SAMS Teach Yourself

- 全球销量逾百万册的系列图书
- 连续十余年打造的经典品牌
- 直观、循序渐进的学习教程
- 掌握关键知识的最佳起点
- “Read Less, Do More”（精读多练）的教学理念
- 以示例引导读者完成最常见的任务

每章内容针对初学者精心设计，1小时轻松阅读学习，
24小时彻底掌握关键知识

每章案例与练习题助你轻松完成常见任务，
通过实践提高应用技能，巩固所学知识

Node.js 入门经典

[英] George Ornbo 著
傅强 陈宗斌 译

人民邮电出版社
POSTS & TELECOM PRESS

Node.js 入门经典

[英] George Ornbo 著
傅强 陈宗斌 译

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Node.js入门经典 /（英）奥尔波（Ornbo,G.）著；傅强，陈宗斌译. -- 北京 : 人民邮电出版社，2013.4
ISBN 978-7-115-31107-8

Ⅰ. ①N… Ⅱ. ①奥… ②傅… ③陈… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第034588号

版 权 声 明

Node.js 入门经典

◆ 著 ［英］George Ornbo
译 傅 强 陈宗斌
责任编辑 傅道坤

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：787×1092 1/16
印张：22
字数：544 千字 2013 年 4 月第 1 版
印数：1 – 3 500 册 2013 年 4 月北京第 1 次印刷

著作权合同登记号 图字：01-2012-7073 号
ISBN 978-7-115-31107-8

定价：59.00 元

读者服务热线：(010)67132692 印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

内容提要

Node.js 是一套用来编写高性能网络服务器的 JavaScript 工具包，从 2009 年诞生之日起，就获得了业内专家和技术社区的强烈关注。而本书采用直观、循序渐进的方法对如何使用 Node.js 来开发及具速度和可扩展性优势的服务器端应用程序进行了讲解。

本书分为 6 部分，第 1 部分介绍了 Node.js 的基本概念和特性；第 2 部分讲解如何借助 HTTP 模块和 Express Web 框架，使用 Node.js 创建基本的网站；第 3 部分介绍了调试和测试 Node.js 应用程序的工具，以及部署 Node.js 应用的方法；第 4 部分讲解了 Node.js 实现实时编程的能力以及 Socket.IO；第 5 部分介绍了 Node.js API 以及构建 Node.js 应用程序所使用的组件；第 6 部分则介绍了 CoffeeScript 这款 Java 预编译器的知识，以及如何在 Node.js 中使用中间件、Backbone.js 来创建单页面应用的知识。

本书内容循序渐进、深入浅出、步骤详尽，而且附有大量适合动手实践的示例，可帮助读者在最短的时间内掌握 Node.js。本书适合对 Node.js 感兴趣的零基础人员阅读，也适合对 Web 前端开发、后端开发感兴趣的技术人员阅读。

关于作者

George Ornbo 是英国的一位 JavaScript 和 Ruby 开发人员。他开发 Web 应用程序已有 8 年时间，一开始是以自由职业者的身份工作，最近则为伦敦的 pebble {code}工作。他的博客地址是 http://shapeshed.com，在网络中大多数常见的地方他都以@shapeshed 出现。

献辞

本书献给我的妻子 Kirsten。没有你的支持，本书就不可能问世。

致谢

感谢 Trina MacDonald 和 Pearson 团队给了我编写本书的机会。你们的鼓励和引导是无价之宝。

感谢本书的技术编辑 Remy Sharp。你找出了大量错误并监督了评审过程。我欠你一顿酒！本书中若还有错误，那肯定是我自己的过失。

感谢我在 pebble {code}的同事。在本书写作伊始，你们就是我的后盾。你们允许我在大型项目中灵活安排时间，让我得以完成本书，我深表感激。

前　言

Node.js 可以让开发人员在服务器上使用 JavaScript，这让熟悉 JavaScript 的开发人员又多了一种服务器端的开发技能，但 Node.js 并非仅限于此。它重新思考了在现代 Web 环境下的网络编程，在这个环境下，应用程序可能需要从许多不同的地方读写数据，也可能有上百万个并发用户。

在具有传统的计算机科学学位的开发人员眼中，JavaScript 就是一种玩具语言。但是，JavaScript 已经经历了无数次的挑战，而且如今在 Web 的浏览器和服务器端（借助于 Node.js）中已经不可或缺。现在是编写 JavaScript（尤其是在服务器上）的最好时节！

Node.js 表示一个开发平台，在创建适用于现在 Web 的应用程序时，Node.js 大有裨益，这些应用程序包括：

- 实时应用程序；
- 多人游戏；
- 单页面应用程序；
- 基于 JSON 的 API。

Node.js 专注于速度和可扩展性，而且在无需昂贵硬件的情况下，能处理上千个并发用户的需求。Node.js 项目最近成为 GitHub 上最受关注的项目，如今，eBay、LinkedIn 和 Microsoft 这样的公司已经开始使用它。

Node.js 绝不仅仅只是服务器上的 JavaScript。它是一个功能齐全的网络编程平台，能够针对现代 Web 编程的需求做出响应。

本书读者对象

本书并不要求读者一定具备编程经验，但是如果有基本的 JavaScript 使用经验会更好。由于 Node.js 主要从终端运行，因此知道什么是终端，以及如何运行基本的命令也会更有帮助。最后，由于 Node.js 主要是一个网络编程工具，因此，如果读者对 Internet 的运行机制略知一二，会更好。

学习 Node.js 的理由

如果读者对创建有许多用户、处理联网数据或者有实时要求的应用程序感兴趣，那么 Node.js 是完成这些任务的极佳工具。此外，如果为浏览器创建应用程序，Node.js 可以让服务器是 JavaScript 的，这可以简化服务器和客户端之间的数据共享。Node.js 是现代 Web 的现代工具箱。

本书组织结构

本书首先讲解了 Node.js 的基础知识，包括运行你的第一个 Node.js 程序以及使用 npm（Node 包管理器），然后介绍了网络编程，以及 Node.js 使用 JavaScript 回调来支持异步编程风格的方法。

在本书第 2 部分，我们将学习如何通过使用 HTTP 模块和 Express（一个 Node.js 的 Web 框架），并借助 Node.js 创建基本的网站。我们还将学习如何使用 MongoDB 来让数据持久化。

第 3 部分介绍用于调试和测试 Node.js 应用程序的工具，其中介绍了许多用来支持开发的调试工具和测试框架。我们还将学习如何将 Node.js 应用程序部署到许多第三方服务上，包括 Heroku 和 Nodester。

第 4 部分讲解 Node.js 的实时能力并介绍 Socket.IO。我们将学习如何在浏览器和服务器之间发送消息，并构建一个完整的聊天服务器示例和一个实时的 Twitter 客户端。最后我们将学习如何使用 Node.js 创建 JSON API。

第 5 部分以 Node.js API 为主，并讲解用于创建 Node.js 应用程序的构件（buiding block）。我们将学习进程、子进程、事件、缓冲区和流。

第 6 部分介绍的是读者可能想了解的一些高级主题。我们将学习 CoffeeScript 这个 JavaScript 预编译器，Node.js 如何使用中间件，以及如何使用 Backbone.js 与 Node.js 一起创建单页面应用程序。第 22 章将介绍如何使用 npm 编写并发布你自己的 Node.js 模块。

代码示例

本书每章都带有几个代码示例。这些示例旨在帮助读者更好地理解 Node.js。读者可从 http://bit.ly/nodejsbook-examples 下载这些代码，也可从 https://github.com/shapeshed/nodejsbook.io.examples 的 GitHub 库下载。

目　录

第 1 部分　入门

第 1 章　Node.js 介绍……2

1.1　什么是 Node.js ……2
1.2　使用 Node.js 能做什么……3
1.3　安装并创建第一个 Node.js 程序……3
1.3.1　验证 Node.js 正确安装……4
1.3.2　创建“Hello World” Node.js 程序 ……4
1.4　小结……5
1.5　问与答……6
1.6　测验……6
1.6.1　问题……6
1.6.2　答案……7
1.7　练习……7

第 2 章　npm（Node 包管理器）……8

2.1　npm 是什么……8
2.2　安装 npm……9
2.3　安装模块……9
2.4　使用模块……10
2.5　如何找模块……11
2.5.1　官方来源……11
2.5.2　非官方来源……12
2.6　本地和全局的安装……13
2.6.1　本地安装……13
2.6.2　全局安装……13
2.7　如何找模块文档……14
2.8　使用 package.json 指定依赖关系（dependency）……14
2.9　小结……16
2.10　问与答……16
2.11　测验……16
2.11.1　问题……16
2.11.2　答案……17
2.12　练习……17

第 3 章　Node.js 的作用……18

3.1　设计 Node.js 的目的……18
3.2　理解 I/O……19
3.3　处理输入……19
3.4　联网的 I/O 是不可预测的……22
3.5　人类是不可预测的……24
3.6　处理不可预测性……25
3.7　小结……26
3.8　问与答……26
3.9　测验……27
3.9.1　问题……27
3.9.2　答案……27
3.10　练习……27

第 4 章　回调（Callback）……29

4.1　什么是回调……29
4.2　剖析回调……33
4.3　Node.js 如何使用回调……34
4.4　同步和异步代码……36
4.5　事件循环……39
4.6　小结……39
4.7　问与答……39
4.8　测验……40
4.8.1　问题……40
4.8.2　答案……40
4.9　练习……40

第 2 部分　使用 Node.js 的基本网站

第 5 章　HTTP……44

5.1　什么是 HTTP……44
5.2　使用 Node.js 的 HTTP 服务器……44
5.2.1　一个基础的服务器……44
5.2.2　加入头（Header）……45
5.2.3　检查响应头……46
5.2.4　Node.js 中的重定向……49
5.2.5　响应不同的请求……50
5.3　使用 Node.js 的 HTTP 客户端……52
5.4　小结……53
5.5　问与答……53
5.6　测验……54
5.6.1　问题……54
5.6.2　答案……54
5.7　练习……54

第 6 章　Express 介绍……55

6.1　什么是 Express……55
6.2　为什么使用 Express……55
6.3　安装 Express……56
6.4　创建一个基础的 Express 站点……56
6.5　探索 Express……58
6.5.1　app.js……58
6.5.2　node_modules……58
6.5.3　package.json……58
6.5.4　public……58
6.5.5　routes……58
6.5.6　views……58
6.6　介绍 Jade……59
6.6.1　使用 Jade 定义页面结构……60
6.6.2　使用 Jade 输出数据……62
6.7　小结……68
6.8　问与答……68
6.9　测验……68
6.9.1　问题……69
6.9.2　答案……69
6.10　练习……69

第 7 章　深入 Express……70

7.1　Web 应用程序中的路由……70
7.2　在 Express 中路由如何工作……70
7.3　添加 GET 路由……71
7.4　添加 POST 路由……72
7.5　在路由中使用参数……73
7.6　让路由保持可维护性……74
7.7　视图渲染……75
7.8　使用本地变量……76
7.9　小结……78
7.10　问与答……78
7.11　测验……78
7.11.1　问题……79
7.11.2　答案……79
7.12　练习……79

第 8 章　数据的持久化……80

8.1　什么是持久的数据……80
8.2　将数据写入文件……81
8.3　从文件读取数据……82
8.4　读取环境变量……83
8.5　使用数据库……84
8.5.1　关系数据库……84
8.5.2　NoSQL 数据库……85
8.6　在 Node.js 中使用 MongoDB……85
8.6.1　安装 MongoDB……86
8.6.2　连接 MongoDB……87
8.6.3　定义文档……89
8.6.4　将 Twitter Bootstrap 包含进来……90

8.6.5 索引（Index）视图……91
8.6.6 创建（Create）视图 ……93
8.6.7 编辑视图……95
8.6.8 删除任务……98
8.6.9 添加闪出消息……99
8.6.10 验证输入的数据……101
8.7 小结……102
8.8 问与答……103
8.9 测验……103
8.9.1 问题……103
8.9.2 答案……103
8.10 练习……104

第 3 部分 调试、测试与部署

第 9 章 调试 Node.js 应用程序…… 106

9.1 调试……106
9.2 STDIO 模块……107
9.3 Node.js 调试器 ……111
9.4 Node Inspector ……113
9.5 关于测试的注释……116
9.6 小结……116
9.7 问与答……116
9.8 测验……117
9.8.1 问题……117
9.8.2 答案……117
9.9 练习……117

