ECMAScript6入门

极客学院出版

前言

《ECMAScript 6 入门》是一本开源的 JavaScript 语言教程，全面介绍 ECMAScript 6 新引入的语法特性。

本书力争覆盖 ES6 与 ES5 的所有不同之处，对涉及的语法知识给予详细介绍，并给出大量简洁易懂的示例代码。

适用人群

本书为中级难度，适合已有一定 JavaScript 语言基础的读者，用来了解这门语言的最新发展；也可当作参考手册，查寻新增的语法点。

学习前提

学习本书前，对 ECMAScript 5 有了解，能够理解涉及 JavaScript 语言的基本概念。

鸣谢：阮一峰

源码地址：https://github.com/ruanyf/es6tutorial/

目录

前言 1

第 1 章　作者简介 7

. 9

. 9

第 2 章　ECMAScript 6简介 11

. 9

. 9

. 9

. 9

. 9

. 9

第 3 章　let 和 const 命令 24

. 9

. 9

. 9

. 9

第 4 章　变量的解构赋值 34

. 9

. 9

. 9

. 9

. 9

. 9

第 5 章　字符串的扩展 47
第 6 章　数值的扩展 63
#. 9
#. 9
#. 9
#. 9
#. 9
第 7 章　数组的扩展 71
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
第 8 章　对象的扩展 82
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
第 9 章　函数的扩展 107
#. 9

#. 9
#. 9
#. 9
#. 9
#. 9
第 10 章　Set 和 Map 数据结构 125
#. 9
#. 9
#. 9
#. 9
第 11 章　Iterator 和 for...of 循环. 140
#. 9
#. 9
第 12 章　Generator 函数 156
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
第 13 章　Promise 对象. 180
#. 9
#. 9
#. 9

#. 9
#. 9
#. 9
#. 9
第 14 章　异步操作 199
#. 9
#. 9
#. 9
#. 9
第 15 章　字符串的扩展 47
第 16 章　编程风格 229
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
第 17 章　参考链接 238
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9
#. 9

#. 9
#. 9
#. 9

1 作者简介

阮一峰，70 后，在上海出生和长大，大学糊里糊涂读了经济学。工作了几年，又去读了世界经济的研究生，毕业后，在上海一所本地高校当了老师，教财经类的课程。最近，去了支付宝的前端团队，在玉伯负责的"体验技术部"工作，目前主要从事 JavaScript 和 Node.js 的开发。

翻译了《软件随想录》和《黑客与画家》，出版了技术专著《ECMAScript 6入门》和博客文集《如何变得有思想》。

# #

关于本书

全书已由电子工业出版社出版（版权页，内页1，内页2），铜版纸全彩印刷，附有索引。纸版是根据电子版排印的，内容截止到2014年10月，感谢张春雨编辑支持我将全书开源的做法。如果您对本书感兴趣，建议考虑购买纸版。这样可以使出版社不因出版开源书籍而亏钱，进而鼓励更多的作者开源自己的书籍。

- 京东
- 当当
- 亚马逊
- China-pub

# #

2

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。

#

ECMAScript 和 JavaScript 的关系

很多初学者会感到困惑：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）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，io.js 不需要。

```
$ node --harmony
 # io.js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。

#

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。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
```

Babel 也可以用于浏览器。

```
<script src="node_modules/babel-core/browser.js"></script>
<script type="text/babel">
// Your ES6 code
</script>
```

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"` 。

#

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。

```
<!-- 加载Traceur编译器 -->
<script src="http://google.github.io/traceur-compiler/bin/traceur.js"
      type="text/javascript"></script>
<!-- 将Traceur编译器用于网页 -->
<script src="http://google.github.io/traceur-compiler/src/bootstrap.js"
      type="text/javascript"></script>
<!-- 打开实验选项，否则有些特性可能编译不成功 -->
<script>
      traceur.options.experimental = true;
</script>
```

接下来，就可以把 ES6 代码放入上面这些代码的下方。

```
<script type="module">
 class Calc {
  constructor(){
   console.log('Calc constructor');
  }
  add(a, b){
   return a + b;
  }
 }

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

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。

```
<script type="module" src="calc.js" >
</script>
```

#

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。

```
<script src="http://google.github.io/traceur-compiler/bin/traceur.js"
      type="text/javascript"></script>
<script src="http://google.github.io/traceur-compiler/src/bootstrap.js"
      type="text/javascript"></script>
<script>
      traceur.options.experimental = true;
</script>
<script>
$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 {};
});
</script>
```

#

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。

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

// 报错
{
  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] );
    }
  });
};
```

#

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。

4 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");
```

5

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 = "?";

s.length // 2
s.charAt(0) // "
s.charAt(1) // "
s.charCodeAt(0) // 55362
s.charCodeAt(1) // 57271
```

上面代码中，汉字“?”的码点是 0x20BB7，UTF-16 编码为 0xD842 0xDFB7（十进制为 55362 57271），需要 4 个字节储存。对于这种 4 个字节的字符，JavaScript 不能正确处理，字符串长度会误判为 2，而且 charAt 方法无法读取字符，charCodeAt 方法只能分别返回前两个字节和后两个字节的值。

ES6 提供了 codePointAt 方法，能够正确处理 4 个字节储存的字符，返回一个字符的码点。

```
var s = "?a";

s.codePointAt(0) // 134071
s.codePointAt(1) // 57271

s.charCodeAt(2) // 97
```

codePointAt 方法的参数，是字符在字符串中的位置（从 0 开始）。上面代码中，JavaScript 将“?a”视为三个字符，codePointAt 方法在第一个字符上，正确地识别了“?”，返回了它的十进制码点 134071（即十六进制的20BB7）。在第二个字符（即“?”的后两个字节）和第三个字符“a”上，codePointAt 方法的结果与 charCodeAt 方法相同。

总之，codePointAt 方法会正确返回四字节的 UTF-16 字符的码点。对于那些两个字节储存的常规字符，它的返回结果与 charCodeAt 方法相同。

codePointAt 方法是测试一个字符由两个字节还是由四个字节组成的最简单方法。

```
function is32Bit(c) {
  return c.codePointAt(0) > 0xFFFF;
}
```

```
is32Bit("?") // true
is32Bit("a") // false
```

#

String.fromCodePoint()

ES5 提供 String.fromCharCode 方法，用于从码点返回对应字符，但是这个方法不能识别辅助平面的字符（编号大于 0xFFFF）。

```
String.fromCharCode(0x20BB7)
// "?"
```

上面代码中，最后返回码点 U+0BB7 对应的字符，而不是码点 U+20BB7 对应的字符。

ES6 提供了 String.fromCodePoint 方法，可以识别 0xFFFF 的字符，弥补了 String.fromCharCode 方法的不足。在作用上，正好与 codePointAt 方法相反。

```
String.fromCodePoint(0x20BB7)
// "?"
```

注意，fromCodePoint 方法定义在 String 对象上，而 codePointAt 方法定义在字符串的实例对象上。

#

String.prototype.at()

ES5 提供 String.prototype.charAt 方法，返回字符串给定位置的字符。该方法不能识别码点大于 0xFFFF 的字符。

```
'?'.charAt(0)
// '\uD842'
```

上面代码中，charAt 方法返回的是 UTF-16 编码的第一个字节，实际上是无法显示的。

ES7 提供了字符串实例的 at 方法，可以识别 Unicode 编号大于 0xFFFF 的字符，返回正确的字符。

```
'?'.at(0)
// '?'
```

#

字符的 Unicode 表示法

JavaScript 允许采用“\uxxxx”形式表示一个字符，其中“xxxx”表示字符的码点。

```
"\u0061"
// "a"
```

但是，这种表示法只限于 \u0000——\uFFFF 之间的字符。超出这个范围的字符，必须用两个双字节的形式表达。

```
"\uD842\uDFB7"
// "?"

"\u20BB7"
// " 7"
```

