本书赞誉

“这是一本绝不辜负 Effective 软件开发系列期望的编程书籍。对于任何一个想要做到严谨编程的 JavaScript 开发者来说，这本书绝对不容错过。这本书阐述了 JavaScript 内部工作的一些细节，以期帮助读者更好地利用 JavaScript 语言优势。”

——**Erik Arvidsson，高级软件工程师**

“很少有像 David 这样的编程语言极客能如此舒适、友好地讲解编程语言。他带领我们领会 JavaScript 语法和语义，这个过程既令人陶醉又极其深刻。本书以舒适的节奏额外提供了一些“有问题”的现实案例。当你读完本书后，你会感觉自己获得了一种强大而全面的掌控能力。”

——**Paul Irish，Google Chrome 开发主管**

“在阅读本书之前，我以为它只是另一本关于如何更好地使用 JavaScript 编程的书籍。然而本书远不止如此，它还会使你更深入地理解 JavaScript 这门语言，这是至关重要的。如果没有这层对 JavaScript 的深入理解，那么你绝不会懂得语言本身的任何东西，只知道其他的程序员是如何编写代码的。”

“如果你想成为一名真正优秀的 JavaScript 开发者，那么请阅读本书。就我来说，我多么希望在第一次开始 JavaScript 编程时就已经阅读了它。”

——**Anton Kovalyov，JSHint 开发者**

“如果你正在寻找一本正式且极具可读性的并极具洞察力的 JavaScript 语言的书籍，那不用舍近求远了。JavaScript 开发者能够从其中找到珍贵的知识宝藏，甚至技术精湛的 JavaScript 程序员也一定能从中获益。对于有其他语言经验而想一头扎进 JavaScript 世界的从业人员来说，本书是迅速学习 JavaScript 的必读之物。然而，不管你的背景如何，但都不得不承认作者 Dave Herman 在探索 JavaScript 方面做得非常棒——JavaScript 的优美部分、不足部分或介于两者之间的所有内容都囊括于本书之中。”

——**Rebecca Murphey，Bocoup 高级 JavaScript 开发者**

“对于任何一位理解 JavaScript 并且想要完全掌握它的人员来说，本书是必不可少的读物。Dave Herman 带给了读者深刻的、具有研究和实践意义的 JavaScript 语言理解，通过一个接一个的例子指导并帮助读者达到与他同样的理解高度。这不是一本寻求捷径的书籍，恰恰相反，是一本难得的将经验提炼为指南的书籍。它是一本为数不多让我毫不犹豫推荐的关于 JavaScript 的书籍。”

——**Alex Russell，TC39 成员，Google 软件工程师**

“很少有人有机会同大师一起学习他们的手艺。这本书弥补了这种遗憾，其对 JavaScript 的研究就像随一位时间旅行哲学家回到公元前 5 世纪与柏拉图一同学习。”

——**Rick Waldron，JavaScript 传教士，Bocoup**

译 者 序

虽然 JavaScript 在诞生之初由于商业原因及缺乏规范，一直饱受诟病。但是随着时间的推移，人们已经逐渐走出了对这门语言的偏见和误解，开始领略它那强大的语言特性威力。当下 JavaScript 语言大红大紫，研究讨论 JavaScript 的相关书籍早已汗牛充栋，但是这本书作为 Effective 软件开发系列中的一员，却是不可或缺的。

学会编写 JavaScript 程序容易，但要成为专家却实属不易。一方面是由于 JavaScript 语言的设计思想与 Java、C# 等大众语言区别很大，另一方面是由于其设计时的仓促性导致 JavaScript 语言本身精华与糟粕并存。本书的作者 David Herman 作为 JavaScript 标准化的参与者，在书中自然对 JavaScript 的精华和糟粕都进行了深入阐述，并且给出了很多实用的建议。这些建议都来自于第一线的实践经验，无论是初学者还是高级程序员，都可以从中吸收养分，进而快速成长。

本书深入阐述了 JavaScript 语言，通过它可以了解到如何有效地编写出高移植性、健壮的程序和库。本书传承了 Effective 软件开发系列的简明场景驱动风格，通过提示、技术及实用的示例代码解释 JavaScript 中的重要概念。

全书共涉及 68 条关于 JavaScript 程序设计的建议。第 1 章可以让初学者快速熟悉 JavaScript，了解 JavaScript 中的原始类型、隐式强制转换、编码类型等基本概念；第 2 章着重讲解了有关 JavaScript 的变量作用域的建议，不仅介绍了怎么做，还介绍了操作背后的原因，帮助读者加深理解；第 3 章和第 4 章的主题涵盖函数、对象及原型三大方面，这可是 JavaScript 区别于其他语言的核心，读者也不必紧张，在 David Herman 大叔的指引下，你可以轻松掌握这些核心内容，了解到业界最佳实践；第 5 章则阐述了数组和字典这两种容易混淆的常用类型及具体使用时的建议，避免陷入一些陷阱；第 6 章讲述了库和 API 设计；第 7 章讲述了并行编程，这是晋升为 JavaScript 专家的必经之路。

想要深入了解 JavaScript 并获取一线专家的宝贵经验吗？那么，这本书正好适合你。

我和同事喻杨在翻译这本书的过程中投入了不少精力，生怕给这本经典之作留下一些遗憾。感谢华章公司的编辑们对我们的支持。

最后，希望本书能给大家带来一次超凡的阅读体验。

黄博文

序

众所周知，我在 1995 年 5 月用了 10 天时间创建了 JavaScript 语言。迫于现实的压力和冲突管理的势在必行，我将 JavaScript 设计为“看起来像 Java”、“方便初学者”、“在网景浏览器中几乎能控制它的一切”的编程语言。

鉴于极具挑战性的要求和非常短的时间表，我的解决方案除了正确获得了两大特性之外（第一类函数、对象原型），还将 JavaScript 设计为一开始就极具延展性。我知道一些开发者不得不对最开始的几个版本“打补丁”来修正错误，这些先驱的方法比我使用内置库胡乱拼凑的方法更好。举例来说，虽然许多编程语言都限制了可变性，在运行时不能修改或扩展内置对象，或者标准库的名称绑定不能通过赋值被覆盖，但是 JavaScript 几乎允许完全改变每个对象。

总的来说，我相信这是一个很好的设计决定。它明确了某些领域的挑战（如在浏览器的安全边界内，安全地混用可信和不可信的代码）。对于 JavaScript 来说，支持所谓的猴子补丁是很重要的。凭借它，开发者可以编辑标准对象来解决 Bug 和仿效“未来”的功能改造一些旧的浏览器（称为 polyfill 库的 shim，在美式英语中称为 spackle）。

除了这些平凡的应用之外，JavaScript 的可塑性鼓励用户沿着几个更富有创造性的方向形成和发展创新网络。这使得用户仿效其他编程语言，创建了许多工具包和框架库：基于 Ruby 的 Prototype、基于 Python 的 MochiKit、基于 Java 的 Dojo 以及基于 Smalltalk 的 TIBET。随后是 jQuery 库（“JavaScript 的新浪潮”），2007 年我第一次看到它，对我来说，我似乎是一个后来者。它暴风雨般地引领着 JavaScript 的世界，异于其他编程语言，而从旧的 JavaScript 库中学习，开辟了浏览器的“查询和做”（query and do）模型，从根本上简化了浏览器。

这引领着 JavaScript 用户群及其创新网络，促成了一种 JavaScript“自家风格”。这种风格仍在仿效和简化其他的程序库，也促成了现代 Web 标准化的工作。

在这一演变的过程中，JavaScript 保持了向后兼容，当然默认情况下，它是可变的。另外，在 ECMAScript 标准最新的版本中新增了一些方法来冻结对象和封闭对象的属性，以防止对象扩展和属性被覆盖。JavaScript 的演进之旅远未结束。这就像生活的语言和生物系统一样，变化始终存在。在此之前，我无法预见一个单一的横扫其他程序库的“标准库”或编码风格。

没有语言不是怪癖的，或者所有语言几乎都是受限地执行常见的最佳实践。JavaScript 是一门怪癖的或者充满限制主义色彩的编程语言。因此，与大多数其他的编程语言相比，想要高效地使用 JavaScript 编程，开发人员必须学习和追求优秀的风格、正确的用法和最佳的实践。当考虑如何做到最高效，我相信避免过度编程和构建刚性或教条式的风格指南是至关重要的。

本书以一种平衡的方式讲解 JavaScript 编程。它基于一些具体的实证和经验，而不迂回于刚性或过度的方法。我认为对于许多寻找以不牺牲表现力，并希望自由采用新思想和编程范式来编写高效 JavaScript 的人来说，它是一位重要的助手和可信赖的向导。它也是一本备受关注、充满乐趣和拥有许多令人叹为观止的示例的读物。

最后，自 2006 年，我有幸结识 David Herman。那是我第一次以 Mozilla 代表的身份邀请他作为特邀专家在 Ecma 标准化机构工作。Dave 深刻、谦逊的专家意见以及他的热情让 JavaScript 在书中的每一页大放异彩。本书绝无仅有！

Brendan Eich

前　言

学习一门编程语言，需要熟悉它的语法、形式和结构，这样我们才能编写合法的、符合语义的、具有意义和行为正确的程序。但除此之外，掌握一门语言需要理解其语用，即使用语言特性构建高效程序的方法。后一种范畴是特别微妙的，尤其是对 JavaScript 这样一种灵活而富有表现力的编程语言来说。

这是一本关于 JavaScript 语用学的书。这不是一本入门书籍，我假设你在一定程度上熟悉了 JavaScript 和通常的编程。很多优秀的 JavaScript 入门书籍可供参考，例如，Douglas Crockford 的《 JavaScript: The Good Parts 》和 Marijn Haverbeke 的《 Eloquent JavaScript 》。本书的目的是帮助你加深理解如何有效地使用 JavaScript 构建更可预测、可靠和可维护的 JavaScript 应用程序和库。

JavaScript 与 ECMAScript

在深入本书之前澄清一些术语是有必要的。这是一本关于举世皆知的 JavaScript 编程语言的书籍。然而，官方标准定义的规范的描述是一门称该语言为 ECMAScript。历史很令人费解，但这可以归结为版权问题：出于法律原因，Ecma 国际标准化组织不能使用“JavaScript”作为其标准名称。（更糟的是，标准化组织将其原来的名称 ECMA（欧洲计算机制造商协会的英文首字缩写）改为不是全大写的 Ecma 国际标准化组织。彼时，ECMAScript 这个名字大约也是早已注定。）

正式来说，当人们提到 ECMAScript 时，通常是指由 Ecma 国际标准化组织制定的“理想”语言。与此同时，JavaScript 这个名字意味着来自语言本身的所有事物，例如某个供应商特定的 JavaScript 引擎。通常情况下，人们经常交替使用这两个术语。为了保持清晰度和一致性，在本书中，我将只使用 ECMAScript 来谈论官方标准，其他情况，我会使用 JavaScript 指代语言。我还会使用常见的缩写 ES5 来指代第 5 版的 ECMAScript 标准。

关于 Web

避开 Web 来谈 JavaScript 是很难的。到目前为止，JavaScript 是唯一为用于客户端应用程

序脚本的所有主流浏览器提供内置支持的编程语言。此外，近年来，随着 Node.js 平台的问世，JavaScript 已经成为一个实现服务器端应用程序的流行编程语言。

不过，本书是关于 JavaScript 而非 Web 的编程。有时，谈论一些 Web 相关的例子和应用程序的概念是帮助读者理解。但是，这本书的重点是 JavaScript 语言的语法、语义和语用，而不是 Web 平台的 API 和技术。

关于并发

JavaScript 一个新奇的方面是在并发环境中其行为是完全不明朗的。ECMAScript 标准（包括第 5 版）关于 JavaScript 程序在交互式或并发环境中的行为只字未提。第 7 章涉及并发，因此，我只是从技术角度介绍一些非官方的 JavaScript 特性。但实际上，所有主流的 JavaScript 引擎都有一个共同的并发模型。尽管在标准中未提及并发，但是致力于并发和交互式的程序是 JavaScript 编程的一个核心概念。事实上，未来版本的 ECMAScript 标准可能会正式地标准化这些 JavaScript 并发模型的共享方面。

致谢

这本书在很大程度上要归功于 JavaScript 的发明者——Brendan Eich。我深深感谢 Brendan 邀请我参与 JavaScript 标准化工作，以及他对我在 Mozilla 的职业生涯中给予的指导和支持。

本书中的大部分材料是受优秀的博客文章和在线论文的启发。我从 Ben“cowboy”Alman、Erik Arvidsson、Mathias Bynens、Tim“creationix”Caswell、Michaeljohn“inimino”Clement、Angus Croll、Andrew Dupont、Ariya Hidayat、Steven Levithan、Pan Thomakos、Jeff Walden，以及 Juriy“kangax”Zaytsev 的博客中学到很多东西。当然，本书的最终资源来自 ECMAScript 规范。ECMAScript 规范自第 5 版以来由 Allen Wirfs-Brock 不知疲倦地编辑和更新。Mozilla 开发者网络仍然是 JavaScript API 和特性最令人印象深刻的、高品质在线资源之一。

在策划和写作这本书的过程中，我有许多顾问。在我开始写作之前，John Resig 就以作者的角度给了我很多有用的建议。Blake Kaplan 和 Patrick Walton 帮我在早期阶段整理我的想法和规划出这本书的组织结构。在写作的过程中，我从 Brian Anderson、Norbert Lindenberg、Sam Tobin-Hochstadt、Rick Waldron 和 Patrick Walton 那里得到了很好的建议。

很高兴能够和 Pearson 的工作人员共事。Olivia Basegio、Audrey Doyle、Trina MacDonald、Scott Meyers 和 Chris Zahn 一直关注我提出的问题，对我的拖延报以耐心，并通融我的请求。我无法想象还能有一个更愉快的写作经历。我对能为 Effective 系列写一本书感到非常荣幸。因为很久

以前我就是《Effective C++》的粉丝，我曾经怀疑我是否有亲自书写一本 Effective 系列书籍的荣幸。

我也简直不敢相信自己有这样的好运气，能够找到梦之队一样的技术编辑。我很荣幸 Erik Arvidsson、Rebecca Murphey、Rick Waldron 和 Richard Worth 同意编辑这本书，他们为我提供了许多宝贵的批评和建议。他们多次纠正了书中一些真正令人尴尬的错误。

写一本书比我预想的要难得多。如果不是朋友和同事的支持，我可能已经失去了勇气。在我怀疑自己的时候，Andy Denmark、Rick Waldron 和 Travis Winfrey 总是给予我鼓励。

我绝大部分时候是在旧金山柏丽附近的神话般的 Java Beach 咖啡厅里写作这本书的。那里的工作人员都知道我的名字，并且我还没点餐之前，他们就知道我想要点什么。我很感谢他们提供了一个舒适的工作场所，并给我提供食物和咖啡。

我的毛茸茸的猫科小友 Schmoopy 为本书做出了它的最大贡献。至少，它不停地跳上我的膝盖，坐在屏幕前（有可能是笔记本电脑比较温暖）。Schmoopy 自 2006 年以来一直是我的忠实伙伴，我不能想象我的生活能离得开这个小毛球。

我的整个家庭对这个项目从开始到结束一直都很支持和激动。遗憾的是，我无法在我的爷爷和奶奶（Frank 和 Miriam Slamar）去世之前和他们分享这本书的成品。但他们会为我感到激动和自豪，而且本书中有一小段我儿时与爷爷 Frank 编写 BASIC 程序的经历。

最后，我要感谢我一生的挚爱 Lisa Silveria，我对她的付出无以为报。

第 1 章
让自己习惯 JavaScript

JavaScript 最初设计令人感觉亲切。由于其语法让人联想到 Java，并且具有许多脚本语言的共同特性（如函数、数组、字典和正则表达式），因此，具有少量编程经验的人也能够快速学习 JavaScript。新手程序员几乎不需要培训就可以开始编写程序，这要归功于 JavaScript 语言提供的为数不多的核心概念。

虽然 JavaScript 是如此的平易近人，但是精通这门语言需要更多的时间，需要更深入地理解它的语义、特性以及最有效的习惯用法。本书每个章节都涵盖了高效 JavaScript 编程的不同主题。第 1 章主要讲述一些最基本的主题。

第 1 条：了解你使用的 JavaScript 版本

像大多数成功的技术一样，JavaScript 已经发展了一段时间。最初 JavaScript 作为 Java 在交互式网页编程方面的补充而推向市场，但它最终完全取代了 Java 成为主流的 Web 编程语言。JavaScript 的普及使得其于 1997 年正式成为国际标准，其官方名称为 ECMAScript[⊖]。目前许多 JavaScript 的竞争实现都提供了 ECMAScript 标准的各种版本的一致性。

1999 年定稿的第 3 版 ECMAScript 标准（通常简称为 ES3），目前仍是最广泛采用的 JavaScript 版本。下一个有重大改进的标准是 2009 年发布的第 5 版，即 ES5。ES5 引入了一些新的特性，并且标准化了一些受到广泛支持但之前未规范的特性。由于 ES5 目前还未得到广泛支持，所以我会适时指出本书中的条款或建议是否特定于 ES5。

除了 ECMAScript 标准存在多个版本之外，还存在一些 JavaScript 实现支持非标准特性，而其他的 JavaScript 实现却并不支持这些特性的情况。例如，许多 JavaScript 引擎支持 const

⊖ ECMAScript 是一种由欧洲计算机制造商协会（ECMA）通过 ECMA-262 标准化的脚本程序设计语言。——译者注

关键字定义变量，但ECMAScript标准并没有定义任何关于const关键字的语义和行为。此外，在不同的实现之间，const关键字的行为也不一样。在某些情况下，const关键字修饰的变量不能被更新。

```
const PI = 3.141592653589793;
PI = "modified!";
PI; // 3.141592653589793
```

而其他的实现只是简单地将const视为var的代名词。

```
const PI = 3.141592653589793;
PI = "modified!";
PI; // "modified!"
```

由于JavaScript历史悠久且实现多样化，因此我们很难确定哪些特性在哪些平台上是可用的。而令事态更加严峻的事实是JavaScript的主要生态系统——Web浏览器，它并不支持让程序员指定某个JavaScript的版本来执行代码。由于最终用户可能使用不同Web浏览器的不同版本，因此，我们必须精心地编写Web程序，使得其在所有的浏览器上始终工作如一。

另外，JavaScript并不只是针对客户端Web编程。JavaScript的其他应用包括服务器端程序、浏览器扩展以及针对移动和桌面应用程序的脚本。某些情况下你可能需要一个特定的JavaScript版本。对于这些情况，利用特定平台支持的JavaScript特定实现的额外特性是有意义的。

本书主要关注的是JavaScript的标准特性，但是也会讨论一些广泛支持的非标准特性。当涉及新标准特性或非标准特性时，了解你的应用程序运行环境是否支持这些特性是至关重要的。否则，你可能会面临这样的困境——应用程序在你自己的计算机或者测试环境上运行良好，但是将它部署在不同的产品环境中时却无法运行。例如，const关键字在支持非标准特性的JavaScript引擎上测试时运行良好，但是，当将它部署在不识别const关键字的Web浏览器上时就会出现语法错误。

ES5引入了另一种版本控制的考量——严格模式（strict mode）。此特性允许你选择在受限制的JavaScript版本中禁止使用一些JavaScript语言中问题较多或易于出错的特性。由于其语法设计向后兼容，因此即使在那些没有实现严格模式检查的环境中仍然可以执行严格代码[⊖]（strict code）。在程序中启用严格模式的方式是在程序的最开始增加一个特定的字符串字面量（literal）。

```
"use strict";
```

同样，你也可以在函数体的开始处加入这句指令以启用该函数的严格模式。

⊖ 严格代码是指期望运行于严格模式下的代码。——译者注

```
function f(x) {
    "use strict";
    // ...
}
```

使用字符串字面量作为指令语法看起来有点怪异，但它的好处是向后兼容。由于解释执行字符串字面量并没有任何副作用，所以ES3引擎执行这条指令是无伤大雅的。ES3引擎解释执行该字符串，然后立即丢弃其值。这使得编写的严格模式的代码可以运行在旧的JavaScript引擎上，但有一个重要的限制：旧的引擎不会进行任何的严格模式检查。如果你没有在ES5环境中做过测试，那么，编写的代码运行于ES5环境中就很容易出错。

```
function f(x) {
    "use strict";
    var arguments = []; // error: redefinition of arguments
    // ...
}
```

在严格模式下，不允许重定义arguments变量，但没有实现严格模式检查的环境会接受这段代码。然而，这段代码部署在实现ES5的产品环境中将导致程序出错。所以，你应该总是在完全兼容ES5的环境中测试严格代码。

“use strict”指令只有在脚本或函数的顶部才能生效，这也是使用严格模式的一个陷阱。这样，脚本连接变得颇为敏感。对于一些大型的应用软件，在开发中使用多个独立的文件，然而部署到产品环境时却需要连接成一个单一的文件。例如，想将一个文件运行于严格模式下：

```
// file1.js
"use strict";
function f() {
    // ...
}
// ...
```

而另一个文件不是运行于严格模式下：

```
// file2.js
// no strict-mode directive
function g() {
    var arguments = [];
    // ...
}
// ...
```

我们怎样才能正确地连接这两个文件呢？如果我们以file1.js文件开始，那么连接后的代码运行于严格模式下：

```
// file1.js
"use strict";
function f() {
    // ...
}
// ...
// file2.js
// no strict-mode directive
function f() {
    var arguments = []; // error: redefinition of arguments
    // ...
}
// ...
```

如果我们以 file2.js 文件开始，那么连接后的代码运行于非严格模式下：

```
// file2.js
// no strict-mode directive
function g() {
    var arguments = [];
    // ...
}
// ...
// file1.js

"use strict";
function f() { // no longer strict
    // ...
}
// ...
```

在自己的项目中，你可以坚持只使用“严格模式”或只使用“非严格模式”的策略，但如果你要编写健壮的代码应对各种各样的代码连接，你有两个可选的方案。

第一个解决方案是不要将进行严格模式检查的文件和不进行严格模式检查的文件连接起来。这可能是最简单的解决方案，但它无疑会限制你对应用程序或库的文件结构的控制力。在最好的情况下，你至少要部署两个独立的文件。一个包含所有期望进行严格检查的文件，另一个则包含所有无须进行严格检查的文件。

第二个解决方案是通过将其自身包裹在立即调用的函数表达式（Immediately Invoked Function Expression, IIFE）中的方式连接多个文件。第 13 条将对立即调用的函数表达式进行深入的讲解。总之，将每个文件的内容包裹在一个立即调用的函数中，即使在不同的模式下，它们都将被独立地解释执行。基于此方案，上面例子的连接版本如下：

```
// no strict-mode directive
(function() {
```

```
    // file1.js
    "use strict";
    function f() {
        // ...
    }
    // ...
})();
(function() {
    // file2.js
    // no strict-mode directive
    function f() {
        var arguments = [];
        // ...
    }
    // ...
})();
```

由于每个文件的内容被放置在一个单独的作用域中，所以使用严格模式指令（或者不使用严格模式指令）只影响本文件的内容。但是这种方式会导致这些文件的内容不会在全局作用域内解释。例如，var 和 function 声明的变量不会被视为全局变量（更多关于全局概念的内容参见第 8 条）。这恰好与流行的模块系统（module system）类似，模块系统通过自动地将每个模块的内容放置在单独的函数中的方式来管理文件和依赖。由于所有文件都放置在局部作用域内，所以每个文件都可以自行决定是否要使用严格模式。

编写文件使其在两种模式下行为一致。想要编写一个库，使其可以工作在尽可能多的环境中，你不能假设库文件会被脚本连接工具置于一个函数中，也不能假设客户端的代码库是否处于严格模式或者非严格模式。要想构建代码以获得最大的兼容性，最简单的方法是在严格模式下编写代码，并显式地将代码内容包裹在本地启用了严格模式的函数中。这种方式类似于前面描述的方案——将每个文件的内容包裹在一个立即调用的函数表达式中，但在这种情况下，你是自己编写立即调用的函数表达式并且显式地选择严格模式，而不是采用脚本连接工具或模块系统帮你实现。

```
(function() {
    "use strict";
    function f() {
        // ...
    }
    // ...
})();
```

要注意的是，无论这段代码是在严格模式还是在非严格模式的环境中连接的，它都被视为是严格的。相比之下，即使一个函数没有选择严格模式，如果它连接在严格代码之后，它

仍被视为是严格的。所以，为了达到更为普遍的兼容性，建议在严格模式下编写代码。

- 决定你的应用程序支持 JavaScript 的哪些版本 。
- 确保你使用的任何 JavaScript 的特性对于应用程序将要运行的所有环境都是支持的。
- 总是在执行严格模式检查的环境中测试严格代码。
- 当心连接那些在不同严格模式下有不同预期的脚本。

第 2 条：理解 JavaScript 的浮点数

大多数编程语言都有几种数值型数据类型，但是 JavaScript 却只有一种。你可以使用 typeof 运算符查看数字的类型。不管是整数还是浮点数，JavaScript 都将它们简单地归类为数字。

```
typeof 17;   // "number"
typeof 98.6; // "number"
typeof -2.1; // "number"
```

事实上，JavaScript 中所有的数字都是双精度浮点数。这是由 IEEE[⊖]754 标准制定的 64 位编码数字——即“doubles”。如果这一事实使你疑惑 JavaScript 是如何表示整数的，请记住，双精度浮点数能完美地表示高达 53 位精度的整数。从 –9 007 199 254 740 992（-2^{53}）到 9 007 199 254 740 992（2^{53}）的所有整数都是有效的双精度浮点数。因此，尽管 JavaScript 中缺少明显的整数类型，但是完全可以进行整数运算。

大多数的算术运算符可以使用整数、实数或两者的组合进行计算。

```
0.1 * 1.9  // 0.19
-99 + 100; // 1
21 - 12.3; // 8.7
2.5 / 5;   // 0.5
21 % 8;    // 5
```

然而位算术运算符比较特殊。JavaScript 不会直接将操作数作为浮点数进行运算，而是会将其隐式地转换为 32 位整数后进行运算。（确切地说，它们被转换为 32 位大端（big-endian）的 2 的补码表示的整数。）以按位或运算表达式为例：

```
8 | 1; // 9
```

看似简单的表达式实际上需要几个步骤来完成运算。如前所述，JavaScript 中的数字 8 和 1 都是双精度浮点数。但是它们也可以表示成 32 位整数，即 32 位 0、1 的序列。整数 8 表示

⊖ 美国电气和电子工程师协会（Institute of Electrical and Electronics Engineers）是一个国际性的电子技术与信息科学工程师的协会，是世界上最大的专业技术组织之一。——译者注

为 32 位二进制序列如下所示：

```
00000000000000000000000000001000
```

你自己也可以使用数字类型的 toString 方法来查看：

```
(8).toString(2); // "1000"
```

toString 方法的参数指定了其转换基数，此例子以基数 2（即二进制）表示。结果值省略了左端多余的 0（位），因为它们并不影响最终值。

整数 1 表示为 32 位二进制如下所示：

```
00000000000000000000000000000001
```

按位或运算表达式合并两个比特序列。只要参与运算的两位比特中任意一位为 1，运算结果的该位就为 1。以位模式表示的结果如下：

```
00000000000000000000000000001001
```

这个序列表示整数 9。你可以使用标准的库函数 parseInt 验证，同样以 2 作为基数：

```
parseInt("1001", 2); // 9
```

（同样，前导 0 位是不必要的，因为它们并不影响运算结果。）

所有位运算符的工作方式都是相同的。它们将操作数转换为整数，然后使用整数位模式进行运算，最后将结果转换为标准的 JavaScript 浮点数。一般情况下，JavaScript 引擎需要做些额外的工作来进行这些转换。因为数字是以浮点数存储的，必须将其转换为整数，然后再转换回浮点数。然而，在某些情况下，算术表达式甚至变量只能使用整数参与运算，优化编译器有时候可以推断出这些情形而在内部将数字以整数的方式存储以避免多余的转换。

关于浮点数的最后警示是，你应该对它们保持时刻警惕。浮点数看似熟悉，但是它们是出了名的不精确。甚至一些看起来最简单的算术运算都会产生不正确的结果。

```
0.1 + 0.2; // 0.30000000000000004
```

尽管 64 位的精度已经相当高了，但是双精度浮点数也只能表示一组有限的数字，而不能表示所有的实数集。浮点运算只能产生近似的结果，四舍五入到最接近的可表示的实数。当你执行一系列的运算，随着舍入误差的积累，运算结果会越来越不精确。舍入也会使我们通常所期望的算术运算定律产生一些出人意料的偏差。例如，实数满足结合律，这意味着，对于任意的实数 x，y，z，总是满足 $(x+y)+z=x+(y+z)$。

然而，对于浮点数来说，却并不总是这样。

```
(0.1 + 0.2) + 0.3; // 0.6000000000000001
0.1 + (0.2 + 0.3); // 0.6
```

浮点数权衡了精度和性能。当我们关心精度时，要小心浮点数的局限性。一个有效的解

决方法是尽可能地采用整数值运算，因为整数在表示时不需要舍入。当进行货币相关的计算时，程序员通常会按比例将数值转换为最小的货币单位来表示再进行计算，这样就可以以整数进行计算。例如，如果上面的计算是以美元为单位，那么，我们可以将其转换为整数表示的美分进行计算。

```
(10 + 20) + 30; // 60
10 + (20 + 30); // 60
```

对于整数运算，你不必担心舍入误差，但是你还是要当心所有的计算只适用于 $-2^{53} \sim 2^{53}$ 的整数。

提示

- JavaScript 的数字都是双精度的浮点数。
- JavaScript 中的整数仅仅是双精度浮点数的一个子集，而不是一个单独的数据类型。
- 位运算符将数字视为 32 位的有符号整数。
- 当心浮点运算中的精度陷阱。

第 3 条：当心隐式的强制转换

JavaScript 对类型错误出奇宽容。许多语言都认为表达式

```
3 + true; // 4
```

是错误的，因为布尔表达式（如 true）与算术运算是不兼容的。在静态类型语言中，含有类似这样表达式的程序甚至不会被允许运行。在一些动态类型语言中，含有类似这样表达式的程序可以运行，但是会抛出一个异常。然而，JavaScript 不仅允许程序运行，而且还会顺利地产生结果 4！

在 JavaScript 中有一些极少数的情况，提供错误的类型会产生一个即时错误。例如，调用一个非函数对象 (nonfunction) 或试图选择 null 的属性。

```
"hello"(1); // error: not a function
null.x;     // error: cannot read property 'x' of null
```

但是在大多数情况下，JavaScript 不会抛出一个错误，而是按照多种多样的自动转换协议将值强制转换为期望的类型。例如，算术运算符 -、*、/ 和 % 在计算之前都会尝试将其参数转换为数字。而运算符 + 更为微妙，因为它既重载了数字相加，又重载了字符串连接操作。具体是数字相加还是字符串连接，这取决于其参数的类型。

```
2 + 3;                  // 5
"hello" + " world"; // "hello world"
```

接下来，合并一个数字和一个字符串会发生什么呢？ JavaScript 打破了这一束缚，它更偏爱字符串，将数字转换为字符串。

```
"2" + 3; // "23"
2 + "3"; // "23"
```

类似这样的混合表达式有时令人困惑，因为 JavaScript 对操作顺序是敏感的。例如，表达式：

```
1 + 2 + "3";     // "33"
```

由于加法运算是自左结合的（即左结合律），因此，它等同于下面的表达式：

```
(1 + 2) + "3";  // "33"
```

与此相反，表达式：

```
1 + "2" + 3;     // "123"
```

的计算结果为字符串“123”。左结合律相当于是将表达式左侧的加法运算包裹在括号中。

```
(1 + "2") + 3;  // "123"
```

位运算符不仅会将操作数转换为数字，而且还会将操作数转换为 32 位整数（表示的数字的子集）。我们在第 2 条已经讨论过。这些运算符包括位算术运算符（~、&、^ 和 |）以及移位运算符（<<、>> 和 >>>）。

这些强制转换十分方便。例如，来自用户输入、文本文件或者网络流的字符串都将被自动转换。

```
"17" * 3;  // 51
"8" | "1"; // 9
```

但是强制转换也会隐藏错误。结果为 null 的变量在算术运算中不会导致失败，而是被隐式地转换为 0；一个未定义的变量将被转换为特殊的浮点数值 NaN（自相矛盾地命名为“not a number”。谴责 IEEE 浮点数标准！）。这些强制转换不是立即抛出一个异常，而是继续运算，往往导致一些令人困惑和不可预测的结果。无奈的是，即便是测试 NaN 值也是异常困难的。这有两个原因。第一，JavaScript 遵循了 IEEE 浮点数标准令人头痛的要求——NaN 不等于其本身。因此，测试一个值是否等于 NaN 根本行不通。

```
var x = NaN;
x === NaN;     // false
```

另外，标准的库函数 isNaN 也不是很可靠，因为它带有自己的隐式强制转换，在测试其参数之前，会将参数转换为数字（isNaN 函数的一个更精确的名称可能是 coercesToNaN）。如果你已经知道一个值是数字，你可以使用 isNaN 函数测试它是否是 NaN。

```
isNaN(NaN); // true
```

但是对于其他绝对不是 NaN，但会被强制转换为 NaN 的值，使用 isNaN 方法是无法区分的。

```
isNaN("foo");               // true
isNaN(undefined);           // true
isNaN({});                  // true
isNaN({ valueOf: "foo" }); // true
```

幸运的是，有一个既简单又可靠的习惯用法用于测试 NaN，虽然稍微有点不直观。由于 NaN 是 JavaScript 中唯一一个不等于其自身的值，因此，你可以随时通过检查一个值是否等于其自身的方式来测试该值是否是 NaN。

```
var a = NaN;
a !== a;                    // true
var b = "foo";
b !== b;                    // false

var c = undefined;
c !== c;                    // false
var d = {};
d !== d;                    // false
var e = { valueOf: "foo" };
e !== e;                    // false
```

你也可以将这种模式抽象为一个清晰命名的实用工具函数。

```
function isReallyNaN(x) {
    return x !== x;
}
```

其实测试一个值是否与其自身相等是非常简洁的，通常没有必要借助于一个辅助函数，但关键在于识别和理解。

隐式的强制转换使得调试一个出问题的程序变得令人异常沮丧，因为它掩盖了错误，使错误更难以诊断。当一个计算出了问题，最好的调试方式是检查这个计算的中间结果，回到出错前的“最后一点”。在那里，你可以检查每个操作的参数，查看错误类型的参数。根据错误的不同，它可能是一个逻辑错误（如使用了错误的算术运算符），也可能是一个类型错误（如传入了一个 undefined 的值而不是数字）。

对象也可以被强制转换为原始值。最常见的用法是转换为字符串。

```
"the Math object: " + Math; // "the Math object: [object Math]"
"the JSON object: " + JSON; // "the JSON object: [object JSON]"
```

对象通过隐式地调用其自身的 toString 方法转换为字符串。你可以调用对象的 toString 方法进行测试。

```
Math.toString(); // "[object Math]"
JSON.toString(); // "[object JSON]"
```

类似地，对象也可以通过其 valueOf 方法转换为数字。通过定义类似下面这些方法，你可以控制对象的类型转换。

```
"J" + { toString: function() { return "S"; } }; // "JS"
2 * { valueOf: function() { return 3; } };       // 6
```

再一次，当你认识到运算符 + 被重载来实现字符串连接和加法时，事情变得棘手起来。特别是，当一个对象同时包含 toString 和 valueOf 方法时，运算符 + 应该调用哪个方法并不明显——做字符串连接还是加法应该根据参数的类型，但是存在隐式的强制转换，因此类型并不是显而易见！ JavaScript 通过盲目地选择 valueOf 方法而不是 toString 方法来解决这种含糊的情况。但是，这就意味着如果有人打算对一个对象执行字符串连接操作，那么产生的行为将会出乎意料。

```
var obj = {
    toString: function() {
        return "[object MyObject]";
    },
    valueOf: function() {
        return 17;
    }
};
"object: " + obj; // "object: 17"
```

这个例子的说明，valueOf 方法才真正是为那些代表数值的对象（如 Number 对象）而设计的。对于这些对象，toString 和 valueOf 方法应返回一致的结果（相同数字的字符串或数值表示），因此，不管是对象的连接还是对象的相加，重载的运算符 + 总是一致的行为。一般情况下，字符串的强制转换远比数字的强制转换更常见、更有用。最好避免使用 valueOf 方法，除非对象的确是一个数字的抽象，并且 obj.toString() 能产生一个 obj.valueOf() 的字符串表示。

最后一种强制转换有时称为真值运算（truthiness）。if、|| 和 && 等运算符逻辑上需要布尔值作为操作参数，但实际上可以接受任何值。JavaScript 按照简单的隐式强制转换规则将值解释为布尔值。大多数的 JavaScript 值都为真值（truthy），也就是能隐式地转换为 true。对于字符串和数字以外的其他对象，真值运算不会隐式调用任何强制转换方法。JavaScript 中有 7 个假值：false、0、-0、""、NaN、null 和 undefined。其他所有的值都为真值。由于数字和字符串可能为假值，因此，使用真值运算检查函数参数或者对象属性是否已定义不是绝对安全的。例如，一个带有默认值的接受可选参数的函数：

```
function point(x, y) {
    if (!x) {
        x = 320;
    }
    if (!y) {
        y = 240;
    }
    return { x: x, y: y };
}
```

此函数忽略任何为假值的参数，包括 0：

```
point(0, 0); // { x: 320, y: 240 }
```

检查参数是否为 undefined 更为严格的方式是使用 typeof。

```
function point(x, y) {
    if (typeof x === "undefined") {
        x = 320;
    }
    if (typeof y === "undefined") {
        y = 240;
    }
    return { x: x, y: y };
}
```

此版本的 point 函数可以正确地识别 0 和 undefined。

```
point();      // { x: 320, y: 240 }
point(0, 0); // { x: 0, y: 0 }
```

另一种方式是与 undefined 进行比较。

```
if (x === undefined) { ... }
```

第 54 条将讨论针对库和 API 设计的真值运算测试的影响。

提示

- 类型错误可能被隐式的强制转换所隐藏。
- 重载的运算符 + 是进行加法运算还是字符串连接操作取决于其参数类型。
- 对象通过 valueOf 方法强制转换为数字，通过 toString 方法强制转换为字符串。
- 具有 valueOf 方法的对象应该实现 toString 方法，返回一个 valueOf 方法产生的数字的字符串表示。
- 测试一个值是否为未定义的值，应该使用 typeof 或者与 undefined 进行比较而不是使用真值运算。

第 4 条：原始类型优于封装对象

除了对象之外，JavaScript 有 5 个原始值类型：布尔值、数字、字符串、null 和 undefined。（令人困惑的是，对 null 类型进行 typeof 操作得到的结果为“object”，然而，ECMAScript 标准描述其为一个独特的类型。）同时，标准库提供了构造函数来封装布尔值、数字和字符串作为对象。你可以创建一个 String 对象，该对象封装了一个字符串值。

```
var s = new String("hello");
```

在某些方面，String 对象的行为与其封装的字符串值类似。你可以通过将它与另一个值连接来创建字符串。

```
s + " world"; // "hello world"
```

你也可以提取其索引的子字符串。

```
s[4]; // "o"
```

但是不同于原始的字符串，String 对象是一个真正的对象。

```
typeof "hello"; // "string"
typeof s;       // "object"
```

这是一个重要的区别，因为这意味着你不能使用内置的操作符来比较两个截然不同的 String 对象的内容。

```
var s1 = new String("hello");
var s2 = new String("hello");
s1 === s2; // false
```

由于每个 String 对象都是一个单独的对象，其总是只等于自身。对于非严格相等运算符，结果同样如此。

```
s1 == s2; // false
```

由于这些封装的行为并不十分正确，所以用处不大。其存在的主要理由是它们的实用方法。结合另外的隐式强制转换，JavaScript 使得我们可以方便地使用这些实用方法因为这里有另一个隐式转换：当对原始值提取属性和进行方法调用时，它表现得就像已经使用了对应的对象类型封装了该值一样。例如，String 的原型对象有一个 toUpperCase 方法，可以将字符串转换为大写。你可以对原始字符串值调用这个方法。

```
"hello".toUpperCase(); // "HELLO"
```

这种隐式封装的一个奇怪后果是你可以对原始值设置属性，但是对其丝毫没有影响。

```
"hello".someProperty = 17;
"hello".someProperty; // undefined
```

因为每次隐式封装都会产生一个新的 String 对象，更新第一个封装对象并不会造成持久的影响。对原始值设置属性的确是没有意义的，但是觉察到这种行为是值得的。事实证明，这是 JavaScript 隐藏类型错误的又一种情形。本来你想给一个对象设置属性，但没注意其实它是个原始值，程序只是忽略更新而继续运行。这容易导致一些难以发现的错误，并且难以诊断。

- 当做相等比较时，原始类型的封装对象与其原始值行为不一样。
- 获取和设置原始类型值的属性会隐式地创建封装对象。

第 5 条：避免对混合类型使用 == 运算符

你认为下面表达式的值是什么？

```
"1.0e0" == { valueOf: function() { return true; } };
```

对这两个看似无关的值使用 == 运算符实际上是相等的。就像第 3 条描述的隐式强制转换一样，在比较之前，它们都被转换为数字。字符串“1.0e0”被解析为数字 1，而匿名对象也通过调用其自身的 valueOf 方法得到结果 true，然后再转换为数字，得到 1。

