

ECMAScript 6.0（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。它的目标，是使得JavaScript语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

ECMAScript和JavaScript的关系

一个常见的问题是，ECMAScript和JavaScript到底是什么关系？

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1.0版。

该标准从一开始就是针对JavaScript语言制定的，但是之所以不叫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（另外的ECMAScript方言还有Jscript和ActionScript）。日常场合，这两个词是可以互换的。

ES6与ECMAScript 2015的关系

媒体里面经常可以看到“ECMAScript 2015”这个词，它与ES6是什么关系呢？

2011年，ECMAScript 5.1版发布后，就开始制定6.0版了。因此，“ES6”这个词的原意，就是指JavaScript语言的下一个版本。

但是，因为这个版本引入的语法功能太多，而且制定过程当中，还有很多组织和个人不断提交新功能。事情很快就变得清楚了，不可能在一个版本里面包括所有将要引入的功能。常规的做法是先发布6.0版，过一段时间再发6.1版，然后是6.2版、6.3版等等。

但是，标准的制定者不想这样做。他们想让标准的升级成为常规流程：任何人在任何时候，都可以向标准委员会提交新语法的提案，然后标准委员会每个月开一次会，评估这些提案是否可以接受，需要哪些改进。如果经过多次会议以后，一个提案足够成熟了，就可以正式进入标准了。这就是说，标准的版本升级成为了一个不断滚动的流程，每个月都会有变动。

标准委员会最终决定，标准在每年的6月份正式发布一次，作为当年的正式版本。接下来的时间，就在这个版本的基础上做改动，直到下一年的6月份，草案就自然变成了新一年的版本。这样一来，就不需要以前的版本号了，只要用年份标记就可以了。

ES6的第一个版本，就这样在2015年6月发布了，正式名称就是《ECMAScript 2015标准》（简称ES2015）。2016年6月，小幅修订的《ECMAScript 2016标准》（简称ES2016）如期发布，这个版本可以看作是ES6.1版，因为两者的差异非常小（只新增了数组实例的 `includes` 方法和指数运算符），基本上是同一个标准。根据计划，2017年6月将发布ES2017标准。

因此，ES6既是一个历史名词，也是一个泛指，含义是5.1版以后的JavaScript的下一代标准，涵盖了ES2015、ES2016、ES2017等等，而ES2015则是正式名称，特指该年发布的正式版本的语言标准。本书中提到“ES6”的地方，一般是指ES2015标准，但有时也是泛指“下一代JavaScript语言”。

语法提案的批准流程

任何人都可以向TC39标准委员会提案。一种新的语法从提案到变成正式标准，需要经历五个阶段。每个阶段的变动都需要由TC39委员会批准。

- Stage 0 - Strawman (展示阶段)
- Stage 1 - Proposal (征求意见阶段)
- Stage 2 - Draft (草案阶段)
- Stage 3 - Candidate (候选人阶段)
- Stage 4 - Finished (定案阶段)

一个提案只要能进入Stage 2，就差不多等于肯定会包括在以后的正式标准里面。ECMAScript当前的所有提案，可以在TC39的官方网站[Github.com/tc39/ecma262](https://github.com/tc39/ecma262)查看。

本书的写作目标之一，是跟踪ECMAScript语言的最新进展，介绍5.1版本以后所有的新语法。对于那些明确将要列入标准的新语法，尤其是那些Babel转码器（详见后文）已经支持的功能，也将予以介绍。

ECMAScript的历史

ES6从开始制定到最后发布，整整用了15年。

前面提到，ECMAScript 1.0是1997年发布的，接下来的两年，连续发布了ECMAScript 2.0（1998年6月）和ECMAScript 3.0（1999年12月）。3.0版是一个巨大的成功，在业界得到广泛支持，成为通行标准，奠定了JavaScript语言的基本语法，以后的版本完全继承。直到今天，初学者一开始学习JavaScript，其实就是在学3.0版的语法。

2000年，ECMAScript 4.0开始酝酿。这个版本最后没有通过，但是它的大部分内容被ES6继承了。因此，ES6制定的起点其实是2000年。

为什么ES4没有通过呢？因为这个版本太激进了，对ES3做了彻底升级，导致标准委员会的一些成员不愿意接受。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。

2007年10月，ECMAScript 4.0版草案发布，本来预计次年8月发布正式版本。但是，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，争论过于激烈，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。TC39委员会的总体考虑是，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。

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。从2000年算起，这时已经过去了15年。

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是JavaScript语言的服务器运行环境，对ES6的支持度比浏览器更高。通过Node，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，因为可以自由切换版本。不过，nvm不支持Windows系统，如果你使用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。本节写作时的版本号是 `v0.29.0`。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的最新版。

```
$ nvm install node
```

安装完成后，切换到该版本。

```
$ nvm use node
```

使用下面的命令，可以查看Node所有已经实现的ES6特性。

```
$ node --v8-options | grep harmony
--harmony_typeof
--harmony_scoping
--harmony_modules
--harmony_symbols
--harmony_proxies
--harmony_collections
--harmony_observation
--harmony_generators
--harmony_iteration
--harmony_numeric_literals
--harmony_strings
--harmony_arrays
--harmony_maths
--harmony
```

上面命令的输出结果，会因为版本的不同而有所不同。

我写了一个[ES-Checker](#)模块，用来检查各种运行环境对ES6的支持情况。访问ruanyf.github.io/es-checker，可以看到您的浏览器支持ES6的程度。运行下面的命令，可以查看你正在使用的Node环境对ES6的支持程度。

```
$ npm install -g es-checker
$ es-checker
=====
Passes 24 feature Decisions
Your runtime supports 57% of ECMAScript 6
=====
```

Babel转码器

[Babel](#)是一个广泛使用的ES6转码器，可以将ES6代码转为ES5代码，从而在现有环境执行。这意味着，你可以用ES6的方式编写程序，又不用担心现有环境是否支持。下面是一个例子。

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

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

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

配置文件 `.babelrc`

Babel的配置文件是 `.babelrc`，存放在项目的根目录下。使用Babel的第一步，就是配置这个文件。

该文件用来设置转码规则和插件，基本格式如下。

```
{
  "presets": [],
  "plugins": []
}
```

`presets` 字段设定转码规则，官方提供以下的规则集，你可以根据需要安装。

```
# ES2015转码规则
$ npm install --save-dev babel-preset-es2015

# react转码规则
$ npm install --save-dev babel-preset-react

# ES7不同阶段语法提案的转码规则（共有4个阶段），选装一个
$ npm install --save-dev babel-preset-stage-0
$ npm install --save-dev babel-preset-stage-1
$ npm install --save-dev babel-preset-stage-2
$ npm install --save-dev babel-preset-stage-3
```

然后，将这些规则加入 `.babelrc`。

```
{
  "presets": [
    "es2015",
    "react",
    "stage-2"
  ],
  "plugins": []
}
```

注意，以下所有Babel工具和模块的使用，都必须先写好 `.babelrc`。

命令行转码 `babel-cli`

Babel提供 `babel-cli` 工具，用于命令行转码。

它的安装命令如下。

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

基本用法如下。

```
# 转码结果输出到标准输出
$ babel example.js
```

```
# 转码结果写入一个文件
# --out-file 或 -o 参数指定输出文件
$ babel example.js --out-file compiled.js
# 或者
$ babel example.js -o compiled.js

# 整个目录转码
# --out-dir 或 -d 参数指定输出目录
$ babel src --out-dir lib
# 或者
$ babel src -d lib

# -s 参数生成source map文件
$ babel src -d lib -s
```

上面代码是在全局环境下，进行Babel转码。这意味着，如果项目要运行，全局环境必须有Babel，也就是说项目产生了对环境的依赖。另一方面，这样做也无法支持不同项目使用不同版本的Babel。

一个解决办法是将 `babel-cli` 安装在项目之中。

```
# 安装
$ npm install --save-dev babel-cli
```

然后，改写 `package.json`。

```
{
  // ...
  "devDependencies": {
    "babel-cli": "^6.0.0"
  },
  "scripts": {
    "build": "babel src -d lib"
  }
}
```

转码的时候，就执行下面的命令。

```
$ npm run build
```

babel-node

`babel-cli` 工具自带一个 `babel-node` 命令，提供一个支持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。

它不用单独安装，而是随 `babel-cli` 一起安装。然后，执行 `babel-node` 就进入REPL环境。

```
$ babel-node
> (x => x * 2)(1)
2
```

`babel-node` 命令可以直接运行ES6脚本。将上面的代码放入脚本文件 `es6.js`，然后直接运行。

```
$ babel-node es6.js
2
```

`babel-node` 也可以安装在项目中。

```
$ npm install --save-dev babel-cli
```

然后，改写 `package.json`。

```
{  
  "scripts": {  
    "script-name": "babel-node script.js"  
  }  
}
```

上面代码中，使用 `babel-node` 替代 `node`，这样 `script.js` 本身就不需要做任何转码处理。

babel-register

`babel-register` 模块改写 `require` 命令，为它加上一个钩子。此后，每当使用 `require` 加载 `.js`、`.jsx`、`.es` 和 `.es6` 后缀名的文件，就会先用Babel进行转码。

```
$ npm install --save-dev babel-register
```

使用时，必须首先加载 `babel-register`。

```
require("babel-register");  
require("./index.js");
```

然后，就不需要手动对 `index.js` 转码了。

需要注意的是，`babel-register` 只会对 `require` 命令加载的文件转码，而不会对当前文件转码。另外，由于它是实时转码，所以只适合在开发环境使用。

babel-core

如果某些代码需要调用Babel的API进行转码，就要使用 `babel-core` 模块。

安装命令如下。

```
$ npm install babel-core --save
```

然后，在项目中就可以调用 `babel-core`。

```
var babel = require('babel-core');  
  
// 字符串转码  
babel.transform('code()', options);  
// => { code, map, ast }  
  
// 文件转码（异步）  
babel.transformFile('filename.js', options, function(err, result) {  
  result; // => { code, map, ast }  
});  
  
// 文件转码（同步）  
babel.transformFileSync('filename.js', options);  
// => { code, map, ast }  
  
// Babel AST转码  
babel.transformFromAst(ast, code, options);  
// => { code, map, ast }
```

配置对象 `options`，可以参看官方文档<http://babeljs.io/docs/usage/options/>。

下面是一个例子。

```
var es6Code = 'let x = n => n + 1';
var es5Code = require('babel-core')
  .transform(es6Code, {
    presets: ['es2015']
  })
  .code;
// '"use strict";\n\nvar x = function x(n) {\n  return n + 1;\n};'
```

上面代码中，`transform` 方法的第一个参数是一个字符串，表示需要被转换的ES6代码，第二个参数是转换的配置对象。

babel-polyfill

Babel默认只转换新的JavaScript句法（syntax），而不转换新的API，比如Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如`Object.assign`）都不会转码。

举例来说，ES6在`Array`对象上新增了`Array.from`方法。Babel就不会转码这个方法。如果想让这个方法运行，必须使用`babel-polyfill`，为当前环境提供一个垫片。

安装命令如下。

```
$ npm install --save babel-polyfill
```

然后，在脚本头部，加入如下一行代码。

```
import 'babel-polyfill';
// 或者
require('babel-polyfill');
```

Babel默认不转码的API非常多，详细清单可以查看 `babel-plugin-transform-runtime` 模块的`definitions.js`文件。

浏览器环境

Babel也可以用于浏览器环境。但是，从Babel 6.0开始，不再直接提供浏览器版本，而是要用构建工具构建出来。如果你没有或不想使用构建工具，可以通过安装5.x版本的`babel-core`模块获取。

```
$ npm install babel-core@5
```

运行上面的命令以后，就可以在当前目录的`node_modules/babel-core/`子目录里面，找到`babel`的浏览器版本`browser.js`（未精简）和`browser.min.js`（已精简）。

然后，将下面的代码插入网页。

```
<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"`。

另一种方法是使用`babel-standalone`模块提供的浏览器版本，将其插入网页。

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
standalone/6.4.4/babel.min.js"></script>
<script type="text/babel">
// Your ES6 code
</script>
```

注意，网页中实时将ES6代码转为ES5，对性能会有影响。生产环境需要加载已经转码完成的脚本。

下面是如何将代码打包成浏览器可以使用的脚本，以 `Babel` 配合 `Browserify` 为例。首先，安装 `babelify` 模块。

```
$ npm install --save-dev babelify babel-preset-es2015
```

然后，再用命令行转换ES6脚本。

```
$ browserify script.js -o bundle.js \
-t [ babelify --presets [ es2015 ] ]
```

上面代码将ES6脚本 `script.js`，转为 `bundle.js`，浏览器直接加载后者就可以了。

在 `package.json` 设置下面的代码，就不用每次命令行都输入参数了。

```
{
  "browserify": {
    "transform": [ ["babelify", { "presets": [ "es2015" ] } ] ]
  }
}
```

在线转换

Babel提供一个[REPL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。

与其他工具的配合

许多工具需要Babel进行前置转码，这里举两个例子：ESLint和Mocha。

ESLint用于静态检查代码的语法和风格，安装命令如下。

```
$ npm install --save-dev eslint babel-eslint
```

然后，在项目根目录下，新建一个配置文件 `.eslintrc`，在其中加入 `parser` 字段。

```
{
  "parser": "babel-eslint",
  "rules": {
    ...
  }
}
```

再在 `package.json` 之中，加入相应的 `scripts` 脚本。

```
{
  "name": "my-module",
  "scripts": {
    "lint": "eslint my-files.js"
  },
  "devDependencies": {
    "babel-eslint": "...",
    "eslint": "..."
  }
}
```

Mocha则是一个测试框架，如果需要执行使用ES6语法的测试脚本，可以修改 `package.json` 的 `scripts.test`。

```
"scripts": {
  "test": "mocha --ui qunit --compilers js:babel-core/register"
}
```

上面命令中，`--compilers` 参数指定脚本的转码器，规定后缀名为 `js` 的文件，都需要使用 `babel-core/register` 先转码。

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。

```
<script src="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
<script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js">
</script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></script>
<script type="module">
  import './Greeter.js';
</script>
```

上面代码中，一共有4个 `script` 标签。第一个是加载Traceur的库文件，第二个和第三个是将这个库文件用于浏览器环境，第四个则是加载用户脚本，这个脚本里面可以使用ES6代码。

注意，第四个 `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代码的标志，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。

除了引用外部ES6脚本，也可以直接在网页中放置ES6代码。

```
<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。

如果想对Traceur的行为有精确控制，可以采用下面参数配置的写法。

```
<script>
  // Create the System object
  window.System = new traceur.runtime.BrowserTraceurLoader();
  // Set some experimental options
  var metadata = {
    traceurOptions: {
      experimental: true,
      properTailCalls: true,
      symbols: true,
      arrayComprehension: true,
      asyncFunctions: true,
```

```

        asyncGenerators: exponentiation,
        forOn: true,
        generatorComprehension: true
    }
};

// Load your module
System.import('./myModule.js', {metadata: metadata}).catch(function(ex) {
    console.error('Import failed', ex.stack || ex);
});
</script>

```

上面代码中，首先生成Traceur的全局对象 `window.System`，然后 `System.import` 方法可以用来加载ES6模块。加载的时候，需要传入一个配置对象 `metadata`，该对象的 `traceurOptions` 属性可以配置支持ES6功能。如果设为 `experimental: true`，就表示除了ES6以外，还支持一些实验性的新功能。

在线转换

Traceur也提供一个[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="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
<script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js">
</script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></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
```

命令行下转换生成的文件，就可以直接放到浏览器中运行。

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);
```

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 < 10; i++) {}

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

上面代码中，计数器 `i` 只在 `for` 循环体内有效，在循环体外引用就会报错。

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
```

上面代码中，变量 `i` 是 `var` 声明的，在全局范围内都有效。所以每一次循环，新的 `i` 值都会覆盖旧值，导致最后输出的是最后一轮的 `i` 的值。

如果使用 `let`，声明的变量仅在块级作用域内有效，最后输出的是 `6`。

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

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

不存在变量提升

`let` 不像 `var` 那样会发生“变量提升”现象。所以，变量一定要在声明后使用，否则报错。

```
// var 的情况
console.log(foo); // 输出undefined
var foo = 2;

// let 的情况
console.log(bar); // 报错ReferenceError
let bar = 2;
```

上面代码中，变量 `foo` 用 `var` 命令声明，会发生变量提升，即脚本开始运行时，变量 `foo` 已经存在了，但是没有值，所以会输出 `undefined`。变量 `bar` 用 `let` 命令声明，不会发生变量提升。这表示在声明它之前，变量 `bar` 是不存在的，这时如果用到它，就会抛出一个错误。

暂时性死区

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变量，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”。

“暂时性死区”也意味着 `typeof` 不再是一个百分之百安全的操作。

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

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

作为比较，如果一个变量根本没有被声明，使用 `typeof` 反而不会报错。

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

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

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（某些实现可能不报错），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”。如果 `y` 的默认值是 `x`，就不会报错，因为此时 `x` 已经声明了。

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

ES6规定暂时性死区和 `let`、`const` 语句不出现变量提升，主要是为了减少运行时错误，防止在变量声明前就使用这个变量，从而导致意料之外的行为。这样的错误在ES5是很常见的，现在有了这种规定，避免此类错误就很容易了。

总之，暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

不允许重复声明

`let`不允许在相同作用域内，重复声明同一个变量。

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

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

因此，不能在函数内部重新声明参数。

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

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

块级作用域

为什么需要块级作用域？

ES5只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。

第一种场景，内层变量可能会覆盖外层变量。

```
var tmp = new Date();

function f() {
  console.log(tmp);
  if (false) {
    var tmp = "hello world";
  }
}

f(); // undefined
```

上面代码中，函数`f`执行后，输出结果为 `undefined`，原因在于变量提升，导致内层的`tmp`变量覆盖了外层的`tmp`变量。

第二种场景，用来计数的循环变量泄露为全局变量。

```
var s = 'hello';

for (var i = 0; i < s.length; i++) {
  console.log(s[i]);
}
```

```
console.log(i); // 5
```

上面代码中，变量*i*只用来控制循环，但是循环结束后，它并没有消失，泄露成了全局变量。

ES6的块级作用域

`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。

ES6允许块级作用域的任意嵌套。

```
{{{{{{let insane = 'Hello World'}}}}}};
```

上面代码使用了一个五层的块级作用域。外层作用域无法读取内层作用域的变量。

```
{{{{
  {let insane = 'Hello World'}
  console.log(insane); // 报错
}}}};
```

内层作用域可以定义外层作用域的同名变量。

```
{{{{{
  let insane = 'Hello World';
  {let insane = 'Hello World'}
}}}};
```

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

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

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

块级作用域与函数声明

函数能不能在块级作用域之中声明，是一个相当令人混淆的问题。

ES5规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。

```
// 情况一
```

```
if (true) {
  function f() {}
}

// 情况二
try {
  function f() {}
} catch(e) {
}
```

上面代码的两种函数声明，根据ES5的规定都是非法的。

但是，浏览器没有遵守这个规定，为了兼容以前的旧代码，还是支持在块级作用域之中声明函数，因此上面两种情况实际都能运行，不会报错。不过，“严格模式”下还是会报错。

```
// ES5严格模式
'use strict';
if (true) {
  function f() {}
}
// 报错
```

ES6 引入了块级作用域，明确允许在块级作用域之中声明函数。

```
// ES6严格模式
'use strict';
if (true) {
  function f() {}
}
// 不报错
```

ES6 规定，块级作用域之中，函数声明语句的行为类似于 `let`，在块级作用域之外不可引用。

```
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!`”，因为在 `if` 内声明的函数 `f` 会被提升到函数头部，实际运行的代码如下。

```
// ES5版本
function f() { console.log('I am outside!'); }
(function () {
  function f() { console.log('I am inside!'); }
  if (false) {
  }
  f();
})();
```

ES6 的运行结果就完全不一样了，会得到“`I am outside!`”。因为块级作用域内声明的函数类似于 `let`，对作用域之外没有影响，实际运行的代码如下。

```
// ES6版本
function f() { console.log('I am outside!'); }
(function () {
```

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

很显然，这种行为差异会对老代码产生很大影响。为了减轻因此产生的不兼容问题，ES6在[附录B](#)里面规定，浏览器的实现可以不遵守上面的规定，有自己的[行为方式](#)。

- 允许在块级作用域内声明函数。
- 函数声明类似于 `var`，即会提升到全局作用域或函数作用域的头部。
- 同时，函数声明还会提升到所在的块级作用域的头部。

注意，上面三条规则只对ES6的浏览器实现有效，其他环境的实现不用遵守，还是将块级作用域的函数声明当作 `let` 处理。

前面那段代码，在 Chrome 环境下运行会报错。

```
// ES6的浏览器环境  
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();  
}());  
// Uncaught TypeError: f is not a function
```

上面的代码报错，是因为实际运行的是下面的代码。

```
// ES6的浏览器环境  
function f() { console.log('I am outside!'); }  
(function () {  
  var f = undefined;  
  if (false) {  
    function f() { console.log('I am inside!'); }  
  }  
  
  f();  
}());  
// Uncaught TypeError: f is not a function
```

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

```
// 函数声明语句  
{  
  let a = 'secret';  
  function f() {  
    return a;  
  }  
}  
  
// 函数表达式  
{  
  let a = 'secret';  
  let f = function () {  
    return a;  
  };  
}
```

另外，还有一个需要注意的地方。ES6的块级作用域允许声明函数的规则，只在使用大括号的情况下成立，如果没有使

用大括号，就会报错。

```
// 不报错
'use strict';
if (true) {
  function f() {}
}

// 报错
'use strict';
if (true)
  function f() {}
```

do 表达式

本质上，块级作用域是一个语句，将多个操作封装在一起，没有返回值。

```
{
  let t = f();
  t = t * t + 1;
}
```

上面代码中，块级作用域将两个语句封装在一起。但是，在块级作用域以外，没有办法得到 `t` 的值，因为块级作用域不返回值，除非 `t` 是全局变量。

现在有一个[提案](#)，使得块级作用域可以变为表达式，也就是说可以返回值，办法就是在块级作用域之前加上 `do`，使它变为 `do` 表达式。

```
let x = do {
  let t = f();
  t * t + 1;
};
```

上面代码中，变量 `x` 会得到整个块级作用域的返回值。

const 命令

`const` 声明一个只读的常量。一旦声明，常量的值就不能改变。

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

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

上面代码表明改变常量的值会报错。

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

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

上面代码表示，对于 `const` 来说，只声明不赋值，就会报错。

`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;
}
```

```
MAX // Uncaught ReferenceError: MAX is not defined
```

`const` 命令声明的常量也是不提升，同样存在暂时性死区，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;
```

对于复合类型的变量，变量名不指向数据，而是指向数据所在的地址。`const` 命令只是保证变量名指向的地址不变，并不保证该地址的数据不变，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 = {} // TypeError: "foo" is read-only
```

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({});

// 常规模式时，下面一行不起作用;
// 严格模式时，该行会报错
foo.prop = 123;
```

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，严格模式时还会报错。

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] );
    }
  });
}
```

```
    }  
});  
};
```

ES5只有两种声明变量的方法：`var` 命令和 `function` 命令。ES6除了添加 `let` 和 `const` 命令，后面章节还会提到，另外两种声明变量的方法：`import` 命令和 `class` 命令。所以，ES6一共有6种声明变量的方法。

顶层对象的属性

顶层对象，在浏览器环境指的是 `window` 对象，在Node指的是 `global` 对象。ES5之中，顶层对象的属性与全局变量是等价的。

```
window.a = 1;  
a // 1  
  
a = 2;  
window.a // 2
```

上面代码中，顶层对象的属性赋值与全局变量的赋值，是同一件事。

顶层对象的属性与全局变量挂钩，被认为是JavaScript语言最大的设计败笔之一。这样的设计带来了几个很大的问题，首先是没法在编译时就报出变量未声明的错误，只有运行时才能知道（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；其次，程序员很容易不知不觉地就创建了全局变量（比如打字出错）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。另一方面，`window` 对象有实体含义，指的是浏览器的窗口对象，顶层对象是一个有实体含义的对象，也是不合适的。

ES6为了改变这一点，一方面规定，为了保持兼容性，`var` 命令和 `function` 命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，`let` 命令、`const` 命令、`class` 命令声明的全局变量，不属于顶层对象的属性。也就是说，从ES6开始，全局变量将逐步与顶层对象的属性脱钩。

```
var a = 1;  
// 如果在Node的REPL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，所以它是顶层对象的属性；全局变量 `b` 由 `let` 命令声明，所以它不是顶层对象的属性，返回 `undefined`。

global 对象

ES5的顶层对象，本身也是一个问题，因为它在各种实现里面是不统一的。

- 浏览器里面，顶层对象是 `window`，但 Node 和 Web Worker 没有 `window`。
- 浏览器和 Web Worker 里面，`self` 也指向顶层对象，但是Node没有 `self`。
- Node 里面，顶层对象是 `global`，但其他环境都不支持。

同一段代码为了能够在各种环境，都能取到顶层对象，现在一般是使用 `this` 变量，但是有局限性。

- 全局环境中，`this` 会返回顶层对象。但是，Node模块和ES6模块中，`this` 返回的是当前模块。
- 函数里面的 `this`，如果函数不是作为对象的方法运行，而是单纯作为函数运行，`this` 会指向顶层对象。但是，严格模式下，这时 `this` 会返回 `undefined`。
- 不管是严格模式，还是普通模式，`new Function('return this')()`，总是会返回全局对象。但是，如果浏览器用了CSP (Content Security Policy, 内容安全政策)，那么 `eval`、`new Function` 这些方法都可能无法使用。

综上所述，很难找到一种方法，可以在所有情况下，都取到顶层对象。下面是两种勉强可以使用的方法。

```
// 方法一
(typeof window !== 'undefined'
? window
: (typeof process === 'object' &&
  typeof require === 'function' &&
  typeof global === 'object')
? global
: this);

// 方法二
var getGlobal = function () {
  if (typeof self !== 'undefined') { return self; }
  if (typeof window !== 'undefined') { return window; }
  if (typeof global !== 'undefined') { return global; }
  throw new Error('unable to locate global object');
};
```

现在有一个提案，在语言标准的层面，引入 `global` 作为顶层对象。也就是说，在所有环境下，`global` 都是存在的，都可以从它拿到顶层对象。

垫片库 `system.global` 模拟了这个提案，可以在所有环境拿到 `global`。

```
// CommonJS的写法
require('system.global/shim')();

// ES6模块的写法
import shim from 'system.global/shim'; shim();
```

上面代码可以保证各种环境里面，`global` 对象都是存在的。

```
// CommonJS的写法
var global = require('system.global')();

// ES6模块的写法
import getGlobal from 'system.global';
const global = getGlobal();
```

上面代码将顶层对象放入变量 `global`。

变量的解构赋值

数组的解构赋值

基本用法

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]

let [x, y, ...z] = ['a'];
x // "a"
y // undefined
z // []
```

如果解构不成功，变量的值就等于 `undefined`。

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

以上两种情况都属于解构不成功，`foo` 的值都会等于 `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
```

上面两个例子，都属于不完全解构，但是可以成功。

如果等号的右边不是数组（或者严格地说，不是可遍历的结构，参见《Iterator》一章），那么将会报错。

```
// 报错
let [foo] = 1;
let [foo] = false;
let [foo] = NaN;
let [foo] = undefined;
let [foo] = null;
let [foo] = {};
```

上面的表达式都会报错，因为等号右边的值，要么转为对象以后不具备Iterator接口（前五个表达式），要么本身就不具备Iterator接口（最后一个表达式）。

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。

```
let [x, y, z] = new Set(["a", "b", "c"]);
x // "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。

默认值

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`。

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

```
function f() {
  console.log('aaa');
}

let [x = f()] = [1];
```

上面代码中，因为`x`能取到值，所以函数`f`根本不会执行。上面的代码其实等价于下面的代码。

```
let x;
if ([1][0] === undefined) {
  x = f();
} else {
  x = [1][0];
```

```
}
```

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。

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

上面最后一个表达式之所以会报错，是因为 `x` 用到默认值 `y` 时，`y` 还没有声明。

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'
```

这实际上说明，对象的解构赋值是下面形式的简写（参见《对象的扩展》一章）。

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

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

```
var { foo: baz } = { foo: "aaa", bar: "bbb" };
baz // "aaa"
foo // error: foo is not defined
```

上面代码中，`foo` 是匹配的模式，`baz` 才是变量。真正被赋值的是变量 `baz`，而不是模式 `foo`。

注意，采用这种写法时，变量的声明和赋值是一体的。对于 `let` 和 `const` 来说，变量不能重新声明，所以一旦赋值的变量以前声明过，就会报错。

```
let foo;
```

```
let {foo} = {foo: 1}; // SyntaxError: Duplicate declaration "foo"  
let baz;  
let {bar: baz} = {bar: 1}; // SyntaxError: Duplicate declaration "baz"
```

上面代码中，解构赋值的变量都会重新声明，所以报错了。不过，因为 `var` 命令允许重新声明，所以这个错误只会在使用 `let` 和 `const` 命令时出现。如果没有第二个 `let` 命令，上面的代码就不会报错。

```
let foo;  
({foo} = {foo: 1}); // 成功  
  
let baz;  
({bar: baz} = {bar: 1}); // 成功
```

上面代码中，`let` 命令下面一行的圆括号是必须的，否则会报错。因为解析器会将起首的大括号，理解成一个代码块，而不是赋值语句。

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"
```

注意，这时 `p` 是模式，不是变量，因此不会被赋值。

```
var node = {  
  loc: {  
    start: {  
      line: 1,  
      column: 5  
    }  
  }  
};  
  
var { loc: { start: { line } } } = node;  
line // 1  
loc // error: loc is undefined  
start // error: start is undefined
```

上面代码中，只有 `line` 是变量，`loc` 和 `start` 都是模式，不会被赋值。

下面是嵌套赋值的例子。

```
let obj = {};  
let arr = [];  
  
({ foo: obj.prop, bar: arr[0] } = { foo: 123, bar: true });  
  
obj // {prop:123}  
arr // [true]
```

对象的解构也可以指定默认值。

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

```
x // 3  
  
var {x, y = 5} = {x: 1};  
x // 1  
y // 5  
  
var {x:y = 3} = {};  
y // 3  
  
var {x:y = 3} = {x: 5};  
y // 5  
  
var { message: msg = 'Something went wrong' } = {};  
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。

如果解构失败，变量的值等于 `undefined`。

```
var {foo} = {bar: 'baz'};  
foo // undefined
```

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错。

```
// 报错  
var {foo: {bar}} = {baz: 'baz'};
```

上面代码中，等号左边对象的 `foo` 属性，对应一个子对象。该子对象的 `bar` 属性，解构时会报错。原因很简单，因为 `foo` 这时等于 `undefined`，再取子属性就会报错，请看下面的代码。

```
var _tmp = {baz: 'baz'};  
_tmp.foo.bar // 报错
```

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。

解构赋值允许，等号左边的模式之中，不放置任何变量名。因此，可以写出非常古怪的赋值表达式。

```
({} = [true, false]);  
({} = 'abc');
```

```
({}) = [];
```

上面的表达式虽然毫无意义，但是语法是合法的，可以执行。

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。

由于数组本质是特殊的对象，因此可以对数组进行对象属性的解构。

```
var arr = [1, 2, 3];
var {0 : first, [arr.length - 1] : last} = arr;
first // 1
last // 3
```

上面代码对数组进行对象解构。数组 `arr` 的 `0` 键对应的值是 `1`，`[arr.length - 1]` 就是 `2` 键，对应的值是 `3`。方括号这种写法，属于“属性名表达式”，参见《对象的扩展》一章。

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
```

数值和布尔值的解构赋值

解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。

```
let {toString: s} = 123;
s === Number.prototype.toString // true

let {toString: s} = true;
s === Boolean.prototype.toString // true
```

上面代码中，数值和布尔值的包装对象都有 `toString` 属性，因此变量 `s` 都能取到值。

解构赋值的规则是，只要等号右边的值不是对象，就先将其转为对象。由于 `undefined` 和 `null` 无法转为对象，所以对它们进行解构赋值，都会报错。

```
let { prop: x } = undefined; // TypeError
let { prop: y } = null; // TypeError
```

函数参数的解构赋值

函数的参数也可以使用解构赋值。

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

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

上面代码中，函数 `add` 的参数表面上是一个数组，但在传入参数的那一刻，数组参数就被解构成变量 `x` 和 `y`。对于函数内部的代码来说，它们能感受到的参数就是 `x` 和 `y`。

下面是另一个例子。

```
[[1, 2], [3, 4]].map(([a, b]) => a + b);
// [3, 7]
```

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，下面的写法会得到不一样的结果。

```
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。

`undefined` 就会触发函数参数的默认值。

```
[1, undefined, 3].map((x = 'yes') => x);
// [1, 'yes', 3]
```

圆括号问题

解构赋值虽然很方便，但是解析起来不容易。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，不能带有圆括号。

```
// 全部报错
var [(a)] = [1];

var {x: (c)} = {};
var ({x: c}) = {};
var {(x: c)} = {};
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) 赋值语句中，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圆括号之中，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({z: 3, y: 2, x: 1});
```

(4) 提取JSON数据

解构赋值对提取JSON对象中的数据，尤其有用。

```
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,
  // ...
}) {
```

```
// ... do stuff  
};
```

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

(6) 遍历Map结构

任何部署了Iterator接口的对象，都可以用 `for...of` 循环遍历。Map结构原生支持Iterator接口，配合变量的解构赋值，获取键名和键值就非常方便。

```
var map = new Map();  
map.set('first', 'hello');  
map.set('second', 'world');  
  
for (let [key, value] of map) {  
  console.log(key + " is " + value);  
}  
// first is hello  
// second is world
```

如果只想获取键名，或者只想获取键值，可以写成下面这样。

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

(7) 输入模块的指定方法

加载模块时，往往需要指定输入那些方法。解构赋值使得输入语句非常清晰。

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

字符串的扩展

ES6加强了对Unicode的支持，并且扩展了字符串对象。

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"  
// "𠮷"  
  
"\u20BB7"  
// "߻"
```

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}"  
// "吉"  
  
"\u{41}\u{42}\u{43}"  
// "ABC"  
  
let hello = 123;  
hell\u{6F} // 123  
  
\u{1F680} === '\uD83D\uDE80'  
// true
```

上面代码中，最后一个例子表明，大括号表示法与四字节的UTF-16编码是等价的。

有了这种表示法之后，JavaScript共有6种方法可以表示一个字符。

```
'\z' === 'z' // true  
\172' === 'z' // true  
\x7A' === 'z' // true  
\u007A' === 'z' // true  
\u{7A}' === 'z' // true
```

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 = "𠮷";  
  
s.length // 2  
s.charCodeAt(0) // 55362  
s.charCodeAt(1) // 57271
```

上面代码中，汉字“𠮷”（注意，这个字不是“吉祥”的“吉”）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` 方法无法读取整个字符，`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 = '𠮷a';  
  
s.codePointAt(0) // 134071  
s.codePointAt(1) // 57271  
  
s.codePointAt(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“𠮷a”视为三个字符，`codePointAt`方法在第一个字符上，正确地识别了“𠮷”，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“𠮷”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32位的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` 方法返回的是码点的十进制值，如果想要十六进制的值，可以使用 `toString` 方法转换一下。

```
var s = '𠮷a';
s.codePointAt(0).toString(16) // "20bb7"
s.codePointAt(2).toString(16) // "61"
```

你可能注意到了，`codePointAt` 方法的参数，仍然是不正确的。比如，上面代码中，字符 `a` 在字符串 `s` 的正确位置序号应该是1，但是必须向 `codePointAt` 方法传入2。解决这个问题的一个办法是使用 `for...of` 循环，因为它会正确识别32位的UTF-16字符。

```
var s = '𠮷a';
for (let ch of s) {
  console.log(ch.codePointAt(0).toString(16));
}
// 20bb7
// 61
```

`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("𠮷") // 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32位的UTF-16字符（Unicode编号大于 `0xFFFF`）。

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

上面代码中，`String.fromCharCode` 不能识别大于 `0xFFFF` 的码点，所以 `0x20BB7` 就发生了溢出，最高位 `2` 被舍弃了，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)
// "𠮷"
String.fromCodePoint(0x78, 0x1f680, 0x79) === '𠮷'
// true
```

上面代码中，如果 `String.fromCodePoint` 方法有多个参数，则它们会被合并成一个字符串返回。

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。

字符串的遍历器接口

ES6为字符串添加了遍历器接口（详见《Iterator》一章），使得字符串可以被 `for...of` 循环遍历。

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

除了遍历字符串，这个遍历器最大的优点是可以识别大于 `0xFFFF` 的码点，传统的 `for` 循环无法识别这样的码点。

```
var text = String.fromCodePoint(0x20BB7);

for (let i = 0; i < text.length; i++) {
  console.log(text[i]);
}

for (let i of text) {
  console.log(i);
}

// "𠮷"
```

上面代码中，字符串 `text` 只有一个字符，但是 `for` 循环会认为它包含两个字符（都不可打印），而 `for...of` 循环会正确识别出这一个字符。

at()

ES5对字符串对象提供 `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。

```
'abc'.charAt(0) // "a"
'𠮷'.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。

目前，有一个提案，提出字符串实例的 `at` 方法，可以识别Unicode编号大于 `0xFFFF` 的字符，返回正确的字符。

```
'abc'.at(0) // "a"
'𠮷'.at(0) // "𠮷"
```

这个方法可以通过[垫片库](#)实现。

normalize()

许多欧洲语言有语调符号和重音符号。为了表示它们，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 `ő` (`\u01D1`)。另一种是提供合成字符（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 `ó` (`\u004F`) 和 `ő` (`\u030C`) 合成 `ő` (`\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提供字符串实例的 `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` 方法可以接受一个参数来指定 `normalize` 的方式，参数的四个可选值如下。

- `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”。（这只是用来举例，`normalize`方法不能识别中文。）
- **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`方法返回一个新字符串，表示将原字符串重复`n`次。

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

参数如果是小数，会被取整。

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

如果`repeat`的参数是负数或者`Infinity`，会报错。

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

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

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

参数 `NaN` 等同于0。

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

如果 `repeat` 的参数是字符串，则会先转换成数字。

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

padStart(), padEnd()

ES7推出了字符串补全长度的功能。如果某个字符串不够指定长度，会在头部或尾部补全。`padStart` 用于头部补全，`padEnd` 用于尾部补全。

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

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

上面代码中，`padStart` 和 `padEnd` 一共接受两个参数，第一个参数用来指定字符串的最小长度，第二个参数是用来补全的字符串。

如果原字符串的长度，等于或大于指定的最小长度，则返回原字符串。

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

如果用来补全的字符串与原字符串，两者的长度之和超过了指定的最小长度，则会截去超出位数的补全字符串。

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

如果省略第二个参数，则会用空格补全长度。

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

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

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

另一个用途是提示字符串格式。

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

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

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}?`
```

上面代码中的模板字符串，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。

```
$('#list').html(`  
  <ul>  
    <li>first</li>  
    <li>second</li>  
  </ul>  
`);
```

上面代码中，所有模板字符串的空格和换行，都是被保留的，比如 `` 标签前面会有一个换行。如果你不想要这个换行，可以使用 `trim` 方法消除它。

```
$('#list').html(`  
  <ul>  
    <li>first</li>  
    <li>second</li>  
  </ul>  
`.  
  .trim());
```

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;

`${x} + ${y} = ${x + y}`
// "1 + 2 = 3"

`${x} + ${y * 2} = ${x + y * 2}`
// "1 + 4 = 5"

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

模板字符串之中还能调用函数。

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

`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。比如，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
```

由于模板字符串的大括号内部，就是执行JavaScript代码，因此如果大括号内部是一个字符串，将会原样输出。

```
`Hello ${'World'}`
// "Hello World"
```

模板字符串甚至还能嵌套。

```
const tmpl = addrs => `


|                |
|----------------|
| \${addr.first} |
| \${addr.last}  |


```

```
`).join('')  
  </table>  
`;
```

上面代码中，模板字符串的变量之中，又嵌入了另一个模板字符串，使用方法如下。

```
const data = [  
  { first: '<Jane>', last: 'Bond' },  
  { first: 'Lars', last: '<Croft>' },  
];  
  
console.log(template(data));  
// <table>  
//  
//   <tr><td><Jane></td></tr>  
//   <tr><td>Bond</td></tr>  
//  
//   <tr><td>Lars</td></tr>  
//   <tr><td><Croft></td></tr>  
//  
// </table>
```

如果需要引用模板字符串本身，在需要时执行，可以像下面这样写。

```
// 写法一  
let str = 'return ' + `Hello ${name}!`;  
let func = new Function('name', str);  
func('Jack') // "Hello Jack!"  
  
// 写法二  
let str = '(name) => `Hello ${name}!`';  
let func = eval.call(null, str);  
func('Jack') // "Hello Jack!"
```

实例：模板编译

下面，我们来看一个通过模板字符串，生成正式模板的实例。

```
var template = `  
<ul>  
  <% for(var i=0; i < data.supplies.length; i++) { %>  
    <li><%= data.supplies[i] %></li>  
  <% } %>  
</ul>  
`;
```

上面代码在模板字符串之中，放置了一个常规模板。该模板使用 `<%...%>` 放置JavaScript代码，使用 `<%= ... %>` 输出JavaScript表达式。

怎么编译这个模板字符串呢？

一种思路是将其转换为JavaScript表达式字符串。

```
echo('<ul>');  
for(var i=0; i < data.supplies.length; i++) {  
  echo('<li>');  
  echo(data.supplies[i]);  
  echo('</li>');  
};  
echo('</ul>');
```

这个转换使用正则表达式就行了。

```
var evalExpr = /<%=(.+?)%>/g;
var expr = /<%([\s\S]+?)%>/g;

template = template
  .replace(evalExpr, '`'); \n  echo( $1 ); \n  echo(`')
  .replace(expr, '`'); \n $1 \n  echo(`');

template = 'echo(`' + template + `');';
```

然后，将 `template` 封装在一个函数里面返回，就可以了。

```
var script =
`(function parse(data){
  var output = "";

  function echo(html){
    output += html;
  }

  ${ template }

  return output;
})`;

return script;
```

将上面的内容拼装成一个模板编译函数 `compile`。

```
function compile(template){
  var evalExpr = /<%=(.+?)%>/g;
  var expr = /<%([\s\S]+?)%>/g;

  template = template
    .replace(evalExpr, '`'); \n  echo( $1 ); \n  echo(`')
    .replace(expr, '`'); \n $1 \n  echo(``);

  template = 'echo(`' + template + `');';

  var script =
`(function parse(data){
  var output = "";

  function echo(html){
    output += html;
  }

  ${ template }

  return output;
})`;

  return script;
}
```

`compile` 函数的用法如下。

```
var parse = eval(compile(template));
div.innerHTML = parse({ supplies: [ "broom", "mop", "cleaner" ] });
//  <ul>
//    <li>broom</li>
//    <li>mop</li>
```

```
//     <li>cleaner</li>
//   </ul>
```

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）。

```
alert`123`
// 等同于
alert(123)
```

标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数。

但是，如果模板字符串里面有变量，就不是简单的调用了，而是会将模板字符串先处理成多个参数，再调用函数。

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

tag`Hello ${ a + b } world ${ a * b }`;
// 等同于
tag(['Hello ', ' world ', ''], 15, 50);
```

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(s[2]);
  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, "&")
        .replace(/</g, "<")
        .replace(/>/g, ">");

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

```

上面代码中，`sender` 变量往往是用户提供的，经过 `SaferHTML` 函数处理，里面的特殊字符都会被转义。

```

var sender = '<script>alert("abc")</script>'; // 恶意代码
var message = SaferHTML`<p>${sender} has sent you a message.</p>`;

message
// <p>&lt;script&gt;alert("abc")&lt;/script&gt; has sent you a message.</p>

```

标签模板的另一个应用，就是多语言转换（国际化处理）。

```

i18n`Welcome to ${siteName}, you are visitor number ${visitorNumber}!`
// "欢迎访问xxx, 您是第xxxx位访问者!"

```

模板字符串本身并不能取代Mustache之类的模板库，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。

```

jsx` 
<div>
  <input
    ref='input'
    onChange='${this.handleChange}'
    defaultValue='${this.state.value}' />
  ${this.state.value}
</div>
`;

```

上面的代码通过 `jsx` 函数，将一个DOM字符串转为React对象。你可以在Github找到 `jsx` 函数的具体实现。

下面则是一个假想的例子，通过 `java` 函数，在JavaScript代码之中运行Java代码。

```

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。

```
console.log`123`  
// ["123", raw: Array[1]]
```

上面代码中，`console.log` 接受的参数，实际上是一个数组。该数组有一个 `raw` 属性，保存的是转义后的原字符串。

请看下面的例子。

```
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 `\n` 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!'
```

如果原字符串的斜杠已经转义，那么 `String.raw` 不会做任何处理。

```
String.raw`Hi\\n`  
// "Hi\\n"
```

`String.raw` 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);
```

模板字符串的限制

前面提到标签模板里面，可以内嵌其他语言。但是，模板字符串默认会将字符串转义，因此导致了无法嵌入其他语言。

举例来说，在标签模板里面可以嵌入Latex语言。

```
function latex(strings) {  
    // ...  
}  
  
let document = latex`  
\newcommand{\fun}{\textbf{Fun!}} // 正常工作  
\newcommand{\unicode}{\textbf{Unicode!}} // 报错  
\newcommand{\xerxes}{\textbf{King!}} // 报错  
  
Breve over the h goes \u{h}ere // 报错  
`
```

上面代码中，变量 `document` 内嵌的模板字符串，对于Latex语言来说完全是合法的，但是JavaScript引擎会报错。原因就在于字符串的转义。

模板字符串会将 `\u00FF` 和 `\u{42}` 当作Unicode字符进行转义，所以 `\unicode` 解析时报错；而 `\x56` 会被当作十六进制字符串转义，所以 `\xerxes` 会报错。

为了解决这个问题，现在有一个[提案](#)，放松对标签模板里面的字符串转义的限制。如果遇到不合法的字符串转义，就返回 `undefined`，而不是报错，并且从 `raw` 属性上面可以得到原始字符串。

```
function tag(strs) {  
    strs[0] === undefined  
    strs.raw[0] === "\u{unicode} and \u{55}";  
}  
tag`\u{unicode} and \u{55}`
```

上面代码中，模板字符串原本是应该报错的，但是由于放松了对字符串转义的限制，所以不报错了，JavaScript引擎将第一个字符设置为 `undefined`，但是 `raw` 属性依然可以得到原始字符串，因此 `tag` 函数还是可以对原字符串进行处理。

注意，这种对字符串转义的放松，只在标签模板解析字符串时生效，不是标签模板的场合，依然会报错。

```
let bad = `bad escape sequence: \unicode`; // 报错
```

正则的扩展

RegExp构造函数

在ES5中，`RegExp`构造函数的参数有两种情况。

第一种情况是，参数是字符串，这时第二个参数表示正则表达式的修饰符（flag）。

```
var regex = new RegExp('xyz', 'i');  
// 等价于  
var regex = /xyz/i;
```

第二种情况是，参数是一个正则表示式，这时会返回一个原有正则表达式的拷贝。

```
var regex = new RegExp(/xyz/i);
// 等价于
var regex = /xyz/i;
```

但是，ES5不允许此时使用第二个参数，添加修饰符，否则会报错。

```
var regex = new RegExp(/xyz/, 'i');
// Uncaught TypeError: Cannot supply flags when constructing one RegExp from another
```

ES6改变了这种行为。如果RegExp构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。

```
new RegExp(/abc/ig, 'i').flags
// "i"
```

上面代码中，原有正则对象的修饰符是 `ig`，它会被第二个参数 `i` 覆盖。

字符串的正则方法

字符串对象共有4个方法，可以使用正则表达式：`match()`、`replace()`、`search()` 和 `split()`。

ES6将这4个方法，在语言内部全部调用RegExp的实例方法，从而做到所有与正则相关的方法，全都定义在RegExp对象上。

- `String.prototype.match` 调用 `RegExp.prototype[Symbol.match]`
- `String.prototype.replace` 调用 `RegExp.prototype[Symbol.replace]`
- `String.prototype.search` 调用 `RegExp.prototype[Symbol.search]`
- `String.prototype.split` 调用 `RegExp.prototype[Symbol.split]`

u修饰符

ES6对正则表达式添加了 `u` 修饰符，含义为“Unicode模式”，用来正确处理大于 `\uFFFF` 的Unicode字符。也就是说，会正确处理四个字节的UTF-16编码。

```
/^\uD83D/u.test('uD83D\uDC2A')
// false
/^\uD83D/.test('uD83D\uDC2A')
// true
```

上面代码中，`\uD83D\uDC2A` 是一个四个字节的UTF-16编码，代表一个字符。但是，ES5不支持四个字节的UTF-16编码，会将其识别为两个字符，导致第二行代码结果为 `true`。加了 `u` 修饰符以后，ES6就会识别其为一个字符，所以第一行代码结果为 `false`。

一旦加上 `u` 修饰符号，就会修改下面这些正则表达式的行为。

(1) 点字符

点（`.`）字符在正则表达式中，含义是除了换行符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 = '𠮷';
/^.$/.test(s) // false
/^$/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('𠮷') // 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码点大于 `0xFFFF` 的Unicode字符。

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

另外，只有在使用 `u` 修饰符的情况下，Unicode表达式当中的大括号才会被正确解读，否则会被解读为量词。

```
/^\u{3}$/.test('uuu') // true
```

上面代码中，由于正则表达式没有 `u` 修饰符，所以大括号被解读为量词。加上 `u` 修饰符，就会被解读为Unicode表达式。

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

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 = '𠮷𠮷';  
  
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。

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。

`y` 修饰符的作用与 `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。

使用 `lastIndex` 属性，可以更好地说明 `y` 修饰符。

```
const REGEX = /a/g;

// 指定从2号位置 (y) 开始匹配
REGEX.lastIndex = 2;

// 匹配成功
const match = REGEX.exec('xaya');

// 在3号位置匹配成功
match.index // 3

// 下一次匹配从4号位开始
REGEX.lastIndex // 4

// 4号位开始匹配失败
REGEX.exec('xaya') // null
```

上面代码中，`lastIndex` 属性指定每次搜索的开始位置，`g` 修饰符从这个位置开始向后搜索，直到发现匹配为止。

`y` 修饰符同样遵守 `lastIndex` 属性，但是要求必须在 `lastIndex` 指定的位置发现匹配。

```
const REGEX = /a/y;

// 指定从2号位置开始匹配
REGEX.lastIndex = 2;

// 不是粘连，匹配失败
REGEX.exec('xaya') // null
```

```
// 指定从3号位置开始匹配
REGEX.lastIndex = 3;

// 3号位置是粘连，匹配成功
const match = REGEX.exec('xaxa');
match.index // 3
REGEX.lastIndex // 4
```

进一步说，`y` 修饰符号隐含了头部匹配的标志`^`。

```
/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`^`在全局匹配中都有效。

在`split`方法中使用`y` 修饰符，原字符串必须以分隔符开头。这也意味着，只要匹配成功，数组的第一个成员肯定是空字符串。

```
// 没有找到匹配
'x##'.split(/#/y)
// [ 'x##' ]

// 找到两个匹配
'##x'.split(/#/y)
// [ '', '', 'x' ]
```

后续的分隔符只有紧跟前面的分隔符，才会被识别。

```
'#x#'.split(/#/y)
// [ '', 'x#' ]

'##'.split(/#/y)
// [ '', '' ]
```

下面是字符串对象的`replace`方法的例子。

```
const REGEX = /a/g;
'aaxa'.replace(REGEX, '-') // '--xa'
```

上面代码中，最后一个`a`因为不是出现下一次匹配的头部，所以不会被替换。

单单一个`y` 修饰符对`match`方法，只能返回第一个匹配，必须与`g` 修饰符联用，才能返回所有匹配。

```
'a1a2a3'.match(/a\d/y) // ["a1"]
'a1a2a3'.match(/a\d/gy) // ["a1", "a2", "a3"]
```

`y` 修饰符的一个应用，是从字符串提取token（词元），`y` 修饰符确保了匹配之间不会有漏掉的字符。

```
const TOKEN_Y = /\s*(\+|[0-9]+)\s*/y;
const TOKEN_G = /\s*(\+|[0-9]+)\s*/g;

 tokenize(TOKEN_Y, '3 + 4')
// [ '3', '+', '4' ]
 tokenize(TOKEN_G, '3 + 4')
// [ '3', '+', '4' ]

function tokenize(TOKEN_REGEX, str) {
  let result = [];
  let match;
  while (match = TOKEN_REGEX.exec(str)) {
```

```
        result.push(match[1]);
    }
    return result;
}
```

上面代码中，如果字符串里面没有非法字符，`y` 修饰符与 `g` 修饰符的提取结果是一样的。但是，一旦出现非法字符，两者的行为就不一样了。

```
tokenize(TOKEN_Y, '3x + 4')
// [ '3' ]
tokenize(TOKEN_G, '3x + 4')
// [ '3', '+', '4' ]
```

上面代码中，`g` 修饰符会忽略非法字符，而 `y` 修饰符不会，这样就很容易发现错误。

sticky属性

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
```

flags属性

ES6为正则表达式新增了 `flags` 属性，会返回正则表达式的修饰符。

```
// ES5的source属性
// 返回正则表达式的正文
/abc/ig.source
// "abc"

// ES6的flags属性
// 返回正则表达式的修饰符
/abc/ig.flags
// 'gi'
```