上面代码表示，如果直接在“\u”后面跟上超过 0xFFFF 的数值（比如 \u20BB7 ），JavaScript 会理解成“\u20BB+7”。由于 \u20BB 是一个不可打印字符，所以只会显示一个空格，后面跟着一个 7。

ES6 对这一点做出了改进，只要将码点放入大括号，就能正确解读该字符。

```
"\u{20BB7}"
// "?"

"\u{41}\u{42}\u{43}"
// "ABC"
```

#

正则表达式的 u 修饰符

ES6 对正则表达式添加了 u 修饰符，用来正确处理大于 \uFFFF 的 Unicode 字符。

（1）点字符

点（.）字符在正则表达式中，解释为除了换行以外的任意单个字符。对于码点大于 0xFFFF 的 Unicode 字符，点字符不能识别，必须加上 u 修饰符。

```
var s = "?";
```

```
/^.$/.test(s) // false
/^.$/u.test(s) // true
```

上面代码表示，如果不添加 u 修饰符，正则表达式就会认为字符串为两个字符，从而匹配失败。

（2）Unicode字符表示法

ES6 新增了使用大括号表示 Unicode 字符，这种表示法在正则表达式中必须加上 u 修饰符，才能识别。

```
/\u{61}/.test('a') // false
/\u{61}/u.test('a') // true
/\u{20BB7}/u.test('?') // true
```

上面代码表示，如果不加 u 修饰符，正则表达式无法识别 `\u{61}` 这种表示法，只会认为这匹配 61 个连续的 u。

（3）量词

使用 u 修饰符后，所有量词都会正确识别大于码点大于 0xFFFF 的 Unicode 字符。

```
/a{2}/.test('aa') // true
/a{2}/u.test('aa') // true
/?{2}/.test('??') // false
/?{2}/u.test('??') // true
```

（4）预定义模式

u 修饰符也影响到预定义模式，能否正确识别码点大于 0xFFFF 的 Unicode 字符。

```
/^\S$/.test('?') // false
/^\S$/u.test('?')
```

上面代码的 `\S` 是预定义模式，匹配所有不是空格的字符。只有加了 u 修饰符，它才能正确匹配码点大于 0xFFFF 的 Unicode 字符。

利用这一点，可以写出一个正确返回字符串长度的函数。

```
function codePointLength(text) {
  var result = text.match(/[\s\S]/gu);
  return result ? result.length : 0;
}

var s = "??";

s.length // 4
codePointLength(s) // 2
```

（5）i 修饰符

有些 Unicode 字符的编码不同，但是字型很相近，比如， \u004B 与 \u212A 都是大写的 K。

```
/[a-z]/i.test('\u212A') // false
/[a-z]/iu.test('\u212A') // true
```

上面代码中，不加 u 修饰符，就无法识别非规范的 K 字符。

#

normalize()

为了表示语调和重音符号，Unicode 提供了两种方法。一种是直接提供带重音符号的字符，比如 Ǒ（\u01D1）。另一种是提供合成符号（combining character），即原字符与重音符号的合成，两个字符合成一个字符，比如 O（\u004F）和 ˇ（\u030C）合成 Ǒ（\u004F\u030C）。

这两种表示方法，在视觉和语义上都等价，但是 JavaScript 不能识别。

```
'\u01D1'==='\u004F\u030C' //false

'\u01D1'.length // 1
'\u004F\u030C'.length // 2
```

上面代码表示，JavaScript 将合成字符视为两个字符，导致两种表示方法不相等。

ES6 提供 String.prototype.normalize()方法，用来将字符的不同表示方法统一为同样的形式，这称为 Unicode 正规化。

```
'\u01D1'.normalize() === '\u004F\u030C'.normalize()
// true
```

normalize 方法可以接受四个参数。

- NFC，默认参数，表示“标准等价合成”（Normalization Form Canonical Composition），返回多个简单字符的合成字符。所谓“标准等价”指的是视觉和语义上的等价。
- NFD，表示“标准等价分解”（Normalization Form Canonical Decomposition），即在标准等价的前提下，返回合成字符分解的多个简单字符。
- NFKC，表示“兼容等价合成”（Normalization Form Compatibility Composition），返回合成字符。所谓“兼容等价”指的是语义上存在等价，但视觉上不等价，比如“囍”和“喜喜”。
- NFKD，表示“兼容等价分解”（Normalization Form Compatibility Decomposition），即在兼容等价的前提下，返回合成字符分解的多个简单字符。

```
'\u004F\u030C'.normalize('NFC').length // 1
'\u004F\u030C'.normalize('NFD').length // 2
```

上面代码表示，NFC 参数返回字符的合成形式，NFD 参数返回字符的分解形式。

不过，normalize 方法目前不能识别三个或三个以上字符的合成。这种情况下，还是只能使用正则表达式，通过 Unicode 编号区间判断。

#

includes(), startsWith(), endsWith()

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

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

```
var s = "Hello world!";

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

这三个方法都支持第二个参数，表示开始搜索的位置。

```
var s = "Hello world!";

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

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

#

repeat()

repeat()返回一个新字符串，表示将原字符串重复 n 次。

```
"x".repeat(3) // "xxx"
"hello".repeat(2) // "hellohello"
```

#

正则表达式的 y 修饰符

除了 u 修饰符，ES6 还为正则表达式添加了 y 修饰符，叫做“粘连”（sticky）修饰符。它的作用与 g 修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始，不同之处在于，g 修饰符只要剩余位置中存在匹配就可，而 y 修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

```
var s = "aaa_aa_a";
var r1 = /a+/g;
var r2 = /a+/y;

r1.exec(s) // ["aaa"]
r2.exec(s) // ["aaa"]

r1.exec(s) // ["aa"]
r2.exec(s) // null
```

上面代码有两个正则表达式，一个使用 g 修饰符，另一个使用 y 修饰符。这两个正则表达式各执行了两次，第一次执行的时候，两者行为相同，剩余字符串都是“_aa_a”。由于 g 修饰没有位置要求，所以第二次执行会返回结果，而 y 修饰符要求匹配必须从头部开始，所以返回 null。

如果改一下正则表达式，保证每次都能头部匹配，y 修饰符就会返回结果了。

```
var s = "aaa_aa_a";
var r = /a+_/y;

r.exec(s) // ["aaa_"]
r.exec(s) // ["aa_"]
```

上面代码每次匹配，都是从剩余字符串的头部开始。

进一步说，y 修饰符号隐含了头部匹配的标志 `&#710;` 。

```
/b/y.exec("aba")
// null
```

上面代码由于不能保证头部匹配，所以返回 null。y 修饰符的设计本意，就是让头部匹配的标志 `&#710;` 在全局匹配中都有效。

与 y 修饰符相匹配，ES6 的正则对象多了 sticky 属性，表示是否设置了 y 修饰符。

```
var r = /hello\d/y;
r.sticky // true
```

#

Regexp.escape()

字符串必须转义，才能作为正则模式。

```
function escapeRegExp(str) {
  return str.replace(/[\-\[\]\/\{\}\(\)\*\+\?\.\\\^\$\|]/g, "\\$&");
}

let str = '/path/to/resource.html?search=query';
escapeRegExp(str)
// "\/path\/to\/resource\.html\?search=query"
```

上面代码中，str 是一个正常字符串，必须使用反斜杠对其中的特殊字符转义，才能用来作为一个正则匹配的模式。

已经有提议将这个需求标准化，作为 `Regexp.escape()` ，放入 ES7。

```
RegExp.escape("The Quick Brown Fox");
// "The Quick Brown Fox"

RegExp.escape("Buy it. use it. break it. fix it.")
// "Buy it\. use it\. break it\. fix it\."

RegExp.escape("(*.*)");
// "\(\*\.\*\)"
```