很容易使用这些强制转换完成一些工作。例如，从一个 Web 表单读取一个字段并与一个数字进行比较。

```
var today = new Date();

if (form.month.value == (today.getMonth() + 1) &&
    form.day.value == today.getDate()) {
    // happy birthday!
    // ...
}
```

但实际上，它只是显式地使用 Number 函数或者一元运算符 + 将值转换为数字。

```
var today = new Date();

if (+form.month.value == (today.getMonth() + 1) &&
    +form.day.value == today.getDate()) {
    // happy birthday!
    // ...
}
```

上面这段代码更加清晰，因为它向读者传达了代码到底在做什么样的转换，而不要求读者记住这些转换规则。一个更好的替代方法是使用严格相等运算符。

```
var today = new Date();

if (+form.month.value === (today.getMonth() + 1) && // strict
    +form.day.value === today.getDate()) {          // strict
    // happy birthday!
    // ...
}
```

当两个参数属于同一类型时，== 和 === 运算符的行为是没有区别的。因此，如果你知道参数属于同一类型，那么，== 和 === 运算符可以互换。但最好使用严格相等运算符，因为读者会非常清晰地知道：在比较操作中并没有涉及任何转换。否则，你需要读者准确地记住这些强制转换规则以解读代码的行为。

事实上，这些强制转换规则一点也不明显。表 1.1 包含了 == 运算符针对不同类型参数的强制转换规则。这些规则具有对称性。例如，第一条规则既适用于 null == undefined，也适用于 undefined == null。在很多时候，这些转换都试图产生数字。但当它们处理对象时会变得难以捉摸。操作符试图将对象转换为原始值，可通过调用对象的 valueOf 和 toString 方法而实现。更令人难以捉摸的是，Date 对象以相反的顺序尝试调用这两个方法。

表 1.1 == 运算符的强制转换规则

参数类型 1	参数类型 2	强制转换
null	undefined	不转换，总是返回 true
null 或 undefined	其他任何非 null 或 undefined 的类型	不转换，总是返回 false
原始类型：string、number 或 boolean	Date 对象	将原始类型转换为数字；将 Date 对象转换为原始类型（优先尝试 toString 方法，再尝试 valueOf 方法）
原始类型：string、number 或 boolean	非 Date 对象	将原始类型转换为数字；将非 Date 对象转换为原始类型（优先尝试 valueOf 方法，再尝试 toString 方法）
原始类型：string、number 或 boolean	原始类型：string、number 或 boolean	将原始类型转换为数字

== 运算符将数据以不同的表现呈现出来，这种纠错有时称为“照我的意思去做”（do what I mean）的语义。但计算机并不能真正地了解你的心思。世界上有太多的数据表现形式，JavaScript 需要知道你使用的是哪种。例如，你可能希望你能将一个包含日期的字符串和一个 Date 对象进行比较。

```
var date = new Date("1999/12/31");
date == "1999/12/31"; // false
```

这个例子失败是因为 Date 对象被转换成一种不同格式的字符串，而不是本例所采用的格式。

```
date.toString(); // "Fri Dec 31 1999 00:00:00 GMT-0800 (PST)"
```

但是，这种错误是一个更普遍的强制转换误解的“症状”。== 运算符并不能推断和统一所有的数据格式。它需要你和读者都能理解其微妙的强制转换规则。更好的策略是显式自定义应用程序转换的逻辑，并使用严格相等运算符。

```
function toYMD(date) {
    var y = date.getYear() + 1900, // year is 1900-indexed
        m = date.getMonth() + 1,   // month is 0-indexed
        d = date.getDate();
    return y
         + "/" + (m < 10 ? "0" + m : m)
         + "/" + (d < 10 ? "0" + d : d);
}
toYMD(date) === "1999/12/31"; // true
```

显式地定义转换的逻辑能确保你不会混淆 == 运算符的强制转换规则，而且免除了读者不得不查找或记住这些规则的麻烦。

提示

- 当参数类型不同时，== 运算符应用了一套难以理解的隐式强制转换规则。
- 使用 === 运算符，使读者不需要涉及任何的隐式强制转换就能明白你的比较运算。
- 当比较不同类型的值时，使用你自己的显式强制转换使程序的行为更清晰。

第 6 条：了解分号插入的局限

JavaScript 的一个便利是能够离开语句结束分号工作。删除分号后，结果变得轻盈而优雅。

```
function Point(x, y) {
    this.x = x || 0
    this.y = y || 0
}

Point.prototype.isOrigin = function() {
    return this.x === 0 && this.y === 0
}
```

上面的代码能工作多亏 JavaScript 的自动分号插入（automatic semicolon insertion）技术，它是一种程序解析技术。它能推断出某些上下文中省略的分号，然后有效地自动地将分号“插入”到程序中。ECMAScript 标准细心地制定了分号插入机制，因此，可选分号可以在不同的 JavaScript 引擎之间移植。

但是同第 3 条和第 5 条的隐式强制转换一样，分号插入也有其陷阱，你根本不能避免学习其规则。即使你从来不省略分号，受分号插入的影响，JavaScript 语法也有一些额外的限制。好消息是，一旦你学会分号插入的规则，你会发现你能从删除不必要的分号的痛苦中解脱出来。

分号插入的第一条规则：

分号仅在 | 标记之前、一个或多个换行之后和程序输入的结尾被插入。

换句话说，你只能在一行、一个代码块和一段程序结束的地方省略分号。因此，下面的函数定义是合法的。

```
function square(x) {
    var n = +x
    return n * n
}
function area(r) { r = +r; return Math.PI * r * r }
function add1(x) { return x + 1 }
```

但是，下面这个却不合法。

```
function area(r) { r = +r return Math.PI * r * r } // error
```

分号插入的第二条规则：

分号仅在随后的输入标记不能解析时插入。

换句话说，分号插入是一种错误校正机制。下面这段代码作为一个简单的例子。

```
a = b
(f());
```

能正确地解析为一条单独的语句，等价于：

```
 a = b(f());
```

也就是说，没有分号插入。与此相反，下面这段代码：

```
a = b
f();
```

被解析为两条独立的语句，因为

```
a = b f();
```

解析有误。

这条规则有一个不幸的影响：你总是要注意下一条语句的开始，从而发现你是否能合法地省略分号。如果某条语句的下一行的初始标记不能被解析为一个语句的延续，那么，你不能省略该条语句的分号。

有 5 个明确有问题的字符需要密切注意：(、[、+、-、和 /。每一个字符都能作为一个表

达式运算符或者一条语句的前缀，这依赖于具体上下文。因此，要小心提防那些以表达式结束的语句，就像上面的赋值语句一样。如果下一行以这 5 个有问题的字符之一开始，那么不会插入分号。到目前为止，最常见的情况是以一个括号开始，就像上面的例子。另一种常见的情况是数组字面量。

```
a = b
["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

这看起来像两条语句。一条赋值语句，紧接着一条按序对字符“r”、“g”和“b”调用函数的语句。但是由于该语句以“[”开始，它被解析为一条语句，等价于：

```
a = b["r", "g", "b"].forEach(function(key) {
    background[key] = foreground[key] / 2;
});
```

这个中括号表达式看起来有点怪，请记住 JavaScript 允许逗号分隔表达式。逗号分隔表达式从左至右依次执行，并返回最后一个表达式的值。对于该例子，它返回字符“b”。

+、- 和 / 字符出现在语句开始并不常见，但也不是闻所未闻。字符“/”有一种非常微妙的情况：它出现在语句的开始实际上不是一个入口标记，而是作为正则表达式标记的开始。

```
/Error/i.test(str) && fail();
```

该语句使用一个不区分大小写的正则表达式 /Error/i 来匹配字符串。如果找到一个匹配，就会调用 fail 函数。但是，如果这段出现在一个未终止的赋值语句之后，例如：

```
a = b
/Error/i.test(str) && fail();
```

那么，这段代码会被解析为一条语句，等价于：

```
a = b / Error / i.test(str) && fail();
```

换句话说，初始的 / 字符被解析为除法运算符！

想省略分号时，有经验的 JavaScript 程序员会在该语句的后面跟一个声明，以保证该语句不会被错误地解析。在重构代码时，他们也会非常小心。例如，一个完全正确的程序，省略了 3 个可推断的分号 。

```
a = b     // semicolon inferred
var x     // semicolon inferred
(f())     // semicolon inferred
```

有可能被出人意料地改成只有两个可推断分号的程序。

```
var x     // semicolon inferred
a = b     // no semicolon inferred
(f())     // semicolon inferred
```

即使把 var 语句提前，这两段程序也应该是等价的（变量作用域的详细信息，请参见第 12 条），但事实是，b 后面跟着一个括号，程序被错误地解析为：

```
var x;
a = b(f());
```

其结果是你总需要注意省略分号，并且检查接下来的一行开始的标记是否会禁用自动插入分号。或者，你也可以采用在 (、[、+、- 和 / 字符的开始前置一个额外的分号语句的方法。例如，前面的例子可以改为下面的代码以保护括号中的函数调用。

```
a = b     // semicolon inferred
var x     // semicolon on next line
;(f())    // semicolon inferred
```

现在，把 var 声明语句移至行首是安全的，而不用担心改变这段程序。

```
var x     // semicolon inferred
a = b     // semicolon on next line
;(f())    // semicolon inferred
```

另一个常见的情况是，省略分号可能导致脚本连接问题（参见第 1 条）。每个文件可能由大量的函数调用表达式组成（参见第 13 条更多关于立即调用的函数表达式的信息）。

```
// file1.js
(function() {
    // ...
})()

// file2.js
(function() {
    // ...
})()
```

当每个文件作为一个单独的程序加载时，分号能自动地插入到末尾，将函数调用转变为一条语句。但是，当这些文件以下面的方式进行连接时：

```
(function() {
    // ...
})()
(function() {
    // ...
})()
```

结果被视为一条单一的语句，等价于：

```
(function() {
    // ...
})()(function() {
    // ...
})();
```

结果是：省略语句的分号不仅需要当心当前文件的下一个标记，而且还需要当心脚本连接后可能出现在语句之后的任一标记。类似上述方法，你可以防御性地为每个文件前缀一个额外的分号以保护脚本免受粗心连接的影响。如果文件最开始的语句以这 5 个脆弱的字符 (、[、+、- 和 / 开头，你就应该这么做。

```
// file1.js
;(function() {
    // ...
})()

// file2.js
;(function() {
    // ...
})()
```

这会确保即使前一个文件忽略了最后的分号，合并后的结果仍然会被视为单独的语句。

```
;(function() {
    // ...
})()
;(function() {
    // ...
})()
```

当然，如果脚本连接程序能够自动地在文件之间增加额外的分号是更好的。但并不是所有的脚本连接工具都写得很好，因此，最安全的选择是防御性地增加分号。

此时，你可能会认为，“这是多余的担心。我从来就不省略分号，我会没事儿的。”事实并不是这样。也有一些情况，尽管不会出现解析错误，JavaScript 仍会强制地插入分号。这就是所谓的 JavaScript 语法限制产生式（restricted production），它不允许在两个字符之间出现换行。最危险的情况是 return 语句，在 return 关键字和其可选参数之间一定不能包含换行符。因此，语句

```
return { };
```

返回一个新对象，而下面这段代码

```
return
{ };
```

被解析为 3 条单独的语句，等价于：

```
return;
{ }
;
```

换句话说，return 关键字后的换行会强制自动地插入分号。该段代码被解析为不带参数的 return 语句，后接一个空的代码块和一条空语句。其他的限制产生式包括：

- throw 语句
- 带有显式标签的 break 或 continue 语句
- 后置自增或自减运算符

最后一条规则是为了消除如下代码的歧义：

```
a
++
b
```

因为自增运算符既可以作为前置运算符也可以作为后置运算符，但是，后者不能出现在换行之前。这段代码被解析为：

```
a; ++b;
```

第三条也是最后一条分号插入规则：

分号不会作为分隔符在 for 循环空语句的头部被自动插入。

这就意味着你必须在 for 循环头部显式地包含分号。否则，类似下面的代码

```
for (var i = 0, total = 1 // parse error
     i < n
     i++) {
    total *= i
}
```

将会导致解析错误。空循环体的 while 循环同样也需要显式的分号。否则，省略分号也会导致解析错误：

```
function infiniteLoop() { while (true) } // parse error
```

因此，这就是一种需要分号的情况。

```
function infiniteLoop() { while (true); }
```

提示

- 仅在“}”标记之前、一行的结束和程序的结束处推导分号。
- 仅在紧接着的标记不能被解析的时候推导分号。
- 在以 (、[、+、- 或 / 字符开头的语句前绝不能省略分号。
- 当脚本连接的时候，在脚本之间显式地插入分号。
- 在 return、throw、break、continue、++ 或 -- 的参数之前绝不能换行。
- 分号不能作为 for 循环的头部或空语句的分隔符而被推导出。

第 7 条：视字符串为 16 位的代码单元序列

Unicode 有一个声誉，就是其复杂性。尽管字符串无处不在，大多数程序员还是抱着乐

观的态度避免学习 Unicode。但是在概念层面，它没有什么可害怕的。Unicode 的基础非常简单。它为世界上所有的文字系统的每个字符单位分配了一个唯一的整数，该整数介于 0 和 1 114 111 之间，在 Unicode 术语中称为代码点（code point）。Unicode 与其他字符编码几乎没有任何不同（例如，ASCII）。然而不同的是，ASCII 将每个索引映射为唯一的二进制表示，但 Unicode 允许多个不同二进制编码的代码点。不同的编码在要求存储的字符串数量和操作速度（如索引到某个字符串）之间进行权衡。目前有多种 Unicode 的编码标准，最流行的几个是：UTF-8、UTF-16 和 UTF-32。

进一步使情况复杂的是，Unicode 的设计师根据历史的数据，错误估算了代码点的容量范围。人们起初认为 Unicode 最多只需要 2^{16} 个代码点，所以产生了 UCS-2，其为 16 位编码的原始标准。这是一个特别有吸引力的选择。由于每个代码点可以容纳一个 16 位的数字，所以简单的方法就是将代码点与其编码元素一对一地映射起来，这称为一个代码单元（code unit）。也就是说，UCS-2 是由独立的 16 位的代码单元组成的，每个代码单元对应一个单独的 Unicode 代码点。这种编码方式的主要好处在于索引字符串是一种代价小的、固定时间的操作。获取某个字符串的第 n 个代码点只是简单地选取数组的第 n 个 16 位元素。图 1.1 显示了一个字符串例子。这些字符仅由最初的 16 位范围中的代码点组成。正如你看到的一样，对于 Unicode 的字符串，编码元素和代码点能完全的匹配。

其结果是，当时许多平台都采用 16 位编码的字符串。Java 便是其中之一，JavaScript 也紧随其后，所以 JavaScript 字符串的每个元素都是一个 16 位的值。现在，如果 Unicode 还是保持 20 世纪 90 年代初的做法，那么 JavaScript 字符串的每个元素仍然对应一个单独的代码点。

16 位的范围是相当大的，囊括了世界上的大多数文字系统，这比 ASCII 或其无数的历史替代者都要多。即便如此，Unicode 也需要及时扩大其最初的范围，标准从当时的 2^{16} 扩展到了超过 2^{20} 个代码点。新增加的范围被组织为 17 个大小为 2^{16} 代码点的子范围。第一个子范围，称为基本多文种平面（Basic Multilingual Plane, BMP），包含最初的 2^{16} 个代码点。余下的 16 个范围称为辅助平面（supplementary plane）。

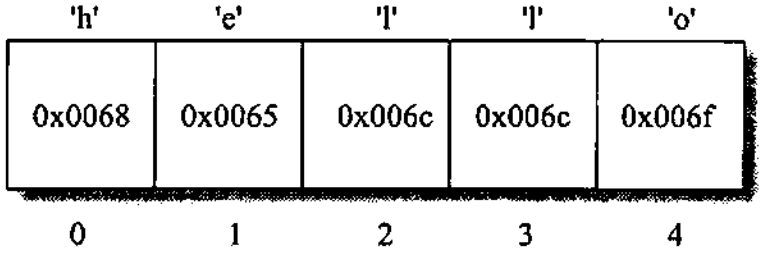

图 1.1 一个只包含来自基本多文种平面的代码点的 JavaScript 字符串

一旦代码点的范围扩展，UCS-2 就变得过时了。它需要通过扩展来表示这些附加的代码点。其替代者 UTF-16 与之类似，但 UTF-16 采用代理对表示附加的代码点。一对 16 位的代

码单元共同编码一个等于或大于 2^{16} 的代码点。例如，分配给高音谱号的音乐符号（“𝄞”）的代码点为 U+1D11E（代码点数 119 070 的 Unicode 的惯用 16 进制写法）。其由 UTF-16 格式的代码单元 0xd834 和 0xddle 共同表示。可以通过合并这两个代码单元选择的位来对这个代码点进行解码。（巧妙的是，这种编码保证了这些代理对绝不会与有效的 BMP 代码点混淆，因此，甚至从字符串中间的某个位置进行搜索，你也可以随时识别一个代理对。）在图 1.2 中你可以看到一个含有代理对的字符串的例子。该字符串的第一个代码点需要一个代理对，从而导致了代码单元的索引与代码点的索引不同。

由于 UTF-16 的每个代码点编码需要一个或两个 16 位的代码单元，因此 UTF-16 是一种可变长度的编码。长度为 n 的字符串在内存中的大小变化基于该字符串特定的代码点。此外，查找字符串的第 n 个代码点不再是一个固定时间的操作，因为它一般需要从字符串的开始处进行搜索。

但是当 Unicode 扩大规模时，JavaScript 已经采用了 16 位的字符串元素。字符串属性和方法（如 length、charAt 和 charCodeAt）都是基于代码单元层级，而不是代码点层级。所以每当字符串包含辅助平面中的代码点时，JavaScript 将每个代码点表示为两个元素而不是一个（一对 UTF-16 代理对的代码点）。简单地说，一个 JavaScript 字符串的元素是一个 16 位的代码单元。

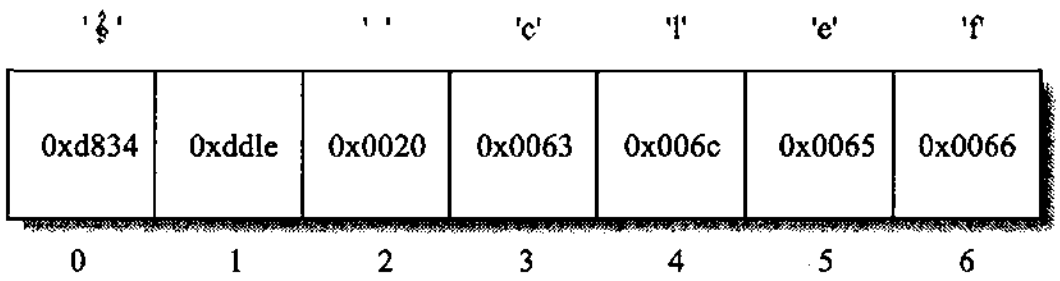

图 1.2 一个包含来自辅助平面的代码点的 JavaScript 字符串

JavaScript 引擎可以在内部优化字符串内容的存储。但是考虑到字符串的属性和方法，字符串表现得就像 UTF-16 的代码单元序列。正如图 1.2 中的字符串，尽管事实上只包含 6 个代码点，但是 JavaScript 报告它的长度为 7。

```
"𝄞 clef".length; // 7
"G clef".length; // 6
```

提取该字符串的某个字符得到的是代码单元，而不是代码点。

```
"𝄞 clef".charCodeAt(0);      // 55348 (0xd834)
"𝄞 clef".charCodeAt(1);      // 56606 (0xdd1e)
"𝄞 clef".charAt(1) === " "; // false
"𝄞 clef".charAt(2) === " "; // true
```

类似地，正则表达式也工作于代码单元层级。其单字符模式（“.”）匹配一个单一的代

码单元。

```
/^.$/.test("𝄞");  // false
/^..$/.test("𝄞"); // true
```

这种状况意味着应用程序同 Unicode 的整个范围一起工作必须更加仔细。应用程序不能信赖字符串方法、长度值、索引查找或者许多正则表达式模式。如果你使用除 BMP 之外的代码点，那么求助于一些支持代码点的库是个好主意。正确地获取编码和解码的细节是相当棘手的，所以最好使用一个现存的库，而不是自己实现这些逻辑。

虽然 JavaScript 内置的字符串数据类型工作于代码单元层级，但这并不能阻止一些 API 意识到代码点和代理对。事实上，一些标准的 ECMAScript 库正确地处理了代理对，例如 URI 操作函数：sendcodeURI、decodeURI、encodeURIComponent 和 decodeURIComponent。每当一个 JavaScript 环境提供一个库操作字符串（例如，操作一个 Web 页面的内容或者执行关于字符串的 I/O 操作），你都需要查阅这些库文档，看它们如何处理 Unicode 代码点的整个范围。

- JavaScript 字符串由 16 位的代码单元组成，而不是由 Unicode 代码点组成。
- JavaScript 使用两个代码单元表示 2^{16} 及其以上的 Unicode 代码点。这两个代码单元被称为代理对。
- 代理对甩开了字符串元素计数，length、charAt、charCodeAt 方法以及正则表达式模式（例如“.”）受到了影响。
- 使用第三方的库编写可识别代码点的字符串操作。
- 每当你使用一个含有字符串操作的库时，你都需要查阅该库文档，看它如何处理代码点的整个范围。

第 2 章
变量作用域

作用域对于程序员来说就像氧气。它无处不在，甚至，你往往不会去想它。但当它被污染时，你会感觉到窒息。

好消息是 JavaScript 核心的作用域规则很简单。其作用域规则被精心设计，且强大得令人难以置信。但也有一些例外情况。有效地使用 JavaScript 需要掌握变量作用域的一些基本概念，并了解一些可能导致难以捉摸的、令人讨厌的问题的极端情况。

第 8 条：尽量少用全局对象

在 JavaScript 中很容易在全局命名空间中创建变量。创建全局变量毫不费力，因为它不需要任何形式的声明，而且能被整个程序的所有代码自动地访问。这种便利很容易诱惑初学者。然而经验丰富的程序员都知道应该避免使用全局变量。定义全局变量会污染共享的公共命名空间，并可能导致意外的命名冲突。全局变量不利于模块化，因为它会导致程序中独立组件间的不必要耦合。虽然“先这样写，以后再调整”（“code now and organize later”）可能比较方便，但优秀的程序员会不断地留意程序的结构、持续地归类相关的功能以及分离不相关的组件，并将这些行为作为编程过程中的一部分。

由于全局命名空间是 JavaScript 程序中独立的组件进行交互的唯一途径，因此，利用全局命名空间的情况是不可避免的。组件或程序库不得不定义一些全局变量，以便程序中的其他部分使用。否则，最好尽量使用局部变量。当然可以写一个只使用全局变量而不使用其他变量的程序，但那是自寻烦恼。即使在很简单的函数中将临时变量定义为全局的，我们都会担心是否有任何其他的代码可能会使用相同的变量名。

```
var i, n, sum; // globals
function averageScore(players) {
```

```
    sum = 0;
    for (i = 0, n = players.length; i < n; i++) {
        sum += score(players[i]);
    }
    return sum / n;
}
```

如果 score 函数出于自身的目的使用了任何同名的全局变量，averageScore 函数的定义将出现问题。

```
var i, n, sum; // same globals as averageScore!
function score(player) {
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

答案是保持这些变量为局部变量，仅将其作为需要使用它们的代码的一部分。

```
function averageScore(players) {
    var i, n, sum;
    sum = 0;
    for (i = 0, n = players.length; i < n; i++) {
        sum += score(players[i]);
    }
    return sum / n;
}

function score(player) {
    var i, n, sum;
    sum = 0;
    for (i = 0, n = player.levels.length; i < n; i++) {
        sum += player.levels[i].score;
    }
    return sum;
}
```

JavaScript 的全局命名空间也被暴露为在程序全局作用域中可以访问的全局对象，该对象作为 this 关键字的初始值。在 Web 浏览器中，全局对象被绑定到全局的 window 变量。添加或修改全局变量会自动更新全局对象。

```
this.foo; // undefined
foo = "global foo";
this.foo; // "global foo"
```

类似地，更新全局对象也会自动地更新全局命名空间：

```
var foo = "global foo";
this.foo = "changed";
foo; // "changed"
```

这意味着你创建一个全局变量有两种方法可供挑选。你可以在全局作用域内使用 var 声明它，或者将其加入到全局对象中。无论使用哪种方法都行，但是 var 声明的好处是更能清晰地表达全局变量在程序范围中的影响。鉴于引用未绑定的变量会导致运行时错误，因此，保持作用域清晰和简洁会使代码的使用者更容易理解程序声明了哪些全局变量。

虽然最好限制使用全局对象，但是它确实提供了一个不可或缺的特别用途。由于全局对象提供了全局环境的动态反应机制，所以可以使用它查询一个运行环境，检测在这个平台下哪些特性是可用的。例如，ES5 引入了一个全局的 JSON 对象来读写 JSON 格式的数据。将代码部署到一个不确定是否提供了 JSON 对象的环境时的一个权宜之计是，你可以测试这个全局对象是否存在并提供一个替代实现。

```
if (!this.JSON) {
    this.JSON = {
        parse: ...,
        stringify: ...
    };
}
```

如果你已经提供了 JSON 的实现，你当然可以简单无条件地使用自己的实现。但是由宿主环境提供的内置实现几乎总是更合适的。因为它们按照一定的标准对正确性和一致性进行了严格检查，并且普遍来说比第三方实现提供了更好的性能。

特性检测技术在 Web 浏览器中特别重要，因为在各种各样的浏览器和浏览器版本中可能会执行同样的代码。特性检测是一种使得程序在平台特性集合的变化中依旧健壮的相对简单的方法。这种技术也适用于其他地方。例如，此技术使得在浏览器和 JavaScript 服务器环境中共享程序库成为可能。

- 避免声明全局变量。
- 尽量声明局部变量。
- 避免对全局对象添加属性。
- 使用全局对象来做平台特性检测。

第 9 条：始终声明局部变量

如果存在比全局变量更麻烦的事，那就是意外的全局变量。遗憾的是，JavaScript 的变

量赋值规则使得意外地创建全局变量太容易了。程序中给一个未绑定的变量赋值将会简单地创建一个新的全局变量并赋值给它，而不是引发错误。这意味着，如果忘记将变量声明为局部变量，那么该变量将会被隐式地转变为全局变量。

```
function swap(a, i, j) {
    temp = a[i]; // global
    a[i] = a[j];
    a[j] = temp;
}
```

尽管该程序没有使用 var 声明 temp 变量，但是执行是不会出错的，只是会导致意外地创建一个全局变量。正确的实现应该使用 var 声明 temp 变量。

```
function swap(a, i, j) {
    var temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

故意地创建全局变量是不好的风格，而意外地创建全局变量将是彻头彻尾的灾难。正因如此，许多程序员使用 lint 工具检查程序源代码中的不好风格和潜在的错误。该工具通常具有报告未绑定变量使用情况的功能。通常情况下，lint 工具使用用户提供的一套已知的全局变量（例如，期望存在于宿主环境中的或在单独文件中定义的全局变量）检查未声明的变量，然后报告出所有既没有在列表中提供的又没有在程序中声明的引用或赋值变量。花一些时间去探索什么样的工具对 JavaScript 可用是值得的。将自动检查一些常见的错误（例如，意外的全局变量）整合到开发过程中可能会成为救命稻草。

- 始终使用 var 声明新的局部变量。
- 考虑使用 lint 工具帮助检查未绑定的变量。

第 10 条：避免使用 with

悲催的 with 特性。在 JavaScript 中可能没有比它更令人诟病的特性了。然而，with 语句是罪有应得。它提供的任何“便利”，都更让其变得不可靠和低效率。

with 语句的动机是可以理解的。程序经常需要对单个对象依次调用一系列方法。使用 with 语句可以很方便地避免对对象的重复引用：

```
function status(info) {
    var widget = new Widget();
    with (widget) {
```

```
            setBackground("blue");
            setForeground("white");
            setText("Status: " + info); // ambiguous reference
            show();
        }
    }
```

使用 with 语句从模块对象中“导入”(import) 变量也是很有诱惑力的。

```
function f(x, y) {
    with (Math) {
        return min(round(x), sqrt(y)); // ambiguous references
    }
}
```

在这两种情况下，使用 with 语句使得提取对象的属性，并将这些属性绑定到块的局部变量中变得非常诱人且容易。

这些例子看起来很有吸引力，但它实际没做它应该做的事。请注意这两个例子有两种不同类型的变量。一种是我们希望引用 with 对象的属性的变量，如 setBackground、round 以及 sqrt。另一种是我们希望引用外部变量绑定的变量，如 info、x 和 y。但其实在语法上并没有区分这两种类型的变量。它们都只是看起来像变量。

事实上，JavaScript 对待所有的变量都是相同的。JavaScript 从最内层的作用域开始向外查找变量。with 语句对待一个对象犹如该对象代表一个变量作用域，因此，在 with 代码块的内部，变量查找从搜索给定的变量名的属性开始。如果在这个对象中没有找到该属性，则继续在外部作用域中搜索。

图 2.1 显示了当执行 with 语句的代码时 ,status 函数的作用域在 JavaScript 引擎中的内部表示图。在 ES5 规范中这称为词法环境 (在旧版本标准中称为作用域链)。该词法环境的最内层作用域由 widget 对象提供。接下来的作用域用来绑定该函数的局部变量 info 和 widget。接下来一层绑定到 status 函数。注意在一个正常的作用域中，会有与局部作用域中的变量同样多的作用域绑定存储在与之对应的环境层级中。但是对于 with 作用域，绑定集合依赖于碰巧在给定时间点时的对象。

我们有多大的信心确信在提供给 with 的对象中可以找到哪些属性，或者找不到哪些属性？ with 块中的每个外部变量的引用都隐式地假设在 with 对象（以及它的任何原型对象）中没有同名的属性。而在程序的其他地方创建或修改 with 对象或其原型对象不一定会遵循这样的假设。JavaScript 引擎当然不会读取局部代码来获取你使用了哪些局部变量。

变量作用域和对象命名空间之间的冲突使得 with 代码块异常脆弱。例如，如果上述例子的 with 对象获得了一个名为 info 的属性，status 函数的行为将被立即改变。status 函数将使用这个属性而不是 status 函数的 info 参数。这种情况可能发生在源代码的演化中。例如，程

序员决定所有的 widget 对象都应该有一个 info 属性。更糟糕的是，有时会给 Widget 的原型对象在运行时加入 info 属性，这将导致 status 函数变得不可预测。

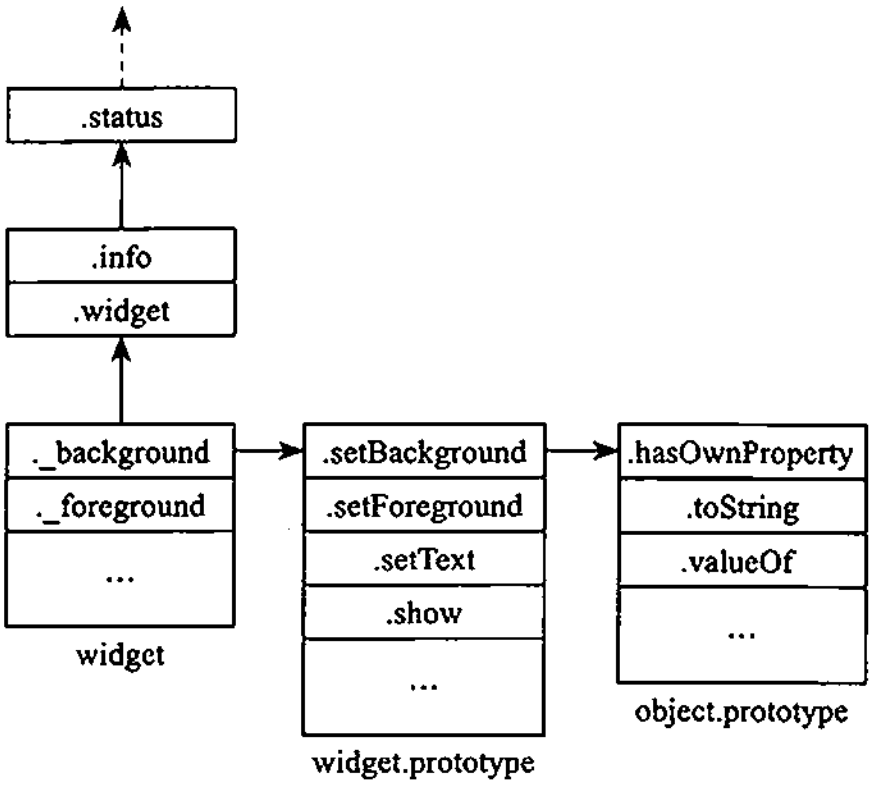

图 2.1 status 函数的词法环境（又称作用域链）

```
status("connecting"); // Status: connecting
Widget.prototype.info = "[[widget info]]";
status("connected");  // Status: [[widget info]]
```

同样，如果某人添加名为 x 或 y 的属性到 Math 对象上，那么上述例子中的 f 函数也会出错。

```
Math.x = 0;
Math.y = 0;
f(2, 9); // 0
```

可能不会有人给 Math 添加 x 和 y 属性。但总是很难预测一个特定的对象是否已被修改，或是否可能拥有你不知道的属性。而事实证明，人力不可预测的特性对于优化编译器同样不可预测。通常情况下，JavaScript 作用域可被表示为高效的内部数据结构，变量查找会非常快速。但是由于 with 代码块需要搜索对象的原型链来查找 with 代码块里的所有变量，因此，其运行速度远远低于一般的代码块。

在 JavaScript 中没有单个特性能作为一个更好的选择直接替代 with 语句。在某些情况下，最好的替代方法是简单地将对象绑定到一个简短的变量名上。

```
function status(info) {
    var w = new Widget();
    w.setBackground("blue");
    w.setForeground("white");
```

```
    w.addText("Status: " + info);
    w.show();
}
```

该版本的行为更具可预测性。没有任何变量引用对于 w 对象的内容是敏感的。所以即使一些代码修改了 Widget 的原型对象，status 函数的行为依旧与预期一致。

```
status("connecting"); // Status: connecting
Widget.prototype.info = "[[widget info]]";
status("connected");  // Status: connected
```

在其他情况下，最好的方法是将局部变量显式地绑定到相关的属性上。

```
function f(x, y) {
    var min = Math.min, round = Math.round, sqrt = Math.sqrt;
    return min(round(x), sqrt(y));
}
```

再次，一旦消除 with 语句，函数的行为变得可以预测。

```
Math.x = 0;
Math.y = 0;
f(2, 9); // 2
```

提示

- 避免使用 with 语句。
- 使用简短的变量名代替重复访问的对象。
- 显式地绑定局部变量到对象属性上，而不要使用 with 语句隐式地绑定它们。

第 11 条：熟练掌握闭包

对于那些使用不支持闭包特性的编程语言的程序员来说，闭包可能是一个陌生的概念。初看起来，它们似乎令人生畏。但请放心，付出努力掌握闭包将会给你带来超值的回报。

幸运的是，闭包真没有什么可害怕的。理解闭包只需要学会三个基本的事实。第一个事实：JavaScript 允许你引用在当前函数以外定义的变量。

```
function makeSandwich() {
    var magicIngredient = "peanut butter";
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make("jelly");
}
makeSandwich(); // "peanut butter and jelly"
```

请注意内部的 make 函数是如何引用定义在外部 makeSandwich 函数内的 magicIngredient 变量的。

第二个事实：即使外部函数已经返回，当前函数仍然可以引用在外部函数所定义的变量。如果这听起来让人难以置信，请记住，JavaScript 的函数是第一类（first-class）对象（请参阅第 19 条）。这意味着，你可以返回一个内部函数，并在稍后调用它。

```
function sandwichMaker() {
    var magicIngredient = "peanut butter";
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var f = sandwichMaker();
f("jelly");        // "peanut butter and jelly"

f("bananas");      // "peanut butter and bananas"
f("marshmallows"); // "peanut butter and marshmallows"
```

这与第一个例子几乎完全相同。不同的是，不是在外部的 sandwichMaker 函数中立即调用 make（"jelly"），而是返回 make 函数本身。因此，f 的值为内部的 make 函数，调用 f 实际上是调用 make 函数。但即使 sandwichMaker 函数已经返回，make 函数仍能记住 magicIngredient 的值。

这是如何工作的？答案是：JavaScript 的函数值包含了比调用它们时执行所需要的代码还要多的信息。而且，JavaScript 函数值还在内部存储它们可能会引用的定义在其封闭作用域的变量。那些在其所涵盖的作用域内跟踪变量的函数被称为闭包。make 函数就是一个闭包，其代码引用了两个外部变量：magicIngredient 和 filling。每当 make 函数被调用时，其代码都能引用到这两个变量，因为该闭包存储了这两个变量。

函数可以引用在其作用域内的任何变量，包括参数和外部函数变量。我们可以利用这点来编写更加通用的 sandwichMaker 函数。

```
function sandwichMaker(magicIngredient) {
    function make(filling) {
        return magicIngredient + " and " + filling;
    }
    return make;
}
var hamAnd = sandwichMaker("ham");
hamAnd("cheese");        // "ham and cheese"
hamAnd("mustard");       // "ham and mustard"
var turkeyAnd = sandwichMaker("turkey");
```

```
turkeyAnd("Swiss");       // "turkey and Swiss"
turkeyAnd("Provolone");   // "turkey and Provolone"
```

该例子创建了 hamAnd 和 turkeyAnd 两个完全不同的函数。尽管它们都是由相同的 make 函数定义的，但是它们是两个截然不同的对象。第一个函数的 magicIngredient 的值为 "ham"，而第二个函数的 magicIngredient 的值为 "turkey"。

闭包是 JavaScript 最优雅、最有表现力的特性之一，也是许多惯用法的核心。JavaScript 甚至还提供了一种更为方便地构建闭包的字面量语法——函数表达式。

```
function sandwichMaker(magicIngredient) {
    return function(filling) {
        return magicIngredient + " and " + filling;
    };
}
```

请注意，该函数表达式是匿名的。由于我们只需要其能产生一个新的函数值，而不打算在局部调用它，因此根本没有必要给该函数命名。函数表达式也可以有名称（请参阅第 14 条）。

学习闭包的第三个也是最后一个事实：闭包可以更新外部变量的值。实际上，闭包存储的是外部变量的引用，而不是它们的值的副本。因此，对于任何具有访问这些外部变量的闭包，都可以进行更新。一个简单的惯用法 box 对象说明了这一切。它存储了一个可读写的内部值。

```
function box() {
    var val = undefined;
    return {
        set: function(newVal) { val = newVal; },
        get: function() { return val; },
        type: function() { return typeof val; }
    };
}
var b = box();
b.type(); // "undefined"
b.set(98.6);
b.get();  // 98.6
b.type(); // "number"
```

该例子产生了一个包含三个闭包的对象。这三个闭包是 set、get 和 type 属性。它们都共享访问 val 变量。set 闭包更新 val 的值，随后调用 get 和 type 查看更新的结果。

提示

- 函数可以引用定义在其外部作用域的变量。
- 闭包比创建它们的函数有更长的生命周期。

❑ 闭包在内部存储其外部变量的引用，并能读写这些变量。

第 12 条：理解变量声明提升

JavaScript 支持词法作用域（lexical scoping），即除了极少的例外，对变量 foo 的引用会被绑定到声明 foo 变量最近的作用域中。但是，JavaScript 不支持块级作用域，即变量定义的作用域并不是离其最近的封闭语句或代码块，而是包含它们的函数。

不明白 JavaScript 的这一特性将会导致一些微妙的 Bug，例如：

```
function isWinner(player, others) {
    var highest = 0;
    for (var i = 0, n = others.length; i < n; i++) {
        var player = others[i];
        if (player.score > highest) {
            highest = player.score;
        }
    }
    return player.score > highest;
}
```

该程序在 for 循环体内声明了一个局部变量 player。但是由于 JavaScript 中变量是函数级作用域 (function-scoped)，而不是块级作用域，所以在内部声明的 player 变量只是简单地重声明了一个已经存在于作用域内的变量（即参数 player)。该循环的每次迭代都会重写同一变量。因此，return 语句将 player 看作 others 的最后一个元素，而不是此函数最初的 player 参数。

理解 JavaScript 变量声明行为的一个好办法是把变量声明看作由两部分组成，即声明和赋值。JavaScript 隐式地提升 (hoists) 声明部分到封闭函数的顶部，而将赋值留在原地。换句话说，变量的作用域是整个函数，但仅在 var 语句出现的位置进行赋值。图 2.2 提供了变量声明提升的可视化图。

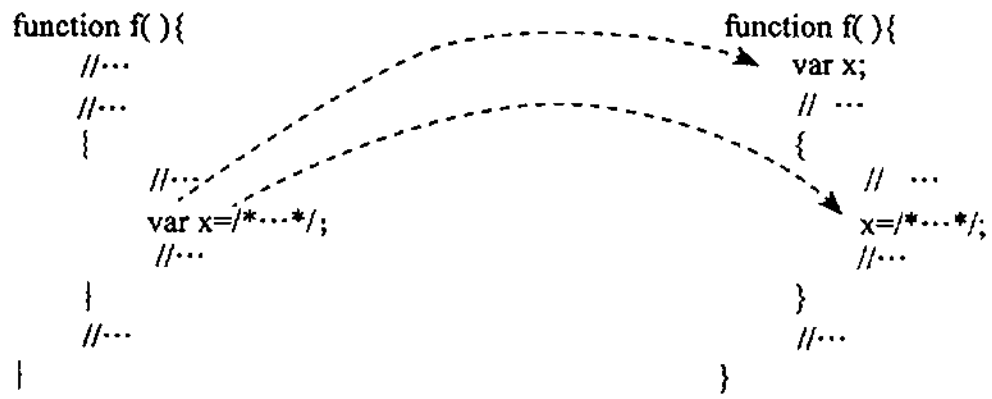

图 2.2 变量声明提升

变量声明提升也可能导致变量重声明的混淆。在同一函数中多次声明相同变量是合法

的。这在写多个循环时会经常出现。

