };
foo // "aaa"
bar // "bbb"
```

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

```
var { bar, foo } = { foo: "aaa", bar: "bbb" };
foo // "aaa"
bar // "bbb"

var { baz } = { foo: "aaa", bar: "bbb" };
baz // undefined
```

上面代码的第一个例子，等号左边的两个变量的次序，与等号右边两个同名属性的次序不一致，但是对取值完全没有影响。第二个例子的变量没有对应的同名属性，导致取不到值，最后等于`undefined`。

如果变量名与属性名不一致，必须写成下面这样。

```
var { foo: baz } = { foo: "aaa", bar: "bbb" };
baz // "aaa"

let obj = { first: 'hello', last: 'world' };
let { first: f, last: l } = obj;
f // 'hello'
l // 'world'
```

和数组一样，解构也可以用于嵌套结构的对象。

```
var obj = {
  p: [
    "Hello",
    { y: "World" }
  ]
};

var { p: [x, { y }] } = obj;
x // "Hello"
y // "World"
```

对象的解构也可以指定默认值。

```
var {x = 3} = {};
x // 3

var {x, y = 5} = {x: 1};
console.log(x, y) // 1, 5

var { message: msg = "Something went wrong" } = {};
console.log(msg); // "Something went wrong"
```

默认值生效的条件是，对象的属性值严格等于undefined。

```
var {x = 3} = {x: undefined};
x // 3

var {x = 3} = {x: null};
x // null
```

上面代码中，如果x属性等于null，就不严格相等于undefined，导致默认值不会生效。

如果要将一个已经声明的变量用于解构赋值，必须非常小心。

```
// 错误的写法

var x;
{x} = {x:1};
// SyntaxError: syntax error
```

上面代码的写法会报错，因为JavaScript引擎会将`{x}`理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免JavaScript将其解释为代码块，才能解决这个问题。

```
// 正确的写法
({x} = {x:1});
```

上面代码将整个解构赋值语句，放在一个圆括号里面，就可以正确执行。关于圆括号与解构赋值的关系，参见下文。

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量。

```
let { log, sin, cos } = Math;
```

上面代码将Math对象的对数、正弦、余弦三个方法，赋值到对应的变量上，使用起来就会方便很多。

字符串的解构赋值

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

```
const [a, b, c, d, e] = 'hello';
a // "h"
b // "e"
c // "l"
d // "l"
e // "o"
```

类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值。

```
let {length : len} = 'hello';
len // 5
```

函数参数的解构赋值

函数的参数也可以使用解构。

```
function add([x, y]){
  return x + y;
}

add([1, 2]) // 3
```

上面代码中，函数add的参数实际上不是一个数组，而是通过解构得到的变量x和y。

函数参数的解构也可以使用默认值。

```
function move({x = 0, y = 0} = {}) {
  return [x, y];
}

move({x: 3, y: 8}); // [3, 8]
move({x: 3}); // [3, 0]
move({}); // [0, 0]
move(); // [0, 0]
```

上面代码中，函数move的参数是一个对象，通过对这个对象进行解构，得到变量x和y的值。如果解构失败，x和y等于默认值。

注意，指定函数参数的默认值时，不能采用下面的写法。

```
function move({x, y} = { x: 0, y: 0 }) {
  return [x, y];
}

move({x: 3, y: 8}); // [3, 8]
move({x: 3}); // [3, undefined]
move({}); // [undefined, undefined]
move(); // [0, 0]
```

上面代码是为函数move的参数指定默认值，而不是为变量x和y指定默认值，所以会得到与前一种写法不同的结果。

圆括号问题

解构赋值虽然很方便，但是解析起来并不容易。对于编译器来说，一个式子到底是模式，还是表达式，没有办法从一开始就知道，必须解析到（或解析不到）等号才能知道。

由此带来的问题是，如果模式中出现圆括号怎么处理。ES6的规则是，只要有可能导致解构的歧义，就不得使用圆括号。

但是，这条规则实际上不那么容易辨别，处理起来相当麻烦。因此，建议只要有可能，就不要在模式中放置圆括号。

不能使用圆括号的情况

以下三种解构赋值不得使用圆括号。

（1）变量声明语句中，模式不能带有圆括号。

```
// 全部报错
var [(a)] = [1];
var { x: (c) } = {};
var { o: ({ p: p }) } = { o: { p: 2 } };
```

上面三个语句都会报错，因为它们都是变量声明语句，模式不能使用圆括号。

（2）函数参数中，模式不能带有圆括号。

函数参数也属于变量声明，因此不能带有圆括号。

```
// 报错
function f([(z)]) { return z; }
```

（3）不能将整个模式，或嵌套模式中的一层，放在圆括号之中。

```
// 全部报错
({ p: a }) = { p: 42 };
([a]) = [5];
```

上面代码将整个模式放在模式之中，导致报错。

```
// 报错
[({ p: a }), { x: c }] = [{}, {}];
```

上面代码将嵌套模式的一层，放在圆括号之中，导致报错。

可以使用圆括号的情况

可以使用圆括号的情况只有一种：赋值语句的非模式部分，可以使用圆括号。

```
[(b)] = [3]; // 正确
({ p: (d) } = {}); // 正确
[(parseInt.prop)] = [3]; // 正确
```

上面三行语句都可以正确执行，因为首先它们都是赋值语句，而不是声明语句；其次它们的圆括号都不属于模式的一部分。第一行语句中，模式是取数组的第一个成员，跟圆括号无关；第二行语句中，模式是p，而不是d；第三行语句与第一行语句的性质一致。

用途

变量的解构赋值用途很多。

（1）交换变量的值

```
[x, y] = [y, x];
```

上面代码交换变量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];
}
var [a, b, c] = example();

// 返回一个对象

function example() {
  return {
    foo: 1,
    bar: 2
  };
}
var { 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({x:1, y:2, z:3})
```

（4）提取JSON数据

解构赋值对提取JSON对象中的数据，尤其有用。

```
var jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
}

let { id, status, data: number } = jsonData;

console.log(id, status, number)
// 42, OK, [867, 5309]
```

上面代码可以快速提取JSON数据的值。

（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。Map结构原生支持Iterator接口，配合变量的解构赋值，获取键名和键值就非常方便。

```
var 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，往往需要指定输入那些方法。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

ES6加强了对Unicode的支持，并且扩展了字符串对象。

codePointAt()

JavaScript内部，字符以UTF-16的格式储存，每个字符固定为2个字节。对于那些需要4个字节储存的字符（Unicode码点大于0xFFFF的字符），JavaScript会认为它们是两个字符。

~~~
var s = "
~~~

数值的扩展

二进制和八进制表示法

ES6提供了二进制和八进制数值的新的写法，分别用前缀0b和0o表示。

```
0b111110111 === 503 // true
0o767 === 503 // true
```

八进制不再允许使用前缀0表示，而改为使用前缀0o。

```
011 === 9 // 不正确
0o11 === 9 // 正确
```

Number.isFinite(), Number.isNaN()

ES6在Number对象上，新提供了Number.isFinite()和Number.isNaN()两个方法，用来检查Infinite和NaN这两个特殊值。

Number.isFinite()用来检查一个数值是否非无穷（infinity）。

```
Number.isFinite(15); // true
Number.isFinite(0.8); // true
Number.isFinite(NaN); // false
Number.isFinite(Infinity); // false
Number.isFinite(-Infinity); // false
Number.isFinite("foo"); // false
Number.isFinite("15"); // false
Number.isFinite(true); // false
```

ES5通过下面的代码，部署Number.isFinite方法。

```
(function (global) {
  var global_isFinite = global.isFinite;

  Object.defineProperty(Number, 'isFinite', {
    value: function isFinite(value) {
      return typeof value === 'number' && global_isFinite(value);
    },
    configurable: true,
    enumerable: false,
    writable: true
  });
})(this);
```

Number.isNaN()用来检查一个值是否为NaN。

```
Number.isNaN(NaN); // true
Number.isNaN(15); // false
Number.isNaN("15"); // false
Number.isNaN(true); // false
```

ES5通过下面的代码，部署Number.isNaN()。

```
(function (global) {
  var global_isNaN = global.isNaN;

  Object.defineProperty(Number, 'isNaN', {
    value: function isNaN(value) {
      return typeof value === 'number' && global_isNaN(value);
    },
    configurable: true,
    enumerable: false,
    writable: true
  });
})(this);
```

它们与传统的全局方法isFinite()和isNaN()的区别在于，传统方法先调用Number()将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，非数值一律返回false。

```
isFinite(25) // true
isFinite("25") // true
Number.isFinite(25) // true
Number.isFinite("25") // false

isNaN(NaN) // true
isNaN("NaN") // true
Number.isNaN(NaN) // true
Number.isNaN("NaN") // false
```

Number.parseInt(), Number.parseFloat()

ES6将全局方法parseInt()和parseFloat()，移植到Number对象上面，行为完全保持不变。

```
// ES5的写法
parseInt("12.34") // 12
parseFloat('123.45#') // 123.45

// ES6的写法
Number.parseInt("12.34") // 12
Number.parseFloat('123.45#') // 123.45
```

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

Number.isInteger()和安全整数

Number.isInteger()用来判断一个值是否为整数。需要注意的是，在JavaScript内部，整数和浮点数是同样的储存方法，所以3和3.0被视为同一个值。

```
Number.isInteger(25) // true
Number.isInteger(25.0) // true
Number.isInteger(25.1) // false
Number.isInteger("15") // false
Number.isInteger(true) // false
```

ES5通过下面的代码，部署Number.isInteger()。

```
(function (global) {
  var floor = Math.floor,
    isFinite = global.isFinite;

  Object.defineProperty(Number, 'isInteger', {
    value: function isInteger(value) {
      return typeof value === 'number' && isFinite(value) &&
        value > -9007199254740992 && value < 9007199254740992 &&
        floor(value) === value;
    },
    configurable: true,
    enumerable: false,
    writable: true
  });
})(this);
```

JavaScript能够准确表示的整数范围在-2^53 and 2^53之间。ES6引入了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.isSafeInteger()则是用来判断一个整数是否落在这个范围之内。

```
var inside = Number.MAX_SAFE_INTEGER;
var outside = inside + 1;

Number.isInteger(inside) // true
Number.isSafeInteger(inside) // true

Number.isInteger(outside) // true
Number.isSafeInteger(outside) // false
```

Math对象的扩展

ES6在Math对象上新增了17个与数学相关的方法。所有这些方法都是静态方法，只能在Math对象上调用。

Math.trunc()

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

```
Math.trunc(4.1) // 4
Math.trunc(4.9) // 4
Math.trunc(-4.1) // -4
Math.trunc(-4.9) // -4
```

对于空值和无法截取整数的值，返回NaN。

```
Math.trunc(NaN);      // NaN
Math.trunc('foo');    // NaN
Math.trunc();         // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.trunc = Math.trunc || function(x) {
  return x < 0 ? Math.ceil(x) : Math.floor(x);
}
```

Math.sign()

Math.sign方法用来判断一个数到底是正数、负数、还是零。

它会返回五种值。

- 参数为正数，返回+1；
- 参数为负数，返回-1；
- 参数为0，返回0；
- 参数为-0，返回-0;
- 其他值，返回NaN。

```
Math.sign(-5) // -1
Math.sign(5) // +1
Math.sign(0) // +0
Math.sign(-0) // -0
Math.sign(NaN) // NaN
Math.sign('foo'); // NaN
Math.sign();      // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.sign = Math.sign || function(x) {
  x = +x; // convert to a number
  if (x === 0 || isNaN(x)) {
    return x;
  }
  return x > 0 ? 1 : -1;
}
```

Math.cbrt()

Math cbrt方法用于计算一个数的立方根

```
Math.cbrt(-1); // -1
Math.cbrt(0);  // 0
Math.cbrt(1);  // 1
Math.cbrt(2);  // 1.2599210498948734
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.cbrt = Math.cbrt || function(x) {
  var y = Math.pow(Math.abs(x), 1/3);
  return x < 0 ? -y : y;
};
```

Math.clz32()

JavaScript的整数使用32位二进制形式表示，Math.clz32方法返回一个数的32位无符号整数形式有多少个前导0。

```
Math.clz32(0) // 32
Math.clz32(1) // 31
Math.clz32(1000) // 22
```

上面代码中，0的二进制形式全为0，所以有32个前导0；1的二进制形式是0b1，只占1位，所以32位之中有31个前导0；1000的二进制形式是0b1111101000，一共有10位，所以32位之中有22个前导0。

对于小数，Math.clz32方法只考虑整数部分。

```
Math.clz32(3.2) // 30
Math.clz32(3.9) // 30
```

对于空值或其他类型的值，Math.clz32方法会将它们先转为数值，然后再计算。

```
Math.clz32() // 32
Math.clz32(NaN) // 32
Math.clz32(Infinity) // 32
Math.clz32(null) // 32
Math.clz32('foo') // 32
Math.clz32([]) // 32
Math.clz32({}) // 32
Math.clz32(true) // 31
```

Math.imul()

Math.imul方法返回两个数以32位带符号整数形式相乘的结果，返回的也是一个32位的带符号整数。

```
Math.imul(2, 4);         // 8
Math.imul(-1, 8);        // -8
Math.imul(-2, -2);       // 4
```

如果只考虑最后32位（含第一个整数位），大多数情况下，`Math.imul(a, b)`与`a * b`的结果是相同的，即该方法等同于`(a * b)|0`的效果。之所以需要部署这个方法，是因为JavaScript有精度限制，超过2的53次方的值无法精确表示。这就是说，对于那些很大的数的乘法，低位数值往往都是不精确的，Math.imul方法可以返回正确的低位数值。

```
(0x7fffffff * 0x7fffffff)|0 // 0
```

上面这个乘法算式，返回结果为0。但是由于这两个数的个位数都是1，所以这个结果肯定是不正确的。这个错误就是因为它们的乘积超过了2的53次方，JavaScript无法保存额外的精度，就把低位的值都变成了0。Math.imul方法可以返回正确的值1。

```
Math.imul(0x7fffffff, 0x7fffffff) // 1
```

Math.fround()

Math.fround方法返回一个数的单精度浮点数形式。

```
Math.fround(0);     // 0
Math.fround(1);     // 1
Math.fround(1.337); // 1.3370000123977661
Math.fround(1.5);   // 1.5
Math.fround(NaN);   // NaN
```

对于整数来说，Math.fround方法返回结果不会有任何不同，区别主要是那些无法用64个二进制位精确表示的小数。这时，Math.fround方法会返回最接近这个小数的单精度浮点数。

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.fround = Math.fround || function(x) {
  return new Float32Array([x])[0];
};
```

Math.hypot()

Math.hypot方法返回所有参数的平方和的平方根。

```
Math.hypot(3, 4);        // 5
Math.hypot(3, 4, 5);     // 7.0710678118654755
Math.hypot();            // 0
Math.hypot(NaN);         // NaN
Math.hypot(3, 4, 'foo'); // NaN
Math.hypot(3, 4, '5');   // 7.0710678118654755
Math.hypot(-3);          // 3
```

上面代码中，3的平方加上4的平方，等于5的平方。

如果参数不是数值，Math.hypot方法会将其转为数值。只要有一个参数无法转为数值，就会返回NaN。

对数方法

ES6新增了4个对数相关方法。

（1）Math.expm1()

`Math.expm1(x)` 返回ex - 1。

```
Math.expm1(-1); // -0.6321205588285577
Math.expm1(0);  // 0
Math.expm1(1);  // 1.718281828459045
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.expm1 = Math.expm1 || function(x) {
  return Math.exp(x) - 1;
};
```

（2）Math.log1p()

`Math.log1p(x)` 方法返回1 + x的自然对数。如果x小于-1，返回NaN。

```
Math.log1p(1);  // 0.6931471805599453
Math.log1p(0);  // 0
Math.log1p(-1); // -Infinity
Math.log1p(-2); // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.log1p = Math.log1p || function(x) {
  return Math.log(1 + x);
};
```

（3）Math.log10()

Math.log10(x) 返回以10为底的x的对数。如果x小于0，则返回NaN。

```
Math.log10(2);      // 0.3010299956639812
Math.log10(1);      // 0
Math.log10(0);      // -Infinity
Math.log10(-2);     // NaN
Math.log10(100000); // 5
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.log10 = Math.log10 || function(x) {
  return Math.log(x) / Math.LN10;
};
```

（4）Math.log2()

Math.log2(x) 返回以2为底的x的对数。如果x小于0，则返回NaN。

```
Math.log2(3);    // 1.584962500721156
Math.log2(2);    // 1
Math.log2(1);    // 0
Math.log2(0);    // -Infinity
Math.log2(-2);   // NaN
Math.log2(1024); // 10
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

```
Math.log2 = Math.log2 || function(x) {
  return Math.log(x) / Math.LN2;
};
```

三角函数方法

ES6新增了6个三角函数方法。

- Math.sinh(x) 返回x的双曲正弦（hyperbolic sine）
- Math.cosh(x) 返回x的双曲余弦（hyperbolic cosine）
- Math.tanh(x) 返回x的双曲正切（hyperbolic tangent）
- Math.asinh(x) 返回x的反双曲正弦（inverse hyperbolic sine）
- Math.acosh(x) 返回x的反双曲余弦（inverse hyperbolic cosine）
- Math.atanh(x) 返回x的反双曲正切（inverse hyperbolic tangent）

数组的扩展

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）。

```
let ps = document.querySelectorAll('p');

Array.from(ps).forEach(function (p) {
  console.log(p);
});
```

上面代码中，querySelectorAll方法返回的是一个类似数组的对象，只有将这个对象转为真正的数组，才能使用forEach方法。

Array.from方法可以将函数的arguments对象，转为数组。

```
function foo() {
  var args = Array.from( arguments );
}

foo( "a", "b", "c" );
```

任何有length属性的对象，都可以通过Array.from方法转为数组。

```
Array.from({ 0: "a", 1: "b", 2: "c", length: 3 });
// [ "a", "b" , "c" ]
```

对于还没有部署该方法的浏览器，可以用Array.prototyp.slice方法替代。

```
const toArray = (() =>
  Array.from ? Array.from : obj => [].slice.call(obj)
)();
```

Array.from()还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理。

```
Array.from(arrayLike, x => x * x);
// 等同于
Array.from(arrayLike).map(x => x * x);
```

下面的例子将数组中布尔值为false的成员转为0。

```
Array.from([1, , 2, , 3], (n) => n || 0)
// [1, 0, 2, 0, 3]
```

Array.from()的一个应用是，将字符串转为数组，然后返回字符串的长度。这样可以避免JavaScript将大于\uFFFF的Unicode字符，算作两个字符的bug。

```
function countSymbols(string) {
  return Array.from(string).length;
}
```

Array.of()

Array.of方法用于将一组值，转换为数组。

```
Array.of(3, 11, 8) // [3,11,8]
Array.of(3) // [3]
Array.of(3).length // 1
```

这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。

```
Array() // []
Array(3) // [undefined, undefined, undefined]
Array(3,11,8) // [3, 11, 8]
```

上面代码说明，只有当参数个数不少于2个，Array()才会返回由参数组成的新数组。

Array.of方法可以用下面的代码模拟实现。

```
function ArrayOf(){
  return [].slice.call(arguments);
}
```

数组实例的find()和findIndex()

数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。

```
var found = [1, 4, -5, 10].find((n) => n < 0);
console.log("found:", found);
```

上面代码找出数组中第一个小于0的成员。

```
[1, 5, 10, 15].find(function(value, index, arr) {
  return value > 9;
}) // 10
```

上面代码中，find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组。

数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

```
[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 9;
}) // 2
```

这两个方法都可以接受第二个参数，用来绑定回调函数的this对象。

另外，这两个方法都可以发现NaN，弥补了数组的IndexOf方法的不足。

```
[NaN].indexOf(NaN)
// -1

[NaN].findIndex(y => Object.is(NaN, y))
// 0
```

上面代码中，indexOf方法无法识别数组的NaN成员，但是findIndex方法可以借助Object.is方法做到。

数组实例的fill()

fill()使用给定值，填充一个数组。

```
['a', 'b', 'c'].fill(7)
// [7, 7, 7]

new Array(3).fill(7)
// [7, 7, 7]
```

上面代码表明，fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去。

fill()还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。

```
['a', 'b', 'c'].fill(7, 1, 2)
// ['a', 7, 'c']
```

数组实例的entries()，keys()和values()

ES6提供三个新的方法——entries()，keys()和values()——用于遍历数组。它们都返回一个遍历器，可以用for...of循环进行遍历，唯一的区别是keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。

```
for (let index of ['a', 'b'].keys()) {
  console.log(index);
}
// 0
// 1

for (let elem of ['a', 'b'].values()) {
  console.log(elem);
}
// 'a'
// 'b'

for (let [index, elem] of ['a', 'b'].entries()) {
  console.log(index, elem);
}
// 0 "a"
// 1 "b"
```

数组实例的includes()

Array.protypeto.includes方法返回一个布尔值，表示某个数组是否包含给定的值。该方法属于ES7。

```
[1, 2, 3].includes(2);     // true
[1, 2, 3].includes(4);     // false
[1, 2, NaN].includes(NaN); // true
```

该方法的第二个参数表示搜索的起始位置，默认为0。

```
[1, 2, 3].includes(3, 3);  // false
[1, 2, 3].includes(3, -1); // true
```

下面代码用来检查当前环境是否支持该方法，如果不支持，部署一个简易的替代版本。

```
const contains = (() =>
  Array.prototype.includes
    ? (arr, value) => arr.includes(value)
    : (arr, value) => arr.some(el => el === value)
)();
contains(["foo", "bar"], "baz"); // => false
```

数组推导

数组推导（array comprehension）提供简洁写法，允许直接通过现有数组生成新数组。这项功能本来是要放入ES6的，但是TC39委员会想继续完善这项功能，让其支持所有数据结构（内部调用iterator对象），不像现在只支持数组，所以就把它推迟到了ES7。Babel转码器已经支持这个功能。

```
var a1 = [1, 2, 3, 4];
var a2 = [for (i of a1) i * 2];

a2 // [2, 4, 6, 8]
```

上面代码表示，通过for...of结构，数组a2直接在a1的基础上生成。

注意，数组推导中，for...of结构总是写在最前面，返回的表达式写在最后面。

for...of后面还可以附加if语句，用来设定循环的限制条件。

```
var years = [ 1954, 1974, 1990, 2006, 2010, 2014 ];

[for (year of years) if (year > 2000) year];
// [ 2006, 2010, 2014 ]

[for (year of years) if (year > 2000) if(year < 2010) year];
// [ 2006]

[for (year of years) if (year > 2000 && year < 2010) year];
// [ 2006]
```

上面代码表明，if语句写在for...of与返回的表达式之间，可以使用多个if语句。

数组推导可以替代map和filter方法。

```
[for (i of [1, 2, 3]) i * i];
// 等价于
[1, 2, 3].map(function (i) { return i * i });

[for (i of [1,4,2,3,-8]) if (i < 3) i];
// 等价于
[1,4,2,3,-8].filter(function(i) { return i < 3 });
```

上面代码说明，模拟map功能只要单纯的for...of循环就行了，模拟filter功能除了for...of循环，还必须加上if语句。

在一个数组推导中，还可以使用多个for...of结构，构成多重循环。

```
var a1 = ["x1", "y1"];
var a2 = ["x2", "y2"];
var a3 = ["x3", "y3"];

[for (s of a1) for (w of a2) for (r of a3) console.log(s + w + r)];
// x1x2x3
// x1x2y3
// x1y2x3
// x1y2y3
// y1x2x3
// y1x2y3
// y1y2x3
// y1y2y3
```

上面代码在一个数组推导之中，使用了三个for...of结构。

需要注意的是，数组推导的方括号构成了一个单独的作用域，在这个方括号中声明的变量类似于使用let语句声明的变量。

由于字符串可以视为数组，因此字符串也可以直接用于数组推导。

```
[for (c of 'abcde') if (/[aeiou]/.test(c)) c].join('') // 'ae'

[for (c of 'abcde') c+'0'].join('') // 'a0b0c0d0e0'
```

上面代码使用了数组推导，对字符串进行处理。

数组推导需要注意的地方是，新数组会立即在内存中生成。这时，如果原数组是一个很大的数组，将会非常耗费内存。

Array.observe()，Array.unobserve()

这两个方法用于监听（取消监听）数组的变化，指定回调函数。

它们的用法与Object.observe和Object.unobserve方法完全一致，也属于ES7的一部分，请参阅《对象的扩展》一章。唯一的区别是，对象可监听的变化一共有六种，而数组只有四种：add、update、delete、splice（数组的length属性发生变化）。

对象的扩展

属性的简洁表示法

ES6允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

```
function f( x, y ) {
  return { x, y };
}

// 等同于

function f( x, y ) {
  return { x: x, y: y };
}
```

上面是属性简写的例子，方法也可以简写。

```
var o = {
  method() {
    return "Hello!";
  }
};

// 等同于

var o = {
  method: function() {
    return "Hello!";
  }
};
```

下面是一个更实际的例子。

```
var Person = {

  name: '张三',

  //等同于birth: birth
  birth,

  // 等同于hello: function ()...
  hello() { console.log('我的名字是', this.name); }

};
```

这种写法用于函数的返回值，将会非常方便。

```
function getPoint() {
  var x = 1;
  var y = 10;

  return {x, y};
}

getPoint()
// {x:1, y:10}
```

属性名表达式

JavaScript语言定义对象的属性，有两种方法。

```
// 方法一
obj.foo = true;

// 方法二
obj['a'+'bc'] = 123;
```

上面代码的方法一是直接用标识符作为属性名，方法二是用表达式作为属性名，这时要将表达式放在方括号之内。

但是，如果使用字面量方式定义对象（使用大括号），在ES5中只能使用方法一（标识符）定义属性。

```
var obj = {
  foo: true,
  abc: 123
};
```

ES6允许字面量定义对象时，用方法二（表达式）作为对象的属性名，即把表达式放在方括号内。

```
let propKey = 'foo';

let obj = {
  [propKey]: true,
  ['a'+'bc']: 123
};
```

下面是另一个例子。

```
var lastWord = "last word";

var a = {
    "first word": "hello",
    [lastWord]: "world"
};

