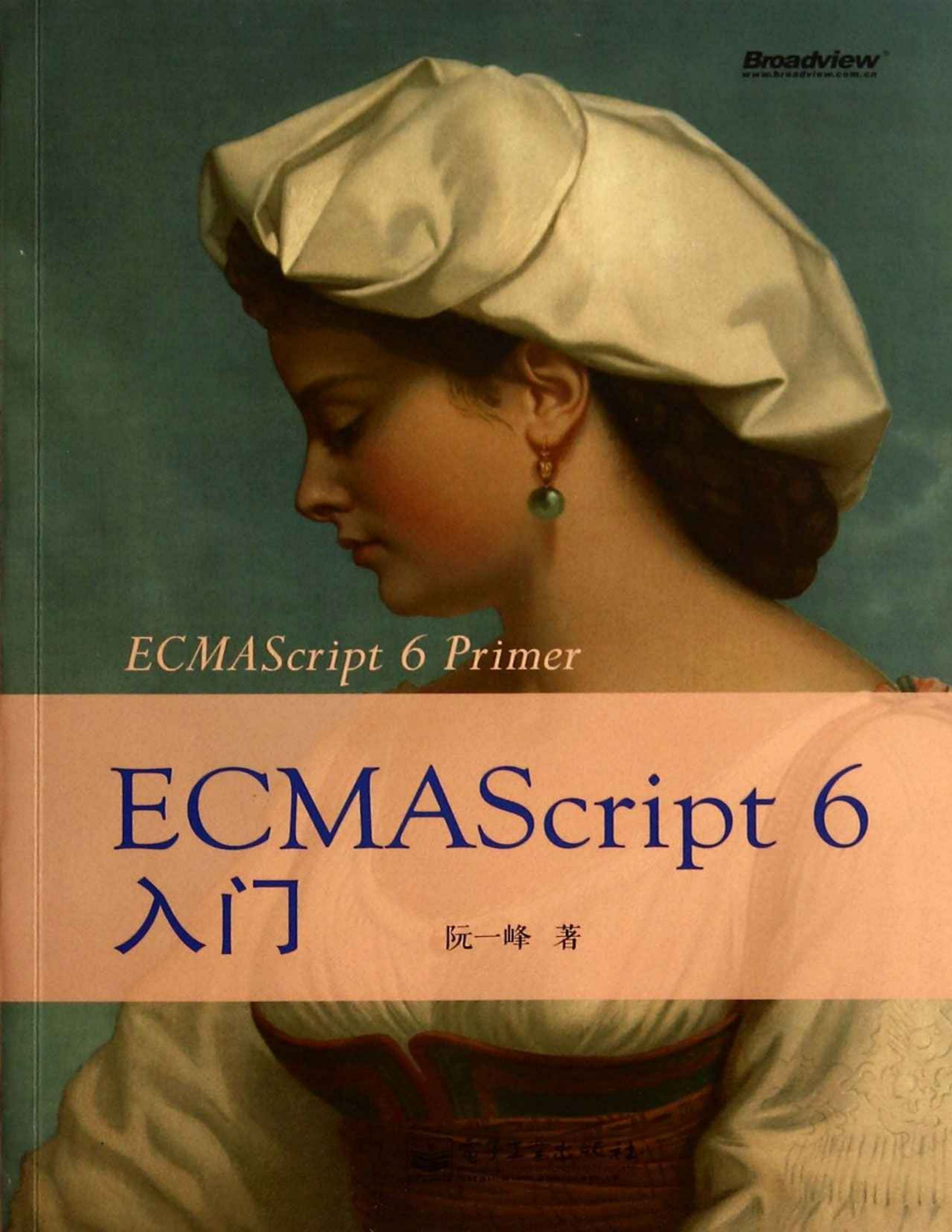
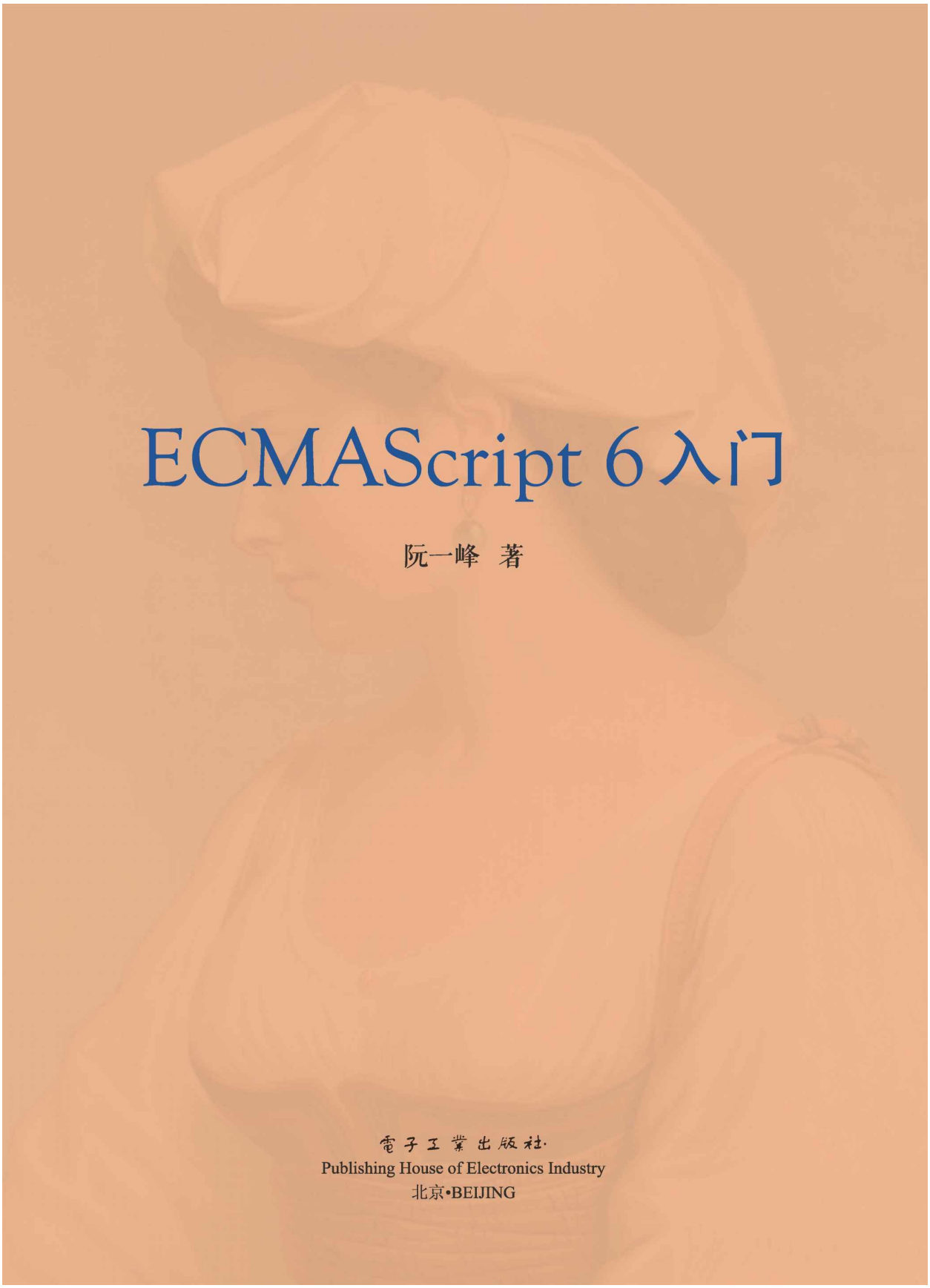