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`对象的静态方法`RegExp.escape()`，放入ES7。2015年7月31日，TC39认为，这个方法有安全风险，又不愿这个方法变得过于复杂，没有同意将其列入ES7，但这不失为一个真实的需求。

```
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\\?"
```

s 修饰符：dotAll 模式

正则表达式中，点（`.`）是一个特殊字符，代表任意的单个字符，但是行终止符（line terminator character）除外。

以下四个字符属于“行终止符”。

- U+000A 换行符（`\n`）
- U+000D 回车符（`\r`）
- U+2028 行分隔符（line separator）
- U+2029 段分隔符（paragraph separator）

```
/foo.bar/.test('foo\nbar')
// false
```

上面代码中，因为 `.` 不匹配 `\n`，所以正则表达式返回 `false`。

但是，很多时候我们希望匹配的是任意单个字符，这时有一种变通的写法。

```
/foo[^]bar/.test('foo\nbar')
// true
```

这种解决方案毕竟不太符合直觉，所以现在有一个提案，引入 `/s` 修饰符，使得 `.` 可以匹配任意单个字符。

```
/foo.bar/s.test('foo\nbar') // true
```

这被称为 `dotAll` 模式，即点（dot）代表一切字符。所以，正则表达式还引入了一个 `dotAll` 属性，返回一个布尔值，表示该正则表达式是否处在 `dotAll` 模式。

```
const re = /foo.bar/s;
// 另一种写法
// const re = new RegExp('foo.bar', 's');

re.test('foo\nbar') // true
re.dotAll // true
re.flags // 's'
```

`/s` 修饰符和多行修饰符 `/m` 不冲突，两者一起使用的情况下，`.` 匹配所有字符，而 `^` 和 `$` 匹配每一行的行首和行尾。

后行断言

JavaScript语言的正则表达式，只支持先行断言（lookahead）和先行否定断言（negative lookahead），不支持后行断言（lookbehind）和后行否定断言（negative lookbehind）。

目前，有一个[提案](#)，在ES7加入后行断言。V8引擎4.9版已经支持，Chrome浏览器49版打开“experimental JavaScript features”开关（地址栏键入 `about:flags`），就可以使用这项功能。

“先行断言”指的是，`x` 只有在 `y` 前面才匹配，必须写成 `/x(?=y)/`。比如，只匹配百分号之前的数字，要写成 `/\d+(?=%)/`。“先行否定断言”指的是，`x` 只有不在 `y` 前面才匹配，必须写成 `/x(?!y)/`。比如，只匹配不在百分号之前的数字，要写成 `/\d+(?!%)/`。

```
/\d+(?=%)/.exec('100% of US presidents have been male') // ["100"]
/\d+(?!%)/.exec('that's all 44 of them') // ["44"]
```

上面两个字符串，如果互换正则表达式，就会匹配失败。另外，还可以看到，“先行断言”括号之中的部分（`(?=%)`），是不计入返回结果的。

“后行断言”正好与“先行断言”相反，`x` 只有在 `y` 后面才匹配，必须写成 `(?<=y)x/`。比如，只匹配美元符号之后的数字，要写成 `(?<=\$)\d+/-`。“后行否定断言”则与“先行否定断言”相反，`x` 只有不在 `y` 后面才匹配，必须写成 `(?<!y)x/-`。比如，只匹配不在美元符号后面的数字，要写成 `(?<!\$)\d+/-`。

```
(?<=\$)\d+/.exec('Benjamin Franklin is on the $100 bill') // ["100"]
(?<!\$)\d+/.exec('it's is worth about €90') // ["90"]
```

上面的例子中，“后行断言”的括号之中的部分（`(?<=\$)`），也是不计入返回结果。

“后行断言”的实现，需要先匹配 `(?<=y)x/` 的 `x`，然后再回到左边，匹配 `y` 的部分。这种“先右后左”的执行顺序，与所有其他正则操作相反，导致了一些不符合预期的行为。

首先，“后行断言”的组匹配，与正常情况下结果是不一样的。

```
(?<=(\d+)(\d+))$/.exec('1053') // [ "", "1", "053"]
^(\d+)(\d+)$/.exec('1053') // ["1053", "105", "3"]
```

上面代码中，需要捕捉两个组匹配。没有“后行断言”时，第一个括号是贪婪模式，第二个括号只能捕获一个字符，所以结果是 `105` 和 `3`。而“后行断言”时，由于执行顺序是从右到左，第二个括号是贪婪模式，第一个括号只能捕获一个字符，所以结果是 `1` 和 `053`。

其次，“后行断言”的反斜杠引用，也与通常的顺序相反，必须放在对应的那个括号之前。

```
(?<=(o)d\1)r/.exec('hodor') // null
(?<=\1d(o))r/.exec('hodor') // ["r", "o"]
```

上面代码中，如果后行断言的反斜杠引用（`\1`）放在括号的后面，就不会得到匹配结果，必须放在前面才可以。

Unicode属性类

目前，有一个[提案](#)，引入了一种新的类的写法 `\p{...}` 和 `\P{...}`，允许正则表达式匹配符合Unicode某种属性的所有字符。

```
const regexGreekSymbol = /\p{Script=Greek}/u;
regexGreekSymbol.test('π') // u
```

上面代码中，`\p{Script=Greek}` 指定匹配一个希腊文字母，所以匹配 `π` 成功。

Unicode属性类要指定属性名和属性值。

```
\p{UnicodePropertyName=UnicodePropertyValue}
```

对于某些属性，可以只写属性名。

```
\p{UnicodePropertyName}
```

\P{...} 是 \p{...} 的反向匹配，即匹配不满足条件的字符。

注意，这两种类只对Unicode有效，所以使用的时候一定要加上 `u` 修饰符。如果不加 `u` 修饰符，正则表达式使用 \p 和 \P 会报错，ECMAScript预留了这两个类。

由于Unicode的各种属性非常多，所以这种新的类的表达能力非常强。

```
const regex = /^\\p{Decimal_Number}+$/u;  
regex.test('') // true
```

上面代码中，属性类指定匹配所有十进制字符，可以看到各种字型的十进制字符都会匹配成功。

\p{Number} 甚至能匹配罗马数字。

```
// 匹配所有数字  
const regex = /^\\p{Number}+$/u;  
regex.test('2311123') // true  
regex.test('313233') // true  
regex.test('IIMMIVVVVIIIVIIIIXxXIXII') // true
```

下面是其他一些例子。

```
// 匹配各种文字的所有字母，等同于Unicode版的\w  
[\\p{Alphabetic}\\p{Mark}\\p{Decimal_Number}\\p{Connector_Punctuation}\\p{Join_Control}]  
  
// 匹配各种文字的所有非字母的字符，等同于Unicode版的\W  
[\\p{Alphabetic}\\p{Mark}\\p{Decimal_Number}\\p{Connector_Punctuation}\\p{Join_Control}]  
  
// 匹配所有的箭头字符  
const regexArrows = /^\\p{Block=Arrows}+$/u;  
regexArrows.test('←↑↓↔↖↗↙↔↖↑↓↔↖↑↓') // true
```

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`（或 `0B`）和 `0o`（或 `0O`）表示。

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

从ES5开始，在严格模式之中，八进制就不再允许使用前缀 `0` 表示，ES6进一步明确，要使用前缀 `0o` 表示。

```
// 非严格模式  
(function(){  
  console.log(0o11 === 011);  
})() // true  
  
// 严格模式  
(function(){  
  'use strict';  
  console.log(0o11 === 011);  
})() // Uncaught SyntaxError: Octal literals are not allowed in strict mode.
```

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

```
Number('0b111') // 7  
Number('0o10') // 8
```

Number.isFinite(), Number.isNaN()

ES6在`Number`对象上，新提供了 `Number.isFinite()` 和 `Number.isNaN()` 两个方法。

`Number.isFinite()` 用来检查一个数值是否为有限的（finite）。

```
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可以通过下面的代码，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  
Number.isNaN(9/NaN) // true  
Number.isNaN('true'/0) // true  
Number.isNaN('true'/'true') // true
```

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.parseInt === parseInt // true
Number.parseFloat === parseFloat // true
```

Number.isInteger()

`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可以通过下面的代码，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);
```

Number.EPSILON

ES6在Number对象上面，新增一个极小的常量 `Number.EPSILON`。

```
Number.EPSILON  
// 2.220446049250313e-16  
Number.EPSILON.toFixed(20)  
// '0.00000000000000022204'
```

引入一个这么小的量的目的，在于为浮点数计算，设置一个误差范围。我们知道浮点数计算是不精确的。

```
0.1 + 0.2  
// 0.3000000000000004  
  
0.1 + 0.2 - 0.3  
// 5.551115123125783e-17  
  
5.551115123125783e-17.toFixed(20)  
// '0.0000000000000005551'
```

但是如果这个误差能够小于 `Number.EPSILON`，我们就可以认为得到了正确结果。

```
5.551115123125783e-17 < Number.EPSILON  
// true
```

因此，`Number.EPSILON` 的实质是一个可以接受的误差范围。

```
function withinErrorMargin (left, right) {  
    return Math.abs(left - right) < Number.EPSILON;  
}  
withinErrorMargin(0.1 + 0.2, 0.3)  
// true  
withinErrorMargin(0.2 + 0.2, 0.3)  
// false
```

上面的代码为浮点数运算，部署了一个误差检查函数。

安全整数和Number.isSafeInteger()

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

```
Math.pow(2, 53) // 9007199254740992  
9007199254740992 // 9007199254740992  
9007199254740993 // 9007199254740992  
  
Math.pow(2, 53) === Math.pow(2, 53) + 1  
// true
```

上面代码中，超出 2 的 53 次方之后，一个数就不精确了。

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

```
// true
Number.MIN_SAFE_INTEGER === -9007199254740991
// true
```

上面代码中，可以看到JavaScript能够精确表示的极限。

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

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

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

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

这个函数的实现很简单，就是跟安全整数的两个边界值比较一下。

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

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

```
Number.isSafeInteger(9007199254740993)
// false
Number.isSafeInteger(990)
// true
Number.isSafeInteger(9007199254740993 - 990)
// true
9007199254740993 - 990
// 返回结果 9007199254740002
// 正确答案应该是 9007199254740003
```

上面代码中，`9007199254740993`不是一个安全整数，但是`Number.isSafeInteger`会返回结果，显示计算结果是安全的。这是因为，这个数超出了精度范围，导致在计算机内部，以`9007199254740992`的形式储存。

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

所以，如果只验证运算结果是否为安全整数，很可能得到错误结果。下面的函数可以同时验证两个运算数和运算结果。

```
function trusty (left, right, result) {
  if (
    Number.isSafeInteger(left) &&
    Number.isSafeInteger(right) &&
    Number.isSafeInteger(result)
  ) {
    return result;
```

```
    }
    throw new RangeError('Operation cannot be trusted!');
}

trusty(9007199254740993, 990, 9007199254740993 - 990)
// RangeError: Operation cannot be trusted!

trusty(1, 2, 3)
// 3
```

Math对象的扩展

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

Math.trunc()

`Math.trunc` 方法用于去除一个数的小数部分，返回整数部分。

```
Math.trunc(4.1) // 4
Math.trunc(4.9) // 4
Math.trunc(-4.1) // -4
Math.trunc(-4.9) // -4
Math.trunc(-0.1234) // -0
```

对于非数值，`Math.trunc` 内部使用 `Number` 方法将其先转为数值。

```
Math.trunc('123.456')
// 123
```

对于空值和无法截取整数的值，返回NaN。

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

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

Math.sign()

`Math.sign` 方法用来判断一个数到底是正数、负数、还是零。

它会返回五种值。

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

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

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

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

Math.cbrt()

`Math.cbrt` 方法用于计算一个数的立方根。

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

对于非数值，`Math.cbrt` 方法内部也是先使用 `Number` 方法将其转为数值。

```
Math.cbrt('8') // 2  
Math.cbrt('hello') // NaN
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

Math.clz32()

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

```
Math.clz32(0) // 32  
Math.clz32(1) // 31  
Math.clz32(1000) // 22  
Math.clz32(0b01000000000000000000000000000000) // 1  
Math.clz32(0b00100000000000000000000000000000) // 2
```

上面代码中，0的二进制形式全为0，所以有32个前导0；1的二进制形式是 `0b1`，只占1位，所以32位之中有31个前导0；1000的二进制形式是 `0b111101000`，一共有10位，所以32位之中有22个前导0。

`clz32` 这个函数名就来自"count leading zero bits in 32-bit binary representations of a number"（计算32位整数的前导0）的缩写。

左移运算符（`<<`）与 `Math.clz32` 方法直接相关。

```
Math.clz32(0) // 32  
Math.clz32(1) // 31  
Math.clz32(1 << 1) // 30  
Math.clz32(1 << 2) // 29  
Math.clz32(1 << 29) // 2
```

对于小数，`Math.clz32` 方法只考虑整数部分。

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

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

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

Math.imul()

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

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

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

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

上面这个乘法算式，返回结果为0。但是由于这两个二进制数的最低位都是1，所以这个结果肯定是不正确的，因为根据二进制乘法，计算结果的二进制最低位应该也是1。这个错误就是因为它们的乘积超过了2的53次方，JavaScript无法保存额外的精度，就把低位的值都变成了0。`Math.imul` 方法可以返回正确的值1。

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

Math.fround()

`Math.fround`方法返回一个数的单精度浮点数形式。

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

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

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

Math.hypot()

`Math.hypot` 方法返回所有参数的平方和的平方根。

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

上面代码中，3的平方加上4的平方，等于5的平方。

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

对数方法

ES6新增了4个对数相关方法。

(1) Math.expm1()

`Math.expm1(x)` 返回 $e^x - 1$ ，即`Math.exp(x) - 1`。

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

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

(2) Math.log1p()

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

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

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

(3) Math.log10()

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

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

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

(4) Math.log2()

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

```
Math.log2(3)      // 1.584962500721156  
Math.log2(2)      // 1  
Math.log2(1)      // 0  
Math.log2(0)      // -Infinity  
Math.log2(-2)     // NaN  
Math.log2(1024)   // 10  
Math.log2(1 << 29) // 29
```

对于没有部署这个方法的环境，可以用下面的代码模拟。

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

三角函数方法

ES6新增了6个三角函数方法。

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

指数运算符

ES7新增了一个指数运算符 (`**`)，目前Babel转码器已经支持。

```
2 ** 2 // 4  
2 ** 3 // 8
```

指数运算符可以与等号结合，形成一个新的赋值运算符 (`**=`)。

```
let a = 2;  
a **= 2;  
// 等同于 a = a * a;  
  
let b = 3;  
b **= 3;  
// 等同于 b = b * b * b;
```

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）。

下面是一个类似数组的对象，`Array.from` 将它转为真正的数组。

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

// ES5的写法
var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']

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

实际应用中，常见的类似数组的对象是DOM操作返回的NodeList集合，以及函数内部的 `arguments` 对象。

`Array.from` 都可以将它们转为真正的数组。

```
// NodeList对象
let ps = document.querySelectorAll('p');
Array.from(ps).forEach(function (p) {
  console.log(p);
});

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

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。

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

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

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

上面代码中，字符串和Set结构都具有Iterator接口，因此可以被 `Array.from` 转为真正的数组。

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

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

值得提醒的是，扩展运算符（`...`）也可以将某些数据结构转为数组。

```
// arguments对象
function foo() {
  var args = [...arguments];
}

// NodeList对象
[...document.querySelectorAll('div')]
```

扩展运算符背后调用的是遍历器接口（`Symbol.iterator`），如果一个对象没有部署这个接口，就无法转换。

`Array.from` 方法则是还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有 `length` 属性。因此，任何有 `length` 属性的对象，都可以通过 `Array.from` 方法转为数组，而此时扩展运算符就无法转换。

```
Array.from({ length: 3 });
// [ undefined, undefined, undefined ]
```

上面代码中，`Array.from` 返回了一个具有三个成员的数组，每个位置的值都是 `undefined`。扩展运算符转换不了这个对象。

对于还没有部署该方法的浏览器，可以用 `Array.prototype.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，将处理后的值放入返回的数组。

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

Array.from([1, 2, 3], (x) => x * x)
// [1, 4, 9]
```

下面的例子是取出一组DOM节点的文本内容。

```
let spans = document.querySelectorAll('span.name');

// map()
let names1 = Array.prototype.map.call(spans, s => s.textContent);

// Array.from()
let names2 = Array.from(spans, s => s.textContent)
```

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]
```

另一个例子是返回各种数据的类型。

```
function typesOf () {
  return Array.from(arguments, value => typeof value)
}
typesOf(null, [], NaN)
// ['object', 'object', 'number']
```

如果 `map` 函数里面用到了 `this` 关键字，还可以传入 `Array.from` 的第三个参数，用来绑定 `this`。

`Array.from()` 可以将各种值转为真正的数组，并且还提供 `map` 功能。这实际上意味着，只要有一个原始的数据结构，你就可以先对它的值进行处理，然后转成规范的数组结构，进而就可以使用数量众多的数组方法。

```
Array.from({ length: 2 }, () => 'jack')
// ['jack', 'jack']
```

上面代码中，`Array.from` 的第一个参数指定了第二个参数运行的次数。这种特性可以让该方法的用法变得非常灵活。

`Array.from()` 的另一个应用是，将字符串转为数组，然后返回字符串的长度。因为它能正确处理各种Unicode字符，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) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

上面代码中，`Array` 方法没有参数、一个参数、三个参数时，返回结果都不一样。只有当参数个数不少于2个时，`Array()` 才会返回由参数组成的新数组。参数个数只有一个时，实际上是指定数组的长度。

`Array.of` 基本上可以用来替代 `Array()` 或 `new Array()`，并且不存在由于参数不同而导致的重载。它的行为非常统一。

```
Array.of() // []
Array.of(undefined) // [undefined]
Array.of(1) // [1]
Array.of(1, 2) // [1, 2]
```

`Array.of` 总是返回参数值组成的数组。如果没有参数，就返回一个空数组。

`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copyWithin()

数组实例的 `copyWithin` 方法，在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。

```
Array.prototype.copyWithin(target, start = 0, end = this.length)
```

它接受三个参数。

- `target`（必需）：从该位置开始替换数据。
- `start`（可选）：从该位置开始读取数据，默认为0。如果为负值，表示倒数。
- `end`（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

这三个参数都应该是数值，如果不是，会自动转为数值。

```
[1, 2, 3, 4, 5].copyWithin(0, 3)
// [4, 5, 3, 4, 5]
```

上面代码表示将从3号位直到数组结束的成员（4和5），复制到从0号位开始的位置，结果覆盖了原来的1和2。

下面是更多例子。

```
// 将3号位复制到0号位
[1, 2, 3, 4, 5].copyWithin(0, 3, 4)
// [4, 2, 3, 4, 5]

// -2相当于3号位, -1相当于4号位
[1, 2, 3, 4, 5].copyWithin(0, -2, -1)
// [4, 2, 3, 4, 5]

// 将3号位复制到0号位
[].copyWithin.call({length: 5, 3: 1}, 0, 3)
// {0: 1, 3: 1, length: 5}

// 将2号位到数组结束, 复制到0号位
var i32a = new Int32Array([1, 2, 3, 4, 5]);
i32a.copyWithin(0, 2);
// Int32Array [3, 4, 5, 4, 5]

// 对于没有部署TypedArray的copyWithin方法的平台
// 需要采用下面的写法
[].copyWithin.call(new Int32Array([1, 2, 3, 4, 5]), 0, 3, 4);
// Int32Array [4, 2, 3, 4, 5]
```

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`。

```
[1, 4, -5, 10].find((n) => n < 0)
// -5
```

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` 方法使用给定值，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` 方法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']
```

上面代码表示，`fill` 方法从1号位开始，向原数组填充7，到2号位之前结束。

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对象（详见《Iterator》一章），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"
```

如果不使用`for...of` 循环，可以手动调用遍历器对象的`next` 方法，进行遍历。

```
let letter = ['a', 'b', 'c'];
let entries = letter.entries();
console.log(entries.next().value); // [0, 'a']
console.log(entries.next().value); // [1, 'b']
console.log(entries.next().value); // [2, 'c']
```

数组实例的includes()

`Array.prototype.includes` 方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的 `includes` 方法类似。该方法属于ES7，但Babel转码器已经支持。

```
[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。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。

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

没有该方法之前，我们通常使用数组的 `index0f` 方法，检查是否包含某个值。

```
if (arr.indexOf(el) !== -1) {
  // ...
}
```

`index0f` 方法有两个缺点，一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观。二是，它内部使用严格相当运算符（`==`）进行判断，这会导致对 `NaN` 的误判。

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

`includes` 使用的是不一样的判断算法，就没有这个问题。

```
[NaN].includes(NaN)
// 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
```

另外，Map和Set数据结构有一个 `has` 方法，需要注意与 `includes` 区分。

- Map结构的 `has` 方法，是用来查找键名的，比如 `Map.prototype.has(key)`、`WeakMap.prototype.has(key)`、`Reflect.has(target, propertyKey)`。
- Set结构的 `has` 方法，是用来查找值的，比如 `Set.prototype.has(value)`、`WeakSet.prototype.has(value)`。

数组的空位

数组的空位指，数组的某一个位置没有任何值。比如，`Array` 构造函数返回的数组都是空位。

```
Array(3) // [ , , ]
```

上面代码中，`Array(3)` 返回一个具有3个空位的数组。

注意，空位不是 `undefined`，一个位置的值等于 `undefined`，依然是有值的。空位是没有任何值，`in` 运算符可以说明这一点。

```
0 in [undefined, undefined, undefined] // true
0 in [ , , ] // false
```

上面代码说明，第一个数组的0号位置是有值的，第二个数组的0号位置没有值。

ES5对空位的处理，已经很不一致了，大多数情况下会忽略空位。

- `forEach()`, `filter()`, `every()` 和 `some()` 都会跳过空位。
- `map()` 会跳过空位，但会保留这个值
- `join()` 和 `toString()` 会将空位视为 `undefined`，而 `undefined` 和 `null` 会被处理成空字符串。

```
// forEach方法
[, 'a'].forEach((x, i) => console.log(i)); // 1

// filter方法
['a', 'b'].filter(x => true) // ['a', 'b']

// every方法
[, 'a'].every(x => x === 'a') // true

// some方法
[, 'a'].some(x => x !== 'a') // false

// map方法
[, 'a'].map(x => 1) // [1]

// join方法
[, 'a', undefined, null].join('#') // "#a##"

// toString方法
[, 'a', undefined, null].toString() // ",a,,,"
```

ES6则是明确将空位转为 `undefined`。

`Array.from` 方法会将数组的空位，转为 `undefined`，也就是说，这个方法不会忽略空位。

```
Array.from(['a', 'b'])
// ["a", undefined, "b"]
```

扩展运算符 (`...`) 也会将空位转为 `undefined`。

```
[...['a', 'b']]
// ["a", undefined, "b"]
```

`copyWithin()` 会连空位一起拷贝。

```
[, 'a', 'b', ,].copyWithin(2, 0) // ["a", , "a"]
```

`fill()` 会将空位视为正常的数组位置。

```
new Array(3).fill('a') // ["a", "a", "a"]
```

`for...of` 循环也会遍历空位。

```
let arr = [, ,];
for (let i of arr) {
  console.log(1);
}
// 1
// 1
```

上面代码中，数组 `arr` 有两个空位，`for...of` 并没有忽略它们。如果改成 `map` 方法遍历，空位是会跳过的。

`entries()`、`keys()`、`values()`、`find()`和`findIndex()`会将空位处理成`undefined`。

```
// entries()  
[...[, 'a'].entries()] // [[0,undefined], [1,"a"]]  
  
// keys()  
[...[, 'a'].keys()] // [0,1]  
  
// values()  
[...[, 'a'].values()] // [undefined,"a"]  
  
// find()  
[, 'a'].find(x => true) // undefined  
  
// findIndex()  
[, 'a'].findIndex(x => true) // 0
```

由于空位的处理规则非常不统一，所以建议避免出现空位。

函数的扩展

函数参数的默认值

基本用法

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`等于空字符串，结果被改为了默认值。

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

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

ES6允许为函数的参数设置默认值，即直接写在参数定义的后面。

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

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，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

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。

与解构赋值默认值结合使用

参数默认值可以与解构赋值的默认值，结合起来使用。

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

foo({}) // undefined, 5
foo({x: 1}) // 1, 5
foo({x: 1, y: 2}) // 1, 2
foo() // TypeError: Cannot read property 'x' of undefined
```

上面代码使用了对象的解构赋值默认值，而没有使用函数参数的默认值。只有当函数 `foo` 的参数是一个对象时，变量 `x` 和 `y` 才会通过解构赋值而生成。如果函数 `foo` 调用时参数不是对象，变量 `x` 和 `y` 就不会生成，从而报错。如果参数对象没有 `y` 属性，`y` 的默认值5才会生效。

下面是另一个对象的解构赋值默认值的例子。

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

fetch('http://example.com', {})
// "GET"

fetch('http://example.com')
// 报错
```

上面代码中，如果函数 `fetch` 的第二个参数是一个对象，就可以为它的三个属性设置默认值。

上面的写法不能省略第二个参数，如果结合函数参数的默认值，就可以省略第二个参数。这时，就出现了双重默认值。

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

fetch('http://example.com')
// "GET"
```

上面代码中，函数 `fetch` 没有第二个参数时，函数参数的默认值就会生效，然后才是解构赋值的默认值生效，变量 `method` 才会取到默认值 `GET`。

再请问下面两种写法有什么差别？

```
// 写法一
function m1({x = 0, y = 0} = {}) {
  return [x, y];
}

// 写法二
function m2({x, y} = { x: 0, y: 0 }) {
  return [x, y];
}
```

上面两种写法都对函数的参数设定了默认值，区别是写法一函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；写法二函数参数的默认值是一个有具体属性的对象，但是没有设置对象解构赋值的默认值。

```
// 函数没有参数的情况
m1() // [0, 0]
m2() // [0, 0]

// x和y都有值的情况
m1({x: 3, y: 8}) // [3, 8]
m2({x: 3, y: 8}) // [3, 8]

// x有值, y无值的情况
m1({x: 3}) // [3, 0]
m2({x: 3}) // [3, undefined]

// x和y都无值的情况
m1({}) // [0, 0];
m2({}) // [undefined, undefined]

m1({z: 3}) // [0, 0]
m2({z: 3}) // [undefined, undefined]
```

参数默认值的位置

通常情况下，定义了默认值的参数，应该是函数的尾参数。因为这样比较容易看出来，到底省略了哪些参数。如果非尾部的参数设置默认值，实际上这个参数是没法省略的。

```
// 例一
function f(x = 1, y) {
  return [x, y];
}

f() // [1, undefined]
f(2) // [2, undefined])
f(, 1) // 报错
f(undefined, 1) // [1, 1]

// 例二
function f(x, y = 5, z) {
  return [x, y, z];
}

f() // [undefined, 5, undefined]
f(1) // [1, 5, undefined]
f(1, ,2) // 报错
f(1, undefined, 2) // [1, 5, 2]
```

上面代码中，有默认值的参数都不是尾参数。这时，无法只省略该参数，而不省略它后面的参数，除非显式输入 `undefined`。

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。

函数的length属性

指定了默认值以后，函数的 `length` 属性，将返回没有指定默认值的参数个数。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。比如，上面最后一个函数，定义了3个参数，其中有一个参数 `c` 指定了默认值，因此 `length` 属性等于 `3` 减去 `1`，最后得到 `2`。

这是因为 `length` 属性的含义是，该函数预期传入的参数个数。某个参数指定默认值以后，预期传入的参数个数就不包括这个参数了。同理，`rest`参数也不会计入 `length` 属性。

```
(function(...args) {}).length // 0
```

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

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

作用域

一个需要注意的地方是，如果参数默认值是一个变量，则该变量所处的作用域，与其他变量的作用域规则是一样的，即先是当前函数的作用域，然后才是全局作用域。

```
var x = 1;

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

f(2) // 2
```

上面代码中，参数 `y` 的默认值等于 `x`。调用时，由于函数作用域内部的变量 `x` 已经生成，所以 `y` 等于参数 `x`，而不是全局变量 `x`。

如果调用时，函数作用域内部的变量 `x` 没有生成，结果就会不一样。

```
let x = 1;

function f(y = x) {
  let x = 2;
```

```
    console.log(y);
}

f() // 1
```

上面代码中，函数调用时，`y` 的默认值变量 `x` 尚未在函数内部生成，所以 `x` 指向全局变量。

如果此时，全局变量 `x` 不存在，就会报错。

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

f() // ReferenceError: x is not defined
```

下面这样写，也会报错。

```
var x = 1;

function foo(x = x) {
  // ...
}

foo() // ReferenceError: x is not defined
```

上面代码中，函数 `foo` 的参数 `x` 的默认值也是 `x`。这时，默认值 `x` 的作用域是函数作用域，而不是全局作用域。由于在函数作用域中，存在变量 `x`，但是默认值在 `x` 赋值之前先执行了，所以这时属于暂时性死区（参见《let和const命令》一章），任何对 `x` 的操作都会报错。

如果参数的默认值是一个函数，该函数的作用域是其声明时所在的作用域。请看下面的例子。

```
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` 的默认值是一个匿名函数，返回值为变量 `foo`。这个匿名函数声明时，`bar` 函数的作用域还没有形成，所以匿名函数里面的 `foo` 指向外层作用域的 `foo`，输出 `outer`。

如果写成下面这样，就会报错。

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

bar() // ReferenceError: foo is not defined
```

上面代码中，匿名函数里面的 `foo` 指向函数外层，但是函数外层并没有声明 `foo`，所以就报错了。

下面是一个更复杂的例子。

```
var x = 1;
function foo(x, y = function() { x = 2; }) {
  var x = 3;
  y();
```

```
    console.log(x);
}

foo() // 3
```

上面代码中，函数 `foo` 的参数 `y` 的默认值是一个匿名函数。函数 `foo` 调用时，它的参数 `x` 的值为 `undefined`，所以 `y` 函数内部的 `x` 一开始是 `undefined`，后来被重新赋值 `2`。但是，函数 `foo` 内部重新声明了一个 `x`，值为 `3`，这两个 `x` 是不一样的，互相不产生影响，因此最后输出 `3`。

如果将 `var x = 3` 的 `var` 去除，两个 `x` 就是一样的，最后输出的就是 `2`。

```
var x = 1;
function foo(x, y = function() { x = 2; }) {
  x = 3;
  y();
  console.log(x);
}

foo() // 2
```

应用

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。

另外，可以将参数默认值设为 `undefined`，表明这个参数是可以省略的。

```
function foo(optional = undefined) { ... }
```

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
function sortNumbers() {
  return 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

含义

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。

```
console.log(...[1, 2, 3])
// 1 2 3

console.log(1, ...[2, 3, 4], 5)
// 1 2 3 4 5

[...document.querySelectorAll('div')]
// [<div>, <div>, <div>]
```

该运算符主要用于函数调用。

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

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，它们都使用了扩展运算符。该运算符将一个数组，变为参数序列。

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]);
```

替代数组的apply方法

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

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

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

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。

下面是另外一个例子。

```
// ES5
new (Date.bind.apply(Date, [null, 2015, 1, 1]))
// ES6
new Date(...[2015, 1, 1]);
```

扩展运算符的应用

(1) 合并数组

扩展运算符提供了数组合并的新写法。

```
// ES5
[1, 2].concat(more)
// ES6
[1, 2, ...more]

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

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

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

(2) 与解构赋值结合

扩展运算符可以与解构赋值结合起来，用于生成数组。

```
// ES5
a = list[0], rest = list.slice(1)
// ES6
[a, ...rest] = list
```

下面是另外一些例子。

```
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 // []
```

如果将扩展运算符用于数组赋值，只能放在参数的最后一一位，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
```

(3) 函数的返回值

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`。

(4) 字符串

扩展运算符还可以将字符串转为真正的数组。

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

上面的写法，有一个重要的好处，那就是能够正确识别32位的Unicode字符。

```
'x\uD83D\uDE80y'.length // 4
[...'x\uD83D\uDE80y'].length // 3
```

上面代码的第一种写法，JavaScript会将32位Unicode字符，识别为2个字符，采用扩展运算符就没有这个问题。因此，正确返回字符串长度的函数，可以像下面这样写。

```
function length(str) {
  return [...str].length;
}

length('x\uD83D\uDE80y') // 3
```

凡是涉及到操作32位Unicode字符的函数，都有这个问题。因此，最好都用扩展运算符改写。

```
let str = 'x\uD83D\uDE80y';

str.split('').reverse().join('')
// 'y\uDE80\uD83Dx'

[...str].reverse().join('')
// 'y\uD83D\uDE80x'
```

上面代码中，如果不使用扩展运算符，字符串的 `reverse` 操作就不正确。

(5) 实现了Iterator接口的对象

任何Iterator接口的对象，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。它不是数组，而是一个类似数组的对象。这时，扩展运算符可以将其转为真正的数组，原因就在于 `NodeList` 对象实现了Iterator接口。

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

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

// TypeError: Cannot spread non-iterable object.
let arr = [...arrayLike];
```

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

(6) Map和Set结构，Generator函数

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象，对这个遍历器对象执行扩展运算符，就会将内部遍历得到的值，转为一个数组。

如果对没有`iterator`接口的对象，使用扩展运算符，将会报错。

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

严格模式

从ES5开始，函数内部可以设定为严格模式。

```
function doSomething(a, b) {
  'use strict';
  // code
}
```

《ECMAScript 2016标准》做了一点修改，规定只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错。

```
// 报错
function doSomething(a, b = a) {
  'use strict';
```

```
// code
}

// 报错
const doSomething = function ({a, b}) {
  'use strict';
  // code
};

// 报错
const doSomething = (...a) => {
  'use strict';
  // code
};

const obj = {
  // 报错
  doSomething({a, b}) {
    'use strict';
    // code
  }
};
```

这样规定的原因是，函数内部的严格模式，同时适用于函数体代码和函数参数代码。但是，函数执行的时候，先执行函数参数代码，然后再执行函数体代码。这样就有一个不合理的地方，只有从函数体代码之中，才能知道参数代码是否应该以严格模式执行，但是参数代码却应该先于函数体代码执行。

```
// 报错
function doSomething(value = 070) {
  'use strict';
  return value;
}
```

上面代码中，参数 `value` 的默认值是八进制数 `070`，但是严格模式下不能用前缀 `0` 表示八进制，所以应该报错。但实际上，JavaScript引擎会先成功执行 `value = 070`，然后进入函数体内部，发现需要用严格模式执行，这时才会报错。

虽然可以先解析函数体代码，再执行参数代码，但是这样无疑就增加了复杂性。因此，标准索性禁止了这种用法，只要参数使用了默认值、解构赋值、或者扩展运算符，就不能显式指定严格模式。

两种方法可以规避这种限制。第一种是设定全局性的严格模式，这是合法的。

```
'use strict';

function doSomething(a, b = a) {
  // code
}
```

第二种是把函数包在一个无参数的立即执行函数里面。

```
const doSomething = (function () {
  'use strict';
  return function (value = 42) {
    return value;
};
}());
```

name 属性

函数的 `name` 属性，返回该函数的函数名。

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

这个属性早就被浏览器广泛支持，但是直到 ES6，才将其写入了标准。

需要注意的是，ES6 对这个属性的行为做出了一些修改。如果将一个匿名函数赋值给一个变量，ES5 的 `name` 属性，会返回空字符串，而 ES6 的 `name` 属性会返回实际的函数名。

```
var f = function () {};  
  
// ES5  
f.name // ""  
  
// ES6  
f.name // "f"
```

上面代码中，变量 `f` 等于一个匿名函数，ES5 和 ES6 的 `name` 属性返回的值不一样。

如果将一个具名函数赋值给一个变量，则 ES5 和 ES6 的 `name` 属性都返回这个具名函数原本的名字。

```
const bar = function baz() {};  
  
// ES5  
bar.name // "baz"  
  
// ES6  
bar.name // "baz"
```

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

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

`bind` 返回的函数，`name` 属性值会加上 `bound` 前缀。

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

箭头函数

基本用法

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.last;
}
```

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。如果不使用箭头函数，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]]
```

使用注意点

箭头函数有几个使用注意点。

- (1) 函数体内的 `this` 对象，就是定义时所在的对象，而不是使用时所在的对象。
- (2) 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。
- (3) 不可以使用 `arguments` 对象，该对象在函数体内不存在。如果要用，可以用Rest参数代替。
- (4) 不可以使用 `yield` 命令，因此箭头函数不能用作Generator函数。

上面四点中，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。

```
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}

var id = 21;

foo.call({ id: 42 });
// id: 42
```

上面代码中，`setTimeout` 的参数是一个箭头函数，这个箭头函数的定义生效是在 `foo` 函数生成时，而它的真正执行要等到100毫秒后。如果是普通函数，执行时 `this` 应该指向全局对象 `window`，这时应该输出 `21`。但是，箭头函数导致 `this` 总是指向函数定义生效时所在的对象（本例是 `{id: 42}`），所以输出的是 `42`。

箭头函数可以让 `setTimeout` 里面的 `this`，绑定定义时所在的作用域，而不是指向运行时所在的作用域。下面是另一个例子。

```
function Timer() {
  this.s1 = 0;
  this.s2 = 0;
  // 箭头函数
  setInterval(() => this.s1++, 1000);
  // 普通函数
  setInterval(function () {
    this.s2++;
  }, 1000);
}

var timer = new Timer();

setTimeout(() => console.log('s1: ', timer.s1), 3100);
setTimeout(() => console.log('s2: ', timer.s2), 3100);
// s1: 3
// s2: 0
```

上面代码中，`Timer` 函数内部设置了两个定时器，分别使用了箭头函数和普通函数。前者的 `this` 绑定定义时所在的作用域（即 `Timer` 函数），后者的 `this` 指向运行时所在的作用域（即全局对象）。所以，3100毫秒之后，`timer.s1` 被更新了3次，而 `timer.s2` 一次都没更新。

箭头函数可以让 `this` 指向固定化，这种特性很有利于封装回调函数。下面是一个例子，DOM事件的回调函数封装在一个对象里面。

```

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，这导致这个箭头函数里面的 `this`，总是指向 `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 `document` 对象。

`this` 指向的固定化，并不是因为箭头函数内部有绑定 `this` 的机制，实际原因是箭头函数根本没有自己的 `this`，导致内部的 `this` 就是外层代码块的 `this`。正是因为它没有 `this`，所以也就不能用作构造函数。

所以，箭头函数转成ES5的代码如下。

```

// ES6
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}

// ES5
function foo() {
  var _this = this;

  setTimeout(function () {
    console.log('id:', _this.id);
  }, 100);
}

```

上面代码中，转换后的ES5版本清楚地说明了，箭头函数里面根本没有自己的 `this`，而是引用外层的 `this`。

请问下面的代码之中有几个 `this`？

```

function foo() {
  return () => {
    return () => {
      return () => {
        console.log('id:', this.id);
      };
    };
  };
}

var f = foo.call({id: 1});

var t1 = f.call({id: 2})(); // id: 1
var t2 = f().call({id: 3})(); // id: 1
var t3 = f()().call({id: 4}); // id: 1

```

上面代码之中，只有一个 `this`，就是函数 `foo` 的 `this`，所以 `t1`、`t2`、`t3` 都输出同样的结果。因为所有的内层函数都是箭头函数，都没有自己的 `this`，它们的 `this` 其实都是最外层 `foo` 函数的 `this`。

除了 `this`，以下三个变量在箭头函数之中也是不存在的，指向外层函数的对应变量：`arguments`、`super`、`new.target`。

```
function foo() {
  setTimeout(() => {
    console.log('args:', arguments);
  }, 100);
}

foo(2, 4, 6, 8)
// args: [2, 4, 6, 8]
```

上面代码中，箭头函数内部的变量 `arguments`，其实是函数 `foo` 的 `arguments` 变量。

另外，由于箭头函数没有自己的 `this`，所以当然也就不能用 `call()`、`apply()`、`bind()` 这些方法去改变 `this` 的指向。

```
(function() {
  return [
    () => this.x).bind({ x: 'inner' })()
];
}).call({ x: 'outer' });
// ['outer']
```

上面代码中，箭头函数没有自己的 `this`，所以 `bind` 方法无效，内部的 `this` 指向外部的 `this`。

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。

嵌套的箭头函数

箭头函数内部，还可以再使用箭头函数。下面是一个ES5语法的多重嵌套函数。

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

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

上面这个函数，可以使用箭头函数改写。

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

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

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

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

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

addThenMult(5)
// 12
```

如果觉得上面的写法可读性比较差，也可以采用下面的写法。

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

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

箭头函数还有一个功能，就是可以很方便地改写λ演算。

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

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

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

绑定 this

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冒号（`::`），双冒号左边是一个对象，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。

```
foo::bar;
// 等同于
bar.bind(foo);

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

const hasOwnProperty = Object.prototype.hasOwnProperty;
function hasOwn(obj, key) {
  return obj::hasOwnProperty(key);
}
```

如果双冒号左边为空，右边是一个对象的方法，则等于将该方法绑定在该对象上面。

```
var method = obj::obj.foo;
// 等同于
var method = ::obj.foo;

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

由于双冒号运算符返回的还是原对象，因此可以采用链式写法。

```
// 例一
import { map, takeWhile, forEach } from "iterlib";

getPlayers()
  ::map(x => x.character())
```

```
:::takeWhile(x => x.strength > 100)
:::forEach(x => console.log(x));

// 例二
let { find, html } = jake;

document.querySelectorAll("div.myClass")
:::find("p")
:::html("hahaha");
```

尾调用优化

什么是尾调用？

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

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

上面代码中，函数f的最后一步是调用函数g，这就叫尾调用。

以下三种情况，都不属于尾调用。

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

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

// 情况三
function f(x){
  g(x);
}
```

上面代码中，情况一是调用函数g之后，还有赋值操作，所以不属于尾调用，即使语义完全一样。情况二也属于调用后还有操作，即使写在一行内。情况三等同于下面的代码。

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

尾调用不一定出现在函数尾部，只要是最后一步操作即可。

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

上面代码中，函数m和n都属于尾调用，因为它们都是函数f的最后一步操作。

尾调用优化

尾调用之所以与其他调用不同，就在于它的特殊的调用位置。

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

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

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

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

// 等同于
g(3);
```

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

这就叫做“尾调用优化”（Tail call optimization），即只保留内层函数的调用帧。如果所有函数都是尾调用，那么完全可以在每次执行时，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。

注意，只有不再用到外层函数的内部变量，内层函数的调用帧才会取代外层函数的调用帧，否则就无法进行“尾调用优化”。

```
function addOne(a){
  var one = 1;
  function inner(b){
    return b + one;
  }
  return inner(a);
}
```

上面的函数不会进行尾调用优化，因为内层函数 `inner` 用到了外层函数 `addOne` 的内部变量 `one`。

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
```

还有一个比较著名的例子，就是计算fibonacci 数列，也能充分说明尾递归优化的重要性

如果是非尾递归的fibonacci 递归方法

```
function Fibonacci (n) {
  if (n <= 1) {return 1};

  return Fibonacci(n - 1) + Fibonacci(n - 2);
}

Fibonacci(10); // 89
// Fibonacci(100)
// Fibonacci(500)
// 堆栈溢出了
```

如果我们使用尾递归优化过的fibonacci 递归算法

```
function Fibonacci2 (n , ac1 = 1 , ac2 = 1) {
  if( n <= 1 ) {return ac2};

  return Fibonacci2 (n - 1, ac2, ac1 + ac2);
}

Fibonacci2(100) // 573147844013817200000
Fibonacci2(1000) // 7.0330367711422765e+208
Fibonacci2(10000) // Infinity
```

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。

严格模式

ES6的尾调用优化只在严格模式下开启，正常模式是无效的。

这是因为在正常模式下，函数内部有两个变量，可以跟踪函数的调用栈。

- `func.arguments`：返回调用时函数的参数。
- `func.caller`：返回调用当前函数的那个函数。

尾调用优化发生时，函数的调用栈会改写，因此上面两个变量就会失真。严格模式禁用这两个变量，所以尾调用模式仅在严格模式下生效。

```
function restricted() {
  "use strict";
  restricted.caller;    // 报错
  restricted.arguments; // 报错
}
restricted();
```

尾递归优化的实现

尾递归优化只在严格模式下生效，那么正常模式下，或者那些不支持该功能的环境中，有没有办法也使用尾递归优化呢？回答是可以的，就是自己实现尾递归优化。

它的原理非常简单。尾递归之所以需要优化，原因是调用栈太多，造成溢出，那么只要减少调用栈，就不会溢出。怎么做可以减少调用栈呢？就是采用“循环”换掉“递归”。

下面是一个正常的递归函数。

```
function sum(x, y) {
  if (y > 0) {
    return sum(x + 1, y - 1);
  } else {
    return x;
  }
}

sum(1, 100000)
// Uncaught RangeError: Maximum call stack size exceeded(...)
```

上面代码中，`sum` 是一个递归函数，参数 `x` 是需要累加的值，参数 `y` 控制递归次数。一旦指定 `sum` 递归100000次，就会报错，提示超出调用栈的最大次数。

蹦床函数（trampoline）可以将递归执行转为循环执行。

```
function trampoline(f) {
  while (f && f instanceof Function) {
    f = f();
  }
  return f;
}
```

上面就是蹦床函数的一个实现，它接受一个函数 `f` 作为参数。只要 `f` 执行后返回一个函数，就继续执行。注意，这里是返回一个函数，然后执行该函数，而不是函数里面调用函数，这样就避免了递归执行，从而就消除了调用栈过大的问题。

然后，要做的就是将原来的递归函数，改写为每一步返回另一个函数。

```
function sum(x, y) {
  if (y > 0) {
    return sum.bind(null, x + 1, y - 1);
  } else {
    return x;
  }
}
```

上面代码中，`sum` 函数的每次执行，都会返回自身的另一个版本。

现在，使用蹦床函数执行 `sum`，就不会发生调用栈溢出。

```
trampoline(sum(1, 100000))
// 100001
```

蹦床函数并不是真正的尾递归优化，下面的实现才是。

```
function tco(f) {
  var value;
  var active = false;
  var accumulated = [];

  return function accumulator() {
    accumulated.push(arguments);
    if (!active) {
      active = true;
      while (accumulated.length) {
```

```

        value = f.apply(this, accumulated.shift());
    }
    active = false;
    return value;
}
};

var sum = tco(function(x, y) {
    if (y > 0) {
        return sum(x + 1, y - 1)
    }
    else {
        return x
    }
});

sum(1, 100000)
// 100001

```

上面代码中，`tco` 函数是尾递归优化的实现，它的奥妙就在于状态变量 `active`。默认情况下，这个变量是不激活的。一旦进入尾递归优化的过程，这个变量就激活了。然后，每一轮递归 `sum` 返回的都是 `undefined`，所以就避免了递归执行；而 `accumulated` 数组存放每一轮 `sum` 执行的参数，总是有值的，这就保证了 `accumulator` 函数内部的 `while` 循环总是会执行。这样就很巧妙地将“递归”改成了“循环”，而后一轮的参数会取代前一轮的参数，保证了调用栈只有一层。

函数参数的尾逗号

ECMAScript 2017 将允许函数的最后一个参数有尾逗号（trailing comma）。

此前，函数定义和调用时，都不允许最后一个参数后面出现逗号。

```

function clownsEverywhere(
    param1,
    param2
) { /* ... */ }

clownsEverywhere(
    'foo',
    'bar'
);

```

上面代码中，如果在 `param2` 或 `bar` 后面加一个逗号，就会报错。

这样的话，如果以后修改代码，想为函数 `clownsEverywhere` 添加第三个参数，就势必要在第二个参数后面添加一个逗号。这对版本管理系统来说，就会显示，添加逗号的那一行也发生了变动。这看上去有点冗余，因此新的语法允许定义和调用时，尾部直接有一个逗号。

```

function clownsEverywhere(
    param1,
    param2,
) { /* ... */ }

clownsEverywhere(
    'foo',
    'bar',
);

```

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。

```
var foo = 'bar';
var baz = {foo};
baz // {foo: "bar"}
```

```
// 等同于
var baz = {foo: foo};
```

上面代码表明，ES6 允许在对象之中，直接写变量。这时，属性名为变量名，属性值为变量的值。下面是另一个例子。

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

// 等同于

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

f(1, 2) // Object {x: 1, y: 2}
```

除了属性简写，方法也可以简写。

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

// 等同于

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

下面是一个实际的例子。

```
var birth = '2000/01/01';

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}
```

CommonJS模块输出变量，就非常合适使用简洁写法。

```
var ms = {};

function getItem (key) {
  return key in ms ? ms[key] : null;
}

function.setItem (key, value) {
  ms[key] = value;
}

function.clear () {
  ms = {};
}

module.exports = { getItem, setItem, clear };
// 等同于
module.exports = {
  getItem: getItem,
  setItem: setItem,
  clear: clear
};
```

属性的赋值器 (setter) 和取值器 (getter) ，事实上也是采用这种写法。

```
var cart = {
  _wheels: 4,

  get wheels () {
    return this._wheels;
  },

  set wheels (value) {
    if (value < this._wheels) {
      throw new Error('数值太小了！');
    }
    this._wheels = value;
  }
}
```

注意，简洁写法的属性名总是字符串，这会导致一些看上去比较奇怪的结果。

```
var obj = {
  class () {}
};

// 等同于

var obj = {
  'class': function() {}
};
```

上面代码中，`class` 是字符串，所以不会因为它属于关键字，而导致语法解析报错。

如果某个方法的值是一个Generator函数，前面需要加上星号。

```
var obj = {
  * m(){
    yield 'hello world';
  }
};
```

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';
  }
};

obj.hello() // hi
```

注意，属性名表达式与简洁表示法，不能同时使用，会报错。

```
// 报错
var foo = 'bar';
var bar = 'abc';
var baz = { [foo] };

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

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串 `[object Object]`，这一点要特别小心。

```
const keyA = {a: 1};
const keyB = {b: 2};

const myObject = {
  [keyA]: 'valueA',
  [keyB]: 'valueB'
};

myObject // Object {[object Object]: "valueB"}
```

上面代码中，`[keyA]` 和 `[keyB]` 得到的都是 `[object Object]`，所以 `[keyB]` 会把 `[keyA]` 覆盖掉，而 `myObject` 最后只有一个 `[object Object]` 属性。

方法的 name 属性

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

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

person.sayName.name // "sayName"
```

上面代码中，方法的 `name` 属性返回函数名（即方法名）。

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

```
const obj = {
  get foo() {},
  set foo(x) {}
};

obj.foo.name
// TypeError: Cannot read property 'name' of undefined

const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');

descriptor.get.name // "get foo"
descriptor.set.name // "set foo"
```

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

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

var doSomething = function() {
```

```
// ...
};

doSomething.bind().name // "bound doSomething"
```

如果对象的方法是一个 Symbol 值，那么 `name` 属性返回的是这个 Symbol 值的描述。

```
const key1 = Symbol('description');
const key2 = Symbol();
let obj = {
  [key1](): {},
  [key2](): {},
};
obj[key1].name // "[description]"
obj[key2].name // ""
```

上面代码中，`key1` 对应的 Symbol 值有描述，`key2` 没有。

Object.is()

ES5比较两个值是否相等，只有两个运算符：相等运算符（`==`）和严格相等运算符（`===`）。它们都有缺点，前者会自动转换数据类型，后者的 `NaN` 不等于自身，以及 `+0` 等于 `-0`。JavaScript缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。

ES6提出“Same-value equality”（同值相等）算法，用来解决这个问题。`Object.is` 就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（`==`）的行为基本一致。

```
Object.is('foo', 'foo')
// true
Object.is({}, {})
// false
```

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()

基本用法

`Object.assign` 方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

```
var target = { a: 1 };
var source1 = { b: 2 };
var source2 = { c: 3 };

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

`Object.assign` 方法的第一个参数是目标对象，后面的参数都是源对象。

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}
```

如果只有一个参数，`Object.assign` 会直接返回该参数。

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

如果该参数不是对象，则会先转成对象，然后返回。

```
typeof Object.assign(2) // "object"
```

由于 `undefined` 和 `null` 无法转成对象，所以如果它们作为参数，就会报错。

```
Object.assign(undefined) // 报错
Object.assign(null) // 报错
```

如果非对象参数出现在源对象的位置（即非首参数），那么处理规则有所不同。首先，这些参数都会转成对象，如果无法转成对象，就会跳过。这意味着，如果 `undefined` 和 `null` 不在首参数，就不会报错。

```
let obj = {a: 1};
Object.assign(obj, undefined) === obj // true
Object.assign(obj, null) === obj // true
```

其他类型的值（即数值、字符串和布尔值）不在首参数，也不会报错。但是，除了字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果。

```
var v1 = 'abc';
var v2 = true;
var v3 = 10;

var obj = Object.assign({}, v1, v2, v3);
console.log(obj); // { "0": "a", "1": "b", "2": "c" }
```

上面代码中，`v1`、`v2`、`v3` 分别是字符串、布尔值和数值，结果只有字符串合入目标对象（以字符数组的形式），数值和布尔值都会被忽略。这是因为只有字符串的包装对象，会产生可枚举属性。

```
Object(true) // {[ [PrimitiveValue] ]: true}
Object(10) // {[ [PrimitiveValue] ]: 10}
Object('abc') // {0: "a", 1: "b", 2: "c", length: 3, [ [PrimitiveValue] ]: "abc"}
```