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

表达式还可以用于定义方法名。

```
let obj = {
  ['h'+'ello']() {
    return 'hi';
  }
};

console.log(obj.hello()); // hi
```

方法的name属性

函数的name属性，返回函数名。ES6为对象方法也添加了name属性。

```
var person = {
  sayName: function() {
    console.log(this.name);
  },
  get firstName() {
    return "Nicholas"
  }
}

person.sayName.name   // "sayName"
person.firstName.name // "get firstName"
```

上面代码中，方法的name属性返回函数名（即方法名）。如果使用了存值函数，则会在方法名前加上get。如果是存值函数，方法名的前面会加上set。

```
var doSomething = function() {
  // ...
};

console.log(doSomething.bind().name);   // "bound doSomething"

console.log((new Function()).name);     // "anonymous"
```

有两种特殊情况：bind方法创造的函数，name属性返回“bound”加上原函数的名字；Function构造函数创造的函数，name属性返回“anonymous”。

```
(new Function()).name // "anonymous"

var doSomething = function() {
  // ...
};
doSomething.bind().name // "bound doSomething"
```

Object.is()

Object.is()用来比较两个值是否严格相等。它与严格比较运算符（===）的行为基本一致，不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

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

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

ES5可以通过下面的代码，部署Object.is()。

```
Object.defineProperty(Object, 'is', {
  value: function(x, y) {
    if (x === y) {
      // 针对+0 不等于 -0的情况
      return x !== 0 || 1 / x === 1 / y;
    }
    // 针对NaN的情况
    return x !== x && y !== y;
  },
  configurable: true,
  enumerable: false,
  writable: true
});
```

Object.assign()

Object.assign方法用来将源对象（source）的所有可枚举属性，复制到目标对象（target）。它至少需要两个对象作为参数，第一个参数是目标对象，后面的参数都是源对象。只要有一个参数不是对象，就会抛出TypeError错误。

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

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

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

注意，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。

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

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

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

assign方法有很多用处。

（1）为对象添加属性

```
class Point {
  constructor(x, y) {
    Object.assign(this, {x, y});
  }
}
```

上面方法通过assign方法，将x属性和y属性添加到Point类的对象实例。

（2）为对象添加方法

```
Object.assign(SomeClass.prototype, {
  someMethod(arg1, arg2) {
    ···
  },
  anotherMethod() {
    ···
  }
});

// 等同于下面的写法
SomeClass.prototype.someMethod = function (arg1, arg2) {
  ···
};
SomeClass.prototype.anotherMethod = function () {
  ···
};
```

上面代码使用了对象属性的简洁表示法，直接将两个函数放在大括号中，再使用assign方法添加到SomeClass.prototype之中。

（3）克隆对象

```
function clone(origin) {
  return Object.assign({}, origin);
}
```

上面代码将原始对象拷贝到一个空对象，就得到了原始对象的克隆。

不过，采用这种方法克隆，只能克隆原始对象自身的值，不能克隆它继承的值。如果想要保持继承链，可以采用下面的代码。

```
function clone(origin) {
  let originProto = Object.getPrototypeOf(origin);
  return Object.assign(Object.create(originProto), origin);
}
```

（4）合并多个对象

将多个对象合并到某个对象。

```
const merge =
  (target, ...sources) => Object.assign(target, ...sources);
```

如果希望合并后返回一个新对象，可以改写上面函数，对一个空对象合并。

```
const merge =
  (...sources) => Object.assign({}, ...sources);
```

（5）为属性指定默认值

```
const DEFAULTS = {
  logLevel: 0,
  outputFormat: 'html'
};

function processContent(options) {
  let options = Object.assign({}, DEFAULTS, options);
}
```

上面代码中，DEFAULTS对象是默认值，options对象是用户提供的参数。assign方法将DEFAULTS和options合并成一个新对象，如果两者有同名属性，则option的属性值会覆盖DEFAULTS的属性值。

proto属性，Object.setPrototypeOf()，Object.getPrototypeOf()

（1）proto属性

proto属性，用来读取或设置当前对象的prototype对象。该属性一度被正式写入ES6草案，但后来又被移除。目前，所有浏览器（包括IE11）都部署了这个属性。

```
// es6的写法

var obj = {
  __proto__: someOtherObj,
  method: function() { ... }
}

// es5的写法

var obj = Object.create(someOtherObj);
obj.method = function() { ... }
```

有了这个属性，实际上已经不需要通过Object.create()来生成新对象了。

（2）Object.setPrototypeOf()

Object.setPrototypeOf方法的作用与**proto**相同，用来设置一个对象的prototype对象。它是ES6正式推荐的设置原型对象的方法。

```
// 格式
Object.setPrototypeOf(object, prototype)

// 用法
var o = Object.setPrototypeOf({}, null);
```

该方法等同于下面的函数。

```
function (obj, proto) {
  obj.__proto__ = proto;
  return obj;
}
```

（3）Object.getPrototypeOf()

该方法与setPrototypeOf方法配套，用于读取一个对象的prototype对象。

```
Object.getPrototypeOf(obj)
```

Symbol

概述

在ES5中，对象的属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法，新方法的名字有可能与现有方法产生冲突。如果有一种机制，保证每个

属性的名字都是独一无二的就好了，这样就从根本上防止属性名的冲突。这就是ES6引入Symbol的原因。

ES6引入了一种新的原始数据类型Symbol，表示独一无二的ID。它通过Symbol函数生成。这就是说，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"
```

上面代码中，变量s就是一个独一无二的ID。typeof运算符的结果，表明变量s是Symbol数据类型，而不是字符串之类的其他类型。

注意，Symbol函数前不能使用new命令，否则会报错。这是因为生成的Symbol是一个原始类型的值，不是对象。也就是说，由于Symbol值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

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，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。

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

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

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

上面代码中，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。

注意，Symbol函数的参数只是表示对当前Symbol类型的值的描述，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
var s1 = Symbol();
var s2 = Symbol();

s1 === s2 // false

// 有参数的情况
var s1 = Symbol("foo");
var s2 = Symbol("foo");

s1 === s2 // false
```

上面代码中，s1和s2都是Symbol函数的返回值，而且参数相同，但是它们是不相等的。

Symbol类型的值不能与其他类型的值进行运算，会报错。

```
var 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类型的值可以转为字符串。

```
var sym = Symbol('My symbol');

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

作为属性名的Symbol

由于每一个Symbol值都是不相等的，这意味着Symbol值可以作为标识符，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。

```
var mySymbol = Symbol();

// 第一种写法
var a = {};
a[mySymbol] = 'Hello!';

// 第二种写法
var a = {
  [mySymbol]: 123
};

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

// 以上写法都得到同样结果
a[mySymbol] // "Hello!"
```

上面代码通过方括号结构和Object.defineProperty，将对象的属性名指定为一个Symbol值。

注意，Symbol值作为对象属性名时，不能用点运算符。

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

a.mySymbol = 'Hello!';
a[mySymbol] // undefined
a['mySymbol'] // "Hello!"
```

上面代码中，因为点运算符后面总是字符串，所以不会读取mySymbol作为标识名所指代的那个值，导致a的属性名实际上是一个字符串，而不是一个Symbol值。

同理，在对象的内部，使用Symbol值定义属性时，Symbol值必须放在方括号之中。

```
let s = Symbol();

let obj = {
  [s]: function (arg) { ... }
};

obj[s](123);
```

上面代码中，如果s不放在方括号中，该属性的键名就是字符串s，而不是s所代表的那个Symbol值。

采用增强的对象写法，上面代码的obj对象可以写得更简洁一些。

```
let obj = {
  [s](arg) { ... }
};
```

Symbol类型还可以用于定义一组常量，保证这组常量的值都是不相等的。

```
log.levels = {
    DEBUG: Symbol('debug'),
    INFO: Symbol('info'),
    WARN: Symbol('warn'),
};
log(log.levels.DEBUG, 'debug message');
log(log.levels.INFO, 'info message');
```

还有一点需要注意，Symbol值作为属性名时，该属性还是公开属性，不是私有属性。

属性名的遍历

Symbol作为属性名，该属性不会出现在for...in、for...of循环中，也不会被`Object.keys()`、`Object.getOwnPropertyNames()`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。

Object.getOwnPropertySymbols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。

```
var obj = {};
var a = Symbol('a');
var b = Symbol.for('b');

obj[a] = 'Hello';
obj[b] = 'World';

var objectSymbols = Object.getOwnPropertySymbols(obj);

objectSymbols
// [Symbol(a), Symbol(b)]
```

下面是另一个例子，Object.getOwnPropertySymbols方法与for...in循环、Object.getOwnPropertyNames方法进行对比的例子。

```
var obj = {};

var foo = Symbol("foo");

Object.defineProperty(obj, foo, {
  value: "foobar",
});

for (var i in obj) {
  console.log(i); // 无输出
}

Object.getOwnPropertyNames(obj)
// []

Object.getOwnPropertySymbols(obj)
// [Symbol(foo)]
```

上面代码中，使用Object.getOwnPropertyNames方法得不到Symbol属性名，需要使用Object.getOwnPropertySymbols方法。

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)
// [Symbol(my_key), 'enum', 'nonEnum']
```

由于以Symbol值作为名称的属性，不会被常规方法遍历得到。我们可以利用这个特性，为对象定义一些非私有的、但又希望只用于内部的方法。

```
var size = Symbol('size');

class Collection {
  constructor() {
    this[size] = 0;
  }

  add(item) {
    this[this[size]] = item;
    this[size]++;
  }

  static sizeOf(instance) {
    return instance[size];
  }
}

var x = new Collection();
Collection.sizeOf(x) // 0

x.add('foo');
Collection.sizeOf(x) // 1

Object.keys(x) // ['0']
Object.getOwnPropertyNames(x) // ['0']
Object.getOwnPropertySymbols(x) // [Symbol(size)]
```

上面代码中，对象x的size属性是一个Symbol值，所以`Object.keys(x)`、`Object.getOwnPropertyNames(x)`都无法获取它。这就造成了一种非私有的内部方法的效果。

Symbol.for()，Symbol.keyFor()

有时，我们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，否则就新建并返回一个以该字符串为名称的Symbol值。

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

s1 === s2 // true
```

上面代码中，s1和s2都是Symbol值，但是它们都是同样参数的`Symbol.for`方法生成的，所以实际上是同一个值。

`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，如果你调用`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。

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

var s2 = Symbol("foo");
Symbol.keyFor(s2) // undefined
```

上面代码中，变量s2属于未登记的Symbol值，所以返回undefined。

需要注意的是，`Symbol.for`为Symbol值登记的名字，是全局环境的，可以在不同的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
```

上面代码中，iframe窗口生成的Symbol值，可以在主页面得到。

内置的Symbol值

除了定义自己使用的Symbol值以外，ES6还提供一些内置的Symbol值，指向语言内部使用的方法。

（1）Symbol.hasInstance

对象的Symbol.hasInstance属性，指向一个内部方法。该对象使用instanceof运算符时，会调用这个方法，判断该对象是否为某个构造函数的实例。比如，`foo instanceof Foo`在语言内部，实际调用的是`Foo[Symbol.hasInstance](foo)`。

（2）Symbol.isConcatSpreadable

对象的Symbol.isConcatSpreadable属性，指向一个方法。该对象使用Array.prototype.concat()时，会调用这个方法，返回一个布尔值，表示该对象是否可以扩展成数组。

（3）Symbol.isRegExp

对象的Symbol.isRegExp属性，指向一个方法。该对象被用作正则表达式时，会调用这个方法，返回一个布尔值，表示该对象是否为一个正则对象。

（4）Symbol.match

对象的Symbol.match属性，指向一个函数。当执行 `str.match(myObject)` 时，如果该属性存在，会调用它，返回该方法的返回值。

（5）Symbol.iterator

对象的Symbol.iterator属性，指向该对象的默认遍历器方法，即该对象进行for...of循环时，会调用这个方法，返回该对象的默认遍历器，详细介绍参见《Iterator和for...of循环》一章。

```
class Collection {
  *[Symbol.iterator]() {
    let i = 0;
    while(this[i] !== undefined) {
      yield this[i];
      ++i;
    }
  }

}

let myCollection = new Collection();
myCollection[0] = 1;
myCollection[1] = 2;

for(let value of myCollection) {
  console.log(value);
}
// 1
// 2
```

（6）Symbol.toPrimitive

对象的Symbol.toPrimitive属性，指向一个方法。该对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值。

（7）Symbol.toStringTag

对象的Symbol.toStringTag属性，指向一个方法。在该对象上面调用`Object.prototype.toString`方法时，如果这个属性存在，它的返回值会出现在toString方法返回的字符串之中，表示对象的类型。也就是说，这个属性可以用来定制`[object Object]`或`[object Array]`中object后面的那个字符串。

```
class Collection {
  get [Symbol.toStringTag]() {
    return 'xxx';
  }
}
var x = new Collection();
Object.prototype.toString.call(x) // "[object xxx]"
```

（8）Symbol.unscopables

对象的Symbol.unscopables属性，指向一个对象。该对象指定了使用with关键字时，那些属性会被with环境排除。

```
Array.prototype[Symbol.unscopables]
// {
//   copyWithin: true,
//   entries: true,
//   fill: true,
//   find: true,
//   findIndex: true,
//   keys: true
// }

Object.keys(Array.prototype[Symbol.unscopables])
// ['copyWithin', 'entries', 'fill', 'find', 'findIndex', 'keys']
```

上面代码说明，数组有6个属性，会被with命令排除。

```
// 没有unscopables时
class MyClass {
  foo() { return 1; }
}

var foo = function () { return 2; };

with (MyClass.prototype) {
    foo(); // 1
}

// 有unscopables时
class MyClass {
  foo() { return 1; }
  get [Symbol.unscopables]() {
    return { foo: true };
  }
}

var foo = function () { return 2; };

with (MyClass.prototype) {
  foo(); // 2
}
```

Proxy

概述

Proxy用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（meta programming），即对编程语言进行编程。

Proxy可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

```
var obj = new Proxy({}, {
  get: function (target, key, receiver) {
    console.log(`getting ${key}!`);
    return Reflect.get(target, key, receiver);
  },
  set: function (target, key, value, receiver) {
    console.log(`setting ${key}!`);
    return Reflect.set(target, key, value, receiver);
  }
});
```

上面代码对一个空对象架设了一层拦截，重定义了属性的读取（get）和设置（set）行为。这里暂时不解释具体的语法，只看运行结果。对设置了拦截行为的对象obj，去读写它的属性，就会得到下面的结果。

```
obj.count = 1
//  setting count!
++obj.count
//  getting count!
//  setting count!
//  2
```

上面代码说明，Proxy实际上重载（overload）了点运算符，即用自己的定义覆盖了语言的原始定义。

ES6原生提供Proxy构造函数，用来生成Proxy实例。

```
var proxy = new Proxy(target, handler)
```

Proxy对象的所用用法，都是上面这种形式，不同的只是handler参数的写法。其中，`new Proxy()`表示生成一个Proxy实例，target参数表示所要拦截的目标对象，handler参数也是一个对象，用来定制拦截行为。

下面是另一个拦截读取属性行为的例子。

```
var proxy = new Proxy({}, {
  get: function(target, property) {
    return 35;
  }
});

proxy.time // 35
proxy.name // 35
proxy.title // 35
```

上面代码中，作为构造函数，Proxy接受两个参数。第一个参数是所要代理的目标对象（上例是一个空对

象），即如果没有Proxy的介入，操作原来要访问的就是这个对象；第二个参数是一个配置对象，对于每一个被代理的操作，需要提供一个对应的处理函数，该函数将拦截对应的操作。比如，上面代码中，配置对象有一个get方法，用来拦截对目标对象属性的访问请求。get方法的两个参数分别是目标对象和所要访问的属性。可以看到，由于拦截函数总是返回35，所以访问任何属性都得到35。

注意，要使得Proxy起作用，必须针对Proxy实例（上例是proxy对象）进行操作，而不是针对目标对象（上例是空对象）进行操作。

一个技巧是将Proxy对象，设置到`object.proxy`属性，从而可以在object对象上调用。

```
var object = { proxy: new Proxy(target, handler) }
```

Proxy实例也可以作为其他对象的原型对象。

```
var proxy = new Proxy({}, {
  get: function(target, property) {
    return 35;
  }
});

let obj = Object.create(proxy);
obj.time // 35
```

上面代码中，proxy对象是obj对象的原型，obj对象本身并没有time属性，所有根据原型链，会在proxy对象上读取该属性，导致被拦截。

同一个拦截器函数，可以设置拦截多个操作。

```
var handler = {
  get: function(target, name) {
    if (name === 'prototype') return Object.prototype;
    return 'Hello, '+ name;
  },
  apply: function(target, thisBinding, args) { return args[0]; },
  construct: function(target, args) { return args[1]; }
};

var fproxy = new Proxy(function(x,y) {
  return x+y;
},  handler);

fproxy(1,2); // 1
new fproxy(1,2); // 2
fproxy.prototype; // Object.prototype
fproxy.foo; // 'Hello, foo'
```

下面是Proxy支持的拦截操作一览。

对于可以设置、但没有设置拦截的操作，则直接落在目标对象上，按照原先的方式产生结果。

（1）get(target, propKey, receiver)

拦截对象属性的读取，比如`proxy.foo`和`proxy['foo']`，返回类型不限。最后一个参数receiver可选，当target对象设置了propKey属性的get函数时，receiver对象会绑定get函数的this对象。

（2）set(target, propKey, value, receiver)

拦截对象属性的设置，比如`proxy.foo = v`或`proxy['foo'] = v`，返回一个布尔值。

（3）has(target, propKey)

拦截`propKey in proxy`的操作，返回一个布尔值。

（4）deleteProperty(target, propKey)

拦截`delete proxy[propKey]`的操作，返回一个布尔值。

（5）enumerate(target)

拦截`for (var x in proxy)`，返回一个遍历器。

6）hasOwn(target, propKey)

拦截`proxy.hasOwnProperty('foo')`，返回一个布尔值。

（7）ownKeys(target)

拦截`Object.getOwnPropertyNames(proxy)`、`Object.getOwnPropertySymbols(proxy)`、`Object.keys(proxy)`，返回一个数组。该方法返回对象所有自身的属性，而`Object.keys()`仅返回对象可遍历的属性。

（8）getOwnPropertyDescriptor(target, propKey)

拦截`Object.getOwnPropertyDescriptor(proxy, propKey)`，返回属性的描述对象。

（9）defineProperty(target, propKey, propDesc)

拦截`Object.defineProperty(proxy, propKey, propDesc）`、`Object.defineProperties(proxy, propDescs)`，返回一个布尔值。

（10）preventExtensions(target)

拦截`Object.preventExtensions(proxy)`，返回一个布尔值。

（11）getPrototypeOf(target)

拦截`Object.getPrototypeOf(proxy)`，返回一个对象。

（12）isExtensible(target)

拦截`Object.isExtensible(proxy)`，返回一个布尔值。

（13）setPrototypeOf(target, proto)

拦截`Object.setPrototypeOf(proxy, proto)`，返回一个布尔值。

如果目标对象是函数，那么还有两种额外操作可以拦截。

（14）apply(target, object, args)

拦截Proxy实例作为函数调用的操作，比如`proxy(...args)`、`proxy.call(object, ...args)`、`proxy.apply(...)`。

（15）construct(target, args, proxy)

拦截Proxy实例作为构造函数调用的操作，比如new proxy(...args)。

下面是其中几个重要拦截方法的详细介绍。

get()

get方法用于拦截某个属性的读取操作。上文已经有一个例子，下面是另一个拦截读取操作的例子。

```
var person = {
  name: "张三"
};

var proxy = new Proxy(person, {
  get: function(target, property) {
    if (property in target) {
      return target[property];
    } else {
      throw new ReferenceError("Property \"" + property + "\" does not exist.");
    }
  }
});

proxy.name // "张三"
proxy.age // 抛出一个错误
```

上面代码表示，如果访问目标对象不存在的属性，会抛出一个错误。如果没有这个拦截函数，访问不存在的属性，只会返回undefined。

利用proxy，可以将读取属性的操作（get），转变为执行某个函数。

```
var pipe = (function () {
  var pipe;
  return function (value) {
    pipe = [];
    return new Proxy({}, {
      get: function (pipeObject, fnName) {
        if (fnName == "get") {
          return pipe.reduce(function (val, fn) {
            return fn(val);
          }, value);
        }
        pipe.push(window[fnName]);
        return pipeObject;
      }
    });
  }
}());

var double = function (n) { return n*2 };
var pow = function (n) { return n*n };
var reverseInt = function (n) { return n.toString().split('').reverse().join('')|0 };

pipe(3) . double . pow . reverseInt . get
// 63
```

上面代码设置Proxy以后，达到了将函数名链式使用的效果。

set()

set方法用来拦截某个属性的赋值操作。假定Person对象有一个age属性，该属性应该是一个不大于200的整数，那么可以使用Proxy对象保证age的属性值符合要求。

```
let validator = {
  set: function(obj, prop, value) {
    if (prop === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError('The age is not an integer');
      }
      if (value > 200) {
        throw new RangeError('The age seems invalid');
      }
    }

    // 对于age以外的属性，直接保存
    obj[prop] = value;
  }
};

let person = new Proxy({}, validator);

person.age = 100;

person.age // 100
person.age = 'young' // 报错
person.age = 300 // 报错
```

上面代码中，由于设置了存值函数set，任何不符合要求的age属性赋值，都会抛出一个错误。利用set方法，还可以数据绑定，即每当对象发生变化时，会自动更新DOM。

apply()

apply方法拦截函数的调用、call和apply操作。

```
var target = function () { return 'I am the target'; };
var handler = {
  apply: function (receiver, ...args) {
    return 'I am the proxy';
  }
};

var p = new Proxy(target, handler);

p() === 'I am the proxy';
// true
```

上面代码中，变量p是Proxy的实例，当它作为函数调用时（p()），就会被apply方法拦截，返回一个字符串。

ownKeys()

ownKeys方法用来拦截Object.keys()操作。

```
let target = {};

let handler = {
  ownKeys(target) {
    return ['hello', 'world'];
  }
};

let proxy = new Proxy(target, handler);

Object.keys(proxy)
// [ 'hello', 'world' ]
```

上面代码拦截了对于target对象的Object.keys()操作，返回预先设定的数组。

Proxy.revocable()

Proxy.revocable方法返回一个可取消的Proxy实例。

```
let target = {};
let handler = {};

let {proxy, revoke} = Proxy.revocable(target, handler);

proxy.foo = 123;
proxy.foo // 123

revoke();
proxy.foo // TypeError: Revoked
```

Proxy.revocable方法返回一个对象，该对象的proxy属性是Proxy实例，revoke属性是一个函数，可以取消Proxy实例。上面代码中，当执行revoke函数之后，再访问Proxy实例，就会抛出一个错误。

Reflect

概述

Reflect对象与Proxy对象一样，也是ES6为了操作对象而提供的新API。Reflect对象的设计目的有这样几个。

（1）将Object对象的一些明显属于语言层面的方法，放到Reflect对象上。现阶段，某些方法同时在

Object和Reflect对象上部署，未来的新方法将只部署在Reflect对象上。

（2）修改某些Object方法的返回结果，让其变得更合理。比如，`Object.defineProperty(obj, name, desc)` 在无法定义属性时，会抛出一个错误，而 `Reflect.defineProperty(obj, name, desc)` 则会返回false。

（3）让Object操作都变成函数行为。某些Object操作是命令式，比如 `name in obj` 和 `delete obj[name]` ，而 `Reflect.has(obj, name)` 和 `Reflect.deleteProperty(obj, name)` 让它们变成了函数行为。

（4）Reflect对象的方法与Proxy对象的方法一一对应，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法。这就让Proxy对象可以方便地调用对应的Reflect方法，完成默认行为，作为修改行为的基础。

```
Proxy(target, {
  set: function(target, name, value, receiver) {
    var success = Reflect.set(target,name, value, receiver);
    if (success) {
      log('property '+name+' on '+target+' set to '+value);
    }
    return success;
  }
});
```

上面代码中，Proxy方法拦截target对象的属性赋值行为。它采用Reflect.set方法将值赋值给对象的属性，然后再部署额外的功能。

下面是get方法的例子。

```
var loggedObj = new Proxy(obj, {
  get: function(target, name) {
    console.log("get", target, name);
    return Reflect.get(target, name);
  }
});
```

方法

Reflect对象的方法清单如下。

- Reflect.getOwnPropertyDescriptor(target,name)
- Reflect.defineProperty(target,name,desc)
- Reflect.getOwnPropertyNames(target)
- Reflect.getPrototypeOf(target)

- Reflect.deleteProperty(target,name)
- Reflect.enumerate(target)
- Reflect.freeze(target)
- Reflect.seal(target)
- Reflect.preventExtensions(target)
- Reflect.isFrozen(target)
- Reflect.isSealed(target)
- Reflect.isExtensible(target)
- Reflect.has(target,name)
- Reflect.hasOwn(target,name)
- Reflect.keys(target)
- Reflect.get(target,name,receiver)
- Reflect.set(target,name,value,receiver)
- Reflect.apply(target,thisArg,args)
- Reflect.construct(target,args)

上面这些方法的作用，大部分与Object对象的同名方法的作用都是相同的。下面是对其中几个方法的解释。

（1）Reflect.get(target,name,receiver)

查找并返回target对象的name属性，如果没有该属性，则返回undefined。

如果name属性部署了读取函数，则读取函数的this绑定receiver。

```
var obj = {
  get foo() { return this.bar(); },
  bar: function() { ... }
}

// 下面语句会让 this.bar()
// 变成调用 wrapper.bar()
Reflect.get(obj, "foo", wrapper);
```

（2）Reflect.set(target, name, value, receiver)

设置target对象的name属性等于value。如果name属性设置了赋值函数，则赋值函数的this绑定receiver。

（3）Reflect.has(obj, name)

等同于`name in obj`。

（4）Reflect.deleteProperty(obj, name)

等同于`delete obj[name]`。

（5）Reflect.construct(target, args)

等同于`new target(...args)`，这提供了一种不使用new，来调用构造函数的方法。

（6）Reflect.getPrototypeOf(obj)

读取对象的**proto**属性，等同于`Object.getPrototypeOf(obj)`。

（7）Reflect.setPrototypeOf(obj, newProto)

设置对象的**proto**属性。注意，Object对象没有对应这个方法的方法。

（8）Reflect.apply(fun,thisArg,args)

等同于`Function.prototype.apply.call(fun,thisArg,args)`。一般来说，如果要绑定一个函数的this对象，可以这样写`fn.apply(obj, args)`，但是如果函数定义了自己的apply方法，就只能写成`Function.prototype.apply.call(fn, obj, args)`，采用Reflect对象可以简化这种操作。

另外，需要注意的是，Reflect.set()、Reflect.defineProperty()、Reflect.freeze()、Reflect.seal()和Reflect.preventExtensions()返回一个布尔值，表示操作是否成功。它们对应的Object方法，失败时都会抛出错误。

