提供各种IT类书籍pdf下载，如有需要，请QQ:2404062482

注：链接至淘宝，不喜者勿入！整理那么多资料也不容易，请多多见谅！非诚勿扰！

更多此类书籍

Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional Node.js: Building JavaScript-Based Scalable Software

Node.js高级编程

[美] Pedro Teixeira 著
胡训强 张欣景 译

清华大学出版社

Node.js 高级编程

[美] Pedro Teixeira 著

胡训强 张欣景 译

清华大学出版社

北 京

Pedro Teixeira

Professional Node.js: Building JavaScript-Based Scalable Software

EISBN：978-1-118-18546-9

Copyright © 2013 by John Wiley & Sons, Inc., Indianapolis, Indiana.

All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2013-5115

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

Node.js 高级编程/(美)特谢拉(Teixeira,P.) 著；胡训强，张欣景 译；—北京：清华大学出版社，2013

书名原文：Professional Node.js:Building JavaScript-Based Scalable Software

ISBN 978-7-302-34441-4

Ⅰ.①N…　Ⅱ.①特…　②胡…　③张…　Ⅲ.①JAVA 语言－程序设计　Ⅳ.①TP312

中国版本图书馆 CIP 数据核字(2013)第 270034 号

责任编辑：王　军　杨信明

装帧设计：牛静敏

责任校对：成凤进

责任印制：李红英

出版发行：清华大学出版社

网　　址：http://www.tup.com.cn，http://www.wqbook.com

地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084

社 总 机：010-62770175　　**邮　　购**：010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：清华大学印刷厂

装 订 者：三河市溧源装订厂

经　　销：全国新华书店

开　　本：185mm×260mm　　**印　张**：23　　**字　　数**：560 千字

版　　次：2013 年 12 月第 1 版　　**印　　次**：2013 年 12 月第1 次印刷

印　　数：1～3000

定　　价：58.00 元

产品编号：053244-01

译 者 序

Node.js 是一套用来编写高性能网络服务器的 JavaScript 工具包，从 2009 年诞生之日起，就获得了业内专家和技术社区的强烈关注。近两年来，Node.js 在 github 上的访问量已经超过了 Rails，其创始人 Ryan Dalh 加盟 Joyent 并获得了企业资助，Node.js 在 Linux 和 Windows 系统上的版本不断更新，在国外，Yahoo!、Google 和惠普等业界翘楚表示有足够的信心利用 Node.js 来开发下一代产品，而国内的淘宝网已在其新的增值业务产品开发中进行了 Node.js 的实际应用，这一系列事件都表明，Node.js 的成长速度和公众认可都令人振奋，也为开发具有可伸缩性能的 Web 后台应用程序提供了新的途径。

Node.js 是基于 Google 著名的开源 JavaScript 引擎 V8 开发的，V8 在处理 JavaScript 的速度方面是其他浏览器难望其项背的，加上 Node.js 的异步编程特性以及众多的可选模块，使其成为服务器端应用程序开发的利器。正是基于这些原因，国内希望了解和学习 Node.js 的开发人员越来越多，尤其对于那些本来就熟悉 JavaScript 的开发人员更是如此。《Node.js 高级编程》的出现恰好迎合了开发人员的需求，全书共分为 25 章，涉及 Node.js 的安装配置、基本概念、核心模块等基础内容以及安全性、创建 HTTP 中间件、创建实时 WEB 应用程序以及连接数据库等进阶话题，可以说囊括了 Node.js 的方方面面，是一本不可多得的优秀书籍。

读者朋友们需要注意的是，请不要被书名中的“高级”二字所惧。在我们看来，更愿意把这本书当作是有关 Node.js 的一本实用大全，因为不管是初级应用还是高级应用你都可以在本书中找到所需的内容。这本书也没有浓厚的理论化色彩，几乎每个相关知识点阐述完毕后，都有相应的示例进行演示以加深学习的效果，这些示例基本上都来自于 Node.js 的官方网站，所以您也不必对它们的正确性和权威性心存犹疑。相信本书一定能为您带来愉悦的学习体验，即使您已经相当熟悉 Node.js，将本书当作一本案头上的工具书以供随时翻阅也是一个不错的主意。

顾名思义，Node.js 采用的编程语言是 JavaScript，对于已经掌握这门语言的读者而言，您需要做的只是打开本书，从第一章开始逐章阅读，即可轻松掌握 Node.js 的一切。而对于还没有掌握 JavaScript 又迫切希望应用 Node.js 的读者而言，我们建议您还是应先补上这一课，好在 JavaScript 的学习周期并不长，相信您不久就会用上本书。

Node.js 问世的时间不长，为国内所熟悉的时间则更短，我们与各位读者一样，也处于一个不断学习摸索的过程中，在翻译本书期间，我们也确实进一步地感受到了 Node.js 的优良特性。但是囿于技术水平和语言能力，译文中出现错误或遗漏在所难免，所以欢迎您

将阅读过程中发现的问题及时反馈给我们，我们的邮箱是 navalboy@163.com 和 10185014@qq.com，我们会仔细查阅读者发来的每一封邮件，以求进一步提高今后译著的质量。

本书全部章节由胡训强和张欣景翻译，参与翻译活动的还有宗昊璇、曹建、梁捷、张群起，没有他们的共同努力，本书也不能顺利付梓，在此一并向他们表示感谢！

胡训强　张欣景

作 者 简 介

Pedro Teixeira 是一位高产的开源项目程序员，同时也是众多 Node.js 模块的构建者。自从在 14 年前获得软件工程学位后，他从事的职业包括咨询师和程序员，并且他还是世界知名 Node.js 社区的活跃成员。

他是 Node 公司的创始人之一，同时也是 Nodejitsu 公司的高级程序员，Nodejitsu 公司是以 Node.js 平台作为服务提供者的领头羊。此外，他还是受到广泛欢迎的 Node Tuts 视频的制作者。

在 Pedro 10 岁时，他的父亲就开始教他在 ZX Spectrum(第一款在英国流行的家庭电脑)上编写程序，从那时开始，他在编程之路上就一发不可收拾。他在父亲的苹果 IIc 型计算机上开始自学程序设计，在随后的 PC 时代，他依然勤学不辍。Pedro 在大学时代进入了 UNIX 和开源项目的天地，并且对它们十分痴迷。在他的职业生涯中，Petro 为大型电信公司、银行、连锁酒店以及其他企业开发过众多系统和产品，使用的编程语言包括 Visual Basic、C、C++、Java、PHP、Ruby 和 JavaScript。

在 Node.js 发轫之初，Pedro 就是其忠实拥趸，并且他已经使用 Node.js 构建了很多为人熟知的应用程序和模块，例如 Fugue、Alfred.js、Carrier 和 Nock 等。

技术编辑简介

Manuel Kiessling 是一位软件开发和系统管理团队的负责人，他在这两个领域都进行了应用和教学方面的敏捷实践，主持了若干开源项目，他是一名活跃的博主，并且撰写了可免费获取的 Node Beginner Book。现在他与妻子以及两个孩子居住在德国的科隆附近。

他是第 22 章“使用 Socket.IO 创建通用的实时 Web 应用程序”和第 23 章“使用 node-mysql 连接 MySQL”的合著者。

致　谢

首先，我要感谢我的妻子 Susana 以及两个孩子 Henrique 和 Beatriz，你们时刻提醒我什么才是生命中真正重要的东西，并且以此给予我勇气和坚持不懈的毅力。

此外，我要感谢 Wiley 出版集团的策划编辑 Mary E. James 给予我的信任，与我签约让我来完成此书。

我还要感谢好友兼前同事 Pedro Mendes，他是一名程序员和摄影师，在里斯本的一番小酌后，他最终说服了我着手写一本有关 Node.js 的书。

我还要感谢好友 Nuno Job，在过去几年里，他在开源项目和 Node.js 创新方面一直是我的合作伙伴。

总之，还要好好感谢 Node.js 社区，这是我参与过的最好、最受欢迎、欣赏的以及有趣的编程社区。

最后，我要感谢你们——本书的读者，感谢你们购买本书，我希望你们能够通过本书掌握精彩的 Node.js 编程世界的来龙去脉。

前　言

在 1995 年，也就是大学二年级时我开始接触 UNIX 网络编程。在 C 语言中，可以针对服务器的开放 TCP 连接创建套接字，并编写服务器代码来接受连接。我还记得第一次创建 TCP 服务器的兴奋劲：我可以接受连接，并且能在服务器上收发消息。

如果我希望服务器接受大量并发连接，一般的解决方案是使用线程，不久之后我就创建了自己的第一个多线程 TCP 服务器，这个服务器会访问一个共享数据结构，这个数据结构要同步大量产生的客户端线程的访问。事实已经证明，让同步精细化(最大化资源和时间)和正确(避免死锁)要比预计的困难。

几年后，我成为一名顾问，从事编程工作，并领导程序员团队实现不同的客户端项目。一开始我还是继续从事 UNIX 领域的工作，但是不久之后就转而关注 Java 和它的企业化风格，最终落脚于使用 PHP 和 Ruby 这样的脚本语言进行丰富多彩的 Web 开发。在从事 Web 开发时，我慢慢地开始熟悉 JavaScript 和事件驱动编程模型，却从未意识到这会让我再次与 UNIX 世界联系到一起。

时间很快就到了 2010 年初，我的一位好朋友跟我谈到 Node.js，他说："你可以用 JavaScript 对 Node.js 进行快速编程"。Node.js 将事件驱动的浏览器编程引入了 UNIX 网络编程世界。

出于好奇，我翻阅了一下 Node.js 的 API 文档，立刻就被它吸引住了。这种不使用线程就能创建具有高度可伸缩性的服务器的易用性以及混搭的服务器与客户端代码促使我深入研究了 Node 的源代码以及周边模块，Node.js 将脚本语言的易用性与 UNIX 网络编程能力全面结合起来，我感觉自己终于回家了。

本书读者对象

本书是为那些熟练使用 JavaScript 进行浏览器或者服务器编程的开发者编写的，读者应该熟悉有关 TCP 和 HTTP 运行的基本概念。对于后面有关 Web 应用程序开发的几章而言，如果读者熟悉传统的 Web 开发也会大有裨益。

如果你已经安装了 Node.js，就可以跳过第 2 章"Node 简介"。

如果你已经了解了 Node.js 内部运行的基本原理，并具有使用 JavaScript 进行服务器端事件驱动编程的基础，那么可以跳过第 3 章"加载模块"。

在介绍了 Node.js 的核心概念及 API 子集之后，我将会在第 17 章(“测试模块及应用程序”)中开始构建应用程序，并在第 18 章(“调试模块及应用程序”)中介绍调试，此外将在第 19 章(“控制回调流程”)中提出几个控制异步流程的要点。

接下来，我会阐述构建 Web 应用程序时几个必不可少的部分，从第 20 章(“构建和使用 HTTP 中间件”)开始，一直持续到 Express.js(第 21 章)，此外还会用 Socket.IO(第 22 章)来创建实时 Web 应用程序。

最后，将会介绍在 Node 中如何访问和使用数据库，包括 MySQL(第 23 章)、CouchDB(第 24 章)和 MongoDB(第 25 章)。

本书涉及的内容

本书涉及 Node.js v0.8、Express.js v2.5、Socket.io 0.9、Node-mysql v0.9、Nano v3.1 以及 Mongoose v2.7。

本书组织结构

本书从安装和介绍 Node.js 开始。

然后阐述了 Node 的核心基础，包括模块、缓冲区、事件发射器模式以及定时器，之后介绍和阐述了 Node 中有关文件和网络的核心基础 API，这些内容都会辅之以一些实用的示例。

介绍完 Node 的核心概念之后，本书接下来就会用 Node.js 进行以一些应用程序开发的最佳实践，比如测试模块、调试应用程序、维护对异步回调流程的控制。

构建实时 Web 应用程序是 Node 的主要用途之一，本书会向你展示如何使用 Connect、Express.js 和 Socket.IO。

由于大多数应用程序都需要连接数据库，因此本书还讲述了如何在 Node.js 中连接和使用 MySQL、CouchDB 以及 MongoDB。

使用本书的需求

为了安装和运行 Node.js 应用程序，需要一台 PC 计算机或者一台 Macintosh 计算机运行最新版本的 Node.js，Windows 版、Linux 版和 MacOS 版可以任选其一。

书中示例的源代码可以在 Wrox 的网站上下载，网址是：www.wrox.com/remtitle.cgi?isbn=P010093766。

源代码

在读者学习本书中的示例时，可以手动输入所有代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 http://www.wrox.com/或 http://www.tupwk.com.cn/downpage 上下载。登录到站点 http://www.wrox.com/，使用 Search 工具或使用书名列表就可以找到本书。接着单击 Download Code 链接，就可以获得所有的源代码。既可以选择下载一个大的包含本书所有代码的 ZIP 文件，也可以只下载某个章节中的代码。

由于许多图书的标题都很类似，因此按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-1-118-18546-9。

在下载代码后，只需用解压缩软件对它进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。记住，可以使用书中列出的程序清单的编号容易地找到所要寻找的代码，如“程序清单 0-1”。

当为大多数可下载的源代码文件命名时，我们会使用这些清单中的数值。对于那些很少的没有用它自己的清单数值命名的程序清单，它们都与文件名匹配，所以很容易就可以在下载的源代码文件中找到它们。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果您在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

要在网站上找到本书英文版的勘误表，可以登录 http://www.wrox.com，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

如果您发现的错误在我们的勘误表里还没有出现的话，请登录 www.wrox.com/contact/techsupport.shtml 并完成那里的表格，把您发现的错误发送给我们。我们会检查您的反馈信息，如果正确，我们将在本书的勘误表页面张贴该错误消息，并在本书的后续版本加以修订。

p2p. wrox.com

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于您张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给您传送感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 http://p2p.wrox.com 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读使用协议，并单击 Agree 按钮。

(3) 填写加入该论坛所需要的信息和自己希望提供的其他可选信息，单击 Submit 按钮。您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须先加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

要想了解更多的有关论坛软件的工作情况，以及 P2P 和 Wrox 图书的许多常见问题的解答，就一定要阅读 FAQ，只需在任意 P2P 页面上单击 FAQ 链接即可。

目　录

第Ⅰ部分　概述和安装

第 1 章　安装 Node ……3
1.1　在 Windows 上安装 Node ……4
1.2　在 MAC OS X 上安装 Node ……6
1.3　使用源代码安装 Node ……7
1.3.1　选择 Node 的版本 ……7
1.3.2　下载 Node 源代码 ……7
1.3.3　编译 Node ……8
1.3.4　安装 Node ……8
1.3.5　运行 Node ……8
1.4　安装和应用 Node 包管理器 ……9
1.5　本章小结 ……14

第 2 章　Node 简介 ……15
2.1　事件驱动编程风格介绍 ……16
2.2　Node 和 JavaScript 如何简化异步应用程序的编写 ……17
2.2.1　什么是闭包 ……17
2.2.2　闭包如何辅助异步编程 ……18
2.3　本章小结 ……19

第Ⅱ部分　Node 核心 API 基础

第 3 章　加载模块 ……23
3.1　理解 Node 如何加载模块 ……24
3.2　导出模块 ……24
3.3　加载模块 ……25
3.3.1　加载核心模块 ……25
3.3.2　加载文件模块 ……26
3.3.3　加载文件夹模块 ……26
3.3.4　从 node_modules 文件夹加载 ……26
3.3.5　缓存模块 ……27
3.4　本章小结 ……28

第 4 章　应用缓冲区处理、编码和解码二进制数据 ……29
4.1　创建缓冲区 ……30
4.2　在缓冲区中获取和设置数据 ……30
4.3　切分缓冲区 ……31
4.4　复制缓冲区 ……32
4.5　缓冲区解码 ……32
4.6　本章小结 ……33

第 5 章　使用事件发射器模式简化事件绑定 ……35
5.1　理解标准回调模式 ……35
5.2　理解事件发射器模式 ……36
5.3　理解事件类型 ……37
5.4　应用事件发生器 API ……38
5.4.1　使用.addListener( )或.on( )绑定回调函数 ……38
5.4.2　绑定多个事件监听器 ……39
5.4.3　使用.removeListener( )从事件发射器中删除一个事件监听器 ……40
5.4.4　使用.once()使回调函数最多执行一次 ……40
5.4.5　使用.removeAllListeners()从事件发射器删除所有事件监听器 ……41
5.5　创建事件发射器 ……41
5.5.1　从 Node 事件发射器继承 ……41
5.5.2　发射事件 ……42
5.6　本章小结 ……42

第 6 章　使用定时器制定函数执行计划…45
6.1　使用 setTimeout 推迟函数执行……46
6.2　使用 clearTimeout 取消函数执行……46
6.3　制定和取消函数的重复执行计划……47
6.4　使用 process.nextTick 将函数执行推迟到下一轮事件循环……47
6.5　阻塞事件循环……48
6.6　退出事件循环……49
6.7　使用 setTimeout 代替 setInterval 强制函数串行执行……49
6.8　本章小结……50

第III部分　文件、进程、流和网络

第 7 章　查询和读写文件……53
7.1　处理文件路径……54
7.1.1　规范化路径……54
7.1.2　连接路径……54
7.1.3　解析路径……55
7.1.4　查找两个绝对路径之间的相对路径……55
7.1.5　提取路径的组成部分……55
7.1.6　确定路径是否存在……56
7.2　fs 模块简介……57
7.3　打开文件……58
7.4　读取文件……58
7.4.1　写入文件……59
7.4.2　关闭文件……60
7.5　本章小结……62
第 8 章　创建和控制外部进程……63
8.1　执行外部命令……63
8.2　生成子进程……68
8.2.1　创建子进程……68
8.2.2　监听子进程的输出数据……69
8.2.3　向子进程发送数据……69
8.2.4　当子进程退出时获得通知……71
8.3　向进程发送信号并终止进程……72
8.4　本章小结……73
第 9 章　读写数据流……75
9.1　使用可读流……76
9.1.1　等待数据……76
9.1.2　暂停与恢复流……76
9.1.3　了解流何时终止……77
9.2　使用可写流……77
9.2.1　将数据写入流……77
9.2.2　等待流被清空……78
9.3　考虑几个流的例子……78
9.3.1　创建文件系统流……78
9.3.2　理解网络流……80
9.4　避免慢客户端问题以及挽救服务器……80
9.4.1　理解慢客户端问题……80
9.4.2　避免慢客户端问题……81
9.4.3　应用 stream.pipe()避免慢客户端问题与使用 pipe()集成可读流和可写流……81
9.5　本章小结……82
第 10 章　构建 TCP 服务器……83
10.1　创建 TCP 服务器……83
10.1.1　应用套接字对象……85
10.1.2　理解空闲套接字……86
10.1.3　设置保持运行……87
10.1.4　应用延时或非延时……87
10.1.5　监听连接……88
10.1.6　关闭服务器……88
10.1.7　处理错误……88
10.2　构建一个简单的 TCP 聊天服务器……89
10.2.1　接受连接……89
10.2.2　从连接中读取数据……89
10.2.3　聚合所有客户端……90
10.2.4　广播数据……91
10.2.5　删除被关闭的连接……91
10.2.6　使用 TCP 聊天服务器……92
10.3　本章小结……93

