O'REILLY®

JavaScript语言精粹（修订版）

JavaScript: The Good Parts

Douglas Crockford 著
赵泽欣 鄢学鹃 译

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

内 容 简 介

JavaScript 曾是“世界上最被误解的语言”，因为它担负太多的特性，包括糟糕的交互和失败的设计，但随着 Ajax 的到来，JavaScript“从最受误解的编程语言演变为最流行的语言”，这除了幸运之外，也证明了它其实是一门优秀的语言。Douglas Crockford 在本书中剥开了 JavaScript 沾污的外衣，抽离出一个具有更好可靠性、可读性和可维护性的 JavaScript 子集，让你看到一门优雅的、轻量级的和非常富有表现力的语言。作者从语法、对象、函数、继承、数组、正则表达式、方法、样式和优美的特性这 9 个方面来呈现这门语言真正的精华部分，通过它们完全可以构建出优雅高效的代码。作者还通过附录列出了这门语言的毒瘤和糟粕部分，且告诉你如何避免它们。最后还介绍了 JSLint，通过它的检验，能有效地保障我们的代码品质。

这是一本介绍 JavaScript 语言本质的权威书籍，值得任何正在或准备从事 JavaScript 开发的人阅读，并且需要反复阅读。学习、理解、实践大师的思想，我们才可能站在巨人的肩上，才有机会超越大师，这本书就是开始。

版权贸易合同登记号 图字：01-2009-0879

图书在版编目（CIP）数据

JavaScript 语言精粹 /（美）克罗克福德（Crockford,D.）著；赵泽欣，鄢学鹃译. —修订本. —北京：电子工业出版社，2012.9
书名原文：JavaScript: The Good Parts
ISBN 978-7-121-17740-8

Ⅰ. ①J… Ⅱ. ①克… ②赵… ③鄢… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 171680 号

策划编辑：张春雨
责任编辑：付 睿
封面设计：Karen Montgomery
印 刷：
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×980 1/16 印张：11 字数：258 千字
印 次：2012 年 9 月第 1 次印刷
定 价：49.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

致敬

给伙计们：Clement、Philbert、Seymore、Stern 和不可遗忘的 C. Twildo。

目录

Table of Contents

前言 xv

第 1 章　精华 1

为什么要使用 JavaScript 2

分析 JavaScript 2

一个简单的试验场 4

第 2 章　语法 5

空白 5

标识符 6

数字 7

字符串 8

语句 10

表达式 15

字面量 18

函数 19

第 3 章　对象 20

对象字面量 20

检索 21

更新 22

引用 22

原型 22

反射 23

枚举 24

删除 24

减少全局变量污染 25

第 4 章　函数 26

函数对象 26

函数字面量……27
调用……27
参数……30
返回……31
异常……31
扩充类型的功能……32
递归……33
作用域……36
闭包……36
回调……39
模块……40
级联……42
柯里化……43
记忆……43
第 5 章　继承……46
伪类……46
对象说明符……49
原型……50
函数化……51
部件……55
第 6 章　数组……57
数组字面量……57
长度……58
删除……59
枚举……59
容易混淆的地方……60
方法……60
指定初始值……62
第 7 章　正则表达式……64
一个例子……65
结构……69
元素……71
第 8 章　方法……77
Array……77
Function……83
Number……84

Object ····· 85
RegExp ····· 86
String ····· 88
第 9 章　代码风格 ····· 94
第 10 章　优美的特性 ····· 98
附录 A　毒瘤 ····· 101
附录 B　糟粕 ····· 109
附录 C　JSLint ····· 115
附录 D　语法图 ····· 127
附录 E　JSON ····· 138
索引 ····· 149

前言
Preface

要是有所得罪请原谅。本是出自一番好意，
只是想显点粗浅技艺，那才是我们的初衷。
——威廉·莎士比亚，《仲夏夜之梦》（*A Midsummer Night's Dream*）

这是一本关于 JavaScript 编程语言的书。它的读者是那些因为偶然事件或好奇心驱使而首次冒险进入 JavaScript 世界的程序员。它也是为那些有着 JavaScript 入门经验但准备更深入了解这门语言的程序员准备的。JavaScript 是一门强大得令人惊讶的语言。它有时会不按常理出牌，但是作为一门轻量级的语言，它是易于掌握的。

本书的目标是帮助你学习 JavaScript 的编程思想。我将展示这门语言的组成部分，并且让你逐步上手，学会如何组合各个部分。这不是一本参考书，它不会对这门语言和它的怪癖进行全面而详尽的介绍。它不包含你希望知道的一切，那些东西你很容易在网上找到。反之，这本书仅包含那些真正重要的东西。

这本书不是写给初学者的。我希望某天写一本叫《JavaScript：第一阶段》（*JavaScript: The First Parts*）的书，但是此书非彼书。这也不是一本关于 Ajax 或 Web 编程的书。本书关注的就是 JavaScript，它只是 Web 开发者必须掌握的语言之一。

这不是一本傻瓜书。这本书虽然薄，但知识点密集，它包括了大量的内容。如果为了理解它而不得不反复阅读，请别沮丧，你的付出将会有所回报。

本书的约定
Conventions Used in This Book

本书使用下列排版约定。

斜体（*Italic*）

表示专业词汇、链接（URL）、文件名和文件扩展名。

等宽字体（`Constant width`）

表示广义上的计算机编码。它们包括命令、配置、变量、属性、键、请求、函数、方

法、类型、类、模块、属性、参数、值、对象、事件、事件处理程序、 XML 与 XHTML 标签、宏和关键字。

等宽粗体（`Constant width bold`）

表示应该由用户按照字面输入的命令或其他文本。

中文版书中切口处的“  ”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引所列页码为原英文版页码。

代码用例
Using Code Examples

这本书是为了帮助你做好工作。一般来说，你可以在程序和文档中使用本书中的代码。你无须联系我们获取许可。例如，使用来自本书的几段代码写一个程序是不需要许可的。出售和散布 O'Reilly 书中用例的光盘（CD-ROM）是需要许可的。通过引用本书和用例代码来回答问题是不需要许可的。把本书中大量的用例代码并入到你的产品文档中是需要许可的。

我们赞赏但不强求注明信息来源。一条信息来源通常包括标题、作者、出版者和国际标准书号（ISBN）。例如：“*JavaScript: The Good Parts* by Douglas Crockford. Copyright 2008 Yahoo! Inc., 978-0-596-51774-8.”。

如果你感到对示例代码的使用超出了正当引用或这里给出的许可范围，请随时通过 *permissions@oreilly.com* 联系我们。

Safari®在线图书

如果你在你最喜爱的技术图书的封面上看到 Safari®联机丛书图标，那意味着此书也可以通过 O'Reilly Network Safari Bookshelf 在线获取。

Safari 提供了比电子书更好的解决方案。它是一个虚拟图书馆，让您可以轻松搜寻成千上万的顶尖技术书籍、剪切和粘贴代码样本、下载某些章节、在你需要最准确和即时的信息时快速找到答案。免费试用请访问 *http://safari.oreilly.com*。

如何联系我们
How to Contact Us

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol , CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们将为本书提供主页，在其中提供勘误表、示例及其他附加信息。读者可从如下网址访问：

http://www.oreilly.com/catalog/9780596517748

如果你想就本书发表评论或提问技术问题，请发送 E-mail 至：

bookquestions@oreilly.com

关于我们的书籍、会议、资源中心和 O'Reilly 网络的更多信息请登录我们的网址：

http://www. oreilly.com/

致谢

Acknowledgments

感谢那些指出我的很多严重错误的审稿者。在生活中，还有什么比有真正聪明的人指出你的过失更好的事情呢？更好的是他们赶在书出版之前做了这个事情。谢谢你们，Steve Souders、Bill Scott、Julien Lecomte、Stoyan Stefanov、Eric Miraglia 和 Elliotte Rusty Harold。

谢谢那些同我一起在 Electric Communities 和 State Software 工作并帮助我发现这门语言实际上有很多精华的人们，特别是 Chip Morningstar、Randy Farmer、John La、Mark Miller、Scott Shattuck 和 Bill Edney。

谢谢雅虎公司（Yahoo! Inc.），因为它给我时间去从事这个项目，并让我在一个如此之棒的地方工作，谢谢过去和现在在 Ajax Strike Force 的所有成员。我也要谢谢 O'Reilly Media, Inc.，尤其是使事情变得如此顺利的 Mary Treseler、Simon St.Laurent 和 Sumita Mukherji。

特别感谢 Lisa Drake 教授所做的所有事情。同时，我要谢谢那些一直为使 ECMAScript 成为一门更好的语言而奋斗的 ECMA TC39 [译注1] 中的伙计们。

最后，谢谢 Brendan Eich，这位世界上最被误解的编程语言[译注2]的设计者，没有他，这本书也就没有必要了。

译注 1：TC39 是研究 JavaScript 语言进化的技术委员会的名字。详情见 *http://www.ecma-international.org/memento/TC39.htm*。

译注 2：本书作者曾写过一篇文章 *JavaScript:The World's Most Misunderstood Programming Language*（*http://javascript.crockford.com/javascript.html*）。在 2008 年的 3 月 3 日他又写了一篇 *The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language*（*http://javascript.crockford.com/popular.html*）。

第1章

精华

Good Parts

……我不过略有一些讨人喜欢的地方而已，怎么会有什么迷人的魔力？
——威廉·莎士比亚，《温莎的风流娘儿们》（*The Merry Wives of Windsor*）

当我还是一个初出茅庐的程序员时，我想掌握自己所用语言的每个特性。我写程序时会尝试使用所有的特性。我认为这是炫耀的好方法，而我也的确出了不少风头，因为我对各个特性了如指掌，谁有问题我都能解答。

最终，我认定这些特性中有一部分特性带来的麻烦远远超出它们的价值。其中，一些特性因为规范很不完善而可能导致可移植性问题，一些特性会导致代码难以阅读或修改，一些特性诱使我追求奇技淫巧但却易于出错，还有一些特性就是设计错误。有时候语言的设计者也会犯错。

大多数编程语言都有精华和糟粕。我发现如果取其精华而弃其糟粕的话，我可以成为一个更好的程序员。毕竟，用坏材料怎么能做出好东西呢？

标准委员会想要移除一门语言中的缺陷部分，这几乎是不可能的，因为这样做会损害所有依赖于那些糟粕的蹩脚的程序。除了在已存在的一大堆缺陷上堆积更多的特性，他们通常无能为力。而且新旧特性并不总是能和谐共处，从而可能产生出更多的糟粕。

但是，你有权力定义你自己的子集。你完全可以基于精华的那部分去编写更好的程序。

JavaScript 中糟粕的比重超出了预料。它一诞生，就在短到令人吃惊的时间里被全世界所接受。它从来没有在实验室里被试用和打磨。当它还非常粗糙时，它就被直接集成到网景的 Navigator 2 浏览器中。随着 Java™的小应用程序（Java™ applets）的失败，JavaScript 变成了默认的“Web 语言”。作为一门编程语言，JavaScript 的流行几乎完全不受它的质量的影响。

好在 JavaScript 有一些非常精华的部分。在 JavaScript 中，美丽的、优雅的、富有表现力的语言特性就像一堆珍珠和鱼目混杂在一起。JavaScript 最本质的部分被深深地隐藏着，以至于多年来对它的主流观点是：JavaScript 就是一个丑陋的、没用的玩具。本书的目的就是要揭示 JavaScript 中的精华，让大家知道它是一门杰出的动态编程语言。JavaScript 就像一块大理石，我要剥落那些不好的特性直到这门语言的真实本质自我显露出来。我相信我精雕细琢出来的优雅子集大大地优于这门语言的整体，它更可靠、更易读、更易于维护。

这本书不打算全面描述这门语言。反之，它将专注在精华部分，同时会偶尔警告要去避免糟粕部分。这里将被描述的子集可以用来构造可靠的、易读的程序，无论规模大小。通过仅专注于精华部分，我们就可以缩短学习时间，增强健壮性，并且还能拯救一些树木[译注1]。

或许，只学习精华的最大好处就是你可以不必头痛如何忘记糟粕。忘掉不好的模式是非常困难的。这是一个非常痛苦的工作，我们中的大多数人都会很不愿意面对。有时候，制定语言的子集是为了让学生更好地学习。但在这里，我制定的 JavaScript 子集是为了让专业人员更好地工作。

为什么要使用 JavaScript
Why JavaScript?

JavaScript 是一门重要的语言，因为它是 Web 浏览器的语言。它与浏览器的结合使它成为世界上最流行的编程语言之一。同时，它也是世界上最被轻视的编程语言之一。浏览器的 API 和文档对象模型（DOM）相当糟糕，连累 JavaScript 遭到不公平的指责。在任何语言中处理 DOM 都是一件痛苦的事情，它的规范制定得很拙劣并且实现互不一致。本书很少涉及 DOM，我认为写一本关于 DOM 的精华的书就像执行一项不可能完成的任务。

JavaScript 是最被轻视的语言，因为它不是所谓的主流语言（SOME OTHER LANGUAGE）[译注2]。如果你擅长某些主流语言，但却在一个只支持 JavaScript 的环境中编程，那么被迫使用 JavaScript 的确是相当令人厌烦的。在这种情形下，大多数人觉得没必要先去学好 JavaScript，但结果他们会惊讶地发现，JavaScript 跟他们宁愿使用的主流语言有很大不同，而且这些不同至为关键。

JavaScript 令人惊异的事情是，在对这门语言没有太多了解，甚至对编程都没有太多了解的情况下，你也能用它来完成工作。它是一门拥有极强表达能力的语言。当你知道要做什么时，它还能表现得更好。编程是很困难的事情。绝不应该在懵懵懂懂的状态下开始你的工作。

3 分析 JavaScript
Analyzing JavaScript

JavaScript 建立在一些非常优秀的想法和少数非常糟糕的想法之上。

那些优秀的想法包括函数、弱类型、动态对象和富有表现力的对象字面量表示法。那些糟糕的想法包括基于全局变量的编程模型。

译注 1：作者这里幽默地暗示这本书只关注精华部分，所以书变薄了，用的纸张少了，就可以少砍伐一些树木。

译注 2：专指一些主流语言，像 C、C++、Java、Perl、Python 等。

JavaScript 的函数是（主要）基于词法作用域（lexical scoping）[译注3]的顶级对象。JavaScript 是第一个成为主流的 Lambda[译注4]语言。实际上，相对于 Java 而言，JavaScript 与 Lisp[译注5]和 Scheme[译注6]有更多的共同点。它是披着 C 外衣的 Lisp。这使得 JavaScript 成为一个非常强大的语言。

现今大部分编程语言中都流行要求强类型。其原理在于强类型允许编译器在编译时检测错误。我们能越早检测和修复错误，付出的代价就越小。JavaScript 是一门弱类型的语言，所以 JavaScript 编译器不能检测出类型错误。这可能让从强类型语言转向 JavaScript 的开发人员感到恐慌。但事实证明，强类型并不会让你的测试工作变得轻松。而且我在工作中发现，强类型检查找到的错误并不是令我头痛的错误。另一方面，我发现弱类型是自由的。我无须建立复杂的类层次，我永远不用做强制造型，也不用疲于应付类型系统以得到想要的行为。

JavaScript 有非常强大的对象字面量表示法。通过列出对象的组成部分，它们就能简单地被创建出来。这种表示法是 JSON 的灵感来源，它现在已经成为流行的数据交换格式[译注7]。（附录 E 中将会有更多关于 JSON 的内容。）

原型继承是 JavaScript 中一个有争议的特性。JavaScript 有一个无类型的（class-free）对象系统，在这个系统中，对象直接从其他对象继承属性。这真的很强大，但是对那些被训练使用类去创建对象的程序员们来说，原型继承是一个陌生的概念。如果你尝试对 JavaScript 直接应用基于类的设计模式，你将会遭受挫折。但是，如果你学会了自如地使用 JavaScript 原型，你的努力将会有所回报。

JavaScript 在关键思想的选择上饱受非议。虽然在大多数情况下，这些选择是合适的。但是有一个选择相当糟糕：JavaScript 依赖于全局变量来进行连接。所有编译单元的所有顶级变量被撮合到一个被称为全局对象（*the global object*）的公共命名空间中。这是一件糟糕的事情，因为全局变量是魔鬼，但它们在 JavaScript 中却是基础。幸好，我们接下来会看到，JavaScript 也给我们提供了缓解这个问题的处理方法。

在某些情况下，我们可能无法忽略糟粕，还有一些毒瘤难以避免，当涉及这些部分时，我会将它们指出来。它们也被总结在附录 A 中。但是我们将成功地避免本书中提到的大多数糟粕，它们中的大部分被总结在附录 B 中。如果你想学习那些糟粕，以及如何拙劣地使用

译注 3：JavaScript 中的函数是根据词法来划分作用域的，而不是动态划分作用域的。具体内容参见《JavaScript 权威指南》中译第 5 版相关章节——“8.8.1 词法作用域”。

译注 4：Lambda 演算是一套用于研究函数定义、函数应用和递归的形式系统。它对函数式编程有巨大的影响，比如 Lisp 语言、ML 语言和 Haskell 语言。更多详细内容请参见 *http://zh.wikipedia.org/wiki/λ 演算*。

译注 5：Lisp（全名 LISt Processor，即链表处理语言），是由约翰·麦卡锡在 1960 年左右创造的一种基于λ演算的函数式编程语言。更多详细内容请参见 *http://zh.wikipedia.org/wiki/Lisp*。

译注 6：Scheme，一种多范型的编程语言，它是两种 Lisp 主要的方言之一。更多详细内容请见 *http://zh.wikipedia.org/wiki/Scheme*。

译注 7：本书作者也是 JSON（JavaScript Object Notation）的创立者。官方网站中文版网址是 *http:// json.org/json-zh.html*。

它们，请参阅任何其他的 JavaScript 书籍。

4 《ECMAScript 编程语言》第 3 版定义了 JavaScript(又称 JScript)的标准，它可以从 *http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf* 获得。本书所描述的是 ECMAScript 的一个特定的子集。本书并不描述整个语言，因为它排除了糟粕的部分。这里排除得也许并不彻底，回避了一些极端情况。你也应该这样，走极端只会带来风险和苦恼。

附录 C 描述了一个叫 JSLint 的编程工具，它是一个 JavaScript 解析器，它能分析 JavaScript 问题并报告它包含的缺点。JSLint 提出了比一般的 JavaScript 开发更严格的要求。它能让你确信你的程序只包含精华部分。

JavaScript 是一门反差鲜明的语言。它包含很多错误，而且多刺，所以你可能会疑惑："为什么我要使用 JavaScript？"有两个答案。第一个是你没有选择。Web 已变成一个重要的应用开发平台，而 JavaScript 是唯一一门所有浏览器都可以识别的语言。很不幸，Java 在浏览器环境中失败了，否则想用强类型语言的人就有其他选择。但是 Java 确实失败了，而 JavaScript 仍在蓬勃发展，这恰恰说明 JavaScript 确有其过人之处。

另一个答案是，尽管 JavaScript 有缺陷，但是它真的很优秀。它既轻量级又富有表现力。并且一旦你熟练掌握了它，就会发现函数式编程是一件很有趣的事。

但是为了更好地使用这门语言，你必须知道它的局限。我将会无情地揭示它们，不要因此而气馁。这门语言的精华足以弥补它的糟粕。

一个简单的试验场
A Simple Testing Ground

如果你有一个 Web 浏览器和任意一个文本编辑器，那么你就有了运行 JavaScript 程序所需要的一切。首先，请创建一个 HTML 文件，起个名字，比如 *program.html*：

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

接下来，在同一个文件夹内，创建一个脚本文件，比如起名为 *program.js*：

```
document.writeln('Hello, world!');
```

下一步，用你的浏览器打开你的 HTML 文件查看结果。本书贯彻始终都会用到一个 `method` 方法去定义新方法。下面是它的定义：

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

我会在第 4 章解释它。

第2章 语法

Grammar

我很熟悉它，早就在文法书上念过了。
——威廉·莎士比亚，《泰特斯·安德洛尼克斯》（*The Tragedy of Titus Andronicus*）

本章介绍 JavaScript 的精华部分的语法，并简要地概述其语言结构。我们将用铁路图（railroad diagram）[译注1] 来表示该语法。

理解这些图的规则很简单。

- 从左边界开始，沿着轨道到右边界。
- 沿途，你在圆框中遇到的是字面量，在方块中遇到的是规则或描述。
- 任何沿着轨道能走通的序列都是合法的。
- 任何不能沿着轨道走通的序列都是非法的。
- 末端只有一个竖条的铁路图，表示允许在任意一对符号中间插入空白。而在末端有两个竖条的铁路图则不允许。

在本章中所展示的精华部分的语法明显比整个语言的语法简单得多。

空白

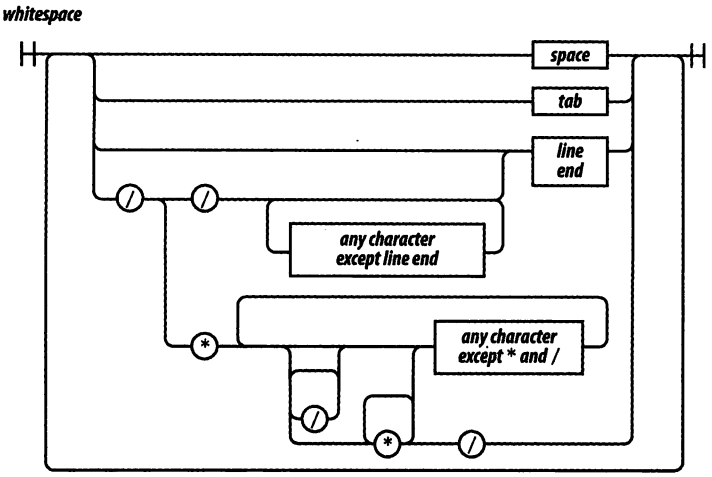
Whitespace

空白可能表现为被格式化的字符或注释的形式。空白通常没有意义，但是有时候必须要用它来分隔字符序列，否则它们就会被合并成一个符号。例如，对如下代码来说：

```
var that = this;
```

`var` 和 `that` 之间的空格是不能移除的，但是其他的空格都可以移除。

译注 1：铁路图，又叫语法图（syntax diagrams），是一种表示形式语法的方式，是巴科斯范式和扩展巴科斯范式的图形化表示。更多详细内容请见 *http://en.wikipedia.org/wiki/Syntax_diagram*。



JavaScript 提供两种注释形式，一种是用 /* */ 包围的块注释，另一种是以//为开头的行注释。注释应该被优先用来提高程序的可读性。请注意，注释一定要精确地描述代码。没有用的注释比没有注释更糟糕。

用 /* */包围的块注释形式来自于一门叫 PL/I[译注2] 的语言。PL/I 选择那些不常见的符号对作为注释的符号标志，因为除了可能出现在字符串字面量里之外，它们不大可能在这门语言的程序中出现。在 JavaScript 中，那些字符对也可能出现在正则表达式字面量里，所以块注释对于被注释的代码块来说是不安全的。例如：

```
/*
    var rm_a = /a*/.match(s);
*/
```

上面的注释导致了一个语法错误。所以，我建议避免使用 /* */注释，而用//注释代替它。在本书中，只会使用//注释。

标识符

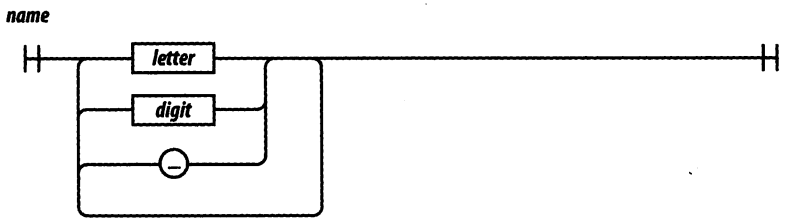
Names

标识符由一个字母开头，其后可选择性地加上一个或多个字母、数字或下画线[译注3]。标识符不能使用下面这些保留字：

译注 2：PL/I，是 Programming Language One 的简写。当中的“I”其实是罗马数字的“一”。它是一种 IBM 公司在 19 世纪 50 年代发明的第三代高级编程语言，有些类似 PASCAL 语言。更多详细内容请见 *http://zh.wikipedia.org/wiki/PL/I*。

译注 3：JavaScript 规范中，标识符除字符外，还允许以下画线（_）和美元符（$）开头。但本书作者描述的是他认为的 JavaScript 精华应该遵循的规范，含有作者个人主观品味。关于 JavaScript 标识符的更多内容，请参考《JavaScript 权威指南》语言核心部分。

```
abstract
boolean break byte
case catch char class const continue
debugger default delete do double
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while with
```

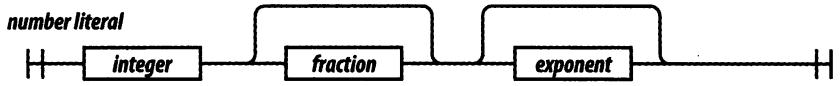



在这个列表中的大部分保留字尚未用在这门语言中。这个列表不包括一些本应该被保留而没有保留的字，诸如 `undefined`、`NaN` 和 `Infinity`。JavaScript 不允许使用保留字来命名变量或参数。更糟糕的是，JavaScript 不允许在对象字面量中，或者用点运算符提取对象属性时，使用保留字作为对象的属性名。

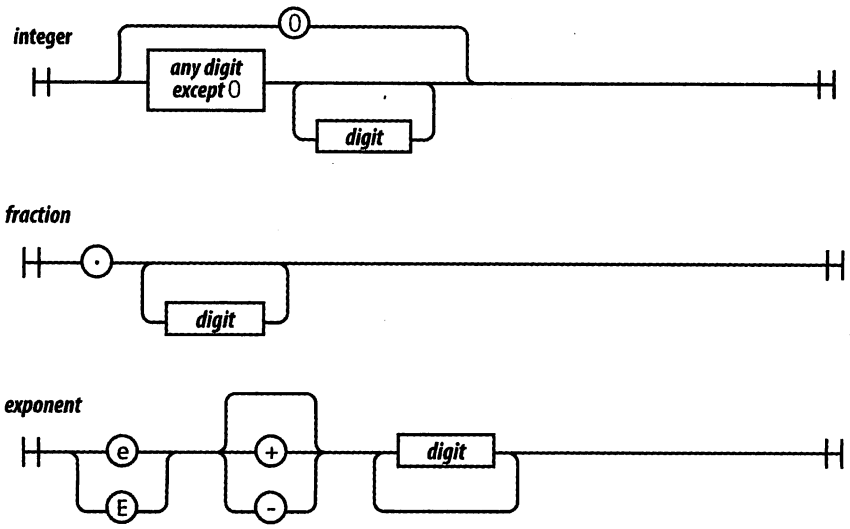
标识符被用于语句、变量、参数、属性名、运算符和标记。

数字

Numbers



JavaScript 只有一个数字类型。它在内部被表示为 64 位的浮点数，和 Java 的 `double` 数字类型一样。与其他大多数编程语言不同的是，它没有分离出整数类型，所以 `1` 和 `1.0` 的值相同。这提供了很大的方便，因为它完全避免了短整型的溢出问题，你只需要知道它是一种数字。这避免了一大堆因数字类型导致的错误。



如果一个数字字面量有指数部分，那么这个字面量的值等于 e 之前的数字与 10 的 e 之后数字的次方相乘。所以 100 和 1e2 是相同的数字。

负数可以用前置运算符 - 加数字构成。

NaN 是一个数值，它表示一个不能产生正常结果的运算结果。NaN 不等于任何值，包括它自己。你可以用函数 isNaN(*number*) 检测 NaN。

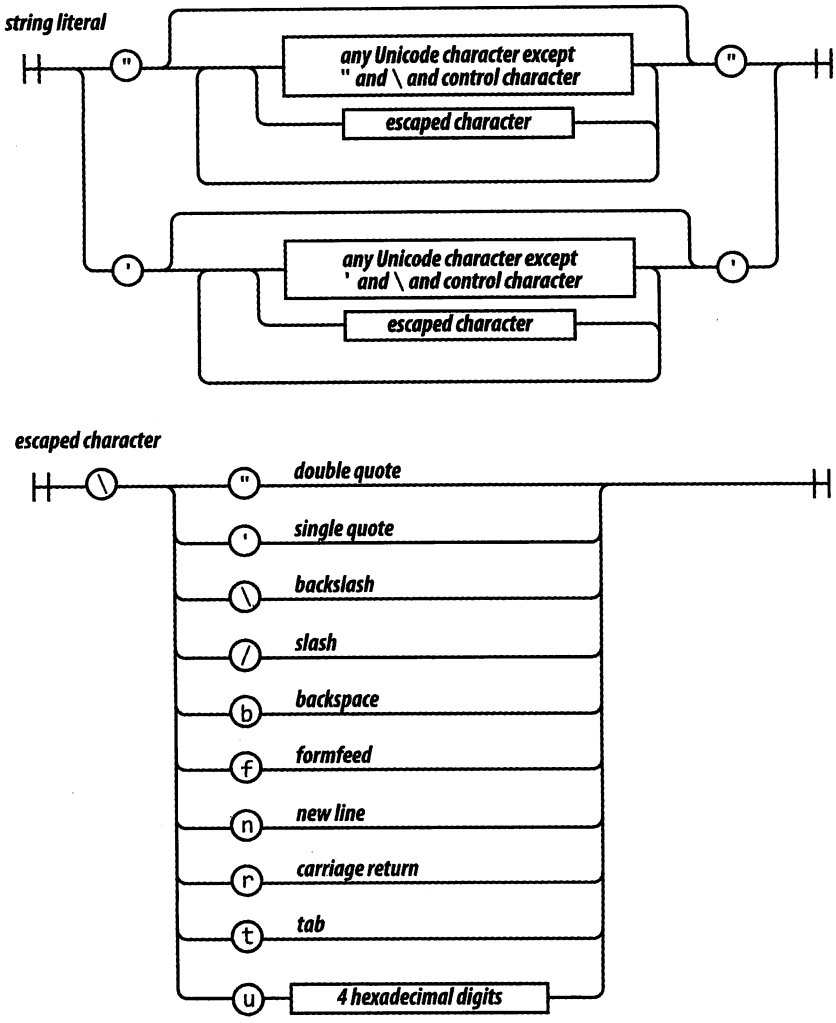
Infinity 表示所有大于 1.79769313486231570e+308 的值。

数字拥有方法（参见第 8 章）。JavaScript 有一个对象 Math，它包含一套作用于数字的方法。例如，可以用 Math.floor(*number*) 方法把一个数字转换成一个整数。

字符串
Strings

字符串字面量可以被包在一对单引号或双引号中，它可能包含 0 个或多个字符。\（反斜线符号）是转义字符。JavaScript 在被创建的时候，Unicode 是一个 16 位的字符集，所以 JavaScript 中的所有字符都是 16 位的。

JavaScript 没有字符类型。要表示一个字符，只需创建仅包含一个字符的字符串即可。



转义字符用来把那些正常情况下不被允许的字符插入到字符串中，比如反斜线、引号和控制字符。`\u` 约定用来指定数字字符编码。

```
"A" === "\u0041"
```

字符串有一个 `length` 属性。例如，`"seven".length` 是 5。

字符串是不可变的。一旦字符串被创建，就永远无法改变它。但你可以很容易地通过 `+` 运算符连接其他字符串来创建一个新字符串。两个包含着完全相同的字符且字符顺序也相同的字符串被认为是相同的字符串。所以：

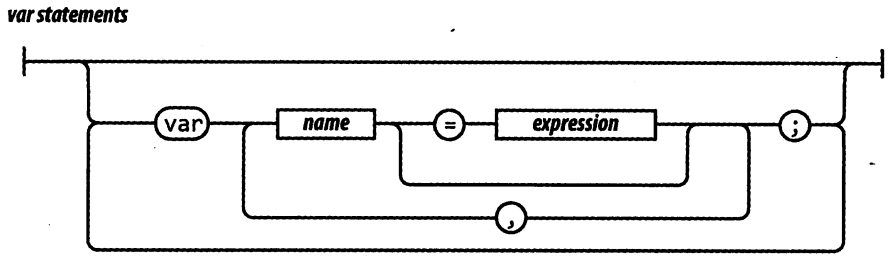

```
'c' + 'a' + 't' === 'cat'
```

是 true。

字符串有一些方法（参见第 8 章）：

```
'cat'.toUpperCase( ) === 'CAT'
```

语句
Statements



一个编译单元包含一组可执行的语句。在 Web 浏览器中，每个<script>标签提供一个被编译且立即执行的编译单元。因为缺少链接器[译注 4]，JavaScript 把它们一起抛到一个公共的全局名字空间中。附录 A 有更多关于全局变量的内容。

当 var 语句被用在函数内部时，它定义的是这个函数的私有变量。

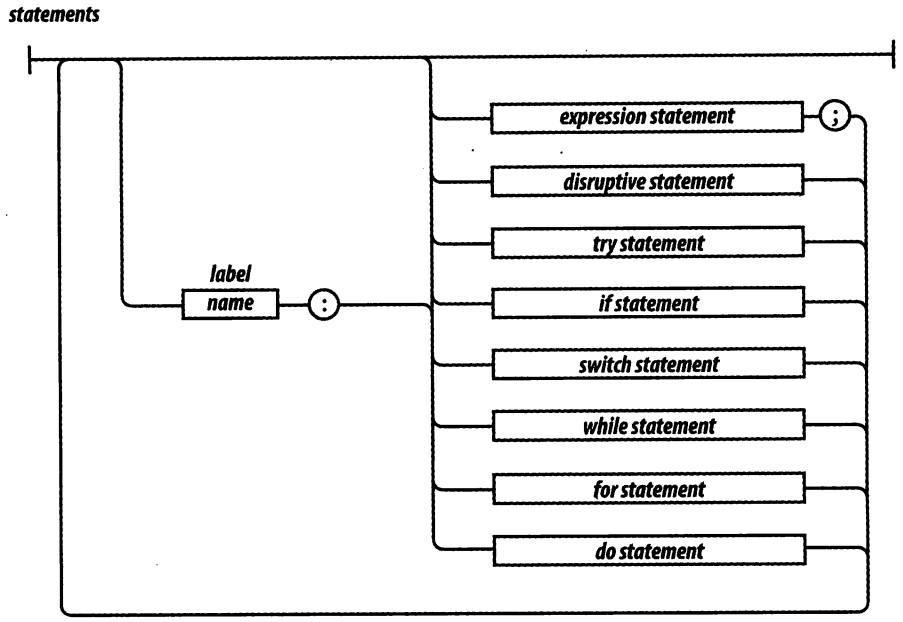
switch、while、for 和 do 语句允许有一个可选的前置标签（*label*），它配合 break 语句来使用。

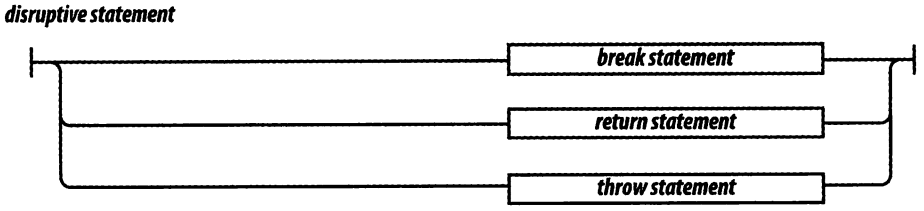
语句通常按照从上到下的顺序被执行。JavaScript 可以通过条件语句（if 和 switch）、循环语句（while、for 和 do）、强制跳转语句（break、return 和 throw）和函数调用来改变执行序列。

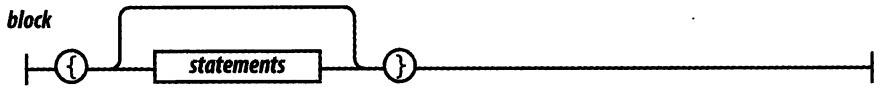
代码块是包在一对花括号中的一组语句。不像许多其他语言，JavaScript 中的代码块不会创建新的作用域，因此变量应该被定义在函数的头部，而不是在代码块中。

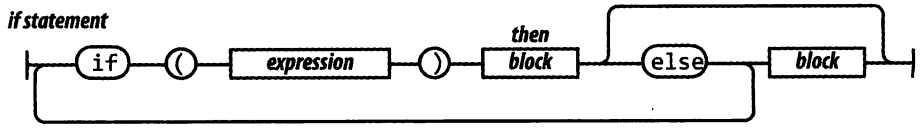
if 语句根据表达式的值改变程序流程。表达式的值为真时执行跟在其后的代码块，否则，执行可选的 else 分支。

译注 4：链接器（Linker）是编程语言或操作系统提供的工具，它的工作就是解析未定义的符号引用，将目标文件中的占位符替换为符号地址。具体参见 *http://zh.wikipedia.org/wiki/链接器*。







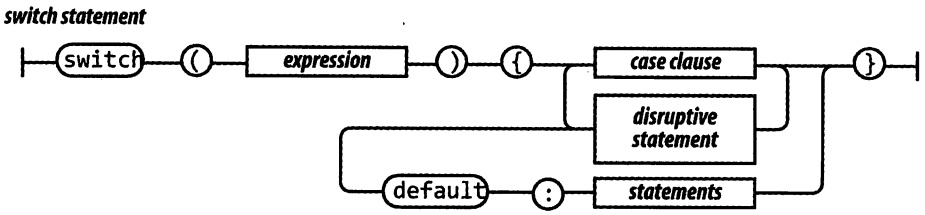


下面列出的值被当做假（*falsy*）：

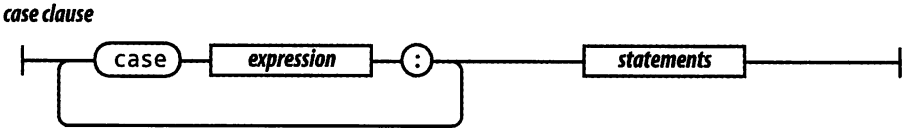
- `false`
- `null`

- undefined
- 空字符串''
- 数字 0
- 数字 NaN

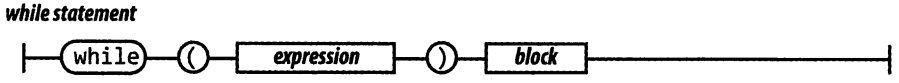
其他所有的值都被当做真，包括 true、字符串"false"，以及所有的对象。



switch 语句执行一个多路分支。它把其表达式的值和所有指定的 case 条件进行匹配。其表达式可能产生一个数字或字符串。当找到一个精确的匹配时，执行匹配的 case 从句中的语句。如果没有找到任何匹配，则执行可选的 default 语句。



