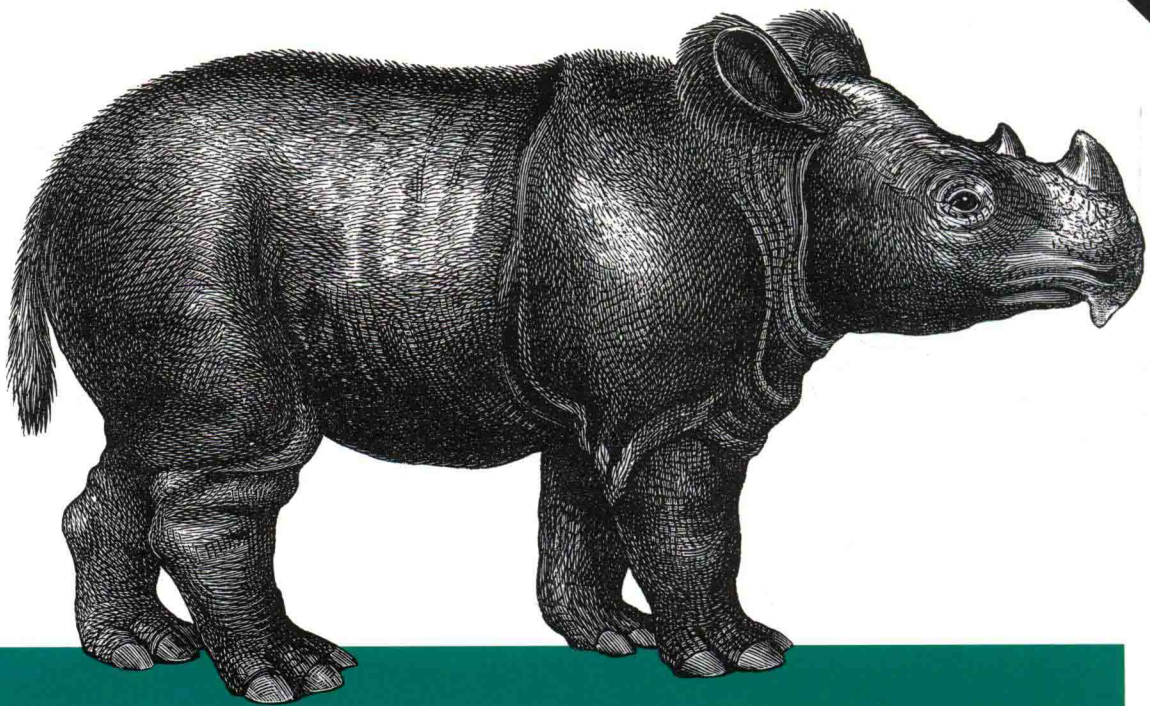
O'REILLY®

异步图书
www.epubit.com.cn

第3版



JavaScript 学习指南

Learning JavaScript　让网页变得栩栩如生的艺术

[美] Ethan Brown 著
娄佳 袁慎建 译

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

O'REILLY

JavaScript 学习指南（第 3 版）

[美] Ethan Brown 著

娄 佳
袁慎建 译

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

JavaScript学习指南 : 第3版 / (美) 布朗(Ethan Brown) 著 ; 娄佳, 袁慎建译. -- 北京 : 人民邮电出版社, 2017.7
ISBN 978-7-115-45632-8

Ⅰ. ①J… Ⅱ. ①布… ②娄… ③袁… Ⅲ. ①JAVA语言－程序设计－指南 Ⅳ. ①TP312.8-62

中国版本图书馆CIP数据核字(2017)第130679号

版权声明

◆ 著 [美]Ethan Brown
译 娄 佳 袁慎建
责任编辑 陈冀康
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：800×1000 1/16
印张：20.75
字数：385 千字 2017 年 7 月第 1 版
印数：1－2 400 册 2017 年 7 月河北第 1 次印刷

著作权合同登记号 图字：01-2016-7578 号

定价：59.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

内容提要

JavaScript 是目前 Web 开发领域非常流行的一种编程语言，得到众多 IT 从业人员和编程爱好者的关注。现在市面上的 JavaScript 图书多数基于 ES5，而本书重点关注 ES6。ES6 是语言的新标准，是目前业界超级活跃的计算机语言。

本书共分为 22 章，在 ES6 的基础上，本书系统地介绍了 JavaScript 的基本语法、语言特性、相关工具、基本对象、技术以及推动现代 JavaScript 开发的范例等方面的知识。其内容由浅及深，从变量、控制流、函数等内容到异步编程、正则表达式等。

本书适合具备一定 JavaScript 基础知识的读者阅读，也适合从事程序设计工作并想要深入探索 JavaScript 语言的读者阅读。

前言

虽然这已经是我在 JavaScript 技术领域的第二本书了，但对于 JavaScript 专家和布道师这个角色，我仍然觉得有些意外。和很多开发人员一样，我对 JavaScript 持有强烈的偏见，这种偏见一直持续到 2012 年。即便现在我的思想已经发生了转变，我的迷惑仍然存在。

我对 JavaScript 持有偏见的缘由很简单：我认为 JavaScript 是一种“玩具”语言（我并没有真正地好好学习过它，因此不是完全了解我所评价的东西），它经常被那些危险、粗心、未经培训的业余程序员所使用。这些现象都是有事实依据的。ES6 开发速度很快，即使是它的发明人 Brendan Eich 后来也承认有些东西在一开始并没有考虑周全，而当他意识到的时候，已经有太多人指望他能够对这些问题做出有效的改进（然而，又有哪个语言没有这方面的困扰呢）。第二个原因是，JavaScript 确实让编程突然变得更加容易上手。这不仅是因为浏览器的普及，更是因为它的高回报/投入比，人们只要付出一点点努力，就可以收获 JavaScript 为他们的网站所创造的巨大价值。人们可以通过试错，通过阅读彼此的代码，甚至在很多情况下，通过模仿那些缺乏理解的不良代码来学习这门编程语言。

我很庆幸自己在学习了足够多的 JavaScript 知识后，意识到它不仅不是一种玩具语言，而且拥有极其雄厚的基础，强大、灵活、具有表现力。我也很庆幸自己能够毫无芥蒂地拥抱 JavaScript 所带来的简易特性。毫无疑问，我对业余爱好者没有任何敌意：每个人都会找到自己的起点，而编程是一项有益的技能，以软件开发作为职业则会带来更多好处。

对于新手和业余的开发人员，我会说：成为一个业余开发人员并不是一件羞耻的事情。但如果一直停留在业余水平，那么就有点不合适了（如果你决定把编程作为职业）。如果你想练习编程，那么就付诸行动。搜集所有你能找到的资源，学习任何你可以学到的东西。保持谦虚、开放的心态，最重要的是，学会质疑。质疑一切权威，质疑任何一个经验丰富的开发人员，多问几句“为什么？”

多数情况下，我都尽量尝试让本书的内容符合 JavaScript 的“事实”，但是依然不可避免地会有不同的意见。当我提出不同意见时，按照原有的意思理解它们就行。

我非常欢迎不同的看法，我也鼓励读者去寻求其他有经验的开发人员的意见。

如今正是学习 JavaScript 最好的时代。互联网正在逐渐脱离蹒跚学步（从技术的角度上）的阶段，网站开发领域也不再处于 5～10 年前那种令人不解的拓荒阶段。一些类似 HTML5 或 ES6 之类的标准使得学习网站开发变得越来越简单，开发出高质量的网站应用也不再是一件困难的事情。Node.js 使 JavaScript 的应用范围不再局限于浏览器，现在的 JavaScript 已经可以用来开发系统脚本、桌面应用程序、网站后端，甚至是嵌入式应用。可以肯定的是，我从 20 世纪 80 年代中期开始编程以来，从未觉得编程是一件如此有趣的事。

JavaScript 简史

Brendan Eich 于 1995 年开发了 JavaScript，他曾就职于网景通信公司（Netscape Communications Corporation）。最早的 JavaScript 是在很短时间内开发出来的，很多批判它的人都说它的开发缺乏远见。然而，Brendan Eich 并不是一个浅尝辄止的人：他有着非常扎实的计算机科学基础，对 JavaScript 有着成熟和颇具先见之明的想法。从很多方面来说，这些想法是超越时代的，主流开发者们花了 15 年的时间才逐渐理解了这门语言的先进性。

JavaScript 最早的名字是 Mocha，在 1995 年网景领航员浏览器（Netscape Navigator）的一次发布中被正式命名为 JavaScript 之前，它曾经短期使用了 LiveScript 这个名字。“JavaScript”中的单词“Java”并不是巧合，但是却有点费解：撇开通用的语法传统不说，相比 Java，JavaScript 更类似于 Self（一种基于原型的语言，20 世纪 80 年代中期由 Xerox PARC 所开发）和 Scheme（一种深受 Lisp 和 ALGOL 影响的语言，20 世纪 70 年代由 Guy Steele 和 Gerald Sussman 所开发）。Eich 精通 Self 和 Scheme，他在开发 JavaScript 时应用了这两种语言中的一些具有前瞻性思考的范例。JavaScript 这个名字的由来一部分是出于市场需求，当时的 Java 语言非常流行，所以将它们关联起来[①]。

1996 年 11 月，网景通信公司声明他们已经把 JavaScript 提交到 Ecma。Ecma 是一家私有的、国际化非营利性标准化组织，它在技术和通信行业有着重大影响力。Ecma 国际发布了第一版 ECMA-26 规格，其本质就是 JavaScript。

Ecma 规格（一种叫作 ECMAScript 的语言）和 JavaScript 的区别更多体现在学术方面。从技术上说，JavaScript 是 ECMA 的一种实现，但在实际应用中，JavaScript 和 ECMAScript 是可以互换的。

① 在 2014 年的一次公开采访中，Eich 对“憎恨 JavaScript”的 Sun Microsystems 公司嗤之以鼻。

ECMAScript 的上一个主版本是 5.1（通常也称为 ES5），发布于 2011 年 6 月。市面上现存的老到不支持 ECMAScript5.1 的“非正规”浏览器已经不足个位数，可以很有把握地说，ECMAScript5.1 已经是现今的网络通用语言了。

ECMAScript6（ES6）作为本书的重点，由 Ecma 国际发布于 2015 年 6 月。在正式发布之前，它被叫作“Harmony”，所以你可能会听到有人把 ES6 称作“Harmony”“ES6 Harmony”“ES6”“ES2015”以及“ECMAScript2015”。在本书中，我们将其统称为 ES6。

ES6

如果说 ES5 是目前的网络通用语言的话，有心的读者可能会奇怪为什么本书要重点关注 ES6 呢。

ES6 代表了 JavaScript 语言的一个重大进步，ES5 的一些缺陷也在 ES6 中得到了改进。你一定会发现，在工作中，ES6 是一门更加令人愉快和强大的语言（从 ES5 开始入门则会很享受）。同时，感谢转换器，你现在可以编写 ES6 代码，然后把它们转化成“浏览器可识别的”ES5 代码。

随着 ES6 的最终发布，浏览器对它的支持也会越来越稳定，总有一天，转换器将从历史的舞台褪去（我不会预测这件事什么时候会发生，即使是粗略地估计）。

毫无疑问，ES6 代表了 JavaScript 开发的未来，你现在花些时间来学，就是为未来做准备，我们现在使用转换器，可以保留代码的可移植性。

然而，并不是每个开发人员都希望编写 ES6 代码。很可能你正在一个非常庞大的基于 ES5 的代码库上工作，将这些代码转换成 ES6 的成本非常高，一些开发人员仅仅是不想为此花费额外的精力。

除了第 1 章以外，本书将重点讲解 ES6，而非 ES5。我会在恰当的时机指出 ES6 和 ES5 的不同之处，但是不会出现对照代码的例子，或者在更适合使用 ES6 的时候出现大量关于“ES5 用法”的讨论。如果你恰好是那种无论如何都要坚持使用 ES5 的开发人员，那么本书可能不适合你（即使这样我还是希望未来的某一天你可以开始使用 ES6）。

从编纂的角度来说，选择 ES6 作为本书的重点这一决定非常谨慎。ES6 里程碑式的进步使得维持其清晰的教学框架变得非常困难。简而言之，如果一本书试图同时涵盖关于 ES5 和 ES6 的内容，很可能会顾此失彼。

本书的目标读者

本书主要针对那些有一定编程经验的读者（即使只听过一些介绍编程的课程，或者在线课程）。如果你刚接触编程，本书将会对你非常有帮助，不过你可能需要一些介绍性的文章或者课程作为补充。

那些已经有一些 JavaScript 经验（尤其是只有 ES5 的经验）的读者，也会在本书中找到实用且全面的重要语言概念。

从其他编程语言转过来的开发人员对本书的内容应该会有一见如故的感觉。

本书将全面涵盖关于语言特性、相关工具、技术，以及推动现代 JavaScript 开发的范例等方面的知识。因此，本书中的素材也将相应地从简明易懂（如变量、控制流、函数）到复杂深奥（如异步编程、正则表达式）变化。根据你的经验情况，你可能会发现阅读其中某些章节颇具挑战性。毫无疑问，初学者将需要反复阅读其中某些章节。

本书的遗憾之处

本书不是关于 JavaScript 或其相关类库的索引大全。Mozilla 开发者网络（Mozilla Developer Network，MDN）维护了一个非常出色、全面、实时更新，并且免费的在线 JavaScript 索引（*https://developer.mozilla.org/en-US/docs/Web/ JavaScript*），本书将会自由地引用以上类库。如果你更喜欢实体书，David Flanagan 的《JavaScript 权威指南》则更加全面（尽管它在本书编写的时候还未涵盖 ES6 的内容）。

本书的排版约定

本书将使用以下印刷相关约定：

斜体

表示新的术语、网址、邮件地址、文件名及文件扩展名。

等宽字体

用于代码列表，包括段落内引用代码元素的地方，比如，变量或者函数名、数据库、数据类型、环境变量、语句及关键字。

等宽加粗

表示命令行或者其他应该由用户输入的文本。

等宽斜体

表示应由用户输入或者由上下文决定的值。

此标志表示小窍门或者建议。

此标志表示一般性的注解。

此标志表示警告或警示。

Safari® Books Online

Safari Books Online 是一个按需数字图书馆，它提供了来自全球各地技术和商业领域优秀作者的专业书籍和视频。

专业技术人员、软件开发人员、网页设计人员，以及商业和创意专业人士等将 Safari Books Online 用作其首选的内容数据库，进行搜索、解决问题、学习和认证培训。

Safari Books Online 为企业、政府、教育机构和个人提供一系列计划和定价。

会员有权在这个完全可搜索的内容数据库中访问来自 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kauf mann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology 等上百家出版商的上千本书籍、培训影片，以及正式出版前的手稿。了解更多有关 Safari Books Online 的信息，请访问我们的官方网站。

我们的联系方式

请通过以下方式把对本书的评价和问题提交给出版商：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
电话：800-998-9938（美国或者加拿大）
707-829-0515（国际或者本地）
707-829-0104（传真）

关于本书的勘误、示例和其他信息，请访问官方页面。

关于本书的意见，建议或技术问题，请发邮件到：*bookquestions@oreilly.com*.

了解更多有关我们的图书、课程、会议的信息以及最新动态，请访问我们的官方推特。也可以在 Youtube 上观看。

致谢

能够有机会为 O'Reilly 写书是一个巨大的荣耀，我欠 Simon St.Laurent 一个大人情，因为他在我身上看到了这种潜能，并且带我走向写书这条路。Meg Foley 是我的编辑，他一直在支持我，鼓励我，对我有前所未有的帮助（我给你寄了一件非常绿的 T 恤衫，Meg!）。每一本 O'Reilly 出版的书都是一个团队努力的成果，我的文字编辑 Rachel Monaghan，制作编辑 Kristen Brown，以及校对员 Jasmine Kwityn，他们在跟我合作中都非常迅速，全面，富有见解：谢谢每个人的努力！

我的技术审稿人，Matt Inman、Shelley Powers、Nick Pinkham 和 Cody Lindley，感谢你们敏锐的反馈，绝妙的想法，以及不遗余力地帮我润色这本书。可以说没有你们我就无法完成这本书。当然，每个人的反馈都非常有帮助，这里我想特别感谢一下 Matt：他深厚的教育学背景为我提供了一个很有价值的教学视角，他经常在反馈中使用的 Stephen Colbert 的图片帮我理清思路！

在此要特别感谢 Shelley Powers——本书上一个版本的作者，不仅仅是因为她把这个头衔传递给我，更因为她为本书提供了非常有经验的反馈，从而让本书变得更好（并且引发了一些激烈的讨论）。

我想感谢所有读过我上一本书（《Node 与 Express 开发》）的读者。如果没有你们的购买，以及积极地评论，我可能就没有机会写本书了。在此特别感谢提出反馈和修正建议的读者：从你们的反馈中我学到了很多！

我很荣幸能在 Pop Art 工作，感谢这里的每一位同事，你们是我坚实的依靠。你们的支持令我自惭形秽，你们的热情激励着我，你们的专业精神和奉献精神是我每天

起床的动力。在这里特别感谢 Tom Paul：他坚定不移的原则，创新的经营理念，以及卓越的领导力时刻激励着我不仅要做好当下，还要放眼未来，不断进步。感谢 Steve Rosenbaum 创办 Pop Art，并在度过惊涛骇浪后将火炬成功地传递给 Tom。在我写作本书的过程中，Colwyn Fritze-Moor 和 Eric Buchmann 帮助处理了一些我的日常工作，感谢你们！感谢 Dylan Hallstrom 成为可靠的模范！感谢 Liz Tom 和 Sam Wilskey 加入 Pop Art 的团队！感谢 Carole Hardy、Nikki Brovold、Jennifer Erts、Randy Keener、Patrick Wu 和 Lisa Melogue 给我的所有支持。最后，感谢我的前辈 Tony Alferez、Paul Inman 和 Del Olds，从他们那里我学到了很多。

我对这本书以及编程语言的热情，是被 Dan Resler 博士点燃的，他是弗吉尼亚联邦大学的副教授。我在参加他的编译原理课程的时候缺乏兴趣，但是课程结束后却对形式语言理论产生了浓厚的兴趣。感谢您用自己的热情感染我，用您深邃的思想影响我。

感谢我所有在 PSU 中兼职 MBA 的朋友们，很高兴认识大家！特别感谢 Cathy、Amanda、Miska、Sahar、Paul S.、Cathy、John R.、Laurie、Joel、Tyler P.、Tyler S. 和 Jess：是你们让我的人生更加充实！

Pop Art 的同事们激励我变得杰出，点亮我的白天；朋友们激励我变得更有深度，照亮我的夜晚。Mark Booth：没有人比你更了解我，也没有人能像你这样获得我最深的信任。你的创造力和天赋让我感到惭愧，千万别把这本书拿到你面前当做炫耀的资本。Katy Roberts 像定期来临的潮水一样可靠，美丽的 Katy，感谢你深厚持久的善良和友谊。Sarah Lewis: 我喜欢你的样子。Byron 和 Amber Clayton 是我真诚而忠实的朋友，他们总是能带给我欢乐。Lorraine，这么多年过去了，你总是能让我展现出最好的一面。在这里我想对 Kate Nahas 说：很高兴多年之后我们恢复了联系；我很期待为我们拥有在杜克大学的共同回忆而举杯。Desember：感谢你的信任，温暖和友谊。最后，感谢我的新朋友 Chris Onstad 和 Jessica Rowe：你们在过去两年中给我的生活带来如此多的欢笑，真不知道没有你们我该怎么办。

在这里我想对我的母亲 Ann 说：感谢您给予我坚定不移的支持、爱和耐心。我的父亲 Tom，使我一直保持好奇心、创新和奉献精神，如果没有他我可能只是一个可怜的工程师（或者压根不是工程师）。还有我的姐姐，Meris，她将永远是我生命中那个代表着忠诚和信念的不动点。

目录

第1章

第一个应用

通常，实践是最佳的学习方式：所以接下来本书会从一个简单的应用开始。首先要申明的是，本章的重点不是解释接下来将发生的一切：因为肯定会有很多读者不熟悉或者困惑的地方。作者的建议是，尽量放轻松，不要试图去搞懂现在所发生的一切，以免让自己陷入泥团中。那么本章的重点是什么呢？没错，就是为了让读者进入兴奋状态。尽情放松地去享受接下来的一切吧，当你读完本书的时候，所有在本章中产生的困惑，都将烟消云散。

假如读者没有太多的编程经验，那么计算机死板的识别编程语言的能力会困扰到你。对于那些混乱的输入，人类大脑可以很容易地处理，而计算机却有点捉襟见肘。打个比方，如果作者犯了一个语法错误，读者可能只是觉得作者写作能力有待提高，却依然能理解我的意思。然而，JavaScript 跟其他所有编程语言一样，并不能这么灵活地处理混乱的输入。大小写、拼写、单词顺序，以及标点符号都很关键。如果大家也被这些问题困扰着，那么就要确保复制的信息都是正确的：不要用分号代替冒号或逗号，不要混淆单引号和双引号，所有代码的大小写都正确。一旦有了编程经验，开发人员就会明白什么地方可以“用自己的方式”来写代码，什么地方必须是严格按照语言的规范。但在此时，只要确保能准确地输入示例中的内容就可以避免这些困扰。

在过去，跟编程有关的书都习惯以一个叫作“Hello，World”的例子作为开始，虽然它只是简单地在终端打印“hello world”。有趣的是，这种做法是由 Bell 实验室的 Brian Kernighan 在 1972 开始的。它第一次出现在 1978 年出版的《C 语言程序设计》（《The C Programming Language》）里，此书由 Brian Kernighan 和 Dennis Ritchie 所著。直到今天，这本书仍然被广泛地认为是最好、最有影响力的编程语言书籍之

一。甚至在编写本书的时候，作者本人也从那部作品中获取了很多灵感。

现代的编程学习者越来越聪明了，“Hello，World”这个例子对于他们来说可能有些过时了。这个简单短语背后隐含的意义在今天看来，依然如同 1978 年那会儿有生命力：这是编程人员为其注入生命时说出的第一句话！它证明了编程人员可以像神话中的普罗米修斯一样，从太阳神阿波罗那里偷到火种，或者像弗兰肯斯坦博士一样，将生命注入到创作中。正是这种意义上的发明和创造，才将作者带入了编程世界。或许有一天，某些程序员（也可能是你），将会创造出某种形式的人工生命体，而它开口说的第一句话就是“hello world.”。

在本章，会在传统教条（44 年前由 Brian Kernighan 开创）与现代编程者日益增长的学习能力之间做好权衡。“hello world”同样会输出在屏幕上，只是它与 1972 年出现在荧光屏上的“hello world”已经大相径庭了。

1.1 从哪里开始

本书涵盖了 JavaScript 在它当前所涉及的领域（服务端、脚本、桌面、基于浏览器，等等）中的用法，但由于一些历史和现实的原因，本章选择从一个基于浏览器的程序开始。

为什么要选择基于浏览器的例子呢？一方面是因为它更容易访问图形库，从而获得良好的可视化效果。研究表明，人类本质上是视觉动物，将编程理念和视觉元素关联起来是一种很强大的学习方式。在学习过程中，因为要花大量时间来阅读这些文本文字，所以亟需一些视觉上充满趣味性的东西来润滑一下枯燥的文字。另一方面是因为它能有组织地介绍一些重要的概念，例如，事件驱动编程，这些概念会帮助更快地理解后面的章节。

1.2 实用的工具

没有一些实用工具的帮助，很难写出好的软件，就好比木匠没有锯子是很难做出一张像样的桌子。幸运的是，在本章，一个浏览器和一个文本编辑器就够用了。告诉大家一个好消息，在编写本书的时候，市面上所有浏览器都可以完成本书中要做的事情。即便是从未入过程序员法眼的 IE 浏览器也改过自新，向 Chrome、Firefox、Safari 和 Opera 这些浏览器看齐了。这里选择了 Firefox。本书也会讲解一些 Firefox 的特性，这些特性会在后续的编程之旅中助你一臂之力。当然，其他浏览器也有这些特性，本书会介绍这些特性在 Firefox 中的实现。所以说，倘若使用 Firefox 来完

成本章中的例子，将会度过一个更加轻松愉快的下午。

还需要一个文本编辑器去编写代码。文本编辑器的选择是一个容易引发争论（甚至是宗教性的）的主题了。大体上来讲，文本编辑器可以分为文本模式编辑器或窗口化编辑器。目前，两个最主流的文本模式的编辑器是 vi/vim 和 Emacs。它们最大的优点是，不仅能在本地环境使用它们，还可以通过 SSH 远程连接到别的机器上，然后使用这个已经非常熟悉的编辑器来编辑文件。窗口化边界器显得更时髦一些了，它们添加了一些有用（并且更加为用户所熟悉）的用户接口元素。然而，直到现在，也只需要编辑一些文本。所以说，窗口化编辑器此时并没有发挥出它固有的优势，而文本编辑器显得更加轻量和便捷。目前主流的窗口化本文编辑器有 Atom、Sublime Text、Coda、Visual Studio、Notepad++、TextPad，以及 Xcode。如果读者已经能很熟练地使用其中一种编辑器，那就没有必要更换了。但是，如果正在 Windows 系统中使用 Notepad，那么，强烈建议升级到一个更加强大的编辑器（Notepad++是一款适用于 Windows 用户并且免费易用的编辑器）

虽然全面描述编辑器的特性超出了本书的范围，但仍然有一些特性是需要去学习并掌握如何使用的。

1.2.1 语法高亮

语法高亮是使用颜色来区分程序中语法元素。比如，字符串可能是一种颜色，变量又是一种颜色（很快将了解这些术语的含义！），这个特性帮助开发人员更容易在代码中发现问题。大部分现代文本编辑器都默认有语法高亮功能；如果所使用代码不是彩色的，赶快查看编辑器的说明文档，了解如何启用它。

1.2.2 括号匹配

大多数编程语言都使用了大量的括号，它们有花括号、方括号（统称为括号）。有时候，这些括号的内容跨越了多行，有的甚至跨越了整个屏幕。除此之外，还会在括号中嵌套括号，而且通常是不同类型的括号。这个时候，括号匹配和“平衡（左右括号个数一致）”就显得至关重要了。否则，程序很难正确运行。另外，括号匹配在代码的起始位置起到了一个视觉提示的作用，它能够帮助发现括号不匹配的问题。在不同的编辑器中，括号匹配的处理方式也是不同的，有的只是细微的提示，有的却有非常明显的提示。在学习的过程中，括号不匹配通常会给初学者带来一些阻碍，所以，强烈建议读者花些时间学习掌握如何在编辑器中使用括号匹配功能。

1.2.3 代码折叠

另外一个与括号匹配相关的当属代码折叠。代码折叠功能能够临时隐藏不相关的代码，从而更加专注于核心代码。这个词源自于一个理念：将一张纸对折来隐藏不重要的细节。同样，类似于括号匹配功能，不同的编辑器对代码折叠的处理也是不同的。

1.2.4 自动补全

自动补全（也叫单词补全或者智能提示[①]）是一个很便捷的功能，它会试图在完成输入之前猜测想要输入的内容。这么做有两个好处。首先，它能帮助节省输入时间。比如说，不用完整输入 encodeURIComponent，只用输入 enc，然后就可以从自动补全的提示列表中选择 encodeURIComponent。第二个好处是它的探索性（discoverability）。比如，当输入 enc 时，本意是想使用 encodeURIComponent 函数，提示列表中还出现了 encodeURI 函数，这样就可以了解更多相关函数。甚至在某些编辑器中，还可能找到区分这两个选项的说明文档。JavaScript 是一门弱类型语言，它有自带的作用域规则（后面会学习它），所以在 JavaScript 中实现自动补全功能的难度要比其他语言大。如果编码时经常使用自动补全功能，花点时间去找一个拥有该功能的编辑器是很有必要的：很多编辑器提供了非常出色的自动补全功能。还有一些编辑器（比如 vim），虽然也提供强大的自动补全功能，但是需要做一些额外的配置。

1.3 关注注释

像很多编程语言一样，JavaScript 也有在代码中添加注释的语法。这些注释在执行时会被 JavaScript 所忽略，所有注释只对开发人员或者和开发人员一起工作的同事有意义。因为它允许使用自然语言解释含义不那么明确的代码。在本书中，会在代码实例中添加注释来解释该段代码。

在 JavaScript 中，有两种不同的注释：单行注释和多行注释。单行注释由两个连在一起的斜线开始（//），直到本行结束。多行注释由一组斜线和星号开始（/*），由另一组星号和斜线作为结束（*/），可以包含多行文字。以下的例子可以说明这两种注释的不同：

```
console.log("echo");          // prints "echo" to the console
/*
```

① 微软的术语。

```
    In the previous line, everything up to the double forward slashes
    is JavaScript code, and must be valid syntax.  The double
    forward slashes start a comment, and will be ignored by JavaScript.

    This text is in a block comment, and will also be ignored
    by JavaScript.  We've chosen to indent the comments of this block
    for readability, but that's not necessary.
*/
/*Look, Ma, no indentation!*/
```

在本书中，会时不时看到层叠样式表（CSS），CSS 中的注释跟 JavaScript 的多行注释语法一样（CSS 不支持单行注释）。HTML（类似 CSS）不支持单行注释，它的多行注释也与 JavaScript 的不一样。HTML 中的注释是被<!- -和- - >这种笨重的符号所包围的：

```
<head>
    <title>HTML and CSS Example</title>
    <!-- this is an HTML comment...
            which can span multiple lines. -->
    <style>
        body: { color: red; }
        /* this is a CSS comment...
            which can span multiple lines. */
    </style>
    <script>
        console.log("echo"); // back in JavaScript...
        /* ...so both inline and block comments
            are supported. */
    </script>
</head>
```

1.4 从这里开始

本书将从创建三个文件开始：一个 HTML 文件、一个 CSS 文件和一个 JavaScript 源文件。可以在 HTML 文件中完成所有事情，然后把 JavaScript 和 CSS 文件引入到 HTML 文件中，这样把内容分离到不同的文件中是有很大好处的。如果读者是一个编程新手，强烈建议你跟着这些说明一步一步来，在本章中将会采取逐步探索形式，这种形式会使学习过程更加便利。

就上例而言，好像为了完成简单的事情做了很多工作，这样做是有原因的。其实也可以用更简单的方式来实现这些，但是这样可能会教给你一些不好的编程习惯。在这里看到的多余步骤以后也会在其他地方重复看到，但是通过这样的步骤，至少可以确保自己学习到了正确的编程方法。

最后需要强调的是，本章是本书中唯一使用 ES5（而非 ES6）标准编写示例代码的一章，这也是本章最后一个重点所在。这样做是为了兼容那些尚且还不支持 ES6 的浏览器。在后续章节中，将会讲到如何使用 ES6 标准编写 JavaScript 代码，以及如何转换词法使其可以在传统浏览器上运行。之后本书的剩余部分都会使用 ES6 标准。本章的示例代码都非常简单，即使使用 ES5 标准来编写也不会有任何困难。

接下来的这个练习，必须确保创建的所有文件都在同一个目录中。建议给这个例子创建一个新目录或者新文件夹，从而确保它不会跟别的文件混在一起。

从 JavaScript 文件开始，使用文本编辑器创建一个叫作 main.js 的文件。现在只需要在文件中写入一行代码：

```
console.log('main.js')loaded;
```

然后创建一个 CSS 文件，main.css. 目前不用给这个文件中写入任何代码。

```
/*Style go here.*/
```

接下来创建一个叫作 index.html 的文件：

```
<!doctype html>
<html>
    <head>
        <link rel="stylesheet" href="main.css">
    </head>
    <body>
        <h1>My first application!</h1>
        <p>Welcome to <i>Learning JavaScript, 3rd Edition</i>.</p>

        <script src="main.js"></script>
    </body>
</html>
```

这本书不是一本讲 HTML 或者网站开发的书，但是很多人学 JavaScript 都是为了网站开发，所以也会提到一些跟 JavaScript 相关的 HTML 知识。一个 HTML 文档包含两个主要部分：文件头（head）和文件体（body）。文件头包含了一些不会直接显示在浏览器上的信息（但是它会影响那些显示在网页上的信息）。文件体包含了会被浏览器渲染的页面内容。要注意文件头中的内容永远不会显示在浏览器中，而文件体中的内容通常会显示（但是某些类型的元素不会显示的，例如<script>标签、CSS 样式这些内容都会隐藏在文件体中），理解这一点非常重要。

在文件头中有这样一行<link rel="stylesheet"href="main.css">；这可以把当前的空 CSS 文件加载到 HTML 文档中。文件体的结尾有<script src="main.js"></script>，这个则是为了加载 JavaScript 文件。大家是不是很好奇为什么把 css 文件引入放在文件开头而把 js 文件的引入放在文件末尾。当然也可以把<script>标签放在开头，但是考虑到性能和其他一些较为复杂的原因，选择把它放在文件末尾。

在文件体中有有这样一行<h1>My first application!</h1>，它定义了一级标题（表示页面中字体最大，最重要的文本），随后是<p>（段落）标签，包含了一些文本内容，其中有些内容是斜体（用<i>标签表示）

接下来在浏览器中打开 index.html 文件。在大多数操作系统中，最简单的做法就是双击这个文件（也可以把文件拖到浏览器窗口中）。这样文件中的内容就能显示在浏览器中了。

在本书中有很多代码示例。由于 HTML 和 JavaScript 文件通常都很庞大，所以不会每次将整个文件展示出来。不过不必太担心，作者会解释这些示例代码属于文件的哪一部分。这可能会让初学编程的人在理解代码结构时有些困难，但是弄清楚代码如何被整合在一起的是学习编程时非常关键的地方，而这也是不可避免的。

1.5 JavaScript 控制台

前面已经写过一些 JavaScript 的代码了，例如：console.log('main.js loaded')。这段代码做了什么呢？控制台是一个纯文本工具，开发人员用它来调试程序。在学习本书的过程中，会大量使用控制台。

打开控制台的方式因浏览器而异。鉴于会频繁用它，建议事先了解打开控制台的快捷键。在 Firefox 中，Ctrl-Shift-K（Windows 或者 Linux 系统）或 Command-Option-K（Mac 操作系统）就可以打开控制台。

在加载 index.html 文件的页面上打开 JavaScript 控制台，会看到控制台中输出了“main.js loaded”（如未输出，尝试刷新页面）。console.log 方法可以在控制台中打印任何内容[①]，这对调试和学习都很有帮助。

控制台有很多有用的功能，其中一个是不仅能看到程序的输出，还可以在控制台直

① 在第 9 章中会详细介绍 function 和 method 的不同之处。

接输入 JavaScript 代码，从而测试输出、学习 JavaScript 特性，甚至临时修改代码。

1.6 jQuery

将会在页面中引入一个非常流行的客户端脚本库 -- jQuery. 虽然这不是必须的，甚至与手头的任务关系不大，但在开发中，往往会最先将这个无处不在的类库引入到网页代码中。在这个例子中，即便没有引入 jQuery，也能很轻易地搞定，但是，越早开始习惯 jQuery 代码会越好。

在 HTML 文件 body 标签的最下面，在引入 main.js 之前，引入 jQuery：

```
<script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>

<script src="main.js"></script>
```

这里使用了一个网络链接，意味着如果网络不可用时，页面就不能正常工作了。从一个公共的内容分发网络（CDN：content delivery network）来加载 jQuery，这点有利于保证性能。如果在无网络的情况下工作，就必须把文件下载到本地，然后从本地加载文件。现在对 main.js 文件做些修改，使之用上一个 jQuery 的特性。

```
$(document).ready(function() {
    'use strict';
    console.log('main.js loaded');
});
```

如果没有使用 jQuery 的经验，上面这段代码看起来可能会有些费解。事实上，这里提到的很多东西在后面的章节中都有详细的讲解。在这里，jQuery 确保了所有的 HTML 文件都在 JavaScript 执行之前加载完成。(虽然现在 JavaScript 代码只有一句简单的 console.log)。每次使用基于浏览器的 JavaScript 时，都会将 JavaScript 代码写在$(document).ready(function(){和})；之间，从而帮助建立好的编程习惯。注意到'use strict'了吗，后面的章节里会详细介绍它，它会让 JavaScript 解释器更严格地对待所写的代码。乍听起来好像不那么友好，事实上它会帮助写出更好的 JavaScript 代码，也避免一些常见却又难以定位的问题。本书中的 JavaScript 都将严格遵循语法规范。

1.7 绘制图元

HTML5 提供了众多的好处，标准的图形化接口就是其一。HTML5 canvas 可以画出很多图元，例如，方形、圆形或多边形等基本图形。但是直接使用 canvas 可能会痛苦，所以会借用一个叫作 Paper.js 的图形库间接使用 canvas。

Paper.js 不是唯一可用的图形库：KineticJS，Fabric.js 和 EaselJS 都是当下流行且非常易用的替代品。这些类库作者都用过，它们都是很棒的图形库。

在使用 Paper.js 画图之前，需要一个可以用来画图的 HTML canvas 元素。在 HTML 文件 body 标签中加入以下代码（可以把这些代码放在任何位置，比如在介绍的段落后面）：

```
<canvas id="mainCanvas"></canvas>
```

注意，这里给 canvas 设置了 id 属性：这样就方便在 JavaScript 和 CSS 中引用它。此时刷新页面不会有任何变化，这是因为还没有在 canvas 上绘制任何东西，而且这个跟背景色一样的白色 canvas 连宽高都不存在，从而导致了它很难被区分。

每个 HTML 元素都可以有一个合法的 ID（格式正确），ID 必须是唯一的。在上面的代码中创建了一个 id 为 “mainCanvas” 的 canvas 元素，就意味着不能在其他元素上使用这个 ID。正因为如此，在编写 HTML 时不应过多的使用 ID。对于初学者来说，一次只考虑一件事情会更简单一些，所以在本例中使用 ID，根据定义，一个 ID 只能引用页面上的一个元素。

继续修改 main.css 文件，让 canvas 在页面上显示出来。即使不熟悉 CSS 也没关系，上述 CSS 代码只是简单地给元素设定了宽度和高度，并且设置了黑色边框：[①]

```
#mainCanvas {
   width: 400px;
   height: 400px;
   border: solid 1px black;
}
```

刷新页面，应该可以看到 canvas 了。

现在有了画图的载体，接下来要引入 Paper.js 帮助绘图。在引入 jQuery 和 main.js 的语句之间插入一行：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/paper.js/0.9.24/
paper-full.min.js"></script>
```

注意，跟引入 jQuery 的方式一样，也从 CDN 上加载 Paper.js。

① 如果你想学习更多关于 CSS 和 HTML 的内容，我推荐学习 Codecademy 上的免费 HTML 和 CSS 课程。

大家可能已经意识到，引入 js 文件的顺序是很重要的。因为需要在 main.js 中使用 jQuery 和 Paper.js，所以这两个文件必须在 main.js 之前被加载。而这两个文件互相之间没有依赖，所以谁先谁后并不重要，作者习惯最先引入 jQuery，因为在网页开发中的很多东西都依赖 jQuery。

代码中引入了 Paper.js 后，还需要对它做些配置。任何时候当遇到这样的重复代码--使用某个工具类库前的初始化设置--通常称为*样板代码*。将以下样板代码加入到 main.js 中“use strict”后面（如果愿意，可以删除 console.log）

```
paper.install(window);
paper.setup(document.getElementById('mainCanvas'));

// TODO

paper.view.draw();
```

第一行将 Paper.js 注册为一个全局变量（第七章中会深入讲）。第二行将 Paper.js 附在 canvas 上，同时准备绘图。在中间放置了 TODO，将从这里开始编写代码来完成那些有趣的事情。最后一行则使用 Paper.js 在屏幕上绘画。

所有的样板代码都就绪，就可以开始绘图了！下面要在 canvas 中间画一个绿色的圆。将“TODO”语句替换成以下代码：

```
var c = Shape.Circle(200, 200, 50);
c.fillColor = 'green';
```

刷新浏览器，会看到一个绿色的圆形。到这儿已经编写了第一行真实的 JavaScript 代码。上面两行代码包含了很多信息，不过现在只需要关注重要的信息。第一行代码通过三个参数创建了一个 circle 对象，参数分别是圆心的 x、y 坐标和半径。回忆一下创建 canvas 的时候给它指定的长和宽都是 400pixels，所以 canvas 中心的坐标是（200，200）。圆形的半径设为 50，恰好是 canvas 的长和宽的 1/8。第二行设置了填充颜色，是和外部截然不同的颜色（这种用法在 Paper.js 中称为 *stroke*）。可以任意改变这些参数看看结果会有什么不同。

1.8 自动执行重复任务

思考一种场景：相比于简单地给画布上添加一个圆，如果想将多个圆以网格布局填充画布，此时要怎么做呢？如果让圆与圆之间相距 50 个像素并设置略小的尺寸，就可以在画布上绘制出 64 个圆。当然，可以将同样的代码复制 63 次，然后手工修

改所有的坐标，让它们正确地显示在网格中。这听起来并非易事，还好计算机擅长这种重复的工作。来见证一下如何将 64 个均匀排列的圆绘制出来的。用下面代码替换掉前面绘制单个圆的代码：

```
var c;
for(var x=25; x<400; x+=50) {
    for(var y=25; y<400; y+=50) {
        c = Shape.Circle(x, y, 20);
        c.fillColor = 'green';
    }
}
```

刷新浏览器，64 个绿色的圆就会如期出现在大家的眼前！如果读者才开始接触编程，刚才那段代码可能会有些费解，但相信也会觉得它比手动复制修改的 128 行代码要好。

刚才的代码使用了 for 循环，它是控制流语句的一部分，关于控制流语句在第四章有详细的讲解。在 For 循环中，可以指定初始条件（25），结束条件（小于 400），以及增量值（50）。在循环中嵌套循环是为了同时在 x 轴和 y 轴两个方向上绘制圆。

针对这个例子，可以有很多不同的写法。上面那种写法是以 x, y 坐标作为一个重要的参考点：明确指定圆的开始位置，以及它们的间距。还可以从另一个方向去解决这个问题：只需要关注要绘制的圆的个数（64），然后让程序去计算怎样将它们均匀的填充在画布上。现在之所以采用第一种方式是因为它与要做的事情有更多的相似性，比如，将绘制圆的代码复制 64 次并自己计算出它们的间距。

1.9 处理用户输入

到目前为止，还没有涉及任何用户输入。用户点击圆不会有任何变化。同样地，拖拽圆也没有任何效果。接下来将设计一些用户交互，比如，让用户来决定从哪里开始画圆。

用户输入是具有异步特征的（时间不受控制），那么如何恰当并顺畅地处理它们显得尤为重要。异步事件有个特点：执行时间完全不受控制。用户点击鼠标就是一个异步事件：猜测不到用户在想什么，也不知道他们什么时候会点击。当然也可以给他们的点击做出提示，但这取决于他们何时点击或者是否真的点击了。用户输入引发的异步事件通常具有直观的表现，但在后面的章节中，所学习到的异步事件就不是那么的直观了。

Paper.js 使用一个 tool 对象来处理用户输入。如果觉得那个名字不是很表意，说明读者确实是一个优秀的程序员：必须承认，作者也不知道 Pager.js 的开发人员为什么使用这个术语[①]。那么为了更好地理解，不妨在意识里将“tool”转化为“user input tool”。使用下面的代码替换掉绘制网格布局的代码：

```
var tool = new Tool();

tool.onMouseDown = function(event) {
   var c = Shape.Circle(event.point.x, event.point.y, 20);
   c.fillColor = 'green';
};
```

这段代码的第一行创建了一个 tool 对象。有了 tool 对象之后，可以给它绑定一个事件处理器。这里给它绑定了一个叫 onMouseDown 的事件处理器。只要用户点击了鼠标，onMouseDown 事件处理器就会被调用，这是需要弄明白的一个关键点。之前编写的代码会立即执行：只要刷新页面，绿色的圆就会自动出现。但此时已经发生了变化：即便圆会显示出来，也是在单击画布后，而且每单击一次屏幕上只会出现一个绿色的圆。这是因为 function(event)紧跟的花括号中的代码有一个执行条件：用户在画布上单击了鼠标。

事件处理器做了两件事情：在用户点击鼠标的时候执行代码，以及告诉鼠标点击的位置。鼠标点击的位置信息存储在参数 event 的 point 属性中，event.point 有两个属性：x 和 y，它们能确定鼠标点击的位置。

其实还可以少写一些代码，直接将参数 event 的 point 属性传给圆（代替单独传入 x, y 坐标值）：

```
var c = Shape.Circle(event.point, 20);
```

上述代码突出了 JavaScript 一个非常重要特征：它能自动识别传入的变量。在之前的例子中，如果参数是三个数字，JavaScirpt 会把它们分别当做 x、y 坐标值和半径来处理。如果只有两个参数，JavaScript 就认为它们分别是 point 对象和半径。关于这点我们在第 6 章和第 8 章有更多的讲解。

1.10 Hello, World

最后用一个例子来结束本章，这个例子跟 Brian Kernighan 在 1972 年编写的例子具有相同的表现力。此时已经完成了所有的核心工作，剩下来要做的是添加文本。在

① 技术评论家 Matt Inman 提到，Page.js 开发者可能是 Photoshop 用户，他们熟悉“手工具”“直接选择工具”等。

onMouseDown 事件处理器之前插入下面的代码：

```
var c = Shape.Circle(200, 200, 80);
c.fillColor = 'black';
var text = new PointText(200, 200);
text.justification = 'center';
text.fillColor = 'white';
text.fontSize = 20;
text.content = 'hello world';
```

这段代码非常直观：创建另一个圆作为文本的背景，并且创建了一个真实的文本对象（PointText）。然后指定它的位置，以及一些额外的属性（对齐方式，颜色和字体尺寸）。最后，给它设置了真实的文字内容（“hello world”）。

注意：这里不是我们第一次使用 JavaScript 来展示文字了。第一次是在本章开始的 console.log 中打印文字。当然也可以将打印的文字改成“hello world.”，在许多方面，这更类似于在 1972 年可能会经历的事情（假如见证了 Brian Kernighan 在 1972 年编写了第一个 hello world）。但是就这个例子，文字本身及如何渲染它不是重点，重点是创造的东西已经产生了显著的效果。

刷新浏览器，此刻犹如在参加一个庄严神圣而传统的“Hello, World”盛宴。如果读者是第一次编写“Hello, World”，欢迎加入“Hello, World”俱乐部。如果不是，希望通过这个例子，读者能对 JavaScript 有一个初步的认识。

第 2 章

JavaScript 开发中的常用工具

虽然编写 JavaScript 代码只需要一个编辑器和一个浏览器（从上一章可以得知），但 JavaScript 开发人员往往需要借助一些实用开发工具来辅助开发。此外，由于本书后面的示例代码都将使用 ES6 的规范编写，所以还需要一个工具将 ES6 代码转换成常用的 ES5 代码。本章中介绍的工具都是些很常见的工具，而且它们经常被用在一些开源项目或者软件开发团队中。它们是：

- Git，版本控制工具，帮助管理日渐成长的项目，以及和其他开发人员的合作。
- Node，允许在除浏览器之外的地方运行 JavaScript 程序（还有它的搭档 npm，npm 可以访问本列表后面列出的工具）。
- Gulp，构建工具，它可以将开发过程中的一些常用任务自动化（Grunt 是另一个非常流行的构建工具）。
- Babel，转换编译器，可以将 ES6 的代码转换成便于使用的 ES5 代码。
- ESLint，一个可以帮助避免常见错误，并且使读者成为一个更称职的开发人员的格式检查工具。

本章介绍了一些 JavaScript 开发中非常重要的工具和技术，这些工具看似与 JavaScript 关系不大，实则它们都是非常有用的。

2.1 在当下编写 ES6 代码

首先要公布两个消息，一好一坏。好消息是 ES6（aka Harmony，aka Java-Script 2015）的出现实属 JavaScript 历史上一个激动人心的变革。坏消息是大家还没有为它的到

来做好充分的准备。但是这并不是说现在不能用它，只是它会给使用它的开发人员带来一些额外的负担，因为必须将 ES6 代码转换成“安全”的 ES5 代码，才能保证这些代码可以运行在任何浏览器上。

一些有经验的开发人员可能会觉得“这不是问题，看看我一天的工作，没有一个编程语言是不需要编译的!”作者本人已经从事软件开发工作很久了，所以对那些时刻需要编译的日子记忆犹新，但并不怀念那些时光。相反，很享受像 JavaScript[1] 这样的解释型语言所带来的轻便。[①]

JavaScript 的优点之一是它几乎无处不在：它几乎在一夜之间变成了浏览器的标准脚本语言，而随着 Node 的出现，JavaScript 的应用场景也不再局限于浏览器。所以意识到后面这一点会有点令人不快，因为可能要等到几年之后，才能随心所欲的编写 ES6 代码而不用担心浏览器是否支持。如果读者是一名 Node 开发者，情况会乐观一些，因为只是需要关注一个 JavaScript 引擎，而且还可以跟踪 Node 对 ES6 的支持状况。

本书中的 ES6 代码可以在 Firefox，以及一些类似 ES6 Fiddle 的网页中运行。不过对于那些“真实项目中的代码”，依旧要使用本章中介绍的工具和技术将 ES6 代码转成 ES5。

比较有意思的是 JavaScript 从 ES5 升级到 ES6 的过程是循序渐进的，这点与过去的编程语言版本升级不太一样。也就是说，目前使用的浏览器可能只支持一部分 ES6 特性。这种循序渐进的方式一方面由 JavaScript 原生的动态特性所决定，另一方面由浏览器本身升级所导致。大家可能听过常青树这个用来描述浏览器的术语，浏览器制造商也在逐渐取消浏览器不同版本之间必须通过升级来保持同步的理念。他们主张浏览器应该时刻保持最新版本，因为它们一直处于联网状态（至少如果它们希望获得最新版本的特性）。如今浏览器依然存在不同的版本，不过现在更有理由假设用户使用的都是最新的—因为常青树浏览器一定会自动替用户升级。

即使是常青树浏览器，也需要一些时间才能支持 ES6 的所有特性，所以目前我们还需要借助转换编译器（也叫转换器）。

2.2 ES6 的新特性

ES6 有太多新的特性，即便是即将提到的转换器在现阶段都不能全部支持。为了解

① 有一些 JavaScript 引擎（比如 Node）确实会编译 JavaScript，但这个过程对开发者是透明的。

决这个问题，纽约的开发者 kangax 维护了一个非常全面的兼容性表格，表格列出了 ES6（和 ES7）的所有特性。直到 2015 年 8 月，最完整的实现（Babel）也只覆盖了 72%的特性。这听起来可能有些沮丧，不过没关系，因为最重要的特性都已经优先被实现了，另外本书中提到的所有 ES6 特性都可以在 Babel 中使用。

在开始转换之前，还需要一些准备工作。首先要确保已经有了必要的工具，并且学会如何在一个新工程使用这些工具。大家会发现，尝试几次之后，这个过程就变成一个自然而然的过程。同时，在开始新项目的之前，可能需要回顾一下本章的内容。

2.2.1 安装 Git

如果还没有安装 Git，可以在 Git 官网的首页上找到符合操作系统的相关下载和介绍。

2.2.2 终端

本章中所有的练习都离不开*终端*（也叫*命令行*或*快捷命令*）。终端是一个基于文本的工具，它可以和计算机进行交互，开发人员基本都会用到它。即便不使用终端，也可以成为一个高效的程序员，但作者认为熟悉终端使用对于开发人员来说是一项很重要的技能：很多教程和书籍都会假设在使用它，并且很多工具也都是针对终端的使用而设计的。

目前业界最常用的终端工具叫作 bash，它被内嵌在 shell（终端的图形用户界面）中，是 Linux 或者 OS X 操作系统的默认终端。Windows 有自带的命令行，但这里还是推荐大家使用 Git（下一步就会安装它）提供的 bash 命令行工具。本书中将使用 bash。

Linux 和 OS X，在系统应用中就可以找到*终端*。而在 Windows 中，首先要安装 Git，然后在系统应用中找 Git Bash。

打开终端后，会看到一个*提示符*，这里就是输入命令的地方。默认提示符一般会包含电脑当前的用户名或者所在的文件目录，末尾一般会有一个美元符号（$）。因此，本章中的示例代码都会以一个美元符号作为提示符。在提示符后面的就是需要敲入到终端的命令了。比如，列出当前目录下的所有文件，在提示符后敲入 ls：

```
$ ls
```

在 Unix 系统的 bash 中，文件路径名由紧跟其后的斜线来划分（/）。

而在 Windows 中，目录通常使用反斜杠（\）。当使用 Git Bash 时，会把反斜杠转换成斜杠。Bash 通常使用波浪线（~）作为当前用户目录（一般用来存储文件）的

快捷键。

最基础的命令行是改变当前目录（cd）和添加新目录（mkdir）。比如，进用户目录：

```
$ cd ~
```

pwd（打印当前目录）命令可以打印当前所在的目录：

```
$ pwd
```

创建一个叫作 test 的子目录：

```
$ mkdir test
```

切换到新创建的目录：

```
$ cd test
```

两个点是“父目录”的快捷键（如果正在使用电脑练习前面的命令，那么以下命令会回到当前用户目录）

```
$ cd ..
```

关于终端，需要学习的知识还有很多，不过对于本章中的示例代码来说，掌握这些基础命令就足够了。如果读者还意犹未尽，推荐你学习 Treehouse 上的 Console Foundations 课程。

2.2.3 项目根目录

实际开发中，通常会给每个项目都创建一个目录。称这个目录为*项目根目录*。比如，如果正在练习本书中的例子，可能会创建一个名为 *lj* 的目录作为项目根目录。而对于书中任何有关命令行的例子，都假设是在项目根目录中执行那些命令。所以如果在练习的时候发现结果不对，首先检查当前工作目录是否是项目根目录。另外，创建文件的路径都是相对于根目录的。例如，如果项目根目录是*/home/joe/work/lj*，新创建一个文件 *public/js/test.js*，那么该文件的全路径应该是*/home/joe/work/lj/public/js/test.js*.

2.2.4 版本控制：Git

本书不会详细讲述版本控制工具，因为它是作为开发人员的必备技能。如果对 Git 不熟悉，那么本书和书中的例子就可以作为学习 Git 的良好材料。

首先，在项目的根目录下，初始化一个 Git 仓库：

```
$ git init
```

这条命令创建了一个项目仓库（在项目的根目录下会有一个隐藏起来的*.git* 文件）。

当然，一定会有一些文件是不想使用 Git 进行追踪的：构建包，临时文件等。这些文件可以在*.gitignore* 文件中被显示地排除掉。创建一个*.gitignore* 文件，添加以下内容：

```
# npm debugging logs
npm-debug.log*

# project dependencies
node_modules

# OSX folder attributes
.DS_Store

# temporary files
*.tmp
*~
```

如果还有其他不需要被追踪的“垃圾”文件，尽管把它们添加进来（比如：有些编辑器会创建*.bak* 的文件，通过**.bak* 可以将这些文件放进来）。

`git status` 是一个非常常用的命令，它可以显示仓库的当前状态。试着输入这个命令，看看结果是不是与下面的一样：

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to
track)
```

Git 会告诉程序员在当前目录下有一个新文件（*.gitignore*），但是它处于*未被追踪*的状态，这意味着 Git 并不会对它进行版本管理。这条信息很重要。

Git 的基本工作单元是*提交*。当前仓库还没有任何提交（因为这个仓库刚刚被初始化，虽然添加了一个文件，但是这个文件不被 Git 管理）。如果不特殊声明，Git 就不知道有哪些文件需要被追踪，所以必须把*.gitignore* 文件添加到仓库中：

```
$ git add .gitignore
```

现在还有没有进行任何提交；只是简单地把.gitignore 放入下一个提交中。运行

git status，可以看到：

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
```

现在.*gitignore* 的状态是待提交。到目前为止仍然没有提交，一旦提交，.gitignore 就会包含在里面。也可以添加更多文件，先来做一次提交：

```
$ git commit -m "Initial commit: added .gitignore."
```

这条命令中，-m 后面的是提交信息：简单描述本次提交中包含的修改。它可以保存项目的提交历史，方便后期回顾。

可以把提交想象成项目快照，它表示项目在某一时刻的状态。这次提交好比做了一次项目快照（只把.*gitignore* 放进来），在日后的任意时刻，都可以回顾这次提交。这时候运行 git status 看到的结果应该是：

```
On branch master
nothing to commit, working directory clean
```

接下来，多做一些修改。在目前的.*gitignore* 文件中，已经忽略了所有名为 npm-debug.log 的文件，这次试着忽略所有以 *.log* 结尾的文件（这是一个比较好的实践）。编辑.*gitignore* 文件，将 npm-debug.log 的那行修改为*.log。再添加一个叫作 README.md 的文件，这是一个用 *Marddown* 格式编写的文件，它是介绍项目的标准文件：

```
= Learning JavaScript, 3rd Edition
== Chapter 2: JavaScript Development Tools

In this chapter we're learning about Git and other
development tools
```

再试试 git status：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
```

```
            modified:   .gitignore
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md
```

现在已经有两处修改：一个是被 git 追踪的文件（*.gitignore*），另一个是新文件（*README.md*）。如果重复之前添加文件的步骤，那么应该运行的命令是：

```
$ git add .gitignore
$ git add README.md
```

不过这次尝试用快捷键一次性添加所有修改，然后做一次提交：

```
$ git add -A
$ git commit -m "Ignored all .log files and added README.md."
```

以上这两个步骤在后面的例子中会经常重复（添加修改的文件和提交这些修改）。在提交时，试着把它们做的小且逻辑性强一些：就好像这些提交是在讲述一个关于项目进度的故事。任何时候，只要在项目仓库中做修改，都要遵循这个步骤：添加修改的文件，然后提交：

```
$ git add -A
$ git commit -m "<brief description of the changes you just made>"
```

初学者经常会对 git add 有些疑惑；顾名思义，add 像是在给项目仓库中添加文件。有时候修改的内容确实是个新文件，不过这些新的修改早已经在代码库中完成了。换句话说，git add 的时候是在添加新的修改，而非文件（增加文件只是一种特殊类型的修改）。

以上内容展示了 Git 中最简单的工作流；如果想了解更多关于 Git 的知识，推荐 GitHub 上的 Git 教程，以及由 Jon Loeliger 和 Matthew McCullough 编写的《Git 版本控制》第二版。

2.2.5 包管理工具：npm

npm 对于 JavaScript 开发不是必须的，不过现在越来越多的前端开发人员都会选择 npm 作为项目的包管理工具。而对于 Node 开发，掌握 npm 是必不可少的。其实不管是 Node 开发还是基于浏览器的开发，npm 都可以让开发工作变得更简单。出于这个原因，本书中将会使用 npm 来安装开发所需的构建工具和转换器。

npm 一般会结合 Node 一起使用，如果还没安装 Node，访问 Node 官网（*https://nodejs.org/*），点击大绿色“INSTALL”按钮即可。

安装 Node 之后，需要验证 npm 和 Node 是否生效。在命令行输入以下命令：

```
$ node -v
v4.2.2
$ npm -v
2.14.7
```

机器上的版本号可能随着 Node 和 npm 的更新而变化。大体上讲，npm 是用来管理安装包的。这里说的包可以是一个功能齐全的应用程序中的任何东西，从示例代码，到一个功能模块，或者一个库，都可以使用 npm 来管理。

npm 支持两种级别的安装包：全局和本地。全局安装的包通常是一些用于开发中过程中的命令行工具，本地安装的包则是用于具体项目上的包。使用 npm install 命令就可以安装包。下面通过安装一个很常用的包 *Underscore* 来熟悉这个过程。切换到项目的根目录，运行下面命令：

```
$ npm install underscore
underscore@1.8.3 node_modules\underscore
```

这些信息表明 npm 已经安装了最新版本的 *Underscore*（1.8.3 是作者运行命令时的最新版本；大家在运行时版本号可能会有更新）。Underscore 这个功能模块没有其他依赖，所以 npm 的输出信息较简单；当安装一些复杂的功能模块时，npm 可能输出多达好几页的信息！如果想安装指定版本的 *Underscore*，可以显式指定版本号：

```
$ npm install underscore@1.8.0
underscore@1.8.0 node_modules\underscore
```

那么这个模块安装到哪里去了呢？安装结束后，在项目的根目录里会看到一个名为 node_*modules* 的目录，所有本地安装的模块都会放在这个目录里。可以删掉 *node_modules* 目录，稍后会重新创建它。

安装模块之后，就需要管理这些模块；这些已安装的模块就是项目的依赖。在项目日趋成熟的时候，需要以简明的方式获得当前项目的依赖信息，npm 通过一个名为 *package.json* 的文件来帮管理它们。这个文件不需要自己创建：在命令行里运行 npm init，然后回答几个关于配置的问题（最简单的办法是一路回车，使用默认的配置；之后可以随时修改文件内容）。运行 npm init 命令，看看生成的 *package.json* 文件有哪些信息。

依赖分为常规依赖和开发依赖。*开发依赖*是指那些只在项目构建时需要的依赖（稍后会有例子），应用程序运行时不需要它们。从现在开始，每安装一个本地依赖时，都需要在命令行后面添加--save 或者--saveDev 的标签；否则这些包虽然会被安装，但是不会出现在 *package.json* 文件里。接下来使用--save 标记重新安装 *Underscore*：

```
$ npm install --save underscore
npm WARN package.json lj@1.0.0 No description
npm WARN package.json lj@1.0.0 No repository field.
underscore@1.8.3 node_modules\underscore
```

这些警告是什么意思呢？这表示在准备安装的包里有一些组件找不到。由于本书不是专门讨论 npm 的书，所以可以暂时忽略这些警告。只有在使用 npm 公开自己的包的时候，才需要担心这些警告，但本书不会涉及这些内容。

此时 *package.json* 文件已经把 *Underscore* 加入到依赖列表中。依赖管理是因为那些被列在 *package.json* 里的有特定版本的依赖包需要被快速重建（下载和安装）。试着再删除 *node_modules* 目录，然后运行 npm install（这一次不用输入任何包名）。npm 就会下载所有在 *package.json* 文件里列出的包。看看新生成的 *node_modules* 目录，就知道下载的对不对了。

2.2.6 构建工具：Gulp 和 Grunt

大多数开发人员都会使用构建工具，可以在开发过程中自动化地运行一些重复任务。当下最火的两款的 JavaScript 构建工具分别是 Grunt 和 Gulp。它们都可以胜任系统构建的工作。Grunt 在几年前就已经为人们所熟知了，它比 Gulp 早一些出现，所以其社区也大一些，不过 Gulp 已经迎头赶上了。因为对于新的 JavaScript 开发人员来说，选择 Gulp 作为构建工具的比例正在迅速上升，本书中也会使用 Gulp。但这并不意味着我觉得 Gulp 比 Grunt 更高级（或者 Grunt 比 Gulp 更高级）。

首先，用以下命令全局安装 Gulp：

```
$ npm install -g gulp
```

如果读者在使用 Liunx 或者 OS X，可能需要在运行-g（全局）的时候切换到高一级的权限，使用：sudo npm install -g gulp。在输入密码后就会获得超级用户权限（只针对这一行命令）。如果在使用被其他人管理的系统，那就需要让管理员把你添加到 *sudoers* 的文件里。

对于一个操作系统，只需全局安装一次 Gulp 即可。而每个项目都需要本地的 Gulp，此时需要切换项目根目录下，运行 `npm install --save-dev gulp`（Gulp 只是开发依赖的一个例子。程序的运行并不依赖它，但是在开发过程中却需要它的帮助）。现在 Gulp 已经安装好了，接下来创建一个 *Gulpfile*（*gulpfile.js*）：

```
const gulp = require('gulp');
// Gulp dependencies go here
gulp.task('default', function() {
    // Gulp tasks go here
```

```
});
```

到目前为止，并没有给 gulp 配置任何任务，不过现在可以验证 gulp 是否能够正常运行：

```
$ gulp
[16:16:28] Using gulpfile~/home/joe/work/lj/gulpfile.js
[16:16:28] Starting 'default'...
[16:16:28] Finished 'default' after 68 µs
```

如果你是一个 Windows 用户，可能会看到这个错误“The build tools for Visual Studio 2010 (Platform Toolset = *v100*) cannot be found。”这是因为在 Windows 上很多 npm 的包都依赖于 Visual Studio 构建工具。可以从它的产品下载页面（*https://www.visualstudio.com/en-us/visual-studio- homepage-vs.aspx*）下载免费版的 Visual Studio。安装之后，在 program 文件下找到“开发人员命令提示符（Developer Command Prompt）”。在命令行快捷方式里，切换到项目的根目录，然后尝试再安装 Gulp，这一次应该会顺利很多。接下来并不需要一直使用 Visual Studio 的开发人员命令提示符，只是在安装对 Visual Studio 有依赖的 npm 的包时需要使用它，从而简化安装过程。

2.2.7 项目结构

使用 Gulp 和 Babel 将 ES6 代码转化成 ES5 之前，需要考虑代码应该放在项目中的什么位置。在 JavaScript 开发中，并没有一个世界通用的项目结构标准。相反，多样性在这个生态系统中体现得淋漓尽致。通常，项目的源代码会放在 *src* 或者 *js* 目录下。这里将把源代码放在 *es6* 这个目录下，体现出在使用 ES6 编写 JavaScript 代码。

因为大多数项目包含了服务器端（Node）代码和客户端（浏览器）代码，下面会在例子中把它们区分开来。服务器端代码放在根目录的 *es6* 目录下，而在浏览器端可以看到的代码放在 *public/es6* 目录下（默认情况下，所有被浏览器加载的代码都是 public 的，这是一个很常见的约定）。

在下一节，会把 ES6 代码转换成 ES5 代码，所以还需要一个地方来存放这些代码（不想把它们与 ES6 代码混在一起）。一个通用的约定是把它们放在一个叫作 *dist* 的目录下（是单词分布“distribution”的缩写）。

把所有代码放置好后，项目的根目录应该看起来是下面的结构：

```
.git                    # Git
```

```
.gitignore

package.json        # npm
node_modules

es6                   # Node source
dist

public/              # browser source
    es6/
    dist/
```

2.3 转换器

在编写此书时，两款最流行的转换器是 Babel 和 Traceur。这两个作者都用过，它们都可以满足需求并且简单易用。不过最近本人的学习方向渐渐转向了 Babel，本书中也将会使用 Babel 作为转换器。下面就开始吧！

Babel 最早是用于 ES5 到 ES6 的转换，随着时间的推移，Babel 逐渐成为一个可以转换多种格式的通用转换器，这其中包括 ES6，React，甚至是 ES7。从 Babel 6 开始，它就不再包含转换器了。为了完成 ES5 到 ES6 的转换，需要安装并配置 Babel 的 ES6 转换器。这些设置都会在本地项目中进行，由此可知会在当前项目中使用 ES6，而在别的项目中使用 React 或者 ES7（或者其他形式的 JavaScript）。首先，安装 ES6 的前置插件（aka ES2015）：

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

接下来在项目的根目录创建一个叫作*.babelrc*（这个文件默认会被隐藏起来）的文件。文件的内容是：

```
{ "presets": ["es2015"] }
```

有了这个文件，Babel 就可以识别出项目中所有使用 ES6 的地方了。

2.3.1 使用 Gulp 运行 Babel

现在可以使用 Gulp 做一些有意思的事情：把 ES6 代码转换成对应的 ES5 代码。接下来会将所有在 es6 文件夹或者 *public/es6* 文件夹下的代码转换成 ES5 代码，生成的代码分别放在 *dist* 和 *public/dist* 目录下。由于会使用一个叫作 `gulp-babel` 的包，所以首先要运行 `npm install --save-dev` 来安装它，接着在 *gulpfile.js* 里加入下面的内容：

```
const gulp = require('gulp');
```

```
const babel = require('gulp-babel');

gulp.task('default', function() {
  // Node source
gulp.src("es6/**/*.js")
  .pipe(babel())
  .pipe(gulp.dest("dist"));
// browser source
gulp.src("public/es6/**/*.js")
  .pipe(babel())
  .pipe(gulp.dest("public/dist"));
});
```

在这里 Gulp 使用了管道的概念。首先需要告诉 Gulp 要处理哪些文件：src（“es6/**/*.js”）。大家可能会问**是什么意思，它是一个通配符，表示“任何目录，包含子目录。”所以这里的数据源过滤器会解析 *es6* 文件夹下所有后缀为*.js* 的文件，包括所有子目录里的文件，而且不管文件的目录层次有多深都能找到。接下来，把这些文件传送给 Babel。最后一步是把已经转换的 ES5 代码输出到它的目标文件夹，也就是 dist 目录。Gulp 会保存源文件的文件名和目录结构。比如，*es6/a.js* 这个文件经过转换后会输出到 *dist/a.js*，而 *es6/a/b/c.js* 则会输出到 *dist/a/b/c.js*。同理，对于 *public/es6* 目录也会重复相同的过程。

到目前为止还没有真正地学习 ES6，不过可以先试着创建一个 ES6 文件来验证 Gulp 配置的正确性。创建 *es6/test.js* 文件，并写入以下可以展现 ES6 特性的代码。（如果读者还不理解这些代码，不用担心，看完这本书后就会明白了！）

```
'use strict';
// es6 特性：基于块作用域的"let" 声明
const sentences = [
    { subject: 'JavaScript', verb: 'is', object: 'great' },
    { subject: 'Elephants', verb: 'are', object: 'large' },
];
// es6 特性：对象解构
function say({ subject, verb, object }) {
    // es6 特性：模板字符串
    console.log('${subject} ${verb} ${object}');
}
// es6 特性：for..of
for(let s of sentences) {
    say(s);
}
```

接下来把这个文件复制到 *public/es6* 文件夹下（可以试着改变 sentances 数组中的内

容，以验证使用了不同文件）。然后在命令行窗口敲入 `gulp` 命令。执行完毕后，查看 *dist* 和 *public/dist* 目录。会发现里面各有一个 *test.js* 目录。如果仔细看那个文件就会发现它跟原始的 ES6 文件不一样。

下面试着直接运行 ES6 代码：

```
$ node es6/test.js
/home/ethan/lje3/es6/test.js:8
function say({ subject, verb, object }) {
             ^


SyntaxError: Unexpected token {
    at exports.runInThisContext (vm.js:53:16)
    at Module._compile (module.js:374:25)
    at Object.Module._extensions..js (module.js:417:10)
    at Module.load (module.js:344:32)
    at Function.Module._load (module.js:301:12)
    at Function.Module.runMain (module.js:442:10)
    at startup (node.js:136:18)
    at node.js:966:3
```

这个错误提示是 Node 输出的，大家得到的错误提示可能跟书中的不一样，这是因为 Node 还未完全实现 ES6 的特性（如果在足够远的未来读这本书，Node 可能已经完全实现了 ES6 的特性！）接下来试试运行 ES5 吧：

```
$ node dist\test.js
JavaScript is great
Elephants are large
```

至此已经成功将 ES6 代码转换成更轻量的 ES5 代码，这样它就能在任何地方运行了。最后将 *dist* 和 *public/dist* 添加到 *.gitignore* 文件中：因为想跟踪的是 ES6 源码，而不是生成的 ES5 代码。

2.3.2 格式审查工具

在参加很时尚的朋友聚会或者重要的面试之前，大家会用除毛滚筒来去除衣服上多余的毛球吗？大部分人会这样做，因为都希望将自己最好的一面展现出来。同理，也可以整理代码，让它（也就让用户）更好看些。整理器会严格审查用户的代码，然后告诉用户其中有哪些常见的错误。即使像作者这样拥有 25 年开发经验的人，整理器还是能在代码里找到错误。对于初学者来说，使用这个工具带来的好处是无与伦比的，它可以避免开发人员纠结于常见的代码错误。

目前有不少常见的 JavaScript 整理器，不过作者更喜欢由 Nicholas Zaka's 开发的

ESLint。安装 ESLint：

```
npm install -g eslint
```

在使用它之前，需要在项目中创建一个叫作.eslintrc 的配置文件。不同的项目会使用风格迥异的技术栈和编程规范，而.eslintrc 文件则可以在各种环境下都妥善处理代码。

创建*.eslintrc* 文件最简单的方法是运行 `eslint --init`，执行这个命令时需要回答一些问题，之后程序会自动创建一个默认文件。

在项目的根目录，运行 `eslint --init`。需要回答的问题如下。

- 你一般使用空格键还是制表键缩进？近期在 StackOverflow 上的投票结果显示大部分程序员更倾向于用制表键，但是具有多年开发经验的程序员却更喜欢用空格键。这里选择自己喜欢的方式就好。
- 你喜欢用单引号还是双引号定义字符串？其实这里选哪个都没关系，因为通常两个都会用到。
- 你在使用哪个终端（Unix 还是 Windows）？如果在使用 Linux 或者 OS X 系统，那么选择 Unix。如果在使用 Windows 系统，就选 Windows。
- 你需要在代码中使用分号吗？当然要。
- 你在使用 ECMAScript6（ES6）的特性吗？是的。
- 你会在哪里执行代码（在 Node 里还是浏览器）？通常情况下，在浏览器中运行的代码和在 Node 中运行的代码需要不同的配置，如果一定要选一个更高端的配置的话，那么就选 Node。
- 你想用 JSX 吗？不（JSX 是基于 XML 扩展的 JavaScript，在 Facebook 开发的 React UI 库中使用。本书中不会涉及。）
- 你希望配置文件使用哪种格式（JSON 还是 YAML）？选 JSON（YAML 是一个很流行的数据序列化格式，就像 JSON 一样，不过 JSON 跟适合 JavaScript 编程）。

所有问题回答完毕后，系统会生成一个.eslintrc 文件，接下来就可以使用 ESLint 了。ESLint 的运行方式有很多种。可以直接运行它（比如，运行 `eslint es6/test.js`），还可以把它集成到编辑器里，或者把它加到 Gulpfile 里。最好的方式

莫过于跟编辑器集成，但是不同的编辑器和操作系统所需的操作是不一样的。如果你想这样做，可以谷歌一下将“eslint”添加到编辑器的具体操作。

不管是否会将 ESlint 集成到编辑器中，作者都建议把 ESLint 加到 Gulpfile 里。毕竟，在构建项目时会运行 Gulp，而此时是一个检查代码的绝佳机会。首先运行：

```
npm install --save-dev gulp-eslint
```

然后修改 gulpfile.js：

```
const gulp = require('gulp');
const babel = require('gulp-babel');
const eslint = require('gulp-eslint');

gulp.task('default', function() {
  // 运行 ESLint
  gulp.src(["es6/**/*.js", "public/es6/**/*.js"])
    .pipe(eslint())
    .pipe(eslint.format());
  // Node 的资源
  gulp.src("es6/**/*.js")
    .pipe(babel())
    .pipe(gulp.dest("dist"));
  // 浏览器的资源
  gulp.src("public/es6/**/*.js")
    .pipe(babel())
    .pipe(gulp.dest("public/dist"));
});
```

现在来看看 ESLint 帮助找出了哪些错误。由于把 ESLint 加到了 Gulpfile 的默认任务里，所以可以直接运行 Gulp：

```
$ gulp
[15:04:16] Using gulpfile ~/git/gulpfile.js
[15:04:16] Starting 'default'...
[15:04:16] Finished 'default' after 84 ms
[15:04:16]
/home/ethan/lj/es6/test.js
  4:59  error  Unexpected trailing comma     comma-dangle
  9:5   error  Unexpected console statement  no-console

✖ 2 problems (2 errors, 0 warnings)
```

很显然，Nicholas Zakas 和作者一样都不喜欢行末的逗号。好在 ESLint 会让自己选择哪些是错误，哪些不是。默认行尾永远不能出现逗号，不过可以选择把这个功能整个关掉，或者改成“允许多个”（本人一般喜欢这样做）。编辑*.eslintrc* 文件来修

改一下设置(不过如果同意 Nicholas 的观点,可以继续使用默认的规则)。在*.eslintrc* 中的每一条规则都用一个数组表示。数组的第一行是数字,0 表示关闭规则,1 表示警告,2 表示错误:

```
{
   "rules": {
      /* changed comma-dangle default...ironically,
          we can't use a dangling comma here because
          this is a JSON file. */
      "comma-dangle": [
          2,
          "always-multiline"
      ],
      "indent": [
         2,
         4
      ],
      /* ... */
```

再次运行 `gulp`,逗号问题就不再是错误了。事实是,如果现在把逗号删掉,就会导致错误!