第 11 章　构建 HTTP 服务器……95
11.1　理解 http.ServerRequest 对象…96
11.2　理解 http.ServerResponse 对象…98
11.2.1　写入响应头……98
11.2.2　修改或设置响应头……98
11.2.3　删除响应头……99
11.2.4　写入一块响应主体……99
11.3　以流的形式传送 HTTP 分块响应……99
11.3.1　传送文件……99
11.3.2　传送其他进程的输出……100
11.4　关闭服务器……100
11.5　示例 1：构建提交静态文件的服务器……101
11.6　示例 2：使用 HTTP 分块响应和定时器……102
11.7　本章小结……102

第 12 章　构建 TCP 客户端……103
12.1　连接服务器……104
12.2　发送和接收数据……104
12.3　终止连接……105
12.4　处理错误……106
12.5　创建命令行 TCP 客户端的示例……106
12.5.1　连接服务器……107
12.5.2　向服务器发送命令行……107
12.5.3　打印服务器消息……107
12.5.4　在连接终止时重新连接…108
12.5.5　关闭连接……110
12.5.6　前述内容综合……111
12.6　本章小结……112

第 13 章　创建 HTTP 请求……113
13.1　创建 GET 请求……113
13.2　使用其他 HTTP 动词……114
13.2.1　查看响应对象……115
13.2.2　获取响应主体……116
13.2.3　以流的方式传送响应主体……116
13.3　使用 HTTP.Agent 维护套接字池……116
13.4　应用第三方请求模块简化 HTTP 请求……118
13.4.1　安装和应用 request 模块……118
13.4.2　创建测试服务器……119
13.4.3　跟随重定向……121
13.4.4　设置请求选项……122
13.4.5　对请求体进行编码……125
13.4.6　流式传送……126
13.4.7　使用 Cookie Jar……127
13.5　本章小结……127

第 14 章　使用用户数据报……129
14.1　理解用户数据报……129
14.2　理解用户数据报的使用……130
14.3　构建数据报服务器……130
14.3.1　监听消息……130
14.3.2　测试服务器……131
14.3.3　查看附加的消息信息……132
14.4　创建简单的数据报回送服务器……132
14.4.1　等待消息……132
14.4.2　向发送端发回消息……132
14.4.3　前述内容综合……133
14.5　构建数据报客户端……133
14.5.1　创建客户端……134
14.5.2　发送消息……134
14.5.3　关闭套接字……134
14.6　创建一个简单的数据报命令行客户端……134
14.6.1　读取命令行……135
14.6.2　向服务器发送数据……135
14.6.3　从服务器接收数据……135
14.6.4　前述内容综合……136
14.7　理解和使用数据报多播……136
14.7.1　接收多播消息……137
14.7.2　发送多播消息……137
14.7.3　理解数据报最大容量……138
14.8　本章小结……138

第 15 章 用 TLS/SSL 保证服务器的安全性 ····· 139
15.1 理解私钥和公钥 ····· 139
15.1.1 产生私钥 ····· 140
15.1.2 产生公钥 ····· 140
15.2 构建 TLS 服务器 ····· 140
15.2.1 初始化服务器 ····· 141
15.2.2 监听连接 ····· 141
15.2.3 从客户端读取数据 ····· 142
15.2.4 向客户端发送数据 ····· 142
15.2.5 终止连接 ····· 142
15.3 构建 TLS 客户端 ····· 142
15.3.1 初始化客户端 ····· 143
15.3.2 连接服务器 ····· 143
15.3.3 验证服务器证书 ····· 143
15.3.4 向服务器发送数据 ····· 144
15.3.5 从服务器读取数据 ····· 144
15.3.6 终止连接 ····· 144
15.4 创建几个示例 ····· 144
15.4.1 创建 TLS 聊天服务器 ····· 145
15.4.2 创建 TLS 命令行聊天客户端 ····· 146
15.4.3 验证客户端证书 ····· 147
15.5 本章小结 ····· 148

第 16 章 用 HTTPS 保证 HTTP 服务器的安全性 ····· 149
16.1 构建安全的 HTTP 服务器 ····· 149
16.1.1 设置服务器选项 ····· 150
16.1.2 监听连接 ····· 150
16.1.3 验证 HTTPS 客户端证书 ····· 151
16.2 创建 HTTPS 客户端 ····· 152
16.2.1 初始化客户端 ····· 152
16.2.2 创建请求 ····· 152
16.2.3 验证 HTTPS 服务器证书 ····· 153
16.3 本章小结 ····· 154

第Ⅳ部分 构建与调试模块及应用程序

第 17 章 测试模块及应用程序 ····· 157
17.1 应用测试运行工具 ····· 157
17.1.1 编写测试 ····· 158
17.1.2 运行测试 ····· 159
17.2 使用断言测试模块 ····· 159
17.2.1 使用断言模块 ····· 159
17.2.2 使用 Node-Tap 中的内置断言函数 ····· 161
17.3 测试异步模块 ····· 163
17.4 本章小结 ····· 166

第 18 章 调试模块及应用程序 ····· 167
18.1 使用 console.log ····· 167
18.2 使用 Node 内置调试器 ····· 169
18.3 使用 Node 检查器 ····· 173
18.4 本章小结 ····· 175

第 19 章 控制回调流程 ····· 177
19.1 理解飞去来器效应 ····· 177
19.2 通过声明函数避免飞去来器效应 ····· 179
19.3 使用 ASYNC 流程控制库 ····· 183
19.3.1 串行执行 ····· 184
19.3.2 并行执行 ····· 185
19.3.3 连续传递 ····· 186
19.3.4 排队 ····· 187
19.3.5 迭代 ····· 189
19.3.6 映射 ····· 190
19.3.7 规约 ····· 191
19.3.8 过滤 ····· 192
19.3.9 检测 ····· 193
19.4 本章小结 ····· 194

第Ⅴ部分 构建 Web 应用程序

第 20 章 构建和使用 HTTP 中间件 ····· 197
20.1 理解 Connect HTTP 中间件框架 ····· 198
20.2 构建自定义 HTTP 中间件 ····· 198
20.2.1 创建异步中间件 ····· 199
20.2.2 在中间件内部注册回调函数 ····· 201
20.2.3 在中间件内处理错误 ····· 202
20.3 使用捆绑在 Connect 中的 HTTP 中间件 ····· 206

20.3.1 记录请求 …… 206
20.3.2 处理错误 …… 208
20.3.3 提交静态文件 …… 209
20.3.4 解析查询字符串 …… 210
20.3.5 解析请求主体 …… 211
20.3.6 解析 Cookies …… 212
20.3.7 使用会话 …… 213
20.3.8 其他可用的中间件 …… 216
20.4 本章小结 …… 216

第 21 章 用 Express.js 创建 Web 应用程序 …… 217
21.1 初始化 Express.js 应用程序 …… 218
21.2 在应用程序中设置中间件 …… 220
21.3 路由请求 …… 222
21.3.1 处理路由 …… 222
21.3.2 使用会话 …… 229
21.3.3 使用路由中间件 …… 234
21.4 本章小结 …… 238

第 22 章 使用 Socket.IO 创建通用的实时 Web 应用程序 …… 241
22.1 理解 WebSockets 如何工作 …… 242
22.2 使用 Socket.IO 创建 WebSocket 应用程序 …… 243
22.2.1 在服务器上安装和运行 Socket.IO …… 243
22.2.2 使用 Socket.IO 创建实时网络聊天应用程序 …… 245
22.2.3 扩展聊天应用程序 …… 250
22.2.4 检测连接断开 …… 253
22.2.5 将用户分隔到聊天室中 …… 255
22.2.6 使用名称空间 …… 259
22.2.7 使用 Redis 分布运行服务器端应用程序 …… 260
22.3 本章小结 …… 263

第Ⅵ部分 连接数据库

第 23 章 使用 node-mysql 连接 MySQL 数据库 …… 267
23.1 应用库与 MySQL 数据库进行连接和通信 …… 267
23.2 向数据库添加数据时请记住安全性 …… 269
23.3 高效读取数据 …… 272
23.4 本章小结 …… 276

第 24 章 使用 Nano 连接 CouchDB 数据库 …… 277
24.1 安装 Nano …… 278
24.2 连接和创建数据库 …… 281
24.3 存储文档 …… 285
24.4 创建和使用 CouchDB 视图 …… 286
24.5 将文件附加到 CouchDB 文档上 …… 299
24.6 本章小结 …… 312

第 25 章 使用 Mongoose 连接 MongoDB 数据库 …… 313
25.1 安装 Mongoose …… 315
25.2 理解 Mongoose 如何使用模型封装对数据库的访问 …… 315
25.3 连接 MongoDB 数据库 …… 316
25.4 定义模式 …… 316
25.5 定义模型 …… 316
25.5.1 使用验证器 …… 326
25.5.2 使用修改器 …… 332
25.5.3 使用取值器 …… 333
25.5.4 使用虚拟属性 …… 334
25.5.5 使用默认值 …… 340
25.5.6 定义索引 …… 341
25.5.7 使用 DB Refs 引用其他文档 …… 343
25.5.8 定义实例方法 …… 349
25.5.9 定义静态方法 …… 350
25.6 本章小结 …… 351

第 I 部分

概述和安装

- 第 1 章：安装 Node
- 第 2 章：Node 简介

第 1 章

安 装 Node

本章提要：

- 安装并运行 Node
- 安装 Node 包管理器(Node Package Manager，NPM)
- 应用 NPM 安装、卸载和更新包

在 2009 年的欧洲 JavaScript 大会上，年轻程序员 Ryan Dahl 展示了他正在从事的一个项目，该项目是一个集成了 Google V8 JavaScript 引擎、事件循环和底层 I/O 应用编程接口(Application Programming Interface, API)的平台。与其他服务器端的 JavaScript 平台不同，Dahl 的平台中所有 I/O 原语[1]都是事件驱动的，除此以外别无他途。借助 JavaScript 的影响力和易用性，Dahl 的项目使得编写基于事件驱动的服务器端应用程序的任务由难变易，因此，该项目受到了热烈欢迎，并且它的发展、普及和被接受程度都是前所未有的。

这个项目被命名为 Node.js，但时至今日它的简称 Node 更为开发者们所熟知。Node 为构建高度并发的软件提供了一个纯事件驱动的、非阻塞的基础架构。

注意：Node 让你能够非常容易地构建快速、可伸缩的网络服务。

Node 出现伊始，就引起了一些业界巨头的注意，它们已经应用 Node 进行快速、可伸缩网络服务的部署。Node 如此吸引人，是因为以下几点原因：

原因之一是 Node 使用的是 JavaScript 语言，JavaScript 语言是世界上使用最广泛的编程语言，大多数网络程序员都曾经在浏览器端编写过 JavaScript 程序，而在服务器端编程

1 译者注：“原语”一词源自于操作系统领域，是指由若干条指令组成并完成一定功能的一个过程，也可引申到程序设计领域。

可以看作是对浏览器端编程的自然扩展。

另一个原因是 Node 的简单性。Node 的核心函数集很小，并且现有的 API 都相当简洁，从而最大程度降低了程序员的开发复杂度。当希望构建一些更加复杂的应用程序时，只需要毫不费力地选择、安装和应用一些可获取的第三方模块即可。

Node 吸引人的另一个原因是它非常容易上手，可以在几分钟内轻松地下载、安装并且运行。

可以遵照 http://nodejs.org 网站上给出的步骤将 Node 安装到用于开发的计算机上，Node 在 Windows、Linux、Macintosh 和 Solaris 上都可以即装即用。

1.1　在 Windows 上安装 Node

Node 从 0.6.0 版本开始支持 Windows 操作系统，要在 Windows 上安装 Node，首先需要将浏览器转向网址 http://nodejs.org/#download，单击 node-v*.msi 下载链接下载 Windows 安装包，此时会出现一个提示性的安全对话框，如图 1-1 所示。