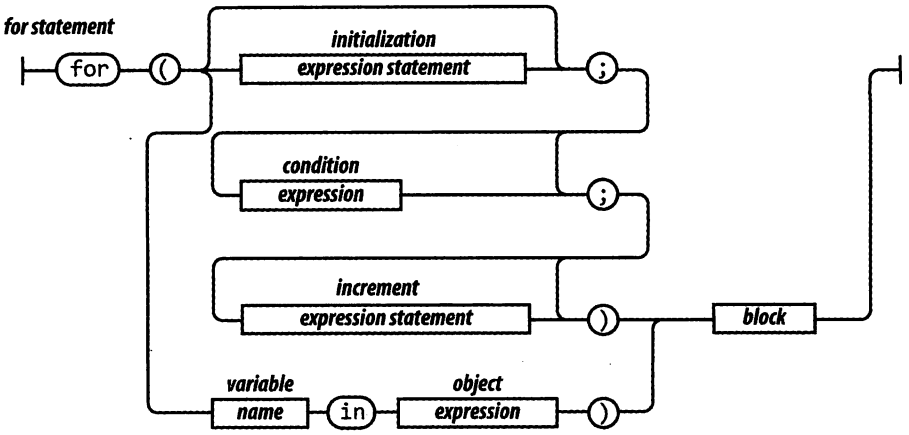
一个 case 从句包含一个或多个 case 表达式。case 表达式不一定必须是常量。要防止继续执行下一个 case，case 从句后应该跟随一个强制跳转语句。你可以用 break 语句退出 switch 语句。



while 语句执行一个简单的循环。如果表达式值为假，就终止循环。而当表达式值为真时，代码块被执行。

for 语句是一个结构更复杂的循环语句。它有两种形式。

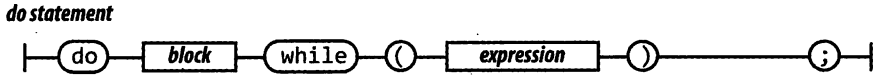
常见的形式由 3 个可选从句控制：初始化从句（*initialization*）、条件从句（*condition*）和增量从句（*increment*）。首先，执行 *condition*，它的作用通常是初始化循环变量。接着，计算 *condition* 的值。典型的情况是它根据一个完成条件检测循环变量。如果 *condition* 被省略掉，则假定返回的条件是真。如果 *condition* 的值为假，那么循环将终止。否则，执行代码块，然后执行 *increment*，接着循环会重复执行 *condition*。



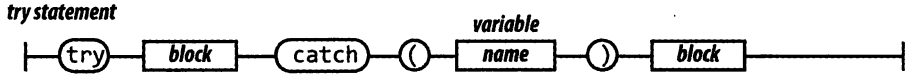
另一种形式（被称为 `for in` 语句）会枚举一个对象的所有属性名（或键名）。在每次循环中，*object* 的下一个属性名字符串被赋值给 *variable*。

通常你需要检测 *object*`.hasOwnProperty(`*variable*`)` 来确定这个属性名是该对象的成员，还是来自于原型链。

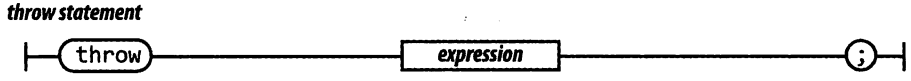
```
for (myvar in obj) {
   if (obj.hasOwnProperty(myvar)) {
      ...
   }
}
```



`do` 语句就像 `while` 语句，唯一的区别是它在代码块执行之后而不是之前检测表达式的值。这就意味着代码块至少要执行一次。

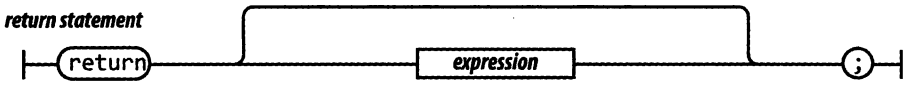


`try` 语句执行一个代码块，并捕获该代码块抛出的任何异常。`catch` 从句定义一个新的变量 *variable* 来接收抛出的异常对象。



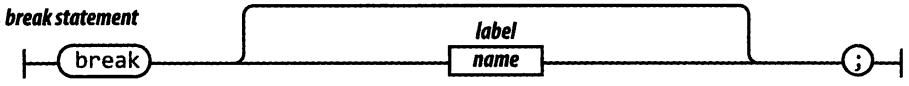
throw 语句抛出一个异常。如果 throw 语句在一个 try 代码块中，那么控制流会跳转到 catch 从句中。如果 throw 语句在函数中，则该函数调用被放弃，控制流跳转到调用该函数的 try 语句的 catch 从句中。

throw语句中的表达式通常是一个对象字面量，它包含一个name属性和一个message属性。异常捕获器可以使用这些信息去决定该做什么。



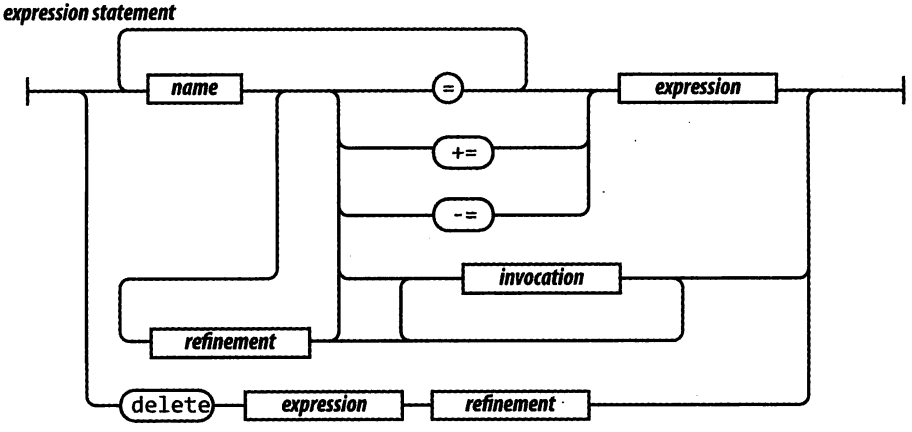
return 语句会导致从函数中提前返回。它也可以指定要被返回的值。如果没有指定返回表达式，那么返回值是 undefined。

JavaScript 不允许在 return 关键字和表达式之间换行。



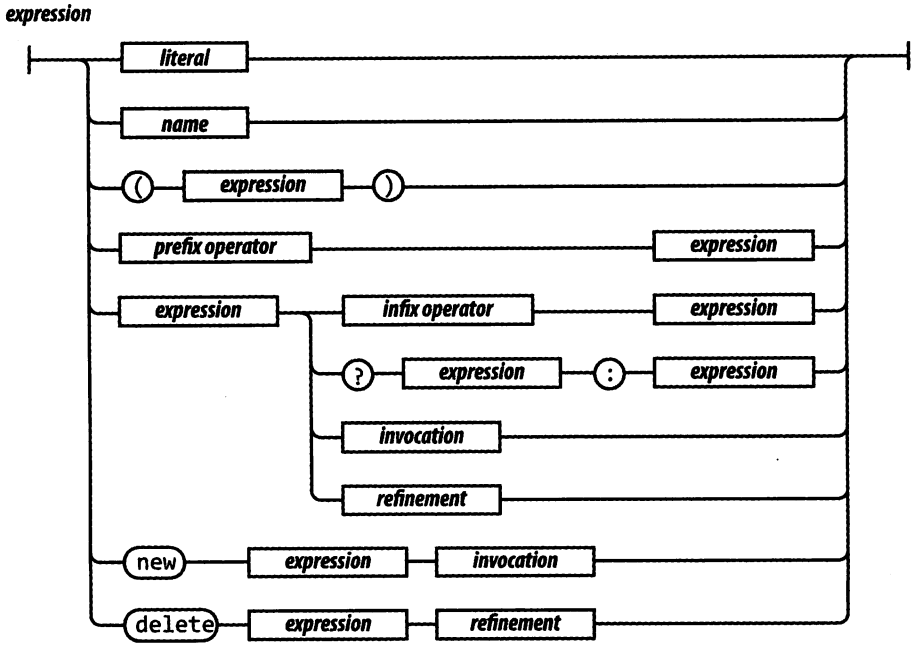
break 语句会使程序退出一个循环语句或 switch 语句。它可以指定一个可选的标签，那退出的就是带该标签的语句。

JavaScript 不允许在 break 关键字和标签之间换行。



15 一个 expression 语句可以给一个或多个变量或成员赋值，或者调用一个方法，或者从对象中删除一个属性。运算符 = 被用于赋值，不要把它和恒等运算符 === 混淆起来。运算符 += 可以用于加法运算或连接字符串。

表达式
Expressions



最简单的表达式是字面量值（比如字符串或数字）、变量、内置的值（`true`、`false`、`null`、`undefined`、`NaN`和`Infinity`）、以`new`开头的调用表达式、以`delete`开头的属性提取表达式、包在圆括号中的表达式、以一个前置运算符作为前导的表达式，或者表达式后面跟着：

- 一个中置运算符与另一个表达式；
- 三元运算符 ? 后面跟着另一个表达式，然后接一个 :，再然后接第 3 个表达式；
- 一个函数调用；
- 一个属性提取表达式。

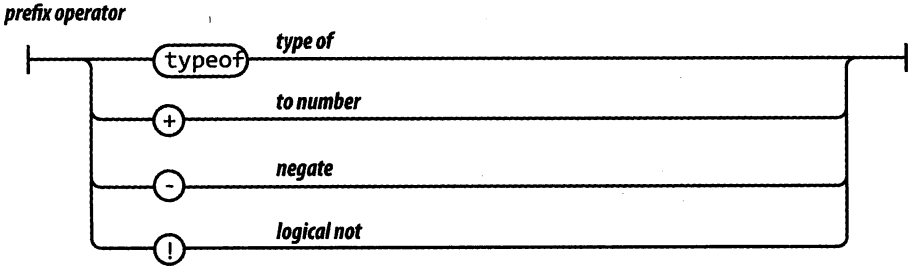
三元运算符 ? 有 3 个运算数。如果第 1 个运算数值为真，产生第 2 个运算数的值。但如果第 1 个运算数值为假，则产生第 3 个运算数的值。

16 在排列运算符优先级的表 2-1 中，排在越上面的运算符优先级越高。它们的结合性[译注5]最强。排在越下面的运算符优先级越低。圆括号可以用来改变正常情况下的优先级，所以：

```
2 + 3 * 5 === 17
(2 + 3) * 5 === 25
```

表 2-1：运算符优先级

`. [] ( )`	提取属性与调用函数
`delete new typeof + - !`	一元运算符
`* / %`	乘法、除法、求余[译注6]
`+ -`	加法/连接、减法
`>= <= > <`	不等式运算符
`=== !==`	等式运算符
`&&`	逻辑与
`||`	逻辑或
`?:`	三元



`typeof` 运算符产生的值有`'number'`、`'string'`、`'boolean'`、`'undefined'`、`'function'`和`'object'`。如果运算数是一个数组或 `null`，那么结果是`'object'`，这其实是不对的。第 6 章和附录 A 将会有更多关于 `typeof` 的内容。

译注 5：在编程语言中，结合性（associativity）是操作符在没有圆括号分组的情况下决定其优先级的一种属性。它可能是从左向右结合（left-associative）、从右向左结合（right-associative）或无结合。比如加运算符的结合性是从左向右，而一元运算符、赋值运算符及三元条件运算符的结合性是从右向左。关于更多的运算符结合性的信息，请参阅 *http://en.wikipedia.org/wiki/Operator_associativity* 或《JavaScript 权威指南》中译第 5 版的章节——“5.2.4 运算符的结合性”。

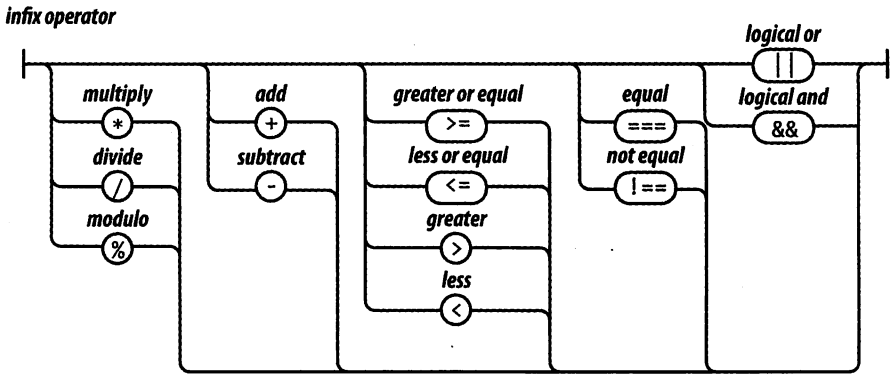
译注 6：在 JavaScript 语言里“%”不是通常数学意义上的模运算，而实际上是“求余”运算。两个运算数都为正数时，求模运算和求余运算的值相同；两个运算数中存在负数时，求模运算和求余运算的值则不相同。求模运算的详细原理请参考 *http://en.wikipedia.org/wiki/Modulo_operation*。你还可以从 StackOverflow 里找到精彩的解答：*http://stackoverflow.com/questions/4467539/javascript-modulo-not-behaving*。

如果 ! 的运算数的值为真，那么产生 false，否则产生 true。

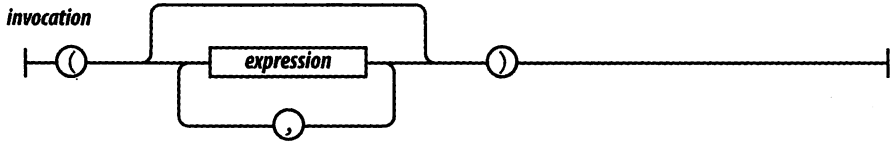
+ 运算符可以进行加法运算或字符串连接。如果你想要的是加法运算，请确保两个运算数都是数字。

/ 运算符可能会产生一个非整数结果，即使两个运算数都是整数。

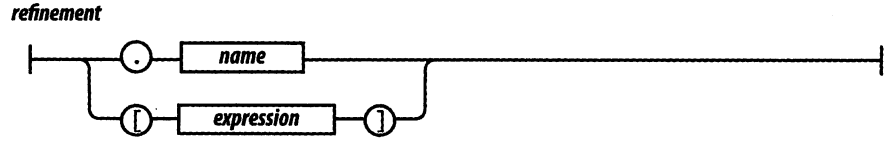
如果第 1 个运算数的值为假，那么运算符 && 产生它的第 1 个运算数的值，否则产生第 2 个运算数的值。



如果第 1 个运算数的值为真，那么运算符 || 产生第 1 个运算数的值，否则产生第 2 个运算数的值。



函数调用引发函数的执行。函数调用运算符是跟随在函数名后面的一对圆括号。圆括号中可能包含传递给这个函数的参数。第 4 章将会有更多关于函数的内容。

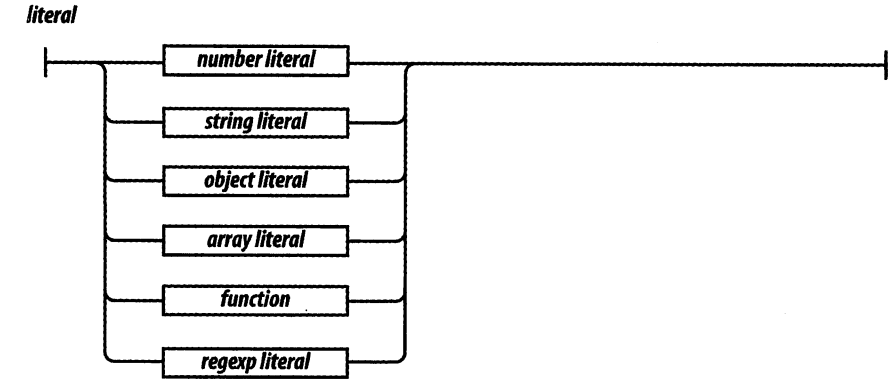


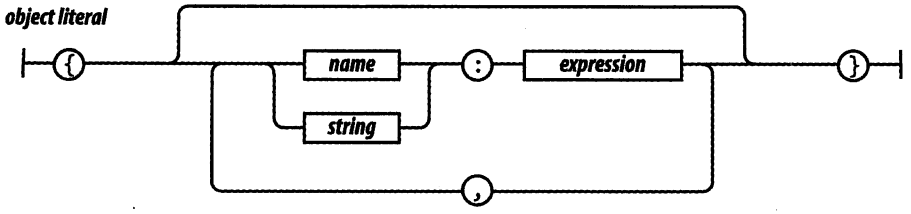
一个属性存取表达式用于指定一个对象或数组的属性或元素。下一章我将详细描述它。

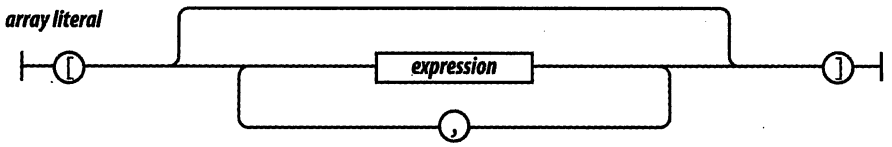
字面量
Literals

对象字面量是一种可以方便地按指定规格创建新对象的表示法。属性名可以是标识符或字符串。这些名字被当做字面量名而不是变量名来对待，所以对象的属性名在编译时才能知道。属性的值就是表达式。下一章会有更多关于对象字面量的内容。

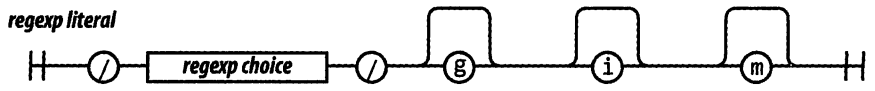
18







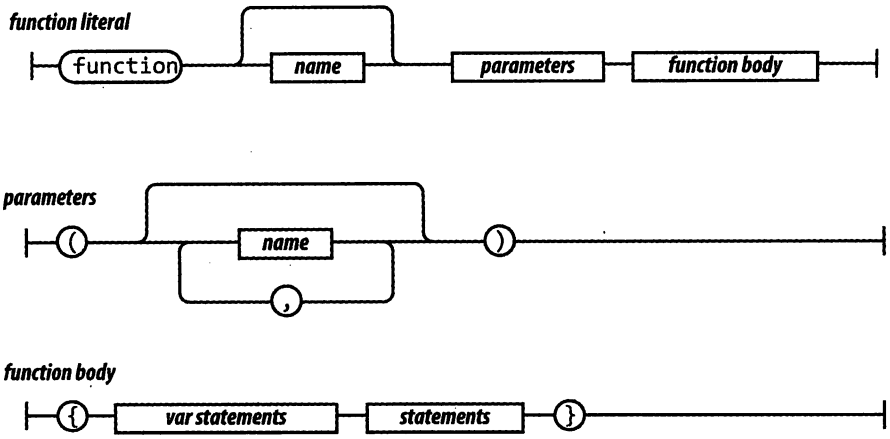
数组字面量是一种可以方便地按指定规格创建新数组的表示法。第 6 章会有更多关于数组字面量的内容。



第 7 章会有更多关于正则表达式的内容。

函数

Functions



函数字面量定义了函数值。它可以有一个可选的名字，用于递归地调用自己。它可以指定一个参数列表，这些参数就像变量一样，在调用时由传递的实际参数（argument）初始化。函数的主体包括变量定义和语句。第 4 章会有更多关于函数的内容。

第 3 章

对象

Objects

对于丑陋的事物，爱会闭目无视。
——威廉·莎士比亚，《维洛那二绅士》（*The Two Gentlemen of Verona*）

JavaScript的简单数据类型包括数字、字符串、布尔值(`true`和`false`)、`null`值和`undefined`值。其他所有的值都是对象。数字、字符串和布尔值“貌似”对象，因为它们拥有方法，但它们是不可变的。JavaScript中的对象是可变的键控集合（keyed collections）。在JavaScript中，数组是对象，函数是对象，正则表达式是对象，当然，对象自然也是对象。

对象是属性的容器，其中每个属性都拥有名字和值。属性的名字可以是包括空字符串在内的任意字符串。属性值可以是除`undefined`值之外的任何值。

JavaScript里的对象是无类型（class-free）的。它对新属性的名字和属性的值没有限制。对象适合用于汇集和管理数据。对象可以包含其他对象，所以它们可以容易地表示成树状或图形结构。

JavaScript包含一种原型链的特性，允许对象继承另一个对象的属性。正确地使用它能减少对象初始化时消耗的时间和内存。

对象字面量

Object Literals

对象字面量提供了一种非常方便地创建新对象值的表示法。一个对象字面量就是包围在一对花括号中的零或多个“名/值”对。对象字面量可以出现在任何允许表达式出现的地方。

```
var empty_object = {};

var stooge = {
    "first-name": "Jerome",
    "last-name": "Howard"
};
```

属性名可以是包括空字符串在内的任何字符串。在对象字面量中，如果属性名是一个合法的 JavaScript 标识符且不是保留字，则并不强制要求用引号括住属性名。所以用引号括住"first-name"是必需的，但是否括住 first_name 则是可选的[译注 1]。逗号用来分隔多个“名/值”对。

属性的值可以从包括另一个对象字面量在内的任意表达式中获得。对象是可嵌套的：

```
var flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

检索

Retrieval

要检索对象里包含的值，可以采用在[]后缀中括住一个字符串表达式的方式。如果字符串表达式是一个字符串字面量，而且它是一个合法的 JavaScript 标识符且不是保留字，那么也可以用 . 表示法代替。优先考虑使用 . 表示法，因为它更紧凑且可读性更好。

```
stooge["first-name"]     // "Jerome"
flight.departure.IATA    // "SYD"
```

如果你尝试检索一个并不存在的成员属性的值，将返回 undefined。

```
stooge["middle-name"]     // undefined
flight.status             // undefined
stooge["FIRST-NAME"]      // undefined
```

|| 运算符可以用来填充默认值：

```
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

尝试从 undefined 的成员属性中取值将会导致 TypeError 异常。这时可以通过 && 运算符来避免错误。

译注 1： JavaScript 的标识符中包含连接符 (-) 是不合法的，但允许包含下画线 (_)。

```
flight.equipment                              // undefined
flight.equipment.model                        // throw "TypeError"
flight.equipment && flight.equipment.model    // undefined
```

22 更新
Update

对象里的值可以通过赋值语句来更新。如果属性名已经存在于对象里，那么这个属性的值就会被替换。

```
stooge['first-name'] = 'Jerome';
```

如果对象之前没有拥有那个属性名，那么该属性就被扩充到对象中。

```
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = {
    model: 'Boeing 777'
};
flight.status = 'overdue';
```

引用
Reference

对象通过引用来传递。它们永远不会被复制：

```
var x = stooge;
x.nickname = 'Curly';
var nick = stooge.nickname;
    // 因为 x 和 stooge 是指向同一个对象的引用，所以 nick 为 'Curly'。

var a = {}, b = {}, c = {};
    // a、b 和 c 每个都引用一个不同的空对象。
a = b = c = {};
    // a、b 和 c 都引用同一个空对象。
```

原型
Prototype

每个对象都连接到一个原型对象，并且它可以从中继承属性。所有通过对象字面量创建的对象都连接到 `Object.prototype`，它是 JavaScript 中的标配对象。

当你创建一个新对象时，你可以选择某个对象作为它的原型。JavaScript 提供的实现机制杂乱而复杂，但其实可以被明显地简化。我们将给 `Object` 增加一个 `create` 方法。这个方法

创建一个使用原对象作为其原型的新对象。下一章将会有更多关于函数的内容。

```
if (typeof Object.beget !== 'function') {
    Object.create = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    };
}
var another_stooge = Object.create(stooge);
```


原型连接在更新时是不起作用的。当我们对某个对象做出改变时，不会触及该对象的原型：

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

原型连接只有在检索值的时候才被用到。如果我们尝试去获取对象的某个属性值，但该对象没有此属性名，那么 JavaScript 会试着从原型对象中获取属性值。如果那个原型对象也没有该属性，那么再从它的原型中寻找，依此类推，直到该过程最后到达终点 `Object.prototype`。如果想要的属性完全不存在于原型链中，那么结果就是 `undefined` 值。这个过程称为*委托*。

原型关系是一种动态的关系。如果我们添加一个新的属性到原型中，该属性会立即对所有基于该原型创建的对象可见。

```
stooge.profession = 'actor';
another_stooge.profession   // 'actor'
```

我们将会在第 6 章中看到更多关于原型链的内容。

反射
Reflection

检查对象并确定对象有什么属性是很容易的事情，只要试着去检索该属性并验证取得的值。`typeof` 操作符对确定属性的类型很有帮助：

```
typeof flight.number          // 'number'
typeof flight.status          // 'string'
typeof flight.arrival         // 'object'
typeof flight.manifest        // 'undefined'
```

请注意原型链中的任何属性都会产生值：

```
typeof flight.toString        // 'function'
typeof flight.constructor     // 'function'
```

有两种方法去处理掉这些不需要的属性。第一个是让你的程序做检查并丢弃值为函数的属性。一般来说，当你想让对象在运行时动态获取自身信息时，你关注更多的是数据，而你

应该意识到一些值可能会是函数。

另一个方法是使用 hasOwnProperty 方法，如果对象拥有独有的属性，它将返回 true。hasOwnProperty 方法不会检查原型链。

```
flight.hasOwnProperty('number')          // true
flight.hasOwnProperty('constructor')     // false
```

24 枚举
Enumeration

for in 语句可用来遍历一个对象中的所有属性名。该枚举过程将会列出所有的属性——包括函数和你可能不关心的原型中的属性——所以有必要过滤掉那些你不想要的值。最为常用的过滤器是 hasOwnProperty 方法，以及使用 typeof 来排除函数：

```
var name;
for (name in another_stooge) {
    if (typeof another_stooge[name] !== 'function') {
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

属性名出现的顺序是不确定的，因此要对任何可能出现的顺序有所准备。如果你想要确保属性以特定的顺序出现，最好的办法就是完全避免使用 for in 语句，而是创建一个数组，在其中以正确的顺序包含属性名：

```
var i;
var properties = [
    'first-name',
    'middle-name',
    'last-name',
    'profession'
];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' +
            another_stooge[properties[i]]);
}
```

通过使用 for 而不是 for in，可以得到我们想要的属性，而不用担心可能发掘出原型链中的属性，并且我们按正确的顺序取得了它们的值。

删除
Delete

delete 运算符可以用来删除对象的属性。如果对象包含该属性，那么该属性就会被移除。

它不会触及原型链中的任何对象。

删除对象的属性可能会让来自原型链中的属性透现出来：

```
another_stooge.nickname    // 'Moe'

// 删除 another_stooge 的 nickname 属性，从而暴露出原型的 nickname 属性。
delete another_stooge.nickname;

another_stooge.nickname    // 'Curly'
```


减少全局变量污染

Global Abatement

JavaScript 可以很随意地定义全局变量来容纳你的应用的所有资源。遗憾的是，全局变量削弱了程序的灵活性，应该避免使用。

最小化使用全局变量的方法之一是为你的应用只创建一个唯一的全局变量：

```
var MYAPP = {};
```

该变量此时变成了你的应用的容器：

```
MYAPP.stooge = {
    "first-name": "Joe",
    "last-name": "Howard"
};

MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

只要把全局性的资源都纳入一个名称空间之下，你的程序与其他应用程序、组件或类库之间发生冲突的可能性就会显著降低。你的程序也会变得更容易阅读，因为很明显，`MYAPP.stooge` 指向的是顶层结构。在下一章中，我们会看到使用闭包来进行信息隐藏的方式，它是另一种有效减少全局污染的方法。

第 4 章

函数

Functions

所有的过失在未犯以前，都已定下应处的惩罚：
——威廉·莎士比亚，《一报还一报》（*Measure for Measure*）

JavaScript 设计得最出色的就是它的函数的实现。它几乎接近于完美。但是，想必你也能预料到，JavaScript 的函数也存在瑕疵。

函数包含一组语句，它们是 JavaScript 的基础模块单元，用于代码复用、信息隐藏和组合调用。函数用于指定对象的行为。一般来说，所谓编程，就是将一组需求分解成一组函数与数据结构的技能。

函数对象

Function Objects

JavaScript 中的函数就是对象。对象是“名/值”对的集合并拥有一个连到原型对象的隐藏连接。对象字面量产生的对象连接到 `Object.prototype`。函数对象连接到 `Function.prototype`（该原型对象本身连接到 `Object.prototype`）。每个函数在创建时会附加两个隐藏属性：函数的上下文和实现函数行为的代码[译注 1]。

每个函数对象在创建时也随配有一个 `prototype` 属性。它的值是一个拥有 `constructor` 属性且值即为该函数的对象。这和隐藏连接到 `Function.prototype` 完全不同。这个令人费解的构造过程的意义将会在下个章节中揭示。

因为函数是对象，所以它们可以像任何其他的值一样被使用。函数可以保存在变量、对象和数组中。函数可以被当做参数传递给其他函数，函数也可以再返回函数。而且，因为函数是对象，所以函数可以拥有方法。

函数的与众不同之处在于它们可以被调用。

译注 1：JavaScript 创建一个函数对象时，会给该对象设置一个“调用”属性。当 JavaScript 调用一个函数时，可理解为调用此函数的“调用”属性。详细的描述请参阅 ECMAScript 规范的“13.2 Creating Function Objects”。

函数字面量

Function Literal

27

函数对象通过函数字面量来创建：

```
// 创建一个名为 add 的变量，并用来把两个数字相加的函数赋值给它。

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

函数字面量包括 4 个部分。第 1 个部分是保留字 `function`。

第 2 个部分是函数名，它可以被省略。函数可以用它的名字来递归地调用自己。此名字也能被调试器和开发工具用来识别函数。如果没有给函数命名，比如上面这个例子，它被称为匿名函数（*anonymous*）。

函数的第 3 个部分是包围在圆括号中的一组参数。多个参数用逗号分隔。这些参数的名称将被定义为函数中的变量。它们不像普通的变量那样将被初始化为 `undefined`，而是在该函数被调用时初始化为实际提供的参数的值。

第 4 个部分是包围在花括号中的一组语句。这些语句是函数的主体，它们在函数被调用时执行。

函数字面量可以出现在任何允许表达式出现的地方。函数也可以被定义在其他函数中。一个内部函数除了可以访问自己的参数和变量，同时它也能自由访问把它嵌套在其中的父函数的参数与变量。通过函数字面量创建的函数对象包含一个连到外部上下文的连接。这被称为闭包（*closure*）。它是 JavaScript 强大表现力的来源。

调用

Invocation

调用一个函数会暂停当前函数的执行，传递控制权和参数给新函数。除了声明时定义的形式参数，每个函数还接收两个附加的参数：`this` 和 `arguments`。参数 `this` 在面向对象编程中非常重要，它的值取决于调用的模式。在 JavaScript 中一共有 4 种调用模式：方法调用模式、函数调用模式、构造器调用模式和 apply 调用模式。这些模式在如何初始化关键参数 `this` 上存在差异。

调用运算符是跟在任何产生一个函数值的表达式之后的一对圆括号。圆括号内可包含零个或多个用逗号隔开的表达式。每个表达式产生一个参数值。每个参数值被赋予函数声明时定义的形式参数名。当实际参数（arguments）的个数与形式参数（parameters）的个数不匹配时，不会导致运行时错误。如果实际参数值过多了，超出的参数值会被忽略。如果实际

28

参数值过少，缺失的值会被替换为 undefined。对参数值不会进行类型检查：任何类型的值都可以被传递给任何参数。

方法调用模式
The Method Invocation Pattern

当一个函数被保存为对象的一个属性时，我们称它为一个方法。当一个方法被调用时，this被绑定到该对象。如果调用表达式包含一个提取属性的动作（即包含一个 . 点表达式或[*subscript*]下标表达式），那么它就是被当做一个方法来调用。

```
// 创建 myObject 对象。它有一个 value 属性和一个 increment 方法。
// increment 方法接受一个可选的参数。如果参数不是数字，那么默认使用数字 1。

var myObject = {
    value: 0,
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
};

myObject.increment();
document.writeln(myObject.value);    // 1

myObject.increment(2);
document.writeln(myObject.value);    // 3
```

方法可以使用 this 访问自己所属的对象，所以它能从对象中取值或对对象进行修改。this到对象的绑定发生在调用的时候。这个“超级”延迟绑定（very late binding）使得函数可以对 this 高度复用。通过 this 可取得它们所属对象的上下文的方法称为公共方法（*public method*）。

函数调用模式
The Function Invocation Pattern

当一个函数并非一个对象的属性时，那么它就是被当做一个函数来调用的：

```
var sum = add(3, 4);    // sum 的值为 7。
```

以此模式调用函数时，this 被绑定到全局对象。这是语言设计上的一个错误。倘若语言设
29 计正确，那么当内部函数被调用时，this 应该仍然绑定到外部函数的 this 变量。这个设计错误的后果就是方法不能利用内部函数来帮助它工作，因为内部函数的 this 被绑定了错误的值，所以不能共享该方法对对象的访问权。幸运的是，有一个很容易的解决方案：如果该方法定义一个变量并给它赋值为 this，那么内部函数就可以通过那个变量访问到 this。按照约定，我把那个变量命名为 that：

```
// 给 myObject 增加一个 double 方法。

myObject.double = function () {
    var that = this;    //解决方法

    var helper = function () {
        that.value = add(that.value, that.value);
    };

    helper();    //以函数的形式调用 helper。
};

// 以方法的形式调用 double。

myObject.double();
document.writeln(myObject.value);    // 6
```

构造器调用模式

The Constructor Invocation Pattern

JavaScript 是一门基于原型继承的语言。这意味着对象可以直接从其他对象继承属性。该语言是无类型的。

这偏离了当今编程语言的主流风格。当今大多数语言都是基于类的语言。尽管原型继承极富表现力，但它并未被广泛理解。JavaScript 本身对它原型的本质也缺乏信心，所以它提供了一套和基于类的语言类似的对象构建语法。有类型化语言编程经验的程序员们很少有愿意接受原型继承的，并且认为借鉴类型化语言的语法模糊了这门语言真实的原型本质。真是两边都不讨好。

如果在一个函数前面带上 `new` 来调用，那么背地里将会创建一个连接到该函数的 `prototype` 成员的新对象，同时 `this` 会被绑定到那个新对象上。

`new` 前缀也会改变 `return` 语句的行为。我们将会在后面看到更多的相关内容。

```
// 创建一个名为 Quo 的构造器函数。它构造一个带有 status 属性的对象。

var Quo = function (string) {
    this.status = string;
};

// 给 Quo 的所有实例提供一个名为 get_status 的公共方法。

Quo.prototype.get_status = function () {
    return this.status;
};

// 构造一个 Quo 实例。

var myQuo = new Quo("confused");
```

30

```
document.writeln(myQuo.get_status());  //打印显示“confused”。
```

一个函数，如果创建的目的就是希望结合 new 前缀来调用，那它就被称为构造器函数。按照约定，它们保存在以大写格式命名的变量里。如果调用构造器函数时没有在前面加上 new，可能会发生非常糟糕的事情，既没有编译时警告，也没有运行时警告，所以大写约定非常重要。

我不推荐使用这种形式的构造器函数。在下一章中我们会看到更好的替代方式。

Apply 调用模式

The Apply Invocation Pattern

因为 JavaScript 是一门函数式的面向对象编程语言，所以函数可以拥有方法。

apply 方法让我们构建一个参数数组传递给调用函数。它也允许我们选择 this 的值。apply 方法接收两个参数，第 1 个是要绑定给 this 的值，第 2 个就是一个参数数组。

```
// 构造一个包含两个数字的数组，并将它们相加。

var array = [3, 4];
var sum = add.apply(null, array);    // sum 值为 7。

// 构造一个包含 status 成员的对象。

var statusObject = {
    status: 'A-OK'
};

// statusObject 并没有继承自 Quo.prototype，但我们可以在 statusObject 上调
// 用 get_status 方法，尽管 statusObject 并没有一个名为 get_status 的方法。

var status = Quo.prototype.get_status.apply(statusObject);
    // status 值为 'A-OK'。
```

31 参数

Arguments

当函数被调用时，会得到一个“免费”配送的参数，那就是 arguments 数组。函数可以通过此参数访问所有它被调用时传递给它的参数列表，包括那些没有被分配给函数声明时定义的形式参数的多余参数。这使得编写一个无须指定参数个数的函数成为可能：

```
// 构造一个将大量的值相加的函数。

// 注意该函数内部定义的变量 sum 不会与函数外部定义的 sum 产生冲突。
```

```
// 该函数只会看到内部的那个变量。

var sum = function () {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};

document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```

这不是一个特别有用的模式。在第 6 章中，我们将会看到如何给数组添加一个相似的方法来达到同样的效果。

因为语言的一个设计错误，arguments 并不是一个真正的数组。它只是一个“类似数组（array-like）”的对象。arguments 拥有一个 length 属性，但它没有任何数组的方法。我们将在本章结尾看到这个设计错误导致的后果。

返回
Return

当一个函数被调用时，它从第一个语句开始执行，并在遇到关闭函数体的 } 时结束。然后函数把控制权交还给调用该函数的程序。

return 语句可用来使函数提前返回。当 return 被执行时，函数立即返回而不再执行余下的语句。

一个函数总是会返回一个值。如果没有指定返回值，则返回 undefined。

如果函数调用时在前面加上了 new 前缀，且返回值不是一个对象，则返回 this（该新对象）。

异常
Exceptions

32

JavaScript 提供了一套异常处理机制。异常是干扰程序的正常流程的不寻常（但并非完全是出乎意料的）的事故。当发现这样的事故时，你的程序应该抛出一个异常：

```
var add = function (a, b) {
    if (typeof a !== 'number' || typeof b !== 'number') {
        throw {
            name: 'TypeError',
            message: 'add needs numbers'
```

```
            };
        }
        return a + b;
    }
```

throw 语句中断函数的执行。它应该抛出一个 exception 对象，该对象包含一个用来识别异常类型的 name 属性和一个描述性的 message 属性。你也可以添加其他的属性。

该 exception 对象将被传递到一个 try 语句的 catch 从句：

```
// 构造一个 try_it 函数，以不正确的方式调用之前的 add 函数。

var try_it = function () {
    try {
        add("seven");
    } catch (e) {
        document.writeln(e.name + ': ' + e.message);
    }
}

try_it();
```

如果在 try 代码块内抛出了一个异常，控制权就会跳转到它的 catch 从句。

一个 try 语句只会有一个捕获所有异常的 catch 代码块。如果你的处理手段取决于异常的类型，那么异常处理器必须检查异常对象的 name 属性来确定异常的类型。

扩充类型的功能
Augmenting Types

JavaScript 允许给语言的基本类型扩充功能。在第 3 章中，我们已经看到，通过给 Object.prototype 添加方法，可以让该方法对所有对象都可用。这样的方式对函数、数组、字符串、数字、正则表达式和布尔值同样适用。

举例来说，我们可以通过给 Function.prototype 增加方法来使得该方法对所有函数可用：

33
```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

通过给 Function.prototype 增加一个 method 方法，我们下次给对象增加方法的时候就不必键入 prototype 这几个字符，省掉了一点麻烦。

JavaScript 没有专门的整数类型，但有时候确实只需要提取数字中的整数部分。JavaScript 本身提供的取整方法有些丑陋。我们可以通过给 Number.prototype 增加一个 integer 方法

来改善它。它会根据数字的正负来判断是使用 Math.ceiling 还是 Math.floor。

```
Number.method('integer', function () {
    return Math[this < 0 ? 'ceil' : 'floor'](this);
});

document.writeln((-10 / 3).integer());  // -3
```

JavaScript 缺少一个移除字符串首尾空白的方法。这个小疏忽很容易弥补：