Broadview
www.broadview.com.cn
ECMAScript 6 Primer
ECMAScript 6
入门
阮一峰 著



ECMAScript 6 入门

阮一峰 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

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。

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. ①E… Ⅱ. ①阮… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

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：

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

推荐序1

为什么我们要关心标准

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

本书第1章的这句话已经清楚地告诉我们，这是一本不实用的书。我们学习了这本书，并不意味着掌握了一项实用的技术，而只是掌握了一个未来可能会发布的技术标准。而标准，有可能在将来被实现，变成主流，也有可能就仅仅是一个标准，没有人真的去实践它。如果你再了解一下第1章里面介绍的ECMAScript　4.0草案的血泪史，或者回顾一下曾经红极一时的XHTML，就更容易明白这一点了。

那我们为什么不直接忽略标准，拥抱实践就好呢？来，我们一起翻开小学课本，跟我念：柏林已经来了命令，阿尔萨斯和洛林的学校只许教德语了……（《最后一课》）

当统治者宣布一门语言成为“标准”的时候，不管是在现实生活还是技术领域里面，往往就意味着所有其他的选项自动消失了，我们只能去学习“统治者”的语言。幸运的是，在技术领域里面，跳出来争取对技术的影响力和主导权，不但不违反任何一国的宪法，往往还是被鼓励的。

因此，技术的未来发展，是我们可以去发出声音，去影响，乃至于去引领的。而要做到这些，我们需要搞清楚，ECMA和各大互联网巨头们，他们正在做什么，正在把技术往哪里引领；他们引领的方向，到底是对所有人有利的，还是只是对某些公司有利；我们中国的开发者和中

