% ECMAScript 6 入门  
% 阮一峰 著

# ECMAScript 6入门

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

[](images/cover.jpg)

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

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

网上为预览版，电子工业出版社不久将出版全书。

### 版权许可

本书采用“保持署名—非商用”创意共享4.0许可证。

只要保持原作者署名和非商用，您可以自由地阅读、分享、修改本书。

详细的法律条文请参见[创意共享](http://creativecommons.org/licenses/by-nc/4.0/)网站。

# ECMAScript 6简介

ECMAScript 6（以下简称ES6）是JavaScript语言的下一代标准，正处在快速开发之中，大部分已经完成了，预计将在2014年底正式发布。Mozilla将在这个标准的基础上，推出JavaScript 2.0。

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

## ECMAScript和JavaScript的关系

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

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

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

## ECMAScript的历史

1998年6月，ECMAScript 2.0版发布。

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

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

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

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

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

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

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

2014年12月，ECMAScript 6预计将发布正式版本。

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

## 部署进度

由于ES6还没有定案，有些语法规则还会变动，目前支持ES6的软件和开发环境还不多。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/](http://kangax.github.io/es5-compat-table/es6/)。

Google公司的V8引擎已经部署了ES6的部分特性。使用node.js 0.11版，就可以体验这些特性。

node.js的0.11版还不是稳定版本，要使用版本管理工具[nvm](https://github.com/creationix/nvm)切换。下载nvm以后，进入项目目录，运行下面的命令。

source nvm.sh  
nvm use 0.11  
node --harmony

启动命令中的--harmony选项可以打开所有已经部署的ES6功能。使用下面的命令，可以查看所有与ES6有关的单个选项。

$ 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

## Traceur编译器

Google公司的[Traceur](https://github.com/google/traceur-compiler)编译器，可以将ES6代码编译为ES5代码。

它有多种使用方式。

**（1）直接插入网页**

Traceur允许将ES6代码直接插入网页。

首先，必须在网页头部加载Traceur库文件。

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

接下来，就可以把ES6代码放入上面这些代码的下方。

<script type="module">  
 class Calc {  
 constructor(){  
 console.log('Calc constructor');  
 }  
 add(a, b){  
 return a + b;  
 }  
 }  
  
 var c = new Calc();  
 console.log(c.add(4,5));  
</script>

正常情况下，上面代码会在控制台打印出9。

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

如果ES6代码是一个外部文件，也可以用script标签插入网页。

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

**（2）在线转换**

Traceur提供一个[在线编译器](http://google.github.io/traceur-compiler/demo/repl.html)，可以在线将ES6代码转为ES5代码。转换后的代码，可以直接作为ES5代码插入网页运行。

上面的例子转为ES5代码运行，就是下面这个样子。

<script src="http://google.github.io/traceur-compiler/bin/traceur.js"  
 type="text/javascript"></script>  
<script src="http://google.github.io/traceur-compiler/src/bootstrap.js"  
 type="text/javascript"></script>   
<script>  
 traceur.options.experimental = true;  
</script>  
<script>  
$traceurRuntime.ModuleStore.getAnonymousModule(function() {  
 "use strict";  
  
 var Calc = function Calc() {  
 console.log('Calc constructor');  
 };  
  
 ($traceurRuntime.createClass)(Calc, {add: function(a, b) {  
 return a + b;  
 }}, {});  
  
 var c = new Calc();  
 console.log(c.add(4, 5));  
 return {};  
});  
</script>

**（3）命令行转换**

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

npm install -g traceur

安装成功后，就可以在命令行下使用traceur了。

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

$ traceur calc.js  
Calc constructor  
9

如果要将ES6脚本转为ES5，要采用下面的写法

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

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

为了防止有些特性编译不成功，最好加上--experimental选项。

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

命令行下转换的文件，就可以放到浏览器中运行。

**（4）Node.js环境的用法**

Traceur的node.js用法如下（假定已安装traceur模块）。

var traceur = require('traceur');  
var fs = require('fs');  
  
// 将ES6脚本转为字符串  
var contents = fs.readFileSync('es6-file.js').toString();  
  
var result = traceur.compile(contents, {  
 filename: 'es6-file.js',  
 sourceMap: true,  
 // 其他设置  
 modules: 'commonjs'  
});  
  
if (result.error)  
 throw result.error;  
  
// result对象的js属性就是转换后的ES5代码   
fs.writeFileSync('out.js', result.js);  
// sourceMap属性对应map文件  
fs.writeFileSync('out.js.map', result.sourceMap);

## ECMAScript 7

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

ES7可能包括的功能有：

（1）**Object.observe**：对象与网页元素的双向绑定，只要其中之一发生变化，就会自动反映在另一者上。

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

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

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

# let和const命令

## let命令

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。

下面的代码如果使用var，最后输出的是9。

var a = [];  
for (var i = 0; i < 10; i++) {  
 var c = i;  
 a[i] = function () {  
 console.log(c);  
 };  
}  
a[6](); // 9

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

let不像var那样，会发生“变量提升”现象。

function do\_something() {  
 console.log(foo); // ReferenceError  
 let foo = 2;  
}

上面代码在声明foo之前，就使用这个变量，结果会抛出一个错误。

注意，let不允许在相同作用域内，重复声明同一个变量。

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

## 块级作用域

let实际上为JavaScript新增了块级作用域。

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

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

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

// IIFE写法  
(function () {  
 var tmp = ...;  
 ...  
}());  
  