```
function trimSections(header, body, footer) {
    for (var i = 0, n = header.length; i < n; i++) {
        header[i] = header[i].trim();
    }

    for (var i = 0, n = body.length; i < n; i++) {
        body[i] = body[i].trim();
    }
    for (var i = 0, n = footer.length; i < n; i++) {
        footer[i] = footer[i].trim();
    }
}
```

trimSections 函数好像声明了 6 个局部变量（3 个变量 i,3 个变量 n），但经过变量声明提升后其实只声明了 2 个。换句话说，经过变量声明提升后，trimSections 函数等同于下面这个重写的版本。

```
    for (var i = 0, n = body.length; i < n; i++) {
        body[i] = body[i].trim();
    }
    for (var i = 0, n = footer.length; i < n; i++) {
        footer[i] = footer[i].trim();
    }
}
```

因为重声明会导致截然不同的变量展现，一些程序员喜欢通过有效地手动提升变量将所有的 var 声明放置在函数的顶部，从而避免歧义。无论你是否喜欢这种风格，重要的是，不管是写代码还是读代码，都要理解 JavaScript 的作用域规则 。

JavaScript 没有块级作用域的一个例外恰好是其异常处理。try...catch 语句将捕获的异常绑定到一个变量，该变量的作用域只是 catch 语句块。

```
function test() {
    var x = "var", result = [];
    result.push(x);
    try {
        throw "exception";
    } catch (x) {
        x = "catch";
    }
    result.push(x);
    return result;
}
test(); // ["var", "var"]
```

提示

- 在代码块中的变量声明会被隐式地提升到封闭函数的顶部。
- 重声明变量被视为单个变量。
- 考虑手动提升局部变量的声明，从而避免混淆。

第 13 条：使用立即调用的函数表达式创建局部作用域

这段程序（Bug 程序）输出什么？

```
function wrapElements(a) {
    var result = [], i, n;
    for (i = 0, n = a.length; i < n; i++) {
        result[i] = function() { return a[i]; };
    }
    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // ?
```

程序员可能希望这段程序输出 10，但实际上它输出 undefined 值。

搞清楚该例子的方法是理解绑定与赋值的区别。在运行时进入一个作用域，JavaScript 会为每一个绑定到该作用域的变量在内存中分配一个“槽”（slot）。wrapElements 函数绑定了三个局部变量：result、i 和 n。因此，当它被调用时，wrapElements 函数会为这三个变量分配“槽”。在循环的每次迭代中，循环体都会为嵌套函数分配一个闭包。该程序的 Bug 在于这样一个事实：程序员似乎期望该函数存储的是嵌套函数创建时变量 i 的值。但事实上，它存储的是变量 i 的引用。由于每次函数创建后变量 i 的值都发生了变化，因此内部函数最终看到的是变量 i 最后的值。值得注意的是，闭包存储的是其外部变量的引用而不是值。

所以，所有由 wrapElements 函数创建的闭包都引用在循环之前创建的变量 i 的同一个共享“槽”。由于每次循环迭代都递增变量 i 直到运行到数组结束，因此，这时候其实当我们调用其中任何一个闭包时，它都会查找数组的索引 5 并返回 undefined 值。

请注意，即使我们把 var 声明置于 for 循环的头部，wrapElements 函数的表现也完全一样。

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
        result[i] = function() { return a[i]; };
```

```
    }
    return result;
}

var wrapped = wrapElements([10, 20, 30, 40, 50]);
var f = wrapped[0];
f(); // undefined
```

这个版本看起来更具欺骗性，因为 var 声明出现在了循环体中。但一如既往，变量声明会被提升到循环的上方。再一次，变量 i 只被分配了一个“槽”。

解决的办法是通过创建一个嵌套函数并立即调用它来强制创建一个局部作用域。

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {

      (function() {
          var j = i;
          result[i] = function() { return a[j]; };
      })();
  }
    return result;
}
```

这种技术被称为立即调用的函数表达式，或 IIFE（发音为“iffy”）。它是一种不可或缺的解决 JavaScript 缺少块级作用域的方法。另一种变种是将作为形参的局部变量绑定到 IIFE 并将其值作为实参传入。

```
function wrapElements(a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
       (function(j) {
           result[i] = function() { return a[j]; };
       })(i);
    }
    return result;
}
```

然而，使用 IIFE 来创建局部作用域要小心，因为在函数中包裹代码块可能会导致代码块发生一些微妙的变化。首先，代码块不能包含任何跳出块的 break 语句和 continue 语句。因为在函数外使用 break 或 continue 是不合法的。其次，如果代码块引用了 this 或特别的 arguments 变量，IIFE 将会改变它们的含义。第 3 章将讨论与 this 和 arguments 变量一起工作的技术。

- 理解绑定与赋值的区别。
- 闭包通过引用而不是值捕获它们的外部变量。
- 使用立即调用的函数表达式（IIFE）来创建局部作用域。
- 当心在立即调用的函数表达式中包裹代码块可能改变其行为的情形。

第 14 条：当心命名函数表达式笨拙的作用域

JavaScript 函数无论放在何处看起来似乎都是一样的，但是根据上下文其含义会发生变化。请看以下代码片段。

```
function double(x) { return x * 2; }
```

这段代码可以是一个函数声明，也可以是一个命名函数表达式(named function expression)，这取决于它出现的地方。这个声明是如此熟悉，它定义一个函数并绑定到当前作用域的一个变量。例如，在程序的最顶层，以上的声明将创建一个名为 double 的全局函数。但是同一段函数代码也可以作为一个表达式，它可以有截然不同的含义。例如：

```
var f = function double(x) { return x * 2; };
```

根据 ECMAScript 规范，此语句将该函数绑定到变量 f，而不是变量 double。当然，给函数表达式命名并不是必要的。我们可以使用匿名的函数表达式形式：

```
var f = function(x) { return x * 2; };
```

匿名和命名函数表达式的官方区别在于后者会绑定到与其函数名相同的变量上，该变量将作为该函数内的一个局部变量。这可以用来写递归函数表达式。

```
var f = function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
           find(tree.right, key);
};
```

注意，变量 find 的作用域只在其自身函数中。不像函数声明，命名函数表达式不能通过其内部的函数名在外部被引用。

```
find(myTree, "foo"); // error: find is not defined
```

使用命名函数表达式进行递归似乎没有必要，因为使用外部作用域的函数名也可达到同样的效果：

```
var f = function(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return f(tree.left, key) ||
           f(tree.right, key);
};
```

或者我们可以只使用一个声明。

```
function find(tree, key) {
    if (!tree) {
        return null;
    }
    if (tree.key === key) {
        return tree.value;
    }
    return find(tree.left, key) ||
           find(tree.right, key);
}
var f = find;
```

命名函数表达式真正的用处是进行调试。大多数现代的 JavaScript 环境都提供对 Error 对象的栈跟踪功能。在栈跟踪中，函数表达式的名称通常作为其入口使用。用于检查栈的设备调试器对命名函数表达式有类似的使用。

遗憾的是，命名函数表达式是作用域和兼容性问题臭名昭著的来源。这要归结于在 ECMAScript 规范的历史中很不幸的错误以及流行的 JavaScript 引擎中的 Bug。规范的错误在 ES3 中已经存在，JavaScript 引擎被要求将命名函数表达式的作用域表示为一个对象，这有点像有问题的 with 结构。该作用域对象只含有单个属性，该属性将函数名和函数自身绑定起来。该作用域对象也继承了 Object.prototype 的属性。这意味着仅仅是给函数表达式命名也会将 Object.prototype 中的所有属性引入到作用域中。结果可能会出人意料：

```
var constructor = function() { return null; };
var f = function f() {
    return constructor();
};
f(); // {} (in ES3 environments)
```

该程序看起来会产生 null，但其实会产生一个新的对象。因为命名函数表达式在其作用

域内继承了 Object.prototype.constructor（即 Object 的构造函数）。就像 with 语句一样，这个作用域会因 Object.prototype 的动态改变而受到影响。程序的一部分可能添加或删除 Object.prototype 属性，命名函数表达式中的所有变量都会受到影响。

幸运的是，ES5 修正了这个错误。但是一些 JavaScript 环境仍然使用过时的对象作用域。更糟的是，有些环境甚至更不符合标准，而且甚至对匿名函数表达式使用对象作为作用域。即使删除上述例子中的函数表达式名也会产生一个对象，而不是预期结果 null.

```
var constructor = function() { return null; };
var f = function() {
    return constructor();
};
f(); // {} (in nonconformant environments)
```

在系统中避免对象污染函数表达式作用域的最好方式是避免任何时候在 Object.prototype 中添加属性，以及避免使用任何与标准 Object.protoype 属性同名的局部变量。

在流行的 JavaScript 引擎中的另一个缺陷是对命名函数表达式的声明进行提升。例如：

```
var f = function g() { return 17; };
g(); // 17 (in nonconformant environments)
```

需要明确的是，这是不符合标准的行为。更糟的是，一些 JavaScript 环境甚至把 f 和 g 这两个函数作为不同的对象，从而导致不必要的内存分配。这种行为的一个合理的解决办法是创建一个与函数表达式同名的局部变量并赋值为 null。

```
var f = function g() { return 17; };
var g = null;
```

即使在没有错误地提升函数表达式声明的环境中，使用 var 重声明变量能确保仍然会绑定变量 g。设置变量 g 为 null 能确保重复的函数可以被垃圾回收。

当然可以得出合理的结论：命名函数表达式由于会导致很多问题，所以并不值得使用。一个不太严肃的回应是在开发阶段使用命名函数表达式用作调试，在发布前通过预处理程序将所有的函数表达式转为匿名的。但有一条是肯定的，你应当总是明确发布的平台（请参阅第 1 条）。你可能做的最糟的事情是为了支持那些甚至没有必要支持的平台将代码弄得一团糟。

提示

- 在 Error 对象和调试器中使用命名函数表达式改进栈跟踪。
- 在 ES3 和有问题的 JavaScript 环境中谨记函数表达式作用域会被 Object.prototype 污染。
- 谨记在错误百出的 JavaScript 环境中会提升命名函数表达式声明，并导致命名函数表

达式的重复存储。

- 考虑避免使用命名函数表达式或在发布前删除函数名。
- 如果你将代码发布到正确实现的 ES5 环境中，那么你没有什么好担心的。

第 15 条：当心局部块函数声明笨拙的作用域

我们继续讨论关于上下文敏感的传奇故事：嵌套函数声明。当你知道没有标准的方法在局部块里声明函数时，你可能会感到惊讶。然而现在，这是完全合法的，而且人们习惯于在另一个函数的顶部嵌套函数声明。

```
function f() { return "global"; }

function test(x) {
    function f() { return "local"; }

    var result = [];
    if (x) {
        result.push(f());
    }
    result.push(f());
    return result;
}

test(true);  // ["local", "local"]
test(false); // ["local"]
```

然而，如果我们把函数 f 移到局部块里，那么，将产生一个完全不同的情形。

```
function f() { return "global"; }

function test(x) {
    var result = [];
    if (x) {
        function f() { return "local"; } // block-local

        result.push(f());
    }
    result.push(f());
    return result;
}

test(true);  // ?
test(false); // ?
```

由于内部的函数f出现在if语句块中，因此你可能认为第一次调用test产生数组["local", "global"]，第二次调用产生数组["global"]。但是要记住JavaScript没有块级作用域，所以内部函数f的作用域应该是整个test函数。第二个例子的合理猜测是["local", "local"]和["local"]。而事实上，一些JavaScript环境的确如此行事。但并不是所有的JavaScript环境都这样。其他一些环境在运行时根据包含函数f的块是否被执行来有条件地绑定函数f。(不仅使代码更难理解，而且还致使性能降低。这与with语句没什么不同。)

关于这一点ECMAScript标准说了什么呢？令人惊讶的是，几乎没有。直到ES5，JavaScript标准才承认局部块函数声明的存在。官方指定函数声明只能出现在其他函数或者程序的最外层。ES5甚至建议将在非标准环境的函数声明转变成警告或错误。一些流行的JavaScript实现在严格模式下将这类函数报告为错误(具有局部块函数声明的处于严格模式下的程序将报告一个语法错误)。这有助于检测出不可移植的代码，并为未来的标准版本在给局部块函数声明指定更明智和可移植的语义开辟了一条路。

在此期间，编写可移植的函数的最好方式是始终避免将函数声明置于局部块或子语句中。如果你想编写嵌套函数声明，应该将它置于其父函数的最外层，正如最开始的示例所示。另外，如果你需要有条件地选择函数，最好的办法是使用var声明和函数表达式来实现。

```
function f() { return "global"; }

function test(x) {
    var g = f, result = [];
    if (x) {
        g = function() { return "local"; }

        result.push(g());
    }
    result.push(g());
    return result;
}
```

这消除了内部变量(重命名为g)作用域的神秘性。它无条件地作为局部变量被绑定，而仅仅只有赋值语句是有条件的。结果很明确，该函数完全可移植。

提示

- 始终将函数声明置于程序或被包含的函数的最外层以避免不可移植的行为。
- 使用var声明和有条件的赋值语句替代有条件的函数声明。

第 16 条：避免使用 eval 创建局部变量

JavaScript 的 eval 函数是一个令人难以置信的强大、灵活的工具。强大的工具容易被滥用，所以了解它们是值得的。错误使用 eval 函数的最简单的方式之一是允许它干扰作用域。

调用 eval 函数会将其参数作为 JavaScript 程序进行解释。但是该程序运行于调用者的局部作用域中。嵌入到程序的全局变量会被创建为调用程序的局部变量。

```
function test(x) {
    eval("var y = x;"); // dynamic binding
    return y;
}
test("hello"); // "hello"
```

这个例子看起来很清晰，但此 var 声明语句与将其直接放置在 test 函数体中的行为略有不同。只有当 eval 函数被调用时此 var 声明语句才会被调用。只有当条件语句被执行时，放置在该条件语句中的 eval 函数才会将其变量加入到作用域中。

```
var y = "global";
function test(x) {
    if (x) {
        eval("var y = 'local';"); // dynamic binding
    }
    return y;
}
test(true);  // "local"
test(false); // "global"
```

基于作用域决定程序的动态行为通常是个坏主意。导致的结果是，即使想简单地理解变量是如何绑定的都需要了解程序执行的细节。当源代码将未在局部作用域内定义的变量传递给 eval 函数时，程序将变得特别棘手：

```
var y = "global";
function test(src) {
    eval(src); // may dynamically bind
    return y;

}
test("var y = 'local';"); // "local"
test("var z = 'local';"); // "global"
```

这段代码很脆弱，也不安全。它赋予了外部调用者能改变 test 函数内部作用域的能力。期望 eval 函数能修改自身包含的作用域对 ES5 严格模式的兼容性也是不可靠的 。ES5 严格模式将 eval 函数运行在一个嵌套的作用域中以防止这种污染。保证 eval 函数不影响外部作用域的一个简单方法是在一个明确的嵌套作用域中运行它。

```
var y = "global";
function test(src) {
    (function() { eval(src); })();
    return y;
}

test("var y = 'local';"); // "global"
test("var z = 'local';"); // "global"
```

提示

- 避免使用 eval 函数创建的变量污染调用者的作用域。
- 如果 eval 函数代码可能创建全局变量，将此调用封装到嵌套的函数中以防止作用域污染。

第 17 条：间接调用 eval 函数优于直接调用

eval 函数有一个秘密武器：它不仅仅是一个函数。

大多数函数只能访问定义它们所在的作用域，而不能访问除此之外的作用域。然而，eval 函数具有访问调用它那时的整个作用域的能力。这是非常强大的能力。当编译器编写者首次设法优化 JavaScript 时，他们发现 eval 函数很难高效地调用任何一个函数，因为一旦被调用的函数是 eval 函数，那么每个函数调用都需要确保在运行时整个作用域对 eval 函数是可访问的。

作为一种折中的解决方案，语言标准演化出了辨别两种不同的调用 eval 函数的方法。函数调用涉及 eval 标识符，被认为是一种“直接”调用 eval 函数的方式。

```
var x = "global";
function test() {
    var x = "local";
    return eval("x"); // direct eval
}
test(); // "local"
```

在这种情况下，编译器需要确保被执行的程序具有完全访问调用者局部作用域的权限。其他调用 eval 函数的方式被认为是“间接”的。这些方式在全局作用域内对 eval 函数的参数求值。例如，绑定 eval 函数到另一个变量名，通过该变量名调用函数会使代码失去对所有局部作用域的访问能力。

```
var x = "global";
function test() {
    var x = "local";
```

```
    var f = eval;
    return f("x"); // indirect eval
}
test(); // "global"
```

直接调用 eval 函数的确切的定义取决于 ECMAScript 标准相当特殊的规范语言。在实践中，唯一能够产生直接调用 eval 函数的语法是可能被（许多的）括号包裹的名称为 eval 的变量。编写间接调用 eval 函数的一种简洁方式是使用表达式序列运算符（,）和一个明显毫无意义的数字字面量。

```
(0,eval)(src);
```

这个奇形怪状的函数调用是如何工作的呢？数字字面量 0 被求值但其值被忽略掉了，括号表示的序列表达式产生的结果是 eval 函数。因此，(0,eval) 的行为几乎与简单的 eval 函数标识符完全一致，一个重要的区别在于整个调用表达式被视为是一种间接调用 eval 函数的方式。

直接调用 eval 函数的能力可能很容易被滥用。例如，对一个来自网络的源字符串进行求值，可能会暴露其内部细节给一些未受信者。第 16 条探讨了使用 eval 函数动态创建局部变量的危害。这些危害只可能与直接调用 eval 函数相关。此外，直接调用 eval 函数性能上的损耗也是相当高昂的。通常情况下，你要承担直接调用 eval 函数导致其包含的函数以及所有直到程序最外层的函数运行相当缓慢的风险。

出于某些原因偶尔也需要使用直接调用 eval 函数。但是，除非有一个检查局部作用域的特别能力的明确需求，否则应当使用更不容易滥用、更廉价的间接调用 eval 函数的方式。

提示

- 将 eval 函数同一个毫无意义的字面量包裹在序列表达式中以达到强制使用间接调用 eval 函数的目的。
- 尽可能间接调用 eval 函数，而不要直接调用 eval 函数。

第 3 章
使用函数

函数是 JavaScript 中的“老黄牛”，既给程序员提供了主要的抽象功能，又提供了实现机制。函数可以独自实现其他语言中的多个不同的特性，例如，过程、方法、构造函数，甚至类和模块。一旦你熟悉了函数的具体细节，你就掌握了 JavaScript 的重要组成部分。然而凡事都有两面性，学会在不同的环境中高效地使用函数是需要下一番工夫的。

第 18 条：理解函数调用、方法调用及构造函数调用之间的不同

如果你熟悉面向对象编程，你很可能习惯性地认为函数、方法和类的构造函数是三种不同的概念。然而在 JavaScript 中，它们只是单个构造对象的三种不同的使用模式。

最简单的使用模式是函数调用。

```
function hello(username) {
    return "hello, " + username;
}
hello("Keyser Söze"); // "hello, Keyser Söze"
```

该代码片段的表现与行为一致。其调用 hello 函数并将给定的实参绑定到 name 形参。

第二种使用模式是方法调用。JavaScript 中的方法不过是对象的属性恰好是函数而已。

```
var obj = {
    hello: function() {
        return "hello, " + this.username;
    },

     username: "Hans Gruber"
 };
 obj.hello(); // "hello, Hans Gruber"
```

请注意 hello 方法是如何通过 this 变量来访问 obj 对象的属性的。你可能倾向于 this 变量

被绑定到 obj 对象是由于 hello 方法被定义在 obj 对象中。但我们可以在另一个对象中复制一份相同函数的引用，并得到一个不同的答案。

```
var obj2 = {
    hello: obj.hello,
    username: "Boo Radley"
};
obj2.hello(); // "hello, Boo Radley"
```

事实是，在方法调用中是由调用表达式自身来确定 this 变量的绑定。绑定到 this 变量的对象被称为调用接收者（receiver）。表达式 obj.hello() 在 obj 对象中查找名为 hello 的属性，并将 obj 对象作为接收者，然后调用该属性。表达式 obj2.hello() 在 obj2 对象中查找名为 hello 的属性，恰巧正是 obj.hello 函数，但是接收者是 obj2 对象。通常，通过某个对象调用方法将查找该方法并将该对象作为该方法的接收者。

由于方法其实就是通过特定对象调用的函数，但不知为何一个普通的函数不能引用 this 变量。

```
function hello() {
    return "hello, " + this.username;
}
```

这对于预定义一个在多个项目中共享的函数很有用。

```
var obj1 = {
    hello: hello,
    username: "Gordon Gekko"
};
obj1.hello(); // "hello, Gordon Gekko"

var obj2 = {
    hello: hello,
    username: "Biff Tannen"
};
obj2.hello(); // "hello, Biff Tannen"
```

然而，一个使用了 this 变量的函数，比起作为方法被调用，将它作为函数被调用并不是特别有用。

```
hello(); // "hello, undefined"
```

一个非方法（nonmethod）的函数调用会将全局对象作为接收者，在这种情况下全局对象没有名为 name 的属性所以产生了 undefined，简直是帮倒忙。如果方法中需要使用 this 变量，则将方法作为函数调用则毫无用处，因为没理由希望全局对象匹配调用对象中的方法。事实上，将 this 变量绑定到全局对象是有问题的，所以 ES5 的严格模式将 this 变量的默认绑定值

改为 undefined。

```
function hello() {
    "use strict";
    return "hello, " + this.username;
}
hello(); // error: cannot read property "username" of undefined
```

这有助于更快地捕获偶然地将方法错误地作为纯函数使用的情况。因为试图访问 undefined 的属性会立即抛出一个错误。

函数的第三种用法是通过构造函数使用。就像方法和纯函数一样，构造函数也是由 function 运算符定义的。

```
function User(name, passwordHash) {
    this.name = name;
    this.passwordHash = passwordHash;
}
```

使用 new 操作符来调用 User 则视其为构造函数。

```
var u = new User("sfalken",
                 "0ef33ae791068ec64b502d6cb0191387");
u.name; // "sfalken"
```

与函数调用和方法调用不同的是，构造函数调用将一个全新的对象作为 this 变量的值，并隐式返回这个新对象作为调用结果。构造函数的主要职责是初始化该新对象。

- 方法调用将被查找方法属性的对象作为调用接收者。
- 函数调用将全局对象（处于严格模式下则为 undefined）作为其接收者。一般很少使用函数调用语法来调用方法。
- 构造函数需要通过 new 运算符调用，并产生一个新的对象作为其接收者。

第 19 条：熟练掌握高阶函数

高阶函数过去曾经是函数式编程的拥趸的一句行话，似乎也是一种先进的编程技术的一个深奥术语。然而，这与事实相差甚远。开发简洁优雅的函数通常可以使代码更加简单明了。在过去的几年中，脚本语言采用了这些技术，并在此过程中揭开了一些函数式编程的最佳惯用法的神秘面纱。

高阶函数无非是那些将函数作为参数或返回值的函数。将函数作为参数（通常称为回调函数，因为高阶函数“随后调用”它）是一种特别强大、富有表现力的惯用法，也在

JavaScript 程序中被大量使用。

考虑数组的标准 sort 方法，为了对所有的数组都能工作，sort 方法需要调用者决定如何比较数组中的任意两个元素。

```
function compareNumbers(x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
}
[3, 1, 4, 1, 5, 9].sort(compareNumbers); // [1, 1, 3, 4, 5, 9]
```

标准库需要调用者传递一个具有 compare 方法的对象，但是由于只有一个方法是必需的，所以直接传递一个函数更为简洁。事实上，上面的例子可以用一个匿名函数进一步简化。

```
[3, 1, 4, 1, 5, 9].sort(function(x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
}); // [1, 1, 3, 4, 5, 9]
```

学会使用高阶函数通常可以简化代码并消除繁琐的样板代码。许多关于数组的常见操作包含值得我们熟悉掌握的亲切的高阶函数抽象。假设有这样一个简单的转换字符串数组的操作我们可以使用循环这样实现：

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = [];
for (var i = 0, n = names.length; i < n; i++) {
    upper[i] = names[i].toUpperCase();
}
upper; // ["FRED", "WILMA", "PEBBLES"]
```

使用数组便利的 map 方法（在 ES5 引入了这个方法），我们可以完全消除循环，仅仅使用一个局部函数就可以实现对元素的逐个转换。

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = names.map(function(name) {
    return name.toUpperCase();
```

```
});
upper; // ["FRED", "WILMA", "PEBBLES"]
```

一旦掌握了高阶函数的使用，你就可以开始寻找机会编写自己的高阶函数。需要引入高阶函数抽象的信号是出现重复或相似的代码。例如，假设我们发现程序的部分代码段使用英文字母构造一个字符串。

```
var aIndex = "a".charCodeAt(0); // 97

var alphabet = "";
for (var i = 0; i < 26; i++) {
    alphabet += String.fromCharCode(aIndex + i)
}
alphabet; // "abcdefghijklmnopqrstuvwxyz"
```

同时，程序的另一部分代码段生成一个包含数字的字符串。

```
var digits = "";
for (var i = 0; i < 10; i++) {
    digits += i;
}
digits; // "0123456789"
```

此外，程序的其他地方还存在创建一个随机的字符串。

```
var random = "";

for (var i = 0; i < 8; i++) {
    random += String.fromCharCode(Math.floor(Math.random() * 26)
                                  + aIndex);
}
random; // "bdwvfrtp" (different result each time)
```

每个例子创建了一个不同的字符串，但它们都有着共同的逻辑。每个循环通过连接每个独立部分的计算结果来创建一个字符串。我们可以提取出共用的部分，将它们移到单个工具函数里。

```
function buildString(n, callback) {
    var result = "";
    for (var i = 0; i < n; i++) {
        result += callback(i);
    }
    return result;
}
```

请注意，buildString 函数的实现包含了每个循环的所有共用部分，并使用参数来替代变化的部分。循环迭代的次数由变量 n 替代，每个字符串片段的构造由 callback 函数替代。我

们现在可以使用 buildString 函数简化这三个例子。

```
var alphabet = buildString(26, function(i) {
    return String.fromCharCode(aIndex + i);
});
alphabet; // "abcdefghijklmnopqrstuvwxyz"

var digits = buildString(10, function(i) { return i; });
digits; // "0123456789"

var random = buildString(8, function() {
    return String.fromCharCode(Math.floor(Math.random() * 26)
                               + aIndex);
});
random; // "ltvisfjr" (different result each time)
```

创建高阶函数抽象有很多好处。实现中存在的一些棘手部分，比如正确地获取循环边界条件，它们可以被放置在高阶函数的实现中。这使得你可以一次性地修复所有逻辑上的错误，而不必去搜寻散布在程序中的该编码模式的所有实例。如果你发现需要优化操作的效率，你也可以仅仅只修改一处。最后，给高阶函数抽象一个清晰的名称（如 buildString），这样能使读者更清晰地了解该代码能做什么，而无须深入实现细节。

当发现自己在重复地写一些相同的模式时，学会借助于一个高阶函数可以使代码更简洁、更高效、更可读。留意一些常见的模式并将它们移到高阶的工具函数中是一个重要的开发习惯。

- 高阶函数是那些将函数作为参数或返回值的函数。
- 熟悉掌握现有库中的高阶函数。
- 学会发现可以被高阶函数所取代的常见的编码模式。

第 20 条：使用 call 方法自定义接收者来调用方法

通常情况下，函数或方法的接收者（即绑定到特殊关键字 this 的值）是由调用者的语法决定的。特别地，方法调用语法将方法被查找的对象绑定到 this 变量。然而，有时需要使用自定义接收者来调用函数，因为该函数可能并不是期望的接收者对象的属性。当然可以将方法作为一个新的属性添加到接收者对象中。

```
obj.temporary = f;    // what if obj.temporary already existed?
var result = obj.temporary(arg1, arg2, arg3);
delete obj.temporary; // what if obj.temporary already existed?
```

但是这种方式不仅让人感觉别扭，而且相当危险。修改 obj 对象往往是不可取的，甚至有时不可能修改。特别地，无论为 temporary 属性选择什么名称，都有可能与 obj 中已有的属性重名。此外，某些对象可能被冻结或密封以防止添加任何新属性。一般来说，随意给对象添加属性是一种不好的实践，尤其是对于不是自己创建的对象（参阅第 42 条）。

幸运的是，函数对象具有一个内置的方法 call 来自定义接收者。可以通过函数对象的 call 方法来调用其自身。

```
f.call(obj, arg1, arg2, arg3);
```

此行为与直接调用函数自身很类似。

```
f(arg1, arg2, arg3);
```

不同的是，第一个参数提供了一个显式的接收者对象。

当调用的方法已经被删除、修改或者覆盖时，call 方法就派上用场了。第 45 条展示了一个有用的实例。hasOwnProperty 方法可被任意的对象调用，甚至该对象可以是一个字典对象。在字典对象中，查找 hasOwnProperty 属性会得到该字典对象的属性值，而不是继承过来的方法。

```
dict.hasOwnProperty = 1;
dict.hasOwnProperty("foo"); // error: 1 is not a function
```

使用 hasOwnProperty 方法的 call 方法使调用字典对象中的方法成为可能，即使 hasOwnProperty 方法并没有存储在该对象中。

```
var hasOwnProperty = {}.hasOwnProperty;
dict.foo = 1;
delete dict.hasOwnProperty;
hasOwnProperty.call(dict, "foo");             // true
hasOwnProperty.call(dict, "hasOwnProperty"); // false
```

定义高阶函数时 call 方法也特别实用。高阶函数的一个惯用法是接收一个可选的参数作为调用该函数的接收者。例如，表示键值对列表的对象可能提供名为 forEach 的方法。

```
var table = {
    entries: [],
    addEntry: function(key, value) {
        this.entries.push({ key: key, value: value });
    },
    forEach: function(f, thisArg) {
        var entries = this.entries;

        for (var i = 0, n = entries.length; i < n; i++) {
            var entry = entries[i];
            f.call(thisArg, entry.key, entry.value, i);
```

```
        }
    }
};
```

上述例子允许 table 对象的使用者将一个方法作为 table.forEach 的回调函数 f，并为该方法提供一个合理的接收者。例如，可以方便地将一个 table 的内容复制到另一个中。

```
table1.forEach(table2.addEntry, table2);
```

这段代码从 table2 中提取 addEntry 方法（甚至可以从 Table.prototype 或者 table1 中提取），forEach 方法将 table2 作为接收者，并反复调用该 addEntry 方法。请注意，虽然 addEntry 方法只期望两个参数，但是 forEach 方法调用它时却传递给它三个参数：键、值、索引。这个多余的索引参数是无害的，因为 addEntry 方法简单地忽略了它。

- 使用 call 方法自定义接收者来调用函数。
- 使用 call 方法可以调用在给定的对象中不存在的方法。
- 使用 call 方法定义高阶函数允许使用者给回调函数指定接收者。

第 21 条：使用 apply 方法通过不同数量的参数调用函数

假设有人给我们提供了一个计算任意数量数字平均值的函数。

```
average(1, 2, 3);                       // 2
average(1);                             // 1
average(3, 1, 4, 1, 5, 9, 2, 6, 5); // 4
average(2, 7, 1, 8, 2, 8, 1, 8);    // 4.625
```

average 函数是一个称为可变参数或可变元的函数（函数的元数是指其期望的参数个数）的例子。它可以接收任意数量的参数。相比较而言，固定元数的 average 函数的版本可能会使用单个数字数组作为其参数。

```
averageOfArray([1, 2, 3]);                      // 2
averageOfArray([1]);                            // 1
averageOfArray([3, 1, 4, 1, 5, 9, 2, 6, 5]); // 4
averageOfArray([2, 7, 1, 8, 2, 8, 1, 8]);    // 4.625
```

无可争辩的是，可变参数的版本更加简洁、更加优雅。可变参数函数具有便捷的语法，至少让调用者预先明确地知道提供了多少个参数。正如上述的例子一样。但倘若我们有这样一个数字数组：

```
var scores = getAllScores();
```

我们如何使用 average 函数计算其平均值呢？

```
average(/* ? */);
```

幸运的是，函数对象配有一个内置的 apply 方法。它与 call 方法非常类似，但它是专为这一目的而设计的。apply 方法需要一个参数数组，然后将数组的每一个元素作为调用的单独参数调用该函数。除了参数数组，apply 方法指定第一个参数绑定到被调用函数的 this 变量。由于 average 函数没有引用 this 变量，因此，我们可以简单地传递 null。

```
var scores = getAllScores();
average.apply(null, scores);
```

例如，如果 scores 有三个元素，那么以上代码的行为与 average(scores[0], scores[1], scores[2]) 一致。

apply 方法也可用于可变参数方法。例如，buffer 对象包含一个可变参数的 append 方法，该方法添加元素到函数内部的 state 数组中（了解 append 方法的实现请参见第 22 条）。

```
var buffer = {
    state: [],
    append: function() {
        for (var i = 0, n = arguments.length; i < n; i++) {
            this.state.push(arguments[i]);
        }
    }
};
```

append 方法可以接受任意数量的参数进行调用。

```
buffer.append("Hello, ");
buffer.append(firstName, " ", lastName, "!");
buffer.append(newline);
```

借助于 apply 方法的 this 参数，我们也可以指定一个可计算的数组调用 append 方法：

```
buffer.append.apply(buffer, getInputStrings());
```

请注意 buffer 参数的重要性。如果我们传递一个不同的对象，那么，append 方法将尝试修改该错误对象的 state 属性。

提示

- 使用 apply 方法指定一个可计算的参数数组来调用可变参数的函数。
- 使用 apply 方法的第一个参数给可变参数的方法提供一个接收者。

第 22 条：使用 arguments 创建可变参数的函数

第 21 条讲述了一个可变参数的函数 average。该函数可处理任意数量的参数并返回这些

参数的平均值。如何自己实现可变参数的函数？固定元数版本的 averageOfArray 函数是很容易实现的。

```
function averageOfArray(a) {
    for (var i = 0, sum = 0, n = a.length; i < n; i++) {
        sum += a[i];
    }
    return sum / n;
}
averageOfArray([2, 7, 1, 8, 2, 8, 1, 8]); // 4.625
```

averageOfArrey 函数定义了一个形参（formal parameter），即参数列表中的变量 a。当调用函数 averageOfArray 时，只需提供单个参数（有时称此参数为实参（actual argument），用于与形参区分开来），即数字数组。

此可变参数的版本与上个示例几乎完全相同，但其没有定义任何显式的形参。相反，它利用了 JavaScript 的一个事实，即 JavaScript 给每个函数都隐式地提供了一个名为 arguments 的局部变量。arguments 对象给实参提供了一个类似数组的接口。它为每个实参提供了一个索引属性，还包含一个 length 属性用来指示参数的个数。从而可以通过遍历 arguments 对象的每个元素来实现可变元数的 average 函数。

```
function average() {
    for (var i = 0, sum = 0, n = arguments.length;
         i < n;
         i++) {
        sum += arguments[i];
    }
    return sum / n;
}
```

可变参数函数提供了灵活的接口。不同的调用者可使用不同数量的参数来调用它们。但它们自身也失去了一点便利。如果使用者想使用计算的数组参数来调用可变参数的函数，只能使用在第 21 条中描述的 apply 方法。好的经验法则是，如果提供了一个便利的可变参数的函数，也最好提供一个需要显式指定数组的固定元数的版本。这通常很容易实现，因为我们可以编写一个轻量级的封装，并委托给固定元数的版本来实现可变参数的函数。

```
function average() {
    return averageOfArray(arguments);
}
```

这样一来，函数的使用者就无需借助 apply 方法，因为 apply 方法会降低可读性而且经常导致性能损失。

- 使用隐式的 arguments 对象实现可变参数的函数。
- 考虑对可变参数的函数提供一个额外的固定元数的版本，从而使使用者无需借助 apply 方法。

第 23 条：永远不要修改 arguments 对象

arguments 对象可能看起来像一个数组，但遗憾的是，它并不总是表现得像数组。熟悉 Perl 和 UNIX Shell 脚本的程序员习惯使用“移除（shifting）”参数数组开头元素的技术。而事实上，JavaScript 的数组确实有 shift 方法。shift 方法删除数组的第一个元素并逐个移动所有后续的元素。但是，arguments 对象自身并不是标准 Array 类型的实例。因此，我们不能直接调用 argument.shift() 方法。

多亏了 call 方法，才有希望能够从数组中提取出 shift 方法，并在 arguments 对象上调用它。这似乎是实现如 callMethod 这样的函数的一种合理的方式。callMethod 函数需要一个对象和一个方法名，并尝试使用剩余的参数调用该对象的指定方法。

```
function callMethod(obj, method) {
    var shift = [].shift;
    shift.call(arguments);
    shift.call(arguments);
    return obj[method].apply(obj, arguments);
}
```

但是，此函数的行为远远超出了预期。

```
var obj = {
    add: function(x, y) { return x + y; }
};
callMethod(obj, "add", 17, 25);
// error: cannot read property "apply" of undefined
```

该函数出错的原因是 arguments 对象并不是函数参数的副本。特别是，所有命名参数都是 arguments 对象中对应索引的别名。因此，即使通过 shift 方法移除 arguments 对象中的元素之后，obj 仍然是 arguments[0] 的别名，method 仍然是 arguments[1] 的别名。这意味着，尽管我们似乎是在提取 obj["add"]，但实际上是在提取 17[25]！此时一切开始失控。由于 JavaScript 的自动强制转换规则，引擎将 17 转换为 Number 对象并提取其“25”属性（该属性并不存在），结果产生 undefined，然后试图提取 undefined 的“apply”属性并将其作为方法来调用，结果当然失败了。

这个例子告诉我们，arguments 对象与函数的命名参数之间的关系极其脆弱。修改

arguments 对象需要承担使函数的命名参数失去意义的风险。在 ES5 的严格模式下，情况甚至更为复杂。在严格模式下，函数参数不支持对其 arguments 对象取别名。我们可以通过编写一个更新 arguments 对象某个元素的函数来说明这个差异。

```
function strict(x) {
    "use strict";
    arguments[0] = "modified";

    return x === arguments[0];
}
function nonstrict(x) {
    arguments[0] = "modified";
    return x === arguments[0];
}
strict("unmodified");    // false
nonstrict("unmodified"); // true
```

因此，永远不要修改 arguments 对象是更为安全的。通过一开始复制参数中的元素到一个真正的数组的方式，很容易避免修改 arguments 对象。下面是实现复制的简单惯用法。

```
var args = [].slice.call(arguments);
```

当不使用额外的参数调用数组的 slice 方法时，它会复制整个数组，其结果是一个真正的标准 Array 类型实例。该实例保证不会有任何别名，并且可以直接使用标准 Array 类型中的所有方法。

我们可以通过复制 arguments 对象修复 callMethod 函数的实现。由于我们只需要 obj 和 method 之后的元素，因此，我们可以指定 slice 方法的开始索引位置为 2。

```
function callMethod(obj, method) {
    var args = [].slice.call(arguments, 2);
    return obj[method].apply(obj, args);
}
```

最终，callMethod 函数如预期般运作。

```
var obj = {
    add: function(x, y) { return x + y; }
};
callMethod(obj, "add", 17, 25); // 42
```

- 永远不要修改 arguments 对象。
- 使用 [].slice.call(arguments) 将 arguments 对象复制到一个真正的数组中再进行修改。

第 24 条：使用变量保存 arguments 的引用

迭代器（iterator）是一个可以顺序存取数据集合的对象。其一个典型的 API 是 next 方法，该方法获得序列中的下一个值。假设我们希望编写一个便利的函数，它可以接收任意数量的参数，并为这些值建立一个迭代器。

```
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // 1
it.next(); // 4
it.next(); // 1
```

由于 values 函数必须能够接受任意数量的参数，所以我们可以构造迭代器对象来遍历 arguments 对象的元素。

```
function values() {
    var i = 0, n = arguments.length;
    return {
        hasNext: function() {
            return i < n;
        },
        next: function() {
            if (i >= n) {
                throw new Error("end of iteration");
            }
            return arguments[i++]; // wrong arguments
        }
    };
}
```

但是这段代码有问题，当试图使用迭代器对象时这个问题立马就会暴露出来。