国的公司，要怎么加入到这些标准的制订过程中，把标准带到更好的方向上。

最近几年，越来越多的中国公司加入到各种国际标准组织中，参与到各种标准（尤其是在东亚文字处理、排版、输入法相关的领域）制订过程中，发出了中国技术人员的声音。随着中国国力的增强，中国开发厂商和技术人员的影响力的发展壮大，可以预见，不久的将来，中国工程师也许会深入参与到ECMAScript 7和HTML6这样的技术标准的制订过程里面，跟各国的专家一起探讨，我们中国的开发者不喜欢这样，更喜欢那样。在那些标准大会上，我们的发言权将来自于我们对标准的深入理解、我们对技术发展的独到眼光和我们建设起来的技术影响力。

作为一个JS开发者，实话说，对于ECMAScript 6里面的很多内容（比如let语句），我并不完全认同。但是很遗憾，这个标准的制定过程没我们什么事。但是如果我们从现在开始关注国际标准，翻译标准文档，让更多人了解标准，更多公司加入标准组织、参与标准制订，也许未来的中国技术圈不但会是很多人的一个圈子，还会是很有影响力的一个圈子。

“我们说的话，让世界都认真听话。”（S.H.E，《中国话》）

腾讯驻W3C顾问委员会代表 黄希彤（stone）

黄希彤（网名emu），Web性能优化（WPO）领域实践者，信息无障碍领域推动者。腾讯Web前端专家，腾讯驻W3C顾问委员会代表，腾讯QQ空间技术总监。

推荐序2

因为一件往事，我现在轻易不敢给别人写序或者书评。那天我在想，如果我要给这本书写序，是不是应该先把这本书拿给贺老（hax）看看。后来呢，我到阮一峰老师的GitHub上看了一看，发现这本书有605个star，若干个已解决和未解决的issue，所以我就放心了。开源真是好啊！

这本书是关于ES6的，我对ES6并没有特别系统的研究，但是也在工作中使用了一部分ES6的特性，使用得最多的是Promise，其他的特性只是研究，很少使用，主要是因为本身支持ES6的环境和工具有限。浏览器就不说了，现在的前端工程师在一些产品中能够抛弃IE6已经是很幸福的事情了，但是即使是IE8，离真正的ES6也还很遥远。在其他领域，比如手机游戏领域，cocos2d-js v3.0使用的脚本引擎是SpiderMonkey v28，因此情况要好很多，但是周边的一些工具，比如closure compiler不能很好地压缩和优化ES6，当然你可以采用转换工具先将ES6转成ES5，然后再做压缩和优化，但是这多出来的一步造成更多出错的可能，而且和享受ES6的语法糖的快乐相比，开销有点大——如果无论如何需要再转一步，那么为什么我们不干脆考虑TypeScript或者其他选择呢？

为什么会选择使用ES6的Promise，那是因为Promise算是比较好解决异步嵌套问题的方案，另外Promise本身在低版本下也有比较好的polyfill实现（*https://github.com/jakearchibald/es6-promise* ），对于我和一些前端工程师来说，是十分乐意为将来去写一些能够向前兼容的符合标准的代码的。

目前这个阶段，前端学习ES6，并不意味着能够很快将ES6的好处带到工作中，因为我们毕竟还受到现在的浏览器环境的制约。但是，即使单纯从学习一门编程语言的核心API的角度来说，ES6也是值得学习的。它的很多新特性，真正涉及现代编程语言概念中很流行的部分，不管是解构赋值还是迭代器或者yield，都是超棒超赞的思想，不但易于理解，也能节省很多键盘操作，而另一些诸如const、作用域之类的设定，则让脚本引擎代替程序员人肉检查做更多的事情，让我们最终上线的代码变得更加安全和更加优美。

不管怎样，ES6代表着一种前端的未来，这种未来，无疑能让前端工程师们工作得更高效，也更有乐趣。更进一步说，ECMAScript还是开放的标准，对这门语言的新特性，有什么好的想法，都是有机会提交为标准的，也就是说，前端程序员的未来，是由我们前端程序员自己来创造的，还有什么是比自由更加美好的呢？所以，为了未来，加油！

360奇舞团团长　月影

吴亮（网名月影），先后在微软亚洲研究院做过访问学生，在金蝶软件有限公司担任过核心开发工程师、设计师和项目经理，在百度电子商务事业部担任过Web开发项目经理。现任奇虎360高级技术经理，360前端团队奇舞团负责人。多年来致力于JavaScript技术和Web标准的推广，活跃于国内各技术社区，现为w3ctech顾问。