// 块级作用域写法  
{  
 let tmp = ...;  
 ...  
}

另外，ES6也规定，函数本身的作用域，在其所在的块级作用域之内。

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

上面代码在ES5中运行，会得到“I am inside!”，但是在ES6中运行，会得到“I am outside!”。

## const命令

const也用来声明变量，但是声明的是常量。一旦声明，常量的值就不能改变。

const PI = 3.1415;  
PI // 3.1415  
  
PI = 3;  
PI // 3.1415  
  
const PI = 3.1;  
PI // 3.1415

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

const的作用域与let命令相同：只在声明所在的块级作用域内有效。

if (condition) {  
 const MAX = 5;  
}  
  
// 常量MAX在此处不可得

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;

# 变量的解构赋值

## 数组的解构赋值

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

以前，为变量赋值，只能直接指定值。

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

ES6允许写成下面这样。

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

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

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

var [foo, [[bar], baz]] = [1, [[2], 3]];  
foo // 1  
bar // 2  
baz // 3  
  
var [,,third] = ["foo", "bar", "baz"];  
third // "baz"  
  
var [head, ...tail] = [1, 2, 3, 4];  
head // 1  
tail // [2, 3, 4]

如果解构不成功，变量的值就等于undefined。

var [foo] = [];  
var [foo] = 1;  
var [foo] = 'Hello';  
var [foo] = false;  
var [foo] = NaN;

以上几种情况都属于解构不成功，foo的值都会等于undefined。但是，如果对undefined或null进行解构，就会报错。

// 报错  
var [foo] = undefined;  
var [foo] = null;

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

解构赋值允许指定默认值。

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

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"

和数组一样，解构也可以用于嵌套结构的对象。

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

对象的解构也可以指定默认值。

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

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};  
// 或者  
({x} = {x:1});

## 用途

变量的解构赋值用途很多。

**（1）交换变量的值**

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

**（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**

function f({x, y, z}) {  
 // ...  
}  
  
f({x:1, y:2, z:3})

这种写法对提取JSON对象中的数据，尤其有用。

**（4）函数参数的默认值**

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

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

**（5）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，配合变量的结构赋值，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) {  
 // ...  
}

**（6）输入模块的指定方法**

加载模块时，往往需要指定输入那些方法。解构赋值使得输入语句非常清晰。

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

# 字符串的扩展

ES6加强了对Unicode的支持，并且扩展了字符串对象。

## codePointAt方法

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编号大于0xFFFF的字符），JavaScript会认为它们是两个字符。

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

上面代码说明，对于4个字节储存的字符，JavaScript不能正确处理。字符串长度会误判为2，而且charAt方法无法读取字符，charCodeAt方法只能分别返回前两个字节和后两个字节的值。

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

var s = "𠮷a";  
  
s.codePointAt(0) // 134071  
s.codePointAt(1) // 97  
   
s.charCodeAt(2) // 97

codePointAt方法的参数，是字符在字符串中的位置（从0开始）。上面代码表明，它会正确返回四字节的UTF-16字符的Unicode编号。对于那些两个字节储存的常规字符，它的返回结果与charCodeAt方法相同。

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

function is32Bit(c) {  
 return c.codePointAt(0) > 0xFFFF;  
}  
  
is32Bit("𠮷") // true  
is32Bit("a") // false

## String.fromCodePoint方法

该方法用于从Unicode编号返回对应的字符串，作用与codePointAt正好相反。

String.fromCodePoint(134071) // "𠮷"

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Unicode编号。

"\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"  
// " 7"

上面代码表示，如果直接在“\u”后面跟上超过0xFFFF的数值（比如\u20BB7），JavaScript会理解成“\u20BB+7”。

ES6对这一点做出了改进，只要将超过0xFFFF的编号放入大括号，就能正确解读该字符。

"\u{20BB7}"  
// " 7"

## 正则表达式的u修饰符

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

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。

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

## contains(), 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。

* **contains()**：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.contains("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("o", 4) // true  
s.endsWith("o", 8) // true  
s.contains("o", 8) // false

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

## repeat()

repeat()返回一个新字符串，表示将原字符串重复n次。

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

## 正则表达式的y修饰符

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

var s = "aaa\_aa\_a";  
var r1 = /a+/g;  
var r2 = /a+/y;  
  
r1.exec(s) // ["aaa"]  
r2.exec(s) // ["aaa"]  
  
r1.exec(s) // ["aa"]  
r2.exec(s) // null

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

如果改一下正则表达式，保证每次都能头部匹配，y修饰符就会返回结果了。

var s = "aaa\_aa\_a";  
var r = /a+\_/y;  
  
r.exec(s) // ["aaa\_"]  
r.exec(s) // ["aa\_"]

上面代码每次匹配，都是从剩余字符串的头部开始。

进一步说，y修饰符号隐含了头部匹配的标志ˆ。

/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ˆ在全局匹配中都有效。

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

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

## 模板字符串

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

// 普通字符串  
`In JavaScript '\n' is a line-feed.`  
  
// 多行字符串  
`In JavaScript this is  
 not legal.`  
  
// 字符串中嵌入变量  
var name = "Bob", time = "today";  
`Hello ${name}, how are you ${time}?`  
  
var x = 1;  
var y = 2;  
console.log(`${ x } + ${ y } = ${ x + y}`)   
// "1 + 2 = 3"