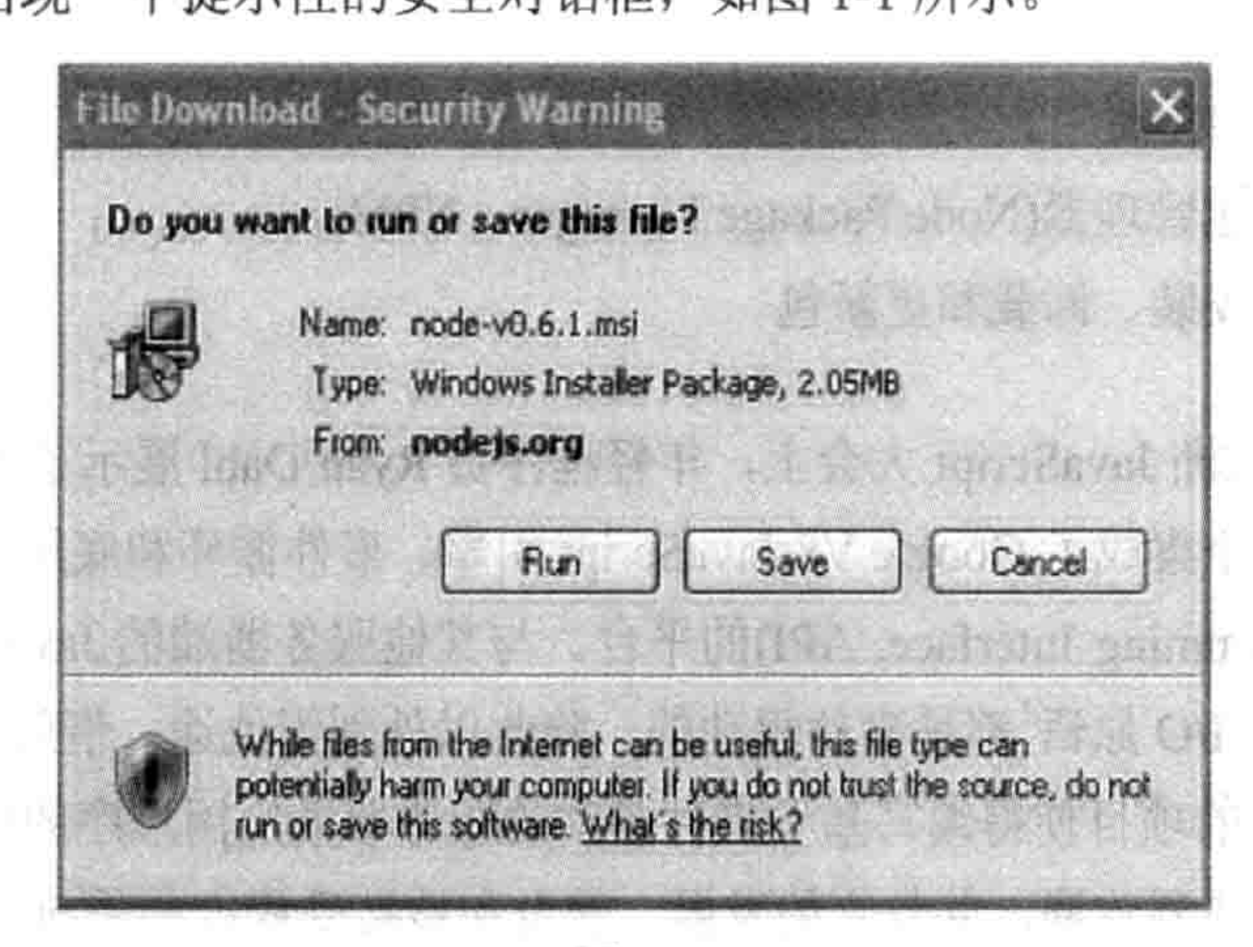

图 1-1

单击 Run 按钮，此时会出现另一个安全对话框提示你进行确认，如图 1-2 所示。确认之后，就会启动 Node 安装向导。

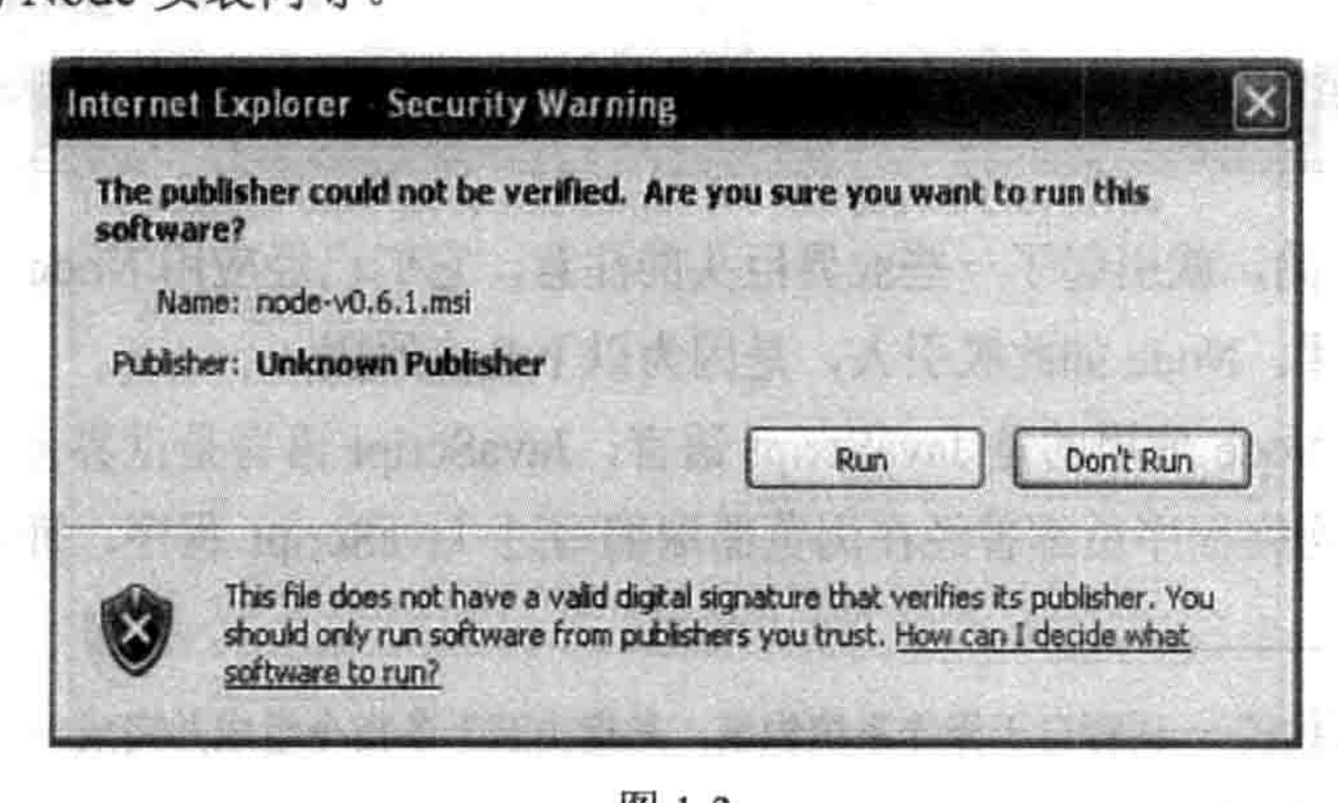

图 1-2

单击 Run 按钮，安装向导开始运行，如图 1-3 所示。

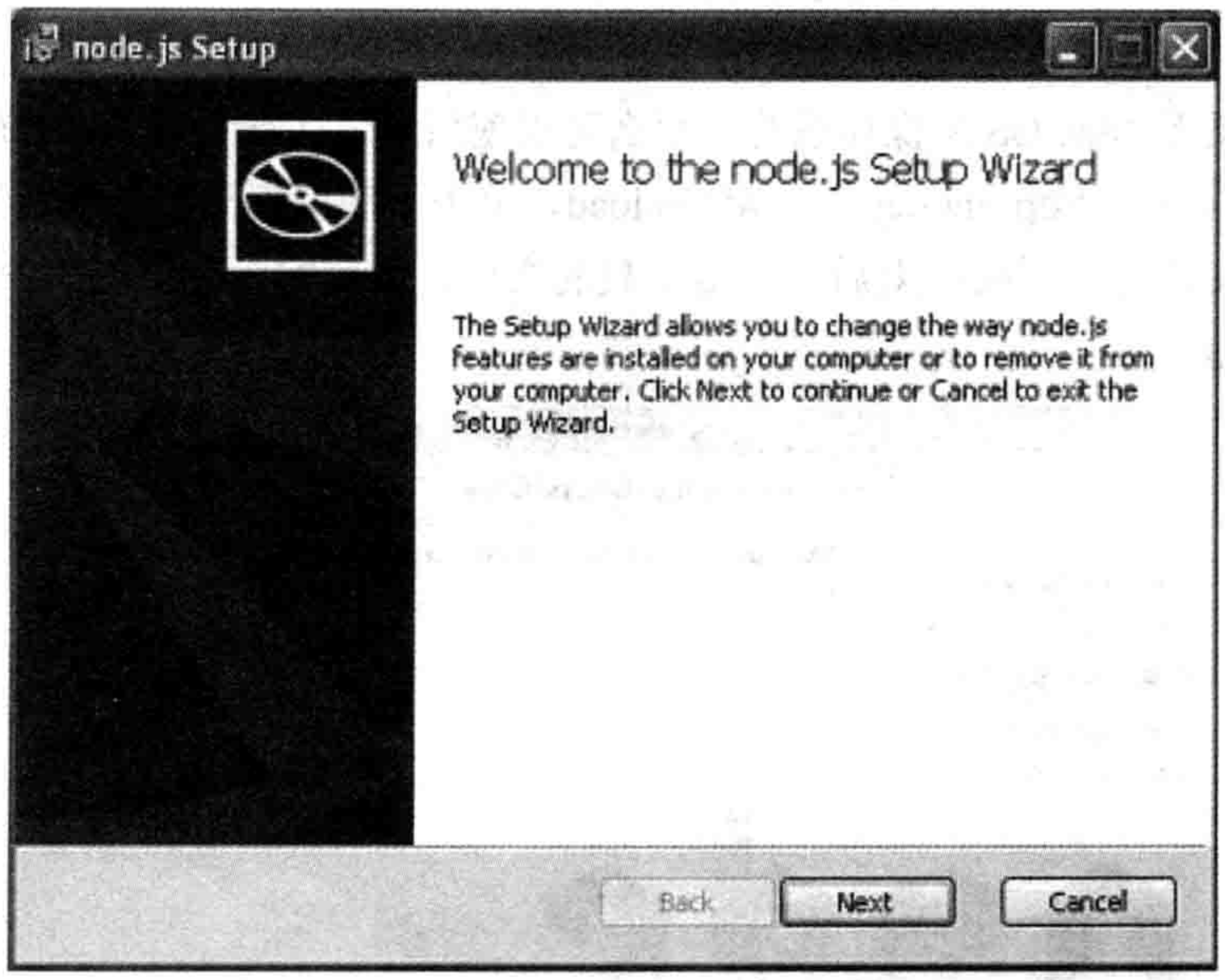

图 1-3

单击 Next 按钮，Node 开始安装，几分钟之后就会看见 Node 已经安装完毕的信息，如图 1-4 所示。

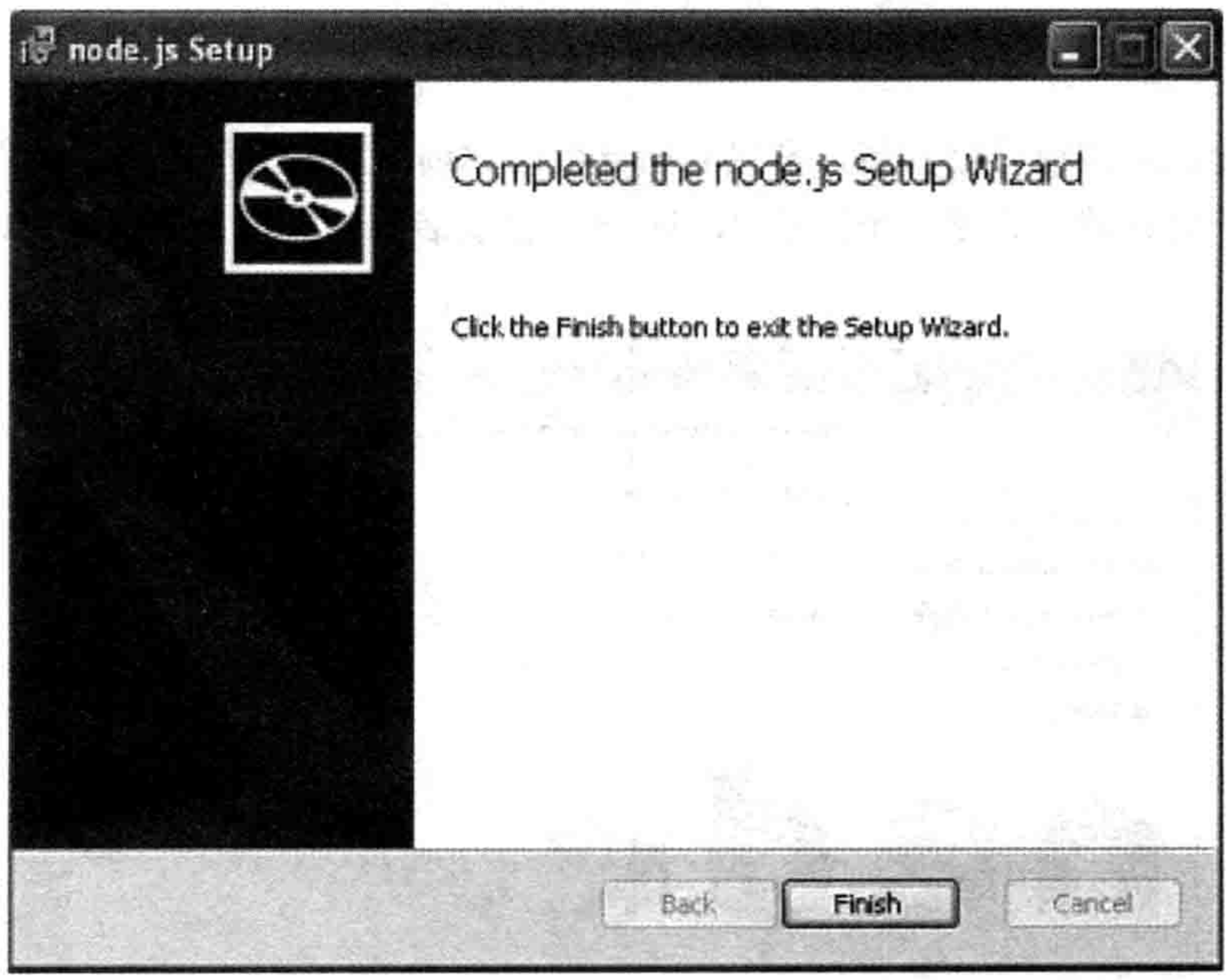

图 1-4

1.2　在 MAC OS X 上安装 Node

假如使用的是 Mac OS X 操作系统，那么可以使用安装向导进行 Node 的安装。首先，在浏览器中转向网址 http://nodejs.org/#download，单击下载链接开始下载名为 node-v*.pkg 的 Macintosh 安装包，下载结束后，单击下载的文件，运行它，会出现如图 1-5 所示的第一个向导对话框。

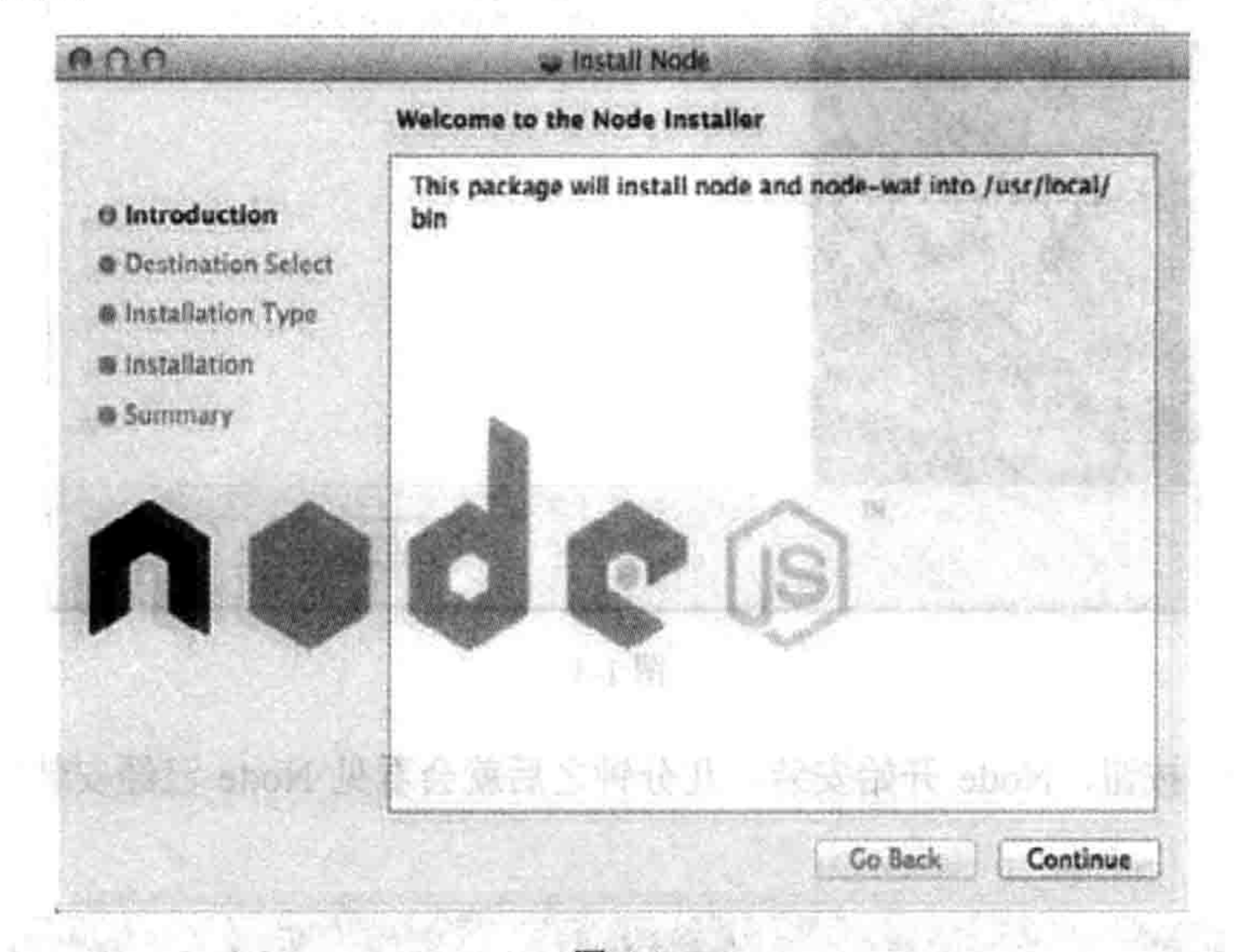

图 1-5

单击 continue 按钮开始安装，紧接着向导会向你询问系统用户密码，输入密码后就开始安装。不一会儿就会看见一个提示 Node 已经被安装到计算机上的确认窗口，如图 1-6 所示。

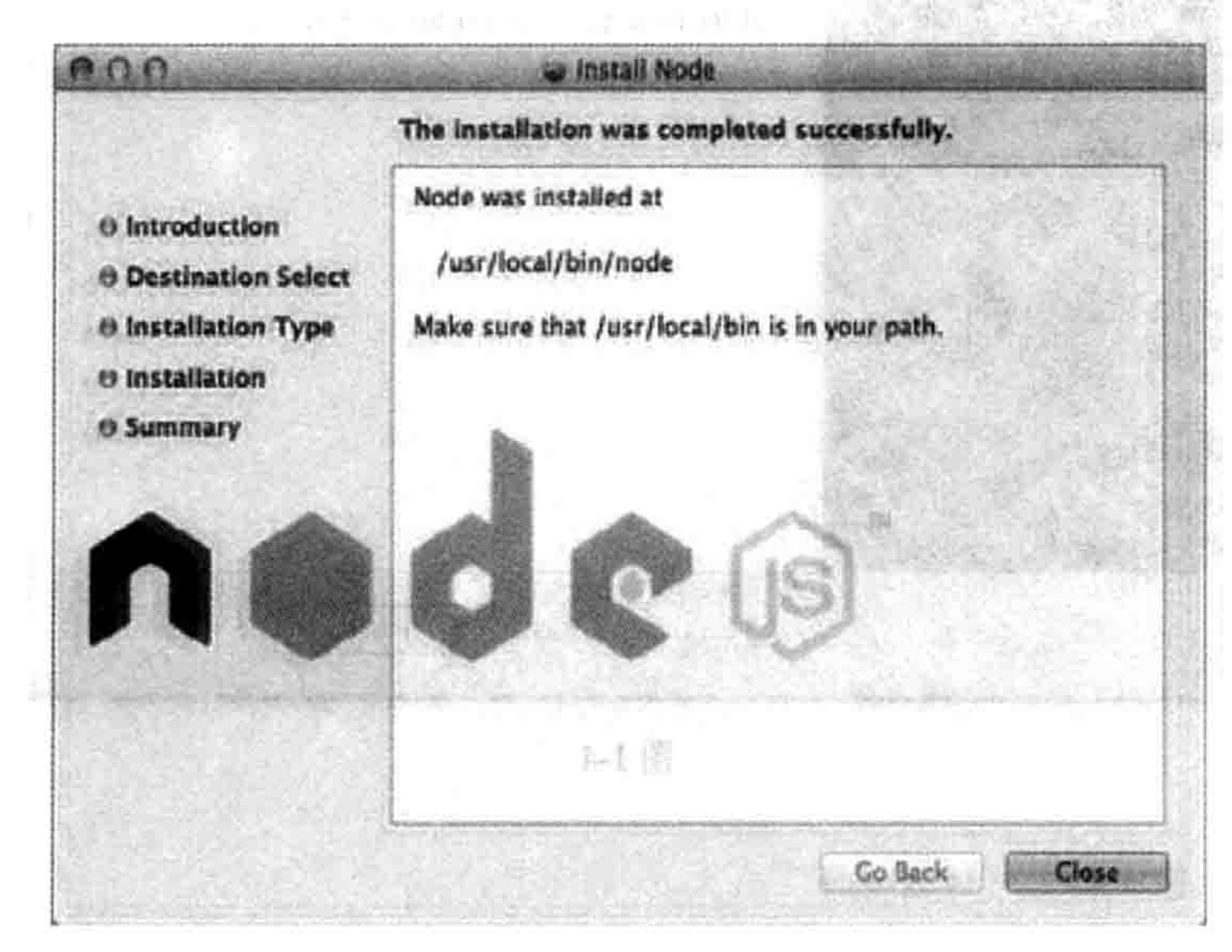

图 1-6

1.3　使用源代码安装 Node

假如使用的是 UNIX 系统，则可以通过编译源代码进行 Node 的安装。首先选择要安装 Node 的版本，之后下载 Node 的源代码并进行编译、安装和运行。

注意：Node 依赖于一些第三方代码库，但幸运的是大多数第三方库都已随 Node 一起发行。如果想从源代码开始进行编译，仅仅需要以下两项工具：

- python(2.4 及以上版本)——运行在 python 上的编译工具，随 Node 一起发行。
- libssl-dev——若是计划在网络程序中应用 SSL/TLS 加密，则必须安装它。libssl 是 openssl 工具中用到的库，在 Linux 和 UNIX 系统中，通常可以用你喜欢的包管理器安装 libssl，而在 Max OS X 系统中已经预安装了 libssl 库。

1.3.1　选择 Node 的版本

在 nodejs.org 网站上通常有两个不同版本的 Node 可供下载：稳定版和非稳定版。Node 使用副版本号表示该版本的稳定性，稳定版使用偶数(0.2、0.4、0.6)，非稳定版使用奇数(0.1、0.3、0.5、0.7)。

非稳定版不仅功能不稳定，并且 API 也有可能在后续版本中发生变化，而已在稳定版中公布的 API 是不会变化的，稳定版的新补丁仅仅包含对 bug 的修正，而非稳定版的补丁中往往还包含对 API 的修改。

除非你是想测试一下只包含在新发行的非稳定版中的新特性，否则都应该选择稳定版。非稳定版对于 Node 核心团队来说就像用来测试新研发成果的战场。

越来越多的项目和公司在他们的产品中成功地应用了 Node(在 nodejs.org 的主页上有相关介绍)，但是你也许得花些精力跟上每当新的稳定版 Node 发行时所导致的 API 更改，这也是使用一项新技术所要付出的代价。

1.3.2　下载 Node 源代码

选择好版本后，就可以到 nodejs.org 网站上复制对应的 tar 压缩包的下载链接进行下载。如果使用的是 UNIX 系统，那么也许已经安装过了 *wget*，这意味着可以在命令行模式下输入以下命令进行下载：

```
$ wget http://nodejs.org/dist/v0.6.1/node-v0.6.12.tar.gz
```

如果没有安装 *wget*，也可以用 *curl* 实用工具下载：

```
$ curl -O http://nodejs.org/dist/v0.6.1/node-v0.6.12.tar.gz
```

假如以上两种工具软件都没有安装，也可以通过别的方法将 tarball 文件下载到本地目录中——比如使用浏览器或者通过局域网传输。

在本书写作时，Node 的最新稳定版本是 0.6.12，书中的所有例子都是应用该版本编写的。

1.3.3　编译 Node

现在 Node 的源代码已经在你的计算机中了，这就可以将其编译生成 Node 可执行文件，首先需要输入以下命令对前面下载的 tar 压缩包进行解压缩：

```
$ tar xfz node-v0.6.12.tar.gz
```

然后进入源代码目录：

```
$ cd node-v0.6.12
```

对其进行配置：

```
$ ./configure
```

如果配置成功，可以看到以下输出信息：

```
'configure' finished successfully (9.278s)
```

现在就可以开始编译了：

```
$ make
```

如果编译成功，就会看到以下输出信息：

```
'build' finished successfully (0.734s)
```

1.3.4　安装 Node

编译生成 Node 可执行文件后，就可以执行以下指令开始安装：

```
$ make install
```

以上指令会将 Node 可执行文件复制到/usr/local/bin/node 目录下。

在执行以上指令时如果遇到权限问题，可以以 root 用户权限执行，或者在指令前加上 sudo：

```
$ sudo make install
```

1.3.5　运行 Node

现在就可以开始运行 Node 了。首先可以简单地体验一下 Node 命令行交互界面(Command-Line-Interface, CLI)，只需要执行以下无参数指令调用 Node 可执行文件即可：

```
$ node
```

上述指令会启动 Node 命令行交互界面，并等待输入，为了检验安装是否成功并查看 Node 实际完成的一些工作，可以输入以下指令试试看：

```
> console.log('Hello World!');
Hello World!
> undefined
```

还可以通过文件执行 JavaScript 脚本。例如，如果创建了一个包含以下内容的文件：

```
> console.log('Hello World!');
```