```
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // undefined
it.next(); // undefined
it.next(); // undefined
```

一个新的 arguments 变量被隐式地绑定到每个函数体内，所以导致了这个问题。我们感兴趣的 arguments 对象是与 values 函数相关的那个，但是迭代器的 next 方法含有自己的 arguments 变量。所以当返回 arguments[i++] 时，我们访问的是 it.next 的参数，而不是 values 函数中的参数。

解决方案很简单：只需简单地在我们感兴趣的 arguments 对象作用域内绑定一个新的局部变量，并确保嵌套函数只能引用这个显式命名的变量。

```
function values() {
    var i = 0, n = arguments.length, a = arguments;
```

```
    return {
        hasNext: function() {
            return i < n;
        },
        next: function() {
            if (i >= n) {
                throw new Error("end of iteration");
            }
            return a[i++];
        }
    };
}
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // 1
it.next(); // 4
it.next(); // 1
```

提示

- 当引用 arguments 时当心函数嵌套层级。
- 绑定一个明确作用域的引用到 arguments 变量，从而可以在嵌套的函数中引用它。

第 25 条：使用 bind 方法提取具有确定接收者的方法

由于方法与值为函数的属性没有区别，因此很容易提取对象的方法并将提取出的函数作为回调函数直接传递给高阶函数。但这也很容易忘记将提取出的函数的接收者绑定到该函数被提取出的对象上。假设一个字符串缓冲对象使用数组来存储字符串，该数组稍后可能被连接起来。

```
var buffer = {
    entries: [],
    add: function(s) {
        this.entries.push(s);
    },
    concat: function() {
        return this.entries.join("")
    }
};
```

似乎通过提取 buffer 对象的 add 方法，并使用 ES5 提供的 forEach 方法在每一个源数组元素上重复地调用 add 方法，就可以复制字符串数组到 buffer 对象中。

```
var source = ["867", "-", "5309"];
source.forEach(buffer.add); // error: entries is undefined
```

但 buffer.add 的接收者并不是 buffer 对象。函数的接收者取决于它是如何被调用的，然而，我们并没有在这里调用它。相反，我们把它传递给了 forEach 方法，而我们并不知道 forEach 方法在哪里调用了它。事实上，forEach 方法的实现使用全局对象作为默认的接收者。由于全局对象没有 entries 属性，因此这段代码抛出了一个错误。幸运的是，forEach 方法允许调用者提供一个可选的参数作为回调函数的接收者，所以，我们可以很轻松地修复该例子。

```
var source = ["867", "-", "5309"];
source.forEach(buffer.add, buffer);
buffer.join(); // "867-5309"
```

并非所有的高阶函数都细心周到地为其使用者提供其回调函数的接收者。如果 forEach 方法不接受额外的接收者参数怎么办？一个好的解决办法是创建使用适当的方法调用语法来调用 buffer.add 的一个局部函数。

```
var source = ["867", "-", "5309"];
source.forEach(function(s) {
    buffer.add(s);
});
buffer.join(); // "867-5309"
```

该版本创建一个显式地以 buffer 对象方法的方式调用 add 的封装函数。请注意该封装函数自身根本没有引用 this 变量。不管封装函数如何被调用（作为函数、作为其他一些对象的方法或者通过 call 方法），它总能确保将其参数推送到目标数组中。

创建一个函数用来实现绑定其接收者到一个指定的对象是非常常见的，因此 ES5 标准库中直接支持这种模式。函数对象的 bind 方法需要一个接收者对象，并产生一个以该接收者对象的方法调用的方式调用原来的函数的封装函数。使用 bind 方法，我们可以简化该例子。

```
var source = ["867", "-", "5309"];
source.forEach(buffer.add.bind(buffer));
buffer.join(); // "867-5309"
```

请记住，buffer.add.bind(buffer) 创建了一个新函数而不是修改了 buffer.add 函数。新函数的行为就像原来函数的行为，但它的接收者绑定到了 buffer 对象，而原有函数的接收者保持不变。换句话说，

```
buffer.add === buffer.add.bind(buffer); // false
```

这很微妙但至关重要。这意味着调用 bind 方法是安全的，即使是一个可能在程序的其他部分被共享的函数。这对于原型对象上的共享方法尤其重要。当在任何的原型后代中调用共享方法时，该方法仍能正常工作。（更多关于对象和原型的知识请参阅第 4 章。）

- 要注意，提取一个方法不会将方法的接收者绑定到该方法的对象上。
- 当给高阶函数传递对象方法时，使用匿名函数在适当的接收者上调用该方法。
- 使用 bind 方法创建绑定到适当接收者的函数。

第 26 条：使用 bind 方法实现函数柯里化

函数对象的 bind 方法除了具有将方法绑定到接收者的用途外，它还有更多的用途。倘若有一个装配 URL 字符串的简单函数。

```
function simpleURL(protocol, domain, path) {
    return protocol + "://" + domain + "/" + path;
}
```

通常情况下，程序可能需要将特定站点的路径字符串构建为绝对路径 URL。一种自然的方式是对数组使用 ES5 提供的 map 方法来实现。

```
var urls = paths.map(function(path) {
    return simpleURL("http", siteDomain, path);
});
```

注意，上述例子中的匿名函数对 map 方法的每次迭代使用相同的协议字符串和网站域名字符串。传给 simpleURL 函数的前两个参数对于每次迭代都是固定的，仅第三个参数是变化的。我们可以通过调用 simpleURL 函数的 bind 方法来自动构造该匿名函数。

```
var urls = paths.map(simpleURL.bind(null, "http", siteDomain));
```

对 simpleURL.bind 的调用产生了一个委托到 simpleURL 的新函数。与往常一样，bind 方法的第一个参数提供了接收者的值。（由于 simpleURL 不需要引用 this 变量，所以可以使用任何值。使用 null 或 undefined 是习惯用法。）simpleURL.bind 的其余参数和提供给新函数的所有参数共同组成了传递给 simpleURL 的参数。换言之，使用单个参数 path 调用 simpleURL.bind，则该执行结果是一个委托到 simpleURL("http", siteDomain, path) 的函数。

将函数与其参数的一个子集绑定的技术称为函数柯里化（currying），以逻辑学家 Haskell Curry 的名字命名。他在数学中推广了这项技术。比起显式的封装函数，函数柯里化是一种简洁的、使用更少引用来实现函数委托的方式。

- 使用 bind 方法实现函数柯里化，即创建一个固定需求参数子集的委托函数。
- 传入 null 或 undefined 作为接收者的参数来实现函数柯里化，从而忽略其接收者。

第 27 条：使用闭包而不是字符串来封装代码

函数是一种将代码作为数据结构存储的便利方式，这些代码可以随后被执行。这使得富有表现力的高阶函数抽象如 map 和 forEach 成为可能。它也是 JavaScript 异步 I/O 方法的核心（请参阅第 7 章）。与此同时，也可以将代码表示为字符串的形式传递给 eval 函数以达到同样的功能。于是，程序员面临做这样一个决定：应该将代码表示为函数还是字符串？

毫无疑问，应该将代码表示为函数。字符串表示代码不够灵活的一个重要原因是：它们不是闭包。

假设有一个简单的多次重复用户提供的动作的函数。

```
function repeat(n, action) {
    for (var i = 0; i < n; i++) {
        eval(action);
    }
}
```

该函数在全局作用域会工作得很好，因为 eval 函数会将出现在字符串中的所有变量引用作为全局变量来解释。例如，一个测试函数基准执行速度的脚本可能恰好使用全局的 start 和 end 变量来存储时间。

```
var start = [], end = [], timings = [];
repeat(1000,
       "start.push(Date.now()); f(); end.push(Date.now())");
for (var i = 0, n = start.length; i < n; i++) {
    timings[i] = end[i] - start[i];
}
```

但是，该脚本很脆弱。如果我们简单地将代码移到一个函数中，那么 start 和 end 变量将不再是全局变量。

```
function benchmark() {
    var start = [], end = [], timings = [];
    repeat(1000,
        "start.push(Date.now()); f(); end.push(Date.now())");
    for (var i = 0, n = start.length; i < n; i++) {
        timings[i] = end[i] - start[i];
    }
    return timings;
}
```

该函数会导致 repeat 函数引用全局的 start 和 end 变量。最好的情况是，其中一个全局变量未定义，调用 benchmark 函数抛出 ReferenceError 异常。如果我们真的不走运，代码就会调用恰好绑定到 start 和 end 全局对象的 push 方法，那么程序的行为将不可预测。

更健壮的 API 应该接受函数而不是字符串。

```
function repeat(n, action) {
    for (var i = 0; i < n; i++) {
        action();
    }
}
```

这样一来，benchmark 脚本就能安全地引用闭包中的局部变量，该闭包以 repeat 函数的回调函数传递进来。

```
function benchmark() {
    var start = [], end = [], timings = [];
    repeat(1000, function() {
        start.push(Date.now());
        f();
         end.push(Date.now());
    });
    for (var i = 0, n = start.length; i < n; i++) {
        timings[i] = end[i] - start[i];
    }
    return timings;
}
```

eval 函数的另一个问题是，通常一些高性能的引擎很难优化字符串中的代码，因为编译器不能尽可能早地获得源代码来及时优化代码。然而函数表达式在其代码出现的同时就能被编译，这使得它更适合标准化编译。


提示

- 当将字符串传递给 eval 函数以执行它们的 API 时，绝不要在字符串中包含局部变量引用。
- 接受函数调用的 API 优于使用 eval 函数执行字符串的 API。

第 28 条：不要信赖函数对象的 toString 方法

JavaScript 函数有一个非凡的特性，即将其源代码重现为字符串的能力。

```
(function(x) {
    return x + 1;
}).toString(); // "function (x) {\n    return x + 1;\n}"
```

反射获取函数源代码的功能很强大，聪明的黑客偶然会通过巧妙的方法用到它。但是使用函数对象的 toString 方法有严重的局限性。

首先，ECMAScript 标准对函数对象的 toString 方法的返回结果（即该字符串）并没有任

何要求。这意味着不同的 JavaScript 引擎将产生不同的字符串，甚至产生的字符串与该函数并不相关。

事实上，如果函数是使用纯 JavaScript 实现的，那么 JavaScript 引擎会试图提供该函数的源代码的真实表示。下面是一个失败的例子。该例子失败的原因是使用了由宿主环境的内置库提供的函数。

```
(function(x) {
    return x + 1;
}).bind(16).toString(); // "function (x) {\n    [native code]\n}"
```

首先，由于在许多宿主环境中，bind 函数是由其他编程语言实现的（通常是 C++）。宿主环境提供的是一个编译后的函数，在此环境下该函数没有 JavaScript 的源代码供显示。

其次，由于标准允许浏览器引擎改变 toString 方法的输出，这就很容易使编写的程序在一个 JavaScript 系统中正确运行，在其他 JavaScript 系统中却无法正确运行。程序对函数的源代码字符串的具体细节很敏感，即使 JavaScript 的实现有一点细微的变化（如空格格式化）都可能破坏程序。

最后，由 toString 方法生成的源代码并不展示闭包中保存的与内部变量引用相关的值。

```
(function(x) {
    return function(y) {
        return x + y;
    }
})(42).toString(); // "function (y) {\n    return x + y;\n}"
```

注意，尽管函数实际上是一个绑定 x 为 42 的闭包，但结果字符串仍然包含一个引用了 x 的变量。

从某种意义上说，toString 方法的这些局限使其用来提取函数源代码并不是特别有用和值得信赖。通常应该避免使用它。对提取函数源代码相当复杂的使用应当采用精心制作的 JavaScript 解释器和处理库。但毫无疑问，将 JavaScript 函数看作一个不该违背的抽象是最稳妥的。

提示

- 当调用函数的 toString 方法时，并没有要求 JavaScript 引擎能够精确地获取到函数的源代码。
- 由于在不同的引擎下调用 toString 方法的结果可能不同，所以绝不要信赖函数源代码的详细细节。
- toString 方法的执行结果并不会暴露存储在闭包中的局部变量值。
- 通常情况下，应该避免使用函数对象的 toString 方法。

第 29 条：避免使用非标准的栈检查属性

曾经许多 JavaScript 环境都提供检查调用栈的功能。调用栈是指当前正在执行的活动函数链（更多关于调用栈的信息请参阅第 64 条）。在某些旧的宿主环境中，每个 arguments 对象都含有两个额外的属性：arguments.callee 和 arguments.caller。前者指向使用该 arguments 对象被调用的函数；后者指向调用该 arguments 对象的函数。许多环境仍然支持 arguments.callee，但它除了允许匿名函数递归地引用其自身之外，就没有更多的用途了。

```
var factorial = (function(n) {
    return (n <= 1) ? 1 : (n * arguments.callee(n - 1));
});
```

但这并不是特别有用，因为更直接的方式是使用函数名来引用函数自身。

```
function factorial(n) {
    return (n <= 1) ? 1 : (n * factorial(n - 1));
}
```

arguments.caller 属性更为强大。它指向的是使用该 arguments 对象调用函数的函数。出于安全的考虑，大多数环境已经移除了此特性，因此它是不可靠的。许多 JavaScript 环境也提供了一个相似的函数对象属性——非标准但普遍适用的 caller 属性。它指向函数最近的调用者。

```
function revealCaller() {
    return revealCaller.caller;
}

function start() {
    return revealCaller();
}

start() === start; // true
```

使用该属性来获取栈跟踪（stack trace）是很有诱惑力的。栈跟踪是一个提供当前调用栈快照的数据结构。构建一个栈跟踪看上去似乎很简单。

```
function getCallStack() {
    var stack = [];
    for (var f = getCallStack.caller; f; f = f.caller) {
        stack.push(f);
    }
    return stack;
}
```

对于简单的调用栈，getCallStack 函数可以很好地工作。

```
function f1() {
    return getCallStack();
}

function f2() {
    return f1();
}

var trace = f2();
trace; // [f1, f2]
```

但 getCallStack 函数非常脆弱。如果某个函数在调用栈中出现了不止一次，那么栈检查逻辑将会陷入循环。

```
function f(n) {
    return n === 0 ? getCallStack() : f(n - 1);
}

var trace = f(1); // infinite loop
```

问题出在哪里？由于函数 f 递归地调用其自身，因此其 caller 属性会自动更新，指回到函数 f。所以，函数 getCallStack 会陷入无限地查找函数 f 的循环之中。即使我们试图检测该循环，但在函数 f 调用其自身之前也没有关于哪个函数调用了它的信息。因为其他调用栈的信息已经丢失了。

这些栈检查属性都是非标准的，在移植性或适用性上很受限制。而且，在 ES5 的严格函数中，它们是被明令禁止使用的。试图获取严格函数或 arguments 对象的 caller 或 callee 属性都将抛出一个错误。

```
function f() {
    "use strict";
    return f.caller;
}

f(); // error: caller may not be accessed on strict functions
```

最好的策略是完全避免栈检查。如果检查栈的理由完全是为了调试，那么更为可靠的方式是使用交互式的调试器。

提示

- 避免使用非标准的 arguments.caller 和 arguments.callee 属性，因为它们不具备良好的移植性。
- 避免使用非标准的函数对象 caller 属性，因为在包含全部栈信息方面，它是不可靠的。

第 4 章
对象和原型

对象是 JavaScript 中的基本数据结构。直观地看，对象表示字符串与值映射的一个表格。但当你深入挖掘时，你会发现对象中有相当多的体系。

与许多面向对象的语言一样，JavaScript 支持继承，即通过动态代理机制重用代码或数据。但不像许多传统的语言，JavaScript 的继承机制基于原型，而不是类。对于许多程序员来说，JavaScript 是第一个不需要类而实现面向对象的语言。

在许多语言中，每个对象是相关类的实例，该类提供在其所有实例间共享代码。相反，JavaScript 并没有类的内置概念，对象是从其他对象中继承而来。每个对象与其他一些对象是相关的，这些对象称为它的原型。尽管依然使用了很多传统的面向对象语言的概念，但使用原型与使用类有很大差异。

第 30 条：理解 prototype、getPrototypeOf 和 __proto__ 之间的不同

原型包括三个独立但相关的访问器。这三个访问器的命名都是对单词 prototype 做了一些变化。这个不幸的重叠自然会导致相当多的混乱。让我们开门见山地讲明白吧。

- C.prototype 用于建立由 new C() 创建的对象的原型。
- Object.getPrototypeOf(obj) 是 ES5 中用来获取 obj 对象的原型对象的标准方法。
- obj.__proto__ 是获取 obj 对象的原型对象的非标准方法。

要理解这些访问器，我们设想一个典型的 JavaScript 数据类型的定义。User 构造函数需要通过 new 操作符来调用。它需要两个参数，即姓名和密码的哈希值，并将它们存储在创建的对象中。

```
function User(name, passwordHash) {
    this.name = name;
```

```
    this.passwordHash = passwordHash;
}

User.prototype.toString = function() {
    return "[User " + this.name + "]";
};

User.prototype.checkPassword = function(password) {
    return hash(password) === this.passwordHash;
};

var u = new User("sfalken",
                 "0ef33ae791068ec64b502d6cb0191387");
```

User 函数带有一个默认的 prototype 属性，其包含一个开始几乎为空的对象。在这个例子中，我们添加了两个方法到 User.prototype 对象 :toString 和 checkPassword。当我们使用 new 操作符创建 User 的实例时，产生的对象 u 得到了自动分配的原型对象，该原型对象被存储在 User.prototype 中。图 4.1 显示了这些对象的图表。

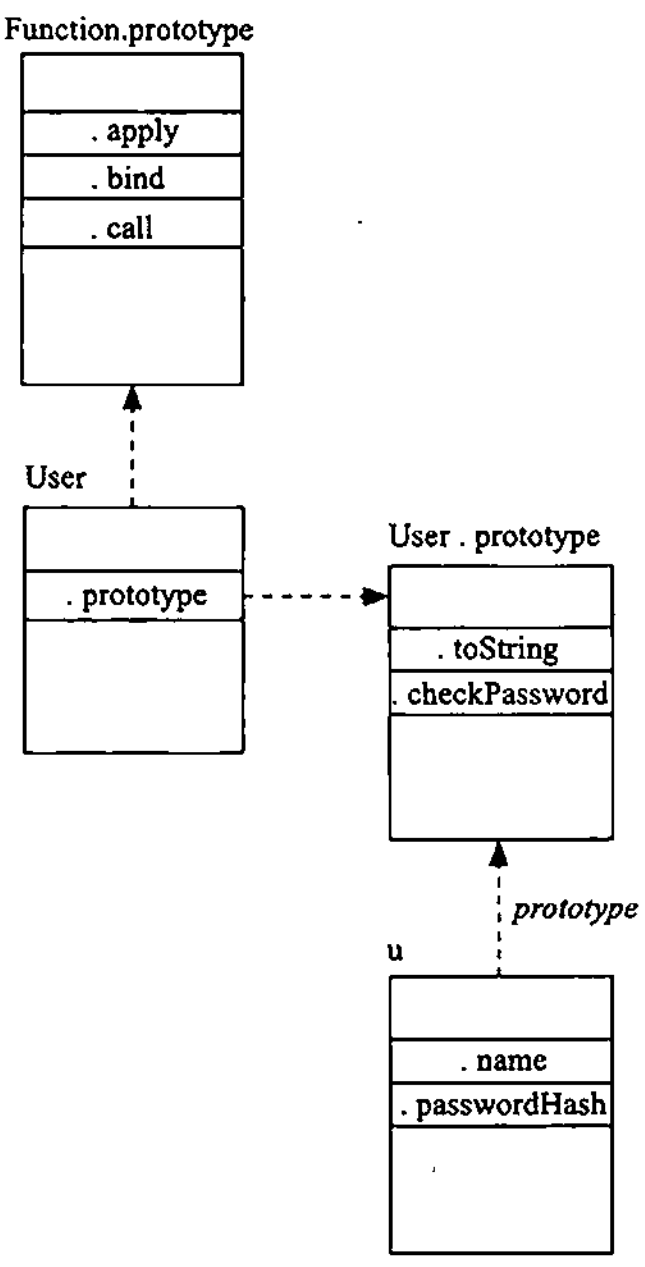

图 4.1 User 构造函数及其实例的原型关系

请注意箭头链接实例对象 u 到其原型对象 User.prototype。此链接描述了它们的继承关系。属性查找会从对象的自身属性开始搜索。例如，u.name 及 u.passwordHash 返回的是对象 u 的直接属性的当前值。假如没有在对象 u 中找到相应的属性，才会接着查找 u 的原型对象。例如，访问 u.checkPassword，返回的是存储在 User.protoype 中的方法。

让我们来看下一个有关原型的方法。构造函数的 prototype 属性用来设置新实例的原型关系。ES5 中的函数 Object.getPrototypeOf() 可以用于检索现有对象的原型。例如，当我们创建了上述例子中的对象 u 后，可以这样测试。

```
Object.getPrototypeOf(u) === User.prototype; // true
```

一些环境提供了一个非标准的方法检索对象的原型，即特殊的 __proto__ 属性。这可作为在不支持 ES5 的 Object.getPrototypeOf 方法的环境中的一个权宜之计。在这些环境中，我

们可以这样检测：

```
u.__proto__ === User.prototype; // true
```

关于原型关系的最后说明：JavaScript 程序员往往将 User 描述为一个类，尽管它跟一个函数差不多。JavaScript 中的类本质上是一个构造函数（User）与一个用于在该类（User.prototype）实例间共享方法的原型对象的结合。

图 4.2 提供了一个很好的方法来理解 User 类的概念。User 函数给该类提供了一个公共的构造函数，而 User.prototype 是实例之间共享方法的一个内部实现。User 和 u 的普通用法都不需要直接访问原型对象。

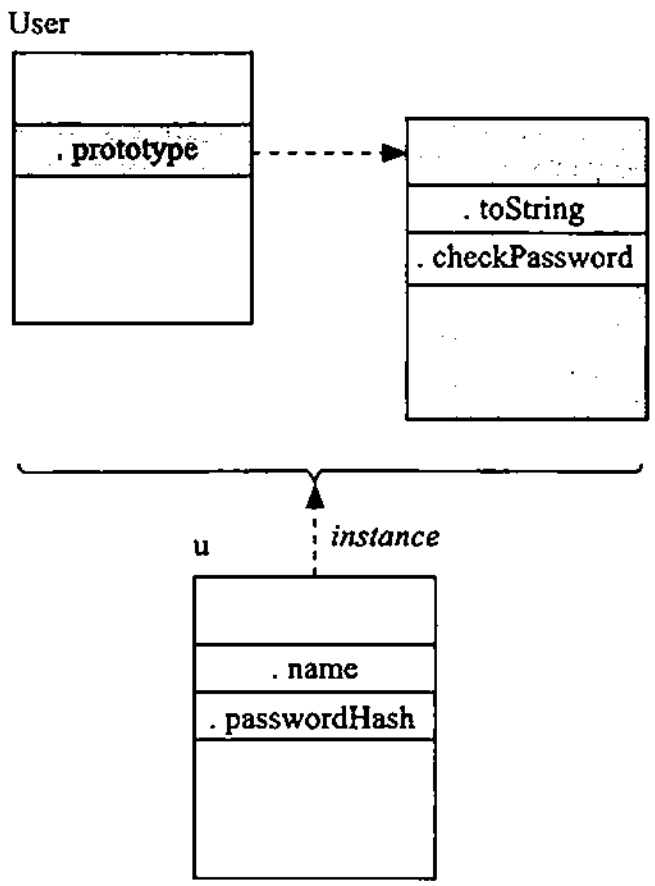

图 4.2　User 类的概念图

提示

- C.prototype 属性是 new C() 创建的对象的原型。
- Object.getPrototypeOf(obj) 是 ES5 中检索对象原型的标准函数。
- obj.__proto__ 是检索对象原型的非标准方法。
- 类是由一个构造函数和一个关联的原型组成的一种设计模式。

第 31 条：使用 Object.getPrototypeOf 函数而不要使用 __proto__ 属性

ES5 引入 Object.getPrototypeOf 函数作为获取对象原型的标准 API，但在这之前大量的 JavaScript 引擎早就使用一个特殊的 __proto__ 属性来达到相同的目的。然而，并不是所有的 JavaScript 环境都支持通过 __proto__ 属性来获取对象的原型，因此，该属性并不是完全兼容的。例如，对于拥有 null 原型的对象，不同的环境处理得不一样。在一些环境中，__proto__ 属性继承自 Object.prototype，因此，拥有 null 原型的对象没有这个特殊的 __proto__ 属性。

```
var empty = Object.create(null); // object with no prototype
"__proto__" in empty; // false (in some environments)
```

而其他的环境总是特别地处理 __proto__ 属性而不管对象的状态。

```
var empty = Object.create(null); // object with no prototype
"__proto__" in empty; // true (in some environments)
```

无论在什么情况下，Object.getPrototypeOf 函数都是有效的，而且，它也是提取对象原

型更加标准、可移植的方法。由于 __proto__ 属性会污染所有的对象，因此它会导致大量的 Bug（请参阅第 45 条）。当前支持这一扩展的 JavaScript 引擎可能选择在未来允许程序禁用它以避免这些 Bug。使用 Object.getPrototypeOf 函数确保即使禁用了 __proto__ 属性，代码也能继续工作。

对于那些没有提供该 ES5 API 的 JavaScript 环境，也可以很容易地利用 __proto__ 属性来实现 Object.getPrototypeOf 函数。

```
if (typeof Object.getPrototypeOf === "undefined") {
    Object.getPrototypeOf = function(obj) {
        var t = typeof obj;
        if (!obj || (t !== "object" && t !== "function")) {
            throw new TypeError("not an object");
        }
        return obj.__proto__;
    };
}
```

该实现是安全的，包括在 ES5 的环境中，因为它避免了即使 Object.getPrototypeOf 函数已经存在而仍然设置该函数的情况。


提示

- 使用符合标准的 Object.getPrototypeOf 函数而不要使用非标准的 __proto__ 属性。
- 在支持 __proto__ 属性的非 ES5 环境中实现 Object.getPrototypeOf 函数。

第 32 条：始终不要修改 __proto__ 属性

__proto__ 属性很特殊，它提供了 Object.getPrototypeOf 方法所不具备的额外的能力，即修改对象原型链接的能力。虽然这种能力看起来是无害的（毕竟，它只是另一个属性，不是吗？），但实际上会造成严重的影响，应当避免使用。避免修改 __proto__ 属性的最明显的原因是可移植性问题。因为并不是所有的平台都支持改变对象原型的特性，所以根本无法编写可移植的代码。

避免修改 __proto__ 属性的另一个原因是性能问题。所有现代的 JavaScript 引擎都深度优化了获取和设置对象属性的行为，因为这些都是一些最常见的 JavaScript 程序的操作。这些优化都是基于引擎对对象结构的认识上。当更改了对象的内部结构（如添加或删除该对象或其原型链中的对象的属性），将会使一些优化失效。修改 __proto__ 属性实际上改变了继承结构本身，这可能是最具破坏性的修改。比起普通的属性修改，修改 __proto__ 属性会导致更多的优化失效。

但避免修改 __proto__ 属性最大的原因是为了保持行为的可预测性。对象的原型链通过

其一套确定的属性及属性值来定义它的行为。修改对象的原型链就像对其进行“大脑移植”（brain transplant），这会交换对象的整个继承层次结构。在某些特殊情况下这样的操作可能是有用的，但是保持继承层次结构的相对稳定是一个基本的准则。

可以使用 ES5 中的 Object.create 函数来创建一个具有自定义原型链的新对象。对于不支持 ES5 的运行环境，第 33 条中提供了一种不依赖于 __proto__ 属性的可移植的 Object.creat 函数实现。

- 始终不要修改对象的 __proto__ 属性。
- 使用 Object.create 函数给新对象设置自定义的原型。

第 33 条：使构造函数与 new 操作符无关

当使用类似第 30 条中的 User 函数创建一个构造函数时，程序依赖于调用者是否记得使用 new 操作符来调用该构造函数。请注意，该函数假设接收者是一个全新的对象。

```
function User(name, passwordHash) {
    this.name = name;
    this.passwordHash = passwordHash;
}
```

如果调用者忘记使用 new 关键字，那么函数的接收者将是全局对象。

```
var u = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
u;                  // undefined
this.name;          // "baravelli"
this.passwordHash; // "d8b74df393528d51cd19980ae0aa028e"
```

该函数不但会返回无意义的 undefined，而且会灾难性地创建（如果这些全局变量已经存在则会修改）全局变量 name 和 passwordHash。

如果将 User 函数定义为 ES5 的严格代码，那么它的接收者默认为 undefined。

```
function User(name, passwordHash) {
    "use strict";
    this.name = name;

    this.passwordHash = passwordHash;
}

var u = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
// error: this is undefined
```

在这种情况下，这种错误的调用会导致一个即时错误：User 的第一行试图给 this.name

赋值，这会抛出 TypeError 异常。因此，使用严格的构造函数至少会帮助调用者尽早地发现该 Bug 并修复它。

而无论在哪种情况下，User 函数都是脆弱的。当和 new 操作符一同使用时，它能按预期工作，然而，当将它作为一个普通的函数调用时便会失败。一个更为健壮的方式是提供一个不管怎样调用都工作如构造函数的函数。实现该函数的一个简单方法是检查函数的接收者是否是一个正确的 User 实例。

```
function User(name, passwordHash) {
    if (!(this instanceof User)) {
        return new User(name, passwordHash);
    }
    this.name = name;
    this.passwordHash = passwordHash;
}
```

使用这种方式，不管是以函数的方式还是以构造函数的方式调用 User 函数，它都返回一个继承自 User.prototype 的对象。

```
var x = User("baravelli", "d8b74df393528d51cd19980ae0aa028e");
var y = new User("baravelli",
                 "d8b74df393528d51cd19980ae0aa028e");
x instanceof User; // true
y instanceof User; // true
```

这种模式的一个缺点是它需要额外的函数调用，因此代价有点高。而且，它也很难适用于可变参数函数（请参阅第 21 条和第 22 条），因为没有一种直接模拟 apply 方法将可变参数函数作为构造函数调用的方式。一种更为奇异的方式是利用 ES5 的 Object.create 函数。

```
function User(name, passwordHash) {
    var self = this instanceof User
             ? this
             : Object.create(User.prototype);
    self.name = name;
    self.passwordHash = passwordHash;

    return self;
}
```

Object.create 需要一个原型对象作为参数，并返回一个继承自该原型对象的新对象。因此，当以函数的方式调用该版本的 User 函数时，结果将返回一个继承自 User.prototype 的新对象，并且该对象具有已经初始化的 name 和 passwordHash 属性。

Object.create 只有在 ES5 环境中才是有效的，但是在一些旧的环境中可以通过创建一个局部的构造函数并使用 new 操作符初始化该构造函数来替代 Object.create。

```
if (typeof Object.create === "undefined") {
```

```
    Object.create = function(prototype) {
        function C() { }
        C.prototype = prototype;
        return new C();
    };
}
```

（请注意，这只实现了单参数版本的 Object.create 函数。真实版本的 Object.create 函数还接受一个可选的参数，该参数描述了一组定义在新对象上的属性描述符。）

如果使用 new 操作符调用该新版本的 User 函数会发生什么？多亏了构造函数覆盖模式，使用 new 操作符调用该函数的行为就如以函数调用它的行为一样。这能工作完全得益于 JavaScript 允许 new 表达式的结果可以被构造函数中的显式 return 语句所覆盖。当 User 函数返回 self 对象时，new 表达式的结果就变为 self 对象。该 self 对象可能是另一个绑定到 this 的对象。

防范误用构造函数可能并不总是值得费心去做，尤其是当仅仅是局部使用构造函数时。但是，理解如果以错误的方式调用构造函数会造成严重后果是很重要的。至少，文档化构造函数期望使用 new 操作符调用是很重要的，尤其是在跨大型代码库中共享构造函数或该构造函数来自一个共享库时。

提示

- 通过使用 new 操作符或 Object.create 方法在构造函数定义中调用自身使得该构造函数与调用语法无关。
- 当一个函数期望使用 new 操作符调用时，清晰地文档化该函数。

第 34 条：在原型中存储方法

JavaScript 完全有可能不借助原型进行编程。我们可以实现第 30 条中的 User 类，而不用在其原型中定义任何特殊的方法。

```
function User(name, passwordHash) {
    this.name = name;
    this.passwordHash = passwordHash;
    this.toString = function() {
        return "[User " + this.name + "]";
    };
    this.checkPassword = function(password) {
        return hash(password) === this.passwordHash;
    };
}
```

大多数情况下，这个类的行为与其原始实现版本几乎是一样的。但当我们构造多个 User 的实例时，一个重要的区别就暴露出来了。

```
var u1 = new User(/* ... */);
var u2 = new User(/* ... */);
var u3 = new User(/* ... */);
```

图 4.3 展示了这三个对象及它们的原型对象的结构图。每个实例都包含 toString 和 checkPassword 方法的副本，而不是通过原型共享这些方法，所以总共有 6 个函数对象。

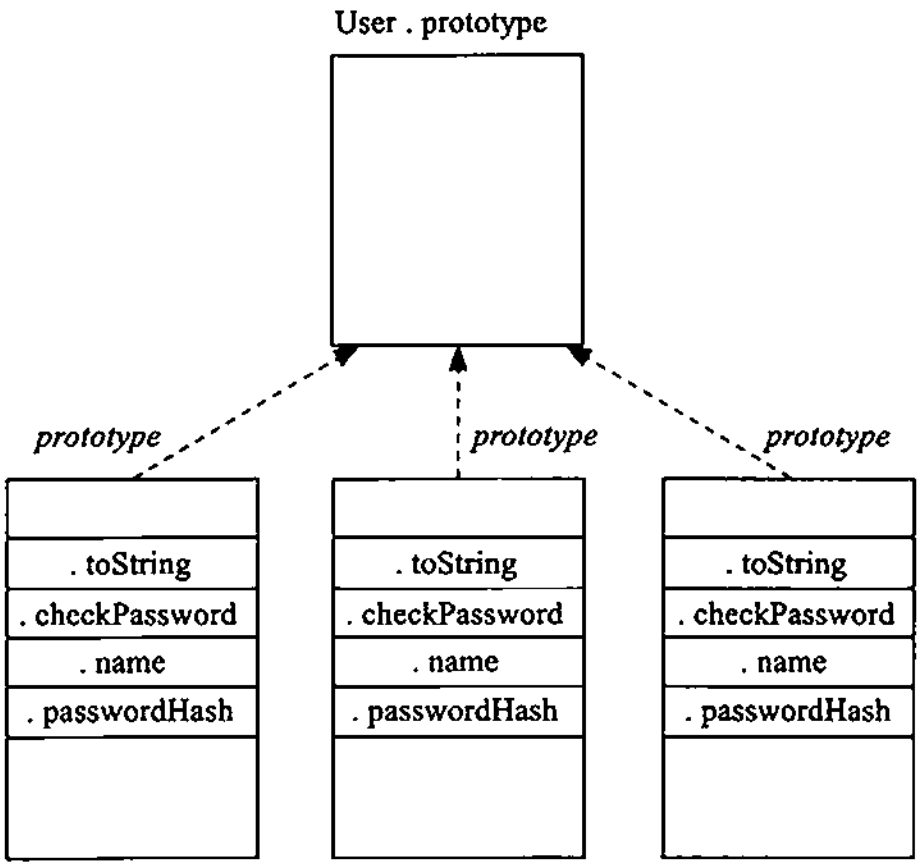

图 4.3 将方法存储在实例对象中

相反，图 4.4 显示了使用原始版本，这三个对象及其原型对象的结构图。toString 和 checkPassword 方法只被创建了一次，对象实例间通过原型来共享它们。

将方法存储在原型中，使其可以被所有的实例使用，而不需要存储方法实现的多个副本，也不需要给每个实例对象增加额外的属性。你可能认为将方法存储在实例对象中会优化方法查找的速度。例如，u3.toString() 方法，不需要

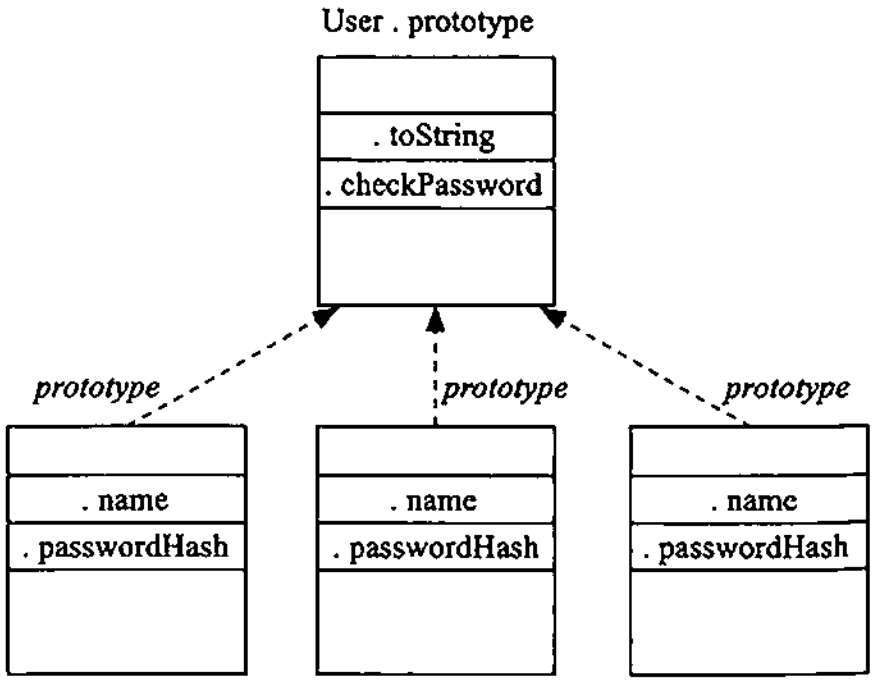

图 4.4 将方法存储在原型对象中

搜索原型链来查找 toString 的实现。然而，现代的 JavaScript 引擎深度优化了原型查找，所以将方法复制到实例对象并不一定保证速度有明显的提升。而且实例方法比起原型方法肯定会占用更多的内存。

- 将方法存储在实例对象中将创建该函数的多个副本，因为每个实例对象都有一份副本。
- 将方法存储于原型中优于存储在实例对象中。

第 35 条：使用闭包存储私有数据

JavaScript 的对象系统并没有特别鼓励或强制信息隐藏。所有的属性名都是一个字符串，任意一段程序都可以简单地通过访问属性名来获取相应的对象属性。例如，for...in 循环、ES5 的 Object.keys() 和 Object.getOwnPropertyNames() 函数等特性都能轻易地获取到对象的所有属性名。

通常，JavaScript 程序员都诉诸于编码规范来对待私有属性，而不是任何绝对的强制机制。例如，一些程序员使用命名规范给私有属性名前置或后置一个下划线字符（_）。这并没有强制信息隐藏，而只是表明对对象的正确行为操作的一个建议。用户不应该检查或修改该属性，以便该对象仍然能自由地改变其实现。

然而实际上，一些程序需要更高程度的信息隐藏。例如，一些安全敏感的平台或应用程序框架。它们希望发送对象到未授信的、缺乏对该对象内部风险干预的应用程序。强制信息隐藏能够派上用场的另外一种情形是频繁使用的程序库。在这些程序库中，当粗心的用户不小心地依赖或干扰了实现细节，一些微妙的 Bug 就会突然出现。

对于这些情形，JavaScript 为信息隐藏提供了一种非常可靠的机制——闭包。

闭包是一种简朴的数据结构。它们将数据存储到封闭的变量中而不提供对这些变量的直接访问。获取闭包内部结构的唯一方式是该函数显式地提供获取它的途径。换句话说，对象和闭包具有相反的策略：对象的属性会被自动地暴露出去，然而闭包中的变量会被自动地隐藏起来。

我们可以利用这一特性在对象中存储真正的私有数据。不是将数据作为对象的属性来存储，而是在构造函数中以变量的方式来存储它，并将对象的方法转变为引用这些变量的闭包。让我们再次回顾下第 30 条中的 User 类。

```
function User(name, passwordHash) {
    this.toString = function() {
        return "[User " + name + "]";
```

```
    };
    this.checkPassword = function(password) {
        return hash(password) === passwordHash;
    };
}
```

请注意，与其他的实现不同，该实现的 toString 和 checkPassword 方法是以变量的方式来引用 name 和 passwordHash 变量的，而不是以 this 属性的方式来引用。现在，User 的实例根本不包含任何实例属性，因此外部的代码不能直接访问 User 实例的 name 和 passwordHash 变量。

该模式的一个缺点是，为了让构造函数中的变量在使用它们的方法的作用域内，这些方法必须置于实例对象中。正如第 34 条所讨论的，这会导致方法副本的扩散。然而，对于那些更看重保障信息隐藏的情形来说，这点额外的代价是值得的。

- 闭包变量是私有的，只能通过局部的引用获取。
- 将局部变量作为私有数据从而通过方法实现信息隐藏。

第 36 条：只将实例状态存储在实例对象中

理解原型对象与其实例之间是一对多的关系对于实现正确的对象行为是至关重要的。一种错误的做法是不小心将每个实例的数据存储到了其原型中。例如，一个实现了树型数据结构的类可能将子节点存储在数组中。将存储子节点的数组放置在原型对象中将会导致实现被完全破坏。