```
// 失败时抛出错误
Object.defineProperty(obj, name, desc);
// 失败时返回false
Reflect.defineProperty(obj, name, desc);
```

上面代码中，Reflect.defineProperty方法的作用与Object.defineProperty是一样的，都是为对象定义一个属性。但是，Reflect.defineProperty方法失败时，不会抛出错误，只会返回false。

Object.observe()，Object.unobserve()

Object.observe方法用来监听对象（以及数组）的变化。一旦监听对象发生变化，就会触发回调函数。

```
var user = {};
Object.observe(user, function(changes){
  changes.forEach(function(change) {
    user.fullName = user.firstName+" "+user.lastName;
  });
});

user.firstName = 'Michael';
user.lastName = 'Jackson';
user.fullName // 'Michael Jackson'
```

上面代码中，Object.observer方法监听user对象。一旦该对象发生变化，就自动生成fullName属性。

一般情况下，Object.observe方法接受两个参数，第一个参数是监听的对象，第二个函数是一个回调函数。一旦监听对象发生变化（比如新增或删除一个属性），就会触发这个回调函数。很明显，利用这个方法可以做很多事情，比如自动更新DOM。

```
var div = $("#foo");

Object.observe(user, function(changes){
  changes.forEach(function(change) {
    var fullName = user.firstName+" "+user.lastName;
    div.text(fullName);
  });
});
```

上面代码中，只要user对象发生变化，就会自动更新DOM。如果配合jQuery的change方法，就可以实现数据对象与DOM对象的双向自动绑定。

回调函数的changes参数是一个数组，代表对象发生的变化。下面是一个更完整的例子。

```
var o = {};

function observer(changes){
  changes.forEach(function(change) {
    console.log('发生变动的属性：' + change.name);
    console.log('变动前的值：' + change.oldValue);
    console.log('变动后的值：' + change.object[change.name]);
    console.log('变动类型：' + change.type);
  });
}

Object.observe(o, observer);
```

参照上面代码，Object.observe方法指定的回调函数，接受一个数组（changes）作为参数。该数组的成员与对象的变化一一对应，也就是说，对象发生多少个变化，该数组就有多少个成员。每个成员是一个对象（change），它的name属性表示发生变化源对象的属性名，oldValue属性表示发生变化前的值，object属性指向变动后的源对象，type属性表示变化的种类。基本上，change对象是下面的样子。

```
var change = {
  object: {...},
  type: 'update',
  name: 'p2',
  oldValue: 'Property 2'
}
```

Object.observe方法目前共支持监听六种变化。

- add：添加属性
- update：属性值的变化
- delete：删除属性
- setPrototype：设置原型
- reconfigure：属性的attributes对象发生变化
- preventExtensions：对象被禁止扩展（当一个对象变得不可扩展时，也就不必再监听了）

Object.observe方法还可以接受第三个参数，用来指定监听的事件种类。

```
Object.observe(o, observer, ['delete']);
```

上面的代码表示，只在发生delete事件时，才会调用回调函数。

Object.unobserve方法用来取消监听。

```
Object.unobserve(o, observer);
```

注意，Object.observe和Object.unobserve这两个方法不属于ES6，而是属于ES7的一部分。不过，Chrome浏览器从33版起就已经支持。

函数的扩展

函数参数的默认值

在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
log('Hello', 'China') // Hello China
log('Hello', '') // Hello World
```

上面代码检查函数log的参数y有没有赋值，如果没有，则指定默认值为World。这种写法的缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。就像上面代码的最后一行，参数y等于空字符，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。这有两种写法。

```
// 写法一
if (typeof y === 'undefined') {
  y = 'World';
}

// 写法二
if (arguments.length === 1) {
  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
```

可以看到　ES6的写法比ES5简洁许多，而且非常自然。下面是另一个例子。

```
function Point(x = 0, y = 0) {
  this.x = x;
  this.y = y;
}

var p = new Point();
// p = { x:0, y:0 }
```

除了简洁，ES6的写法还有两个好处：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，即使未来的版本彻底拿到这个参数，也不会导致以前的代码无法运行。

默认值的写法非常灵活，下面是一个为对象属性设置默认值的例子。

```
fetch(url, { body = '', method = 'GET', headers = {} }){
  console.log(method);
}
```

上面代码中，传入函数fetch的第二个参数是一个对象，调用的时候可以为它的三个属性设置默认值。

甚至还可以设置双重默认值。

```
fetch(url, { method = 'GET' } = {}){
  console.log(method);
}
```

上面代码中，调用函数fetch时，如果不含第二个参数，则默认值为一个空对象；如果包含第二个参数，则它的method属性默认值为GET。

定义了默认值的参数，必须是函数的尾部参数，其后不能再有其他无默认值的参数。这是因为有了默认值以后，该参数可以省略，只有位于尾部，才可能判断出到底省略了哪些参数。

```
// 以下两种写法都是错的

function f(x = 5, y) {
}

function f(x, y = 5, z) {
}
```

如果传入undefined，将触发该参数等于默认值，null则没有这个效果。

```
function foo(x = 5, y = 6){
  console.log(x,y);
}

foo(undefined, null)
// 5 null
```

上面代码中，x参数对应undefined，结果触发了默认值，y参数等于null，就没有触发默认值。

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。也就是说，指定了默认值后，length属性将失真。

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

上面代码中，length属性的返回值，等于函数的参数个数减去指定了默认值的参数个数。

利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误。

```
function throwIfMissing() {
  throw new Error('Missing parameter');
}

function foo(mustBeProvided = throwIfMissing()) {
  return mustBeProvided;
}

foo()
// Error: Missing parameter
```

上面代码的foo函数，如果调用的时候没有参数，就会调用默认值throwIfMissing函数，从而抛出一个错误。

从上面代码还可以看到，参数mustBeProvided的默认值等于throwIfMissing函数的运行结果（即函数名之后有一对圆括号），这表明参数的默认值不是在定义时执行，而是在运行时执行（即如果参数已经赋值，默认值中的函数就不会运行），这与python语言不一样。

另一个需要注意的地方是，参数默认值所处的作用域，不是全局作用域，而是函数作用域。

```
var x = 1;

function foo(x, y = x) {
  console.log(y);
}

foo(2) // 2
```

上面代码中，参数y的默认值等于x，由于处在函数作用域，所以x等于参数x，而不是全局变量x。

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; // error
  const x = 2; // error
}
```

上面代码中，参数变量x是默认声明的，在函数体中，不能用let或const再次声明，否则会报错。

参数默认值可以与解构赋值，联合起来使用。

```
function foo({x, y = 5}) {
  console.log(x, y);
}

foo({}) // undefined, 5
foo({x: 1}) // 1, 5
foo({x: 1, y: 2}) // 1, 2
```

上面代码中，foo函数的参数是一个对象，变量x和y用于解构赋值，y有默认值5。

rest参数

ES6引入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 (var val of values) {
    sum += val;
  }

  return sum;
}

add(2, 5, 3) // 10
```

上面代码的add函数是一个求和函数，利用rest参数，可以向该函数传入任意数目的参数。

下面是一个rest参数代替arguments变量的例子。

```
// arguments变量的写法
const sortNumbers = () =>
  Array.prototype.slice.call(arguments).sort();

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

上面代码的两种写法，比较后可以发现，rest参数的写法更自然也更简洁。

rest参数中的变量代表一个数组，所以数组特有的方法都可以用于这个变量。下面是一个利用rest参数改写数组push方法的例子。

```
function push(array, ...items) {
  items.forEach(function(item) {
    array.push(item);
    console.log(item);
  });
}

var a = [];
push(a, 1, 2, 3)
```

注意，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。

```
// 报错
function f(a, ...b, c) {
  // ...
}
```

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
```

扩展运算符

扩展运算符（spread）是三个点（...）。它好比rest参数的逆运算，将一个数组转为用逗号分隔的参数序列。该运算符主要用于函数调用。

```
function push(array, ...items) {
  array.push(...items);
}

function add(x, y) {
  return x + y;
}

var numbers = [4, 38];
add(...numbers) // 42
```

上面代码中，`array.push(...items)`和`add(...numbers)`这两行，都是函数的调用，它们的都使用了扩展运算符。该运算符将一个数组，变为参数序列。

下面是Date函数的参数使用扩展运算符的例子。

```
const date = new Date(...[2015, 1, 1]);
```

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){}
var args = [0, 1, 2];
f(...args);
```

扩展运算符与正常的函数参数可以结合使用，非常灵活。

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

下面是扩展运算符取代apply方法的一个实际的例子，应用Math.max方法，简化求出一个数组最大元素的写法。

```
// ES5的写法
Math.max.apply(null, [14, 3, 77])

// ES6的写法
Math.max(...[14, 3, 77])

// 等同于
Math.max(14, 3, 77);
```

上面代码表示，由于JavaScript不提供求数组最大元素的函数，所以只能套用Math.max函数，将数组转为一个参数序列，然后求最大值。有了扩展运算符以后，就可以直接用Math.max了。

另一个例子是通过push函数，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
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1.push(...arr2);
```

上面代码的ES5写法中，push方法的参数不能是数组，所以只好通过apply方法变通使用push方法。有了扩展运算符，就可以直接将数组传入push方法。

扩展运算符还可以用于数组的赋值。

```
var a = [1];
var b = [2, 3, 4];
var c = [6, 7];
var d = [0, ...a, ...b, 5, ...c];

d
// [0, 1, 2, 3, 4, 5, 6, 7]
```

上面代码其实也提供了，将一个数组拷贝进另一个数组的便捷方法。

```
const arr2 = [...arr1];
```

扩展运算符也可以与解构赋值结合起来，用于生成数组。

```
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest  // [2, 3, 4, 5]

const [first, ...rest] = [];
first // undefined
rest  // []:

const [first, ...rest] = ["foo"];
first  // "foo"
rest   // []

const [first, ...rest] = ["foo", "bar"];
first // "foo"
rest  // ["bar"]

const [first, ...rest] = ["foo", "bar", "baz"];
first // "foo"
rest  // ["bar","baz"]
```

如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
const [...butLast, last] = [1, 2, 3, 4, 5];
// 报错

const [first, ...middle, last] = [1, 2, 3, 4, 5];
// 报错
```

JavaScript的函数只能返回一个值，如果需要返回多个值，只能返回数组或对象。扩展运算符提供了解决这个问题的一种变通方法。

```
var dateFields = readDateFields(database);
var d = new Date(...dateFields);
```

上面代码从数据库取出一行数据，通过扩展运算符，直接将其传入构造函数Date。

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" ]
```

任何类似数组的对象，都可以用扩展运算符转为真正的数组。

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

上面代码中，querySelectorAll方法返回的是一个nodeList对象，扩展运算符可以将其转为真正的数组。

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'],
]);

let arr = [...map.keys()]; // [1, 2, 3]
```

Generator函数运行后，返回一个遍历器对象，因此也可以使用扩展运算符。

```
var go = function*(){
  yield 1;
  yield 2;
  yield 3;
};

[...go()] // [1, 2, 3]
```

上面代码中，变量go是一个Generator函数，执行后返回的是一个遍历器，对这个遍历器执行扩展运算符，就会将内部遍历得到的值，转为一个数组。

箭头函数

ES6允许使用“箭头”（=>）定义函数。

```
var f = v => v;
```

上面的箭头函数等同于：

```
var f = function(v) {
  return v;
};
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

```
var f = () => 5;
// 等同于
var f = function (){ return 5 };

var sum = (num1, num2) => num1 + num2;
// 等同于
var sum = function(num1, num2) {
  return num1 + num2;
};
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用return语句返回。

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

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号。

```
var getTempItem = id => ({ id: id, name: "Temp" });
```

箭头函数可以与变量解构结合使用。

```
const full = ({ first, last }) => first + ' ' + last;

// 等同于
function full( person ){
  return person.first +  ‘  ‘ + person.name;
}
```

箭头函数使得表达更加简洁。

```
const isEven = n => n % 2 == 0;
const square = n => n * n;
```

上面代码只用了两行，就定义了两个简单的工具函数。如果不用箭头函数，可能就要占用多行，而且还不如现在这样写醒目。

箭头函数的一个用处是简化回调函数。

```
// 正常函数写法
[1,2,3].map(function (x) {
  return x * x;
});

// 箭头函数写法
[1,2,3].map(x => x * x);
```

另一个例子是

```
// 正常函数写法
var result = values.sort(function(a, b) {
  return a - b;
});

// 箭头函数写法
var result = values.sort((a, b) => a - b);
```

下面是rest参数与箭头函数结合的例子。

```
const numbers = (...nums) => nums;

numbers(1, 2, 3, 4, 5)
// [1,2,3,4,5]

const headAndTail = (head, ...tail) => [head, tail];

headAndTail(1, 2, 3, 4, 5)
// [1,[2,3,4,5]]
```

箭头函数有几个使用注意点。

- 函数体内的this对象，绑定定义时所在的对象，而不是使用时所在的对象。
- 不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。
- 不可以使用arguments对象，该对象在函数体内不存在。

上面三点中，第一点尤其值得注意。this对象的指向是可变的，但是在箭头函数中，它是固定的。下面的代码是一个例子，将this对象绑定定义时所在的对象。

```
var handler = {
  id: "123456",

  init: function() {
    document.addEventListener("click",
      event => this.doSomething(event.type), false);
  },

  doSomething: function(type) {
    console.log("Handling " + type  + " for " + this.id);
  }
};
```

上面代码的init方法中，使用了箭头函数，这导致this绑定handler对象，否则回调函数运行时，this.doSomething这一行会报错，因为此时this指向全局对象。

由于this在箭头函数中被绑定，所以不能用call()、apply()、bind()这些方法去改变this的指向。

长期以来，JavaScript语言的this对象一直是一个令人头痛的问题，在对象方法中使用this，必须非常小心。箭头函数绑定this，很大程度上解决了这个困扰。

箭头函数内部，还可以再使用箭头函数。下面是一个ES5语法的多重嵌套函数。

```
function insert(value) {
  return {into: function (array) {
    return {after: function (afterValue) {
      array.splice(array.indexOf(afterValue) + 1, 0, value);
      return array;
    }};
  }};
}

insert(2).into([1, 3]).after(1); //[1, 2, 3]
```

上面这个函数，可以使用箭头函数改写。

```
let insert = (value) => ({into: (array) => ({after: (afterValue) => {
  array.splice(array.indexOf(afterValue) + 1, 0, value);
  return array;
}})});

insert(2).into([1, 3]).after(1); //[1, 2, 3]
```

下面是一个部署管道机制（pipeline）的例子，即前一个函数的输出是后一个函数的输入。

```
const pipeline = (...funcs) =>
  val => funcs.reduce((a, b) => b(a), val);

const plus1 = a => a + 1;
const mult2 = a => a * 2;
const addThenMult = pipeline(plus1, mult2);

addThenMult(5)
// 12
```

如果觉得上面的写法可读性比较差，也可以采用下面的写法。

```
const plus1 = a => a + 1;
const mult2 = a => a * 2;

mult2(plus1(5))
// 12
```

箭头函数还有一个功能，就是可以很方便地改写λ演算。

```
// λ演算的写法
fix = λf.(λx.f(λv.x(x)(v)))(λx.f(λv.x(x)(v)))

// ES6的写法
var fix = f => (x => f(v => x(x)(v)))
               (x => f(v => x(x)(v)));
```

上面两种写法，几乎是一一对应的。由于λ演算对于计算机科学非常重要，这使得我们可以用ES6作为替代工具，探索计算机科学。

函数绑定

箭头函数可以绑定this对象，大大减少了显式绑定this对象的写法（call、apply、bind）。但是，箭头函数并不适用于所有场合，所以ES7提出了“函数绑定”（function bind）运算符，用来取代call、apply、bind调用。虽然该语法还是ES7的一个提案，但是Babel转码器已经支持。

函数绑定运算符是并排的两个双引号（::），双引号左边是一个对象，右边是一个函数。该运算符会自动将左边的对象，作为上下文环境（即this对象），绑定到右边的函数上面。

```
let log = ::console.log;
// 等同于
var log = console.log.bind(console);

foo::bar;
// 等同于
bar.call(foo);

foo::bar(...arguments);
i// 等同于
bar.apply(foo, arguments);
```

尾调用优化

什么是尾调用？

尾调用（Tail Call）是函数式编程的一个重要概念，本身非常简单，一句话就能说清楚，就是指某个函数的最后一步是调用另一个函数。

```
function f(x){
  return g(x);
}
```

上面代码中，函数f的最后一步是调用函数g，这就叫尾调用。

以下三种情况，都不属于尾调用。

```
// 情况一
function f(x){
  let y = g(x);
  return y;
}

// 情况二
function f(x){
  return g(x) + 1;
}

// 情况三
function f(x){
  g(x);
}
```

上面代码中，情况一是调用函数g之后，还有别的操作，所以不属于尾调用，即使语义完全一样。情况二也

属于调用后还有操作，即使写在一行内。情况三等同于下面的代码。

```
function f(x){
  g(x);
  return undefined;
}
```

尾调用不一定出现在函数尾部，只要是最后一步操作即可。

```
function f(x) {
  if (x > 0) {
    return m(x)
  }
  return n(x);
}
```

上面代码中，函数m和n都属于尾调用，因为它们都是函数f的最后一步操作。

尾调用优化

尾调用之所以与其他调用不同，就在于它的特殊的调用位置。

我们知道，函数调用会在内存形成一个“调用记录”，又称“调用帧”（call frame），保存调用位置和内部变量等信息。如果在函数A的内部调用函数B，那么在A的调用帧上方，还会形成一个B的调用帧。等到B运行结束，将结果返回到A，B的调用帧才会消失。如果函数B内部还调用函数C，那就还有一个C的调用帧，以此类推。所有的调用帧，就形成一个“调用栈”（call stack）。

尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用帧，因为调用位置、内部变量等信息都不会再用到了，只要直接用内层函数的调用帧，取代外层函数的调用帧就可以了。

```
function f() {
  let m = 1;
  let n = 2;
  return g(m + n);
}
f();

// 等同于
function f() {
  return g(3);
}
f();

// 等同于
g(3);
```

上面代码中，如果函数g不是尾调用，函数f就需要保存内部变量m和n的值、g的调用位置等信息。但由于调用g之后，函数f就结束了，所以执行到最后一步，完全可以删除 f(x) 的调用帧，只保留 g(3) 的调用帧。

这就叫做“尾调用优化”（Tail call optimization），即只保留内层函数的调用帧。如果所有函数都是尾调用，那么完全可以做到每次执行时，调用帧只有一项，这将大大节省内存。这就是“尾调用优化”的意义。

尾递归

函数调用自身，称为递归。如果尾调用自身，就称为尾递归。

递归非常耗费内存，因为需要同时保存成千上百个调用帧，很容易发生“栈溢出”错误（stack overflow）。但对于尾递归来说，由于只存在一个调用帧，所以永远不会发生“栈溢出”错误。

```
function factorial(n) {
  if (n === 1) return 1;
  return n * factorial(n - 1);
}

factorial(5) // 120
```

上面代码是一个阶乘函数，计算n的阶乘，最多需要保存n个调用记录，复杂度 O(n) 。

如果改写成尾递归，只保留一个调用记录，复杂度 O(1) 。

```
function factorial(n, total) {
  if (n === 1) return total;
  return factorial(n - 1, n * total);
}

factorial(5, 1) // 120
```

由此可见，“尾调用优化”对递归操作意义重大，所以一些函数式编程语言将其写入了语言规格。ES6也是如此，第一次明确规定，所有 ECMAScript 的实现，都必须部署“尾调用优化”。这就是说，在 ES6 中，只要使用尾递归，就不会发生栈溢出，相对节省内存。

递归函数的改写

尾递归的实现，往往需要改写递归函数，确保最后一步只调用自身。做到这一点的方法，就是把所有用到的内部变量改写成函数的参数。比如上面的例子，阶乘函数 factorial 需要用到一个中间变量 total ，那就把这个中间变量改写成函数的参数。这样做的缺点就是不太直观，第一眼很难看出来，为什么计算5的阶乘，需要传入两个参数5和1？

两个方法可以解决这个问题。方法一是在尾递归函数之外，再提供一个正常形式的函数。

```
function tailFactorial(n, total) {
  if (n === 1) return total;
  return tailFactorial(n - 1, n * total);
}

function factorial(n) {
  return tailFactorial(n, 1);
}

factorial(5) // 120
```

上面代码通过一个正常形式的阶乘函数 factorial ，调用尾递归函数 tailFactorial ，看起来就正常多了。

函数式编程有一个概念，叫做柯里化（currying），意思是将多参数的函数转换成单参数的形式。这里也可以使用柯里化。

```
function currying(fn, n) {
  return function (m) {
    return fn.call(this, m, n);
  };
}

function tailFactorial(n, total) {
  if (n === 1) return total;
  return tailFactorial(n - 1, n * total);
}

const factorial = currying(tailFactorial, 1);

factorial(5) // 120
```

上面代码通过柯里化，将尾递归函数 tailFactorial 变为只接受1个参数的 factorial 。

第二种方法就简单多了，就是采用ES6的函数默认值。

```
function factorial(n, total = 1) {
  if (n === 1) return total;
  return factorial(n - 1, n * total);
}

factorial(5) // 120
```

上面代码中，参数 total 有默认值1，所以调用时不用提供这个值。

总结一下，递归本质上是一种循环操作。纯粹的函数式编程语言没有循环操作命令，所有的循环都用递归实现，这就是为什么尾递归对这些语言极其重要。对于其他支持“尾调用优化”的语言（比如 Lua，ES6），只需要知道循环可以用递归代替，而一旦使用递归，就最好使用尾递归。

Set和Map数据结构

Set

基本用法

ES6提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。

Set本身是一个构造函数，用来生成Set数据结构。

```
var s = new Set();

[2,3,5,4,5,2,2].map(x => s.add(x))

for (i of s) {console.log(i)}
// 2 3 5 4
```

上面代码通过add方法向Set结构加入成员，结果表明Set结构不会添加重复的值。

Set函数可以接受一个数组作为参数，用来初始化。

```
var items = new Set([1,2,3,4,5,5,5,5]);
items.size // 5
```

向Set加入值的时候，不会发生类型转换，所以5和“5”是两个不同的值。Set内部判断两个值是否不同，使用的算法类似于精确相等运算符（===），这意味着，两个对象总是不相等的。唯一的例外是NaN等于自身（精确相等运算符认为NaN不等于自身）。

```
let set = new Set();

set.add({})
set.size // 1

set.add({})
set.size // 2
```

上面代码表示，由于两个空对象不是精确相等，所以它们被视为两个值。

Set实例的属性和方法

Set结构的实例有以下属性。

- Set.prototype.constructor：构造函数，默认就是Set函数。
- Set.prototype.size：返回Set实例的成员总数。

Set实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

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

上面这些属性和方法的实例如下。

```
s.add(1).add(2).add(2);
// 注意2被加入了两次

s.size // 2

s.has(1) // true
s.has(2) // true
s.has(3) // false

s.delete(2);
s.has(2) // false
```

下面是一个对比，看看在判断是否包括一个键上面，Object结构和Set结构的写法不同。

```
// 对象的写法
var properties = {
  "width": 1,
  "height": 1
};

if (properties[someName]) {
  // do something
}

// Set的写法
var properties = new Set();

properties.add("width");
properties.add("height");

if (properties.has(someName)) {
  // do something
}
```

Array.from方法可以将Set结构转为数组。

```
var items = new Set([1, 2, 3, 4, 5]);
var array = Array.from(items);
```

这就提供了一种去除数组的重复元素的方法。

```
function dedupe(array) {
  return Array.from(new Set(array));
}

dedupe([1,1,2,3]) // [1, 2, 3]
```

遍历操作

Set结构的实例有四个遍历方法，可以用于遍历成员。

- keys()：返回一个键名的遍历器
- values()：返回一个键值的遍历器
- entries()：返回一个键值对的遍历器
- forEach()：使用回调函数遍历每个成员

key方法、value方法、entries方法返回的都是遍历器（详见《Iterator对象》一章）。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key方法和value方法的行为完全一致。

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

for ( let item of set.keys() ){
  console.log(item);
}
// red
// green
// blue

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"]
```

上面代码中，entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。

Set结构的实例默认可遍历，它的默认遍历器就是它的values方法。

```
Set.prototype[Symbol.iterator] === Set.prototype.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
```

由于扩展运算符（...）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']);
let arr = [...set];
// ['red', 'green', 'blue']
```

这就提供了另一种便捷的去除数组重复元素的方法。

```
let arr = [3, 5, 2, 2, 5, 5];
let unique = [...new Set(arr)];
// [3, 5, 2]
```

而且，数组的map和filter方法也可以用于Set了。

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

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

因此使用Set，可以很容易地实现并集（Union）和交集（Intersect）。

```
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);

let union = new Set([...a, ...b]);
// [1, 2, 3, 4]

let intersect = new Set([...a].filter(x => b.has(x)));
// [2, 3]
```

Set结构的实例的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, 2, 3]);
set.forEach((value, key) => console.log(value * 2) )
// 2
// 4
// 6
```

上面代码说明，forEach方法的参数就是一个处理函数。该函数的参数依次为键值、键名、集合本身（上例省略了该参数）。另外，forEach方法还可以有第二个参数，表示绑定的this对象。

如果想在遍历操作中　同步改变原来的Set结构，目前没有直接的方法，但有两种变通方法。一种是利用原

Set结构映射出一个新的结构，然后赋值给原来的Set结构；另一种是利用Array.from方法。

```
// 方法一
let set = new Set([1, 2, 3]);
set = new Set([...set].map(val => val * 2));
// set的值是2, 4, 6

// 方法二
let set = new Set([1, 2, 3]);
set = new Set(Array.from(set, val => val * 2));
// set的值是2, 4, 6
```