将文件命名为 hello_world.js，然后在命令行模式下以该文件名作为第一个参数调用 Node 可执行文件：

```
$ node hello_world.js
Hello World!
```

可以通过输入 Ctrl+D 或者 Ctrl+C[2]退出命令行交互界面。

1.4　安装和应用 Node 包管理器

到目前为止，我们还只能使用 Node 的语言特性及核心函数，这也是为什么大多数编程平台都具有一个系统用来下载、安装和管理第三方模块的原因。在 Node 中这个系统被称为 Node 包管理器(Node Package Manager, NPM)。

NPM 是三位一体的系统——第三方包库、管理计算机中安装的包的机制以及定义包依赖关系的标准。NPM 提供了一种公共注册服务，它包含了程序员在 NPM 中发布的所有包，NPM 还提供了一个命令行工具用来下载、安装和管理这些包。你还可以使用标准化的包描述符来指定自己的模块或者应用程序需要依赖的第三方包。

你并不需要了解 NPM 也一样可以使用 Node，但是一旦涉及使用第三方包，那么了解一下 NPM 就十分必要。因为 Node 仅仅提供了底层 API，所以如果想不必事事躬亲就能实现一个复杂的应用程序，那么在应用程序中包含第三方包几乎不可或缺。正如将看到的那样，NPM 允许你不必无一遗漏地安装所有包就可以下载和应用包，这个特性使我们可以尽情地来做些试验。

在早期，NPM 和 Node 是要分别独立安装的，但是从 0.6.0 版开始，NPM 就已经包含在 Node 的安装包中。

应用 NPM 安装、更新和卸载包

NPM 是一个强大的管理工具，可以用很多方法来使用它。NPM 集中维护了一个公共包库，这个库可以通过网址 http://search.npmjs.org 来查看。大多数 Node 开源包的作者都会选择在 NPM 中发布他们的作品，而包的用户则可以按照安装提示根据 NPM 中包的名称进

2　译者注：在 Windows 下需要输入两次 Ctrl+C。

行下载和安装。

本节将要讲述 NPM 的大多数典型用法，即安装和移除包，这些知识足够你管理依赖于 NPM 中发布的第三方包的应用程序。当然，首先我们需要了解全局模式和本地模式的区别，以及这些区别如何影响模块查找。

使用全局模式和本地模式

NPM 有两种主要的工作模式：全局模式和本地模式，这两种模式会引起包的存储目标目录发生变化，这种变化对 Node 载入模块的方式有着深刻的影响。

本地模式是 NPM 中默认的工作模式，该模式下 NPM 只工作在本地文件夹层次上，不会引起系统范围内的变化。如果想在安装包时不会对那些已经在本地安装的应用程序造成影响，那么本地模式十分合适。

全局模式适合安装那些全局可用的包，比如提供命令行实用工具的包或者不会被应用程序直接调用的包。

假如在两种模式之间犹豫不决，那么就应该使用本地模式，如果模块的作者要求某个特定的模块需要以全局模式安装，他会在安装说明中指出。

全局模式

如果在安装 Node 时使用的是默认目录，那么在全局模式下 NPM 会将包安装到/usr/local/lib/node_modules 目录下。如果在命令行模式下输入以下指令，NPM 就会在查找、下载最新版本的 sax 包，并将其安装到/usr/local/lib/node_modules/sax 目录下：

```
$ npm install -g sax
```

注意：如果的当前 shell 用户没有足够权限，那么你必须以 root 用户登录或者在上述指令前加上 *sudo*：

```
$ sudo npm install -g sax
```

如果需要在 Node 脚本中应用 sax 模块，则应通过以下语句加载：

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

Node 将会在前述目录加载该模块(除非该模块是以本地模式安装的，在本地模式中，Node 将会优先加载该模块的本地版本)。

由于本地模式是默认模式，因此必须在 NPM 指令中显式地指定-g 标志位才能启用全局模式。

本地模式

本地模式是 NPM 的默认工作模式，也是推荐的模块依赖关系解决方案，在该模式下，

NPM 将所有模块都安装到当前目录中——一般而言就是应用程序的根目录，这样就不会影响任何全局设置。可以选择这种机制逐个设置应用程序依赖的包以及它们的版本而不会污染全局模块空间。这意味着可以让两个应用程序分别依赖同一个包的不同版本而不致产生冲突。

在本地模式下，NPM 在当前工作目录下的 node_modules 子目录中存储包。假如当前工作目录是/home/user/apps/my_app，那么 NPM 就会把包块存储到/home/user/apps/my_app/node_modules 目录中。

如果运行安装在/home/user/apps/my_app 目录中的应用程序，Node 会首先在/home/user/apps/my_app/node_modules 目录中查找模块，如果没找到就查找其上一级目录，最后在全局模式下的包安装目录中寻找。也就是说 Node 在解析包依赖关系时，以本地模式安装的包优先级高于以全局模式安装的包。

安装包

输入以下命令就可以下载和安装最新版本的包：

```
$ npm install <package name>
```

例如，要下载和安装最新版本的 sax 包，首先要将当前文件工作目录设置为应用程序的根目录，然后输入以下指令：

```
$ npm install sax
```

如果 node_modules 文件夹不存在，那么以上命令将会创建一个，并将 sax 包安装到该文件夹下。

也可以输入以下命令，来选择要安装的包的版本：

```
$ npm install <package name>@<version spec>
```

实际操作中使用特定的版本号替换上述命令中的<version spec>占位符即可。比如，可以输入以下命令安装 0.2.5 版的 sax 包：

```
$ npm install sax@0.2.5
```

此外还可以用一个版本号范围替换<version spec>占位符。比如，输入以下命令可以安装版本号前缀为 0.2 的最新版 sax 包：

```
$ npm install sax@0.2.x
```

还可以通过输入以下命令下载版本号小于 0.3 的最新版的 sax 模块：

```
$ npm install sax@"<0.3"
```

还可以输入类似如下所示的命令，以更为复杂的组合形式指定包的版本号：

```
$ npm install sax@">=0.1.0 <0.3.1"
```

卸载包

使用以下命令，NPM 就会通过指定的名称查找和卸载包：

```
$ npm uninstall <package name>
```

如果要卸载以全局模式安装的包，则应输入以下命令：

```
$ npm uninstall -g <package name>
```

更新包

我们还可以通过以下指令更新已经安装的包：

```
$ npm update <package name>
```

该命令会获取最新版本的包，并更新对应的本地版本，如果要更新的包不存在的话，则会安装这个包。

同样，我们也可以加上全局切换标志-g 更新以全局模式安装的包：

```
$ npm update -g <package name>
```

应用可执行文件

一个包可能包含一个或者多个可执行文件，假如使用默认的安装目录设置，并以全局模式安装，那么 NPM 将会把可执行文件安装到/usr/local/bin 目录中，该路径一般已被包含在默认的可执行文件的 PATH 环境变量中。

如果是以本地模式安装该包，那么 NPM 将会把可执行文件安装到./node_modules/.bin 目录中。

解析依赖关系

NPM 不仅会安装你指定的包，还会安装这些指定包所依赖的其他包。举个例子来说，假设你指定需要安装 A 包，而 A 包又依赖于 B 包和 C 包，那么 Node 将会获取 B 包和 C 包并将它们安装到./node_modules/A/node_modules 目录中。

例如，如果应用以下命令以本地模式安装 nano 包：

```
$ npm install nano
```

NPM 会输出以下信息：

```
nano@0.9.3 ./node_modules/nano
├── underscore@1.1.7
└── request@2.1.1
```

以上信息说明 nano 包依赖 undercore 包和 request 包，并指出了这两个包安装的版本。如果现在用以下命令查看一下./node_modules/nano/node_modules 文件夹，就会发现 undercore

包和 request 包都已被安装在这个文件夹下：

```
$ ls ./node_modules/nano/node_modules
request undercore
```

应用 package.json 定义依赖关系

在编写 Node 应用程序时，可以在根目录中包含一个 package.json 文件，可以在该文件中定义应用程序的一些元数据，例如名称、作者、代码库和协议等，应用程序依赖的外部包也在该文件中定义。

你不会应用 package.json 将应用程序发布到 NPM，因为也许你希望应用程序是私有的，但是你依然可以用它指定应用程序依赖的包。

package.json 是 JSON 格式的文件，包含一系列属性，但是如果只是为了声明依赖关系，那么仅仅需要一个属性：dependencies。如果一个应用程序依赖 sax、nano 和 request 包，那么它对应的 package.json 文件如下所示：

```
{
  "name" : "MyApp",
  "version" : "1.0.0",
  "dependencies" : {
   "sax" : "0.3.x"
   "nano" : "*"
   "request" : ">0.2.0"
  }
}
```

在以上文件中指定了应用程序或者包依赖于 0.3 版本的 sax 包、任意版本的 nano 包以及 0.2 版本以上的 request 包。

注意：如果遗漏了 name 和 version 域，会导致 NPM 无法工作，因为 NPM 最初用来描述公共模块，而不是私有应用程序。

然后在应用程序根目录下输入如下指令：

```
$ npm install
```

命令执行后，NPM 将会分析依赖关系和 node_modules 文件夹，并自动下载和安装缺失的模块。

也可以根据自己指定的依赖关系，使用以下命令将以本地模式安装的模块更新至最新版本：

```
$ npm update
```

实际上可以仅使用最后一条命令即可，因为它总是获取任何缺失的模块。

1.5　本章小结

本章你学习了如何安装 Node 和 Node 包管理器，并且现在可以应用 NPM 安装、卸载和移除第三方包，而且你还学会了使用 package.json 文件配合 NPM 管理所要依赖的第三方包。

既然已经安装了 Node 和 NPM，后面就可以使用它们了，但事先需要了解一些有关 Node 以及事件驱动编程风格的背景知识。

第 2 章

Node 简 介

本章提要：

- 什么是事件驱动编程风格？使用这种风格有什么优点？
- Node.js 和 JavaScript 如何简化了事件驱动编程？

在传统编程中，I/O 操作和本地函数调用的处理方式相同：处理过程需要一直等待直到某个操作结束才能继续下去。这种进行 I/O 操作的阻塞式编程模型继承自早期的分时系统，在这类系统中，每一个进程都对应着一个用户，这样做的目的是使用户之间相互隔离。并且在这类系统中，用户在决定下一步操作之前，通常都必须先结束一个操作。但是随着计算机网络和 Internet 的广泛应用，这种“单用户、单进程”的模型不具备较好的伸缩性，管理众多进程在内存和上下文交换代价方面给操作系统带来了沉重的负担，当任务的数量达到一定值后，其性能就开始急剧下降。

除了上述编程模型之外，多线程编程模型是另一个选择。线程可以看做是一种轻量级进程，它与同一进程内的其他线程共享内存。线程作为早期模型的扩展而被创建，以适应若干并发线程的执行。当一个线程等待 I/O 操作时，另一个线程就可以接管 CPU，等待线程将在 I/O 操作结束时被唤醒，这意味着线程在其执行期间可以被中断，并且之后又能够恢复执行，并且有些系统允许线程在不同的 CPU 核上并行运行。

这意味着程序员并不知道在给定时刻究竟是哪些线程在执行，所以他们必须仔细处理对共享内存状态的并发访问，使用诸如线程锁和信号量这样的同步原语协调对某些数据结构的访问，强制它们对线程执行调度的各种途径进行预测以避免问题的发生。假如应用程序非常依赖于线程间的共享状态，那么这种编程方式容易导致随机发生的危险错误，并且这些错误很难查找。

如果不让操作系统进行线程调度，那么程序员的另一个选择是协同多线程，在这种模

式下，程序员负责显式地让 CPU 为其他线程的执行分配时间，因为是人为负责线程调度，所以可以放宽同步需求，但是基于和普通多线程同样的原因，这种方法也很复杂并且容易出错。

2.1　事件驱动编程风格介绍

事件驱动编程是指程序的执行流程取决于事件的编程风格，事件由事件处理程序或者事件回调函数进行处理。当某些重要的事件发生时——例如数据库查询结果可用或者用户单击了某个按钮，就会调用事件回调函数。

以下面的代码为例，看看在典型的阻塞式 I/O 编程中，如何实现对数据库的查询：

```
result = query('SELECT * FROM posts WHERE id = 1');
do_something_with(result);
```

上述查询过程需要当前线程或者进程一直等待，直到数据库层完成对查询的处理为止。

而在事件驱动系统中，上述查询则以如下方式进行：

```
query_finished = function(result) {
  do_something_with(result);
}
query('SELECT * FROM posts WHERE id = 1', query_finished);
```

此处首先定义了在查询完成之后将会发生的处理过程，并且将这个过程存储在一个名为 query_finished 的函数当中，然后将函数名作为查询的一个参数。这样，当查询完成之后将会调用 query_finished 函数，而不仅仅只是返回查询结果。

这种以定义当感兴趣事件发生时由系统调用的函数来取代应用返回值的编程风格被称为事件驱动编程或者异步编程，它是 Node 的显著特征之一。这种编程风格意味着当前进程在处理 I/O 操作时不会发生阻塞，因此，多个 I/O 操作可以并行进行，当每个操作结束时，将会分别调用其对应的回调函数。

事件驱动编程风格和事件循环相伴相生，事件循环是一个处于不间断循环中的结构，该结构主要具有两项功能——事件检测和事件触发处理，在每一轮循环中，它都必须检测发生了什么事件。当事件发生时，事件循环还要决定调用哪个回调函数。

事件循环只是在一个进程中运行的单个线程，这意味着当事件发生时，可以不用中断就运行事件处理程序，这样做有以下两个特点：

- 在任一给定时刻，最多运行一个事件处理程序。
- 事件处理程序可以不间断地运行直到结束。

这使得程序员能放宽同步要求，并且不必担心执行并发线程会改变共享内存的状态。

众所周知的秘密

在相当一段时间内，系统编程领域已经知道事件驱动编程是创建处理众多并发连接的服务的最佳方法。众所周知，由于不用保存很多上下文，因此节省了大量内存；又因为也没有那么多上下文切换，又节省了大量执行时间。

这种理念已经渗透进其他平台和领域，例如 Ruby 的 *Event Machine*、Perl 的 *AnyEvent* 和 Python 的 *Twisted* 都是一些知名的事件循环实现，当然还包括其他一些语言。

使用上述框架之一实现应用程序时需要该框架指定的编程语言和库，例如使用 Event Machine 时，应该避免使用同步库，为了利用非阻塞模式的优势，你被限制使用专门为 Event Machine 构建的异步库。如果使用了阻塞库(例如 Ruby 的大多数标准库)，那么所搭建的服务器无法进行理想的伸缩，因为事件循环经常会被阻塞，从而妨碍处理 I/O 事件。

从诞生的第一天开始，Node 就是作为一个非阻塞 I/O 服务器平台设计的，所以基于它构建的所有应用程序都会是非阻塞的。因为 JavaScript 非常小，并且没有为 I/O 操作增加额外的东西(JavaScript 没有标准 I/O 库)，所以 Node 的构建基础非常纯净。

2.2 Node 和 JavaScript 如何简化异步应用程序的编写

Node 的作者 Ryan Dahl 是从构建一个 C 语言平台开始他的项目的，但是这样一来，维护函数调用之间的上下文就十分复杂，导致编写的代码晦涩难懂。然后他转向了 Lua，但是 Lua 已经有了一些阻塞式 I/O 库，将阻塞式和非阻塞式混合起来，会让开发者产生混淆，使他们难以构建伸缩性良好的应用程序，因此 Lua 也不是一个理想的选择。

Dahl 最后转向了 JavaScript，JavaScript 具有闭包和第一类函数，这使得它非常适合于事件驱动编程，JavaScript 的影响力是 Node 如此流行的主要原因之一。

2.2.1 什么是闭包

闭包就是函数，但是它可以继承并访问它自身被声明的那个作用域里的变量。当你将一个回调函数作为参数传递给另外一个进行 I/O 操作的函数时，回调函数稍后会被调用，神奇的是，在被调用时，回调函数会记住它自身声明时所在的上下文，并且可以访问该上下文及其父上下文里的所有变量，这个强大的特性是 Node 成功的核心。

下面的例子展示了在 Web 浏览器中进行 JavaScript 编程时，闭包是如何工作的。例如，如果想监听按钮的单击事件，可以按如下方式：

```
var clickCount = 0;
document.getElementById('myButton').onclick = function() {
  clickCount += 1;
  alert("clicked " + clickCount + " times.");
};
```

或者应用 jQuery：

```
var clickCount = 0;
$('button#mybutton').click(function() {
  clickedCount ++;
  alert('Clicked ' + clickCount + ' times.');
});
```

在 JavaScript 中，函数是第一类对象，这意味着可以将函数作为参数传递给其他函数。在上述两个例子中，你把一个函数赋值给另一个函数，或者将函数作为参数传递给另一个函数以备稍后执行。单击处理程序——也就是你的回调函数——可以访问函数声明作用下的每一个变量，在本例中也就是说回调函数可以访问父闭包中声明的 clickCount 变量。

变量 clickCount 存储了用户单击按钮的次数，该变量在全局域中存储(你在 JavaScript 中可到达的最外层作用域)。在全局域存储变量是一种不好的做法，因为很容易和其他代码发生冲突，你应该将变量放在需要调用函数的局部作用域中，大多数时候将代码包装在函数中以额外创建闭包就可以避免污染全局作用域，如下例所示：

```
(function() {
  var clickCount = 0;
  $('button#mybutton').click(function() {
    clickCount ++;
    alert('Clicked ' + clickCount + ' times.');
  });
}());
```

注意：第 7 行定义了一个函数之后立即对其进行了调用，这是一种常见的 JavaScript 模式：通过创建函数来创建新的作用域。

2.2.2 闭包如何辅助异步编程

在进行事件驱动编程时，从定义事件发生时执行的代码开始，然后将代码放入一个函数当中，最后将该函数作为一个参数进行传递以备稍后调用。

在 JavaScript 中，函数操作并不是孤立地工作而是会记住它被声明时所在的上下文，这能让函数操作其声明时所在的上下文以及父上下文中的所有变量。

当你将回调函数作为参数进行传递时，它会在稍后某个时刻被调用，但即使声明回调函数的那个作用域已经返回，回调函数依然可以操作该作用域或者父作用域中的任意变量。在上述最后一个例子中，虽然回调函数是在 jQuery 的 click()函数内部被调用的，但是它依然可以访问或操作 clickCount 变量。

这显示通过应用闭包模式，可以从两个方面获得极大的好处：可以通过将状态变量传递给函数而不必维护它就能进行事件驱动编程，JavaScript 的闭包会为你维护状态变量。

2.3　本章小结

事件驱动编程是一种程序流程取决于事件发生的编程风格，程序员为他们感兴趣的事件注册回调函数作为事件的处理程序，在那些事件发生时系统就会调用这些处理程序。这种编程模型有很多传统阻塞编程模型所不具备的优势，在传统编程模型中要实现可伸缩性，你必须使用多进程或者多线程才行。

JavaScript 是一门强大的语言，因为它具有第一类函数和闭包，所以很适合事件驱动编程风格。

第 II 部分

Node核心API基础

- 第 3 章：加载模块
- 第 4 章：应用缓冲区处理、编码和解码二进制数据
- 第 5 章：使用事件发射器模式简化事件绑定
- 第 6 章：使用定时器制定函数执行计划

