Learning Node

Node 学习指南

[美] *Shelley Powers* 著
夏思雨 高亮 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

Node学习指南

本书帮助你将Web开发技能从浏览器端转向Node服务器，并且学习如何使用Node这种基于JavaScript的平台编写出快速和高可扩展性的网络应用。在本书的指导下，你可以快速掌握Node的核心技能，获得使用内建和扩展模块的经验，并了解客户端编程和服务器端编程的不同和相同之处。

为了加快Node事件驱动的速度，在开发计算简单但是访问频繁的数据密集型程序时，可以使用异步的I/O模型。

如果你喜欢使用JavaScript，本书提供了很多代码和开发的示例来帮助你学习Node服务器端的开发。

- 探索Node独特的异步开发的实现方式；
- 使用Express架构和Connect中间件构建Node应用示例；
- 使用NoSQL解决方案，比如Redis和MongoDB，探索Node的关系数据库模块；
- 使用PDF文件，提供HTML5媒体，使用Canvas创建图形；
- 使用WebSockets创建浏览器和服务器的双向通信；
- 深入学习如何调试和测试程序；
- 在云服务器或者自己的系统上部署Node应用程序。

Shelley Powers从JavaScript刚发布时，就开始使用和编写Web技术相关书籍。她之前在O'Reilly出版了8本书，包括Developing ASP Components（2001），Adding Ajax（2007）和JavaScript Cookbook（2010）。

"一本非常好的介绍Node.js的书。我一直把它放在触手可及的地方。"

——Mike Amundsen，*Building Hypermedia APIs with HTML5 and Node*一书作者

O'REILLY®
oreilly.com.cn

封面设计：Karen Montgomery，张健

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

分类建议：计算机/程序设计/JavaScript
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33796-2
定价：69.00 元

Node 学习指南

[美] Shelley Powers　著

夏思雨　高亮　译

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

Node学习指南 / (美) 鲍尔丝 (Powers, S.) 著 ; 夏思雨，高亮译. -- 北京 : 人民邮电出版社，2014.3
ISBN 978-7-115-33796-2

Ⅰ. ①N… Ⅱ. ①鲍… ②夏… ③高… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第277709号

版权声明

Copyright© 2012 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2014. Authorized translation of the English edition, 2014 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版由 O' Reilly Media, Inc.授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

◆ 著　　[美] Shelley Powers
　译　　夏思雨　高　亮
　责任编辑　陈冀康
　责任印制　程彦红　杨林杰

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

◆ 开本：787×1000　1/16
　印张：23.75
　字数：451 千字　　2014 年 3 月第 1 版
　印数：1 – 3 000 册　　2014 年 3 月河北第 1 次印刷

著作权合同登记号　图字：01-2013-3678 号

定价：69.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 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。它可以让 JavaScript 在服务器端运行，因此，可用来快速构建网络服务及应用的平台。

本书是学习 Node 编程的入门指南。全书共 16 章。前 4 章主要介绍 Node 基本知识，包管理工具（npm）的安装和使用等。第 5 章介绍了 Node 处理异步开发的独特的实现方式等。第 6~8 章，讲解了路由、代理、Web 服务器、中间件等基本概念，包括 Express。第 9 章到第 11 章分别介绍了基于 Redis、MongoDB 以及关系型数据库的 Node 应用开发。第 12 章到第 14 章分别介绍了图形和媒体、Sockets.io 模块、调试和测试等主题。第 15 章介绍了安全和权限的问题，第 16 章介绍了 Node 应用的扩展和部署。

本书适合有一定基础的 JavaScript 程序员阅读，也适合对学习 Node 应用开发感兴趣的读者学习参考。

前言

非同寻常的 JavaScript

目前正是学习 Node 的好时机。

Node 相关的技术依然年轻充满生机，经常出现有趣的变化和改动。同时，这项技术也达到了一定的成熟度，可以确保你在学习 Node 上花费的时间是值得的：即使在 Windows 上安装也非常简单；从成百上千的可用模块中涌现出了最佳组合模块；对于产品环境来说这种结构足够健壮。

当使用 Node 时需要记得两个要点。第一，Node 是基于 JavaScript 的，与你之前用于客户端开发的 JavaScript 多少有些类似。当然，你也可以使用另一种变形的语言，如 CoffeeScript，但是 JavaScript 是通用的语言。

第二个需要注意的要点是，Node 并不是常规的 JavaScript。它是一门服务器端的技术，这意味着很多你在浏览器环境中认为应该有的功能——如保护措施——都不会出现在这里，但也会有很多其他新的不熟悉的功能。

当然，如果 Node 和浏览器端的 JavaScript 一样的话，那有什么乐趣呢？

为什么是 Node

如果你想要看 Node 的源码，你可以找一下 Google V8 的源代码。Google V8 是 JavaScript 引擎（从技术角度来讲，是 ECMAScript），也是 Google Chrome 浏览器的核心。那么，Node.js 的一个优点是你可以只为一种 JavaScript 实现开发 Node 程序，而不是一大堆不同版本的不同浏览器。

Node 被设计用于那些需要频繁 I/O 操作，但计算量不大的程序。更重要的是，它提供的这个功能是直接可用的。在等待一个文件加载完成或者数据库更新的过程中，不需要担心程序会阻塞其他进程，因为 Node 中大部分功能默认都是 I/O 异步的；也不需要担心线程的工作，因为 Node 的实现是单线程的。

异步 I/O 意味着程序并不会等待输入/输出操作处理完成之后才处理代码中的下一个步骤。第 1 章会介绍更多 Node 异步特性的细节。

更重要的一点是，Node 是由很多传统 Web 开发人员都熟悉的语言 JavaScript 编写的。你会学习到如何使用新的技术，如 WebSocket 或者基于 Express 这种框架进行开发，但是至少你不需要在学习新概念的同时学习一门新的语言。对语言的熟悉使你可以只专注新的特性。

本书的目标读者

使用 Node 的一个挑战就是假设学习 Node 的部分人有 Ruby 或者 Python 背景，或者使用过 Redis。我没有假定这一点，所以在解释 Node 组件时我不会说这就“像 Sinatra 一样”。

这本书唯一的假设就是读者使用过 JavaScript 并且喜欢它。你并不需要是个专家，但是你需要在我提到“闭包”的时候知道我在说什么，并且使用过 Ajax 以及对客户端环境的事件处理比较熟悉。如果你做过传统的 Web 开发，熟悉一些概念，如 HTTP 方法（GET、POST）、Web session、cookie 等，你会从本书中获益良多。除了这些，你需要熟悉 Windows 控制台，或者 Unix、Linux、MAC OSX 的命令行。

如果你对一些新技术感兴趣，诸如 WebSocket 或者使用架构创建程序，就会喜欢这本书的。我通过这些方面向你介绍如何在现实世界使用 Node。

最重要的一点，在你阅读本书时要保持思维开放，要有思想准备、你可能碰上版本不成熟的问题，也可能会撞上这种处在发展中的技术陷阱。无论如何，带着期待开始你的学习旅程吧，因为这是一个很有趣的过程。

如果你不确定你达到了“熟悉”JavaScript 的标准，可以查看一下我对 JavaScript 的介绍：*Learning JavaScrpt*，第二版（O’ Reilly）。

怎么更好地使用本书

如果你不想按顺序阅读本书，有一些途径可供选择，这取决于你想要知道些什么以及你有多少 Node 经验。

如果你从来没有用过 Node，建议你从第 1 章开始阅读至少到第 5 章。这几章主要讲了 Node 基本知识、包管理工具（npm）安装、如何使用、创建你的第一个程序

以及可用的模块。第 5 章还涉及了一些 Node 中的样式问题，包括 Node 处理异步开发的独特实现方式。

如果你有一点 Node 经验，使用过内建和第三方的一些模块，也熟悉 REPL（read-eval-print loop，交互控制台），可以跳过第 1 章到第 4 章，但是我建议第 5 章还是必读的。

在本书中，我使用了 Express 架构，Express 又使用了 Connect 中间件。如果你没用过 Express，请阅读第 6～第 8 章，这几章讲解了路由、代理、Web 服务器、中间件等基本概念，包括 Express。尤其是如果你好奇在 MVC（Model-View-Controller）框架中如何使用 Express，一定要阅读第 7 章和第 8 章。

在这些基础章节之后，你可以有选择地进行阅读。比如，如果你主要使用 key/value（键值对），你应该阅读第 9 章关于 Redis 的讨论。如果你对基于文档的数据感兴趣，查阅第 10 章，这一章介绍了如何在 Node 中使用 MongoDB。当然，如果你只需要使用关系型数据库，可以直接跳过 Redis 和 MongoDB 到第 11 章，记得经常检查数据库的更新——它们可能提供了一种处理数据的新视角。

在这三章关于数据的讲解之后，我们会介绍具体程序的使用。第 12 章主要关注于图形和媒体访问，包括如何提供 HTML5 视频媒体，以及使用 Canvas 和 PDF 文件。第 13 章主要讲了非常流行的 Sockets.io 模块，特别是如何使用新的 web socket 功能。

在第 12 章和第 13 章两章关于 Node 两种不同的具体用法介绍之后，本书末尾再次回到同一点上。当你在其他章节花费了一些时间尝试示例之后，需要读一下第 14 章，深入了解如何调试和测试 Node 程序。

第 15 章可能是最难的一章，也更重要。这一章主要介绍了安全和权限的问题。我不建议把这章作为你阅读的第一章，但是在你发布一个 Node 应用之前需要花点时间读一下这一章节。

第 16 章是最后一章，不管你兴趣是什么、经验多少，你都可以很放心地把它留到最后。这章主要介绍如何使你的产品上线，包括如何在流行的云服务上或者你自己的系统中部署 Node 程序，如何确保你的程序可以与其他 Web 服务器兼容，比如 Apache，如何确保你的程序在崩溃或者系统重启之后自动重启。

Node 与 Git 版本控制联系紧密，并且绝大部分（如果不是全部的话）的 Node 模块都在 GitHub 上。附录里为那些不了解 Git/GitHub 的人提供了教程。

虽然我之前说不需要按章节进行阅读，但是我建议你还是按顺序阅读。很多章节的

工作都是在前一章的基础上完成，如果你跳跃阅读的话可能会错过一些重点。尽管本书有很多独立的程序示例，但是我主要使用了一个简单的 Express 程序——叫做 Widget Factory，从第 7 章开始，在剩下的章节中都有涉及。我相信如果你从开始阅读，跳过那些你了解的小节会比跳过整个章节好很多。

就像《爱丽丝漫游奇境记》里的国王说的一样：从“开头开始，一直念到末尾，然后停止。”

技术

本书中的示例是在 Node 0.6.x 不同的发布版本中创建的，大部分都在 Linux 环境中测试过，应该在各种 Node 环境中都可以正常工作。

Node 0.8.x 发布的时候本书刚刚上市。大部分示例都兼容 Node 0.8.x。有一些例子需要做一些修改以兼容新版 Node，我在书中也有指出。

示例

本书中的所有示例在 O'Reilly 官网本书主页上（http://oreil.ly/Learning_node）有压缩文件可以下载。下载解压，安装好 Node 之后，可以切换 examples 目录，输入以下命令来安装示例需要的所有依赖：

npm install–d

在第 4 章中我会讲到更多关于 npm 的用法。

本书中的体例

以下是本书中使用的印刷体例：

文本

菜单标题，菜单选项，菜单按钮，键盘快捷键（比如 Alt 和 Ctrl）。

等宽字体

命令，选项，switch，变量，属性，key，函数，类型，类，命名空间，方法，模块，参数，值，对象，事件，事件处理，XML 标记，HTML 标记，宏，文件内容或者命令行输出。

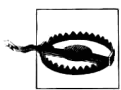
等宽粗体

显示命令或者其他应该由用户输入的文本。

等宽斜体

显示应该由用户提供的内容替换的文本。

这个图标表示提示、建议或者一般的记录。



这个图标表示一个警告或者提醒。

使用代码示例

本书有助于你完成工作。一般来说，可以在程序或者文档中使用本书的代码。关于权限的问题你并不需要联系我们，除非大量复制代码。比如，使用本书中的几段代码编写一段程序并不需要我们的允许，但是如果使用 O' Reilly 书中的代码买卖或者制成 CD，则需要我们允许才可以。将本书中的大量代码作为你的产品文档也需要权限。

我们非常感激你在引用时标明出处，但是并不强制。引用包括标题、作者、出版商、ISBN。比如："Learning Node by Shelley Powers (O'Reilly). Copyright 2012 Shelley Powers, 978-1-449-32307-3"。

如果你觉得你的使用不在上述范围之内，请联系我们 permissions@oreilly.com。

Safari Bookd Online

Safari Books Online（www.safaribooksonline.com）是一个即时的电子图书馆，传递全世界在技术和商业领域著名作者的书籍和视频。

技术专家、软件开发人员、网站设计、商业和创意专家都使用 Safari Books Online 作为他们搜索、解决问题，学习和培训的主要资源。

Safari Books Online 为组织、政府代理和个人提供了不同的产品搭配组合和定价。购买者可以访问数以千计的书籍、培训视频，以及从出版商的数据库访问还未发布的手稿，比如 O' Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、MicrosoftPress、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、

FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，等等。更多信息请访问我们的网站。

联系我们

有关本书的提问和评价请联系出版商：

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

我们为本书提供了网页列出勘误表、示例，以及其他附加信息。可以访问：http://oreil.ly/Learning_node。

关于本书的评价或者任何技术问题，请发送邮件到 bookquestions@oreily.com。

更多有关书籍、课程、会议和最新信息，请访问我们的网站：http://www.oreily.com。

Facebook: http://facebook.com/oreilly

Twitter: http://twitter.com/oreillymedia

YouTube: http://www.youtube.com/oreillymedia

感谢

感谢所有的朋友和家人在我写书的时候帮助我保持清醒。特别感谢我的编辑 Simon St. Laurent，经常听我发牢骚。

还要感谢整个产品组帮助这本书从一个想法变成一个实物：Rachel Steely、Rachel Monaghan、Kiel Van Horn、Aaron Hazelton 和 Rebecca Demarest。

当你使用 Node 时，你会接收很多的帮助来自于 Node.js 创始人 Ryan Dahl，npm 创造者 Issac Schlueter，也是现在的 Node.js 掌门人。

其他为本书提供了有用的代码和模块的是：Bert Belder、TJ Holowaychuk、Jeremy

Ashkenas、Mikeal Rogers、Guillerno Rauch、Jared Hanson、Felix Geisendörfer、Steve Sanderson、Matt Ranney、Caolan McMahon、Remy Sharp、Chris O'Hara、Mariano Iglesias、Marco Aurelio、Damian Suarez、Jeremy Ashkenas、Nathan Rajlich、Christian Amor Kvalheim 和 Gianni Chiappetta。如果有任何漏掉没有提到的模块开发人员，我在此表示歉意。

如果没有那些好心人提供教程、做法以及有用的指导，本书还有些什么呢？感谢 Tim Caswell、Felix Geisendorfer、Mikato Takada、Geo Paul、Manuel Kiessling、Scott Hanselman、Peter Krumins、Tom Hughes-Croucher、Ben Nadel，以及 Nodejistu 和 Joyent 的全体人员。

目录

第 1 章　Node.js：启动与运行……1
1.1　搭建 Node 开发环境……2
1.1.1　Linux（Ubuntu）下安装 Node……2
1.1.2　Windows 7 平台下 Node+WebMatrix……4
1.1.3　升级 Node……8
1.2　开始 Node 开发……9
1.2.1　Hello, World in Node……9
1.2.2　分析“Hello,World”……11
1.3　异步函数及 Node 事件循环……13
1.3.1　使用异步方式读取文件……14
1.3.2　观察异步程序流程……15
1.4　Node 的优势……19
第 2 章　Node 与 REPL……20
2.1　REPL：先睹为快和未定义的表达式……20
2.2　REPL 的优势：更好地理解表层之下的 JavaScript……22
2.3　多行以及更复杂的 JavaScript……23
2.3.1　REPL 命令……26
2.3.2　REPL 和 rlwrap……27
2.3.3　定制 REPL……28
2.4　不可预计的意外——记得经常保存……32
第 3 章　Node 核心库……33
3.1　全局对象：global、process 和 Buffer……34
3.1.1　global……34
3.1.2　process……36
3.1.3　Buffer……38
3.2　定时器：setTimeout、clearTimeout、setInterval 和 clearInterval……39
3.3　Servers、Streams 和 Sockets……40
3.3.1　TCP Sockets 和 Servers……41
3.3.2　HTTP……43
3.3.3　UDP 数据报套接字……45

3.3.4 流、管道和 Readline ……47
3.4 子进程……49
3.4.1 child_process.spawn ……50
3.4.2 child_process.exec 和 child_process.execFile ……52
3.4.3 child_process.fork……52
3.4.4 在 Windows 系统中使用子进程……53
3.5 域名解析和 URL 处理……54
3.6 Utilities 模块和对象继承……55
3.7 Events 和 EventEmitter……59
第 4 章 Node 模块系统……63
4.1 使用 require 和默认路径加载模块……63
4.2 外部模块和 Node 包管理工具……65
4.3 如何找到你需要的模块……69
4.3.1 Colors：简单至上……71
4.3.2 Optimist：另一个简单的小模块……73
4.3.3 Underscore……74
4.4 创建自定义模块……75
4.4.1 打包整个目录……76
4.4.2 为你的模块发布做准备……76
4.4.3 发布模块……80
第 5 章 控制流、异步模式和异常处理……82
5.1 使用 Callback 而不使用 Promises……82
5.2 顺序调用、嵌套回调、异常捕获……85
5.3 异步模式和控制流模块……92
5.3.1 Step……93
5.3.2 Async……96
5.4 Node 编码风格……101
第 6 章 路由寻址、服务文件和中间件……103
6.1 从头开始：创建一个简单的静态文件服务器……103
6.2 中间件……110
6.2.1 Connect 基本知识……111
6.2.2 Connect 中间件……113
6.2.3 定制 Connect 中间件……118
6.3 Routers……121
6.4 Proxies……124

第 7 章　Express 框架 …… 128
7.1　Express：启动和运行 …… 129
7.2　app.js 文件 …… 130
7.3　错误处理 …… 133
7.4　Express 与 Connect 的关系 …… 134
7.5　路由 …… 135
7.5.1　路由路径 …… 137
7.5.2　路由和 HTTP 动词 …… 140
7.6　关于 MVC …… 147
7.7　使用 cURL 测试 Express 应用程序 …… 152
第 8 章　Express、模板系统和 CSS …… 154
8.1　EJS 模板系统（Embedded JavaScript Template System） …… 154
8.1.1　基本语法 …… 155
8.1.2　Node 与 EJS …… 156
8.1.3　EJS 与 Node Filters …… 158
8.2　在 Express 中使用 EJS …… 159
8.2.1　多对象环境的改造 …… 161
8.2.2　静态文件路由 …… 162
8.2.3　处理一个新对象的 Post 请求 …… 164
8.2.4　Widget 索引和生成 picklist …… 166
8.2.5　显示单个对象并确认对象的删除操作 …… 168
8.2.6　提供更新信息的表达以及处理 PUT 请求 …… 170
8.3　Jade 模板系统 …… 173
8.3.1　Jade 语法简介 …… 173
8.3.2　使用 block 和 extends 模块化视图模板 …… 176
8.3.3　Widget View 转换为 Jade 模板 …… 178
8.3.4　转换 edit 和 delete 表单 …… 179
8.4　使用 Stylus 完成简单的 CSS 样式 …… 182
第 9 章　结构化数据、Noe 和 Redis …… 187
9.1　Node 和 Redis …… 188
9.2　构建游戏得分排行榜 …… 190
9.3　创建消息队列 …… 197
9.4　为 Express 应用程序添加统计中间件 …… 201
第 10 章　Node 和 MongoDB：文档中心数据 …… 206
10.1　MongoDB Native Node.js Driver（MongoDB 原生 Node.js 驱动） …… 207

10.1.1 MongoDB 入门 ······207
10.1.2 定义、创建以及销毁 MongoDB Collection ······208
10.1.3 为 Collection 添加数据 ······209
10.1.4 查询数据 ······212
10.1.5 使用 Updates、Upserts、Find 和 Remove ······216
10.2 使用 Mongoose 实现 Widget 模块 ······221
10.3 重构 Widget 工厂 ······222
10.4 添加 MongoDB 后台 ······223
第 11 章 Node 与关系型数据库 ······228
11.1 db-mysql 入门 ······229
11.1.1 查询字符串和方法链 ······229
11.1.2 使用查询字符串更新数据库 ······233
11.1.3 使用方法链更新数据库 ······235
11.2 使用 node-mysql 实现本地 MySQL 访问 ······237
11.2.1 使用 node-mysql 做基本的 CRUD 操作 ······237
11.2.2 MySQL 事务与 mysql-queues ······239
11.3 ORM 与 Sequelize ······241
11.3.1 定义模型 ······241
11.3.2 ORM 风格的 CRUD 实现 ······243
11.3.3 添加多个对象 ······246
11.3.4 从关系型到 ORM ······247
第 12 章 图形和 HTML5 Video ······248
12.1 创建和使用 PDF ······248
12.1.1 使用子进程访问 PDF 工具 ······249
12.1.2 使用 PDFKit 创建 PDF ······257
12.2 从子进程访问 ImageMagick ······258
12.3 通过 HTTP 提供 HTML5 Video 服务 ······263
12.4 创建和流化画布内容（Canvas Content） ······267
第 13 章 WebSockets 和 Socket.IO ······271
13.1 WebSockets ······271
13.2 Socket.IO 简介 ······272
13.2.1 一个简单的通信范例 ······273
13.2.2 异步世界里的 WebSockets ······276
13.2.3 关于客户端代码 ······277
13.3 配置 Socket.IO ······278

13.4 Chat：WebSockets 版本的“Hello, World” ······279
13.5 在 Express 中使用 Socket.IO ······282
第 14 章 Node 应用程序的测试和调试 ······284
14.1 调试 ······284
14.1.1 Node.js Debugger ······284
14.1.2 使用 Node Inspector 的客户端调试 ······287
14.2 单元测试（Unit Testing） ······289
14.2.1 Assert 与单元测试 ······289
14.2.2 Nodeunit 与单元测试 ······293
14.2.3 其他测试框架 ······295
14.3 验收测试 ······299
14.3.1 Soda 和 Selenium 测试 ······299
14.3.2 通过 Tobi 和 Zombie 模拟浏览器 ······303
14.4 性能测试：基准问题和负载测试 ······304
14.4.1 ApacheBench 基准测试 ······305
14.4.2 Nodeload 与负载测试 ······309
14.5 Nodemon 更新代码 ······312
第 15 章 安全及防护 ······313
15.1 数据加密 ······314
15.1.1 TSL / SSL 配置 ······314
15.1.2 使用 HTTPS ······315
15.1.3 如何安全的保存密码 ······317
15.2 认证/授权及 Passport ······320
15.2.1 授权/认证策略：Oauth、OpenID、用户名/密码验证 ······321
15.2.2 Local Passport Strategy ······323
15.2.3 Twitter Passport Strategy（OAuth） ······330
15.3 保护应用程序，防止攻击 ······336
15.3.1 不要使用 eval ······336
15.3.2 尽量使用复选框、单选按钮和下拉式选项 ······337
15.3.3 使用 node-validator ······337
15.4 在沙箱中执行代码 ······339
第 16 章 扩展和部署 Node 应用 ······343
16.1 把你的节点部署到服务器上 ······343
16.1.1 编写 package.json 文件 ······344
16.1.2 使用 Forever 让你的应用“永不掉线” ······347

16.1.3 使用 Node 和 Apache ······350
16.1.4 改善性能 ······352
16.2 部署到云服务 ······352
16.2.1 通过 Cloud9 IDE 部署到 Windows Azure ······353
16.2.2 Joyent Development SmartMachine ······355
16.2.3 Heroku ······355
16.2.4 Amazon EC2 ······356
16.2.5 Nodejitsu ······356
附录 Node、Git 和 GitHub ······357

第 1 章

Node.js：启动与运行

Node.js 是以 Google V8 JavaScript 引擎为基础的服务器端技术。它具有很好的可扩展性，并使用了异步事件驱动 IO，而没有使用线程或者独立进程。它能很好地满足那些需要频繁访问但是计算简单的网络应用的需求。

使用传统的 Web 服务器时，比如 Apache，每次接收到用户对网络资源的请求时，Apache 都会创建一个线程或者调用新的进程来处理。尽管 Apache 对请求的响应速度非常快，并在请求处理完毕后清理现场，但这种实现仍然占用了很多资源。访问频繁的网络应用会因此产生严重的性能问题。

相较而言，Node 不会为每个请求创建新的进程或者线程。相反，它对特定事件进行监听，当事件发生时按需做出响应。在等待事件的过程中 Node 并不阻止任何请求，并且事件循环是按照先到先得的简单方式进行处理。

与编写客户端应用程序所使用的语言一样，Node 应用程序也是用 JaveScript 编写的（或者其他可以编译成 JaveScript 的语言）。不过与前者不同的是，做 Node 应用程序开发前，首先需要搭建开发环境。

Node 支持 Unix/Linux、MaxOS 以及 Windows 等多种操作系统。本章会告诉你在安装 Node 之前需要完成的准备工作，并带领你学习如何在 Windows7 和 Linux（Ubuntu）系统上搭建 Node 开发环境。Mac 系统的安装过程与 Linux 类似。

当搭建好开发环境后，我们会通过一个简单的 Node 示例应用，来说明 Node 中最重要的事件循环机制。

1.1 搭建 Node 开发环境

安装 Node 有多种方法。选择什么样的方法取决于你当前使用的开发环境，以及你希望如何使用源代码或者计划如何在你现有的应用程序中使用 Node。

Windows 和 Mac OS 都有相应的安装包，但是你也可以复制 Node 源代码并自行编译安装。在 Windows、Linux 和 Mac OS 环境中，你也可以使用 Git 的 clone 命令来复制 Node 的 repo（repository，代码库）。

本节演示如何在 Linux 系统中（Ubuntu 10.04 VPS，或者虚拟服务器）通过编译源码搭建 Node 环境，同时也会演示如何在 Windosw7 系统上安装 Node，以便配合 Microsoft WebMatrix 使用。

提示

可以从这个地址下载 Node 源代码及安装包: http://no dejs.org/#download。Wiki 关于多种环境中安装 Node 的说明: https://github.com/joyent/node/iki/Installing-Node-via-package-manager。鉴于 Node 不时会有版本更新，建议读者自己搜索最新的关于特定环境下安装 Node 的教程。

1.1.1 Linux（Ubuntu）下安装 Node

在 Linux 下安装 Node 之前，你需要先做一些准备工作。按照 Wiki Node 词条中提到的步骤首先要确认是否安装 Python，如果计划使用 SSL/TLS（Secure Sockets Layer，安全套接层/Transport Layer Security，传输层安全）还需要安装 libssl-dev。某些 Linux 系统中默认安装了 Python。如果没有，则可以使用系统包安装工具安装 Python 的稳定版本，如 2.6 或者 2.7（Node 最新版本所要求的 Python 版本号）。

提示

本书假定你有 JavaScript 和传统 Web 开发的经验。如果这样的话，我可能是过于谨慎了，并且在描述如何安装 Node 的前期准备上太啰唆了。

对 Ubuntu 和 Debian 来说还需要安装其他的库。大部分 Debian GNU/Linux 系统中都支持 APT（Advanced Packaging Tool）工具，你可以用以下 apt 命令来确认是否已安装了需要的库：

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential openssl libssl-dev pkg-config
```

update 命令用于确认系统上的包都更新了，upgrade 命令用于升级过期的包。第三条命令用于安装需要的包。任何包依赖都由包管理工具管理。

当准备好环境后，下载 Node tarball（源代码的压缩文件）。本书使用 wget，你也可以使用 curl。在编写本书的时候 Node 的最新版本是 0.8.2：

```
wget http://nodejs.org/dist/v0.8.2/node-v0.8.2.tar.gz
```

下载完成后，解压：

```
tar -zxf node-v0.8.2.tar.gz
```

得到目录 node-v0.6.18。进入该目录，使用以下命令编译安装 Node：

```
./configure
make
sudo make install
```

考虑到某些读者之前没在 Unix 中使用过 make 命令，同此，对这三条命令简单说明一下：第一条命令首先进行依赖检查，然后根据你的系统环境和安装情况建立 makefile，然后执行 make 命令编译最后一条命令执行安装操作。在这些命令执行后，Node 即安装完毕并可以通过命令行全局访问。

提示

编程的挑战在于没有两个系统是一样的。在大部分 Linux 环境下这一系列安装步骤应该是能工作且可以成功的。但是，关键词是“应该”。

是否注意到最后一条命令的 sudo？这是为了以 root 权限在 Linux 中安装 Node。不过，你还可以用以下命令在指定的下级目录中安装 Node：

```
mkdir ~/working
./configure --prefix=~/working
make
make install
echo 'export PATH=~/working/bin:${PATH}' >> ~/.bashrc
. ~/.bashrc
```

可以看到，通过设置 prefix 配置选项，你可以将 Node 安装到本地指定路径的目录中。另外别忘记你可能还需要相应地为 PATH 环境变量做些更新。

提示

如果使用 sudo，你需要 root 或者超级用户的权限。此时你的用户名必须存在于/ect/sudoers 文件的列表中。

尽管可以将 Node 安装到本地路径，但是如果考虑到我们所有的安装步骤都是在一个共享

的主机环境下进行的，是否能采用这种安装方式还需三思。因为安装 Node 只是第一步，接下来你还需要权限来编译Node应用程序，需要权限来让程序运行在特定端口(比如80)。事实上绝大多数共享主机环境并不会允许我们在本地路径下随意安装特定版本的 Node。

除非有特殊原因，依然推荐使用 sudo 安装 Node。

提示

在本书第 4 章我们会提到，使用 root 权限运行 Node 包管理工具(npm)曾经存在的一个安全顾虑。但是，目前这些安全问题已经解决。

1.1.2 Windows 7 平台下 Node+WebMatrix

你可以按照之前 wiki 页面中的安装步骤完成 Windows 系统下 Node 的安装。但一般来说，Node 只是作为 Windows Web 开发架构的一部分。

目前有两种 Windows Web 开发架构适合使用 Node。一种叫做 Windows Azure 云平台，允许开发人员将程序托管在远端服务器（称为云）上。Microsoft 提供了关于如何在 Windows Azure SDK 中安装 Node 的说明，所以本章不涉及这一过程（不过稍后会对该 SDK 进行说明）。

提示

Windows Azure SDK for Node 安装说明：https://www.windowsazure.com/en-us/develop/nodejs/。

另一种在 Windows 系统(本处指 Windows 7)使用 Node 的方式是将 Node 与 Microsoft WebMatrix 集成。WebMatrix 是网络开发人员用于集成开源技术的一种工具。以下是在 Windows 7 WebMatrix 中集成并运行 Node 的步骤：

1. 安装 WebMatrix；

2. 使用最新的 Windows 安装包安装 Node；

3. 安装 iisnode for IIS Express 7.x，以便让 Windows 上的 IIS 支持 Node 应用程序；

4. 为 WebMatrix 安装 Node 模板，使用模板可以简化 Node 开发。

如图 1-1 所示，使用 Microsoft Web Platform Installer 安装 WebMatrix，同时会安装 IIS Express。IIS Express 是 Microsoft Web 服务器的开发版本。

WebMatrix 的下载地址：http://www.microsoft.com/Web/Webmatrix/。

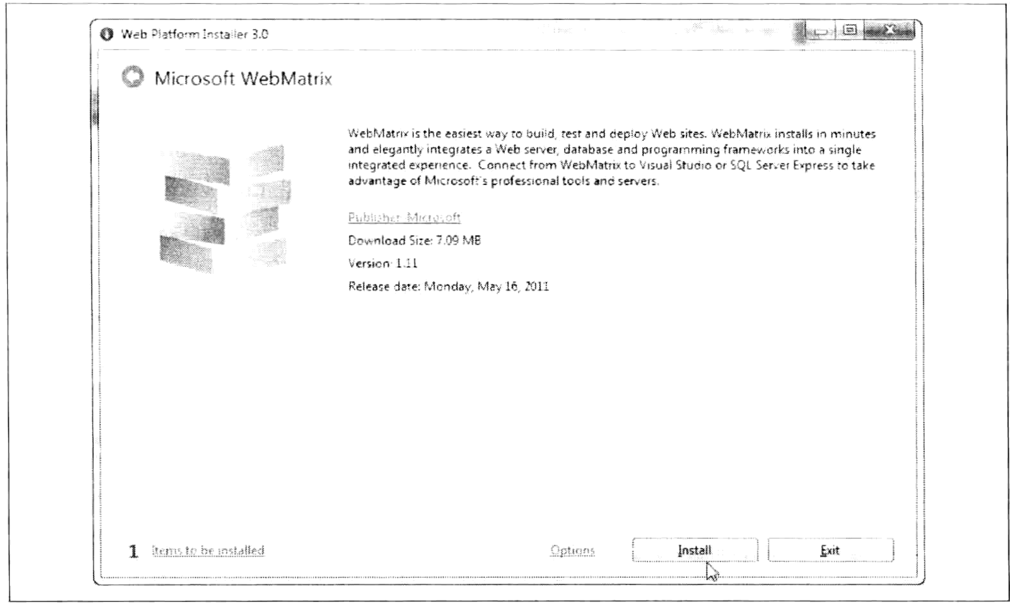


图 1-1　在 Windows 7 中安装 WebMatrix

WebMatrix 安装完成后，可以使用 Node 官网（http://nodejs.org/#download）提供的安装包安装最新版本的 Node。过程很简单，一键安装完成后打开命令行窗口输入 node 检查是否能正常运行，如图 1-2 所示。

为了能让 Node 与 Windows 下的 IIS 一起工作，我们需要安装 iisnode。iisnode 是一个本地 IIS7.x 模块，由 Tomasz Janczuk 创建并维护。它在 Git Hub 站点中的链接地址为：https://github.com/tjanczuk/iisnode。

iisnode 有 x86 和 x64 两种，但是对于 x64 系统，两个都需要安装。

图 1-2　在 Windows 命令窗口中测试 Node 是否被正确安装

在安装 iisnode 的过程中，可能会碰到图 1-3 所示的弹出窗口，提示系统环境中未安装 Microsoft Visual C++ 2010 Redistributable Package。如果你碰到了这种情况，则需要安装该组件，同时还需要保证你安装的版本与当前正在安装的 iisnode 的版本是兼容的：可以从 x86 版本（从地址 http://www.microsoft.com/download/en/details. aspx?id=5555 获得）或 x64 版本（从地址 http://www .microsoft.com/download/en/details. spx?id=14632 获得）中选择安装，或者两个版本都安装。在成功安装好 C++ Redistributable Package 后，就可以再次运行 iisnode 安装程序了。

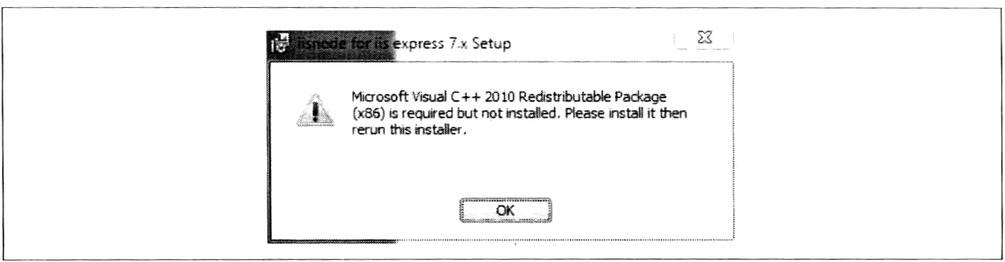


图 1-3　提示对话框：需要安装 C++ redistributable package

如果你还想安装 iisnode 的附带的示例代码，则需要以管理员权限打开命令窗口，进入到 iisnode 的安装目录中（“Program Files for 64- bit”或者“Program Files (x86)”）然后运行名为 setupsamples.bat 的批处理文件。

最后还需要为 WebMatrix 下载并安装 Node 模板，这样就完成了 WebMatrix/Node 的所有安装。Node 模板由 Steve Serson 创建，可以在地址 https://github.com/SteveSanderson/Node-Site-Templates-for-WebMatrix 下载。

你可以通过如下步骤来测试以上工作的正确性，首先运行 WebMatrix，在打开页面中选择“Site from Template”选项。然后，图 1-4 所示页面会打开，你可以看到两个 Node 模板选项：一个是“Express”（在第 7 章介绍），另一个是“a basic, empty site configured for Node”。选择后者，并使用“First Node Site”或者一个你喜欢的名称为新建的站点命名。

使用 WebMatrix 生成的新站点如图 1-5 所示。点击页面左上角的 Run 按钮，浏览器窗口会弹出并显示包含有“Hello, world!”信息的页面。

如果你在使用 Windows 防火墙，第一次运行一个 Node 应用程序时，你可能会得到一个如图 1-6 所示的警告信息。这时，你需要点击“Private networks”选项然后按下“Allow access”按钮，以便让防火墙知道该程序是被允许在开发机器的私有网

络上通信的。

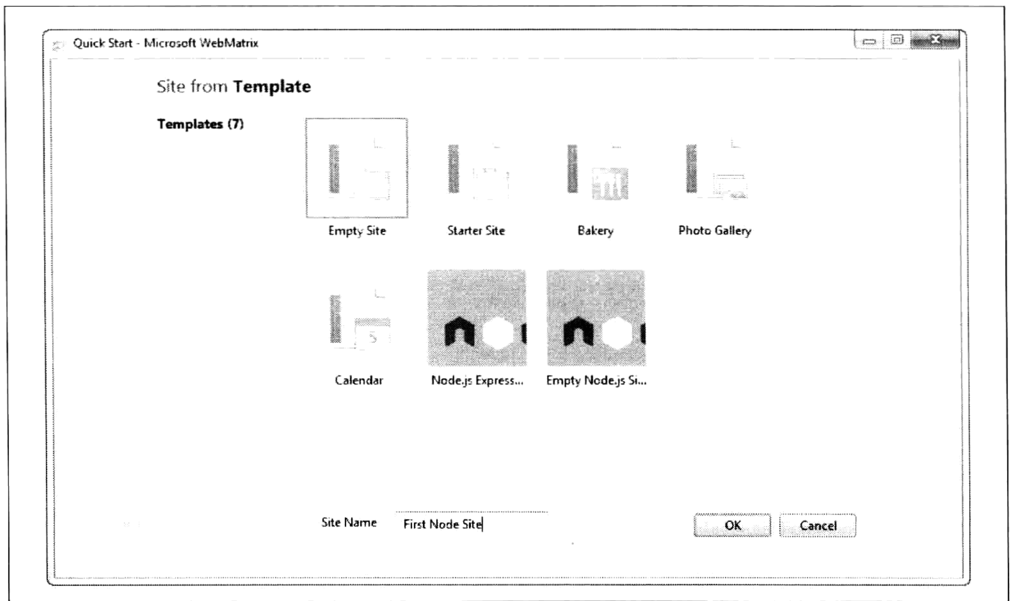


图 1-4 在 WebMartix 中使用模板创建 Node 站点

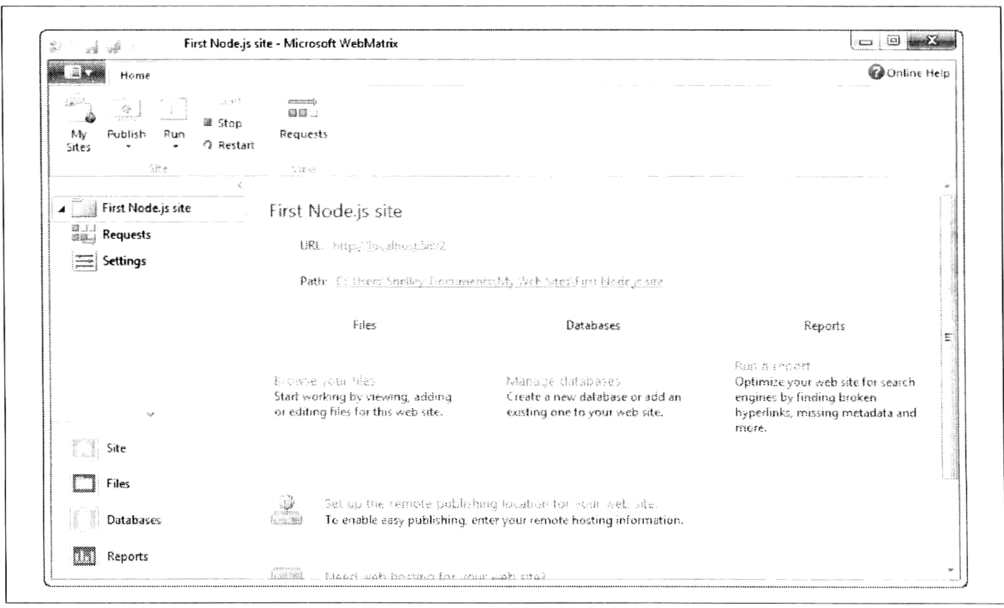


图 1-5 使用 WebMatrix 新生成的 Node 站点

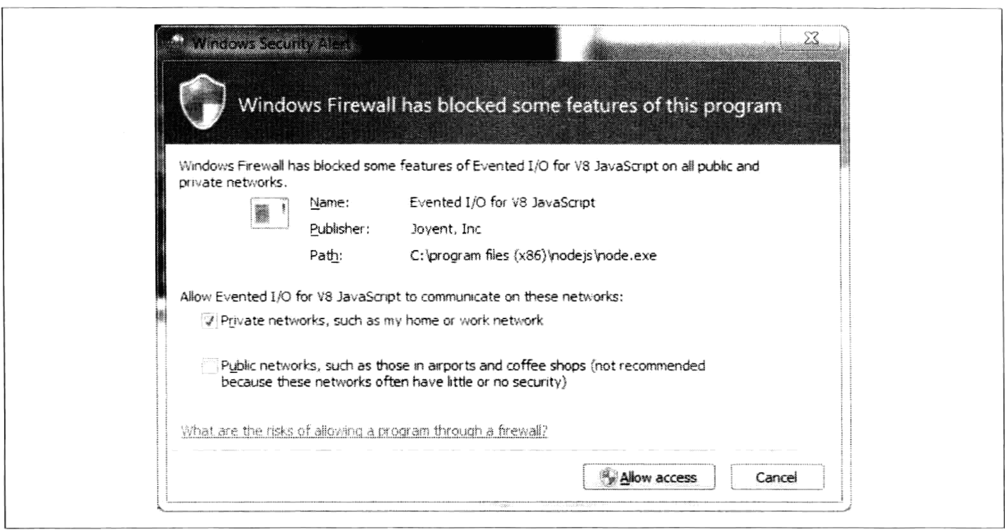


图 1-6 Windows 防火墙拦截 Node 应用程序时的警告信息，以及允许访问所需勾选项

在新生成的 WebMatrix Node 工程中，存在一个名为 app.js 的文件。这是一个 Node 程序文件，包含了如下代码：

```
var http = require('http');
http.createServer(function (req, res) {
   res.writeHead(200, { 'Content-Type': 'text/html' });
   res.end('Hello, world!');
}).listen(process.env.PORT || 8080);
```

我会在本章第二部分对这段代码的相关部分进行详细介绍。目前，我们只需要知道这段代码实现了一个简单的服务端，它只给客户端返回“Hello, world!”。而且我们可以在任何安装有 Node 环境的系统中运行这段代码，并且能得到同样的功能。

提示

若需要在 WebMatrix 中查看 iisnode 的示例，则需要在 WebMatrix 中选择选项“Site from Folder”，然后在弹出的对话框中输入如下内容：%localappdata%\iisnode\www。

1.1.3 升级 Node

偶数版本号代表了 Node 的稳定发行版本，例如当前的 0.8.x 版本，而奇数版本号表示 Node 的开发版本（当前是 0.9.x）。在你有一些 Node 使用经验前，我建议选择稳定发行版本。

升级 Node 版本并不复杂。如果你使用安装包来做升级，那么旧版本的 Node 将自动被新版本覆盖。如果你直接使用源代码升级，为了避免潜在的混乱或者文件冲突，

你始终应该先在旧版源代码目录中执行卸载命令，然后再安装新版本。在 Node 源代码目录中，可以运行如下 make 命令执行卸载：

```
make uninstall
```

下载新的源代码，编译然后安装它。当再有版本更新时，你需要再次执行以上过程。

升级 Node 的挑战在于，新版本的 Node 是否还能兼容特定的环境、模块或者 Node 应用程序。大多数情况下，你不会碰到版本问题。然而，如果你碰到了，你可以使用 Node 版本管理器（Nvm, Node Version Manager）在多个 Node 版本之间切换。

你可以从 GitHub 上下载 Nvm，地址是 https://github.com/creationix/nvm。与 Node 一样，你必须在你的系统中编译并安装 Nvm。

使用 Nvm 安装指定版本的 Node：

```
nvm install v0.4.1
```

使用如下命令切换到指定 Node 版本：

```
nvm run v0.4.1
```

查看可用的 Node 版本信息：

```
nvm ls
```

1.2 开始 Node 开发

现在你已经安装了 Node，是时候开始编写第一个 Node 应用程序了。

1.2.1 Hello, World in Node

为了测试新的开发环境、语言或者工具，第一个写出来的程序往往是“Hello,World”。我们同样也将使用 Node 创建一个“Hello,World”程序，它仅仅简单的向访问它的用户输出问候语。

示例 1-1 包含了使用 Node 创建 Hello,World 程序需要的全部文本代码。

示例 1-1 Node 版 Hello, World

```
// load http module
var http = require('http');

// create http server
http.createServer(function (req, res) {

  // content header
```

```
  // content header
  res.writeHead(200, {'content-type': 'text/plain'});

  // write message and signal communication is complete
  res.end("Hello, World!\n");
}).listen(8124);

console.log('Server running on 8124');
```

代码被保存在名为 *helloworld.js* 的文件中。作为服务端开发使用的 Node，其代码既不冗长，也不模糊。即使是一个不曾接触过 Node 的人，也可以直观地看出代码所表达的意思。不过可能最吸引人的是，它采用了我们很熟悉的 JavaScript 语言编写程序。

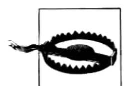
可以在 Linux 系统中使用命令行，或在 Mac OS 中使用终端窗口，或在 Windows 中使用命令窗口，运行如下命令来启动示例程序：

```
node helloworld.js
```

在程序成功运行后，会输出如下信息：

```
Server running at 8124
```

现在，你应该可以使用任何浏览器访问站点了。如果应用程序是在本地机器上运行的，可以使用 localhost:8124。如果是在远程机器上运行的，需要使用远程机器的 URL 并访问 8124 端口。在浏览器中，一个显示有"Hello,World!"内容的页面将会被显示出来。到目前为止，你已经成功创建了第一个完整的并且可以正常工作的 Node 应用程序了。



警告

如果是在 Fedora 系统中安装 Node 环境，需要留意 Node 会被重命名，以避免与系统中现有功能的冲突。更多详细信息请查阅 http://nodejs.tchol.org/。

由于我们没有在 node 命令后使用&（使应用程序在后台运行），在程序启动后，你就不能返回到命令行了。不过你依然可以继续正常访问应用程序，并且同样的信息会被显示在浏览器窗口中，直到你使用 Ctrl+C 来停止程序，或使用 kill 命令来中止 node 进程。

如果想在后台运行应用程序，在 Linux 系统中可以使用如下命令：

```
node helloworld.js &
```

之后，你需要通过"ps–ef"命令找到进程对应的 ID，然后使用 kill 命令手动关闭该进程（比如进程 ID 为 3747）：

```
ps -ef | grep node
kill 3747
```

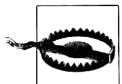
如果你退出终端窗口，node 进程同样也会中止。

提示

在第 16 章中，我会讨论如何创建一个可持久的 Node 应用程序安装。

你不能启动另外一个监听同一端口的 Node 应用程序：因为在同一时间、同一端口上，只能运行一个 Node 应用程序。如果你的 Apache 工作在端口 80 上，你也不能在该端口上启动 Node 应用程序。你必须为每一个应用使用不同的端口号。

如果你在使用 WebMatrix 的话，也可以将 helloworld.js 作为一个新文件添加到之前生成的 WebMatrix 站点项目中。你只需要打开站点，从菜单中选择“New File…”选项，然后将示例 1-1 中的代码输入到创建的文件中，点击运行按钮。



警告

WebMatrix 会覆盖 Node 程序中使用的端口号。当你运行应用程序后，你只能通过项目中定义好的端口访问站点，而不能使用在 http.Server.listen 方法中指定的端口。

1.2.2 分析“Hello,World”

我会在后续章节对 Node 应用程序进行更多剖析。但是现在，我们先来仔细看看“Hello, World”程序。

在示例 1-1 中，第一行代码是：

```
var http = require('http');
```

Node 中的许多功能通过外部程序或库来提供，我们叫它模块（modules）。这句代码其实就是用来加载 HTTP 模块，然后指派给一个本地变量。HTTP 模块能提供基本的 HTTP 功能，可以让应用程序支持对网络的访问。

下一句代码是：

```
http.createServer(function (req, res) { ...
```

在这行代码中，使用了 createServer 方法创建了一个新的服务器，并且传递了一个匿名函数来作为该方法时的参数。这个匿名函数就是 requestListener 函数，它有两个参数：一个代表服务器收到的请求（http.ServerRequest），另一个代表服务器的响应（http.ServerResponse）。

在匿名函数中，有如下代码：

```
res.writeHead(200, {'content-Type': 'text/plain'});
```

在 http.ServerResponse 对象中有一个 writeHead 方法，我们用它来发送响应信息的 HTTP 头，并且指定了 HTTP 状态码（status code）为 200，同时还提供了内容类型 content-type。你同样可以通过 headers 对象来设置其他 HTTP 响应头中需要的信息，例如 content-length 或者 connection：

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'accept': '*/*' }
```

writeHead 的第二个可选参数是 reasonPhrase，用来对状态码指定文本描述。

依据下面代码将“Hello,World!”内容放入响应信息中，并发送响应：

```
res.end("Hello, World!\n");
```

调用 http.ServerResponse.end 方法表示本次通信已经完成，所有响应信息的头和内容均已经被发送。注意：你必须为每一个 http.SErverResponse 对象使用该方法。

end 方法有两个参数：

- 一个数据块，可以是一个字符串或者 buffer 对象；
- 如果数据块是字符串对象，第二个参数用于指定编码方式。

这两个参数都是可选的，而且只有在字符串是非 utf8 编码的情况下才需要指定第二个参数，因为其默认值是 utf8。

我们也可以不在 end 方法中传递数据块，而使用另一个 write 方法：

```
res.write("Hello, World!\n");
```

然后：

```
res.end();
```

下面一句代码，表示了匿名函数和 createServer 函数的结束：

```
}).listen(8124);
```

http.Server.listen 方法紧接在 createServer 之后调用，用于在指定端口（本例中为 8124）监听接入的客户端连接。它的可选参数是一个 hostname 和一个回调函数。如果指定了 hostname，客户端将能通过 Web 地址的形式访问服务端了，比如 http://oreilly.com 或者 http://examples.burningbird .net。

提示

本章后半部分对 callback 函数有更多介绍。

listen 方法是异步的，这意味着在应用程序等待客户端连接建立时，不会阻塞程序的执行。listen 方法之后的所有代码都会被执行。而且当连接建立起来后，一个 listening 事件会被触发，传给 listen 方法的回调函数会被执行。

最后一句代码是：

```
console.log('Server running on 8124/');
```

console 是一个起源于浏览器环境并被 Node 采用的众多对象之一，大多数 JavaScript 开发人员对它都很熟悉。在此处，它提供了将文本信息输出到命令行（或者开发环境）的功能，而不再是输出信息到客户端浏览器中。

1.3 异步函数及 Node 事件循环

Node 的基本设计原则是将应用程序放置在单线程（或单进程）中执行，同时异步处理所有事件。

考虑下典型的 Web 服务器（如 Apache）是如何工作的。Apache 可以采用两种不同的方式处理传入的请求：一种方式是将传入的每个请求分配到独立的进程中直至请求被处理完毕；另一种方式则是为每一个请求生成单独的处理线程。

第一种方式（也称为 prefork multiprocessing model，或 prefork MPM）可以根据 Apache 配置文件中指定的值创建多个子进程。使用进程的优势在于被请求的应用（如 PHP 应用）无需考虑线程安全问题；缺点是每个进程占用独立内存，内存消耗大，应用的扩展性也不是很好。

第二种方式（也称为 worker MPM）是进程-线程混合方式。Apache 为传入的每个请求创建一个新的处理线程，这样对内存的使用更加有效，但这种方式要求应用必须是线程安全的。虽然现在流行的 PHP 语言是线程安全的，但却无法保证和它一起被使用的各种库也是线程安全的。

不管哪种方法，它们都可以应对并发请求。如果五个用户在同一时间访问一个 Web 应用，并且服务器也进行了相应设置，那么 Web 服务器就可以同时处理五个请求。

Node 的处理方式与上面两种不同。当您启动 Node 应用程序时，它会被创建并运行在一个单线程上。Node 会等待应用程序启动完成并开始捕获请求。在未处理完当前请求时，其他请求是不能被处理的。

这种处理方式听起来并不是很有效率，如果 Node 是通过事件循环和回调函数实现异步运行（在 Node 中，事件循环一般指轮询指定事件类型并在合适的时间调用事

件处理程序，而回调函数就是事件处理程序）的话，它是不应该低效的。

实际上，与一般单线程应用不同，当 Node 应用程序接收到用户请求时，虽然它会严格按照请求顺序初始化这些资源请求操作（如数据库请求或文件访问），但并不会一直等待操作完成或结果返回。相反，它会在操作请求中附加回调函数。当任何被请求的资源准备好或被请求的操作完成时，特定的事件会被触发，关联的回调函数也会被执行，回调函数会用请求到的资源或操作结果来做另一些事情。

如果五个用户在同一时间访问 Node 应用程序，并且该应用程序需要访问同一个文件中的资源时，Node 会为每个文件访问请求附加一个回调函数但并不等待返回。当资源变为可用时，回调函数会被调用，最终依次满足每个用户的需求。在此期间，Node 应用仍然可以处理其他同样或不同类型的用户请求。

尽管 Node 应用程序不是真正的并行处理用户请求，但其设计方式使得应用能繁忙且高效地处理用户请求，所以大多数人通常不会察觉到任何的响应延迟。最重要的是，它能非常有效地使用内存和其他有限的计算机资源。

1.3.1 使用异步方式读取文件

为了描述 Node 的异步特性，示例 1-2 修改了之前章节使用的 Hello World 程序。它不再输出“Hello,World!”，而是打开先前创建的 helloworld.js 文件并将其内容输出给客户端。

示例 1-2 异步方式地打开文件并写入数据

```
// load http module
var http = require('http');
var fs = require('fs');

// create http server
http.createServer(function (req, res) {

   // open and read in helloworld.js
   fs.readFile('helloworld.js', 'utf8', function(err, data) {

      res.writeHead(200, {'Content-Type': 'text/plain'});
      if (err)
         res.write('Could not find or open file for reading\n');
      else

         // if no error, write JS file to client
         res.write(data);
      res.end();
     });
}).listen(8124, function() { console.log('bound to port 8124');});

console.log('Server running on 8124/');
```

本示例中使用了一个新的文件系统模块（fs）。该模块对标准的 POSIX 文件操作进行了封装，提供了包括打开文件和访问文件内容等操作。示例 1-2 使用了模块中的 readFile 方法，并传入了多个参数，包括文件名称、文件编码方式以及匿名回调函数。

在示例 1-2 中，我想指出两个有关异步行为的实例，它们分别是附加在 readFile 方法和 listen 方法上的回调函数。

正如前面所讨论的，使用 listen 方法可以告诉 HTTP server 对象监听指定端口上的连接。Node 不会阻塞并等待连接建立，所以如果我们需要在连接建立时做些事情，就需要提供了一个回调函数，如示例 1-2 所示。

当网络连接建立时会触发监听事件，该事件会触发 listen 方法绑定的回调函数，进而将信息输出到控制台。

第二，也是更重要的实例是附加在 readFile 上的回调函数。相对来说，访问文件是一个耗时的操作。如果一个单线程应用程序被多个客户同时访问，而该应用处理每一个请求时都需要进行文件访问操作的话，它可能很快就会陷入瘫痪而无法使用。

解决方法就是采用异步方式打开文件和读取文件内容。只有当内容已经读入数据缓冲区（或读取失败时），附加在 readFile 方法上的回调函数才会被调用。错误信息（如果有的话）和读取到的数据（如果没有错误发生时）会作为参数传送给回调函数。

在回调函数中需要进行错误检查，如果不存在错误，则将读取到的数据返回给客户端。

1.3.2 观察异步程序流程

大多数人使用 JavaScript 编写客户端应用程序，这些程序只能被用户在单个浏览器中运行。而在服务端使用 JavaScript 编写程序可能看上去会有些古怪和陌生。创建允许多人同时访问的 JavaScript 服务应用可能就更让人觉得陌生了。

Node 的事件循环和异步函数调用可以帮助我们，这让编写服务端 JavaScript 程序变得容易且更有信心。但一定要注意的是，我们正在一个新的不同以往的环境中做 JavaScript 开发。

为了更好地描述新环境的不同，我创建了两个新的应用：一个提供服务，另一个用于测试服务。示例 1-3 显示了服务程序的代码。

在代码中，一个函数被调用，以同步方式按顺序输出从 1～100 的数字。然后程序以类似于示例 1-2 的方式打开一个文件，但这次文件名是以字符串参数的形式传递给函数的。此外，程序还使用了一个定时器，文件打开操作被安排在定时器

超时之后执行。

示例 1-3　输出数字序列和文件内容的服务程序

```
var http = require('http');
var fs = require('fs');

// write out numbers
function writeNumbers(res) {

  var counter = 0;

  // increment global, write to client
  for (var i = 0; i<100; i++) {
    counter++;
    res.write(counter.toString() + '\n');
  }
}

// create http server
http.createServer(function (req, res) {

   var query = require('url').parse(req.url).query;
   var app = require('querystring').parse(query).file + ".txt";

   // content header
   res.writeHead(200, {'Content-Type': 'text/plain'});

   // write out numbers
   writeNumbers(res);

   // timer to open file and read contents
   setTimeout(function() {

      console.log('opening ' + app);
      // open and read in file contents
      fs.readFile(app, 'utf8', function(err, data) {
         if (err)
            res.write('Could not find or open file for reading\n');
         else {
            res.write(data);
         }
         // response is done
         res.end();
      });
   },2000);
}).listen(8124);

console.log('Server running at 8124');
```

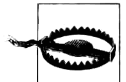
输出数字的循环体起到了延迟应用程序执行的效果，以便模拟密集计算过程，该过程会引起应用程序阻塞直到计算完成。在这里 setTimeout 是另一个异步函数，它会紧接着调用第二个异步函数：readFile。所以该应用程序结合了异步和同步流程。

创建一个名为 main.txt 的文本文件，可以包含任何你想要的内容。运行应用程序并通过 Chrome 浏览器访问，访问时使用的 url 需要带有 file=name 的查询字段，应用程序将生成如下控制台输出：

```
Server running at 8124/
opening main.txt
opening undefined.txt
```

前两行输出信息很容易理解。第一行由程序末尾 console.log 输出，第二行是在文件被打开时输出的。但是，第三行的 undefined.txt 是怎么回事？

其实，当处理来自浏览器的 Web 请求时，浏览器可能会发送多个请求。例如，一般浏览器可以发送第二个请求，寻找一个叫 favicon.ico 的文件。正因为如此，当你在处理查询字符串时，你必须检查看看需要的数据是否被提供，并忽略没有数据的请求。



警告

当期望从查询字符串中获取某些参数时，浏览器发送多个请求的特点可能会影响到你的应用程序。因此，必须相应的调整应用，并在几个不同的浏览器上进行测试。

到目前为止，我们对 Node 应用程序所做的所有测试都是从浏览器中进行的。这样我们无法对其进行压力测试来体现 Node 应用程序的异步特性。

示例 1-4 是一段非常简单的测试代码。它使用 HTTP 模块多次向服务程序发送请求。这些请求并不是按异步方式发送的。然而，我们同时也可以使用浏览器访问该服务。两者相结合，就可以达到异步测试应用程序的目的。

提示

14 章将介绍如何创建异步测试应用程序。

示例 1-4　测试小程序，调用 Node 服务程序 2000 次

```
var http = require('http');

//The url we want, plus the path and options we need
var options = {
   host: 'localhost',
   port: 8124,
   path: '/?file=secondary',
   method: 'GET'
};

var processPublicTimeline = function(response) {
   // finished? ok, write the data to a file
   console.log('finished request');
```

```
};

for (var i = 0; i < 2000; i++) {
   // make the request, and then end it, to close the connection
   http.request(options, processPublicTimeline).end();
}
```

创建第二个文本文件，并命名为 secondary.txt。内容与 main.txt 有显著不同即可。

在确定 Node 服务程序运行起来后，启动测试程序：

```
node test.js
```

在测试程序运行的同时，使用浏览器手动访问服务程序。观察服务程序在控制台的输出信息，你会看到来自浏览器的手动请求和来自测试程序的自动请求都能被处理。并且，结果与我们所期望的一致，请求到的页面中包含了如下信息：

- 1 到 100 的数字；
- 文本文件的内容，在本示例中是 main.txt 的内容。

现在，让我们尝试做一些改动。在示例 1-3 中，将循环体中计数用的局部变量 counter 改为全局变量，并重新启动应用程序。然后运行测试程序，并在浏览器中访问该页面。

输出结果显然发生改变。返回的页面内容不再是从 1 开始到 100 的数字，而是返回从类似 2601 和 26301 这样的数字开始的，按顺序排列的连续 99 个数字，只是初始值不同。

原因必然是因为使用了全局变量 counter。因为在浏览器中手动访问页面时，自动测试程序也在做同样的事，他们都会更新 counter。另外由于手动和自动测试程序的请求被按照顺序一个个的处理，因此没有争用共享数据的情况发生（在多线程环境中，并行访问共享数据同时保证线程安全是最主要的问题），如果你之前有期望输出一致的起始值，这里的结果可能会让你感到些许意外。

现在再次更改应用程序，但这次我们删除变量 app 之前的 var 关键字（“不小心的”使其成为一个全局变量）。曾几何时，在编写客户端 JavaScript 时，我们总是忘记使用 var 关键字。或许也只有当我们程序中用到的某些库使用了相同的变量名时，才会发现这种错误。

运行测试程序同时通过浏览器手动访问 Node 服务程序多次。你会发现浏览器得到的页面中偶尔会包含 secondary.txt 文件的内容，而不是期望的 main.txt 文件内容。这是因为在应用程序处理请求（带有文件名）和真正执行文件打开操作之间有一段时间间隔，在此间隔期间测试程序的持续访问会使得服务程序修改 app 变量。测试程序之所以能够引起这样的问题，是因为我们做了一个异步功能调用，在异步调用开始执行而没有完成前，Node 会放弃对当前请求处理过程的控制权来处理另一个用户请求。

提示

这个示例说明了正确使用 var 关键字在 Node 中是至关重要的。

1.4 Node 的优势

现在，你至少已经安装了一个（或多个）可用的 Node 版本。

你也有机会去创建多个 Node 应用程序，并且通过测试发现同步与异步编码之间的差异（以及不小心忘记了 var 关键字的情况）。

Node 使用的函数调用并不都是异步的。一些对象针对同一功能可能会同时提供同步与异步版本的实现。尽管如此，如果你能尽可能多的进行异步编码，将会使 Node 工作得更好。

Node 的事件循环和回调函数给我们带来了两大好处。

首先，应用程序可以很容易地扩展，因为执行一个单线程并不会有非常大的资源开销。如果我们用 PHP 创建一个类似于示例 1-3 的 Node 应用时，用户会看到相同的页面（但你的系统肯定会注意到其中的差别）。如果你在 Apache 中运行该 PHP 应用程序并默认使用 prefork MPM，那么在每次应用程序接收到请求时，请求将被放在一个单独的子进程中来处理。倘若你能够拥有一个强大且高效的负载均衡系统，同时并行运行（最多）几百个子进程或许是有可能的。不过当访问量超过这个数字时，就意味着客户端需要等待响应了。

Node 的第二个优势在于无需诉求于多线程开发，却达到了节约又能高效使用资源的目的。换句话说，你不必创建一个线程安全的应用程序。倘若你曾经开发过要求线程安全的应用程序，或许此时你会感到非常欣喜。

无论如何，正如前面示例应用程序所展示的那样，你不是在开发浏览器中运行的 JavaScript 应用程序。当你开发异步应用时，不能假设一个异步函数在另一个函数被调用前就执行完成，因为这是无法保证的（除非你在第一个回调函数中调用另一个）。此外，全局变量在 Node 中是非常危险的，特别是在你忘记了 var 关键字时。

虽然这些问题存在，但并不妨碍我们在工作使用 Node，特别是考虑到它的低资源需求优势，以及不必担心线程安全问题。

提示

或许喜欢 Node 的原因也可以是毫无顾虑地写 JavaScript 代码而无需担心 IE6 了。

第2章

Node 与 REPL

尝试使用 Node 编写自定义的模块或者应用程序时，并不需要每次运行写好的 JavaScript 文件来测试代码功能。Node 有一个交互式组件称为 REPL（read-eval-print-loop，读取求值列印循环），这将是本章的主题。

REPL（发音为“repple”）支持简化的 Emacs 风格行编辑和一小部分基本命令。在 REPL 中输入任何内容都与用 Node 运行 JavaScript 编写的文件具有相同的处理方式。事实上，可以使用 REPL 编写整个应用程序——这样就可以频繁地对程序进行测试。

本章涉及 REPL 的一些有趣的技巧以及如何使用这些技巧，包括如何替换浏览历史命令的底层机制以及命令行编辑等内容。

最后，如果内建的 REPL 不能提供你所需要的交互环境，本章的后续部分会介绍用于创建自定义 REPL 的 API。

提示

如何使用 REPL：http://docs.nodejitsu.com/articles/REPL/how-to-use-nodejs- replNodejitsu。网站提供的如何创建自定义 REPL 的教程：http://docs.nodejitsu.com/ articles/REPL/how-to-create-a-custom-repl。

2.1 REPL：先睹为快和未定义的表达式

只需要输入 node 命令就可以运行 repl，不需要提供任何 Node 应用文件作参数：

```
$ node
```

REPL 默认尖括号>为命令行提示符。在该符号之后输入的任何内容都由底层的 V8 JavaScript 引擎进行处理。

REPL 的使用很简单，就像在文件中编写 JavaScript 一样：

```
> a = 2;
2
```

REPL 可以即时打印输入的任何表达式的结果。在上面例子中，表达式的结果是 2。下面这个例子中表达式结果是有三个元素的数组：

```
> b = ['a','b','c'];
['a','b','c']
```

可以使用下划线“_”调用上一个表达式。本例中，a 为 2，结果表达式两次自增 1：

```
>  a = 2;
2
>  _ ++;
3
>  _ ++;
4
```

还可以用下划线访问该对象的属性或者调用方法：

```
> ['apple','orange','lime']
[ 'apple','orange','lime']
> _.length
3
>3+ 4
7
> _.toString();
'7'
```

在 REPL 中也可以使用 var 关键字。可以在之后通过变量名访问表达式或者变量。但是这样可能会得到意料之外的结果。比如，在 REPL 中输入以下命令行：

```
var a = 2;
```

该表达式返回值并不是 2，而是 undefined。表达式结果为 undefined 的原因是变量赋值的表达式并不返回变量的值作为表达式的值。

理解以下代码，多少可以解释 REPL 中的这种现象：

```
console.log(eval('a = 2'));
console.log(eval('var a = 2'));
```

将上两行代码写入文件并用 Node 运行，返回值如下：

```
2
undefined
```

第二行代码并没有返回结果给 eval，因此返回值为 undefined。要记得，REPL 是

read-eval-print loop，重点在 eval，就是求值。

但是，在 REPL 中你仍旧可以使用该变量，像在 Node 应用中一样：

```
> var a = 2;
undefined
> a++;
2
> a++;
3
```

后两条命令有返回值，由 REPL 打印输出。

提示

之后会说明如何创建自定义的 REPL——不输出 undefined，参见 2.3.3 节。

按 Ctrl+C 键两次或者 Ctrl+D 键一次退出 REPL。2.3.1 节介绍了其他退出方法。

2.2 REPL 的优势：更好地理解表层之下的 JavaScript

下例是一个 REPL 的典型示范：

```
> 3 > 2 > 1;
false
```

这段代码很好地解释了 REPL 的工作原理。一眼看上去会认为期望的输出值为 true，因为 3 大于 2，2 大于 1。但是在 JavaScript 中，表达式是从左到右计算的，每个表达式的返回值作为下一个表达式的一部分进行计算。

以下 REPL 中的语句可以帮助你更好地理解前端代码：

```
> 3 > 2 > 1;
false
> 3 > 2;
true
> true > 1;
false
```

现在这个结果看起来就合理多了。整个计算过程如下：首先计算表达式 3>2，返回 true；之后用 true 值与数字 1 进行比较。JavaScript 提供了自动类型转换，true 和 1 被认为是相等的值。因此，true 不大于 1，返回值为 false。

REPL 有助于我们发现 JavaScript 中这些有趣的地方。希望代码经过 REPL 的测试之后，应用程序中不会出现无法预测的结果（比如期望得到 true 却得到了 false）。

2.3 多行以及更复杂的 JavaScript

你可以像写文件一样在 REPL 中输入 JavaScript，包括导入 module 的 require 语句。以下代码显示了如何使用 Query String(qs)module：

```
$ node
> qs = require('querystring');
{ unescapeBuffer: [Function],
  unescape: [Function],
  escape: [Function],
  encode: [Function],
  stringify: [Function],
  decode: [Function],
  parse: [Function] }
> val = qs.parse('file=main&file=secondary&test=one').file;
[ 'main', 'secondary' ]
```

由于没有使用 var 关键字，表达式的结果被直接输出，在本例中是 querystring 对象的接口。预期之外的收获是用这种方式不仅可以访问对象，同时还可以了解更多关于对象的可用接口。但是，如果不想看到可能出现的长文本输出，请使用 var 关键字：

```
> var qs = require('querystring');
```

可以用 qs 变量访问 querystring 对象的任一方法。

为了兼容外部模块，REPL 可以处理多行表达式，提供了可以嵌套使用的文本标识符，跟在大括号{}之后：

```
> var test = function (x, y) {
... var val = x * y;
... return val;
... };
undefined
> test(3,4);
12
```

REPL 提供重复的点 "." 符号跟在开放的大括号后面表示输入命令未完成，该符号同样可以用于不闭合的小括号：

```
> test(4,
... 5);
20
```

层级间的递进需要更多的点符号。这在交互式环境中是必须的，否则会在输入过程中迷失了自己当前所在的位置：

```
> var test = function (x, y) {
```

```
... var test2 = function (x, y) {
..... return x * y;
..... }
... return test2(x,y);
...}
undefined
> test(3,4);
12
>
```

以下代码是一个完整的Node应用程序，可以在REPL中输入或者复制粘贴并运行：

```
> var http = require('http');
undefined
> http.createServer(function (req, res) {
...
...     // content header
...     res.writeHead(200, {'Content-Type': 'text/plain'});
...
...    res.end("Hello person\n");
... }).listen(8124);
{ connections: 0,
    allowHalfOpen: true,
    _handle:
   { writeQueueSize: 0,
     onconnection: [Function: onconnection],
     socket: [Circular] },
    _events:
    { request: [Function],
     connection: [Function: connectionListener] },
    httpAllowHalfOpen: false }
 >
undefined
> console.log('Server running at http://127.0.0.1:8124/');
Server running at http://127.0.0.1:8124/
Undefined
```

可以通过浏览器访问该应用，这与用Node运行程序文件没有差别。而且，从REPL返回的response如上述粗体文本所示。

事实上，REPL最实用之处在于快捷查看对象。例如，Node核心对象global在Node.js官网上文档很少。为了更好地了解该对象，在REPL中将global对象传递给console.log方法，如下：

```
> console.log(global)
```

下面这行代码具有相同的结果：

```
> gl = global;
```

这里不再复制 REPL 中的运行结果。global 对象接口非常多，你可以安装后自己尝试。这个练习的关键在于告知一种可以在任何时候简单快捷地查看对象接口的方法，不必去死记硬背需要调用什么方法、什么属性是可用的。

提示

更多关于 global 对象请见第 3 章。

在 REPL 中可以使用上下箭头遍历之前输入的命令。这样可以很方便地查看之前的操作，也可以一定程度上提供编辑历史命令的能力。

阅读 REPL 中的如下代码：

```
> var myFruit = function(fruitArray,pickOne) {
... return fruitArray[pickOne - 1];
... }
undefined
> fruit = ['apples','oranges','limes','cherries'];
[ 'apples',
  'oranges',
  'limes',
  'cherries' ]
> myFruit(fruit,2);
'oranges'
> myFruit(fruit,0);
undefined
> var myFruit = function(fruitArray,pickOne) {
... if (pickOne <= 0) return 'invalid number';
 ...return fruitArray[pickOne - 1];
  ... };
undefined
> myFruit(fruit,0);
 'invalid number'
> myFruit(fruit,1);
'apples'
```

在以上输入中没有体现出来的是，每次在修改方法检查输入值时，首先向上查找之前的命令找到函数定义，回车重新运行该方法。每添加新的语句，再用方向键重复上述操作直到完成该函数。同时也使用向上方向键重复函数调用，输出 undefined。

看起来似乎多做了很多工作只是为了避免重复输入，但是如果使用正则表达式，如以下例子：

```
> var ssRe = /^\d{3}-\d{2}-\d{4}$/;
```

```
undefined
> ssRe.test('555-55-5555');
true
> var decRe = /^\s*(\+|-)?((\d+(\.\d+)?)|(\.\d+))\s*$/;
undefined
> decRe.test(56.5);
true
```

则在写出一个正确的正则表达式之前通常要反复修改好几次（我并不擅长正则表达式）。用 REPL 做正则表达式的测试非常便捷，可以避免重复输入很长的正则表达式的痛苦。

幸运的是在 REPL 中只需要使用方向键就可以找到创建正则表达式的命令，修改，回车然后继续测试。

除了方向键，还可以使用 Tab 键自动补全。例如，在命令行中输入 va，按 Tab 键，REPL 会自动补全为 var。Tab 也可以用于自动补全任意的全局或者局部变量。表 2-1 列出了一些 REPL 中的按键功能。

表 2-1　　REPL 中的键盘控制

键盘输入	功能
Ctrl+C	终止当前命令。按 Ctrl+C 键两次直接退出
Ctrl+D	退出 REPL
Tab	自动补全全局或者局部变量
向上键	查找该条命令之前的输入
向下键	查找该条命令之后的输入
下划线（_）	上一条表达式的输出

如果你担心花很多时间在 REPL 中编程但是结束时却没有什么可以保存下来的文件，不用担心，.save 命令可以保存当前的上下文输入。这一命令和其他 REPL 的命令会在下一节中讲到。

2.3.1　REPL 命令

REPL 提供了一些常用命令的接口。在前一节中提到了.save 命令，该命令将当前语境中的输入保存在文件中。除非特意创建了一个新的语境或者使用.clear 命令，该文件会包含在当前 REPL 中所有的输入：

```
> .save ./dir/session/save.js
```

保存下来的只有你自己直接输入的文本，就像在文本编辑器中直接输入的一样。

以下是一个完整的 REPL 命令以及功能列表：

.break

如果多行输入发生混乱不知道当前位置时，使用.break 会重新开始。不过会丢失之前输入的多行内容。

.clear

重置语境并清空所有表达式。该命令可以使你重头再来。

.exit

退出 REPL。

.help

显示所有可用的 REPL 命令。

.save

将当前 REPL 会话保存至文件。

.load

将文件加载到当前会话（.load/path/to/file.js）。

如果使用 REPL 作为开发应用程序的编辑器，以下提示或许会有帮助：

经常使用.save 命令保存当前工作。尽管当前命令可以在历史记录中查找，但是重建代码依然是个很痛苦的过程。

提到关于命令的记录，接下来就会涉及如何定制自己的 REPL。

2.3.2 REPL 和 rlwrap

Node.js 官网上关于 REPL 的文档提到过设置环境变量，所以可以在 REPL 中使用 rlwrap。那么，rlwrap 是什么？又为什么要在 REPL 中使用 rlwrap 呢？

rlwrap 将 GNU readline 库的功能添加至命令行，增加键盘输入的灵活性。它监听键盘输入并提供更多的功能，比如增强行编辑以及提供命令历史浏览功能。

需要安装 rlwrap 和 readline 以便在 REPL 中使用这一功能，大部分 Unix 系统提供了简单的包安装。比如，在我的 Ubuntu 系统中，安装 rlwrap 只需要一行命令：

```
apt-get install rlwrap
```

Mac 用户可以使用自己的安装工具安装该程序。Windows 用户需要使用 Unix 环境模拟器，比如 Cygwin。

下面是一个简单的示例，如何在 REPL 中使用 rlwrap 将 REPL 的提示改为紫色：

```
env NODE_NO_READLINE=1 rlwrap -ppurple node
```

如果希望 REPL 的提示符一直是紫色的，可以在 bashrc 文件中添加别名（alias）：

```
alias node="env NODE_NO_READLINE=1 rlwrap -ppurple node"
```

同时改变提示符和颜色，命令如下：

```
env NODE_NO_READLINE=1 rlwrap -ppurple -S "::>" node
```

现在提示符变为以下符号并且是紫色的：

```
::>
```

rlwrap 组件的特殊之处在于它在多个 REPL 窗口中浏览命令历史的功能。REPL 默认只能浏览当前 REPL 会话的命令历史，但是使用 rlwrap，在关闭当前会话下一次重新进入 REPL 时，不仅可以浏览当前会话的历史命令，并且可以浏览之前会话的命令历史（以及其他命令行输入）。在下面例子中，显示的命令行不是手动输入的，而是通过方向键从命令历史中找出来的：

```
# env NODE_NO_READLINE=1 rlwrap -ppurple -S "::>" node
::>e = ['a','b'];
 [ 'a', 'b' ]
::>3 > 2 > 1;
false
```

即使 rlwrap 如此强大，但是每次输入无返回值的表达式时依然得到 undefined。然而，这一现象是可以改变的，这就是下一节中将要讨论的功能——创建自定义的 REPL。

2.3.3 定制 REPL

Node 提供了定制 REPL 的功能。为了实现该功能，首先需要引入 REPL 模块（repl）：

```
var repl = require("repl");
```

通过在 repl 对象上调用 start 方法创建新的 REPL。调用该方法的语法是：

```
repl.start([prompt], [stream], [eval], [useGlobal], [ignoreUndefined]);
```

所有参数都是可选择的。如果不传入参数，将会使用各参数的默认值，参数列表如下：

prompt

Default is>.默认值为>。

stream

Default is process.stdin.默认值为 process.stdin。

eval

Default is the async wrapper for eval.eval 的默认值为 async。

useGlobal

默认值为 false，新建一个语境而不是使用全局对象。

ignoreUndefined

默认值为 false。不要忽略 undefined 的返回值。

当我开始慢慢厌倦 REPL 在无返回值的表达式输出 undefined 时，我决定创建自己的 REPL。事实上只需要两行代码就可以实现（不包括注释）：

```
repl = require("repl");
//设置 ignoreUndefined 为 true，启动 REPL
repl.start("node via stdin> ", null, null, null, true);
```