第二个错误与 `console.log` 的使用有关,如果该错误出现在产品代码上,通常会被当成“粗心大意”(如果使用传统浏览器甚至会埋下隐患)。然而出于学习的目的,可以禁用这个规则,因为本书中很多地方会使用到 console.log。同时,可能会想关闭那个“括号”的规则。这个就留给读者去练习。

ESLint 有很多配置选项;详情请查看 ESlint 官方网站。

现在可以编写 ES6 代码,并将其转换成 ES5 代码,然后对代码进行审查和优化,至此已经完成了 ES6 之旅的准备工作!

2.4 小结

通过本章的学习,我们已经了解到并不是所有浏览器都支持 ES6,不过这并不能阻止我们从 ES6 中获益,因为可以将使用 ES6 规范编写的代码转换成对应的 ES5 代码。

当开发人员配置一台新的开发机器时,需要以下这些工具:

- 一个好的编辑器(见第 1 章)。
- Git。

- Gulp (`npm install -g gulp`)。
- ESLint (`npm install -g eslint`) 。

每当开始一个新项目时（不管用来运行本书中示例代码的学习项目，还是一个真实的项目），都需要以下这些组件：

- 一个专门放置项目的目录，称之为*项目根目录*。
- Git 仓库（使用 `git init` 创建）。
- 一个 package.json 文件（使用 `npm init` 创建）。
- 一个 Gulpfile 文件（可以使用本章中的 *gulpfile.js*）。
- Gulp 和 Babel 的本地包（`npm install --save-dev gulp gulp-babel babel-preset-es2015`）。
- 一个 *.babelrc* 文件（contents: `{ "presets": ["es2015"] }`）。
- 一个 *.eslintrc* 文件（使用 `eslint --init` 创建，然后根据自己的喜好修改它）。
- 一个给 Node 使用的子目录（也就是例子中的 *es6*）。
- 一个给浏览器代码使用的子目录（即 public/es6）。

一旦准备好了所有东西，就可以开始了，基本的工作流程如下：

（1）编写代码，并修改相关逻辑。

（2）运行 Gulp 来测试程序，同时使用 ESLint 来规范代码。

（3）重复上述两个步骤直到改变生效，并且没有语法错误。

（4）使用 git status 检查当前代码状态，确保不会提交所有代码。如果有些不需要被 Git 管理的文件，记得把它们放到 *.gitignore* 文件里。

（5）把所有的修改添加到 Git 里（使用 git add -A；如果不准备添加所有文件，那就对每个需要添加的文件使用 git add）。

（6）提交你的修改（使用 `git commit -m “<本次修改的描述>”`）。

除上述步骤外还可能有其他步骤，这取决于实际项目。比如，有些项目需要运行测试（这通常会是一个 Gulp task），或者提交代码到一个用于分享的远端代码库（使

用 git push），例如，GitHub 或者 Bitbucket。不管怎样，这里列出的步骤可以用来构建一个很好的项目骨架。

本书的剩下部分将不再赘述上述步骤，只会展示源代码。除非特殊说明会浏览器运行，示例代码都会使用 Node 运行。打个比方，如果有一个叫作 *example.js* 的示例代码，需要把它放在 *es6* 文件夹下，然后运行：

```
$ gulp
$ node dist/example.js
```

也可以跳过运行 Gulp 的步骤，直接运行 `babel-node`（不过这并不会缩短运行时间，因为在运行 `bable-node` 时仍然需要编译）：

```
$ babel-node es6/example.js
```

接下来正式开始学习 JavaScript!

第3章

字面量、变量、常量和数据类型

本章将讲解数据，以及如何把数据转换成 JavaScript 可识别的格式。

大家可能已经知道，在计算机内部，所有的数据本质上是以 0 和 1 组成的长序列。但是对于大多数日常任务，则会以一种更自然的方式去思考数据，比如数字、文本、日期等。将它们统一叫作抽象数据类型。

在深入学习 JavaScript 的数据类型之前，先讨论变量、常量，以及字面量，这些都是 JavaScript 提供用来持有数据的机制。

在学习编程的时候，词汇的重要性通常会被忽视。虽然理解字面量和值的区别，或者声明和表达式的区别看上去似乎没那么重要，但是不知道这些术语有可能会妨碍后续的学习。其实，大多数术语并不是 JavaScript 特有的，它们在整个计算机科学中都很常见。理解这些概念术语很重要，多留心这些词汇也能够更容易切换编程语言，并拓展学习资源。

3.1 变量和常量

变量实质上是一个具名的值。顾名思义，变量的值是可变的。例如，在开发一个温度监控系统，系统中可能有个变量叫作 `currentTempC`：

```
let currentTempC = 22; // degrees Celsius
```

let 关键字是 ES6 新出的。在 ES6 之前，var 是唯一用来定义变量的关键字，第 7 章中有关于它的介绍。

这条语句做了两件事：声明（创建）变量 currentTempC 并给它指定了一个初始值。可以随时改变 currentTempC 的值。

```
currentTempC = 22.5;
```

注意，这里没有使用 let 关键字；let 仅用来声明变量，并且只能声明一次。

JavaScript 是不能给数字指定单位的。也就是说，不能限定变量 currentTempC 的单位为摄氏度。如果赋给 currentTempC 一个值，而该值是以华氏度换算出来的，程序就会发生异常（执行结果有误）。考虑到这一点，会给变量名加上后缀“C”来说明它的单位是摄氏度。语言本身并没有这种强约束，这么做其实是以文档的形式来避免一些不经意间的错误。

声明变量的时候也可以不指定初始值，此时变量会有一个特殊的默认值：undefined：

```
let targetTempC; //   等价于 "let targetTempC = undefined";
```

let 关键字还可以同时声明多个变量：

```
let targetTempC, room1 = "conference_room_a", room2 = "lobby";
```

上面的实例代码中声明了三个变量：targetTempC 没有初始值，所以它的值是 undefined，room1、room2 分别指定了初始值"conference_room_a"和"lobby"，它们都是字符串（文本）变量。

常量（ES6 的新特性）也可以存储值，但它与变量不同：常量一旦初始化就不能再改变。可以用常量来表示舒适的室内温度和最高温度（关键字 const 也可以声明多个常量）：

```
const ROOM_TEMP_C = 21.5, MAX_TEMP_C = 30;
```

有这样一个惯例（非强制）：凡是代表明确数字或字符串的常量名都应该由大写字母组成，并用下划线间隔多个单词。这么做是为了便于区分，同时也是种视觉提示：不应该试图改变它的值。

3.2 变量和常量：用哪个

一般情况下，应该优先使用常量。因为往往只是在努力地给某个数据取个好名字，而并非改变它的值。使用常量的优点是可以防止一些不应该更改的值被意外更改。例如，正在开发应用的某个模块，它的功能是执行一些用户操作，可能会使用变量 user。如果只有一个用户，但是 user 的值被修改了，那么程序很可能会发生异常。

如果同时有两个用户，可能会将他们分别命名为 `user1` 和 `user2`，而不是简单的复用变量 `user`。

经验法则告诉我们应该优先使用常量，一旦找到了修改这个常量的正当理由，再将它改成变量也不迟。

有一种情况必须用变量而非常量：for 循环（在第 4 章中详细讲解）。另外就是那些要随着时间推移而变化的值（比如，本章提到的 `targetTempC` 和 `currentTemp`）。一旦养成了优先使用常量的习惯，大家会很惊讶地发现变量其实很少被用到。

本书的例子中，会尽可能地使用常量。

3.3 标识符命名

变量和常量的名字（还有函数的名字，将在第 6 章讲解）统统称作标识符，它们有一些命名规则：

- 必须以字母、$、下划线（_）开头。
- 必须是由字母、数字、$和下划线（_）组成。
- 可以使用 Unicode 字符（例如，π 或者 ö）。
- 不可以使用保留字（参考附录 A）。

注意，$在某些语言中是一个特殊字符，但在 JavaScript 中它只是可以用在标识符中的普通字符（在诸多类库中，比如 jQuery 就很好利用了这一点，它使用$代表自身）。

保留字是一些因为容易混淆而不宜在 JavaScript 中定义为变量名的特殊关键字。例如，不能将一个变量命名为 `let`。

在 JavaScript 中，标识符没有独立的命名规范，不过以下两个是最常用的。

驼峰命名法

`currentTempC`、`anIdentifierName`（单词的首字母大写，看起来像骆驼背上的驼峰，所以叫驼峰命名法）

蛇型命名法

`current_temp_c`、`an_identifier_name`（流行度仅次于驼峰命名法）。

大家可以采用任意一种规范，但要保持一致性：选定一个后就坚持下去。不管是参与团队开发，还是将项目发布到社区，都尽量使用已经选定好的规范。

这里还有一些建议大家去遵守的规范：

- 标识符不应该以大写字母开头，但类名（在第 9 章中介绍）除外。
- 在大多数情况下，以 1～2 个下划线开始的标识符代表特殊变量或内部变量。除非需要创建专用的特殊类别的变量，否则就尽量避免使用以下划线开头的变量。
- 在 jQuery 中，以$开始的标识符一般特指 jQuery-wrapped 对象（详情见第 19 章）。

3.4 字面量

前面已经见过一些字面量：当给 `currentTempC` 赋值时，使用了一个数字字面量（初始值为 22，下一个例子中的值为 22.5）。同样，在变量 `room1` 初始化时指定了一个字符串字面量（`"conference_room_a"`）。单词 *literal* 指在程序中直接提供的值。本质上，字面量是一种创建值的方式，JavaScript 会使用开发人员提供的字面量去创建一个数据值。

理解字面量和标识符的区别尤为重要。回忆一下之前的例子，创建了一个变量 `room1`，它的值是`"conference_room_a"`，`room1` 是标识符（表示这是一个常量）。通过引号（数字不需要任何形式的引号，因为标识符不能以数字开头），JavaScript 可以区分标识符和字面量。看看下面这个例子：

```
let room1 = "conference_room_a";        // "conference_room_a"(引号内部)
                                        //是一个字面值

let currentRoom = room1;                // currentRoom 和 room1 有相同的值
                                        // ("conference_room_a")

let currentRoom = conference_room_a;    // 产生一个错误；因为不存在名为
                                        // conference_room_a 的标识符
```

任何能够使用标识符（需要给定一个值）的地方，都可以使用字面量。比如在程序中，出现 ROOM_TEMP_C 的地方都可以用字面量 22.5 来代替。如果代码中只有两处使用了该数字字面量还好，一旦有 10 处甚至 100 处都使用了该值，就应该使用常量或者变量代替了，这不但可以提高代码的可读性，而且当需要改变这个值的时候，只需要修改一个地方。

作为程序员，可以决定什么时候使用变量，什么时候使用常量。有些很明显属于常量的值，比如，近似值 π（圆周率：圆的周长直径比）或者 DAYS_IN_MARCH（三月的天数）。而其他像 ROOM_TEMP_C 这样的值就不是那么显而易见，也许有人觉得 21.5℃是最舒服的室内温度，但其他人不一定这么想。所以，如果这个值在应用中是可以配置的，变量就派上用场了。

3.5 基本类型和对象

在 JavaScript 中，只有基本类型和对象这两种值。基本类型（比如，字符串和数字）的值是不可变的。数字 5 始终是数字 5；字符串"alpha"也始终是"alpha"。数字不可变很好理解，但是很多人容易在字符串上犯错。比如，将多个字符串连接在一起（"alpha" +"omega"）后，以为只是在原来的字符串上做了修改。其实不然，连接后会生成一个新的字符串，就好比 6 和 5 不是同一个数字。后续会深入了解以下 6 种基本类型：

- 数字。
- 字符串。
- 布尔。
- null。
- undefined。
- 符号。

注意，不可变不是指变量的内容不可变，来看个例子：

```
let str = "hello";
str = "world";
```

第一个变量 str 初始化了一个不可变的值"hello"，紧接着它被指定了一个新的值

"world"（不可变）。这里关键点在于"hello"和"world"是不同的字符串，只是变量 str 所存储的值改变了。虽然很多时候，这种区别只是理论上的，但在第 6 章学习函数的时候这些知识就变得有用了。

除了上述的 6 种基本类型，剩下的都是对象。与基本类型不太一样的是，对象的形式和值非常多样化，犹如变色龙一般。

由于对象的灵活性，它可以用来构造自定义的数据类型。实际上，JavaScript 提供了一些内置的对象类型。这里将讨论以下几种内置对象：

- Array。
- Date。
- RegExp。
- Map 和 WeakMap。
- Set 和 WeakSet。

最后，基本类型的数字、字符串、布尔型都有对应的对象类型，Number、String、Boolean。这些对象不会真的存储一个值（基本类型则会存储值），它们只是具备关联到对应基本类型的功能。后面会结合这些对象类型对应的基本类型来学习它们。

3.6 数字

有些数字（如 3、5.5 和 1 000 000）可以用计算机精确地表示，而很多数字只能取近似值。比如，计算机不能将圆周率 π 完全表示出来，因为 π 的小数位是无限不循环的。其他一些数字，比如 1/3，可以用特殊的技术来表示，但是由于它们的小数部分是无限重复的（3.33333.....），所以通常会对它们取近似值。

JavaScript--跟大多数编程语言一样—有两种格式的近似值：IEEE-764 双精度和浮点型（从现在开始更倾向于简单地使用“double”）。关于这种格式的详细介绍超出了本书的范围，但是除非要做复杂的数字分析，否则不需要完全搞懂它（译者注：适当的“不求甚解”有时候是为了更好地专注自己的事情上）。然而，按照这种格式取近似值，其结果通常会造成困惑。比如，JavaScript 执行 0.1+0.2 的结果是 0.30000000000000004。看到这个结果，不要以为 JavaScript“崩溃”了或认为它不擅长算术，因为在有限的内存中给无限值取近似值，这是不可避免的结果。

JavaScript 是一门与众不同的编程语言，它只有一种数值数据类型[①]。而大多数编程语言有多种整数类型和 2 种及以上的浮点型。这样做，一方面简化了 JavaScript，尤其是针对初学者。另一方面，它也降低了 JavaScript 对那些对整数算法有性能要求，或者对固定精度的数字的精度有较高要求的应用的适应性。

JavaScript 能识别 4 种类型的数字字面量：十进制数、二进制数、八进制数和十六进制数。十进制数可以用来表示整数（不含小数位）、十进制数，以及以 10 为底的指数（科学计数法的缩写，例如 3.0e6）。此外，还有一些特殊值，比如，正负无穷大，以及 NaN（从技术上讲，它们不是数字，但会返回数值，所以将它们归类在一起）：

```
let count = 10;          // 即便赋的值是 integer；但 count 仍然是 double 类型
const blue = 0x0000ff;   // 十六进制数 (十六进制 ff 等于 十进制 255)
const umask = 0o0022;    // 八进制 (八进制 22 等于 十进制 18)
const roomTemp = 21.5;   // 十进制
const c = 3.0e6;         // 指数(3.0 × 10^6 = 3,000,000)
const e = -1.6e-19;      // 指数(-1.6 × 10^-19 = 0.00000000000000000016)
const inf = Infinity;
const ninf = -Infinity;
const nan = NaN;         // "不是数字"
```

无论是用什么格式的字面量（十进制、十六进制、指数，等等），创建的数字都是以一种特定格式存储的：双精度。利用字面量的多种不同格式，可以按照任意格式来指定数字，但如果想以不同的格式展示数字，JavaScript 就有点捉襟见肘了，这点会在第 16 章讨论。

绝大多数数学家可能认为上述观点是不恰当的：无穷大不是数字，事实上它的确不是；当然，NaN 也不是。它们不能代替数字去做运算。但是，它们可以用作占位符。

此外，数字对应的 Number 对象还有很多实用的属性，这些属性代表着一些重要的数值。

```
const small = Number.EPSILON;             // 可以被添加到 1 从而获取一个比 1 大的
                                          // 不同数字，它的值近似于 2.2e-16
const bigInt = Number.MAX_SAFE_INTEGER;   // 最大可表示的整数
const max = Number.MAX_VALUE;             // 最大可表示的数字
const minInt = Number.MIN_SAFE_INTEGER;   // 最小可表示的整数
const min = Number.MIN_VALUE;             // 最小可表示的数字
const nInf = Number.NEGATIVE_INFINITY;    // -Infinity
```

① 这点在未来可能发生改变：专用的整数类型是一个经常被讨论的语言特征。

```
const nan= Number.NaN;                    // the same as Nan
const inf= Number.POSIT IVE_INFINITY;     // the same as Infinity
```

我们会在第 16 章讨论这些值的重要性。

3.7 字符串

字符串是简单的文本数据（单词 string 来自于“string of characters”--string 最早是在 18 世纪末被排字工人用到，后来一些数学家也沿用了它，用来表示具有一定顺序的符号序列。）

字符串在 JavaScript 中表示 Unicode 文本。Unicode 是一个计算机行业标准，用来表示文本数据，它涵盖了字符的字符码，以及大多数人类语言中的符号（甚至包含了让人惊讶的“语言”，比如说 Emoji）。Unicode 可以在任何语言中表示文本，但这不意味着能识别 Unicode 的软件也能够正确的识别每一个字符码。本书中，会坚持使用一些很常见的 Unicode 字符，它们几乎都能在浏览器和控制台中可用。如果遇到独特的字符或者语言文字，可能需要针对 Unicode 做一些额外的学习研究，弄明白怎么识别字符码。

在 JavaScript 中，单引号、双引号或者重音符[①]都可以表示字符串字面量。重音符是在 ES6 引入的，它是为了启用模板字符串，这些内容马上会讲到。

转义

当尝试在一段由文本写成的程序中表示文本数据时，如何区分文本数据和程序本身将会是一个问题。可以给字符串加上引号，但如果字符串本身也包含引号时会怎么样呢？要解决这个问题，需要对特殊字符进行转义，转义之后它们就不会被作为字符串终止符（"或'）处理了 。来看一个例子（不需要转义）

```
const dialog = 'Sam looked up, and said "hello, old friend!", as Max
walked in.';
const imperative = "Don't do that!";
```

在常量 dialog 中，因为采用了单引号的赋值方式，所以可以在内部放心地使用双引号。同样，在 imperative 中，双引号的赋值方式允许在其内部使用单引号。但如果在内部嵌套使用双引号会怎么样呢？比如下面的例子：

```
// 这行代码会产生错误
const dialog = "Sam looked up and said "don't do that!" to Max.";
```

① 也叫重音符。

不管使用单引号还是双引号，字符串常量 dialog 都会报错。不过还好，可以使用反斜杠（\）对引号进行转义。反斜杠作为一个信号告诉 JavaScript 该字符串并没有结束。下面使用两种类型的引号来重写之前的列子：

```
const dialog1 = "He looked up and said \"don't do that!\" to Max.";
const dialog2 = 'He looked up and said "don\'t do that!" to Max.';
```

当在字符串中使用反斜线时，就会遇到“先有鸡还是先有蛋”的问题。要解决这个问题，可以在反斜线前面再加一个反斜线：

```
const s = "In JavaScript, use \\ as an escape character in strings.";
```

至于采用单引号还是双引号完全取决于个人的喜好。针对一些提供给用户阅读的文字信息，因为缩写（像 *don't*）出现的频率较多，作者会倾向于使用双引号。而当我在 JavaScritp 字符串中插入 HTML 脚本时，则更喜欢使用单引号，这样就可以使用双引号表示 HTML 标签相关属性的值。

3.8 特殊字符

除了转义引号，反斜线还有很多用途：它可以表示某些不可打印的字符，例如，换行和任意的 Unicode 字符。表 3-1 列出一些常用的特殊字符。

表 3-1 常用的特殊字符

字符	说　明	示　例
\n	换行（技术层面的换行：ASCII/Unicode 10）	"Line1\nLine2"
\r	回车（ASCII/Unicode 13）	"Windows line 1\r\nWindows line 2"
\t	制表符（ASCII/Unicode 9）	"Speed:\t60kph"
\'	单引号（注意，即使没有必要，你也可以用它）	"Don\'t"
\"	双引号（注意，即使没有必要，你也可以用它）	'Sam said \"hello\".'
\`	重音符（ES6 的新特性）	'New in ES6: \` strings. '
\$	美元符（ES6 的新特性）	'New in ES6: ${interpolation}'
\\	反斜线	"Use \\\\ to represent \\!"
\u*XXXX*	任意的 Unicode 码（*XXXX* 表示一个 16 进制的字符码）	"De Morgàn's law: \u2310(P \u22c0 Q) \u21D4 (\u2310P) \u22c1 (\u2310Q)"
\x*XX*	Latin1 字符（*XX* 表示一个十六进制的 Latin1 字符码）	"\xc9p\xe9e is fun, but foil is more fun."

注意，Latin-1 字符集是 Unicode 的一个子集，任何 Latin-1 字符\xXX 都可以使用相同的 Unicode 字符码\u00XX 来表示。对于十六进制数，字母大小写没有限制；作者个人更喜欢小写字母，因为可读性更好。

Unicode 字符是不需要进行转义的，可以直接在编辑器中输入。不同的编辑器和操作系统（通常不止一种），对 Unicode 字符的处理方式也不相同，所以，在编辑器输入 Unicode 字符前，需要查看编辑器和操作系统的相关文档，从而正确地使用它们。

此外，表 3-2 列出了一些不常用的特殊字符。在作者的印象中，从来没有在 JavaScript 代码中使用过它们，但为了完整起见，还是把它们一并列了出来。

表 3-2　　不常用的特殊字符

字符	说　明	示　例
\0	NUL 字符（ASCII/Unicode 0）	"ASCII NUL: \0"
\v	垂直制表符（ASCII/Unicode 11）	"Vertical tab: \v"
\b	退格（ASCII/Unicode 8）	"Backspace: \b"
\f	分页（ASCII/Unicode 12）	"Form feed: \f"

3.8.1　模板字符串

在字符串表示数值是一种很常见的需求。可以通过字符串连接机制来完成：

```
let currentTemp = 19.5;
// 00b0 是"度数"符号的 Unicode
const message = "The current temperature is " + currentTemp + "\u00b0C";
```

在 ES6 之前，字符串连接是唯一的实现方式（如果不用第三方库）。ES6 引进了字符串模板（又叫字符串插值）。字符串模板提供了一种往字符串中快速插入值的方式。它使用重音符（`）代替引号。将之前的例子使用字符串模板进行重写：

```
let currentTemp = 19.5;
const message = 'The current temperature is ${currentTemp}\u00b0C';
```

在一个字符串模板中，$符号是特殊字符（也可以使用反斜线转义它）：如果紧跟$的值[①]被包裹在大括号中，那么该值就会被注入到字符串中。

在 ES6 中，模板字符串是作者最喜欢的特性之一，大家会看到本书通篇都在使用它。

① 你可以在花括号内部使用任意*表达式*。第 5 章会讲到。

3.8.2 多行字符串

在 ES6 之前，JavaScript 对多行字符的支持勉强说得过去。单从语言规范上讲，JavaScript 是允许在源代码行末对换行符进行转移的，但由于浏览器的兼容性问题，作者从来没有用过这个特性。ES6 发布以后，这个特性就变得可用了，但还是有一些特殊的地方需要知道。注意，这些技术很可能在 JavaScript 控制台中不可用（比如在浏览器控制台中），所以必须把它们写在 JavaScript 文件去使用。对于由单引号和双引号引用起来的字符串，可以转义换行符：

```
const multiline = "line1\
line2";
```

如果以为 multiline 里面会另起一行，那么可能会很吃惊，因为行末的斜杠将换行符转义了，所以这里并没有出现新的一行。最终的结果将是"line1line2"。如果想要另起一行，可以这么做：

```
const multiline = "line1\n\
line2";
```

重音符字符串也可以基本满足期望：

```
const multiline = 'line1
line2';
```

此时字符串就会换行了。然而，这两种方式都会将缩进包含在字符串中。例如，在以下字符串中，line2 和 line3 之前除了换行还存在空格，而这可能不是我们想要的结果：

```
const multiline = 'line1
    line2
    line3';
```

由于这个原因，通常会避免使用多行字符串的语法。如果使用它，要么选择放弃缩进来提高代码的可读性，要么在多行字符串中插入可能不想要的空格。如果确实要表示多行字符串，通常使用字符串连接：

```
const multiline = "line1\n" +
    "line2\n" +
    "line3";
```

这样缩进代码不但能提高代码的可读性，而且还能得到想要的结果。注意，字符串连接机制允许混合及匹配多种不同类型的字符串：

```
const multiline = 'Current temperature:\n' +
    '\t${currentTemp}\u00b0C\n' +
```

```
"Don't worry...the heat is on!";
```

3.8.3 数字用作字符串

如果给一个数字加上引号，它就不再是数字--而是一个字符串。也就是说，必要的时候，JavaScript 会自动将包含数字的字符串转换成数字。这种转换何时发生，以及如何发生，可能会使人感到困惑，这些都会在第 5 章中讲到。下面用一个例子说明什么时候会发生这种转换，以及什么时候不会：

```
const result1 = 3 + '30';  // 3 被转换成字符串；结果为字符串 '330'
const result2 = 3 * '30';  // '30' 被转换成数字；结果为数值 90
```

理论上讲，当需要数字时，就使用数字（没有引号），当需要字符串时，就使用字符串。但实际情况不总是那么泾渭分明，比如在需要用户输入的地方，用户通常会输入字符串，这就需要在合适的情况下将其转换为数字。本章的后续内容中，会讨论如何在不同的数据类型之间做转换。

3.9 布尔型

布尔型只会出现两种值：`true` 和 `false`。一些编程语言（比如 C 语言）用数字代替布尔值：0 代表 `false`，其他非 0 的数字都代表 true。JavaScript 也有类似的机制，它允许使用任何值（不仅仅是数字）来代表“正确”或“错误”，这些将在第 5 章中详细讲解。

使用布尔值时要格外当心不能使用引号。特别是当很多人看到字符串`"false"`返回结果是 `true` 时，他们可能会“尖叫”起来。下面是布尔值的正确表示方法：

```
let heating = true;
let cooling = false;
```

3.10 符号

符号是 ES6 的新特性：它是一种新的数据类型，代表一个唯一的标志。符号一经创建就是独一无二的：它不会匹配其他任何符号。这样看来，符号就有点类似对象（每个对象都是唯一的）。然而，从其他所有方面来看，符号是一种基本类型，它所具备的可扩展性使其成为一种非常有用的编程语言特性，关于这点会在第 9 章中详述。

Symbol()构造方法[①]可以用来创建符号。为方便起见，还可以在构造方法中传入一些描述信息。

```
const RED = Symbol();
const ORANGE = Symbol("The color of a sunset!");
RED === ORANGE          // false: 每个符号都是唯一的
```

如果想创建一个唯一的标识符，但又不想无意中跟其他标识符混淆，建议使用符号。

3.11 null 和 undefined

JavaScript 有两种特殊的类型，null 和 undefined，它们两个都只有一个唯一的值，分别是 null 和 undefined。这两者都表示不存在。实际上存在两种独特的数据类型就已经造成了很大的困惑，尤其是对于初学者。

一般的经验是，null 是给开发者用的，而 undefined 则是留给 JavaScript 用的，用来表示未赋值的内容。这并不是强制的规则，开发人员也可以随时使用 undefined。但常识表明，应该非常谨慎地使用它。在以往的经验中，只有在有意的模仿变量未被赋值的时候，才会使用 undefined。当需要表示一个变量的值未知或者不适用的时候，常见的做法是使用 null。这似乎有点小题大做，但这确实很重要--建议编程新手在不确定该使用哪一个的时候使用 null。注意，如果声明变量的时候没有赋值，变量会有一个默认的值 undefined。下面例子使用了 null 和 undefind 的字面量：

```
let currentTemp;              // 隐含值 undefined
const targetTemp = null;      // targetTemp 为 null -- "还不知道"
currentTemp = 19.5;           // currentTemp 此时已经有值
currentTemp = undefined;      // currentTemp 看上去跟未初始化一样，不推荐这么做
```

3.12 对象

对象跟不可变的基本类型不太一样，基本类型只能代表一个值，而对象可以代表多个值或者复杂的值，而且这个值在其生命周期内都是可变的。本质上，对象是一个容器，容器的内容可以随着时间推移而改变（也就是同一个对象拥有不同的内容）。跟基本类型一样，对象也有自己语法：大括号（{}）。因为大括号是成对出现的，

① 如果你已经熟悉了 JavaScript 中面向对象编程，需要注意的是，你不能使用关键字 new 创建一个符号，这是 JavaScript 约定中的一个例外。因为约定指出，以大写字母开头的标识符应该和 new 一起使用。

所以可以在大括号内填充对象的内容。下面从一个空对象开始：

```
const obj = {};
```

可以给对象任意命名，通常建议取一个表意的名字，比如，user 或 shoppingCart。这里只是为了学习对象的原理，例子中的对象没有任何特殊的意义，所以就简单称之为 obj。

对象的内容称作属性（或成员），属性是由名称（或键）和值组成。属性名必须是字符串或者符号，值可以是任意类型（包括其他对象）。接下来给 obj 添加一个 color 属性：

```
obj.size;              // undefined
obj.color;             // "yellow"
```

为了使用成员访问运算符，属性名必须是一个合法的标识符。如果想使用非法的标识符作为属性名，必须使用计算机成员访问符（它对合法的标识符也起作用）：

```
obj["not an identifier"] = 3;
obj["not an identifier"];          // 3
obj["color"];                      // "yellow"
```

也可以使用计算机成员访问操作符来访问符号属性：

```
const SIZE = Symbol();
obj[SIZE] = 8;
obj[SIZE];                         // 8
```

上面例子中，obj 有三个属性，它们的键（key）分别是"color"（合法的字符串标识符），"not an identifier"（不合法的字符串标识符），SIZE（符号）。

如果读者在使用 JavaScript 控制台，可能会注意到控制台不会将 SIZE 符号当作 obj 的属性列出来。的确（可以输入 obj[SIZE] 来验证），符号属性的处理方式会有些不同，一般默认是不显示的。其次要注意，符号属性的键是符号 SIZE，而不是字符串"SIZE"。为了证明这一点，可以输入 obj.SIZE = 0（成员访问操作符始终作用于字符串属性），然后输入 obj[SIZE] 和 obj.SIZE（或者 obj["SIZE"]）。

现在，回忆一下基本类型和对象的区别。本节中，一直在操作和修改变量 obj 所包含的对象，但是 obj 始终指向同一个对象。假如 obj 被字符串或数字或者任意其他基本类型取代，每一次改变的时候它将变成一个不同的值。换言之，obj 自始至终指向同一个对象，只是对象自身的内容被修改了。

在 obj 的例子中，创建了一个空对象，对象的语法还允许创建包含初始属性的对象。在大括号内部，属性用逗号间隔，属性名和属性值用冒号隔开。

```
const sam1 = {
    name: 'Sam',
    age: 4,
};

const sam2 = { name: 'Sam', age: 4 };          // 在同一行声明

const sam3 = {
    name: 'Sam',
    classification: {                          //属性值也可以是对象
        kingdom: 'Anamalia',
        phylum: 'Chordata',
        class: 'Mamalia',
        order: 'Carnivoria',
        family: 'Felidae',
        subfaimily: 'Felinae',
        genus: 'Felis',
        species: 'catus',
    },
};
```

这个例子中，又创建了三个对象来展示对象字面量语法。注意，sam1 和 sam2 包含的属性是一样的，然而，它们却是两个不同的对象（再一次拿基本类型做比较：两个包含数字 3 的基本类型变量有相同的引用）。对象 sam3 的 classification 属性本身就是一个对象。考虑一下访问 sam3 中 classification 的属性 family 的不同方式（无所谓使用单引号、双引号还是重音符）：

```
sam3.classification.family;                    // "Felinae"
sam3["classification"].family;                 // "Felinae"
sam3.classification["family"];                 // "Felinae"
sam3["classification"]["family"];              // "Felinae"
```

对象还可以包含函数。第 6 章会深入学习函数。但现在，只需要知道函数内部包含了代码（实质上是子程序）。来看看如何给 sam3 添加一个函数：

```
sam3.speak = function() { return "Meow!"; };
```

现在可以使用方法名+括号的方式调用它：

```
sam3.speak();              // "Meow!"
```

最后，可以通过 delete 操作来删除对象中的属性：

```
delete sam3.classification;          // 整个 classification 树被移除
```

```
delete sam3.speak;                    // speak 函数被移除
```

如果熟悉面向对象编程（OOP），可能想知道 JavaScript 对象是如何跟 OOP 关联起来的。现在，可以先将对象看成是一般的容器，后面会在第 9 章再详细讨论 OOP。

3.13 Number, String 和 Boolean 对象

本章一开始就提到：数字、字符串和布尔型都有对应的对象类型（Number，String，Boolean）。这些对象有两个用途：一是存储特殊值（比如 Number. INFINITY），二是以函数的形式提供某些功能。看看下面的例子：

```
const s = "hello";
s.toUpperCase();                      // "HELLO"
```

这个例子中，s 看起来像是一个对象（就好像它有一个函数属性，这里在访问它的函数属性）。但事实很清楚：s 是基本的字符串类型。那么这是怎么回事呢？JavaScript 所做的事情就是创建了一个临时的 String 对象（该对象有一个 toUpperCase 函数）。一旦这个函数被调用了，该临时对象就会被删除。为了证明这一点，来尝试给字符串指定一个属性：

```
const s = "hello";
s.rating = 3;                         // 咦，竟然没报错，成功了？
s.rating;                             // undefined
```

JavaScript 允许这么做，这看起来像是给字符串 s 指定了一个属性。而实际上，是在给 JavaScript 创建的临时 String 对象指定了一个属性。这个对象在使用后会立刻被删除，这就是为什么 s.rating 返回的结果是 undefined。

JavaScript 的这个行为对程序员来说是透明的，程序员几乎不用（如果有）去思考它，但是了解 JavaScript 在背后做了什么是非常有用的。

3.14 数组

数组是 JavaScript 的一种特殊类型的对象。不像一般的对象，数组的内容天生具有自然排序特性（元素 0 始终出现在元素 1 之前），它的键是数字，并且是有序的。数组有很多有用的方法，因而具有强大的传递信息的能力，将在第 8 章学习数组。

如果读者是从其他编程语言转过来的，会发现 JavaScript 数组是集其他语言优点于

一身的混合体，它拥有 C 语言数组的高效索引，同时拥有强大的动态数组和链表的特性。JavaScript 中的数组有以下特性：

- 数组长度不固定，可以随时添加和删除元素。
- 数组中元素的类型是多样的，每个元素都可以是任意类型。
- 数组下标从 0 开始。也就是说，数组的第一个元素是元素 0。

由于数组是具有一些额外功能的特殊对象，所以可以给数组指定非数字（部分或全部）的键。虽然可以这么做，但它违背了数组的设计初衷，会造成混乱，并且增加了定位 bug 的难度，所以请尽量避免这样做。

JavaScript 中用方括号创建数组字面量，多个元素之间用逗号隔开：

```
const a1 = [1, 2, 3, 4];                          // 包含数值的数组
const a2 = [1, 'two', 3, null];                    // 包含混合类型的数组
const a3 = [                                       // 跨越多行的数组
    "What the hammer?  What the chain?",
    "In what furnace was thy brain?",
    "What the anvil?  What dread grasp",
    "Dare its deadly terrors clasp?",
];
const a4 = [                                       // 包含对象的数组
    { name: "Ruby", hardness: 9 },
    { name: "Diamond", hardness: 10 },
    { name: "Topaz", hardness: 8 },
];
const a5 = [                                       // 包含数组的数组
    [1, 3, 5],
    [2, 4, 6],
];
```

数组有一个属性叫 length，它返回数组元素的个数：

```
const arr = ['a', 'b', 'c'];
arr.length;                                        // 3
```

在方括号中使用数字下标可以访问数组的指定元素（类似于访问对象的属性）：

```
const arr = ['a', 'b', 'c'];

// 获取第一个元素：
arr[0];                                            // 'a'
```

```
// 数组最后一个元素的下标是 arr.length-1:
arr[arr.length - 1];                              // 'c'
```

如果要改变数组中特定下标的元素的值，只要给它赋一个新值就可以了：[①]

```
const arr = [1, 2, 'c', 4, 5];
arr[2] = 3;      // arr 的值为[1, 2, 3, 4, 5]
```

在第 8 章中会学习更多修改数组及其内容的技巧。

3.15 对象和数组的拖尾逗号

警觉的读者可能已经注意到：在下面的例子中，对象和数组的内容跨越了多行，并且有个拖尾（或悬挂于末端）逗号：

```
const arr = [
    "One",
    "Two",
    "Three",
];
const o = {
    one: 1,
    two: 2,
    three: 3,
};
```

因为在早期版本的浏览器中，拖尾逗号会产生错误（尽管 JavaScript 语法允许这么做），所以很多开发人员会避免这种写法。而作者更喜欢使用拖尾逗号，因为作者本人会频繁地剪切复制数组和对象，或者在对象末尾添加内容，有了拖尾逗号后，就不用刻意在添加的内容前面加逗号了，这给本人带来了极大的便利。这是一个比较极端的编程习惯，也是本人的偏好。如果觉得拖尾逗号很麻烦（或者团队中没有这样的编程习惯），那么，请忽略它。

JavaScript 对象表示法（JSON）作为一个类 JavaScript 的数据语法会频繁地被使用，但它不允许存在拖尾符号。

① 如果你给的索引值大于或等于数组的长度，数组的尺寸会增加，从而能存储新元素。

3.16 日期

在 JavaScript 中，日期和时间是通过内置的 Date 对象表示的。Date 是编程语言中一个比较棘手的部分。(JavaScript 与 Java 有直接关系的地方本来就不多，而 Date 就是一个)。Date 对象很难处理，尤其是不同时区的日期。

创建一个表示当前日期和时间的日期对象，使用 new Date()即可：

```
const now = new Date();
now;   // 示例: Thu Aug 20 2015 18:31:26 GMT-0700 (Pacific Daylight Time)
```

创建一个指定日期（上午 12:00）的日期对象：

```
const halloween = new Date(2016, 9, 31);  // 注意，月份是从 0 开始的：9=十月
```

创建一个指定日期和时间的日期对象：

```
const halloweenParty = new Date(2016, 9, 31, 19, 0);  // 19:00 = 7:00 pm
```

创建日期对象后，就可以调用一些方法来检索该对象的组件：

```
halloweenParty.getFullYear();              // 2016
halloweenParty.getMonth();                 // 9
halloweenParty.getDate();                  // 31
halloweenParty.getDay();                   // 1 (Mon; 0=Sun, 1=Mon,...)
halloweenParty.getHours();                 // 19
halloweenParty.getMinutes();               // 0
halloweenParty.getSeconds();               // 0
halloweenParty.getMilliseconds();          // 0
```

在第 15 章，会详细介绍日期对象。

3.17 正则表达式

有规则的表达式（或正则表达式）可以算作 JavaScript 的一门子语言。很多编程语言都提供了这种通用的语言扩展功能。它通过一种简洁的方式完成字符串的复杂搜索和替换。第 17 章会讲解正则表达式。JavaScript 中正则表达式通过 RegExp 对象来表示，它们由一对正斜线包裹起来的若干字符和符号组成。下面是一些例子（如果没见过正则表达式，它看起来就像乱码）：

```
// 非常简单的邮件识别器
const email = /\b[a-z0-9._-]+@[a-z_-]+(?:\.[a-z]+)+\b/;
```

```
// US 手机号码识别器
const phone = /(:?\+1)?(:?\(\d{3}\)\s?|\d{3}[\s-]?)\d{3}[\s-]?\d{4}/;
```

3.18 映射和集合

ES6 引进了 Map 和 Set，以及它们的“弱”引用类型 WeakMap 和 WeakSet。映射也是一种对象，它将键和值关联映射在一起，但在某些特定的场合它比对象更有优势。集合类似于数组，但它不允许重复元素。弱引用类型的功能与其对应的类型相似，但在某些情况下，它们在功能上做了权衡以换取更好的性能。

第 10 章会讲解映射和集合。

3.19 数据类型转换

不同类型数据之间的转换是很常见的任务。来自于用户输入或者其他系统的数据通常需要做一些转换。本章节会讲解一些很常用的数据转换技巧。

3.19.1 转换成数字

字符串转数字是一种很常见的场景。当收集用户的输入时，用户输入的数据通常是字符串，即便想要收集的是数字。JavaScript 提供一组方法用来将字符串转换成数字。第一种是使用 Number 对象的构造方法：[①]

```
const numStr = "33.3";
const num = Number(numStr);    // 创建了一个数值，*不是*一个 Number 对象的实例
```

如果字符串内容不符合数字格式，就会返回 NaN。

第二种方式是使用内置函数 parseInt 和 parseFloat。它们的行为跟 Number 的构造方法非常类似，但有一些不同之处。使用 parseInt 时，可以指定一个基数，它代表要将数字转换成什么样的格式（译者注：比如十进制，十六进制等）。例如，可以指定一个基数 16 用来解析十六进制数。建议始终明确指定一个基数，甚至为默认值 10 的时候。parseInt 和 parseFloat 都会忽略任何跟数字不相关的信息，这允许插入一些不相干的字符。比如下面的例子：

```
const a = parseInt("16 volts", 10); // " volts" 被忽略，16 被当做十进制数解析
const b = parseInt("3a", 16);       // 解析十六进制数 3a；结果为 58
const c = parseFloat("15.5 kph");   // " kph" 被忽略；
```

① 通常情况下，你都会将构造器与关键字 new 结合起来使用，我们将在第九章学习这点。这儿是一个特殊的情况。

```
//parseFloat 始终假定参数为十进制数
```

日期对象也可以转成数字，转换后的结果代表从 1970 年 1 月 1 日凌晨 0:00 到当前时间的毫秒数。使用的是 valueOf() 方法：

```
const d = new Date();          // 当前日期
const ts = d.valueOf();        // 数字值：距离 UTC 1970.1.1 00:00:00 的毫秒数
```

有时，将布尔值转换成 1（true）或 0（false）会非常有用。可以使用条件操作符（会在第 5 章学到）来完成转换：

```
const b = true;
const n = b ? 1 : 0;
```

3.19.2 转换成字符串

JavaScript 中任何对象都有 toString() 方法，toString() 方法返回该对象的字符串形式。在实践中，toString() 方法的默认实现不是非常实用。该方法适用于数字，虽然将数字转换成字符串通常来说不是很有必要：这种转换会在字符串连接或插值的时候自动进行。但如果要将数字转换成字符串值，toString() 方法就派上用场了：

```
const n = 33.5;
n;                                    // 33.5 - 数字
const s = n.toString();
s;                                    // "33.5" - 字符串
```

Date 对象重新实现了一个很有用的 toString() 方法，但是大部分对象都只是简单地返回字符串"[object Object]"。对象可以被修改，从而返回更有用的字符串表现形式，不过这是第 9 章的话题了。数组的 toString()方法的实现也相当有用：它将每个元素转换成字符串，然后用逗号将这些字符串连接起来组成一个大的字符串。

```
const arr = [1, true, "hello"];
arr.toString();                       // "1,true,hello"
```

3.19.3 转换成布尔型

在第 5 章，会了解到 JavaScript 中的“正确”和“错误”的概念，它是一种强迫所有值非 true 即 false 的方式，所以这里不打算深入学习它。下面马上会看到，通过使用两次“not”操作符（!），任何值都可以转换成布尔值，使用一个 ！也可以将它转换成布尔值，只不过它会跟预期的值相反；再加一个 ！就转换成你想要的值了。对于数字转布尔值，Boolean 对象的构造方法（同样，这里没有 new 关键字）

也是一个很好的方式：

```
const n = 0;                          // "错误的" 值
const b1 = !!n;                       // false
const b2 = Boolean(n);                // false
```

3.20 小结

在任何编程语言中，可用的数据类型都是构建应用的基础，它们能有效地帮助开发人员在该语言中表达各种信息。基于大部分日常编码的需要，本章需要掌握的核心内容如下：

- JavaScript 有 6 种基本类型（字符串、数字、布尔、null、undefined，以及符号）和一种对象类型。
- 所有数字都是双精度浮点型。
- 数组是特殊的对象类型，它将对象组合起来，能表示强大并且灵活的数据类型。
- 其他常用的数据类型（日期，映射，集合和正则表达式）都是特殊类型的对象。后续很有可能会频繁的使用字符串，所以强烈建议在进一步学习之前，确保自己掌握字符串转义的规则，以及字符串模板的工作原理。

第 4 章

控制流

对于初学烹饪的人来说，“跟着菜谱”是一种常规的学习方法。这个方法可能很有用，但它却有一个缺点：因为在厨房里，为了保证每次做出来的食物味道一样，就需要将选择最小化。当这个菜谱被简化之后，就变得可重复了，步骤分明，其中变动的地方逐渐减少直到消失。当然，也可能会存在不同的选择，比如“用黄油代替猪肉”，或者“时令食材”。总体上说，菜谱是一系列有序步骤的集合。

本章的主要围绕改变和选择展开：让所编写的程序可以处理各种变化的条件，并且能自动执行重复任务。

如果已经有一些编程经验，尤其是那些继承自 C 语言的编程语言（C++、Java、C#），并且熟知控制流表达式，大可放心地简要浏览或者直接跳过本章的开头部分。不过，若跳过开头，会错过源自 19 世纪的赌博游戏哦。

4.1 控制流的底层

读者很有可能已经接触过流程图的概念，它是一种用来表示控制流的虚拟方法。下面将会写一个模拟器，来作为本章的示例代码。为了更加具体深入一些，下面例子模拟一个在 19 世纪中叶特别流行的赌博游戏——皇冠和锚的玩家，皇家海军的候补航海员。

游戏很简单，有一个 6 面分别标有“皇冠”“锚”“红桃”“梅花”“黑桃”和“方块”。水手可以在这些面的组合中放任何数量的钱数，这些钱就是赌注。接着他开始掷骰子，骰子总共有三个，每一个都有上面提到的 6 个面。如果掷出来的结果跟水手下

注的一样，那水手就赢了。下面这些例子演示了水手可能会掷出的结果，以及相应的盈利。如表 4-1 所示。

表 4-1　　赌博游戏

押　注	掷骰子结果	盈　利
5 便士压皇冠	皇冠，皇冠，皇冠	15 便士
5 便士压皇冠	皇冠，皇冠，锚	10 便士
5 便士压皇冠	皇冠，红桃，梅花	5 便士
5 便士压皇冠	红桃，锚，梅花	0
3 便士压皇冠，2 便士压梅花	皇冠，皇冠，皇冠	9 便士
3 便士压皇冠，2 便士压梅花	皇冠，梅花，锚	5 便士
所有花色都压 1 便士	任何一面	3 便士(这可不是一个好策略!)

挑选这个例子是因为它不复杂，而且还保留了一点想象的空间，重点是它能用来演示一个控制流的主要过程。实际上，不用模拟 19 世纪的海员来玩这个游戏，而是抽象这个行为即可，这种模拟在很多应用中是很常见的。在皇冠和锚的例子中，已经抽象出一个数学模型，来决定是否要开一局游戏来为公司的下一次活动筹钱。本章中构造的模拟器可以用来支撑模型的正确性。

游戏本身很简单，但是确有上千种玩法。船员——托马斯（这是一个很好的英国名字）——将会从基础的玩起，他的行为会随着游戏的发展而逐渐包含更多细节。

从基础开始：起始和结束条件。每次托马斯离港，都会带着 50 便士的本金来玩皇冠和锚的游戏。他自己有一个规矩：如果自己运气好使本金翻倍，他就带走这 100 多便士（几乎是他工资的一半）。

把游戏分为三部分：下注、掷骰子、收集赢到的钱（如果有的话）。现在已经勾勒出一个很简单且高度概括的托马斯行为。接下来就可以画一个流程图来表示它，如图 4-1 所示。

在流程图中，菱形表示“是或否”，方形表示动作。圆形表示起始和终止。

图 4-1 所画的流程图，并不能直接转换成程序。其中的步骤对普通人来说很简单，但是对于电脑来说却太过复杂。比如，“掷骰子”对于电脑而言并不那么显而易见。什么是骰子？怎么掷？为了解决这个问题，“下注”“掷骰子”“计算钱数”这几个步骤分别有各自的流程图（图 4-1 中的图形分别表示这三个步骤）。如果有一张足够大的纸，就可以把这些流程图画在一起，但本书只能分开来画。

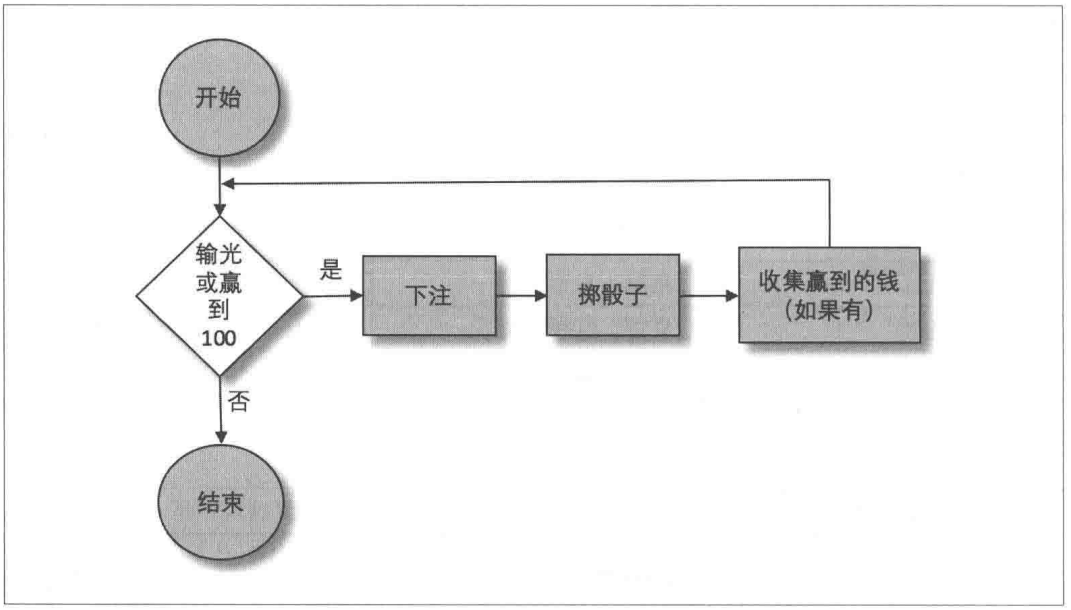


图 4-1　皇冠和锚模拟流程图

而且图中的决策点对于电脑来说太含糊了："破产或者赢到 100 便士？"这已经超出了电脑的理解范围。所以什么才是电脑能理解的呢？在本章中，这样约定流程图中的动作。

- 变量定义：`funds = 50, bets = {}, hand = []`。
- 一个 *m* 到 *n* 之间的随机数，包括 `rand(1,6)`（rand 是一个辅助方法，稍后会讲到）。
- 随机的字符串表示骰子向上的一面（比如：红桃、皇冠）：`randFace()`（另一个辅助方法）。
- 对象属性赋值：`bets["heart"] = 5, bets[randFace()] = 5`。
- 给数组添加元素：`hand.push(randFace())`。
- 基础计算：`funds - totalBet, funds + winnings`。
- 递增：`roll++`（这是一个常用的快捷方式，用来表示"给 roll 这个变量加一"）。

并且我们会限制流程图中的决策：

- 数字区间（`funds > 0, funds < 100`）。

- 判等（`totalBet === 7`；在第 5 章讨论为什么需要三个等号）。
- 逻辑操作（`funds > 0 && funds < 100`；两个“&”符号表示“并列”，也将在第 5 章中讲到。）

所有这些“被允许的操作”都是可以直接用 JavaScript 来编写，有些可能会需要一些解释和转换。

终极词汇笔记：本章中会使用一些词语如“*真的*”和“*假的*”。它们并不是简单的在真和假后面加后缀，或者只是一种“可爱”的说法，在 JavaScript 中它们是有意义的。这些都将会在第 5 章中讲到，现在可以简单地把他们想象成“真”和“假”。到此，已经对即将使用的语言有一些了解，现在需要用它们来重画图 4-1 所示流程图，得到图 4-2 中的流程图。

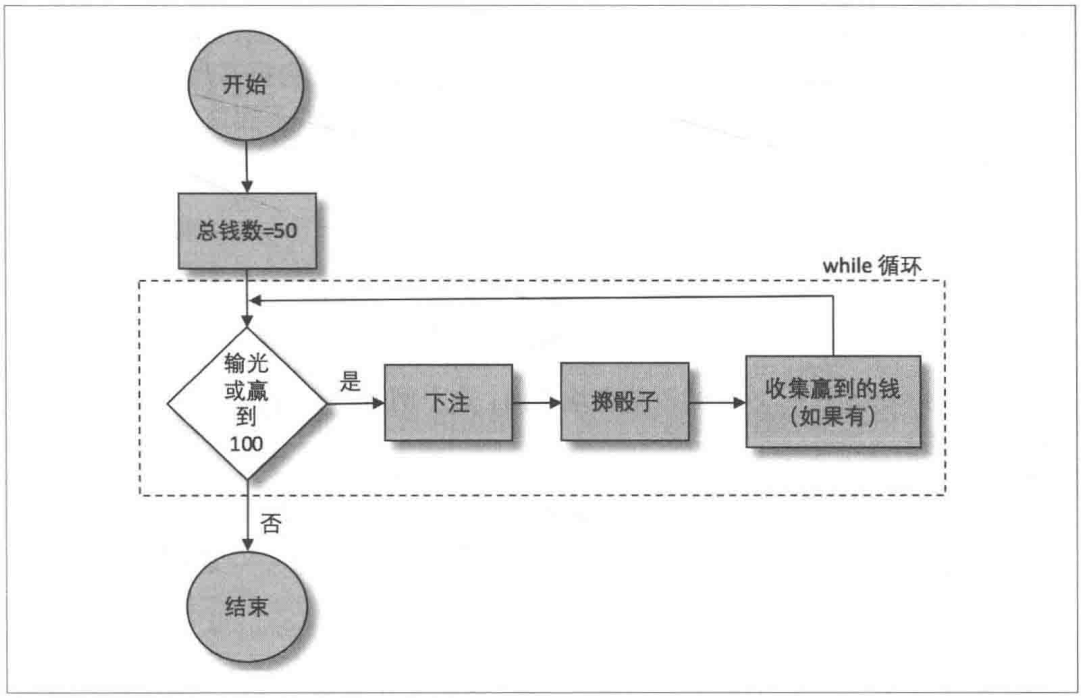


图 4-2　皇冠和锚模拟流程图（修改版）

4.1.1　while 循环

终于有一些可以直接转换成代码的东西了。图 4-2 中的流程图中已经包含了第一个控制语句：一个 `while` 循环。只要条件满足，`while` 循环会不断重复执行代码。在这个控制流中，条件是 `funds>1 && funds<100`。下面来看看用代码如何实现：

```
let funds = 50;      // starting conditions

while(funds > 1 && funds < 100) {
      // 下注

      // 掷骰子

      // 收集赢到的钱
}
```

运行这段代码，它会一直循环执行下去，因为 funds 这个变量的初始值是 50 便士，它永远不会增多或变少，所以 while 语句中的条件会一直为真。在实现代码细节之前，需要先讨论一下*块语句*。

4.1.2 块语句

块语句（有时也称*复合语句*）不是控制流语句，但它却和控制流语句相辅相成。块语句是被花括号包裹起来的一系列语句的集合，JavaScript 会把它当成一个程序单元。然而块语句也可能单独存在，它却没有太大用途。比如下面这个例子：

```
{  // 代码段开始
   console.log("statement 1");
   console.log("statement 2");
}  // 代码段结束

console.log("statement 3");
```

前两个调用 `console.log` 的语句在块里面；这个例子并不具备实际的意义，但它却是合法的。

块语句和控制流语句结合起来后就会变得有用起来。比如，前面执行过的 `while` 循环会执行*完整的块语句*后才会再次判断条件是否满足。又比如，如果想“每次给 funds 值增加 2，然后减去 1”，就可以这样写：

```
let funds = 50;            // 起始条件

while(funds > 1 && funds < 100) {

    funds = funds + 2;    // 往前两步
    funds = funds - 1;    // 后退一步
}
```

`while` 循环最终会结束：因为每次执行循环的时候，`funds` 都会加 2 然后减 1，总值会加 1。最终 funds 会达到 100，从而终止循环。

将块语句和控制流结合使用是一种很常见的用法，但不是必须的。比如，如果只想简单的通过每次加 2 把 funds 的值增加到 100，就不需要块语句：

```
let funds = 50;          // 起始条件

while(funds > 1 && funds < 100)
    funds = funds + 2;
```

4.1.3 空格

在大多数情况下，JavaScript 并不关心多余的空格（或者空行[①]）：一个空格和十个空格没有区别，十个空格和十个空行也没有区别。不过这并不是说可以任意使用空格。比如，前面例子中的 while 语句可以写成：

```
while(funds > 1 && funds < 100)

funds = funds + 2;
```

不过这样把两行连在一起，就很难看出来这是一个 while 循环！应该尽量避免这种容易引起误解的代码缩进。下面这种格式类似跟上面的类似，不过相对来说更常用一些，也更清晰。

```
// 不用另起一行
while(funds > 1 && funds < 100) funds = funds + 2;

// 不用另起一行，代码块中只有一行语句
while(funds > 1 && funds < 100) { funds = funds + 2; }
```

有些人为了强调控制流的一致性和清晰度，坚持使控制流放在块语句中（即使它只有一行）。作者并不觉得这是必须的，需要指出的是，如果不小心写成下面这样的缩进，也很容易引起歧义的。

```
while(funds > 1 && funds < 100)
    funds = funds + 2;
    funds = funds - 1;
```

快速扫过上面的例子，看起来好像是 while 循环中执行了两个语句（前进两步然后退后一步），但是因为它们没有被块语句包起来，JavaScript 会这样理解它：

```
while(funds > 1 && funds < 100)
    funds = funds + 2;              // while 循环体

funds = funds - 1;                  // while 循环结束
```

① return 语句后面如果有空行将会报错，第 6 章会详细介绍。

作者同意当块语句只有一行的时候可以省略花括号，不过要时刻注意使用恰当的缩进使代码的意图更清楚。还有，如果在一个团队中工作或者在开源项目中，就应该遵守团队约定好的格式规范，而不是按照自己喜欢的格式。

对于是否对单行代码使用块这个问题，至今还存在争议，一个合乎语法但却饱受责难的选择是：在 if 语句中混合使用块语句和单行语句。

```
// 不要这样做
if(funds > 1) {
    console.log("There's money left!");
    console.log("That means keep playing!");
} else
    console.log("I'm broke!  Time to quit.");

// 或者这样做
if(funds > 1)
    console.log("There's money left!  Keep playing!");
else {
    console.log("I'm broke"!);
    console.log("Time to quit.")
}
```

4.1.4 辅助方法

为了完成本章中的例子，还需要两个*辅助方法*。至此还没有提及方法（或者伪随机数生成器），不过下一章会讲到。现在，先照抄下面的两个方法：

```
// 返回在[m,n]之间的随机整数 (包含边界)
function rand(m, n) {
    return m + Math.floor((n - m + 1)*Math.random());
}

// 随机返回代表皇冠和锚游戏中六个面其中之一的字符串
function randFace() {
    return ["crown", "anchor", "heart", "spade", "club", "diamond"]
        [rand(0, 5)];
}
```

4.1.5 if else 语句

流程图中有一个带阴影的方块，其中写着“下注”，下面来完成这部分的代码。那么托马斯怎样才能下注呢？托马斯有一个惯例，这个惯例是这样的：他会从右边口袋里随机拿出一把硬币（少则一枚硬币，多则全部拿出）。这些就是他本轮的资金。另外，托马斯很迷信，他相信数字 7 可以给他带来好运。所以如果他正好拿出了 7 枚硬币，他就会把它们都放回去然后把所有钱都压在“红桃”上。否则，他会把拿

到的钱随机下注（后面会提到如何下注）。图 4-3 就是“下注”这个模块的流程图。

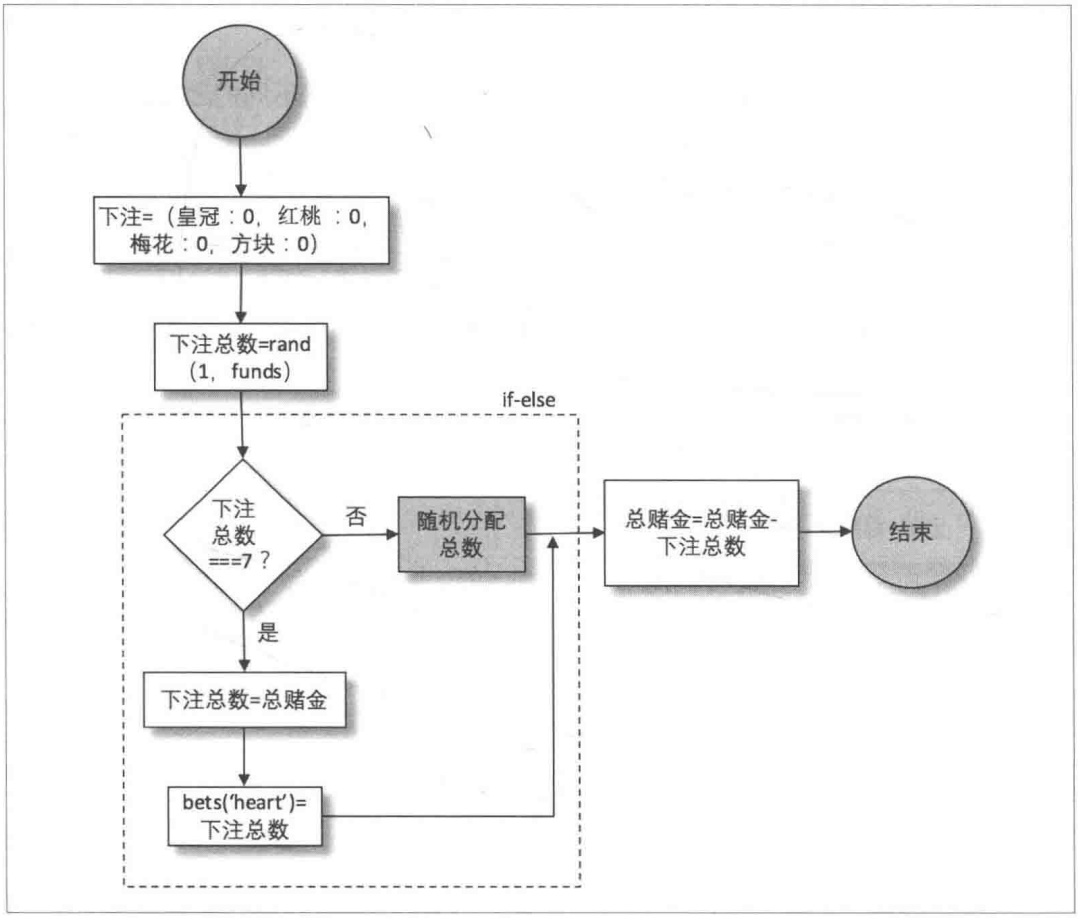


图 4-3　皇冠和锚模拟器：下注流程图

中间的条件节点（totalBet===7）代表一个 if else 语句。注意它不像 while 语句会循环执行，一旦确定，就继续往下走。这个流程可转化成下面的 JavaScript 代码：

```
const bets = { crown: 0, anchor: 0, heart: 0,
    spade: 0, club: 0, diamond: 0 };
let totalBet = rand(1, funds);
if(totalBet === 7) {
    totalBet = funds;
    bets.heart = totalBet;
} else {
    // 分配下注总数
}
funds = funds - totalBet;
```

if...else 语句中的 else 部分不是必须的，我们后面会看到省去了 else 的实现。

4.1.6 do...while 循环

当托马斯拿出的银币不是 7 个时，他会随机把这些钱分配到不同的地方。对此他也有惯例：他右手拿钱，左手随机从右手的钱里拿出一些（少则一枚，多则全部），然后把左手的钱随机下注到其中一个里面（有时候他会把钱重复下注到一个里面）。在下注这个模块里更新随机分配的流程，如图 4-4 所示。

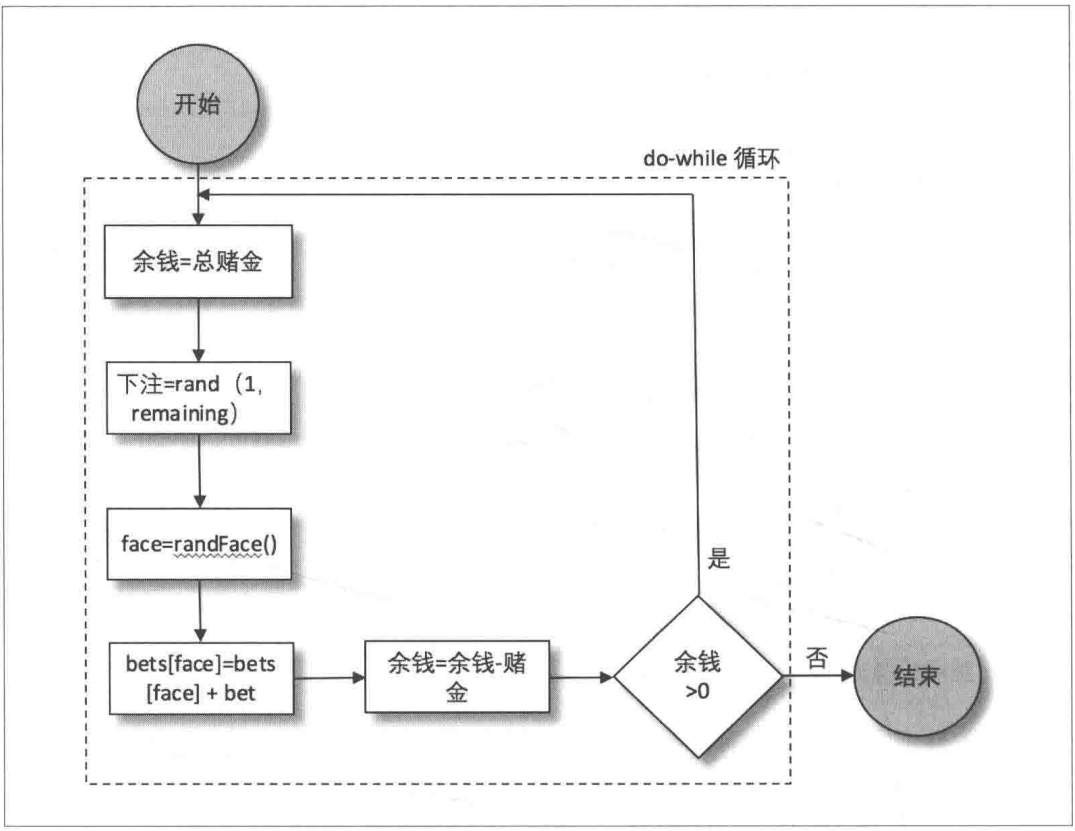


图 4-4 皇冠和锚模拟器：分配赌金流程图

注意，这个与 while 循环不一样：判断语句在最后面，而不是开始。Do... while 循环是为了实现那些循环体至少被执行一次的情况（在 while 语句中，如果最开始的判断返回 falsy，循环体一次都不会执行）。JavaScript 代码如下：

```
let remaining = totalBet;
do {
    let bet = rand(1, remaining);
    let face = randFace();
```

```
    bets[face] = bets[face] + bet;
    remaining = remaining - bet;
} while(remaining > 0);
```

4.1.7 for 循环

托马斯已经下完注！是时候掷骰子了。

for 循环非常灵活（它甚至可以替换 while 循环和 do... while 循环），它更适合那些需要执行固定次数的情况（尤其是当需要知道执行到哪一步的时候），所以用 for 循环来投掷固定数字的骰子（本例中需要的数字是 3）再合适不过了。还是从“掷骰子”流程图开始，如图 4-5 所示。

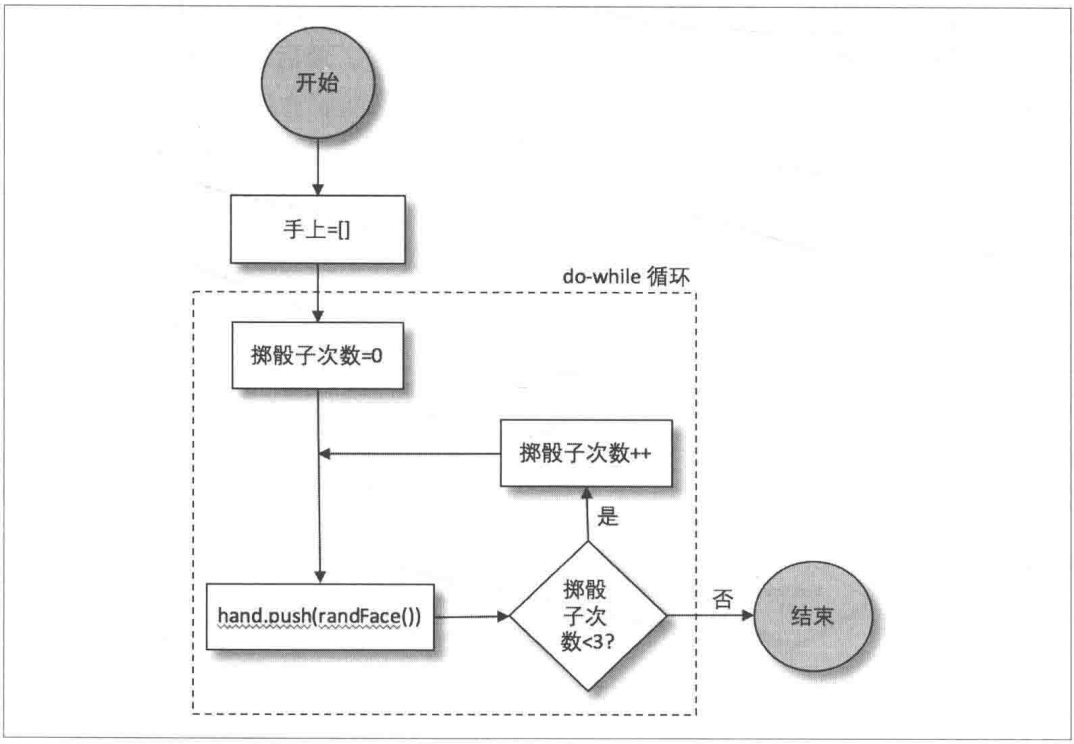


图 4-5 皇冠和锚模拟器：掷骰子流程图

for 循环由三个部分组成：*初始值*（roll = 0），*条件*（roll < 3），和*最终表达式*（roll++）。虽然 while 也可以做这些事情，但是 for 循环能很方便地把循环所需的所有信息放在一起。下面 JavaScript 中 for 循环的实现：

```
const hand = [];
for(let roll = 0; roll < 3; roll++) {
    hand.push(randFace());
}
```

开发人员习惯于从 0 开始计数，这也就是为什么这段代码中会从 0 开始，到 2 停止。

在 for 循环中使用变量 i（index 单词的简写）计数已经是约定俗成的规范，不管给什么计数，都可以随心所欲地用开发人员想要的变量名进行计数。这里使用 roll 是为了明确表示是在给投掷次数计数，不过还是需要当心一点：因为在作者第一次写这个例子的时候，出于习惯用了 i！

4.1.8 if 语句

马上就要完成了！已经有了下注和掷骰子，剩下的就是收集赢钱的信息。在 hand 数组里有三个随机值，所以这里还需要一个 for 循环来找出其中的胜者。为了完成这个任务，需要 if 语句（这次不需要 else 部分）。最后这部分的流程图如图 4-6 所示。

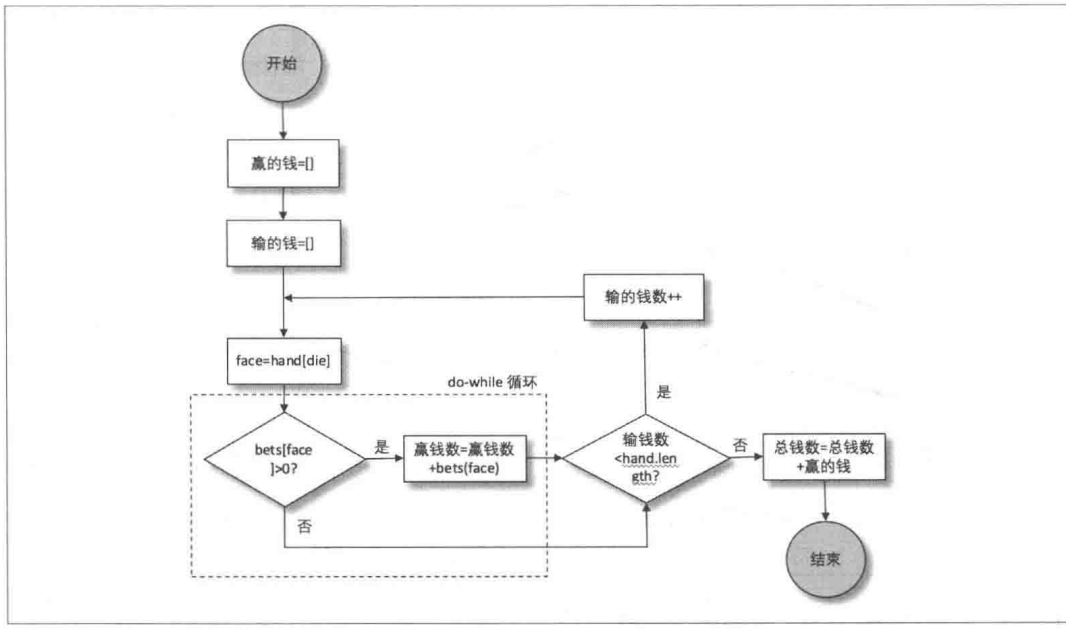


图 4-6 皇冠和锚模拟器：收集赢钱信息的流程图

注意 if...else 语句和 if 语句之间的区别：在 if 语句中，只有一个分支会存在可执行的动作，而在 if... else 语句中是两个分支。这个例子中最后一部分的代码实现如下：

```
let winnings = 0;
for(let die=0; die < hand.length; die++) {
    let face = hand[die];
```

```
        if(bets[face] > 0) winnings = winnings + bets[face];
    }
    funds = funds + winnings;
```