第 10 章 测试 Node.js 应用程序…… 119

10.1 为什么测试……119
10.2 Assert（断言）模块……120
10.3 第三方测试工具……122
10.4 行为驱动的开发（Behavior Driven Development）……125
10.4.1 Vows……125
10.4.2 Mocha……128
10.5 小结……131
10.6 问与答……131
10.7 测验……132
10.7.1 问题……132
10.7.2 答案……132
10.8 练习……132

第 11 章 部署 Node.js 应用程序…… 133

11.1 准备好部署……133
11.2 在云上托管……133
11.3 Heroku……135
11.3.1 注册 Heroku……135
11.3.2 为 Heroku 准备应用程序……136
11.3.3 将应用程序部署到 Heroku……137
11.4 Cloud Foundry ……138
11.4.1 注册 Cloud Foundry……138
11.4.2 为 Cloud Foundry 准备应用程序……139
11.4.3 将应用程序部署到 Cloud Foundry ……140
11.5 Nodester ……141
11.5.1 注册 Nodester ……141
11.5.2 为 Nodester 准备应用程序……142
11.5.3 将应用程序部署到 Nodester ……143
11.6 其他 PaaS 提供商……144
11.7 小结……144
11.8 问与答……144
11.9 测验……145
11.9.1 测验……145
11.9.2 答案……145
11.10 练习……145

第 4 部分 使用 Node.js 的中间站点

第 12 章 介绍 Socket.IO…… 148

12.1 现在要开始学习一些完全不同的技术了……148
12.2 动态 Web 简史 ……148
12.3 Socket.IO ……149
12.4 基础的 Socket.IO 示例……150
12.5 从服务器发送数据到客户端……152
12.6 将数据广播给客户端……156
12.7 双向数据……160
12.8 小结……163

12.9 问与答……163
12.10 测验……164
12.10.1 问题……164
12.10.2 答案……164
12.11 练习……165

第 13 章 一个 Socket.IO 聊天服务器…… 166

13.1 Express 和 Socket.IO……166
13.2 添加昵称……168
13.2.1 将昵称发送给服务器 ……169
13.2.2 管理昵称列表……171
13.2.3 使用回调来验证……174
13.2.4 广播昵称列表……178
13.2.5 添加消息收发功能……179
13.3 小结……183
13.4 问与答……184
13.5 测验……184
13.5.1 问题……184
13.5.2 答案……184
13.6 练习……185

第 14 章 一个流 Twitter 客户端…… 186

14.1 流 API ……186
14.2 注册 Twitter ……187
14.3 和 Node.js 一起使用 Twitter 的 API ……189
14.4 从数据中挖掘含义……191
14.5 将数据推送到浏览器……194
14.6 创建一个实时的爱恨表……197
14.7 小结……206
14.8 问与答……206
14.9 测验……206
14.9.1 问题……206
14.9.2 答案……206
14.10 练习……207

第 15 章 JSON API …… 208

15.1 API ……208
15.2 JSON ……209
15.3 使用 Node.js 发送 JSON 数据……211
15.4 从 JavaScript 对象创建 JSON …212
15.5 使用 Node.js 消费 JSON 数据……213
15.6 使用 Node.js 创建 JSON API ….216
15.6.1 在 Express 中以 JSON 发送数据……216
15.6.2 构建应用程序……219
15.7 小结……224
15.8 问与答……225
15.9 测验……225
15.9.1 问题……225
15.9.2 答案……225
15.10 练习……226

第 5 部分 探索 Node.js API

第 16 章 进程模块 …… 228

16.1 进程是什么……228
16.2 退出进程以及进程中的错误 ……230
16.3 进程与信号……230
16.4 向进程发送信号……231
16.5 使用 Node.js 创建脚本 ……233
16.6 给脚本传递参数……234
16.7 小结……236
16.8 问与答……236
16.9 测验……237
16.9.1 问题……237
16.9.2 答案……237
16.10 练习……238

第 17 章 子进程模块 …… 239

17.1 什么是子进程……239
17.2 杀死子进程……241
17.3 与子进程通信……242
17.4 集群（Cluster）模块 ……244
17.5 小结……246
17.6 问与答……246
17.7 测验……246
17.7.1 问题……246
17.7.2 答案……246

17.8 练习……247

第 18 章 事件模块 ……248

18.1 理解事件……248
18.2 通过 HTTP 演示事件……251
18.3 用事件玩乒乓……254
18.4 动态编写事件侦听器程序……255
18.5 小结……258
18.6 问与答……258
18.7 测验……259
18.7.1 问题……259
18.7.2 答案……259
18.8 练习……259

第 19 章 缓冲区模块 ……260

19.1 二进制数据初步……260
19.2 从二进制到文本……261
19.3 二进制和 Node.js……262
19.4 Node.js 中的缓冲区是什么？ …264
19.5 写入缓冲区……265
19.6 向缓冲区追加数据……266
19.7 复制缓冲区……267
19.8 修改缓冲区中的字符串……267
19.9 小结……268
19.10 问与答……268
19.11 测验……268
19.11.1 问题……268
19.11.2 答案……269
19.12 练习……269

第 20 章 流模块 ……270

20.1 流简介……270
20.2 可读流……272
20.3 可写流……275
20.4 通过管道连接流……276
20.5 流的 MP3……277
20.6 小结……278
20.7 问与答……278
20.8 测验……279
20.8.1 问题……279
20.8.2 答案……279
20.9 练习……279

第 6 部分 进一步的 Node.js 开发

第 21 章 CoffeeScript ……282

21.1 什么是 CoffeeScript ……282
21.2 安装与运行 CoffeeScript ……284
21.3 为什么要使用预编译器……285
21.4 CoffeeScript 的功能 ……286
21.4.1 最小语法……286
21.4.2 条件和比较……287
21.4.3 循环……288
21.4.4 字符串……289
21.4.5 对象……290
21.4.6 类、继承和 super……291
21.5 调试 CoffeeScript ……294
21.6 对 CoffeeScript 的反应 ……294
21.7 小结……295
21.8 问与答……295
21.9 测验……296
21.9.1 问题……296
21.9.2 答案……296
21.10 练习……296

第 22 章 创建 Node.js 模块……298

22.1 为什么创建模块……298
22.2 流行的 Node.js 模块 ……298
22.3 package.json 文件……299
22.4 文件夹结构……301
22.5 开发和测试模块……302
22.6 添加可执行文件……304
22.7 使用面向对象或者基于原型的编程……305
22.8 通过 GitHub 共享代码……306
22.9 使用 Travis CI……307
22.10 发布到 npm……309
22.11 公开模块……310
22.12 小结……310
22.13 问与答……310
22.14 测验……311
22.14.1 问题……311

22.14.2 答案……311
22.15 练习……311

第 23 章 使用 Connect 创建中间件…… 312

23.1 什么是中间件……312
23.2 Connect 中的中间件……313
23.3 使用中间件的访问控制……317
23.4 按 IP 地址限制访问 ……319
23.5 将用户强制到单个域上……322
23.6 小结……324
23.7 问与答……324
23.8 测验……324
23.8.1 问题……324
23.8.2 答案……325
23.9 练习……325

第 24 章 结合使用 Backbone.js 与 Node.js…… 326

24.1 什么是 Backbone.js……326
24.2 Backbone.js 如何工作 ……327
24.3 一个简单的 Backbone.js 视图……332
24.4 使用 Backbone.js 创建记录 ····336
24.5 小结……337
24.6 问与答……337
24.7 测验……338
24.7.1 问题……338
24.7.2 答案……338
24.8 练习……338

第1部分　入门

第 1 章　Node.js 介绍

第 2 章　npm（Node 包管理器）

第 3 章　Node.js 的作用

第 4 章　回调（Callback）

第 3 章

Node.js 的作用

在本章中你将学到：

- I/O 的意义；
- Node.js 想解决的问题；
- 并发的意义；
- 实现并发的不同方法。

3.1 设计 Node.js 的目的

在对运行 Node.js 程序的方法以及使用 npm 安装模块的方法有了简单的了解之后，本章要讲的是 Node.js 的设计目的。以下是 Node.js 网站提供的对 Node.js 的一段简短描述。

> Node.js 是构建在 Chrome 的 JavaScript 运行时之上的一个平台，用于简单构建快速的、可扩展的网络应用程序。Node.js 使用事件驱动的、非阻塞的 I/O 模型，这让其既轻量又高效，是运行于不同发布设备上的数据密集型实时应用程序的完美平台。

对于一位普通的 Web 开发人员而言，这里有很多让人混淆的术语！在本书编写的时候，Node.js 是 GitHub 上受围观最多的项目，在 Web 开发人员和更为新兴的商业领导者中赚足了眼球。这样的关注度导致的结果是会有大量天花乱坠的宣传，虽然这是受欢迎的，但是也经常让人对 Node.js 是什么有所误解。读者可能还听说过诸如“Node.js 是新的 Rails”或者“Node 是 Web 3.0”这样的说法。这两句话都不正确。Node.js 不是像 Rails 或者 Django 那样的 MVC 框架，也不会每天早晨为你叠被子。在本章的末尾，读者将对 Node.js 是什么以及它的设计目的有一个更为清楚的理解。

3.2 理解 I/O

读者可能听说过与 Node.js 相关的术语“I/O”。I/O 是输入/输出的简写，指的是计算机和人或者数据处理系统之间的通信。可以将 I/O 想成是数据在一次输入和一次输出之间的移动。以下是一些例子。

- 使用键盘敲入文本（输入）并在屏幕上看到文本显示（输出）。
- 移动鼠标（输入）并在屏幕上看到鼠标移动（输出）。
- 将数据传递给外壳脚本（shell script）（输入）并在终端上看到输出。

I/O 的思想可以通过在终端里运行一个程序来演示（见图 3.1）：

```
echo 'Each peach pear plum'
```

在 Mac OSX 或 Linux 下输出会是：

```
each peach pear plum
```

在 Windows 下输出会是：

```
'each peach pear plum'
```

图 3.1
数据流经计算机

TRY IT YOURSELF

我们在这里演示一下输入和输出。

1．打开终端并输入如下命令：

```
echo 'I must learn about Node.js'
```

2．可看到该行内容的回显。

3．注意在这里键盘是输入。

4．注意在这里显示器是输出。

虽然这看起来很简单，但有许多事务在这里运转着。这里所用的程序是 echo，这是一个简单的用于回显任何所提供的文本的工具。从数据移动的角度看，所发生的事情如下。

1．文本字符串被传递给 echo 程序（输入）。

2．文本字符串流经 echo 程序的逻辑。

3．each 程序将其输出到终端（输出）。

3.3 处理输入

在计算机程序里，经常需要来自用户的输入。这可以是命令行脚本提示符，也可以是表

单、电子邮件或者文本信息等形式。计算机编程基本上就是编写解决某一个问题的软件并且处理围绕着该问题的可能的各种不可预测性。比如一个简单的请求用户输入如下信息的 Web 表单：

- 姓；
- 名；
- 电子邮件地址。

当用户提交表单时，数据将会被保存到数据库中并显示在网页上。但是，也会有许多出错的可能，比如：

- 用户没有输入姓；
- 用户没有输入名；
- 用户没有输入电子邮件地址；
- 用户输入的电子邮件地址不合法；
- 用户输入的文本太长，数据库无法保存；
- 用户没有输入任何数据。

对于开发人员而言，识别这些场景并定义对这些场景的响应方法是家常便饭。这是重要的事情，因为这是软件稳定的要求；而且开发人员经常选择为这样的场景编写自动化测试，以便测试他们的代码库，确保代码按期待中的方式工作。以下的示例来自名为 Cucumber 的测试框架，这个框架可以让开发人员使用平实的英语编写测试：

```
Feature: Managing user
  In order to manage users
  As an administrator
  I want to be able to manage users on the system

Scenario: User does not enter a first name
  Given I am on the homepage
  When I press "Save"
  Then I should see "First name can't be blank"
```

这是软件开发的良好方法，因为将软件世界内的路线图标出来并且按照一份可预测的输入列表及其被接受的顺序来编写软件，是可行的。程序的响应方式可经过仔细编写，从而在所收到的输入的基础上发送正确的输出。在这个示例中，软件有一个输入：一个人将数据输入表单。于是，识别场景、编写能够正确响应的代码以及编写用来验证代码的测试，都变得容易了。

