ES6-in-Depth

深入浅出ES6

Jason Orendorff Benjamin Peterson Nick Fitzgerald 等著

刘振涛 午川 译

InfoQ

免费在线版本

（非印刷免费在线版）

InfoQ 中文站出品

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com

卷首语

曾经听许多前端从业者说：“前端发展太快了。”这里的快，十有八九是说层出不穷的新概念，余下的一二，大抵只是抒发一下心中的苦闷罢——前两日刚习得的新技术转眼就“落后”——仔细品味这苦闷，除却不得不持续奔跑的无奈，更多的是一口气，一口卯足了劲儿也要把新知识全数揽入囊中的不服气。作为刚入行的新人，对这点体会颇深。就像是蓦地从某个时间点切入，半数时间向前走，半数时间向后看，瞻前顾后，回味揣摩这十年间的岁月精魄，还得翘首盼着花花新世界，不时再问自己一句，这样走下去真的会好么？是的，其实答案人尽皆知，同时也无人知晓，因为没人能预言未来，顶多只能预测未来，但有一件事情永不会错，当你笃定地沿着一条路走下去，结果通常不会太糟糕，但凡能在浮躁的社会冷静下来潜心磨砺，多少总会有收获。幸而我有意弱化了对新信息的执念，开始做一些事情，《深入浅出 ES6》就是其中一件。

纵观整个系列，亦即纵观 ECMAScript 2015 的整个体系，吸取了诸多成功经验：借鉴自 CoffeeScript 的箭头函数；始于 C++项目 Xanadu，接着被 E 语言采用，后来分别于 Python 和 JavaScript 框架 Dojo 中以 Deferred 对象的面貌出现的 Promise 规范（详见 Async JavaScript 一书 3.1 章）；借鉴了 C++、Java、C#以及 Python 等语言的 for-of 循环语句；部分借鉴 Mustache、Nunjucks 的模板字符串。

当然，新的语言体系也在之前的基础上查漏补缺：弥补块级作用域变量缺失的 let 和 const 关键字；弥补面向大型项目缺失的模块方案；标准委员会甚至为 JavaScript 增加了类特性，有关这一方面的特性褒贬不一，Douglas Crockford 曾在 2014 年的 Nordic.js 大会发表了题为《The Better Parts》的演讲，重新阐述了他个人对于 ECMAScript 6 的看法，他认为 Class 特性是所有新标准中最糟糕的创新（我个人也略赞同这一说法，类的加入虽然有助于其它语言的使用者开始使用 JavaScript，但是却无法发挥出 JavaScript 原型继承的巨大优势）；以及为了保持非侵入式弥补其它新特性而诞生的 Symbols。

其它的新特性也相当诱人，熟练掌握可以大幅度提升开发效率：迭代器 Iterator、生成器 Generators、不定参数 Rest、默认参数 Default、解构 Destructuring、生成器 Generator、代理 Proxy，以及几种新类型：Set、Map、WeakSet、WeakMap、集合 Collection。

以上提及的新特性只是其中的一部分，更多的新特性等待着大家进一步挖掘。整个系列的翻译历时 150 余天，坚持专栏翻译的日子艰苦也快乐，编辑徐川细心地帮我审校每一篇文章，编辑丁晓昀赠予钱歌川先生详解翻译之著作让我大开眼界，与李松峰老师的交流也让我深刻理解了“阅读、转换、表达”的奥义所在，最感谢我的母亲，在我遇到困难需要力量的时候永远支持着我。选择 ES6 作为前端生涯的切入点实之我幸，恰遇这样的机会让我可以一心一意地向前走，向未来走。我很敬佩在“洪荒”和“战乱”年代沉淀无数经 验的前辈们，你们在各种不确定的因素中左右互搏，为终端用户提供统一的用户体验，直到如今你们依然孜孜不倦地吸取业内新鲜的经验。技术在进步，也为前端人 提供着无限的可能性，我们有责任也有义务去推动新标准的发展和普及，诚然在商业的大环境下我们不愿放弃每一寸用户的土壤，但携众人之力定将能推动用户终端 的革新。ES7 标准的提案纷纷提上日程，用不了多久也将登上前端大舞台。也感谢午川同学友情提供译文第十章“集合 Collection”，让我在困难时期得以顺利过渡。最后祝愿国内前端社区向着更光明美好的未来蓬勃生长！

目录

1

ES6 是什么

欢迎来到 ES6 深入浅出！JavaScript 的新版本离我们越来越近，我们将通过每周一节的系列课程一起探索 ECMAScript 6 新世界。ES6 中包含了许多新的语言特性，它们将使 JS 变得更加强大，更富表现力。在接下来的几周内，我们将一一深入了解它们。但在我们开始详细学习之前，我认为十分有必要花几分钟讲解一下 ES6 到底是什么，以及你可以从中学到什么！

ECMAScript 发生了什么变化

编程语言 JavaScript 是 ECMAScript 的实现和扩展，由 ECMA（一个类似 W3C 的标准组织）参与进行标准化。ECMAScript 定义了：