上面代码提供了两种方法，直接在遍历操作中改变原来的Set结构。

WeakSet

WeakSet结构与Set类似，也是不重复的值的集合。但是，它与Set有两个区别。

首先，WeakSet的成员只能是对象，而不能是其他类型的值。

其次，WeakSet中的对象都是弱引用，即垃圾回收机制不考虑WeakSet对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于WeakSet之中。这个特点意味着，无法引用WeakSet的成员，因此WeakSet是不可遍历的。

```
var ws = new WeakSet();
ws.add(1)
// TypeError: Invalid value used in weak set
```

上面代码试图向WeakSet添加一个数值，结果报错。

WeakSet是一个构造函数，可以使用new命令，创建WeakSet数据结构。

```
var ws = new WeakSet();
```

作为构造函数，WeakSet可以接受一个数组或类似数组的对象作为参数。（实际上，任何具有iterable接口的对象，都可以作为WeakSet的对象。）该数组的所有成员，都会自动成为WeakSet实例对象的成员。

```
var a = [[1,2], [3,4]];
var ws = new WeakSet(a);
```

上面代码中，a是一个数组，它有两个成员，也都是数组。将a作为WeakSet构造函数的参数，a的成员会自动成为WeakSet的成员。

WeakSet结构有以下三个方法。

- **WeakSet.prototype.add(value)**：向WeakSet实例添加一个新成员。
- **WeakSet.prototype.delete(value)**：清除WeakSet实例的指定成员。
- **WeakSet.prototype.has(value)**：返回一个布尔值，表示某个值是否在WeakSet实例之中。

下面是一个例子。

```
var ws = new WeakSet();
var obj = {};
var foo = {};

ws.add(window);
ws.add(obj);

ws.has(window); // true
ws.has(foo);    // false

ws.delete(window);
ws.has(window);    // false
```

WeakSet没有size属性，没有办法遍历它的成员。

```
ws.size // undefined
ws.forEach // undefined

ws.forEach(function(item){ console.log('WeakSet has ' + item)})
// TypeError: undefined is not a function
```

上面代码试图获取size和forEach属性，结果都不能成功。

WeakSet不能遍历，是因为成员都是弱引用，随时可能消失，遍历机制无法保存成员的存在，很可能刚刚遍历结束，成员就取不到了。WeakSet的一个用处，是储存DOM节点，而不用担心这些节点从文档移除时，会引发内存泄漏。

Map

Map结构的目的和基本用法

JavaScript的对象（Object），本质上是键值对的集合（Hash结构），但是只能用字符串当作键。这给它的使用带来了很大的限制。

```
var data = {};
var element = document.getElementById("myDiv");

data[element] = metadata;
data["[Object HTMLDivElement]"] // metadata
```

上面代码原意是将一个DOM节点作为对象data的键，但是由于对象只接受字符串作为键名，所以element被自动转为字符串`[Object HTMLDivElement]`。

为了解决这个问题，ES6提供了Map数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object结构提供了“字符串—值”的对应，Map结构提供了“值—值”的对应，是一种更完善的Hash结构实现。

```
var m = new Map();
var o = {p: "Hello World"};

m.set(o, "content")
m.get(o) // "content"

m.has(o) // true
m.delete(o) // true
m.has(o) // false
```

上面代码使用set方法，将对象o当作m的一个键，然后又使用get方法读取这个键，接着使用delete方法删除了这个键。

作为构造函数，Map也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。

```
var 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"
```

上面代码在新建Map实例时，就指定了两个键name和title。

如果对同一个键多次赋值，后面的值将覆盖前面的值。

```
let map = new Map();
map.set(1, 'aaa');
map.set(1, 'bbb');
map.get(1) // "bbb"
```

上面代码对键1连续赋值两次，后一次的值覆盖前一次的值。

如果读取一个未知的键，则返回undefined。

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

注意，只有对同一个对象的引用，Map结构才将其视为同一个键。这一点要非常小心。

```
var map = new Map();

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

上面代码的set和get方法，表面是针对同一个键，但实际上这是两个值，内存地址是不一样的，因此get方法无法读取该键，返回undefined。

同理，同样的值的两个实例，在Map结构中被视为两个键。

```
var map = new Map();

var k1 = ['a'];
var k2 = ['a'];

map.set(k1, 111);
map.set(k2, 222);

map.get(k1) // 111
map.get(k2) // 222
```

上面代码中，变量k1和k2的值是一样的，但是它们在Map结构中被视为两个键。

由上可知，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，包括0和-0。另外，虽然NaN不严格相等于自身，但Map将其视为同一个键。

```
let map = new Map();

map.set(NaN, 123);
map.get(NaN) // 123

map.set(-0, 123);
map.get(+0) // 123
```

实例的属性和操作方法

Map结构的实例有以下属性和操作方法。

- size：返回成员总数。
- set(key, value)：设置key所对应的键值，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。
- get(key)：读取key对应的键值，如果找不到key，返回undefined。
- has(key)：返回一个布尔值，表示某个键是否在Map数据结构中。
- delete(key)：删除某个键，返回true。如果删除失败，返回false。
- clear()：清除所有成员，没有返回值。

set()方法返回的是Map本身，因此可以采用链式写法。

```
let map = new Map()
  .set(1, 'a')
  .set(2, 'b')
  .set(3, 'c');
```

下面是has()和delete()的例子。

```
var m = new Map();

m.set("edition", 6)        // 键是字符串
m.set(262, "standard")     // 键是数值
m.set(undefined, "nah")    // 键是undefined

var hello = function() {console.log("hello");}
m.set(hello, "Hello ES6!") // 键是函数

m.has("edition")     // true
m.has("years")       // false
m.has(262)           // true
m.has(undefined)     // true
m.has(hello)         // true

m.delete(undefined)
m.has(undefined)       // false

m.get(hello)  // Hello ES6!
m.get("edition")  // 6
```

下面是size属性和clear方法的例子。

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

map.size // 2
map.clear()
map.size // 0
```

遍历方法

Map原生提供三个遍历器。

- keys()：返回键名的遍历器。
- values()：返回键值的遍历器。
- entries()：返回所有成员的遍历器。

下面是使用实例。

```
let 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);
}

// 等同于使用map.entries()
for (let [key, value] of map) {
  console.log(key, value);
}
```

上面代码最后的那个例子，表示Map结构的默认遍历器接口（Symbol.iterator属性），就是entries方法。

```
map[Symbol.iterator] === map.entries
// true
```

Map结构转为数组结构，比较快速的方法是结合使用扩展运算符（...）。

```
let 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方法、filter方法，可以实现Map的遍历和过滤（Map本身没有map和filter方法）。

```
let map0 = new Map()
  .set(1, 'a')
  .set(2, 'b')
  .set(3, 'c');

let map1 = new Map(
  [...map0].filter(([k, v]) => k < 3)
);
// 产生Map结构 {1 => 'a', 2 => 'b'}

let map2 = new Map(
  [...map0].map(([k, v]) => [k * 2, '_' + v])
    );
// 产生Map结构 {2 => '_a', 4 => '_b', 6 => '_c'}
```

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

```
map.forEach(function(value, key, map)) {
  console.log("Key: %s, Value: %s", key, value);
};
```

forEach方法还可以接受第二个参数，用来绑定this。

```
var reporter = {
  report: function(key, value) {
    console.log("Key: %s, Value: %s", key, value);
  }
};

map.forEach(function(value, key, map) {
  this.report(key, value);
}, reporter);
```

上面代码中，forEach方法的回调函数的this，就指向reporter。

WeakMap

WeakMap结构与Map结构基本类似，唯一的区别是它只接受对象作为键名（null除外），不接受原始类型的值作为键名，而且键名所指向的对象，不计入垃圾回收机制。

WeakMap的设计目的在于，键名是对象的弱引用（垃圾回收机制不将该引用考虑在内），所以其所对应的对象可能会被自动回收。当对象被回收后，WeakMap自动移除对应的键值对。典型应用是，一个对应DOM元素的WeakMap结构，当某个DOM元素被清除，其所对应的WeakMap记录就会自动被移除。基本上，WeakMap的专用场合就是，它的键所对应的对象，可能会在将来消失。WeakMap结构有助于防止内存泄漏。

下面是WeakMap结构的一个例子，可以看到用法上与Map几乎一样。

```
var wm = new WeakMap();
var element = document.querySelector(".element");

wm.set(element, "Original");
wm.get(element) // "Original"

element.parentNode.removeChild(element);
element = null;
wm.get(element) // undefined
```

上面代码中，变量wm是一个WeakMap实例，我们将一个DOM节点element作为键名，然后销毁这个节点，element对应的键就自动消失了，再引用这个键名就返回undefined。

WeakMap与Map在API上的区别主要是两个，一是没有遍历操作（即没有key()、values()和entries()方法），也没有size属性；二是无法清空，即不支持clear方法。这与WeakMap的键不被计入引用、被垃圾回收机制忽略有关。因此，WeakMap只有四个方法可用：get()、set()、has()、delete()。

```
var wm = new WeakMap();

wm.size
// undefined

wm.forEach
// undefined
```

前文说过，WeakMap应用的典型场合就是DOM节点作为键名。下面是一个例子。

```
let myElement = document.getElementById('logo');
let myWeakmap = new WeakMap();

myWeakmap.set(myElement, {timesClicked: 0});

myElement.addEventListener('click', function() {
  let logoData = myWeakmap.get(myElement);
  logoData.timesClicked++;
  myWeakmap.set(myElement, logoData);
}, false);
```

上面代码中，myElement是一个DOM节点，每当发生click事件，就更新一下状态。我们将这个状态作为键值放在WeakMap里，对应的键名就是myElement。一旦这个DOM节点删除，该状态就会自动消失，不存在内存泄漏风险。

WeakMap的另一个用处是部署私有属性。

```
let _counter = new WeakMap();
let _action = new WeakMap();

class Countdown {
  constructor(counter, action) {
    _counter.set(this, counter);
    _action.set(this, action);
  }
  dec() {
    let counter = _counter.get(this);
    if (counter < 1) return;
    counter--;
    _counter.set(this, counter);
    if (counter === 0) {
      _action.get(this)();
    }
  }
}

let c = new Countdown(2, () => console.log('DONE'));

c.dec()
c.dec()
// DONE
```

上面代码中，Countdown类的两个内部属性_counter和_action，是实例的弱引用，所以如果删除实例，它们也就随之消失，不会造成内存泄漏。

Iterator和for...of循环

Iterator（遍历器）的概念

JavaScript原有的表示“集合”的数据结构，主要是数组（Array）和对象（Object），ES6又添加了Map和Set。这样就有了四种数据集合，用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是Map，Map的成员是对象。这样就需要一种统一的接口机制，来处理所有不同的数据结构。

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是ES6创造了一种新的遍历命令for...of循环，Iterator接口主要供for...of消费。

Iterator的遍历过程是这样的。

（1）创建一个指针，指向当前数据结构的起始位置。也就是说，遍历器的返回值是一个指针对象。

（2）第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。

（3）第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。

（4）调用指针对象的next方法，直到它指向数据结构的结束位置。

每一次调用next方法，都会返回当前成员的信息，具体来说，就是返回一个包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

下面是一个模拟next方法返回值的例子。

```
function makeIterator(array){
  var nextIndex = 0;
  return {
    next: function(){
      return nextIndex < array.length ?
        {value: array[nextIndex++], done: false} :
        {value: undefined, done: true};
    }
  }
}

var it = makeIterator(['a', 'b']);

it.next() // { value: "a", done: false }
it.next() // { value: "b", done: false }
it.next() // { value: undefined, done: true }
```

上面代码定义了一个makeIterator函数，它的作用就是返回数组的指针对象。对数组`['a', 'b']`执行这个函数，就会返回该数组的指针对象it。

指针对象的next方法，用来移动指针。开始时，指针指向数组的开始位置。然后，每次调用next方法，指针就会指向数组的下一个成员。第一次调用，指向a；第二次调用，指向b。

next方法返回一个对象，表示当前数据成员的信息。这个对象具有value和done两个属性，value属性返回当前位置的成员，done属性是一个布尔值，表示遍历是否结束，即是否还有必要再一次调用next方法。

总之，指针对象具有next方法。调用next方法，就可以遍历事先给定的数据结构。

由于Iterator只是把接口规格加到数据结构之上，所以，遍历器与它所遍历的那个数据结构，实际上是分开的，完全可以写出没有对应数据结构的遍历器，或者说用遍历器模拟出数据结构。下面是一个无限运行的遍历器例子。

```
function idMaker(){
  var index = 0;

  return {
    next: function(){
      return {value: index++, done: false};
    }
  }
}

var it = idMaker();

it.next().value // '0'
it.next().value // '1'
it.next().value // '2'
// ...
```

上面的例子中，遍历器idMaker函数返回的指针对象，并没有对应的数据结构，或者说遍历器自己描述了一个数据结构出来。

在ES6中，有些数据结构原生提供遍历器（比如数组），即不用任何处理，就可以被for...of循环遍历，有些就不行（比如对象）。原因在于，这些数据结构原生部署了System.iterator属性（详见下文），有些没有。凡是部署了System.iterator属性的数据结构，就称为部署了遍历器接口。调用这个接口，就会返回一个指针对象。

如果使用TypeScript的写法，遍历器接口（Iterable）、指针对象（Iterator）和next方法返回值的规格可以描述如下。

```
interface Iterable {
  [System.iterator]() : Iterator,
}

interface Iterator {
  next(value?: any) : IterationResult,
}

interface IterationResult {
  value: any,
  done: boolean,
}
```

数据结构的默认Iterator接口

Iterator接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即for...of循环（详见下文）。当

使用for...of循环遍历某种数据结构时，该循环会自动去寻找Iterator接口。

ES6规定，默认的Iterator接口部署在数据结构的`Symbol.iterator`属性，或者一个数据结构只要具有`Symbol.iterator`属性，就可以认为是“可遍历的”（iterable）。也就是说，调用`Symbol.iterator`方法，就会得到当前数据结构的默认遍历器。`Symbol.iterator`本身是一个表达式，返回Symbol对象的iterator属性，这是一个预定义好的、类型为Symbol的特殊值，所以要放在方括号内（请参考Symbol一节）。

在ES6中，有三类数据结构原生具备Iterator接口：数组、某些类似数组的对象、Set和Map结构。

```
let arr = ['a', 'b', 'c'];
let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }
iter.next() // { value: 'b', done: false }
iter.next() // { value: 'c', done: false }
iter.next() // { value: undefined, done: true }
```

上面代码中，变量arr是一个数组，原生就具有遍历器接口，部署在arr的Symbol.iterator属性上面。所以，调用这个属性，就得到遍历器。

上面提到，原生就部署iterator接口的数据结构有三类，对于这三类数据结构，不用自己写遍历器，for...of循环会自动遍历它们。除此之外，其他数据结构（主要是对象）的Iterator接口，都需要自己在Symbol.iterator属性上面部署，这样才会被for...of循环遍历。

对象（Object）之所以没有默认部署Iterator接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。本质上，遍历器是一种线性处理，对于任何非线性的数据结构，部署遍历器接口，就等于部署一种线性转换。不过，严格地说，对象部署遍历器接口并不是很必要，因为这时对象实际上被当作Map结构使用，ES5没有Map结构，而ES6原生提供了。

一个对象如果要有可被for...of循环调用的Iterator接口，就必须在Symbol.iterator的属性上部署遍历器方法（原型链上的对象具有该方法也可）。

```
class RangeIterator {
  constructor(start, stop) {
    this.value = start;
    this.stop = stop;
  }

  [Symbol.iterator]() { return this; }

  next() {
    var value = this.value;
    if (value < this.stop) {
      this.value++;
      return {done: false, value: value};
    } else {
      return {done: true, value: undefined};
    }
  }
}

function range(start, stop) {
  return new RangeIterator(start, stop);
}

for (var value of range(0, 3)) {
  console.log(value);
}
```

上面代码是一个类部署Iterator接口的写法。Symbol.iterator属性对应一个函数，执行后返回当前对象的遍历器。

下面是通过遍历器实现指针结构的例子。

```
function Obj(value){
  this.value = value;
  this.next = null;
}

Obj.prototype[Symbol.iterator] = function(){

  var iterator = {
    next: next
  };

  var current = this;

  function next(){
    if (current){
      var value = current.value;
      var done = current == null;
      current = current.next;
      return {
        done: done,
        value: value
      }
    } else {
      return {
        done: true
      }
    }
  }
  return iterator;
}

var one = new Obj(1);
var two = new Obj(2);
var three = new Obj(3);

one.next = two;
two.next = three;

for (var i of one){
  console.log(i)
}
// 1
// 2
// 3
```

上面代码首先在构造函数的原型链上部署Symbol.iterator方法，调用该方法会返回遍历器对象iterator，

调用该对象的next方法，在返回一个值的同时，自动将内部指针移到下一个实例。

下面是另一个为对象添加Iterator接口的例子。

```
let obj = {
  data: [ 'hello', 'world' ],
  [Symbol.iterator]() {
    const self = this;
    let index = 0;
    return {
      next() {
        if (index < self.data.length) {
          return {
            value: self.data[index++],
            done: false
          };
        } else {
          return { value: undefined, done: true };
        }
      }
    };
  }
};
```

对于类似数组的对象（存在数值键名和length属性），部署Iterator接口，有一个简便方法，就是`Symbol.iterator`方法直接引用数值的Iterator接口。

```
NodeList.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];
```

如果Symbol.iterator方法返回的不是遍历器，解释引擎将会报错。

```
var obj = {};

obj[Symbol.iterator] = () => 1;

[...obj] // TypeError: [] is not a function
```

上面代码中，变量obj的Symbol.iterator方法返回的不是遍历器，因此报错。

有了遍历器接口，数据结构就可以用for...of循环遍历（详见下文），也可以使用while循环遍历。

```
var $iterator = ITERABLE[Symbol.iterator]();
var $result = $iterator.next();
while (!$result.done) {
  var x = $result.value;
  // ...
  $result = $iterator.next();
}
```

上面代码中，ITERABLE代表某种可遍历的数据结构，$iterator是它的遍历器。遍历器每次移动指针（next方法），都检查一下返回值的done属性，如果遍历还没结束，就移动遍历器的指针到下一步（next方法），不断循环。

调用默认Iterator接口的场合

有一些场合会默认调用iterator接口（即Symbol.iterator方法），除了下文会介绍的for...of循环，还有几个别的场合。

（1）解构赋值

对数组和Set结构进行解构赋值时，会默认调用iterator接口。

```
let set = new Set().add('a').add('b').add('c');

let [x,y] = set;
// x='a'; y='b'

let [first, ...rest] = set;
// first='a'; rest=['b','c'];
```

（2）扩展运算符

扩展运算符（...）也会调用默认的iterator接口。

```
// 例一
var str = 'hello';
[...str] //  ['h','e','l','l','o']

// 例二
let arr = ['b', 'c'];
['a', ...arr, 'd']
// ['a', 'b', 'c', 'd']
```

上面代码的扩展运算符内部就调用iterator接口。

实际上，这提供了一种简便机制，可以将任何部署了iterator接口的数据结构，转为数组。也就是说，只要某个数据结构部署了iterator接口，就可以对它使用扩展运算符，将其转为数组。

```
let arr = [...iterable];
```

（3）其他场合

以下场合也会用到默认的iterator接口，可以查阅相关章节。

- yield*
- Array.from()
- Map(), Set(), WeakMap(), WeakSet()
- Promise.all(), Promise.race()

原生具备Iterator接口的数据结构

《数组的扩展》一章中提到，ES6对数组提供entries()、keys()和values()三个方法，就是返回三个遍历器。

```
var arr = [1, 5, 7];
var arrEntries = arr.entries();

arrEntries.toString()
// "[object Array Iterator]"

arrEntries === arrEntries[Symbol.iterator]()
// true
```

上面代码中，entries方法返回的是一个遍历器（iterator），本质上就是调用了`Symbol.iterator`方法。

字符串是一个类似数组的对象，也原生具有Iterator接口。

```
var someString = "hi";
typeof someString[Symbol.iterator]
// "function"

var iterator = someString[Symbol.iterator]();

iterator.next()  // { value: "h", done: false }
iterator.next()  // { value: "i", done: false }
iterator.next()  // { value: undefined, done: true }
```

上面代码中，调用`Symbol.iterator`方法返回一个遍历器，在这个遍历器上可以调用next方法，实现对于

字符串的遍历。

可以覆盖原生的`Symbol.iterator`方法，达到修改遍历器行为的目的。

```
var str = new String("hi");

[...str] // ["h", "i"]

str[Symbol.iterator] = function() {
  return {
    next: function() {
      if (this._first) {
        this._first = false;
        return { value: "bye", done: false };
      } else {
        return { done: true };
      }
    },
    _first: true
  };
};

[...str] // ["bye"]
str // "hi"
```

上面代码中，字符串str的`Symbol.iterator`方法被修改了，所以扩展运算符（...）返回的值变成了bye，而字符串本身还是hi。

Iterator接口与Generator函数

`Symbol.iterator`方法的最简单实现，还是使用下一章要介绍的Generator函数。

```
var myIterable = {};

myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};
[...myIterable] // [1, 2, 3]

// 或者采用下面的简洁写法

let obj = {
  * [Symbol.iterator]() {
    yield 'hello';
    yield 'world';
  }
};

for (let x of obj) {
  console.log(x);
}
// hello
// world
```

上面代码中，`Symbol.iterator`方法几乎不用部署任何代码，只要用yield命令给出每一步的返回值即可。

遍历器的return()，throw()

遍历器返回的指针对象除了具有next方法，还可以具有return方法和throw方法。其中，next方法是必须部署的，return方法和throw方法是否部署是可选的。

return方法的使用场合是，如果for...of循环提前退出（通常是因为出错，或者有break语句或continue语句），就会调用return方法。如果一个对象在完成遍历前，需要清理或释放资源，就可以部署return方法。

throw方法主要是配合Generator函数使用，一般的遍历器用不到这个方法。请参阅《Generator函数》一章。

for...of循环

ES6借鉴C++、Java、C#和Python语言，引入了for...of循环，作为遍历所有数据结构的统一的方法。一个数据结构只要部署了`Symbol.iterator`方法，就被视为具有iterator接口，就可以用for...of循环遍历它的成员。也就是说，for...of循环内部调用的是数据结构的`Symbol.iterator`方法。

for...of循环可以使用的范围包括数组、Set和Map结构、某些类似数组的对象（比如arguments对象、DOM NodeList对象）、后文的Generator对象，以及字符串。

数组

数组原生具备iterator接口，for...of循环本质上就是调用这个接口产生的遍历器，可以用下面的代码证明。

```
const arr = ['red', 'green', 'blue'];
let iterator  = arr[Symbol.iterator]();

for(let v of arr) {
  console.log(v); // red green blue
}

for(let v of iterator) {
  console.log(v); // red green blue
}
```

上面代码的for...of循环的两种写法是等价的。

for...of循环可以代替数组实例的forEach方法。

```
const arr = ['red', 'green', 'blue'];

arr.forEach(function (element, index) {
  console.log(element); // red green blue
  console.log(index);   // 0 1 2
});
```

JavaScript原有的for...in循环，只能获得对象的键名，不能直接获取键值。ES6提供for...of循环，允许遍历获得键值。

```
var arr = ["a", "b", "c", "d"];

for (a in arr) {
  console.log(a); // 0 1 2 3
}

for (a of arr) {
  console.log(a); // a b c d
}
```

上面代码表明，for...in循环读取键名，for...of循环读取键值。如果要通过for...of循环，获取数组的索引，可以借助数组实例的entries方法和keys方法，参见《数组的扩展》章节。

Set和Map结构

Set和Map结构也原生具有Iterator接口，可以直接使用for...of循环。

```
var engines = Set(["Gecko", "Trident", "Webkit", "Webkit"]);
for (var e of engines) {
  console.log(e);
}
// Gecko
// Trident
// Webkit

var es6 = new Map();
es6.set("edition", 6);
es6.set("committee", "TC39");
es6.set("standard", "ECMA-262");
for (var [name, value] of es6) {
  console.log(name + ": " + value);
}
// edition: 6
// committee: TC39
// standard: ECMA-262
```

上面代码演示了如何遍历Set结构和Map结构。值得注意的地方有两个，首先，遍历的顺序是按照各个成员被添加进数据结构的顺序。其次，Set结构遍历时，返回的是一个值，而Map结构遍历时，返回的是一个数组，该数组的两个成员分别为当前Map成员的键名和键值。

```
let map = new Map().set('a', 1).set('b', 2);
for (let pair of map) {
  console.log(pair);
}
// ['a', 1]
// ['b', 2]

for (let [key, value] of map) {
  console.log(key + ' : ' + value);
}
// a : 1
// b : 2
```

计算生成的数据结构

有些数据结构是在现有数据结构的基础上，计算生成的。比如，ES6的数组、Set、Map都部署了以下三个方法，调用后都返回遍历器。

- entries() 返回一个遍历器，用来遍历 [键名, 键值] 组成的数组。对于数组，键名就是索引值；对于

Set，键名与键值相同。Map结构的iterator接口，默认就是调用entries方法。

- keys() 返回一个遍历器，用来遍历所有的键名。
- values() 返回一个遍历器，用来遍历所有的键值。

这三个方法调用后生成的遍历器，所遍历的都是计算生成的数据结构。

```
let arr = ['a', 'b', 'c'];
for (let pair of arr.entries()) {
  console.log(pair);
}
// [0, 'a']
// [1, 'b']
// [2, 'c']
```

类似数组的对象

类似数组的对象包括好几类。下面是for...of循环用于字符串、DOM NodeList对象、arguments对象的例子。

```
// 字符串
let str = "hello";

for (let s of str) {
  console.log(s); // h e l l o
}

// DOM NodeList对象
let paras = document.querySelectorAll("p");

for (let p of paras) {
  p.classList.add("test");
}

// arguments对象
function printArgs() {
  for (let x of arguments) {
    console.log(x);
  }
}
printArgs('a', 'b');
// 'a'
// 'b'
```

对于字符串来说，for...of循环还有一个特点，就是会正确识别32位UTF-16字符。