上面代码中，布尔值、数值、字符串分别转成对应的包装对象，可以看到它们的原始值都在包装对象的内部属性 `[[PrimitiveValue]]` 上面，这个属性是不会被 `Object.assign` 拷贝的。只有字符串的包装对象，会产生可枚举的实义属性，那些属性则会被拷贝。

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

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

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

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

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

注意点

`Object.assign` 方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。

```
var obj1 = {a: {b: 1}};  
var obj2 = Object.assign({}, obj1);  
  
obj1.a.b = 2;  
obj2.a.b // 2
```

上面代码中，源对象 `obj1` 的 `a` 属性的值是一个对象，`Object.assign` 拷贝得到的是这个对象的引用。这个对象的任何变化，都会反映到目标对象上面。

对于这种嵌套的对象，一旦遇到同名属性，`Object.assign` 的处理方法是替换，而不是添加。

```
var target = { a: { b: 'c', d: 'e' } }  
var source = { a: { b: 'hello' } }  
Object.assign(target, source)  
// { a: { b: 'hello' } }
```

上面代码中，`target` 对象的 `a` 属性被 `source` 对象的 `a` 属性整个替换掉了，而不会得到 `{ a: { b: 'hello', d: 'e' } }` 的结果。这通常不是开发者想要的，需要特别小心。

有一些函数库提供 `Object.assign` 的定制版本（比如Lodash的 `_.defaultsDeep` 方法），可以解决浅拷贝的问题，得到深拷贝的合并。

注意，`Object.assign` 可以用来处理数组，但是会把数组视为对象。

```
Object.assign([1, 2, 3], [4, 5])  
// [4, 5, 3]
```

上面代码中，`Object.assign` 把数组视为属性名为0、1、2的对象，因此源数组的0号属性 `4` 覆盖了目标数组的0号属性 `1`。

常见用途

`Object.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 `Object.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) {
  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。`Object.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。

注意，由于存在深拷贝的问题，`DEFAULTS` 对象和 `options` 对象的所有属性的值，都只能是简单类型，而不能指向另一个对象。否则，将导致 `DEFAULTS` 对象的该属性不起作用。

属性的可枚举性

对象的每个属性都有一个描述对象（Descriptor），用来控制该属性的行为。`Object.getOwnPropertyDescriptor` 方法可以获取该属性的描述对象。

```
let obj = { foo: 123 };
Object.getOwnPropertyDescriptor(obj, 'foo')
// {
//   value: 123,
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
```

描述对象的 `enumerable` 属性，称为“可枚举性”，如果该属性为 `false`，就表示某些操作会忽略当前属性。

ES5有三个操作会忽略 `enumerable` 为 `false` 的属性。

- `for...in` 循环：只遍历对象自身的和继承的可枚举的属性
- `Object.keys()`：返回对象自身的所有可枚举的属性的键名
- `JSON.stringify()`：只串行化对象自身的可枚举的属性

ES6新增了一个操作 `Object.assign()`，会忽略 `enumerable` 为 `false` 的属性，只拷贝对象自身的可枚举的属性。

这四个操作之中，只有 `for...in` 会返回继承的属性。实际上，引入 `enumerable` 的最初目的，就是让某些属性可以规避掉 `for...in` 操作。比如，对象原型的 `toString` 方法，以及数组的 `length` 属性，就通过这种手段，不会被 `for...in` 遍历到。

```
Object.getOwnPropertyDescriptor(Object.prototype, 'toString').enumerable
// false

Object.getOwnPropertyDescriptor([], 'length').enumerable
// false
```

上面代码中，`toString` 和 `length` 属性的 `enumerable` 都是 `false`，因此 `for...in` 不会遍历到这两个继承自原型的属性。

另外，ES6规定，所有Class的原型的方法都是不可枚举的。

```
Object.getOwnPropertyDescriptor(class {foo() {}}.prototype, 'foo').enumerable
// false
```

总的来说，操作中引入继承的属性会让问题复杂化，大多数时候，我们只关心对象自身的属性。所以，尽量不要用 `for...in` 循环，而用 `Object.keys()` 代替。

属性的遍历

ES6一共有5种方法可以遍历对象的属性。

(1) `for...in`

`for...in` 循环遍历对象自身的和继承的可枚举属性（不含Symbol属性）。

(2) `Object.keys(obj)`

`Object.keys` 返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含Symbol属性）。

(3) `Object.getOwnPropertyNames(obj)`

`Object.getOwnPropertyNames` 返回一个数组，包含对象自身的所有属性（不含Symbol属性，但是包括不可枚举属性）。

(4) `Object.getOwnPropertySymbols(obj)`

`Object.getOwnPropertySymbols` 返回一个数组，包含对象自身的所有Symbol属性。

(5) `Reflect.ownKeys(obj)`

`Reflect.ownKeys` 返回一个数组，包含对象自身的所有属性，不管是属性名是Symbol或字符串，也不管是否可枚举。

以上的5种方法遍历对象的属性，都遵守同样的属性遍历的次序规则。

- 首先遍历所有属性名为数值的属性，按照数字排序。
- 其次遍历所有属性名为字符串的属性，按照生成时间排序。
- 最后遍历所有属性名为Symbol值的属性，按照生成时间排序。

```
Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })
// ['2', '10', 'b', 'a', Symbol()]
```

上面代码中，`Reflect.ownKeys` 方法返回一个数组，包含了参数对象的所有属性。这个数组的属性次序是这样的，首先是数值属性 `2` 和 `10`，其次是字符串属性 `b` 和 `a`，最后是Symbol属性。

__proto__ 属性，`Object.setPrototypeOf()`，`Object.getPrototypeOf()`

(1) __proto__ 属性

__proto__ 属性（前后各两个下划线），用来读取或设置当前对象的 `prototype` 对象。目前，所有浏览器（包括IE11）都部署了这个属性。

```
// es6的写法
var obj = {
  method: function() { ... }
};
obj.__proto__ = someOtherObj;

// es5的写法
var obj = Object.create(someOtherObj);
obj.method = function() { ... };
```

该属性没有写入ES6的正文，而是写入了附录，原因是 __proto__ 前后的双下划线，说明它本质上是一个内部属性，而不是一个正式的对外的API，只是由于浏览器广泛支持，才被加入了ES6。标准明确规定，只有浏览器必须部署这个属性，其他运行环境不一定需要部署，而且新的代码最好认为这个属性是不存在的。因此，无论从语义的角度，还是从

兼容性的角度，都不要使用这个属性，而是使用下面的 `Object.setPrototypeOf()`（写操作）、`Object.getPrototypeOf()`（读操作）、`Object.create()`（生成操作）代替。

在实现上，`__proto__` 调用的是 `Object.prototype.__proto__`，具体实现如下。

```
Object.defineProperty(Object.prototype, '__proto__', {
  get() {
    let _thisObj = Object(this);
    return Object.getPrototypeOf(_thisObj);
  },
  set(proto) {
    if (this === undefined || this === null) {
      throw new TypeError();
    }
    if (!isObject(this)) {
      return undefined;
    }
    if (!isObject(proto)) {
      return undefined;
    }
    let status = Reflect.setPrototypeOf(this, proto);
    if (!status) {
      throw new TypeError();
    }
  },
});
function isObject(value) {
  return Object(value) === value;
}
```

如果一个对象本身部署了 `__proto__` 属性，则该属性的值就是对象的原型。

```
Object.getPrototypeOf({ __proto__: null })
// null
```

(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;
}
```

下面是一个例子。

```
let proto = {};
let obj = { x: 10 };
Object.setPrototypeOf(obj, proto);

proto.y = 20;
proto.z = 40;
```

```
obj.x // 10  
obj.y // 20  
obj.z // 40
```

上面代码将proto对象设为obj对象的原型，所以从obj对象可以读取proto对象的属性。

(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);
```

下面是一个例子。

```
function Rectangle() {}  
  
var rec = new Rectangle();  
  
Object.getPrototypeOf(rec) === Rectangle.prototype  
// true  
  
Object.setPrototypeOf(rec, Object.prototype);  
Object.getPrototypeOf(rec) === Rectangle.prototype  
// false
```

Object.keys(), Object.values(), Object.entries()

Object.keys()

ES5 引入了 `Object.keys` 方法，返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键名。

```
var obj = { foo: 'bar', baz: 42 };  
Object.keys(obj)  
// ["foo", "baz"]
```

ES2017 引入了跟 `Object.keys` 配套的 `Object.values` 和 `Object.entries`，作为遍历一个对象的补充手段。

```
let {keys, values, entries} = Object;  
let obj = { a: 1, b: 2, c: 3 };  
  
for (let key of keys(obj)) {  
  console.log(key); // 'a', 'b', 'c'  
}  
  
for (let value of values(obj)) {  
  console.log(value); // 1, 2, 3  
}  
  
for (let [key, value] of entries(obj)) {  
  console.log([key, value]); // ['a', 1], ['b', 2], ['c', 3]  
}
```

Object.values()

`Object.values` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值。

```
var obj = { foo: 'bar', baz: 42 };
```

```
Object.values(obj)
// ["bar", 42]
```

返回数组的成员顺序，与本章的《属性的遍历》部分介绍的排列规则一致。

```
var obj = { 100: 'a', 2: 'b', 7: 'c' };
Object.values(obj)
// ["b", "c", "a"]
```

上面代码中，属性名为数值的属性，是按照数值大小，从小到大遍历的，因此返回的顺序是 `b`、`c`、`a`。

`Object.values` 只返回对象自身的可遍历属性。

```
var obj = Object.create({}, {p: {value: 42}});
Object.values(obj) // []
```

上面代码中，`Object.create` 方法的第二个参数添加的对象属性（属性 `p`），如果不显式声明，默认是不可遍历的，因为 `p` 是继承的属性，而不是对象自身的属性。`Object.values` 不会返回这个属性。

`Object.values` 会过滤属性名为 Symbol 值的属性。

```
Object.values({ [Symbol()]: 123, foo: 'abc' });
// ['abc']
```

如果 `Object.values` 方法的参数是一个字符串，会返回各个字符组成的一个数组。

```
Object.values('foo')
// ['f', 'o', 'o']
```

上面代码中，字符串会先转成一个类似数组的对象。字符串的每个字符，就是该对象的一个属性。因此，`Object.values` 返回每个属性的键值，就是各个字符组成的一个数组。

如果参数不是对象，`Object.values` 会先将其转为对象。由于数值和布尔值的包装对象，都不会为实例添加非继承的属性。所以，`Object.values` 会返回空数组。

```
Object.values(42) // []
Object.values(true) // []
```

Object.entries

`Object.entries` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对数组。

```
var obj = { foo: 'bar', baz: 42 };
Object.entries(obj)
// [ ["foo", "bar"], ["baz", 42] ]
```

除了返回值不一样，该方法的行为与 `Object.values` 基本一致。

如果原对象的属性名是一个Symbol值，该属性会被省略。

```
Object.entries({ [Symbol()]: 123, foo: 'abc' });
// [ [ 'foo', 'abc' ] ]
```

上面代码中，原对象有两个属性，`Object.entries` 只输出属性名非Symbol值的属性。将来可能会有 `Reflect.ownEntries()` 方法，返回对象自身的所有属性。

`Object.entries` 的基本用途是遍历对象的属性。

```
let obj = { one: 1, two: 2 };
for (let [k, v] of Object.entries(obj)) {
  console.log(`${JSON.stringify(k)}: ${JSON.stringify(v)}`);
}
// "one": 1
// "two": 2
```

`Object.entries` 方法的一个用处是，将对象转为真正的 `Map` 结构。

```
var obj = { foo: 'bar', baz: 42 };
var map = new Map(Object.entries(obj));
map // Map { foo: "bar", baz: 42 }
```

自己实现 `Object.entries` 方法，非常简单。

```
// Generator函数的版本
function* entries(obj) {
  for (let key of Object.keys(obj)) {
    yield [key, obj[key]];
  }
}

// 非Generator函数的版本
function entries(obj) {
  let arr = [];
  for (let key of Object.keys(obj)) {
    arr.push([key, obj[key]]);
  }
  return arr;
}
```

对象的扩展运算符

目前，ES7有一个[提案](#)，将Rest运算符（解构赋值）/扩展运算符（`...`）引入对象。Babel转码器已经支持这项功能。

(1) 解构赋值

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

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x // 1
y // 2
z // { a: 3, b: 4 }
```

上面代码中，变量 `z` 是解构赋值所在的对象。它获取等号右边的所有尚未读取的键（`a` 和 `b`），将它们连同值一起拷贝过来。

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

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

解构赋值必须是最后一个参数，否则会报错。

```
let { ...x, y, z } = obj; // 句法错误
let { x, ...y, ...z } = obj; // 句法错误
```

上面代码中，解构赋值不是最后一个参数，所以会报错。

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

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

上面代码中，`x` 是解构赋值所在的对象，拷贝了对象 `obj` 的 `a` 属性。`a` 属性引用了一个对象，修改这个对象的值，会影响到解构赋值对它的引用。

另外，解构赋值不会拷贝继承自原型对象的属性。

```
let o1 = { a: 1 };
let o2 = { b: 2 };
o2.__proto__ = o1;
let o3 = { ...o2 };
o3 // { b: 2 }
```

上面代码中，对象 `o3` 是 `o2` 的拷贝，但是只复制了 `o2` 自身的属性，没有复制它的原型对象 `o1` 的属性。

下面是另一个例子。

```
var o = Object.create({ x: 1, y: 2 });
o.z = 3;

let { x, ...{ y, z } } = o;
x // 1
y // undefined
z // 3
```

上面代码中，变量 `x` 是单纯的解构赋值，所以可以读取继承的属性；解构赋值产生的变量 `y` 和 `z`，只能读取对象自身的属性，所以只有变量 `z` 可以赋值成功。

解构赋值的一个用处，是扩展某个函数的参数，引入其他操作。

```
function baseFunction({ a, b }) {
  // ...
}
function wrapperFunction({ x, y, ...restConfig }) {
  // 使用x和y参数进行操作
  // 其余参数传给原始函数
  return baseFunction(restConfig);
}
```

上面代码中，原始函数 `baseFunction` 接受 `a` 和 `b` 作为参数，函数 `wrapperFunction` 在 `baseFunction` 的基础上进行了扩展，能够接受多余的参数，并且保留原始函数的行为。

(2) 扩展运算符

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

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

这等同于使用 `Object.assign` 方法。

```
let aClone = { ...a };
```

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

扩展运算符可以用于合并两个对象。

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

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

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

上面代码中，`a` 对象的 `x` 属性和 `y` 属性，拷贝到新对象后会被覆盖掉。

这用来修改现有对象部分的部分属性就很方便了。

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

上面代码中，`newVersion` 对象自定义了 `name` 属性，其他属性全部复制自 `previousVersion` 对象。

如果把自定义属性放在扩展运算符前面，就变成了设置新对象的默认属性值。

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

扩展运算符的参数对象之中，如果有取值函数 `get`，这个函数是会执行的。

```
// 并不会抛出错误，因为x属性只是被定义，但没执行
let aWithXGetter = {
  ...a,
  get x() {
    throws new Error('not thrown yet');
  }
};
```

```
// 会抛出错误，因为x属性被执行了
let runtimeError = {
  ...a,
  ...{
    get x() {
      throws new Error('thrown now');
    }
  }
};
```

如果扩展运算符的参数是 `null` 或 `undefined`，这个两个值会被忽略，不会报错。

```
let emptyObject = { ...null, ...undefined }; // 不报错
```

Object.getOwnPropertyDescriptors()

ES5有一个 `Object.getOwnPropertyDescriptor` 方法，返回某个对象属性的描述对象（descriptor）。

```
var obj = { p: 'a' };

Object.getOwnPropertyDescriptor(obj, 'p')
// Object { value: "a",
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
```

ES7有一个提案，提出了 `Object.getOwnPropertyDescriptors` 方法，返回指定对象所有自身属性（非继承属性）的描述对象。

```
const obj = {
  foo: 123,
  get bar() { return 'abc' }
};

Object.getOwnPropertyDescriptors(obj)
// { foo:
//   { value: 123,
//     writable: true,
//     enumerable: true,
//     configurable: true },
//   bar:
//   { get: [Function: bar],
//     set: undefined,
//     enumerable: true,
//     configurable: true } }
```

`Object.getOwnPropertyDescriptors` 方法返回一个对象，所有原对象的属性名都是该对象的属性名，对应的属性值就是该属性的描述对象。

该方法的实现非常容易。

```
function getOwnPropertyDescriptors(obj) {
  const result = {};
  for (let key of Reflect.ownKeys(obj)) {
    result[key] = Object.getOwnPropertyDescriptor(obj, key);
  }
  return result;
}
```

该方法的提出目的，主要是为了解决 `Object.assign()` 无法正确拷贝 `get` 属性和 `set` 属性的问题。

```
const source = {
  set foo(value) {
    console.log(value);
  }
};

const target1 = {};
Object.assign(target1, source);

Object.getOwnPropertyDescriptor(target1, 'foo')
// { value: undefined,
//   writable: true,
//   enumerable: true,
```

```
// configurable: true }
```

上面代码中，`source` 对象的 `foo` 属性的值是一个赋值函数，`Object.assign` 方法将这个属性拷贝给 `target1` 对象，结果该属性的值变成了 `undefined`。这是因为 `Object.assign` 方法总是拷贝一个属性的值，而不会拷贝它背后的赋值方法或取值方法。

这时，`Object.getOwnPropertyDescriptors` 方法配合 `Object.defineProperties` 方法，就可以实现正确拷贝。

```
const source = {
  set foo(value) {
    console.log(value);
  }
};

const target2 = {};
Object.defineProperties(target2, Object.getOwnPropertyDescriptors(source));
Object.getOwnPropertyDescriptor(target2, 'foo')
// { get: undefined,
//   set: [Function: foo],
//   enumerable: true,
//   configurable: true }
```

上面代码中，将两个对象合并的逻辑提炼出来，就是下面这样。

```
const shallowMerge = (target, source) => Object.defineProperties(
  target,
  Object.getOwnPropertyDescriptors(source)
);
```

`Object.getOwnPropertyDescriptors` 方法的另一个用处，是配合 `Object.create` 方法，将对象属性克隆到一个新对象。这属于浅拷贝。

```
const clone = Object.create(Object.getPrototypeOf(obj),
  Object.getOwnPropertyDescriptors(obj));

// 或者

const shallowClone = (obj) => Object.create(
  Object.getPrototypeOf(obj),
  Object.getOwnPropertyDescriptors(obj)
);
```

上面代码会克隆对象 `obj`。

另外，`Object.getOwnPropertyDescriptors` 方法可以实现，一个对象继承另一个对象。以前，继承另一个对象，常常写成下面这样。

```
const obj = {
  __proto__: prot,
  foo: 123,
};
```

ES6规定 `__proto__` 只有浏览器要部署，其他环境不用部署。如果去除 `__proto__`，上面代码就要改成下面这样。

```
const obj = Object.create(prot);
obj.foo = 123;

// 或者

const obj = Object.assign(
  Object.create(prot),
```

```
{  
  foo: 123,  
}  
);
```

有了 `Object.getOwnPropertyDescriptors`，我们就有了另一种写法。

```
const obj = Object.create(  
  prot,  
  Object.getOwnPropertyDescriptors({  
    foo: 123,  
  })  
)
```

`Object.getOwnPropertyDescriptors` 也可以用来实现Mixin（混入）模式。

```
let mix = (object) => ({  
  with: (...mixins) => mixins.reduce(  
    (c, mixin) => Object.create(  
      c, Object.getOwnPropertyDescriptors(mixin)  
    ), object  
)  
};  
  
// multiple mixins example  
let a = {a: 'a'};  
let b = {b: 'b'};  
let c = {c: 'c'};  
let d = mix(c).with(a, b);
```

上面代码中，对象 `a` 和 `b` 被混入了对象 `c`。

出于完整性的考虑，`Object.getOwnPropertyDescriptors` 进入标准以后，还会有 `Reflect.getOwnPropertyDescriptors` 方法。

Symbol

概述

ES5的对象属性名都是字符串，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（Mixin模式），新方法的名字就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，表示独一无二的值。它是JavaScript语言的第七种数据类型，前六种是：`Undefined`、`Null`、`布尔值(Boolean)`、`字符串(String)`、`数值(Number)`、`对象(Object)`。

Symbol值通过 `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值。`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。

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

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

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

```
// 没有参数的情况
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`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`值可以显式转为字符串。

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

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

另外，`Symbol`值也可以转为布尔值，但是不能转为数值。

```
var sym = Symbol();
Boolean(sym) // true
```

```
!sym // false
if (sym) {
  // ...
}

Number(sym) // TypeError
sym + 2 // TypeError
```

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]: 'Hello!'
};

// 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');
```

下面是另外一个例子。

```
const COLOR_RED = Symbol();  
const COLOR_GREEN = Symbol();  
  
function getComplement(color) {  
  switch (color) {  
    case COLOR_RED:  
      return COLOR_GREEN;  
    case COLOR_GREEN:  
      return COLOR_RED;  
    default:  
      throw new Error('Undefined color');  
  }  
}
```

常量使用Symbol值最大的好处，就是其他任何值都不可能有相同的值了，因此可以保证上面的 `switch` 语句会按设计的方式工作。

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。

实例：消除魔术字符串

魔术字符串指的是，在代码之中多次出现、与代码形成强耦合的某一个具体的字符串或者数值。风格良好的代码，应该尽量消除魔术字符串，该由含义清晰的变量代替。

```
function getArea(shape, options) {  
  var area = 0;  
  
  switch (shape) {  
    case 'Triangle': // 魔术字符串  
      area = .5 * options.width * options.height;  
      break;  
    /* ... more code ... */  
  }  
  
  return area;  
}  
  
getArea('Triangle', { width: 100, height: 100 }); // 魔术字符串
```

上面代码中，字符串“Triangle”就是一个魔术字符串。它多次出现，与代码形成“强耦合”，不利于将来的修改和维护。

常用的消除魔术字符串的方法，就是把它写成一个变量。

```
var shapeType = {  
  triangle: 'Triangle'  
};
```

```
function getArea(shape, options) {
  var area = 0;
  switch (shape) {
    case shapeType.triangle:
      area = .5 * options.width * options.height;
      break;
  }
  return area;
}

getArea(shapeType.triangle, { width: 100, height: 100 });
```

上面代码中，我们把“Triangle”写成 `shapeType` 对象的 `triangle` 属性，这样就消除了强耦合。

如果仔细分析，可以发现 `shapeType.triangle` 等于哪个值并不重要，只要确保不会跟其他 `shapeType` 属性的值冲突即可。因此，这里就很适合改用Symbol值。

```
const shapeType = {
  triangle: Symbol()
};
```

上面代码中，除了将 `shapeType.triangle` 的值设为一个Symbol，其他地方都不用修改。

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()`、`JSON.stringify()` 返回。但是，它也不是私有属性，有一个 `Object.getOwnPropertySymbols` 方法，可以获取指定对象的所有 Symbol 属性名。

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

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。

实例：模块的 Singleton 模式

`Singleton` 模式指的是调用一个类，任何时候返回的都是同一个实例。

对于 `Node` 来说，模块文件可以看成是一个类。怎么保证每次执行这个模块文件，返回的都是同一个实例呢？

很容易想到，可以把实例放到顶层对象 `global`。

```
// mod.js
function A() {
  this.foo = 'hello';
}

if (!global._foo) {
  global._foo = new A();
}

module.exports = global._foo;
```

然后，加载上面的 `mod.js`。

```
var a = require('./mod.js');
```

```
console.log(a.foo);
```

上面代码中，变量 `a` 任何时候加载的都是 `A` 的同一个实例。

但是，这里有一个问题，全局变量 `global._foo` 是可写的，任何文件都可以修改。

```
var a = require('./mod.js');
global._foo = 123;
```

上面的代码，会使得别的脚本加载 `mod.js` 都失真。

为了防止这种情况出现，我们就可以使用 `Symbol`。

```
// mod.js
const FOO_KEY = Symbol.for('foo');

function A() {
  this.foo = 'hello';
}

if (!global[FOO_KEY]) {
  global[FOO_KEY] = new A();
}

module.exports = global[FOO_KEY];
```

上面代码中，可以保证 `global[FOO_KEY]` 不会被无意间覆盖，但还是可以被改写。

```
var a = require('./mod.js');
global[Symbol.for('foo')] = 123;
```

如果键名使用 `Symbol` 方法生成，那么外部将无法引用这个值，当然也就无法改写。

```
// mod.js
const FOO_KEY = Symbol('foo');

// 后面代码相同 .....
```

上面代码将导致其他脚本都无法引用 `FOO_KEY`。但这样也有一个问题，就是如果多次执行这个脚本，每次得到的 `FOO_KEY` 都是不一样的。虽然 Node 会将脚本的执行结果缓存，一般情况下，不会多次执行同一个脚本，但是用户可以手动清除缓存，所以也不是完全可靠。

内置的 `Symbol` 值

除了定义自己使用的 `Symbol` 值以外，ES6 还提供了 11 个内置的 `Symbol` 值，指向语言内部使用的方法。

`Symbol.hasInstance`

对象的 `Symbol.hasInstance` 属性，指向一个内部方法。当其他对象使用 `instanceof` 运算符，判断是否为该对象的实例时，会调用这个方法。比如，`foo instanceof Foo` 在语言内部，实际调用的是 `Foo[Symbol.hasInstance](foo)`。

```
class MyClass {
  [Symbol.hasInstance](foo) {
    return foo instanceof Array;
  }
}

[1, 2, 3] instanceof new MyClass() // true
```

上面代码中，`MyClass` 是一个类，`new MyClass()` 会返回一个实例。该实例的 `Symbol.hasInstance` 方法，会在进行 `instanceof` 运算时自动调用，判断左侧的运算子是否为 `Array` 的实例。

下面是另一个例子。

```
class Even {
  static [Symbol.hasInstance](obj) {
    return Number(obj) % 2 === 0;
}
}

1 instanceof Even // false
2 instanceof Even // true
12345 instanceof Even // false
```

Symbol.isConcatSpreadable

对象的 `Symbol.isConcatSpreadable` 属性等于一个布尔值，表示该对象使用 `Array.prototype.concat()` 时，是否可以展开。

```
let arr1 = ['c', 'd'];
['a', 'b'].concat(arr1, 'e') // ['a', 'b', 'c', 'd', 'e']
arr1[Symbol.isConcatSpreadable] // undefined

let arr2 = ['c', 'd'];
arr2[Symbol.isConcatSpreadable] = false;
['a', 'b'].concat(arr2, 'e') // ['a', 'b', ['c', 'd'], 'e']
```

上面代码说明，数组的默认行为是可以展开。`Symbol.isConcatSpreadable` 属性等于 `true` 或 `undefined`，都有这个效果。

类似数组的对象也可以展开，但它的 `Symbol.isConcatSpreadable` 属性默认为 `false`，必须手动打开。

```
let obj = {length: 2, 0: 'c', 1: 'd'};
['a', 'b'].concat(obj, 'e') // ['a', 'b', obj, 'e']

obj[Symbol.isConcatSpreadable] = true;
['a', 'b'].concat(obj, 'e') // ['a', 'b', 'c', 'd', 'e']
```

对于一个类来说，`Symbol.isConcatSpreadable` 属性必须写成实例的属性。

```
class A1 extends Array {
  constructor(args) {
    super(args);
    this[Symbol.isConcatSpreadable] = true;
  }
}

class A2 extends Array {
  constructor(args) {
    super(args);
    this[Symbol.isConcatSpreadable] = false;
  }
}

let a1 = new A1();
a1[0] = 3;
a1[1] = 4;
let a2 = new A2();
a2[0] = 5;
a2[1] = 6;
[1, 2].concat(a1).concat(a2)
```

```
// [1, 2, 3, 4, [5, 6]]
```

上面代码中，类 A1 是可展开的，类 A2 是不可展开的，所以使用 concat 时有不一样的结果。

Symbol.species

对象的 `Symbol.species` 属性，指向当前对象的构造函数。创造实例时，默认会调用这个方法，即使用这个属性返回的函数当作构造函数，来创造新的实例对象。

```
class MyArray extends Array {  
  // 覆盖父类 Array 的构造函数  
  static get [Symbol.species]() { return Array; }  
}
```

上面代码中，子类 `MyArray` 继承了父类 `Array`。创建 `MyArray` 的实例对象时，本来会调用它自己的构造函数（本例中被省略了），但是由于定义了 `Symbol.species` 属性，所以会使用这个属性返回的的函数，创建 `MyArray` 的实例。

这个例子也说明，定义 `Symbol.species` 属性要采用 `get` 读取器。默认的 `Symbol.species` 属性等同于下面的写法。

```
static get [Symbol.species]() {  
  return this;  
}
```

下面是一个例子。

```
class MyArray extends Array {  
  static get [Symbol.species]() { return Array; }  
}  
var a = new MyArray(1,2,3);  
var mapped = a.map(x => x * x);  
  
mapped instanceof MyArray // false  
mapped instanceof Array // true
```

上面代码中，由于构造函数被替换成了 `Array`。所以，`mapped` 对象不是 `MyArray` 的实例，而是 `Array` 的实例。

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。

```
String.prototype.match(regexp)  
// 等同于  
regexp[Symbol.match](this)  
  
class MyMatcher {  
  [Symbol.match](string) {  
    return 'hello world'.indexOf(string);  
  }  
}  
  
'e'.match(new MyMatcher()) // 1
```

Symbol.replace

对象的 `Symbol.replace` 属性，指向一个方法，当该对象被 `String.prototype.replace` 方法调用时，会返回该方法的返回值。

```
String.prototype.replace(searchValue, replaceValue)
// 等同于
searchValue[Symbol.replace](this, replaceValue)
```

下面是一个例子。

```
const x = {};
x[Symbol.replace] = (...s) => console.log(s);

'Hello'.replace(x, 'World') // ["Hello", "World"]
```

`Symbol.replace` 方法会收到两个参数，第一个参数是 `replace` 方法正在作用的对象，上面例子是 `Hello`，第二个参数是替换后的值，上面例子是 `World`。

Symbol.search

对象的 `Symbol.search` 属性，指向一个方法，当该对象被 `String.prototype.search` 方法调用时，会返回该方法的返回值。

```
String.prototype.search(regexp)
// 等同于
regexp[Symbol.search](this)

class MySearch {
  constructor(value) {
    this.value = value;
  }
  [Symbol.search](string) {
    return string.indexOf(this.value);
  }
}
'foobar'.search(new MySearch('foo')) // 0
```

Symbol.split

对象的 `Symbol.split` 属性，指向一个方法，当该对象被 `String.prototype.split` 方法调用时，会返回该方法的返回值。

```
String.prototype.split(separator, limit)
// 等同于
separator[Symbol.split](this, limit)
```

下面是一个例子。

```
class MySplitter {
  constructor(value) {
    this.value = value;
  }
  [Symbol.split](string) {
    var index = string.indexOf(this.value);
    if (index === -1) {
      return string;
    }
    return [
      string.substr(0, index),
      string.substr(index + this.value.length)
    ];
  }
}
```

```
'foobar'.split(new MySplitter('foo'))
// ['', 'bar']

'foobar'.split(new MySplitter('bar'))
// ['foo', '']

'foobar'.split(new MySplitter('baz'))
// 'foobar'
```

上面方法使用 `Symbol.split` 方法，重新定义了字符串对象的 `split` 方法的行为，

Symbol.iterator

对象的 `Symbol.iterator` 属性，指向该对象的默认遍历器方法。

```
var myIterable = {};
myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};

[...myIterable] // [1, 2, 3]
```

对象进行 `for...of` 循环时，会调用 `Symbol.iterator` 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
```

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。

`Symbol.toPrimitive` 被调用时，会接受一个字符串参数，表示当前运算的模式，一共有三种模式。

- Number：该场合需要转成数值
- String：该场合需要转成字符串
- Default：该场合可以转成数值，也可以转成字符串

```
let obj = {
```

```

[Symbol.toPrimitive](hint) {
  switch (hint) {
    case 'number':
      return 123;
    case 'string':
      return 'str';
    case 'default':
      return 'default';
    default:
      throw new Error();
  }
}

2 * obj // 246
3 + obj // '3default'
obj == 'default' // true
String(obj) // 'str'

```

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。

```

// 例一
({[Symbol.toStringTag]: 'Foo'}.toString())
// "[object Foo]"

// 例二
class Collection {
  get [Symbol.toStringTag]() {
    return 'xxx';
  }
}
var x = new Collection();
Object.prototype.toString.call(x) // "[object xxx]"

```

ES6新增内置对象的 `Symbol.toStringTag` 属性值如下。

- `JSON[Symbol.toStringTag]` : 'JSON'
- `Math[Symbol.toStringTag]` : 'Math'
- `Module`对象 `M[Symbol.toStringTag]` : 'Module'
- `ArrayBuffer.prototype[Symbol.toStringTag]` : 'ArrayBuffer'
- `DataView.prototype[Symbol.toStringTag]` : 'DataView'
- `Map.prototype[Symbol.toStringTag]` : 'Map'
- `Promise.prototype[Symbol.toStringTag]` : 'Promise'
- `Set.prototype[Symbol.toStringTag]` : 'Set'
- `%TypedArray%.prototype[Symbol.toStringTag]` : 'Uint8Array'等
- `WeakMap.prototype[Symbol.toStringTag]` : 'WeakMap'
- `WeakSet.prototype[Symbol.toStringTag]` : 'WeakSet'
- `%MapIteratorPrototype%[Symbol.toStringTag]` : 'Map Iterator'
- `%SetIteratorPrototype%[Symbol.toStringTag]` : 'Set Iterator'
- `%StringIteratorPrototype%[Symbol.toStringTag]` : 'String Iterator'
- `Symbol.prototype[Symbol.toStringTag]` : 'Symbol'
- `Generator.prototype[Symbol.toStringTag]` : 'Generator'
- `GeneratorFunction.prototype[Symbol.toStringTag]` : 'GeneratorFunction'

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，哪些属性会被 `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
}
```

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 (let 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（或类似数组的对象）作为参数，用来初始化。

```
// 例一
var set = new Set([1, 2, 3, 4, 4]);
[...set]
// [1, 2, 3, 4]

// 例二
var items = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
items.size // 5

// 例三
function divs () {
  return [...document.querySelectorAll('div')];
}

var set = new Set(divs());
set.size // 56

// 类似于
divs().forEach(div => set.add(div));
set.size // 56
```

上面代码中，例一和例二都是 `Set` 函数接受数组作为参数，例三是接受类似数组的对象作为参数。

上面代码中，也展示了一种去除数组重复成员的方法。

```
// 去除数组的重复成员
[...new Set(array)]
```

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，使用的算法叫做“Same-value equality”，它类似于精确相等运算符（`==`），主要的区别是 `NaN` 等于自身，而精确相等运算符认为 `NaN` 不等于自身。

```
let set = new Set();
let a = NaN;
let b = NaN;
set.add(a);
set.add(b);
set // Set {NaN}
```

上面代码向Set实例添加了两个 `NaN`，但是只能加入一个。这表明，在Set内部，两个 `NaN` 是相等。

另外，两个对象总是不相等的。

```
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);
```

这就提供了去除数组重复成员的另一种方法。

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

需要特别指出的是，`Set` 的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用Set保存一个回调函数列表，调用时就能保证按照添加顺序调用。

(1) `keys()`, `values()`, `entries()`

`keys` 方法、`values` 方法、`entries` 方法返回的都是遍历器对象（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 `keys` 方法和 `values` 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，它的默认遍历器生成函数就是它的 `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
```

(2) `forEach()`

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。

(3) 遍历的应用

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

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

扩展运算符和Set结构相结合，就可以去除数组的重复成员。

```
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）、交集（Intersect）和差集（Difference）。

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

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

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

// 差集
let difference = new Set([...a].filter(x => !b.has(x)));
// Set {1}
```

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
ws.add(Symbol())
// TypeError: invalid value used in weak set
```

上面代码试图向WeakSet添加一个数值和 `Symbol` 值，结果报错，因为WeakSet只能放置对象。

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的参数。）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。

注意，是 `a` 数组的成员成为WeakSet的成员，而不是 `a` 数组本身。这意味着，数组的成员只能是对象。

```
var b = [3, 4];
var ws = new WeakSet(b);
// Uncaught TypeError: Invalid value used in weak set(...)
```

上面代码中，数组 `b` 的成员不是对象，加入WeakSet就会报错。

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，遍历机制无法保证成员的存在，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。

下面是WeakSet的另一个例子。

```
const foos = new WeakSet()
class Foo {
  constructor() {
    foos.add(this)
  }
  method () {
    if (!foos.has(this)) {
      throw new TypeError('Foo.prototype.method 只能在Foo的实例上调用!');
    }
  }
}
```

上面代码保证了 `Foo` 的实例方法，只能在 `Foo` 的实例上调用。这里使用WeakSet的好处是，`foos` 对实例的引用，不会被计入内存回收机制，所以删除实例的时候，不用考虑 `foos`，也不会出现内存泄漏。

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），但是传统上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。如果你需要“键值对”的数据结构，Map比Object更合适。

```
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`。

`Map`构造函数接受数组作为参数，实际上执行的是下面的算法。

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

下面的例子中，字符串 `true` 和布尔值 `true` 是两个不同的键。

```
var m = new Map([
  [true, 'foo'],
  ['true', 'bar']
]);

m.get(true) // 'foo'
m.get('true') // 'bar'
```

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')
  .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)
.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。

(1) `size` 属性

`size` 属性返回Map结构的成员总数。

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

map.size // 2
```

(2) `set(key, value)`

`set` 方法设置 `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。

```
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
```

`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');
```

(3) `get(key)`

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

```
var m = new Map();

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

m.get(hello) // Hello ES6!
```

(4) `has(key)`

`has` 方法返回一个布尔值，表示某个键是否在Map数据结构中。

```
var m = new Map();

m.set("edition", 6);
m.set(262, "standard");
m.set(undefined, "nah");

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

(5) `delete(key)`

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

```
var m = new Map();
m.set(undefined, "nah");
m.has(undefined) // true

m.delete(undefined)
m.has(undefined) // false
```

(6) `clear()`

`clear` 方法清除所有成员，没有返回值。

```
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生成函数和一个遍历方法。

- `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。
- `forEach()`：遍历Map的所有成员。

需要特别注意的是，Map的遍历顺序就是插入顺序。

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`。

与其他数据结构的互相转换

(1) Map转为数组

前面已经提过，Map转为数组最方便的方法，就是使用扩展运算符 (...)。

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

(2) 数组转为Map

将数组转入Map构造函数，就可以转为Map。

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

(3) Map转为对象

如果所有Map的键都是字符串，它可以转为对象。

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

(4) 对象转为Map

```
function objToStrMap(obj) {  
  let strMap = new Map();  
  for (let k of Object.keys(obj)) {  
    strMap.set(k, obj[k]);  
  }  
  return strMap;  
}  
  
objToStrMap({yes: true, no: false})  
// [ [ 'yes', true ], [ 'no', false ] ]
```

(5) Map转为JSON

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

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

另一种情况是，Map的键名有非字符串，这时可以选择转为数组JSON。

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

(6) JSON转为Map

JSON转为Map，正常情况下，所有键名都是字符串。

```
function jsonToStrMap(jsonStr) {
```

```
    return objToStrMap(JSON.parse(jsonStr));
}

jsonToStrMap('{"yes":true,"no":false}')
// Map {'yes' => true, 'no' => false}
```

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

```
function jsonToMap(jsonStr) {
    return new Map(JSON.parse(jsonStr));
}

jsonToMap('[[true,7],[{"foo":3},["abc"]]]')
// Map {true => 7, Object {foo: 3} => ['abc']}
```

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），不接受其他类型的值作为键名，而且键名所指向的对象，不计入垃圾回收机制。

```
var map = new WeakMap()
map.set(1, 2)
// TypeError: 1 is not an object!
map.set(Symbol(), 2)
// TypeError: Invalid value used as weak map key
```

上面代码中，如果将 `1` 和 `Symbol` 作为`WeakMap`的键名，都会报错。

`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++;
}, 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。

Proxy

概述

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

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

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

```
    return Reflect.get(target, key, receiver);
},
set: function (target, key, value, receiver) {
  console.log(`setting ${key}!`);
  return Reflect.set(target, key, value, receiver);
}
});
```

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

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

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

ES6 原生提供 `Proxy` 构造函数，用来生成 `Proxy` 实例。

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

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

下面是另一个拦截读取属性行为的例子。

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

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

上面代码中，作为构造函数，`Proxy` 接受两个参数。第一个参数是所要代理的目标对象（上例是一个空对象），即如果没有 `Proxy` 的介入，操作原来要访问的就是这个对象；第二个参数是一个配置对象，对于每一个被代理的操作，需要提供一个对应的处理函数，该函数将拦截对应的操作。比如，上面代码中，配置对象有一个 `get` 方法，用来拦截对目标对象属性的访问请求。`get` 方法的两个参数分别是目标对象和所要访问的属性。可以看到，由于拦截函数总是返回 `35`，所以访问任何属性都得到 `35`。

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

如果 `handler` 没有设置任何拦截，那就等同于直接通向原对象。

```
var target = {};
var handler = {};
var proxy = new Proxy(target, handler);
proxy.a = 'b';
target.a // "b"
```

上面代码中，`handler` 是一个空对象，没有任何拦截效果，访问 `handler` 就等同于访问 `target`。

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

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

Proxy 实例也可以作为其他对象的原型对象。

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

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

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

同一个拦截器函数，可以设置拦截多个操作。

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

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

fproxy(1, 2) // 1
new fproxy(1,2) // {value: 2}
fproxy.prototype === Object.prototype // true
fproxy.foo // "Hello, foo"
```

下面是 Proxy 支持的拦截操作一览。

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

(1) `get(target, propKey, receiver)`

拦截对象属性的读取，比如 `proxy.foo` 和 `proxy['foo']`。

最后一个参数 `receiver` 是一个对象，可选，参见下面 `Reflect.get` 的部分。

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

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

(3) `has(target, propKey)`

拦截 `propKey in proxy` 的操作，返回一个布尔值。

(4) `deleteProperty(target, propKey)`

拦截 `delete proxy[propKey]` 的操作，返回一个布尔值。

(5) `ownKeys(target)`

拦截 `Object.getOwnPropertyNames(proxy)`、`Object.getOwnPropertySymbols(proxy)`、`Object.keys(proxy)`，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而 `Object.keys()` 的返回结果仅包括目标对象自身的可遍历属性。

(6) `getOwnPropertyDescriptor(target, propKey)`

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

(7) `defineProperty(target, propKey, propDesc)`

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

(8) `preventExtensions(target)`

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

(9) `getPrototypeOf(target)`

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

(10) `isExtensible(target)`

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

(11) `setPrototypeOf(target, proto)`

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

如果目标对象是函数，那么还有两种额外操作可以拦截。

(12) `apply(target, object, args)`

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

(13) `construct(target, args)`

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

Proxy 实例的方法

下面是上面这些拦截方法的详细介绍。

get()

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

```
var person = {
  name: "张三"
};

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

```
    }
  });
};

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

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

`get` 方法可以继承。

```
let proto = new Proxy({}, {
  get(target, propertyKey, receiver) {
    console.log('GET '+propertyKey);
    return target[propertyKey];
  }
});

let obj = Object.create(proto);
obj.xxx // "GET xxx"
```

上面代码中，拦截操作定义在 `Prototype` 对象上面，所以如果读取 `obj` 对象继承的属性时，拦截会生效。

下面的例子使用 `get` 拦截，实现数组读取负数的索引。

```
function createArray(...elements) {
  let handler = {
    get(target, propKey, receiver) {
      let index = Number(propKey);
      if (index < 0) {
        propKey = String(target.length + index);
      }
      return Reflect.get(target, propKey, receiver);
    }
  };

  let target = [];
  target.push(...elements);
  return new Proxy(target, handler);
}

let arr = createArray('a', 'b', 'c');
arr[-1] // c
```

上面代码中，数组的位置参数是 `-1`，就会输出数组的倒数最后一个成员。

利用 `Proxy`，可以将读取属性的操作（`get`），转变为执行某个函数，从而实现属性的链式操作。

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

```

        return oproxy;
    }
}());

var double = n => n * 2;
var pow    = n => n * n;
var reverseInt = n => 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。

下面的例子则是利用 `get` 拦截，实现一个生成各种DOM节点的通用函数 `dom`。

```

const dom = new Proxy({}, {
  get(target, property) {
    return function(attrs = {}, ...children) {
      const el = document.createElement(property);
      for (let prop of Object.keys(attrs)) {
        el.setAttribute(prop, attrs[prop]);
      }
      for (let child of children) {
        if (typeof child === 'string') {
          child = document.createTextNode(child);
        }
        el.appendChild(child);
      }
      return el;
    }
  }
});

const el = dom.div({
  'Hello, my name is ',
  dom.a({href: '//example.com'}, 'Mark'),
  '. I like:',
  dom.ul([
    dom.li({}, 'The web'),
    dom.li({}, 'Food'),
    dom.li({}, '...actually that\'s it')
  ])
);

document.body.appendChild(el);

```

如果一个属性不可配置 (`configurable`) 和不可写 (`writable`)，则该属性不能被代理，通过 `Proxy` 对象访问该属性会报错。

```

const target = Object.defineProperties({}, {
  foo: {
    value: 123,
    writable: false,
    configurable: false
  },
});

const handler = {
  get(target, propKey) {
    return 'abc';
  }
};

const proxy = new Proxy(target, handler);

```

```
proxy.foo  
// TypeError: Invariant check failed
```

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` 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，这是数据验证的一种实现方法。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。

有时，我们会在对象上面设置内部属性，属性名的第一个字符使用下划线开头，表示这些属性不应该被外部使用。结合 `get` 和 `set` 方法，就可以做到防止这些内部属性被外部读写。

```
var handler = {  
  get (target, key) {  
    invariant(key, 'get');  
    return target[key];  
  },  
  set (target, key, value) {  
    invariant(key, 'set');  
    target[key] = value;  
    return true;  
  }  
};  
function invariant (key, action) {  
  if (key[0] === '_') {  
    throw new Error(`Invalid attempt to ${action} private "${key}" property`);  
  }  
}  
var target = {};  
var proxy = new Proxy(target, handler);  
proxy._prop  
// Error: Invalid attempt to get private "_prop" property  
proxy._prop = 'c'
```

```
// Error: Invalid attempt to set private "_prop" property
```

上面代码中，只要读写的属性名的第一个字符是下划线，一律抛错，从而达到禁止读写内部属性的目的。

注意，如果目标对象自身的某个属性，不可写也不可配置，那么 `set` 不得改变这个属性的值，只能返回同样的值，否则报错。

apply()

`apply` 方法拦截函数的调用、`call` 和 `apply` 操作。

`apply` 方法可以接受三个参数，分别是目标对象、目标对象的上下文对象（`this`）和目标对象的参数数组。

```
var handler = {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments);
  }
};
```

下面是一个例子。

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

var p = new Proxy(target, handler);

p()
// "I am the proxy"
```

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。

下面是另外一个例子。

```
var twice = {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments) * 2;
  }
};
function sum (left, right) {
  return left + right;
};
var proxy = new Proxy(sum, twice);
proxy(1, 2) // 6
proxy.call(null, 5, 6) // 22
proxy.apply(null, [7, 8]) // 30
```

上面代码中，每当执行 `proxy` 函数（直接调用或 `call` 和 `apply` 调用），就会被 `apply` 方法拦截。

另外，直接调用 `Reflect.apply` 方法，也会被拦截。

```
Reflect.apply(proxy, null, [9, 10]) // 38
```

has()

`has` 方法用来拦截 `HasProperty` 操作，即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是 `in` 运算符。

下面的例子使用 `has` 方法隐藏某些属性，不被 `in` 运算符发现。

```
var handler = {
  has (target, key) {
    if (key[0] === '_') {
      return false;
    }
    return key in target;
  }
};
var target = { _prop: 'foo', prop: 'foo' };
var proxy = new Proxy(target, handler);
'_prop' in proxy // false
```

上面代码中，如果原对象的属性名的第一个字符是下划线，`proxy.has` 就会返回 `false`，从而不会被 `in` 运算符发现。

如果原对象不可配置或者禁止扩展，这时 `has` 拦截会报错。

```
var obj = { a: 10 };
Object.preventExtensions(obj);

var p = new Proxy(obj, {
  has: function(target, prop) {
    return false;
  }
});
'a' in p // TypeError is thrown
```

上面代码中，`obj` 对象禁止扩展，结果使用 `has` 拦截就会报错。也就是说，如果某个属性不可配置（或者目标对象不可扩展），则 `has` 方法就不得“隐藏”（即返回 `false`）目标对象的该属性。

值得注意的是，`has` 方法拦截的是 `HasProperty` 操作，而不是 `HasOwnProperty` 操作，即 `has` 方法不判断一个属性是对对象自身的属性，还是继承的属性。

另外，虽然 `for...in` 循环也用到了 `in` 运算符，但是 `has` 拦截对 `for...in` 循环不生效。

```
let stu1 = {name: '张三', score: 59};
let stu2 = {name: '李四', score: 99};

let handler = {
  has(target, prop) {
    if (prop === 'score' && target[prop] < 60) {
      console.log(`"${target.name}" 不及格`);
      return false;
    }
    return prop in target;
  }
}

let oproxy1 = new Proxy(stu1, handler);
let oproxy2 = new Proxy(stu2, handler);

'score' in oproxy1
// 张三 不及格
// false

'score' in oproxy2
// true

for (let a in oproxy1) {
  console.log(oproxy1[a]);
```

```
}

// 张三
// 59

for (let b in oproxy2) {
    console.log(oproxy2[b]);
}

// 李四
// 99
```

上面代码中，`has` 拦截只对 `in` 循环生效，对 `for...in` 循环不生效，导致不符合要求的属性没有被排除在 `for...in` 循环之外。

construct()

`construct` 方法用于拦截 `new` 命令，下面是拦截对象的写法。

```
var handler = {
    construct (target, args, newTarget) {
        return new target(...args);
    }
};
```

`construct` 方法可以接受两个参数。

- `target`：目标对象
- `args`：构建函数的参数对象

下面是一个例子。

```
var p = new Proxy(function () {}, {
    construct: function(target, args) {
        console.log('called: ' + args.join(', '));
        return { value: args[0] * 10 };
    }
});

(new p(1)).value
// "called: 1"
// 10
```

`construct` 方法返回的必须是一个对象，否则会报错。

```
var p = new Proxy(function() {}, {
    construct: function(target, argumentsList) {
        return 1;
    }
});

new p() // 报错
```

deleteProperty()

`deleteProperty` 方法用于拦截 `delete` 操作，如果这个方法抛出错误或者返回 `false`，当前属性就无法被 `delete` 命令删除。

```
var handler = {
    deleteProperty (target, key) {
        invariant(key, 'delete');
        return true;
    }
};
```

```
}

function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" property`);
  }
}

var target = { _prop: 'foo' };
var proxy = new Proxy(target, handler);
delete proxy._prop
// Error: Invalid attempt to delete private "_prop" property
```

上面代码中，`deleteProperty` 方法拦截了 `delete` 操作符，删除第一个字符为下划线的属性会报错。

注意，目标对象自身的不可配置（configurable）的属性，不能被 `deleteProperty` 方法删除，否则报错。

defineProperty()

`defineProperty` 方法拦截了 `Object.defineProperty` 操作。

```
var handler = {
  defineProperty (target, key, descriptor) {
    return false;
  }
};
var target = {};
var proxy = new Proxy(target, handler);
proxy.foo = 'bar'
// TypeError: proxy defineProperty handler returned false for property '"foo"'
```

上面代码中，`defineProperty` 方法返回 `false`，导致添加新属性会抛出错误。

注意，如果目标对象不可扩展（extensible），则 `defineProperty` 不能增加目标对象上不存在的属性，否则会报错。另外，如果目标对象的某个属性不可写（writable）或不可配置（configurable），则 `defineProperty` 方法不得改变这两个设置。

getOwnPropertyDescriptor()

`getOwnPropertyDescriptor` 方法拦截 `Object.getOwnPropertyDescriptor`，返回一个属性描述对象或者 `undefined`。

```
var handler = {
  getOwnPropertyDescriptor (target, key) {
    if (key[0] === '_') {
      return;
    }
    return Object.getOwnPropertyDescriptor(target, key);
  }
};
var target = { _foo: 'bar', baz: 'tar' };
var proxy = new Proxy(target, handler);
Object.getOwnPropertyDescriptor(proxy, 'wat')
// undefined
Object.getOwnPropertyDescriptor(proxy, '_foo')
// undefined
Object.getOwnPropertyDescriptor(proxy, 'baz')
// { value: 'tar', writable: true, enumerable: true, configurable: true }
```

上面代码中，`handler.getOwnPropertyDescriptor` 方法对于第一个字符为下划线的属性名会返回 `undefined`。

getPrototypeOf()

`getPrototypeOf` 方法主要用来拦截 `Object.getPrototypeOf()` 运算符，以及其他一些操作。

- `Object.prototype.__proto__`
- `Object.prototype.isPrototypeOf()`
- `Object.getPrototypeOf()`
- `Reflect.getPrototypeOf()`
- `instanceof` 运算符

下面是一个例子。