语言语法 – 语法解析规则、关键字、语句、声明、运算符等。

类型 – 布尔型、数字、字符串、对象等。

原型和继承

内建对象和函数的标准库 – JSON、Math、数组方法、对象自省方法等。

ECMAScript 标准不定义 HTML 或 CSS 的相关功能，也不定义类似 DOM（文档对象模型）的 Web API，这些都在独立的标准中进行定义。ECMAScript 涵盖了各种环境中 JS 的使用场景，无论是浏览器环境还是类似 node.js 的非浏览器环境。

新标准

上周，ECMAScript 语言规范第 6 版最终草案提请 Ecma 大会审查，这意味着什么呢？

这意味着在今年夏天，我们将迎来最新的 JavaScript 核心语言标准。

这无疑是一则重磅新闻。早在 2009 年，上一版 ES5 刚刚发布，自那时起，ES 标准委员会一直在紧锣密鼓地筹备新的 JS 语言标准——ES6。

ES6 是一次重大的版本升级，与此同时，由于 ES6 秉承着最大化兼容已有代码的设计理念，你过去编写的 JS 代码将继续正常运行。事实上，许多浏览器已经支持部分 ES6 特性，并将继续努力实现其余特性。这意味着，在一些已经实现部分特性的浏览器中，你的 JS 代码已经可以正常运行。如果到目前为止你尚未遇到任何兼容性问题，那么你很有可能将不会遇到这些问题，浏览器正飞速实现各种新特性。

版本号 6

ECMAScript 标准的历史版本分别是 1、2、3、5。

那么为什么没有第 4 版？其实，在过去确实曾计划发布提出巨量新特性的第 4 版，但最终却因想法太过激进而惨遭废除（这一版标准中曾经有一个极其复杂的支持泛型和类型推断的内建静态类型系统）。

ES4 饱受争议，当标准委员会最终停止开发 ES4 时，其成员同意发布一个相对谦和的 ES5 版本，随后继续制定一些更具实质性的新特性。这一明确的协商协议最终命名为“Harmony”，因此，ES5 规范中包含这样两句话：

> ECMAScript 是一门充满活力的语言，并在不断进化中。
>
> 未来版本的规范中将持续进行重要的技术改进。

这一声明许下了一个未来的承诺。

兑现承诺

2009 年发布的改进版本 ES5，引入了 Object.create()、Object.defineProperty()、getters 和 setters、严格模式以及 JSON 对象。我已经使用过所有这些新特性，并且我非常喜欢 ES5 做出的改进。但事实上，这些改进并没有深入影响我编写 JS 代码的方式，对我来说最大的革新大概就是新的数组方法：.map()、. filter()这些。

但是，ES6 并非如此！经过持续几年的磨砺，它已成为 JS 有史以来最实质的升级，新的语言和库特性就像无主之宝，等待有识之士的发掘。新的语言特性涵盖范围甚广，小到受欢迎的语法糖，例如箭头函数（arrow functions）和简单的字符串插值（string interpolation），大到烧脑的新概念，例如代理（proxies）和生成器（generators）。

ES6 将彻底改变你编写 JS 代码的方式！

这一系列旨在向你展示如何仔细审阅 ES6 提供给 JavaScript 程序员的这些新特性。

我们将从一个经典的“遗漏特性”说起，十年来我一直期待在 JavaScript 中看到的它。所以从现在起就加入我们吧，一起领略一下 ES6 迭代器（iterators）和新的 for-of 循环！

2

迭代器和 for-of 循环

我们如何遍历数组中的元素？20 年前 JavaScript 刚萌生时，你可能这样实现数组遍历：

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

自 ES5 正式发布后，你可以使用内建的 forEach 方法来遍历数组：

```
myArray.forEach(function (value) {
console.log(value);
});
```

这段代码看起来更加简洁，但这种方法也有一个小缺陷：你不能使用 break 语句中断循环，也不能使用 return 语句返回到外层函数。

当然，如果只用 for 循环的语法来遍历数组元素也很不错。

那么，你一定想尝试一下 for-in 循环：

```
for (var index in myArray) { // 千万别这样做
  console.log(myArray[index]);
}
```

这绝对是一个糟糕的选择，为什么呢？

- 在这段代码中，赋给 index 的值不是实际的数字，而是字符串“0”、“1”、“2”，此时很可能在无意之间进行字符串算数计算，例如：“2” + 1 == “21”，这给编码过程带来极大的不便。
- 作用于数组的 for-in 循环体除了遍历数组元素外，还会遍历自定义属性。举个例子，如果你的数组中有一个可枚举属性 myArray.name，循环将额外执行一次，遍历到名为“name”的索引。就连数组原型链上的属性都能被访问到。
- 最让人震惊的是，在某些情况下，这段代码可能按照随机顺序遍历数组元素。
- 简而言之，for-in 是为普通对象设计的，你可以遍历得到字符串类型的键，因此不适用于数组遍历。

强大的 for-of 循环

还记得在第一章“ES6 是什么”中我向你们承诺过的话么？ES6 不会破坏你已经写好的 JS 代码。目前看来，成千上万的 Web 网站依赖 for-in 循环，其中一些网站甚至将其用于数组遍历。如果想通过修正 for-in 循环增加数组遍历支持会让这一切变得更加混乱，因此，标准委员会在 ES6 中增加了一种新的循环语法来解决目前的问题。