字符串转义以后，可以使用 RegExp 构造函数生成正则模式。

```
var str = 'hello. how are you?';
var regex = new RegExp(RegExp.escape(str), 'g');
assert.equal(String(regex), '/hello\. how are you\?/g');
```

目前，该方法可以用上文的 escapeRegExp 函数或者垫片模块 `regexp.escape` 实现。

```
var escape = require('regexp.escape');
escape('hi. how are you?')
"hi\\. how are you\\?"
```

#

模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的。

```
$("#result").append(
  "There are <b>" + basket.count + "</b> " +
  "items in your basket, " +
  "<em>" + basket.onSale +
  "</em> are on sale!"
);
```

上面这种写法相当繁琐不方便，ES6 引入了模板字符串解决这个问题。

```
$("#result").append(`
  There are <b>${basket.count}</b> items
   in your basket, <em>${basket.onSale}</em>
  are on sale!
`);
```

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

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

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

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

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

上面代码中的字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。

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

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

```
$("#warning").html(`
  <h1>Watch out!</h1>
  <p>Unauthorized hockeying can result in penalties
  of up to ${maxPenalty} minutes.</p>
`);
```

模板字符串中嵌入变量，需要将变量名写在 `${}` 之中。

```
function authorize(user, action) {
  if (!user.hasPrivilege(action)) {
    throw new Error(
      // 传统写法为
      // 'User '
      // + user.name
      // + ' is not authorized to do '
      // + action
      // + '.'
      `User ${user.name} is not authorized to do ${action}.`);
  }
}
```

大括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属性。

```
var x = 1;
var y = 2;

console.log(`${x} + ${y} = ${x+y}`)
// "1 + 2 = 3"

console.log(`${x} + ${y*2} = ${x+y*2}`)
// "1 + 4 = 5"

var obj = {x: 1, y: 2};
console.log(`${obj.x + obj.y}`)
// 3
```

模板字符串之中还能调用函数。

```
function fn() {
  return "Hello World";
}

console.log(`foo ${fn()} bar`);
// foo Hello World bar
```

如果大括号中的值不是字符串，将按照一般的规则转为字符串。不如，大括号中是一个对象，将默认调用对象的 toString 方法。

如果模板字符串中的变量没有声明，将报错。

```
// 变量place没有声明
var msg = `Hello, ${place}`;
// 报错
```

#

标签模板

模板字符串的功能，不仅仅是上面这些。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为“标签模板”功能（tagged template）。

```
var a = 5;
var b = 10;

tag`Hello ${ a + b } world ${ a * b }`;
```

上面代码中，模板字符串前面有一个标识名 tag，它是一个函数。整个表达式的返回值，就是 tag 函数处理模板字符串后的返回值。

函数 tag 依次会接收到多个参数。

```
function tag(stringArr, value1, value2){
  // ...
}

// 等同于
function tag(stringArr, ...values){
  // ...
}
```

tag 函数的第一个参数是一个数组，该数组的成员是模板字符串中那些没有变量替换的部分，也就是说，变量替换只发生在数组的第一个成员与第二个成员之间、第二个成员与第三个成员之间，以此类推。

tag 函数的其他参数，都是模板字符串各个变量被替换后的值。由于本例中，模板字符串含有两个变量，因此tag会接受到 value1 和 value2 两个参数。

tag 函数所有参数的实际值如下。

- 第一个参数：['Hello ', ' world ']
- 第二个参数: 15
- 第三个参数：50

也就是说，tag 函数实际上以下面的形式调用。

```
tag(['Hello ', ' world '], 15, 50)
```

我们可以按照需要编写 tag 函数的代码。下面是 tag 函数的一种写法，以及运行结果。

```
var a = 5;
var b = 10;

function tag(s, v1, v2) {
  console.log(s[0]);
  console.log(s[1]);
  console.log(v1);
  console.log(v2);

  return "OK";
}

tag`Hello ${ a + b } world ${ a * b}`;
// "Hello "
// " world "
// 15
// 50
// "OK"
```

下面是一个更复杂的例子。

```
var total = 30;
var msg = passthru`The total is ${total} (${total*1.05} with tax)`;

function passthru(literals) {
  var result = "";
  var i = 0;

  while (i < literals.length) {
    result += literals[i++];
    if (i < arguments.length) {
      result += arguments[i];
    }
  }
```

```
  return result;

}

msg
// "The total is 30 (31.5 with tax)"
```

上面这个例子展示了，如何将各个参数按照原来的位置拼合回去。

passthru 函数采用 rest 参数的写法如下。

```
function passthru(literals,...values) {
  var output = "";
  for (var index = 0; index < values.length; index++) {
    output += literals[index] + values[index];
  }

  output += literals[index]
  return output;
}
```

“标签模板”的一个重要应用，就是过滤 HTML 字符串，防止用户输入恶意内容。

```
var message =
  SaferHTML`<p>${sender} has sent you a message.</p>`;

function SaferHTML(templateData) {
  var s = templateData[0];
  for (var i = 1; i < arguments.length; i++) {
    var arg = String(arguments[i]);

    // Escape special characters in the substitution.
    s += arg.replace(/&/g, "&amp;")
            .replace(/</g, "&lt;")
            .replace(/>/g, "&gt;");

    // Don't escape special characters in the template.
    s += templateData[i];
  }
  return s;
}
```

上面代码中，经过 SaferHTML 函数处理，HTML 字符串的特殊字符都会被转义。

标签模板的另一个应用，就是多语言转换（国际化处理）。

```
i18n`Hello ${name}, you have ${amount}:c(CAD) in your bank account.`
// Hallo Bob, Sie haben 1.234,56 $CA auf Ihrem Bankkonto.
```

模板字符串本身并不能取代 Mustache 之类的模板函数，因为没有条件判断和循环处理功能，但是通过标签函数，你可以自己添加这些功能。

```
// 下面的hashTemplate函数
// 是一个自定义的模板处理函数
var libraryHtml = hashTemplate`
  <ul>
    #for book in ${myBooks}
      <li><i>#{book.title}</i> by #{book.author}</li>
    #end
  </ul>
`;
```

除此之外，你甚至可以使用标签模板，在 JavaScript 语言之中嵌入其他语言。

```
java`
class HelloWorldApp {
  public static void main(String[] args) {
    System.out.println(“Hello World!”); // Display the string.
  }
}
`
HelloWorldApp.main();
```

模板处理函数的第一个参数（模板字符串数组），还有一个 raw 属性。

```
tag`First line\nSecond line`

function tag(strings) {
  console.log(strings.raw[0]);
  // "First line\\nSecond line"
}
```

上面代码中，tag 函数的第一个参数 strings，有一个 raw 属性，也指向一个数组。该数组的成员与 strings 数组完全一致。比如，strings数组是 `["First line\nSecond line"]` ，那么 strings.raw 数组就是 `["First line\\nSecond line"]` 。两者唯一的区别，就是字符串里面的斜杠都被转义了。比如，strings.raw 数组会将 `\n` 视为\和n两个字符，而不是换行符。这是为了方便取得转义之前的原始模板而设计的。

#

String.raw()

ES6 还为原生的 String 对象，提供了一个 raw 方法。

String.raw 方法，往往用来充当模板字符串的处理函数，返回一个斜杠都被转义（即斜杠前面再加一个斜杠）的字符串，对应于替换变量后的模板字符串。

```
String.raw`Hi\n${2+3}!`;
// "Hi\\n5!"

String.raw`Hi\u000A!`;
// 'Hi\\u000A!'
```

它的代码基本如下。

```
String.raw = function (strings,...values) {
  var output = "";
  for (var index = 0; index < values.length; index++) {
    output += strings.raw[index] + values[index];
  }

  output += strings.raw[index]
  return output;
}
```