在 Node 中运行 repl.js 文件：

```
node repl.js
```

然后就可以像使用 REPL 内建版本一样使用自定义的 REPL，除了自定义的提示符以及不会在变量赋值之后再看到讨厌的 undefined 之外，依然可以看到除了 undefined 之外的其他输出。

```
node via stdin> var ct = 0;
node via stdin> ct++;
0
node via stdin> console.log(ct);
1
node via stdin> ++ct;
2
node via stdin> console.log(ct);
2
```

在代码中我希望除了提示符和 ignoreUndefined 以外都使用默认值。设置其他参数

为 null 会使 Node 使用该参数的默认值。

可以用自定义的 REPL 替换 eval 方法。唯一的要求是有具体的格式:

```
function eval(cmd, callback) {
   callback(null, result);
}
```

stream 选项比较有意思。可以运行多个版本的 REPL，从标准输入（默认）或者 socket 中获取输入内容。Node.js 网站提供的 REPL 文档中给出了 REPL 监听 TCP socket 的例子，代码与下面这个例子类似:

```
var repl = require("repl"),
     net = require("net");
//设置 ignoreUndefined 为 true，启动 REPL
repl.start("node via stdin> ", null, null, null, true);
net.createServer(function (socket) {
  repl.start("node via TCP socket> ", socket);
}).listen(8124);
```

在 Node 中运行应用程序的时候会看到标准输入提示符。还可以通过 TCP 进入 REPL。我使用 PuTTY 作为 Telnet 客户端来登录支持 TCP 版本的 REPL，某种程度上来说是可行的。我必须先运行.clear 清理样式，但在尝试使用下划线表示上一行命令的时候，Node 无法解析该符号，如图 2-1 所示。

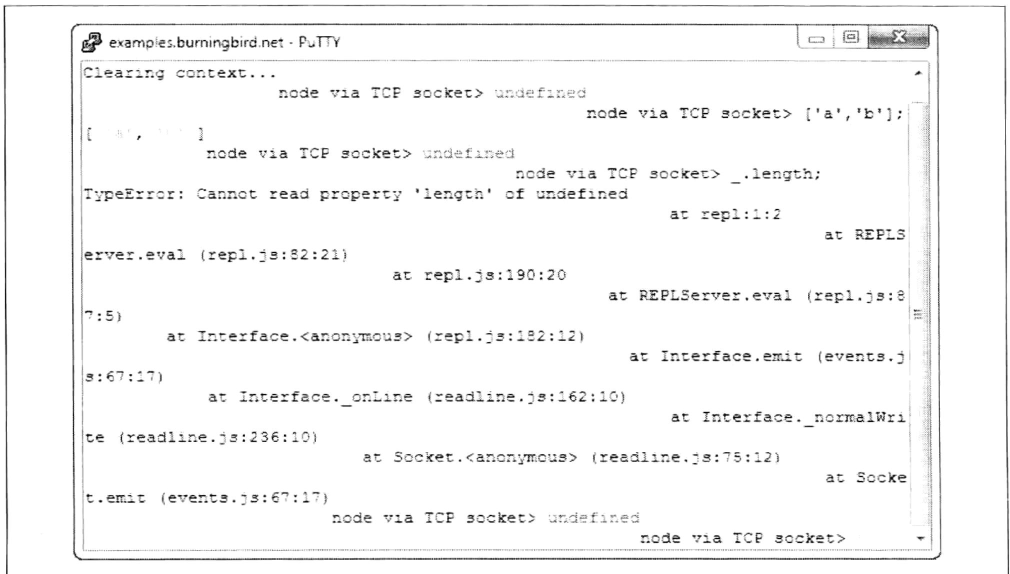


图 2-1　通过 TCP 的 PuTTY 和 REPL 并不一样

同样我也尝试过 Windows 7 Telnet 客户端，结果更糟糕。只有在 Linux Telnet 客户端使用时没有任何问题。

你可能预计到问题在于 Telnet 客户端的设置。然而，我并没有深究这一问题。因为从不安全的 Telnet Socket 运行 REPL 并不是我计划要做的事情，也并不推荐这一存在安全隐患的行为。就像在客户端代码中使用 eval()一样，并没有破坏或者泄露客户发给你需要运行的内容，但是结果比这样更糟糕。

可以用 UNIX socket 通信运行 REPL，比如 GNU Netcat：

```
nc -U /tmp/node-repl-sock
```

可以像使用 stdin 一样输入命令。但是需要了解的是，如果使用 TCP 或者 UNIX socket，任何 console.log 命令都会在 server 端打印输出，而不是客户端。

```
console.log(someVariable);//在 server 端输出
```

我想到一个很有用的应用程序，创建一个 REPL 程序，可以预加载模块。示例 2-1 的应用中，在 REPL 启动之后，http、os 和 util 模块被加载并赋值给当前语境的对应属性。

示例 2-1　创建可以预加载模块的自定义 REPL

```
var repl = require('repl');
var context = repl.start(">>", null, null, null, true).context;
// 预加载模块
context.http = require('http');
context.util = require('util');
context.os = require('os');
```

用 Node 运行该程序，显示 REPL 的提示符，可以访问之前加载的那些模块：

```
>>os.hostname();
'einstein'
>>util.log('message');
5 Feb 11:33:15 - message
>>
```

如果希望像 Linux 中的可执行程序一样运行 REPL 程序，将下行代码加入应用程序开头：

```
#!/usr/local/bin/node
```

修改文件权限为可执行并运行：

```
# chmod u+x replcontext.js
# ./replcontext.js
  >>
```

2.4 不可预计的意外——记得经常保存

Node 的 REPL 是一个便捷的交互式工具，可以使开发任务变得简单点。REPL 不仅可以在引入文件之前对 JavaScript 进行测试，并且可以边编写边测试直到完成时保存代码内容。

REPL 另一个有用的特性是可以创建自定义的 REPL，减少无用的 undefined 输出，预加载模块以及修改提示符或者 eval 方法等。

我强烈推荐在 REPL 中使用 rlwrap，可以跨 session 浏览历史命令。这一特性可以节省大量的时间。话说回来，我们之中谁不喜欢更多更强大的编辑特性呢？

当你进一步探索 REPL 的时候，要记住本章的一个重点：

意外经常发生，频繁保存。

如果你花费很多时间在 REPL 中进行开发，使用 rlwrap 浏览历史命令，则需要频繁地保存代码。在 REPL 中开发与其他编辑环境一样，意外的发生不可预计。所以我一再重复：意外的发生不可避免——频繁保存为上策。

提示

REPL 在 Node 0.8 中有较大修改，输入内建的模块名称，比如 fs，就可以加载该模块了。其他一些改进标注在 Node.js 官网提供的新的 REPL 文档中。

第3章

Node 核心库

在第 1 章中，我们实现了一个能够在 Node 中运行的经典的 Hello World 程序，尽管这是我们实现的第一个 Node 程序，但它依然使用到了 Node 核心库中的多个模块。Node 核心库包含了大量 API，这些 API 为我们提供了构建 Node 应用程序时所需要的功能模块。

本章将涉及更多 Node 核心库系统的细节知识。但是我们并不想提供一份详尽细致的核心库规范综述，因为它的确太过于庞大并且不时会有更新。相反，我们将着重对和核心库中的一些关键元素进行介绍，因为在后续章节我们将经常使用到或提及它们，并且其中有些也值得我们深入探讨。

本章所涉及的主题包括：

- Node 全局对象：例如 global、process 和 buffer
- 定时器方法，如 setTimeout
- socket 和 stream 模块功能综述
- Utilities 对象，特别是与 Node 继承特性相关的方法
- EventEmitter 对象及事件（event）

提示

当前最新 Node.js 稳定版的文档可参见 http://nodejs.or/api/。

3.1 全局对象：global、process 和 Buffer

这里有几种对象总是可以被所有 Node 应用程序所访问到，即使用户没有明确的在应用程序中包含这些对象的模块。在 Node.js 站点上，这种对象被统一归类并放在 globals 标签下。

我们已经使用过一种全局对象 require 来将其他模块包含进应用程序。同样也大量使用了另一个全局对象 console 来输出信息到控制台。全局对象对于实现 Node 的整体框架来说非常重要，但我们并没有必要立即了解或学会使用所有对象。只是其中一些关键对象还是值得我们仔细看看，因为它们可以帮助我们从某些关键方面理解 Node 是如何工作的。

本节我们将着重探索以下内容：

- global 对象，也是 Node 的全局命名空间；
- process 对象，它提供了一些关键功能，例如对三种标准 I/O 流的封装，以及将同步函数转换为异步回调的功能；
- Buffer 类，它提供了存储和操作原始数据的功能，同样它也是全局可见的；
- 子进程；
- 用于域名解析和 URL 处理的模块。

3.1.1 global

global 是 Node 的全局命名空间对象。从某些方面来说，它与浏览器环境中的 window 对象是相似的，都为用户提供全局属性和全局方法的访问能力，并且用户也不需要显式使用 global 命名空间。

在 REPL 里面，你可以使用下面这条命令将 global 对象输出到 console 里面：

```
> console.log(global)
```

输出信息中包含了其他全局对象的接口，以及大量当前系统的大量环境信息。

之前提到 global 与浏览器中的 window 对象是相似的，它们除了在一些方法和属性有效性上不一致外，还有一个关键不同是：浏览器中的 window 对象是一个真正全局的对象。如果你在客户端 JavaScript 中定义一个全局变量，它将能够被 Web 页面以及每一个独立的库访问到。然而，如果在 Node 模块中创建一个顶层

变量（在函数之外的变量），它仅仅在该模块中是全局的，而在其他模块中是不可见的。

当你在 REPL 里面定义一个全局变量时，你可以观察到 global 对象实际所发生的事情：

```
> var test = "This really isn't global, as we know global";
```

查看 global 对象内容：

```
> console.log(global);
```

你可以看到定义的变量已经变为一个 global 对象的属性，并且出现在输出信息的底部。另一个有趣的测试是将 global 对象赋值给一个变量，但是这个变量不使用 var 来定义：

```
gl = global;
```

global 对象将被输出到控制台中，同样在输出信息底部看到 gl 变量为 'circular refernce'：

```
> gl = global;
…
  gl: [Circular],
  _: [Circular] }
```

其他全局对象（包括 require 对象）以及全局方法都包含在 global 对象接口中。

当 Node 开发人员谈及 context 时，一般是指 global 对象。在第 2 章示例 2-1 的代码中，自定义的 REPL 对象被创建时，使用到了 context 对象。context 对象是一个全局对象。当应用程序创建一个自定义 REPL 对象时，还会生成一个新的 context 对象，并且这个 context 对象拥有自己的 global 对象。如果你在创建 REPL 对象时将 true 值传递给 useGlobal 参数的话，默认行为（新建 global 对象）将被覆盖，当前的 global 对象将被使用来创建一个 REPL 对象。

模块存在于自己的命名空间，这意味着，如果你在一个模块中定义一个顶层变量，它是不能被其他模块使用的。更重要的是，这意味着，只有那些被模块显式导出的部分才能被引用该模块的应用程序所使用。事实上，你不能在应用程序或其他模块中访问另一个模块的顶层变量，即使你刻意这样做。

为了说明这一点，下面的代码实现了一个非常简单的模块，模块中定义了一个名为 globalValue 的顶层变量，以及对该变量进行设置和读取的函数。在读取函数中，我们使用 console.log 方法调用将全局对象的内容打印出来。

```
var globalValue;
exports.setGlobal = function(val) {
   globalValue = val;
};
```

```
exports.returnGlobal = function() {
   console.log(global);
   return globalValue;
};
```

我们可能希望在打印出全局对象内容时，能看到 globalValue 变量，包括我们在程序中为其所设置的值。但实际并不如此。

启动 REPL 并通过 require 加载我们编写的新模块：

```
> var mod1 = require('./mod1.js');
```

调用模块提供的设置函数为模块中的顶层变量 globalValue 赋值，然后再调用读取函数将值取回来：

```
> mod1.setGlobal(34);
> var val = mod1.returnGlobal();
```

在返回 globalValue 变量值之前，console.log 方法会先输出 global 对象的内容。我们可以看到，在输出信息的最后部分包含了被引用模块的信息，但 val 变量的值是不确定的，因为该变量尚未设置。此外，输出信息中并不包含任何关于模块顶级变量 globalValue 的信息：

```
mod1: { setGlobal: [Function], returnGlobal: [Function] },
_: undefined,
val: undefined }
```

如果我们再次运行命令，可以看到应用程序变量 val 被设置了新值，但我们仍然不能看到 globalValue：

```
mod1: { setGlobal: [Function], returnGlobal: [Function] },
_: undefined,
val: 34 }
```

调用模块对外提供的方法是访问模块内数据的唯一途径。对于 JavaScript 开发人员来说，这意味着，由于不小心使用了重复的全局变量名称而引起的数据冲突问题将会大大减少。

3.1.2 process

每个 Node 应用程序都是一个 process 对象实例，正因为如此，应用程序自然就能直接使用某些内建于 process 对象的功能。

process 对象中的许多方法和属性能提供关于应用程序身份标识和当前运行环境的信息。调用 process.execPath 方法可以返回当前 Node 应用程序的执行

路径，process.version 提供了 Node 版本信息，process.platform 提供服务器平台信息：

```
console.log(process.execPath);
console.log(process.version);
console.log(process.platform);
```

在我的机器上运行代码，返回如下信息：

```
/usr/local/bin/node
v0.6.9
linux
```

process 对象对一些标准输入输出流也进行了封装，包括标准输入 stdin，标准输出 stdout 和标准错误输出 stderr。stdin 和 stdout 支持异步操作，前者可读，后者可写。然而，要注意的是 stderr 是一个同步可阻塞流。

为了演示如何从 stdin 和 stdout 读取和写入数据，示例 3-1 中的 Node 应用程序监听标准输入的数据，然后再将数据复制到标准输出。stdin 流默认情况下是不允许直接操作的，所以在发送数据之前，我们必须先调用 resume。

示例 3-1　使用 stdin 和 stdout 来读取和写入数据

```
process.stdin.resume();

process.stdin.on('data', function (chunk) {
  process.stdout.write('data: ' + chunk);
});
```

使用 Node 运行该示例程序，然后在终端上用键盘输入信息。每次你输入信息并按回车键后，这些信息都会被回显。

process 对象中另一个常用的方法是 memoryUsage，通过它我们可以查询当前 Node 应用程序的内存使用量。这对于应用程序的性能调整是非常有用的，或许也能满足那些对程序有好奇心的人。调用该函数后的输出示例如下：

```
{ rss: 7450624,heapTotal: 2783520,heapUsed: 1375720}
```

其中 heapTotal 和 heapUsed 属性指示了 V8 引擎的内存使用情况。

最后，我还想提及下 process 对象的 nextTick 方法。这个方法可以将一个回调函数挂载到 Node 程序的事件循环机制中，并在下一个事件循环发生时调用该函数。

如果由于某种原因，你想延迟并且异步的执行某个函数调用，那么你可以使用 process.nextTick。一个很好的例子是，你需要创建一个新的函数，该函数有一个回

调函数作为其参数，而且你期望该回调函数能真正的被异步执行。下面的代码是一个演示：

```
function asynchFunction = function (data, callback) {
   process.nextTick(function() {
      callback(val);
   });
);
```

如果我们直接调用回调函数，行为将是同步的。而在上面代码中，对回调函数的调用会被延迟到下一个事件循环，而不是被立即调用。

虽然你也可以使用 setTimeout 方法并传入一个（0）毫秒的延迟来达到同样的目的，而不采用 process.nextTick 方法，例如：

```
setTimeout(function() {
   callback(val);
}, 0);
```

然而，setTimeout 方法并不像 process.nextTick 方法那样高效。当对它们进行对比测试时，process.nextTick 的调用速度远远快于使用 setTimeout 方法。另外，当你的应用程序需要执行一些复杂的计算或者其他费时的操作时，你也可以考虑使用 process.nextTick 方法。首先你需要将耗时的处理过程打散并分解成多个部分，每个部分分别通过 process.nextTick 调用，最终使得应用程序可以对其他请求进行处理，而无需等待耗时计算过程完成。

当然，与此相反的是，你不能打散一个必须按序执行的处理过程，否则你最终可能会得到无法预料的处理结果。

3.1.3 Buffer

Buffer 是 Node 中的另一个全局对象，是用于处理二进制数据的一种方式。在本章后半部分（参考 3.3 节），我们将会了解这样一个事实：即流处理往往采用的是二进制数据，而非字符串。为了将二进制数据转换为字符串，需要调用流套接字的 setEncoding 函数来改变当前使用的数据编码方式。

作为示例，你可以使用如下代码来创建一个新的 buffer 对象：

```
var buf = new Buffer(string);
```

若需要将一个字符串保存在 buffer 中时，你可以通过传入第二个可选参数来设置对该字符串的编码方式。支持的编码方式包括：

ascii

七位 ASCII。

utf8

多字节编码的 Unicode 字符。

usc2

两字节，little endian 方式编码的 Unicode 字符。

base64

Base64 编码。

hex

每个字节编码为两个十六进制字符。

你也可以将字符串写入到一个现有的 buffer 对象中，并指定该写入操作的偏移，数据长度和编码方式：

```
buf.write(string); //写入的默认偏移为 0，默认数据长度为
                    buffer.length - offset，编码默认采用 utf8
```

套接字接口之间传递数据时，会默认将数据包装在 buffer 对象中（采用二进制数据格式）。如果想要发送字符串数据，你可以直接调用套接字接口上的 setEncoding 函数，或者在将数据写入套接字接口时，使用参数指定编码方式。默认情况下，TCP（传输控制协议）的 socket.write 方法会将第二个参数设为 utf8，但对于通过 TCP 服务端中 connectionListener 回调函数返回的套接字，其发送的数据是一个 buffer 对象，而非字符串。

3.2 定时器：setTimeout、clearTimeout、setInterval 和 clearInterval

在客户端 JavaScript 编程中，定时器功能由全局窗口对象 windows 提供。虽然 JavaScript 本身并不提供定时器功能，但却在 JavaScript 相关的开发中被广泛使用，因而 Node 将其也纳入了核心 API 中。

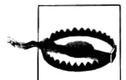
Node 对定时器功能所提供的操作与浏览器提供的操作相似。事实上，Node 与 Chrome 中的定时器函数接口完全一致，因为 Node 就是基于 Chrome 的 V8

JavaScript 引擎工作的。

调用 Node 的 setTimeout 函数时，需要传入一个回调函数作为第一个参数，第二个参数为延迟时间（以毫秒为单位），以及一个可选参数列表：

```
// timer to open file and read contents to HTTP response object
function on_OpenAndReadFile(filename, res) {

  console.log('opening ' + filename);
  // open and read in file contents
  fs.readFile(filename, 'utf8', function(err, data) {
     if (err)
         res.write('Could not find or open file for reading\n');
     else {
         res.write(data);
     }
  // reponse is done
  res.end();
}

setTimeout(openAndReadFile, 2000, filename, res);
```

在上述代码中，在 setTimeout 函数被执行大约 2000 毫秒后，回调函数 on_OpenAndReadFile 会被调用，打开一个文件并读取内容，然后将文件内容作为 HTTP 响应数据返回。



警告

正如 Node 文档所严谨标注的一样，无法保证回调函数能够在绝对准确的毫秒级延迟后被调用。在浏览器环境中使用 setTimeout 也存在同样问题，因为我们没有对运行环境的绝对控制权，许多因素都可能对定时器的计时准确度有略微影响。

函数 clearTimeout 可以清除通过 setTimeout 预设的定时器。如果你有重复计时的需求，可以使用 setInterval 函数来设置时间间隔，在每隔 n 毫秒（n 是传递给函数的第二个参数）后调用一个回调函数（第一个参数）。函数 clearInterval 可以用来清除时间间隔设置。

3.3 Servers、Streams 和 Sockets

许多 Node 核心 API 需要通过创建服务的方式来监听特定类型的通信。在第 1 章的示例中，我们使用 HTTP 模块创建了一个 HTTP Web 服务器。还有其他方法可以建立 TCP 服务器，TLS（传输层安全）服务器和 UDP（用户数据报协议）/数据报套接字。我会在第 15 章介绍 TLS，但在本节中，我要为大家介绍 Node 核心库对 TCP 和 UDP 的支持。不过首先还是需要对在本节中使用的术语做一个简要的介绍。

套接字（socket）是指通信端点；网络套接字（network socket）指的是工作在

同一个网络中，在两台不同计算机上运行的应用程序之间的通信端点。套接字之间传送的数据被称为流（stream）。我们可以通过一个 buffer 对象在流中传送二进制数据，或通过 Unicode 编码方式来传送一个字符串。两种类型的数据最终均会被包装为数据包（packets）进行传送，部分数据会被分拆成特定大小的包。套接字可以通过发送一种特殊的数据包 FIN（完成数据包）来表明本次传输已经完成。通信的可管理性以及流的可靠性，取决于创建的套接字类型。

3.3.1 TCP Sockets 和 Servers

我们可以使用 Node 中的 Net 模块来创建一个基本的 TCP 服务器和客户端。大多数的互联网应用都是基于 TCP 的，如 Web 服务和电子邮件，它是一种能在客户端和服务器套接字之间提供可靠传输数据的方式。

在第 1 章示例 1-1 中，我们创建了 HTTP 服务器并传入一个 requestListener 回调函数来处理用户请求。但创建 TCP 服务器有些许不同，传入的回调函数只有一个参数，这个参数是一个套接字对象的实例，代表连接到该 TCP 服务器的客户端连接。

在示例 3-2 的代码中，我们创建了一个 TCP 服务端。一旦与客户端相连接的服务端套接字创建好后，它就会持续监听两个事件：一个表示是否有来自客户端的数据需要接收，另一个表示客户端连接是否断开或关闭。

示例 3-2　一个简单的 TCP 服务器，并在端口 8124 上监听客户端连接请求

```
var net = require('net');

var server = net.createServer(function(conn) {
   console.log('connected');

   conn.on('data', function (data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);
      conn.write('Repeating: ' + data);
   });

   conn.on('close', function() {
        console.log('client closed connection');
   });

}).listen(8124);

console.log('listening on port 8124');
```

createServer 方法有一个可选参数 allowHalfOpen。如果将该参数设置为 true，那么当套接字从客户端接收到一个 FIN 包后，它不会发送另一个 FIN 作为回应。这样

做可使得套接字接口始终打开并且可写（当然不可读）。如果想关闭套接字，你需要明确地使用 end 方法。默认情况下，allowHalfOpen 是 false 的。

注意回调函数是如何通过 on 方法被绑定到两个不同的事件上的。在 Node 中，许多可以产生事件的对象都支持通过 on 方法来绑定事件监听器。此方法第一个参数为事件名称，第二个参数为事件处理函数。

提示

Node 中有一个较为特殊的对象 EventEmitter，所有继承自它的对象都会提供 on 方法，在本章的后面会继续讨论该对象。

创建 TCP 客户端与创建 TCP 服务端一样简单，如示例 3-3 中所示。在客户端套接字接口上调用 setEncoding 方法来改变对接收数据的编码处理方式。正如我们在“Buffer”一节所提到的，数据被当作 buffer 对象传送，但我们可以通过调用 setEncoding 方法，将收到的数据转换为 UTF8 字符串后处理它。套接字接口的 write 方法是用来发送数据的。代码中我们同样监听了两个事件：数据接收事件和服务端连接关闭事件，以便在服务端关闭时做一些收尾处理。

示例 3-3　使用 TCP 客户端套接字发送数据给 TCP 服务端

```
 var net = require('net');

var client = new net.Socket();
client.setEncoding('utf8');

// connect to server
client.connect ('8124','localhost', function () {
    console.log('connected to server');
    client.write('Who needs a browser to communicate?');
});

// prepare for input from terminal
process.stdin.resume();

// when receive data, send to server
process.stdin.on('data', function (data) {
   client.write(data);
});

// when receive data back, print to console
client.on('data',function(data) {
    console.log(data);
});

// when server closed
client.on('close',function() {
    console.log('connection is closed');
});
```

当我们在终端里面输入信息并按下回车键后，输入的内容将在服务端和客户端套接字接口之间传送。客户端应用程序会将你刚才输入的字符串发送给服务端，而服务端会将收到的内容输出到控制台。同时服务端会将同样的内容再次发送回客户端，这样客户端反过来又将内容输出到控制台。通过读取与客户端相连的套接字接口上的 remoteAddress 和 remotePort 属性，服务端还能将客户端使用的 IP 地址和端口打印出来。下面是客户端与服务端通信时的控制台输出示例（出于安全考虑 IP 地址已被修改）：

```
Hey, hey, hey, hey-now.
 from #ipaddress 57251
Don't be mean, we don't have to be mean.
 from #ipaddress 57251
Cuz remember, no matter where you go,
 from #ipaddress 57251
there you are.
 from #ipaddress 57251
```

客户端和服务端之间的 TCP 连接会一直保持，直到你通过使用 Ctrl+C 关闭任何一端。而未关闭的另外一端接会接收到 close 事件并在控制台输出关闭提示信息。由于所有操作都是异步进行的，服务端可以支持来自多个客户端的多个连接。

正如之前提到的，今天我们使用的大部分互联网应用功能，其底层传输机制都采用了 TCP，HTTP 就是一个例子，下一小节我们会对它有更多了解。

3.3.2 HTTP

在第 1 章中，我们曾经使用 HTTP 模块的 createServer 方法创建 HTTP 服务器，并传入一个回调函数作为 requestListener，以便异步处理收到的 HTTP 请求。

在网络体系结构中，TCP 是运输层而 HTTP 是应用层。如果留意下 Node 应用程序所包含的模块的话，你会看到，当创建一个 HTTP 服务器时，我们实际上已经从 net.Server 模块继承了很多功能，而 net.Server 则实现了对 TCP 的封装。

对于 HTTP 服务器来说，requestListener 就像是套接字，而 http.ServerRequest 对应于可读流，http.ServerResponse 对应于可写流。HTTP 之所以增加了另一个层面上的技术复杂性，是因为它需要支持分块传输编码。分块传输编码可以在响应数据未完全生成时进行数据传输，注意此时还无法确定响应信息的具体大小。如果分块中所包含信息的长度为 0，则表示响应信息的结束。当你需要将大型数据库的查询结果输出到一个 HTML 表格时，这种编码方式将非常有用，因为在你收到所有查询结果之前，就可以将数据写入响应信息进行输出了。

提示

更多关于流的说明会在“Stream、Pipes 和 Readline”一节中进行说明。

本章前面的 TCP 示例，以及第 1 章的 HTTP 示例，都是工作在网络套接字接口上的。然而，所有的 server/socket 模块除了可以连接并工作在特定的网络端口上，还支持连接到 Unix 套接字。与网络套接字不同，Unix 或 IPC（进程间通信）套接字支持主要用来支持同一系统内的进程间通信。

为了说明基于 Unix 套接字的通信，我重用了示例 1-3 中的代码，但并非绑定到一个网络端口，而是绑定到了一个 Unix 套接字上，如示例 3-4 代码所示。示例程序还使用到了 readFileSync 函数，它是一个同步版本的文件打开和内容读取的函数。

示例 3-4　基于 Unix 套接字的 HTTP 服务器

```
// create server
// and callback function
var http = require('http');
var fs = require('fs');

http.createServer(function (req, res) {


  var query = require('url').parse(req.url).query;
  console.log(query);
  file = require('querystring').parse(query).file;

  // content header
  res.writeHead(200, {'Content-Type': 'text/plain'});

  // increment global, write to client
  for (var i = 0; i<100; i++) {
    res.write(i + '\n');
  }

  // open and read in file contents
  var data = fs.readFileSync(file, 'utf8');
  res.write(data);
  res.end();
}).listen('/tmp/node-server-sock');
```

通过修改 Node.js 主站文档中提供的有关 http.request 对象的示例代码，我们得到了用于测试的客户端代码。注意 http.request 对象在默认情况下会使用套接字池 http.globalAgent。而该池默认可以保存最多五个套接字接口，不过可以通过改变 agent.maxSockets 值来调整它。

在示例 3-5 的代码中，客户端会接收从服务器返回的分块数据，并输出到终端。另外客户端还通过往请求中放入一些重复数据来触发服务器端的响应。

示例 3-5　连接到 Unix 套接字并输出接收的数据

```
var http = require('http');

var options = {
   method: 'GET',
   socketPath: '/tmp/node-server-sock',
   path: "/?file=main.txt"
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('chunk o\' data: ' + chunk);
  });
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// write data to request body
req.write('data\n');
req.write('data\n');
req.end();
```

这里没有使用异步文件读取功能，是因为异步文件读取函数可能会在连接关闭后才被调用，而无法输出文件内容到客户端。

在我们即将结束对 HTTP 模块的说明前，你应当留意到 Node HTTP 服务器并没有整合 Apache 或其他 Web 服务。举例来说，如果你希望网站有密码保护功能，你可以使用 Apache 来弹出一个窗口询问用户名和密码，但 Node HTTP 服务器不会。所以你将不得不自己编码来实现它。

提示

第 15 章介绍支持 SSL 版本的 HTTP 和 HTTPS，以及 Crypto 和 TLS/SSL。

3.3.3　UDP 数据报套接字

TCP 需要在通信的两个端点之间建立一个专用连接。而 UDP 是无连接协议，这意味着两个端点之间的连接是没有保证的。出于这个原因，相比 TCP 来说 UDP 是不可靠和不健全的。不过另一方面，UDP 比 TCP 要快，这使得它更适合于实时通信，比如应用在 VoIP（互联网语音传输协议）中；如果使用 TCP，由于必须建立连接从而会对信号质量产生不利影响。

Node 核心库支持这两种类型的套接字。在过去的几节中，我们对 TCP 功能进行了

说明。现在，我们来了解下 Node 对 UDP 的支持。

UDP 模块的标识符是 " dgram " ：

```
require ('dgram');
```

要创建一个 UDP 套接字，可以使用 createSocket 方法并传入套接字类型参数 udp4 或 udp6。你同样也可以传递一个回调函数用于监听事件。与使用 TCP 发送消息不同的是，使用 UDP 发送消息时必须使用 buffer，而不能是字符串。

示例 3-6 是一个 UDP 客户端示例代码。我们通过 process.stdin 得到输入数据，然后将其通过 UDP 套接字发送出去。请注意，我们没有设置字符串的编码方式，因为 UDP 套接字只接受 buffer 对象，而从 process.stdin 获得的数据已经被包装在一个 buffer 对象中。但我们必须调用 buffer 对象的 toString 方法将缓冲区的数据转换为一个字符串，以便我们将缓冲区的数据编码为可读信息并通过 console.log 方法输出到终端上。

示例 3-6　UDP 客户端，将输入到终端的信息通过 UDP 套接字发送出去

```
var dgram = require('dgram');

var client = dgram.createSocket("udp4");

// prepare for input from terminal
process.stdin.resume();

process.stdin.on('data', function (data) {
   console.log(data.toString('utf8'));
   client.send(data, 0, data.length, 8124, "examples.burningbird.net",
      function (err, bytes) {
        if (err)
          console.log('error: ' + err);
        else
          console.log('successful');
   });
});
```

示例 3-7 是 UDP 服务端代码，比客户端代码更简单。服务端应用程序创建了套接字，并将其与端口（8124）绑定，然后监听消息事件。当有新的数据到达时，应用程序使用 console.log 方法将其打印出来，一并打印输出的还有数据发送方的 IP 地址和端口号。特别要注意的是，在输出接收到的信息时，buffer 对象会自动将数据转换成一个字符串。

我们并非必须将 UDP 套接字绑定到某个端口上。如果没有指定绑定的端口，UDP 套接字将尝试监听所有端口。

示例 3-7　创建 UDP 服务端，绑定 8124 端口并接收数据

```
var dgram = require('dgram');

var server = dgram.createSocket("udp4");

server.on ("message", function(msg, rinfo) {
   console.log("Message: " + msg + " from " + rinfo.address + ":"
                 + rinfo.port);
});

server.bind(8124);
```

无论在 UDP 客户端还是服务端的代码中，当发送或者接收到数据后，我们并没有调用任何 close 方法来关闭套接字。因为在客户端和服务端之间并没有维护一个持续连接，我们仅仅是使用套接字提供的功能来发送和接收消息。

3.3.4　流、管道和 Readline

在前面几节所讨论的套接字之间的通信流实际上是底层抽象接口 stream 的一个实现。流可读、可写或可读写，而且所有流也都是 EventEmitter 的实例（即将在“Events 和 EventEmitter”一节讨论）。

实际上所有与通信相关的流，包括 process.stdin 和 process.stdout，它们都是抽象接口 stream 的具体实现。由于实现了这一基本接口，Node 中所有流都支持一套基本的功能调用：

- 你可以通过 setEncoding 方法更改流数据所使用的编码方式；
- 你可以检查当前流是否可读，是否可写，或着是否可读写；
- 你可以捕捉流事件，如接收到新数据或连接关闭，并能为每个事件附加回调函数；
- 你可以挂起和恢复流；
- 你可以使用 pipe 将一个可读流与一个可写流连接起来。

上面的最后一个 pipe 功能是我们之前未曾使用过的。我们可以打开一个 REPL 会话并键入以下内容来简单测试下这个功能：

```
> process.stdin.resume();
> process.stdin.pipe(process.stdout);
```

此时，你输入的任何信息将立即回显给你。

如果你想让输出流保持打开状态并接收连续输入的数据，可以在调用 pipe 方法时传入参数{ end: false }：

```
process.stdin.pipe(process.stdout, { end : false });
```

Node 中还为只读流提供了特定功能：readline。你可以使用如下代码来将 Readline 模块包含到你的程序中：

```
var readline = require('readline');
```

Readline 模块支持按行读取流。但是需要注意的是，一旦你在 Node 程序中包括这个模块并创建了接口来使用它，程序将不会终止，直到你关闭这个接口以及标准输入流。Node 主站文档中包含了一个如何开始和终止 Readline 接口的示例代码，我对其进行适当修改后得到了示例 3-8 的代码。当你运行这段示例代码时，它会提出一个问题，然后输出你输入的信息作为答案。它也可以监听其他任何"命令"，其实就是以换行符结尾的字符串。如果命令是".leave"，则退出程序；否则，将命令内容重复输出到终端。当然使用 CTRL+C 或 Ctrl+D 组合键可以终止应用程序。

示例 3-8　使用 Readline 库创建一个简单的命令驱动型用户界面

```
var readline = require('readline');

// create a new interface
var interface = readline.createInterface(process.stdin, process.stdout, null);

// ask question
interface.question(">>What is the meaning of life?  ", function(answer) {
   console.log("About the meaning of life, you said " + answer);
   interface.setPrompt(">>");
   interface.prompt();
});

// function to close interface
function closeInterface() {
   console.log('Leaving interface...');
   process.exit();
}
// listen for .leave
interface.on('line', function(cmd) {
   if (cmd.trim() == '.leave') {
      closeInterface();
      return;
   } else {
      console.log("repeating command: " + cmd);
   }
   interface.setPrompt(">>");
   interface.prompt();
});

interface.on('close', function() {
    closeInterface();
});
```

下面是执行结果示例：

```
>>What is the meaning of life?  ===
About the meaning of life, you said ===
>>This could be a command
repeating command: This could be a command
>>We could add eval in here and actually run this thing
repeating command: We could add eval in here and actually run this thing
>>And now you know where REPL comes from
repeating command: And now you know where REPL comes from
>>And that using rlwrap replaces this Readline functionality
repeating command: And that using rlwrap replaces this Readline functionality
>>Time to go
repeating command: Time to go
>>.leave
Leaving interface...
```

这应该很熟悉。还记得我们在第 2 章中用 rlwrap 来覆盖 REPL 命令行功能，并使用以下指令来使其生效：

```
env NODE_NO_READLINE=1 rlwrap node
```

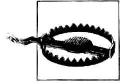
现在我们可以知道这句指令背后具体做了什么，其实就是让 REPL 使用 rlwrap 代替 Node 的 Readline 模块来进行命令行处理。

至此，对 Node 中的 stream 模块的简要介绍已经完成。是时候改变主题，去了解下 Node.js 中的子进程了。

3.4 子进程

操作系统提供给我们访问计算机资源的很多功能，其中很大一部分需要通过命令行方式来进行。如果能从 Node 应用程序中访问这些功能，就再好不过了。这也正是在 Node 中使用子进程的目的。

Node 允许我们在一个新的子进程中运行系统命令，并能监听并获取子进程的输入/输出信息。这包括能够将参数传递给该子进程中运行的命令，甚至能将一个命令的执行结果管道给另一个命令作为输入。接下来的几节会更详细地探讨这个功能。



警告

除了本节中的最后一段示例代码，所有其他示例代码都使用了 Unix 命令。它们可以工作在 Linux 系统上，应该也可以工作在 Mac 系统中。但是，它们不能工作在 Windows 命令窗口中。

3.4.1 child_process.spawn

有四种不同的技术来创建一个子进程。其中最常见的是使用 spawn 方法。这个方法会启动一个新的子进程，然后在该子进程中执行命令，同时还可以为需要执行的命令指定任何参数。在下面的示例中，我们将创建一个子进程并调用 Unix 的 pwd 命令来显示当前目录。该命令不带任何参数：

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd');

pwd.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

pwd.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

pwd.on('exit', function (code) {
  console.log('child process exited with code ' + code);
});
```

注意代码中对子进程的 stdout 和 stderr 相关事件的捕获。如果没有错误发生，任何命令的输出将被发送到子进程的 stdout，然后触发一个 data 事件。如果发生错误，比如我们传递了无效的命令参数：

```
var spawn = require('child_process').spawn,
    pwd = spawn('pwd', ['-g']);
```

错误信息将被发送到子进程的 stderr，然后被输出到控制台上显示：

```
stderr: pwd: invalid option -- 'g'
Try `pwd --help' for more information.
child process exited with code 1
```

子进程退出代码 1，表示发生了错误。注意退出代码会有所不同，这取决于操作系统以及具体的错误原因。当没有发生错误时，子进程退出代码为 0。

前面的代码描述了在何种情况下会输出何种信息给子进程的 stdout 和 stderr，但如何使用 stdin 呢？Node 主站文档中对如何将数据导入 stdin 有一段示例。它模拟了 Unix 的管道(|)功能，可以将一个命令的结果传递给另一个命令做为输入。我改编了该示例代码，以说明如何实现按文件名递归搜索文件，这也是我喜欢的 Unix 管道用法之一：

```
find . -ls | grep test
```

示例 3-9 通过子进程实现了这一功能。需要注意的是第一条命令用于执行查找，它有两个参数，而第二条命令仅有一个参数：即从 stdin 传递进来的搜索条件（文件名所包含的关键字）。还要注意，与 Node 文档示例代码不同的是，grep 子进程对 stdout 的编码方式需要通过 setEncoding 来修改。否则，数据只能被当作一个 buffer 对象输出。

示例 3-9 使用子进程实现对包含有关键词“test”的文件名的目录递归搜索

```
var spawn = require('child_process').spawn,
    find = spawn('find',['.','-ls']),
    grep = spawn('grep',['test']);

grep.stdout.setEncoding('utf8');

// direct results of find to grep
find.stdout.on('data', function(data) {
   grep.stdin.write(data);
});

// now run grep and output results
grep.stdout.on('data', function (data) {
  console.log(data);
});

// error handling for both
find.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});
grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});
// and exit handling for both
find.on('exit', function (code) {
  if (code !== 0) {
    console.log('find process exited with code ' + code);
  }

  // go ahead and end grep process
  grep.stdin.end();
});

grep.on('exit', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

当你运行应用程序后，你会得到当前目录以及所有子目录中包含“test”字符的文件名。

至今为止，所有的示例程序都可以工作在 Node 0.8 版本和 Node 0.6 版本。但由于 Node 不同版本之间底层 API 的变化，示例 3-9 是一个例外。

在 Node 0.6 版本中，只有子进程退出并且所有的 STDIO 管道关闭后，才能产生 exit 事件。而在 Node 0.8 版本中，当子进程结束后，exit 事件会立即产生。这会导致我们的应用程序崩溃，因为 grep 子进程会在 STDIO 管道关闭后试图处理其数据。如果希望应用程序能够工作在 Node 0.8 版本中，应用程序需要监听 find 子进程的 close 事件，而不是 exit 事件：

```
// and exit handling for both
find.on('close', function (code) {
  if (code !== 0) {
      console.log('find process exited with code ' + code);
  }
  // go ahead and end grep process
  grep.stdin.end();
});
```

在 Node 0.8 版本，close 事件会在子进程退出并且所有的 STDIO 管道关闭后产生。

3.4.2 child_process.exec 和 child_process.execFile

除了通过 spawn 来生成并执行一个子进程，你也可以使用 child_process.exec 和 child_process.execFile 来启动 shell 执行命令，同时命令执行结果可以被缓存。child_process.exec 和 child_process.execFile 之间的唯一区别在于 execFile 会执行指定的可执行文件，而不是运行一条命令。

这两种方法的第一个参数是需要执行的命令或文件路径，第二个是一个可选参数列表，第三个参数是一个回调函数，该回调函数有三个参数：error、stdout 和 stderr。如果没有错误发生，执行结果会保存到 stdout。

如果存在一个可执行文件，并包含如下内容：

```
#!/usr/local/bin/node
console.log(global);
```

以下代码会打印并输出被缓存的执行结果：

```
var execfile = require('child_process').execFile,
    child;
child = execfile('./app.js', function(error, stdout, stderr) {
  if (error == null) {
      console.log('stdout: ' + stdout);
  }
});
```

3.4.3 child_process.fork

最后一个子进程方法是 child_process.fork。其实是对 spawn 方法的封装，目的是为了启动子进程并运行 Node.js 模块。

不同于其他子进程创建方法，fork 方法会在父进程与子进程之间建立一个真实的通信管道。但是要注意，通过 fork 生成的每个子进程都需要一个全新的 V8 实例，这需要耗费更多时间和内存。

提示

Node 文档中对 fork 的使用提供了一些很好的示例。

3.4.4 在 Windows 系统中使用子进程

之前我们提到，调用 Unix 系统命令的子进程无法工作在 Windows 系统中，反之亦然。虽然这是显而易见的，但并不是每个人都能注意到它，不像运行在浏览器中的 JavaScript 程序，在不同操作系统环境中，Node 应用程序会表现出不同的行为。

直到最近，Windows 系统上的 Node 二进制安装包才提供了对子进程的访问支持。你需要通过 Windows 命令解释器 cmd.exe 来运行任何你想执行的命令。

示例 3-10 演示了如何执行一个 Windows 命令。在这段示例代码中，我们使用 Windows 的 cmd.exe 来执行 dir 命令（显示目录内容），并通过监听 data 事件来获取命令执行结果，然后输出。

示例 3-10　在 Windows 系统中运行一个子进程

```
var cmd = require('child_process').spawn('cmd', ['/c', 'dir\n']);

cmd.stdout.on('data', function (data) {
    console.log('stdout: ' + data);
});

cmd.stderr.on('data', function (data) {
    console.log('stderr: ' + data);
});

cmd.on('exit', function (code) {
    console.log('child process exited with code ' + code);
});
```

传递给 cmd.exe 的第一个参数是‘/c’，它用来指示在 cmd.exe 执行完命令后就立即终止并退出。如果没有这个参数，这段 Node 程序将不能正常工作。当然，我们更不想传送参数‘/k’给 cmd.exe，因为它会告诉 cmd.exe 执行命令并保持，这样你的应用程序将不会终止。

提示

我们会在第 9 章和第 12 章对子进程有更详细的说明。

3.5 域名解析和 URL 处理

DNS 模块使用了库 c-ares 来提供 DNS 解析功能，c-ares 是一个采用 C 语言开发的库，可以提供异步方式的 DNS 请求和解析。对于需要处理域名或 IP 地址的应用程序来说，Node 中 DNS 模块和其他一些相关模块是非常有用的。

使用 dns.lookup 方法可以解析并得到一个域名的 IP 地址，在下面的代码中，我们会将解析到的 IP 地址输出到终端：

```
var dns = require('dns');
dns.lookup('burningbird.net',function(err,ip) {
    if (err) throw err;
    console.log(ip);
});
```

使用 dns.reverse 方法则会根据给定的 IP 地址返回一组域名：

```
dns.reverse('173.255.206.103', function(err,domains) {
domains.forEach(function(domain) {
  console.log(domain);
  });
});
```

dns.resolve 方法会根据指定的记录类型返回一组记录，常见的记录类型有 A、MX、NS 等。下面的代码描述了如何查找出解析域名“burningbird.net”所用的域名服务器：

```
var dns = require('dns');
dns.resolve('burningbird.net', 'NS', function(err,domains) {
domains.forEach(function(domain) {
  console.log(domain);
  });
});
```

返回的结果是：

```
ns1.linode.com
ns3.linode.com
ns5.linode.com
ns4.linode.com
```

在第 1 章示例 1-3 中，我们使用了 URL 模块。这个简单的模块可以提供 URL 解析功能，并能返回一个对象来描述 URL 中的所有信息。如下代码所示：

```
var url = require('url');
var urlObj =url.parse('http://examples.burningbird.net:8124/?file=main');
```

返回值为一个 JavaScript 对象：

```
{ protocol: 'http:',
  slashes: true,
  host: 'examples.burningbird.net:8124',
  port: '8124',
  hostname: 'examples.burningbird.net',
  href: 'http://examples.burningbird.net:8124/?file=main',
  search:'?file=main',
  query: 'file=main',
  pathname: '/',
  path: '/?file=main' }
```

对象中的每个属性可以被分别访问，像这样：

```
var qs = urlObj.query; // 获得查询字符串
```

调用 URL.format 方法可以执行相反的操作：

```
console.log(url.format(urlObj)); // 返回原始 URL
```

URL 模块常常与 Query String 模块一并使用。后者是一个简单的工具模块，它提供对查询字符串的解析功能，也可以被用来生成查询字符串。

可以使用 querystring.parse 方法取得查询字符串中的键值对。如下所示：

```
var vals = querystring.parse('file=main&file=secondary&type=html");
```

查询结果同样保存在一个 JavaScript 对象中，可以轻松简单地对每个键值进行单独访问：

```
{ file: [ 'main', 'secondary' ], type: 'html' }
```

由于 file 在查询字符串中出现了两次，所以其对应的两个值被保存到了一个数组中，每一个都可以被单独访问：

```
console.log(vals.file[0]); // returns main
```

你同样也可以将一个保存有键值对的对象转换为查询字符串，使用 querystring.stringify 方法即可：

```
var qryString = querystring.stringify(vals)
```

3.6 Utilities 模块和对象继承

Utilities 模块提供了一些非常实用的功能。你可以通过'util'来包含这个模块：

```
var util = require('util');
```

你可以使用 Utilities 模块来测试一个对象是否是数组（util.isArray）或正则表达式（util.isRegExp），或者将其格式化成一个字符串（util.format）。最近还增加了一个新的实验性功能，用于将可读流的数据输出到可写流（util.pump）：

```
util.pump(process.stdin, process.stdout);
```

不过，我不想在 REPL 中运行此代码，因为这会将你键入的任何字符回显出来，从而使得当前的 REPL 显得混乱。

我经常使用 util.inspect 来获得一个对象的描述信息。当你想了解某个对象的更多信息时，这是一个很好的方法。inspect 方法的第一参数是必选的，传入你想查看的对象即可；第二个是可选参数，决定了是否需要查看对象的中不可枚举（nonenumerable）属性；第三个可选参数指定了递归次数，即所能查看的对象信息的深度；第四个参数也是可选的，指定是否使用 ANSI 颜色输出信息。如果第三个参数是一个空值 null，则表示无限递归（默认为 2），该方法会尽可能详尽的检查对象。从以往的经验来看，我们还是需要小心使用空深度，因为你可能获得大量的输出信息。

你可以在 REPL 中练习使用 util.inspect，不过我推荐使用下面这个简单的小程序：

```
var util = require('util');
var jsdom = require('jsdom');

console.log(util.inspect(jsdom,true,null,true));
```

当你运行它时，请将输出重定向到一个文件：

```
node inspectjsdom.js > jsdom.txt
```

现在，你可以随时的查阅这个对象接口。再次提醒，如果你使用空深度，可能会得到非常大的输出文件。

Utilities 模块还提供了另外一些方法，util.inherits 是比较常用的一个。它需要两个参数，constructor 和 superConstructor。方法的执行结果是 constructor 将继承 superConstructor 的功能。

示例 3-11 展示了在使用 util.inherits 方法时所需要留意的一些细节问题。示例代码后面是关于代码的具体说明。

提示

示例 3-11 及其说明包括了一些你可能已经很熟悉的 JavaScript 特性。但我们的目标是要使得所有读完本小节的读者对于所发生的事情有着同样的理解。

示例 3-11　使用 util.inherits 方法实现继承

```
var util = require('util');

// define original object
function first() {
  var self = this;
  this.name = 'first';
  this.test = function() {
    console.log(self.name);
   };
}

first.prototype.output = function() {
   console.log(this.name);
}

// inherit from first
function second() {
  second.super_.call(this);
  this.name = 'second';
}
util.inherits(second,first);

var two = new second();


function third(func) {
   this.name = 'third';
   this.callMethod = func;
}

var three = new third(two.test);

// all three should output "second"
two.output();
two.test();
three.callMethod();
```

在示例程序中，我们创建了三个对象，分别命名为 first、second 和 third。

第一个对象包含有两个方法:test 和 output。test 方法直接定义在对象中，而 output 方法是通过原型后添加进来的。之所以使用两种方式来定义同一个对象上的两个方法，是为了能更好的说明在使用 util.inherits 进行对象继承时需要留意的技术细节。

第二个对象的定义中包含如下代码：

```
second.super_.call(this);
```

如果我们将这一行代码从第二个对象的构造函数中删除，那么我们依然可以成功的调用 second 对象中的 output 方法，但如果尝试调用 test 方法，则应用程序会产生

一个错误并且终止，同时获得一个错误信息，提示 test 方法未定义。

由于 second 对象继承了 first 对象，这句代码中的 call 方法能将两个对象的构造函数关联起来，以确保在调用 second 对象的构造函数时，first 对象的构造函数也同时能够被调用。

我们之所以需要在 second 构造函数中调用 first 的构造函数，因为在 first 的构造函数被调用前 test 方法是不存在的。然而，对于 output 方法我们并不需要这么做，因为它是在 first 原型上直接定义的。当 second 继承了 first 的属性时，它同时也继承这个方法。

在 util.inherits 的实现中，我们可以看到有关 super_的定义：

```
exports.inherits = function(ctor, superCtor) {
  ctor.super_ = superCtor;
  ctor.prototype = Object.create(superCtor.prototype, {
    constructor: {
      value: ctor,
      enumerable: false,
      writable: true,
      configurable: true
    }
  });
};
```

调用 util.inherits 方法时，super_会作为一个新的属性被添加到 second 对象上：

```
util.inherits(second,first);
```

在示例应用程序中还使用到了第三个对象 third，它同样也定义了 name 属性。third 并没有继承自 first 或 second，但却需要在其实例化时传入一个函数作为参数。被传入的函数会在 callMethod 方法中调用。在示例代码中，我们将 second 对象的 test 方法传递给了 third 对象的构造函数来实例化它：

```
var three = new third(two.test);
```

乍看之下，我们可能会以为在 three.callMethod 方法被调用时会输出“third”，但实际上输出的是“second”。这也正是我们在 first 对象中使用 self 引用的原因。

在 JavaScript 中，this 用于描述一个对象的上下文，当一个函数被作为参数传递时（比如传递给事件处理程序），this 将会发生切换。如果期望此函数仍然使用所属对象的数据的话，唯一方法是将 this 赋值给一个对象内变量（self），然后在函数定义中使用该对象变量。

运行这个应用程序后得到的输出结果如下：

```
second
second
second
```

对于客户端 JavaScript 开发人员来说，这些知识点应该比较熟悉，而且有助于正确理解 Utilities 模块的继承功能。接下来的一节中我们会介绍 Node 中的 EventEmitter，这项功能在很大程度上依赖于本节所描述的继承行为。

3.7 Events 和 EventEmitter

Node 核心库中许多对象的背后实现中都使用到了 EventEmitter。对于可以产生事件并能通过 on 方法绑定事件处理函数的对象来说，几乎无一例外都是通过继承 EventEmitter 来实现的。学习使用 Node 开发时，了解 EventEmitter 的工作原理以及如何使用它是非常重要的。

EventEmitter 是 Node 提供的可以为其他对象提供异步事件处理功能的对象。为了说明其核心功能，我们将快速实现一个测试程序。

首先，包含 Events 模块：

```
var events = require('events');
```

接下来，创建一个 EventEmitter 的实例：

```
var em = new events.EventEmitter();
```

使用新创建的 EventEmitter 实例来完成两个基本任务：为指定事件添加事件处理程序，激发并产生事件。当一个特定的事件产生时，名为 on 的事件处理函数将被触发。该方法的第一个参数是事件名称，第二个参数为事件处理函数：

```
em.on('someevent', function(data) { ... });
```

当某些条件满足之后，我们可以通过 emit 方法来激发对象上的事件：

```
if (somecriteria) {
   en.emit('data');
}
```

在示例 3-12 中，我们创建了一个 EventEmitter 的实例用于产生事件 timed，该事件每隔三秒被触发一次。而在对应该事件的处理函数中，我们会将一个计数器信息输出到控制台。

示例 3-12 EventEmitter 基本功能示例

```
var eventEmitter = require('events').EventEmitter;
var counter = 0;

var em = new eventEmitter();

setInterval(function() { em.emit('timed', counter++); }, 3000);

em.on('timed', function(data) {
  console.log('timed ' + data);
});
```

运行示例程序后，计数信息被定时输出到控制台，直到应用程序被终止。

这是一个有趣的示例。但是，我们需要的往往是将 EventEmitter 功能添加到代码的现有对象中，而不仅仅是在整个应用中直接使用 EventEmitter 的实例。

为了实现这一目的，我们可以使用上一节提到的 util.inherits 方法：

```
util.inherits(someobj, EventEmitter);
```

在对象上使用 util.inherits 方法后，你就可以在定义该对象的方法时调用 emit 方法来激发事件，同时也可以为该对象的实例编写事件处理函数了：

```
someobj.prototype.somemethod = function() { this.emit('event'); };
...
someobjinstance.on('event', function() { });
```

在下面示例 3-13 的代码中，说明了如何通过继承的方法为一个对象添加 EventEmitter 所提供的功能，它能从更加实用的角度让我们理解 EventEmitter 是如何工作的。在这个示例中，我们创建了一个新对象 inputChecker。该对象的构造函数需要两个参数：第一个代表人名，用来传值给对象变量 name；第二个是一个文件名，被用来生成可写流对象 writeStream。代码中我们通过调用 Node 文件系统模块中的 createWriteStream 方法来创建一个可写流（在下面的说明栏中，我们对文件系统中的可读流和可写流有更多介绍）。

> **可读写流**
>
> 使用 Node 的文件系统模块（`fs`）可以打开文件并进行读写操作，或者监视指定文件是否有新的活动，还可以对文件系统的目录结构进行维护。同时它还为我们提供可读流和可写流来操作文件内容。
>
> 你可以使用 `fs.createReadStream` 方法并传入文件名称、文件路径或其他可选项来创建一个可读流。也可以使用 `fs. createWriteStream` 方法并传入文件名称和路径来创建一个可写流。
>
> 如果你期望通过事件驱动方式来操作文件，并且需要频繁读写文件内容时，使用可读写流是一个好的选择。程序后台会打开流并将所有读写操作放入队列然后按序进行处理。

check 是另一个对象方法，用于从输入数据中解析特定命令信息。命令“wr:”会触发

write 事件，命令“en:”是一个 end 事件。当无法解析出命令信息时，则触发“echo”事件。代码中的对象实例为这三种事件类型均提供了处理函数。对 write 事件的处理是将“wr:”命令之后的信息写入到可写流 writeStream 中，对 echo 事件的处理是将收到的信息回显输出到终端，对 end 事件的处理是使用 process.exit 方法来结束并退出程序。所有输入都来自标准输入（process.stdin）。

示例 3-13　通过继承 EventEmitter 创建支持事件功能的对象

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

function inputChecker (name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./' + file + '.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});
};

util.inherits(inputChecker,eventEmitter);

inputChecker.prototype.check = function check(input) {
  var command = input.toString().trim().substr(0,3);
  if (command == 'wr:') {
     this.emit('write',input.substr(3,input.length));
  } else if (command == 'en:') {
     this.emit('end');
  } else {
     this.emit('echo',input);
  }
};
// testing new object and event handling
var ic = new inputChecker('Shelley','output');

ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.on('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
    ic.check(input);
});
```

示例代码中，与 EventEmitter 相关的代码均用粗体标注。值得注意的是

process.stdin.on 也用粗体标注了出来，这是因为 process.stdin 也是 Node 中众多继承了 EventEmitter 功能的对象之一。

前面章节我们介绍过 util.inherits 方法，同时在示例 3-13 中我们并没有在对象的构造函数中调用 EventEmitter 构造函数，这是因为我们需要的 on 和 emit 方法定义在 EventEmitter 对象的原型中，而并没有定义在实例属性中。on 方法就像是 EventEmitter.addListener 方法的别名，它们有着同样的参数，因此：

```
ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

与下面这两句代码是完全一致的：

```
ic.on('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});
```

另外你可以只在某个事件第一次被触发时调用事件处理函数：

```
ic.once(event, function);
```

默认情况下，如果一个事件挂载了多于 10 个处理函数（也就是说有 10 个 listeners），那么你会得到一条警告信息。不过通过调用 setMaxListeners 方法并传入一个数字，我们可以轻松修改这个限制，注意使用 0 则表示无限制。

Node 中的许多对象，包括一些第三方的模块都使用到了 EventEmitter。在第 4 章中，我会描述如何将示例 3-13 的代码转换成一个模块。

第4章 Node 模块系统

开发人员在实现 Node 的基本功能的时候尽量进行了简化，通过 Node 模块提供其他扩展的功能，而不是将每个可能用到的组件都直接集成在 Node 中。

Node 模块系统（Node Module System）以 CommonJS 模块系统为模式。CommonJS 模块可以创建互相兼容的模块。Node 模块系统的关键在于向开发人员保证了他们的模块可以和其他模块一起工作。

Node 模块需要实现 CommonJS 模块系统的以下需求：

1. 支持 require 方法，接收模块标识作为参数返回可用的 API；
2. 模块名称是字符串，可能包含斜杠（指代路径）；
3. 模块必须明确指出需要对外暴露的接口；
4. 模块的变量都是私有的。

在之后几个章节中，我们会了解 Node 是如何实现这些需求的。

4.1 使用 require 和默认路径加载模块

Node 支持简单的模块加载系统：文件和模块间具有一一对应的关系。

在 Node 应用中引入模块，需要使用 require 语句，传入参数为代表模块标识符的字符串：

```
var http = require ('http');
```

还可以引入模块中的特定对象，而不是整个模块：

```
var spawn = require ('child_process').spawn;
```

你可以用模块标识符加载 Node 原生模块或者 node_modules 文件夹中的模块，比如字符串 http 代表 HTTP 模块。对于非 Node 原生支持和不在 node_modules 文件夹里的模块，需要在字符串之前加上相对路径（斜杠“/”表示）。例如，Node 想引入名为 mymodule.js 的模块，mymodule.js 与 Node 应用在同一目录下，require 语句为：

```
require ('./mymodule.js');
```

或者可以使用绝对路径：

```
require ('/home/myname/myapp/mymodule.js');
```

模块文件的扩展名可以为.js，.node 或者.json。.node 扩展名表示编译好的二进制文件，而不是包含 JavaScript 的文本文件。

Node 核心模块的优先级要高于外部模块。当你尝试加载一个自定义的 http 模块，Node 会首先加载内部核心 HTTP 模块，除非你换个模块名字或者提供绝对路径。

之前我曾提到 node_modules 文件夹。如果提供的模块名没有包含路径信息，或者该模块不是核心模块，Node 首先根据当前工程在 node_modules 文件夹中查找。如果在该文件夹中没找到该模块，就会到该文件夹的父目录进行查找，以此类推。

如果模块名为 mymodule，应用程序在/home/myname/myprojects/myapp 的子目录中。Node 会依次按照以下次序进行查找：

- /home/myname/myprojects/myapp/node_modules/mymodule.js
- /home/myname/myprojects/node_modules/mymodule.js
- /home/myname/node_modules/mymodule.js
- /node_modules/mymodule.js

Node 可以根据文件在哪里声明 require 语句来优化查找。比如，如果调用 require 语句的文件本身就是 node_modules 文件夹某个子目录中的一个模块，Node 就会从最高层的 node_modules 文件夹中开始搜索被引用的模块。

require 还有其他两种形式：require.resolve 和 require.cache。require.resolve 方法负责查找给定的模块但是并不加载该模块，只返回文件名。require.cache 对象包含所有加载模块的缓存版本。当你在相同语境中再次加载同一模块时，Node 会选择从 cache 中加载该模块来优化性能。如果需要强制重新加载某个 cache 中的模块，先

从 cache 中删除该模块然后重新加载。

如果引用某块路径为：

```
var circle=require('./circle.js');
```

删除模块命令：

```
delete require.cache('./circle.js');
```

该命令使得下次调用 require 的时候会重新加载该模块。

4.2 外部模块和 Node 包管理工具

正如之前提到过的，Node 庞大的功能库很多都是由第三方模块提供的。比如路由模块、与文档数据库系统交互的模块、模版模块、测试模块，还有支付网关相关的模块。

尽管没有官方的 Node 模块开发系统，开发人员们还是很热衷于将自己的模块上传到 github 上。以下是一些常用的查找 Node 模块的软件源：

- npm registry (http://search.npmjs.org/)
- Node module wiki(https://github.com/joyent/node/wiki/modules)
- The node-toolbox(http://toolbox.no.de/)
- Nipster!(http://eirikb.github.com/nipster/)

模块被大致分门别类为不同的类型，比如之前提及的路由模块、数据库、模版、支付网关等。

你需要从 GitHub（或者其他源）下载模块源码，安装到你的应用环境，然后才可以使用该模块。绝大部分模块提供了基本的安装说明，至少也可以从模块的文件和目录中找到安装方法。然而，更简单的安装模块的方法是：使用 Node 包管理工具（Node Package Manager， 简称 npm，后文都使用 npm 表示）。

提示

npm 官网为：http://npmjs.org/。可以在 http://npmjs.org/doc/README.html 找到 npm 的简介。对 Node 模块开发人员来说，深入理解 Node 的内容在 npm 手册的开发者章节:http://npmjs.org/doc/developers.html。关于解释本地和全局安装的差异，参考：http://blog.nodejs.org/2011/03/23/npm-1-0-global-vs-local-installation/。

安装好 npm 后，在与访问 Node 相同的环境中，可以在命令行输入 npm 来确保 npm 是否安装成功。

用以下命令可以查看 npm 全部命令列表：

```
$npm help npm
```

安装模块时可以选择局部或者全局安装模块。如果你工作的项目并不是共用该系统的每个人都需要访问该模块，那么局部安装是最好的实现方式。默认安装方式为局部安装，目录为 node_modules。

```
$npm install modulename
```

例如，安装一个非常流行的中间件架构 Connect：

```
$npm install connect
```

npm 不只安装 Connect 本身，并且也安装 Connect 依赖的模块，如图 4-1 所示。

一旦安装完毕，就可以在本地 node_modules 目录中找到该模块。相关的依赖都被安装在该模块的 node_modules 目录中。

如果想要全局安装，使用-g 或者--global 选项：

```
$npm-g install connect
```

以上例子安装的模块都在 npm 观望中。同样也可以安装本地文件系统中的模块，或者来自本地或者 url 得到的压缩文件：

```
npm install http://somecompany.com/somemodule.tgz
```

如果安装包有版本号需要指定具体的版本：

```
npm install modulename@0.1
```

提示

npm 也可以和 Git 一起使用，附录中有说明。

还可以通过这种方式安装我们非常熟悉的包，jQuery：

```
npm install jquery
```

现在在 Node 应用开发中你可以使用熟悉的$语法了。

如果你不再需要一个模块，可以卸载：

```
npm uninstall modulename
```

下一行命令告知 npm 检查新的模块，如果存在的话对其进行更新：

```
npm update
```

也可以更新单个模块：

```
npm update modulename
```

如果只是希望查看是否有过期的包，命令为：

```
npm outdated
```

同样这个命令也可以对单一模块执行。

显示安装的包和依赖命令为：list，ls，la 或者 ll：

```
npm ls
```

la 和 ll 选项提供了更多的描述。以下内容是我在 Windows 7 机器上运行 npmll 的结果：

```
C:\Users\Shelley>npm ls ll
npm WARN jsdom >= 0.2.0 Unmet dependency in C:\Users\Shelley\node_modules\html5
C:\Users\Shelley
├── async@0.1.15
├── colors@0.6.0-1
├── commander@0.5.2
├─┬ connect@1.8.5
│ ├── formidable@1.0.8
│ ├── mime@1.2.4
│ └── qs@0.4.1
├─┬ html5@v0.3.5
│ ├── UNMET DEPENDENCY jsdom >= 0.2.0
│ ├── opts@1.2.2
│ └─┬ tap@0.0.13
│   ├── inherits@1.0.0
│   ├── tap-assert@0.0.10
│   ├── tap-consumer@0.0.1
│   ├── tap-global-harness@0.0.1
│   ├── tap-harness@0.0.3
│   ├── tap-producer@0.0.1
│   ├── tap-results@0.0.2
│   ├─┬ tap-runner@0.0.7
│   │ ├── inherits@1.0.0
│   │ ├── slide@1.1.3
│   │ ├── tap-assert@0.0.10
│   │ ├── tap-consumer@0.0.1
│   │ ├── tap-producer@0.0.1
│   │ ├── tap-results@0.0.2
│   │ └── yamlish@0.0.3
│   ├── tap-test@0.0.2
│   └── yamlish@0.0.2
└─┬ optimist@0.3.1
  └── wordwrap@0.0.2
```

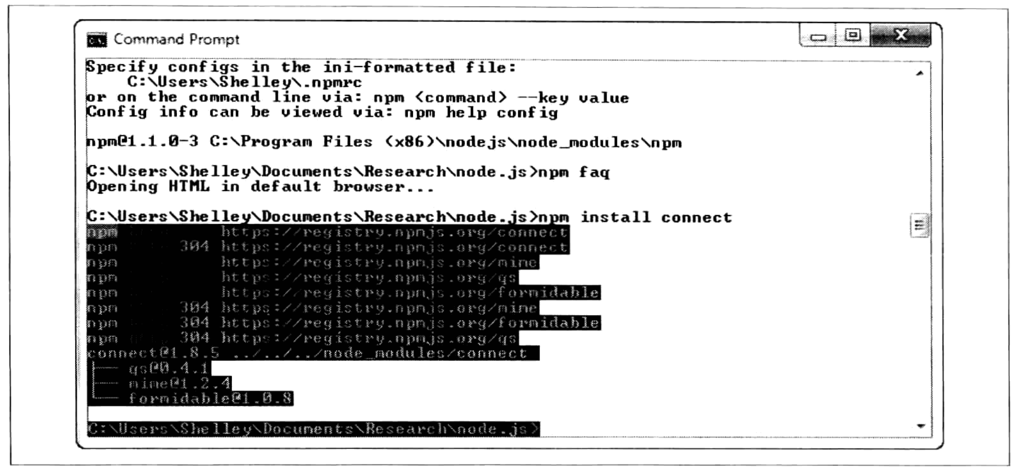


图 4-1 在 Windows7 系统中通过 npm 安装 Connect

注意到关于 HTML5 模块的一个 unmet dependency（未解决的依赖）。HTML5 模块需要一个较早版本的 JSDOM 库。修复这一问题，需手动安装该模块需要的版本：

```
npm install jsdom@0.2.0
```

也可以直接用-d 标识安装所有的依赖。比如，在模块的目录中输入：

```
npm install -d
```

如果你需要安装的模块版本还没有上传到 npm registry，可以直接从 Git 目录安装：

```
npm install https://github.com/visionmedia/express/tarball/master
```

关于这一点务必要格外谨慎。因为我发现当你安装一个该版本尚未发布的模块，又使用了 npm update 进行升级后，npm registry 版本会覆盖当前你使用的版本。

查看当前全局安装的模块，命令是：

```
npm ls -g
```

你可以用 config 命令学习更多关于 npm 的安装。以下命令显示 npm 的配置设置：

```
npm config list
```

更深入的了解配置的设置，使用：

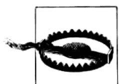
```
npm config ls -l
```

可以用命令行来修改或者删除配置项：

```
npm config delete keyname
npm condig set keyname value
```

或者直接编辑配置文件：

```
$ npm config edit
```



警告

我强烈建议不要修改 npm 的配置，除非你非常了解该修改可能带来的所有影响。

搜索模块时，可以挑选合适的关键字，从而返回最符合你需求的搜索结果：

```
npm search html5 parser
```

当你第一次搜索时，npm 会建立一个索引，这需要花费几分钟的时间。当搜索结束时，你会得到一系列符合搜索条件或者关键字的模块列表。html5 和 parser 只返回了两个模块：HTML5 和支持 SVG、MathML 的 HTML parser，支持 HTML Canvas、SVG-to-Canvas parser 的对象模块 Fabric。

npm 官网提供了可供浏览的模块注册表，还有当前引用最多的模块列表显示被 Node 应用模块引用最多的模块。在下一节中，会涉及部分这类模块。

提示

我会在本章稍后 4.4 节介绍其他 npm 的命令。

4.3 如何找到你需要的模块

尽管 Node.js 是近几年才流行起来的，但是已经拥有了大量用户。当你浏览 Node.js 的 wiki 页面时，会发现非常多的模块。好处在于你可以找到很多有用的模块来实现你需要的功能，但是随之而来的坏处在于很难决定需要使用哪个模块，换句话说，很难决定哪个模块是“最佳选项”。

哪个模块最受欢迎，使用类似 Google 等搜索工具可以提供一个相对公平的观点。比如，当我搜索中间件和架构模块的时候很明显 Connect 和 Express 是最受欢迎的。

还有，当你在 GitHub 注册表中查找某个模块时，可以看到该模块是否拥有大量支持者、是否更新及时且兼容当前 Node 安装包。作为另一个例子，我查看了 Apricot。

Apricot 是一个 HTML 解析的工具，在 Node 文档中也有推荐。但是我注意到这个模块已经很久没有更新过了，当尝试使用该模块的时候，发现它无法与我的 Node 兼容（至少，在写这本书的时候）。

提示

很多模块都提供了应用范例，以及一系列能够快速让你知道该模块能否在你的环境中工作的测试。

正如之前提到的，Node 官方文档提供了推荐的第三方模块的列表，以 npm 为例，虽然 npm 现在已经集成在 Node 安装包中了。但是，npm 网站和它的模块注册表提供了更好的服务，显示当前大部分应用中使用的模块。

在 npm 注册页面，可以搜索模块，也可以查看“最常用依赖”模块的列表，包括在其他模块中引用或者 Node 应用中使用的模块。在写这本书的时候，排名靠前的模块为：

Underscore

提供普遍使用的 JavaScript 函数。

Coffee-script

可以使用 CoffeeScript。CoffeeScript 是一种可以编译为 JavaScript 的语言。

Request

简化的 HTTP 请求客户端。

Express

一种架构。

Optimist

提供轻量级的选项解析。

Async

提供方法和模式同步代码。

Connect

中间件。

Colors

为控制台添加颜色。

Uglify-js

解析器和压缩器/或者美化工具。

Socket.IO

客户端/服务器通信。

Redis

Redis 客户端。

Jade

一个模版引擎。

Commander

命令行程序使用。

Mime

提供对文件扩展名的支持和 MIME 映射。

JSDOM

实现 W3C DOM。

在以后的章节中会涉及这些模块，但是现在介绍三个——因为这三个模块不仅可以帮助我们更好地理解 Node 工作原理，而且模块本身也非常有用：

- Colors
- Optimist
- Underscore

4.3.1 Colors：简单至上

Colors 是一个很简单的模块，可以给 console.log 输出提供不同的颜色以及风格，这基本是它全部的功能。但是，它也很好地说明了什么样是一个高效的模块：使用简

单，提供单一的功能并且完成得很完美。

测试模块是否正常工作是使用 REPL 的一个重要原因。为了测试 Colors，用 npm 安装：

```
$ npm install colors
```

新建一个 REPL 会话，引入 colors 库：

```
>var colors = require('colors');
```

因为 Colors 模块包含在当前路径的 node_modules 目录中，因此 Node 加载速度非常快。

现在进行测试，例如：

```
console.log('This Node kicks it! '.rainbow.underline);
```

输出信息是彩色的，并且带下划线。样式只对一个信息有效，你需要对另一个信息重新添加样式。

如果你使用过 jQuery，你会发现可以使用链式调用来组合不同的效果。该例子有两个效果：字体效果——下划线，以及字体颜色——彩虹色。

尝试下 zebra 和 bold：

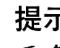
```
console.log('We be Nodin'.zebra.bold);
```

你可以对 console 信息的不同部分进行不同的样式修改：

```
console.log('rainbow'.rainbow, 'zebra'.zebra);
```

为什么像 Colors 这样的模块很有用呢？一个解释是，它可以使我们对不同事件定制不同的样式，比如对一个模块的中的错误显示一种颜色，第二个模块中的警告用另一个颜色等。为了实现这种功能，你可以使用 Colors 的预先设置或者创建自己的主题：

```
>colors.setTheme({
…….mod1_warn: 'cyan',
…….md1_error: 'red',
…….md2_note: 'yellow'
…….});
>console.log("This is a helpful message".mod2_note);
This is a helpful message
>console.log("This is a a bad message".mod1_error);
This is a bad message
```



提示

更多关于 Colors，参考：https://github.com/Marak/colors.js。

4.3.2 Optimist：另一个简单的小模块

Optimist 是我们接下来要介绍的专注于解决某个具体问题的模块。它的全部功能就是解析命令中的选项参数，虽然简单，但是功能很强大而且完备。

例如，以下代码利用 Optimist 模块输出命令行选项：

```
#!/usr/local/bin/node
var argv = require('optimist').argv;
console.log(argv.o + "" + argv.t);
```

可以用几个选项测试该段代码。以下这段代码在控制台中打印出 1 和 2 的值：

```
./app.js -o 1 -t 2
```

同样的方法也可以处理长选项：

```
#!/usr/local/bin/node
var argv = require('optimist').argv;
console.log('argv.one + ""' + argv.two);
```

测试代码如下，打印出 My Name：

```
./app2.js -one="My"--two="Name"
```

这样的方法还可以处理布尔值和未加连字号的选项。

提示

更多关于 Optimist 参考：https://github.com/substack/node-optimist。

按独立应用程序方式运行 Node 应用

本书中的绝大多数例子都用如下语法运行：

```
node appname.js
```

然后，对 Node 应用程序文件做出一些修改，就可以作为独立应用程序运行。

首先，在程序文件第一行加入以下代码：

```
#!/usr/local/bin/node
```

该路径为 Node 的安装路径。

接下来，修改该文件的权限：

```
chmod a+x appname.js
```

然后就可以运行了：

```
./appname.js
```

4.3.3 Underscore

安装 Underscore 模块：

```
npm install underscore
```

根据开发人员的描述，Underscore 模块是 Node 主要支柱之一。Underscore 提供了很多用于第三方库的 JavaScript 扩展功能，比如 jQuery 或者 Prototype.js。

类似于 jQuery 的$符号，用下划线(_)可以访问 Underscore 库函数，Underscore 因此而得名。例如：

```
var _ = require('underscore');
_.each(['apple', 'cherry'], function(fruit){ console.log(fruit)});
```

上述例子的一个问题在于下划线在 REPL 中有特殊的含义。不过不必担心，我们可以使用另一个变量 us 代替：

```
var us = require('underscore');
us.each(['apple', 'cherry'], function(fruit) { console.log(friut)});
```

Underscore 提供了对数组、集合、函数、对象、链式，以及其他工具功能的扩展功能。幸运的是，由于 Underscore 有完善的文档来说明其全部功能，所以我在这里不对其功能的细节深入描述了。

唯一一个需要特别提一下的功能是：可以通过 mixin 方法用你自己的函数扩展 Underscore 的功能。在 REPL 中可以快速体验一下：

```
>var us = require('underscore');
undefined
>us.mixin({
…betterWithNode: function(str) {
```

```
......return str + 'is better with node';
......}
...});
>console.log(us.betterWithNode('chocolate'));
chocolate is better with Node
```

提示

在很多 Node 模块中都可以看到 mixin 这个单词。它基于一种模式，一个对象的属性可以添加到（"mixed in"）另一个对象中。

当然，从应用角度来说，从一个可重用的模块扩展 Underscore 模块更有意义。这就是下一节即将介绍的内容——创建自定义模块。

4.4 创建自定义模块

正如在客户端 JavaScript 中所做的一样，你希望将可重用的 JavaScript 分离出来作为库文件。将 JavaScript 库文件转换成为 Node 可用的模块只需要多做几个步骤。

假设有一个 JavaScript 库函数——concatArray，接收 string 和 string 数组作为参数，并将第一个 string 拼接到数组中的每一个 string 上：

```
function cancatArray(str, array) {
  return array.map(function(element) {
       return str + '' + element;
  });
}
```

接下来，你希望可以在 Node 应用程序中像其他函数一样使用该函数。

将 JavaScript 库函数转换为 Node 可用的模块，需要用 exports 对象将所有需要暴露给外部使用的函数导出，如下代码所示：

```
exports.concatArray = function(str, array) {
  return array.map(function(element) {
       return str + '' + element;
  });
};
```

在 Node 应用程序中使用 concatArray，需要先用 require 导入该库文件，并将 require 语句赋值给一个变量。完成之后，你就可以调用代码中暴露的任何函数：

```
var newArray  = require('./arrayfunctions.js');
console.log(newArray.concatArray('hello', ['test1', 'test2']));
```

这个过程并不复杂，只需要记住以下两件事情：

1. 使用 exports 对象将函数暴露给外部可用；

2. 将库文件作为一个简单的导入对象，赋值给变量。就可以通过变量访问接口函数。

4.4.1 打包整个目录

你可以将你的模块分解为独立的 JavaScript 文件放在同一个目录中。以下有两种方式组织文件内容以便于 Node 加载整个目录。

第一种方法：提供一个名为 package.json 的 JSON 文件，该文件包含目录信息。文件结构中可以包含其他信息，但是与 Node 相关的接口为：

```
{ "name": "mylibrary",
  "main": "./mymodule/mylibrary.js"  }
```

第一个属性 name 是指 module 的名字。第二个属性 main 是指模块的入口。

第二种方法：在该目录中引入 index.js 或者 index.node 作为模块的主入口。

你可能会问为什么提供一个目录而不是一个简单的模块呢？很大一部分原因是因为你在使用现有的 JavaScript 库，只需要用 exports 提供打包文件就可以对所有需要使用的函数打包。另一个原因是库文件很大，需要分解以便于修改。