```
var proto = {};
var p = new Proxy({}, {
  getPrototypeOf(target) {
    return proto;
  }
});
Object.getPrototypeOf(p) === proto // true
```

上面代码中，`getPrototypeOf` 方法拦截 `Object.getPrototypeOf()`，返回 `proto` 对象。

注意，`getPrototypeOf` 方法的返回值必须是对象或者 `null`，否则报错。另外，如果目标对象不可扩展（extensible），`getPrototypeOf` 方法必须返回目标对象的原型对象。

isExtensible()

`isExtensible` 方法拦截 `Object.isExtensible` 操作。

```
var p = new Proxy({}, {
  isExtensible: function(target) {
    console.log("called");
    return true;
  }
});

Object.isExtensible(p)
// "called"
// true
```

上面代码设置了 `isExtensible` 方法，在调用 `Object.isExtensible` 时会输出 `called`。

注意，该方法只能返回布尔值，否则返回值会被自动转为布尔值。

这个方法有一个强限制，它的返回值必须与目标对象的 `isExtensible` 属性保持一致，否则就会抛出错误。

```
Object.isExtensible(proxy) === Object.isExtensible(target)
```

下面是一个例子。

```
var p = new Proxy({}, {
  isExtensible: function(target) {
    return false;
  }
});

Object.isExtensible(p) // 报错
```

ownKeys()

`ownKeys` 方法用来拦截以下操作。

- `Object.getOwnPropertyNames()`
- `Object.getOwnPropertySymbols()`
- `Object.keys()`

下面是拦截 `Object.keys()` 的例子。

```
let target = {
  a: 1,
  b: 2,
  c: 3
};

let handler = {
  ownKeys(target) {
    return ['a'];
  }
};

let proxy = new Proxy(target, handler);

Object.keys(proxy)
// [ 'a' ]
```

上面代码拦截了对于 `target` 对象的 `Object.keys()` 操作，只返回 `a`、`b`、`c` 三个属性之中的 `a` 属性。

下面的例子是拦截第一个字符为下划线的属性名。

```
let target = {
  _bar: 'foo',
  _prop: 'bar',
  prop: 'baz'
};

let handler = {
  ownKeys (target) {
    return Reflect.ownKeys(target).filter(key => key[0] !== '_');
  }
};

let proxy = new Proxy(target, handler);
for (let key of Object.keys(proxy)) {
  console.log(target[key]);
}
// "baz"
```

注意，使用 `Object.keys` 方法时，有三类属性会被 `ownKeys` 方法自动过滤，不会返回。

- 目标对象上不存在的属性
- 属性名为 `Symbol` 值
- 不可遍历（`enumerable`）的属性

```
let target = {
  a: 1,
  b: 2,
  c: 3,
  [Symbol.for('secret')]: '4',
};
```

```
Object.defineProperty(target, 'key', {
  enumerable: false,
  configurable: true,
  writable: true,
  value: 'static'
});

let handler = {
  ownKeys(target) {
    return ['a', 'd', Symbol.for('secret'), 'key'];
  }
};

let proxy = new Proxy(target, handler);

Object.keys(proxy)
// ['a']
```

上面代码中，`ownKeys` 方法之中，显式返回不存在的属性（`d`）、`Symbol` 值（`Symbol.for('secret')`）`、不可遍历的属性（key），结果都被自动过滤掉。`

`ownKeys` 方法还可以拦截 `Object.getOwnPropertyNames()`。

```
var p = new Proxy({}, {
  ownKeys: function(target) {
    return ['a', 'b', 'c'];
  }
});

Object.getOwnPropertyNames(p)
// ['a', 'b', 'c']
```

`ownKeys` 方法返回的数组成员，只能是字符串或 `Symbol` 值。如果有其他类型的值，或者返回的根本不是数组，就会报错。

```
var obj = {};

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return [123, true, undefined, null, {}, []];
  }
});

Object.getOwnPropertyNames(p)
// Uncaught TypeError: 123 is not a valid property name
```

上面代码中，`ownKeys` 方法虽然返回一个数组，但是每一个数组成员都不是字符串或 `Symbol` 值，因此就报错了。

如果目标对象自身包含不可配置的属性，则该属性必须被 `ownKeys` 方法返回，否则报错。

```
var obj = {};
Object.defineProperty(obj, 'a', {
  configurable: false,
  enumerable: true,
  value: 10
});

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return ['b'];
  }
});
```

```
Object.getOwnPropertyNames(p)
// Uncaught TypeError: 'ownKeys' on proxy: trap result did not include 'a'
```

上面代码中，`obj` 对象的 `a` 属性是不可配置的，这时 `ownKeys` 方法返回的数组之中，必须包含 `a`，否则会报错。

另外，如果目标对象是不可扩展的（non-extensibility），这时 `ownKeys` 方法返回的数组之中，必须包含原对象的所有属性，且不能包含多余的属性，否则报错。

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

Object.preventExtensions(obj);

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return ['a', 'b'];
  }
});

Object.getOwnPropertyNames(p)
// Uncaught TypeError: 'ownKeys' on proxy: trap returned extra keys but proxy target
is non-extensible
```

上面代码中，`Obj` 对象是不可扩展的，这时 `ownKeys` 方法返回的数组之中，包含了 `obj` 对象的多余属性 `b`，所以导致了报错。

preventExtensions()

`preventExtensions` 方法拦截 `Object.preventExtensions()`。该方法必须返回一个布尔值，否则会被自动转为布尔值。

这个方法有一个限制，只有目标对象不可扩展时（即 `Object.isExtensible(proxy)` 为 `false`），`proxy.preventExtensions` 才能返回 `true`，否则会报错。

```
var p = new Proxy({}, {
  preventExtensions: function(target) {
    return true;
  }
});

Object.preventExtensions(p) // 报错
```

上面代码中，`proxy.preventExtensions` 方法返回 `true`，但这时 `Object.isExtensible(proxy)` 会返回 `true`，因此报错。

为了防止出现这个问题，通常要在 `proxy.preventExtensions` 方法里面，调用一次 `Object.preventExtensions`。

```
var p = new Proxy({}, {
  preventExtensions: function(target) {
    console.log('called');
    Object.preventExtensions(target);
    return true;
  }
});

Object.preventExtensions(p)
// "called"
// true
```

setPrototypeOf()

`setPrototypeOf` 方法主要用来拦截 `Object.setPrototypeOf` 方法。

下面是一个例子。

```
var handler = {
  setPrototypeOf(target, proto) {
    throw new Error('Changing the prototype is forbidden');
  }
};
var proto = {};
var target = function () {};
var proxy = new Proxy(target, handler);
proxy.setPrototypeOf(proxy, proto);
// Error: Changing the prototype is forbidden
```

上面代码中，只要修改 `target` 的原型对象，就会报错。

注意，该方法只能返回布尔值，否则会被自动转为布尔值。另外，如果目标对象不可扩展（extensible），`setPrototypeOf` 方法不得改变目标对象的原型。

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。

`Proxy.revocable` 的一个使用场景是，目标对象不允许直接访问，必须通过代理访问，一旦访问结束，就收回代理权，不允许再次访问。

this 问题

虽然 `Proxy` 可以代理针对目标对象的访问，但它不是目标对象的透明代理，即不做任何拦截的情况下，也无法保证与目标对象的行为一致。主要原因就是在 `Proxy` 代理的情况下，目标对象内部的 `this` 关键字会指向 `Proxy` 代理。

```
const target = {
  m: function () {
    console.log(this === proxy);
  }
};
const handler = {};

const proxy = new Proxy(target, handler);

target.m() // false
proxy.m() // true
```

上面代码中，一旦 `proxy` 代理 `target.m`，后者内部的 `this` 就是指向 `proxy`，而不是 `target`。

下面是一个例子，由于 `this` 指向的变化，导致 `Proxy` 无法代理目标对象。

```
const _name = new WeakMap();

class Person {
  constructor(name) {
    _name.set(this, name);
  }
  get name() {
    return _name.get(this);
  }
}

const jane = new Person('Jane');
jane.name // 'Jane'

const proxy = new Proxy(jane, {});
proxy.name // undefined
```

上面代码中，目标对象 `jane` 的 `name` 属性，实际保存在外部 `WeakMap` 对象 `_name` 上面，通过 `this` 键区分。由于通过 `proxy.name` 访问时，`this` 指向 `proxy`，导致无法取到值，所以返回 `undefined`。

此外，有些原生对象的内部属性，只有通过正确的 `this` 才能拿到，所以 `Proxy` 也无法代理这些原生对象的属性。

```
const target = new Date();
const handler = {};
const proxy = new Proxy(target, handler);

proxy.getDate();
// TypeError: this is not a Date object.
```

上面代码中，`getDate` 方法只能在 `Date` 对象实例上面拿到，如果 `this` 不是 `Date` 对象实例就会报错。这时，`this` 绑定原始对象，就可以解决这个问题。

```
const target = new Date('2015-01-01');
const handler = {
  get(target, prop) {
    if (prop === 'getDate') {
      return target.getDate.bind(target);
    }
    return Reflect.get(target, prop);
  }
};
const proxy = new Proxy(target, handler);

proxy.getDate() // 1
```

实例：Web 服务的客户端

`Proxy` 对象可以拦截目标对象的任意属性，这使得它很合适用来写 Web 服务的客户端。

```
const service = createWebService('http://example.com/data');

service.employees().then(json => {
  const employees = JSON.parse(json);
  // ...
});
```

上面代码新建了一个 Web 服务的接口，这个接口返回各种数据。`Proxy` 可以拦截这个对象的任意属性，所以不用为每

一种数据写一个适配方法，只要写一个 `Proxy` 拦截就可以了。

```
function createWebService(baseUrl) {
  return new Proxy({}, {
    get(target, propKey, receiver) {
      return () => httpGet(baseUrl + '/' + propKey);
    }
  });
}
```

同理，`Proxy` 也可以用来实现数据库的 ORM 层。

Reflect

概述

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

(1) 将 `Object` 对象的一些明显属于语言内部的方法（比如 `Object.defineProperty`），放到 `Reflect` 对象上。现阶段，某些方法同时在 `Object` 和 `Reflect` 对象上部署，未来的新方法将只部署在 `Reflect` 对象上。也就是说，从 `Reflect` 对象上可以拿到语言内部的方法。

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

```
// 老写法
try {
  Object.defineProperty(target, property, attributes);
  // success
} catch (e) {
  // failure
}

// 新写法
if (Reflect.defineProperty(target, property, attributes)) {
  // success
} else {
  // failure
}
```

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

```
// 老写法
'assign' in Object // true

// 新写法
Reflect.has(Object, 'assign') // true
```

(4) `Reflect` 对象的方法与 `Proxy` 对象的方法一一对应，只要是 `Proxy` 对象的方法，就能在 `Reflect` 对象上找到对应的方法。这就让 `Proxy` 对象可以方便地调用对应的 `Reflect` 方法，完成默认行为，作为修改行为的基础。也就是说，不管 `Proxy` 怎么修改默认行为，你总可以在 `Reflect` 上获取默认行为。

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

```
        return success;
    }
});
```

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

下面是另一个例子。

```
var loggedObj = new Proxy(obj, {
  get(target, name) {
    console.log('get', target, name);
    return Reflect.get(target, name);
  },
  deleteProperty(target, name) {
    console.log('delete' + name);
    return Reflect.deleteProperty(target, name);
  },
  has(target, name) {
    console.log('has' + name);
    return Reflect.has(target, name);
  }
});
```

上面代码中，每一个 `Proxy` 对象的拦截操作（`get`、`delete`、`has`），内部都调用对应的 `Reflect` 方法，保证原生行为能够正常执行。添加的工作，就是将每一个操作输出一行日志。

有了 `Reflect` 对象以后，很多操作会更易读。

```
// 老写法
Function.prototype.apply.call(Math.floor, undefined, [1.75]) // 1

// 新写法
Reflect.apply(Math.floor, undefined, [1.75]) // 1
```

静态方法

`Reflect` 对象一共有13个静态方法。

- `Reflect.apply(target,thisArg,args)`
- `Reflect.construct(target,args)`
- `Reflect.get(target,name,receiver)`
- `Reflect.set(target,name,value,receiver)`
- `Reflect.defineProperty(target,name,desc)`
- `Reflect.deleteProperty(target,name)`
- `Reflect.has(target,name)`
- `Reflect.ownKeys(target)`
- `Reflect.isExtensible(target)`
- `Reflect.preventExtensions(target)`
- `Reflect.getOwnPropertyDescriptor(target, name)`
- `Reflect.getPrototypeOf(target)`
- `Reflect.setPrototypeOf(target, prototype)`

上面这些方法的作用，大部分与 `Object` 对象的同名方法的作用都是相同的，而且它与 `Proxy` 对象的方法是一一对应的。下面是对它们的解释。

Reflect.get(target, name, receiver)

`Reflect.get` 方法查找并返回 `target` 对象的 `name` 属性，如果没有该属性，则返回 `undefined`。

```
var myObject = {
  foo: 1,
  bar: 2,
  get baz() {
    return this.foo + this.bar;
  },
}

Reflect.get(myObject, 'foo') // 1
Reflect.get(myObject, 'bar') // 2
Reflect.get(myObject, 'baz') // 3
```

如果 `name` 属性部署了读取函数（getter），则读取函数的 `this` 绑定 `receiver`。

```
var myObject = {
  foo: 1,
  bar: 2,
  get baz() {
    return this.foo + this.bar;
  },
};

var myReceiverObject = {
  foo: 4,
  bar: 4,
};

Reflect.get(myObject, 'baz', myReceiverObject) // 8
```

如果第一个参数不是对象，`Reflect.get` 方法会报错。

```
Reflect.get(1, 'foo') // 报错
Reflect.get(false, 'foo') // 报错
```

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

`Reflect.set` 方法设置 `target` 对象的 `name` 属性等于 `value`。

```
var myObject = {
  foo: 1,
  set bar(value) {
    return this.foo = value;
  },
}

myObject.foo // 1

Reflect.set(myObject, 'foo', 2);
myObject.foo // 2

Reflect.set(myObject, 'bar', 3)
myObject.foo // 3
```

如果 `name` 属性设置了赋值函数，则赋值函数的 `this` 绑定 `receiver`。

```
var myObject = {
  foo: 4,
  set bar(value) {
    return this.foo = value;
}
```

```
},
};

var myReceiverObject = {
  foo: 0,
};

Reflect.set(myObject, 'bar', 1, myReceiverObject);
myObject.foo // 4
myReceiverObject.foo // 1
```

如果第一个参数不是对象，`Reflect.set` 会报错。

```
Reflect.set(1, 'foo', {}) // 报错
Reflect.set(false, 'foo', {}) // 报错
```

Reflect.has(obj, name)

`Reflect.has` 方法对应 `name in obj` 里面的 `in` 运算符。

```
var myObject = {
  foo: 1,
};

// 旧写法
'foo' in myObject // true

// 新写法
Reflect.has(myObject, 'foo') // true
```

如果第一个参数不是对象，`Reflect.has` 和 `in` 运算符都会报错。

Reflect.deleteProperty(obj, name)

`Reflect.deleteProperty` 方法等同于 `delete obj[name]`，用于删除对象的属性。

```
const myObj = { foo: 'bar' };

// 旧写法
delete myObj.foo;

// 新写法
Reflect.deleteProperty(myObj, 'foo');
```

该方法返回一个布尔值。如果删除成功，或者被删除的属性不存在，返回 `true`；删除失败，被删除的属性依然存在，返回 `false`。

Reflect.construct(target, args)

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

```
function Greeting(name) {
  this.name = name;
}

// new 的写法
const instance = new Greeting('张三');

// Reflect.construct 的写法
```

```
const instance = Reflect.construct(Greeting, ['张三']);
```

Reflect.getPrototypeOf(obj)

`Reflect.getPrototypeOf` 方法用于读取对象的 `__proto__` 属性，对应 `Object.getPrototypeOf(obj)`。

```
const myObj = new FancyThing();

// 旧写法
Object.getPrototypeOf(myObj) === FancyThing.prototype;

// 新写法
Reflect.getPrototypeOf(myObj) === FancyThing.prototype;
```

`Reflect.getPrototypeOf` 和 `Object.getPrototypeOf` 的一个区别是，如果第一个参数不是对象（包括 `null` 和 `undefined`），`Object.getPrototypeOf` 会将这个参数转为对象，然后再运行，而 `Reflect.getPrototypeOf` 会报错。

```
Object.getPrototypeOf(1) // undefined
Reflect.getPrototypeOf(1) // 报错
```

Reflect.setPrototypeOf(obj, newProto)

`Reflect.setPrototypeOf` 方法用于设置对象的 `__proto__` 属性，对应 `Object.setPrototypeOf(obj, newProto)`。

```
const myObj = new FancyThing();

// 旧写法
Object.setPrototypeOf(myObj, OtherThing.prototype);

// 新写法
Reflect.setPrototypeOf(myObj, OtherThing.prototype);
```

如果第一个参数不是对象，`Reflect.setPrototypeOf` 和 `Object.setPrototypeOf` 都会报错。

```
Object.setPrototypeOf(1) // 报错
Reflect.setPrototypeOf(1) // 报错
```

Reflect.apply(func, thisArg, args)

`Reflect.apply` 方法等同于 `Function.prototype.apply.call(func, thisArg, args)`，用于绑定 `this` 对象后执行给定函数。

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

```
const ages = [11, 33, 12, 54, 18, 96];

// 旧写法
const youngest = Math.min.apply(Math, ages);
const oldest = Math.max.apply(Math, ages);
const type = Object.prototype.toString.call(youngest);

// 新写法
const youngest = Reflect.apply(Math.min, Math, ages);
const oldest = Reflect.apply(Math.max, Math, ages);
```

```
const type = Reflect.apply(Object.prototype.toString, youngest);
```

Reflect.defineProperty(target, propertyKey, attributes)

`Reflect.defineProperty` 方法基本等同于 `Object.defineProperty`，用来为对象定义属性。未来，后者会被逐渐废除，请从现在开始就使用 `Reflect.defineProperty` 代替它。

```
function MyDate() {
  /*...*/
}

// 旧写法
Object.defineProperty(MyDate, 'now', {
  value: () => new Date.now()
});

// 新写法
Reflect.defineProperty(MyDate, 'now', {
  value: () => new Date.now()
});
```

如果 `Reflect.defineProperty` 的第一个参数不是对象，就会抛出错误，比如

```
Reflect.defineProperty(1, 'foo').
```

Reflect.getOwnPropertyDescriptor(target, propertyKey)

`Reflect.getOwnPropertyDescriptor` 基本等同于 `Object.getOwnPropertyDescriptor`，用于得到指定属性的描述对象，将来会替代掉后者。

```
var myObject = {};
Object.defineProperty(myObject, 'hidden', {
  value: true,
  enumerable: false,
});

// 旧写法
var theDescriptor = Object.getOwnPropertyDescriptor(myObject, 'hidden');

// 新写法
var theDescriptor = Reflect.getOwnPropertyDescriptor(myObject, 'hidden');
```

`Reflect.getOwnPropertyDescriptor` 和 `Object.getOwnPropertyDescriptor` 的一个区别是，如果第一个参数不是对象，`Object.getOwnPropertyDescriptor(1, 'foo')` 不报错，返回 `undefined`，而 `Reflect.getOwnPropertyDescriptor(1, 'foo')` 会抛出错误，表示参数非法。

Reflect.isExtensible (target)

`Reflect.isExtensible` 方法对应 `Object.isExtensible`，返回一个布尔值，表示当前对象是否可扩展。

```
const myObject = {};

// 旧写法
Object.isExtensible(myObject) // true

// 新写法
Reflect.isExtensible(myObject) // true
```

如果参数不是对象，`Object.isExtensible` 会返回 `false`，因为非对象本来就是不可扩展的，而 `Reflect.isExtensible` 会报错。

```
Object.isExtensible(1) // false  
Reflect.isExtensible(1) // 报错
```

Reflect.preventExtensions(target)

`Reflect.preventExtensions` 对应 `Object.preventExtensions` 方法，用于让一个对象变为不可扩展。它返回一个布尔值，表示是否操作成功。

```
var myObject = {};  
  
// 旧写法  
Object.isExtensible(myObject) // true  
  
// 新写法  
Reflect.preventExtensions(myObject) // true
```

如果参数不是对象，`Object.isExtensible` 在 ES5 环境报错，在 ES6 环境返回这个参数，而 `Reflect.preventExtensions` 会报错。

```
// ES5  
Object.preventExtensions(1) // 报错  
  
// ES6  
Object.preventExtensions(1) // 1  
  
// 新写法  
Reflect.preventExtensions(1) // 报错
```

Reflect.ownKeys (target)

`Reflect.ownKeys` 方法用于返回对象的所有属性，基本等同于 `Object.getOwnPropertyNames` 与 `Object.getOwnPropertySymbols` 之和。

```
var myObject = {  
  foo: 1,  
  bar: 2,  
  [Symbol.for('baz')]: 3,  
  [Symbol.for('bing')]: 4,  
};  
  
// 旧写法  
Object.getOwnPropertyNames(myObject)  
// ['foo', 'bar']  
  
Object.getOwnPropertySymbols(myObject)  
//[Symbol.for('baz'), Symbol.for('bing')]  
  
// 新写法  
Reflect.ownKeys(myObject)  
// ['foo', 'bar', Symbol.for('baz'), Symbol.for('bing')]
```

实例：使用 Proxy 实现观察者模式

观察者模式（Observer mode）指的是函数自动观察数据对象，一旦对象有变化，函数就会自动执行。

```
const person = observable({  
  name: '张三',  
  age: 20
```

```
});  
  
function print() {  
  console.log(` ${person.name}, ${person.age}`)  
}  
  
observe(print);  
person.name = '李四';  
// 输出  
// 李四, 20
```

上面代码中，数据对象 `person` 是观察目标，函数 `print` 是观察者。一旦数据对象发生变化，`print` 就会自动执行。

下面，使用 `Proxy` 写一个观察者模式的最简单实现，即实现 `observable` 和 `observe` 这两个函数。思路是 `observable` 函数返回一个原始对象的 `Proxy` 代理，拦截赋值操作，触发充当观察者的各个函数。

```
const queuedObservers = new Set();  
  
const observe = fn => queuedObservers.add(fn);  
const observable = obj => new Proxy(obj, {set});  
  
function set(target, key, value, receiver) {  
  const result = Reflect.set(target, key, value, receiver);  
  queuedObservers.forEach(observer => observer());  
  return result;  
}
```

上面代码中，先定义了一个 `Set` 集合，所有观察者函数都放进这个集合。然后，`observable` 函数返回原始对象的代理，拦截赋值操作。拦截函数 `set` 之中，会自动执行所有观察者。

Iterator和for...of循环

Iterator（遍历器）的概念

JavaScript原有的表示“集合”的数据结构，主要是数组（Array）和对象（Object），ES6又添加了Map和Set。这样就有了四种数据集合，用户还可以组合使用它们，定义自己的数据结构，比如数组的成员是Map，Map的成员是对象。这样就需要一种统一的接口机制，来处理所有不同的数据结构。

遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是ES6创造了一种新的遍历命令 `for...of` 循环，Iterator接口主要供 `for...of` 消费。

Iterator的遍历过程是这样的。

- (1) 创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。
- (2) 第一次调用指针对象的 `next` 方法，可以将指针指向数据结构的第一个成员。
- (3) 第二次调用指针对象的 `next` 方法，指针就指向数据结构的第二个成员。
- (4) 不断调用指针对象的 `next` 方法，直到它指向数据结构的结束位置。

每一次调用 `next` 方法，都会返回数据结构的当前成员的信息。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。

```
var it = makeIterator(['a', 'b']);
```

```

it.next() // { value: "a", done: false }
it.next() // { value: "b", done: false }
it.next() // { value: undefined, done: true }

function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {value: array[nextIndex++], done: false} :
        {value: undefined, done: true};
    }
  };
}

```

上面代码定义了一个 `makeIterator` 函数，它是一个遍历器生成函数，作用就是返回一个遍历器对象。对数组 `['a', 'b']` 执行这个函数，就会返回该数组的遍历器对象（即指针对象）`it`。

指针对象的 `next` 方法，用来移动指针。开始时，指针指向数组的开始位置。然后，每次调用 `next` 方法，指针就会指向数组的下一个成员。第一次调用，指向 `a`；第二次调用，指向 `b`。

`next` 方法返回一个对象，表示当前数据成员的信息。这个对象具有 `value` 和 `done` 两个属性，`value` 属性返回当前位置的成员，`done` 属性是一个布尔值，表示遍历是否结束，即是否还有必要再一次调用 `next` 方法。

总之，调用指针对象的 `next` 方法，就可以遍历事先给定的数据结构。

对于遍历器对象来说，`done: false` 和 `value: undefined` 属性都是可以省略的，因此上面的 `makeIterator` 函数可以简写成下面的形式。

```

function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {value: array[nextIndex++]} :
        {done: true};
    }
  };
}

```

由于 Iterator 只是把接口规格加到数据结构之上，所以，遍历器与它所遍历的那个数据结构，实际上是分开的，完全可以写出没有对应数据结构的遍历器对象，或者说用遍历器对象模拟出数据结构。下面是一个无限运行的遍历器对象的例子。

```

var it = idMaker();

it.next().value // '0'
it.next().value // '1'
it.next().value // '2'
// ...

function idMaker() {
  var index = 0;

  return {
    next: function() {
      return {value: index++, done: false};
    }
  };
}

```

上面的例子中，遍历器生成函数 `idMaker`，返回一个遍历器对象（即指针对象）。但是并没有对应的数据结构，或者说，遍历器对象自己描述了一个数据结构出来。

在ES6中，有些数据结构原生具备Iterator接口（比如数组），即不用任何处理，就可以被 `for...of` 循环遍历，有些就不行（比如对象）。原因在于，这些数据结构原生部署了 `Symbol.iterator` 属性（详见下文），另外一些数据结构没有。凡是部署了 `Symbol.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对象。

如果使用TypeScript的写法，遍历器接口（Iterable）、指针对象（Iterator）和next方法返回值的规格可以描述如下。

```
interface Iterable {
  [Symbol.iterator](): Iterator,
}

interface Iterator {
  next(value?: any): IterationResult,
}

interface IterationResult {
  value: any,
  done: boolean,
}
```

数据结构的默认Iterator接口

Iterator接口的目的，就是为所有数据结构，提供了一种统一的访问机制，即 `for...of` 循环（详见下文）。当使用 `for...of` 循环遍历某种数据结构时，该循环会自动去寻找Iterator接口。

一种数据结构只要部署了Iterator接口，我们就称这种数据结构是“可遍历的”（iterable）。

ES6规定，默认的Iterator接口部署在数据结构的 `Symbol.iterator` 属性，或者说，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）。`Symbol.iterator` 属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。至于属性名 `Symbol.iterator`，它是一个表达式，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。（参见Symbol一章）。

```
const obj = {
  [Symbol.iterator] : function () {
    return {
      next: function () {
        return {
          value: 1,
          done: true
        };
      }
    };
  }
};
```

上面代码中，对象 `obj` 是可遍历的（iterable），因为具有 `Symbol.iterator` 属性。执行这个属性，会返回一个遍历器对象。该对象的根本特征就是具有 `next` 方法。每次调用 `next` 方法，都会返回一个代表当前成员的信息对象，具有 `value` 和 `done` 两个属性。

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，就得到遍历器对象。

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，不用自己写遍历器生成函数，`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生成方法（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对象。

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;
    current = current.next;
    return {
      done: false,
      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` 方法直接引用数组的Iterator接口。

```

NodeList.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];
// 或者
NodeList.prototype[Symbol.iterator] = [] [Symbol.iterator];

```

```
[...document.querySelectorAll('div')] // 可以执行了
```

下面是类似数组的对象调用数组的 `Symbol.iterator` 方法的例子。

```
let iterable = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3,
  [Symbol.iterator]: Array.prototype[Symbol.iterator]
};
for (let item of iterable) {
  console.log(item); // 'a', 'b', 'c'
}
```

注意，普通对象部署数组的 `Symbol.iterator` 方法，并无效果。

```
let iterable = {
  a: 'a',
  b: 'b',
  c: 'c',
  length: 3,
  [Symbol.iterator]: Array.prototype[Symbol.iterator]
};
for (let item of iterable) {
  console.log(item); // undefined, undefined, undefined
}
```

如果 `Symbol.iterator` 方法对应的不是遍历器生成函数（即会返回一个遍历器对象），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`方法对应的不是遍历器生成函数，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` 是它的遍历器对象。遍历器对象每次移动指针（`next` 方法），都检查一下返回值的 `done` 属性，如果遍历还没结束，就移动遍历器对象的指针到下一步（`next` 方法），不断循环。

调用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 `Symbol.iterator` 方法。

```
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) yield **

yield*后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。

```
let generator = function* () {
  yield 1;
  yield* [2,3,4];
  yield 5;
};

var iterator = generator();

iterator.next() // { value: 1, done: false }
iterator.next() // { value: 2, done: false }
iterator.next() // { value: 3, done: false }
iterator.next() // { value: 4, done: false }
iterator.next() // { value: 5, done: false }
iterator.next() // { value: undefined, done: true }
```

(4) 其他场合

由于数组的遍历会调用遍历器接口，所以任何接受数组作为参数的场合，其实都调用了遍历器接口。下面是一些例子。

- for...of
- Array.from()
- Map(), Set(), WeakMap(), WeakSet() (比如 `new Map([['a', 1], ['b', 2]])`)
- Promise.all()
- Promise.race()

字符串的Iterator接口

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对象，在这个遍历器上可以调用`next`方法，实现对于字符串的遍历。

可以覆盖原生的 `Symbol.iterator` 方法，达到修改遍历器行为的目的。

```
var str = new String("hi");

[...str] // ["h", "i"]

str[Symbol.iterator] = function() {
  return {
    next: function() {
      if (this._first) {
        this._first = false;
        return { value: "bye", done: false };
      } else {
        return { done: true };
      }
    },
    _first: true
  };
};

[...str] // ["bye"]
str // "hi"
```

上面代码中，字符串`str`的 `Symbol.iterator` 方法被修改了，所以扩展运算符（`...`）返回的值变成了 `bye`，而字符串本身还是 `hi`。

Iterator接口与Generator函数

`Symbol.iterator` 方法的最简单实现，还是使用下一章要介绍的Generator函数。

```
var myIterable = {};

myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};

[...myIterable] // [1, 2, 3]
```

// 或者采用下面的简洁写法

```
let obj = {
  * [Symbol.iterator]() {
    yield 'hello';
    yield 'world';
  }
};
```

```
for (let x of obj) {
  console.log(x);
}
// hello
// world
```

上面代码中，`Symbol.iterator` 方法几乎不用部署任何代码，只要用`yield`命令给出每一步的返回值即可。

遍历器对象的`return()`, `throw()`

遍历器对象除了具有 `next` 方法，还可以具有 `return` 方法和 `throw` 方法。如果你自己写遍历器对象生成函数，那么 `next` 方法是必须部署的，`return` 方法和 `throw` 方法是否部署是可选的。

`return` 方法的使用场合是，如果 `for...of` 循环提前退出（通常是因为出错，或者有 `break` 语句或 `continue` 语句），就会调用 `return` 方法。如果一个对象在完成遍历前，需要清理或释放资源，就可以部署 `return` 方法。

```
function readLinesSync(file) {
  return {
    next() {
      if (file.isAtEndOfFile()) {
        file.close();
        return { done: true };
      }
    },
    return() {
      file.close();
      return { done: true };
    },
  };
}
```

上面代码中，函数 `readLinesSync` 接受一个文件对象作为参数，返回一个遍历器对象，其中除了 `next` 方法，还部署了 `return` 方法。下面，我们让文件的遍历提前返回，这样就会触发执行 `return` 方法。

```
for (let line of readLinesSync(fileName)) {
  console.log(line);
  break;
}
```

注意，`return` 方法必须返回一个对象，这是Generator规格决定的。

`throw` 方法主要是配合Generator函数使用，一般的遍历器对象用不到这个方法。请参阅《Generator函数》一章。

for...of循环

ES6 借鉴 C++、Java、C# 和 Python 语言，引入了 `for...of` 循环，作为遍历所有数据结构的统一的方法。

一个数据结构只要部署了 `Symbol.iterator` 属性，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（即默认部署了 `Symbol.iterator` 属性），`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'];
```

```
for(let v of arr) {
  console.log(v); // red green blue
}

const obj = {};
obj[Symbol.iterator] = arr[Symbol.iterator].bind(arr);

for(let v of obj) {
  console.log(v); // red green blue
}
```

上面代码中，空对象 `obj` 部署了数组 `arr` 的 `Symbol.iterator` 属性，结果 `obj` 的 `for...of` 循环，产生了与 `arr` 完全一样的结果。

`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 (let a in arr) {
  console.log(a); // 0 1 2 3
}

for (let 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。

`for...of` 循环调用遍历器接口，数组的遍历器接口只返回具有数字索引的属性。这一点跟 `for...in` 循环也不一样。

```
let arr = [3, 5, 7];
arr.foo = 'hello';

for (let i in arr) {
  console.log(i); // "0", "1", "2", "foo"
}

for (let i of arr) {
  console.log(i); // "3", "5", "7"
}
```

上面代码中，`for...of` 循环不会返回数组 `arr` 的 `foo` 属性。

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 = new 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对象。

- `entries()` 返回一个遍历器对象，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()` 返回一个遍历器对象，用来遍历所有的键名。
- `values()` 返回一个遍历器对象，用来遍历所有的键值。

这三个方法调用后生成的遍历器对象，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 (let e in es6) {
  console.log(e);
}
// edition
// committee
// standard
```

```
for (let 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` 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。

- 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。

- `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。
- 某些情况下，`for...in` 循环会以任意顺序遍历键名。

总之，`for...in` 循环主要是为遍历对象而设计的，不适用于遍历数组。

`for...of` 循环相比上面几种做法，有一些显著的优点。

```
for (let value of myArray) {  
    console.log(value);  
}
```

- 有着同`for...in`一样的简洁语法，但是没有`for...in`那些缺点。
- 不同用于`forEach`方法，它可以与`break`、`continue`和`return`配合使用。
- 提供了遍历所有数据结构的统一操作接口。

下面是一个使用`break`语句，跳出`for...of` 循环的例子。

```
for (var n of fibonacci) {  
    if (n > 1000)  
        break;  
    console.log(n);  
}
```

上面的例子，会输出斐波纳契数列小于等于1000的项。如果当前项大于1000，就会使用`break`语句跳出`for...of` 循环。

Generator 函数

简介

基本概念

Generator函数是ES6提供的一种异步编程解决方案，语法行为与传统函数完全不同。本章详细介绍Generator函数的语法和API，它的异步编程应用请看《异步操作》一章。

Generator函数有多种理解角度。从语法上，首先可以把它理解成，Generator函数是一个状态机，封装了多个内部状态。

执行Generator函数会返回一个遍历器对象，也就是说，Generator函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历Generator函数内部的每一个状态。

形式上，Generator函数是一个普通函数，但是有两个特征。一是，`function` 关键字与函数名之间有一个星号；二是，函数体内部使用`yield`语句，定义不同的内部状态（`yield`语句在英语里的意思就是“产出”）。

```
function* helloWorldGenerator() {  
    yield 'hello';  
    yield 'world';  
    return 'ending';  
}  
  
var hw = helloWorldGenerator();
```

上面代码定义了一个Generator函数`helloWorldGenerator`，它内部有两个`yield`语句“hello”和“world”，即该函数有三个状态：hello，world和`return`语句（结束执行）。

然后，Generator函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是上一章介绍的遍历器对

象 (Iterator Object)。

下一步，必须调用遍历器对象的 `next` 方法，使得指针移向下一个状态。也就是说，每次调用 `next` 方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个 `yield` 语句（或 `return` 语句）为止。换言之，Generator 函数是分段执行的，`yield` 语句是暂停执行的标记，而 `next` 方法可以恢复执行。

```
hw.next()  
// { value: 'hello', done: false }  
  
hw.next()  
// { value: 'world', done: false }  
  
hw.next()  
// { value: 'ending', done: true }  
  
hw.next()  
// { value: undefined, done: true }
```

上面代码一共调用了四次 `next` 方法。

第一次调用，Generator函数开始执行，直到遇到第一个 `yield` 语句为止。`next` 方法返回一个对象，它的 `value` 属性就是当前 `yield` 语句的值hello，`done` 属性的值false，表示遍历还没有结束。

第二次调用，Generator函数从上次 `yield` 语句停下的地方，一直执行到下一个 `yield` 语句。`next` 方法返回的对象的 `value` 属性就是当前 `yield` 语句的值world，`done` 属性的值false，表示遍历还没有结束。

第三次调用，Generator函数从上次 `yield` 语句停下的地方，一直执行到 `return` 语句（如果没有 `return` 语句，就执行到函数结束）。`next` 方法返回的对象的 `value` 属性，就是紧跟在 `return` 语句后面的表达式的值（如果没有 `return` 语句，则 `value` 属性的值为 `undefined`），`done` 属性的值true，表示遍历已经结束。

第四次调用，此时Generator函数已经运行完毕，`next` 方法返回对象的 `value` 属性为 `undefined`，`done` 属性为 `true`。以后再调用 `next` 方法，返回的都是这个值。

总结一下，调用Generator函数，返回一个遍历器对象，代表Generator函数的内部指针。以后，每次调用遍历器对象的 `next` 方法，就会返回一个有着 `value` 和 `done` 两个属性的对象。`value` 属性表示当前的内部状态的值，是 `yield` 语句后面那个表达式的值；`done` 属性是一个布尔值，表示是否遍历结束。

ES6 没有规定，`function` 关键字与函数名之间的星号，写在哪个位置。这导致下面的写法都能通过。

```
function * foo(x, y) { ... }  
  
function *foo(x, y) { ... }  
  
function* foo(x, y) { ... }  
  
function*foo(x, y) { ... }
```

由于Generator函数仍然是普通函数，所以一般的写法是上面的第三种，即星号紧跟在 `function` 关键字后面。本书也采用这种写法。

yield语句

由于Generator函数返回的遍历器对象，只有调用 `next` 方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。`yield` 语句就是暂停标志。

遍历器对象的 `next` 方法的运行逻辑如下。

- (1) 遇到 `yield` 语句，就暂停执行后面的操作，并将紧跟在 `yield` 后面的那个表达式的值，作为返回的对象的 `value` 属性值。
- (2) 下一次调用 `next` 方法时，再继续往下执行，直到遇到下一个 `yield` 语句。

(3) 如果没有再遇到新的 `yield` 语句，就一直运行到函数结束，直到 `return` 语句为止，并将 `return` 语句后面的表达式的值，作为返回的对象的 `value` 属性值。

(4) 如果该函数没有 `return` 语句，则返回的对象的 `value` 属性值为 `undefined`。

需要注意的是，`yield` 语句后面的表达式，只有当调用 `next` 方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

```
function* gen() {
  yield 123 + 456;
}
```

上面代码中，`yield` 后面的表达式 `123 + 456`，不会立即求值，只会在 `next` 方法将指针移到这一句时，才会求值。

`yield` 语句与 `return` 语句既有相似之处，也有区别。相似之处在于，都能返回紧跟在语句后面的那个表达式的值。区别在于每次遇到 `yield`，函数暂停执行，下一次再从该位置继续向后执行，而 `return` 语句不具备位置记忆的功能。一个函数里面，只能执行一次（或者说一个）`return` 语句，但是可以执行多次（或者说多个）`yield` 语句。正常函数只能返回一个值，因为只能执行一次 `return`；Generator 函数可以返回一系列的值，因为可以有任意多个 `yield`。从另一个角度看，也可以说 Generator 生成了一系列的值，这也就是它的名称的来历（在英语中，generator 这个词是“生成器”的意思）。

Generator 函数可以不用 `yield` 语句，这时就变成了一个单纯的暂缓执行函数。

```
function* f() {
  console.log('执行了！');
}

var generator = f();

setTimeout(function () {
  generator.next()
}, 2000);
```

上面代码中，函数 `f` 如果是普通函数，在为变量 `generator` 赋值时就会执行。但是，函数 `f` 是一个 Generator 函数，就变成只有调用 `next` 方法时，函数 `f` 才会执行。

另外需要注意，`yield` 语句不能用在普通函数中，否则会报错。

```
(function () {
  yield 1;
})()
// SyntaxError: Unexpected number
```

上面代码在一个普通函数中使用 `yield` 语句，结果产生一个句法错误。

下面是另外一个例子。

```
var arr = [1, [[2, 3], 4], [5, 6]];

var flat = function* (a) {
  a.forEach(function (item) {
    if (typeof item !== 'number') {
      yield* flat(item);
    } else {
      yield item;
    }
  });
};

for (var f of flat(arr)) {
```

```
    console.log(f);
}
```

上面代码也会产生句法错误，因为 `forEach` 方法的参数是一个普通函数，但是在里面使用了 `yield` 语句（这个函数里面还使用了 `yield*` 语句，这里可以不用理会，详细说明见后文）。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
```

另外，`yield` 语句如果用在一个表达式之中，必须放在圆括号里面。

```
console.log('Hello' + yield); // SyntaxError
console.log('Hello' + yield 123); // SyntaxError

console.log('Hello' + (yield)); // OK
console.log('Hello' + (yield 123)); // OK
```

`yield` 语句用作函数参数或赋值表达式的右边，可以不加括号。

```
foo(yield 'a', yield 'b'); // OK
let input = yield; // OK
```

与Iterator接口的关系

上一章说过，任意一个对象的 `Symbol.iterator` 方法，等于该对象的遍历器生成函数，调用该函数会返回该对象的一个遍历器对象。

由于Generator函数就是遍历器生成函数，因此可以把Generator赋值给对象的 `Symbol.iterator` 属性，从而使得该对象具有Iterator接口。

```
var myIterable = {};
myIterable[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};

[...myIterable] // [1, 2, 3]
```

上面代码中，Generator函数赋值给 `Symbol.iterator` 属性，从而使得 `myIterable` 对象具有了Iterator接口，可以被 `...` 运算符遍历了。

Generator函数执行后，返回一个遍历器对象。该对象本身也具有 `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对象 `g`。它的 `Symbol.iterator` 属性，也是一个遍历器对象生成函数，执行后返回它自己。

next方法的参数

`yield` 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:true}

var b = foo(5);
b.next() // { value:6, done:false }
b.next(12) // { value:8, done:false }
b.next(13) // { value:42, done:true }
```

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。上面代码第一次调用 `b` 的 `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。从语义上讲，第一个 `next` 方法用来启动遍历器对象，所以不用带有参数。

如果想要第一次调用 `next` 方法时，就能够输入值，可以在Generator函数外面再包一层。

```
function wrapper(generatorFunction) {
  return function (...args) {
    let generatorObject = generatorFunction(...args);
    generatorObject.next();
    return generatorObject;
  };
}

const wrapped = wrapper(function* () {
  console.log(`First input: ${yield}`);
  return 'DONE';
});

wrapped().next('hello!')
// First input: hello!
```

上面代码中，Generator函数如果不包一层，是无法第一次调用 `next` 方法，就输入参数的。

再看一个通过 `next` 方法的参数，向Generator函数内部输入值的例子。

```
function* dataConsumer() {
  console.log('Started');
  console.log(`1. ${yield}`);
  console.log(`2. ${yield}`);
  return 'result';
}

let genObj = dataConsumer();
genObj.next();
// Started
genObj.next('a')
// 1. a
genObj.next('b')
// 2. b
```

上面代码是一个很直观的例子，每次通过 `next` 方法向Generator函数输入值，然后打印出来。

for...of循环

`for...of` 循环可以自动遍历Generator函数时生成的 `Iterator` 对象，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。

利用 `for...of` 循环，可以写出遍历任意对象（object）的方法。原生的JavaScript对象没有遍历接口，无法使用 `for...of` 循环，通过Generator函数为它加上这个接口，就可以用了。

```
function* objectEntries(obj) {
  let propKeys = Reflect.ownKeys(obj);

  for (let propKey of propKeys) {
    yield [propKey, obj[propKey]];
  }
}

let jane = { first: 'Jane', last: 'Doe' };

for (let [key, value] of objectEntries(jane)) {
  console.log(` ${key}: ${value}`);
}
// first: Jane
// last: Doe
```

上面代码中，对象 `jane` 原生不具备Iterator接口，无法用 `for...of` 遍历。这时，我们通过Generator函数 `objectEntries` 为它加上遍历器接口，就可以用 `for...of` 遍历了。加上遍历器接口的另一种写法是，将Generator函数加到对象的 `Symbol.iterator` 属性上面。

```
function* objectEntries() {
  let propKeys = Object.keys(this);

  for (let propKey of propKeys) {
    yield [propKey, this[propKey]];
  }
}

let jane = { first: 'Jane', last: 'Doe' };

jane[Symbol.iterator] = objectEntries;

for (let [key, value] of jane) {
  console.log(` ${key}: ${value}`);
}
```

```
}
```

```
// first: Jane
```

```
// last: Doe
```

除了 `for...of` 循环以外，扩展运算符（`...`）解构赋值和 `Array.from` 方法内部调用的，都是遍历器接口。这意味着，它们都可以将Generator函数返回的Iterator对象，作为参数。

```
function* numbers () {
  yield 1
  yield 2
  return 3
  yield 4
}

// 扩展运算符
[...numbers()] // [1, 2]

// Array.from 方法
Array.from(numbers()) // [1, 2]

// 解构赋值
let [x, y] = numbers();
x // 1
y // 2

// for...of 循环
for (let n of numbers()) {
  console.log(n)
}
// 1
// 2
```

Generator.prototype.throw()

Generator函数返回的遍历器对象，都有一个 `throw` 方法，可以在函数体外抛出错误，然后在Generator函数体内捕获。

```
var g = function* () {
  try {
    yield;
  } catch (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，遍历器对象 `i` 连续抛出两个错误。第一个错误被Generator函数体内的 `catch` 语句捕获。`i` 第二次抛出错误，由于Generator函数内部的 `catch` 语句已经执行过了，不会再捕捉到这个错误了，所以这个错误就被抛出了Generator函数体，被函数体外的 `catch` 语句捕获。

`throw` 方法可以接受一个参数，该参数会被 `catch` 语句接收，建议抛出 `Error` 对象的实例。

```
var g = function* () {
  try {
    yield;
  } catch (e) {
    console.log(e);
  }
};

var i = g();
i.next();
i.throw(new Error('出错了! '));
// Error: 出错了! (...)
```

注意，不要混淆遍历器对象的 `throw` 方法和全局的 `throw` 命令。上面代码的错误，是用遍历器对象的 `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 `try` 代码块里面剩余的语句了。

如果Generator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，Generator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Generator函数内部和外部，都没有部署 `try...catch` 代码块，那么程序将报错，直接中断执行。

```
var gen = function* gen(){
  yield console.log('hello');
  yield console.log('world');
}

var g = gen();
g.next();
g.throw();
// hello
// Uncaught undefined
```

上面代码中，`g.throw` 抛出错误以后，没有任何 `try...catch` 代码块可以捕获这个错误，导致程序报错，中断执行。

`throw` 方法被捕获以后，会附带执行下一条 `yield` 语句。也就是说，会附带执行一次 `next` 方法。

```
var gen = function* gen(){
  try {
    yield console.log('a');
  } catch (e) {
    // ...
  }
  yield console.log('b');
  yield console.log('c');
}

var g = gen();
g.next() // a
g.throw() // b
g.next() // c
```

上面代码中，`g.throw` 方法被捕获以后，自动执行了一次 `next` 方法，所以会打印 `b`。另外，也可以看到，只要 Generator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。

另外，`throw` 命令与 `g.throw` 方法是无关的，两者互不影响。

```
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，都进行了正确的操作。

这种函数体内捕获错误的机制，大大方便了对错误的处理。多个 `yield` 语句，可以只用一个 `try...catch` 代码块来捕获错误。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内部写一个错误处理语句，现在只在 Generator函数内部写一次 `catch` 语句就可以了。

Generator函数体外抛出的错误，可以在函数体内捕获；反过来，Generator函数体内抛出的错误，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，且没有被内部捕获，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。

Generator.prototype.return()

Generator函数返回的遍历器对象，还有一个 `return` 方法，可以返回给定的值，并且终结遍历Generator函数。

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

var g = gen();

g.next()      // { value: 1, done: false }
g.return('foo') // { value: "foo", done: true }
g.next()      // { value: undefined, done: true }
```

上面代码中，遍历器对象 `g` 调用 `return` 方法后，返回值的 `value` 属性就是 `return` 方法的参数 `foo`。并且，Generator函数的遍历就终止了，返回值的 `done` 属性为 `true`，以后再调用 `next` 方法，`done` 属性总是返回 `true`。

如果 `return` 方法调用时，不提供参数，则返回值的 `value` 属性为 `undefined`。

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

var g = gen();

g.next()      // { value: 1, done: false }
g.return() // { value: undefined, done: true }
```

如果Generator函数内部有 `try...finally` 代码块，那么 `return` 方法会推迟到 `finally` 代码块执行完再执行。

```
function* numbers () {
  yield 1;
  try {
    yield 2;
    yield 3;
  } finally {
    yield 4;
    yield 5;
  }
  yield 6;
}
var g = numbers()
g.next() // { done: false, value: 1 }
g.next() // { done: false, value: 2 }
g.return(7) // { done: false, value: 4 }
g.next() // { done: false, value: 5 }
g.next() // { done: true, value: 7 }
```

上面代码中，调用 `return` 方法后，就开始执行 `finally` 代码块，然后等到 `finally` 代码块执行完，再执行 `return` 方法。

yield* 语句

如果在 Generator 函数内部，调用另一个 Generator 函数，默认情况下是没有效果的。

```
function* foo() {
  yield 'a';
  yield 'b';
}

function* bar() {
  yield 'x';
  foo();
  yield 'y';
}

for (let v of bar()){
  console.log(v);
}
// "x"
// "y"
```

上面代码中，`foo` 和 `bar` 都是 Generator 函数，在 `bar` 里面调用 `foo`，是不会有效果的。

这个就需要用到 `yield*` 语句，用来在一个 Generator 函数里面执行另一个 Generator 函数。

```
function* bar() {
  yield 'x';
  yield* foo();
  yield 'y';
}

// 等同于
function* bar() {
  yield 'x';
  yield 'a';
  yield 'b';
  yield 'y';
}

// 等同于
function* bar() {
  yield 'x';
  for (let v of foo()) {
    yield v;
  }
  yield 'y';
}

for (let v of bar()){
  console.log(v);
}
// "x"
// "a"
// "b"
// "y"
```

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().value // "open"
gen.next() // 返回一个遍历器对象
gen.next().value // "close"

function* outer2() {
  yield 'open'
  yield* inner()
  yield 'close'
}

var gen = outer2()
gen.next().value // "open"
gen.next().value // "hello!"
gen.next().value // "close"

```

上面例子中，`outer2` 使用了 `yield*`，`outer1` 没使用。结果就是，`outer1` 返回一个遍历器对象，`outer2` 返回该遍历器对象的内部值。

从语法角度看，如果 `yield` 命令后面跟的是一个遍历器对象，需要在 `yield` 命令后面加上星号，表明它返回的是一个遍历器对象。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Generator函数，有递归的效果。

`yield*` 后面的Generator函数（没有 `return` 语句时），等同于在Generator函数内部，部署一个 `for...of` 循环。

```

function* concat(iter1, iter2) {
  yield* iter1;
  yield* iter2;
}

// 等同于

function* concat(iter1, iter2) {
  for (var value of iter1) {
    yield value;
}

```

```
    }
    for (var value of iter2) {
        yield value;
    }
}
```

上面代码说明，`yield*` 后面的Generator函数（没有 `return` 语句时），不过是 `for...of` 的一种简写形式，完全可以用后者替代前者。反之，则需要用 `var value = yield* iterator` 的形式获取 `return` 语句的值。

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对象。

实际上，任何数据结构只要有Iterator接口，就可以被 `yield*` 遍历。

```
let read = (function* () {
    yield 'hello';
    yield* 'hello';
})();

read.next().value // "hello"
read.next().value // "h"
```

上面代码中，`yield` 语句返回整个字符串，`yield*` 语句返回单个字符。因为字符串具有Iterator接口，所以被 `yield*` 遍历。

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()
// {value: 1, done: false}
it.next()
// {value: 2, done: false}
it.next()
// {value: 3, done: false}
it.next();
// "v: foo"
// {value: 4, done: false}
it.next()
// {value: undefined, done: true}
```

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。

再看一个例子。

```
function* genFuncWithReturn() {
  yield 'a';
  yield 'b';
  return 'The result';
}
function* logReturned(genObj) {
  let result = yield* genObj;
  console.log(result);
}

[...logReturned(genFuncWithReturn())]
// The result
// 值为 [ 'a', 'b' ]
```

上面代码中，存在两次遍历。第一次是扩展运算符遍历函数 `logReturned` 返回的遍历器对象，第二次是 `yield*` 语句遍历函数 `genFuncWithReturn` 返回的遍历器对象。这两次遍历的效果是叠加的，最终表现为扩展运算符遍历函数 `genFuncWithReturn` 返回的遍历器对象。所以，最后的数据表达式得到的值等于 `['a', 'b']`。但是，函数 `genFuncWithReturn` 的 `return` 语句的返回值 `The result`，会返回给函数 `logReturned` 内部的 `result` 变量，因此会有终端输出。

`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函数的 `this`

Generator函数总是返回一个遍历器，ES6规定这个遍历器是Generator函数的实例，也继承了Generator函数的 `prototype` 对象上的方法。

```

function* g() {}

g.prototype.hello = function () {
  return 'hi!';
};

let obj = g();

obj instanceof g // true
obj.hello() // 'hi!'