上面代码表示，在模板字符串中嵌入变量，需要将变量名写在${}之中。

# 数值的扩展

## 二进制和八进制表示法

ES6提供了二进制和八进制数值的新的写法，分别用前缀0b和0o表示。

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

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

## Number.isFinite(), Number.isNaN()

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

它们与传统的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。

这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

## Number.isInteger()和安全整数

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

Number.isInteger(25) // true  
Number.isInteger(25.0) // true  
Number.isInteger(25.1) // false

JavaScript能够准确表示的整数范围在-2ˆ53 and 2ˆ53之间。ES6引入了Number.MAX\_SAFE\_INTEGER和Number.MIN\_SAFE\_INTEGER这两个常量，用来表示这个范围的上下限。Number.isSafeInteger()则是用来判断一个整数是否落在这个范围之内。

var inside = Number.MAX\_SAFE\_INTEGER;  
var outside = inside + 1;  
  
Number.isInteger(inside) // true  
Number.isSafeInteger(inside) // true  
  
Number.isInteger(outside) // true  
Number.isSafeInteger(outside) // false

## Math对象的扩展

**（1）Math.trunc()**

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

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

**（2）数学方法**

ES6在Math对象上还提供了许多新的数学方法。

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

# 数组的扩展

## Array.from()

Array.from()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，其中包括ES6新增的Set和Map结构。

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

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

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

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

## 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).length // 1

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

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

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

## 数组实例的find()和findIndex()

数组实例的find()用于找出第一个符合条件的数组元素。它的参数是一个回调函数，所有数组元素依次遍历该回调函数，直到找出第一个返回值为true的元素，然后返回该元素，否则返回undefined。

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

从上面代码可以看到，回调函数接受三个参数，依次为当前的值、当前的位置和原数组。

数组实例的findIndex()的用法与find()非常类似，返回第一个符合条件的数组元素的位置，如果所有元素都不符合条件，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IndexOf()的不足。

[NaN].indexOf(NaN)   
// -1  
  
[NaN].findIndex(y => Object.is(NaN, y))  
// 0

## 数组实例的fill()

fill()使用给定值，填充一个数组。

['a', 'b', 'c'].fill(7)  
// [7, 7, 7]  
  
new Array(3).fill(7)  
// [7, 7, 7]

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

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

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

## 数组实例的entries()，keys()和values()

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

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

## 数组推导

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

var a1 = [1, 2, 3, 4];  
var a2 = [i \* 2 for (i of a1)];  
  
a2 // [2, 4, 6, 8]

上面代码表示，通过for...of结构，数组a2直接在a1的基础上生成。

数组推导可以替代map和filter方法。

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

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

新引入的for...of结构，可以直接跟在表达式的前面或后面，甚至可以在一个数组推导中，使用多个for...of结构。

var a1 = ["x1", "y1"];  
var a2 = ["x2", "y2"];  
var a3 = ["x3", "y3"];  
  
[(console.log(s + w + r)) for (s of a1) for (w of a2) for (r of a3)];  
// x1x2x3  
// x1x2y3  
// x1y2x3  
// x1y2y3  
// y1x2x3  
// y1x2y3  
// y1y2x3  
// y1y2y3

上面代码在一个数组推导之中，使用了三个for...of结构。

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

由于字符串可以视为数组，因此字符串也可以直接用于数组推导。

[c for (c of 'abcde') if (/[aeiou]/.test(c))].join('') // 'ae'  
  
[c+'0' for (c of 'abcde')].join('') // 'a0b0c0d0e0'

上面代码使用了数组推导，对字符串进行处理。

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

## Array.observe()，Array.unobserve()

这两个方法用于监听（取消监听）数组的变化，指定回调函数。

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

# 对象的扩展

## Object.is()

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

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

## Object.assign()

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

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

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

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

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

**（1）proto属性**

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

var obj = {  
 \_\_proto\_\_: someOtherObj,  
 method: function() { ... }  
}

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

**（2）Object.setPrototypeOf()**

Object.setPrototypeOf方法的作用与\_\_proto\_\_相同，用来设置一个对象的prototype对象。

// 格式  
Object.setPrototypeOf(object, prototype)  
  
// 用法  
var o = Object.setPrototypeOf({}, null);

该方法等同于下面的函数。

function (obj, proto) {  
 obj.\_\_proto\_\_ = proto;  
 return obj;  
}

**（3）Object.getPrototypeOf()**

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

Object.getPrototypeOf(obj)

## 增强的对象写法

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 Person = {  
  
 name: '张三',  
  
 //等同于birth: birth  
 birth,  
  
 // 等同于hello: function ()...  
 hello() { console.log('我的名字是', this.name); }  
  
};

这种写法用于函数的返回值，将会非常方便。

function getPoint() {  
 var x = 1;  
 var y = 10;  
  
 return {x, y};  
}  
  
getPoint()  
// {x:1, y:10}

## 属性名表达式

ES6允许定义对象时，用表达式作为对象的属性名。在写法上，要把表达式放在方括号内。

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

上面代码中，对象a的属性名lastWord是一个变量。

下面是一个将字符串的加法表达式作为属性名的例子。

var suffix = " word";  
  
var a = {  
 ["first" + suffix]: "hello",  
 ["last" + suffix]: "world"  
};  
  
a["first word"] // "hello"  
a["last word"] // "world"

## symbols

ES6引入了一种新的原始数据类型symbol。它通过Symbol函数生成。

var mySymbol = Symbol('Test');  
  
mySymbol.name  
// Test  
  
typeof mySymbol  
// "symbol"