不管是什么原因，需要知道的是所有导出的对象都必须在同一个 Node 加载的那个主要文件中。

4.4.2 为你的模块发布做准备

如果你希望别人也可以使用你的模块，你可以将它放在自己的网站上，但是这样你会损失一大批用户。当你准备好发布一个模块的时候，你需要将它添加到 Node js 官网的模块列表中和 npm 注册表中。

之前曾提到的 package.json 文件，它是基于 CommonJS 模块系统建议的，可参考：http://wiki.commonjs.org/wiki/Packages/1.0#Package_Descriptor_File（可查看是否有更新的版本）。

package.json 文件需要的属性：

name:

包的名字。

description:

包的描述信息。

version:

符合版本要求的当前版本信息。

maintainers:

包维护信息的数组（包括名字，邮箱和网站）。

contributors:

包作者信息的数组（包括名字，邮箱和网站）。

bugs:

用于提交 Bugs 的 URL。

licenses:

licenses 的数组。

repositories:

可以找到该包的地址目录组成的数组。

dependencies:

必要的包及其版本号。

其他还有很多的属性，但是它们都是可选的。多亏了 npm，我们可以更容易的创建这个文件。如果在命令行中输入以下命令：

```
npm init
```

会遍历所有需要的属性，依次提示你完成输入。全部输入完成时，会生成 package.json 文件。

在第 3 章示例 3-13 中，创建了一个叫做 inputChecker 的对象，检查传递给命令的数据并执行命令。这个示例介绍了如何兼容 EventEmitter。现在来修改这个简单的对象，使它可以被其他应用或者模块重用。

首先，在 node_modules 路径下创建一个子目录，命名为 inputcheck。然后把现有的 inputChecker 代码放进去，需要重命名该文件为 index.js。接下来，修改代码，抽出实现该对象的部分，将这部分存储为测试文件。最后需要做的修改是添加 exports 对象，代码如如例 4-1 所示。

示例 4-1　将示例 3-13 中的代码修改为一个模块对象

```
var util = require('util');
var eventEmitter = require('events').EventEmitter;
var fs = require('fs');

exports.inputChecker = inputChecker;

function inputChecker(name, file) {
   this.name = name;
   this.writeStream = fs.createWriteStream('./ ' + file + '/txt',
      { 'flags' : 'a',
        'encoding' : 'utf8',
        'mode' : 0666 });
};

util.inherits(inputChecker, eventEmitter);
inputChecker.prototype.check = function check(input) {
   var self = this;
   var command = input.toString().trim().substr(0,3);
   if(command == 'wr') {
      self.emit('write', input.substr(3, input.length));
   } else if (command == 'en;') {
      self.emit('end');
   } else {
      self.emit('echo', input);
   }
};
```

我们没办法直接返回对象的方法，因为 util.inherits 期望在 inputChecker 文件中存在一个对象，之后会在文件中修改 inputChecker 对象的原型（prototype）方法。使用 exporst.inputChecker 修改代码是可以的，这和单独赋值给对象一样简单，但这并不是标准的做法。

运行 npminit 并回答每一个提示问题来创建 package.json 文件。该文件如示例 4-2 所示。

示例 4-2　为 inputChecker 模块生成 package.json 文件

```
{
   "author": "Shelley Powers <shelleyp@burningbird.net> (http://burningbird.net) ",
   "name": "inputcheck",
   "description": "Looks for commands within the string and implements the commands",
   "version": "0.0.1",
```

```
    "homepage": "http://inputcheck.burningbird.net"
    "repository": {
      "url": "
    },
    "main": "inputcheck.js",
    "engines": {
      "node": "~0.6.10"
    },
    "dependencies": {},
    "devDependencies": {},
    "optionalDependencies": {}
}
```

npminit 命令并不提示关于 dependencies 的内容，所以需要直接添加在该文件中。在本例中，inputChecker 模块并不依赖其他任何外部模块，所以这部分可以不添加任何内容。

提示

第 16 章会对 package.json 文件进行更深入的介绍。

现在，我们可以测试这个新模块确保它作为模块使用时的功能正常。示例 4-3 是之前 inputChecker 应用中测试新对象的部分，被抽出来作为独立的测试应用。

示例 4-3　InputChecker 测试程序

```
var inputChecker = require('inputcheck').inputChecker;

//测试新对象和事件处理
var ic = new inputChecker('Shelley','output');

ic.on('write', function(data) {
   this.writeStream.write(data, 'utf8');
});

ic.addListener('echo', function( data) {
   console.log(this.name + ' wrote ' + data);
});

ic.on('end', function() {
   process.exit();
});

process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdin.on('data', function(input) {
     ic.check(input);
});
```

现在在模块目录中新建一个 example 目录，将测试代码复制进去，作为例子与模块一起打包。好的实践要求我们不仅需要提供 test 目录，包含一个或者多个测试应用，还需要 doc 目录，包含关于模块的说明文档。像 inputChecker 这样的小模块，README 文件应该足够了。最后，我们创建模块 gzipped tarball 压缩文件。

当我们准备好所有需要的东西以后，就可以发布该模块了。

4.4.3 发布模块

给我们带来 npm 的人同样也为 Node 开发人员提供了非常棒的开源资源：开发人员指导（the Developer Guide）。它列出了所有我们需要知道的关于如何发布模块的内容。

指导文件详细说明了一些对 package.json 文件的附加要求。在已创建的现有属性的技术上，需要添加 directories 属性，值为目录与路径的哈希表，比如之前提到的 test 和 doc：

```
"directories" : {
    "doc" : ".",
    "test" : "test",
    "example" : "examples"
}
```

在发布前，指导文件建议测试模块是否可以完全安装。在模块根目录下输入以下命令进行测试：

```
npm install . -g
```

截至此时，已经测试了 inputChecker 模块，修改过 package.json 文件添加新的 directories 属性，并确认了模块可以成功安装。

接下来，需要将自己添加为 npm 用户。命令：

```
npm adduser
```

然后根据提示输入用户名，密码以及邮箱地址。

最后一件需要做的事情是：

```
npm publish
```

可以提供 tarball 或者目录的地址。指导文件中提示过，除非在 package.json 文件中用.npmignore 指定忽略某些内容，否则目录中的任何内容都会被暴露给用户。所以，

在发布之前最好将不必要的东西删除。

一旦模块发布并且代码上传到 GitHub（如果这是你使用的管理代码的工具），该模块就可以被其他用户使用了。你可以通过 Twitter、Google+、Facebook、个人网站，或者其他任何你觉得人们可以了解该模块信息的方式进行推广。这种推广并不是自我吹嘘，而是一种分享。

第 5 章

控制流、异步模式和异常处理

在了解了 Node 的异步事件、回调函数以及一些如 EventEmitter 的新对象后，或许会让你觉得想学好 Node 不是那么容易，更不用说之后的一些服务端功能了。但是，如果你之前使用过任何现代 JavaScript 库，或者已经练习并使用了本章节之前提及的一些 Node 功能的话，那么其实你已经接触过异步模式开发了。

举例来说，当你在 JavaScript 中使用计时器时，就代表你已经使用了异步功能。如果你曾经使用过 Ajax 进行开发，你同样也接触到了异步函数。即使是普通又古老的 onclick 事件处理程序也是一个异步函数，因为我们永远也不会知道用户什么时候点击鼠标或敲击键盘。

任何等待某个事件发生，或者等待处理完成以便得到结果，同时又不会阻塞控制线程执行的方法，就是一个异步函数。在应用程序进入 onclick 事件处理函数前，也就是等待用户点击鼠标的整个过程中，其他所有应用程序功能是不会被阻塞的；在定时器工作时，或者在服务器等待一个 Ajax 调用返回时，服务器上的其他功能不会被阻塞也是同样的道理。

在本章中，我们会更加深入的了解什么是异步控制。特别要去看看一些异步设计模式，并探索一些 Node 模块。使用这些模块，我们可以更加精细地控制程序流程。另外，我们还会看看在使用异步控制后，Node 程序如何捕获并处理异常，以及进行错误处理时使用的一些新的有趣特性。

5.1 使用 Callback 而不使用 Promises

在早期的 Node 版本中，使用 promises 来实现异步功能。promises 是一个出现在 20

世纪 70 年代的概念，用来描述异步操作的结果。它也被称作 future，delay 或者简单的 deferred。CommonJS 的设计模型就实现了 promise 的概念。

在早期的 Node 实现中，promise 对象仅仅会激发两个事件类型：success 和 error。它的使用也很简单：如果一个异步操作成功，success 事件会被触发，否则，error 事件会被触发。除此之外，promise 对象不会触发其他事件类型，而且它只能触发两种事件中的一个，不会也不可能同时触发两个事件。另外，一个 promise 对象总共只能触发一次事件，而不论这个事件是 success 还是 error。在示例 5-1 的代码中，我们在一个函数中使用了旧版本的 promise 实现，这个函数同时还会打开并读取一个文件内容。

示例 5-1　使用 Node promise

```
function test_and_load(filename) {
  var promise = new process.Promise();
  fs.stat(filename).addCallback(function (stat) {

    // Filter out non-files
    if (!stat.isFile()) { promise.emitSuccess(); return; }

    // Otherwise read the file in
    fs.readFile(filename).addCallback(function (data) {
      promise.emitSuccess(data);
    }).addErrback(function (error) {
      promise.emitError(error);
    });

  }).addErrback(function (error) {
    promise.emitError(error);
  });
  return promise;
}
```

每个对象都能返回 promise 对象。通过 promise 对象的 addCallback 方法可以绑定一个回调函数来做异步操作成功后的处理，该回调函数只有一个参数 data。通过 promise 对象的 addErrback 方法可以绑定一个回调函数来处理异步操作失败的情况，该回调函数也同样只有一个唯一的参数 error。

```
var File = require('file');
var promise = File.read('mydata.txt');
promise.addCallback(function (data) {
   // process data
});
promise.addErrback(function (err) {
   // deal with error
})
```

在对异步操作得到的执行结果或者错误信息处理完成后，promise 对象会确保适当的功能被调用执行。

提示

示例5-1的代码是参考了链接http://groups.google.com/group/nodejs/browse_thread/thread/8dab9f0a5ad753d5 中提供的示例，该文档中还包含了其他一些示例，主要讨论 Node 中使用的一些异步技术。

promise 对象在 Node 0.1.30 版本中被移除。Ryan Dahl 当时也给出了这么做的原因：

许多人（也包括我自己）往往习惯于调用底层接口来操作文件系统，而不是每次都要创建一个对象；当然，也有很多人会比较喜欢类似于 promises 的方式。因此，我们将使用 last callback functionality（即使用函数的最后一个参数作为回调）并在用户库中建立更好的抽象层来取代 promises。

为了取代 promise 对象，Node 使用了 last callback functionality（特函数的最后一个参数作为回调）。正如我们在之前章节使用的那样，所有异步方法的最后一个参数都可以被指派一个回调函数，而这个回调函数的第一个参数始终是一个 error 对象。

为了说明该回调功能的基本结构，示例 5-2 实现了一个完整的 Node 应用程序。在程序中我们创建了一个对象，并且该对象只有一个 someMethod 方法。这个方法有三个参数，其中第二个必须是一个字符串，第三个是回调函数。如果第二个参数丢失或不是一个字符串，一个 Error 对象就会被创建并传递给回调函数。否则，该方法会传递参数给回调函数。

示例 5-2　last callback functionality 的基本结构

```
var obj =function() { };

obj.prototype.doSomething = function(arg1, arg2_) {
  var arg2 = typeof(arg2_) === 'string' ? arg2_ : null;

  var callback_ = arguments[arguments.length - 1];
  callback = (typeof(callback_) == 'function' ? callback_ : null);

  if (!arg2)
     return callback(new Error('second argument missing or not a string'));

  callback(arg1);
}
var test = new obj();

try {
   test.doSomething('test', 3.55, function(err,value) {
      if (err) throw err;

      console.log(value);
   });
} catch(err) {
```

```
        console.error(err);
    }
```

last callback functionnality 结构所需的关键要素已经在代码中以粗体标注。

第一个关键点是确保最后一个参数是一个回调函数。我们不能确定用户的意图，但可以确保最后一个参数是一个函数而且也不得不这样做。第二个关键点是如果异步操作发生错误时，要创建新的 Node Error 对象并传递给回调函数。最后一个关键点是在异步操作没有发生任何错误时，调用回调函数并传入处理结果。总之，其他都可以变，但这三个关键点必须被实现：

*确保最后一个参数是一个函数；

*创建 Node Error 对象，如果发生错误，则传递它给回调函数；

*如果没有发生错误，则调用回调函数，并传递处理结果。

运行示例 5-1 的代码后，应用程序会输出以下错误消息到控制台：

```
[Error: second argument missing or not a string]
```

修改代码中的方法调用：

```
test.doSomething('test','this',function(err,value) {
```

test 将被输出到控制台上。再次修改代码：

```
test.doSomething('test',function(err,value) {
```

运行后会再次得到一个错误信息，这次是因为缺少第二个参数。

如果你浏览 Node 安装目录中 lib 目录下的所有代码，你会发现 last callback pattern 在很多地方反复出现。虽然功能可能改变，但这种模式保持不变。

虽然这是一种相当简单的方法，但却能保证不同异步函数有一致的使用方法。然而，这种方式也存在一些问题，我们将在下一节讨论。

5.2 顺序调用、嵌套回调、异常捕获

在客户端的 JavaScript 应用程序中，经常会看到以下代码：

```
val1 = callFunctionA();
val2 = callFunctionB(val1);
val3 = callFunctionC(val2);
```

函数被依次调用执行，先执行函数的输出结果会作为后续函数的输入参数。由于所有的函数都是同步调用执行，我们不必担心因为函数调用顺序发生变化而得到意料之外的执行结果。

示例 5-3 是采用了常见的顺序编程方法。该应用程序使用 Node 提供的非异步版本的文件系统方法，在打开并读取一个文件后，对其内容进行了修改，将所有 “apple” 替换为 “orange”，最后将修改后的数据输出到一个新文件中。

示例 5-3　顺序执行的示例程序

```
var fs = require('fs');

try {
   var data = fs.readFileSync('./apples.txt','utf8');
   console.log(data);
   var adjData = data.replace(/[A|a]pple/g,'orange');

   fs.writeFileSync('./oranges.txt', adjData);
} catch(err) {
  console.error(err);
}
```

由于不能确定所使用的模块是否能够捕获并处理错误，所以我们将整段代码放在一个 try 块中，以便程序在产生任何可能的错误时能获得更多的异常处理信息。下面展示了当应用程序无法找到要读取的文件时给出的错误信息：

```
{ [Error: ENOENT, no such file or directory './apples.txt']
  errno: 34,
  code: 'ENOENT',
  path: './apples.txt',
  syscall: 'open' }
```

虽然这段错误信息看起来并不是特别的友好，但至少它比下面这种要好很多：

```
node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
              ^
Error: ENOENT, no such file or directory './apples.txt'
    at Object.openSync (fs.js:230:18)
    at Object.readFileSync (fs.js:120:15)
    at Object.<anonymous> (/home/examples/public_html/node/read.js:3:18)
    at Module._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:31)
    at Function._load (module.js:308:12)
    at Array.0 (module.js:479:10)
    at EventEmitter._tickCallback (node.js:192:40)
```

在这个例子中，我们可以得到预期的结果，因为每个函数都是按顺序被调用执行的。

要将这个同步方式的程序采用异步方式实现的话，需要做一些修改。首先，必须将所有函数调用更换为异步版本。然而，我们也必须考虑到的一个事实是每个异步函数调用都是不阻塞的，这意味着这些函数是相互独立的，这样我们就无法保证原本正确的逻辑顺序了。唯一能保证多个异步函数能按正确逻辑顺序被调用执行的方法是使用嵌套回调（nested callbacks）。

示例 5-4 是修改示例 5-3 后得到的异步版本程序。所有的文件系统相关的函数调用都采用异步版本取代，并且使用了嵌套回调以保证所有函数能按照正确顺序被调用执行。

示例 5-4　示例 5-3 程序的异步实现

```
var fs = require('fs');

try {
  fs.readFile('./apples2.txt','utf8', function(err,data) {

    if (err) throw err;

    var adjData = data.replace(/[A|a]pple/g,'orange');

    fs.writeFile('./oranges.txt', adjData, function(err) {

      if (err) throw err
    });
  });
} catch(err) {
  console.error(err);
}
```

在示例 5-4 中，程序首先打开指定文件，然后读取文件内容；只有当这两个动作完成后，作为最后一个参数的回调函数才会被调用执行。回调函数首先检查 error 是否为空，如果不为空，则抛出该错误对象。最外层的异常捕获代码会处理该错误信息。

提示

一些编程规范指导并不建议在代码中抛出 error，或采用复杂的框架来保证所有错误都能被捕获并处理。但无论如何，错误需要被捕获并处理。

如果没有错误发生时，数据会先经过替换操作处理，然后异步版本的 writeFile 方法会被调用。这个异步函数的回调函数只有一个参数 error。如果 error 不为 null，它将被抛出并传递给外层的异常处理块。

如果发生错误，输出信息看起来会类似于以下内容：

```
/home/examples/public_html/node/read2.js:11
        if (err) throw err;
                       ^
Error: ENOENT, no such file or directory './boogabooga/oranges.txt'
```

如果你想查看调用堆栈来跟踪错误，可以打印出 error 对象 stack 属性：

```
catch(err) {
   console.log(err.stack);
}
```

示例 5-5 的代码中，我们为了访问一个目录中的文件清单，实现了更深层的嵌套调用，目的是为了访问一个目录中的文件清单。程序使用字符串类提供的 replace 方法将每个文件中的域名替换为指定域名，并把结果写回到原文件中。同时，我们还打开了一个可写流来保存对每个文件的修改记录。

示例 5-5　读取目录文件列表并修改文件内容

```
var fs = require('fs');

var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

try {
   // get list of files
   fs.readdir('./data/', function(err, files) {

      // for each file
      files.forEach(function(name) {

         // modify contents
         fs.readFile('./data/' + name,'utf8', function(err,data) {

            if (err) throw err;
            var adjData = data.replace(/somecompany\.com/g,'burningbird.net');

            // write to file
            fs.writeFile('./data/' + name, adjData, function(err) {

               if (err) throw err;

               // log write
               writeStream.write('changed ' + name + '\n', 'utf8', function(err) {

                  if(err) throw err;
               });
            });
         });
      });
   });
} catch(err) {
  console.error(util.inspect(err));
}
```

尽管应用程序看起来像是单独处理完一个文件然后再移动并处理下一个文件，但它的确是异步工作的。如果你在多次运行应用程序后观察 log.txt 文件的内容，会发现

程序在每次运行时处理文件的顺序都是不一样的，而且看起来更像是随机的。在我的 data 子目录中有五个文件。运行应用程序三次后，我得到的 log.txt 文件内容如下：

```
changed data1.txt
changed data3.txt
changed data5.txt
changed data2.txt
changed data4.txt

changed data3.txt
changed data1.txt
changed data5.txt
changed data2.txt
changed data4.txt

changed data1.txt
changed data3.txt
changed data5.txt
changed data4.txt
changed data2.txt
```

这里有一个问题：怎么才能知道所有文件都被处理完成的时间点呢？也许我们想知道这个时间点，以便做一些后续处理操作。不过，代码中为 forEach 方法指定的回调函数都是异步的，因此不会阻塞程序。让我们先在 forEach 语句后面添加如下代码：

```
console.log('all done');
```

当程序输出“all done”时，并不表示所有处理都完成了，它仅能说明 forEach 方法没有被阻塞。

再次修改程序，在输出信息到 log.txt 文件的地方增加 console.log 输出，如下所示：

```
writeStream.write('changed ' + name + '\n', 'utf8', function(err) {
    if(err) throw err;
    console.log('finished ' + name);
});
```

同时在 forEach 方法后添加如下代码：

```
console.log('all finished');
```

运行程序后，你会得到如下控制台输出：

```
all done
finished data3.txt
finished data1.txt
finished data5.txt
finished data2.txt
finished data4.txt
```

为了解决这个问题，需要添加一个计数器。在每次记录日志信息的时候让这个计数器递增，然后将计数值与文件数组的长度对比来决定是否打印输出 “all done” 消息：

```
// before accessing directory
var counter = 0;
...
                writeStream.write('changed ' + name + '\n', 'utf8', function(err) {
                  if(err) throw err;
                  console.log('finished ' + name);
                  counter++;
                  if (counter >= files.length)
                     console.log('all done');
                });
```

再次运行程序，你会得到预期结果：“all done” 消息在所有的文件被更新后才输出。

在程序所访问的目录中如果没有子目录的话，这段示例程序会工作得很好。如果存在子目录的话，程序会输出以下错误信息：

```
/home/examples/public_html/node/example5.js:20
              if (err) throw err;
                             ^
Error: EISDIR, illegal operation on a directory
```

示例 5-6 使用 fs.stats 方法来防止发生这种错误。fs.stats 方法会返回一个对象，用于描述 Unix 中 stat 命令所返回的数据。这个对象包含的信息可以用来判断当前对象是否是一个文件。同样的，fs.stats 也是以异步方式工作的，因此我们需要做更深的回调嵌套。

示例 5-6　使用 stats 函数检查文件类型

```
var fs = require('fs');

var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});

try {
   // get list of files
   fs.readdir('./data/', function(err, files) {

      // for each file
      files.forEach(function(name) {

         // check to see if object is file
         fs.stat('./data/' + name, function(err, stats) {

            if (err) throw err;

            if (stats.isFile())
```

```
                // modify contents
                fs.readFile('./data/' + name,'utf8', function(err,data) {

                   if (err) throw err;
                   var adjData = data.replace(/somecompany\.com/g,'burningbird.net');

                   // write to file
                   fs.writeFile('./data/' + name, adjData, function(err) {

                      if (err) throw err;

                      // log write
                      writeStream.write('changed ' + name + '\n', 'utf8',
                                                             function(err) {
                         if(err) throw err;
                      });
                   });
                });
            });
         });
      });
   } catch(err) {
     console.error(err);
   }
```

这样应用程序又可以按照预期执行了，并且执行得很好，但是由于使用了多层嵌套，对代码的阅读和维护就显得很困难了。我曾经听说人们把这种回调嵌套也叫做 callback spaghetti，还有一个更形象的说法是 pyramid of doom，这两个叫法都能说明这个问题。

嵌套回调越多，代码文档的边沿就更向右扩展，也使得我们更难以保证之后的回调函数中代码的正确性。然而，我们不能打破回调嵌套，因为是它保证了函数能依次按照如下顺序被调用执行：

1. 首先检索目录内容；
2. 过滤掉子目录；
3. 读取每个文件的内容；
4. 修改内容；
5. 将内容写回到原文件。

因此，我们需要做的是找到另一种方法来完成按序对这一系列方法的调用，而不必依赖于嵌套回调。为了达到这样的目的，就需要使用提供异步控制流的第三方模块。

提示

为所有异步方法使用同一个回调函数也是一种办法。通过这种方式，你可以使得代码扁平化，也可以简化调试。然而，这种方法还存在着其他一些问题，比如如何确定所有处理已经完成的时间点等。为了解决这个问题，你仍然需要使用第三方库。

5.3 异步模式和控制流模块

示例 5-6 实现的应用程序实际上就展示了一种异步模式，每个函数被依次调用，并且前一个函数将处理结果传递给下一个处理函数，并仅在有错误发生时整个处理链条才停止。另外还有几个与此类似的模式，它们之间有些许差异而且叫法也不尽相同。

在 Node 中 Async 模块可以支持最广泛的异步控制流模式，并提供了如下模式列表：

waterfall

所有函数按照顺序被依次调用执行，所有的处理结果以数组形式传递给最后一个回调函数（也叫着 series 或者 sequence）。

series

所有函数按照顺序被依次调用执行，所有的处理结果随机的以数组形式传递给最后一个回调函数。

parallel

所有函数被并行执行，在处理完成后，处理结果被传递给最后一个回调函数（在其他一些对 parallel 模式的解释中，结果数组并不是模式的一部分）。

whilst

重复调用一个函数，除非某些预设的起始条件返回 false 或者发生错误后才调用最后一个回调函数。

queue

函数被并行调用执行，但是同一时间可并行执行的函数总数有一定限制，没有被执行的函数会被队列起来等待执行。

until

重复调用一个函数，直到后处理判断规则返回 false 或者发生错误后才调用最

后一个回调函数。

auto

按需调用函数，每个函数会接收并处理上一个函数的处理结果。

iterator

每一个函数会调用下一个，并且每一个函数都有访问 next 迭代器的能力。

apply

根据参数创建一个函数，与其他控制流功能结合使用。

nextTick

在 Node 的下一轮事件循环中调用回调函数，该模式基于 process.nextTick 实现。

在 Node.js 网站提供的模块列表中，有一个"控制流量/异步（Control Flow/Async Goodies）"分类。在这份列表中我们可以找到 Async 模块，并使用它来实现上述异步控制模式。虽然不是每一个控制流量模块都能提供对所有模式的支持，但他们大都提供了对常用模式的支持，比如：series（也叫 sequence 或者 waterfall，如上面列表所列，尽管 Async 将 waterfall 与 series 分别单独列出）和 parallel。此外，一些模块还依照早期 Node 版本实现了 promises 的概念，还有一些模块实现了 fibers 的概念，它能模拟线程。

在接下来的几个小节中，我会使用两个比较流行的并且一直有作者维护的控制流模块：Step 和 Async。尽管这两个模块都提供了一些必要功能，但每个模块也都提供了各自独特的异步控制流管理。

5.3.1 Step

Step 是一个用于简化串行和并行控制流的实用模块。通过下面这条 npm 指令可以安装它：

```
npm install step
```

Step 模块对外仅暴露出一个对象。要使用该对象做串行顺序调用的话，需要将你的异步函数分别包装在多个函数中，然后将该函数序列当作参数传递给该对象。示例 5-7 展示了如何使用 Step 对象，程序首先读取文件内容，然后修改内容，最后把修改后的内容写回原文件。

示例 5-7　使用 Step 执行串行异步任务

```
var fs = require('fs'),
    Step = require('step');

try {

   Step (
      function readData() {
          fs.readFile('./data/data1.txt', 'utf8', this);
      },
      function modify(err, text) {
         if (err) throw err;
         return text.replace(/somecompany\.com/g,'burningbird.net');
      },
      function writeData(err, text) {
          if (err) throw err;
          fs.writeFile('./data/data1.txt', text, this);
      }
    );
} catch(err) {
   console.error(err);
}
```

在 Step 的调用序列中，第一个函数是 readData，该函数读取并将文件内容保存到一个字符串对象，然后将其传递给第二个函数。第二个函数使用替换方法修改该字符串，将修改结果传递到第三个函数。第三个函数负责将修改后的字符串写回原文件。

提示

欲了解更多信息，请参阅 Step 在 GitHub 上的站点：*https://github.com/creationix/step*。

让我们更详细地看下这段代码，会注意到第一个函数包装了对异步方法 fs.readFile 的调用。然而，在调用该异步函数时，this 上下文对象被作为最后一个参数传递给了该函数，而非我们通常使用回调函数。在异步函数执行完毕后，它得到的所有数据或者任何可能的错误信息都会传递给 Step 调用序列中的下一个函数 modify。modify 并没有调用另一个异步函数，它所做的所有工作仅仅是对字符串中的某些子串做替换。它并不需要使用 this 上下文对象，而是通过 return 语句将处理结果返回。

最后一个函数得到了新修改后的字符串，并将其写回原文件中。不过因为它是一个异步函数，我们同样将 this 对象作为最后一个参数传入。如果不将 this 作为该异步函数的最后一个参数，那么当产生错误信息时，就只能在更外层进行捕获和处理了。可以做个测试，假如将代码做如下修改，指定一个不存在的子目录：

```
function writeFile(err, text) {
    if (err) throw err;
    fs.writeFile('./boogabooga/data/data1.txt');
```

```
}
```

这样我们将永远不会知道写入操作实际上是失败了。

即使第二个函数没有使用异步方式，但在 Step 调用序列中，除第一个函数外，其余函数的第一个参数都应该是 error。

示例 5-7 实现了示例 5-6 中的部分功能。那么它是否能对多个文件做修改呢？答案是肯定的，它可以做到。不过还需要先做一些修改，丢掉一些有缺陷的代码。

在示例 5-8 中，我添加了用于获取指定子目录文件列表的异步函数 readir。像示例 5-6 中一样通过 forEach 命令处理文件列表，但是在调用 readFile 函数时传入的最后一个参数并不是回调函数或者 this 对象。在 Step 模块中，调用 group 对象表示需要保留一个参数用于保存一组处理结果；在 readFile 函数中对 group 对象的调用意味着所有回调函数被依次调用后，所有处理结果将被放置到一个数组中并传递给调用链中的下一个函数。

示例 5-8 使用 Step 的 group()将异步处理分组

```
var fs = require('fs'),
    Step = require('step'),
    files,
    _dir = './data/';

try {

   Step (
      function readDir() {
          fs.readdir(_dir, this);
      },
      function readFile(err, results) {
          if (err) throw err;
          files = results;
          var group = this.group();
          results.forEach(function(name) {
             fs.readFile(_dir + name, 'utf8', group());
          });
      },
      function writeAll(err, data) {
         if (err) throw err;
         for (var i = 0; i < files.length; i++) {
           var adjdata = data[i].replace(/somecompany\.com/g,'burningbird.net');
           fs.writeFile(_dir + files[i], adjdata, 'utf8',this);
         }
      }
    );
} catch(err) {
   console.log(err);
}
```

代码会将 readdir 读取到的文件列表保存在全局变量 files 中。在最后一个函数里，

我们在一个循环体中通过 files 变量拿到所有文件名称并修改相应的文件内容。文件名称以及修改后的文件内容都在最后调用异步函数 writeFile 时用到了。

如果需要修改的文件是固定的，那么就可以进行硬编码并使用 Step 提供的另一种方法 parallel 来实现该程序。在示例 5-9 中，一个 readFile 异步函数被调用多次，每次打开不同的文件，并指定 this.parallel()作为最后一个参数。这样，每次调用 readFile 函数读取到的文件内容会作为第二个调用链函数的一个参数。同样在第二个函数中对 writeFile 的调用也要使用 parallel()，以保证每个回调函数被依次处理。

示例 5-9　使用 Step 模块的 parallel 功能对一组文件进行读写操作

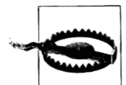
```
var fs = require('fs'),
    Step = require('step'),
    files;

try {

   Step (
      function readFiles() {
         fs.readFile('./data/data1.txt', 'utf8',this.parallel());
         fs.readFile('./data/data2.txt', 'utf8',this.parallel());
         fs.readFile('./data/data3.txt', 'utf8',this.parallel());
      },
      function writeFiles(err, data1, data2, data3) {
         if (err) throw err;
         data1 = data1.replace(/somecompany\.com/g,'burningbird.net');
         data2 = data2.replace(/somecompany\.com/g,'burningbird.net');
         data3 = data3.replace(/somecompany\.com/g,'burningbird.net');

         fs.writeFile('./data/data1.txt', data1, 'utf8', this.parallel());
         fs.writeFile('./data/data2.txt', data2, 'utf8', this.parallel());
         fs.writeFile('./data/data3.txt', data3, 'utf8', this.parallel());
      }
    );
} catch(err) {
   console.log(err);
}
```

这段代码看起来稍显笨拙但却能够正常工作。所以，当你需要调用一序列功能各不相同但又可以并行执行的异步函数，并且之后的处理分析又需要这些函数返回的数据时，使用 parallel 是比较合适的。

不过当 Step 无法满足需要时，我们无需在代码中牵强的使用它，因为还有另外一个库：Async，它能为我们提供更好的灵活性。

5.3.2　Async

Async 模块提供了对集合的管理功能，例如 each、map 和 filter。此外，它还提供了一些实用功能，例如 memoization。然而，此刻我们关心的是它对流程控制的支持。



警告

请注意 Async 和 Async.js 模块的区别，不要将两者混淆。本章节所讲的是由 Caolan McMahon 编写的 Async 模块。它在 GitHub 的地址是：*https://github.com/caolan/async*。

使用如下命令安装 Async 模块：

```
npm install async
```

我们之前对 Async 模块做过介绍，它提供的流程控制功能可以支持多种异步模式，包括 serial、parallel 和 waterfall。与 Step 模块一样，它也是一个可以解决嵌套回调问题的工具，但其实现方式却完全不同。因为，无需再将 this 插入每个函数的实现中来替换相应的回调函数。相反，它是通过 callback 来融合整个执行过程。

我们已经知道了之前的示例程序符合 Async 所定义的 waterfall 模式，因此我们将会使用 async.waterfall 方法。在示例 5-10 中，我使用 async.waterfall 来实现一序列操作，包括用 fs.readFile 打开和读取一个数据文件，执行一个同步的字符串替换操作，然后再使用 fs.writeFile 将字符串写回到文件中。要特别留意下程序每一步操作中对 callback 函数的使用。

示例 5-10　使用 async.waterfall 异步实现读取，修改和写入文件内容

```
var fs = require('fs'),
    async = require('async');

try {
   async.waterfall([
      function readData(callback) {
         fs.readFile('./data/data1.txt', 'utf8', function(err, data){
              callback(err,data);
          });
      },
      function modify(text, callback) {
         var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
         callback(null, adjdata);
      },
      function writeData(text, callback) {
          fs.writeFile('./data/data1.txt', text, function(err) {
             callback(err,text);
          });
      }
    ], function (err, result) {
         if (err) throw err;
         console.log(result);
    });
 } catch(err) {
   console.log(err);
 }
```

async.waterfall 方法有两个参数：一个任务数组以及一个可选的最终回调函数（final callback function）。任务数组中的每一个元素都是一个任务函数，而且每个任务函数都需要使用一个 callback 来作为其最后一个参数。正是由于使用了 callback，才能够将异步调用的返回处理结果链接起来，而无需再将函数嵌套起来。你可以在代码中看到，对每个函数 callback 的使用与正常使用嵌套回调时一样，不过还有一个事实那就是我们无需再在每个函数中判断错误信息了。callback 需要一个 error 对象作为第一个参数。如果我们传递一个 error 对象给 callback 函数，整个处理过程将会在这一点结束，而最终回调函数（waterfall 方法的第二个参数）会被调用。我们可以在最终回调函数中测试是否有错误发生，并抛出错误到外层的异常处理块（或以其他方式处理）。

readData 函数包装了 fs.readFile 调用。它首先会检查是否有错误发生，如果发生错误，它会抛出错误并结束处理过程。如果不是，它会调用 callback 函数作为其最后一步操作。通过该操作告诉 Async 调用下一个函数并传递相关数据。而下一个任务函数并不是异步的，在它完成处理后会通过调用 callback 来传递修改后的数据以及值为 null 的错误对象。最后一个函数 writeData 调用了异步的 writeFile，我们将上一步处理的结果传入该异步函数，同时在该异步函数自己的回调例程中进行错误检查。

提示

示例 5-10 的代码中，我们使用了命名函数，而在 Async 文档中使用匿名函数。不过，命名函数可以简化调试和错误处理。但无论如何，使用命名函数还是匿名函数，Async 都能很好地工作。

处理过程与我们在示例 5-4 做的非常相似，但没有使用嵌套（也无需在每个函数中添加错误判断）。不过代码看起来比示例 5-4 要复杂一些。事实上，我的确不推荐在可以使用简单嵌套调用的情况下使用 Async，不过倒是可以考虑在碰到复杂嵌套调用时使用它。示例 5-11 实现了示例 5-6 的所有功能，但没有使用嵌套回调，从而避免了代码的过度缩进。

示例 5-11　从目录中获取对象，测试并寻找文件；读取文件内容，修改并写回；记录修改日志

```
var fs = require('fs'),
    async = require('async'),
    _dir = './data/';

var writeStream = fs.createWriteStream('./log.txt',
      {'flags' : 'a',
       'encoding' : 'utf8',
       'mode' : 0666});
try {
   async.waterfall([
      function readDir(callback) {
```

```
            fs.readdir(_dir, function(err, files) {
               callback(err,files);
            });
         },
         function loopFiles(files, callback) {
            files.forEach(function (name) {
               callback (null, name);
            });
         },
         function checkFile(file, callback) {
            fs.stat(_dir + file, function(err, stats) {
               callback(err, stats, file);
            });
         },
         function readData(stats, file, callback) {
            if (stats.isFile())
               fs.readFile(_dir + file, 'utf8', function(err, data){
                 callback(err,file,data);
             });
         },
         function modify(file, text, callback) {
            var adjdata=text.replace(/somecompany\.com/g,'burningbird.net');
            callback(null, file, adjdata);
         },
         function writeData(file, text, callback) {
             fs.writeFile(_dir + file, text, function(err) {
                callback(err,file);
             });
         },
         function logChange(file, callback) {
             writeStream.write('changed ' + file + '\n', 'utf8', function(err) {
                callback(err, file);
            });
         }
      ], function (err, result) {
            if (err) throw err;
            console.log('modified ' + result);
      });
   } catch(err) {
      console.log(err);
   }
```

与示例 5-6 实现的所有功能一样。fs.readdir 方法被用来取得目录下的对象数组。Node 提供的 forEach 函数（没有使用 Async 的 forEach）用来访问每一个文件对象。fs.stats 方法用于取得每个对象的 stats 状态信息，以便检查该对象是否是一个文件，如果是文件则打开并访问文件数据。然后，数据被修改，并通过 fs.writeFile 将数据写回文件。操作都记录在日志文件中，同时也会输出到控制台。

请注意，我们可能需要在回调函数间传递大量数据。在本示例代码中，由于很多函数都使用到了 filename 和 text，所以它们在最后几个方法间传递。我们可以在方法间传递任何数据，只要第一个参数是 error 对象（如果没有错误产生，则为 null），并保证每个函数的最后一个参数是 callback 函数。

我们不必在每个异步任务函数中进行错误检查，因为 Async 会在每个 callback 被调用时检查 error 对象，如果发生错误，则停止处理并调用最终回调函数。我们也无需再考虑如何使用特别的方式来处理一组任务了，就像在之前章节使用 Step 时做的一样。

Async 模块还支持其他一些流程控制方法，例如 async.parallel 和 async.serial。它们的使用方法也很类似，首先需要一个任务数组作为第一个参数，而第二个参数是可选的最终回调函数。不过不同的流程控制方法对异步任务的处理还是有稍许不同的。

提示

在本书第 9 章 9.2 节，我们会使用 async.searial 方法构建一个 Redis 应用程序。

async.parallel 方法会一次性调用所有异步任务，当所有任务执行完毕时，将调用最终回调函数。示例 5-12 使用 async.parallel 并行读取三个文件的内容。然而，该程序没有使用函数数组，而采用了 Async 模块提供的另一种替代方法：传入一个包含有所有异步任务的对象，每个任务都是该对象的一个属性。当三个任务都已经完成后，结果会输出到控制台。

示例 5-12　使用并行方式打开三个文件并读取内容

```
var fs = require('fs'),
    async = require('async');

try {
   async.parallel({
      data1 : function (callback) {
         fs.readFile('./data/data1.txt', 'utf8', function(err, data){
              callback(err,data);
          });
      },
      data2 : function (callback) {
         fs.readFile('./data/data2.txt', 'utf8', function(err, data){
              callback(err,data);
          });
      },
      data3 : function readData3(callback) {
         fs.readFile('./data/data3.txt', 'utf8', function(err, data){
              callback(err,data);
          });
      },

    }, function (err, result) {
         if (err) throw err;
         console.log(result);
    });
} catch(err) {
   console.log(err);
}
```

返回的结果是一个对象数组，每个对象的属性中包含相应的处理结果。如果在本例中的三个数据文件有以下内容：

- data1.txt: apples
- data2.txt: oranges
- data3.txt: peaches

运行示例 5-12 的结果是：

```
{ data1: 'apples\n', data2: 'oranges\n', data3: 'peaches\n' }
```

Async 模块支持的其他控制流模式就留给读者自己练习吧！只要记住：当你在工作中使用异步控制流的相关方法时，需要传递一个 callback 给每个异步任务，并且当任务完成后需要调用这个 callback 并传入一个错误对象（或 null）以及任何你需要传递的数据。

5.4 Node 编码风格

本书中我很多次提到并给出了一些编码上的建议，例如在 Node 中使用命名函数而非匿名函数。所有这些建议可以被理解为首选风格（preferred Node style），尽管并没这样一套编程风格指南或一套共享风格定义。事实上，也有人提出过一些不同的 Node 风格建议。

提示

Felix's node.js Style Guide 是一个比较好的 Node.js 风格指南，可以在 *http://nodeguide.com/style.html* 找到。

这里有一些建议，还有我自己对这些建议的看法：

尽量使用异步函数而非同步函数。

没错，这对 Node 应用程序至关重要。

使用两个空格的缩进。

我的看法：对不起，我习惯使用三个空格，而且我也将继续用三个空格。我认为更重要的是要保持一致并且不要使用 tab，而非对空格数吹毛求疵。

使用分号还是不使用分号。

对此有争议的确令人惊讶。我使用分号，不过是否使用需要看你自己的习惯了。

使用单引号。

我更习惯用双引号，但也在慢慢改掉这个习惯（或多或少）。但无论如何，在定义有单引号内容的字符串时，最好能使用双引号将字符串包起来。

当定义多个变量时，是否只使用一个 var 关键字。

在这本书的应用程序中有一些使用了一个 var 关键字，有些则没有。再说一次，老习惯很难打破，而且我也并不认为这个问题像一些人说的那么严重。

常量应该是大写的。

我同意这个。

变量定义应该采用驼峰式大小写（camel case）。

我或多或少同意这种说法，但并不是不能改变。

使用全等符（===）。

中肯的意见，但我再说一遍，老习惯很难打破。我建议使用严格相等，但平常情况下仅使用相等（==）。不要像我一样有这个坏习惯。

命名你的闭包。

我再次没有遵守。但这真的是非常中肯的建议，我想改掉我的毛病，但我的很多代码中仍然使用了匿名函数。

一行代码的最大长度应小于 80 个字符。

同样中肯的意见。

大括号必须在同一行上开始。

我的确遵守了这条建议。

除过这些建议之外，最重要是要记住：随时随地的使用异步函数。毕竟，异步功能就像 Node 的心脏。

第 6 章

路由寻址、服务文件和中间件

当你在网页上点击一个链接的时候心里一定期望着什么事情发生，一般都期待看到一个新加载的页面。但是，在网络资源加载前有很多事情发生，而且大部分都不在我们的控制下（比如封包路由）。还有一部分依赖于我们是否安装了一些软件知道如何回应点击的该链接的内容。

当然，我们在使用类似于 Apache 的服务器和 Drupal 这类软件时，大部分对文件和资源的处理机制都在幕后进行了。然而，当用 Node 创建自己的服务器端应用程序而没有采用常用技术的时候，我们必须进一步深入研究，以确保正确的资源在正确的时间被传递。

本章关注的焦点是供 Node 开发人员使用的技术，提供基本的路由以及中间件功能，确保资源 A 准确快速地到达用户 B。

6.1 从头开始：创建一个简单的静态文件服务器

我们现有的功能可以用于构建一个简单的路由和给 Node 内置一个提供静态文件服务。但是“能做到”和“容易做”是两件事情。

当考虑构建一个简单但是功能完整的静态文件服务器时，可能会有以下几个步骤：

1. 创建 HTTP 服务器并监听是否有请求；
2. 当接收到请求时，解析请求的 URL，决定文件地址；
3. 检查请求的文件是否存在；
4. 如果文件不存在，给出适当的响应；

5. 如果文件存在，打开文件进行读取；

6. 准备响应（respoonse）的头部（header）;

7. 将文件写入响应内容；

8. 等待下一次请求。

创建 HTTP 服务器并对文件进行读取需要用到 HTTP 和 FileSystem 模块。Path 模块也派得上用场，它可以在读取文件前检查确保该文件存在。而且我们希望为根目录定义一个全局变量，或者也可以使用预先定义好的__dirname（更多关于__dirname 内容请见接下来的说明栏“为什么不使用__dirname 呢？”，110 页）。

该应用开头部分如下代码所示：

```
var http = require('http'),
        path = require('path'),
          fs = require('fs'),
        base = 'home/examples/public_html';
```

借助 HTTP 模块创建服务器并不是什么新的想法，而且应用程序可以通过访问 HTTP 请求对象的 url 属性得到请求的文件。为了根据请求确认 response，我们在 console.log 中输出请求的文件路径名。当服务器第一次启动时，该信息会显示在 console.log 的输出的信息中。

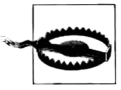
```
http.createServer(function (req, res) {

    pathname = base + req.url;
    console.log(pathname);

}).listen(8124);

console.log('Server running at 8124/');
```

在尝试打开文件，读取文件内容并写入 HTTP response 中之前，应用程序需要检查文件是否存在。此时，path.exists 方法是个很好的选择。如果文件不存在，填写一个简单的信息并设置状态码（status code）为 404：找不到该文件。

```
path.exist(pathname, function(exists) {
    if (exits) {
        //插入处理 request 的代码
    } else {
         res.writeHead(404);
         res.write('Bad request 404\n');
         res.end();
```

```
}
```

接下来，我们要进入这个新应用程序的核心了。在前面章节的例子中，我们使用 fs.readFile 读取文件。fs.readFile 的问题在于读取的内容必须等到整个文件全部读取到内存后才可以使用。网络上提供的文件可能很大，并且对同一个文件在给定的时间可能有很多请求，类似与 fs.readFile 这样的功能无法满足需求。



警告

path.exists 方法在 Node 0.8 版本中已经弃用了，新的替代品是 fs.exists。前言中引用的示例文件包含的应用程序对两种方法都支持。

该应用程序通过 fs.createReadStream 方法使用默认设置创建文件读取流来替代 fs.readFile 方法。这样将文件内容用 pipe 方法写入 HTTP response 对象就很容易了。而且流对象在内容传输结束时会发送结束信号，所以我们并不需要自己调用 end 方法：

```
res.setHeader('Content-Type', 'test/html');

//Status code:200  --找到文件，无错误;
res.statusCode = 200;

//创建读取流，传输内容
var file = fs.createReadStream(pathname);
file.on("open", function() {
    file.pipe(res);
});
file.on("error", function(err) {
    console.log(err);
});
```

流读取（read stream）有两个事件：open 和 error。当流准备好的时候触发 open 事件，发生错误时则是 error 事件。该程序在 open 事件的回调函数中调用 pipe 方法。

在这一点上，静态文件服务器的应用如示例 6-1 所示。

示例 6-1　一个简单的静态文件网络服务器

```
var http = require('http'),
     path = require('path'),
     fs    = require('fs'),
     base = '/home/examples/public_html';

http.createServer(function (req, res) {

    pathname = base + req.url;
    console.log(pathname);

    path.exists(pathname, function(exists) {
```

```
    if (!exists) {
      res.writeHead(404);
      res.write('Bad request 404\n');
      res.end();
    } else {
      res.setHeader('Content-Type', 'text/html');
      //Status code:200  --找到文件，无错误;
      res.statusCode = 200;
      //创建读取流，传输内容
      var file = fs.createReadStream(pathname);
      file.on("open", function() {
         file.pipe(res);
      });
      file.on("error", function(err) {
         console.log(err);
      });
    }
  });
}).listen(8124);

console.log('Server running at 8124/');
```

用一个简单的 HTML 文件来进行测试。文件内只有一个 img 元素，检测文件加载以及显示是否正常：

```
<!DOCTYPE html>
<head>
   <title>Test</title>
   <meta charset="utf-8" />
</head>
<body>
<img src="./phoenix5a.png" />
</body>
```

接下来我用另一个 html 文件测试，该文件包含 HTML5 的 video 元素：

```
<!DOCTYPE html>
<head>
   <title>Video</title>
   <meta charset="utf-8" />
</head>
<body>
   <videoid="meadow" controls>
      <source src="videofile.mp4" />
      <source src="videofile.ogv" />
      <source src="videofile.webm" />
   </video>
</body>
```

在 Chrome 中文件可以打开，视频正常播放。但是在 Internet Exploer 10 中，video 无法正常显示。控制台输出错误信息显示了失败的原因：

```
Server running at 8124/
/home/examples/public_html/html5media/chapter1/example2.html
/home/examples/public_html/html5media/chapter1/videofile.mp4
/home/examples/public_html/html5media/chapter1/videofile.ogv
/home/examples/public_html/html5media/chapter1/videofile.webm
```

尽管 IE10 支持 MP4 格式的视频，但是对上述例子中的三个视频文件 response 头部都是 text/html。在我看来，其他的浏览器忽略了错误的头部类型信息并正确显示了媒体元素，但 IE 没有。这是我的猜测，因为我并没有很快发现应用程序中有什么错误。

提示

表面上看，照理服务器端应用程序可以只使用一种浏览器进行测试，但是为什么我们要在所有要求支持的浏览器上对应用程序进行测试，上述例子就是一个完美的解释。

我们需要修改应用程序，可以通过检测文件扩展名在 response 头部文件中返回适当的 MIME 类型。我们可以自己编码写该功能，但是我更推荐使用一个现有的模块：node_mime。

提示

可以用 npm 安装 node_mime: npm install mime。GitHub 地址：https://github.com/broofa/node_mime。

node_mime 模块可以根据给定的文件名（有或者没有路径都可以）返回对应的 MIME 类型，还可以根据给定的类型返回文件扩展名。将 node_mime 添加到 require 列表中：

```
mime = require('mime');
```

返回的类型一方面会用于 response 头文件，另一方面也会在控制台输出，方便我们测试应用程序：

```
//类型
var type = mime.lookup(pathname);
console.log(type);
res.setHeader('Content-Type', type);
```

这样，在 IE10 中的 video 元素加载该文件，视频就能正常播放了。

这样操作的一个问题在于当我们访问的不是一个文件而是一个目录的时候，控制台

会报错，用户看到的网页是空白的：

```
{ [Error: EISDIR, illegal operation on a directory] errno: 28, code 'EISDIR' }
```

这样，我们不仅需要检查访问的资源文件是否存在，还需要检查访问的路径是文件还是一个目录。如果需要访问的是一个目录，要么显示目录中的内容，或者直接返回错误信息——这取决于开发人员。

一个小的静态文件服务器最终版本如示例 6-2 所示，使用 fs.stats 检查请求对象是否存在，是否为文件。如果资源不存在，返回 HTTP 状态码 404。如果资源存在但是为一个目录，返回的 HTTP 状态码 403——禁止访问。在所有情况下，request 都有相应的处理。

示例 6-2　最基本的静态文件服务器最终版本

```
var http = require('http'),
    url = require('url'),
    fs = require('fs'),
    mime = require('mime');
    base = '/home/examples/public_html';

http.createServer(function (req, res) {

   pathname = base + req.url;
   console.log(pathname);

   fs.stat(pathname, function (err, stats) {
      if (err) {
         res.writeHead(404);
         res.write('Bad request 404\n');
         res.end();
      } else if (stats.isFile()) {
         //类型
         var type = mime.lookup(pathname);
         console.log(type);
         res.setHeader('Content-Type', type);

         // 200 status -找到文件，无错误
          res.statusCode = 200;

         // 创建文件流读取
         var file = fs.createReadStream(pathname);
         file.on("open", function () {

            file.pipe(res);
         });
         file.on("error", function (err) {
            console.log(err);
         });
      } else {
```

```
            res.writeHead(403);
            res.write('Directory access is forbidden');
            res.end();
        }
    });
}).listen(8124);
console.log('Server running at 8124/');
```

以下是访问一个带有图片和视频页面时控制台的输出：

```
/home/examples/public_html/html5media/chapter2/example16.html
text/html
/home/examples/public_html/html5media/chapter2/bigbuckposter.jpg
image/jpeg
/home/examples/public_html/html5media/chapter2/butterfly.png
image/png
/home/examples/public_html/favicon.ico
image/x-icon
/home/examples/public_html/html5media/chapter2/videofile.mp4
video/mp4
/home/examples/public_html/html5media/chapter2/videofile.mp4
video/mp4
```

这样就处理好了不同的类型。图 6-1 显示了 Chrome 下显示的包含视频元素的页面以及控制台中显示的对网络资源的访问。

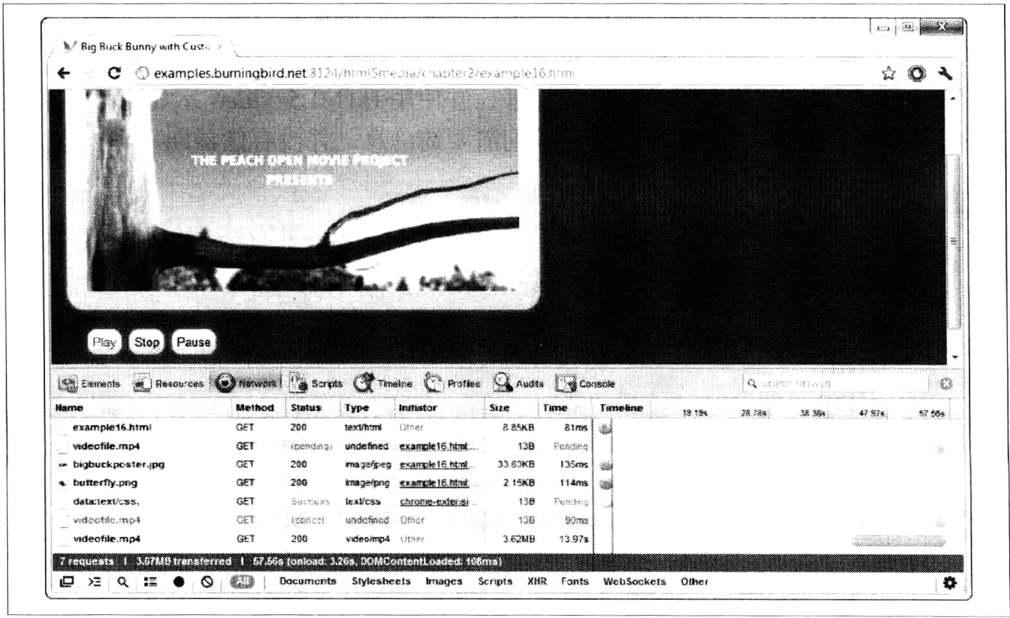


图 6-1　Chrome 加载示例 6-2 中静态文件服务器提供的网页时，控制台显示内容

现在你应该更了解当你加载一个含有视频元素的网页并播放视频的时候数据流是怎么工作的。浏览器以一个可控的速度从流中读取数据，填充内部变量，然后暂停输出。如果在视频播放的时候关闭服务器，视频会继续播放直到当前变量中的内容播放完毕。之后无法从流中获取内容，视频元素就变成空白。很有成就感的是我们付出了一点努力就可以看到各个组件是怎么一起工作的。

为什么不用__dirname 呢？

在本章的一些例子中，我把文件的路径例如/home/examples/public_html 硬编码在代码中。你可能很好奇为什么不直接使用__dirname。

在 Node 中可以使用预先定好的__dirname 作为描述当前 Node 应用工作目录的变量。尽管上述例子中访问的文件独立于 Node 程序，但是你应该对__dirname 以及它在 Node 开发中的用处有一定了解。__dirname 提供了测试程序的方法，而且在部署到生产环境的时候不需要修改文件路径变量的值。

按如下方法使用__dirname:

```
var pathname = __dirname + req.url;
```

注意__dirname 前缀为双下划线。

尽管通过不同类型文件的测试表明程序工作良好，但是它并不完善。很多其他类型的网络请求都没有处理，没有安全和缓存机制，也并没有很好地处理视频请求。我测试了一个网页，使用 HTML video 元素，并使用 HTML5 video 元素的 API 来输出视频加载进度。服务器程序没有获取到需要的信息，并没有像预期一样工作。

提示

第 12 章回顾了这个程序，并讨论了如何添加其他的部分来完成一个功能完善的 HTML5 视频服务器。

创建一个静态文件服务器还有其他许多细节容易犯错，这需要我们了解。另一种实现方式是使用现有的静态文件服务器。在下一节中，我们会学习一种由 Connect 中间件完成的方法。

6.2 中间件

什么是中间件？好问题，但是不幸的是，对这一问题并没有明确的回答。

一般来说，中间件是一个软件，存在于开发人员和底层系统之间。这里说的系统，

既可以是操作系统，也可以是底层的技术支持，比如 Node 提供给我们的。更确切地说，中间件将自己插入应用程序和底层系统的通信链中——因此而得名中间件。

例如，你可以使用中间件来处理通过网络服务器提供静态文件服务的全部必需功能，而无需自己处理。中间件可以处理冗长乏味的数据流，这样你就可以关注应用程序中那些满足独一无二需求的方面。但是，中间件能做的远远不止静态文件的服务。一些中间件提供验证功能、代理、路由、cookie 和 session 管理，以及其他必要的网络技术。

中间件并不是一个可用的函数库或者单纯的一系列函数集合。你所选择的中间件决定了你的程序如何设计及开发。在工作开始前你需要慎重选择所使用的中间件，一旦开发工作进行，替换中间件是件非常痛苦的事情。

目前 Node 应用中两个主流中间件应用程序为：JSGI（JavaScript Gateway Interface，JavaScript 网关接口）和 Connect。JSGI 是针对于一般 JavaScript 的中间件技术，并不特别面向 Node。Node 中可以使用 JSGI-node 模块。而 Connect 则是设计为专门用于 Node 开发的。

提示

JSGI 网站：http://wiki.commonjs.org/wiki/JSGI/Level0/A/Draft2。

JSGI-node GitHub 地址：https://github.com/persvr/jsgi-node。

本书中只介绍 Connect 主要基于以下三个原因。第一，Connect 用法简单。相较而言，JSGI 要求我们花费更多的是理解它是如何工作的（独立于 Node 的用法），而 Connect 则可以立即开始。第二，Connect 提供了对 Express（一个非常流行的架构，第 7 章中会讲到）支持。第三，也是最重要的一点，实践证明了 Connect 是最好的选择。从 npm registry 可以看出 Connect 是使用次数最多的中间件。

提示

关于 Connect 2.0 的介绍：http://tjholowaychuk.com/post/18418627138/connect-2-0。Connect 源代码：https://github.com/senchalabs/Connect。（更多关于安装事宜，参考 112 页的说明栏。）

6.2.1 Connect 基本知识

可以用 npm 安装 Connect：

```
npm install connect
```

事实上，Connect 是一种架构，在该架构中你可以使用一种或多种的中间件。关于

Connect 的文档很少。但是当你看过一些关于 Connect 的例子之后会发现使用 Connect 是件很简单的事情。

> **使用 Alpha 模块**
>
> 在我写本章初稿的时候，npm registry 上 Connect 稳定的版本为 1.8.5，但是我想讲的是开发版本 2.x，因为这很可能是你会使用的版本。
>
> 我从 GitHub 上下载了 Connect 2.x 的源代码，放在我开发环境中 node_modules 目录下。然后进入 Connect 目录，用 npm 安装。但是我并没有给出具体的模块名称，而是用-d 参数来安装所有的依赖：
>
> ```
> npm install -d
> ```
>
> 你可以选择用 npm 从 Git 软件库中直接安装，或者克隆整个版本然后按照我刚描述的方法进行安装。
>
> 要记得一点，如果你从源代码直接安装模块，当运行 npm update 之后，npm 会用自己认为是最新版本的模块覆盖之前的模块——即使事实上被覆盖的那个模块版本更高。

在示例 6-3 中，我使用了 Connect 以及 Connect 自带的两个中间件 [1]connect.logger 和 connect.favicon 创建了一个简单的服务器程序。logger 记录了所有对流的请求(在本例中是 stdio.output 输出流)，favicon 中间件则提供了 favicon.ico 文件的服务。该程序包含在 Connect 请求监听器上使用 use 方法的中间件。Connect 将 request 作为参数传递给 HTTP 对象的 createServer 方法。

示例 6-3　在基于 Connect 的程序中集成 logger 和 favicon 中间件

```
var connect = require('connect');
var http = require('http');

var app = connect()
    .use(connect.favicon())
    .use(connect.logger())
    .use(function(req, res) {
       res.end('Hello World\n');
     });
http.createServer(app).listen(8124);
```

你可以使用任意数量的中间件，不管是 Connect 内建的或者第三方提供的，只需要使用 use 语句进行添加。

[1]. Connect 和 middleware 都指代独立的中间件选项。在本章遵循这一管理。

在示例 6-4 中，我们可以直接在 createServer 方法中使用 Connect 中间件，而不必先创建 Connect request listener。

示例 6-4　直接在程序中加入 Connect 内建的中间件

```
var connect = require('Connect');
var http = require('http');

http.createServer( connect()
   .use(connect.favicon())
   .use(connect.logger())
   .use(function(req, res) {
      res.end('Hello World\n');
})).listen(8124);
```

6.2.2 Connect 中间件

Connect 集成了至少 20 个中间件，在本节中无法一一介绍，但是需要大致说明一下便于你更好地理解它们是如何一起工作的。

提示

如何使用 Connect 中间件的其他例子在第 7 章中创建的 Express 应用程序中。

connect.static

早前我们曾从头做起，创建了一个简单的静态文件服务器。但是 Connect 提供了中间件可以完成该服务器功能，并提供更多其他功能。这使用起来非常简单，只需要设置 connect.static 中间件选项，传递给所有请求的根目录。下面这段代码基本实现了示例 6-2 的功能，但是代码更简洁：

```
var  connect = require('connect'),
     http = require('http'),
     __dirname = '/home/examples';

http.createServer( connect()
     .use(connect.logger())
     .use(connect.static(__dirname + 'public_html'), {redirect:true})
     ).listen(8124);
```

connect.static 将根目录作为第一个参数，第二个参数可选。第二个参数可以有以下选项：

maxAge

　浏览器缓存，毫秒为单位。默认为 0。

hidden

设置为 true 时允许传递隐藏文件，默认为 false。

redirect

设置为 true 时允许重定向，路径名为目录。

这个简短的 Connect 应用程序显示了与之前程序在行为上的很大差异。Connect 解决方式处理了浏览器缓存，保护程序免受非法 URL 入侵，以及更好地处理了 HTTP HTML5 视频元素，这在之前的服务器程序中并没有完成。与之前的服务器程序相比，这种解决方式的唯一缺点在于对错误的处理较少。但是 connect.static 确实为浏览器提供了不同的 response 和状态码。

这段代码以及本节之前的例子都用到了另一个中间件：connect.logger，接下来我们就要介绍这个中间件。

Connect.logger

logger 中间件模块记录对流的请求，默认输出到 stdout。你可以修改输出流和其他一些选项，包括缓冲时间、样式，还有 immediate 标识，决定立即写入 log 或者在响应时写入。

除了四个预先定义好的样式，还有一些标识用于定义样式：

```
    default
        ':remote-addr - - [:date] ":method :url HTTP/:http-version" :status :res[con
tent-length] ":referrer" ":user-agent"'
    short
        ':remote-addr - :method :url HTTP/:http-version :status :res[content-
length] - :response-time ms'
    tiny
        ':method :url :status :res[content-length] - :response-time ms'
    dev
```

开发人员在使用的时候，根据 response 状态确定输出颜色。

默认的 log 样式如下所示：

```
99.28.217.189 - - [Sat, 25 Feb 2012 02:18:22 GMT] "GET /example1.html
HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11
(KHTML, like Gecko)
Chrome/17.0.963.56 Safari/535.11"
99.28.217.189--[Sat, 25 Feb 2012 02:18:22 GMT] "GET /phoenix5a.png HTTP/1.1" 304
- http://examples.burningbird.net:8124/example1.html
```

```
"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.56 Safari/535.11"
99.28.217.189 - - [Sat, 25 Feb 2012 02:18:22 GMT] "GET /favicon.ico
HTTP/1.1"
304 - "-" "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML,
like Gecko)
Chrome/17.0.963.56 Safari/535.11"
99.28.217.189 - - [Sat, 25 Feb 2012 02:18:28 GMT]
"GET /html5media/chapter2/example16.html HTTP/1.1" 304 - "-"
"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.56 Safari/535.11"
```

这种输出信息量很大，但同时冗余信息太多，与我们从类似于 Apache 这样的服务器中得到的 log 样式类似。你可以修改 log 的样式或者将输出重定向到文件中。示例 6-5 使用 connect.logger，将 log 重定向到文件中，并设置预定义的 dev 样式。

示例 6-5　将 log 写入文件并改变样式

```
var connect = require('connect'),
    http = require('http'),
    fs = require('fs'),
    __dirname = '/home/examples';

var writeStream = fs.createWriteStream('./log.txt',
     {'flags':'a',
      'encoding':'utf8',
     'mode':0666});
http.createServer(connect()
   .use(connect.logger({format:'dev', stream:writeStream }))
   .use(connect.static(__dirname + '/public_html'))
   ).listen(8124);
```

现在的 log 输出为：

```
GET /example1.html 304 4ms
GET /phoenix5a.png 304 1ms
GET /favicon.ico 304 1ms
GET /html5media/chapter2/example16.html 304 2ms
GET /html5media/chapter2/bigbuckposter.jpg 304 1ms
GET /html5media/chapter2/butterfly.png 304 1ms
GET /html5media/chapter2/example1.html 304 1ms
GET /html5media/chapter2/bigbuckposter.png 304 0ms
GET /html5media/chapter2/videofile.mp4 304 0ms
```

虽然信息量减少了，但这种方式更便于查看请求状态以及加载时间的方式。

connect.parseCookie 和 connect.cookieSession

之前我们自己创建的文件服务器并没有提供对 HTTP cookies 和 session 状态功能的支持。幸运的是，Connect 中间件替我们做了这两件事情。

可能出现的情况的是，你的第一个 JavaScript 应用程序用于创建一个 HTTP request cookie。connect.parseCookie 中间件提供了功能上的支持，允许我们访问服务器上的 cookie 数据。它解析 cookie 头部，用 cookie/data 得到 req.cookies。示例 6-6 为一个简单的网络服务器，根据 username 从 cookie 中提取信息，在 stdout 中写入相关信息。

示例 6-6　访问 HTTP request cookie，用于 console 信息

```
var connect = require('connect')
    ,http = require('http');
var app = connect()
   .use(connect.logger('dev'))
   .use(connect.cookieParser())
   .use(function(req, res, next) {
       console.log('tracking ' + req.cookies.username);
       next();
     })
   .use(connect.static('/home/examples/public_html'));

http.createServer(app).listen(8124);
console.log('Server listening on port 8124');
```

在 6.2.3 小节中我会介绍匿名函数的用法，尤其是 next 函数。现在注意力集中在 connect.cookieParser 上，可以看到这个中间件解析收到的请求，从 header 中读取 cookie 的数据并存储在 request 对象中。之后的匿名函数通过 cookies 对象访问 username 数据，输出到控制台。

为了创建一个 HTTP response cookie，我们配合 connect.parseCookie 使用 connect.cookieSession。connect.cookieSession 提供安全的会话持久性。文本内容被当做字符串传递给 connect.cookieParser 函数，为 session 数据提供安全码。session 数据直接添加到 session 对象中。清除 session 数据的时候，设置该 session 对象为 null。

示例 6-7　使用 session cookie 来跟踪资源访问

```
var connect = require('connect')
    ,http = require('http');
// 清除 session 数据
function clearSession(req, res, next) {
   if ('/clear' == req.url) {
       req.session = null;
       res.statusCode = 302;
       res.setHeader('Location', '/');
       res.end();
   } else {
       next();
   }
}
// 跟踪用户
```

```
function trackUser(req, res, next) {
  req.session.ct = req.session.ct || 0;
  req.session.username = req.session.username || req.cookies.username;
  console.log(req.cookies.username + ' requested ' +
                    req.session.ct++ + ' resources this session');
  next()
}

// cookie 和 session
var app = connect()
  .use(connect.logger('dev'))
  .use(connect.cookieParser('mumble'))
  .use(connect.cookieSession({key:'tracking'}))
  .use(clearSession)
  .use(trackUser);

// 静态服务器
app.use(connect.static('/home/examples/public_html'));
// 启动服务器开始监听
http.createServer(app).listen(8124);
console.log('Server listening on port 8124');
```

图 6-2 显示了通过示例 6-8 的服务器应用访问到的网页。打开 JavaScript 控制台可以看到 request 和 response 的 cookie。注意到 response cookie 和 request 不一样，是经过加密的。

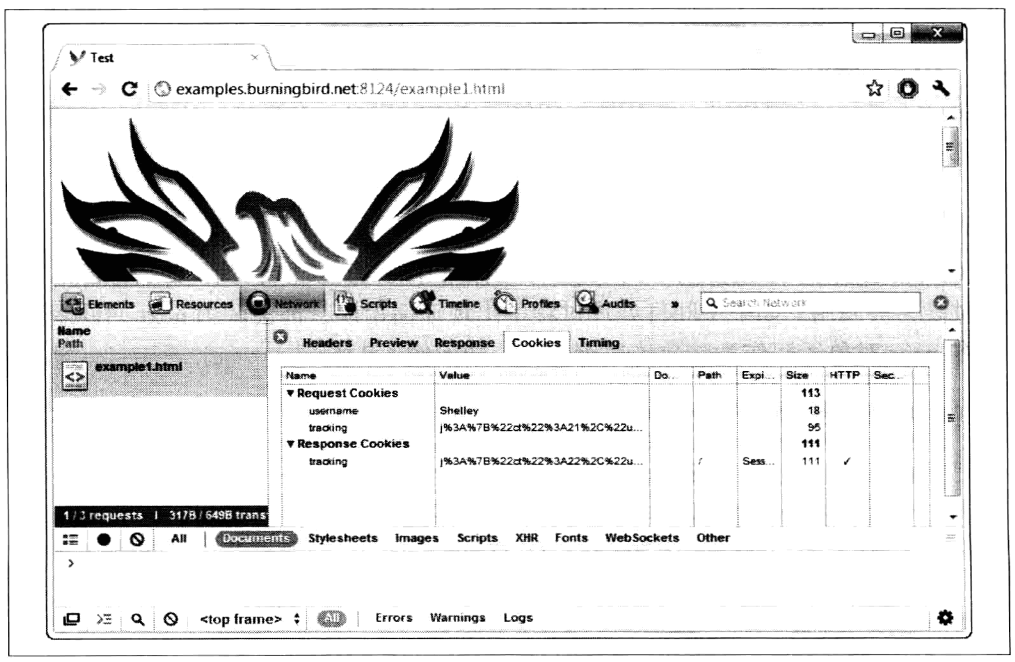


图 6-2　Chrome 浏览器，JavaScript 控制台，显示 request cookie 和 response cookie

用户访问的文件数目可以持续跟踪，直到用户访问/clear URL（在这种情况下

session 对象被设置为 null）或者关闭浏览器，session 终止。

示例 6-7 中也使用了其他一些自定义的中间件函数。在下一个关于 Connect 的章节（也是最后一节）中，我们会讨论这些函数如何与 Connect 交互以及如何创建第三方的中间件。

6.2.3 定制 Connect 中间件

在上一小节的示例 6-7 中，我们创建了两个函数作为 Connect 中间件的一部分在 request 到达最终服务器之前进行处理。传递给该函数的三个参数为 HTTP request、HTTP response 和回调函数 next。这三个参数构成了一个 Connect 中间件函数的签名。

为了更深入地了解 Connect 中间件是如何工作的，我们来看看一个在之前代码中曾使用过的 connect.favicon。这个函数的功能很简单，就是一个简单的函数，提供默认的 favicon 或者提供路径设置自定义的 favicon：

```
connect()
.use ( connect.favicon('someotherloc/favicon.ico'))
```

了解 connect.favicon 工作原理的原因除了它的用处之外，主要是因为它是最简单的中间件，很容易还原工程。

connect.favicon 的源代码，尤其是与其他源代码对比之后会发现：所有导出的中间件都返回一个如下最简签名的函数：

```
return function(req, res, next)
```

next 回调函数作为函数的第三个参数 ，当中间件没有处理当前请求或者处理中断的时候会被调用。如果中间件处理过程中出现任何错误，都会调用 next 回调函数，并将 error 对象返回作为参数传递给 next，如示例 6-8 所示。

示例 6-8 favicon Connect 中间件

```
module.exports = function favicon(path, options) {
  var options = options || {}
    , path = path || __dirname + '/../public/favicon.ico'
    , maxAge = options.maxAge || 86400000;

  return function favicon(req, res, next) {
    if ('/favicon.ico' == req.url) {
      if (icon) {
        res.writeHead(200, icon.headers);
        res.end(icon.body);
      } else {
```

```
            fs.readFile(path, function (err, buf) {
               if (err) return next(err);
                  icon = {
                    headers:{
                         'Content-Type':'image/x-icon'
                         ,'Content-Length':buf.length
                         ,'ETag':'"' + utils.md5(buf) + '"'
                         ,'Cache-Control':'public, max-age=' + (maxAge / 1000)
                    },
                    body:buf
                  };
                  res.writeHead(200, icon.headers);
                  res.end(icon.body);
                });
             }
          } else {
             next();
          }
       };
    };
```

next 回调函数调用方法就是链式函数的调用方案，按照序列进行。在收到的 request 请求中，如果中间件对 request 处理完成，比如请求 favicon 的 request，不会有更多的中间件被调用。这就是为什么在你的应用程序中要在 connect.logger 之前引入 connect.favicon——为了防止对 favicon.ico 的请求造成 log 的混乱。

```
http.createServer(connect()
    .use(connect.favicon('/public_html/favicon.ico'))
    .use(connect.logger())
    .use(connect.static(_dirname + '/public_html'))
    ).listen(8124);
```

你已经学会了如何在程序之中创建自定义的 Connect 中间件以及 Connect 内建中间件应该是什么样子的，但是如何创建一个第三方的中间件而不是直接嵌入在程序中呢？

创建第三方 Connect 中间件以及其他任何你需要的模块，但要确保它包含 Connect 所需要的部分——描述三个参数（req、res、next），如果对请求的处理中断则调用 next 函数。

示例 6-9 创建了一个 Connect 中间件，用于检查请求的文件是否存在以及判断请求内容是否是个文件（而不是目录）。如果请求内容为一个目录，则返回 403 状态码以及自定义的错误信息。如果没有上述情况发生，则调用 next 触发 connect 中间件，来唤醒下一个函数（本例中为 connect.static）。

示例 6-9　创建一个定制的错误处理中间件模块

```
var fs = require('fs');

module.exports = function customHandler(path, missingmsg, directorymsg) {
   if (arguments.length < 3) throw new Error('missing parameter in customHandler');
   return function customHandler(req, res, next) {
      var pathname = path + req.url;
      console.log(pathname);
      fs.stat(pathname, function (err, stats) {
         if (err) {
            res.writeHead(404);
            res.write(missingmsg);
            res.end();
         } else if (!stats.isFile()) {
            res.writeHead(403);
            res.write(directorymsg);
            res.end();
         } else {
            next();
         }
      });
   }
}
```

定制的 Connect 中间件在错误发生时抛出错误信息，如果错误发生在返回函数的执行过程中，调用 next 时会传入 error 对象：

```
next(err);
```

以下代码告诉我们如何在程序中使用自定义的中间件：

```
var connect = require('connect'),
    http = require('http'),
    fs = require('fs'),
    custom = require('./custom'),
    base = '/home/examples/public_html';

http.createServer(connect()
    .use(connect.favicon(base + '/favicon.ico'))
    .use(connect.logger())
    .use(custom(base + '/public_html', '404 File Not Found',
            '403 Directory Access Forbidden'))
    .use(connect.static(base))
    ).listen(8124);
```

Connect 中间件有自己的 errorHandler 函数，但是它并不能达到我们的目标，因为它的目的是提供一个统一格式的异常信息输出。在第 7 章中你可以看到该方法在 Express 程序中的使用。

还有其他一些与 Connect 绑定的中间件和数量巨大的兼容 Connect 的第三方中间件。在第 7 章中，会进一步介绍在 Express 应用架构中 Connect 的中间件层的角色。在这之前，我们先来看两个其他类型的服务，这两个服务对很多 Node 应用都是必需的：routers 和 proxies。

6.3 Routers

路由是指从一个源接收转发到另一个地方。一般来说，转发的内容都是数据包，但是从应用层面来看，也可以是一个对资源的请求。

如果你用过 Drupal 或者 WordPress，你已经见过了路由是如何工作的。如果没有任何的 URL 重定向，用户访问一篇文章的链接应该是：

http://yourplace.org/node/174

而不是：

http://yourplace.org/article/your-title

http://yourplace.org/node/174 这个 URL 是路由工作的一个例子。本例中 URL 提供了关于网络程序如何处理链接的信息：

- 访问 node 数据库（这里的 node 是 Drupal node）；
- 查找并显示 id 为 174 的 node。

另一个例子为：

http://yourplace.org/user/3

同样，访问用户数据库，查找并显示 id 为 3 的用户。

在 Node 中，路由的基本作用在于从 URI 中获取我们需要的信息（通常按照某种模式获取），用这些信息触发正确的处理过程，并将这些信息传递给该处理过程。

对 Node 开发人员来说有好几种路由可以使用，包括 Express 内建的。但是接下来我要介绍的是一种更通用的：Crossroads。

提示

Crossroads 官网：http://millermedeiros.github.com/crossroads.js。

使用 npm 安装 Crossroads:

```
npm install crossroads
```

该模块提供了大量以及文献完整的 API，但是我主要关注在以下三个方法：

addRoute

　　定义一个新模式的路由监听。

parse

　　解析字符串，并将符合的字符串转发到正确的路由。

matched.add

　　将路由处理与路由做映射。

我们用正则表达式定义一个路由。该正则表达式包含花括号（{}），由此界定传递给路由处理函数的命名变量。比如，以下两个路由模式：

```
{type}/{id}
node/{id}
```

都可以匹配：

http://something.org/node/174

两者的差异在于，type 在第一个模式中被作为参数传递给处理函数，而第二个模式中 node 不是参数。

可以使用冒号（:）表示可选择的部分。如下：

```
category/:type:/:id:
```

会匹配：

category/

category/tech/

category/history/143

将 request 传给 parse 函数，触发路由处理的过程：

```
parse(request);
```

如果 request 与任一个现有的路由处理函数匹配，该函数将被调用。

示例 6-10 创建了一个简单的程序用于查找任意给定的分类以及出版物。程序按照 request 中的要求将结果输出到控制台。

示例 6-10　使用 Crossroads 将 URL 请求定位到具体的操作

```
var crossroads = require('crossroads'),
    http = require('http');

crossroads.addRoute('/category/{type}/:pub:/:id:', function (type, pub, id) {
   if (!id && !pub) {
     console.log('Accessing all entries of category ' + type);
     return;
   } else if (!id) {
     console.log('Accessing all entries of category ' + type +
                 ' and pub ' + pub);
     return;
   } else {
     console.log('Accessing item ' + id + ' of pub ' + pub +
                 ' of category ' + type);
   }
});
http.createServer(function (req, res) {

   crossroads.parse(req.url);
   res.end('and that\'s all\n');
}).listen(8124);
```

针对以下这些请求：

```
http://examples.burningbird.net:8124/category/history
http://examples.burningbird.net:8124/category/history/journal
http://examples.burningbird.net:8124/category/history/journal/174
```