当然，计算机程序可以接受超过一个输入。比如，有个游戏控制台——就像任天堂的 Wii 或 Xbox 那样的。自己玩游戏是可能的，但与别人对打就更有趣！让我们假设你的朋友来了，而你正在 Wii 上玩 Mario Kart。在这个游戏里，要选择角色然后在 Go-Karts 里围绕一条赛道跑。马上，关于数据输入的方法的许多可能场景就变得复杂得多。这里的输入可以被认为是两个 Wii 手柄，而输出则是一台电视机（见图 3.2）。不像上一个示例里的一名用户和一个表单，现在有许多事情需要考虑，比如：

- 两名游戏者；

- 两个 Wii 遥控器，每个各有 8 个数字按钮。

由于游戏者可以以任意方向在任何时间移动遥控器并按 8 个按钮中的任何一个，有时候还组合上述动作，所以要想解决所有可能发生的场景数量就是件巨大的任务。要精确预测用户玩游戏的方式以及事情的发生顺序不容易。

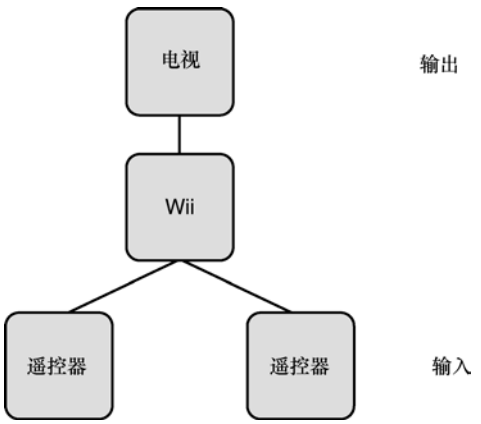


图 3.2
两个输入让事情复杂

有些游戏允许游戏者连接 Internet 并与其他游戏者在线游戏。在诸如 Xbox 上的 Battlefield 这样的游戏里，游戏者可以在 Internet 上通过语音和文本来沟通。软件面对的输入数量很快成了天文数字，比如：

- 可能会有上百万个游戏者；
- 可能会有上百万个 Xbox 手柄；
- 可能会有上百万个耳麦；
- 游戏者在 3D 虚拟世界中随意移动。

人类的不可预测性太过于优美，要想识别出每件可能发生的事情及其顺序就成了不可能的任务。读者可能还注意到，通过连接到 Internet，更多人类输入都会被加入到软件中，而即便如此，还有其他一些输入和输出，比如：

- 负载平衡器；
- 数据库服务器；
- 语音服务器；
- 消息服务器；
- 游戏服务器。

可能还有更多。要注意的是，对于这类软件来说要想识别各种场景及其顺序将是极为复杂的事情，因为这里有巨大数量的，包括了人类和网络在内的各种变量（见图 3.3）。

在开发 Web 软件时，从历史上看，开发人员能够可靠地预测输入和输出，这在编程风格上反映了出来。Web 最初是以一种读取 HTML 文档的方法来设计的，HTML 文档储存在服务器上，任何人只要有 Internet 连接就可以通过 Web 服务器来访问。用图来表示很简单（见图 3.4）。

随着 Web 越来越成熟，给基于 Web 的软件加入数据库和脚本语言越来越普遍（见图 3.5）。

这可在无需大量增加输入和输出的复杂性的情况下极大地增加了软件开发人员所能做的事情的可能性。预测输入如何被使用并且将场景数量映射到代码上相对容易。

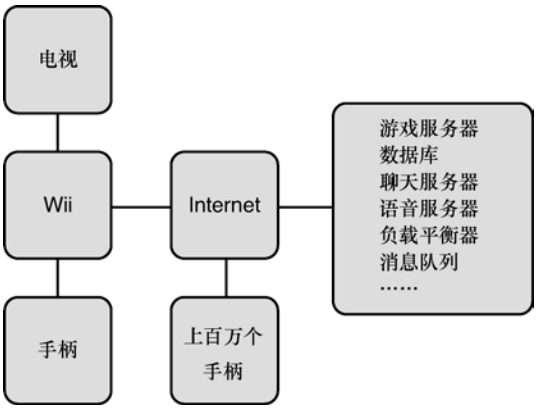


图 3.3
输入和输出变得复杂

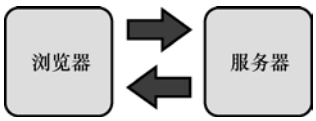


图 3.4
提供 HTML 页面服务的 Web 服务器

图 3.5
增加数据库服务器

今天，Web 应用程序的设计要复杂得多得多。I/O 是碎片化的，而且 I/O 操作更为频繁。

- 与第三方应用程序编程接口（API）的交互繁重。
- 许多不同设备发送与接收数据，包括移动设备、电视和公告板。
- 巨大数量的客户同时连接并实时交互。

与早先的 Battlefield 示例很像，输入输出图现在看起来要复杂得多了（见图 3.6）。这就是 Node.js 所感兴趣的并且提供了一种解决问题的方法。

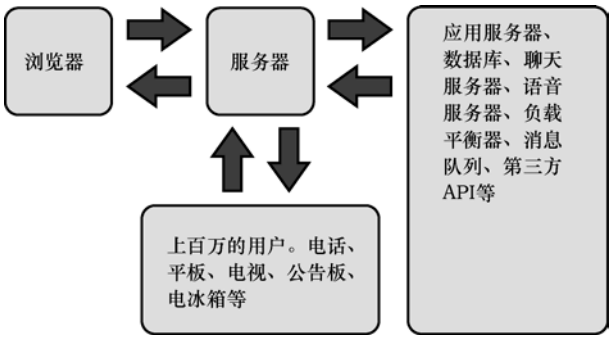


图 3.6
复杂的 Web 应用程序

3.4 联网的 I/O 是不可预测的

在对 I/O 的思想以及现代应用程序的复杂性有了更多的理解之后，重要的是要理解 Web

应用程序的 I/O 是不可预测的，尤其在与时间有关的时候。为了演示这个问题，我们将运行一个小的 Node.js 程序来从不同的 Web 服务器获取主页（见程序清单 3.1）。如果不能完全理解代码也不用担心，因为在第 5 章的“HTTP”中将讲解它。

程序清单 3.1 演示网络 I/O

```
var http = require('http'),
    urls = ['shapeshed.com', 'www.bbc.co.uk', 'edition.cnn.com'];

function fetchPage(url) {
  var start = new Date();
  http.get({ host: url }, function(res) {
    console.log("Got response from: " + url);
    console.log('Request took:', new Date() - start, 'ms');
  });
}

for(var i = 0; i < urls.length; i++) {
  fetchPage(urls[i]);
}
```

在这个示例中，我们要求 Node.js 访问三个 URL 并报告收到响应的情况以及所耗费的时间。当程序运行时，输出会打印在终端上：

```
Got response from: edition.cnn.com
Request took: 188 ms
Got response from: www.bbc.co.uk
Request took: 252 ms
Got response from: shapeshed.com
Request took: 293 ms
```

这里的输入是来自三个不同 Web 服务器的响应，Node.js 将输出发送到终端上。如果再次运行同样的代码，虽然我们会期望得到相同的结果，但看到的却是不同的输出：

```
Got response from: edition.cnn.com
Request took: 192 ms
Got response from: shapeshed.com
Request took: 294 ms
Got response from: www.bbc.co.uk
Request took: 404 ms
```

如果读者自己运行这段代码，会看到响应的顺序改变了，或者至少响应时间改变了。这是因为响应时间不是一成不变的。Web 服务器响应的时间会随着如下这些因素中的某些因素的不同而变得极为不同。

- 解析 DNS 请求的时间。
- 服务器的繁忙程度。
- 要应答的数据有多大。
- 服务器和客户的可用带宽。
- 为响应而服务的软件的效率。
- 所使用的网络的繁忙程度。

- 数据要传输多远。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour03/example01 找到。下面我们演示网络 I/O。

1．将程序清单 3.1 中的代码复制到系统上以 app.js 为名的文件中。

2．从终端运行它：

```
node app.js
```

3．观察来自服务器的响应。

4．从终端再次运行它：

```
node app.js
```

5．检查输出并比较响应时间。

这个简单的示例演示了与时间相关时，基于网络的 I/O 是不可预测的。

3.5 人类是不可预测的

如果读者为浏览器开发过 JavaScript 程序，就会理解在编写给人类用来与 Web 页面交互的代码的时候，要使用不同的编程风格。要说出人类执行某个动作的顺序或时间是不可能的。这里是一些 JavaScript 代码，用于在用户单击 id 为“target”的链接时显示提示：

```
var target = document.getElementById("target");
target.onclick = function(){
  alert('Hey you clicked me!');
}
```

如果读者更熟悉 jQuery，那么以 jQuery 写同样的代码就会是：

```
$('#target').click(function() {
  alert('Hey you clicked me!');
});
```

上述两个示例都在用户单击链接时在浏览器中显示提示信息。这些示例创建了对单击事件的侦听器（listener）并将其绑定到超文本标记语言（HTML）或者文档对象模型（DOM）中的一个元素上。当用户单击链接时，该事件被触发，提示信息就会被显示。这段代码并不是对一组用户可能进行的动作按线性排序列出，然后以此构架代码；而是围绕事件来构架。事件可在任何时刻发生，也可发生不止一次。我们将此描述为事件驱动的编程，因为在程序中要是有事情发生的话那么有个事件必须发生。事件驱动编程是处理不可预测性的极佳方式，因为我们可以识别将要发生的事件，即使我们并不知道事件什么时候会发生。

JavaScript 在浏览器中极高效地使用了这个模型，允许开发人员创建基于浏览器的富应用程序，这样的应用程序围绕着事件和用户与页面之间的交互方式编写。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour03/example02 找到。下面我们演示 JavaScript 的代码执行。