上面代码表示，Symbol函数接受一个字符串作为参数，用来指定生成的symbol的名称，可以通过name属性读取。typeof运算符的结果，表明Symbol是一种单独的数据类型。

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。

symbol的最大特点，就是每一个symbol都是不相等的，保证产生一个独一无二的值。

let w1 = Symbol();  
let w2 = Symbol();  
let w3 = Symbol();  
  
function f(w) {  
 switch (w) {  
 case w1:  
 ...  
 case w2:  
 ...  
 case w3:  
 ...  
 }  
}

上面代码中，w1、w2、w3三个变量都等于Symbol()，但是它们的值是不相等的。

由于这种特点，Symbol类型适合作为标识符，用于对象的属性名，保证了属性名之间不会发生冲突。如果一个对象由多个模块构成，这样就不会出现同名的属性。

Symbol类型作为属性名，可以被遍历，Object.getOwnPropertySymbols()和Object.getOwnPropertyKeys()都可以获取该属性。

var a = {};  
var mySymbol = Symbol();  
  
a[mySymbol] = 'Hello!';  
  
//另一种写法  
Object.defineProperty(a, mySymbol, { value: 'Hello!' });

上面代码通过点结构和Object.defineProperty两种方法，为对象增加一个属性。

下面的写法为Map结构添加了一个成员，但是该成员永远无法被引用。

let a = Map();  
a.set(Symbol(), 'Noise');  
a.size // 1

如果要在对象内部使用symbol属性名，必须采用属性名表达式。

let specialMethod = Symbol();  
  
let obj = {  
 [specialMethod]: function (arg) {  
 ...  
 }  
};  
  
obj[specialMethod](123);

## Proxy

所谓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，可以被过滤和改写。

ES6原生提供Proxy构造函数，用来生成proxy实例对象。

var proxy = new Proxy({}, {  
 get: function(target, property) {  
 return 35;  
 }  
});  
  
proxy.time // 35  
proxy.name // 35  
proxy.title // 35

上面代码就是Proxy构造函数使用实例，它接受两个参数，第一个所要代理的目标对象（上例是一个空对象），第二个是拦截函数，它有一个get方法，用来拦截对目标对象的访问请求。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。

下面是另一个拦截函数的例子。

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。

## Object.observe()，Object.unobserve()

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

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

上面代码中，Object.observe方法监听一个空对象o，一旦o发生变化（比如新增或删除一个属性），就会触发回调函数。

Object.observe方法指定的回调函数，接受一个数组（changes）作为参数。该数组的成员与对象的变化一一对应，也就是说，对象发生多少个变化，该数组就有多少个成员。每个成员是一个对象（change），它的name属性表示发生变化源对象的属性名，oldValue属性表示发生变化前的值，object属性指向变动后的源对象，type属性表示变化的种类，目前共支持六种变化：add、update、delete、setPrototype、reconfigure（属性的attributes对象发生变化）、preventExtensions（当一个对象变得不可扩展时，也就不必再观察了）。

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

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

上面的代码表示，只在发生delete事件时，才会调用回调函数。

Object.unobserve方法用来取消监听。

Object.unobserve(o, observer);

注意，Object.observe和Object.unobserve这两个方法不属于ES6，而是属于ES7的一部分，Chrome 36已经开始支持了。

# 函数的扩展

## 函数参数的默认值

ES6允许为函数的参数设置默认值。

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

任何带有默认值的参数，被视为可选参数。不带默认值的参数，则被视为必需参数。

利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误。

function throwIfMissing() {  
 throw new Error('Missing parameter');  
 }  
  
function foo(mustBeProvided = throwIfMissing()) {  
 return mustBeProvided;  
}  
  
foo()  
// Error: Missing parameter

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

## rest参数

ES6引入rest参数（...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。

前面说过，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，否则会报错。

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

## 扩展运算符

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

function push(array, ...items) {  
 array.push(...items);  
}  
  
function add(x, y) {  
 return x + y;  
}  
  
var numbers = [4, 38];  
add(...numbers) // 42

扩展运算符可以简化求出一个数组最大元素的写法。

// ES5  
Math.max.apply(null, [14, 3, 77])  
  
// ES6  
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。

扩展运算符还可以用于数组的赋值。

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

## 箭头函数

ES6允许使用“箭头”（=>）定义函数。

var f = v => v;

上面的箭头函数等同于：

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

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

var f = () => 5;   
// 等同于  
var f = function (){ return 5 };  
  
var sum = (num1, num2) => num1 + num2;  
// 等同于  
var sum = function(num1, num2) {  
 return num1 + num2;  
};

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

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

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

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

箭头函数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);

箭头函数有几个使用注意点。

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

关于this对象，下面的代码将它绑定定义时的对象。

var handler = {  
  
 id: "123456",  
  
 init: function() {  
 document.addEventListener("click",  
 event => this.doSomething(event.type), false);  
 },  
  
 doSomething: function(type) {  
 console.log("Handling " + type + " for " + this.id);  
 }  
};

上面代码的init方法中，使用了箭头函数，这导致this绑定handler对象。否则，doSomething方法内部的this对象就指向全局对象，运行时会报错。

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

# Set和Map数据结构

## Set

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

Set本身是一个构造函数，用来生成Set数据结构。

var s = new Set();  
  