```
String.method('trim', function () {
    return this.replace(/^\s+|\s+$/g, '');
});

document.writeln('"' + "   neat   ".trim() + '"');
```

我们的 trim 方法使用了一个正则表达式。我们将在第 7 章看到更多关于正则表达式的内容。

通过给基本类型增加方法，我们可以极大地提高语言的表现力。因为 JavaScript 原型继承的动态本质，新的方法立刻被赋予到所有的对象实例上，哪怕对象实例是在方法被增加之前就创建好了。

基本类型的原型是公用结构，所以在类库混用时务必小心。一个保险的做法就是只在确定没有该方法时才添加它。

```
// 符合条件时才增加方法。

Function.prototype.method = function (name, func) {
    if (!this.prototype[name]) {
        this.prototype[name] = func;
    }
    return this;
};
```

另一个要注意的就是 for in 语句用在原型上时表现很糟糕。我们在第 3 章已经看到了几个减轻这个问题的影响的办法：我们可以使用 hasOwnProperty 方法筛选出继承而来的属性，或者我们可以查找特定的类型。

递归

34

Recursion

递归函数就是会直接或间接地调用自身的一种函数。递归是一种强大的编程技术，它把一

个问题分解为一组相似的子问题，每一个都用一个寻常解[译注2]去解决。一般来说，一个递归函数调用自身去解决它的子问题。

“汉诺塔”[译注3]是一个著名的益智游戏。塔上有 3 根柱子和一套直径各不相同的空心圆盘。开始时源柱子上的所有圆盘都按照从小到大的顺序堆叠。目标是通过每次移动一个圆盘到另一根柱子，最终把一堆圆盘移动到目标柱子上，过程中不允许把较大的圆盘放置在较小的圆盘之上。这个问题有一个寻常解：

```
var hanoi = function (disc, src, aux, dst) {
    if (disc > 0) {
        hanoi(disc - 1, src, dst, aux);
        document.writeln('Move disc ' + disc +
                '  from ' + src + ' to ' + dst);
        hanoi(disc - 1, aux, src, dst);
    }
};

hanoi(3, 'Src', 'Aux', 'Dst');
```

圆盘数量为 3 时它返回这样的解法：

```
Move disc 1 from Src to Dst
Move disc 2 from Src to Aux
Move disc 1 from Dst to Aux
Move disc 3 from Src to Dst
Move disc 1 from Aux to Src
Move disc 2 from Aux to Dst
Move disc 1 from Src to Dst
```

hanoi 函数把一堆圆盘从一根柱子移到另一根柱子，必要时使用辅助的柱子。它把该问题分解成 3 个子问题。首先，它移动一对圆盘中较小的圆盘到辅助柱子上，从而露出下面较大的圆盘，然后移动下面的圆盘到目标柱子上。最后，它将刚才较小的圆盘从辅助柱子上再移动到目标柱子上。通过递归地调用自身去处理一对圆盘的移动，从而解决那些子问题。

传递给 hanoi 函数的参数包括当前移动的圆盘编号和它将要用到的 3 根柱子。当它调用自身的时候，它去处理当前正在处理的圆盘之上的圆盘。最终，它会以一个不存在的圆盘编号去调用。在这样的情况下，它不执行任何操作。由于该函数对非法值不予理会，我们也就不必担心它会导致死循环。

递归函数可以非常高效地操作树形结构，比如浏览器端的文档对象模型（DOM）。每次递归调用时处理指定的树的一小段。

译注 2：原文中使用了“trivial solution”，该词组为数学中的术语，可翻译为寻常解或明显解。作者用在此处意在说明递归用一般的方式去解决每个子问题。具体可参考 *http://zh.wikipedia.org/wiki/格林函数*。

译注 3：汉诺塔是印度一个古老传说，也是程序设计中的经典递归问题。详细的说明请参见 *http:// baike.baidu.com/view/191666.htm*。

```
// 定义 walk_the_DOM 函数，它从某个指定的节点开始，按 HTML 源码中的顺序
// 访问该树的每个节点。
// 它会调用一个函数，并依次传递每个节点给它。walk_the_DOM 调用自身去处理
// 每一个子节点。

var walk_the_DOM = function walk(node, func) {
    func(node);
    node = node.firstChild;
    while (node) {
        walk(node, func);
        node = node.nextSibling;
    }
};

// 定义 getElementsByAttribute 函数。它以一个属性名称字符串
// 和一个可选的匹配值作为参数。
// 它调用 walk_the_DOM，传递一个用来查找节点属性名的函数作为参数。
// 匹配的节点会累加到一个结果数组中。

var getElementsByAttribute = function (att, value) {
    var results = [];

    walk_the_DOM(document.body, function (node) {
        var actual = node.nodeType === 1 && node.getAttribute(att);
        if (typeof actual === 'string' &&
                (actual === value || typeof value !== 'string')) {
            results.push(node);
        }
    });

    return results;
};
```

一些语言提供了尾递归[译注4]优化。这意味着如果一个函数返回自身递归调用的结果，那么调用的过程会被替换为一个循环，它可以显著提高速度。遗憾的是，JavaScript 当前并没有提供尾递归优化。深度递归的函数可能会因为堆栈溢出而运行失败。

```
// 构建一个带尾递归的函数。因为它会返回自身调用的结果，所以它是尾递归。

// JavaScript 当前没有对这种形式的递归做出优化。

var factorial = function factorial(i, a) {
    a = a || 1;
    if (i < 2) {
        return a;
    }
    return factorial(i - 1, a * i);
};

document.writeln(factorial(4));    // 24
```

译注 4：尾递归（tail recursion 或 tail-end recursion）是一种在函数的最后执行递归调用语句的特殊形式的递归。参见 *http://en.wikipedia.org/wiki/Tail_recursion*。

作用域
Scope

在编程语言中，作用域控制着变量与参数的可见性及生命周期。对程序员来说这是一项重要的服务，因为它减少了名称冲突，并且提供了自动内存管理。

```
var foo = function () {
    var a = 3, b = 5;

    var bar = function () {
        var b = 7, c = 11;

// 此时， a 为 3， b 为 7, c 为 11。

        a += b + c;

// 此时, a 为 21， b 为 7, c 为 11。

    };

// 此时, a 为 3， b 为 5， 而 c 还没有定义。

    bar();

// 此时， a 为 21， b 为 5。

};
```

大多数类 C 语言语法的语言都拥有块级作用域。在一个代码块中（括在一对花括号中的一组语句）定义的所有变量在代码块的外部是不可见的。定义在代码块中的变量在代码块执行结束后会被释放掉。这是件好事。

糟糕的是，尽管 JavaScript 的代码块语法貌似支持块级作用域，但实际上 JavaScript 并不支持。这个混淆之处可能成为错误之源。

JavaScript 确实有函数作用域。那意味着定义在函数中的参数和变量在函数外部是不可见的，而在一个函数内部任何位置定义的变量，在该函数内部任何地方都可见。

很多现代语言都推荐尽可能延迟声明变量。而用在 JavaScript 上的话却会成为糟糕的建议，因为它缺少块级作用域。所以，最好的做法是在函数体的顶部声明函数中可能用到的所有变量。

37 闭包
Closure

作用域的好处是内部函数可以访问定义它们的外部函数的参数和变量（除了 this 和

arguments)。这太美妙了。

我们的 getElementsByAttribute 函数可以工作，是因为它声明了一个 results 变量，而传递给 walk_the_DOM 的内部函数也可以访问 results 变量。

一个更有趣的情形是内部函数拥有比它的外部函数更长的生命周期。

之前，我们构造了一个 myObject 对象，它拥有一个 value 属性和一个 increment 方法。假定我们希望保护该值不会被非法更改。

和以对象字面量形式去初始化 myObject 不同，我们通过调用一个函数的形式去初始化 myObject，该函数会返回一个对象字面量。函数里定义了一个 value 变量。该变量对 increment 和 getValue 方法总是可用的，但函数的作用域使得它对其他的程序来说是不可见的。

```
var myObject = (function () {
    var value = 0;

    return {
        increment: function (inc) {
            value += typeof inc === 'number' ? inc : 1;
        },
        getValue: function () {
            return value;
        }
    };
}());
```

我们并没有把一个函数赋值给 myObject。我们是把调用该函数后返回的结果赋值给它。注意最后一行的()。该函数返回一个包含两个方法的对象，并且这些方法继续享有访问 value 变量的特权。

本章之前的 Quo 构造器产生一个带有 status 属性和 get_status 方法的对象。但那看起来并不是十分有趣。为什么要用一个 getter 方法去访问你本可以直接访问到的属性呢？如果 status 是私有属性，它才是更有意义的。所以，让我们定义另一种形式的 quo 函数来做此事：

```
// 创建一个名为 quo 的构造函数。
// 它构造出带有 get_status 方法和 status 私有属性的一个对象。
var quo = function (status) {
    return {
        get_status: function () {
            return status;
        }
    };
};

// 构造一个 quo 实例。

var myQuo = quo("amazed");
```

38

```
document.writeln(myQuo.get_status());
```

这个 quo 函数被设计成无须在前面加上 new 来使用，所以名字也没有首字母大写。当我们调用 quo 时，它返回包含 get_status 方法的一个新对象。该对象的一个引用保存在 myQuo 中。即使 quo 已经返回了，但 get_status 方法仍然享有访问 quo 对象的 status 属性的特权。get_status 方法并不是访问该参数的一个副本，它访问的就是该参数本身。这是可能的，因为该函数可以访问它被创建时所处的上下文环境。这被称为闭包。

让我们来看一个更有用的例子：

```
// 定义一个函数，它设置一个 DOM 节点为黄色，然后把它渐变为白色。

var fade = function (node) {
    var level = 1;
    var step = function () {
        var hex = level.toString(16);
        node.style.backgroundColor = '#FFFF' + hex + hex;
        if (level < 15) {
            level += 1;
            setTimeout(step, 100);
        }
    };
    setTimeout(step, 100);
};

fade(document.body);
```

我们调用 fade，把 document.body 作为参数传递给它（HTML <body>标签所创建的节点）。fade 函数设置 level 为 1。它定义了一个 step 函数，接着调用 setTimeout，并传递 step 函数和一个时间（100 毫秒）给它。然后它返回，fade 函数结束。

在大约十分之一秒后，step 函数被调用。它把 fade 函数的 level 变量转化为 16 位字符。接着，它修改 fade 函数得到的节点的背景颜色。然后查看 fade 函数的 level 变量。如果背景色尚未变成白色，那么它增大 fade 函数的 level 变量，接着用 setTimeout 预定让它自己再次运行。

39 step 函数很快再次被调用。但这次，fade 函数的 level 变量值变成 2。fade 函数在之前已经返回了，但只要 fade 的内部函数需要，它的变量就会持续保留。

为了避免下面的问题，理解内部函数能访问外部函数的实际变量而无须复制是很重要的：

```
// 糟糕的例子

// 构造一个函数，用错误的方式给一个数组中的节点设置事件处理程序。
// 当点击一个节点时，按照预期，应该弹出一个对话框显示节点的序号，
// 但它总是会显示节点的数目。

var add_the_handlers = function (nodes) {
    var i;
```

```
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (e) {
            alert(i);
        };
    }
};

// 结束糟糕的例子。
```

add_the_handlers 函数的本意是想传递给每个事件处理器一个唯一值（i）。但它未能达到目的，因为事件处理器函数绑定了变量 i 本身，而不是函数在构造时的变量 i 的值。

```
// 改良后的例子

// 构造一个函数，用正确的方式给一个数组中的节点设置事件处理程序。
// 点击一个节点，将会弹出一个对话框显示节点的序号。


var add_the_handlers = function (nodes) {
    var helper = function (i) {
        return  function (e) {
            alert(i);
        };
    };
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = helper(i);
    }
};
```

避免在循环中创建函数，它可能只会带来无谓的计算，还会引起混淆，正如上面那个糟糕的例子。我们可以先在循环之外创建一个辅助函数，让这个辅助函数再返回一个绑定了当前 i 值的函数，这样就不会导致混淆了。

回调

40

Callbacks

函数使得对不连续事件的处理变得更容易。例如，假定有这么一个序列，由用户交互行为触发，向服务器发送请求，最终显示服务器的响应。最自然的写法可能会是这样的：

```
request = prepare_the_request();
response = send_request_synchronously(request);
display(response);
```

这种方式的问题在于，网络上的同步请求会导致客户端进入假死状态。如果网络传输或服务器很慢，响应会慢到让人不可接受。

更好的方式是发起异步请求，提供一个当服务器的响应到达时随即触发的回调函数。异步函数立即返回，这样客户端就不会被阻塞。

```
request = prepare_the_request();
send_request_asynchronously(request, function (response) {
        display(response);
    });
```

我们传递一个函数作为参数给 send_request_asynchronously 函数，一旦接收到响应，它就会被调用。

模块
Module

我们可以使用函数和闭包来构造模块。模块是一个提供接口却隐藏状态与实现的函数或对象。通过使用函数产生模块，我们几乎可以完全摒弃全局变量的使用，从而缓解这个 JavaScript 的最为糟糕的特性之一所带来的影响。

举例来说，假定我们想要给 String 增加一个 deentityify 方法。它的任务是寻找字符串中的 HTML 字符实体并把它们替换为对应的字符。这就需要在一个对象中保存字符实体的名字和它们对应的字符。但我们该在哪里保存这个对象呢？我们可以把它放到一个全局变量中，但全局变量是魔鬼。我们可以把它定义在该函数的内部，但是那会带来运行时的损耗，因为每次执行该函数的时候该字面量都会被求值一次。理想的方式是把它放入一个闭包，而且也许还能提供一个增加更多字符实体的扩展方法：

```
String.method('deentityify', function () {

// 字符实体表。它映射字符实体的名字到对应的字符。

    var entity = {
        quot: '"',
        lt:   '<',
        gt:   '>'
    };

// 返回 deentityify 方法。

    return function () {

// 这才是 deentityify 方法。它调用字符串的 replace 方法，
// 查找'&'开头和';'结束的子字符串。如果这些字符可以在字符实体表中找到，
// 那么就将该字符实体替换为映射表中的值。它用到了一个正则表达式（参见第 7 章）。

        return this.replace(/&([^&;]+);/g,
            function (a, b) {
                var r = entity[b];
                return typeof r === 'string' ? r : a;
            }
        );
    };
}());
```

请注意最后一行。我们用()运算法立刻调用我们刚刚构造出来的函数。这个调用所创建并返回的函数才是 deentityify 方法。

```
document.writeln(
    '&lt;&quot;&gt;'.deentityify( ));  // <">
```

模块模式利用了函数作用域和闭包来创建被绑定对象与私有成员的关联，在这个例子中，只有 deentityify 方法有权访问字符实体表这个数据对象。

模块模式的一般形式是：一个定义了私有变量和函数的函数；利用闭包创建可以访问私有变量和函数的特权函数；最后返回这个特权函数，或者把它们保存到一个可访问到的地方。

使用模块模式就可以摒弃全局变量的使用。它促进了信息隐藏和其他优秀的设计实践。对于应用程序的封装，或者构造其他单例[译注5]对象，模块模式非常有效。

模块模式也可以用来产生安全的对象。假定我们想要构造一个用来产生序列号的对象：

```
var serial_maker = function () {

// 返回一个用来产生唯一字符串的对象。
// 唯一字符串由两部分组成：前缀+序列号。
// 该对象包含一个设置前缀的方法，一个设置序列号的方法
// 和一个产生唯一字符串的 gensym 方法。

    var prefix = '';
    var seq = 0;
    return {
        set_prefix: function (p) {
            prefix = String(p);
        },
        set_seq: function (s) {
            seq = s;
        },
        gensym: function () {
            var result = prefix + seq;
            seq += 1;
            return result;
        }
    };
};

var seqer = serial_maker();
seqer.set_prefix('Q');
seqer.set_seq(1000);
var unique = seqer.gensym();    // unique 是"Q1000"
```

42

译注 5：模块模式通常结合单例模式（Singleton Pattern）使用。JavaScript 的单例就是用对象字面量表示法创建的对象，对象的属性值可以是数值或函数，并且属性值在该对象的生命周期中不会发生变化。它通常作为工具为程序其他部分提供功能支持。单例模式的更多内容请参见 *http://zh.wikipedia.org/wiki/*单例模式。

seqer 包含的方法都没有用到 this 或 that，因此没有办法损害 seqer。除非调用对应的方法，否则没法改变 prefix 或 seq 的值。seqer 对象是可变的，所以它的方法可能会被替换掉，但替换后的方法依然不能访问私有成员。seqer 就是一组函数的集合，而且那些函数被授予特权，拥有使用或修改私有状态的能力。

如果我们把 seqer.gensym 作为一个值传递给第三方函数，那个函数能用它产生唯一字符串，但却不能通过它来改变 prefix 或 seq 的值。

级联

Cascade

有一些方法没有返回值。例如，一些设置或修改对象的某个状态却不返回任何值的方法就是典型的例子。如果我们让这些方法返回 this 而不是 undefined，就可以启用级联。在一个级联中，我们可以在单独一条语句中依次调用同一个对象的很多方法。一个启用级联的 Ajax 类库可能允许我们以这样的形式去编码：

```
getElement('myBoxDiv')
    .move(350, 150)
    .width(100)
    .height(100)
    .color('red')
    .border('10px outset')
    .padding('4px')
    .appendText("Please stand by")
    .on('mousedown', function (m) {
        this.startDrag(m, this.getNinth(m));
    })
    .on('mousemove', 'drag')
    .on('mouseup', 'stopDrag')
    .later(2000, function () {
        this
            .color('yellow')
            .setHTML("What hath God wraught?")
            .slide(400, 40, 200, 200);
    })
    .tip('This box is resizeable');
```

43

在这个例子中，getElement 函数产生一个对应于 id="myBoxDiv"的 DOM 元素且给其注入了其他功能的对象。该方法允许我们移动元素，修改它的尺寸和样式，并添加行为。这些方法每一个都返回该对象，所以每次调用返回的结果可以被下一次调用所用。

级联技术可以产生出极富表现力的接口。它也能给那波构造“全能”接口的热潮降降温，一个接口没必要一次做太多事情。

柯里化[译注6]

Curry

函数也是值，从而我们可以用有趣的方式去操作函数值。柯里化允许我们把函数与传递给它的参数相结合，产生出一个新的函数。

```
var add1 = add.curry(1);
document.writeln(add1(6));    // 7
```

add1 是把 1 传递给 add 函数的 curry 方法后创建的一个函数。add1 函数把传递给它的参数的值加 1。JavaScript 并没有 curry 方法，但我们可以给 Function.prototype 扩展此功能：

```
Function.method('curry', function ( ) {
    var args = arguments, that = this;
    return function ( ) {
        return that.apply(null, args.concat(arguments));
    };
});    // 有些事好像看起来不太对头……
```

curry 方法通过创建一个保存着原始函数和要被套用的参数的闭包来工作。它返回另一个函数，该函数被调用时，会返回调用原始函数的结果，并传递调用 curry 时的参数加上当前调用的参数。它使用 Array 的 concat 方法连接两个参数数组。

糟糕的是，就像我们先前看到的那样，arguments 数组并非一个真正的数组，所以它并没有 concat 方法。要避开这个问题，我们必须在两个 arguments 数组上都应用数组的 slice 方法。这样产生出拥有 concat 方法的常规数组。

```
Function.method('curry', function ( ) {
    var slice = Array.prototype.slice,
        args = slice.apply(arguments),
        that = this;
    return function ( ) {
        return that.apply(null, args.concat(slice.apply(arguments)));
    };
});
```

44

记忆

Memoization

函数可以将先前操作的结果记录在某个对象里，从而避免无谓的重复运算。这种优化被称

译注 6：柯里化，也常译为“局部套用”，是把多参数函数转换为一系列单参数函数并进行调用的技术。这项技术以数学家 Haskell Curry（Haskell 编程语言也是以该数学家命名）的名字命名。更多内容请参考 *http://zh.wikipedia.org/wiki/柯里化*。本书的上一版中译为“套用”，再版采用开发人员更为认可的“柯里化”的翻译。

为记忆[译注7]（*memoization*）。JavaScript的对象和数组要实现这种优化是非常方便的。

比如说，我们想要一个递归函数来计算 Fibonacci 数列[译注8]。一个 Fibonacci 数字是之前两个 Fibonacci 数字之和。最前面的两个数字是 0 和 1。

```
var fibonacci = function (n) {
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
};

for (var i = 0; i <= 10; i += 1) {
    document.writeln('// ' + i + ': ' + fibonacci(i));
}

// 0: 0
// 1: 1
// 2: 1
// 3: 2
// 4: 3
// 5: 5
// 6: 8
// 7: 13
// 8: 21
// 9: 34
// 10: 55
```

这样是可以工作的，但它做了很多无谓的工作。fibonacci 函数被调用了 453 次。我们调用了 11 次，而它自身调用了 442 次去计算可能已被刚计算过的值。如果我们让该函数具备记忆功能，就可以显著地减少运算量。

我们在一个名为 memo 的数组里保存我们的存储结果，存储结果可以隐藏在闭包中。当函数被调用时，这个函数首先检查结果是否已存在，如果已经存在，就立即返回这个结果。

```
var fibonacci = function () {
    var memo = [0, 1];
    var fib = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = fib(n - 1) + fib(n - 2);
            memo[n] = result;
        }
        return result;
    };
    return fib;
}();
```

45

这个函数返回同样的结果，但它只被调用了 29 次。我们调用了它 11 次，它调用了自己 18 次去取得之前存储的结果。

译注7：在计算机领域，记忆（memoization）是主要用于加速程序计算的一种优化技术，它使得函数避免重复演算之前已被处理的输入，而返回已缓存的结果——摘自 *http://en.wikipedia. org/wiki/Memoization*。

译注8：Fibonacci 数列，中文名为斐波那契数列。它的特点是，前面相邻两项之和等于后一项的值。更多请参考 *http://zh.wikipedia.org/wiki/斐波那契数列*。

我们可以把这种技术推而广之，编写一个函数来帮助我们构造带记忆功能的函数。memoizer函数取得一个初始的memo数组和formula函数。它返回一个管理meno存储和在需要时调用formula函数的recur函数。我们把这个recur函数和它的参数传递给formula函数：

```
var memoizer = function (memo, formula) {
    var recur = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = formula (recur, n);
            memo[n] = result;
        }
        return result;
    };
    return recur;
};
```

现在，我们可以使用 memoizer 函数来定义 fibonacci 函数，提供其初始的 memo 数组和 formula 函数：

```
var fibonacci = memoizer([0, 1], function (recur, n) {
    return recur (n - 1) + recur (n - 2);
});
```

通过设计这种产生另一个函数的函数，极大地减少了我们的工作量。例如，要产生一个可记忆的阶乘函数，我们只需提供基本的阶乘公式即可：

```
var factorial = memoizer([1, 1], function (recur, n) {
     return n * recur (n - 1);
});
```

第 5 章

46

继承

Inheritance

……往往会把一件完整的东西化成无数的形象。就像凹凸镜一般，从正面望去，只见一片模糊。

——威廉·莎士比亚，《理查二世》（*The Tragedy of King Richard the Second*）

在大多数编程语言中，继承都是一个重要的主题。

在那些基于类的语言（比如 Java）中，继承（inheritance 或 `extends`）提供了两个有用的服务。首先，它是代码重用的一种形式。如果一个新的类与一个已存在的类大部分相似，那么你只需具体说明其不同点即可。代码重用的模式极为重要，因为它们可以显著地减少软件开发的成本。类继承的另一个好处是引入了一套类型系统的规范。由于程序员无需编写显式类型转换的代码，他们的工作量将大大减轻，这是一件很好的事情，因为类型转换会丧失类型系统在安全上的优势。

JavaScript 是一门弱类型语言，从不需要类型转换。对象继承关系变得无关紧要。对于一个对象来说重要的是它能做什么，而不是它从哪里来。

JavaScript 提供了一套更为丰富的代码重用模式。它可以模拟那些基于类的模式，同时它也可以支持其他更具表现力的模式。在 JavaScript 中可能的继承模式有很多。在本章中，我们将研究几种最为直接的模式。当然还有很多更为复杂的构造模式，但保持简单通常是最好的。

在基于类的语言中，对象是类的实例，并且类可以从另一个类继承。JavaScript 是一门基于原型的语言，这意味着对象直接从其他对象继承。

47 伪类

Pseudoclassical

JavaScript 的原型存在着诸多矛盾。它的某些复杂的语法看起来就像那些基于类的语言，这些语法问题掩盖了它的原型机制。它不直接让对象从其他对象继承，反而插入了一个多余的间接层：通过构造器函数产生对象。

当一个函数对象被创建时，Function 构造器产生的函数对象会运行类似这样的一些代码：

```
this.prototype = {constructor: this};
```

新函数对象被赋予一个 prototype 属性，它的值是一个包含 constructor 属性且属性值为该新函数的对象。这个 prototype 对象是存放继承特征的地方。因为 JavaScript 语言没有提供一种方法去确定哪个函数是打算用来做构造器的，所以每个函数都会得到一个 prototype 对象。constructor 属性没什么用，重要的是 prototype 对象。

当采用构造器调用模式，即用 new 前缀去调用一个函数时，函数执行的方式会被修改。如果 new 运算符是一个方法而不是一个运算符，它可能会像这样执行：

```
Function.method('new', function ( ) {

// 创建一个新对象，它继承自构造器函数的原型对象。

    var that = Object.create(this.prototype);

// 调用构造器函数，绑定 -this- 到新对象上。

    var other = this.apply(that, arguments);

// 如果它的返回值不是一个对象，就返回该新对象。

    return (typeof other === 'object' && other) || that;
});
```

我们可以定义一个构造器并扩充它的原型：

```
var Mammal = function (name) {
    this.name = name;
};

Mammal.prototype.get_name = function ( ) {
    return this.name;
};

Mammal.prototype.says = function ( ) {
    return this.saying || '';
};
```

48

现在，我们可以构造一个实例：

```
var myMammal = new Mammal('Herb the Mammal');
var name = myMammal.get_name( ); // 'Herb the Mammal'
```

我们可以构造另一个伪类来继承 Mammal，这是通过定义它的 constructor 函数并替换它的 prototype 为一个 Mammal 的实例来实现的：

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
};
```

```
// 替换 Cat.prototype 为一个新的 Mammal 实例。

Cat.prototype = new Mammal();

// 扩充新原型对象，增加 purr 和 get_name 方法。

Cat.prototype.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};
Cat.prototype.get_name = function () {
    return this.says() + ' ' + this.name + ' ' + this.says();
};

var myCat = new Cat('Henrietta');
var says = myCat.says(); // 'meow'
var purr = myCat.purr(5); // 'r-r-r-r-r'
var name = myCat.get_name(); //'meow Henrietta meow'
```

伪类模式本意是想向面向对象靠拢，但它看起来格格不入。我们可以隐藏一些丑陋的细节，通过使用 method 方法来定义一个 inherits 方法实现：

```
Function.method('inherits', function (Parent) {
    this.prototype = new Parent();
    return this;
});
```

49 我们的 inherits 和 method 方法都返回 this，这样允许我们可以采用级联的形式编程。现在可以只用一行语句构造我们的 Cat 对象。

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
}
.inherits(Mammal)
.method('purr', function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
})
.method('get_name', function () {
    return this.says() + ' ' + this.name + ' ' + this.says();
});
```

通过隐藏那些无谓的 prototype 操作细节，现在它看起来没那么怪异了。但是否真的有所改善呢？我们现在有了行为像“类”的构造器函数，但仔细看它们，你会惊讶地发现：没有私有环境，所有的属性都是公开的。无法访问 super （父类）的方法。

更糟糕的是，使用构造器函数存在一个严重的危害。如果你在调用构造器函数时忘记了在前面加上 new 前缀，那么 this 将不会被绑定到一个新对象上。悲剧的是，this 将被绑定到全局对象上，所以你不但没有扩充新对象，反而破坏了全局变量环境。那真是糟透了。发生那样的情况时，既没有编译时警告，也没有运行时警告。

这是一个严重的语言设计错误。为了降低这个问题带来的风险，所有的构造器函数都约定命名成首字母大写的形式，并且不以首字母大写的形式拼写任何其他的东西。这样我们至少可以通过目视检查去发现是否缺少了 new 前缀。一个更好的备选方案就是根本不使用 new。

“伪类”形式可以给不熟悉 JavaScript 的程序员提供便利，但它也隐藏了该语言的真实的本质。借鉴类的表示法可能误导程序员去编写过于深入与复杂的层次结构。许多复杂的类层次结构产生的原因就是静态类型检查的约束。JavaScript 完全摆脱了那些约束。在基于类的语言中，类继承是代码重用的唯一方式。而 JavaScript 有着更多且更好的选择。

对象说明符

Object Specifiers

50

有时候，构造器要接受一大串参数。这可能令人烦恼，因为要记住参数的顺序非常困难。在这种情况下，如果我们在编写构造器时让它接受一个简单的对象说明符，可能会更加友好。那个对象包含了将要构建的对象规格说明。所以，与其这样写：

```
var myObject = maker(f, l, m, c, s);
```

不如这么写：

```
var myObject = maker({
    first: f,
    middle: m,
    last: l,
    state: s,
    city: c
});
```

现在多个参数可以按任何顺序排列，如果构造器会聪明地使用默认值，一些参数可以忽略掉，并且代码也更容易阅读。

当与 JSON（参见附录 E）一起工作时，这种形式还可以有一个间接的好处。JSON 文本只能描述数据，但有时数据表示的是一个对象，把该数据与它的方法关联起来是有用的。如果构造器取得一个对象说明符，就能让它轻松实现，因为我们可以简单地把 JSON 对象传

递给构造器，而它将返回一个构造完全的对象。

原型
Prototypal

在一个纯粹的原型模式中，我们会摒弃类，转而专注于对象。基于原型的继承相比基于类的继承在概念上更为简单：一个新对象可以继承一个旧对象的属性。也许你对此感到陌生，但它真的很容易理解。你通过构造一个有用的对象开始，接着可以构造更多和那个对象类似的对象。这就可以完全避免把一个应用拆解成一系列嵌套抽象类的分类过程。

让我们先用对象字面量去构造一个有用的对象：

```
var myMammal = {
    name : 'Herb the Mammal',
    get_name : function () {
        return this.name;
    },
    says : function () {
        return this.saying || '';
    }
};
```

51 一旦有了一个想要的对象，我们就可以利用第 3 章中介绍过的 `Object.create` 方法构造出更多的实例来。接下来我们可以定制新的实例：

```
var myCat = Object.create(myMammal);
myCat.name = 'Henrietta';
myCat.saying = 'meow';
myCat.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};
myCat.get_name = function () {
    return this.says + ' ' + this.name + ' ' + this.says;
};
```

这是一种“*差异化继承（differential inheritance）*”[译注1]。通过定制一个新的对象，我们指明它与所基于的基本对象的区别。

译注 1：关于差异化继承的更多内容请参见 *https://developer.mozilla.org/en/Differential_inheritance_in_JavaScript*。

有时候，它对某些数据结构继承于其他数据结构的情形非常有用。这里就有一个例子：假定我们要解析一门类似 JavaScript 或 TEX[译注2]那样用一对花括号指示作用域的语言。定义在某个作用域里定义的条目在该作用域之外是不可见的。但在某种意义上，一个内部作用域会继承它的外部作用域。JavaScript 在表示这样的关系上做得非常好。当遇到一个左花括号时 block 函数被调用。parse 函数将从 scope 中寻找符号，并且当它定义了新的符号时扩充 scope：

```
var block = function () {

// 记住当前的作用域。构造一个包含了当前作用域中所有对象的新作用域。

    var oldScope = scope;
    scope = Object.create(scope);

// 传递左花括号作为参数调用 advance。

    advance('{');

// 使用新的作用域进行解析。

    parse(scope);

// 传递右花括号作为参数调用 advance 并抛弃新作用域，恢复原来老的作用域。

    advance('}');
    scope = oldScope;
};
```

函数化

Functional

52

迄今为止，我们所看到的继承模式的一个弱点就是没法保护隐私。对象的所有属性都是可见的。我们无法得到私有变量和私有函数。有时候这样没关系，但有时候却是大麻烦。遇到这些麻烦的时候，一些无知的程序员接受了一种伪装私有（*pretend privacy*）的模式。如果想构造一个私有属性，他们就给它起一个怪模怪样的名字，并且希望其他使用代码的用户假装看不到这些奇怪的成员。幸运的是，我们有一个更好的选择，那就是应用模块模式。

我们从构造一个生成对象的函数开始。我们以小写字母开头来命名它，因为它并不需要使用 new 前缀。该函数包括 4 个步骤。

译注 2：TEX 是一个由美国计算机教授高德纳（Donald E. Knuth）编写的功能强大的排版软件。它在学术界十分流行，特别是用在处理复杂的数学公式时。TEX 对数学公式的描述语法就是使用花括号。详细内容请参见 *http://zh.wikipedia.org/wiki/TeX*。

1. 创建一个新对象。有很多的方式去构造一个对象。它可以构造一个对象字面量，或者它可以和 new 前缀连用去调用一个构造器函数，或者它可以使用 Object.create 方法去构造一个已经存在的对象的新实例，或者它可以调用任意一个会返回一个对象的函数。

2. 有选择地定义私有实例变量和方法。这些就是函数中通过 var 语句定义的普通变量。

3. 给这个新对象扩充方法。这些方法拥有特权去访问参数，以及在第 2 步中通过 var 语句定义的变量。

4. 返回那个新对象。

这里是一个函数化构造器的伪代码模板（加粗的文本表示强调）：

```
var constructor = function (spec, my) {
    var that, 其他的私有实例变量;
    my = my || {};

    把共享的变量和函数添加到 my 中

    that = 一个新对象

    添加给 that 的特权方法

    return that;
};
```

spec 对象包含构造器需要构造一个新实例的所有信息。spec 的内容可能会被复制到私有变量中，或者被其他函数改变，或者方法可以在需要的时候访问 spec 的信息。(一个简化的方式是替换 spec 为一个单一的值。当构造对象过程中并不需要整个 spec 对象的时候，这是有用的。)

53 my 对象是一个为继承链中的构造器提供秘密共享的容器。my 对象可以选择性地使用。如果没有传入一个 my 对象，那么会创建一个 my 对象。

接下来，声明该对象私有的实例变量和方法。通过简单地声明变量就可以做到。构造器的变量和内部函数变成了该实例的私有成员。内部函数可以访问 spec、my、that，以及其他私有变量。

接下来，给 my 对象添加共享的秘密成员。这是通过赋值语句来实现的：

```
my.member = value;
```

现在，我们构造了一个新对象并把它赋值给 that。有很多方式可以构造一个新对象。我们可以使用对象字面量，可以用 new 运算符调用一个伪类构造器，可以在一个原型对象上使用 Object.create 方法，或者可以调用另一个函数化的构造器，传给它一个 spec 对象(可能就是传递给当前构造器的同一个 spec 对象) 和 my 对象。my 对象允许其他的构造器分

享我们放到 my 中的资料。其他构造器可能也会把自己可分享的秘密成员放进 my 对象里，以便我们的构造器可以利用它。

接下来，我们扩充 that，加入组成该对象接口的特权方法。我们可以分配一个新函数成为 that 的成员方法。或者，更安全地，我们可以先把函数定义为私有方法，然后再把它们分配给 that：

```
var methodical = function () {
    ...
};
that.methodical = methodical;
```

分两步去定义 methodical 的好处是，如果其他方法想要调用 methodical，它们可以直接调用 methodical() 而不是 that.methodical()。如果该实例被破坏或篡改，甚至 that.methodical 被替换掉了，调用 methodical 的方法同样会继续工作，因为它们私有的 methodical 不受该实例被修改的影响。

最后，我们返回 that。

让我们把这个模式应用到 mammal 例子里。此处不需要 my，所以我们先抛开它，但会使用一个 spec 对象。

name 和 saying 属性现在是完全私有的。只有通过 get_name 和 says 两个特权方法才可以访问它们。

```
var mammal = function (spec) {
    var that = {};

    that.get_name = function () {
        return spec.name;
    };
    that.says = function () {
        return spec.saying || '';
    };

    return that;
};

var myMammal = mammal({name: 'Herb'});
```

54

在伪类模式里，构造器函数 Cat 不得不重复构造器 Mammal 已经完成的工作。在函数化模式中那不再需要了，因为构造器 Cat 将会调用构造器 Mammal，让 Mammal 去做对象创建中的大部分工作，所以 Cat 只需关注自身的差异即可。

```
var cat = function (spec) {
    spec.saying = spec.saying || 'meow';
    var that = mammal(spec);
    that.purr = function (n) {
        var i, s = '';
        for (i = 0; i < n; i += 1) {
```

```
            if (s) {
                s += '-';
            }
            s += 'r';
        }
        return s;
    };
    that.get_name = function () {
        return that.says() + ' ' + spec.name + ' ' + that.says();
    };
    return that;
};

var myCat = cat({name: 'Henrietta'});
```

函数化模式还给我们提供了一个处理父类方法的方法。我们会构造一个 superior 方法，它取得一个方法名并返回调用那个方法的函数。该函数会调用原来的方法，尽管属性已经变化了。

```
Object.method('superior', function (name) {
    var that = this,
        method = that[name];
    return function () {
        return method.apply(that, arguments);
    };
});
```

让我们在 coolcat 上试验一下，coolcat 就像 cat 一样，除了它有一个更酷的调用父类方法的 get_name 方法。它只需要一点点的准备工作。我们会声明一个 super_get_name 变量，并且把调用 superior 方法所返回的结果赋值给它。

55

```
var coolcat = function (spec) {
    var that = cat(spec),
        super_get_name = that.superior('get_name');
    that.get_name = function (n) {
        return 'like ' + super_get_name() + ' baby';
    };
    return that;
};
var myCoolCat = coolcat({name: 'Bix'});
var name = myCoolCat.get_name();
//       'like meow Bix meow baby'
```

函数化模式有很大的灵活性。它相比伪类模式不仅带来的工作更少，还让我们得到更好的封装和信息隐藏，以及访问父类方法的能力。

如果对象的所有状态都是私有的，那么该对象就成为一个“防伪（tamper-proof）”对象。该对象的属性可以被替换或删除，但该对象的完整性不会受到损害。如果我们用函数化的样式创建一个对象，并且该对象的所有方法都不使用 this 或 that，那么该对象就是持久性（*durable*）的。一个持久性对象就是一个简单功能函数的集合。

一个持久性的对象不会被入侵。访问一个持久性的对象时，除非有方法授权，否则攻击者

不能访问对象的内部状态。

部件
Parts

我们可以从一套部件中把对象组装出来。例如，我们可以构造一个给任何对象添加简单事件处理特性的函数。它会给对象添加一个 on 方法、一个 fire 方法和一个私有的事件注册表对象：

```
var eventuality = function (that) {
    var registry = {};

    that.fire = function (event) {

//在一个对象上触发一个事件。该事件可以是一个包含事件名称的字符串，
//或者是一个拥有包含事件名称的 type 属性的对象。
//通过'on' 方法注册的事件处理程序中匹配事件名称的函数将被调用。

        var array,
            func,
            handler,
            i,
            type = typeof event === 'string' ? event : event.type;
// 如果这个事件存在一组事件处理程序，那么就遍历它们并按顺序依次执行。

        if (registry.hasOwnProperty(type)) {
            array = registry[type];
            for (i = 0; i < array.length; i += 1) {
                handler = array[i];