生成的信息输出为：

```
Accessing all entries of category history
Accessing all entries of category history and pub journal
Accessing item 174 of pub journal of category history
```

为了匹配类似 Drupal 这样，对象具有类型和 id 组合，示例 6-11 使用了 Crossroads 另一个方法：matched.add，来将路由处理映射到一个具体的路径上。

示例 6-11　根据给定的路径映射到路由处理函数

```
var crossroads = require('crossroads'),
```

```
    http = require('http');

var typeRoute = crossroads.addRoute('/{type}/{id}');

function onTypeAccess(type,id) {
  console.log('access ' + type + ' ' + id);
 };
typeRoute.matched.add(onTypeAccess);
http.createServer(function(req,res) {
   crossroads.parse(req.url);
   res.end('processing');
}).listen(8124);
```

该段程序对以下两个请求都能匹配:

/node/174

/user/3

Router 一般用于访问数据库来生成返回的页面内容，也可以和中间件或者架构模块一起使用来处理收到的请求，尽管这些程序可能提供了自己的路由功能。在下一节中会介绍 Crossroads 与 Connect 和 Proxy 的配合。

6.4 Proxies

代理（Proxy）是一种路由请求方式，将不同源的请求通过同一个服务器处理，原因可能有很多：缓存、安全，甚至是故意模糊请求的来源。例如，公开访问代理会被用于解决一些人对一些网站内容的限制访问问题，它会使从一些请求的发起方看起来并不是原来的来源。这类型的代理被称为转发代理。

还有一种被称为反向代理，用于控制请求如何被发送到服务器。例如，现在有五个服务器，但是有四个不希望有用户直接访问。因而，将所有的请求转发到第五个服务器，然后再代理给其他服务器。反向代理也被用于平衡负载和通过缓存请求改进系统的整体表现。

提示

代理的另一个用法是将一个局部的服务暴露给云服务。这类型的代理称为 Reddish 代理，该代理将本地的 Redis 实例暴露给新的 Reddish 服务 https://reddi.sh/。

在 Node 中最受欢迎的代理模块为 http-proxy。该模块提供了所有我能想到以及想不到的代理功能。http-proxy 提供了转发以及反向代理功能，可以用于 WebSockets，

支持 HTTPS 以及处理延时。有名的 nodejitsu.com 也使用该模块，所以就如开发者声明的一样，该模块久经考验。

提示

http-proxy GitHub 网页：https://github.com/nodejitsu/node-http-proxy。

用 npm 安装 http-proxy:

```
npm install http-proxy
```

http-proxy 最简单的用法是创建一个独立的代理服务器，在某端口上监听请求，代理给监听在另一个端口的网络服务器：

```
var http = require('http'),
    httpProxy = require('http-proxy');

httpProxy.createServer(8124, 'localhost').listen(8000);

http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.write('request successfully proxied!' + '\n' + JSON.stringify(req.headers, true,
  2));
  res.end();
}).listen(8124);
```

这段程序所做的全部功能就是在 8000 端口上监听请求，并将收到的请求代理到 8124 端口的 HTTP 服务器。

我的系统上运行这段程序输出到浏览器的内容为：

```
request successfully proxied!
{
  "host":"examples.burningbird.net:8000",
  "connection":"keep-alive",
  "user-agent":"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11
  (KHTML, like Gecko) Chrome/17.0.963.56 Safari/535.11",
  "accept":"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "accept-encoding":"gzip,deflate,sdch",
  "accept-language":"en-US,en;q=0.8",
  "accept-charset":"ISO-8859-1,utf-8;q=0.7,*;q=0.3",
  "cookie":"username=Shelley",
  "x-forwarded-for":"99.190.71.234",
  "x-forwarded-port":"54344",
  "x-forwarded-proto":"http"
}
```

本例中使用代理的内容在输出中用黑体标出。注意到一点，之前的例子中的 request cookie 依然显示出来了。

也可以在命令行使用 http-proxy。在 bin 目录中有一个命令行程序、接收端口、目标地址、配置文件、配置代理日记输出的标志位作为参数，还可以使用-h 参数显示帮助内容。监听 8000 端口并将请求转发给 localhost 8124 端口，代码如下：

```
./node-http-proxy --port 8000 --target localhost:8124
```

这已经够简单了。如果你希望在后台执行这段代理程序，在末尾加上符号（&）。

稍后我会在本书中介绍 http-proxy 与 WebSockets、HTTPS 配合使用，但现在，我们先来总结一下本章中用到的所有技术——静态文件服务器、Connect 中间件、Crossroads 路由，以及 http-proxy 代理。来创建最后一个例子，你可以尝试综合使用全部这些技术。

示例 6-12 中会使用 http-proxy 测试动态收到的请求（请求的 URL 都以/node/开始）。如果找到匹配，路由会将该请求转发给服务器，服务器会使用 Crossroads 解析相关的数据。如果请求的不是一个动态资源，代理会将请求转发给静态文件服务器，该服务器会用到 Connect 中间件，包括 logger、favicon 和 static。

示例 6-12　使用 Connect、Crossroads 和 http-proxy 处理动态和静态的请求

```
var connect = require('connect'),
    http = require('http'),
    fs = require('fs'),
    crossroads = require('crossroads'),
    httpProxy = require('http-proxy'),
    base = '/home/examples/public_html';

//创建代理，监听请求
httpProxy.createServer(function(req,res,proxy) {

   if (req.url.match(/^\/node\//))
      proxy.proxyRequest(req, res, {
        host: 'localhost',
        port: 8000
      });
   else
      proxy.proxyRequest(req,res, {
        host: 'localhost',
        port: 8124
      });
}).listen(9000);
```

```
//对动态资源的请求添加路由
crossroads.addRoute('/node/{id}/', function(id) {
  console.log('accessed node ' + id);
});

//动态文件服务器
http.createServer(function (req, res) {
  crossroads.parse(req.url);
  res.end('that\'s all!');
}).listen(8000);

//静态文件服务器
http.createServer(connect()
   .use(connect.favicon())
   .use(connect.logger('dev'))
   .use(connect.static(base))
).listen(8124);
```

用以下 URL 请求测试该服务器：

```
/node/345
/example1.html
/node/800
/html5media/chapter2/example14.html
```

控制台输出结果以及浏览器收到的响应如下所示：

```
accessed node 345
GET /example1.html 304 3ms
GET /phoenix5a.png 304 1ms
accessed node 800
GET /html5media/chapter2/example14.html 304 1ms
GET /html5media/chapter2/bigbuckposter.jpg 304 1ms
```

不能说我们已经完成了一半的 CMS（内容管理系统）系统，但是如果我们希望构建一个 CMS 系统的话已经拥有了需要的工具。但是既然可以使用 Node 支持的架构（下一章会讲到）我们为什么还要自己构建呢？

第7章

Express 框架

框架软件能够提供基础设施支持，使我们能够更迅速地创建网站和应用程序。框架通常还提供了一个构建骨架，实现了许多在开发中常见但又必须实现的代码，因而我们可以更专注于创建我们的应用程序或站点所真正需要的功能。它还能让我们的代码更加紧凑，以使代码更易于管理和维护。

我们已经交替使用过框架和库的概念，因为在各种应用程序的开发过程中，两者都为开发者提供了可重用的功能。两者也都能降低代码的耦合度，但它们也有不同，因为框架通常还提供了一个基础设施，可以影响应用程序的整体设计。

Node.js 中有一些非常完善的框架，包括 Connect（在第 6 章介绍过），尽管在我看来 Connect 更像是一个中间件而非框架。如果综合考虑框架的支持、功能完善度以及流行度的话，有两个 Node 框架是比较出色的：Express 和 Geddy。如果你在网上搜索两者之间的差异，人们可能会说 Express 更像 Sinatra 一些，而 Geddy 更像是 Rails。这意味如果用非 Ruby 术语来说，Geddy 基于 MVC（Model-View-Controller），而 Express 则更多使用 REST 风格（后面的章节对此有更详细的说明）。

此外还有一个新来的小子——Flatiron，以前它作为一些独立组件存在，但现在已经被整合在一起而形成一个产品。另一个在 node 工具网站上流行的框架是 Ember.js，原名 SproutCore 2.0。它是除了 CoreJS 以外的另一个基于 MVC 的框架。

我曾试图在一个章节中对上述每个框架都做下介绍，但事实上能够在单独一个章节介绍清楚一个框架都是很困难的，更不用说几个了。因此，我决定在本章着重介绍 Express 框架。虽然其他框架也都非常出色，但我喜欢 Express 的开放性和扩展性，并且它也是目前最流行的框架。

提示

Geddy.js 的网站地址是 *http://geddyjs.org/*。Flatiron 可以在 *http://flatironjs.org/*找到，Ember.js 在 Github 上的页面地址是 *https://github.com/emberjs/ember.js*，而 CoreJS 的主站是 *http://echo.nextapp.com/site/corejs*。Express 的 GitHub 页面在 *https://github.com/visionmedia/express*。Express 的文档放在 *http://expressjs*。

7.1 Express：启动和运行

安装 Express 很容易，使用以下 npm 命令即可：

```
npm install express
```

为了快速上手 Express，最好通过命令行工具来生成一个 Express 应用程序。既然目前我们还不知道想在程序中实现些什么东西，那么干脆就在一个干净的目录中创建并运行程序，而不要在一个保存了各种有用资料的目录中创建程序。

我给这个新应用程序起了一个足够简单的名字 site：

```
express site
```

该命令会生成几个目录：

```
create : site
  create : site/package.json
  create : site/app.js
  create : site/public
  create : site/public/javascripts
  create : site/public/images
  create : site/routes
  create : site/routes/index.js
  create : site/public/stylesheets
  create : site/public/stylesheets/style.css
  create : site/views
  create : site/views/layout.jade
  create : site/views/index.jade
```

同时你还会看到一条有用的提示信息，它要求你进入到 site 目录并运行 npm 命令：

```
npm install -d
```

一旦执行该命令安装应用程序后，就可以使用 node 运行 app.js 文件了：

```
node app.js
```

它会在端口 3000 启动一个服务端应用程序。如果访问该应用程序，则会显示一个包含如下信息的网页：

```
Express
Welcome to Express
```

至此，你已经创建了第一个 Express 应用程序。现在，让我们来看看是否还能用它来做更多有趣的事情。

7.2 app.js 文件

示例 7-1 显示了我们刚运行过的 app.js 文件的源代码。

示例 7-1 app.js 文件的源代码

```
/*
 * Module dependencies.
 */

var express = require('express')
  , routes = require('./routes')
  , http = require('http');

var app = express();

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
});


app.configure('development', function(){
  app.use(express.errorHandler());
});

app.get('/', routes.index);

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

从代码的开始部位可以看出程序包含了三个模块——Express、Node 的 HTTP 以及一个刚刚生成的模块 routes。在 routes 子目录中，index.js 文件包含如下代码：

```
/*
```

```
 * GET home page.
 */
exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

从代码中可以轻易看出，我们使用了 Express 中 response 对象的 render 方法来渲染指定视图，调用该方法时，还可以指定一组选项用于渲染，此处我们指定了 title 值为“Express”。我将在 7.5 节对此进行介绍。

现在，让我们再回到 app.js。在包含了所有需要的模块后，它又创建了一个 Express 对象实例，然后通过调用 configure 方法并传递一组可选参数来配置它（在下面“设置应用程序模式”说明栏中对 configure 有更多介绍）。configure 方法的第一参数可以是一个可选参数，它用于指明当前的配置项是否针对特定环境（例如 development 或 production）。当没有通过该参数指定环境类型时，配置项会被应用到所有环境。app.js 中对 configure 的第二次调用指定了 development 环境。如果愿意的话，你可以为每一个可能的环境类型分别调用 configure 方法。与当前环境类型匹配的配置项将会被处理。

设置应用程序模式

在 Express 应用程序中，我们可以使用 configure 方法来为应用程序支持的任何模式指定其所使用的中间件、配置项以及可选项。在下面的例子中，该方法会将指定的配置项应用到所有模式：

```
app.config(function() { ... }
```

而下面对 configure 方法的调用则保证了配置项仅被应用到 developmen 模式：

```
app.config('development', function() { ... }
```

你可以通过环境变量 NODE_ENV 来为程序指定模式：

```
$ export NODE_ENV=production
```

或是：

```
$ export NODE_ENV=ourproduction
```

你可以使用你想要的任何名称。默认情况下，该环境变量值为 development。

为了确保应用程序始终运行在指定模式下，可以在用户配置文件中添加并导出 NODE_ENV 环境变量。

Configure 方法的第二个参数是一个匿名函数，包含了对几个中间件的引用。这与我们在第 6 章中使用 Connect 中间件的方式类似。其实，这并不是偶然现象，因为

Connect 和 Express 的主要作者是同一个人：TJ Holowaychuk。不一样的是这里对 app.set 方法的调用发生了两次。

app.set 方法用于设置各种配置项变量，例如程序视图的存储位置：

```
app.set('views', __dirname + '/views');
```

以及使用的视图引擎（这里使用 Jade）：

```
app.set('view engine', 'jade');
```

在 app.js 文件中，还使用了 favicon、logger 和 static 等中间件，这些就无需进一步解释了：

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.static(__dirname + '/public'));
```

其实，我们还可以在创建服务器的时候调用 Express 中间件：

```
var app = express.createServer(
    express.logger(),
    express.bodyParts()
);
```

不管使用哪种方法都是可以的。

接下来，在生成的代码中，我们还会看到程序使用了下面三个中间件/框架组件：

```
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
```

bodyParser 中间件与其他中间件一样也直接来自 Connect。Express 所做的仅仅是重用它。

我在前面的章节中介绍了 logger、favicon 以及 static，但是没有介绍 bodyParse。这个中间件会解析传入的请求内容，将其转换为一个 request 对象的属性。Express 的 methodOverride 也是来自 Connect，它能通过表单中的一个隐藏域_method 来实现对 Full REST 的支持。

提示

支持 Full REST（表述性状态传输）意味着不但要支持 HTTP GET 和 POST 操作，还要支持 HTTP 的 PUT 和 DELETE 操作。我们会在 7.5.2 小节对此进行讨论。

最后的配置项是 app.router。这个中间件是可选的，它包含了所有已定义的路由信息并能根据任何给定的路由来执行查找。如果没有定义路由信息，那么当 app.get 或者 app.post 等相关方法第一次被调用时，路由信息会自动生成。

与 Connect 一样，Express 中间件的调用顺序也是非常重要的。favicon 中间件需要在 logger 之前调用，因为我们不希望日志中包含一大堆访问 favicon.ico 的信息。static 中间件应该在 bodyParser 以及 methodOverride 之前被包含，因为处理静态页面时使用这两个中间件是没有意义的。Express 应用程序对表单的处理是动态的，不适用于静态页面。

提示

在 7.4 节对 Express/Connect 有更多的介绍。

第二次对 configure 方法的调用指定了 development 模式，以便添加 errorHandler 支持。接下来我将介绍 errorHandler 以及其他一些用于错误处理的相关技术。

7.3 错误处理

Express 提供了错误处理功能，同样也是使用 Connect 的 errorHandler 中间件。

Connect 的 errorHandler 为我们提供了一种处理异常的方法。当异常产生时，它能让我们更好地了解并处理它。可以像包含其他中间件一样使用如下命令来包含 errorHandler：

```
app.use(express.errorHandler());
```

可以使用 dumpExceptions 标志将异常信息导出到 stderr：

```
app.use(express.errorHandler({dumpExceptions : true }));
```

也还可以使用 showStack 为异常生成 HTML 描述：

```
app.use(express.errorHandler({showStack : true; dumpExceptions : true}));
```

再重申下：这种错误捕获方法仅仅是给开发阶段使用的，我们绝对不希望真实用户看到这些异常信息。然而，我们却需要为另一些情况提供有效的错误处理，比如无法找到页面时，或者用户试图访问一个受限的子目录时。

这里有一种方法可以让我们达到该目的，那就是使用一个自定义匿名函数来作为中间件，并将该中间件设置为处理序列中的最后一个中间件。如果没有其他中间件需

要再处理当前请求的话，请求应该可以被正常地传递到最后一个中间件。

```
app.configure(function(){
  app.use(express.favicon());
  app.use(express. logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express. bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(function(req, res, next){
    res.send('Sorry ' + req.url + ' does not exist');
  });
});
```

在下一章中，我们将使用模板来渲染 response 对象并生成一个漂亮的 404 页面。

我们还可以使用另外一种方式来捕获程序抛出的异常并做相应的处理。在 Express 文档中，将其称为 app.error，但在我写这本书的时候它似乎还没有实现。不管怎样，通过签名我们可以了解到它是一个包含 4 个参数的函数：error、request、response 和 next。

我添加了第二个用于处理错误的中间件函数，同时修改了之前的 404 中间件函数，使其抛出错误而不是直接处理错误：

```
app.configure(function(){
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(function(req, res, next){
    throw new Error(req.url + ' not found');
  });
  app.use(function(err, req, res, next) {
    console.log(err);
    res.send(err.message);
  });
});
```

现在我可以在同一个函数中处理 404 以及其他错误了。当然，我还可以使用模板来生成一个更具吸引力的页面。

7.4 Express 与 Connect 的关系

纵观本章，到目前为止我们已经看到过 Express 和 Connect 的协作关系。Express 通

过 Connect 获得了很多基本功能。

例如，可以使用 Connect 中间件 cookieParser、cookieSession 和 session 实现 Express 对会话的支持。但要记住的是必须使用 Express 版本的中间件：

```
app.use(express.cookieParser('mumble'))
app.use(express.cookieSession({key : 'tracking'}))
```

也可以通过 staticCache 中间件来支持对静态文件的缓存功能：

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.staticCache());
app.use(express.static(__dirname + '/public'));
```

默认情况下，缓存模块最多可以保存 128 个对象，每个对象最大 256 KB 空间，一共大约 32 MB。可以通过 maxObjects 和 maxObjects 两个选项来调整这些限制：

```
app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
```

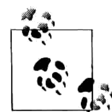
使用 directory 中间件美化目录清单：

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
app.use(express.directory(__dirname + '/public'));
app.use(express.static(__dirname + '/public'));
```

如果在使用 express.directory 的同时还使用 routing，务必保证 directory 中间件在 app.router 中间件之后，否则它可能与路由产生冲突。

一个好的经验法则：将 express.directory 放到其他中间件之后，任何错误处理之前。

express.directory 中间件还支持一些可选项，包括是否显示隐藏文件（默认为 false），是否显示图标（默认为 false），以及一个过滤器。



提示

也可以在 Express 中使用 Connect 的第三方中间件。不过，当与路由一起使用时需要谨慎。

现在是时候看看 Express 的关键组件：路由。

7.5 路由

所有 Node 核心框架，事实上也包括许多现代框架，都实现了路由的概念。我在第

6 章介绍过一个独立的路由模块，并描述了如何使用它从 URL 中提取服务请求。

Express 是基于 HTTP 的 GET、PUT、DELETE 以及 POST 来管理路由的，同时还提供了对应名字的方法，例如用 app.get 表示 GET，用 app.post 表示 POST。在一般的应用程序中，如示例 7-1 所示，app.get 方法被用来访问程序的根（'/'），该方法同时还接收一个请求监听器以处理请求数据，本例中 routes.index 函数就是个监听器。

routes.index 函数很简单：

```
exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```

它调用了 resource 对象上的 render 方法。render 方法需要提供模板文件名称。由于应用程序已经确定了视图引擎：

```
app.set('view engine', 'jade');
```

所以我们没有必要提供文件扩展名。当然，也可以使用扩展名：

```
res.render('index.jade', { title: 'Express' });
```

可以在另一个由 Express 生成的文件夹 views 中找到模板文件。它有两个文件：index.jade 和 layout.jade。index.jade 是在 render 方法中引用的模板文件，内容如下：

```
extends layout
block content
  h1= title
  p Welcome to #{title}
```

文档内容是一个包含有标题内容的 H1 元素，还有一个包含了 title 变量值的段落元素。layout.jade 模板提供了文档的整体布局，包括 head 元素中的一个标题和一个链接，以及 body 元素和该元素内的正文内容：

```
!!!
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

index.jade 文件为 layout.jade 文件中定义的 body 提供实际内容。

我会在第 8 章介绍如何在 Express 应用程序中使用 Jade 模板以及 CSS。

现在，我们来回顾下这个应用中都发生了什么：

1. Express 应用程序使用 app.get 将请求监听函数（routes.index）与一个 HTTP GET 请求相关联起来；

2. routes.index 函数调用 res.render 方法来呈现对应于该 GET 请求的响应内容；

3. res.render 函数调用应用程序对象的渲染功能；

4. 应用程序的渲染功能用于呈现指定的视图，视图中可以包含有任何可能的选项，本例中我们使用了 titile；

5. 渲染出的内容被写入到 response 对象，并返回到用户浏览器。

整个处理过程的最后一步是将生成的内容作为响应消息写回到浏览器。其实，render 方法有第三个参数可以接收一个回调函数。当有任何错误发生或者响应内容准备好后，该回调函数会被调用。

由于想仔细看看生成的内容，我修改了 route.index 文件，将内容文本输出到控制台。由于覆盖了默认功能，所以需要再使用 res.write 函数将生成的内容发送回浏览器，然后再调用 res.end 函数表明传输完成。

```
exports.index = function(req, res){
  res.render('index', { title: 'Express' }, function(err, stuff) {
     if (!err) {
        console.log(stuff);
        res.write(stuff);
        res.end();
     }
  });
};
```

正如我们希望的那样，应用程序现在能将内容同时输出到控制台以及浏览器。这恰恰表明，尽管我们使用的是一个陌生的框架，但它们都是基于 Node 以及 Node 提供的模块。当然，由于是框架，它必然也提供了一些比 res.write 和 res.end 更易用的方法。还有就是，在下一节我们会讨论与此有更紧密联系的路由路径。

7.5.1 路由路径

在示例 7-1 中，我们给出的路由及路由路径是非常简单的基于根地址的。Express 内部会将所有路由编译成正则表达式对象，因此你可以使用含有特殊字符的字符串，或者直接在路径字符串中使用正则表达式。

为了进行说明，我在示例 7-2 中创建了一个纯粹使用路由路径的应用程序并监听三种不同的路由。如果有一个请求匹配到了任何一个路由项，那么我们会使用 Express 中 response 对象的 send 方法将请求中包含的参数返回给发送方。

示例 7-2　测试不同路由路径模式

```
var express = require('express')
  , http = require('http');

var app = express();

app.configure(function(){
});

app.get(/^\/node?(?:\/(\d+)(?:\.\.(\d+))?)?/, function(req, res){
   console.log(req.params);
   res.send(req.params);
});

app.get('/content/*',function(req,res) {
   res.send(req.params);
});

app.get("/products/:id/:operation?", function(req,res) {
   console.log(req);
   res.send(req.params.operation + ' ' + req.params.id);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

我们无需处理任何视图请求也无需处理任何静态内容，因此并不需要在 app.configure 方法中提供中间件。然而，如果想要正确地处理不匹配任何路由的请求，我们确实需要调用 app.configure 方法。另外，这个程序默认使用的是 development 环境。

在示例程序中，第一个 app.get 方法调用时使用了正则表达式来匹配路径。这个正则表达式改编自 Express 用户手册中的示例，可以监听任何对 node 的请求。如果请求中提供了独立或者指定了范围的标识符，则能被 request 对象捕获到参数列表中，参数列表随后会被作为响应内容返回。请求示例：

```
node
nodes
 /node/566
/node/1..10
/node/50..100/something
```

返回下面的参数列表：

```
[null, null]
[null, null]
["566", null]
["1", "10"]
["50", "100"]
```

正则表达式会寻找一个单一标识符或一对范围标识符，在两个数之间使用（..）可以指定范围值，其后的任何内容都会被忽略。如果没有标识符或范围设置，参数值就为null。

处理请求的代码并没有使用底层HTTP response对象的write和end方法来将参数返回给请求者，而是使用Express的send方法。send方法会为响应内容生成恰当的消息头（设置数据类型），然后使用底层的HTTP end方法发送消息内容。

接下来的第二个app.get调用使用了字符串来定义路由匹配项。在本例中，该匹配项将匹配以/content/开始的任何内容。请求示例如下：

```
/content/156
/content/this_is_a_story
/content/apples/oranges
```

返回下面的参数列表：

```
["156"]
["this_is_a_story"]
["apples/oranges"]
```

星号*表示可以接受一切content之后的任何内容。最后一个app.get方法期望匹配对product信息的请求。如果请求中有id信息的话，我们就能通过params.id来直接访问它。

如果请求中还提供了operation的话，通过params.operation可以直接访问它。在一个请求中，至少应该包含这两个参数中的一个，当然也可以同时包含两个参数。

请求示例：

```
/products/laptopJK3444445/edit
/products/fordfocus/add
/products/add
/products/tablet89/delete
/products/
```

相应的返回如下结果：

```
edit laptopJK3444445
add fordfocus
undefined add
delete tablet89
```

```
Cannot GET /products/
```

应用程序会将 request 对象输出到控制台。不过为了能更详细地查看 request 对象，我将输出保存到了 output.txt 文件：

```
node app.js > output.txt
```

当然，request 对象就是一个 socket，在之前对 Node HTTP request 对象做介绍时，我们已经对它有了很多了解。目前，我们主要感兴趣的是 route 对象。在程序接收到一次请求后，我们得到了如下有关 route 对象的输出信息：

```
route:
 { path: '/products/:id/:operation?',
   method: 'get',
   callbacks: [ [Function] ],
   keys: [ [Object], [Object] ],
   regexp: /^\/products\/(?:([^\/]+?))(?:\/([^\/]+?))?\/?$/i,
   params: [ id: 'laptopJK3444445', operation: 'edit' ] },
```

请注意到其中生成的正则表达式对象，它将我们在请求字符串中使用冒号分割的字符串转换成了有意义的 JavaScript 引擎可以理解的正则表达式（谢天谢地了，因为我的正则表达式的确很糟糕）。

现在，我们对路由如何工作应该已经有了更好的理解，再来看看 HTTP 动词的使用吧。

提示

如果请求不匹配示例代码中三个路由项的任何一个的话，将会产生一个 404 的响应：Cannot GET /whatever。

7.5.2 路由和 HTTP 动词

在前面的例子中，我们使用 app.get 处理传入的请求。这种方法是基于 HTTPGET 方法的，在查询数据时很有用。如果要输入数据或编辑删除现有数据的话，使用这种方法就会碰到问题。因此，我们应该使用其他 HTTP 动词来创建一个应用程序用于维护以及检索数据。换句话说，我们需要让应用程序更 *RESTful*。

提示

如前所述，REST 表示表述性状态转移（Representational State Transfer）。*RESTful* 是一个术语，用来描述任何符合 HTTP 和 REST 原则的 Web 应用程序，这些原则包括：目录结构式的 URL 地址，无状态，数据被包装为某种互联网媒体类型后传送（如 JSON），以及 HTTP 方法的使用（GET、POST、DELETE 和 PUT）。

比方说，我们需要一个管理 widget 的应用程序。为了创建一个新的 widget，那么就需要提供一个包含表单的 Web 页面来收集 widget 的具体信息。我们可以通过应用程序自动生成这种表单（将在第 8 章中介绍此方法），不过当前我们使用一个静态页面来实现该表单，如示例 7-3 所示。

示例 7-3　将 widget 数据传递给 Express 应用程序的 HTML 表单

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8" />
 <title>Widgets</title>
</head>
<body>
<form method="POST" action="/widgets/add"
enctype="application/x-www-form-urlencoded">

 <p>Widget name: <input type="text" name="widgetname" id="widgetname"
size="25" required/></p>

 <p>Widget Price: <input type="text"
pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
name="widgetprice" id="widgetprice" size="25" required/></p>

 <p>Widget Description: <br /><textarea name="widgetdesc" id="widgetdesc"
cols="20" rows="5">No Description</textarea>
 <p>

 <input type="submit" name="submit" id="submit" value="Submit"/>
 <input type="reset" name="reset" id="reset" value="Reset"/>
 </p>
 </form>
</body>
```

这个页面采用了 HTML5 的新特性 required 和 pattern 来完成对输入数据的校验。当然，这仅适用于支持 HTML5 的浏览器，这里我会假设你正在使用的是支持 HTML5 的现代浏览器。

widget 表单需要名字、价钱（包括绑定在该字段上的正则表达式，它能根据 pattern 属性来验证输入数据有效性），以及描述信息。基于浏览器的验证功能确保了我们能够得到这三个值，并且价钱是符合美元格式要求的。

在该 Express 应用程序中，所有的 widget 信息仅仅保存在内存中，因为此刻我们希望把重点放在纯粹的 Express 技术上。当任何一个新创建的 widget 被传递给应用程序时，应用程序会通过 app.post 方法将它添加到 widget 数组中。可以通过 app.get 方法来访问每个 widget，但需要传递该 widget 的 id 值，该值在创建 widget 时由程序生成。示例 7-4 展示了所有程序代码。

示例 7-4　用于添加及显示 widget 的 Express 应用程序

```
var express = require('express')
  , http = require('http')
  , app = express();

app.configure(function(){
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(app.router);
});

app.configure('development', function(){
  app.use(express.errorHandler());
});

// in memory data store
var widgets = [
  { id : 1,
    name : 'My Special Widget',
    price : 100.00,
    descr : 'A widget beyond price'
  }
]

// add widget
app.post('/widgets/add', function(req, res) {
  var indx = widgets.length + 1;
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     descr : req.body.widgetdesc };
  console.log('added ' + widgets[indx-1]);
  res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
});
// show widget
app.get('/widgets/:id', function(req, res) {
   var indx = parseInt(req.params.id) - 1;
   if (!widgets[indx])
      res.send('There is no widget with id of ' + req.params.id);
   else
      res.send(widgets[indx]);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

我们在数组中保存了 widget，以便在没有做任何添加操作的情况下可以立即进行查询操作。如果请求了一个不存在的 widget 是如何处理的？留意下在 app.get 中的条件判断就应该清楚了。

运行这个应用程序（示例代码中的 example4.js），然后使用“/”或者“/index.html”（或“/example3.html”，例子中的真实文件名）来访问包含有表单的静态页面。在我们输入信息并提交表单后会生成一个页面，显示被提交的 widget 信息以及系统自动生成的 id 值。然后，我们就可以使用该 id 值来查看 widget 了。实际上，我们只是简单地将 widget 对象的内容进行了输出：

```
http://whateverdomain.com:3000/widgets/2
```

这个程序能工作，但还存在一个问题。

首先，不管是有意还是无意，我们可能会在 widget 表单中输入错误的信息。你不能在价格一栏中输入除货币以外的任何数据格式，但是你可以输入一个错误的价格。你也可能很容易就键入错误的名字或描述信息。所以我们需要支持对 widget 的更新操作，以解决这些问题，同样我们也需要支持对无用 widget 的删除操作。

因此，我们的应用程序就需要支持另外两个 RESTful 操作：PUT 和 DELETE。PUT 用于更新 widget，而 delete 用于删除 widget。

为了要更新指定 widget，首先需要显示一个表单来展示 widget 的原有数据，以便对其进行编辑操作。要删除 widget，我们也需要一个确认删除的表单。在一般应用程序中，这些都是使用模板动态生成的，但现在，因为我们只需要关注 HTTP 动词的使用，因此我只创建了一个静态页面，用于编辑及删除已经创建的 id 为 1 的 widget。

下面列出了更新 widget 1 所使用的表单代码。除了将 widget 1 的具体内容进行展示处中，表单中还有一个：隐藏域_method，在代码中以粗体标注：

```
<form method="POST" action="/widgets/1/update"
enctype="application/x-www-form-urlencoded">

 <p>Widget name: <input type="text" name="widgetname"
 id="widgetname" size="25" value="My Special Widget" required/></p>
 <p>Widget Price: <input type="text"
 pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
 name="widgetprice" id="widgetprice" size="25" value="100.00" required/></p>

 <p>Widget Description: <br />
 <textarea name="widgetdesc" id="widgetdesc" cols="20"
 rows="5">A widget beyond price</textarea>
 <p>

 <input type="hidden" value="put" name="_method" />

 <input type="submit" name="submit" id="submit" value="Submit"/>
 <input type="reset" name="reset" id="reset" value="Reset"/>
 </p>
 </form>
```

由于表单属性并不支持 PUT 和 DELETE，因此我们使用了名为_method 的特殊隐藏域来指定具体操作——是 put 还是 delete 操作。

删除 widget 所使用的表单比较简单：它包含了隐藏域_method 以及一个删除确认按钮：

```
<p>Are you sure you want to delete Widget 1?</p>
<form method="POST" action="/widgets/1/delete"
 enctype="application/x-www-form-urlencoded">

 <input type="hidden" value="delete" name="_method" />

 <p>
 <input type="submit" name="submit" id="submit" value="Delete Widget 1"/>
 </p>
</form>
```

为了确保妥善处理 HTTP 动词，我们需要为应用程序添加另一个中间件 express.methodOverride，并将它紧随 express.bodyParser 放置。express.methodOverride 中间件会根据隐藏字段的值来修改当前请求所使用的 HTTP 方法：

```
app.configure(function(){
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
});
```

接下来，我们需要添加代码来处理这两种 HTTP 请求。更新请求会使用新内容替换 widget 的旧内容，而删除请求则会找到对应的 widget 并从列表中删除，但会留下一个空值，因为我们不希望对 widget 数组重新排序。

要完成这个 widget 应用程序，还需要再添加一个索引页面，以便用户可以在不指定任何标识或操作的情况下就可以访问 widget 列表。因此，索引页要完成的功能就是将内存中所有 widget 罗列出来。

示例 7-5 展示了完成后的应用程序代码，并且所有新添加的功能以粗体文字显示。

示例 7-5　修改后的 Widget 应用程序，支持编辑和删除 widget 并能列出所有 widget

```
var express = require('express')
  , http = require('http')
  , app = express();

app.configure(function(){
  app.use(express.favicon());
```

```
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
});

app.configure('development', function(){
  app.use(express.errorHandler());
});
// in memory data store
var widgets = [
  { id : 1,
    name : 'My Special Widget',
    price : 100.00,
    descr : 'A widget beyond price'
  }
]

// index for /widgets/
app.get('/widgets', function(req, res) {
   res.send(widgets);
});

// show a specific widget
app.get('/widgets/:id', function(req, res) {
   var indx = parseInt(req.params.id) - 1;
   if (!widgets[indx])
      res.send('There is no widget with id of ' + req.params.id);
   else
      res.send(widgets[indx]);
});

// add a widget
app.post('/widgets/add', function(req, res) {
  var indx = widgets.length + 1;
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     descr : req.body.widgetdesc };
  console.log(widgets[indx-1]);
  res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
});

// delete a widget
app.del('/widgets/:id/delete', function(req,res) {
   var indx = req.params.id - 1;
   delete widgets[indx];

   console.log('deleted ' + req.params.id);
   res.send('deleted ' + req.params.id);
});

// update/edit a widget
app.put('/widgets/:id/update', function(req,res) {
```

```
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice),
     descr : req.body.widgetdesc };
  console.log(widgets[indx]);
  res.send ('Updated ' + req.params.id);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

在运行该应用程序后，我首先尝试添加 widget，再罗列所有 widget，然后更新 id 为 1 的 widget 的价格信息，再删除它，最后再次罗列所有 widget。控制台的输出如下：

```
Express server listening on port 3000
{ id: 2,
  name: 'This is my Baby',
  price: 4.55,
  descr: 'baby widget' }
POST /widgets/add 200 4ms
GET /widgets 200 2ms
GET /edit.html 304 2ms
{ id: 0,
  name: 'My Special Widget',
  price: 200,
  descr: 'A widget beyond price' }
PUT /widgets/1/update 200 2ms
GET /del.html 304 2ms
deleted 1
DELETE /widgets/1/delete 200 3ms
GET /widgets 200 2ms
```

注意输出信息中以粗体标识的部分，分别对应 HTTP PUT 和 DELETE 请求。当我第二次列出所有 widget 后，返回的值是：

```
[
  null,
  {
    "id": 2,
    "name": "This is my Baby",
    "price": 4.55,
    "descr": "baby widget"
  }
]
```

现在，我们有了一个 RESTful 的 Express 应用程序。但是还存在另一个问题。

如果应用程序只管理一个对象，那么把所有功能塞进一个文件或许没有问题。然而，大多数的应用程序都需要管理更多的对象，而且所有这些应用的功能并不会像我们的示例程序这样简单。所以，我们还需要将这个 RESTful 的 Express 应用程序转换为 RESTful 的 MVC Express 应用程序。

7.6 关于 MVC

对于一个很小的应用程序来说，将所有实现代码集中放置在一个文件中是没有太大问题的，但大多数应用程序都需要更好的组织结构。MVC 是一个流行的软件架构，因此我们希望在 Express 程序中使用这种架构来获取它的优点。这件事情其实并没有看起来的那么困难，因为我们可以效仿 Ruby on Rails。

我们可以从 Ruby on Rails 获得许多有关 MVC 的基本设计原则，以便将其引入并支持 Node 的 MVC 设计。Express 已经采用了路由的概念（Rails 的基本原则），所以我们已经完成一半 MVC 功能。现在，还需要提供第二部功能，即分离的模型：视图和控制器。控制器会定义一套动作以操作视图中的每个对象。

Rails 提供多种方式来将路由信息（包括请求类型及请求路径）映射到相应的数据操作。这个映射是基于 CRUD（create 创建、read 读取、update 更新和 delete 删除，它们是持久存储所需要的四个基本功能）概念实现的。Rails 的网站提供了一个很好的有关映射的说明文档，我们可以根据它来为应用程序创建映射。我根据 Rails 的表格推导出了表 7-1，它展示我们的 widget 应用程序所需要的映射。

表 7-1　widget 对象的 REST / route / CRUD 映射

HTTP verb	Path	Action	Usedfor
CET	/widgets	index	显示所有 widgets
CET	/widgets/new	new	返回创建新 widget 的 HTML 表单
POST	/widgets	create	创建一个新的 widget
GET	/widgets/:id	show	显示特定的 widget
GET	/widgets/:id/edit	edit	返回用于修改指定的 widget 的 HTML
PUT	/Widgets/:id	update	更新一个指定的 widget
DELETE	/Widgets/id	destroy	删除指定的 widget

其中的很多功能我们已经实现了，现在只需要把它整理的干净一点。

提示

只是一个提醒：当实现这些与 MVC 相关的修改时，你可能会碰到一些与现有中间件冲突的问题。例如，当使用 directory 中间件提供目录输出功能时，将会与 create 动作冲突，因为它们需要使用同一个路由。解决方法就是将 configure 方法调用中对 express.directory 中间件的使用放在 app.router 之后。

首先，创建一个 *controllers* 子目录，并在该目录内新建一个 *widgets.js* 文件。然后，将所有有关 apt.get 和 apt.put 方法调用的代码复制到这个新文件。

接下来，我们需要将其转换为适合 MVC 格式的方法调用。这意味着要将现有的每个路由方法调用转换为单独的函数然后导出。例如，对于创建新 widget 的函数：

```
// add a widget
app.post('/widgets/add', function(req, res) {
  var indx = widgets.length + 1;
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice)};
  console.log(widgets[indx-1]);
  res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
});
```

它将被转换为 widgets.create：

```
// add a widget
exports.create = function(req, res) {
  var indx = widgets.length + 1;
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice)},
  console.log(widgets[indx-1]);
  res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
};
```

转换后的每个函数仍然需要 request 和 resource 对象。唯一的区别是，转换后的函数中不包括路由映射逻辑。

示例 7-6 展示了在 controllers 目录下的 widgets.js 文件内容。new 和 edit 两个方法目前没有实现，我会在第 8 章实现它们。我们依然使用内存来存储 widget 数据，因为这样能简化 widget 对象，同时删除描述字段可以使应用程序更容易进行测试。

示例 7-6 widget 控制器

```
var widgets = [
   { id : 1,
     name : "The Great Widget",
     price : 1000.00
   }
]

// index listing of widgets at /widgets/
exports.index = function(req, res) {
   res.send(widgets);
};

// display new widget form
exports.new = function(req, res) {
   res.send('displaying new widget form');
};

// add a widget
exports.create = function(req, res) {
  var indx = widgets.length + 1;
  widgets[widgets.length] =
   { id : indx,
     name : req.body.widgetname,
     price : parseFloat(req.body.widgetprice) };
  console.log(widgets[indx-1]);
  res.send('Widget ' + req.body.widgetname + ' added with id ' + indx);
};

// show a widget
exports.show = function(req, res) {
   var indx = parseInt(req.params.id) - 1;
   if (!widgets[indx])
      res.send('There is no widget with id of ' + req.params.id);
   else
      res.send(widgets[indx]);
};

// delete a widget
exports.destroy = function(req, res) {
   var indx = req.params.id - 1;
   delete widgets[indx];

   console.log('deleted ' + req.params.id);
   res.send('deleted ' + req.params.id);
};

// display edit form
exports.edit = function(req, res) {
   res.send('displaying edit form');
};
// update a widget
exports.update = function(req, res) {
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
```

```
  { id : indx,
    name : req.body.widgetname,
    price : parseFloat(req.body.widgetprice)}
  console.log(widgets[indx]);
  res.send ('Updated ' + req.params.id);
};
```

请注意，edit 和 new 都使用了 GET 方法，他们唯一的目的是呈现表单。相应的，create 和 updata 方法会真正去修改数据，所以前者使用 POST，而后者使用 PUT。

为了将路由映射到这些新函数上，我创建了第二个模块 maproutecontroller，它仅导出了一个函数 mapRoute。该函数有两个参数：Express 的 app 对象和一个用于描述控制器对象的前缀（此处我们使用 widgets）。它使用前缀来访问 widgets 控制器对象，然后为已知的控制器方法（控制器对象中的方法集是固定的）做恰当的路由映射配置。示例 7-7 显示了这个新模块的代码。

示例 7-7　将路由映射到控制器方法

```
exports.mapRoute = function(app, prefix) {

   prefix = '/' + prefix;

   var prefixObj = require('./controllers/' + prefix);

   // index
   app.get(prefix, prefixObj.index);

   // add
   app.get(prefix + '/new', prefixObj.new);

   // show
   app.get(prefix + '/:id', prefixObj.show);

   // create
   app.post(prefix + '/create', prefixObj.create);

   // edit
   app.get(prefix + '/:id/edit', prefixObj.edit);

   // update
   app.put(prefix + '/:id', prefixObj.update);

   // destroy
   app.del(prefix + '/:id', prefixObj.destroy);

};
```

mapRoute 是一个非常简单的函数，比较表 7-1 中给出的路由可以更容易地理解它。

最后，让我们将这些代码整合起来完成主应用程序代码吧。谢天谢地，代码清晰多了，我们无需再一项项指定路由方法调用了。为了应付对象数量增长的可能，我将

每个控制器的前缀名统一放在了一个数组中。当需要添加一个新对象时，我只需要向数组中添加一个前缀即可。

提示

Express 自带的 MVC 应用程序的放在 *examples* 子目录中。它使用一个例程访问 *controllers* 目录，并根据搜索到的文件名称来推断前缀名。通过这种方法，我们不必改变程序文件，就能添加一个新的对象。

示例 7-8 显示了完成后的应用程序。我将原先使用的 routes.index 视图拿了回来，还将 routes/index.js 文件中的标题值从“Express”改为“Widget Factory”。

示例 7-8　使用了 MVC 架构的 widget 应用程序

```
var express = require('express')
  , routes = require('./routes')
  , map = require('./maproutecontroller')
  , http = require('http')
  , app = express();

app.configure(function(){
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.staticCache({maxObjects: 100, maxLength: 512}));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.directory(__dirname + '/public'));
  app.use(function(req, res, next){
    throw new Error(req.url + ' not found');
  });
  app.use(function(err, req, res, next) {
    console.log(err);
    res.send(err.message);
  });
});

app.configure('development', function(){
  app.use(express.errorHandler());
});

app.get('/', routes.index);
var prefixes = ['widgets'];

// map route to controller
prefixes.forEach(function(prefix) {
  map.mapRoute(app, prefix);
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

现在代码看起来即干净又简单，并且可扩展性更好了。但我们的代码仍然缺少 MVC 架构中的视图部分，我会在下一章做介绍。

7.7 使用 cURL 测试 Express 应用程序

我们可以使用 cURL 替代浏览器对应用程序进行测试。这个 Unix 工具对于测试 RESTful 应用程序是非常有用的，而且无需创建任何表单。

使用如下 cURL 命令可以测试 widgets 索引页(启动例子程序，然后监听 3000 端口；你可能需要根据你自己的配置环境对命令做相应调整）

```
curl --request GET http://examples.burningbird.net:3000/widgets
```

在 request 选项后，我们指定了请求类型（本例中为 GET），然后是请求的 URL 地址。运行命令后，你应该能够获得当前所保存的所有 widget 信息。为了测试新建 widget 功能，首先要发送一个创建新对象的请求：

```
curl --request GET http://examples.burningbird.net:3000/widgets/new
```

我们会得到用于收集 widget 信息的表单。接下来，通过将请求类型改为 POST 并在请求中指定 widget 数据，就可以测试添加 widget 了：

```
curl --request POST http://examples.burningbird.net:3000/widgets/create
 --data 'widgetname=Smallwidget&widgetprice=10.00'
```

所以，再次请求索引页以验证新的 widge 被正确添加：

```
curl --request GET http://examples.burningbird.net:3000/widgets
```

结果应该是：

```
[
  {
    "id": 1,
    "name": "The Great Widget",
    "price": 1000
  },
  {
    "id": 2,
    "name": "Smallwidget",
    "price": 10
  }
```

接下来测试 widget 更新操作，将价格修改为 75.00。对应的 HTTP 请求类型则为 PUT：

```
curl --request PUT http://examples.burningbird.net:3000/widgets/2
--data 'widgetname=Smallwidget&widgetprice=75.00'
```

在验证数据更新成功后，就可以继续测试删除功能了，注意使用 HTTP 的 DELETE 方法：

```
curl --request DELETE http://examples.burningbird.net:3000/widgets/2
```

现在，我们已经有了 MVC 控制器组件，但还需要添加视图组件，这将在第 8 章中详细介绍。不过，在继续后续内容之前，阅读下“其他框架”说明栏可以知道一些有用的技巧。

其他框架

虽然 Express 是一个框架，但它却是一个非常简单和基本的框架。如果你想用它来做更多的事（如创建一个内容管理系统），还是需要相当多的工作量。

因此，有一些以 Express 为基础的第三方应用程序能够为我们提供更多功能。Calipso 就是一个建立在 Express 之上完整的内容管理系统（CMS），它使用 MongoDB 做持久性存储。

而 Express-Resource 是一个小型框架，它为 Express 提供了简化的 MVC 功能，这样你就不自己写了。

Tower.js 是另一个能够支持完整 MVC 的 Web 框架，它提供了一个更高层次的抽象，并且以 Ruby on Rails 为原型。RailwayJS 也是一个 MVC 框架，基于 Express 并仿照 Ruby on Rails。

还有一个名为 Strata 的框架，它采取了与 Tower.js 和 RailwayJS 不同的策略。它遵循由 WSGI（Python）和 Rack（Ruby）建立的模型，而不是 Rails 模型。这是一个低级别的抽象，如果你没有 Ruby 和 Rails 编程工作经验，使用它会更简单些。

第 8 章

Express、模板系统和 CSS

类似 Rexpress 这种架构提供了很多有用的功能，但是并没有提供一种将数据与数据呈现分开的方法。你可以使用 JavaScript 生成 HTML 文件的方式处理查询或者修改结果，但是工作量很快就变得不可估量，特别是如果你想要生成页面的各个部分，包括侧边栏、页首、页脚等。当然，你也可以使用一些函数，但是这个工作量也足够让你崩溃了。

幸运的是，当架构系统开发出来的时候，模板系统也会随之产生，对 Node 和 Express 来说也是如此。在第 7 章中，我们主要使用了 Express 默认安装的模板系统 Jade，包括另一个很流行的组件 EJS（embedded JavaScript，嵌入式 JavaScript）。Jade 和 EJS 采用了完全不同的实现方式，但是都实现了相同的结果。

同时，在传统方式中你可以为自己的网站或者应用程序手动创建 CSS 文件，现在你也可以使用 CSS 引擎，简化网页的设计和开发。你需要一个简单的结构易于编写 CSS，而不是每次都要自己输入花括号和分号。一个可以与 Express 和其他 Node 应用完美融合的 CSS 引擎叫做 Stylus。

在本章中，我主要关注 Jade，因为它是由 Express 默认安装的。但是我也会大概介绍一下 EJS，这样你就可以看出来两个模板系统的差别以及了解它们是如何工作的。同时我还会介绍一下 Stylus 是如何管理 CSS 以保证页面正确显示的。

8.1 EJS 模板系统（Embedded JavaScript Template System）

对于 EJF 来说，嵌入式 JavaScript 是个好名字，很好地表述了 EJS 是如何工作的：

嵌在 HTML 标志中的 JavaScript，用于融合数据和 HTML 文档结构。EJS 是一个基于 Ruby ERB（embedded Ruby）的简单模板系统。

提示

EJS GitHub 页面：https://github.com/visionmedia/ejs。

8.1.1 基本语法

如果你用过很多 CMS（内容管理系统，Content Management System），你很快就能理解 EJS 的基本原理。以下代码是 EJS 模板的一个示例：

```
<% if (names.length) { %>
  <ul>
    <% names.forEach(function(name){ %>
      <li><%= name %></li>
    <% }) %>
  </ul>
<% } %>
```

在代码中，EJS 直接嵌入在 HTML 中，在本例中用于提供数据给独立的无序列表项。尖括号和百分号组成的符号对（<%，%>）用于表达 EJS：条件表达式用于确保数组存在，之后 JavaScript 对数组进行处理，输出数组中每一项的值。

提示

EJS 是基于 Ruby ERB 的模板系统，这就是为什么你会经常看到用“类 erb”来描述它的格式。

输出一个变量值本身用等号，代表“在此处打印该值”：

```
<%= name %>
```

当打印出来的时候，该值转义了，要打印出来转义的值，使用一个间隔符，如下所示：

```
<%- name %>
```

可能有时候你并不想使用表述 EJS 的开闭标志（<%, %>）,你可以通过 EJS 对象提供的 open，close 方法自定义标识符：

```
ejs.open('<<');
ejs.close('>>');
```

然后就可以使用自己定义的标识符替代默认的：

```
<h1><<=title >></h1>
```

不过除非你坚持如此，还是推荐使用默认的符号。

尽管是 EJS 嵌入在 HTML 中的，但它还是 JavaScript，所以你需要提供开闭的花括号，并且在使用 array 对象的 forEach 方法时保持适当的格式。

作为完整的产品，HTML 会通过 EJS 函数调用进行渲染，返回一个可以生成结果的 JavaScript 函数，或者直接生成最终结果。当我们为 Node 安装好 EJS 后我会详细说明这一点。现在就来完成安装吧。

8.1.2 Node 与 EJS

我们需要安装的模块是与 Node 兼容的 EJS 版本。它与你直接在 EJS 官网下载的不是同一个东西。Node EJS 可以用于客户端 JavaScript，但是我们只关注如何在 Node 程序中使用它。

npm 安装该模块：

```
npm install ejs
```

EJS 安装完成后，就可以直接在一般的 Node 程序中使用了，并不一定需要类似 Express 的架构。作为例子，以下代码说明了从模板文件如何生成 HTML：

```
<html>
<head>
<title><%= title %></title>
</head>
<body>
<% if (names.length) { %>
  <ul>
    <% names.forEach(function(name){ %>
      <li><%= name %></li>
    <% }) %>
  </ul>
<% } %>
</body>
```

直接调用 EJS 对象的 renderFile 方法，这样做会打开模板文件并用提供的数据作为参数生成 HTML。

示例 8-1 使用 Node 提供的标准 HTTP 服务器，在端口 8124 上监听请求。当收到请求时，程序会调用 EJS 的 renderFile 方法，将模板文件的路径、names 数组和文档的 title 作为参数传递进去。最后一个参数是回调函数，显示错误（可以阅读的错误

信息）或者返回生成的HTML。在该例子中，如果没有错误发生结果则会通过 response 发送回去。如果出错，错误信息会被加载到结果中，错误对象内容输出到控制台。

示例 8-1　根据数据和 EJS 模板生成 HTML

```
var http = require('http')
   , ejs = require('ejs')
;
// 创建 HTTP 服务器
http.createServer(function (req, res) {

   res.writeHead(200, {'content-type':'text/html'});

   // 加载数据
   var names = ['Joe', 'Mary', 'Sue', 'Mark'];
   var title = 'Testing EJS';

   // 生成或者处理错误信息
   ejs.renderFile(__dirname + '/views/test.ejs',
                  {title:'testing', names:names},
                    function (err, result) {
       if (!err) {
          res.end(result);
       } else {
          res.end('An error occurred accessing page');
          console.log(err);
       }
   });
}).listen(8124);
console.log('Server running on 8124/');
```

另外一种 rendering 方法是 render，接收 EJS 模板作为字符串类型参数，返回生成好的 HTML：

```
var str = fs.readFileSync(__dirname + '/view/test.ejs', 'utf8');

var html = ejs.render(str, {name : names, title: title});

res.end(html);
```

第三种 rendering 的方式是 compile，接收 EJS 模板字符串并返回可以生成 HTML 的 JavaScript 方法供调用。我不会对这一方法做说明，但是你可以使用这种方法在 Node 客户端程序中使用 EJS。

提示

Compile 的用法会在 9.2 节介绍。

8.1.3 EJS 与 Node Filters

除了支持渲染 EJS 模板外，Node EJS 还提供了一系列预先定义的 filter，可以进一步地简化 HTML 的生成过程。比如，一个名为 first 的 filter，对给定的数组可以取出第一个值。另一个 fliter——downcase，接收 first filter 的结果并改为小写：

```
var names = ['Joe Brown', 'Mary Smith', 'Tom Thumb', 'Cinder Ella'];

var str = '<p><%=: users | first | downcase %></p>';

var html = ejs.render(str, {users:names });
```

结果为：

```
<p>joe brown</p>
```

filters 可以链式调用，上一个 filter 的结果会传递给下一个作为参数。filter 的使用由等号之后紧跟的冒号触发，之后跟着数据对象。以下代码的示例接收一系列 people 对象为参数，构造一个新的对象仅有 peaple 的名字构成，按名字排序，并输出由名字拼接好的字符串：

```
var people = [
   {name:'Joe Brown', age:32},
   {name:'Mary Smith', age:54},
   {name:'Tom Thumb', age:21},
   {name:'Cinder Ella', age:16}];
var str = "<p><%=: people | map:'name' | sort | join %></p>";
var html = ejs.render(str, {people:people });
```

filter 组合使用的结果如下：

```
Cinder Ella, Joe Brown, Mary Smith, Tom Thumb
```

filters 并没有记录在 Node EJS 的文档中，在交换顺序使用这些 filter 时必须特别小心，因为一些 filter 需要 string 作为参数，而不是数组对象。表 8-1 包含了一系列的 filter，并简要说明了该 filter 需要的数据类型以及作用。

表 8-1 Node EJS filters

Filter	数据类型	作用
first	接收并返回数组	返回数组第一个元素
last	接收并返回数组	返回数组最后一个元素
capitalize	接收并返回字符串	字符串第一个字母大写
downcasa	接收并返回字符串	字符串全部转换为小写

续表

Filter	数据类型	作　用
upcase	接收并返回字符串	字符串全部转换为大写
sort	接收并返回数组	对数组应用 Array.sort 方法
sort_bt:'prop'	接收数组和属性名称；返回数组	创建自定义的排序方法，根据属性对数组排序
Size	接收数组；返回数值	返回 Array.length
Plus:n	接收两个数字或者字符串；返回数值	返回 a+b
Minus:n	接收两个数值或者字符串；返回数值	返回 b−a
Times:n	接收两个数值或者字符串；返回数值	返回 a*b
Devided_by:n	接收两个数值或者字符串；返回数值	返回 a/b
Join: 'val'	接收数组；返回字符串	用给定值或默认值应用 Array.join 方法
truncate:n	接收字符串和长度数值；返回字符串	引用 String.substr
truncate_words:n	接收字符串和单词数；返回字符串	应用 String.spit,String.splice
Replace:pattern, substitution	接收字符串、模版和替换内容；返回字符串	应用 String.replace
Prepend:value	接收字符串和 value 字符串；返回字符串	把 value 加在字符串之前
Append:value	接收字符串和 value 字符串；返回字符串	把 value 拼在字符串之后
Map:'prop'	接收字符串和属性；返回数组	用 Array.map 方法根据给定对象属性创建新的字符串
Reverse	接收数组或者字符串	如果是数组，应用 Array.reverse；如果是字符串，分解单词，翻转，再拼接
Get	接收对象和属性	返回给定对象的该属性值
Json	接收对象	转换为 JSON 字符串

8.2　在 Express 中使用 EJS

模板系统提供了 MVC（Model-View-Controller）应用程序（在第 7 章中介绍过这种架构）中 view 部分我们需要的部分。

提示

MVC 的 model 部分会在第 10 章中介绍。

在第 7 章示例 7-1 中，我简单介绍了关于模板系统的用法。该例子使用 Jade，但是我们很容易可以转换为 EJS。有多简单呢？示例 8-2 基本是示例 7-1 的一个复制，除了使用 EJS 替代 Jade。准确的说只有一行变动了，用粗体标了出来。

示例 8-2　在应用程序中使用 EJS 作为模板系统

```
var express = require('express')
    ,routes = require('./routes')
    ,http = require('http');
var app = express();

app.configure(function () {
  app.set('views', _dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
});

app.configure('development', function () {
  app.use(express.errorHandler());
});

app.get('/', routes.index);

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

index.js 路由并没有任何改变，因为它并没有使用任何与模板系统有关的内容，只使用了 Express resource 对象的 render 方法，这与模板系统无关（只要系统兼容 Express）：

```
exports.index = function (req, res) {
  res.render('index', { title:'Express' }, function (err, stuff) {
     if (!err) {
         console.log(stuff);
         res.write(stuff);
         res.end();
     }
  });
};
```

在 views 目录下，index.ejs 文件（注意扩展名）使用 Node EJS 标注替代了第 7 章中我们看到的 Jade：

```
<html>
```

```
<head>
<title><%= title %></title>
</head>
<body>
<h1><%= title %></title>
<p>Welcome to<%= title %></p>
</body>
```

这段代码显示了在程序中将 model、controller、view 分离是一件多么美好的事情：你可以切换所使用的技术，比如使用不同的模板系统，而不会影响到应用程序的逻辑或者数据访问。

简单回顾一下这个程序的过程：

1. Express 核心程序使用 app.get 将一个请求监听方法（routes.index）与 HTTP Get 请求联系起来；

2. routes.index 方法调用 res.render 方法来为请求生成响应；

3. res.render 方法唤醒程序对象的 render 方法；

4. 程序的 render 方法根据选项内容——本例中是 title——找到特定的 view；

5. 需要被渲染的内容被写入 response 对象，然后返回到用户浏览器上。

在第 7 章中，我们主要关注了程序的路由方面，现在我们需要关注视图。我们使用第 7 章最后示例 7-6 到示例 7-8 所创建的程序，对其添加视图。但首先需要对环境做一些小改变来确保程序可以按照我们需要的方向继续发展。

8.2.1 多对象环境的改造

尽管程序被称为 Widget Factory，但是 widget 并不是该程序唯一的产品。我们需要对环境进行改造来添加需要的对象。

现在，结构如下：

```
/application 目录
   /routes - controller 的根目录
   /controllers -对象的 controller
   /public -静态文件
   /views -模版文件
```

routes 和 controllers 目录保留不变，views 和 public 目录需要修改，以允许不同对象存在。为了不把所有 widget 都放在 views 根目录下，我们在 views 下添加一个子目

录来放置所有 widgets：

```
/application 目录
   /views
     /widgets
```

同样，为了不把 widget 静态文件放在 public 目录下，创建一个子目录 widget 在 public 目录下：

```
/application 目录
   /public
     /widgets
```

现在我们可以通过添加新的目录来添加新的对象，而且对每一个对象都可以使用类似 new.html 和 edit.ejs 的文件名而不会引发新文件覆盖旧文件的问题。

注意到这种结构假定了我们的应用程序中存在静态文件。下一步需要做的是找到如何将静态文件与新的动态环境集成的问题。

8.2.2 静态文件路由

应用程序第一个需要修改的部分就是添加一个新的 widget。它包括两个部分：显示一个可以获取新的 widget 信息的表单和将新的 widget 存储在已有的 widget 数据存储中。

我们可以为需要的表单创建一个 EJS 模板，但是它并不包含任何动态的部分，或者说，至少在页面设计这个点上没有。所以，通过模板系统为某部分提供服务，但是该部分并不需要模板系统的功能，这是没有意义的事情。

我们可以仅仅改变表单访问方式来完成这一功能，通过访问/widgets/new.html 来访问表单，替代之前的/widgets/new。但是这会导致程序中路由表达的不一致。如果我们后面再为该表单页面添加动态组件，需要修改指向新表单的方式。

一个好的实现方式是按照处理动态文件的方式来处理请求和静态页面，但是不经过模版系统。

Express resource 对象有个 redirect 方法可以用于将请求重定向到 new.html，但是 new.html 会在处理结束时显示在浏览器的地址栏中，同时也会返回 302 状态码，我们并不希望看到这一点。所以，我们使用 resource 对象的 sendfile 方法替代。Sendfile 方法接收一个文件路径作为参数，可能的选项值，以及一个只有 error 作为参数的回调函数，该回调函数可选。Widget controller 只使用第一个参数。

文件路径为：

```
__dirname + "/../public/widgets/widget.html"
```

我们使用相对路径符号“..”，因为 public 目录与 controller 目录的上一级平行。但是在 sendfile 方法中并不能这样使用路径，这会产生 403 错误的状态码。作为替代，我们使用 path 模块的 normalize 方法将相对路径转换为对应的绝对路径。

```
//显示新的 widget 表单
exports.new = function(req, res) {
    var filepath = require('path').normalize(__dirname +
                                     "/../public/widgets/new.html");
    res.sendfile(filepath);
};
```

这个表单的 HTML 没有什么惊喜，只是一个简单的表单，如示例 8-3 所示。但是，我们添加了 description 区域来使数据更有趣一点。

示例 8-3　HTML 新 widget 表单

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8"/>
 <title>Widgets</title>
 </head>
<body>
<h1>Add Widget:</h1>

<form method="POST" action="/widgets/create"
enctype="application/x-www-form-urlencoded">

 <p>Widget name: <input type="text" name="widgetname"
 id="widgetname" size="25" required/></p>
 <p>Widget Price: <input type="text"
 pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
 name="widgetprice" id="widgetprice" size="25" required/> </p>

 <p>Widget Description: <br/>
 <textarea name="widgetdesc" id="widgetdesc" cols="20"
 rows="5"></textarea>
 <p>

 <input type="submit" name="submit" id="submit" value="Submit"/>
 <input type="reset" name="reset" id="reset" value="Reset"/>
 </p>
```

```
</form>
</body>
```

表单的方法肯定是 POST。

现在程序可以显示一个创建新 widget 的表单了，我们需要修改 widget controller 来处理表单的 post 过程。

提示

还有一个 Express 的扩展模块——express-rewrite，提供 URL 重定向功能。用以下命令可以安装：

```
npm install express-rewirte
```

8.2.3 处理一个新对象的 Post 请求

我们需要修改程序的主要文件来兼容使用 EJS 模板系统，这比添加新的可以支持的模板优先级更高。我不会完全重复第 7 章中示例 7-8 的 app.js 文件，因为修改仅仅在 configure 方法调用包含 EJS 模板引擎和 views 目录中：

```
app.configure(function () {
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.staticCache({maxObjects:100, maxLength:512}));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.directory(__dirname + '/public'));
  app.use(function (req, res, next) {
    throw new Error(req.url + ' not found');
  });
  app.use(function (err, req, res, next) {
    console.log(err);
    res.send(err.message);
  });
});
```

现在我们准备好转换 widget controller 使用模板了，从添加一个新的 widget 开始。

事实上，Widget controller 中对一个新的 widget 的处理过程并没有变化。我们依然从 request 内容中获取数据，添加到 widget 存储中。不同之处在于，我们已经可以访问模板系统，需要修改的是我们如何回应成功添加的一个新的 widget。

我创建了一个新的 EJS 模板——added.js，如示例 8-4 所示。该文件所做的就是提供一系列 widget 的属性和一个由 widget 对象的 title 组成的消息。

示例 8-4 “Widget added”确认信息模板

```
<head>
<title><%= title %></title>
</head>
<body>
<h1><%= title %> | <%= widget.name %></h1>
<ul>
<li>ID: <%= widget.id %></li>
<li>Name: <%= widget.name %></li>
<li>Price: <%= widget.price.toFixed(2) %></li>
<li>Desc: <%= widget.desc %></li>
</ul>
</body>
```

update 的处理过程与第 7 章中显示的略有不同，我们现在会返回给用户一个视图而不是简单的消息（修改的部分用粗体标出）。

```
// 添加一个 widget
exports.create = function (req, res) {

  // 获取 widget id
  var indx = widgets.length + 1;

  // 添加 widget
  widgets[widgets.length] =
   {id : indx,
    name : req.body.widgetname,
    price : parseFloat(req.body.widgetprice),
    desc : req.body.widgetdesc};

  //输出到控制台并对用户确认添加信息
  console.log(widgets[indx-1]);
  res.render('widgets/added', {title:'Widget Added', widget:widgets[indx - 1]});
};
```

发送给视图的两个选项分别是页面 title 和 widget 对象。图 8-1 显示了该信息，目前还是文本描述形式。

提示

处理新 widget 的过程并没有对数据做任何验证或者检查 SQL 注入侵害。数据验证，安全以及权限问题在第 15 章中涉及。

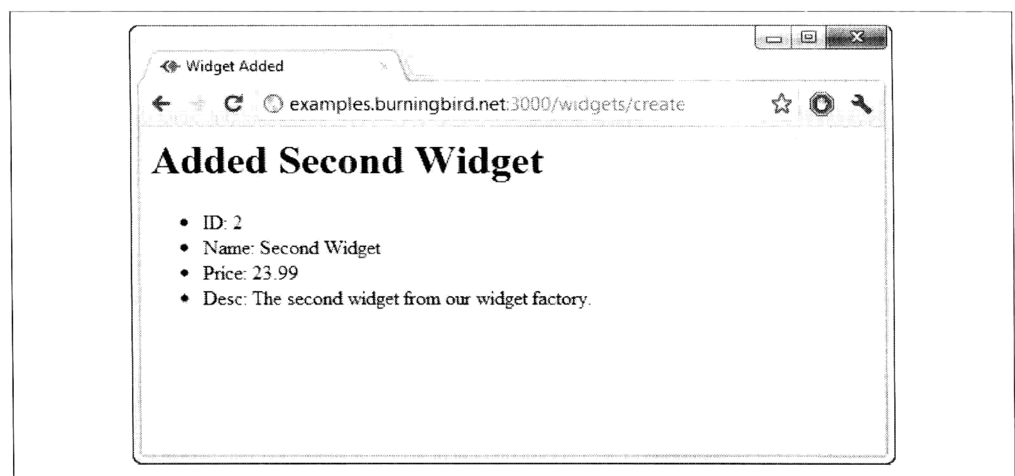


图 8-1　添加 widget 后的确认信息

接下来转换为模板的两个过程是 update（更新）和 deletion（删除），需要一种方式确定操作在哪个 widget 上执行。同时，我们也需要使 index 显示所有的 widget。在这三个过程中，我们将会使用一个视图创建 widget 索引页面和选用列表，接下来会介绍。

8.2.4　Widget 索引和生成 picklist

picklist（选用列表）就是指一系列可供选择的选项列表。对于 widget 程序来说，picklist 可能是一个集成在更新或者删除页面的选择区域或者下拉列表，使用 Ajax 或者客户端脚本语言实现。但是，我们将要完成的是将该功能集成在 widget 程序的 index 页面中。

现在的 widget index 页面仅显示 widget 数据存储中的数据。信息量很大，但是不易阅读没有太大实际用处。为了改进这一部分，我们需要添加一个新的视图来显示数据表中所有的 widget，每个 widget 一行，由 widget 属性组成。还需要添加两个新的列，一列链接到对该 widget 的修改，另一列用于删除。这些补全了程序缺少的部分：不需要记住 widget id 可以编辑或删除某个 widget 的方式。

示例 8-5 为需要的新视图模板的内容，名为 index.ejs。该文件在 widgets 子目录下，所以我们不需要担心它是否与更高层的 index.ejs 重名。

示例 8-5　Widgets index 页面，每个 widgets 带有编辑和删除链接

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8"/>
```

```
  <title><%= title %></title>
  </head>
 <body>
 <% if (widgets.length) { %>
   <table>
   <caption>Widgets</caption>
   <tr><th>ID</th><th>Name</th><th>Price</th><th>Description</th></tr>
     <% widgets.forEach(function(widget){ %>
        <tr>
        <td><%= widget.id %></td>
        <td><%= widget.name %></td>
        <td>$<%= widget.price.toFixed(2) %></td>
        <td><%= widget.desc %></td>
        <td><a href="/widgets/<%= widget.id %>/edit">Edit</a></td>
        <td><a href="/widgets/<%= widget.id %>">Delete</a></td>
        </tr>
     <% }) %>
   </table>
 <% } %>
 </body>
 </html>
```

controller 部分用于触发新视图调用的代码相当简单：只需要调用 render 方法，将所有 widget 组成的数组作为参数：

```
//访问/widgets 显示所有 widgets 的索引

exports.index = function(req, res) {
   res.render('widgets/index', {title: 'Widgets', widgets: widgets});
};
```

示例 8-5 中，如果对象有 length 属性（说明对象为数组），该元素对象会被遍历，属性会输出到表格中，同时还有编辑和删除链接。图 8-2 显示存储了几个 widget 之后的表格信息。

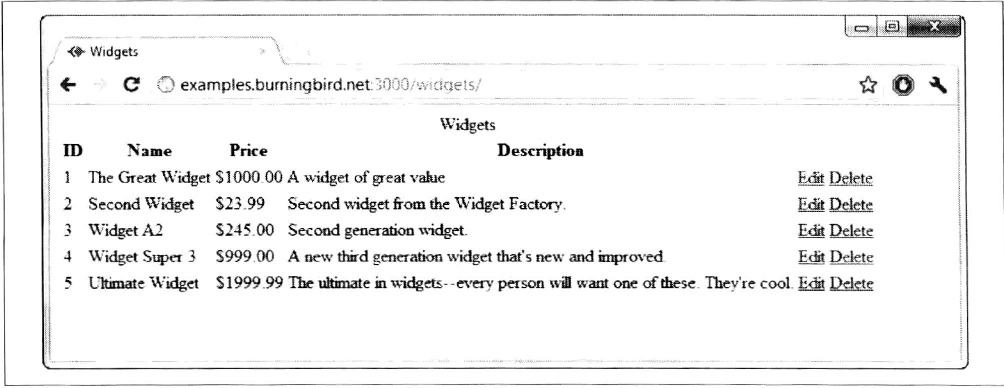


图 8-2 添加几个 widget 之后 Widget 显示的表格信息

删除对象的链接（delete）实际上与显示对象的链接一致：/widgets/:id。我们会添加一个包含一个删除按钮的隐藏的表单在 Show Widget 页面上，这使我们可以不用添加新的路由而完成删除功能。同时，这也提供了一种保护，确保用户知道他们删除的是哪个 widget。

提示

除了在 Show Widget 页面上完成删除请求之外，还可以创建一个新的路由，比如/widgets/:id/delete，并在删除操作发生时生成一个“Are you sure?”的确认信息。

8.2.5 显示单个对象并确认对象的删除操作

显示单个 widget 和为属性提供占位一样简单，可以嵌入在任何你想使用的 HTML 中。在 widget 程序中，我选择 ul（unordered list，无序列表）显示 widget 的所有属性。

因为我们在页面中包含了删除对象的功能：在页面底部添加一个表单，包含一个显示“Delete Widget”的按钮。表单中隐藏的用于生成 HTML 删除动作的_method 区域可以匹配程序的 destroy 方法。整个模板如示例 8-6 所示。

示例 8-6　显示 widget 及其所有属性的页面和用于删除 widget 的表单

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8"/>
 <title><%= title %></title>
</head>
<body>
<h1><%= widget.name %></h1>
<ul>
<li>ID: <%= widget.id %></li>
<li>Name: <%= widget.name %></li>
<li>Price: $<%= widget.price.toFixed(2) %></li>
<li>Description: <%= widget.desc %></li>
</ul>
<form method="POST" action="/widgets/<%= widget.id %>"
enctype="application/x-www-form-urlencoded">

 <input type="hidden" value="delete" name="_method"/>

 <input type="submit" name="submit" id="submit" value="Delete Widget"/>
</form>
</body>
```

对于 controller 代码中的 show 或者 destroy 方法基本不需要修改。destroy 方法维持原

状，该方法所有功能就是从存储中删除对象，并发送消息说明该操作结果：

```
exports.destroy = function (req, res) {
   var indx = req.params.id - 1;
   delete widgets[indx];

   console.log('deleted ' + req.params.id);
   res.send('deleted ' + req.params.id);
};
```

show 方法没什么修改，只是简单地将 send 方法替换成 render，用 render 新视图来替换发送信息：

```
// 显示 widget
exports.show = function (req, res) {
   var indx = parseInt(req.params.id) - 1;
   if (!widgets[indx])
       res.send('There is no widget with id of ' + req.params.id);
  else
       res.render('widgets/show', {title:'Show Widget',widget:widgets[indx]});
};
```

图 8-3 显示了 Show Widget 页面的样式，页面底部包含 Delete Widget 按钮。

图 8-3 带有删除按钮的 Show Widget 页面

到现在，你应该清楚在程序中使用视图是个多么简单的事情了。这个系统最好的地方在于你可以直接对视图进行修改而无须重启程序：视图的改变会在该视图下一次被访问时生效。

最后一个视图用于更新 widget，实现之后我们就完成了在 widget 程序中使用 EJS

模板系统的转换。

8.2.6 提供更新信息的表达以及处理 PUT 请求

用于编辑 widget 信息的表达和添加新 widget 的表达基本一样，除了添加一个新的区域：_method。并且，表单中存在被编辑的 widget 的数据信息，所以我们需要将模板标签和数据对应起来。

示例 8-7 包含了 edit.ejs 模板文件的内容。注意一下 input 元素中模板标签与值域的使用，还有_method 区域。

示例 8-7　编辑视图模板文件，以及数据显示

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8"/>
 <title><%= title %></title>
</head>
<body>
<h1>Edit <%= widget.name %></h1>
<form method="POST" action="/widgets/<%= widget.id %>"
enctype="application/x-www-form-urlencoded">

 <p>Widget name: <input type="text" name="widgetname"
 id="widgetname" size="25" value="<%=widget.name %>" required/></p>

 <p>Widget Price: <input type="text"
 pattern="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)(.[0-9][0-9])?$"
 name="widgetprice" id="widgetprice" size="25" value="<%= widget.price %>"
required/></p>
 <p>Widget Description: <br/>
 <textarea name="widgetdesc" id="widgetdesc" cols="20"
 rows="5"><%= widget.desc %></textarea>
 <p>

 <input type="hidden" value="put" name="_method"/>
 <input type="submit" name="submit" id="submit" value="Submit"/>

 <input type="reset" name="reset" id="reset" value="Reset"/>
</p>
 </form>
 </body>
```

图 8-4 显示了编辑页面。你需要做的事情就是修改数值，然后点击 Submit 按钮提交修改。

controller 代码的修改与之前的一样简单。用 res.render 访问 Edit 视图，widget 对象

作为数据：

图 8-4　Edit Widget 页面

```
// 显示 edit 表单
exports.edit = function (req, res) {
   var indx = parseInt(req.params.id) - 1;
   res.render('widgets/edit', {title:'Edit Widget', widget:widgets[indx]});
};
```

处理 update 过程的代码与第 7 章中的非常类似，除了在这里我们使用视图替代发送信息。但是我们并没有创建新的视图，而是使用了之前的 widgets/added.ejs 文件。因为两个操作都是显示对象所有属性，并且接收 title 作为数据，所以我们可以简单地修改 title 来重用这个视图：

```
// 更新 widget
exports.update = function (req, res) {
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
   { id:indx + 1,
     name:req.body.widgetname,
     price:parseFloat(req.body.widgetprice),
     desc:req.body.widgetdesc}
  console.log(widgets[indx]);
  res.render('widgets/added', {title:'Widget Edited', widget:widgets[indx]})
};
```

并且，视图的使用并不影响显示出来的 URL，所以我们的重用并不会造成问题。重用可以在程序开发过程中节省很多时间和精力。

在我们切换到使用模板的过程中你已经看到 controller 代码的不同片段。示例 8-8 是文件修改后的全部内容，你可以对照示例 7-6 和示例 7-7，可以看到将视图融入代码是个多么简单的事情，以及视图节省了我们多少不必要的工作。

示例 8-8　使用视图的 widget controller 实现

```
var widgets = [
   { id:1,
     name:"The Great Widget",
     price:1000.00,
     desc:"A widget of great value"
   }
]
// /widgets/ 显示 widgets 索引
exports.index = function (req, res) {
   res.render('widgets/index', {title:'Widgets', widgets:widgets});
};
// 显示新的 widget 表单
exports.new = function (req, res) {
    var filePath = require('path').normalize(__dirname + "/../public/
    widgets/new.html");
    res.sendfile(filePath);
};

// 添加 widget
exports.create = function (req, res) {
   // 生成 widget id
   var indx = widgets.length + 1;
   // 添加 widget
   widgets[widgets.length] =
    { id:indx,
      name:req.body.widgetname,
      price:parseFloat(req.body.widgetprice),
      desc:req.body.widgetdesc };
   //输出到控制台并对用户确认添加信息
   console.log(widgets[indx - 1]);
   res.render('widgets/added', {title:'Widget Added', widget:widgets[indx - 1]});
};
// 显示 widget
exports.show = function (req, res) {
   var indx = parseInt(req.params.id) - 1;
   if (!widgets[indx])
      res.send('There is no widget with id of ' + req.params.id);
   else
      res.render('widgets/show', {title:'Show Widget', widget:widgets[indx]});
};
// 删除 widget
exports.destroy = function (req, res) {
   var indx = req.params.id - 1;
   delete widgets[indx];
```

```
    console.log('deleted ' + req.params.id);
    res.send('deleted ' + req.params.id);
};

// 显示编辑表单
exports.edit = function (req, res) {
    var indx = parseInt(req.params.id) - 1;
    res.render('widgets/edit', {title:'Edit Widget', widget:widgets[indx]});
};

// 更新 widget 信息
exports.update = function (req, res) {
  var indx = parseInt(req.params.id) - 1;
  widgets[indx] =
   { id:indx + 1,
     name:req.body.widgetname,
     price:parseFloat(req.body.widgetprice),
     desc:req.body.widgetdesc}
  console.log(widgets[indx]);
  res.render('widgets/added', {title:'Widget Edited', widget:widgets[indx]})
};
```

这是本章中你最后一次看到 controller 的代码，因为我们即将对程序做出重大改变：我们要替换掉模板系统。

8.3 Jade 模板系统

Jade 是由 Express 默认安装的模板系统。与 EJS 不同的是，Jade 不会在 HTML 中直接嵌入模板标签，而是使用简化的 HTML。

提示

Jade 官网：http://jade-lang.com/。

8.3.1 Jade 语法简介

在 Jade 模板系统中，HTML 元素用不带尖括号的名字表示，嵌套关系用缩进代表。所以，对于以下代码：