第 3 章

加载模块

本章提要：

- 加载模块
- 创建模块
- 使用 node_modules 文件夹

JavaScript 是世界上使用频率最高的编程语言之一，它是网络世界的通用语言，所有浏览器都使用它。JavaScript 的核心是在 Netscape 公司大行其道的岁月里创建的，而当时正处于浏览器战争的白热化时期，JavaScript 就是作为打击微软的利器而匆匆上马的。JavaScript 并没发展得十分成熟就被发布，这意味着它不可避免地带有一些糟糕的特性。

虽然开发的时间不长，但是 JavaScript 被赋予了一些强大的特性，不过，“脚本间共享全局名称空间”这一特性却不在其中。

一旦 JavaScript 代码被载入网页，就会被添加进全局名称空间，全局名称空间是被所有已载入的脚本所共享的通用地址空间，这样就会导致安全性问题、冲突以及一些难以跟踪和解决的一般性错误。

值得庆幸的是，Node 为服务器端的 JavaScript 制定了一些规范，并实现了 CommonJS 模块标准。在这个标准中，每个模块都拥有一个上下文，将该模块和其他模块隔离开来，这意味着模块不会污染全局作用域——因为根本就没有全局作用域——并且也不会对其他模块造成干扰。

本章你将学习不同类型的模块，以及如何将它们载入到你的应用程序中。

将代码拆分成一系列定义良好的模块可以帮助你有效地控制代码，你还将学习如何创建和使用自定义模块。

3.1 理解 Node 如何加载模块

在 Node 中既可以用文件路径也可以用名称来引用模块，除非是核心模块，否则用名称引用的模块最终都会被映射为一个文件路径。Node 核心模块将一些核心函数暴露给程序员，它们在 Node 进程启动时会被预先载入。

其他模块包括使用 NPM 安装的第三方模块，以及你或你的同事创建的本地模块。

不管什么类型的模块，在被导入当前脚本之后，程序员都可以使用其对外暴露的一组公共 API。不管使用什么模块，你都可以使用 require 函数，如下所示：

```
var module = require('module_name');
```

上面的代码会导入一个核心模块或者由 NPM 安装的模块，*require* 函数会返回一个对象，该对象表示模块对外暴露的 JavaScript API。根据模块的不同，该对象可能是任意的 JavaScript 值——可以是一个函数，也可以是一个具有若干属性的对象，属性可能是函数、数组或其他任何类型的 JavaScript 对象。

3.2 导出模块

在 Node 中，CommonJS 模块系统是文件之间共享对象或函数的唯一方式，对于足够复杂的应用程序，你应该将一些类、对象或者函数划分成定义良好的可重用模块，模块只对外暴露你指定的内容。

在 Node 中，文件和模块是一一对应的，这在下面的例子中将会看到。下面从创建文件 circle.js 开始，该文件只是导出了 Circle 类的构造函数：

```
function Circle(x, y, r) {

  function r_squared() {
    return Math.pow(r, 2);
  }

  function area() {
    return Math.PI * r_squared()
  }

  return {
    area: area
  };
}

module.exports = Circle;
```

上面这段代码的重点在最后一行，最后一行处定义了该模块导出的内容。module 是一

个变量，它表示当前模块自身。而 module.exports 表示模块向需要它的脚本所导出的对象，它可以是任意对象。在本例中只是导出了 Circle 类的构造函数，用户可以使用该函数来创建具备完整功能的 Circle 实例。

还可以导出一些更为复杂的对象，module.export 被初始化成一个空对象，可以为这个空对象附加上任意想导出的属性。比如，可以设计模块，让其导出一组函数：

```
function printA() {
  console.log('A');
}

function printB() {
  console.log('B');
}

function printC() {
  console.log('C');
}

module.exports.printA = printA;
module.exports.printB = printB;
module.exports.pi = Math.PI;
```

该模块导出了两个函数(printA 和 printB)以及一个数字(pi)，使用该模块的客户脚本如下所示：

```
var myModule2 = require('./myModule2');
myModule2.printA(); // -> A
myModule2.printB(); // -> B
console.log(myModule2.pi); // -> 3.141592653589793
```

3.3 加载模块

前面已经说过，可以使用 require 函数来加载模块，在代码中调用 require 函数不会改变全局名称空间的状态，因为在 Node 中根本就没有全局名称空间这个概念。如果模块存在，并且没有任何语法或初始化错误，那么调用 require()函数就会返回这个模块对象，然后就可以将这个对象赋值给任意一个局部变量。

有几种方法来引用模块，具体采用何种方法取决于模块的类型——究竟是核心模块、通过 NPM 安装的第三方模块还是本地模块，下面让我们一起了解一下这些不同的引用方法。

3.3.1 加载核心模块

Node 中有一些以二进制形式发布的模块，这些模块被称为核心模块。核心模块只能通过模块名引用，而不能通过文件路径引用，即使已经存在一个与其同名的第三方模块，也

会优先加载核心模块。

例如，如果想要加载和使用 http 核心模块，应该如下所示：

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

上述代码会返回 http 模块对象，它实现了由 Node API 文档描述的 HTTP API。

3.3.2 加载文件模块

此外，还可以通过提供绝对路径从文件系统中加载非核心模块，如下所示：

```
var myModule = require('/home/pedro/my_modules/my_module');
```

或者也可以提供基于当前文件的相对路径，如下所示：

```
var myModule = require('.. /my_modules/my_module');
var myModule2 = require('. /lib/my_module_2');
```

注意，在上面的代码中可以省略文件扩展名.js，如果没有找到这个文件，Node 会在文件名后加上.js 扩展名再次查找路径。所以，如果当前目录中存在 my_module.js 文件，那么下面两条语句是等效的：

```
var myModule = require('. /my_module');
var myModule = require('. / my_module.js');
```

3.3.3 加载文件夹模块

还可以使用文件夹路径来加载模块，如下所示：

```
var myModule = require('. / myModuleDir');
```

如此一来，Node 就会在指定的文件夹下查找模块。Node 会假定该文件夹是一个包，并试图查找包定义。包定义包含在名为 package.json 的文件中。

如果文件夹中不存在包定义文件 package.json，那么包的入口点会假定为默认值 index.js[1]，就本例而言，Node 将会在路径./myModuleDir/index.js 下查找文件。

反之，如果文件夹中存在 package.json 文件，那么 Node 就会尝试解析该文件并查找 main 属性，将 main 属性当作入口点的相对路径。例如，如果./myModuleDir/package.json 文件如下所示，那么 Node 就会根据路径./myModuleDir/lib/myModules.js 加载文件：

```
{
  "name" : "myModule",
  "main" : "./lib/myModule.js"
}
```

3.3.4 从 node_modules 文件夹加载

如果一个模块名既不是相对路径，也不是核心模块，那么 Node 就会尝试在当前目录

1 译者注：除了 index.js 还会查找 index.node，.Node 文件是 Node 中的二进制扩展包，具体见官方文档。

下的 node_modules 文件夹中查找该模块。

例如，通过以下代码，Node 就会尝试查找文件./node_modules/myModules.js：

```
var myModule = require('myModule.js');
```

如果没有找到该文件，Node 就会继续在父文件夹../node_modules/myModule.js 中查找；如果还是没有找到，Node 会继续查找上一级的父文件夹，这个过程一直持续到到达根目录或者找到所需的模块为止。

可以利用这个特性管理 node_modules 目录中的内容，不过最好还是让 NPM 来管理模块(见第 1 章)。本地目录 node_modules 是 NPM 安装模块的默认位置，这项功能将 Node 和 NPM 关联到一起。作为一名程序员，你一般不必太关注这个特性，只是简单地使用 NPM 安装、更新和删除包，NPM 则会替你管理 node_modules 目录。

3.3.5 缓存模块

模块在首次加载时会被缓存起来，这意味着如果模块名能被解析为相同的文件名，那么每次调用 require('myModule')都会确切地返回同一模块。

例如，如果文件 my_modules.js 中包含如下所示的模块：

```
console.log('module my_module initializing...');
module.exports = function() {
  console.log('Hi!');
};
console.log('my_module initialized.');
```

然后使用如下所示的脚本加载一次上述模块：

```
var myModuleInstance1 = require('./my_module');
```

则会输出以下信息：

```
module my_module initializing…
module my_module initialized.
```

如果两次加载该模块，如下所示：

```
var myModuleInstance1 = require('./my_module');
var myModuleInstance2 = require('./my_module');
```

那么请注意，输出信息并没有发生变化：

```
module my_module initializing…
module my_module initialized.
```

这意味着模块的初始化过程只执行了一次，当构建自定义模块时，如果模块在初始化时可能会产生副作用，那么理解这一点十分重要。

3.4　本章小结

Node 取消了 JavaScript 默认的全局名称空间，而用 CommonJS 模块系统取而代之，这样可以让你更好地组织代码，也因此避免了一些安全性问题和错误。可以使用 require() 函数从文件或者文件夹加载核心模块、第三方模块或者自定义模块。

可以使用相对或者绝对文件路径加载非核心模块。如果将模块放入 node_modules 文件夹或者使用 NPM 安装模块，也可以通过模块名加载。

还可以编写 JavaScript 文件导出表示模块 API 的对象，以此创建自定义模块。

第 4 章

应用缓冲区处理、编码和解码二进制数据

本章提要：

- 理解 Node 中为何需要缓冲区
- 用字符串创建缓冲区
- 将缓冲区转换成字符串
- 在缓冲区中处理数据
- 切分和复制缓冲区

JavaScript 善于处理字符串，但由于它最初是被设计用来处理 HTML 文档的，因此它并不善于处理二进制数据。JavaScript 中没有字节类型，也没有结构化类型，甚至没有字节数组类型，只有数值类型和字符串类型。

由于 Node 是基于 JavaScript 开发的，所以它可以处理类似 HTTP 这样的文本协议，但是也可以用它来与数据库通信、操作图像以及上传文件。可以想象，如果只是用字符串完成上述任务将会相当困难。在早期，Node 通过将每个字节编码为文本字符来处理二进制数据，但现在已经证明该方法既浪费资源，速度又缓慢，还不可靠，并且难以控制。

为了使这类二进制数据处理任务变得容易些，Node 引入了一个二进制缓冲区实现，该实现以 Buffer 伪类中的 JavaScript API 形式暴露给外界。缓冲区的长度以字节为计量单位，并且可以随机地设置和获取缓冲区中的数据。

注意：Buffer 类的另一个特别之处是数据占用的内存并不是分配在 JavaScript VM 内存堆中，也就是说这些对象不会被垃圾收集算法处理：它会占据一个不会被修改的永久内存地址，这避免了因缓冲区内容的内存复制所造成的 CPU 浪费。

4.1 创建缓冲区

可以使用 UTF-8 编码格式的字符串来创建缓冲区，示例代码如下所示：

```
var buf = new Buffer('Hello World!');
```

只要指定编码格式作为构造函数的第二个参数，也可以用其他编码格式的字符串创建缓冲区：

```
var buf = new Buffer('8b76fde713ce', 'base64');
```

可以被接受的编码格式以及它们的标识符如下所示：

- ascii—ASCII。该编码格式仅适用于 ASCII 字符集。
- utf8—UTF-8。这是一种变宽度的编码格式，可以表示 Unicode 字符集中的任意字符，它是网络上的首选编码格式。在你没有指定编码格式参数的情况下，utf8 是默认的编码格式。
- base64—Base64。这种编码格式基于 64 个可打印的 ASCII 字符来表示二进制数据，Base64 通常用于在字符文档内嵌入可以被转换成字符串的二进制数据，在需要时又可以完整无损地转换回原来的二进制格式。

如果缓冲区没有用具体内容初始化，则可以通过指定容量大小来创建缓冲区，以备将来容纳数据。可以通过指定长度创建新的缓冲区，如下所示：

```
var buf = new Buffer(1024); // 创建长度为 1024 字节的缓冲区
```

4.2 在缓冲区中获取和设置数据

在创建或者获取缓冲区之后，你也许想查看或者修改其内容，那么可以使用[]操作符来访问缓冲区中的某个字节，如下所示：

```
var buf = new Buffer('my buffer content');
// 访问 buf 缓冲区的第 10 个字节
console.log(buf[10]); // -> 99
```

注意：当创建一个已被初始化的缓冲区时，请记住该缓冲区中包含的数据并非是 0，而是一些随机值。

```
var buf = new Buffer(1024);
console.log(buf[100]); // -> 5(某个随机值)
```

同样，也可以处理任意位置上的数据：

```
buf[99] = 125; // 将第 100 个位置上的数据设置为 125
```

注意：在某些情况下，一些缓冲区操作并不会产生错误，例如：

- 如果将缓冲区中的某个位置设置为一个大于 255 的数，那么将会用 256 对该数取模，最终将取模结果赋给该位置。
- 如果将缓冲区中某个位置设置为 256，那么该位置实际上是被赋值为 0。
- 如果将缓冲区中某个位置设置成 100.7 这样的小数，那么该位置仅会存储整数部分，本例中存储的是 100。
- 如果尝试给超出缓冲区边界的位置赋值，那么赋值操作将以失败告终，缓冲区也不会发生变化。

还可以通过查询缓冲区的 length 属性来获取其长度，如下所示：

```
var buf = new Buffer(100);
console.log(buf.length); // -> 100
```

然后可以使用缓冲区长度迭代缓冲区内容，来设置或者获取每个字节：

```
var buf = new Buffer(100);

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

在上面的代码中新建一个大小为 100 字节的缓冲区，并从 0～99 依次设置缓冲区中每个字节。

4.3　切分缓冲区

一旦创建或获取一个缓冲区，也许需要将此缓冲区的一部分提取出来。可以通过指定起始位置和结束位置来切分缓冲区，从而创建一个更小的缓冲区，如下所示：

```
var buffer = new Buffer("this is the content of my buffer");
var smallerBuffer = buffer.slice(8, 19);
console.log(smallerBuffer.toString()); // -> "the content"
```

注意，在切分缓冲区时并没有分配新的内存，也没有进行任何复制，新缓冲区还是使用父缓冲区的内存区域，只不过引用了父缓冲区不同的起始位置和/或结束位置，这一点蕴含以下几个方面的意思。

首先，如果程序修改了父缓冲区，并且修改只要影响到子缓冲区范围内的数据，那么子缓冲区也会修改。因为父缓冲区和子缓冲区是不同的 JavaScript 对象，所以很容易忽略这个问题，并因此导致一些错误。

其次，在用这种方法从父缓冲区创建一个较小的子缓冲区时，父缓冲区在操作结束后依然继续被保留，并不会被垃圾收集器回收，如果不谨慎处理的话，很容易造成内存泄露。

注意：如果担心因保留父缓冲区而产生内存泄露问题，可以使用 copy 方法来替代 slice 方法，下一节将会介绍 copy 方法。

4.4 复制缓冲区

还可以应用 copy 方法，将缓冲区的一部分复制到另一个缓冲区中，如下所示：

```
var buffer1 = new Buffer("this is the content of my buffer");
var buffer2 = new Buffer(11);

var targetStart = 0;
var sourceStart = 8;
var sourceEnd = 19;

buffer1.copy(buffer2, targetStart, sourceStart, sourceEnd);
console.log(buffer2.toString()); // -> "the content"
```

在上面的代码中，将源缓冲区的第 8 到第 19 个位置上的数据复制到目标缓冲区的起始位置。

4.5 缓冲区解码

缓冲区可以转换成一个 UTF-8 编码格式的字符串，如下所示：

```
var str = buf.toString();
```

也可以将缓冲区转换成其他指定编码格式的字符串，例如，如果要将一个缓冲区转换

成 base64 编码格式的字符串，可以这样做：

```
var b64Str = buf.toString("base64");
```

还可以使用 toString 方法，将 UTF-8 编码格式的字符串转换成 base64 编码格式的字符串，如下所示：

```
var utf8String = 'my string';
var buf = new Buffer(utf8String);
var base64String = buf.toString('base64')
```

4.6 本章小结

有时你不得不对二进制数据进行处理，但是“原生态”的 JavaScript 并没有为处理该任务提供明确的方式。

Node 中的 Buffer 类对访问连续内存块的操作进行了封装，可以处理内存中的数据以及切分缓冲区，还可以在两个缓冲区之间进行内存复制。

此外，还可以将缓冲区转换成某种编码格式的字符串，或者反过来将字符串转换成缓冲区，以便访问或处理每个比特。

第 5 章

使用事件发射器模式简化事件绑定

本章提要：

- 事件发射器模式简介
- 绑定和解绑定事件监听器
- 创建自定义的事件发射器

在 Node 中，很多对象都能够发射事件，例如，每当有新客户端连接时，TCP 服务器就会发射“connect”事件，又比如每读取一整块数据，文件流就会发射“data 事件，这些对象在 Node 中被称为事件发射器，它可以让程序员订阅感兴趣的事件，并将回调函数绑定到相关的事件上，每当事件发射器发射事件时，对应的回调函数就会被调用。这种发布/订阅模式非常类似于传统的图形用户界面(Graphic User Interface, GUI)模式，比如按钮被单击时，应用程序就会得到通知。通过这种模式，服务器端的程序就会对某些事件的发生做出响应，比如有客户端连接、套接字上有可用数据或者是文件关闭。

还可以创建自定义的事件发射器，实际上 Node 提供了一个伪类 EventEmitter，可以将其作为基类来创建自定义的事件发射器。

5.1 理解标准回调模式

异步编程并不使用函数返回值表示函数调用的结束，而是采用后继传递风格(continuation-passing style, CPS)取而代之。

> 后继传递风格是一种程序的流程控制权以后继的形式被显式传递的编程风格。(...)
>
> 按后继传递风格编写的函数以一个显式的“后继”作为额外参数，“后继”实际是一个函数，当 CPS 函数计算出返回值时，它就会调用后继函数，并将 CPS 函数的返回值作

为其参数。

出自维基百科——HTTP://EN.WIKIPEDIA.ORG/WIKI/CONTINUATION-PASSING_STYLE

在后继传递风格中，每个函数在执行完毕后都会调用一个回调函数，以使程序能够继续运行下去。正如将看到的那样，JavaScript 很适合这种编程风格，下面是一个在 Node 中将文件加载到内存的例子：

```
var fs = require('fs');
fs.readFile('/etc/passwd', function(err, fileContent) {
  if (err) {
    throw err;
  }
  console.log('file content', fileContent.toString());
});
```

在这个例子中，将一个内联匿名函数传递给 fs.readFile 函数。作为其第二个参数，其实这就是在使用 CPS，因为在这个内联匿名函数内将函数执行继续了下去。

如你所见，回调函数的第一个参数是一个错误对象，如果有错误发生，这个参数将会是 Error 类的一个实例，这是 Node 中应用 CPS 编程的一般模式。

5.2 理解事件发射器模式

在标准回调模式中，将一个回调函数作为参数传递给将被执行的函数，这种模式在函数执行结束后需要通知客户端的情况下工作得很好。但是如果在函数执行过程中发生了多个事件或者事件反复发生了多次，那这种模式就工作得不是那么好了。例如，如果有兴趣在套接字上每当有可用数据时都能够得到通知，标准回调模式就帮不上什么忙了，此时正是事件发射器模式派上用场的时候，并且可以用标准的接口清楚地区分事件发射器和事件监听器。

在使用事件发生器模式时，会涉及两个或者更多的对象，这些对象包括事件发射器以及一个或多个事件监听器。

顾名思义，事件发射器就是可以发射事件的对象，而事件监听器则是绑定到事件发射器上的代码，负责监听特定类型的事件，如下面的例子所示：