1．创建一个名为 index.html 的新文件并加入下列内容：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>JavaScript Events</title>
    <style type="text/css">
      p { width: 200px; }
    </style>
    <!--[if lt IE 9]><script src="http://html5shiv.googlecode.com/svn/trunk/
➥html5.js"></script><![endif]-->
    <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/
➥jquery/1.7.1/jquery.min.js"></script>
    <script type="text/javascript">
      $(document).ready(function() {
        $('#click-trigger').click(function() {
          alert('You triggered the click event');
          });
        $("p").mouseover(function () {
          alert('You triggered the mouseover event');
        });
      });
    </script>
  </head>
  <body>
    <h1>JavaScript Events</h1>
    <h2>Click events</h2>
    <button id="click-trigger">Click me</button>
    <h2>Mouseover events</h2>
    <p>Move your mouse over this paragraph</p>
  </body>
</html>
```

2．使用 Web 浏览器打开这个文件。

3．单击按钮。注意随着单击事件的触发会有提示框显示。

4．鼠标移过文本段落。注意随着鼠标移过（mouseover）事件的触发会有提示框显示。

3.6 处理不可预测性

读者已经看到了，现在的 Web 应用程序，和仅仅给浏览器提供 HTML 页面服务相比，要复杂得太多了。以下是现代 Web 应用程序的一些趋势。

- 许多不同类型的设备可连接到 Web 应用程序。
- 设备可作为输入和输出。
- 在一个应用程序内，不同的服务由不同的服务器来完成。

- 应用程序与许多第三方数据源的交互很繁重。
- 客户与服务器之间的数据实时双向流动。

所有这些趋势都指向并发，这是计算机学科中著名的难题。并发这个术语描述的是事情会在同时发生并可能互相交互。ql.io 是 Node.js 用例的很好的例子，这是由 eBay 创建的给开发人员提供进入多个 eBay 数据源的单一接口的服务。这个服务使得我们可以很容易地跨不同 eBay 组织请求数据。这个服务使用 Node.js 创建。其技术团队的负责人 Subbu Allamaraju 提到，选择使用 Node.js 将他们从一些传统上与并发有关的问题中解放出来。

> Node 的事件化的 I/O 模型让我们无需担心互锁和并发这两个在多线程异步 I/O 中常见的问题。

Node.js 将 JavaScript 解决不确定性所用的事件驱动方法加入到解决并发编程的可能方法清单中。事件驱动编程并不是新的思想。比如 Python 的 Twisted 和 Ruby 的 Event Machine 都是与 Node.js 相似的服务器技术。解决并发问题的其他方法还包括线程以及使用不同的进程。让 Node.js 与众不同的是，它给开发人员提供的用于处理并发的语言是 JavaScript。JavaScript 是一种事件驱动的语言，旨在能够对外界的事件作出响应。虽然我们绝对可以使用 Ruby 或 Python 写出事件驱动的代码，但如果使用事件方式是解决并发问题最好的方法，那么考虑一种围绕着事件来设计的语言，至少是值得的。

By the Way

注意：有一些对 Node.js 不友好的反应

最初，其他语言的开发人员对使用 JavaScript 来解决并发问题嗤之以鼻。有人指责 JavaScript 就是个玩具语言，它是给浏览器设计的而不是给服务器设计的，在单个进程中想解决并发是愚蠢的。如果读者有兴趣了解更多这方面的争议，可搜索“Node.js is Cancer（Node.js 是毒瘤）”看看。

3.7 小结

在本章，我们对 Node.js 是什么以及 Node.js 旨在解决什么问题做了更多介绍。读者理解了 Web 应用程序已经从服务单个 HTML 页面变成了复杂得多的系统。我们介绍了 I/O 的思想以及在现代 Web 应用程序中输入和输出的数量如何巨大。我们了解了，在软件开发中要按时间和顺序预测人类的行为是困难的，也看到了 JavaScript 如何通过提供事件驱动方法来响应的思想。而后我们学习了并发的思想，这是 Node.js 想要解决的主要问题之一。

我们介绍了这么一个思想：并发是软件开发中一直存在的问题，Node.js 是对该问题的一个响应，尤其是在网络环境中。在本章中，我们学习了更多关于 Node.js 尝试解决的是什么的知识。在下一章，我们将学习并发的解决方法。

3.8 问与答

问：我在开发内容不多的小网站，Node.js 适合吗？

答：你绝对可以使用 Node.js 创建小网站，而且已经有许多框架可以帮助你。不过还是要指出，在设计和创建 Node.js 时，并没有考虑这样的要求。

问：并发似乎难以理解，如果我不能完全理解它会有问题吗？

答：并发这个概念很难，在将来的章节中我们将介绍一些实际的示例。现在，理解并发只需知道它指的是许多人同时尝试做同样的事情。

问：I/O 与我在 Web 上看到的.io 域名有关吗？

答：是的。实时和 Node.js 应用程序的开发人员认识到他们需要在应用程序中频繁处理 I/O。.io 域名实际上与计算机学科没有任何关系，它是印度洋的域名。读者会看到这个域名用在了与实时软件相关的地方，比如 http://socket.io。

3.9 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

3.9.1 问题

1．与 Web 站点只是 HTML 文档的那个时候不同，现在的 Web 应用程序有哪些方面的变化？

2．你觉得 Node.js 适合哪些范畴？

3．为什么 JavaScript 是一个事件驱动的语言？

3.9.2 答案

1．Web 应用程序变得更为复杂的地方包括了脚本语言和数据库的加入等。越来越多的数据现在分布在 Web 的不同地方，使用 API 和网络粘合在一起。

2．当应用程序需要在网络上发送和接收数据时 Node.js 最为适合。这可能是第三方的 API、联网设备或者浏览器与服务器之间的实时通信。

3．JavaScript 围绕着最初与文档对象模型（DOM）相关的事件构架。开发人员可以在事件发生时做事情。这些事件有用户单击一个元素、页面完成加载等。使用事件，开发人员可以编写事件的侦听器，当事件发生时被触发。

3.10 练习

1．如果读者开发过网站或者软件，请选择一个然后画出其输入输出图。理解代表输入和输出的设备或事物。

2．选一个你最喜欢的计算机游戏并尝试理解该游戏的输入和输出。有多少个输入？是否有会同时发生或者几乎同时发生的事情？

3．花一些时间阅读以下两篇文章：“Node.js is good for solving problems I don’t have（Node.js 适于解决我所没有的问题）”（http://bit.ly/LyYFMx）以及“Node.js is not a cancer, you are just a moron（Node.js 不是毒瘤，你真是个二师兄）”（http://bit.ly/KB1HcW）。如果不能完全理解这些文章也不必担心，只要理解 Node.js 用来解决并发的方法是革新的、有争议的并且引发了有益的争论。

第 11 章

部署 Node.js 应用程序

在本章中你将学到：

- 云托管以及平台即服务（Platform as a Service，PaaS）的意义；
- 部署到 Heroku；
- 部署到 Cloud Foundry；
- 部署到 Nodester。

11.1 准备好部署

不用多久，你就会想部署你的 Node.js 项目，这样就能与整个世界一起分享它，然后就可以成名发财了！你需要在 Web 上找个地方来托管 Node.js 应用程序。如果你有 UNIX 技能，可能会想自己完成这个任务，但你更可能是只想部署应用程序，而不是构建并维护一台服务器。在本章，我们将了解如何将站点部署到大量支持 Node.js 的云托管提供商那里。

11.2 在云上托管

读者可能听说过云计算这个词，它描述的是通过 Internet 来交付计算服务。云计算的示例有：

- 在 Amazon S3 上托管文件；
- 使用 Dropbox 在许多计算机上共享文件；
- 使用 Internet 上诸如 Gmail 这样的服务访问邮件；
- 自动备份到诸如 iCloud 这样的服务上。

云计算的基本思想是，数据不是储存在自己的硬件上，而是储存在别人的硬件上，而后

可使用 Internet 来访问数据。云计算有许多优点，如下所示。

- 自己无需负责服务的运行和维护。
- 云服务的启动与运行要比自己构建基础设施更快。
- 通常更便宜。
- 将自己无法提供的那部分基础设施外包给提供商。

许多开发人员不具备构建以及维护 Web 服务器的技能，所以使用云提供商可带来许多优势，如下所示。

- 无需负责构建服务器。
- 无需负责网络。
- 托管专业人员已经为你创建了服务。

云托管服务主要是指，我们开发应用程序，将其指向某个云托管提供商，部署，然后忘记它。用来描述提供部署和托管解决方案的服务的术语是平台即服务（PaaS）。

这些服务会管理部署和托管你的网站的端到端过程，而且通常包括许多其他与托管有关的服务，比如备份和数据库。在 Node.js 的 PaaS 市场上有许多参与者，其中有许多都为 Node.js 应用程序提供免费的托管服务。

在本章，我们使用一个简单的 Express 应用程序作为示例应用程序来部署。当然，读者要是有自己的应用程序，欢迎部署你自己的应用程序！

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour11/example01 中找到。以下是一个 Node.js 应用程序示例。

1. 下载代码示例并打开 hour11/example01 文件夹，可以看到一个简单的 Express 应用程序。
2. 为这个应用程序安装依赖模块：

```
npm install
```

3. 打开浏览器并浏览 http://127.0.0.1:3000。
4. 可看到这个应用程序示例。这是我们要在本章部署的应用程序（见图 11.1）。

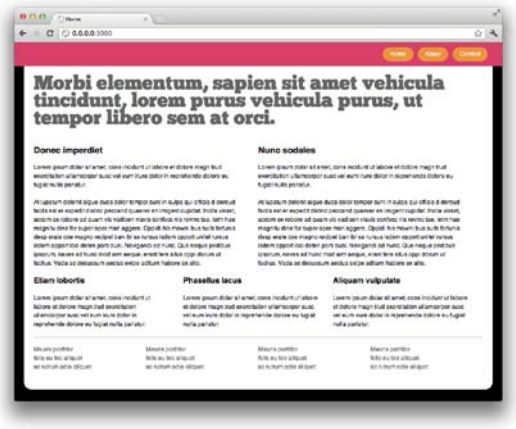


图 11.1

将要在本章部署的 Express 应用程序示例

11.3 Heroku

Heroku 是首批服务即平台（PaaS）提供商之一，在开发人员中广受欢迎。这个服务围绕着基于 Git 的工作流设计，所以如果读者熟悉用于版本控制的 Git，部署就非常简单。这个服务原本是为托管 Ruby 应用程序而设计，但 Heroku 之后加入了对 Node.js、Clojure、Java、Python 和 Scala 的支持。Heroku 的基础服务是免费的。

11.3.1 注册 Heroku

为了使用 Heroku 服务，必须先注册一个账号。请按如下步骤进行。

1. 访问 http://api.heroku.com/signup，网站会要求用户输入电子邮件地址（见图 11.2）。

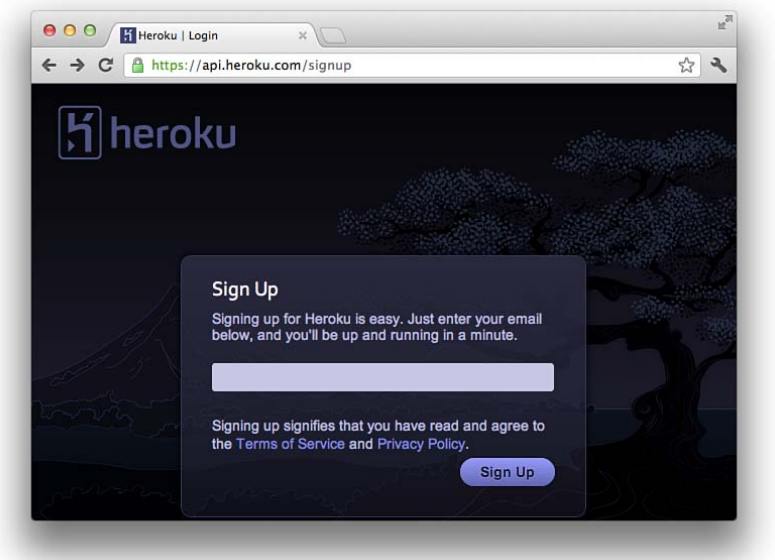


图 11.2
注册 Heroku

2. 一旦成功输入电子邮件地址，网站会邀请你检查邮件，邮件中有确认链接。

3. 打开电子邮件，进入所提供的 Heroku 链接，会邀请你选择密码。

4. 按照 http://devcenter.heroku.com/articles/quickstart#step_2_install_the_heroku_toolbelt 中的指南安装 Heroku 工具条。它提供了能让我们将站点部署到 Heroku 的命令行工具。

5. 一旦完成了适用于自己平台的安装之后，最后要做的事情就是登录账号。为了完成这一过程，请从要求你输入证书的终端上运行 heroku login。注意，如果这是你第一次登录，会为你生成一个 SSH 公共密钥。Heroku 以此来管理对服务的访问。

```
heroku login
Enter your Heroku credentials.
Email: george@shapeshed.com
Password:
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/george/.ssh/id_rsa.pub
```

搞定！你已经有了一个 Heroku 账号，可以准备部署了！

By the Way

SSH 密钥是密码的替代

SSH 密钥通常用于授予用户访问服务器的权限。可将它们用在某些配置中，以便允许无需密码即可访问服务器。许多 PaaS 提供商都使用了公共密钥。

TRY IT YOURSELF

按照如下步骤注册 Heroku 账号。

1．在 http://api.heroku.com/signup 注册 Heroku 账号。

2．访问邮件中的链接并选择密码。

3．从 http://devcenter.heroku.com/articles/quickstart#step_2_install_the_heroku_toolbelt 为你的平台安装安装程序。

4．打开终端窗口并登录到 Heroku。

```
heroku login
```

11.3.2 为 Heroku 准备应用程序

有了账号并在计算机上设置好了 Heroku 之后，就可部署应用程序了。为了让 Express 应用程序能够工作，需要对其做点小更改，因为 Heroku 会随机分配端口，供应用程序使用。

如果站点位于 Heroku 上，在 app.js 文件的顶部加一行代码来正确设置端口：

```
var port = process.env.PORT || 3000;
```

然后将下列行

```
app.listen(3000);
```

替换成

```
app.listen(port);
```

如果应用程序运行在 Heroku 上，这就正确地设置了端口；而如果应用程序不在 Heroku（或者本地计算机）上，则继续使用 3000 端口。

Heroku 有一个名为 Foreman 的工具用来管理进程，用 Procfile 文件来声明应该运行什么。为了将应用部署到 Heroku，必须在应用程序的根目录下添加一个名为 Procfile 的文件，如下所示：

```
web: node app.js
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour11/example02 找到。以下是准备 Node.js 应用程序以便部署到 Heroku 的方法。