```

上面代码表明，Generator函数 `g` 返回的遍历器 `obj`，是 `g` 的实例，而且继承了 `g.prototype`。但是，如果把 `g` 当作普通的构造函数，并不会生效，因为 `g` 返回的总是遍历器对象，而不是 `this` 对象。

```
function* g() {
  this.a = 11;
}

let obj = g();
obj.a // undefined
```

上面代码中，Generator函数 `g` 在 `this` 对象上面添加了一个属性 `a`，但是 `obj` 对象拿不到这个属性。

Generator函数也不能跟 `new` 命令一起用，会报错。

```
function* F() {
  yield this.x = 2;
  yield this.y = 3;
}

new F()
// TypeError: F is not a constructor
```

上面代码中，`new` 命令跟构造函数 `F` 一起使用，结果报错，因为 `F` 不是构造函数。

那么，有没有办法让Generator函数返回一个正常的对象实例，既可以用 `next` 方法，又可以获得正常的 `this`？

下面是一个变通方法。首先，生成一个空对象，使用 `call` 方法绑定Generator函数内部的 `this`。这样，构造函数调用以后，这个空对象就是Generator函数的实例对象了。

```
function* F() {
  this.a = 1;
  yield this.b = 2;
  yield this.c = 3;
}
var obj = {};
var f = F.call(obj);

f.next(); // Object {value: 2, done: false}
f.next(); // Object {value: 3, done: false}
f.next(); // Object {value: undefined, done: true}

obj.a // 1
obj.b // 2
obj.c // 3
```

上面代码中，首先是 `F` 内部的 `this` 对象绑定 `obj` 对象，然后调用它，返回一个Iterator对象。这个对象执行三次 `next` 方法（因为 `F` 内部有两个 `yield` 语句），完成 `F` 内部所有代码的运行。这时，所有内部属性都绑定在 `obj` 对象上了，因此 `obj` 对象也就成了 `F` 的实例。

上面代码中，执行的是遍历器对象 `f`，但是生成的对象实例是 `obj`，有没有办法将这两个对象统一呢？

一个办法就是将 `obj` 换成 `F.prototype`。

```
function* F() {
  this.a = 1;
  yield this.b = 2;
  yield this.c = 3;
}
var f = F.call(F.prototype);

f.next(); // Object {value: 2, done: false}
f.next(); // Object {value: 3, done: false}
f.next(); // Object {value: undefined, done: true}

f.a // 1
```

```
f.b // 2  
f.c // 3
```

再将 F 改成构造函数，就可以对它执行 new 命令了。

```
function* gen() {  
    this.a = 1;  
    yield this.b = 2;  
    yield this.c = 3;  
}  
  
function F() {  
    return gen.call(gen.prototype);  
}  
  
var f = new F();  
  
f.next(); // Object {value: 2, done: false}  
f.next(); // Object {value: 3, done: false}  
f.next(); // Object {value: undefined, done: true}  
  
f.a // 1  
f.b // 2  
f.c // 3
```

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) {  
        console.log('Tick!');  
        yield;  
        console.log('Tock!');  
        yield;  
    }  
};
```

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。

```

Promise.resolve(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(value1) {
  try {
    var value2 = yield step1(value1);
    var value3 = yield step2(value2);
    var value4 = yield step3(value3);
    var value5 = yield step4(value4);
    // 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(initialValue));

function scheduler(task) {
  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);
  }
}
```

注意，上面这种做法，只适合同步操作，即所有的 `task` 都必须是同步的，不能有异步操作。因为这里的代码一得到返回值，就继续往下执行，没有判断异步操作何时完成。如果要控制异步的操作流程，详见后面的《异步操作》一章。

下面，利用 `for...of` 循环会自动依次执行 `yield` 命令的特性，提供一种更一般的控制流管理的方法。

```
let steps = [step1Func, step2Func, step3Func];

function *iterateSteps(steps){
  for (var i=0; i< steps.length; i++){
    var step = steps[i];
    yield step();
  }
}
```

上面代码中，数组 `steps` 封装了一个任务的多个步骤，Generator函数 `iterateSteps` 则是依次为这些步骤加上 `yield` 命令。

将任务分解成步骤之后，还可以将项目分解成多个依次执行的任务。

```
let jobs = [job1, job2, job3];

function *iterateJobs(jobs){
  for (var i=0; i< jobs.length; i++){
    var job = jobs[i];
    yield *iterateSteps(job.steps);
  }
}
```

上面代码中，数组 `jobs` 封装了一个项目的多个任务，Generator函数 `iterateJobs` 则是依次为这些任务加上 `yield *` 命令。

最后，就可以用 `for...of` 循环一次性依次执行所有任务的所有步骤。

```
for (var step of iterateJobs(jobs)){
  console.log(step.id);
}
```

再次提醒，上面的做法只能用于所有步骤都是同步操作的情况，不能有异步操作的步骤。如果想要依次执行异步的步骤，必须使用后面的《异步操作》一章介绍的方法。

`for...of` 的本质是一个 `while` 循环，所以上面的代码实质上执行的是下面的逻辑。

```
var it = iterateJobs(jobs);
var res = it.next();

while (!res.done){
  var result = res.value;
  // ...
  res = it.next();
}
```

(3) 部署Iterator接口

利用Generator函数，可以在任意对象上部署Iterator接口。

```
function* iterEntries(obj) {
  let keys = Object.keys(obj);
  for (let i=0; i < keys.length; i++) {
    let key = keys[i];
    yield [key, obj[key]];
  }
}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {
  console.log(key, value);
}

// foo 3
// bar 7
```

上述代码中，`myObj` 是一个普通对象，通过 `iterEntries` 函数，就有了Iterator接口。也就是说，可以在任意对象上部署 `next` 方法。

下面是一个对数组部署Iterator接口的例子，尽管数组原生具有这个接口。

```
function* makeSimpleGenerator(array){
  var nextIndex = 0;

  while(nextIndex < array.length){
    yield array[nextIndex++];
  }
}

var gen = makeSimpleGenerator(['yo', 'ya']);

gen.next().value // 'yo'
gen.next().value // 'ya'
gen.next().done // true
```

(4) 作为数据结构

Generator可以看作是数据结构，更确切地说，可以看作是一个数组结构，因为Generator函数可以返回一系列的值，这意味着它可以对任意表达式，提供类似数组的接口。

```
function *doStuff() {
  yield fs.readFile.bind(null, 'hello.txt');
  yield fs.readFile.bind(null, 'world.txt');
  yield fs.readFile.bind(null, 'and-such.txt');
}
```

上面代码就是依次返回三个函数，但是由于使用了Generator函数，导致可以像处理数组那样，处理这三个返回的函数。

```
for (task of doStuff()) {
  // task是一个函数，可以像回调函数那样使用它
}
```

实际上，如果用ES5表达，完全可以用数组模拟Generator的这种用法。

```
function doStuff() {
  return [
    fs.readFile.bind(null, 'hello.txt'),
    fs.readFile.bind(null, 'world.txt'),
    fs.readFile.bind(null, 'and-such.txt')
  ];
}
```

上面的函数，可以用一模一样的for...of循环处理！两相一比较，就不难看出Generator使得数据或者操作，具备了类似数组的接口。

Promise对象

Promise的含义

Promise是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6将其写进了语言标准，统一了用法，原生提供了 **Promise** 对象。

所谓 **Promise**，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise是一个对象，从它可以获取异步操作的消息。Promise提供统一的API，各种异步操作都可以用同样的方法进行处理。

Promise 对象有以下两个特点。

- (1) 对象的状态不受外界影响。**Promise** 对象代表一个异步操作，有三种状态：**Pending**（进行中）、**Resolved**（已完成，又称Fulfilled）和**Rejected**（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 **Promise** 这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。
- (2) 一旦状态改变，就不会再变，任何时候都可以得到这个结果。**Promise** 对象的状态改变，只有两种可能：从 **Pending** 变为 **Resolved** 和从 **Pending** 变为 **Rejected**。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 **Promise** 对象添加回调函数，也会立即得到这个结果。这与事件 (Event) 完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了 **Promise** 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，**Promise** 对象提供统一的接口，使得控制异步操作更加容易。

Promise 也有一些缺点。首先，无法取消 **Promise**，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，**Promise** 内部抛出的错误，不会反应到外部。第三，当处于 **Pending** 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果某些事件不断地反复发生，一般来说，使用stream模式是比部署 `Promise` 更好的选择。

基本用法

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

下面代码创造了一个`Promise`实例。

```
var promise = new Promise(function(resolve, reject) {
  // ... some code

  if /* 异步操作成功 */{
    resolve(value);
  } else {
    reject(error);
  }
});
```

`Promise`构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。它们是两个函数，由 JavaScript引擎提供，不用自己部署。

`resolve` 函数的作用是，将`Promise`对象的状态从“未完成”变为“成功”（即从Pending变为Resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去； `reject` 函数的作用是，将`Promise`对象的状态从“未完成”变为“失败”（即从Pending变为Rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

`Promise`实例生成以后，可以用 `then` 方法分别指定 `Resolved` 状态和 `Reject` 状态的回调函数。

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

`then` 方法可以接受两个回调函数作为参数。第一个回调函数是`Promise`对象的状态变为Resolved时调用，第二个回调函数是`Promise`对象的状态变为Reject时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受`Promise`对象传出的值作为参数。

下面是一个`Promise`对象的简单例子。

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

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

上面代码中，`timeout` 方法返回一个`Promise`实例，表示一段时间以后才会发生的结果。过了指定的时间（`ms` 参数）以后，`Promise`实例的状态变为Resolved，就会触发 `then` 方法绑定的回调函数。

`Promise`新建后就会立即执行。

```
let promise = new Promise(function(resolve, reject) {
  console.log('Promise');
  resolve();
});

promise.then(function() {
```

```
    console.log('Resolved.');
});

console.log('Hi!');

// Promise
// Hi!
// Resolved
```

上面代码中，Promise新建后立即执行，所以首先输出的是“Promise”。然后，`then`方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以“Resolved”最后输出。

下面是异步加载图片的例子。

```
function loadImageAsync(url) {
  return new Promise(function(resolve, reject) {
    var image = new Image();

    image.onload = function() {
      resolve(image);
    };

    image.onerror = function() {
      reject(new Error('Could not load image at ' + url));
    };
    image.src = url;
  });
}
```

上面代码中，使用Promise包装了一个图片加载的异步操作。如果加载成功，就调用`resolve`方法，否则就调用`reject`方法。

下面是一个用Promise对象实现的Ajax操作的例子。

```
vargetJSON = 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.readyState !== 4) {
        return;
      }
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
  });

  return promise;
};

getJSON("/posts.json").then(function(json) {
  console.log('Contents: ' + json);
}, function(error) {
  console.error('出错了', error);
});
```

```
});
```

上面代码中，`getJSON` 是对 XMLHttpRequest 对象的封装，用于发出一个针对 JSON 数据的 HTTP 请求，并且返回一个 Promise 对象。需要注意的是，在 `getJSON` 内部，`resolve` 函数和 `reject` 函数调用时，都带有参数。

如果调用 `resolve` 函数和 `reject` 函数时带有参数，那么它们的参数会被传递给回调函数。`reject` 函数的参数通常是 Error 对象的实例，表示抛出的错误；`resolve` 函数的参数除了正常的值以外，还可能是另一个 Promise 实例，表示异步操作的结果有可能是一个值，也有可能是另一个异步操作，比如像下面这样。

```
var p1 = new Promise(function (resolve, reject) {
  // ...
});

var p2 = new Promise(function (resolve, reject) {
  // ...
  resolve(p1);
})
```

上面代码中，`p1` 和 `p2` 都是 Promise 的实例，但是 `p2` 的 `resolve` 方法将 `p1` 作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时 `p1` 的状态就会传递给 `p2`，也就是说，`p1` 的状态决定了 `p2` 的状态。如果 `p1` 的状态是 `Pending`，那么 `p2` 的回调函数就会等待 `p1` 的状态改变；如果 `p1` 的状态已经是 `Resolved` 或者 `Rejected`，那么 `p2` 的回调函数将会立刻执行。

```
var p1 = new Promise(function (resolve, reject) {
  setTimeout(() => reject(new Error('fail')), 3000)
})

var p2 = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(p1), 1000)
})

p2
  .then(result => console.log(result))
  .catch(error => console.log(error))
// Error: fail
```

上面代码中，`p1` 是一个 Promise，3 秒之后变为 `Rejected`。`p2` 的状态在 1 秒之后改变，`resolve` 方法返回的是 `p1`。此时，由于 `p2` 返回的是另一个 Promise，所以后面的 `then` 语句都变成针对后者（`p1`）。又过了 2 秒，`p1` 变为 `Rejected`，导致触发 `catch` 方法指定的回调函数。

Promise.prototype.then()

Promise 实例具有 `then` 方法，也就是说，`then` 方法是定义在原型对象 `Promise.prototype` 上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过，`then` 方法的第一个参数是 `Resolved` 状态的回调函数，第二个参数（可选）是 `Rejected` 状态的回调函数。

`then` 方法返回的是一个新的 Promise 实例（注意，不是原来那个 Promise 实例）。因此可以采用链式写法，即 `then` 方法后面再调用另一个 `then` 方法。

```
getJSON('/posts.json').then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

上面的代码使用 `then` 方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

采用链式的 `then`，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个Promise对象（即有异步操作），这时后一个回调函数，就会等待该Promise对象的状态发生变化，才会被调用。

```
getJSON("/post/1.json").then(function(post) {
  return getJSON(post.commentURL);
}).then(function funcA(comments) {
  console.log("Resolved: ", comments);
}, function funcB(err){
  console.log("Rejected: ", err);
});
```

上面代码中，第一个 `then` 方法指定的回调函数，返回的是另一个Promise对象。这时，第二个 `then` 方法指定的回调函数，就会等待这个新的Promise对象状态发生变化。如果变为Resolved，就调用 `funcA`，如果状态变为Rejected，就调用 `funcB`。

如果采用箭头函数，上面的代码可以写得更简洁。

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

Promise.prototype.catch()

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

```
getJSON("/posts.json").then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误!', error);
});
```

上面代码中，`getJSON` 方法返回一个Promise对象，如果该对象状态变为 `Resolved`，则会调用 `then` 方法指定的回调函数；如果异步操作抛出错误，状态就会变为 `Rejected`，就会调用 `catch` 方法指定的回调函数，处理这个错误。另外，`then` 方法指定的回调函数，如果运行中抛出错误，也会被 `catch` 方法捕获。

```
p.then((val) => console.log("fulfilled:", val))
  .catch((err) => console.log("rejected:", err));

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

下面是一个例子。

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

上面代码中，`promise` 抛出一个错误，就被 `catch` 方法指定的回调函数捕获。注意，上面的写法与下面两种写法是等价的。

```
// 写法一
var promise = new Promise(function(resolve, reject) {
  try {
    throw new Error('test');
  } catch(e) {
    reject(e);
  }
});
promise.catch(function(error) {
  console.log(error);
});

// 写法二
var promise = new Promise(function(resolve, reject) {
  reject(new Error('test'));
});
promise.catch(function(error) {
  console.log(error);
});
```

比较上面两种写法，可以发现 `reject` 方法的作用，等同于抛出错误。

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。

一般来说，不要在 `then` 方法里面定义Reject状态的回调函数（即 `then` 的第二个参数），总是使用 `catch` 方法。

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

// good
promise
  .then(function(data) { //cb
```

```
// success
})
.catch(function(err) {
  // error
});
```

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面 `then` 方法执行中的错误，也更接近同步的写法（`try/catch`）。因此，建议总是使用 `catch` 方法，而不使用 `then` 方法的第二个参数。

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，但是由于没有指定 `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。注意，Chrome浏览器不遵守这条规定，它会抛出错误“`ReferenceError: x is not defined`”。

```
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，结果由于没有指定使用 `try...catch` 语句，就冒泡到最外层，成了未捕获的错误。因为此时，Promise的函数体已经运行结束了，所以这个错误是在Promise函数体外抛出的。

Node 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()
.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。如果没有报错，则会跳过 `catch` 方法。

```
Promise.resolve()
  .catch(function(error) {
    console.log('oh no', error);
})
  .then(function() {
    console.log('carry on');
});
// carry on
```

上面的代码因为没有报错，跳过了 `catch` 方法，直接执行后面的 `then` 方法。此时，要是 `then` 方法里面报错，就与前面的 `catch` 无关了。

`catch` 方法之中，还能再抛出错误。

```
var someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
  // 下面一行会报错，因为y没有声明
  y + 2;
}).then(function() {
  console.log('carry on');
});
// oh no [ReferenceError: x is not defined]
```

上面代码中，`catch` 方法抛出一个错误，因为后面没有别的 `catch` 方法了，导致这个错误不会被捕获，也不会传递到外层。如果改写一下，结果就不一样了。

```
someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
  // 下面一行会报错，因为y没有声明
  y + 2;
}).catch(function(error) {
  console.log('carry on', error);
});
// oh no [ReferenceError: x is not defined]
// carry on [ReferenceError: y is not defined]
```

上面代码中，第二个 `catch` 方法用来捕获，前一个 `catch` 方法抛出的错误。

Promise.all()

`Promise.all` 方法用于将多个Promise实例，包装成一个新的Promise实例。

```
var p = Promise.all([p1, p2, p3]);
```

上面代码中，`Promise.all`方法接受一个数组作为参数，`p1`、`p2`、`p3`都是Promise对象的实例，如果不是，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.all`方法的参数可以不是数组，但必须具有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){
  // ...
});
```

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

下面是另一个例子。

```
const databasePromise = connectDatabase();

const booksPromise = databasePromise
  .then(findAllBooks);

const userPromise = databasePromise
  .then(getCurrentUser);

Promise.all([
  booksPromise,
  userPromise
])
  .then(([books, user]) => pickTopRecommendations(books, user));
```

上面代码中，`booksPromise`和`userPromise`是两个异步操作，只有等到它们的结果都返回了，才会触发`pickTopRecommendations`这个回调函数。

Promise.race()

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

```
var p = Promise.race([p1, p2, p3]);
```

上面代码中，只要`p1`、`p2`、`p3`之中有一个实例率先改变状态，`p`的状态就跟着改变。那个率先改变的Promise实例的返回值，就传递给`p`的回调函数。

`Promise.race`方法的参数与`Promise.all`方法一样，如果不是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。

下面是一个例子，如果指定时间内没有获得结果，就将Promise的状态变为 `reject`，否则变为 `resolve`。

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

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

Promise.resolve()

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

```
var jsPromise = Promise.resolve($.ajax('/whatever.json'));
```

上面代码将jQuery生成的 `deferred` 对象，转为一个新的Promise对象。

`Promise.resolve` 等价于下面的写法。

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

`Promise.resolve` 方法的参数分成四种情况。

(1) 参数是一个Promise实例

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

(2) 参数是一个 `thenable` 对象

`thenable` 对象指的是具有 `then` 方法的对象，比如下面这个对象。

```
let thenable = {
  then: function(resolve, reject) {
    resolve(42);
  }
};
```

`Promise.resolve` 方法会将这个对象转为Promise对象，然后就立即执行 `thenable` 对象的 `then` 方法。

```
let thenable = {
  then: function(resolve, reject) {
    resolve(42);
  }
};

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

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

(3) 参数不是具有 `then` 方法的对象，或根本就不是对象

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

```
var p = Promise.resolve('Hello');

p.then(function (s){
  console.log(s)
});

// Hello
```

上面代码生成一个新的Promise对象的实例 `p`。由于字符串 `Hello` 不属于异步操作（判断方法是它不是具有`then`方法的对象），返回Promise实例的状态从一生成就是 `Resolved`，所以回调函数会立即执行。`Promise.resolve` 方法的参数，会同时传给回调函数。

(4) 不带有任何参数

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

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。

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

```
setTimeout(function () {
  console.log('three');
}, 0);

Promise.resolve().then(function () {
  console.log('two');
});

console.log('one');

// one
// two
// three
```

上面代码中，`setTimeout(fn, 0)` 在下一轮“事件循环”开始时执行，`Promise.resolve()` 在本轮“事件循环”结束时执行，`console.log('one')` 则是立即执行，因此最先输出。

Promise.reject()

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

```
var p = Promise.reject('出错了');
// 等同于
var p = new Promise((resolve, reject) => 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。

注意，`Promise.reject()` 方法的参数，会原封不动地作为 `reject` 的理由，变成后续方法的参数。这一点与 `Promise.resolve` 方法不一致。

```
const thenable = {
  then(resolve, reject) {
    reject('出错了');
  }
};

Promise.reject(thenable)
  .catch(e => {
    console.log(e === thenable)
})
// true
```

上面代码中，`Promise.reject` 方法的参数是一个 `thenable` 对象，执行以后，后面 `catch` 方法的参数不是 `reject` 抛出的“出错了”这个字符串，而是 `thenable` 对象。

两个有用的附加方法

ES6的Promise API提供的方法不是很多，有些有用的方法可以自己部署。下面介绍如何部署两个不在ES6之中、但很有用的方法。

done()

Promise对象的回调链，不管以 `then` 方法或 `catch` 方法结尾，要是最后一个方法抛出错误，都有可能无法捕捉到（因为Promise内部的错误不会冒泡到全局）。因此，我们可以提供一个 `done` 方法，总是处于回调链的尾端，保证抛出任何可能出现的错误。

```
asyncFunc()
  .then(f1)
  .catch(r1)
  .then(f2)
  .done();
```

它的实现代码相当简单。

```
Promise.prototype.done = function (onFulfilled, onRejected) {
  this.then(onFulfilled, onRejected)
    .catch(function (reason) {
      // 抛出一个全局错误
      setTimeout(() => { throw reason }, 0);
    });
};
```

从上面代码可见，`done` 方法的使用，可以像 `then` 方法那样用，提供 `Fulfilled` 和 `Rejected` 状态的回调函数，也可以不提供任何参数。但不管怎样，`done` 都会捕捉到任何可能出现的错误，并向全局抛出。

finally()

`finally` 方法用于指定不管Promise对象最后状态如何，都会执行的操作。它与 `done` 方法的最大区别，它接受一个普通的回调函数作为参数，该函数不管怎样都必须执行。

下面是一个例子，服务器使用Promise处理请求，然后使用 `finally` 方法关掉服务器。

```
server.listen(0)
  .then(function () {
    // run test
  })
  .finally(server.stop);
```

它的实现也很简单。

```
Promise.prototype.finally = function (callback) {
  let P = this.constructor;
  return this.then(
    value => P.resolve(callback()).then(() => value),
    reason => P.resolve(callback()).then(() => { throw reason })
  );
};
```

上面代码中，不管前面的Promise是 `fulfilled` 还是 `rejected`，都会执行回调函数 `callback`。

应用

加载图片

我们可以将图片的加载写成一个 `Promise`，一旦加载完成，`Promise` 的状态就发生变化。

```
const preloadImage = function (path) {
  return new Promise(function (resolve, reject) {
    var image = new Image();
    image.onload = resolve;
    image.onerror = reject;
    image.src = path;
  });
};
```

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));
    });
  }

  go({ value: undefined });
}
```

```
    }, function (error) {
      return go(it.throw(error));
    });
}

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。

Promise.try()

实际开发中，经常遇到一种情况：不知道或者不想区分，函数 `f` 是同步函数还是异步操作，但是想用 `Promise` 来处理它。因为这样就可以不管 `f` 是否包含异步操作，都用 `then` 方法指定下一步流程，用 `catch` 方法处理 `f` 抛出的错误。一般就会采用下面的写法。

```
Promise.resolve().then(f)
```

上面的写法有一个缺点，就是如果 `f` 是同步函数，那么它会在本轮事件循环的末尾执行。

```
const f = () => console.log('now');
Promise.resolve().then(f);
console.log('next');
// next
// now
```

上面代码中，函数 `f` 是同步的，但是用 `Promise` 包装了以后，就变成异步执行了。

那么有没有一种方法，让同步函数同步执行，异步函数异步执行，并且让它们具有统一的 API 呢？回答是可以的，并且还有两种写法。第一种写法是用 `async` 函数来写。

```
const f = () => console.log('now');
(async () => f())();
console.log('next');
// now
// next
```

上面代码中，第二行是一个立即执行的匿名函数，会立即执行里面的 `async` 函数，因此如果 `f` 是同步的，就会得到同步的结果；如果 `f` 是异步的，就可以用 `then` 指定下一步，就像下面的写法。

```
(async () => f())()
.then(...)
```

需要注意的是，`async () => f()` 会吃掉 `f()` 抛出的错误。所以，如果想捕获错误，要使用 `promise.catch` 方法。

```
(async () => f())()
.then(...)
.catch(...)
```

第二种写法是使用 `new Promise()`。

```
const f = () => console.log('now');
(
  () => new Promise(
    resolve => resolve(f()))
)
```

```
)();
console.log('next');
// now
// next
```

上面代码也是使用立即执行的匿名函数，执行 `new Promise()`。这种情况下，同步函数也是同步执行的。

鉴于这是一个很常见的需求，所以现在有一个[提案](#)，提供 `Promise.try` 方法替代上面的写法。

```
const f = () => console.log('now');
Promise.try(f);
console.log('next');
// now
// next
```

事实上，`Promise.try` 存在已久，Promise 库 `Bluebird`、`Q` 和 `when`，早就提供了这个方法。

由于 `Promise.try` 为所有操作提供了统一的处理机制，所以如果想用 `then` 方法管理流程，最好都用 `Promise.try` 包装一下。这样有[许多好处](#)，其中一点就是可以更好地管理异常。

```
function getUsername(userId) {
  return database.users.get({id: userId})
    .then(function(user) {
      return user.name;
    });
}
```

上面代码中，`database.users.get()` 返回一个 Promise 对象，如果抛出异步错误，可以用 `catch` 方法捕获，就像下面这样写。

```
database.users.get({id: userId})
  .then(...)
  .catch(...)
```

但是 `database.users.get()` 可能还会抛出同步错误（比如数据库连接错误，具体要看实现方法），这时你就不得不用 `try...catch` 去捕获。

```
try {
  database.users.get({id: userId})
    .then(...)
    .catch(...)
} catch (e) {
  // ...
}
```

上面这样的写法就很笨拙了，这时就可以统一用 `promise.catch()` 捕获所有同步和异步的错误。

```
Promise.try(database.users.get({id: userId}))
  .then(...)
  .catch(...)
```

事实上，`Promise.try` 就是模拟 `try` 代码块，就像 `promise.catch` 模拟的是 `catch` 代码块。

异步操作和Async函数

异步编程对 JavaScript 语言太重要。Javascript 语言的执行环境是“单线程”的，如果没有异步编程，根本没法用，非卡死不可。本章主要介绍如何 Generator 函数完成异步操作。

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

Generator 函数将 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不是连续完成的，可以理解成该任务被人为分成两段，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。回调函数的英语名字 `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', 'utf-8', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

上面代码中，`readFile` 函数的第三个参数，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 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，第一段执行完以后，任务所在的上下文环境就已经结束了。在这以后抛出的错误，原来的上下文环境已经无法捕捉，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, 'utf-8', function (err, data) {
  fs.readFile(fileB, 'utf-8', function (err, data) {
    // ...
  });
});
```

不难想象，如果依次读取两个以上的文件，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。因为多个异步操作形成了强耦合，只要有一个操作需要修改，它的上层回调函数和下层回调函数，可能都要跟着修改。这种情况就称为“回调函数地狱”（callback hell）。

Promise 对象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，允许将回调函数的嵌套，改成链式调用。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 *asyncJob() {
  // ...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 Thunk = function (fileName){
  return function (callback){
    return fs.readFile(fileName, callback);
  };
};
```

```
var readFileThunk = Thunk(fileName);
readFileThunk(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。

```
// ES5版本
var Thunk = function(fn){
  return function () {
    var args = Array.prototype.slice.call(arguments);
    return function (callback) {
      args.push(callback);
      return fn.apply(this, args);
    }
  };
};

// ES6版本
var Thunk = function(fn) {
  return function (...args) {
    return function (callback) {
      return fn.call(this, ...args, callback);
    }
  };
};
```

使用上面的转换器，生成 `fs.readFile` 的`Thunk`函数。

```
var readFileThunk = Thunk(fs.readFile);
readFileThunk(fileA)(callback);
```

下面是另一个完整的例子。

```
function f(a, cb) {
  cb(a);
}
let ft = Thunk(f);

let log = console.log.bind(console);
ft(1)(log) // 1
```

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);
var print = console.log.bind(console);
ft(1, 2)(print);
// 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。

Generator函数可以自动执行。

```
function* gen() {
  // ...
}

var g = gen();
var res = g.next();

while(!res.done){
  console.log(res.value);
  res = g.next();
```

```
}
```

上面代码中，Generator函数 `gen` 会自动执行完所有步骤。

但是，这不适合异步操作。如果必须保证前一步执行完，才能执行后一步，上面的自动执行就不可行。这时，`Thunk`函数就能派上用处。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();
}

function* g() {
  // ...
}
```

```
run(g);
```

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。不管内部有多少个异步操作，直接把Generator函数传入 `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 g = function* (){
  var f1 = yield readFile('fileA');
  var f2 = yield readFile('fileB');
  // ...
  var fn = yield readFile('fileN');
};

run(g);
```

上面代码中，函数 `g` 封装了 `n` 个异步的读取文件操作，只要执行 `run` 函数，这些操作就会自动完成。这样一来，异步操作不仅可以写得像同步操作，而且一行代码就可以执行。

Thunk函数并不是Generator函数自动执行的唯一方案。因为自动执行的关键是，必须有一种机制，自动控制Generator函数的流程，接收和交还程序的执行权。回调函数可以做到这一点，Promise 对象也可以做到这一点。

co模块

基本用法

`co`模块是著名程序员TJ Holowaychuk于2013年6月发布的一个小工具，用于Generator函数的自动执行。

比如，有一个Generator函数，用于依次读取两个文件。

```
var gen = function* (){
  var f1 = yield readFile('/etc/fstab');
  var f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

`co`模块可以让你不用编写Generator函数的执行器。

```
var co = require('co');
co(gen);
```

上面代码中，Generator函数只要传入`co`函数，就会自动执行。

`co`函数返回一个Promise对象，因此可以用`then`方法添加回调函数。

```
co(gen).then(function (){
  console.log('Generator 函数执行完成');
});
```

上面代码中，等到Generator函数执行结束，就会输出一行提示。

co模块的原理

为什么 `co` 可以自动执行 Generator 函数？

前面说过，Generator 就是一个异步操作的容器。它的自动执行需要一种机制，当异步操作有了结果，能够自动交回执行权。

两种方法可以做到这一点。

(1) 回调函数。将异步操作包装成 Thunk 函数，在回调函数里面交回执行权。

(2) Promise 对象。将异步操作包装成 Promise 对象，用 `then` 方法交回执行权。

co 模块其实就是将两种自动执行器（Thunk 函数和 Promise 对象），包装成一个模块。使用 co 的前提条件是，Generator 函数的 `yield` 命令后面，只能是 Thunk 函数或 Promise 对象。

上一节已经介绍了基于 Thunk 函数的自动执行器。下面来看，基于 Promise 对象的自动执行器。这是理解 co 模块必须的。

基于 Promise 对象的自动执行

还是沿用上面的例子。首先，把 `fs` 模块的 `readFile` 方法包装成一个 Promise 对象。

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

var readFile = function (fileName){
  return new Promise(function (resolve, reject){
    fs.readFile(fileName, function(error, data){
      if (error) return reject(error);
      resolve(data);
    });
  });
};

var gen = function* (){
  var f1 = yield readFile('/etc/fstab');
  var f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};


```

然后，手动执行上面的 Generator 函数。

```
var g = gen();

g.next().value.then(function(data){
  g.next(data).value.then(function(data){
    g.next(data);
  });
});
```

手动执行其实就是用 `then` 方法，层层添加回调函数。理解了这一点，就可以写出一个自动执行器。

```
function run(gen){
  var g = gen();

  function next(data){
    var result = g.next(data);
    if (result.done) return result.value;
    result.value.then(function(data){
      next(data);
    });
  }

  next();
}
```

```
run(gen);
```

上面代码中，只要 Generator 函数还没执行到最后一步，`next` 函数就调用自身，以此实现自动执行。

co 模块的源码

co 就是上面那个自动执行器的扩展，它的源码只有几十行，非常简单。

首先，co 函数接受 Generator 函数作为参数，返回一个 Promise 对象。

```
function co(gen) {
  var ctx = this;

  return new Promise(function(resolve, reject) {
  });
}
```

在返回的 Promise 对象里面，co 先检查参数 `gen` 是否为 Generator 函数。如果是，就执行该函数，得到一个内部指针对象；如果不是就返回，并将 Promise 对象的状态改为 `resolved`。

```
function co(gen) {
  var ctx = this;

  return new Promise(function(resolve, reject) {
    if (typeof gen === 'function') gen = gen.call(ctx);
    if (!gen || typeof gen.next !== 'function') return resolve(gen);
  });
}
```

接着，co 将 Generator 函数的内部指针对象的 `next` 方法，包装成 `onFulfilled` 函数。这主要是为了能够捕捉抛出的错误。

```
function co(gen) {
  var ctx = this;

  return new Promise(function(resolve, reject) {
    if (typeof gen === 'function') gen = gen.call(ctx);
    if (!gen || typeof gen.next !== 'function') return resolve(gen);

    onFulfilled();
    function onFulfilled(res) {
      var ret;
      try {
        ret = gen.next(res);
      } catch (e) {
        return reject(e);
      }
      next(ret);
    }
  });
}
```

最后，就是关键的 `next` 函数，它会反复调用自身。

```
function next(ret) {
  if (ret.done) return resolve(ret.value);
  var value = toPromise.call(ctx, ret.value);
  if (value && isPromise(value)) return value.then(onFulfilled, onRejected);
  return onRejected(
    new TypeError(
```

```
'You may only yield a function, promise, generator, array, or object, '
+ 'but the following object was passed: "'"
+ String(ret.value)
+
)
);
}
```

上面代码中，`next` 函数的内部代码，一共只有四行命令。

第一行，检查当前是否为 Generator 函数的最后一步，如果是就返回。

第二行，确保每一步的返回值，是 Promise 对象。

第三行，使用 `then` 方法，为返回值加上回调函数，然后通过 `onFulfilled` 函数再次调用 `next` 函数。

第四行，在参数不符合要求的情况下（参数非 Thunk 函数和 Promise 对象），将 Promise 对象的状态改为 `rejected`，从而终止执行。

处理并发的异步操作

`co` 支持并发的异步操作，即允许某些操作同时进行，等到它们全部完成，才进行下一步。

这时，要把并发的操作都放在数组或对象里面，跟在 `yield` 语句后面。

```
// 数组的写法
co(function* () {
  var res = yield [
    Promise.resolve(1),
    Promise.resolve(2)
  ];
  console.log(res);
}).catch(onerror);

// 对象的写法
co(function* () {
  var res = yield {
    1: Promise.resolve(1),
    2: Promise.resolve(2),
  };
  console.log(res);
}).catch(onerror);
```

下面是另一个例子。

```
co(function* () {
  var values = [n1, n2, n3];
  yield values.map(somethingAsync);
});

function* somethingAsync(x) {
  // do something async
  return y
}
```

上面的代码允许并发三个 `somethingAsync` 异步操作，等到它们全部完成，才会进行下一步。

async 函数

含义

ES2017 标准提供了 `async` 函数，使得异步操作变得更加方便。

`async` 函数是什么？一句话，`async` 函数就是 Generator 函数的语法糖。前文有一个 Generator 函数，依次读取两个文件。

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

var readFile = function (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function(error, data) {
      if (error) reject(error);
      resolve(data);
    });
  });
};

var gen = function* (){
  var f1 = yield readFile('/etc/fstab');
  var f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};


```

写成 `async` 函数，就是下面这样。

```
var asyncReadFile = async function (){
  var f1 = await readFile('/etc/fstab');
  var f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};


```

一比较就会发现，`async` 函数就是将 Generator 函数的星号（`*`）替换成 `async`，将 `yield` 替换成 `await`，仅此而已。

`async` 函数对 Generator 函数的改进，体现在以下四点。

(1) 内置执行器。Generator 函数的执行必须靠执行器，所以才有了 `co` 模块，而 `async` 函数自带执行器。也就是说，`async` 函数的执行，与普通函数一模一样，只要一行。

```
var result = asyncReadFile();
```

上面的代码调用了 `asyncReadFile` 函数，然后它就会自动执行，输出最后结果。这完全不像 Generator 函数，需要调用 `next` 方法，或者用 `co` 模块，才能得到真正执行，得到最后结果。

(2) 更好的语义。`async` 和 `await`，比起星号和 `yield`，语义更清楚了。`async` 表示函数里有异步操作，`await` 表示紧跟在后面的表达式需要等待结果。

(3) 更广的适用性。`co` 模块约定，`yield` 命令后面只能是 Thunk 函数或 Promise 对象，而 `async` 函数的 `await` 命令后面，可以是 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）。

(4) 返回值是 Promise。`async` 函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用 `then` 方法指定下一步的操作。

进一步说，`async` 函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而 `await` 命令就是内部 `then` 命令的语法糖。

语法

`async` 函数的语法规则总体上比较简单，难点是错误处理机制。

(1) `async` 函数返回一个Promise对象。

`async` 函数内部 `return` 语句返回的值，会成为 `then` 方法回调函数的参数。

```
async function f() {
  return 'hello world';
}

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

上面代码中，函数 `f` 内部 `return` 命令返回的值，会被 `then` 方法回调函数接收到。

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

```
async function f() {
  throw new Error('出错了');
}

f().then(
  v => console.log(v),
  e => console.log(e)
)
// Error: 出错了
```

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

下面是一个例子。

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

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

(3) 正常情况下，`await` 命令后面是一个 Promise 对象。如果不是，会被转成一个立即 `resolve` 的 Promise 对象。

```
async function f() {
  return await 123;
}

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

上面代码中，`await` 命令的参数是数值 `123`，它被转成Promise对象，并立即 `resolve`。

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

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

f()
```

```
.then(v => console.log(v))
.catch(e => console.log(e))
// 出错了
```

注意，上面代码中，`await` 语句前面没有 `return`，但是 `reject` 方法的参数依然传入了 `catch` 方法的回调函数。这里如果在 `await` 前面加上 `return`，效果是一样的。

只要一个 `await` 语句后面的Promise变为 `reject`，那么整个 `async` 函数都会中断执行。

```
async function f() {
  await Promise.reject('出错了');
  await Promise.resolve('hello world'); // 不会执行
}
```

上面代码中，第二个 `await` 语句是不会执行的，因为第一个 `await` 语句状态变成了 `reject`。

为了避免这个问题，可以将第一个 `await` 放在 `try...catch` 结构里面，这样第二个 `await` 就会执行。

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

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

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

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

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

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

```
async function main() {
  try {
    var val1 = await firstStep();
    var val2 = await secondStep(val1);
    var val3 = await thirdStep(val1, val2);

    console.log('Final: ', val3);
  }
  catch (err) {
    console.error(err);
  }
}
```

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

```
async function f() {
  await new Promise(function (resolve, reject) {
```

```
        throw new Error('出错了');
    });
}

f()
.then(v => console.log(v))
.catch(e => console.log(e))
// Error: 出错了
```

上面代码中，`async` 函数 `f` 执行后，`await` 后面的Promise对象会抛出一个错误对象，导致 `catch` 方法的回调函数被调用，它的参数就是抛出的错误对象。具体的执行机制，可以参考后文的“`async`函数的实现”。

防止出错的方法，也是将其放在 `try...catch` 代码块之中。

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

async函数的实现

`async` 函数的实现，就是将 `Generator` 函数和自动执行器，包装在一个函数里。

```
async function fn(args){
  // ...
}

// 等同于

function fn(args){
  return spawn(function*() {
    // ...
  });
}
```

所有的 `async` 函数都可以写成上面的第二种形式，其中的 `spawn` 函数就是自动执行器。

下面给出 `spawn` 函数的实现，基本就是前文自动执行器的翻版。

```
function spawn(genF) {
  return new Promise(function(resolve, reject) {
    var gen = genF();
    function step(nextF) {
      try {
        var next = nextF();
      } catch(e) {
        return reject(e);
      }
      if(next.done) {
        return resolve(next.value);
      }
      Promise.resolve(next.value).then(function(v) {
        step(function() { return gen.next(v); });
      }, function(e) {
        step(function() { return gen.throw(e); });
      });
    }
  })
}
```

```
    step(function() { return gen.next(undefined); });
  });
}
```

async 函数的用法

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

下面是一个例子。

```
async function getStockPriceByName(name) {
  var symbol = await getStockSymbol(name);
  var stockPrice = await getStockPrice(symbol);
  return stockPrice;
}

getStockPriceByName('goog').then(function (result) {
  console.log(result);
});
```

上面代码是一个获取股票报价的函数，函数前面的 `async` 关键字，表明该函数内部有异步操作。调用该函数时，会立即返回一个 `Promise` 对象。

下面的例子，指定多少毫秒后输出一个值。

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

async function asyncPrint(value, ms) {
  await timeout(ms);
  console.log(value)
}

asyncPrint('hello world', 50);
```

上面代码指定50毫秒以后，输出 `hello world`。

Async函数有多种使用形式。

```
// 函数声明
async function foo() {}

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

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

// Class 的方法
class Storage {
  constructor() {
    this.cachePromise = caches.open('avatars');
  }

  async getAvatar(name) {
    const cache = await this.cachePromise;
    return cache.match(`/avatars/${name}.jpg`);
```

```
}

const storage = new Storage();
storage.getAvatar('jake').then(...);

// 箭头函数
const foo = async () => {};
```

注意点

第一点，`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);
    });
}
```

第二点，多个 `await` 命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

```
let foo = await getFoo();
let bar = await getBar();
```

上面代码中，`getFoo` 和 `getBar` 是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有 `getFoo` 完成以后，才会执行 `getBar`，完全可以让它们同时触发。

```
// 写法一
let [foo, bar] = await Promise.all([getFoo(), getBar()]);

// 写法二
let fooPromise = getFoo();
let barPromise = getBar();
let foo = await fooPromise;
let bar = await barPromise;
```

上面两种写法，`getFoo` 和 `getBar` 都是同时触发，这样就会缩短程序的执行时间。

第三点，`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 of 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。

实例：按顺序完成异步操作

实际开发中，经常遇到一组异步操作，需要按照顺序完成。比如，依次远程读取一组URL，然后按照读取的顺序输出结果。

Promise 的写法如下。

```
function logInOrder(urls) {
  // 远程读取所有URL
  const textPromises = urls.map(url => {
    return fetch(url).then(response => response.text());
  });

  // 按次序输出
  textPromises.reduce((chain, textPromise) => {
    return chain.then(() => textPromise)
      .then(text => console.log(text));
  }, Promise.resolve());
}
```

上面代码使用 `fetch` 方法，同时远程读取一组URL。每个 `fetch` 操作都返回一个 `Promise` 对象，放入 `textPromises` 数组。然后，`reduce` 方法依次处理每个 `Promise` 对象，然后使用 `then`，将所有 `Promise` 对象连起来，因此就可以依次输出结果。

这种写法不太直观，可读性比较差。下面是 `async` 函数实现。

```
async function logInOrder(urls) {
  for (const url of urls) {
    const response = await fetch(url);
    console.log(await response.text());
  }
}
```

上面代码确实大大简化，问题是所有远程操作都是继发。只有前一个URL返回结果，才会去读取下一个URL，这样做效率很差，非常浪费时间。我们需要的是并发发出远程请求。

```
async function logInOrder(urls) {
  // 并发读取远程URL
  const textPromises = urls.map(async url => {
    const response = await fetch(url);
    return response.text();
  });

  // 按次序输出
  for (const textPromise of textPromises) {
    console.log(await textPromise);
  }
}
```

上面代码中，虽然 `map` 方法的参数是 `async` 函数，但它是并发执行的，因为只有 `async` 函数内部是继发执行，外部不受影响。后面的 `for..of` 循环内部使用了 `await`，因此实现了按顺序输出。

异步遍历器

《遍历器》一章说过，`Iterator` 接口是一种数据遍历的协议，只要调用遍历器对象的 `next` 方法，就会得到一个对象，表示当前遍历指针所在的那个位置的信息。`next` 方法返回的对象的结构是 `{value, done}`，其中 `value` 表示当前的数据的值，`done` 是一个布尔值，表示遍历是否结束。

这里隐含着一个规定，`next` 方法必须是同步的，只要调用就必须立刻返回值。也就是说，一旦执行 `next` 方法，就必须同步地得到 `value` 和 `done` 这两个属性。如果遍历指针正好指向同步操作，当然没有问题，但对于异步操作，就不

太合适了。目前的解决方法是，Generator 函数里面的异步操作，返回一个 Thunk 函数或者 Promise 对象，即 `value` 属性是一个 Thunk 函数或者 Promise 对象，等待以后返回真正的值，而 `done` 属性则还是同步产生的。

目前，有一个提案，为异步操作提供原生的遍历器接口，即 `value` 和 `done` 这两个属性都是异步产生，这称为“异步遍历器”（Async Iterator）。

异步遍历的接口

异步遍历器的最大的语法特点，就是调用遍历器的 `next` 方法，返回的是一个 Promise 对象。

```
asyncIterator
  .next()
  .then(
    ({ value, done }) => /* ... */
  );
```

上面代码中，`asyncIterator` 是一个异步遍历器，调用 `next` 方法以后，返回一个 Promise 对象。因此，可以使用 `then` 方法指定，这个 Promise 对象的状态变为 `resolve` 以后的回调函数。回调函数的参数，则是一个具有 `value` 和 `done` 两个属性的对象，这个跟同步遍历器是一样的。

我们知道，一个对象的同步遍历器的接口，部署在 `Symbol.iterator` 属性上面。同样地，对象的异步遍历器接口，部署在 `Symbol.asyncIterator` 属性上面。不管是什么样的对象，只要它的 `Symbol.asyncIterator` 属性有值，就表示应该对它进行异步遍历。

下面是一个异步遍历器的例子。

```
const asyncIterable = createAsyncIterable(['a', 'b']);
const asyncIterator = asyncIterable[Symbol.asyncIterator]();

asyncIterator
  .next()
  .then(iterResult1 => {
    console.log(iterResult1); // { value: 'a', done: false }
    return asyncIterator.next();
})
  .then(iterResult2 => {
    console.log(iterResult2); // { value: 'b', done: false }
    return asyncIterator.next();
})
  .then(iterResult3 => {
    console.log(iterResult3); // { value: undefined, done: true }
});
```

上面代码中，异步遍历器其实返回了两次值。第一次调用的时候，返回一个 Promise 对象；等到 Promise 对象 `resolve` 了，再返回一个表示当前数据成员信息的对象。这就是说，异步遍历器与同步遍历器最终行为是一致的，只是会先返回 Promise 对象，作为中介。

由于异步遍历器的 `next` 方法，返回的是一个 Promise 对象。因此，可以把它放在 `await` 命令后面。

```
async function f() {
  const asyncIterable = createAsyncIterable(['a', 'b']);
  const asyncIterator = asyncIterable[Symbol.asyncIterator]();
  console.log(await asyncIterator.next());
  // { value: 'a', done: false }
  console.log(await asyncIterator.next());
  // { value: 'b', done: false }
  console.log(await asyncIterator.next());
  // { value: undefined, done: true }
}
```

上面代码中，`next` 方法用 `await` 处理以后，就不必使用 `then` 方法了。整个流程已经很接近同步处理了。

注意，异步遍历器的 `next` 方法是可以连续调用的，不必等到上一步产生的Promise对象 `resolve` 以后再调用。这种情况下，`next` 方法会累积起来，自动按照每一步的顺序运行下去。下面是一个例子，把所有的 `next` 方法放在 `Promise.all` 方法里面。

```
const asyncGenObj = createAsyncIterable(['a', 'b']);
const [{value: v1}, {value: v2}] = await Promise.all([
  asyncGenObj.next(), asyncGenObj.next()
]);
console.log(v1, v2); // a b
```

另一种用法是一次性调用所有的 `next` 方法，然后 `await` 最后一步操作。

```
const writer = openFile('someFile.txt');
writer.next('hello');
writer.next('world');
await writer.return();
```

for await...of

前面介绍过，`for...of` 循环用于遍历同步的 `Iterator` 接口。新引入的 `for await...of` 循环，则是用于遍历异步的 `Iterator` 接口。

```
async function f() {
  for await (const x of createAsyncIterable(['a', 'b'])) {
    console.log(x);
  }
}
// a
// b
```

上面代码中，`createAsyncIterable()` 返回一个异步遍历器，`for...of` 循环自动调用这个遍历器的 `next` 方法，会得到一个Promise对象。`await` 用来处理这个Promise对象，一旦 `resolve`，就把得到的值（`x`）传入 `for...of` 的循环体。

`for await...of` 循环的一个用途，是部署了 `asyncIterable` 操作的异步接口，可以直接放入这个循环。

```
let body = '';
for await(const data of req) body += data;
const parsed = JSON.parse(body);
console.log('got', parsed);
```

上面代码中，`req` 是一个 `asyncIterable` 对象，用来异步读取数据。可以看到，使用 `for await...of` 循环以后，代码会非常简洁。

如果 `next` 方法返回的Promise对象被 `reject`，那么就要用 `try...catch` 捕捉。

```
async function () {
  try {
    for await (const x of createRejectingIterable()) {
      console.log(x);
    }
  } catch (e) {
    console.error(e);
  }
}
```

注意，`for await...of` 循环也可以用于同步遍历器。

```
(async function () {
  for await (const x of ['a', 'b']) {
    console.log(x);
  }
})();
// a
// b
```

异步Generator函数

就像 Generator 函数返回一个同步遍历器对象一样，异步 Generator 函数的作用，是返回一个异步遍历器对象。

在语法上，异步 Generator 函数就是 `async` 函数与 Generator 函数的结合。

```
async function* readLines(path) {
  let file = await fileOpen(path);

  try {
    while (!file.EOF) {
      yield await file.readLine();
    }
  } finally {
    await file.close();
  }
}
```

上面代码中，异步操作前面使用 `await` 关键字标明，即 `await` 后面的操作，应该返回Promise对象。凡是使用 `yield` 关键字的地方，就是 `next` 方法的停下来的地方，它后面的表达式的值（即 `await file.readLine()` 的值），会作为 `next()` 返回对象的 `value` 属性，这一点是于同步Generator函数一致的。

可以像下面这样，使用上面代码定义的异步Generator函数。

```
for await (const line of readLines(filePath)) {
  console.log(line);
}
```

异步Generator函数可以与 `for await...of` 循环结合起来使用。

```
async function* prefixLines(asyncIterable) {
  for await (const line of asyncIterable) {
    yield '> ' + line;
  }
}
```

`yield` 命令依然是立刻返回的，但是返回的是一个Promise对象。

```
async function* asyncGenerator() {
  console.log('Start');
  const result = await doSomethingAsync(); // (A)
  yield 'Result: ' + result; // (B)
  console.log('Done');
}
```

上面代码中，调用 `next` 方法以后，会在 `B` 处暂停执行，`yield` 命令立刻返回一个Promise对象。这个Promise对象不同于 `A` 处 `await` 命令后面的那个Promise对象。主要有两点不同，一是 `A` 处的Promise对象 `resolve` 以后产生的值，会放入 `result` 变量；二是 `B` 处的Promise对象 `resolve` 以后产生的值，是表达式 `'Result: ' + result` 的值；二是 `A` 处的Promise对象一定先于 `B` 处的Promise对象 `resolve`。

如果异步Generator函数抛出错误，会被Promise对象 `reject`，然后抛出的错误被 `catch` 方法捕获。

```
async function* asyncGenerator() {
  throw new Error('Problem!');
}

asyncGenerator()
.next()
.catch(err => console.log(err)); // Error: Problem!
```

注意，普通的 `async` 函数返回的是一个Promise对象，而异步Generator函数返回的是一个异步Iterator对象。基本上，可以这样理解，`async` 函数和异步Generator函数，是封装异步操作的两种方法，都用来达到同一种目的。区别在于，前者自带执行器，后者通过 `for await...of` 执行，或者自己编写执行器。下面就是一个异步Generator函数的执行器。

```
async function takeAsync(asyncIterable, count=Infinity) {
  const result = [];
  const iterator = asyncIterable[Symbol.asyncIterator]();
  while (result.length < count) {
    const {value,done} = await iterator.next();
    if (done) break;
    result.push(value);
  }
  return result;
}
```

上面代码中，异步Generator函数产生的异步遍历器，会通过 `while` 循环自动执行，每当 `await iterator.next()` 完成，就会进入下一轮循环。

下面是这个自动执行器的一个使用实例。

```
async function f() {
  async function* gen() {
    yield 'a';
    yield 'b';
    yield 'c';
  }

  return await takeAsync(gen());
}

f().then(function (result) {
  console.log(result); // ['a', 'b', 'c']
})
```

异步Generator函数出现以后，JavaScript就有了四种函数形式：普通函数、`async` 函数、Generator函数和异步Generator函数。请注意区分每种函数的不同之处。

最后，同步的数据结构，也可以使用异步Generator函数。

```
async function* createAsyncIterable(syncIterable) {
  for (const elem of syncIterable) {
    yield elem;
  }
}
```

上面代码中，由于没有异步操作，所以也就没有使用 `await` 关键字。

yield* 语句

`yield*` 语句也可以跟一个异步遍历器。