就像这样：

```
for (var value of myArray) {
  console.log(value);
}
```

是的，与之前的内建方法相比，这种循环方式看起来是否有些眼熟？那好，我们将要探究一下 for-of 循环的外表下隐藏着哪些强大的功能。现在，只需记住：

- 这是最简洁、最直接的遍历数组元素的语法
- 这个方法避开了 for-in 循环的所有缺陷
- 与 forEach()不同的是，它可以正确响应 break、continue 和 return 语句

for-in 循环用来遍历对象属性。

for-of 循环用来遍历数据—例如数组中的值。

但是，不仅如此！

for-of 循环也可以遍历其它的集合

for-of 循环不仅支持数组，还支持大多数类数组对象，例如 DOM NodeList 对象。

for-of 循环也支持字符串遍历，它将字符串视为一系列的 Unicode 字符来进行遍历：

```
for (var chr of "") {
  alert(chr);
}
```

它同样支持 Map 和 Set 对象遍历。

对不起，你一定没听说过 Map 和 Set 对象。他们是 ES6 中新增的类型。我们将在后续的文章讲解这两个新的类型。如果你曾在其它语言中使用过 Map 和 Set，你会发现 ES6 中的并无太大出入。

举个例子，Set 对象可以自动排除重复项：

```
// 基于单词数组创建一个 set 对象
var uniqueWords = new Set(words);
```

生成 Set 对象后，你可以轻松遍历它所包含的内容：

```
for (var word of uniqueWords) {
   console.log(word);
}
```

Map 对象稍有不同：内含的数据由键值对组成，所以你需要使用解构（destructuring）来将键值对拆解为两个独立的变量：

```
for (var [key, value] of phoneBookMap) {
   console.log(key + "'s phone number is: " + value);
}
```

解构也是 ES6 的新特性，我们将在另一篇文章中讲解。看来我应该记录这些优秀的主题，未来有太多的新内容需要一一剖析。

现在，你只需记住：未来的 JS 可以使用一些新型的集合类，甚至会有更多的类型陆续诞生，而 for-of 就是为遍历所有这些集合特别设计的循环语句。

for-of 循环不支持普通对象，但如果你想迭代一个对象的属性，你可以用 for-in 循环（这也是它的本职工作）或内建的 Object.keys()方法：

```
// 向控制台输出对象的可枚举属性
for (var key of Object.keys(someObject)) {
  console.log(key + ": " + someObject[key]);
}
```

深入理解

> “能工摹形，巧匠窃意。”——巴勃罗 毕卡索

ES6 始终坚持这样的宗旨：凡是新加入的特性，势必已在其它语言中得到强有力的实用性证明。

举个例子，新加入的 for-of 循环像极了 C++、Java、C#以及 Python 中的循环语句。与它们一样，这里的 for-of 循环支持语言和标准库中提供的几种不同的数据结构。它同样也是这门语言中的一个扩展点（译注：关于扩展点，建议参考 1. 浅析扩展点 2. What are extensions and extension points?）。

正如其它语言中的 for/foreach 语句一样，**for-of 循环语句通过方法调用来遍历各种集合**。数组、Maps 对象、Sets 对象以及其它在我们讨论的对象有一个共同点，它们都

有一个迭代器方法。

你可以给任意类型的对象添加迭代器方法。

当你为对象添加 myObject.toString()方法后，就可以将对象转化为字符串，同样地，当你向任意对象添加 myObject[Symbol.iterator]()方法，就可以遍历这个对象了。

举个例子，假设你正在使用 jQuery，尽管你非常钟情于里面的.each()方法，但你还是想让 jQuery 对象也支持 for-of 循环，你可以这样做：

```
// 因为 jQuery 对象与数组相似
// 可以为其添加与数组一致的迭代器方法
jQuery.prototype[Symbol.iterator] = Array.prototype[Symbol.iterator];
```

好的，我知道你在想什么，那个[Symbol.iterator]语法看起来很奇怪，这段代码到底做了什么呢？这里通过 Symbol 处理了一下方法的名称。标准委员会可以把这个方法命名为.iterator()方法，但是如果你的代码中的对象可能也有一些.iterator()方法，这一定会让你感到非常困惑。于是在 ES6 标准中使用 symbol 来作为方法名，而不是使用字符串。

你大概也猜到了，Symbols 是 ES6 中的新类型，我们会在后续的文章中讲解。现在，你需要记住，基于新标准，你可以定义一个全新的 symbol，就像 Symbol.iterator，如此一来可以保证不与任何已有代码产生冲突。这样做的代价是，这段代码的语法看起来会略显生硬，但是这微乎其微代价却可以为你带来如此多的新特性和新功能，并且你所做的这一切可以完美地向后兼容。

所有拥有[Symbol.iterator]()的对象被称为可迭代的。在接下来的文章中你会发现，可迭代对象的概念几乎贯穿于整门语言之中，不仅是 for-of 循环，还有 Map 和 Set 构造函数、解构赋值，以及新的展开操作符。