String.raw 方法可以作为处理模板字符串的基本方法，它会将所有变量替换，而且对斜杠进行转义，方便下一步作为字符串来使用。

String.raw 方法也可以作为正常的函数使用。这时，它的第一个参数，应该是一个具有 raw 属性的对象，且 raw 属性的值应该是一个数组。

```
String.raw({ raw: 'test' }, 0, 1, 2);
// 't0e1s2t'

// 等同于
String.raw({ raw: ['t','e','s','t'] }, 0, 1, 2);
```

6

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
```

八进制用 0o 前缀表示的方法，将要取代已经在 ES5 中被逐步淘汰的加前缀 0 的写法。

#

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&#710;53 and 2&#710;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

（1）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
```

（2）Math.sign()

Math.sign 方法用来判断一个数到底是正数、负数、还是零。如果参数为正数，返回 +1；参数为负数，返回 -1；参数为 0，返回 0；参数为 NaN，返回 NaN。

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

ES5 通过下面的代码，可以部署 Math.sign()。

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

  Object.defineProperty(Math, 'sign', {
    value: function sign(value) {
      var n = +value;
      if (isNaN(n))
        return n /* NaN */;

      if (n === 0)
        return n; // Keep the sign of the zero.

      return (n < 0) ? -1 : 1;
    },
    configurable: true,
    enumerable: false,
    writable: true
  });
})(this);
```

（3）数学方法

ES6 在 Math 对象上还提供了许多新的数学方法。

- Math.acosh(x) 返回 x 的反双曲余弦（inverse hyperbolic cosine）
- Math.asinh(x) 返回 x 的反双曲正弦（inverse hyperbolic sine）
- Math.atanh(x) 返回 x 的反双曲正切（inverse hyperbolic tangent）
- Math.cbrt(x) 返回 x 的立方根
- Math.clz32(x) 返回 x 的 32 位二进制整数表示形式的前导 0 的个数
- Math.cosh(x) 返回 x 的双曲余弦（hyperbolic cosine）
- Math.expm1(x) 返回 e^x − 1
- Math.fround(x) 返回 x 的单精度浮点数形式
- Math.hypot(...values) 返回所有参数的平方和的平方根
- Math.imul(x, y) 返回两个参数以 32 位整数形式相乘的结果
- Math.log1p(x) 返回 1 + x 的自然对数
- Math.log10(x) 返回以 10 为底的x的对数
- Math.log2(x) 返回以 2 为底的 x 的对数
- Math.tanh(x) 返回 x 的双曲正切（hyperbolic tangent）

7

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

ES6 提供简洁写法，允许直接通过现有数组生成新数组，这被称为数组推导（array comprehension）。

```
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）。

8

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
```

#

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。

ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例。

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

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

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；第二个参数是一个设置对象，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，设置对象有一个 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。

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。

对于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。

Proxy 支持的拦截操作一览。

- defineProperty(target, propKey, propDesc)：返回一个布尔值，拦截Object.defineProperty(proxy, propKey, propDesc)
- deleteProperty(target, propKey)：返回一个布尔值，拦截delete proxy[propKey]
- enumerate(target)：返回一个遍历器，拦截for (x in proxy)
- get(target, propKey, receiver)：返回类型不限，拦截对象属性的读取
- getOwnPropertyDescriptor(target, propKey)：返回属性的描述对象，拦截Object.getOwnPropertyDescriptor(proxy, propKey)
- getPrototypeOf(target)：返回一个对象，拦截Object.getPrototypeOf(proxy)
- has(target, propKey)：返回一个布尔值，拦截propKey in proxy
- isExtensible(target)：返回一个布尔值，拦截Object.isExtensible(proxy)
- ownKeys(target)：返回一个数组，拦截Object.getOwnPropertyPropertyNames(proxy)、Object.getOwnPropertyPropertySymbols(proxy)、Object.keys(proxy)
- preventExtensions(target)：返回一个布尔值，拦截Object.preventExtensions(proxy)
- set(target, propKey, value, receiver)：返回一个布尔值，拦截对象属性的设置
- setPrototypeOf(target, proto)：返回一个布尔值，拦截Object.setPrototypeOf(proxy, proto)

如果目标对象是函数，那么还有两种额外操作可以拦截。

- apply 方法：拦截 Proxy 实例作为函数调用的操作，比如 proxy(···)、proxy.call(···)、proxy.apply(···)。
- construct 方法：拦截 Proxy 实例作为构造函数调用的操作，比如 new proxy(···)。

#

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。

#

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。

9

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。

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
```

#

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);
```

箭头函数还有一个功能，就是可以很方便地改写 λ 微积分。

```
// λ微积分的写法
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 微积分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。

# #

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。

以下两种情况，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;
}
```

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。

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。

10

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。

11

Iterator 和 for...of 循环

# #

Iterator（遍历器）

#

概念

JavaScript 原有的数据结构，主要是数组（Array）和对象（Object），ES6 又添加了 Map 和 Set，用户还可以组合使用它们，定义自己的数据结构。这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。它属于一种接口规格，任何数据结构只要部署这个接口，就可以完成遍历操作，即依次处理该结构的所有成员。它的作用有两个，一是为各种数据结构，提供一个统一的、简便的接口，二是使得数据结构的成员能够按某种次序排列。在 ES6 中，遍历操作特指 for...of 循环，即 Iterator 接口主要供 for...of 消费。

遍历器的遍历过程是这样的：它提供了一个指针，默认指向当前数据结构的起始位置。也就是说，遍历器返回一个内部指针。第一次调用遍历器的 next 方法，可以将指针指向到第一个成员，第二次调用 next 方法，就指向第二个成员，直至指向数据结构的结束位置。每一次调用，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遍历器。对数组 ['a', 'b'] 执行这个函数，就会返回该数组的遍历器 it。

遍历器 it 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，表示当前位置的信息。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，遍历器是一个对象，具有 next 方法。调用 next 方法，就可以遍历事先给定的数据结构。

由于遍历器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，idMaker 函数返回的对象就是遍历器，但是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，这些数据结构部署了 System.iterator 属性（详见下文）。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，就会返回一个遍历器。

如果使用 TypeScript 的写法，遍历器接口、遍历器和 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,
}
```

#

默认的 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

部署 Symbol.iterator 方法的最简单实现，还是使用下一节要提到的 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
```

#

return()，throw()

遍历器除了具有 next 方法（必备），还可以具有 return 方法和 throw 方法（可选）。

for...of 循环如果提前退出（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》的章节。

# #

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。

12

Generator 函数

# #

简介

所谓 Generator，有多种理解角度。首先，可以把它理解成一个函数的内部状态的遍历器，每调用一次，函数的内部状态发生一次改变（可以理解成发生某些事件）。ES6 引入 Generator 函数，作用就是可以完全控制函数的内部状态的变化，依次遍历这些状态。

在形式上，Generator 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，它的遍历器有两个成员“hello”和“world”。调用这个函数，就会得到遍历器。

当调用 Generator 函数的时候，该函数并不执行，而是返回一个遍历器（可以理解成暂停执行）。以后，每次调用这个遍历器的 next 方法，就从函数体的头部或者上一次停下来的地方开始执行（可以理解成恢复执行），直到遇到下一个 yield 语句为止。也就是说，next 方法就是在遍历 yield 语句定义的内部状态。

```
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，函数开始执行，直到遇到第一句 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，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，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，Generator 函数使用 iterator 接口，每次调用 next 方法的返回值，就是一个标准的 iterator 返回值：有着 value 和 done 两个属性的对象。其中，value 是 yield 语句后面那个表达式的值，done 是一个布尔值，表示是否遍历结束。

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，即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。