[2,3,5,4,5,2,2].map(x => s.add(x))  
  
for (i of s) {console.log(i)}  
// 2 3 4 5

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函数接受一个数组作为参数，用来初始化。

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

向Set加入值的时候，不会发生类型转换。这意味在Set中，5和“5”是两个不同的值。

set数据结构有以下方法。

* size()：返回成员总数。
* add(value)：添加某个值。
* delete(value)：删除某个值。
* 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。

下面是这些属性和方法的使用演示。

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，对象和Set的写法不同。

// 对象的写法  
  
var properties = {  
 "width": 1,  
 "height": 1  
};  
  
if (properties[someName]) {  
 // do something  
}  
  
// Set的写法  
  
var properties = new Set();  
  
properties.add("width");  
properties.add("height");  
  
if (properties.has(someName)) {  
 // do something  
}

Array.from方法可以将Set结构转为数组。

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

这就提供了一种去除数组中重复元素的方法。

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

## Map

**（1）基本用法**

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

var data = {};  
var element = document.getElementById("myDiv");  
  
data[element] = metadata;

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，对象也可以当作键。

var m = new Map();  
  
o = {p: "Hello World"};  
  
m.set(o, "content")  
  
console.log(m.get(o))  
// "content"

上面代码将对象o当作m的一个键。

Map函数也可以接受一个数组进行初始化。

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"

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

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

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

同理，同样的值的两个实例，在Map结构中被视为两个键。

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

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

**（2）属性和方法**

Map数据结构有以下属性和方法。

* size：返回成员总数。
* set(key, value)：设置一个键值对。
* get(key)：读取一个键。
* has(key)：返回一个布尔值，表示某个键是否在Map数据结构中。
* delete(key)：删除某个键。
* clear()：清除所有成员。

下面是一些用法实例。

var m = new Map();   
  
m.set("edition", 6) // 键是字符串  
m.set(262, "standard") // 键是数值  
m.set(undefined, "nah") // 键是undefined  
  
var hello = function() {console.log("hello");}  
m.set(hello, "Hello ES6!") // 键是函数  
  
m.has("edition") // true  
m.has("years") // false  
m.has(262) // true  
m.has(undefined) // true  
m.has(hello) // true  
  
m.delete(undefined)  
m.has(undefined) // false  
  
m.get(hello) // Hello ES6!  
m.get("edition") // 6

**（3）遍历**

Map原生提供三个遍历器。

* keys()：返回键名的遍历器。
* values()：返回键值的遍历器。
* entries()：返回所有成员的遍历器。

下面是使用实例。

for (let key of map.keys()) {  
 console.log("Key: %s", key);  
}  
  
for (let value of map.values()) {  
 console.log("Value: %s", value);  
}  
  
for (let item of map.entries()) {  
 console.log("Key: %s, Value: %s", item[0], item[1]);  
}  
  
// same as using map.entries()  
for (let item of map) {  
 console.log("Key: %s, Value: %s", item[0], item[1]);  
}

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

map.forEach(function(value, key, map)) {  
 console.log("Key: %s, Value: %s", key, value);  
};

forEach方法还可以接受第二个参数，用来绑定this。

var reporter = {  
 report: function(key, value) {  
 console.log("Key: %s, Value: %s", key, value);  
 }  
};  
  
map.forEach(function(value, key, map) {  
 this.report(key, value);  
}, reporter);

上面代码中，forEach方法的回调函数的this，就指向reporter。

## WeakMap

WeakMap结构与Map结构基本类似，唯一的区别是它只接受对象作为键名（null除外），不接受原始类型的值作为键名。

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 map = new WeakMap();  
var element = document.querySelector(".element");  
  
map.set(element, "Original");  
  
var value = map.get(element);  
console.log(value); // "Original"  
  
element.parentNode.removeChild(element);  
element = null;  
  
value = map.get(element);  
console.log(value); // undefined

WeakMap还有has和delete方法，但没有size方法，也无法遍历它的值，这与WeakMap的键不被计入引用、被垃圾回收机制忽略有关。

# Iterator和for...of循环

## Iterator（遍历器）

遍历器（Iterator）是一种协议，任何对象只要部署这个协议，就可以完成遍历操作。在ES6中，遍历操作特指for...of循环。

它的作用主要有两个，一是为遍历对象的属性提供统一的接口，二是为使得对象的属性能够按次序排列。

ES6的遍历器协议规定，部署了next方法的对象，就具备了遍历器功能。next方法必须返回一个包含value和done两个属性的对象。其中，value属性是当前遍历位置的值，done属性是一个布尔值，表示遍历是否结束。

function makeIterator(array){  
 var nextIndex = 0;  
  
 return {  
 next: function(){  
 return nextIndex < array.length ?  
 {value: array[nextIndex++], done: false} :  
 {value: undefined, done: true};  
 }  
 }  
}  
  
var it = makeIterator(['a', 'b']);  
  
it.next().value // 'a'  
it.next().value // 'b'  
it.next().done // true

上面代码定义了一个makeIterator函数，它的作用是返回一个遍历器对象，用来遍历参数数组。请特别注意，next返回值的构造。

下面是一个无限运行的遍历器例子。

function idMaker(){  
 var index = 0;  
   
 return {  
 next: function(){  
 return {value: index++, done: false};  
 }  
 }  
}  
  
var it = idMaker();  
  
it.next().value // '0'  
it.next().value // '1'  
it.next().value // '2'  
// ...