迭代器对象

现在，你将无须亲自从零开始实现一个对象迭代器，我们会在下一篇文章详细讲解。

为了帮助你理解本文，我们简单了解一下迭代器（如果你跳过这一章，你将错过非常精彩的技术细节）。

for-of 循环首先调用集合的[Symbol.iterator]()方法，紧接着返回一个新的迭代器对象。迭代器对象可以是任意具有.next()方法的对象；for-of 循环将重复调用这个方法，每次循环调用一次。举个例子，这段代码是我能想出来的最简单的迭代器：

```
var zeroesForeverIterator = {
 [Symbol.iterator]: function () {
   return this;
  },
  next: function () {
  return {done: false, value: 0};
 }
};
```

每一次调用.next()方法，它都返回相同的结果，返回给 for-of 循环的结果有两种可能：(a) 我们尚未完成迭代；(b) 下一个值为 0。这意味着(value of zeroesForeverIterator) {}将会是一个无限循环。当然，一般来说迭代器不会如此简单。

这个迭代器的设计，以及它的.done 和.value 属性，从表面上看与其它语言中的迭代器不太一样。在 Java 中，迭代器有分离的.hasNext()和.next()方法。在 Python 中，他们只有一个.next() 方法，当没有更多值时抛出 StopIteration 异常。但是所有这三种设计从根本上讲都返回了相同的信息。

迭代器对象也可以实现可选的.return()和.throw(exc)方法。如果 for-of 循环过早退出会调用.return()方法，异常、break 语句或 return 语句均可触发过早退出。如果迭代器需要执行一些清洁或释放资源的操作，可以在.return()方法中实现。大多数迭代器方法无须实现这一方法。.throw(exc)方法的使用场景就更特殊了：for-of 循环永远不会调用它。但是我们还是会在下一篇文章更详细地讲解它的作用。

现在我们已了解所有细节，可以写一个简单的 for-of 循环然后按照下面的方法调用重写被迭代的对象。

首先是 for-of 循环：

```
for (VAR of ITERABLE) {
  一些语句 }
```

然后是一个使用以下方法和少许临时变量实现的与之前大致相当的示例，：

```
var $iterator = ITERABLE[Symbol.iterator]();
var $result = $iterator.next();
while (!$result.done) {
   VAR = $result.value;
   一些语句
   $result = $iterator.next();
 }
```

这段代码没有展示.return()方法是如何处理的，我们可以添加这部分代码，但我认为这对于我们正在讲解的内容来说过于复杂了。for-of 循环用起来很简单，但是其背后有着非常复杂的机制。

我何时可以开始使用这一新特性？

目前，对于 for-of 循环新特性，所有最新版本 Firefox 都（部分）支持（译注：从 FF 13 开始陆续支持相关功能，FF 36 - FF 40 基本支持大部分特性），在 Chrome 中可以通过访问 chrome://flags 并启用“实验性 JavaScript”来支持。微软的 Spartan 浏览器支持，但是 IE 不支持。如果你想在 web 环境中使用这种新语法，同时需要支持 IE 和 Safari，你可以使用 Babel 或 Google 的 Traceur 这些编译器来将你的 ES6 代码翻译为 Web 友好的 ES5 代码。

而在服务端，你不需要类似的编译器，io.js 中默认支持 ES6 新语法（部分），在 Node 中需要添加--harmony 选项来启用相关特性。

{done: true}

哟！

好的，我们今天的讲解就到这里，但是对于 for-of 循环的使用远没有结束。

在 ES6 中有一种新的对象与 for-of 循环配合使用非常契合，我没有提及它因为它是我们下周文章的主题，我认为这种新特性是 ES6 种最梦幻的地方，如果你尚未在类似 Python 和 C#的语言中遇到它，你一开始很可能会发现它令人难以置信，但是这是编写迭代器最简单的方式，在重构中非常有用，并且它很可能改变我们书写异步代码的方式，无论是在浏览器环境还是服务器环境，所以，下周的深入浅出 ES6 中，请务必一起来仔细看看 ES6 的生成器：generators。

3

生成器 Generators

今天的这篇文章令我感到非常兴奋，我们将一起领略 ES6 中最具魔力的特性。

为什么说是“最具魔力的”？对于初学者来说，此特性与 JS 之前已有的特性截然不同，可能会觉得有点晦涩难懂。但是，从某种意义上来说，它使语言内部的常态行为变得更加强大，如果这都不算有魔力，我不知道还有什么能算。

不仅如此，此特性可以极大地简化代码，它甚至可以帮助你逃离“回调地狱”。

既然新特性如此神奇，那么就一起深入了解它的魔力吧！

ES6 生成器（Generators）简介

什么是生成器？

我们从一个示例开始：

```
function* quips(name) {
  yield "你好 " + name + "!";
  yield "希望你能喜欢这篇介绍 ES6 的译文";
  if (name.startsWith("X")) {
    yield "你的名字 " + name + "  首字母是 X，这很酷！";
  }
  yield "我们下次再见！";
}
```

这是一只会说话的猫，这段代码很可能代表着当今互联网上最重要的一类应用。（试

着点击这个链接，与这只猫互动一下，如果你感到有些困惑，回到这里继续阅读）。

这段代码看起来很像一个函数，我们称之为生成器函数，它与普通函数有很多共同点，但是二者有如下区别：

普通函数使用 function 声明，而生成器函数使用 function*声明。

在生成器函数内部，有一种类似 return 的语法：关键字 yield。二者的区别是，普通函数只可以 return 一次，而生成器函数可以 yield 多次（当然也可以只 yield 一次）。在生成器的执行过程中，遇到 yield 表达式立即暂停，后续可恢复执行状态。