```
var req = http.request(options, function(response) {
  response.on("data", function(data) {
    console.log("some data from the response", data);
  });
  response.on("end", function() {
    console.log("response ended");
  });
});
req.end();
```

在上面的例子中，可以看到几个必要的步骤，在这几个步骤中，Node 的 http.request API(后面将介绍)创建了一个 HTTP 请求，并将其发送到远程服务器。第一行用到了后继传递风格，传递了一个当 HTTP 服务器响应时就会被调用的内联函数。HTTP 请求 API 在此处应用 CPS 是因为 http.request 函数执行完毕之后，程序还要继续运行下去。

http.request 函数执行完毕后，就会将响应对象传递给那个回调函数并调用它。响应对象是一个事件发射器，根据 Node 文档，响应对象能够发射包括 data、end 在内的很多事件，每当这些事件发生时，你注册的那些回调函数都会被调用。

一般而言，当需要在请求的操作完成后重新获取控制权时就使用 CPS 模式，当事件可能发生多次时就使用事件发射器模式。

5.3　理解事件类型

请注意发射的事件具有类型，类型用一个字符串表示，前面的例子中有“data”和“end”两种事件类型，它们都是由事件发射器定义的任意字符串，不过约定俗成的是，事件类型通常都是由不包含空格的小写单词组成的。

你无法通过编程来判断事件发射器到底能够发射哪些类型的事件，因为事件发射器 API 没有提供内省机制，因此你用到的 API 应该有文档来记录它能够发射的事件类型。

一旦有相关事件发生，事件发射器就会调用相应的事件监听器，并将相关数据作为参数传递给事件监听器。在前面的 http.request 示例中，“data”事件的回调函数接收一个 data 对象，作为其第一个也是唯一的一个参数，而“end”事件回调函数则不接收任何参数。这些参数是 API 协议的一部分，也是随机指定的，这些回调函数的参数签名也会在每个事件发射器的 API 规范里有所说明。

虽然事件发射器是服务任意类型事件的通用接口，但在 Node 的实现中有一个特例，即“error”事件，Node 中的大多数事件发射器实现在程序发生错误时都会发射“error”事件。如果程序员选择不监听“error”事件，则当“error”事件发生时，事件发射器会注意到它，并向上抛出一个未捕获的异常。

可以在 Node REPL 里运行下面的代码来测试一下效果，这段代码模拟了一个能发射两种类型事件的事件发射器：

```
var em = new (require('events').EventEmitter)();
em.emit('event1');
em.emit('error', new Error('My mistake'));
```

然后会看到如下所示的输出信息：

```
> var em = new (require('events').EventEmitter)();
undefined
> em.emit('event1');
false
> em.emit('error', new Error('My mistake'));
```

```
Error: My mistake
    at repl:1:18
    at REPLServer.eval (repl.js:80:21)
    at repl.js:190:20
    at REPLServer.eval (repl.js:87:5)
    at Interface.<anonymous> (repl.js:182:12)
    at Interface.emit (events.js:67:17)
    at Interface._onLine (readline.js:162:10)
    at Interface._line (readline.js:426:8)
    at Interface._ttyWrite (readline.js:603:14)
    at ReadStream.<anonymous> (readline.js:82:12)
>
```

随便发射一个名为“event1”的事件没有任何效果，但是在发射“error”事件时，错误就会被抛出到堆栈。如果程序不是运行在 REPL 命令行环境里，就会因为这个未捕获到的错误而挂起。

一般而言，应该总是监听错误事件，并对其进行恰当地处理。

5.4 应用事件发生器 API

任何实现了事件发射器模式的对象(比如 TCP 套接字、HTTP 请求等)，都实现了如下所示的一组方法：

- .addLinstener 和.on——为指定类型的事件添加事件监听器。
- .once——为指定类型的事件绑定一个仅会被调用一次的事件监听器。
- .removeEventListener——删除绑定到指定事件上的某个指定的事件监听器。
- .removeAllEventListener——删除绑定到指定事件上的所有事件监听器。

下面将详细介绍它们。

5.4.1 使用.addListener( )或.on( )绑定回调函数

通过指定事件类型和回调函数，就可以注册当事件发生时所要调用的操作。例如，当有可用的数据块时，文件可读流就会发射“data”事件，下面的代码展示了如何通过传入一个回调函数来通知发生了“data”事件：

```
function receiveData(data) {
  console.log("got data from file read stream: %j", data);
}
readStream.addListener("data", receiveData);
```

还可以用函数.on 来代替.addListener 函数，它只是.addListen 函数的简写形式，下面这段代码和上面那段代码是等效的：

```
function receiveData(data) {
  console.log("got data from file read stream: %j", data);
}
```

```
readStream.on("data", receiveData);
```

在上面的代码中用一个事先声明的命名函数作为回调函数，当然也可以使用内联匿名函数代替命名函数以简化代码：

```
readStream.on("data", function(data) {
  console.log("got data from file read stream: %j", data);
});
```

正如前面所指出的那样，传递给回调函数的参数取决于特定的事件发射器对象与事件类型，它并没有被标准化，“data”事件可能会传递数据缓冲区，“error”事件可能会传递一个错误对象，而流的“end”事件则不会向事件监听器传递任何参数。

5.4.2　绑定多个事件监听器

事件发射器模式允许多个事件监听器监听同一事件发射器发射的同一类型的事件，例如：

```
readStream.on("data", function(data) {
    console.log('I have some data here.');
});
readStream.on("data", function(data) {
    console.log('I have some data here too.');
});
```

在上面的代码中，readStream 对象的“data”类型事件上绑定了两个函数，每当 readStream 对象发射“data”事件时，就会看到如下所示的输出信息：

```
I have some data here.
I have some data here too.
```

根据事件类型，事件发射器负责按照事件所绑定的监听器的注册顺序依次调用事件监听器，这意味着以下两件事：

- 某个事件监听器也许并不会在事件发射之后立即被调用，也许在它之前会有别的事件监听器被调用。
- 异常被抛出到堆栈并不正常，它通常是由代码中的错误引起的。当事件被发射时，如果其中某个事件监听器在被调用时抛出错误，可能会导致一些事件监听器永远都不会被调用，在这种情况下，事件发射器将会捕捉到错误，也许还会处理它。考虑下面这个例子：

```
readStream.on("data", function(data) {
    throw new Error("Something wrong has happened")
});
readStream.on("data", function(data) {
    console.log('I have some data here too.');
});
```

本例中由于第一个事件监听器抛出了一个错误，所以第二个事件监听器也就不会被调用。

5.4.3 使用.removeListener( )从事件发射器中删除一个事件监听器

如果希望当一个对象发射了某个特定事件时不再收到通知的话，可以通过指定事件类型和回调函数来取消事件监听器的注册，示例代码如下所示：

```
function receiveData(data) {
  console.log("got data from file read stream: %j", data);
}
readStream.on("data", receiveData);
// ...
readStream.removeListener("data", receiveData);
```

在上面的例子中，最后一行把一个可能在将来被随时调用的事件监听器从事件发射器对象中删除。

为了删除指定的事件监听器，就必须为回调函数命名，因为回调函数名至少要在两处用到——增加和删除事件监听器时。

5.4.4 使用.once()使回调函数最多执行一次

如果想监听最多只发生一次的事件，或者只是对某个类型的事件第一次发生时感兴趣，则可以使用.once()函数。

该方法增加了一个事件监听器，并在第一个事件发生后删除它。

```
function receiveData(data) {
  console.log("got data from file read stream: %j", data);
}
readStream.once("data", receiveData);
```

在上面的代码中，receiveData 函数只会被调用一次，如果在 readStream 对象上发射 data 事件，receiveData 回调函数仅会被触发一次。

这是个方便的方法，因为可以很容易就实现它，如下所示：

```
var EventEmitter = require("events").EventEmitter;
EventEmitter.prototype.once = function(type, callback) {
  var that = this;
  this.on(type, function listener() {
    that.removeListener(type, listener);
    callback.apply(that, arguments);
  });
};
```

在上面的代码中，重新定义了 EventEmitter.prototype.once 函数，这相当于重新定义了继承自 EventEmitter 所有对象的 once 方法。在这里只是使用了.on()方法，并在获取事件后使用.removeEvetnListener()方法取消了回调函数的注册，同时调用原来的回调函数。

注意：在上面的代码中用到了 function.apply()方法，该方法接受一个对象以及一个参数数组，并将接受的对象作为一个隐含的 this 变量。在上面的例子中，向回调函数传入了一个未经修改的参数数组，也即允许事件发射器将所有参数原封不动且透明地传送给回调函数。

5.4.5　使用.removeAllListeners()从事件发射器删除所有事件监听器

可以从事件发射器中删除所有针对某一指定类型的事件而注册的事件监听器，如下所示：

```
emitter.removeAllListeners(type);
```

例如，如果想删除进程中断事件的所有事件监听器，那么可以这样做，如下所示：

```
process.removeAllListener("SIGTERM");
```

注意：一般而言，本书建议你仅在确实知道要删除的对象时才使用该函数，否则就应该让应用程序的其他部分删除事件监听器集合，或者也可以让应用程序的那些部分自己负责删除事件监听器集合。但不管怎样，在一些罕见的情况下，该函数还是很有用的，例如在按顺序关闭事件发射器甚至关闭整个进程时。

5.5　创建事件发射器

事件发射器提供了一种很好的方法来构建更为通用的编程接口。在一个常见易懂的编程模式中，客户端用绑定事件来代替调用回调函数，这会使程序更加灵活。

此外，通过使用事件发射器，还可以免费获得一些特性，比如在同一事件上绑定多个独立的事件监听器。

5.5.1　从 Node 事件发射器继承

如果有兴趣在自己的应用程序中自始至终都使用 Node 事件发射器，那么可以创建一个继承自 EventEmitter 的伪类，示例代码如下所示：

```
util = require('util');
var EventEmitter = require('events').EventEmitter;
// Here is the MyClass constructor:
var MyClass = function() {
}
util.inherits(MyClass, EventEmitter);
```

注意：uti.inherits 建立了一条原型链，使 MyClass 类实例能够使用 EventEmitter 类的原型方法。

5.5.2 发射事件

通过创建继承自 EventEmitter 的类，MyClass 的类实例就可以发射事件了：

```
MyClass.prototype.someMethod = function() {
  this.emit("custom event", "argument 1", "argument 2");
};
```

此处当 someMethod 方法被 MyClass 类实例调用时，就会发射一个名为 custom event 的事件，该事件还可以发射若干数据，本例中是两个字符串："argumrnt 1"和"argumrnt 2"，它们将会作为参数被传递给事件监听器。

MyClass 类实例的客户端可以监听 custom event 事件，示例代码如下所示：

```
var myInstance = new MyClass();
myInstance.on('custom event', function(str1, str2) {
  console.log('got a custom event with the str1 %s and str2 %s!', str1, str2);
});
```

例如，可以创建一个名为 Ticker 的伪类，让它每秒发射一个"tick"事件：

```
var util = require('util'),
EventEmitter = require('events').EventEmitter;
var Ticker = function() {
  var self = this;
  setInterval(function() {
    self.emit('tick');
  }, 1000);
};
util.inherits(Ticker, EventEmitter);
```

Ticker 类的客户端可以实例化该类，并监听"tick"事件，如下所示：

```
var ticker = new Ticker();
ticker.on("tick", function() {
  console.log("tick");
});
```

5.6 本章小结

事件发射器模式是 Node 中的一种可重入模式，可以用它将事件发射器器对象与关注

一组特定事件的代码解耦。

可以使用 event_emitter.on()为特定类型的事件注册事件监听器，还可以使用 event_emitter.removeListener()取消注册。

此外，还可以通过继承 EventEmitter 类以及简单地使用.emit()函数来创建自定义的事件发射器。

第 6 章

使用定时器制定函数执行计划

本章提要：

- 推迟函数的执行
- 取消执行计划
- 制定函数的周期性执行计划
- 将函数执行推迟到下一轮事件循环

如果你已习惯于 JavaScript 编程，也许会用到 setTimeout 和 setInterval 函数，这两个函数可以将某个函数推迟一段时间执行。举个例子，如果下面这段代码载入网页，会在 1 秒后将字符串“Hello there”附加到页面文档：

```
var oneSecond = 1000 * 1; // 1 秒 = 1000 x 1 毫秒
setTimeout(function() {
   document.write('<p>Hello there.</p>');
}, oneSecond);
```

而 setInterval 允许一个函数以指定的时间间隔重复执行，如果将下面这段代码加入网页，那么它将会每隔 1 秒在页面文档后附加一次短语“Hello there”：

```
var oneSecond = 1000 * 1; // one 秒 = 1000 x 1 毫秒
setInterval(function() {
   document.write('<p>Hello there.</p>');
}, oneSecond);
```

之所以出现对这些函数的需求，是因为网络早已变成了一个构建应用程序的平台，而不仅仅用于创建静态网页。这些任务计划函数辅助开发人员构建周期性的表单验证、延迟远程数据同步以及各种各样需要延时响应的用户界面交互。Node 全面实现了这样一组函数，它们被用来在服务器端辅助很多不同进程的周期性或者推迟执行，这些进程包括缓存

过期、连接池清理、会话超时、轮询等。

现在介绍将这些函数，然后讨论它们的一些局限性。

6.1 使用 setTimeout 推迟函数执行

setTimeout 可以让你制定一个在将来某个时刻让指定函数执行一次的执行计划，下面是一个示例：

```
var timeout_ms = 2000; // 2 seconds
var timeout = setTimeout(function() {
  console.log("timed out!");
}, timeout_ms);
```

与浏览器端的 JavaScript 完全一样，setTimeout 接受要被推迟执行的函数作为第一个参数，接受推迟执行的时间(以毫秒为单位)作为第二个参数。

调用 setTimeout 会返回一个超时句柄，该句柄是一个内部对象，除了将它作为参数调用 clearTimeout 取消函数执行外，不能用于它途。

6.2 使用 clearTimeout 取消函数执行

一旦获得了一个超时句柄，就可以用它调用 clearTimeout 函数，来取消函数的执行，如下所示：

```
var timeoutTime = 1000; // 1 秒
var timeout = setTimeout(function() {
  console.log("timed out!");
}, timeoutTime);
clearTimeout(timeout);
```

在上面的代码中，超时句柄永远不会起作用，“time out”字符串也永远不会打印至控制台。此外，也可以在将来的某个时刻取消已计划好的函数执行，下面这段代码展示了发生这种情况的一个示例：

```
var timeout = setTimeout(function A() {
  console.log("timed out!");
}, 2000);
setTimeout(function B() {
  clearTimeout(timeout);
}, 1000);
```

本例中，安排了两个要在将来执行的函数——A 和 B，A 函数被安排在 2 秒后执行，B 函数被安排在 1 秒后执行。因为在本例中，B 函数执行时取消了 A 函数的执行，所以 A 函数永远不会执行。

6.3　制定和取消函数的重复执行计划

setInterval 与 setTimeout 类似，但不同的是它会安排一个函数每隔一个指定的时间间隔重复执行一次，也许你想用它周期性地触发一段程序，用来完成一些类似清理、收集、登录、获取数据、轮询等需要重复执行的任务。

下面这段代码会每隔 1 秒向控制台输出一个“tick”字符串：

```
var period = 1000; // 1秒
setInterval(function() {
  console.log("tick");
}, period);
```

如果不想让这个过程无限期地进行下去，或者不想让其自行结束，可以调用 clearInterval()来取消它的执行。

setInterval 返回一个执行计划句柄，可以将该句柄用作 clearInterval 的参数来取消执行计划。

```
var interval = setInterval(function() {
  console.log("tick");
}, 1000);
// …
clearInterval(interval);
```

6.4　使用 process.nextTick 将函数执行推迟到下一轮事件循环

有时，浏览器端的 JavaScript 程序员会用 setTimout(callback, 0)将任务推迟一段很短的时间，第二个参数是 0 毫秒，它告诉 JavaScript 运行时在所有被挂起的事件处理完毕后立刻执行这个回调函数，有时候这种技术被用来执行一些并不需要立刻执行的操作。例如，有时这种技术用于在用户事件处理完毕后开始播放动画或者做一些计算。

顾名思义，Node 中的事件循环在一个处理事件队列的循环里运行，事件循环每执行一次就被称为一个“tick”。

可以安排在事件循环开始下一轮(下一个 tick)时调用回调函数一次，然而，setTimeout 使用的是 JavaScript 运行时的内部执行队列，而不是使用事件循环。

通过用 process.nextTick(callback)来取代 setTimeout(callback, 0)，回调函数会在事件队列内的所有事件处理完毕后立刻执行，它要比激活 JavaScript 的超时队列快得多(以 CPU 时间来衡量)。

你把函数延迟到下一轮事件循环再执行，如下所示：

```
process.nextTick(function() {
  my_expensive_computation_function();
});
```

注意：process 对象是 Node 中为数不多的全局对象之一。

6.5 阻塞事件循环

Node 和 JavaScript 的运行时采用的是单线程事件循环。每次循环，运行时通过调用相关回调函数来处理队列内的下个事件。当事件处理完毕，事件循环获取处理的结果并开始处理下个事件，如此反复，直到事件队列为空。如果其中某个回调函数占用了很长时间，事件循环在此期间就不能处理其他挂起的事件，这会让应用程序或服务变得非常缓慢。

在处理事件时，如果使用了内存敏感或者处理器敏感的函数，会导致事件循环变得缓慢，而且造成大量事件堆积，不能得到及时处理，甚至堵塞队列。

下面是一个阻塞事件循环的例子：

```
process.nextTick(function nextTick1() {
  var a = 0;
  while(true) {
    a ++;
  }
});

process.nextTick(function nextTick2() {
  console.log("next tick");
});

setTimeout(function timeout() {
  console.log("timeout");
}, 1000);
```

在上面的例子中，nextTick2 和 timeout 函数无论等待多久都没机会运行，因为事件循环被 nextTick 函数里的无限循环阻塞了，即使 timeout 函数被计划在 1 秒钟后执行也不可能。

在使用 setTimeout 时，回调函数会被添加到执行计划队列，而在这个例子里它们甚至不会被添加到队列。虽然这是个极端例子，但是可以看到，执行一个处理器敏感的任务时可能会堵塞或者拖慢事件循环。

6.6　退出事件循环

通过使用 process.nextTick，可以将一个非关键性的任务推迟到事件循环的下一轮再执行，这样可以释放事件循环，让它可以继续执行其他挂起的事件。

查看下面的例子，如果需要删除一个之前创建的临时文件，但又不想在对客户端做出响应之前进行该操作，就可以延迟删除操作，如下所示：

```
stream.on("data", function(data) {
  stream.end("my response");
  process.nextTick(function() {
    fs.unlink("/path/to/file");
  });
});
```

6.7　使用 setTimeout 代替 setInterval 强制函数串行执行

假如希望函数 my_async_function 重复执行一些 I/O 操作(比如解析日志文件)，可以使用 setInterval 函数，如下所示：

```
var interval = 1000;
setInterval(function() {
  my_async_function(function() {
    console.log('my_async_function finished!');
  });
});
```

不过，你必须确保这些函数不会同时执行，但是使用 setInterval 就无法保证这一点。假如 my_async_function 函数的执行时间比 interval 变量多了 1 毫秒，它们就会被同时执行，而不是按顺序串行执行。

因此，需要强制指定 my_async_function 函数执行结束与下个 my_async_function 函数开始执行之间的时间间隔，这个时间间隔应该由 interval 指定，可以这样做：