```
function Tree(x) {
    this.value = x;
}

Tree.prototype = {
    children: [],                    // should be instance state!
    addChild: function(x) {
        this.children.push(x);
    }
};
```

如果我们使用这个类构造一棵树将会发生什么呢？

```
var left = new Tree(2);
left.addChild(1);
left.addChild(3);
```

```
var right = new Tree(6);
right.addChild(5);
right.addChild(7);

var top = new Tree(4);
top.addChild(left);
top.addChild(right);

top.children; // [1, 3, 5, 7, left, right]
```

每次调用 addChild 方法，都会将值添加到 Tree.prototype.children 数组中。Tree.prototype.children 数组包含了任何地方按序调用 addChild 方法时传入的所有节点。这使得 Tree 对象呈现出不相干的状态，如图 4.5 所示。

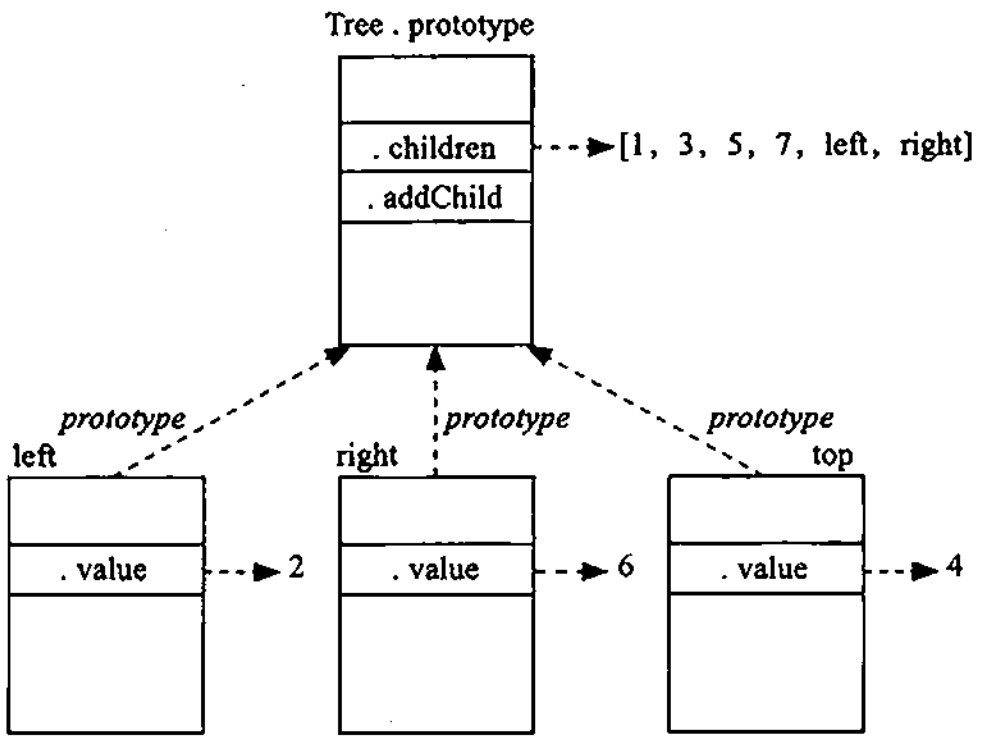

图 4.5 将实例状态存储在原型对象中

实现 Tree 类的正确方式是为每个实例对象创建一个单独的 children 数组。

```
function Tree(x) {
    this.value = x;
    this.children = []; // instance state
}
Tree.prototype = {
    addChild: function(x) {
        this.children.push(x);
    }
};
```

运行上面同一个例子的代码，我们可以得到期望的状态，如图 4.6 所示。

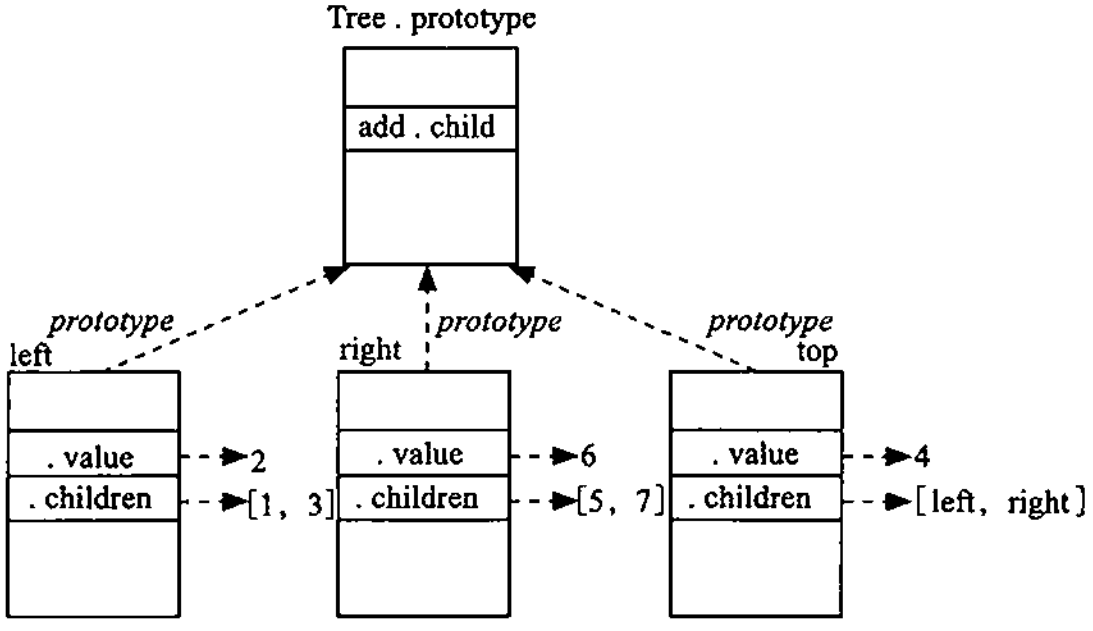

图 4.6 将实例状态存储在实例对象中

这个故事的寓意是共享有状态的数据可能会导致问题。通常在一个类的多个实例之间共享方法是安全的，因为方法通常是无状态的，这不同于通过 this 来引用实例状态。（因为方法调用的语法确保了 this 被绑定到实例对象，即使该方法是从原型中继承来的，共享方法仍然可以访问实例状态。）一般情况下，任何不可变的数据可以被存储在原型中从而被安全地共享。有状态的数据原则上也可以存储在原型中，只要你真正想共享它。然而迄今为止，在原型对象中最常见的数据是方法，而每个实例的状态都存储在实例对象中。


提示

- 共享可变数据可能会出问题，因为原型是被其所有的实例共享的。
- 将可变的实例状态存储在实例对象中。

第 37 条：认识到 this 变量的隐式绑定问题

CSV（逗号分隔型取值）文件格式是一种表格数据的简单文本表示。

```
Bösendorfer,1828,Vienna,Austria
Fazioli,1981,Sacile,Italy
Steinway,1853,New York,USA
```

我们可以编写一个简单的、可定制的读取 CSV 数据的类。（为了简单起见，我们摒弃了解析类似“hello,world”这样带有引号条目的功能。）尽管从其命名上看，它是基于逗号的，但是目前 CSV 也存在一些允许不同的字符作为分隔器的变种。因此，构造函数需要一个可选的分隔器字符数组并构造出一个自定义的正则表达式以将每一行分成不同的条目。

```
function CSVReader(separators) {
    this.separators = separators || [","];
```

```
    this.regexp =
        new RegExp(this.separators.map(function(sep) {
            return "\\" + sep[0];
        }).join("|"));
}
```

实现一个简单的 read 方法可以分为两步来处理。第一步，将输入字符串分为按行划分的数组；第二步，将数组的每一行再分为按单元格划分的数组。因此，结果应该是一个二维的字符串数组。使用 map 方法最为合适。

```
CSVReader.prototype.read = function(str) {
    var lines = str.trim().split(/\n/);
    return lines.map(function(line) {
        return line.split(this.regexp); // wrong this!
    });
};

var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n"); // [["a,b,c"], ["d,e,f"]]
```

表面看起来，这段简单的代码有一个严重而微妙的 Bug。传递给 line.map 的回调函数引用了 this，它期望能提取到 CSVReader 对象的 regexp 属性。然而，map 函数将其回调函数的接收者绑定到了 lines 数组，该 lines 数组并没有 regexp 属性。其结果是，this.regexp 产生 undefined 值，使得调用 line.split 陷入混乱。

导致该 Bug 的是这样一个事实：this 变量是以不同的方式被绑定的。正如第 18 条和第 25 条解释的一样，每个函数都有一个 this 变量的隐式绑定。该 this 变量的绑定值是在调用该函数时确定的。对于一个词法作用域的变量，你总能通过查找显式命名的绑定名（如在一个 var 声明的列表中或作为一个函数的参数）来识别出其绑定的接收者。相反，this 变量是隐式地绑定到最近的封闭函数。因此，对于 CSVReader.prototype.read 函数，this 变量的绑定不同于传递给 lines.map 回调函数的 this 绑定。

幸运的是，数组的 map 方法可以传入一个可选的参数作为其回调函数的 this 绑定，我们可以利用这一点。这与第 25 条中的 forEach 例子异曲同工。所以在这种情况下，修复该 Bug 的最简单的方法是将外部的 this 绑定通过 map 的第二个参数传递给回调函数。

```
CSVReader.prototype.read = function(str) {
    var lines = str.trim().split(/\n/);
    return lines.map(function(line) {
        return line.split(this.regexp);
    }, this); // forward outer this-binding to callback
};
```

```
var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n");
// [["a","b","c"], ["d","e","f"]]
```

目前，不是所有基于回调函数的 API 都是考虑周全的。假如 map 方法不接受这个额外的参数将会怎样？我们将需要另一种仍然能获取到外部函数 this 绑定的方式，以便回调函数仍然能引用它。直截了当的解决方案是使用词法作用域的变量来存储这个额外的外部 this 绑定的引用。

```
CSVReader.prototype.read = function(str) {
    var lines = str.trim().split(/\n/);
    var self = this; // save a reference to outer this-binding
    return lines.map(function(line) {
        return line.split(self.regexp); // use outer this
    });
};

var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n");
// [["a","b","c"], ["d","e","f"]]
```

对于此模式，程序员通常会使用变量名 self，以表明该变量的唯一目的是作为当前作用域的 this 绑定的额外别名。（此模式其他流行的变量名还有 me 和 that。）选择哪一个作为变量名并不是特别重要，重要的是选择一个通用的变量名，以使得其他程序员能快速地识别出该模式。

然而在 ES5 的环境中，另一种有效的方法是使用回调函数的 bind 方法。这与第 25 条中描述的方法类似。

```
CSVReader.prototype.read = function(str) {
    var lines = str.trim().split(/\n/);
    return lines.map(function(line) {
        return line.split(this.regexp);
    }.bind(this)); // bind to outer this-binding
};

var reader = new CSVReader();
reader.read("a,b,c\nd,e,f\n");
// [["a","b","c"], ["d","e","f"]]
```

提示

- this 变量的作用域总是由其最近的封闭函数所确定。
- 使用一个局部变量（通常命名为 self、me 或 that）使得 this 绑定对于内部函数是可用的。

第 38 条：在子类的构造函数中调用父类的构造函数

场景图（scene graph）是在可视化的程序中（如游戏或图形仿真场景）描述一个场景的对象集合。一个简单的场景包含了在该场景中的所有对象（称为角色），以及所有角色的预加载图像数据集，还包含一个底层图形显示的引用（通常被称为 context）。

```
function Scene(context, width, height, images) {
    this.context = context;
    this.width = width;
    this.height = height;
    this.images = images;
    this.actors = [];
}

Scene.prototype.register = function(actor) {
    this.actors.push(actor);
};

Scene.prototype.unregister = function(actor) {
    var i = this.actors.indexOf(actor);
    if (i >= 0) {
        this.actors.splice(i, 1);
    }
};

Scene.prototype.draw = function() {
    this.context.clearRect(0, 0, this.width, this.height);
    for (var a = this.actors, i = 0, n = a.length;
         i < n;
         i++) {
        a[i].draw();
    }
};
```

场景中所有的角色都继承自基类 Actor。基类 Actor 抽象出了一些通用的方法。每个角色都存储了其自身场景的引用以及坐标位置，然后将自身添加到场景的角色注册表中。

```
function Actor(scene, x, y) {
    this.scene = scene;
    this.x = x;
    this.y = y;
    scene.register(this);
}
```

为了能改变角色在场景中的位置，我们提供了一个 moveTo 方法。该方法改变角色的坐

标，然后重绘场景。

```
Actor.prototype.moveTo = function(x, y) {
    this.x = x;
    this.y = y;
    this.scene.draw();
};
```

当一个角色离开了场景，我们从场景图的注册表中删除它，并重新绘制场景。

```
Actor.prototype.exit = function() {
    this.scene.unregister(this);
    this.scene.draw();
};
```

想要绘制一个角色，我们需要查找它在场景图图像表中的图像。我们假设每个 actor 有一个 type 字段，可以用来查找它在图像表中的图像。一旦我们有了这个图像数据，就可以使用底层图形库将其绘制到图形上下文中。(本例使用 HTML Canvas API，它提供了一个 drawImage 方法用于绘制一个 Image 对象到 Web 页面的 <canvas> 元素中。)

```
Actor.prototype.draw = function() {
    var image = this.scene.images[this.type];
    this.scene.context.drawImage(image, this.x, this.y);
};
```

同样，我们可以通过角色的图像数据确定其尺寸。

```
Actor.prototype.width = function() {
    return this.scene.images[this.type].width;
};

Actor.prototype.height = function() {
    return this.scene.images[this.type].height;
};
```

我们将角色的特定类型实现为 Actor 的子类。例如，在街机游戏中太空飞船就会有一个扩展自 Actor 的 SpaceShip 类。像所有的类一样，SpaceShip 被定义为一个构造函数。但是为了确保 SpaceShip 的实例能作为角色被正确地初始化，其构造函数必须显式地调用 Actor 的构造函数。通过将接收者绑定到该新对象来调用 Actor 可以达到此目的。

```
function SpaceShip(scene, x, y) {
    Actor.call(this, scene, x, y);
    this.points = 0;
}
```

首先调用 Actor 的构造函数能确保通过 Actor 创建的所有实例属性都被添加到了新对象中。然后，SpaceShip 可以定义自身的实例属性，如飞船当前的积分数。为了使 SpaceShip 成

为 Actor 的一个正确的子类，其原型必须继承自 Actor.prototype。做这种扩展的最好的方式是使用 ES5 提供的 Object.create 方法。

```
SpaceShip.prototype = Object.create(Actor.prototype);
```

（第 33 条描述了在不支持 ES5 的环境中对 Object.create 的一种实现。）如果我们试图使用 Actor 的构造函数来创建 SpaceShip 的原型对象，会有几个问题。第一个问题是我们没有任何合理的参数传递给 Actor。

```
SpaceShip.prototype = new Actor();
```

当初始化 SpaceShip 的原型时，我们尚未创建任何能作为第一个参数来传递的场景。并且 SpaceShip 原型还不具有有效的 x 或 y 坐标。这些属性应当作为每个 SpaceShip 对象的实例属性，而不是 SpaceShip.prototype 的属性。更为严重的是，Actor 的构造函数会将 SpaceShip 的原型加入到场景的注册表中，而这绝对不是我们想做的。这是一种使用子类时司空见惯的现象。应当仅仅在子类构造函数中调用父类构造函数，而不是当创建子类原型时调用它。

一旦创建了 SpaceShip 的原型对象，我们就可以向其添加所有的可被实例共享的属性，包含一个用于在场景的图像数据表中检索的 type 名，以及一些太空飞船的特定方法。

```
SpaceShip.prototype.type = "spaceShip";

SpaceShip.prototype.scorePoint = function() {
    this.points++;
};

SpaceShip.prototype.left = function() {
    this.moveTo(Math.max(this.x - 10, 0), this.y);
};

SpaceShip.prototype.right = function() {
    var maxWidth = this.scene.width - this.width();
    this.moveTo(Math.min(this.x + 10, maxWidth), this.y);
};
```

图 4.7 为 SpaceShip 实例的继承层次结构图。注意 scene、x 以及 y 属性只被定义在实例对象中，而不是被定义在原型对象中，尽管 SpaceShip 是被 Actor 构造函数创建的。

提示

- 在子类构造函数中显式地传入 this 作为显式的接收者调用父类构造函数。
- 使用 Ojbect.create 函数来构造子类的原型对象以避免调用父类的构造函数。

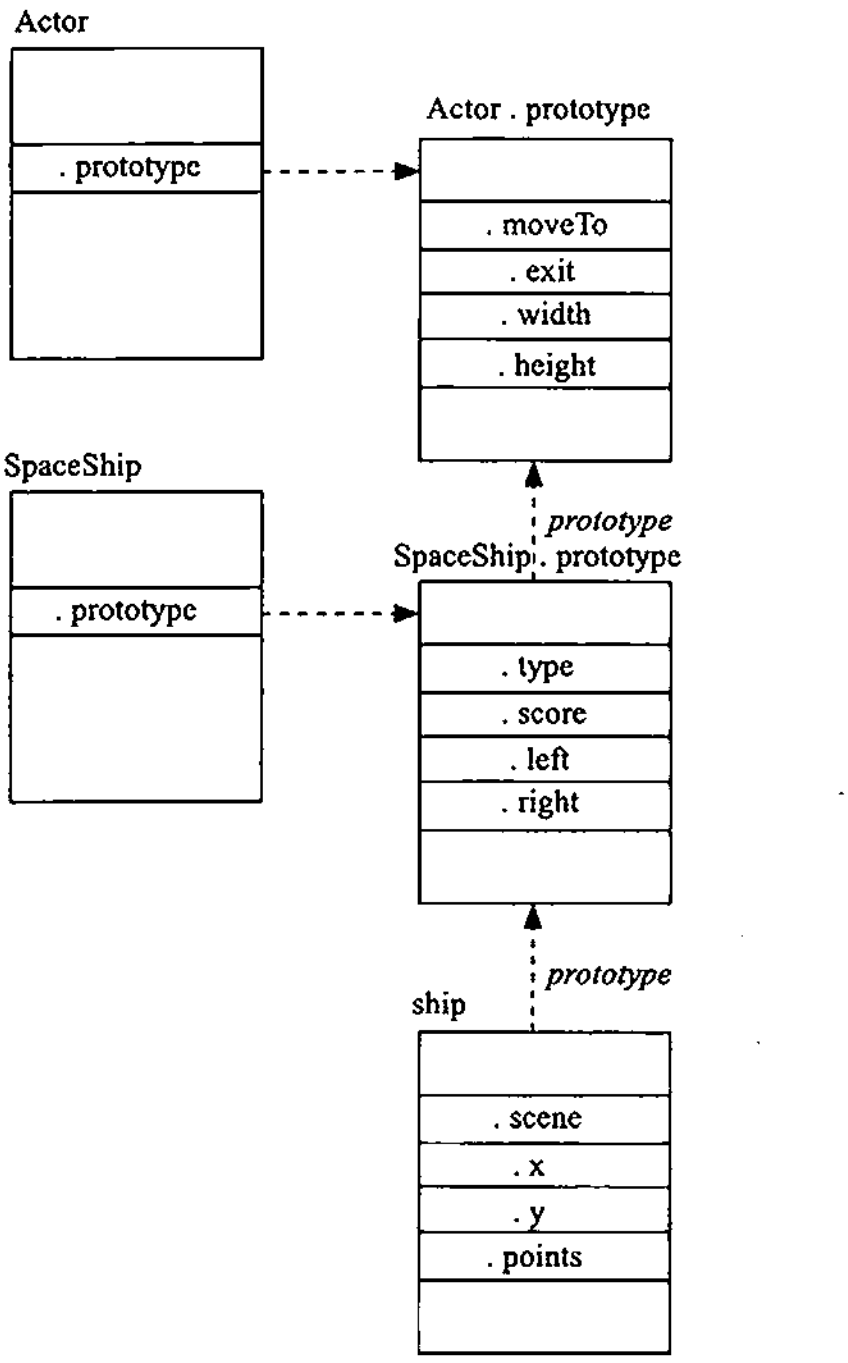

图 4.7 子类的继承层次结构

第 39 条：不要重用父类的属性名

假设我们想给第 38 条中的场景图库增加收集诊断信息的功能。这对于调试和性能分析是很有用的。为了做到这一点，我们要给每个 Actor 实例一个唯一的标识数。

```
function Actor(scene, x, y) {
    this.scene = scene;
    this.x = x;
    this.y = y;
    this.id = ++Actor.nextID;
    scene.register(this);
}

Actor.nextID = 0;
```

现在我们对 Actor 的子类做同样的事。假设，Alien 类代表太空飞船的敌人。除了其角色标识数外，我们还希望每个外星人有一个单独的外星人标识数。

```
function Alien(scene, x, y, direction, speed, strength) {
    Actor.call(this, scene, x, y);
    this.direction = direction;
    this.speed = speed;
    this.strength = strength;
    this.damage = 0;
    this.id = ++Alien.nextID; // conflicts with actor id!
}

Alien.nextID = 0;
```

这段代码导致了 Alien 类与其父类 Actor 之间的冲突。两个类都试图给实例属性 id 写数据。虽然每个类都认为该属性是“私有”的（即只有直接定义在该类中的方法才能获取该属性），然而事实是该属性存储在实例对象上并命名为一个字符串。如果在继承体系中的两个类指向相同的属性名，那么它们指向的是同一个属性。

因此，子类必须始终留意其父类使用的所有属性，即使那些属性在概念上是私有的。该例子显而易见的解决方法是对 Actor 标识数和 Alien 标识数使用不同的属性名。

```
function Actor(scene, x, y) {
    this.scene = scene;
    this.x = x;
    this.y = y;
    this.actorID = ++Actor.nextID; // distinct from alienID
    scene.register(this);
}

Actor.nextID = 0;

function Alien(scene, x, y, direction, speed, strength) {
    Actor.call(this, scene, x, y);
    this.direction = direction;
    this.speed = speed;
    this.strength = strength;
    this.damage = 0;
    this.alienID = ++Alien.nextID; // distinct from actorID
}

Alien.nextID = 0;
```

- 留意父类使用的所有属性名。

❑ 不要在子类中重用父类的属性名。

第 40 条：避免继承标准类

ECMAScript 标准库虽小，但它配备了许多重要的类，如 Array、Function 以及 Date 等。扩展这些类生成子类是很有诱惑力的，但不幸的是它们的定义具有很多特殊的行为，所以很难写出行为正确的子类。

Array 类是个很好的例子。一个操作文件系统的库可能希望创建一个抽象的目录，该目录继承了数组的所有行为。

```
function Dir(path, entries) {
    this.path = path;
    for (var i = 0, n = entries.length; i < n; i++) {
        this[i] = entries[i];
    }
}

Dir.prototype = Object.create(Array.prototype);
// extends Array
```

遗憾的是，这种方式破坏了数组的 length 属性的预期行为。

```
var dir = new Dir("/tmp/mysite",
                  ["index.html", "script.js", "style.css"]);
dir.length; // 0
```

失败的原因是 length 属性只对在内部被标记为“真正的”数组的特殊对象起作用。ECMAScript 标准规定它是一个不可见的内部属性，称为 [[Class]]。不要被这个名字所误导。

JavaScript 并不具有秘密的内部类系统。[[Class]] 的值仅仅是一个简单的标签。数组对象（通过 Array 构造函数或 [] 语法创建）被加上了值为“Array”的 [[Class]] 属性，函数被加上了值为“Function”的 [[Class]] 属性，以此类推。表 4.1 显示了在 ECMAScript 中定义的完整的 [[Class]] 属性值集合。

表 4.1 由 ECMAScript 标准定义的内部 [[Class]] 属性的值

[[Class]]	Construction
“Array”	new Array (…), […]
“Boolean”	new Boolean (…)
“Date”	new Date (…)
“Error”	new Error (…), new EvalError (…), new RangeError (…), new ReferenceError (…), new SyntaxError (…), new TypeError (…), new URIError (…)

（续）

[[Class]]	Construction
“Function”	new Function（…），function（…）{…}
“JSON”	JSON
“Math”	Math
“Number”	new Number（…）
“Object”	new object（…），{…}，new MyClass（…）
“RegExp”	new RegExp（…），/…/
“String”	new String（…）

那么，神奇的 [[Class]] 属性对 length 做了什么呢？事实证明，length 的行为只被定义在内部属性 [[Class]] 的值为“ Array”的特殊对象中。对于这些对象，JavaScript 保持 length 属性与该对象的索引属性的数量同步。如果给该对象添加了更多的索引属性，length 属性会自动增加。如果减少了 length 属性，也会自动删除任何索引大于该新值的索引属性。

但当我们扩展 Array 类时，子类的实例并不是通过 new Array() 或字面量 [] 语法创建的。所以，Dir 的实例的 [[Class]] 属性值为“Object”。

甚至我们可以测试它。默认的 Object.prototype.toString 方法可以通过查询其接收者的内部 [[Class]] 属性来创建对象的通用描述。所以你可以传递任何给定的对象来显式地调用它。

```
var dir = new Dir("/", []);
Object.prototype.toString.call(dir); // "[object Object]"
Object.prototype.toString.call([]);  // "[object Array]"
```

其结果是，Dir 的实例并未继承数组的 length 属性所期望的特殊行为。

更好的实现是定义一个 entries 数组的实例属性。

```
function Dir(path, entries) {
    this.path = path;
    this.entries = entries; // array property
}
```

在原型中重新定义 Array 的方法，我们可以将这些相应的方法委托给 entries 属性来完成。

```
Dir.prototype.forEach = function(f, thisArg) {
    if (typeof thisArg === "undefined") {
        thisArg = this;
    }
    this.entries.forEach(f, thisArg);
};
```

ECMAScript 标准库中的大多数构造函数都有类似的问题。某些属性或方法期望具有正确的 [[Class]] 属性或其他特殊的内部属性，然而子类却无法提供。基于这个原因，最好避免继承以下的标准类：Array、Boolean、Date、Function、Number、RegExp 或 String。

提示

- 继承标准类往往会由于一些特殊的内部属性（如 [[Class]]）而被破坏。
- 使用属性委托优于继承标准类。

第 41 条：将原型视为实现细节

一个对象给其使用者提供了轻量、简单、强大的操作集。使用者与一个对象最基本的交互是获取其属性值和调用其方法。这些操作不是特别在意属性存储在原型继承结构的哪个位置。随着时间的推移，实现对象时可能会将一个属性实现在对象原型链的不同位置，但是只要其值保持不变，那么这些基本操作的行为也不变。简而言之，原型是一种对象行为的实现细节。

与此同时，JavaScript 提供了便利的内省机制（introspection mechanisms）来检查对象的细节。Object.prototype.hasOwnProperty 方法确定一个属性是否为对象“自己的”属性（即一个实例属性），而完全忽视原型继承结构。Object.getPrototypeOf 和 __proto__ 特性（请参阅第 30 条）允许程序员遍历对象的原型链并单独查询其原型对象。这些特性是非常强大的，有时也是很有用的。

但是，优秀的程序员知道什么时候考虑抽象边界。检查实现细节（即使没有修改它们）也会在程序的组件之间创建依赖。如果对象的生产者修改了实现细节，那么依赖于这些对象的使用者就会被破坏。很难诊断出这类 Bug，因为它们构成了超距作用（action at a distance），即一个作者修改了一个组件的实现，导致另一个组件（通常由不同的程序员编写）被破坏。

类似地，JavaScript 并不区分对象的公有属性和私有属性（请参阅第 35 条）。取而代之的是，你的责任是依靠文档和约束。如果一个程序库提供的对象的属性没有文档化或者明文标注为内部属性，对于使用者来说，最好不要干涉那些属性。

提示

- 对象是接口，原型是实现。
- 避免检查你无法控制的对象的原型结构。
- 避免检查实现在你无法控制的对象内部的属性。

第 42 条：避免使用轻率的猴子补丁

在经历了第 41 条中对违反抽象原则行为的痛骂之后，现在让我们来考虑终极违例。由于对象共享原型，因此每一个对象都可以增加、删除或修改原型的属性。这个有争议的实践

通常被称为猴子补丁（monkey-patching）。

猴子补丁的吸引力在于它的强大。数组缺少一个有用的方法吗？你自己就可以增加它。

```
Array.prototype.split = function(i) { // alternative #1
    return [this.slice(0, i), this.slice(i)];
};
```

Voilà: Every array instance has a split method.

但是当多个库以不兼容的方式给同一个原型打猴子补丁时，问题便出现了。另外的库可能使用同一个方法名给 Array.prototype 打猴子补丁。

```
Array.prototype.split = function() { // alternative #2
    var i = Math.floor(this.length / 2);
    return [this.slice(0, i), this.slice(i)];
};
```

现在，任一对数组 split 方法的使用都大约有 50% 的机会被破坏，这取决于它们期望这两个方法中的哪一个被调用。

至少，任一修改共享原型（如 Array.prototype）的程序库都应当清晰地记录其修改。这至少能给使用者在关于不同库之间潜在的冲突提供足够的警告。但是，两个以冲突的方式给原型打猴子补丁的程序库不能在同一个程序中使用。一种替代的方法是，如果库仅仅是将给原型打猴子补丁作为一种便利，那么可以将这些修改置于一个函数中，用户可以选择调用或忽略。

```
function addArrayMethods() {
    Array.prototype.split = function(i) {
        return [this.slice(0, i), this.slice(i)];
    };
};
```

当然，这种方法只有在程序库提供了 addArrayMethods 函数时才能工作，而实际上并不依赖于 Array.prototype.split 函数。

尽管猴子补丁很危险，但是有一种特别可靠而且有价值的使用场景：polyfill。JavaScript 程序和库经常部署在多个平台，例如不同运营商提供的 Web 浏览器的不同版本。这些平台实现了多少个标准 API 可能是有区别的。例如，ES5 定义一些新的 Array 方法（如 forEach、map 和 filter），而一些浏览器版本可能并不支持这些方法。这些缺失的方法的行为是由广泛支持的标准所定义的，而且许多程序和库都可能依赖这些方法。由于它们的行为是标准化的，因此实现这些方法并不会造成与库之间不兼容性类似的风险。事实上，多个库都可以给同一个标准方法提供实现（假设它们都被正确地实现），因为它们都实现了相同的标准 API。

你可以通过使用带有测试条件的守护猴子补丁来安全地弥补这些平台的差距。

```
if (typeof Array.prototype.map !== "function") {
    Array.prototype.map = function(f, thisArg) {
        var result = [];
        for (var i = 0, n = this.length; i < n; i++) {
            result[i] = f.call(thisArg, this[i], i);
        }
        return result;
    };
}
```

测试 Array.prototype.map 是否存在，以确保内置的实现不被覆盖。内置的实现很可能更高效、测试更充分。

- 避免使用轻率的猴子补丁。
- 记录程序库所执行的所有猴子补丁。
- 考虑通过将修改置于一个导出函数中，使猴子补丁成为可选的。
- 使用猴子补丁为缺失的标准 API 提供 polyfills。

第 5 章

数组和字典

对象是 JavaScript 中最万能的数据结构。取决于不同的环境，对象可以表示一个灵活的键值关联记录，一个继承了方法的面向对象数据抽象，一个密集或稀疏的数组，或一个散列表。当然，使用这一多用途的工具需要掌握针对不同需求的不同惯用法。在上一章中，我们学习了结构对象和继承的用法。在本章中，我们来解决将对象作为集合（比如不同数目元素的聚集数据结构）的用法。

第 43 条：使用 Object 的直接实例构造轻量级的字典

JavaScript 对象的核心是一个字符串属性名称与属性值的映射表。这使得使用对象实现字典易如反掌，因为字典就是可变长的字符串与值的映射集合。JavaScript 甚至提供了枚举一个对象属性名的利器——for...in 循环。

```
var dict = { alice: 34, bob: 24, chris: 62 };
var people = [];

for (var name in dict) {
    people.push(name + ": " + dict[name]);
}

people; // ["alice: 34", "bob: 24", "chris: 62"]
```

但是，每个对象还继承了其原型对象中的属性（请参阅第 4 章），for...in 循环除了枚举出对象“自身”的属性外，还会枚举出继承过来的属性。例如，如果我们创建一个自定义的字典类并将其元素作为该字典对象自身的属性来存储会怎么样？

```
function NaiveDict() { }
```

```
NaiveDict.prototype.count = function() {
    var i = 0;
    for (var name in this) { // counts every property

        i++;
    }
    return i;
};

NaiveDict.prototype.toString = function() {
    return "[object NaiveDict]";
};

var dict = new NaiveDict();

dict.alice = 34;
dict.bob = 24;
dict.chris = 62;

dict.count(); // 5
```

问题在于我们使用同一个对象来存储 NaiveDict 数据结构的固定属性（count 和 toString）和特定字典的变化条目（alice、bob 和 chris）。因此，当调用 count 来枚举字典的所有属性时，它会枚举出所有的属性（count、toString、alice、bob 和 chris），而不是仅仅枚举出我们关心的条目。一个改进的 Dict 类请参阅第 45 条。该 Dict 类实现不是将其元素作为实例属性来存储，取而代之的是提供 dict.get(key) 和 dict.set(key, value) 方法。这一条中，我们重点关注将对象属性作为字典元素的模式。

一个相似的错误是使用数组类型来表示字典。熟悉 Perl 和 PHP 等语言的程序员尤其容易掉入这个陷阱。这些语言中通常将字典称为“关联数组”。极具欺骗性的是，由于我们可以给任意类型的 JavaScript 对象增加属性，因此这种使用模式有时似乎能工作。

```
var dict = new Array();

dict.alice = 34;
dict.bob = 24;
dict.chris = 62;

dict.bob; // 24
```

不幸的是，这段代码面对原型污染时很脆弱。原型污染是指当枚举字典的条目时，原型对象中的属性可能会导致出现一些不期望的属性。例如，应用程序中的其他库可能决定增加一些便利的方法到 Array.prototype 中。

```
Array.prototype.first = function() {
    return this[0];
};

Array.prototype.last = function() {
    return this[this.length - 1];
};
```

现在，我们尝试枚举数组的元素看看会发生什么。

```
var names = [];

for (var name in dict) {
    names.push(name);
}

names; // ["alice", "bob", "chris", "first", "last"]
```

这告诉我们将对象作为轻量级字典的首要原则是：应该仅仅将 Object 的直接实例作为字典，而不是其子类（例如 NaiveDict），当然也不是数组。例如，我们可以简单地将上例中的 new Array() 替换为 new Object()，甚至直接使用空对象字面量。这样的结果很难受到原型污染的影响。

```
var dict = {};

dict.alice = 34;
dict.bob = 24;
dict.chris = 62;

var names = [];

for (var name in dict) {
    names.push(name);
}

names; // ["alice", "bob", "chris"]
```

现在，新版本仍然不能保证对于原型污染是安全的。任何人仍然能增加属性到 Object.prototype 中，我们又会面对同样的问题。但是通过使用 Object 的直接实例，我们可以将风险仅仅局限于 Object.prototype。

那么，这种做法为什么更好呢？举例来说，正如第 47 条解释的一样，所有人都不应当增加属性到 Object.prototype 中，因为这样做可能会污染 for...in 循环。相比之下，增加属性到 Array.prototype 中是合理的。例如，第 42 条解释了如何在不支持数组标准方法的环境中将这些方法增加到 Array.prototype 中。这些属性会导致污染 for...in 循环。类似地，一个用户

自定义的类通常也会含有其原型中的属性。坚持 Object 的直接实例原则（总是遵守第 47 条的规则）可以使得 for...in 循环摆脱原型污染的影响。

但是要当心：正如第 44、45 条证实的一样，这条规则对于构建行为正确的字典是必要非充分的。虽然轻量级字典很方便，但是它们却面临许多危险。学习这 3 条规则是很重要的。或者如果你不喜欢记住这些规则，你可以使用类似第 45 条中的 Dict 类。

- 使用对象字面量构建轻量级字典。
- 轻量级字典应该是 Object.prototype 的直接子类，以使 for...in 循环免受原型污染。

第 44 条：使用 null 原型以防止原型污染

防止原型污染的最简单的方式之一就是一开始就不使用原型。但在 ES5 未发布之前，并没有标准的方式创建一个空原型的新对象。你可能会尝试设置一个构造函数的原型属性为 null 或者 undefined。

```
function C() { }
C.prototype = null;
```

但实例化该构造函数仍然得到的是 Object 的实例。

```
var o = new C();
Object.getPrototypeOf(o) === null;                // false
Object.getPrototypeOf(o) === Object.prototype;    // true
```

ES5 首先提供了标准方法来创建一个没有原型的对象。Object.create 函数能够使用一个用户指定的原型链和一个属性描述符动态地构造对象。属性描述符描述了新对象属性的值及特性。通过简单地传递一个 null 原型参数和一个空的描述符，我们就可以建立一个真正的空对象。

```
var x = Object.create(null);
Object.getPrototypeOf(o) === null;    // true
```

原型污染无法影响这样的对象的行为。

一些不支持 Object.create 函数的旧的 JavaScript 环境可能支持另一种值得一提的方式。在许多环境中，特殊的属性 __proto__（参阅第 31 条和第 32 条）提供了对对象内部原型链的读写访问。对象字面量语法也支持初始化一个原型链为 null 的新对象。

```
var x = { __proto__: null };
x instanceof Object;                    // false (non-standard)
```

这样的语法同样方便，但有了 Object.create 函数后，Object.create 函数是更值得信赖的

方式。__proto__ 属性是非标准的并且并不是所有使用都是可移植的。JavaScript 的实现不能保证在未来仍然支持它，所以应当尽可能地坚持使用标准的 Object.create 函数。

不幸的是，虽然非标准的 __proto__ 可以解决一些问题，但也带来了自身的额外问题，即阻止自由原型（prototype-free）对象作为真正健壮的字典实现。第 45 条描述了在某些 JavaScript 环境中，属性关键字“__proto__”自身是如何污染对象的，甚至可以污染没有原型的对象。如果不能确定字典中从未将字符串“__proto__”作为 key 使用，那么你应当考虑使用第 45 条中描述的更健壮的 Dict 类。

提示

- 在 ES5 环境中，使用 Object.create(null) 创建的自由原型的空对象是不太容易被污染的。
- 在一些较老的环境中，考虑使用 { __proto__: null }。
- 但要注意 __proto__ 既不标准，也不是完全可移植的，并且可能会在未来的 JavaScript 环境中去除。
- 绝不要使用“__proto__”名作为字典中的 key，因为一些环境将其作为特殊的属性对待。

第 45 条：使用 hasOwnProperty 方法以避免原型污染

第 43 和第 44 条讨论了属性枚举，但是我们并没有解决属性查找中原型污染的问题。很容易想到使用 JavaScript 的原生对象操作语法作为字典的所有操作。

```
"alice" in dict;  // membership test
dict.alice;       // retrieval
dict.alice = 24;  // update
```

但是谨记住 JavaScript 的对象操作总是以继承的方式工作。即使是一个空的对象字面量也继承了 Object.prototype 的大量属性。

```
var dict = {};

"alice" in dict;          // false
"bob" in dict;            // false
"chris" in dict;          // false
"toString" in dict;       // true
"valueOf" in dict;        // true
```

幸运的是，Object.prototype 提供了 hasOwnProperty 方法。当测试字典条目时它可以避免原型污染，因此正好是我们需要的方法。

```
dict.hasOwnProperty("alice");    // false
dict.hasOwnProperty("toString"); // false
dict.hasOwnProperty("valueOf");  // false
```

类似地，我们可以通过在属性查找时使用一个测试来防止其受污染的影响。

```
dict.hasOwnProperty("alice") ? dict.alice : undefined;
dict.hasOwnProperty(x) ? dict[x] : undefined;
```

不幸的是，我们并没有完全解决问题。当调用 dict.hasOwnProperty 时，我们请求查找 dict 对象的 hasOwnProperty 方法。通常情况下，该方法会简单地继承自 Object.prototype 对象。然而如果在字典中存储一个同为“hasOwnProperty”名称的条目，那么原型中的 hasOwnProperty 方法不能再被获取到。

```
dict.hasOwnProperty = 10;
dict.hasOwnProperty("alice");
// error: dict.hasOwnProperty is not a function
```

你可能认为字典绝不会存储像“hasOwnProperty”这样奇异的属性名。当然，这取决于你给定的应用程序环境是否有你期望的情形。然而这确实可能发生，尤其是当使用外部文件，网络资源或用户输入接口给字典填充条目时。你无法控制这些来自第三方的数据，因此你就无法确定字典中最终有哪些属性名。

最安全的方法是不做任何假设。我们可以采用第 20 条中描述的 call 方法，而不用将 hasOwnProperty 作为字典的方法来调用。首先，我们提前在任何安全的位置提取出 hasOwnProperty 方法：

```
var hasOwn = Object.prototype.hasOwnProperty;
```

或更为简明地：

```
var hasOwn = {}.hasOwnProperty;
```

现在有了绑定到正确函数的局部变量，我们可以通过函数的 call 方法在任意对象上调用它。