```
for (let x of 'a\uD83D\uDC0A') {
  console.log(x);
}
// 'a'
// '\uD83D\uDC0A'
```

并不是所有类似数组的对象都具有iterator接口，一个简便的解决方法，就是使用Array.from方法将其转为数组。

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

// 报错
for (let x of arrayLike) {
  console.log(x);
}

// 正确
for (let x of Array.from(arrayLike)) {
  console.log(x);
}
```

对象

对于普通的对象，for...of结构不能直接使用，会报错，必须部署了iterator接口后才能使用。但是，这样情况下，for...in循环依然可以用来遍历键名。

```
var es6 = {
  edition: 6,
  committee: "TC39",
  standard: "ECMA-262"
};

for (e in es6) {
  console.log(e);
}
// edition
// committee
// standard

for (e of es6) {
  console.log(e);
}
// TypeError: es6 is not iterable
```

上面代码表示，对于普通的对象，for...in循环可以遍历键名，for...of循环会报错。

一种解决方法是，使用Object.keys方法将对象的键名生成一个数组，然后遍历这个数组。

```
for (var key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

在对象上部署iterator接口的代码，参见本章前面部分。一个方便的方法是将数组的Symbol.iterator属性，直接赋值给其他对象的Symbol.iterator属性。比如，想要让for...of循环遍历jQuery对象，只要加上下面这一行就可以了。

```
jQuery.prototype[Symbol.iterator] =
  Array.prototype[Symbol.iterator];
```

另一个方法是使用Generator函数将对象重新包装一下。

```
function* entries(obj) {
  for (let key of Object.keys(obj)) {
    yield [key, obj[key]];
  }
}

for (let [key, value] of entries(obj)) {
  console.log(key, "->", value);
}
// a -> 1
// b -> 2
// c -> 3
```

与其他遍历语法的比较

以数组为例，JavaScript提供多种遍历语法。最原始的写法就是for循环。

```
for (var index = 0; index < myArray.length; index++) {
  console.log(myArray[index]);
}
```

这种写法比较麻烦，因此数组提供内置的forEach方法。

```
myArray.forEach(function (value) {
  console.log(value);
});
```

这种写法的问题在于，无法中途跳出forEach循环，break命令或return命令都不能奏效。

for...in循环可以遍历数组的键名。

```
for (var index in myArray) {
  console.log(myArray[index]);
}
```

for...in循环有几个缺点。

1）数组的键名是数字，但是for...in循环是以字符串作为键名“0”、“1”、“2”等等。

2）for...in循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键。

3）某些情况下，for...in循环会以任意顺序遍历键名。

总之，for...in循环主要是为遍历对象而设计的，不适用于遍历数组。

for...of循环相比上面几种做法，有一些显著的优点。

```
for (let value of myArray) {
  console.log(value);
}
```

- 有着同for...in一样的简洁语法，但是没有for...in那些缺点。
- 不同用于forEach方法，它可以与break、continue和return配合使用。
- 提供了遍历所有数据结构的统一操作接口。

下面是一个使用break语句，跳出for...of循环的例子。

```
for (var n of fibonacci) {
  if (n > 1000)
    break;
  console.log(n);
}
```

上面的例子，会输出斐波纳契数列小于等于1000的项。如果当前项大于1000，就会使用break语句跳出for...of循环。

Generator 函数

简介

基本概念

Generator函数是ES6提供的一种异步编程解决方案，语法行为与传统函数完全不同。本章详细介绍Generator函数的语法和API，它的异步编程应用请看《异步操作》一章。

Generator函数有多种理解角度。从语法上，首先可以把它理解成一个函数的内部状态的遍历器（也就是说，Generator函数是一个状态机）。它每调用一次，就进入下一个内部状态。Generator函数可以控制内部状态的变化，依次遍历这些状态。

形式上，Generator函数是一个普通函数，但是有两个特征。一是，function命令与函数名之间有一个星号；二是，函数体内部使用yield语句，定义遍历器的每个成员，即不同的内部状态（yield语句在英语里的意思就是“产出”）。

```
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

var hw = helloWorldGenerator();
```

上面代码定义了一个Generator函数helloWorldGenerator，它内部有两个yield语句“hello”和“world”，即该函数有三个状态：hello，world和return语句（结束执行）。

然后，Generator函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的遍历器对象（Iterator Object）。

下一步，必须调用遍历器对象的next方法，使得指针移向下一个状态。也就是说，每次调用next方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个yield语句（或return语句）为止。换言之，Generator函数是分段执行的，yield命令是暂停执行的标记，而next方法可以恢复执行。

```
hw.next()
// { value: 'hello', done: false }

hw.next()
// { value: 'world', done: false }

hw.next()
// { value: 'ending', done: true }

hw.next()
// { value: undefined, done: true }
```

上面代码一共调用了四次next方法。

第一次调用，Generator函数开始执行，直到遇到第一个yield语句为止。next方法返回一个对象，它的value属性就是当前yield语句的值hello，done属性的值false，表示遍历还没有结束。

第二次调用，Generator函数从上次yield语句停下的地方，一直执行到下一个yield语句。next方法返回的对象的value属性就是当前yield语句的值world，done属性的值false，表示遍历还没有结束。

第三次调用，Generator函数从上次yield语句停下的地方，一直执行到return语句（如果没有return语句，就执行到函数结束）。next方法返回的对象的value属性，就是紧跟在return语句后面的表达式的值（如果没有return语句，则value属性的值为undefined），done属性的值true，表示遍历已经结束。

第四次调用，此时Generator函数已经运行完毕，next方法返回对象的value属性为undefined，done属性为true。以后再调用next方法，返回的都是这个值。

总结一下，调用Generator函数，返回一个部署了Iterator接口的遍历器对象，用来操作内部指针。以后，每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。value属性表示当前的内部状态的值，是yield语句后面那个表达式的值；done属性是一个布尔值，表示是否遍历结束。

yield语句

由于Generator函数返回的遍历器，只有调用next方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。yield语句就是暂停标志。

遍历器next方法的运行逻辑如下。

（1）遇到yield语句，就暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回的对象的value属性值。

（2）下一次调用next方法时，再继续往下执行，直到遇到下一个yield语句。

（3）如果没有再遇到新的yield语句，就一直运行到函数结束，直到return语句为止，并将return语句后面的表达式的值，作为返回的对象的value属性值。

（4）如果该函数没有return语句，则返回的对象的value属性值为undefined。

需要注意的是，yield语句后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，因此等于为JavaScript提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

```
function* gen{
  yield  123 + 456;
}
```

上面代码中，yield后面的表达式`123 + 456`，不会立即求值，只会在next方法将指针移到这一句时，才会求值。

yield语句与return语句既有相似之处，也有区别。相似之处在于，都能返回紧跟在语句后面的那个表达式的值。区别在于每次遇到yield，函数暂停执行，下一次再从该位置继续向后执行，而return语句不具备位置记忆的功能。一个函数里面，只能执行一次（或者说一个）return语句，但是可以执行多次（或者说多个）yield语句。正常函数只能返回一个值，因为只能执行一次return；Generator函数可以返回一系列的值，因为可以有任意多个yield。从另一个角度看，也可以说Generator生成了一系列的值，这也就是它的名称的来历（在英语中，generator这个词是“生成器”的意思）。

Generator函数可以不用yield语句，这时就变成了一个单纯的暂缓执行函数。

```
function* f() {
  console.log('执行了！')
}

var generator = f();

setTimeout(function () {
  generator.next()
}, 2000);
```

上面代码中，函数f如果是普通函数，在为变量generator赋值时就会执行。但是，函数f是一个Generator函数，就变成只有调用next方法时，函数f才会执行。

另外需要注意，yield语句不能用在普通函数中，否则会报错。

```
(function (){
  yield 1;
})()
// SyntaxError: Unexpected number
```

上面代码在一个普通函数中使用yield语句，结果产生一个句法错误。

下面是另外一个例子。

```
var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function* (a){
  a.forEach(function(item){
    if (typeof item !== 'number'){
      yield* flat(item);
    } else {
      yield item;
    }
  }
};

for (var f of flat(arr)){
  console.log(f);
}
```

上面代码也会产生句法错误，因为forEach方法的参数是一个普通函数，但是在里面使用了yield语句。一种修改方法是改用for循环。

```
var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function* (a){
  var length = a.length;
  for(var i =0;i<length;i++){
    var item = a[i];
    if (typeof item !== 'number'){
      yield* flat(item);
    } else {
      yield item;
    }
  }
};

for (var f of flat(arr)){
  console.log(f);
}
// 1, 2, 3, 4, 5, 6
```

与Iterator的关系

上一章说过，任意一个对象的Symbol.iterator属性，等于该对象的遍历器函数，调用该函数会返回该对象的一个遍历器。

遍历器本身也是一个对象，它的Symbol.iterator属性执行后，返回自身。

```
function* gen(){
  // some code
}

var g = gen();

g[Symbol.iterator]() === g
// true
```

上面代码中，gen是一个Generator函数，调用它会生成一个遍历器g。遍历器g的Symbol.iterator属性是一个遍历器函数，执行后返回它自己。

next方法的参数

yield语句本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield语句的返回值。

```
function* f() {
  for(var i=0; true; i++) {
    var reset = yield i;
    if(reset) { i = -1; }
  }
}

var g = f();

g.next() // { value: 0, done: false }
g.next() // { value: 1, done: false }
g.next(true) // { value: 0, done: false }
```

上面代码先定义了一个可以无限运行的Generator函数f，如果next方法没有参数，每次运行到yield语句，变量reset的值总是undefined。当next方法带一个参数true时，当前的变量reset就被重置为这个参数（即true），因此i会等于-1，下一轮循环就会从-1开始递增。

这个功能有很重要的语法意义。Generator函数从暂停状态到恢复运行，它的上下文状态（context）是不变的。通过next方法的参数，就有办法在Generator函数开始运行之后，继续向函数体内部注入值。也就是说，可以在Generator函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。

再看一个例子。

```
function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var a = foo(5);

a.next() // Object{value:6, done:false}
a.next() // Object{value:NaN, done:false}
a.next() // Object{value:NaN, done:false}
```

上面代码中，第二次运行next方法的时候不带参数，导致y的值等于`2 * undefined`（即NaN），除以3以后还是NaN，因此返回对象的value属性也等于NaN。第三次运行Next方法的时候不带参数，所以z等于undefined，返回对象的value属性等于`5 + NaN + undefined`，即NaN。

如果向next方法提供参数，返回结果就完全不一样了。

```
function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var it = foo(5);

it.next()
// { value:6, done:false }
it.next(12)
// { value:8, done:false }
it.next(13)
// { value:42, done:true }
```

上面代码第一次调用next方法时，返回`x+1`的值6；第二次调用next方法，将上一次yield语句的值设为12，因此y等于24，返回`y / 3`的值8；第三次调用next方法，将上一次yield语句的值设为13，因此z等于13，这时x等于5，y等于24，所以return语句的值等于42。

注意，由于next方法的参数表示上一个yield语句的返回值，所以第一次使用next方法时，不能带有参数。V8引擎直接忽略第一次使用next方法时的参数，只有从第二次使用next方法开始，参数才是有效的。

for...of循环

for...of循环可以自动遍历Generator函数，且此时不再需要调用next方法。

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
  return 6;
}

for (let v of foo()) {
  console.log(v);
}
// 1 2 3 4 5
```

上面代码使用for...of循环，依次显示5个yield语句的值。这里需要注意，一旦next方法的返回对象的done属性为true，for...of循环就会中止，且不包含该返回对象，所以上面代码的return语句返回的6，不包括在for...of循环之中。

下面是一个利用generator函数和for...of循环，实现斐波那契数列的例子。

```
function* fibonacci() {
  let [prev, curr] = [0, 1];
  for (;;) {
    [prev, curr] = [curr, prev + curr];
    yield curr;
  }
}

for (let n of fibonacci()) {
  if (n > 1000) break;
  console.log(n);
}
```

从上面代码可见，使用for...of语句时不需要使用next方法。

throw方法

Generator函数还有一个特点，它可以在函数体外抛出错误，然后在函数体内捕获。

```
var g = function* () {
  while (true) {
    try {
      yield;
    } catch (e) {
      if (e != 'a') throw e;
      console.log('内部捕获', e);
    }
  }
};

var i = g();
i.next();

try {
  i.throw('a');
  i.throw('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 内部捕获 a
// 外部捕获 b
```

上面代码中，遍历器i连续抛出两个错误。第一个错误被Generator函数体内的catch捕获，然后Generator函数执行完成，于是第二个错误被函数体外的catch捕获。

注意，上面代码的错误，是用遍历器的throw方法抛出的，而不是用throw命令抛出的。后者只能被函数体外的catch语句捕获。

```
var g = function* () {
  while (true) {
    try {
      yield;
    } catch (e) {
      if (e != 'a') throw e;
      console.log('内部捕获', e);
    }
  }
};

var i = g();
i.next();

try {
  throw new Error('a');
  throw new Error('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 外部捕获 [Error: a]
```

上面代码之所以只捕获了a，是因为函数体外的catch语句块，捕获了抛出的a错误以后，就不会再继续执行try语句块了。

如果遍历器函数内部没有部署try...catch代码块，那么throw方法抛出的错误，将被外部try...catch代码块捕获。

```
var g = function* () {
  while (true) {
    yield;
    console.log('内部捕获', e);
  }
};

var i = g();
i.next();

try {
  i.throw('a');
  i.throw('b');
} catch (e) {
  console.log('外部捕获', e);
}
// 外部捕获 a
```

上面代码中，遍历器函数g内部，没有部署try...catch代码块，所以抛出的错误直接被外部catch代码块捕获。

如果遍历器函数内部部署了try...catch代码块，那么遍历器的throw方法抛出的错误，不影响下一次遍历，否则遍历直接终止。

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();

try {
  g.throw();
} catch (e) {
  g.next();
}
// hello
```

上面代码只输出hello就结束了，因为第二次调用next方法时，遍历器状态已经变成终止了。但是，如果使用throw方法抛出错误，不会影响遍历器状态。

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();

try {
  throw new Error();
} catch (e) {
  g.next();
}
// hello
// world
```

上面代码中，throw命令抛出的错误不会影响到遍历器的状态，所以两次执行next方法，都取到了正确的操作。

这种函数体内捕获错误的机制，大大方便了对错误的处理。如果使用回调函数的写法，想要捕获多个错误，就不得不为每个函数写一个错误处理语句。

```
foo('a', function (a) {
  if (a.error) {
    throw new Error(a.error);
  }

  foo('b', function (b) {
    if (b.error) {
      throw new Error(b.error);
    }

    foo('c', function (c) {
      if (c.error) {
        throw new Error(c.error);
      }

      console.log(a, b, c);
    });
  });
});
```

使用Generator函数可以大大简化上面的代码。

```
function* g(){
  try {
    var a = yield foo('a');
    var b = yield foo('b');
    var c = yield foo('c');
  } catch (e) {
    console.log(e);
  }

  console.log(a, b, c);
}
```

反过来，Generator函数内抛出的错误，也可以被函数体外的catch捕获。

```
function *foo() {
  var x = yield 3;
  var y = x.toUpperCase();
  yield y;
}

var it = foo();

it.next(); // { value:3, done:false }

try {
  it.next(42);
} catch (err) {
  console.log(err);
}
```

上面代码中，第二个next方法向函数体内传入一个参数42，数值是没有toUpperCase方法的，所以会抛出一个TypeError错误，被函数体外的catch捕获。

一旦Generator执行过程中抛出错误，就不会再执行下去了。如果此后还调用next方法，将返回一个value属性等于undefined、done属性等于true的对象，即JavaScript引擎认为这个Generator已经运行结束了。

```
function* g() {
  yield 1;
  console.log('throwing an exception');
  throw new Error('generator broke!');
  yield 2;
  yield 3;
}

function log(generator) {
  var v;
  console.log('starting generator');
  try {
    v = generator.next();
    console.log('第一次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  try {
    v = generator.next();
    console.log('第二次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  try {
    v = generator.next();
    console.log('第三次运行next方法', v);
  } catch (err) {
    console.log('捕捉错误', v);
  }
  console.log('caller done');
}

log(g());
// starting generator
// 第一次运行next方法 { value: 1, done: false }
// throwing an exception
// 捕捉错误 { value: 1, done: false }
// 第三次运行next方法 { value: undefined, done: true }
// caller done
```

上面代码一共三次运行next方法，第二次运行的时候会抛出错误，然后第三次运行的时候，Generator函数就已经结束了，不再执行下去了。

yield*语句

如果yield命令后面跟的是一个遍历器，需要在yield命令后面加上星号，表明它返回的是一个遍历器。这被

称为yield*语句。

```
let delegatedIterator = (function* () {
  yield 'Hello!';
  yield 'Bye!';
}());

let delegatingIterator = (function* () {
  yield 'Greetings!';
  yield* delegatedIterator;
  yield 'Ok, bye.';
}());

for(let value of delegatingIterator) {
  console.log(value);
}
// "Greetings!
// "Hello!"
// "Bye!"
// "Ok, bye."
```

上面代码中，delegatingIterator是代理者，delegatedIterator是被代理者。由于`yield* delegatedIterator`语句得到的值，是一个遍历器，所以要用星号表示。运行结果就是使用一个遍历器，遍历了多个Genertor函数，有递归的效果。

yield*语句等同于在Generator函数内部，部署一个for...of循环。

```
function* concat(iter1, iter2) {
  yield* iter1;
  yield* iter2;
}

// 等同于

function* concat(iter1, iter2) {
  for (var value of iter1) {
    yield value;
  }
  for (var value of iter2) {
    yield value;
  }
}
```

上面代码说明，yield*不过是for...of的一种简写形式，完全可以用后者替代前者。

再来看一个对比的例子。

```
function* inner() {
  yield 'hello!'
}

function* outer1() {
  yield 'open'
  yield inner()
  yield 'close'
}

var gen = outer1()
gen.next() // -> 'open'
gen.next() // -> a generator
gen.next() // -> 'close'

function* outer2() {
  yield 'open'
  yield* inner()
  yield 'close'
}

var gen = outer2()
gen.next() // -> 'open'
gen.next() // -> 'hello!'
gen.next() // -> 'close'
```

上面例子中，outer2使用了`yield*`，outer1没使用。结果就是，outer1返回一个遍历器，outer2返回该遍历器的内部值。

如果`yield*`后面跟着一个数组，由于数组原生支持遍历器，因此就会遍历数组成员。

```
function* gen(){
  yield* ["a", "b", "c"];
}

gen().next() // { value:"a", done:false }
```

上面代码中，yield命令后面如果不加星号，返回的是整个数组，加了星号就表示返回的是数组的遍历器。

如果被代理的Generator函数有return语句，那么就可以向代理它的Generator函数返回数据。

```
function *foo() {
  yield 2;
  yield 3;
  return "foo";
}

function *bar() {
  yield 1;
  var v = yield *foo();
  console.log( "v: " + v );
  yield 4;
}

var it = bar();

it.next(); //
it.next(); //
it.next(); //
it.next(); // "v: foo"
it.next(); //
```

上面代码在第四次调用next方法的时候，屏幕上会有输出，这是因为函数foo的return语句，向函数bar提供了返回值。

`yield*` 命令可以很方便地取出嵌套数组的所有成员。

```
function* iterTree(tree) {
  if (Array.isArray(tree)) {
    for(let i=0; i < tree.length; i++) {
      yield* iterTree(tree[i]);
    }
  } else {
    yield tree;
  }
}

const tree = [ 'a', ['b', 'c'], ['d', 'e'] ];

for(let x of iterTree(tree)) {
  console.log(x);
}
// a
// b
// c
// d
// e
```

下面是一个稍微复杂的例子，使用yield*语句遍历完全二叉树。

```
// 下面是二叉树的构造函数，
// 三个参数分别是左树、当前节点和右树
function Tree(left, label, right) {
  this.left = left;
  this.label = label;
  this.right = right;
}

// 下面是中序（inorder）遍历函数。
// 由于返回的是一个遍历器，所以要用generator函数。
// 函数体内采用递归算法，所以左树和右树要用yield*遍历
function* inorder(t) {
  if (t) {
    yield* inorder(t.left);
    yield t.label;
    yield* inorder(t.right);
  }
}

// 下面生成二叉树
function make(array) {
  // 判断是否为叶节点
  if (array.length == 1) return new Tree(null, array[0], null);
  return new Tree(make(array[0]), array[1], make(array[2]));
}
let tree = make([[['a'], 'b', ['c']], 'd', [['e'], 'f', ['g']]]);

// 遍历二叉树
var result = [];
for (let node of inorder(tree)) {
  result.push(node);
}

result
// ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

作为对象属性的Generator函数

如果一个对象的属性是Generator函数，可以简写成下面的形式。

```
let obj = {
  * myGeneratorMethod() {
    ...
  }
};
```

上面代码中，myGeneratorMethod属性前面有一个星号，表示这个属性是一个Generator函数。

它的完整形式如下，与上面的写法是等价的。

```
let obj = {
  myGeneratorMethod: function* () {
    // ···
  }
};
```

Generator函数推导

ES7在数组推导的基础上，提出了Generator函数推导（Generator comprehension）。

```
let generator = function* () {
  for (let i = 0; i < 6; i++) {
    yield i;
  }
}

let squared = ( for (n of generator()) n * n );
// 等同于
// let squared = Array.from(generator()).map(n => n * n);

console.log(...squared);
// 0 1 4 9 16 25
```

“推导”这种语法结构，不仅可以用于数组，ES7将其推广到了Generator函数。for...of循环会自动调用遍历器的next方法，将返回值的value属性作为数组的一个成员。

Generator函数推导是对数组结构的一种模拟，它的最大优点是惰性求值，即直到真正用到时才会求值，这样可以保证效率。请看下面的例子。

```
let bigArray = new Array(100000);
for (let i = 0; i < 100000; i++) {
  bigArray[i] = i;
}

let first = bigArray.map(n => n * n)[0];
console.log(first);
```

上面例子遍历一个大数组，但是在真正遍历之前，这个数组已经生成了，占用了系统资源。如果改用Generator函数推导，就能避免这一点。下面代码只在用到时，才会生成一个大数组。

```
let bigGenerator = function* () {
  for (let i = 0; i < 100000; i++) {
    yield i;
  }
}

let squared = ( for (n of bigGenerator()) n * n );

console.log(squared.next());
```

含义

Generator与状态机

Generator是实现状态机的最佳结构。比如，下面的clock函数就是一个状态机。

```
var ticking = true;
var clock = function() {
  if (ticking)
    console.log('Tick!');
  else
    console.log('Tock!');
  ticking = !ticking;
}
```

上面代码的clock函数一共有两种状态（Tick和Tock），每运行一次，就改变一次状态。这个函数如果用Generator实现，就是下面这样。

```
var clock = function*(_) {
  while (true) {
    yield _;
    console.log('Tick!');
    yield _;
    console.log('Tock!');
  }
};
```

上面的Generator实现与ES5实现对比，可以看到少了用来保存状态的外部变量ticking，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅。Generator之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。

Generator与协程

协程（coroutine）是一种程序运行的方式，可以理解成“协作的线程”或“协作的函数”。协程既可以用

单线程实现，也可以用多线程实现。前者是一种特殊的子例程，后者是一种特殊的线程。

（1）协程与子例程的差异

传统的“子例程”（subroutine）采用堆栈式“后进先出”的执行方式，只有当调用的子函数完全执行完毕，才会结束执行父函数。协程与其不同，多个线程（单线程情况下，即多个函数）可以并行执行，但是只有一个线程（或函数）处于正在运行的状态，其他线程（或函数）都处于暂停态（suspended），线程（或函数）之间可以交换执行权。也就是说，一个线程（或函数）执行到一半，可以暂停执行，将执行权交给另一个线程（或函数），等到稍后收回执行权的时候，再恢复执行。这种可以并行执行、交换执行权的线程（或函数），就称为协程。

从实现上看，在内存中，子例程只使用一个栈（stack），而协程是同时存在多个栈，但只有一个栈是在运行状态，也就是说，协程是以多占用内存为代价，实现多任务的并行。

（2）协程与普通线程的差异

不难看出，协程适合用于多任务运行的环境。在这个意义上，它与普通的线程很相似，都有自己的执行上下文、可以分享全局变量。它们的不同之处在于，同一时间可以有多个线程处于运行状态，但是运行的协程只能有一个，其他协程都处于暂停状态。此外，普通的线程是抢先式的，到底哪个线程优先得到资源，必须由运行环境决定，但是协程是合作式的，执行权由协程自己分配。

由于ECMAScript是单线程语言，只能保持一个调用栈。引入协程以后，每个任务可以保持自己的调用栈。这样做的最大好处，就是抛出错误的时候，可以找到原始的调用栈。不至于像异步操作的回调函数那样，一旦出错，原始的调用栈早就结束。

Generator函数是ECMAScript 6对协程的实现，但属于不完全实现。Generator函数被称为“半协程”（semi-coroutine），意思是只有Generator函数的调用者，才能将程序的执行权还给Generator函数。如果是完全执行的协程，任何函数都可以让暂停的协程继续执行。

如果将Generator函数当作协程，完全可以将多个需要互相协作的任务写成Generator函数，它们之间使用yield语句交换控制权。

应用

Generator可以暂停函数执行，返回任意表达式的值。这种特点使得Generator有多种应用场景。

（1）异步操作的同步化表达

Generator函数的暂停执行的效果，意味着可以把异步操作写在yield语句里面，等到调用next方法时再往后执行。这实际上等同于不需要写回调函数了，因为异步操作的后续操作可以放在yield语句下面，反正要等到调用next方法时再执行。所以，Generator函数的一个重要实际意义就是用来处理异步操作，改写回调函数。

```
function* loadUI() {
  showLoadingScreen();
  yield loadUIDataAsynchronously();
 hideLoadingScreen();
}
var loader = loadUI();
// 加载UI
loader.next()

// 卸载UI
loader.next()
```

上面代码表示，第一次调用loadUI函数时，该函数不会执行，仅返回一个遍历器。下一次对该遍历器调用next方法，则会显示Loading界面，并且异步加载数据。等到数据加载完成，再一次使用next方法，则会隐藏Loading界面。可以看到，这种写法的好处是所有Loading界面的逻辑，都被封装在一个函数，按部就班非常清晰。

Ajax是典型的异步操作，通过Generator函数部署Ajax操作，可以用同步的方式表达。