1．制作一个 hour11/example01 代码示例所提供的 Express 应用程序的副本。

2．在 app.js 文件的顶部，添加如下行：

```
var port = (process.env.PORT || 3000);
```

3．将下列行从 app.js 中移除：

```
app.listen(3000);
```

4．用下列行替换它：

```
app.listen(port);
```

5．在应用程序的根目录中，添加一个名为 Procfile 的文件并加入如下内容：

```
web: node app.js
```

6．用如下命令安装依赖模块：

```
npm install
```

7．启动应用程序并检查其是否运行正常：

```
node app.js
```

11.3.3 将应用程序部署到 Heroku

现在，我们已经准备好部署应用程序了！如果读者使用自己的应用程序而不是我们所提供的示例 Express 应用程序，则需要使用 package.json 文件来声明依赖模块，因为 Heroku 也使用它。

为了使用 Heroku，必须使用 Git，这是一个版本控制工具。Git 会由 Heroku 安装程序安装，所以如果安装了 Heroku 安装程序，Git 就已经在系统上了！Node.js 社区上的许多开发人员使用 Git 来管理源代码。

Heroku 建议不将 node_modules 文件夹加入到 Git 中，这可通过创建一个带有如下内容的.gitignore 文件来实现：

```
node_modules
```

现在可以创建一个 Git 库，并通过从应用程序的根目录运行如下命令添加源代码了。

```
git init
git add .
git commit -m 'initial commit'
```

接下来，应用程序会在 Heroku 上创建。雪松堆栈（cedar stack）会被声明，因为它支持 Node.js。

```
heroku create --stack cedar
```

这会为我们创建站点并自动设置好需要部署的一切。我们会看到如下输出：

```
Creating afternoon-light-5818... done, stack is cedar
http://afternoon-light-5818.herokuapp.com/ | git@heroku.com:afternoon-light-5818.
git
Git remote heroku added
```

最后，可以使用如下命令部署应用程序：

```
git push heroku master
```

而后，就可以访问在 Heroku 创建步骤中所生成的 URL 并看到你的站点了！

TRY IT YOURSELF

为了将 Node.js 应用程序部署到 Heroku，请按如下步骤进行。

1．返回示例 Express 应用程序。

2．使用如下命令创建 Git 库：

```
git init
git add .
git commit -m 'initial commit'
```

3. 使用如下命令在 Heroku 上创建应用程序。注意从这一命令返回的 URL：

```
heroku create --stack cedar
```

4. 用如下命令将站点发布到 Heroku：

```
git push heroku master
```

5. 访问之前得到的 URL，可看到所部署的网站了！

如果需要在将来更新站点，只需将新文件提交给 Git 然后推送到 Heroku 即可：

```
git push heroku master
```

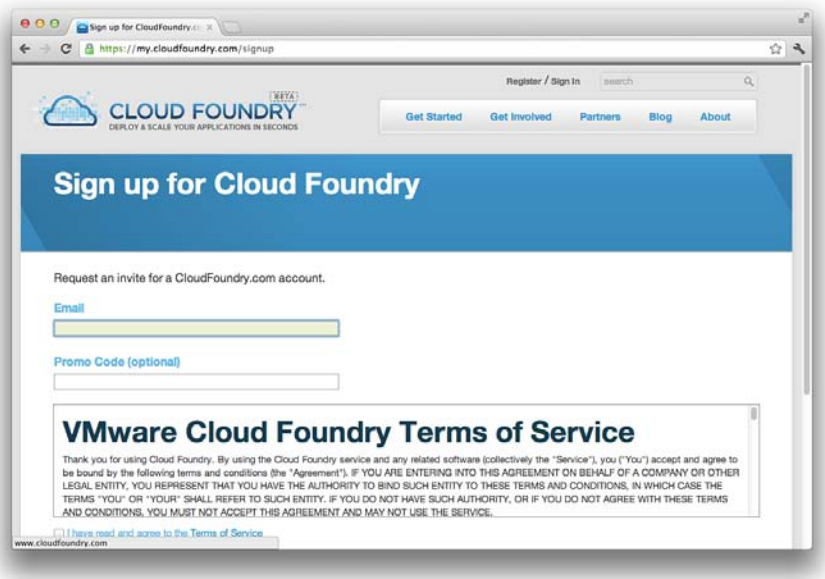
11.4 Cloud Foundry

Cloud Foundry 是另外一个支持 Node.js 的 PaaS 提供商。Cloud Foundry 和其他服务的主要不同是它不依赖 Git，所以如果读者不喜欢将 Git 作为部署过程的一部分，它可能是个好选择。

11.4.1 注册 Cloud Foundry

为了注册这一服务，请访问 https://my.cloudfoundry.com/signup 并输入用户详细信息（见图 11.3）。而后会收到一个确认链接，邀请用户选择密码。

图 11.3
注册 Cloud Foundry



Cloud Foundry 需要在系统上安装有 vmc，这是一个用于与服务交互的命令行工具。而这需要在系统上安装 Ruby 和 RubyGems。许多人可能会因此而放弃使用这一服务，不过 Cloud Foundry 的确提供了能让所有一切安装好的综合文档，地址是 http://start.cloudfoundry.com/tools/

vmc/installing-vmc.html。

Cloud Foundry 是开源的

Cloud Foundry 平台是开源的，可用于创建自己的私有云。这就意味着，只要你愿意，就可以在自己的服务器上运行一个 Cloud Foundry 软件的实例！不过，通过使用 Cloud Foundry 的服务器，就无需运行自己的服务器了。

Did you Know?

TRY IT YOURSELF

为了创建 Cloud Foundry 账户，请按如下步骤进行。

1．在 https://my.cloudfoundry.com/signup 注册一个 Cloud Foundry 账户。

2．访问邮件中的链接并添加密码。

3．按照 http://start.cloudfoundry.com/tools/vmc/installing-vmc.html 中的指南安装 vmc 及相关软件。

11.4.2 为 Cloud Foundry 准备应用程序

为了给 Cloud Foundry 准备应用程序，需要对 app.js 文件做一些修改。如果读者完成了 Heroku 示例，则应该在一个全新的示例应用程序副本上操作（可在 hour11/example01 找到）。

和 Heroku 一样，Cloud Foundry 动态分配端口号，所以需要更新 app.js 文件以反映这一要求。在 app.js 文件的顶部添加以下内容：

```
var port = (process.env.VMC_APP_PORT || 3000);
```

接下来，将内容为

```
app.listen(3000);
```

的行修改为：

```
app.listen(port);
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour11/example03 找到。以下是准备 Node.js 应用程序以便部署到 Cloud Foundry 的方法。

1．制作一个 hour11/example01 代码示例所提供的 Express 应用程序的副本。

2．在 app.js 文件的顶部，添加如下行：

```
var port = (process.env.VMC_APP_PORT || 3000);
```

3．将下列行从 app.js 中移除：

```
app.listen(3000);
```

4．用下列行替换它：

```
app.listen(port);
```

5．用如下命令安装依赖模块：

```
npm install
```

6．启动应用程序并检查其是否运行正常：

```
node app.js
```

11.4.3 将应用程序部署到 Cloud Foundry

要部署应用程序，需要使用 vmc 工具。这是一个用于将站点部署到 Cloud Foundry 的命令行工具。首先，将目标设置到 Cloud Foundry 服务器：

```
vmc target api.cloudfoundry.com
```

接下来，必须添加登录证书：

```
vmc login
```

这会提示用户输入用户名和密码并告知是否成功：

```
Attempting login to [http://api.cloudfoundry.com]
Email: george@shapeshed.com
Password: ********
Successfully logged into [http://api.cloudfoundry.com]
```

Cloud Foundry 不支持通过 npm 安装 Node 模块，所以必须在发布站点之前确保这些模块都已经安装。如果还没有安装，运行如下命令：

```
npm install
```

在 vmc 中目前有一个 bug，所以在运行 npm install 之后，还需要运行如下命令：

```
rm -r node_modules/.bin/
```

要从项目的根目录部署站点，运行如下命令：

```
vmc push
```

这会提示一组问题并告知应用是否成功部署：

```
Would you like to deploy from the current directory? [Yn]: y
Application Name: yourappname
Application Deployed URL [yourappname.cloudfoundry.com]:
Detected a Node.js Application, is this correct? [Yn]: Y
Memory Reservation (64M, 128M, 256M, 512M, 1G) [64M]:
```

而后可以在 http://yourappname.cloudfoundry.com 访问你的应用程序，其中“yourappname”是你对自己的应用程序所起的名称。

如果需要在将来更新应用程序，可以将目录更改到包含源代码的文件夹并运行下述命令：

```
vmc update yourappname
```

TRY IT YOURSELF

以下是将 Node.js 应用程序布署到 Cloud Foundry 的方法。

1．返回示例 Express 应用程序。

2．将 vmc 的目标设置为 Cloud Foundry 服务器：

```
vmc target api.cloudfoundry.com
```

3．输入登录证书：

```
vmc login
```

4．确保所有的依赖模块都已使用 npm 安装：

```
npm install
```

5．用下列命令修补 vmc 的 bug：

```
rm -r node_modules/.bin/
```

6．部署站点：

```
vmc push
```

7．访问显示在输出中的 URL，可看到所部署的网站了！

11.5 Nodester

Nodester 是另外一个支持 Node.js 的平台即服务提供商。它是一个开源软件。这个服务目前是完全免费的！Nodester 声称用户只需 1 分钟即可部署 Node.js 应用程序！

11.5.1 注册 Nodester

要想使用 Nodester 服务，必须首先注册一个账号。按如下步骤注册账号。

1．访问 http://nodester.com/并输入电子邮件地址，请求一张注册券（见图 11.4）。注册券由 Nodester 的维护人员定期发送，所以有可能需要等待几天才能收到。

图 11.4 从 Nodester 请求一张令牌

2．收到券之后，就可以创建账户。首先，需要从 npm 安装 nodester 命令行工具：

```
npm install nodester-cli -g
```

3．Nodester 使用 cURL 来发送请求。对于许多开发人员来说，这可能过于技术性了，但如果你熟悉 cURL，可使用如下 cURL 请求来创建账户：

```
curl -X POST -d
"coupon=yourcoupon&user=youruser&password=123&email=your@email.com&rsakey=ssh-
➥ rsa
AAAA..." http://nodester.com/user
```

4．如果没有在系统上安装 cURL 或者不想使用 cURL，那么也可在 http://nodester.com/help.html 上输入这些信息。RSA 密钥是需要输入的信息之一。如果运行 UNIX 类型的系统（OSX 或者

Linux），那么可以使用如下命令生成一个密钥：

```
ssh-keygen -t rsa -C "your@email.com"
```

5．如果使用 Windows，可从 http://code.google.com/p/msysgit/downloads/list 上的安装程序安装 Windows 版的 Git。如何安装这个软件，在 http://help.github.com/win-set-up-git/上有完整的说明。安装好了这个软件之后，可以运行下述命令：

```
ssh-keygen -t rsa -C "your@email.com"
```

6．使用上述方法生成 RSA 密钥之后，它将输出文件所保存的位置。为了使用 Nodester，需要打开 id_rsa.pub 文件并复制其内容。用户既可以将内容添加到第 3 步提到的 cURL 命令中，也可将其粘贴到 http://nodester.com/help.html 表单上。

7．一旦有了账户，就可以设置本地计算机，以便将应用程序部署到 Nodester 上。注意要将这里的<*username*>和<*password*>替换成你自己的证书。

```
nodester user <username> <password>
nodester info verifying credentials
nodester info user verified..
nodester info writing user data to config
```

TRY IT YOURSELF

要想注册 Nodester 账号，请执行如下步骤。

1．从 http://nodester.com 请求一张 Nodester 券。

2．收到券之后，如果需要的话，生成一个 RSA 密钥：

```
ssh-keygen -t rsa -C "your@email.com"
```

3．使用 cURL 请求或者在 http://nodester.com/help.html 填写表单，完成注册过程：

```
curl -X POST -d "coupon=yourcoupon&user=youruser&password=123&email=your@
➥ email.com&rsakey=ssh-rsa AAAA..." http://nodester.com/user
```

4．确认在本地计算机上安装了 Nodester CLI：

```
npm install nodester-cli -g
```

5．为 Nodester 设置本地计算机：

```
nodester user <username> <password>
```

11.5.2 为 Nodester 准备应用程序

要在 Nodester 上创建新的应用程序，请运行如下命令：