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成员，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，next 方法遇到 yield，就会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，作为返回对象的 value 属性的值。当下一次调用next方法时，再继续往下执行，直到遇到下一个 yield 语句。如果没有再遇到新的 yield 语句，就一直运行到函数结束，将 return 语句后面的表达式的值，作为 value 属性的值，如果该函数没有 return 语句，则 value 属性的值为 undefined。另一方面，由于 yield 后面的表达式，直到调用 next 方法时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

yield 语句与 return 语句有点像，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
```

#

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。

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，在 ES6 只能用于数组，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。

13

Promise 对象

#

基本用法

ES6 原生提供了 Promise 对象。所谓 Promise 对象，就是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。

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 对象提供的接口，使得控制异步操作更加容易。Promise 对象的概念的详细解释，请参考《JavaScript标准参考教程》。

ES6 的 Promise 对象是一个构造函数，用来生成 Promise 实例。

```
var promise = new Promise(function(resolve, reject) {
  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});

promise.then(function(value) {
  // success
}, function(value) {
  // failure
});
```

上面代码中，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 方法和 reject 方法。如果异步操作成功，则用 resolve 方法将 Promise 对象的状态，从“未完成”变为“成功”（即从 pending 变为 resolved）；如果异步操作失败，则用 reject 方法将 Promise 对象的状态，从“未完成”变为“失败”（即从 pending 变为 rejected）。

Promise 实例生成以后，可以用 then 方法分别指定 resolve 方法和 reject 方法的回调函数。

下面是一个使用 Promise 对象的简单例子。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

timeout(100).then(() => {
  console.log('done');
});
```

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。一旦 Promise 对象的状态变为 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 方法和 reject 方法调用时，都带有参数。

如果调用 resolve 方法和 reject 方法时带有参数，那么它们的参数会被传递给回调函数。reject 方法的参数通常是 Error 对象的实例，表示抛出的错误；resolve 方法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，p1 的状态就会传递给 p2。

注意，这时 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 fulfilled 或者 rejected，那么 p2 的回调函数将会立刻执行。

#

Promise.prototype.then()

Promise.prototype.then 方法返回的是一个新的Promise对象，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。

如果前一个回调函数返回的是Promise对象，这时后一个回调函数就会等待该Promise对象有了运行结果，才会进一步调用。

```
getJSON("/post/1.json").then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // ...
});
```

then方法还可以接受第二个参数，表示Promise对象的状态变为rejected时的回调函数。

#

Promise.prototype.catch()

Promise.prototype.catch方法是 Promise.prototype.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，如果该对象运行正常，则会调用then方法指定的回调函数；如果该方法抛出错误，则会调用catch方法指定的回调函数，处理这个错误。

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，也不会到传递到外层。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.race()

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的返回值。

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()，Promise.reject()

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ES6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fulfilled。

```
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，它的状态为fulfilled，所以回调函数会立即执行，Promise.resolve方法的参数就是回调函数的参数。

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对象的实例，则会被原封不动地返回。

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

#

概述

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。

下面是一个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());
};
```

上面代码中，readFile函数是 fs.readFile 的Promise版本。

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，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函数库约定，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）。

#

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); });
```

```
  });
}
```

#

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。

上面的例子用Generator函数表达，就是下面这样。

```
function getStockPriceByName(name) {
  return spawn(function*(name) {
    var symbol = yield getStockSymbol(name);
    var stockPrice = yield getStockPrice(symbol);
    return stockPrice;
  });
}
```

上面的例子中，spawn函数是一个自动执行器，由JavaScript引擎内置。它的参数是一个Generator函数。async...await结构本质上，是在语言层面提供的异步任务的自动执行器。

下面是一个更一般性的例子，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。

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。Haskell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函数库

如果并发执行异步操作，可以将异步操作都放入一个数组，跟在yield语句后面。

```
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。

15

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 = "?";

s.length // 2
s.charAt(0) // "
s.charAt(1) // "
s.charCodeAt(0) // 55362
s.charCodeAt(1) // 57271
```

上面代码中，汉字“?”的码点是 0x20BB7，UTF-16 编码为 0xD842 0xDFB7（十进制为 55362 57271），需要 4 个字节储存。对于这种 4 个字节的字符，JavaScript 不能正确处理，字符串长度会误判为 2，而且 charAt 方法无法读取字符，charCodeAt 方法只能分别返回前两个字节和后两个字节的值。

ES6 提供了 codePointAt 方法，能够正确处理 4 个字节储存的字符，返回一个字符的码点。

```
var s = "?a";

s.codePointAt(0) // 134071
s.codePointAt(1) // 57271

s.charCodeAt(2) // 97
```

codePointAt 方法的参数，是字符在字符串中的位置（从 0 开始）。上面代码中，JavaScript 将“?a”视为三个字符，codePointAt 方法在第一个字符上，正确地识别了“?”，返回了它的十进制码点 134071（即十六进制的20BB7）。在第二个字符（即“?”的后两个字节）和第三个字符“a”上，codePointAt 方法的结果与 charCodeAt 方法相同。

总之，codePointAt 方法会正确返回四字节的 UTF-16 字符的码点。对于那些两个字节储存的常规字符，它的返回结果与 charCodeAt 方法相同。

codePointAt 方法是测试一个字符由两个字节还是由四个字节组成的最简单方法。

```
function is32Bit(c) {
  return c.codePointAt(0) > 0xFFFF;
}
```

```
is32Bit("?") // true
is32Bit("a") // false
```

#

String.fromCodePoint()

ES5 提供 String.fromCharCode 方法，用于从码点返回对应字符，但是这个方法不能识别辅助平面的字符（编号大于 0xFFFF）。

```
String.fromCharCode(0x20BB7)
// "?"
```

上面代码中，最后返回码点 U+0BB7 对应的字符，而不是码点 U+20BB7 对应的字符。

ES6 提供了 String.fromCodePoint 方法，可以识别 0xFFFF 的字符，弥补了 String.fromCharCode 方法的不足。在作用上，正好与 codePointAt 方法相反。

```
String.fromCodePoint(0x20BB7)
// "?"
```

注意，fromCodePoint 方法定义在 String 对象上，而 codePointAt 方法定义在字符串的实例对象上。

#

String.prototype.at()

ES5 提供 String.prototype.charAt 方法，返回字符串给定位置的字符。该方法不能识别码点大于 0xFFFF 的字符。

```
'?'.charAt(0)
// '\uD842'
```

上面代码中，charAt 方法返回的是 UTF-16 编码的第一个字节，实际上是无法显示的。

ES7 提供了字符串实例的 at 方法，可以识别 Unicode 编号大于 0xFFFF 的字符，返回正确的字符。

```
'?'.at(0)
// '?'
```

#

字符的 Unicode 表示法

JavaScript 允许采用“\uxxxx”形式表示一个字符，其中“xxxx”表示字符的码点。

```
"\u0061"
// "a"
```

但是，这种表示法只限于 \u0000——\uFFFF 之间的字符。超出这个范围的字符，必须用两个双字节的形式表达。

```
"\uD842\uDFB7"
// "?"

"\u20BB7"
// " 7"
```

上面代码表示，如果直接在“\u”后面跟上超过 0xFFFF 的数值（比如 \u20BB7 ），JavaScript 会理解成“\u20BB+7”。由于 \u20BB 是一个不可打印字符，所以只会显示一个空格，后面跟着一个 7。

ES6 对这一点做出了改进，只要将码点放入大括号，就能正确解读该字符。

```
"\u{20BB7}"
// "?"