```
function* main() {
  var result = yield request("http://some.url");
  var resp = JSON.parse(result);
    console.log(resp.value);
}

function request(url) {
  makeAjaxCall(url, function(response){
    it.next(response);
  });
}

var it = main();
it.next();
```

上面代码的main函数，就是通过Ajax操作获取数据。可以看到，除了多了一个yield，它几乎与同步操作的写法完全一样。注意，makeAjaxCall函数中的next方法，必须加上response参数，因为yield语句构成的表达式，本身是没有值的，总是等于undefined。

下面是另一个例子，通过Generator函数逐行读取文本文件。

```
function* numbers() {
  let file = new FileReader("numbers.txt");
  try {
    while(!file.eof) {
      yield parseInt(file.readLine(), 10);
    }
  } finally {
    file.close();
  }
}
```

上面代码打开文本文件，使用yield语句可以手动逐行读取文件。

（2）控制流管理

如果有一个多步操作非常耗时，采用回调函数，可能会写成下面这样。

```
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // Do something with value4
      });
    });
  });
});
```

采用Promise改写上面的代码。

```
Q.fcall(step1)
  .then(step2)
  .then(step3)
  .then(step4)
  .then(function (value4) {
    // Do something with value4
  }, function (error) {
    // Handle any error from step1 through step4
  })
  .done();
```

上面代码已经把回调函数，改成了直线执行的形式，但是加入了大量Promise的语法。Generator函数可以进一步改善代码运行流程。

```
function* longRunningTask() {
  try {
    var value1 = yield step1();
    var value2 = yield step2(value1);
    var value3 = yield step3(value2);
    var value4 = yield step4(value3);
    // Do something with value4
  } catch (e) {
    // Handle any error from step1 through step4
  }
}
```

然后，使用一个函数，按次序自动执行所有步骤。

```
scheduler(longRunningTask());

function scheduler(task) {
  setTimeout(function() {
    var taskObj = task.next(task.value);
    // 如果Generator函数未结束，就继续调用
    if (!taskObj.done) {
      task.value = taskObj.value
      scheduler(task);
    }
  }, 0);
}
```

注意，yield语句是同步运行，不是异步运行（否则就失去了取代回调函数的设计目的了）。实际操作中，一般让yield语句返回Promise对象。

```
var Q = require('q');

function delay(milliseconds) {
  var deferred = Q.defer();
  setTimeout(deferred.resolve, milliseconds);
  return deferred.promise;
}

function* f(){
  yield delay(100);
};
```

上面代码使用Promise的函数库Q，yield语句返回的就是一个Promise对象。

多个任务按顺序一个接一个执行时，yield语句可以按顺序排列。多个任务需要并列执行时（比如只有A任

务和B任务都执行完，才能执行C任务），可以采用数组的写法。

```
function* parallelDownloads() {
  let [text1,text2] = yield [
    taskA(),
    taskB()
  ];
  console.log(text1, text2);
}
```

上面代码中，yield语句的参数是一个数组，成员就是两个任务taskA和taskB，只有等这两个任务都完成了，才会接着执行下面的语句。

（3）部署iterator接口

利用Generator函数，可以在任意对象上部署iterator接口。

```
function* iterEntries(obj) {
  let keys = Object.keys(obj);
  for (let i=0; i < keys.length; i++) {
    let key = keys[i];
    yield [key, obj[key]];
  }
}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {
  console.log(key, value);
}

// foo 3
// bar 7
```

上述代码中，myObj是一个普通对象，通过iterEntries函数，就有了iterator接口。也就是说，可以在任意对象上部署next方法。

下面是一个对数组部署Iterator接口的例子，尽管数组原生具有这个接口。

```
function* makeSimpleGenerator(array){
  var nextIndex = 0;

  while(nextIndex < array.length){
    yield array[nextIndex++];
  }
}

var gen = makeSimpleGenerator(['yo', 'ya']);

gen.next().value // 'yo'
gen.next().value // 'ya'
gen.next().done  // true
```

（4）作为数据结构

Generator可以看作是数据结构，更确切地说，可以看作是一个数组结构，因为Generator函数可以返回一系列的值，这意味着它可以对任意表达式，提供类似数组的接口。

```
function *doStuff() {
  yield fs.readFile.bind(null, 'hello.txt');
  yield fs.readFile.bind(null, 'world.txt');
  yield fs.readFile.bind(null, 'and-such.txt');
}
```

上面代码就是依次返回三个函数，但是由于使用了Generator函数，导致可以像处理数组那样，处理这三个返回的函数。

```
for (task of doStuff()) {
  // task是一个函数，可以像回调函数那样使用它
}
```

实际上，如果用ES5表达，完全可以用数组模拟Generator的这种用法。

```
function doStuff() {
  return [
    fs.readFile.bind(null, 'hello.txt'),
    fs.readFile.bind(null, 'world.txt'),
    fs.readFile.bind(null, 'and-such.txt')
  ];
}
```

上面的函数，可以用一模一样的for...of循环处理！两相一比较，就不难看出Generator使得数据或者操作，具备了类似数组的接口。

Promise对象

Promise的含义

Promise在JavaScript语言早有实现，ES6将其写进了语言标准，统一了用法，原生提供了Promise对象。

所谓Promise，就是一个对象，用来传递异步操作的消息。它代表了某个未来才会知道结果的事件（通常是一个异步操作），并且这个事件提供统一的API，可供进一步处理。

Promise对象有以下两个特点。

（1）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）、Resolved（已完成，又称Fulfilled）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。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

（2）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Resolved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。就算改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。第三，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）。

如果某些事件不断地反复发生，一般来说，使用stream模式是比部署Promise更好的选择。

基本用法

ES6规定，Promise对象是一个构造函数，用来生成Promise实例。

下面代码创造了一个Promise实例。

```
var promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由JavaScript引擎提供，不用自己部署。

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实例生成以后，可以用then方法分别指定Resolved状态和Reject状态的回调函数。

```
promise.then(function(value) {
  // success
}, function(value) {
  // failure
});
```

then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为Resolved时调用，第二个回调函数是Promise对象的状态变为Reject时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受Promise对象传出的值作为参数。

下面是一个Promise对象的简单例子。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms, 'done');
  });
}

timeout(100).then((value) => {
  console.log(value);
});
```

上面代码中，timeout方法返回一个Promise实例，表示一段时间以后才会发生的结果。过了指定的时间（ms参数）以后，Promise实例的状态变为Resolved，就会触发then方法绑定的回调函数。

下面是一个用Promise对象实现的Ajax操作的例子。

```
var getJSON = function(url) {
  var promise = new Promise(function(resolve, reject){
    var client = new XMLHttpRequest();
    client.open("GET", url);
    client.onreadystatechange = handler;
    client.responseType = "json";
    client.setRequestHeader("Accept", "application/json");
    client.send();

    function handler() {
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
  });

  return promise;
};

getJSON("/posts.json").then(function(json) {
  console.log('Contents: ' + json);
}, function(error) {
  console.error('出错了', error);
});
```

上面代码中，getJSON是对XMLHttpRequest对象的封装，用于发出一个针对JSON数据的HTTP请求，并且返回一个Promise对象。需要注意的是，在getJSON内部，resolve函数和reject函数调用时，都带有参数。

如果调用resolve函数和reject函数时带有参数，那么它们的参数会被传递给回调函数。reject函数的参数通常是Error对象的实例，表示抛出的错误；resolve函数的参数除了正常的值以外，还可能是另一个Promise实例，表示异步操作的结果有可能是一个值，也有可能是另一个异步操作，比如像下面这样。

```
var p1 = new Promise(function(resolve, reject){
  // ...
});

var p2 = new Promise(function(resolve, reject){
  // ...
  resolve(p1);
})
```

上面代码中，p1和p2都是Promise的实例，但是p2的resolve方法将p1作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时p1的状态就会传递给p2，也就是说，p1的状态决定了p2的状态。如果p1的状态是Pending，那么p2的回调函数就会等待p1的状态改变；如果p1的状态已经是Resolved或者Rejected，那么p2的回调函数将会立刻执行。

Promise.prototype.then()

Promise实例具有then方法，也就是说，then方法是定义在原型对象Promise.prototype上的。它的作用是为Promise实例添加状态改变时的回调函数。前面说过，then方法的第一个参数是Resolved状态的回调函数，第二个参数（可选）是Rejected状态的回调函数。

then方法返回的是一个新的Promise实例（注意，不是原来那个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。

```
getJSON("/posts.json").then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

上面的代码使用then方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

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(function(post) {
  return getJSON(post.commentURL);
}).then(function funcA(comments) {
  console.log("Resolved: ", comments);
}, function funcB(err){
  console.log("Rejected: ", err);
});
```

上面代码中，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funcA，如果状态变为Rejected，就调用funcB。

如果采用箭头函数，上面的代码可以写得更简洁。

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

Promise.prototype.catch()

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

```
getJSON("/posts.json").then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理前一个回调函数运行时发生的错误
  console.log('发生错误！', error);
});
```

上面代码中，getJSON方法返回一个Promise对象，如果该对象状态变为Resolved，则会调用then方法指定的回调函数；如果异步操作抛出错误，状态就会变为Rejected，就会调用catch方法指定的回调函数，处理这个错误。

```
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));
```

下面是一个例子。

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

上面代码中，Promise抛出一个错误，就被catch方法指定的回调函数捕获。

如果Promise状态已经变成resolved，再抛出错误是无效的。

```
var 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
```

上面代码中，Promise在resolve语句后面，再抛出错误，不会被捕获，等于没有抛出。

Promise对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个catch语句捕获。

```
getJSON("/post/1.json").then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});
```

上面代码中，一共有三个Promise对象：一个由getJSON产生，两个由then产生。它们之中任何一个抛出的错误，都会被最后一个catch捕获。

跟传统的try/catch代码块不同的是，如果没有使用catch方法指定错误处理的回调函数，Promise对象抛出的错误不会传递到外层代码，即不会有任何反应。

```
var someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  console.log('everything is great');
});
```

上面代码中，someAsyncThing函数产生的Promise对象会报错，但是由于没有调用catch方法，这个错误不会被捕获，也不会传递到外层代码，导致运行后没有任何输出。

```
var promise = new Promise(function(resolve, reject) {
  resolve("ok");
  setTimeout(function() { throw new Error('test') }, 0)
});
promise.then(function(value) { console.log(value) });
// ok
// Uncaught Error: test
```

上面代码中，Promise指定在下一轮“事件循环”再抛出错误，结果由于没有指定catch语句，就冒泡到最外层，成了未捕获的错误。

Node.js有一个unhandledRejection事件，专门监听未捕获的reject错误。

```
process.on('unhandledRejection', function (err, p) {
  console.error(err.stack)
});
```

上面代码中，unhandledRejection事件的监听函数有两个参数，第一个是错误对象，第二个是报错的Promise实例，它可以用来了解发生错误的环境信息。。

需要注意的是，catch方法返回的还是一个Promise对象，因此后面还可以接着调用then方法。

```
var someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
}).then(function() {
  console.log('carry on');
});
// oh no [ReferenceError: x is not defined]
// carry on
```

上面代码运行完catch方法指定的回调函数，会接着运行后面那个then方法指定的回调函数。

catch方法之中，还能再抛出错误。

```
var someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
  // 下面一行会报错，因为y没有声明
  y + 2;
}).then(function() {
  console.log('carry on');
});
// oh no [ReferenceError: x is not defined]
```

上面代码中，catch方法抛出一个错误，因为后面没有别的catch方法了，导致这个错误不会被捕获，也不会传递到外层。如果改写一下，结果就不一样了。

```
someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
  // 下面一行会报错，因为y没有声明
  y + 2;
}).catch(function(error) {
  console.log('carry on', error);
});
// oh no [ReferenceError: x is not defined]
// carry on [ReferenceError: y is not defined]
```

上面代码中，第二个catch方法用来捕获，前一个catch方法抛出的错误。

Promise.all()

Promise.all方法用于将多个Promise实例，包装成一个新的Promise实例。

```
var p = Promise.all([p1,p2,p3]);
```

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是Promise对象的实例。
（Promise.all方法的参数不一定是数组，但是必须具有iterator接口，且返回的每个成员都是Promise实例。）

p的状态由p1、p2、p3决定，分成两种情况。

（1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

（2）只要p1、p2、p3之中有一个被rejected，p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p的回调函数。

下面是一个具体的例子。

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

Promise.all(promises).then(function(posts) {
  // ...
}).catch(function(reason){
  // ...
});
```

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。

```
var p = Promise.race([p1,p2,p3]);
```

上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p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p的回调函数。

如果Promise.all方法和Promise.race方法的参数，不是Promise实例，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。

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。

```
var jsPromise = Promise.resolve($.ajax('/whatever.json'));
```

上面代码将jQuery生成deferred对象，转为一个新的Promise对象。

如果Promise.resolve方法的参数，不是具有then方法的对象（又称thenable对象），则返回一个新的Promise对象，且它的状态为Resolved。

```
var p = Promise.resolve('Hello');

p.then(function (s){
  console.log(s)
});
// Hello
```

上面代码生成一个新的Promise对象的实例p。由于字符串Hello不属于异步操作（判断方法是它不是具有then方法的对象），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。

Promise.resolve方法允许调用时不带参数。所以，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。

```
var p = Promise.resolve();

p.then(function () {
  // ...
});
```

上面代码的变量p就是一个Promise对象。

如果Promise.resolve方法的参数是一个Promise实例，则会被原封不动地返回。

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。Promise.reject方法的参数reason，会被传递给实例的回调函数。

```
var p = Promise.reject('出错了');

p.then(null, function (s){
  console.log(s)
});
// 出错了
```

上面代码生成一个Promise对象的实例p，状态为rejected，回调函数会立即执行。

Generator函数与Promise的结合

使用Generator函数管理流程，遇到异步操作的时候，通常返回一个Promise对象。

```
function getFoo () {
  return new Promise(function (resolve, reject){
    resolve('foo');
  });
}

var g = function* () {
  try {
    var foo = yield getFoo();
    console.log(foo);
  } catch (e) {
    console.log(e);
  }
};

function run (generator) {
  var it = generator();

  function go(result) {
    if (result.done) return result.value;

    return result.value.then(function (value) {
      return go(it.next(value));
    }, function (error) {
      return go(it.throw(value));
    });
  }

  go(it.next());
}

run(g);
```

上面代码的Generator函数g之中，有一个异步操作getFoo，它返回的就是一个Promise对象。函数run用来处理这个Promise对象，并调用下一个next方法。

async函数

async函数与Promise、Generator函数一样，是用来取代回调函数、解决异步操作的一种方法。它本质上是Generator函数的语法糖。async函数并不属于ES6，而是被列入了ES7，但是traceur、Babel.js、regenerator等转码器已经支持这个功能，转码后立刻就能使用。

async函数的详细介绍，请看《异步操作》一章。

异步操作

异步编程对JavaScript语言太重要。JavaScript只有一根线程，如果没有异步编程，根本没法用，非卡死不可。

ES6诞生以前，异步编程的方法，大概有下面四种。

- 回调函数
- 事件监听
- 发布/订阅
- Promise 对象

ES6将JavaScript异步编程带入了一个全新的阶段。

基本概念

异步

所谓"异步"，简单说就是一个任务分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段。

比如，有一个任务是读取文件进行处理，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。这种不连续的执行，就叫做异步。

相应地，连续的执行就叫做同步。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着。

回调函数

JavaScript语言对异步编程的实现，就是回调函数。所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。它的英语名字callback，直译过来就是"重新调用"。

读取文件进行处理，是这样写的。

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

上面代码中，readFile函数的第二个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了

`/etc/passwd` 这个文件以后，回调函数才会执行。

一个有趣的问题是，为什么Node.js约定，回调函数的第一个参数，必须是错误对象err（如果没有错误，该参数就是null）？原因是执行分成两段，在这两段之间抛出的错误，程序无法捕捉，只能当作参数，传入第二段。

Promise

回调函数本身并没有问题，它的问题出现在多个回调函数嵌套。假定读取A文件之后，再读取B文件，代码如下。

```
fs.readFile(fileA, function (err, data) {
  fs.readFile(fileB, function (err, data) {
    // ...
  });
});
```

不难想象，如果依次读取多个文件，就会出现多重嵌套。代码不是纵向发展，而是横向发展，很快就会乱成一团，无法管理。这种情况就称为“回调函数噩梦”（callback hell）。

Promise就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的横向加载，改成纵向加载。采用Promise，连续读取多个文件，写法如下。

```
var readFile = require('fs-readfile-promise');

readFile(fileA)
.then(function(data){
  console.log(data.toString());
})
.then(function(){
  return readFile(fileB);
})
.then(function(data){
  console.log(data.toString());
})
.catch(function(err) {
  console.log(err);
});
```

上面代码中，我使用了fs-readfile-promise模块，它的作用就是返回一个Promise版本的readFile函数。Promise提供then方法加载回调函数，catch方法捕捉执行过程中抛出的错误。

可以看到，Promise 的写法只是回调函数的改进，使用then方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。

Promise 的最大问题是代码冗余，原来的任务被Promise 包装了一下，不管什么操作，一眼看去都是一堆then，原来的语义变得很不清楚。

那么，有没有更好的写法呢？

Generator函数

协程

传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做"协程"（coroutine），意思是多个线程互相协作，完成异步任务。

协程有点像函数，又有点像线程。它的运行流程大致如下。

- 第一步，协程A开始执行。
- 第二步，协程A执行到一半，进入暂停，执行权转移到协程B。
- 第三步，（一段时间后）协程B交还执行权。
- 第四步，协程A恢复执行。

上面流程的协程A，就是异步任务，因为它分成两段（或多段）执行。

举例来说，读取文件的协程写法如下。

```
function asnycJob() {
  // ...其他代码
  var f = yield readFile(fileA);
  // ...其他代码
}
```

上面代码的函数asyncJob是一个协程，它的奥妙就在其中的yield命令。它表示执行到此处，执行权将交给其他协程。也就是说，yield命令是异步两个阶段的分界线。

协程遇到 yield 命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。它的最大优点，就是代码的写法非常像同步操作，如果去除yield命令，简直一模一样。

Generator函数的概念

Generator函数是协程在ES6的实现，最大特点就是可以交出函数的执行权（即暂停执行）。

整个Generator函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用yield语句注明。Generator函数的执行方法如下。

```
function* gen(x){
  var y = yield x + 2;
  return y;
}

var g = gen(1);
g.next() // { value: 3, done: false }
g.next() // { value: undefined, done: true }
```

上面代码中，调用Generator函数，会返回一个内部指针（即遍历器）g 。这是Generator函数不同于普通函数的另一个地方，即执行它不会返回结果，返回的是指针对象。调用指针g的next方法，会移动内部指针（即执行异步任务的第一段），指向第一个遇到的yield语句，上例是执行到`x + 2`为止。

换言之，next方法的作用是分阶段执行Generator函数。每次调用next方法，会返回一个对象，表示当前阶段的信息（value属性和done属性）。value属性是yield语句后面表达式的值，表示当前阶段的值；done属性是一个布尔值，表示Generator函数是否执行完毕，即是否还有下一个阶段。

Generator函数的数据交换和错误处理

Generator函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。

next方法返回值的value属性，是Generator函数向外输出数据；next方法还可以接受参数，这是向Generator函数体内输入数据。

```
function* gen(x){
  var y = yield x + 2;
  return y;
}

var g = gen(1);
g.next() // { value: 3, done: false }
g.next(2) // { value: 2, done: true }
```

上面代码中，第一个next方法的value属性，返回表达式`x + 2`的值（3）。第二个next方法带有参数2，这个参数可以传入 Generator 函数，作为上个阶段异步任务的返回结果，被函数体内的变量y接收。因此，这一步的 value 属性，返回的就是2（变量y的值）。

Generator 函数内部还可以部署错误处理代码，捕获函数体外抛出的错误。

```
function* gen(x){
  try {
    var y = yield x + 2;
  } catch (e){
    console.log(e);
  }
  return y;
}

var g = gen(1);
g.next();
g.throw ( '出错了' ) ;
// 出错了
```

上面代码的最后一行，Generator函数体外，使用指针对象的throw方法抛出的错误，可以被函数体内的try ...catch代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的。

异步任务的封装

下面看看如何使用 Generator 函数，执行一个真实的异步任务。

```
var fetch = require('node-fetch');

function* gen(){
  var url = 'https://api.github.com/users/github';
  var result = yield fetch(url);
  console.log(result.bio);
}
```

上面代码中，Generator函数封装了一个异步操作，该操作先读取一个远程接口，然后从JSON格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了yield命令。

执行这段代码的方法如下。

```
var g = gen();
var result = g.next();

result.value.then(function(data){
  return data.json();
}).then(function(data){
  g.next(data);
});
```

上面代码中，首先执行Generator函数，获取遍历器对象，然后使用next 方法（第二行），执行异步任务

的第一阶段。由于Fetch模块返回的是一个Promise对象，因此要用then方法调用下一个next 方法。

可以看到，虽然 Generator 函数将异步操作表示得很简洁，但是流程管理却不方便（即何时执行第一阶段、何时执行第二阶段）。

Thunk函数

参数的求值策略

Thunk函数早在上个世纪60年代就诞生了。

那时，编程语言刚刚起步，计算机学家还在研究，编译器怎么写比较好。一个争论的焦点是"求值策略"，即函数的参数到底应该何时求值。

```
var x = 1;

function f(m){
  return m * 2;
}

f(x + 5)
```

上面代码先定义函数f，然后向它传入表达式`x + 5`。请问，这个表达式应该何时求值？

一种意见是"传值调用"（call by value），即在进入函数体之前，就计算`x + 5`的值（等于6），再将这个值传入函数f 。C语言就采用这种策略。

```
f(x + 5)
// 传值调用时，等同于
f(6)
```

另一种意见是"传名调用"（call by name），即直接将表达式`x + 5`传入函数体，只在用到它的时候求值。Hskell语言采用这种策略。

```
f(x + 5)
// 传名调用时，等同于
(x + 5) * 2
```

传值调用和传名调用，哪一种比较好？回答是各有利弊。传值调用比较简单，但是对参数求值的时候，实际上还没用到这个参数，有可能造成性能损失。

```
function f(a, b){
  return b;
}

f(3 * x * x - 2 * x - 1, x);
```

上面代码中，函数f的第一个参数是一个复杂的表达式，但是函数体内根本没用到。对这个参数求值，实际上是不必要的。因此，有一些计算机学家倾向于"传名调用"，即只在执行时求值。

Thunk函数的含义

编译器的"传名调用"实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做Thunk函数。

```
function f(m){
  return m * 2;
}

f(x + 5);

// 等同于

var thunk = function () {
  return x + 5;
};

function f(thunk){
  return thunk() * 2;
}
```

上面代码中，函数f的参数`x + 5`被一个函数替换了。凡是用到原参数的地方，对`Thunk`函数求值即可。这就是Thunk函数的定义，它是"传名调用"的一种实现策略，用来替换某个表达式。

JavaScript语言的Thunk函数

JavaScript语言是传值调用，它的Thunk函数含义有所不同。在JavaScript语言中，Thunk函数替换的不是表达式，而是多参数函数，将其替换成单参数的版本，且只接受回调函数作为参数。

```
// 正常版本的readFile（多参数版本）
fs.readFile(fileName, callback);

// Thunk版本的readFile（单参数版本）
var readFileThunk = Thunk(fileName);
readFileThunk(callback);

var Thunk = function (fileName){
  return function (callback){
    return fs.readFile(fileName, callback);
  };
};
```

上面代码中，fs模块的readFile方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。这个单参数版本，就叫做Thunk函数。

任何函数，只要参数有回调函数，就能写成Thunk函数的形式。下面是一个简单的Thunk函数转换器。

```
var Thunk = function(fn){
  return function (){
    var args = Array.prototype.slice.call(arguments);
    return function (callback){
      args.push(callback);
      return fn.apply(this, args);
    }
  };
};
```

使用上面的转换器，生成`fs.readFile`的Thunk函数。

```
var readFileThunk = Thunk(fs.readFile);
readFileThunk(fileA)(callback);
```

Thunkify模块

生产环境的转换器，建议使用Thunkify模块。

首先是安装。

```
$ npm install thunkify
```

使用方式如下。

```
var thunkify = require('thunkify');
var fs = require('fs');

var read = thunkify(fs.readFile);
read('package.json')(function(err, str){
  // ...
});
```

Thunkify的源码与上一节那个简单的转换器非常像。

```
function thunkify(fn){
  return function(){
    var args = new Array(arguments.length);
    var ctx = this;

    for(var i = 0; i < args.length; ++i) {
      args[i] = arguments[i];
    }

    return function(done){
      var called;

      args.push(function(){
        if (called) return;
        called = true;
        done.apply(null, arguments);
      });

      try {
        fn.apply(ctx, args);
      } catch (err) {
        done(err);
      }
    }
  }
};
```

它的源码主要多了一个检查机制，变量called确保回调函数只运行一次。这样的设计与下文的Generator函数相关。请看下面的例子。

```
function f(a, b, callback){
  var sum = a + b;
  callback(sum);
  callback(sum);
}

var ft = thunkify(f);
ft(1, 2)(console.log);
// 3
```

上面代码中，由于thunkify只允许回调函数执行一次，所以只输出一行结果。

Generator 函数的流程管理

你可能会问， Thunk函数有什么用？回答是以前确实没什么用，但是ES6有了Generator函数，Thunk函数现在可以用于Generator函数的自动流程管理。

以读取文件为例。下面的Generator函数封装了两个异步操作。

```
var fs = require('fs');
var thunkify = require('thunkify');
var readFile = thunkify(fs.readFile);

var gen = function* (){
  var r1 = yield readFile('/etc/fstab');
  console.log(r1.toString());
  var r2 = yield readFile('/etc/shells');
  console.log(r2.toString());
};
```