// 每个处理程序包含一个方法和一组可选的参数。
// 如果该方法是一个字符串形式的名字，那么寻找到该函数。

                func = handler.method;
                if (typeof func === 'string') {
                    func = this[func];
                }

// 调用一个处理程序。如果该条目包含参数，那么传递它们过去。否则，传递该事件对象。
                func.apply(this,
                    handler.parameters || [event]);
            }
        }
        return this;
    };

    that.on = function (type, method, parameters) {

// 注册一个事件。构造一条处理程序条目。将它插入到处理程序数组中，
// 如果这种类型的事件还不存在，就构造一个。
```

56

```
            var handler = {
                method: method,
                parameters: parameters
            };
            if (registry.hasOwnProperty(type)) {
                registry[type].push(handler);
            } else {
                registry[type] = [handler];
            }
            return this;
        };
        return that;
    };
```

我们可以在任何单独的对象上调用 `eventuality`，授予它事件处理方法。我们也可以赶在 `that` 被返回前在一个构造器函数中调用它。

```
eventuality(that);
```

57 用这种方式，一个构造器函数可以从一套部件中把对象组装出来。JavaScript 的弱类型在此处是一个巨大的优势，因为我们无须花费精力去了解对象在类型系统中的继承关系。相反，我们只需专注于它们的个性特征。

如果我们想要 `eventuality` 访问该对象的私有状态，可以把私有成员集 `my` 传递给它。

第 6 章

数组

Arrays

你这披着羊皮的狼，我要把你赶走。
——威廉·莎士比亚，《亨利六世上篇》（*The First Part of Henry the Sixth*）

数组是一段线性分配的内存，它通过整数计算偏移并访问其中的元素。数组是一种性能出色的数据结构。不幸的是，JavaScript 没有像此类数组一样的数据结构。

作为替代，JavaScript 提供了一种拥有一些类数组（array-like）特性的对象。它把数组的下标转变成字符串，用其作为属性。它明显地比一个真正的数组慢，但它使用起来更方便。它的属性的检索和更新的方式与对象一模一样，只不过多一个可以用整数作为属性名的特性。数组有自己的字面量格式。数组也有一套非常有用的内置方法，我将在第 8 章描述它们。

数组字面量

Array Literals

数组字面量提供了一种非常方便地创建新数组的表示法。一个数组字面量是在一对方括号中包围零个或多个用逗号分隔的值的表达式。数组字面量允许出现在任何表达式可以出现的地方。数组的第一个值将获得属性名`'0'`，第二个值将获得属性名`'1'`，依此类推：

```
var empty = [];
var numbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'eight', 'nine'
];

empty[1]          // undefined
numbers[1]        // 'one'

empty.length      // 0
numbers.length    // 10
```

对象字面量：

```
var numbers_object = {
    '0': 'zero',  '1': 'one',   '2': 'two',
```

```
    '3': 'three', '4': 'four',  '5': 'five',
    '6': 'six',   '7': 'seven', '8': 'eight',
    '9': 'nine'
};
```

两者产生的结果相似。numbers 和 numbers_object 都是包含 10 个属性的对象，并且那些属性刚好有相同的名字和值。但是它们也有一些显著的不同。numbers 继承自 Array.prototype，而 numbers_object 继承自 Object.prototype，所以 numbers 继承了大量有用的方法。同时，numbers 也有一个诡异的 length 属性，而 numbers_object 则没有。

在大多数语言中，一个数组的所有元素都要求是相同的类型。JavaScript 允许数组包含任意混合类型的值：

```
var misc = [
    'string', 98.6, true, false, null, undefined,
    ['nested', 'array'], {object: true}, NaN,
    Infinity
];
misc.length    // 10
```

长度
Length

每个数组都有一个 length 属性。和大多数其他语言不同，JavaScript 数组的 length 是没有上界的。如果你用大于或等于当前 length 的数字作为下标来存储一个元素，那么 length 值会被增大以容纳新元素，不会发生数组越界错误。

length 属性的值是这个数组的最大整数属性名加上 1。它不一定等于数组里的属性的个数：

```
var myArray = [];
myArray.length           // 0

myArray[1000000] = true;
myArray.length           // 1000001
// myArray 只包含一个属性。
```

[]后置下标运算符把它所含的表达式转换成一个字符串，如果该表达式有 toString 方法，就使用该方法的值。这个字符串将被用做属性名。如果这个字符串看起来像一个大于等于这个数组当前的 length 且小于 4294967295 的正整数，那么这个数组的 length 就会被重新设置为新的下标加 1[译注1]。

译注 1：根据 ECMAScript262 的标准，数组的下标必须是大于等于 0 并小于 232-1 的整数。更多内容请参见《JavaScript 权威指南》中译第 6 版中的章节 ——“7.2 数组元素的读和写”。

你可以直接设置 length 的值。设置更大的 length 不会给数组分配更多的空间。而把 length 设小将导致所有下标大于等于新 length 的属性被删除：


```
numbers.length = 3;
// numbers 是 ['zero', 'one', 'two']
```

通过把下标指定为一个数组的当前 length，可以附加一个新元素到该数组的尾部：

```
numbers[numbers.length] = 'shi';
// numbers 是 ['zero', 'one', 'two', 'shi']
```

有时用 push 方法可以更方便地完成同样的事情：

```
numbers.push('go');
// numbers 是 ['zero', 'one', 'two', 'shi', 'go']
```

删除
Delete

由于 JavaScript 的数组其实就是对象，所以 delete 运算符可以用来从数组中移除元素：

```
delete numbers[2];
// numbers 是 ['zero', 'one', undefined, 'shi', 'go']
```

不幸的是，那样会在数组中留下一个空洞。这是因为排在被删除元素之后的元素保留着它们最初的属性。而你通常想要的是递减后面每个元素的属性。

幸运的是，JavaScript 数组有一个 splice 方法。它可以对数组做个手术，删除一些元素并将它们替换为其他的元素。第 1 个参数是数组中的一个序号，第 2 个参数是要删除的元素个数。任何额外的参数会在序号那个点的位置被插入到数组中：

```
numbers.splice(2, 1);
// numbers 是 ['zero', 'one', 'shi', 'go']
```

值为 'shi'的属性的键值从'3'变到'2'。因为被删除属性后面的每个属性必须被移除，并且以一个新的键值重新插入，这对于大型数组来说可能会效率不高。

枚举
Enumeration

因为 JavaScript 的数组其实就是对象，所以 for in 语句可以用来遍历一个数组的所有属性。遗憾的是，for in 无法保证属性的顺序，而大多数要遍历数组的场合都期望按照阿拉伯数字顺序来产生元素。此外，可能从原型链中得到意外属性的问题依旧存在。

61 幸运的是，常规的 for 语句可以避免这些问题。JavaScript 的 for 语句和大多数类 C（C-like）语言相似。它被 3 个从句控制——第 1 个初始化循环，第 2 个执行条件检测，第 3 个执行增量运算：

```
var i;
for (i = 0; i < myArray.length; i += 1) {
    document.writeln(myArray[i]);
}
```

容易混淆的地方

Confusion

在 JavaScript 编程中，一个常见的错误是在必须使用数组时使用了对象，或者在必须使用对象时使用了数组。其实规则很简单：当属性名是小而连续的整数时，你应该使用数组。否则，使用对象。

JavaScript 本身对于数组和对象的区别是混乱的。typeof 运算符报告数组的类型是 'object'，这没有任何意义。

JavaScript 没有一个好的机制来区别数组和对象。我们可以通过定义自己的 is_array 函数来弥补这个缺陷：

```
var is_array = function (value) {
    return value &&
        typeof value === 'object' &&
        value.constructor === Array;
};
```

遗憾的是，它在识别从不同的窗口（window）或帧（frame）里构造的数组时会失败。有一个更好的方式去判断一个对象是否为数组：

```
var is_array = function (value) {
    return Object.prototype.toString.apply(value) === '[object Array]';
};
```

方法

Methods

JavaScript 提供了一套数组可用的方法。这些方法是被储存在 Array.prototype 中的函数。在第 3 章里，我们看到 Object.prototype 是可以被扩充的。同样，Array.prototype 也可以被扩充。

举例来说，假设我们想要给 array 增加一个方法，它允许我们对数组进行计算：

```
Array.method('reduce', function (f, value) {
    var i;
    for (i = 0; i < this.length; i += 1) {
        value = f(this[i], value);
    }
    return value;
});
```

通过给 Array.prototype 扩充一个函数，每个数组都继承了这个方法。在这个例子里，我们定义了一个 reduce 方法，它接受一个函数和一个初始值作为参数。它遍历这个数组，以当前元素和该初始值为参数调用这个函数，并且计算出一个新值。当完成时，它返回这个值。如果我们传入一个把两个数字相加的函数，它会计算出相加之和。如果我们传入把两个数字相乘的函数，它会计算两者的乘积：

62

```
// 创建一个数字数组。

var data = [4, 8, 15, 16, 23, 42];

// 定义两个简单的函数。一个是把两个数字相加，另一个是把两个数字相乘。

var add = function (a, b) {
    return a + b;
};

var mult = function (a, b) {
    return a * b;
};

// 调用 data 的 reduce 方法，传入 add 函数。

var sum = data.reduce(add, 0);    // sum is 108

// 再次调用 reduce 方法，这次传入 mult 函数。

var product = data.reduce(mult, 1);

// product 是 7418880。
```

因为数组其实就是对象，所以我们可以直接给一个单独的数组添加方法：

```
// 给 data 数组添加一个 total 方法。

data.total = function () {
    return this.reduce(add, 0);
};

total = data.total();    // total 是 108。
```

因为字符串 'total'不是整数，所以给数组增加一个 total 属性不会改变它的 length。当属性名是整数时，数组才是最有用的，但它们依旧是对象，并且对象可以接受任何字符串作为属性名。

来自第 3 章的 Object.create 方法用在数组是没有意义的，因为它产生一个对象，而不是一个数组。产生的对象将继承这个数组的值和方法，但它没有那个特殊的 length 属性。

63 指定初始值

Dimensions

JavaScript 的数组通常不会预置值。如果你用[]得到一个新数组，它将是空的。如果你访问一个不存在的元素，得到的值则是 undefined。如果你知道这个问题，或者你在尝试获取每个元素之前都很有预见性地设置它的值，那就万事大吉了。但是，如果你实现的算法是假设每个元素都从一个已知的值开始(例如 0)，那么你必须自己准备好这个数组。JavaScript 应该提供一些类似 Array.dim 这样的方法来做这件事情，但我们可以很容易纠正这个疏忽：

```
Array.dim = function (dimension, initial) {
    var a = [], i;
    for (i = 0; i < dimension; i += 1) {
        a[i] = initial;
    }
    return a;
};

// 创建一个包含 10 个 0 的数组。

var myArray = Array.dim(10, 0);
```

JavaScript 没有多维数组，但就像大多数类 C 语言一样，它支持元素为数组的数组：

```
var matrix = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
];
matrix[2][1]    // 7
```

为了创建一个二维数组或者说数组的数组，你必须自己去创建那个第二维的数组：

```
for (i = 0; i < n; i += 1) {
    my_array[i] = [];
}

//注意： Array.dim(n, [])在这里不能工作。
//如果使用它，每个元素都指向同一个数组的引用，那后果不堪设想。
```

一个空的矩阵的每个单元会拥有一个初始值 undefined。如果你希望它们有不同的初始值，你必须明确地设置它们。同样地，JavaScript 应该对矩阵提供更好的支持。好在我们也可以补上它：

```
Array.matrix = function (m, n, initial) {
    var a, i, j, mat = [];
```

```
    for (i = 0; i < m; i += 1) {
        a = [];
        for (j = 0; j < n; j += 1) {
            a[j] = initial;
        }
        mat[i] = a;
    }
    return mat;
}

// 构造一个用 0 填充的 4×4 矩阵。

var myMatrix = Array.matrix(4, 4, 0);

document.writeln(myMatrix[3][3]);    // 0

// 用来构造一个单位矩阵的方法。

Array.identity = function (n) {
    var i, mat = Array.matrix(n, n, 0);
    for (i = 0; i < n; i += 1) {
        mat[i][i] = 1;
    }
    return mat;
};

myMatrix = Array.identity(4);

document.writeln(myMatrix[3][3]);    // 1
```

64

第 7 章

正则表达式

Regular Expressions

相反地，选到一个称心如意的配偶，就能百年谐合，幸福无穷。我们不应该选谁来配亨利国王，他作为一国之君，……

——威廉·莎士比亚，《亨利六世上篇》（*The First Part of Henry the Sixth*）

JavaScript 的许多特性都借鉴自其他语言。语法借鉴自 Java，函数借鉴自 Scheme[译注 1]，原型继承借鉴自 Self[译注 2]。而 JavaScript 的正则表达式特性则借鉴自 Perl。

正则表达式是一门简单语言的语法规范。它应用在一些方法中，对字符串中的信息实现查找、替换和提取操作。可处理正则表达式的方法有 `regexp.exec`、`regexp.test`、`string.match`、`string.replace`、`string.search` 和 `string.split`。我会在第 8 章详述它们。通常来说，在 JavaScript 中正则表达式相较于等效的字符串处理有着显著的性能优势。

正则表达式起源于对形式语言[译注 3](formal language)的数学研究。Ken Thompson 基于 Stephen Kleene 对 type-3 语言的理论研究写出了一个切实可用的模式匹配器，它能够被嵌入到编程语言和像文本编辑器这样的工具中。

在 JavaScript 中，正则表达式的语法是对 Perl 版本的改进和发展，它非常接近于贝尔实验室（Bell Labs）最初提出的构想。正则表达式的书写规则出奇地复杂，在某些位置上的字符串可能解析为运算符，而仅在位置上稍微不同的相同字符串却可能被当做字面量。比不易书写更糟糕的是，这使得正则表达式不仅难以阅读，而且修改时充满危险。要想正确地阅读它们，就必须对正则表达式的整个复杂性有相当透彻的理解。为了缓解这个问题，我对它的规则进行了些许简化。这里所展示的正则表达式可能稍微有些不够简洁，但使用它们的时候不会那么容易出错。这是值得的，因为维护和调试正则表达式可能非常困难。

译注 1：Scheme，一种多范型的编程语言，它是两种 lisp 主要的方言之一。而 lisp（全名 LISt Processor，即链表处理语言）是由约翰·麦卡锡在 1960 年左右创造的一种基于 λ 演算的函数式编程语言。更多详细内容请参见 *http://zh.wikipedia.org/wiki/Scheme* 和 *http://zh.wikipedia.org/wiki/Lisp*。

译注 2：Self 语言，是一种基于原型的面向对象程序设计语言，于 1986 年由施乐帕洛阿尔托研究中心的 David Ungar 和 Randy Smith 给出了最初的设计。更多详细内容请参见 *http://zh.wikipedia.org/wiki/Self_*语言。

译注 3：在数学、逻辑和计算机科学中，形式语言是用精确的数学或机器可处理的公式定义的语言。更多相关内容请参见 *http:// zh.wikipedia.org/wiki/*形式语言。

现在的正则表达式的规则并不总是严格的，但它们非常有用。正则表达式趋向于极致的简洁，甚至不惜容忍含义的模糊。在最简单的形式下，它们是易于使用的，但可能很快就会变得让人费解。JavaScript 的正则表达式难以分段阅读，因为它们不支持注释和空白。正则表达式的所有部分都被紧密排列在一起，使得它们几乎无法被辨认。当它们在安全应用中进行扫描和验证时，这点就需要特别地留意。如果你不能阅读和理解一个正则表达式，你如何能确保它对所有的输入都能正确地工作呢？然而，尽管有这些明显的缺点，但正则表达式还是被广泛地应用着。

一个例子
An Example

这里有一个例子。它是一个用来匹配 URL 的正则表达式。书页的行宽有限，所以我把它拆开来写成两行。在 JavaScript 程序中，正则表达式必须写在一行中。空白需要特别注意：

```
var parse_url = /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)
(?::(\d+))?(?:\/([^?#]*))?(?:\?([^#]*))?(?:#(.*))?$/;

var url = "http://www.ora.com:80/goodparts?q#fragment";
```

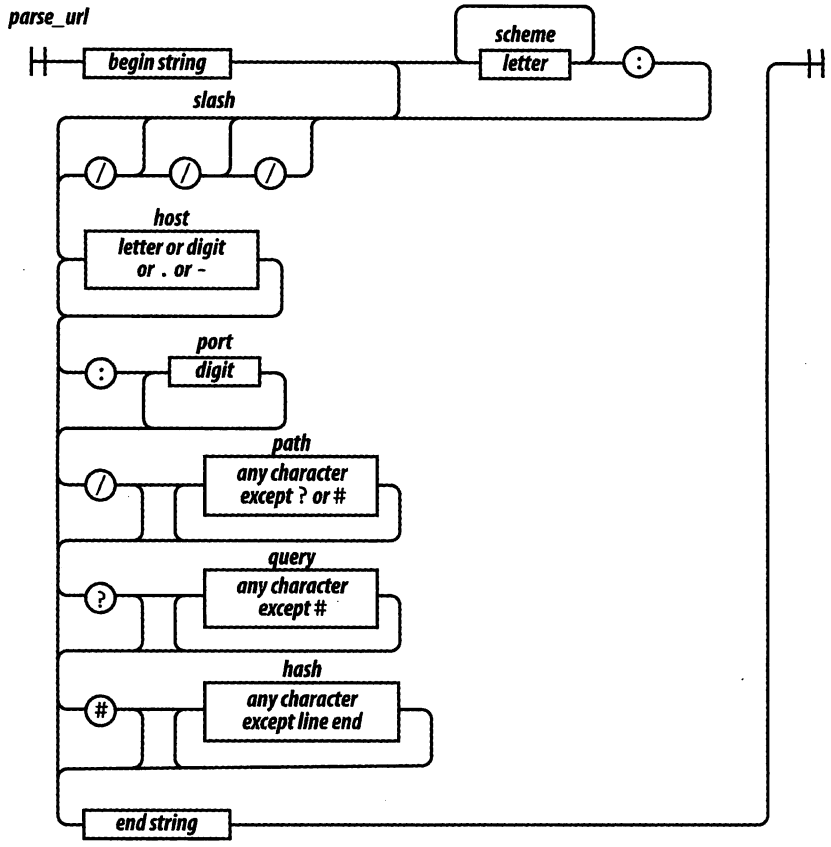
让我们调用 parse_url 的 exec 方法。如果能成功匹配我们传给它的字符串，它会返回一个数组，该数组包含了从这个 url 中提取出来的片段：

```
var url = "http://www.ora.com:80/goodparts?q#fragment";

var result = parse_url.exec(url);

var names = ['url', 'scheme', 'slash', 'host', 'port',
    'path', 'query', 'hash'];

var blanks = '       ';
var i;

for (i = 0; i < names.length; i += 1) {
    document.writeln(names[i] + ':' +
        blanks.substring(names[i].length), result[i]);
}
```

这段代码产生的结果如下：

```
url:    http://www.ora.com:80/goodparts?q#fragment
scheme: http
slash:  //
host:   www.ora.com
port:   80
path:   goodparts
query:  q
hash:   fragment
```

67 在第 2 章里，我们用铁路图来描述 JavaScript 语言。我们也能用它来描述正则表达式所定义的语言。这可以让我们更容易地看出一个正则表达式做了什么。下面是用来描述 parse_url 的铁路图。



正则表达式不能像函数那样被分解成小片段，所以表示 parse_url 的轨道很长。

让我们分解 parse_url 的各个部分，看看它是如何工作的：

```
^
```

^字符表示此字符串的开始。它是一个锚，指引 exec 不要跳过那些不像 URL（non-URL-like）的前缀，只匹配那些从开头就像 URL 一样的字符串：

```
(?:([A-Za-z]+):)?
```

这个因子匹配一个协议名，但仅当它后面跟随一个 :（冒号）的时候才匹配。(?: . . .) 表示一个非捕获型分组（noncapturing group）。后缀 ? 表示这个分组是可选的。

它表示重复 0 次或 1 次。(. . .) 表示一个捕获型分组 (capturing group)。一个捕获型分组会复制它所匹配的文本，并把其放到 result 数组里。每个捕获型分组都会被指定一个编号。第一个捕获型分组的编号是 1，所以该分组所匹配的文本副本会出现在 result[1] 中。[. . .]表示一个字符类。A-Za-z 这个字符类包含 26 个大写字母和 26 个小写字母。连字符 (-) 表示范围从 A 到 Z。后缀 + 表示这个字符类会被匹配一次或多次。这个组后面跟着字符:，它会按字面进行匹配：

68

```
(\/{0,3})
```

下一个因子是捕获型分组 2。\/ 表示应该匹配 / (斜杠)。它用 \ (反斜杠) 来进行转义，这样它就不会被错误地解释为这个正则表达式的结束符。后缀 {0,3}表示 / 会被匹配 0 次，或者 1～3 次：

```
([0-9.\-A-Za-z]+)
```

下一个因子是捕获型分组 3。它会匹配一个主机名，由一个或多个数字、字母，以及 . 或 - 字符组成。-会被转义为 \- 以防止与表示范围的连字符相混淆：

```
(?::(\d+))?
```

下一个可选的因子匹配端口号，它是由一个前置 : 加上一个或多个数字而组成的序列。\d 表示一个数字字符。一个或多个数字组成的数字串会被捕获型分组 4 捕获：

```
(?:\/([^?#]*))?
```

接下来是另一个可选的分组。该分组以一个 / 开始。之后的字符类[^?#]以一个 ^ 开始，它表示这个类包含除 ? 和 # 之外的所有字符。* 表示这个字符类会被匹配 0 次或多次。

注意我在这里的处理是不严谨的。这个类匹配除 ? 和 # 之外的所有字符，其中包括了行结束符、控制字符，以及其他大量不应在此被匹配的字符。大多数情况下，它会按照我们的预期去做，但某些恶意文本可能会有渗漏进来的风险。不严谨的正则表达式是一个常见的安全漏洞发源地。写不严谨的正则表达式比写严谨的正则表达式要容易得多。

```
(?:\?([^#]*))?
```

接下来，我们还有一个以一个 ? 开始的可选分组。它包含捕获型分组 6，这个分组包含 0 个或多个非 # 字符。

```
(?:#(.*))?
```

我们的最后一个可选分组是以 # 开始的。. 会匹配除行结束符以外的所有字符。

```
$
```

$表示这个字符串的结束。它保证在这个 URL 的尾部没有其他更多内容了。

69

以上便是正则表达式 parse_url 的所有因子[注1]。

parse_url 的正则表达式还可以编写得更复杂，但我不建议这样做。短小精悍的正则表达式是最好的。唯有如此，我们才有信心让它们正确地工作并在需要时能顺利地修改它们。

JavaScript 的语言处理程序之间兼容性非常高。这门语言中最没有移植性的部分就是对正则表达式的实现。结构复杂或令人费解的正则表达式很有可能导致移植性问题。在执行某些匹配时，嵌套的正则表达式也能导致极恶劣的性能问题。因此简单是最好的策略。

让我们来看另一个例子：一个匹配数字的正则表达式。数字可能由一个整数部分加上一个可选的负号、一个可选的小数部分和一个可选的指数部分组成：

```
var parse_number = /^-?\d+(?:\.\d*)?(?:e[+\-]?\d+)?$/i;

var test = function (num) {
    document.writeln(parse_number.test(num));
};

test('1');                  // true
test('number');             // false
test('98.6');               // true
test('132.21.86.100');      // false
test('123.45E-67');         // true
test('123.45D-67');         // false
```

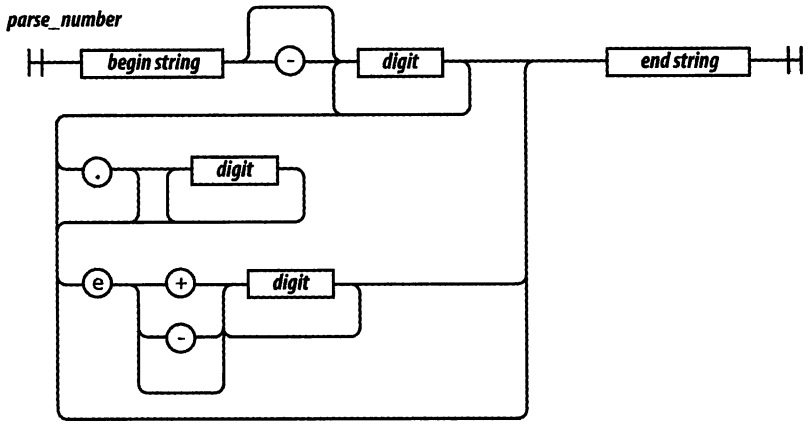
parse_number 成功地检验出这些字符串中，哪些符合我们的规范，哪些不符合，但对那些不符合的字符串，它并没有告诉我们测试失败的缘由和位置。

让我们来分解 parse_number：

```
/^   $/i
```

我们又用^和$来框定这个正则表达式。它指引这个正则表达式对文本中的所有字符都进行匹配。如果我们省略了这些标识，那么只要一个字符串包含一个数字，这个正则表达式就会进行匹配。但有了这些标识，只有当一个字符串的内容仅为一个数字时，它才会告诉我们。如果我们仅包含^，它将匹配以一个数字开头的字符串。如果我们仅包含$，则匹配以一个数字结尾的字符串。

注 1：当你把它们全部写在一起时，在视觉上是相当难以辨识的：/^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^?#]*))?(?:\?([^#]*))?(?:#(.*))?$/。



i 标识表示匹配字母时忽略大小写。在我们的模式中唯一可能出现的字母是 e。我们希望既能匹配 e，也能匹配 E。我们可以把 e 因子写成[Ee]或(?:E|e)，但不必这么麻烦，因为我们使用了标识符 i。

```
-?
```

负号后面的?后缀表示这个负号是可选的。

```
\d+
```

\d 的含义和[0-9]一样。它匹配一个数字。后缀 + 指引它可以匹配一个或多个数字。

```
(?:\.\d*)?
```

(?: . . .)?表示一个可选的非捕获型分组。通常用非捕获型分组来替代少量不优美的捕获型分组是很好的方法，因为捕获会有性能上的损失。这个分组会匹配后面跟随的 0 个或多个数字的小数点：

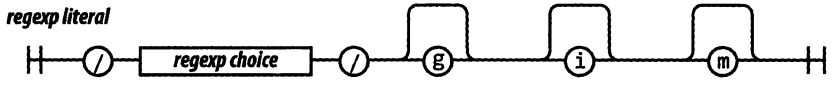
```
(?:e[+\-]?\d+)?
```

这是另外一个可选的非捕获型分组。它会匹配一个 e（或 E）、一个可选的正负号及一个或多个数字。

结构

Construction

有两个方法来创建一个 RegExp 对象。在以前的例子中我们可以看到，优先考虑的方法是使用正则表达式字面量。



71 正则表达式字面量被包围在一对斜杠中。这有点令人难以捉摸，因为斜杠也被用做除法运算符和注释符。

RegExp 能设置 3 个标识。它们分别由字母 g、i 和 m 来标示，我把它们列在表 7-1 中。这些标识被直接添加在 RegExp 字面量的末尾：

```
// 构造一个匹配 JavaScript 字符串的正则表达式对象。

var my_regexp = /"(?:\\.|[^\\\"])*"/g;
```

表 7-1：正则表达式标识

标识	含义
g	全局的（匹配多次；不同的方法对 g 标识的处理各不相同[译注4]
i	大小写不敏感（忽略字符大小写）
m	多行（^和$能匹配行结束符）

创建一个正则表达式的另一个方法是使用 RegExp 构造器。这个构造器接收一个字符串，并把它编译为一个 RegExp 对象。创建这个字符串时请多加小心，因为反斜杠在正则表达式和在字符串字面量中有一些不同的含义。通常需要双写反斜杠，以及对引号进行转义：

```
// 创建一个匹配 JavaScript 字符串的正则表达式。

var my_regexp = new RegExp("\"(?:\\\\.|[^\\\\\\\\\"])*\"", 'g');
```

第 2 个参数是一个指定标识的字符串。RegExp 构造器适用于必须在运行时动态生成正则表达式的情形。

RegExp 对象包含的属性列在表 7-2 中。

表 7-2：RegExp 对象的属性

属性	用法
global	如果标识 g 被使用，值为 true
ignoreCase	如果标识 i 被使用，值为 true
lastIndex	下一次 exec 匹配开始的索引。初始值为 0
multiline	如果标识 m 被使用，值为 true
source	正则表达式源码文本

72 用正则表达式字面量创建 RegExp 对象共享同一个单例：

```
function make_a_matcher() {
```

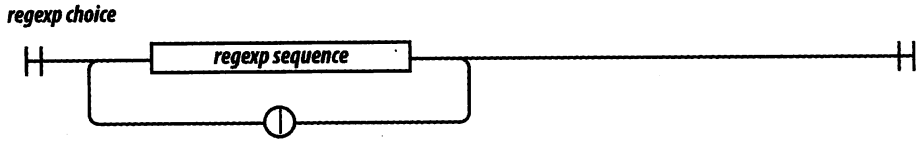
译注 4：比如 RegExp 对象的 exec 方法和 test 方法，作者不建议在 test 方法中对正则表达式使用 g 标识。再比如 String 的 match 方法和 search 方法，search 方法会忽略 g 标识。更多内容可参见下一章的内容。

```
    return /a/gi;
}

var x = make_a_matcher();
var y = make_a_matcher();

// 当心：x 和 y 是相同的对象！

x.lastIndex = 10;

document.writeln(y.lastIndex);    // 10
```

元素
Elements

让我们进一步看看那些构成正则表达式的元素。

正则表达式分支
Regexp Choice

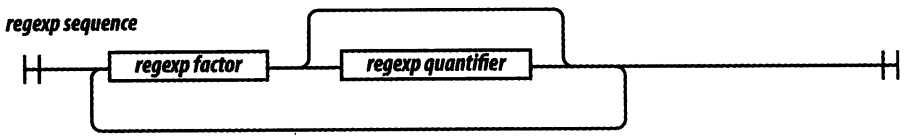


一个*正则表达式分支*包含一个或多个*正则表达式序列*。这些序列被|（竖线）字符分隔。如果这些序列中的任何一项符合匹配条件，那么这个选择就被匹配。它尝试按顺序依次匹配这些序列项。所以：

```
"into".match(/in|int/)
```

会在 into 中匹配 in。但它不会匹配 int，因为 in 已被成功匹配了。

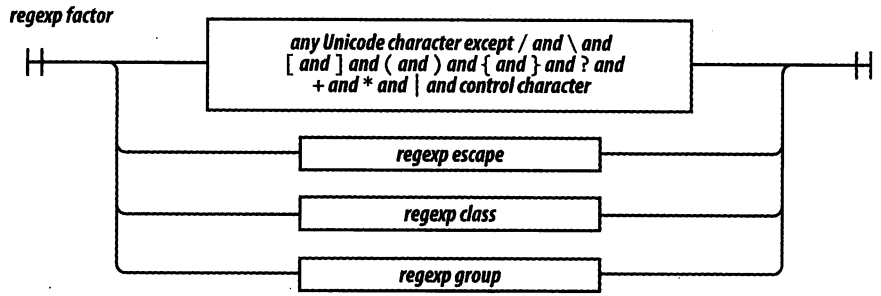
正则表达式序列
Regexp Sequence



一个*正则表达式序列*包含一个或多个*正则表达式因子*。每个因子能选择是否跟随一个量词，这个量词决定着这个因子被允许出现的次数。如果没有指定这个量词，那么该因子只会被

匹配一次。

73 正则表达式因子
Regexp Factor



一个正则表达式因子可以是一个字符、一个由圆括号包围的组、一个字符类，或者是一个转义序列。除了控制字符和特殊字符以外，所有的字符都会被按照字面处理：

```
\ / [ ] () { } ? + * | . ^ $
```

如果你希望上面列出的字符按字面去匹配，那么必须要用一个 \ 前缀来进行转义。你如果拿不准的话，可以给任何特殊字符都添加一个 \ 前缀来使其字面化。注意\ 前缀不能使字母或数字字面化。

一个未被转义的 . 会匹配除行结束符以外的任何字符。

当 `lastIndex` 属性值为 0 时，一个未转义的 ^ 会匹配文本的开始。当指定了 `m` 标识时，它也能匹配行结束符。

一个未转义的$将匹配文本的结束。当指定了 `m` 标识时，它也能匹配行结束符。

正则表达式转义
Regexp Escape

反斜杠字符在正则表达式因子中与其在字符串中一样均表示转义，但是在正则表达式因子中，它稍有一点不同。

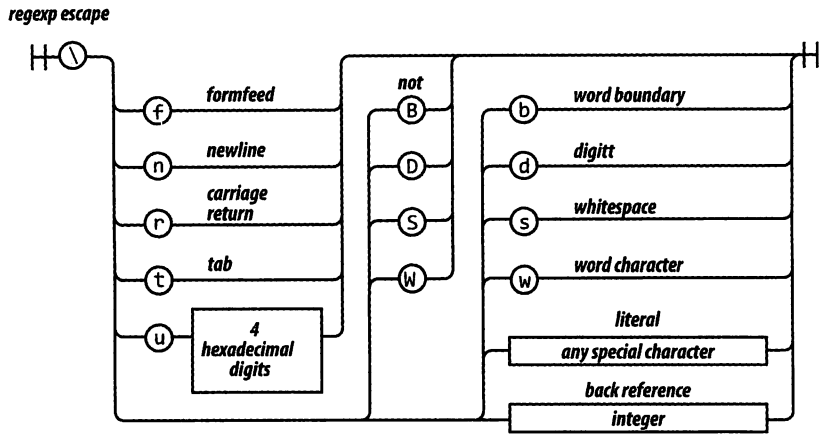
像在字符串中一样，`\f` 是换页符，`\n` 是换行符，`\r` 是回车符，`\t` 是制表（tab）符，并且 `\u` 允许指定一个 Unicode 字符来表示一个十六进制的常量。但在正则表达式因子中，`\b` 不是退格（backspace）符。

`\d` 等同于`[0-9]`，它匹配一个数字。`\D` 则表示与其相反的：`[^0-9]`。

`\s` 等同于`[\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]`。这是 Unicode 空白（whitespace）

符的一个不完全子集。\S 则表示与其相反的：[^\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]。

\w 等同于[0-9A-Z_a-z]。\W 则表示与其相反的：[^0-9A-Z_a-z]。\W 本意是希望表示出现在话语中的字符。遗憾的是，它所定义的类实际上对任何真正的语言来说都不起作用。如果你需要匹配信件一类的文本，你必须指定自己的类。



一个简单的字母类是[A-Za-z\u00C0-\u1FFF\u2800-\uFFFD]。它包括所有的 Unicode 字母，但它也包括成千上万非字母的字符。Unicode 是巨大而复杂的。构造一个基本多语言面[译注5]的精确字母类是有可能的，但它将会变得庞大而低效。JavaScript 的正则表达式对国际化的支持非常有限。

\b 被指定为一个字边界（word-boundary）标识，它方便用于对文本的字边界进行匹配。遗憾的是，它使用 \w 去寻找字边界，所以它对多语言应用来说是完全无用的。这并不是一个好的特性。

\1 是指向分组 1 所捕获到的文本的一个引用，所以它能被再次匹配。例如，你能用下面的正则表达式来搜索文本中的重复的单词：

```
var doubled_words =
    /([A-Za-z\u00C0-\u1FFF\u2800-\uFFFD]+)\s+\1/gi;
```

doubled_words 会寻找重复的单词（包含一个或多个字母的字符串），该单词的后面跟着一个或多个空白，然后再跟着与它相同的单词。.

\2 是指向分组 2 的引用，\3 是指向分组 3 的引用，依此类推。

译注 5：基本多语言面（Basic Multilingual Plane，BMP），或者称第 0 平面（Plane 0），是 Unicode 中的一个编码区段。编码从 U+0000 至 U+FFFF。详细信息见 *http://zh.wikipedia.org/wiki/基本多文种平面*。

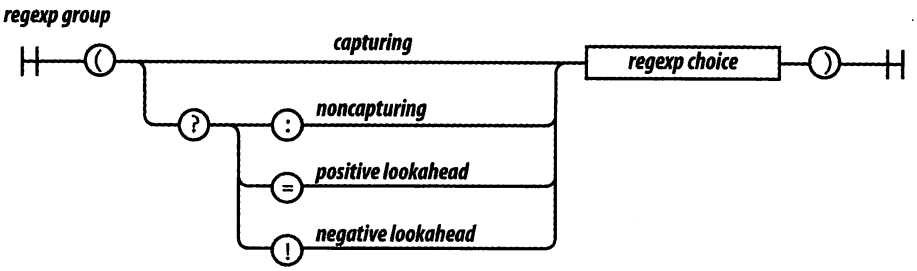
正则表达式分组
Regexp Group

分组共有 4 种。

捕获型

一个捕获型分组是一个被包围在圆括号中的正则表达式分支。任何匹配这个分组的字符都会被捕获。每个捕获型分组都被指定了一个数字。在正则表达式中第 1 个捕获（ 的是分组 1，第 2 个捕获 （ 的是分组 2。

75



非捕获型

非捕获型分组有一个(?:前缀。非捕获型分组仅做简单的匹配，并不会捕获所匹配的文本。这会带来微弱的性能优势。非捕获型分组不会干扰捕获型分组的编号。

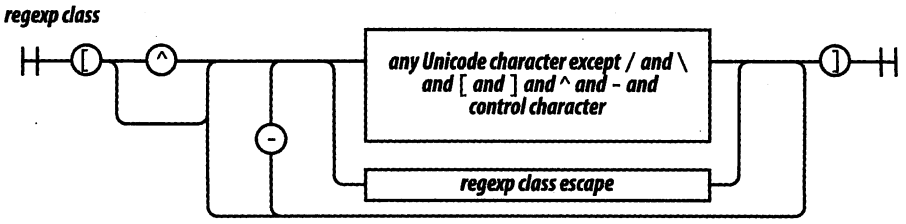
向前正向匹配（Positive lookahead）

向前正向匹配分组有一个(?=前缀。它类似于非捕获型分组，但在这个组匹配后，文本会倒回到它开始的地方，实际上并不匹配任何东西。这不是一个好的特性。

向前负向匹配（Negative lookahead）

向前负向匹配分组有一个(?!前缀。它类似于向前正向匹配分组，但只有当它匹配失败时它才继续向前进行匹配。这不是一个好的特性。

正则表达式字符集
Regexp Class



正则表达式字符集是一种指定一组字符的便利方式。例如，如果想匹配一个元音字母，我们可以写做`(?:a|e|i|o|u)`，但它可以被更方便地写成一个类`[aeiou]`。

类提供另外两个便利。第 1 个是能够指定字符范围。所以，一组由 32 个 ASCII 的特殊字符组成的集合：

```
! " # $ % & ' () * +, - . / :
; < = > ? @ [ \ ] ^ _ ` { | } ~
```

你可以写成这样：

```
?:!|"|#|\$|%|&|'|\(|\)|\*|\+|,|-|\.|\/|:|;|<|=|>|@|\[|\\|]|\^|_|` |\{|\||\}|~)
```

稍微好看一些的写法是：

76