注意，for 循环并没有循环到 3，而是到 hand.length（这个值恰好是 3）。这部分代码的作用是计算所有的赢钱情况，不管 hand 有多少种情况。这个游戏规定手里有三个骰子，不过规则是可以更改的，可能以后会有更多的骰子作为奖励，或者更少的骰子作为惩罚。重点是，只需要做少量的工作就可以让这段代码变得更通用。如果游戏规则改变了，允许更多或者更少的骰子，开发人员就不用花精力修改这段代码，不管有多少骰子它都会准确无误的执行并给出结果。

4.1.9 最后的整合

可以找一张大纸把所有的流程图拼在一起，相比较而言，写出整块代码要容易得多。

在下面的代码中（包含辅助函数），有一些调用 `console.log` 的地方，这样就查看托马斯游戏的进度（暂时不用急于理解 log 是如何工作的，因为它用到了一些后面章节才会讲到的先进技术）。

另外，还添加了轮数变量来计算托马斯玩了多少轮游戏，该变量只用于显示：

```
// 随机返回[m,n]之间的整数（包含边界）
function rand(m, n) {
    return m + Math.floor((n - m + 1)*Math.random());
}
//随机返回代表皇冠和锚游戏中六个面其中之一的字符串
function randFace() {
    return ["crown", "anchor", "heart", "spade", "club", "diamond"]
        [rand(0, 5)];
}

let funds = 50;      // starting conditions
let round = 0;

while(funds > 1 && funds < 100) {
    round++;
    console.log('round ${round}:');
    console.log('\tstarting funds: ${funds}p');
    // 掷骰子
    let bets = { crown: 0, anchor: 0, heart: 0,
        spade: 0, club: 0, diamond: 0 };
    let totalBet = rand(1, funds);
    if(totalBet === 7) {
        totalBet = funds;
        bets.heart = totalBet;
```

```
        } else {
            // 分配下注总数
            let remaining = totalBet;
            do {
                let bet = rand(1, remaining);
                let face = randFace();
                bets[face] = bets[face] + bet;
                remaining = remaining - bet;
            } while(remaining > 0)
        }
        funds = funds - totalBet;
        console.log('\tbets: ' +
           Object.keys(bets).map(face => '${face}: ${bets[face]} pence').join('
,')+'(total: ${totalBet} pence)');

    // 掷骰子
    const hand = [];
    for(let roll = 0; roll < 3; roll++) {
        hand.push(randFace());
    }
    console.log('\thand: ${hand.join(', ')}');

    // 收集赢到的钱
    let winnings = 0;
    for(let die=0; die < hand.length; die++) {
        let face = hand[die];
        if(bets[face] > 0) winnings = winnings + bets[face];
    }
    funds = funds + winnings;
    console.log('\twinnings: ${winnings}');
    }
    console.log('\tending funs: ${funds}');
```

4.2 JavaScript 中的控制语句

至此，大家应该很清楚控制流是干什么的了，以及最基础的控制流怎么写，这些都是在为深入学习 JavaScript 中的控制语句打基础。

先暂时不看流程图。虽然它是很不错的可视化工具（尤其是对那些注重视觉的学者来说），但却不易操作和更改。

从广义上讲，控制流分为两种：条件（或者分支）控制流和循环控制流。条件控制流（已经讲过的 `if` 和 `if...else`，以及稍后会讲到另一种 `switch`）就像路上的分叉：有两条或更多的路可以走，一旦走了其中一条，就不能回头。循环控制流

(while, do... while, 和 for 循环) 则会不断重复循环体直到满足某种条件。

4.2.1 控制流异常

有四个语句可以改变控制流的正常处理流程。通俗地讲，可以把它们想象成控制流的“杀手锏”：

`break`

提前结束循环。

`continue`

跳到循环的下一步。

`return`

退出当前方法（不管进行到控制流的哪一步）。详情见第 6 章。

`throw`

指出必须被异常处理器所捕获的异常（包括超出当前控制流的异常）。详情见第 11 章。

这些语句的用法会伴随后续学习的推进而逐渐明晰。目前要重点理解的是，这四个语句可以重写当前控制流结构中的一些行为，这也是接下来要讨论的内容。

从定义上讲，控制流可以分为两大类：条件控制流和循环控制流。

4.2.2 链式 if...else 语句

链式 `if...else` 语句并不是一种特殊的语法：它只是一系列简单的 `if...else` 语句的集合，其中的每个 `else` 字句包含了另一个 `if...else` 语句。这是一种非常值得一提的通用模式。比如，如果托马斯的迷信行为延续了整个星期，而且每到周三，他只拿一便士下注。这段逻辑用 `if...else` 来写就是这样的：

```
if(new Date().getDay() === 3) {    // new Date().getDay() 返回当前所在的星期
    totalBet = 1;                  // 的数字，星期天是 0
} else if(funds === 7) {
    totalBet = funds;
} else {
    console.log("No superstition here!");
}
```

按照这种方式组合了 `if...else` 语句后，就有了三种选择。聪明的读者可能意识

到这种做法违反了自己建立的规则（不要把单行语句和块语句放在一起），这种情况是一个特例：它是一个很常见的模式，并且代码清晰易读。这个例子也可以用块语句重写为：

```
if(new Date().getDay() === 3) {
    totalBet = 1;
} else {
    if(funds === 7) {
        totalBet = funds;
    } else {
        console.log("No superstition here!");
    }
}
```

不难看出，这样写并没有使代码更清楚，而且代码更加冗长了。

4.2.3 元语法

*元语法*也是一种语法，可以用它来描述或传达另一种语法。有计算机专业背景的读者可能马上会联想到扩展巴科斯-瑙尔范式（EBNF），这个很高深的名字背后其实蕴含了一个简单的概念。

本章剩下的部分，会用元语法来简单的描述 JavaScript 控制流语法。这里用的都是一些简单且非正式的元语法，最重要的是，它们都来自火狐开发者网站 MDN（*https://developer.mozilla.org/en-US/docs/Web/JavaScript*）上的 JavaScript 文档。毫无疑问，MDN 将会是一个大家经常使用的网站，熟悉它会对日后的开发会很有帮助。

元语法中只有两个真正的元素：方括号内的内容不是必须的，省略号（一般是三个句点）表示“这里还有更多内容。”单词作为占位符，它们所表示的意思可以从上下文猜出。比如：表达式 1 和表达式 2 是两个不同的表达式，表达式可以计算出值，条件也是一种表达式，只不过它们只会被当成真或假。

块语句也是语句，所以任何可以使用语句的地方，都可以使用块语句。

前面早已熟悉了一些控制流语句，它们的元语法是这样的：

while 语句

```
while(condition)
    statement
```

当条件为真时，语句会被执行。

if... else 语句

```
if(condition)
    statement1
[else
    statement2]
```

如果条件为真，表达式 1 会被执行；否则，表达式 2 被执行（假设 else 的部分存在）

do... while 语句

```
do
    statement
while(condition);
```

do while 语句会至少执行一次，只要条件为真，它会一直重复地执行下去。

for 语句

```
for([initialization]; [condition]; [final-expression])
    statement
```

在循环开始之前，初始化部分先被执行。一旦条件为真，语句就会被执行，然后终止表达式会在下一次判断条件之前执行。

4.2.4 其他循环模式

通过使用逗号操作符（将在第 5 章详细介绍），可以将多个赋值操作和终止表达式连起来。比如，下面这个 for 循环可以打印斐波那契数列的前 8 个数字。

```
for(let temp, i=0, j=1; j<30; temp = i, i = j, j = i + temp)
    console.log(j);
```

上面这个例子中，定义了多个变量（temp，i 和 j），同时在终止表达式中对每它们都进行了操作。正如可以通过在 for 循环中使用逗号运算符做更多的事情一样，所以也可以什么都不写，这样就会创建一个永远不会停止的循环。

```
for(;;) console.log("I will repeat forever!");
```

在这个循环中，条件表达式只会去计算 undefined 的值，也就是假的。这意味着循环永远不会退出。

不过，最常见的 for 循环还是按照数字递增或者递减，当然，数字不是必须的，任何表达式都可以。来看一些例子：

```
let s = '3';                                    // 包含一个数字的字符串
for(; s.length<10; s = ' ' + s);                // 没有填充初始表达式；注意必须添加
                                                // 一个分号

for(let x=0.2; x<3.0; x += 0.2)                 // 增量使用了非整数
    console.log(x);

for(; !player.isBroke;)                         // 使用对象的属性作为条件
    console.log("Still playing!");
```

注意 for 循环总是可以用 while 循环写出来，换句话说：

```
for ( [初始值]; [条件]; [终止表达式] )
    语句
```

和

```
[初始值]
while([条件]){
    语句
    [终止表达式]
}
```

是等效的。

然而，事实上是，for 循环可以被写成 while 循环并不代表就应该这么做。for 循环的好处是所有与循环相关的信息都会出现在第一行，它可以很清楚地说明这个循环在做什么。同时，对于 for 循环来说，初始变量的作用范围仅仅在循环体内部（在第 7 章详细介绍）。如果把 for 循环转换成 while 语句，控制变量的作用范围将会扩大到循环体外。

4.2.5 switch 语句

if…else 语句可以选择两条路径中的一条，而 switch 语句允许在一个单一的条件下采取多条路径。不过条件必须是比简单的真/假值更加多样化的值：对于一个 switch 语句来说，条件是一个可以计算值的表达式。如下 switch 语句的语法：

```
switch（表达式）{
    case 值1:
        //当表达式的执行结果跟值 1 匹配的时候执行
        [break;]
    case 值2:
        //当表达式的执行结果跟值 2 匹配的时候执行
```

```
        [break;]
        ...
    case 值N:
        //当表达式的执行结果跟值N匹配的时候执行
        [break;]
    default:
        //当表达式的结果跟任何一个值都不匹配的时候执行
       [break;]
}
```

JavaScript 会计算表达式，选取第一个与结果匹配的分支，然后执行分支中的语句直到遇到 break、return、continue、throw 或者执行到 switch 语句的最后(后面会讲到这些概念)。如果觉得这些有点复杂，也没关系：因为很多开发人员经常碰到关于 switch 语句的错误，所以它颇受指责，这也是 switch 语句的缺点之一。所以新手通常都不喜欢用它。其实，如果能正确使用 switch 语句，它会成为一个非常有用工具，不过也得勤加练习，才能在恰当的场景发挥它的作用。

下面以一个很直观的例子开始 switch 语句的学习。在皇冠和锚游戏中，如果虚构的水手有多个迷信的数字,那么就可以用 switch 语句来合理的处理这个场景：

```
switch(totalBet) {
    case 7:
        totalBet = funds;
        break;
    case 11:
        totalBet = 0;
        break;
    case 13;
        totalBet = 0;
        break;
    case 21:
        totalBet = 21;
        break;
}
```

注意当 totalBet 的值为 11 或 13 时，执行了相同的语句。这就是使用 *fall-through execution* 的地方。在前面说过，switch 语句会一直执行，直到遇到 break 语句。利用这个特性的地方就叫作 fall-through execution。

```
switch(totalBet) {
    case 7:
        totalBet = funds;
        break;
    case 11:
    case 13:
```

```
            totalBet = 0;
            break;
        case 21:
            totalBet = 21;
            break;
    }
```

到目前为止，代码看起来还非常直观。很显然，当托马斯拿出了 11 或者 13 便士时，他就会停止下注。但是如果 13 比 11 更不吉利呢？当拿到 13 时，不仅要停止下注，还要拿出 1 便士来做慈善。通过一些巧妙的安排来实现，可以这样做：

```
switch(totalBet) {
    case 7:
        totalBet = funds;
        break;
    case 13:
        funds = funds - 1;  // 拿出一便士做慈善!
    case 11:
        totalBet = 0;
        break;
    case 21:
        totalBet = 21;
        break;
}
```

如果 totalBet 是 13，就拿出一便士做慈善，但是因为这里没有 break 语句，会进入下一个 case（11），紧接着把 totalBet 设为 0。这是一段合法的 JavaScript 代码，此外，它也是正确的代码：它做了开发人员期望它做的事情。不过也存在缺点：看起来像是不小心写错了（即使它是正确的）。想象一下当另一个同事看到这段代码时，他可能会说“哎呀，这里少一个 break 语句。”然后加上一个 break 语句，如此一来这段代码就不正确了。所以很多人觉得比起它的好处，fall-through execution 带来的麻烦更大，不过如果打算用这个特性，建议最好加上注释让代码的意图更明晰。

通常总是能够定义一种特殊的情况，称之为 default，当其他情况都不匹配的时候，它就会被执行。普遍（不是必须的）的做法是在 switch 语句的最后放一个 default：

```
switch(totalBet) {
    case 7:
        totalBet = funds;
        break;
    case 13:
        funds = funds - 1;  // 拿出一便士做慈善!
    case 11:
```

```
        totalBet = 0;
        break;
    case 21:
        totalBet = 21;
        break;
    default:
        console.log("No superstition here!");
        break;
}
```

最后一个 break 其实可以省略，因为 default 后面并没有其他 case，不过给每个 case 后面加上 break 是一个好习惯。即使在使用 fall-through execution，也最好保持添加 break 语句的习惯：这样可以随时用注释替换 break 来激活 fall-through execution，但是如果在正确的情况下省略 break 语句，那将成为一个很难定位的代码缺陷。这个规则也有一个例外：当在函数里使用 switch 语句的时候（详情见第 6 章），可以把 break 语句换成 return 语句（因为 return 语句会立刻结束函数执行）。

```
function adjustBet(totalBet, funds) {
    switch(totalBet) {
        case 7:
            return funds;
        case 13:
            return 0;
        default:
            return totalBet;
    }
}
```

与往常一样，JavaScript 并不关心开发人员使用了多少空格，所以为了使 switch 语句更简洁，通常会把 break（或 return）语句跟 case 语句放在同一行。

```
switch(totalBet) {
    case 7: totalBet = funds; break;
    case 11: totalBet = 0; break;
    case 13: totalBet = 0; break;
    case 21: totalBet = 21; break;
}
```

注意，在这里选择在 totalBet 等于 11 或 13 的时候重复所要执行的操作：当每种 case 只有一条执行语句且整个 switch 语句没有 fall-through execution 的时候，为了让代码更加清晰，可以省略 break 语句占用的一行。

当想要基于一个表达式执行多个不同的动作时，switch 语句是最适合的。即便如此，在学习了第 9 章的动态分配后，依然会发现自己使用 switch 语句的频率

降低了。

4.2.6 for...in 循环

for...in 循环是为那些循环对象中有一个属性 *key* 而设计的。语法是：

```
for（变量 in 对象）
    语句
```

来看一个例子：

```
const player = { name: 'Thomas', rank: 'Midshipman', age: 25 };
for(let prop in player) {
    if(!player.hasOwnProperty(prop)) continue;  // 代码解释如下
    console.log(prop + ': ' + player[prop]);
}
```

如果这里觉得有点困惑，别担心，在第 9 章中对这个例子有更多的了解。特别说明一下，调用 player.hasOwnProperty 并不是必须的，但是省略它却容易造成一个常见的错误，这个内容也将在第 9 章中介绍。现在，只需要知道它是一种循环控制流语句。

4.2.7 for...of 循环

for...of 运算符是 ES6 中的新语法，它提供了另一种在集合中遍历元素的方法。它的语法是：

```
for（变量 of 对象）
    语句
```

for...of 循环可以用在数组上，但一般它可以用于遍历任何可迭代的对象（详情见第 9 章）。下面这个例子使用 for..of 来循环一个数组中的内容：

```
const hand = [randFace(), randFace(), randFace()];
for(let face of hand)
    console.log('You rolled...${face}!');
```

如果要遍历一个数组，但不需要知道每个元素的索引，for...of 是一个绝佳的选择。如果需要知道索引，就用常规 for 循环：

```
const hand = [randFace(), randFace(), randFace()];
for(let i=0; i<hand.length; i++)
    console.log('Roll ${i+1}: ${hand[i]}');
```

4.3 实用的控制流模式

现在已经知道了 JavaScript 中基本的控制流结构了，接下来把注意力转到一些以后会用到的常见模式中。

4.3.1 使用 continue 减少条件嵌套

常见的情况是，在循环体中，只希望在某些特定条件下继续执行循环体（尤其是当循环控制和条件控制结合起来的时候）。比如：

```
while(funds > 1 && funds < 100) {
    let totalBet = rand(1, funds);
    if(totalBet === 13) {
        console.log("Unlucky!  Skip this round....");
    } else {
        // play...
    }
}
```

这是一个典型的嵌套控制流：在 `while` 语句的循环体中，有一大部分语句在 `else` 中；在 `if` 语句只是简单调用了 `console.log`。这里就可以使用 `continue` 语句使结构更加“扁平”：

```
while(funds > 1 && funds < 100) {
    let totalBet = rand(1, funds);
    if(totalBet === 13) {
        console.log("Unlucky!  Skip this round....");
        continue;
    }
    // play...
}
```

在这个简单的例子中，使用 `continue` 并不能产生立竿见影的好处，但是想象一下，如果循环体不是 1 行，而是 20 行。从嵌套的控制流中删除这些行，会使代码更易于理解且大大提高了可读性。

4.3.2 使用 break 或 return 避免不必要的计算

如果循环仅仅是为了找到一个特定值，这样就没有必要在已经找到目标值后还继续执行到最后。

比如，计算一个数字是否为素数是一个相对耗费资源的计算。如果想从 1 000 个数字中查找第一个素数，比较幼稚的做法可能是：

```
let firstPrime = null;
for(let n of bigArrayOfNumbers) {
    if(isPrime(n) && firstPrime === null) firstPrime = n;
}
```

如果 bigArrayOfNumbers 有上百万个数字，而且只有最后一个是素数（事先并不知道），这样的做法还行。但是如果第一个就是素数呢？或者第 5 个，第 15 个？本可以早点结束这个计算，却需要挨个检查近百万个数字！听起来就很累。其实，当找到目标值后，就可以立刻使用 break 语句来结束循环：

```
let firstPrime = null;
for(let n of bigArrayOfNumbers) {
    if(isPrime(n)) {
        firstPrime = n;
        break;
    }
}
```

如果循环写在一个方法中，可以用 return 语句来替代 break。

4.3.3 在循环结束后使用索引的值

有时，需要获得循环被 break 语句终止时当前元素的索引值。这时候就能从下面这个特性中获益：当 for 循环结束时，索引变量的值依然被保留在循环体内。如果决定使用这个模式，记住有一种边缘情况是，循环成功地执行到最后，break 语句没有被调用。比如，可以用该模式找到数组中第一个素数的索引：

```
let i = 0;
for(; i < bigArrayOfNumbers.length; i++) {
    if(isPrime(bigArrayOfNumbers[i])) break;
}
if(i === bigArrayOfNumbers.length)  console.log('No prime numbers!');
else console.log(`First prime number found at position ${i}`);
```

4.3.4 列表变动时索引递减

当在循环一个列表的同时还在修改它，情况就变得有趣了。因为当列表修改时，需要相应的修改循环终止条件。最好的情况是循环的结果不是你想要的；最糟糕的情况是这个循环永远不会结束。常见的做法是使用索引递减的循环方式，也就是循环顺序由终止的地方开始，到起始位置结束。这时，如果列表有增删操作，就不会影响循环终止条件。

比如，从 bigArrayOfNumbers 中删除所有质数。使用数组中的 splice 方法给数组添加或者删除一个元素（详情见第 8 章）。下面代码就不会达到预期的效果：

```
for(let i=0; i<bigArrayOfNumbers.length; i++) {
    if(isPrime(bigArrayOfNumbers[i])) bigArrayOfNumbers.splice(i, 1);
}
```

因为索引是递增的，而程序执行中还在删除元素，所以循环的时候很可能跳过一些素数（如果它们是相邻的）。可以通过索引递减来解决这个问题：

```
for(let i=bigArrayOfNumbers.length-1; i >= 0; i--) {
    if(isPrime(bigArrayOfNumbers[i])) bigArrayOfNumbers.splice(i, 1);
}
```

这里要小心循环的起始和判断条件：索引必须从比数组长度小一的位置开始，因为数组下标是从零开始的。同时，当 i 大于或等于 0 的时结束循环；否则会漏掉第一个元素（如果第一个元素恰好是素数，结果就会出问题了）。

4.4 小结

控制流确实是编程界的一个里程碑。变量和常量中也许包含了所有的有用信息，不过得益于控制流语句，才能基于那些数据做出有用的选择。

流程图是一个可以把控制流进行可视化的有用方法，通常情况下，在开始写代码前，一幅高层次的流程图可以帮助开发人员清晰地描述清楚问题。最后可能会发现流程图不够简洁，代码才是表述控制流的一个既高效又（在勤加练习后）自然的方式（有很多人尝试发明一种只能可视化的结构化编程语言，可直到今天基于文本的编程语言都依然闪耀着光辉）。

第5章

表达式和运算符

表达式是一种特殊的语句，它可以计算出一个值。表达式语句（结果产生一个值）和非表达式语句（不产生值）的区别至关重要，理解它们的差异能够帮助你用实用的方式组合编程语言中的元素。

大家可以认为非表达式语句是一条指令，而表达式语句是一个请求。想象如果第一天上班，领班走过来说："你的工作是将部件 A 固定到法兰 B（译者注：法兰是轴与轴之间相互连接的零件，用于管端之间的连接）上。"这是一个非表达式语句：领班没有让你上交装配后的零件，仅仅是命令你去做装配。如果领班说："将部件 A 固定到法兰 B 上，然后交给我检查"，这就相当于一个表达式语句：你不仅仅接收到一个命令，而且执行完命令后要返回结果。大家可能会觉得这两种方法都会导致同一个结果：不论是把装配好的零件放回到生产线还是上交给领班检查，零件本身始终是存在的。在编程语言中也是类似的：非表达式语句通常会产生某种结果，但是只有表达式语句会对生成的结果做显式的转换。

因为表达式能解析成值，所以可以将它们与其他表达式组合在一起，进而将返回的结果再与其他表达式进行组合，以此类推。从另一方面讲，非表达式语句也有一些用处，但它们不能以同样的方式组合在一起。

因为表达式能解析成值，所以可以用它们来赋值。也就是说，可以将表达式的值赋给变量、常量或者属性。下面来考虑一种常见的运算表达式：乘法。乘法肯定是一个表达式：两个数字相乘，返回一个结果。来看两个非常简单的语句：

```
let x;
x = 3 * 5;
```

第一行是一个声明语句，声明了一个变量 *x*。当然，也可以合并这两行，不过这么

做会给当前的讨论造成困惑。有意思的是第二行：这一行中存在两个组合在一起的表达式。第一个表达式是 3×5，结果为 15。然后赋值表达式将 15 赋给变量 x。注意，赋值本身就是一个表达式，而表达式最终会解析成值。所以赋值表达式的值是什么呢？实践证明，赋值表达式的值就是所赋的值。所以，不仅是 x 被赋值为 15，而且整个表达式的值也是 15。因为赋值本身也是一个能解析成值的表达式，可以把它赋给其他变量。来看下面这个例子（可能有点无聊）

```
let x, y;
y = x = 3 * 5;
```

注意，这里有两个变量，*x* 和 *y*，它们的值都是 15。之所以可以这么做是因为乘法和赋值都是表达式。当 JavaScript 遇到这样的表达式的时候，必须先将它们分解，然后分别求值，就像这样：

```
let x, y;
y = x = 3 * 5;                  // 原始语句
y = x = 15;                     // 执行乘法表达式
y = 15;                         // 执行第一个赋值表达式；x 的值为 15，
                                // y 的值仍然是 undefined
15;                             // 执行第二个赋值表达式；y 的值为 15，
                                // 结果为 15，因为没有继续被使用，所以最终结果被丢弃
```

自然会有人会问：“avaScript 怎么知道要按照那样的顺序去执行？”也就是说，它首先会恰当地执行 y = x 的赋值，此时 y 的值为 undefined，然后对乘法求值，最后才赋值，而 y 仍然是 undefined，x 却变成 15 了。JavaScript 对表达式求值的顺序叫作*运算符优先级*，这部分内容会在本章进行讲解。

大部分表达式，比如乘法和赋值，都是*运算符表达式*。也就是说，一个乘法表达式是由一个*乘法运算符*（*）及两个*操作数*（相乘的数字本身也是表达式）组成。

有两种非运算符表达式，它们是*标识符表达式*（变量名和常量名）和*字面量表达式*：变量和常量本身就是一个表达式，字面量也是。理解这一点后会明白表达式的本质：任何能够产生值的语句都是表达式，所以不难理解变量、常量以及字面量都是表达式。

5.1 运算符

可以认为运算符是表达式的“动作”。也就是说，表达式会产生一个值；运算符就是产生这个值所要做的事情。两者的产出物都是一个值。下面以算术运算符作为切入点进行讨论。大家可能会觉得这有点像数学，因为大部分人都有使用算术运算符

的经验，所以从算术运算符切入会更直观：

运算符使用一个以上的操作数进行运算。例如，表达式 1 + 2，1 和 2 是操作数，+是运算符。如果操作数在技术上是正确的，它们通常也可以被称作参数。

5.2 算术运算符

JavaScript 的算术运算符如表 5-1。

表 5-1 算术运算符

运算符	说　明	示　例
+	加法（以及字符串连接）	3 + 2 // 5
-	减法	3 – 2 // 1
/	除法	3/2 // 1.5
*	乘法	3*2 // 6
%	取余数	3%2 // 1
-	一元负号	-x // 负 x; 如果 x 是 5，-x 就是-5
+	一元正号	+x // 如果 x 不是数字，会试图将它转换成数字
++	前置自增运算符	++x // 给 x 的值加 1，并计算新的值
++	后置自增运算符	x++ // 给 x 的值加 1，并在 x 自增前计算它的值 increments
--	前置自减运算符	--x // 给 x 的值减 1，并计算新的值
--	后置自减运算符	x-- // 给 x 的值减 1，并在 x 自减前计算它的值

记住，JavaScript 中所有的数字都是双精度的，这意味着整数做算术运算的时候（如 3/2），也有可能会返回小数（1.5）。

减法和一元否定都使用相同的符号（-），那么 JavaScript 是如何区分它们的呢？这个问题的答案非常复杂，超出了本书的范围。重点是要知道一元否定的运算优先级高于减法运算：

```
const x = 5;
const y = 3 - -x;       // y 的值为 8
```

一元正号的优先级也高于加法。它是一个不常用的运算符。该运算符通常是强制将

字符串转换成数字，或者调整那些否定的值：

```
const s = "5";
const y = 3 + +s;       // y is 8; 如果不使用一元加号运算符，结果为字符串"35"

// 使用不必要的一元加号运算符将代码排成一行
const x1 = 0, x2 = 3, x3 = -1.5, x4 = -6.33;
const p1 = -x1*1;
const p2 = +x2*2;
const p3 = +x3*3;
const p3 = -x4*4;
```

注意，在这些例子中，特意让一元否定、一元正号跟变量结合在一起使用。因为如果将它们直接跟数字一起使用，减号就会变成数字的一部分，这样就不是一个严格意义上的运算符了。

余数运算符返回除过之后的余数。比如，表达式 x % y，返回的结果是被除数 x 除以除数 y 后的余数。例如，`10 % 3` 结果为 1（10 是 3 的 3 倍余 1）。注意，负数取余的结果会带上被除数的符号，而不是除数的，这样就防止这个运算符成为一个真正的模运算符。然而，余数运算符通常不仅用在整数上，在 JavaScript 中，它也可以用在小数上。例如，`10 % 3.6` 等于 3（10 是 3.6 的两倍余了 2.8）。

自增运算符（++）实际上是将赋值和加法运算合二为一了。同样，自减运算符是赋值和减法运算的结合体。这是两个很有用的快捷方式，但要谨慎使用。因为如果将它们隐藏在一个表达式的深处，其副作用可能很难被察觉（变量被更改了）。理解前置和后置运算符的区别很重要。前置运算符修改了变量，表达式的值是修改后的值；后置运算符也修改了变量值，但是表达式的值没有变。下面看看读者是否能够猜出下面表达式的值（提示：自增和自减运算符的优先级高于加法运算符，在这个例子中，按照从左到右的顺序计算它们的值）：

```
let x = 2;
const r1 = x++ + x++;
const r2 = ++x + ++x;
const r3 = x++ + ++x;
const r4 = ++x + x++;
let y = 10;
const r5 = y-- + y--;
const r6 = --y + --y;
const r7 = y-- + --y;
const r8 = --y + y--;
```

试着在 JavaScript 控制台中运行这些代码，看看是否能猜对 `r1` 到 `r8` 的值，以及 x 和 y 的值在每一步的变化。如果在做这个练习时遇到问题，试着把它们写在一张纸

上，根据运算顺序添加括号，然后按照顺序执行运算操作，例如：

```
let x = 2;
const r1 = x++  +  x++;
//       ((x++) + (x++))
//       ( 2    + (x++))          从左到右计算；x 的值为 3
//       ( 2    +  3  )          x 的值变为 4
//              5                结果为 5；x 的值为 4
const r2 = ++x + ++x;
//       ((++x) + (++x))
//       ( 5    + (++x))          从左到右计算；x 的值为 5
//       ( 5    +  6  )          x 的值变为 6
//             11                结果为 11；x 的值为 6
const r3 = x++  +  ++x;
//       ((x++) + (++x))
//       ( 6    + (++x))          从左到右计算；x 的值为 7
//       ( 6    +  8  )          x 的值变为 8
//             14                结果为 14；x 的值为 8
//
// ... 等等
```

5.3 运算符优先级

还有一个需要理解的关键地方是，JavaScript 会按照运算符的优先级将表达式解析成值。这是了解 JavaScript 程序工作原理的至关重要的一步。

前面已经介绍了算术运算符，现在先暂停一下关于 JavaScript 中多种运算符的讨论，来看看它们的优先级。如果上过小学，那么大家就已经接触过运算符的优先级了，只是可能没有意识到。

大家是否还记得小学时解决过的问题（这里提前向那些对数学充满焦虑的同学表示歉意）：

```
8÷2+3× (4×2-1)
```

如果计算所得的答案是 25，说明能正确使用运算符优先级。也就是很清楚需要从括号开始，然后才是先乘除，后加减的顺序。

JavaScript 会运用类似的规则解析所有的表达式，而不仅是算术表达式。看到这里大家可能会很开心，因为 JavaScript 处理算术表达式的方式与小学学到的一样—可能还会有一些助记符，比如“PEMDAS”或者“Please Excuse My Dear Aunt Sally”

来帮助记忆。（译者注：PEMDAS: 括号（Parentheses），指数（Exponents），乘法（Multiplication），除法（Division），加法（addition），减法（subtraction）。）

在 JavaScript 中，除了算术运算符，还有很多其他运算符，需要记住很多东西，这可能不是一个人们喜闻乐见的消息。但也有好消息，因为这些运算符优先级与数学中的一样，括号具有最高优先级：对于一个给定表达式，如果不确定它的运算顺序，总是可以通过添加括号来保证表达式按照想要的顺序执行。

截至目前，JavaScript 总共有 56 种运算符，它们被归类到 19 个优先等级中。高优先级的运算符先于低优先级的运算符执行。尽管多年来，作者已经慢慢的记住了优先级表（并没有刻意去记），有时仍然会查阅它来巩固记忆，或者了解一门新的编程语言特性中哪些地方能对应上这些优先级。运算符优先级表见附录 B。

同一优先级的运算符按照从左到右的顺序执行。例如，乘法和除法的优先级都是 14，它们在执行时会按照从左到右的顺序，赋值运算符（优先级 13）则是从右到左。有了这些知识，就能弄清楚下面这些表达式的运算顺序了：

```
let x = 3, y;
x += y = 6*5/2;
// 我们会按照优先级顺序，用括号将下一个运算括起来
// 乘法和出发（优先级为 14，从做到右）
//       x += y = (6*5)/2
//       x += y = (30/2)
//       x += y = 15
// 赋值(优先级为 13，从右到左)：
//       x += (y = 15)
//       x += 15              (y 的值为 15)
//      18                    (x 的值为 18)
```

刚开始了解运算符优先级似乎会让人望而却步，但是它很快就会变成一件自然而然的事情。

5.4 比较运算符

比较运算符，顾名思义，可以用来比较两个不同的值。一般来说，有三种类型的比较运算符：严格相等、非严格（松散）相等、相关性。（通常不会把不相等看作一种不同的类型。不相等其实就是“相等”的反面，尽管为了方便起见，它有自己运算符）。

初学者最难理解的是严格相等和非严格相等之间的区别。下面将从严格相等开始，

因为建议大家一般情况下尽量使用严格相等。两个值严格相等的前提是它们引用了同一个对象，或者拥有相同的类型和相同的值（这里指基本类型）。严格相等的好处是，它有一个简单明确的规则，降低了错误的发生率，并且不容易引发误解。要确定两个值是否严格相等，使用严格相等运算符（===）或者严格不相等运算符（!==）即可。在开始看示例代码之前，先来看看非严格相等运算符。

两个值非严格相等的前提是它们属于同一个对象（到目前为止），或者可以强制转换成相同的值。后者在实际操作时会有一些麻烦。不过有时候它也很有用。例如，想知道数字 33 和字符串"33"是否相等，非严格相等运算符会返回 yes，但是严格相等运算符会返回 no（因为它们的类型不同）。这种情况下，非严格相等用起来更方便，但同时也产生一些不好的编码行为。正因为如此，在这里建议先将字符串转换成数字，然后用严格相等运算符去做比较。非严格相等运算符是==，它的否定形式是!=。如果想了解更多关于非严格相等运算符在使用时需要注意的问题和陷阱，作者推荐一本由 Douglas Crockford 写的书：*JavaScript: the Good Parts* (O'Reilly)（《JavaScript 语言精粹》）。

在使用非严格相等运算符时，产生的大多数问题都与 null、undefined、空字符串，以及数字 0 有关。如果能确定要比较的值里没有这些值，使用非严格相等运算符也会安全一些。不过，不要低估机械性习惯的力量。所以建议始终选择严格等式运算符作为默认选项，这样就不用在比较时思考运算符是否合适。不用打断自己的思路去思考非严格相等运算符是否安全或使用它的好处；只需要使用严格相等，然后继续专注的工作。如果后面发现严格相等运算符的运行结果不对，可以适当做一些类型转换，而避免使用可能会带来更多问题的非严格相等运算符。编程是一件困难的事情，就当帮帮自己，避免使用那些有问题的非严格相等运算符。

下面是一些关于严格相等和非严格相等的例子。注意，即便对象 a 和对象 b 的内容相同，它们依旧是不同的对象，所以不论使用严格相等还是非严格相等比较它们，返回的结果都是 false：

```
const n = 5;
const s = "5";
n === s;                          // false - 类型不同
n !== s;                          // true
n === Number(s);                  // true - 字符串"5" 转换成数字 5
n !== Number(s);                  // false
n == s;                           // true; 不推荐
```

```
n != s;                                    // false; 不推荐

const a = { name: "an object" };
const b = { name: "an object" };
a === b;                                   // false - 不是同一个对象
a !== b;                                   // true
a == b;                                    // false; 不推荐
a != b;                                    // true; 不推荐
```

关系运算符比较的是两个值的关系，它们仅仅适用于拥有自然排序特性的数据类型，比如，字符串（“a”排在“b”之前）、数字（0 排在 1 之前）。关系运算符有小于（<）、小于等于（<=）、大于（>）、大于等于（>=）：

```
3 > 5;          // false
3 >= 5;         // false
3 < 5;          // true
3 <= 5;         // true
5 > 5;          // false
5 >= 5;         // true
5 < 5;          // false
5 <= 5;         // true
```

5.5 比较数字

在给数字做一致性和相等性比较的时候，需要格外小心。

首先要注意，特殊的数值 NaN 与任何值都不相等，包括它自己（即，`NaN === NaN` 和 `NaN == NaN` 都是 `false`）。如果想测试某个数字是否为 NaN，可以使用内置函数 isNaN：如果 x 是 NaN，`isNaN(x)` 会返回 `true`，反之返回 `false`。

还记得在 JavaScript 中，所有的数字都是双精度的吗？因为双精度都是近似值（这是不可避免的），所以对其进行比较时会遇到一些令人讨厌的意外情况。

如果比较的是整数（在 `Number.MIN_SAFE_INTEGER` 和 `Number.MAX_SAFE_INTEGER` 之间，包括两端的值），就可以放心地使用数字本身做比较。如果比较的是小数，最好使用关系运算符来测试它是否足够接近目标值。怎样才是足够接近呢？这其实取决于应用。JavaScript 中存在一个数值常量，`Number.EPSILON`。这个值非常小（大约是 `2.22e-16`），它通常代表在需要考虑两个数字差别时的差异程度。看看下面的例子：

```
let n = 0;
while(true) {
    n += 0.1;
```

```
    if(n === 0.3) break;
}
console.log('Stopped at ${n}');
```

运行这段代码，结果可能会让人非常惊讶：while 循环跳过了 0.3，无限地执行下去了。这是为什么呢？众所周知，0.1 并不能精确地表示一个双精度数值，它介于两个二进制小数之间。所以循环执行到第三次时，n 的值为 0.30000000000000004，判断条件返回 fasle，结束循环唯一的机会就这样消失了。

可以使用 Number.EPSILON 重写上面的循环，此时关系运算符可以让比较“更缓和”，从而成功跳出循环：

```
let n = 0;
while(true) {
    n += 0.1;
    if(Math.abs(n - 0.3) < Number.EPSILON) break;
}
console.log('Stopped at ${n}');
```

注意，用数字 n 减去目标（0.3），然后取绝对值（使用 Math.abs，第 16 章会讲解它）。还可以在这里做一个更简单的计算（比如，仅仅判断 n 是否大于 0.3），不过取绝对值来比较两个值是否足够接近是测试两个双精度数值是否相等的通用方法。

5.6 字符串连接

JavaScript 中，+运算符既可以用作数字的加法，也可以用作字符串连接（这点十分常见：但 Perl 和 PHP 这两门语言是典型的反例，它们没有将+用作字符串连接）。

JavaScript 会根据运算对象的类型来决定执行加法还是字符串连接。这两种的执行顺序都是从左到右。JavaScript 会从左到右地检测每一对操作数，如果其中有一个是字符串，它就会执行字符串连接。如果两个都是数字，它才会执行加法。看下面两个例子：

```
3 + 5 + "8"         // 结果为 "88"
"3" + 5 + 8         // 结果为 "358"
```

第一个例子中，JavaScript 首先执行加法（3+5）。然后按字符串连接计算（8 + "8"）。第二个例子中，首先按字符串连接计算（"3" + 5），而后依然是字符串连接（"35" + 8）。

5.7 逻辑运算符

大家熟悉的算术运算符可以计算无穷大的值（或者至少是一个非常大的值，毕竟计算机的内存是有限的），然而，逻辑运算符只关心布尔值，且只有两种值：true 或 flase。

在数学中或很多其他编程语言中，逻辑运算符只能操作或返回布尔值。而 JavaScript 允许操作非布尔类型的值，甚至可以返回非布尔类型的值。这并不意味着 JavaScript 对逻辑运算符的实现有误或者不严谨：如果只使用布尔值，那么结果也只会是布尔值。

在学习它们之前，需要了解 JavaScript 中非布尔值和布尔值的映射机制。

真值和假值

很多编程语言都有“真”值和“假”值的概念，甚至在连布尔值都不存在的 C 语言中，数字 0 代表 false，所有非 0 数字代表 true。JavaScript 有些类似，不过它可以使用任何类型，这样一来，就可以有效地将任意值归类为真值或者假值。下面的值在 JavaScript 中都代表 false：

- `undefined`。
- `null`。
- `false`。
- `0`。
- `NaN`。
- `''`（空字符串）。

除了上面的值，其他值都为真。由于为真的东西太多了，这里就不一一列举了。不过下面这些是需要掌握的：

- 所有对象（包括 `valueOf()` 方法返回 `false` 的对象）。
- 所有数组（空数组也是）。
- 仅仅包含空格的字符串（比如`" "`）。
- 字符串`"false"`。

字符串"false"的值为 true 这点会使一些人觉得困惑，但对于大多数人来说，这些区别是很容易理解的，通常也会容易记住和使用。有一个需要特别注意的情况是：空数组也为真。如果想让空数组 arr 的值为假，使用 arr.length（空数组返回 0，0 代表假）可以达到目的。

5.8 与、或和非

JavaScrit 支持三种逻辑运算符：与（&&）、或（||）、和非（!）。如果有一些数学背景，大家也许知道“与”表示同时满足，“或”表示满足一个即可，而“非”则用来取反。布尔值不像数字那样可能会存在一个无穷大的值，它只有两种可能的值，所以这些运算符通常用*真值表*来描述，该表完整地描述了它们的行为（见表 5-2 到表 5-4）。

表 5-2　　与（&&）的真值表

x	y	x && y
false	false	false
false	true	false
true	false	false
true	true	true

表 5-3　　或（||）的真值表

x	y	x || y
false	false	false
false	true	true
true	false	true
true	true	true

表 5-4　　非（!）的真值表

x	!x
false	true
true	false

纵观这些真值表，会发现，“与”为 true 的前提是两个值同时为 true，“或”为 false 的前提是两个值同时为 false。“非”很直观：它只是对值取反。

或运算符有时候又叫作“同或”，因为如果两个值都为 ture，结果就为 ture。同时还存在“异或”（或 XOR），当两个值为 true 时，它的值为 false。JavaScript 不支持 XOR 运算符，但它有一个位异或运算符（^），后面章节会有讲解。

如果要对变量 x 和 y 做异或 (XOR)运算，可以使用等价表达式 `(x || y) && x !== y`。

5.8.1 短路求值

如果仔细看与的真假表（表 5-2），会发现有一个快捷判断方式：如果 x 是 false，不管 y 的值是什么，结果都为 false。同样，对于 `x || y`，一旦 x 是 true，就不用再计算 y 的值了。JavaScript 也确实是这么做的，这种方式叫*短路求值*。

为什么短路求值如此重要呢？因为如果第二个操作数有*副作用*，短路后程序就不会执行它们了。一般来说，“副作用”在程序中是一件坏事，但它也并非总是这样：如果副作用是故意引入的，并且用法很清楚，那么它就不再是件坏事了。

在表达式中，自增、自减、赋值，以及函数调用都可以引发副作用。前文已经讲解了自增和自减运算符，来看一个例子：

```
const skipIt = true;
let x = 0;
const result = skipIt || x++;
```

例子中的第二行代码会直接将值存储在变量 `result` 中。因为第一个操作数 (`skipIt`) 是 `true`，所以 result 的值为 `true`。然而，有趣的是，由于短路求值，自增表达式没有执行，所以 x 的值还是 `0`。如果将 `skipIt` 改成 `false`，那么两个表达式都会被执行，所以自增表达式也会被执行。在这里，自增就是副作用。同样的事情也会发生在与运算符中：

```
const doIt = false;
let x = 0;
const result = doIt && x++;
```

同理，JavaScript 不会解析第二个包含自增表达式的操作数。因为第一个操作数为 `false`，执行到这里会被短路，所以 `result` 的值为 `false`，`x` 的值也没有变。如果把 `doIt` 的值改为 `true` 会发生什么呢？此时 JavaScript 会解析两个操作数，所以自增也会执行，而 `result` 的值变成了 `0`。等等，什么？`result` 为什么是 `0` 而不是 `false`？这个问题的答案将巧妙地把我们带入下一个主题。

5.8.2 非布尔值的逻辑运算符

如果使用布尔值做逻辑运算，结果只能为布尔值。如果使用非布尔值，能够确定结

果的那个值就是逻辑运算的结果，如下表 5-5 和表 5-6。

表 5-5　　非布尔值的真值表“与”（&&）

x	y	x && y
falsy	falsy	x(falsy)
falsy	truthy	x(falsy)
truthy	falsy	y (falsy)
truthy	truthy	y(truthy)

表 5-6　　非布尔值的真值表“或”（||）

x	y	x || y
falsy	falsy	y (falsy)
falsy	truthy	y(truthy)
truthy	falsy	x(truthy)
truthy	truthy	x(truthy)

注意，如果将结果转换成布尔值，转换后的结果与在布尔值上使用与和或的结果一致。逻辑运算符的操作中可以用一些“快捷方式”。下面就是一个很常见的例子：

```
const options = suppliedOptions || { name: "Default" }
```

记住，对象（即便是空对象）的值永远为真。所以如果 `suppliedOptions` 是一个对象，`options` 将引用 `suppliedOptions`。如果没有任何类型，在这种情况下，`suppliedOptions` 值将为 `null` 或 `undefined`，`options` 也会有一个默认值。

在使用非的时候，没有理由不返回一个布尔值，所以非运算符（!）永远返回布尔值，不管操作数是什么。如果操作数是真，则返回 false，反之返回 true。

5.8.3　条件运算符

条件运算符是 JavaScript 中唯一的三元运算符，顾名思义，它有三个操作数（其他运算符都是一个或两个）。条件运算符是一个等价于 `if...else` 语句的表达式。下面是一个条件运算符的例子：

```
const doIt = false;
const result = doIt ? "Did it!" : "Didn't do it.";
```

如果第一个操作数（本例中是问号之前的 `doIt`）为真，表达式会解析第二个操作数（问号和冒号之间），反之，表达式将解析第三个操作数（问号之后）。很多编程初学者会觉得它是 `if...else` 语句的复杂形式，但实际上它是一个表达式，这是一个很有用的特性：它可以跟其他表达式组合使用（比如，前一个例子中给 `result`

的赋值)。

5.8.4 逗号运算符

逗号运算符可以简单地将表达式组合起来：它会按顺序执行两个表达式，并返回第二个表达式的结果。如果想执行多个表达式，但只关心最后一个表达式的结果，使用逗号运算符就很方便。下面是一个简单的例子：

```
let x = 0, y = 10, z;
z = (x++, y++);
```

在这个例子中，x 和 y 都自增了，z 最后的值为 10（由 y++返回的结果）。注意，逗号运算符的优先级是最低的，这就是要用括号的原因：如果没有括号，z 的值就变成 0（x++的值），y 的值也会增加。组合表达式最常见的场景有两种：for 循环（见第 4 章），以及函数在返回前组合执行多种操作。

5.9 分组运算符

如前文所述，分组运算符（括号）除了修改或澄清运算符的优先级外，没有任何影响。因此，分组运算符是“最安全”的运算符，它只会影响运算的顺序。

5.9.1 位运算符

位运算符允许在数字的每个二进制位上执行操作。如果读者没有类似 C 语言这样偏底层的编程语言经验，或者对计算机内部如何存储数字不太了解，可能需要先了解相关的知识（可以跳过这部分内容，因为很少有应用使用到位运算符）。位运算符将其操作数当成二进制补码格式的 32 位有符号整型数字。因为在 JavaScript 中，所有的数字都是双精度的，JavaScript 在执行位运算前会先将数字转换成 32 位的整型，并在返回结果之前转换回来。

位运算符与逻辑运算符的共同点在于它们也是执行逻辑操作（与、或、非、异或），不同的是它们在整型的每一个二进制位上进行运算。如表 5-7 所示，它们还包含了能将二进制位进行位移的位移运算符。

表 5-7 位运算符

运 算 符	描 述	示 例
&	位与	0b1010 & 0b1100 // 结果: 0b1000
|	位或	0b1010 | 0b1100 // 结果: 0b1110
^	位异或	0b1010 ^ 0b1100 // 结果: 0b0110

续表

运 算 符	描 述	示 例
~	位非	~0b1010 // 结果: 0b0101
<<	左移位	0b1010 << 1 // 结果: 0b10100 0b1010 << 2 // 结果: 0b101000
>>	符号传播右移位	(参见下文)
>>>	补零右移位	(参见下文)

注意，左移位实际上是乘以 2，右位移则是除以 2 然后舍去尾数。

存在两种补码，最左边的二进制位为 1 时表示负数，0 表示正数，它们都可以执行右移位。以–22 为例。如果想要获取二进制表示法，以正数 22 开始，取反（反码）再加 1（补码）。

```
let n = 22          //  32 位二进制数:    00000000000000000000000000010110
n >> 1              //                    00000000000000000000000000001011
n >>> 1             //                    00000000000000000000000000001011
n = ~n              // 取反:               11111111111111111111111111101001
n++                 // 加 1:               11111111111111111111111111101010
n >> 1              //                    11111111111111111111111111110101
n >>> 1             //                    01111111111111111111111111110101
```

除非从事的是硬件编程，或者想更好理解数字在计算机内部的表示原理，否则几乎用不上位运算符（一般被戏称为“玩位（味）”）。一个与硬件无关的使用场景可能是用二进制位来高效存储“标记位”（布尔值）。

例如，考虑一个 Unix 风格的文件权限：读、写和执行。一个给定的用户可以任意组合这三种权限，以达到预期目标。因为存在三种标志，所以需要三个二进制位来存储这些信息。

```
const FLAG_READ 1              // 0b001
const FLAG_WRITE 2             // 0b010
const FLAG_EXECUTE 4           // 0b100
```

有了位运算符，可以在一个单一的数值中组合、切换，以及检测每一个二进制位标记。

```
let p = FLAG_READ | FLAG_WRITE;          // 0b011
let hasWrite = p & FLAG_WRITE;           // 0b010 - truthy
let hasExecute = p & FLAG_EXECUTE;       // 0b000 - falsy
p = p ^ FLAG_WRITE;                      // 0b001 - 写标记开关 (关闭)
p = p ^ FLAG_WRITE;                      // 0b011 - 写标记开关 (开启)

// 我们甚至可以确定在一个表达式中确定多个标记
```

```
const hasReadAndExecute = p & (FLAG_READ | FLAG_EXECUTE);
```

注意常量 hasReadAndExecute，这里不得不使用分组运算符，因为与运算符比或运算符的优先级高，所以用括号来强制优先执行或运算。

5.9.2 类型判断运算符

类型判断运算符返回一个字符串形式的类型名称。遗憾的是，这个运算符并没有精确地对应 JavaScript 中的七种数据类型（undefined，null，布尔，数字，字符串，符号，及对象），这一点造成了无休止的诟病和疑惑。

类型判断运算符有个奇怪的地方，这通常被认为是一个错误：typeof null 的结果是 "object"，很明显，null 不可能是对象（它是基本类型）。这个问题由来已久，且其起因不那么有趣，它已经多次被建议修改，但是因为很多现存代码中都用到这个运算符，所以它就一直保留在 JavaScript 的语言规范中。

类型判断运算符另外一个经常受到批判的地方是它不能区分数组和非数组对象。它能正确识别函数对象（函数也是一种特殊的对象），但是 typeof []的结果却是 "object"。

表 5-8 列出了 typeof 所有可能的返回值。

表 5-8　　typeof 返回值

表 达 式	返 回 值	备　注
typeof undefined	"undefined"	
typeof null	"object"	很不幸，但这是事实
typeof {}	"object"	
typeof true	"boolean"	
typeof 1	"number"	
typeof ""	"string"	
typeof Symbol()	"symbol"	ES6 新特性
typeof function(){}	"function"	

因为 typeof 是一个运算符，不需要搭配括号。也就是说，判断变量 x 的类型，可以直接使用 typeof x，而不是 typeof(x)。后者也是合法的，括号只是用来构建不必要的表达式组。

5.9.3 void 运算符

void 运算符只有一个用途：计算它的操作数并返回 undefined。听起来好像没什么用，实际上确实没用。它可以强制表达式返回 undefined，但作者从来没有遇到过这种情况。本书提及它的唯一理由是可能偶尔会碰到它被用做 HTML 标签<a>的 URI。

```
<a href="javascript:void 0">Do nothing.</a>
```

虽然不推荐这样做，但它确实是一种常见的用法。

5.9.4 赋值运算符

赋值运算符很直观：它将一个值指派给某个变量。等号左边（有时候叫左值）必须是变量、属性或者数组元素。也就是说，左边必须是能够持有值的东西（从技术上讲，给常量赋值是声明的一种，而非赋值）。

回想一下本章一开始所讲过的，赋值本身是一个表达式，所以它的返回值就是被赋的那个值。这就可以链式赋值，就像是从其他表达式赋值一样：

```
let v, v0;
v = v0 = 9.8;          // 链式赋值；首先给 v0 赋值 9.8，然后给 v 赋值 9.8

const nums = [ 3, 5, 15, 7, 5 ];
let n, i=0;
// 注意 while 条件中的赋值；n 得到 nums[i] 的值后，整个赋值表达式也会计算该值，所以
// 可以进行比较：
while((n = nums[i]) < 10, i++ < nums.length) {
    console.log('Number less than 10: ${n}.');
}
console.log('Number greater than 10 found: ${n}.');
console.log('${nums.length} numbers remain.');
```

注意第二个例子，由于赋值运算符的优先级低于关系运算符，所以必须使用分组运算符。

除了普通的赋值运算符，还有一些赋值运算符能够非常方便地一次性完成与运算和赋值。就像普通的赋值运算符一样，这些运算符的计算结果就是最终要赋的那个值。表 5-9 中总结了这些便捷的赋值运算符。

表 5-9　　运算中赋值

运　算　符	等价表达式
x += y	x = x + y
x -= y	x = x + y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x <<= y	x = x << y
x >>= y	x = x >> y
x >>>= y	x = x >>> y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y

5.10　解构赋值

ES6 中一个倍受欢迎的新特性是解构赋值，它允许将一个对象或者数组“分解”成多个单独的值。以对象的解构开始：

```
// 一个普通的对象
const obj = { b: 2, c: 3, d: 4 };

// 对象解构赋值
const {a, b, c} = obj;
a;                          // undefined: obj 中不存在属性"a"
b;                          // 2
c;                          // 3
d;                          // 引用 error: "d" 未定义
```

解构一个对象时，变量名必须与对象中的属性名保持一致（数组解构只能指派那些作为标识符的属性名）。这个例子中，由于 a 跟对象的任何属性都不匹配，所以它的值为 undefined。同样，因为声明中没有指定 d，所以它没有被赋值。

上例中，在同一个语句中既有声明也有赋值。对象解构也可以在一个赋值语句中完成，但是这个语句必须被括号括起来。否则，JavaScript 解释器会认为左边的部分是一个代码块。

```
const obj = { b: 2, c: 3, d: 4 };
let a, b, c;
```

```
// 会报错：
{a, b, c} = obj;

// 正常运行：
({a, b, c} = obj);
```

数组解构时，可以给数组的元素任意指定变量名（按顺序）。

```
// 一个普通数组
const arr = [1, 2, 3];

// 数组解构赋值
let [x, y] = arr;
x;                          // 1
y;                          // 2
z;                          // 错误：z 未定义
```

这个例子中，x 接收了数组第一个元素，y 接收了第二个元素，所有没被接收的元素都被丢弃了。也可以把所有剩下的元素放入一个新的数组中，展开运算符（...）可以完成这个功能，在第 6 章会学习这个运算符。

```
const arr = [1, 2, 3, 4, 5];

let [x, y, ...rest] = arr;
x;                                  // 1
y;                                  // 2
rest;                               // [3, 4, 5]
```

这个例子中，x 和 y 接收前两个元素，变量 rest 接收所有剩下的元素（可以将 rest 命名为任何想要的名字）。数组解构可以很方便地交换变量的值（以前可能需要一个中间变量来完成）。

```
let a = 5, b = 10;
[a, b] = [b, a];
a;                                  // 10
b;                                  // 5
```

数组解构不仅适用于数组，还适用于任何可迭代的对象（在第 9 章会学习）。

在这些简单的例子中，可能简单直接地给变量赋值比解构来得更容易。不过，当对象和数组的值需要从其他地方获取时，解构就派上用场了，它可以轻松地获取某个元素。大家将在第 6 章中看到它大显身手。

5.11 对象和数组运算符

对象、数组和函数都有一个特殊的运算符集合。有一些我们已经见过了（比如成员访问和计算机成员访问运算符），剩下的会陆续出现在第 6、8 和第 9 章。为了本章内容的完整性，这里将它们总结在表 5-8 中。

表 5-10 对象和数组运算符

运 算 符	描 述	章 节
.	成员访问	第 3 章
[]	计算机成员访问	第 3 章
in	判断属性是否存在	第 9 章
new	实例化对象	第 9 章
instanceof	原型链测试	第 9 章
...	展开运算符	第 6 章，第 8 章
delete	删除运算符	第 3 章

5.12 模板字符串中的表达式

模板字符串在第 3 章中已经介绍过，它可以将任意表达式的值注入到字符串中。第 3 章中的例子是用模板字符串表示当前温度。如果要表示温差，或者要表示华氏温度而非摄氏度，应该怎么做呢？可以在模板字符串中使用表达式：

```
const roomTempC = 21.5;
let currentTempC = 19.5;
const message = 'The current temperature is' +
    '${currentTempC-roomTempC}\u00b0C different than room temperature.';
const fahrenheit =
    'The current temperature is ${currentTempC * 9/5 + 32}\u00b0F';
```

这里又一次看到了表达式所带来的优美对称性。由于变量本身就是一种简单的表达式，所以可以在模板字符串中使用变量。

5.13 表达式和控制流模式

在第 4 章中，讲解了一些常见的控制流模式。至此已经知道有些表达式会影响控制流（三元表达式和短路求值），接下来看看一些其他的控制流模式。

5.13.1 将 if...else 语句转化成条件表达式

如果 if...else 语句用于确定某个值，不论该值是赋值语句的一部分，还是表达式的一小部分，再或者是函数的返回值，一般建议使用条件运算符来代替。条件运算符让代码更紧凑易读。例如：

```
if(isPrime(n)) {
    label = 'prime';
} else {
    label = 'non-prime';
}
```

可以被写成：

```
label = isPrime(n) ? 'prime' : 'non-prime';
```

5.13.2 将 if 语句转化成短路求值的逻辑或（||）表达式

好比一个用来确定值的 if...else 语句可以很容易转换成条件表达式。同样，类似的 if 语句也能容易地转换成短路逻辑或（||）表达式。只是后者不如前者的优势明显，但编程中会经常见到它，所以了解它也是有好处的。例如：

```
if(!options) options = {};
```

可以容易地转换成：

```
options = options || {};
```

5.14 小结

与大部分现代编程语言一样，JavaScript 也有一个广泛而实用的运算符集合，这些运算符构成了基本的数据操作集合。有一些可能在平时几乎用不上，如位运算符。还有其他的，比如成员访问运算符，甚至不会把它当做运算符（当试图解决一个困难的运算符优先级问题时，它就能派上用场了）。

赋值、算术、比较，以及布尔运算符是最常见的运算符，编程中将会频繁地使用它们，所以在继续学习之前，请确保自己已经完全掌握这些运算符的使用了。

第6章

函数

函数是一组语句的集合，它是一个独立运行的程序单元。本质上，它是一段子程序。函数是 JavaScript 的核心，本章将介绍函数的基本用法和机制。

每个函数都有一个函数体，它是构成该函数的一组语句集合。

```
function sayHello() {
    // 这是函数体；它从一个左花括号开始...

    console.log("Hello world!");
    console.log("¡Hola mundo!");
    console.log("Hallo wereld!");
    console.log("Привет мир!");

    // ...到右花括号结束
}
```

这是一个典型的函数声明：声明了一个名为 sayHello 的函数。仅仅声明函数并不会去执行函数体。如果运行这个例子，console 中并不会打印出多个“Hello, World”。为了调用（也称为运行，执行，请求，或者调度）这个函数，需要在函数名后跟上一对圆括号：

```
sayHello();         // 在控制台中打印出各种语言的"Hello, World!"
```

调用、请求、执行（或着运行），这些叫法可以互换，本书中会全部用上，以便更加熟悉它们。在某些特殊的场景或者编程语言中，这些叫法可能会有些差别，但通常来说，它们表示的意思都是一样的。

6.1 返回值

函数调用是一种表达式，由前面的学习可知，表达式会产生一个值。那么函数调用会产生什么值呢？这里就要说到函数的返回值了。在函数体中，return 关键字会立即结束函数并且返回一个特定值，这就是函数调用产生的值。我们修改一下上例，不再向控制台写入值，而是返回一句问候语：

```
function getGreeting() {
   return "Hello world!";
}
```

现在我们再调用 getGreeting 方法就会得到一个返回值：

```
getGreeting();          // "Hello, World!"
```

如果没有明确指定 return 语句，返回值将会是 undefined。函数可以返回任何类型的值。给读者留一个小练习：试着写一个函数，getGreetings，返回一个包含各种语言的“Hello，World”的数组。

6.2 引用调用

在 JavaScript 中，函数是一个对象，像其他对象一样，可以被传递和赋值。所以理解函数调用和引用之间的差别很重要。当在函数名后面添加圆括号时，JavaScript 就知道要调用它，然后执行函数体，最后返回结果。如果没有圆括号，仅仅是在引用这个函数，并没有调用它。试着在 console 中执行下面的代码：

```
getGreeting();          // 打印"Hello, World!"
getGreeting;            // 函数 getGreeting()
```

能够像引用其他值一样引用函数这点能给编程语言带来很大的灵活性。比如，可以把一个函数赋值给一个变量，就可以通过其他名字来调用这个函数：

```
const f = getGreeting;
f();                    // "Hello, World!"
```

或者把函数赋值给一个对象的属性：

```
const o = {};
o.f = getGreeting;
o.f();                  // "Hello, World!"
```

甚至是把函数添加到数组里：

```
const arr = [1, 2, 3];
arr[1] = getGreeting;       // 现在的arr是 [1, function getGreeting(), 2]
arr[1]();                   // "Hello, World!"
```

最后一个例子清楚地表示了圆括号的作用。如果 JavaScript 遇到圆括号跟在一个值后面，这个值就会被当做函数，同时调用这个函数。在上例中，arr[1]是一个计算值的表达式。这个值后面有一对圆括号，这就是在告诉 JavaScript 这个值是一个函数，并且需要被执行。

如果在值不是函数的变量后面添加圆括号，JavaScript 就会报错。比如，执行“whoops()”的结果就是 TypeError 的错误："whoops" is not a function.

6.3 函数参数

现在已经知道如何调用一个函数并在函数外获得它的返回值，那么如何给函数内部传值呢？给函数调用传值的主要途径称为函数参数。参数是指那些在函数调用结束后就不再存在的变量。现在我们来看看这个函数，它有两个数字作为参数，返回这两个数字的平均值。

```
function avg(a, b) {
   return (a + b)/2;
}
```

在这个函数声明中，a 和 b 称为形参。当函数被调用时，形参会被赋值然后变成实参。

```
avg(5, 10);                 // 7.5
```

在这个例子中，形参 a 和 b 分别被赋值 5 和 10，成为实参（跟变量赋值很相似，特殊的是它们在函数内）。

初学者很容易犯迷糊的一个概念是：参数只存在于函数内部，即使它们在函数外有相同变量的名字。看下面这个例子：

```
const a = 5, b = 10;
avg(a, b);
```

这里的变量 a 和 b 是分开的，这与函数 avg 参数中的 a 和 b 是完全不同的，即使它们的名字一样。在调用函数时，函数的参数只会接收传入的值，而不是变量本身。再看以下代码：

```
function f(x) {
    console.log('inside f: x=${x}');
    x = 5;
    console.log('inside f: x=${x} (after assignment)');
}

let x = 3;
console.log('before calling f: x=${x}');
f(x);
console.log('after calling f: x=${x}');
```

运行这个例子，你会看到：

```
before calling f: x=3
inside f: x=3
inside f: x=5 (after assignment)
after calling f: x=3
```

这里需要理解的是：在函数中给 x 赋值并不会影响到函数外的变量 x，因为它们是两个不同的实体，只是名字碰巧相同而已。

不论何时给函数中的参数赋值，都不会影响任何函数外的变量。不过，如果在函数内修改了一个对象类型的变量，那么这个改动在函数外也会生效：

```
function f(o) {
    o.message = 'set in f (previous value: '${o.message}')';
}
let o = {
    message: "initial value"
};
console.log('before calling f: o.message="${o.message}"');
f(o);
console.log('after calling f: o.message="${o.message}"');
```

上面的代码会输出如下结果：

```
before calling f: o.message="initial value"
after calling f: o.message="set in f (previous value: 'initial value')"
```

这个例子中，可以看到在 f 函数内部修改了 o，这些改动也会影响函数外的 o 对象。这里强调了基本类型和对象之间的主要区别。基本类型不能被修改（可以改变基本类型所赋值的变量，但是基本类型本身的值不会改变）。从另一方面来说，对象是可以被修改的。

需要清楚的是，函数内的 o 和函数外的 o 是完全不同的，但是它们都引用了同一个对象。可以通过下面的代码看出它们的不同之处：

```
function f(o) {
```

```
    o.message = "set in f";
    o= {

        message: "new object!"
    };
    console.log('inside f: o.message="${o.message}"(after assignment)');
}
let o = {
    message: 'initial value'
};
console.log('before calling f: o.message="${o.message}"');
f(o);
console.log('after calling f: o.message="${o.message}"');
```

如果执行这个例子，会得到如下结果：

```
before calling f: o.message="initial value"
inside f: o.message="new object!" (after assignment)
after calling f: o.message="set in f"
```

理解这段代码的关键在于明白参数 o（函数内部）和变量 o（函数外）是不同的。当 f 函数被执行时，它们两个都指向了同一个变量，不过当 o 在 f 函数内被赋值时，它指向了一个全新的、完全独立的对象。而函数外的 o 依旧指向原始对象。

可以把 JavaScript 中的基本类型想象成计算机科学中的值类型，因为它们在赋值的过程中都对原始值进行了复制。对象则称为引用类型，因为当它们在赋值时，被赋值的变量和原来的变量都引用了同一个对象（也就是说，它们拥有同一个对象的引用）。

6.3.1 参数会让函数有所区别吗

在很多编程语言中，一个函数的签名会包含它的参数列表。比如，在 C 语言中，函数 f()（没有参数）和函数 f(x)（包含一个参数）是两个不同的函数，同时它们又与函数 f(x, y)（包含两个参数）不同。而 JavaScript 则对此不作区分，当有一个名为 f 的函数时，调用时可以给它 0 个、1 个，甚至是 10 个参数，不管调用时有多少参数，都始终调用了同一个函数。

这里暗含的思想是，可以使用任意多个参数去调用任何一个函数。如果在调用时没有传入参数，函数就会接收到一个 undefined 的参数。

```
function f(x) {
   return 'in f: x=${x}';
}
f();      // "in f: x=undefined"
```

本章的后面会讲到如何处理那些传入的参数多于函数定义的参数的情况。

6.3.2 解构参数

就像解构赋值（详情见第 5 章）一样，JavaScript 中也有*解构参数*（毕竟参数跟定义变量很类似）。考虑把一个对象解构到不同的变量中：

```
function getSentence({ subject, verb, object }) {
    return '${subject} ${verb} ${object}';
}

const o = {
   subject: "I",
   verb: "love",
   object: "JavaScript",
};

getSentence(o);         // "I love JavaScript"
```

在解构赋值中，参数属性名必须是字符串标识符，而且如果传入的对象不存在与参数属性名匹配的属性，该属性将会接受一个 undefined 值。

甚至可以解构一个数组：

```
function getSentence([ subject, verb, object ]) {
   return '${subject} ${verb} ${object}';
}

const arr = [ "I", "love", "JavaScript" ];
getSentence(arr);           // "I love JavaScript"
```

最后，可以使用展开操作符（...）来收集任何多出来的参数：

```
function addPrefix(prefix, ...words) {
    // 稍后会用一种更好的方法来实现!
    const prefixedWords = [];
    for(let i=0; i<words.length; i++) {
        prefixedWords[i] = prefix + words[i];
    }
    return prefixedWords;
}

addPrefix("con", "verse", "vex");   // ["converse", "convex"]
```

注意，如果在函数定义中使用展开操作符，它必须是最后一个参数。如果在它之后还有参数，JavaScript 就无法分辨出哪些参数使用展开参数，哪些使用它之后的参数。

在 ES5 中，类似的功能可以被只存在函数体中的特殊变量所实现：参数。这个变量并不是严格意义上的数组，它是一个“类数组”的对象，经常需要特殊处理，或者转化成合理的数组。ES6 中的展开数组解决了这个问题，所以相比参数变量（现在依然存在），更应该使用展开参数。

6.3.3 默认参数

ES6 中的一个新特性是可以指定参数的默认值。通常，如果没有给参数传值，它的值将会是 undefined。默认值则可以给这些参数指定其他的默认值。

```
function f(a, b = "default", c = 3) {
    return '${a} - ${b} - ${c}';
}

f(5, 6, 7);                 // "5 - 6 - 7"
f(5, 6);                    // "5 - 6 - 3"
f(5);                       // "5 - default - 3"
f();                        // "undefined - default - 3"
```

6.4 函数作为对象属性

当函数作为一个对象的属性时，通常被称为*方法*，从而跟一般函数区分开来（稍后会介绍更多关于*函数*和*方法*之间的细微差别）。前面已在第 3 章中学习了如何将方法添加到一个已有的对象中。也可以在对象初始化的时候添加方法：

```
const o = {
   name: 'Wallace',                          // 简单属性
   bark: function() { return 'Woof!'; },     // 函数属性（或者方法属性）
}
```

ES6 为了方法引入了一个新的快捷语法。以下代码等效于上面的例子：

```
const o = {
   name: 'Wallace',                          //简单属性
   bark() { return 'Woof!'; },               //函数属性（或者方法属性）
}
```

6.5 this 关键字

在函数体中，有一个特殊的只读字段 this。这个关键字通常与面向对象编程一起出现，在第 9 章会了解到更多关于 this 在面向对象中的用法。然而，在 JavaScript

中，this 关键字还是有很多使用场景的。

this 关键字通常关联那些作为对象属性的函数。当方法被调用时，this 关键字的值就是被调用的对象：

```
const o = {
   name: 'Wallace',
   speak() { return 'My name is ${this.name}!'; },
}
```

当调用 o.speak() 的时候，this 关键字跟 o 进行了绑定：

```
o.speak();            // "My name is Wallace!
```

如何绑定 this 是由*方法如何被调用所决定的，而非函数定义所决定*，理解这一点非常重要。也就是说，在这里，this 绑定到 o 上并不是因为 speak 是 o 的属性，而是因为它直接由 o 调用（o.speak）。如果把同一个函数赋值给一个对象，会发生什么呢？

```
const speak = o.speak;
speak === o.speak;                    // 为真；两个变量都指向了同一个函数
speak();                              // "My name is !"
```

由于调用函数的方式，JavaScript 并不知道原始函数是在 o 中定义的，所以 this 将会绑上 undefined。

如果以一种不清楚如何绑定 this 的方式来调用函数（就像前面的例子中调用函数变量 speak），this 的绑定就会变得很复杂。这取决于是否使用了严格模式，以及函数在哪里被调用。之所以刻意掩盖了这些细节是因为作者认为最好避开这种情况。如果想深入了解，可以参阅关于代码格式化的 MDN 文档。

*方法*这个叫法最初是与面向对象编程结合在一起的，在本书中，用它来表示那些作为对象属性的函数，同时它们被设计成可以直接从对象实例（就像 o.speak()）中调用。如果一个函数不使用 this，JavaScript 就会将它当成一个普通函数，而不管它在哪里定义。

关于 this 变量的一个细节是，当在嵌套函数中访问它时经常会出错。下面这个例子就是在方法中使用了辅助函数：

```
const o = {
    name: 'Julie',
    greetBackwards: function() {
       function getReverseName() {
```

```
            let nameBackwards = '';
            for(let i=this.name.length-1; i>=0; i--) {
                nameBackwards += this.name[i];
            }
            return nameBackwards;
        }
        return '${getReverseName()} si eman ym ,olleH';
    },
};
o.greetBackwards();
```

这里使用了嵌套函数，getReverseName，是为了反序列名字。但是，getReverseName 并没有像所期望的那样运行：当调用 o.greetBackwards() 时，JavaScript 像预期那样给 this 绑定了值。然而，在 greetBackwards 中调用 getReverseName 函数时，this 却绑到了其他地方[①]。一个常用的解决方案给 this 赋另一个变量：

```
const o = {
    name: 'Julie',
    greetBackwards: function() {
        const self = this;
        function getReverseName() {
            let nameBackwards = '';
            for(let i=self.name.length-1; i>=0; i--) {
                nameBackwards += self.name[i];
            }
            return nameBackwards;
        }
        return '${getReverseName()} si eman ym ,olleH';
    },
};
o.greetBackwards();
```

这是一个常用的技巧，通过它 this 会被赋值给 self。箭头函数也可以用来解决这个问题，本章的后面会讲到。

6.6 函数表达式和匿名函数

至此，已经详细介绍了*函数声明*，也就是声明函数体（定义函数做了什么）和标识符（便于后期调用）的地方。JavaScript 也支持*匿名函数*，这种函数没有标识符。

大家可能很好奇一个没有标识符的函数能有什么用。没有标识符，怎么调用它呢？

① this 会被绑定一个到全局对象还是 undefined 取决于是否使用了严格模式。本书中不会覆盖到所有细节，因为你应该避开这种情况。

答案就在于理解*函数表达式*。大家知道一个表达式可以计算值，而函数也可以计算值。函数表达式是声明（基本都没有名字）函数的一种简单方式。函数表达式可以用来赋值（类似给它一个标识符），或者立刻被调用[①]。

除了可以省略函数名，函数表达式在语法上跟函数定义是完全一样的。来看这样一个例子，定义一个函数表达式然后把它赋给一个变量（这实际上等效于函数定义）：

```
const f = function() {
    // ...
};
```

这段代码与使用普通方式声明的函数带来的结果是一样的：标识符 f 可以用来引用函数。和普通的函数声明一样，可以通过 f() 来调用函数。唯一的不同是这里实际上创建了一个匿名函数（通过使用函数表达式）并把它赋给了一个变量。