上面代码中，yield命令用于将程序的执行权移出Generator函数，那么就需要一种方法，将执行权再交还给Generator函数。

这种方法就是Thunk函数，因为它可以在回调函数里，将执行权交还给Generator函数。为了便于理解，我们先看如何手动执行上面这个Generator函数。

```
var g = gen();

var r1 = g.next();
r1.value(function(err, data){
  if (err) throw err;
  var r2 = g.next(data);
  r2.value(function(err, data){
    if (err) throw err;
    g.next(data);
  });
});
```

上面代码中，变量g是Generator函数的内部指针，表示目前执行到哪一步。next方法负责将指针移动到下一步，并返回该步的信息（value属性和done属性）。

仔细查看上面的代码，可以发现Generator函数的执行过程，其实是将同一个回调函数，反复传入next方法的value属性。这使得我们可以用递归来自动完成这个过程。

Thunk函数的自动流程管理

Thunk函数真正的威力，在于可以自动执行Generator函数。下面就是一个基于Thunk函数的Generator执行器。

```
function run(fn) {
  var gen = fn();

  function next(err, data) {
    var result = gen.next(data);
    if (result.done) return;
    result.value(next);
  }

  next();
}

run(gen);
```

上面代码的run函数，就是一个Generator函数的自动执行器。内部的next函数就是Thunk的回调函数。next函数先将指针移到Generator函数的下一步（gen.next方法），然后判断Generator函数是否结束（result.done 属性），如果没结束，就将next函数再传入Thunk函数（result.value属性），否则就直接退出。

有了这个执行器，执行Generator函数方便多了。不管有多少个异步操作，直接传入run函数即可。当然，前提是每一个异步操作，都要是Thunk函数，也就是说，跟在yield命令后面的必须是Thunk函数。

```
var gen = function* (){
  var f1 = yield readFile('fileA');
  var f2 = yield readFile('fileB');
  // ...
  var fn = yield readFile('fileN');
};

run(gen);
```

上面代码中，函数gen封装了n个异步的读取文件操作，只要执行run函数，这些操作就会自动完成。这样一来，异步操作不仅可以写得像同步操作，而且一行代码就可以执行。

Thunk函数并不是Generator函数自动执行的唯一方案。因为自动执行的关键是，必须有一种机制，自动控制Generator函数的流程，接收和交还程序的执行权。回调函数可以做到这一点，Promise 对象也可以做到这一点。

co模块

基本用法

co模块是著名程序员TJ Holowaychuk于2013年6月发布的一个小工具，用于Generator函数的自动执行。

比如，有一个Generator函数，用于依次读取两个文件。

```
var gen = function* (){
  var f1 = yield readFile('/etc/fstab');
  var f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

co模块可以让你不用编写Generator函数的执行器。

```
var co = require('co');
co(gen);
```

上面代码中，Generator函数只要传入co函数，就会自动执行。

co函数返回一个Promise对象，因此可以用then方法添加回调函数。

```
co(gen).then(function (){
  console.log('Generator 函数执行完成');
})
```

上面代码中，等到Generator函数执行结束，就会输出一行提示。

co模块的原理

为什么co可以自动执行Generator函数？

前面说过，Generator就是一个异步操作的容器。它的自动执行需要一种机制，当异步操作有了结果，能够自动交回执行权。

两种方法可以做到这一点。

（1）回调函数。将异步操作包装成Thunk函数，在回调函数里面交回执行权。

（2）Promise 对象。将异步操作包装成Promise对象，用then方法交回执行权。

co模块其实就是将两种自动执行器（Thunk函数和Promise对象），包装成一个模块。使用co的前提条件是，Generator函数的yield命令后面，只能是Thunk函数或Promise对象。

上一节已经介绍了基于Thunk函数的自动执行器。下面来看，基于Promise对象的自动执行器。这是理解co模块必须的。

基于Promise对象的自动执行

还是沿用上面的例子。首先，把fs模块的readFile方法包装成一个Promise对象。

```
var fs = require('fs');

var readFile = function (fileName){
  return new Promise(function (resolve, reject){
    fs.readFile(fileName, function(error, data){
      if (error) reject(error);
      resolve(data);
    });
  });
};

var gen = function* (){
  var f1 = yield readFile('/etc/fstab');
  var f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

然后，手动执行上面的Generator函数。

```
var g = gen();

g.next().value.then(function(data){
  g.next(data).value.then(function(data){
    g.next(data);
  });
})
```

手动执行其实就是用then方法，层层添加回调函数。理解了这一点，就可以写出一个自动执行器。

```
function run(gen){
  var g = gen();

  function next(data){
    var result = g.next(data);
    if (result.done) return result.value;
    result.value.then(function(data){
      next(data);
    });
  }

  next();
}

run(gen);
```

上面代码中，只要Generator函数还没执行到最后一步，next函数就调用自身，以此实现自动执行。

co模块的源码

co就是上面那个自动执行器的扩展，它的源码只有几十行，非常简单。

首先，co函数接受Generator函数作为参数，返回一个 Promise 对象。

```
function co(gen) {
  var ctx = this;

  return new Promise(function(resolve, reject) {
  });
}
```

在返回的Promise对象里面，co先检查参数gen是否为Generator函数。如果是，就执行该函数，得到一个内部指针对象；如果不是就返回，并将Promise对象的状态改为resolved。

```
function co(gen) {
  var ctx = this;

  return new Promise(function(resolve, reject) {
    if (typeof gen === 'function') gen = gen.call(ctx);
    if (!gen || typeof gen.next !== 'function') return resolve(gen);
  });
}
```

接着，co将Generator函数的内部指针对象的next方法，包装成onFulefilled函数。这主要是为了能够捕捉抛出的错误。

```
function co(gen) {
  var ctx = this;

  return new Promise(function(resolve, reject) {
    if (typeof gen === 'function') gen = gen.call(ctx);
    if (!gen || typeof gen.next !== 'function') return resolve(gen);

    onFulfilled();
    function onFulfilled(res) {
      var ret;
      try {
        ret = gen.next(res);
      } catch (e) {
        return reject(e);
      }
      next(ret);
    }
  });
}
```

最后，就是关键的next函数，它会反复调用自身。

```
function next(ret) {
  if (ret.done) return resolve(ret.value);
  var value = toPromise.call(ctx, ret.value);
  if (value && isPromise(value)) return value.then(onFulfilled, onRejected);
  return onRejected(new TypeError('You may only yield a function, promise, generator, array, o
r object, '
    + 'but the following object was passed: "' + String(ret.value) + '"'));
}
```

上面代码中，next 函数的内部代码，一共只有四行命令。

- 第一行，检查当前是否为 Generator 函数的最后一步，如果是就返回。
- 第二行，确保每一步的返回值，是 Promise 对象。
- 第三行，使用 then 方法，为返回值加上回调函数，然后通过 onFulfilled 函数再次调用 next 函数。
- 第四行，在参数不符合要求的情况下（参数非 Thunk 函数和 Promise 对象），将 Promise 对象的状态改为 rejected，从而终止执行。

处理并发的异步操作

co支持并发的异步操作，即允许某些操作同时进行，等到它们全部完成，才进行下一步。

这时，要把并发的操作都放在数组或对象里面，跟在yield语句后面。

```
// 数组的写法
co(function* () {
  var res = yield [
    Promise.resolve(1),
    Promise.resolve(2)
  ];
  console.log(res);
}).catch(onerror);

// 对象的写法
co(function* () {
  var res = yield {
    1: Promise.resolve(1),
    2: Promise.resolve(2),
  };
  console.log(res);
}).catch(onerror);
```

下面是另一个例子。

```
co(function* () {
  var values = [n1, n2, n3];
  yield values.map(somethingAsync);
});

function* somethingAsync(x) {
  // do something async
  return y
}
```

上面的代码允许并发三个somethingAsync异步操作，等到它们全部完成，才会进行下一步。

async函数

含义

async 函数是什么？一句话，async函数就是Generator函数的语法糖。

前文有一个Generator函数，依次读取两个文件。

```
var fs = require('fs');

var readFile = function (fileName){
  return new Promise(function (resolve, reject){
    fs.readFile(fileName, function(error, data){
      if (error) reject(error);
      resolve(data);
    });
  });
};

var gen = function* (){
  var f1 = yield readFile('/etc/fstab');
  var f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

写成 async 函数，就是下面这样。

```
var asyncReadFile = async function (){
  var f1 = await readFile('/etc/fstab');
  var f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

一比较就会发现，async函数就是将Generator函数的星号（*）替换成async，将yield替换成await，仅此而已。

async 函数对 Generator 函数的改进，体现在以下三点。

（1）内置执行器。Generator函数的执行必须靠执行器，所以才有了co模块，而async 函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，只要一行。

```
var result = asyncReadFile();
```

（2）更好的语义。async和await，比起星号和yield，语义更清楚了。async表示函数里有异步操作，await 表示紧跟在后面的表达式需要等待结果。

（3）更广的适用性。 co模块约定，yield命令后面只能是Thunk函数或Promise对象，而async函数的await命令后面，可以跟Promise对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

async函数的实现

async 函数的实现，就是将 Generator 函数和自动执行器，包装在一个函数里。

```
async function fn(args){
  // ...
}

// 等同于

function fn(args){
  return spawn(function*() {
    // ...
  });
}
```

所有的 async 函数都可以写成上面的第二种形式，其中的 spawn 函数就是自动执行器。

下面给出 spawn 函数的实现，基本就是前文自动执行器的翻版。

```
function spawn(genF) {
  return new Promise(function(resolve, reject) {
    var gen = genF();
    function step(nextF) {
      try {
        var next = nextF();
      } catch(e) {
        return reject(e);
      }
      if(next.done) {
        return resolve(next.value);
      }
      Promise.resolve(next.value).then(function(v) {
        step(function() { return gen.next(v); });
      }, function(e) {
        step(function() { return gen.throw(e); });
      });
    }
    step(function() { return gen.next(undefined); });
  });
}
```

async 函数是非常新的语法功能，新到都不属于 ES6，而是属于 ES7。目前，它仍处于提案阶段，但是转码器 Babel 和 regenerator 都已经支持，转码后就能使用。

async 函数的用法

同Generator函数一样，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，等到触发的异步操作完成，再接着执行函数体内后面的语句。

下面是一个例子。

```
async function getStockPriceByName(name) {
  var symbol = await getStockSymbol(name);
  var stockPrice = await getStockPrice(symbol);
  return stockPrice;
}

getStockPriceByName('goog').then(function (result){
  console.log(result);
});
```

上面代码是一个获取股票报价的函数，函数前面的async关键字，表明该函数内部有异步操作。调用该函数时，会立即返回一个Promise对象。

下面的例子，指定多少毫秒后输出一个值。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

async function asyncPrint(value, ms) {
  await timeout(ms);
  console.log(value)
}

asyncPrint('hello world', 50);
```

上面代码指定50毫秒以后，输出"hello world"。

注意点

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);
  };
}
```

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);
  });
}
```

上面代码会报错，因为await用在普通函数之中了。但是，如果将forEach方法的参数改成async函数，也有问题。

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

  // 可能得到错误结果
  docs.forEach(async function (doc) {
    await db.post(doc);
  });
}
```

上面代码可能不会正常工作，原因是这时三个`db.post`操作将是并发执行，也就是同时执行，而不是继发执行。正确的写法是采用for循环。

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

  for (let doc of docs) {
    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。

```
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);
}
```

ES6将await增加为保留字。使用这个词作为标识符，在ES5是合法的，在ES6将抛出SyntaxError。

与Promise、Generator的比较

我们通过一个例子，来看Async函数与Promise、Generator函数的区别。

假定某个DOM元素上面，部署了一系列的动画，前一个动画结束，才能开始后一个。如果当中有一个动画出错，就不再往下执行，返回上一个成功执行的动画的返回值。

首先是Promise的写法。

```
function chainAnimationsPromise(elem, animations) {

  // 变量ret用来保存上一个动画的返回值
  var ret = null;

  // 新建一个空的Promise
  var p = Promise.resolve();

  // 使用then方法，添加所有动画
  for(var anim in animations) {
    p = p.then(function(val) {
      ret = val;
      return anim(elem);
    })
  }

  // 返回一个部署了错误捕捉机制的Promise
  return p.catch(function(e) {
    /* 忽略错误，继续执行 */
  }).then(function() {
    return ret;
  });

}
```

虽然Promise的写法比回调函数的写法大大改进，但是一眼看上去，代码完全都是Promise的API（then、catch等等），操作本身的语义反而不容易看出来。

接着是Generator函数的写法。

```
function chainAnimationsGenerator(elem, animations) {

  return spawn(function*() {
    var ret = null;
    try {
      for(var anim of animations) {
        ret = yield anim(elem);
      }
    } catch(e) {
      /* 忽略错误，继续执行 */
    }
      return ret;
  });

}
```

上面代码使用Generator函数遍历了每个动画，语义比Promise写法更清晰，用户定义的操作全部都出现在spawn函数的内部。这个写法的问题在于，必须有一个任务运行器，自动执行Generator函数，上面代码的spawn函数就是自动执行器，它返回一个Promise对象，而且必须保证yield语句后面的表达式，必须返回一个Promise。

最后是Async函数的写法。

```
async function chainAnimationsAsync(elem, animations) {
  var ret = null;
  try {
    for(var anim of animations) {
      ret = await anim(elem);
    }
  } catch(e) {
    /* 忽略错误，继续执行 */
  }
  return ret;
}
```

可以看到Async函数的实现最简洁，最符合语义，几乎没有语义不相关的代码。它将Generator写法中的自动执行器，改在语言层面提供，不暴露给用户，因此代码量最少。如果使用Generator写法，自动执行器需要用户自己提供。

Class

Class基本语法

（1）概述

JavaScript语言的传统方法是通过构造函数，定义并生成新对象。下面是一个例子。

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

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

上面这种写法跟传统的面向对象语言（比如C++和Java）差异很大，很容易让新学习这门语言的程序员感到困惑。

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。基本上，ES6的class可以看作只是一个语法糖，它的绝大部分功能，ES5都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用ES6的“类”改写，就是下面这样。

```
//定义类
class Point {

  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '('+this.x+', '+this.y+')';
  }

}
```

上面代码定义了一个“类”，可以看到里面有一个constructor方法，这就是构造方法，而this关键字则代表实例对象。也就是说，ES5的构造函数Point，对应ES6的Point类的构造方法。

Point类除了构造方法，还定义了一个toString方法。注意，定义“类”的方法的时候，前面不需要加上function这个保留字，直接把函数定义放进去了就可以了。

ES6的类，完全可以看作构造函数的另一种写法。

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

typeof Point // "function"
```

上面代码表明，类的数据类型就是函数。

构造函数的prototype属性，在ES6的“类”上面继续存在。事实上，除了constructor方法以外，类的方法都定义在类的prototype属性上面。

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

  toString(){
    // ...
  }

  toValue(){
    // ...
  }
}

// 等同于

Point.prototype = {
  toString(){},
  toValue(){}
}
```

由于类的方法（除constructor以外）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(){
    // ...
  }
}

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
```

另外，类的内部所有定义的方法，都是不可枚举的（enumerable）。

```
class Point {
  constructor(x, y) {
    // ...
  }

  toString() {
    // ...
  }
}

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。

```
var Point = function (x, y){
  // ...
}

Point.prototype.toString = function() {
  // ...
}

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。

类的属性名，可以采用表达式。

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

  [methodName]() {
    // ...
  }
}
```

上面代码中，Square类的方法名getArea，是从表达式得到的。

（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。

```
constructor() {}
```

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。

（3）实例对象

生成实例对象的写法，与ES5完全一样，也是使用new命令。如果忘记加上new，像函数那样调用Class，将会报错。

```
// 报错
var point = Point(2, 3);

// 正确
var point = new Point(2, 3);
```

与ES5一样，实例的属性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）。

```
//定义类
class Point {

  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '('+this.x+', '+this.y+')';
  }

}

var 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变量上），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。这些都与ES5的行为保持一致。

与ES5一样，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
```

上面代码中，p1和p2都是Point的实例，它们的原型都是Point，所以**proto**属性是相等的。

这也意味着，可以通过实例的**proto**属性为Class添加方法。

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

p1.__proto__.printName = function () { return 'Oops' };

p1.printName() // "Oops"
p2.printName() // "Oops"

var p3 = new Point(4,2);
p3.printName() // "Oops"
```

上面代码在p1的原型上添加了一个printName方法，由于p1的原型就是p2的原型，因此p2也可以调用这个方法。而且，此后新建的实例p3也可以调用这个方法。这意味着，使用实例的**proto**属性改写原型，必须相当谨慎，不推荐使用，因为这会改变Class的原始定义，影响到所有实例。

（4）name属性

由于本质上，ES6的Class只是ES5的构造函数的一层包装，所以函数的许多特性都被Class继承，包括name属性。

```
class Point {}
Point.name // "Point"
```

name属性总是返回紧跟在class关键字后面的类名。

（5）Class表达式

与函数一样，Class也可以使用表达式的形式定义。

```
const MyClass = class Me {
  getClassName() {
    return Me.name;
  }
};
```

上面代码使用表达式定义了一个类。需要注意的是，这个类的名字是MyClass而不是Me，Me只在Class的内部代码可用，指代当前类。

```
let inst = new MyClass();
inst.getClassName() // Me
Me.name // ReferenceError: Me is not defined
```

上面代码表示，Me只在Class内部有定义。

如果Class内部没用到的话，可以省略Me，也就是可以写成下面的形式。

```
const MyClass = class { /* ... */ };
```

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"
```

上面代码中，person是一个立即执行的Class的实例。

（6）不存在变量提升

Class不存在变量提升（hoist），这一点与ES5完全不同。

```
new Foo(); // ReferenceError
class Foo {}
```

上面代码中，Foo类使用在前，定义在后，这样会报错，因为ES6不会把变量声明提升到代码头部。这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

```
{
  let Foo = class {};
  class Bar extends Foo {
  }
}
```

如果存在Class的提升，上面代码将报错，因为let命令也是不提升的。

（7）严格模式

类和模块的内部，默认就是严格模式，所以不需要使用`use strict`指定运行模式。只要你的代码写在类或模块之中，就只有严格模式可用。

考虑到未来所有的代码，其实都是运行在模块之中，所以ES6实际上把整个语言升级到了严格模式。

Class的继承

基本用法

Class之间可以通过extends关键字，实现继承，这比ES5的通过修改原型链实现继承，要清晰和方便很多。

```
class ColorPoint extends Point {}
```

上面代码定义了一个ColorPoint类，该类通过extends关键字，继承了Point类的所有属性和方法。但是由于没有部署任何代码，所以这两个类完全一样，等于复制了一个Point类。下面，我们在ColorPoint内部加上代码。

```
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()
  }

}
```

上面代码中，constructor方法和toString方法之中，都出现了super关键字，它指代父类的实例（即父类的this对象）。

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

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

class ColorPoint extends Point {
  constructor() {
  }
}

let cp = new ColorPoint(); // ReferenceError
```

上面代码中，ColorPoint继承了父类Point，但是它的构造函数没有调用super方法，导致新建实例时报错。

ES5的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（`Parent.apply(this)`）。ES6的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

如果子类没有定义constructor方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有constructor方法。

```
constructor(...args) {
  super(...args);
}
```

另一个需要注意的地方是，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，是基于对父类实例加工，只有super方法才能返回父类实例。

```
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方法之后就是正确的。

下面是生成子类实例的代码。

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

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

上面代码中，实例对象cp同时是ColorPoint和Point两个类的实例，这与ES5的行为完全一致。

类的prototype属性和proto属性

在ES5中，每一个对象都有`__proto__`属性，指向对应的构造函数的prototype属性。Class作为构造函数

的语法糖，同时有prototype属性和__proto__属性，因此同时存在两条继承链。

（1）子类的__proto__属性，表示构造函数的继承，总是指向父类。

（2）子类prototype属性的__proto__属性，表示方法的继承，总是指向父类的prototype属性。

```
class A {
}

class B extends A {
}

B.__proto__ === A // true
B.prototype.__proto__ === A.prototype // true
```

上面代码中，子类A的__proto__属性指向父类B，子类A的prototype属性的**proto**属性指向父类B的prototype属性。

这两条继承链，可以这样理解：作为一个对象，子类（B）的原型（__proto__属性）是父类（A）；作为一个构造函数，子类（B）的原型（prototype属性）是父类的实例。

```
B.prototype = new A();
// 等同于
B.prototype.__proto__ = A.prototype;
```

此外，考虑三种特殊情况。第一种特殊情况，子类继承Object类。

```
class A extends Object {
}

A.__proto__ === Object // true
A.prototype.__proto__ === Object.prototype // true
```

这种情况下，A其实就是构造函数Object的复制，A的实例就是Object的实例。

第二种特性情况，不存在任何继承。

```
class A {
}

A.__proto__ === Function.prototype // true
A.prototype.__proto__ === Object.prototype // true
```

这种情况下，A作为一个基类（即不存在任何继承），就是一个普通函数，所以直接继承

Funciton.prototype。但是，A调用后返回一个空对象（即Object实例），所以A.prototype.__proto__指向构造函数（Object）的prototype属性。

第三种特殊情况，子类继承null。

```
class A extends null {
}

A.__proto__ === Function.prototype // true
A.prototype.__proto__ === null // true
```

这种情况与第二种情况非常像。A也是一个普通函数，所以直接继承Funciton.prototype。但是，A调用后返回的对象不继承任何方法，所以它的__proto__指向Function.prototype，即实质上执行了下面的代码。

```
class C extends null {
  constructor() { return Object.create(null); }
}
```

Object.getPrototypeOf()

Object.getPrototypeOf方法可以用来从子类上获取父类。

```
Object.getPrototypeOf(ColorPoint) === Point
// true
```

实例的proto属性

父类实例和子类实例的**proto**属性，指向是不一样的。

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

p2.__proto__ === p1.__proto // false
p2.__proto__.__proto__ === p1.__proto__ // true
```

通过子类实例的**proto**属性，可以修改父类实例的行为。

```
p2.__proto__.__proto__.printName = function () {
  console.log('Ha');
};

p1.printName() // "Ha"
```

上面代码在ColorPoint的实例p2上向Point类添加方法，结果影响到了Point的实例p1。

原生构造函数的继承

原生构造函数是指语言内置的构造函数，通常用来生成数据结构，比如`Array()`。以前，这些原生构造函数是无法继承的，即不能自己定义一个Array的子类。

```
function MyArray() {
  Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
  constructor: {
    value: MyArray,
    writable: true,
    configurable: true,
    enumerable: true
  }
});
```

上面代码定义了一个继承Array的MyArray类。但是，这个类的行为与Array完全不一致。

```
var colors = new MyArray();
colors[0] = "red";
colors.length  // 0

colors.length = 0;
colors[0]  // "red"
```

之所以会发生这种情况，是因为原生构造函数无法外部获取，通过`Array.apply()`或者分配给原型对象都不行。ES5是先新建子类的实例对象this，再将父类的属性添加到子类上，由于父类的属性无法获取，导致无法继承原生的构造函数。

ES6允许继承原生构造函数定义子类，因为ES6是先新建父类的实例对象this，然后再用子类的构造函数修饰this，使得父类的所有行为都可以继承。下面是一个继承Array的例子。

```
class MyArray extends Array {
  constructor(...args) {
    super(...args);
  }
}

var arr = new MyArray();
arr[0] = 12;
arr.length // 1

arr.length = 0;
arr[0] // undefined
```

上面代码定义了一个MyArray类，继承了Array构造函数，因此就可以从MyArray生成数组的实例。这意味着，ES6可以自定义原生数据结构（比如Array、String等）的子类，这是ES5无法做到的。

上面这个例子也说明，extends关键字不仅可以用来继承类，还可以用来继承原生的构造函数。下面是一个自定义Error子类的例子。

```
class MyError extends Error {
}

throw new MyError('Something happened!');
```

class的取值函数（getter）和存值函数（setter）

与ES5一样，在Class内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。

```
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'
```

上面代码中，prop属性有对应的存值函数和取值函数，因此赋值和读取行为都被自定义了。

存值函数和取值函数是设置在属性的descriptor对象上的。

```
class CustomHTMLElement {
  constructor(element) {
    this.element = element;
  }

  get html() {
    return this.element.innerHTML;
  }

  set html(value) {
    this.element.innerHTML = value;
  }
}

var descriptor = Object.getOwnPropertyDescriptor(
  CustomHTMLElement.prototype, "html");
"get" in descriptor  // true
"set" in descriptor  // true
```

上面代码中，存值函数和取值函数是定义在html属性的描述对象上面，这与ES5完全一致。

下面的例子针对所有属性，设置存值函数和取值函数。

```
class Jedi {
  constructor(options = {}) {
    // ...
  }

  set(key, val) {
    this[key] = val;
  }

  get(key) {
    return this[key];
  }
}
```

上面代码中，Jedi实例所有属性的存取，都会通过存值函数和取值函数。

Class的Generator方法

如果某个方法之前加上星号（*），就表示该方法是一个Generator函数。

```
class Foo {
  constructor(...args) {
    this.args = args;
  }
  * [Symbol.iterator]() {
    for (let arg of this.args) {
      yield arg;
    }
  }
}

for (let x of new Foo('hello', 'world')) {
  console.log(x);
}
// hello
// world
```

上面代码中，Foo类的Symbol.iterator方法前有一个星号，表示该方法是一个Generator函数。Symbol.iterator方法返回一个Foo类的默认遍历器，for...of循环会自动调用这个遍历器。

Class的静态方法

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'

var foo = new Foo();
foo.classMethod()
// TypeError: undefined is not a function
```

上面代码中，Foo类的classMethod方法前有static关键字，表明该方法是一个静态方法，可以直接在Foo类上调用（`Foo.classMethod()`），而不是在Foo类的实例上调用。如果在实例上调用静态方法，会抛出一个错误，表示不存在该方法。

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'
```

上面代码中，父类Foo有一个静态方法，子类Bar可以调用这个方法。

静态方法也是可以从super对象上调用的。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

class Bar extends Foo {
  static classMethod() {
    return super.classMethod() + ', too';
  }
}

Bar.classMethod();
```

new.target属性

new是从构造函数生成实例的命令。ES6为new命令引入了一个 `new.target` 属性，（在构造函数中）返回new命令作用于的那个构造函数。如果构造函数不是通过new命令调用的， `new.target` 会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的。

```
function Person(name) {
  if (new.target !== undefined) {
    this.name = name;
  } else {
    throw new Error('必须使用new生成实例');
  }
}

// 另一种写法
function Person(name) {
  if (new.target === Person) {
    this.name = name;
  } else {
    throw new Error('必须使用new生成实例');
  }
}

var person = new Person('张三'); // 正确
var notAPerson = Person.call(person, '张三');  // 报错
```