这就是普通函数和生成器函数之间最大的区别，普通函数不能自暂停，生成器函数可以。

生成器做了什么？

当你调用 quips()生成器函数时发生了什么？

```
> var iter = quips("jorendorff");
  [object Generator]
> iter.next()
  { value: "你好 jorendorff!", done: false }
> iter.next()
  { value: "希望你能喜欢这篇介绍 ES6 的译文", done: false }
> iter.next()
  { value: "我们下次再见！", done: false }
> iter.next()
  { value: undefined, done: true }
```

你大概已经习惯了普通函数的使用方式，当你调用它们时，它们立即开始运行，直到遇到 return 或抛出异常时才退出执行，作为 JS 程序员你一定深谙此道。

生成器调用看起来非常类似：quips("jorendorff")。但是，当你调用一个生成器时，

它并非立即执行，而是返回一个已暂停的生成器对象（上述实例代码中的 iter）。你可将这个生成器对象视为一次函数调用，只不过立即冻结了，它恰好在生成器函数的最顶端的第一行代码之前冻结了。

每当你调用生成器对象的.next()方法时，函数调用将其自身解冻并一直运行到下一个 yield 表达式，再次暂停。

这也是在上述代码中我们每次都调用 iter.next()的原因，我们获得了 quips()函数体中 yield 表达式生成的不同的字符串值。

调用最后一个 iter.next()时，我们最终抵达生成器函数的末尾，所以返回结果中 done 的值为 true。抵达函数的末尾意味着没有返回值，所以返回结果中 value 的值为 undefined。

现在回到会说话的猫的 demo 页面，尝试在循环中加入一个 yield，会发生什么？

如果用专业术语描述，每当生成器执行 yields 语句，生成器的堆栈结构（本地变量、参数、临时值、生成器内部当前的执行位置）被移出堆栈。然而，生成器对象保留了对这个堆栈结构的引用（备份），所以稍后调用.next()可以重新激活堆栈结构并且继续执行。

值得特别一提的是，**生成器不是线程**，在支持线程的语言中，多段代码可以同时运行，通通常导致竞态条件和非确定性，不过同时也带来不错的性能。生成器则完全不同。当生成器运行时，它和调用者处于同一线程中，拥有确定的连续执行顺序，永不并发。与系统线程不同的是，生成器只有在其函数体内标记为 yield 的点才会暂停。

现在，我们了解了生成器的原理，领略过生成器的运行、暂停恢复运行的不同状态。那么，这些奇怪的功能究竟有何用处？

生成器是迭代器！

上周，我们学习了 ES6 的迭代器，它是 ES6 中独立的内建类，同时也是语言的一个扩展点，通过实现[Symbol.iterator]()和.next()两个方法你就可以创建自定义迭代器。

实现一个接口不是一桩小事，我们一起实现一个迭代器。举个例子，我们创建一个简单的 range 迭代器，它可以简单地将两个数字之间的所有数相加。首先是传统 C 的 for(;;)循环：

```
// 应该弹出三次 "ding"
for (var value of range(0, 3)) {
  alert("Ding! at floor #" + value);
}
```

使用 ES6 的类的解决方案（如果不清楚语法细节，无须担心，我们将在接下来的文章中为你讲解）：

```
class RangeIterator {
  constructor(start, stop) {
    this.value = start;
    this.stop = stop;
  }
  [Symbol.iterator]() { return this; }
  next() {
    var value = this.value;
    if (value < this.stop) {
      this.value++;
      return {done: false, value: value};
    } else {
      return {done: true, value: undefined};
    }
  }
}
```

```
// 返回一个新的迭代器，可以从 start 到 stop 计数。
function range(start, stop) {
  return new RangeIterator(start, stop);
}
```

查看代码运行情况。

这里的实现类似 Java 或 Swift 中的迭代器，不是很糟糕，但也不是完全没有问题。我们很难说清这段代码中是否有 bug，这段代码看起来完全不像我们试图模仿的传统 for (;;)循环，迭代器协议迫使我们拆解掉循环部分。

此时此刻你对迭代器可能尚无感觉，他们用起来很酷，但看起来有些难以实现。

你大概不会为了使迭代器更易于构建从而建议我们为 JS 语言引入一个离奇古怪又野蛮的新型控制流结构，但是既然我们有生成器，是否可以在这里应用它们呢？一起尝试一下：

```
function* range(start, stop) {
  for (var i = start; i < stop; i++)
    yield i;
}
```

查看代码运行情况。

以上 4 行代码实现的生成器完全可以替代之前引入了一整个 RangeIterator 类的 23 行代码的实现。可行的原因是：**生成器是迭代器**。所有的生成器都有内建.next()和[Symbol.iterator]()方法的实现。你只须编写循环部分的行为。