匿名函数的使用范围很广：可以作为其他函数或方法的参数，还可以用来创建对象的函数属性。将在本书的后面介绍这些用法。

前面提到过在函数表达式中函数名*不是必须的*...那么当把一个具名的函数赋给一个变量会发生什么呢（以及为什么我们要这么做）？比如：

```
const g = function f() {
    // ...
}
```

当一个函数被这样创建时，g 这个名字具有较高的优先级，在引用函数时（在函数外），应该使用 g；如果使用 f 引用函数则会得到一个 undefined variable 的错误。回到这个例子中，如果直接引用 f 会出错，那么为什么要这样做呢？因为这样就可以在函数内部引用它本身（称为*递归调用*）：

```
const g = function f(stop) {
    if(stop) console.log('f stopped');
    f(true);
};
g(false);
```

在函数内部，使用 f 来引用这个函数，在函数外部则用 g。给一个函数定义两个毫不相关的名字并没有什么特殊原因，只是为了说明函数表达式是如何工作的。

由于函数声明和函数表达式看起来是一样的，大家可能会好奇 JavaScript 是如何区分它们的（或者它们之间究竟有什么不同）。答案是取决于*上下文*：如果函数声明

① 也称为“立即执行的函数表达式”（IIFX），第 7 章会详细介绍。

作为一个表达式使用，那么它就是函数表达式，否则就是函数声明。

其中的区别大都是理论上的，平时并不需要特别注意这些区别。当定义了一个具名函数并准备之后调用它，大部分时候会想都不想地去声明一个函数，而当需要创建一个函数来赋值或者将其传入其他函数中时，自然会使用函数表达式。

6.7 箭头符号

ES6 引入了一个新的语法叫作*箭头符号*（也称为“大箭头”语法，因为这里的箭头使用了等于号，而非横线）。从本质上说这也是一种语法糖（却跟语法糖有很大的区别，稍后会讲到），它可以减少敲击 function 这个单词的次数，以及敲击花括号的次数。

箭头函数允许使用以下 3 种方式来简化语法：

- 可以省略 `function` 这个单词。
- 如果函数只有一个参数，则可以省略花括号。
- 如果函数体是一个单独表达式，则可以省略花括号和返回语句。

箭头函数大多是匿名的。但仍然可以把它们赋给变量，不过不能再使用 `function` 关键字创建一个具名函数。

以下这些例子都与函数表达式等效：

```
const f1 = function() { return "hello!"; }
// OR
const f1 = () => "hello!";

const f2 = function(name) { return 'Hello, ${name}!'; }
// OR
const f2 = name => 'Hello, ${name}!';

const f3 = function(a, b) { return a + b; }
// OR
const f3 = (a,b) => a + b;
```

这些例子看起来好像有点假。通常，如果需要一个具名函数，只要使用一般的函数声明就行了。当需要创建或者传递一个匿名函数时，箭头函数是最实用的，从第 8 章开始大家会经常见到这种用法。

箭头函数跟普通函数之间的主要区别是：`this` 是跟语句绑定起来的，就像其他变

量一样。回想一下前面的 greetBackwards 的例子。通过箭头函数，可以在函数体中使用 this：

```
const o = {
   name: 'Julie',
   greetBackwards: function() {
      const getReverseName = () => {
         let nameBackwards = '';
         for(let i=this.name.length-1; i>=0; i--) {
            nameBackwards += this.name[i];
         }
         return nameBackwards;
      };
      return '${getReverseName()} si eman ym ,olleH';
   },
};
o.greetBackwards();
```

箭头函数与普通函数还有两个小区别：不能把它们当作对象构造器（详情见第 9 章），同时，指定的参数变量在箭头函数中也不生效（这里就没有必要使用展开操作符了。）

6.8 调用、请求和绑定

上面已经知道 this 绑定的一般方式（跟其他面向对象语言一样）。然而，JavaScript 还允许指定 this 绑定的值，不论函数在何处被如何调用。从 call 讲起，call 方法在所有函数上都可用，它允许使用指定的 this 来调用函数。

```
const bruce = { name: "Bruce" };
const madeline = { name: "Madeline" };

// this function isn't associated with any object, yet
// it's using 'this'!
function greet() {
    return 'Hello, I'm ${this.name}!';
}

greet();               // "Hello, I'm !" - 'this' 没有绑定任何值
greet.call(bruce);     // "Hello, I'm Bruce!" - 'this' 绑定了 'bruce'
greet.call(madeline); // "Hello, I'm Madeline!" -'this' 绑定了 'madeline'
```

可以看到 call 方法允许在调用函数时给 this 绑定一个对象，就好像有一个方法在做这件事。call 方法的第一个参数是想要的绑定的值，剩下的参数则变成了要调用的函数的参数：

```
function update(birthYear, occupation) {
    this.birthYear = birthYear;
    this.occupation = occupation;
}

update.call(bruce, 1949, 'singer');
    // 现在的bruce是 { name: "Bruce", birthYear:1949,occupation:"singer"}
update.call(madeline, 1942, 'actress');
    // 现在的madeline是 { name: "Madeline", birthYear: 1942,
    // occupation: "actress" }
```

除了处理函数参数的方式不同，apply 与 call 基本是一致的。call 会像一般的函数一样直接获取参数。apply 则以数组的方式获取参数：

```
update.apply(bruce, [1955, "actor"]);
    // 现在的bruce是 { name: "Bruce", birthYear: 1955,occupation:"actor"}
update.apply(madeline, [1918, "writer"]);
    // 现在的madeline是 { name: "Madeline", birthYear: 1918,
    //    occupation: "writer" }
```

apply 比较适合用于将数组作为函数参数的场景。一个经典的例子就是找出数组内的最大值和最小值。JavaScript 内建的 Math.max 和 Math.min 函数可以接收任意一串数字作为参数，并分别返回最大值和最小值。可以用 apply 和一个已有的数组调用这些函数：

```
const arr = [2, 3, -5, 15, 7];
Math.min.apply(null, arr);    // -5
Math.max.apply(null, arr);    // 15
```

注意，这里给 this 传了一个 null。这是因为 Math.max 和 Math.min 并不使用 this，所用不管传给它什么值都没关系。

通过 ES6 中的展开运算符（...），可以实现与 apply 同样的功能。在 update 函数的实例中，由于大家关心 this 的值，所以还是要使用 call 方法，不过在 Math.min 和 Math.max 中，大家并不关心 this 的值，所以可以直接用展开操作符来调用函数：

```
const newBruce = [1940, "martial artist"];
update.call(bruce, ...newBruce);          // 跟apply(bruce, newBruce)等价
Math.min(...arr);                         // -5
Math.max(...arr);                         // 15
```

最后一个允许指定 this 值的函数是 bind。bind 方法可以给一个函数永久地邦定 this 值。想象一下在传递 update 方法时，需要保证该方法在任何时候被调用时 this 的值始终是 bruce（即使用了 call，apply，或者其他绑定方式）。bind

可以做到这一点：

```
const updateBruce = update.bind(bruce);

updateBruce(1904, "actor");
    // 现在的bruce是{ name: "Bruce", birthYear: 1904, occupation:"actor"}
updateBruce.call(madeline, 1274, "king");
    // 现在bruce是 { name: "Bruce", birthYear: 1274, occupation:"king"};
    // madeline跟之前一样
```

给函数绑定一个永久的 this 值可能会成为一个潜在的并且很难定位的 bug：因为使用 bind 之后，函数实际上已经不能再有效的使用 call，apply 或者 bind（再次使用）了。想象一下传递一个在其他地方调用了 call 或者 apply 的函数，然后期望它绑定了预期的值的情况。并不是建议不要使用 bind，毕竟它很有用，但是要留意如何使用被绑定的函数。

调用 bind 方法时也可以传参数，这跟创建一个有固定参数的新函数效果一样。比如，希望 update 函数总是把 bruce 的出生年份设为 1949，但却允许修改他的职业，可以这样做：

```
const updateBruce1949 = update.bind(bruce, 1949);
updateBruce1949("singer, songwriter");
    // bruce现在是 { name: "Bruce", birthYear: 1949,
    //    occupation: "singer, songwriter" }
```

6.9 小结

函数是 JavaScript 语言中一个极其重要的部分。它们能做的远比模块化代码要多得多：可以用来构建极其有用的算法单元。本章主要对函数的用法做了简明扼要但却重要的介绍。有了本章的基础知识，在后续的章节中将会看到函数的作用被发挥得淋漓尽致。

第 7 章

作用域

作用域决定了变量、常量和参数被定义的时间和位置。前面已经接触过作用域：知道函数参数的作用域仅限于函数体中，来看一个例子：

```
function f(x) {
    return x + 3;
}
f(5);       // 8
x;          // ReferenceError: x 未定义
```

单从代码上看，x 的的确确存在过（否则它就没法计算出 x+3 的值），但很快就会看到，在函数体外引用 x 的时候会报错。因此，x 是函数 f 的一个内部变量，它的作用域就是函数 f。

当某个变量的作用域是一个给定的函数时，必须要记住：形参只有在函数被调用的时候才存在（变成实参）。一个函数可能会被调用多次：每次函数调用开始时，参数才是真实存在的，在函数返回后参数就失去作用域了。

另外，变量和常量也只有在创建后才存在。也就是说，在使用 let 或 const 声明之前，它们是没有作用域的（var 是一个特例，本章后面会讲到）。

在一些编程语言中，声明和定义之间存在明确的区分。比较典型的是，声明一个变量意味着只是给它指定一个标识符来宣称它的存在。另一方面，定义通常意味着声明后还会赋值。在 JavaScript 中，这两个术语是可以互换的，因为所有变量都会在声明的时候给定一个值（如果没有显式赋值，会有一个隐含值 undefined）。

7.1 作用域和存在

很显然，当一个变量不存在时，它是没有作用域的。也就是说，尚未声明的变量，或者只存在于函数内部的变量，是没有作用域的。

反过来呢？如果变量没有作用域，就意味着它不存在吗？不一定，这就需要区分作用域和存在这两个不同的概念。

作用域（或者可见性）指的是当前可见并且可以被正在执行的代码块（称作执行上下文）访问的标识符。存在也指的是标识符，只不过它们保存了已经分配（预留）过内存的数据。后续很快会看到一些变量存在但没有作用域的例子。

当某些变量不复存在时，JavaScript 不一定会立即回收内存：它只是标记这些条目不再需要保留，垃圾回收进程会定期去回收。在 JavaScript 中，不需要关心垃圾回收，因为这个过程是自动化的，除非应用有较高的性能要求，才需要去考虑垃圾回收。

7.2 静态作用域与动态作用域

通常，在阅读某段程序源代码的时候，是在看它的语法结构。而一旦程序真正运行起来后，执行逻辑却窜来窜去的。来看一个包含了两个函数的例子：

```
function f1() {
    console.log('one');
}
function f2() {
    console.log('two');
}

f2();
f1();
f2();
```

从语法结构上看，这段程序其实就是一长串语句，通常会自上而下阅读它。然而，当运行这段程序时，执行逻辑就窜来窜去：首先跳到函数 `f2` 中，然后跳到 `f1`（即便它先于 `f2` 定义），最后又回到 `f2`。

在 JavaScript 中，作用域是静态的，这意味着只通过阅读源代码就能确定变量的作用域。但这不代表作用域总能在源代码中很直观地呈现出来。在本章中，大家将会看到一些例子，这些例子需要细心检查后才能确定其作用域。

静态作用域指的是，在某个作用域内定义了某个函数（而不是调用函数），该作用域包含的所有变量也在该函数的作用域内。看看下面这个例子：

```
const x = 3;
function f() {
    console.log(x);          // 会正常运行
    console.log(y);          // 会导致程序崩溃
}

const y = 3;
f();
```

不难看出，当方法 f 被定义的时候，变量 x 就已经存在，但 y 还不存在。紧接着才声明了 y，然后调用函数 f。大家会发现当 f 被调用时，x 存在于 f 的作用域中，而 y 没有。这是个静态作用域的例子：函数 f 可以访问在函数定义时就已经存在的标识符，而不能访问在函数调用时才存在的标识符 。

在 JavaScript 中，静态作用域适用于全局作用域、块作用域，以及函数作用域。

7.3 全局作用域

作用域具有层次结构，它有一个顶端：任何程序只要开始运行，就隐式地运行在某个作用域中，这就是全局作用域。当一个 JavaScript 程序开始运行时，在任何函数被调用之前，它都运行在全局作用域内。也就是说，在全局作用域中声明的一切，在当前程序的所有作用域内都是可用的。

在全局作用域中声明的变量叫全局变量，然而，全局变量的名声并不好。几乎所有关于编程的书都会这样警告读者：全局变量简直是洪水猛兽！为什么会这样呢？

全局变量本身并没有什么问题——事实上，它们还是必须的。问题出在全局作用域的滥用上。在前文提到，全局作用域中声明的一切在所有作用域中都可用。所以一定要谨慎地使用全局变量。

聪明的读者可能会想“那好，我在全局作用域中只创建一个函数，这样可以把全局变量减少到只剩一个！”聪明，不过这样只是将问题向下降了一级。因为在这个函数的作用域内声明的变量，对这个函数内部的任何调用来说，也是可用的……这其实不比全局变量好到哪去。

关于全局变量的底线是：大家可能会在全局作用域中声明某些变量，而这不一定就是糟糕的，而是应该努力避免依赖全局作用域。看一个简单的例子：跟踪一个用户

信息。程序会跟踪用户的名字和年龄，有一些函数可以更改这些信息。一种方式是使用全局变量：

```
let name = "Irena";          // 全局变量
let age = 25;                //全局变量

function greet() {
    console.log(`Hello, ${name}!`);
}
function getBirthYear() {
    return new Date().getFullYear() - age;
}
```

这种方式的问题在于，函数高度依赖它们被调用时的上下文（或者作用域）。整个程序中，任何地方的任意函数，都可以改变 name 的值（有意或者无意）。而“name”和“age”是这样的变量名太过普通，很可能被用在其他地方，用于其他的用途。因为 greet 和 getBirthYear 依赖全局变量，它们能正常工作的前提是，程序中其他地方正确地使用了 name 和 age。

另一种更好的方式是将用户信息放到一个单独的对象中：

```
let user = {
    name = "Irena",
    age = 25,
};

function greet() {
    console.log('Hello, ${user.name}!');
}
function getBirthYear() {
    return new Date().getFullYear() - user.age;
}
```

在这个简单的例子中，仅仅是将全局作用域中标识符的数量减少了一个（以 user 代替了 name 和 age），但想象一下如果有 10 条用户信息，会怎么样呢？如果有 100 个呢？

仍然有更好的实现方式，因为，即便函数 greet 和 getBirthYear 依赖了全局变量 user，而 user 还是可以被任意修改。可以对函数做些优化，从而让它不再依赖全局作用域：

```
function greet(user) {
    console.log('Hello, ${user.name}!');
}
function getBirthYear(user) {
```

```
    return new Date().getFullYear() - user.age;
}
```

此时函数可以在任何作用域被调用了，需要显式传入一个 user（当学习了模块和面向对象编程后，会有更好的方式来处理它）。

如果所有程序都这么简单，那么用不用全局变量似乎都没有太大的影响。而当程序长达 1 000 行时，或者 10 万行时，就不可能凭借大脑去记住所有的作用域了（甚至全部显示在屏幕上都不可能）。此时，避免依赖全局作用域就显得更加至关重要了。

7.4 块作用域

let 和 const 关键字声明的变量名处在块作用域中。回忆第 5 章的内容：块是由一对花括号括起来的一系列语句。那么，块作用域指的就是那些仅仅在代码块内有效的变量。

```
console.log('before block');
{
    console.log('inside block');
    const x = 3;
    console.log(x);                          // 打印 3
}
console.log('outside block; x=${x}');        // ReferenceError: x 未定义
```

这里有一个独立的块：块通常是控制流语句的一部分，例如 if 或 for，但定义独立的块也是合法的。在块内定义了变量 x，一旦出了此块，x 就不在作用域内，就会被认为是未定义。

大家可能记得，在第 4 章中讲到一些独立块的使用，它们可以用来控制作用域（跟在本章中将看到的一样），但这通常是不必要的。不过借助它们来解释作用域的工作原理却非常方便，这就是本章中使用了它们的原因。

7.5 变量屏蔽

在 JavaScript 开发中，有一个常常引发混淆的场景是，不同作用域中存在相同名字的变量或常量。而当作用域一个挨着一个的时候，这种场景会显得相对直观一些：

```
{
    // block 1
    const x = 'blue';
```

```
    console.log(x);              // 打印 "blue"
}
console.log(typeof x);           // 打印"undefined"; x 不在作用域内
{
    // block 2
    const x = 3;
    console.log(x);              // 打印"3"
}
console.log(typeof x);           // 打印 "undefined"; x 不在作用域内
```

这个例子非常容易理解：有两个相互独立的变量，在不同的作用域中都以 x 命名。那么，如果作用域嵌套在一起有会发生什么呢？

```
{
    // outer block
    let x = 'blue';
    console.log(x);              // 打印 "blue"
    {
        // inner block
        let x = 3;
        console.log(x);          // 打印"3"
    }
    console.log(x);              // 打印 "blue"
}
console.log(typeof x);           // 打印 "undefined"; x 不在作用域内
```

这个例子演示了变量屏蔽。在内部块中，x 是一个独立于外部块的（同名）变量，这样实际上屏蔽（或隐藏）了外部块定义的 x。

这里有一个关键的地方需要理解，当执行到内部块时，一个新的变量 x 被定义，两个变量都在作用域内：只是没有办法访问外部块的变量（因为具有相同的名字）。相对而言，在前一个例子中，一个变量 x 进入作用域，然后退出，接着第二个变量 x 做了相同的事情。

为了更好地理解这点，看下面例子：

```
{
    // outer block
    let x = { color: "blue" };
    let y = x;                                  // y 和 x 引用同一个对象
    let z = 3;
    {
        // inner block
        let x = 5;                  // 外部 x 被屏蔽
        console.log(x);             // 打印 5
```

```
        console.log(y.color);      // 打印 "blue"; y（以及 x 在外部作用域）
                                      指向的对象仍然在
                                   // 作用域内
        y.color = "red";
        console.log(z);            // 打印 3; 因为 z 没有被屏蔽
    }
    console.log(x.color);          // 打印"red"; 因为 object 在内部作用域被修改
    console.log(y.color);          // 打印"red"; 因为 x 和 y 指向同一个对象
    console.log(z);                // logs 3
}
```

变量屏蔽有时候也叫变量阴影（即一个相同名字的变量会将外部作用域的变量屏蔽起来）。作者不喜欢这个术语，因为阴影通常不会将事物完全隐藏，只是使之变得灰暗。而当一个变量被屏蔽时，这个被屏蔽的变量就无法再通过名字去访问了。

到目前为止，大家应该清楚了解了作用域的层次结构了：可以在原有的作用域中进入一个新的作用域。此时会建立一个作用域链来决定哪些变量在作用域中：当前作用域链中的所有变量都在作用域内，并且（只要它们没有被屏蔽）都是可以被访问的。

7.6 函数、闭包和静态作用域

截止目前，只针对块进行了学习，块有助于理解静态作用域，特别是当对块进行了缩进以后。而对于函数，因为可以在一个地方定义，在另一个地方调用，这就意味着可能要费些周折才能理解它们的作用域。

在“传统”程序中，可能会把所有函数都定义在全局作用域内，而在函数中避免去访问全局变量（作者推荐的方式），甚至不用思考函数可以访问什么作用域。

然而，在现代 JavaScript 开发中，通常会把函数定义在需要使用的地方。将它们赋给变量和对象属性、添加到数组中、当做参数传给函数、作为函数的返回值，甚至有时候连名字也省掉了。

有一个十分常见的场景：故意将某个函数定义在一个指定的作用域中，并明确地指出它对该作用域所具备访问权限，通常称这种形式为闭包（可以认为封闭了函数的作用域）。看一个闭包的例子：

```
let globalFunc;                    // 未定义的全局函数
{
    let blockVar = 'a';            // 块作用域变量
```

```
    globalFunc = function() {
        console.log(blockVar);
    }
}
globalFunc();                                    // 打印"a"
```

globalFunc 在块内被赋值：该块（以及它的父作用域，即全局作用域）构成了一个闭包。不论在哪里调用 globalFunc，它都有权限访问闭包内的变量。

这里面隐含了一层重要的含义：当调用 globalFunc 时，尽管程序已经退出了变量 blockVar 的作用域，它仍然有权限访问它。而通常情况下，某个作用域退出后，该作用域中声明的变量会安全地消亡。在这里例子中，JavaScript 注意到函数被定义在指定作用域内（并且这个函数可以在该作用域外被引用），所以该函数会持有该作用域的访问权限。

在闭包内定义的函数不但可以影响闭包的生命周期，它还允许访问一些正常情况下无法访问到的信息。来看个例子：

```
let f;                                           // 未定义的函数
{
    let o = { note: 'Safe' };
    f = function() {
        return o;
    }
}
let oRef = f();
oRef.note = "Not so safe after all!";
```

通常情况下，作用域之外的信息会受到严格的访问限制。而函数比较特殊，因为它提供了一个绿色通道，让原本封闭的作用域可以被外界访问。在接下来的章节中，大家会看到这个特性的重要性。

7.7 即时调用函数表达式

第 6 章介绍了函数表达式。函数表达式允许创建即时调用函数表达式（IIIFE）。IIFE 声明了一个函数，并立即执行该函数。现在已经掌握了作用域和闭包的概念，也具备了必要的背景知识去理解为什么要这样做。接下来学习 IIFE，它的形式如下：

```
(function() {
   // 这里为 IIFE 代码
})();
```

这段代码中，使用函数表达式创建了一个匿名函数，然后立即调用该函数。IIFE

的好处是，任何内部信息都有自己的作用域，并且因为它本身就是函数，它还可以向作用域外传递信息。

```
const message = (function() {
   const secret = "I'm a secret!";
   return `The secret is ${secret.length} characters long.`;
})();
console.log(message);
```

在 IIFE 的作用域内，变量 secret 是安全的，并且外部不能访问它。可以从 IIFE 中返回任何想要的结果，比较常见的返回值有数组、对象，以及函数。考虑这样一个函数，它可以打印出自己被调用的次数，而且次数不会被篡改：

```
const f = (function() {
   let count = 0;
   return function() {
      return `I have been called ${++count} time(s).`;
   }
})();
f();    // "I have been called 1 time(s)."
f();    // "I have been called 2 time(s)."
//...
```

因为变量 count 被安全地隐藏在 IIFE 中，外界没法篡改它：函数 f 始终能准确地计算出自己被调用的次数。

在 ES6 中，虽然块作用域变量从某种程度上减少了对 IIFE 的需求，但 IIFE 仍然很常用。比如，当想创建一个闭包并给外界返回信息时，它们就很给力了。

7.8 函数作用域和提升

在 ES6 引入 let 关键字之前，变量都是用 var 来声明的，并且存在函数作用域的说法（用关键字 var 声明的全局变量，虽然不在显式定义的函数内，却具有相同的行为）。

使用 let 声明变量，变量只在声明之后才存在。而使用 var 声明，变量在当前作用域中任意地方都可用，甚至可以在声明前使用。在学习示例代码之前，要记住一点，未声明的变量和值为 undefined 的变量不是同一个概念。使用未声明的变量会报错，但可以安全使用已经存在且值为 undefined 的变量。

```
let var1;
let var2 = undefined;
var1;                          // undefined
var2;                          // undefined
```

```
undefinedVar;                    // ReferenceError: undefinedVar 未定义
```

使用 let 时，如果引用的变量未声明，将会得到一个错误。

```
x;                               // ReferenceError: x 未定义
let x = 3;                       // 永远不会执行 - 因为错误终止了程序
```

而用 var 声明的变量，可以在声明前被引用：

```
x;                               // undefined
var x = 3;
x;                               // 3
```

怎么会这样呢？表面上看，这有点难以理解，因为不可能去访问一个还未声明的变量。实际上，var 声明的变量采用了提升机制。JavaScript 会扫描整个作用域（函数或者全局作用域），任何使用 var 声明的变量都会被提升至作用域的顶部。重点要理解的是被提升的声明，而不是赋值。所以 JavaScript 会将之前例子中的代码翻译成下面的形式：

```
var x;                           // 声明（而非赋值）
x;                               // undefined
x = 3;
x;                               // 3
```

下面是一些更复杂的例子，左边是自己写的代码，右边是 JavaScript 翻译后的代码：

```
// 你的代码

if(x !== 3) {
    console.log(y);
    var y = 5;
    if(y === 5) {
        var x = 3;
    }
    console.log(y);
}
if(x === 3) {
    console.log(y);
}
```

```
// JavaScript 解释代码的方式
var x;
var y;
if(x !== 3) {
    console.log(y);
    y = 5;
    if(y === 5) {
        x = 3;
    }
    console.log(y);
}
if(x === 3) {
    console.log(y);
}
```

并不是说这样的 JavaScript 代码写得有多好，相反，使用未声明的变量会带来不必要的困惑，并且容易引发错误（没有实用性理由要这么做）。这个例子只是为了说明变量提升的工作原理。

另外，使用 var 声明的变量，JavaScript 不会关心它是否有重复声明。

```
// 你的代码

var x = 3;
if(x === 3) {
    var x = 2;
    console.log(x);
}
console.log(x);
```

```
// JavaScript 解释的方式
var x;
x = 3;
if(x === 3) {
    x = 2;
    console.log(x);
}
console.log(x);
```

这个例子证明了在同一个函数或者全局作用域内，var 不能用来创建新的变量，使用 let 时出现的变量屏蔽现象在这里不会出现。在这个例子中，仅仅只有一个变量 x，即便在块里面再次使用 var 去定义它，变量依然只有一个。

同样，不建议这么做，因为它只会带来困惑。不经心的读者（尤其是熟悉其他编程语言的）可能瞥一眼这个例子就能做出合理的假设：作者有意在 if 语句构成的作用域内创建一个新变量 x，但实际上并不能达到预期效果。

如果在思考“为什么 var 允许开发人员做一些让人困惑且无用的事情”，那么你应该已经清楚 let 为何而生了。的确，可以以一种合理且清晰的方式使用 var，但苦恼的是，一不留神就会写出令人困惑且模糊的代码。因为有很多遗留代码使用了 var，ES6 不能简单地“修复”var，因此，它引入了新的关键字 let。

任何使用 var 的地方，都可以使用 let 对其进行优化。换言之，var 与 let 比起来毫无优势。许多 JavaScript 社区的开发者（包括我自己）相信 let 最终会完全取代 var（关键字 var 甚至有可能最终会被废弃掉）。

既然如此，那么为什么还要理解 var 和提升呢？有两个原因。首先，ES6 不会立刻普及，这意味着代码还需要转译成 ES5，而且大部分遗留代码是用 ES5 写的。所以在一段时间内，理解 var 的工作原理还是很重要的。其次，函数声明也会被提升，这是下一个主题。

7.9 函数提升

类似于 var 声明的变量，函数声明也会被提升至它们作用域的顶部，这允许在函数声明之前调用。

```
f();                          // 打印"f"
function f() {
    console.log('f');
}
```

注意，赋给变量的函数表达式不会被提升，它们的作用域规则跟变量是一样的，

例如：

```
f();                          // TypeError: f 不是一个函数
let f = function() {
    console.log('f');
}
```

7.10 临时死区

在 JavaScript 中，使用 let 声明的变量只有在被显式声明之后才存在，这是个很直观的概念，临时死区（TDZ）就是针对这个概念的一种戏剧性表达。也就是说，在给定的作用域内，变量的 TDZ 是指该变量被声明之前的代码。

大多数情况下，这都不会引起任何混乱或问题，但有一点，TDZ 会给那些在 ES6 之前就熟悉 JavaScript 的人造成困扰。

typeof 是一个很常用的运算符，它可以确定某个变量是否已被声明，并且它被认为是测试存在性的一种“安全”的方式。也就是说，在 let 诞生之前，任何位于 TDZ 中的标识符 x，使用 typeof 来测试它的存在性始终都是安全、无误的：

```
if(typeof x === "undefined") {
    console.log("x doesn't exist or is undefined");
} else {
    // safe to refer to x....
}
```

这段代码如果改用 let 来声明变量，typeof 就不再是一个安全的操作。下面例子会发生错误：

```
if(typeof x === "undefined") {
    console.log("x doesn't exist or is undefined");
} else {
    // 此时可以安全地引用 x....
}
let x = 5;
```

在 ES6 中，没有必要使用 typeof 来检查变量是否被定义。所以，实际使用的时候，typeof 在 TDZ 中的行为不应该引发任何问题。

7.11 严格模式

ES5 的语法允许存在隐式全局变量，而这正是那些令人懊恼的编程错误的源头。简

而言之，如果忘记使用 var 声明某个变量，JavaScript 会不假思索地认为开发人员在引用一个全局变量。如果该全局变量不存在，它会替开发人员创建一个！大家可以想象一下这会造成什么问题。

出于这个（以及其他的）原因，JavaScript 引进了严格模式，它能阻止隐式全局变量。在开始编写代码之前，单独插入一行字符串"use strict"（单引号和双引号都可以）就可以启用严格模式。如果在全局作用域中这么做，整个脚本会以严格模式执行。而如果在函数中使用它，该函数就会以严格模式执行。

因为在全局作用域中使用严格模式之后，它会应用于所有脚本代码，所以使用时要谨慎一些。很多流行的网站在部署前会整合所有脚本，一旦某个脚本文件中开启了全局严格模式，那么所有的文件都会启用这个模式。当然，如果所有脚本都正常工作就万事大吉了，但往往不会这么顺利。所以通常不建议在全局作用域中使用严格模式。如果不想在每个函数中都开启一次严格模式（谁会愿意这么麻烦呢？），那么可以将所有代码封装在一个立即执行的函数中（将在第 13 章中了解更多）：

```
(function() {
    'use strict';

    // 所有代码从这里开始...代码会按照严格模式执行，不过严格模式不会干扰组合在一
    // 起的其他脚本

})();
```

严格模式通常被认为是个好东西，推荐使用它。如果使用了格式检查工具（应该这么做），它会帮助用户从类似的问题中解脱出来，而这种双重校验只会带来更多的好处。

想要了解更多严格模式的用途，欢迎阅读 MDN 关于严格模式的文章（*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode*）。

7.12 小结

理解作用域在任何编程的语言学习过程中都是一个重要的环节。let 的出现使得 JavaScript 跻身现代编程编程语言行列。虽然 JavaScript 不是首个支持闭包的语言，但它是第一批流行起来的（非学术）语言之一。JavaScript 社区在闭包的使用上成果显著，使得闭包成为了现代 JavaScript 开发中重要的组成部分。

第 8 章

数组及其处理

JavaScript 的数组方法是作者最喜欢的特性之一。因为有太多的编程问题都跟数据集合的操作有关，而 JavaScript 的数组方法能让这些操作都变得简单起来。熟练掌握这些方法可以在精通 JavaScript 的道路上更进一步。

8.1 数组概览

在深入讲解之前，先来回顾一下数组的基础知识。数组（不像对象）是有序的，它的下标从 0 开始。JavaScript 中的数组可以是非齐次的，也就是说，数组中的元素可以是不同的类型（遵循这个原则的话，数组的元素可以是另一个数组，或对象）。字面量数组是由方括号构建的，这个方括号还能根据下标来访问数组元素。每个数组都有一个 length 属性，表示数组中有多少元素。给大于数组元素长度的下标赋值，会增加数组长度，而尝试访问超过数组元素下标的元素时，会得到 undefined。也可以用数组构造器创建数组，即便它不常用。在继续之前，先看看下面的代码，确保这些内容都理解了：

```
// 数组字面量
const arr1 = [1, 2, 3];                                    //整数数组
const arr2 = ["one", 2, "three"];                           // 非齐次数组
const arr3 = [[1, 2, 3], ["one", 2, "three"]];              //包含数组的非齐次数组
const arr4 = [                                              //非齐次数组
   { name: "Fred", type: "object", luckyNumbers = [5, 7, 13] },
   [
      { name: "Susan", type: "object" },
      { name: "Anthony", type: "object" },
   ],
   1,
```

```
    function() { return "arrays can contain functions too"; },
    "three",
];

// 给数组元素赋值
arr1[0];                          // 1
arr1[2];                          // 3
arr3[1];                          // ["one", 2, "three"]
arr4[1][0];                       // { name: "Susan", type: "object" }

// 获取数组长度
arr1.length;                      // 3
arr4.length;                      // 5
arr4[1].length;                   // 2

// 增加数组长度
arr1[4] = 5;
arr1;                             // [1, 2, 3, undefined, 5]
arr1.length;                      // 5

// 访问超过数组元素下标的元素
// 并*不会*改变数组大小
arr2[10];                         // undefined
arr2.length;                      // 3

// 构造数组 (很少用)
const arr5 = new Array();                 // empty array
const arr6 = new Array(1, 2, 3);          // [1, 2, 3]
const arr7 = new Array(2);                // array of length 2 (all
                                             elements undefined)
const arr8 = new Array("2");              // ["2"]
```

8.2 操作数组内容

在开始讲解那些激动人心的方法之前，先看一些常用的数组方法（非常有用）。数组方法中最容易让人混淆的是：修改当前数组和返回新数组的方法之间的区别。关于如何区分这两者，并没有相关的约定，所以这是需要记住的内容（比如，push 会修改当前数组，而 concat 则会返回一个新数组）。

在其他语言中，比如 Ruby，有一个约定来界定这个函数是修改当前值还是返回当前值的拷贝。例如，在 Ruby 中，如果有一个字符串 str，调用 str.downcase，会返回该字符串的全小写版本，但是 str 的值不会改变。不过，如果调用 str.downcase!，则 str 会被修改。在 JavaScript 标准库中，并没有说明哪些方法会返回新的拷贝值，而哪些

JavaScript 标准库中，并没有说明哪些方法会返回新的拷贝值，而哪些会修改原值。在作者看来，需要额外在大脑中记住这些信息就是这门语言的一个缺点。

8.2.1 在起始和末尾添加或删除元素

当说数组起始（或者开头）的时候，其实是在说数组的第一个元素（element 0）。同样，数组的末尾（或者最后）说的是下标最大的元素（如果数组是 arr，那么最大的下标就是 arr.length-1）。push 和 pop 的作用分别是添加或删除数组的最后一个元素（修改当前数组）。相应地，shift 和 unshift 则是删除和添加数组的第一个元素（修改当前元素）。

这些方法的名字来源于计算机科学中的专业术语。push 和 pop 是栈的操作，其中最重要的元素是最新添加的。shift 和 unshift 则将数组当做队列操作，最重要的元素是最早添加的。

push 和 unshift 会返回添加了新元素后的数组长度，pop 和 shift 则会返回被删除的元素。下面列出了这些方法的用法：

```
const arr = ["b", "c", "d"];
arr.push("e");                  // 返回 4; arr 现在是 ["b", "c", "d", "e"]
arr.pop();                      // 返回 "e"; arr 现在是["b", "c", "d"]
arr.unshift("a");               // 返回 4; arr 现在是["a", "b", "c", "d"]
arr.shift();                    // 返回 "a"; arr 现在是["b", "c", "d"]
```

8.2.2 在末尾添加多个元素

concat 方法可以给数组添加多个元素并返回新数组的拷贝。如果给 concat 传一些数组作为参数，它会把这些数组拆分再把拆分后的元素添加到原始数组中。例如：

```
const arr = [1, 2, 3];
arr.concat(4, 5, 6);            // 返回[1, 2, 3, 4, 5, 6]; arr 没有改动
arr.concat([4, 5, 6]);          // 返回 [1, 2, 3, 4, 5, 6]; arr 没有改动
arr.concat([4, 5], 6);          // 返回 [1, 2, 3, 4, 5, 6]; arr 没有改动
arr.concat([4, [5, 6]]);        // 返回 [1, 2, 3, 4, [5, 6]]; arr 没有改动
```

需要注意的是，concat 只会拆分直接传给它的数组；并不会拆分数组内的数组。

8.2.3 获取子数组

如果想要从一个数组中获取子数组，可以用 slice。slice 有两个参数，第一个参数是子数组的起始位置，第二个参数则是它的终止位置（不包括特殊字符）。如

果省略终止参数，则会返回从当前位置到数组末尾的所有内容。在这个方法中，下标可以使用负值来表示倒数的元素，这是一个非常实用的功能。例如：

```
const arr = [1, 2, 3, 4, 5];
arr.slice(3);          // 返回[4, 5]; arr 没有改动
arr.slice(2, 4);       //返回[3, 4]; arr 没有改动
arr.slice(-2);         //返回[4, 5]; arr 没有改动
arr.slice(1, -2);      //返回[2, 3]; arr 没有改动
arr.slice(-2, -1);     // 返回[4]; arr 没有改动
```

8.2.4 从任意位置添加或删除元素

slice 允许对当前字符串做修改，可以从任意一个下标增加和/或删除元素。这个方法的第一个参数是起始修改位置的数组下标；第二个参数是需要删除的元素个数（如果不想删除任何元素就传入 0），其他参数则是需要添加到数组中的元素。例如：

```
const arr = [1, 5, 7];       // 返回[]; arr 现在是 [1, 2, 3, 4, 5, 7]
arr.splice(1, 0, 2, 3, 4);  // 返回 []; arr 现在是 [1, 2, 3, 4, 5, 6, 7]
arr.splice(5, 0, 6);         // 返回 [2, 3]; arr 现在是 [1, 4, 5, 6, 7]
arr.splice(1, 2);
arr.splice(2, 1, 'a', 'b'); // 返回 [5]; arr 现在是 [1, 4, 'a', 'b', 6, 7]
```

8.2.5 数组内的分割和替换

ES6 引入了一个新方法，copyWithin，它会将数组中一串有序的元素复制到数组的另一个位置，复制的同时会覆盖原来数组中的内容。这个方法的第一个参数表明要复制到哪里（目标位置），第二个参数是从哪里开始复制，最后一个（可选的）参数是复制到哪里结束。与 slice 一样，这里也可以用负数来表示开始位置和结束位置，这样就会从数组末尾开始反向计数。例如：

```
const arr = [1, 2, 3, 4];
arr.copyWithin(1, 2);           // arr 现在是[1, 3, 4, 4]
arr.copyWithin(2, 0, 2);        // arr 现在是[1, 3, 1, 3]
arr.copyWithin(0, -3, -1);      // arr 现在是[3, 1, 1, 3]
```

8.2.6 用指定值填充数组

ES6 还有一个很受欢迎的方法，fill。它可以将一个固定值赋给数组中任意位置元素（修改当前数组）。这个方法跟数组构造器（可以指定数组的初始大小）结合起来非常实用。可以任意指定一个起始位置和结束位置，然后填充这部分的数组值（下标还可以为负值）。例如：

```
const arr = new Array(5).fill(1);     // arr 被初始化为 [1, 1, 1, 1, 1]
```

```
arr.fill("a");                    // arr 现在是 ["a", "a", "a", "a", "a"]
arr.fill("b", 1);                 // arr 现在是 ["a", "b", "b", "b", "b"]
arr.fill("c", 2, 4);              // arr 现在是 ["a", "b", "c", "c", "b"]
arr.fill(5.5, -4);                // arr 现在是 ["a", 5.5, 5.5, 5.5, 5.5]
arr.fill(0, -3, -1);              // arr 现在是 ["a", 5.5, 0, 0, 5.5]
```

8.2.7 数组反转和排序

reverse 的用法很简单，它会颠倒当前数组的顺序：

```
const arr = [1, 2, 3, 4, 5];
arr.reverse();                    // arr is now [5, 4, 3, 2, 1]
```

sort 则会对当前数组进行排序：

```
const arr = [5, 3, 2, 4, 1];
arr.sort();                      // arr is now [1, 2, 3, 4, 5]
```

sort 还允许指定一个排序函数，这个功能不仅方便而且好用。比如，直接对数组进行排序是没有实际意义的：

```
const arr = [{ name: "Suzanne" }, { name: "Jim" },
   { name: "Trevor" }, { name: "Amanda" }];
arr.sort();                                    // arr 没有变化
arr.sort((a, b) => a.name > b.name);           // arr 按照元素 name 属性的字母顺
                                               //序排序
arr.sort((a, b) => a.name[1] < b.name[1]); // arr 按照元素 name 属性的第二个字
                                               //母的字母顺序就行排序
```

在这个例子中，sort 会返回一个布尔值。不过，sort 也可以将数字作为返回值。如果返回 0，则表示这两个元素“相等”，函数不改变数组的顺序。这样一来，这个方法的灵活性就很大。比如，需要对除去以字母 k 开头的单词按照字母顺序排序，排序的结果是所有元素按照字母顺序排列，所有以 k 开头的元素都在以 j 开头的元素后面，同时在以 l 开头的元素之前，但是以 k 开头的元素会保持它们的原始顺序。

8.3 数组搜索

如果想在数组中查找元素，有以下几种选择。从最简单的 indexOf 开始介绍，JavaScript 很久以前就开始支持它了。indexOf 将返回跟所要查找的内容完全相同的第一个元素的下标（对应的还有 lastIndexOf()，从数组末尾开始查找，同样返回完全匹配的第一个元素的下标）。如果只想在数组的某一部分查找，可以指定

查找的起始下标。如果 indexOf（或者 lastIndexOf()）返回-1，则表示没有匹配的元素：

```
const o = { name: "Jerry" };
const arr = [1, 5, "a", o, true, 5, [1, 2], "9"];
arr.indexOf(5);                    // returns 1
arr.lastIndexOf(5);                // returns 5
arr.indexOf("a");                  // returns 2
arr.lastIndexOf("a");              // returns 2
arr.indexOf({ name: "Jerry" });    // returns -1
arr.indexOf(o);                    // returns 3
arr.indexOf([1, 2]);               // returns -1
arr.indexOf("9");                  // returns 7
arr.indexOf(9);                    // returns -1

arr.indexOf("a", 5);               // returns -1
arr.indexOf(5, 5);                 // returns 5
arr.lastIndexOf(5, 4);             // returns 1
arr.lastIndexOf(true, 3);          // returns -1
```

另一个查找元素的方法是 findIndex，它与 indexOf 一样，返回一个下标（返回-1 表示没有匹配元素）。相比于 indexOf，findIndex 更灵活，因为它可以指定一个函数来表示元素是否匹配（findIndex 不能指定查找的起始位置，不过它有对应的 findLastIndex 函数）：

```
const arr = [{ id: 5, name: "Judith" }, { id: 7, name: "Francis" }];
arr.findIndex(o => o.id === 5);              // returns 0
arr.findIndex(o => o.name === "Francis");    // returns 1
arr.findIndex(o => o === 3);                 // returns -1
arr.findIndex(o => o.id === 17);             // returns -1
```

find 还是一个用于查找的方法，它和 findIndex 都很适合查找特定元素的下标。不过，如果并不关心元素下标，只想返回元素本身呢？像 findIndex 一样，find 也允许指定一个函数来进行查找，只不过它会返回元素本身而非元素下标（没找到元素时返回 null）。

```
const arr = [{ id: 5, name: "Judith" }, { id: 7, name: "Francis" }];
arr.find(o => o.id === 5);   // returns object { id: 5, name: "Judith" }
arr.find(o => o.id === 2);   // returns null
```

那些传入 find 和 findIndex 中的函数，会把数组的每一个元素作为第一个参数，其他参数还有当前元素的下标和数组本身。它们可以完成这样一些事情，比如：寻找以指定下标开始的元素的平方数。

```
const arr = [1, 17, 16, 5, 4, 16, 10, 3, 49];
arr.find((x, i) => i > 2 && Number.isInteger(Math.sqrt(x)));// returns 4
```

find 和 findIndex 还允许在方法调用过程中指定 this 变量使用的值。它们可以用在需要调用某个对象方法的场景中。以下这些方法都会在 Person 对象中通过 ID 来查找元素，对比一下这些实现：

```
class Person {
   constructor(name) {
      this.name = name;
      this.id = Person.nextId++;
   }
}
Person.nextId = 0;
const jamie = new Person("Jamie"),
   juliet = new Person("Juliet"),
   peter = new Person("Peter"),
   jay = new Person("Jay");
const arr = [jamie, juliet, peter, jay];

// 选项 1：直接比较 ID：
arr.find(p => p.id === juliet.id);            // 返回 juliet 对象

// 选项 2：使用 "this"参数：
arr.find(p => p.id === this.id, juliet);     // 返回 juliet 对象
```

大家可能会发现在 find 和 findIndex 中，this 值的作用是有限的。但实际上这是后面会讲到的技术，届时它将会变得更加实用。

正如并不总关心数组中元素的索引一样，有时候甚至不关心下标或是元素本身，只想单纯的知道这个元素是否存在。显然，可以使用之前提到的函数来检查元素是否存在，看它是返回-1 还是 null，不过，JavaScript 专门提供了两个方法来解决这个问题：some 和 every。

一旦找到符合条件的元素（只需要一个符合条件的元素，所以在找到第一个元素后会停止查找），some 方法就会返回 true，否则返回 false。例如：

```
const arr = [5, 7, 12, 15, 17];
arr.some(x => x%2===0);                          // true; 12 是偶数
arr.some(x => Number.isInteger(Math.sqrt(x)));   // false; 没有平方数
```

而当数组中的每个元素都符合条件的时候，every 方法才会返回 true，否则返回 false。当 every 方法发现一个不符合条件的元素时，就会停止查找并返回 false；否则，会扫描整个数组：

```
const arr = [4, 6, 16, 36];
arr.every(x => x%2===0);                         // true; 没有奇数
```

```
arr.every(x => Number.isInteger(Math.sqrt(x)));  // false; 6不是平方数
```

像本章中所提及的其他方法一样，some 和 every 也可以接收一个方法作为参数，它们还能再接收一个参数，用来指定方法被调用时的 this 值。

8.4 数组的基本操作：map 和 filter

在所有的数组操作中，大家会发现最有用的还是 map 和 filter。这两个方法能做的事情绝对不能不提。

map 可以把元素转换成数组。至于转成什么类型的数组，这就是 map 的优美之处：类型由开发人员决定。大家有没有遇到过对象中含有数字，但是只需要数字的情况？有没有遇到过数组中包含函数，而需要 promise 的情况？map 很容易满足这些需求。*任何时候如果数组格式与所需要的格式不一样，就用* map。map 和 filter 都不会修改原始数组，而是返回数组的拷贝。来看一些例子：

```
const cart = [ { name: "Widget", price: 9.95 }, { name: "Gadget", price:
22.95}];
const names = cart.map(x => x.name);              // ["Widget", "Gadget"]
const prices = cart.map(x => x.price);            // [9.95, 22.95]
const discountPrices = prices.map(x => x*0.8);    // [7.96, 18.36]
const lcNames = names.map(String.toLowerCase);    // ["widget", "gadget"]
```

大家可能会好奇 lcNames 是怎么工作的：它看起来跟其他数组不一样。这里讨论到的所有方法，包括 map，都可以接收一个函数作为参数，而且它们不关心函数是以哪种方式传入的。在 names，prices，和 discountPrices 的例子中，都构造了自定义的方法（用箭头符号）。而在 lcNames 中，用了一个早已存在的方法，String.toLowerCase。这个方法会接收一个字符串，返回该字符串的小写字符串。这使得可以很简单地写下 names.map(x⇒ x.toLowerCase())，不过一定要知道，方法就是方法，无论它是哪种格式，理解这一点很重要。

数组每个元素在调用提供的方法时都会传入三个参数：元素本身、元素的下标，以及数组本身（这个不常用）。来看一下这个例子：items 和其对应的 prices 分别在两个独立的数组中，将它们合并起来：

```
const items = ["Widget", "Gadget"];
const prices = [9.95, 22.95];
const cart = items.map((x, i) => ({ name: x, price: prices[i]}));
// cart: [{ name: "Widget", price: 9.95 }, { name: "Gadget", price: 22.9
5 }]
```

这个例子有点复杂，不过它很好地演示了 map 函数的作用。在这里，不仅使用了

元素本身（x），还有它的索引（i）。之所以需要索引是因为需要根据索引来关联 items 和 prices 中的元素。这里 map 通过从不同的数组中提取信息，从而把一个字符串类型的数组转化成了一个对象数组。（注意，在这里要把对象用小括号包裹起来，因为如果没有小括号，箭头操作符会把花括号中的内容当成程序块）。

filter，顾名思义，它是用来删除数组中不需要的元素。像 map 一样，它返回一个删除了某些元素的数组。那么，哪些元素会被删除呢？同样，这也完全取决于开发人员。如果读者觉得这里会定义一个函数来决定哪些元素要被删除，那么值得恭喜，答对了！来一起看个例子吧：

```
// 创建一副牌
const cards = [];
for(let suit of ['H', 'C', 'D', 'S']) // hearts, clubs, diamonds, spades
    for(let value=1; value<=13; value++)
        cards.push({ suit, value });

// 找到所有含有 2 的卡片：
cards.filter(c => c.value === 2);     // [
                                      //     { suit: 'H', value: 2 },
                                      //     { suit: 'C', value: 2 },
                                      //     { suit: 'D', value: 2 },
                                      //     { suit: 'S', value: 2 }
                                      // ]

// （简单起见，下面的代码中只列出数组长度）

// 找到所有方块：
cards.filter(c => c.suit === 'D');        // length: 13

// 找到所有花色牌
cards.filter(c => c.value > 10);          // length: 12

// 找到所有为红桃的花色牌
cards.filter(c => c.value > 10 && c.suit === 'H');  // length: 3
```

希望大家已经了解了如何将 map 跟 filter 结合起来使用从而产生良好的效果。例如，想为上述的卡片创建一个简短的介绍。这里将使用 Unicode 来表示牌的花色，还用到了“A”“J”“Q”“Q”表示王牌和人头牌。由于构造出的这个函数太长了，另外会创建一个单独的函数，而非一个匿名函数。

```
    function cardToString(c) {
       const suits={'H': '\u2665','C': '\u2663', 'D': '\u2666', 'S': '\u266
0'};
       const values = { 1: 'A', 11: 'J', 12: 'Q', 13: 'K' };
```

```
    // 每次调用 cardToString 的时候去构建值，不是一个高效的方式
    // 如何找出一个高效的方法就作为读者练习吧
    for(let i=2; i<=10; i++) values[i] = i;
    return values[c.value] + suits[c.suit];
}

// 找到所有包含 2 的牌：
cards.filter(c => c.value === 2)
   .map(cardToString); // [ "2♥", "2♣", "2♦", "2♠" ]

// 找到所有红桃的花色牌
cards.filter(c => c.value > 10 && c.suit === 'H')
   .map(cardToString); // [ "J♥", "Q♥", "K♥" ]
```

8.5 数组魔法：reduce

在所有数组方法中，作者最喜欢的是 reduce。map 方法可以转化数组中每一个元素，而 reduce 则可以转化整个数组。之所以叫它 reduce，是因为它经常被用于将一个数组归纳成一个单独的值。比如，对数组所有元素求和或者计算它们的平均值就是几种常见的场景。然而，提供单值归纳功能可以是对象或者是其他数组——事实上，reduce 复制了一部分 map 和 filter（或者说，任何一个我们讲过的数组方法）的功能。

Reduce 像 map 和 filter 一样，允许提供一个可以控制输出的函数。在以往处理的回调函数中，第一个传入回调函数的参数一般是当前数组元素。而在 reduce 中，第一个参数是一个累加值，表示数组将被归纳成的值。剩下的参数就跟大家想的一样了：当前数组元素、当前元素的下标和数组的。

除了有回调函数外，reduce 还可以接收一个累加值的初始值（可选的）。下面来看一个简单的例子：给数组中的数字求和：

```
const arr = [5, 7, 2, 4];
const sum = arr.reduce((a, x) => a += x, 0);
```

传入 reduce 中的函数有两个参数：累加值（a）和当前数组的元素（x）。在本例中，让累加值从 0 开始。由于这是第一次使用 reduce，所以首先来了解一下 JavaScript 的操作步骤，有助于理解它的工作原理。

（1）函数（匿名）被数组的第一个元素（5）调用。a 表示初始值 0，x 表示第一个元素的值：5。该函数返回 a 和 x 的和（还是 5），这将作为下一步中 a 的值。

(2) 函数被数组的第二个元素（7）调用。a 的初始值为 5（从上一步传过来的），x 的值是 7。函数会返回 a 和 x 的和（12），这将作为下一步中 a 的值。

(3) 以此类推，当函数被数组的第三个元素（2）调用时。a 的初始值是 12，x 的值为 2。函数会返回 a 和 x 的和（14）。

(4) 函数被数组的第四个元素，也是最后一个元素（4）调用。a 的值为 14，x 为 4。该函数会返回 a 和 x 的和（18），这也是 reduce 函数将会返回的值（也是接下来赋给 sum 的值）。

聪明的读者可能已经发现，这是一个很简单的例子，甚至不需要给 a 赋值。重要的是，函数返回什么（还记得吗？箭头标识符并不需要明确声明 return 语句），所以可以直接返回 a + x。不过，在一些更复杂的例子中，可能希望对累加结果做更多操作，所以，好的习惯是尽量在函数内部修改累加结果。

在继续学习 reduce 那些更有趣的用法之前，先来看一种特殊情况：累加结果的初始值是 undefined。在这种情况下，reduce 会把数组的第一个元素当做初始值，然后从数组的第二个元素开始调用函数。下面重新看一下这个例子，这次省略初始值：

```
const arr = [5, 7, 2, 4];
const sum = arr.reduce((a, x) => a += x);
```

(1) 函数（匿名）被数组的第二个元素（7）调用。a 的初始值是 5（数组第一个元素的值），x 的值是 7。调用后函数返回 a 和 x 的和（12），这会成为下一步的值。

(2) 函数被数组的第三个元素（2）调用。a 的初始者是 12，x 是 2。函数返回 a 和 x 的和（14）。

(3) 函数被数组的第四个元素（4）调用。a 的值为 14，x 为 4。函数依旧返回 a 和 x 的和（18），它就是 reduce 函数的返回值（同时赋值给 sum）。

可以看到，这里虽少了一步函数调用，但结果是一样的。在本例（或者任何第一个元素可以作为累加结果初始值的例子）中，可以省略初始值来简化代码。

在 reduce 中使用原子值（数字或者字符串）是一个很常见的用法，不过如果用对象作为累加结果，其功能将会非常强大（这个用法经常被人们忽视）。比如：如果你想将一个 string 类型的数组按照字母顺序（以 A 开头的单词，以 B 开头的单词，等等）进行分组，然后每个组作为一个数组元素，这种情况下就可以使用对象：

```
const words = ["Beachball", "Rodeo", "Angel",
```

```
    "Aardvark", "Xylophone", "November", "Chocolate",
    "Papaya", "Uniform", "Joker", "Clover", "Bali"];
const alphabetical = words.reduce((a, x) => {
    if(!a[x[0]]) a[x[0]] = [];
    a[x[0]].push(x);
    return a; }, {});
```

这个例子有点复杂，但本质上它与之前的例子是一样的。对于数组中的每个元素，函数都会检查累加结果，来判断结果中是否包含元素的第一个字母。如果不包含，就创建一个空数组（比如，当遇到"Beachball"的时候，它没有 a.B 的属性，这时元素的值就会被放入一个新创建的空数组中）。然后它会把元素放在对应的数组（就是刚刚创建的数组）中。最终，累加结果（a）会被返回（还记得吗，返回的结果将作为数组中下个元素调用函数时的累加结果。）

另一个相关的例子是计算统计数据。比如，计算某个数据集的平均值和变化：

```
const data = [3.3, 5, 7.2, 12, 4, 6, 10.3];
// 这个计算方差的方法来源于 Donald Knuth 于 1998 年出版的
// 计算机程序设计艺术，第三版中的第二卷：半数值算法
const stats = data.reduce((a, x) => {
        a.N++;
        let delta = x - a.mean;
        a.mean += delta/a.N;
        a.M2 += delta*(x - a.mean);
        return a;
    }, { N: 0, mean: 0, M2: 0 });
if(stats.N > 2) {
    stats.variance = stats.M2 / (stats.N - 1);
    stats.stdev = Math.sqrt(stats.variance);
}
```

这里再一次使用了对象作为累加结果，因为需要多个变量（mean 和 M2，也就是说，如果需要的话，可以用下标参数减一的方式来替换 N）。

来看最后一个关于 reduce 的例子，它体现了 reduce 的灵活性，这个例子用了一个之前没用过的累加结果类型—字符串：

```
const words = ["Beachball", "Rodeo", "Angel",
    "Aardvark", "Xylophone", "November", "Chocolate",
    "Papaya", "Uniform", "Joker", "Clover", "Bali"];
const longWords=words.reduce((a, w)=>w.length>6 ? a+" "+w:a,"").trim();
// longWords: "Beachball Aardvark Xylophone November Chocolate Uniform"
```

这里用了字符串作为累加结果，它存放所有长度大于 6 个字母的字符串。作为一个读者练习留给你：试着用 filter 和 join（一个字符串方法）重新实现这段代码。

(提示：可以从思考为什么需要在 reduce 后调用 trim 开始)。

至此，希望大家已经了解了 reduce 的强大功能。在所有的数组方法中，它是最常用且作用最大的。

8.6 数组方法，已删除或者未定义的元素

在数组方法中，人们经常犯错的地方是这些函数处理未定义和已删除元素的方式。当数组中的元素未被赋值或已被删除时，map、filter 和 reduce 就不会调用所传入的函数。比如：在 ES6 之前，如果想耍点小聪明，用下面的方式来初始化一个数组，那将会令人很失望：

```
const arr = Array(10).map(function(x) { return 5 });
```

执行上述语句后，arr 还是一个有 10 个元素的数组，但所有元素都是 undefined。类似地，如果从数组中删除一个元素，然后再调用 map，就会得到一个有“缺口”的数组：

```
const arr = [1, 2, 3, 4, 5];
delete arr[2];
arr.map(x => 0);      // [0, 0, <1 empty slot>, 0, 0]
```

实际上，这是一个很罕见的情况，因为编程中遇到的大多数数组，其元素都是被明确定义好的（除非就是想让数组中有一个缺口，这种情况很罕见。一般情况下不会在数组中用 delete），不过，意识到这种特殊情况的存在还是会对大家有所帮助。

8.7 字符串连接

有时候，你可能希望把数组中元素的值通过一些分隔符连接起来。Array.prototype.join 方法接收一个分隔符作为参数（在不指定的时候会用逗号作为默认值），返回一个连接了所有元素的字符串（包含未定义和删除的元素，这些元素会变成空数组、空元素、undefined、或空字符串）：

```
const arr = [1, null, "hello", "world", true, undefined];
delete arr[3];
arr.join();           // "1,,hello,,true,"
arr.join('');         // "1hellotrue"
arr.join(' -- ');     // "1 -- -- hello -- -- true --"
```

如果该方法用得够巧妙，结合字符串连接 Array.prototype.join 可以用来创建类似于 HTML 中<ul>的列表：

```
const attributes = ["Nimble", "Perceptive", "Generous"];
const html = '<ul><li>' + attributes.join('</li><li>') + '</li></ul>';
// html: "<ul><li>Nimble</li><li>Perceptive</li><li>Generous</li></ul>";
```

这里一定要注意，不要在空数组上调用这个方法：因为最终会得到一个空的<li>元素！

8.8 小结

JavaScript 的 Array 类有很多功能强大而且灵活性高的内建方法，但是知道在什么情况下用哪个方法却是一件让人望而却步的事情。表 8-1 到 8-4 总结了所有的数组方法。

对于 Array.prototype 方法，它会接收一个函数（比如 find、findIndex、some、every、map、filter 和 reduce）作为参数，这些函数将会接收表 8-1 中所列举的参数，继而被数组中的每个元素调用。

表 8-1 数组函数参数（按顺序）

方　法	描　述
只是 Reduce	累加结果（初始值，或者上一步调用的结果）
全部	元素（当前元素）
全部	当前元素的下标
全部	当前数组本身（一般不常用）

所有接收函数的 Array.prototype 方法都能接收一个（可选的）this 值，该值在函数被调用的时候会派上用场，这样就可以将该函数当做一个对象方法进行调用。

表 8-2 数组内容操作

当需要…	使用…	修改当前数组还是返回拷贝数组
创建一个栈（先进后出[LIFO]）	push（返回新数组的长度），pop	修改当前数组
创建一个队列（先进先出[FIFO]）	unshift（返回新数组的长度），shift	修改当前数组
在数组末尾添加多个元素	concat	返回数组的拷贝
获取子数组	slice	返回数组的拷贝
在任意位置添加或删除元素	splice	修改当前数组
剪切并替换数组元素	copyWithin	修改当前数组
填充数组	fill	修改当前数组

续表

当需要…	使用…	修改当前数组还是返回拷贝数组
反转数组	reverse	修改当前数组
数组排序	sort（传入函数来进行自定义排序）	修改当前数组

表 8-3　　数组搜索

当需要查找…	使用…
元素的下标	indexOf（简单的值），findIndex（复杂的值）
最后一个元素的下标	lastIndexOf（简单值）
元素本身	find
数组中符合条件的元素	some
数组中所有元素都符合给定条件	every

表 8-4　　数组转化

当需要…	使用…	修改当前数组还是返回拷贝数组
转化数组中的所有元素	map	返回数组的拷贝
根据给定条件排除数组元素	filter	返回数组的拷贝
把整个数组转化成另一种数组类型	reduce	返回数组的拷贝
把元素转化成字符串并合并	join	返回数组的拷贝

第 9 章

对象以及面向对象编程

第 3 章大家已经了解了对象的基础知识，现在是时候深入学习 JavaScript 对象了。

与数组一样，在 JavaScript 中对象也是一种容器（也称作聚合或复杂数据类型）。对象和数组的两个主要区别是：

- 数组包含值，使用数字做索引；而对象包含属性，使用字符串和符号作为索引。
- 数组是有序的（arr[0]始终在 arr[1]之前）；而对象是无序的（你不能保证 obj.a 在 obj.b 之前）。

这些区别相当晦涩难懂（但很重要），所以接下来先了解一下属性（没有双关语）——对象真正的特别之处。属性由键（字符串或符号）和值组成。对象的特别之处就在于，可以使用键来访问其对应的属性。

9.1 属性枚举

通常，如果想列举容器的内容（称作枚举），很可能会想到数组而非对象。但是别忘了，对象也是容器，它也支持属性枚举；而开发人员只需要留意一下下面这些特性。

第一点是，对象属性枚举是无序的。可能做了一些测试，发现取出的顺序和放入的顺序是一致的，而且大多数情况下也确实如此。然而，JavaScript 明确声明了不保证顺序，出于性能原因，其实现方式也可能有些变化。所以不要被那些轶事般的测试得出的结果所迷惑，永远不要假设对象的属性是有序的。

在清楚了这些警告之后，再来看看枚举对象属性的主要方法。

9.1.1 for...in

for...in 是枚举对象属性的传统方式。来看一个包含若干字符串属性和一个符号属性的对象的例子：

```
const SYM = Symbol();

const o = { a: 1, b: 2, c: 3, [SYM]: 4 };

for(let prop in o) {
  if(!o.hasOwnProperty(prop)) continue;
  console.log(`${prop}: ${o[prop]}`);
}
```

这似乎很直观，不过大家可能会好奇 hasOwnProperty 起什么作用。它解决了 for...in 循环内部可能存在的一个风险，本章后面讲解属性继承的时候会解答这个问题。这在个例子中，它不会对结果造成影响，可以暂时忽略它。然而，如果需要枚举其他类型的对象中的属性，尤其是从别处传来的对象，可能会得到一些意料之外的属性。所以鼓励大家养成使用 hasOwnProperty 的习惯。当然很快就会发现它的重要性，也将清楚在哪些情况下不需要使用它。

注意，for...in 循环不会枚举出键为符号的属性。

虽然 for...in 也可以用来迭代数组，但这通常被认为是一个不好的实践。建议大家使用一般的 for 循环或 forEach 迭代数组。

9.1.2 Object.keys

Object.keys 提供了一种方式用来获取对象中所有可枚举的字符串属性，并将结果封装成一个数组。

```
const SYM = Symbol();

const o = { a: 1, b: 2, c: 3, [SYM]: 4 };

Object.keys(o).forEach(prop => console.log('${prop}: ${o[prop]}'));
```

这个例子的结果跟使用 for...in 循环的结果一样（不需要检查 hasOwnProperty）。当需要一个由对象中所有属性的键组成的数组时，使用它就会很方便。例如，它可以很容易地列出某个对象中所有以字符 x 开头的属性。

```
const o = { apple: 1, xochitl: 2, balloon: 3, guitar: 4, xylophone: 5,};

Object.keys(o)
  .filter(prop => prop.match(/^x/))
  .forEach(prop => console.log('${prop}: ${o[prop]}'));
```

9.2 面向对象编程

面向对象编程（OOP）是计算机科学中一个年代久远的编程范式。现在所知的一些OOP概念最开始出现于上个世纪50年代，可是直到1967年Simula及后来Smalltalk语言的出现，可辨识的OOP才应运而生。

OOP的基础理念非常简单直观：对象是一个逻辑相关的数据和功能的集合。它以人类对世界的自然理解为设计理念。好比一辆车，它有数据（品牌、型号、门数、VIN，等等）和功能（加速、移动、开门、开大灯，等等）。此外，OOP使得编程人员能够以抽象（一辆车）和具体（某辆具体的车）的思维去思考事物。

在开始之前，先了解一些跟OOP相关的基本词汇。*类*指的是通用的东西（车）。*实例*（或者对象实例）指具体的东西（一辆具体车，比如“My Car”）。功能（加速）称作*方法*。跟类相关，但不涉及特定实例的功能叫作*类方法*（比如，“create new VIN”可能就是一个类方法：它不特指某辆具体的车，的确，我们并不希望某辆具体的车能够创建一个新的、合法的VIN）。

OOP还提供了一个层次分明的类继承框架。例如，可以存在一个更加通用的*交通工具*类。每个交通工具可能有行驶范围（最大续航距离），但不像汽车，它可能有轮子（船也是一种交通工具，但它很可能没有轮子）。可以说交通工具是车的父类，车是交通工具的子类。而交通工具类可能有多个子类：车、船、飞机、摩托车，等等。同样，这些子类也可能还有子类。例如，船可能会有它自己的子类，帆船、划艇、独木舟、拖船、摩托艇等。

下面会使用汽车作为例子来贯穿本章，因为它是每个人在现实生活中都可能会触及到的（即使不感兴趣）真实对象。

9.2.1 创建类和实例

在ES6之前，JavaScript中创建类的过程比较繁琐且不直观。ES6新引入了一些创建类的便捷语法：

```
class Car {
  constructor() {
  }
```

```
}
```

这段代码创建了一个名为 Car 的类。还没有创建它的任何实例（具体的车），不过现在已经能够创建实例了。使用关键字 new 可以创建一个具体的 car：

```
const car1 = new Car();
const car2 = new Car();
```

上述代码创建了两个 Car。在将 Car 类复杂化之前，来看看 instanceof 运算符，这个运算符可以告诉大家一个给定的对象是否属于某个类。

```
car1 instanceof Car         // true
car1 instanceof Array       // false
```

从上面的代码中，可以看到 car1 是一个 Car 的实例，而非数组的实例。接下来把 Car 类变得更有趣一些。给它一些数据（品牌，型号），以及一些功能（移动）。

```
class Car {
   constructor(make, model) {
      this.make = make;
      this.model = model;
      this.userGears = ['P', 'N', 'R', 'D'];
      this.userGear = this.userGears[0];
   }
   shift(gear) {
      if(this.userGears.indexOf(gear) < 0)
         throw new Error('Invalid gear: ${gear}');
      this.userGear = gear;
   }
}
```

这里面，this 关键字有它特殊的用途：它引用了方法被调用时所绑定的实例。可以认为它是一个占位符：当在写一个类的时候——抽象类，this 关键字就是某个具体实例的占位符。构造方法允许在创建实例的时候指定车辆的品牌和型号，还可以设置一些默认值：有效的档位（userGears），以及当前档位（gear），它的初始值为第一个有效的档位。（一般把它叫作用户档位，是因为如果这辆车有一个自动变速箱，当车辆在行驶的过程中，会有一个真实的机械齿轮，可能就会不一样了）。除了构造器—在创建对象时候调用，还创建了一个 shift 方法，该方法允许在有效档位中进行切换。下面看一个使用场景：

```
const car1 = new Car("Tesla", "Model S");
const car2 = new Car("Mazda", "3i");
car1.shift('D');
car2.shift('R');
```

上面这个例子，当调用 car1.shift('D')时，this 跟 car1 绑定在一起。同理，

在 car2.shift('R') 的调用中，this 跟 car2 绑定在一起。可以验证 car1 处于前进档（D），car2 处于倒车档（R）。

```
> car1.userGear      // "D"
> car2.userGear      // "R"
```

9.2.2 动态属性

Car 类中的 shift 方法貌似很智能，因为它能防止选择一个无效的档位。不过这种保护也有它的局限性，因为可以直接为它赋值：car1.userGear = 'X'。大部分面向对象语言都会竭尽全力提供一些机制来阻止这种直接赋值被滥用，它们允许为类的方法和属性指定访问级别。而 JavaScript 没有这种机制，这也是 JavaScript 语言中饱受非议的一点。

动态属性可以稍微有效地弥补这种不足[①]。它们具有属性的语义，但同时可以像方法一样被调用。可以通过修改 Car 类来看看它的好处：

```
class Car {
   constructor(make, model) {
      this.make = make;
      this.model = model;
      this._userGears = ['P', 'N', 'R', 'D'];
      this._userGear = this._userGears[0];
   }

   get userGear() { return this._userGear; }
   set userGear(value) {
      if(this._userGears.indexOf(value) < 0)
         throw new Error('Invalid gear: ${value}');
      this._userGear = vaule;
   }

   shift(gear) { this.userGear = gear; }
}
```

聪明的读者会注意到其实还没有解决直接赋值的问题：car1._userGear= 'X'。这个例子中，使用“穷人访问限制”给私有属性加上下划线作为前缀（_）。但这只是一种约定俗成的做法，它能让大家快速识别出哪些代码访问了被保护的属性。

如果确实想要强制属性私有化，可以使用一个 WeakMap 的实例（见第 10 章），它是被作用域保护的（如果不用 WeakMap，私有属性将永远不会跑出作用域，即使是它们引用的实例）。以下代码是通过修改 Car 类，使得当前档位属性真正私有化：

① 动态属性，更准确地说，应该叫访问属性。我们将在第 21 章就访问属性做更深入的探讨。

```
const Car = (function() {

   const carProps = new WeakMap();

   class Car {
      constructor(make, model) {
         this.make = make;
         this.model = model;
         this._userGears = ['P', 'N', 'R', 'D'];
         carProps.set(this, { userGear: this._userGears[0] });
      }

      get userGear() { return carProps.get(this).userGear; }
      set userGear(value) {
         if(this._userGears.indexOf(value) < 0)
            throw new Error('Invalid gear: ${value}');
         carProps.get(this).userGear = value;
      }

      shift(gear) { this.userGear = gear; }
   }

   return Car;
})();
```

这里使用即时调用函数表达式将 WeakMap 隐藏在一个闭包内，从而阻止了外界的访问。这个 WeakMap 可以安全地存储任何不想被 Car 类外部访问的属性。

另一种方式是使用符号代替属性名。它们有一些针对误操作的保护措施，但是类中的符号属性是可以被访问的，所以即使是这种方法也可能会失效。

9.2.3 类即函数

在 ES6 引入关键字 class 之前，要创建一个类，需要创建一个函数充当类的构造方法。虽然 class 的语法更加直观，但 JavaScript 底层的实现方式并没有发生变化（class 只是增加了一些语法糖），所以 JavaScript 中类的表示很重要 。

类实际上就是函数。在 ES5 中，会这样编写 Car 类：

```
function Car(make, model) {
   this.make = make;
   this.model = model;
   this._userGears = ['P', 'N', 'R', 'D'];
   this._userGear = this.userGears[0];
}
```

当然也可以在 ES6 中做这件事：结果完全一样（稍后会讲具体怎么做）。可以尝试

使用两种方式来验证它。

```
class Es6Car {}                    // 简单起见，这里省略了构造器
function Es5Car {}
> typeof Es6Car                    // "function"
> typeof Es5Car                    // "function"
```

所以 ES6 中并没有什么新东西，仅仅是多出了一些更易用的新语法。

9.2.4 原型

在类的实例中，当引用一个方法时，实际上是在引用原型方法。例如，当讨论 Car 的实例中的 shift 方法时，引用的是一个原型方法，并且会看到它被写成 Car.prototype.shift。（类似的，数组的 forEach 函数会被写成 Array.prototype.forEach.）现在是时候好好学习什么是原型了，以及 JavaScirpt 中如何使用原型链进行动态调度的。

使用数字符号（#）来描述原型方法已经成为一种普遍的约定。例如，大家会经常看到 Car.prototype.shift 被简单地写成 Car#shift。

每个函数都有一个叫作 prototype 的特殊属性（可以通过在控制台中输入 **f.prototype** 来将它更改为任意函数）。一般的函数不需要使用原型，但是对于那些作为对象构造器的函数，它就至关重要了。

按照约定，对象的构造器（又指类名）始终以大写字母开头，例如 Car。这个约定并非强制的，但如果试图以大写字母开头来命名某个函数，或者以小写字母给构造器命名时，很多格式检查工具都会发出警告。

当使用关键字 new 来创建一个新的实例时，函数的原型属性就会变得很重要：新创建的对象可以访问其构造器的原型对象。对象实例会将它存储在自己的 __proto__属性中。

__proto__属性是 JavaScript 的内部属性，就像任何被双下划线包围的属性一样。可以用这些属性做一些非常邪恶的事情。有时候它们也有一些聪明并有效的使用场景，但只有在完全理解了 JavaScript 之后才知道如何使用它。强烈建议大家看看这些属性但是不要改变它们。

关于原型，有一个重要的机制叫作动态调度（“调度”是方法调用的另一种说法）。

当试图访问对象的某个属性或者方法时，如果它不存在于当前对象中，JavaScritp 会检查它是否存在于对象原型中。因为同一个类的所有实例共用同一个原型，如果原型中存在某个属性或者方法，则该类的所有实例都可以访问这个属性或方法。

通常不要在类的原型中设置数据属性，因为所有的实例都会共享这些属性的值。但是如果该值被设置在任意一个实例中，那么它就只在特定的实例中存在，而原型中不存在，这会引发困惑和产生 bug。所以如果需要给对象实例初始化一些数据，最好是通过构造器来设置。