```
[!-\/:-@\[-`{-~]
```

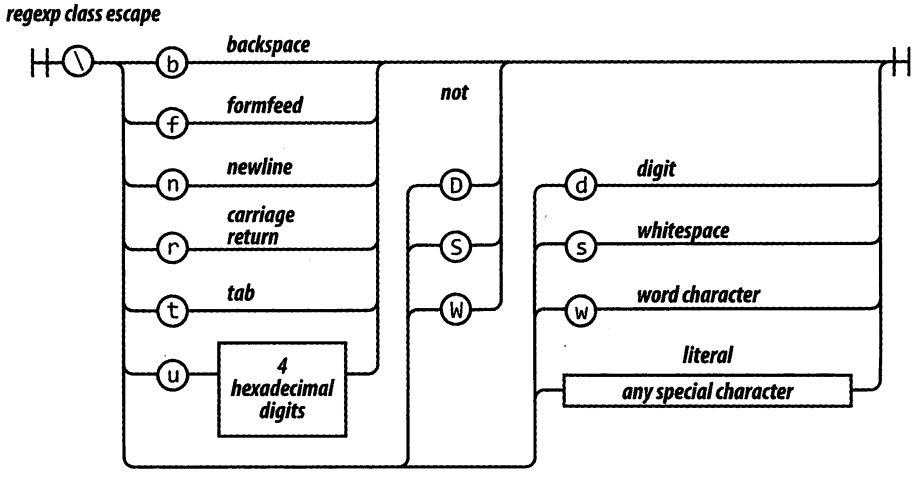
它包括从`!`到`/`、从`:`到`@`、从`[`到`` ` ``和从`[`到`~`的字符。但它看起来依旧相当难以阅读。

另一个方便之处是类的求反。如果`[`后的第一个字符是`^`，那么这个类会排除这些特殊字符。

所以`[^!-\/:-@\[-`{-~]`会匹配任何一个非 ASCII 特殊字符的字符。

正则表达式字符转义

Regexp Class Escape



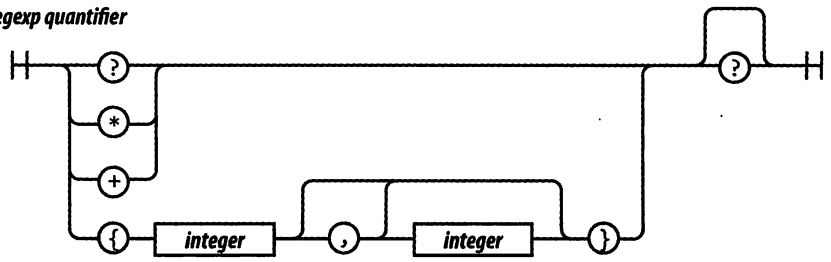
字符类内部的转义规则和正则表达式因子的相比稍有不同。此处的`[\b]`是退格符。下面是在字符类中需要被转义的特殊字符：

```
- / [ \ ] ^
```

正则表达式量词
Regexp Quantifier

正则表达式因子可以用一个正则表达式量词后缀来决定这个因子应该被匹配的次数。包围在一对花括号中的一个数字表示这个因子应该被匹配的次数。所以，/www/匹配的和/w{3}/一样，{3,6}会匹配 3、4、5 或 6 次，{3,}会匹配 3 次或更多次。

77



?等同于{0,1}，*等同于{0,}，+则等同于{1,}。

如果只有一个量词，表示趋向于进行贪婪性匹配，即匹配尽可能多的副本直至达到上限。如果这个量词附加一个后缀?，则表示趋向于进行非贪婪匹配，即只匹配必要的副本就好。一般情况下最好坚持使用贪婪性匹配。

第 8 章

78

方法

Methods

他虽疯，但却有他的一套理论。

——威廉·莎士比亚，《丹麦王子，哈姆雷特的悲剧》（*The Tragedy of Hamlet, Prince of Denmark*）

JavaScript 包含了一套小型的可用在标准类型上的标准方法集。

Array

array.concat(*item*...)

concat 方法产生一个新数组，它包含一份 *array* 的浅复制（shallow copy）并把一个或多个参数 *item* 附加在其后。如果参数 *item* 是一个数组，那么它的每个元素会被分别添加。后面你还会看到和它功能类似的 *array*.push(*item*...)方法。

```
var a = ['a', 'b', 'c'];
var b = ['x', 'y', 'z'];
var c = a.concat(b, true);
// c 变成 ['a', 'b', 'c', 'x', 'y', 'z', true]
```

array.join(*separator*)

join 方法把一个 *array* 构造成一个字符串。它先把 *array* 中的每个元素构造成一个字符串，接着用一个 *separator* 分隔符把它们连接在一起。默认的 *separator* 是逗号','。要想做到无间隔的连接，我们可以使用空字符串作为 *separator*。

如果你想把大量的字符串片段组装成一个字符串，把这些片段放到一个数组中并用 join 方法连接起来通常比用 +元素运算符连接这些片段要快[译注1]。

译注 1：作者编写本书的时候，IE6/7 在浏览器市场还占据着大量的市场份额。对于 IE6/7，使用 Array.join()连接大量字符串的效率确实优于使用 + 元素运算符。但目前主流的浏览器，包括 IE8 以后的版本，都对 + 元素运算符连接字符串做了特别优化，性能已经显著高于 Array.join()。目前在大多数情况下，建议连接字符串首选使用+元素运算符。更为详细的内容请参考《高性能网站建设进阶指南》中字符串优化相关章节。

```
var a = ['a', 'b', 'c'];
a.push('d');
var c = a.join('');    // c 是 'abcd';
```

79 *array*.pop()

pop 和 push 方法使得数组 *array* 可以像堆栈（stack）一样工作。pop 方法移除 *array* 中的最后一个元素并返回该元素。如果该 *array* 是 empty，它会返回 undefined。

```
var a = ['a', 'b', 'c'];
var c = a.pop();     // a 是 ['a', 'b'] & c 是 'c'
```

pop 可以像这样实现：

```
Array.method('pop', function () {
    return this.splice(this.length - 1, 1)[0];
});
```

array.push(*item*...)

push 方法把一个或多个参数 *item* 附加到一个数组的尾部。和 concat 方法不同的是，它会修改 *array*，如果参数 *item* 是一个数组，它会把参数数组作为单个元素整个添加到数组中，并返回这个 *array* 的新长度值。

```
var a = ['a', 'b', 'c'];
var b = ['x', 'y', 'z'];
var c = a.push(b, true);
// a 是 ['a', 'b', 'c', ['x', 'y', 'z'], true]
// c 是 5
```

push 可以像这样实现：

```
Array.method('push', function () {
    this.splice.apply(
        this,
        [this.length, 0].
            concat(Array.prototype.slice.apply(arguments)));
    return this.length;
});
```

array.reverse()

reverse 方法反转 *array* 里的元素的顺序，并返回 *array* 本身：

```
var a = ['a', 'b', 'c'];
var b = a.reverse();
// a 和 b 都是 ['c', 'b', 'a']
```

array.shift()

shift 方法移除数组 *array* 中的第 1 个元素并返回该元素。如果这个数组 *array* 是空的，它会返回 undefined。shift 通常比 pop 慢得多：

```
var a = ['a', 'b', 'c'];
var c = a.shift();    // a 是 ['b', 'c'] & c 是 'a'
```

shift 可以这样实现：

```
Array.method('shift', function () {
    return this.splice(0, 1)[0];
});
```

array.slice(*start, end*)

80

slice 方法对 *array* 中的一段做浅复制。首先复制 *array*[*start*]，一直复制到 *array*[*end*] 为止。*end* 参数是可选的，默认值是该数组的长度 *array*.length。如果两个参数中的任何一个是负数，*array*.length 会和它们相加，试图让它们成为非负数。如果 *start* 大于等于 *array*.length，得到的结果将是一个新的空数组。千万别把 slice 和 splice 弄混了。字符串也有一个同名的方法，参见本章后面的 *string*.slice 。

```
var a = ['a', 'b', 'c'];
var b = a.slice(0, 1);    // b 是 ['a']
var c = a.slice(1);       // c 是 ['b', 'c']
var d = a.slice(1, 2);    // d 是 ['b']
```

array.sort(*comparefn*)

sort 方法对 *array* 中的内容进行排序。它不能正确地给一组数字排序：

```
var n = [4, 8, 15, 16, 23, 42];
n.sort();
// n 是 [15, 16, 23, 4, 42, 8]
```

JavaScript 的默认比较函数把要被排序的元素都视为字符串。它尚未足够智能到在比较这些元素之前先检测它们的类型，所以当它比较这些数字的时候，会把它们转化为字符串，于是得到一个错得离谱的结果。

幸运的是，你可以使用自己的比较函数来替换默认的比较函数。你的比较函数应该接受两个参数，并且如果这两个参数相等则返回 0，如果第 1 个参数应该排列在前面，则返回一个负数，如果第 2 个参数应该排列在前面，则返回一个正数。（这或许使编程老前辈们想起了 FORTRAN II 的算术 IF 语句[译注2]。）

译注 2：FORTRAN 中的算术 IF 语句，形如：if(i)label1,label2,label3，i<0 执行 label1，i=0 执行 label2，i>0 执行 label3。新版本的 FORTRAN 已不再支持算术 IF 语句。

```
n.sort(function (a, b) {
    return a - b;
});
// n 是 [4, 8, 15, 16, 23, 42];
```

上面这个函数可以使数字正确排序，但它不能使字符串排序。如果我们想要给任何包含简单值的数组排序，必须要做更多的工作：

```
var m = ['aa', 'bb', 'a', 4, 8, 15, 16, 23, 42];
m.sort(function (a, b) {
    if (a === b) {
        return 0;
    }
    if (typeof a === typeof b) {
        return a < b ? -1 : 1;
    }
    return typeof a < typeof b ? -1 : 1;
});
// m 是 [4, 8, 15, 16, 23, 42, 'a', 'aa', 'bb']
```

如果大小写不重要，你的比较函数应该在比较之前先将两个运算数转化为小写。此外请参见本章后面的 *string*.localeCompare。

如果有一个更智能的比较函数，我们也可以使对象数组排序。为了让这个事情更能满足一般的情况，我们将编写一个构造比较函数的函数：

81

```
// by 函数接受一个成员名字符串作为参数，
// 并返回一个可以用来对包含该成员的对象数组进行排序的比较函数。
var by = function (name) {
    return function (o, p) {
        var a, b;
        if (typeof o === 'object' && typeof p === 'object' && o && p) {
            a = o[name];
            b = p[name];
            if (a === b) {
                return 0;
            }
            if (typeof a === typeof b) {
                return a < b ? -1 : 1;
            }
            return typeof a < typeof b ? -1 : 1;
        } else {
            throw {
                name: 'Error',
                message: 'Expected an object when sorting by ' + name
            };
        }
    };
};

var s = [
    {first: 'Joe',   last: 'Besser'},
    {first: 'Moe',   last: 'Howard'},
    {first: 'Joe',   last: 'DeRita'},
```

```
    {first: 'Shemp', last: 'Howard'},
    {first: 'Larry', last: 'Fine'},
    {first: 'Curly', last: 'Howard'}
];
s.sort(by('first'));    // s 是 [
//    {first: 'Curly', last: 'Howard'},
//    {first: 'Joe',   last: 'DeRita'},
//    {first: 'Joe',   last: 'Besser'},
//    {first: 'Larry', last: 'Fine'},
//    {first: 'Moe',   last: 'Howard'},
//    {first: 'Shemp', last: 'Howard'}
// ]
```

sort 方法是不稳定的[译注3]，所以下面的调用：

```
s.sort(by('first')).sort(by('last'));
```

不能保证产生正确的序列。如果你想基于多个键值进行排序，你需要再次做更多的工作。我们可以修改 by 函数，让其可以接受第 2 个参数，当主要的键值产生一个匹配的时候，另一个 compare 方法将被调用以决出高下。

```
// by 函数接受一个成员名字符串和一个可选的次要比较函数作为参数，
// 并返回一个可以用来对包含该成员的对象数组进行排序的比较函数。
// 当 o[name] 和 p[name]相等时，次要比较函数被用来决出高下。
var by = function (name, minor) {
    return function (o, p) {
        var a, b;
        if (o && p && typeof o === 'object' && typeof p === 'object') {
            a = o[name];
            b = p[name];
            if (a === b) {
                return typeof minor === 'function' ? minor(o, p) : 0;
            }
            if (typeof a === typeof b) {
                return a < b ? -1 : 1;
            }
            return typeof a < typeof b ? -1 : 1;
        } else {
            throw {
                name: 'Error',
                message: 'Expected an object when sorting by ' + name;
            };
        }
    };
};

s.sort(by('last', by('first')));    // s 是 [
```

82

译注 3：排序的稳定性是指排序后数组中的相等值的相对位置没有发生改变，而不稳定性排序则会改变相等值的相对位置。详细内容请参见 *http://zh.wikipedia.org/wiki/排序算法*。JavaScript 的 sort 方法的稳定性根据不同浏览器的实现而不一致。可参见 *http://developer.mozilla.org/Cn/Core_JavaScript_1.5_Reference/Global_Objects/Array/Sort* 中的介绍。

```
//    {first: 'Joe',   last: 'Besser'},
//    {first: 'Joe',   last: 'DeRita'},
//    {first: 'Larry', last: 'Fine'},
//    {first: 'Curly', last: 'Howard'},
//    {first: 'Moe',   last: 'Howard'},
//    {first: 'Shemp', last: 'Howard'}
// ]
```

array.splice(*start, deleteCount, item...*)

splice 方法从 *array* 中移除一个或多个元素，并用新的 *item* 替换它们。参数 *start* 是从数组 *array* 中移除元素的开始位置。参数 *deleteCount* 是要移除的元素个数。如果有额外的参数，那些 *item* 会插入到被移除元素的位置上。它返回一个包含被移除元素的数组。

splice 最主要的用处是从一个数组中删除元素。千万不要把 splice 和 slice 弄混了：

```
var a = ['a', 'b', 'c'];
var r = a.splice(1, 1, 'ache', 'bug');
// a是['a', 'ache', 'bug', 'c']
// r是['b']
```

splice 可以像这样实现：

```
Array.method('splice', function (start, deleteCount) {
    var max = Math.max,
        min = Math.min,
        delta,
        element,
        insertCount = max(arguments.length - 2, 0),
        k = 0,
        len = this.length,
        new_len,
        result = [],
        shift_count;

    start = start || 0;
    if (start < 0) {
        start += len;
    }
    start = max(min(start, len), 0);
    deleteCount = max(min(typeof deleteCount === 'number' ?
            deleteCount : len, len - start), 0);
    delta = insertCount - deleteCount;
    new_len = len + delta;
    while (k < deleteCount) {
        element = this[start + k];
        if (element !== undefined) {
            result[k] = element;
        }
        k += 1;
    }
    shift_count = len - start - deleteCount;
    if (delta < 0) {
        k = start + insertCount;
```

83

```
            while (shift_count) {
                this[k] = this[k - delta];
                k += 1;
                shift_count -= 1;
            }
            this.length = new_len;
        } else if (delta > 0) {
            k = 1;
            while (shift_count) {
                this[new_len - k] = this[len - k];
                k += 1;
                shift_count -= 1;
            }
            this.length = new_len;
        }
        for (k = 0; k < insertCount; k += 1) {
            this[start + k] = arguments[k + 2];
        }
        return result;
    });
```

array.unshift(*item...*)

unshift 方法像 push 方法一样，用于把元素添加到数组中，但它是把 *item* 插入到 *array* 的开始部分而不是尾部。它返回 *array* 的新的 length[译注4]：

```
var a = ['a', 'b', 'c'];
var r = a.unshift('?', '@');
// a 是 ['?', '@', 'a', 'b', 'c']
// r 是 5
```

unshift 可以像这样实现：

84

```
Array.method('unshift', function () {
    this.splice.apply(this,
        [0, 0].concat(Array.prototype.slice.apply(arguments)));
    return this.length;
});
```

Function

function.apply(*thisArg, argArray*)

apply 方法调用 *function*，传递一个会被绑定到 this 上的对象和一个可选的数组作为参数。apply 方法被用在 apply 调用模式（apply invocation pattern）中（参见第 4 章）。

```
Function.method('bind', function (that) {
```

译注 4：IE6 之前的浏览器中，JScript 引擎对 unshift 方法的实现有错误，它的返回值永远是 undefined。IE7 之后的 IE 系列浏览器已经修正了这个错误。

```
// 返回一个函数，调用这个函数就像调用那个对象的一个方法。

    var method = this,
        slice = Array.prototype.slice,
        args = slice.apply(arguments, [1]);
    return function () {
        return method.apply(that,
            args.concat(slice.apply(arguments, [0])));
    };
});

var x = function () {
    return this.value;
}.bind({value: 666});
alert(x()); // 666
```

Number

number.toExponential(*fractionDigits*)

toExponential 方法把这个 *number* 转换成一个指数形式的字符串。可选参数 *fractionDigits* 控制其小数点后的数字位数。它的值必须在 0～20：

```
document.writeln(Math.PI.toExponential(0));
document.writeln(Math.PI.toExponential(2));
document.writeln(Math.PI.toExponential(7));
document.writeln(Math.PI.toExponential(16));
document.writeln(Math.PI.toExponential());

// 结果

3e+0
3.14e+0
3.1415927e+0
3.1415926535897930e+0
3.141592653589793e+0
```

85

number.toFixed(*fractionDigits*)

toFixed 方法把这个 *number* 转换成为一个十进制数形式的字符串。可选参数 *fractionDigits* 控制其小数点后的数字位数。它的值必须在 0～20，默认为 0：

```
document.writeln(Math.PI.toFixed(0));
document.writeln(Math.PI.toFixed(2));
document.writeln(Math.PI.toFixed(7));
document.writeln(Math.PI.toFixed(16));
document.writeln(Math.PI.toFixed());

// 结果
```

```
3
3.14
3.1415927
3.1415926535897930
3
```

number.toPrecision(*precision*)

`toPrecision` 方法把这个 *number* 转换成为一个十进制数形式的字符串。可选参数 *precision* 控制数字的精度。它的值必须在 0～21：

```
document.writeln(Math.PI.toPrecision(2));
document.writeln(Math.PI.toPrecision(7));
document.writeln(Math.PI.toPrecision(16));
document.writeln(Math.PI.toPrecision());

// 结果

3.1
3.141593
3.141592653589793
3.141592653589793
```

number.toString(*radix*)

`toString` 方法把这个 *number* 转换成为一个字符串。可选参数 *radix* 控制基数。它的值必须在 2～36。默认的 *radix* 是以 10 为基数的。*radix* 参数最常用的是整数，但是它可以用任意的数字。

在最普通的情况下，*number*.`toString()` 可以更简单地写为 `String(` *number* `)`：

```
document.writeln(Math.PI.toString(2));
document.writeln(Math.PI.toString(8));
document.writeln(Math.PI.toString(16));
document.writeln(Math.PI.toString());

// 结果

11.001001000011111101101010100010001000010110100011
3.1103755242102643
3.243f6a8885a3
3.141592653589793
```

86

Object

object.hasOwnProperty(*name*)

如果这个 *object* 包含一个名为 *name* 的属性，那么 `hasOwnProperty` 方法返回 `true`。原型

链中的同名属性是不会被检查的。这个方法对 *name* 就是“hasOwnProperty”时不起作用，此时会返回 false：

```
var a = {member: true};
var b = Object.create(a);               // 来自第 3 章
var t = a.hasOwnProperty('member');     // t 是 true
var u = b.hasOwnProperty('member');     // u 是 false
var v = b.member;                       // v 是 true
```

RegExp

regexp.exec(*string*)

exec 方法是使用正则表达式的最强大（和最慢）的方法。如果它成功地匹配 *regexp* 和字符串 *string*，它会返回一个数组。数组中下标为 0 的元素将包含正则表达式 *regexp* 匹配的子字符串。下标为 1 的元素是分组 1 捕获的文本，下标为 2 的元素是分组 2 捕获的文本，依此类推。如果匹配失败，它会返回 null。

如果 *regexp* 带有一个 g 标识（全局标识），事情会变得更加复杂。查找不是从这个字符串的起始位置开始，而是从 *regexp*.lastIndex（初始值为 0）位置开始。如果匹配成功，那么 *regexp*.lastIndex 将被设置为该匹配后第一个字符的位置。不成功的匹配会重置 *regexp*.lastIndex 为 0。

这就允许你通过循环调用 exec 去查询一个匹配模式在一个字符串中发生了几次。有两件事情需要注意。如果你提前退出了这个循环，再次进入这个循环前必须把 *regexp*.lastIndex 重置到 0。而且，^ 因子仅匹配 *regexp*.lastIndex 为 0 的情况。

```
// 把一个简单的 HTML 文本分解为标签和文本。
//  (entityify 方法请参见 string.replace)

// 对每个标签和文本，都产生一个数组包含如下元素：
// [0] 整个匹配的标签和文本
// [1] / (斜杠)，如果有的话
// [2] 标签名
// [3] 属性，如果有任何属性的话

var text = '<html><body bgcolor=linen><p>' +
       'This is <b>bold<\/b>!<\/p><\/body><\/html>';
var tags = /[^<>]+|<(\/?)([A-Za-z]+)([^<>]*)>/g;
var a, i;

while ((a = tags.exec(text))) {
    for (i = 0; i < a.length; i += 1) {
        document.writeln(('// [' + i + '] ' + a[i]).entityify());
    }
    document.writeln();
```

```
}

// 结果:

// [0] <html>
// [1]
// [2] html
// [3]

// [0] <body bgcolor=linen>
// [1]
// [2] body
// [3]  bgcolor=linen

// [0] <p>
// [1]
// [2] p
// [3]

// [0] This is
// [1] undefined
// [2] undefined
// [3] undefined

// [0] <b>
// [1]
// [2] b
// [3]

// [0] bold
// [1] undefined
// [2] undefined
// [3] undefined

// [0] </b>
// [1] /
// [2] b
// [3]

// [0] !
// [1] undefined
// [2] undefined
// [3] undefined

// [0] </p>
// [1] /
// [2] p
// [3]

// [0] </body>
// [1] /
// [2] body
// [3]

// [0] </html>
// [1] /
```

```
// {2} html
// [3]
```

regexp.test(*string*)

test 方法是使用正则表达式的最简单（和最快）的方法。如果该 *regexp* 匹配 *string*，它返回 true；否则，它返回 false。不要对这个方法使用 g 标识：

```
var b = /&.+;/.test('frank &amp; beans');
// b 是 true
```

test 可以像这样实现：

```
RegExp.method('test', function (string) {
    return this.exec(string) !== null;
});
```

String

string.charAt(*pos*)

charAt 方法返回在 *string* 中 *pos* 位置处的字符。如果 *pos* 小于 0 或大于等于字符串的长度 *string*.length，它会返回空字符串。JavaScript 没有字符类型（character type）。这个方法返回的结果是一个字符串：

```
var name = 'Curly';
var initial = name.charAt(0);    // initial 是 'C'
```

charAt 可以像这样实现：

```
String.method('charAt', function (pos) {
    return this.slice(pos, pos + 1);
});
```

string.charCodeAt(*pos*)

charCodeAt 方法同 charAt 一样，只不过它返回的不是一个字符串，而是以整数形式表示的在 string 中的 *pos* 位置处的字符的字符码位。如果 *pos* 小于 0 或大于等于字符串的长度 *string*.length，它返回 NaN。

```
var name = 'Curly';
var initial = name.charCodeAt(0);   // initial 是 67
```

string.concat(*string...*)

concat 方法把其他的字符串连接在一起来构造一个新的字符串。它很少被使用，因为用 + 运算符更为方便：

```
var s = 'C'.concat('a', 't');   // s 是 'Cat'
```

string.indexOf(*searchString, position*)

indexOf 方法在 *string* 内查找另一个字符串 *searchString*。如果它被找到，返回第 1 个匹配字符的位置，否则返回 −1。可选参数 *position* 可设置从 *string* 的某个指定的位置开始查找：


```
var text = 'Mississippi';
var p = text.indexOf('ss');        // p 是 2
p = text.indexOf('ss', 3);         // p 是 5
p = text.indexOf('ss', 6);         // p 是 -1
```

string.lastIndexOf(*searchString, position*)

lastIndexOf 方法和 indexOf 方法类似，只不过它是从该字符串的末尾开始查找而不是从开头：

```
var text = 'Mississippi';
var p = text.lastIndexOf('ss');     // p 是 5
p = text.lastIndexOf('ss', 3);      // p 是 2
p = text.lastIndexOf('ss', 6);      // p 是 5
```

string.localeCompare(*that*)

localeCompare 方法比较两个字符串。如何比较字符串的规则没有详细说明。如果 *string* 比字符串 *that* 小，那么结果为负数。如果它们是相等的，那么结果为 0。这类似于 *array*.sort 比较函数的约定：

```
var m = ['AAA', 'A', 'aa', 'a', 'Aa', 'aaa'];
m.sort(function (a, b) {
    return a.localeCompare(b);
});
// m (在某些 locale 设置下) 是 ['a', 'A', 'aa', 'Aa', 'aaa', 'AAA']
```

string.match(*regexp*)

match 方法让字符串和一个正则表达式进行匹配。它依据 g 标识来决定如何进行匹配。如果没有 g 标识，那么调用 *string*.match(*regexp*)的结果与调用 *regexp*.exec(*string*) 的结果相同。然而，如果 *regexp* 带有 g 标识，那么它生成一个包含所有匹配（除捕获分组之外）的数组：

```
var text = '<html><body bgcolor=linen><p>' +
       'This is <b>bold<\/b>!<\/p><\/body><\/html>';
var tags = /[^<>]+|<(\/?)([A-Za-z]+)([^<>]*)>/g;
var a, i;
```

```
a = text.match(tags);
for (i = 0; i < a.length; i += 1) {
    document.writeln(('// [' + i + '] ' + a[i]).entityify());
}

// 结果:

// [0] <html>
// [1] <body bgcolor=linen>
// [2] <p>
// [3] This is
// [4] <b>
// [5] bold
// [6] </b>
// [7] !
// [8] </p>
// [9] </body>
// [10] </html>
```


string.replace(*searchValue, replaceValue*)

replace 方法对 *string* 进行查找和替换操作，并返回一个新的字符串。参数 *searchValue* 可以是一个字符串或一个正则表达式对象。如果它是一个字符串，那么 *searchValue* 只会在第 1 次出现的地方被替换，所以下面的代码结果是 "mother-in_law"：

```
var result = "mother_in_law".replace('_', '-');
```

这或许令你失望。

如果 *searchValue* 是一个正则表达式并且带有 g 标识，它会替换所有的匹配。如果没有带 g 标识，它会仅替换第 1 个匹配。

replaceValue 可以是一个字符串或一个函数。如果 *replaceValue* 是一个字符串，字符 $ 拥有特别的含义：

```
// 捕获括号中的 3 个数字

var oldareacode = /\((\d{3})\)/g;
var p = '(555)666-1212'.replace(oldareacode, '$1-');
// p 是 '555-666-1212'
```

美元符号序列	替换对象
$$	$
$&	整个匹配的文本
$number	分组捕获的文本
$`	匹配之前的文本
$'	匹配之后的文本

如果 *replaceValue* 是一个函数，那么每遇到一次匹配函数就会被调用一次，而该函数返回的字符串会被用做替换文本。传递给这个函数的第 1 个参数是整个被匹配的文本，第 2 个参数是分组 1 捕获的文本，再下一个参数是分组 2 捕获的文本，依此类推：

```
String.method('entityify', function () {

    var character = {
        '<' : '&lt;',
        '>' : '&gt;',
        '&' : '&amp;',
        '"' : '&quot;'
    };

// 返回 string.entityify 方法，它返回调用替换方法的结果。
// 它的 replaceValue 函数返回在一个对象中查找一个字符的结果。
// 这种对象的用法通常优于 switch 语句。
    return function () {
        return this.replace(/[<>&"]/g, function (c) {
            return character[c];
        });
    };
}());
alert("<&>".entityify());  // &lt;&amp;&gt;
```


string.search(*regexp*)

search 方法和 indexOf 方法类似，只是它接受一个正则表达式对象作为参数而不是一个字符串。如果找到匹配，它返回第 1 个匹配的首字符位置，如果没有找到匹配，则返回 −1。此方法会忽略 g 标识，且没有 *position* 参数：

```
var text = 'and in it he says "Any damn fool could';
var pos = text.search(/["']/);    // pos 是 18
```

string.slice(*start, end*)

slice 方法复制 *string* 的一部分来构造一个新的字符串。如果 *start* 参数是负数，它将与 *string*.length 相加。*end* 参数是可选的，且默认值是 *string*.length。如果 *end* 参数是负数，那么它将与 *string*.length 相加。*end* 参数等于你要取的最后一个字符的位置值加上 1。要想得到从位置 p 开始的 n 个字符，就用 *string*.slice(p, p + n)。同类的方法请参见本章随后要介绍的 *string*.substring 和之前介绍的 *array*.slice。

```
var text = 'and in it he says "Any damn fool could';
var a = text.slice(18);
// a 是 '"Any damn fool could'
var b = text.slice(0, 3);
// b 是 'and'
var c = text.slice(-5);
// c 是 'could'
```

```
var d = text.slice(19, 32);
// d 是 'Any damn fool'
```

string.split(*separator, limit*)

split 方法把这个 *string* 分割成片段来创建一个字符串数组。可选参数 *limit* 可以限制被分割的片段数量。*separator* 参数可以是一个字符串或一个正则表达式。

如果 *separator* 是一个空字符，会返回一个单字符的数组：

```
var digits = '0123456789';
var a = digits.split('', 5);
// a 是 ['0', '1', '2', '3', '4']
```

否则，此方法会在 *string* 中查找所有 *separator* 出现的地方。分隔符两边的每个单元文本都会被复制到该数组中。此方法会忽略 g 标识：

```
var ip = '192.168.1.0';
var b = ip.split('.');
// b 是 ['192', '168', '1', '0']

var c = '|a|b|c|'.split('|');
// c 是 ['', 'a', 'b', 'c', '']
```

92

```
var text = 'last,  first ,middle';
var d = text.split(/\s*,\s*/);
// d 是 [
//    'last',
//    'first',
//    'middle'
// ]
```

有一些特例须特别注意。来自分组捕获的文本会被包含在被分割后的数组中：

```
var e = text.split(/\s*(,)\s*/);
// e 是 [
//    'last',
//    ',',
//    'first',
//    ',',
//    'middle'
// ]
```

当 *separator* 是一个正则表达式时，有一些 JavaScript 的实现[译注5]在输出数组中会排除掉空字符串：

```
var f = '|a|b|c|'.split(/\|/);
// 在一些系统中，f 是 ['', 'a', 'b', 'c', ''],
// 在另外一些系统中，f 是 ['a', 'b', 'c']。
```

译注 5：根据译者的测试，在主流浏览器中，只有 IE8 及之前版本的 IE 浏览器会在输出的数组结果中排除空字符串，IE9 已经修复了这个问题。

string.substring(*start, end*)

substring 的用法和 slice 方法一样，只是它不能处理负数参数。没有任何理由去使用 substring 方法。请用 slice 替代它。

string.toLocaleLowerCase()

toLocaleLowerCase 方法返回一个新字符串，它使用本地化的规则把这个 *string* 中的所有字母转换为小写格式。这个方法主要用在土耳其语上，因为在土耳其语中'I'转换为'ı'，而不是'i'。

string.toLocaleUpperCase()

toLocaleUpperCase 方法返回一个新字符串，它使用本地化的规则把这个 *string* 中的所有字母转换为大写格式。这个方法主要用在土耳其语上，因为在土耳其语中'i'转换为'İ'，而不是'I'。

string.toLowerCase()

toLowerCase 方法返回一个新的字符串，这个 *string* 中的所有字母都被转换为小写格式。

string.toUpperCase()

93

toUpperCase 方法返回一个新的字符串，这个 *string* 中的所有字母都被转换为大写格式。

String.fromCharCode(*char...*)

String.fromCharCode 函数根据一串数字编码返回一个字符串。

```
var a = String.fromCharCode(67, 97, 116);
// a 是 'Cat'
```

第 9 章

代码风格

Style

好一串啷啷噪噪的头衔！
——威廉·莎士比亚，《亨利六世上篇》（*The First Part of Henry the Sixth*）

电脑程序是人类制造出来的最复杂的玩意儿。程序通常由很多部分组成，表现为函数、语句和表达式，它们必须准确无误地按照顺序排列。而运行时行为（runtime behavior）几乎和实现它的程序没有什么相似之处。在软件的产品生命周期中，它们通常都会被修改。把一个正确的程序转化为另一个同样正确但风格不同的程序，是一个极具挑战性的过程。

优秀的程序拥有一个前瞻性的结构，它会预见到在未来才可能需要的修改，但不会让其成为过度的负担。优秀的程序还会具备一种清晰的表达方式。如果一个程序被表达得很好，那么我们就能更加容易地去理解它，以便成功地改造或修补它。

这些观点适用于所有的编程语言，而对 JavaScript 来说尤为如此。JavaScript 的弱类型和过度的容错性导致程序质量无法在编译时获得保障，所以为了弥补，我们应该按照严格的规范进行编码。

JavaScript 包含大量脆弱的、问题缠身的特性，它们会妨碍我们写出优秀的程序。显然我们应该避免 JavaScript 里那些糟糕的特性。但别吃惊，或许我们也应该避免掉那些平时很有用但偶尔也有害的特性。这样的特性让人既爱又恨，然而，一旦避免它们，就能避免一大类潜在的错误。

对于一个组织机构来说，软件的长远价值和代码库的质量成正比。在程序的生命周期里，会经历很多人的测试、使用和修改。如果一个程序能很清楚地传达它的结构和特性，那么当它在并不遥远的将来被修改时，它被破坏的可能性就小很多。

JavaScript 代码经常被直接发布。它应该自始至终具备发布质量，要干净利落。通过在一个清晰且始终如一的风格下编写，你的程序会变得易于阅读。

程序员会无休止地讨论良好的风格是由什么构成的。大多数程序员会坚持他们过去的应用经验，比如他们在学校或在他们第一份工作时学到的流行的风格。他们中的一些人拥有高薪的工作，但完全没有代码风格的意识。这是否证明了风格其实根本不重要？就算风格不重要，风格之间是否有优劣之分呢？

事实证明代码风格在编程中是很重要的，就像文字风格对于写作很重要一样。好的风格让代码能更好地被阅读。

电脑程序有时候被认为不是用来读的，所以只要它能工作，写成怎样是不重要的。但是结果证明，如果程序可读性强，它正常运行的可能性，以及是否准确按照我们的意图去工作的可能性也显著增强。它还决定了软件在其生命周期中能否进行扩展。如果我们能阅读并且理解程序，那么就有希望去修改和完善它。

贯穿本书，我始终采用一致的风格。我的目的是使代码实例尽可能易于阅读。我使用一致的留白来帮助你理解我的程序的逻辑思路。

我对代码块内容和对象字面量缩进 4 个空格。我放了一个空格在 `if` 和 `(` 之间，以致 `if` 不会看起来像一个函数调用。只有真的是在调用时，我才使 `(` 和其前面的符号相毗连。我在除了 `.` 和 `[` 外的所有中置运算符的两边都放了空格，它们俩无须空格是因为它们有更高的优先级。我在每个逗号和冒号后面都使用一个空格。

我在每行最多放一个语句，在一行里放多条语句可能会被误读。如果一个语句一行放不下，我会在一个冒号或二元运算符后拆开它，这将更好地防止自动插入分号的机制掩盖复制/粘贴的错误（自动插入分号带来的悲剧会在附录 A 里披露）。我给折断后的语句的其余部分多缩进 4 个空格，如果 4 个还不够明显，就缩进 8 个空格（例如在一个 `if` 语句的条件部分插入一个换行符的时候）。

在诸如 `if` 和 `while` 这样结构化的语句里，我始终使用代码块，因为这样会减少出错的概率。我曾看到过：

```
if (a)
    b();
```

变成：

```
if (a)
    b();
    c();
```

这是一个很难被发现的错误。它看起来像是这样：

96

```
if (a) {
    b();
    c();
}
```

但它的本意是：

```
if (a) {
    b();
}
c();
```

看起来想要做一件事情但实际上却在做另一件事情的代码很可能导致 bug。一对花括号可以用很低廉的成本去防止那些需要昂贵的代价才能发现的 bug。

我一直使用 K&R[译注 1] 风格，把 { 放在一行的结尾而不是下一行的开头，因为它会避免 JavaScript 的 `return` 语句中的一个可怕的设计错误。

我的代码包含了一些注释。我喜欢在程序中放入注释来留下一些信息，以后，它将会被那些需要理解我当时思路的人们（也可能是我自己）阅读。有时候觉得注释就像一个时间机器，我用它发送重要的信息给未来的我。

我努力保持注释是最新的。错误的注释甚至可能会使程序更加难以阅读和理解。我不能容忍犯下那样的错误。

我尽量不用类似这样的无用注释去浪费你的时间：

```
i = 0;  // 设置 i 为 0。
```

在 JavaScript 里，我更喜欢用行注释。我把块注释用于正式的文档记录和注释。

我更喜欢使我的程序结构能自我说明（self-illuminating），从而消除对注释的需要。我并非每次都能做到，所以只要我的程序还不尽完美，我就会编写注释。

JavaScript 拥有 C 语言的语法，但它的代码块没有作用域。所以，变量在它们第 1 次使用时被声明的惯例，对 JavaScript 来说却是糟糕的建议。JavaScript 有函数作用域，但是没有块级作用域，所以我在每个函数的开始部分声明我的所有变量。JavaScript 允许变量在它们使用后被声明。那对我来说感觉像是一个错误，我不希望我写的程序看起来像有错误。我希望我的错误被突出显示出来。相似地，我绝不在一个 `if` 的条件部分使用赋值表达式，因为：