推荐序3

同大多数读者一样，我最早看到阮一峰先生的文字是在其博客上。他的第一篇博文于2003年写就，迄今已有1500多篇文章，可谓高产。阮先生并非计算机相关专业，但这一点并没有妨碍他从事技术写作，其文字朴实，思路清晰，所有人都能看懂，更能感受到他写文章的用心程度，而这本书完美地体现了他的一贯风格。另外，这本书是开源作品，也很好地践行了他一贯的贡献原则。

自我写下第一行前端代码到现在已经十来年了，前端的基础设施也发生了巨大的变化。变化最大的还是浏览器环境，从原来烂熟IE6的各种bug和hack，到现在IE6已经完全不在我的考虑范围内。其次是前端的工程化程度，2011年，我做FIS（*http://fis.baidu.com* ）时，完全没想到前端的工程化进展会如此之快。而变化最慢的，要数语言本身了，1999年发布的ECMAScript 3.0其实相当于第1版；十年后的2009年，才发布第2版：ECMAScript 5.0；预计ECMAScript 6要到2015年发布。

我的一贯主张是，要学好JavaScript，ECMAScript标准比什么书都强。在标准中，已经用最严谨的语言和最完美的角度展现了语言的实质和特性。理解语言的本质后，你已经从沙堆里挑出了珍珠，能经受得起时光的磨砺。

我从2009年开始正式接触ECMAScript规范，当时我在写百度的JavaScript基础库Tangram 1.0，ECMAScript5还处于草案状态。我自己打印了一本小册子，上下班时在地铁上慢慢看。才知道有很多问题在网络上被包装了太多次，解释得千奇百怪，但用规范的语言来描述，竟是如

此简单。

ECMAScript标准经历了很多变故——尤其是ECMAScript4那次——也从语言的角度反映了各大厂商之间的立场差异。不过，ECMAScript5的正式发布和发展，为所有Web开发者奠定了稳定的基础，尽管浏览器之间存在大量差异，尤其是DOM，但在JavaScript语言层面，都相对严格地遵循着ECMAScript5的规范。

JavaScript遵守“一个JavaScript”的原则，所有版本都需要向后兼容。Web语言的解释器版本不是由开发者而是由用户决定的，所以JavaScript无法像Python、Ruby、Perl那样，发布一个不向下兼容的大版本，这也就是ECMAScript4失败的根源，由于它会导致大量已有网页的“bug”，浏览器厂商会强烈反对。当然，ECMAScript6的strict mode也在尝试逐步淘汰一些不良实践。

ECMAScript6相比5，有了很大的进步。经过这次改进，JavaScript语法更精简，变得更有表现力了；在严格模式下，开发者受到了适当而必要的约束；新增了几种数据类型（map、set）和函数能力（Generator、迭代器）；进一步强化了JavaScript的特点（promise、proxy）；并且让JavaScript能适用于更大型的程序开发（modules、class）。更重要的是，这个规范会被浏览器厂商、不同的平台广泛支持。

实际上，所有的语言改进都是从使用者的最佳实践中提炼出来的。JavaScript的约束一直很少，这一灵活性让开发者能相当自由地积累形形色色的使用经验和实践，也就是说，我们所有ECMAScript的使用者，也是其标准的间接贡献者。

百度高级工程师，前端通用组技术负责人　雷志兴

雷志兴（网名berg），资深工程师，2007年加入百度工作至今，负责过多项前端基础技术、架构的设计和搭建；骑行爱好者，行程万余公里；微信公众号“行云出岫”（DevLife）的维护者。

前言

2012年年底，我开始动手做一个开源项目《JavaScript标准参考教程》（*https://github.com/ruanyf/jstutorial* ）。原来的设想是将自己的学习笔记整理成一本书，哪里料到，这个项目不断膨胀，最后变成了关于ECMAScript 5及其外围API的全面解读和参考手册，写了一年多还没写完。

那个项目的最后一章就是ECMAScript 6的语法简介。那一章也是越写越长，最后我不得不决定，把它独立出来，作为一个新项目，也就是您现在看到的这本书。

JavaScript已经是互联网开发的第一大语言，而且正在变成一种全领域的语言。著名程序员Jeff Atwood甚至提出了一条“Atwood定律”：“所有可以用JavaScript编写的程序，最终都会出现JavaScript的版本。”（Any application that can be written in JavaScript will eventually be written in JavaScript.）

ECMAScript正是JavaScript的国际标准，这就决定了该标准的重要性。而ECMAScript 6是ECMAScript历史上最大的一次版本升级，在语言的各个方面都有极大的变化，即使是熟练的JavaScript程序员，也需要重新学习。由于ES6的设计目标是企业级开发和大型项目，所以可以预料，除了互联网开发者，将来还会有大量应用程序开发者（甚至操作系统开发者）成为ES6的学习者。