```
var interval = 1000; // 1 second
(function schedule() {
  setTimeout(function do_it() {
    my_async_function(function() {
      console.log('async is done!');
      schedule();
    });
  }, interval);
}());
```

在上面的代码中，声明了一个函数 schedule(第 3 行)，并且在声明后立刻调用它(第 10 行)，schedule 函数安排 do_it 函数在 1 秒(选定的 interval 值)后执行。1 秒钟过后，第 5 行

的 my_async_function 函数会被调用，当它执行完毕后，会调用作为其参数的匿名回调函数(第 6 行)，而这个匿名回调函数又会再次安排 do_it 函数在 1 秒钟后重新执行，这样，就会重新启动循环过程。

6.8 本章小结

可以使用 setTimeout()函数预先制定函数的执行计划，并可用 clearTimeout()函数取消执行计划。还可以使用 setInterval()让某个函数周期性地重复执行，相应的，可以使用 clearInterval()取消这个重复执行计划。

如果由于执行了一个处理器敏感的操作而堵塞了事件循环，那些原计划应该被执行的函数就会被推迟执行，甚至永远无法执行。所以不要在事件循环内使用 CPU 敏感的操作。另外，可以使用 process.nextTick()将函数的执行延迟到事件循环的下一轮。

I/O 操作和 setInterval()一起使用时，无法保证在任意时刻只有一个挂起的调用，但可以使用递归函数和 setTimeout()函数来规避这个潜在的问题。

第Ⅲ部分

文件、进程、流和网络

- 第 7 章：查询和读写文件
- 第 8 章：创建和控制外部进程
- 第 9 章：读写数据流
- 第 10 章：构建 TCP 服务器
- 第 11 章：构建 HTTP 服务器
- 第 12 章：构建 TCP 客户端
- 第 13 章：创建 HTTP 请求
- 第 14 章：使用用户数据报
- 第 15 章：使用 TLS/SSL 保证服务器的安全性
- 第 16 章：使用 HTTPS 保证 HTTP 服务器的安全性

第 7 章

查询和读写文件

本章提要：

- 处理文件路径
- 从文件路径中提取信息
- 理解文件描述符
- 应用 fs.stat()获取文件统计信息
- 打开、读写和关闭文件
- 避免文件描述符泄露

Node 中有一组流 API，它们可以像处理网络流那样处理文件。流 API 用起来非常方便，但是它只允许以连续的方式处理文件。如果需要在文件的指定位置进行读写，流 API 就行不通了。所以，必须使用一些更底层的操作来对文件系统自身进行处理。

本章内容囊括了文件处理的基础，包括如何打开文件、读取文件的指定部分、写入文件以及关闭文件。

Node 的很多文件 API 几乎是 UNIX(POSIX)中对应文件 API 的翻版，比如，使用文件描述符的方式就和 UNIX 中的一样。文件描述符句柄在 Node 中也是一个整数，代表进程文件描述符表的某个入口的索引。

有三个特殊的文件描述符——1、2 和 3。它们分别表示标准输入文件、标准输出文件和标准错误文件的描述符。顾名思义，标准输入是一个只读流，进程可以用它读取控制台的输入或者其他进程传送的数据。标准输出和标准错误是只对外输出的文件描述符，它们可以向控制台、其他进程或文件输出数据。标准错误专门用作输出错误信息，而标准输出负责输出普通的进程信息。

一旦进程启动完毕，就能使用上述几个文件描述符了，它们其实并不对应实际的文件。

与操作网络数据流非常相似，你不能在文件内指定的位置读写。使用这些文件描述符，只能连续地读写，已写入的数据就不能再修改了。

文件则不受这样的限制。例如，在 Node 中可以创建只能向尾部追加数据的文件，也能在指定的位置写入数据。

几乎所有处理文件的代码都会涉及处理文件路径，本章首先会介绍这些实用函数，然后再深入讲解文件读取和文件内容操作。

7.1　处理文件路径

文件路径用来表示具体的文件，它可以分为相对路径和绝对路径。可以连接文件路径、提取文件名信息，甚至可以检测文件是否存在。

在 Node 中，可以使用字符串来处理文件路径，但是这样做会使问题变得难以处理。例如，当你想连接路径的不同部分时，有些部分是以“/”结尾的，有些却不是，而且在不同的操作系统中路径分隔符也不同，考虑到这些情况，问题就会变得十分麻烦。

幸好，Node 中有一个 path 模块可以帮助你规范化、连接和解析路径，还可以将绝对路径转换成相对路径、提取路径的组成部分以及确定路径是否存在。总而言之，path 模块只是对字符串进行处理，而并不需要通过和文件系统交互来验证字符串(path.exists()函数除外)。

7.1.1　规范化路径

在存储和使用路径之前对其进行规范化通常是个好主意，比如由用户输入或者由配置文件获得的文件路径，还有由两个或多个路径连接得到的路径通常都需要规范化。为此，可以使用 path 模块中的 normalize()函数来规范化路径字符串，而且它还能处理..、.以及//。例如：

```
var path = require('path');
path.normalize('/foo/bar//baz/asdf/quux/..');
// => '/foo/bar/baz/asdf'
```

7.1.2　连接路径

通过使用 path.join()函数，可以连接任意多个路径字符串，此时需要将所有路径字符串依次传递给 path.join()函数作为其参数，如下所示：

```
var path = require('path');
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');
// => '/foo/bar/baz/asdf'
```

如你所见，path.join()函数同时也对路径进行了规范化。

7.1.3　解析路径

可以使用 path.resolve()函数将多个路径解析为一个规范化的绝对路径，该函数的作用就好像是对这些路径挨个进行 cd 操作，但与 cd 操作不同的是，这些路径可以是文件，并且可以不必实际存在——即它不会利用底层的文件系统来尝试判断路径是否存在，而只是对路径字符串进行处理。

例如：

```
var path = require('path');
path.resolve('/foo/bar', './baz');
// => /foo/bar/baz
path.resolve('/foo/bar', '/tmp/file/');
// => /tmp/file
```

如果解析的结果不是绝对路径，那么 path.resolve 函数会将当前工作目录附加到解析结果的前面，下面是一个例子：

```
path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif');
// 如果当前工作目录是/home/myself/node/，那么上面的语句将返回
// => /home/myself/node/ wwwroot/ static_files/gif/image.gif
```

7.1.4　查找两个绝对路径之间的相对路径

通过使用 path.relative()函数，可以确定如何从一个绝对路径跳转到另一个绝对路径，例如：

```
var path = require('path');
path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb');
// => ../../impl/bbb
```

7.1.5　提取路径的组成部分

以路径/foo/bar/myfile.txt 为例，也许你想获得其父目录(/foo/bar)的内容，或者读取相同目录下的其他文件。为此，必须使用 path.dirname 函数获取文件路径的目录部分，例如：

```
var path = require('path');
path.dirname('/foo/bar/baz/asdf/quux.txt');
// => /foo/bar/baz/asdf
```

在某些情况下，也许想从文件路径中提取文件名，即文件路径的最后一部分。此时可以使用 path.basename 函数完成该任务：

```
var path = require('path');
path.basename('/foo/bar/baz/asdf/quux.html')
// => quux.html
```

文件路径可能还包括一个扩展名，它通常是文件名中.字符(包含最后一个.字符)后面的那部分字符串。

可以将扩展名传递给 path.basename 函数，作为其可选的第二个参数，以此在文件名中提取扩展名，如下所示：

```
var path = require('path');
path.basename('/foo/bar/baz/asdf/quux.html', '.html');
// => quux
```

当按照这种方式使用 path.basename 函数时，必须事先知道扩展名，可以使用 path.extname() 函数获取扩展名，如下所示：

```
var path = require('path');
path.extname('/a/b/index.html');
// => '.html'
path.extname('/a/b.c/index');
// => ''
path.extname('/a/b.c/.');
// => ''
path.extname('/a/b.c/d.');
// => '.'
```

7.1.6　确定路径是否存在

到目前为止，所进行的路径处理和提取都还没有涉及底层的文件系统，但有些时候也许需要判断给定的路径究竟是否存在。例如，在创建目录或文件之前，也许首先要确定它并不存在。可以使用 path.exist()函数达到这一目的，如下所示：

```
var path = require('path');
path.exists('/etc/passwd', function(exists) {
  console.log('exists:', exists);
  // => true
});
path.exists('/does_not_exist', function(exists) {
  console.log('exists:', exists);
  // => false
});
```

注意： Node 0.8 版用 fs.exists 函数代替了 path.exists 函数，两者在语义上没有任何区别。

为了使用 fs.exists 函数，只需要导入“fs”模块来代替“path”模块：

```
var fs = require('fs');
fs.exists('/does_not_exist', function(exists) {
  console.log('exists:', exists);
  // => false
});
```

因为 path.exists()函数进行了 I/O 操作，因此它是异步的，所以必须要向其传递一个回调函数，当 I/O 操作结束后通知这个回调函数。如果不是在 I/O 回调函数当中，也可以使用同步版本的 path.existsSync()函数，功能完全一样，只是它不使用回调函数，而是直接返回结果：

```
var path = require('path');
path.existsSync('/etc/passwd');
// => true
```

7.2　fs 模块简介

fs 模块中存放了所有文件查询和操作函数，利用这些函数，可以查询文件的统计信息、打开文件、读写文件和关闭文件：

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

查询文件的统计信息

有时可能需要了解文件的一些特征，例如大小、创建时间或者权限，可以使用 fs.stat 函数来查询文件或目录的元信息：

```
var fs = require('fs');
fs.stat('/etc/passwd', function(err, stats) {
  if (err) { throw err;}
  console.log(stats);
});
```

运行上述这段代码会输出如下所示的信息：

```
{ dev: 234881026,
  ino: 95028917,
  mode: 33188,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  size: 5086,
  blksize: 4096,
  blocks: 0,
  atime: Fri, 18 Nov 2011 22:44:47 GMT,
  mtime: Thu, 08 Sep 2011 23:50:04 GMT,
  ctime: Thu, 08 Sep 2011 23:50:04 GMT }
```

fs.stat()函数调用会将 stats 类的一个实例传递给其回调函数，可以用这个实例调用以下函数：

- stats.isFile()——如果是标准文件而不是目录、套接字、符号链接或者设备的话，返回 true。
- stats.isDirectory()——如果是目录，返回 true。
- stats.isBlockDevice()——如果是块设备，返回 true，在大多数 UNIX 系统中块设备一般都位于/dev 目录下。
- stats.isCharacterDevice()——如果是字符设备，返回 true。
- stats.isSymbolicLink()——如果是符号链接，返回 true。
- stats.isFifo()——如果是 FIFO(UNIX 中一种特殊类型的命名管道)，返回 true。
- stats.isSocket()——如果是 UNIX 套接字，返回 true。

7.3 打开文件

在读取或者处理文件之前，必须首先使用 fs.open 函数打开文件，然后使用文件描述符调用所提供的回调函数，稍后就可以用这个回调函数对打开的文件进行读写。

```
var fs = require('fs');
fs.open('/path/to/file', 'r', function(err, fd) {
    // 获取文件描述符 fd
});
```

fs.open()函数的第一个参数是文件路径。第二个参数是一个标志位，表明文件以何种模式打开，标志位可以是 r、r+、w、w+、a 或者 a+。下面是这些标志位的含义(摘自 UNIX 手册的 fopen 页)：

- r——打开文本文件进行读取。数据流的位置在文件的起始处。
- r+——打开文件进行读写。数据流的位置在文件的起始处。
- w——如果文件存在，将其清零，如果不存在，就创建文件写入数据。数据流的位置在文件的起始处。
- w+——打开文件进行读写，如果文件不存在就创建它，如果存在就将文件清零，数据流的位置在文件的起始处。
- a——打开文件写入数据。如果文件不存在就创建它，如果存在就将文件清零，数据流的位置在文件的结尾处，此后的写操作都将数据追加到文件后面。
- a+——打开文件进行读写。如果文件不存在就创建它，如果存在就将文件清零，数据流的位置在文件的结尾处，此后的写操作都将数据追加到文件后面。

7.4 读取文件

文件一旦打开，就可以对其进行读取，但在读取之前，必须创建一个缓冲区来放置数据。缓冲区被传递到充满文件数据的+s.read 函数。

```
var fs = require('fs');
fs.open('./my_file.txt', 'r', function opened(err, fd) {
  if (err) { throw err }
  var readBuffer = new Buffer(1024),
      bufferOffset = 0,
      bufferLength = readBuffer.length,
      filePosition = 100;
  fs.read(fd,
          readBuffer,
          bufferOffset,
          bufferLength,
          filePosition,
          function read(err, readBytes) {
            if (err) { throw err; }
            console.log('just read ' + readBytes + ' bytes');
            if (readBytes > 0) {
              console.log(readBuffer.slice(0, readBytes));
            }
          });
});
```

在上面的代码中，尝试打开一个文件，当文件成功打开之后(调用 opened 函数)，要求从所提供的缓冲区的第 100 个字节(第 11 行)开始，读取随后的 1024 个字节的数据。

fs.read()调用的最后一个参数是一个回调函数(第 16 行)，当下列三种情况之一发生时就会调用该回调函数：

- 有错误发生。
- 成功读取了数据。
- 没有数据可读。

当发生错误时，回调函数通过第一个参数获取一个错误对象，否则该参数为 null。如果成功读取了数据，第二个参数(readBytes)获得读入缓冲区的字节数，如果其值为 0，则代表到达文件结尾。

注意：一旦将缓冲区传递给 fs.open()函数，缓冲区的控制权就转交给了 read 命令，只有在回调函数被调用之后，缓冲区的控制权才会返还给你。在此之前不应对该缓冲区进行读写或让其他函数调用该缓冲区，否则可能会读取不完整的数据，更糟糕的情况是可能会并发地往该缓冲区中写入数据。

7.4.1　写入文件

通过向 fs.write()函数传递一个包含数据的缓冲区，可以向一个已打开的文件中写入数据：

```
var fs = require('fs');
fs.open('./my_file.txt', 'a', function opened(err, fd) {
  if (err) { throw err; }
  var writeBuffer = new Buffer('writing this string'),
      bufferPosition = 0,
      bufferLength = writeBuffer.length, filePosition = null;
  fs.write( fd,
            writeBuffer,
            bufferPosition,
            bufferLength,
            filePosition,
            function wrote(err, written) {
              if (err) { throw err; }
              console.log('wrote ' + written + ' bytes');
            });
});
```

在上面的例子中，第 3 行尝试以追加模式打开一个文件，然后在第 9 行向该文件写入数据，传递的缓冲区包含以下一些信息：

- 准备写入缓冲区的数据
- 待写入数据在缓冲区中的起始位置
- 待写入数据的长度
- 从文件中的什么位置开始写入数据
- 写入操作结束后被调用的回调函数 wrote

本例中，向文件中写入数据的起始位置是 null，表明写入操作将从当前文件的游标处开始。由于是以追加模式打开文件的，因此此时文件的游标位于文件的结尾处。

再次提醒，和读取操作一样，在 fs.write 函数执行之后请不要使用提供的缓冲区。一旦将其传递给 fs.write 函数，缓冲区就处于写入操作的控制之下，直到回调函数被调用之后才能重新使用它。

7.4.2 关闭文件

你也许已经注意到了，到目前为止，在本章见过的所有例子中都没有关闭文件的代码，这是因为它们都是只需要运行一次的简单小例子。当 Node 进程退出之后，操作系统会确保所有文件都被关闭。

然而，在实际的应用程序中，一旦打开一个文件，最后就必须关闭它。为此，必须跟踪那些已经打开的文件描述符，在最后不再需要它们的时候就使用 fs.close(fd[,callback)关闭它们。如果不细心的话，很容易就会遗漏某个文件描述符。下面是一个例子，在这个例子中提供了一个 openAndWriteToSystemLog 函数，该函数具有关闭文件的功能：

```
var fs = require('fs');
function openAndWriteToSystemLog(writeBuffer, callback) {
  fs.open('./my_file', 'a', function opened(err, fd) {
    if (err) { return callback(err); }
```

```
function notifyError(err) {
  fs.close(fd, function() {
    callback(err);
  });
}
var bufferOffset = 0,
    bufferLength = writeBuffer.length,
    filePosition = null;
fs.write( fd, writeBuffer, bufferOffset, bufferLength, filePosition,
  function wrote(err, written) {
      if (err) { return notifyError(err); }
      fs.close(fd, function() {
        callback(err);
      });
    }
  );
});
}
openAndWriteToSystemLog(
   new Buffer('writing this string'),
   function done(err) {
     if (err) {
       console.log("error while opening and writing:", er
       return;
     }
     console.log('All done with no errors');
   }
);
```

在上面的代码中，提供了一个名为 openAndWriteToSystemLog 的函数，它接受一个缓冲区用于写入数据，它还接受了一个回调函数，这个回调函数在操作结束或出现错误时被调用。如果发生了错误，错误对象就会被传递给回调函数，作为其第一个参数。

注意内部函数 notifyError，它会关闭文件描述符并报告发生的错误。

注意：到目前为止，你已经学习了使用底层的原语操作来打开、读写和关闭文件，然而 Node 还有一组更高级的构造函数，允许以更简单的方式处理文件。

例如，也许你想以一种安全的方式，让两个或者多个 write 操作并发地向文件中追加数据，此时应该使用 WriteStream。

还有，当想读取文件中的某个区域时，可以考虑使用 ReadStream。

以上两个用例会在第 9 章“读写数据流”中介绍。

7.5　本章小结

在处理文件时，绝大多数时候都需要处理和提取文件的路径。通过使用 path 模块，可以连接路径、规范化路径、计算路径之间的差别，以及将相对路径转换成绝对路径。还可以提取指定文件路径的扩展名、文件名和目录等路径的组成部分。

Node 在 fs 模块中提供了一组底层 API 来访问文件系统，底层文件系统使用文件描述符来处理它们对应的文件。可以用 fs.open 函数打开文件、用 fs.write 函数写入文件、用 fs.read 函数读取文件以及用 fs.close 函数关闭文件。

当出现错误时，应该总是通过对错误进行正确的处理来关闭文件——此时应确保在调用返回之前关闭所有的文件描述符。

第 8 章

创建和控制外部进程

本章提要：

- 执行命令和子进程
- 子进程发送和接收数据
- 向子进程发送信号后终止子进程

Node 是被设计用来高效处理 I/O 操作的，但正如你所见，某些类型的程序并不适用于这种模式。比如当用 Node 处理一个 CPU 密集型任务时可能会阻塞事件循环，并会因此降低使用程序的响应能力。替代的办法是，CPU 密集型任务应该被分配给另一个进程处理，从而释放事件循环。Node 允许创建进程，并将这些进程当作启动它们的进程(父进程)的子进程。在 Node 中，子进程和父进程能够进行双向通信，并且在一定程度上，父进程可以监视和控制子进程。

另一种要使用子进程的情况只是执行一个外部命令，并在执行结束后让 Node 获得返回的结果。例如，你也许想执行一个 UNIX 命令、脚本或者其他不能在 Node 中直接执行的实用程序。

本章将向你展示如何生成外部命令、创建子进程、与子进程通信以及终止子进程，重点是在 Node 进程之外实现多任务操作。

8.1 执行外部命令

当需要执行一个外部 shell 命令或者可执行文件时，可以使用 child_process 模块，可以像下面这样导入该模块：

```
var child_process = require('child_require');
```

然后，可以像下面这样使用该模块中的 exec 函数：

```
var exec = child_process.exec;
exec(command, callback);
```

exec 函数的第一个参数是一个字符串，它表示你准备执行的 shell 命令。第二个参数是一个回调函数，它会在命令结束或发生错误时被 exec 函数调用，回调函数应该有三个参数：error、stdout 和 stderr。这里有一个例子：