```
if (a = b) { ... }
```

可能的本意是：

```
if (a === b) { ... }
```

97 我想要避免那些看起来像有错误的习惯用法。

我绝不允许 switch 语句块中的条件穿越到下一个 case 语句。我曾经在我的代码里发现了一个无意识的“穿越”导致的 bug，而在此之前，我刚刚激情澎湃地做完一次关于如何妙用“穿越”有时很有用的演讲。我很幸运能够从这个教训中有所收获。当我现在评审一门语言的特性的时候，我把注意力放在那些有时很有用但偶尔很危险的特性上。那些是最糟糕的部分，因为我们很难辨别它们是否被正确使用。那是 bug 的藏身之地。

译注 1：K&R 代码风格，因在 Kernighan 与 Ritchie 合著的 *The C Programming Language* 一书中广泛采用而得名。它是最为普遍的 C 语言代码风格。更多具体内容请参见 *http://en.wikipedia.org/wiki/Indent_style#K.26R_style* 和 *http://en.wikipedia.org/wiki/K&R*。

在 JavaScript 的设计、实现和标准化的过程中，质量没有被特别关注。这给使用这门语言的用户增加了避免其缺陷的难度。

JavaScript 为大型程序提供了支持，但它也带有不利于大型程序的形式和习惯用法。举例来说：JavaScript 可以方便地使用全局变量，但随着程序的日益复杂，全局变量逐渐变得问题重重。

对一个脚本应用或工具库，我只用唯一一个全局变量。每个对象都有它自己的命名空间，所以我很容易使用对象去管理代码。使用闭包能提供进一步的信息隐藏，增强我的模块的健壮性[译注 2]。

译注 2：作者的这个思想在 YAHOO 的 JavaScript 库 YUI 中得到了彻底的贯彻。在 YUI 中仅用到两个全局变量：YAHOO 和 YAHOO_config。YUI 的一切都是基于一种模块模式来实现的。具体细节请参见 *http://dancewithnet.com/2007/12/04/a-javascript-module-pattern/*。

第 10 章

优美的特性

Beautiful Features

我让你的脚玷污我的嘴唇，让你的肖像玷污我的眼睛，让你的每一部分玷污我的心，等候着你的答复。你的最忠实的……

——威廉·莎士比亚，《空爱一场》（*Love's Labor's Lost*）

去年我被邀请为 Andy Oram 和 Greg Wilson 的 *Beautiful Code*[译注 1]（O'Reilly 出版）一书写一篇文章，这是一本以计算机程序的表达之美为主题的选集。我负责的章节将介绍 JavaScript，通过那一部分来证明 JavaScript 不虚其名，它的确是抽象、强大且有用的。然而，我想避开不谈浏览器和其他适合使用 JavaScript 的地方。我想要强调其更有分量的内容，以显示它是值得尊敬的语言。

我立即想到 Vaughn Pratt 的自顶向下的运算符优先级解析器[译注 2]（Top Down Operator Precedence parser），我在 JSLint 中运用了它。在计算机的信息处理技术中，解析是一个重要的主题。一门语言是否具备为其自身编写一个编译器的能力，仍然是对这门语言完整性的一个测试。

我想把用 JavaScript 编写的 JavaScript 解析器的全部代码都包含在文章中。但是我的章节仅是 30 章或 40 章之一，我感觉被束缚在那几页短短的篇幅里。更大的困难是大部分读者没有使用 JavaScript 的经验，我也不得不介绍这门语言和它的特色。

所以，我决定提炼这门语言的子集。这样，我就不必解析整个语言，并且也就不需要描述整个语言了。我把这个子集叫做精简的 JavaScript（Simplified JavaScript）。提炼子集并不难：它包括的就是我编写解析器所需要的特性。我在 *Beautiful Code*（《代码之美》）一书中是这样描述的。

> 精简的 JavaScript 里都是好东西，包括以下主要内容。
>
> 函数是顶级对象
>
> 在精简 JavaScript 中，函数是有词法作用域的闭包（lambda）。

译注 1：*Beautiful Code* 是 O'Reilly 2007 年 6 月出版的书籍，参见 *http://oreilly.com/catalog/9780596510046/*。

译注 2：详细内容请参阅作者的文章 *Top Down Operator Precedence parser*——*http://javascript.crockford.com/tdop/tdop.html*。

基于原型继承的动态对象

对象是无类别的。我们可以通过普通的赋值给任何对象增加一个新成员属性。一个对象可以从另一个对象继承成员属性。

对象字面量和数组字面量

这对创建新的对象和数组来说是一种非常方便的表示法。JavaScript 字面量是数据交换格式 JSON 的灵感之源。

子集包含了 JavaScript 精华中最好的部分。尽管它是一门小巧的语言，但它很强大且非常富有表现力。JavaScript 有许多本不应该加入的额外特性，正如你在随后的附录中会看到的那样，它有大量的会带来负面价值的特性。在子集中没有丑陋或糟糕的内容，它们全部都被筛除了。

精简的 JavaScript 不是一个严格的子集。我添加了少许新特性。最简单的是增加了 `pi` 作为一个简单的常量。我这么做是为了证明解析器的一个特性。我也展示了一个更好的保留字策略并证明哪些保留字是多余的。在一个函数中，一个单词不能既被用做变量或参数名，又被用做一个语言特性。你可以让某个单词用在其中之一上，并允许程序员自己选择。这会使一门语言易于学习，因为你没必要知道你不会使用的特性。并且它会使这门语言易于扩展，因为它无须保留更多的保留字来增加新特性。

我也增加了块级作用域。块级作用域不是一个必需的特性，但没有它会让有经验的程序员感到困惑。包含块级作用域是因为我预期解析器可能会被用于解析非 JavaScript 语言，并且那些语言能正确地界定作用域。我编写这个解析器的代码风格不关心块作用域是否可用。我推荐你也采用这种方式来写。

当开始构思本书的时候，我想进一步地发展这个子集，我想展示除了排除低价值特性外，如何通过不做任何改变来获得一个现有的编程语言，并且使它得到有效的改进。

我们看到大量的特性驱动的产品设计，其中特性的成本没有被正确计算。对于用户来说，某些特性可能有一些负面价值，因为它们使产品更加难以理解和使用。我们发现人们想要的产品其实只要能工作即可。事实证明产生恰好可以工作的设计比集合一大串特性的设计要困难得多。

特性有规定成本、设计成本和开发成本，还有测试成本和可靠性成本。特性越多，某个特性出现问题，或者和其他特性相互干扰的可能性就越大。在软件系统中，存储成本是无足轻重的，但在移动应用中，它又变得重要了。它们抬高了电池的效能成本，因为摩尔定律并不适用于电池。

特性有文档成本。每个特性都会让产品指南变得更厚，从而增加了培训成本。只对少数用户有价值的特性增加了所有用户的成本。所以在设计产品和编程语言时，我们希望直接使用核心的精华部分，因为是这些精华创造了大部分的价值。

在我们使用的产品中，总能找到好的部分。我们喜欢简单，追求简洁易用，但是当产品缺乏这种特性时，就要自己去创造它。微波炉有一大堆特性，但是我只会用烹调和定时，使用定时功能就足够麻烦的了。对于特性驱动型的设计，我们唯有靠找出它的精华并坚持使用，才能更好地应对其复杂性。

如果产品和编程语言被设计得仅留下精华，那该有多好。

毒瘤
Awful Parts

那会在一言一行中证明其可怕。
——威廉·莎士比亚，《泰尔亲王佩里克利斯》（*Pericles, Prince of Tyre*）

在本附录中，我会展示 JavaScript 的一些难以避免的问题特性。你必须知道这些问题并准备好应对的措施。

全局变量
Global Variables

在 JavaScript 所有的糟糕特性之中，最为糟糕的一个就是它对全局变量的依赖。全局变量就是在所有作用域中都可见的变量。全局变量在微型程序中可能会带来方便，但随着程序变得越来越大，它们很快变得难以管理。因为一个全局变量可以被程序的任何部分在任意时间修改，它们使得程序的行为变得极度复杂。在程序中使用全局变量降低了程序的可靠性。

全局变量使得在同一个程序中运行独立的子程序变得更难。如果某些全局变量的名称碰巧和子程序中的变量名称相同，那么它们将会相互冲突，可能导致程序无法运行，而且通常难以调试。

许多编程语言都有全局变量。例如，Java 中的 `public static` 成员属性就是全局变量。JavaScript 的问题不仅在于它允许使用全局变量，而且在于它依赖全局变量。JavaScript 没有链接器（linker），所有的编译单元都载入一个公共全局对象中。

共有 3 种方式定义全局变量。第 1 种是在任何函数之外放置一个 `var` 语句：

```
var foo = value;
```

第 2 种是直接给全局对象添加一个属性。全局对象是所有全局变量的容器。在 Web 浏览器里，全局对象名为 `window`：

```
window.foo = value;
```

102 第3种是直接使用未经声明的变量，这被称为隐式的全局变量：

```
foo = value;
```

这种方式本来是为方便初学者，有意让变量在使用前无须声明。遗憾的是，忘记声明变量成了一个非常普遍的错误。JavaScript 的策略是让那些忘记预先声明的变量成为全局变量，这导致查找 bug 非常困难。

作用域
Scope

JavaScript 的语法来源于 C。在所有其他类似 C 语言风格的语言里，一个代码块（括在一对花括号中的一组语句）会创造一个作用域。代码块中声明的变量在其外部是不可见的。JavaScript 采用了这样的块语法，却没有提供块级作用域：代码块中声明的变量在包含此代码块的函数的任何位置都是可见的。这让有其他语言编码经验的程序员们大为意外。

在大多数语言中，一般来说，声明变量的最好的地方是在第一次用到它的地方。但这种做法在 JavaScript 里反而是一个坏习惯，因为它没有块级作用域。更好的方式是在每个函数的开头部分声明所有变量。

自动插入分号
Semicolon Insertion

JavaScript 有一个自动修复机制，它试图通过自动插入分号来修正有缺损的程序。但是，千万不要指望它，它可能会掩盖更为严重的错误。

有时它会不合时宜地插入分号。请考虑在 `return` 语句中自动插入分号导致的后果。如果一个 `return` 语句返回一个值，这个值表达式的开始部分必须和 `return` 位于同一行：

```
return
{
    status: true
};
```

这看起来是要返回一个包含 `status` 成员元素的对象。遗憾的是，自动插入分号让它变成了返回 `undefined`。自动插入分号导致程序被误解，却没有任何警告提醒。如果把{放在上一行的尾部而不是下一行的头部就可以避免该问题：

```
return {
    status: true
};
```

保留字
Reserved Words

下面的单词在 JavaScript 里被保留：

```
abstract boolean break byte case catch char class const continue debugger default
delete do double else enum export extends false final finally float for function goto
if implements import in instanceof int interface long native new null package private
protected public return short static super switch synchronized this throw throws
transient true try typeof var volatile void while with
```

这些单词中的大多数并没有在语言中使用。

它们不能被用来命名变量或参数[译注1]。当保留字被用做对象字面量的键值时，它们必须被引号括起来。它们不能被用在点表示法中，所以有时必须使用括号表示法：

```
var method;                 // ok
var class;                  // 非法
object = {box: value};      // ok
object = {case: value};     // 非法
object = {'case': value};   // ok
object.box = value;         // ok
object.case = value;        // 非法
object['case'] = value;     // ok
```

Unicode

JavaScript 设计之初，Unicode 预期最多会有 65536 个字符。但从那以后它的容量慢慢增长到了拥有一百万个字符。

JavaScript 的字符是 16 位的，那足以覆盖原有的 65536 个字符（现在被称为基本多文种平面[译注2]（Basic Multilingual Plane））。剩下的百万字符中的每一个都可以用一对字符来表示。Unicode 把一对字符视为一个单一的字符。而 JavaScript 认为一对字符是两个不同的字符。

译注 1：各个浏览器在对保留字的使用限制上不同版本有不同的处理，根据译者的测试，比如文中的代码：

```
object = {case: value};
```

在目前主流浏览器的主流版本中都是合法的，但在老版本中则可能不合法。而类似 `int/long/float` 等保留字，在各浏览器中都可以用做变量名及对象字面量的键值。尽管如此，在这些场合依然不建议使用任何保留字。

译注 2：基本多文种平面（Basic Multilingual Plane，BMP）或称第 0 平面（Plane 0），是 Unicode 中的一个编码区段。编码从 U+0000 至 U+FFFF。更多详细内容请参见 *http://zh.wikipedia.org/wiki/基本多文种平面*。

typeof

typeof 运算符返回一个用于识别其运算数类型的字符串。所以：

```
typeof 98.6
```

返回'number'。遗憾的是：

```
typeof null
```

返回'object'而不是'null'。这简直太糟糕了。其实，有更简单也更好的检测 null 的方式：

```
my_value === null
```

104

一个更大的问题是检测对象的值。typeof 不能辨别出 null 与对象，但你可以像下面这样做，因为 null 值为假，而所有对象值为真：

```
if (my_value && typeof my_value === 'object') {
    // my_value 是一个对象或数组!
}
```

相关的内容，请参阅后面的小节“NaN”和“伪数组”。

在对正则表达式的类型识别上，各种 JavaScript 的实现不太一致。对于下面的代码：

```
typeof /a/
```

一些实现会返回 'object'，而其他的返回 'function'[译注3]。如果返回 'regexp' 可能会更有用些，但标准不允许那么做。

parseInt

parseInt 是一个把字符串转换为整数的函数。它在遇到非数字时会停止解析，所以 parseInt("16") 与 parseInt("16 tons") 产生相同的结果。如果该函数会提醒我们出现了额外文本就好了，但它不会那么做。

如果该字符串第 1 个字符是 0，那么该字符串会基于八进制而不是十进制来求值。在八进制中，8 和 9 不是数字，所以 parseInt("08") 和 parseInt("09") 都产生 0 作为结果。这个错误会导致程序解析日期和时间时出现问题。幸运的是，parseInt 可以接受一个基数作为参数，如此一来 parseInt("08", 10) 结果为 8。我建议你总是加上这个基数参数。

译注 3：经译者测试，在对正则表达式执行 typeof 操作时，主流浏览器中的 IE/Firefox/Opera 都返回'object'，而 Safari，在 3.x 版本系列中，返回的是'function'，5.x 版本后与其他浏览器调整为保持一致。

+

+ 运算符可以用于加法运算或字符串连接。它究竟会如何执行取决于其参数的类型。如果其中一个运算数是一个空字符串，它会把另一个运算数转换成字符串并返回。如果两个运算数都是数字，它返回两者之和。否则，它把两个运算数都转换为字符串并连接起来。这个复杂的行为是 bug 的常见来源。如果你打算用 + 去做加法运算，请确保两个运算数都是整数。

浮点数
Floating Point

二进制的浮点数不能正确地处理十进制的小数，因此 0.1 + 0.2 不等于 0.3。这是 JavaScript 中最经常被报告的 bug，并且它是遵循二进制浮点数算术标准（IEEE 754）[译注4]而有意导致的结果。这个标准对很多应用都是适合的，但它违背了大多数你在中学所学过的关于数字的知识。幸运的是，浮点数中的整数运算是精确的，所以小数表现出来的错误可以通过指定精度来避免。 105

举例来说，美元可以通过乘以 100 而全部转成美分，然后就可以准确地将美分相加。它们的和可以再除以 100 转换回美元。当人们计算货币时当然会期望得到精确的结果。

NaN

NaN 是 IEEE 754 中定义的一个特殊的数量值。它表示的不是一个数字，尽管下面的表达式返回的是 true：

```
typeof NaN === 'number'    // true
```

该值可能会在试图把非数字形式的字符串转换为数字时产生。例如：

```
+ '0'         // 0
+ 'oops'      // NaN
```

如果 NaN 是数学运算中的一个运算数，那么结果就是 NaN。所以，如果你有一个公式链产生出 NaN 的结果，那肯定要么其中一个输入项是 NaN，要么在某个地方产生了 NaN。

你可以对 NaN 进行检测。正如我们之前所见，typeof 不能辨别数字和 NaN，而且 NaN 也不等同于它自己。所以，下面的代码结果令人惊讶：

```
NaN === NaN    // false
NaN !== NaN    // true
```

译注 4：IEEE 754 是最广泛使用的浮点数运算标准，为许多 CPU 与浮点运算器所采用。更多详细内容请参见 *http://zh.wikipedia.org/wiki/IEEE_754*。

JavaScript 提供了一个 isNaN 函数，可以辨别数字与 NaN：

```
isNaN(NaN)        // true
isNaN(0)          // false
isNaN('oops')     // true
isNaN('0')        // false
```

判断一个值是否可用做数字的最佳方法是使用 isFinite 函数，因为它会筛除掉 NaN 和 Infinity。遗憾的是，isFinite 会试图把它的运算数转换为一个数字，所以，如果值事实上不是一个数字，它就不是一个好的测试。你可以这样定义自己的 isNumber 函数：

```
var isNumber = function isNumber(value) {
                return typeof value === 'number' && isFinite(value);
}
```

伪数组
Phony Arrays

JavaScript 没有真正的数组。这也不全是坏事。JavaScript 的数组确实非常容易使用。你不必给它们设置维度，而且它们永远不会产生越界（out-of-bounds）错误。但它们的性能相比真正的数组可能相当糟糕。

106 typeof 运算符不能辨别数组和对象。要判断一个值是否为数组，你还需要检查它的 constructor 属性：

```
if (my_value && typeof my_value === 'object' &&
        my_value.constructor === Array) {
    // my_value 是一个数组。
}
```

上面的检测对于在不同帧或窗口创建的数组将会给出 false。当数组有可能在其他的帧中被创建时，下面的检测更为可靠：

```
if (Object.prototype.toString.apply(my_value) === '[object Array]'){
    // my_value 确实是一个数组!
}
```

arguments 数组不是一个数组，它只是一个有着 length 成员属性的对象。上面的检测会分辨出 arguments 并不是一个数组。

假值
Falsy Values

JavaScript 拥有一组数量奇大的假值，请参见表 A-1。

表 A-1：JavaScript 的众多假值

值	类型
0	Number
NaN（非数字）	Number
''（空字符串）	String
false	Boolean
null	Object
undefined	Undefined

这些值全部都等同于假，但它们是不可互换的。例如，要想确定一个对象是否缺少一个成员属性，这是一种错误的方式：

```
value = myObject[name];
if (value == null) {
   alert(name + ' not found.');
}
```

undefined 是缺失的成员属性的值，但这段代码里用 null 来测试。它使用了会强制转换类型的==运算符（参见附录 B），而不是更可靠的===运算符。有时那两个错误会彼此抵消，有时则不会。

undefined 和 NaN 并不是常量[译注 5]。它们居然是全局变量，而且你可以改变它们的值。那本是不应该的，但事实确实如此。千万不要这样做。 107

hasOwnProperty

在第 3 章中，hasOwnProperty 方法被用做一个过滤器去避开 for in 语句的一个隐患。遗憾的是，hasOwnProperty 是一个方法，而不是一个运算符，所以在任何对象中，它可能会被一个不同的函数甚至一个非函数的值所替换：

```
var name;
another_stooge.hasOwnProperty = null;       // 地雷
for (name in another_stooge) {
    if (another_stooge.hasOwnProperty(name)) { // 触雷
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

译注 5：在 ECMAScript 规范第 5 版中，明确规定了 NaN 和 undefined 为常量，而之前的版本中都未明确规定。经过译者的测试，目前主流浏览器的主流版本，都无法变更 NaN 和 undefined 的值。而 IE8 及以下的 IE 浏览器是可以的。

对象
Object

JavaScript 的对象永远不会是真的空对象，因为它们可以从原型链中取得成员属性。有时候那会带来些麻烦。例如，假设你正在编写一个程序去计算一段文本中每个单词的出现次数。我们可以使用 toLowerCase 方法统一转换文本为小写格式，接着使用 split 方法传一个正则表达式为参数去产生一个单词数组。然后可以遍历该组单词并统计我们看到的每个单词出现的次数：

```
var i;
var word;
var text =
       "This oracle of comfort has so pleased me, " +
       "That when I am in heaven I shall desire " +
       "To see what this child does, " +
       "and praise my Constructor.";

var words = text.toLowerCase( ).split(/[\s,.]+/);
var count = {};
for (i = 0; i < words.length; i += 1) {
    word = words[i];
    if (count[word]) {
        count[word] += 1;
    } else {
        count[word] = 1;
    }
}
```

让我们来研究该结果，count['this']的值为 2，count.heaven 的值是 1，但是 count.constructor 却包含着一个看上去令人不可思议的字符串[译注 6]。其原因在于 count
108 对象继承自 Object.prototype，而 Object.prototype 包含着一个名为 constructor 的成员对象，它的值是一个 Object。+=运算符，就像+运算符一样，当它的运算数不是数字时会执行字符串连接操作而不是做加法。因为该对象是一个函数，所以+=运算符把它转换成一个莫名其妙的字符串，然后再把一个数字 1 加在它的后面。

我们可以采用处理 for in 中的问题的相同方法去避免类似的问题：用 hasOwnProperty 方法检测成员关系，或者查找特定的类型。在当前情形下，我们对似是而非的 count[word] 的测试条件指定得不够具体[译注 7]。我们可以这样写：

```
if (typeof count[word] === 'number') {
```

译注 6：在主流浏览器上，打印 count.constructor 将会返回字符串 function Object() { [native code] }。

译注 7：作者此处的意思是说，因为难以确定哪个单词可能与对象原型链中的属性或方法名重合，所以无法列出充分的测试条件。

糟粕
Bad Parts

现在要请你告诉我，你究竟为了我哪一点坏处而开始爱起我来呢？
——威廉·莎士比亚，《无事生非》（*Much Ado About Nothing*）

在本附录中，我会展示 JavaScript 一些有问题的特性，但我们很容易就能避免它们。通过这些简单的做法，你可以使 JavaScript 成为一门更好的语言，也让你自己成为一个更好的程序员。

==

JavaScript 有两组相等运算符：=== 和!==，以及它们邪恶的孪生兄弟 == 和 !=。=== 和!== 这一组运算符会按照你期望的方式工作。如果两个运算数类型一致且拥有相同的值，那么 === 返回 true，!== 返回 false。而它们邪恶的孪生兄弟只有在两个运算数类型一致时才会做出正确的判断，如果两个运算数是不同的类型，它们试图去强制转换值的类型。转换的规则复杂且难以记忆。这里有一些有趣的例子：

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true

false == 'false'    // false
false == '0'        // true

false == undefined  // false
false == null       // false
null == undefined   // true

' \t\r\n ' == 0     // true
```

== 运算符对传递性[译注1]的缺乏值得我们警惕。我的建议是永远不要使用那对邪恶的孪生兄弟。相反，请始终使用 === 和 !==。如果以上所有的比较使用 === 运算符，结果都是 false。

译注 1：传递性是一种编程约定。可以这么理解：对于任意的引用值 x、y 和 z，如果 x == y 和 y == z 为 true，那么 x == z 为 true。而 JavaScript 中的 == 运算符在某些特例上违背了传递性。

110 with 语句

with Statement

JavaScript 提供了一个 `with` 语句，本意是想用它来快捷地访问对象的属性。不幸的是，它的结果可能有时不可预料，所以应该避免使用它。

下面的语句：

```
with (obj) {
    a = b;
}
```

和下面的代码做的是同样的事情：

```
if (obj.a === undefined) {
    a = obj.b === undefined ? b : obj.b;
} else {
    obj.a = obj.b === undefined ? b : obj.b;
}
```

所以，它等于这些语句中的某一条：

```
a = b;
a = obj.b;
obj.a = b;
obj.a = obj.b;
```

通过阅读程序代码，你不可能辨别出你会得到的是这些语句中的哪一条。它可能随着程序运行到下一步时发生变化。它甚至可能在程序运行过程中就发生了变化。如果你不能通过阅读程序而了解它将会做什么，你就无法确信它会正确地做你想要做的事情。

`with` 语句在这门语言里存在，本身就严重影响了 JavaScript 处理器的速度，因为它阻断了变量名的词法作用域绑定。它的本意是好的，但如果没有它，JavaScript 语言会更好一点。

eval

`eval` 函数传递一个字符串给 JavaScript 编译器，并且执行其结果。它是一个被滥用得最多的 JavaScript 特性。那些对 JavaScript 语言一知半解的人们最常用到它。例如，如果你知道点表示法，但不知道下标表示法，就可能会这么写：

```
eval("myValue = myObject." + myKey + ";");
```

而不是这么写：

```
myvalue = myObject[myKey];
```

使用 `eval` 形式的代码更加难以阅读。这种形式使得性能显著降低，因为它需要运行编译

器，但也许只是为了执行一个微不足道的赋值语句。它也会让 JSLint（参见附录 C）失效，让此工具检测问题的能力大打折扣。

`eval` 函数还减弱了你的应用程序的安全性，因为它给被求值的文本授予了太多的权力。而且就像 `with` 语句执行的方式一样，它降低了语言的性能。

`Function` 构造器是 `eval` 的另一种形式，同样也应该避免使用它。

浏览器提供的 `setTimeout` 和 `setInterval` 函数，它们能接受字符串参数或函数参数。当传递的是字符串参数时，`setTimeout` 和 `setInterval` 会像 `eval` 那样去处理。同样也应该避免使用字符串参数形式。

continue 语句
continue Statement

`continue` 语句跳到循环的顶部。我发现一段代码通过重构移除 `continue` 语句之后，性能都会得到改善。

switch 穿越
switch Fall Through

`switch` 语句的由来可以追溯到 FORTRAN IV[译注2]的 `go to` 语句。除非你明确地中断流程，否则每次条件判断后都穿越到下一个 case 条件。

有人曾写信给我，建议 JSLint 应该在一个 case 条件向下穿越到另一个 case 条件时给出一个警告。他指出这是一个非常常见的错误来源，并且它很难通过查看代码发现错误。我回信说完全同意他的意见，但从穿越中得到的紧凑性的好处可以降低它出错的概率。

第二天，他报告说在 JSLint 里有一个错误。它是一个无法正确识别错误的错误。我调查了一番，结果证明是我有一个 case 条件穿越导致的。那一刻，我受到了启发。我不再刻意地使用 case 条件穿越。那条原则使得我们可以更加容易地发现不小心造成的 case 条件穿越。

一门语言最糟糕的特性不是那些一看就知道很危险或者没有价值的特性。那些特性很容易被避免。最糟糕的特性就像带刺的玫瑰，它们是有用的，但也是危险的。

译注 2：FORTRAN 语言最初是由数值计算方面的需要而发展起来的。FORTRAN IV 在 1962 年推出，并开始被广泛使用。更多详细内容请参见 *http://zh.wikipedia.org/wiki/Fortran*。

缺少块的语句
Block-less Statements

If、while、do 或 for 语句可以接受一个括在花括号中的代码块，也可以接受单行语句。单行语句的形式是另一种带刺的玫瑰。它带来的好处是可以节约两个字节，但这是不是一个好处值得商榷。它模糊了程序的结构，使得在随后的操作代码中可能很容易插入错误。例如：

112

```
if (ok)
t = true;
```

可能变成：

```
if (ok)
    t = true;
    advance();
```

它看起来像是要这样：

```
if (ok) {
    t = true;
    advance();
}
```

但实际上它本意却是：

```
if (ok) {
    t = true;
}
advance();
```

貌似在做一件事，但实际上却是在做另一件事的程序是非常难理清楚的。制定严格的规范要求始终使用代码块会使得代码更容易理解。

++ --

递增和递减运算符使得程序员可以用非常简洁的风格去编码。比如在 C 语言中，它们使得用一行代码实现字符串的复制成为可能：

```
for (p = src, q = dest; *p; p++, q++) *q = *p;
```

事实上，这两个运算符鼓励了一种不够谨慎的编程风格。大多数的缓冲区溢出错误所造成的安全漏洞，都是由像这样编码而导致的。

在我自己的实践中，我观察到，当我使用 ++ 和 -- 时，代码往往变得过于拥挤、复杂和隐晦。因此，作为一条原则，我不再使用它们。我想那样会让我的代码风格变得更为整洁。

位运算符
Bitwise Operators

JavaScript 有着与 Java 相同的一套位运算符：

```
&    and 按位与
|    or  按位或
^    xor 按位异或
~    not 按位非
>>   带符号的右位移
>>>  无符号的(用 0 补足的)右位移
<<   左位移
```

在 Java 里，位运算符处理的是整数。JavaScript 没有整数类型，它只有双精度的浮点数。因此，位操作符把它们的数字运算数先转换成整数，接着执行运算，然后再转换回去。在大多数语言中，这些位运算符接近于硬件处理，所以非常快。但 JavaScript 的执行环境一般接触不到硬件，所以非常慢。JavaScript 很少被用来执行位操作。

113

还有，在 JavaScript 程序中，`&` 非常容易被误写为 `&&` 运算符。位运算符出现在 JavaScript 中降低了这门语言的冗余度[译注3]，使得 bug 更容易被隐藏起来。

function 语句对比 function 表达式
The function Statement Versus the function Expression

JavaScript 既有 `function` 语句，同时也有 `function` 表达式。这令人困惑，因为它们看起来好像就是相同的。一个 `function` 语句就是其值为一个函数的 `var` 语句的速记形式。

下面的语句：

```
function foo() {}
```

意思相当于：

```
var foo = function foo() {};
```

在整本书中，我一直使用的是第 2 种形式，因为它能明确表示 `foo` 是一个包含一个函数值的变量。要用好这门语言，理解函数就是数值是很重要的。

`function` 语句在解析时会发生被提升的情况。这意味着不管 `function` 被放置在哪里，它会被移动到被定义时所在作用域的顶层。这放宽了函数必须先声明后使用的要求，而我认为这会导致混乱。在 `if` 语句中使用 `function` 语句也是被禁止的。结果表明大多数的浏览器都允许在 `if` 语句里使用 `function` 语句，但它们在解析时的处理上各不相同。这就造

译注 3：关于语言的冗余度，请参见 *http://en.wikipedia.org/wiki/Redundancy_(language)*。

成了可移植性的问题。

一个语句不能以一个函数表达式开头，因为官方的语法假定以单词 function 开头的语句是一个 function 语句。解决方法就是把函数调用括在一个圆括号之中。

```
(function () {
   var hidden_variable;

   // 这个函数可能对环境有一些影响，但不会引入新的全局变量。
}());
```

114

类型的包装对象
Typed Wrappers

JavaScript 有一套类型的包装对象。例如：

```
new Boolean(false)
```

会返回一个对象，该对象有一个 valueOf 方法会返回被包装的值。这其实完全没有必要，并且有时还令人困惑。不要使用 new Boolean、new Number 或 new String。

此外也请避免使用 new Object 和 new Array。可使用 {} 和 [] 来代替。

new

JavaScript 的 new 运算符创建一个继承于其运算数的原型的新对象，然后调用该运算数，把新创建的对象绑定给 this。这给运算数（它应该是一个构造器函数）一个机会在返回给请求者前自定义新创建的对象。

如果你忘记了使用此 new 运算符，你得到的就是一个普通的函数调用，并且 this 被绑定到全局对象，而不是新创建的对象。这意味着当你的函数尝试去初始化新成员属性时它将会污染全局变量。这是一件非常糟糕的事情。而且既没有编译时警告，也没有运行时警告。

按照惯例，打算与 new 结合使用的函数应该以首字母大写的形式命名，并且首字母大写的形式应该只用来命名那些构造器函数。这个约定帮助我们进行区分，便于我们发现那些 JavaScript 语言自身经常忽略但却会带来昂贵代价的错误。

一个更好的应对策略就是根本不去使用 new。

void

在很多语言中，void 是一种类型，表示没有值。而在 JavaScript 里，void 是一个运算符，它接受一个运算数并返回 undefined。这没有什么用，而且令人非常困惑。应避免使用它。

附录 C

115

JSLint

难道我的眼睛耳朵都有了毛病？

——威廉·莎士比亚，《错误的喜剧》（*The Comedy of Errors*）

在 C 语言还是一门新生的编程语言的时候，有一些常见的编程错误不能被原始的编译器捕获，所以一个名为 `lint` 的辅助程序被开发出来，它可以通过扫描源文件来查找问题。

随着 C 语言趋于成熟，语言的定义被强化以消除一些不安全因素，并且编译器的预警能力得到了加强。所以 `lint` 不再需要了。

JavaScript 是一门“年轻”的语言。它最初被设计出来用于执行 web 页面上那些用 Java 完成过于笨拙的小型任务。然而，JavaScript 是一门能力很强的语言，现在已经被应用于一些大型项目之中。但对大型项目而言，很多本意是希望提高这门语言易用性的特性却成了麻烦。于是催生出一个针对 JavaScript 的 `lint`：JSLint，一个 JavaScript 语法检查器和校验器。

JSLint 是一个 JavaScript 的代码质量工具。它读取源文本并进行扫描。如果发现问题，它会返回一个消息描述该问题并指明该问题在源文件中的大概位置。被发现的问题往往是语法错误，但也不一定全是。JSLint 还会查看一些代码风格惯例及结构上的问题。它不会证明你的程序是否正确，只是提供了另一种视角帮助你辨认问题。

JSLint 定义了 JavaScript 的一个特定子集，一个比《ECMAScript 语言规范》第 3 版（ECMA-262）的定义更严格的语言。该子集与第 9 章所推荐的代码风格密切相关。

JavaScript 是一门（表面看起来）散漫的语言，但在它里面隐藏着一门更好的优雅的语言。JSLint 帮助你在这门更好的语言中编程，并尽量避免越界。

JSLint 可以在 *http://www.JSLint.com/*找到。

未定义的变量和函数

116

Undefined Variables and Functions

JavaScript 最大的问题是它对全局变量的依赖，特别是隐式的全局变量。如果一个变量没有被显式声明（通常采用 `var` 语句），那么 JavaScript 就假定该变量是全局的。这可能掩盖名称拼写错误或其他的问题。

JSLint 期望所有的变量和函数在使用或调用前都已被声明。这样它就可以探测隐式的全局变量。这也是一种良好的编码实践，因为它使得程序更容易阅读。

有时候，一个文件会依赖在别处定义的全局变量与函数。你可以在文件中包含一个注释，列出那些程序依赖的但却没有定义在你的程序或脚本文件中的全局函数与对象，标注给 JSLint 知道。

全局声明注释可以用来列出所有你明确用做全局变量的名字。JSLint 可以用此信息去辨别拼写错误和被遗忘的 `var` 声明。一个全局声明可能看起来像这样：

```
/*global getElementByAttribute, breakCycles, hanoi */
```

全局声明以 `/*global` 字样开头。注意在 `g` 之前没有空白。你可以使用任意多个 `/*global` 注释，但它们必须出现在指定的变量被使用之前。

你可以预先定义一些全局属性(参见后面的 C.3 节)。选择“模拟浏览器(Assume a browser)”（`browser`）选项可以预定义由 Web 浏览器提供的标准全局属性，比如 `window`、`document` 和 `alert`。选择“模拟 Rhino（Assume Rhino)”（`rhino`）选项可以预定义由 Rhino[译注1] 环境提供的全局属性。选择“模拟 YAHOO Widget（Assume a YAHOO Widget)”（`widget`）选项可以预定义由 YAHOO! Widgets[译注2] 环境提供的全局属性。

成员属性
Members

因为 JavaScript 是一门弱类型的动态对象语言，所以它无法在编译时确定属性名是否拼写正确。JSLint 为此提供了一些帮助。

在它的报告的底部[译注3]，JSLint 显示了一个 `/*members*/` 注释。它包含所有在点表示法、下标表示法中使用到的名字和字符串字面量，涵盖对象字面量中的成员属性。你可以从该列表中检查拼写错误。只用到过一次的成员属性的名字以斜体显示，让你更容易辨认出拼写错误。

你可以复制 `/*members*/` 注释到你的脚本文件中。JSLint 会依照该列表来检查所有属性名字的拼写。这样就可以让 JSLint 根据你提供的线索查找拼写错误。

译注 1：Rhino 是一个开源的完全由 Java 编写的 JavaScript 引擎，它的官方网站在 *http://www.mozilla.org/rhino/*。

译注 2：Yahoo! Widgets，又称 Yahoo! Widgets Engine，是 Yahoo!所推出的一套 Widget 引擎，使用了 JavaScript 及 XML 等技术，可在 Windows 及 Mac OS X 上运行（摘自 *http://zh.wikipedia.org/wiki/Yahoo!_Widgets*）。

译注 3：作者这里指的是使用在线版本的 JSLint（*http://www.jslint.com/*）得到的分析报表的底部。

```
/*members doTell, iDoDeclare, mercySakes,
   myGoodness, ohGoOn, wellShutMyMouth */
```

选项
Options

JSLint在运行时接受一个选项对象作为参数，通过它可以限定你能接受的JavaScript的子集。你也可以在脚本的源码中设置那些选项。

选项的规范看起来像这样：

```
/*jslint nomen: true, evil: false */
```

选项规定以/*jslint开头。注意在j之前没有空白。后面包含一串“名/值”对，名称是JSLint的选项名，值为true或false。在源码中指定的选项的优先级高于选项对象参数。所有的选项默认值为false。表C-1列出了在JSLint中可用的选项。

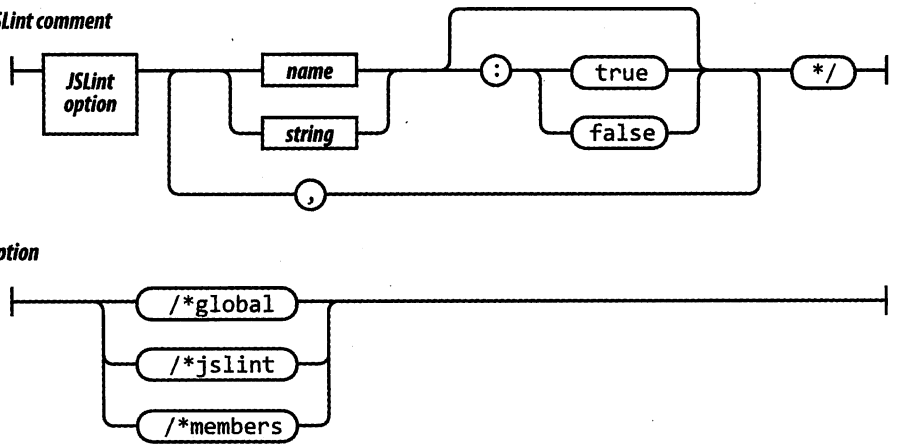
表C-1：JSLint的选项

选项	含义
adsafe	如果为true，表示强制实施ADsafe.org的规则[译注4]
bitwise	如果为true，表示不允许使用位运算符
browser	如果为true，表示预定义浏览器的通用全局属性
cap	如果为true，表示允许大写字母的HTML
continue	如果为true，表示允许使用continue语句
css	如果为true，表示允许CSS hacks或错误语法
debug	如果为true，表示允许使用debugger语句
devel	如果为true，表示预定义开发调试常用的全局属性（如console、alert）
eqeq	如果为true，表示允许使用==和!==
es5	如果为true，表示允许使用ES5规范中的语法
evil	如果为true，表示允许eval
forin	如果为true，表示允许未过滤的for in语句
fragment	如果为true，表示允许HTML片段
newcap	如果为true，表示允许构造器函数首字母非大写
node	如果为true，表示预定义Node环境下的全局属性
nomen	如果为true，表示检查名字
on	如果为true，表示允许在HTML标签中注册事件处理器
passfail	如果为true，表示在遇到第1个错误时停止扫描

译注4：ADsafe.org定义了一个专用于第三方广告代码的脚本子集，以防止恶意的广告脚本。其官方站点是*http://www.adsafe.org/*。

续表

plusplus	如果为 true，表示允许使用 ++ 和 --
regexp	如果为 true，表示允许在正则表达式中使用有安全风险的 . 和[^...]
rhino	如果为 true，表示预定义 Rhino 环境下的全局属性
sub	如果为 true，表示允许使用低效的下标记法[译注 5]
undef	如果为 true，表示允许使用未定义的变量或函数
vars	如果为 true，表示允许单个函数中存在多行 var 语句
white	如果为 true，表示应用严格的空白规则[译注 6]
widget	如果为 true，表示预定义 Yahoo! Widgets 的全局属性



分号
Semicolon

JavaScript 使用类似 C 语言风格的语法，它要求使用分号去界定语句。JavaScript 试图通过自动插入分号机制使得分号可以省略。这是危险的。

像 C 语言一样，JavaScript 有++、-- 和 (运算符，它们可以前置也可以后置。这须通过分号去消除歧义。

在 JavaScript 中，换行符有时被当做空白，有时也能起到分号的作用。这样的结果是用一个

译注 5：作者认为，提取对象属性时，点表示法比下标表示法效率更高。此选项更为详细的解释，请参考 *http://stackoverflow.com/questions/2448367/jslint-tolerate-inefficient-subscripting*。

译注 6：关于严格的空白规则，请参照 *http://javascript.crockford.com/code.html* 中 Whitespace 部分。

歧义去替代了另一个歧义[译注7]。

JSLint 期望在除了 for、function、if、switch、try 和 while 之外的每个语句后面都跟着一个分号。JSLint 不期望看到不必要的分号或空语句。

换行
Line Breaking

为进一步地防范被自动插入分号机制掩盖的错误，JSLint 期望代码很长的语句只在下面所列的这些标点符号字符或运算符之后换行：