我写作这本书的目标，就是想为上面这些学习者，提供一本篇幅较短、简明易懂、符合中文表达习惯的ES6教程。它由浅入深、循序渐

进，既有重要概念的讲解，又有API接口的罗列，便于日后当作参考手册查阅，还提供大量示例代码，让读者不仅一看就懂，还能举一反三，直接复制用于实际项目之中。

需要声明的是，为了突出重点，本书只涉及ES6与ES5的不同之处，不对JavaScript已有的语法做全面讲解，毕竟市场上这样的教程已有很多了。因此，本书不是JavaScript入门教材，不适合初学者。阅读本书之前，需要对JavaScript的基本语法有所了解。

我本人也是一个ES6的学习者，不敢说自己有多高的水平，只是较早地接触了这个主题，持续地读了许多资料，追踪标准的进展，做了详细的笔记而已。虽然我尽了最大努力，并且原稿在GitHub上公开后，已经得到了大量的勘误，但是本书的不如人意之处恐怕还是有不少。

欢迎大家访问本书的项目主页（*https://github.com/ruanyf/es6tutorial*），提出意见，以及提交pull request。这些都会包括在本书的下一个版本中。

阮一峰

2014年6月4日，写于上海

目　录

第1章　ECMAScript 6简介

ECMAScript6（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JavaScript2.0。

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ECMAScript1.0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，ECMAScript2.0版发布。

1999年12月，ECMAScript3.0版发布，成为JavaScript的通行标准，得到了广泛支持。

2007年10月，ECMAScript4.0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ECMAScript4.0的开发，将其中涉及现有功能改善的一小部分，发布为ECMAScript3.1，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，ECMAScript3.1就改名为ECMAScript5。

2009年12月，ECMAScript5.0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ECMAScript6；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，ECMAScript5.1版发布，并且成为ISO国际标准（ISO/IEC 16262:2011）。

2013年3月，ECMAScript6草案冻结，不再添加新功能。新的功能设想将被放到ECMAScript7。

2013年12月，ECMAScript6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，ECMAScript6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/* 。

Google公司的V8引擎已经部署了ES6的部分特性。使用Node.js0.11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
<!-- 将Traceur 编译器用于网页 -->
<script src="http://google.github.io/traceur-compiler/src/bootstr
<!-- 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
<script src="http://google.github.io/traceur-compiler/src/bootstr
<script>
    traceur.options.experimental = true;
</script>
<script>
$traceurRuntime.ModuleStore.getAnonymousModule(function() {
    "use strict";

    var Calc = function Calc() {
```

```
        console.log('Calc constructor');
    };

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

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

命令行转换

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

```
npm install -g traceur
```

安装成功后，就可以在命令行下使用traceur了。

traceur直接运行ES6脚本文件，会在标准输出中显示运行结果，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, {
   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。

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。

第2章　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。

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 (vari = 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。

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;
```

第3章　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，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
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");
```

第4章　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

在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
```

上面的代码说明，对于4个字节储存的字符，JavaScript不能正确处理。字符串长度会被误判为2，而且charAt方法无法读取到字符，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）。上面的代码表明，它会正确返回四字节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四个字节组成的最简单方法。

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


is32Bit("吉") // 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) // "吉"
```

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“\u*xxxx* ”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"
// "吉"


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

上面的代码表示，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，只要将超过0xFFFF的编号放入大括号，就能得到正确解读。

```
"\u{20BB7}"
// "吉"
```

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
```

上面的代码表示，如果不添加u修饰符，正则表达式就会认为字符串s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中只有indexOf方法，可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
```

上面的代码表示，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，而其他两个方法则针对从第*n* 个位置直到字符串结束的字符。

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，叫作“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修饰符则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
```

上面的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修饰符没有位置要求，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_"]
```

上面的代码每次匹配，都是从剩余字符串的头部开始。

进一步说，y修饰符隐含了头部匹配的标志^。

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

上面的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。

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 '' 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"
```

上面代码的表示，在模板字符串中嵌入变量，需要将变量名写在${}之中。

第5章　数值的扩展

二进制和八进制数值表示法

ES6提供了二进制和八进制数值的新写法，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，对于非数值一律返回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()，移植到了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，整数和浮点数使用同样的储存方法，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为-2^{53} ～2^{53} 。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

Math.trunc()

Math.trunc 方法用于去除一个数的小数部分，返回其整数部分。

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

数学方法

ES6还在Math对象上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，即$\sqrt[3]{x}$。
- Math.clz32(x)返回x的32位二进制整数表示形式的前导0的个数。
- Math.cosh(x)返回x的双曲余弦（hyperbolic cosine）。
- Math.expm1(x)返回$e^x - 1$。
- Math.fround(x)返回x的单精度浮点数形式。
- Math.hypot(...values)返回所有参数的平方和的平方根。

- Math.imul(x, y)返回两个参数以32位整数形式相乘的结果。
- Math.log1p(x)返回ln(1+x)。
- Math.log10(x)返回以10为底的x的对数，即lg$x$ 。
- Math.log2(x)返回以2为底的x的对数，即$\log_2 x$ 。
- 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）。

第6章　数组的扩展

Array.from()

Array.from()用于将两类对象转换为真正的数组：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);
});
```