注意，在实例中定义的方法或者属性会覆盖掉原型中的定义。记住 JavaScript 的检查顺序是先实例后原型。下面来看看它们的实际用法：

```
// 类 Car 跟之前定义的一样，具备移动方法
const car1 = new Car();
const car2 = new Car();
car1.shift === Car.prototype.shift;              // true
car1.shift('D');
car1.shift('d');                                 // error
car1.userGear;                                   // 'D'
car1.shift === car2.shift                        // true

car1.shift = function(gear) { this.userGear = gear.toUpperCase(); }
car1.shift === Car.prototype.shift;              // false
car1.shift === car2.shift;                       // false
car1.shift('d');
car1.userGear;                                   // 'D'
```

这个例子很清楚地演示了 JavaScirpt 执行动态调度的方式。一开始，对象 car1 没有 shift 方法，但是当调用 car1.shift('D')时，JavaScript 就会查看 car1 的原型，找到一个同名的方法。当用自定义的方法替换掉 shift 后，car1 自身和它的原型都有 shift 方法了。再调用 car1.shift('d')时，实际上调用了 car1 自己的 shift 方法。

大多数时候，不必搞懂原型链和动态调度的机制。但在不同的阶段，可能会遇到一些迫使大家去深入学习它们的问题。最好在学习的时候能都明白其中的细节。

9.2.5 静态方法

目前为止，讨论的方法都是*实例方法*。也就是说，它们只针对每个具体的实例才有用。还有一种*静态方法*（也叫*类方法*），它不与实例绑定。在静态方法中，this 绑定的是类本身，但通常使用类名来代替 this 是公认的最佳实践。

静态方法通常用来执行一些与类相关的任务，而非跟具体的实例相关。下面将沿用

汽车 VINs（车辆识别码）的例子。单独的汽车能够产生自己的 VIN 显然是不合理的，那么如何防止两辆汽车使用相同的 VIN 呢？分配 VIN 是一个对于大部分汽车都适用的抽象概念，因此，这是一个适合静态方法的场景。另外，静态方法通常是一些用来操作多辆汽车的方法。例如，可能希望有一个叫 areSimilar 方法来判断两辆汽车是否具有相同的品牌和型号，或者 areSame 方法来判断两辆车是否具有相同的 VIN。下面来看看这些静态方法在 Car 中的实现：

```
class Car {
   static getNextVin() {
      return Car.nextVin++;          // 也可以使用 this.nextVin++,
                                     //这里使用 Car 是为了强调这是一个静态方法
   }
   constructor(make, model) {
      this.make = make;
      this.model = model;
      this.vin = Car.getNextVin();
   }
   static areSimilar(car1, car2) {
      return car1.make===car2.make && car1.model===car2.model;
   }
   static areSame(car1, car2) {
      return car1.vin===car2.vin;
   }
}
Car.nextVin = 0;

const car1 = new Car("Tesla", "S");
const car2 = new Car("Mazda", "3");
const car3 = new Car("Mazda", "3");

car1.vin;      // 0
car2.vin;      // 1
car3.vin       // 2

Car.areSimilar(car1, car2);     // false
Car.areSimilar(car2, car3);     // true
Car.areSame(car2, car3);        // false
Car.areSame(car2, car2);        // true
```

9.2.6 继承

在分析原型的时候，已经看到了继承的身影：当创建一个类的实例时，它继承了类原型中所有的功能。尽管如此，它并没有就此打住：如果一个方法没有在对象原型中找到其定义，它会检查原型的原型。这样就建立了一个原型链。JavaScript 会沿着原型链走下去，直到某个原型满足了需求。如果找不到这样的原型，程序最终会

报错。

原型链最大的好处是能够建立类的层次结构。前面已经讨论了汽车如何归属于一种交通工具。原型链允许将功能置于最合适的继承层次上。例如，汽车可能有个方法叫 deployAirbags，可以将它当做一般的交通工具方法，但是，大家坐过配备安全气囊的船吗？另一方面，几乎所有的交通工具都可以搭载乘客，所以交通工具都可能有 addPassenger 的方法（可以在超载时抛出异常）。下面看看如何在 JavaScript 中实现这种场景：

```
class Vehicle {
   constructor() {
      this.passengers = [];
      console.log("Vehicle created");
   }
   addPassenger(p) {
      this.passengers.push(p);
   }
}

class Car extends Vehicle {
   constructor() {
      super();
      console.log("Car created");
   }
   deployAirbags() {
      console.log("BWOOSH!");
   }
}
```

上面这段代码中，第一次出现了 extends 关键字，这个语法标志着 Car 是 Vehicle 的子类。其次 super()也是之前没有见过的。在 JavaScript 中，super 是一个特殊的函数，它调用了父类的构造器。子类必须调用这个方法，否则会报错。

看看下面这个例子：

```
const v = new Vehicle();
v.addPassenger("Frank");
v.addPassenger("Judy");
v.passengers;                          // ["Frank", "Judy"]
const c = new Car();
c.addPassenger("Alice");
c.addPassenger("Cameron");
c.passengers;                          // ["Alice", "Cameron"]
v.deployAirbags();                     // 报错
c.deployAirbags();                     // "BWOOSH!"
```

注意，可以在 c 上调用 deployAirbags，但是不能在 v 上调用。换言之，继承是单向的。Car 类的实例可以访问所有 Vehicle 类的方法，反之却不行。

9.2.7 多态

多态这个吓人的词汇，是面向对象的一个术语，意思是一个实例不仅是它自身类的实例，也可以被当做它的任何父类的实例来使用。在很多面向对象语言中，多态是 OOP 中的一个很特殊的特性。而在 JavaScript 中，因为没有类型的概念，所以任何对象可以在任何地方被使用（虽然不能保证正确的结果），从这个角度来说，JavaScript 具备终极的多态性。

在 JavaScript 中，所编写的代码经常采用某种形式的鸭子类型。这个方法来自于一句话“如果它走起来像鸭子，并且叫起来像只鸭子...那么它很可能就是一只鸭子”。我们继续沿用 Car，如果一个对象含有 deployAirbags 方法，可能会被合理地认为是一个 Car 的实例。或对或错，但它至少是个很明显的提示。

JavaScript 提供了 instanceof 运算符，它会指出某个对象是否属于某个给定类。也许不够智能，但是只要没有设置 prototype 和__proto__属性，它就是可靠的。

```
class Motorcycle extends Vehicle {}
const c = new Car();
const m = new Motorcyle();
c instanceof Car;                    // true
c instanceof Vehicle;                // true
m instanceof Car;                    // false
m instanceof Motorcycle;             // true
m instanceof Vehicle;                // true
```

JavaScript 中所有对象都是基类 Object 的实例。也就是说，对于任何对象 o，o instanceof Object 的结果为 true（除非明确地设置了__proto__属性，而开发人员不应该这么做）。这个其实没有什么实际的效果，它的主要是为了提供一些对象都必须具备的重要方法，比如本章后面会讲到的 toString。

9.2.8 枚举对象属性，回顾

前面已经学习了如何使用 for...in 来枚举对象的属性，也学习了原型继承，完全理解 hasOwnProperty 在枚举对象属性时的用途。对于对象 obj 和属性 x，如果 obj 中存在 x 属性，obj.hasOwnProperty(x) 返回 true。反之，如果属性没有定义或者定义在原型链中，obj.hasOwnProperty(x) 结果为 false。

如果按照正确的用法使用 ES6 中的类，数据属性始终被定义在实例中，而不在原型链中。然而，因为没有办法防止在原型上直接添加属性，所以最好使用 hasOwnProperty 做校验。看看这个例子：

```
class Super {
   constructor() {
      this.name = 'Super';
      this.isSuper = true;
   }
}

// 合法，但不推荐这么做
Super.prototype.sneaky = 'not recommended!';

class Sub extends Super {
   constructor() {
      super();
      this.name = 'Sub';
      this.isSub = true;
   }
}

const obj = new Sub();

for(let p in obj) {
   console.log('${p}: ${obj[p]}' +
      (obj.hasOwnProperty(p) ? '' : ' (inherited)'));
}
```

如果运行这个程序，会看到：

```
name: Sub
isSuper: true
isSub: true
sneaky: not recommended! (inherited)
```

name，isSuper 和 isSub 三个属性都被定义在实例中，而不是在原型链中（注意在父类构造器中定义的属性也会被定义在子类的实例中）。另一方面，属性 sneaky 被手动添加到父类的原型中。

使用 Object.keys 就可以完全避免这个问题，因为 Object.keys 只包含了原型中定义的属性。

9.2.9 字符串表示

每个对象最终都是继承自 Object 类，所以可以在 Object 中使用的方法在任何

其他对象中都可以用。其中的一个方法是 toString，该方法是为了给对象提供一个默认的字符串表示。toString 默认的方法会返回"[object Object]"，这种表示并没有什么实际用处。

在调试的时候，添加一个用于返回对象的描述信息的 toString 方法将会很有用，这可以一下子就获取对象的重要信息。例如，可以修改本章前面出现的 Car 类，为其增加一个 toString 方法，用来返回品牌、型号和 VIN：

```
class Car {
  toString() {
    return '${this.make} ${this.model}: ${this.vin}';
  }
  //...
```

现在调用 Car 实例的 toString 方法会返回该对象的一些身份信息。

9.3 多继承、混合类和接口

一些面向对象语言支持多继承，也就是说一个类可以有两个直接的父类（不像某个类有一个父类，而该父类又有一个父类）。多继承有引发冲突的风险。好比说，如果某个类继承自两个父类，而两个父类都有 greet 方法，那么子类该继承谁的方法呢？所以很多编程语言选择单继承来避免这个棘手的问题。

然而，当思考现实世界的问题时，多继承往往是有意义的。例如，汽车可能继承自交通工具和“可接受保险的”（可以给一辆汽车或房子上保险，但是房子肯定不属于交通工具）。不支持多继承的编程语言通常会引入接口的概念来绕过这个问题。一个类（Car）只能有一个父类（Vehicle），但它可以有多个接口（Insurable、Container 等等）。

JavaScript 是一种有趣的混合。技术上，它是一个单继承的语言，因为在原型链并不会去寻找多个父类，但是它确实提供了一些方法，有时候这些方法比多继承或者接口还要好用（当然也不全是这样）。

多继承产生问题的主要原因在于混入的概念。混入是指功能按需“混合”。因为 JavaScript 是一门弱类型且语法自由的语言，任何时候都可以将任意功能混合到任何对象中。

下面创建一个“可接受保险的”混合类，同时保证它可以用在汽车类上。当然要尽量简化它。除了可接受保险的混合类，还要创建一个 InsurancePolicy 类。可接受保险的混合类需要 addInsurancePolicy、getInsurancePolicy 和

isInsured 方法，接下来看看它是如何工作的：

```
class InsurancePolicy() {}
function makeInsurable(o) {
    o.addInsurancePolicy = function(p) { this.insurancePolicy = p; }
    o.getInsurancePolicy = function() { return this.insurancePolicy; }
    o.isInsured = function() { return !!this.insurancePolicy; }
}
```

现在可以让任何对象变成可接受保险的。那么对于汽车，需要做什么保险呢？大家的第一反应可能是这样子：

```
makeInsurable(Car);
```

大家可能会被下面这个粗暴的方式吓到：

```
const car1 = new Car();
car1.addInsurancePolicy(new InsurancePolicy());        // 报错
```

如果读者认为“当然是错的，因为 addInsurancePolicy 方法并不在原型链中，”那就再去看看这个类的头部。它对使汽车可以接受保险没有好处，而且没有意义：汽车这个抽象概念并不是可接受保险的，但是具体的汽车可以。所以下一步做法可能是这样的：

```
const car1 = new Car();
makeInsurable(car1);
car1.addInsurancePolicy(new InsurancePolicy());        // 正常运行
```

这样就可以正常工作了，但是现在必须记住为每个实例化的汽车调用 makeInsurable。可以在 Car 类的构造方法中进行调用，这样的话，相当于是在给每一辆汽车重复这一个动作。幸好，解决该问题并不难：

```
makeInsurable(Car.prototype);
const car1 = new Car();
car1.addInsurancePolicy(new InsurancePolicy());        // 正常运行
```

现在，好像这些方法已经成为 Car 类的一部分。并且，从 JavaScript 的角度来看，它们确实如此。从开发的角度看，让这两个重要的类的维护变得更简单了，汽车工程组负责管理和开发 Car 类，保险组负责管理 InsurancePolicy 类和 makeInsurable 混合方法。当然，两个组仍然存在互相干扰的空间，但这总比让每个人都工作在一个巨大的 Car 类中要好的多。

混合方法并不能消除冲突：如果出于某些原因，保险组需要在他们的混合类中创建一个 shift 方法，就会破坏 Car 类。同时，不能使用 instanceof 来鉴别对象是否是可接受保险的。最佳实践是使用鸭子类型（如果它有个方法叫

addInsurancePolicy，那么它一定是可接受保险的）。

可以使用符号来缓解这些问题。假设保险组会不断增加一些通用的方法，而且无意间破坏了 Car 类中的方法。可以要求它们使用符号作为键。它们的混合方法会是这样：

```
class InsurancePolicy() {}
const ADD_POLICY = Symbol();
const GET_POLICY = Symbol();
const IS_INSURED = Symbol();
const _POLICY = Symbol();
function makeInsurable(o) {
   o[ADD_POLICY] = function(p) { this[_POLICY] = p; }
   o[GET_POLICY] = function() { return this[_POLICY]; }
   o[IS_INSURED] = function() { return !!this[_POLICY]; }
}
```

由于符号是唯一的，这就保证了混合方法将不会干扰到 Car 类中已有的功能。这种用法略显笨拙，但更安全。一个折中的办法可能是，字符串用于定义方法，而符号（例如_POLICY）用于定义数据属性。

9.4 小结

面向对象编程是一个极其流行的编程范式，而且它确实很实用。对于很多现实世界中的问题，它鼓励对代码进行组织和封装，以便维护、调式和修复。JavaScript 对 OOP 的实现受到诸多批评，甚至有人说它不符合面向对象的语言定义（通常是因为缺乏对数据访问的控制）。这些批评也有道理，但是一旦习惯了 JavaScript 的 OOP，会发现它实际上十分灵活和强大。而且，它还能做一些其他面向对象语言做不到的事情。

第 10 章

maps 和 sets

ES6 引入了两个非常受欢迎的数据结构：*maps* 和 *sets*。maps 跟对象类似，可以用键（key）来 map 值（value），sets 则类似于数组，只不过不允许出现重复值。

10.1 maps

在 ES6 之前，当需要把键和值映射起来的时候，一般会首选对象，因为对象能把字符串类型的键映射到任意类型的对象。不过出于这个原因而使用对象会有很多弊端：

- 对象原型中可能存在并不需要的映射。
- 弄清楚对象中有多少映射并没有那么简单。
- 因为键必须是一个字符串或者符号，这样一来就不能将对象映射到值。
- 对象不能保证自身属性的顺序。

Map 对象帮助解决了这些问题，它是将键和值映射起来的（即使键是一个字符串）的绝佳选择。例如：当想把 user 对象映射到 role 的时候：

```
const u1 = { name: 'Cynthia' };
const u2 = { name: 'Jackson' };
const u3 = { name: 'Olive' };
const u4 = { name: 'James' };
```

从创建一个 map 对象开始：

```
const userRoles = new Map();
```

可以使用 map 中的 set()方法把 user 赋给 role：

```
userRoles.set(u1, 'User');
userRoles.set(u2, 'User');
userRoles.set(u3, 'Admin');
// poor James...we don't assign him a role
```

链式调用 set()方法，还可以节省打字的时间：

```
userRoles
   .set(u1, 'User')
   .set(u2, 'User')
   .set(u3, 'Admin');
```

还可以给 map 的构造函数传一个包含了数组的数组：

```
const userRoles = new Map([
   [u1, 'User'],
   [u2, 'User'],
   [u3, 'Admin'],
]);
```

现在如果想知道 u2 中有什么 role，使用 get()方法就行：

```
userRoles.get(u2);      // "User"
```

如果调用的 key 在 map 中不存在，就会返回一个 undefined。当然，也可以用 has()方法来查看 map 中是否包含给定的 key。

```
userRoles.has(u1);    // true
userRoles.get(u1);    // "User"
userRoles.has(u4);    // false
userRoles.get(u4);    // undefined
```

如果 key 已经在 map 中了，那么调用 set()后 key 对应的 value 就会被替换：

```
userRoles.get(u1);            // 'User'
userRoles.set(u1, 'Admin');
userRoles.get(u1);            // 'Admin'
```

size 属性返回 map 中的元素个数：

```
userRoles.size;               // 3
```

使用 keys()方法可以拿到 map 中所有的键，values()可以拿到所有的值，entries()则可以以数组的方式获取键值对，数组的第一个元素为键，第二个元素为值。所有这些方法都返回一个可以迭代的对象，从而能用 for…of 循环来迭代：

```
for(let u of userRoles.keys())
```

```
    console.log(u.name);

for(let r of userRoles.values())
    console.log(r);

for(let ur of userRoles.entries())
    console.log('${ur[0].name}: ${ur[1]}');

// 这里可以通过解构让迭代更自然
for(let [u, r] of userRoles.entries())
    console.log('${u.name}: ${r}');

// entries()方法是Map的默认迭代器，所以上例可以简写为：
for(let [u, r] of userRoles)
    console.log('${u.name}: ${r}');
```

如果需要一个数组（而不是一个可迭代的对象），可以使用展开运算符：

```
[...userRoles.values()];    // [ "User", "User", "Admin" ]
```

使用 `delete()` 方法可以删除 map 中的一个条目：

```
userRoles.delete(u2);
userRoles.size;             // 2
```

最后，如果想删除 map 中的所有条目，可以调用 clear()方法：

```
userRoles.clear();
userRoles.size;             // 0
```

10.2 Weak maps

WeakMap 跟 Map 在本质上是相同的，除了以下几点：

- key 必须是对象。
- `WeakMap` 中的 key 可以被垃圾回收。
- `WeakMap` 不能迭代或者清空。

通常，只要还有地方在引用某个对象，JavaScript 就会将它保留在内存中。例如：如果有一个对象是 `Map` 中的 `key`，那么只要这个 `Map` 存在，这个对象就会一直在内存中。但 `WeakMap` 却不是这样。正因为如此，`WeakMap` 不能被迭代（因为在迭代中，暴露处于垃圾回收过程中的对象，是非常危险的）。

正是因为 `WeakMap` 具备这些特性，才可以用它存储对象实例中的私有 key。

```
const SecretHolder = (function() {
   const secrets = new WeakMap();
   return class {
      setSecret(secret) {
         secrets.set(this, secret);
      }
      getSecret() {
         return secrets.get(this);
      }
   }
})();
```

这里把 WeakMap 放在 IIFE 中，同时还放入了一个使用它的类。在 IIFE 外，有一个叫作 SecretHolder 的类，这个类的实例可以存储 secrets。这样一来，secret 的赋值和取值只能分别通过 setSecret 方法和 getSecret 方法完成：

```
const a = new SecretHolder();
const b = new SecretHolder();

a.setSecret('secret A');
b.setSecret('secret B');

a.getSecret();    // "secret A"
b.getSecret();    // "secret B"
```

这里也可以用普通的 Map，但是这样会导致 SecretHolder 实例中的 secret 永远不会被垃圾回收！

10.3 sets

set 是一个不允许重复数据的集合（跟数学中集合的概念一样）。这里继续沿用前面的例子，如果想要给 user 赋多个 role。例如：所有 user 都会有一个“User”role，但是管理员既有“User”role，也有“Admin”role。不过，从逻辑上讲，一个 user 拥有重复的 role 很不合理。而 set 就是处理这种情况的理想数据结构，下面来看看它的实现：

首先，创建一个 Set 实例：

```
const roles = new Set();
```

如果想添加一个 user role，可以用 add() 方法：

```
roles.add("User");     // Set [ "User" ]
```

如果想把这个 user 变成管理员，继续调用 add() 方法：

```
roles.add("Admin");       // Set [ "User", "Admin" ]
```

跟 Map 一样，Set 也有 size 属性：

```
roles.size;               // 2
```

这就是 set 的优美之处：不需要在添加元素的时候检查 set 中是否已经有这个元素了。如果添加早已存在于 set 中的值，什么都不会发生：

```
roles.add("User");        // Set [ "User", "Admin" ]
roles.size;               // 2
```

删除 role 的时候，调用 delete() 方法就行，当它返回 true 的时候表示这个 role 在 set 中，否则返回 false。

```
roles.delete("Admin");          // true
roles;                          // Set [ "User" ]
roles.delete("Admin");          // false
```

10.4 Weak sets

Weak sets 只能包含对象，这些对象可能会被垃圾回收。跟 WeakMap 类似，WeakSet 中的值不能被迭代，这就让 weak sets 变得很特殊；所以很难找到它的用例。事实上，weak sets 的唯一用处是判断给定对象是不是一个 set。

例如：圣诞老人可能会有一个叫作调皮（naughty）的 WeakSet，这样他就知道该给谁送煤块（不调皮的孩子才会得到正常的礼物）：

```
const naughty = new WeakSet();

const children = [
   { name: "Suzy" },
   { name: "Derek" },
];

naughty.add(children[1]);

for(let child of children) {
   if(naughty.has(child))
      console.log('Coal for ${child.name}!');
   else
      console.log('Presents for ${child.name}!');
}
```

10.5 打破对象习惯

如果读者是一名有经验的 JavaScript 程序员，同时也是 ES6 新手，在需要映射时，对象很可能会成为首选。毫无疑问，此时你已经学会了所有的技巧来避开将对象当做 map 来使用时的陷阱。但是现在有了真正的 map，它们就应该被使用起来！同样，读者可能已经习惯了将属性值为布尔值的对象当做集合来使用，而从现在开始，不用再那么做了。当正在创建一个对象时，先问问自己，“这个对象仅仅是为了创建映射吗？”如果回答“是”，那么该考虑使用 Map 了。

第 11 章

异常和错误处理

每个人都希望生活在不会出错的世界里，但这只是一种奢望。即便是最小的应用，都难免因为一些无法事先预料的情况而产生错误。所以，编写能够稳定运行的高质量软件的第一步，就是要承认软件会有错误。第二步就是提前识别出那些错误，并以恰当的方式处理它们。

*异常处理*是一种以可控的方式处理错误的机制。它之所以叫作异常处理（而不是错误处理），是因为其本意是处理异常情况。也就是说，它并非想处理预期的错误，而是预期之外的错误。

预期错误和非预期错误（异常）之间往往界线模糊，情景各异。相比于那些为受过培训的用户设计的应用，那些被未经过培训的普通用户使用的应用可能面临更多不可预测的行为。

这里有一个关于预期错误的典型例子—用户在表单中填了一个无效的邮件地址，毕竟，人总会有拼写错误的时候。而非预期错误的典型例子可能会是：磁盘空间不足，或者一个平日里很稳定的服务突然不可用了。

11.1 Error 对象

JavaScript 有一个内建的 Error 对象，它可以用来处理任意类型的错误（异常或预期错误）。还可以在创建 Error 实例的时候提供一些错误信息：

```
const err = new Error('invalid email');
```

创建出的 `Error` 实例本身不会做任何事。它只提供一个传递错误的载体。假设有一个验证邮箱地址的函数。如果验证成功，函数返回字符串格式的邮箱。否则返回

一个 Error 实例。为了方便起见，把任何包含@符号的东西看做是合法邮箱（见第 17 章）：

```
function validateEmail(email) {
   return email.match(/@/) ?
      email :
      new Error('invalid email: ${email}');
}
```

为了使用返回值，可以用 typeof 运算符来判断返回的是不是 Error 实例。然后通过 Error 的 message 属性来获取错误信息：

```
const email = "jane@doe.com";

const validatedEmail = validateEmail(email);
if(validatedEmail instanceof Error) {
   console.error(`Error: ${validatedEmail.message});
} else {
   console.log('Valid email: ${validatedEmail}');
}
```

虽然这样使用 Error 实例完全合法，也很有用，但实际上，它的大部分应用场景都在异常处理中，这也是接下来要讲的内容。

11.2 使用 try 和 catch 处理异常

使用 try…catch 语句可以完成异常处理。它的理念是：首先“尝试”去做一些事情，一旦在做的过程中发生异常，就“捕获”这些异常。在前面的例子中，即便 validateEmail 处理了那些邮件中没有包含@的预期错误，但还是有可能产生非预期的错误：比如，某个开发人员不小心给 email 赋了一个非字符串的值。从代码来看，如果将前面例子中的 email 设为 null、数字、对象等任何非字符串的值时都会出错，此时程序将会非常不友好地崩溃掉。为了防范这种非预期错误，可以将用于验证邮箱的代码封装在 try…catch 语句中，例如：

```
const email = null; // whoops

try {
   const validatedEmail = validateEmail(email);
   if(validatedEmail instanceof Error) {
      console.error(`Error: ${validatedEmail.message});
   } else {
      console.log('Valid email: ${validatedEmail}');
   }
} catch(err) {
```

```
    console.error('Error: ${err.message}');
}
```

捕获异常后，程序就不会再崩溃了，而是打印了错误日志后继续执行。不过可能还会有别的问题：如果这里需要输入一个合法的邮箱，而用户输入了无效的邮箱，那程序继续运行下去也没有意义了。不过，至少现在我们可以用一种更优雅的方式处理错误了。

注意，一旦有错误产生，执行逻辑就会立即跳转到 `catch` 块中。也就是说，`validateEmail` 调用语句后面的 `if` 语句不会再执行。也可以在 `try` 块中写入任何期望的语句，最先产生错误的语句会使执行逻辑跳转到 `catch` 块中。如果 `try` 块中的语句没有任何错误，`catch` 块中的代码就不会被执行，程序会继续运行下去。

11.3 抛出异常

在前面的例子中，用 `try...catch` 语句捕获 JavaScript 自身产生的错误（比如，当试图让非字符串调用 `match` 方法时）。也可以自己“抛出”（或“向上抛出”）错误，此时异常处理机制就会被启动。

不像其他编程语言中的异常处理，在 JavaScript 中，可以抛出任何值：数字、字符串，或其他任何类型。不过抛出 `Error` 实例可以更方便地处理异常。所以大多数 `catch` 块都期望捕获一个 `Error` 实例。始终记住一点，没法准确地知道抛出的错误会在什么时候被捕获（编写的函数可能会被其他人使用，而他们会理所当然地认为这个函数抛出的任何错误都是 `Error` 实例）。

例如，如果需要给一个银行应用开发付款功能，可能会在账户余额不足的时候抛出一个异常（这是意料之中的异常，因为在买单之前，首先应该检查账户余额够不够）：

```
function billPay(amount, payee, account) {
    if(amount > account.balance)
        throw new Error("insufficient funds");
    account.transfer(payee, amount);
}
```

调用 `throw` 的时候，当前函数会立即停止执行（所以在本例中，`account.transfer` 将不会被调用，这也是期望的结果）。

11.4 异常处理和调用栈

一个标准的应用程序中会存在很多函数调用，而这些函数又会调用其他函数，继而调用更多的函数，以此类推。JavaScript 解释器需要追踪这些函数调用。如果函数 a 调用了函数 b，函数 b 调用了函数 c，那么当函数 c 结束的时候，执行逻辑会返回到 b，而当 b 结束的时候，执行逻辑则会返回到 a。当 c 正在执行的时候，a 和 b 都未“完成”。这种未完成的嵌套函数调用就叫作*调用栈*。

如果函数 c 出错了，那么 a 和 b 会怎样呢？一旦 c 出错，b 也会出错（因为 b 可能会依赖 c 的返回值），这也会让 a 出错（因为 a 可能会依赖 b 的返回值）。本质上，错误会沿着调用栈传递，直到被捕获。

错误可以在调用栈中的任一级别被捕获，如果它们没有被捕获，JavaScript 解释器就会强行终止程序。这种错误被称为*未被处理的异常*或者*未被捕获的异常*，它会使程序崩溃。由于程序中可能出错的地方太多了，想要捕获所有可能出现的错误不仅很困难，而且不现实，这也是程序会崩溃的原因。

当错误被捕获后，调用栈会提供一些用来诊断错误的有用信息。例如，如果函数 a 调用了函数 b，b 又调用了 c，而 c 中出错了，此时调用栈不仅说明函数 c 中出错了，同时会说明错误是在被 b 调用的时候发生的，而这些都发生在被 a 调用的时候。当程序中有很多地方都调用了 c 的时候，这个信息就非常有价值。

在大多数 JavaScript 实现中，Error 实例包含了一个 stack 属性，它是调用栈的字符串形式（它不是标准的 JavaScript 特性，不过却可以用在绝大多数环境中）。了解了这些知识后，可以写一个异常处理的例子演示这些功能：

```
function a() {
   console.log('a: calling b');
   b();
   console.log('a: done');
}
function b() {
   console.log('b: calling c');
   c();
   console.log('b: done');
}
function c() {
   console.log('c: throwing error');
   throw new Error('c error');
   console.log('c: done');
}
```

```
function d() {
   console.log('d: calling c');
   c();
   console.log('d: done');
}

try {
   a();
} catch(err) {
   console.log(err.stack);
}

try {
   d();
} catch(err) {
   console.log(err.stack);
}
```

在 Firefox 中运行这段代码，将会在控制台中看到以下输出：

```
a: calling b
b: calling c
c: throwing error
c@debugger eval code:13:1
b@debugger eval code:8:4
a@debugger eval code:3:4
@debugger eval code:23:4

d: calling c
c: throwing error
c@debugger eval code:13:1
d@debugger eval code:18:4
@debugger eval code:29:4
```

出现@符号的行是栈轨迹，它会从“最深层”函数开始（c），直到没有函数调用（浏览器自身）。可以看到，这里有两个不同的栈轨迹：一个显示了在 b 中调用了 c，而 b 又被 a 调用，另一个则显示 c 直接被 d 调用。

11.5 try...catch... finally

很多时候，try 块中会包含一些对资源的引用，比如，HTTP 连接或者文件。不管有没有发生错误，都需要释放这些资源，防止应用程序永远占用着资源。由于 try 块中可以包含很多语句，任何一条都有可能发生错误，所以在这里释放资源并不安全（因为错误可能发生在释放资源之前，这样就没有机会释放资源了）。同样，在 catch 中释放资源也不安全，因为 catch 中的代码只有在发生错误的时候才会执行。这时

finally 就派上用场了，不管是否发生错误，finally 中的代码都会被执行。

由于还没有讲文件处理和 HTTP 连接，所以这里只是用一个简单的包含了 console.log 的语句来演示 finally 的作用：

```
try {
   console.log("this line is executed...");
  throw new Error("whoops");
    console.log("this line is not...");
} catch(err) {
   console.log("there was an error...");
} finally {
   console.log("...always executed");
   console.log("perform cleanup here");
}
```

分别在有 throw 语句和没有 throw 语句的情况下运行这个例子。大家会发现，两种情况下 finally 块中的代码均被执行了。

11.6 让异常成为例外

到目前为止已经讲解了什么是异常处理，以及如何处理异常，大家可能想把它用在所有错误处理中：不论是一般的预期错误，还是那些非预期错误。毕竟，抛出一个错误很简单，当遇到一个不知该如何处理的情况时，“放弃”又是一件很容易的事情。不过，异常处理是有一定成本的。除了异常未被捕获时可能存在的风险（程序崩溃），异常本身也会带来一定的性能开销。因为异常必须“展开”堆栈轨迹，直到遇到 catch 块。这样一来，JavaScript 解释器就需要一些额外的开支。随着计算机运算速度的增加，这一点逐渐变得不需要太过担心了，但是，在频繁使用的执行路径中抛出异常还是会造成一些性能问题。

记住，一旦抛出异常，就一定要捕获它，除非想让程序崩溃，否则绝不能视而不见。在处理那些没有预期到的错误时，最好用控制流语句来处理预期错误，而把异常当做最后一道防线。

第 12 章

迭代器和生成器

ES6 引入了两个非常重要的概念：迭代器和生成器。由于生成器依赖于迭代器，所以本章将从迭代器开始。

迭代器可以粗略地比做书签：它可以帮助用户追踪当前的位置。数组就是一个可迭代对象的例子：它包含多个东西（比如，一本书的书页），同时提供一个迭代器（就像书签一样）。可以把这个比喻再具体化一些：想象有一个叫作 book 的数组，其中的每个元素都是一个代表书页的字符串。为了与本书的格式保持一致，这里使用路易斯 • 卡罗尔所著《爱丽丝梦游仙境》中的“一闪一闪小蝙蝠”的歌词为例。（可以把它想象成那种每页只有一句话的儿童读物。）

```
const book = [
   "Twinkle, twinkle, little bat!",
   "How I wonder what you're at!",
   "Up above the world you fly,",
   "Like a tea tray in the sky.",
   "Twinkle, twinkle, little bat!",
   "How I wonder what you're at!",
];
```

有了 book 数组，就可以通过数组的 values 方法获取迭代器：

```
const it = book.values();
```

继续图书的例子，迭代器（通常很短小）是一个书签，不过它只适用于这本特定的书。此外，我们还没有在别的地方使用它，也没有开始阅读。在“开始阅读”时，调用迭代器的 next 方法，next 返回的对象中有两个属性：value（保存当前“页”）和 done，在阅读过一页后，done 的值就变成 false。由于书只有 6 页，所以很容易模拟阅读的过程：

```
it.next();  // { value: "Twinkle, twinkle, little bat!", done: false }
it.next();  // { value: "How I wonder what you're at!", done: false }
it.next();  // { value: "Up above the world you fly,", done: false }
it.next();  // { value: "Like a tea tray in the sky.", done: false }
it.next();  // { value: "Twinkle, twinkle, little bat!", done: false }
it.next();  // { value: "How I wonder what  you're at!", done: false }
it.next();  // { value: undefined, done: true }
it.next();  // { value: undefined, done: true }
it.next();  // { value: undefined, done: true }
```

这里有几件重要的事情要说明一下。首先，当 next 方法返回本书的最后一页时，它需要说明阅读尚未结束。这也是用书来做比喻的不恰当之处：当读完一本书的最后一页时，才算是读完了。而迭代器的使用范围远远超过书本，想知道什么时候结束并不是那么容易。运行结束的时候，需要注意此时 value 的值是 undefined，仍然可以继续调用 next，只不过它的返回值都是一样的。当迭代过程结束的时候，才算真正结束，此时它就不能再提供数据了[①]。

虽然这个例子不是那么直接，不过大家也应该已经发现了，可以在两次 `it.next()` 调用之间做别的事情。也就是说，它已经预留了操作空间。

如果需要枚举数组，可以使用 `for` 循环，或者 `for...of` 循环。`for` 循环的结构很简单：因为数组中的元素都是可数并且有序的，所以可以用 `index` 变量依次获取数组中的每个元素。那么 `for...of` 循环呢？它是如何在没有 `index` 的情况下完成这个魔法般的过程的？事实证明，它使用了迭代器：`for...of` 循环对任何可以提供迭代器的结构都适用。接下来大家马上就会知道如何利用这一点。首先，来看看如何在 while 循环中用上述的新发现（迭代器）来模拟一个 `for...of` 循环：

```
const it = book.values();
let current = it.next();
while(!current.done) {
   console.log(current.value);
   current = it.next();
}
```

注意，迭代器是不一样的，也就是说，每当创建一个新的迭代器时，就会从头开始，这样就可以在不同的地方使用多个迭代器了：

```
const it1 = book.values();
const it2 = book.values();
// 两个迭代器都未开始
```

① 由于对象有责任提供它们自己的迭代原理，这个我们马上就会看到。所以创建一个“坏的迭代器”来逆转 value 的值也不是不可能；这将被看成一个错误的迭代器。一般来说，我们应该依赖正确的迭代器行为。

```
// 通过 it2 读取两页：
it1.next();  // { value: "Twinkle, twinkle, little bat!", done: false }
it1.next();  // { value: "How I wonder what you're at!", done: false }

// 通过 it2 读取一页：
it2.next();  // { value: "Twinkle, twinkle, little bat!", done: false }

// 通过 it1 读取另一页：
it1.next();  // { value: "Up above the world you fly,", done: false }
```

在本例中，两个迭代器是互相独立的，它们可以分别依照各自的安排进行遍历。

12.1 迭代协议

迭代器本身并不那么有趣，但它却可以支持那些有趣的行为。迭代器协议让任何对象变得可迭代。试想一下，如果你准备创建一个 logging 类，并且把时间戳作为附加消息。在 logging 类内部，使用了一个数组来存储时间戳消息：

```
class Log {
   constructor() {
      this.messages = [];
   }
   add(message) {
      this.messages.push({ message, timestamp: Date.now() });
   }
}
```

看起来还不错……但是如果想对 log 记录进行迭代呢？当然可以这么做，只要访问 `log.messages` 就行，不过如果 log 可以像数组一样直接迭代，不是更好吗？迭代器协议就可以实现这个。迭代器协议是说，如果一个类提供了一个符号方法 `Symbol.iterator`，这个方法返回一个具有迭代行为的对象（比如：对象有 next 方法，同时 next 方法返回一个包含 `value` 和 `done` 的对象），那么这个类就是可迭代的！修改 `Log` 类，给它添加一个 `Symbol.iterator` 方法吧：

```
class Log {
   constructor() {
      this.messages = [];
   }
   add(message) {
      this.messages.push({ message, timestamp: Date.now() });
   }
   [Symbol.iterator]() {
      return this.messages.values();
   }
}
```

下面可以像数组那样迭代 Log 类的实例了：

```
const log = new Log();
log.add("first day at sea");
log.add("spotted whale");
log.add("spotted another vessel");
//...

// 像数组一样迭代 log!
for(let entry of log) {
   console.log(`${entry.message} @ ${entry.timestamp}`);
}
```

在这个例子中，通过从 messages 数组中取出一个迭代器的方式保持迭代器协议，当然也可以编写自己的迭代器：

```
class Log {
   //...

   [Symbol.iterator]() {
      let i = 0;
      const messages = this.messages;
      return {
         next() {
            if(i >= messages.length)
               return { value: undefined, done: true };
            return { value: messages[i++], done: false };
         }
      }
   }
}
```

至此，以上所使用的例子都是迭代预先定义好元素个数的数组：一本书的页数，或者 log 中的日期消息记录。其实，迭代器还可以用来表示那些含有无穷值的对象。

为了演示这个，大家来看一个很简单的例子：菲波那切数列。确切地说，菲波那切数列不难生成，但是它们却依赖之前的数字。对于外行人来说，菲波那切数列就是前两个数字的和。这个序列从 1 和 1 开始，下一个数字是 1+1，也就是 2。再下一个数字是 1+2，也就是 3。第四个数字是 2+3，也就是 5，以此类推。序列看起来是这样的：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...
```

菲波那切数列永远不会停止。同样，应用程序也不可能知道需要多少元素，这就促使它成为了一个使用迭代器的理想应用。这个例子跟之前的例子唯一不同的地方在

于，它的 done 值永远不会是 true：

```
class FibonacciSequence {
   [Symbol.iterator]() {
      let a = 0, b = 1;
      return {
         next() {
            let rval = { value: b, done: false };
            b += a;
            a = rval.value;
            return rval;
         }
      };
   }
}
```

如果将 for...of 循环用在 FibonacciSequence 的一个实例中，就会得到一个无限循环...菲波那切数列永远都用不完！为了防止这个发生，在循环了 10 个元素后加一个 break 语句。

```
const fib = new FibonacciSequence();
let i = 0;
for(let n of fib) {
   console.log(n);
   if(++i > 9) break;
}
```

12.2 生成器

生成器是使用迭代器来控制其运行的函数。一般来说，函数会获取参数然后返回结果，但是函数调用者并没办法控制该函数。当调用一个函数的时候，实际上是放弃了对函数的控制，直到函数返回。有了生成器，就可以在函数执行时对它进行控制。

生成器提供了两种能力：首先，是控制函数执行的能力，使函数能够分步执行；其次，是与执行中的函数对话的能力。

生成器与一般的函数有两个不同的地方：

- 函数可以通过使用域（yield），在其运行的任意时刻将控制权交还给调用方。
- 调用生成器的时候，它并不是立即执行。而是会回到迭代器中。函数会在调用迭代器的 next 方法时执行。

在 JavaScript 中，生成器需要在 function 关键字后添加一个通配符（*）来指明；

其他情况下，它的语法跟普通函数一样。如果一个函数是生成器，就可以在 return 中添加 yield 关键字了。

下面来看一个简单的例子，一个返回彩虹中所有颜色的生成器：

```
function* rainbow() {     // 通配符让它变成了一个生成器
   yield 'red';
   yield 'orange';
   yield 'yellow';
   yield 'green';
   yield 'blue';
   yield 'indigo';
   yield 'violet';
}
```

接下来看看如何调用这个生成器。记住，当调用生成器的时候，实际上是回到了迭代器中。下例中将会调用这个函数，然后逐步调用迭代器：

```
const it = rainbow();
it.next();  // { value: "red", done: false }
it.next();  // { value: "orange", done: false }
it.next();  // { value: "yellow", done: false }
it.next();  // { value: "green", done: false }
it.next();  // { value: "blue", done: false }
it.next();  // { value: "indigo", done: false }
it.next();  // { value: "violet", done: false }
it.next();  // { value: undefined, done: true }
```

因为 rainbow 生成器返回了一个迭代器，所以也可以对它使用 for...of 循环：

```
for(let color of rainbow()) {
   console.log(color):
}
```

这段代码会将彩虹中所有的颜色都输出到控制台中！

12.2.1 yield 表达式和双向交流

之前提到过，生成器可以让它和其调用者进行双向交流。这个功能是通过 yield 表达式实现的。表达式可以计算出值，而 yield 也是一个表达式，所以它一定可以计算出一个值。它计算的是调用方每次在生成器的迭代器上调用 next 时提供的参数（如果有的话）。下面是一个可以进行对话的生成器：

```
function* interrogate() {
   const name = yield "What is your name?";
   const color = yield "What is your favorite color?";
   return `${name}'s favorite color is ${color}.`;
}
```

当调用这个生成器的时候，会得到一个迭代器，而此时这个生成器还没有运行过。调用 next 的时候，它会先运行第一行。然而由于第一行包含了一个 yield 表达式，生成器的控制权必须回到调用方。调用方必须在第一行被执行前调用 next，此时 name 就会接收一个准备传给 next 的值。以下是生成器调用结束时的输出：

```
const it = interrogate();
it.next();        // { value: "What is your name?", done: false }
it.next('Ethan');// { value: "What is your favorite color?", done:false}
it.next('orange');//{value:"Ethan's favorite color is orange.",done:true}
```

图 12-1 展示了在生成器运行时发生的事件顺序。

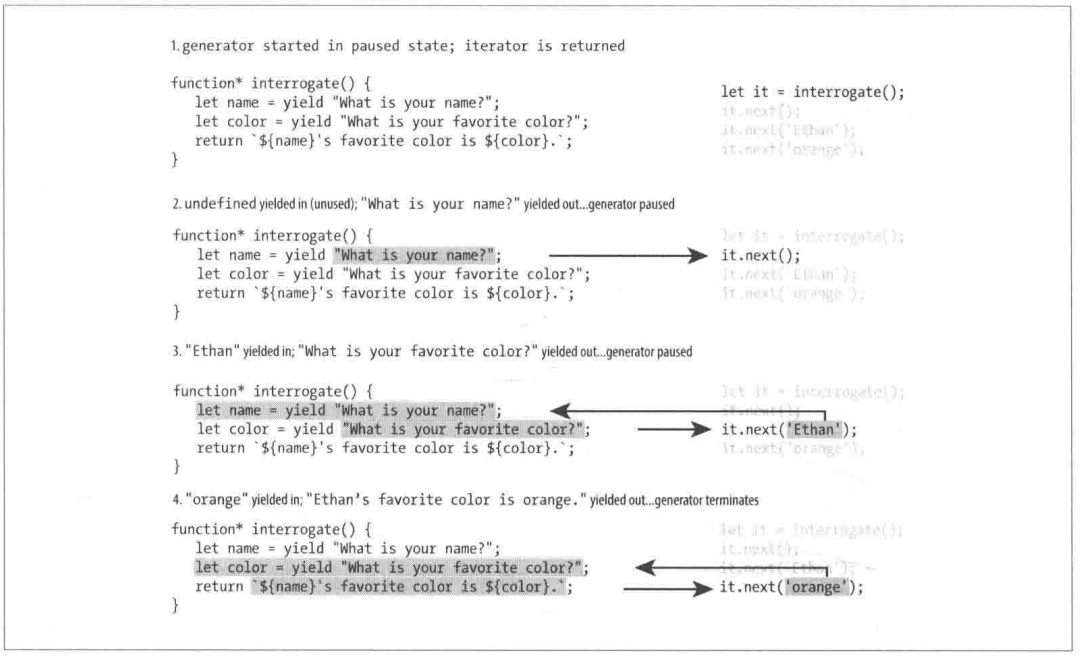


图 12-1　生成器运行时的事件顺序

这个例子演示了生成器强大的功能，它能够让函数在执行过程中被调用方控制。同时，由于调用方可以给生成器传递信息，所以生成器甚至可以根据传递来的信息来修改自身的行为。

不能通过箭头符号来创建生成器，必须使用 function*关键字。

12.2.2 生成器和返回值

yield 表达式本身并不能让生成器结束，即使它是生成器的最后一个语句。在生成器的任何位置调用 return 都会使 done 的值变为 true，而 value 的值则是任何被返回的值。例如：

```
function* abc() {
   yield 'a';
   yield 'b';
   return 'c';
}

const it = count();
it.next();  // { value: 'a', done: false }
it.next();  // { value: 'b', done: false }
it.next();  // { value: 'c', done: true }
```

虽然这是正确的行为，但要记住，使用生成器的东西并不总会在意在 done 为 true 时的 value 值。比如，如果在 for...of 循环中用它，那么“c”是不会被打印的。

```
//将会打印“a”和“b”，但是没有“c”
for(let l of abc()) {
   console.log(l);
}
```

建议不要在 return 中提供一个对生成器有意义的值。如果想在生成器之外使用某个值，应该用 yield；return 应该只被用做提前停止生成器。出于这个原因，通常建议：在生成器中调用 return 的时候，不提供返回值。

12.3 小结

迭代器为集合或对象这类可以提供多个值的数据类型提供了一个标准的方式。而迭代器在 ES6 出现之前是不提供任何东西的，它们只是标准化那些重要且常见的活动。

生成器使得函数更易于控制和定制化：函数调用方不再局限于提供数据后等待函数返回，从而获取函数返回值。生成器实际上是将计算延迟了，只在需要的时候进行。将在第 14 章中看到生成器是如何提供用于管理异步执行的强大模式。

第 13 章

函数和抽象思考的力量

如果将 JavaScript 比作一部百老汇戏剧，那么函数就是剧中最耀眼的明星：它们会抢尽风头，在观众的掌声（也可能有唏嘘声，毕竟无法取悦所有人）中鞠下最后一躬。第 6 章讲解了函数的机制，现在，本书将会进一步探讨函数的使用方式，以及它如何改变你解决问题的方法。

函数的概念就像变色龙：在不同的情况下，表现也会不一样。首先，也是最简单的，我们将以“函数是复用代码的工具”这一视角来思考函数。

13.1 函数作为子程序

子程序是一个非常老的概念，实际上，它是为了管理日趋复杂的程序而做出的让步。的确，如果没有子程序，编程将是一件不断重复的枯燥事情。子程序将一些重复的功能进行简单地封装，并赋予它一个名字，接下来，就可以在任何时候通过引用名字来执行那些功能。

子程序有很多别名，诸如：过程、例行程序、次程序、宏，以及可调用单元这个无比平淡的名字。事实上，在 JavaScript 中，并不用子程序这个名字，就把函数就叫作函数（或者方法）。这里使用术语子程序，是为了强调这种函数用法。

通常，子程序用来封装某个算法，该算法只是一个可被理解的执行单元，用来执行给定任务。下面看一个计算当前年份是否是闰年的算法：

```
const year = new Date().getFullYear();
if(year % 4 !== 0) console.log('${year} is NOT a leap year.')
else if(year % 100 != 0) console.log('${year} IS a leap year.')
```

```
else if(year % 400 != 0) console.log('${year} is NOT a leap year.')
else console.log('{$year} IS a leap year');
```

设想一下，如果要在程序中执行 10 次这段代码，更有甚者，可能会执行 100 次。这时如果要修改打印在 console 中的信息，就必须找到所有使用该代码的实例，然后修改这 4 个输出字符串！这个噩梦可以很方便地通过子程序解决。在 JavaScript 中，函数就可以满足这个需求：

```
function printLeapYearStatus() {
   const year = new Date().getFullYear();
   if(year % 4 !== 0) console.log('${year} is NOT a leap year.')
   else if(year % 100 != 0) console.log('${year} IS a leap year.')
   else if(year % 400 != 0) console.log('${year} is NOT a leap year.')
   else console.log('{$year} IS a leap year');
}
```

上述代码创建了一个名为 printLeapYearStatus 的可复用子程序（函数）。这个大家应该已经很熟悉了。

注意为函数选的名字：printLeapYearStatus。为什么不给它命名为 getLeapYearStatus、leapYearStatus 或 leapYear 呢？虽然这些名字更简短，但它们忽略了一个重要的细节：这个函数只是打印了当前日期是否为闰年这一状态，没有做别的事情。有意义地命名函数可以说是一门科学，或者说是一门艺术。因为名字不是给 JavaScript 看的，它并不关心命名。名字是给其他人看的（或者是将来的自己）。在命名函数时，需要仔细想想如何做到让他人在看函数名的时候就能明白函数的意图。理想情况下，当然希望名字本身就能准确地表达函数的意图。而另一方面，函数名也可能会因此变得过于冗长。例如，可以给这个函数命名为 calculateCurrentLeapYearStatusAndPrintToConsole，但这会因为名字过长而得不偿失。此时，命名的艺术性就体现出来了。

13.2 函数作为有返回值的子程序

在之前的例子中，printLeapYearStatus 是一个普通意义上的子程序：它只包含了一些便于重用的功能，仅此而已。这种简单的函数用法，平时用的其实不多，当编程问题变得更复杂或更抽象时，用的就更少了。接下来进行一些抽象思考，考虑函数作为有返回值的子程序时的情况。

printLeapYearStatus 固然很不错，但是当进一步构建程序时，很快就不满足于将信息输出到控制台上了。比如，想用 HTML 或文件作为输出，或者将当前的闰年状态信息用在其他计算中，此时，子程序就爱莫能助了。可是，当需要知道当

前年份是否为闰年的时候，我们也不想去重新实现算法。

庆幸的是，重写（和重命名）函数，使之变成一个有返回值的子程序很容易。

```
function isCurrentYearLeapYear() {
   const year = new Date().getFullYear();
   if(year % 4 !== 0) return false;
   else if(year % 100 != 0) return true;
   else if(year % 400 != 0) return false;
   else return true;
}
```

再来看一些例子，它们演示了如何使用新函数的返回值：

```
const daysInMonth =
   [31, isCurrentYearLeapYear() ? 29 : 28, 31, 30, 31, 30,
      31, 31, 30, 31, 30, 31];
if(isCurrentYearLeapYear()) console.log('It is a leap year.');
```

在继续之前，想想之前给函数起的名字。给一个返回 boolean 值（或者为在一个具有 boolean 值的上下文中使用而设计）的函数命名时，将 is 作为前缀是很常见的做法。该函数名还包含了单词 *current*，这又是为什么呢？因为这个函数的意图很明显，就是获取当前日期。也就是说，在 2016.12.31 和 2017.1.1 这两天分别运行函数时，它会有不同的返回值。

13.3 函数即……函数

至此我们已经讨论了通过一些更明显的公式来思考函数，是时候以函数来思考函数了。如果你是一个数学家，可能会把函数想象成关系，先有输入，而后有输出。每一个输入都有其对应的输出。那些遵守数学定义的函数会被开发人员称为纯函数。甚至在有些编程语言（例如 Haskell）中，只允许纯函数存在。

纯函数与前面提到的函数有什么不同呢？首先，纯函数必须做到，对同一组输入，始终返回相同的结果。这么看来，`isCurrentYearLeapYear` 就不是一个纯函数，因为它的返回值取决于被调用的时间（某一年可能返回 true，而下一年可能返回 false）。其次，该函数不能有副作用。也就是说，调用该函数不能改变程序的状态。在关于函数的讨论中，还没有碰到产生副作用的函数（一般不把控制台输出当成副作用）。看一个简单的例子：

```
const colors = ['red', 'orange', 'yellow', 'green',
   'blue', 'indigo', 'violet'];
let colorIndex = -1;
function getNextRainbowColor() {
```

```
    if(++colorIndex >= colors.length) colorIndex = 0;
    return colors[colorIndex];
}
```

getNextRainbowColor 函数每次被调用时，都会循环返回彩虹中 7 种颜色中的一种。这个函数不符合纯函数的两大规则：相同的输入（没有参数，所以没有输入）每次返回的结果不同；而且会产生副作用（改变了 color 变量的索引）。很明显，colorIndex 变量不是函数的一部分，而调用 getNextRainbowColor 后它的值却变了，这就是一个副作用。

回到闰年问题上来，怎么把它变成一个纯函数呢？非常简单：

```
function isLeapYear(year) {
    if(year % 4 !== 0) return false;
    else if(year % 100 != 0) return true;
    else if(year % 400 != 0) return false;
    else return true;
}
```

修改后，针对相同的输入，函数的返回值始终一样，并且没有副作用，这样它就变成了一个纯函数。

函数 getNextRainbowColor 更有意思。可以通过将外部变量放入闭包从而消除副作用：

```
const getNextRainbowColor = (function() {
    const colors = ['red', 'orange', 'yellow', 'green',
        'blue', 'indigo', 'violet'];
    let colorIndex = -1;
    return function() {
        if(++colorIndex >= colors.length) colorIndex = 0;
        return colors[colorIndex];
    };
})();
```

这样，就得到了一个没有副作用的函数，但这仍不是一个纯函数，因为对于相同的输入，它的返回值可能不一样。为了解决这个问题，一定要小心地使用该函数。因为可能会反复调用它，例如，每半秒钟就改变一个浏览器中元素的颜色（在第 18 章会学习更多关于浏览器的代码）。

```
setInterval(function() {
    document.querySelector('.rainbow')
        .style['background-color'] = getNextRainbowColor();
}, 500);
```

这段代码看起来也许没那么糟糕，意图也很明显：一些 class 为 rainbow 的 HTML

元素会循环显示彩虹的颜色。然而问题在于，如果其他地方也调用了getNextRainbowColor()，就会和这段代码产生冲突！所以应该停下想想，使用有副作用的函数是否合适。在这个例子中，使用迭代器可能会更好一些：

```
function getRainbowIterator() {
   const colors = ['red', 'orange', 'yellow', 'green',
      'blue', 'indigo', 'violet'];
   let colorIndex = -1;
   return {
      next() {
         if(++colorIndex >= colors.length) colorIndex = 0;
         return { value: colors[colorIndex], done: false };
      }
   };
}
```

这样一来，getRainbowIterator就是一个纯函数：每次的返回值都相同（也就是一个迭代器），并且没有副作用。它的用法变了，但也更安全了：

```
const rainbowIterator = getRainbowIterator();
setInterval(function() {
   document.querySelector('.rainbow')
      .style['background-color'] = rainbowIterator.next().value;
}, 500);
```

大家可能会觉得这种做法有掩耳盗铃的嫌疑：next()方法每次的返回值都不一样啊！确实不一样，但要注意，next()是一个方法，不是函数。它运行在自己所属对象的上下文中，所以它的行为受控于该对象。如果在程序的其他地方调用了 getRainbowIterator，就会生成不同的迭代器，并且各自独立，互不影响。

13.4 那又如何

至此大家已经见识了函数的三种不同形式（子程序、有返回值的子程序，以及纯函数）。是时候停下来问问自己，如此区分的意义何在？

本章的重点不是解释 JavaScript 语法，而是思考为什么。为什么需要函数？用函数来定义子程序就是答案之一：为了避免重复代码。因为子程序可以帮助打包常用功能，多么明显的好处。

通过封装代码来避免重复是一个如此基础的概念，以至于它都有自己的缩略词：DRY（don't repeat yourself，不要重复你自己）。虽然从语言层面看来可能有问题，但会发现人们将这个缩略词当做形容词来描述代码。比如，“这段代码可以更 DRY 一些。”如果有人这样说，他们其实是想说明代码中写了没必要的重复代码。

纯函数稍微难以理解一些，因为它是以一种更抽象的方式来回答“为什么”这个问题的。一个可能的答案是“因为它们使编程更像数学!”不过这也许会引发另一个问题：“为什么编程要像数学呢？这样做有什么好处？”一个更好的回答可能是“因为纯函数让代码更易于测试，更轻量、可读性更高”。

当一个函数在不同情况下返回不同的结果或者会产生副作用，那么称之为上下文相关。举个例子，如果有个函数，它会产生副作用，但是确实有用，当把它从一段程序移到另一段程序后，它就可能失效了。或者更糟糕的是，99%的时间里它能正常工作，但在剩下的 1%时间里造成严重的漏洞。任何开发人员都知道，间歇性的漏洞是最可怕的：它们可以长时间潜伏在系统中，一旦爆发，定位这些漏洞就犹如大海捞针。

如果大家想知道本书是否推荐纯函数，作者只能说：是的，*应该始终优先使用纯函数*。这里用了“优先”这个词，是因为有时候用一个带有副作用的函数会更简单。如果你是个编程初学者，可能会更倾向于经常这样做。并不是说不能这么做，不过更希望大家能停下来思考一下，试试能否想出一种使用纯函数的替代方式。这样一来，随着时间的推移，你会发现自己很自然地倾向于使用纯函数。

在第 9 章中讲过的面向对象编程提供了一种范式，它允许通过严格控制副作用的作用域，从而来以一种受控和合理的方式使用它。

函数即对象

在 JavaScript 中，函数是 `Function` 对象的实例。从实用性的角度来说，这应该不会影响函数的使用，只要知道有这么回事儿就行了。值得一提的是，如果尝试识别函数变量 v 的类型，`typeof v` 会返回`"function"`，如果 v 是一个函数。这样很合乎情理，对比一下 v 是数组的情况：`typeof v` 会返回`"Object"`。所以我们可以用 `typeof v` 来识别函数。注意，即便 v 是函数，`v instanceof Object` 也会返回 `ture`，所以，如果想要区分函数类型和其他类型的对象，应该优先使用 `typeof`。

13.5 IIEFs 和异步代码

第 6 章已经介绍了 IIEFs（即时调用函数表达式），实际上，它们是一种创建闭包的方式。看一个重要的例子（在第 14 章中会再回头看这个例子），这个例子演示了 IIFEs 如何用异步代码提供帮助。

IIFEs 的一个重要用途是在一个全新的作用域中创建新变量，从而让异步代码正确执行。这里有一个经典的计时器例子：从 5 秒开始倒计时，到 0 秒时输出“go!”。这段代码使用了内建函数 `setTimeout`，该函数会根据第二个参数（毫秒数）设置的时间延迟执行第一个参数（一个函数）。例如，下面的代码会在 1.5 秒后输出 hello。

```
setTimeout(function() { console.log("hello"); }, 1500);
```

有了这些知识储备，就可以编写倒计时代码了：

```
var i;
for(i=5; i>=0; i--) {
   setTimeout(function() {
       console.log(i===0 ? "go!" : i);
   }, (5-i)*1000);
}
```

注意，这里使用了 `var` 而不是 `let`，此时大家必须要理解 IIFEs 为什么重要。如果希望这段代码打印 `5,4,3,2,1,go!`，结果可能会让你失望。实际上，会看到`-1` 被打印了 6 次。为什么会这样呢？因为在这段代码中，传给 `setTimeout` 的函数没有在循环中被调用，它们会在未来的某个时间点被调用。所以循环会正常运行，从 5 开始，到`-1` 结束……而这发生在函数被调用之前。所以，当函数被调用时，`i` 的值已经变成了`-1`。

即使块级别的作用域（使用 `let` 变量）能从本质上解决了这个问题，不过如果你对异步编程还比较陌生，那这个例子还是很重要的。大家可能很难一下子完全明白，但理解异步执行（第 14 章的主题）真的非常重要。

在块作用域变量出现前，要解决这个问题需要借助一个额外函数。使用这个额外的函数创建一个新的作用域，就可以在每一步执行中“捕获”（在闭包中）i 的值。先考虑一个具名的函数：

```
function loopBody(i) {
   setTimeout(function() {
      console.log(i===0 ? "go!" : i);
   }, (5-i)*1000);
```

```
}
var i;
for(i=5; i>0; i--) {
   loopBody(i);
}
```

在循环的每一步，函数 loopBody 都被调用了。回忆一下 JavaScript 中是如何给函数传参的：通过传值。每一步，传入函数的不是变量 i 本身，而是 i 的值。所以第一次传入 5，第二次是 4，以此类推。可以看到，这里使用了同名变量 i，但这没关系，因为实际上，创建了 6 个不同的作用域，以及 6 个独立的变量（一个给外部作用域用，其他 5 个在调用 loopBody 时使用）。

为循环创建一个只会被用到一次的具名函数有点多余，可以使用 IIFEs。本质上，它们创建了一个等价的匿名函数，这个函数会被立即调用。这是使用了 IIFE 的版本：

```
var i;
for(i=5; i>0; i--) {
   (function(i) {
      setTimeout(function() {
         console.log(i===0 ? "go!" : i);
      }, (5-i)*1000);
   })(i);
}
```

括号好多！不过，如果仔细想想，这正是我们想要实现的功能：创建一个只有一个参数的函数，然后在每一次循环中调用该函数（见图 13-1）。

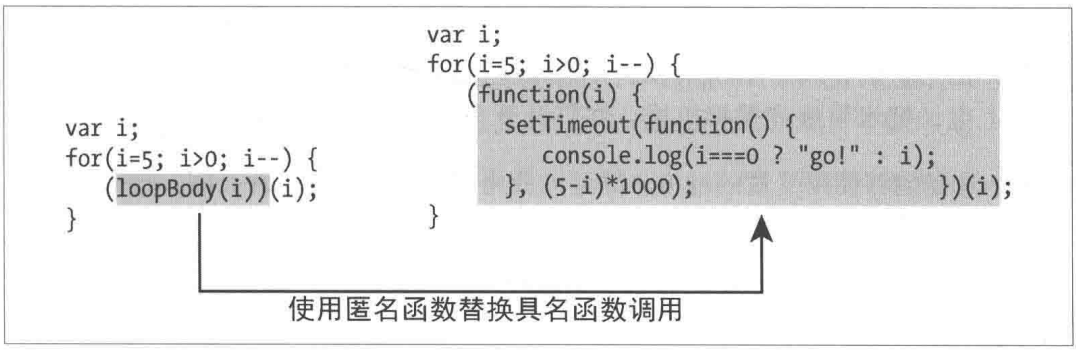


图 13-1　即时调用函数表达式

块作用域变量解决了这个问题，省去了用函数创建新作用域的麻烦。使用块作用域变量可以大大简化这个例子：

```
for(let i=5; i>0; i--) {
   setTimeout(function() {
```

```
        console.log(i===0 ? "go!" : i);
    }, (5-i)*1000);
}
```

注意，for 循环的参数内部使用了 let 关键字。如果把它放到循环外部，之前的问题又会出现。这样使用 let 关键字，是为了告诉 JavaScript，在每一次循环中，为变量 i 生成一个新的、独立的拷贝。所以当 setTimeout 中的函数在未来的某个时刻执行时，它们接收的值都是来自自身作用域中的变量。

13.6 函数变量

如果你是编程初学者，那你可能需要坐下来再给自己倒一杯咖啡，平复一下情绪：因为接下来要讲的内容是初学者经常纠结的地方，也是需要掌握的重要概念之一。

通常，将数值、字符串、甚至数组当作变量很容易理解，这也导致了习惯性地认为变量就是一些数据（或一个集合，就像数组和对象）。然而，如果认为变量只是数据，就很难意识到函数的所有潜能，因为可以像传递变量那样传递函数。由于函数是动态的，一般我们不会认为函数是一种数据（通常，我们认为数据是静态的）。的确，当函数被调用的时候，它是动态的。不过在调用之前，它与其他变量一样，是静态的。

下面这个比喻可能会帮助理解。假如去超市，你可以将水果看成是数据，比如 2 个香蕉，1 个苹果等。此时决定用水果做冰沙，所以需要买一个搅拌机。搅拌机更像一个函数：它能做一些事情（比如，将水果搅成冰沙）。但当搅拌机在购物车里时，没有通电，此时它仅仅是购物车中的一件商品，可以像对待水果一样对待它：把它从购物车中拿出来放到传送带上、付款、放入购物袋中带回家。只有当插上电，把水果放到搅拌机里，并打开开关的时候，它才变得跟水果不一样。

因此，凡是能够使用变量的地方，都可以使用函数，这意味着什么呢？这意味除了变量的普通用法外，还可以做下面这些事：

- 通过创建一个指向函数的变量来给函数起一个别名。
- 将函数放入数组中（可能混合其他类型的数据）。
- 将函数当做对象的属性（见第 9 章）。
- 将函数传入到另一个函数中。
- 从一个函数中返回一个函数。

- 从一个把函数当做参数的函数中返回一个函数。

头晕吗？单是写出来，就难以置信的抽象，大家可能会奇怪“为什么地球上会有人做这种事情？”事实上，它的灵活性强大到难以置信，而人们也在频繁地做这些事。

先从上述列表中最容易理解的一项开始：给函数起别名。设想一下：有一个名字非常长的函数，并且想在短短几行代码中多次使用它。全部敲出来会让人筋疲力尽，而且写出来的代码也非常难看。因为函数也是一种数据类型，所以可以创建一个名字更短的变量来代替它。

```
function addThreeSquareAddFiveTakeSquareRoot(x) {
   // 这是一个非常傻瓜式的函数，不是吗？
   return Math.sqrt(Math.pow(x+3, 2)+5);
}

// 起别名之前
const answer = (addThreeSquareAddFiveTakeSquareRoot(5) +
   addThreeSquareAddFiveTakeSquareRoot(2)) /
   addThreeSquareAddFiveTakeSqureRoot(7);

// 起别名之后
const f = addThreeSquareAddFiveTakeSquareRoot;
const answer = (f(5) + f(2)) / f(7);
```

注意，在“之后”的例子中，作者没有在 addThreeSquareAddFiveTakeSquareRoot 后面使用括号。因为一旦加了括号，就成了函数调用，f 就不是 addThreeSquareAddFiveTakeSquareRoot 的别名，而是函数调用的结果。那么，当使用它（比如，f(5)）时就会报错，这是因为 f 不是一个函数，而只有函数才能被调用。这个例子是故意写成这样的，不过这并不常见。它最开始出现的地方是命名空间，（*namespacing*），这在 Node 开发（见第 20 章）中很常见。例如：

```
const Money = require('math-money'); // require 是一个用来导入类库的 Node 函数

const oneDollar = Money.Dollar(1);
// 或者，如果我们不希望在所有调用的地方都使用 "Money.Dollar"：
const Dollar = Money.Dollar;
const twoDollars = Dollar(2);
// 注意 oneDollar 和 twoDollars 是同一种类型的实例
```

在上例中，并没有使用别名 Dollar 去简化 Money.Dollar，即使这样做也挺合理。

至此已经做了很多锻炼之前的“伸展运动”了，准备活动已经够了，接下来来进行

一些更烧脑的抽象思考吧。

13.6.1 数组中的函数

以往，数组中的函数并不经常被使用，但它的使用率却在升高，并且在一些特定场景中极其有用。管道就是一个这样的场景：它好比要频繁执行的一系列独立操作。使用数组的好处是可以随时修改它。比如，如何移除一个操作？把它从数组中移除就行。增加操作呢？向数组中添加一个元素就可以了。

图形转换就是一个这样的例子。如果开发一个可视化软件，通常会有一个在很多点上都会用到的转换“管道”。这是一个常见的二维转换示例：

```
const sin = Math.sin;
const cos = Math.cos;
const theta = Math.PI/4;
const zoom = 2;
const offset = [1, -3];

const pipeline = [
   function rotate(p) {
      return {
         x: p.x * cos(theta) - p.y * sin(theta),
         y: p.x * sin(theta) + p.y * cos(theta),
      };
   },
   function scale(p) {
      return { x: p.x * zoom, y: p.y * zoom };
   },
   function translate(p) {
      return { x: p.x + offset[0], y: p.y + offset[1]; };
   },
];

//此时 pipeline 是一个包含了特殊 2D 转换的函数数组
//我们可以转换一个点：

const p = { x: 1, y: 1 };
let p2 = p;
for(let i=0; i<pipeline.length; i++) {
   p2 = pipeline[i](p2);
}

// 此时 p2 是 p1 基于开始位置旋转 45 度（pi/4 弧度），然后向前移动 2 个单位，向右 1 一个
// 单位，向下 3 个单位，所得出的点
```

这是一个非常基础的图形转换示例，希望通过它，你能够感受到将函数存储在数组

中的力量。注意，在管道中使用函数的语法：pipeline[i]访问索引为i的元素，该元素是一个函数。那么此时对应的函数会被调用（使用括号）。点被传到函数中，然后又被赋给自己。这样一来，该点的值就是管道中每一步执行的累积结果。

管道处理不仅存在于图形应用中，同样流行于音频处理、科学和工程学应用中。事实上，任何时候当需要按照指定顺序执行一系列函数时，管道都是一个有用的抽象原则。

13.6.2 将函数传给函数

大家已经接触了很多将函数传给函数的例子：比如，把函数传给 setTimeout 或 forEach。这样做的另一个原因是为了管理异步编程，作为异步编程的一个范式，它已经越来越流行了。实现异步执行的常见方法是将一个函数（通常叫*回调函数*，缩写为 *cb*）传给另一个函数。该函数在闭包函数执行完成时（不论它执行了什么操作）被调用（回调）。在第 14 章中，作者会在更大范围内讨论回调函数。

把一个函数传给另一个函数的用法除了用在回调上，它还是一个“注入”功能的好方法。来看一个叫作 sum 的函数，它会统计一个数组中所有数字的总和（为了简化例子，对于数组中的非数字元素不做任何检查和错误处理）。这是一个很简单的练习，但是如果需要一个返回数字平方和的函数呢？当然，可以简单地编写一个名为 sumOfSquares 的函数……如果还需要立方和呢？这正是传递函数能够起作用的地方。下面看看 sum 函数的实现。

```
function sum(arr, f) {
   // 如果没有提供任何函数，使用一个将参数原样返回的"空函数"
   if(typeof f != 'function') f = x => x;

   return arr.reduce((a, x) => a += f(x), 0);
}
sum([1, 2, 3]);                                    // 返回 6
sum([1, 2, 3], x => x*x);                          // 返回 14
sum([1, 2, 3], x => Math.pow(x, 3));               // 返回 36
```

通过给 sum 函数传入一个函数，就可以完成任何想做的事情。需要平方根的和？没问题！需要数字的 4.233 次方的和？简单！注意，如果不想在调用 sum 函数的时候做任何特别的事情，那就不传入函数。此时，在函数内部，参数 f 的值是 undefined，这时如果调用它就会出错。为了防止这种错误，把非函数的参数转换成一个“空函数”，其实就是什么都不做。也就是说，如果传入 5，它就返回 5，以此类推。也有一些更高效的方式可以处理这种情况（例如，调用另外的函数，这

样就节省了在每个元素上调用空函数的开销），但像这样创建“安全”函数是一个不错的实践。

13.6.3 在函数中返回函数

在一个函数中返回另一个函数恐怕是函数所有用法中最难懂的一个了，但它非常有用，可以把从函数中返回函数看成 3D 打印机：它自己制造一些东西（像函数一样），然后这些东西还可以继续制造东西。激动人心的是，可以定制返回函数：类似于可以定制 3D 打印机打印出的东西。

回想一下之前的 sum 函数，它有一个可选的函数参数，该函数会在求和之前对数组中每一个元素做一些操作。还记得之前提到过，如果有需要，可以创建一个名为 sumOfSquares 的独立函数。假设现在就需要这样一个函数。但是，一个函数的参数中又有数组，又有函数是不太好的：需要的显然是一个只接收一个数组作为参数的函数，而返回的是平方和。（如果想知道什么时候这种假设会成立，考虑这样一个 API，它允许提供一个 sum 函数，但它只接收具有单个参数的函数）

其中一种实现方式是创建一个新函数，让该函数调用原来的函数。

```
function sumOfSquares(arr) {
   return sum(arr, x => x*x);
}
```

需要的只是一个函数时，这种方式固然不错。但如果需要不断重复这种模式呢？一个有效的解决方案可能是，创建一个返回特定函数的函数：

```
function newSummer(f) {
   return arr => sum(arr, f);
}
```

新的函数 newSummer 创建了一个只接收一个自定义函数作为唯一参数的求和函数。下面看看如何使用它进行不同种类的求和运算：

```
const sumOfSquares = newSummer(x => x*x);
const sumOfCubes = newSummer(x => Math.pow(x, 3));
sumOfSquares([1, 2, 3]);    // 返回 14
sumOfCubes([1, 2, 3]);      // 返回 36
```

把接收多个参数的函数转换成接收单个参数的函数的技术叫作柯里化（*currying*），该名字出自于它的开发者：美国数学家 Haskell Curry。

在函数中返回函数的应用程序通常都很深奥难懂。如果想看更多的例子，可以看看Express 或者 Koa（流行的 JavaScript web 开发框架）的中间件。一般来说，中间件就是一个返回函数的函数。

13.7 递归

另一个常见并且重要的函数用法是递归，递归是指那些调用自身的函数。当递归函数的输入集合不断缩小的时候，这项技术就会变得异常强大。

接下来看一个不太现实的例子：干草寻针。如果有一堆干草，要从中找一枚针，大家可能会这样做：

(1) 如果找到针了，跳到第 3 步。

(2) 从草堆中拿走一根干草，回到第 1 步。

(3) 完成。

它的基本原理是不断地削减草堆，直到找到针为止。而这本质上是递归。下面看看如何使用代码实现这个例子：

```
function findNeedle(haystack) {
   if(haystack.length === 0) return "no haystack here!";
   if(haystack.shift() === 'needle') return "found it!"
   return findNeedle(haystack);    // haystack 又少了一个元素
}

findNeedle(['hay', 'hay', 'hay', 'hay', 'needle', 'hay', 'hay']);
```

这个递归函数中需要注意的关键点是，它处理了所有的可能性：haystack 是空的（这样就没有东西可找了），第一个元素就是针（找到了!），或者第一个元素不是针（那么针就在数组中的其他地方，所以移除了第一个元素，然后重复该函数：记住 Array.prototype.shift 移除的是数组第一个元素）。

注意：递归函数一定要有一个结束条件，不然它会一直递归下去，直到 JavaScript 解释器认为调用栈太深了（会导致程序崩溃）。在 findNeedle 函数中，有两个结束条件：找到了，或者干草堆没有干草了。因为每次递归调用都会减少草堆的体积，所以最终达到结束条件是不可避免的。

再来看一个历史悠久的实用例子：计算一个数的阶乘。某个数的阶乘等于这个数和小于它的所有正整数相乘，阶乘的表示方式是在数字后加一个感叹号（!）。所以 4!

指的是 4×3×2×1 =24。下面是使用递归函数的实现方式：