```
hasOwn.call(dict, "alice");
```

不管其接收者的 hasOwnProperty 方法是否被覆盖，该方法都能工作。

```
var dict = {};

dict.alice = 24;
hasOwn.call(dict, "hasOwnProperty"); // false
hasOwn.call(dict, "alice");          // true

dict.hasOwnProperty = 10;
hasOwn.call(dict, "hasOwnProperty"); // true
hasOwn.call(dict, "alice");          // true
```

为了避免在所有查找属性的地方都插入这段样本代码，我们可以将该模式抽象到 Dict 的构造函数中。该构造函数封装了所有在单一数据类型定义中编写健壮字典的技术细节。

```
function Dict(elements) {
    // allow an optional initial table
    this.elements = elements || {};    // simple Object
}

Dict.prototype.has = function(key) {
    // own property only
    return {}.hasOwnProperty.call(this.elements, key);
};

Dict.prototype.get = function(key) {
    // own property only
    return this.has(key)
         ? this.elements[key]
         : undefined;
};

Dict.prototype.set = function(key, val) {
    this.elements[key] = val;
};

Dict.prototype.remove = function(key) {
    delete this.elements[key];
};
```

请注意我们并没有保护 Dict.prototype.set 函数的实现，因为即使在 Object.prototype 中已经存在一个同名的属性，给字典对象增加该 key 仍会成为 elements 对象自己的属性。

该抽象比使用 JavaScript 默认的对象语法更健壮，而且也同样方便使用。

```
var dict = new Dict({
    alice: 34,
    bob: 24,
    chris: 62
});

dict.has("alice");   // true
dict.get("bob");     // 24
dict.has("valueOf"); // false
```

回顾第 44 条，在一些 JavaScript 的环境中，特殊的属性名 __proto__ 可能导致其自身的污染问题。在某些环境中，__proto__ 属性只是简单地继承自 Object.prototype，因此空对象（幸运地）是真正的空对象。

```
var empty = Object.create(null);
"__proto__" in empty;
// false (in some environments)

var hasOwn = {}.hasOwnProperty;
hasOwn.call(empty, "__proto__");
// false (in some environments)
```

在其他的环境中，只有 in 操作符输入为 true。

```
var empty = Object.create(null);
"__proto__" in empty;              // true (in some environments)

var hasOwn = {}.hasOwnProperty;
hasOwn.call(empty, "__proto__"); // false (in some
environments)
```

但是不幸的是，某些环境会因为存在一个实例属性 __proto__ 而永久地污染所有的对象。

```
var empty = Object.create(null);
"__proto__" in empty;              // true (in some environments)

var hasOwn = {}.hasOwnProperty;
hasOwn.call(empty, "__proto__"); // true (in some environments)
```

这意味着，在不同的环境中，下面的代码可能有不同的结果。

```
var dict = new Dict();
dict.has("__proto__"); // ?
```

为了达到最大的可移植性和安全性，我们没有其他选择，只能为每个 Dict 方法的 "__proto__" 关键字增加一种特例。结果便是下面更复杂但更安全的最终实现。

```
function Dict(elements) {
    // allow an optional initial table
    this.elements = elements || {};      // simple Object
    this.hasSpecialProto = false;        // has "__proto__" key?
    this.specialProto = undefined;       // "__proto__" element
}

Dict.prototype.has = function(key) {
    if (key === "__proto__") {
        return this.hasSpecialProto;
    }
    // own property only
    return {}.hasOwnProperty.call(this.elements, key);
};

Dict.prototype.get = function(key) {
    if (key === "__proto__") {
```

```
        return this.specialProto;
    }
    // own property only
    return this.has(key)
         ? this.elements[key]
         : undefined;
};

Dict.prototype.set = function(key, val) {
    if (key === "__proto__") {
        this.hasSpecialProto = true;

        this.specialProto = val;
    } else {
        this.elements[key] = val;
    }
};

Dict.prototype.remove = function(key) {
    if (key === "__proto__") {
        this.hasSpecialProto = false;
        this.specialProto = undefined;
    } else {
        delete this.elements[key];
    }
};
```

不管环境是否处理 __proto__ 属性，该实现保证是可工作的，因为它避免了到处处理该名称的属性。

```
var dict = new Dict();
dict.has("__proto__"); // false
```

提示

- 使用 hasOwnProperty 方法避免原型污染。
- 使用词法作用域和 call 方法避免覆盖 hasOwnProperty 方法。
- 考虑在封装 hasOwnProperty 测试样板代码的类中实现字典操作。
- 使用字典类避免将“__proto__”作为 key 来使用。

第 46 条：使用数组而不要使用字典来存储有序集合

直观地说，一个 JavaScript 对象是一个无序的属性集合。获取和设置不同的属性与顺序无关，都会以大致相同的效率产生相同的结果。ECMAScript 标准并未规定属性存储的任何

特定顺序，甚至对于枚举对象也未涉及。

但这将导致的问题是：for...in 循环会挑选一定的顺序来枚举对象的属性。由于标准允许 JavaScript 引擎自由选择一个顺序，它们的选择会微妙地改变程序行为。一个常见的错误是提供一个 API，要求一个对象表示一个从字符串到值的有序映射，例如创建一个有序的报表。

```
function report(highScores) {
    var result = "";
    var i = 1;
    for (var name in highScores) { // unpredictable order
        result += i + ". " + name + ": " +

                  highScores[name] + "\n";
        i++;
    }
    return result;
}

report([{ name: "Hank", points: 1110100 },
        { name: "Steve", points: 1064500 },
        { name: "Billy", points: 1050200 }]);
// ?
```

由于不同的环境可以选择以不同的顺序来存储和枚举对象属性，所以这个函数会导致产生不同的字符串，得到顺序混乱的“最高分”报表。

请记住你的程序是否依赖对象枚举的顺序并不总是显而易见的。如果没有在多个 JavaScript 环境中测试过你的程序，你甚至可能不会注意到程序的行为会因为一个 for ... in 循环的确切顺序而被改变。

如果你需要依赖一个数据结构中的条目顺序，请使用数组而不是字典。如果上述例子中的 report 函数的 API 使用一个对象数组而不是单个对象，那么它完全可以工作在任何 JavaScript 环境中。

```
function report(highScores) {
    var result = "";
    for (var i = 0, n = highScores.length; i < n; i++) {
        var score = highScores[i];
        result += (i + 1) + ". " +
                  score.name + ": " + score.points + "\n";
    }
    return result;
}

report([{ name: "Hank", points: 1110100 },
        { name: "Steve", points: 1064500 },
```

```
        { name: "Billy", points: 1050200 }]);
// "1. Hank: 1110100\n2. Steve: 1064500\n3. Billy: 1050200\n"
```

通过接受一个对象数组，每个对象含有 name 和 points 属性，这个版本可以以可预见的顺序遍历从 0 到 highScores.length-1 的所有元素。

一个微妙的顺序依赖的典型例子是浮点型运算。假设有一个映射标题和等级的电影字典。

```
var ratings = {
    "Good Will Hunting": 0.8,
    "Mystic River": 0.7,

    "21": 0.6,
    "Doubt": 0.9
};
```

正如在第 2 条中讲过的一样，浮点型算术运算的四舍五入会导致对计算顺序的微妙依赖。当组合未定义顺序的枚举时，可能会导致循环不可预知。

```
var total = 0, count = 0;
for (var key in ratings) { // unpredictable order
    total += ratings[key];
    count++;
}
total /= count;
total; // ?
```

事实证明，流行的 JavaScript 环境实际上使用不同的顺序执行这个循环。例如，一些环境根据加入对象的顺序来枚举对象的 key，从而进行有效的计算。

```
(0.8 + 0.7 + 0.6 + 0.9) / 4     // 0.75
```

其他环境总是先枚举潜在的数组索引，然后才是其他 key。由于电影 21 的名字恰好是一个可行的数组索引，它首先被枚举，导致如下结果：

```
(0.6 + 0.8 + 0.7 + 0.9) / 4     // 0.7499999999999999
```

在这种情况下，更好的表示方式是在字典中使用整数值。因为整数加法可以以任意顺序执行。这样，敏感的除法运算只会发生在最后，最关键的是发生在循环完成之后。

```
(8 + 7 + 6 + 9) / 4 / 10     // 0.75
(6 + 8 + 7 + 9) / 4 / 10     // 0.75
```

通常当执行 for...in 循环时应当时刻小心，确保执行操作的行为与顺序无关。

提示

- 使用 for...in 循环来枚举对象属性应当与顺序无关。

- 如果聚集运算字典中的数据，确保聚集操作与顺序无关。
- 使用数组而不是字典来存储有序集合。

第 47 条：绝不要在 Object.prototype 中增加可枚举的属性

for...in 循环非常便利，然而正如我们在第 43 条中所看到的一样，它很容易受到原型污染的影响。到目前为止，for...in 循环最常见的用法是枚举字典中的元素。这暗示我们如果想允许对字典对象使用 for...in 循环，那么不要在共享的 Object.prototype 中增加可枚举的属性。

这条规则可能会令人非常失望。在 Object.prototype 中增加便利的方法以使所有的对象都能共享，还有什么比这更强大的呢？例如，如果我们增加一个产生对象属性名数组的 allKeys 方法将会怎么样？

```
Object.prototype.allKeys = function() {
    var result = [];
    for (var key in this) {
        result.push(key);
    }
    return result;
};
```

遗憾的是，该方法也污染了其自身。

```
({ a: 1, b: 2, c: 3 }).allKeys(); // ["allKeys", "a", "b", "c"]
```

当然，我们可以改进 allKeys 方法忽略掉 Object.prototype 中的属性。但是自由伴随责任而生，我们应该对使用该高度共享原型对象的所有人所造成的影响有所行动。仅仅通过在 Object.prototype 中增加单个属性我们就能强制每个人在任何地方都能保护 for...in 循环不被污染。

更为友好的是将 allKeys 定义为一个函数而不是方法，虽然这稍微有点不方便。

```
function allKeys(obj) {
    var result = [];
    for (var key in obj) {
        result.push(key);
    }
    return result;
}
```

如果你确实想在 Object.prototype 中增加属性，ES5 提供了一种更加友好的机制。

Object.defineProperty 方法可以定义一个对象的属性并指定该属性的元数据。例如，我们可以用与之前完全一样的方式定义上面的属性而通过设置其可枚举属性为 false 使其在 for...in

循环中不可见。

```
Object.defineProperty(Object.prototype, "allKeys", {
    value: function() {
        var result = [];
        for (var key in this) {
            result.push(key);
        }
        return result;
    },
    writable: true,
    enumerable: false,
    configurable: true
});
```

诚然，这段代码有点晦涩。但这一版本有明显的优势，它不会污染其他所有 Object 实例的所有 for...in 循环。事实上，针对其他对象使用这一技术也是值得的。每当你需要增加一个不应该在 for...in 循环中出现的属性时，Obejct.defineProperty 便是你的选择。

- 避免在 Object.prototype 中增加属性。
- 考虑编写一个函数代替 Object.prototype 方法。
- 如果你确实需要在 Object.prototype 中增加属性，请使用 ES5 中的 Object.defineProperty 方法将它们定义为不可枚举的属性。

第 48 条：避免在枚举期间修改对象

一个社交网络有一组成员，每个成员有一个存储其朋友信息的注册列表。

```
function Member(name) {
    this.name = name;
    this.friends = [];
}

var a = new Member("Alice"),
    b = new Member("Bob"),
    c = new Member("Carol"),
    d = new Member("Dieter"),
    e = new Member("Eli"),
    f = new Member("Fatima");

a.friends.push(b);
```

```
b.friends.push(c);
c.friends.push(e);
d.friends.push(b);
e.friends.push(d, f);
```

搜索该网络意味着需要遍历该社交网络图（参见图 5.1）。这通常通过工作集（work-set）来实现。工作集以单个根节点开始，然后添加发现的节点，移除访问过的节点。使用 for...in 循环来实现该遍历是很有诱惑力的。

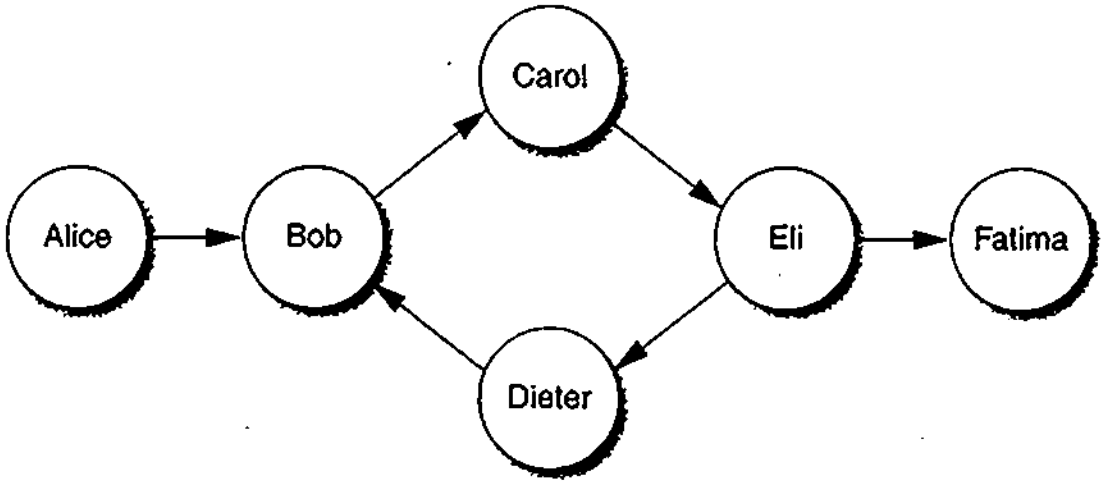

图 5.1 社交网络图

```
Member.prototype.inNetwork = function(other) {
    var visited = {};
    var workset = {};

    workset[this.name] = this;

    for (var name in workset) {
        var member = workset[name];
        delete workset[name];   // modified while enumerating
        if (name in visited) {  // don't revisit members
            continue;
        }
        visited[name] = member;
        if (member === other) { // found?
            return true;
        }
        member.friends.forEach(function(friend) {
            workset[friend.name] = friend;
        });
    }

    return false;
};
```

不幸的是，在许多 JavaScript 环境中这段代码根本不能工作：

```
a.inNetwork(f); // false
```

发生了什么呢？这表明 for...in 循环并没有要求枚举对象的修改与当前保持一致。事实上，ECMAScript 对并发修改在不同 JavaScript 环境下的行为的规范留有余地。特别是，标准规定了：

如果被枚举的对象在枚举期间添加了新的属性，那么在枚举期间并不能保证新添加的属性能够被访问。

这个隐式规范的实际后果是：如果我们修改了被枚举的对象，则不能保证 for...in 循环的行为是可预见的。

让我们进行另一种遍历图的尝试。这次自己管理循环控制。当我们使用循环时，应该使用自己的字典抽象以避免原型污染。我们可以将字典放置在 WorkSet 类中来追踪当前集合中的元素数量。

```
function WorkSet() {
    this.entries = new Dict();
    this.count = 0;
}

WorkSet.prototype.isEmpty = function() {
    return this.count === 0;
};

WorkSet.prototype.add = function(key, val) {
    if (this.entries.has(key)) {
        return;
    }
    this.entries.set(key, val);
    this.count++;
};

WorkSet.prototype.get = function(key) {
    return this.entries.get(key);
};

WorkSet.prototype.remove = function(key) {
    if (!this.entries.has(key)) {
        return;
    }
    this.entries.remove(key);
    this.count--;
};
```

为了提取集合的任意一个元素，我们需要给 Dict 类添加一个新方法。

```
Dict.prototype.pick = function() {
    for (var key in this.elements) {
        if (this.has(key)) {
            return key;
        }
    }
    throw new Error("empty dictionary");
};

WorkSet.prototype.pick = function() {
    return this.entries.pick();
};
```

现在我们可以使用简单的 while 循环来实现 inNetwork 方法。每次选择任意一个元素并从工作集中删除。

```
Member.prototype.inNetwork = function(other) {
    var visited = {};
    var workset = new WorkSet();
    workset.add(this.name, this);
    while (!workset.isEmpty()) {
        var name = workset.pick();
        var member = workset.get(name);
        workset.remove(name);
        if (name in visited) { // don't revisit members
            continue;
        }
        visited[name] = member;
        if (member === other) { // found?
            return true;
        }
        member.friends.forEach(function(friend) {
            workset.add(friend.name, friend);
        });
    }
    return false;
};
```

pick 方法是一个不确定性的例子。不确定性指的是一个操作并不能保证使用语言的语义产生一个单一的可预见的结果。这个不确定性来源于这样一个事实：for...in 循环可能在不同的 JavaScript 环境中选择不同的枚举顺序（甚至在同一个 JavaScript 环境中执行也不相同，至少原则上存在这样的情况）。使用不确定性可能会比较棘手，因为它给你的程序引入了一个不可预测的元素。测试可能会在某个平台上通过，但是在其他平台上失败，甚至在同一平台也会出现间歇性的失败。

一些不确定性的来源是难以避免的。随机数发生器应该产生不可预测的结果；检查当前的日期和时间总是得到不同的答案；响应用户操作（如单击鼠标或敲击键盘）的行为必然不同，因为这取决于用户。但是应该弄清楚程序的哪部分有一个预期的结果，哪部分的结果可能有所不同。

基于这些原因，考虑使用一个确定的工作集算法替代方案是值得的，即工作列表（worklist）算法。将工作条目存储到数组中而不是集合中，则 inNetwork 方法将总是以完全相同的顺序遍历图。

```
Member.prototype.inNetwork = function(other) {
    var visited = {};
    var worklist = [this];

    while (worklist.length > 0) {
        var member = worklist.pop();
        if (member.name in visited) {    // don't revisit
            continue;
        }
        visited[member.name] = member;
        if (member === other) {          // found?
            return true;
        }
        member.friends.forEach(function(friend) {
            worklist.push(friend);       // add to work-list
        });
    }
    return false;
};
```

该版本的 inNetwork 方法会确定性地添加和删除工作条目。由于无论发现什么路径，该方法对于连接的成员总是返回 true，所以最终结果是一样的。但是对于其他的方法可能并不适用，在编写程序时要小心对待，例如 inNetwork 中的改动，产生成员之间在图中的真实路径。

- 当使用 for...in 循环枚举一个对象的属性时，确保不要修改该对象。
- 当迭代一个对象时，如果该对象的内容可能会在循环期间被改变，应该使用 while 循环或经典的 for 循环来代替 for...in 循环。
- 为了在不断变化的数据结构中能够预测枚举，考虑使用一个有序的数据结构，例如数组，而不要使用字典对象。

第 49 条：数组迭代要优先使用 for 循环而不是 for...in 循环

下面这段代码中 mean 的输出值是多少？

```
var scores = [98, 74, 85, 77, 93, 100, 89];
var total = 0;
for (var score in scores) {
    total += score;
}
var mean = total / scores.length;
mean; // ?
```

你有没有发现 Bug？如果你认为答案为 88，那么你理解了这段程序的意图，但这并不是实际的结果。程序混淆了数字数组 key 和 value 这是极易犯下的错误。for...in 循环始终枚举所有 key。下一个看似合理的猜测是（0 + 1 + … + 6）/ 7 = 21，但是这也不正确。请记住即使是数组的索引属性，对象属性 key 始终是字符串。因此，“ +=”操作符将执行字符串的连接操作，结果便得到一个意想不到的 total 值“00123456”。那最终结果到底是什么？一个难以置信的 mean 值 17636.571428571428。

迭代数组内容的正确方法是使用传统的 for 循环。

```
var scores = [98, 74, 85, 77, 93, 100, 89];
var total = 0;
for (var i = 0, n = scores.length; i < n; i++) {
    total += scores[i];
}
var mean = total / scores.length;
mean; // 88
```

该方法确保当你需要整数索引和数组元素值时就能获取它们，并且绝不会混淆它们或引发意想不到的字符串强制转换。此外，它还确保以正确的顺序迭代数组，并且不会意外地包括存储在数组对象或其原型链中的非整数属性。

请注意上面 for 循环中数组长度变量 n 的使用。如果循环体不修改该数组，那么在每次迭代中，循环都会简单地重新计算数组的长度。

```
for (var i = 0; i < scores.length; i++) { ... }
```

另外，在循环的一开始就计算出数组的长度还有几个小的好处。首先，即使是优化的 JavaScript 编译器可能有时也很难保证避免重新计算 scores.length 是安全的。不过更重要的是，它给阅读该代码的程序员传递了一个信息：循环的终止条件是简单且确定的。

提示

- 迭代数组的索引属性应当总是使用 for 循环而不是 for...in 循环。

❑ 考虑在循环之前将数组的长度存储在一个局部变量中以避免重新计算数组长度。

第 50 条：迭代方法优于循环

优秀的程序员讨厌编写重复的代码。复制和粘贴样板代码会重复错误，使程序很难更改，将程序陷入重复的模式，并让程序员无休止地重复发明轮子。也许最糟糕的是，重复的代码使人阅读代码时太容易忽略一个模式实例与另一个的细微差别。

与许多其他的语言（如 C、Java 和 C#）相似，JavaScript 的 for 循环相当简洁。然而其 for 循环还允许细微的语法变化引入不同的行为。编程中的一些最臭名昭著的错误都是在确定循环的终止条件时引入的一些简单错误。

```
for (var i = 0; i <= n; i++) { ... }
// extra end iteration
for (var i = 1; i < n; i++) { ... }
// missing first iteration
for (var i = n; i >= 0; i--) { ... }
// extra start iteration
for (var i = n - 1; i > 0; i--) { ... }
// missing last iteration
```

让我们来面对这样一个现实：搞清楚终止条件是一个累赘。它非常无聊，而且有太多搞乱它的方式。

值得庆幸的是，JavaScript 的闭包（请参阅第 11 条）是一种为这些模式建立迭代抽象方便的、富有表现力的手法，从而使我们避免复制、粘贴循环头部。

ES5 为最常用的一些模式提供了便利的方法。Array.prototype.forEach 是其中最简单的一个。例如：

```
for (var i = 0, n = players.length; i < n; i++) {
    players[i].score++;
}
```

我们可以用以下代码替换上面的循环。

```
players.forEach(function(p) {
    p.score++;
});
```

这段代码不仅更简单可读，而且还消除了终止条件和任何数组索引。

另一种常见的模式是对数组的每个元素进行一些操作后建立一个新的数组。我们可以使用循环来实现。

```
var trimmed = [];
for (var i = 0, n = input.length; i < n; i++) {
    trimmed.push(input[i].trim());
}
```

或者，我们也可以使用 forEach 方法来实现。

```
var trimmed = [];
input.forEach(function(s) {
    trimmed.push(s.trim());
});
```

但是通过现有的数组建立一个新的数组的模式是如此的普遍，所以 ES5 引入了 Array.prototype. map 方法使该模式更简单、更优雅。

```
var trimmed = input.map(function(s) {
    return s.trim();
});
```

另一种常见的模式是计算一个新的数组，该数组只包含现有数组的一些元素。

Array.prototype.filter 使其变得很简便。它需要一个谓词。该谓词是一个函数，如果元素应该存在于新数组中则返回真值，如果元素应该被剔除则返回假值。例如，我们可以从价格表中提取出符合一个特定价格区间的列表。

```
listings.filter(function(listing) {
    return listing.price >= min && listing.price <= max;
});
```

当然，这些都只是 ES5 中的默认方法。我们完全可以定义自己的迭代抽象。例如，有时需要这样一个模式，即提取出满足谓词的数组的前几个元素。

```
function takeWhile(a, pred) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i++) {
        if (!pred(a[i], i)) {
            break;
        }
        result[i] = a[i];
    }
    return result;
}

var prefix = takeWhile([1, 2, 4, 8, 16, 32], function(n) {
    return n < 10;
}); // [1, 2, 4, 8]
```

请注意我们将数组索引 i 赋给了 pred，我们可以选择使用或忽略该索引。事实上，标准

库中的所有迭代方法（包括 forEach、map 和 filter）都将数组索引传递给了用户自定义的函数。

我们也可以将 takeWhile 函数添加到 Array.prototype 中使其作为一个方法（请参阅第 42 条关于对类似 Array.prototype 的标准原型添加猴子补丁的影响的讨论）。

```
Array.prototype.takeWhile = function(pred) {
    var result = [];
    for (var i = 0, n = this.length; i < n; i++) {
        if (!pred(this[i], i)) {
            break;
        }

        result[i] = this[i];
    }
    return result;
};

var prefix = [1, 2, 4, 8, 16, 32].takeWhile(function(n) {
    return n < 10;
}); // [1, 2, 4, 8]
```

循环只有一点优于迭代函数，那就是前者有控制流操作，如 break 和 continue。举例来说，使用 forEach 方法来实现 takeWhile 函数将是一个尴尬的尝试。

```
function takeWhile(a, pred) {
    var result = [];
    a.forEach(function(x, i) {
        if (!pred(x)) {
            // ?
        }
        result[i] = x;
    });
    return result;
}
```

我们可以使用一个内部异常来提前终止该循环，但是这既尴尬又效率低下。

```
function takeWhile(a, pred) {
    var result = [];
    var earlyExit = {}; // unique value signaling loop break
    try {
        a.forEach(function(x, i) {
            if (!pred(x)) {
                throw earlyExit;
            }
            result[i] = x;
        });
    } catch (e) {
```

```
            if (e !== earlyExit) { // only catch earlyExit
                throw e;
            }
        }
        return result;
    }
```

一旦一个抽象的概念比它替代的代码更罗嗦，就可视为弄巧成拙。

此外，ES5 的数组方法 some 和 every 可以用于提前终止循环。可以说，创建这些方法并不是为此目的。我们将这些方法称为谓词，重复地对数组的每个元素应用回调的谓词。具体来说，some 方法返回一个布尔值表示其回调对数组的任何一个元素是否返回了一个真值。

```
[1, 10, 100].some(function(x) { return x > 5; }); // true
[1, 10, 100].some(function(x) { return x < 0; }); // false
```

类似的，every 方法返回一个布尔值表示其回调是否对所有元素返回了一个真值。

```
[1, 2, 3, 4, 5].every(function(x) { return x > 0; }); // true
[1, 2, 3, 4, 5].every(function(x) { return x < 3; }); // false
```

这两个方法都是短路循环（short-circuiting）。如果对 some 方法的回调一旦产生了一个真值，则 some 方法会直接返回，不会执行其余的元素。相似的，every 方法的回调一旦产生了假值，则会立即返回。

这种行为使得这些方法在实现 forEach 提前终止循环的变种时派上了用场。例如，可以使用 every 实现 takeWhile 函数。

```
function takeWhile(a, pred) {
    var result = [];
    a.every(function(x, i) {
        if (!pred(x)) {
            return false; // break
        }
        result[i] = x;
        return true; // continue
    });
    return result;
}
```

提示

- 使用迭代方法（如 Array.prototype.forEach 和 Array.prototype.map）替换 for 循环使得代码更可读，并且避免了重复循环控制逻辑。
- 使用自定义的迭代函数来抽象未被标准库支持的常见循环模式。
- 在需要提前终止循环的情况下，仍然推荐使用传统的循环。另外，some 和 every 方法也可用于提前退出。

第 51 条：在类数组对象上复用通用的数组方法

Array.prototype 中的标准方法被设计成其他对象可复用的方法，即使这些对象并没有继承 Array。事实证明，许多这样的类数组对象接踵而至地出现在 JavaScript 的不同地方。

一个很好的例子是第 22 条所描述的函数的 arguments 对象。不幸的是，arguments 对象没有继承 Array.prototype，因此我们不能简单地通过调用 arguments.forEach 方法来遍历每一个参数。取而代之的是，我们不得不提取出 forEach 方法对象的引用并使用其 call 方法（请参阅第 20 条）。

```
function highlight() {
    [].forEach.call(arguments, function(widget) {
        widget.setBackground("yellow");
    });
}
```

forEach 方法是一个 Function 对象。这意味着它继承了 Function.prototype 中的 call 方法。这使得我们可以使用一个自定义的值作为 forEach 方法内部的 this 绑定来调用它（在这个例子中，也就是 arguments 对象），并紧随任意数量的参数（在这个例子中，也就是那个回调函数）。总而言之，这段代码的行为正是我们想要的。

在 Web 平台，DOM（Document Object Model）的 NodeList 类是另一个类数组对象的实例。类似 document.getElementsByTagName 这样的操作会查询 Web 页面中的节点，并返回 NodeList 作为搜索的结果。像 arguments 对象一样，NodeList 的行为类似数组也没有继承 Array.prototype。

那到底怎样使得一个对象“看起来像数组”呢？数组对象的基本契约总共有两个简单的规则。

- 具有一个范围在 0 到 $2^{32}-1$ 的整型 length 属性。
- length 属性大于该对象的最大索引。索引是一个范围在 0 到 $2^{32}-2$ 的整数，它的字符串表示的是该对象中的一个 key。

这就是一个对象需要实现的与 Array.prototype 中任一方法兼容的所有行为。甚至是一个简单的对象字面量也可以用来创建一个类数组对象。

```
var arrayLike = { 0: "a", 1: "b", 2: "c", length: 3 };
var result = Array.prototype.map.call(arrayLike, function(s) {
    return s.toUpperCase();
}); // ["A", "B", "C"]
```

字符串也表现为不可变的数组，因为它们是可索引的，并且其长度也可以通过 length 属性获取。因此，Array.prototype 中的方法操作字符串时并不会修改原始数组。

```
var result = Array.prototype.map.call("abc", function(s) {
    return s.toUpperCase();
}); // ["A", "B", "C"]
```

目前，模拟 JavaScript 数组的所有行为很精妙。这要归功于数组行为的两个方面。

- 将 length 属性值设为小于 n 的值会自动地删除索引值大于或等于 n 的所有属性。
- 增加一个索引值为 n（大于或等于 length 属性值）的属性会自动地设置 length 属性为 n+1。

第二条规则尤其难以完成，因为它需要监控索引属性的增加以自动地更新 length 属性。幸运的是，对于使用 Array.prototype 中的方法，这两条规则都不是必须的，因为在增加或删除索引属性的时候它们都会强制地更新 length 属性。

只有一个 Array 方法不是完全通用的，即数组连接方法 concat。该方法可以由任意的类数组接收者调用，但它会检查其参数的 [[Class]] 属性。如果参数是一个真实的数组，那么 concat 会将该数组的内容连接起来作为结果；否则，参数将以一个单一的元素来连接。这意味着，例如，我们就不能简单地连接一个以 arguments 对象作为内容的数组。

```
function namesColumn() {
    return ["Names"].concat(arguments);
}
namesColumn("Alice", "Bob", "Chris");
// ["Names", { 0: "Alice", 1: "Bob", 2: "Chris" }]
```

为了使 concat 方法将一个类数组对象视为真实的数组，我们不得不自己转换该数组。实现该转换的一个流行而且简洁的惯用法是在类数组对象上调用 slice 方法。

```
function namesColumn() {
    return ["Names"].concat([].slice.call(arguments));
}
namesColumn("Alice", "Bob", "Chris");
// ["Names", "Alice", "Bob", "Chris"]
```


提示

- 对于类数组对象，通过提取方法对象并使用其 call 方法来复用通用的 Array 方法。
- 任意一个具有索引属性和恰当 length 属性的对象都可以使用通用的 Array 方法。

第 52 条：数组字面量优于数组构造函数

JavaScript 的优雅很大程度上要归功于其程序中最常见的构造块（对象、函数及数组）的简明的字面量语法。字面量是一种表示数组的优雅的方法。

```
var a = [1, 2, 3, 4, 5];
```

你也可以使用数组构造函数来替代。

```
var a = new Array(1, 2, 3, 4, 5);
```

但是事实证明，即使撇开美学不谈，Array 构造函数都存在一些微妙的问题。首先，你必须要确保，没有人重新包装过 Array 变量。

```
function f(Array) {
    return new Array(1, 2, 3, 4, 5);
}
f(String); // new String(1)
```

你还必须确保没有人修改过全局的 Array 变量。

```
Array = String;
new Array(1, 2, 3, 4, 5); // new String(1)
```

而且，你还要担心一种特殊的情况。如果使用单个数字参数来调用 Array 构造函数，效果完全不同。它试图创建一个没有元素的数组，但其长度属性为给定的参数。这意味着 [“hello”] 和 new Array（“hello”）的行为相同，但 [17] 和 new Array（17）的行为完全不同！

这些规则学起来并不困难，然而使用数组字面量更清晰，而且不易导致异常错误，因为数组字面量具有更规范、更一致的语义。

提示

- 如果数组构造函数的第一个参数是数字则数组的构造函数行为是不同的。
- 使用数组字面量替代数组构造函数。

第 6 章
库和 API 设计

从某种意义上讲每个程序员都是 API 设计者。也许你并未计划立即写下一个流行的 JavaScript 库，但当你的程序已经在一个平台上运行足够长的时间，你对常见的问题建立了完善的解决方案，迟早你会开始开发可重用的程序和组件。即使你没有将其发布为单独的库，提升自己的技能成 为一个库作者能让你编写更好的组件。

设计程序库是一个棘手的业务，更多地表现为科学艺术，这是非常重要的。API 是程序员的基本词汇。设计良好的 API 能让你的用户（这可能包括你自己）清楚、简洁和明确地表达自己的程序。

第 53 条：保持一致的约定

对于 API 使用者来说，你所使用的命名和函数签名是最能产生普遍影响的决策。这些约定具有巨大的影响力。它们建立了基本的词汇和使用它们的应用程序的惯用法。库的使用者必须学会阅读和使用这些惯用法。因此，你的工作应当使得学习曲线尽可能的简单。不一致的约定会使得很难记住哪些约定应该应用在哪些场景，从而导致较多的时间花费在查阅库的文档上，而只有较少的时间来完成实际的工作。

其中一个关键的约定是参数的顺序。例如，用户界面库通常具有一些接收多个测量值（如宽度、高度）的函数。多为你的用户考虑，确保这些参数总是以相同的顺序出现。选择与其他库匹配的顺序是值得的，因为几乎所有的库接收的顺序都是宽度第一，然后接收高度。

```
var widget = new Widget(320, 240); // width: 320, height: 240
```

除非你有一个很强烈的需要改变普遍做法的理由，否则请你遵守这些熟悉的约定。如果你编写的库是为 Web 量身打造的，请记住 Web 开发人员通常会处理多门语言（至少包括 HTML、CSS 和 JavaScript）。不要引入一些不必要的约定变化（这些约定很可能是他们日常

工作中使用的），这会使得他们的工作变得困难。例如，无论什么时候，CSS 接收描述矩形的四条边的参数时，总是要求参数以从 top 开始的顺时针顺序给定（top, right, bottom, left）。因此，在设计库的类似的 API 时请保持这一顺序。这样，你的用户会感谢你的。或者，也许他们不会察觉到，那就更好了。但可以肯定，如果你违背了标准的约定，他们一定会注意到。

如果你的 API 使用选项对象（请参阅第 55 条），你可以避免参数顺序的依赖。对于标准选项，如宽度或高度测量值，你应该挑选一个命名约定并虔诚地坚持它。如果其中的一个函数签名查找 width 和 height 选项而其他的查找 w 和 h 选项，那么你的用户一辈子都会检查文档以记住哪个应该用在哪里。同样地，如果 Widget 类具有设置属性的方法，请确保这些更新方法使用同一套命名约定。一个类使用 setWidth 方法设置宽度，而另一个类使用 width 方法做同样的事，这并不存在一个很好的理由。

每一个优秀的库都需要详尽的文档，而一个极优秀的库会将其文档作为辅助。一旦你的用户习惯了库中的约定，他们就可以做一些常见的任务而不需要每次去查看文档。一致的约定甚至能帮助你的用户推测出哪些属性或方法是可用的而根本不需要去查看它们，或者在控制台里发现它们，进而可以根据其命名推测出它们的行为。

- 在变量命名和函数签名中使用一致的约定。
- 不要偏离用户在他们的开发平台中很可能遇到的约定。

第 54 条：将 undefined 看做“没有值”

undefined 值很特殊，每当 JavaScript 无法提供具体的值时，就产生 undefined。未赋值的变量的初始值即为 undefined。

```
var x;
x; // undefined
```

访问对象中不存在的属性也会产生 undefined。

```
var obj = {};
obj.x; // undefined
```

一个函数体结尾使用未带参数的 return 语句，或未使用 return 语句都会产生返回值 undefined。

```
function f() {
    return;
}

function g() { }
```

```
f(); // undefined
g(); // undefined
```

未给函数参数提供实参则该函数参数值为 undefined。

```
function f(x) {
    return x;
}

f(); // undefined
```

在以上这些情况中，undefined 值表明操作结果并不是一个特定的值。当然，有一种关于值的有点自相矛盾的值叫“没有值”(no value)。但是每个操作都要产出点儿什么，所以可以说 JavaScript 使用 undefined 来填补这个空白。

将 undefined 看做缺少某个特定的值是 JavaScript 语言建立的一种公约。将它用于其他目的具有很高的风险。例如，一个用户界面元素库可能支持一个 highlight 方法用于改变一个元素的背景颜色。

```
element.highlight();          // use the default color
element.highlight("yellow");  // use a custom color
```

如果我们想提供一种方式来设置一个随机颜色，可能会使用 undefined 作为特殊的值来实现。

```
element.highlight(undefined); // use a random color
```

但这会对 undefined 的通常含义产生歧义。这会使得从其他来源获取参数时更容易导致错误的行为，特别是在没有提供值时。例如，程序可能使用一个可选的颜色偏好的配置对象。

```
var config = JSON.parse(preferences);
// ...
element.highlight(config.highlightColor); // may be random
```

如果该偏好设置未指定一个颜色，程序员最有可能期望得到默认值，就像没提供值一样。但是由于重利用了 undefined，实际上造成这种代码产生了一个随机颜色。更好的 API 设计可能会使用一种特殊的颜色名来实现随机颜色。

```
element.highlight("random");
```

有时一个 API 不能够从通常函数可接受的字符串集合中区分出一个特殊的字符串值。在这种情况下，可以使用除 undefined 以外的其他特殊值，如 null 或 true。但是这往往导致代码的可读性下降。

```
element.highlight(null);
```

那些阅读代码的人可能并没有记住你的代码库，这段代码是非常难以理解的。事实上，

起初的猜测可能是该方法取消了高亮。一个更明确、更具描述性的可选做法是将随机情况表示为一个具有 random 属性的对象（请参阅第 55 条关于选项对象的更多信息）。

```
element.highlight({ random: true });
```

另一个需要提防 undefined 的地方是可选参数的实现。理论上 arguments 对象（请参阅第 51 条）可检测是否传递了一个参数，但实际上，测试是否为 undefined 会打造出更为健壮的 API。例如，一个 Web 服务器可以接收一个可选的主机名称。

```
var s1 = new Server(80, "example.com");
var s2 = new Server(80); // defaults to "localhost"
```

可以通过判断 arguments.length 来实现 Server 构造函数。

```
function Server(port, hostname) {
    if (arguments.length < 2) {
        hostname = "localhost";
    }

    hostname = String(hostname);
    // ...
}
```

但这种实现与上述的 element.highlight 方法有个相似的问题。如果程序通过从另一个源（如配置对象）请求一个值来提供了一个显式的参数，那么可能会产生 undefined。

```
var s3 = new Server(80, config.hostname);
```

如果 config 中并未设置 hostname，正常行为是使用默认值“localhost”。但上述实现结果主机名是 undefined。最好测试 undefined，因为不传递该参数会产生 undefined，或者一个参数表达式的结果是 undefined。

```
function Server(port, hostname) {
    if (hostname === undefined) {
        hostname = "localhost";
    }
    hostname = String(hostname);
    // ...
}
```

另一种合理的替代方案是测试 hostname 是否为真（请参阅第 3 条）。使用逻辑运算符很容易实现。

```
function Server(port, hostname) {
    hostname = String(hostname || "localhost");
    // ...
}
```

此版本使用了逻辑运算符或（||）。当第一个参数为真值则会返回第一个参数，否则返回第二个参数值。所以，如果 hostname 值为 undefined 或空字符串，该表达式 (hostname || "localhost") 的求值结果为 "localhost"。因此，技术上来说测试了比 undefined 更多的值。它将所有为假值的值与 undefined 同等对待。对于 Server 是可以接受的，因为空字符串并不是有效的主机名。所以，如果你喜欢更宽松的 API，那么可以强制所有假值为默认值，真值测试是实现参数默认值的一种简明的方式。

但要注意，真值测试并不总是安全的。如果一个函数应该接收空字符串为合法值，真值测试将覆盖空字符串并替换为默认值。类似的，接收数字为参数的函数如果允许 0 为可接受的参数（或 NaN, 虽然这不太常见），则不应该使用真值测试。例如，一个用于创建用户界面元素的函数可能允许一个元素的宽度或高度为 0，但提供的默认值却不一样。

```
var c1 = new Element(0, 0); // width: 0, height: 0
var c2 = new Element();     // width: 320, height: 240
```

使用真值测试的实现会有问题。