```
async function* gen1() {
  yield 'a';
  yield 'b';
  return 2;
}

async function* gen2() {
  const result = yield* gen1();
}
```

上面代码中，`gen2` 函数里面的 `result` 变量，最后的值是 `2`。

与同步Generator函数一样，`for await...of` 循环会展开 `yield*`。

```
(async function () {
  for await (const x of gen2()) {
    console.log(x);
  }
})();
// a
// b
```

Class

Class基本语法

概述

JavaScript语言的传统方法是通过构造函数，定义并生成新对象。下面是一个例子。

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

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

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

上面这种写法跟传统的面向对象语言（比如C++和Java）差异很大，很容易让新学习这门语言的程序员感到困惑。

ES6提供了更接近传统语言的写法，引入了Class（类）这个概念，作为对象的模板。通过 `class` 关键字，可以定义类。基本上，ES6的 `class` 可以看作只是一个语法糖，它的绝大部分功能，ES5都可以做到，新的 `class` 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。上面的代码用ES6的“类”改写，就是下面这样。

```
// 定义类
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
}
}
```

上面代码定义了一个“类”，可以看到里面有一个 `constructor` 方法，这就是构造方法，而 `this` 关键字则代表实例对象。也就是说，ES5的构造函数 `Point`，对应ES6的 `Point` 类的构造方法。

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

ES6的类，完全可以看作构造函数的另一种写法。

```
class Point {  
  // ...  
}  
  
typeof Point // "function"  
Point === Point.prototype.constructor // true
```

上面代码表明，类的数据类型就是函数，类本身就指向构造函数。

使用的时候，也是直接对类使用 `new` 命令，跟构造函数的用法完全一致。

```
class Bar {  
  doStuff() {  
    console.log('stuff');  
  }  
}  
  
var b = new Bar();  
b.doStuff() // "stuff"
```

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

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

在类的实例上面调用方法，其实就是调用原型上的方法。

```
class B {}  
let b = new B();  
  
b.constructor === B.prototype.constructor // true
```

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

由于类的方法都定义在 `prototype` 对象上面，所以类的新方法可以添加在 `prototype` 对象上面。`Object.assign` 方

法可以很方便地一次向类添加多个方法。

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

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

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

另外，类的内部所有定义的方法，都是不可枚举的（non-enumerable）。

```
class Point {  
  constructor(x, y) {  
    // ...  
  }  
  
  toString() {  
    // ...  
  }  
}  
  
Object.keys(Point.prototype)  
// []  
Object.getOwnPropertyNames(Point.prototype)  
// ["constructor", "toString"]
```

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

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

上面代码采用ES5的写法，`toString` 方法就是可枚举的。

类的属性名，可以采用表达式。

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

```
}
```

上面代码中，`Square` 类的方法名 `getArea`，是从表达式得到的。

constructor方法

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

```
constructor() {}
```

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

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

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

类的构造函数，不使用 `new` 是没法调用的，会报错。这是它跟普通构造函数的一个主要区别，后者不用 `new` 也可以执行。

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

类的实例对象

生成类的实例对象的写法，与ES5完全一样，也是使用 `new` 命令。如果忘记加上 `new`，像函数那样调用 `Class`，将会报错。

```
// 报错  
var point = Point(2, 3);  
  
// 正确  
var point = new Point(2, 3);
```

与ES5一样，实例的属性除非显式定义在其本身（即定义在 `this` 对象上），否则都是定义在原型上（即定义在 `class` 上）。

```
// 定义类  
class Point {  
  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  toString() {
```

```
        return '(' + this.x + ', ' + this.y + ')';
    }

}

var point = new Point(2, 3);

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

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

上面代码中，`x` 和 `y` 都是实例对象 `point` 自身的属性（因为定义在 `this` 变量上），所以 `hasOwnProperty` 方法返回 `true`，而 `toString` 是原型对象的属性（因为定义在 `Point` 类上），所以 `hasOwnProperty` 方法返回 `false`。这些都与ES5的行为保持一致。

与ES5一样，类的所有实例共享一个原型对象。

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

p1.__proto__ === p2.__proto__
//true
```

上面代码中，`p1` 和 `p2` 都是`Point`的实例，它们的原型都是`Point.prototype`，所以 `__proto__` 属性是相等的。

这也意味着，可以通过实例的 `__proto__` 属性为Class添加方法。

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

p1.__proto__.printName = function () { return 'Oops' };

p1.printName() // "Oops"
p2.printName() // "Oops"

var p3 = new Point(4,2);
p3.printName() // "Oops"
```

上面代码在 `p1` 的原型上添加了一个 `printName` 方法，由于 `p1` 的原型就是 `p2` 的原型，因此 `p2` 也可以调用这个方法。而且，此后新建的实例 `p3` 也可以调用这个方法。这意味着，使用实例的 `__proto__` 属性改写原型，必须相当谨慎，不推荐使用，因为这会改变Class的原始定义，影响到所有实例。

不存在变量提升

Class不存在变量提升（hoist），这一点与ES5完全不同。

```
new Foo(); // ReferenceError
class Foo {}
```

上面代码中，`Foo` 类使用在前，定义在后，这样会报错，因为ES6不会把类的声明提升到代码头部。这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

```
{
  let Foo = class {};
  class Bar extends Foo {
  }
}
```

上面的代码不会报错，因为 `Bar` 继承 `Foo` 的时候，`Foo` 已经有定义了。但是，如果存在 `class` 的提升，上面代码就会报错，因为 `class` 会被提升到代码头部，而 `let` 命令是不提升的，所以导致 `Bar` 继承 `Foo` 的时候，`Foo` 还没有定义。

Class表达式

与函数一样，类也可以使用表达式的形式定义。

```
const MyClass = class Me {
  getClassClassName() {
    return Me.name;
  }
};
```

上面代码使用表达式定义了一个类。需要注意的是，这个类的名字是 `MyClass` 而不是 `Me`，`Me` 只在Class的内部代码可用，指代当前类。

```
let inst = new MyClass();
inst.getClassName() // Me
Me.name // ReferenceError: Me is not defined
```

上面代码表示，`Me` 只在Class内部有定义。

如果类的内部没用到的话，可以省略 `Me`，也就是可以写成下面的形式。

```
const MyClass = class { /* ... */ };
```

采用Class表达式，可以写出立即执行的Class。

```
let person = new class {
  constructor(name) {
    this.name = name;
  }

  sayName() {
    console.log(this.name);
  }
}('张三');

person.sayName(); // "张三"
```

上面代码中，`person` 是一个立即执行的类的实例。

私有方法

私有方法是常见需求，但ES6不提供，只能通过变通方法模拟实现。

一种做法是在命名上加以区别。

```
class Widget {

  // 公有方法
  foo (baz) {
    this._bar(baz);
  }

  // 私有方法
  _bar(baz) {
    return this.snaf = baz;
  }
}
```

```
}

// ...
}
```

上面代码中，`_bar` 方法前面的下划线，表示这是一个只限于内部使用的私有方法。但是，这种命名是不保险的，在类的外部，还是可以调用到这个方法。

另一种方法就是索性将私有方法移出模块，因为模块内部的所有方法都是对外可见的。

```
class Widget {
  foo (baz) {
    bar.call(this, baz);
  }

  // ...
}

function bar(baz) {
  return this.snaf = baz;
}
```

上面代码中，`foo` 是公有方法，内部调用了`bar.call(this, baz)`。这使得`bar` 实际上成为了当前模块的私有方法。

还有一种方法是利用`Symbol` 值的唯一性，将私有方法的名字命名为一个`Symbol` 值。

```
const bar = Symbol('bar');
const snaf = Symbol('snaf');

export default class myClass{

  // 公有方法
  foo(baz) {
    this[bar](baz);
  }

  // 私有方法
  [bar](baz) {
    return this[snaf] = baz;
  }

  // ...
};
```

上面代码中，`bar` 和`snaf` 都是`Symbol` 值，导致第三方无法获取到它们，因此达到了私有方法和私有属性的效果。

this的指向

类的方法内部如果含有`this`，它默认指向类的实例。但是，必须非常小心，一旦单独使用该方法，很可能报错。

```
class Logger {
  printName(name = 'there') {
    this.print(`Hello ${name}`);
  }

  print(text) {
    console.log(text);
  }
}

const logger = new Logger();
```

```
const { printName } = logger;
printName(); // TypeError: Cannot read property 'print' of undefined
```

上面代码中，`printName`方法中的`this`，默认指向`Logger`类的实例。但是，如果将这个方法提取出来单独使用，`this`会指向该方法运行时所在的环境，因为找不到`print`方法而导致报错。

一个比较简单的解决方法是，在构造方法中绑定`this`，这样就不会找不到`print`方法了。

```
class Logger {
  constructor() {
    this.printName = this.printName.bind(this);
  }

  // ...
}
```

另一种解决方法是使用箭头函数。

```
class Logger {
  constructor() {
    this.printName = (name = 'there') => {
      this.print(`Hello ${name}`);
    };
  }

  // ...
}
```

还有一种解决方法是使用`Proxy`，获取方法的时候，自动绑定`this`。

```
function selfish (target) {
  const cache = new WeakMap();
  const handler = {
    get (target, key) {
      const value = Reflect.get(target, key);
      if (typeof value !== 'function') {
        return value;
      }
      if (!cache.has(value)) {
        cache.set(value, value.bind(target));
      }
      return cache.get(value);
    }
  };
  const proxy = new Proxy(target, handler);
  return proxy;
}

const logger = selfish(new Logger());
```

严格模式

类和模块的内部，默认就是严格模式，所以不需要使用`use strict`指定运行模式。只要你的代码写在类或模块之中，就只有严格模式可用。

考虑到未来所有的代码，其实都是运行在模块之中，所以ES6实际上把整个语言升级到了严格模式。

name属性

由于本质上，ES6的类只是ES5的构造函数的一层包装，所以函数的许多特性都被`Class`继承，包括`name`属性。

```
class Point {}  
Point.name // "Point"
```

`name` 属性总是返回紧跟在 `class` 关键字后面的类名。

Class的继承

基本用法

Class之间可以通过 `extends` 关键字实现继承，这比ES5的通过修改原型链实现继承，要清晰和方便很多。

```
class ColorPoint extends Point {}
```

上面代码定义了一个 `ColorPoint` 类，该类通过 `extends` 关键字，继承了 `Point` 类的所有属性和方法。但是由于没有部署任何代码，所以这两个类完全一样，等于复制了一个 `Point` 类。下面，我们在 `ColorPoint` 内部加上代码。

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

上面代码中，`constructor` 方法和 `toString` 方法之中，都出现了 `super` 关键字，它在这里表示父类的构造函数，用来新建父类的 `this` 对象。

子类必须在 `constructor` 方法中调用 `super` 方法，否则新建实例时会报错。这是因为子类没有自己的 `this` 对象，而是继承父类的 `this` 对象，然后对其进行加工。如果不调用 `super` 方法，子类就得不到 `this` 对象。

```
class Point { /* ... */ }  
  
class ColorPoint extends Point {  
  constructor() {  
  }  
}  
  
let cp = new ColorPoint(); // ReferenceError
```

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

ES5的继承，实质是先创造子类的实例对象 `this`，然后再将父类的方法添加到 `this` 上面（`Parent.apply(this)`）。ES6的继承机制完全不同，实质是先创造父类的实例对象 `this`（所以必须先调用 `super` 方法），然后再用子类的构造函数修改 `this`。

如果子类没有定义 `constructor` 方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有 `constructor` 方法。

```
constructor(...args) {  
  super(...args);  
}
```

另一个需要注意的地方是，在子类的构造函数中，只有调用 `super` 之后，才可以使用 `this` 关键字，否则会报错。这是因为子类实例的构建，是基于对父类实例加工，只有 `super` 方法才能返回父类实例。

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

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

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

下面是生成子类实例的代码。

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

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

上面代码中，实例对象 `cp` 同时是 `ColorPoint` 和 `Point` 两个类的实例，这与ES5的行为完全一致。

类的`prototype`属性和`__proto__`属性

大多数浏览器的ES5实现之中，每一个对象都有 `__proto__` 属性，指向对应的构造函数的`prototype`属性。Class作为构造函数的语法糖，同时有`prototype`属性和 `__proto__` 属性，因此同时存在两条继承链。

- (1) 子类的 `__proto__` 属性，表示构造函数的继承，总是指向父类。
- (2) 子类 `prototype` 属性的 `__proto__` 属性，表示方法的继承，总是指向父类的 `prototype` 属性。

```
class A {}

class B extends A {}

B.__proto__ === A // true
B.prototype.__proto__ === A.prototype // true
```

上面代码中，子类 `B` 的 `__proto__` 属性指向父类 `A`，子类 `B` 的 `prototype` 属性的 `__proto__` 属性指向父类 `A` 的 `prototype` 属性。

这样的结果是因为，类的继承是按照下面的模式实现的。

```
class A {}

class B {}

// B的实例继承A的实例
Object.setPrototypeOf(B.prototype, A.prototype);
const b = new B();

// B的实例继承A的静态属性
```

```
Object.setPrototypeOf(B, A);
const b = new B();
```

《对象的扩展》一章给出过 `Object.setPrototypeOf` 方法的实现。

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

因此，就得到了上面的结果。

```
Object.setPrototypeOf(B.prototype, A.prototype);
// 等同于
B.prototype.__proto__ = A.prototype;

Object.setPrototypeOf(B, A);
// 等同于
B.__proto__ = A;
```

这两条继承链，可以这样理解：作为一个对象，子类（`B`）的原型（`__proto__` 属性）是父类（`A`）；作为一个构造函数，子类（`B`）的原型（`prototype` 属性）是父类的实例。

```
Object.create(A.prototype);
// 等同于
B.prototype.__proto__ = A.prototype;
```

Extends 的继承目标

`extends` 关键字后面可以跟多种类型的值。

```
class B extends A {
```

上面代码的 `A`，只要是一个有 `prototype` 属性的函数，就能被 `B` 继承。由于函数都有 `prototype` 属性（除了 `Function.prototype` 函数），因此 `A` 可以是任意函数。

下面，讨论三种特殊情况。

第一种特殊情况，子类继承`Object`类。

```
class A extends Object {
```

```
A.__proto__ === Object // true
A.prototype.__proto__ === Object.prototype // true
```

这种情况下，`A` 其实就是构造函数 `Object` 的复制，`A` 的实例就是 `Object` 的实例。

第二种特殊情况，不存在任何继承。

```
class A {
```

```
A.__proto__ === Function.prototype // true
A.prototype.__proto__ === Object.prototype // true
```

这种情况下，`A`作为一个基类（即不存在任何继承），就是一个普通函数，所以直接继承 `Funciton.prototype`。但是，`A` 调用后返回一个空对象（即 `Object` 实例），所以 `A.prototype.__proto__` 指向构造函数（`Object`）的

`prototype` 属性。

第三种特殊情况，子类继承 `null`。

```
class A extends null {  
}  
  
A.__proto__ === Function.prototype // true  
A.prototype.__proto__ === undefined // true
```

这种情况与第二种情况非常像。`A` 也是一个普通函数，所以直接继承 `Function.prototype`。但是，`A` 调用后返回的对象不继承任何方法，所以它的 `__proto__` 指向 `Function.prototype`，即实质上执行了下面的代码。

```
class C extends null {  
  constructor() { return Object.create(null); }  
}
```

Object.getPrototypeOf()

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

```
Object.getPrototypeOf(ColorPoint) === Point  
// true
```

因此，可以使用这个方法判断，一个类是否继承了另一个类。

super 关键字

`super` 这个关键字，既可以当作函数使用，也可以当作对象使用。在这两种情况下，它的用法完全不同。

第一种情况，`super` 作为函数调用时，代表父类的构造函数。ES6 要求，子类的构造函数必须执行一次 `super` 函数。

```
class A {}  
  
class B extends A {  
  constructor() {  
    super();  
  }  
}
```

上面代码中，子类 `B` 的构造函数之中的 `super()`，代表调用父类的构造函数。这是必须的，否则 JavaScript 引擎会报错。

注意，`super` 虽然代表了父类 `A` 的构造函数，但是返回的是子类 `B` 的实例，即 `super` 内部的 `this` 指的是 `B`，因此 `super()` 在这里相当于 `A.prototype.constructor.call(this)`。

```
class A {  
  constructor() {  
    console.log(new.target.name);  
  }  
}  
class B extends A {  
  constructor() {  
    super();  
  }  
}  
new A() // A  
new B() // B
```

上面代码中，`new.target` 指向当前正在执行的函数。可以看到，在`super()` 执行时，它指向的是子类 `B` 的构造函数，而不是父类 `A` 的构造函数。也就是说，`super()` 内部的 `this` 指向的是 `B`。

作为函数时，`super()` 只能用在子类的构造函数之中，用在其他地方就会报错。

```
class A {}

class B extends A {
  m() {
    super(); // 报错
  }
}
```

上面代码中，`super()` 用在 `B` 类的 `m` 方法之中，就会造成句法错误。

第二种情况，`super` 作为对象时，指向父类的原型对象。

```
class A {
  p() {
    return 2;
  }
}

class B extends A {
  constructor() {
    super();
    console.log(super.p()); // 2
  }
}

let b = new B();
```

上面代码中，子类 `B` 当中的 `super.p()`，就是将 `super` 当作一个对象使用。这时，`super` 指向 `A.prototype`，所以 `super.p()` 就相当于 `A.prototype.p()`。

这里需要注意，由于 `super` 指向父类的原型对象，所以定义在父类实例上的方法或属性，是无法通过 `super` 调用的。

```
class A {
  constructor() {
    this.p = 2;
  }
}

class B extends A {
  get m() {
    return super.p;
  }
}

let b = new B();
b.m // undefined
```

上面代码中，`p` 是父类 `A` 实例的属性，`super.p` 就引用不到它。

如果属性定义在父类的原型对象上，`super` 就可以取到。

```
class A {}
A.prototype.x = 2;

class B extends A {
  constructor() {
    super();
  }
}
```

```
        console.log(super.x) // 2
    }
}

let b = new B();
```

上面代码中，属性 `x` 是定义在 `A.prototype` 上面的，所以 `super.x` 可以取到它的值。

ES6 规定，通过 `super` 调用父类的方法时，`super` 会绑定子类的 `this`。

```
class A {
  constructor() {
    this.x = 1;
  }
  print() {
    console.log(this.x);
  }
}

class B extends A {
  constructor() {
    super();
    this.x = 2;
  }
  m() {
    super.print();
  }
}

let b = new B();
b.m() // 2
```

上面代码中，`super.print()` 虽然调用的是 `A.prototype.print()`，但是 `A.prototype.print()` 会绑定子类 `B` 的 `this`，导致输出的是 `2`，而不是 `1`。也就是说，实际上执行的是 `super.print.call(this)`。

由于绑定子类的 `this`，所以如果通过 `super` 对某个属性赋值，这时 `super` 就是 `this`，赋值的属性会变成子类实例的属性。

```
class A {
  constructor() {
    this.x = 1;
  }
}

class B extends A {
  constructor() {
    super();
    this.x = 2;
    super.x = 3;
    console.log(super.x); // undefined
    console.log(this.x); // 3
  }
}

let b = new B();
```

上面代码中，`super.x` 赋值为 `3`，这时等同于对 `this.x` 赋值为 `3`。而当读取 `super.x` 的时候，读的是 `A.prototype.x`，所以返回 `undefined`。

注意，使用 `super` 的时候，必须显式指定是作为函数、还是作为对象使用，否则会报错。

```
class A {}
```

```
class B extends A {  
    constructor() {  
        super();  
        console.log(super); // 报错  
    }  
}
```

上面代码中，`console.log(super)` 当中的 `super`，无法看出是作为函数使用，还是作为对象使用，所以 JavaScript 引擎解析代码的时候就会报错。这时，如果能清晰地表明 `super` 的数据类型，就不会报错。

```
class A {}  
  
class B extends A {  
    constructor() {  
        super();  
        console.log(super.valueOf() instanceof B); // true  
    }  
}  
  
let b = new B();
```

上面代码中，`super.valueOf()` 表明 `super` 是一个对象，因此就不会报错。同时，由于 `super` 绑定 `B` 的 `this`，所以 `super.valueOf()` 返回的是一个 `B` 的实例。

最后，由于对象总是继承其他对象的，所以可以在任意一个对象中，使用 `super` 关键字。

```
var obj = {  
    toString() {  
        return "MyObject: " + super.toString();  
    }  
};  
  
obj.toString(); // MyObject: [object Object]
```

实例的`__proto__`属性

子类实例的`__proto__`属性的`__proto__`属性，指向父类实例的`__proto__`属性。也就是说，子类的原型的原型，是父类的原型。

```
var p1 = new Point(2, 3);  
var p2 = new ColorPoint(2, 3, 'red');  
  
p2.__proto__ === p1.__proto__ // false  
p2.__proto__.__proto__ === p1.__proto__ // true
```

上面代码中，`ColorPoint` 继承了 `Point`，导致前者原型的原型是后者的原型。

因此，通过子类实例的 `__proto__.__proto__` 属性，可以修改父类实例的行为。

```
p2.__proto__.__proto__.printName = function () {  
    console.log('Ha');  
};  
  
p1.printName() // "Ha"
```

上面代码在 `ColorPoint` 的实例 `p2` 上向 `Point` 类添加方法，结果影响到了 `Point` 的实例 `p1`。

原生构造函数的继承

原生构造函数是指语言内置的构造函数，通常用来生成数据结构。ECMAScript的原生构造函数大致有下面这些。

- Boolean()
- Number()
- String()
- Array()
- Date()
- Function()
- RegExp()
- Error()
- Object()

以前，这些原生构造函数是无法继承的，比如，不能自己定义一个 `Array` 的子类。

```
function MyArray() {
  Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
  constructor: {
    value: MyArray,
    writable: true,
    configurable: true,
    enumerable: true
  }
});
```

上面代码定义了一个继承`Array`的 `MyArray` 类。但是，这个类的行为与 `Array` 完全不一致。

```
var colors = new MyArray();
colors[0] = "red";
colors.length // 0

colors.length = 0;
colors[0] // "red"
```

之所以会发生这种情况，是因为子类无法获得原生构造函数的内部属性，通过 `Array.apply()` 或者分配给原型对象都不行。原生构造函数会忽略 `apply` 方法传入的 `this`，也就是说，原生构造函数的 `this` 无法绑定，导致拿不到内部属性。

ES5是先新建子类的实例对象 `this`，再将父类的属性添加到子类上，由于父类的内部属性无法获取，导致无法继承原生的构造函数。比如，`Array`构造函数有一个内部属性 `[[DefineOwnProperty]]`，用来定义新属性时，更新 `length` 属性，这个内部属性无法在子类获取，导致子类的 `length` 属性行为不正常。

下面的例子中，我们想让一个普通对象继承 `Error` 对象。

```
var e = {};

Object.getOwnPropertyNames(Error.call(e))
// [ 'stack' ]

Object.getOwnPropertyNames(e)
// []
```

上面代码中，我们想通过 `Error.call(e)` 这种写法，让普通对象 `e` 具有 `Error` 对象的实例属性。但是，`Error.call()` 完全忽略传入的第一个参数，而是返回一个新对象，`e` 本身没有任何变化。这证明了 `Error.call(e)` 这种写法，无法继承原生构造函数。

ES6允许继承原生构造函数定义子类，因为ES6是先新建父类的实例对象 `this`，然后再用子类的构造函数修饰 `this`

, 使得父类的所有行为都可以继承。下面是一个继承 `Array` 的例子。

```
class MyArray extends Array {  
    constructor(...args) {  
        super(...args);  
    }  
}  
  
var arr = new MyArray();  
arr[0] = 12;  
arr.length // 1  
  
arr.length = 0;  
arr[0] // undefined
```

上面代码定义了一个 `MyArray` 类, 继承了 `Array` 构造函数, 因此就可以从 `MyArray` 生成数组的实例。这意味着, ES6 可以自定义原生数据结构 (比如 `Array`、`String` 等) 的子类, 这是 ES5 无法做到的。

上面这个例子也说明, `extends` 关键字不仅可以用来继承类, 还可以用来继承原生的构造函数。因此可以在原生数据结构的基础上, 定义自己的数据结构。下面就是定义了一个带版本功能的数组。

```
class VersionedArray extends Array {  
    constructor() {  
        super();  
        this.history = [[]];  
    }  
    commit() {  
        this.history.push(this.slice());  
    }  
    revert() {  
        this.splice(0, this.length, ...this.history[this.history.length - 1]);  
    }  
}  
  
var x = new VersionedArray();  
  
x.push(1);  
x.push(2);  
x // [1, 2]  
x.history // [[]]  
  
x.commit();  
x.history // [[], [1, 2]]  
x.push(3);  
x // [1, 2, 3]  
  
x.revert();  
x // [1, 2]
```

上面代码中, `VersionedArray` 结构会通过 `commit` 方法, 将自己的当前状态存入 `history` 属性, 然后通过 `revert` 方法, 可以撤销当前版本, 回到上一个版本。除此之外, `VersionedArray` 依然是一个数组, 所有原生的数组方法都可以在它上面调用。

下面是一个自定义 `Error` 子类的例子。

```
class ExtendableError extends Error {  
    constructor(message) {  
        super();  
        this.message = message;  
        this.stack = (new Error()).stack;  
        this.name = this.constructor.name;  
    }  
}
```

```
class MyError extends ExtendableError {
  constructor(m) {
    super(m);
  }
}

var myerror = new MyError('ll');
myerror.message // "ll"
myerror instanceof Error // true
myerror.name // "MyError"
myerror.stack
// Error
//     at MyError.ExtendableError
//     ...

```

注意，继承 `Object` 的子类，有一个行为差异。

```
class NewObj extends Object{
  constructor(){
    super(...arguments);
  }
}

var o = new NewObj({attr: true});
console.log(o.attr === true); // false
```

上面代码中，`NewObj` 继承了 `Object`，但是无法通过 `super` 方法向父类 `Object` 传参。这是因为ES6改变了 `Object` 构造函数的行为，一旦发现 `Object` 方法不是通过 `new Object()` 这种形式调用，ES6规定 `Object` 构造函数会忽略参数。

Class的取值函数（getter）和存值函数（setter）

与ES5一样，在Class内部可以使用 `get` 和 `set` 关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。

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

let inst = new MyClass();

inst.prop = 123;
// setter: 123

inst.prop
// 'getter'
```

上面代码中，`prop` 属性有对应的存值函数和取值函数，因此赋值和读取行为都被自定义了。

存值函数和取值函数是设置在属性的descriptor对象上的。

```
class CustomHTMLElement {
  constructor(element) {
```

```

    this.element = element;
}

get html() {
    return this.element.innerHTML;
}

set html(value) {
    this.element.innerHTML = value;
}
}

var descriptor = Object.getOwnPropertyDescriptor(
    CustomHTMLElement.prototype, "html");
"get" in descriptor // true
"set" in descriptor // true

```

上面代码中，存值函数和取值函数是定义在 `html` 属性的描述对象上面，这与ES5完全一致。

Class的Generator方法

如果某个方法之前加上星号（`*`），就表示该方法是一个Generator函数。

```

class Foo {
    constructor(...args) {
        this.args = args;
    }
    * [Symbol.iterator]() {
        for (let arg of this.args) {
            yield arg;
        }
    }
}

for (let x of new Foo('hello', 'world')) {
    console.log(x);
}
// hello
// world

```

上面代码中，`Foo`类的`Symbol.iterator`方法前有一个星号，表示该方法是一个Generator函数。`Symbol.iterator`方法返回一个`Foo`类的默认遍历器，`for...of`循环会自动调用这个遍历器。

Class的静态方法

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

```

class Foo {
    static classMethod() {
        return 'hello';
    }
}

Foo.classMethod() // 'hello'

var foo = new Foo();
foo.classMethod()
// TypeError: foo.classMethod is not a function

```

上面代码中，`Foo`类的 `classMethod` 方法前有 `static` 关键字，表明该方法是一个静态方法，可以直接在 `Foo` 类上

调用 (`Foo.classMethod()`)，而不是在 `Foo` 类的实例上调用。如果在实例上调用静态方法，会抛出一个错误，表示不存在该方法。

父类的静态方法，可以被子类继承。

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

上面代码中，父类 `Foo` 有一个静态方法，子类 `Bar` 可以调用这个方法。

静态方法也是可以从 `super` 对象上调用的。

```
class Foo {  
    static classMethod() {  
        return 'hello';  
    }  
}  
  
class Bar extends Foo {  
    static classMethod() {  
        return super.classMethod() + ', too';  
    }  
}  
  
Bar.classMethod();
```

Class的静态属性和实例属性

静态属性指的是Class本身的属性，即 `Class.propname`，而不是定义在实例对象（`this`）上的属性。

```
class Foo {  
}  
  
Foo.prop = 1;  
Foo.prop // 1
```

上面的写法为 `Foo` 类定义了一个静态属性 `prop`。

目前，只有这种写法可行，因为ES6明确规定，Class内部只有静态方法，没有静态属性。

```
// 以下两种写法都无效  
class Foo {  
    // 写法一  
    prop: 2  
  
    // 写法二  
    static prop: 2  
}  
  
Foo.prop // undefined
```

ES7有一个静态属性的[提案](#)，目前Babel转码器支持。

这个提案对实例属性和静态属性，都规定了新的写法。

(1) 类的实例属性

类的实例属性可以用等式，写入类的定义之中。

```
class MyClass {  
  myProp = 42;  
  
  constructor() {  
    console.log(this.myProp); // 42  
  }  
}
```

上面代码中，`myProp` 就是 `MyClass` 的实例属性。在 `MyClass` 的实例上，可以读取这个属性。

以前，我们定义实例属性，只能写在类的 `constructor` 方法里面。

```
class ReactCounter extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      count: 0  
    };  
  }  
}
```

上面代码中，构造方法 `constructor` 里面，定义了 `this.state` 属性。

有了新的写法以后，可以不在 `constructor` 方法里面定义。

```
class ReactCounter extends React.Component {  
  state = {  
    count: 0  
  };  
}
```

这种写法比以前更清晰。

为了可读性的目的，对于那些在 `constructor` 里面已经定义的实例属性，新写法允许直接列出。

```
class ReactCounter extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      count: 0  
    };  
  }  
  state;  
}
```

(2) 类的静态属性

类的静态属性只要在上面的实例属性写法前面，加上 `static` 关键字就可以了。

```
class MyClass {  
  static myStaticProp = 42;  
  
  constructor() {  
    console.log(MyClass.myProp); // 42  
  }  
}
```

同样的，这个新写法大大方便了静态属性的表达。

```
// 老写法
class Foo {
}
Foo.prop = 1;

// 新写法
class Foo {
    static prop = 1;
}
```

上面代码中，老写法的静态属性定义在类的外部。整个类生成以后，再生成静态属性。这样让人很容易忽略这个静态属性，也不符合相关代码应该放在一起的代码组织原则。另外，新写法是显式声明（declarative），而不是赋值处理，语义更好。

new.target属性

`new` 是从构造函数生成实例的命令。ES6 为 `new` 命令引入了一个 `new.target` 属性，（在构造函数中）返回 `new` 命令作用于的那个构造函数。如果构造函数不是通过 `new` 命令调用的，`new.target` 会返回 `undefined`，因此这个属性可以用来确定构造函数是怎么调用的。

```
function Person(name) {
    if (new.target !== undefined) {
        this.name = name;
    } else {
        throw new Error('必须使用new生成实例');
    }
}

// 另一种写法
function Person(name) {
    if (new.target === Person) {
        this.name = name;
    } else {
        throw new Error('必须使用new生成实例');
    }
}

var person = new Person('张三'); // 正确
var notAPerson = Person.call(person, '张三'); // 报错
```

上面代码确保构造函数只能通过 `new` 命令调用。

Class 内部调用 `new.target`，返回当前 Class。

```
class Rectangle {
    constructor(length, width) {
        console.log(new.target === Rectangle);
        this.length = length;
        this.width = width;
    }
}

var obj = new Rectangle(3, 4); // 输出 true
```

需要注意的是，子类继承父类时，`new.target` 会返回子类。

```
class Rectangle {
    constructor(length, width) {
```

```

        console.log(new.target === Rectangle);
        // ...
    }

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}

var obj = new Square(3); // 输出 false

```

上面代码中，`new.target` 会返回子类。

利用这个特点，可以写出不能独立使用、必须继承后才能使用的类。

```

class Shape {
    constructor() {
        if (new.target === Shape) {
            throw new Error('本类不能实例化');
        }
    }
}

class Rectangle extends Shape {
    constructor(length, width) {
        super();
        // ...
    }
}

var x = new Shape(); // 报错
var y = new Rectangle(3, 4); // 正确

```

上面代码中，`Shape` 类不能被实例化，只能用于继承。

注意，在函数外部，使用 `new.target` 会报错。

Mixin模式的实现

Mixin模式指的是，将多个类的接口“混入”（mix in）另一个类。它在ES6的实现如下。

```

function mix(...mixins) {
    class Mix {}

    for (let mixin of mixins) {
        copyProperties(Mix, mixin);
        copyProperties(Mix.prototype, mixin.prototype);
    }

    return Mix;
}

function copyProperties(target, source) {
    for (let key of Reflect.ownKeys(source)) {
        if (key !== "constructor"
            && key !== "prototype"
            && key !== "name")
        ) {
            let desc = Object.getOwnPropertyDescriptor(source, key);
            Object.defineProperty(target, key, desc);
        }
    }
}

```

```
}
```

上面代码的 `mix` 函数，可以将多个对象合成为一个类。使用的时候，只要继承这个类即可。

```
class DistributedEdit extends mix(Loggable, Serializable) {  
    // ...  
}
```

修饰器

类的修饰

修饰器（Decorator）是一个函数，用来修改类的行为。这是ES7的一个[提案](#)，目前Babel转码器已经支持。

修饰器对类的行为的改变，是代码编译时发生的，而不是在运行时。这意味着，修饰器能在编译阶段运行代码。

```
function testable(target) {  
    target.isTestable = true;  
}  
  
@testable  
class MyTestableClass {}  
  
console.log(MyTestableClass.isTestable) // true
```

上面代码中，`@testable` 就是一个修饰器。它修改了 `MyTestableClass` 这个类的行为，为它加上了静态属性 `isTestable`。

基本上，修饰器的行为就是下面这样。

```
@decorator  
class A {}  
  
// 等同于  
  
class A {}  
A = decorator(A) || A;
```

也就是说，修饰器本质就是编译时执行的函数。

修饰器函数的第一个参数，就是所要修饰的目标类。

```
function testable(target) {  
    // ...  
}
```

上面代码中，`testable` 函数的参数 `target`，就是会被修饰的类。

如果觉得一个参数不够用，可以在修饰器外面再封装一层函数。

```
function testable(isTestable) {  
    return function(target) {  
        target.isTestable = isTestable;  
    }  
}  
  
@testable(true)  
class MyTestableClass {}
```

```
MyTestableClass.isTestable // true  
  
@testable(false)  
class MyClass {}  
MyClass.isTestable // false
```

上面代码中，修饰器 `testable` 可以接受参数，这就等于可以修改修饰器的行为。

前面的例子是为类添加一个静态属性，如果想添加实例属性，可以通过目标类的 `prototype` 对象操作。

```
function testable(target) {  
  target.prototype.isTestable = true;  
}  
  
@testable  
class MyTestableClass {}  
  
let obj = new MyTestableClass();  
obj.isTestable // true
```

上面代码中，修饰器函数 `testable` 是在目标类的 `prototype` 对象上添加属性，因此就可以在实例上调用。

下面是另外一个例子。

```
// mixins.js  
export function mixins(...list) {  
  return function (target) {  
    Object.assign(target.prototype, ...list)  
  }  
}  
  
// main.js  
import { mixins } from './mixins'  
  
const Foo = {  
  foo() { console.log('foo') }  
};  
  
@mixins(Foo)  
class MyClass {}  
  
let obj = new MyClass();  
obj.foo() // 'foo'
```

上面代码通过修饰器 `mixins`，把 `Foo` 类的方法添加到了 `MyClass` 的实例上面。可以用 `Object.assign()` 模拟这个功能。

```
const Foo = {  
  foo() { console.log('foo') }  
};  
  
class MyClass {}  
  
Object.assign(MyClass.prototype, Foo);  
  
let obj = new MyClass();  
obj.foo() // 'foo'
```

方法的修饰

修饰器不仅可以修饰类，还可以修饰类的属性。

```
class Person {
  @readonly
  name() { return `${this.first} ${this.last}` }
}
```

上面代码中，修饰器 `readonly` 用来修饰“类”的 `name` 方法。

此时，修饰器函数一共可以接受三个参数，第一个参数是所要修饰的目标对象，第二个参数是所要修饰的属性名，第三个参数是该属性的描述对象。

```
function readonly(target, name, descriptor) {
  // descriptor对象原来的值如下
  // {
  //   value: specifiedFunction,
  //   enumerable: false,
  //   configurable: true,
  //   writable: true
  // };
  descriptor.writable = false;
  return descriptor;
}

readonly(Person.prototype, 'name', descriptor);
// 类似于
Object.defineProperty(Person.prototype, 'name', descriptor);
```

上面代码说明，修饰器 `(readonly)` 会修改属性的描述对象 `(descriptor)`，然后被修改的描述对象再用来定义属性。

下面是另一个例子，修改属性描述对象的 `enumerable` 属性，使得该属性不可遍历。

```
class Person {
  @nonenumerable
  get kidCount() { return this.children.length; }
}

function nonenumerable(target, name, descriptor) {
  descriptor.enumerable = false;
  return descriptor;
}
```

下面的 `@log` 修饰器，可以起到输出日志的作用。

```
class Math {
  @log
  add(a, b) {
    return a + b;
  }
}

function log(target, name, descriptor) {
  var oldValue = descriptor.value;

  descriptor.value = function() {
    console.log(`Calling "${name}" with`, arguments);
    return oldValue.apply(null, arguments);
  };
}

return descriptor;
}

const math = new Math();
```

```
// passed parameters should get logged now  
math.add(2, 4);
```

上面代码中，`@log` 修饰器的作用就是在执行原始的操作之前，执行一次 `console.log`，从而达到输出日志的目的。修饰器有注释的作用。

```
@testable  
class Person {  
  @readonly  
  @nonenumerable  
  name() { return `${this.first} ${this.last}` }  
}
```

从上面代码中，我们一眼就能看出，`Person` 类是可测试的，而 `name` 方法是只读和不可枚举的。

如果同一个方法有多个修饰器，会像剥洋葱一样，先从外到内进入，然后由内向外执行。

```
function dec(id){  
  console.log('evaluated', id);  
  return (target, property, descriptor) => console.log('executed', id);  
}  
  
class Example {  
  @dec(1)  
  @dec(2)  
  method(){}
}  
// evaluated 1  
// evaluated 2  
// executed 2  
// executed 1
```

上面代码中，外层修饰器 `@dec(1)` 先进入，但是内层修饰器 `@dec(2)` 先执行。

除了注释，修饰器还能用来类型检查。所以，对于类来说，这项功能相当有用。从长期来看，它将是JavaScript代码静态分析的重要工具。

为什么修饰器不能用于函数？

修饰器只能用于类和类的方法，不能用于函数，因为存在函数提升。

```
var counter = 0;  
  
var add = function () {  
  counter++;  
};  
  
@add  
function foo() {  
}
```

上面的代码，意图是执行后 `counter` 等于1，但是实际上结果是 `counter` 等于0。因为函数提升，使得实际执行的代码是下面这样。

```
@add  
function foo() {  
}  
  
var counter;  
var add;
```

```
counter = 0;

add = function () {
  counter++;
};
```

下面是另一个例子。

```
var readOnly = require("some-decorator");

@readOnly
function foo() {  
}
```

上面代码也有问题，因为实际执行是下面这样。

```
var readOnly;

@readOnly
function foo() {  
}

readOnly = require("some-decorator");
```

总之，由于存在函数提升，使得修饰器不能用于函数。类是不会提升的，所以就没有这方面的问题。

core-decorators.js

[core-decorators.js](#)是一个第三方模块，提供了几个常见的修饰器，通过它可以更好地理解修饰器。

(1) @autobind

`autobind` 修饰器使得方法中的 `this` 对象，绑定原始对象。

```
import { autobind } from 'core-decorators';

class Person {
  @autobind
  getPerson() {
    return this;
  }
}

let person = new Person();
let getPerson = person.getPerson;

getPerson() === person;
// true
```

(2) @readonly

`readonly` 修饰器使得属性或方法不可写。

```
import { readonly } from 'core-decorators';

class Meal {
  @readonly
  entree = 'steak';
}
```

```
var dinner = new Meal();
dinner.entree = 'salmon';
// Cannot assign to read only property 'entree' of [object Object]
```

(3) @override

`override` 修饰器检查子类的方法，是否正确覆盖了父类的同名方法，如果不正确会报错。

```
import { override } from 'core-decorators';

class Parent {
  speak(first, second) {}
}

class Child extends Parent {
  @override
  speak() {}
  // SyntaxError: Child#speak() does not properly override Parent#speak(first, second)
}

// or

class Child extends Parent {
  @override
  speaks() {}
  // SyntaxError: No descriptor matching Child#speaks() was found on the prototype chain.
  //
  // Did you mean "speak"?
}
```

(4) @deprecate (别名@deprecated)

`deprecate` 或 `deprecated` 修饰器在控制台显示一条警告，表示该方法将废除。

```
import { deprecate } from 'core-decorators';

class Person {
  @deprecate
  facepalm() {}

  @deprecate('We stopped facepalming')
  facepalmHard() {}

  @deprecate('We stopped facepalming', { url:
    'http://knowyourmeme.com/memes/facepalm' })
  facepalmHarder() {}

let person = new Person();

person.facepalm();
// DEPRECATION Person#facepalm: This function will be removed in future versions.

person.facepalmHard();
// DEPRECATION Person#facepalmHard: We stopped facepalming

person.facepalmHarder();
// DEPRECATION Person#facepalmHarder: We stopped facepalming
//
// See http://knowyourmeme.com/memes/facepalm for more details.
//
```

(5) @suppressWarnings

`suppressWarnings` 修饰器抑制 `decorated` 修饰器导致的 `console.warn()` 调用。但是，异步代码发出的调用除外。

```
import { suppressWarnings } from 'core-decorators';

class Person {
  @deprecated
  facepalm() {}

  @suppressWarnings
  facepalmWithoutWarning() {
    this.facepalm();
  }
}

let person = new Person();

person.facepalmWithoutWarning();
// no warning is logged
```

使用修饰器实现自动发布事件

我们可以使用修饰器，使得对象的方法被调用时，自动发出一个事件。

```
import postal from "postal/lib/postal.lodash";

export default function publish(topic, channel) {
  return function(target, name, descriptor) {
    const fn = descriptor.value;

    descriptor.value = function() {
      let value = fn.apply(this, arguments);
      postal.channel(channel || target.channel || "/").publish(topic, value);
    };
  };
}
```

上面代码定义了一个名为 `publish` 的修饰器，它通过改写 `descriptor.value`，使得原方法被调用时，会自动发出一个事件。它使用的事件“发布/订阅”库是[Postal.js](#)。

它的用法如下。

```
import publish from "path/to/decorators/publish";

class FooComponent {
  @publish("foo.some.message", "component")
  someMethod() {
    return {
      my: "data"
    };
  }
  @publish("foo.some.other")
  anotherMethod() {
    // ...
  }
}
```

以后，只要调用 `someMethod` 或者 `anotherMethod`，就会自动发出一个事件。

```
let foo = new FooComponent();
```

```
foo.someMethod() // 在"component"频道发布"foo.some.message"事件，附带的数据是{ my: "data"
}
foo.anotherMethod() // 在"/"频道发布"foo.some.other"事件，不附带数据
```

Mixin

在修饰器的基础上，可以实现 **Mixin** 模式。所谓 **Mixin** 模式，就是对象继承的一种替代方案，中文译为“混入”（mix in），意为在一个对象之中混入另外一个对象的方法。

请看下面的例子。

```
const Foo = {
  foo() { console.log('foo') }
};

class MyClass {}

Object.assign(MyClass.prototype, Foo);

let obj = new MyClass();
obj.foo() // 'foo'
```

上面代码之中，对象 **Foo** 有一个 **foo** 方法，通过 **Object.assign** 方法，可以将 **foo** 方法“混入”**MyClass** 类，导致 **MyClass** 的实例 **obj** 对象都具有 **foo** 方法。这就是“混入”模式的一个简单实现。

下面，我们部署一个通用脚本 **mixins.js**，将**mixin**写成一个修饰器。

```
export function mixins(...list) {
  return function (target) {
    Object.assign(target.prototype, ...list);
  };
}
```

然后，就可以使用上面这个修饰器，为类“混入”各种方法。

```
import { mixins } from './mixins';

const Foo = {
  foo() { console.log('foo') }
};

@mixins(Foo)
class MyClass {}

let obj = new MyClass();
obj.foo() // "foo"
```

通过**mixins**这个修饰器，实现了在**MyClass**类上面“混入”**Foo**对象的 **foo** 方法。

不过，上面的方法会改写 **MyClass** 类的 **prototype** 对象，如果不喜欢这一点，也可以通过类的继承实现**mixin**。

```
class MyClass extends MyBaseClass {
  /* ... */
}
```

上面代码中，**MyClass** 继承了 **MyBaseClass**。如果我们想在 **MyClass** 里面“混入”一个 **foo** 方法，一个办法是在 **MyClass** 和 **MyBaseClass** 之间插入一个混入类，这个类具有 **foo** 方法，并且继承了 **MyBaseClass** 的所有方法，然后 **MyClass** 再继承这个类。

```
let MyMixin = (superclass) => class extends superclass {
  foo() {
    console.log('foo from MyMixin');
  }
};
```

上面代码中，`MyMixin` 是一个混入类生成器，接受 `superclass` 作为参数，然后返回一个继承 `superclass` 的子类，该子类包含一个 `foo` 方法。

接着，目标类再去继承这个混入类，就达到了“混入”`foo` 方法的目的。

```
class MyClass extends MyMixin(MyBaseClass) {
  /* ... */
}

let c = new MyClass();
c.foo(); // "foo from MyMixin"
```

如果需要“混入”多个方法，就生成多个混入类。

```
class MyClass extends Mixin1(Mixin2(MyBaseClass)) {
  /* ... */
}
```

这种写法的一个好处，是可以调用 `super`，因此可以避免在“混入”过程中覆盖父类的同名方法。

```
let Mixin1 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin1');
    if (super.foo) super.foo();
  }
};

let Mixin2 = (superclass) => class extends superclass {
  foo() {
    console.log('foo from Mixin2');
    if (super.foo) super.foo();
  }
};

class S {
  foo() {
    console.log('foo from S');
  }
}

class C extends Mixin1(Mixin2(S)) {
  foo() {
    console.log('foo from C');
    super.foo();
  }
}
```

上面代码中，每一次 `混入` 发生时，都调用了父类的 `super.foo` 方法，导致父类的同名方法没有被覆盖，行为被保留了下来。

```
new C().foo()
// foo from C
// foo from Mixin1
// foo from Mixin2
// foo from S
```

Trait

Trait也是一种修饰器，效果与Mixin类似，但是提供更多功能，比如防止同名方法的冲突、排除混入某些方法、为混入的方法起别名等等。

下面采用[traits-decorator](#)这个第三方模块作为例子。这个模块提供的traits修饰器，不仅可以接受对象，还可以接受ES6类作为参数。

```
import { traits } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') }
};

@traits(TFoo, TBar)
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.bar() // bar
```

上面代码中，通过traits修饰器，在 `MyClass` 类上面“混入”了 `TFoo` 类的 `foo` 方法和 `TBar` 对象的 `bar` 方法。

Trait不允许“混入”同名方法。

```
import { traits } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};

@traits(TFoo, TBar)
class MyClass { }
// 报错
// throw new Error('Method named: ' + methodName + ' is defined twice.');
//          ^
// Error: Method named: foo is defined twice.
```

上面代码中，`TFoo`和`TBar`都有`foo`方法，结果traits修饰器报错。

一种解决方法是排除`TBar`的`foo`方法。

```
import { traits, excludes } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};
```

```
@traits(TFoo, TBar::excludes('foo'))
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.bar() // bar
```

上面代码使用绑定运算符 (::) 在TBar上排除foo方法，混入时就不会报错了。

另一种方法是为TBar的foo方法起一个别名。

```
import { traits, alias } from 'traits-decorator';

class TFoo {
  foo() { console.log('foo') }
}

const TBar = {
  bar() { console.log('bar') },
  foo() { console.log('foo') }
};

@traits(TFoo, TBar::alias({foo: 'aliasFoo'}))
class MyClass { }

let obj = new MyClass();
obj.foo() // foo
obj.aliasFoo() // foo
obj.bar() // bar
```

上面代码为TBar的foo方法起了别名aliasFoo，于是MyClass也可以混入TBar的foo方法了。

alias和excludes方法，可以结合起来使用。

```
@traits(TExample::excludes('foo', 'bar')::alias({baz: 'exampleBaz'}))
class MyClass { }
```

上面代码排除了TExample的foo方法和bar方法，为baz方法起了别名exampleBaz。

as方法则为上面的代码提供了另一种写法。

```
@traits(TExample::as({excludes: ['foo', 'bar'], alias: {baz: 'exampleBaz'}}))
class MyClass { }
```

Babel转码器的支持

目前，Babel转码器已经支持Decorator。

首先，安装 `babel-core` 和 `babel-plugin-transform-decorators`。由于后者包括在 `babel-preset-stage-0` 之中，所以改为安装 `babel-preset-stage-0` 亦可。

```
$ npm install babel-core babel-plugin-transform-decorators
```

然后，设置配置文件 `.babelrc`。

```
{
  "plugins": ["transform-decorators"]
}
```

这时，Babel就可以对Decorator转码了。

脚本中打开的命令如下。

```
babel.transform("code", {plugins: ["transform-decorators"]})
```

Babel的官方网站提供一个[在线转码器](#)，只要勾选Experimental，就能支持Decorator的在线转码。

Module

历史上，JavaScript 一直没有模块（module）体系，无法将一个大程序拆分成互相依赖的小文件，再用简单的方法拼装起来。其他语言都有这项功能，比如 Ruby 的 `require`、Python 的 `import`，甚至就连 CSS 都有 `@import`，但是 JavaScript 任何这方面的支持都没有，这对开发大型的、复杂的项目形成了巨大障碍。

在 ES6 之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种。前者用于服务器，后者用于浏览器。ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代现有的 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

ES6 模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS 和 AMD 模块，都只能在运行时确定这些东西。比如，CommonJS 模块就是对象，输入时必须查找对象属性。

```
// CommonJS模块
let { stat, exists, readFile } = require('fs');

// 等同于
let _fs = require('fs');
let stat = _fs.stat, exists = _fs.exists, readfile = _fs.readFile;
```

上面代码的实质是整体加载 `fs` 模块（即加载 `fs` 的所有方法），生成一个对象（`_fs`），然后再从这个对象上面读取3个方法。这种加载称为“运行时加载”，因为只有运行时才能得到这个对象，导致完全没办法在编译时做“静态优化”。

ES6 模块不是对象，而是通过 `export` 命令显式指定输出的代码，再通过 `import` 命令输入。

```
// ES6模块
import { stat, exists, readFile } from 'fs';
```

上面代码的实质是从 `fs` 模块加载3个方法，其他方法不加载。这种加载称为“编译时加载”或者静态加载，即 ES6 可以在编译时就完成模块加载，效率要比 CommonJS 模块的加载方式高。当然，这也导致了没法引用 ES6 模块本身，因为它不是对象。

由于 ES6 模块是编译时加载，使得静态分析成为可能。有了它，就能进一步拓宽 JavaScript 的语法，比如引入宏（macro）和类型检验（type system）这些只能靠静态分析实现的功能。

除了静态加载带来的各种好处，ES6 模块还有以下好处。

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

严格模式

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

严格模式主要有以下限制。

- 变量必须声明后再使用
- 函数的参数不能有同名属性，否则报错
- 不能使用 `with` 语句

- 不能对只读属性赋值，否则报错
- 不能使用前缀0表示八进制数，否则报错
- 不能删除不可删除的属性，否则报错
- 不能删除变量 `delete prop`，会报错，只能删除属性 `delete global[prop]`
- `eval` 不会在它的外层作用域引入变量
- `eval` 和 `arguments` 不能被重新赋值
- `arguments` 不会自动反映函数参数的变化
- 不能使用 `argumentscallee`
- 不能使用 `argumentscaller`
- 禁止 `this` 指向全局对象
- 不能使用 `fn.caller` 和 `fn.arguments` 获取函数调用的堆栈
- 增加了保留字（比如 `protected`、`static` 和 `interface`）

上面这些限制，模块都必须遵守。由于严格模式是 ES5 引入的，不属于 ES6，所以请参阅相关 ES5 书籍，本书不再详细介绍。

export 命令

模块功能主要由两个命令构成：`export` 和 `import`。`export` 命令用于规定模块的对外接口，`import` 命令用于输入其他模块提供的功能。

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用 `export` 关键字输出该变量。下面是一个 JS 文件，里面使用 `export` 命令输出变量。

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

上面代码是 `profile.js` 文件，保存了用户信息。ES6 将其视为一个模块，里面用 `export` 命令对外部输出了三个变量。

`export` 的写法，除了像上面这样，还有另外一种。

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

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

上面代码在 `export` 命令后面，使用大括号指定所要输出的一组变量。它与前一种写法（直接放置在 `var` 语句前）是等价的，但是应该优先考虑使用这种写法。因为这样就可以在脚本尾部，一眼看清楚输出了哪些变量。

`export` 命令除了输出变量，还可以输出函数或类（class）。

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

上面代码对外输出一个函数 `multiply`。

通常情况下，`export` 输出的变量就是本来的名字，但是可以使用 `as` 关键字重命名。

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

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

上面代码使用 `as` 关键字，重命名了函数 `v1` 和 `v2` 的对外接口。重命名后，`v2` 可以用不同的名字输出两次。

需要特别注意的是，`export` 命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