我们都非常讨厌被迫用被动语态写一封很长的邮件，不借助生成器实现迭代器的过程与之类似，令人痛苦不堪。当你的语言不再简练，说出的话就会变得难以理解。RangeIterator 的实现代码很长并且非常奇怪，因为你需要在不借助循环语法的前提下为它添加循环功能的描述。所以生成器是最好的解决方案！

我们如何发挥作为迭代器的生成器所产生的最大效力？

使任意对象可迭代。编写生成器函数遍历这个对象，运行时 yield 每一个值。然后将这个生成器函数作为这个对象的[Symbol.iterator]方法。

简化数组构建函数。假设你有一个函数，每次调用的时候返回一个数组结果，就像这样：

```
// 拆分一维数组 icons
// 根据长度 rowLength
function splitIntoRows(icons, rowLength) {
  var rows = [];
  for (var i = 0; i < icons.length; i += rowLength) {
    rows.push(icons.slice(i, i + rowLength));
  }
  return rows;
}
```

使用生成器创建的代码相对较短：

```
function* splitIntoRows(icons, rowLength) {
  for (var i = 0; i < icons.length; i += rowLength) {
    yield icons.slice(i, i + rowLength);
  }
}
```

行为上唯一的不同是，传统写法立即计算所有结果并返回一个数组类型的结果，使用生成器则返回一个迭代器，每次根据需要逐一地计算结果。

获取异常尺寸的结果。你无法构建一个无限大的数组，但是你可以返回一个可以生成一个永无止境的序列的生成器，每次调用可以从中取任意数量的值。

重构复杂循环。你是否写过又丑又大的函数？你是否愿意将其拆分为两个更简单的

部分？现在，你的重构工具箱里有了新的利刃——生成器。当你面对一个复杂的循环时，你可以拆分出生成数据的代码，将其转换为独立的生成器函数，然后使用 for (var data of myNewGenerator(args))遍历我们所需的数据。

构建与迭代相关的工具。ES6 不提供用来过滤、映射以及针对任意可迭代数据集进行特殊操作的扩展库。借助生成器，我们只须写几行代码就可以实现类似的工具。

举个例子，假设你需要一个等效于 Array.prototype.filter 并且支持 DOM NodeLists 的方法，可以这样写：

```
function* filter(test, iterable) {
  for (var item of iterable) {
    if (test(item))
      yield item;
  }
}
```

你看，生成器魔力四射！借助它们的力量可以非常轻松地实现自定义迭代器，记住，迭代器贯穿 ES6 的始终，它是数据和循环的新标准。

以上只是生成器的冰山一角，最重要的功能请继续观看！

生成器和异步代码

这是我在一段时间以前写的一些 JS 代码

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

可能你已经在自己的代码中见过类似的片段，异步 API 通常需要一个回调函数，

这意味着你需要为每一次任务执行编写额外的异步函数。所以如果你有一段代码需要完成三个任务，你将看到类似的三层级缩进的代码，而非简单的三行代码。

后来我就这样写了：

```
}).on('close', function () {
  done(undefined, undefined);
}).on('error', function (error) {
  done(error);
});
```

异步 API 拥有错误处理规则，不支持异常处理。不同的 API 有不同的规则，大多数的错误规则是默认的；在有些 API 里，甚至连成功提示都是默认的。

这些是到目前为止我们为异步编程所付出的代价，我们正慢慢开始接受异步代码不如等效同步代码美观又简洁的这个事实。

生成器为你提供了避免以上问题的新思路。

实验性的 Q.async()尝试结合 promises 使用生成器产生异步代码的等效同步代码。举个例子：

```
// 制造一些噪音的同步代码。
function makeNoise() {
  shake();
  rattle();
  roll();
}
// 制造一些噪音的异步代码。
// 返回一个 Promise 对象
// 当我们制造完噪音的时候会变为 resolved
function makeNoise_async() {
```

```
  return Q.async(function* () {
    yield shake_async();
    yield rattle_async();
    yield roll_async();
  });
}
```

二者主要的区别是，异步版本必须在每次调用异步函数的地方添加 yield 关键字。

在 Q.async 版本中添加一个类似 if 语句的判断或 try/catch 块，如同向同步版本中添加类似功能一样简单。与其它异步代码编写方法相比，这种方法更自然，不像是学一门新语言一样辛苦。

如果你已经看到这里，你可以试着阅读来自 James Long 的更深入地讲解生成器的文章。

生成器为我们提供了一个新的异步编程模型思路，这种方法更适合人类的大脑。相关工作正在不断展开。此外，更好的语法或许会有帮助，ES7 中有一个有关异步函数的提案，它基于 promises 和生成器构建，并从 C#相似的特性中汲取了大量灵感。

如何应用这些疯狂的新特性？

在服务器端，现在你可以在 io.js 中使用 ES6（在 Node 中你需要使用--harmony 这个命令行选项）。

在浏览器端，到目前为止只有 Firefox 27+和 Chrome 39+支持了 ES6 生成器。如果要在 web 端使用生成器，你需要使用 Babel 或 Traceur 来将你的 ES6 代码转译为 Web 友好的 ES5。

起初，JS 中的生成器由 Brendan Eich 实现，他的设计参考了 Python 生成器，而此 Python 生成器则受到 Icon 的启发。他们早在 2006 年就在 Firefox 2.0 中移植了相关代码。但是，标准化的道路崎岖不平，相关语法和行为都在原先的基础上有所改动。Firefox