```
function Element(width, height) {
    this.width = width || 320;   // wrong test
    this.height = height || 240; // wrong test
    // ...
}

var c1 = new Element(0, 0);

c1.width;  // 320
c1.height; // 240
```

相反，我们必须求助于更详细的测试来测试 undefined。

```
function Element(width, height) {
    this.width = width === undefined ? 320 : width;
    this.height = height === undefined ? 240 : height;
    // ...
}

var c1 = new Element(0, 0);

c1.width;  // 0
c1.height; // 0

var c2 = new Element();

c2.width;  // 320
c2.height; // 240
```

- 避免使用 undefined 表示任何非特定值。
- 使用描述性的字符串值或命名布尔属性的对象，而不要使用 undefined 或 null 来代表特定应用标志。
- 提供参数默认值应当采用测试 undefined 的方式，而不是检查 arguments.length。
- 在允许 0、NaN 或空字符串为有效参数的地方，绝不要通过真值测试来实现参数默认值。

第 55 条：接收关键字参数的选项对象

正如第 53 条建议的一样，保持参数顺序的一致约定对于帮助程序员记住每个参数在函数调用中的意义是很重要的。参数较少时它是适用的。但参数过多后，它根本不可扩展。尝试理解下面的函数调用。

```
var alert = new Alert(100, 75, 300, 200,
                      "Error", message,
                      "blue", "white", "black",
                      "error", true);
```

我们已经见过许多与之类似的 API。这通常是参数蔓延（argument creep）的结果。一个函数起初很简单，然而一段时间后，随着库功能的扩展，该函数的签名便会获得越来越多的参数。

幸运的是，JavaScript 提供了一个简单、轻量的惯用法：选项对象（options object）。选项对象在应对较大规模的函数签名时运作良好。一个选项参数就是一个通过其命名属性来提供额外参数数据的参数。对象字面量的形式使得读写选项对象尤其舒适。

```
var alert = new Alert({
    x: 100, y: 75,
    width: 300, height: 200,
    title: "Error", message: message,
    titleColor: "blue", bgColor: "white", textColor: "black",
    icon: "error", modal: true
});
```

该 API 有点烦琐，但明显更易于阅读。每个参数都是自我描述的（self-documenting）。不需要注释来解释参数的职责，因为其属性名清楚地解释了它。这对布尔值参数如 modal 更有帮助。一些人在读到 new Alert 调用时很可能会根据字符串参数的内容推断其目的，但是对于裸露的 true 或 false 来说并不能提供特别的信息。

选项对象的另一个好处是所有的参数都是可选的，调用者可以提供任一可选参数的子

集。与普通参数（有时也称为位置参数，因为它们不是由其命名而是由它们在参数列表中的位置区分的）相比，可选参数通常会引入一些歧义。例如，如果我们希望 Alert 对象的位置和大小属性都是可选的，那么很难解释清楚如下的调用。

```
var alert = new Alert(app,
                      150, 150,
                      "Error", message,
                      "blue", "white", "black",
                      "error", true);
```

最开始的两个数字是指定 x 和 y 参数还是 width 和 height 参数呢？而使用选项对象就没有任何问题。

```
var alert = new Alert({
    parent: app,
    width: 150, height: 100,
    title: "Error", message: message,
    titleColor: "blue", bgColor: "white", textColor: "black",
    icon: "error", modal: true
});
```

习惯上，选项对象仅包括可选参数，因此省略掉整个对象甚至都是可能的。

```
var alert = new Alert(); // use all default parameter values
```

如果有一个或者两个必选参数，最好使它们独立于选项对象。

```
var alert = new Alert(app, message, {
    width: 150, height: 100,
    title: "Error",
    titleColor: "blue", bgColor: "white", textColor: "black",
    icon: "error", modal: true
});
```

实现一个接收选项对象的函数需要做更多的工作。下面是一个详细的实现。

```
function Alert(parent, message, opts) {
    opts = opts || {}; // default to an empty options object
    this.width = opts.width === undefined ? 320 : opts.width;
    this.height = opts.height === undefined
                ? 240
                : opts.height;
    this.x = opts.x === undefined
           ? (parent.width / 2) - (this.width / 2)
           : opts.x;

    this.y = opts.y === undefined
           ? (parent.height / 2) - (this.height / 2)
           : opts.y;
```

```
    this.title = opts.title || "Alert";
    this.titleColor = opts.titleColor || "gray";
    this.bgColor = opts.bgColor || "white";
    this.textColor = opts.textColor || "black";
    this.icon = opts.icon || "info";
    this.modal = !!opts.modal;
    this.message = message;
}
```

实现从使用或（||）操作符（请参阅第 54 条）提供一个默认空选项对象开始。由于 0 是一个有效值但不是默认值，所以正如第 54 条建议的一样，我们需要测试数值参数是否为 undefined。基于空字符串是无效的、应该被默认值取代的假设，我们使用逻辑或来应对字符串参数。modal 参数使用双重否定模式将其参数强制转换为一个布尔值。

比起使用位置参数的方式，这段代码可能有点烦琐。目前，如果库能简化我们的工作，那么付出点代价都是值得的。但我们可以使用有用的抽象（对象扩展或合并函数）来简化我们的工作。许多 JavaScript 库和框架都提供一个 extend 函数。该函数接收一个 target 对象和一个 source 对象，并将后者的属性复制到前者中。该实用程序最有用的应用之一是抽象出了合并默认值和用户提供的选项对象值的逻辑。借助于 extend 函数，Alert 函数变得简洁许多。

```
function Alert(parent, message, opts) {
    opts = extend({
        width: 320,
        height: 240
    });
    opts = extend({
        x: (parent.width / 2) - (opts.width / 2),
        y: (parent.height / 2) - (opts.height / 2),
        title: "Alert",
        titleColor: "gray",
        bgColor: "white",
        textColor: "black",
        icon: "info",
        modal: false
    }, opts);

    this.width = opts.width;
    this.height = opts.height;
    this.x = opts.x;
    this.y = opts.y;
    this.title = opts.title;
    this.titleColor = opts.titleColor;
    this.bgColor = opts.bgColor;
    this.textColor = opts.textColor;
```

```
        this.icon = opts.icon;
        this.modal = opts.modal;
    }
```

这避免了不断地重新实现检查每个参数是否存在的逻辑。请注意我们调用了两次 extend 函数，因为 x 和 y 的默认值依赖于早前计算出的 width 和 height 值。

如果我们想要做的是将整个 options 复制到 this 对象，那么我们可以进一步简化它。

```
function Alert(parent, message, opts) {
    opts = extend({
        width: 320,
        height: 240
    });
    opts = extend({
        x: (parent.width / 2) - (opts.width / 2),
        y: (parent.height / 2) - (opts.height / 2),
        title: "Alert",
        titleColor: "gray",
        bgColor: "white",
        textColor: "black",
        icon: "info",
        modal: false
    }, opts);
    extend(this, opts);
}
```

不同的框架提供了不同的 extend 函数变种，然而典型的实现是枚举源对象的属性，并当这些属性不是 undefined 时将其复制到目标对象中。

```
function extend(target, source) {
    if (source) {
        for (var key in source) {
            var val = source[key];

            if (typeof val !== "undefined") {
                target[key] = val;
            }
        }
    }
    return target;
}
```

请注意原来的 Alert 版本和使用 extend 函数实现的版本有些细微的不同。首先，早前版本中的条件逻辑如果不需要默认值则会避免计算默认值。只要计算默认值对诸如修改用户接口或发送网络请求没有影响，那么这根本不是一个问题。另一个区别是判断一个值是否已经提供了的逻辑。在早前版本中，对于字符串参数，我们将空字符串视为与 undefined 等价。

然而，只是将 undefined 视为缺省的参数更为恰当。使用或（||）操作符是一种提供默认参数值有效但非一致的策略。一致性是库设计的一个良好目标，因为它会给 API 的使用者带来更好的可预测性。


提示

- 使用选项对象使得 API 更具可读性、更容易记忆。
- 所有通过选项对象提供的参数应当被视为可选的。
- 使用 extend 函数抽象出从选项对象中提取值的逻辑。

第 56 条：避免不必要的状态

API 有时被归为两类：有状态的和无状态的。无状态的 API 提供的函数或方法的行为只取决于输入，而与程序的状态改变无关。字符串的方法是无状态的。字符串的内容不能被修改，方法只取决于字符串的内容及传递给方法的参数。不管程序其他部分的情况，表达式 "foo".toUpperCase() 总是产生 "FOO"。相反，Date 对象的方法却是有状态的。对相同的 Date 对象调用 toString 方法会产生不同的结果，这取决于 Date 的各种 set 方法是否已经将 Date 的属性修改。

虽然状态有时是必需的，无状态的 API 往往更容易学习和使用，更自我描述，且不易出错。一个著名的有状态的 API 是 Web 的 Canvas 库。它提供了绘制形状和图片到其平面的用户界面元素方法。一个程序可以使用 fillText 方法绘制文本到一个画布。

```
c.fillText("hello, world!", 75, 25);
```

该方法提供了一个用于绘制的字符串和画布中的位置。但其并未制定被绘制文本的其他属性，例如颜色、透明度或文本样式。所有的这些属性通过改变画布的内部状态来单独指定。

```
c.fillStyle = "blue";
c.font = "24pt serif";
c.textAlign = "center";
c.fillText("hello, world!", 75, 25);
```

该 API 的无状态版本可能如下：

```
c.fillText("hello, world!", 75, 25, {
    fillStyle: "blue",
    font: "24pt serif",
    textAlign: "center"
});
```

为什么后者更可取呢？首先，它并不脆弱。为了实现定制化，有状态的 API 需要修改画布的内部状态，这将导致绘制操作之间互相影响，即使它们之间并没有关联。例如，默认的填充样式是黑色。你指望得到默认值需要建立在只有你知道没人已经改变了默认值的前提下。如果你想在默认值被修改之后使用默认值进行绘制操作，你必须显式指定默认值。

```
c.fillText("text 1", 0, 0); // default color
c.fillStyle = "blue";
c.fillText("text 2", 0, 30); // blue
c.fillStyle = "black";
c.fillText("text 3", 0, 60); // back in black
```

相比于有状态的 API，无状态的 API 会自动重用默认值。

```
c.fillText("text 1", 0, 0); // default color
c.fillText("text 2", 0, 30, { fillStyle: "blue" }); // blue
c.fillText("text 3", 0, 60); // default color
```

还要注意每个语句如何变得更可读。要了解任何单独的调用 fillText 方法，你不必了解它前面的所有修改。事实上，画布可能已经在程序的一些完全独立的部分被修改。其他地方的一块代码修改了画布的状态，这很容易导致错误。

```
c.fillStyle = "blue";
drawMyImage(c); // did drawMyImage change c?
c.fillText("hello, world!", 75, 25);
```

为了理解最后一行发生了什么，我们需要知道 drawMyImage 方法可能对画布做了什么修改。无状态的 API 会使代码更加模块化，从而避免代码不同部分相互影响，同时也使代码更易于阅读。

有状态的 API 也更难学习。阅读 fillText 的文档，你很难说出画布中哪些方面的状态会影响绘制。即使其中一些很容易被猜到，但对于非专业的人很难知道是否已经正确地初始化了所有必要的状态。这当然可以在 fillText 的文档中提供一个详尽的清单。当需要一个有状态的 API 时，你需要小心地记录这些状态依赖项。但无状态的 API 完全消除了这些隐式依赖，所以并不需要在最开始提供额外的文档。

无状态的 API 的另一个好处是简洁。有状态的 API 往往会导致额外的声明，仅仅在调用方法前设置对象的内部状态。考虑一个流行的“INI”配置文件格式的解析器。例如，一个简单的 INI 文件是这样的。

```
[Host]
address=172.0.0.1
name=localhost
[Connections]
timeout=10000
```

操作这种数据的 API 的一种方式是提供一个 setSection 方法在使用 get 方法查找配置参数之前选择一个区域。

```
var ini = INI.parse(src);

ini.setSection("Host");
var addr = ini.get("address");
var hostname = ini.get("name");

ini.setSection("Connection");
var timeout = ini.get("timeout");
var server = new Server(addr, hostname, timeout);
```

但对于无状态的 API，没必要创建额外的变量（例如 addr 和 hostname）在更新区域前保存提取的数据。

```
var ini = INI.parse(src);
var server = new Server(ini.Host.address,
                        ini.Host.name,
                        ini.Connection.timeout);
```

注意一旦显式地指定区域，可以简单地将 ini 对象表示为一个字典，每个区域作为一个字典使得 API 更简单。（请参阅第 5 章学习关于字典对象的更多知识。）

- 尽可能地使用无状态的 API。
- 如果 API 是有状态的，标示出每个操作与哪些状态有关联。

第 57 条：使用结构类型设计灵活的接口

想象一个创建 wiki 的库。wiki 网站包含用户可以交互式地创建、删除和修改的内容。许多 wiki 都以简单、基于文本标记语言创建内容为特色。通常，这些标记语言只提供了 HTML 可用功能的一个子集，但是却有一个更简单、更清晰的源格式。例如，环绕星号的文本被格式化为粗体，环绕下划线的被格式化为带有下划线的文本，环绕斜杠的被格式化为斜体。用户可以输入如下文本。

```
This sentence contains a *bold phrase* within it.
This sentence contains an _underlined phrase_ within it.
This sentence contains an /italicized phrase/ within it.
```

该网站会以下面的形式将内容呈现给 wiki 读者。

This sentence contains a **bold phrase** within it.

This sentence contains an underlined phrase within it.

This sentence contains an *italicized phrase* within it.

一个灵活的 wiki 库应该给应用程序编写者提供可供选择的标记语言，因为多年来已经涌现出了许多不同的流行文本格式。

要给应用程序编写者提供备选的标记语言，我们需要将提取用户创建的标记源文本内容的功能与其他功能分离。其他 wiki 功能包括账户管理、修订历史记录和内容存储等。其余的应用程序可以通过一套文档完善的属性和方法的接口与提取出的功能交互。通过针对接口完备 API 的严格编程，并忽略这些方法的实现细节，从而不论选择使用哪个源格式，应用程序都能正常运行。

让我们更深入地了解需要什么样的 wiki 内容提取接口。wiki 库必须能提取元数据，如页面标题和作者信息等，并能将页面内容格式化为 HTML 呈现给 wiki 读者。我们可以将 wiki 中的每一个页面表示为提供了通过 getTile、getAuthor 和 toHTML 页面方法获取这些数据的一个对象。

接下来，该 wiki 库需要提供创建一个自定义 wiki 格式化器的应用程序的方法，以及一些针对流行标记格式的内置格式化器。例如，应用程序的编写者可能希望使用 MediaWiki 格式化器（这是维基百科所使用的格式）。

```
var app = new Wiki(Wiki.formats.MEDIAWIKI);
```

该 wiki 库将该格式化函数存储在 wiki 实例对象的内部。

```
function Wiki(format) {
    this.format = format;
}
```

每当读者想要查看页面时，应用程序都会检索出其源文本并使用内部的格式化器将源文本渲染为 HTML 页面。

```
Wiki.prototype.displayPage = function(source) {
    var page = this.format(source);
    var title = page.getTitle();
    var author = page.getAuthor();
    var output = page.toHTML();
    // ...
};
```

类似 Wiki.formats.MEDIAWIKI 的格式化器是如何实现的呢？熟悉基于类编程的程序员可能倾向于创建一个 Page 的基类，该 Page 类表示用户创建的内容，每个 Page 的子类实现不同的格式。MediaWiki 格式化可能实现为一个继承 Page 的 MWPage 类，而 MEDIAWIKI 则

是一个返回 MWPage 实例的“工厂函数”。

```
function MWPage(source) {
    Page.call(this, source); // call the super-constructor
    // ...
}

// MWPage extends Page
MWPage.prototype = Object.create(Page.prototype);

MWPage.prototype.getTitle = /* ... */;
MWPage.prototype.getAuthor = /* ... */;
MWPage.prototype.toHTML = /* ... */;

Wiki.formats.MEDIAWIKI = function(source) {
    return new MWPage(source);
};
```

（更多关于使用构造函数和原型的实现类继承请参阅第 4 章。）但是 Page 基类的实用目的是什么？由于 MWPage 类需要自己实现所有 wiki 应用程序需要的 getTitle、getAuthor 和 toHTML 方法，因此未必可继承任何有用的实现代码。而且，上文提到的 displayPage 方法根本不关心页面对象的继承体系。它只需要实现如何显示页面的相关方法。所以，wiki 格式的实现很自由，任何喜欢的方式都可以。

尽管大多数面向对象语言都鼓励使用类和继承来结构化程序，然而 JavaScript 却不拘于形式。提供一个使用简单对象字面量构建 MediaWiki 页面格式的接口实现通常已经足够了。

```
Wiki.formats.MEDIAWIKI = function(source) {
    // extract contents from source
    // ...
    return {
        getTitle: function() { /* ... */ },
        getAuthor: function() { /* ... */ },
        toHTML: function() { /* ... */ }
    };
};
```

此外，继承有时会导致比它解决的更多的问题。当几个不同的 wiki 格式共享不相重叠的功能集时，继承的问题就显而易见了。也许没有任何继承结构才是对的。举例来说，想象以下 3 种格式。

Format A: *bold*, [Link], /italics/

Format B: **bold**, [[Link]], *italics*

Format C: **bold**, [Link], *italics*

我们想要实现各个部分的功能来识别每种不同类型的输入，然而功能的混合和匹配并没有映射到任何清晰的 A、B 和 C 之间的继承层次关系（欢迎你来试试！）。正确的做法是分别实现每种输入匹配的函数（单星号、双星号、斜杠、中括号等），然后根据每种格式的需要混合和匹配功能。

请注意消除 Page 基类，我们不需要用任何东西来替代它。这正是 JavaScript 动态类型的亮点。任何人希望实现一个新的自定义格式都可以这样做，而不需要在某处“注册”它。只要 displayPage 方法结构正确，具有预期行为的 getTile、getAuthor 和 getHTML 方法，那么它就适用于任何 JavaScript 对象。

这种接口有时称为结构类型（structural typing）或鸭子类型（duck typing）。任何对象只要具有预期的结构就属于该类型（如果它看起来像只鸭子、游泳像只鸭子或叫声像只鸭子）。在类似 JavaScript 这样的动态语言中，这是一种优雅、特别轻量的编程模式，因为它不需要编写任何显式的声明。一个调用某个对象方法的函数能够与任何实现了相同接口的对象一起工作。当然，你应当在 API 的文档中列出对象接口的预期结构。这样，接口实现者便会知道哪些属性和方法是必需的以及你的库或应用程序期望的行为是什么。

灵活的结构类型的另一个好处是有利于单元测试。我们的 wiki 库可能期望嵌入一个 HTTP 服务器对象来实现 wiki 网站的网络功能。如果我们想要在没有连接网络的情况下测试 wiki 网站的交互时序，那么我们可以实现一个 mock 对象来模拟 HTTP 服务器的行为。这些行为是遵照预定的脚本而不是真实的接触网络。这种方式提供了与虚拟服务器重复交互的行为，而不是依赖不可预知的网络行为。这使得测试组件与服务器的交互行为成为可能。


提示

- 使用结构类型（也称为鸭子类型）来设计灵活的对象接口。
- 结构接口更灵活、更轻量，所以应该避免使用继承。
- 针对单元测试，使用 mock 对象即接口的替代实现来提供可复验的行为。

第 58 条：区分数组对象和类数组对象

设想有两个不同类的 API。第一个是位向量：有序的位集合。

```
var bits = new BitVector();

bits.enable(4);
bits.enable([1, 3, 8, 17]);

bits.bitAt(4); // 1
bits.bitAt(8); // 1
bits.bitAt(9); // 0
```

注意 enable 方法被重载了，你可以传入一个索引或索引的数组。第二个类的 API 是字符串集合：无序的字符串集合。

```
var set = new StringSet();

set.add("Hamlet");
set.add(["Rosencrantz", "Guildenstern"]);
set.add({ "Ophelia": 1, "Polonius": 1, "Horatio": 1 });

set.contains("Polonius");      // true
set.contains("Guildenstern"); // true
set.contains("Falstaff");      // false
```

与位向量的 enable 方法类似，add 方法也被重载了，除了可以接收字符串和字符串数组外，它还可以接收一个字典对象。

为了实现 BitVector.prototype.enable 方法，我们可以通过首先测试其他情况来避免如何判断一个对象是否为数组的问题。

```
BitVector.prototype.enable = function(x) {
    if (typeof x === "number") {
        this.enableBit(x);
    } else { // assume x is array-like
        for (var i = 0, n = x.length; i < n; i++) {
            this.enableBit(x[i]);
        }
    }
};
```

这很简单。那如何实现 StringSet.prototype.add 方法呢？现在我们似乎要区分数组和对象。但这个问题完全没道理，因为 JavaScript 数组就是对象。我们真正想做的就是分离数组对象和非数组对象。

这样的区分其实与 JavaScript 的灵活的类数组对象（请参阅第 51 条）的概念是有争执的。任何对象都可被视为数组，只要它遵循正确的接口。而且也没有明确的方法来测试一个对象是否满足一个接口。我们可能尝试猜测具有 length 属性的对象可被视为数组，但这并不保险。如果我们碰巧使用一个拥有 key 为“length”的字典对象呢？

```
dimensions.add({
    "length": 1, // implies array-like?
    "height": 1,
    "width":  1
});
```

使用不精确的启发探索法来确定接口是一个容易被误解和滥用的方法。猜测一个对象是否实现了结构类型有时被称为鸭子测试（duck testing）（第 57 条描述了鸭子类型），这是不好

的实践。因为对象没有明确的信息标记来表示它们实现的结构类型，并没有可靠的编程方法来检测该信息。

重载两种类型意味着必须有一种方法来区分这两种情况。不可能检测一个值是否实现了一种结构性的接口。这引出了以下规则。

API 绝不应该重载与其他类型有重叠的类型。

对于 StringSet，答案是一开始就不要使用结构性的类数组接口。相反，我们应当选择一种类型，这种类型具有明确定义的“标签”，可以表明用户真想将其作为一个数组。一个显而易见但不完美的选择是使用 instanceof 操作符测试一个对象是否继承自 Array.prototype。

```
StringSet.prototype.add = function(x) {
    if (typeof x === "string") {
        this.addString(x);
    } else if (x instanceof Array) { // too restrictive
        x.forEach(function(s) {
            this.addString(s);
        }, this);
    } else {
        for (var key in x) {
            this.addString(key);
        }
    }
};
```

毕竟，我们知道任何时候一个对象如果是 Array 的实例，它的行为则会像一个数组。但这次截然不同。在一些允许多个全局对象的环境中可能会有标准的 Array 构造函数和原型对象的多份副本。在浏览器中有这种情况，每个 frame 会有标准库的一份单独副本。当跨 frame 通信时，一个 frame 中的数组不会继承自另一个 frame 的 Array.prototype。

出于这个原因，ES5 引入了 Array.isArray 函数，其用于测试一个值是否是数组，而不管原型继承。在 ECMAScript 标准中，该函数测试对象的内部 [[Class]] 属性值是否是 Array。当需要测试一个对象是否是真数组，而不仅仅是类数组对象，Array.isArray 方法比 instanceof 操作符更可靠。

这引出了 add 方法的一个更健壮的实现。

```
StringSet.prototype.add = function(x) {
    if (typeof x === "string") {
        this.addString(x);
    } else if (Array.isArray(x)) { // tests for true arrays
        x.forEach(function(s) {
            this.addString(s);
        }, this);
    } else {
```

```
        for (var key in x) {
            this.addString(key);
        }
    }
};
```

在不支持 ES5 的环境中，可以使用标准的 Object.prototype.toString 方法测试一个对象是否为数组。

```
var toString = Object.prototype.toString;

function isArray(x) {
    return toString.call(x) === "[object Array]";
}
```

Object.prototype.toString 函数使用对象内部的 [[Class]] 属性创建结果字符串，所以在测试一个对象是否为数组时，它比 instance of 操作符更可靠。

注意该版本的 add 方法存在不同的影响该 API 的使用者的行为。重载 API 的数组版本不接收随意的类数组对象。例如，你不能传入 arguments 对象并期待它被视为数组。

```
function MyClass() {
    this.keys = new StringSet();
    // ...
}

MyClass.prototype.update = function() {
    this.keys.add(arguments); // treated as a dictionary
};
```

相反，使用 add 方法的正确方式是将对象转换为真正的数组。可以使用第 51 条中描述的方法。

```
MyClass.prototype.update = function() {
    this.keys.add([].slice.call(arguments));
};
```

当调用者想传递一个类数组对象给一个期望接收真数组的 API 时，需要做这种转换。基于这个原因，很有必要使用文档注明 API 接收哪两种类型。在上面的例子中，enable 方法接收数字和类数组对象，而 add 方法接收字符串、真数组及（非数组）对象。

提示

- 绝不重载与其他类型有重叠的结构类型。
- 当重载一个结构类型与其他类型时，先测试其他类型。
- 当重载其他对象类型时，接收真数组而不是类数组对象。

- 文档标注你的 API 是否接收真数组或类数组值。
- 使用 ES5 提供的 Array.isArray 方法测试真数组。

第 59 条：避免过度的强制转换

众所周知，JavaScript 语言是弱类型的（请参阅第 3 条）。许多标准的操作符和代码库会自动地将非预期的输入参数强制转换为期望的类型而不是抛出异常。如果未提供额外的逻辑，使用这些内置操作符构建的程序会继承其强制转换的行为。

```
function square(x) {
    return x * x;
}

square("3"); // 9
```

强制转换无疑是很方便的。但正如第 3 条指出的一样，它们也会引起麻烦的、隐匿的错误，并导致不稳定的和难以诊断的行为。

当强制转换与重载的函数一起工作时尤其令人困惑，就像第 58 条中所涉及的位向量类的 enable 方法。该方法使用其参数的类型来决定其行为。如果 enable 方法尝试将其参数强制转换为一个期望的类型，那么方法签名可能会变得更难理解。我们应该选择哪种类型？将方法的参数强制转换为一个数字完全破坏了重载。

```
BitVector.prototype.enable = function(x) {
    x = Number(x);
    if (typeof x === "number") { // always true
        this.enableBit(x);
    } else {                     // never executed
        for (var i = 0, n = x.length; i < n; i++) {
            this.enableBit(x[i]);
        }
    }
};
```

作为一般规则，在那些使用参数类型来决定重载函数行为的函数中避免强制转换参数是明智的。强制转换使得很难识别出最终会得到哪个变量。尝试理解下面这种用法。

```
bits.enable("100"); // number or array-like?
```

这种 enable 方法的使用是含糊不清的。调用者可以合理地认为参数可以为一个数字或一个位数组值，然而我们的构造函数并不是为字符串设计的，因此无法识别它。这很可能表明调用者没有理解 API。事实上，如果我们想要更小心地设计 API，我们可以强制只接收数字和对象。

```
BitVector.prototype.enable = function(x) {
    if (typeof x === "number") {
        this.enableBit(x);
    } else if (typeof x === "object" && x) {
        for (var i = 0, n = x.length; i < n; i++) {
            this.enableBit(x[i]);
        }
    } else {
        throw new TypeError("expected number or array-like");
    }
}
```

enable 方法的最终版本是一种风格更加谨慎的示例，被称为防御性编程。防御性编程试图以额外的检查来抵御潜在的错误。一般情况下，抵御所有可能的错误是不可能的。例如，我们可能使用检查来确保如果 x 具有 length 属性，那么它应该是一个对象，然而这并不是安全的，比如，一个意外使用的 String 对象。JavaScript 除了提供实现这些检查的基本工具外，比如 typeof 操作符，还可以编写更加简洁的工具函数来监视函数签名。例如，我们可以使用一个预先检查来监视 BitVector 的构造函数。

```
function BitVector(x) {
    uint32.or(arrayLike).guard(x);
    // ...
}
```

为了使其工作，我们可以借助于共享原型对象来实现 guard 方法以构建一个监视对象的工具库。

```
var guard = {
    guard: function(x) {
        if (!this.test(x)) {

            throw new TypeError("expected " + this);
        }
    }
};
```

每个监视对象实现自己的 test 方法和错误消息的字符串描述。

```
var uint32 = Object.create(guard);

uint32.test = function(x) {
    return typeof x === "number" && x === (x >>> 0);
};

uint32.toString = function() {
    return "uint32";
};
```

uint32 的监视对象使用 JavaScript 位操作符的一个诀窍来实现到 32 位无符号整数的转换。无符号右移位运算符在执行移位运算前会将其第一个参数转换为一个 32 位的无符号整数（请参阅第 2 条）。移入零位对整数值没有影响。因此，uint32.test 实际上是把一个数字与该数字转换为 32 位无符号整数的结果做比较。

接下来，我们来实现 arrayLike 的监视对象。

```
var arrayLike = Object.create(guard);

arrayLike.test = function(x) {
    return typeof x === "object" && x && uint32.test(x.length);
};

arrayLike.toString = function() {
    return "array-like object";
};
```

请注意我们又进一步地采取了防御性编程来确保一个类数组对象应该具有一个无符号整数的 length 属性。

最后，我们实现一些作为原型方法的“链”方法（请参阅第 60 条），比如 or 方法。

```
guard.or = function(other) {
    var result = Object.create(guard);

    var self = this;
    result.test = function(x) {
        return self.test(x) || other.test(x);
    };

    var description = this + " or " + other;
    result.toString = function() {
        return description;
    };

    return result;
};
```

该方法合并了接收者监视对象（绑定到 this 的对象）和另一个监视对象（other 参数），产生一个新的监视对象。新监视对象的 test 和 toString 方法合并了这两个输入对象的方法。请注意我们使用了一个局部的 self 变量来保存 this 的引用（请参阅第 25 条和第 37 条）以确保能在合成的监视对象的 test 方法中引用。

当遇到错误时，这些测试能帮助我们更早地捕获错误，这使得它们更容易诊断。然而，它们可能扰乱代码库并潜在地影响应用程序的性能。是否使用防御性编程是一个成本（你不

得不编写和执行的额外测试的数量）和收益（你更早捕获的错误数，节省的开发和调试时间）的问题。

- 避免强制转换和重载的混用。
- 考虑防御性地监视非预期的输入。

第 60 条：支持方法链

无状态的 API（请参阅第 56 条）的部分能力是将复杂操作分解为更小的操作的灵活性。一个很好的例子是字符串的 replace 方法。由于结果本身是字符串，我们可以对前一个 replace 操作重复执行替换。这种模式的一个常见用例是在将字符串插入到 HTML 前替换字符串中的特殊字母。

```
function escapeBasicHTML(str) {
    return str.replace(/&/g, "&amp;")
              .replace(/</g, "&lt;")
              .replace(/>/g, "&gt;")
              .replace(/"/g, "&quot;")
              .replace(/'/g, "&apos;");
}
```

对 replace 的第一次调用返回了一个将所有的特殊字符 "&" 替换为 HTML 字符串转义序列 "&" 的字符串；第二次调用则将所有的 "<" 实例替换为转义序列 "<"，以此类推。这种重复的方法调用风格叫做方法链。虽然没必要写成这种风格，但是比起将每个中间结果保存为变量，它更为简洁。

```
function escapeBasicHTML(str1) {
    var str2 = str1.replace(/&/g, "&amp;");
    var str3 = str2.replace(/</g, "&lt;");
    var str4 = str3.replace(/>/g, "&gt;");
    var str5 = str4.replace(/"/g, "&quot;");
    var str6 = str5.replace(/'/g, "&apos;");
    return str6;
}
```

消除临时变量使代码更加可读，中间结果只是得到最终结果中的一个重要的步骤而已。

如果一个 API 产生了一个接口对象，调用这个接口对象的方法产生的对象如果具有相同的接口，那么就可以使用方法链。第 50 条和第 51 条描述的数组迭代方法是另一个链式 API 的很好的例子。

```
var users = records.map(function(record) {
                        return record.username;
                    })
                   .filter(function(username) {
                        return !!username;
                    })
                   .map(function(username) {
                        return username.toLowerCase();
                    });
```

此链式操作接收代表用户记录的对象的数组，提取每个记录中的 username 属性，过滤掉所有的空用户名，最后将用户名转换为小写字符串。

这种风格非常灵活，并且对于 API 的使用者富有表现力，所以将 API 设计为支持这种风格是值得的。通常情况下，无状态的 API 中，如果你的 API 不修改对象，而是返回一个新对象，则链式得到了自然的结果。因此，API 的方法提供了更多相似方法集的对象。

在有状态的设置中方法链也是值得支持的。这里的技巧是方法在更新对象时返回 this，而不是 undefined。这使得通过一个链式方法调用的序列来对同一个对象执行多次更新成为可能。

```
element.setBackgroundColor("yellow")
       .setColor("red")
       .setFontWeight("bold");
```

有状态的 API 的方法链有时被称为流畅式（fluent style）。（这个被程序员创建的词类似于 Smalltalk 的“方法级联”，这是一个对单个对象调用多个方法的内置的语法。）如果更新方法没返回 this，那么 API 的使用者不得不每次重复该对象的名称。如果该对象被简单命名为一个变量，这没有太大的区别。但当结合无状态的方法用于检索更新的对象，方法链非常简洁，并且代码更可读。前端库 jQuery 普遍采用了这种方法。它有一组（无状态的）方法用于从用户界面元素中查询网页，还有一组（有状态的）方法用于更新这些元素。

```
$("#notification")                    // find notification element
   .html("Server not responding.") // set notification message
   .removeClass("info")               // remove one set of styling
   .addClass("error");                // add more styling
```

通过调用有状态的 html、removeClass 和 addClass 方法而返回相同对象来支持流畅式，我们甚至不用创建临时变量存储 jQuery 函数执行查询的结果。当然，如果用户认为这种风格过于简洁，他们随时都可以引入一个变量来命名查询结果。

```
var element = $("#notification");
element.html("Server not responding.");
element.removeClass("info");
element.addClass("error");
```

但是通过支持方法链，API 允许程序员按自己的喜好选择风格。如果方法返回 undefined，用户会被强制使用更罗嗦的风格。

提示

- 使用方法链来连接无状态的操作。
- 通过在无状态的方法中返回新对象来支持方法链。
- 通过在有状态的方法中返回 this 来支持方法链。

第 7 章
并　　发

JavaScript 被设计为一种嵌入式的脚本语言。JavaScript 程序不是以独立的应用程序运行，而是作为大型应用程序环境下的脚本运行。典型的例子当然是 Web 浏览器。一个浏览器可能具有许多窗体和标签运行多个 Web 应用程序，每个应用程序响应不同的输入和触发源——用户通过键盘、鼠标、触摸板的动作，来自网络的数据到达，或定时警报。这些事件可能在 Web 应用程序的生命周期的任何时刻发生，甚至同时发生。针对每种事件，应用程序可能希望得到消息通知，并响应自定义行为。

在 JavaScript 中，编写响应多个并发事件的程序的方法非常人性化，而且强大，因为它使用了一个简单的执行模型（有时称为事件队列或事件循环并发）和被称为异步的 API。多亏了这一方法的有效性以及 JavaScript 独立于 Web 浏览器标准化的事实，使得 JavaScript 成为其他多种应用程序的编程语言，比如从桌面应用程序到服务器端框架的 Node.js。

奇怪的是，到目前为止，ECMAScript 标准从来没有关于并发的任何说明。因此，本章将讨论一些“约定成俗”的 JavaScript 特性，而不是官方的标准。然而，绝大多数 JavaScript 的环境都使用相同的并发策略，未来标准的版本很有可能会基于广泛实现的执行模型来标准化。不管标准如何定义，使用事件和异步 API 是 JavaScript 编程的基础部分。

第 61 条：不要阻塞 I/O 事件队列

JavaScript 程序是构建在事件之上的。输入可能同时来自于各种各样的外部源，比如用户的交互操作（点击鼠标按钮、按下按键或触摸屏幕）、输入的网络数据或定时警报。在一些语言中，我们会习惯性地编写代码来等待某个特定的输入。

```
var text = downloadSync("http://example.com/file.txt");
console.log(text);
```

（console.log API 是 JavaScript 平台中的一个通用工具方法，用以将调试信息打印到开发者控制台。）形如 downloadSync 这样的函数被称为同步函数（或阻塞函数）。程序会停止做任何工作，而等待它的输入。在这个例子中，也就是等待从网络下载文件的结果。由于在等待下载完成的期间，计算机可以做其他有用的工作，因此这样的语言通常为程序员提供一种方法来创建多个线程，即并行执行子计算。它允许程序的一部分停下来等待（阻塞）一个低速的输入，而程序的另一部分可以继续进行独立的工作。

在 JavaScript 中，大多的 I/O 操作都提供了异步的或非阻塞的 API。程序员提供一个回调函数（请参阅第 19 条），一旦输入完成就可以被系统调用，而不是将程序阻塞在等待结果的线程上。

```
downloadAsync("http://example.com/file.txt", function(text) {
    console.log(text);
});
```

该 API 初始化下载进程，然后在内部注册表中存储了回调函数后立刻返回，而不是被网络请求阻塞。当下载完成之后，系统会将下载完的文件的文本作为参数调用该已注册的回调函数。

现在，系统不仅会适时地介入其中，并且会在下载完成的瞬间调用回调函数。JavaScript 有时被称为提供一个运行到完成机制（run-to-completion）的担保。任何当前正在运行于共享上下文的用户代码，比如浏览器中的单个 Web 页面或者单个运行的 Web 服务器实例，只有在执行完成后才能调用下一个事件处理程序。实际上，系统维护了一个按事件发生顺序排列的内部事件队列，一次调用一个已注册的回调函数。

图 7.1 显示了一个客户端和服务器端应用程序事件队列的例子说明。随着事件的发生，它们被添加到应用程序的事件队列的末尾（在图 7.1 的顶部）。JavaScript 系统使用一个内部循环机制来执行应用程序。该循环机制每次都拉取队列底部的事件，也就是说，以接收到这些事件的顺序来调用这些已注册的 JavaScript 事件处理程序（类似于上面传递给 downloadAsync 函数的回调函数），并将事件的数据作为该事件处理程序的参数。

运行到完成机制担保的好处是当代码运行时，你完全掌握应用程序的状态。你根本不必担心一些变量和对象属性的改变由于并发执行代码而超出你的控制。这是一个非常好的结果。并发编程在 JavaScript 中往往比使用线程和锁的 C++、Java 或 C # 要容易得多。

然而，运行到完成机制的不足是，实际上所有你编写的代码支撑着余下应用程序的继续执行。像浏览器这样的交互式应用程序中，一个阻塞的事件处理程序会阻塞任何将被处理的其他用户输入，甚至可能阻塞一个页面的渲染，从而导致页面失去响应的用户体验。在服务器端环境中，一个阻塞的事件处理程序可能会阻塞将被处理的其他网络请求，从而导致服务器失去响应。

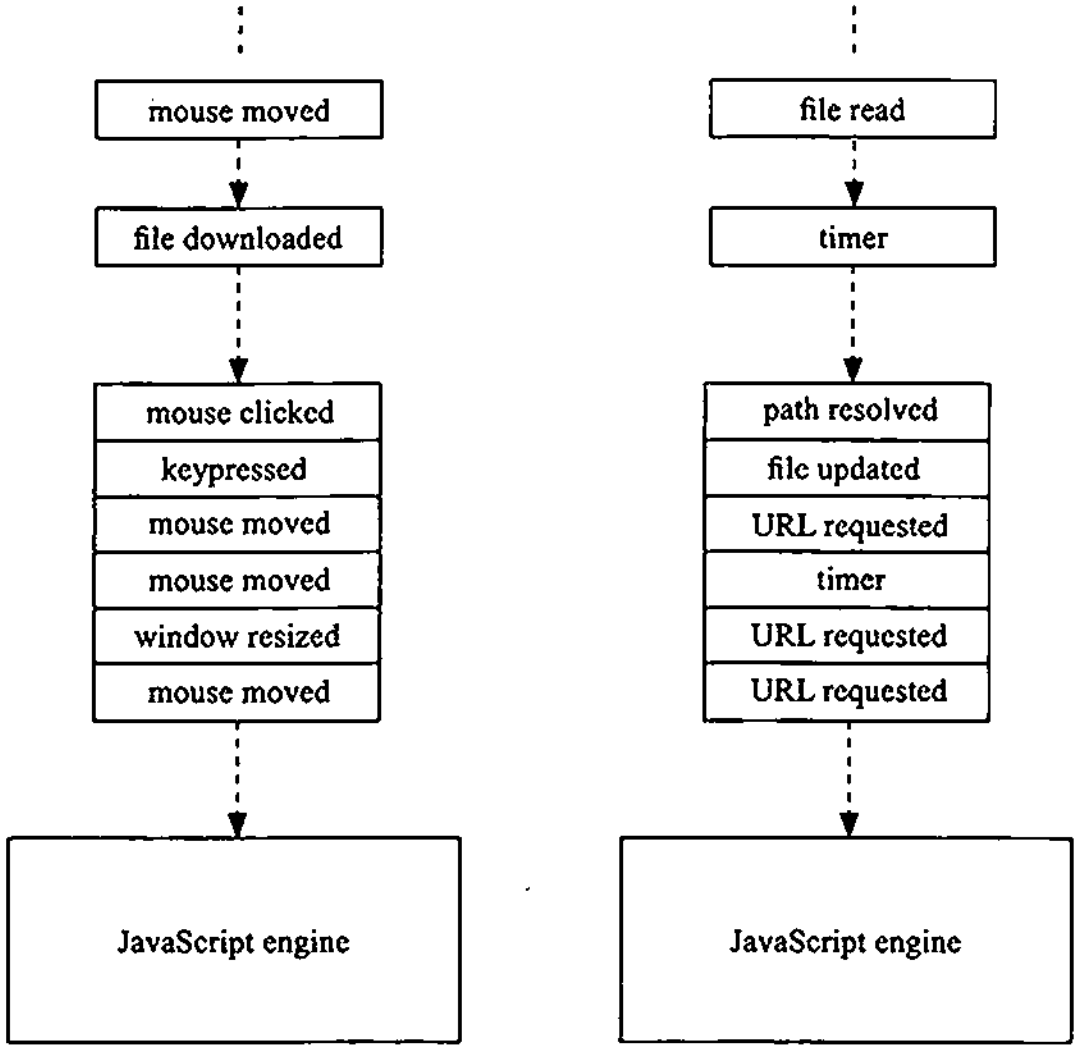

图 7.1 Web 客户端应用程序和 Web 服务器端的事件队列示例

JavaScript 并发的一个最重要的规则是绝不要在应用程序事件队列中使用阻塞 I/O 的 API。在浏览器中，甚至几乎没有任何阻塞 API 是可用的，尽管有一些已经很遗憾地泄漏到平台中很多年。提供类似于上面 downloadAsync 功能的网络 I/O 的 XMLHttpRequest 库有一个同步的版本实现，被认为是一种不好的实现。对于 Web 应用程序的交互性，同步的 I/O 会导致灾难性的后果，它在 I/O 操作完成之前一直会阻塞用户与页面的交互。

相比之下，异步的 API 用在基于事件的环境中是安全的，因为它们迫使应用程序逻辑在一个独立的事件循环“轮询”中继续处理。在上面的例子中，假设需要几秒钟来下载网络资源。在这段时间里，数量庞大的其他事件很可能发生。在同步的实现中，这些事件就会堆积在事件队列中，而事件循环将停留等待该 JavaScript 代码执行完成，这将阻塞任何其他事件的处理。但是在异步的版本中，JavaScript 代码注册一个事件处理程序并立即返回，这将在下载完成之前，允许其他事件处理程序处理这期间的事件。

在主应用程序事件队列不受影响的环境中，阻塞操作很少出问题。例如，Web 平台提供了 Worker 的 API，该 API 使得产生大量的并行计算成为可能。不同于传统的线程执行，Workers 在一个完全隔离的状态下执行，没有获取全局作用域或应用程序主线程 Web 页面内容的能力。因此，它们不会妨碍主事件队列中运行的代码的执行。在一个 Worker 中，使用

XMLHttpRequest 同步的变种很少出问题；下载操作的确会阻塞 Worker 继续执行，但这并不会阻止页面的渲染或事件队列中的事件响应。在服务器端环境中，阻塞的 API 在启动一开始是没有问题的，也就是在服务器开始响应输入的请求之前。然而在处理请求期间，浏览器事件队列中存在阻塞的 API 就是彻头彻尾的灾难。

- 异步 API 使用回调函数来延缓处理代价高昂的操作以避免阻塞主应用程序。
- JavaScript 并发地接收事件，但会使用一个事件队列按序地处理事件处理程序。
- 在应用程序事件队列中绝不要使用阻塞的 I/O。

第 62 条：在异步序列中使用嵌套或命名的回调函数

第 61 条讲述了异步 API 如何执行潜在的代价高昂的 I/O 操作，而不阻塞应用程序继续处理其他输入。理解异步程序的操作顺序刚开始有点混乱。例如，下面的程序在打印“finished”之前打印“starting”，即使这两个打印动作在程序源文件中以相反的顺序呈现。