上面的代码中，querySelectorAll方法返回的是一个类似数组的对象，只有将这个对象转换为真正的数组，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]
```

上面的代码说明，只有当参数个数不少于2个，Array()才会返回由所提供参数组成的新数组。

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
```

从上面的代码可以看到，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，从而弥补了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]
```

上面的代码表明，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

别在于，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提供了简洁写法，允许直接通过现有数组生成新数组，称为数组推导（array comprehension）。

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

```
 2];

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

上面的代码表示，通过for...of结构，数组a2直接在a1的基础上生成。

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

for...of 后面还可以附加if语句，用来设定循环的限制条件。

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

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

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

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

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

数组推导可以替代map和filter方法。

```
[for (i of [1, 2, 3]) i *


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

```
 i });

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

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

在一个数组推导中，还可以使用多个for...of结构，构成多重循环。

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

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

上面的代码在一个数组推导之中，使用了三个for...of结构。

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

由于字符串可以视为数组，因此字符串也可以直接用于数组推导。

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

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

上面的代码使用了数组推导对字符串进行处理。

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

Array.observe()，Array.unobserve()

这两个方法用于监听（取消监听）数组的变化，指定回调函数。

它们的用法与Object.observe　和Object.unobserve方法完全一致，也属于ES7的一部分，请参阅第7章“对象的扩展”。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）。

第7章　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，不同之处只有两点：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　方法用于将源对象（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()

__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，所有主流浏览器（包括IE11）都部署了这个属性。

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

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

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;
}
```

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"
```

上面的代码中，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"
```

Symbol

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"
```

上面的代码表示，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:
        ...
    }
}
```

上面的代码中，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!' });
```

上面的代码通过点结构和Object.defineProperty 两种方法，为对象增加了一个属性。

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，可以理解为在目标对象之前，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
```

上面的代码就是Proxy 构造函数使用实例，它接受两个参数，第一个是所要代理的目标对象（上例中是一个空对象），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将得到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
```

上面的代码表示，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);
```

上面的代码中，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，对象发生多少变化，该数组就有多少成员。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ES7的一部分，Chrome自版本36起开始支持。

第8章　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，都被视为可选参数。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
```

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

rest参数

ES6引入了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
```

上面代码中的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量的参数。

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，将一个数组转换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);
```

上面的代码表示，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，将数组转换为一个参数序列，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，就使用一对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，因而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，下面的代码将它与定义时的对象绑定。

```
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。

第9章　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，只不过其成员值都是唯一的，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
```

上面的代码通过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。这意味着，在Set中5和“5”是两个不同的值。

Set结构有以下属性。

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

Set结构有以下方法。

- 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的写法有哪些不同。

```
//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结构转换为数组。

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

这样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

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;
```

上面的代码原意是将一个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被自动转换为字符串“[Object HTMLDivElement]”。

为了解决这个问题，ES6 提供了Map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"
```

上面的代码将对象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
```

上面代码中的set和get方法，表面上是针对同一个键，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
```

上面的代码中，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。

属性和方法

Map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结构中。
- delete(key)：删除某个键。
- clear()：清除所有成员。

下面是一些用法示例。

```
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
```

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。

下面是使用示例。

```
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);
```

上面的代码中，forEach方法的回调函数中的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属性，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。

第10章　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，二是使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
```

上面的代码定义了一个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
```

上面的代码说明，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提供了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
```

上面的代码表明，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
```

上面的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，否则会报错，必须部署了iterator接口才能使用。但是，在这种情况下，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
```

上面的代码表示，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");
}
```

第11章　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引入的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在英语里的意思就是“产出”）。

```
function*
```

```
 helloWorldGenerator() {
    yield 'hello';
    yield 'world';
    return 'ending';
}
var hw = helloWorldGenerator();
```

上面的代码定义了一个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），直到遇到下一条yield语句为止。也就是说，next方法就是在遍历yield语句定义的内部状态。