```
nodester app create yourapp app.js
```

在这个命令中，应用程序的名称被设为 yourapp，主 Node.js 文件被声明为 app.js。接下来，设置应用程序，为在 Nodester 上部署做好准备：

```
nodester app init yourapp
```

这会在 Nodester 上创建一个远程 Git 库并设置好本地计算机。它也启动了一个 Hello World 应用程序，所以如果访问 http://yourapp.nodester.com，就可以看到一些东西了！还应该看到在文件系统中添加了一个新的“yourapp”文件夹（或者你所命名的名称）。如果查看这一文件夹，可看到只有一个名为 app.js 的 Hello World 应用程序文件。为了发布示例 Express 站点，只需

将代码复制到这个文件夹中即可。如果读者完成了 Heroku 或者 Cloud Foundry 的示例的话，应该在一个全新的示例应用程序副本上操作（可在 hour11/example01 找到）。

就如 Heroku 和 Cloud Foundry 一样，站点所使用的端口号是动态分配的，所以需要更新 app.js 文件以反映这一要求。在 app.js 文件的顶部添加以下内容：

```
var port = process.env.app_port || 3000;
```

删除有下列内容的行：

```
app.listen(3000);
```

将其替换为：

```
app.listen(port);
```

11.5.3 将应用程序部署到 Nodester

要想从“yourapp”文件夹在 Nodester 上安装依赖模块，运行如下命令：

```
nodester npm install
```

这个命令会读取 package.json 文件并安装任何包。

最后，可以将文件添加到 Git 并推送发布站点！

```
git add .
git commit -m 'adding site files'
git push origin master
```

如果打开浏览器并浏览 http://yourapp.nodester.com，这里的“yourapp”是应用程序的名称，就应该看到你的站点了！

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour11/example04 找到。为了将 Node.js 应用程序部署到 Nodester，请按照如下步骤进行。

1. 创建一个新的 Nodester 应用程序。用应用程序名称替换 yourapp：

   ```
   nodester app create yourapp app.js
   ```

2. 初始化 Nodester 应用程序。将 yourapp 更改为在第 1 步中所用的应用程序名称：

   ```
   nodester app init yourapp
   ```

3. 将 hour11/example01 中的代码示例文件复制到在第 2 步中创建的文件夹中。这个文件夹的名称就是用来替代 yourapp 的名称。

4. 在 app.js 文件的顶部，添加如下一行：

   ```
   var port = process.env.app_port || 3000;
   ```

5. 将下行从 app.js 中移除：

   ```
   app.listen(3000);
   ```

6. 用如下内容替换它：

   ```
   app.listen(port);
   ```

7. 用下列命令安装依赖模块：

   ```
   nodester npm install
   ```

8．将文件添加到 Git 库中：

```
git add .
git commit -m 'adding site files'
```

9．推送文件以便发布站点：

```
git push origin master
```

10．访问显示在输出中的 URL，可看到所部署的网站了！

11.6 其他 PaaS 提供商

Node.js 的托管市场增长迅速，还有其他一些服务提供商可供使用，如下所示。

- Nodejitsu——http://nodejitsu.com/。
- Cure——http://cure.willsave.me/。
- Joyent——http://node.de。
- Windows Azure——https://www.windowsazure.com。

完整的 Node.js 托管提供商清单可见 https://github.com/joyent/node/wiki/Node- Hosting。

11.7 小结

本章探究了在云上托管 Node.js 应用程序的思想。我们介绍了 Heroku、Cloud Foundry 以及 Nodester 这些 Node.js PaaS 提供商，然后讲解了如何在这些平台上部署应用程序，也可以看到其中有一些遵循 Git 工作流进行部署，而 Cloud Foundry 却不是。我们还看到，对于每个提供商，我们都需要对主应用程序做一些小更改（在本例中是 app.js），以便应用程序能运行在这些服务商上。最后，我们了解到有许多 PaaS 提供商都支持 Node.js！

11.8 问与答

问：我应该使用哪个 PaaS 提供商？

答：选择哪个 PaaS 提供商取决于应用程序做的是什么以及估计有多少用户会使用它。Nodester 非常适合于快速原型建立以及与用户和其他开发人员共享应用程序。目前无需花费任何金钱就可使用 Nodester，但以我的经验来看，Nodester 的正常运行时间无法保证。Heroku 是另外一个免费的服务，它分配一定数量的 RAM（随机存储器）给你的 Node.js 进程。如果你的应用程序很受欢迎，那么你就可能需要添加更多的进程，这时就该付费了。Heroku 和 Nodester 都依赖于 Git，如果也鼓励你使用 Git 来做源代码的版本控制。如果不想使用 Git，那么 Cloud Foundry 可能就是一个好选择了。

问：我应该使用哪个数据库呢？

答：在部署的示例 Express 应用程序中，没有使用数据存储。大多数服务即平台提供商提供这样那样的访问数据库的方法。Heroku 支持 PostgreSQL 并通过第三方插件支持大量其他数据存储。Cloud Foundry 支持 MongoDB、MySQL、PostgreSQL、RabbitMQ 和 Redis。Nodester 不提供数据存储，但可通过 Heroku 使用许多第三方服务。

问：我应该使用哪个 WebSocket 呢？

答：对 WebSocket 的支持随提供商的不同而不同。在本书编写时，Heroku 和 Cloud Foundry 不支持 WebSocket，但 Nodester 支持。最新的情况请参考 https://github.com/joyent/node/wiki/Node-Hosting。

问：我可否在自己的服务器上托管 Node.js 站点？

答：如果你有 UNIX 或者 Windows 管理技能，可预备一个 VPS（虚拟私有服务器），安装 Node.js，然后开始托管 Node.js 站点。这部分内容不在本章范围之内，但以下 URL 中的内容会是一个良好的开端：http://howtonode.org/deploying-node-upstart-monit。

11.9 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

11.9.1 测验

1．PaaS 代表什么？

2．本章讨论的 PaaS 提供商中哪些是开源的？

3．什么时候应该使用你自己的托管服务而不使用 PaaS？

11.9.2 答案

1．PaaS 代表服务即平台，意思是为开发人员提供整个托管平台，而开发人员无需涉入系统管理工作。

2．Cloud Foundry 和 Nodester 都是开源项目，如果想更改它们的工作方式，可以叉出（fork）它们的代码。如果发现 bug 或问题，鼓励你在 GitHub 项目（https://github.com/nodester 和 https://github.com/cloudfoundry）上将问题归档。

3．虽然许多大型网站运行于 PaaS 提供商之上，但用户有可能想完全按照自己的方式提供托管环境。如果想获得完整的控制，可创建自己的服务器，但代价就是需要投入一些时间来构建并维护服务器，而且需要拥有相关技能。

11.10 练习

1．对于本章介绍的每一个 PaaS 提供商，请都对你的 Node.js 应用程序做一些小更改并

第 22 章

创建 Node.js 模块

在本章中你将学到：

- 为 Node.js 模块创建一个 package.json 文件；
- 测试和开发自己的模块；
- 给模块添加一个可执行文件；
- 将项目挂到 Travis CI 上；
- 将模块发布到 npm 注册库上。

22.1 为什么创建模块

在第 2 章中，我们学习了 npm。我们可以用它来访问大量来自世界各地的第三方开发人员所创建的模块。随着读者对 Node.js 越来越熟练或者项目越来越复杂，有可能到了某个时候我们会因为如下原因而需要创建自己的模块。

- 为了方便起见并避免一次一次编写同样的代码。
- 提供在 Node.js 核心中不存在的特定功能。这可以是和第三方 API 交互或者使用 WebSockets。在第 14 章中所创建的流 Twitter 客户端中，我们在 nTwitter 中使用了两个这样的模块来与 Twitter API 和 WebSockets 的 Socket.IO 交互。

22.2 流行的 Node.js 模块

为了更好地理解对编写模块的需要，可看一些在 npm 上最为流行的模块。在本书编写的时候，在 npm 注册上排名前 5 的模块如下所示。

- Underscore（https://github.com/documentcloud/underscore）。
- CoffeScript（https://github.com/jashkenas/coffee-script）。
- Request（https://github.com/mikeal/request）。
- Express（https://github.com/visionmedia/express）。
- Optimist（https://github.com/substack/node-optimist）。

这些模块的流行是因为它们解决开发人员在使用 Node.js 或者 JavaScript 编程语言时每天都会遇见的常见问题和场景。最为流行的模块 Underscore 是 JavaScript 语言自身的一把瑞士军刀，在开发人员中很流行。因为它将 JavaScript 编程语言的许多复杂性和独特部分抽象开来。它添加了一些在原生 JavaScript 中不存在的功能。考虑以下在原生 JavaScript 中的一个简单循环：

```
var numbers = [1,2,3];
for (var i = 0; i < numbers.length; i++) {
  console.log(numbers[i])
}
```

Underscore 提供了方便的“each”方法来迭代一个数组。许多开发人员认为使用“each”这个词要比冗长的原生 JavaScript 方法更方便：

```
var numbers = [1,2,3];
_.each(numbers, function(n){
  console.log(n);
});
```

Request 是另外一个给开发人员带来方便的模块，这一次是做 HTTP 请求。通过这一模块，开发人员可以使用一个更为简单、更为直观的接口，而无需使用 Node.js 的底层 HTTP 客户端。这个模块的流行说明了许多开发人员选择使用直观的抽象而不是原生的 Node.js 接口。

像 Express 这样的模块有更大的代码库，它提供一个创建 Web 应用程序的框架；模块也可以像 Optimist 库这样只有一个代码文件。总的来说，模块解决一个问题并且能很好地解决这个问题。这个思想来自 UNIX 操作系统背后的哲学：做一件事情并且把它做好。这种软件开发方法认为，将小的组件粘合在一起要比用一个方法来解决所有问题更容易。Node.js 社区遵循这一哲学，读者会注意到模块总是被创建用来做一件事情并且做好这件事情。实际上，许多模块本身也依赖于其他模块，这也是 npm 中排名前 5 的模块流行的另一个原因——它们自身也被其他模块所使用。

注意：Node.js 模块本身经常依赖于其他模块

与其重新发明功能，许多模块包括其他模块以避免重新发明轮子。npm 注册库中最为流行的模块经常是在其他模块中使用频繁的模块。

By the Way

22.3 package.json 文件

开始创建 Node.js 模块时以 package.json 文件作为切入点会是个好主意。在前面的几章中，

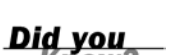
我们使用 package.json 文件来声明创建应用程序所需的依赖模块。作为模块开发人员，我们现在使用 package.json 文件来提供与模块有关的信息，包括名称、描述以及模块的作者。npm 命令行工具提供了一个启动 Node.js 模块和创建 package.json 文件的简单方法。作为开始，我们从终端运行如下命令：

```
npm init
```

这个命令提示用户输入许多内容并创建一个带有将模块与其他人共享所需的最少量信息的 package.json 文件。这些信息包括：

- 包名称——模块的名称；
- 描述——模块的描述；
- 包版本——模块的版本；
- 项目主页——模块的网站，如果有的话；
- 项目的 Git 库——模块的 Git 库，如果有的话；
- 作者姓名——模块的作者；
- 作者电子邮件——模块作者的电子邮件；
- 作者 URL——模块作者的网站；
- 主模块/进入点——模块的主文件；
- 测试命令——运行模块测试的命令；
- Node 版本——该模块支持哪些 Node.js 版本。

在输入这些项目后，package.json 文件会显示给用户，如果用户满意，就会创建这个文件。注意如果要将模块发布到 npm 注册库，那么包的名称必须是唯一的，所以要在选择名称之前搜索一下注册库。



提示：我们可以将选项设置保存在~/.npmrc

我们可以将许多 npm 的选项设置保存在用户主目录的.npmrc 文件中，包括作者名称和主页等。可以设置的选项的完整列表请见 http://npmjs.org/doc/config.html。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour22/example01 找到。按照下列步骤创建 package.json 文件。

1．打开终端并输入如下命令：