"\u{41}\u{42}\u{43}"
// "ABC"
```

#

正则表达式的 u 修饰符

ES6 对正则表达式添加了 u 修饰符，用来正确处理大于 \uFFFF 的 Unicode 字符。

（1）点字符

点（.）字符在正则表达式中，解释为除了换行以外的任意单个字符。对于码点大于 0xFFFF 的 Unicode 字符，点字符不能识别，必须加上 u 修饰符。

```
var s = "?";
```

```
/^.$/.test(s) // false
/^.$/u.test(s) // true
```

上面代码表示，如果不添加 u 修饰符，正则表达式就会认为字符串为两个字符，从而匹配失败。

（2）Unicode字符表示法

ES6 新增了使用大括号表示 Unicode 字符，这种表示法在正则表达式中必须加上 u 修饰符，才能识别。

```
/\u{61}/.test('a') // false
/\u{61}/u.test('a') // true
/\u{20BB7}/u.test('?') // true
```

上面代码表示，如果不加 u 修饰符，正则表达式无法识别 `\u{61}` 这种表示法，只会认为这匹配 61 个连续的 u。

（3）量词

使用 u 修饰符后，所有量词都会正确识别大于码点大于 0xFFFF 的 Unicode 字符。

```
/a{2}/.test('aa') // true
/a{2}/u.test('aa') // true
/?{2}/.test('??') // false
/?{2}/u.test('??') // true
```

（4）预定义模式

u 修饰符也影响到预定义模式，能否正确识别码点大于 0xFFFF 的 Unicode 字符。

```
/^\S$/.test('?') // false
/^\S$/u.test('?')
```

上面代码的 `\S` 是预定义模式，匹配所有不是空格的字符。只有加了 u 修饰符，它才能正确匹配码点大于 0xFFFF 的 Unicode 字符。

利用这一点，可以写出一个正确返回字符串长度的函数。

```
function codePointLength(text) {
  var result = text.match(/[\s\S]/gu);
  return result ? result.length : 0;
}

var s = "??";

s.length // 4
codePointLength(s) // 2
```

（5）i 修饰符

有些 Unicode 字符的编码不同，但是字型很相近，比如， \u004B 与 \u212A 都是大写的 K。

```
/[a-z]/i.test('\u212A') // false
/[a-z]/iu.test('\u212A') // true
```

上面代码中，不加 u 修饰符，就无法识别非规范的 K 字符。

#

normalize()

为了表示语调和重音符号，Unicode 提供了两种方法。一种是直接提供带重音符号的字符，比如 Ǒ（\u01D1）。另一种是提供合成符号（combining character），即原字符与重音符号的合成，两个字符合成一个字符，比如 O（\u004F）和 ˇ（\u030C）合成 Ǒ（\u004F\u030C）。

这两种表示方法，在视觉和语义上都等价，但是 JavaScript 不能识别。

```
'\u01D1'==='\u004F\u030C' //false

'\u01D1'.length // 1
'\u004F\u030C'.length // 2
```

上面代码表示，JavaScript 将合成字符视为两个字符，导致两种表示方法不相等。

ES6 提供 String.prototype.normalize()方法，用来将字符的不同表示方法统一为同样的形式，这称为 Unicode 正规化。

```
'\u01D1'.normalize() === '\u004F\u030C'.normalize()
// true
```

normalize 方法可以接受四个参数。

- NFC，默认参数，表示“标准等价合成”（Normalization Form Canonical Composition），返回多个简单字符的合成字符。所谓“标准等价”指的是视觉和语义上的等价。
- NFD，表示“标准等价分解”（Normalization Form Canonical Decomposition），即在标准等价的前提下，返回合成字符分解的多个简单字符。
- NFKC，表示“兼容等价合成”（Normalization Form Compatibility Composition），返回合成字符。所谓“兼容等价”指的是语义上存在等价，但视觉上不等价，比如“囍”和“喜喜”。
- NFKD，表示“兼容等价分解”（Normalization Form Compatibility Decomposition），即在兼容等价的前提下，返回合成字符分解的多个简单字符。

```
'\u004F\u030C'.normalize('NFC').length // 1
'\u004F\u030C'.normalize('NFD').length // 2
```

上面代码表示，NFC 参数返回字符的合成形式，NFD 参数返回字符的分解形式。

不过，normalize 方法目前不能识别三个或三个以上字符的合成。这种情况下，还是只能使用正则表达式，通过 Unicode 编号区间判断。

#

includes(), startsWith(), endsWith()

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

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

```
var s = "Hello world!";

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

这三个方法都支持第二个参数，表示开始搜索的位置。

```
var s = "Hello world!";

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

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

#

repeat()

repeat()返回一个新字符串，表示将原字符串重复 n 次。

```
"x".repeat(3) // "xxx"
"hello".repeat(2) // "hellohello"
```

#

正则表达式的 y 修饰符

除了 u 修饰符，ES6 还为正则表达式添加了 y 修饰符，叫做“粘连”（sticky）修饰符。它的作用与 g 修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始，不同之处在于，g 修饰符只要剩余位置中存在匹配就可，而 y 修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

```
var s = "aaa_aa_a";
var r1 = /a+/g;
var r2 = /a+/y;

r1.exec(s) // ["aaa"]
r2.exec(s) // ["aaa"]

r1.exec(s) // ["aa"]
r2.exec(s) // null
```

上面代码有两个正则表达式，一个使用 g 修饰符，另一个使用 y 修饰符。这两个正则表达式各执行了两次，第一次执行的时候，两者行为相同，剩余字符串都是“_aa_a”。由于 g 修饰没有位置要求，所以第二次执行会返回结果，而 y 修饰符要求匹配必须从头部开始，所以返回 null。

如果改一下正则表达式，保证每次都能头部匹配，y 修饰符就会返回结果了。

```
var s = "aaa_aa_a";
var r = /a+_/y;

r.exec(s) // ["aaa_"]
r.exec(s) // ["aa_"]
```

上面代码每次匹配，都是从剩余字符串的头部开始。

进一步说，y 修饰符号隐含了头部匹配的标志 `&#710;` 。

```
/b/y.exec("aba")
// null
```

上面代码由于不能保证头部匹配，所以返回 null。y 修饰符的设计本意，就是让头部匹配的标志 `&#710;` 在全局匹配中都有效。

与 y 修饰符相匹配，ES6 的正则对象多了 sticky 属性，表示是否设置了 y 修饰符。

```
var r = /hello\d/y;
r.sticky // true
```

#

Regexp.escape()

字符串必须转义，才能作为正则模式。

```
function escapeRegExp(str) {
  return str.replace(/[\-\[\]\/\{\}\(\)\*\+\?\.\\\^\$\|]/g, "\\$&");
}

let str = '/path/to/resource.html?search=query';
escapeRegExp(str)
// "\/path\/to\/resource\.html\?search=query"
```

上面代码中，str 是一个正常字符串，必须使用反斜杠对其中的特殊字符转义，才能用来作为一个正则匹配的模式。

已经有提议将这个需求标准化，作为 `Regexp.escape()` ，放入 ES7。

```
RegExp.escape("The Quick Brown Fox");
// "The Quick Brown Fox"

RegExp.escape("Buy it. use it. break it. fix it.")
// "Buy it\. use it\. break it\. fix it\."

RegExp.escape("(*.*)");
// "\(\*\.\*\)"
```

字符串转义以后，可以使用 RegExp 构造函数生成正则模式。

```
var str = 'hello. how are you?';
var regex = new RegExp(RegExp.escape(str), 'g');
assert.equal(String(regex), '/hello\. how are you\?/g');
```

目前，该方法可以用上文的 escapeRegExp 函数或者垫片模块 `regexp.escape` 实现。

```
var escape = require('regexp.escape');
escape('hi. how are you?')
"hi\\. how are you\\?"
```

#

模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的。

```
$("#result").append(
  "There are <b>" + basket.count + "</b> " +
  "items in your basket, " +
  "<em>" + basket.onSale +
  "</em> are on sale!"
);
```

上面这种写法相当繁琐不方便，ES6 引入了模板字符串解决这个问题。

```
$("#result").append(`
  There are <b>${basket.count}</b> items
   in your basket, <em>${basket.onSale}</em>
  are on sale!
`);
```

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

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

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

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

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

上面代码中的字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。

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

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

```
$("#warning").html(`
  <h1>Watch out!</h1>
  <p>Unauthorized hockeying can result in penalties
  of up to ${maxPenalty} minutes.</p>
`);
```

模板字符串中嵌入变量，需要将变量名写在 `${}` 之中。