```
, . ; : { } ( [ = < > ? ! + - * / % ~ ^ | &
== != <= >= += -= *= /= %= ^= |= &= << >> || &&
=== !== <<= >>= >>> >>>=
```

JSLint 不期望看到代码很长的语句在标识符、字符串、数字、闭合符或后置运算符之后换行：

```
) ] ++ --
```

JSLint 允许你开启“允许随意换行（Tolerate sloppy line breaking）”（laxbreak）选项。 119

自动插入分号机制可能掩盖“复制/粘贴”导致的错误。如果你总是在运算符之后换行，那么 JSLint 可以更好地去发现这些错误。

逗号
Comma

逗号运算符可能导致过于复杂的表达式，也可能掩盖一些编程错误。

JSLint 期望看到逗号被用做一个分隔符，而不是一个运算符（除了在 for 语句的初始化部分和增量部分中以外）。它不期望看到数组字面量中省略掉一些元素。多余的逗号不应该被使用，它不应该出现在数组字面量或对象字面量的最后一个元素之后，因为它可能会被一些浏览器错误地解析。

译注7：在 ECMAScript 的规范中，换行符被称为“行结束符（Line Terminators）”，它会影响自动插入分号机制的处理过程。规范的“7.9 Automatic Semicolon Insertion”一节详细地说明了自动插入分号机制和众多范例，将帮助读者理解本节的内容。

必需的代码块
Required Blocks

JSLint 期望 `if` 和 `for` 语句由代码块构成，即语句都由一对花括号（{}）围起来。

JavaScript 允许一个 `if` 语句写成如下的样子：

```
if (condition)
    statement;
```

众所周知，在由许多程序员协作开发共享代码的项目中，这种形式容易导致错误。所以 JSLint 期望使用代码块：

```
if (condition) {
    statements;
}
```

经验表明，这种形式更可靠。

被禁止的代码块
Forbidden Blocks

在很多语言中，代码块具有作用域。在一个代码块中引入的变量，在该代码块之外是不可见的。

在 JavaScript 中，代码块并没有作用域。JavaScript 中只有函数作用域。在一个函数中的任意位置引入的变量在该函数中到处可见。JavaScript 的代码块让有经验的程序员感到迷惑，并且导致错误，因为他们本来熟悉的语法这次仿佛中了邪。

JSLint 期望只有 `function`、`if`、`switch`、`while`、`for`、`do` 和 `try` 语句使用代码块[译注 8]。有一个例外就是在 `else` 或 `for in` 语句中的 `if` 语句可以不使用代码块。

120

表达式语句
Expression Statements

表达式语句被期望是赋值、函数 / 方法调用或 `delete` 操作。所有其他的表达式语句都被认为是错误的。

译注 8：JavaScript 的语法允许代码块不与 `if/for` 等语句一起使用而单独存在，但因为没有块级作用域，这样的代码块没有任何意义。JSLint 遇到单独的代码块时，总是会当做 JSON 对象去解析。

for in 语句

for in Statement

for in 语句可以用来遍历对象的所有属性的名字。糟糕的是，它也会遍历出所有从原型链中继承而来的成员属性。这带来了糟糕的副作用：或许你只对数据成员感兴趣，但它却提供了一些方法函数。

每个 for in 语句的主体都应该被包围在一个用于过滤的 if 语句中。if 语句可以选择某种特定的类型或某个范围内的值，它可以排除函数，或者排除从原型继承而来的属性。例如：

```
for (name in object) {
    if (object.hasOwnProperty(name)) {
      ....
    }
}
```

switch 语句

switch Statement

在 switch 语句中常见的错误是忘记在每个 case 语句后放一个 break 语句，结果导致意外的穿越。JSLint 期望在下一个 case 或 default 语句之前有下面这些语句的其中一条：break、return 或 throw。

var 语句

var Statement

JavaScript 允许 var 定义语句出现在函数内部的任意位置。JSLint 的要求则更为严格。

JSLint 期望的如下：

- 一个 var 语句只会被声明一次，并且它会在使用前被声明。
- 函数会在使用前被声明。
- 参数不会用 var 再声明一次。

JSLint 不期望的如下：

- 把 arguments 当做变量名来用 var 语句声明。
- 在代码块中定义变量。这是因为 JavaScript 没有块级作用域。这可能带来意想不到的后果，所以在函数体的顶部定义所有的变量。

121 with 语句

with Statement

with 语句的本意是提供一个访问深层嵌套对象成员的快捷方式。糟糕的是，当设置新成员属性时，它的行为非常糟糕。所以永远不要使用 with 语句，而用 var 去代替它。

JSLint 不期望看到 with 语句。

=

JSLint 不期望在 if 或 while 语句的条件部分看到赋值语句。因为像下面这样的代码：

```
if (a = b) {
    ...
}
```

本意更可能是：

```
if (a == b) {
    ...
}
```

==和!=

== and !=

== 和 != 运算符在执行比较前会做强制类型转换。这很糟糕，因为它导致 ' \f\r \n\t ' == 0 的结果为 true。从而可能掩盖因类型引发的错误。

当和如下所列的任意一个值进行比较时，总是使用 === 或 !==运算符，这一对运算符不会做强制类型转换：

```
0 '' undefined null false true
```

如果你希望做强制类型转换，那么就使用简易格式。对如下的形式：

```
(foo != 0)
```

就直接使用：

```
(foo)
```

对于：

```
(foo == 0)
```

就替代为：

```
(!foo)
```

122

使用 === 与 !== 运算符始终是首选的。虽然有一个“允许==和!=”（eqeq）的选项，但不建议使用这个选项。

标签
Labels

JavaScript 允许任何语句都拥有一个标签，并且标签有一个单独的名称空间[译注 9]。JSLint 更为严格。

JSLint 期望标签只用在会与 break 语句进行交互的下列语句中：switch、while、do 和 for。JSLint 期望标签有别于变量和参数。

不可达代码[译注 10]
Unreachable Code

JSLint 期望 return、break、continue 或 throw 语句的后面会紧接一个}、case 或 default 语句。

混乱的正负号
Confusing Pluses and Minuses

JSLint 期望 + 不会跟在 + 或 ++ 的后面，而 - 不会跟在 - 或 -- 的后面。一个位置不当的空格可能将 + + 变成 ++，这样的错误很难被发现。使用圆括号可以避免混淆。

译注 9：作者此处的意思是，在 JavaScript 中，标签标识符与命名变量及函数的标识符使用的是不同的命名空间，以避免冲突。关于标签的规范描述，请参考 *http://www.ecmainternational.org/publications/standards/Ecma-262.htm* 中标签的部分。

译注 10：在计算机编程领域中，不可达代码（Unreachable Code），又称为死代码（dead code），指的是程序源代码中永远不会被执行到的代码。详细内容请参见 *http://en.wikipedia.org/wiki/Unreachable_code*。

++ and --

++（递增）运算符和 --（递减）运算符晦涩诡异的写法只会让代码更糟糕。它们是导致病毒和其他安全威胁的第二大元凶（第一是架构缺陷）。如果你真的允许使用这些运算符，可以开启 JSLint 的 `plusplus` 选项。

位运算符
Bitwise Operators

JavaScript 没有整数类型，但它有位运算符。位运算符把它们的运算数从浮点数转换为整数后接着返回，所以它们的效率根本无法和它们在 C 或其他语言中的表现相比。它们很少用于浏览器应用程序。它们和逻辑运算符的相似性可能会掩盖一些编程错误。使用 `bitwise` 选项可以禁止使用这些运算符。

eval 是魔鬼
eval is Evil

`eval` 函数及它的亲戚（`Function`、`setTimeout` 和 `setInterval`）提供了访问 JavaScript 编译器的机会。有时候这是很有用的，但大多数情况下它表明存在着相当糟糕的代码。`eval` 函数是 JavaScript 被误用得最多的特性。

123 void

在大多数类C的语言中，`void`是一种类型。而在JavaScript中，`void`是一个总返回 `undefined` 的前置运算符。JSLint 不期望看到 `void`，因为它令人困惑，而且没什么用。

正则表达式
Regular Expressions

正则表达式以一种简洁而有些神秘的表示法编写。JSLint 会查找可能导致隐患的问题。它也尝试对那些看上去存在歧义的地方提出应该明确进行转义的建议。

JavaScript 的正则表达式字面量的语法重载了 / 字符。为避免混淆，JSLint 期望出现在一个正则表达式之前的字符是 (、=、: 或 ,。

构造器函数和 new 运算符

Constructors and new

构造器函数是被设计成结合 new 运算符一起使用的函数。new 运算符基于该函数的原型创建一个新对象，并且把该对象绑定到该函数隐含的 this 参数上。如果你忽略使用 new，新的对象不会被创建，并且 this 会被绑定到全局对象上。这是一个严重的错误。

JSLint 强制约定构造器函数必须以首字母大写的形式命名。JSLint 不期望看到一个首字母大写的函数在没有前置 new 的情况下被调用，JSLint 也不期望看到与 new 连用的函数名不是以大写字母开头的。

JSLint 不期望看到这些封装形式：new Number、new String 或 new Boolean。

JSLint 不期望看到 new Object（使用对象字面量{}代替）。

JSLint 不期望看到 new Array（使用数组字面量[]代替）。

忽略的检查

Not Looked For

JSLint 不会通过流程分析去确定变量在使用前是否已经赋了值。这是因为变量的默认值（undefined）对很多应用程序来说是合理的。

JSLint 不会去做任何形式的全局分析。它不会尝试去确定与 new 连用的函数是否是真正的构造器函数（除了强制首字母大写的约定）。

HTML

124

JSLint 能够处理 HTML 文本。它可以查看包含在<script>...</script>标签内的 JavaScript 内容及事件处理程序。它也查看 HTML 内容，寻找那些已知的影响 JavaScript 执行的问题。

- 所有标签的名字必须小写。
- 所有标签中，应该匹配结束标签（比如</p>）的标签必须有一个结束标签。
- 所有的标签都被正确嵌套。
- 对字面上的 < 符号必须使用 <实体字符。

JSLint 并不严格到要与 XHTML 的规定完全一致，但比一般的浏览器要严格。

JSLint 也会检查字符串字面量中 </ 的出现。你应该总是写为 <\/替换它。多余的反斜杠会被 JavaScript 编译器忽略，但不会被 HTML 解析器忽略。像这样的小窍门本来没有必要，可是现在只能这么做。

JSLint 有一个选项允许使用大写格式的标签名。此外，还有一个选项允许使用行内的 HTML 事件处理程序。

JSON

JSLint 也能检查 JSON 数据结构是否格式良好。如果 JSLint 看的第 1 个字符是 { 或 [，那么它就严格地实施 JSON 规则。参见附录 E。

报告

Report

如果 JSLint 能够完成扫描，它会生成一个函数报告。它会为每个函数列出以下内容：

- 函数开始的行号。
- 函数的名称。至于匿名函数，JSLint 将会“猜测”它的名字。
- 参数列表。
- Closure（闭包）：被内部函数用到的变量及在该函数中声明的参数。
- Variables（变量）：在函数中声明且只被该函数用到的变量。
- Unused（未使用的）：在函数中声明但未使用的变量。这可能是一个错误的征兆。
- Outer（外部的）：在本函数用到的，但在另一个函数中声明的变量。
- Global（全局的）：这个函数用到的全局变量。
- Lablel（标签）：这个函数用到的语句标签。

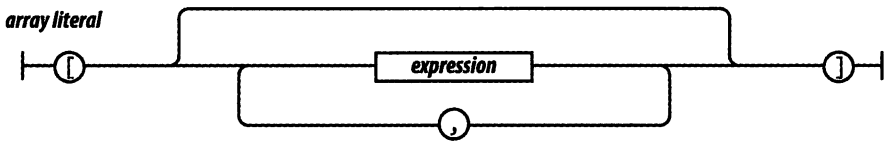
该报告还包括一个清单，列出所有被使用到的成员属性的名字。

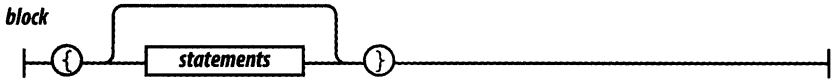
语法图

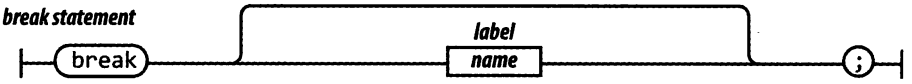
Syntax Diagrams

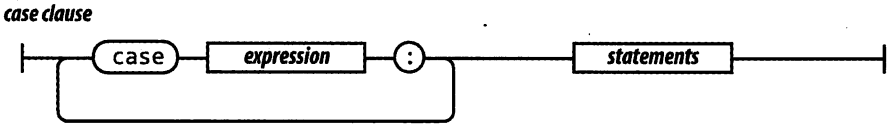
你这苦恼的化身，你在用表情向我们说话吗？

——威廉·莎士比亚，《泰特斯·安德洛尼克斯》（*The Tragedy of Titus Andronicus*）



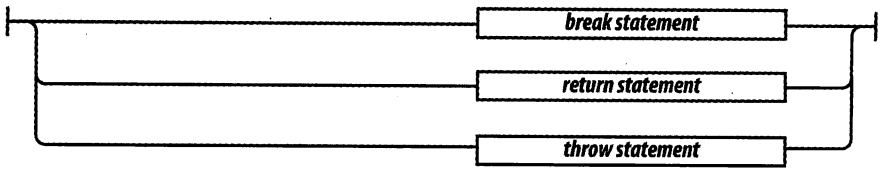




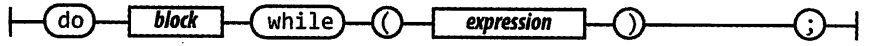


126

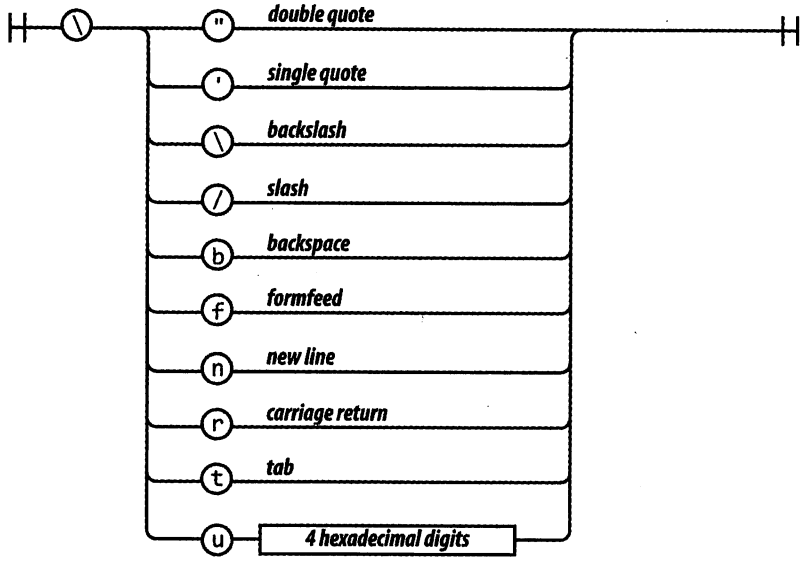
disruptive statement



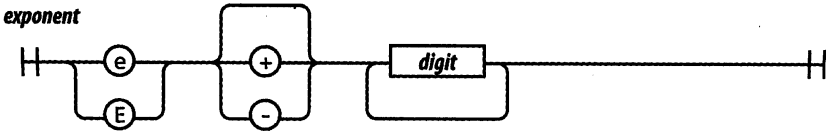
do statement



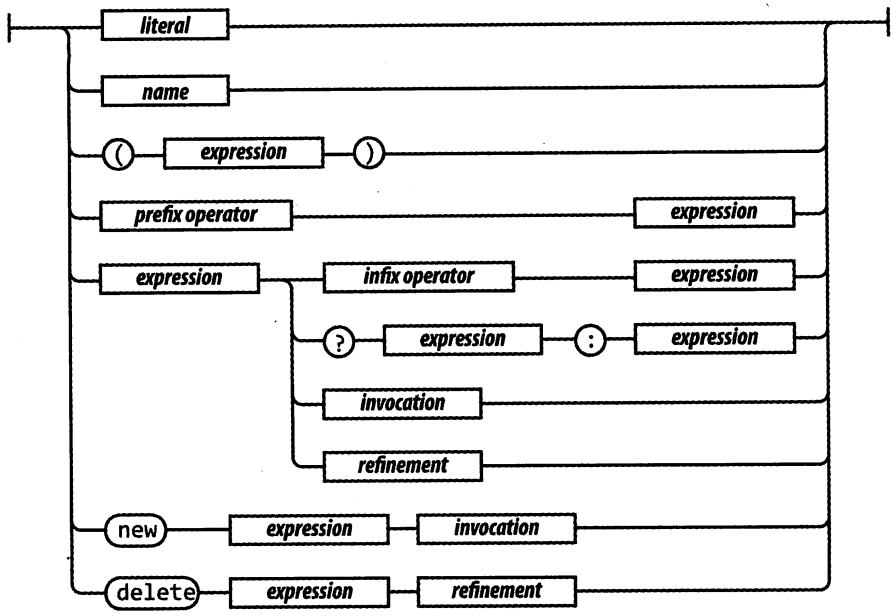
escaped character



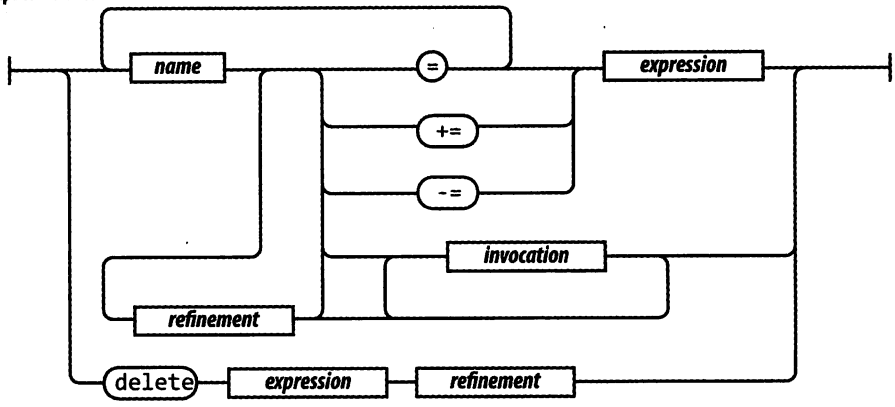
exponent



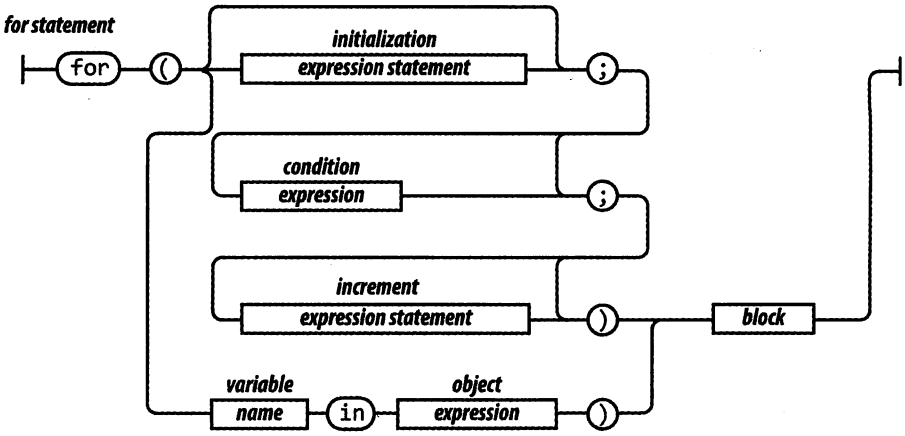
expression

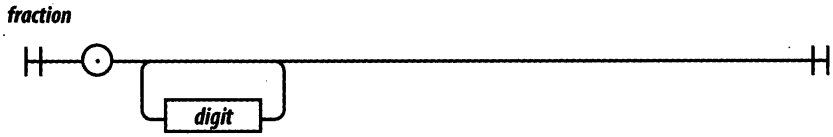


expression statement

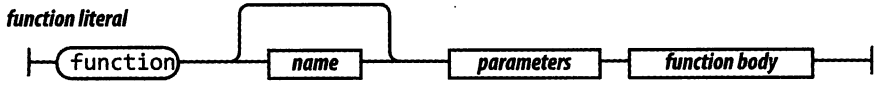


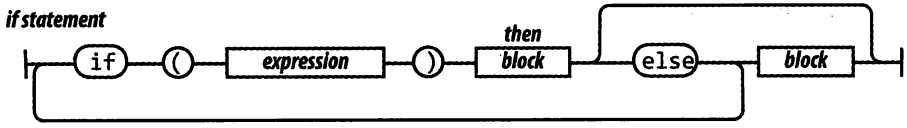
128

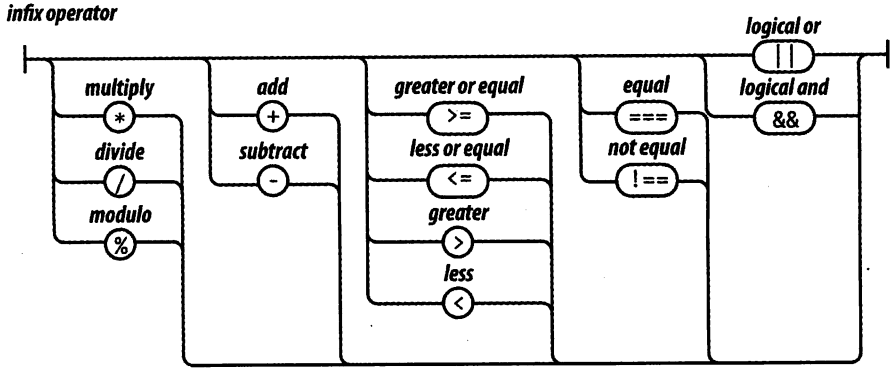

for statement
for
(
initialization
expression statement
;
condition
expression
;
increment
expression statement
)
block
variable
name
in
object
expression
)

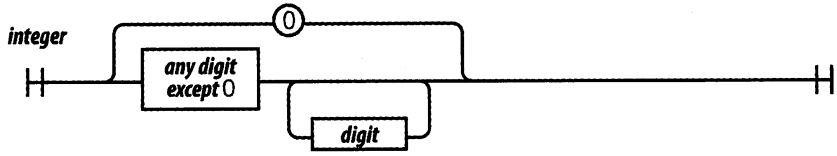

fraction
.
digit

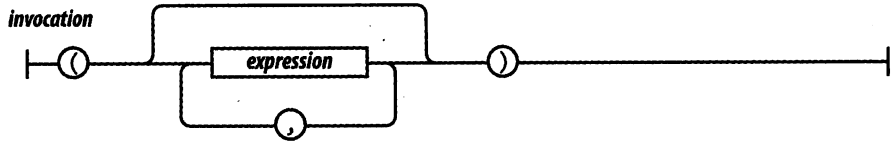

function body
{
var statements
statements
}

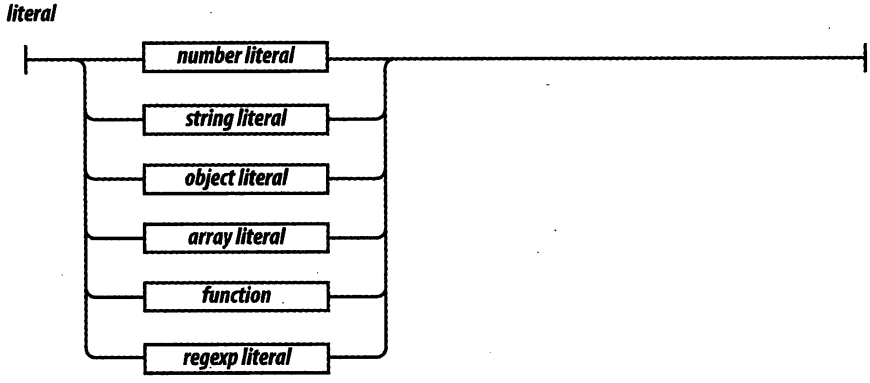

function literal
function
name
parameters
function body

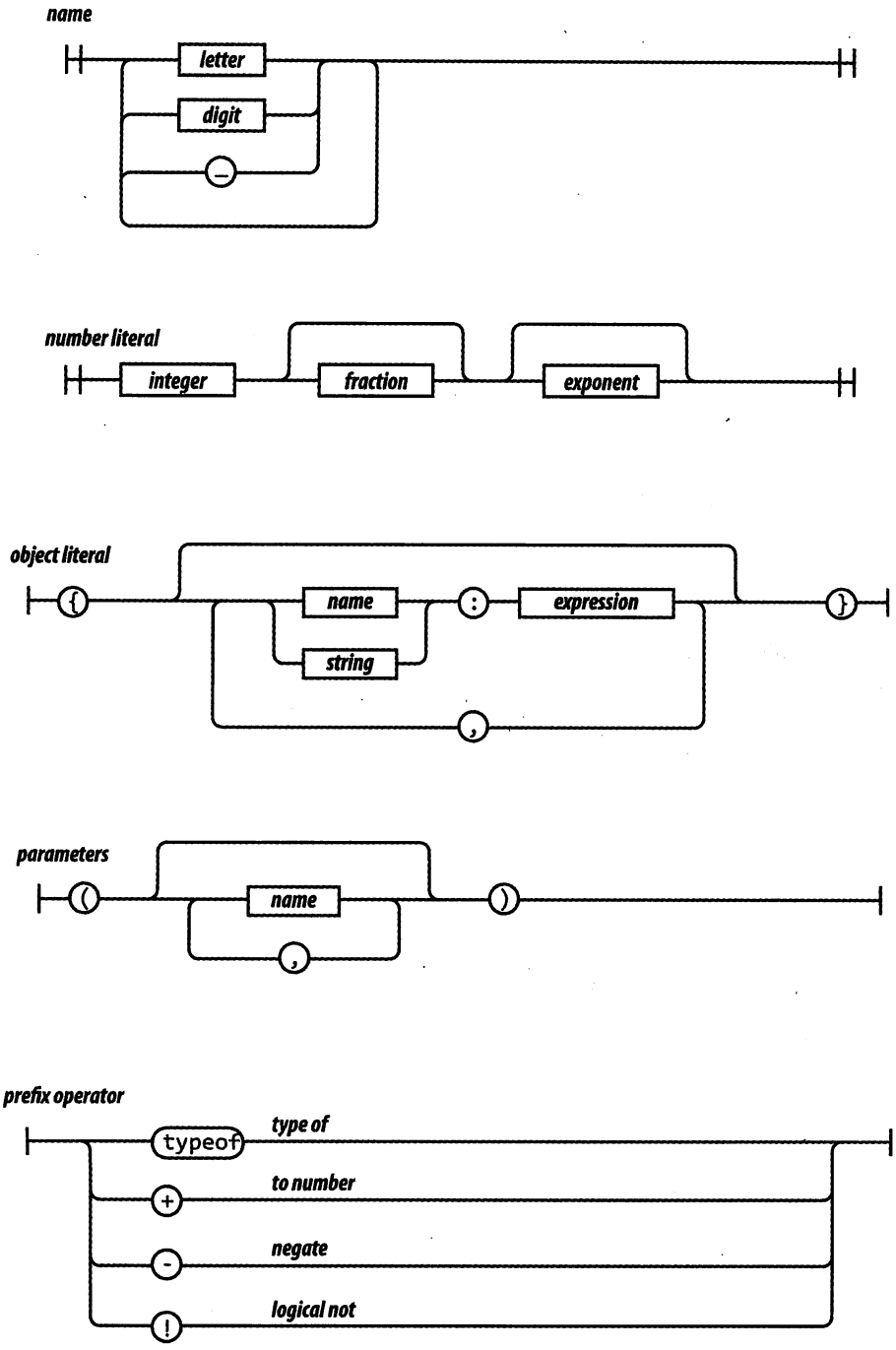

if statement
if
(
expression
)
then
block
else
block


infix operator
multiply
*
divide
/
modulo
%
add
+
subtract
-
greater or equal
>=
less or equal
<=
greater
>
less
<
equal
===
not equal
!==
logical or
||
logical and
&&

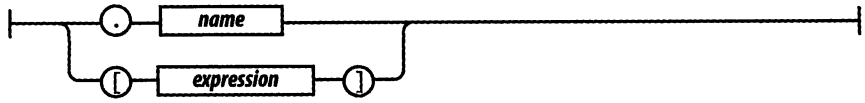

integer
0
any digit except 0
digit


invocation
(
expression
,
)

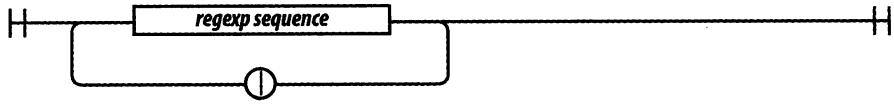

literal
number literal
string literal
object literal
array literal
function
regexp literal


name
letter
digit
_
number literal
integer
fraction
exponent
object literal
{
name
string
:
expression
}
,
parameters
(
name
)
,
prefix operator
typeof
type of
+
to number
-
negate
!
logical not

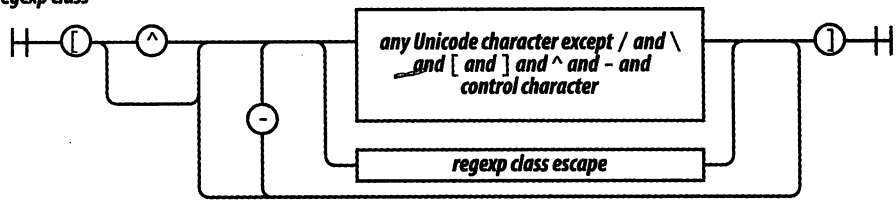
refinement



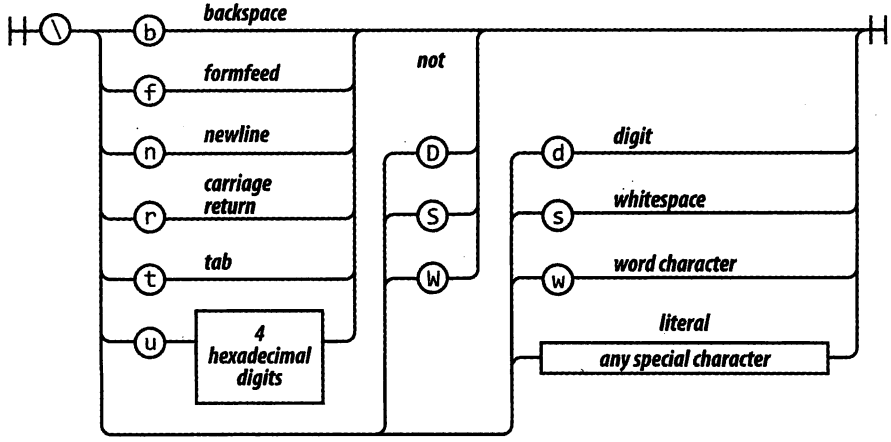
regexp choice



regexp class

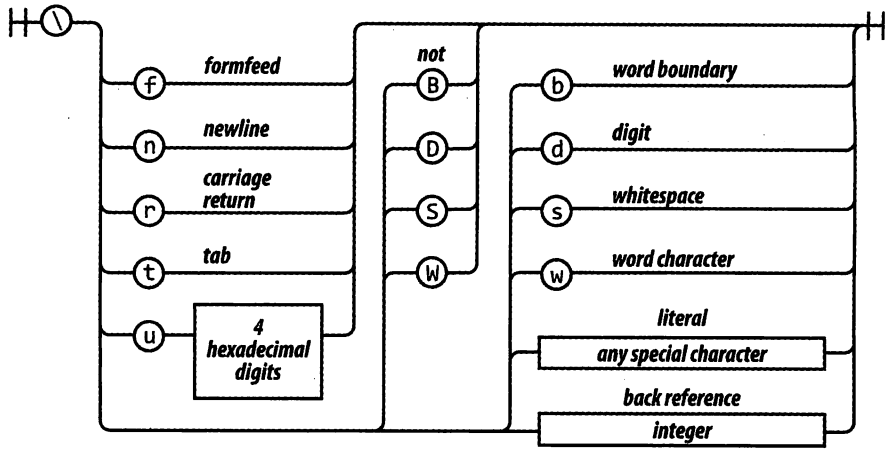


regexp class escape

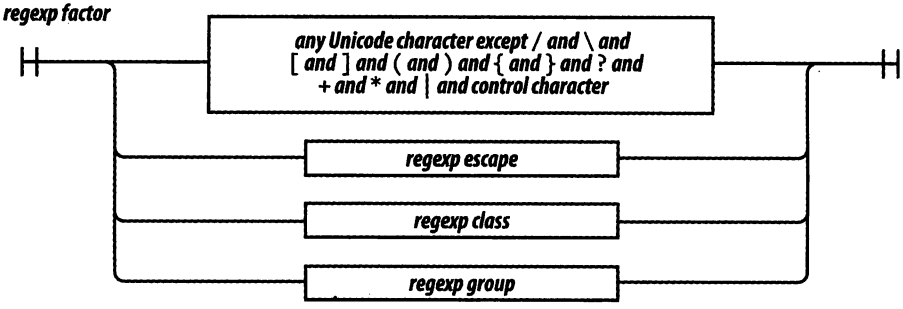


132

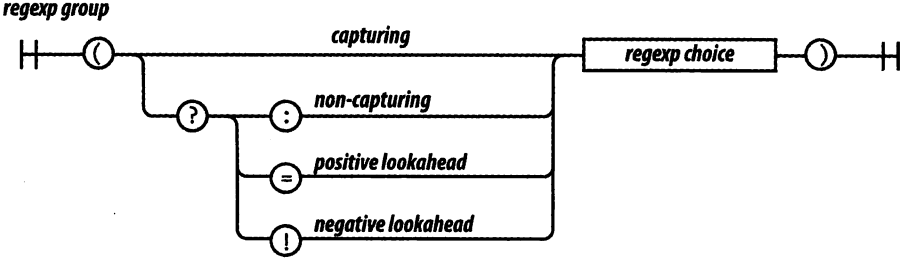
regexp escape



regexp factor

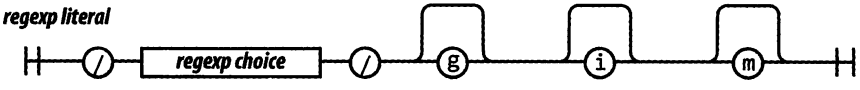


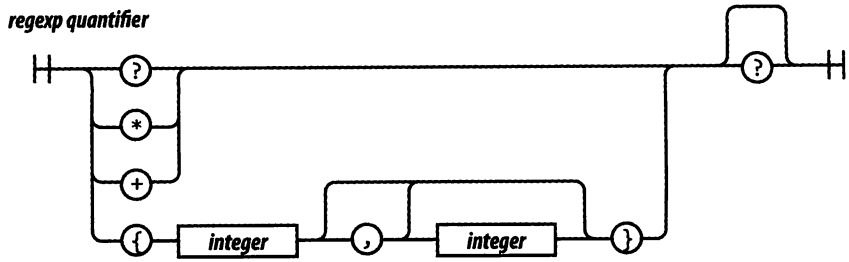
regexp group

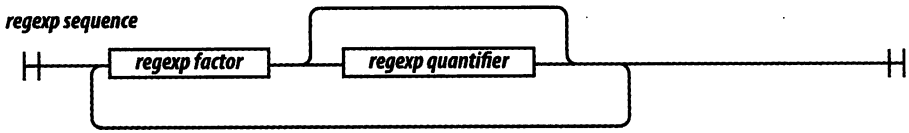


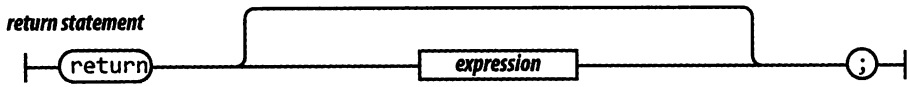
133

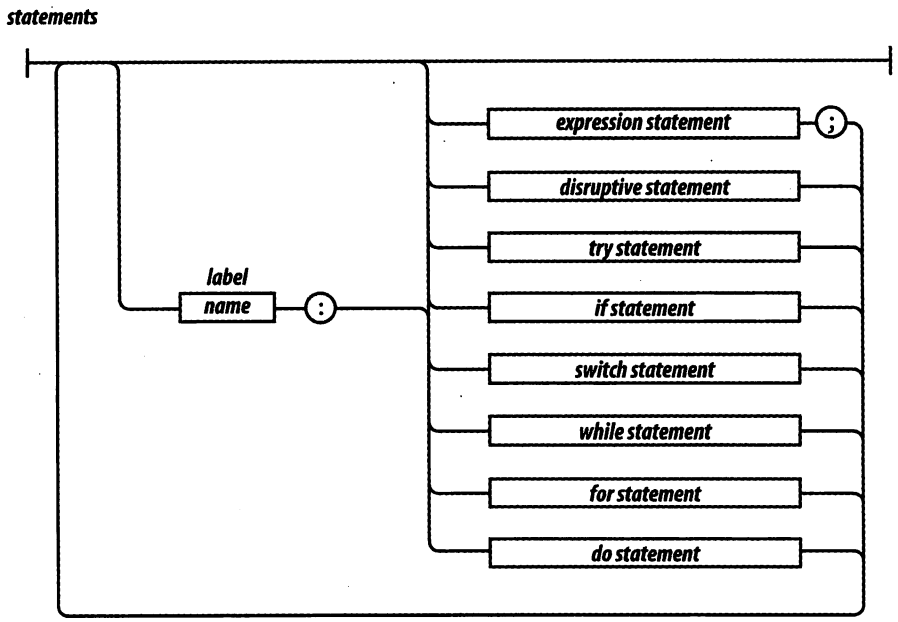
regexp literal



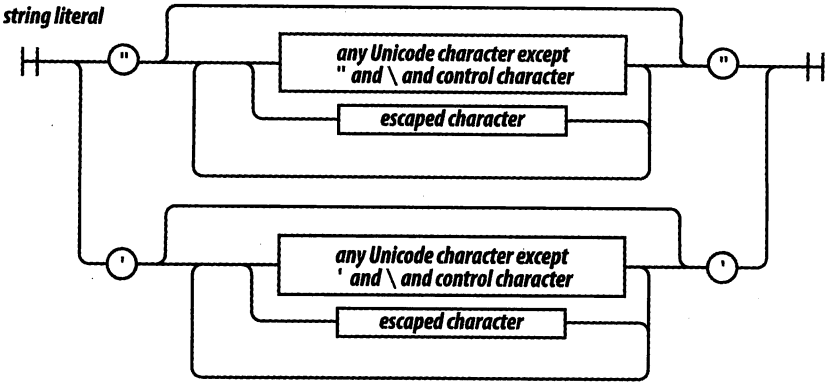

regexp quantifier
?
*
+
{
integer
,
integer
}
?


regexp sequence
regexp factor
regexp quantifier

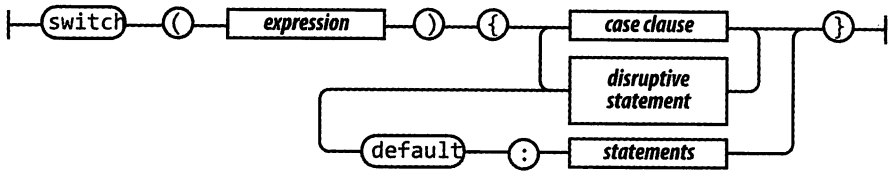

return statement
return
expression
;


statements
label
name
:
expression statement
;
disruptive statement
try statement
if statement
switch statement
while statement
for statement
do statement

string literal

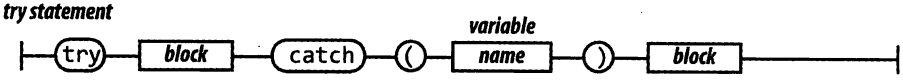


switch statement

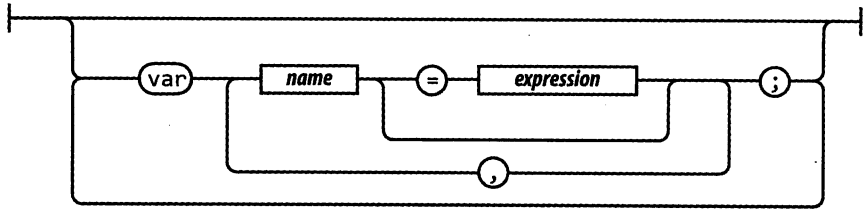


throw statement

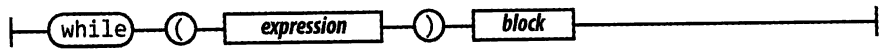
try statement



var statements

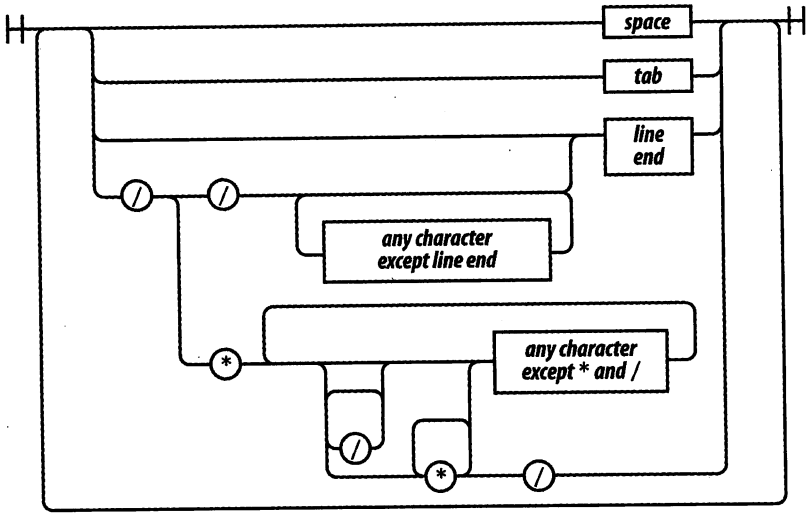


while statement



whitespace

135



附录 E
JSON

再会吧，这宝贵的片刻和短暂的时机限制了我在情义上的真挚表示，也不能容我们畅叙衷曲，这本来是亲友久违重逢所应有的机缘；愿上帝赐给我们美好的将来，好让我们开怀畅谈！再一次告别；勇敢作战吧，祝你胜利！
——威廉·莎士比亚，《理查三世》（*The Tragedy of Richard the Third*）

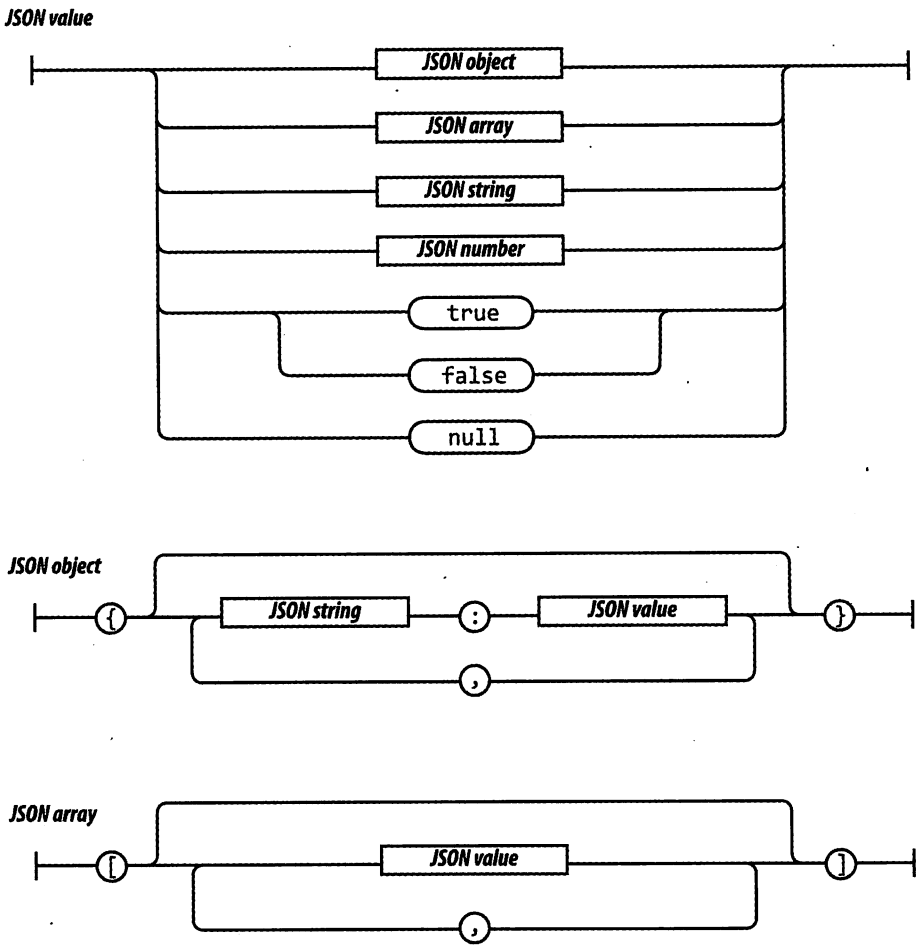
JavaScript 对象表示法（JavaScript Object Notation，简称 JSON）是一种轻量级的数据交换格式。它基于 JavaScript 的对象字面量表示法，那是 JavaScript 最精华的部分之一。尽管只是 JavaScript 的一个子集，但它与语言无关。所有以现代编程语言编写的程序，都可以用它来彼此交换数据。它是一种文本格式，所以可以被人和机器阅读。它易于实现且易于使用。大量关于 JSON 的资料都可以在 *http://www.JSON.org/*中找到。

JSON 语法
JSON Syntax

JSON 有 6 种类型的值：对象、数组、字符串、数字、布尔值（`true` 和 `false`）和特殊值 `null`。空白（空格符、制表符、回车符和换行符）可被插到任何值的前后。这使得 JSON 文本能更容易被人阅读。为了减少传输和存储的成本，空白可以省略。

JSON 对象是一个容纳“名/值”对的无序集合。名字可以是任何字符串。值可以是任何类型的 JSON 值，包括数组和对象。JSON 对象可以被无限层地嵌套，但一般来说保持其结构的相对扁平是最高效的。大多数语言都有容易映射为 JSON 对象的数据类型，比如对象（object）、结构（struct）、字典（dictionary）、哈希表（hash table）、属性列表（property list）或关联数组（associative array）。

JSON 数组是一个值的有序序列。其值可以是任何类型的 JSON 值，包括数组和对象。大多数语言都有容易被映射为 JSON 数组的数据类型，比如数组（array）、向量（vector）、列表（list）或序列（sequence）。



JSON 字符串被包围在一对双引号之间。\字符被用于转义。JSON 允许 /字符被转义，所以 JSON 可以嵌入 HTML 的<script>标签之中。除非是</script>标签，否则 HTML 不允许使用 </ 字符序列。但 JSON 允许使用<\/，它能产生同样的结果却不会与 HTML 相混淆[译注1]。

JSON 数字与 JavaScript 的数字相似。整数的首位不允许为 0，因为一些语言用它来标示八进制数。这种基数的混乱在数据交换格式中是不可取的。数字可以是整数、实数或科学计数。

就是这样。这就是 JSON 的全部。JSON 的设计目标是成为一个极简的、轻便的和文本式的 JavaScript 子集。实现互通所需要的共识越少，互通就越容易实现。

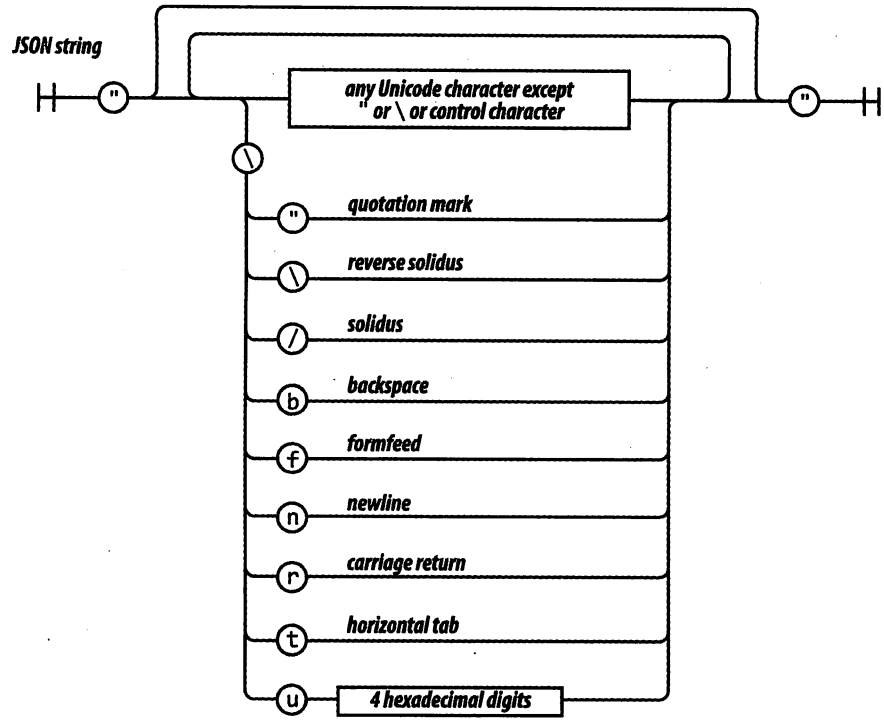
译注 1：作者此处所指的是类似这样的问题：

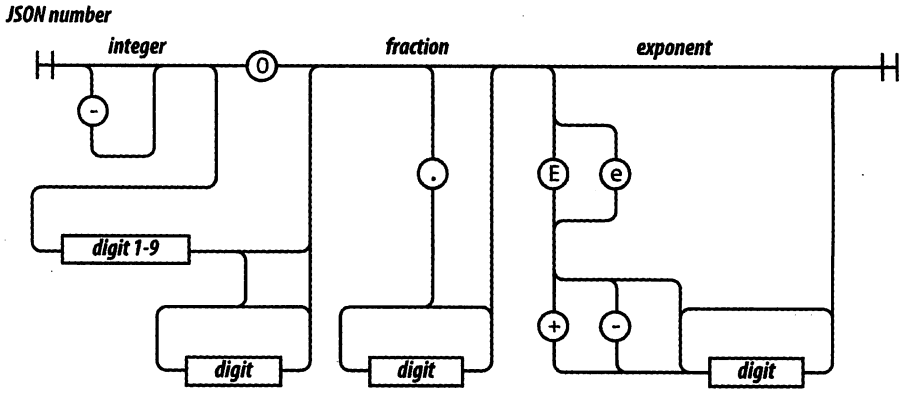
```
<script type='text/javascript'>JSON={"foo": "</script>"};</script>
```

如上代码在浏览器中执行时会导致脚本错误，加上\字符进行转义即可：