```
<html>
<head>
<title>This is the title</title>
</head>
<body>
```

```
<p>Say hi World
</body>
</html>
```

在 Jade 中等价为以下写法：

```
html
   head
      title This is it
   body
      p Say Hi to the world
```

title 和 p 标签的内容都跟在标签名之后。没有结束标签符号，也是由缩进表示。以下例子使用了 class 和 id 属性以及多层嵌套：

```
   html
head
   title This is it
body
   div.content
      div#title
         p nested data
```

该模板生成的 HTML 代码为：

```
<html>
<head>
<title>This is it</title>
</head>
<body>
<div class="content">
<div id="title">
<p>nested data</p>
</div>
</div>
</body>
</html>
```

如果某个标签的内容过长，比如一个段落，可以使用竖线（|）来拼接内容：

```
p
  | some text
  | more text
  | and even more
```

对应的 HTML 为：

```
<p>some text more text and even more</p>
```

另一种做法是使用句号（.），代表该标签区域内只包含文字，可以省略竖线：

```
p.
  some text
  more text
  and even more
```

还可以引用 HTML 代码作为内容，在生成的代码中也会被当做 HTML 代码处理：

```
body.
  <h1>A header</h1>
  <p>A paragraph</p>
```

form 元素通常有很多属性，在 Jade 中用括号表示，并可以设置属性的值（如果有）。属性和属性之间只需要空格分离，不过我每行只列出一个属性以增加可读性：

以下是 Jade 模板：

```
html
   head
     title This is it
   body
      form(method="POST"
           action="/widgets"
           enctype="application/x-www-form-urlencoded")
         input(type="text"
               name="widgetname"
               id="widgetname"
               size="25")
         input(type="text"
               name="widgetprice"
               id="widgetprice"
               size="25")
         input(type="submit"
               name="submit"
               id="submit"
               value="Submit")
```

生成 HTML 代码：

```
<html>
<head>
<title>This is it</title>
</head>
<body>
<form method="POST" action="/widgets"
enctype="application/x-www-form-urlencoded">
<input type="text" name="widgetname" id="widgetname" size="25"/>
<input type="text" name="widgetprice" id="widgetprice" size="25"/>
<input type="submit" name="submit" id="submit" value="Submit"/>
</form>
</body>
</html>
```

8.3.2 使用 block 和 extends 模块化视图模板

现在我们需要修改 widget 程序使用 Jade 代替 EJS。我们只需要修改 widget 程序中的 app.js 文件，修改模板引擎：

```
app.set('view engine', 'jade');
```

application.None.Zip 不需要其他修改。

所有的模板共用同一个页面布局。布局很简单，没有侧边栏页脚，也没有使用任何 CSS 样式。正是因为布局的简单，在之前的例子中我们并没有担心在每个 view 中会产生重复的布局代码。但是如果我们希望添加更多的页面结构，比如侧边栏、页首、页尾等，那么在每个文件中维持相同的布局信息就变得很麻烦了。

所以第一件我们需要做的事情就是创建一个可以被其他模板使用的布局的模板。

提示

Express3.x 完全改变了处理 view 的方式，以及如何使用 partial 和 layout。本章中使用 Express2.x，需要在 configure 方法中添加以下代码来使用 Jade 模板：

```
app.set('view options', {layout: false});
```

示例 8-9 是完成的 layout.jade 文件，使用 HTML5 文件类型，添加了带有 title 和 meta 元素的 head 元素，加入了 body 元素，以及一个名为 content 的 block。

示例 8-9　Jade 中简单的布局模板

```
doctype 5
html(lang="en")
  head
    title #{title}
    meta(charset="utf-8")
  body
    block content
```

注意下 title 中井号和花括号（#{}）的使用。在 Jade 中我们用这种方式将数据传递给模板。这种标记符的使用并不随 EJS 改变，只是简单的语法。

在每个 content 模板的开始加入以下代码来使用新的布局模板：

```
extends layout
```

extends 告诉模板引擎在哪里可以找到页面需要的布局信息，而 block 则告诉模板引擎在哪里放置生成的 content 内容。

你不必使用 content 作为 block 的名字，还可以使用多个 block，并且如果你想进一步分解布局，还可以引用其他模板文件。我修改了 layout.jade 文件，包含一个 header，而不是直接在 layout 文件中使用 head：

```
doctype 5
html(lang="en")
  include header
  body
    block content
```

接下来在一个名为 header.jade 的文件中定义 header content，内容如下：

```
head
  title #{title}
  meta(charset="utf-8")
```

在 layout.jade 和 header.jade 文件中有两件事需要注意下。

第一，include 相对路径。如果把 views 分解为以下目录结构：

```
/views
   /widgets
       layout.jade
      /standard
         header.jade
```

在 layout 文件中引用 header 模板时使用：

```
include standard/header
```

文件类型并不一定要是 Jade，也可以是 HTML。当引用的文件不为 Jade 文件类型时，需要加上文件扩展名：

```
include standard/header.html
```

第二，在 header.jade 文件中不要使用缩进。父文件中已经带有缩进了，引用的模板文件中不需要重复缩进。事实上，如果引用文件中带有缩进，生成代码时会报错。

提示

现在你也许会考虑将静态的 Add Widget 文件转换为动态的，以使用新的布局模板的特性。

8.3.3 Widget View 转换为 Jade 模板

第一个需要从 EJS 转换为 Jade 的是 added.ejs 模板文件，该文件提供了成功添加一个 widget 之后的反馈信息。新的模板文件命名为 added.jade（扩展名不同但是名字必须相同，才能继续使用现有的 controller 代码），使用新定义的 lauout.jade 文件，如示例 8-10 所示。

示例 8-10 "添加 Widget"页面转换为 Jade

```
extends layout
block content
  h1 #{titile} | #{widget.name}
  ul
    li id:#{widget.id}
    li Name:#{widget.name}
    li Price:$#{widget.price.toFixed()}
    li Desc:#{widget.desc}
```

注意，我们依然可以使用 toFixed 方法对 price 输出格式化。

Block 名为 content，与 layout.jade 文件设置的 block 名相同。简化的 HTML 代码 h1 和 ul 标签对应从 controller 获取的数据，本例中，数据为 widget 对象的信息。

运行 widget 程序并添加一个新的 widget，生成与 EJS 同样的 HTML 代码：一个 header。新添加的 widget 属性列表——controller 代码完全不需要修改。

转换 widget 最主要的显示页面

下一个需要转换的是 index 模板，在表中显示所有 widgets，带有修改和删除选项。在这次转换中我们会尝试些与之前不一样的东西。我们会从生成整张表转为单独生成每个 widget 的表项。

首先，我们创建一个名为 row.jade 的模板。假定数据是一个名为 widget 的对象，可以访问该对象的以下属性：

```
tr
  td #{widget.id}
  td #{widget.name}
  td $#{widget.price.toFixed(2)}
  td #{widget.desc}
  td
    a(href='/widgets/#{widget.id}/edit') Edit
  td
```

```
      a(href='/widgets/#{widget.id}') Delete
```

每一行都必须独立成行，否则没有缩进就失去了嵌套关系。

index.jade 文件使用新创的 row 模板代码如示例 8-11 所示。该模板介绍了两个新的 Jade 构造器：条件判断和迭代器（iteration）。条件语句用于测试 widgets 对象的长度，以确保待处理的对象是一个非空数组。迭代器使用简化的 Array.forEach 方法，遍历数组，并将每个实例赋值给新的对象 widget。

示例 8-11　创建 widgets 表格的 index 模板

```
extends layout

block content
  table
    caption Widgets
      if widgets.length
       tr
         th ID
         th Name
         th Price
         th Description
         th
         th
       each widget in widgets
         include row
```

这种写法比 HTML 手动加入全部的加括号省事很多，特别是类似于 table th 标签这种。Jade 模板的结果与 EJS 模板结果相同：HTML table，每行显示 widget，可以删除或者修改每个 widget。

8.3.4　转换 edit 和 delete 表单

下面需要做的两件事情是对表单进行修改。

首先，我们用 Jade 来实现 edit 模板。这部分唯一有趣的地方是处理多种属性。可以用空格分隔不同属性，但是我觉得每行单独显示一个属性更容易阅读。这种方法你可以很容易检查是否列出了全部的属性以及值是否正确。示例 8-12 中代码较长，是 Edit Widget 表单的全部代码。

示例 8-12　Edit Widget 表单 Jade 模板

```
extends layout

block content
```

```
h1 Edit #{widget.name}
form(method="POST"
     action="/widgets/#{widget.id}"
     enctype="application/x-www-form-urlencoded")
  p Widget Name:
     input(type="text"
           name="widgetname"
           id="widgetname"
           size="25"
           value="#{widget.name}"
           required)
  p Widget Price:
    input(type="text"
          name="widgetprice"
          id="widgetprice"
          size="25"
          value="#{widget.price}"
          pattern="="^\$?([0-9]{1,3},([0-9]{3},)*[0-9]{3}|[0-9]+)
(.[0-9][0-9])?$"
          required)
  p Widget Description:
    br
    textarea(name="widgetdesc"
             id="widgetdesc"
             cols="20"
             rows="5") #{widget.desc}
  p
    input(type="hidden"
          name="_method"
          id="_method"
          value="put")
    input(type="submit"
          name="submit"
          id="submit"
          value="Submit")
    input(type="reset"
          name="reset"
          id="reset"
          value="reset")
```

在转换 Show Widget 页面过程中，我注意到页面顶部的信息与示例 8-10 中 added.jade 模板中一致，都是所有 widgets 属性的无序列表。又可以进行一次简化！

我创建了一个新的模板——widget.jade，只用于显示 widget 属性的列表：

```
ul
  li id: #{widget.id}
  li Name: #{widget.name}
  li Price: #{widget.price.toFixed(2)}
```

```
    li Desc: #{widget.desc}
```

接下来，修改示例 8-10 中的 added.jade 文件，使用新建的模板：

```
extends layout

block content
  h1 #{title} | #{widget.name}
  include widget
```

新的 Show Widget 模板也可以使用 widget.jade 文件，如示例 8-13 所示。

示例 8-13　新 Show Wdget 页面 Jade 模板

```
extends layout

block content
  h1 #{widget.name}
  include widget
  form(method="POST"
       action="/widget/#{widget.id}"
       enctype="application/x-www-form-urlencoded")
       input(type="hidden"
             name="_method"
             id="_method"
             value="delete")
       input(type="submit"
             name="submit"
             id="submit"
             value="Delete Widget")
```

可以看到模块化使每个模板都变得更简单易读，容易维护了。

我们现在可以通过新的模块化模板显示和删除特定的 widget，这样就会产生疑问：Jade 模板与 EJS 模板区别在哪里？

在 widget 程序中，当删除某个 widget 时，widget 被“原地”删除。这意味着 array 元素被设置为 null，所以 widget 在数组中的位置与 widget id 有关。这种方法在使用 EJS 添加、删除和显示 widget 的时候不会引起问题，但是在 Jade 中会产生问题：会产生缺失属性的错误。因为 Jade 中并不像 EJS 模板处理一样过滤掉值为 null 的 array 元素。

不过这一问题很容易解决。如示例 8-11 所示，只需要在 index.jade 文件中添加一个条件判断，确保 widget 对象存在（不为空）即可：

```
extends layout

block content
```

```
table
  caption Widgets
    if widgets.length
      tr
        th ID
        th Name
        th Price
        th Description
        th
        th
      each widget in widgets
        if widget
        include row
```

截至现在，所有的模板 view 都被转换为 Jade 实现，应用程序的功能也全部实现。（除了第 10 章中我们会添加数据部分）

虽然程序完成了，但是对用户来说并没有太大吸引力。当然，我们可以很容易地在 header 中添加样式文件来修饰所有显示的元素，但是我们选择另一种实现方式：使用 Stylus。

8.4 使用 Stylus 完成简单的 CSS 样式

很容易在模板文件中添加样式。在 Jade 模板文件中，我们给 header.jade 文件添加样式：

```
head
  title #{title}
  meta(charset="utf-8")
  link(type="text/css"
       rel="stylesheet"
       href="/stylesheets/main.css"
       media="all")
```

这里定义的样式会对程序中所有的 view 起作用，因为所有 view 都是用了 layout 模板，而 layout 模板引用了该 header 文件。

提示

现在你肯定发现了将静态 new.html 文件转换为模板 view 的价值所在：对 header 文件的修改并不会影响静态文件，你需要手动编辑。

如果你已经开始喜欢上了 Jade 的语法，你可以在程序中使用 Stylus，用该语法描述 CSS。

首先安装 Stylus：

```
npm install stylus
```

Stylus 与 Jade 模板系统不同，它并不会创建动态的 CSS views。Stylus 所做的是在第一次访问模板或者每次模板被修改时根据 Stylus 模板生成静态样式文件。

提示

更多关于 Stylus：http://learnboost.github.com/stylus/docs/js.html。

想在程序中使用 Stylus，需要在主文件（app.js）的 require 部分引用该 module。然后需要像其他中间件一样在 configure 方法调用中引入 Stylus 中间件，传入 Stylus 模板的选项参数以及设置编译好的样式文件的目的路径。示例 8-14 显示了修改后的 app.js 文件，修改用粗体标出。

示例 8-14　在 widget 程序中加入 CSS 模板支持

```
var express = require('express')
  , routes = require('./routes')
  , map = require('./maproutecontroller')
  , http = require('http')
  , stylus = require('stylus')
  , app = express();

app.configure(function () {
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.staticCache({maxObjects:100, maxLength:512}));
  app.use(stylus.middleware({
      src: __dirname + '/views'
      ,dest: __dirname + '/public'
  }));
  app.use(express.static(__dirname + '/public'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.directory(__dirname + '/public'));
  app.use(function (req, res, next) {
    throw new Error(req.url + ' not found');
  });
  app.use(function (err, req, res, next) {
    console.log(err);
    res.send(err.message);
  });
});

app.configure('development', function () {
```

```
  app.use(express.errorHandler());
});

app.get('/', routes.index);

var prefixes = ['widgets'];

// map route to controller
prefixes.forEach(function(prefix) {
  map.mapRoute(app, prefix);
});
http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

做完以上修改后当你第一次访问 widget 程序时，你可能会注意到一个短暂的停顿。原因在于 Stylus 模块正在生成样式文件。这个过程发生在新添加或者修改一个样式模板后重启程序的时候。在样式文件生成后，提供样式的实际是生成的文件，并不会在访问每个页面时重新编译。

提示

如果对样式模板进行修改，需要重启 Express 程序。

Stylus 样式模板扩展名为.styl。源目录设置为 views，但是样式模板期望的路径是在 views 目录下有一个名为 stylesheets 的子目录。当成生静态样式文件时，会被放在目标目录下的 stylesheets 子目录中（本例中为/public）。

在习惯了 Jade 之后，你会发现 Stylus 的语法非常熟悉。每个需要添加样式的元素都被列出，缩进表示该元素的样式属性。这种语法省略了花括号，逗号以及分号的使用。

例如，设置网页的背景色为黄色，字体颜色会红色，Stylus 模板如下：

```
body
   background-color yellow
   color red
```

如果几个元素需要共享某些样式，将它们列在同一行以逗号分隔，这与 CSS 一致：

```
p, tr
   background-color yellow
   color red
```

或者可以用统一缩进写在不同行：

```
p
tr
   background-color yellow
   color red
```

如果你需要使用悬停伪类，比如:hover,:visited，语法如下：

```
textarea
input
  background-color #fff
  &:hover
    background-color cyan
```

与符号（&）表示父选择器。综合起来，以下 Stylus 模板：

```
p, tr
   background-color yellow
   color red

textarea
input
  background-color #fff
  &:hover
    background-color cyan
```

生成的 CSS 文件为：

```
p,
tr {
  background-color: #ff0;
  color: #f00;
}
textarea,
input {
  background-color: #fff;
}
textarea:hover,
input:hover {
  background-color: #0ff;
}
```

关于 Stylus 还有很多内容，我把这些留给你们作为课外练习。Stylus 官网提供了很详细的 Stylus 语法的文档。在这章结束之前，我们创建了 Stylus 样式表来优化 widget 程序的展示。

另外，我们需要给 index 列表页面的 HTMLtable 元素添加 border 和空间。还需要修改 header 的字体，删除列表前面的圆点标识符。这些都是很小的修改，但是会让 widget 程序有一个全新的展示。

示例 8-15 显示了新的样式模板。文件并不大也没没有任何复杂的 CSS。使用了最基本的东西，但某种程度上改善了程序的外观。

示例 8-15　widget 程序的 Stylus 模板

```
body
  margin 50px
table
  width 90%
  border-collapse collapse
table,td,th,caption
  border 1px solid black
td
  padding 20px
caption
  font-size larger
  background-color yellow
  padding 10px
h1
  font 1.5em Georgia, serif
ul
  list-style-type none
form
  margin 20px
  padding 20px
```

图 8-5 显示了添加几个 widget 之后的 index 页面。新的样式并不太精致，但是数据内容比之前更容易阅读和查找。

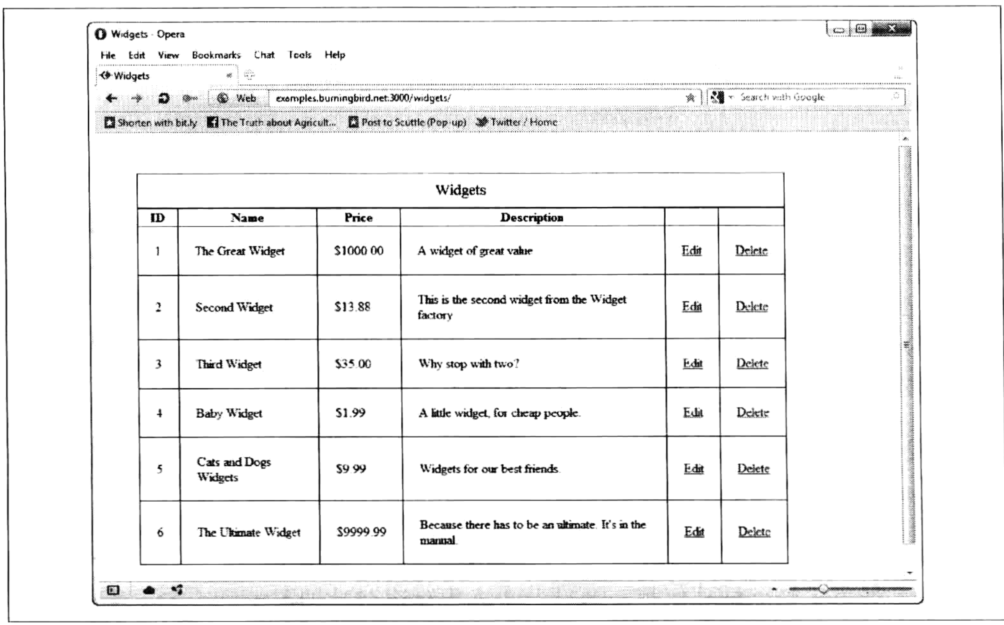


图 8-5　添加了 Stylus 样式的 Widget 程序 index 页面

第 9 章

结构化数据、Node 和 Redis

当谈到数据时，我们会想到关系型数据库以及其他一些被称为 NoSQL 的数据库。在 NoSQL 分类中，有一种基于键值对（key/value pairs）的结构化数据类型，它通常被存储在内存以支持快速访问。三种最流行的基于内存键值对的存储系统是 Memcached，Cassandra 和 Redis。Node 开发人员应该值得高兴，因为 Node 对这三种存储系统都提供支持。

Memcached 主要被用来缓存数据请求，以便快速存取在内存中缓存的数据。它对分布式计算的支持也很好，但是对于复杂数据类型的支持却比较有限。所以，对于需要做很多查询的应用程序来说 Memcached 非常好用，而对于需要进行大量数据读写操作的应用程序来说却不是很合适。但是 Redis 能够支持后一种应用程序，因为它有着卓越的数据存储支持。此外，Redis 支持持久性存储，能够提供比 Memcached 更好的灵活性，支持更多数据类型。然而，不同于 Memcached 的是，Redis 只能工作在一台机器上。

Redis 和 Cassandra 的比较结果与上面类似。与 Memcached 一样，Cassandra 支持集群，而且同样对数据结构的支持比较有限。它对 ad hoc 查询支持的非常好，而 Redis 则不适合这种查询方式。不过 Redis 简单易用，不复杂，一般情况下也比 Cassandra 有更好的效率。由于这些以及其他一些原因，Redis 已经获得了非常多的 Node 开发者的支持，这就是为什么我在本章中选择使用它讲解有关键值对存储的原因，而非 Memcached 或 Cassandra。

在前几章中，我们使用了类似于教程式的风格来讲解和说明相关技术，本章中我将稍作改变，通过三个包含有具体功能的实例来说明有关 Node 和 Redis 的相关技术：

- 建立一个游戏排行榜

- 创建一个消息队列
- 跟踪网页统计

这些应用程序还会使用前几个章中提到的模块和技术，如 Jade 模板系统（在第 8 章），Async 模块（在第 5 章），Express 框架（在第 7 和第 8 章）。

提示

Redis 的网址 *http://redis.io/*。更多有关 Memcached 的信息 *http://memcached.org/*，以及 Apache Cassandra 的网址 *http://cassandra.apache.org/*。

9.1 Node 和 Redis

支持 Redis 的模块有很多，例如 Redback，它提供了一个高抽象层次的接口。但在本章中，我们将关注另一个由 Matt Ranney 编写的 node_redis 模块（之后我会使用"redis"表示该模块）。我喜欢 redis 是因为它提供了一个简单而优雅的接口来直接执行 Redis 命令，这样你就可以充分利用自己对数据的了解并在给予系统最小干预的情况下来操作存储系统了。

提示

redis 的 GitHub 页面在 *https://github.com/mranney/node_redis*。

使用 npm 安装 redis 模块：

```
npm install redis
```

我同样也建议使用 hiredis 库，因为它是非阻塞的且能提高性能。使用下面的命令安装它：

```
npm install hiredis redis
```

想要在 Node 应用程序中使用 redis 的话，首先要包含模块：

```
var redis = require('redis');
```

然后，使用 createClient 方法创建一个 Redis 客户端：

```
var client = redis.createClient();
```

createClient 方法包含三个可选参数：port，host 和 options。host 默认值为 12 7.0.0.1，port 默认值为 6379。port 的默认值也是 Redis 服务器默认使用的端口号，所以如果 Redis

服务器与Node应用程序都托管在同一台机器上的话，就无需修改这些默认值了。

第三个参数是一个对象，它所支持的选项如下：

parser

Redis 协议 replay 解析器，默认为 hiredis。也可以使用 javascript。

return_buffers

默认为 false。如果为 true，所有的回复将以 Node buffer 对象返回，而不是字符串。

detect_buffers

默认为 false。若为 true 并且有原始操作命令被缓存起来时，回复信息将被包装在 Node buffer 对象中。

socket_nodelay

默认为 true，指定是否在 TCP 流中调用 setNoDelay。

no_ready_check

默认为 false。设置为 true 时，会阻止"ready check"被发送到服务器，以便准备更多的命令。

在你更熟悉 Node 和 Redis 前，最好使用默认设置。

一旦建立了客户端与 Redis 数据存储系统的连接，你就可以发送命令给服务器直到调用 client.quit 方法关闭该连接。如果想强制关闭连接，你可以调用 client.end 方法，该方法不会等待答复解析完毕。如果你的应用程序被卡住，或者你想重新开始，client.end 方法是一个好的选择。

通过客户端连接发出 Redis 命令是一个相当直观的过程，所有的命令都暴露在客户对象的各个方法中，命令所需的参数则作为方法参数传递。同样遵循 Node 程序开发的特点，这些方法的最后一个参数都是回调函数，用于接收并处理返回的错误信息或任何对应于 Redis 命令的数据或答复。

在下面的代码中，我们使用 client.hset 方法设置哈希属性：

```
client.hset("hashid", "propname", "propvalue", function(err, reply) {
   // do something with error or reply
});
```

hset 命令可设置一个值，因此没有返回数据，只返回 Redis 的确认信息。如果你调用了一个可以返回多个值的方法，例如 client.hvals，那么在回调函数的第二个参数中将会保存字符串数组或者是对象数组：

```
client.hvals(obj.member, function (err, replies) {
   if (err) {
       return console.error("error response - " + err);
   }

   console.log(replies.length + " replies:");
   replies.forEach(function (reply, i) {
     console.log(" " + i + ": " + reply);
   });
});
```

因为 Node 回调无所不在，但很多 Redis 命令的返回信息中仅包含了类似于操作成功的确认回复。因此，redis 模块提供了 *redis*.print 方法，你可以将其作为回调函数的最后一个参数传入：

```
client.set("somekey", "somevalue", redis.print);
```

redis.print 方法会将错误信息或答复内容输出到控制台并返回。

现在，你是否已经迫不及待的想要知道 redis 模块是如何应用的了吧？接下来我们就开始在实际的应用程序中使用它吧。

9.2 构建游戏得分排行榜

我们可以用 Redis 创建一个游戏排行榜。排行榜通常用来记录电子游戏的得分，包括家用电脑游戏、智能手机游戏，以及平板电脑游戏。OpenFeint 就是被广泛使用的一个排行榜，它允许游戏玩家在线创建个人档案并记录各种游戏的得分。这样玩家就可以与朋友竞争，还可以尝试挑战任何游戏的最高得分。

这是一个可以混合使用多种数据存储系统来实现的应用程序。玩家的个人档案可以保存在一个关系型数据存储系统中，而他们的游戏得分则可以保存在 Redis 存储系统中。分数信息的数据需求很简单，但会被大量的用户频繁地访问和修改。据一个 Facebook 游戏开发人员估计，在游戏高峰时间，大约有 10000 个并发用户以每分钟 200000 次请求的速度访问排行榜。然而，想要让系统处理这些请求其实并不困难，因为数据不复杂，也没有必要使用事务。坦率地说，如果使用关系或文档数据库来处理这种需求就太过于杀鸡用牛刀了，而比较理想的选择则是使用像 Redis 一类的键值对数据存储系统。

对于这个应用程序，Redis 的 hash 和 sorted set 是最适用的数据结构。选择 hash 是因为每条分数信息只需要不多的域就可以描述。通常情况下，你会存储一个成员 ID，也许还包括玩家的名字（为了不必频繁地查询或保存到关系或文档型数据库），可能还要包含游戏名称（如果系统提供多个游戏的排行榜），以及最后一次游戏的时间，当然还有游戏得分，以及任何其他相关信息。

而选择 sorted set 数据结构来追踪分数及用户名是因为通过它能够快速地访问前 10 个或者前 100 个最高游戏得分信息。

为了创建一个可以更新 Redis 数据库的应用程序，我修改了第 3 章中的 TCP 客户端及服务端应用程序。客户端会发送数据给服务端，服务端则会更新 Redis。对于一个游戏应用程序来说，比起使用 HTTP 或其他方式，使用 TCP 套接字来保存数据到服务器是非常常见的。

TCP 客户端会将我们在命令行中输入的信息发送给服务端，因此我们完全采用了示例 3-3 的代码，在此不再重复说明。但是与之前测试不同的是，当客户端运行后，我们需要输入一些文本信息。在本例中，我输入了一段 JSON 文本，描述了需要 Redis 数据库进行排序处理的分数信息。如下所示：

```
{"member" : 400, "first_name" : "Ada", "last_name" : "Lovelace", "score" :
53455,"date" : "10/10/1840"}
```

接着，我们修改了服务端应用程序，以便它能将接收到的文本信息转换为 JavaScript 对象，并能将每个成员信息保存到 hash 表中，成员编号以及分数也会被添加到 sorted set，并以游戏得分做排序。示例 9-1 显示了修改后的 TCP 服务器应用程序。

示例 9-1　可以更新 Redis 数据存储的 TCP 服务端程序

```
var net = require('net');
var redis = require('redis');

var server = net.createServer(function(conn) {
   console.log('connected');

   // create Redis client
   var client = redis.createClient();

   client.on('error', function(err) {
     console.log('Error ' + err);
   });

   // fifth database is game score database
   client.select(5);
   conn.on('data', function(data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);
```

```
        try {
           var obj = JSON.parse(data);

           // add or overwrite score
           client.hset(obj.member, "first_name", obj.first_name, redis.print);
           client.hset(obj.member, "last_name", obj.last_name, redis.print);
           client.hset(obj.member, "score", obj.score, redis.print);
           client.hset(obj.member, "date", obj.date, redis.print);

           // add to scores for Zowie!
           client.zadd("Zowie!", parseInt(obj.score), obj.member);
        } catch(err) {
           console.log(err);
        }
     });
     conn.on('close', function() {
          console.log('client closed connection');
          client.quit();
     });

}).listen(8124);

console.log('listening on port 8124');
```

当服务端被创建时，与 Redis 的连接也被建立起来；当服务端关闭时，与 Redis 的连接会被断开。另一种方法是创建一个静态的客户端连接，在当前访问请求完成前，这个连接一直存在，但是这种方法也存在弊端。关于什么时候创建 Redis 客户端的问题，可以参见本章后面的“何时创建 Redis 客户端？”部分。Redis 的数据对象转换以及数据保存代码被包含在异常处理代码块中，这样便能捕获任何由于无效或错误输入而引起的服务程序异常中止时产生的错误信息。

如上所述，我们的应用程序会更新两个不同的数据存储：一个是用于保存每个玩家得分信息（包括名称、得分和日期）的 hash，另一个 sorted set 则被用于保存按照得分排序的成员 ID 信息。成员 ID 被用来作为 hash 的 key，而在 sorted set 中游戏得分被用来作为成员 ID 的排序依据。所以，让程序能正常工作的关键就是在两个数据存储中均出现的成员 ID。

接下来，应用程序需要显示 Zowie 游戏前五名得分者的信息（该游戏以及分数信息是为了测试而虚构的）。在 sorted set 中，你可以通过 Redis 提供的 zrange 命令得到一组按照得分排序的数据。然而，这个函数返回的数据是按照游戏得分从低到高的顺序排列的，这与我们想要的得分数据的顺序恰好相反。所以，这里需要使用 Redis 提供的另一个命令 zrevrange。

为了显示得分前五名的游戏玩家，我们将会创建一个 HTTP 服务端，并将查询结果作为一个简单列表返回。为了得到一个相对体面的显示结果，我们打算使用 Jade 模板系统，但由于当前的应用程序并不是基于 Express 框架的，所以只能直接使用 Jade。

当需要在 Express 以外使用 Jade 时，首先需要读取模板文件的内容，然后调用 compile 方法并传入文件内容和一些可选项。在本例中，我只使用了 filename 选项，这是因为我在模板文件中使用了 include 指令，该指令需要使用 filename 选项。实际上，我需要的是模板的文件名及路径信息，这样就可以在 Jade 模板中包含并使用与模板同路径的其他文件了。

示例 9-2 展示了程序所使用的 Jade 模板。请注意其中的 include 指令，它能直接在文件中嵌入 CSS 代码。因此，我就没有再在应用程序中实现静态文件服务来响应浏览器对 CSS 文件的请求了。请注意我们在样式标签 style 的开始和关闭处还使用了管道符（|），它表示这个 style 标签是 HTML 语法而非 Jade 语法，以便 Jade 能正常处理 style 标签中通过 include 指令包含的文件。

示例 9-2　用于显示五个最高得分的 Jade 模板文件

```
doctype 5
html(lang="en")
  head
    title Zowie! Top Scores
    meta(charset="utf-8")
    | <style type="text/css">
    include main.css
    | </style>
  body
    table
      caption Zowie! Top Scorers!
        tr
th Score
th Name
th Date
if scores.length
  each score in scores
    if score
      tr
        td #{score.score}
        td #{score.first_name} #{score.last_name}
        td #{score.date}
```

为了呈现模板，应用程序首先读取了模板文件的内容（我们使用了同步版本的文件读取操作，因为该操作只在应用程序启动时执行一次），然后用它来编译一个模板函数：

```
var layout = require('fs').readFileSync(__dirname + '/score.jade', 'utf8');
var fn = jade.compile(layout, {filename: __dirname + '/score.jade'});
```

然后，只需要传入渲染模板所需要的数据信息，我们就可以在任何时间使用模板函数呈现 HTML 了：

```
var str = fn({scores : result});
res.end(str);
```

当看到完整的服务端应用程序后，你会对此有更好地理解。现在，让我们再来看看应用程序如何操作 Redis。

排行榜程序需要使用两个 Redis 函数：首先使用 zrevrange 得到一组分数信息，然后从这组分数信息中取得多个成员 ID 信息，再多次调用 hgetall 并传入不同的成员 ID 以获取对应玩家的详细信息。不过，正是对 hgetall 的调用让事情变得有点棘手。

当你使用一个关系型数据库时，你可以轻松地将多个表的结果合并。但在类似于 Redis 的键值对系统中，却不能实现该功能。另外，由于这是一个 Node 应用程序，我们还需要做一些额外的工作来保证每一个 Redis 调用都是异步的。

这正是流程控制模块（如 Async）的用武之地。我曾在第 5 章中对 Async 做了说明和介绍，并示例了一些异步方法（waterfall 和 parallel），有一个没有提到的方法是 series，不过此时，它是我们的理想选择。对 hgetall 函数的多次调用需要按顺序进行，以确保返回的数据是有序的，但每一个调用都是独立的，并不依赖于之前调用的处理结果。Async 的 parallel 功能会并行执行所有调用，这很好，但每个调用的执行结果则会以随机顺序返回，无法保证最先返回最高得分信息。另外，我们也没有必要使用 waterfall 方法，因为每一个步骤都不依赖于之前步骤的执行结果。如果，使用 Async 的 series 方法则能确保每一个 hgetall 调用及返回的数据都是按序进行的，同时也能保证每个调用都是相互独立没有依赖关系的。

现在，我们有了一种解决方法来保证按序调用 Redis 命令，并能以正确的顺序返回数据。但写出来的代码却看起来很笨拙，因为我们必须为每一个 Redis.hgetall 调用书写单独的步骤并处理返回结果，以便在 series 方法中使用它们。如果只需要 5 个查询结果的话，这不是什么问题，但是如果想要返回 10 个结果呢？或 100 个？手动实现每个 Redis 调用代码并将其放在 Async.series 的调用序列中是件非常繁琐并很容易出错的事情，而且代码还很难维护。

在本例中，我们通过一个循环操作，将 Redis.zrevrange 返回的数组中的每个成员 ID 值传给 MakeCallbackFunc 函数。这个辅助函数所做的是返回一个回调函数，这个回调函数会调用 Redis 的 hgetall 方法并传递参数来确定应该从 Redis 获取哪个成员的数据，然后在回调函数的最后一行调用 callback 函数（Async 需要它来取得所有步骤的处理结果）。使用 MakeCallbackFunc 函数生成的所有回调函数都会被保存在一个数组中，这个数组会作为一个参数传递给 Async.series 方法。此外，因为 redis 模块会将 hgetall 的查询结果作为对象返回，而 Async.series 函数执行完所有步骤后会将这些对象保存到一个数组，所以我们可以把最后得到的处理结果直接传递给模板引擎用以生成文本信息。

示例 9-3 是分数排行榜服务端应用程序的完整代码。虽然上述工作听起来需要很大的工作量，但实际上的代码并不多，这要感谢 Redis 异步模块以及 Async 模块的优雅和可用性了。

示例 9-3　游戏排行榜服务端程序

```
var http = require('http');
var async = require('async');
var redis = require('redis');
var jade = require('jade');

// set up Jade template
var layout = require('fs').readFileSync(__dirname + '/score.jade', 'utf8');
var fn = jade.compile(layout, {filename: __dirname + '/score.jade'});

// start Redis client
var client = redis.createClient();

// select fifth database
client.select(5);

// helper function
function makeCallbackFunc(member) {
   return function(callback) {
      client.hgetall(member, function(err, obj) {
         callback(err,obj);
      });
   };
}
http.createServer(function(req,res) {

   // first filter out icon request
   if (req.url === '/favicon.ico') {
      res.writeHead(200, {'Content-Type': 'image/x-icon'} );
      res.end();
      return;
   }

   // get scores, reverse order, top five only
   client.zrevrange('Zowie!',0,4, function(err,result) {
      var scores;
      if (err) {
         console.log(err);
         res.end('Top scores not currently available, please check back');
         return;
      }

      // create array of callback functions for Async.series call
      var callFunctions = new Array();

      // process results with makeCallbackFunc, push newly returned
      // callback into array
      for (var i = 0; i < result.length; i++) {
         callFunctions.push(makeCallbackFunc(result[i]));
      }
```

```
      // using Async series to process each callback in turn and return
      // end result as array of objects
      async.series(
         callFunctions,
         function (err, result) {
            if (err) {
               console.log(err);
               res.end('Scores not available');
               return;
            }

            // pass object array to template engine
            var str = fn({scores : result});
            res.end(str);
        });
     });
}).listen(3000);

console.log('Server running on 3000/');
```

在创建 HTTP 服务端对象之前，我们建立了 Jade 模板函数，同时还建立了一个 Redis 客户端。当一个新的请求到达服务器时，我们先筛选出所有有关 favicon.ico 文件的请求（因为请求 favicon.ico 并不需要调用 Redis），然后使用 zrevrange 方法筛选出前五个最高得分。一旦应用程序取得了分数，它便会使用 Async.series 方法来处理 Redis 请求以保证调用的时间和顺序，以便能得到一个有序结果。最终查询结果数组会被传递给 Jade 模板引擎。

图 9-1 展示了在添加了几个测试数据后的应用程序处理结果。

Zowie! Top Scores

http://examples.burningbird.net:8124/

Zowie! Top Scorers!

Score	Name	Date
53455	Ada Lovelace	10/10/1840
49555	Grace Hopper	03/24/1950
20000	Abe Lincoln	03/10/2012
15000	Mark Twain	12/10/2011
10000	Shelley Powers	01/01/2012

图 9-1 Zowie 游戏得分排行榜

9.3 创建消息队列

消息队列是一个将输入信息作为某种通信形式并将信息存储在队列中的应用程序。消息在被接收前一直存储在队列中，当消息接收器接收它们时，消息会被弹出队列并传送到接收器（消息数量可以是一个或批量的）。这种通信是异步的，因为保存消息的应用程序并不需要与消息接收器连接。同样的，消息接收应用程序也不需要与存储消息的应用程序连接。

Redis 是实现这种类型应用程序的理想存储媒介。当产生消息的应用程序需要保存消息时，它将消息放在消息队列的尾部。当消息接收应用程序检索需要的信息时，它会从消息队列前面取得消息。

为了更好地说明消息队列，我创建了一个 Node 应用程序来访问几个不同子域的 Web 日志文件。该应用程序使用一个 Node 子进程和 Unix 的 tail -f 命令来取得不同日志文件的新增条目。应用程序使用两个正则表达式对象处理这些日志条目：一个用于提取访问的资源路径，另一个测试资源是否是一个图像文件。如果访问的资源是一个图像文件，应用程序会通过 TCP 消息将资源的 URL 信息发送到消息队列应用程序。

消息队列应用程序所做的只是监听端口 3000 上传入的消息，然后将它们保存到 Redis 中。

接下来还需要一个 Web 服务端应用程序，它会监听端口 8124 上的用户请求。对于每一个请求，它会访问 Redis 数据库并取出图像数据存储区中的第一个条目，再通过 response 对象将其返回给客户端。如果在获取图像资源的时候 Redis 数据库返回 null，那么程序会输出一条消息来表示已经处理到了消息队列的末尾。

示例 9-4 展示了第一个应用程序的代码，它用于处理 Web 日志条目。Unix 的 tail 命令可以用来显示一个文本文件的最后几行（或管道的数据）。当使用-f 标志时，该实用程序就会在显示文件的最后几行后进行等待，监听是否有新条目被添加到文件。当有新条目被添加时，它就返回该新条目内容。tail -f 命令可以同时监听几个不同文件的内容，当任何文件有新内容到达时，它会输出这条新内容并打上对应的文件标签以区别不同的信息源。我们的程序并不关心最新条目是在哪一个日志文件中产生的，而只是关心条目内容。

一旦应用程序获得日志条目，它会执行两个正则表达式来匹配条目数据，以便确定是否为图像资源的访问（对后缀名为.jpg、.gif、.svg 或.png 文件的访问）。如果找

到匹配项，应用程序会将资源 URL 发送到消息队列应用程序中（一个 TCP 服务端）。

示例 9-4　Web 日志处理程序：处理条目并发送图像资源 URL 到消息队列程序

```
var spawn = require('child_process').spawn;
var net = require('net');

var client = new net.Socket();
client.setEncoding('utf8');

// connect to TCP server
client.connect ('3000','examples.burningbird.net', function() {
    console.log('connected to server');
});

// start child process
var logs = spawn('tail', ['-f',
        '/home/main/logs/access.log',
        '/home/tech/logs/access.log',
        '/home/shelleypowers/logs/access.log',
        '/home/green/logs/access.log',
        '/home/puppies/logs/access.log']);

// process child process data
logs.stdout.setEncoding('utf8');
logs.stdout.on('data', function(data) {
   // resource URL
   var re = /GET\s(\S+)\sHTTP/g;

   // graphics test
   var re2 = /\.gif|\.png|\.jpg|\.svg/;

   // extract URL, test for graphics
   // store in Redis if found
   var parts = re.exec(data);
   console.log(parts[1]);
   var tst = re2.test(parts[1]);
   if (tst) {
      client.write(parts[1]);
   }
});
logs.stderr.on('data', function(data) {
   console.log('stderr: ' + data);
});

logs.on('exit', function(code) {
   console.log('child process exited with code ' + code);
   client.end();
});
```

运行日志处理程序后可以得到类似于下面的控制台输出信息，包括了常见的 Web 访问条目，并以粗体显示了对图像资源的访问条目：

```
/robots.txt
/weblog
```

```
/writings/fiction?page=10
/images/kite.jpg
/node/145
/culture/book-reviews/silkworm
/feed/atom/
/images/visitmologo.jpg
/images/canvas.png
/sites/default/files/paws.png
/feeds/atom.xml
```

示例 9-5 包含了消息队列程序的源代码。这是一个简单的 Node 应用程序，只是启动了 TCP 服务端，并侦听传入的消息。当它接收到消息后，会从消息中提取数据并存储在 Redis 的数据库中。该应用程序使用了 Redis 的 rpush 命令来将数据保存到名为 images 的列表中（代码中的粗体部分）。

示例 9-5　消息队列程序：处理传入的消息，并将其保存到 Redis 列表

```
var net = require('net');
var redis = require('redis');

var server = net.createServer(function(conn) {
   console.log('connected');

   // create Redis client
   var client = redis.createClient();
   client.on('error', function(err) {
     console.log('Error ' + err);
   });

   // sixth database is image queue
   client.select(6);
   // listen for incoming data
   conn.on('data', function(data) {
      console.log(data + ' from ' + conn.remoteAddress + ' ' +
        conn.remotePort);

      // store data
      client.rpush('images',data);
   });

}).listen(3000);
server.on('close', function(err) {
   client.quit();
});

console.log('listening on port 3000');
```

运行消息队列应用程序后，对应的控制台输出类似于下面这样：

```
listening on port 3000
connected
/images/venus.png from 173.255.206.103 39519
```

```
/images/kite.jpg from 173.255.206.103 39519
/images/visitmologo.jpg from 173.255.206.103 39519
/images/canvas.png from 173.255.206.103 39519
/sites/default/files/paws.png from 173.255.206.103 39519
```

示例 9-6 是最后一个与消息队列相关的示例程序，它是一个 HTTP 服务端并在端口 8124 监听来自用户的请求信息。当该程序接收到 HTTP 请求时，它会访问 Redis 数据库，并从 images 列表取得图像信息条目，然后将其输出给用户。如果列表中已经没有更多条目时（即如果 Redis 的答复返回 null），它则会输出消息队列为空的提示。

示例 9-6　HTTP 服务端：从 Redis 列表中取得消息，并返回给用户

```
var redis = require("redis"),
    http = require('http');

var messageServer = http.createServer();

// listen for incoming request
messageServer.on('request', function (req, res) {

   // first filter out icon request
   if (req.url === '/favicon.ico') {
      res.writeHead(200, {'Content-Type': 'image/x-icon'} );
      res.end();
      return;
   }

  // create Redis client
  var client = redis.createClient();


  client.on('error', function (err) {
    console.log('Error ' + err);
  });

  // set database to 1
  client.select(6);

   client.lpop('images', function(err, reply) {
      if(err) {
        return console.error('error response ' + err);
      }

      // if data
      if (reply) {
         res.write(reply + '\n');
      } else {
         res.write('End of queue\n');
      }
      res.end();
   });
   client.quit();
```

```
});

messageServer.listen(8124);

console.log('listening on 8124');
```

每次使用 Web 浏览器访问该 HTTP 服务程序时（刷新浏览器页面），都会得到一个有关图像资源的请求信息，直到消息队列为空。

> **何时创建 Redis 客户端？**
>
> 在本章的一些示例代码中，Redis 客户端与应用程序具有相同生命周期。但是，我也可能在创建 Redis 客户端并在执行完相应的 Redis 命令后需要尽快释放它。那么，什么时候应该创建一个持久的 Redis 连接？而什么时候立即释放它会比较好呢？
>
> 这是个好问题。
>
> 为了测试这两种不同的使用方法，我创建了一个 TCP 服务端并监听请求，然后简单地将请求内容存储在一个 Redis 数据库中。然后，我又创建了另一个应用程序，把它作为 TCP 客户端，能将一个对象放在 TCP 报文中并发送到到服务端。
>
> 为了测试，我用 ApacheBench 程序并发地运行多个客户端程序并观察响应时间。第一次测试使用长生命周期的 Redis 客户端连接，而第二次测试使用立即释放 Redis 客户端连接。
>
> 我期望具有持久 Redis 客户端连接的程序会更快一些，测试结果表明我是对的。对于具有持久 Redis 连接的程序，当测试进行到大约一半时，应用程序的响应速度在一个短暂的时间内大幅下降，然后又恢复到较快的速度。
>
> 当然，最有可能发生的是，Redis 数据库的请求队列阻塞了 Node 应用程序，至少是暂时的，直到队列被释放。我在测试针对每个请求都打开和关闭 Redis 连接的服务端应用程序时，并没有碰到同样的问题。这可能是因为建立 Redis 连接的过程需要额外开销，从而放缓了对 Redis 的并发请求速度，使 Redis 有足够时间处理请求。
>
> 在第 14 章和第 16 章，我会使用 ApacheBench 进行更多相关测试，并会介绍另一些性能测试工具。

9.4 为 Express 应用程序添加统计中间件

Redis 的最初设计原本是用来实现统计应用程序的。因此，使用 Redis 的理想情况是：一个简单的数据存储，能支持快速且频繁的写入操作，而且需要支持统计功能。

在本小节中，我们将使用 Redis 来为 widget 应用程序（之前章节中创建的示例程序）提供统计功能。统计信息被保存在两个集合中：一个包含了所有曾访问过 widget 程序的客户端的 IP 地址，另一个保存了不同 widget 程序资源被访问的次数。为了实现该功能，我们需要使用 Redis 的 set 数据结构及递增数字字符串功能。另外，应用程序还使用了 Redis 的事务控制 multi，以便能在同一时间取得两个独立数据集合的内容。

第一步应该做的是为应用程序添加新的中间件，该中间件能将访问信息保存到 Redis 数据库中。中间件使用了 Redis 的 set 集合并使用 sadd 方法将每个访问者的 IP 地址添加到集合中。set 集合可以确保一个已经存在于数据库中的值不会被记录两次，因为我们关注于收集访问者的 IP 地址信息，但无需记录该 IP 用户的每次访问。我们还使用了 Redis 的增量功能，但并没有采用用于递增字符串的 incr 方法，而是使用了 hincrby 方法。因为资源 URL 及其关联的访问计数器是作为哈希存储在 Redis 中的。

示例 9-7 展示了中间件源代码，它保存在 stats.js 文件中。中间件使用了 Redis 数据库，并将访问者的 IP 地址信息保存在一组以 ip 为标识的集合中，而资源 URL 及访问计数器被保存在一个以 myurls 为标识的哈希中。

示例 9-7　基于 Redis 的统计中间件

```
var redis = require('redis');
module.exports = function getStats() {

   return function getStats(req, res, next) {
      // create Redis client
      var client = redis.createClient();

      client.on('error', function (err) {
         console.log('Error ' + err);
      });

      // set database to 2
      client.select(2);

      // add IP to set
      client.sadd('ip',req.socket.remoteAddress);

      // increment resource count
      client.hincrby('myurls',req.url, 1);

     client.quit();
      next();
   }
}
```

由于统计接口需要能够被顶级域名访问，因此我们将在 routes 子目录为 index.js 文

件添加相关的路由代码。

首先，需要在主程序文件中添加路由，并将其放在顶级 index 路由之后：

```
app.get('/', routes.index);
app.get('/stats',routes.stats);
```

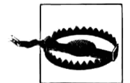
在 routes.stas 的实现代码中，我们调用了 multi 函数来使用 Redis 的事务控制功能。这段代码访问了两组数据：一组是通过 smembers 返回的 IP 地址信息（不会包含重复 IP），另一组是通过 hgetall 返回的包含有资源 URL 及访问计数器的哈希表。smembers 函数和 hgetall 函数被按序调用，执行结果被按序追加到一个数组中，并最终作为 exec 方法回调函数的参数，如示例 9-8 所示。一旦拿到处理结果，我们便用它来渲染 stats 视图。在 index.js 文件中新添加的代码以粗体显示。

示例 9-8　包含有统计程序控制代码的路由索引文件

```
var redis = require('redis');

// home page
exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};


// stats
exports.stats = function(req, res){

   var client = redis.createClient();

   client.select(2);

   // Redis transaction to gather data
   client.multi()
   .smembers('ip')
   .hgetall('myurls')
   .exec(function(err, results) {
        var ips = results[0];
        var urls = results[1];
        res.render('stats',{ title: 'Stats', ips : ips, urls : urls});
        client.quit();
   });
};
```

我前面提到的 multi 和 exec 是 Redis 的事务控制命令。它们与关系数据库的事务控制不是一回事。multi 会收集一组 Redis 命令，在执行 exec 时，它会按照顺序处理这一组命令。这种功能在 Node 中是非常有用的，因为它提供了一种一次性获取多个集合数据的方式，而无需再使用嵌套回调函数或流程控制模块（如 Step 和 Async）。

说了这么多，不要让这些串起来的 Redis 命令迷惑了你，不要认为后一条命令的执行需要依赖于前一条命令的执行结果。实际上每个 Redis 命令的处理都是相互独立的，只是所有处理结果会被按序保存在一个数组中并一次性返回。



警告

事务的执行过程中是没有锁的，这与关系型数据库的事务有所不同。所以在 Redis 数据库查询过程中的任何修改都可能会影响查询结果。

最后我们还需要为应用程序添加视图，创建一个 Jade 模板。这个模板非常简单：所有 IP 地址信息会在一个无序 list 中呈现，而 URL 和计数统计信息则显示在一个 table 中。我们可以在 Jade 文件中使用 for ... in 语法来遍历包含了 IP 地址信息的数组，而 hgetall 方法所返回对象中的属性名称和属性值则可以通过 each ... in 语法取得。示例 9-9 展示了模板文件的源代码。

示例 9-9　用于统计应用程序的 Jade 模板

```
extends layout

block content
  h1= title

  h2 Visitor IP Addresses
  ul
    for ip in ips
      li=ip

  table
    caption Page Visits
    each val, key in urls
      tr
        td #{key}
        td #{val}
```

当多个不同 IP 地址访问 widget 应用的不同资源页面后，我们会得到一个类似于图 9-2 所示的统计页面。

在使用 hincrby 函数前是否必须先创建哈希表呢？答案是否定的。如果对应的哈希键不存在的话，Redis 会自动创建它并初始化哈希值为 0，然后再对其做递增操作。只有在对应哈希键存在但哈希值不为数字型字符串时，hincrby 操作才会失败，因为我们不能对非数字型字符串的哈希值做递增操作。

另一种实现资源访问计数的方法是：使用 Redis 字符串，将资源 URL 作为 key：

```
client.incr(url);
```

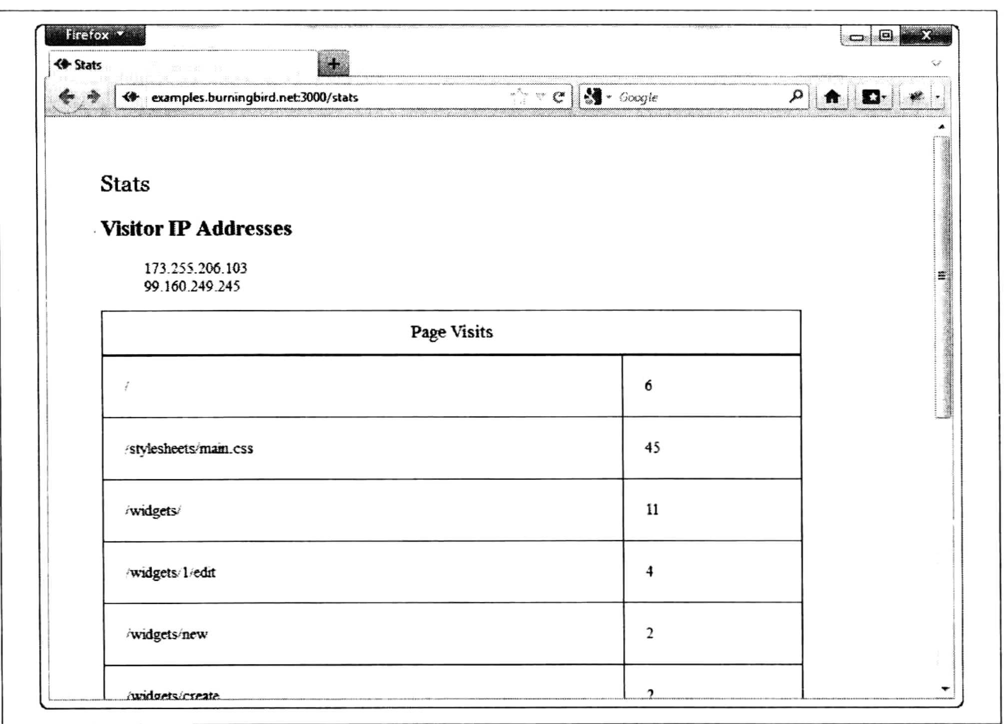


图 9-2　基于 Redis 的统计页面

然而，这种方法意味着程序必须取得所有 key（即所有资源的 URL 地址），然后再取得每个 URL 的计数器值。此时，我们无法单独使用 multi 来完成所有工作。如果再考虑到访问数据的异步特性，最终就必须使用嵌套回调或其他方法来将所有这些数据整合在一起。

因此，当可以使用 Redis 内置 hash 以及 hincrby 命令来完成这些功能时，我们就没有必要再花费更多精力来实现它们了。

第 10 章

Node 和 MongoDB：文档中心数据

第 9 章介绍了一个非常受欢迎的 NoSQL 数据库结构（Redis 的 key/value 对），本章介绍另一个：MongoDB 以文档为中心的数据存储。

MongoDB 有别于其他关系型数据库系统，比如 MySQL。区别在于 MongoDB 支持将数据结构存储为文档，而不是实现传统意义的 table。这些文档被编码为 BSON，是 JSON 的二进制格式，这很大程度上解释了为什么 MongoDB 很受 JavaScript 开发人员欢迎。使用 MongoDB，你会拥有一个 BSON 的文档而不是一个 table 中的某行，你会有一系列数据集合而不是一个 table。

MongoDB 并不是唯一的以文档为中心的数据库，其他一些这类型的数据存储包括 Apache 的 CouchDB，Amazon 的 SimpleDB，甚至是很久远的 Lotus Notes。很多现在流行的基于文档的数据存储对 Node 都有一定支持，但是以 MongoDB 和 CouchDB 为首。我选择 MongoDB 而不是 CouchDB 的原因与选择 Express 架构的原因一致：我觉得对一个不太接触二级技术（在这里指数据存储）的人来说更容易学习 Node 的使用而不用关注过多非 Node 的技术。通过 MongoDB 我们可以直接查询数据，而 CouchDB 需要了解 view 的概念。高级别的抽象需要更多时间。在我看来，使用 MongoDB 比 CouchDB 能更快地达到目的。

MongoDB 有许多模块可以使用，我们关注两个：MongoDB 原生 Node.js 驱动（用 JavaScript 编写的驱动）和 Mongoose，一个对象模型工具提供了 ORM（对象关系映射，object-relational mapping）支持。

提示

尽管本章不会讲到太多关于 MongoDB 的工作原理，但是即使你以前没有用过这种数据库你也应该可以跟上以下这些例子。更多关于 MongoDB 的信息包括安装帮助参考：http://www.mongodb.org/。

10.1 MongoDB Native Node.js Driver（MongoDB 原生 Node.js 驱动）

MongoDB Native Node.js Driver 模块是 MongoDB 自带的对 Node 的驱动。该驱动发出的 MongoDB 指令与 MongoDB 客户端接口发出的指令基本一致。

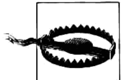
提示

Node-mongodb-nativeGitHub 页面：https://github.com/mongodb/node-mongodb- native，文档参考：http://mongodb.github.com/node-mongodb-native。

安装完成 MongoDB 之后（根据 MongoDB 官网的指南），启动数据库，安装 MongoDB Native Node.js Driver：

```
npm install mongodb
```

在开始学习之后的例子之前，确保 MongoDB 本地安装完成并运行。



警告

如果你已经使用了 MongoDB，在开始本章的例子之前确保对数据库中的数据做好备份。

10.1.1 MongoDB 入门

你需要引用模块才能使用 MongoDB 驱动：

```
var mongodb = require('mongodb');
```

之后构造 mongodb.Server 对象来建立 MongoDB 数据库的连接：

```
var server = new mongodb.Server('localhost', :27017, {auto_reconnect:true});
```

所有与数据库的连接都是 TCP 连接。Server 的构造函数前两个参数为 host 主机号和 port 端口号，本例中为 localhost 和 27017 端口。第三个参数可选。在代码中，

auto_reconnect 选项被设置为 true，表示如果连接断开 driver 会自动进行重连。另一个选项是 poolSize，决定了并发的 TCP 连接数量。

提示

MongoDB 为每个连接新建一个线程，这就是为什么数据库创建者推荐开发人员使用连接池。

完成了与 MongoDB 的连接后，就可以创建一个数据库或者连接到一个现有的数据库。可以构造 mongodb.Db 对象创建数据库：

```
var db = new mongodb.Db('mydb', server);
```

第一个参数为数据库名称，第二个为建立好连接的 MongoDB server。第三个参数表示选项。默认选项对于本章需要做的工作来说已经足够，MongoDB 文档中描述了不同选项的意义以及值的设定，在这里不再重复。

如果你之前没有使用过 MongoDB，你可能会注意到在代码中并没有提供用户名和密码作为认证。默认情况下，MongoDB 不需要认证。在没有验证的情况下，需要确保数据库运行在一个安全的环境下。这意味着 MongoDB 只允许来自于信任的主机的连接，主要是 localhost 地址。

10.1.2 定义、创建以及销毁 MongoDB Collection

MongoDB collection 与关系型数据库中的 table 从根本上来说功能一致，但是与 table 并没有任何形式上的相似之处。

当你定义一个 MongoDBcolletion 的时候，你可以描述是否希望立即创建这个 colletion 对象，或者在添加第一行数据后再创建。使用 MongoDB driver，以下两行代码描述了不同的情况：第一行并没有创建实际的 colletion，第二个则创建了：

```
db.collection('mycollection', function(err, collection{}));
db.createCollection('mycollection', function(err, collection{}));
```

以上两个方法都可以接收第二个参数作为可选项，{safe：true}，该参数用于在 db.collection 中 collection 不存在或者 db.createCollection 中 collection 存在的情况下告知 driver 错误信息：

```
db.collection('mycollection', {safe:true},function(err, collection{}));
db.createCollection('mycollection', {safe:true},function(err, collection{}));
```

如果你对一个已经存在的 collection 使用 db.createCollection 方法，你可以直接访问

该 collection——driver 并不会覆盖这一部分。两种方法都在 callback 函数中返回 collection 对象，你可以对其进行添加，修改或者获取数据。

如果希望彻底销毁一个 collection，可以使用 db.dropCollection：

```
db.dropCollection('mycollection', function(err, result){});
```

值得注意的是，如果你希望按顺序处理命令，所有这些方法都是同步的，并且依赖于嵌套的回调函数。这在下一节我们为 collection 添加数据的时候会详细说明。

10.1.3 为 Collection 添加数据

在开始了解如果向 collection 中添加数据以及实现功能之前，我想花点时间讨论下数据类型。更确切地说，我想重复一下 MongoDB driver 文档中提到的数据类型，因为 JavaScript 的使用导致了 driver 与 MongoDB 之间一些有趣的约定。

表 10-1 显示了一些 MongoDB 支持的数据类型以及对应的 JavaScript 类型。注意到大部分的数据类型转换很彻底，不会带来任何超出预期的影响。但是一些类型需要一些背后的处理机制，你应当对这部分有所了解。并且一些数据类型由 MongoDB Native Node.js Driver 提供，并没有对等的 MongoDB 数值。Driver 将我们提供的数据转换为 MongoDB 可以接收的数据。

表 10-1　Node.js MongoDB driver 与 MongoDB 数据类型的映射关系

MongoDB 类型	JavaScript 类型	备注/例子
JSON 数组	Array[1,2,3]	[1,2,3].
String	String	Utf8 编码
Boolean	Boolean	True，false
Integer	Number	MongoDB 支持 32-64bit 的数字，JavaScript number 是 64bitfloat。MongoDB driver 首先尝试将数值转换为 32bit，如果失败，转换为 64bit，如果依然失败，则转换为 Long 类型
Interger	Long Class	Long 类型支持 64bit interger
Float	Number	
Float	Double Class	用于表示 float 值的特殊类型
Date	Date	
Regular expression	RegExp	
Null	Null	
Object	Object	
Object id	ObjectID class	用于表示 MongoDB id 的特殊类

续表

MongoDB 类型	JavaScript 类型	备注/例子
Binary data	Binary Class Code Class DbRef Class Symbol class	存储二进制数据类型 存储 JavaScript 函数和方法 存储对另一个文档的引用 描述符号（对于符号语言不仅是 JavaScript）

当你已经有了 collection 之后，就可以向 collection 中添加文档。数据格式为 JSON，所以你可以创建一个 JSON 对象，然后直接将其添加到 collection。

示例 10-1 创建了第一个 MongoDBcollection（名为 Widgets），并添加了两个文档来说明如何向 collection 中添加数据。你可能会多次运行这个例子，所以先用 remove 方法删除已有的 collection 文档以防创建失败。remove 方法接收三个可选参数：

- 文档的选择器。如果没有该参数，所有文档会被删除；
- safe 模式标识，safe {true | {w:n, wtimeout:n} | {fsync:true}, default:false}；
- 回调函数（如果 safe 模式被设置为 true 则必须要有回调函数）。

在例子中，程序使用了 safe 模式的 remove 方法，第一个参数为 null（作为参数的占位符，表示所有文档会被删除），并提供了回调函数。当文档被删除后，程序会插入两个新的文档，第二个文档插入时使用了 safe 模式。程序将第二次插入的结果输出到控制台。

insert 方法也接收三个参数：文档或者带添加的文档，可选参数，回调函数。你可以用数组一次插入多个数据，insert 方法的可选参数为：

Safe 模式

```
safe { true | {w:n, wtimeout:n} | {fsync:true}, default:false}
```

keepGoing

设置为 true 时，当插入文档报错时会继续执行后续代码。

serializeFunctions

文档中的序列化方法。

insert 方法的调用是异步的，意味着没办法保证第一个文档一定在第二个文档之前插入。但这一点在 widget 这个程序中不构成问题，至少在这个例子中。本章稍后的部分我们会详细了解关于数据库异步操作的问题。

示例 10-1　创建/打开数据库，删除所有文档并添加两个新的文档

```
var mongodb = require('mongodb');

var server = new mongodb.Server('localhost', 27017, {auto_reconnect:true});
var db = new mongodb.Db('exampleDb', server);
// 打开数据库连接
db.open(function (err, db) {
  if (!err) {

      // 访问或者创建 widgets collection
      db.collection('widgets', function (err, collection) {

        // 删除所有 widgets 数据文档
        collection.remove(null, {safe:true}, function (err, result) {
           if (!err) {
             console.log('result of remove ' + result);

             // 创建两条数据
             var widget1 = {title:'First Great widget',
                            desc:'greatest widget of all',
                            price:14.99};
             var widget2 = {title:'Second Great widget',
                            desc:'second greatest widget of all',
                            price:29.99};

             collection.insert(widget1);

             collection.insert(widget2, {safe:true}, function (err, result) {
                if (err) {
                    console.log(err);
                } else {
                    console.log(result);

                    //关闭数据库连接
                    db.close();
                }
             });
           }
        });
      });
  }
});
```

第二次数据插入完成后输出到控制台的内容是一个变量：

```
[ { title:'Second Great widget',
   desc:'second greatest widget of all',
   price:29.99,
   _id:4fc108e2f6b7a3e252000002 } ]
```

MongoDB 为每一个文档都生成唯一的系统标识符。你可以用这个标识符访问特定文

档，但是你可以为每个文档添加一个更有意义的唯一标示符——可以由使用的语境决定。

我们之前提到过，可以用数组的形式一次添加多个文档。以下代码演示了两个widget记录如何用一条命令插入数据库的，同时也使用了id：

```
//创建两条数据
var widget1 = {id:1, title:'First Great widget',
              desc:'greatest widget of all',
              price:14.99};
var widget2 = {id:2, title:'Second Great widget',
              desc:'second greatest widget of all',
              price:29.99};
collection.insert([widget1, widget2], {safe:true},
                                        function(err, result) {
   if (err) {
       console.log(err);
   } else {
       console.log(result);

       // close database
       db.close();
   }
});
```

如果你选择批量处理文档数据，你可能需要设置keepGoing属性为true。这样即使中间的某个文档操作失败也可以继续后续文档的操作。默认情况下，如果一个操作失败程序就会终止。

10.1.4 查询数据

对于MongoDB Native Node.js Driver来说有四种查询数据的方法：

find

返回查询语句查到的所有文档指示。

findOne

返回查询语句查到的第一个文档。

findAndRemove

找到查询的文档并删除。

findAndModify

找到查询的文档并进行操作（比如remove或者upsert）。

本节中我会使用 collection.find 和 collection.findOne 两种方法，另外两个方法会在 10.1.5 小节讲到。

collection.find 和 collection.findOne 都支持以下三个参数：查询语句，可选参数，回调函数。可选参数和回调函数都是可选项，并且这两个方法的选项可选值非常多。表 10-2 显示了所有选项，默认值以及对每个选项的描述。

表 10-2

选　项	默 认 值	描　述
Limit	数字，默认为 0	显示返回的文档数目（0 表示无限制）
Sort	数组索引	对查询语句返回的文档排序
Field	Object	查询语句返回的 field。使用属性名作为 key，值为 1 表示返回该 field；为 0 表示不返回该 field；比如{'prop':1}或者{'prop':0}，但不能同时出现
Skip	数字，默认为 0	表示 skip n 个文档（用于分页）
Hint	Object	告诉数据库使用特定的索引，{'_id':1}
Explain	Boolean，默认为 false	解释查询语句而不是返回数据
Snapshot	Boolean，默认为 false	抓拍查询语句（必须激活 MongoDB 的 journaling）
Timeout	Boolean，默认为 false	超时
Tailable	Boolean，默认为 false	XXXXXXXXX
batchSize	数字，默认为 0	对结果遍历时 getMoreCommand 的 batchSize
returnKey	Boolean，默认为 false	只返回索引 key
maxScan	数字	设置可查询的数据项数目
Min	数字	设置索引边界
Max	数字	设置索引边界
showDiscLoc	Boolean，默认为 false	显示结果在磁盘中的位置
Comment	String	为查询在 log 中添加描述
Raw	Boolean，默认为 false	返回 BSON 结果作为原始缓冲文档
Read	Boolean，默认为 false	将查询定向到二级服务器

尽管大部分查询只使用到上述列表中的一小部分选项，但是这些选项增加了查询过程的灵活性。例子中只会涉及其中一些选项，所以推荐你们自己尝试一下其他的选项。

对 collection 中所有文档内容来说最简单的查询就是不带任何参数的 find 方法。你可以直接用 toArray 方法将结果转换为数组，传递给接收错误信息和文档数组的回调函数作为参数。示例 10-2 代码描述了上述功能：

示例 10-2　插入四条文档数据然后用 find 方法获取该数据

```
var mongodb = require('mongodb');

var server = new mongodb.Server('localhost', 27107, {auto_reconnect:true});
var db = new mongodb.Db('exampleDb', Server);

//打开数据库连接
db.open(function(err,db)) {
  if(!err) {

     //访问或者创建widgets 文档
     db.collection('widgets', function(err, collection) {

        //删除所有 widgets 文档
        collection.remove(null, {safe:true}, function(err, result) {
           if (!err) {
             //创建四条记录
             var widget1 = {id:1, title:'First Great widget',
                           desc:'greatest widget of all',
                           price:14.99, type:'A'};
             var widget2 = {id:2, title:'Second Great widget',
                           desc:'second greatest widget of all',
                           price:29.99, type:'A'};
             var widget3 = {id:3, title:'third widget', desc:'third widget',
                           price:45.00, type:'B'};
             var widget4 = {id:4, title:'fourth widget', desc:'fourth widget',
                           price:60.00, type:'B'};

             collection.insert([widget1,widget2,widget3, widget4],
{safe:false},
                                                function(err,result) {
                if(err) {
                   console.log(err);
                }else {

                   //返回所有文档
                   collection.find().toArray(function(err,docs) {
                      console.log(docs);

                      //关闭数据库连接
                      db.close();
                   });
                }
             });
           }
        });
     });
  }
});
```

输出到控制台的查询结果会显示四个新添加的文档，以及系统生成的 id：

```
[ { id:1,
```

```
    title:'First Great widget',
    desc:'greatest widget of all',
    price:14.99,
    type:'A',
    _id:4fc109ab0481b9f652000001},
  { id:2,
    title:'Second Great widget',
    desc:'second greatest widget of all',
    price: 29.99,
    type:'A',
    _id:4fc109ab0481b9f652000002},
  { id:3,
    title:'third widget',
    desc:'third widget',
    price:45,
    type:'B',
    _id:4fc109ab0481b9f652000003},
  { id:4,
    title:'fourth widget',
    desc:'fourth widget',
    price:60,
    type:'B',
    _id:4fc109ab0481b9f652000004 } ]
```

如果不希望返回所有内容，我们可以提供选择器。在以下代码中，我们查询所有具有 type 为 A 的数据，返回除 type 之外的所有 field：

```
//返回所有文档
collection.find({type:'A'}, {fields:{type:0}}).toArray(function (err, docs) {
   if(err) {
     console.log(err);
   } else {
     console.log(docs);

     //关闭数据库连接
     db.close();
   }
});
```

查询结果为：

```
[ { id:1,
    title:'First Great widget',
    desc:'greatest widget of all',
    price:14.99,
    _id:4f7ba035c4d2204c49000001 },
  { id:2,
    title:'Second Great widget',
    desc:'second greatest widget of all',
    price:29.99,
    _id:4f7ba035c4d2204c49000002 } ]
```

我们也可以使用 findOne 访问一条数据。这样查询的结果并不需要被转换为数组，可以直接访问。以下代码查询了 ID 为 1 的文档，只返回 title：

```
//返回一个文档
collection.findOne({id:1}, {fields:{title:1}}, function (err, doc) {
  if (err) {
     console.log(err);
  } else {
      console.log(doc);

      //关闭数据库连接
      db.close();
  }
});
```

查询结果为：

```
{ title: 'First Great widget', _id: 4f7ba0fcbfede06649000001 }
```

系统生成的唯一标识符总会出现在查询结果中。

即使我修改了查询语句返回所有 type 为 A 的数据（有两条），collection.findOne 方法也只会返回一个。在选项参数中改变 limit 的值没有任何变化：查询成功的情况下永远只返回一条结果。

10.1.5 使用 Updates、Upserts、Find 和 Remove

MongoDB Native Node.js Driver 支持几种修改或者删除文档的方法，或者同时进行以上两个操作。

update

更新或者 upserts（如果不存在就添加）文档。

remove

删除文档。

findAndModify

查找并修改或者删除一个文档（返回被修改或者删除的文档）。

findAndRemove

查找并删除一个文档（返回被删除的文档）。

update/remove 和 findAndModify/findAndRemove 之间最本质的区别就在于后者的两个方法都返回了被操作的文档。

这些方法的使用与之前看到的 insert 没有太大区别。你需要创建一个数据库连接，找到你想要的那个文档索引，然后进行操作。

如果 MongoDB 现在有以下数据：

```
{id: 4,
  title: 'fourth widget',
  desc: 'fourth widget'.
  price: 60.00,
  type: '8'}
```

当你想要修改 title 的时候可以使用 update 方法，如示例 10-3 所示。你可以提供所有的 field，MongoDB 会对文档进行全部替换。但是更好的方法是使用 MongoDB 修改方法，比如$set。$set 修改符号告诉数据库只修改作为属性传递给修改器的 field。

示例 10-3　更新一个 MongoDB 文档

```
var mongodb = require('mongodb');
var server = new mongodb.Server('localhost', 27017, {auto_reconnect:true});
var db = new mongodb.Db('exampleDb', server);

// 打开数据库连接
db.open(function (err, db) {
  if (!err) {
     // 访问或者创建 widgets collection
     db.collection('widgets', function (err, collection) {

        //更新
        collection.update({id:4},
          {$set:{title:'Super Bad Widget'}},
             {safe:true}, function (err, result) {
          if (err) {
             console.log(err);
          } else {
             console.log(result);
             // 查询更新的文档
             collection.findOne({id:4}, function (err, doc) {
                if (!err) {
                    console.log(doc);

                    //关闭数据库
                    db.close();
                }
             });
          }
        });
```

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

结果返回的文档显示了修改的区域。

提示

$set 中可以添加多个 field。

还有其他一些修改符号用于原子数据更新：

$inc

对某一个 field 的值增加指定数值。

$set

如例子所示，设置某个 field。

$unset

删除一个 field。

$push

如果 field 是一个数组，给数组中添加一个数值（如果不是数组则转换为数组）。

$pushAll

向数组中添加几个值。

$addToSet

只有 field 为数组时添加值到该数组。

$pull

从数组中删除一个元素。

$pullAll

一次性从数组中删除几个元素。

$rename

重命名一个 field。

$bit

按位操作。

我们为什么不是直接删除文档然后插入一条新的数据而是选择修改呢？原因在于，尽管我们需要提供所有用户定义的 field，但不包括系统生成的唯一标识符。在更新过程中该标识符不变。如果在另一个文档中这个标识符被当做某个 field 存储，比如父文档，删除该 id 的文档会使父文档中的该元素失去依赖。

提示

我在本章不涉及关于树（复杂的父子数据结构）的概念，MongoDB 官网有关于这一结构的文档。

更重要的一点是，修改方法确保了操作发生在正确的位置，一定程度上保证一个用户的更新不会覆盖另一个的。

在例子中没有使用 update 的参数，update 方法接收四个参数：

- safe，用于安全更新；
- upsert，布尔值，设置为 true 表示如果文档不存在则执行 insert 操作（默认为 false）；
- multi，布尔值，设置为 true 表示所有符合选择条件的文档都要被更新；
- serializFunction 使文档中所有方法序列化。

示如果你不确定数据库是否包含某个文档，设置 upsert 选项为 true。

示例 10-3 在修改的结果上做了一次查找操作以确保该文档确实被修改了。另一种较好的方法是使用 findAndModify。该方法参数与 update 基本一致，添加了数组作为第二个参数。如果返回多个文档查询结果，update 按顺序执行。

```
//更新
collection.findAndModify({id:4}, [[ti]],
  {$set:{title:'Super Widget', desc:'A really great widget'}},
     {new:true}, function (err, doc) {
   if (err) {
     console.log(err);
  } else {
     console.log(doc);DB
  }
  db.close();
  });
```

还可以使用 remove 选项利用 findAndModify 方法删除某个文档，不过这样的话回调函数中没有任何返回结果。还可以使用 remove 和 findAndRemove 方法进行删除。之前的例子在插入数据前直接使用 remove，不设置选择器，删除了全部文档。提供选择器可以删除指定的某个文档：

```
collection.remove({id:4},
     {safe:true}, function (err, result) {
   if (err) {
      console.log(err);
   } else {
      console.log(result);
   }
```

result 为删除的文档数目（本例中为 1）。查看被删除的文档使用 findAndRemove：

```
collection.findAndRemove({id:3}, [['id', 1]],
     function (err, doc) {
   if (err) {
      console.log(err);
   } else {
      console.log(doc);
   }
});
```

到这里我们已经介绍了从 Natibe driver Node 应用可以完成的基本的 CRUD 操作（creae、read、update、delete），还有其他更多的功能，比如固定集合、索引、其他一些 MongoDB 修改符、分区（跨机器分区数据）等。Native driver 的文档对此有完善的例子进行说明。

这些例子中有一些关于异步环境下的数据访问的问题，会下面的“异步数据访问的挑战”说明栏中详细讨论。

异步数据访问的挑战

异步开发和数据访问的一个挑战是为了确保一个操作在下一个操作开始前完成所做的嵌套的层次问题。在最后几节中，你会看到只是一些简单的操作回调函数很快地成为嵌套关系，比如访问 MongoDB，获取 collection，进行某种操作以及验证操作是否发生。

MongoDB Native Node.js Driver 文档包含一些实例，开发人员使用 timer（定时器）来确保下一个方法执行前上一个方法已经结束。你不会想要使用这种实现方式的。为了避免回调函数深层次嵌套带来的问题，你可以使用命名函数，或者某些异步模块，比如 Step 和 Async。

事实上最好的实现方式是确保更新数据库的每个方法中都做了只实现最小的功能块。如果你努力避免嵌套回调函数，程序又很难切换使用一些类似 Async 的模块，可能的原因在于每个方法中所做的操作太多了。这种情况下，你需要将复杂的对数据库多操作的函数分解为可控制的单元。

评价异步编程最重要的一点就是简单。

10.2 使用 Mongoose 实现 Widget 模块

MongoDB Native Node.js Driver 提供了对 MongoDB 数据库的绑定，但是没有更高级别的抽象。所以你需要使用类似 Mongoose 的 ODM（object-document mapping，对象文档映射）。

提示

Mongoose 官网：http://mongoosejs.com/。

安装 Mongoose：

```
npm install mongoose
```

不同于之前的直接对 MongoDB 发出命令进行操作，首先使用 Mongoose Schema 定义对象，然后用 Mongoose model 对象同步数据库：

```
var Widget =  new Schema({
  sn: {type: String, require: true, trim:true,unique:true},
  name: {type:String, required: true, trim:true},
  desc: String,
  price: Number
});

var widget = mongoose.model('Widget', Widget);
```

