

*ECMAScript 6 Primer*

# ECMAScript 6 入门

阮一峰 著



电子工业出版社

http://www.pke.com.cn

## 内 容 简 介

本书全面介绍了 ECMAScript 6 新引入的语法特性,覆盖了 ECMAScript 6 与 ECMAScript 5 的所有不同之处,对涉及的语法知识给予了详细介绍,并给出了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。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有,侵权必究。

### 图书在版编目(CIP)数据

ECMAScript 6 入门 / 阮一峰著. — 北京: 电子工业出版社, 2014.8  
ISBN 978-7-121-23836-9

I. ① E…II. ① 阮…III. ① 程序设计 IV. ① TP311.1  
中国版本图书馆 CIP 数据核字 (2014) 第 159646 号

责任编辑: 白 涛

印 刷: 中国电影出版社印刷厂

装 订: 三河市鹏成印业有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 900×640 1/16 印张: 10.5 字数: 150 千字

版 次: 2014 年 8 月第 1 版

印 次: 2014 年 8 月第 1 次印刷

定 价: 49.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 [zlts@phei.com.cn](mailto:zlts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线: (010) 88258888。



# ECMAScript 6 简介



ECMAScript 6 (以下简称 ES6) 是 JavaScript 语言的下一代标准, 正处在快速开发中, 大部分已经完成, 预计将于 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。

## 4 第 1 章 • ECMAScript 6 简介

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 本身也已被 Netscape 公司注册为商标。二是想体现这门语言的制定者是 ECMA，而不是 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。不过，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，原计划次年 8 月发布正式版本。然而在草案发布后，由于 4.0 版的目标过于激进，各方对于是否通过这个标准，产生了严重分歧。以 Yahoo、Microsoft、Google 为首的大公司，反对 JavaScript 的大幅升级，主张小幅改动；而以 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，鉴于会议的气氛，该版本的项目代号取名为 Harmony（和谐）。会后不久，ECMAScript 3.1 就改名为 ECMAScript 5。

## ECMAScript 6 入门

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。此后是 12 个月的讨论期，以听取各方反馈意见。

2015 年 6 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，而当第六版发布以后，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。关于各大浏览器的最新版本对 ES6 的支持，可以查看<http://kangax.github.io/es5-compat-table/es6/>。

## 6 第 1 章 • ECMAScript 6 简介

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，需要使用版本管理工具 `nvm` (<https://github.com/creationix/nvm>) 切换。操作如下，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。

### 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/bootstr
ap.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');
        }
    }
```

## 8 第 1 章 • ECMAScript 6 简介

```
        add(a, b){
            return a + b;
        }
    }

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

正常情况下，上面的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 编译器用来识别 ES6 代码的标识，编译器会自动将所有标记了 type=module 的代码编译为 ES5 代码，然后交给浏览器执行。

如果 ES6 代码是一个外部文件，那么可以用 script 标签插入网页。

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

### 在线转换

Traceur 提供一个在线编译器 (<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
```

ECMAScript 6 入门

```

.js" type="text/javascript"></script>
<script src="http://google.github.io/traceur-compiler/src/bootstr
ap.js" type="text/javascript"></script>
<script>
    traceur.options.experimental = true;
</script>
<script>
$traceurRuntime.ModuleStore.getAnonymousModule(function() {
    "use strict";

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

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

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

```

## 命令行转换

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

```
npm install -g traceur
```

## 10 第 1 章 • ECMAScript 6 简介

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，会在标准输出中显示运行结果，以前面的 calc.js 为例。

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

如果要将 ES6 脚本转为 ES5 代码，要采用下面的写法：

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

上面代码的 --script 选项用于指定输入文件，--out 选项用于指定输出文件。

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
```

命令行下转换得到的文件，可以放到浏览器中运行。

### Node.js 环境的用法

Traceur 的 Node.js 用法如下（假定已安装 traceur 模块）。

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

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

var result = traceur.compile(contents, {
```

ECMAScript 6 入门

```
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，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方上。
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。

## 12 第 1 章 • ECMAScript 6 简介

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，用于声明变量。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
```

上面的代码在代码块之中，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 声明的变量返回正确的值。这表明，let 声明的变量只在它所在的代码块内有效。

## 16 第2章 • let 和 const 命令

下面的代码如果使用 var，则最后输出的是“9”。

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

而如果使用 let，声明的变量仅在块级作用域内有效，于是最后输出的是“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;  
}
```

上面的代码在声明 foo 之前，就使用了这个变量，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。

## 18 第2章 • let 和 const 命令

块级作用域的出现，实际上使得广为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();
})();
```

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

## const 命令

const 用来声明常量。一旦声明，其值就不能改变。

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

```
PI = 3;  
PI // 3.1415
```

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

上面的代码表明改变常量的值是起不到作用的。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!";
```

## 20 第2章 • let 和 const 命令

```
const age = 30;
```

3

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;
```

而 ES6 允许写成下面这样。

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

上面的代码表示，可以从数组中提取值，按照位置的对应关系，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，

## 24 第3章 • 变量的解构赋值

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
```

上面代码中的第一个例子，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

## 26 第3章 • 变量的解构赋值

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
```

ECMAScript 6 入门

上面代码中的写法会报错，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。

### 交换变量的值

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

### 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();
```

### 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。

### 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。

## 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
```

### 30 第3章 • 变量的解构赋值

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

#### 输入模块的指定方法

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

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