上面代码确保构造函数只能通过new命令调用。

Class内部调用 `new.target` ，返回当前Class。

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    this.length = length;
    this.width = width;
  }
}

var obj = new Rectangle(3, 4); // 输出 true
```

需要注意的是，子类继承父类时，`new.target`会返回子类。

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    // ...
  }
}

class Square extends Rectangle {
  constructor(length) {
    super(length, length);
  }
}

var obj = new Square(3); // 输出 false
```

上面代码中，`new.target`会返回子类。

利用这个特点，可以写出不能独立使用、必须继承后才能使用的类。

```
class Shape {
  constructor() {
    if (new.target === Shape) {
      throw new Error('本类不能实例化');
    }
  }
}

class Rectangle extends Shape {
  constructor(length, width) {
    super();
    // ...
  }
}

var x = new Shape();  // 报错
var y = new Rectangle(3, 4);  // 正确
```

上面代码中，Shape类不能被实例化，只能用于继承。

注意，在函数外部，使用`new.target`会报错。

修饰器

类的修饰

修饰器（Decorator）是一个表达式，用来修改类的行为。这是ES7的一个提案，目前Babel转码器已经支持。

修饰器对类的行为的改变，是代码编译时发生的，而不是在运行时。这意味着，修饰器能在编译阶段运行代码。

```
function testable(target) {
  target.isTestable = true;
}

@testable
class MyTestableClass () {}

console.log(MyTestableClass.isTestable) // true
```

上面代码中，`@testable`就是一个修饰器。它修改了MyTestableClass这个类的行为，为它加上了静态属性isTestable。

修饰器函数可以接受三个参数，依次是目标函数、属性名和该属性的描述对象。后两个参数可省略。上面

代码中，testable函数的参数target，就是所要修饰的对象。如果希望修饰器的行为，能够根据目标对象的不同而不同，就要在外面再封装一层函数。

```
function testable(isTestable) {
  return function(target) {
    target.isTestable = isTestable;
  }
}

@testable(true) class MyTestableClass () {}
console.log(MyTestableClass.isTestable) // true

@testable(false) class MyClass () {}
console.log(MyClass.isTestable) // false
```

上面代码中，修饰器testable可以接受参数，这就等于可以修改修饰器的行为。

如果想要为类的实例添加方法，可以在修饰器函数中，为目标类的prototype属性添加方法。

```
function testable(target) {
  target.prototype.isTestable = true;
}

@testable
class MyTestableClass () {}

let obj = new MyClass();

console.log(obj.isTestable) // true
```

上面代码中，修饰器函数testable是在目标类的prototype属性添加属性，因此就可以在类的实例上调用添加的属性。

下面是另外一个例子。

```
// mixins.js
export function mixins(...list) {
  return function (target) {
    Object.assign(target.prototype, ...list)
  }
}

// main.js
import { mixins } from './mixins'

const Foo = {
  foo() { console.log('foo') }
}

@mixins(Foo)
class MyClass {}

let obj = new MyClass()

obj.foo() // 'foo'
```

上面代码通过修饰器mixins，可以为类添加指定的方法。

修饰器可以用`Object.assign()`模拟。

```
const Foo = {
  foo() { console.log('foo') }
}

class MyClass {}

Object.assign(MyClass.prototype, Foo);

let obj = new MyClass();
obj.foo() // 'foo'
```

方法的修饰

修饰器不仅可以修饰类，还可以修饰类的属性。

```
class Person {
  @readonly
  name() { return `${this.first} ${this.last}` }
}
```

上面代码中，修饰器readonly用来修饰”类“的name方法。

此时，修饰器函数一共可以接受三个参数，第一个参数是所要修饰的目标对象，第二个参数是所要修饰的属性名，第三个参数是该属性的描述对象。

```
readonly(Person.prototype, 'name', descriptor);

function readonly(target, name, descriptor){
  // descriptor对象原来的值如下
  // {
  //   value: specifiedFunction,
  //   enumerable: false,
  //   configurable: true,
  //   writable: true
  // };
  descriptor.writable = false;
  return descriptor;
}

Object.defineProperty(Person.prototype, 'name', descriptor);
```

上面代码说明，修饰器（readonly）会修改属性的描述对象（descriptor），然后被修改的描述对象再用来定义属性。下面是另一个例子。

```
class Person {
  @nonenumerable
  get kidCount() { return this.children.length; }
}

function nonenumerable(target, name, descriptor) {
  descriptor.enumerable = false;
  return descriptor;
}
```

修饰器有注释的作用。

```
@testable
class Person {
  @readonly
  @nonenumerable
  name() { return `${this.first} ${this.last}` }
}
```

从上面代码中，我们一眼就能看出，MyTestableClass类是可测试的，而name方法是只读和不可枚举的。

除了注释，修饰器还能用来类型检查。所以，对于Class来说，这项功能相当有用。从长期来看，它将是JavaScript代码静态分析的重要工具。

core-decorators.js

core-decorators.js是一个第三方模块，提供了几个常见的修饰器，通过它可以更好地理解修饰器。

（1）@autobind

autobind修饰器使得方法中的this对象，绑定原始对象。

```
import { autobind } from 'core-decorators';

class Person {
  @autobind
  getPerson() {
    return this;
  }
}

let person = new Person();
let getPerson = person.getPerson;

getPerson() === person;
// true
```

（2）@readonly

readonly修饰器是的属性或方法不可写。

```
import { readonly } from 'core-decorators';

class Meal {
  @readonly
  entree = 'steak';
}

var dinner = new Meal();
dinner.entree = 'salmon';
// Cannot assign to read only property 'entree' of [object Object]
```

（3）@override

override修饰器检查子类的方法，是否正确覆盖了父类的同名方法，如果不正确会报错。

```
import { override } from 'core-decorators';

class Parent {
  speak(first, second) {}
}

class Child extends Parent {
  @override
  speak() {}
  // SyntaxError: Child#speak() does not properly override Parent#speak(first, second)
}

// or

class Child extends Parent {
  @override
  speaks() {}
  // SyntaxError: No descriptor matching Child#speaks() was found on the prototype chain.
  //
  //   Did you mean "speak"?
}
```

（4）@deprecate (别名@deprecated)

deprecate或deprecated修饰器在控制台显示一条警告，表示该方法将废除。

```
import { deprecate } from 'core-decorators';

class Person {
  @deprecate
  facepalm() {}

  @deprecate('We stopped facepalming')
  facepalmHard() {}

  @deprecate('We stopped facepalming', { url: 'http://knowyourmeme.com/memes/facepalm'
})
  facepalmHarder() {}
}

let person = new Person();

person.facepalm();
// DEPRECATION Person#facepalm: This function will be removed in future versions.

person.facepalmHard();
// DEPRECATION Person#facepalmHard: We stopped facepalming

person.facepalmHarder();
// DEPRECATION Person#facepalmHarder: We stopped facepalming
//
//     See http://knowyourmeme.com/memes/facepalm for more details.
//
```

（5）@suppressWarnings

suppressWarnings修饰器抑制decorated修饰器导致的`console.warn()`调用。但是，异步代码出发的调用除外。

```
import { suppressWarnings } from 'core-decorators';

class Person {
  @deprecated
  facepalm() {}

  @suppressWarnings
  facepalmWithoutWarning() {
    this.facepalm();
  }
}

let person = new Person();

person.facepalmWithoutWarning();
// no warning is logged
```

Mixin

在修饰器的基础上，可以实现Mixin模式。所谓Mixin模式，就是对象继承的一种替代方案，中文译为“混入”（mix in），意为在一个对象之中混入另外一个对象的方法。

请看下面的例子。

```
const Foo = {
  foo() { console.log('foo') }
};

class MyClass {}

Object.assign(MyClass.prototype, Foo);

let obj = new MyClass();
obj.foo() // 'foo'
```

上面代码之中，对象Foo有一个foo方法，通过`Object.assign`方法，可以将foo方法“混入”MyClass类，导致MyClass的实例obj对象都具有foo方法。这就是“混入”模式的一个简单实现。

下面，我们部署一个通用脚本`mixins.js`，将mixin写成一个修饰器。

```
export function mixins(...list) {
  return function (target) {
    Object.assign(target.prototype, ...list);
  };
}
```

然后，就可以使用上面这个修饰器，为类“混入”各种方法。

```
import { mixins } from './mixins'

const Foo = {
  foo() { console.log('foo') }
};

@mixins(Foo)
class MyClass {}

let obj = new MyClass();

obj.foo() // "foo"
```

通过mixins这个修饰器，实现了在MyClass类上面“混入”Foo对象的foo方法。

Trait

Trait也是一种修饰器，功能与Mixin类型，但是提供更多功能，比如防止同名方法的冲突、排除混入某些方法、为混入的方法起别名等等。

下面采用traits-decorator这个第三方模块作为例子。这个模块提供的traits修饰器，不仅可以接受对象，还可以接受ES6类作为参数。

```
import {traits } from 'traits-decorator'

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') }
}

@traits(TFoo, TBar)
class MyClass { }

let obj = new MyClass()
obj.foo() // foo
obj.bar() // bar
```

上面代码中，通过traits修饰器，在MyClass类上面“混入”了TFoo类的foo方法和TBar对象的bar方法。

Trait不允许“混入”同名方法。

```
import {traits } from 'traits-decorator'

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
}

@traits(TFoo, TBar)
class MyClass { }
// 报错
// throw new Error('Method named: ' + methodName + ' is defined twice.');
//         ^
// Error: Method named: foo is defined twice.
```

上面代码中，TFoo和TBar都有foo方法，结果traits修饰器报错。

一种解决方法是排除TBar的foo方法。

```
import { traits, excludes } from 'traits-decorator'

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
}

@traits(TFoo, TBar::excludes('foo'))
class MyClass { }

let obj = new MyClass()
obj.foo() // foo
obj.bar() // bar
```

上面代码使用绑定运算符（::）在TBar上排除foo方法，混入时就不会报错了。

另一种方法是为TBar的foo方法起一个别名。

```
import { traits, alias } from 'traits-decorator'

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
}

@traits(TFoo, TBar::alias({foo: 'aliasFoo'}))
class MyClass { }

let obj = new MyClass()
obj.foo() // foo
obj.aliasFoo() // foo
obj.bar() // bar
```

上面代码为TBar的foo方法起了别名aliasFoo，于是MyClass也可以混入TBar的foo方法了。

alias和excludes方法，可以结合起来使用。

```
@traits(TExample::excludes('foo','bar')::alias({baz:'exampleBaz'}))
class MyClass {}
```

上面代码排除了TExample的foo方法和bar方法，为baz方法起了别名exampleBaz。

as方法则为上面的代码提供了另一种写法。

```
@traits(TExample::as({excludes:['foo', 'bar'], alias: {baz: 'exampleBaz'}}))
class MyClass {}
```

Babel转码器的支持

目前，Babel转码器已经支持Decorator，命令行的用法如下。

```
$ babel --optional es7.decorators
```

脚本中打开的命令如下。

```
babel.transfrom("code", {optional: ["es7.decorators"]})
```

Babel的官方网站提供一个在线转码器，只要勾选Experimental，就能支持Decorator的在线转码。

Module

ES6的Class只是面向对象编程的语法糖，升级了ES5的构造函数的原型链继承的写法，并没有解决模块化问题。Module功能就是为了解决这个问题而提出的。

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。其他语言都有这项功能，比如Ruby的require、Python的import，甚至就连CSS都有@import，但是JavaScript任何这方面的支持都没有，这对开发大型的、复杂的项目形成了巨大障碍。

在ES6之前，社区制定了一些模块加载方案，最主要的有CommonJS和AMD两种。前者用于服务器，后者用于浏览器。ES6在语言规格的层面上，实现了模块功能，而且实现得相当简单，完全可以取代现有的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。CommonJS和AMD模块，都只能在运行时确定这些东西。比如，CommonJS模块就是对象，输入时必须查找对象属性。

```
var { stat, exists, readFile } = require('fs');
```

ES6模块不是对象，而是通过export命令显式指定输出的代码，输入时也采用静态命令的形式。

```
import { stat, exists, readFile } from 'fs';
```

所以，ES6可以在编译时就完成模块编译，效率要比CommonJS模块高。

export命令

模块功能主要由两个命令构成：export和import。export命令用于用户自定义模块，规定对外接口；import命令用于输入其他模块提供的功能，同时创造命名空间（namespace），防止函数名冲突。

ES6允许将独立的JS文件作为模块，也就是说，允许一个JavaScript脚本文件调用另一个脚本文件。该文件内部的所有变量，外部无法获取，必须使用export关键字输出变量。下面是一个JS文件，里面使用export命令输出变量。

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

上面代码是profile.js文件，保存了用户信息。ES6将其视为一个模块，里面用export命令对外部输出了三

个变量。

export的写法，除了像上面这样，还有另外一种。

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

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

上面代码在export命令后面，使用大括号指定所要输出的一组变量。它与前一种写法（直接放置在var语句前）是等价的，但是应该优先考虑使用这种写法。因为这样就可以在脚本尾部，一眼看清楚输出了哪些变量。

export命令除了输出变量，还可以输出函数或类（class）。

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

上面代码对外输出一个函数multiply。

import命令

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（文件）。

```
// main.js

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

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

上面代码属于另一个文件main.js，import命令就用于加载profile.js文件，并从中输入变量。import命令接受一个对象（用大括号表示），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，必须与被导入模块（profile.js）对外接口的名称相同。

如果想为输入的变量重新取一个名字，import语句中要使用as关键字，将输入的变量重命名。

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

ES6支持多重加载，即所加载的模块中又加载其他模块。

```
import { Vehicle } from './Vehicle';

class Car extends Vehicle {
  move () {
    console.log(this.name + ' is spinning wheels...')
  }
}

export { Car }
```

上面的模块先加载Vehicle模块，然后在其基础上添加了move方法，再作为一个新模块输出。

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 { es6 as default } from './someModule';

// 等同于
import { es6 } from './someModule';
export default es6;
```

上面代码中，export和import语句可以结合在一起，写成一行。但是从可读性考虑，不建议采用这种写法，h应该采用标准写法。

模块的整体输入

下面是一个circle.js文件，它输出两个方法area和circumference。

```
// circle.js

export function area(radius) {
  return Math.PI * radius * radius;
}

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

然后，main.js文件输入circlek.js模块。

```
// 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输入的方法。另一种写法是整体输入。

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

console.log("圆面积：" + circle.area(4));
console.log("圆周长：" + circle.circumference(14));
```

module命令

module命令可以取代import语句，达到整体输入模块的作用。

```
// main.js

module circle from 'circle';

console.log("圆面积：" + circle.area(4));
console.log("圆周长：" + circle.circumference(14));
```

module命令后面跟一个变量，表示输入的模块定义在该变量上。

export default命令

从前面的例子可以看出，使用import的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到`export default`命令，为模块指定默认输出。

```
// export-default.js
export default function () {
  console.log('foo');
}
```

上面代码是一个模块文件`export-default.js`，它的默认输出是一个函数。

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。需要注意的是，这时import命令后面，不使用大括号。

export default命令用在非匿名函数前，也是可以的。

```
// export-default.js
export default function foo() {
  console.log('foo');
}

// 或者写成

function foo() {
  console.log('foo');
}

export default foo;
```

上面代码中，foo函数的函数名foo，在模块外部是无效的。加载的时候，视同匿名函数加载。

下面比较一下默认输出和正常输出。

```
import crc32 from 'crc32';
// 对应的输出
export default function crc32(){}

import { crc32 } from 'crc32';
// 对应的输出
export function crc32(){};
```

上面代码的两组写法，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。

export default 命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此 export deault 命令只能使用一次。所以，import命令后面才不用加大括号，因为只可能对应一个方法。

本质上， export default 就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效的。

```
// modules.js
export default function (x, y) {
  return x * y;
};
// app.js
import { default } from 'modules';
```

有了`export default`命令，输入模块时就非常直观了，以输入jQuery模块为例。

```
import $ from 'jquery';
```

如果想在一条import语句中，同时输入默认方法和其他变量，可以写成下面这样。

```
import customName, { otherMethod } from './export-default';
```

如果要输出默认的值，只需将值跟在`export default`之后即可。

```
export default 42;
```

`export default`也可以用来输出类。

```
// MyClass.js
export default class { ... }

// main.js
import MyClass from 'MyClass'
let o = new MyClass();
```

模块的继承

模块之间也可以继承。

假设有一个circleplus模块，继承了circle模块。

```
// circleplus.js

export * from 'circle';
export var e = 2.71828182846;
export default function(x) {
    return Math.exp(x);
}
```

上面代码中的“export *”，表示输出circle模块的所有属性和方法，export default命令定义模块的默认方法。

这时，也可以将circle的属性或方法，改名后再输出。

```
// circleplus.js

export { area as circleArea } from 'circle';
```

上面代码表示，只输出circle模块的area方法，且将其改名为circleArea。

加载上面模块的写法如下。

```
// main.js

module math from "circleplus";
import exp from "circleplus";
console.log(exp(math.pi));
```

上面代码中的"import exp"表示，将circleplus模块的默认方法加载为exp方法。

ES6模块的转码

浏览器目前还不支持ES6模块，为了现在就能使用，可以将转为ES5的写法。

ES6 module transpiler

ES6 module transpiler是square公司开源的一个转码器，可以将ES6模块转为CommonJS模块或AMD模块的写法，从而在浏览器中使用。

首先，安装这个转玛器。

```
$ npm install -g es6-module-transpiler
```

然后，使用`compile-modules convert`命令，将ES6模块文件转码。

```
$ compile-modules convert file1.js file2.js
```

o参数可以指定转码后的文件名。

```
$ compile-modules convert -o out.js file1.js
```

SystemJS

另一种解决方法是使用SystemJS。它是一个垫片库（polyfill），可以在浏览器内加载ES6模块、AMD模块和CommonJS模块，将其转为ES5格式。它在后台调用的是Google的Traceur转码器。

使用时，先在网页内载入system.js文件。

```
<script src="system.js">
```

然后，使用`System.import`方法加载模块文件。

```
  System.import('./app');
```

上面代码中的`./app`，指的是当前目录下的app.js文件。它可以是ES6模块文件，`System.import`会自动将其转码。

需要注意的是，`System.import`使用异步加载，返回一个Promise对象，可以针对这个对象编程。下面是一个模块文件。

```
// app/es6-file.js:

export class q {
  constructor() {
    this.es6 = 'hello';
  }
}
```

然后，在网页内加载这个模块文件。

```
System.import('app/es6-file').then(function(m) {
  console.log(new m.q().es6); // hello
});
```

上面代码中，`System.import`方法返回的是一个Promise对象，所以可以用then方法指定回调函数。

编程风格

本章探讨如何将ES6的新语法，运用到编码实践之中，与传统的JavaScript语法结合在一起，写出合理的、易于阅读和维护的代码。多家公司和组织已经公开了它们的风格规范，具体可参阅jscs.info，下面的内容主要参考了Airbnb的JavaScript风格规范。

块级作用域

（1）let取代var

ES6提出了两个新的声明变量的命令：let和const。其中，let完全可以取代var，因为两者语义相同，而且let没有副作用。

```
"use strict";

if(true) {
  let x = 'hello';
}

for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

上面代码如果用var替代let，实际上就声明了一个全局变量，这显然不是本意。变量应该只在其声明的代码块内有效，var命令做不到这一点。

var命令存在变量提升效用，let命令没有这个问题。

```
"use strict";

if(true) {
  console.log(x); // ReferenceError
  let x = 'hello';
}
```

上面代码如果使用var替代let，console.log那一行就不会报错，而是会输出undefined，因为变量声明提升到代码块的头部。这违反了变量先声明后使用的原则。

所以，建议不再使用var命令，而是使用let命令取代。

（2）全局常量和线程安全

在let和const之间，建议优先使用const，尤其是在全局环境，不应该设置变量，只应设置常量。这符合函

数式编程思想，有利于将来的分布式运算。

```
// bad
var a = 1, b = 2, c = 3;

// good
const a = 1;
const b = 2;
const c = 3;

// best
const [a, b, c] = [1, 2, 3];
```

const声明常量还有两个好处，一是阅读代码的人立刻会意识到不应该修改这个值，二是防止了无意间修改变量值所导致的错误。

所有的函数都应该设置为常量。

let表示的变量，只应出现在单线程运行的代码中，不能是多线程共享的，这样有利于保证线程安全。

（3）严格模式

V8引擎只在严格模式之下，支持let和const。结合前两点，这实际上意味着，将来所有的编程都是针对严格模式的。

字符串

静态字符串一律使用单引号或反引号，不使用双引号。动态字符串使用反引号。

```
// bad
const a = "foobar";
const b = 'foo' + a + 'bar';

// acceptable
const c = `foobar`;

// good
const a  = 'foobar';
const b = `foo${a}bar`;
const c = 'foobar';
```

解构赋值

使用数组成员对变量赋值，优先使用解构赋值。

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

函数的参数如果是对象的成员，优先使用解构赋值。

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;
}

// good
function getFullName(obj) {
  const { firstName, lastName } = obj;
}

// best
function getFullName({ firstName, lastName }) {
}
```

如果函数返回多个值，优先使用对象的解构赋值，而不是数组的解构赋值。这样便于以后添加返回值，以及更改返回值的顺序。

```
// bad
function processInput(input) {
  return [left, right, top, bottom];
}

// good
function processInput(input) {
  return { left, right, top, bottom };
}

const { left, right } = processInput(input);
```

对象

单行定义的对象，最后一个成员不以逗号结尾。多行定义的对象，最后一个成员以逗号结尾。

```
// bad
const a = { k1: v1, k2: v2, };
const b = {
  k1: v1,
  k2: v2
};

// good
const a = { k1: v1, k2: v2 };
const b = {
  k1: v1,
  k2: v2,
};
```

对象尽量静态化，一旦定义，就不得随意添加新的属性。如果添加属性不可避免，要使用Object.assign方法。

```
// bad
const a = {};
a.x = 3;

// if reshape unavoidable
const a = {};
Object.assign(a, { x: 3 });

// good
const a = { x: null };
a.x = 3;
```

如果对象的属性名是动态的，可以在创造对象的时候，使用属性表达式定义。

```
// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;

// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```

上面代码中，对象obj的最后一个属性名，需要计算得到。这时最好采用属性表达式，在新建obj的时候，将该属性与其他属性定义在一起。这样一来，所有属性就在一个地方定义了。

另外，对象的属性和方法，尽量采用简洁表达法，这样易于描述和书写。

```
var ref = 'some value';

// bad
const atom = {
  ref: ref,

  value: 1,

  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  ref,

  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

数组

使用扩展运算符（...）拷贝数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

使用Array.from方法，将类似数组的对象转为数组。

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

函数

立即执行函数可以写成箭头函数的形式。

```
(() => {
  console.log('Welcome to the Internet.');
})();
```

那些需要使用函数表达式的场合，尽量用箭头函数代替。因为这样更简洁，而且绑定了this。

```
// bad
[1, 2, 3].map(function (x) {
  return x * x;
});

// good
[1, 2, 3].map((x) => {
  return x * x;
});
```

箭头函数取代Function.prototype.bind，不应再用self/_this/that绑定 this。

```
// bad
const self = this;
const boundMethod = function(...params) {
  return method.apply(self, params);
}

// acceptable
const boundMethod = method.bind(this);

// best
const boundMethod = (...params) => method.apply(this, params);
```

所有配置项都应该集中在一个对象，放在最后一个参数，布尔值不可以直接作为参数。

```
// bad
function divide(a, b, option = false ) {
}

// good
function divide(a, b, { option = false } = {}) {
}
```

不要在函数体内使用arguments变量，使用rest运算符（...）代替。因为rest运算符显式表明你想要获取参数，而且arguments是一个类似数组的对象，而rest运算符可以提供一个真正的数组。

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

使用默认值语法设置函数参数的默认值。

```
// bad
function handleThings(opts) {
  opts = opts || {};
}

// good
function handleThings(opts = {}) {
  // ...
}
```

Map结构

注意区分Object和Map，只有模拟实体对象时，才使用Object。如果只是需要key:value的数据结构，使用Map。因为Map有内建的遍历机制。

```
let map = new Map(arr);

for (let key of map.keys()) {
  console.log(key);
}

for (let value of map.values()) {
  console.log(value);
}

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
```

Class

总是用class，取代需要prototype操作。因为class的写法更简洁，更易于理解。

```
// bad
function Queue(contents = []) {
  this._queue = [...contents];
}
Queue.prototype.pop = function() {
  const value = this._queue[0];
  this._queue.splice(0, 1);
  return value;
}

// good
class Queue {
  constructor(contents = []) {
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

使用extends实现继承，因为这样更简单，不会有破坏instanceof运算的危险。

```
// bad
const inherits = require('inherits');
function PeekableQueue(contents) {
  Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function() {
  return this._queue[0];
}

// good
class PeekableQueue extends Queue {
  peek() {
    return this._queue[0];
  }
}
```

模块

首先，Module语法是JavaScript模块的标准写法，坚持使用这种写法。使用import取代require。

```
// bad
const moduleA = require('moduleA');
const func1 = moduleA.func1;
const func2 = moduleA.func2;

// good
import { func1, func2 } from 'moduleA';
```

使用export取代module.exports。

```
// commonJS的写法
var React = require('react');

var Breadcrumbs = React.createClass({
  render() {
    return <nav />;
  }
});

module.exports = Breadcrumbs;

// ES6的写法
import React from 'react';

const Breadcrumbs = React.createClass({
  render() {
    return <nav />;
  }
});

export default Breadcrumbs
```

不要在模块输入中使用通配符。因为这样可以确保你的模块之中，有一个默认输出（export default）。

```
// bad
import * as myObject './importModule';

// good
import myObject from './importModule';
```

如果模块默认输出一个函数，函数名的首字母应该小写。

```
function makeStyleGuide() {
}

export default makeStyleGuide;
```

如果模块默认输出一个对象，对象名的首字母应该大写。

```
const StyleGuide = {
  es6: {
  }
};

export default StyleGuide;
```