```
<script type='text/javascript'>JSON={"foo": "<\/script>"};</script>
```

138






```
[
  {
    "first": "Jerome",
    "middle": "Lester",
    "last": "Howard",
    "nick-name": "Curly",
    "born": 1903,
    "died": 1952,
    "quote": "nyuk-nyuk-nyuk!"
  },
  {
```

139

```
        "first": "Harry",
        "middle": "Moses",
        "last": "Howard",
        "nick-name": "Moe",
        "born": 1897,
        "died": 1975,
        "quote": "Why, you!"
    },
    {
        "first": "Louis",
        "last": "Feinberg",
        "nick-name": "Larry",
        "born": 1902,
        "died": 1975,
        "quote": "I'm sorry. Moe, it was an accident!"
    }
]
```

安全地使用 JSON

Using JSON Securely

JSON 特别易于用在 Web 应用中，因为 JSON 就是 JavaScript。使用 `eval` 函数可以把一段 JSON 文本转化成一个有用的数据结构：

```
var myData = eval('(' + myJSONText + ')');
```

（用圆括号把 JSON 文本括起来是一种避免 JavaScript 语法歧义[译注2]的变通方案。）

然而，`eval` 函数有着骇人的安全问题。用 `eval` 去解析 JSON 文本安全吗？目前，在 Web 浏览器中从服务器端获取数据的最佳技术是 `XMLHttpRequest`。`XMLHttpRequest` 只能从生成 HTML 的同源服务器获取数据。使用 `eval` 解析来自那个服务器的文本与解析原来的 HTML 一样不安全。那是假定该服务器存有恶意的前提下，但如果它只是存在漏洞呢？

有漏洞的服务器或许并不能正确地对 JSON 进行编码。如果它通过拼凑一些字符串而不是使用一个合适的 JSON 编码器来创建 JSON 文本，那么它可能在无意间发送了危险的数据。如果它充当的是代理的角色，并且尚未确定 JSON 文本是否格式良好就简单地传递它，那么它可能再次发送危险数据。

译注 2：在 JavaScript 的语法中，表达式语句（Expression Statement）不允许以左花括号“{”开始，因为那会与块语句（Block Statement）产生混淆。详细说明请参见 ECMAScript 规范的相关章节——“12.4 Expression Statement”。在使用 `eval()` 解析 JSON 文本时，为了解决此问题，可以将 JSON 文本套上一对圆括号。圆括号在此处作为表达式的分组运算符，能对包围在其中的表达式进行求值。它能正确地识别对象字面量。详细说明请参见 ECMAScript 规范的相关章节——“11.1.6 The Grouping Operator”。

通过使用 JSON.parse[译注3]方法替代 eval（见 *http://www.JSON.org/json2.js*）就能避免这种危险。如果文本中包含任何危险数据，那么 JSON.parse 将抛出一个异常。为了防止服务器出现漏洞的状况，我推荐你总是用 JSON.parse 来替代 eval。即使有一天浏览器提供了连到其他服务器的安全数据访问，使用它同样是个好习惯。

140 在外部数据与 innerHTML 进行交互时还存在另一种危险。一种常见的 Ajax 模式是把服务器端发送过来的一个 HTML 文本片段赋值给某个 HTML 元素的 innerHTML 属性。这是一个非常糟糕的习惯。如果这个 HTML 包含一个 <script>标签或其等价物，那么一个恶意脚本将被执行。这可能也是因为服务器端存在漏洞。

具体有什么危险呢？如果一个恶意脚本在你的页面上被运行，它就有权访问这个页面的所有状态和执行该页面能做的所有操作。它能与你的服务器进行交互，而你的服务器将不能区分正当请求和恶意请求。恶意脚本还能访问全局对象，这使得它有权访问应用中除隐藏于闭包中的变量之外的所有数据。它可以访问 document 对象，这会使它有权访问用户所能看到的一切。它还给这个恶意脚本提供了与用户进行会话的能力。浏览器的地址栏和所有的反钓鱼程序会告诉用户这个会话是可靠的。document 对象还给该恶意脚本授权访问网络，允许它去下载更多的恶意脚本，或者是在你的防火墙之内探测站点，或者是把它已经窃取的隐私内容发送给世界的任何一个服务器。

这个危险是 JavaScript 全局变量的直接后果，它是 JavaScript 众多糟糕的特性之中最糟糕的一个。这些危险并不是由 Ajax、JSON、XMLHttpRequest 或 Web 2.0（不管它是什么）导致的。自从 JavaScript 被引入浏览器，这些危险就已经存在了，并且它将一直存在，直到有一天 JavaScript 被取代或修补。所以，请务必当心。

一个 JSON 解析器

A JSON Parser

这是一个用 JavaScript 编写 JSON 解析器的实现方案：

```
var json_parse = function () {

// 这是一个能把 JSON 文本解析成 JavaScript 数据结构的函数。
// 它是一个简单的递归降序解析器。

// 我们在另一个函数中定义此函数，以避免创建全局变量。

    var at,     // 当前字符的索引
```

译注3：在译者翻译此书时，原生 JSON 支持的需求已被提交为 ES3.1 的工作草案（参见 *http://wiki. ecmascript.org/doku.php?id=es3.1:es3.1_proposal_working_draft*），另外 IE8 已经提供了原生的 JSON 支持（参见 *http://blogs.msdn.com/ie/archive/2008/09/10/native-json-in-ie8.aspx*）。

```
        ch,     // 当前字符
        escapee = {
            '"':  '"',
            '\\': '\\',
            '/':  '/',
            b:    'b',
            f:    '\f',
            n:    '\n',
            r:    '\r'
            t:    '\t'
        },
        text,

        error = function (m) {

// 当某处出错时，调用 error。

            throw {
                name:    'SyntaxError',
                message: m,
                at:      at,
                text:    text
            };
        },

        next = function (c) {

// 如果提供了参数 c ，那么检验它是否匹配当前字符。

            if (c && c !== ch) {
                error("Expected '" + c + "' instead of '" + ch + "'");
            }

// 获取下一个字符。当没有下一个字符时，返回一个空字符串。

            ch = text.charAt(at);
            at += 1;
            return ch;
        },

        number = function () {

// 解析一个数字值。

            var number,
                string = '';

            if (ch === '-') {
                string = '-';
                next('-');
            }
            while (ch >= '0' && ch <= '9') {
                string += ch;
                next();
            }
            if (ch === '.') {
```

```
            string += '.';
            while (next() && ch >= '0' && ch <= '9') {
                string += ch;
            }
        }
        if (ch === 'e' || ch === 'E') {
            string += ch;
            next();
            if (ch === '-' || ch === '+') {
                string += ch;
                next();
            }
            while (ch >= '0' && ch <= '9') {
                string += ch;
                next();
            }
        }
        number = +string;
        if (isNaN(number)) {
            error("Bad number");
        } else {
            return number;
        }
    },

    string = function () {

// 解析一个字符串值。

        var hex,
            i,
            string = '',
            uffff;

// 当解析字符串值时，我们必须找到 " 和 \ 字符。

        if (ch === '"') {
            while (next()) {
                if (ch === '"') {
                    next();
                    return string;
                } else if (ch === '\\') {
                    next();
                    if (ch === 'u') {
                        uffff = 0;
                        for (i = 0; i < 4; i += 1) {
                            hex = parseInt(next(), 16);
                            if (!isFinite(hex)) {
                                break;
                            }
                            uffff = uffff * 16 + hex;
                        }
                        string += String.fromCharCode(uffff);
                    } else if (typeof escapee[ch] === 'string') {
                        string += escapee[ch];
                    } else {
```

142

```
                    break;
                }
            } else {
                string += ch;
            }
        }
    }
    error("Bad string");
},

white = function () {

// 跳过空白。

    while (ch && ch <= ' ') {
        next();
    }
},

word = function () {
// true、false 或 null。

    switch (ch) {
        case 't':
            next('t');
            next('r');
            next('u');
            next('e');
            return true;
        case 'f':
            next('f');
            next('a');
            next('l');
            next('s');
            next('e');
            return false;
        case 'n':
            next('n');
            next('u');
            next('l');
            next('l');
            return null;
    }
    error("Unexpected '" + ch + "'");
},

value,    // 值函数的占位符。

array = function () {

// 解析一个数组值。

    var array = [];

    if (ch === '[') {
        next('[');
```

```
                white();
                if (ch === ']') {
                    next(']');
                    return array;   // 空数组
                }
                while (ch) {
                    array.push(value());
                    white();
                    if (ch === ']') {
                        next(']');
                        return array;
                    }
                    next(',');
                    white();
                }
            }
            error("Bad array");
        },

        object = function () {

// 解析一个对象值。

            var key,
                object = {};

            if (ch === '{') {
                next('{');
                white();
                if (ch === '}') {
                    next('}');
                    return object;   // 空对象
                }
                while (ch) {
                    key = string();
                    white();
                    next(':');
                    object[key] = value();
                    white();
                    if (ch === '}') {
                        next('}');
                        return object;
                    }
                    next(',');
                    white();
                }
            }
            error("Bad object");
        };

    value = function () {


// 解析一个 JSON 值。它可以是对象、数组、字符串、数字或一个词。

        white();
```

```
        switch (ch) {
        case '{':
            return object();
        case '[':
            return array();
        case '"':
            return string();
        case '-':
            return number();
        default:
            return ch >= '0' && ch <= '9' ? number() : word();
        }
    };

// 返回 json_parse 函数。它能访问上述所有的函数和变量。

    return function (source, reviver) {
        var result;

        text = source;
        at = 0;
        ch = ' ';
        result = value();
        white();
        if (ch) {
            error("Syntax error");
        }

// 如果存在 reviver 函数，我们就递归地对这个新结构调用 walk 函数，
// 开始时先创建一个临时的启动对象，并以一个空字符串作为键名保存结果，
// 然后传递每个“名/值”对给 reviver 函数去处理可能存在的转换。
// 如果没有 reviver 函数，我们就简单地返回这个结果。

        return typeof reviver === 'function' ?
            function walk(holder, key) {
                var k, v, value = holder[key];
                if (value && typeof value === 'object') {
                    for (k in value) {
                        if (Object.hasOwnProperty.call(value, k)) {
                            v = walk(value, k);
                            if (v !== undefined) {
                                value[k] = v;
                            } else {
                                delete value[k];
                            }
                        }
                    }
                }
                return reviver.call(holder, key, value);
            }({'': result}, '') : result;

    };
}();
```

索引†

Index

Symbols

-- decrement operator, 112, 118, 122
- negation operator, 122
-- operator, confusing pluses and minuses, 122
- subtraction operator, 122
!= operator, 109, 121
!== operator, 109
& and, 112
&& operator, 16
+ operator, 16, 104
  confusing pluses and minuses, 122
++ increment operator, 112, 118, 122
  confusing pluses and minuses, 122
+= operator, 15
<< left shift, 112
= operator, 15, 121
== operator, 106, 109, 121
=== operator, 15, 106, 109
>> signed right shift, 112
>>> unsigned right shift, 112
? ternary operator, 15
[ ] postfix subscript operator, 59
\ escape character, 8
^ xor, 112
| or, 112
|| operator, 17, 21
~ not, 112
/ operator, 16
/* */ form of block comments, 6
// comments, 6

A

adsafe option (JSLint), 117
Apply Invocation Pattern, 30
arguments, 31
arguments array, 106
array literals, 18
array.concat( ) method, 78
array.join( ) method, 78
array.pop( ) method, 79
Array.prototype, 62
array.push( ) method, 79
array.reverse( ) method, 79
array.shift( ) method, 79
array.slice( ) method, 80
array.sort( ) method, 80–82
array.splice( ) method, 82–83
array.unshift( ) method, 83
arrays, 58–64, 105
  appending new elements, 60
  arrays of arrays, 63
  cells of an empty matrix, 64
  confusion, 61
  delete operator, 60
  dimensions, 63
  elements of, 59
  enumeration, 60
  length property, 59
  literals, 58
  methods, 62
  Object.beget method, 63
  splice method, 60
  typeof operator, 61
  undefined value, 63
assignment, 121

† 中文版书中切口处的“▭”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引所列页码为原英文版页码。

assignment statement, 121
augmenting types, 32

B

beautiful features, 98–100
bitwise operators, 112, 122
bitwise option (JSLint), 117
block comments, 6, 96
block scope, 36, 99
blockless statements, 111
blocks, 10, 119
booleans, 20
braces, 96
break statement, 12, 14, 122
browser option (JSLint), 117
built-in value, 15

C

callbacks, 40
cap option (JSLint), 117
cascades, 42
case clause, 12
casting, 46
catch clause, 13
character type, 8
closure, 37–39
code quality tool (see JSLint)
comma operator, 119
comments, 6, 96
concatenation, 104
constructor functions, 123
  hazards, 49
  new prefix, forgetting to include, 49
Constructor Invocation Pattern, 29
constructor property, 47
constructors, 30
  defining, 47
continue statement, 111, 122
curly braces, 10
curry method, 43

D

debug option (JSLint), 117
deentityify method, 40
delegation, 23
delete operator, 24, 60
differential inheritance, 51
do statement, 10, 13
Document Object Model (DOM), 34
durable object, 55

E

ECMAScript Language Specification, 115
empty string, 12
enumeration, 24
eqeqeq option (JSLint), 117
equality operators, 109
escape character, 8
escape sequences, 9
eval function, 110, 122
  security problems, 139
evil option (JSLint), 117
exceptions, 32
executable statements, 10
expression statements, 120
expressions, 15–17
  ? ternary operator, 15
  built-in value, 15
  infix operator, 15
  invocation, 15
  literal value, 15
  operator precedence, 16
  preceded by prefix operator, 15
  refinement, 15
  refinement expression preceded by delete, 15
  variables, 15
  wrapped in parentheses, 15

F

factorial, 35, 45
Fibonacci numbers, 44, 45
floating-point numbers, 104
for in statement, 13, 120
  objects, 24
for statements, 10, 12, 119
forin option (JSLint), 117
fragment option (JSLint), 117
Function constructor, 47, 111
function invocation, 95
Function Invocation Pattern, 28
function object, when object is created, 47
function statement versus function expression, 113
function.apply( ) method, 84
functional pattern (inheritance), 52–55
functions, 19, 26–45, 116
  arguments, 31
  augmenting types, 32
  callbacks, 40

cascades, 42
closure, 37–39
curry method, 43
exceptions, 32
general pattern of a module, 41
invocation, 27–30
  Apply Invocation Pattern, 30
  Constructor Invocation Pattern, 29
  Function Invocation Pattern, 28
  Method Invocation Pattern, 28
  new prefix, 29
invocation operator, 28
invoked with constructor invocation, 47
literals, 27
memoization, 44
modules, 40–42
objects, 26
recursive, 34–36
  Document Object Model (DOM), 34
  Fibonacci numbers, 44
  tail recursion optimization, 35
  Towers of Hanoi puzzle, 34
return statement, 31
scope, 36
that produce objects, 52
var statements, 10

G

global declarations, 116
global object, 140
global variables, 25, 97, 101, 116
glovar option (JSLint), 117
good style, 95
grammar, 5–19
  expressions (see expressions)
  functions, 19
  literals, 17
  names, 6
  numbers, 7
    methods, 8
    negative, 8
  object literals, 17
  rules for interpreting diagrams, 5
  statements (see statements)
  strings, 8
    immutability, 9
    length property, 9
  whitespace, 5

H

hasOwnProperty method, 23, 107, 108
HTML
  <script> tags (JSON), 137
  innerHTML property, 140
  JSLint, 124

I

if statements, 10, 119
implied global, 102
Infinity, 7, 8, 15
inheritance, 3, 46–57
  differential, 51
  functional pattern, 52–55
  object specifiers, 50
  parts, 55–57
  prototypal pattern (inheritance), 50
  pseudoclassical pattern, 47–49, 54
inherits method, 49
innerHTML property, 140
instances, creating, 48
invocation operator, 17, 28
invocations, 95

J

JavaScript
  analyzing, 3–4
  standard, 4
  why use, 2
JavaScript Object Notation (see JSON)
JSLint, 4, 115–124
  -- decrement operator, 122
    confusing pluses and minuses, 122
  - operator, confusing pluses and minuses, 122
  != operator, 121
  + operator, confusing pluses and minuses, 122
  ++ increment operator, 122
    confusing pluses and minuses, 122
  = operator, 121
  == operator, 121
  assignment statement, 121
  bitwise operators, 122
  blocks, 119
  break statement, 122
  comma operator, 119
  constructor functions, 123

JSLint *(continued)*
continue statement, 122
eval function, 122
expression statements, 120
for in statement, 120
for statements, 119
function report, 124
functions, 116
global declarations, 116
global variables, 116
HTML, 124
if statements, 119
JSON, 124
labels, 122
line breaking, 118
members, 116
new prefix, 123
options, 117
regular expressions, 123
return statement, 122
semicolons, 118
switch statements, 120
throw statement, 122
var statements, 120
variables, 116
void, 123
where to find, 115
with statement, 121
JSON, 124
JSON (JavaScript Object Notation), 3, 136–140
/ character, 137
array, 136
eval function, 139
HTML <script> tags, 137
innerHTML property, 140
JSLint, 124
numbers, 137
object, 136
string, 137
syntax, 136–139
text example, 138
using securely, 139
JSON.parse method, 139

K

K&R style, 96
Kleene, Stephen, 65

L

labeled statement, 14
labels, 122
language, structure (see grammar)
laxbreak option (JSLint), 117
length property (arrays), 59
line breaking, 118
line comments, 96
line-ending comments, 6
looping statement, 12, 14
loosely typed language, 46

M

Math object, 8
memoization, 44
message property, 14
Method Invocation Pattern, 28
method method, 49
methods, 78–93
array.concat( ), 78
array.join( ), 78
array.pop( ), 79
array.push( ), 79
array.reverse( ), 79
array.shift( ), 79
array.slice( ), 80
array.sort( ), 80–82
array.splice( ), 82–83
array.unshift ( ), 83
arrays, 62
function.apply( ), 84
number.toExponential( ), 84
number.toFixed( ), 85
number.toPrecision( ), 85
number.toString( ), 85
object.hasOwnProperty( ), 86
regexp.exec( ), 65, 86
regexp.test( ), 65, 88
string.charAt( ), 88
string.charCodeAt( ), 88
string.concat( ), 88
String.fromCharCode( ), 93
string.indexOf( ), 88
string.lastIndexOf( ), 89
string.localeCompare( ), 89
string.match( ), 65, 89
string.replace( ), 65, 90
string.search( ), 65, 91
string.slice( ), 91
string.split( ), 65, 91

string.substring( ), 92
string.toLocaleLowerCase( ), 92
string.toLocaleUpperCase( ), 92
string.toLowerCase( ), 92
string.toUpperCase( ), 93
that work with regular expressions, 65
modules, 40–42
general pattern, 41
multiple statements, 95
my object, 53

N

name property, 14
names, 6
NaN (not a number), 7, 8, 12, 15, 105
negative numbers, 8
new operator, 15, 47, 114, 123
forgetting to include, 49
functions, 29
newline, 73
nomen option (JSLint), 117
null, 11, 15, 20, 106
number literal, 8
number.toExponential( ) method, 84
number.toFixed( ) method, 85
number.toPrecision( ) method, 85
number.toString( ) method, 85
numbers, 7, 20
methods, 8
negative, 8
numbers object, 59

O

object literals, 17, 59
object specifiers, 50
Object.beget method, 53, 63
object.hasOwnProperty( ) method, 86
Object.prototype, 62
objects, 20–25, 107
|| operator, 21
creating new, 22
defined, 20
delegation, 23
delete operator, 24
durable, 55
enumeration, 24
for in statement, 24
functions, 26
global variables, 25
hasOwnProperty method, 23
literals, 20
properties, 21
property on prototype chain, 23
prototype, 22
link, 23
reference, 22
reflection, 23
retrieving values, 21
undefined, 21, 23
updating values, 22
on option (JSLint), 117
operator precedence, 16

P

parseInt function, 104
passfail option (JSLint), 117
pi as simple constant, 99
plusplus option (JSLint), 117
Pratt, Vaughn, 98
private methods, 53
privileged methods, 53
problematic features of JavaScript, 101–114
+ operator, 104
arrays, 105
bitwise operators, 112
blockless statements, 111
continue statement, 111
equality operators, 109
eval function, 110
falsy values, 106
floating-point numbers, 104
function statement versus function expression, 113
global variables, 101
hasOwnProperty method, 107
increment and decrement operators, 112
NaN (not a number), 105
new operator, 114
objects, 107
parseInt function, 104
reserved words, 103
scope, 102
semicolons, 102
single statement form, 111
string argument form, 111
switch statement, 111
typed wrappers, 114
typeof operator, 103
Unicode, 103
void, 114
with statement, 110

prototypal inheritance, 3
prototypal inheritance language, 29
prototypal pattern, 50
prototype property, 47
prototypes of basic types, 33
pseudoclass, creating, 48
pseudoclassical pattern (inheritance), 47–49, 54
punctuation characters or operators, 118

R

railroad diagrams, 67
recursion, 34–36
  Document Object Model (DOM), 34
  Fibonacci numbers, 44, 45
  tail recursion optimization, 35
  Towers of Hanoi puzzle, 34
reflection, 23
RegExp objects, properties, 72
regexp.exec( ) method, 65, 86
regexp.test( ) method, 65, 88
regular expressions, 65–77, 123
  $ character, 69
  (...), 68
  (?! prefix, 75
  (?: prefix, 75
  (?:...)?, 70
  (?= prefix, 75
  ? character, 67
  \1 character, 74
  \b character, 73, 74
  \d, 70
  \D character, 73
  \d character, 73
  \f character, 73
  \n character, 73
  \r character, 73
  \S character, 73
  \s character, 73
  \t character, 73
  \u character, 73
  \W character, 73
  \w character, 73
  ^ character, 67, 69
  / character, 68
  backslash character, 73–74
  capturing group, 68, 74
  carriage return character, 73
  construction, 70–72
  elements, 72–77
    regexp choice, 72
    regexp class, 75
    regexp class escape, 76
    regexp escape, 73–74
    regexp factor, 73, 76
    regexp group, 74
    regexp quantifier, 76
    regexp sequence, 72
  flags, 71
  formfeed character, 73
  matching digits, 70
  matching URLs, 66–70
  methods that work with, 65
  negative lookahead group, 75
  newline character, 73
  noncapturing group, 67, 75
  optional group, 68
  optional noncapturing group, 70
  positive lookahead group, 75
  railroad diagrams, 67
  repeat zero or one time, 68
  simple letter class, 74
  sloppy, 68
  tab character, 73
  Unicode characters, 73
reserved words, 7, 103
return statement, 14, 31, 122
rhino option (JSLint), 117

S

says method, 53
scope, 10, 36, 102
semicolons, 102, 118
seqer object, 42
setInterval function, 111
setTimeout function, 111
Simplified JavaScript, 98
single statement form, 111
spec object, 52, 53
splice method (arrays), 60
statements, 10–15
  blocks, 10
  break, 12, 14
  case clause, 12
  catch clause, 13
  do, 10, 13
  executable, 10
  execution order, 10

for, 10, 12
for in, 13
if, 10
labeled, 14
loop, 14
looping, 12
return, 14
switch, 10, 12, 14
then block, 10
throw, 14
try, 13
var, functions, 10
while, 10, 12
string argument form, 111
string literal, 8
String, augmenting with deentityify method, 40
string.charAt( ) method, 88
string.charCodeAt( ) method, 88
string.concat( ) method, 88
String.fromCharCode( ) method, 93
string.indexOf( ) method, 88
string.lastIndexOf( ) method, 89
string.localeCompare( ) method, 89
string.match( ) method, 65, 89
string.replace( ) method, 65, 90
string.search( ) method, 65, 91
string.slice( ) method, 91
string.split( ) method, 65, 91
string.substring( ) method, 92
string.toLocaleLowerCase( ) method, 92
string.toLocaleUpperCase( ) method, 92
string.toLowerCase( ) method, 92
string.toUpperCase( ) method, 93
strings, 8, 20
empty, 12
immutability, 9
length property, 9
structure of language (see grammar)
structured statements, 95
style, 94–97
block comments, 96
braces, 96
comments, 96
global variables, 97
good, 95
invocations, 95
K&R, 96
line comments, 96
multiple statements, 95
structured statements, 95
switch cases, 97
super methods, 49
superior method, 54
switch statement, 10, 12, 14, 97, 111, 120
syntax checker (see JSLint)
syntax diagrams, 125–135

T

tail recursion optimization, 35
testing environment, 4
then block, 10
this keyword, 49
Thompson, Ken, 65
throw statement, 14, 122
Top Down Operator Precedence parser, 98
Towers of Hanoi puzzle, 34
trim method, 33
try statement, 13
typed wrappers, 114
TypeError exception, 21
typeof operator, 16, 61, 103, 106
types, 20
prototypes of, 33

U

undef option (JSLint), 117
undefined, 7, 12, 15, 20, 21, 23, 28, 31, 63, 106
Unicode, 103

V

var statements, 120
functions, 10
variables, 15, 116
verifier (see JSLint)
void operator, 114, 123

W

while statement, 10, 12
white option (JSLint), 117
whitespace, 5
widget option (JSLint), 117
Wilson, Greg, 98
with statement, 110, 121
wrappers, typed, 114

作者简介

Douglas Crockford 是一名来自 Yahoo!的资深 JavaScript 架构师，以创造和维护 JSON（JavaScript Object Notation）格式而为大家所熟知。他定期于各类会议上发表有关高级 JavaScript 的主题演讲，并且他也是 ECMAScript 委员会的成员之一。

封面介绍

本书封面动物：金斑蝶（桦斑蝶）。在亚洲以外的地方，这种昆虫也被称为非洲帝王蝶。这是一种中等个头的蝴蝶，其标志为醒目的亮橙色鳞翅及鳞翅上 6 个黑点和黑白交替的条纹。

其惊艳的外表吸引了众多科学家与艺术家的关注。作家弗拉基米尔·纳博科夫（同时为著名鳞翅目昆虫学家），也曾在以苛刻著称的纽约时报书评中为爱丽丝·福特的《奥杜邦的蝴蝶、蛾类及其他研究》（The Studio Publications）撰文，不吝溢美之词对其大加赞赏。在本书中，福特指出，19 世纪直到奥杜邦时期内，那些对于金斑蝶的描绘都是不科学的。

纳博科夫在给福特的回复中写道，“在 1797 年 John Abbot 关于北美鳞翅目的大量资料或 18 世纪至 19 世纪初的德国鳞翅类资料中很可能找到金斑蝶的踪影。它甚至在 3300 多年前的图特摩斯四世或阿梅诺菲斯三世时期出现过。在古埃及的壁画上，人们发现了描绘精美的蝴蝶——不可思议的是，并非以往常出现的金龟子形象——它巧妙地将金斑蝶的躯干与苎胥蝶（蝴蝶的一种）的图案结合在一起。”

但是，金斑蝶美丽的外表下却深藏杀机。在其幼虫阶段，它从植物中摄取对鸟类有毒的生物碱——鸟类是其主要的捕食者，往往被其鲜艳色彩引诱而来。鸟类捕食金斑蝶后会呕吐甚至死亡。幸存的鸟儿将向其他鸟儿传递信息避免误食金斑蝶并对其敬而远之。因此，金斑蝶得以悠闲地生活在地球上。

该封面图片取材于《英国多佛港的动物》。