```
function authorize(user, action) {
  if (!user.hasPrivilege(action)) {
    throw new Error(
      // 传统写法为
      // 'User '
      // + user.name
      // + ' is not authorized to do '
      // + action
      // + '.'
      `User ${user.name} is not authorized to do ${action}.`);
  }
}
```

大括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属性。

```
var x = 1;
var y = 2;

console.log(`${x} + ${y} = ${x+y}`)
// "1 + 2 = 3"

console.log(`${x} + ${y*2} = ${x+y*2}`)
// "1 + 4 = 5"

var obj = {x: 1, y: 2};
console.log(`${obj.x + obj.y}`)
// 3
```

模板字符串之中还能调用函数。

```
function fn() {
  return "Hello World";
}

console.log(`foo ${fn()} bar`);
// foo Hello World bar
```

如果大括号中的值不是字符串，将按照一般的规则转为字符串。不如，大括号中是一个对象，将默认调用对象的 toString 方法。

如果模板字符串中的变量没有声明，将报错。

```
// 变量place没有声明
var msg = `Hello, ${place}`;
// 报错
```

#

标签模板

模板字符串的功能，不仅仅是上面这些。它可以紧跟在一个函数名后面，该函数将被调用来处理这个模板字符串。这被称为“标签模板”功能（tagged template）。

```
var a = 5;
var b = 10;

tag`Hello ${ a + b } world ${ a * b }`;
```

上面代码中，模板字符串前面有一个标识名 tag，它是一个函数。整个表达式的返回值，就是 tag 函数处理模板字符串后的返回值。

函数 tag 依次会接收到多个参数。

```
function tag(stringArr, value1, value2){
  // ...
}

// 等同于
function tag(stringArr, ...values){
  // ...
}
```

tag 函数的第一个参数是一个数组，该数组的成员是模板字符串中那些没有变量替换的部分，也就是说，变量替换只发生在数组的第一个成员与第二个成员之间、第二个成员与第三个成员之间，以此类推。

tag 函数的其他参数，都是模板字符串各个变量被替换后的值。由于本例中，模板字符串含有两个变量，因此tag会接受到 value1 和 value2 两个参数。

tag 函数所有参数的实际值如下。

- 第一个参数：['Hello ', ' world ']
- 第二个参数: 15
- 第三个参数：50

也就是说，tag 函数实际上以下面的形式调用。

```
tag(['Hello ', ' world '], 15, 50)
```

我们可以按照需要编写 tag 函数的代码。下面是 tag 函数的一种写法，以及运行结果。

```
var a = 5;
var b = 10;

function tag(s, v1, v2) {
  console.log(s[0]);
  console.log(s[1]);
  console.log(v1);
  console.log(v2);

  return "OK";
}

tag`Hello ${ a + b } world ${ a * b}`;
// "Hello "
// " world "
// 15
// 50
// "OK"
```

下面是一个更复杂的例子。

```
var total = 30;
var msg = passthru`The total is ${total} (${total*1.05} with tax)`;

function passthru(literals) {
  var result = "";
  var i = 0;

  while (i < literals.length) {
    result += literals[i++];
    if (i < arguments.length) {
      result += arguments[i];
    }
  }
```

```
  return result;

}

msg
// "The total is 30 (31.5 with tax)"
```

上面这个例子展示了，如何将各个参数按照原来的位置拼合回去。

passthru 函数采用 rest 参数的写法如下。

```
function passthru(literals,...values) {
  var output = "";
  for (var index = 0; index < values.length; index++) {
    output += literals[index] + values[index];
  }

  output += literals[index]
  return output;
}
```

“标签模板”的一个重要应用，就是过滤 HTML 字符串，防止用户输入恶意内容。

```
var message =
  SaferHTML`<p>${sender} has sent you a message.</p>`;

function SaferHTML(templateData) {
  var s = templateData[0];
  for (var i = 1; i < arguments.length; i++) {
    var arg = String(arguments[i]);

    // Escape special characters in the substitution.
    s += arg.replace(/&/g, "&amp;")
            .replace(/</g, "&lt;")
            .replace(/>/g, "&gt;");

    // Don't escape special characters in the template.
    s += templateData[i];
  }
  return s;
}
```

上面代码中，经过 SaferHTML 函数处理，HTML 字符串的特殊字符都会被转义。

标签模板的另一个应用，就是多语言转换（国际化处理）。

```
i18n`Hello ${name}, you have ${amount}:c(CAD) in your bank account.`
// Hallo Bob, Sie haben 1.234,56 $CA auf Ihrem Bankkonto.
```

模板字符串本身并不能取代 Mustache 之类的模板函数，因为没有条件判断和循环处理功能，但是通过标签函数，你可以自己添加这些功能。

```
// 下面的hashTemplate函数
// 是一个自定义的模板处理函数
var libraryHtml = hashTemplate`
  <ul>
    #for book in ${myBooks}
      <li><i>#{book.title}</i> by #{book.author}</li>
    #end
  </ul>
`;
```

除此之外，你甚至可以使用标签模板，在 JavaScript 语言之中嵌入其他语言。

```
java`
class HelloWorldApp {
  public static void main(String[] args) {
    System.out.println(“Hello World!”); // Display the string.
  }
}
`
HelloWorldApp.main();
```

模板处理函数的第一个参数（模板字符串数组），还有一个 raw 属性。

```
tag`First line\nSecond line`

function tag(strings) {
  console.log(strings.raw[0]);
  // "First line\\nSecond line"
}
```

上面代码中，tag 函数的第一个参数 strings，有一个 raw 属性，也指向一个数组。该数组的成员与 strings 数组完全一致。比如，strings数组是 `["First line\nSecond line"]` ，那么 strings.raw 数组就是 `["First line\\nSecond line"]` 。两者唯一的区别，就是字符串里面的斜杠都被转义了。比如，strings.raw 数组会将 `\n` 视为\和n两个字符，而不是换行符。这是为了方便取得转义之前的原始模板而设计的。

#

String.raw()

ES6 还为原生的 String 对象，提供了一个 raw 方法。

String.raw 方法，往往用来充当模板字符串的处理函数，返回一个斜杠都被转义（即斜杠前面再加一个斜杠）的字符串，对应于替换变量后的模板字符串。

```
String.raw`Hi\n${2+3}!`;
// "Hi\\n5!"

String.raw`Hi\u000A!`;
// 'Hi\\u000A!'
```

它的代码基本如下。

```
String.raw = function (strings,...values) {
  var output = "";
  for (var index = 0; index < values.length; index++) {
    output += strings.raw[index] + values[index];
  }

  output += strings.raw[index]
  return output;
}
```

String.raw 方法可以作为处理模板字符串的基本方法，它会将所有变量替换，而且对斜杠进行转义，方便下一步作为字符串来使用。

String.raw 方法也可以作为正常的函数使用。这时，它的第一个参数，应该是一个具有 raw 属性的对象，且 raw 属性的值应该是一个数组。

```
String.raw({ raw: 'test' }, 0, 1, 2);
// 't0e1s2t'

// 等同于
String.raw({ raw: ['t','e','s','t'] }, 0, 1, 2);
```

16

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，以及如何形成良好的编码风格。

#

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';
```

# #

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;
```

# #

函数

使用匿名函数的场合，一律改为使用箭头函数。

```
// bad
arr.reduce(function(x, y) { return x + y; }, 0);

// good
arr.reduce((x, y) => x + y, 0);
```

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 } = {}) {
}
```

#

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]);
}
```

# #

模块

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
```

17

参考链接

# #

官方文件

- ECMAScript 6 Language Specification: 语言规格草案
- harmony:proposals: ES6的各种提案
- Draft Specification for ES.next (Ecma-262 Edition 6): ES6草案各版本之间的变动

# #

综合介绍

- Sayanee Basu, Use ECMAScript 6 Today
- Ariya Hidayat, Toward Modern Web Apps with ECMAScript 6
- Dale Schouten, 10 Ecmascript-6 tricks you can perform right now
- Colin Toh, Lightweight ES6 Features That Pack A Punch: ES6的一些“轻量级”的特性介绍
- Domenic Denicola, ES6: The Awesome Parts
- Nicholas C. Zakas, Understanding ECMAScript 6
- Justin Drake, ECMAScript 6 in Node.JS
- Ryan Dao, Summary of ECMAScript 6 major features
- Luke Hoban, ES6 features: ES6新语法点的罗列
- Traceur-compiler, Language Features: Traceur文档列出的一些ES6例子
- Axel Rauschmayer, ECMAScript 6: what’s next for JavaScript?: 关于ES6新增语法的综合介绍，有很多例子
- Toby Ho, ES6 in io.js
- Guillermo Rauch, ECMAScript 6
- Charles King, The power of ECMAScript 6
- Benjamin De Cock, Frontend Guidelines: ES6最佳实践
- Jani Hartikainen, ES6: What are the benefits of the new features in practice?