当我们定义对象时，我们提供了之后对文档中 field 的控制。在上段代码中，我们定义了一个 Widget 对象，有三个类型为 String，一个类型为 Number 的 field。sn 和 name 字段都是必需的 trim（删去首尾空格），并且 sn 字段在该文档中必须是唯一的。

Collection 在此时还没有被创建，直到有一个文档被创建之后才会有 Collection。创建出来的 collection 被命名为 widgets——widget，对象名是小写并且复数形式。

任何时候访问该 collection，用同样的方法调用 mongoose.model。

上段代码是讲添加 widget 程序 MVC 实现的最后一个部分的第一步。接下来的几节中，我们会完成从内存数据存储到 MongoDB 的转换。不过首先我们要对 Widget 程序做一些重构。

10.3 重构 Widget 工厂

重构是指对现有代码结构的调整过程，在基本不影响用户接口的前提下使代码和结构变得清晰简单。既然我们现在正在对 widget 程序使用 MongoDB 做修改，刚好可以看一下还有哪些修改可以做。

现在 widget 程序的文件系统结构为：

```
/application 目录
   /routes -根目录 controller
   /controllers - 对象的 controllers
   /public- 静态文件
      /widgets
   /views- 模板文件
      /widgets
```

routes 子目录提供了高级别（非对象层面）的功能。命名并不表意，所以重命名为 main。这导致对 app.js 文件做出一些小的修改：

```
//高级别
app.get('/', main.index);

app.get('/stats', main.stats);
```

接下来，我添加了一个 model 子目录。像 controller 的代码放在 controllers 子目录中一样，MongoDB model 的定义存储在这个 model 子目录中。现在目录结构是：

```
/application 目录
   /main- 根目录 controller
   /controllers - 对象的 controllers
   /public - 静态文件
      /widgets
   /views - 模板文件
      /widgets
```

下一个需要做的修改与数据结构有关系。目前程序的主键为 ID 字段，由系统生成但是用户可以通过路由系统访问。显示一个 widget 的 URL 如下所示：

```
http://localhost:3000/widgets/1
```

这是个一般化的做法。一个很流行的内容管理系统（CMD）Drupal，除非用户使用URL重定向模块，也使这种实现方式访问Drupal节点（故事，stories）和用户：

```
http://burningbird.net/node/78
```

问题在于MongoDB为每个对象生成了唯一标识符并使用了路由无法识别的形式。有一种变通的方案，要求创建另一个collection包含一个id，用于替换系统生成标示符，但是实现过程很丑陋，与MongoDB的底层结构相反，而且Mongoose也不会使用这种方式。

Widget对象中的title字段是唯一的，但是包含空格和字符，使它无法作为URL。所以，我们定义了一个新的字段 sn，作为产品的序列化数字。当创建一个新的widget对象的时候，用户可以任意制定一个系列号给该对象，之后我们可以用这个序列号访问该对象。例如，一个 widget 的序列号为 1A1A，可以通过下列 url 访问：

```
http://localhost:3000/widgets/1A1A
```

从程序角度来看，新的数据结构为：

```
sn:string
title:string
desc:string
price:number
```

这部分的修改会导致用户接口的一些修改，但是这些修改是值得的。Jade模板有很小的地方需要修改：只需要将id替换为sn，为表单添加一个放序列号的字段。

提示

为了避免重复大段代码来显示这些小的修改，我将例子都放在O'Reily中本书的目录页：http://oreilly.com/catalog/9781449323073。在第 12 章目录下你会找到所有最新的widget程序文件。

更重要一些的修改是对widget.js文件中controller的代码修改。该文件的修改，以及其他与MongoDB后台有关的内容将在下一节介绍。

10.4 添加MongoDB后台

第一个修改是添加MongoDB数据库的连接，添加在app.js文件中，这样对数据的连接存在于程序的整个生命周期。

首先，在该文件中引入 Mongoose：

```
var mongoose = require('mongoose');
```

建立数据库连接：

```
//MongoDB
mongoose.connect('mongodb://127.0.0.1/WidgetDB');

mongoose.connection.on('open', function() {
   console.log('Connected to Mongoose');
});
```

注意到 MongoDB 的 URI，指定的数据库被当做 URI 的最后一部分。

这个修改和之前提到的将 routes 转换为 main 的修改对 app.js 都是必需的修改。

下一个需要修改 maproutecontroller.js。路由中使用 id 的部分需要替换为 sn，修改后的路由如下段代码所示：

```
// show
app.get( prefix + '/:sn', prefixObj.show);

// edit
app.get( prefix + '/:sn/edit', prefixObj.edit);

// update
app.put( prefix + '/:sn', prefixObj.update);

//destroy
app.del( prefix + '/:sn', prefixObj.destroy);
```

如果不做这个修改，controller 代码期望得到 sn 作为参数，但实际得到的是 id。

下面需要添加一段代码而不是修改。在 models 子目录中，创建一个新的文件 widgets.js，定义为 widget model。为了使 model 在模块外可以访问，需要暴露给外部，如示例 10-4 所示。

示例 10-4　新的 widget model 定义

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema
     , ObjectId = Schema.ObjectId;

// 创建 Widget model
var Widget = new Schema({
  sn:{type:String, require:true, trim:true, unique:true},
  name:{type:String, required:true, trim:true},
```

```
  desc:String,
  price:Number
});

module.exports = mongoose.model('Widget', Widget);
```

最后需要做的是对 widget controller 代码的修改，我们需要使用 Mongoose，将内存中的数据存储到 MongoDB。尽管这部分修改对程序来说很重要，但是代码本身的修改并不难。只是一些小的修改，类似于将 id 换为 sn 之类的。示例 10-5 包含了 widget controller 代码的全部内容。

示例 10-5　修改后的 widget controller

```
var Widget = require('../models/widget.js');

// i /widgets/显示所有widgets
exports.index = function (req, res) {
   Widget.find({}, function (err, docs) {
      console.log(docs);
      res.render('widgets/index', {title:'Widgets', widgets:docs});
   });
};

// display new widget form
exports.new = function (req, res) {
    console.log(req.url);
    var filePath = require('path').normalize(__dirname +
                                    "/../public/widgets/new.html");
   res.sendfile(filePath);
};

// 添加 widget
exports.create = function (req, res) {

   var widget = {
     sn:req.body.widgetsn,
     name:req.body.widgetname,
     price:parseFloat(req.body.widgetprice),
     desc:req.body.widgetdesc};

   var widgetObj = new Widget(widget);

   widgetObj.save(function (err, data) {
     if (err) {
        res.send(err);
     } else {
        console.log(data);
        res.render('widgets/added', {title: 'Widget Added', widget: widget});
```

```
      }
   });
};
// 显示 widget
exports.show = function (req, res) {
   var sn = req.params.sn;
   Widget.findOne({sn:sn}, function (err, doc) {
      if (err)
         res.send('There is no widget with sn of ' + sn);
   else
      res.render('widgets/show', {title:'Show Widget', widget:doc});
   });
};

// 删除 widget
exports.destroy = function (req, res) {
   var sn = req.params.sn;

   Widget.remove({sn:sn}, function (err) {
      if (err) {
         res.send('There is no widget with sn of ' + sn);
      } else {
         console.log('deleted ' + sn);
         res.send('deleted ' + sn);
      }
   });
};

// 显示 edit form
exports.edit = function (req, res) {
   var sn = req.params.sn;
   Widget.findOne({sn:sn}, function (err, doc) {
      console.log(doc);
      if (err)
          res.send('There is no widget with sn of ' + sn);
      else
        res.render('widgets/edit', {title:'Edit Widget', widget:doc});
   });
};

// 更新 widget
exports.update = function (req, res) {
  var sn = req.params.sn;
  var widget = {
     sn:req.body.widgetsn,
     name:req.body.widgetname,
     price:parseFloat(req.body.widgetprice),
     desc:req.body.widgetdesc};

  Widget.update({sn:sn}, widget, function (err) {
```

```
        if (err)
          res.send('Problem occured with update' + err)
        else
         res.render('widgets/added', {title:'Widget Edited', widget:widget})
    });
  };
```

现在 widget 程序的数据都可以保存在数据库中，而不会随着程序关闭消失了。整个程序的建立在为对现有稳定的程序组件影响最小情况下添加新的数据支持。

提示

本章中的 widget 应用程序例子是在前一章的基础上完成的。这意味着除了 MongoDB 之外，你需要启动 Redis 服务器来使程序正常工作。

第 11 章

Node 与关系型数据库

在传统的 Web 开发中，关系型数据库是最流行的数据存储手段。Node 却没有遵循这个模式，或许因为 Node 应用程序本身的特性，也或许是因为很多开发人员并不愿意在 Node 中使用这些传统技术。相比关系型数据库来说，Node 对类似于 Redis 和 MongoDB 这样的数据存储系统的支持更多一些。

如果与 PHP 和 Python 等语言对关系型数据库的支持比较的话，Node 提供的关系型数据库支持模块并不是非常完整。在我看来，这些模块甚至还没有到达产品化的程度。

不过，从积极的一面看，这些支持关系型数据库的模块还是相当易用的。在这一章中，我会通过两种不同的方法来将关系型数据库 MySQL 整合到我们的 Node 应用程序中。一种方法是使用 mysql（node-mysql），它是一个较为流行的 JavaScriptMySQL 客户端；另一种方法是使用 db-mysql，它是 node-db（Node 尝试构建的一个通用数据库引擎）的一部分，并是使用 C++编写的。

这两个模块目前都不支持事务操作。但是，通过使用 mysql-series 模块，我们可以为 node-mysql 添加类似的功能，本章会有一个简单的演示。另外，我们还会对 Sequelize 做个简要介绍，它是一个基于 MySQL 数据库的 ORM（对象—关系映射）库。

关系型数据库有很多种，包括 SQL Server、Oracle 以及 SQLite 等。我选择使用 MySQL，是因为它能支持 Windows 和 Unix 环境，对于非商业用途它是免费的，并且它也是最普遍的被大多数应用程序所使用的数据库。另外，Node 关系型数据库模块对它的支持也是最多的。

在本章中，我们使用名为 nodetest2 的数据库进行所有测试，表结构如下：

```
id - int(11), primary key, not null, autoincrement
```

```
title - varchar(255), unique key, not null
text - text, nulls allowed
created - datetime, nulls allowed
```

11.1 db-mysql 入门

在使用 db-mysql 之前，我要先安装 MySQLclient 库。你可以在 *http://nodejsdb.org/db-mysql/*找到安装包和安装说明。

当配置好上述环境后，就可以使用 npm 来安装 DB-mysql 了：

```
npm install db-mysql
```

db-mysql 模块提供了两个类来支持与 MySQL 数据库的交互。第一个是 database 类，可以使用它来建立或断开数据库连接，查询操作也需要基于它来进行。第二个是 query 类，当我们对数据库对象做查询操作时就会用到。使用 query 类来创建并执行查询操作，有两种方式：一种是使用方法链，每个方法描述了该查询操作的一部分信息；另一种则是直接使用 SQL 查询字符串。可见 db-mysql 的使用是非常灵活的。

所有方法的最后一个参数都是一个回调函数，查询结果以及任何可能的错误信息都会传递给该回调函数。你可以使用嵌套回调或 EventEmitter 事件机制来将多个操作步骤衔接起来，以便处理错误信息或者对执行数据库命令所得到的结果进行处理。

当创建与 MySQL 数据库的连接时，你还可以使用一些可选参数。但必须提供的参数包括：主机名（hostname）或端口（port）或套接字（socket）、用户名（user）、密码（password）以及数据库名称（database）。

```
var db = new mysql.Database({
  hostname: 'localhost',
  user: 'username',
  password: 'userpass',
  database: 'databasenm'
});
```

可选参数的细节信息可以参考 db-mysql 文档以及 MySQL 文档。

11.1.1 查询字符串和方法链

为了演示 db-mysql 的灵活性，示例 11-1 应用程序在连接到数据库后，执行了两次相同的查询操作：第一次使用 query 类的方法链，第二次使用查询字符串。第一个

查询操作使用了一个嵌套回调函数来处理查询结果，而第二个则监听 succes 和 error 事件并做出响应。无论如何，在这两种情况下返回的查询结果都是 rows 对象，该对象包含了一个可以描述查询结果中每行数据的对象数组。

示例 11-1　使用两种不同的查询方式展示 db-mysql 的灵活性

```
var mysql = require('db-mysql');

// define database connection
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
    password: 'userpass',
    database: 'databasenm'
});

// connect
db.connect();

db.on('error', function(error) {
  console.log("CONNECTION ERROR: " + error);
});

// database connected
db.on('ready', function(server) {

  // query using chained methods and nested callback
  this.query()
      .select('*')
      .from('nodetest2')
      .where('id = 1')
      .execute(function(error, rows, columns) {
        if (error) {
            return console.log('ERROR: ' + error);
        }
        console.log(rows);
        console.log(columns);
  });

  // query using direct query string and event
  var qry = this.query();

  qry.execute('select * from nodetest2 where id = 1');

  qry.on('success', function(rows, columns) {
    console.log(rows); // print out returned rows
    console.log(columns); // print out returns columns
  });
  qry.on('error', function(error) {
    console.log('Error: ' + error);
  });
});
```

当与数据库的连接被成功建立时，database 对象会触发一个 ready 事件；如果发生

错误，则触发一个 error 事件。ready 事件的回调函数参数中包含了一个 server 对象，该对象具有以下属性：

hostname

数据库主机名。

user

建立数据库连接所使用的用户名。

database

连接到的数据库。

version

服务器软件版本。

在示例程序的第一次查询操作中，我们使用了函数链来构建查询操作。下面列出了你可以使用的 SQL 查询链方法：

select

包含查询的选择标准（如列名清单或使用星号*表示所有列）或选择字符串。

from

包含一组数据库表名或 from 声明所包含的字符串。

join

join 子句包含一个可选对象，可以用来指定该操作的具体信息，被用来连接的表名，表的别名（如有），连接条件（如有），以及是否转义表名及其别名（默认为 true）。

where

条件语句，其中可能包含占位符和用 and 或 or 条件连接起来的方法。

order

追加一个 ORDER BY 子句。

limit

　　追加一个 LIMIT 子句。

add

　　添加一个通用子句，例如 UNION。

方法链提供了更加通用的方法来为不同的数据库执行相同的 SQL 语句操作。目前，Node.js 的数据库驱动程序支持 MySQL（db-mysql）和 Drizzle（db-drizzle）。方法链通过提供统一的方法调用接口，屏蔽了两者的不同。方法链还可以自动处理并转义 SQL 语句中的数据，使得查询操作符合数据库的安全要求。否则，如果直接使用查询语句，你就必须使用 query.escape 方法来正确地转义 SQL 语句。

如果查询操作成功，query 对象会触发一个 success 事件，如果失败则触发一个 error 事件。另外在得到每一行数据时，each 事件都会被触发。当执行一次查询操作后，对应于 success 事件的回调函数参数中会包含 rows 对象以及 columns 对象。rows 对象包含了多个 row 对象，而每一个 row 对象也是一个数组，每个数组元素包含一个名/值对（name/value pairs）对象。colums 对象则描述了查询结果的列信息，每个 column 对象包含了列名和数据类型。如果本例中的测试表格包含 id、title、text 和 created 列的话，rows 对象的内容看起来会像：

```
{ id: 1,
    title: 'this is a nice title',
    text: 'this is a nice text',
    created: Mon, 16 Aug 2010 09:00:23 GMT }
```

而列对象看起来像这样：

```
[ { name: 'id', type: 2 },
  { name: 'title', type: 0 },
  { name: 'text', type: 1 },
  { name: 'created', type: 6 } ]
```

如果是执行更新、删除或者插入等查询操作后的 success 事件，对应的回调函数会使用 result 对象作为其参数。关于该对象的更多的细节将在下一节中做介绍。

虽然可以使用不同的方法来处理查询操作，但它们都必须在数据库连接成功建立后进行。由于 db-mysql 是 Node 提供的一项功能，因此相关的方法都是异步的。如果你试图在数据库连接回调函数之外做 query 操作，是不会成功的，因为那时数据库连接还未建立起来。

11.1.2 使用查询字符串更新数据库

如前所述，db-mysql 模块提供了两种不同的方式来更新关系型数据库中的数据：使用查询字符串，或使用方法链。现在我们来看看查询字符串的使用。

当使用查询字符串时，你可以直接使用 MySQL 客户端所支持的 SQL 语句：

```
qry.execute('update nodetest2 set title = "This is a better title" where id = 1');
```

或者你也可以使用占位符：

```
qry.execute('update nodetest2 set title = ? where id = ?', ["This was a better title", 1]);
```

占位符可以在查询字符串或方法链中使用。占位符是一种提前创建查询语句的方式，你可以在使用时再用任何你需要的值替代占位符。你可以在字符串中使用问号符（?）来表示占位符，然后在使用该字符串的方法调用中以数组的方式为多个占位符赋值，而这个数组一般作为该方法的第二个参数传入。

在数据库上执行操作后的结果会作为 success 事件回调函数的参数。在示例 11-2 中，我们为 test 数据库中插入了一行新数据。请注意，程序使用了 MySQL 的 NOW 函数来将 created 字段的值设置为当前时间。当使用一个 MySQL 函数时，你需要把它直接写进查询字符串，而不能使用占位符。

示例 11-2 在查询字符串中使用占位符

```
var mysql = require('db-mysql');

// define database connection
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
    password: 'userpass',
    database: 'databasenm'
});

// connect
db.connect();

db.on('error', function(error) {
   console.log("CONNECTION ERROR: " + error);
});

// database connected
db.on('ready', function(server) {

  // query using direct query string and event
  var qry = this.query();
```

```
  qry.execute('insert into nodetest2 (title, text, created) values(?,?,NOW())',
              ['Third Entry','Third entry in series']);

  qry.on('success', function(result) {
    console.log(result);
  });

  qry.on('error', function(error) {
    console.log('Error: ' + error);
  });
});
```

如果操作成功，操作结果将以参数的形式传递给回调函数并包含以下内容：

```
{ id: 3, affected: 1, warning: 0 }
```

id 是生成的行标识符，affected 属性表示受影响的行数（1），以及 warning 显示本次查询操作所生成的警告信息数量（这里为 0）。

与上述创建新数据条目的做法一样，当需要更新或删除数据信息时也可以直接使用 SQL 语句，或者使用占位符。在示例 11-3 实现的程序中，我们为 test 数据库添加了一条新记录，然后更新标题字段，最后删除该记录。你会注意到，不同的操作使用了不同的 query 对象，不过你也可以为同一个 query 对象传递不同的参数并多次运行它。在 insert 语句中我使用了 4 个占位符，如果只为其提供两个参数，那么程序就会报错。另外，该示例程序仍然使用的是嵌套回调，而没有使用事件捕获。

示例 11-3　使用嵌套回调实现插入，更新和删除记录

```
var mysql = require('db-mysql');

// define database connection
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
    password: 'password',
    database: 'databasenm'
});

// connect
db.connect();

db.on('error', function(error) {
   console.log("CONNECTION ERROR: " + error);
});

// database connected
db.on('ready', function(server) {

  // query using direct query string and nested callbacks
```

```
    var qry = this.query();

    qry.execute('insert into nodetest2 (title, text,created) values(?,?,NOW())',
                ['Fourth Entry','Fourth entry in series'], function(err,result) {
        if (err) {
          console.log(err);
        } else {
           console.log(result);

           var qry2 = db.query();
           qry2.execute('update nodetest2 set title = ? where id = ?',
                       ['Better title',4], function(err,result) {
              if(err) {
                 console.log(err);
              } else {
                 console.log(result);
                 var qry3 = db.query();
                 qry3.execute('delete from nodetest2 where id = ?',[4],
                       function(err, result) {
                    if(err) {
                       console.log(err);
                    } else {
                       console.log(result);
                    }
                 });
              }
           });
        }
     });
  });
```

在本例你可能会注意到，如果任何地方发生错误，我们是没有办法回滚已经执行的 SQL 语句的。目前来说，db-mysql 还不支持事务管理功能，如果需要确保数据库的一致性，你必须自己在应用程序中实现它。你可以在每个 SQL 语句执行之后的错误检查代码中对之前成功执行的 SQL 语句进行回滚操作。但这并不是一个理想的解决方法，而且还必须谨慎使用自动递增字段。

提示

在另一个模块——mysql-queues 中实现了一种类似事务的功能，我将在本章节的后续部分介绍它。

11.1.3 使用方法链更新数据库

db-mysql 为数据库的插入、更新和删除操作提供了独立的方法，包括：insert、update 以及 delete。update 和 delete 方法可以结合 where 方法使用，而 where 方法可以使用 and 和 or 方法并按顺序处理它们。update 方法还可以使用另一个 set 方法来为 SQL 指定需要更新的值。

示例 11-4 完成了与示例 11-3 相同的功能，但它使用了方法链来实现插入和更新记录。它没有使用 delete 方法，因为在我写这本书时，delete 方法还不能正常工作。

示例 11-4　使用方法链插入一个新的记录，然后更新它

```
var mysql = require('db-mysql');

// define database connection
var db = new mysql.Database({
    hostname: 'localhost',
    user: 'username',
    password: 'password',
    database: 'databasenm'
});

// connect
db.connect();

db.on('error', function(error) {
   console.log("CONNECTION ERROR: " + error);
});

// database connected
db.on('ready', function(server) {
  // query using direct query string and nested callbacks
  var qry = this.query();
  qry.insert('nodetest2',['title','text','created'],
            ['Fourth Entry', 'Fourth entry in series', 'NOW()'])
     .execute(function(err,result) {
       if (err) {
        console.log(err);
       } else {
         console.log(result);

         var qry2 = db.query();
         qry2.update('nodetest2')
             .set({title: 'Better title'})
             .where('id = ?',[4])
             .execute(function(err, result) {
            if(err) {
               console.log(err);
            } else {
               console.log(result);
            }
         });
      }
    });
});
```

当你需要从应用程序中获取信息然后进行 SQL 查询操作，或者当你的应用程序需要支持多种数据库时，使用方法链的确会很方便，但是，我个人并不喜欢使用方法链。

11.2 使用 node-mysql 实现本地 MySQL 访问

与 db-mysql 不同，你不需要安装 MySQL 客户端软件就可以使用它。你只需要安装该模块就行：

```
npm install mysql
```

它的使用也很简单。创建一个客户端，然后连接到 MySQL 数据库中，选择要使用的数据库，再通过 query 方法完成所有的数据库操作。query 方法的最后一个参数是一个回调函数，通过它能够取得操作的处理结果。如果不使用回调函数，还可以通过监听事件的方式来确定处理过程何时完成。

11.2.1 使用 node-mysql 做基本的 CRUD 操作

正如刚才所说的，使用 node-mysql API 是非常简单的：创建客户端，设置数据库，并通过客户端发送 SQL 查询语句。回调函数是可选的，并且有一些基本的事件支持。当你使用回调函数时，回调函数的参数通常包含 error 和 result，当使用 SELECT 查询语句时，回调函数中还会包含一个 fields 参数。

示例 11-5 演示如何使用 node-mysql 连接数据库，创建一条新记录并更新它，最后再删除它。这是个很简单的例子，但却能描述 node-mysql 支持的所有功能。

示例 11-5　使用 node-mysql 实现 CRUD 操作

```
var mysql = require('mysql');

var client = mysql.createClient({
   user: 'username',
   password: 'password'
  });

client.query('USE databasenm');

// create
client.query('INSERT INTO nodetest2 ' +
   'SET title = ?, text = ?, created = NOW()',
   ['A seventh item', 'This is a seventh item'], function(err, result) {
   if (err) {
     console.log(err);
   } else {
     var id = result.insertId;
     console.log(result.insertId);

     // update
     client.query('UPDATE nodetest2 SET ' +
         'title = ? WHERE ID = ?', ['New title', id], function (err, result) {
        if (err) {
```

```
          console.log(err);
        } else {
          console.log(result.affectedRows);

          // delete
          client.query('DELETE FROM nodetest2 WHERE id = ?',
            [id], function(err, result) {
            if(err) {
               console.log(err);
            } else {

               console.log(result.affectedRows);

               // named function rather than nested callback
               getData();
            }
          });
        }
     });
   }
});

// retrieve data
function getData() {
   client.query('SELECT * FROM nodetest2 ORDER BY id', function(err, result,fields) {
     if(err) {
       console.log(err);
     } else {
       console.log(result);
       console.log(fields);
     }
     client.end();
   });
}
```

正如我们所期望的，查询结果被保存到了一个对象数组中，每个数组元素表示一行数据。下面是该对象数组的内容示例（只显示了第一行数据的内容）：

```
[ { id: 1,
    title: 'This was a better title',
    text: 'this is a nice text',
    created: Mon, 16 Aug 2010 15:00:23 GMT },
  ... ]
```

另外，回调函数同时还包含了 fields 参数，不过它描述信息的格式与其他模块不大相同。我们得到的不是一个对象数组，而是一个对象。该对象的属性名称和数据库表格的字段一一对应，而属性值则是一个描述了字段信息的对象。由于整个对象的信息量比较大，因此这里我只展示有关第一个字段 id 的内容：

```
{ id:
   { length: 53,
     received: 53,
```

```
number: 2,
type: 4,
catalog: 'def',
db: 'nodetest2',
table: 'nodetest2',
originalTable: 'nodetest2',
name: 'id',
 originalName: 'id',
charsetNumber: 63,
fieldLength: 11,
fieldType: 3,
flags: 16899,
decimals: 0 }, ...
```

该模块不支持 SQL 语句嵌套，它也不支持事务。接下来我们会讨论 mysql-queues，它是唯一能提供类似事务支持的库。

11.2.2 MySQL 事务与 mysql-queues

mysql-queues 模块对 node-mysql 进行了包装，并对多重查询以及数据库事务提供了支持。当使用它时，可能会觉得有点奇怪，看上去它似乎并没有提供对异步的支持，但实际并非如此。

通常情况下，为了确保完成异步功能，你可以使用嵌套回调、命名函数，或类似于 Async 这样的模块。在示例 11-6 中，mysql-queues 控制执行流程，确保队列中的所有 SQL 语句都能在最后 SELECT 被处理前执行完成。这些 SQL 语句按照如下顺序依次完成处理：insert，update，最后是 retrieve。

示例 11-6 使用队列管理多个 SQL 语句的执行

```
var mysql = require('mysql');
var queues = require('mysql-queues');

// connect to database
var client = mysql.createClient({
   user: 'username',
   password: 'password'
   });

client.query('USE databasenm');

//associated queues with query
// using debug
queues(client, true);

// create queue
q = client.createQueue();

// do insert
```

```
q.query('INSERT INTO nodetest2 (title, text, created) ' +
         'values(?,?,NOW())',
         ['Title for 8', 'Text for 8']);

// update
q.query('UPDATE nodetest2 SET title = ? WHERE title = ?',
           ['New Title for 8','Title for 8']);

q.execute();

// select won't work until previous queries finished
client.query('SELECT * FROM nodetest2 ORDER BY ID', function(err, result, fields) {
    if (err) {
      console.log(err);
    } else {

      // should show all records, including newest
      console.log(result);
      client.end();
    }
});
```

如果需要实现事务功能，那么你需要启动一个事务，而非队列（queue）。并且你需要在发生错误时调用 rollback 来执行回滚操作，当成功完成操作后执行 commit 来结束事务。同样的，当你调用了事务对象的 execute 方法后，之后所有的查询操作都会被放入队列等待执行直到事务被处理完毕。示例 11-7 与示例 11-6 实现的功能相同，但却使用了事务。

示例 11-7　使用事务实现更可控的 SQL 更新操作

```
var mysql = require('mysql');
var queues = require('mysql-queues');

// connect to database
var client = mysql.createClient({
   user: 'username',
   password: 'password'
   });

client.query('USE databasenm');

//associated queues with query
// using debug
queues(client, true);

// create transaction
var trans = client.startTransaction();
// do insert
trans.query('INSERT INTO nodetest2 (title, text, created) ' +
          'values(?,?,NOW())',
          ['Title for 8', 'Text for 8'], function(err,info) {

   if (err) {
```

```
            trans.rollback();
        } else {
          console.log(info);

          // update
          trans.query('UPDATE nodetest2 SET title = ? WHERE title = ?',
                 ['Better Title for 8','Title for 8'], function(err,info) {
             if(err) {
                trans.rollback();
             } else {
                console.log(info);
                trans.commit();
             }
           });
        }
    });
    trans.execute();

    // select won't work until transaction finished
    client.query('SELECT * FROM nodetest2 ORDER BY ID', function(err, result, fields) {
        if (err) {
          console.log(err);
        } else {

          // should show all records, including newest
          console.log(result);
          client.end();
        }
    });
```

mysql-queues 为 node-mysql 模块添加了两个重要的组件：

- 支持多重查询，而无需使用嵌套回调；
- 支持事务。

如果你打算使用 node-mysql 的话，我强烈建议你也使用 mysql-queues。

11.3 ORM 与 Sequelize

之前提到的所有模块都为 MySQL 数据库提供绑定支持，但它们没有再提供更高层次抽象的支持。而 Sequelize 模块实现了对 ORM 的支持，尽管它还不支持事务。

11.3.1 定义模型

要使用 Sequelize，首先需要定义模型。模型表示了数据库表和 JavaScript 对象之间的映射关系。在前面的示例程序中，我们用到了一个简单的表格 nodetest2，结构如下：

```
id - int(11), primary key, not null
title - varchar(255), unique key, not null
text - text, nulls allowed,
created - datetime, nulls allowed
```

我们需要为每个字段使用适当的数据库标志来创建这个数据库表对应的模型：

```
// define model
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize.INTEGER, primaryKey: true},
   title : {type: Sequelize. STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   created : Sequelize.DATE
   });
```

目前支持的数据类型映射：

- Sequelize.STRING =>VARCHAR(255)
- Sequelize.TEXT => TEXT
- Sequelize.INTEGER => INTEGER
- Sequelize.DATE => DATETIME
- Sequelize.FLOAT => FLOAT
- Sequelize.BOOLEAN =>TINYINT(1)

可以用来进一步细化字段的选项包括：

type

字段的数据类型。

allowNull

false 表示不允许为空，默认为 true。

unique

设置为 true 可以防止写入重复值，默认为 false。

primaryKey

true 表示设置为主键。

autoIncrement

true 表示为自动递增字段。

如果你正打算新建一个应用程序和对应的数据库的话，那么在完成模型定义后，你就可以直接通过运行 sync 方法来创建对应的数据库表格：

```
// sync
Nodetest2.sync().error(function(err) {
   console.log(err);
});
```

如果你这么做了，你会发现数据库表和模型会有一些不同，这是由于 Sequelize 修改了数据库表。首先，表的名称变为 nodetest2s 而非 nodetest2；其次，新增了两个表字段：

```
id - int(11), primary key, autoincrement
title - varchar(255), unique key, nulls not allowed
text - text, nulls allowed
created - datetime, nulls allowed
createdAt - datetime, nulls not allowed
updatedAt - datetime, nulls not allowed
```

由于没有办法阻止 Sequelize 对数据库表做此类修改，因此你要相应地调整你之前对处理结果的预期。作为开胃菜，先来删除 created 字段吧，因为你不再需要它了。步骤很简单，先在 Sequelize 模型中删除该字段，然后再次运行 sync 方法：

```
// define model
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize. INTEGER, primaryKey: true},
   title : {type: Sequelize.STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   });
// sync
Nodetest2.sync().error(function(err) {
   console.log(err);
});
```

现在，你已经拥有一个能够描述模型的 JavaScript 对象了，而且还建立了模型与数据库表的映射关系。接下来，我们会向数据库表中添加一些数据。

11.3.2 ORM 风格的 CRUD 实现

让我们继续看看使用 MySQL 数据库绑定和使用 ORM 之间的差异。使用 ORM

时，你无需为新数据做插入操作，而是要创建一个新的对象实例并保存它。对于更新操作，也是同样的道理：不再使用 SQL 语句来完成更新操作，而是直接修改对象属性，或者使用 updateAttributes 并传入需要修改的对象属性。当需要从数据库中删除一行数据时，你只需要访问对应的对象实例，然后调用 destroy 方法。

为了说明所有这些操作，我在示例 11-8 中实现了模型创建、数据库同步（创建表，如果它已经不存在的话）、创建新实例以及保存实例等操作。在示例程序中，我们首先创建了一个新实例，然后对它做了两次更新操作。而在新添加的对象被销毁前，我们还检索并输出了所有对象内容。

示例 11-8　使用 Sequelize 实现 CRUD 操作

```
var Sequelize = require('sequelize');

var sequelize = new Sequelize('databasenm',
                'username', 'password',
                { logging: false});

// define model
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize.INTEGER, primaryKey: true},
   title : {type: Sequelize.STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   });

// sync
Nodetest2.sync().error(function(err) {
   console.log(err);
});

var test = Nodetest2.build(
   { title: 'New object',
     text: 'Newest object in the data store'});
// save record
test.save().success(function() {

  // first update
  Nodetest2.find({where : {title: 'New object'}}).success(function(test) {
     test.title = 'New object title';
     test.save().error(function(err) {
       console.log(err);
     });
     test.save().success(function() {

        // second update
        Nodetest2.find(
            {where : {title: 'New object title'}}).success(function(test) {
           test.updateAttributes(
           {title: 'An even better title'}).success(function() {});
           test.save().success(function() {
```

```
                // find all
                Nodetest2.findAll().success(function(tests) {
                   console.log(tests);


                  // find new object and destroy
                  Nodetest2.find({ where: {title: 'An even better title'}}).
                     success(function(test) {
                     test.destroy().on('success', function(info) {
                       console.log(info);
                       });
                  });
                });
             });
           })
         });
    });
  });
```

在输出 findAll 得到的查询结果后，对于其所包含的信息量你可能会感到惊讶。其实，你也可以直接访问查询结果中的属性值，首先访问数组元素，然后取得属性值：

```
tests[0].id; // returns identifier
```

查询结果所包含的其他数据信息则完全说明了 ORM 与关系型数据库绑定的区别。下面是查询结果的内容示例：

```
[ { attributes: [ 'id', 'title', 'text', 'createdAt', 'updatedAt' ],
    validators: {},
    __factory:
     { options: [Object],
       name: 'nodetest2',
       tableName: 'nodetest2s',
       rawAttributes: [Object],
       daoFactoryManager: [Object],
       associations: {},
       validate: {},
       autoIncrementField: 'id' },
    __options:
     { underscored: false,
       hasPrimaryKeys: false,
       timestamps: true,
       paranoid: false,
       instanceMethods: {},
       classMethods: {},
       validate: {},
       freezeTableName: false,
       id: 'INTEGER NOT NULL auto_increment PRIMARY KEY',
       title: 'VARCHAR(255) NOT NULL UNIQUE',
       text: 'TEXT',
    createdAt: 'DATETIME NOT NULL',
    updatedAt: 'DATETIME NOT NULL' },
```

```
  id: 14,
  title: 'A second object',
  text: 'second',
  createdAt: Sun, 08 Apr 2012 20:58:54 GMT,
  updatedAt: Sun, 08 Apr 2012 20:58:54 GMT,
  isNewRecord: false },...
```

11.3.3 添加多个对象

Sequelize 的异步特性在示例 11-8 中有明显的体现。当你不需要一次性增加多条数据记录时，使用嵌套回调没有问题。否则，使用嵌套回调则会碰到问题。

幸运的是，Sequelize 提供了一种简单的方式来链接多个操作，这样你就可以做一些事情，比如创建多个对象实例，然后一次性将它们全部保存起来。该模块提供了一个名为 chainer 的 helper，你可以按顺序为 chainer 添加多个 EventEmitter 任务（比如查询操作），不过在你调用 run 之前，所有任务都不会被执行。而调用 run 后所有操作的处理结果都会被返回，无论操作是成功还是失败。

示例 11-9 演示了如何使用 chainerhelper，该示例程序首先创建了三个对象实例并保存它们，然后运行了 findAll 来验证这些实例是否被成功保存。

示例 11-9　使用 chainer 简化多个对象实例的添加操作

```
var Sequelize = require('sequelize');

var sequelize = new Sequelize('databasenm',
                'username', 'password',
                { logging: false});

// define model
var Nodetest2 = sequelize.define('nodetest2',
  {id : {type: Sequelize.INTEGER, primaryKey: true},
   title : {type: Sequelize.STRING, allowNull: false, unique: true},
   text : Sequelize.TEXT,
   });

// sync
Nodetest2.sync().error(function(err) {
   console.log(err);
});
var chainer = new Sequelize.Utils.QueryChainer;
chainer.add(Nodetest2.create({title: 'A second object',text: 'second'}))
       .add(Nodetest2.create({title: 'A third object', text: 'third'}));

chainer.run()
      .error(function(errors) {
         console.log(errors);
       })
```

```
.success(function() {
   Nodetest2.findAll().success(function(tests) {
     console.log(tests);
   });
});
```

相比之前的代码，这段代码更简单，也更容易阅读。而且这种方法更适合包含有用户界面或使用了 MVC 的应用程序。

Sequelize 模块的文档网站上有更多相关介绍，包括如何处理相关联的对象（数据库表之间的关联）。

11.3.4 从关系型到 ORM

当使用 ORM 时，你需要牢记它有关数据结构的若干假设。第一个假设是：如果模型对象的名字为 Widget，那么对应的数据库表名为 widgets。另一个假设是：数据库表的每一行数据中都包含了该行数据的添加及更新时间。然而，当需要将一个现存的，已经使用了绑定技术的数据库系统转换为使用 ORM 时，这两个假设可能就会成为障碍（大多 ORM 实现都存在这种问题）。

使用 Sequelize 的真正问题在于它会复数化表名，不管你愿不愿意。因此，如果你为某个表定义模型时，Sequelize 会期望将复数化后的模型名称作为数据库表名。即使你指定一个表名，Sequelize 也要将它复数化。当你使用一个干净的数据库时，这种操作没有任何问题，它会在调用 sync 的时候自动为你创建这些表。但是当你使用一个现有的关系数据库时，这将会是一个问题。因此，如果你不是在构建一个全新系统的话，我强烈建议你不要使用 Sequelize 模块。

第 12 章

图形和 HTML5 Video

Node 提供了很多使用不同图形程序和程序库的机会。作为服务器端技术，你的 Node 程序可以使用任何基于服务器的图形软件，比如 ImageMagick 或者 GD。并且因为 Node 也是基于与驱动 Chrome 浏览器相同的 JavaScript 引擎，你还可以使用客户端的图形程序，比如 Canvas、WebGL 等等。

同时，Node 对现代浏览器支持的 HTML5 新的媒体元素 audio 和 video 文件提供了一定支持。虽然在直接使用 video 和 audio 上有所限制，但是在之前的章节中了解到程序可以提供这两种类型的文件。我们也可以使用一些基于服务器的技术，比如 FFmpeg。

没有任何一个关于 Web 图形的章节不提到 PDF 的。对在网站上使用 PDF 文档的用户来说有个好消息，我们可以使用一个非常好的 Node 模块用于生成 PDF，服务器上还安装了很多非常有用的 PDF 工具和库。

我不会详尽列举 Node 中每一个图形或者媒体实现和管理功能的每种形式。一个原因是我并不熟悉所有的方式，另一个原因是有些支持并不完整，或者该技术需要特别多资源。所以，我会关注在对 Node 程序更有意义更稳定的技术上：ImageMagick 基本的图片管理，HTML5 video，创建 PDF，用 Canvas 创建/流化图像。

12.1 创建和使用 PDF

操作系统、HTML 版本、开发技术，这些都会有所更改，但是没有变化的是无所不在的 PDF。不论你创建什么类型的程序、提供什么类型的服务，你都很有可能需要提供 PDF 文档服务。就像谁说的那句：PDF 很帅。

在 Node 程序中使用 PDF 有很多选择。一种实现方式是使用 Node 子进程访问操作系统工具，比如 PDFToolkit 或者 Linux 上的 wkhtmltopdf。另一种方式是使用模块，比如很流行的 PDFKit。或者两种混合使用。

12.1.1 使用子进程访问 PDF 工具

尽管 Windows 平台很少有命令行可以操作 PDF，但对于 Linux 和 OS X 来说还是有一些可用的命令。幸运的是，我使用过的两个——PDF Toolkit 和 wkhtmltopdf，都可以在以上三个环境中安装。

Wkhtmltopdf 对页面截屏

Wkhtmltopdf 使用 WebKit 渲染机制将 HTML 转换为 PDF 文件，是一种对网站或者图片截屏的便捷方式。一些网站提供了将页面内容生成 PDF 的功能，但是经常在生成的 PDF 中去掉所有图片。Wkhtmltopdf 工具可以维持页面所有的内容和样式。

Wkhtmltopdf 有 OS X 和 Windows 的安装版本，对于 Unix 环境你可以下载源代码自己 build。如果你在服务器上运行程序，你需要做些修改，因为它对 X Windows 的依赖。

为了在我的系统上（Ubuntu）使用 wkhtmltopdf，我需要安装支持这一功能的库：

```
apt-get install openssl build-essential xorg libssl-dev
```

然后我需要安装 xvfb 工具，允许 wkhtmltopdf 运行在虚拟的 X server 上（绕过对 X Windows 依赖）：

```
apt-get install xvfb
```

下一步，创建一个 shell 脚本——wkhtmltopdf.sh，将 wkhtmltopdf 包裹在 xvfb 中。代码行为：

```
xvfb-run -a -s "-screen 0 640×480×16 wkhtmltopdf $*
```

将这个 shell 脚本放在/usr/bin 目录下，并用 chmoda+x 改变文件权限。现在，可以通过 Node 程序访问 wkhtmltopdf。

Wkhtmltopdf 工具支持一系列选项，但是我只打算简单介绍从 Node 程序如何使用该工具。在命令行中，下面这行代码接收一个链接到远程网页的 URL 然后使用所有默认设置生成 PDF：

```
Wkhtmltopdf.sh http://remoteweb.com/page1.html page1.pdf
```

在 Node 中实现这一过程，需要使用子进程。作为扩展，程序需要接收输入的 URL 和输出文件。示例 12-1 显示了整个程序。

示例 12-1　简单的包含 wkhtmltopdf 的 Node 程序

```
var spawn = require('child_process').spawn;
// 命令行参数
var url = process.argv[2];
var output = process.argv[3];
if (url&& output) {
    var wkhtmltopdf = spawn('wkhtmltopdf.sh', [url, output]);
    wkhtmltopdf.stdout.setEncoding('utf8');
    wkhtmltopdf.stdout.on('data', function (data) {
        console.log(data);
    });
    wkhtmltopdf.stderr.on('data', function (data) {
        console.log('stderr: ' + data);
    });
    wkhtmltopdf.on('exit', function (code) {
        console.log('child process exited with code ' + code);
    });
} else {
    console.log('You need to provide a URL and output file name');
}
```

你一般不会在 Node 程序中单独使用 wkhtmltopdf,但是对于想要提供创建网页 PDF 功能的程序和网站来说这是种很便捷的方式。

使用 PDF Toolkit 访问 PDF 文件中的数据

PDF Toolkit 或者 pdfkt，提供了将分解 PDF 文档或者将多个 PDF 合并为一个的功能，同样还可以用于填写 PDF 表单，加水印，旋转 PDF 文档，压缩或者解压，修改 PDF。MAC 和 Windows 系统都有安装文件，大部分 Unix 系统只需要按照安装说明即可。

PDF Toolkit 可以通过 Node 的子进程访问。以下例子中的代码创建了一个子进程，调用 PDF Toolkit 的 dump_data 来确认 PDF 的信息，比如包含多少页：

```
   var spawn = require('child_process').spawn;
   var pdftk = spawn('pdftk', [__dirname + '/pdfs/datasheet-node.pdf', 'dump_
data']);
   pdftk.stout.on('data', function(data) {
      //将结果转换为对象
      var array = data.toString().split('\n');
      var obj = {};
      array.forEach(function(line) {
```

```
        var tmp = line.split(':');
        obj[tmp[0]] = tmp[1];
      });
      //输出页面总数
      console.log(obj['NumberOfPages']);
   });
   pdftk.stderr.on('data', function(data) {
      console.log('stderr: ' + data);
   });
   pdftk.on('exit', function(code) {
      console.log('child process exited with code' + code);
   });
```

PDF Toolkitdata_dump 返回结果如下所示：

```
stdout: InfoKey: Creator
InfoValue: PrintServer150&#0;
InfoKey: Title
InfoValue: &#0;
InfoKey: Producer
InfoValue: Corel PDF Engine Version 15.0.0.431
InfoKey: ModDate
InfoValue: D:20110914223152Z
InfoKey: CreationDate
InfoValue: D:20110914223152Z
PdfID0: 7fbe73224e44cb152328ed693290b51a
PdfID1: 7fbe73224e44cb152328ed693290b51a
NumberOfPages: 3
```

这种格式很容易转换为一个易于访问个体属性的对象。

PDF Toolkit 是一个响应式工具，当你在暂停一个网页等待响应结束时必须要小心。接下来我们会构建一个简单的 PDF 上传工具，来说明如何从 Node Web 程序使用 PDF Toolkit，以及如何应对开销很大的图形处理程序导致的时延。

创建一个 PDF uploader 和处理由图像导致的延时

PDF Toolkit 分解和合并 PDF 文档的能力对于允许用户上传和下载 PDF 的网站来说非常有用，可以提供对每个 PDF 页面的独立访问接口。考虑下 Google Docs 或者类似 Scribd 的网站，都可以进行 PDF 的分享。

这类型程序的组件包括：

- 一个 form，选择用于上传的 PDF 工具；
- Web service（网络服务），接收 PDF 文档，初始化 PDF 处理过程；

- 子进程，包含 PDF Toolkit，将 PDF 文档分解为独立的页面；
- 对用户的响应，提供上传文档的链接和访问独立页面的链接。

分解 PDF 的组件首先必须为分解后产生的页面创建目录以及决定分解操作后各页面的名字。这就需要访问 Node 文件系统模块来为分级后的文件创建目录。很明显，较大的文件花费的时间会比较长，所以不必挂起页面等待 PDF Toolkit 完成的响应，程序可以给用户发送包含新上传的文件链接的邮件。这会使用到我们之前没有接触过的模块——-Emailjs。该模块提供基本的邮件功能。

通过 npm 安装 Emailjs：

```
npm install emailjs
```

上传 PDF 的表单很简单，不需要解释什么。除了用户名和邮件地址之外额外添加了一个用于输入文件的区域，并设置 method 属性为 POST，action 为该网络服务。因为我们上传的是文件，所以 enctype 字段必须被设置为 multipart/from-data。示例 12-2 显示了完成的表单页面。

例 12-2　上传 PDF 文件的表单

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8"/>
 <title>Upload PDF</title>
 <script>
   window.onload = function () {
      document.getElementById('upload').onsubmit = function () {
         document.getElementById('submit').disabled = true;
      };
   }
   </script>
 </head>
<body>
<form id="upload" method="POST" action="http://localhost:8124"
enctype="multipart/form-data">
 <p><label for="username">User Name:</label>
   <input id="username" name="username" type="text" size="20" required/></p>
 <p><label for="email">Email:</label>
   <input id="email" name="email" type="text" size="20" required/></p>
 <p><label for="pdffile">PDF File:</label>
   <input type="file" name="pdffile" id="pdffile" required/></p>
 <p>
 <p>
 <input type="submit" name="submit" id="submit" value="Submit"/>
```

```
</p>
</form>
</body>
```

我们现在可以复习一下客户端 JavaScript 的技能了，在表单提交的时候禁止 button 的提交事件。上述表单使用了 HTML required 属性，可以确保需要的属性数据存在。

提供网络服务的应用程序可以使用 Connect 中间件同时处理对表单的请求以及 PDF 上传。这次没有使用 Express 架构。

在程序中，Connect static 中间件用于提供静态文件服务，directory 中间件用于在目录被访问时美化打印目录中的列表。其他需要的功能就是解析上传 PDF 文件和表单数据的过程。程序使用 Connect parseBody 方法，用于处理 POST 过来的各种类型的数据：

```
connect()
        .use(connect.bodyParser({uploadDir: __dirname + '/pdfs'}))
        .use(connect.static(__dirname + '/public'))
        .use(connect.directory(__dirname + '/public'))
        .listen(8124);
```

接下来这些数据会传递给一个自定义的中间件 upload。upload 可以处理数据，并调用自定义的模块来处理 PDF 文件。bodyParser 中间件将可以从 request.body 对象中取得 username、email 信息，并从 request.files 对象中获取上传的文件。如果文件上传完成，会被作为 pdffile 对象，这是表单里文件上传区域的名字。你需要对文件 type 进行额外的测试来保证上传的文件是 PDF 类型。

示例 12-3 中包含 PDF 服务程序的全部代码。

示例 12-3　PDF 上传网络服务程序

```
var connect = require('connect');
var pdfprocess = require('./pdfprocess');
// 如果 POST
// 上传文件, PDF 分解, ack 响应
function upload(req, res, next) {
  if ('POST' != req.method) return next();

  res.setHeader('Content-Type', 'text/html');
  if (req.files.pdffile&&req.files.pdffile.type === 'application/pdf') {
     res.write('<p>Thanks ' + req.body.username +
             ' for uploading ' + req.files.pdffile.name + '</p>');
     res.end("<p>You'll receive an email with file links when processed.</p>");

     // post 上传处理
     pdfprocess.processFile(req.body.username, req.body.email,
    req.files.pdffile.path, req.files.pdffile.name
  )
      ;
```

```
    } else {
  res.end('The file you uploaded was not a PDF');
  }
  }
  // 按顺序
  // 静态文件
  // POST -上传文件
  // 或者，目录列表
  connect()
          .use(connect.bodyParser({uploadDir:__dirname + '/pdfs'}))
          .use (connect.static(__dirname + '/public'))
          .use(upload)
           .use(connect.directory(__dirname + '/public'))
          .listen(8124);

  console.log('Server started on port 8124');
```

在自定义的模块 pdfprocess 中，程序按以下步骤处理 PDF 文件：

1. 如果 pdfs 目录下没有 users 子目录则创建一个；
2. 为当前上传的 PDF 文件创建一个包含时间戳的唯一命名；
3. 用 PDF 文件名和时间戳一起在 users 子目录下为 PDF 创建一个新的子目录；
4. PDF 从临时的上传文件目录转移到新的目录中，并重命名；
5. PDF Toolkit 对该文件进行操作，所有的独立 PDF 文件都放在 pdfs 目录中；
6. 发送邮件给用户，包含可以访问上传的 PDF 文件以及页面的链接。

文件系统的功能有 Node File System 模块提供，email 功能则由 Emailjs 模块负责，PDF Toolkit 功能在子进程中完成。子进程并不返回任何数据，所以只能捕捉到子进程的 exit 和 error 事件。示例 12-4 包含了程序最后一部分代码。

示例 12-4　处理 PDF 文件模块和发送包含可访问文件的链接给用户

```
var fs = require('fs');
var spawn = require('child_process').spawn;
var emailjs = require('emailjs');
module.exports.processFile = function (username, email, path, filename) {
     // 首先如果 user 目录不存在则创建 user 目录
     fs.mkdir(__dirname + '/public/users/' + username, function (err) {
         // 如果不存在则创建文件目录
         var dt = Date.now();
         // 之后信息使用的链接
var url = 'http://examples.burningbird.net:8124/users/' + username +
         '/' + dt + filename;
         // 放置文件的目录
```

```
        var dir = __dirname + '/public/users/' + username + '/' +
                dt + filename;
        fs.mkdir(dir, function (err) {
          if (err)

              return console.log(err);
            // 重命名
            var newfile = dir + '/' + filename;
            fs.rename(path, newfile, function (err) {
              if (err)
                 return console.log(err);

              //burst pdf
              var pdftk = spawn('pdftk', [newfile, 'burst', 'output',
              dir + '/page_%02d.pdf' ]);
              pdftk.on('exit', function (code) {
                 console.log('child process ended with ' + code);
                 if (code != 0)
                     return;
              console.log('sending email');
              // 发送邮件

              var server = emailjs.server.connect({
                  user:'gmail.account.name',
                  password:'gmail.account.passwod',
                  host:'smtp.gmail.com',
                  port:587,
                  tls:true
                  });

              var headers = {
                  text:'You can find your split PDF at ' + url,
                  from:'youremail',
                  to:email,
                  subject:'split pdf'
              };

              var message = emailjs.message.create(headers);

              message.attach({data:"<p>You can find your split PDF at " +
                                  "<a href='" + url + "'>" + url + "</a></p>",
                                  alternative:true});
              server.send(message, function (err, message) {
                  console.log(err || message);
              });
              pdftk.kill();
            });
            pdftk.stderr.on('data', function (data) {
              console.log('stderr: ' + data);
            });
          });
       });
    });
};
```

代码中粗体部分是子进程调用 PDF Toolkit。语法如下：

```
pdftk filename.pdf burst output /home/location/page_%02d.pdf
```

参数按次序依次是文件名，操作以及输出目录。在这里的操作 burst，是指将 PDF 分解为不同的独立页面。Output 目录告诉 PDF Toolkit 将分解后的页面放在指定的目录中，并提供了格式化的页面名称。比如第一页为 page_01.pdf，第二页为 page_02.pdf，以此类推。我们本来可以使用 Node 的 process.chdir 来改变进程执行的目录，但是指定了 PDF Toolkit 操作的目标路径就不需要这种方式了。

发送 E-mail 使用 Gmail 的 SMTP 服务器，使用 TLS（Transport Layer Security，安全传输层），端口为 587，需要指定 Gmail 用户名和密码。当然，你还可以使用自己的 SMTP 服务器。发送的邮件内容既有纯文本也有 HTML 格式附件（对于使用邮件客户端可以处理 HTML 格式的用户）。

程序的结果就是发送一个链接给用户，使用户可以直接访问上传的 PDF 和分解后的页面。Connect directory 中间件确保了目录的内容可以正常显示。图 12-1 显示上传了一个很大的关于全球变暖的 PDF 文件的结果。

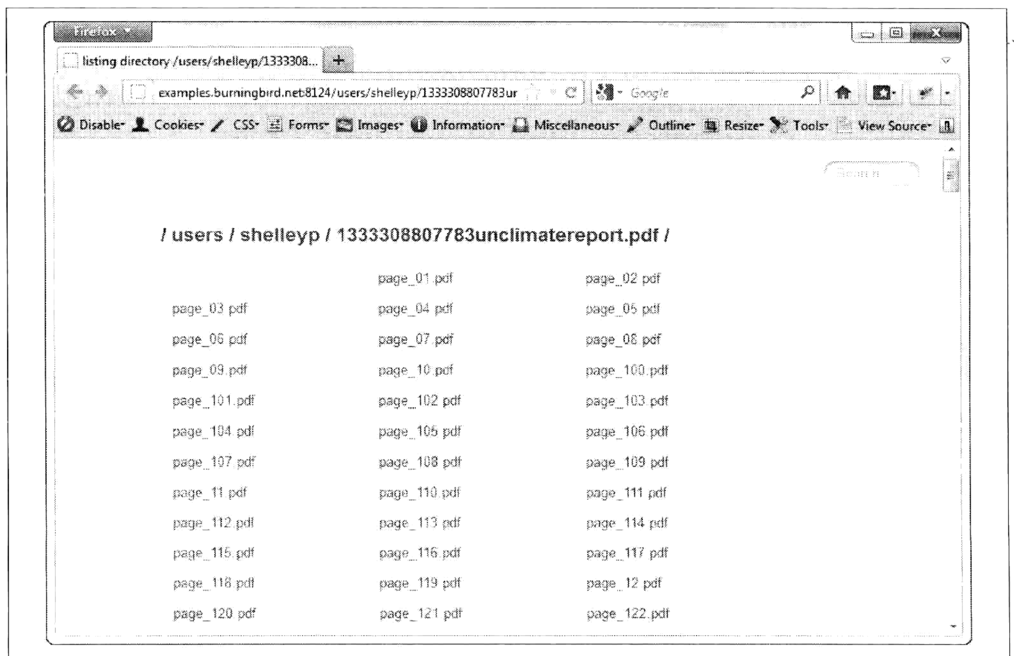


图 12-1 上传大文件，通过 PDF Toolkit 分解后的最终结果

用这种通过 E-mail 发送确认信息给用户的实现方式，用户不需要一直等待 PDF 的处理过程（Node 服务也不需要一直挂起等待处理过程完成）。

提示

当然，用户依然需要花费时间在上传 PDF 文件上，本程序并不涉及与大文件上传有关的问题。

12.1.2 使用 PDFKit 创建 PDF

如果你想要的不是使用子进程和命令行工具，或者你需要创建 PDF 并且可以管理现有的 PDF，Node 有一些模块提供了这种 PDF 功能的支持，使用最多的是 PDFKit。

PDFKit 是用 CoffeeScript 编写的，但是你并不需要了解 CoffeeScript 才能使用该模块，因为 API 是暴露给 JavaScript 使用的。该模块提供了创建 PDF 文档，添加页面，组织文本和图形以及嵌入图片的功能。该模块未来的功能除了这些，还应该添加其他的，如 PDF 提纲、渐变、表格以及其他特性。

使用 npm 安装 PDFKit：

```
npm installpdfkit
```

在程序中，以创建一个新的 PDF 文档作为开始：

```
var doc = new PDFDocument();
```

接下来，你可以使用 API 添加字体、新的网页、图形。为了简化开发过程可以链式调用 API 方法。

为了说明如何在 JavaScript 中使用该模块，我将模块开发人员的一个 CoffeeScript 例子转换为 JavaScript。从头开始，创建好 PDF 文档之后，为文档添加一个 TrueType 字体，字号设置为 25pix，设置文本坐标为（100，100）：

```
doc.font('fonts/GoodDog-webfont.ttf')
   .fontSize(25)
   .text('Some text with an embedded font! ', 100, 100);
```

之后，需要添加一个新的 PDF 页面，重新设置字号为 25pix，新文本坐标为（100，100）：

```
doc.addPage()
   .fontSize(25)
   .text('Here is some vector graphics…', 100, 100);
```

保存文档的坐标系统，调用向量图形功能来画一个红色的三角：

```
doc.save()
    .moveTo(100,150)
    .lineTo(100,150)
    .lineTo(200,250)
    .fill("#FF3300");
```

下一部分代码设置坐标系统比例为 0.6，转换了原点，画了一个星星的边沿，填充红色，然后将文档恢复为初始的坐标系统和比例：

```
doc.scale(0.6)
    .translate(470, -380)
    .path('M 25,75 L 323, 301 131, 161 369,161 177, 301 z')
    .fill('red', 'even-odd')
    .restore();
```

如果你使用其他的向量图形系统，比如 Canvas，这部分看起来应该很熟悉。如果没有使用过，你可以先学习后续部分的 Canvas 例子，然后再回到这个例子。

添加另一个页面，填充颜色修改为蓝色，并在页面添加链接。随后将该文档写成一个 output.pdf 文件：

```
doc.addPage()
    .fillColor("blue")
    .text('Here is a link!', 100, 100)
    .underline(100, 100, 160, 27, {color:"#0000FF"})
    .link(100, 100, 160, 27, 'http://google.com/');
doc.write('output.pdf');
```

手动创建 PDF 文档是一个繁琐的过程。但是，我们可以很容易编写程序利用 PDFKit API 接收数据存储的内容并生成 PDF。我们也可以使用 PDFKit 按要求生成网页的 PDF 文档，或者提供可供保存数据的截图。

要知道的是，模块的很多方法都不是异步的，所以在生成 PDF 的过程中需要一直等待，所以具体情况具体分析。

12.2 从子进程访问 ImageMagick

ImageMagick 是一个支持 MAC，Windows 和 Unix 操作系统的很强大的命令行图形工具。可以用 ImageMagick 来裁剪图片或者重定义图片大小，访问图片的数据元，制作动画及很多特效。同时 ImageMagick 占用很多资源，操作时间较长，通常由图片的不同大小和操作决定。

Node 有很多有关 ImageMagick 的模块。其中一个模块就是 imagemadick，提供了对 ImageMagick 功能的封装。但是这个模块已经一段时间没有更新了。另一个模块是 gm，提供了一系列预定义的功能，在后台与 ImageMagick 交互。你可能会发现使用这些模块与直接使用 ImageMagick 一样简单。在 Node 程序中使用 ImageMagick 你所需要做的全部事情就是安装 ImageMagick 以及调用子进程。

ImageMagick 提供了可供使用的不同工具来完成不同功能：

animate

在图形界面上制作动画。

compare

提供原图与修改后的图片在参数和视觉效果上的差异对比。

composite

重叠两个图像。

conjure

执行 Magick Scripting Language（MSL）语言描述的脚本。

convert

使用任意可能的操作对图形进行转换，比如裁剪、重定义大小以及添加特效。

display

在图形界面上显示图像。

identify

描述一个图像或者多个图像文件的格式和其他特性。

import

在图形界面上对可见的窗口截图并保存为文件。

mogrify

在原图上对图形进行修改（裁剪、重定义大小、抖动等）并直接保存修改后的结果覆盖原图。

montage

合并图形。

stream

流式存储图片，一次一个像素。

其中几种工具都与图形界面有关，从 Node 程序角度来看没有什么意义。但是，convert、mogrify、montage、identify 和 stream 几个工具在 Node 程序中有非常有趣的用法。在本节和下一节中，我们主要介绍其中一个：convert。

提示

尽管我们的关注点是 convert，要知道本章中所讲到的内容都可以用于 mogrify，除了 mogrify 本身的修改会覆盖原图。

convert 工具是 ImageMagick 的主要部分。借助 convert，你可以对图形进行一些奇妙的转换，然后将结果存为另一个文件。你可以提供一个适度的模糊或者锐化图片，给图片添加文字注释，将图片作为背景，裁剪，重定义大小甚至用颜色填充工具替换掉图像中每一个像素的颜色。几乎没有什么是 ImageMagick 不能做到的。当然，并不是每个操作都是等价的，特别是当你考虑到操作可能耗费时间的时候。一些图形转换可能很快，另一些可能相当漫长。

为了演示如何在 Node 程序中使用 convert，示例 12-5 中的小自含型程序通过命令行指定了一个图片名，然后调整图片大小使其适应一个宽度不超过 150px 的空间。不管其原始类型，将图片格式转换为 PNG。

这个过程的命令行实现方式为：

```
convert photo.jpg -resize '150' photo.jpg.png
```

我们需要为子进程参数数组传递四个数据：原始图片，-resize 标识，-resize 标识的数值，新图片的名称。

示例 12-5　在 Node 程序中使用 ImageMagick convert 工具调整图片大小

```
var spawn = require('child_process').spawn;
// 获取图片
var photo = process.argv[2];
// 转换参数数组
var opts = [
photo,
```

```
'-resize',
'150',
photo + ".png"];

// convert
var im = spawn('convert', opts);
im.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});
im.on('exit', function (code) {
   if (code === 0)
        console.log('photo has been converted and is accessible at '
                    + photo + '.png');
});
```

ImageMagick convert 工具悄无声息地处理完图片，没有子进程的 data 事件需要处理。我们唯一需要关心的事件是当图像处理完毕后的 error 和 exit 事件。

当你想完成一个更复杂的图形处理的时候像 ImageMagick 这种程序开始变得有趣。一个很受欢迎的 ImageMagick 图形效果是拍立得特效：沿中心轻微旋转图片，添加 border 和 shadow（阴影）使图片看起来像拍立得的照片。这种效果非常受欢迎，所以已经有预先定义的设置。但是在使用新的预定义设置之前，我们需要使用如下的命令行（来自于 ImageMagick 用例）：

```
convert thumbnail.gif \
        -bordercolor white -border 6 \
        -bordercolor grey60 -border 1 \
        -background none -rotate 6 \
        -background black \( +clone -shadow 60x4+4+4 \) +swap \
        -background none -flatten \
        polaroid.png
```

参数很多，并且参数的格式之前并没有见到过。接下来应该怎样将其转换为子进程的参数数组呢？

详细分析下。

在命令行中看起来像单个参数的（\(+clone -shadow 60x4+4+4 \)）绝不是 Node 子进程。示例 12-6 是示例 12-5 中的转换工具的一个变形，用图片的拍立得特效替换了原来对图片大小的调整。特别注意一下粗体的部分。

示例 12-6　在 Node 程序中使用 ImageMagick 为图片添加 Polaroid 效果

```
var spawn = require('child_process').spawn;
// 获取图片
var photo = process.argv[2];
```

```
// 转换为数组
var opts = [
photo,
"-bordercolor", "snow",
"-border", "6",
"-background", "grey60",
"-background", "none",
"-rotate", "6",
"-background", "black",
"(", "+clone", "-shadow", "60x4+4+4", ")",
"+swap",
"-background", "none",
"-flatten",
photo + ".png"];

var im = spawn('convert', opts);
```

代码中粗体部分说明了命令行中的一个参数如何成为子进程的五个参数。程序运行结果如图 12-2 所示。

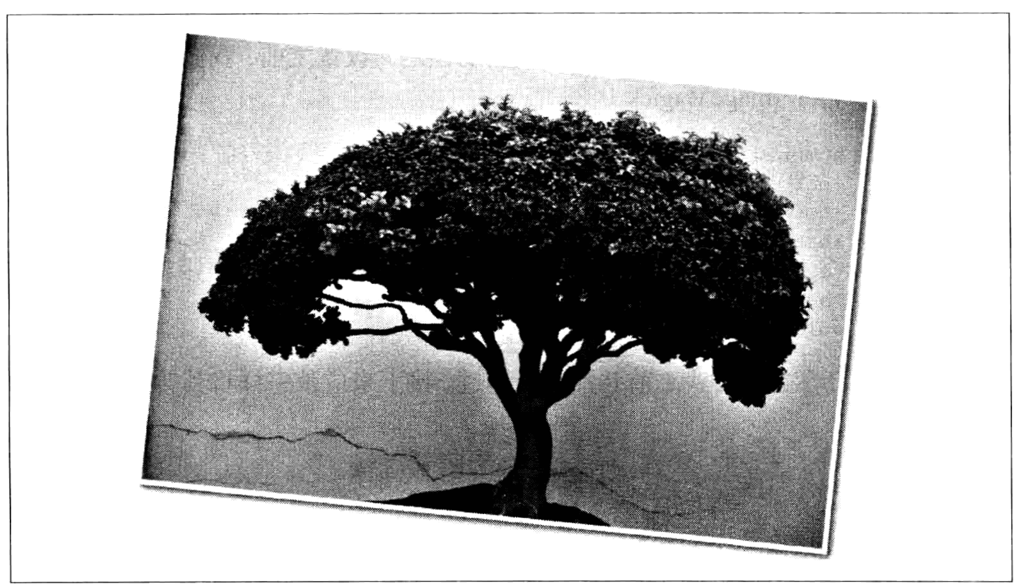


图 12-2　Node 程序运行结果，图片添加了拍立得特效

在 Node 程序不可能通过命令行使用 ImageMagick 子进程，毕竟你可以直接运行 ImageMagick 工具。但是你可以使用进程和 ImageMagick 工具混合的方式在一张图片上进行多个会话，或者通过网站提供服务（比如允许用户修改图片大小作为头像，或者在资源功效的网站上为上传图片打水印）。

创建一个使用 ImageMagick 的 Web 应用的关键点与之前的 PDF 程序一样：如果处理过程变得很慢（特别是应对数量巨大的并发用户），你需要考虑提供允许上传图片文件然后提供链接给用户完成操作（网站或者 E-mail 都可以）的异步处理方式，而不是一直等到所有处理完成。

我们可以修改示例 12-3 和示例 12-4 的代码，对任何上传的图片都添加拍立得的效果。并且，我们还可以将示例 12-3 代码转换为一个模块，可以被重用于类似的场景：一个文件处理过程，为上传文件创建子目录，运行进程，并在同一个目录中存储处理完成的结果。

12.3 通过 HTTP 提供 HTML5 Video 服务

在第 6 章中我们创建了一个简单的 HTTP 服务器，提供静态文件和基本目录服务，可以处理 404 错误。我们用于测试该程序的一个网页包含了嵌入的 HTML5 video。该网页通过一个工具允许用户在视频播放期间任意时间点上点击来暂停或者播放视频。

包含 HTMl5 video 的程序需要使用 Connect 模块的静态网络服务器而不是自制的 Web 服务器。这一选择的原因在于自制 Web 服务器无法处理 HTTP ranges。类似于 Apache 和 IIS 的 HTTP 服务器都支持 range，Connect 模块也支持，但是我们的静态服务器不支持。

在本节中，我们会对示例 6-2 中的 Web 服务器添加支持 range 的功能。

提示

支持 range 所能提供的服务远远不止 HTML5 video，还可以用于下载大容量的文件。

Range 位于 HTTP header 中，用于提供下载资源的开始和结束位置，比如视频文件。以下是我们支持 HTTP range 的步骤：

1. response header 中添加 Accept-Ranges:bytes 表示可以处理带有 range 的请求；

2. 在 request header 中查找 range 请求信息；

3. 如果找到 range 请求，解析出开始和结束位置的信息；

4. 验证开始和结束信息的值都是数字，并且不超出所访问的资源的长度；

5. 如果没有提供结束信息，则设置结束位置为资源长度。如果没有提供开始信息，设置为 0；

6. 在 response header 中创建 Content-Range 用于保存开始，结束和资源长度信息；
7. 在 response-header 中创建 Content-Length 保存通过结束和开始信息计算出来的请求资源信息长度的结果；
8. 将状态码从 200 设置为 206（部分情况下）；
9. 将一个包含开始和结束位置信息的对象传递给 createReadStream 方法。

当一个 Web 客户端请求 Web 服务器的某个资源，Web 服务器可以告知客户端服务器支持 range，并通过以下 header 内容提供 range 基本单位：

```
Accept-Ranges: bytes
```

所以对 Web 服务器第一个需要修改的内容是添加新的 header 内容：

```
res.setHeader('Accept-Ranges', 'bytes');
```

客户端接下来会按以下格式发送 range 请求：

```
bytes=startnum-endnum
```

startnum/endnum 值是 range 的开始值和结束值。在播放过程中可以多次发送这类请求。比如，以下内容就是实际情况中在包含 HTML5 video 的网页上，当视频开始播放后在播放过程中点击时间轴，从页面发出的 range 请求内容：

```
bytes=0-
bytes=7751445-53195861
bytes=18414853-53195861
bytes=15596601-18415615
bytes=29172188-53195861
bytes=39327650-53195861
bytes=4987620-7751679
bytes=17251881-18415615
bytes=17845749-18415615
bytes=24307069-29172735
bytes=33073712-39327743
bytes=52468462-53195861
bytes=35020844-39327743
bytes=42247622-52468735
```

对我们的服务器来说，下一步需要添加的就是检查是否为 range 请求，如果是的话，解析出来开始和结束值。检查 range 请求代码为：

```
if(req.headers.range) {…}
```

我创建了一个 processRange 函数来解析 range 中的开始和结束值。processRange 按横杠（-）分解字符串，然后从分解后的两个字符串中提取数字。该函数检查了起始值是否存在，类型是否为数字并且有没有超过文件的长度（如果请求中的 range 值不合法，则返回 416 状态码），同时也对结束点的值做了类似上述的检查，当结束值不存在的时候则设置结束点为视频文件长度。该方法返回一个包含开始和结束值的对象：

```
function processRange(res, ranges, len) {
  var start, end;
  // 从 range 中提取 start 和 stop
  var rangearray = ranges.split('-');

  start = parseInt(rangearray[0].substr(6));
  end = parseInt(rangearray[1]);

  if (isNaN(start)) start = 0;
  if (isNaN(end)) end = len -1;

  // 若 start 超过文件长度
  if (start >len - 1) {
      res.setHeader('Content-Range', 'bytes */' + len);
      res.writeHead(416);
      res.end();
  }
  // end 不能超过文件长度
  if (end >len - 1)
     end = len - 1;
  return {start:start, end:end};
}
```

该功能的下一个部分是在 reponse header 中添加 Content-Range，提供 range 的起始值和请求资源的长度，格式如下：

```
Content-Range bytes 44040192-44062881/44062882
```

Response header 中也要包含内容的长度（Content-Length），由结束值减去开始值计算，HTTP 状态码设置为 206，表示 Partial Content（部分内容）。

最后，start 和 end 值也被作为选项在调用时传递 createReadStream 方法。这一实现确保了请求到的流资源在完整流媒体中的正确位置。

示例 12-7 将对 Web 服务器的修改汇总到一起，现在可以提供 HTML5 video range 服务。

示例 12-7　支持 range 的 Web 服务器

```
var http = require('http'),
    url = require('url'),
    fs = require('fs'),
```

```
    mime = require('mime');

function processRange(res, ranges, len) {

  var start, end;

  // 从 range 中提取 start 和 end
  var rangearray = ranges.split('-');

  start = parseInt(rangearray[0].substr(6));
  end = parseInt(rangearray[1]);

  if (isNaN(start)) start = 0;
  if (isNaN(end)) end = len -1;

  // 若 start 超过文件长度
  if (start >len - 1) {
      res.setHeader('Content-Range', 'bytes */' + len);
      res.writeHead(416);
      res.end();
  }
  //end 不能超过文件长度
  if (end >len - 1)
      end = len - 1;

  return {start:start, end:end};
}
http.createServer(function (req, res) {

   pathname = __dirname + '/public' + req.url;

   fs.stat(pathname, function (err, stats) {
      if (err) {
         res.writeHead(404);
         res.write('Bad request 404\n');
         res.end();
      } else if (stats.isFile()) {

         var opt = {};

         // 如果没有 range
         res.statusCode = 200;

         varlen = stats.size;

         // 带有 range 的请求
         if (req.headers.range) {
            opt = processRange(res, req.headers.range, len);
```

```
            // 设置长度
            len = opt.end - opt.start + 1;

            // 设置状态码 206
            res.statusCode = 206;
            // 设置 header
            var ctstr = 'bytes ' + opt.start + '-' +
                        opt.end + '/' + stats.size;

            res.setHeader('Content-Range', ctstr);
         }
         console.log('len ' + len);
         res.setHeader('Content-Length', len);

         // 内容类型
         var type = mime.lookup(pathname);
         res.setHeader('Content-Type', type);
         res.setHeader('Accept-Ranges', 'bytes');

         // 创建可读取的流
         var file = fs.createReadStream(pathname, opt);
         file.on("open", function () {

            file.pipe(res);
         });
         file.on("error", function (err) {
           console.log(err);
         });

         } else {
          res.writeHead(403);
          res.write('Directory access is forbidden');
          res.end();
         }
   });
}).listen(8124);
console.log('Server running at 8124/');
```

对 Web 服务器的修改说明了 HTTP 以及其他网络功能并不复杂，只是很繁琐。关键之处在于将每个功能分解为较小的独立功能，然后每次添加一个小功能的代码（完成之后进行测试）。

现在，用户可以在网页（例子中的网页）上视频播放的时间轴上进行点击了。

12.4 创建和流化画布内容（Canvas Content）

canvas 元素成为游戏开发人员、艺术家和统计人员的最爱，因为 canvas 元素可以

在客户端页面上创建动态和交互式的图形。Node 中也有支持 canvas 的模块，比如本节即将介绍的 node-canvas，或者 canvas（本节我们使用 node-canvas）。Node-canvas 模块是基于 Cario 的跨平台的向量图形库，长久以来深受开发人员喜爱。

Npm 安装 node-canvas：

```
npm install canvas
```

node-canvas 模块可以提供你在客户端页面上使用到的所有 canvas 功能。创建一个 Canvas 对象和 context，然后在 context 中进行绘画，可以显示结果或者将结果保存为 JPEG 或者 PNG。

提示

需要知道的是，Canvas 中的一些功能，比如在图形上进行操作，都要求 Cario 1.10 版本以上。

有一些附加的功能只能用于服务器而不能用于客户端。服务器端允许流化 Canvas 对象为一个文件（PNG 或者 JPEG），为之后的访问固化资源。还可以将 Canvas 对象转换为数据的 URI，包含 img 元素的 HTML 页面，或者从外部资源（比如文件或者 Redis 数据库）读取图片直接用于 Canvas 对象。

进入正题介绍如何使用 node-canvas 模块，示例 12-8 创建 canvas 画布并为之后的访问流化图片为 PNG 文件。例子中使用了 Mozilla Developer Network 例子中的旋转后的图片，添加了边界和阴影。完成之后，流化为 PNG 文件方便之后访问。很多客户端程序可以使用的功能也可以用于 Node 程序。真正只属于 Node 的部分是在最后将图形存储为文件。

示例 12-8　通过 node-canvas 创建图形并存储为 PNG 文件

```
var Canvas = require('canvas');
var fs = require('fs');

  // 创建 canvas and context
  var canvas = new Canvas(350, 350);
  var ctx = canvas.getContext('2d');

  // 创建带阴影的长方形
  // 存储 context
  ctx.save();
  ctx.shadowOffsetX = 10;
  ctx.shadowOffsetY = 10;
  ctx.shadowBlur = 5;
  ctx.shadowColor = 'rgba(0,0,0,0.4)';

  ctx.fillStyle = '#fff';
```

```
ctx.fillRect(30, 30, 300, 300);
// 完成阴影
ctx.restore();
ctx.strokeRect(30, 30, 300, 300);

// MDN 例子：优化图片，在之前创建出来的正方形
//中插入偏移量
ctx.translate(125, 125);
for (i = 1; i< 6; i++) {
  ctx.save();
  ctx.fillStyle = 'rgb(' + (51 * i) + ',' + (255 - 51 * i) + ',255)';
  for (j = 0; j <i * 6; j++) {
  ctx.rotate(Math.PI * 2 / (i * 6));
  ctx.beginPath();
  ctx.arc(0, i * 12.5, 5, 0, Math.PI * 2, true);
  ctx.fill();
  }
  ctx.restore();
}
// 存储为 PNG 文件
var out = fs.createWriteStream(__dirname + '/shadow.png');
var stream = canvas.createPNGStream();

stream.on('data', function (chunk) {
  out.write(chunk);
});
stream.on('end', function () {
  console.log('saved png');
});
```

一旦运行了 Node 程序，用你喜欢的浏览器访问 shadow.png。图 12-3 显示了生成的图片。

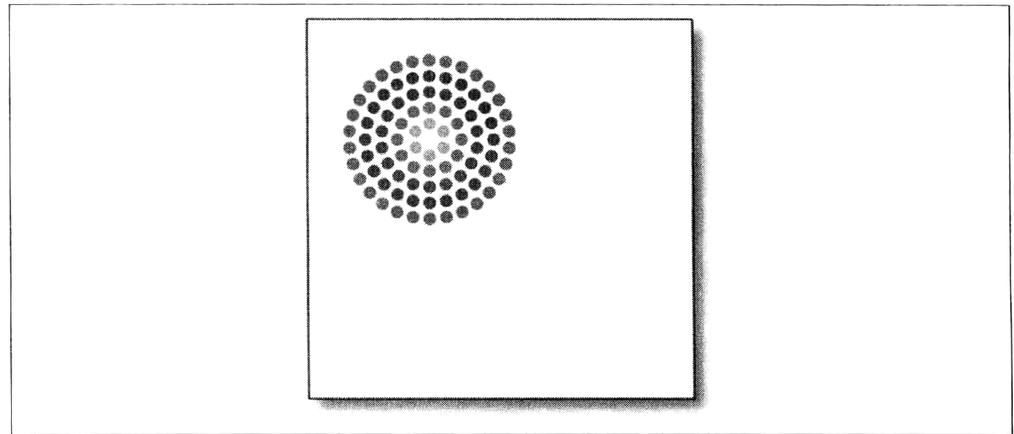


图 12-3　node-canvas 生成的图形

你并不会像在网页中一样在 Node 程序中使用 Canvas 对象。一个使用 node-canvas 的例子是动态的时钟，需要不断的 HTTP 刷新。如果你需要一个客户端的动态时钟，则需要在客户端使用 canvas 元素。