```
hw.next()
// { value: 'hello', done: false }

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

hw.next()
// { value: 'ending', done: true }

hw.next()
// { value: undefined, done: true }
```

上面的代码一共调用了四次next方法。

第一次调用，函数开始执行，直到遇到第一条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属性的值为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，一直执行到下一条

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属性的值为false，表示遍历还没有结束。

第三次调用，函数从上次yield语句停下的地方，一直执行到return语句（如果没有return语句，就执行到函数结束）。next方法返回的对象的value属性，就是紧跟在return语句后面的表达式的值（如果没有return语句，则value属性的值为undefined），done属性的值为true，表示遍历已经结束。

第四次调用，此时函数已经运行完毕，next方法返回对象的value属性为undefined，done属性为true。以后再调用next方法，返回的都是这个值。

总结一下，Generator函数使用了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，直到遇到下一条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);
```

上面的代码中，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，该参数会被当作上一条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 }
```

上面的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 函数的一个重要的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()
```

上面的代码表示，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中，按部就班，非常清晰。

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();
    }
}
```

上面的代码打开文本文件，使用yield语句可以手动逐行读取文件。

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（否则就违背了取代回调函数的设计目的了）。实际操作中，一般让yield语句返回Promise对象。

```
var Q = require('q');

function delay(milliseconds) {
    var deferred = Q.defer();
    setTimeout(deferred.resolve, milliseconds);
    return deferred.promise;
```

```
}

function*
```

```
 f(){
    yield delay(100);
};
```

上面的代码使用了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);
}
```

从上面的代码可以看出，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，则需要在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."
```

上面的代码中，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些的例子，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']
```

第12章　Promise对象

基本用法

ES6原生提供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。关于Promise对象这一概念的详细解释，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
});
```

上面的代码表示，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');
});
```

上面代码中的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地改写，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换为Promise对象，Promise.resolve方法就起这个作用。

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

上面的代码将jQuery生成的deferred对象，转换为一个新的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
```

上面的代码会生成一个新的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。

```
async function getStockPrice(symbol, currency) {
    let price = await getStockPrice(symbol);
    return convert(price, currency);
}
```

上面的代码是一个获取股票报价的函数，函数前面的async关键字表明该函数将返回一个Promise对象。调用该函数时，当遇到await关键字，则立即返回它后面的表达式（getStockPrice函数）产生的Promise对象，不再执行函数体内后面的语句。等到getStockPrice完成，再自动回到函数体内，执行剩下的语句。

下面是一个更具一般性的例子。

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

async function asyncValue(value) {
  await timeout(50);
```

```
  return value;
}
```

上面的代码中，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。

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。

第13章　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)
```

上面的代码定义了一个“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();
  }

}
```

上面的代码定义了一个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

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 的内容，ES6 将其视为一个模块，里面用export关键字输出了三个变量。export 的写法，除了像上面这样，还有另外一种，两者是等价的。

```
// profile.js
var firstName = 'David';
var lastName = 'Belle';
var year = 1973;
export {firstName, lastName, year}
```

使用export定义模块以后，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;
}
```

上面的代码中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 语句要写成下面这样。

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

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circle.js文件的内容，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));
```

上面的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。

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命令定义了模块的默认方法。

这时，可以对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。

参考链接

官方文件

- ECMAScript 6 Language Specification
  http://people.mozilla.org/~jorendorff/es6-draft.html
  语言规格草案。
- harmony:proposals
  http://wiki.ecmascript.org/doku.php?id=harmony:proposals
  ES6的各种提案。

综合介绍

- Sayanee Basu
  Use ECMAScript 6 Today
  http://net.tutsplus.com/articles/news/ecmascript-6-today/
- Ariya Hidayat
  Toward Modern Web Apps with ECMAScript 6
  http://www.sencha.com/blog/toward-modern-web-apps-with-ecmascript-6/
- Dale Schouten
  10 Ecmascript-6 tricks you can perform right now
  http://html5hub.com/10-ecmascript-6-tricks-you-can-perform-right-now/
- Domenic Denicola
  ES6: The Awesome Parts
  http://www.slideshare.net/domenicdenicola/es6-the-awesome-parts

- Nicholas C. Zakas
  Understanding ECMAScript 6
  https://github.com/nzakas/understandinges6
- Justin Drake
  ECMAScript 6 in Node.JS
  https://github.com/JustinDrake/node-es6-examples
- Ryan Dao
  Summary of ECMAScript 6 major features
  http://ryandao.net/portal/content/summary-ecmascript-6-major-features
- Luke Hoban
  ES6 features
  https://github.com/lukehoban/es6features
- Traceur-compiler
  Language Features
  https://github.com/google/traceur-compiler/wiki/LanguageFeatures
  Traceur文档列出的一些ES6 例子。

语法点

- Nick Fitzgerald
  Destructuring Assignment in ECMAScript 6
  http://fitzgeraldnick.com/weblog/50/
  详细介绍解构赋值的用法。
- Nicholas C. Zakas
  Understanding ECMAScript 6 arrow functions
  http://www.nczonline.net/blog/2013/09/10/understanding-ecmascript-6-arrow-functions/