```
npm init
```

2．将包名称设为 ohaithere。

3．输入所有能输入的信息。

4．检查添加的信息并回应说你乐于创建文件。

5．在文本编辑器中打开 package.json 文件来检查所添加的信息。可看到系统用你所提交的信息创建了 package.json 文件。

提示：Node.js 模块是 CommonJS 模块

Did you Know?

CommonJS 是个事实上的标准，其目标是可在浏览器、服务器和不同的 JavaScript 框架之间共享 JavaScript 模块。其理论是在使用 JavaScript 的地方就能使用 CommonJS 模块。实际上这并不总是管用——尤其如果使用 Node 特定的功能的时候，而这也在 CommonJS 社区中导致了一些不安情绪。读者可在 http://wiki.commonjs.org/wiki/Modules/1.1 中阅读更多与 CommonJS 有关的信息。

22.4 文件夹结构

Node.js 模块的文件夹结构没有强制要求，但许多开发人员使用常见模式，读者也可如法炮制。如果选择使用下面建议的文件夹结构的话，就使用与你的项目相关的文件夹。于是，如果没有任何文档的话，就无需使用 doc 文件夹！按照项目复杂性的不同，也可能需要添加额外的文件夹或者文件。对于一个非常小的库而言，可能根本不使用任何文件夹并将模块的代码放在单个 index.js 文件中。选择最好的方法由你决定，但要记得如果要和其他开发人员一起工作的话，遵循某种惯例可让他们更容易与你一起开发。以下是建议的文件夹结构。

- .gitignore——从 Git 库中忽略的文件清单。
- .npmignore——不包括在 npm 注册库中的文件清单。
- LICENSE——模块的授权文件。
- README.md——以 Markdown 格式编写的模块 README 文件。
- bin——保存模块可执行文件的文件夹。
- doc——保存模块文档的文件夹。
- examples——保存如何使用模块的实际示例的文件夹。
- lib——保存模块代码的文件夹。
- man——保存模块的任何手册页的文件夹。
- package.json——描述模块的 JSON 文件。
- src——保存源文件的文件夹，经常用于 CoffeeScript 文件。
- test——保存模块测试的文件夹。

如果读者下载了本书的代码示例，那么文件夹结构可在 hour22/example02 中看到。

提示：.npmignore 和.gitignore 是互补的

Did you Know?

如果不想将文件放在 npm 注册库，就将它们加入到.npmignore 文件中。如果模块没有.npmignore 文件但有.gitignore 文件，则 npm 使用.gitignore 文件的内容来忽略注册库中的内容。如果想将一些文件排除在 Git 之外而不是 npm 注册库之外，那么要同时使用.gitignore 和.npmignore 文件。

22.5 开发和测试模块

既然模块已经设置完成，就可以开始开发它了。为了协助开发，npm 带有一个工具，将正在开发的模块全局地安装在计算机上。如果创建了 package.json 文件，可以从同一个目录运行 npm link 命令，该模块就会全局地安装在计算机上。注意模块名称会采自在 pacakge.json 文件中所给出的名称：

```
npm link
/usr/local/lib/node_modules/ohaithere ->
➥/Users/george/code/nodejsbook.io.examples/hour22/example03
```

这会为我们的模块在计算机上创建一个全局的链接，我们现在可以从终端启动 Node 并且如同我们已经在系统上安装的其他模块那样请求这个模块。在本章中我们要开发的是一个简单的模块。它只有一个 hello()函数，返回一个说 Hello from the ohaithere module 的字符串。

既然有了模块的链接，就可以按下列步骤创建模块了。

1．如果遵循建议的惯例，请在 lib 文件夹中添加一个新文件并将该文件命名为与模块一样的名称。在本例中，在 lib 文件夹中要添加一个名为 ohaithere.js 的文件。

2．一旦添加了新文件，就可以修改 package.json 文件，指出模块的进入点：

```
"main" : "./lib/ohaithere.js"
```

3．按照我们在第 10 章中所学的测试驱动开发（TDD）方法，我们首先要给这一功能写一个测试，然后编写代码让测试通过。需要创建一个名为 test 的新文件夹来保存模块的测试。对于本测试而言，我们想测试 hello()函数是否返回某个字符串。使用 Node.js 的原生 assert 模块，测试如下：

```
var assert = require ('assert'),
    ohaithere = require('../lib/ohaithere.js');

/**
 * Test that hello() returns the correct string
 */
assert.equal(
  ohaithere.hello(),
  'Hello from the ohaithere module',
  'Expected "Hello from the ohaithere module". Got "' + ohaithere.hello() +
➥'"'
)
```

4．复制本示例并将其以 ohaithere.js 为名添加到 test 文件夹中。注意在 assert 模块之后，我们正在开发的模块主文件也包括了进来。

5．package.json 文件可以注册如何在应用程序上运行测试。这样就可以通过 npm test 命令来运行测试。为了注册如何为模块运行测试，要在 package.json 文件中添加如下内容：

```
"scripts": {
  "test": "node ./test/ohaithere.js"
}
```

6．这些完成后，就可以使用 npm test 来运行测试：

```
npm test

> ohaithere@0.0.0 test /Users/george/code/nodejsbook.io.examples/hour22/
➥example03
> node ./test/ohaithere.js

node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
              ^
TypeError: Object #<Object> has no method 'hello'
```

7．我们看到测试失败了，因为我们还没有完成 hello 方法。要完成这件事，在 lib/ohaithere.js 文件中添加如下内容：

```
exports.hello = function() {
  var message = "Hello from the ohaithere module";
  return message;
};
```

8．如果再次运行这个测试，可看到测试通过！

注意在这里用了“exports”这个词。这是用于将函数暴露给模块外部或者模块的公共作用域，让任何想使用模块的人都可访问它。如果要将一系列的函数公开给模块的用户，就得用这一模式。如果有只想在模块里面使用的私有函数，那么就将它们声明为普通函数即可。它们不可从模块外部访问。如果读者以面向对象或者基于原型（prototype-based）的风格编程，则将模块的一部分做成公共的方式稍有不同。我们很快就会讲解。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour22/example03 找到。以下步骤演示如何使用测试驱动的方法开发一个模块。

1．在示例 1 创建 package.json 文件的基础上，在 package.json 文件的同一个目录下创建 lib 和 test 文件夹。

2．在 lib 文件夹中，创建一个名为 ohaithere.js 的空白文件。

3．在 package.json 文件中添加一行主声明，如下所示：

```
"main" : "./lib/ohaithere.js"
```

4．在 test 文件夹中，创建一个名为 ohaithere.js 的文件并添加如下内容：

```
var assert = require ('assert'),
    ohaithere = require('../lib/ohaithere.js');

/**
 * Test that hello() returns the correct string
 */
assert.equal(
  ohaithere.hello(),
  'Hello from the ohaithere module',
  'Expected "Hello from the ohaithere module". Got "' + ohaithere.hello() +
➥'"'
)
```

5．将如下内容添加到 package.json 文件中以便让 npm 知道在哪儿找测试：

```
"scripts": {
  "test": "node ./test/ohaithere.js"
}
```

6．在终端上通过运行如下命令从模块的根目录下运行测试。应该看到测试失败：

```
npm test
```

7．打开 lib/ohaithere.js 文件并添加如下代码：

```
exports.hello = function() {
  var message = "Hello from the ohaithere module";
  return message;
};
```

8．再次运行测试。可看到测试通过。

在 hour22/example04 中还有一个使用 Mocha 来测试模块的示例。

22.6 添加可执行文件

可执行文件是可以从终端直接运行的命令。比如，npm 命令是可执行文件。如果遵循建议的惯例的话，可执行文件是添加在模块的 bin 文件夹中的。在本例中，创建了一个名为 ohaithere.js 的文件来调用 hello()函数并将输出记录到控制台上：

```
#!/usr/bin/env node
var ohaithere = require("../lib/ohaithere");
console.log (ohaithere.hello());
```

第一行称为 shebang。它告诉操作系统如何运行这一文件（在本例中，使用 Node.js）。在添加了这个文件之后就可以更新 package.json 文件来声明模块中有一个可执行文件并且指出它所在的位置：

```
"bin" : { "ohaithere" : "./bin/ohaithere.js" }
```

这告诉 npm 有一个 ohaithere 可执行文件，它可以在./bin/ohaithere.js 找到。如果本可执行文件与模块同名，则语法可以更短，因为可以省略命令的名称：

```
"bin": "./bin/ohaithere.js"
```

在添加了这些文件之后，必须再次运行 npm link 将新的可执行文件链接到系统中：

```
npm link

/usr/local/bin/ohaithere -> /usr/local/lib/node_modules/ohaithere/bin/ohaithere.js
/usr/local/lib/node_modules/ohaithere -> /Users/george/code/nodejsbook.io.examples/
➥hour22/example05
```

现在，可以在系统的任何位置运行 ohaithere 命令并在终端上看到输出了！

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour22/example05 找到。按下列步骤给模块添加可执行文件。

1. 在示例 3 的基础上，给模块添加一个名为 bin 的文件夹。
2. 在 bin 文件夹中，创建一个名为 ohaithere.js 的文件并添加如下内容：

```
#!/usr/bin/env node
var ohaithere = require("../lib/ohaithere");
console.log (ohaithere.hello());
```

3. 修改 package.json 文件，加入可执行文件的声明：

```
"bin": "./bin/ohaithere.js"
```

4. 从模块的根目录运行 npm link 命令将可执行文件链接到系统中：

```
npm link
```

5. 从终端运行 ohaithere 命令：

```
ohaithere
```

6. 可看到“Hello from the ohaithere module.”

22.7 使用面向对象或者基于原型的编程

许多开发人员喜欢使用面向对象或者基于原型的编程风格，以便更好地组织代码并管理变量和方法的作用域。ohaithere 模块目前只有一个通过使用 exports 成为公共的方法：

```
exports.hello = function() {
  var message = "Hello from the ohaithere module";
  return message;
};
```

如果以基于原型的风格来编写这段代码，则它应该成为对象内的一个方法：

```
module.exports = new Ohaithere;

function Ohaithere(){}

Ohaithere.prototype.hello = function(){
  var message = "Hello from the ohaithere module";
  return message;
};
```

如果使用面向对象的编程风格，必须使用 module.exports 而不是 exports。exports 实际所做的是收集属性并将它们附加到 module.exports 上，如果 module.exports 还没有任何东西附加在上面的话。不过，如果我们创建自己的对象来组织代码和代码的作用域的话，那么就应该直接附加到 module.exports 上。注意如果这样的话我们就要负责定义对象中方法的作用域。

由于 JavaScript 是灵活的语言，读者可在 Node.js 社区中看到许多不同的编程风格。有些开发人员，比如 marak（Marak Squires）和在 Nodejitsu 工作的开发人员，只使用 exports 和函数。其他开发人员，比如 mikeal（Mikeal Rogers），使用基于原型的风格和 module.exports。此外，有些开发人员像 substack（James Halliday）那样使用 this 关键字在对象中创建特权方法（privileged method）：

```
module.exports = new Ohaithere;

function Ohaithere(){
```

```
  this.hello = function(){
    var message = "Hello from the ohaithere module";
    return message;
  };
}
```

这些技术通常可以并行使用。这里的规则是，如果使用 JavaScript 编程中的面向对象风格的话，就必须使用 module.exports 来导出对象并使用编程技巧来让方法私有或公开。如果只是用函数风格的编程，就可使用 exports 来让一个方法是公开的，或者不使用 exports 来声明一个函数来让其是私有的。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour22/example06 找到。还有一个使用特权方法的示例可在 hour22/example07 找到。以下步骤演示如何以基于模型风格编写模块。

1．在示例 5 的基础上，打开 lib/ohaithere.js 文件并更改其内容以便使用基于原型的编程风格。

```
module.exports = new Ohaithere;

function Ohaithere(){}

Ohaithere.prototype.hello = function(){
  var message = "Hello from the ohaithere module";
  return message;
};
```

2．从模块的根目录运行 test tests。可看到测试通过。

22.8 通过 GitHub 共享代码

在 Node.js 社区里，大多数开发人员以开源软件发布模块并且频繁地使用 GitHub 来协作。GitHub 是个围绕着 Git 创建的用于源代码协作的 Web 应用程序。Git 是一个分布式版本控制系统，Node.js 社区选择了它，于是就通过 GitHub 在协作中大量地使用。如果读者乐于对代码开源，应强烈考虑使用 Git 和 GitHub 来共享源代码。GitHub 对开源项目是免费的，并且提供许多工具来帮助我们管理项目，包括一个问题记录器和一个 wiki。如果刚接触 Git 或 GitHub，在 http://help.github.com/ 有一份全面的指南来指导用户开始。如果选择使用 GitHub，可在 package.json 文件中添加进一步的信息。首先，如果在 GitHub 上储存源代码的话，可以让 npm 知道库在哪儿：