## for...of循环

ES6中，一个对象只要部署了next方法，就被视为具有iterator接口，就可以用for...of循环遍历它的值。下面用上一节的idMaker函数生成的it遍历器作为例子。

for (var n of it) {  
 if (n > 5)  
 break;  
 console.log(n);  
}  
// 0  
// 1  
// 2  
// 3  
// 4  
// 5

上面代码说明，for...of默认从0开始循环。

数组原生具备iterator接口。

const arr = ['red', 'green', 'blue'];  
  
for(let v of arr) {  
 console.log(v);  
}  
// red  
// green  
// blue

JavaScript原有的for...in循环，只能获得对象的键名，不能直接获取键值。ES6提供for...of循环，允许遍历获得键值。

var arr = ["a", "b", "c", "d"];  
for (a in arr) {  
 console.log(a);  
}  
// 0  
// 1  
// 2  
// 3  
  
for (a of arr) {  
 console.log(a);   
}  
// a  
// b  
// c  
// d

上面代码表明，for...in循环读取键名，for...of循环读取键值。

对于Set和Map结构的数据，可以直接使用for...of循环。

var engines = Set(["Gecko", "Trident", "Webkit", "Webkit"]);  
for (var e of engines) {  
 console.log(e);  
}  
// Gecko  
// Trident  
// Webkit  
  
var es6 = new Map();  
es6.set("edition", 6);  
es6.set("committee", "TC39");  
es6.set("standard", "ECMA-262");  
for (var [name, value] of es6) {  
 console.log(name + ": " + value);  
}  
// edition: 6  
// committee: TC39  
// standard: ECMA-262

上面代码演示了如何遍历Set结构和Map结构，后者是同时遍历键名和键值。

对于普通的对象，for...of结构不能直接使用，会报错，必须部署了iterator接口后才能使用。但是，这样情况下，for...in循环依然可以用来遍历键名。

var es6 = {  
 edition: 6,  
 committee: "TC39",  
 standard: "ECMA-262"  
};  
  
for (e in es6) {  
 console.log(e);  
}  
// edition  
// committee  
// standard  
  
for (e of es6) {  
 console.log(e);  
}  
// TypeError: es6 is not iterable

上面代码表示，for...in循环可以遍历键名，for...of循环会报错。

总结一下，for...of循环可以使用的范围包括数组、类似数组的对象（比如arguments对象、DOM NodeList对象）、Set和Map结构、后文的Generator对象，以及字符串。下面是for...of循环用于字符串和DOM NodeList对象的例子。

// 字符串的例子  
  
let str = "hello";  
  
for (let s of str) {  
 console.log(s);  
}  
// h  
// e  
// l  
// l  
// o  
  
// DOM NodeList对象的例子  
  
let paras = document.querySelectorAll("p");  
  
for (let p of paras) {  
 p.classList.add("test");  
}

# Generator 函数

## 含义

所谓Generator，简单说，就是一个内部状态的遍历器，即每调用一次遍历器，内部状态发生一次改变（可以理解成发生某些事件）。ES6引入了generator函数，作用就是可以完全控制内部状态的变化，依次遍历这些状态。

generator函数就是普通函数，但是有两个特征。一是，function关键字后面有一个星号；二是，函数体内部使用yield语句，定义遍历器的每个成员，即不同的内部状态（yield语句在英语里的意思就是“产出”）。

function\* helloWorldGenerator() {  
 yield 'hello';  
 yield 'world';  
}  
  
var hw = helloWorldGenerator();

上面代码定义了一个generator函数helloWorldGenerator，它的遍历器有两个成员“hello”和“world”。调用这个函数，就会得到遍历器。

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

hw.next()   
// { value: 'hello', done: false }  
  
hw.next()  
// { value: 'world', done: false }  
  
hw.next()  
// { value: undefined, done: true }  
  
hw.next()  
// { value: undefined, done: true }

上面代码一共调用了四次next方法。

第一次调用，函数开始执行，直到遇到第一句yield语句为止。next方法返回一个对象，它的value属性就是当前yield语句的值hello，done属性的值false，表示遍历还没有结束。

第二次调用，函数从上次yield语句停下的地方，一直执行到下一个yield语句。next方法返回的对象的value属性就是当前yield语句的值world，done属性的值false，表示遍历还没有结束。

第三次调用，函数从上次yield语句停下的地方，一直执行到函数结束。next方法返回的对象的value属性就是函数最后的返回值，由于上例的函数没有return语句（即没有返回值），所以value属性的值为undefined，done属性的值true，表示遍历已经结束。

第四次调用，此时函数已经运行完毕，next方法返回与前一次一样的值，value属性为undefined，done属性为true。以后再调用next方法，返回的都是这个值。

总结一下，Generator函数使用iterator接口，每次调用next方法的返回值，就是一个标准的iterator返回值：有着value和done两个属性的对象。其中，value是yield语句后面那个表达式的值，done是一个布尔值，表示是否遍历结束。

Generator函数的本质，其实是提供一种可以暂停执行的函数。yield语句就是暂停标志，next方法遇到yield，就会暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回对象的value属性的值。当下一次调用next方法时，再继续往下执行，直到遇到下一个yield语句。如果没有再遇到新的yield语句，就一直运行到函数结束，将return语句后面的表达式的值，作为value属性的值，如果该函数没有return语句，则value属性的值为undefined。

由于yield后面的表达式，直到调用next方法时才会执行，因此等于为JavaScript提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

yield语句与return语句有点像，都能返回紧跟在语句后面的那个表达式的值。区别在于每次遇到yield，函数暂停执行，下一次再从该位置继续向后执行，而return语句不具备位置记忆的功能。