和 Chrome 中的 ES6 生成器都是由编译器 hacker Andy Wingo 实现的。这项工作由彭博赞助支持（没听错，就是大名鼎鼎的那个彭博！）。

yield;

生成器还有更多未提及的特性，例如：.throw()和.return()方法、可选参数.next()、yield*表达式语法。由于行文过长，估计观众老爷们已然疲乏，我们应该学习一下生成器，暂时 yield 在这里，剩下的干货择机为大家献上。

下一次，我们变换一下风格，由于我们接连搬了两座大山：迭代器和生成器，下次就一起研究下不会改变你编程风格的 ES6 特性好不？就是一些简单又实用的东西，你一定会喜笑颜开哒！你还别说，在什么都要“微”一下的今天，ES6 当然要有微改进了！

4

模板字符串

在上一章中，我说过要写一篇风格迥异的新文章，在了解了迭代器和生成器后，是时候来品味一些不烧脑的简单知识，如果你们觉得太难了，还不快去啃犀牛书！

现在，就让我们从最简单的知识学起吧！

反撇号（`）基础知识

ES6 引入了一种新型的字符串字面量语法，我们称之为模板字符串（template strings）。除了使用反撇号字符 ` 代替普通字符串的引号 ' 或 " 外，它们看起来与普通字符串并无二致。在最简单的情况下，它们与普通字符串的表现一致：

```
context.fillText(`Ceci n'est pas une chaîne.`, x, y);
```

但是我们并没有说：“原来只是被反撇号括起来的普通字符串啊”。模板字符串名之有理，它为 JavaScript 提供了简单的字符串插值功能，从此以后，你可以通过一种更加美观、更加方便的方式向字符串中插值了。

模板字符串的使用方式成千上万，但是最让我会心一暖的是将其应用于毫不起眼的错误消息提示：

```
function authorize(user, action) {
  if (!user.hasPrivilege(action)) {
    throw new Error(
      `用户 ${user.name} 未被授权执行 ${action} 操作。`);
  }
}
```

在这个示例中，${user.name}和${action}被称为模板占位符，JavaScript 将把 user.name 和 action 的值插入到最终生成的字符串中，例如：用户 jorendorff 未被授权打冰球。（这是真的，我还没有获得冰球许可证。）

到目前为止，我们所了解到的仅仅是比 + 运算符更优雅的语法，下面是你可能期待的一些特性细节：

模板占位符中的代码可以是任意 JavaScript 表达式，所以函数调用、算数运算等这些都可以作为占位符使用，你甚至可以在一个模板字符串中嵌套另一个，我称之为模板套构（template inception）。

如果这两个值都不是字符串，可以按照常规将其转换为字符串。例如：如果 action 是一个对象，将会调用它的.toString()方法将其转换为字符串值。

如果你需要在模板字符串中书写反撇号，你必须使用反斜杠将其转义：`\``等价于"`"。

同样地，如果你需要在模板字符串中引入字符$和{。无论你要实现什么样的目标，你都需要用反斜杠转义每一个字符：`\$`和`\{`。

与普通字符串不同的是，模板字符串可以多行书写：

```
$("#warning").html(`
  <h1>小心！>/h1>
  <p>未经授权打冰球可能受罚
  将近${maxPenalty}分钟。</p>
`);
```

模板字符串中所有的空格、新行、缩进，都会原样输出在生成的字符串中。

好啦，我说过要让你们轻松掌握模板字符串，从现在起难度会加大，你可以到此为止，去喝一杯咖啡，慢慢消化之前的知识。真的，及时回头不是一件令人感到羞愧的事情。Lopes Gonçalves 曾经向我们证明过，船只不会被海妖碾压，也不会从地球的边缘坠落下去，他最终跨越了赤道，但是他有继续探索整个南半球么？并没有，他回家了，

吃了一顿丰盛的午餐，你一定不排斥这样的感觉。

反撇号的未来

当然，模板字符串也并非事事包揽：

它们不会为你自动转义特殊字符，为了避免跨站脚本漏洞，你应当像拼接普通字符串时做的那样对非置信数据进行特殊处理。

它们无法很好地与国际化库（可以帮助你面向不同用户提供不同的语言）相配合，模板字符串不会格式化特定语言的数字和日期，更别提同时使用不同语言的情况了。

它们不能替代模板引擎的地位，例如：Mustache、Nunjucks。

模板字符串没有内建循环语法，所以你无法通过遍历数组来构建类似 HTML 中的表格，甚至它连条件语句都不支持。你当然可以使用模板套构（template inception）的方法实现，但在我看来这方法略显愚钝啊。

不过，ES6 为 JS 开发者和库设计者提供了一个很好的衍生工具，你可以借助这一特性突破模板字符串的诸多限制，我们称之为标签模板（tagged templates）。

标签模板的语法非常简单，在模板字符串开始的反撇号前附加一个额外的标签即可。我们的第一个示例将添加一个 SaferHTML 标签，我们要用这个标签来解决上述的第一个限制：自动转义特殊字符。