- Jack Franklin
  Real Life ES6 - Arrow Functions
  http://javascriptplayground.com/blog/2014/04/real-life-es6-arrow-fn/
- Axel Rauschmayer
  Handling required parameters in ECMAScript 6
  http://www.2ality.com/2014/04/required-parameters-es6.html
- Axel Rauschmayer
  ECMAScript 6's new array methods
  http://www.2ality.com/2014/05/es6-array-methods.html
  对ES6新增的数组方法的全面介绍。
- Nicholas C. Zakas
  Creating defensive objects with ES6 proxies
  http://www.nczonline.net/blog/2014/04/22/creating-defensive-objects-with-es6-proxies/
- Addy Osmani
  Data-binding Revolutions with Object.observe()
  http://www.html5rocks.com/en/tutorials/es7/observe/
  介绍Object.observer()的概念。

Generator

- Mozilla Developer Network
  Iterators and generators
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators
- Matt Baker
  Replacing callbacks with ES6 Generators

http://flippinawesome.org/2014/02/10/replacing-callbacks-with-es6-generators/

- Steven Sanderson
  Experiments with Koa and JavaScript Generators
  http://blog.stevensanderson.com/2013/12/21/experiments-with-koa-and-javascript-generators/
- jmar777
  What's the Big Deal with Generators?
  http://devsmash.com/blog/whats-the-big-deal-with-generators
- Marc Harter
  Generators in Node.js: Common Misconceptions and Three Good Use Cases
  http://strongloop.com/strongblog/how-to-generators-node-js-yield-use-cases/
  讨论Generator函数的作用。
- Axel Rauschmayer
  Iterators and generators in ECMAScript 6
  http://www.2ality.com/2013/06/iterators-generators.html
  探讨Iterator和Generator 的设计目的。

Promise对象

- Jake Archibald
  JavaScript Promises: There and back again
  http://www.html5rocks.com/en/tutorials/es6/promises/
- Tilde
  rsvp.js

https://github.com/tildeio/rsvp.js

- Sandeep Panda
  An Overview of JavaScript Promises
  http://www.sitepoint.com/overview-javascript-promises/
  ES6 Promise入门介绍
- Jafar Husain
  Async Generators
  https://docs.google.com/file/d/0B4PVbLpUIdzoMDR5dWstRllXblU/view?pli=1
  对async 与Generator 混合使用的一些讨论

工具

- Google
  traceur-compiler
  https://github.com/google/traceur-compiler
  Traceur 编译器
- Casper Beyer
  ECMAScript 6 Features and Tools
  http://caspervonb.github.io/2014/03/05/ecmascript6-features-and-tools.html
- Stoyan Stefanov
  Writing ES6 today with jstransform
  http://www.phpied.com/writing-es6-today-with-jstransform/
- ES6 Module Loader
  ES6 Module Loader Polyfill
  https://github.com/ModuleLoader/es6-module-loader

在浏览器和Node.js加载ES6模块的一个库，文档里对ES6模块有详细解释

- Paul Miller
  es6-shim
  https://github.com/paulmillr/es6-shim
  一个针对老式浏览器，模拟ES6 部分功能的垫片库（shim）
- army8735
  Javascript Downcast
  https://github.com/army8735/jsdc
  国产的ES6 到ES5 的转码器

索引

Arrow function

async

B

Binary

block

C

Class

Comprehension

const

D

Destructuring

E

export

export default

F

for...of

Function default parameters

G

Generator

H

Harmony

I

import

Iterator

J

JavaScript

L

let

M

Map

Math

Module

Module.Inheritance

N

next

Node.js

Number.isFinite

Number.isInteger

Number.isNaN

Number.parseFloat

Number.parseInt

O

Object computed property names

Object literal shorthand

Object.assign

Object.getPrototypeOf

Object.is

Object.observe

Object.setPrototypeOf

Object.unobserve

Octonary

P

Promise

Promise.all

Promise.prototype.catch

Promise.prototype.then

Promise.resolve

Proxy

R

RegExp

Rest parameters

S

SafeInteger

Set

spread operator

String.fromCodePoint

String.prototype.codePointAt

String.prototype.contains

String.prototype.endsWith

String.prototype.repeat

String.prototype.startsWith

Symbol

T

TC39

Template string

Traceur

U

Unicode

W

WeakMap

Y

yield

yield*

关于封面

封面图像为奥地利画家Wolfgang Böhm（1823–1890）的《意大利少女》（*The Italian Beauty*），作于1875年，收藏于波士顿公共图书馆。