```
function fact(n) {
   if(n === 1) return 1;
   return n * fact(n-1);
}
```

代码里面有一个结束条件（n === 1），并且每次递归调用的时候，n 的值都会减 1。所以 n 最终会达到 1（如果传入的是 0 或者负数，该函数将会无限执行下去直至崩溃，当然可以添加一些错误条件来防止这种事情的发生）。

13.8 小结

如果你有其他函数式编程语言的经验，像 ML、Haskell、Clojure 或者 F#，那么本章内容对你来说也许轻而易举。如果没有，它或许能扩展你的思维，但是你可能对函数式编程的抽象可能性还有点模糊（当作者第一次接触这些概念的时候，也是这样的）。面对完成同一件事情的不同方式，大家或许有点不知所措，并且会好奇哪种方式“更好”。恐怕这个问题的答案没有那么简单。通常，它取决于所面临的问题：因为有些问题强烈建议使用某种特定的技术。但更多的取决于自己：什么技术能产生共鸣？如果大家发现本章提到的技术很难懂，本人建议重新读几遍。因为那些概念的作用极其强大，如果不花些时间了解它们，你可能无法判断这些技术是否真的对你有用。

第 14 章

异步编程

在第 1 章响应用户交互的时候我们第一次见到了异步编程。回忆一下用户交互，它本质上是一个异步：因为无法控制用户的点击、触摸、说话或者敲击等事件。不过，用户输入并不是需要执行异步编程的唯一原因：在 JavaScript 中，很多东西本质上都是由异步编程实现的。

当一个 JavaScript 应用在运行的时候，它会以单线程的方式运行。也就是说，JavaScript 在同一个时刻只做一件事。大部分现代计算机都可以同时处理多件事情（假设它们是多核的），即使是单核计算机，也因为具有极高的运算速度从而可以模拟同时处理多件事，它会先处理一部分 A 任务，然后处理一部分 B 任务，再处理一部分 C 任务，如此循环直到所有任务都处理完（也叫抢占式多任务处理）。对于用户来说，任务 A、B、C 是同时运行的，不管它们是否真的在多个内核中同步进行。

JavaScript 的单线程性质可能会让开发人员觉得被限制了，但实际上它避免了开发人员为多线程编程中可能出现的棘手问题而担心。不过这种自由是有条件的：为了写出能够流畅运行的软件，必须考虑异步，而不仅仅是用户输入。这种思考方式刚开始可能会有些困难，尤其是在之前使用的编程语言都是同步执行的情况下。

JavaScript 这门编程语言从早期开始就有一套自己的异步执行方式。不过，随着 JavaScript 的普及，它所能完成的软件也越来越复杂，一些管理异步编程的新理念也被吸收进这门语言中。就事实而言，JavaScript 对异步编程的支持有三个不同的阶段：回调（callback）阶段、promise（承诺）阶段和生成器（generator）阶段。如果只是简单地说生成器比任何出现在它之前的阶段都好，那么只了释生成器的工

作原理，而跳过其他的就好。但实际上并不是这么简单。生成器本身并不提供任何对异步的支持：它们依赖于承诺或特定类型的回调来提供异步行为。同样，像承诺这样有用的东西，会依赖于回调（而回调本身又由于具有对象而变得更有用）。

除了用户输入以外，异步编程技术的三个主要使用场景是：

- 网络请求（如 Ajax 请求）。
- 文件系统操作（读/写文件等）。
- 刻意的时间延迟功能（比如警告）。

14.1 类比

在解释回调和承诺的时候作者很喜欢用一个比喻，就是在一个人满为患且没有预定的餐厅里找一张空桌子。此时不需要排队等位，当有位子的时候餐厅会给打电话。这就类似于回调：给餐厅工作人员提供了一些信息，允许他们在有位子的时候通知客户。这样一来，餐厅可以忙自己的事情，而客户也可以做别的事情；没有人在等其他人。另一家餐厅也许会给客户一个传呼机，在位子准备好后它就会响。这更像是一个承诺：餐厅工作人员给客户一个承诺，承诺在有空桌子的时候通知客户。在继续讲解回调和承诺时，时刻谨记这个比喻，尤其是当读者是异步编程初学者的时候。

14.2 回调

回调是 JavaScript 中最古老的异步方式，其实我们已经在处理用户输入和超时中见过它的用法了。简单来说回调就是写一个函数，然后在未来的某个时刻调用它。这个函数本身并没有什么特别之处：它只是一个一般的 JavaScript 的函数。通常，会把这些回调函数提供给别的函数，或者将它们设为对象属性（又或者，在数组中使用它们，这种情况极少）。一般来说（并非总是如此），回调都是匿名函数。

从一个简单的例子开始，使用 `setTimeout`，这是 JavaScript 的内建函数，可以将函数的执行推迟指定的毫秒数：

```
console.log("Before timeout: " + new Date());
function f() {
   console.log("After timeout: " + new Date());

}
```

```
setTimeout(f, 60*1000); // 1分钟
console.log("I happen after setTimeout!");
console.log("Me too!");
```

在控制台运行这些代码，如果打字速度不是特别慢，大家会看到类似下面的输出：

```
Before timeout: Sun Aug 02 2015 17:11:32 GMT-0700 (Pacific Daylight Time)
I happen after setTimeout!
Me too!
After timeout: Sun Aug 02 2015 17:12:32 GMT-0700 (Pacific Daylight Time)
```

这里，代码编写的顺序与实际执行的顺序之间没有必然联系，这通常是初学者纠结的地方。有些人会觉得，甚至期望，计算机执行代码的顺序跟我们编写代码的顺序是一样的。也就是说，希望看到如下输出：

```
Before timeout: Sun Aug 02 2015 17:11:32 GMT-0700 (Pacific Daylight Time)
After timeout: Sun Aug 02 2015 17:12:32 GMT-0700 (Pacific Daylight Time)
I happen after setTimeout!
Me too!
```

这可能才是大家想看到的结果，但它却不是异步。异步执行的主旨在于，它不会阻塞任何事。由于 JavaScript 是单线程的，如果告诉程序在执行某些代码之前等待 60 秒，而此时立刻执行这段代码的话，就什么都不会发生。程序将会暂停：它不会接收用户数据，更不会做刷新屏幕等事情。大家可能都经历过类似的事情，那绝不是令人满意的结果。异步技术可以帮助避免这种“锁定”发生。

在本例中，为了解释得更清楚，作者给 `setTimeout` 传入一个有名字的函数。一般情况下，除非因为编译问题从而不得不给函数命名，通常都会用匿名函数：

```
setTimeout(function() {
   console.log("After timeout: " + new Date());
}, 60*1000);
```

`setTimeout` 在使用上有一个小问题：因为数字类型的超时参数在最后，当它跟特别长的匿名函数一起作为参数时，这个参数很容易漏掉，或者被看做是匿名函数的一部分。这很容易出错，不过，要习惯于将 `setTimeout` 和匿名函数一起使用。只要记住最后一行需要包含一个超时参数即可。

14.2.1 setInterval 和 clearInterval

`setTimeout` 在运行过一次函数后就不再运行了。在此基础上，还有 `setInterval` 函数，它会每隔一段特定的时间运行回调函数，并且一直运行下去，直到调用 `clearInterval`。这里有一个例子，每隔 5 秒钟运行一次函数直到一分钟结束，或者运行 10 次，以先到的为准：

```
const start = new Date();
let i=0;
const intervalId = setInterval(function() {
   let now = new Date();
   if(now.getMinutes() !== start.getMinutes() || ++i>10)
      return clearInterval(intervalId);
   console.log('${i}: ${now}');
}, 5*1000);
```

可以看到 setInterval 返回了一个 ID，在后面可以用它来取消（停止）这次调用。与之对应的 clearInterval 在 timeout 之前停止本次调用也正是使用了这种方式。

setTimeout、setInterval 和 clearInterval 都定义在全局对象中（在浏览器中是 window，在 Node 中是 global）

14.2.2 scope 和异步执行

异步执行中容易让人疑惑或犯错的一点是：scope 和闭包是如何影响异步执行的。每当一个函数被调用时，都创建了一个闭包（closure）：所有在函数内部创建的变量（包括形参）只在有被访问的时候才存在。

之前见过这个例子，但是由于其中有一些重要的内容需要学习，所以这里再次强调。考虑一个叫作 countdown 的函数的例子，其目的是创造一个 5 秒的倒计时。

```
function countdown() {
   let i;                          // 注意这里的 let 是定义在 for 循环之外的
   console.log("Countdown:");
   for(i=5; i>=0; i--) {
      setTimeout(function() {
         console.log(i===0 ? "GO!" : i);
      }, (5-i)*1000);
   }
}
countdown();
```

先在大脑中过一下这个例子。大家可能会记得上一次看到这个例子的时候遇到的错误。该例子看起来是期望从 5 开始倒计时。但实际上得到的却是 6 次-1，没有一个"GO! "。我们第一次看到它的时候，使用的是 var；这次则用了 let，但由于 let 的定义在 for 循环之外，所以会有同样的问题：for 循环执行完毕时，i 的值已经为-1，而之后 callback 函数才开始执行。这里的问题在于，当函数在执

行时，i 的值已经被赋为-1 了。

这里我们需要重要了解的内容是：理解 scope 和异步执行是如何关联的。当调用 countdown 时，创建了一个包含变量 i 的闭包。所有在 for 循环中创建的（匿名）回调函数都可以访问 i，并且是同一个 i。

本例中有个小细节，就是在 for 循环中，i 有两种不同的使用方式。当用它来计算超时（(5-i)*1000）时，它可以正常工作：第一个超时是 0、第二个是 1000、第三个是 2000，以此类推。之所以这样是因为计算是同步的。实际上，对 setTimeout 的调用也是同步的（这就需要先进行计算，这样 setTimeout 才知道什么时候调用回调函数）。真正的异步调用是传到 setTimeout 的中的函数，而这才是问题的关键。

还记得之前通过即时调用函数表达式（IIFE）来解决这个问题吗？或者可以用更简单的方式，直接把 i 的定义挪到 for 循环中。

```
function countdown() {
   console.log("Countdown:");
   for(let i=5; i>=0; i--) {      // i 在块语句的 scope 中
      setTimeout(function() {
         console.log(i===0 ? "GO!" : i);
      }, (5-i)*1000);
   }
}
countdown();
```

本小节学到的是，必须时刻注意定义回调函数的作用域：回调函数可以访问闭包内的所有内容。正是由于这个原因，回调函数的实际执行结果可能会跟预期的不一样。这个原则适用于所有的异步技术，而不仅仅是回调。

14.2.3 错误优先回调

在 Node 确定其主导地位后，一个叫作*错误优先回调*的约定产生了。由于回调使异常处理变得很棘手（你马上就会看到），所以需要一个标准化的方式将错误传到回调中。于是就出现了在回调中使用第一个参数来接收错误对象的约定。如果该对象为 null 或者 undefined，就表示没有错误。

任何时候当处理一个错误优先的回调时，都应该先检查错误参数并且采取合适的措施。比如，在 Node 中读取一个文件的内容，就坚持采用了错误优先的约定：

```
const fs = require('fs');

const fname = 'may_or_may_not_exist.txt';
```

```
fs.readFile(fname, function(err, data) {
   if(err) return console.error('error reading file ${fname}: ${err.message}');
   console.log('${fname} contents: ${data}');
});
```

在回调函数中，要做的第一件事就是判断 err 是否为真。如果为真，就表示在读文件时出错了，程序会将错误报告打印到 console 中并立即返回（console.error 并不是一个有意义的值，同时由于并不会使用这个返回值，所以就把它们合并为一句）。这可能是使用错误优先的回调中最容易被忽视的错误了：开发人员通常都会记得检查错误，还可能把错误记在 log 中，但是却忘了立即返回。如果函数可以继续执行，它可能要依赖于成功的回调，但实际上回调已经失败了（当然，也有可能回调并不依赖成功的返回，这种情况下出错并且继续运行尚可接受）。

错误优先的回调是 Node 开发中已经存在的标准（那时承诺还没有被广泛使用），如果在写一个使用回调的接口，强烈建议你坚持遵守错误优先的约定。

14.2.4 回调地狱

回调可以帮助管理异步执行，但它却有一个实际的缺陷：当需要在执行过程中等待多个事件的时候，使用回调来管理就有点捉襟见肘了。还记得编写的第一个 Node 应用吗？读取三个不同文件中的内容，然后等待 60 秒，再把这些内容合并并写进第 4 个文件：

```
const fs = require('fs');

fs.readFile('a.txt', function(err, dataA) {
   if(err) console.error(err);
   fs.readFile('b.txt', function(err, dataB) {
      if(err) console.error(err);
      fs.readFile('c.txt', function(err, dataC) {
         if(err) console.error(err);
         setTimeout(function() {
            fs.writeFile('d.txt', dataA+dataB+dataC, function(err) {
               if(err) console.error(err);
            });
         }, 60*1000);
      });
   });
});
```

这就是被开发人员称为“回调地狱”的场景，这段用花括号堆起来的三角形的代码简直要逆天了，它比错误处理中出现的问题还要糟糕。在本例中，只是将错误打在了 log 里，但如果试着抛出异常，可能会被结果吓到。来看一个稍微简单点的例子：

```
const fs = require('fs');
```

```
function readSketchyFile() {
   try {
      fs.readFile('does_not_exist.txt', function(err, data) {
         if(err) throw err;
      });
   } catch(err) {
      console.log('warning: minor issue occurred, program continuing');
   }
}
readSketchyFile();
```

一眼看上去，这个例子似乎很合理，同时我们也许还会为自己是一个使用异常处理的防御型开发人员而沾沾自喜。除此之处它并不能工作。如果运行这个例子你就会发现：它会导致程序崩溃。即使已经采取了一些措施来确保这些非预期的错误不会引发错误。出现这个问题的原因是 try…catch 块只在同一个函数的作用域内才有效。在本例中，try…catch 块在 readSkechyFile 函数中，但是错误却是在作为回调函数被调用的匿名函数 fs.readFile 中抛出的。

总之，没办法阻止回调函数被意外地调用两次，或者压根没有被调用。如果寄希望于它被调用且只被调用一次，那么结果只会令人失望，因为 JavaScript 中并没有提供防止这种意外出现的保护机制。

虽然说世上没有克服不了的困难，但随着异步代码的流行，编写低错误率且可维护的代码变得非常困难，此时 promise 就粉墨登场了。

14.3 promise

promise 的出现是为了弥补回调中的一些不足。使用 promise 可以编写更加安全，且易于维护的代码（关于这一点尚有争议）。

promise 并不排斥回调。实际上，依然需要将回调和 promise 结合在一起使用。promise 所做的就是确保回调始终被以可预期的方式处理，从而避免了一些在只使用 callback 的情况发生的不愉快的意外和难以定位的 bug。

promise 的初衷很简单：在调用基于 promise 的异步函数时，它会返回一个 promise 实例。其中只可能发生两件事情：被满足（success）或被拒绝（failure）。promise 可以保证只有一件事会发生（不可能既满足又拒绝），而结果只会发生一次（如果满足，那么只会被满足一次；如果拒绝，也只会被拒绝一次）。一旦 promise 被满足或者被拒绝了，就会认为它被处理（*settled*）了。

promise 相比于回调的另一个方便的好处在于，因为 promise 只是对象，所以它们

可以被到处传递。如果想开始一段异步进程，但却希望结果在其他地方被处理，可以直接把 promise 传给它们（这就类似于把用于订餐的传呼机给自己的朋友，只要需求一样，餐厅显然不会在意谁在预约）。

14.3.1 创建 promise

创建 promise 的方式很直接：创建一个带有函数的 promise 实例，它应该包含一个 resolve（满足）和 reject 的回调（看吧，前面曾提醒过，就算使用了 promise，还是需要回调函数）。继续看 coundown 函数，给它指定参数（这样就能设置 5 秒以上的倒计时了），当倒计时结束时返回一个 promise：

```
function countdown(seconds) {
   return new Promise(function(resolve, reject) {
      for(let i=seconds; i>=0; i--) {
         setTimeout(function() {
            if(i>0) console.log(i + '...');
            else resolve(console.log("GO!"));
         }, (seconds-i)*1000);
      }
   });
}
```

这个函数现在还不太灵活。大家并不想使用相同的方式，或者压根不想用控制台。如果想用 countdown 来更新网页中的 DOM 元素，效果可能就不太理想了。但这只是开始，它演示了如何创建一个 promise。要注意 `resolve`（就像 `reject`）是一个函数。大家可能会想“哈！可以多次调用 resolve，从而跳出……呃，promise 的 promise。”的确可以多次调用 `resolve` 或者 `reject`，甚至一起调用…但是只有第一次调用时会起作用。promise 可以保证的是，不管谁使用了 promise，都只会得到一个满足或者拒绝的响应（目前，所讲的函数中还未涉及拒绝的路径）。

14.3.2 使用 promise

下面看看如何使用 `countdown` 函数。可以直接调用它并忽略 promise 部分：比如，直接调用 `countdown(5)`。这里还是会有倒数的功能，并且不会被 promise 搞晕。但如果想用 promise 的特性呢？这个例子中展示了如何使用 promise 的返回值：

```
countdown(5).then(                                    \
   function() {
      console.log("countdown completed successfully");
   },
   function(err) {
```

```
        console.log("countdown experienced an error: " + err.message);
    }
);
```

在本例中，不用把返回的 promise 赋给一个变量，而是直接调用它的 then 处理器。这个处理器有两个回调：第一个是满足的回调，第二个是错误的回调。其中最多只会有一个函数被调用。promise 也支持 catch 处理器，这样就可以把两个处理器分开了（在演示时，会把 promise 存在一个变量中）：

```
const p = countdown(5);
p.then(function() {
      console.log("countdown completed successfully");
});
p.catch(function(err) {
      console.log("countdown experienced an error: " + err.message);
});
```

试着修改 countdown 函数，给它添加一个错误的情况。想象一下：如果我们很迷信，在数到数字 13 的时候就返回一个错误。

```
function countdown(seconds) {
   return new Promise(function(resolve, reject) {
      for(let i=seconds; i>=0; i--) {
         setTimeout(function() {
            if(i===13) return reject(new Error("DEFINITELY NOT COUNTING THAT"));
            if(i>0) console.log(i + '...');
            else resolve(console.log("GO!"));
         }, (seconds-i)*1000);
      }
   });
}
```

不妨动手试试这个例子。大家会看到一些有意思的行为。很显然，可以从小于 13 的任何数字开始倒数，这样就不会出错了。从 13 或是大于 13 的数字开始，则会在数到 13 的时候出错。但是，控制台会一直打 log。调用 reject（或者 resolve）并没能终止函数，它们只是修改了 promise 的状态。

显然 countdown 函数需要优化。通常，并不希望一个函数在被处理后还继续运行（不管是成功还是失败），但是 countdown 却继续运行了。之前早已提到过控制台中的 log 一点也不灵活，它们并不会真的提供想要的控制权。

promise 提供了一个定义极其良好，并且可以安全地处理那些满足或者拒绝的异步任务的方式，但是它却没有（就目前而言）报告过程进度的能力。也就是说，promise 只可能是满足或拒绝，绝不会出现“50%完成”。有的 promise 库中增加了一些很有用的功能，比如，可以报告过程，而且在未来，很可能 JavaScript 中的 promise 也

会具备那些功能，不过现在，我们只能在没有这些功能的情况下工作。如果想要这些功能，需要继续学习下面的内容。

14.3.3 事件

事件是 JavaScript 中另一个备受瞩目且历久弥新的概念。它的概念很简单：事件发射器可以广播事件，任何愿意监听（或者“订阅”）这些事件的人都可以去做这件事。如何订阅事件呢？当然非回调莫属了。创建自己的事件系统其实很简单，即便如此，Node 还是为我们提供了内建的支持。如果使用浏览器，jQuery 同样提供了一个事件机制（*http://api.jquery.com/category/events/*）。为了改进 countdown，我们通常会用 Node 的 `EventEmitter`。虽然也可以在像 `countdown` 这样的函数中使用 `EventEmitter`，不过实际上它的设计初衷是跟类一起使用。所以可以把 `countdown` 函数放在 `Countdown` 类中：

```
const EventEmitter = require('events').EventEmitter;

class Countdown extends EventEmitter {
   constructor(seconds, superstitious) {
      super();
      this.seconds = seconds;
      this.superstitious = !!superstitious;
   }
   go() {
      const countdown = this;
      return new Promise(function(resolve, reject) {
         for(let i=countdown.seconds; i>=0; i--) {
            setTimeout(function() {
               if(countdown.superstitious && i===13)
                 return reject(new Error("DEFINITELY NOT COUNTING THAT"));
               countdown.emit('tick', i);
               if(i===0) resolve();
            }, (countdown.seconds-i)*1000);
         }
      });
   }
}
```

`Countdown` 类继承了 `EventEmitter`，这样 Countdown 就可以“发射”事件。`Go` 方法是正式开始倒计时并返回 promise 的地方。注意，在 go 函数中，我们所做的第一件事就是把 `this` 赋给 countdown。这是因为在回调中，不论倒计时是否迷信数字，都需要 this 的值来获取倒计时的长度。（译者注：在基督教文化中，13 是个不详的数字。作者在这里实际上开了个玩笑：如果倒计时是迷信的 (countdown.superstitious === true)，那么它会在数到 13 时报错。）这里要记住，this

是一个特殊变量，它与回调中的 this 不是同一个东西。所以我们需要保存当前的 this 值，从而在 promise 中使用它。

神奇的事情就发生在调用 `countdown.emit('tick', i)` 的时候。任何想要监听 `tick` 事件（可以任意给它命名，“tick”听起来跟其他事件一样好）的人都可以监听它。接下来看看如何使用这经过改进后的全新 countdown：

```
const c = new Countdown(5);

c.on('tick', function(i) {
   if(i>0) console.log(i + '...');
});

c.go()
   .then(function() {
      console.log('GO!');
   })
   .catch(function(err) {
      console.error(err.message);
   })
```

`EventEmitter` 中的 on 方法允许监听事件。本例为每个 `tick` 事件提供了一个回调。如果 `tick` 的值不是 0，就打印它。接下来调用 go，从而开始倒计时。当倒计时结束时，在 log 中打印 `GO!`。当然这里也可以把 `GO!` 放在 `tick` 事件的监听器中，这么做主要是为了强调事件和 promise 之间的区别。

使用这种方式比原始的 `countdown` 函数定义确实麻烦不少，但是获得了更多功能。现在就可以随心所欲地记录倒计时中的每一秒，同时当倒计时完成时也会有一个被满足的 promise。

还有一件事没做：还没有解决一个迷信倒计时的实例，即使它已经拒绝了 promise，却还是会跳过 13 继续倒计时的问题。

```
const c = new Countdown(15, true)
   .on('tick', function(i) {                    //注意这里可以链式调用'on'
      if(i>0) console.log(i + '...');
   });

c.go()
   .then(function() {
      console.log('GO!');
   })
   .catch(function(err) {
      console.error(err.message);
   })
```

还是能获取倒数到 0 的所有标记（即使不打印出来）。解决这个问题有些复杂，因为已经创建好所有的超时了（当然，可以“作弊”，当创建的迷信计时器的时长大于或等于 13 秒时，就立刻失败，但这就达不到该例子的效果了）。为了解决这个问题，一旦发现不能继续倒计时，就必须清理所有未处理的超时：

```
const EventEmitter = require('events').EventEmitter;

class Countdown extends EventEmitter {
   constructor(seconds, superstitious) {
      super();
      this.seconds = seconds;
      this.superstitious = !!superstitious;
   }
   go() {
      const countdown = this;
      const timeoutIds = [];
      return new Promise(function(resolve, reject) {
         for(let i=countdown.seconds; i>=0; i--) {
            timeoutIds.push(setTimeout(function() {
               if(countdown.superstitious && i===13) {
                  // 清除所有 pending 的 timeouts
                  timeoutIds.forEach(clearTimeout);
                  return reject(new Error("DEFINITELY NOT COUNTING THAT"));
               }
               countdown.emit('tick', i);
               if(i===0) resolve();
            }, (countdown.seconds-i)*1000));
         }
      });
   }
}
```

14.3.4 promise 链

promise 的一个优点就是它可以被链式调用；也就是说，当一个 promise 被满足时，可以立即用它调用另一个返回 promise 的函数，以此类推。创建一个叫作 `launch` 的函数，来链式调用 countdown：

```
function launch() {
   return new Promise(function(resolve, reject) {
      console.log("Lift off!");
      setTimeout(function() {
         resolve("In orbit!");
      }, 2*1000);    // a very fast rocket indeed
   });
}
```

此时链式调用 countdown 就很容易了：

```
const c = new Countdown(5)
   .on('tick', i => console.log(i + '...'));

c.go()
   .then(launch)
   .then(function(msg) {
      console.log(msg);
   })
   .catch(function(err) {
      console.error("Houston, we have a problem....");
   })
```

promise 链的一个好处是不用在每一步都捕获错误；如果错误可能发生在链中的任何一环，promise 链都会停止，继而调到 catch 中。试试把 countdown 的迷信时间改成 15 秒，这时会发现 launch 函数永远都不会被调用。

14.3.5 避免不被处理的 promise

promise 可以简化异步代码，同时确保回调函数不会被多次调用，但却不能避免那些因为 promise 没有被处理而产生的问题（没被处理指在这个 promise 中，既没有调用 `resolve`，也没有调用 `reject`）。这种错误很难被发现，因为它并没有显式的错误。在一个复杂的系统中，一个没有被处理的 promise 很容易丢失。

有一种方式可以避免这种错误，就是给 promise 一个特定的超时。如果 promise 没有在一段合理的时间内被处理，就会自动被拒绝。那么，什么是“一段合理的时间”呢，这需要视情况而定了。比如，有一个很复杂的算法需要执行 10 分钟，那就不要给它设置 1 秒的超时。

往 launch 函数中插入一个人工失败。假设火箭还处在实验阶段，而且有将近一半的时间都是失败的：

```
function launch() {
   return new Promise(function(resolve, reject) {
      if(Math.random() < 0.5) return;     // rocket failure
      console.log("Lift off!");
      setTimeout(function() {
         resolve("In orbit!");
      }, 2*1000);     //火箭的确很快
   });
}
```

在这个例子中，失败的方式不是很合理：因为没有调用 `reject`，甚至没有在控制台打出任何 log。只是悄悄地运行到一半时间的时候失败了。如果多运行几次

这段代码，会发现它只是偶尔起作用，并且没有任何错误消息。这显然不是一个好的处理方式。

可以写一个函数来给 promise 添加一个超时：

```
function addTimeout(fn, timeout) {
   if(timeout === undefined) timeout = 1000; // 默认超时
   return function(...args) {
        return new Promise(function(resolve, reject) {
          const tid = setTimeout(reject, timeout,
             new Error("promise timed out"));
          fn(...args)
             .then(function(...args){
                clearTimeout(tid);
                resolve(...args);
             })
             .catch(function(...args) {
                clearTimeout(tid);
                reject(...args);
             });
        ]};
    }
}
```

如果想说“哇哦……一个返回另一个函数的函数，被返回的函数又返回了一个 promise，而这个 promise 又调用了返回另一个 promise 的函数……我要晕掉了！”这并不奇怪：因为给一个返回 promise 的函数添加超时并不是一件无关紧要的事情，它需要前面那些费解的部分。完全理解这个函数就留给那些想要更加深入学习的读者吧。不过这个函数的使用却很简单：可以给所有需要返回 promise 的函数添加超时。假如最慢的火箭需要 10 秒才能抵达同步轨道（怎么样，未来的火箭技术很厉害吧？），所以将超时设为 11 秒：

```
c.go()
   .then(addTimeout(launch, 4*1000))
   .then(function(msg) {
      console.log(msg);
   })
   .catch(function(err) {
      console.error("Houston, we have a problem: " + err.message);
   });
```

这样的话 promise 链一定会被处理，即使 `launch` 函数的运行情况很糟糕。

14.4 生成器

已经在第 12 章中讨论过，生成器允许函数和其调用方之间的双向通信。它是一个天生的同步，但如果与 promise 结合起来，它们就能为管理 JavaScript 中的异步代码提供强大的技术支持。

来重新看看异步代码中最主要的难点：异步代码比同步代码更难编写。当试图解决一个问题时，通常总是希望以同步的方式来解决它：第一步怎么做，然后是第二步，第三步，以此类推。但是这样解决问题可能会产生性能问题，这就是为什么会有异步。如果能够既享受异步代码带来的性能优化，又避免额外接触那些复杂的概念，岂不是两全其美。这也正是生成器的作用。

回想一下之前在“回调地狱”中讲到的例子：读取三个文件的内容，等待一分钟后，将从前三个文件中读到的内容写入第四个文件。以正常的思维大家可能会写出伪代码：

```
dataA = read contents of 'a.txt'
dataB = read contents of 'b.txt'
dataC = read contents of 'c.txt'
wait 60 seconds
write dataA + dataB + dataC to 'd.txt'
```

生成器可以让大家写出跟上面这些代码非常类似的代码，但是却不会直接实现想要的功能，需要做一些事情才行。要做的第一件事就是找到一个能将 Node 中错误优先的回调转化成 promise 的办法。会把它封装成一个叫作 nfcall 的函数（Node 函数调用）：

```
function nfcall(f, ...args) {
   return new Promise(function(resolve, reject) {
      f.call(null, ...args, function(err, ...args) {
         if(err) return reject(err);
         resolve(args.length<2 ? args[0] : args);
      });
   });
}
```

这个函数是以 Q promise 库中的 nfcall 函数命名的。如果需要这个功能，那么应该使用 Q 这个库。它不仅包含了这个方法，同时还有很多有用的跟 promise 相关的方法。在这里展示 nfcall 的实现，是为了说明其实这并不难。

接下来可以将任何 Node 格式的方法转化成接收一个回调的 promise。同时也需要

setTimeout，它可以接收一个回调……但由于它的出现早于 Node，因而并不适用于错误优先的惯例。所以这里会创建 ptimeout（promise 超时）：

```
function ptimeout(delay) {
   return new Promise(function(resolve, reject) {
      setTimeout(resolve, delay);
   });
}
```

下一步就是需要一个生成器运行器。回想一下生成器并不是天生异步的。但是由于生成器允许函数和其调用方对话，所以可以创建一个用来管理对话的函数，同时这个函数需要知道如何处理异步调用。这里会创建一个叫作 grun（generator run）的函数：

```
function grun(g) {
   const it = g();
   (function iterate(val) {
      const x = it.next(val);
      if(!x.done) {
         if(x.value instanceof Promise) {
            x.value.then(iterate).catch(err => it.throw(err));
         } else {
            setTimeout(iterate, 0, x.value);
         }
      }
   })();
}
```

grun 严重依赖于 runGenerator，它在 Kyle Simpson 写的一系列关于生成器的优秀文章中出现过。强烈建议大家读一下这些文章，来作为内容的补充。

这是一个非常现代化的递归的生成器运行器。当传入一个生成器函数时，它会返回这个函数。就像在第 6 章学到的，生成器会在调用 yield 的时候暂停运行，直到在它们的迭代器上调用 next 函数。这是一个递归函数。如果迭代器返回一个 promise，它会在 promise 被满足后才继续运行。另一方面，如果迭代器返回一个简单值，那么函数就会立即继续执行迭代部分。大家可能会奇怪为什么调用 setTimeout，而不是直接调用 iterate。原因就在于，通过避免同步的递归调用我们能够获得一些性能上的优化（异步递归允许 JavaScript 更快速地将可用资源投入使用）。

读者可能会觉得“这太小题大做了!”或者“这可以让编程更简单吗？”，不要担

心，复杂的部分到这里就结束了。nfcall 允许将过去的方式（Node 中的错误优先回调函数）用在现在（promise）的代码中，而 grun 允许使用未来（在 ES7 会出现一个叫作 await 的关键字，其作用与函数 grun 一样，同时语法也更自然）的技术。终于讲解完复杂的部分了，现在来看看这些东西如何让开发人员更轻松地写代码。

还记得本章前面出现的“可以变得更好吗”的伪代码吗？现在大家可以编写实现了：

```
function* theFutureIsNow() {
   const dataA = yield nfcall(fs.readFile, 'a.txt');
   const dataB = yield nfcall(fs.readFile, 'b.txt');
   const dataC = yield nfcall(fs.readFile, 'c.txt');
   yield ptimeout(60*1000);
   yield nfcall(fs.writeFile, 'd.txt', dataA+dataB+dataC);
}
```

这看起来比回调地狱好很多，不是吗？同时也比只使用 promise 要整洁一些。它改变了原有的思维方式。运行起来也很简单：

```
grun(theFutureIsNow);
```

14.4.1 向前一步和退后两步

大家可能（相当合理）会觉得在理解异步执行时遇到了很多麻烦，来简化一下这些麻烦：除了了解了生成器所带来的额外复杂度和把一般内容转化成 promise 和 grun 的方法之外，貌似又回到了起点。而事实上，在 theFutureIsNow 函数中，有一点不分好坏的排斥。它更容易阅读和编写，而开发人员也正在从异步执行中受益，但并非受益于它的全部内容。这里就有一个尖锐的问题了：“如果并行地去读取三个文件中的内容，效率会更高吗？”这个问题的回答取决于很多因素，比如，JavaScript 引擎是如何实现的，在用什么操作系统，以及所使用的文件系统。不过可以暂时把这些复杂的因素抛到一边，识别出其中的关键部分，那就是不管用什么样的顺序读取这三个文件，可以明确知道的是，运行效率的提升是因为允许这些文件的读操作可以并行发生。而这也是生成器运行器使人自鸣得意，误以为问题就这么被解决了的原因：这样写函数，是因为它看起来简单且直接。

问题（假设这里有问题）很好解决。Promise 提供了一个叫作 all 的方法，它会在数组中所有 promise 都被解决后才被调用，并且它有可能会同时执行异步代码。我们需要做的仅仅是修改函数，让它使用 Promise.all：

```
function* theFutureIsNow() {
   const data = yield Promise.all([
      nfcall(fs.readFile, 'a.txt'),
```

```
        nfcall(fs.readFile, 'b.txt'),
        nfcall(fs.readFile, 'c.txt'),
    ]);
    yield ptimeout(60*1000);
    yield nfcall(fs.writeFile, 'd.txt', data[0]+data[1]+data[2]);
}
```

`Promise.all` 返回的 promise 中会提供一个包含了所有被满足的 promise 值的数组，顺序跟数组中原来的 *promise* 顺序保持一致。即便 c.txt 可能在 a.txt 之前被读取，`data[0]`始终会保持 a.txt 的内容，而 `data[1]`则始终存储 c.txt 的内容。

本章节中需要大家掌握的并不是 `Promise.all`(即使这是一个很好用的工具)；需要掌握的是思考在程序中，哪些部分应该并行运行，哪些不用。在本例中，甚至超时都可以在读取文件的时候并行运行：具体如何判断取决于你要解决的问题。如果说本例中重要的是读取三个文件完毕后，然后再等待 60 秒，最后把文件内容合并后并写入另一个文件，其实这些已经实现了。另一方面，也许希望程序在读取三个文件的同时异步等待 60 秒，在读取文件结束后或者 60 秒结束后，将结果写入第四个文件。这种情况下，可能就需要把超时挪到 `Promise.all` 中了。

14.4.2 不要自己编写生成器运行器

虽然编写自己的生成器运行器是一个很好的练习，虽然已经完成了 grun，但它存在很多细微差别和改进的空间。最好不要重复造轮子。有一个叫作 co 的生成器运行器有很完备的功能，同时也很稳定。如果读者正在写网站，作者推荐 Koa，它是被设计用来与 co 一起用的，从而允许使用 yield 编写网络处理器，就像在 theFuturesIsNow 中一样。

14.4.3 生成器运行器中的异常处理

生成器运行器的另一个重要的好处是它们可以在 `try/catch` 中处理异常。还记得异常处理在 callback 和 promise 中的问题吗？在 callback 中抛出的异常不能在 callback 外面被捕获。由于生成器处理器在保留异步执行的同时能够使用同步语法，这样的好处就可以使用 `try/catch`。下面代码中给 `theFuturesIsNow` 函数添加几个异常处理：

```
function* theFutureIsNow() {
    let data;
    try {
        data = yield Promise.all([
          nfcall(fs.readFile, 'a.txt'),
```

```
            nfcall(fs.readFile, 'b.txt'),
            nfcall(fs.readFile, 'c.txt'),
        ]);
    } catch(err) {
        console.error("Unable to read one or more input files: " + err.
        message);
        throw err;
    }
    yield ptimeout(60*1000);
    try {
        yield nfcall(fs.writeFile, 'd.txt', data[0]+data[1]+data[2]);
    } catch(err) {
        console.error("Unable to write output file: " + err.message);
        throw err;
    }
}
```

并不是说 try…catch 中的异常处理就天生优于 promise 中的 catch 处理器，或者优于错误优先的回调，但是这却是一个容易理解的异常处理机制，如果读者更喜欢同步语法，那可能会希望使用 try…catch 来做异常处理。

14.5 小结

全面深入地理解异步编程中的复杂性，以及为了管理它所引入的各种机制—它们是理解现代 JavaScript 开发的关键。本章中学到了：

- JavaScript 中的异步执行是通过回调来管理的。
- promise 并不能替代回调函数，相反，promise 需要 then 和 catch 回调函数。
- promise 可以解决一个回调函数被多次调用的问题。
- 如果你需要多次调用一个回调函数，考虑使用事件（它可以跟 promise 结合使用）。
- promise 并不能保证它自己被处理，不过，可以把它们封装在超时中来避免 promise 不被处理。
- promise 可以被链式调用，实现简单的组合。
- promise 可以与生成器运行器结合使用，从而可以在不丢失异步执行的好处的前提下使用同步语法。
- 当用同步语法编写生成器函数时，应该仔细地去理解算法中的哪一部分可以被

并行运行，然后使用 Promise.all 来运行它们。

- 开发人员不应该编写自己的生成器运行器，应该使用 co 或者 Koa。
- 开发人员也不应该自己编写代码来将 Node 格式的回调转化到 promise 中，应该使用 Q。

异常处理通过生成器运行器可以在同步语法中工作。如果读者的编程经验仅限于使用具有同步语法的编程语言，那么学习 JavaScript 中的同步编程可能会让人望而生畏；至少对于作者来说是这样的。不过，这是现代 JavaScript 项目中的一项基本技能。

第 15 章

日期和时间

大部分实际中的应用都会涉及日期和时间数据。不幸的是，JavaScript 的 `Date` 对象（也存储了时间数据）不是这门语言最佳的特性之一。因为这个内置对象的功能有限，作者会引入 *Moment.js*，它扩展了 `Date` 对象的功能以满足日常需求。

最初，JavaScript 的 `Date` 对象是由美国网景公司的程序员 Ken Smith 实现的。实际上，他只是将 Java 中的 `java.util.Date` 的实现移植到 JavaScript 中。所以，说 JavaScript 与 Java 没有任何关系也不完全正确。如果有人问 JavaScript 和 Java 有什么关系，可以说，“除了 Date 对象和一个共同的语法祖先，就没什么了。”

因为一直使用“日期和时间”这个叫法显得冗长而又乏味，所以会用“时间”来表示“日期和时间”。没有指明时间的日期默认表示当天的凌晨 12:00。

15.1 日期、时区、时间戳以及 Unix 时间

事实是，现代的公历非常繁琐复杂，比如，时间是从 1 计数，奇怪的划分时间方式，以及闰年的出现。时区的计算更是增加了这种复杂度。然而必须面对它，因为它很通用。

我们从简单的内容开始：秒。不像公历中复杂的时间划分，秒很简单。日期和时间当用秒表示的时候，只是一个数字，在数轴上均匀排列。用秒表示日期和时间，是为了更好地计算。然而，它并不是沟通友好的：“Hey，Byron，想在第 1 437 595 200 秒一起吃午餐吗？”（1 437 595 200 表示 2015.7.22 13:00 周三，太平洋标准时间）。如果日期使用秒来表示，那么 0 表示什么日期呢？事实上，它并不代表耶稣诞生，

它只是一个普通的时间：1970 年 1 月 1 日 00:00:00 UTC。

大家可能已经知道，世界被划分为不同的时区（TZs），所以不论在哪里，7:00 AM 都是早上，7:00 PM 都是晚上。时区可以迅速地变复杂，尤其是把夏令时也考虑进来以后。作者不打算在本书中解释关于公历和时区的所有细节，维基百科上已经解释的很好了。然而，掌握一些基础知识还是很重要的，这有助于更好地理解 JavaScript 的 Date 对象（以及 Moment.js 带来的好处）。

所有时区都定义为以协调世界时（*Coordinated Universal Time*，缩写为 UTC。通过查阅维基百科可以了解命名背后复杂却有趣的原因）为基准的时间偏移量。有时候，UTC（并不完全正确）也叫格林威治标准时间（GMT）。例如，处在太平洋时区的俄勒冈州，太平洋时间比 UTC 晚 7～8 个小时。7～8？到底是哪个呢？其实这取决于季节。比如，夏天处在夏令时，时差是 7 个小时。而其他时候则是标准时间，时差为 8 个小时。关于时区，重点不是记住它们，而是理解如何表示不同时区的时间偏移量。打开一个 JavaScript console，输入 new Date()，就会看到如下内容：

```
Sat Jul 18 2015 11:07:06 GMT-0700 (太平洋时间)
```

注意，在这个冗长的格式中，时区由两种信息组成：相比于 UTC 的偏移量，和时区的名字（太平洋时间）。

在 JavaScript 中，所有 Date 实例都被存储成一个数字：距离 Unix 新纪元（1970 年 1 月 1 日 00:00:00）的毫秒数（不是秒）。不论何时请求一个 Date 实例（就像刚才看到的），JavaScript 会将那个数字转换成人们可以读懂的公历日期。如果想看日期的数字表示，调用 valueOf() 方法即可：

```
const d = new Date();
console.log(d);                 // 使用 TZ 格式化后的公历日期
console.log(d.valueOf());       // 距离 Unix 新纪元的毫秒数
```

15.2 构造 Date 对象

Date 对象有 4 种构造方式。构造时不传任何参数（就像已经看到的），它就只会返回一个表示当前日期的 Date 对象。也可以传入一个会被 JavaScript 解析的字符串，或者传入一个用毫秒来表示的、特定（本地）的日期。下面是一些例子：

```
// 以下所有的解释均是本地时间
new Date();                                    // 当前日期

//注意，在 JavaScript 中月份是从 0 开始的： 0=1 月，1=2 月，以此类推
```

```
new Date(2015, 0);                          // 12:00 A.M., Jan 1, 1970 UTC
new Date(2015, 1);                          // 12:00:01 A.M., Jan 1, 1970 UTC
new Date(2015, 1, 14);                      // 5:00 P.M., May 16, 2016 UTC
new Date(2015, 1, 14, 13);                  // 3:00 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30);              // 3:30 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30, 5);           // 3:30:05 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30, 5, 500); // 3:30:05.5 P.M., Feb 14, 2015

// 从 Unix 新纪元时间戳创建日期
new Date(0);                                // 12:00 A.M., Jan 1, 1969 UTC
new Date(1000);                             // 12:00:01 A.M., Jan 1, 1969 UTC
new Date(1463443200000);                    // 5:00 P.M., May 16, 2016 UTC

// 使用负数日期获取早于 Unix 新纪元的日期
new Date(-365*24*60*60*1000);          // 12:00 A.M., Jan 1, 1969 UTC

// 解析日期字符串（默认使用当地时间）
new Date('June 14, 1903');          // 12:00 A.M., Jun 14, 1903 local time
new Date('June 14, 1903 GMT-0000');   // 12:00 A.M., Jun 14, 1903 UTC
```

如果运行这些例子，会发现所有结果都是当地时间。除非处在 UTC（比如，延巴克图、马德里、格林威治）时区，那么返回的时间会与本例有所不同。这也成为使用 Date 对象的主要障碍：没有办法指定时区。其实，在 JavaScript 内部，时间都是以 UTC 存储的，然后根据当地时间（由计算机的操作系统决定）进行格式化。由于 JavaScript 起初是一门基于浏览器的脚本语言，这种设定时间的方式历来都是“正确的做法”。当你用 Date 的时候，你可能希望日期显示为用户所在的时区。然而，由于网络全球化的普及，同时 Node 使 JavaScript 在服务端得到应用，更加健全的时区处理将会很有用。

15.3 Moment.js

虽然本书是一本关于 JavaScript 语言的书，而非关于类库的书，但由于日期操作非常重要而且常见，所以作者决定在这里介绍一个功能强大而且非常好用的日期类库：*Moment.js*。

Moment.js 有两个版本：一种支持时区，一种不支持。由于支持时区的版本很大（因为它包含了全世界所有的时区信息），日常工作中可以选择不支持时区的版本。简单起见，后文中没有特殊说明的情况下，引用的都是支持时区的版本。如果想使用较小的版本，可以通过访问 *http://momentjs.com* 网站查找相关信息。

如果正在开发一个基于 Web 的项目，可以从像 cdnjs 这样的 CDN 中引用 Moment.js：

```
<script src="//cdnjs.cloudflare.com/ajax/libs/moment-timezone/0.4.0/
moment-timezone.min.js"></script>
```

如果在用 Node，可以用 `npm install -save moment-timezone` 安装 Moment.js，然后在代码中通过 `require` 引用它：

```
const moment = require('moment-timezone');
```

Moment.js 是一个很大的库，功能也很强大，如果需要操作日期，它几乎可以提供所有需要的功能。Moment.js 还有一个非常详细的官方文档可供参考。（*http://momentjs.com/*）。

15.4 JavaScript 中 Date 的实际用法

到现在为止，大家已经掌握了关于 Date 对象的基础知识，作者也介绍过 Moment.js 了，下面的内容会用一种跟之前略有不同的方式来介绍。因为对大多数读者来说，把 `Date` 对象中的方法全部介绍一遍不仅枯燥无味而且没什么实际意义。而且，如果读者确实需要它们，MDN 网站（*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/ Global_Objects/Date*）对 Date 对象的介绍非常详尽而且富有文采。

相对而言，本书将更多地以“菜谱”的方式（译者注：意为通过实际练习来讲解）来讲解，从而满足处理日期的一般需求，并且讲解如何以恰当地使用 `Date` 和 Moment.js 库。

15.5 构造日期对象

大家已经知道了构造 Date 对象的几种方式，它们已经能够满足绝大部分需求了。但是，在任何时候，当构造了一个没有明确指定时区的日期对象时，需要考虑程序中使用了哪个时区，而这取决于这个日期在哪里被构造。在过去，这难倒了很多初学者：比如，位于弗吉尼亚州的阿灵顿地区的服务器，和从加利福尼亚州的洛杉矶连接过来的浏览器，两者虽然使用了相同的日期代码，但人们会惊讶地发现这两者的时间相差了三个小时。

15.5.1 在服务端构造日期对象

如果在服务端构造日期对象，建议要么用 UTC，要么明确指定时区。如今，随着基于云的应用开发的兴起，同一套代码可以在世界上任何任何一台服务器上运行。如果构造的是当地的日期对象，那么就在自找麻烦。如果可以用 UTC 日期，那就

可以用 Date 对象的 UTC 方法来构造日期。

```
const d = new Date(Date.UTC(2016, 4, 27));  // May 27, 2016 UTC
```

Date.UTC 跟 Date 构造器具有相同的参数，不过它并不会返回一个新的 Date 实例，而是返回该日期的数值。然后将该数值传给 Date 构造器去创建一个 Date 实例。

如果需要在服务端构建一个具有指定时区（而非手动转换时区）的 Date 实例，可以使用 moment.tz：

```
// 给 Moment.js 传入一个数组时使用了与 JavaScript 的 Date 构造器一样的参数，其中包含
// 了从 0 开始的月份（0=1 月，1=2 月，等等）。toDate() 方法将结果转换成过一个
// JavaScript 日期对象
const d = moment.tz([2016, 3, 27, 9, 19],'America/Los_Angeles').toDate();
```

15.5.2 在浏览器中构造 Date 对象

一般来说，JavaScript 的默认行为都是适合浏览器的。浏览器能检测出当前操作系统所在的时区，用户通常也倾向于使用当地时间。如果你正在开发一个用其他时区来处理日期的应用，可能需要 Moment.js 来帮助处理这个需求，并将日期显示为其他时区。

15.6 传递日期

随着传递日期（不管是服务器给浏览器传送日期还是反过来）的出现，事情开始变得有趣起来。服务器和浏览器可能处于不同的时区，而用户也想使用本地时间。幸运的是，JavaScript 的 Date 实例存储的是距离 Unix 新纪元 UTC 时间的毫秒数，所以通常情况下，反复传递 Date 对象是安全的。

对于“传递”这个词的意思似乎有些模棱两可，那么它究竟是什么意思？确保日期被安全传递的最可靠方式是使用 *JavaScript 对象表示法*（*JavaScript Object Notation*，也就是常说的 JSON）。然而遗憾的是，在 JSON 中使用日期会破坏 JSON 的对称解析特性，因为 JSON 规范中并没有支持日期的数据类型：

```
const before = { d: new Date() };
before.d instanceof date                    // true
const json = JSON.stringify(before);
const after = JSON.parse(json);
after.d instdanceof date                    // false
typeof after.d                              // "string"
```

也就是说，在 JavaScript 中，JSON 不能无缝且对称地处理日期。不过幸运的是，JavaScript 中的序列化字符串功能能够始终保持数据的一致性，所以可以从字符串中“恢复”一个日期。

```
after.d = new Date(after.d);
after.d instanceof date          // true
```

不论一开始用什么时区去创建日期，当它被编码成 JSON 的时候，它使用的都是 UTC 时间，同时，当这个 JSON 格式的字符串传给 Date 构造器的时候，其中的日期都会被转换成当地时区然后显示出来。

另一个在客户端和服务端之间安全传递日期的方式是简单地使用日期的数字值：

```
const before = { d: new Date().valueOf() };
typeof before.d                                  // "number"
const json = JSON.stringify(before);
const after = JSON.parse(json);
typeof after.d                                   // "number"
const d = new Date(after.d);
```

JavaScript 将日期编码成 JSON 字符串后能够始终保持一致，这是其他语言和平台提供的 JSON 类库所不具备的特性。在.NET 中，JSON 序列化器会按照它们自己专有的格式序列化日期对象。所以如果与其他系统的交互是通过 JSON 来进行的，你就要留心其序列化日期的方式。如果可以操作源代码，最好使用基于 Unix 新纪元偏移量的数字传递日期。除此之外，你还是要格外小心：因为日期类库通常使用秒作为日期的数值，而非毫秒。

15.7 展示日期

对于初学者来说，将日期格式化并展示出来，通常是最让人沮丧的问题之一。JavaScript 的内建 Date 对象只包含了少数预置的日期格式，如果它们不能满足需求，自己去做格式化会是一件很痛苦的事。幸运的是，Moment.js 在这个领域很擅长，如果需要展示特殊格式的日期，建议使用它。

使用 Moment.js 格式化日期时，调用它的 format 方法即可。format 方法接收一个字符串元字符，该元字符会被对应的日期组件替代。例如，字符串"YYYY"会被替换成 4 位数的年份。这里有一些例子，分别是使用 Date 对象的内建方法格式化日期，以及使用更强大的 Moment.js 方法来进行格式化：

```
const d = new Date(Date.UTC(1930, 4, 10));
```

```
// 这些是洛杉矶所在时区的输出结果

d.toLocaleDateString()     // "5/9/1930"
d.toLocaleFormat()         // "5/9/1930 4:00:00 PM"
d.toLocaleTimeString()     // "4:00:00 PM"
d.toTimeString()           // "17:00:00 GMT-0700 (Pacific Daylight Time)"
d.toUTCString()            // "Sat, 10 May 1930, 00:00:00 GMT"

moment(d).format("YYYY-MM-DD");          // "1930-05-09"
moment(d).format("YYYY-MM-DD HH:mm");    // "1930-05-09 17:00
moment(d).format("YYYY-MM-DD HH:mm Z");  // "1930-05-09 17:00 -07:00
moment(d).format("YYYY-MM-DD HH:mm [UTC]Z");// "1930-05-09 17:00 UTC-07:00

moment(d).format("dddd, MMMM [the] Do, YYYY"); // "Friday, May the 9th,1930"

moment(d).format("h:mm a");                 // "5:00 pm"
```

从这些例子中大家可以看出，JavaScript 内建的格式化日期方法有很多不一致的地方，而且使用起来不灵活。JavaScript 尝试用这些内建的格式化选项来支持用户所属地区的日期格式化。如果需要支持多个地区的日期格式化，这会是一个相对廉价但缺乏灵活性的方式。

这个例子并不是为了演示 Moment.js 中多种多样的格式化方式（想了解所有格式化方式，可以参考 Moment.js 的在线文档），它想传达的意思是：Moment.js 几乎可以满足任何格式化日期的需求。像很多格式化日期的元语言一样，它也有一些通用的约定。比如，字符越多表示信息越详细；也就是说，“M”的结果是 1、2、3……；“MM”是 01、02、03……；“MMM”为 Jan、Feb、Mar……，“MMMM”为 January、February、March，……小写的“o”会生成序号：“Do”的的结果是 1st、2nd，等等。如果不想让字母被解释为元字符，可以用方括号将它们括起来：“[M]M”生成的结果是 M1、M2，以此类推。

Moment.js 还未完全解决的一个问题是时区缩写的使用，例如，EST 和 PST。因为没有一个统一的国际标准，Moment.js 已经弃用格式化字符 z。关于时区缩写问题的详细讨论可以参考 Moment.js 官方文档。

15.8 日期的组成

如果需要访问一个 Date 实例的某个组成部分，可以使用这些方法：

```
const d = new Date(Date.UTC(1815, 9, 10));

// these are the results someone would see in Los Angeles
// 在洛杉矶的人看到的结果
d.getFullYear()              // 1815
d.getMonth()                 // 9 - October
d.getDate()                  // 9
d.getDay()                   // 1 - Monday
d.getHours()                 //17
d.getMinutes()               // 0
d.getSeconds()               // 0
d.getMilliseconds()          // 0

// 上述时间所对应的 UTC 时间
d.getUTCFullYear()           // 1815
d.getUTCMonth()              // 9 - October
d.getUTCDate()               // 10
// ...等等.
```

如果你在用 Moment.js，你会发现很少会用到某个单独的日期组件，但知道有这些组件总不是一件坏事。

15.9 日期的比较

简单的日期比较，如日期 A 和日期 B 谁在前谁在后？可以使用 JavaScript 内建的比较运算符。记住，因为 Date 实例是以数值存储日期的，所以比较运算符比较的是它们的数值大小。

```
const d1 = new Date(1996, 2, 1);
const d2 = new Date(2009, 4, 27);

d1 > d2     // false
d1 < d2     // true
```

15.10 日期的四则运算

因为日期只是一些数字，所以可以将日期相减从而获取它们相差的毫秒数：

```
const msDiff = d2 - d1;                    // 417740400000 ms
const daysDiff = msDiff/1000/60/60/24;     // 4834.96 days
```

这个特性使得用 Array.prototype.sort 对日期进行排序变得很简单：

```
const dates = [];
// 创建一些随机日期
```

```
const min = new Date(2017, 0, 1).valueOf();
const delta = new Date(2020, 0, 1).valueOf() - min;
for(let i=0; i<10; i++)
   dates.push(new Date(min + delta*Math.random()));
// 日期是随机的，并且（有可能）是无序的，我们可以对它们进行排序（降序）
dates.sort((a, b) => b - a);
// or ascending:
dates.sort((a, b) => a - b);
```

Moment.js 提供了很多强大的方法来对日期做一般的四则运算，你可以增加或减少任意的时间单元：

```
const m = moment();                    // 当前时间
m.add(3, 'days');                      // m 为当前时间往后推 3 天
m.subtract(2, 'years');                // m 为 2 年前再往后推 3 天

m = moment();                          // 重置
m.startOf('year');                     // m 为今年 1 月 1 日
m.endOf('month');                      // m 为今年 1 月 31 日
```

Moment.js 还提供了方法的链式调用：

```
const m = moment()
   .add(10, 'hours')
   .subtract(3, 'days')
   .endOf('month');

// m 为未来 10 个小时以后的时间点倒退 3 天后所在月份的月底。
```

15.11 用户友好的相对日期

通常，一个比较好的实践是使用相对日期来表示日期信息：比如，相对于某个日期的“三天前”。Moment.js 让这件事情变得非常容易：

```
moment().subtract(10, 'seconds').fromNow();          // 10 秒之前
moment().subtract(44, 'seconds').fromNow();          // 44 秒之前
moment().subtract(45, 'seconds').fromNow();          // 45 秒之前
moment().subtract(5, 'minutes').fromNow();           // 5 分钟之前
moment().subtract(44, 'minutes').fromNow();          // 44 分钟之前
moment().subtract(45, 'minutes').fromNow();          // 1 个小时之前
moment().subtract(5, 'hours').fromNow();             // 4 小时之前
moment().subtract(21, 'hours').fromNow();            // 21 小时之前
moment().subtract(22, 'hours').fromNow();            // 一天之前
moment().subtract(344, 'days').fromNow();            // 344 天之前
```

```
moment().subtract(345, 'days').fromNow();          // 一年以前
```

正如大家所见，Moment.js 任意（但合理）选择了一些断点来决定什么时候展示成另一种时间单元。这就方便我们获取一个用户友好的相对日期。

15.12 小结

如果你可以从本章中学到三样东西，它们应该是：

- 在 JavaScript 内部，日期是以距离 Unix 新纪元（1970 年 1 月 1 日 UTC）的毫秒数存储的。
- 构建日期的时候要注意时区。
- 如果对日期进行复杂的格式化，考虑使用 Moment.js。

在大部分真实应用中，日期和时间处理是难以避免的。希望本章已经为读者理解这些重要概念打下了坚实的基础。同时，火狐开发者社区（Mozilla Developer Network）和 Moment.js 文档也提供了十分详细的参考。

第 16 章

数学运算

本章将讨论 JavaScript 的内置 `Math` 对象，它包含了在应用程序开发中会经常遇到的与数学运算相关的函数（如果你需要做精细的数字分析，那么可能还需要找一些第三方类库）。

在深入讲解类库之前，先来回忆一下 JavaScript 是如何处理数字的。尤其是 JavaScript 中没有专门的整数类型的 class，所有数字都是 IEEE 754 标准中的 64 位浮点型数字。对 math 库中的大部分函数来说，事情都被简化了：数字就是数字。虽然计算机无法完全表示一个实数，不过出于实际目标，可以把 JavaScript 中的数字当成实数。注意，在 JavaScript 中，没有对复杂数字类型的内建支持。如果需要复杂数字，大数字，更复杂的结构或算法，建议使用 Math.js。

除去基本内容，本章并不是在教大家数学，如果需要，会有一本（或 2 本、或 10 本）专门的书来讲解数学。

在本章的代码注释中，会用波浪线（~）作为前缀来表示近似值。同时，对 Math 对象中的属性，也会用*函数*来指明，而非*方法*。从技术上讲，它们都是静态方法，而其中的区别也只是理论上的，这是因为 Math 对象为开发人员提供了命名空间，而非上下文。

16.1 格式化数字

格式化数字是一个很常见的需求，也就是说，不显示 2.0093，而显示 2.1。或者把

1949032 显示成 1,949,032[①]。

JavaScript 对格式化数字的内建支持很少，不过还是包括了对固定位数的小数位，固定精度，和指数符号的支持。此外，还支持显示基于不同进制的数字，比如二进制，八进制，和十六进制。

不可避免的是，JavaScript 中所有格式化数字的方法都会返回一个字符串，而非数字，因为只有字符串可以保证其格式正是我们想要的（不过，想要把字符串转换成数字却很简单）。这样做就迫使大家只有在显示数字之前才对其进行格式化，而在存储数字或者使用它们进行计算时，它们应该保持被格式化之前的类型。

16.1.1 固定小数

如果需要一个有固定小数位的数字，可以用 Number.prototype.toFixed：

```
const x = 19.51;
x.toFixed(3);           // "19.510"
x.toFixed(2);           // "19.51"
x.toFixed(1);           // "19.5"
x.toFixed(0);           // "20"
```

注意这里并不是简单粗暴的将数字截断：这个函数的输出是对指定小数位进行四舍五入的结果。

16.1.2 指数符号

如果希望用指数形式来显示数字，可以用：Number.prototype.toExponential:

```
const x = 3800.5;
x.toExponential(4);         // "3.8005e+4";
x.toExponential(3);         // "3.801e+4";
x.toExponential(2);         // "3.80e+4";
x.toExponential(1);         // "3.8e+4";
x.toExponential(0);         // "4e+4";
```

与 Number.prototype.toFixed 一样，结果也是四舍五入的。指定的精度是小数位的个数。

16.1.3 固定精度

如果读者更关心数字的个数（不论小数点位在哪里），可以用 Number.prototype. toPrecision：

① 在一些文化中，小数点被用作千位分隔符，而逗号用作小数点分隔符，这可能跟你的使用习惯有所出入。

```
let x = 1000;
x.toPrecision(5);       // "1000.0"
x.toPrecision(4);       // "1000"
x.toPrecision(3);       // "1.00e+3"

x.toPrecision(2);       // "1.0e+3"
x.toPrecision(1);       //\"1e+3"
x = 15.335;
x.toPrecision(6);       // "15.3350"
x.toPrecision(5);       // "15.335"
x.toPrecision(4);       // "15.34"
x.toPrecision(3);       // "15.3"
x.toPrecision(2);       // "15"
x.toPrecision(1);       // "2e+1"
```

输出结果依旧会四舍五入，同时有一个指定位数的精度。必要时，输出会是指数。

16.1.4 不同进制

如果想要显示不同进制的数字（比如二进制，八进制，或者十六进制），`Number.prototype.toString` 可以接收一个指定进制（从 2 到 36）从而实现格式化：

```
const x = 12;
x.toString();           // "12"  (base 10)
x.toString(10);         // "12"  (base 10)
x.toString(16);         // "c"   (hexadecimal)
x.toString(8);          // "14"   (octal)
x.toString(2);          // "1100" (binary)
```

16.1.5 进一步格式化数字

如果要在应用中显示很多数字，可能需要迅速掌握 JavaScript 提供的内建方法。经常使用的有：

- 千位分隔符。
- 用不同形式来显示负数（比如使用括号）。
- 工程符号（跟指数符号类似）。
- 国际单位制的前缀（毫[milli]、微[micro]、千[kilo]、百万[mega]，等等）。

如果读者需要一些练习，可以试着实现上面这些方法，就能学到很多东西。如果不需要练习，那作者推荐看看 Numberal.js 库，除了上面提到的功能，这个库还提供了很多其他功能。

16.2 常量

有一些很常用且重要的常量可以作为 Math 对象的属性来使用：

```
// 基本常数
Math.E              // 自然对数的底：~2.718
Math.PI             // 圆周率：~3.142

// 常用对数-这些值都可以通过调用库方法来计算得出，但是它们的使用频率如此之高，
// 以至于被设置为常用对数。
Math.LN2            // 2 的自然对数：~0.693
Math.LN10           // 10 的自然对数：~2.303
Math.LOG2E          // 以 2 为底的 Math.E 的对数：~1.433
Math.LOG10E         // 以 10 为底的 Math.E 的对数：0.434

// 代数常数
Math.SQRT1_2        // 1/2  的平方根：~0.707
Math.SQRT2          // 2 的平方根：~1.414
```

16.3 代数函数

16.3.1 幂运算

基本的幂函数是 Math.pow，还有一些便捷函数用于计算平方根、立方根和 e 的幂，如表 16-1 所示。

表 16-1　　幂函数

函　数	描　述	例　子
Math.pow(x, y)	x^y	Math.pow(2, 3) // 8 Math.pow(1.7, 2.3)　// ~3.39
Math.sqrt(x)	251658240 $\sqrt{x}$ 等于 Math.pow(x, 0.5)	Math.sqrt(16) // 4 Math.sqrt(15.5)　// ~3.94
Math.cbrt(x)	X 的立方根，等于 Math.pow(x, 1/3)	Math.cbrt(27) //3 Math.cbrt(22) // ~2.8
Math.exp(x)	e^x 等于　Math.pow(Math.E, x)	Math.exp(1) // ~2.718 Math.exp(5.5)　// ~244.7
Math.expm1(x)	$e^x - 1$ 等于 Math.exp(x) - 1	Math.expm1(1) // ~1.718 Math.expm1(5.5)　// ~243.7
Math.hypot(x1, x2,...)	参数的平方根之和： $\sqrt{x1^2 + x2^2 + \cdots}$	Math.hypot(3, 4)　// 5 Math.hypot(2, 3, 4) // ~5.36

16.3.2 对数函数

基本的对数函数是 `Math.log`。在一些编程语言中，“log”就表示“以 10 为底的对数”，“ln”则表示“自然对数”，所以要记住在 JavaScript 中，“log”表示“自然对数”。ES6 中为了方便计算，引入了 `Math.log10`。表 16-2 所列为对数函数。

表 16-2　　对数函数

函　数	描　述	例　子
Math.log(x)	x 的自然对数	Math.log(Math.E) // 1 Math.log(17.5) // ~2.86
Math.log10(x)	以 10 为底的 x 的对数，等于 Math.log(x)/ Math.log(10)	Math.log10(10) // 1 Math.log10(16.7) // ~1.22
Math.log2(x)	以 2 为底的 x 的对数，等于 Math.log(x)/ Math.log(2)	Math.log2(2) // 1 Math.log2(5) // ~2.32
Math.log1p(x)	1+x 的自然对数，等于 Math.log(1 + x)	Math.log1p(Math.E - 1) // 1 Math.log1p(17.5) // ~2.92

16.3.3 其他函数

表 16-3 列出了一些其他的数学函数，这些函数可以完成一些常用的操作，比如，计算绝对值、向上取整（大于或等于某个数的最小整数）、向下取整（小于或等于某个数的最大整数）或符号计算（正负号），以及查找一系列数字中的最大值和最小值。

表 16-3　　数字相关的代数函数

函　数	描　述	例　子
Math.abs(x)	x 的绝对值	Math.abs(-5.5) // 5.5 Math.abs(5.5) // 5.5
Math.sign(x)	x 的符号： 如果 x 是负数，返回−1; 如果 x 是正数，返回 1; 如果 x 是 0，返回 0	Math.sign(−10.5) // −1 Math.sign(6.77) // 1
Math.ceil(x)	x 向上取整：大于或等于 x 的最小整数	Math.ceil(2.2) // 3 Math.ceil(−3.8) // −3
Math.floor(x)	x 向下取整：小于或等于 x 的最大整数	Math.floor(2.8) // 2 Math.floor(−3.2) // −4
Math.trunc(x)	x 的整数值（去掉所有小数位）	Math.trunc(7.7) //7 Math.trunc(−5.8) // −5

续表

函　数	描　述	例　子

Math.round(x)	四舍五入	Math.round(7.2) //7
		Math.round(7.7) //8
		Math.round(−7.7) // −8
		Math.round(−7.2) // −7

Math.min(x1, x2,...)	返回参数中的最小值	Math.min(1, 2) // 1
		Math.min(3, 0.5, 0.66) // 0.5
		Math.min(3, 0.5, −0.66) // −0.66

Math.max(x1, x2,...)	返回参数中的最大值	Math.max(1, 2)　// 2
		Math.max(3, 0.5, 0.66) // 3
		Math.max(−3, 0.5, −0.66) // 0.5

16.3.4　伪随机数生成器

伪随机数生成器是由 Math.random 提供的，它会生成一个大于等于 0 且小于 1 的伪随机数字。读者可能记得在代数中，数字的范围是用方括号（包含）和圆括号（不包含）表示的。在这种表示法下，Math.random 返回一个在[0, 1)范围内的数字。

`Math.random` 并没有提供任何常用函数，可以生成其他范围内的伪随机数，表 16-4 展示了一些实现这个功能的常用公式。在这个表中，x 和 y 表示实数，而 m 和 n 表示整数。

表 16-4　　生成伪随机数

范　围	例　子
[0, 1)	Math.random()
[x, y)	x + (y-x)*Math.random()
在 [m, n)范围内的整数	m + Math.floor((n-m)*Math.random())
在 [m, n] 范围内的整数	m + Math.floor((n-m+1)*Math.random())

JavaScript 的伪随机数经常被诟病的一点是：无法对它设置随机种子，但是它对有些涉及伪随机数的算法的测试又很重要。如果需要使用随机种子来产生伪随机数，可以看看 David Bau 的 seedrandom.js 包（https://github.com/davidbau/seedrandom）。

将伪随机数生成器（PRNGs）简单称作“随机数生成器”很常见（但这是不对的）。在大多数实际应用中，PRNGs 生成的数字只是看似很随机，但实际上实现真正的随机数生成器，是非常困难的。

16.4 三角函数

关于三角函数没有什么特别的内容。正弦、余弦、正切，以及它们的反函数这里都有，如表 16-5 所示。在 Math 库中，所有的三角函数都是基于弧度的，而非角度。

表 16-5 数字三角函数

函　数	描　述	例　子
Math.sin(x)	弧度 x 的正弦	Math.sin(Math.PI/2) // 1 Math.sin(Math.PI/4) // ~0.707
Math.cos(x)	弧度 x 的余弦	Math.cos(Math.PI) // −1 Math.cos(Math.PI/4) // ~0.707
Math.tan(x)	弧度 x 的正切	Math.tan(Math.PI/4) // ~1 Math.tan(0) // 0
Math.asin(x)	X 的反正弦（arcsin）（结果为弧度）	Math.asin(0)　// 0 Math.asin(Math.SQRT1_2) // ~0.785
Math.acos(x)	X 的反余弦（arccos）（结果为弧度）	Math.acos(0) // ~1.57+ Math.acos(Math.SQRT1_2) // ~0.785+
Math.atan(x)	X 的反正切（arctan）（结果为弧度）	Math.atan(0) // 0 Math.atan(Math.SQRT1_2) // ~0.615
Math.atan2(y, x0)	从 x 轴到某一点 (x, y) 逆时针旋转的角度（弧度）	Math.atan2(0, 1) // 0 Math.atan2(1, 1) // ~0.785

如果读者在使用角度，那就需要把它们转化成弧度。计算很简单：度数除以 180 然后乘以π。也很容易编写一个辅助函数：

```
function deg2rad(d) { return d/180*Math.PI; }
function rad2deg(r) { return r/Math.PI*180; }
```

16.5 双曲线函数

像三角函数一样，双曲线函数也是标准函数，如表 16-6 所示。

表 16-6 数字双曲线函数

Function	Description	Examples
Math.sinh(x)	X 的双曲正弦	Math.sinh(0) // 0 Math.sinh(1) // ~1.18
Math.cosh(x)	X 的双曲余弦	Math.cosh(0) // 1 Math.cosh(1) // ~1.54

续表

Function	Description	Examples
Math.tanh(x)	X 的双曲正切	Math.tanh(0) // 0 Math.tanh(1) // ~0.762
Math.asinh(x)	X 的反双曲正弦（arcsinh）	Math.asinh(0) // 0 Math.asinh(1) // ~0.881

Math.acosh(x)	X 的反双曲余弦（arccosh）	Math.acosh(0) // NaN
		Math.acosh(1) // 0

Math.atanh(x)	X 的反双曲正切（arctanh）	Math.atanh(0) // 0 Math.atanh(0) // ~0.615

第 17 章

正则表达式

正则表达式提供了复杂的字符串匹配功能。如果想匹配那些“看起来像”一个邮件地址、URL 或手机号的字符串，正则表达式能够手到擒来。有了字符串匹配之后，很自然地会联想到字符串替换，而正则表达式也支持这个功能。比如，如果想匹配一个看起来像邮件地址的字符串并且将它替换成该邮件地址的超链接。

很多关于正则表达式的介绍中都会用一些晦涩难懂的例子，例如，“匹配 aaabb 和 abaaba 而不匹配 abba”，它的好处是大大降低了正则表达式的复杂度，坏处是它几乎没有任何意义（读者什么时候才会匹配 aaaba？）。所以，作者打算从一开始就用实际的例子来介绍正则表达式的特性。

正则表达式通常缩写为“regex”或“regexp”；简单起见，本书会使用前者。

17.1 子字符串匹配和替换

正则表达式的本质是匹配一个字符串的子字符串，或者替换该子字符串。它将以难以想象的强大功能和灵活性来帮助程序员完成这件事，所以在开始进一步学习之前，先简单地过一下 `String.prototype` 的非正则搜索和替换功能，该功能可以满足一些非常基本的搜索和匹配需求。

如果只想知道某个子字符串是否包含在一个更大的字符串中，下面的 `String.prototype` 方法就可以实现：

```
const input = "As I was going to Saint Ives";
input.startsWith("As")          // true
input.endsWith("Ives")          // true
input.startsWith("going", 9)    // true - 从下标 9 开始
```

```
input.endsWith("going", 14)      // true - 将下标 14 当做字符串的结尾
input.includes("going")          // true
input.includes("going", 10)      // false - 从下标 10 开始
input.indexOf("going")           // 9
input.indexOf("going", 10)       // -1
input.indexOf("nope")            // -1
```

注意，这里的所有方法都区分大小写。所以 input.startsWith("as")会返回 fasle。如果不想区分大小写，将输入转化成小写即可：

```
input.toLowerCase().startsWith("as")          // true
```

还要注意，上面的语句并没有改变原始字符串；String.prototype.toLowerCase 会返回一个新字符串，而不会修改原始字符串（记住，在 JavaScript 中，字符串是不可变的）。

如果想做进一步操作，比如替换掉刚才匹配到的子字符串，可以使用 String. prototype.replace：

```
const input = "As I was going to Saint Ives";
const output = input.replace("going", "walking");
```

同样，原始字符串（input）没有被这个替换语句修改。替换后，在 output 这个新字符串中，“going”已经被替换为“walking”（当然，如果真的想更改 input 的值，可以将替换后的值再赋给 input）。

17.2 构造正则表达式

在开始接触复杂的正则表达式元语言之前，先来看看它们在 JavaScript 中是如何被构造和使用的。在下面的例子中，作者会像之前一样搜索一个特定的字符串，虽然这有点杀鸡用牛刀的意思，但却不失为一种简单理解如何使用正则表达式的方式。

在 JavaScript 中，正则表达式可以通过 RegExp 这个类来表示。虽然可以用它的构造器来构造一个正则表达式，但是如果没有独立的字面量语法，怎么能体现它的重要性呢？正则表达式的字面量被包在一对斜杠中：