```
// 报错
export 1;

// 报错
var m = 1;
export m;
```

上面两种写法都会报错，因为没有提供对外的接口。第一种写法直接输出 `1`，第二种写法通过变量 `m`，还是直接输出 `1`。`1` 只是一个值，不是接口。正确的写法是下面这样。

```
// 写法一
export var m = 1;

// 写法二
var m = 1;
export {m};

// 写法三
var n = 1;
export {n as m};
```

上面三种写法都是正确的，规定了对外的接口 `m`。其他脚本可以通过这个接口，取到值 `1`。它们的实质是，在接口名与模块内部变量之间，建立了一一对应的关系。

同样的，`function` 和 `class` 的输出，也必须遵守这样的写法。

```
// 报错
function f() {}
export f;

// 正确
export function f() {};

// 正确
function f() {}
export {f};
```

另外，`export` 语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

```
export var foo = 'bar';
setTimeout(() => foo = 'baz', 500);
```

上面代码输出变量 `foo`，值为 `bar`，500毫秒之后变成 `baz`。

这一点与CommonJS规范完全不同。CommonJS模块输出的是值的缓存，不存在动态更新，详见下文《ES6模块加载的实质》一节。

最后，`export` 命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的 `import` 命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了ES6模块的设计初衷。

```
function foo() {
```

```
export default 'bar' // SyntaxError
}
foo()
```

上面代码中，`export` 语句放在函数之中，结果报错。

import 命令

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

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

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

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

如果想为输入的变量重新取一个名字，`import` 命令要使用 `as` 关键字，将输入的变量重命名。

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

`import` 后面的 `from` 指定模块文件的位置，可以是相对路径，也可以是绝对路径，`.js` 路径可以省略。如果只是模块名，不带有路径，那么必须有配置文件，告诉 JavaScript 引擎该模块的位置。

```
import {myMethod} from 'util';
```

上面代码中，`util` 是模块文件名，由于不带有路径，必须通过配置，告诉引擎怎么取到这个模块。

注意，`import` 命令具有提升效果，会提升到整个模块的头部，首先执行。

```
foo();

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

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

由于 `import` 是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

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

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

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

上面三种写法都会报错，因为它们用到了表达式、变量和 `if` 结构。在静态分析阶段，这些语法都是没法得到值的。

最后，`import` 语句会执行所加载的模块，因此可以有下面的写法。

```
import 'lodash';
```

上面代码仅仅执行 `lodash` 模块，但是不输入任何值。

如果多次重复执行同一句 `import` 语句，那么只会执行一次，而不会执行多次。

```
import 'lodash';
import 'lodash';
```

上面代码加载了两次 `lodash`，但是只会执行一次。

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

// 等同于
import { foo, bar } from 'my_module';
```

上面代码中，虽然 `foo` 和 `bar` 在两个语句中加载，但是它们对应的是同一个 `my_module` 实例。也就是说，`import` 语句是 Singleton 模式。

模块的整体加载

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

下面是一个 `circle.js` 文件，它输出两个方法 `area` 和 `circumference`。

```
// circle.js

export function area(radius) {
  return Math.PI * radius * radius;
}

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

现在，加载这个模块。

```
// main.js

import { area, circumference } from './circle';

console.log('圆面积: ' + area(4));
console.log('圆周长: ' + circumference(14));
```

上面写法是逐一指定要加载的方法，整体加载的写法如下。

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

console.log('圆面积: ' + circle.area(4));
console.log('圆周长: ' + circle.circumference(14));
```

export default 命令

从前面的例子可以看出，使用 `import` 命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到 `export default` 命令，为模块指定默认输出。

```
// export-default.js
export default function () {
  console.log('foo');
}
```

上面代码是一个模块文件 `export-default.js`，它的默认输出是一个函数。

其他模块加载该模块时，`import` 命令可以为该匿名函数指定任意名字。

```
// import-default.js
import customName from './export-default';
customName(); // 'foo'
```

上面代码的 `import` 命令，可以用任意名称指向 `export-default.js` 输出的方法，这时就不需要知道原模块输出的函数名。需要注意的是，这时 `import` 命令后面，不使用大括号。

`export default` 命令用在非匿名函数前，也是可以的。

```
// export-default.js
export default function foo() {
  console.log('foo');
}

// 或者写成

function foo() {
  console.log('foo');
}

export default foo;
```

上面代码中，`foo` 函数的函数名 `foo`，在模块外部是无效的。加载的时候，视同匿名函数加载。

下面比较一下默认输出和正常输出。

```
// 第一组
export default function crc32() { // 输出
  // ...
}

import crc32 from 'crc32'; // 输入

// 第二组
export function crc32() { // 输出
  // ...
};

import {crc32} from 'crc32'; // 输入
```

上面代码的两组写法，第一组是使用 `export default` 时，对应的 `import` 语句不需要使用大括号；第二组是不使用 `export default` 时，对应的 `import` 语句需要使用大括号。

`export default` 命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此 `export default` 命令只能使用一次。所以，`import` 命令后面才不用加大括号，因为只可能对应一个方法。

本质上，`export default` 就是输出一个叫做 `default` 的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效的。

```
// modules.js
function add(x, y) {
  return x * y;
```

```
}

export {add as default};
// 等同于
// export default add;

// app.js
import { default as xxx } from 'modules';
// 等同于
// import xxx from 'modules';
```

正是因为 `export default` 命令其实只是输出一个叫做 `default` 的变量，所以它后面不能跟变量声明语句。

```
// 正确
export var a = 1;

// 正确
var a = 1;
export default a;

// 错误
export default var a = 1;
```

上面代码中，`export default a` 的含义是将变量 `a` 的值赋给变量 `default`。所以，最后一种写法会报错。

有了 `export default` 命令，输入模块时就非常直观了，以输入 `lodash` 模块为例。

```
import _ from 'lodash';
```

如果想在一条 `import` 语句中，同时输入默认方法和其他变量，可以写成下面这样。

```
import _, { each } from 'lodash';
```

对应上面代码的 `export` 语句如下。

```
export default function (obj) {
  // ...
}
export function each(obj, iterator, context) {
  // ...
}
export { each as forEach };
```

上面代码的最后一行的意思是，暴露出 `forEach` 接口，默认指向 `each` 接口，即 `forEach` 和 `each` 指向同一个方法。

如果要输出默认的值，只需将值跟在 `export default` 之后即可。

```
export default 42;
```

`export default` 也可以用来输出类。

```
// MyClass.js
export default class { ... }

// main.js
import MyClass from 'MyClass';
let o = new MyClass();
```

export 与 import 的复合写法

如果在一个模块之中，先输入后输出同一个模块，`import` 语句可以与 `export` 语句写在一起。

```
export { foo, bar } from 'my_module';
// 等同于
import { foo, bar } from 'my_module';
export { foo, bar };
```

上面代码中，`export` 和 `import` 语句可以结合在一起，写成一行。

模块的接口改名和整体输出，也可以采用这种写法。

```
// 接口改名
export { foo as myFoo } from 'my_module';

// 整体输出
export * from 'my_module';
```

默认接口的写法如下。

```
export { default } from 'foo';
```

具名接口改为默认接口的写法如下。

```
export { es6 as default } from './someModule';
// 等同于
import { es6 } from './someModule';
export default es6;
```

同样地，默认接口也可以改名为具名接口。

```
export { default as es6 } from './someModule';
```

另外，ES7有一个[提案](#)，简化先输入后输出的写法，拿掉输出时的大括号。

```
// 现行的写法
export {v} from 'mod';

// 提案的写法
export v from 'mod';
```

模块的继承

模块之间也可以继承。

假设有一个 `circleplus` 模块，继承了 `circle` 模块。

```
// circleplus.js

export * from 'circle';
export var e = 2.71828182846;
export default function(x) {
  return Math.exp(x);
}
```

上面代码中的 `export *`，表示再输出 `circle` 模块的所有属性和方法。注意，`export *` 命令会忽略 `circle` 模块的 `default` 方法。然后，上面代码又输出了自定义的 `e` 变量和默认方法。

这时，也可以将 `circle` 的属性或方法，改名后再输出。

```
// circleplus.js
export { area as circleArea } from 'circle';
```

上面代码表示，只输出 `circle` 模块的 `area` 方法，且将其改名为 `circleArea`。

加载上面模块的写法如下。

```
// main.js
import * as math from 'circleplus';
import exp from 'circleplus';
console.log(exp(math.e));
```

上面代码中的 `import exp` 表示，将 `circleplus` 模块的默认方法加载为 `exp` 方法。

ES6模块加载的实质

ES6模块加载的机制，与CommonJS模块完全不同。CommonJS模块输出的是一个值的拷贝，而ES6模块输出的是值的引用。

CommonJS模块输出的是被输出值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。请看下面这个模块文件 `lib.js` 的例子。

```
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
module.exports = {
  counter: counter,
  incCounter: incCounter,
};
```

上面代码输出内部变量 `counter` 和改写这个变量的内部方法 `incCounter`。然后，在 `main.js` 里面加载这个模块。

```
// main.js
var mod = require('./lib');

console.log(mod.counter); // 3
mod.incCounter();
console.log(mod.counter); // 3
```

上面代码说明，`lib.js` 模块加载以后，它的内部变化就影响不到输出的 `mod.counter` 了。这是因为 `mod.counter` 是一个原始类型的值，会被缓存。除非写成一个函数，才能得到内部变动后的值。

```
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
module.exports = {
  get counter() {
    return counter
  },
  incCounter: incCounter,
};
```

上面代码中，输出的 `counter` 属性实际上是一个取值器函数。现在再执行 `main.js`，就可以正确读取内部变量 `counter` 的变动了。

```
$ node main.js
3
4
```

ES6模块的运行机制与CommonJS不一样，它遇到模块加载命令 `import` 时，不会去执行模块，而是只生成一个动态的只读引用。等到真的需要用到时，再到模块里面去取值，换句话说，ES6的输入有点像Unix系统的“符号连接”，原始值变了，`import` 输入的值也会跟着变。因此，ES6模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

还是举上面的例子。

```
// lib.js
export let counter = 3;
export function incCounter() {
  counter++;
}

// main.js
import { counter, incCounter } from './lib';
console.log(counter); // 3
incCounter();
console.log(counter); // 4
```

上面代码说明，ES6模块输入的变量 `counter` 是活的，完全反应其所在模块 `lib.js` 内部的变化。

再举一个出现在 `export` 一节中的例子。

```
// m1.js
export var foo = 'bar';
setTimeout(() => foo = 'baz', 500);

// m2.js
import {foo} from './m1.js';
console.log(foo);
setTimeout(() => console.log(foo), 500);
```

上面代码中，`m1.js` 的变量 `foo`，在刚加载时等于 `bar`，过了500毫秒，又变为等于 `baz`。

让我们看看，`m2.js` 能否正确读取这个变化。

```
$ babel-node m2.js
bar
baz
```

上面代码表明，ES6模块不会缓存运行结果，而是动态地去被加载的模块取值，并且变量总是绑定其所在的模块。

由于ES6输入的模块变量，只是一个“符号连接”，所以这个变量是只读的，对它进行重新赋值会报错。

```
// lib.js
export let obj = {};

// main.js
import { obj } from './lib';

obj.prop = 123; // OK
obj = {}; // TypeError
```

上面代码中，`main.js` 从 `lib.js` 输入变量 `obj`，可以对 `obj` 添加属性，但是重新赋值就会报错。因为变量 `obj` 指向的地址是只读的，不能重新赋值，这就好比 `main.js` 创造了一个名为 `obj` 的 `const` 变量。

最后，`export` 通过接口，输出的是同一个值。不同的脚本加载这个接口，得到的都是同样的实例。

```
// mod.js
function C() {
  this.sum = 0;
  this.add = function () {
    this.sum += 1;
  };
  this.show = function () {
    console.log(this.sum);
  };
}

export let c = new C();
```

上面的脚本 `mod.js`，输出的是一个 `C` 的实例。不同的脚本加载这个模块，得到的都是同一个实例。

```
// x.js
import {c} from './mod';
c.add();

// y.js
import {c} from './mod';
c.show();

// main.js
import './x';
import './y';
```

现在执行 `main.js`，输出的是1。

```
$ babel-node main.js
1
```

这就证明了 `x.js` 和 `y.js` 加载的都是 `C` 的同一个实例。

浏览器的模块加载

浏览器使用 ES6 模块的语法如下。

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

上面代码在网页中插入一个模块 `foo.js`，由于 `type` 属性设为 `module`，所以浏览器知道这是一个 ES6 模块。

浏览器对于带有 `type="module"` 的 `<script>`，都是异步加载外部脚本，不会造成堵塞浏览器。

对于外部的模块脚本（上例是 `foo.js`），有几点需要注意。

- 该脚本自动采用严格模块。
- 该脚本内部的顶层变量，都只在该脚本内部有效，外部不可见。
- 该脚本内部的顶层的 `this` 关键字，返回 `undefined`，而不是指向 `window`。

循环加载

“循环加载”（circular dependency）指的是，`a` 脚本的执行依赖 `b` 脚本，而 `b` 脚本的执行又依赖 `a` 脚本。

```
// a.js
var b = require('b');

// b.js
var a = require('a');
```

通常，“循环加载”表示存在强耦合，如果处理不好，还可能导致递归加载，使得程序无法执行，因此应该避免出现。

但是实际上，这是很难避免的，尤其是依赖关系复杂的大项目，很容易出现 `a` 依赖 `b`，`b` 依赖 `c`，`c` 又依赖 `a` 这样的情况。这意味着，模块加载机制必须考虑“循环加载”的情况。

对于JavaScript语言来说，目前最常见的两种模块格式CommonJS和ES6，处理“循环加载”的方法是不一样的，返回的结果也不一样。

CommonJS模块的加载原理

介绍ES6如何处理“循环加载”之前，先介绍目前最流行的CommonJS模块格式的加载原理。

CommonJS的一个模块，就是一个脚本文件。`require` 命令第一次加载该脚本，就会执行整个脚本，然后在内存生成一个对象。

```
{
  id: '...',
  exports: { ... },
  loaded: true,
  ...
}
```

上面代码就是Node内部加载模块后生成的一个对象。该对象的 `id` 属性是模块名，`exports` 属性是模块输出的各个接口，`loaded` 属性是一个布尔值，表示该模块的脚本是否执行完毕。其他还有很多属性，这里都省略了。

以后需要用到这个模块的时候，就会到 `exports` 属性上面取值。即使再次执行 `require` 命令，也不会再次执行该模块，而是到缓存之中取值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。

CommonJS模块的循环加载

CommonJS模块的重要特性是加载时执行，即脚本代码在 `require` 的时候，就会全部执行。一旦出现某个模块被“循环加载”，就只输出已经执行的部分，还未执行的部分不会输出。

让我们来看，Node[官方文档](#)里面的例子。脚本文件 `a.js` 代码如下。

```
exports.done = false;
var b = require('./b.js');
console.log('在 a.js 之中, b.done = %j', b.done);
exports.done = true;
console.log('a.js 执行完毕');
```

上面代码之中，`a.js` 脚本先输出一个 `done` 变量，然后加载另一个脚本文件 `b.js`。注意，此时 `a.js` 代码就停在这里，等待 `b.js` 执行完毕，再往下执行。

再看 `b.js` 的代码。

```
exports.done = false;
var a = require('./a.js');
console.log('在 b.js 之中, a.done = %j', a.done);
exports.done = true;
console.log('b.js 执行完毕');
```

上面代码之中，`b.js` 执行到第二行，就会去加载 `a.js`，这时，就发生了“循环加载”。系统会去 `a.js` 模块对应对象的 `exports` 属性取值，可是因为 `a.js` 还没有执行完，从 `exports` 属性只能取回已经执行的部分，而不是最后的值。

`a.js` 已经执行的部分，只有一行。

```
exports.done = false;
```

因此，对于 `b.js` 来说，它从 `a.js` 只输入一个变量 `done`，值为 `false`。

然后，`b.js` 接着往下执行，等到全部执行完毕，再把执行权交还给 `a.js`。于是，`a.js` 接着往下执行，直到执行完毕。我们写一个脚本 `main.js`，验证这个过程。

```
var a = require('./a.js');
var b = require('./b.js');
console.log('在 main.js 之中, a.done=%j, b.done=%j', a.done, b.done);
```

执行 `main.js`，运行结果如下。

```
$ node main.js
在 b.js 之中, a.done = false
b.js 执行完毕
在 a.js 之中, b.done = true
a.js 执行完毕
在 main.js 之中, a.done=true, b.done=true
```

上面的代码证明了两件事。一是，在 `b.js` 之中，`a.js` 没有执行完毕，只执行了第一行。二是，`main.js` 执行到第二行时，不会再次执行 `b.js`，而是输出缓存的 `b.js` 的执行结果，即它的第四行。

```
exports.done = true;
```

总之，CommonJS输入的是被输出值的拷贝，不是引用。

另外，由于CommonJS模块遇到循环加载时，返回的是当前已经执行的部分的值，而不是代码全部执行后的值，两者可能会有差异。所以，输入变量的时候，必须非常小心。

```
var a = require('a'); // 安全的写法
var foo = require('a').foo; // 危险的写法

exports.good = function (arg) {
  return a.foo('good', arg); // 使用的是 a.foo 的最新值
};

exports.bad = function (arg) {
  return foo('bad', arg); // 使用的是一个部分加载时的值
};
```

上面代码中，如果发生循环加载，`require('a').foo` 的值很可能后面会被改写，改用 `require('a')` 会更保险一点。

ES6模块的循环加载

ES6处理“循环加载”与CommonJS有本质的不同。ES6模块是动态引用，如果使用 `import` 从一个模块加载变量（即 `import foo from 'foo'`），那些变量不会被缓存，而是成为一个指向被加载模块的引用，需要开发者自己保证，真正取值的时候能够取到值。

请看下面这个例子。

```
// a.js如下
```

```
import {bar} from './b.js';
console.log('a.js');
console.log(bar);
export let foo = 'foo';

// b.js
import {foo} from './a.js';
console.log('b.js');
console.log(foo);
export let bar = 'bar';
```

上面代码中，`a.js` 加载 `b.js`，`b.js` 又加载 `a.js`，构成循环加载。执行 `a.js`，结果如下。

```
$ babel-node a.js
b.js
undefined
a.js
bar
```

上面代码中，由于 `a.js` 的第一行是加载 `b.js`，所以先执行的是 `b.js`。而 `b.js` 的第一行又是加载 `a.js`，这时由于 `a.js` 已经开始执行了，所以不会重复执行，而是继续往下执行 `b.js`，所以第一行输出的是 `b.js`。

接着，`b.js` 要打印变量 `foo`，这时 `a.js` 还没执行完，取不到 `foo` 的值，导致打印出来是 `undefined`。`b.js` 执行完，开始执行 `a.js`，这时就一切正常了。

再看一个稍微复杂的例子（摘自 Dr. Axel Rauschmayer 的《Exploring ES6》）。

```
// a.js
import {bar} from './b.js';
export function foo() {
  console.log('foo');
  bar();
  console.log('执行完毕');
}
foo();

// b.js
import {foo} from './a.js';
export function bar() {
  console.log('bar');
  if (Math.random() > 0.5) {
    foo();
  }
}
```

按照CommonJS规范，上面的代码是没法执行的。`a` 先加载 `b`，然后 `b` 又加载 `a`，这时 `a` 还没有任何执行结果，所以输出结果为 `null`，即对于 `b.js` 来说，变量 `foo` 的值等于 `null`，后面的 `foo()` 就会报错。

但是，ES6可以执行上面的代码。

```
$ babel-node a.js
foo
bar
执行完毕

// 执行结果也有可能是
foo
bar
foo
bar
执行完毕
执行完毕
```

上面代码中，`a.js`之所以能够执行，原因就在于ES6加载的变量，都是动态引用其所在的模块。只要引用存在，代码就能执行。

下面，我们详细分析这段代码的运行过程。

```
// a.js
// 这一行建立一个引用,
// 从`b.js`引用`bar`
import {bar} from './b.js';

export function foo() {
  // 执行时第一行输出 foo
  console.log('foo');
  // 到 b.js 执行 bar
  bar();
  console.log('执行完毕');
}
foo();

// b.js
// 建立`a.js`的`foo`引用
import {foo} from './a.js';

export function bar() {
  // 执行时, 第二行输出 bar
  console.log('bar');
  // 递归执行 foo, 一旦随机数
  // 小于等于0.5, 就停止执行
  if (Math.random() > 0.5) {
    foo();
  }
}
```

我们再来看ES6模块加载器[SystemJS](#)给出的一个例子。

```
// even.js
import { odd } from './odd'
export var counter = 0;
export function even(n) {
  counter++;
  return n == 0 || odd(n - 1);
}

// odd.js
import { even } from './even';
export function odd(n) {
  return n != 0 && even(n - 1);
}
```

上面代码中，`even.js`里面的函数`even`有一个参数`n`，只要不等于0，就会减去1，传入加载的`odd()`。`odd.js`也会做类似操作。

运行上面这段代码，结果如下。

```
$ babel-node
> import * as m from './even.js';
> m.even(10);
true
> m.counter
6
> m.even(20)
```

```
true
> m.counter
17
```

上面代码中，参数 `n` 从10变为0的过程中，`even()` 一共会执行6次，所以变量 `counter` 等于6。第二次调用 `even()` 时，参数 `n` 从20变为0，`even()` 一共会执行11次，加上前面的6次，所以变量 `counter` 等于17。

这个例子要是改写成CommonJS，就根本无法执行，会报错。

```
// even.js
var odd = require('./odd');
var counter = 0;
exports.counter = counter;
exports.even = function(n) {
  counter++;
  return n == 0 || odd(n - 1);
}

// odd.js
var even = require('./even').even;
module.exports = function(n) {
  return n != 0 && even(n - 1);
}
```

上面代码中，`even.js` 加载 `odd.js`，而 `odd.js` 又去加载 `even.js`，形成“循环加载”。这时，执行引擎就会输出 `even.js` 已经执行的部分（不存在任何结果），所以在 `odd.js` 之中，变量 `even` 等于 `null`，等到后面调用 `even(n-1)` 就会报错。

```
$ node
> var m = require('./even');
> m.even(10)
TypeError: even is not a function
```

跨模块常量

本书介绍 `const` 命令的时候说过，`const` 声明的常量只在当前代码块有效。如果想设置跨模块的常量（即跨多个文件），可以采用下面的写法。

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

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

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

如果要使用的常量非常多，可以建一个专门的 `constants` 目录，将各种常量写在不同的文件里面，保存在该目录下。

```
// constants/db.js
export const db = {
  url: 'http://my.couchdbserver.local:5984',
  admin_username: 'admin',
  admin_password: 'admin password'
};
```

```
// constants/user.js
export const users = ['root', 'admin', 'staff', 'ceo', 'chief', 'moderator'];
```

然后，将这些文件输出的常量，合并在 `index.js` 里面。

```
// constants/index.js
export {db} from './db';
export {users} from './users';
```

使用的时候，直接加载 `index.js` 就可以了。

```
// script.js
import {db, users} from './constants';
```

import()

上面说过了，`import` 语句会被JavaScript引擎静态分析，先于模块内的其他模块执行（叫做“更合适”）。所以，下面的代码会报错。

```
// 报错
if (x === 2) {
  import MyModual from './myModual';
}
```

上面代码中，引擎处理 `import` 语句是在执行之前，所以 `import` 语句放在 `if` 代码块之中毫无意义，因此会报句法错误，而不是执行时错误。

这样的设计，固然有利于编译器提高效率，但也导致无法在运行时加载模块。从长远来看，`import` 语句会取代 Node 的 `require` 方法，但是 `require` 是运行时加载模块，`import` 语句显然无法取代这种动态加载功能。

```
const path = './' + fileName;
const myModual = require(path);
```

上面的语句就是动态加载，`require` 到底加载哪一个模块，只有运行时才知道。`import` 语句做不到这一点。

因此，有一个[提案](#)，建议引入 `import()` 函数，完成动态加载。

```
import(specifier)
```

上面代码中，`import` 函数的参数 `specifier`，指定所要加载的模块的位置。`import` 语句能够接受什么参数，`import()` 函数就能接受什么参数，两者区别主要是后者为动态加载。

`import()` 返回一个 Promise 对象。下面是一个例子。

```
const main = document.querySelector('main');

import(`./section-modules/${someVariable}.js`)
  .then(module => {
    module.loadPageInto(main);
  })
  .catch(err => {
    main.textContent = err.message;
});
```

`import()` 函数可以用在任何地方，不仅仅是模块，非模块的脚本也可以使用。它是运行时执行，也就是说，什么时候运行到这一句，也会加载指定的模块。另外，`import()` 函数与所加载的模块没有静态连接关系，这点也是与 `import` 语句不相同。

`import()` 类似于 Node 的 `require` 方法，区别主要是前者是异步加载，后者是同步加载。

ES6模块的转码

浏览器目前还不支持ES6模块，为了现在就能使用，可以将转为ES5的写法。除了Babel可以用来转码之外，还有以下两个方法，也可以用来转码。

ES6 module transpiler

[ES6 module transpiler](#)是 square 公司开源的一个转码器，可以将 ES6 模块转为 CommonJS 模块或 AMD 模块的写法，从而在浏览器中使用。

首先，安装这个转码器。

```
$ npm install -g es6-module-transpiler
```

然后，使用 `compile-modules convert` 命令，将 ES6 模块文件转码。

```
$ compile-modules convert file1.js file2.js
```

`-o` 参数可以指定转码后的文件名。

```
$ compile-modules convert -o out.js file1.js
```

SystemJS

另一种解决方法是使用 [SystemJS](#)。它是一个垫片库（polyfill），可以在浏览器内加载 ES6 模块、AMD 模块和 CommonJS 模块，将其转为 ES5 格式。它在后台调用的是 Google 的 Traceur 转码器。

使用时，先在网页内载入 `system.js` 文件。

```
<script src="system.js"></script>
```

然后，使用 `System.import` 方法加载模块文件。

```
<script>
  System.import('./app.js');
</script>
```

上面代码中的 `./app`，指的是当前目录下的app.js文件。它可以是ES6模块文件，`System.import` 会自动将其转码。

需要注意的是，`System.import` 使用异步加载，返回一个 Promise 对象，可以针对这个对象编程。下面是一个模块文件。

```
// app/es6-file.js:

export class q {
  constructor() {
    this.es6 = 'hello';
  }
}
```

然后，在网页内加载这个模块文件。

```
<script>
  System.import('app/es6-file').then(function(m) {
```

```
    console.log(new m.q().es6); // hello
});
</script>
```

上面代码中，`System.import` 方法返回的是一个 Promise 对象，所以可以用 `then` 方法指定回调函数。

编程风格

本章探讨如何将ES6的新语法，运用到编码实践之中，与传统的JavaScript语法结合在一起，写出合理的、易于阅读和维护的代码。

多家公司和组织已经公开了它们的风格规范，具体可参阅[jscs.info](#)，下面的内容主要参考了Airbnb的JavaScript风格规范。

块级作用域

(1) `let` 取代 `var`

ES6提出了两个新的声明变量的命令：`let` 和 `const`。其中，`let` 完全可以取代 `var`，因为两者语义相同，而且 `let` 没有副作用。

```
'use strict';

if (true) {
  let x = 'hello';
}

for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

上面代码如果用 `var` 替代 `let`，实际上就声明了两个全局变量，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。

`const` 优于 `let` 有几个原因。一个是 `const` 可以提醒阅读程序的人，这个变量不应该改变；另一个是 `const` 比较符合函数式编程思想，运算不改变值，只是新建值，而且这样也有利于将来的分布式运算；最后一个原因是 JavaScript 编译器会对 `const` 进行优化，所以多使用 `const`，有利于提供程序的运行效率，也就是说 `let` 和 `const` 的本质区别，其实是编译器内部的处理不同。

```
// 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。

长远来看，JavaScript可能会有多线程的实现（比如Intel的River Trail那一类的项目），这时 `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。

字符串

静态字符串一律使用单引号或反引号，不使用双引号。动态字符串使用反引号。

```
// bad
const a = "foobar";
const b = 'foo' + a + 'bar';

// acceptable
const c = `foobar`;

// good
const a = 'foobar';
const b = `foo${a}bar`;
const c = 'foobar';
```

解构赋值

使用数组成员对变量赋值时，优先使用解构赋值。

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

函数的参数如果是对象的成员，优先使用解构赋值。

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;
}

// good
function getFullName(obj) {
  const { firstName, lastName } = obj;
}
```

```
// best
function getFullName({ firstName, lastName }) {
```

如果函数返回多个值，优先使用对象的解构赋值，而不是数组的解构赋值。这样便于以后添加返回值，以及更改返回值的顺序。

```
// bad
function processInput(input) {
  return [left, right, top, bottom];
}

// good
function processInput(input) {
  return { left, right, top, bottom };
}

const { left, right } = processInput(input);
```

对象

单行定义的对象，最后一个成员不以逗号结尾。多行定义的对象，最后一个成员以逗号结尾。

```
// bad
const a = { k1: v1, k2: v2, };
const b = {
  k1: v1,
  k2: v2
};

// good
const a = { k1: v1, k2: v2 };
const b = {
  k1: v1,
  k2: v2,
};
```

对象尽量静态化，一旦定义，就不得随意添加新的属性。如果添加属性不可避免，要使用 `Object.assign` 方法。

```
// bad
const a = {};
a.x = 3;

// if reshape unavoidable
const a = {};
Object.assign(a, { x: 3 });

// good
const a = { x: null };
a.x = 3;
```

如果对象的属性名是动态的，可以在创造对象的时候，使用属性表达式定义。

```
// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;

// good
```

```
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```

上面代码中，对象 `obj` 的最后一个属性名，需要计算得到。这时最好采用属性表达式，在新建 `obj` 的时候，将该属性与其他属性定义在一起。这样一来，所有属性就在一个地方定义了。

另外，对象的属性和方法，尽量采用简洁表达法，这样易于描述和书写。

```
var ref = 'some value';

// bad
const atom = {
  ref: ref,
  value: 1,
  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  ref,
  value: 1,
  addValue(value) {
    return atom.value + value;
  },
};
```

数组

使用扩展运算符 (...) 拷贝数组。

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

使用`Array.from`方法，将类似数组的对象转为数组。

```
const foo = document.querySelectorAll('.foo');
const nodes = Array.from(foo);
```

函数

立即执行函数可以写成箭头函数的形式。

```
(() => {
  console.log('Welcome to the Internet.');
})();
```

那些需要使用函数表达式的场合，尽量用箭头函数代替。因为这样更简洁，而且绑定了this。

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

// good
[1, 2, 3].map((x) => {
  return x * x;
});

// best
[1, 2, 3].map(x => x * x);
```

箭头函数取代 `Function.prototype.bind`，不应再用self/_this/that绑定 this。

```
// bad
const self = this;
const boundMethod = function(...params) {
  return method.apply(self, params);
}

// acceptable
const boundMethod = method.bind(this);

// best
const boundMethod = (...params) => method.apply(this, params);
```

简单的、单行的、不会复用的函数，建议采用箭头函数。如果函数体较为复杂，行数较多，还是应该采用传统的函数写法。

所有配置项都应该集中在一个对象，放在最后一个参数，布尔值不可以直接作为参数。

```
// bad
function divide(a, b, option = false) {}

// good
function divide(a, b, { option = false } = {}) {}
```

不要在函数体内使用arguments变量，使用rest运算符 (...) 代替。因为rest运算符显式表明你想要获取参数，而且arguments是一个类似数组的对象，而rest运算符可以提供一个真正的数组。

```
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

使用默认值语法设置函数参数的默认值。

```
// bad
function handleThings(opts) {
  opts = opts || {};
}

// good
function handleThings(opts = {}) {
  // ...
}
```

Map结构

注意区分Object和Map，只有模拟现实世界的实体对象时，才使用Object。如果只是需要 `key: value` 的数据结构，使用Map结构。因为Map有内建的遍历机制。

```
let map = new Map(arr);

for (let key of map.keys()) {
  console.log(key);
}

for (let value of map.values()) {
  console.log(value);
}

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
```

Class

总是用Class，取代需要prototype的操作。因为Class的写法更简洁，更易于理解。

```
// bad
function Queue(contents = []) {
  this._queue = [...contents];
}

Queue.prototype.pop = function() {
  const value = this._queue[0];
  this._queue.splice(0, 1);
  return value;
}

// good
class Queue {
  constructor(contents = []) {
    this._queue = [...contents];
  }
  pop() {
    const value = this._queue[0];
    this._queue.splice(0, 1);
    return value;
  }
}
```

使用 `extends` 实现继承，因为这样更简单，不会有破坏 `instanceof` 运算的危险。

```
// bad
const inherits = require('inherits');
function PeekableQueue(contents) {
```

```
    Queue.apply(this, contents);
}
inherits(PeekableQueue, Queue);
PeekableQueue.prototype.peek = function() {
  return this._queue[0];
}

// good
class PeekableQueue extends Queue {
  peek() {
    return this._queue[0];
  }
}
```

模块

首先，Module语法是JavaScript模块的标准写法，坚持使用这种写法。使用 `import` 取代 `require`。

```
// bad
const moduleA = require('moduleA');
const func1 = moduleA.func1;
const func2 = moduleA.func2;

// good
import { func1, func2 } from 'moduleA';
```

使用 `export` 取代 `module.exports`。

```
// commonJS的写法
var React = require('react');

var Breadcrumbs = React.createClass({
  render() {
    return <nav />;
  }
});

module.exports = Breadcrumbs;

// ES6的写法
import React from 'react';

const Breadcrumbs = React.createClass({
  render() {
    return <nav />;
  }
});

export default Breadcrumbs
```

如果模块只有一个输出值，就使用 `export default`，如果模块有多个输出值，就不使用 `export default`，不要 `export default` 与普通的 `export` 同时使用。

不要在模块输入中使用通配符。因为这样可以确保你的模块之中，有一个默认输出（`export default`）。

```
// bad
import * as myObject './importModule';

// good
import myObject from './importModule';
```

如果模块默认输出一个函数，函数名的首字母应该小写。

```
function makeStyleGuide() {  
}  
  
export default makeStyleGuide;
```

如果模块默认输出一个对象，对象名的首字母应该大写。

```
const StyleGuide = {  
  es6: {  
  }  
};  
  
export default StyleGuide;
```

ESLint的使用

ESLint是一个语法规则和代码风格的检查工具，可以用来保证写出语法正确、风格统一的代码。

首先，安装ESLint。

```
$ npm i -g eslint
```

然后，安装Airbnb语法规则。

```
$ npm i -g eslint-config-airbnb
```

最后，在项目的根目录下新建一个 `.eslintrc` 文件，配置ESLint。

```
{  
  "extends": "eslint-config-airbnb"  
}
```

现在就可以检查，当前项目的代码是否符合预设的规则。

`index.js` 文件的代码如下。

```
var unusued = 'I have no purpose!';  
  
function greet() {  
  var message = 'Hello, World!';  
  alert(message);  
}  
  
greet();
```

使用ESLint检查这个文件。

```
$ eslint index.js  
index.js  
  1:5  error  unusued is defined but never used          no-unused-vars  
  4:5  error  Expected indentation of 2 characters but found 4  indent  
  5:5  error  Expected indentation of 2 characters but found 4  indent  
  
✖ 3 problems (3 errors, 0 warnings)
```

上面代码说明，原文件有三个错误，一个是定义了变量，却没有使用，另外两个是行首缩进为4个空格，而不是规定的2个空格。

二进制数组

二进制数组（`ArrayBuffer` 对象、`TypedArray` 视图和 `DataView` 视图）是 JavaScript 操作二进制数据的一个接口。这些对象早就存在，属于独立的规格（2011年2月发布），ES6 将它们纳入了 ECMAScript 规格，并且增加了新的方法。

这个接口的原始设计目的，与 WebGL 项目有关。所谓 WebGL，就是指浏览器与显卡之间的通信接口，为了满足 JavaScript 与显卡之间大量的、实时的数据交换，它们之间的数据通信必须是二进制的，而不能是传统的文本格式。文本格式传递一个32位整数，两端的 JavaScript 脚本与显卡都要进行格式转化，将非常耗时。这时要是存在一种机制，可以像 C 语言那样，直接操作字节，将4个字节的32位整数，以二进制形式原封不动地送入显卡，脚本的性能就会大幅提升。

二进制数组就是在这种背景下诞生的。它很像 C 语言的数组，允许开发者以数组下标的形式，直接操作内存，大大增强了 JavaScript 处理二进制数据的能力，使得开发者有可能通过 JavaScript 与操作系统的原生接口进行二进制通信。

二进制数组由三类对象组成。

(1) `ArrayBuffer` 对象：代表内存之中的一段二进制数据，可以通过“视图”进行操作。“视图”部署了数组接口，这意味着，可以用数组的方法操作内存。

(2) `TypedArray` 视图：共包括9种类型的视图，比如 `Uint8Array`（无符号8位整数）数组视图，`Int16Array`（16位整数）数组视图，`Float32Array`（32位浮点数）数组视图等等。

(3) `DataView` 视图：可以自定义复合格式的视图，比如第一个字节是 `Uint8`（无符号8位整数）、第二、三个字节是 `Int16`（16位整数）、第四个字节开始是 `Float32`（32位浮点数）等等，此外还可以自定义字节序。

简单说，`ArrayBuffer` 对象代表原始的二进制数据，`TypedArray` 视图用来读写简单类型的二进制数据，`DataView` 视图用来读写复杂类型的二进制数据。

`TypedArray` 视图支持的数据类型一共有9种（`DataView` 视图支持除 `Uint8C` 以外的其他8种）。

| 数据类型 | 字节长度 | 含义 | 对应的C语言类型 |
|----------------------|------|------------------|-----------------------------|
| <code>Int8</code> | 1 | 8位带符号整数 | <code>signed char</code> |
| <code>Uint8</code> | 1 | 8位不带符号整数 | <code>unsigned char</code> |
| <code>Uint8C</code> | 1 | 8位不带符号整数（自动过滤溢出） | <code>unsigned char</code> |
| <code>Int16</code> | 2 | 16位带符号整数 | <code>short</code> |
| <code>Uint16</code> | 2 | 16位不带符号整数 | <code>unsigned short</code> |
| <code>Int32</code> | 4 | 32位带符号整数 | <code>int</code> |
| <code>Uint32</code> | 4 | 32位不带符号的整数 | <code>unsigned int</code> |
| <code>Float32</code> | 4 | 32位浮点数 | <code>float</code> |
| <code>Float64</code> | 8 | 64位浮点数 | <code>double</code> |

注意，二进制数组并不是真正的数组，而是类似数组的对象。

很多浏览器操作的 API，用到了二进制数组操作二进制数据，下面是其中的几个。

- File API
- XMLHttpRequest
- Fetch API
- Canvas
- WebSockets

ArrayBuffer对象

概述

`ArrayBuffer` 对象代表储存二进制数据的一段内存，它不能直接读写，只能通过视图（`TypedArray`视图和 `DataView` 视图）来读写，视图的作用是以指定格式解读二进制数据。

`ArrayBuffer` 也是一个构造函数，可以分配一段可以存放数据的连续内存区域。

```
var buf = new ArrayBuffer(32);
```

上面代码生成了一段32字节的内存区域，每个字节的值默认都是0。可以看到，`ArrayBuffer` 构造函数的参数是所需要的内存大小（单位字节）。

为了读写这段内容，需要为它指定视图。`DataView` 视图的创建，需要提供 `ArrayBuffer` 对象实例作为参数。

```
var buf = new ArrayBuffer(32);
var dataView = new DataView(buf);
dataView.getUint8(0) // 0
```

上面代码对一段32字节的内存，建立 `DataView` 视图，然后以不带符号的8位整数格式，读取第一个元素，结果得到0，因为原始内存的 `ArrayBuffer` 对象，默认所有位都是0。

另一种`TypedArray`视图，与 `DataView` 视图的一个区别是，它不是一个构造函数，而是一组构造函数，代表不同的数据格式。

```
var buffer = new ArrayBuffer(12);

var x1 = new Int32Array(buffer);
x1[0] = 1;
var x2 = new Uint8Array(buffer);
x2[0] = 2;

x1[0] // 2
```

上面代码对同一段内存，分别建立两种视图：32位带符号整数（`Int32Array` 构造函数）和8位不带符号整数（`Uint8Array` 构造函数）。由于两个视图对应的是同一段内存，一个视图修改底层内存，会影响到另一个视图。

`TypedArray`视图的构造函数，除了接受 `ArrayBuffer` 实例作为参数，还可以接受普通数组作为参数，直接分配内存生成底层的 `ArrayBuffer` 实例，并同时完成对这段内存的赋值。

```
var typedArray = new Uint8Array([0,1,2]);
typedArray.length // 3

typedArray[0] = 5;
typedArray // [5, 1, 2]
```

上面代码使用`TypedArray`视图的 `Uint8Array` 构造函数，新建一个不带符号的8位整数视图。可以看到，`Uint8Array` 直接使用普通数组作为参数，对底层内存的赋值同时完成。

ArrayBuffer.prototype.byteLength

`ArrayBuffer` 实例的 `byteLength` 属性，返回所分配的内存区域的字节长度。

```
var buffer = new ArrayBuffer(32);
buffer.byteLength
// 32
```

如果要分配的内存区域很大，有可能分配失败（因为没有那么多的连续空余内存），所以有必要检查是否分配成功。

```
if (buffer.byteLength === n) {
  // 成功
} else {
  // 失败
}
```

ArrayBuffer.prototype.slice()

`ArrayBuffer` 实例有一个 `slice` 方法，允许将内存区域的一部分，拷贝生成一个新的 `ArrayBuffer` 对象。

```
var buffer = new ArrayBuffer(8);
var newBuffer = buffer.slice(0, 3);
```

上面代码拷贝 `buffer` 对象的前3个字节（从0开始，到第3个字节前面结束），生成一个新的 `ArrayBuffer` 对象。

`slice` 方法其实包含两步，第一步是先分配一段新内存，第二步是将原来那个 `ArrayBuffer` 对象拷贝过去。

`slice` 方法接受两个参数，第一个参数表示拷贝开始的字节序号（含该字节），第二个参数表示拷贝截止的字节序号（不含该字节）。如果省略第二个参数，则默认到原 `ArrayBuffer` 对象的结尾。

除了 `slice` 方法，`ArrayBuffer` 对象不提供任何直接读写内存的方法，只允许在其上方建立视图，然后通过视图读写。

ArrayBuffer.isView()

`ArrayBuffer` 有一个静态方法 `isView`，返回一个布尔值，表示参数是否为 `ArrayBuffer` 的视图实例。这个方法大致相当于判断参数，是否为 `TypedArray` 实例或 `DataView` 实例。

```
var buffer = new ArrayBuffer(8);
ArrayBuffer.isView(buffer) // false

var v = new Int32Array(buffer);
ArrayBuffer.isView(v) // true
```

TypedArray视图

概述

`ArrayBuffer` 对象作为内存区域，可以存放多种类型的数据。同一段内存，不同数据有不同的解读方式，这就叫做“视图”（view）。`ArrayBuffer` 有两种视图，一种是 `TypedArray` 视图，另一种是 `DataView` 视图。前者的数组成员都是同一个数据类型，后者的数组成员可以是不同的数据类型。

目前，`TypedArray` 视图一共包括9种类型，每一种视图都是一种构造函数。

- `Int8Array`：8位有符号整数，长度1个字节。
- `Uint8Array`：8位无符号整数，长度1个字节。
- `Uint8ClampedArray`：8位无符号整数，长度1个字节，溢出处理不同。
- `Int16Array`：16位有符号整数，长度2个字节。
- `Uint16Array`：16位无符号整数，长度2个字节。

- `Int32Array` : 32位有符号整数，长度4个字节。
- `Uint32Array` : 32位无符号整数，长度4个字节。
- `Float32Array` : 32位浮点数，长度4个字节。
- `Float64Array` : 64位浮点数，长度8个字节。

这9个构造函数生成的数组，统称为TypedArray视图。它们很像普通数组，都有`length`属性，都能用方括号运算符(`[]`)获取单个元素，所有数组的方法，在它们上面都能使用。普通数组与TypedArray数组的差异主要在以下方面。

- TypedArray数组的所有成员，都是同一种类型。
- TypedArray数组的成员是连续的，不会有空位。
- TypedArray数组成员的默认值为0。比如，`new Array(10)`返回一个普通数组，里面没有任何成员，只是10个空位；`new Uint8Array(10)`返回一个TypedArray数组，里面10个成员都是0。
- TypedArray数组只是一层视图，本身不储存数据，它的数据都储存在底层的`ArrayBuffer`对象之中，要获取底层对象必须使用`buffer`属性。

构造函数

TypedArray数组提供9种构造函数，用来生成相应类型的数组实例。

构造函数有多种用法。

(1) `TypedArray(buffer, byteOffset=0, length?)`

同一个`ArrayBuffer`对象之上，可以根据不同的数据类型，建立多个视图。

```
// 创建一个8字节的ArrayBuffer
var b = new ArrayBuffer(8);

// 创建一个指向b的Int32视图，开始于字节0，直到缓冲区的末尾
var v1 = new Int32Array(b);

// 创建一个指向b的Uint8视图，开始于字节2，直到缓冲区的末尾
var v2 = new Uint8Array(b, 2);

// 创建一个指向b的Int16视图，开始于字节2，长度为2
var v3 = new Int16Array(b, 2, 2);
```

上面代码在一段长度为8个字节的内存(`b`)之上，生成了三个视图：`v1`、`v2`和`v3`。

视图的构造函数可以接受三个参数：

- 第一个参数（必需）：视图对应的底层`ArrayBuffer`对象。
- 第二个参数（可选）：视图开始的字节序号，默认从0开始。
- 第三个参数（可选）：视图包含的数据个数，默认直到本段内存区域结束。

因此，`v1`、`v2`和`v3`是重叠的：`v1[0]`是一个32位整数，指向字节0~字节3；`v2[0]`是一个8位无符号整数，指向字节2；`v3[0]`是一个16位整数，指向字节2~字节3。只要任何一个视图对内存有所修改，就会在另外两个视图上反应出来。

注意，`byteOffset`必须与所要建立的数据类型一致，否则会报错。

```
var buffer = new ArrayBuffer(8);
var i16 = new Int16Array(buffer, 1);
// Uncaught RangeError: start offset of Int16Array should be a multiple of 2
```

上面代码中，新生成一个8个字节的`ArrayBuffer`对象，然后在这个对象的第一个字节，建立带符号的16位整数视图，结果报错。因为，带符号的16位整数需要两个字节，所以`byteOffset`参数必须能够被2整除。

如果想从任意字节开始解读`ArrayBuffer`对象，必须使用`DataView`视图，因为TypedArray视图只提供9种固定的解

读格式。

(2) TypedArray(length)

视图还可以不通过 `ArrayBuffer` 对象，直接分配内存而生成。

```
var f64a = new Float64Array(8);
f64a[0] = 10;
f64a[1] = 20;
f64a[2] = f64a[0] + f64a[1];
```

上面代码生成一个8个成员的 `Float64Array` 数组（共64字节），然后依次对每个成员赋值。这时，视图构造函数的参数就是成员的个数。可以看到，视图数组的赋值操作与普通数组的操作毫无两样。

(3) TypedArray(typedArray)

`TypedArray`数组的构造函数，可以接受另一个`TypedArray`实例作为参数。

```
var typedArray = new Int8Array(new Uint8Array(4));
```

上面代码中，`Int8Array` 构造函数接受一个 `Uint8Array` 实例作为参数。

注意，此时生成的新数组，只是复制了参数数组的值，对应的底层内存是不一样的。新数组会开辟一段新的内存储存数据，不会在原数组的内存之上建立视图。

```
var x = new Int8Array([1, 1]);
var y = new Int8Array(x);
x[0] // 1
y[0] // 1

x[0] = 2;
y[0] // 1
```

上面代码中，数组 `y` 是以数组 `x` 为模板而生成的，当 `x` 变动的时候，`y` 并没有变动。

如果想基于同一段内存，构造不同的视图，可以采用下面的写法。

```
var x = new Int8Array([1, 1]);
var y = new Int8Array(x.buffer);
x[0] // 1
y[0] // 1

x[0] = 2;
y[0] // 2
```

(4) TypedArray(arrayLikeObject)

构造函数的参数也可以是一个普通数组，然后直接生成`TypedArray`实例。

```
var typedArray = new Uint8Array([1, 2, 3, 4]);
```

注意，这时`TypedArray`视图会重新开辟内存，不会在原数组的内存上建立视图。

上面代码从一个普通的数组，生成一个8位无符号整数的`TypedArray`实例。

`TypedArray`数组也可以转换回普通数组。

```
var normalArray = Array.prototype.slice.call(typedArray);
```

普通数组的操作方法和属性，对TypedArray数组完全适用。

- `TypedArray.prototype.copyWithin(target, start[, end = this.length])`
- `TypedArray.prototype.entries()`
- `TypedArray.prototype.every(callbackfn, thisArg?)`
- `TypedArray.prototype.fill(value, start=0, end=this.length)`
- `TypedArray.prototype.filter(callbackfn, thisArg?)`
- `TypedArray.prototype.find(predicate, thisArg?)`
- `TypedArray.prototype.findIndex(predicate, thisArg?)`
- `TypedArray.prototype.forEach(callbackfn, thisArg?)`
- `TypedArray.prototype.indexOf(searchElement, fromIndex=0)`
- `TypedArray.prototype.join(separator)`
- `TypedArray.prototype.keys()`
- `TypedArray.prototype.lastIndexOf(searchElement, fromIndex?)`
- `TypedArray.prototype.map(callbackfn, thisArg?)`
- `TypedArray.prototype.reduce(callbackfn, initialValue?)`
- `TypedArray.prototype.reduceRight(callbackfn, initialValue?)`
- `TypedArray.prototype.reverse()`
- `TypedArray.prototype.slice(start=0, end=this.length)`
- `TypedArray.prototype.some(callbackfn, thisArg?)`
- `TypedArray.prototype.sort(comparefn)`
- `TypedArray.prototype.toLocaleString(reserved1?, reserved2?)`
- `TypedArray.prototype.toString()`
- `TypedArray.prototype.values()`

上面所有方法的用法，请参阅数组方法的介绍，这里不再重复了。

注意，`TypedArray`数组没有 `concat` 方法。如果想要合并多个`TypedArray`数组，可以用下面这个函数。

```
function concatenate(resultConstructor, ...arrays) {  
  let totalLength = 0;  
  for (let arr of arrays) {  
    totalLength += arr.length;  
  }  
  let result = new resultConstructor(totalLength);  
  let offset = 0;  
  for (let arr of arrays) {  
    result.set(arr, offset);  
    offset += arr.length;  
  }  
  return result;  
}  
  
concatenate(Uint8Array, Uint8Array.of(1, 2), Uint8Array.of(3, 4))  
// Uint8Array [1, 2, 3, 4]
```

另外，`TypedArray`数组与普通数组一样，部署了`Iterator`接口，所以可以被遍历。

```
let ui8 = Uint8Array.of(0, 1, 2);  
for (let byte of ui8) {  
  console.log(byte);  
}  
// 0  
// 1  
// 2
```

字节序

字节序指的是数值在内存中的表示方式。

```
var buffer = new ArrayBuffer(16);
var int32View = new Int32Array(buffer);

for (var i = 0; i < int32View.length; i++) {
    int32View[i] = i * 2;
}
```

上面代码生成一个16字节的 `ArrayBuffer` 对象，然后在它的基础上，建立了一个32位整数的视图。由于每个32位整数占据4个字节，所以一共可以写入4个整数，依次为0, 2, 4, 6。

如果在这段数据上接着建立一个16位整数的视图，则可以读出完全不一样的结果。

```
var int16View = new Int16Array(buffer);

for (var i = 0; i < int16View.length; i++) {
    console.log("Entry " + i + ": " + int16View[i]);
}

// Entry 0: 0
// Entry 1: 0
// Entry 2: 2
// Entry 3: 0
// Entry 4: 4
// Entry 5: 0
// Entry 6: 6
// Entry 7: 0
```

由于每个16位整数占据2个字节，所以整个 `ArrayBuffer` 对象现在分成8段。然后，由于x86体系的计算机都采用小端字节序（little endian），相对重要的字节排在后面的内存地址，相对不重要字节排在前面的内存地址，所以就得到了上面的结果。

比如，一个占据四个字节的16进制数 `0x12345678`，决定其大小的最重要的字节是“12”，最不重要的是“78”。小端字节序将最不重要的字节排在前面，储存顺序就是 `78563412`；大端字节序则完全相反，将最重要的字节排在前面，储存顺序就是 `12345678`。目前，所有个人电脑几乎都是小端字节序，所以 `TypedArray` 数组内部也采用小端字节序读写数据，或者更准确的说，按照本机操作系统设定的字节序读写数据。

这并不意味大端字节序不重要，事实上，很多网络设备和特定的操作系统采用的是大端字节序。这就带来一个严重的问题：如果一段数据是大端字节序，`TypedArray` 数组将无法正确解析，因为它只能处理小端字节序！为了解决这个问题，JavaScript 引入 `DataView` 对象，可以设定字节序，下文会详细介绍。

下面是另一个例子。