```
"repository": {
  "type": "git",
  "url": "https://github.com/yourusername/yourproject.git"
}
```

如果使用 GitHub 来记录 bug 和问题的话，也可在 package.json 中对此指定。许多更大一些的项目都有邮件列表，这也可包括在 bug 节中：

```
"bugs": {
  "email": "yourproject@googlegroups.com",
  "url": "http://github.com/shapeshed/ohaithere/issues"
}
```

读者可在 http://github.com/shapeshed/ohaithere 看到本例的 GitHub repository（见图 22.1）。

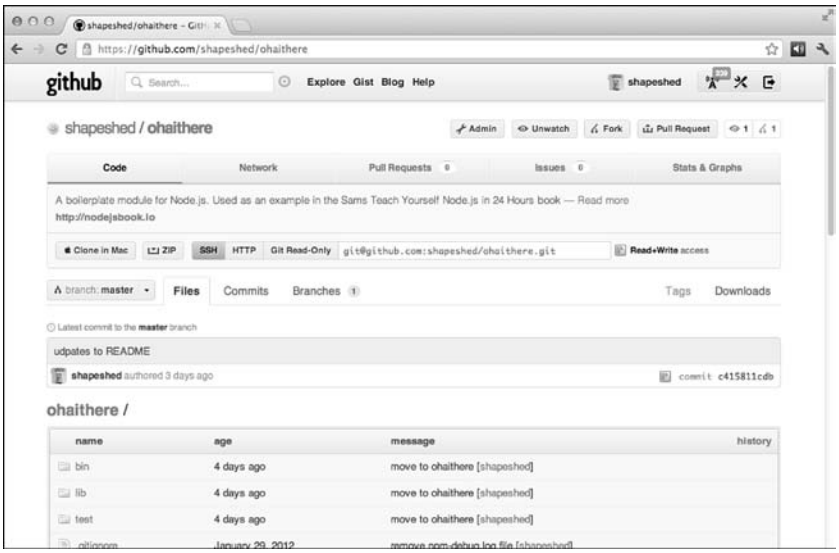


图 22.1
将模块发布到 GitHub

22.9 使用 Travis CI

Travis CI（http://travis-ci.org/）是 Node.js 社区中的一个流行工具。这是一个免费的基于云的分布式持续集成（Continuous Integration）服务器。在这一上下文中，持续集成的意思是每次当代码库有变化的时候就会运行测试。于是，每次将文件推送到 GitHub repository 的时候，Travis CI 就会运行测试并报告是否有问题存在。这可以增加代码的稳定性，因为如果测试出任何问题的话，我们就一定会知道。

为了使用 Travis CI，需要如下东西。

- GitHub 账户（可免费注册：http://github.com）。
- 源代码必须位于 GitHub 库中。

要开始使用 Travis CI，必须使用你的 GitHub 细节信息注册 Travis CI 站点（见图 22.2）。一旦完成注册，请点击你的个人资料页右上角上的链接并按提示进行。

图 22.2
登录 Travis CI

用户可看到自己的 GitHub 库清单（见图 22.3）。为了和 Travis CI 一起使用库，必须首选打开它的开关。

在另外一个浏览器中，打开项目的 GitHub 库并单击 Admin 按钮。接下来，单击左边的 Service Hooks 链接并照提示进行。滚动页面并在清单中找到 Travis（见图 22.4）。单击 Travis 并滚动到顶部。我们会看到需要输入如下的一些信息。

图 22.3

启用库以便 Travis CI 使用

- Domain——输入 travis-ci.org。
- User——输入你的 GitHub 用户名。
- Token——输入来自 Travis CI Profile 的令牌。
- Active——勾选本复选框。

图 22.4

将 Travis CI 添加到 GitHub 库

最后一个步骤是在项目内创建一个文件告诉 Travis CI 要测试的是什么以及如何在项目中运行测试。这是一个声明了应当使用 Node.js 版本 0.4、0.6 和 0.7 测试代码的.yml（Yet Another Markup Language）文件：

```
language: node_js
node_js:
  - 0.4
  - 0.6
  - 0.7
```

将这个文件以.travis.yml 为名保存到项目的根目录下。现在当我们将更新推送到 GitHub 的时候，测试就会在 Travis CI 上运行。通过添加如下程序片段，可将持续集成测试的状态展现在 README 页面上。这是 Travis CI 的一个优秀的功能。

```
 [![Build Status](https://secure.travis-
➥ci.org/yourgithubuser/yourproject.png)](http://travis-ci.org/yourgithubuser/
➥yourproject)
```

如果在推送到 GitHub 之后立即访问 http://travis-ci.org，可以看到测试正被运行（见

图 22.5）。

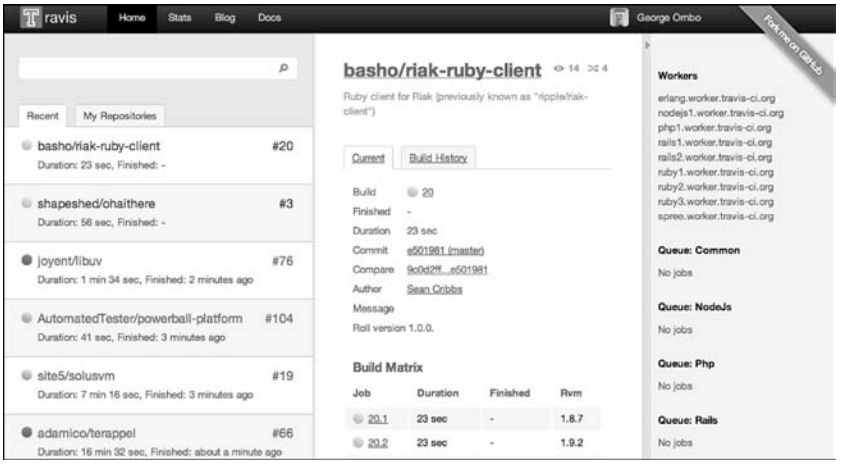


图 22.5

正在 Travis CI 上运行的测试

22.10 发布到 npm

到现在，我们已经对模块做了许多工作了！比如：

- 创建了 package.json 文件；
- 创建了对模块应该如何工作的测试；
- 添加了代码让模块按我们所期望的工作；
- 给模块添加了可执行文件；
- 将项目添加到 GitHub；
- 将项目挂到 Travis CI 上以便进行持续集成测试。

最后要做的事情就是将模块发布到 npm registry。其工作方式是这样的：代码会被压缩到一个 tar 包中然后发送到 npm 注册库服务器保存，其他开发人员可以安装和使用（见图 22.6）。为了发布到注册库必须得有一个账户。要想创建账户，请运行如下命令：

```
npm adduser
```

系统会提示用户输入用户名、密码和电子邮件地址。一旦验证成功，就可以发布模块了。从模块的根文件夹运行以下命令：

```
npm publish
```

这会将模块的一个副本发布到注册库服务器。如果一切成功，就不会看到任何输出，但可以检查 npm 注册库网站来看是否成功发布。如果想在将来发布更新，可修改代码然后运行 npm version 命令后跟新版本号。于是，如果当前版本是 0.0.1，可以使用如下命令发布一个小版本更新：

```
npm version 0.0.2
```

这会递增 package.json 文件中的版本号，并且如果使用 Git 库的话，也会为新的版本号创建一条提交消息。准备好的时候，从模块的根目录运行如下命令：

```
npm publish
```

图 22.6
将模块发布到 npm 注册库

22.11 公开模块

我们花费了大量时间炮制自己的模块，现在该是让人们知道它的时候了！可以在如下的许多地方公开模块。

- 通过 Node.js Google Groups 邮件列表，地址是 http://groups.google.com/group/nodejs。
- 通过 irc.freenode.net 的#node.js 通道上的 IRC。
- 通过使用#nodejs 井号标签在 Twitter 上发布。

22.12 小结

在本章，我们学习了如何创建 Node.js 模块。我们学习了如何创建 pakcage.json 文件，然后探索了开发和测试模块的方法。我们给模块添加了一个可执行文件然后了解了与其他开发人员共享代码的方法。我们看到了将 Travis CI 与项目集成的方法以及最终将模块发布到 npm 注册库的方法。

22.13 问与答

问：我刚刚开始使用 Node.js。我是否必须考虑创建模块的事情？

答：如果在开发中一次一次遇到相同的问题，首先要做的是检查一下 npm 注册库。可能已经有其他开发人员创建了一个模块。如果没有模块可用，而且你觉得自己编码技艺良好，那么当然可以创建模块！这是给社区做出贡献的绝佳方式！

问：实现抽象的模块是否减低性能？

答：抽象在让功能更容易实现的同时，经常需要以性能为代价。在本章早些时候的 underscore 示例中，我们看到 underscore 的 each 方法如何替代 JavaScript 更为冗长的 for。在

本例中，underscore 的性能比起原生 JavaScript 要差一些。读者可以在这里运行测试来展现这个问题：http://jsperf.com/jquery-each-vs-for-loop/58。除非性能至上，否则这点区别很可能是不重要的。不过要记得，创建我们自己的、替代原生代码的接口会对性能造成影响。

问：如果模块已经存在但不完全能实现我想要的功能，我是否应该创建自己的模块？

答：有时候，我们会发现有一个模块已经存在但不完全匹配需求。在这样的场景里，可考虑在现有模块中能够添加需求。如果源代码在 GitHub 中，就可以叉出（fork）现有的模块，添加自己的代码，然后提交（pull request）一份给模块作者。在创建整个新模块之前，最起码要考虑联系模块作者。

问：我是否必须对模块开源？

答：不是。虽然大多数开发人员以开源授权方式发布模块，但这不是必须的。可以只为自己或者一起工作的团队创建模块。

22.14 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

22.14.1 问题

1．如何搜索现有的 npm 模块？

2．在开发一个模块的时候，用来将模块全局地安装在计算机上的命令是什么？

3．使用 npm 的时候如何取得帮助？

22.14.2 答案

1．可通过在命令行上使用 npm search 或者通过使用 http://search.npmjs.org/上的 Web 界面来搜索现有 npm 模块。

2．这个命令是 npm link，它在计算机上全局安装我们自己的模块。如果想反链接（unlink）模块，可运行 npm unlink 命令。

3．可通过运行 man npm 来寻找 npm 的帮助。对于各个命令，可运行 npm help link 来获得想要运行的命令的全面信息。

22.15 练习

1．给模块添加另一个名为“goodbye”的方法。它应打印说再见的消息。

2．在 npm 注册库中浏览一些流行模块的源代码。尝试理解其编码风格并指出使用 exports 和 module.exports 的地方。

3．将你的模块以项目发布在 GitHub 上。

4．将你的项目挂接到 Travis CI 上。

Node.js 入门经典

SAMS

Teach Yourself

学习如何：

- 完全使用JavaScript创建端到端的应用；
- 掌握Node.js的基本概念（如回调），并快速创建第一个程序；
- 使用HTTP模块和Express Web框架创建基本的站点；
- 使用Node.js和MongoDB管理数据的持久化；
- 调试和测试Node.js应用；
- 将Node.js应用部署到第三方服务，比如Heroku和Nodester；
- 构建强大的实时解决方案，从聊天服务器到Twitter客户端；
- 使用JavaScript在服务器上创建JSON API；
- 使用Node.js API的核心组件，包括进程、子进程、事件、缓冲区和流；
- 创建并发布Node.js模块。

24章阶梯教学

通过阅读本书，读者将能掌握Node.js平台，并学会使用它来创建具有非凡速度和可扩展性的服务器端应用。本书采用直观、循序渐进的方法，引导读者掌握Node.js的基本安装、配置，并通过浏览器和服务器之间的实时通信来掌握Node.js的编程、测试和部署方法。本书每章内容都建立在已学的知识之上，读者可通过本书打下坚实的基础，为走向成功铺平道路。

循序渐进的示例引导读者完成最常见的Node.js开发任务。

问与答、测验和练习帮助读者检验知识的掌握情况。

“注意”、“提示”和“警告”指出捷径和解决方案。

读者可以通过www.ptpress.com.cn或http://vdisk.weibo.com/s/s7fmW下载本书的所有源代码。

ISBN 978-7-115-31107-8

美术编辑 王建国

分类建议：计算机/程序设计/Web开发

人民邮电出版社网址：www.ptpress.com.cn