```
const re1 = /going/;                  // 可以搜索"going"的正则表达式
const re2 = new RegExp("going");      // 使用对象构造器的等价形式
```

本章后面会讲到一个使用需要 RegExp 构造器的特殊情况，除此之外，都应该优先使用更方便的字面量语法。

17.3 使用正则表达式进行搜索

有了正则表达式之后，当需要在字符串中进行搜索时，就有了更多选择 。

为了理解不同的替换方式，先稍微预习一下正则表达式元语言。在这里使用静态字符串就太无趣了，所以用了动态表达式/\w{3.}/ig，它会匹配所有包含三个或三个以上字母的单词（不区分大小写）。现在先不用急着去理解这个表达式，到了本章后面自然会理解。先来看一些可用的搜索方法：

```
const input = "As I was going to Saint Ives";
const re = /\w{3,}/ig;

// 从字符串（input）开始
input.match(re);          // ["was", "going", "Saint", "Ives"]
input.search(re);         // 5 (首次在下标 5 中出现了满足表达式的单词)

// 从正则表达式开始（re）
re.test(input);           // true (input 至少包含一个三个字母的单词)
re.exec(input);           // ["was"] (第一个匹配)
re.exec(input);           // ["going"] (exec 会"记住"它所在的位置)
re.exec(input);           // ["Saint"]
re.exec(input);           // ["Ives"]
re.exec(input);           // null - 匹配完毕

// 注意，所有这些方法都可以直接使用字面量语法
input.match(/\w{3,}/ig);
input.search(/\w{3,}/ig);
/\w{3,}/ig.test(input);
/\w{3,}/ig.exec(input);
// ...
```

在这些方法中，提供信息最多的是 RegExp.prototype.exec，但读者会发现实际中用的最少的也是它。作者自己用的最多的是 String.prototype.match 和 RegExp.prototype.test。

17.4 使用正则表达式进行替换

前面讲到的用来替换简单字符串的 String.prototype.replace 方法还可以接收一个正则表达式作为参数，如此一来它就做更多的事情。来看一个简单的例子：替换所有的 4 字母单词。

```
const input = "As I was going to Saint Ives";
const output = input.replace(/\w{4,}/ig, '****');  // "As I was ****
                                                    // to **** ****"
```

在本章后面，读者会学到更为复杂的替换方法。

17.5 消费输入

关于正则表达式，从“天真”的角度来看，它只是“在一个大字符串中查找子字符串”（文艺点就叫作“海底捞针”）。虽然这个天真的想法通常能够满足全部需求，但它却限制了真正理解正则表达式的含义，以及如何利用正则表达式去完成更有挑战性的任务。

而从复杂的角度来看，正则表达式是一个消费给定字符串的模式。匹配（要寻找的内容）则成为了这种理念的副产品。

为了将正则表达式的工作方式概念化，有一个比较好的办法是，将它比作一个普通的儿童文字游戏：一个需要找出其中所含单词的字母网格。简单起见，忽略斜向和纵向匹配。事实上，这里只考虑网格中的第一行。

```
X J A N L I O N A T U R E J X E E L N P
```

人们很擅长玩这个游戏。当看到它的时候，就能很快地找出 LION、NATURE 和 EEL（以及 ION，其实已经找出来了）。计算机或者正则表达式可没有这么聪明。如果以正则表达式的视角来玩这个游戏，这样不仅能看到正则表达式的工作方式，而且还能了解很多需要注意的限制因素。

简单起见，告诉正则表达式要查找的单词是 LION、ION、NATURE 和 EEL；也就是说，预先给出答案，然后再看答案是否能通过验证。

正则表达式会从第一个字符 X 开始查找。它注意到要查找的单词中没有以 X 开始的，所以就说“没有匹配项”。不过它并没有放弃，而是移动到下一个字符 J 继续查找。随后它发现 J 的情况跟 X 一样，继续移动到 A。在沿字符串移动时，就认为正则表达式移动所经过的字母被消费了。碰到 L 的时候，事情开始变得有趣起来 。此时，正则表达式引擎会说：“呀，这个可能是 LION!”由于这可能是一个潜在的匹配项，所以 *L 不会被消费*，这是理解此处的关键。接着正则表达式会继续执行，匹配 I、O，以及 N。至此，它就识别出了一个匹配项：匹配成功！当识别出一个匹配项后就可以消费掉整个单词，所以 L、I、O 和 N 都被消费了。此时事情变得更有趣了，LION 和 NATURE 有重叠的地方。作为人类，不会被这个现象迷惑。但是正则表达式非常严格，它不会再去查找已经消费过的字符。所以它不会“退回去”

尝试在已经被消费的字符中查找匹配项。这样一来，正则表达式就找不到 NATURE 了，因为 N 已经被消费了。它只能找到 ATURE，而这不符合所要查找的单词。当然，最后它还会找到 EEL。

现在重新回顾这个例子，将 LION 中的 O 改成 X。那么会发生什么呢？当正则表达式碰到 L 的时候，它还是会识别出一个潜在的匹配项（LION），因此它不会消费 L。当它移动到 I 的时候，也不会消费 I。而当移动到 X 的时候，此时，它意识到这不是一个匹配项：因为没有以 LIX 开始的目标单词。接下来，正则表达式会回到 L 并消费它，然后继续正常地移动。在这种情况下，它将匹配上 NATURE，因为 N 没有被当做 LION 的一部分消费掉。

图 17-1 是这个过程的一部分示例：

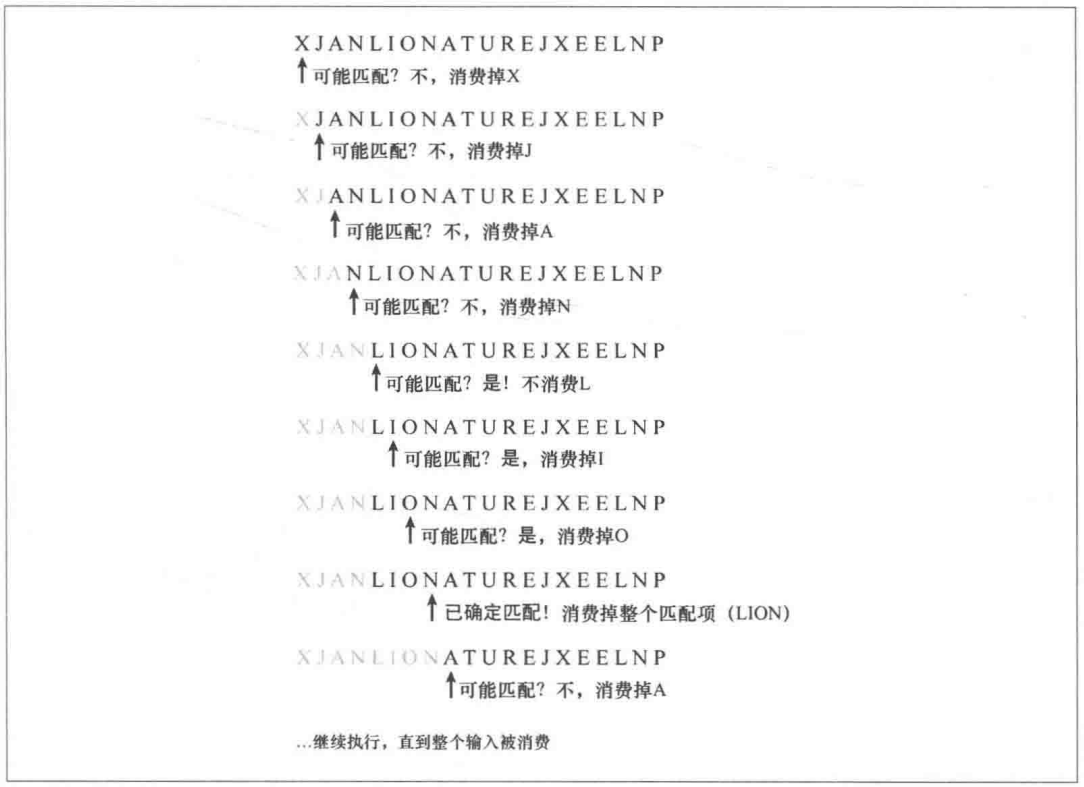


图 17-1　正则表达式示例

在讨论正则表达式元语言之前，先来抽象地思考一下正则表达式在“消费”字符串时使用的算法：

- 字符串从左到右被消费。

- 一旦某个字符已经被消费，就不会被再次访问。
- 如果匹配失败，正则表达式每次只会往下找一个字符，从而试图查找匹配项。
- 如果匹配成功，正则表达式会立即消费匹配项的所有字符；然后继续用下一个字符进行匹配（如果正则表达式是全局的话，这个会在后面讲到）

这是个普通的算法，即使细节有些复杂，可能也不会感到惊讶。特别是，当正则表达式确定没有任何匹配项时，该算法会被提前终止。

在学习正则表达式元语言特性的时候，试着记住这个算法。想象一下字符串从左到右被消费掉，每次一个字符，直到找到匹配项，此时，整个匹配项的字符会一次性被消费。

17.6 分支

如果有一个以字符串形式存储的 HTML 页面，想找出其中所有引用外部资源的标签（<a>、<area>、<link>、<script>、<source>，有时候还有<meta>）。此外，可能还有一些标签是混合大小写的（例如<Area>和<LINKS>）。正则表达式中的分支可以解决这个问题：

```
const html = 'HTML with <a href="/one">one link</a>,and some JavaScript.' +
   '<script src="stuff.js"></script>';
const matches = html.match(/area|a|link|script|source/ig);  // 第一次尝试
```

竖线（|）是一个正则表达式元字符，它表示分支。ig 表示忽略大小写（i）和全局搜索（g）。没有 g 的话，只会返回第一个匹配项。这个正则表达式会被解读为“查找文本中所有的 *area*、*a*、*link*、*script* 或 *source*，并忽略大小写”。聪明的读者可能会好奇为什么把 *area* 放在 *a* 之前，这是因为正则表达式是从左到右执行分支匹配的。换句话说，如果字符串包含了 *area* 标签，它会匹配上 a 然后继续移动。此时 a 会被消费，而 rea 则匹配不上任何元素。所以必须先匹配 area，然后再匹配 a，否则，area 将匹配失败。

如果运行这个例子，会发现有很多意料之外的匹配项：单词 *link*（在<a>标签内部），字母 a 不是一个 HTML 标签，它只是英文单词中的一部分。解决这个问题的一个办法是将正则表达式修改为/<area|<a|<link|<script|<source/（尖括号不是正则表达式元字符），但即便如此，还是会遇到更加复杂的情况。

17.7 匹配 HTML

在之前的例子中，作者用正则表达式执行了一个很常见的任务：匹配 HTML。虽然这只是一个常见任务，但也必须提醒大家，通过使用正则表达式，能用 HTML 做很多有用的事情，但却不能用正则表达式解析 HTML。解析表示将东西全部拆分成组件。正则表达式只能解析正规的语言（顾名思义）。正规语言非常简单，在绝大多数时候，只会在更加复杂的语言上使用正则表达式。既然正则表达式可以在更加复杂的语言上物尽其用，那为什么不能用来解析 HTML 呢？因为一定要先搞清楚正则表达式的不足，才能知道到什么时候应该使用更强大的工具。即使正则表达式在 HTML 上是有用的，但是未来会出现的正则表达式所不能应付的 HTML，这也是有可能的。为了完美地解决这个问题，这里需要引入一个解析器。来看一个例子：

```
const html = '<br> [!CDATA[[<br>]]';
const matches = html.match(/<br>/ig);
```

在这个例子中，正则表达式会匹配两次，但其实只有一个真正的
标签，另一个匹配的字符串只是一个非 HTML 字符（CDATA）。除此之外，正则表达式在匹配具有等级结构的内容时也会受限（例如，<p>标签内部的<a>标签）。从理论上解释这些限制因素已经远远超出了本书的范围，但要记住一点：如果正在纠结于构建一个正则表达式去匹配非常复杂的东西（例如 HTML），那么就该考虑一下它是否是一个合适的工具。

17.8 字符集

字符集提供了一种简洁的方式，来表示单个字符的分支（稍后会把它与“重复”结合使用，看看它是如何扩展到多字符的）。这样说吧，如果想在一个字符串中查找所有的数字。可以使用分支：

```
const beer99 = "99 bottles of beer on the wall " +
   "take 1 down and pass it around -- " +
   "98 bottles of beer on the wall.";
const matches = beer99.match(/0|1|2|3|4|5|6|7|8|9/g);
```

太冗长了！如果想匹配字母而非数字或数字和字母，或是任何非数字的内容，又该如何呢？此时字符集就派上用场了。简单来说，它们提供了一个更加简洁的方式来表示单个数字的分支。开发人员甚至还能指定范围。来改写之前的例子：

```
const m1 = beer99.match(/[0123456789]/g);     // 可以
const m2 = beer99.match(/[0-9]/g);            // 更好!
```

开发人员还可以组合范围。来看看如何匹配字母、数字和各种标点符号（这个表达式会匹配原始字符串中除了空格之外的所有字符）：

```
const match = beer99.match(/[\-0-9a-z.]/ig);
```

注意，查找时没有先后顺序，也可以写成/[.a-z0-9\-]/。查找时必须对破折号进行转义，否则，JavaScript 会尝试将它翻译成查找范围的一部分（也可以将它放在右方括号之前，那样就不用转义了）

字符集的另一个非常强大的特性是能够取反。对字符集取反是指："匹配这些字符之外的所有内容"。要对一个字符集取反，将插入符号（^）作为字符集的第一个字符即可。

```
const match = beer99.match(/[^\-0-9a-z.]/);
```

这个正则表达式只会匹配原始字符串中的空格（而如果真的想只匹配空格，还有更好的方式，很快就会学到它）。

17.9 具名字符集

有一些字符集很常见而且实用，人们为这些字符集创建了一些方便的缩写，如表 17-1 所示。

表 17-1 具名字符集

具名字符集	等 价 物	注 释
\d	[0-9]	
\D	[^0-9]	
\s	[\t\v\n \r]	包含制表符、空格和垂直制表符
\S	[^ \t\v\n \r]	
\w	[a-zA-Z_]	注意，破折号和句号没有被包含进来，所以它不能用于域名和 CSS 类
\W	[^a-zA-Z_]	

上表中最常用的缩写可能是空格集（\s）。例如，空格通常用于行对齐，但如果想写一段代码来解析它，那么可能希望能够兼顾不同数量的空格：

```
const stuff =
   'hight:      9\n' +
   'medium:     5\n' +
   'low:        2\n';
const levels = stuff.match(/:\s*[0-9]/g);
```

(\s 后面的*表示"零个或多个"，很快会学到这点)。

千万不要忽视反字符集的用途（\D，\S 和\W)，它们用了一种很好的方式来帮助大家摆脱那些不想要的东西。例如，在将电话号码存到数据库之前对其进行标准化，就是一个很不错的想法。因为人们可能会用各种奇怪的方式输入电话号码：破折号、句号、括号和空白。如果只有 10 个数字，那么对它搜索、监控和识别不就方便多了吗？（或者超出 10 位的国际电话)。使用\D 后事情就简单了：

```
const messyPhone = '(505) 555-1515';
const neatPhone = messyPhone.replace(/\D/g, '');
```

同样地，经常使用\S 来确保必填字段中有数据（它们必须至少包含一个非空格的字符)

```
const field = '   something   ';
const valid = /\S/.test(field);
```

17.10 重复

重复元字符允许指定匹配目标重复出现的次数。回想一下之前匹配单个数字的例子。如果要匹配数字（可能由多个连续的数字组成）呢？可以用一个前面提到的方法来实现：

```
const match = beer99.match(/[0-9][0-9][0-9]|[0-9][0-9]|[0-9]/);
```

这里要注意如何在匹配不太特殊的字符串（两位数字）之前，必须先匹配最特殊的字符串（三位数字)。这对一位、两位或三位数字都是可行的，但当需要匹配四位数字时，分支也必须加入对 4 位数字的匹配。幸运的是，有一种更好的写法：

```
const match = beer99.match(/[0-9]+/);
```

注意字符组后面的+：它表示前一个元素应该至少匹配一次。“前一个元素”通常会困扰初学者。重复元字符是一种用来修饰它前面元素的修饰器。它们不会（且不能）单独出现。一共有 5 种重复修饰符，如表 17-2 所示。

表 17-2 重复修饰符

重复修饰符	描　述	示　例
{n}	精确 n 次	/d{5}/ 匹配 5 位数字（例如 zip 码)。
{n, }	至少 n 次	/\d{5,}/ 匹配 5 位或 5 位以上的数字。
{n, m}	最少 n 次，最多 m 次	/\d{2,5}/匹配 2 到 5 位数字。
?	0 或 1 次，等价于{0, 1}	/[a-z]\d?/i 匹配跟随了 0 个或一个数字的字符。

续表

重复修饰符	描　述	示　例
*	0 次或多次（有时称为“Klene 星号”或“Klene 闭包”）	/[a-z]\d*/i 匹配跟随了 0 个或多个数字的字母。
+	1 次或多次	/[a-z]\d+/i 匹配至少跟随一个数字的字母

17.11 句点元字符和转义

在正则表达式中，句点是一个特殊的字符，它表示“匹配任何内容”（除了新的一行）。通常，这个匹配一切的元字符用来消费那些输入中并不关心的内容。来看一个例子，查找一个 5 位数的邮政编码，此时并不关心这一行中的其他内容：

```
    const input = "Address: 333 Main St., Anywhere, NY, 55532. Phone: 555-55
5- 2525.";
    const match = input.match(/\d{5}.*/);
```

大家可能会发现自己经常需要匹配文字中的句点，比如，域名或 IP 地址中的句点。同样地，可能也会经常匹配一些正则表达式的元字符，比如，星号或圆括号，此时需要转义元字符。如果要转义任何特殊字符，只要在前面加一个反斜杠即可：

```
const equation = "(2 + 3.5) * 7";
const match = equation.match(/\(\d \+ \d\.\d\) \* \d/);
```

很多读者可能有使用文件名通配符的经验，或者会使用*.txt 去搜索“所有文本文件”。这里的*是一个“通配符”元字符，它表示可以匹配任何东西。如果很熟悉这种用法，那么正则表达式中*的使用可能会让人觉得迷惑，因为它表示一个完全不一样的功能，并且不能单独使用。在正则表达式中，句点更接近文件名通配符中的*，除非需要匹配的是单个字符，而非整个字符串。

真正的通配符

如果说句点能匹配除了换行之外的任何东西，那么如何匹配包括换行在内的所有字符呢？（这种需求往往比想象中的更容易出现。）它有多种实现方式，但最常用的还是[\s\S]。它能匹配所有空格和非空格。简而言之，它能匹配任何东西。

17.12 分组

到目前为止，所构造的正则表达式能够识别单个字符（重复允许多次匹配，但这依

旧是一个单字符匹配)。而分组则允许构造子表达式，它可以被当做一个独立单元来使用。

除了可以创建子表达式，分组还可以帮助“捕获”分组结果，以便后续使用。“捕获”结果是默认功能，不过也有办法创建“非捕获组“，这也是接下来要学习的内容。如果读者已经有一些使用正则表达式的经验，那这个内容可能并不陌生，不过还是鼓励大家把非捕获组当成默认情况来使用，因为它们有一些性能优势，同时，如果不需要分组的结果，也应该使用非捕获组。分组是使用圆括号来指定的，非捕获组看起来像这样(?:<*subexpression*>)，其中，<*subexpression*>是需要匹配的内容。下面看一些例子，假设现在要匹配后缀为.com、.org 和.edu 的域名：

```
const text = "Visit oreilly.com today!";
const match = text.match(/[a-z]+(?:\.com|\.org|\.edu)/i);
```

分组的另一个好处是可以在分组时使用重复。一般情况下，重复仅被用在重复元字符前面的单个字符上。分组则允许将其用在一整个字符串上。有一个常见的例子是，如果想匹配 URL，以及那些以 http://、https://或//(独立于协议的 URL)开始的 URL，可以在分组上使用代表匹配 0 个或 1 个（?）的重复元字符：

```
const html = '<link rel="stylesheet" href="http://insecure.com/stuff. cs
s">\n' +
      '<link rel="stylesheet" href="https://secure.com/securestuff.css">\n'
 +
      '<link rel="stylesheet" href="//anything.com/flexible.css">';

const matches = html.match(/(?:https?)?\/\/[a-z][a-z0-9-]+[a-z0-9]+/ig);
```

看起来是不是像是一堆乱码？作者也有同感。但这个例子有着很强的代表性，它值得放慢脚步并认真思考。以一个非捕获组开始(?:https?)?，注意这里有两个匹配 0 个或 1 个的重复元字符。第一个表示“*s* 是可选的“。记住，一般情况下，重复元字符只作用于它左边最近的字符。第二个指向它左边的整个组。所以，整体来看，它会匹配空字符串（没有 https)、http 或者 https。继续执行，匹配了两个斜杠（注意，必须对斜杠进行转义：\/)。然后，得到了一个相当复杂的字符类。显然，域名可以包含字母和数字，但它们同样可以包含斜杠（只是它们必须以字母开始，且不能以破折号结尾)。

这个例子并不完美。例如，就像匹配*//valid.com* 一样，它会匹配 *//gotcha*（非顶级域名（TLD))。但退一步说，匹配完整、合法的 URL 是一件非常复杂的任务，没有必要在这个例子中实现。

如果觉得这些说明很烦人（比如，“它会匹配非法的 URL”），那么只要记住一点，使用正则时并不需要一次做完所有的事情。事实上，每当浏览网站的时候，作者都会使用一个与上例非常相似的正则表达式。第一步，先找出所有的 URL 或者疑似 URL 的东西，然后做二次分析，筛选出那些非法的，以及不完整的 URL 等等。不要太纠结于构造一个完美的正则表达式来覆盖所有可能的情况。这样不仅不现实，而且经常会耗费没必要的精力。很显然，有时候也需要考虑所有可能的情况，例如，为了防止注入攻击而检查用户输入。在这种情况下需要额外注意，一定要让正则表达式滴水不漏。

17.13 懒惰匹配，贪婪匹配

区分正则表达式专业开发者和业余爱好者的标志是对懒惰匹配和贪婪匹配的理解。正则表达式默认是贪婪的，这意味着它们会在停止运行前竭尽所能地去匹配。来看一个经典的例子。

有一些 HTML 文本，想将其中的标签替换成标签，下面是第一次尝试：

```
const input = "Regex pros know the difference between\n" +
   "<i>greedy</i> and <i>lazy</i> matching.";
input.replace(/<i>(.*)<\/i>/ig, '<strong>$1</strong>');
```

替换字符串中的$1 会被正则表达式（稍后详述）中的分组（.*）内容替换掉。如果运行这段代码，就会看到让人失望的结果：

```
"Regex pros know the difference between
<strong>greedy</i> and <i>lazy</strong> matching."
```

为了搞清楚这是怎么回事，就得回忆一下正则表达式引擎的工作方式，它只有在找到符合要求的匹配后，才会消费输入并且继续运行。默认情况下，它是通过贪婪模式来实现的，它会找到第一个然后说，“在找到并且确定在这个之后不存在同样的，查找是不会停止的。”因为这里有两个，所以正则会匹配到第二个，而非第一个。

有很多方式可以解决这个问题，但因为是在讨论贪婪匹配和懒惰匹配，所以使用重复元字符（*）将其转换成懒惰匹配来解决它。在后面添加一个问号即可：

```
input.replace(/<i>(.*?)<\/i>/ig, '<strong>$1</strong>');
```

与之前的相比，这个正则表达式除了在*后面添加了一个问号，其他完全一样。现在，正则表达式引擎会这样来思考："一旦我看到一个</i>，查找就会停止。"所以每当碰到</i>时，它就会懒惰地停止匹配，而非继续扫描后面的内容。虽然懒惰这个词一般是负面的，但此时，懒惰正是想要的行为。

所有的重复元字符：*、+、?、{n}、{n,}和{n,m}都可以在其后跟随一个问号将它变成懒惰的（虽然在实践中，通常只把它和*、+一起使用过）。

17.14 反向引用

分组使用了一个叫作*反向引用*的技术。在以往的经验里，这是正则表达式中最不常用的特性之一，但有一种情况使用它很方便。在开始学习真正实用的例子之前，先来看一个非常简单的例子。

假设想匹配符合 XYYX 格式的乐队名字（作者敢打赌读者肯定可以想出一个符合该格式且真实的乐队名字）。所以当希望匹配 PJJP，GOOG 和 ANNA 这些乐队名字时，反向引用就登场了。正则表达式中的每个组（包括子组）都被分配了一个数字，从左到右依次是 1、2、3...，可以通过在反斜杠后加一个数字的方式来引用特定的组。换句话说，\1 表示"#1 组匹配上的所有内容。"有点迷惑了吧？下面来看一个例子：

```
    const promo = "Opening for XAAX is the dynamic GOOG!  At the box office
now!";
    const bands = promo.match(/(?:[A-Z])(?:[A-Z])\2\1/g);
```

从左往右看，可以找到两个组，然后紧跟着的是\2\1。所以，如果第一个组匹配了 X，第二个组匹配了 A，那么\2 和\1 一定分别匹配了 A 和 B。

如果你觉得这只是看起来很酷却没多大作用，其实作者也有同感。因为在以前的经验中，唯一一次需要使用反向引用（除了解决难题）的场景是引号匹配。

在 HTML 中，属性值即可以使用单引号也可以使用双引号，这么一来，就可以简单地这样做：

```
    // 这里使用重音符是因为我们将单引号和双引号都用过了
    const html = '<img alt='A "simple" example.'>' +
            '<img alt="Don't abuse it!"> ';
    const matches = html.match(/<img alt=(?:['"]).*?\1/g);
```

注意，这个例子中做了一些简化工作，如果 alt 属性不在前面，这个表达式就不会生效，如果有多余空格也不生效。我们将在后面重新回顾这个例子的时候看到如何

解决这些问题。

与之前一样，第一个组匹配了单引号或双引号，然后是 0 个或多个字符（注意匹配问号时用了懒惰匹配），接下来是\1—第一个组匹配上的内容，不管是单引号还是双引号。

下面需要花点时间加强对懒惰匹配和贪婪匹配的理解。试试删除*后面的问号，使其变成贪婪匹配，然后再运行这个表达式，会看到什么？大家知道这是为什么吗？如果想精通正则表达式，这是一个必须掌握的重要概念。所以如果还有不清楚的地方，作者希望大家花点时间再回顾一下懒惰匹配和贪婪匹配那部分的内容。

17.15 替换组

分组带来的一个好处是，可以利用它做一些更加复杂的替换。继续看 HTML 的例子，假如想去掉一个<a>标签中除了 href 以外的内容。

```
let html = '<a class="nope" href="/yep">Yep</a>';
html = html.replace(/<a .*?(href=".*?").*?>/, '<a $1>');
```

正如反向引用中讲到的，所有的组都被分配了一个从 1 开始的数字。在这个正则表达式中，通过\1 来引用第一个组；而在替换字符串上，用的是$1。注意，在这个表达式中使用懒惰量词是为了防止它在匹配时跨越多个 <a> 标签。不过，如果<a>标签的 href 属性使用的是单引号而非双引号，也会匹配失败。

下面来扩展这个例子。希望保持 class 属性和 href 属性，仍旧删除其他元素：

```
let html = '<a class="yep" href="/yep" id="nope">Yep</a>';
html = html.replace(/<a .*?(class=".*?").*?(href=".*?").*?>/, '<a $2 $1>
');
```

注意在这个正则表达式中，将 class 和 href 的顺序颠倒了一下，使得 href 始终先出现。这个表达式的问题在于 class 和 href 始终要保持相同的顺序，并且（就像之前提到的）一旦<a>标签中的属性使用了单引号（而非双引号），就会匹配失败。我们将在下一节看到一个更为复杂的解决方案。

除了$1、$2、$3……这些组引用，还有$‘（匹配项之前的所有内容）、$&（匹配目标本身）和$’（匹配项之后的所有内容）。如果想使用一个美元符号，可以使用$$：

```
const input = "One two three";
input.replace(/two/, '($`)');     // "One (One ) three"
input.replace(/\w+/g, '($&)');    // "(One) (two) (three)"
input.replace(/two/, "($')");     // "One ( three) three"
```

```
input.replace(/two/, "($$)");     // "One ($) three"
```

这些替换宏经常会被忽视，但是作者见过它们在一些非常聪明的解决方案中出现，所以不要忘记它们哦。

17.16 函数替换

这是作者最喜欢的正则表达式特性，因为它允许将一个非常复杂的正则表达式拆分成一些简单的正则表达式。

再来看一个实际修改 HTML 元素的例子。假设我们正在编写一段程序来将所有<a>链接转换成一个特定的格式：希望保留 class、id 和 href 属性，并删除其他内容。问题在于，用户输入的 HTML 内容可能是散乱的。而且这些属性也可能不存在，即便存在，也无法保证他们有着相同的顺序。所以，必须考虑这样一些不同的输入（这只是众多不同输入中的一部分）：

```
const html =
   '<a class="foo" href="/foo" id="foo">Foo</a>\n' +
   '<A href='/foo' Class="foo">Foo</a>\n' +
   '<a href="/foo">Foo</a>\n' +
   '<a onclick="javascript:alert('foo!')" href="/foo">Foo</a>';
```

至此大家应该意识到，用正则表达式来实现这个功能是一项非常艰巨的任务：因为存在太多可能的变化！不过，可以通过将表达式拆分成两个，从而大大减少变化的数量：一个用于识别<a>标签，而另一个用于将<a>标签替换成你期望的内容。

先来考虑第二个问题。如果只存在一个单独的<a>标签，而希望移除 class、id 和 href 之外的所有属性，这就简单了。但即便是这样，如果不能保证属性按特定的顺序出现，还是会有问题。虽然有很多方法可以解决这个问题，但这里会使用 String.prototype.split 进行切割，这样就可以一次只考虑一个属性：

```
function sanitizeATag(aTag) {
   // 获取标签...
   const parts = aTag.match(/<a\s+(.*?)>(.*?)<\/a>/i);
   // parts[1] 是<a>标签中的属性
   // parts[2] 是<a>和</a>之间的内容
   const attributes = parts[1]
       // 接下来将其分割成独立的属性
       .split(/\s+/);
   return '<a ' + attributes
      // 过滤掉 class、id 和 href 之外属性
      .filter(attr => /^(?:class|id|href)[\s=]/i.test(attr))
```

```
        // 将它们用空格连接起来
        .join(' ')
        // 关闭<a>标签
        + '>'
        // 添加内容
        + parts[2]
        // 添加闭合标签
        + '</a>';
}
```

这个函数比想象中的要长，不过为了更加清晰，可以将它分成不同的部分。注意即使在这个函数中，依旧使用了多个正则表达式：一个用来匹配<a>标签，一个用来切割（使用一个正则表达式来识别一个或多个空格字符）字符串，还有一个用来过滤期望的属性。如果只用一个正则表达式来完成这些工作将会非常复杂。

接下来这部分就有意思了：在一个可能包含很多<a>标签的 HTML 块中使用 sanitizeATag 函数。编写一个只匹配<a>标签的正则表达式很简单：

```
html.match(/<a .*?>(.*?)<\/a>/ig);
```

但是要怎么使用它呢？在匹配时，可以将函数当做一个替换参数传给 String.prototype.replace。到目前为止，只用过字符串作为替换参数。而使用函数则允许对每一个替换执行一个特定的操作。在完成这个例子之前，用 console.log 来看看它是如何工作的：

```
html.replace(/<a .*?>(.*?)<\/a>/ig, function(m, g1, offset) {
   console.log(`<a> tag found at ${offset}.  contents: ${g1}`);
});
```

传给 String.prototype.replace 的函数会按顺序接收以下参数：

- 整个匹配的字符串（等价于$&）。
- 匹配上的组（如果存在）。有多少个组，这种参数就会有多少个 。
- 原始字符串中的匹配偏移量（一个数字）。
- 原始字符串（很少使用）。

该函数的返回值就是用来替换正则表达式的字符串。在刚才的例子中，没有指定返回值，所以默认返回 undefined，它会被转换成字符串后当做替换字符串使用。这个例子的重点是强调这种工作机制，而非真实的替换，所以这里并没有返回最终结果。现在来回顾一下这个例子，有了能够清理单个<a>标签的函数，以及在 HTML 中查找<a>标签的方法，所以可以将它们结合起来使用：

```
html.replace(/<a .*?<\/a>/ig, function(m) {
   return sanitizeATag(m);
});
```

还可以进一步简化代码，因为 sanitizeATag 函数能够精确匹配 String.prototype.replace 所期望的内容，所以这里可以省去匿名函数，直接使用 sanitizeATag：

```
html.replace(/<a .*?<\/a>/ig, sanitizeATag);
```

希望上面的例子已经讲清楚了这个功能的强大之处 。记住一点，不论什么时候，当需要从一个大字符串中匹配小字符串，并且还要对小字符串做额外处理，都可以通过向 String.prototype.replace 中传入函数来解决这个问题！

17.17 锚点

通常，大家会关心一个字符串的开始和结束，或者整个字符串（而不只是一部分），这时候锚点就派上用场了。有两种锚点，分别是用于匹配行开始的^，以及用于匹配行结束的$。

```
const input = "It was the best of times, it was the worst of times";
const beginning = input.match(/^\w+/g);   // "It"
const end = input.match(/\w+$/g);         // "times"
const everything = input.match(/^.*$/g);  // 跟输入一样
const nomatch1 = input.match(/^best/ig);
const nomatch2 = input.match(/worst$/ig);
```

关于锚点，有个细节需要知道。一般情况下，它们匹配的是整个字符串的开始和末尾，即使字符串中有换行。如果想把某个字符串当做多行字符串（以换行符分隔）来处理，就需要用到 m（多行）选项：

```
const input = "One line\nTwo lines\nThree lines\nFour";
const beginnings = input.match(/^\w+/mg);  // ["One", "Two", "Three", "F
our"]
const endings = input.match(/\w+$/mg);     // ["line", "lines", "lines",
 "Four"]
```

17.18 单词边界匹配

正则表达式中一个经常被忽视，但却非常有用的特性是单词边界匹配。类似开始锚点和行末锚点，单词边界匹配是\b，取反是\B，它不消费输入内容。这是一个非常好用的特性，马上就可以看到如何使用它。

单词边界界定为一个\w 匹配之前或之后紧挨着一个\W（非单词）字符，或字符串的开始或结尾。假设试图将英文中的邮件地址替换成超链接（为了方便解释，假定邮件地址以一个字母开始和一个字母结束）。想想那些需要考虑到的不同情况：

```
const inputs = [
   "john@doe.com",                // nothing but the email
   "john@doe.com is my email",    // email at the beginning
   "my email is john@doe.com",    // email at the end
   "use john@doe.com, my email", // email in the middle, with comma afte
rward
   "my email:john@doe.com.",      // email surrounded with punctuation
];
```

虽然有很多种不同的情况，但是所有这些邮件地址有一个共同点：它们都处在单词的边界。单词边界标记的另一个好处是，因为它们不消费输入，所以不用担心“将它们放回“到替换字符串中：

```
const emailMatcher =
   /\b[a-z][a-z0-9._-]*@[a-z][a-z0-9_-]+\.[a-z]+(?:\.[a-z]+)?\b/ig;
inputs.map(s => s.replace(emailMatcher, '<a href="mailto:$&">$&</a>'));
// returns [
//    "<a href="mailto:john@doe.com">john@doe.com</a>",
//    "<a href="mailto:john@doe.com">john@doe.com</a> is my email",
//    "my email is <a href="mailto:john@doe.com">john@doe.com</a>",
//    "use <a href="mailto:john@doe.com">john@doe.com</a>, my email",
//    "my email:<a href="mailto:john@doe.com>john@doe.com</a>.",
// ]
```

除了单词边界匹配，这个正则表达式还使用了本章中讲到的很多特性：初看时可能会让人望而生畏，但如果能花些时间将它们搞懂，则离成为正则表达式大师更近了（这里要注意特别替换宏，$&，它确实没有包含邮件地址周围的字符，因为那些字符没有被消费）。

当需要搜索以其他单词开始、结束或包含其他单词的文本时，使用单词边界也非常方便。例如，/\bcount/会找到 *count* 和 *countdown*，但不会找到 *discount*、*recount* 或 *accountable*。而/\bcount\B/只能找到 *countdown*，/\Bcount\b/会找到 *discount* 和 *recount*，而/\Bcount\B/只能找到 *accountable*。

17.19 向前查找

如果说熟悉懒惰匹配和贪婪匹配是专业开发者和业余爱好者的分界线，那么使用向前查找则是大师级开发者和专业开发者的分水岭，与锚点和单词边界元字符一样，

它也不消费输入。然而，不同于锚点和单词边界的是，它们是通用的：可以匹配任何子表达式却不消费它。事实上，正如单词边界元字符，向前查找的这种不消费的特性，解决了有时候不得不进行“原封不动”的替换的问题。虽然“原封不动”的替换可能是个不错的技巧，但并不是必须掌握的。只要有内容重复，向前查找就是必须的，而且它们可以简化某些特定类型的匹配。

验证密码是否符合预设的规则是一个经典的例子。简单起见，假设密码必须由字母和数字组成，且至少包含一个大写字母、一个数字和一个小写字母。当然，可以使用多个正则表达式来实现它：

```
function validPassword(p) {
    return /[A-Z]/.test(p) &&          // at least one uppercase letter
        /[0-9]/.test(p) &&             // at least one number
        /[a-z]/.test(p) &&             // at least one lowercase letters
        !/[^a-zA-Z0-9]/.test(p);       // only letters and numbers
}
```

假如想将它们组合成一个正则表达式。第一次尝试将以失败告终：

```
function validPassword(p) {
    return /[A-Z].*[0-9][a-z]/.test(p);
}
```

这个表达式对顺序有要求，它不仅要求大写字母出现在数字之前，数字出现在两个小写字母前，而且没有对非法字符做任何校验。实际上也没有一个好办法可以实现它，因为字符在正则表达式运行时就被消费了。

向前查找通过不消费输入来解决这个问题。本质上每个向前查找都是一个不消费输入的独立正则表达式。在 JavaScript 中，向前查找是这样的（(?=<subexpression>)。还有一个“否定向前查找“：((?!<subexpression>)）只会匹配不存在于子表达式中的内容。下面可以单独写一个正则表达式来验证密码：

```
function validPassword(p) {
    return /(?=.*[A-Z])(?=.*[0-9])(?=.*[a-z])(?!.*[^a-zA-Z0-9])/.test(p);
}
```

看到这些类似乱码的内容，你可能觉得还不如使用多个正则表达式，至少可读性会更高一些。就这个例子而言，作者同意这种看法。这个例子展示了向前检查（以及否定向前检查）的一个非常重要的使用场景。虽然向前检查确实归属于“高级正则表达式”的范畴，但它对于解决某些特定的问题也很重要。

17.20 动态构造正则表达式

本章从一开始，就提倡优先使用正则表达式字面语法而非构造器。除了能少打四个字母，使用字面语法的另一个原因是不用对反斜杠进行转义。真正需要使用构造器的地方是动态构造正则表达式。例如，可能想在一个字符串中匹配一个包含多个用户名的数组，但却没有（好的）办法将这些用户名整合在一个正则表达式字面量中。此时正则表达式构造器就派上用场了，因为它可以通过字符串来构造正则表达式，这样就实现了动态构造。来看一个例子：

```
const users = ["mary", "nick", "arthur", "sam", "yvette"];
const text = "User @arthur started the backup and 15:15, " +
    "and @nick and @yvette restored it at 18:35.";
const userRegex = new RegExp('@(?:${users.join('|')})\\b', 'g');
text.match(userRegex);  // [ "@arthur", "@nick", "@yvette" ]
```

与该例中正则表达式等价的字面量是：`/@(?:mary|nick|arthur|sam|yvette)\b/g`，不过已经成功的将它动态地构造出来了。需要注意的是，必须在 `b`（单词边界元字符）之前使用双反斜杠，第一个反斜杠是用来转义第二个反斜杠的。

17.21 小结

虽然本章已经涉及了正则表达式中的主要知识点，但对于正则表达式中包含的技术、例子和其固有的复杂性，也只是浅尝辄止。然而，要精通正则表达式，需要的是 2 分的理论知识和 8 分的实践操练。另外，一个健壮的正则检验器（例如，正则表达式 101）或许会对后续的学习有所帮助。本章最重要的知识点是理解正则表达式引擎消费输入的方式，如果对此缺乏理解可能会给以后使用正则表达式带来很多阻碍。

第 18 章

浏览器中的 JavaScript

JavaScript 最开始是作为浏览器脚本语言出现的，现在它已经在这个领域占了据垄断地位。本章就是为那些在浏览器中使用 JavaScript 的开发者准备的。语言本身没有什么变化，但是会讲到一些特殊的使用场景，以及针对这些场景的 API。

全面详细地介绍基于浏览器的 JavaScript 开发已经可以写成一本书了。本章的目的只是介绍浏览器开发中一些重要的核心概念，这会为 JavaScript 学习打下一个坚实的基础。在本章最后，会推荐一些附加的学习资料。

18.1 ES5 还是 ES6

希望 ES6 的功能增强带来的好处已经足够让人心悦诚服了。然而现实并不完美，因为 Web 端还需要一段时间才能完全支持强大且一致的 ES6 标准。

在服务端，可以确定地知道哪些 ES6 特性能被支持（假设你能控制 JavaScript 引擎）。但在 Web 端，当把精心编写的代码通过 HTTP(S)发送出去后，这些代码会在那些无法控制的 JavaScript 引擎中运行。更糟的是，你对那些可能被用到的浏览器毫不知情。

当然，这个问题也并非没法解决，“常青树”浏览器就是一个解决方案：通过自动更新（不需征求用户的意见），新的 web 标准可以更快、更一致地在这些浏览器上被推广。然而，这只能减少而不能彻底解决这些问题。

除非可以通过某种途径控制用户的环境，否者在可预见的未来中，只能部署 ES5 代码。不过不要担心，这并不是世界末日：转换编译器依旧可以让开发人员安心的编写 ES6 代码。虽然这会加大开发和调试的难度，但这也是为进步付出的代价。

本章会假设大家都在使用转换编译器，就像在第 2 章中讲过的。本章的所有样例都可以在最新的 Firefox 中运行（不需要使用转换编译器）。如果要将代码对外发布，那就需要对代码进行转换编译，从而确保它能在更多浏览器中稳定地工作。

18.2 文档对象模型

文档对象模型（*Document Object Model*），又称 DOM，是一种描述 HTML 文档结构的约定，它是与浏览器进行交互的核心。

从概念上讲，DOM 是一棵由节点构成的树：每个节点都有一个父节点（除了根节点），以及 0 个或多个子节点。根节点是一个文档，它只有一个子节点，也就是<html>元素。<html>元素有两个子节点：<head>元素和<body>元素（图 18-1 就是一个 DOM 树的例子）。

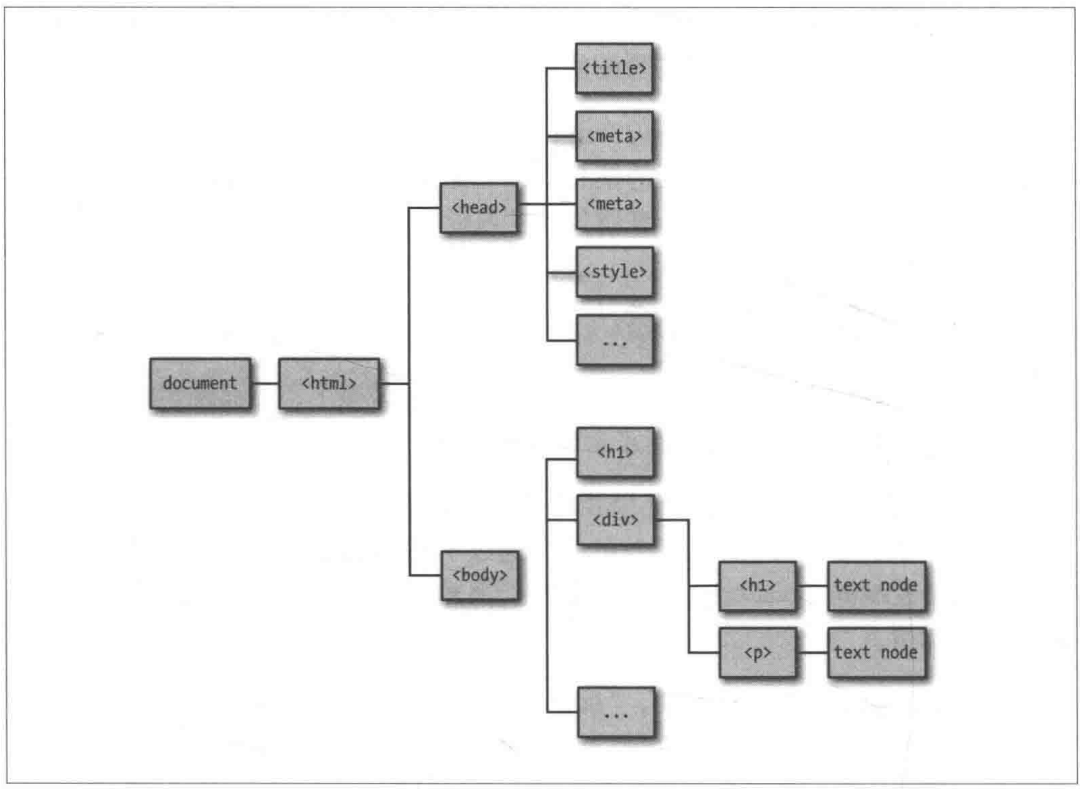


图 18-1 DOM 树

DOM 树（包括文档本身）中的每一个节点都是一个节点类（Node class）（这里的节点跟 Node.js 没有关系，下一章会讲到）的实例。节点（Node）对象拥有父节点

（parentNode）和子节点（childNode）属性，以及像节点名（nodeName）和节点类型（nodeType）这些用来标识节点的属性。

DOM 全部由节点组成，其中只有一部分节点是 HTML 元素。比如，段落标签（<p>）是 HTML 元素，但段落中的文字却是文本节点。节点和元素这两个术语经常会被交叉使用，虽然这样不会引起混淆，但从技术的角度来讲是不对的。本章中，接触到的大多数节点都是 HTML 元素，所以当讲到“元素”的时候，实际上说的是“元素节点。”

在下面的例子中，作者会用一个很简单的 HTML 文件来展示这些特性。首先，创建一个名为 *simple.html* 的文件：

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Simple HTML</title>
        <style>
            .callout {
                border: solid 1px #ff0080;
                margin: 2px 4px;
                padding: 2px 6px;
            }
            .code {
                background: #ccc;
                margin: 1px 2px;
                padding: 1px 4px;
                font-family: monospace;
            }
        </style>
    </head>
    <body>
        <header>
            <h1>Simple HTML</h1>
        </header>
        <div id="content">
            <p>This is a <i>simple</i> HTML file.</p>
            <div class="callout">
                <p>This is as fancy as we'll get!</p>
            </div>
            <p>IDs (such as <span class="code">#content</span>)
                are unique (there can only be one per page).</p>
            <p>Classes (such as <span class="code">.callout</span>)
                can be used on many elements.</p>
            <div id="callout2" class="callout fancy">
                <p>A single HTML element can have multiple classes.</p>
```

```
            </div>
        </div>
    </body>
</html>
```

每个节点都有 nodeType 和 nodeName 的属性（当然还有其他属性）。nodeType 是一个用来标识节点类型的整数，并且在节点对象中都包含了这些数字所对应的常量。本章中主要用到的节点类型是 Node.ELEMENT_NODE（HTML 元素）和 Node.TEXT_NODE（文本内容，一般存在于 HTML 元素中）。如果想了解更多内容，可以参考 MDN 上关于 nodeType 的文档。

下面来看一个颇具教学意义的练习：编写一个函数来实现从 document 开始，横向打印整个 DOM 的功能。

```
function printDOM(node, prefix) {
    console.log(prefix + node.nodeName);
    for(let i=0; i<node.childNodes.length; i++) {
        printDOM(node.childNodes[i], prefix + '\t');
    }
}
printDOM(document, '');
```

这个递归函数实际上是实现了树的*深度优先遍历*，也叫*前序遍历*。也就是说，它会先遍历一个分支中的所有节点，然后才会遍历下一个分支。如果在页面加载的时候运行这段代码，控制台中就会打印整个页面的结构。

然而这只是一个用于教学的练习，如果用这种方式操作 HTML（必须遍历整个 DOM 才能找到需要的元素），不仅过程冗长，而且效率低下。好在 DOM 提供了更为直接的定位 HTML 元素的方式。

自己编写遍历函数固然是一个很好的练习，不过 DOM API 已经提供了用来遍历 DOM 中的所有元素的 TreeWalker 对象（而且还能有选择性地过滤某些类型的元素）。想了解更多内容，可以看看 MDN 上关于 document.createTreeWalker 的文档。

18.3 关于树的专用语

树本身是一个很直观的概念，这使得树这个术语也同样直观。一个节点的父节点指的是它的直接父节点（也就是说，不会是“祖父”），其子节点也是直接子节点（而非“孙子”）。子孙节点指的是子节点、子节点的子节点，等等。而祖先节点则是父节点、父节点的父节点，等等。

18.4 DOM 中的“Get”方法

DOM 提供了“get”方法，可以帮助快速定位指定的 HTML 元素。

第一个方法是 document.getElementById。页面中的每个 HTML 元素都可以指定一个独一无二的 ID，而 document.getElementById 可以通过一个元素的 ID 来获取它。

```
document.getElementById('content');      // <div id="content">...</div>
```

浏览器并不会强制要求给元素指定唯一的 ID（虽然 HTML 检查器会发现这个问题），所以保证每个元素拥有唯一的 ID 是开发人员义不容辞的责任。随着 web 页面结构复杂度的增加（页面中的组件也可能来自多个代码源），避免出现重复 ID 的困难性也随之升高。所以，作者建议大家在使用 ID 时小心一些，节省一些。

document.getElementsByClassName 会返回通过 class 名查找到的元素集合。

```
const callouts = document.getElementsByClassName('callout');
```

而 document.getElementsByTagName 则返回符合给定 tag 名称的元素集合。

```
const paragraphs = document.getElementsByTagName('p');
```

所有返回集合的 DOM 方法返回的都不是一个 JavaScript 数组，而是一个 HTMLCollection 的实例，这是一个“类似数组”的对象。可以在 for 循环中迭代它们，但却不能使用 Array.prototype 方法（同样不能使用的方法还有 map，filter 和 reduce）。通过使用展开操作符（spread operator），可以把一个 HTMLCollection 转化成数组：[...document.getElementsByTag Name(p)]。

18.5 查询 DOM 元素

getElementById,getElementsByClassName 和 getElementsByTagName 这些方法都非常有用，但其实还有更多常用（且强大）的方法可以用来定位元素，它们不仅可以通过单个条件（ID，class，和 name）定位元素，还能通过元素之间的关系来定位元素。document 上的 querySelector 和 querySelectorAll 方法就允许使用 CSS 选择器来定位元素。

CSS 选择器允许通过元素的名字（<p>、 、等）、ID、class（或组合 class）、

或者这些内容的组合来识别元素。要通过名字来定位元素，只需要使用元素的名字（不加尖括号）即可。所以 a 会匹配 DOM 中所有的<a>标签，br 则会匹配所有
标签。而要通过 class 识别元素，只需要在 class 名字前面加一个句点符号：比如.callout 可以匹配所有使用了 callout 这个 class 的元素。要匹配多个 class 时，多个 class 之间也用句点符号来分隔：.callout.fancy 会匹配所有使用了 callout 和 fancy 这两个 class 的元素。最后，还可以把这些选择器组合起来使用，比如，`a#callout2.cal lout.fancy`会匹配所有 ID 为 callout2，且具有 callout 和 fancy class 的<a>标签（一般很少会见到在选择器中既使用元素名字和 ID，又使用 class（多个）的情况，但不排除这种可能性）。

熟悉 CSS 选择器的最好方法是在浏览器中加载本章中的 HTML，打开 console，然后试着用 `querySelectorAll` 来选择元素。比如，可以在 console 中输入 **`document.querySelector.All('.callout')`**。本节中所有的例子都会显示出至少一条查询结果。

到目前为止，我们已经讨论过如何定位指定元素，不论它在 DOM 中的什么位置。而 CSS 选择器也允许通过元素在 DOM 中所处的位置来定位它们。

如果将多个 CSS 选择器通过空格分隔，就能查找到一个具有特定祖先的节点。比如，#content p 会选择所有在 ID 为 content 的元素中的<p>元素。同样，`#content div p` 会选择出所有在 中的<p>元素，而这些 又在 ID 为 `content` 的元素中。

如果将多个元素选择器用大于号（>）分隔，就可以找到那些直接的子节点。例如，`#content>p` 会选出那些 ID 为 content 的元素中作为子节点的<p>元素。（注意和“#content p”区分开）。

注意，也可以把祖先选择器和直接子节点选择器结合起来使用。比如，body .content>p 会从<body>的子节点中选出 class 属性为 content 的所有直接子节点中的 <p>标签。

还有更复杂的选择器，上面提到的都是最常见的。如果想学习更多的选择器用法，参阅 MDN 上关于 CSS 选择器的文档（*http://mzl.la/1Pxcg2f*）。

18.6 多个 DOM 元素

现在大家已经知道如何遍历、获取和查询元素，那么可以对这些元素做些什么呢？就从修改元素内容开始吧。由前面的学习可以知道，每个元素都有 text Content 和

innerHTML 属性，可以用来访问（和修改）元素的内容。textContent 可以去除所有的 HTML 标签，只留下纯文本数据，而 innerHTML 则可以用来创建 HTML 元素（产生一个新的 DOM 节点）。下面来看看如何访问并修改下面例子中的第一个段落：

```
const para1 = document.getElementsByTagName('p')[0];
para1.textContent;     // "This is a simple HTML file."
para1.innerHTML;       // "This is a <i>simple</i> HTML file."
para1.textContent = "Modified HTML file";          // 在浏览器中看看有什么变化
para1.innerHTML = "<i>Modified</i> HTML file";  // 在浏览器中看看有什么变化
```

textContent 和 innerHTML 是一种灾难性的操作：它会替换掉元素中的所有内容，不管里面有什么。例如，可以通过在<body>元素上设置 innerHTML 来替换掉整个页面的内容！

18.7 创建 DOM 元素

大家已经知道如何通过设置元素的 innerHTML 属性来隐式地创建 DOM 节点。其实还可以通过 document.createElement 来显式地创建节点。这个函数可以创建一个新元素，但它并不会将元素添加到 DOM 中，必须再单独做一次添加操作。下面，创建两个段落元素，分别作为 中的第一个元素和第二个元素：

```
const p1 = document.createElement('p');
const p2 = document.createElement('p');
p1.textContent = "I was created dynamically!";
p2.textContent = "I was also created dynamically!";
```

为了把新创建的元素添加到DOM中，这里要用到 insertBefore 和 appendChild 方法。首先，需要获取父元素（ ）和它的第一个子节点：

```
const parent = document.getElementById('content');
const firstChild = parent.childNodes[0];
```

现在可以添加新创建的元素：

```
parent.insertBefore(p1, firstChild);
parent.appendChild(p2);
```

insertBefore 接收两个参数，第一个是要插入的新元素，第二个是“参考节点”，新元素会插入到参考节点之前。appendChild 则很简单，它会把指定元素作为节点的最后一个子节点插入。

18.8 样式元素

通过 DOM API，大家对元素的样式有了彻底且细粒度的掌控。然而，相比于修改元素属性，使用 CSS 类来修改样式通常是一个比较好的实践。也就是说，如果想修改一个元素的样式，只需要创建一个新的 CSS 类，然后把它应用到需要修改样式的元素（可以是多个）上。在 JavaScript 中，把一个 CSS 类应用到元素上非常容易。例如，如果想高亮所有包含单词“*unique*”的段落，可以先创建一个 CSS 类：

```
.highlight {
    background: #ff0;
    font-style: italic;
}
```

接下来，要找到所有<p>标签，如果它们包含单词“*unique*”，就给它们加上 highlight 的 class。HTML 中的每个元素都包含一个叫作 classList 的属性，它包含了该元素的所有 class（如果有的话）。classList 的 add 方法可以给元素上添加新的 class。本章后面还会用到这个例子，所以这里创建一个叫作 highlightParas 的函数来实现功能：

```
function highlightParas(containing) {
    if(typeof containing === 'string')
        containing = new RegExp('\\b${containing}\\b', 'i');
    const paras = document.getElementsByTagName('p');
    console.log(paras);
    for(let p of paras) {
        if(!containing.test(p.textContent)) continue;
        p.classList.add('highlight');
    }
}
highlightParas('unique');
```

如果想删除 highlight 这个类，可以用 classList.remove：

```
function removeParaHighlights() {
    const paras = document.querySelectorAll('p.highlight');
    for(let p of paras) {
        p.classList.remove('highlight');
    }
}
```

在删除 highlight 类时，可以复用 paras 变量，只要在每个段落元素上调用 remote('highlight')即可；如果元素上已经不存在这个 class，该方法就什么都不做。不过，后面很可能会有删除 class 的需要，同时其他代码也可能会添加高亮段落，所以如果代码的目的是删除所有的高亮样式，在删除之前之前先执行查询语句会安全得多（译者注：类似在数据库中执行删除语句，删除前先查询，确保所删内容的正确性）。

18.9 数据属性

HTML5 引入了*数据属性*，它允许在 HTML 元素中加入任意数据，该数据并不是由浏览器产生的，不过它却可以给元素添加一些信息，从而让元素很容易被 JavaScript 读取和修改。下面给 HTML 中添加一个 button，这个 button 的行为会最终绑定到 highlightParas 函数，以及 remoteParaHighlight 函数：

```
<button data-action="highlight" data-containing="unique">
    Highlight paragraphs containing "unique"
</button>
<button data-action="removeHighlights">
    Remove highlights
</button>
```

将数据属性命名为 action 和 contains（当然也可以给它们起别的名字），然后就可以通过 document.querySelectorAll 来查找所有包含"highlight"这个 action 的元素：

```
const highlightActions = document.querySelectorAll('[data-action=
"highlight"]');
```

这里引入了一个新的 CSS 选择器类型。到目前为止，我们已经见过可以匹配指定标签、class 和 ID 的选择器。方括号则允许根据属性来匹配元素，本例中就使用了一个指定的数据属性。

因为只有一个 button，所以这里可以使用 querySelector 来代替 querySelectorAll，不过 querySelectorAll 允许同一个页面上的多个元素触发同一个 action（这很常见：想想在同一个页面中，通过菜单栏、链接或者工具栏共同触发的 action 有哪些）。如果认真观察 highlightActions 其中的一个元素，会发现它有一个 dataset 属性：

```
highlightActions[0].dataset;
```

```
// DOMStringMap { containing: "unique", action: "highlight" }
```

在 DOM API 中，数据是以字符串形式存储的（这个功能隐藏在 DOMStringMap 类中），也就是说不能存储对象。jQuery 扩展了数据属性的功能，它通过提供接口从而允许用对象来存储数据属性，第 19 章将详细介绍它。

也可以通过 JavaScript 添加或修改数据属性。比如，如果想要高亮包含单词“giraffe”的段落，同时指明匹配时要大小写敏感，就可以这么做：

```
highlightActions[0].dataset.containing = "giraffe";
highlightActions[0].dataset.caseSensitive = "true";
```

18.10 事件

DOM API 中包含了大约 200 个事件，而每个浏览器还实现了各自的非标准事件，因此，本书不会介绍所有事件，只会涵盖那些你需要知道的事件。从一个很容易理解的事件开始：点击事件（click）。下面会通过 click 事件将“highlight” button 跟 highlightParas 函数绑定起来：

```
const highlightActions = document.querySelectorAll('[data-action=
"highlight"]');
for(let a of highlightActions) {
    a.addEventListener('click', evt => {
        evt.preventDefault();
        highlightParas(a.dataset.containing);
    });
}

const removeHighlightActions =
    document.querySelectorAll('[data-action="removeHighlights"]');
for(let a of removeHighlightActions) {
    a.addEventListener('click', evt => {
        evt.preventDefault();
        removeParaHighlights();
    });
}
```

每个元素都有一个叫作 addEventListener 的方法，它允许事件发生时指定调用方法。这个函数只接受一个参数：一个 Event 类型的对象。event 对象中包含了所有关于这个事件的信息，而不同类型的事件将会有不同类型的信息。比如，click 事件包含 clientX 和 clientY 属性，它们可以提供点击事件发生时的坐标。另外，target 属性代表触发点击事件的元素。

事件模型是为了让多个处理器同时处理一个事件而设计的。很多事件都有默认处理器。比如，如果用户点击一个<a>链接，浏览器通过加载链接中请求的页面来处理这个事件。如果想阻止这个默认的处理事件，可以在 event 对象上调用 `preventDefault()`。开发人员编写的大部分事件处理器都会调用 `preventDefault()`方法（除非明确知道自己想在默认处理器的基础上做一些额外处理）。

为了让元素高亮，可以调用 `highlightParas` 函数，给它传入 `button` 的 `containing` 这个数据属性的值：这样就可以通过简单地修改 HTML 来改变要查找的文本。

18.11 事件捕获与事件冒泡

由于 HTML 是层级结构，这使得可以在多个地方处理事件。比如，当一个 button 被点击时，不仅 button 本身可以处理这个点击事件，它的父亲、父亲的父亲，也可以处理这个事件。那么问题来了，“这些元素是按照什么顺序来处理事件呢？”

这里有两种可能性。一个是从最外层的祖先节点开始，称为事件捕获（*capturing*）。在本例中，botton 是 的孩子节点，而它又是<body>的孩子节点。因此，<body>最先“捕获”到最终会达到 button 的事件。

另一个选择是从触发事件的元素开始，然后按照 HTML 的层级结构向上走，这样所有的祖先都有机会处理事件。这称作事件冒泡（*bubbling*）。

为了支持这两种方式，在 HTML5 中，事件的传播是从处理器捕获事件开始（从最外层的祖先节点，一直到目标元素），然后才是事件冒泡，也就是从目标元素到最外层的祖先节点。

任何一个处理器都可以通过以下三件事中的一件来影响（或阻止）其他处理器的调用。第一件同时也是最常见的，是已经见过的 `preventDefault`，它可以取消事件。取消的事件会继续传播，但是它们的 `defaultPrevented` 属性会被置成 true。浏览器内置的事件处理器都会遵从 `defaultPrevented` 属性，不执行任何操作。而自己编写的事件处理器则可以（通常也会这么做）选择性地忽略这个属性。第二件事是调用 stopPropagation，时间传播到当前元素之后，将不会继续传播（所有绑定到当前元素上的处理器都会被调用，但绑定到其他元素上的处理器不会被调用）。最后一件事，也是重头戏：调用 `stopImmediatePropagation` 会阻止所有还未调用的处理器被调用（即使这些处理器被绑定在当前元素上）。

为了演示这些功能，我们来看看下面的 HTML：

```
<!doctype html>
<html>
    <head>
        <title>Event Propagation</title>
        <meta charset="utf-8">
    </head>
    <body>
        <div>
            <button>Click Me!</button>
        </div>
        <script>

            // 创建一个事件处理器并返回这个处理器
            function logEvent(handlerName, type, cancel,
                    stop, stopImmediate) {
                // 这才是真正的事件处理器
                return function(evt) {
                    if(cancel) evt.preventDefault();
                    if(stop) evt.stopPropagation();
                    if(stopImmediate) evt.stopImmediatePropagation();
                    console.log(`${type}: ${handlerName}` +
                        (evt.defaultPrevented ? ' (canceled)' : ''));
                }
            }

            // 给元素上添加一个 logger 事件
            function addEventLogger(elt, type, action) {
                const capture = type === 'capture';
                elt.addEventListener('click',
                    logEvent(elt.tagName, type, action==='cancel',
                    action==='stop', action==='stop!'), capture);
            }

            const body = document.querySelector('body');
            const div = document.querySelector('div');
            const button = document.querySelector('button');

            addEventLogger(body, 'capture');
            addEventLogger(body, 'bubble');
            addEventLogger(div, 'capture');
            addEventLogger(div, 'bubble');
            addEventLogger(button, 'capture');
            addEventLogger(button, 'bubble');

        </script>
    </body>
```

```
</html>
```

点击 button，会在 console 中打印以下内容：

```
capture: BODY
capture: DIV
capture: BUTTON
bubble: BUTTON
bubble: DIV
bubble: BODY
```

从这个例子中可以清楚地看到事件传播的顺序：先是事件捕获，然后是事件冒泡。注意，在产生事件的元素上，处理器被调用的顺序就是它们被添加的顺序，不论它们是捕获事件还是传播事件（如果把捕获处理器和冒泡处理器的添加顺序颠倒一下，那冒泡处理器将会在捕获处理器之前被调用）。

下面来看一下如果取消事件传播会发生什么。修改前面的例子，取消 上的捕获事件传播。

```
addEventLogger(body, 'capture');
addEventLogger(body, 'bubble');
addEventLogger(div, 'capture', 'cancel');
addEventLogger(div, 'bubble');
addEventLogger(button, 'capture');
addEventLogger(button, 'bubble');
```

可以看到传播还在继续，但是事件已经被标记为取消：

```
capture: BODY
capture: DIV (canceled)
capture: BUTTON (canceled)
bubble: BUTTON (canceled)
bubble: DIV (canceled)
bubble: BODY (canceled)
```

现在停止<button>上的捕获事件传播：

```
addEventLogger(body, 'capture');
addEventLogger(body, 'bubble');
addEventLogger(div, 'capture', 'cancel');
addEventLogger(div, 'bubble');
addEventLogger(button, 'capture', 'stop');
addEventLogger(button, 'bubble');
```

看，事件在传播到<button>元素后就停止传播了。但是，即使事件捕获先行触发，并且停止了传播，<button>上的冒泡事件依然会触发。不过 和<body>元素却接收不到它们的冒泡事件：

```
capture: BODY
capture: DIV (canceled)
capture: BUTTON (canceled)
bubble: BUTTON (canceled)
```

最后，立即停止<button>上的事件捕获：

```
addEventLogger(body, 'capture');
addEventLogger(body, 'bubble');
addEventLogger(div, 'capture', 'cancel');
addEventLogger(div, 'bubble');
addEventLogger(button, 'capture', 'stop!');
addEventLogger(button, 'bubble');
```

至此可以看到事件传播被<button>捕获之后就完全停止了，传播没有再发生：

```
capture: BODY
capture: DIV (canceled)
capture: BUTTON (canceled)
```

addEventListener 取代了以前用来添加 event 的方法：使用“on”属性添加 event。例如，在元素 elt 上添加一个点击处理器可以用 `elt.onclick = function(evt) {/*handler */}`。用这种方式注册事件处理器的最大缺点是一次只能注册一个处理器。

讲了这么多，并不意味着在开发中需要经常提前知道事件的传播并控制它，事件传播这个话题经常会让初学者觉得困惑。不过，牢牢掌握事件传播机制可以将高级程序员跟一般的开发人员区分开来。

在 jQuery 的事件监听器中，直接在处理器中返回 false 与调用 `stopPropagation` 是等价的，这是 jQuery 的约定，但这种快捷方式不能在 DOM API 中使用。

事件的种类

MDN 上有一个很不错的参考文档，它将所有的 DOM 事件按照种类进行分组。

常用的事件种类包括以下 6 类。

拖拽事件（Drag events）

实现了拖放事件的接口，比如，`drag start`, `drag`, `dragend`, `drop`。

焦点事件（Focus events）

用户与可编辑的元素（比如，表单里的字段）交互时执行的一些操作。当用户“进入”一个字段（通过点击、敲 Tab 键或者触摸）时会触发 focus 事件，用户“离开”字段（通过点击其他地方、敲击 Tab 键或触摸别的地方）时会触发 blur 事件。而当用户修改字段的时候则会触发 `change` 事件。

表单事件（Form events）

当用户提交一个表单（通过敲击 Submit 按钮或在对应的地方敲回车）时，会触发表单上的 submit 事件。

输入设备事件（Input device events）

前面已经学习过 click 了，但其实还有很多别的鼠标事件（`mousedown`、`move`、`mouseup`、`mouseenter`、`mouseleave`、`mouseover`、`mousewheel`）和键盘事件（`keydown`、`keypress`、`keyup`）。注意“触摸”事件（对于可触摸设备而言）是优先于鼠标事件执行的，但如果触摸事件没有对应的处理器，那么他们将会触发鼠标事件。比如，如果用户触摸一个按钮，但是触摸事件并没有显式的处理器，那么此时就会触发一个 `click` 事件。

媒体事件（Media events）

用于追踪用户与 HTML5 中的视频、音频播放器的交互（比如，暂停（`pause`）、播放（`play`）等）。

进度事件（Progress events）

通知浏览器加载网页的进度。最常使用的是 load，它会在浏览器加载元素和所有依赖资源的时候调用一次。另外，`error` 事件也很有用，它允许在元素无法使用（比如，一个无法使用的图标链接）时做一些事情。

触摸事件（Touch events）

触摸事件为可触摸设备提供了高级支持。允许多点触摸（在 event 中看看 touches 属性就知道了）可以处理复杂的触摸事件，比如，收缩、滑动等手势。

18.12 Ajax

Ajax（它来源于“Asynchronous JavaScript and XML”的首字母缩写）允许客户端与服务端进行异步通信：页面上的元素在不加载整个页面的情况下刷新服务器

(server) 端的数据。在本世纪初引入 XMLHttpRequest 对象之后，这个创新才得以实现，从而开启了众所周知的“Web 2.0”时代。

Ajax 的核心概念很简单：运行在浏览器端的 JavaScript 以可编程的的方式将 HTTP request 发送到 server，server 再返回所请求的数据，这种数据通常以 JSON 格式传输（在 JavaScript 中，JSON 格式比 XML 格式更容易使用）。这些数据可以在浏览器上实现更多的功能。Ajax 是基于 HTTP 的（像不使用 Ajax 的页面一样），这样一来，就降低了渲染和传输页面的开销，从而提高了 Web 应用的性能，至少从用户角度来看是这样的。

使用 Ajax 之前，必须得先有一个服务器（*server*）。这里用 Node.js（先预习一下 20 章的内容）写一个最简单的服务器，它会暴露一个 Ajax 的 *endpoint*（一个被其他服务或应用使用的特殊 service）。创建一个叫作 ajaxServer.js 的文件：

```
const http = require('http');

const server = http.createServer(function(req, res) {
    res.setHeader('Content-Type', 'application/json');
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.end(JSON.stringify({
        platform: process.platform,
        nodeVersion: process.version,
        uptime: Math.round(process.uptime()),
    }));
});

const port = 7070;
server.listen(port, function() {
    console.log(`Ajax server started on port ${port}`);
});
```

这段代码创建了一个非常简单的 server，它可以说明当前使用的操作系统（“linux”“darwin”“win32”等）、Node.js 的版本和 server 的启动时间。

Ajax 引入了一个叫作跨资源共享（CORS- cross-origin resource sharing）的安全漏洞。在本例中，在 header 中将 Access-Control- Allow-Origin 的值设为*，这告诉客户端不要因为安全问题而阻止调用。在产品环境上，一般不会使用跟本地环境一样的网络协议、域名和端口号（默认允许访问任何端口），或指明可以访问 endpoint 的协议、域名和端口号。这里为了演示功能，禁用了 CORS 的检查。

运行下面这行命令可以启动 server：

```
$ babel-node ajaxServer.js
```

在浏览器中打开 *http://localhost:7070*，就可以看到 server 的输出。有了一个 server 之后，就可以在 HTML 页面（可以继续沿用本章中一直使用的那个页面）中编写 Ajax 代码了。首先在 body 中添加一个占位符，用来接收信息：

```
<div class="serverInfo">
    Server is running on <span data-replace="platform">???</span>
    with Node <span data-replace="nodeVersion">???</span>.  It has
    been up for <span data-replace="uptime">???</span> seconds.
</div>
```

这样就有了存放从 server 端获取到的数据的地方了，然后就可以使用 XMLHttpRequest 执行 Ajax 调用。在 HMTL 文件的最下面（在闭合标签</body>的前面），添加如下代码：

```
<script type="application/javascript;version=1.8">
    function refreshServerInfo() {
        const req = new XMLHttpRequest();
        req.addEventListener('load', function() {
            // TODO: 将这些值放在 HTML 中
            console.log(this.responseText);
        });
        req.open('GET', 'http://localhost:7070', true);
        req.send();
    }
    refreshServerInfo();
</script>
```

这段脚本执行了一个基本的 Ajax 调用。首先，它创建了一个新的 XMLHttpRequest 对象，接下来添加一个监听器来监听 load 事件（在 Ajax 调用成功的时候会被调用）。现在只是把 server 端的 response（在 this.responseText 中）打印到 console 中。接下来调用 open，才真正建立了跟 server 端的连接。调用 open 的时候明确指明这是一个 HTTP GET 请求，这种方式与在浏览器地址栏上输入网址来打开页面是一样的（还有 POST、DELETE 等方法），open 方法还接收一个 server 端的 URL。最后，调用 send，这才是真正执行 request 的地方。在本例中，并没有显式地向 server 端发送任何数据，虽然可以这么做。

运行这个例子，就能在 console 中看到 server 端返回的数据了。下一个目标是把这些数据插入到 HTML 中。此时需要构建一个 HTML，这样就可以查找那些具有 replace 这个数据属性的元素了，然后将元素中的内容替换成 server 端返回的数据。为了实现这个，将 server 端返回的数据进行迭代（使用 Object.keys），如果其中的内容可以匹配 replace 的数据属性，就执行替换操作：

```
req.addEventListener('load', function() {
    // this.responseText 是一个包含 JSON 的字符串;
    //使用 JSON.parse 将它转换成对象
    const data = JSON.parse(this.responseText);

    // 在本例中，我们只想替换<div>中含有 "serverInfo "这个 class 的元素的文本
    const serverInfo = document.querySelector('.serverInfo');

    // 迭代从 server 端返回的对象中的 key
    // ("platform", "nodeVersion", and "uptime"):
    Object.keys(data).forEach(p => {
        // 查找替换属性的元素（如果有的话）
        const replacements =
            serverInfo.querySelectorAll(`[data-replace="${p}"]`);
        // 将所有元素中的内容替换成从 server 返回的数据
        for(let r of replacements) {
            r.textContent = data[p];

        }
    });
});
```

因为 refreshServerInfo 是一个函数，所以可以在任何时候调用它。特别是需要定期更新 server 端信息（这也是为什么这里添加了一个 uptime 的字段）的时候。比如，希望每秒更新 5 次 server 端的信息（也就是 200 毫秒一次），以下是具体的代码实现：

```
setInterval(refreshServerInfo, 200);
```