Canvas 对于服务器的意义在于提供了服务器通过数据表达图形的能力，比如数据库查询，Redis 数据库中的数据，日志文件或者其他一些服务器上的原生数据。在服务器上生成图形不仅提供了后续访问的能力，还可以通过在服务器上处理图形来限制客户端的数据流，而不是通过发送数据到客户端再生成图形。

在 Node 程序中使用 canvas 的意义还存在于用于处理游戏中需要适应用户操作的组件，特别是生成之后需要多次访问的情况。

第 13 章

WebSockets 和 Socket.IO

在本章中，我们将通过实现一些客户端与服务端示例程序，来对 WebSockets 和 Socket.IO 进行说明。

WebSockets 是一个相对较新的 Web 技术，它能在客户端与服务器应用程序之间建立直接的实时双向通信。该通信是基于 TCP 协议（传输控制协议）的，并通过套接字接口实现。Socket.IO 库为实施这项技术提供了必要的支持。Socket.IO 不仅只为 Node 应用程序提供了可用模块，它还包含了一个客户端 JavaScript 库，以便于建立客户端通信信道；此外，它还可以作为一个 Express 中间件使用。

在本章中，我将对 Socket.IO 在客户端以及服务端上的工作原理进行介绍，通过它我们能更好地了解 WebSockets。

13.1 WebSockets

在使用 Socket-IO 之前，我想对 WebSockets 先做下简要介绍。要做到这一点，首先需要解释下*双向全双工通信*（*bidirectional full duplex communication*）。

*全双工*是指在任何形式的数据通信过程中，两个方向上的传输同时进行。*双工*是指一个通信的两个端点都可以发起通信。与此相对的是*单工*通信，它只允许通信中的一个端点作为数据发送方，而其他端点都是接收方。WebSockets 为 Web 客户端（如浏览器）提供了与服务器应用程序建立双向全双工通信的能力，而且它没有使用 HTTP 通信，因为这会为通信处理增加不必要的开销。

WebSockets 是标准化的，它属于万维网联盟（W3C）制定的 WebSockets API 规范的一部分。该技术的起步比较曲折，因为早期的一些浏览器于 2009 年开始实施了

WebSockets，但却导致了严重的安全问题，使得他们放弃了对 WebSockets 的支持，或者只将它作为一个可选项来启用。

后来 WebSockets 协议做了改进并解决了安全问题，FireFox、Chrome 和 Internet Explorer 浏览器都支持该新协议。目前，Safari 和 Opera 还只能支持旧版本的 WebSockets，而且还必须通过配置选项来手动启用它。而大多数移动浏览器对 WebSockets 的支持也都比较有限，或者仅支持早期的 WebSockets 规范。

Socket.IO 库定位于解决这个问题，它能灵活采用不同机制在客户端和服务器之间建立双向通信，一般情况下它会按序尝试使用如下几种机制：

- WebSockets
- Adobe Flash Socket
- Ajax long polling
- Ajax multipart streaming
- Forever iFrame for IE
- JSONP Polling

通过这个列表，我们可以看出：Socket.IO 能够为当下我们使用的大多数浏览器提供双向通信支持。

提示

虽然技术上讲，WebSockets 并不是一个具体实现，但在本章的示例中，我依然使用了 “WebSockets” 这个名字来描述这项通信技术，因为它比 bidirectional full-duplex communication 更简短。

13.2 Socket.IO 简介

在开发 WebSockets 应用程序之前，你首先需要在服务器上安装 Socket.IO。我们可以使用 npm 来安装该模块：

```
npm install socket.io
```

一个 Socket.IO 应用程序包含两个组成部分：服务端应用和客户端应用。在本节的示例代码中，服务端是一个 Node 应用程序，而客户端是一个包含有 JavaScript 代码块的 HTML 页面。两者都是通过修改 Socket.IO 网站提供的示例代码得到的。

13.2.1 一个简单的通信范例

本节的 client/server 应用程序演示了如何在客户端和服务器之间建立通信，然后往返发送文本字符串并将此过程展现在页面上。客户端会将最近接收到的字符串回应给服务端，服务端则修改该字符串并将其再发送回客户端。

客户端应用程序使用 Socket.IO 库创建一个 WebSockets 连接，并侦听任何标记为 news 的事件。当接收到事件后，应用程序会提取事件所携带的文本信息，并把它输出到 Web 页面。同时通过 echo 事件将文本返回给服务器。示例 13-1 包含了客户端网页的完整代码。

示例 13-1 Socket-IO 应用：客户端 HTML 页面

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>bi-directional communication</title>
  <script src="/socket.io/socket.io.js"></script>
  <script>
     var socket = io.connect('http://localhost:8124');
     socket.on('news', function (data) {
        var html = '<p>' + data.news + '</p>';
        document.getElementById("output").innerHTML=html;
        socket.emit('echo', { back: data.news });
    });
</script>
</head>
<body>
<div id="output"></div>
</body>
</html>
```

服务端应用程序则会创建一个 HTTP 服务对象，但它只用一个 HTML 页面文件来响应客户端的 HTTP 请求。此外，当服务端与客户端建立套接字连接时，服务端会向客户端发送一个内容为“Counting ...”并且标记了 news 事件的消息。

当服务端收到 echo 事件时，它则会提取事件中包含的文本信息，并增加计数器值。该计数器保存在服务端应用程序中，当接收到 echo 事件后会被递增。当计数器到达 50 时，服务端将不再给客户端返回数据。示例 13-2 包含了服务端应用程序的源代码。

示例 13-2 Socket.IO 应用：服务端应用程序

```
var app = require('http').createServer(handler)
  , io = require('socket.io').listen(app)
  , fs = require('fs')

var counter;
```

```
app.listen(8124);

function handler (req, res) {
  fs.readFile(__dirname + '/index.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading index.html');
    }
    counter = 1;
    res.writeHead(200);
    res.end(data);
  });
}
io.sockets.on('connection', function (socket) {
  socket.emit('news', { news: 'world' });
  socket.on('echo', function (data) {
    if (counter <= 50) {
       counter++;
       data.back+=counter;
       socket.emit('news', {news: data.back});
    }
  });
});
```

在运行客户端和服务端应用程序后，用户无需做任何刷新或其他操作就可以通过页面看到计数器值不停更新，直到到达目标值。该 Socket.IO 应用程序在所有现代浏览器上都具有相同的行为，尽管在不同浏览器上使用的底层技术实现可能不一样。

news 和 echo 都是自定义事件。在新的套接字连接建立时，Socket.IO 对象还会收到 connection 事件。另外，服务端 socket 对象还支持如下事件：

message

　　每当收到通过 socket.send 发送的消息后会被触发。

disconnect

　　客户端或服务器断开时触发。

此外，客户端 socket 对象支持的事件列表如下：

connect

　　在建立好套接字连接后触发。

connecting

　　在尝试建立连接时触发。

disconnect

当套接字连接断开时触发。

connect_failed

建立连接失败时触发。

error

发生错误时触发。

message

收到通过 socket.send 发送的消息时触发。

reconnect_failed

Socket.IO 重建连接失败时触发。

reconnect

断开的连接被重新建立后时触发。

reconnecting

尝试建立断开的连接时触发。

如果你想控制 WebSockets 的行为，那么可以使用 send 代替 emit 发送消息，并监听 message 事件。例如，在服务器上应用程序可以使用 send 发送消息到客户端，然后通过监听 message 事件获取客户端的响应：

```
io.sockets.on('connection', function (socket) {
   socket.send("All the news that's fit to print");
   socket.on('message', function(msg) {
      console.log (msg);
   });
});
```

在客户端，应用程序也可以监听 message 事件，并使用 send 发送消息与服务端通信：

```
socket.on('message', function (data) {
   var html = '<p>' + data + '</p>';
   document.getElementById("output").innerHTML=html;
   socket.send('OK, got the data');
});
```

在上面的示例中，客户端使用了 send 方法来手动确认消息的接收。如果希望客户端在接收到事件后能自动应答，我们可以为 emit 方法传递一个回调函数作为它的最后一个参数：

```
io.sockets.on('connection', function (socket) {
   socket.emit('news', { news: "All the news that's fit to print" },
                      function(data) {
                         console.log (data);
                      } );
});
```

在客户端，我们就可以使用这个回调函数来向服务端返回一条消息：

```
socket.on('news', function (data, fn) {
   var html = '<p>' + data.news + '</p>';
   document.getElementById("output").innerHTML=html;
   fn('Got it! Thanks!');
});
```

以参数形式传递给 connection 事件处理函数的 socket 对象表示了服务器和客户端之间的一个连接，只要该连接没有断开，这个对象就一直存在。当连接中断后，Socket.IO 还会尝试重新建立连接。

13.2.2 异步世界里的 WebSockets

在只有一个客户端时，之前实现的应用程序可以正常工作。但它的失败之处在于没有考虑到 Node 的异步特性。在应用程序中，计数器是一个全局变量，如果一次只有一个客户访问应用程序，它工作得很好。但是，如果两个用户在同一时间访问应用程序，他们就会得到奇怪的输出结果：其中一个浏览器中显示的数字可能比另外一个浏览器中的小，而且在这两个浏览器中，我们都无法获得预期的结果。当并发用户增多后，这个影响会更明显。

因此，我们需要一种能将数据与套接字绑定的能力，这样数据与套接字就具有相同的生存周期和作用范围。幸运的是，获得这种能力并不困难，我们只需要在连接建立后，将数据直接写入到 socket 对象中即可。示例 13-3 是由示例 13-2 修改而来的，counter 不再是一个全局变量，而是被直接绑定在 socket 对象上。我用粗体文字标注了有变动的代码。

示例 13-3　将数据绑定到套接字

```
var app = require('http').createServer(handler)
  , io = require('socket.io').listen(app)
  , fs = require('fs')
```

```
app.listen(8124);

function handler (req, res) {
  fs.readFile(__dirname + '/index.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading index.html');
    }
    res.writeHead(200);
    res.end(data);
  });
}
io.sockets.on('connection', function (socket) {
   socket.counter = 1;
   socket.emit('news', { news: 'Counting...' });

   socket.on('echo', function (data) {
      if (socket.counter <= 50) {
         data.back+=socket.counter;
         socket.counter++;
         socket.emit('news', {news: data.back});
      }
   });
});
```

现在程序可以支持多个用户的并发访问了，并且每个用户都能得到完全相同的通信输出信息。注意，socket 对象会一直存在，直到套接字连接被关闭或者连接不能重建时。

警告

由于每个浏览器的行为都不会是完全一致的，因此计数速度可以快也可以慢，这取决于你所使用的浏览器以及用于建立通信的底层机制。

13.2.3 关于客户端代码

为了能让 Socket.IO 正常工作，客户端应用程序必须使用 Socket.IO 提供的客户端 JavaScript 库。通过如下脚本元素，这个库能被包含进了页面之中：

```
<script src="/Socket.IO/Socket.IO.js"></script>
```

那么是否必须将这个库文件放在 Web 服务器的顶层呢？答案是不需要。

在服务端应用程序创建了 HTTP Web 服务器对象后，该对象会被传递给 Socket-IO 的 listen 函数：

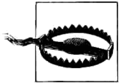
```
var app = require('http').createServer(handler)
  , io = require('Socket.IO'). listen(app)
```

这样，Socket.IO 就可以对所有发送到 Web 服务器的请求进行拦截并检查是否有如

下请求：

```
/Socket.IO/Socket.IO.js
```

Socket.IO 做了一些后台操作来决定应该为此请求提供什么样的响应内容。如果客户端支持 WebSockets，返回的文件将是一个支持客户端使用 WebSockets 建立连接的 JavaScript 库。如果客户端不支持 WebSockets，但却支持 Forever iFrame（IE9），它将返回对应的 JavaScript 客户端代码，并依此类推。



警告

不要尝试修改这个相对 URL，如果你这样做了，你的 Socket.IO 应用程序可能无法正常工作。

13.3　配置 Socket.IO

通常情况下我们不需要修改 Socket.IO 的默认配置。在之前几节的示例程序中，我没有改变过任何默认设置。不过，如果有需要的话，我们可以像在使用 Express 和 Connect 时一样，使用 Socket.IO 的 configure 方法来修改默认设置。你甚至可以为应用程序的各个运行环境指定不同的配置。

你可以参考 Socket.IO 的 wiki 页面（https://github.com/learnboost/Socket-IO/wiki/）中罗列的所有配置选项，我并不会在这里重复所有这些内容。相反，我只对一些我们学习使用 Socket.IO 时可能会用到的配置项进行说明。

你可以通过 transports 选项来更新可用传输方式。默认情况下，Socket.IO 在选择传输方时采用的优先级顺序是：

- websocket
- htmlfile
- xhr-polling
- jsonp-polling

还有一个默认情况下不启用的传输选项是 Flash Socket。如果将下面的代码添加到示例 13-3，那么当我们通过 Opera 或 IE 浏览器访问应用程序时，应用程序将使用 Flash Socket 进行通信（而不是 Ajax long polling 或 Forever iFrame）：

```
io.configure('development', function() {
   io.set('transports', [
```

```
            'websocket',
            'flashsocket',
            'htmlfile',
            'xhr-polling',
            'jsonp-polling']}
});
```

你还可以为不同的运行环境定义不同的配置：

```
io.configure('production', function() {
    io.set('transports', [
            'websocket',
            'jsonp-polling']);
});
io.configure('development', function() {
    io.set('transports', [
            'websocket',
            'flashsocket',
            'htmlfile',
            'xhr-polling',
            'jsonp-polling']);
});
```

还有一个用于控制日志输出信息等级的配置项（它能将调试信息输出到服务端控制台）。如果你想关闭它，可以将 log level 选项设置为 1：

```
io.configure('development', function() {
io.set('log level', 1);
});
```

有一些选项不仅仅只是通过 configuration 方法配置后就可以工作的，它们还需要其他一些条件，例如 store 配置项（它来决定客户端数据的保存位置）。

一般情况下，除过 log level 和 transports 选项外，我们无需再修改 Socket.IO 其他配置项的默认值了。

13.4 Chat：WebSockets 版本的"Hello, World"

每种技术都有其自己版本的"Hello, World"（通常指人们在学习某项技术时的首次应用），而对于 WebSockets 和 Socket.IO 来说，它是一个聊天程序。在 Socket.IO 的 GitHub 站点上还提供了一些聊天客户端（就像一个 IRC，互联网聊天客户端），通过搜索"Socket.IO and chat"就可以找到几个很好的例子。

在本节中，我将演示一个很简单的聊天客户端代码。它没有太多功能，只使用了 Socket.IO（并且客户端或服务端均没有使用其他库），但它演示了如何使用 Socket.IO 漂亮而优雅地实现一个对于其他技术来说比较难实现的应用程序。

该应用程序还使用了 Socket.IO 提供的另一些方法。在前面的例子中，我们使用 Socket.IO 的 send 和 emit 方法在客户端和服务器之间发送消息进行通信。但这类通信被限制在了套接字上，无论有多少人同时连接在服务器上，也只有消息中指定的接收方才能看到消息内容。

为了将消息广播给每个已经连接到服务端的用户，你可以在 Socket.IO 框架对象上直接调用 emit 方法：

```
io.sockets.emit();
```

这样所有与服务端建立了连接的客户都能收到消息。你也可以通过调用 socket 对象上的 broadcast.emit 方法将消息广播给除该 socket 以外的所有套接字，当然使用该 socket 对象的用户是无法接收到这条广播消息的：

```
socket.broadcast.emit();
```

当一个新的客户端连接到服务端后，聊天程序会提示并要求用户输入一个名称，然后将其广播给已经与服务端建立连接的其他客户端，表示此人现在已经进入了聊天室。客户端应用程序则为用户提供一个文本输入框以及用于发送消息的按钮，还提供了一块地方用于显示来自其他用户的新消息。示例 13-4 是客户端应用程序的源代码。

示例 13-4　Chat 程序客户端

```
 <!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>bi-directional communication</title>
  <script src="/socket.io/socket.io.js"></script>
  <script>
     var socket = io.connect('http://localhost:8124');
     socket.on('connect', function() {
        socket.emit('addme', prompt('Who are you?'));
     });

     socket.on('chat',function(username, data) {
       var p = document.createElement('p');
       p.innerHTML = username + ':  ' + data;
       document.getElementById('output').appendChild(p);
     });
     window.addEventListener('load',function() {
        document.getElementById('sendtext').addEventListener('click',
            function() {
               var text = document.getElementById('data').value;
               socket.emit('sendchat', text);
            }, false);
      }, false);
```

```
</script>
</head>
<body>
<div id="output"></div>
<div id="send">
  <input type="text" id="data" size="100" /><br />
  <input type="button" id="sendtext" value="Send Text" />
</div>
</body>
</html>
```

相比前面的例子，除了增加基本的 JavaScript 代码来捕捉按钮单击事件以及得到用户名字外，其他的功能没有太大的不同。

在服务端，新接入者的用户名被作为数据附加到了套接字上。服务端会对请求做出直接回应，然后将接入者的名字广播给聊天室里面的其他参与者。当服务端接收到任何新的聊天消息时，会自动为这条消息添加用户名，以便让每个人都能看到是谁发送的这条信息。最后，当客户端连接从聊天室断开时，另一条消息会被广播到当前依然与服务端保持连接的所有用户，以表明这个人不再参与聊天。示例 13-5 是服务端应用程序的完整源代码。

示例 13-5 Chat 程序服务端

```
var app = require('http').createServer(handler)
  , io = require('socket.io').listen(app)
  , fs = require('fs');

app.listen(8124);

function handler (req, res) {
  fs.readFile(__dirname + '/chat.html',
  function (err, data) {
    if (err) {
      res.writeHead(500);
      return res.end('Error loading chat.html');
    }
    res.writeHead(200);
    res.end(data);
  });
}

io.sockets.on('connection', function (socket) {

   socket.on('addme',function(username) {
      socket.username = username;
      socket.emit('chat', 'SERVER', 'You have connected');
      socket.broadcast.emit('chat', 'SERVER', username + ' is on deck');
   });

   socket.on('sendchat', function(data) {
      io.sockets.emit('chat', socket.username, data);
   });
```

```
    socket.on('disconnect', function() {
       io.sockets.emit('chat', 'SERVER', socket.username + ' has left the building');
    });

 });
```

我使用四个不同的浏览器（Chrome、FireFox、Opera 和 IE）来测试应用程序，得到了图 13-1 所示的运行结果。

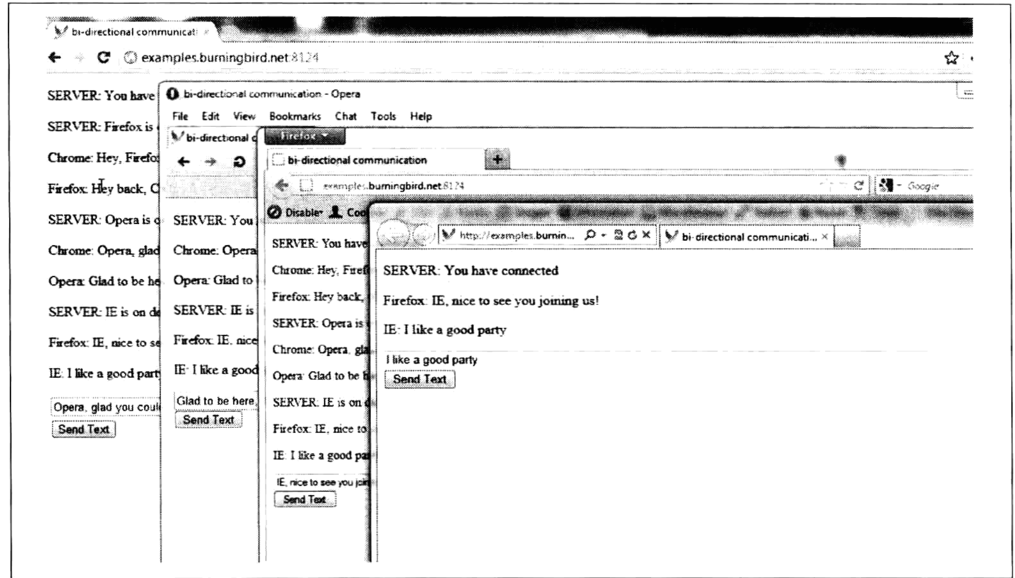


图 13-1　在不同浏览器中测试基于 Socket.IO 的聊天程序

我们还可以再改进下这个聊天程序，例如为其添加用户列表，以便新加入聊天室的用户可以看到当前已经有谁在线了。这可能需要一个全局数组，因为它与用户名不同，需要被所有客户端共享并访问。不过，我打算将这个功能实现作为课后练习作业留给你。

13.5　在 Express 中使用 Socket.IO

之前的示例程序一直在使用 Node 提供的 HTTP Web 服务。其时，我们还可以同时使用 Express 和 Socket.IO。关键是要记住 Socket.IO 必须要先于 Express 侦听到用户请求并处理它。

我修改了上一节的聊天程序，让它使用 Express 来处理所有 Web 服务请求，得到了

示例 13-6 的代码。有关 Socket.IO 和 Express 整合的关键代码已经用粗体标识了出来。另外，我没有对示例 13-5 的代码做任何修改。

示例 13-6　Chat 程序的 Express 实现

```
var express = require('express'),
    sio = require('socket.io'),
    http = require('http'),
    app = express();

var server = http.createServer(app);

app.configure(function () {
  app.use(express.static(__dirname + '/public'));
  app.use(app.router);
});

app.get('/', function (req, res) {
  res.send('hello');
});

var io = sio.listen(server);

server.listen(8124);

io.sockets.on('connection', function (socket) {

   socket.on('addme',function(username) {
      socket.username = username;
      socket.emit('chat', 'SERVER', 'You have connected');
      socket.broadcast.emit('chat', 'SERVER', username + ' is on deck');
   });

   socket.on('sendchat', function(data) {
      io.sockets.emit('chat', socket.username, data);
   });

   socket.on('disconnect', function() {
      io.sockets.emit('chat', 'SERVER', socket.username + ' has left the building');
   });

});
```

Express 对象被传递给了 HTTP server 对象，而 HTTP server 对象又被传递给了 Socket.IO 对象。三个模块协同工作确保了所有用户请求（包括 Web 请求或是聊天通信数据）都能被正确处理。

由于 chat 应用客户端是一个静态页面，所以引入模板也是比较容易的事。不过需要注意的是负责与服务端通信的脚本块需要被包含在模板文件中，同时还要保证引用 Socket.IO 库的连接也存在于模板文件中。

第 14 章

Node 应用程序的测试和调试

在之前的章节中，我们唯一使用过的调试手段就是在例子中打印信息到控制台。对于功能简单的小型程序的开发过程来说这种方法暂且可行。但是当程序规模逐渐变大，功能逐渐复杂，你需要另外一种更有效的调试工具。

同样，你可能也需要更正规一点的测试，比如使用一些创建测试的工具，也可用其他人员在自己的环境中测试你所编写的模块或者程序。

14.1 调试

坦白地说，console.log 一直是我进行调试时的一个选择，但是当程序的规模和复杂度都快速增长时，console.log 就显得无能为力了。一旦你的程序具有一定规模和复杂的功能，你就该考虑使用更复杂的调试工具。接下来我们会介绍一些可用的调试工具以供选择。

14.1.1 Node.js Debugger

V8 引擎内建的 debugger 可用于调试 Node 程序。Node 同时提供了客户端简化使用。我们在代码中希望打断点的地方添加 debugger 语句：

```
// 创建 proxy，监听所有请求
httpProxy.createServer(function (req, res, proxy) {
   debugger;
   if (req.url.match(/^\/node\//))
       proxy.proxyRequest(req, res, {
       host:'localhost',
        port:8000
        });
   else
       proxy.proxyRequest(req, res, {
        host:'localhost',
```

```
        port:8124
        });
}).listen(9000);
```

接下来以 debug 模式运行程序：

```
node debug debugger.js
```

在 debug 模式中，程序在一开始是暂停的。输入 cont 或者缩写 c 可以运行到第一个断点，程序会停在第一个断点处，等待用户的输入（比如 Web request）：

```
<debugger listening on port 5858
connecting... ok
break in app2.js:1
  1 var connect = require('connect'),
  2 http = require('http'),
  3 fs = require('fs'),
debug>cont (--> note it is just waiting at this point for a Web request)
break in app2.js:11
  9 httpProxy.createServer(function(req,res,proxy) {
 10
 11 debugger;
 12 if (req.url.match(/^\/node\//))
 13 proxy.proxyRequest(req, res, {
debug>
```

在断点处你有几个选择。输入 n（next）命令跳到下一行代码，o（out）跳出一个函数。在以下代码中，debugger 停在断点处，接下来的几行会通过 next 命令跳过直到 13 行，调用函数。我使用 step 进入函数体内部。然后可以用 next 遍历函数内部代码，再使用 out 返回程序：

```
debug>cont
break in app2.js:11
  9 httpProxy.createServer(function(req,res,proxy) {
 10
 11    debugger;
 12    if (req.url.match(/^\/node\//))
 13        proxy.proxyRequest(req,res, {
debug> next
break in app2.js:12
 10
 11    debugger;
 12    if (req.url.match(/^\node\//))
 13        proxy.proxyRequest(req,res, {
 14          host: 'localhost',
debug> next
break in app2.js:13
 11    debugger;
```

```
 12    if (req.url.match(/^\/node\//))
 13       proxy.proxyRequest(req, res, {
 14         host: 'localhost',
 15         port: 8000
debug> step
break in /home/examples/public_html/node/node_modules/http-proxy/lib/
node-http-proxy/routing-proxy.js:144
142 //
143 RoutingProxy.prototype.proxyRequest = function (req, res, options) {
144   options = options || {};
145
146   //
debug> next
break in /home/examples/public_html/node/node_modules/http-proxy/lib/
node-http-proxy/routing-proxy.js:152
150   // proxyRequest 参数
151   //
152   if (this.proxyTable&& !options.host) {
153   location = this.proxyTable.getProxyLocation(req);
154
debug> out
break in app2.js:22
20          port: 8124
21        });
22 }).listen(9000);
23
24  // 为动态资源的请求添加路由
```

你还可以添加新的断点，用setBreakpont(sb)设置断点在当前行，也可以在命名函数或者脚本文件的第一行：

```
break in app2.js:22
 20          port: 8124
 21        });
 22 }).listen(9000);
 23
 24 //为动态资源的请求添加路由
debug>sb()
 17   else
 18      proxy.proxyRequest(req,res, {
 19        host: 'localhost',
 20        port: 8124
 21      });
*22}).listen(9000);
 23
 24//为动态资源的请求添加路由
 25 crossroads.addRoute('/node/{id}/', function(id) {
 26    debugger;
 27 });
```

使用 clearBreakpoint（cb）清除断点。

除了使用 REPL 来查看变量值，你可以在 watch 列表中添加表达式或者列出当前查看的内容：

```
break in app2.js:11
 9 httpProxy.createServer(function(req,res,proxy) {
10
11    debugger;
12    if (req.url.match(/^\/node\//))
13        proxy.proxyRequest(req, res, {
debug>repl
Press Ctrl + C to leave debug repl>
req.url
'/node/174'
debug>
```

backtrace 命令可用于打印当前执行栈的回溯（backtrace）：

```
debug>backtracke
#0  app2.js:22:1
#1exports.createServer.handler node-http-proxy.js:174:39
```

任何时候都可以使用 help 来查看可供使用的命令：

```
debug> help
Commands: run (r), cont (c), next (n), step (s), out (o), backtrace (bt),
setBreakpoint (sb), clearBreakpoint (cb), watch, unwatch, watchers, repl,
restart, kill, list, scripts, breakpoints, version
```

虽然内建的调试工具很有用，但是有时候你需要的不仅仅如此。你还有其他选择，比如通过命令行参数--debug 直接访问 V8 调试工具：

```
node --debug app.js
```

这建立了一个到 debugger 的 TCP 连接，你可以在命令行提示符中输入 V8 调试的命令。这是个很有意思的做法，但是要求对 V8 debugger 的工作原理有一定的理解（以及有哪些命令可以使用）。

另一种选择是通过 WebKit 浏览器进行调试，通过类似 Node Inspector 的程序，将在下一节中介绍。

14.1.2 使用 Node Inspector 的客户端调试

Node Inspector 在开始调试前需要一些设置，但是磨刀不误砍柴工嘛。

首先，全局安装 Node Inspector:

```
npm install-g node-inspector
```

为了使用该功能，首先需要用 V8 debugger 参数运行程序：

```
node--debug app.js
```

接下来你需要运行 Node Inspector，前端或者后端运行都可以：

```
node-inspector
```

程序启动之后，你会看到以下信息：

```
node-inspector
   info - socket.io started
visit http://0.0.0.0:8080/debug?port=5858 to start debugging
```

使用基于 WebKit 的浏览器（Safari 或者 Chrome）打开调试页面。我的程序示例运行在本地，所以访问以下 URL：

```
http://examples.burningbird.net:8080/debug?port=5858
```

在浏览器中，打开客户端的调试工具（开发工具套件的一部分），停在第一个断点处。现在你可以使用你可能很熟悉的客户端 JavaScript 开发工具，比如执行几行代码或者查看属性的值，如图 14-1 所示。

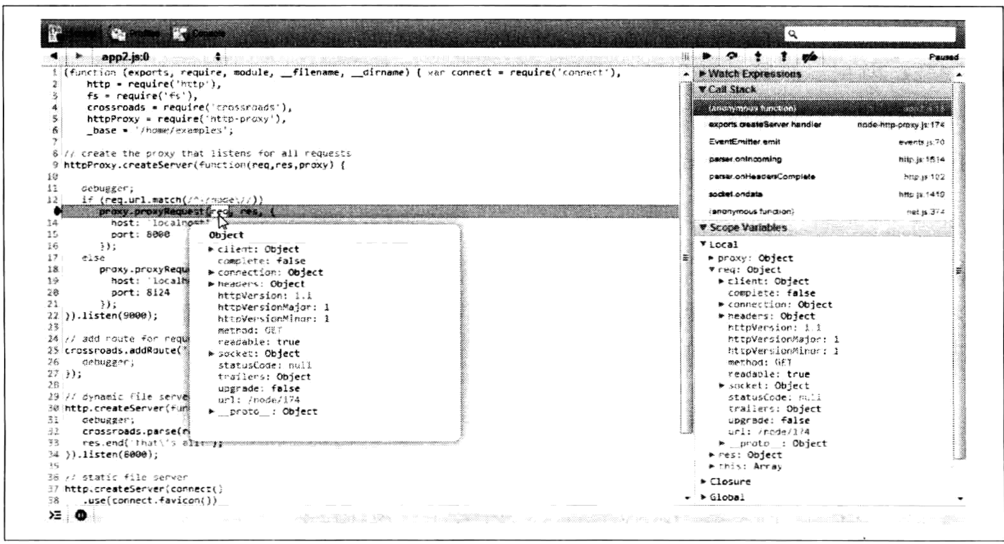


图 14-1　Node 程序作为服务器，Chrome 中运行 Node Inspector

Node Inspector 是截止目前为止最好的调试服务器程序的工具。当然可以使用命令

行进行调试，但是 Node Inspector 可以一次看到所有代码，并可以使用我们熟悉的工具套件，这弥补了对 Node Inspector 所做的设置。

提示

如果你的 Node 程序由云服务提供，该服务一般也会提供特有的开发工具，包括调试工具。

14.2 单元测试（Unit Testing）

单元测试是一种将特定组件从整体程序中分离出来进行测试的方式。Node 模块中 tests 子目录下的很多测试都是单元测试，而 Node 安装目录中的 test 子目录下全都是单元测试。

你可以使用 npm 运行模块的测试，在模块的子目录中输入：

```
npm test
```

这条命令会运行模块的测试脚本（如果有的话）。当我在运行 node-redis 的测试脚本时，输出的结果显示了成功的测试用例，显示如下：

```
Connected to 127.0.0.1:6379, Redis server version 2.4.11
Using reply parser hiredis
 - flushdb: 1 ms
- multi_1: 3 ms
- multi_2: 9 ms
- multi_3: 2 ms
- multi_4: 1 ms
- multi_5: 0 ms
- multi_6: 7 ms
- eval_1:Skipping EVAL_1 because server version isn't new enough.
0 ms
- watch_multi: 0 ms
```

这些测试很多都是用 Assert 模块创建的，我们会在下一节介绍。

14.2.1 Assert 与单元测试

Assertion tests（断言测试）计算表达式的值，计算的结果只可能是 true 或者 false。如果你希望测试一个函数调用的返回结果，首先你可能需要测试的是调用返回的是一个数组（第一个断言）。如果数组需要为一个固定长度，你可以按长度作条件测试（第二个断言），以此类推。有一个 Node 内建的模块可以完成这一系列的断言测试：Assert。

你可以在程序中用 require 引入 Assert 模块：

```
var assert = require('asset');
```

我们可以通过了解现有的模块如何使用 Assert 来学习。以下这个 test.js 中的测试是 node-redis 安装目录中的一个例子：

```
    var name = "FLUSHDB";
    client.select(test_db_num, require_string("OK", name));
```

该测试使用了一个 require_string 函数。require_string 返回一个函数，使用 Assert 模块的 assert.equal、assert.stringEqual 方法：

```
function require_string(str, label) {
    return function (err, results) {
        assert.strictEqual(null, err, "result sent back unexpected error: " + err);
        assert.equal(str, results, label + " " + str + " does not match " + results);
        return true;
    };
}
```

第一个测试 assert.stringEqual，如果 Redis 测试返回的 err 对象不为空则失败。第二个测试使用 assert.equal，如果返回结果与预期结果不相等则失败。该函数只有当两个测试都成功的情况下才能执行到 return true 语句。

真正测试的内容是 Redis select 命令是否成功。如果发生了错误，则输出错误信息。如果选择的结果不是预期得到的结果，也会输出错误信息，包括标识出测试哪里失败了。

Node 程序在自己的模块单元测试中也使用 Assert 模块。例如，一个名为 test-util.js 的测试程序，用于测试 Utilities 模块。以下代码测试了 isArray 方法：

```
// isArray
assert.equal(true, util.isArray([]));
assert.equal(true, util.isArray(Array()));
assert.equal(true, util.isArray(new Array()));
assert.equal(true, util.isArray(new Array(5)));
assert.equal(true, util.isArray(new Array('with', 'some', 'entries')));
assert.equal(true, util.isArray(context('Array')()));
assert.equal(false, util.isArray({}));
assert.equal(false, util.isArray({ push: function () {} }));
assert.equal(false, util.isArray(/regexp/));
assert.equal(false, util.isArray(new Error));
assert.equal(false, util.isArray(Object.create(Array.prototype)));
```

assert.equal 和 assert.strictEqual 都必须包含两个参数：一个是期待得到的返回结果，一个是表达式计算的结果。在之前的 Redis 测试中，assert.strictEqual 期待结果为 null 或者 err 对象。如果期望不匹配，则测试失败。Node 源代码中的测试 assert.equalisArray，如果表达式计算结果为 true，期望得到的结果也是 true，assert.equal 方法成功，没有任何输出——成功的结果是沉默的。

但是如果表达式的计算结果不是所期待的结果，assert.equal 方法会抛出异常。如果我对 isArray 测试中的第一个参数进行修改：

```
assert.equal(false, util.isArray([]));
```

结果为：

```
node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
              ^
AssertionError: false == true
    at Object.<anonymous> (/home/examples/public_html/node/chap14/testassert.
js:5:8)
    atModule._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:31)
    at Function._load (module.js:308:12)
    at Array.0 (module.js:479:10)
    at EventEmitter._tickCallback (node.js:192:40)
```

assert.equal 和 assert.strictEqual 方法还提供了第三个可选的参数，在失败时显示错误信息而不是默认的失败处理方式：

```
assert.equal(false, util.isArray([]), 'Test 1Ab failed');
```

这在同一个测试脚本中运行多个测试时可以有效地指出是哪一个测试失败了。在 node-redis 测试代码中可以看到 message 的使用：

```
assert.equal(str, results, label + " " + str + " does not match " + results);
```

当测试失败捕获异常时会显示这个 message 作为错误信息。

以下这些 Assert 模块方法参数内容都各自不同，但是都接收与之前方法同样的三个参数：

assert.equal

如果表达式结果与给出结果不相等则失败。

assert.strictEqual

如果表达式结果和给出结果不严格相等则失败。

assert.notEqual

如果表达式结果和给出结果相等则失败。

assert.notStrictEqual

如果表达式结果和给出结果严格相等则失败。

assert.deepEqual

如果表达式结果与给出结果不相等则失败。

assert.notDeepEqual

如果表达式结果和给出结果相等则失败。

最后两个方法，assert.deepEqual 和 assert.notDeepEqual 用于处理复杂对象，比如数组或者 object。下面这个使用 assert.deepEqual 测试成功：

```
assert.deepEqual([1,2,3],[1,2,3]);
```

但是如果使用 assert.equal 则会失败。

其余的 assert 方法接收不同参数。调用 assert 方法，传入值和一个信息，等同于调用 assert.isEqual，true 作为第一个参数，其余两个参数为表达式、信息。以下代码：

```
var val = 3;
assert(val == 3, 'Equal');
```

等价于：

```
assert.equal(true, val == 3, 'Equal');
```

另一种变形且等价的方法为 assert.ok：

```
assert.ok(val == 3, 'Equal');
```

assert.fail 方法抛出异常。该方法接收四个参数：值、表达式、信息和一个操作符，操作符用于在抛出异常时分隔值和表达式。在以下代码段中：

```
try {
  var val = 3;
```

```
  assert.fail(3, 4, 'Fails Not Equal', '==');
} catch (e) {
  console.log(e);
}
```

控制台信息为：

```
{ name: 'AssertionError',
  message: 'Fails Not Equal',
  actual:3,
  expected:4,
  operator:'=='}
```

assert.ifError 函数接收一个值，当该值不为 false 时则抛出异常。正如 Node 文档中描述的一样，这是一个好的测试，对作为回调函数第一个参数的 error 对象进行测试：

```
assert.ifError(err); //只有 err 值为 true 时抛出异常
```

最后一个 assert 方法为 assert.throws 和 assert.doesNotThrow。第一个方法期待有异常抛出，第二个则相反。这两个方法参数相同，第一个参数为一个代码区块，第二三个参数为可选的 error 对象和 message。Error 对象可以是一个构造函数，正则表达式或者验证函数。在以下代码段中，由于第二个参数正则表达式与错误信息不匹配所以输出错误信息：

```
assert.throws(
  function () {
    throw new Error("Wrong value");
  },
  /something/
  )
} catch(e) {
  console.log(e.message);
}
```

你可以用 Assert 模块创建更复杂的测试。使用该模块的一个主要限制在于你需要做很多对测试的包装，以防整个测试脚本不会因为一个测试的失败而失败。这也是为什么我们可以使用更高级的单元测试架构，比如 Nodeunit（下一节介绍）更派得上用处。

14.2.2 Nodeunit 与单元测试

Nodeunit 提供了编写多个测试脚本的方法。编写完成后，测试按顺序进行，测试运行结果会报告在同一个文件中。在使用 Nodeunit 之前可以用 npm 全局安装：

```
npm install nodeunit-g
```

Nodeunit 提供了一个简单的方法运行一系列测试而不需要用 try/catch 进行包裹。Nodeunit 支持所有的 Assert 模块测试，并且提供了自己的方法来控制测试。测试按照测试用例（test case）分组，每个测试用例做为测试脚本的一个对象方法。每个测试用例有一个控制（control）对象，一般命名为 test。测试用例中第一个调用的方法是 test 对象的 expect 方法，告诉 Nodeunit 该测试用例中有多少个测试。而测试用例中调用的最后一个方法是 test 对象的 done 方法，告诉 Nodeunit 测试用例运行完毕。这两者之间的所有东西都是实际的单元测试：

```
module.exports = {
    'Test 1': function (test) {
       test.expect(3); // 三个测试
       ... // 测试
       test.done();
    },
    'Test 2': function (test) {
       test.expect(1); // 只有一个测试
       ... // 测试
       test.done();
    }
};
```

运行测试，输入 nodeunit，之后输出测试脚本的名字：

```
nodeunit thetest.js
```

示例 14-1 是一个小但完整的测试脚本，包含六个测试，由两个测试单元组成，Test1 和 Test2。第一个测试单元运行四个独立测试，第二个测试单元包含两个。调用 expect 方法反映了单元中包含多少个测试。

示例 14-1　Nodeunit 测试脚本，两个测试单元，总测试数为 6 个

```
var util = require('util');

module.exports = {
    'Test 1': function (test) {
        test.expect(4);
        test.equal(true, util.isArray([]));
        test.equal(true, util.isArray(new Array(3)));
        test.equal(true, util.isArray([1, 2, 3]));
        test.notEqual(true, (1 > 2));
        test.done();
    },
    'Test 2': function (test) {
        test.expect(2);
        test.deepEqual([1, 2, 3], [1, 2, 3]);
```

```
            test.ok('str' === 'str', 'equal');
            test.done();
        }
};
```

Nodeunit 运行示例 14-1 测试脚本的结果为：

```
example.js
✓ Test 1
✓ Test 2
OK: 6 assertions (3ms)
```

每个测试之前的标记符号表示测试结果：√对号表示成功，×叉号表示失败。在这个脚本中测试没有失败，所以没有错误信息或者堆栈信息输出。

提示

对于 CoffeeScript 粉丝来说有个好消息，最新版的 Nodeunit 支持 CoffeeScript 程序。

14.2.3 其他测试框架

除了前一章讲到的 Nodeunit，对 Node 开发人员来说还有其他几种可用的测试框架。这些框架有的相对于其他一些较容易上手，但是各有各的优缺点。下一节，我会介绍其他三个框架：Mocha、Jasmine 和 Vows。

Mocha

使用 npm 安装 Mocha：

```
npm install mocha -g
```

Mocha 被看作是另一个受欢迎的测试框架 Espresso 的后续版本。

Mocha 既可以用于浏览器，也可以用于 Node 程序中。Mocha 允许通过 done 方法进行异步测试，尽管这个方法在同步测试中会被忽略掉。Mocha 可以兼容任何 assertion 的代码库。

下面是一个 Mocha 测试的例子，使用了 should.js 库：

```
should = require('should')
describe('MyTest', function () {
  describe('First', function () {
    it('sample test', function () {
      "Hello".should.equal("Hello");
    });
```

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

在运行该测试之前需要安装 should.js：

```
npm install should
```

然后运行测试：

```
mocha testcase.js
```

测试应该通过：

```
1 test complete (2ms)
```

Jasmine

Jasmine 是一个行为驱动开发的框架，可以用于很多不同的技术，包括 Node 中的 node-jasmine 模块。可以用 npm 安装 node-jasmine：

```
Npm install jasmine-node -g
```

提示

注意一下模块名：jasmine-node，而不同于你在本书中看到的一般的 node 模块命名格式 node-模块名（或者直接是模块名）。

jasmine-node GitHub 目录下 specs 子目录中包含示例。和其他大多数测试框架一样，Jasmine Node 模块也接收 done 方法作为回调函数允许异步测试。

使用 jasmine-node 模块时有一些对环境的要求。首先，测试必须在 specs 子目录中。Jasmine-node 模块是一个命令行程序，你可以指定根目录，但是测试最好在 specs 目录中。

然后，测试名需要符合固定的格式。如果测试是由 JavaScript 编写，测试文件名则必须以.spec.js 结尾。如果测试由 CoffeeScript 编写，文件名则需以.spec.coffee 结尾。你可以在 specs 目录中添加其他子目录。当运行 jasmine-node 时，会运行 specs 目录下所有测试。

为了具体说明，我创建了一个简单的测试脚本，使用 Zombie（稍后介绍）对 Web 服务器发起请求并访问页面内容。文件命名为 tst.spec.js，放置在我的开发环境中的 specs 目录下：

```
var zombie = require('zombie');
describe('jasmine-node', function () {
```

```
      it("should respond with Hello, World!", function (done) {
        zombie.visit("http://examples.burningbird.net:8124",
                                                          function (error, browser,
status) {
          expect(browser.text()).toEqual("Hello, World!\n");
        done();
       });
      });
   });
```

Web 服务器是第 1 章中的内容，所做的就是返回“Hello, World!”信息。注意一下换行符，如果没有换行符测试会失败。

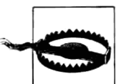
用以下命令运行测试：

```
jasmine-node --test-dir /home/examples/public_html/node
```

运行结果如下：

```
Finished in 0.133 seconds
1 test, 1 assertion, 0 failures
```

测试成功。



警告

Jasmine 使用 path.existsSync，这已经在 Node 0.8 的 js.existsSync 弃用，希望不久就会有修复。

如果脚本是由 CoffeeScript 编写，运行命令时会添加-coffee 参数：

```
jasmine-node --test-dir /home/examples/public_html/node -coffee
```

Vows

Vows 是另一个行为驱动开发的测试框架，相比其他来说，Vows 有一个优点：更全面的文档。测试由测试套件组成，而测试套件又由一系列分批的可执行的测试组成。一个批次由一个或多个语境（context）组成，并行执行，并且每个语境都包含一个主题（topic），最终得到可执行的代码。在代码中的测试称为 vow。Vows 与其他测试框架不同的优点在于更清楚地区分开了测试的内容（主题）和测试本身（vow）。

我知道这里的用法有一些特别，所以我们先来看个简单的例子以便更好地理解 Vows 测试框架是如何工作的。首先，需要安装 Vows：

```
npm install vows
```

为了尝试 Vows，我使用本书之前创造的 simple circle 模块，修改其精度：

```
var PI = Math.PI;

exports.area = function (r) {
  return (PI * r * r).toFixed(4);
};
exports.circumference = function (r) {
  return (2 * PI * r).toFixed(4);
};
```

我需要修改结果的精度，因为需要对结果在 Vows 程序中进行相等的测试。

在 Vows 测试程序中，circle 对象就是主题，area 和 circumference 方法就是 vows。这两个都封装为 Vows 的语境（context）。测试套件为整个测试程序，batch 为测试实例（circle 和两个方法）。

示例 14-2 Vows 测试程序，一个 batch，一个 context，一个 topic，两个 vows

```
var vows = require('vows'),
    assert = require('assert');

var circle = require('./circle');

var suite = vows.describe('Test Circle');

suite.addBatch({
    'An instance of Circle': {
         topic: circle,
         'should be able to calculate circumference': function (topic) {
         assert.equal(topic.circumference(3.0), 18.8496);
         },
         'should be able to calculate area': function (topic) {
             assert.equal(topic.area(3.0), 28.2743);
         }
    }
}).run();
```

因为在 addBatch 方法末尾添加的 run 方法，所以运行 Node 程序就会运行该测试：

```
node example2.js
```

结果中应该是两个成功的测试：

```
·· OK » 2 honored (0.003s)
```

topic 总是一个异步方法或者数值。不用 circle 作为 topic，可以借助函数闭包直接将 object 方法作为 topic：

```
var vows = require('vows'),
    assert = require('assert');

var circle = require('./circle');
```

```
var suite = vows.describe('Test Circle');

suite.addBatch({
    'Testing Circle Circumference': {
        topic: function () { return circle.circumference; },
        'should be able to calculate circumference': function (topic) {
            assert.equal(topic(3.0), 18.8496);
        },
    },
    'Testing Circle Area': {
        topic: function () { return circle.area; },
        'should be able to calculate area': function (topic) {
            assert.equal(topic(3.0), 28.2743);
        }
    }
}).run();
```

在上述代码中，每个 context 都是一个有 title 的对象：Testing Circle Circumference 和 Testing Circle Area。每一个 context 中都有一个 topic 和一个 vow。

你可以创建多个 batch，每个 batch 包含多个 context，而每个 context 可以依次包含多个 topics 和 vows。

14.3 验收测试

验收测试与单元测试不同，主要目的在于检查程序是否满足用户需求。单元测试用于确保程序强健（robust），而验收测试用于确保程序有效（useful）。

验收测试一般通过实际用户设计和实现的预定义脚本完成。验收测试可以自动化完成——通过一些脚本，但是脚本由工具自动运行而不是人力完成。这些工具并不能完全满足验收测试的所有需求，因为工具没办法从主观角度进行度量（“这个网页太难用了！”），也不能精确定位那些由用户操作驱动出来但是很难发现的 bug，但是可以保证满足了程序的需求。

14.3.1 Soda 和 Selenium 测试

如果你想要一个更高层面的测试，使用实际的浏览器而不是模拟器，并且愿意支付测试服务的费用，你可以了解一下 Selenium、Sauce Labs 和 Node 模块 Soda。

Selenium 满足了开发人员对自动化测试工具的需求，Selenium 由核心库、一个 Selenium remote control（RC）和 Selecnium 集成开发环境（IDE）组成。Selenium IDE 是一个 Fiefox 插件，RC 是一个 Java .jar 文件。Selenium 的第一个版本（Selenium1）是基于 JavaScript 的，这也造成了这个工具套件的问题：Selenium 拥有一切 JavaScript 所有的限制。另一

个自动化测试套件是 WebDriver，WebDriver 的出现主要目的在于解决 Selenium 的限制。Selenium2 的开发正在进行中，是 Selenium1 和 WebDriver 的综合版本。

Sauce Labs 为 Selenium1 测试提供宿主。Sauce Labs 为应用程序提供了多环境中的多浏览器的测试，比如 Linux 下的 Opera、Windows7 的 IE9。Sauce Labs 有两个主要的限制：对 MAC OS X 的支持和没有移动平台的测试环境。但是，这确实是一个对应用程序的多浏览器支持的测试方法，比如 IE，如果只有一台机器的时候很难测试。

Sauce Labs 提供了多种订阅计划，包括一个基本的免费的订阅计划用于尝试该服务。基本方案允许两个并发的用户，提供了每月 200 分钟的 OnDemand（处理器资源按需供应）和 45 分钟 Scout——对开发人员来说足够进行尝试。该网站目标人群是 Ruby 开发人员，但是你可以使用 Node 模块 Soda。

Soda 是对 Selenium 测试的 Node 包装。模块文档中使用 Soda 的范例代码为：

```
var soda = require('soda');

var browser = soda.createClient({
host: 'localhost'
     ,port: 4444,
     url: 'http://www.google.com',
     browser: 'firefox'
});

browser.on('command', function (cmd, args) {
  console.log(' \x1b[33m%s\x1b[0m: %s', cmd, args.join(', ')];
});

browser.
chain
  .session()
  .open('/')
  .type('q', 'Hello World')
  .end(function (err) {
    browser.testComplete(function () {
      console.log('done');
      if (err) throw err;
    });
  });
```

这段代码看起来很直观。首先，你需要创建一个浏览器对象，描述需要打开哪个浏览器，主机名和端口号，以及需要访问的网站名。新建一个浏览器 session，加载页面（'/'），在 id 为 q 的输入区域输入内容。结束后输出 done 到 console.log，并抛出过程中产生的任何异常。

你需要确保 Java 已安装才能运行 Soda 程序。然后复制 Selenium RC Java .jar 文件

到你的系统并运行：

```
java-jar selenium.jar
```

Selenium 中定义了使用 Firefox 浏览器，所以需要安装 Firefox。我的 Linux 系统中并没有 Firefox，但是 Windows 系统的笔记本很容易运行该程序。看到 Selenium RC 运行过程中浏览器窗口弹出并消失是个很有趣但是有点烦人的过程。

另一种实现是使用 Sauce Labs 作为远程测试环境，对给定测试定义使用哪一个浏览器。你需要先创建一个账户，然后找到你的用户名和 API key。用户名会显示在最上面的工具条上，你可以在 Account 标签下点击"View my API Key"链接找到 API key。在这里你还可以看到你剩余的 OnDemand 和 Scout 时间（我们创建的测试程序会使用 OnDemand 时间）。

为了尝试远程测试，我创建了一个简单的测试，测试我们在 15 章中创建的登陆表单。登录表单有两个文本框和两个按钮。文本框的内容为用户名和密码，一个按钮为 Submit（提交按钮）。测试脚本测试失败而非成功，所以测试场景应该为：

1. 访问网站程序（http://examples.burningbird.net:3000）;
2. 打开登录页面（/login）;
3. 在用户名处输入 Sally；
4. 在密码处输入 badpassword；
5. 网页会显示"Invalid Password"。

这些步骤组成示例 14-3 的测试。

示例 14-3　测试登录表单密码错误

```
var soda = require('soda');

var browser = soda.createSauceClient({
    'url': 'http://examples.burningbird.net:3000/',
    'username': 'your username',
    'access-key': 'your access key',
    'os': 'Linux',
    'browser': 'firefox',
    'browser-version': '3.',
    'max-duration': 300 // 5 分钟
});

// 完成后的日志信息
browser.on('command', function (cmd, args) {
```

```
  console.log(' \x1b[33m%s\x1b[0m: %s', cmd, args.join(', ')];
  });

  browser
    .chain
    .session()
    .setTimeout(8000)
    .open('/login')
    .waitForPageToLoad(5000)
    .type('username', 'Sally')
    .type('password', 'badpassword')
    .clickAndWait('//input[@value="Submit"]')
    .assertTextPresent('Invalid password')
    .end(function (err) {
      browser.setContext('sauce:job-info={"passed": ' + (err === null) +
'}', function () {
        browser.testComplete(function () {
          console.log(browser.jobUrl);
          console.log(browser.videoUrl);
          console.log(browser.logUrl);
          if (err) throw err;
        });
      });
    });
```

在测试程序中，浏览器对象根据给定的浏览器、版本以及操作系统创建，本例中是Linux上的Firefox 3.x。注意到浏览器的客户端也有所不同：soda.createSauceClient，而不是soda.createClient。在浏览器对象中，我严格限制了测试时间不超过五分钟。访问的网站为http://examples.burningbird.net:3000。我们刚刚讲过了如何获取用户名和API key。

每一个命令的问题都会被日志记录。我们希望有日志信息以便检查回应，查找失败和异常情况：

```
// 完成后的日志信息
browser.on('command', function (cmd, args) {
  console.log(' \x1b[33m%s\x1b[0m: %s', cmd, args.join(', '));
});
```

最后才是实际的测试。一般来说，测试需要嵌套在回调中（这是一个异步测试环境），但是Soda提供了链式的getter大大简化了添加任务的过程。第一个任务是创建新的session，然后添加测试脚本中的每个任务。最后，程序输出测试中job、log、video的URL。

运行程序的输出结果为：

```
setTimeout: 8000
open: /login
waitForPageToLoad: 5000
```

```
    type: username, Sally
    type: password, badpassword
    clickAndWait: //input[@value="Submit"]
    assertTextPresent: Invalid password
    setContext: sauce:job-info={"passed": true}
    testComplete:
    https://saucelabs.com/jobs/d709199180674dc68ec6338f8b86f5d6
   https://saucelabs.com/rest/shelleyjust/jobs/d709199180674dc68ec6338f8b
86f5d6/results/video.flv
   https://saucelabs.com/rest/shelleyjust/jobs/d709199180674dc68ec6338f8b
86f5d6/results/selenium-server.log
```

你可以直接访问结果，或者登录 Sauce Labs 查看你所有测试的结果，如图 14-2 所示。

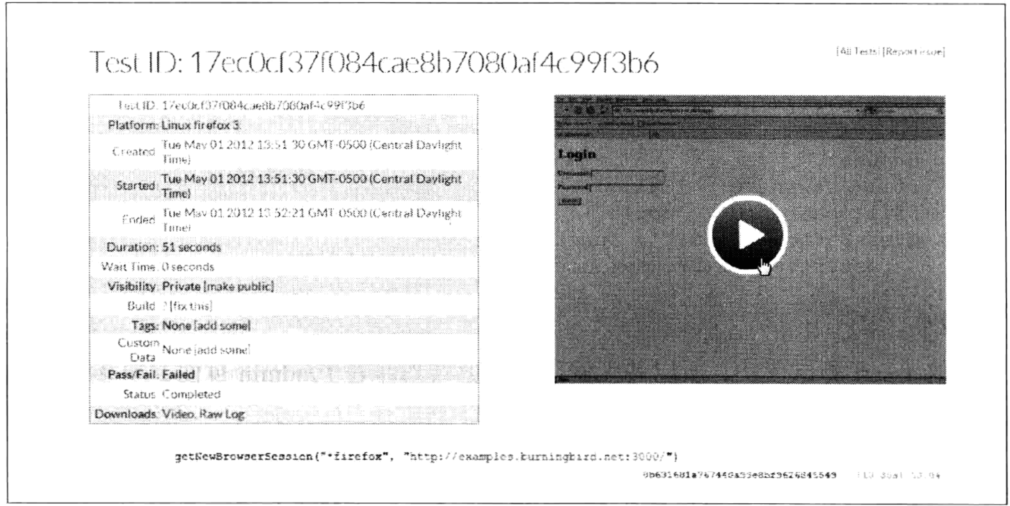


图 14-2　Sauce Labs Selenium core 运行 Soda 测试的结果

正如之前提到的，Soda 是 Selenium 的包装，所以模块中没有什么关于 Selenium 命令的文档。你需要在 Selenium 官网上搜索相关命令，并推测如何在 Soda 中使用。

提示

Selenium 官网：http://selenium.org/。

14.3.2　通过 Tobi 和 Zombie 模拟浏览器

你可以用 Node 模块模拟浏览器，而不使用其他特定的浏览器。Tobi 和 Zombie 都提供了这种功能。使用这些模块的最大的优点在于你可以在没有安装浏览器的环境里运行程序。在本节中，我会简要介绍如何使用 Zombie 完成验收测试。

首先，用 npm 安装 Zombie：

```
npm install zombie
```

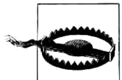
Zombie 与 Soda 类似，你可以创建浏览器并且模拟用户对浏览器的操作进行测试，并且还支持链式调用来解决嵌套的回调问题。

我将示例 14-3 中对登录表单的测试转换为 Zombie，只是这次使用了正确的密码来测试登录成功（用户会被重定向到/admin 页面）。示例 14-4 中为该验收测试的代码。

示例 14-4　用 Zombie 测试登录表单

```
var Browser = require('zombie');
var assert = require('assert');

var browser = new Browser();

browser.visit('http://examples.burningbird.net:3000/login',
   function () {
   browser.
   fill('username', 'Sally').
   fill('password', 'apple').
   pressButton('Submit', function () {
      assert.equal(browser.location.pathname, '/admin');
   });
});
```

最后的 assert 测试成功，所以并无输出。浏览器当前处于/admin 页面，当登录成功后才转到这个页面，显示测试成功。



警告

几个例子都依赖于 Node 模块 jsdom。但是这个模块在 Node 0.7.10 的不稳定版本中还有些问题，希望在 Node 0.8.x 中被修复。

14.4　性能测试：基准问题和负载测试

一个满足用户所有需求的程序如果性能太差也无法存活太久。我们需要对 Node 程序进行性能测试，特别是当我们对过程进行调整以提高性能的时候。我们不能只是调整程序，部署到产品环境供用户使用，然后由用户发现性能的问题。

性能测试包含基准（Benchmark）测试和负载测试。基准测试，也称为比较测试，会运行多个版本的程序来决定哪个版本更好。当你在调整程序来改进和扩展时基本测试则非常有用。你创建一个基准测试，根据变化运行，并分析结果。

负载测试，从另一个角度，对程序进行压力测试。你需要看到当有太多的并发用户

或者有太多对资源的请求时，在哪个点上你的程序会崩溃。你希望持续驱动你的程序直到失败，对于负载测试来说失败就是成功。

有现成的工具可以处理这两种性能测试，很受欢迎的一个就是 ApacheBench。ApacheBench 受欢迎的原因在于任何装有 Apache 的服务器上都可以使用，而且很少有服务器没有安装 Apache。ApacheBench 也是一个易于上手，强大而简单的测试工具。当我在犹豫创建一个静态数据库的可重用的链接还是创建一个链接使用后丢弃，就用到了 ApacheBench 运行测试。

ApacheBench 是基于网站的程序，意味着你需要提供 URL 而不是程序名。如果我们选择了 Node，或者其他应用可以运行程序（而不只是查询网站），还有另一个命令行/模块的混合工具:Nodeload。Nodeload 可以与 stats 模块交互，输出图形结果并提供实时的监控。Nodeload 也支持分布式的负载测试。

提示

在之后的几章节中，需要对 Redis 进行测试，如果你还没有读过第 9 章，你可能现在需要了解一下了。

14.4.1 ApacheBench 基准测试

ApacheBench 通常被称为 ab，之后我们都会使用 ab 指代 ApacheBench。ab 是一个命令行工具，允许我们指定程序运行的次数以及并发用户的数目。如果我们希望模拟 20 个用户访问一个网站总共 100 次，我们需要使用以下命令：

```
ab -n 100 -c 20 http://somewebsite.com/
```

最后一个斜杠很重要，因为 ab 需要接收一个完整的 URL，包括路径。

Ab 提供的输出内容很丰富。一个测试范例的输出（不包括工具定义）如下所示：

```
Concurrency Level:    10
Time taken for tests: 20.769 seconds
Complete requests:    15000
Failed  requests:     0
Write errors:         0
Total transferred:    915000  bytes
HTML transferred:     345000  bytes
Requests per second:  722.22  [#/sec]  (mean)
Time per request:     13.846  [ms]  (mean)
Time per request:     1.385   [ms]  (mean, across all concurrent requests)
Transfer rate:        43.02   [kbytes/sec]  received

Connection Times  (ms)
```

```
              Min    mean[+/-sd] median   max
Connect:       0     0    0.1    0       4
Processing:    1    14   15.7   12     283
Waiting:       1    14   15.7   12     283
Total:         1    14   15.7   12     283
Percentage of the requests serverd within a certain time (ms)
  50% 12
  66% 14
  75% 15
  80% 16
  90% 18
  95% 20
  98% 24
  99% 40
 100% 283 (longest request)
```

测试运行了 15 000 次，10 个并发用户。

我们最感兴趣的几行（粗体表示）表示每个测试花费多长时间，以及测试结束时的累积时间（按比例）。根据输出，每个请求的平均时间（第一个 Time per request）是 13.846 毫秒。这是平均每个用户需要等待多长时间来看到浏览器回应。第二个 Time per request 是吞吐量，可能没有第一个那么有用。

累积分布显示了在一个特定时段请求的处理情况。这意味着对每一个用户来说，我们可以期望网站的响应时间在 12 毫秒到 283 毫秒，大部分的响应处理少于 20 毫秒。

最后一个我们需要关注的值为 requests per second，本例中为 722.22。这个值某些程度上可以预测应用程序的规模，它告诉我们每秒钟程序可以处理的最大请求数，这是程序访问的上限。然后，你需要在不同的时候和不同的负载情况下运行测试，特别是当你在一个同时提供其他服务的系统上运行测试。

被测试的程序由一个监听请求的 Web 服务器组成。每个请求触发一次对 Redis 数据存储的查询。程序创建了一个对 Redis 数据存储的持久性连接，在整个 Node 程序的生命周期内维持该连接。测试程序如示例 14-5 所示。

示例 14-5　简单的 Redis 访问程序，用于测试稳定的 Redis 连接

```
var redis = require("redis"),
   http = require('http');

// 创建 Redis 客户端
var client = redis.createClient();

client.on('error', function (err) {
   console.log('Error ' + err);
```

```
});

// 设置数据库为 1
client.select(1);
var scoreServer = http.createServer();

// 监听收到的请求
scoreServer.on('request', function (req, res) {

   console.time('test');
req.addListener("end", function () {

   varobj = {
       member: 2366,
       game: 'debiggame',
       first_name: 'Sally',
       last_name: 'Smith',
       email: 'sally@smith.com',
       score: 50000 };

   // 添加或者覆盖数据
   client.hset(obj.member, "game", obj.game, redis.print);
   client.hset(obj.member, "first_name", obj.first_name, redis.print);
   client.hset(obj.member, "last_name", obj.last_name, redis.print);
   client.hset(obj.member, "email", obj.email, redis.print);
   client.hset(obj.member, "score", obj.score, redis.print);

   client.hvals(obj.member, function (err, replies) {
      if (err) {
         return console.error("error response - " + err);
      }
      console.log(replies.length + " replies:");
      replies.forEach(function (reply, i) {
        console.log(" " + i + ": " + reply);
      });
   });
   res.end(obj.member + ' set score of ' + obj.score);
   console.timeEnd('test');
  });
});
scoreServer.listen(8124);

// HTTP 服务器关闭，客户端连接关闭
scoreServer.on('close', function() {
client.quit();
});
console.log('listening on 8124');
```

我很好奇如果改变程序中的一个参数性能会有什么变化：从维持一个稳定的 Redis

连接改为每次访问的时候再建立连接，并在请求处理完毕时释放连接。这产生了第二个版本的程序，如示例 14-6 所示。与第一版程序不同之处由粗体标出。

示例 14–6　修改程序，非持久化的 Redis 连接

```
var redis = require("redis"),
    http = require('http');
var scoreServer = http.createServer();
// 监听收到的请求
scoreServer.on('request', function (req, res) {

  console.time('test');

  // 创建 Redis 客户端
  var client = redis.createClient();

  client.on('error', function (err) {
    console.log('Error ' + err);
  });

  // 设置数据库为 1
  client.select(1);

  req.addListener("end", function () {

     var obj = {
         member: 2366,
         game: 'debiggame',
         first_name: 'Sally',
         last_name: 'Smith',
         email: 'sally@smith.com',
         score: 50000 };

     // 添加或者覆盖数据
     client.hset(obj.member, "game", obj.game, redis.print);
     client.hset(obj.member, "first_name", obj.first_name, redis.print);
     client.hset(obj.member, "last_name", obj.last_name, redis.print);
     client.hset(obj.member, "email", obj.email, redis.print);
     client.hset(obj.member, "score", obj.score, redis.print);

     client.hvals(obj.member, function (err, replies) {
        if (err) {
           return console.error("error response - " + err);
        }

        console.log(replies.length + " replies:");
        replies.forEach(function (reply, i) {
          console.log("  " + i + ": " + reply);
        });
```

```
    });

    res.end(obj.member + ' set score of ' + obj.score);
    client.quit();
    console.timeEnd('test');
  });
});

scoreServer.listen(8124);

console.log('listening on 8124');
```

对第二版程序运行 ab 测试，相关的测试结果如下：

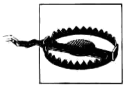
```
Requests per second: 515.40 [#/sec] (mean)
Time per request: 19.402 [ms] (mean)
...
Percentage of the requests served within a certain time (ms)
  50% 18
  66% 20
  75% 21
  80% 22
  90% 24
  95% 27
  98% 33
  99% 40
 100% 341 (longest request)
```

这些测试给了我们很好的指示，持久化的连接有助于提升性能。这在后来的测试中都得到了证实。

当我用 1 000 个并发用户运行测试 100 000 次的时候，持久化的 Redis 连接程序完成了测试，而另一个程序则失败了。过多的并发用户等待 Redis，而 Redis 开始拒绝连接。精确点，在程序崩溃之前完成了 67985 个测试。

14.4.2 Nodeload 与负载测试

Nodeload 提供了命令行工具实现与 ab 一样的测试，但是附加了很多易于阅读的图形结果。Nodeload 也提供了一个模块用于开发你自己的性能测试应用。



警告

还有一个程序也成为 Nodeload，用于构建 Git 代码库和打包为 zip 文件。为了确保使用正确的 Nodeload，用以下命令安装：

```
npm install nodeload-g
```

当 nodeload 全局安装完成后，你可以在任何地方访问该模块的命令行版本。命令行参数与 ab 类似：

```
nl.js -c 10 -n 10000 -i 2 http://examples.burningbird.net:8124
```

程序访问网站 10 000 次，模拟了 10 个并发用户。-i 标识表示数据报告的频率（每两秒，而不是默认的每十秒）。参数列表如下：

-n–number

发起的请求数。

-c --concurrency

并发的用户数。

-t --time-limit

测试的时间限制。

-m --method

使用的 HTTP 方法。

-d --data

PUT 或者 POST 请求发送的数据。

-r --request-generator

getRequest 方法的模块路径（如果有自定义的）。

-q --quiet

不显示过程信息。

-h --help

帮助。

Nodeload 有趣的地方在于测试运行时显示的动态图形。如果你访问测试服务器的 8000 端口（http://localhost:8000 或者通过域名），你可以看到一个测试结果的图形。图 14-3 显示了一个测试结果的截屏。

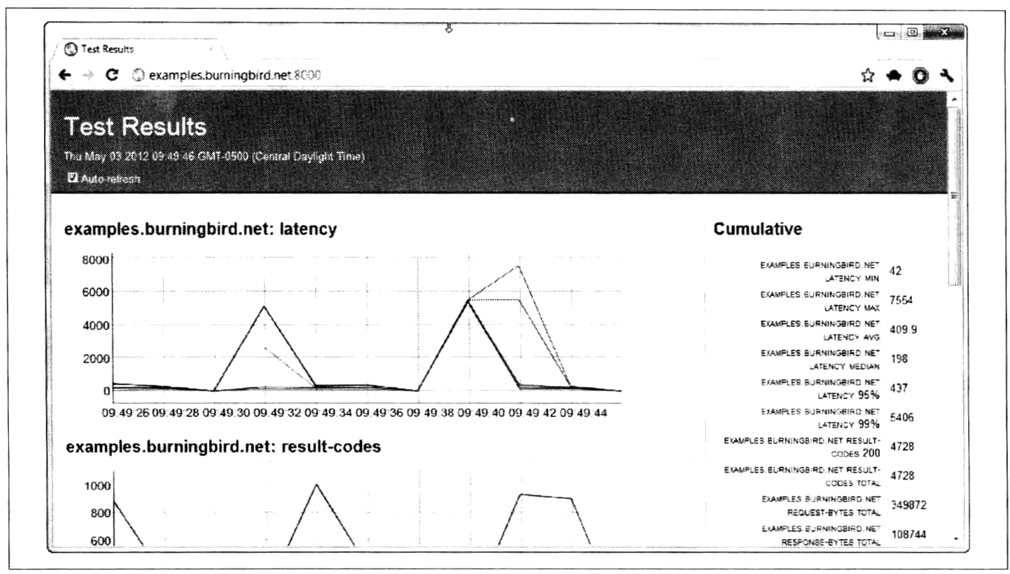


图 14-3　进行中的 Nodeload 测试结果的动态图形

该图形会作为测试结果的日志文件，可以随时访问。在测试结束时，显示的总体结果与 ab 非常类似。一个输出的例子如下所示：

```
Server:                           examples.burningbird.net:8124
HTTP Method:                      GET
Document Path:                    /
Concurrency Level:                100
Number of requests:               10000
Body bytes transferred:           969977
Elapsed time(s):                  19.59
Requests per second:              510.41
Mean time per request(ms):        192.74
Time per request standard deviation: 47.75

Percentages of requests served within a certain time (ms)
  Min: 23
  Avg: 192.7
  50%: 191
  95%: 261
  99%: 372
  Max: 452
```

如果你想提供自定义的测试，你可以使用 Nodeload 模块开发测试应用。Nodeload 提供了动态监控、图形、数据，以及分布式的测试能力。

警告

Nodeload 目前使用 http.createClient，在 Node 0.8.x 的 http.request 中已弃用。尽管看起来尚且可以工作，应该很快就会升级。

14.5 Nodemon 更新代码

在本章结束前，我还想再介绍一个模块：Nodemon。尽管与测试或者调试无关，Nodemon 是一个很便捷的开发工具。

首先，用 npm 安装：

```
npm install nodemon
```

Nodemon 会对你的程序进行封装。可以使用 Nodemon 代替 Node 运行程序：

```
Nodemon app.js
```

Nodemon 在程序运行时监控整个程序目录，检查文件是否有过修改。如果文件变动，Nodemon 会重启程序使当前修改生效。

可以传递参数给程序：

```
nodemon app.ks param1 param2
```

对于 Coffee 也可以使用该模块：

```
nodemon someapp.coffee
```

如果希望 Nodemon 监控其他目录而不是当前目录，使用-watch 标志：

```
nodemon--watch dir1 --watch libs app.js
```

文档记录该模块还有其他一些参数，可以参考 https://github.com/remy/nodemon。

提示

第 16 章介绍了如何在 Forever 使用 Nodemon，Forever 会在程序因为某些原因关闭时重启应用。

第 15 章

安全及防护

Web 应用程序的安全性要比限制人们访问应用服务器更加重要。对安全性的要求有时复杂得有点吓人。幸运的是，对于 Node 应用程序开发者来说，与安全相关的大部分组件已经具备。我们只需在恰当的地方和合适的时间使用它们。

在本章中，我通过四个方面对站点安全进行描述：加密（encryption）、身份验证（authentication）和授权（authorization）、攻击防范（attack prevention），以及沙箱（sandboxing）：

加密

用于确保在互联网上传输数据的安全性，即使它在传输过程中被截获。唯一的可以对数据进行解密的接收机具备了适当的认证（通常是一个 key）。对于需要保密存储的数据也可以采用加密方法。

身份验证及授权

通常指我们在需要访问应用程序的某些保护区域时，需要先进行登录操作。除了通过登录来保证一个人可以访问应用程序的某些区域（授权）外，还需要确保这个人就是他自己所描述的那个人（认证）。

攻击防范

确保用户提交的表单数据中不能含有可以攻击服务器或数据库的文本信息。

沙箱

隔离脚本，使得它只能在一个有限的上下文环境中工作，而不能访问系统资源。

15.1 数据加密

我们在互联网上会发送大量的数据。其中大部分并非关键信息，例如，Twitter 的更新、网页历史、博客文章的评论。不过还有一部分数据具有隐私性，包括信用卡数据、机密电子邮件，或访问服务器时我们输入的登录信息。为了确保这些数据的隐私性，特别是在传输过程中不被窃取，我们就需要为通信加密。

15.1.1 TSL / SSL 配置

我们可以使用 SSL（安全套接字层，Secure Sockets Layer）以及它的升级 TLS（传输层安全，Transport Layer Security）来建立安全、防篡改的客户端和服务端通信。TSL / SSL 为 HTTPS 提供底层加密，我将在下一节进行说明。然而，在做 HTTPS 相关开发工作之前，我们必须做一些环境设置。

一个 TSL / SSL 连接需要客户端和服务器之间进行握手。在握手期间，客户端（通常是浏览器）会让服务器知道它支持哪些安全功能。服务器会从中选择将在通信中采用的一些功能，并通过一个 SSL 证书发送到客户端，该证书中还包括一个公共密钥。客户端在确认证书有效性后，将采用服务器提供的密钥加密生成一个随机数，然后将其发送回服务器。然后，服务器将使用其私有密钥来解密并得到这个随机数，它将被用于建立客户端与服务器的安全通信。

为了使一切可以正常工作，你需要同时生成公钥、私钥以及证书。对于生产环境，证书将由受信任机构签署（例如我们的域名注册商），但在开发环境中，你可以使用一个自签名证书。这会在浏览器中生成一个明显的警告信息，不过这不是问题，因为并没有真实的用户访问我们在开发环境中的站点。

我们可以使用 OpenSSL 来生成这些文件。如果你使用的是 Linux，它应该已经被安装了。Windows 则有一个二进制安装包，而苹果使用自己的 Crypto 库。此处，以 Linux 环境设置为例。

首先，在命令行中输入以下内容：

```
openssl genrsa -des3 -out site.key 1024
```

该命令使用 Triple-DES 加密生成私钥，并以 PEM（增强型保密邮件）格式保存，以便私钥是 ASCII 可读的。

系统将提示你输入一个密码，你会在下一步创建证书签名请求（CSR）时用到它。

生成 CSR 时，你需要输入刚刚创建的密码。然后还需要回答很多问题，包括所在国家（如 US 代表美国）、所在州或省、城市名称、公司名称和组织、电子邮件地址。而其中最重要的一项是通用名称（Common Name）。你需要使用站点的主机名，例如 burningbird.net 或 yourcompany.com。它用于承载服务应用程序的主机名称。此处我使用了 examples.burningbird.net。

```
openssl req -new -key site.key -out site.csr
```

使用私钥需要提供之前的密码作为通行口令。但如果每次启动服务器时就必须提供口令，这在生产环境中将是一个问题。在接下来的步骤中，你需要从私钥中删除口令。首先，重命名私钥：

```
mv site.key site.key.org
```

然后输入：

```
openssl rsa -in site.key.org -out site.key
```

如果删除了口令，请务必确保秘钥文件只能对 root 用户可读，以保证服务器的安全性。

接下来的任务是生成自签名证书。下面的命令创建了一个只有 365 天有效期的证书：

```
openssl x509 -req -days 365 -in site.csr -signkey site.key -out final.crt
```

现在所有需要的组件已经备齐，可以开始使用 TLS / SSL 和 HTTPS 了。

15.1.2 使用 HTTPS

如果 Web 页面需要用户登录或需要处理信用卡信息，那么最好能支持 HTTPS。HTTPS 是 HTTP 协议的变种，它能与 SSL 相结合以确保网站的真实性和有效性，还能确保在传输过程中对数据进行加密，以及数据能完好到达且没有任何篡改。

在 Node 应用程序中添加 HTTPS 支持与添加 HTTP 支持的过程非常类似，除了需要通过一个 options 对象来提供公共加密密钥和签名的证书。HTTPS 服务器的默认端口也不同：HTTP 通常使用 80 端口提供服务，而 HTTPS 服务则使用 443 端口。

示例 15-1 演示了一个非常简单的 HTTPS 服务器。当通过浏览器访问它时，会得到 HelloWorld 信息。

示例 15-1　一个简单的 HTTPS 服务器