```
downloadAsync("file.txt", function(file) {
    console.log("finished");
});
console.log("starting");
```

downloadAsync 调用会立即返回，不会等待文件完成下载。同时，JavaScript 的运行到完成机制确保下一行代码会在处理其他事件处理程序之前被执行。这意味着“starting”一定会在“finished”之前被打印。

理解操作序列的最简单的方式是异步 API 是发起操作而不是执行操作。上面的代码发起了一个文件的下载然后立即打印出了“starting”。当下载完成后，在事件循环的某个单独的轮次中，被注册的事件处理程序才会打印出“finished”。

如果你需要在发起一个操作后做一些事情，如果只能在一行中放置好几个声明，那么如何串联已完成的异步操作呢？例如，如果我们需要在异步数据库中查找一个 URL，然后下载这个 URL 的内容？不可能发起两个连续请求。

```
db.lookupAsync("url", function(url) {
    // ?
});
downloadAsync(url, function(text) { // error: url is not bound
    console.log("contents of " + url + ": " + text);
});
```

这不可能工作，因为从数据库查询到的 URL 结果需要作为 downloadAsync 方法的参数，

但是它并不在作用域内。这有很好的理由：我们所做的这一步是发起数据库查找，查找的结果还不可用。

最简单的答案是使用嵌套。借助于闭包的魔力（参见第 11 条），我们可以将第二个动作嵌套在第一个动作的回调函数中。

```
db.lookupAsync("url", function(url) {
    downloadAsync(url, function(text) {
        console.log("contents of " + url + ": " + text);
    });
});
```

这仍然有两个回调函数，但第二个被包含在第一个中，创建闭包能够访问外部回调函数的变量。请注意第二个回调函数是如何引用 url 变量的。

嵌套的异步操作很容易，但当扩展到更长的序列时会很快变得笨拙。

```
db.lookupAsync("url", function(url) {
    downloadAsync(url, function(file) {
        downloadAsync("a.txt", function(a) {
            downloadAsync("b.txt", function(b) {
                downloadAsync("c.txt", function(c) {
                    // ...
                });
            });
        });
    });
});
```

减少过多嵌套的方法之一是将嵌套的回调函数作为命名的函数，并将它们需要的附加数据作为额外的参数传递。上述的例子可改为：

```
db.lookupAsync("url", downloadURL);

function downloadURL(url) {
    downloadAsync(url, function(text) { // still nested
        showContents(url, text);
    });
}

function showContents(url, text) {
    console.log("contents of " + url + ": " + text);
}
```

为了合并外部的 url 变量和内部的 text 变量作为 showContents 方法的参数，在 downloadURL 方法中仍然使用了嵌套的回调函数。我们可以使用 bind 方法（参见第 25 条）

消除最深层的嵌套回调函数。

```
db.lookupAsync("url", downloadURL);

function downloadURL(url) {
    downloadAsync(url, showContents.bind(null, url));
}

function showContents(url, text) {
    console.log("contents of " + url + ": " + text);
}
```

这种做法导致了代码看起来更具顺序性，但需要为操作序列的每个中间步骤命名，并且一步步地使用绑定。这可能导致尴尬的情况，正如上述例子。

```
db.lookupAsync("url", downloadURLAndFiles);

function downloadURLAndFiles(url) {
    downloadAsync(url, downloadABC.bind(null, url));
}

// awkward name
function downloadABC(url, file) {
    downloadAsync("a.txt",
                  // duplicated bindings
                  downloadFiles23.bind(null, url, file));
}

// awkward name
function downloadBC(url, file, a) {
    downloadAsync("b.txt",
                  // more duplicated bindings
                  downloadFile3.bind(null, url, file, a));
}

// awkward name
function downloadC(url, file, a, b) {
    downloadAsync("c.txt",
                  // still more duplicated bindings
                  finish.bind(null, url, file, a, b));
}

function finish(url, file, a, b, c) {
    // ...
}
```

有时结合这两种方法会取得更好的平衡，但仍然存在一些嵌套。

```
db.lookupAsync("url", function(url) {
    downloadURLAndFiles(url);
});

function downloadURLAndFiles(url) {
    downloadAsync(url, downloadFiles.bind(null, url));
}

function downloadFiles(url, file) {
    downloadAsync("a.txt", function(a) {
        downloadAsync("b.txt", function(b) {
            downloadAsync("c.txt", function(c) {
                // ...
            });
        });
    });
}
```

更胜一筹的方法是最后一步可以使用一个额外的抽象来简化，可以下载多个文件并将它们存储在数组中。

```
function downloadFiles(url, file) {
    downloadAllAsync(["a.txt", "b.txt", "c.txt"],
                     function(all) {

        var a = all[0], b = all[1], c = all[2];
        // ...
    });
}
```

使用 downloadAllAsync 函数也允许我们同时下载多个文件。排序意味着每个操作只有等前一个操作完成后才能启动。一些操作本质上是连续的，比如下载我们从数据库查询到的URL。但如果我们有一个文件列表要下载，没理由等每个文件完成下载后才请求接下来的一个。第 66 条解释了如何实现对并行的抽象，正如 downloadAllAsync 函数一样。

除了嵌套和命名回调，还可能建立更高层的抽象使异步控制流更简单、更简洁。第 68 条描述了一个特别流行的做法。除此之外，也值得探索一些异步的程序库或尝试使用自己的抽象。

提示

- 使用嵌套或命名的回调函数按顺序地执行多个异步操作。
- 尝试在过多的嵌套的回调函数和尴尬的命名的非嵌套回调函数之间取得平衡。
- 避免将可被并行执行的操作顺序化。

第 63 条：当心丢弃错误

管理异步编程的一个比较困难的方面是对错误的处理。对于同步的代码，通过使用 try 语句块包装一段代码很容易一下子处理所有的错误。

```
try {
    f();
    g();
    h();
} catch (e) {
    // handle any error that occurred...
}
```

而对于异步的代码，多步的处理通常被分隔到事件队列的单独轮次中，因此，不可能将它们全部包装在一个 try 语句块中。事实上，异步的 API 甚至根本不可能抛出异常，因为，当一个异步的错误发生时，没有一个明显的执行上下文来抛出异常！相反，异步的 API 倾向于将错误表示为回调函数的特定参数，或使用一个附加的错误处理回调函数（有时被称为 errbacks）。例如，一个类似于第 61 条中涉及的下载文件的异步 API 可能会有一个额外的回调函数来处理网络错误。

```
downloadAsync("http://example.com/file.txt", function(text) {
    console.log("File contents: " + text);
}, function(error) {
    console.log("Error: " + error);
});
```

如果下载多个文件，你可以参照第 62 条所讲述的将回调函数嵌套起来。

```
downloadAsync("a.txt", function(a) {
    downloadAsync("b.txt", function(b) {
        downloadAsync("c.txt", function(c) {
            console.log("Contents: " + a + b + c);
        }, function(error) {
            console.log("Error: " + error);
        });
    }, function(error) { // repeated error-handling logic
        console.log("Error: " + error);
    });
}, function(error) { // repeated error-handling logic
    console.log("Error: " + error);
});
```

请注意在这个例子中，每一步的处理都使用了相同的错误处理逻辑，然而我们在多个地方重复了相同的代码。在编程领域里，我们应该努力坚持避免重复代码。通过在一个共享的

作用域中定义一个错误处理的函数，我们很容易将重复代码抽象出来。

```
function onError(error) {
    console.log("Error: " + error);
}

downloadAsync("a.txt", function(a) {
    downloadAsync("b.txt", function(b) {
        downloadAsync("c.txt", function(c) {
            console.log("Contents: " + a + b + c);
        }, onError);
    }, onError);
}, onError);
```

当然，如果我们使用工具函数 downloadAllAsync（正如第 62 条和第 66 条推荐的）将多个步骤合并到一个复合的操作中，那么很自然，我们最终只需要提供一个错误处理的回调函数。

```
downloadAllAsync(["a.txt", "b.txt", "c.txt"], function(abc) {
    console.log("Contents: " + abc[0] + abc[1] + abc[2]);
}, function(error) {
    console.log("Error: " + error);
});
```

另一种错误处理 API 的风格受到 Node.js 平台的推广。该风格只需要一个回调函数，该回调函数的第一个参数如果有错误发生那就表示为一个错误；否则就为一个假值，比如 null。对于这类 API，我们仍然可以定义一个通用的错误处理函数，但是我们需要使用 if 语句来控制每个回调函数。

```
function onError(error) {
    console.log("Error: " + error);
}

downloadAsync("a.txt", function(error, a) {
    if (error) {
        onError(error);
        return;
    }
    downloadAsync("b.txt", function(error, b) {
        // duplicated error-checking logic
        if (error) {
            onError(error);
            return;
        }
        downloadAsync(url3, function(error, c) {
```

```
            // duplicated error-checking logic
            if (error) {
                onError(error);
                return;
            }
            console.log("Contents: " + a + b + c);
        });
    });
});
```

在一些使用这种错误处理回调函数风格的框架中，程序员通常会放弃 if 语句而使用大括号结构跨越多行的约定，以使得错误处理更简洁、更集中。

```
function onError(error) {
    console.log("Error: " + error);
}

downloadAsync("a.txt", function(error, a) {
    if (error) return onError(error);

    downloadAsync("b.txt", function(error, b) {
        if (error) return onError(error);

        downloadAsync(url3, function(error, c) {
            if (error) return onError(error);

            console.log("Contents: " + a + b + c);
        });
    });
});
```

或者，一如既往地使用一个抽象合并步骤来帮助消除重复。

```
var filenames = ["a.txt", "b.txt", "c.txt"];

downloadAllAsync(filenames, function(error, abc) {
    if (error) {
        console.log("Error: " + error);
        return;
    }
    console.log("Contents: " + abc[0] + abc[1] + abc[2]);
});
```

try...catch 语句和在异步 API 中典型的错误处理逻辑的一个实际差异是，try 语句使得定义一个“捕获所有”的逻辑很容易导致程序员难以忘怀整个代码区的错误处理。而像上面给出的异步 API，我们非常容易忘记在进程的任意一步提供错误处理。通常，这将导致错误被

默默地丢弃。忽视错误处理的程序会令用户非常沮丧：应用程序出错时没有任何的反馈（有时会导致出现一个从不被清理的悬挂进程通知）。类似的，默认的错误也是调试的噩梦，因为它们没有提供问题来源的线索。最好的治疗方法是做好防御，即使用异步 API 需要警惕，确保明确地处理所有的错误状态条件。

- 通过编写共享的错误处理函数来避免复制和粘贴错误处理代码。
- 确保明确地处理所有的错误条件以避免丢弃错误。

第 64 条：对异步循环使用递归

设想有一个函数接收一个 URL 的数组并尝试依次下载每个文件直到有一个文件被成功下载。如果 API 是同步的，很容易使用一个循环来实现。

```
function downloadOneSync(urls) {
    for (var i = 0, n = urls.length; i < n; i++) {
        try {
            return downloadSync(urls[i]);
        } catch (e) { }
    }
    throw new Error("all downloads failed");
}
```

但是这种方式实现的 downloadOneAsync 并不能正确工作。因为我们不能在回调函数中暂停循环并恢复。如果我们尝试使用循环，它将启动所有的下载，而不是等待一个完成再试下一个。

```
function downloadOneAsync(urls, onsuccess, onerror) {
    for (var i = 0, n = urls.length; i < n; i++) {
        downloadAsync(urls[i], onsuccess, function(error) {
            // ?
        });
        // loop continues
    }
    throw new Error("all downloads failed");
}
```

因此我们要实现一个类似循环的东西，但只有我们显式地说继续执行，它才会继续执行。解决方案是将循环实现为一个函数，所以我们可以决定何时开始每次迭代。

```
function downloadOneAsync(urls, onsuccess, onfailure) {
    var n = urls.length;
```

```
    function tryNextURL(i) {
        if (i >= n) {

            onfailure("all downloads failed");
            return;
        }
        downloadAsync(urls[i], onsuccess, function() {
            tryNextURL(i + 1);
        });
    }

    tryNextURL(0);
}
```

局部函数 tryNextURL 是一个递归函数。它的实现调用了其自身。目前典型的 JavaScript 环境中一个递归函数同步调用自身过多次会导致失败。例如，下例中的递归函数试图调用自身 10 万次，在大多数的 JavaScript 环境中会产生一个运行时错误。

```
function countdown(n) {
    if (n === 0) {
        return "done";
    } else {
        return countdown(n - 1);
    }
}
```

当 n 太大时 countdown 函数会执行失败，那么如何确保 downloadOneAsync 函数是安全的呢？回答这个问题之前，让我们走个小弯路，查看一下 countdown 函数提供的错误信息。

JavaScript 环境通常在内存中保存一块固定的区域，称为调用栈，用于记录函数调用返回前下一步该做什么。想象执行下面的小程序。

```
function negative(x) {
    return abs(x) * -1;
}

function abs(x) {
    return Math.abs(x);
}

console.log(negative(42));
```

当程序使用参数 42 调用 Math.abs 方法时，有好几个其他的函数调用也在进行，每个都在等待另一个的调用返回。图 7.2 说明了这一刻的调用栈。在每个函数调用时，项目符号 (•) 描述了在程序中已经发生的函数调用地方及这次调用完成后将返回哪里。就像传统的栈数

据结构，这个信息遵循“先进后出”协议。最新的函数调用将信息推入栈（被表示为栈的最底层的帧），该信息也将首先从栈中弹出。当 Math.abs 执行完毕，将会返回给 abs 函数，其将返回给 negative 函数，然后将返回到最外面的脚本。

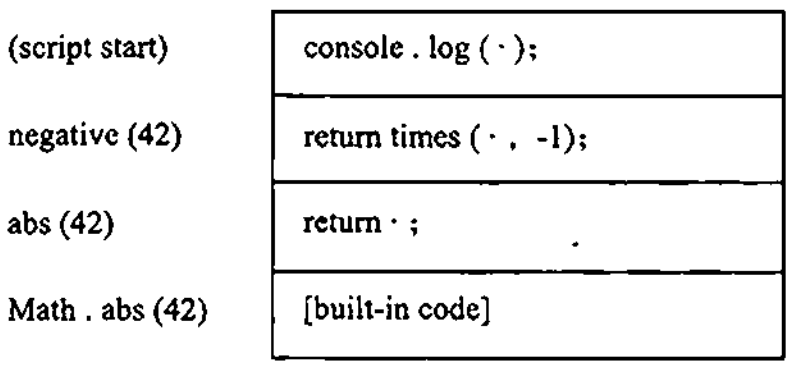

图 7.2 在一个简单的程序执行期间的调用栈

当一个程序执行中有太多的函数调用，它会耗尽栈空间，最终抛出异常。这种情况被称为栈溢出。在此例中，调用 countdown(10 万次) 需要 countdown 调用自身 10 万次，每次推入一个栈帧，如图 7.3 所示。存储这么多栈帧需要的空间显会耗尽大多数 JavaScript 环境分配的空间，导致运行时错误。

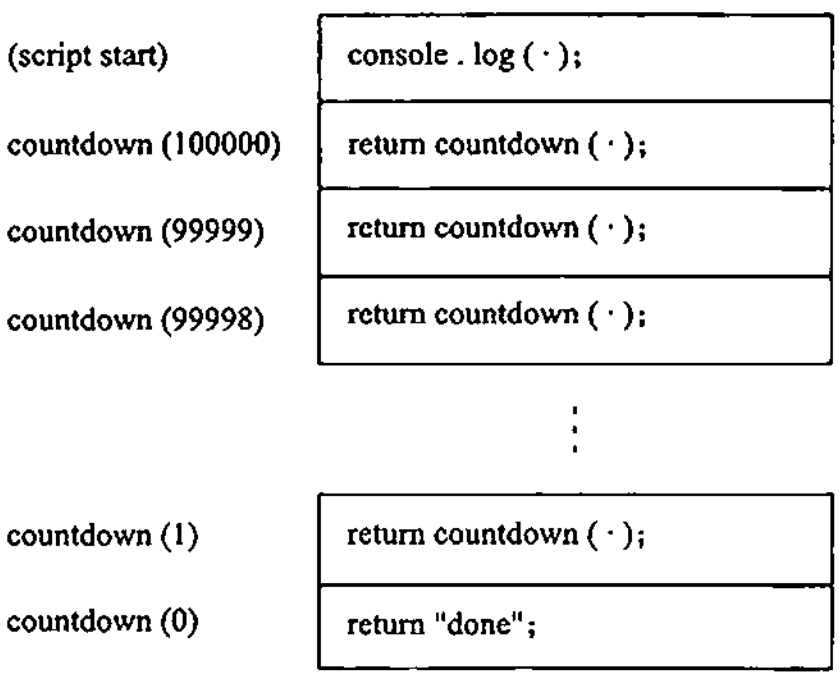

图 7.3 递归函数执行期间的调用栈

现在再看看 downloadOneAsync 函数。不像 countdown 直到递归调用返回后才会返回，downloadOneAsync 只在异步回调函数中调用自身。记住异步 API 在其回调函数被调用前会立即返回。所以 downloadOneAsync 返回，导致其栈帧在任何递归调用将新的栈帧推入栈前，会从调用栈中弹出。（事实上，回调函数总在事件循环的单独轮次中被调用，事件循环的每个轮次中调用其事件处理程序的调用栈最初是空的。）所以无论 downloadOneAsync 需要多少次迭代，都不会耗尽栈空间。

提示

- 循环不能是异步的。
- 使用递归函数在事件循环的单独轮次中执行迭代。
- 在事件循环的单独轮次中执行递归，并不会导致调用栈溢出。

第 65 条：不要在计算时阻塞事件队列

第 61 条解释了异步 API 怎样帮助我们防止一段程序阻塞应用程序的事件队列。但这并不是故事的全部。毕竟，每个程序员都可以告诉你，即使没有一个函数调用也很容易使一个应用程序陷入泥潭。

```
while (true) { }
```

而且它并不需要一个无限循环来写一个缓慢的程序。代码需要时间来运行，而低效的算法或数据结构可能导致运行长时间的计算。

当然，效率不是 JavaScript 唯一关注的。但是基于事件的编程的确强加了一些特殊的约束。为了保持客户端应用程序的高度交互性和确保所有传入的请求在服务器应用程序中得到充分的服务，保持事件循环的每个轮次尽可能短是至关重要的。否则，事件队列会滞销，其增长速度会超过分发处理事件处理程序的速度。在浏览器环境中，一些代价高昂的计算也会导致糟糕的用户体验，因为一个页面的用户界面无响应多数是由于在运行 JavaScript 代码。

那么，如果你的应用程序需要执行代价高昂的计算你该怎么办呢？没有一个完全正确的答案，但有一些通用的技术可用。也许最简单的方法是使用像 Web 客户端平台的 Worker API 这样的并发机制。这对于需要搜索大量可移动距离的人工智能游戏是一个很好的方法。游戏可能以生成大量的专门计算移动距离的 worker 开始。

```
var ai = new Worker("ai.js");
```

这将使用 ai.js 源文件作为 worker 的脚本，产生一个新的线程独立的事件队列的并发执行线程。该 worker 运行在一个完全隔离的状态——没有任何应用程序对象的直接访问。但是，应用程序与 worker 之间可以通过发送形式为字符串的 messages 来交互。所以，每当游戏需要程序计算移动时，它会发送一个消息给 worker。

```
var userMove = /* ... */;

ai.postMessage(JSON.stringify({
    userMove: userMove
}));
```

postMessage 的参数被作为一个消息增加到 worker 的事件队列中。为了处理 worker 的响应，游戏会注册一个事件处理程序。

```
ai.onmessage = function(event) {
    executeMove(JSON.parse(event.data).computerMove);
};
```

与此同时，源文件 ai.js 指示 worker 监听消息并执行计算下一步移动所需的工作。

```
self.onmessage = function(event) {
    // parse the user move
    var userMove = JSON.parse(event.data).userMove;

    // generate the next computer move
    var computerMove = computeNextMove(userMove);

    // format the computer move
    var message = JSON.stringify({
        computerMove: computerMove
    });

    self.postMessage(message);
};

function computeNextMove(userMove) {
    // ...
}
```

不是所有的 JavaScript 平台都提供了类似 Worker 这样的 API。而且有时传递消息的开销可能会过于昂贵。另一种方法是将算法分解为多个步骤，每个步骤组成一个可管理的工作块。考虑下第 48 条中搜索社交网络图的工作表算法。

```
Member.prototype.inNetwork = function(other) {
    var visited = {};
    var worklist = [this];
    while (worklist.length > 0) {
        var member = worklist.pop();
        // ...
        if (member === other) { // found?
            return true;
        }
        // ...
    }
    return false;
};
```

如果这段程序核心的 while 循环代价太过高昂，搜索工作很可能会以不可接受的时间运行而阻塞应用程序事件队列。即使我们可以使用 Worker API，它也是昂贵或不方便实现的，因为它需要复制整个网络图的状态或在 worker 中存储网络图的状态，并总是使用消息传递来更新和查询网络。

幸运的是，这种算法被定义为一个步骤集的序列——while 循环的迭代。我们可以通过增加一个回调参数将 inNetwork 转换为一个匿名函数，并像第 64 条讲述的，将 while 循环替换为一个匿名的递归函数。

```
Member.prototype.inNetwork = function(other, callback) {
    var visited = {};
    var worklist = [this];
    function next() {
        if (worklist.length === 0) {
            callback(false);
            return;
        }
        var member = worklist.pop();
        // ...
        if (member === other) { // found?
            callback(true);
            return;
        }
        // ...
        setTimeout(next, 0); // schedule the next iteration
    }
    setTimeout(next, 0); // schedule the first iteration
};
```

让我们仔细地看看这段代码是如何工作的。为了替换 while 循环，我们写了一个局部的 next 函数，该函数执行循环中的单个迭代然后调度应用程序事件队列来异步运行下一次迭代。这使得在此期间已经发生的其他事件被处理后才继续下一次迭代。当搜索完成后，通过找到一个匹配或遍历完整个工作表，我们使用结果值调用回调函数并通过调用没有调度任何迭代的 next 来返回，从而有效地完成循环。

要调度迭代，我们使用多数 JavaScript 平台都可用的、通用的 setTimeout API 来注册 next 函数，使 next 函数经过一段最少时间（0 毫秒）后运行。这具有几乎立刻将回调函数添加到事件队列上的作用。值得注意的是，虽然 setTimeout 有相对稳定的跨平台移植性，但通常还有更好的替代方案。例如，在浏览器环境中，最低的超时时间被压制为 4 毫秒，我们可以采用一种替代方案，使用 postMessage 立即压入一个事件。

如果应用程序事件队列的每个轮次中只执行算法的一个迭代，那就杀鸡用牛刀了。我们可以调整算法，自定义每个轮次中的迭代次数。这很容易实现，只须在 next 函数的主要部分的外围使用一个循环计数器。

```
Member.prototype.inNetwork = function(other, callback) {
    // ...
    function next() {
        for (var i = 0; i < 10; i++) {
            // ...
        }
        setTimeout(next, 0);
```

```
        }
        setTimeout(next, 0);
    };
```

- 避免在主事件队列中执行代价高昂的算法。
- 在支持 Worker API 的平台，该 API 可以用来在一个独立的事件队列中运行长计算程序。
- 在 Worker API 不可用或代价昂贵的环境中，考虑将计算程序分解到事件循环的多个轮次中。

第 66 条：使用计数器来执行并行操作

第 63 条建议使用工具函数 downloadAllAsync 接收一个 URL 数组并下载所有文件，结果返回一个存储了文件内容的数组，每个 URL 对应一个字符串。downloadAllAsync 并不只有清理嵌套回调函数的好处，其主要好处是并行下载文件。我们可以在同一个事件循环中一次启动所有文件的下载，而不用等待每个文件完成下载。

并行逻辑是微妙的，很容易出错。下面的实现有一个隐蔽的缺陷。

```
function downloadAllAsync(urls, onsuccess, onerror) {
    var result = [], length = urls.length;

    if (length === 0) {
        setTimeout(onsuccess.bind(null, result), 0);
        return;
    }

    urls.forEach(function(url) {
        downloadAsync(url, function(text) {
            if (result) {
                // race condition
                result.push(text);
                if (result.length === urls.length) {
                    onsuccess(result);
                }
            }
        }, function(error) {
            if (result) {
                result = null;
                onerror(error);
            }
```

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

这个函数有严重的错误，但首先让我们看看它是如何工作的。我们先确保如果数组是空的，则会使用空结果数组调用回调函数。如果不这样做，这两个回调函数将不会被调用，因为 forEach 循环是空的。（第 67 条解释了为什么我们使用 setTimeout 函数来调用 onsuccess 回调函数，而不是直接调用 onsuccess。）接下来，我们遍历整个 URL 数组，为每个 URL 请求一个异步下载。每次下载成功，我们就将文件内容加入到 result 数组中。如果所有的 URL 都被成功下载，我们使用 result 数组调用 onsuccess 回调函数。如果有任何失败的下载，我们使用错误值调用 onerror 回调函数。如果有多个下载失败，我们还设置 result 数组为 null，从而保证 onerror 只被调用一次，即在第一次错误发生时。

想要了解到底哪里错了，可以参见下面的代码。

```
var filenames = [
    "huge.txt",  // huge file
    "tiny.txt",  // tiny file
    "medium.txt" // medium-sized file
];
downloadAllAsync(filenames, function(files) {
    console.log("Huge file: " + files[0].length);   // tiny
    console.log("Tiny file: " + files[1].length);   // medium
    console.log("Medium file: " + files[2].length); // huge
}, function(error) {
    console.log("Error: " + error);
});
```

由于这些文件是并行下载的，事件可以以任意的顺序发生（因此被添加到应用程序事件序列）。例如，如果 tiny.txt 先下载完成，接下来是 medium.txt 文件，最后是 huge.txt 文件，则注册到 downloadAllAsync 的回调函数并不会按照它们被创建的顺序进行调用。但 downloadAllAsync 的实现是一旦下载完成就立即将中间结果保存在 result 数组的末尾。所以 downloadAllAsync 函数提供的保存下载文件内容的数组的顺序是未知的。几乎不可能正确使用这样的 API，因为调用者无法找出哪个结果对应哪个文件。上面的例子假设结果与输入的数组有相同的顺序，在这种情况下则完全错误。

第 48 条介绍了不确定性的概念。如果行为不可预知，则不能信赖程序中不确定的行为。并发事件是 JavaScript 中不确定性的重要来源。具体来说，程序的执行顺序不能保证与事件发生的顺序一致。

当一个应用程序依赖于特定的事件顺序才能正常工作时，这个程序会遭受数据竞争 (data race)。数据竞争是指多个并发操作可以修改共享的数据结构，这取决于它们发生的顺序。（直

觉上，并发操作相互之间“竞争”哪个先完成。）数据竞争是真正棘手的错误。它们可能不会出现于特定的测试中，因为运行相同的程序两次，每次可能会得到不同的结果。例如，downloadAllAsync 的使用者可能会对文件重新排序，基于的顺序是哪个文件可能会最先完成下载。

```
downloadAllAsync(filenames, function(files) {
    console.log("Huge file: " + files[2].length);
    console.log("Tiny file: " + files[0].length);
    console.log("Medium file: " + files[1].length);
}, function(error) {
    console.log("Error: " + error);
});
```

在这种情况下大多时候结果是相同的顺序，但偶尔由于改变了服务器负载均衡或网络缓存，文件可能不是期望的顺序。这往往是诊断 Bug 的最大挑战，因为它们很难重现。当然，我们可以顺序下载文件，但却失去了并发的性能优势。

下面的方式可以实现 downloadAllAsync 不依赖不可预期的事件执行顺序而总能提供预期结果。我们不将每个结果放置到数组末尾，而是存储在其原始的索引位置中。

```
function downloadAllAsync(urls, onsuccess, onerror) {
var length = urls.length;
var result = [];

if (length === 0) {
    setTimeout(onsuccess.bind(null, result), 0);
    return;
}

urls.forEach(function(url, i) {
    downloadAsync(url, function(text) {
        if (result) {
            result[i] = text; // store at fixed index

            // race condition
            if (result.length === urls.length) {
                onsuccess(result);
            }
        }
    }, function(error) {
        if (result) {
            result = null;
            onerror(error);
        }
    });
```

```
    });
}
```

该实现利用了 forEach 回调函数的第二个参数。第二个参数为当前迭代提供了数组索引。不幸的是，这仍然不正确。第 51 条描述了数组更新的契约，即设置一个索引属性，总是确保数组的 length 属性值大于索引。假设有如下的一个请求。

```
downloadAllAsync(["huge.txt", "medium.txt", "tiny.txt"]);
```

如果 tiny.txt 文件最先被下载，结果数组将获取索引为 2 的属性，这将导致 result.length 被更新为 3。用户的 success 回调函数将被过早地调用，其参数为一个不完整的结果数组。

正确的实现应该是使用一个计数器来追踪正在进行的操作数量。

```
function downloadAllAsync(urls, onsuccess, onerror) {
    var pending = urls.length;
    var result = [];

    if (pending === 0) {
        setTimeout(onsuccess.bind(null, result), 0);
        return;
    }

    urls.forEach(function(url, i) {
        downloadAsync(url, function(text) {
            if (result) {
                result[i] = text; // store at fixed index
                pending--;        // register the success
                if (pending === 0) {
                    onsuccess(result);
                }
            }
        }, function(error) {
            if (result) {
                result = null;
                onerror(error);
            }
        });
    });
}
```

现在不论事件以什么样的顺序发生，pending 计数器都能准确地指出何时所有的事件会被完成，并以适当的顺序返回完整的结果。

提示

- JavaScript 应用程序中的事件发生是不确定的，即顺序是不可预测的。

❑ 使用计数器避免并行操作中的数据竞争。

第 67 条：绝不要同步地调用异步的回调函数

设想有 downloadAsync 函数的一种变种，它持有一个缓存（实现为一个 Dict，请参阅第 45 条）来避免多次下载同一个文件。在文件已经被缓存的情况下，立即调用回调函数是最优选择。

```
var cache = new Dict();

function downloadCachingAsync(url, onsuccess, onerror) {
    if (cache.has(url)) {
        onsuccess(cache.get(url)); // synchronous call
        return;
    }
    return downloadAsync(url, function(file) {
        cache.set(url, file);
        onsuccess(file);
    }, onerror);
}
```

通常情况下，如果可以，它似乎会立即提供数据，但这以微妙的方式违反了异步 API 客户端的期望。首先，它改变了操作的预期顺序。第 62 条显示了下面的例子，对于一个循规蹈矩的异步 API 应该总是以一种可预测的顺序来记录日志消息。

```
downloadAsync("file.txt", function(file) {
    console.log("finished");
});
console.log("starting");
```

使用上面的 downloadCachingAsync 实现，这样的客户端代码可能最终会以任意的顺序记录事件，这取决于文件是否已被缓存起来。

```
downloadCachingAsync("file.txt", function(file) {
    console.log("finished"); // might happen first
});
console.log("starting");
```

日志消息的顺序是一回事。更一般的是，异步 API 的目的是维持事件循环中每轮的严格分离。正如第 61 条解释的，这简化了并发，通过减轻每轮事件循环的代码量而不必担心其他代码并发地修改共享的数据结构。同步地调用异步的回调函数违反了这一分离，导致在当前轮完成之前，代码用于执行一轮隔离的事件循环。

例如，应用程序可能会持有一个剩余的文件队列给用户下载和显示消息。

```
downloadCachingAsync(remaining[0], function(file) {
    remaining.shift();
    // ...
});

status.display("Downloading " + remaining[0] + "...");
```

如果同步地调用该回调函数，那么将显示错误的文件名的消息（或者更糟糕的是，如果队列为空会显示“undefined”）。

同步的调用异步的回调函数甚至可能会导致一些微妙的问题。第 64 条解释了异步的回调函数本质上是以空的调用栈来调用，因此将异步的循环实现为递归函数是安全的，完全没有累积超越调用栈空间的危险。同步的调用不能保障这一点，因而使得一个表面上的异步循环很可能会耗尽调用栈空间。另一种问题是异常。对于上面的 downloadCachingAsync 实现，如果回调函数抛出一个异常，它将会在每轮的事件循环中，也就是开始下载时而不是期望的一个分离的回合中抛出该异常。

为了确保总是异步地调用回调函数，我们可以使用已存在的异步 API。就像我们在第 65 条和第 66 条中所做的一样，我们使用通用的库函数 setTimeout 在每隔一个最小的超时时间后给事件队列增加一个回调函数。可能有比 setTimeout 函数更完美的替代方案来调度即时事件，这取决于特定平台。

```
var cache = new Dict();

function downloadCachingAsync(url, onsuccess, onerror) {
    if (cache.has(url)) {
        var cached = cache.get(url);
        setTimeout(onsuccess.bind(null, cached), 0);
        return;
    }
    return downloadAsync(url, function(file) {
        cache.set(url, file);
        onsuccess(file);
    }, onerror);
}
```

我们使用 bind 函数（请参阅第 25 条）将结果保存为 onsuccess 回调函数的第一个参数。

提示

- 即使可以立即得到数据，也绝不要同步地调用异步回调函数。
- 同步地调用异步的回调函数扰乱了预期的操作序列，并可能导致意想不到的交错代码。
- 同步地调用异步的回调函数可能导致栈溢出或错误地处理异常。

- 使用异步的 API，比如 setTimeout 函数来调度异步回调函数，使其运行于另一个回合。

第 68 条：使用 promise 模式清洁异步逻辑

构建异步 API 的一种流行的替代方式是使用 promise（有时也被称为 deferred 或 future）模式。已经在本章讨论过的异步 API 使用回调函数作为参数。

```
downloadAsync("file.txt", function(file) {
    console.log("file: " + file);
});
```

相比之下，基于 promise 的 API 不接收回调函数作为参数。相反，它返回一个 promise 对象，该对象通过其自身的 then 方法接收回调函数。

```
var p = downloadP("file.txt");

p.then(function(file) {
    console.log("file: " + file);
});
```

到目前为止，还看不出这与原先的版本有什么不同。但是 promise 的力量在于它们的组合性。传递给 then 方法的回调函数不仅产生影响（在上述的例子中，打印到控制台），也可以产生结果。通过回调函数返回一个值，我们可以构造一个新的 promise。

```
var fileP = downloadP("file.txt");

var lengthP = fileP.then(function(file) {
    return file.length;
});


lengthP.then(function(length) {
    console.log("length: " + length);
});
```

理解 promise 的一种方法是将它理解为表示最终值的对象。它封装了一个还未完成的并发操作，但最终会产生一个结果值。then 方法允许我们提供一个代表最终值的一种类型的 promise 对象，并产生一个新的 promise 对象来代表最终值的另一种类型，而不管回调函数返回了什么。

从现有的 promise 中构造新 promise 的能力带来了很大的灵活性，并且具有一些简单但强大的惯用法。例如，可以非常容易地构造一个实用程序来拼接多个 promise 的结果。

```
var filesP = join(downloadP("file1.txt"),
                  downloadP("file2.txt"),
                  downloadP("file3.txt"));

filesP.then(function(files) {
    console.log("file1: " + files[0]);
    console.log("file2: " + files[1]);
    console.log("file3: " + files[2]);
});
```

promise 库也经常提供一个叫做 when 的工具函数，其使用也比较类似。

```
var fileP1 = downloadP("file1.txt"),
    fileP2 = downloadP("file2.txt"),
    fileP3 = downloadP("file3.txt");

when([fileP1, fileP2, fileP3], function(files) {
    console.log("file1: " + files[0]);
    console.log("file2: " + files[1]);
    console.log("file3: " + files[2]);
});
```

使 promise 成为卓越的抽象层级的部分原因是通过 then 方法的返回值来联系结果，或者通过工具函数如 join 来构成 promise，而不是在并行的回调函数间共享数据结构。这本质上是安全的，因为它避免了第 66 条中讨论过的数据竞争。即使最小心谨慎的程序员也可能会在保存异步操作的结果到共享的变量或数据结构时犯下简单的错误。

```
var file1, file2;

downloadAsync("file1.txt", function(file) {
    file1 = file;
});

downloadAsync("file2.txt", function(file) {
    file1 = file; // wrong variable
});
```

promise 避免了这种 Bug，因为简单风格的组合 promise 避免了修改共享数据。

注意异步逻辑的有序链事实上也可用有序的 promise，而不是在第 62 条中展现的笨重的嵌套模式。更重要的是，错误处理会自动地通过 promise 传播。当你通过 promise 串联异步操作的集合时，你可以为整个序列提供一个简单的 error 回调函数，而不是将 error 回调函数传递给每一步，正如第 63 条中的代码所示。

尽管这样，有时故意创建某些种类的数据竞争是有用的。Promise 为此提供了一个很好的机制。例如，一个应用程序可能需要尝试从多个不同的服务器上同时下载同一份文件，而

选择最先完成的那个文件。select(或 choose) 工具函数接收几个 promise 并产生一个其值是最先完成下载的文件的 promise。换句话说，几个 promise 彼此竞争。

```
var fileP = select(downloadP("http://example1.com/file.txt"),
                   downloadP("http://example2.com/file.txt"),
                   downloadP("http://example3.com/file.txt"));

fileP.then(function(file) {
    console.log("file: " + file);
});
```

select 函数的另一个用途是提供超时来终止长时间的操作。

```
var fileP = select(downloadP("file.txt"), timeoutErrorP(2000));

fileP.then(function(file) {
    console.log("file: " + file);
}, function(error) {
    console.log("I/O error or timeout: " + error);
});
```

在最后的例子中，我们展示了提供 error 回调函数作为第二个参数给 promise 的 then 方法的机制。

提示

- promise 代表最终值，即并行操作完成时最终产生的结果。
- 使用 promise 组合不同的并行操作。
- 使用 promise 模式的 API 避免数据竞争。
- 在要求有意的竞争条件时使用 select(也被称为 choose)。