通过这段代码，可以在浏览器中看到 server 的启动时间在动态增加！

在这个例子中，页面最开始加载的时候， 包含了一个用问号表示的占位符。在网速很慢的时候，用户可能会在内容被替换成 server 端的数据之前看到这些问号一闪而过。这是“闪现未加载样式内容（FOUC）”问题的变种。一种解决方案是在页面初次加载的时候就把 server 端的数据渲染到页面上。另一种解决方案是在内容被更新前先把元素隐藏起来；这可能还是会让用户觉得不快，但这比出现无意义的问号更容易让用户接受。

这里只介绍了关于发送 Ajax 请求的基本概念，想了解更多的话，可以参考 MDN 的文档“使用 XMLHttpRequest”。

18.13 小结

从本章可以看出，Web 开发引入了大量除 JavaScript 语言本身之外的概念和复杂度。对此本书也只是浅尝辄止，如果读者是一个 Web 开发人员，建议阅读 Semmy Purewal 的《*Learning Web App Development*》（译者注：翻译本书时还未找到这本书的中文版，所以此处用了英文原版书名）。如果还想学一些 CSS，那么 Eric A. Meyer 的每一本书都是绝佳的选择。

第 19 章

jQuery

jQuery 是一个流行的类库，可以用来操作 DOM，以及执行 Ajax 请求。所有 jQuery 能做的事情，DOM API 都可以做（毕竟，jQuery 是基于 DOM API 实现的），不过 jQuery 有三大主要优势：

- jQuery 屏蔽了浏览器对 DOM API 实现的差异性（尤其是老的浏览器）。
- jQuery 提供了更简单的 Ajax API（由于现在的网站中大量使用了 Ajax，所以这一点非常受欢迎）。
- jQuery 在内置的 DOM API 中提供了很多强大且简洁的增强功能。但是 Web 开发者社区中越来越多的人认为，随着 DOM API 和浏览器性能的提升，jQuery 将不再被人们需要。开发者社区推崇高性能的纯“vanilla JavaScript（原生 JavaScript）”。事实上，第一点（浏览器的差异性）已经慢慢地不那么明显了，但还没有完全消失。但作者个人觉得 jQuery 依然有意义，而且用它提供的特性来重写 DOM API 的功能可以节省大量的时间。不论是否使用 jQuery，它的普遍存在性都使开发人员很难完全抛弃它，而明智的 Web 开发人员则会掌握一些 jQuery 的基础知识。

19.1 万能的美元（符号）

jQuery 是第一个将 JavaScript 中的美元符号作为标识符的类库。这个决定最开始看上去有些狂妄自大，但是如今，jQuery 的普遍存在已经证实了这个决定的前瞻性。

将 jQuery 引入到项目中后，就可以使用变量 jQuery 或者是更短的$[①]。本章中将使用别名$。

19.2 引入 jQuery

最简单的做法是通过 CDN 来引入 jQuery：

```
<script src="//code.jquery.com/jquery-2.1.4.min.js"></script>
```

jQuery2.x 不支持 IE6、IE7 和 IE8。如果需要支持这些浏览器，只能使用 jQuery1.x。而正是因为不用支持这些老旧的浏览器，jQuery2.x 才变得异常轻便和小巧。

19.3 等待 DOM 加载

浏览器读取、解析，以及渲染 HTML 文件的方式很复杂，一些粗心的 web 开发人员试图在浏览器加载完 DOM 之前用代码访问 DOM 元素，但结果通常会让他们措手不及。

通过 jQuery，可以将代码放在一个回调函数中，而只有在页面加载和 DOM 构建都完成的时候，才会调用这个回调函数：

```
$(document).ready(function() {
    // 这段代码在所有 HTML 加载完毕以及 DOM 构建完成后才执行
});
```

即使多次使用这个方法也不会有安全隐患，可以将 jQuery 代码放在不同的地方，它仍然会安全地等待 DOM 加载完毕后执行。它还有一个等效的快捷方式：

```
$(function() {
    // 这段代码在所有 HTML 加载完毕以及 DOM 构建完成后才执行
});
```

在使用 jQuery 的时候，普遍做法是把所有代码放入这样的块中。

① 如果 jQuery 跟其他类库产生冲突，我们也可以禁止它使用$；详情请参考 jQuery.noConflict。

19.4 jQuery 封装的 DOM 元素

jQuery 操作 DOM 的主要技巧是使用 *jQuery 封装的 DOM 元素*。任何使用 jQuery 的 DOM 操作，都是从一个“封装”了一系列 DOM 元素集合的 jQuery 对象开始（这里要注意，集合可能为空或只包含一个元素）。

jQuery 函数（$或者 jQuery）创建了一个 jQuery 包装的 DOM 元素集合（后面简称为“jQuery 对象”，但要记住，jQuery 对象持有一个 DOM 元素的集合）。jQuery 函数的主要调用方式有两种：通过 CSS 选择器调用和通过 HTML 调用。

使用 CSS 选择器调用 jQuery 会返回一个与之匹配的 jQuery 对象（类似于 document.querySelectorAll 的返回结果）。例如，要获得一个匹配所有<p>标签的 jQuery 对象，只需要这样做：

```
const $paras = $('p');
$paras.length;                          // 匹配的<p>标签数量
typeof $paras;                          // "object"
$paras instanceof $;                    // true
$paras instanceof jQuery;               // true
```

另一方面，使用 HTML 调用 jQuery 时，会根据提供的 HTML 创建一个新的 DOM 元素（类似于为某个元素设置 innerHTML 属性时所做的事情）

```
const $newPara = $('<p>Newly created paragraph...</p>');
```

大家应该已经注意到，在这两个例子中，赋给 jQuery 对象的变量都以美元符号开头。虽然这不是必须的，但却是一个好的编程习惯。因为有了美元符号，就能迅速在代码中识别出 jQuery 对象。

19.5 操作元素

现在已经有了一些 jQuery 对象，可以用它们做什么呢？使用 jQuery 可以很容易的添加和移除元素。熟悉这些例子的最好方式就是在浏览器中加载示例中的 HTML，并在控制台中执行它们。准备好不断重新加载文件吧！接下来会有很多移除、添加和修改内容的操作。

jQuery 提供的 text 和 html 方法分别等同于对 DOM 元素的 textContent 和 innerHTML 属性进行赋值。例如，将每个段落替换成相同的文本：

```
$('p').text('ALL PARAGRAPHS REPLACED');
```

同样地，可以调用 html 方法来使用 HTML 的内容：

```
$('p').html('<i>ALL</i> PARAGRAPHS REPLACED');
```

这里蕴含了 jQuery 的一个重要知识点：jQuery 可以很容易地一次性操作多个元素。虽然在使用 DOM API 时，document.querySelectorAll()也会返回多个元素，但是依然要手动遍历它们，才能执行一些操作。而 jQuery 会处理所有的迭代工作，它会默认假定要对 jQuery 对象中的每个元素都执行操作。那么，如果只想修改第三个段落怎么办？jQuery 提供了 eq 方法，通过它可以获得一个包含单个元素的新的 jQuery 对象：

```
$('p')              // 匹配所有段落
    .eq(2)          // 第三个段落 (下标从 0 开始)
    .html('<i>THIRD</i> PARAGRAPH REPLACED');
```

要移除元素，调用 jQuery 对象的 remove 方法即可。下面的代码可以移除所有段落：

```
$('p').remove();
```

上例演示了 jQuery 开发中另一个重要的范式：*链式调用*。由于所有 jQuery 方法都返回了一个 jQuery 对象，这样一来，就可以像刚才那样进行*链式调用*。链式调用是一种非常强大且简洁的多元素操作方式。

jQuery 提供了很多添加内容的方法。其中一个是 append，它只是简单地往 jQuery 对象中的每个元素追加所提供的内容。比如，如果想给每个段落添加脚注，只要这样做：

```
$('p')
    .append('<sup>*</sup>');
```

append 会给所有匹配的元素添加一个子节点，还可以使用 before 或 after 来插入兄弟节点。下面看一个例子，它给每个段落的前、后分别添加<hr>：

```
$('p')
    .after('<hr>')
    .before('<hr>');
```

与这些插入方法对应，appendTo、insertBefore 和 insertAfter 这些方法颠倒了插入顺序，它们在一些特定的场景中非常有用，来看个例子：

```
$('<sup>*</sup>').appendTo('p');  // 等同于 $('p').append('<sup>*</sup>')
$('<hr>').insertBefore('p');      // 等同于$('p').before('<hr>')
$('<hr>').insertAfter('p');       // 等同于$('p').after('<hr>');
```

jQuery 也可以很容易的修改元素的样式。比如，使用 addClass 添加一个 class、

removeClass 可以移除 class，或者用 toggleClass（如果元素当前不包含指定 class，就添加，反之移除）来切换 class。另外，可以直接使用 css 方法来操作样式。后续还会介绍:even 和:odd 选择器，可以用来间隔地选择元素。例如，如果想让段落间隔地变成红色，可以这么做：

```
$('p:odd').css('color', 'red');
```

在 jQuery 链中，有时候需要从匹配的元素中选出一个子集。前面已经知道使用 eq 方法可以将 jQuery 对象缩减至一个元素，不过还可以使用 filter、not 和 find 修改选中的元素。filter 会将集合缩减至符合选择器的元素。例如，可以先在调用链中修改每个段落，再用 filter 将段落间隔地变成红色：

```
$('p')
    .after('<hr>')
    .append('<sup>*</sup>')
    .filter(':odd')
    .css('color', 'red');
```

not 实际上是反选 filter。例如，如果想在每个段落之后添加一个<hr>，然后给那些不包含 highlight 这个 class 的段落加上缩进：

```
$('p')
    .after('<hr>')
    .not('.highlight')
    .css('margin-left', '20px');
```

最后，find 会返回一个符合查询条件并且包含所有子元素的元素集合（不像 filter 会过滤现有的集合）。例如，想在每个段落之前添加一个<hr>，然后再将具有 code 这个 class 的元素字体放大（在这个例子中，这些元素是段落的子节点）：

```
$('p')
    .before('<hr>')
    .find('.code')
    .css('font-size', '30px');
```

19.6 展开 jQuery 对象

如果需要“展开”一个 jQuery 对象（访问底层的 DOM 元素），可以使用 get 方法。比如，为了获取第二个段落的 DOM 元素，可以这么做：

```
const para2 = $('p').get(1);        // 第二个<p>（小标从零开始）
```

获得一个包含所有段落的 DOM 元素的数组：

```
const paras = $('p').get();         // 包含所有<p>元素的数组
```

19.7 Ajax

jQuery 提供了一些可以简化 Ajax 调用的便捷方法。它暴露了一个名为 ajax 的方法，该方法允许通过 Ajax 调用来实现复杂的控制逻辑。另外，它还提供了方便的 get 和 post 方法，这两个方法涵盖了最常用的 Ajax 调用类型。虽然这些方法支持回调，不过它们也会返回 promise，而用 promise 来处理服务器响应是一种比较推荐的方式。例如，可以使用 get 来重写之前的 refreshServerInfo 例子：

```
function refreshServerInfo() {
    const $serverInfo = $('.serverInfo');
    $.get('http://localhost:7070').then(
        // 成功返回
        function(data) {
            Object.keys(data).forEach(p => {
                $('[data-replace="${p}"]').text(data[p]);
            });
        },
        function(jqXHR, textStatus, err) {
            console.error(err);
            $serverInfo.addClass('error')
                .html('Error connecting to server.');
        }
    );
}
```

如大家所见，用 jQuery 大大简化了 Ajax 代码。

19.8 小结

jQuery 的未来不是很明朗。JavaScript 和浏览器 API 的提升会导致 jQuery 被淘汰吗？坚持“vanilla JavaScript”的纯粹主义者会被证明是正确的吗？这些问题只有时间会给出答案。而作者个人觉得 jQuery 在可预见的未来中仍然有用而且意义重大。因为就目前而言，jQuery 依旧有着相当高的使用率，另外，任何具有极客精神的开发人员都最起码会掌握关于 jQuery 的基本知识。

如果想学习更多有关 jQuery 的知识，推荐阅读 Earle Castledine 和 Craig Sharkie 写的书《*jQuery:菜鸟到忍者*》。另外，在线 jQuery 文档也是一个不错的学习资料。

第 20 章

Node

直到 2009 年，JavaScript 还只是一门浏览器端脚本语言[①]。2009 年，在被服务器端各种状态折磨得筋疲力尽的情况下，Joyent 公司的一位名叫 Ryan Dahl 的开发人员发明了 Node。此后，Node 迅速地被采用，甚至在这个众所周知接受新事物速度极慢的企业市场中获得了巨大的成功。

对于喜欢 JavaScript 的人来说，Node 能够完成以前只能由其他语言完成的任务。对于网站开发人员来说，它的吸引力比 JavaScript 这门语言更为强烈。在 server 端写 JavaScript，能够保持编程语言的一致性——再也不用煞费苦心的切换上下文，减少了对特定语言专家的依赖，另外（可能也是最重要的），它使得同一段代码在服务器端和客户端都能运行。

最初引入 Node 的目的是为了网页应用开发，后来它却无意间进入了后端开发，从而让一些非传统开发也能够使用它，比如，桌面应用开发和系统脚本处理。从某种意义上说，是 Node 让 JavaScript 成长起来，并加入全栈开发的行列。

20.1 Node 基础

如果会写 JavaScript，那么就能写 Node。这并不是说可以直接把基于浏览器的 JavaScript 程序运行在 Node 上：因为浏览器端的 JavaScript 使用了浏览器专用的 API。而且，Node 中没有 DOM（这也说得通：因为没有 HTML）。同样，浏览器中也没有针对 Node 的 API。还有一些对操作系统及文件系统的支持，也因为安全

① 在 Node 出现之前，也有尝试将 JavaScript 在服务器端使用的先例。尤其值得一提的是，网景企业服务器早在 1995 就支持了服务端 JavaScript。然而，直到 Node 出现之前，服务器端 JavaScript 都没有获得更多的支持。

原因不能在浏览器中使用（大家能想象黑客通过浏览器删除个人电脑上文件的情形吗？）。其他像创建 Web 服务器这样的操作，在浏览器上也没有多大作用。

在 Node 中，区分哪些是 *JavaScript* 代码，哪些是 *API* 非常重要。经常编写浏览器端 JavaScript 的开发人员可能会想当然地认为 window 和 document 都属于 JavaScript。但其实它们都是浏览器提供的 *API*（在 18 章中讲过）。本章将会讨论 Node 所提供的 API。

开始本章的学习之前，请确保计算机上安装了 Node 和 npm（详情见第 2 章）。

20.2 模块（Module）

模块（*Module*）是一种打包和命名空间的机制。命名空间（Namespace）可以有效避免命名冲突。比如，如果 Amanda 和 Tyler 都写了一个名为 `calculate` 的函数，把这些函数复制粘贴到程序中，那么第二个函数就会覆盖第一个函数。而命名空间则允许分别引用“Amanda 的 calculate”和“Tyler 的 calculate”。下面看看如何通过 Node module 来解决这个问题。首先创建一个名为 amanda.js 的文件：

```
function calculate(a, x, n) {
    if(x === 1) return a*n;
    return a*(1 - Math.pow(x, n))/(1 - x);
}

module.exports = calculate;
```

添加 *tyler.js* 文件：

```
function calculate(r) {
    return 4/3*Math.PI*Math.pow(r, 3);
}

module.exports = calculate;
```

看到这些代码后，可以得出一个合理的结论：Amanda 和 Tyler 在命名上都很懒，他们的函数都没什么特点，但是为了讲解这个例子，就纵容他们一回吧。这些文件中都有一行很重要的代码：`modules.export = calculate.module`，它是 Node 中一个可以实现模块化的特殊对象。给 exports 这个属性赋的值，将会暴露到该 module 之外。前面已经写了一些 module 了，下面来看看如何在第三个程序中使用它们。创建一个名为 app.js 的文件，并引入这些 module：

```
const amanda_calculate = require('./amanda.js');
const tyler_calculate = require('./tyler.js');
```

```
console.log(amanda_calculate(1, 2, 5));   // logs 31
console.log(tyler_calculate(2));          // logs 33.510321638291124
```

注意，这里随便起了几个名字（amanda_calculate 和 tyler_calculate），因为它们只是变量，它们的值就是 Node 执行完 require 函数之后的结果。

有数学背景的读者可能已经发现这两个 calculate 函数的不同了：Amanda 的函数计算了一个等比数列的和：$a+ax+ax^2+\cdots+axn^{-1}$，而 Tyler 则提供了一个根据球体半径 r 来计算体积的算法。知道了函数的用法后，就可以跟 Amanda 和 Tyler 的不良命名说再见了，然后在 *app.js* 中给它们一个合适的名字：

```
const geometricSum = require('./amanda.js');
const sphereVolume = require('./tyler.js');

console.log(geometricSum(1, 2, 5));     // logs 31
console.log(sphereVolume(2));           // logs 33.510321638291124
```

module 可以对外暴露任何类型的值（甚至是基本类型，虽然没有什么理由去这么做）。一般情况下，开发人员会希望自己的 module 包含多个函数，但是大多数情况下，这样会暴露一个带有函数属性的对象。想象一下如果 Amanda 是一个代数学家，那么除了等比数列求和之外，他还能提供很多有用的代数函数：

```
module.exports = {
    geometricSum(a, x, n) {
        if(x === 1) return a*n;
        return a*(1 - Math.pow(x, n))/(1 - x);
    },
    arithmeticSum(n) {
        return (n + 1)*n/2;
    },
    quadraticFormula(a, b, c) {
        const D = Math.sqrt(b*b - 4*a*c);
        return [(-b + D)/(2*a), (-b - D)/(2*a)];
    },
};
```

讲到这里，已经越来越接近传统的命名空间了，开发人员给返回值命名，而返回值（一个对象）通常会包含自己的名字：

```
const amanda = require('./amanda.js');
console.log(amanda.geometricSum(1, 2, 5));          // logs 31
console.log(amanda.quadraticFormula(1, 2, -15));    // logs [ 3, -5 ]
```

这并不是魔法：module 只暴露包含函数属性的原始对象（不要被缩写的 ES6 语法所迷惑，他们只是一个函数）。这种范例太常见了，所以又有一个关于它的快捷语

法：一个叫作 exports 的特殊变量。可以用一种更加简洁（但是是等价的）的方式重写 Amanda 的 exports：

```
exports.geometricSum = function(a, x, n) {
    if(x === 1) return a*n;
    return a*(1 - Math.pow(x, n))/(1 - x);
};

exports.arithmeticSum = function(n) {
    return (n + 1)*n/2;
};

exports.quadraticFormula = function(a, b, c) {
    const D = Math.sqrt(b*b - 4*a*c);
    return [(-b + D)/(2*a), (-b - D)/(2*a)];
};
```

exports 的快捷语法只能导出对象，如果想导出函数或者其他值，必须使用 module.exports。此外，这两个不能混用：只能二选其一。

20.3 核心模块、文件模块和 npm 模块

模块分为三大类，核心模块（*core modules*）、文件模块（*file modules*）和 *npm* 模块。（npm modules）核心模块的模块名都是 Node 的保留字，比如，fs 和 os（本章后面会详细介绍）。前面已经见过文件模块了：把创建好的文件赋给 module.exports，然后再引用该文件。npm 模块只是一些放在一个名为 *node_modules* 的特殊目录下的文件模块。在使用 require 函数时，Node 会根据传入的字符串来决定模块的类型（如表 20-1）。

表 20-1 模块类型

类　型	传到 require 的字符串	例　子
核心模块	不以 /, ./ 或 ../ 开始	require('fs') require('os') require('http') require('child_process')
文件模块	以 /, ./ 或 ../ 开始	require('./debug.js') require('/full/path/to/module.js') require('../a.js') require('../../a.js')
npm 模块	不是一个核心 module，也不以/,./或../开始	require('debug') require('express') require('chalk') require('koa') require('q')

有一些像 process 和 buffer 这样的全局（global）核心模块，在 Node 程序中始终都是可用的，不用再显式声明 require 语句。表 20-2 中列出了所有的全局模块。

表 20-2　　全局模块

模　块	是否全局	描　述
assert	No	用于测试目的
buffer	Yes	用于操作输入/输出（IO）（主要是文件和网络）
child_process	No	运行外部程序（Node 或者其他）的函数
cluster	No	允许利用多进程所带来的性能
crypto	No	内建的密码库
dns	No	针对网络名字解析的域名系统（DNS）函数
domain	No	允许对 I/O 和其他异步操作进行分组来隔离错误
events	No	支持异步事件的实用工具
fs	No	操作文件系统
http	No	HTTP server 和相关工具
https	No	HTTPS server 和相关工具
net	No	基于 socket 的异步网络 API
os	No	操作系统工具
path	No	文件系统路径名工具
punycode	No	使用 ASCII 码的子集对 Unicode 进行编码
querystring	No	解析和构建 URL 的查询字符串
readline	No	I/O 交互功能，主要用于命令行编程
smalloc	No	允许显式分配缓存
stream	Yes	基于流的数据传递
string_decoder	No	将缓存转换成字符串
tls	No	传输层安全（Transport Layer Security - TLS）通信工具
tty	No	低端电传打字机（Low-level TeleTYpewriter -TTY）工具
dgram	No	用户数据协议（User Datagram Protocol -UDP）网络工具
url	Yes	URL 解析 工具
util	No	Node 内部实用工具
vm	No	（JavaScript）虚拟机：能够在其中进行元编程并创建上下文
zlib	No	压缩工具

讨论 Node 中的所有模块已经超出了本书的范围（本章只会讨论那些最重要的模

块），但是有了这个列表，就可以查找到更多的信息。关于这些 module 的细节可以参考 Node API 文档。

最后，还有 *npm* 模块。npm 模块是一些有特殊命名规范的文件模块。如果需要模块 x（x 不是核心模块），Node 会在当前目录下找一个叫作 *node_modules* 的子目录。如果找到了，就在那个目录下找 x。如果没找到，Node 就会继续在它的上层目录找 *node_modules*，然后一直重复这个过程，直到找到模块或者到达根目录。比如，如果项目目录是/home/jdoe/test_project，在应用文件中，调用了 require('x')，Node 会依次在以下这些目录中查找模块 x（按照先后顺序）：

- /home/jdoe/test_project/node_modules/x。
- /home/jdoe/node_modules/x。
- /home/node_modules/x。
- /node_modules/x。

大多数项目中都只有一个 node_modules 目录，它一般在项目的根目录下。更进一步说，不应该在这个目录中手动添加或删除任何内容，而应该让 npm 完成这些繁重的任务。不过，了解如何在 Node 中解析并引入模块仍然很有用，尤其是需要在第三方模块中 debug 的时候。

对于那些自己编写的模块，不要把它们放进 *node_modules* 中。虽然这样也行，但是 node_modules 的作用就在于，这个目录可以由 npm 根据 package.json 中的 dependency 列表随时删除和重建（见第 2 章）。

当然也可以发布自己的 npm 模块，并用 npm 来管理它，但还是要避免直接修改 *node_modules* 目录下的内容。

20.4 自定义函数模块

最常见的模块是对外暴露一个对象，有时候也会是单个函数。另一个很常见的模式是：一个模块暴露了一个会立即被调用的函数。该函数的返回值（返回值也可能是个函数）会被使用（换句话说，开发人员并不使用模块返回的函数，而是使用函数调用后的返回值）。当需要对模块进行一定程度上的自定义，或者模块需要接受当前执行上下文中的信息的时候，通常会使用这种模式。下面来看一个真正的 npm 包：debug。在引入 debug 时，它会接受一个 string 参数，用来作 log 前缀，以便区分程序中不同地方的 log。它的用法是：

```
const debug = require('debug')('main'); // 注意这里我们立即调用模块
                                        //返回的函数

debug("starting");                      // 在 debug 模式下会打印出
                                        // "main starting +0ms"
```

为了使用 debug 库中的 debug 功能，设置一个叫作 DEBUG 的环境变量。在上例中，让 DEBUG=main。也可以让 DEBUG=*，从而让所有 debug 消息都变得可用。

从这个例中可以清楚地看到，debug 模块返回了一个函数（因为立即将它当做一个函数去调用了），而这个函数又返回了一个能“记住”上一个函数中字符串的函数。其实，已经把那个值“烙”在模块中了。下面来看看如何实现自己的 debug 模块：

```
let lastMessage;

module.exports = function(prefix) {
    return function(message) {
        const now = Date.now();
        const sinceLastMessage = now - (lastMessage || now);
        console.log('${prefix} ${message} +${sinceLastMessage}ms');
        lastMessage = now;
    }
}
```

这个模块输出了一个函数，它被设计成立即调用，所以 prefix 的值可以被合并到模块中。注意这里还有另一个值，lastMessage，这是打印上一条消息时的时间戳，它可以用来计算消息之间的时间差。

这里就引出了一个非常重要的问题：多次引入模块的话会怎样呢？想象一下，如果把自己写的 debug module 引入两次会怎样呢？

```
const debug1 = require('./debug')('one');
const debug2 = require('./debug')('two');

debug1('started first debugger!')
debug2('started second debugger!')

setTimeout(function() {
    debug1('after some time...');
    debug2('what happens?');
}, 200);
```

大家期望的输出可能是：

```
one started first debugger! +0ms
two started second debugger! +0ms
one after some time... +200ms
two what happens? +200ms
```

但实际上的输出是（毫秒数可能会略有偏差）：

```
one started first debugger! +0ms
two started second debugger! +0ms
one after some time... +200ms
two what happens? +0ms
```

这就证明，在每个 Node 应用启动的时候，Node 只会引入模块一次。所以即使引入了两次 debug 模块，Node 还是会“记住”已经引入过一次，然后使用上次引入的实例。因此，即使 debug1 和 debug2 是两个独立的函数，他们依然共享相同的 lastMessage 引用。

Node 的这种行为是一种安全且合适的行为。因为考虑到性能、内存使用和可维护性等因素，每个模块最好只被导入一次。

其实编写自定义 debug 模块的方式与 npm 中的同名模块方式一样。不过，如果真的需要包含独立时间的多个 debug log 时，可以把 lastMessage 时间戳挪到模块返回函数的函数体中，那样的话，在每次创建 logger 的时候它都能接收到一个全新且独立的值。

20.5 访问文件系统

很多入门级的编程书都会涵盖文件系统的访问，因为它被认为是“标准”程序设计中一个非常重要的部分。而对于可怜的 JavaScript，直到 Node 的出现才让它加入了可以操作文件系统的行列。

本章中的例子都假设项目根目录是*/home/<jdoe>/fs*，这是一个典型的 Unix 系统（将*<jdoe>*替换成开发人员自己设定的用户名）。这个原则在 Window 操作系统同样适用（项目根目录可能是 *C:\Users\<John Doe>\Documents\fs*）。

使用 `fs.writeFile` 可以创建一个文件。在根目录下创建一个叫作 write.js 的文件：

```
const fs = require('fs');

fs.writeFile('hello.txt', 'hello from Node!', function(err) {
    if(err) return console.log('Error writing to file.');
```

```
});
```

这段代码会在当前目录下创建一个文件（假设在这个目录下有足够的权限，同时这个目录下不存在一个名为 *hello.txt* 的目录或只读文件）。无论何时运行一个 Node 应用，它的当前工作目录都会继承自运行时所在的目录（可能会与文件所在目录不一样）。比如：

```
$ cd /home/jdoe/fs
$ node write.js                          # 当前工作目录是/home/jdoe/fs
                                         # 创建 /home/jdoe/fs/hello.txt

$ cd ..                                  # 当前工作目录是 /home/jdoe
$ node fs/write.js                       # 创建/home/jdoe/hello.txt
```

Node 提供了一个特殊的变量，`__dirname`，它总是会把目录指定为源文件所在的目录。比如，可以这样修改代码：

```
const fs = require('fs');

fs.writeFile(__dirname + '/hello.txt',
        'hello from Node!', function(err) {
    if(err) return console.error('Error writing to file.');
});
```

现在，write.js 将始终在/home/<jdoe>/fs（也就是 write.js 所在的目录）目录下创建 hello.txt。使用字符串连接把__dirname 和文件名拼接在一起并不是一个好的做法，因为这样做就使得路径名与运行平台有关联了，比如，这个路径名在 Windows 上就会有问题。Node 的 path 模块提供了平台无关的路径名，所以可以重写这个模块，让它在所有平台上都能正常工作：

```
const fs = require('fs');
const path = require('path');
fs.writeFile(path.join(__dirname, 'hello.txt'),
        'hello from Node!', function(err) {
    if(err) return console.error('Error writing to file.');
});
```

path.join 会把文件路径用平台可识别的连接符连接起来，这通常都是一个好的实践。

如果还想读取文件内容呢？可以用 `fs.readFile`。创建 read.js 文件：

```
const fs = require('fs');
const path = require('path');

fs.readFile(path.join(__dirname, 'hello.txt'), function(err, data) {
```

```
    if(err) return console.error('Error reading file.');
    console.log('Read file contents:');
    console.log(data);
});
```

运行这段代码后，大家可能会被结果吓到：

```
Read file contents:
<Buffer 68 65 6c 6c 6f 20 66 72 6f 6d 20 4e 6f 64 65 21>
```

如果把这些十六进制码转化成对应的 ASCII/Unicode，会发现打出来了 hello from Node!，但程序本身的输出看起来却不太友好。如果不给 fs.readFile 指定解码格式，它只会输出含有二进制数据的 *buffer*。即使不给 *write.js* 指定编码格式，它也会用默认的 UTF-8（一种统一编码方式）。可以通过修改 *read.txt* 来指定解码格式为 UTF-8，从而得到想要的输出：

```
const fs = require('fs');
const path = require('path');

fs.readFile(path.join(__dirname, 'hello.txt'),
        { encoding: 'utf8' }, function(err, data) {
    if(err) return console.error('Error reading file.');
    console.log('File contents:');
    console.log(data);
});
```

fs 模块中所有的函数都有与其等价的同步执行方式（以“Sync”结尾）。在 *write.js* 中，与它等价的是：

```
fs.writeFileSync(path.join(__dirname, 'hello.txt'), 'hello from Node!');
```

而在 *read.js* 中则是：

```
const data = fs.readFileSync(path.join(__dirname, 'hello.txt'),
    { encoding: 'utf8' });
```

有了这些同步执行的方式，就可以通过 exception 来处理错误了，为了让代码更加健壮，可以把代码块用 try/catch 块包起来。比如：

```
try {
    fs.writeFileSync(path.join(__dirname, 'hello.txt'), 'hello from Node!');
} catch(err) {
    console.error('Error writing file.');
}
```

同步的文件系统函数特别好用。不过，如果在编写一个 webserver 或者网络应用程序，一定要知道，Node 的性能来源于异步执行；在这种情况下一定要使用异步的方式。当然，如果在编写命令行应用，使用同步方式一般不会有问题。

可以通过 `fs.readdir` 来列出当前目录下的所有文件。创建一个名为 ls.js 的文件：

```
const fs = require('fs');

fs.readdir(__dirname, function(err, files) {
    if(err) return console.error('Unable to read directory contents');
    console.log(`Contents of ${__dirname}:`);
    console.log(files.map(f => '\t' + f).join('\n'));
});
```

fs 模块包含了很多关于文件系统的函数。通过这些函数可以删除（fs.unlink）、移动、重命名（fs.rename）文件，以及获取文件信息和目录（fs.stat）。想了解更多内容，可以参阅 Node API 文档（*https://nodejs.org/api/fs.html*）。

20.6 进程

每个正在运行的 Node 程序都能访问一个名为 process 的变量，通过这个变量能够获取程序的相关信息，同时控制程序的执行。比如，如果应用程序在运行时出现了严重的错误，那么继续运行（通常被称为重大错误—*fatal error*）显然不明智，也没有意义，这时就能用 proces.exit 立刻中止程序。也可以提供一个用数字表示的错误码，在脚本中通过这个数字就可以知道程序是否正常退出。按照惯例，0 代表“没有错误”，而其他数字则对应不同的代码错误。下面来看一个例子，有一个会处理 *data* 子目录中.txt 文件的脚本：如果没有文件可处理，那就什么都不做，程序会立即退出，但这并不是错误。另一方面，如果 *data* 子目录不存在，就意味着出现了一个很严重的问题，程序应该立即报错并退出。所以代码可能就是这个样子的：

```
const fs = require('fs');

fs.readdir('data', function(err, files) {
    if(err) {
        console.error("Fatal error: couldn't read data directory.");
        process.exit(1);
    }
    const txtFiles = files.filter(f => /\.txt$/i.test(f));
    if(txtFiles.length === 0) {
        console.log("No .txt files to process.");
```

```
        process.exit(0);
    }
    // 处理后缀为.txt 的文件
});
```

使用 process 对象还可以访问包含了传入到程序中的命令行参数的数组。在执行 Node 应用程序时，可以选择传入一些命令行参数。比如，可以写一段程序，它将从命令行接收多个文件名作为参数，然后打印出每个文件有多少行。可以这样来执行它：

```
$ node linecount.js file1.txt file2.txt file3.txt
```

命令行参数会保存在一个叫作 process.argv 的数组[①]中。在开始计算行数前，可以先打印出 process.argv，这样就能明确地看到传入的参数了：

```
console.log(process.argv);
```

连同 *le1.txt*、*le2.txt*、*le3.txt* 的输出，会发现在数组开头有一些额外的元素：

```
[ 'node',
  '/home/jdoe/linecount.js',
  'file1.txt',
  'file2.txt',
  'file3.txt' ]
```

其中，第一个元素是解释器，或者解释源文件的程序（在本例中是 node）。第二个元素是所执行脚本的完整路径，剩下的内容则是传入程序中的参数。因为不需要这些额外信息，所以可以在执行程序前，用 Array.slice 去掉这些多余的信息：

```
const fs = require('fs');

const filenames = process.argv.slice(2);
let counts = filenames.map(f => {
    try {
        const data = fs.readFileSync(f, { encoding: 'utf8' });
        return `${f}: ${data.split('\n').length}`;
    } catch(err) {
        return `${f}: couldn't read file`;
    }
});

console.log(counts.join('\n'));
```

process 还可以通过 process.env 访问环境变量。环境变量是以系统变量命名的变量，它们主要被用在命令行编程中。在大多数 Unix 系统中，可以简单地通过

① argv 这个名字是为了向 C 语言致敬。v 代表 vector（矢量），跟数组类似。

export VAR_NAME=*some value* 来设置环境变量（环境变量的传统写法都是使用大写）。在 Windows 上，可以用 VAR_NAME=some value。环境变量通常用来配置一些应用程序的行为（这样就不用在每次执行程序的时候从命令行传入了）。

例如，大家可能想用环境变量来控制是否打印 debug 的 log，还是“只让程序安静的运行”。通过一个叫作 DEBUG 的环境变量就可以控制这个行为，当 DEBUG 设为 1 的时候，就表示希望 debug（其他值表示不用 debug）：

```
const debug = process.env.DEBUG === "1" ?
    console.log :
    function() {};

debug("Visible only if environment variable DEBUG is set!");
```

在这个例子中，创建一个名为 debug 的函数，如果将 DEBUG 设为 1，那这个函数就是 console.log 的别名，如果 DEBUG 是其他值，那这个函数就等价于一个空函数：什么也没做（如果 debug 是 undefined，就在调用到它的时候报错！）。

在上一章节中，我们提到默认的是当前工作目录，就是执行程序的目录（而非程序所在的目录）。process.cwd 可以说明当前工作目录是什么，而 process.chdir 则可以修改它。比如，如果想要打印程序开始运行时的目录，然后将当前工作目录切换成程序所在目录，可以这样做：

```
console.log(`Current directory: ${process.cwd()}`);
process.chdir(__dirname);
console.log(`New current directory: ${process.cwd()}`);
```

20.7 操作系统

os 模块提供了一些关于 app 所运行的计算机平台相关的信息。下面的例子展示了 os 暴露出来的最有用的信息（这些值是我的云开发机器上的一些参数）：

```
const os = require('os');

console.log("Hostname: " + os.hostname());          // prometheus
console.log("OS type: " + os.type());               // Linux
console.log("OS platform: " + os.platform());       // linux
console.log("OS release: " + os.release());         // 3.13.0-52-generic
console.log("OS uptime: " +
    (os.uptime()/60/60/24).toFixed(1) + " days");   // 80.3 days
console.log("CPU architecture: " + os.arch());      // x64
console.log("Number of CPUs: " + os.cpus().length);// 1
console.log("Total memory: " +
    (os.totalmem()/1e6).toFixed(1) + " MB");        // 1042.3 MB
```

```
console.log("Free memory: " +
    (os.freemem()/1e6).toFixed(1) + " MB");          // 195.8 MB
```

20.8 子进程

child_process 模块可以让应用执行其他程序，不论是 Node 程序、可执行程序、还是由其他语言编写的脚本。讨论管理子进程的所有细节已经超出了本书的范围，不过这里还是会用一个简单的例子来演示子进程的用法。

child_process 模块对外主要暴露了三个函数：exec、execFile、和 fork。与 fs 一样，这三个函数都有各自的同步版本（execSync、execFileSync 和 forkSync）。exec 和 execFile 可以运行任何操作系统支持的可执行程序。exec 会调用一个 *shell*（它位于操作系统命令行底层，如果可以通过命令行来运行一个程序，那么 exec 也能运行它）。execFile 允许直接运行可执行程序，这种方式能略微优化内存和资源的使用，但用的时候要格外小心。最后，fork 允许执行另一个 Node 脚本（这也通过 exec 来完成）。

fork 会启动一个独立的 Node 引擎，所以它的资源开销与 exec 一样。不过，fork 可以进行一些进程间通信的设置。详情见官方文档（*http://bit.ly/1PxcnL9*）。

由于 exec 是最常见的，也最好用，所以本章会一直使用它。

为了演示这个功能，下例中会执行 dir 命令，它可以显示一个目录列表（熟悉 Unix 的读者可能对 ls 更熟悉，在大部分 Unix 系统中，dir 是 ls 的别名）：

```
const exec = require('child_process').exec;

exec('dir', function(err, stdout, stderr) {
    if(err) return console.error('Error executing "dir"');
    stdout = stdout.toString(); // 把 Buffer 转化成 string
    console.log(stdout);
    stderr = stderr.toString();
    if(stderr !== '') {
        console.error('error:');
        console.error(stderr);
    }
});
```

因为 exec 会调用 shell，所以不需要提供 dir 这个可执行文件的路径。如果需要调用一个不能从系统 shell 中直接访问的特殊程序，就需要提供可执行文件的全路径。

被调用的 callback 会接收两个 Buffer 对象，stdout（程序的正常输出）和 stderr（错误输出，如果有的话）。在本例中，由于并不想要输出任何内容到 stderr，所以在打印输出之前先检查一下是否有错。

exec 会接收一个可选的 options 对象，这可以由开发人员指定一些诸如工作目录、环境变量等信息。详情参见官方文档。

注意看这里是如何引入 exec 的。此处没有使用 `const child_process=require('child_process')` 来引入 child_process，然后以 `child_process.exec` 来调用 exec，而是直接给 exec 定义了别名。两种方法都能用，但是上面所用的方法更为常见。

20.9 流

在 Node 中，流是一个很重要的概念。顾名思义，流是用来处理数据的对象（流这个词可能让人联想到流动（*flow*），由于流动是随着时间的推移而发生的，所以说它是异步的也讲得通）。

流可以是读取流、写入流或者二者结合（统称为流）。只有当数据的流动是随着时间持续发生的，流才有意义。相关的例子可以是敲击键盘事件，或者是一个跟客户端反复交互的 web 服务器，又或者是文件访问。这些情况下都会用到流（即使现在已经不需要流就可以对文件进行读写了）。接下来会用文件流来演示如何创建一个读取流或者写入流，以及如何通过管道将流连接起来。

第一步，先创建一个写入流，并给它写入一些内容：

```
const ws = fs.createWriteStream('stream.txt', { encoding: 'utf8' });
ws.write('line 1\n');
ws.write('line 2\n');
ws.end();
```

end 方法有时会接收一个数据类型的参数，这种时候与调用 write 是等价的。因此，如果只需要发送一次数据，就可以简单地调用 end，把想要传送的数据作为参数传给它。

写入流（ws）可以通过 write 方法进行写入操作，直到调用 end 方法，然后流会关闭，在这之后所有调用 write 方法的地方都会出错。不过，在调用 end 之前，可以随意调用 write 方法，写入流是在一段固定时间内写入数据的理想工具。

类似地，可以创建一个读取流，在需要的时候读取数据：

```
const rs = fs.createReadStream('stream.txt', { encoding: 'utf8' });
rs.on('data', function(data) {
    console.log('>> data: ' + data.replace('\n', '\\n'));
});
rs.on('end', function(data) {
    console.log('>> end');
});
```

在本例中，只是将文件内容打印到控制台中（出于整齐的目的，这里将空行都替换掉）。如果把这两个例子放到同一个文件中：那么就有了一个写入流来写文件，还有一个读取流来读文件。

双向流并不常见，而且也超出了本书的范围。可能就像大家所预期的那样，可以调用 write 方法来给双向流中写数据，也可以监听双向流上的 data 和 end 事件。

由于数据会通过流来“流动”，所以，如果从一个读取流中读出数据，然后立即将数据写进一个写入流，原理上也是可行的。这个过程叫作管道（*piping*）。比如，可以通过管道将读取流中的内容放入写入流，从而完成文件复制：

```
const rs = fs.createReadStream('stream.txt');
const ws = fs.createWriteStream('stream_copy.txt');
rs.pipe(ws);
```

注意本例中不需要特殊编码：因为 rs 只是从 *stream.txt* 文件中将字节码通过管道传输到 ws 中（也就是写入 *stream_copy.txt* 文件中）；只有试图解释数据的时候才需要编码。

管道传输对于流动的数据来说是一个很常见的技术。比如，可以将文件内容通过管道转换成一个 webserver 的 response。或者可以将压缩过的数据传入解压引擎，解压引擎最终会将数据传输到文件写入流中。

20.10 Web 服务器

虽然现在很多应用都在使用 Node，但是它最初的目的其实是提供一个 Web 服务器。要是连这个用法都漏掉的话，那也太大意了。

如果读者曾经配置过 Apache、IIS 或者其他 Web 服务器，可能会惊讶于用 Node 创建一个可运行 Web 服务器是如此简单。http 模块（以及它的安全版本，https 模块）暴露了一个 createServer 方法，它可以创建一个基本的 Web 服务器。开发人员要做的仅仅是提供一个处理 request 的回调函数。在使用时，调用 server 的 listen 方法并

指定一个端口，就能启动服务器。

```
const http = require('http');

const server = http.createServer(function(req, res) {
    console.log(`${req.method} ${req.url}`);
    res.end('Hello world!');
});

const port = 8080;
server.listen(port, function() {
    // 可以传入一个 callback 来监听 server 的启动
    console.log(`server startd on port ${port}`);
});
```

出于安全因素，大多数浏览器都不会让开发人员在没有权限的情况下监听默认的 HTTP 端口（80）。事实上，如果需要监听任何 1024 以下的端口，都需要相应的权限。当然获取权限也很容易：如果有 sudo 权限，可以直接用 sudo 来运行 server，从而得到较高的权限，然后监听 80 端口（只要 80 端口没有被其他进程占用）。出于开发和测试的目的，一般来说，监听 1024 以上的端口比较常见。比如，3000、8000、3030 或者 8080 都是常见的端口，之所以常用是因为这些端口很容易记住。

运行这段代码，然后在浏览器访问 *http://localhost:8080*，就能看到 Hello world!。在 console 中打出了所有的 request，它们是由一个方法（有时候也称为动作）和一个 URL 路径构成。大家可能会觉得奇怪，为什么每次访问那个 URL 都会出现两个 request：

```
GET /
GET /favicon.ico
```

大多数浏览器在访问 URL 的时候都会请求一个 icon，用来显示在 URL 输出框或者浏览器标签中。浏览器会隐式地做这件事，所以才能在 console 中看到这条 log。

Node web 服务器的核心是开发人员提供的 callback 函数，它会处理所有服务器接收到的 request。它接收两个参数，一个 IncomingRequest 对象（经常简写为 req）和一个 ServerRequest 对象（经常简写为 res）。IncomingRequest 对象包含了所有与 HTTP request 相关的信息：请求的 URL，以及可能存在的 headers、body 中发送的数据，等等。ServerRequest 对象包含了会被发送到客户端（通常是浏览器）的 response 中的属性和方法。这里调用了 req.end，如果想知道 req 是否为写入流，看看 class 头部就行了。ServerResponse 对象实现了一个写入流的接口，用来定义如何向客户

端写数据。由于 ServerResponse 对象是一个写入流，所以用它来发送文件很简单，可以直接创建一个文件读取流，然后通过管道把它传输到 HTTP response 中。比如，如果有可以让网页变得更好看的 *favicon.ico* 文件，那么就能察觉到这个 request，并且直接发送对应的文件：

```
const server = http.createServer(function(req, res) {
    if(req.method === 'GET' && req.url === '/favicon.ico') {
        const fs = require('fs');
        fs.createReadStream('favicon.ico');
        fs.pipe(res);        // 这行代码代替了调用 'end'
    } else {
        console.log('${req.method} ${req.url}');
        res.end('Hello world!');
    }
});
```

这是一个最小的 web 服务器，虽然看起来很无聊。通过 `IncomingRequest` 中的信息，可以扩展这个模型，从而创建任何想要的网站。

如果在用 Node 为网站提供服务，可能想了解一些有用的框架，比如，Express（*http://expressjs.com/*）或 Koa（*http://koajs.com/*），从而避免从头搭建 web 服务器这种苦差事。

Express 是一个非常流行的框架，Koa 是它的继承者，这不是巧合：因为他们都是由 TJ Holowaychuk 编写的。如果读者已经熟悉了 Express，那么使用 Koa 时就会得心应手。除此之外，还会享受到更趋近于 ES6 的 web 开发过程。

20.11 小结

本章粗略地介绍了大部分有用的 Node API。同时也详细介绍了在应用程序中最常见的 API（比如，fs、Buffer、process 和 stream），除此之外，还有很多需要学习的 API。官方文档全面地将这些内容综合了起来，不过由于文档太过复杂，可能会让初学者望而却步。如果读者对 Node 开发感兴趣，那么 Shelley Power 的《Node 学习指南》会是一个很好的开始。

第 21 章

对象属性配置和代理

21.1 存取器属性：getter 和 setter

JavaScript 中有两种对象属性：数据属性和存取器属性。其实前面已经见过这两种属性了，只不过存取器属性被隐藏在一些 ES6 的语法背后了（在第 9 章，称之为“动态属性”）。

前面大家已经熟悉了函数属性（或方法），存取器属性也是类似的，只不过有两个函数——getter 和 setter。当被访问的时候，它们的行为更像是一个数据属性而非函数。

先来回顾一下动态属性。假设有一个 User 类，它有 setEmail 和 getEmail 两个方法。这里之所以使用“get”和“set”方法，而没有直接把 email 当做属性，是因为这里不希望用户拿到一个不合法的邮箱。User 类非常简单（简单起见，将任何包含了@符号的字符串都当做合法邮箱）：

```
const USER_EMAIL = Symbol();
class User {
    setEmail(value) {
        if(!/@/.test(value)) throw new Error('invalid email: ${value}');
        this[USER_EMAIL] = value;
    }
    getEmail() {
        return this[USER_EMAIL];
    }
}
```

在这个例子中，唯一引人注目的是使用了两个方法（而不是一个属性）来防止 USER_EMAIL 属性接收非法的邮件地址。这里使用符号属性是为了防止意外地直

接访问属性（如果有一个名为 email 或者是_email 字符串属性，就很可能会不小心直接访问到它）。

这个模式很常见，而且很有效，不过它可能比大家所期望的笨重一些。下面是使用了这个类的一个例子：

```
const u = new User();
u.setEmail("john@doe.com");
console.log('User email: ${u.getEmail()}');
```

虽然这段代码可以正常工作，不过使用下面这种写法会更自然：

```
const u = new User();
u.email = "john@doe.com";
console.log('User email: ${u.email}');
```

这就是存取器属性的优势：它能让开发人员在使用后者的自然语法时，还能保持前者的优点。试着用存取器属性重写 User 类：

```
const USER_EMAIL = Symbol();
class User {
    set email(value) {
        if(!/@/.test(value)) throw new Error('invalid email: ${value}');
        this[USER_EMAIL] = value;
    }
    get email() {
        return this[USER_EMAIL];
    }
}
```

这里有两个不同的函数，不过它们都跟单独的 email 属性绑定在一起了。该属性被赋值的时候，*setter* 函数会被调用（所赋的值作为 *setter* 的第一个参数），而 *getter* 函数被调用时，就说明有地方正在获取该属性的值。

这里也可以只提供一个 getter，而不提供 setter，例如有一个返回矩形周长的 getter：

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }
    get perimeter() {
        return this.width*2 + this.height*2;
    }
}
```

此处没有提供设置周长的 setter 函数，因为很难从周长去推断矩形的宽。这么一来，

将其设置为只读属性会更容易理解。

同样，也可以只提供 setter 而不提供 getter，虽然这种模式很少见。

21.2 对象属性的属性

此时，大家已经拥有了丰富的对象属性经验。大家知道它是以键值对的形式存在（键可以是字符串或者符号，值可以是任意类型）；知道对象中各属性的顺序是无法保证的（数组或 Map 则是有序的）；也知道两种访问对象属性的方式（成员访问符（.），以及计算机成员访问符（[]））。最后，了解了三种使用对象字面符号来创建属性的方式（键作为标识符的常规属性、键是非标识符或符号可计算属性，以及方法快捷方式）。

不过，还有很多关于属性的知识需要了解。尤其是，它可以有一些自己的属性来控制其在所属对象中的行为。下面用刚才学到的知识创建一个字段（译者注：原文中是 property，原翻译为“属性”，为避免“属性的属性”引起混淆，所以此处翻译为“字段”），然后通过 `Object.getOwnPropertyDescriptor` 来测试它具备的属性：

```
const obj = { foo: "bar" };
Object.getOwnPropertyDescriptor(obj, 'foo');
```

返回结果如下：

```
{ value: "bar", writable: true, enumerable: true, configurable: true }
```

字段的属性、属性描述符，以及属性配置这些术语可以互相替换使用，因为它们指的是相同内容。

以上信息暴露了这个字段的三个属性：

可写的

控制该字段的值是否可变。

可枚举的

当枚举对象中的属性时，控制该字段是否被包含在其中（使用 `for...in`、`Object.keys` 或者展开运算符）。

可配置的

控制该字段是否可以从对象中删除，或者字段的属性是否可以被更改。

可以使用 Object.defineProperty 来控制字段的属性，它可以创建一个新的字段，或者更改已有的字段（只要该字段是可配置的）。

例如，如果想让 obj 中的 foo 属性变成只读的，就可以使用 Object.defineProperty：

```
Object.defineProperty(obj, 'foo', { writable: false });
```

此时，如果试图给 foo 赋值，程序就会报错：

```
obj.foo = 3;
// TypeError: Cannot assign to read only property 'foo' of [object Object]
```

在严格模式下试图给一个只读字段赋值会引发错误。在非严格模式下，虽然赋值会失败，但程序不会报错。

还可以使用 Object.defineProperty 给对象添加新字段。这对有属性的字段来说格外有用，不同于数据属性，没有其他办法能在对象被创建后向其添加存取器属性。下面来给 obj 添加一个 color 属性（简单起见，这次没有使用符号，也省略了校验）。

```
Object.defineProperty(obj, 'color', {
    get: function() { return this.color; },
    set: function(value) { this.color = value; },
});
```

要创建一个数据属性，将 value 属性传入方法 Object.defineProperty 即可。下面给 obj 添加 name 和 greet 属性：

```
Object.defineProperty(obj, 'name', {
    value: 'Cynthia',
});
Object.defineProperty(obj, 'greet', {
    value: function() { return 'Hello, my name is ${this.name}!'; }
});
```

Object.defineProperty 的一个常见用法是在数组中将属性设为不可枚举的。前面已经提到过，在数组中使用字符串属性或符号属性不是一个明智的做法（因为它跟数组的用法是相反的），不过如果在使用时认真一些，并且考虑周全，它还是很有用的。虽然在数组中使用 for...in 或者 Object.keys 时也会让人灰心丧气（除非使用 for、for...of 或者 Array.prototype.forEach），而也没法阻止别

人这么做。所以，如果在数组中添加了非数值属性，那么应该将其设置为不可枚举的，从而防止有人（不听劝地）使用 `for..in` 或者 `Object.keys`。下面来看一个往数组中添加 `sum` 和 `avg` 方法的例子：

```
const arr = [3, 1.5, 9, 2, 5.2];
arr.sum = function() { return this.reduce((a, x) => a+x); }
arr.avg = function() { return this.sum()/this.length; }
Object.defineProperty(arr, 'sum', { enumerable: false });
Object.defineProperty(arr, 'avg', { enumerable: false });
```

也可以每次在添加属性的时候设置：

```
const arr = [3, 1.5, 9, 2, 5.2];
Object.defineProperty(arr, 'sum', {
    value: function() { return this.reduce((a, x) => a+x); },
    enumerable: false
});
Object.defineProperty(arr, 'avg', {
    value: function() { return this.sum()/this.length; },
    enumerable: false
});
```

最后，还有一个方法：Object.defineProperties（注意这是复数的），它接收一个将属性名和其定义映射起来的对象。可以这样重写之前的例子：

```
const arr = [3, 1.5, 9, 2, 5.2];
Object.defineProperties(arr,
    sum: {
        value: function() { return this.reduce((a, x) => a+x); },
        enumerable: false
    },
    avg: {
        value: function() { return this.sum()/this.length; },
        enumerable: false
    }
);
```

21.3 对象保护：冻结、封装、以及阻止扩展

JavaScript 的灵活性非常强大，但它也可能会带来麻烦。因为任何地方的任何代码都可以用任意方式修改一个对象，所以稍不留神就会写出危险甚至是带有恶意攻击的代码。

为了防止对象被无意间修改，JavaScript 提供了三种机制（且增加了蓄意修改的难度）：冻结、封装，以及阻止扩展。

冻结可以阻止对象的任何修改。一旦对象被冻结，以下操作都是不允许的：

- 给属性设值。
- 调用更改属性值的方法。
- 调用 setter（setter 可以更改对象的属性值）。
- 添加新属性。
- 添加新方法。
- 更改已有属性或方法的配置。

从本质上讲，冻结对象会让对象不可变。它对只包含数据的对象最为实用，因为在冻结包含方法的对象时，会导致一些修改对象状态的无用方法产生。

使用 Object.freeze（可以用 `Object.isFrozen` 来判断该对象是否被冻结）可以冻结一个对象。例如，假设有一个用来存储程序中固定信息的对象（例如，公司、版本、构建 ID，以及获取版权信息的方法）：

```
const appInfo = {
    company: 'White Knight Software, Inc.',
    version: '1.3.5',
    buildId: '0a995448-ead4-4a8b-b050-9c9083279ea2',
    // 这个函数只访问了属性，所以并不会受冻结的影响
    copyright() {
        return `© ${new Date().getFullYear()}, ${this.company}`;
    },
};
Object.freeze(appInfo);
Object.isFrozen(appInfo);   // true

appInfo.newProp = 'test';
// TypeError: Can't add property newProp, object is not extensible

delete appInfo.company;
// TypeError: Cannot delete property 'company' of [object Object]

appInfo.company = 'test';
// TypeError: Cannot assign to read-only property 'company' of [object O
bject]

Object.defineProperty(appInfo, 'company', { enumerable: false });
// TypeError: Cannot redefine property: company
```

封装对象能够阻止在对象上添加属性、重新配置属性，以及移除已有属性。封装可以用在类的实例上，这时操作对象属性的方法仍然有效（只要它们没有试图重新配置某

个属性)。可以使用 Object.seal 封装一个对象，用 Object.isSealed 判断对象是否被封装：

```
class Logger {
    constructor(name) {
        this.name = name;
        this.log = [];
        }
        add(entry) {
            this.log.push({
                log: entry,
                timestamp: Date.now(),
            });
        }
    }

const log = new Logger("Captain's Log");
Object.seal(log);
Object.isSealed(log);   // true

log.name = "Captain's Boring Log";                  // OK
log.add("Another boring day at sea....");           // OK

log.newProp = 'test';
// TypeError: Can't add property newProp, object is not extensible

log.name = 'test';      // OK

delete log.name;
// TypeError: Cannot delete property 'name' of [object Object]

Object.defineProperty(log, 'log', { enumerable: false });
// TypeError: Cannot redefine property: log
```

最后，最脆弱的保护是阻止对象被扩展，不过这也只能防止给类上添加新属性。开发人员仍然可以复制、删除或重新配置属性。继续使用之前的 Logger 类来演示 Object.preventExtensions 和 Object.isExtensible 这两个方法的用途：

```
const log2 = new Logger("First Mate's Log");
Object.preventExtensions(log2);
Object.isExtensible(log2);   // true

log2.name = "First Mate's Boring Log";              // OK
log2.add("Another boring day at sea....");          // OK

log2.newProp = 'test';
// TypeError: Can't add property newProp, object is not extensible

log2.name = 'test';                                 // OK
```

```
delete log2.name;                                    // OK
Object.defineProperty(log2, 'log',
    { enumerable: false });                          // OK
```

其实作者本人很少使用 Object.preventExtensions。假如要阻止某个对象被扩展，通常也会阻止它被删除和重新配置，因此，经常会选择封装这个对象。表 21-1 所示为对象保护选项。

表 21-1　　对象保护选项

操　作	常规对象	冻结对象	密封对象	不可扩展的对象
添加属性	允许	禁止	禁止	禁止
读取属性	允许	允许	允许	允许
设置属性值	允许	禁止	允许	允许
重新配置属性	允许	禁止	禁止	允许
删除属性	允许	禁止	禁止	允许

21.4　代理

代理是 ES6 的新特性，它提供了额外的元编程功能（元编程是指程序自我修改的能力）。

对象代理本质上是指拦截和（可选地）修改该对象上的操作的能力。以一个简单的例子开始：修改属性访问权限。下面看一个包含一些属性的常规对象：

```
const coefficients = {
    a: 1,
    b: 2,
    c: 5,
};
```

假设该对象的属性表示一个数学方程的系数。那么可能会这样使用它：

```
function evaluate(x, c) {
    return c.a + c.b * x + c.c * Math.pow(x, 2);
}
```

到目前为止一切都还正常，至此，可以把一个二次方程的系数存储在对象中，然后计算该方程的值(x 可以是任何数字)。不过，如果传入的对象缺少一个系数会怎么样呢?

```
const coefficients = {
    a: 1,
    c: 3,
};
```

```
evaluate(5, coefficients);        // NaN
```

可以通过将 coefficients.b 设置为 0 来解决这个问题，不过代理为开发人员提供了更好的方案。因为代理可以拦截操作，从而确保未定义的属性返回 0。下列代码给 coefficients 创建一个代理对象：

```
const betterCoefficients = new Proxy(coefficients, {
    get(target, key) {
        return target[key] || 0;
    },
});
```

在作者写这本书的时候，Babel 还不支持代理。不过最新版的 Firefox 已经开始支持代理了，所以可以用 Firefox 来测试这些示例代码。

代理构造方法的第一个参数是要代理的目标，或者被代理的对象。第二参数是处理器，它可以指定要拦截的动作。在这种情况下，只拦截了 get 方法（要把它和 get 属性访问器区分开：它对一般的属性和 get 访问器都有效）表示的属性访问。get 函数接收三个参数（只用了两个）：它们分别是目标、属性的 key（字符串或符号），以及接收者（代理自身，或某个继承于代理的东西）。

在这个例子中，只检查了目标对象中是否包含指定的 key，如果没有，就返回 0。运行代码后的结果如下：

```
betterCoefficients.a;                    // 1
betterCoefficients.b;                    // 0
betterCoefficients.c;                    // 3
betterCoefficients.d;                    // 0
betterCoefficients.anything;             // 0
```

实际上已经给 `coefficients` 对象创建了一个代理，这个代理使 `coefficients` 对象有无穷多的数值属性（都为 0，除了定义的属性之外）！

还可以进一步修改代理，使它只处理单个小写字母为 key 的属性：

```
const betterCoefficients = new Proxy(coefficients, {
    get(target, key) {
        if(!/^[a-z]$/.test(key)) return target[key];
        return target[key] || 0;
    },
});
```

与其简单地校验 `target[key]` 是否为真，还不如在结果不是数字时，直接返回 0。这个实现就留做读者练习吧。

同样地，可以使用 set 处理器来拦截被设置的属性（或访问器）。比如，有一个对象，其中包含了一些危险的属性。而开发人员不希望外界直接设置这些属性，或者调用某些方法，除非有额外的设置。使用的额外设置是一个叫作 allowDangerousOperations 的属性，要先将它设置为 true，才能访问这些危险的属性。

```
const cook = {
    name: "Walt",
    redPhosphorus: 100,          // 危险
    water: 500,                  // 安全
};
const protectedCook = new Proxy(cook, {
    set(target, key, value) {
        if(key === 'redPhosphorus') {
            if(target.allowDangerousOperations)
                return target.redPhosphorus = value;
            else
                return console.log("Too dangerous!");
        }
        // all other properties are safe
        target[key] = value;
    },
});

protectedCook.water = 550;                  // 550
protectedCook.redPhosphorus = 150;          // Too dangerous!

protectedCook.allowDangerousOperations = true;
protectedCook.redPhosphorus = 150;          // 150
```

讲到这里，我们也只是触到了代理的冰山一角。想要一探究竟的话，建议可以从 Axel Rauschmayer's 的文章“使用 ECMA-Script 6 代理进行元编程”开始，（*http://www.2ality.com/2014/12/es6-proxies.html*），后续可以阅读 MDN 文档（*http://mzl.la/1QZKM7U*）。

21.5 小结

在本章中，已经揭开了 JavaScript 对象机制的面纱，了解了对象属性的工作原理，学会了如何配置对象的行为。此外，还学会了如何保护对象不被修改。

最后，学习了一个 ES6 中非常有用的新概念：代理。它能为开发人员提供强大的元编程技术，而且据作者估计，当 ES6 流行起来以后，将会看到更多关于代理的有趣用法。

第 22 章

附加资源

前不久作者才认识到 JavaScript 是一个如此有表现力，且功能强大的语言，而编写本书的过程则让作者更是对此深以为然。JavaScript 不是一种能够很容易精通，或者是像“初学者语言”那样可以随意摒弃“玩具语言”。大家已经阅读完了本书，相信读者一定对此深有感触！

写本书的目的不是为了详细的介绍 JavaScript 中的每一种特性，更不是为了对重要的编程技巧进行逐个解释。如果 JavaScript 是读者的主要编程语言，那么本书只是一个开始。希望这本书能为读者成为 JavaScript 专家打下坚实的基础。

本章中的大部分资料都来自于作者的第一本书，《*Node 和 Express 开发*》(O'Reilly)。

22.1 在线文档

在 JavaScript、CSS 和 HTML 这些前端领域，火狐开发者社区（MDN）(*https://developer.mozilla.org*) 的文档无出其右。如果需要 JavaScript 文档，要么直接在 MDN 中搜索，要么就在搜索的内容后加上“mdn”。否则，搜索结果会不可避免地出现 w3school。我想管理 w3school 的 SEO 一定是个天才，才能让搜索结果中总能出现 w3school。但我还是建议尽量避免去这个网站，因为作者经常发现那里的文档内容严重欠缺。

MDN 中的 HTML 很值得参考，如果读者是 HTML5 的初学者（或者不是），应该读 Mark Pilgrim 的 *Dive Into HTML5*(*http://diveintohtml5.info*)。WHATWG（超 Web Hypertext Application Technology Working Group，文本应用工作技术组）维护了一个非常出色的 HTML5“动态标准”说明书，这也是作者每次遇到难以回答的 HTML

问题时第一个求助的地方。最后，需要指出的是，HTML 和 CSS 的官方说明书都在 W3C 的网站上（*http://www.w3.org*），他们是一些干巴巴的，又很难阅读的文档，不过有时候，当遇到难以解决的问题的时，这些文档又会是唯一的资源。

ES6 坚持了 ECMA-262 ECMAScript 2015 语言说明书。如果想了解 Node（以及多种浏览器）对 ES6 的支持，可以参考由@kangax 所维护的一个非常出色的说明书。

jQuery 和 Bootstrap 都有极其优秀的在线文档。

Node 文档不仅质量高，而且很全面，如果需要关于 Node 模块（比如，http、https 和 fs）的权威文档，那么它应该是首选。npm 文档也很全面，而且很有用，尤其是关于 package.json 这个文件的篇章。

22.2 期刊

有三个值得订阅并且坚持每周阅读的免费期刊：

- JavaScript Weekly。
- Node Weekly。
- HTML5 Weekly。

这三个免费期刊刊载关于该技术的最新消息、服务、博客和可用教程。

22.3 博客和教程

博客是能够让开发人员与 JavaScript 的快速发展保持同步。作者经常会在阅读下面这些博客的时候产生一种“原来如此”的感觉：

- Axel Rauschmayer 的博客有很多关于 ES6 及相关技术的好文章。Rauschmayer 博士会以计算机科学学术的角度来研究 JavaScript，不过他的文章会从基础内容开始，所以不必担心看不懂，而那些有计算机科学背景的也人会由衷的感谢他提供的附加信息。
- Nolan Lawson 的博客中，提到了很多在 JavaScript 开发中需要注意的细节。有时间的话一定要拜读他的“We Have a Problem with Promises”。
- David Walsh 的博客，包含了很多关于 JavaScript 开发的精彩文章和相关技术。

如果读者对 14 章的样例不太了解，那就一定要读他的“The Basics of ES6 Generators”。

- @kangax 的博客，Prefection Kills，里面有很多精彩的教程、练习、测试题。强烈建议初学者和专业人员都去看看。

已经读完本书了，在线课程和教程对读者来说应该很简单。但如果觉得自己还需要多了解一些基础知识，或者加强基础练习，那推荐下面这些资源：

- Lynda.com 上的 JavaScript 教程。
- Treehouse 上面的 JavaScript 程序。
- Codecademy 上的 JavaScript 相关课程。
- Microsoft Virtual Academy 上面介绍 JavaScript 的课程(http://bit.ly/ms_js_intro)；如果读者在用 Windows 进行 JavaScript 开发，建议可以把这些资料作为使用 Visual Studio 进行 JavaScript 开发时的参考。

22.4 Stack Overflow

可能读者早已经用过 Stack Overflow（SO）了，它从 2008 年出现起，到现在已经在开发人员在线问答这个领域占统治性地位了，同时它也是解决任何 JavaScript 问题（或者本书提到过的任何技术问题）的最佳资源。Stack Overflow 是一个由社区维护的、信誉良好的在线问答网站。它良好的信誉来源于它对高质量网站追求和责任心，以及持续的成功。用户可以通过问问题、被点赞的回复或者被接受的回复而获得良好的信誉。而提问时，则不需要任何信誉。同时，注册也是免费的。不过，还是有一些方式可以提高问题被回答的几率，本节将就此做一些讨论。

信誉就是 Stack Overflow 中的货币，如果有人真心帮助，那就有机会获取信誉，它就像蛋糕上的糖霜，可以驱动人们给出更好的答案。SO 上有很多非常聪明的人，他们都在为提供第一个答案/或最正确的答案而竞争（谢天谢地，这方式可以避免那些迅速但错误的回答）。完成下面这些事情可以帮助你得到一个好的答案。

成为一个 SO 用户

看看 SO 上的概述，阅读《如何问出好问题？》。如果读者愿意，还可以阅读所有的帮助性文档，读完所有文档后甚至可以获得一个徽章！

不要问那些已经被问过的问题

在提问之前先试着找找是否有人问过同样的问题。如果所提出的问题很容易就在SO上找到之前回答过的答案，那么这个问题将很快会因为重复而被关掉，人们也会因此对此问题点“踩”，这会对提问人的信誉产生负面影响。

不要让别人给你写代码

如果只问“我要怎样做XX”，那么问题很快会被人“踩”，并关掉。因为SO社区鼓励人们在提出问题之前先自己尝试解决它。在提出的问题中，需要描述所做的尝试，以及为什么这些尝试不成功。

一次只问一个问题

那些一次问了太多事情的问题，如“我应该怎么做A，然后是B，接着是CED，怎么才是最好的方式？”将很难回答，而且很容易让人丧失想要回答的欲望。

构造一个能描述问题的最小例子

作者在SO上回答过很多问题，而每当看到一个有好几页代码（或者更多）的时候，都会自动跳过这些问题。把项目中上千行的代码贴到SO中并不能帮助获取答案(但人们却经常这么干)。这是一个比较懒的做法，通常都不会有好的回报。这不仅会让提问的人很难获得以一个有用的答案，同时，如果提问的人够细心，那么在排除这些不会造成问题的事情的过程中，就能引导找出答案（那样就不用在SO上提问了)。构造一个最小例子能提高debug的技巧和进行关键思考的能力，同时有助于成为一个合格的SO公民。

学习Markdown

Stack Overflow的用户用Markdown来格式化问题和答案。一个格式良好的问题更有可能被回答，所以应该花一些时间学习这个不仅用处大，而且正渐渐变得无所不在的标记语言。

接受并给回答点赞

如果有人圆满地解决了所提的问题，那么应该点“赞”并接受这个答案，这会增加回答者的信誉，而信誉正是SO上的驱动力。如果有多个人都给出了正确的答案，则应该选择一个提问人认为最合适的答案然后接受它，然后给其他觉得有用的答案点“赞”。

如果先于别人找到了问题的根源，那就自己回答自己的问题

SO 是一个社区资源，如果某个开发人员遇到一个问题，那么很可能有人遇到了同样的问题。如果提问者本人已经找出了解决方案，那么就回答自己的问题，从而帮助他人。

如果喜欢帮助社区，可以考虑在社区中回答一些问题，既有趣，又能获得奖励，而且往往能收获到比声誉积分更实在的好处。如果发现所提的问题在两天之内都没有得到回答，那么可以使用自己积分在问题上设置“赏金”，这样积分会立即被扣除，而且不会返还。如果有人给出了令人满意的回答，那么点击接受他们的回答，他们就会获得你设置的赏金。当然前提是提问者必须有积分才行，且不少于 50 积分，因为最少的赏金是 50 积分。相比于问出高质量的问题， 提供高质量的回答一般会更快的获得积分。

回答别人的问题对自己的学习也很有帮助。作者常常会觉得相比于得到问题的答案，在回答别人问题的时候能够学到更多。如果想完全了解一门技术，可以在学习了基本知识后，开始在 SO 上回答别人的问题。起初可能会被一些早已成为专家的人比下去，但久而久之，会发现自己也变成了专家。

最后，读者应该马上使用自己的信誉积分来发展事业。如果积分够高，那它绝对应该被写入简历。作者就这么干过，而现在，作者已经是一名程序员面试官了。每次看到 SO 高分（作者认为 3000 分以上就算高了；如果是五位数，那简直是学霸级别），这个简历都会给人留下深刻的印象。一个 SO 高分账号表明，这个人不仅是这个领域的佼佼者，更是一个善于沟通和乐于助人的人。

22.5 给开源项目做贡献

有一个很棒的学习方式，就是给开源项目做贡献：不仅能够碰到那些激发潜力的挑战，同时自己的代码也会被社区中的其他人 review，这些都会帮助成为一个更好的开发人员。同时也能让自己的简历更好看。

如果是一个初学者，那么可以从文档开始贡献。很多开源项目都深受文档问题的困扰，作为一名初学者，正处于一个绝佳的位置：可以自己学习一些东西，然后用初学者能够理解的方式解释给他们。

有时候开源社区可能会让人望而却步，但如果能够坚持不懈，并且总是善于接受建设性的意见，那么最终会发现自己的贡献还是很受欢迎的。可以从阅读 Scot

Hanselman 的博客“将善良带回开源”开始，在这篇文章中，推荐了 Up for Grabs 这个致力于将程序员和开源项目联系起来的网站。在网站中搜索 JavaScript，就会看到很多正在寻求帮助的开源项目。

22.6 小结

恭喜！至此，读者即将踏上成为 JavaScript 开发者的道路！本书中有一部分内容看起来会非常难懂，但是只要你花时间将它们完全搞懂，那么这将帮助为这门重要的语言打下非常坚实的基础。如果有些内容实在搞不懂，也不要灰心。JavaScript 是一门非常复杂和强大的语言，以至于不可能一夜之间（甚至一年之内）就融会贯通。如果读者是编程初学者，那么时常回来翻翻这本书吧，那些第一次看起来晦涩难懂的内容，说不定回头再看的时候会有新的收获！

ES6 的出现衍生出新一代的开发者，同时也带来了很多奇思妙想。作者建议：竭尽全力去阅读，寻找一切机会去跟 JavaScript 开发者们交流，从一切能获取到的源码中学习。在 JavaScript 领域，将会出现一次有深度而又富有创造力的大爆发。开发者社区，以及作者本人都希望读者能为这次大爆发贡献自己的力量。

O'REILLY®

JavaScript学习指南（第3版）

这是学习JavaScript的一个大时代。最新的JavaScript标准——ECMAScript 6.0 (ES6)已经定稿，学习如何使用这种语言来开发高质量的应用程序变得前所未有地简单和舒服。

本书将带领程序员开启一次充实的ES6之旅，同时也会介绍一些相关的工具和技术。本书不仅会介绍基础知识（比如变量、控制流和数组），还会讲解函数式编程和异步编程等复杂的概念。你将会学习如何在客户端（或者在服务端使用Node.js）构建强大的响应式Web应用。

通过阅读本书，你将学会：

- **使用ES6编程，然后通过转译来兼容ES5；**
- **将数据转换为JavaScript可以使用的格式；**
- **理解JavaScript中函数的基本用法和机制；**
- **探索对象和面向对象编程；**
- **了解新的概念，比如迭代器、生成器和代理；**
- **理解异步编程的复杂性；**
- **利用文档对象模型（DOM）来构建基于浏览器的应用；**
- **学习Node.js的基础知识来构建服务器端应用程序。**

Ethan Brown是互动营销公司Pop Art的工程总监，负责网站和网络服务的架构和实现，面向从小公司到跨国企业的多种客户。他有着超过20年的编程经验。

“这是开发者学习JS的一个大时代。但是我说的并不是那种‘我的代码能跑起来’一样的小打小闹。这本书将带你更深入地学习我们所需的那些知识。”

——Kyle Simpson
《你不知道的JavaScript》
系列书作者

“这本书写得很好，内容紧凑，介绍了JavaScript的一切，甚至包含了ECMAScript 6。”

——Axel Rauschmayer
《深入理解JavaScript》作者

WEB PROGRAMMING / JAVASCRIPT

异步社区 人民邮电出版社 www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：Karen Montgomery 张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

分类建议：计算机／软件开发／JavaScript
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-45632-8

ISBN 978-7-115-45632-8
定价:59.00 元