```
var   fs = require("fs"),
      https = require("https");

var privateKey = fs.readFileSync('site.key').toString();
var certificate = fs.readFileSync('final.crt').toString();
var options = {
   key: privateKey,
   cert: certificate
};

https.createServer(options, function(req,res) {
   res.writeHead(200);
   res.end("Hello Secure World\n");
}).listen(443);
```

公钥和证书都是公开的，程序会读取他们的内容。然后将这些内容附加到 options 对象中，并在调用 https.createServer 方法时作为第一个参数传入。该方法的回调函数同样会接收服务端的 request 对象和 response 对象并作为参数。

由于使用了自签名的证书，访问页面时会发生一些情况，如图 15-1 所示。这是为什么只有在测试过程中才能使用自签名证书的原因。

图 15-1　用 Chrome 访问使用了自签名证书的 HTTPS 网站

浏览器的地址栏也以另一种方式显示了该网站的证书不被信任，如图 15-2 所示。通常显示一个锁表示该网站是通过 HTTPS 访问的，现在它显示了一个红色的带 X 的锁，表示该 HTTPS 站点的证书不能被信任。单击该图标会打开一个信息窗口，显示了有关证书的详细信息。

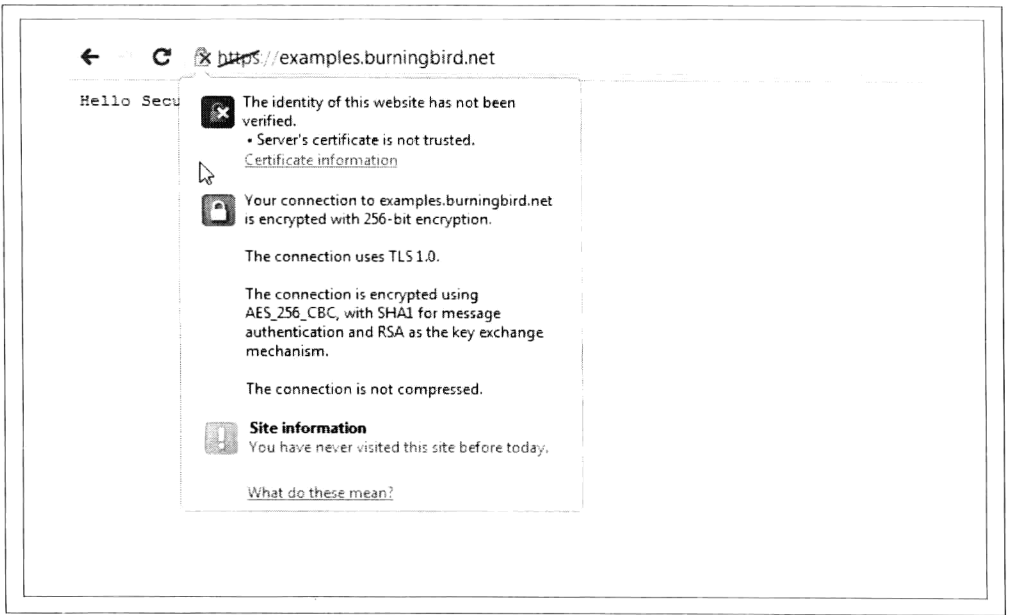


图 15-2　点击锁图标可以获取更多证书信息

对于 Web 应用程序来说，加密不仅仅在通信时使用，在保存用户名和其他敏感数据时也会用到。

15.1.3　如何安全的保存密码

Node 提供了一个名为 Crypto 的模块用于实现加密功能。下面是摘自该模块文档的一段描述：

Crypto 模块依赖于 OpenSSL 所提供的底层平台。OpenSSL 对安全认证进行了封装，以便于我们建立安全的 HTTPS 和 HTTP 链接。

Crpto 模块还对 OpenSSL 提供的 shash、hmac、cipher、decipher、sign 和 verify 方法进行了封装以便于使用。

本节会使用的是其中的 OpenSSLhash 功能。

对于大多数 Web 应用程序来说，保存用户登录信息及密码是必须支持的一项功能，但同时这项功能也是最为脆弱的。当用户名和密码被存储在 Web 应用程序的纯文本数据库后，我们可能只需要花费五分钟时间就能破解并得到登录信息，如果这些信息被泄露的话，天知道会带来什么影响。

所以，你不能将密码以纯文本格式来保存。幸运的是，我们可以使用 Node 提供的 Crypto 模块。

Crypto 模块的 createHash 方法可以用来加密密码。在下面的例子中，我们首先创建了一个使用 SHA1 算法的 hash，然后使用该 hash 对密码进行编码，最后再提取加密数据摘要并将其保存在数据库中：

```
Var hashpassword = crypto.createHash('sha1')
                                 .update(password)
                                 .digest('hex');
```

我们使用的摘要编码格式为十六进制。默认编码是二进制的，也可以使用 base64 编码。

许多应用程序都采用这种方式来加密密码。然而，如果将使用 hash 加密后的密码保存在数据库的话，也是存在问题的。因为这种密码容易通过彩虹表（rainbow table）来破解。

简单地说，彩虹表是一个针对各种可能的字母组合预先计算好的哈希值集合。所以，即使有人认为自己的密码不可能被破解（说实话，我们大多数人不会这么想），但如果这段字符序列恰好在彩虹表中有对应条目，那么判断你的密码就变得非常容易了。

解决这个问题的方法是使用 salt，并将它与密码拼接然后再进行加密计算。salt（不要以为是那种常见的晶体）是我们使用某种方法生成的一段唯一值。它可以一直不变，并在所有密码加密时使用，但是一定要安全地保存在服务器上。不过更好的办法是为每个用户密码生成唯一的 salt，然后将其与密码一起存储。诚然，salt 也可能与密码一起被窃取，但试图破解该密码的人则需要单独为此密码生成彩虹表，从而极大地增加了破解的复杂性。

示例 15-2 是一个简单的命令行应用程序，我们需要为其传递用户名和密码作为参数，程序会对密码进行加密，然后将结果存储在 MySQL 数据库表中。下面的 SQL 用于创建使用到的数据库表：

```
CREATE TABLE user (userid INT NOT NULL AUTO_INCREMENT, PRIMARY KEY(userid),
username VARCHAR(400) NOT NULL, password VARCHAR(400) NOT NULL);
```

salt 可以通过将一个日期值乘以一个随机数然后四舍五入取整后得到。它与密码拼接在一起后得到一个字符串，该字符串将会被用来做加密处理。最终得到的处理结果将作为用户数据插入到 MySQL 用户表中。

示例 15-2　使用 Crypto.createHash 方法和 salt 来加密密码

```
var mysql = require('mysql'),
    crypto = require('crypto');

var client = mysql.createClient({
   user: 'username',
   password: 'password'
  });
client.query('USE databasenm');

var username = process.argv[2];
var password = process.argv[3];

var salt = Math.round((new Date().valueOf() * Math.random())) + '';

var hashpassword = crypto.createHash('sha512')
                   .update(salt + password)
                   .digest('hex');
// create user record
client.query('INSERT INTO user ' +
   'SET username = ?, password = ?, salt = ?',
   [username, hashpassword, salt], function(err, result) {
      if (err) console.log(err);
      client.end();
});
```

示例 15-3 的示例程序根据用户名查询数据库中的对应的密码及 salt，以便测试用户名和密码的有效性。在程序中，我们再次使用 salt 对用户提供的登录密码进行加密。并将加密结果与存储在数据库中的加密数据做比较。如果二者不匹配，则用户不能通过验证。反之，用户信息匹配，身份验证通过。

示例 15-3　验证用户名和已加密密码

```
var mysql = require('mysql'),
    crypto = require('crypto');

var client = mysql.createClient({
   user: 'username',
   password: 'password'
  });

client.query('USE databasenm');

var username = process.argv[2];
var password = process.argv[3];

client.query('SELECT password, salt FROM user WHERE username = ?',
   [username], function(err, result, fields) {
   if (err) return console.log(err);

   var newhash = crypto.createHash('sha512')
                 .update(result[0].salt + password)
```

```
                .digest('hex');

      if (result[0].password === newhash) {
         console.log("OK, you're cool");
      } else {
         console.log("Your password is wrong. Try again.");
      }
            client.end();
   });
```

现在试着运行一下这些示例程序，我们首先来创建一条用户信息，用户名为 Michael，密码为 applef*rk13*：

```
node password.js Michael apple*frk13*
```

我们再使用同样的用户名和密码进行验证：

```
node check.js Michael apple*frk13*
```

并获得预期的结果：

```
OK, you're cool
```

再尝试一个不同的密码：

```
node check.js Michael badstuff
```

同样得到了预期的结果：

```
Your password is wrong. Try again
```

当然，我们并不希望在真实情况下让用户通过命令行登录系统。同样也不希望总是使用本地密码系统来验证用户。所以，接下来让我们看看 Node 对认证及授权的支持。

15.2 认证/授权及 Passport

你如何证明自己的身份？你有没有权限做某项操作？这个操作是否会引起麻烦？身份验证与授权管理这两种不同的技术可以帮助我们回答以上问题。

认证（Authentication）所关注的是确保你就是你说的那个人。当 Twitter 为账户附加一个验证标志时，就表示当前用户就是其本人。授权（Authorization）则从另一方面确保你只能访问你能访问的。在 Drupal 站点的众多用户中，可能有一半用户有权限发表评论，有五人可以发表文章和评论，但只有一人可以控制所有一切。这

个网站并不在乎名为 Big Daddy 的用户是谁，他只可以发表评论，但不能删帖。

人们常常将认证及授权作为一个功能看待。通常情况下，当你尝试做一些操作时，你需要通过一些验证手段来证明你是谁。你可能会被要求提供用户名和密码。然后，一旦通过验证，你将可以做更多操作。但应用程序依然会根据你提供的认证信息来限制你只能访问某些网页，或只能执行某些操作。

有时候认证是通过第三方完成的。第三方认证的一个例子是使用 OpenID。在支持 OpenID 的应用程序中，用户将通过 OpenID 来验证身份并提供相应的应用程序访问权限，而非通过其注册的用户名和密码。

有时身份验证和授权都会放在第三方网站进行。例如，如果应用程序要访问 Twitter 或 Facebook 账户（可能为了发布消息或获得信息），那么就必须通过这些网站的身份验证，你的应用程序才能取得对应的访问授权。这种方式叫 OAuth，它采用了不同的授权策略。

在 Node 中，使用 Passport 以及一个或多个 Passportstrategy 模块就可以支持相关功能并帮助我们实现上述所有场景。

提示

Passport 并不是唯一一个提供身份验证和授权的模块，但我发现它是最容易使用的。

15.2.1 授权/认证策略：Oauth、OpenID、用户名/密码验证

让我们来仔细看看三种不同类型的授权/认证策略。

当你访问一个内容管理系统（CMS，例如 Drupal 或 Amazon）的后台管理部分时，你需要使用身份验证。此时你需要提供用户名和密码，在这两者通过网站验证之前，你无法获得访问权限。这是使用的最为广泛的授权/认证策略。而且在大多数情况下，它也是有效的一种策略。

此前的章节中，我演示了如何让数据库中保存的用户密码更加安全。即使系统被攻破，数据窃贼也很难取得以纯文本格式保存的用户密码。当然，他们可以破解你的密码。但是，如果你的密码是一个相对无语义的，而且是包含了字母、符号和数字的组合时，破解它就要花费相当多的时间和 CPU 资源了。

OAuth 是一种无需直接使用用户密码来访问数据的方式(比如访问一个人的 Twitter

账户数据)。它是一种数据访问授权方式，用户无需再将个人凭据信息保存在多个不同的地方(这样会增加个人凭据被危害的可能性)。它也为用户提供了更大的控制权，因为用户通常可以随时从主账户中撤销授权。

OAuth 主要用于数据访问的授权。而 OpenID 则主要关注用户身份验证，尽管也包含了一些授权处理。

OpenID 的使用并没有 OAuth 广泛，它主要用于评论系统以及不同媒体网站的用户注册。评论系统需要解决的问题之一是：任何人都可以声称自己是某个人，但却没有办法确认他就是他说的那个人。如果使用 OpenID 的话，一个人可以登录评论系统或注册用户，OpenID 会确保对这个人的身份验证，至少在 OpenID 系统内是被验证过的。

OpenID 也是一种解决用户多地注册的方案，我们无需再为不同的身份验证环境创建各自的用户名和密码。你只要提供自己的 OpenID 给当前信息系统，它会从 OpenID 提供商那里获得对你的身份验证结果，你就大功告成了。

这三种策略互相之间并不冲突。许多应用程序同时支持所有三种策略：通过本地身份验证来做一些管理操作，通过 OAuth 来共享数据(例如 Facebook 和 Twitter)，使用 OpenID 来支持用户注册和评论。

在 Node 中有许多模块可以用来实现上述所有身份验证和授权方式，但我打算把重点放在 Passport 模块。Passport 是同时支持 Connect 和 Express 的一款中间件，用以提供身份验证和授权。你可以使用 npm 来安装它：

```
npm install passport
```

Passport 还可以使用一些 stragey，不过它们需要单独安装。使用各种 stragey 时，有一些基本要求：

- strategy 必须被正确安装；
- 必须在应用程序中对 strategy 进行配置；
- 作为配置的一部分，strategy 需要包括一个用来验证用户凭据的回调函数；
- 使用 strategy 所需要的额外工作量取决于对用户凭据进行审核的权威机构：Facebook 和 Twitter 需要一个账户及账户的 key，而 local strategy 需要保存有用户名和密码的数据库；
- 所有 strategy 都需要一个本地数据存储区来做授权机构名称与应用程序的

映射；

- 直接使用 Passport 模块提供的用户会话保存功能。

在本章中，我们会使用两种 Passportstrategy，分别用于本地认证和授权，以及 Twitter 的 OAuth 认证授权。

15.2.2 Local Passport Strategy

我们可以使用 npm 安装 Local Passport Strategy 模块（passport-local）：

```
npm install passport-local
```

Passport 是中间件，因此在使用前必须与其他 Express 中间件一样被实例化。首先需要在代码中包含 passport 和 passport-local 模块：

```
var express = require('express');
var passport = require('passport');
var localStrategy = require('passport-local').Strategy;
```

使用下面的代码启用 Passport 中间件：

```
var app = express();
app.configure(function(){
   ...
   app.use(passport.initialize());
   app.use(passport.session());
   ...
});
```

接下来需要配置 local strategy。与配置其他 strategy 所采用的格式是一致的：需要使用 use 方法将一个 strategy 的新实例传递给 Passport，这和在 Express 中使用 Passport 模块的方式很类似：

```
passport.use(new localStrategy( function (user, password, done) { ... }
```

passport-local 模块期望用户的账户信息（用户名和密码）是以 Post 表单的形式传递到 Web 应用程序，并且用户名和密码的具体值应该被保存在名为 username 和 password 的字段中。如果你想为用户名和密码使用其他字段名称，则需要在创建 strategy 实例时将字段名称作为可选项传入：

```
var options =
 { usernameField : 'appuser',
    passwordField : 'userpass'
  };
passport.use(new localStrategy(options, function(user, password, done) { ... }
```

在从请求中提取了用户名和密码后，传递给 strategy 构造函数的回调函数会被调用。我们需要在该函数中实现身份验证逻辑，并返回如下值：

- 一个 error，如果发生错误的话；
- 一条信息，如果用户身份验证失败；
- user 对象，如果用户通过身份验证；

每当用户试图访问站点的受保护区域，Passport 都会查询该用户是否已经被授权。在下面的代码中，当用户试图访问受限制的管理页面时，一个名为 ensure Authenticated 的函数将会被调用，以确定用户是否被授权：

```
app.get('/admin', ensureAuthenticated, function(req, res){
  res.render('admin', { title: 'authenticate', user: req.user });
});
```

ensureAuthenticated 函数会检查 req.isAuthenticated 方法的返回值，它是 Passport 为 request 对象增加的一个扩展方法。如果该方法返回 false，当前用户的访问将被重定向到登录页面：

```
function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) { return next(); }
  res.redirect('/login')
}
```

为了使登录状态能够在一个会话中保持有效，Passport 提供了两个方法：serializeUser 和 deserializeUser。我们必须在这两个方法的回调函数中实现相关功能。简单来说，passport.serializeUser 会序列化用户标识符，而 passport.deserializeUser 则使用标识符在数据存储中进行查找，并返回一个包含用户详细信息的对象：

```
passport.serializeUser(function(user, done) {
  done(null, user.id);
});
passport.deserializeUser(function(id, done) {
  ...
});
```

在 Passport 中，序列化并不是必须的。如果你不想序列化用户信息，只要不在代码中使用 passport.session 中间件即可：

```
app.use(passport.session());
```

如果你决定要序列化并将用户信息保存在会话中（你也应该这样做，否则用户会因

为不断地收到登录请求而非常恼火），则必须确保 Passport 中间件被包含在代码中，并且放在包含 session 中间件的代码之后：

```
app.use(express.cookieParser('keyboard cat'));
app.use(express.session());
app.use(passport.initialize());
app.use(passport.session());
```

如果没有保持适当的包含顺序，身份认证功能将不能正常工作。

最后一块需要考虑的功能是如何处理验证不通过的情况。在认证过程中，如果一个用户的用户名没有在数据存储中被搜索到，会生成一条错误消息。如果搜索到了用户名，但是密码不匹配时，就会产生错误。我们需要将这些错误信息返回给用户。

Passport 使用 Express 2.x 的 req.flash 方法将需要返回给用户的错误信息保存在队列中。在前面章节中我们并没有使用过 req.flash，因为它在 Express 3.x 中已经被废弃。然而，为了确保 Passport 可以正常地工作在 Express 2.x 和 3.x 中，Passport 开发者创建了一个新模块 connect-flash，它实现了与 req.flash 相似的功能。

使用 npm 安装 connect-flash 模块：

```
npm install connect-flash
```

在应用程序中使用：

```
var flash = require('connect-flash');
```

然后作为 Express 中间件使用：

```
app.use(flash());
```

此时，在 POSTlogin 路由项中，如果用户没有通过身份验证，他将被重定向到登录表单并给出相关错误信息：

```
app.post('/login',
  passport.authenticate('local', { failureRedirect: '/login',
failureFlash: true }),
  function(req, res) {
    res.redirect('/admin');
  });
```

在身份验证过程中产生的错误信息通过 req.flash 传递到了视图渲染引擎，并在呈现登录表单时显示出来：

```
app.get('/login', function(req, res){
  var username = req.user ? req.user.username : '';
  res.render('login', { title: 'authenticate', username: username,
                  message: req.flash('error') });
});
```

视图引擎会将错误消息呈现在登陆表单之外的其他元素上，如下述 Jade 模板代码所示：

```
extends layout

block content
  h1 Login
  if message
    p= message
  form(method="POST"
       action="/login"
       enctype="application/x-www-form-urlencoded")
    p Username:
        input(type="text"
              name="username"
              id="username"
              size="25"
              value="#{username}"
              required)
    p Password:
        input(type="password"
              name="password"
              id="password"
              size="25"
              required)
      input(type="submit"
            name="submit"
            id="submit"
            value="Submit")
      input(type="reset"
            name="reset"
            id="reset"
            value="reset")
```

为了能更好地说明这些技术细节，我将示例 15-3 实现的命令行认证应用程序修改为一个 Express 应用程序，代码如示例 15-4 所示，并通过 Passport 实现身份验证。通过登录页面的表单提交身份认证信息是访问应用程序管理页面的唯一途径，而访问顶层索引页则无需身份验证。

我将示例 15-3 中有关 MySQL 的代码直接纳入到了该程序的认证过程中(虽然通常情况下这部分功能会被划分在一个更加正式的独立的应用程序中)。另一些 MySQL 访问代码则通过用户标识符来反序列化用户信息。

示例 15-4　整合了密码 hash、MySQL 用户表，以及 Passport 身份验证的 Express 应用程序

```
// modules
var express = require('express')
  , flash = require('connect-flash')
  , passport = require('passport')
  , LocalStrategy = require('passport-local').Strategy
  , http = require('http');

var mysql = require('mysql')
  , crypto = require('crypto');

// check user authentication

function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) { return next(); }
  res.redirect('/login')
}

// serialize user to session
passport.serializeUser(function(user, done) {
  done(null, user.id);
});

// find user in MySQL database
passport.deserializeUser(function(id, done) {

   var client = mysql.createClient({
      user : 'username',
      password: 'password'
   });

   client.query('USE databasenm');

   client.query('SELECT username, password FROM user WHERE userid = ?',
              [id], function(err, result, fields) {
       var user = {
           id : id,
           username : result[0].username,
           password : result[0].password};
       done(err, user);
       client.end();


  });
});

// configure local strategy
// authenticate user against MySQL user entry
passport.use(new LocalStrategy(
  function(username, password, done) {

     var client = mysql.createClient({
        user : 'username',
        password: 'password'
```

```
    });

    client.query('USE nodetest2');

    client.query('SELECT userid, password, salt FROM user WHERE username = ?',
             [username], function(err, result, fields) {

        // database error
        if (err) {
           return done(err);

        // username not found
        } else if (result.length == 0) {
           return done(null, false, {message: 'Unknown user ' + username});

        // check password
        } else {
           var newhash = crypto.createHash('sha512')
                         .update(result[0].salt + password)
                         .digest('hex');

           // if passwords match
           if (result[0].password === newhash) {
              var user = {id : result[0].userid,
                          username : username,
                          password : newhash };
              return done(null, user);

           // else if passwords don't match
           } else {
              return done(null, false, {message: 'Invalid password'});
           }
        }
        client.end();
    });
}));

var app = express();

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser('keyboard cat'));
  app.use(express.session());
  app.use(passport.initialize());
  app.use(passport.session());
  app.use(flash());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

```
app.get('/', function(req, res){
  res.render('index', { title: 'authenticate', user: req.user });
});

app.get('/admin', ensureAuthenticated, function(req, res){
  res.render('admin', { title: 'authenticate', user: req.user });
});

app.get('/login', function(req, res){
  var username = req.user ? req.user.username : '';
  res.render('login', { title: 'authenticate', username: username,
               message: req.flash('error') });
});

app.post('/login',
  passport.authenticate('local', { failureRedirect: '/login', failureFlash: true }),
  function(req, res) {
    res.redirect('/admin');
});

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

相比较之前的示例代码来说，示例 15-4 的代码比较长，但它的确能比较好地说明在真实环境中 Passport 以及之前提及的组件之间是如何工作的。

让我们再来仔细看看身份验证方法。一旦应用程序通过用户名查询用户记录后，它会调用回调函数并传入一个数据库错误信息（如果发生错误的话）。如果没有产生任何错误，但也没有找到对应的用户名时，应用程序会调用回调函数并将 username 设置为 false 以表示该用户不存在，并给出相应的描述消息。如果该用户存在，但密码不匹配时，处理方法类似：为 username 返回 false 值，并生成一条消息。

当没有数据库错误，用户在 user 表中存在，并且密码匹配时，用户对象才能被创建并传递给回调函数：

```
// database error
if (err) {
   return done(err);
// username not found
} else if (result.length == 0) {
   return done(null, false, {message: 'Unknown user ' + username});

// check password
} else {
   var newhash = crypto.createHash('sha512')
              .update(result[0].salt + password)
              .digest('hex');
```

```
    // if passwords match
    if (result[0].password === newhash) {
       var user = {id : result[0].userid,
                   username : username,
                   password : newhash };
       return done(null, user);

    // else if passwords don't match
    } else {
       return done(null, false, {message: 'Invalid password'});
    }
 }
```

该用户对象随后被序列化到 session 中，同时用户被授权访问管理页面。只要当前会话不结束，用户便可以继续访问管理页面而无需再次输入验证信息。

15.2.3 Twitter Passport Strategy（OAuth）

我们可以使用 OAuth 来完成身份验证，而不需要在本地存储用户名和密码，也不需要再自己实现身份验证逻辑。同时，这也是一种将我们的站点与其他第三方站点（如 Facebook、Google+或 Twitter 等）进行紧密集成的方式。

Passport 可以通过 passport-twitter 模块来使用 Twitter 提供的身份验证功能。首先使用 npm 安装该模块：

```
npm install passport-twitter
```

为了使用 Twitter 的 OAuth 来进行用户身份验证，你需要拥有 Twitter 开发人员账户，并能得到 consumerkey 和 consumersecret。它们被用于在应用程序中生成 OAuth 请求。

一旦你得到了 consumerkey 和 secret，就可以在创建 Twitterstrategy 实例时，将它们与 callbackURL 一起传递给构造函数：

```
passport.use(new TwitterStrategy(
       { consumerKey: TWITTER_CONSUMER_KEY,
         consumerSecret: TWITTER_CONSUMER_SECRET,
         callbackURL: "http://examples.burningbird.net:3000/auth/twitter/callback"},
       function(token, tokenSecret,profile,done) {
          findUser(profile.id, function(err,user) {
             console.log(user);
             if (err) return done(err);
             if (user) return done(null, user);
             createUser(profile, token, tokenSecret, function(err, user) {
               return done(err,user);
             });
          })
       })
);
```

尽管 Twitter 可以提供身份认证功能，但你仍然很有可能需要一种方法来存储用户信息。在创建 Twitter strategy 实例的代码块中，你会注意到其回调函数可接受的参数包括：token、tokenSecret、profile，以及另一个回调函数。在通过 Twitter 的身份验证后，其返回的响应信息中会包含 token 和 tokenSecret。token 和 tokenSecret 用于与 Twitter 上的个人账户进行交互，例如发布最新的 tweets、tweet 到好友的账户，或查看好友的关注者（follower）信息等。总之用户在 Twitter 页面上能够看到的所有信息都能够通过 Twitter 的 API 取得。

其实，我们真正感兴趣的是 profile 对象。它包含了有关个人账户的许多信息，包括：Twitter 网名、全名、描述、位置、头像图片、follower 数量、followed 数量、tweet 数量等。这些正是我们需要提取并在本地数据库存储的用户信息。我们没有保存密码，而且 Oauth 也没有公开个人的身份验证信息。相反，我们只是存储一些我们会在 web 程序中使用到的有助于区分用户的个性化信息。

当对用户进行身份验证时，应用程序会根据他的 Twitter 标识符在本地数据库中查找。如果对应的标识符存在，则返回本地存储的用户信息。如果没有找到对应的用户标识符，则会在数据库中创建并记录一条新用户信息。在程序中我们使用 findUser 和 createUser 两个函数用来实现上述处理过程。findUser 函数还为 Passport 从 session 反序列化用户信息提供支持：

```
passport.deserializeUser(function(id, done) {
   findUser(id, function(err, user) {
      done(err,user);
   });
});
```

我们不再需要登录页面，因为 Twitter 可以提供登录表单。因此在应用程序中只包含了一个可以通过 Twitter 完成认证的链接：

```
extends layout
block content
   h1= title
   p
      a(href='/auth/twitter') Login with Twitter
```

如果用户没有登录到 Twitter，那么程序将会为他呈现一个登录页面，如图 15-3 所示。

一旦用户成功登录，网页将被重新重定到应用程序，然后显示用户管理页面。目前，管理页面只是简单地输出一些从 Twitter 上取得的用户信息，包括用户的网络昵称和头像：

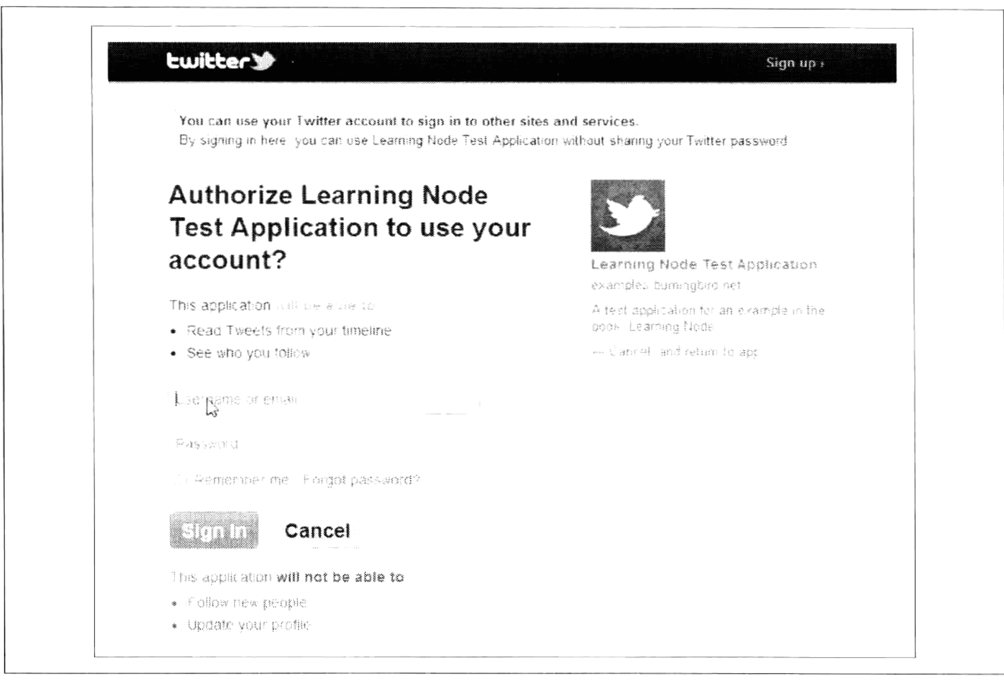


图 15-3 Node 应用程序中的 Twitter 登录和授权页面

```
extends layout

block content
  h1 #{title} Administration
  p Welcome to #{user.name}
  p
     img(src='#{user.img}',alt='avatar')
```

而这些数据是在完成第一次认证时保存的。如果打开你的 Twitter 账户设置页面，然后点击应用程序，你会在列表中看到我们的应用程序，如图 15-4 所示。

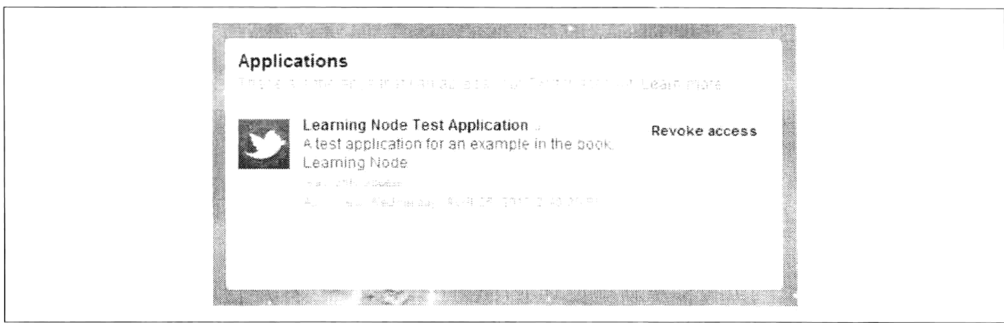


图 15-4 包含了 Node 示例程序的 Twitter 应用设置页面

示例 15-5 是一个完整示例代码，实现了 Twitter 用户认证并将用户信息保存在 MySQL 数据库中。当然，你也可以将数据存储在 MongoDB 中，甚至是 Redis 也行。我们没有使用 Crypto 模块了，因为我们不再需要存储密码了，这也是通过第三方服务进行身份验证的一个明显优势。

示例 15-5 通过 Twitter 实现用户验证的完整示例程序

```
var express = require('express')
  , flash = require('connect-flash')
  , passport = require('passport')
  , TwitterStrategy = require('passport-twitter').Strategy
  , http = require('http');

var mysql = require('mysql');

var TWITTER_CONSUMER_KEY = "yourkey";
var TWITTER_CONSUMER_SECRET = "yoursecret";

var client = mysql.createClient({
   user : 'username',
   password : 'password'
});

client.query('USE nodetest2');

function findUser(id, callback) {
   var user;

   client.query('SELECT * FROM twitteruser WHERE id = ?',
              [id], function(err, result, fields) {
       if (err) return callback(err);
       user = result[0];
       console.log(user);
       return callback(null,user);
  });
};
function createUser(profile, token, tokenSecret, callback) {

   var qryString = 'INSERT INTO twitteruser ' +
                   '(id, name, screenname, location, description,' +
                   'url, img, token, tokensecret)' +
                   ' values (?,?,?,?,?,?,?,?,?)';
   client.query(qryString, [
             profile.id,
             profile.displayName,
             profile.username,
             profile._json.location,
             profile._json.description,
             profile._json.url,
             profile._json.profile_image_url,
             token,
             tokenSecret], function(err, result) {
                if (err) return callback(err);
                var user = {
```

```
                   id : profile.id,
                   name : profile.displayName,
                   screenname : profile.screen_name,
                   location : profile._json.location,
                   description: profile._json.description,
                   url : profile._json.url,
                   img : profile._json.profile_image_url,
                   token : token,
                   tokensecret : tokenSecret};
                   console.log(user);
                   return callback(null,user);
               });
};

function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) { return next(); }
  res.redirect('/auth/twitter')
}

passport.serializeUser(function(user, done) {
  done(null, user.id);
});

passport.deserializeUser(function(id, done) {
   findUser(id, function(err, user) {
      done(err,user);
   });
});

passport.use(new TwitterStrategy(
       { consumerKey: TWITTER_CONSUMER_KEY,
         consumerSecret: TWITTER_CONSUMER_SECRET,
         callbackURL: "http://examples.burningbird.net:3000/auth/twitter/callback"},
       function(token, tokenSecret,profile,done) {
          findUser(profile.id, function(err,user) {
             console.log(user);
             if (err) return done(err);
             if (user) return done(null, user);
             createUser(profile, token, tokenSecret, function(err, user) {
               return done(err,user);
             });
          })
       })
);

var app = express();


app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser('keyboard cat'));
```

```
  app.use(express.session());
  app.use(passport.initialize());
  app.use(passport.session());
  app.use(flash());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.get('/', function(req, res){
  res.render('index', { title: 'authenticate', user: req.user });
});

app.get('/admin', ensureAuthenticated, function(req, res){
  res.render('admin', { title: 'authenticate', user: req.user });
});

app.get('/auth', function(req,res) {
   res.render('auth', { title: 'authenticate' });
});

app.get('/auth/twitter',
  passport.authenticate('twitter'),
  function(req, res){
  });

app.get('/auth/twitter/callback',
  passport.authenticate('twitter', { failureRedirect: '/login' }),
  function(req, res) {
    res.redirect('/admin');
  });

http.createServer(app).listen(3000);

console.log("Express server listening on port 3000");
```

在 Node 中，其他 OAuth 服务的使用步骤与 Twitter Passport strategy 类似。你甚至能直接使用上述 Twtter 认证代码来通过 Facebook 的 OAuth 服务验证用户。唯一的区别是，你需要提供 Facebook key 和 secret，而非 Twitter 的。考虑到认证处理代码的相似性，今天的许多应用程序都支持使用多种 OAuth 服务完成用户身份验证。

Passport 对不同服务返回的数据也做了很好地整理和封装，即使使用不同的服务，处理 profile 对象的代码改动也不会过大。然而，你还是需要对返回的 profile 信息进行分析，以确定哪些是需要的，哪些不需要，以及哪些需要存储。

如果用户撤销了 OAuth 服务对应用程序的访问授权，并且还决定使用另一个服务来通过身份验证。在这种情况下，应用程序将会创建一条新的用户信息并存储起来，然后继续正常运行。但数据库中仍然保存了之前用户认证时的信息，而且并没有因为用户更换认证服务而得到更新，这是唯一的负面影响。我将把这个问题留给你作

为练习。现在是时候来看看影响应用程序安全性的另一方面：表单数据。

15.3 保护应用程序，防止攻击

作为一名 JavaScript 开发人员，你可能明白直接将用户输入传递给 eval 语句执行的危害。作为一个 Web 开发人员，你也明白将用户通过表单提交的文本信息未经处理直接附加在 SQL 语句 where 子句后的危害。

其实，所有 Node 应用程序都具有与客户端 JavaScript 应用程序一样的弱点。此外，作为使用数据库系统（特别是关系型数据库系统）的服务器端程序，Node 应用程序也同样存在着上述使用 SQL 语句的弱点。

正如上节所述，为了确保你的应用程序是安全的，你需要提供良好的授权和认证系统。同时你还需要保护你的应用程序免受注入式攻击，或其他企图越过授权系统以获取重要保密数据的行为。

此前我们通过用户提交的登录表单接收文本信息并将其直接使用在 SQL 查询语句中。这不是明智的做法，因为任何人都可以在文本中附加内容而对 SQL 数据库造成伤害。例如，如果文本是为 WHERE 子句提供数据，并被直接附加在 WHERE 语句后：

```
var whereString = "WHERE name = " + name;\
```

如果 name 字符串包含以下内容：

```
'johnsmith; drop table users'
```

你就会碰到问题。当处理来自用户输入的文本或 JSON 数据时，都有可能发生类似问题：用户的输入数据可能对系统造成伤害。

不过我们可以在使用输入信息前对其净化，以防止这两种类型的漏洞。我们还需要利用其他一些工具和技术，来尽可能确保我们的应用程序是安全的。

15.3.1 不要使用 eval

不管是否是 Node 环境中使用 JavaScript，一个简单的规则可以让你的应用程序更安全：不要使用 eval。eval 函数是限制最少且最宽松的 JavaScript 组件，我们应该谨慎使用它。

在大多数情况下，我们并不需要使用 eval。例如，当需要将一个 JSON 字符串转换

成一个对象时，我们可能会想到使用它。然而，还有另一种简单的可以防止 JavaScript 注入攻击，并能将字符串转换成对象的方法是使用 JSON.parse 来处理 JSON，而非 eval。eval 语句不会对文本中包含的内容进行辨识，而 JSON.parse 则会验证文本内容并判断是否为有效的 JSON 格式：

```
var someObj = JSON.parse(jsonString);
```

考虑到 Node 使用的是 V8 引擎，我们可以直接访问 JSON 对象；当然，我们更不必担心跨浏览器问题。

15.3.2 尽量使用复选框、单选按钮和下拉式选项

另一条开发 Web 应用程序应该遵循的简单规则是：尽量减少用户在 Web 表单中输入自由文本的机会。尽量使用下拉选项、复选框或单选按钮，而非开放的文本字段。这样不仅能确保得到安全的数据，还能保证数据的一致性和可靠性。

几年前，我曾清理过一个数据库表，其中的数据大多来自客户（航空工程师）使用的一个表单。表单中的所有输入项都是通过开放文本收集的，其中有一个数据字段需要用户输入零件标识信息，但并不对输入内容做任何验证，这正是噩梦的根源。

工程师决定将该字段当做“备注或任何其他内容”来使用，因为表单中恰好没有设置这种字段。最终我发现这个字段中保存的数据范围非常广，从零件标识到一个工程师设置的提醒信息（与供应商的午餐预订）。这些信息读起来非常有趣，但对这个公司并没有什帮助。这种数据也是非常难以清理的，因为来自不同供应商的部件编号格式并不相似，以至于我们也很难用正则表达式来清理数据。

这是一个非故意伤害的例子。而在上一小节中，我们曾经将一条删除数据库的 SQL 语句附加在了 wsehame 后面，这是一个故意伤害的例子。

如果你必须使用一个自由文本来取得用户输入，例如在用户登录系统时需要输入用户名，那么你就需要先对输入信息进行预处理，然后再使用它做更新或查询操作。

15.3.3 使用 node-validator

如果你的应用程序必须通过文本输入方式取得数据，那么在使用数据前，务必要对该数据进行有效性验证和预处理。node-mysql 模块提供了 client.escape 方法，它能对传入的文本进行编码以防止潜在的 SQL 注入攻击。你还可以禁用那些具有潜在破坏性的功能。在第 10 章有关 MongoDB 的讨论中，我曾提到过如何标记一个应

该被序列化存储的 JavaScript 函数。

你还可以使用验证工具来确保传入的数据是安全和一致的。node-validator 就是一个比较优秀的验证工具。

使用 npm 安装 node-validator：

```
npm install node-validator
```

该模块对外暴露两个对象：check 和 sanitize：

```
var check = require('validator').check,
    sanitize = require('validator').sanitize;
```

你可以通过上述两个对象来检查输入数据的格式是否符合使用要求，如检查输入文本以确保它是一个电子邮件格式：

```
try {
   check(email).isEmail();
} catch (err) {
   console.log(err.message); // Invalid email
}
```

当数据无法匹配期望格式时，node-validator 会抛出一个错误。如果想让错误信息更加可读，你可以自定义错误信息并将其作为 check 方法的第二个可选参数传入：

```
try {
   check(email, "Please enter a proper email").isEmail();
}catch (err) {
   console.log(err.message); // Please enter a proper email
}
```

而 sanitize 过滤器可以用来检测数据并防止特定类型的攻击：

```
var newstr = sanitize(str).xss(); // prevent XSS attack
```

示例 15-6 使用了 check 和 sanitize 方法处理三种不同的字符串。

示例 15-6 使用 node-validator

```
var check = require('validator').check,
    sanitize = require('validator').sanitize;

var email = 'shelleyp@burningbird.net';
var email2 = 'this is a test';

var str = '<SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT>';
```

```
try {
   check(email).isEmail();
   check(email2).isEmail();
} catch (err) {
   console.log(err.message);
}

var newstr = sanitize(str).xss();
console.log(newstr);
```

运行该应用程序的结果是：

```
Invalid email
[removed][removed]
```

还有 Express 中间件支将 node 验证器：express-validator。当我们将其包含到 Express 应用程序中：

```
var expressValidator = require('express-validator');
...
app.use(expressValidator);
```

可以直接在请求对象上访问 check、sanitize 以及其他提供的方法。

```
app.get('/somepage', function (req, rest) {
   ...
   req.check('zip', 'Please enter zip code').isInt(6);
   req.sanitize('newdata').xss();
   ...
});
```

15.4 在沙箱中执行代码

Node 中的 vm 模块可以提供一个沙箱并安全地执行 JavaScript 代码。它能虚拟一个全新的 V8 环境，并通过参数传入 JavaScript 代码然后运行。

提示

沙箱通常意味着隔离代码，使其不能做执行任何有害操作。

使用 vm 的方法很多。首先我们可以使用 vm.createScript 方法并传入一段脚本作为方法参数。vm 模块会编译它并返回一个脚本对象：

```
var vm = require('vm');
var script_obj = vm.createScript(js_text);
```

然后，你可以在一个单独的上下文中运行脚本并将脚本执行所需要的任何数据作为可选对象传入：

```
script_obj.runInNewContext(sandbox);
```

示例 15-7 的代码较少，但是麻雀虽小五脏俱全，它使用 vm 编译了一段 JavaScript 语句，还使用了两个沙箱对象属性，并且创建了第三个。

示例 15-7　使用 vm 模块在沙箱中执行脚本

```
var vm = require('vm');
var util = require('util');

var obj = { name: 'Shelley', domain: 'burningbird.net'};

// compile script
var script_obj = vm.createScript("var str = 'My name is ' + name + ' at ' + domain",
                                 'test.vm');

// run in new context
script_obj.runInNewContext(obj);

// inspect sandbox object
console.log(util.inspect(obj));
```

运行应用程序后返回如下输出：

```
{ name: 'Shelley',
  domain: 'burningbird.net',
  str: 'My name is Shelley at burningbird.net' }
```

obj 对象是应用程序和沙箱脚本之间的连接点。脚本无法通过其他方式访问到父上下文环境。如果你尝试在脚本中使用全局对象，例如 console，那么程序会报错。

为了能更好地进行说明，示例 15-8 修改了示例 15-7 的代码，从文件中加载脚本并运行。被加载的脚本文件内容与前面例子中使用的脚本内容基本一致，只是增加了一句对 console.log 的调用：

```
vars tr = 'My name is ' + name + ' from ' + domain;
 console.log(str):
```

vm.createScript 不支持直接从文件中加载脚本。其第二个（可选）参数并不是文件名，而是一个名字标签，用于在调试时输出堆栈调用信息。因此，我们将使用文件系统模块提供的 readFile 方法来读取脚本文件的内容。

示例 15-8　从文件加载脚本，并使用 vm 模块在沙箱中执行

```
var vm = require('vm');
var util = require('util');
var fs = require('fs');

fs.readFile('suspicious.js', 'utf8', function(err, data) {
```

```
  if (err) return console.log(err);

  try {

    console.log(data);
    var obj = { name: 'Shelley', domain: 'burningbird.net'};

    // compile script
    var script_obj = vm.createScript(data, 'test.vm');

    // run in new context
    script_obj.runInNewContext(obj);

    // inspect sandbox object
    console.log(util.inspect(obj));
  } catch(e) {
    console.log(e);
  }
});
```

运行应用程序返回以下内容：

```
[SyntaxError: Unexpected token :]
```

程序发生了错误，不过这也正是我们期望的，因为在虚拟机中并不存在 console 对象。注意它是一个 V8 虚拟机，而不是一个 Node 虚拟机。我们已经看到过 Node 应用程序的子进程可以实现任何处理操作。我们当然不希望沙箱代码也拥有这种权限。

由于我们可以在一个 V8 上下文中运行脚本，这意味着它可以访问 global 对象。示例 15-9 重新修改了示例 15-8 的程序代码，除了使用 runInContext 方法外，还为其传递了一个上下文对象。而这个上下文对象则通过一个包含了执行脚本所需参数的 obj 对象来初始化。在脚本执行后输出 obj 对象的内容，你会发现其中并没有新定义的 str 属性。而检查上下文对象的内容时，却能发现它。

示例 15-9　在 vm 中使用 context 对象运行代码

```
var vm = require('vm');
var util = require('util');
var fs = require('fs');

fs.readFile('suspicious.js', 'utf8', function(err, data) {
  if (err) return console.log(err);
  try {

    var obj = { name: 'Shelley', domain: 'burningbird.net' };

    // compile script
    var script_obj = vm.createScript(data, 'test.vm');
```

```
    // create context
    var ctx = vm.createContext(obj);

    // run in new context
    script_obj.runInContext(ctx);

    // inspect object
    console.log(util.inspect(obj));

    // inspect context
    console.log(util.inspect(ctx));

  } catch(e) {
    console.log(e);
  }
});
```

示例程序使用了预编译脚本块，如果你要多次运行脚本，这会很方便。如果只需要运行一次脚本，你可以直接在 vm 对象上调用 runInContext 和 runInThisContext 方法来执行脚本并关闭虚拟机。不同的是，你必须将脚本作为第一个参数传递给这两个方法：

```
var obj = { name: 'Shelley', domain: 'burningbird.net' };
// create context
var ctx = vm.createContext(obj);

// run in new context
vm.runInContext(data,ctx,'test.vm');

// inspect context
console.log(util.inspect(ctx));
```

同样的，我们通过 createContext 创建上下文，然后使用数据对其初始化，沙箱代码就能够访问上下文中的 global 对象。而且在沙箱代码执行完毕后，所有的处理结果都能够从上下文对象中获得。

第 16 章

扩展和部署 Node 应用

现在，你希望把你的 Node 应用从开发和测试环境转移到产品环境上去。这个过程可能很简单也可能很复杂，这一切取决于你的程序功能及其提供的服务（或者需要的组件）。

下面我会简要介绍 Node 应用在产品环境上部署的过程和可能遇见的问题。一些要求可能只需要很少的精力就可以完成，比如需要安装 Forever 来确保 Node 程序可以一直运行。其他一些，比如把程序部署到云节点上，可能需要提前规划并且会花费很多时间。

16.1 把你的节点部署到服务器上

将程序从开发环境转移到产品环境并不是特别复杂，但是需要做一些部署的准备，以确保程序部署之后发挥最佳性能以及将潜在风险降到最低。

部署 Node 应用的一些前提条件：

- 程序必须通过用户和开发人员的双重测试；
- 需要安全地部署程序，确保协调好修改和修复；
- 应用程序必须安全；
- 必须确保如果有意外发生程序会自动重启；
- 需要将 Node 应用与其他服务器集成，比如 Apache；

- 需要监控程序性能，并且在性能下降时可以调整程序参数；
- 必须充分利用服务器资源；

第 14 章讲到了单元测试、验收测试和性能测试，第 15 章中讲到了安全问题。在这里，我们学习一下其他部署 Node 程序到你自己的产品环境服务器中的必要操作。

16.1.1 编写 package.json 文件

每个 Node 模块都有一个 package.json 文件，包含该模块的基本信息，以及模块可能需要的依赖。在第 4 章中讨论模块时我曾提到过 package.json 文件。现在，我们来进一步研究该文件，特别是当你需要使用它来进行部署时。

从文件名可以看出，package.json 肯定是某种 JSON。可以运行 npminit 来跃过 package.json 过程。我在第 4 章中运行 npminit 命令时并没有提供任何依赖，但是绝大多数的 Node 程序都有自己的依赖。

我们在本书之前几章中创建的 widget 程序尽管很小，但是作为应用程序的一个范例，它是我们考虑部署时一个很好的例子。那么，它的 package.json 文件是怎样的呢？

提示

我并没有涉及 package.json 文件中的全部数据，只涉及了那些对 Node 程序来说有意义的数据。

开始前，我们需要提供应用程序的基本信息，包括程序名称、版本、主要开发人员：

```
{
  "name": "WidgetFactory",
   "preferGlobal": "false",
   "version": "1.0.0",
   "author": "Shelley Powers shelley.just@gmail.com (http://burningbird.net)"
   "description": "World's best Wdiget Factory",
```

注意到 name 属性的值中不能有空格。

Author 属性的值也可以划分为不同部分，如下：

```
  "author": { "name": "Shelley Powers",
          "email": shelley.just@gmail.com,
           "url": http://burningbird.net},
```

不过 author 属性使用单一的值比较简单。

如果程序还有其他的开发人员，你可以将他们列为数组，作为 contributors 关键词的值。每个人的信息与 author 相同。

如果 Widget Factory 程序有二进制码程序，可以放在 bin 属性中。一个使用 bin 目录的例子是 Noadload，在 package.json 文件中表现为：

```
"bin" : {
      "nodeload.js": "./nodeload.js",
      "n1.js": "./n1.js"
          },
```

这段配置告诉我们的内容是当模块全局安装时，可以输入 n1.js 来运行该 Node 程序。

Widget 应用并没有命令行工具，也没有任何脚本。Scripts 关键词表示在包生命周期内运行的所有脚本。在生命周期内可能发生的事件包括 preinstall、install、publish、start、test 和 updtae 等，每个都可以有自己的脚本。

如果在 Node 程序或者模块的目录中输入以下 npm 命令：

```
npm test
```

test.js 脚本会执行：

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

在 widget 程序的 scripts 中除了安装必须的脚本（比如设置应用程序环境的脚本）之外，还应该包含单元测试的脚本。尽管 Widget Factory 暂时还没有启动脚本，但你的程序应该有，特别是如果程序准备部署在云服务上的时候（下一节讨论这个问题）。

如果你没有任何 scripts 的值，npm 提供默认值。对启动脚本来说，如果程序包的根目录下有 server.js 文件，Node 程序默认的启动脚本为：

```
node server.js
```

repository 属性提供了程序所使用的源代码版本管理工具的信息，其中的 url 属性表示源代码的链接（如果源代码公开的话）：

```
"reposotory": {
```

```
      "type": "git",
       "url": https://github.com/yourname/yourapp.git
    },
```

repository 属性只有在公布程序源代码（尽管你还是可以限制特定用户组访问）时才比较重要。提供 repository 属性的一个优点是用户可以通过 npm docs 访问你的文档：

```
npm docs packagename
```

在我的 Ubuntu 系统中，首先设置浏览器的配置选项为 Lynx：

```
npm config set browser lynx
```

然后我打开第 15 章中讲到的认证模块 Passport 的文档：

```
npm docs passport
```

repository 属性的设置有助于 npm 找到文档。

Package.json 文件的一个更重要的设计是指出什么版本的程序可以运行，可以用 engine 属性描述这一信息。在 Widget Factory 的例子中，发布的 0.6.x 和 0.8.2 通过测试为稳定版本，意味着之后的 0.8 版本也可以工作。抱着好的希望，设置 engine 选项为：

```
"engines": {
  "node": ">= 0.6.0 < 0.9.0"
},
```

widget 程序在开发和产品环境中都有一些依赖。这些都会单独列出来，前者的依赖在 devDependencies 中，后者在 dependencies 中。每一个模块依赖都作为一个属性，其版本作为对象的值：

```
 "dependencies" : {
   "express" :  "3.0" ,
    "jade" :  "*" ,
    "stylus" :  "*" ,
    "redis" :  "*" ,
    "mongoose" :  "*"
},
"devDependencies" : {
    "nodeunit" :  "*"
}
```

如果有关于操作系统或者 CPU 的依赖，也可以在这里列出来：

```
"cpu": ["x64", "ia32"],
"os": ["darwin", "linux"]
```

还有一些公开的值，包括 private，确保程序不是意外发布的：

```
"private": "true"
```

同时，publishConfig 用于设置 npm 配置的值。

截止目前我们所做的这些，Widget Factory 的 package.json 文件如示例 16-1 所示：

示例 16-1　Widget Factory 应用的 package.json 文件

```
{
  "name": "WidgetFactory",
  "version": "1.0.0",
  "author": "Shelley Powers <shelley.just@gmail.com> (http://burningbird.net)",
  "description": "World's best Widget Factory",
  "engines": {
    "node": ">= 0.6.0"
  },
  "dependencies": {
    "express": "3.0",
    "jade": "*",
    "stylus": "*",
    "redis": "*",
    "mongoose": "*"
  }, "devDependencies": {
    "nodeunit": "*"
  },
  "private": true
}
```

我们可以将 Widget Factory 的代码复制到一个新的目录，然后输入 npm install –d 来查看是否可以安装所有的依赖以及程序是否运行。通过这种方式可以测试 package.json 文件。

16.1.2　使用 Forever 让你的应用“永不掉线”

尽全力使你的程序完美。尽管对程序进行了严格的测试，添加了错误处理来管理错误信息，但是，仍然可能出现其他问题，比如你没有预料到的事情使程序终止或者退出。如果这种情况发生，你需要确保有一种方式可以重启程序，即使在你没注意到的情况下。

Forever 就是这样一个工具。它确保在程序崩溃后重启该程序。它也是一种按守护

进程模式在当前终端会话后台启动程序的方式。

Forever 既可以作为命令行也可以集成到程序中作为程序的一部分。如果在命令行上使用，需要全局安装：

```
npm install forever -g
```

然后不再直接通过 Node 启动程序，而是通过 Forever 启动：

```
forever start -a -l forever.log -o out.log err.log httpserver.js
```

这个命令会运行一个 httpserver.js 的脚本，指定 Forever 的日志文件，输出日志以及错误日志，如果日志文件存在的话也会告诉程序将日志条目添加到现有文件里。

如果脚本运行出现问题导致程序崩溃，Forever 会重启程序，同时也可以保障即使关掉启动程序的终端窗口，Node 程序也可以一直运行。

Forever 有选项和动作。命令行中的 start 值是一个动作。所有可用的动作如下：

start

启动脚本。

stop

终止脚本。

stopall

终止所有脚本。

restart

重启脚本。

restartall

重启所有运行中的 Forever 脚本。

cleanlogs

删除所有日志内容。

logs

列出 Forever 所有进程的所有日志。

list

列出所有运行的脚本。

config

列出所有用户配置。

set<key><val>

设置配置键值对。

clear<key>

清除配置键值对。

logs<script|index>

追踪<script|index>的日志。

columns add <col>

在 Forever 列表输出中添加一列。

columnsrm<col>

从 Forever 列表输出中删除一列。

columns set <cols>

设置 Forever 输出的所有列。

httpserver.js 作为 Forever 守护进程运行，查看后输出的内容如下：

```
info:    Forever processes running
data:       uid command script    forever pid logfile      uptime
data:    [0] ZRYB node    httpserver.js 2854 2855/home/examples/.forever/
 forever.log 0:0:9:38.72
```

同样还有相当多的选项，包括讲到过的日志文件的设置，运行脚本（-s 或者--silent），打开 Forever 动态输出（-v 或者--verbose），设置脚本源代码目录（--sourceDir），还有其他很多。可以通过输入 forever –help 来查看具体内容。

还可以在程序中集成 Forever，如程序文档中介绍的一样：

```
var forever = require('forever');

  var child = new (forever.Monitor)('your-filename.js', {
    max: 3,
    silent: true,
    options: []
  });

  child.on('exit', this.callback);
  child.start();
```

还可以在 Nodemon（14 章中有介绍）中使用 Forever，不仅可以在出现异常时重启程序，还可以保证代码更新时刷新程序。在 Forever 中使用 Nodemon 很简单，需要 --exitcrash 参数确保在程序崩溃时 Nodemon 退出，将控制权交给 Forever：

```
forever nodemon -exitcrash httpserver.js
```

如果程序崩溃，Forever 会启动 Nodemon，Nodemon 会按顺序执行 Node 脚本，不仅确保了如果源代码变化运行脚本会更新，并且保证了程序不会因为意料之外的错误而永久下线。

如果希望重启时程序会自动启动，则需要将其设置为守护进程（daemon）。Forever 提供的例子中有一个 initd-example。这个例子是在系统重启时用 Forever 运行你的程序。你需要修改该脚本来匹配你的环境，同时将其放在/etc/init.d 目录下。一旦完成之后，即使系统重新启动，程序也可以不需要干涉而自动重启。

16.1.3 使用 Node 和 Apache

本书中所有例子启动时的端口都是默认的网络服务端口 80，也有其他一些端口如 3000 或者 8124。在我的系统中我需要使用另一个端口，因为 Apache 是通过 80 端口处理 web 请求的。人们在访问一个网站的时候并不希望指定端口。我们需要的是让 Node 程序与其他服务器并存，比如 Apache、Nignx。

如果系统运行 Apache 并且你没办法修改 Apache 的端口，可以使用.htaccess 文件来重写对 Node 的 Web 请求，将这些请求重定向到合适的端口，而用户对这一过程并不知情：

```
<IfModule mod_rewrite.c>
     RewriteEngine on

     # Redirect a whole subdirectory:
     RewriteRule ^node/(.+) http://examples.burningbird.net:8124/$1 [P]
</IfModule>
```

如果你有权限，还可以为 Node 程序创建一个子域名并将 Apache 所有请求代理到 Node 程序。这是这类型问题在其他环境中的解决办法，比如同时拥有 Apache 和 Tomcat：

```
<VirtualHost someipaddress:80>
    ServerAdmin admin@server.com
    ServerName examples.burningbird.net
    ServerAlias www.examples.burningbird.net

    ProxyRequests off

    <Proxy*>
        Order deny,allow
        Allow from all
    </Proxy>
    <Location/>
        ProxyPass http://localhost:8124/
        ProxyPassReverse http://localhost:8124/
    </Location>
</VirtualHost>
```

这种方式可以工作。如果不想你的 Node 程序被频繁访问，这种方式下服务器性能很不错。这两种实现方式的共同问题在于所有的请求都需要通过 Apache，对每一个请求，Apache 都需要创建进程来进行处理。Node 的重点在于避免这部分开销过大。如果希望你的 Node 程序被大量使用，另一种实现方式也可以完成，但是这种方式需要对系统有 root 权限。修改 Apache 的 ports.conf 文件，将 Apache 监听的端口从：

```
Listen 80
```

改为你想要的端口，比如 78：

```
Listen 78
```

然后使用 Node 代理，比如 http-proxy，监听请求并将请求代理到适当的端口。比如，如果 Apache 处理所有对子目录 public 的请求，Node 处理所有对 node 的请求，需要创建一个独立的代理服务器根据情况处理接收的请求：

```
var httpProxy = require('http-proxy');

var options = {
router: {
   'burningbird.net/public_html': '127.0.0.1:78',
   'burningbird.net/node': '127.0.0.1:3000'
}
};
```

```
var proxyServer = httpProxy.createServer(options);
proxyServer.listen(80);
```

用户永远也不会看到表面之后的端口转换。Http-proxy 模块也可以用于 WebSocket 请求和 HTTPS。

为什么要继续使用 Apache 呢？因为类似 Drupal 的应用程序使用.htaccess 文件控制对内容的访问。并且，在我的网站中的一些子域名使用.htpasswd 来对内容进行密码保护。这些都是 Apache 的设计，Node 服务器程序并没有等价的概念。

我们从很久以前就开始使用 Apache 了。在 Node 程序中想要弃置 Apache 比用 Express 创建一个静态服务器要复杂多了。

16.1.4 改善性能

根据系统资源，你还可以采取一些步骤来提高 Node 应用的性能。这些操作并不麻烦，但不在本书介绍范围之内。

如果你的系统是多核的并且希望尝试实验技术，可以使用 Node 集群（Node clustering）。Node.js 文档包含一个集群的例子，尽管所有的进程都在同一个端口上监听请求，但是每个进程都分派在不同的 CPU 上。

在未来的 Node 版本中，也许我们可以通过在启动程序时传递一个 balance 参数来自动利用多核环境。

你还可以使用分布式计算的架构，比如 hook.io 模块。

有很多技巧和技术可以用来提高 Node 程序性能，大部分需要花费一定的工作量。所以，你可以将程序部署在云服务上，以充分利用云服务提供的性能改善。

16.2 部署到云服务

现在越来越受欢迎的方式是将应用程序放在云服务上而不是自己的主机上。这样做的原因可能有很多，比如以下几个：

- 安全性更强（就像有自己的安全团队一样）；
- 24 小时监控（所以你可以休息了）；
- 即时扩展（如果程序突然到达高峰，服务器也不会宕机）；

- 花费（云服务器上的主机比你自己的服务器更便宜）；
- 部署工具（云提供了简化 Node 程序部署的工具）；
- 很潮（列表中唯一一个不是为了把 Node 应用部署到云服务上的原因）。

当然，凡事必有两面性。云服务的一个缺点在于你对程序的操作会受到一定限制。比如，你的程序需要使用类似于 ImageMagick 的工具，但大部分云服务没有安装这个工具或者不允许安装。再比如，程序是基于 Node 6.x（或者 8.x 或者其他），而云服务可能仅仅设置了另外一个版本（如 4.x）。

将程序部署在云上也是件挺麻烦的事情。一些云服务提供了部署的工具，只需要输入目标和点下按钮。而还有一些，则需要大量的准备工作——说到准备工作，我必须说明，可能有也可能没有容易阅读的文档。

在最后一节中，我会简要介绍一些常用的可以提供 Node 程序宿主的云服务，并且会介绍各自不同的特点。

16.2.1 通过 Cloud9 IDE 部署到 Windows Azure

如果你的开发环境是基于 Windows 的，并且你之前使用过 Windows 的工具（比如用.NET 开发程序），那么你一定希望将 Node 程序部署到 Windows Azure。为了简化将 Node 程序部署到 Azure 的过程，可以使用 Cloud9 IDE（集成开发环境，Integrated Development Enviroment）来进行部署。

Cloud9 是一个基于 web 的 IDE，相对于其他 IDE 来说，可以与你的 GitHub 账户交互。当你打开程序时，可以看到程序管理接口，如图 16-1 所示。

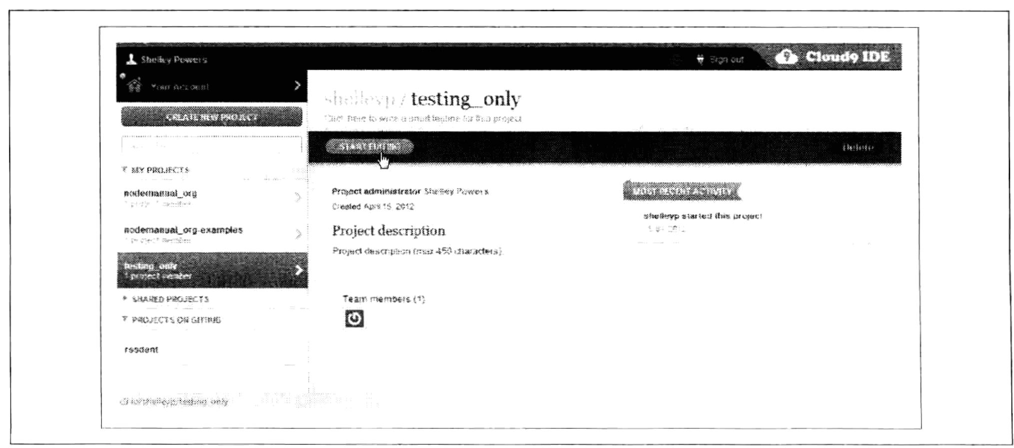


图 16-1　Cloud9 IDE 程序管理页面

在程序管理页面上，以独立页面方式打开一个程序，可以选择任何程序文件进行修改，如图 16-2 所示。可以在 IDE 中直接复制一个 GitHub 上的工程。

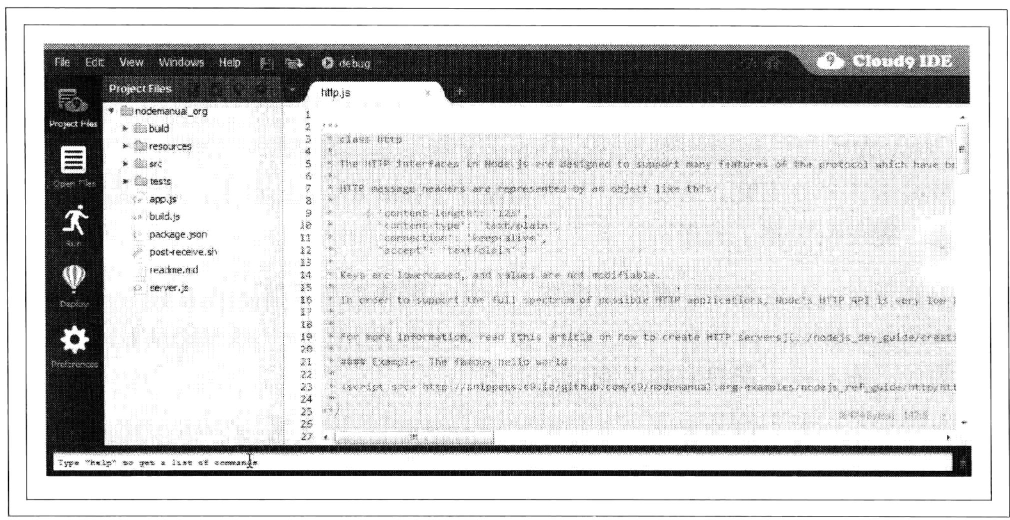


图 16-2　Cloud9 IDE 程序编辑页面

可以添加并且编辑文件，然后在 IDE 中直接运行程序，Cloud9 也支持调试功能。

Cloud9 IDE 对于程序开发是免费的，但是如果需要部署则需要注册交费。Cloud9 支持很多种语言，主要是 HTML 和 Node 程序，同时也支持多种程序源代码仓库，包括 GitHub、Bitbucket、Mercurial repository、Git repository 和 FTP 服务器。

Cloud9 IDE 接口简化了将程序转移到 Azure（还有其他服务——稍后介绍）的过程。如果你已经有了 Azure 账户，部署 Node 程序到 Azure 的过程就仅仅是点击 Deploy 按钮，然后提供弹出对话框所需要的信息。预先提示：首先你需要熟悉 Azure。在提交之前有 90 天的免费使用，可以尝试该服务。

Azure 的费用取决于服务器的节点数，SQL server 数据库节点的大小，blob（二进制大对象）存储的数量以及带宽。Azure 提供了一系列方便阅读的文档以供入门，包括如何在 Azure 上创建 Express 程序的教程。

我之前提到过 Cloud9 IDE 可以部署在不同的云服务上，目前支持以下三种：

- Windows Azure
- Heroku

- Joyent

接下来我会介绍 Joyent Development SmartMachine 和 Heroku。

16.2.2 Joyent Development SmartMachine

JoyenSmartMachine 是虚拟机，可以运行在 Windows 或者 Linux 下，可以对运行 Node 程序进行预编和优化。Joyent 开发人员还提供了 Node.js Development SmartMachine，使 Node 开发人员可以免费在云服务上部署他们的程序。如果你已经准备好产品上线了，则可以升级为产品环境。

Joyent 提供了关于如何使用 Node.js Development SmartMachine 的详细文档，包括以下步骤：

1. 创建一个 Joyent 云服务的账户；
2. 如果没有的话创建一个 SSH key；
3. 更新~/.ssh/config 文件为你的机器配置端口号；
4. 用 Git 部署程序到 SmartMachine；
5. 确保程序包含 package.json 文件，并指定启动脚本。

再强调一次，Node.js Development SmartMachine 仅用于开发。

那么，Joyent Development SmartMachine 有什么用呢？好吧，在开始阶段，没有前期成本。这是很明智的一步，程序员有机会尝试云主机而无需支付可观的费用。

Joyent 还提供简化的 Git 部署，可以同时部署到多个机器，npm 可以支持管理程序的依赖。

16.2.3 Heroku

我喜欢不需要任何费用就可以试用的云服务，比如 Heroku 账户就是免费并且即时的。如果你决定使用 Heroku 作为你的产品系统，像 Azure 一样需要一些配置。Heroku 的云服务器有详细的文档，也提供了可以在你的开发环境安装的工具来简化部署到 Heroku 的过程（如果你没有使用 Cloud9 IDE）。

Heroku 云服务还带有一些预先打包好的插件，可以添加到自己的账户，包括对我最喜欢的数据存储 Redis 的支持。Heroku 对插件管理得很好，并且很多插件都是免费的。

之前提到过的 Heroku 的文档是云服务器中文档最详尽的服务器之一，并且开发工具也很大程度上简化了部署过程。创建好应用程序，编写 package.json 文件并列出依赖，通过一个简单的 Procfile（内容类似 web：node app.js）声明一个进程类型，然后使用 Heroku 工具套件中的某个工具启动程序。

部署时，将程序提交到 Git，然后用 Git.Simple 部署程序。

16.2.4 Amazon EC2

Amazon Elastic Compute Cloud，简称 EC2，有一定的历史背景使它成为一个很受欢迎的选择。在 EC2 上部署 Node 程序对开发人员也没有太多要求。

设置 Amazon EC2 与设置传统的 VPN（虚拟专用网络，Virtual Private Network）没有太大区别。指定需要的操作系统，安装运行 Node 程序必须的软件，用 Git 部署程序，然后使用类似 Forever 的工具确保程序一直在线。

Amazon EC2 服务提供了网站，可以简化设置节点的过程。EC2 并不像 Joyent 一样免费，但是收费比较合理，大概是每小时 0.02 美元。

16.2.5 Nodejitsu

Nodejitsu 目前还是测试版，并且提供了测试账户。它像很多其他优秀的云服务一样提供了免费试用。

Nodejistu 像 Heroku 一样提供了简化部署过程的工具：jistu。可以使用 npm 安装，用 jitsu 登录到 Nodejitsu，输入以下命令就可以进行部署了：

```
Jitsu deploy
```

Nodejistu 从 package.json 文件中获取所需要的信息，提示一些小的问题，然后就可以了。

Nodejistu 也提供了自己的基于 Web 的 IDE，但是我还没有试用过。看起来确实比 Cloud9 IDE 容易很多。

附录

Node、Git 和 GitHub

Git 是一个版本控制系统，类似于 CVS（Concurrent Versioning System）或者 Subversion。Git 与其他更传统的版本控制系统相比较，不同之处在于当你修改代码时如何维护源代码。类似于 CVS 的版本控制系统在版本变化时，将变化作为与原始文件不同的部分进行存储。而 Git，则是存储了代码在某个特定时间点的快照。如果文件没有修改，Git 则继续链接到之前的快照上。

开始使用 Git 前，你需要在你的系统中进行安装。对 Windows 和 MAC OS 有二进制文件，Unix 系统可以通过源代码安装。在我的 Linux 服务器（Ubuntu 10.04）上安装 Git 只需要一个命令：

```
sudo apt-get install git
```

所有的依赖都会自动下载并且安装。

提示

之后输入的命令行都是假定你在使用基于 Unix 的操作系统。Git 在 Windows 操作系统中有图形界面。你需要根据接口文档来配置你的系统，但是所有环境中这一基本过程是相同的。

安装完 Git 之后需要一些配置。你需要提供用户名（一般来说姓和名字）和 E-mail 地址。这两部分是 commit author（提交作者）的组成部分，用于标记你的修改：

```
git config --global user.name "your name"
git config --global user.email "your email"
```

接下来你需要使用 GitHub，提供大部分（如果不是全部的话）Node 模块，你还需

要创建 GitHub 账户。你可以使用任何你想用的 GitHub 用户名——并不一定和刚指定的用户名一样。你还需要根据 GitHub 帮助文档中的过程为 GitHub 生成 SSH key。

大部分的 Git 教程都是以创建一个简单的 repository（更一般的说法是 repo）作为开始的。但是我们感兴趣的部分是如何在 Git 中提供 Node 程序服务，我们复制一个现成的 repo 而不需要创建自己的。在你复制源代码之前，首先你需要在 GitHub 网站的 repository 主页中点击右上角的 fork 按钮来 fork 一个 repository（包含一个项目快照），如图 A-1 所示。

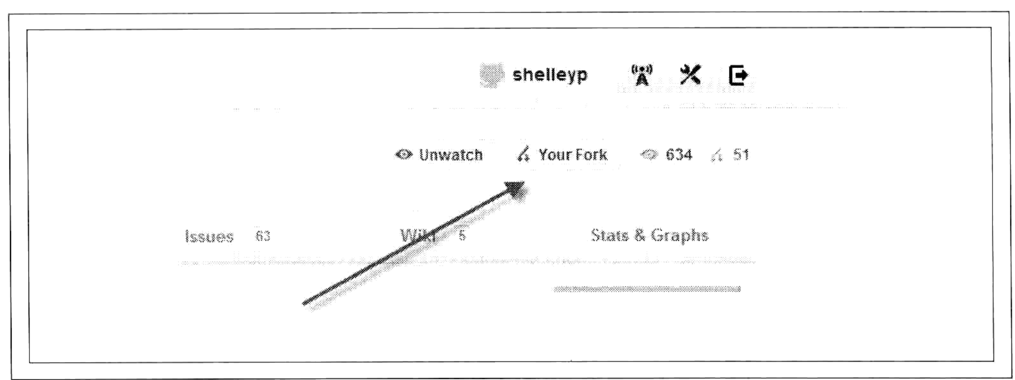


图 A-1　在 GitHub 上 fork 一个现有的 Node 模块

然后你就可以在自己的主页中访问 fork 的 repository，还可以在新 fork 的 repository 网页上访问 Git URL。例如，我 fork 了 node-canvas 模块（第 12 章中讲到过），URL 为 git@github.com:shelleyp/node-canvas.git。在 git 中复制 fork 的 repository 命令为 git clone URL：

```
git clone git@github.com:shelleyp/node-canvas.git
```

你还可以通过 HTTP 复制，但是 GitHub 并不推荐这一做法。不过如果你希望引入在用 npm 安装模块可能没有包含的例子以及其他内容时，这是一种很好的实现方式，可以提供一个源代码的只读版本。

可以从每个 repository 的网页上获取只读的 URL，对 node-canvas 来说命令如下：

```
git clone https://github.com:username/node-whatever.git
```

提示

你可以通过制定 Git URL 的方式安装模块：

```
npm install git://github.com/username/node-whatever.git
```

现在，你已经有了一份 node-canvas（或者其他你想要的 repository）代码的复本。你可以修改任意你想修改的源文件。通过 git add 命令添加新增或者修改后的文件，然后通过 git commit 提交修改（-m 参数可以指定提交时的信息，比如做了哪些修改）：

```
git add somefile.js
git commit -m 'note about what this change is'
```

如果你想要查看当前状态是否已经可以提交，可以输入以下命令：

```
git status
```

如果你希望自己的修改可以添加到被 fork 的原 repository 中，你需要发起 pull request。在浏览器中打开你 fork 出来的 repository 主页，找到 Pull Request 按钮，如图 A-2 所示。

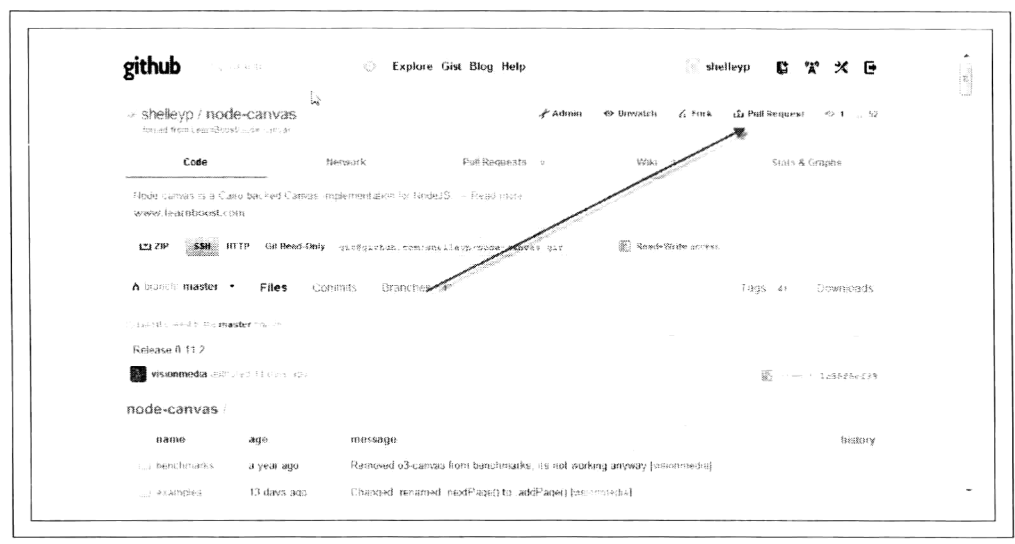


图 A-2　在 GitHub 上点击 Pull Request 按钮来发起 Pull Request

点击 Pull Request 按钮会打开一个 Pull Request 预览页面，你需要输入用户名和对修改的描述，以及哪些需要提交。这时候你可以修改提交范围和目标 repository。

完成之后发送请求。这样会使 Pull Request 中的提交进入队列等待原作者合并。原作者可以查看修改，讨论一下相关内容，如果决定接受请求，会进行 fetch 操作合并修改、打补丁或者自动合并。

提示

GitHub 有文档介绍如何合并修改，以及使用 Git 的其他功能。

如果你创建了自己的模块并希望与别人分享，你需要创建一个 repository。可以通过 GitHub 来做，点击 GitHub 网页中的 New Repository 按钮，然后输入模块名，指定 repository 访问权限为 public（公开）或者 private（私有）。

用 git init 来初始化一个空的目录：

```
mkdir ~/mybeautiful-module
cd ~/mybeautiful-module
git init
```

可以用你最喜欢的文本编辑器为 repository 提供一个 README 文件。当用户点击 GitHub 模块页面上的 Read More 的时候会显示该文件内容。创建好文件之后，添加并提交：

```
git add README
git commit -m 'readme commit'
```

为了将本地 repository 链接到 GitHub，你需要为模块建立一个远端 repository，然后 push 到远端 repo：

```
git remote add origin git@github.com:username/Mybeautiful-Module.git
git push -u origin master`
```

一旦将新的模块 push 到 GitHub，你就可以开始进行推广来确保模块可以被列在 Node 模块列表和 npm registry 中。

你可以在 GitHub 官网上 Help 链接下找到快速完成的文档。

作者简介

Shelley Powers 从事和编写 Web 技术内容超过 12 年了，从第一版的 JavaScript 发布到最新的图形设计工具。她最近的 O' Reilly 系列著作涉及语义网，Ajax，JavaScript 和 Web 图形。她是一个充满激情的业余摄影师和 Web 开发狂热者，乐衷于在自己的很多网站上尝试最新的实验技术。

封底图片

Learning Node 封面上的动物是一只仓鼠(Beamys)。仓鼠有两个种类:大仓鼠属(Beamys Major）和小仓鼠属（Beamyshindei)。

仓鼠主要栖息于从肯尼亚到坦桑尼亚的非洲丛林。这种大型啮齿类动物喜欢在潮湿的环境安家：河畔或者植被茂密的地区。主要繁衍在海岸或者山地地区，尽管森林砍伐威胁到它们的自然栖息地。仓鼠住在地洞中，非常善于攀爬。

这种啮齿类动物外观非常好辨认：7～12 英尺长，大概三分之一磅。头很短，全身灰色皮毛，肚皮是白色的，黑白斑驳的尾巴。和其他啮齿类动物一样，仓鼠的食物非常多样化，它们使用颊囊作食物存储。