# #

语法点

- Kyle Simpson, For and against `let`: 讨论let命令的作用域
- kangax, Why `typeof` is no longer “safe”: 讨论在块级作用域内，let命令的变量声明和赋值的行为
- Axel Rauschmayer, Variables and scoping in ECMAScript 6: 讨论块级作用域与let和const的行为
- Nick Fitzgerald, Destructuring Assignment in ECMAScript 6: 详细介绍解构赋值的用法
- Nicholas C. Zakas, Understanding ECMAScript 6 arrow functions
- Jack Franklin, Real Life ES6 − Arrow Functions
- Axel Rauschmayer, Handling required parameters in ECMAScript 6
- Axel Rauschmayer, ECMAScript 6’s new array methods: 对ES6新增的数组方法的全面介绍
- Dmitry Soshnikov, ES6 Notes: Default values of parameters: 介绍参数的默认值
- Ragan Wald, Destructuring and Recursion in ES6: rest参数和扩展运算符的详细介绍

# #

Collections

- Mozilla Developer Network, WeakSet：介绍WeakSet数据结构
- Dwayne Charrington, What Are Weakmaps In ES6?: WeakMap数据结构介绍
- Axel Rauschmayer, ECMAScript 6: maps and sets: Set和Map结构的详细介绍
- Jason Orendorff, ES6 In Depth: Collections：Set和Map结构的设计思想

# #

字符串

- Mathias Bynens, Unicode-aware regular expressions in ES6: 详细介绍正则表达式的u修饰符
- Nicholas C. Zakas, A critical review of ECMAScript 6 quasi-literals
- Mozilla Developer Network, Template strings
- Addy Osmani, Getting Literal With ES6 Template Strings: 模板字符串的介绍
- Blake Winton, ES6 Templates: 模板字符串的介绍

# #

Object

- Nicholas C. Zakas, Creating defensive objects with ES6 proxies
- Addy Osmani, Data-binding Revolutions with Object.observe(): 介绍Object.observe()的概念
- Sella Rafaeli, Native JavaScript Data-Binding: 如何使用Object.observe方法，实现数据对象与DOM对象的双向绑定
- Axel Rauschmayer, Symbols in ECMAScript 6: Symbol简介
- Axel Rauschmayer, Meta programming with ECMAScript 6 proxies: Proxy详解
- Daniel Zautner, Meta-programming JavaScript Using Proxies: 使用Proxy实现元编程

# #

Symbol

- MDN, Symbol: Symbol类型的详细介绍
- Jason Orendorff, ES6 In Depth: Symbols
- Keith Cirkel, Metaprogramming in ES6: Symbols and why they're awesome: Symbol的深入介绍

# #

Iterator

- Mozilla Developer Network, Iterators and generators
- Mozilla Developer Network, The Iterator protocol
- Jason Orendorff, ES6 In Depth: Iterators and the for-of loop: 遍历器与for...of循环的介绍
- Axel Rauschmayer, Iterators and generators in ECMAScript 6: 探讨Iterator和Generator的设计目的
- Axel Rauschmayer, Iterables and iterators in ECMAScript 6: Iterator的详细介绍
- Kyle Simpson, Iterating ES6 Numbers: 在数值对象上部署遍历器
- Mahdi Dibaiee, ES7 Array and Generator comprehensions：ES7的Generator推导

# #

Generator

- Matt Baker, Replacing callbacks with ES6 Generators
- Steven Sanderson, Experiments with Koa and JavaScript Generators
- jmar777, What's the Big Deal with Generators?
- Marc Harter, Generators in Node.js: Common Misconceptions and Three Good Use Cases: 讨论Generator函数的作用
- StackOverflow, ES6 yield : what happens to the arguments of the first call next()?: 第一次使用next方法时不能带有参数
- Kyle Simpson, ES6 Generators: Complete Series: 由浅入深探讨Generator的系列文章，共四篇
- Gajus Kuizinas, The Definitive Guide to the JavaScript Generators: 对Generator的综合介绍
- Jan Krems, Generators Are Like Arrays: 讨论Generator可以被当作数据结构看待
- Harold Cooper, Coroutine Event Loops in Javascript: Generator用于实现状态机
- Ruslan Ismagilov, learn-generators: 编程练习，共6道题
- Steven Sanderson, Experiments with Koa and JavaScript Generators: Generator入门介绍，以Koa框架为例

# #

Promise对象

- Jake Archibald, JavaScript Promises: There and back again
- Tilde, rsvp.js
- Sandeep Panda, An Overview of JavaScript Promises: ES6 Promise入门介绍
- Jafar Husain, Async Generators: 对async与Generator混合使用的一些讨论
- Axel Rauschmayer, ECMAScript 6 promises (2/2): the API: 对ES6 Promise规格和用法的详细介绍
- Jack Franklin, Embracing Promises in JavaScript: catch方法的例子
- Luke Hoban, Async Functions for ECMAScript: Async函数的设计思想，与Promise、Gernerator函数的关系
- Jafar Husain, Asynchronous Generators for ES7: Async函数的深入讨论
- Nolan Lawson, Taming the asynchronous beast with ES7: async函数通俗的实例讲解

# #

Class与模块

- Sebastian Porto, ES6 classes and JavaScript prototypes: ES6 Class的写法与ES5 Prototype的写法对比
- Jack Franklin, An introduction to ES6 classes: ES6 class的入门介绍
- Jack Franklin, JavaScript Modules the ES6 Way: ES6模块入门
- Axel Rauschmayer, ECMAScript 6 modules: the final syntax: ES6模块的介绍，以及与CommonJS规格的详细比较
- Dave Herman, Static module resolution: ES6模块的静态化设计思想
- Axel Rauschmayer, ECMAScript 6: new OOP features besides classes
- Axel Rauschmayer, Classes in ECMAScript 6 (final semantics): Class语法的详细介绍和设计思想分析

# #

工具

- Google, traceur-compiler: Traceur编译器
- Casper Beyer, ECMAScript 6 Features and Tools
- Stoyan Stefanov, Writing ES6 today with jstransform
- ES6 Module Loader, ES6 Module Loader Polyfill: 在浏览器和node.js加载ES6模块的一个库，文档里对ES6模块有详细解释
- Paul Miller, es6-shim: 一个针对老式浏览器，模拟ES6部分功能的垫片库（shim）
- army8735, Javascript Downcast: 国产的ES6到ES5的转码器
- esnext, ES6 Module Transpiler：基于node.js的将ES6模块转为ES5代码的命令行工具
- Sebastian McKenzie, BabelJS: ES6转译器
- SystemJS, SystemJS: 在浏览器中加载AMD、CJS、ES6模块的一个垫片库
- Modernizr, HTML5 Cross Browser Polyfills: ES6垫片库清单
- Facebook, regenerator: 将Generator函数转为ES5的转码器