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，只有调用next方法时，函数f才会执行。

## next方法的参数

yield语句本身没有返回值，或者说总是返回undefined。next方法可以带一个参数，该参数就会被当作上一个yield语句的返回值。

function\* f() {  
 for(var i=0; true; i++) {  
 var reset = yield i;  
 if(reset) { i = -1; }  
 }  
}  
  
var g = f();  
  
g.next() // { value: 0, done: false }  
g.next() // { value: 1, done: false }  
g.next(true) // { value: 0, done: false }

上面代码先定义了一个可以无限运行的generator函数f，如果next方法没有参数，每次运行到yield语句，变量reset的值总是undefined。当next方法带一个参数true时，当前的变量reset就被重置为这个参数（即true），因此i会等于-1，下一轮循环就会从-1开始递增。

## 异步操作的应用

generator函数的这种暂停执行的效果，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();   
}   
  
// 加载UI  
loadUI.next()   
  
// 卸载UI  
loadUI.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。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。

总结一下，如果某个操作非常耗时，可以把它拆成N步。

function\* longRunningTask() {  
 yield step1();  
 yield step2();  
 // ...  
 yield stepN();  
}

然后，使用一个函数，按次序自动执行所有步骤。

scheduler(longRunningTask());  
  
function scheduler(task) {  
 setTimeout(function () {  
 if (!task.next().done) {  
 scheduler(task);  
 }  
 }, 0);  
}

注意，yield语句是同步运行，不是异步运行（否则就失去了取代回调函数的设计目的了）。实际操作中，一般让yield语句返回Promises对象。

var Q = require('q');  
   
function delay(milliseconds) {  
 var deferred = Q.defer();  
 setTimeout(deferred.resolve, milliseconds);  
 return deferred.promise;  
}  
  
function \*f(){  
 yield delay(100);  
};

上面代码使用Promise的函数库Q，yield语句返回的就是一个Promise对象。

## for...of循环

for...of循环可以自动遍历Generator函数，且此时不再需要调用next方法。

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 (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。

## yield\*语句

如果yield命令后面跟的是一个遍历器，需要在yield命令后面加上星号，表明它返回的是一个遍历器。这被称为yield\*语句。

let delegatedIterator = (function\* () {  
 yield 'Hello!';  
 yield 'Bye!';  
}());  
  
let delegatingIterator = (function\* () {  
 yield 'Greetings!';  
 yield\* delegatedIterator;  
 yield 'Ok, bye.';  
}());  
  
for(let value of delegatingIterator) {  
 console.log(value);  
}  
// "Greetings!  
// "Hello!"  
// "Bye!"  
// "Ok, bye."

上面代码中，delegatingIterator是代理者，delegatedIterator是被代理者。由于yield\* delegatedIterator语句得到的值，是一个遍历器，所以要用星号表示。

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']

# Promise对象

## 基本用法

ES6原生提供了Promise对象。所谓Promise对象，就是代表了未来某个将要发生的事件（通常是一个异步操作）。它的好处在于，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对象还提供了一整套完整的接口，使得可以更加容易地控制异步操作。Promise对象的概念的详细解释，请参考[《JavaScript标准参考教程》](http://javascript.ruanyifeng.com/)。

ES6的Promise对象是一个构造函数，用来生成Promise实例。下面是Promise对象的基本用法。

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

上面代码表示，Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve方法和reject方法。如果异步操作成功，则用resolve方法将Promise对象的状态变为“成功”（即从pending变为resolved）；如果异步操作失败，则用reject方法将状态变为“失败”（即从pending变为rejected）。

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

下面是一个使用Promise对象的简单例子。

function timeout(ms) {  
 return new Promise((resolve) => {  
 setTimeout(resolve, ms);  
 });  
}  
  
timeout(100).then(() => {  
 console.log('done');  
});

上面代码的timeout方法返回一个Promise实例对象，表示一段时间以后改变自身状态，从而触发then方法绑定的回调函数。

下面是一个用Promise对象实现的Ajax操作的例子。

var getJSON = function(url) {  
 var promise = new Promise(function(resolve, reject){  
 var client = new XMLHttpRequest();  
 client.open("GET", url);  
 client.onreadystatechange = handler;  
 client.responseType = "json";  
 client.setRequestHeader("Accept", "application/json");  
 client.send();  
  
 function handler() {  
 if (this.readyState === this.DONE) {  
 if (this.status === 200) {   
 resolve(this.response);   
 } else {   
 reject(this);   
 }  
 }  
 };  
 });  
  
 return promise;  
};  
  
getJSON("/posts.json").then(function(json) {  
 // continue  
}, function(error) {  
 // handle errors  
});

## 链式操作

then方法返回的是一个新的Promise对象，因此可以采用链式写法。

getJSON("/posts.json").then(function(json) {  
 return json.post;  
}).then(function(post) {  
 // proceed  
});

上面的代码使用then方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

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

getJSON("/post/1.json").then(function(post) {  
 return getJSON(post.commentURL);  
}).then(function(comments) {  
 // 对comments进行处理  
});

这种设计使得嵌套的异步操作，可以被很容易得改写，从回调函数的“横向发展”改为“向下发展”。

## catch方法：捕捉错误

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) {  
 // some code  
}).catch(function(error) {  
 // 处理前一个回调函数运行时发生的错误  
 console.log('发生错误！', error);  
});