```
// 假定某段buffer包含如下字节 [0x02, 0x01, 0x03, 0x07]
var buffer = new ArrayBuffer(4);
var v1 = new Uint8Array(buffer);
v1[0] = 2;
v1[1] = 1;
v1[2] = 3;
v1[3] = 7;

var uInt16View = new Uint16Array(buffer);

// 计算机采用小端字节序
// 所以头两个字节等于258
if (uInt16View[0] === 258) {
    console.log('OK'); // "OK"
}

// 赋值运算
```

```
uInt16View[0] = 255; // 字节变为[0xFF, 0x00, 0x03, 0x07]
uInt16View[0] = 0xff05; // 字节变为[0x05, 0xFF, 0x03, 0x07]
uInt16View[1] = 0x0210; // 字节变为[0x05, 0xFF, 0x10, 0x02]
```

下面的函数可以用来判断，当前视图是小端字节序，还是大端字节序。

```
const BIG_ENDIAN = Symbol('BIG_ENDIAN');
const LITTLE_ENDIAN = Symbol('LITTLE_ENDIAN');

function getPlatformEndianness() {
  let arr32 = Uint32Array.of(0x12345678);
  let arr8 = new Uint8Array(arr32.buffer);
  switch ((arr8[0]*0x1000000) + (arr8[1]*0x10000) + (arr8[2]*0x100) + (arr8[3])) {
    case 0x12345678:
      return BIG_ENDIAN;
    case 0x78563412:
      return LITTLE_ENDIAN;
    default:
      throw new Error('Unknown endianness');
  }
}
```

总之，与普通数组相比，`TypedArray`数组的最大优点就是可以直接操作内存，不需要数据类型转换，所以速度快得多。

BYTES_PER_ELEMENT属性

每一种视图的构造函数，都有一个 `BYTES_PER_ELEMENT` 属性，表示这种数据类型占据的字节数。

```
Int8Array.BYTES_PER_ELEMENT // 1
Uint8Array.BYTES_PER_ELEMENT // 1
Int16Array.BYTES_PER_ELEMENT // 2
Uint16Array.BYTES_PER_ELEMENT // 2
Int32Array.BYTES_PER_ELEMENT // 4
Uint32Array.BYTES_PER_ELEMENT // 4
Float32Array.BYTES_PER_ELEMENT // 4
Float64Array.BYTES_PER_ELEMENT // 8
```

这个属性在`TypedArray`实例上也能获取，即有 `TypedArray.prototype.BYTES_PER_ELEMENT`。

ArrayBuffer与字符串的互相转换

`ArrayBuffer` 转为字符串，或者字符串转为 `ArrayBuffer`，有一个前提，即字符串的编码方法是确定的。假定字符串采用UTF-16编码（JavaScript的内部编码方式），可以自己编写转换函数。

```
// ArrayBuffer转为字符串，参数为ArrayBuffer对象
function ab2str(buf) {
  return String.fromCharCode.apply(null, new Uint16Array(buf));
}

// 字符串转为ArrayBuffer对象，参数为字符串
function str2ab(str) {
  var buf = new ArrayBuffer(str.length * 2); // 每个字符占用2个字节
  var bufView = new Uint16Array(buf);
  for (var i = 0, strLen = str.length; i < strLen; i++) {
    bufView[i] = str.charCodeAt(i);
  }
  return buf;
}
```

溢出

不同的视图类型，所能容纳的数值范围是确定的。超出这个范围，就会出现溢出。比如，8位视图只能容纳一个8位的二进制值，如果放入一个9位的值，就会溢出。

TypedArray数组的溢出处理规则，简单来说，就是抛弃溢出的位，然后按照视图类型进行解释。

```
var uint8 = new Uint8Array(1);

uint8[0] = 256;
uint8[0] // 0

uint8[0] = -1;
uint8[0] // 255
```

上面代码中，`uint8` 是一个8位视图，而256的二进制形式是一个9位的值 `100000000`，这时就会发生溢出。根据规则，只会保留后8位，即 `00000000`。`uint8` 视图的解释规则是无符号的8位整数，所以 `00000000` 就是 `0`。

负数在计算机内部采用“2的补码”表示，也就是说，将对应的正数值进行否运算，然后加 `1`。比如，`-1` 对应的正值是 `1`，进行否运算以后，得到 `11111110`，再加上 `1` 就是补码形式 `11111111`。`uint8` 按照无符号的8位整数解释 `11111111`，返回结果就是 `255`。

一个简单转换规则，可以这样表示。

- 正向溢出（overflow）：当输入值大于当前数据类型的最大值，结果等于当前数据类型的最小值加上余值，再减去 `1`。
- 负向溢出（underflow）：当输入值小于当前数据类型的最小值，结果等于当前数据类型的的最大值减去余值，再加上 `1`。

上面的“余值”就是模运算的结果，即 JavaScript 里面的 `%` 运算符的结果。

```
12 % 4 // 0
12 % 5 // 2
```

上面代码中，`12`除以`4`是没有余值的，而除以`5`会得到余值`2`。

请看下面的例子。

```
var int8 = new Int8Array(1);

int8[0] = 128;
int8[0] // -128

int8[0] = -129;
int8[0] // 127
```

上面例子中，`int8` 是一个带符号的8位整数视图，它的最大值是`127`，最小值是`-128`。输入值为 `128` 时，相当于正向溢出 `1`，根据“最小值加上余值（`128`除以`127`的余值是`1`），再减去`1`”的规则，就会返回 `-128`；输入值为 `-129` 时，相当于负向溢出 `1`，根据“最大值减去余值（`-129`除以`-127`的余值是`2`），再加上`1`”的规则，就会返回 `127`。

`Uint8ClampedArray` 视图的溢出规则，与上面的规则不同。它规定，凡是发生正向溢出，该值一律等于当前数据类型的最大值，即`255`；如果发生负向溢出，该值一律等于当前数据类型的最小值，即`0`。

```
var uint8c = new Uint8ClampedArray(1);

uint8c[0] = 256;
uint8c[0] // 255

uint8c[0] = -1;
uint8c[0] // 0
```

上面例子中，`uint8C` 是一个 `Uint8ClampedArray` 视图，正向溢出时都返回255，负向溢出都返回0。

TypedArray.prototype.buffer

TypedArray实例的 `buffer` 属性，返回整段内存区域对应的 `ArrayBuffer` 对象。该属性为只读属性。

```
var a = new Float32Array(64);
var b = new Uint8Array(a.buffer);
```

上面代码的 `a` 视图对象和 `b` 视图对象，对应同一个 `ArrayBuffer` 对象，即同一段内存。

TypedArray.prototype.byteLength, TypedArray.prototype.byteOffset

`byteLength` 属性返回TypedArray数组占据的内存长度，单位为字节。`byteOffset` 属性返回TypedArray数组从底层 `ArrayBuffer` 对象的那个字节开始。这两个属性都是只读属性。

```
var b = new ArrayBuffer(8);

var v1 = new Int32Array(b);
var v2 = new Uint8Array(b, 2);
var v3 = new Int16Array(b, 2, 2);

v1.byteLength // 8
v2.byteLength // 6
v3.byteLength // 4

v1.byteOffset // 0
v2.byteOffset // 2
v3.byteOffset // 2
```

TypedArray.prototype.length

`length` 属性表示TypedArray数组含有多少个成员。注意将 `byteLength` 属性和 `length` 属性区分，前者是字节长度，后者是成员长度。

```
var a = new Int16Array(8);

a.length // 8
a.byteLength // 16
```

TypedArray.prototype.set()

TypedArray数组的 `set` 方法用于复制数组（普通数组或TypedArray数组），也就是将一段内容完全复制到另一段内存。

```
var a = new Uint8Array(8);
var b = new Uint8Array(8);

b.set(a);
```

上面代码复制 `a` 数组的内容到 `b` 数组，它是整段内存的复制，比一个个拷贝成员的那种复制快得多。

`set` 方法还可以接受第二个参数，表示从 `b` 对象的哪一个成员开始复制 `a` 对象。

```
var a = new Uint16Array(8);
var b = new Uint16Array(10);
```

```
b.set(a, 2)
```

上面代码的 `b` 数组比 `a` 数组多两个成员，所以从 `b[2]` 开始复制。

TypedArray.prototype.subarray()

`subarray` 方法是对于TypedArray数组的一部分，再建立一个新的视图。

```
var a = new Uint16Array(8);
var b = a.subarray(2,3);

a.byteLength // 16
b.byteLength // 2
```

`subarray` 方法的第一个参数是起始的成员序号，第二个参数是结束的成员序号（不含该成员），如果省略则包含剩余的全部成员。所以，上面代码的 `a.subarray(2,3)`，意味着`b`只包含 `a[2]` 一个成员，字节长度为2。

TypedArray.prototype.slice()

TypeArray实例的 `slice` 方法，可以返回一个指定位置的新的TypedArray实例。

```
let ui8 = Uint8Array.of(0, 1, 2);
ui8.slice(-1)
// Uint8Array [ 2 ]
```

上面代码中，`ui8` 是8位无符号整数数组视图的一个实例。它的 `slice` 方法可以从当前视图之中，返回一个新的视图实例。

`slice` 方法的参数，表示原数组的具体位置，开始生成新数组。负值表示逆向的位置，即-1为倒数第一个位置，-2表示倒数第二个位置，以此类推。

TypedArray.of()

TypedArray数组的所有构造函数，都有一个静态方法 `of`，用于将参数转为一个TypedArray实例。

```
Float32Array.of(0.151, -8, 3.7)
// Float32Array [ 0.151, -8, 3.7 ]
```

下面三种方法都会生成同样一个TypedArray数组。

```
// 方法一
let tarr = new Uint8Array([1,2,3]);

// 方法二
let tarr = Uint8Array.of(1,2,3);

// 方法三
let tarr = new Uint8Array(3);
tarr[0] = 1;
tarr[1] = 2;
tarr[2] = 3;
```

TypedArray.from()

静态方法 `from` 接受一个可遍历的数据结构（比如数组）作为参数，返回一个基于这个结构的TypedArray实例。

```
Uint16Array.from([0, 1, 2])
```

```
// Uint16Array [ 0, 1, 2 ]
```

这个方法还可以将一种TypedArray实例，转为另一种。

```
var ui16 = Uint16Array.from(Uint8Array.of(0, 1, 2));
ui16 instanceof Uint16Array // true
```

`from` 方法还可以接受一个函数，作为第二个参数，用来对每个元素进行遍历，功能类似 `map` 方法。

```
Int8Array.of(127, 126, 125).map(x => 2 * x)
// Int8Array [ -2, -4, -6 ]

Int16Array.from(Int8Array.of(127, 126, 125), x => 2 * x)
// Int16Array [ 254, 252, 250 ]
```

上面的例子中，`from` 方法没有发生溢出，这说明遍历不是针对原来的8位整数数组。也就是说，`from` 会将第一个参数指定的TypedArray数组，拷贝到另一段内存之中，处理之后再将结果转成指定的数据格式。

复合视图

由于视图的构造函数可以指定起始位置和长度，所以在同一段内存之中，可以依次存放不同类型的数据，这叫做“复合视图”。

```
var buffer = new ArrayBuffer(24);

var idView = new Uint32Array(buffer, 0, 1);
var usernameView = new Uint8Array(buffer, 4, 16);
var amountDueView = new Float32Array(buffer, 20, 1);
```

上面代码将一个24字节长度的 `ArrayBuffer` 对象，分成三个部分：

- 字节0到字节3：1个32位无符号整数
- 字节4到字节19：16个8位整数
- 字节20到字节23：1个32位浮点数

这种数据结构可以用如下的C语言描述：

```
struct someStruct {
  unsigned long id;
  char username[16];
  float amountDue;
};
```

DataView视图

如果一段数据包括多种类型（比如服务器传来的HTTP数据），这时除了建立 `ArrayBuffer` 对象的复合视图以外，还可以通过 `DataView` 视图进行操作。

`DataView` 视图提供更多操作选项，而且支持设定字节序。本来，在设计目的上，`ArrayBuffer` 对象的各种 TypedArray视图，是用来向网卡、声卡之类的本机设备传送数据，所以使用本机的字节序就可以了；而 `DataView` 视图的设计目的，是用来处理网络设备传来的数据，所以大端字节序或小端字节序是可以自行设定的。

`DataView` 视图本身也是构造函数，接受一个 `ArrayBuffer` 对象作为参数，生成视图。

```
DataView(ArrayBuffer buffer [, 字节起始位置 [, 长度]]);
```

下面是一个例子。

```
var buffer = new ArrayBuffer(24);
var dv = new DataView(buffer);
```

`DataView` 实例有以下属性，含义与`TypedArray`实例的同名方法相同。

- `DataView.prototype.buffer`：返回对应的`ArrayBuffer`对象
- `DataView.prototype.byteLength`：返回占据的内存字节长度
- `DataView.prototype.byteOffset`：返回当前视图从对应的`ArrayBuffer`对象的那个字节开始

`DataView` 实例提供8个方法读取内存。

- `getInt8`：读取1个字节，返回一个8位整数。
- `getUint8`：读取1个字节，返回一个无符号的8位整数。
- `getInt16`：读取2个字节，返回一个16位整数。
- `getUint16`：读取2个字节，返回一个无符号的16位整数。
- `getInt32`：读取4个字节，返回一个32位整数。
- `getUint32`：读取4个字节，返回一个无符号的32位整数。
- `getFloat32`：读取4个字节，返回一个32位浮点数。
- `getFloat64`：读取8个字节，返回一个64位浮点数。

这一系列 `get` 方法的参数都是一个字节序号（不能是负数，否则会报错），表示从哪个字节开始读取。

```
var buffer = new ArrayBuffer(24);
var dv = new DataView(buffer);

// 从第1个字节读取一个8位无符号整数
var v1 = dv.getUint8(0);

// 从第2个字节读取一个16位无符号整数
var v2 = dv.getUint16(1);

// 从第4个字节读取一个16位无符号整数
var v3 = dv.getUint16(3);
```

上面代码读取了 `ArrayBuffer` 对象的前5个字节，其中有一个8位整数和两个十六位整数。

如果一次读取两个或两个以上字节，就必须明确数据的存储方式，到底是小端字节序还是大端字节序。默认情况下，`DataView` 的 `get` 方法使用大端字节序解读数据，如果需要使用小端字节序解读，必须在 `get` 方法的第二个参数指定 `true`。

```
// 小端字节序
var v1 = dv.getUint16(1, true);

// 大端字节序
var v2 = dv.getUint16(3, false);

// 大端字节序
var v3 = dv.getUint16(3);
```

`DataView` 视图提供8个方法写入内存。

- `setInt8`：写入1个字节的8位整数。
- `setUint8`：写入1个字节的8位无符号整数。
- `setInt16`：写入2个字节的16位整数。
- `setUint16`：写入2个字节的16位无符号整数。
- `setInt32`：写入4个字节的32位整数。
- `setUint32`：写入4个字节的32位无符号整数。
- `setFloat32`：写入4个字节的32位浮点数。

- `setFloat64` : 写入8个字节的64位浮点数。

这一系列 `set` 方法，接受两个参数，第一个参数是字节序号，表示从哪个字节开始写入，第二个参数为写入的数据。对于那些写入两个或两个以上字节的方法，需要指定第三个参数，`false` 或者 `undefined` 表示使用大端字节序写入，`true` 表示使用小端字节序写入。

```
// 在第1个字节，以大端字节序写入值为25的32位整数
dv.setInt32(0, 25, false);

// 在第5个字节，以大端字节序写入值为25的32位整数
dv.setInt32(4, 25);

// 在第9个字节，以小端字节序写入值为2.5的32位浮点数
dv.setFloat32(8, 2.5, true);
```

如果不确定正在使用的计算机的字节序，可以采用下面的判断方式。

```
var littleEndian = (function() {
  var buffer = new ArrayBuffer(2);
  new DataView(buffer).setInt16(0, 256, true);
  return new Int16Array(buffer)[0] === 256;
})();
```

如果返回 `true`，就是小端字节序；如果返回 `false`，就是大端字节序。

二进制数组的应用

大量的Web API用到了 `ArrayBuffer` 对象和它的视图对象。

AJAX

传统上，服务器通过AJAX操作只能返回文本数据，即 `responseType` 属性默认为 `text`。`XMLHttpRequest` 第二版 `XHR2` 允许服务器返回二进制数据，这时分成两种情况。如果明确知道返回的二进制数据类型，可以把返回类型（`responseType`）设为 `arraybuffer`；如果不知道，就设为 `blob`。

```
var xhr = new XMLHttpRequest();
xhr.open('GET', someUrl);
xhr.responseType = 'arraybuffer';

xhr.onload = function () {
  let arrayBuffer = xhr.response;
  // ...
};

xhr.send();
```

如果知道传回来的是32位整数，可以像下面这样处理。

```
xhr.onreadystatechange = function () {
  if (req.readyState === 4) {
    var arrayResponse = xhr.response;
    var dataView = new DataView(arrayResponse);
    var ints = new Uint32Array(dataView.byteLength / 4);

    xhrDiv.style.backgroundColor = "#00FF00";
    xhrDiv.innerText = "Array is " + ints.length + " uints long";
  }
}
```

Canvas

网页 `Canvas` 元素输出的二进制像素数据，就是 `TypedArray` 数组。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
var uint8ClampedArray = imageData.data;
```

需要注意的是，上面代码的 `uint8ClampedArray` 虽然是一个 `TypedArray` 数组，但是它的视图类型是一种针对 `Canvas` 元素的专有类型 `Uint8ClampedArray`。这个视图类型的特点，就是专门针对颜色，把每个字节解读为无符号的8位整数，即只能取值0~255，而且发生运算的时候自动过滤高位溢出。这为图像处理带来了巨大的方便。

举例来说，如果把像素的颜色值设为 `Uint8Array` 类型，那么乘以一个 `gamma` 值的时候，就必须这样计算：

```
u8[i] = Math.min(255, Math.max(0, u8[i] * gamma));
```

因为 `Uint8Array` 类型对于大于255的运算结果（比如 `0xFF+1`），会自动变为 `0x00`，所以图像处理必须要像上面这样算。这样做很麻烦，而且影响性能。如果将颜色值设为 `Uint8ClampedArray` 类型，计算就简化许多。

```
pixels[i] *= gamma;
```

`Uint8ClampedArray` 类型确保将小于0的值设为0，将大于255的值设为255。注意，IE 10不支持该类型。

WebSocket

`WebSocket` 可以通过 `ArrayBuffer`，发送或接收二进制数据。

```
var socket = new WebSocket('ws://127.0.0.1:8081');
socket.binaryType = 'arraybuffer';

// Wait until socket is open
socket.addEventListener('open', function (event) {
  // Send binary data
  var typedArray = new Uint8Array(4);
  socket.send(typedArray.buffer);
});

// Receive binary data
socket.addEventListener('message', function (event) {
  var arrayBuffer = event.data;
  // ...
});
```

Fetch API

Fetch API 取回的数据，就是 `ArrayBuffer` 对象。

```
fetch(url)
  .then(function(request){
    return request.arrayBuffer()
  })
  .then(function(arrayBuffer){
    // ...
  });
```

File API

如果知道一个文件的二进制数据类型，也可以将这个文件读取为 `ArrayBuffer` 对象。

```
var fileInput = document.getElementById('fileInput');
var file = fileInput.files[0];
var reader = new FileReader();
reader.readAsArrayBuffer(file);
reader.onload = function () {
  var arrayBuffer = reader.result;
  // ...
};
```

下面以处理bmp文件为例。假定 `file` 变量是一个指向bmp文件的文件对象，首先读取文件。

```
var reader = new FileReader();
reader.addEventListener("load", processimage, false);
reader.readAsArrayBuffer(file);
```

然后，定义处理图像的回调函数：先在二进制数据之上建立一个 `DataView` 视图，再建立一个 `bitmap` 对象，用于存放处理后的数据，最后将图像展示在 `Canvas` 元素之中。

```
function processimage(e) {
  var buffer = e.target.result;
  var datav = new DataView(buffer);
  var bitmap = {};
  // 具体的处理步骤
}
```

具体处理图像数据时，先处理bmp的文件头。具体每个文件头的格式和定义，请参阅有关资料。

```
bitmap.fileheader = {};
bitmap.fileheader.bfType = datav.getUint16(0, true);
bitmap.fileheader.bfSize = datav.getUint32(2, true);
bitmap.fileheader.bfReserved1 = datav.getUint16(6, true);
bitmap.fileheader.bfReserved2 = datav.getUint16(8, true);
bitmap.fileheader.bfOffBits = datav.getUint32(10, true);
```

接着处理图像元信息部分。

```
bitmap.infoheader = {};
bitmap.infoheader.biSize = datav.getUint32(14, true);
bitmap.infoheader.biWidth = datav.getUint32(18, true);
bitmap.infoheader.biHeight = datav.getUint32(22, true);
bitmap.infoheader.biPlanes = datav.getUint16(26, true);
bitmap.infoheader.biBitCount = datav.getUint16(28, true);
bitmap.infoheader.biCompression = datav.getUint32(30, true);
bitmap.infoheader.biSizeImage = datav.getUint32(34, true);
bitmap.infoheader.biXPelsPerMeter = datav.getUint32(38, true);
bitmap.infoheader.biYPelsPerMeter = datav.getUint32(42, true);
bitmap.infoheader.biClrUsed = datav.getUint32(46, true);
bitmap.infoheader.biClrImportant = datav.getUint32(50, true);
```

最后处理图像本身的像素信息。

```
var start = bitmap.fileheader.bfOffBits;
bitmap.pixels = new Uint8Array(buffer, start);
```

至此，图像文件的数据全部处理完成。下一步，可以根据需要，进行图像变形，或者转换格式，或者展示在 `Canvas` 网页元素之中。

SharedArrayBuffer

目前有一种场景，需要多个进程共享数据：浏览器启动多个WebWorker。

```
var w = new Worker('myworker.js');
```

上面代码中，主窗口新建了一个 Worker 进程。该进程与主窗口之间会有一个通信渠道，主窗口通过 `w.postMessage` 向 Worker 进程发消息，同时通过 `message` 事件监听 Worker 进程的回应。

```
w.postMessage('hi');
w.onmessage = function (ev) {
  console.log(ev.data);
}
```

上面代码中，主窗口先发一个消息 `hi`，然后在监听到 Worker 进程的回应后，就将其打印出来。

Worker 进程也是通过监听 `message` 事件，来获取主窗口发来的消息，并作出反应。

```
onmessage = function (ev) {
  console.log(ev.data);
  postMessage('ho');
}
```

主窗口与 Worker 进程之间，可以传送各种数据，不仅仅是字符串，还可以传送二进制数据。很容易想到，如果有大量数据要传送，留出一块内存区域，主窗口与 Worker 进程共享，两方都可以读写，那么就会大大提高效率。

现在，有一个 `SharedArrayBuffer` 提案，允许多个 Worker 进程与主窗口共享内存。这个对象的 API 与 `ArrayBuffer` 一模一样，唯一的区别是后者无法共享。

```
// 新建 1KB 共享内存
var sab = new SharedArrayBuffer(1024);

// 主窗口发送数据
w.postMessage(sab, [sab])
```

上面代码中，`postMessage` 方法的第一个参数是 `SharedArrayBuffer` 对象，第二个参数是要写入共享内存的数据。

Worker 进程从事件的 `data` 属性上面取到数据。

```
var sab;
onmessage = function (ev) {
  sab = ev.data; // 1KB 的共享内存，就是主窗口共享出来的那块内存
};
```

共享内存也可以在 Worker 进程创建，发给主窗口。

`SharedArrayBuffer` 与 `SharedArray` 一样，本身是无法读写，必须在上面建立视图，然后通过视图读写。

```
// 分配 10 万个 32 位整数占据的内存空间
var sab = new SharedArrayBuffer(Int32Array.BYTES_PER_ELEMENT * 100000);

// 建立 32 位整数视图
var ia = new Int32Array(sab); // ia.length == 100000

// 新建一个质数生成器
var primes = new PrimeGenerator();

// 将 10 万个质数，写入这段内存空间
for (let i=0 ; i < ia.length ; i++)
  ia[i] = primes.next();

// 向 Worker 进程发送这段共享内存
```

```
w.postMessage(ia, [ia.buffer]);
```

Worker 收到数据后的处理如下。

```
var ia;
onmessage = function (ev) {
  ia = ev.data;
  console.log(ia.length); // 100000
  console.log(ia[37]); // 输出 163, 因为这是第138个质数
};
```

SIMD

概述

SIMD（发音 /sim-dee/）是“Single Instruction/Multiple Data”的缩写，意为“单指令，多数据”。它是JavaScript操作CPU对应指令的接口，你可以看做这是一种不同的运算执行模式。与它相对的是SISD（“Single Instruction/Single Data”），即“单指令，单数据”。

SIMD的含义是使用一个指令，完成多个数据的运算；SISD的含义是使用一个指令，完成单个数据的运算，这是JavaScript的默认运算模式。显而易见，SIMD的执行效率要高于SISD，所以被广泛用于3D图形运算、物理模拟等运算量超大的项目之中。

为了理解SIMD，请看下面的例子。

```
var a = [1, 2, 3, 4];
var b = [5, 6, 7, 8];
var c = [];

c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
c // Array[6, 8, 10, 12]
```

上面代码中，数组 `a` 和 `b` 的对应成员相加，结果放入数组 `c`。它的运算模式是依次处理每个数组成员，一共有四个数组成员，所以需要运算4次。

如果采用SIMD模式，只要运算一次就够了。

```
var a = SIMD.Float32x4(1, 2, 3, 4);
var b = SIMD.Float32x4(5, 6, 7, 8);
var c = SIMD.Float32x4.add(a, b); // Float32x4[6, 8, 10, 12]
```

上面代码之中，数组 `a` 和 `b` 的四个成员的各自相加，只用一条指令就完成了。因此，速度比上一种写法提高了4倍。

一次SIMD运算，可以处理多个数据，这些数据被称为“通道”（lane）。上面代码中，一次运算了四个数据，因此就是四个通道。

SIMD通常用于矢量运算。

$$\begin{aligned} v + w &= \langle v_1, \dots, v_n \rangle + \langle w_1, \dots, w_n \rangle \\ &= \langle v_1+w_1, \dots, v_n+w_n \rangle \end{aligned}$$

上面代码中，`v` 和 `w` 是两个多元矢量。它们的加运算，在SIMD下是一个指令、而不是n个指令完成的，这就大大提高了效率。这对于3D动画、图像处理、信号处理、数值处理、加密等运算是非常重要的。比如，Canvas的 `getImageData()` 会将图像文件读成一个二进制数组，SIMD就很适合对于这种数组的处理。

总的来说，SIMD是数据并行处理（parallelism）的一种手段，可以加速一些运算密集型操作的速度。

数据类型

SIMD提供12种数据类型，总长度都是128个二进制位。

- Float32x4: 四个32位浮点数
- Float64x2: 两个64位浮点数
- Int32x4: 四个32位整数
- Int16x8: 八个16位整数
- Int8x16: 十六个8位整数
- Uint32x4: 四个无符号的32位整数
- Uint16x8: 八个无符号的16位整数
- Uint8x16: 十六个无符号的8位整数
- Bool32x4: 四个32位布尔值
- Bool16x8: 八个16位布尔值
- Bool8x16: 十六个8位布尔值
- Bool64x2: 两个64位布尔值

每种数据类型被 `x` 符号分隔成两部分，后面的部分表示通道数，前面的部分表示每个通道的宽度和类型。比如，`Float32x4` 就表示这个值有4个通道，每个通道是一个32位浮点数。

每个通道之中，可以放置四种数据。

- 浮点数（float，比如1.0）
- 带符号的整数（Int，比如-1）
- 无符号的整数（Uint，比如1）
- 布尔值（Bool，包含 `true` 和 `false` 两种值）

每种SIMD的数据类型都是一个函数方法，可以传入参数，生成对应的值。

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
```

上面代码中，变量 `a` 就是一个128位、包含四个32位浮点数（即四个通道）的值。

注意，这些数据类型方法都不是构造函数，前面不能加 `new`，否则会报错。

```
var v = new SIMD.Float32x4(0, 1, 2, 3);
// TypeError: SIMD.Float32x4 is not a constructor
```

静态方法：数学运算

每种数据类型都有一系列运算符，支持基本的数学运算。

SIMD.%type%.abs(), SIMD.%type%.neg()

`abs` 方法接受一个SIMD值作为参数，将它的每个通道都转成绝对值，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(-1, -2, 0, NaN);
SIMD.Float32x4.abs(a)
// Float32x4[1, 2, 0, NaN]
```

`neg` 方法接受一个SIMD值作为参数，将它的每个通道都转成负值，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(-1, -2, 3, 0);
SIMD.Float32x4.neg(a)
// Float32x4[1, 2, -3, -0]

var b = SIMD.Float64x2(NaN, Infinity);
SIMD.Float64x2.neg(b)
// Float64x2[NaN, -Infinity]
```

SIMD.%type%.add(), SIMD.%type%.addSaturate()

`add` 方法接受两个SIMD值作为参数，将它们的每个通道相加，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var b = SIMD.Float32x4(5.0, 10.0, 15.0, 20.0);
var c = SIMD.Float32x4.add(a, b);
```

上面代码中，经过加法运算，新的SIMD值为 `(6.0, 12.0, 18.0, 24.0)`。

`addSaturate` 方法跟 `add` 方法的作用相同，都是两个通道相加，但是溢出的处理不一致。对于 `add` 方法，如果两个值相加发生溢出，溢出的二进制位会被丢弃；`addSaturate` 方法则是返回该数据类型的最大值。

```
var a = SIMD.Uint16x8(65533, 65534, 65535, 65535, 1, 1, 1, 1);
var b = SIMD.Uint16x8(1, 1, 1, 5000, 1, 1, 1, 1);
SIMD.Uint16x8.addSaturate(a, b);
// Uint16x8[65534, 65535, 65535, 65535, 2, 2, 2, 2]

var c = SIMD.Int16x8(32765, 32766, 32767, 32767, 1, 1, 1, 1);
var d = SIMD.Int16x8(1, 1, 1, 5000, 1, 1, 1, 1);
SIMD.Int16x8.addSaturate(c, d);
// Int16x8[32766, 32767, 32767, 32767, 2, 2, 2, 2]
```

上面代码中，`Uint16` 的最大值是65535，`Int16` 的最大值是32767。一旦发生溢出，就返回这两个值。

注意，`Uint32x4` 和 `Int32x4` 这两种数据类型没有 `addSaturate` 方法。

SIMD.%type%.sub(), SIMD.%type%.subSaturate()

`sub` 方法接受两个SIMD值作为参数，将它们的每个通道相减，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(-1, -2, 3, 4);
var b = SIMD.Float32x4(3, 3, 3, 3);
SIMD.Float32x4.sub(a, b)
// Float32x4[-4, -5, 0, 1]
```

`subSaturate` 方法跟 `sub` 方法的作用相同，都是两个通道相减，但是溢出的处理不一致。对于 `sub` 方法，如果两个值相减发生溢出，溢出的二进制位会被丢弃；`subSaturate` 方法则是返回该数据类型的最小值。

```
var a = SIMD.Uint16x8(5, 1, 1, 1, 1, 1, 1, 1);
var b = SIMD.Uint16x8(10, 1, 1, 1, 1, 1, 1, 1);
SIMD.Uint16x8.subSaturate(a, b)
// Uint16x8[0, 0, 0, 0, 0, 0, 0, 0]

var c = SIMD.Int16x8(-100, 0, 0, 0, 0, 0, 0, 0);
var d = SIMD.Int16x8(32767, 0, 0, 0, 0, 0, 0, 0);
SIMD.Int16x8.subSaturate(c, d)
// Int16x8[-32768, 0, 0, 0, 0, 0, 0, 0]
```

上面代码中，`Uint16` 的最小值是 `0`，`subSaturate` 的最小值是 `-32678`。一旦运算发生溢出，就返回最小值。

SIMD.%type%.mul(), SIMD.%type%.div(), SIMD.%type%.sqrt()

mul 方法接受两个SIMD值作为参数，将它们的每个通道相乘，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(-1, -2, 3, 4);
var b = SIMD.Float32x4(3, 3, 3, 3);
SIMD.Float32x4.mul(a, b)
// Float32x4[-3, -6, 9, 12]
```

div 方法接受两个SIMD值作为参数，将它们的每个通道相除，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(2, 2, 2, 2);
var b = SIMD.Float32x4(4, 4, 4, 4);
SIMD.Float32x4.div(a, b)
// Float32x4[0.5, 0.5, 0.5, 0.5]
```

sqrt 方法接受一个SIMD值作为参数，求出每个通道的平方根，作为一个新的SIMD值返回。

```
var b = SIMD.Float64x2(4, 8);
SIMD.Float64x2.sqrt(b)
// Float64x2[2, 2.8284271247461903]
```

SIMD.%FloatType%.reciprocalApproximation(), SIMD.%type%.reciprocalSqrtApproximation()

reciprocalApproximation 方法接受一个SIMD值作为参数，求出每个通道的倒数 ($1 / x$)，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(1, 2, 3, 4);
SIMD.Float32x4.reciprocalApproximation(a);
// Float32x4[1, 0.5, 0.3333333432674408, 0.25]
```

reciprocalSqrtApproximation 方法接受一个SIMD值作为参数，求出每个通道的平方根的倒数 ($1 / (x^{0.5})$)，作为一个新的SIMD值返回。

```
var a = SIMD.Float32x4(1, 2, 3, 4);
SIMD.Float32x4.reciprocalSqrtApproximation(a)
// Float32x4[1, 0.7071067690849304, 0.5773502588272095, 0.5]
```

注意，只有浮点数的数据类型才有这两个方法。

SIMD.%IntegerType%.shiftLeftByScalar()

shiftLeftByScalar 方法接受一个SIMD值作为参数，然后将每个通道的值左移指定的位数，作为一个新的SIMD值返回。

```
var a = SIMD.Int32x4(1, 2, 4, 8);
SIMD.Int32x4.shiftLeftByScalar(a, 1);
// Int32x4[2, 4, 8, 16]
```

如果左移后，新的值超出了当前数据类型的位数，溢出的部分会被丢弃。

```
var ix4 = SIMD.Int32x4(1, 2, 3, 4);
var jx4 = SIMD.Int32x4.shiftLeftByScalar(ix4, 32);
// Int32x4[0, 0, 0, 0]
```

注意，只有整数的数据类型才有这个方法。

SIMD.%IntegerType%.shiftRightByScalar()

`shiftRightByScalar` 方法接受一个SIMD值作为参数，然后将每个通道的值右移指定的位数，返回一个新的SIMD值。

```
var a = SIMD.Int32x4(1, 2, 4, -8);
SIMD.Int32x4.shiftRightByScalar(a, 1);
// Int32x4[0, 1, 2, -4]
```

如果原来通道的值是带符号的值，则符号位保持不变，不受右移影响。如果是不带符号位的值，则右移后头部会补 0。

```
var a = SIMD.UInt32x4(1, 2, 4, -8);
SIMD.UInt32x4.shiftRightByScalar(a, 1);
// UInt32x4[0, 1, 2, 2147483644]
```

注意，只有整数的数据类型才有这个方法。

静态方法：通道处理

SIMD.%type%.check()

`check` 方法用于检查一个值是否为当前类型的SIMD值。如果是的，就返回这个值，否则就报错。

```
var a = SIMD.Float32x4(1, 2, 3, 9);

SIMD.Float32x4.check(a);
// Float32x4[1, 2, 3, 9]

SIMD.Float32x4.check([1,2,3,4]) // 报错
SIMD.Int32x4.check(a) // 报错
SIMD.Int32x4.check('hello world') // 报错
```

`SIMD.%type%.extractLane()`, `SIMD.%type%.replaceLane()`

`extractLane` 方法用于返回给定通道的值。它接受两个参数，分别是SIMD值和通道编号。

```
var t = SIMD.Float32x4(1, 2, 3, 4);
SIMD.Float32x4.extractLane(t, 2) // 3
```

`replaceLane` 方法用于替换指定通道的值，并返回一个新的SIMD值。它接受三个参数，分别是原来的SIMD值、通道编号和新的通道值。

```
var t = SIMD.Float32x4(1, 2, 3, 4);
SIMD.Float32x4.replaceLane(t, 2, 42)
// Float32x4[1, 2, 42, 4]
```

SIMD.%type%.load()

`load` 方法用于从二进制数组读入数据，生成一个新的SIMD值。

```
var a = new Int32Array([1,2,3,4,5,6,7,8]);  
SIMD.Int32x4.load(a, 0);
```

```
// Int32x4[1, 2, 3, 4]
var b = new Int32Array([1,2,3,4,5,6,7,8]);
SIMD.Int32x4.load(a, 2);
// Int32x4[3, 4, 5, 6]
```

`load` 方法接受两个参数：一个二进制数组和开始读取的位置（从0开始）。如果位置不合法（比如 `-1` 或者超出二进制数组的大小），就会抛出一个错误。

这个方法还有三个变种 `load1()`、`load2()`、`load3()`，表示从指定位置开始，只加载一个通道、二个通道、三个通道的值。

```
// 格式
SIMD.Int32x4.load(tarray, index)
SIMD.Int32x4.load1(tarray, index)
SIMD.Int32x4.load2(tarray, index)
SIMD.Int32x4.load3(tarray, index)

// 实例
var a = new Int32Array([1,2,3,4,5,6,7,8]);
SIMD.Int32x4.load1(a, 0);
// Int32x4[1, 0, 0, 0]
SIMD.Int32x4.load2(a, 0);
// Int32x4[1, 2, 0, 0]
SIMD.Int32x4.load3(a, 0);
// Int32x4[1, 2, 3, 0]
```

SIMD.%type%.store()

`store` 方法用于将一个SIMD值，写入一个二进制数组。它接受三个参数，分别是二进制数组、开始写入的数组位置、SIMD值。它返回写入值以后的二进制数组。

```
var t1 = new Int32Array(8);
var v1 = SIMD.Int32x4(1, 2, 3, 4);
SIMD.Int32x4.store(t1, 0, v1)
// Int32Array[1, 2, 3, 4, 0, 0, 0, 0]

var t2 = new Int32Array(8);
var v2 = SIMD.Int32x4(1, 2, 3, 4);
SIMD.Int32x4.store(t2, 2, v2)
// Int32Array[0, 0, 1, 2, 3, 4, 0, 0]
```

上面代码中，`t1` 是一个二进制数组，`v1` 是一个SIMD值，只有四个通道。所以写入 `t1` 以后，只有前四个位置有值，后四个位置都是0。而 `t2` 是从2号位置开始写入，所以前两个位置和后两个位置都是0。

这个方法还有三个变种 `store1()`、`store2()` 和 `store3()`，表示只写入一个通道、二个通道和三个通道的值。

```
var tarray = new Int32Array(8);
var value = SIMD.Int32x4(1, 2, 3, 4);
SIMD.Int32x4.store1(tarray, 0, value);
// Int32Array[1, 0, 0, 0, 0, 0, 0, 0]
```

SIMD.%type%.splat()

`splat` 方法返回一个新的SIMD值，该值的所有通道都会设成同一个预先给定的值。

```
SIMD.Float32x4.splat(3);
// Float32x4[3, 3, 3, 3]
SIMD.Float64x2.splat(3);
// Float64x2[3, 3]
```

如果省略参数，所有整数型的SIMD值都会设定 `0`，浮点型的SIMD值都会设成 `NaN`。

SIMD.%type%.swizzle()

`swizzle` 方法返回一个新的SIMD值，重新排列原有的SIMD值的通道顺序。

```
var t = SIMD.Float32x4(1, 2, 3, 4);
SIMD.Float32x4.swizzle(t, 1, 2, 0, 3);
// Float32x4[2,3,1,4]
```

上面代码中，`swizzle` 方法的第一个参数是原有的SIMD值，后面的参数对应将要返回的SIMD值的四个通道。它的意思是新的SIMD的四个通道，依次是原来SIMD值的1号通道、2号通道、0号通道、3号通道。由于SIMD值最多可以有16个通道，所以 `swizzle` 方法除了第一个参数以外，最多还可以接受16个参数。

下面是另一个例子。

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
// Float32x4[1.0, 2.0, 3.0, 4.0]

var b = SIMD.Float32x4.swizzle(a, 0, 0, 1, 1);
// Float32x4[1.0, 1.0, 2.0, 2.0]

var c = SIMD.Float32x4.swizzle(a, 3, 3, 3, 3);
// Float32x4[4.0, 4.0, 4.0, 4.0]

var d = SIMD.Float32x4.swizzle(a, 3, 2, 1, 0);
// Float32x4[4.0, 3.0, 2.0, 1.0]
```

SIMD.%type%.shuffle()

`shuffle` 方法从两个SIMD值之中取出指定通道，返回一个新的SIMD值。

```
var a = SIMD.Float32x4(1, 2, 3, 4);
var b = SIMD.Float32x4(5, 6, 7, 8);

SIMD.Float32x4.shuffle(a, b, 1, 5, 7, 2);
// Float32x4[2, 6, 8, 3]
```

上面代码中，`a` 和 `b` 一共有8个通道，依次编号为0到7。`shuffle` 根据编号，取出相应的通道，返回一个新的SIMD值。

静态方法：比较运算

SIMD.%type%.equal(), SIMD.%type%.notEqual()

`equal` 方法用来比较两个SIMD值 `a` 和 `b` 的每一个通道，根据两者是否精确相等（`a === b`），得到一个布尔值。最后，所有通道的比较结果，组成一个新的SIMD值，作为掩码返回。`notEqual` 方法则是比较两个通道是否不相等（`a !== b`）。

```
var a = SIMD.Float32x4(1, 2, 3, 9);
var b = SIMD.Float32x4(1, 4, 7, 9);

SIMD.Float32x4.equal(a,b)
// Bool32x4[true, false, false, true]

SIMD.Float32x4.notEqual(a,b);
// Bool32x4[false, true, true, false]
```

SIMD.%type%.greaterThan(), SIMD.%type%.greaterThanOrEqual()

`greatThan` 方法用来比较两个SIMD值 `a` 和 `b` 的每一个通道，如果在该通道中，`a` 较大就得到 `true`，否则得到 `false`。最后，所有通道的比较结果，组成一个新的SIMD值，作为掩码返回。`greaterThanOrEqual` 则是比较 `a` 是否大于等于 `b`。

```
var a = SIMD.Float32x4(1, 6, 3, 11);
var b = SIMD.Float32x4(1, 4, 7, 9);

SIMD.Float32x4.greaterThan(a, b)
// Bool32x4[false, true, false, true]

SIMD.Float32x4.greaterThanOrEqual(a, b)
// Bool32x4[true, true, false, true]
```

SIMD.%type%.lessThan(), SIMD.%type%.lessThanOrEqual()

`lessThan` 方法用来比较两个SIMD值 `a` 和 `b` 的每一个通道，如果在该通道中，`a` 较小就得到 `true`，否则得到 `false`。最后，所有通道的比较结果，会组成一个新的SIMD值，作为掩码返回。`lessThanOrEqual` 方法则是比较 `a` 是否等于 `b`。

```
var a = SIMD.Float32x4(1, 2, 3, 11);
var b = SIMD.Float32x4(1, 4, 7, 9);

SIMD.Float32x4.lessThan(a, b)
// Bool32x4[false, true, true, false]

SIMD.Float32x4.lessThanOrEqual(a, b)
// Bool32x4[true, true, true, false]
```

SIMD.%type%.select()

`select` 方法通过掩码生成一个新的SIMD值。它接受三个参数，分别是掩码和两个SIMD值。

```
var a = SIMD.Float32x4(1, 2, 3, 4);
var b = SIMD.Float32x4(5, 6, 7, 8);

var mask = SIMD.Bool32x4(true, false, false, true);

SIMD.Float32x4.select(mask, a, b);
// Float32x4[1, 6, 7, 4]
```

上面代码中，`select` 方法接受掩码和两个SIMD值作为参数。当某个通道对应的掩码为 `true` 时，会选择第一个SIMD值的对应通道，否则选择第二个SIMD值的对应通道。

这个方法通常与比较运算符结合使用。

```
var a = SIMD.Float32x4(0, 12, 3, 4);
var b = SIMD.Float32x4(0, 6, 7, 50);

var mask = SIMD.Float32x4.lessThan(a,b);
// Bool32x4[false, false, true, true]

var result = SIMD.Float32x4.select(mask, a, b);
// Float32x4[0, 6, 3, 4]
```

上面代码中，先通过 `lessThan` 方法生成一个掩码，然后通过 `select` 方法生成一个由每个通道的较小值组成的新的

SIMD值。

SIMD.%BooleanType%.allTrue(), SIMD.%BooleanType%.anyTrue()

`allTrue` 方法接受一个SIMD值作为参数，然后返回一个布尔值，表示该SIMD值的所有通道是否都为 `true`。

```
var a = SIMD.Bool32x4(true, true, true, true);
var b = SIMD.Bool32x4(true, false, true, true);
```

```
SIMD.Bool32x4.allTrue(a); // true
SIMD.Bool32x4.allTrue(b); // false
```

`anyTrue` 方法则是只要有一个通道为 `true`，就返回 `true`，否则返回 `false`。

```
var a = SIMD.Bool32x4(false, false, false, false);
var b = SIMD.Bool32x4(false, false, true, false);
```

```
SIMD.Bool32x4.anyTrue(a); // false
SIMD.Bool32x4.anyTrue(b); // true
```

注意，只有四种布尔值数据类型（`Bool32x4`、`Bool16x8`、`Bool8x16`、`Bool64x2`）才有这两个方法。

这两个方法通常与比较运算符结合使用。

```
var ax4    = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var bx4    = SIMD.Float32x4(0.0, 6.0, 7.0, 8.0);
var ix4    = SIMD.Float32x4.lessThan(ax4, bx4);
var b1     = SIMD.Int32x4.allTrue(ix4); // false
var b2     = SIMD.Int32x4.anyTrue(ix4); // true
```

SIMD.%type%.min(), SIMD.%type%.minNum()

`min` 方法接受两个SIMD值作为参数，将两者的对应通道的较小值，组成一个新的SIMD值返回。

```
var a = SIMD.Float32x4(-1, -2, 3, 5.2);
var b = SIMD.Float32x4(0, -4, 6, 5.5);
SIMD.Float32x4.min(a, b);
// Float32x4[-1, -4, 3, 5.2]
```

如果有一个通道的值是 `NaN`，则会优先返回 `NaN`。

```
var c = SIMD.Float64x2(NaN, Infinity)
var d = SIMD.Float64x2(1337, 42);
SIMD.Float64x2.min(c, d);
// Float64x2[NaN, 42]
```

`minNum` 方法与 `min` 的作用一模一样，唯一的区别是如果有一个通道的值是 `NaN`，则会优先返回另一个通道的值。

```
var ax4 = SIMD.Float32x4(1.0, 2.0, NaN, NaN);
var bx4 = SIMD.Float32x4(2.0, 1.0, 3.0, NaN);
var cx4 = SIMD.Float32x4.min(ax4, bx4);
// Float32x4[1.0, 1.0, NaN, NaN]
var dx4 = SIMD.Float32x4.minNum(ax4, bx4);
// Float32x4[1.0, 1.0, 3.0, NaN]
```

SIMD.%type%.max(), SIMD.%type%.maxNum()

`max` 方法接受两个SIMD值作为参数，将两者的对应通道的较大值，组成一个新的SIMD值返回。

```
var a = SIMD.Float32x4(-1, -2, 3, 5.2);
var b = SIMD.Float32x4(0, -4, 6, 5.5);
SIMD.Float32x4.max(a, b);
// Float32x4[0, -2, 6, 5.5]
```

如果有一个通道的值是 `Nan`，则会优先返回 `Nan`。

```
var c = SIMD.Float64x2(NaN, Infinity)
var d = SIMD.Float64x2(1337, 42);
SIMD.Float64x2.max(c, d)
// Float64x2[Nan, Infinity]
```

`maxNum` 方法与 `max` 的作用一模一样，唯一的区别是如果有一个通道的值是 `Nan`，则会优先返回另一个通道的值。

```
var c = SIMD.Float64x2(NaN, Infinity)
var d = SIMD.Float64x2(1337, 42);
SIMD.Float64x2.maxNum(c, d)
// Float64x2[1337, Infinity]
```

静态方法：位运算

`SIMD.%type%.and()`, `SIMD.%type%.or()`, `SIMD.%type%.xor()`,
`SIMD.%type%.not()`

`and` 方法接受两个SIMD值作为参数，返回两者对应的通道进行二进制 `AND` 运算（`&`）后得到的新的SIMD值。

```
var a = SIMD.Int32x4(1, 2, 4, 8);
var b = SIMD.Int32x4(5, 5, 5, 5);
SIMD.Int32x4.and(a, b)
// Int32x4[1, 0, 4, 0]
```

上面代码中，以通道 `0` 为例，`1` 的二进制形式是 `0001`，`5` 的二进制形式是 `01001`，所以进行 `AND` 运算以后，得到 `0001`。

`or` 方法接受两个SIMD值作为参数，返回两者对应的通道进行二进制 `OR` 运算（`|`）后得到的新的SIMD值。

```
var a = SIMD.Int32x4(1, 2, 4, 8);
var b = SIMD.Int32x4(5, 5, 5, 5);
SIMD.Int32x4.or(a, b)
// Int32x4[5, 7, 5, 13]
```

`xor` 方法接受两个SIMD值作为参数，返回两者对应的通道进行二进制“异或”运算（`^`）后得到的新的SIMD值。

```
var a = SIMD.Int32x4(1, 2, 4, 8);
var b = SIMD.Int32x4(5, 5, 5, 5);
SIMD.Int32x4.xor(a, b)
// Int32x4[4, 7, 1, 13]
```

`not` 方法接受一个SIMD值作为参数，返回每个通道进行二进制“否”运算（`~`）后得到的新的SIMD值。

```
var a = SIMD.Int32x4(1, 2, 4, 8);
SIMD.Int32x4.not(a)
// Int32x4[-2, -3, -5, -9]
```

上面代码中，`1` 的否运算之所以得到 `-2`，是因为在计算机内部，负数采用“2的补码”这种形式进行表示。也就是说，整数 `n` 的负数形式 `-n`，是对每一个二进制位取反以后，再加上1。因此，直接取反就相当于负数形式再减去1，比如 `1` 的负数形式是 `-1`，再减去1，就得到了 `-2`。

静态方法：数据类型转换

SIMD提供以下方法，用来将一种数据类型转为另一种数据类型。

- SIMD.%type%.fromFloat32x4()
- SIMD.%type%.fromFloat32x4Bits()
- SIMD.%type%.fromFloat64x2Bits()
- SIMD.%type%.fromInt32x4()
- SIMD.%type%.fromInt32x4Bits()
- SIMD.%type%.fromInt16x8Bits()
- SIMD.%type%.fromInt8x16Bits()
- SIMD.%type%.fromUint32x4()
- SIMD.%type%.fromUint32x4Bits()
- SIMD.%type%.fromUint16x8Bits()
- SIMD.%type%.fromUint8x16Bits()

带有 Bits 后缀的方法，会原封不动地将二进制位拷贝到新的数据类型；不带后缀的方法，则会进行数据类型转换。

```
var t = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
SIMD.Int32x4.fromFloat32x4(t);
// Int32x4[1, 2, 3, 4]

SIMD.Int32x4.fromFloat32x4Bits(t);
// Int32x4[1065353216, 1073741824, 1077936128, 1082130432]
```

上面代码中，`fromFloat32x4` 是将浮点数转为整数，然后存入新的数据类型；`fromFloat32x4Bits` 则是将二进制位原封不动地拷贝进入新的数据类型，然后进行解读。

Bits 后缀的方法，还可以用于通道数目不对等的拷贝。

```
var t = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
SIMD.Int16x8.fromFloat32x4Bits(t);
// Int16x8[0, 16256, 0, 16384, 0, 16448, 0, 16512]
```

上面代码中，原始SIMD值 `t` 是4通道的，而目标值是8通道。

如果数据转换时，原通道的数据大小，超过了目标通道的最大宽度，就会报错。

实例方法

SIMD.%type%.prototype.toString()

`toString` 方法返回一个SIMD值的字符串形式。

```
var a = SIMD.Float32x4(11, 22, 33, 44);
a.toString() // "SIMD.Float32x4(11, 22, 33, 44)"
```

实例：求平均值

正常模式下，计算 `n` 个值的平均值，需要运算 `n` 次。

```
function average(list) {
  var n = list.length;
  var sum = 0.0;
  for (var i = 0; i < n; i++) {
```

```
    sum += list[i];
}
return sum / n;
}
```

使用SIMD，可以将计算次数减少到 `n` 次的四分之一。

```
function average(list) {
  var n = list.length;
  var sum = SIMD.Float32x4.splat(0.0);
  for (var i = 0; i < n; i += 4) {
    sum = SIMD.Float32x4.add(
      sum,
      SIMD.Float32x4.load(list, i)
    );
  }
  var total = SIMD.Float32x4.extractLane(sum, 0) +
    SIMD.Float32x4.extractLane(sum, 1) +
    SIMD.Float32x4.extractLane(sum, 2) +
    SIMD.Float32x4.extractLane(sum, 3);
  return total / n;
}
```

上面代码先是每隔四位，将所有的值读入一个SIMD，然后立刻累加。然后，得到累加值四个通道的总和，再除以 `n` 就可以了。