请注意，ES6 标准库不提供类似 SaferHTML 功能，我们将在下面自己来实现这个功能。

```
var message =
  SaferHTML`<p>${bonk.sender} 向你示好。</p>`;
```

这里用到的标签是一个标识符 SaferHTML；也可以使用属性值作为标签，例如：SaferHTML.escape；还可以是一个方法调用，例如：SaferHTML.escape({unicodeControlCharacters: false})。精确地说，任何 ES6 的成员表达

式（MemberExpression）或调用表达式（CallExpression）都可作为标签使用。

可以看出，无标签模板字符串简化了简单字符串拼接，标签模板则完全简化了函数调用！

上面的代码等效于：

```
var message =
  SaferHTML(templateData, bonk.sender);
```

templateData 是一个不可变数组，存储着模板所有的字符串部分，由 JS 引擎为我们创建。因为占位符将标签模板分割为两个字符串的部分，所以这个数组内含两个元素，形如 Object.freeze([" ", " has sent you a bonk. "]。

（事实上，templateData 中还有一个属性，在这篇文章中我们不会用到，但是它是标签模板不可分割的一环：templateData.raw，它同样是一个数组，存储着标签模板中所有的字符串部分，如果我们查看源码将会发现，在这里是使用形如\n 的转义序列分行，而在 templateData 中则为真正的新行，标准标签 String.raw 会用到这些原生字符串。）

如此一来，SaferHTML 函数就可以有成千上万种方法来解析字符串和占位符。

在继续阅读以前，可能你苦苦思索到底用 SaferHTML 来做什么，然后着手尝试去实现它，归根结底，它只是一个函数，你可以在 Firefox 的开发者控制台里测试你的成果。

以下是一种可行的方案（在 gist 中查看）：

```
function SaferHTML(templateData) {
  var s = templateData[0];
  for (var i = 1; i < arguments.length; i++) {
 var arg = String(arguments[i]);
 // 转义占位符中的特殊字符。
 s += arg.replace(/&/g, "&")
```

```
            .replace(/</g, "<")
            .replace(/</g, ">");
      // 不转义模板中的特殊字符。
      s += templateData[i];
    }
    return s;
  }
```

通过这样的定义，标签模板 SaferHTML`<p>${bonk.sender} 向你示好。</p>` 可能扩展为字符串 " ES6<3er 向你示好。 "。即使一个恶意命名的用户，例如“黑客 Steve<script>alert('xss');</script>”，向其他用户发送一条骚扰信息，无论如何这条信息都会被转义为普通字符串，其他用户不会受到潜在攻击的威胁。

（顺便一提，如果你感觉上述代码中在函数内部使用参数对象的方式令你感到枯燥乏味，不妨期待下一篇大作，ES6 中的另一个新特性一定会让你眼前一亮！）

仅一个简单的示例不足以说明标签模板的灵活性，我们一起回顾下我们之前有关模板字符串限制的列表，看一下你还能做些什么不一样的事情。

模板字符串不会自动转义特殊字符。但是正如我们看到的那样，通过标签模板，你可以自己写一个标签函数来解决这个问题。

事实上，你可以做的比那更好。

站在安全角度来说，我实现的 SaferHTML 函数相当脆弱，你需要通过多种不同的方式将 HTML 不同部分的特殊字符转义，SaferHTML 就无法做到全部转义。但是稍加努力，你就可以写出一个更加智能的 SaferHTML 函数，它可以针对 templateData 中字符串中的 HTML 位进行解析，分析出哪一个占位符是纯 HTML；哪一个是元素内部属性，需要转义'和"；哪一个是 URL 的 query 字符串，需要进行 URL 转义而非 HTML 转义，等等。智能 SaferHTML 函数可以将每个占位符都正确转义。

HTML 的解析速度很慢，这种方法听起来是否略显牵强？幸运的是，当模板重新

求值的时候标签模板的字符串部分是不改变的。SaferHTML 可以缓存所有的解析结果，来加速后续的调用。（缓存可以按照 ES6 的另一个特性——WeakMap 的形式进行存储，我们将在未来的文章中继续深入讨论。）

模板字符串没有内建的国际化特性，但是通过标签，我们可以添加这些功能。Jack Hsu 的一篇博客文章展示了具体的实现过程。我谨在此处抛砖引玉：

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

注意观察这个示例中的运行细节，name 和 amount 都是 JavaScript，进行正常插值处理，但是有一段与众不同的代码，:c(CAD)，Jack 将它放入了模板的字符串部分。JavaScript 理应由 JavaScript 引擎进行处理，字符串部分由 Jack 的 i18n 标签进行处理。使用者可以通过 i18n 的文档了解到，:c(CAD)代表加拿大元的货币单位。

这就是标签模板的大部分实际应用了。

模板字符串不能代替 Mustache 和 Nunjucks，一部分原因是在模板字符串没有内建的循环或条件语句语法。我们一起来看如何解决这个问题，如果 JS 不提供这个特性，我们就写一个标签来提供相应支持。

```
// 基于纯粹虚构的模板语言
// ES6 标签模板。
var libraryHtml = hashTemplate`
  <ul>
    #for book in ${myBooks}
      <li><i>#{book.title}</i> by #{book.author}</li>
    #end
  </ul>
`;
```