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。

getJSON("/post/1.json").then(function(post) {  
 return getJSON(post.commentURL);  
}).then(function(comments) {  
 // some code  
}).catch(function(error) {  
 // 处理前两个回调函数的错误  
});

## Promise.all方法

Promise.all方法用于将多个异步操作（或Promise对象），包装成一个新的Promise对象。当这些异步操作都完成后，新的Promise对象的状态才会变为fulfilled；只要其中一个异步操作失败，新的Promise对象的状态就会变为rejected。

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

## Promise.resolve方法

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

var jsPromise = Promise.resolve($.ajax('/whatever.json'));

上面代码将jQuery生成deferred对象，转为一个新的ES6的Promise对象。

如果Promise.resolve方法的参数，不是具有then方法的对象（又称thenable对象），则返回一个新的Promise对象，且它的状态为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，它的状态为fulfilled，所以回调函数会立即执行，Promise.resolve方法的参数就是回调函数的参数。

## async函数

async函数是用来取代回调函数的另一种方法。

只要函数名之前加上async关键字，就表明该函数内部有异步操作。该异步操作应该返回一个promise对象，前面用await关键字注明。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。

function timeout(ms) {  
 return new Promise((resolve) => {  
 setTimeout(resolve, ms);  
 });  
}  
  
async function asyncValue(value) {  
 await timeout(50);  
 return value;  
}  
  
(async function() {  
 var value = await asyncValue(42);  
 assert.equal(42, value);  
 done();  
})();

上面代码中，asyncValue函数前面有async关键字，表明函数体内有异步操作。执行的时候，遇到await语句就会先返回，等到timeout函数执行完毕，再返回value。后面的匿名函数前也有async关键字，表明该函数也需要暂停，等到await后面的asyncValue(42)得到值以后，再执行后面的语句。

async函数并不属于ES6，而是被列入了ES7，但是traceur编译器已经实现了这个功能。

# Class和Module

## Class

ES6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。

//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)

上面代码定义了一个class类，可以看到里面有一个constructor函数，这就是构造函数。而this关键字则代表实例对象。

class之间可以通过extends关键字，实现继承。

class ColorPoint extends Point {  
  
 constructor(x, y, color) {  
 super(x, y); // 等同于super.constructor(x, y)  
 this.color = color;  
 }  
  
 toString() {  
 return this.color+' '+super();  
 }  
  
}

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。在constructor方法内，super就指代父类Point；在toString方法内，super()表示对父类求值，由于此处需要字符串，所以会自动调用父类的toString方法。

## Module的基本用法

**（1）export和import**

ES6实现了模块功能，试图解决JavaScript代码的依赖和部署上的问题，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。

模块功能有两个关键字：export和import。export用于用户自定义模块，规定对外接口；import用于输入其他模块提供的功能，同时创造命名空间（namespace），防止函数名冲突。

ES6允许将独立的JS文件作为模块，也就是说，允许一个JavaScript脚本文件调用另一个脚本文件。最简单的模块就是一个JS文件，里面使用export关键字输出变量。

// profile.js  
export var firstName = 'David';  
export var lastName = 'Belle';  
export var year = 1973;

上面是profile.js，ES6将其视为一个模块，里面用export关键字输出了三个变量。其他js文件就可以通过import关键字加载这个模块（文件）。

import {firstName, lastName, year} from './profile';  
  
function setHeader(element) {  
 element.textContent = firstName + ' ' + lastName;  
}

上面代码中import关键字接受一个对象（用大括号表示），里面指定要从其他模块导入的变量。大括号里面的变量名，必须与被导入模块对外接口的名称相同。

如果想为输入的属性或方法重新取一个名字，import语言要写成下面这样。

import { someMethod, another as newName } from './exporter';

**（2）模块的整体加载**

export关键字除了输出变量，还可以输出方法或类（class）。下面是一个circle.js文件，它输出两个方法。

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

然后，main.js引用这个模块。

// 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导入的方法。另一种写法是使用module关键字，整体导入。

// main.js  
  
module circle from 'circle';  
  
console.log("圆面积：" + circle.area(4));  
console.log("圆周长：" + circle.circumference(14));

module关键字后面跟一个变量，表示导入的模块定义在该变量上。

**（3）export default语句**

如果不想为某个属性或方法，指定输入的名称，可以使用export default语句。

// export-default.js  
  
export default function foo() {  
 console.log('foo');  
}

上面代码中的foo方法，就被称为该模块的默认方法。

在其他模块导入该模块时，import语句可以为默认方法指定任意名字。

// import-default.js  
  
import customName from './export-default';  
  
customName(); // 'foo'

显然，一个模块只能有一个默认方法。

如果要输出默认属性，只需将值跟在export default之后即可。

export default 42;

## 模块的继承

模块之间也可以继承。

假设有一个circleplus模块，继承了circle模块。

// circleplus.js  
  
export \* from 'circle';  
export var e = 2.71828182846;  
export default function(x) {  
 return Math.exp(x);  
}

上面代码中的“export \*”，表示输出circle模块的所有属性和方法，export default命令定义模块的默认方法。

这时，可以为circle中的属性或方法，改名后再输出。

export { area as circleArea } from 'circle';

加载上面模块的写法如下。

// main.js  
  
module math from "circleplus";  
import exp from "circleplus";  
console.log(exp(math.pi));

上面代码中的"import exp"表示，将circleplus模块的默认方法加载为exp方法。