```
exec('ls', function(err, stdout, stderr) {
  // ...
});
```

如果出现错误，第一个参数是 Error 类的一个实例，如果第一个参数不包含错误，第二个参数 stdout 将会包含命令的输出信息。最后一个参数包含命令的错误输出信息。

程序清单 8-1 展示了一个较为复杂的执行外部命令的示例。

程序清单 8-1：执行外部命令(chapter8/01_external_command.js)

```
// 导入 child_process 模块中的 exec 函数
var exec = require('child_process').exec;
// 执行命令 "cat *.js | wc -l"
exec('cat *.js | wc -l', function(err, stdout, stderr)  {
   // the command exited or the launching failed
   if (err) {
      // we had an error launching the process
      console.log('child process exited with error code', err.code);
      return;
   }
   console.log(stdout);
});
```

在第 4 行中将字符串 cat *.js | wc -l 作为第一个参数传递给 exec 函数，也可以使用它执行其他命令，就跟你在 shell 命令行中输入命令一样。

然后传递一个回调函数作为 exec 函数的第二个参数，当发生错误或者子进程终止后就会调用该回调函数。

还可以向 exec 函数传递一个包含若干配置选项的可选参数了，该参数应放在回调函数参数之前，如下所示：

```
var exec = require('child_process').exec;
var options = {
   timeout: 10000,
   killSignal: 'SIGKILL'
};
exec('cat *.js | wc -l', options, function(err, stdout, stderr) {
   // ...
});
```

可用的选项包括以下这些：

- cwd——当前工作目录。如果想强制进入当前工作目录，就应使用该选项。
- encoding——子进程输出所使用的编码格式，默认是 utf8，表明采用 UTF-8 编码格式，如果希望进程的输出不采用 UTF-8 编码格式，则应使用该选项。Node 支持以下编码格式：
  - ascii
  - utf8
  - ucs2
  - base64

  如果想了解更多有关 Node 支持的编码格式信息，请参见第 4 章“使用缓冲区操作、编码和解码二进制数据”。
- timeout——命令执行的超时时间，单位为毫秒，默认值是 0，即无限等待下去，直到子进程终止。
- maxbuffer——指定 stdout 流和 stderr 流的最大容量，单位为字节，如果超过指定值，子进程就会终止。默认值是 200*1024 字节。
- killSignal——如果超时或者超过输出缓存容量，该信号就会被发送到子进程。

  killSignal 的默认值是 SIGTERM，它向进程发送一个终止信号，这通常是终止进程的一种有序方法。当使用 SIGTERM 信号时，进程可以处理和重载默认行为。如果目标进程需要，还可以将其他信号(例如 SIGUSR1)与 SIGTERM 一并发送给它。还可以向目标进程发送 SIGKILL 信号，该信号由操作系统处理，并且不会对被触发的程序做任何清理就立即强制终止进程。

  如果想进一步控制进程的终止，可以用 child_process.spawn 函数代替发送终止信号，该函数在后面会有介绍。
- evn——传递给子进程的环境变量，默认值是 null，这意味着子进程会继承所有父进程的环境变量，这些环境变量在子进程创建之前就已经被定义好了。

注意：通过 killSignal 选项可以以字符串的形式向目标进程发送信号，信号在 Node 中是用字符串标识的，此处列出了 UNIX 中的所有信号以及它们的默认行为：

名称	默认行为	描述
SIGUP	终止进程	终端挂起
SIGINT	终止进程	中断程序
SUGQUIT	创建核心映像	退出程序
SIGILL	创建核心映像	非法指令
SIGTRAP	创建核心映像	跟踪捕获
SIGABRT	创建核心映像	中止程序

(续表)

名称	默认行为	描述
SIGEMT	创建核心映像	模拟执行指令
SIGFPE	创建核心映像	浮点异常
SIGKILL	终止进程	终止程序
SIGBUS	创建核心映像	总线错误
SIGSEGV	创建核心映像	段违规
SIGSYS	创建核心映像	调用了不存在的系统调用
SUGPIPE	终止进程	软件终止信号
SIGALRM	终止进程	实时定时器到时
SIGTERM	终止进程	软件终止信号
SIGURG	丢弃信号	套接字上出现紧急状况
SIGSTOP	停止进程	停止(无法捕获或忽略)
SIGTSTP	停止进程	键盘产生的停止信号
SIGCONT	忽略该信号	停止之后继续
SIGCHLD	忽略该信号	子进程状态发生改变
SIGTTIN	停止进程	后台进程试图从控制终端读取数据
SIGTTOU	停止进程	后台进程试图向控制终端写入数据
SIGIO	丢弃信号	某个描述符上的 I/O 可行
SIGXCPU	终止进程	超过 CPU 时间限制
SIGXFSZ	终止进程	超过文件大小限制
SIGVTALRM	终止进程	虚拟定时器到时
SIGPROF	终止进程	统计分布图定时器到时
SIGWINCH	忽略该信号	窗口大小发生改变
SIGINFO	忽略该信号	键盘产生的状态请求
SIGUSR1	终止进程	用户自定义信号 1
SIGUSR2	终止进程	用户自定义信号 2

也许你想给子进程提供一组环境变量，作为父进程环境变量的扩展，如果直接修改 process.env 对象，就会改变 Node 进程中每个模块的环境变量，这样做会招致很多不便。可以创建一个新对象作为 process.env 对象的增强复制版本代替上述做法，如程序清单 8-2 所示。

程序清单 8-2：使用增强版的环境变量执行命令

```
var env = process.env,
    varName,
```

```
    envCopy = {},
    exec = require('child_process').exec;
// 将 process.env 对象的内容复制到 envCopy 中
for (varName in env) {
  envCopy[varName] = env[varName];
}
// 分配一些自定义变量
envCopy['CUSTOM ENV VAR'] = 'some value';
envCopy['CUSTOM ENV VAR 2'] = 'some other value';
// 结合 process_env 对象和自定义变量执行命令
exec('ls -la', { env: envCopy }, function(err, stdout, stderr) {
    if (err) { throw err; }
    console.log('stdout:', stdout);
    console.log('stderr:', stderr);
});
```

在上面的例子中，创建了一个保存 envCopy 变量的对象，该对象在初始化时复制了 process_env 对象的内容，然后就可以添加环境变量或者替换掉你希望修改的环境变量。最后在执行命令时将环境变量对象 envCopy 传递给 exec 函数。

请记住，环境变量是通过操作系统在进程之间传递的，因此所有环境变量值都是以字符串的形式传入子进程的。例如，如果父进程将数字 123 作为环境变量进行传递，那么子进程接收的则是字符串形式的 123。

可以在同一文件夹中创建两个 Node 脚本作为示例，一个命名为 parent.js，另一个命名为 child.js，由 parent.js 启动 child.js。为此，必须将这两个脚本放在合适的位置。

程序清单 8-3 是一个可快速完成的例子，用来展示上述过程。

程序清单 8-3：父进程设置环境变量(chapter8/03_environment_number_parent.js)

```
var exec = require('child_process').exec;
exec('node child.js', {env: {number: 123}}, function(err, stdout, stderr) {
    if (err) { throw err; }
    console.log('stdout:\n', stdout);
    console.log('stderr:\n', stderr);
})
```

将上述代码存为文件 parent.js，下面是子进程的源代码，应该将其存入文件 child.js(见程序清单 8-4)。

程序清单 8-4：子进程解析环境变量(chapter8/03_environment_number_child.js)

```
var number = process.env.number;
console.log(typeof(number)); // → "string"
number = parseInt(number, 10);
console.log(typeof(number)); // → "number
```

将上述代码存进文件 child.js 后，可以进入源代码所在的文件夹，使用命令行运行它：

```
$ node parent.js
```

此时会输出以下信息：

```
stdout:
 string
number

stderr:
```

在上面的信息中可以看到，尽管父进程传递了一个数字型的环境变量，但是子进程接收的却是一个字符串(正如你在输出信息的第 2 行中看到的那样)。在第 3 行中，将字符串解析成一个数字。

8.2 生成子进程

如你所见，可以使用 child_process.exec()函数启动外部进程，并在进程结束时调用回调函数。该方法十分简单，不过也有一些缺点：

- 除了命令行参数和环境变量之外，exec()函数不允许与子进程通信。
- 子进程的输出是被缓存的，结果是无法对其进行流操作，它可能会耗尽内存。

幸运的是，Node 的 child_process 模块允许对子进程的启动、终止以及与其进行交互进行更精细的控制。你也许会在程序(也就是父进程)中新建一个进程(也就是子进程)，一旦启动了一个新的子进程，Node 就创建了一个双向通信通道，两个进程可以利用这条通道互相收发字符串形式的数据，父进程还可以对子进程施加一些控制、向其发送信号或者强制终止子进程。

8.2.1 创建子进程

可以基于 child_process.spawn 函数创建子进程，如程序清单 8-5 所示：

程序清单 8-5：创建子进程(chapter8/05_spawning_child.js)

```
// 导入 child_process 模块中定义的 spawn 函数
var spawn = require('child_process').spawn;

// 用"tail -f /var/log/system.log"命令启动子进程
var child = spawn('tail', ['-f', '/var/log/system.log']);
```

在上面的代码中，创建了一个子进程，通过传递-f and /var/log/system.log 参数执行 tail 命令，tail 命令监视文件/var/log/system.log(如果该文件存在的话)，并且输出的新数据都被附加到 stdout 流中，spawn 函数会返回一个 ChildProcess 对象，该对象是一个句柄，它封装了对实际进程的访问，此处将这个对象的描述符赋值给 child 变量。

8.2.2　监听子进程的输出数据

任何一个子进程句柄都有一个属性 stdout，它以流的形式表示子进程的标准输出信息，然后可以在这个流上绑定事件，就如同说“对于从子进程输出中获取的每个数据块，都要调用回调函数”。下面是一个例子：

```
// 将子进程的输出打印至控制台
child.stdout.on('data', function(data) {
    console.log('tail output: ' + data);
});
```

每当子进程将数据输出到其标准输出时，父进程就会得到通知并将其打印至控制台。

除了标准输出之外，进程还有一个默认输出流：*standard error* 流，进程通常利用该流输出错误消息。

在上面的例子中，如果文件/var/log/system.log 不存在，tail 进程就会输出这样的消息：tail: /var/log/system.log: No such file or directory。父进程通过监听 stderr 流，就会在出现错误时得到通知。

父进程还可以从错误流中监听数据，如下所示：

```
child.stderr.on('data', function(data) {
    console.log('tail error output:', data);
});
```

与 stdout 一样，stderr 也是一个可读流，在本例中，每当子进程往标准错误流中输出数据时，父进程都会得到通知并打印出这些数据。

8.2.3　向子进程发送数据

除了从子进程的输出流中获取数据之外，父进程也向子进程的标准输入流中写入数据，这相当于向子进程发送数据，标准输入流是用 childProcess.stdin 属性表示的。

子进程也可以使用 process.stdin 流来监听数据，但是注意，首先得恢复流，因为在默认情况下它处于暂停状态。

在程序清单 8-6 中，将使用以下部分构建一个例子：

- +1 app：一个简单的应用程序，在其标准输入流上获取整数，并将整数加 1，然后将加 1 后的整数输出到标准输出流中。
- 1 app：一个简单的应用程序，在其标准输入流上获取整数，并将整数加 1，然后将加 1 后的整数输出到标准输出流中。

根据程序清单 8-6 中的代码创建 plus_one.js 文件。

程序清单 8-6：+1 app 示例应用程序(chapter8/06_plus_one.js)

```
// 解除 stdin 流的暂停状态
process.stdin.resume();
process.stdin.on('data', function(data)  {
```

```
var number;
try {
  // 将输入数据解析为一个数
  number = parseInt(data.toString(), 10);
  // 加1
  number += 1;

  // 输出加1后的数
  process.stdout.write(number + "\n");
} catch(err) {
  process.stderr.write(err.message + "\n");
}
});
```

在上面的代码中，你在等待来自 stdin 流的数据，每当有数据可获取时都假定它是一个整数，并将数据解析到一个整型变量 number 中，然后加 1，并将加 1 后的结果输出到 stdout 流。

可以调用如下命令运行这个简单的程序：

```
$ node plus_one.js
```

运行之后，应用程序就等待输入，如果输入一个整数然后按回车键，就会在屏幕上看到一个加 1 后返回的整数。

可以输入 Ctrl+C 退出应用程序。

1. 测试客户端

现在要创建一个 Node 进程来使用前面的“+1 应用程序”提供的计算服务。

首先创建文件 plus_one_test.js，如程序清单 8-7 所示：

程序清单 8-7：测试+1 app 使用程序(chapter8/07_plus_one_test.js)

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

// 使用 node 进程创建一个子进程执行 plus_one 使用程序
var child = spawn('node', ['plus_one.js']);

// 每隔 1 秒钟调用一次该函数
setInterval(function()  {

    // 产生一个小于 10 的随机数
    var number = Math.floor(Math.random() * 10000);

    // 将该随机数发送到子进程
    child.stdin.write(number + "\n");

    // 获得子进程的响应并打印出来
```

```
    child.stdout.once('data', function(data) {
        console.log('child replied to ' + number + ' with: ' + data);
    });
}, 1000);
child.stderr.on('data', function(data) {
    process.stdout.write(data);
});
```

在上面的代码中，第 1 行至第 4 行启动了一个运行“+1 app”的子进程。

然后使用 SetInterval 函数每隔一秒执行一次以下操作：

- 创建一个小于 10 000 的随机自然数。
- 将产生的随机数作为一个字符串发送给子进程。
- 等待子进程返回一个字符串

因为针对每个数你只希望从子进程中获取一次数据，所以使用的是 child.stdout.once 函数而不是 child.stdout.on 函数，如果使用的是后者，那么随着时间的推移，将要注册多个回调函数，这样的话每当从子进程的 stdout 上获得数据时，所有回调函数都会被调用，这样会导致同一数据被输出多次，这显然是错误的。

8.2.4　当子进程退出时获得通知

当子进程退出时，会在父进程上触发一个事件，程序清单 8-8 展示了如何监听该事件：

程序清单 8-8：监听子进程退出事件(chapter8/09_listen_child_exit.js)

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

// Spawn the child with a "ls -la" command
var child = spawn('ls', ['-la']);

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

// 当子进程退出时:
child.on('exit', function(code) {
    console.log('child process terminated with code ' + code);
});
```

最后加粗显示的代码表示开始监听子进程的 exit 事件，当该事件出现时，就将子进程终止的相关信息打印至控制台。子进程的退出码会被传递给回调函数，作为其第一个参数。一些程序以非 0 的退出码表示发生错误。例如，如果尝试执行命令 ls –la filename.txt，而在当前工作目录中并不存在文件 filename.txt，就会得到退出码 1，如果试着运行程序清单 8-9 中的例子，将会得到一样的结果。

程序清单 8-9：监听子进程退出事件(chapter8/10_child_exit_code.js)

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

// 以命令"ls does_not_exist.txt"创建子进程
var child = spawn('ls', ['does_not_exist.txt']);

// 当子进程退出时：
child.on('exit', function(code) {
   console.log('child process terminated with code ' + code);
});
```

在本例中，exit 事件触发了一个回调函数，并将退出码传递给它，作为其第一个参数，如果子进程是被一个信号终止而不是正常退出的话，那么相应的信号会也会被传递给这个回调函数，作为其第二个参数，如程序清单 8-10 所示。

程序清单 8-10：获取子进程退出信号(chapter8/11_child_exit_signal.js)

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

// 生成子进程运行"sleep 10"命令
var child = spawn('sleep', ['10']);

setTimeout(function() {
  child.kill();
}, 1000);

child.on('exit', function(code, signal) {
   if (code) {
      console.log('child process terminated with code ' + code);
   } else if (signal) {
      console.log('child process terminated because of signal ' + signal);
   }
});
```

在上面的代码中，启动了一个子进程来执行一条为时 10 秒的休眠命令，但还没有到 10 秒，就又向子进程发送了一个 SIGKILL 信号，这将会打印出如下所示的输出信息：

```
child process terminated because of signal SIGTERM
```

8.3 向进程发送信号并终止进程

在本节中，你将学习如何使用信号来控制子进程，信号是父进程和子进程进行通信的一种简单方式，甚至可以用它来终止子进程。

不同的信号代码具有不同的含义，而且信号的类型多样，最常见的一些类型是用来终止进程的。如果进程收到一个它不知如何处理的信号，它就会终止。一些信号可以由子进

程处理，而另一些信号却只能由操作系统处理。

一般而言，可以使用 child.kill 方法向子进程发送一个信号，默认发送的是 SIGTERM 信号，如下所示：

```
var spawn = require('child_process').spawn;
var child = spawn('sleep', ['10']);
setTimeout(function() {
  child.kill();
}, 1000);
```

还可以将一个标识信号类型的字符串传递给 child.kill 方法，作为其唯一的参数，以此发送某个特定的信号，如下所示：

```
child.kill('SIGUSR2');
```

注意，尽管方法名是 kill，但是发送的信号却不一定会终止进程。如果由子进程来处理该信号的话，那么该信号的默认行为就被重写。在 Node 中，子进程可以定义一个信号处理程序来重写信号，如下所示：

```
process.on('SIGUSR2', function() {
   console.log('Got a SIGUSR2 signal');
});
```

现在为 SIGUSR2 定义了一个信号处理程序，进程在收到该信号后不会被终止，取而代之的是输出“Got a SIGUSR2 signal”。通过使用这种机制，可以设计一种简单的方法来和子进程通信，甚至控制子进程，这种方法虽然不像标准输入流那样功能丰富，但是却简单得多。

SIGKILL 和 SIGSTOP 是由操作系统处理的特殊信号，并且进程不能重写其默认行为，即使已经为这两个信号定义了处理程序，它们也会终止进程。

8.4　本章小结

在本章中，使用 child_process.exec 函数执行了外部命令，该函数允许你可以不使用命令行参数，而是通过定义环境变量的方式将参数传递给子进程。

你还学习了使用 child_process.spawn 方法，通过创建子进程来执行外部命令，这样可以使用输入流和输出流与子进程进行交互，另外还允许你使用信号和子进程交互甚至终止子进程。

第 9 章

读写数据流

本章提要：

- 流接口简介
- 从可读流获取数据
- 控制可读流的流动
- 向可写流写入数据
- 解决慢速客户端问题
- 使用管道将两种流混合在一起

Node 中有一个抽象概念：流。更为特别的是两个非常有用的抽象概念：可读流和可写流，它们是通过几个 Node 对象来实现的，代表数据的向内流动(可读流)与向外流动(可写流)。流的一个例子是 TCP 套接字，可以在其上进行读写操作，流的另一个例子是文件，可以对其按顺序进行追加和读取。你已经和它们中的一些成员见过面了，但本章将以一种更为正式和抽象的方式介绍它们，为一些更具体的例子做好准备。

注意：流是由几个 Node 对象实现的抽象概念，创建或者获取流的方式取决于使用流的类型。例如，可以基于文件显式地创建一个可读流或者可写流。但是，服务器端的 TCP 套接字流只有在客户端连接时才能为你所用。

除了可读流或可写流的特性外，一个对象还可以具有其他一些特殊的属性或行为。例如，文件可读流还包含“路径”属性，而这一属性在其他类型的流中也许并不存在。

9.1 使用可读流

可读流就好像是一个数据阀门。在创建或者获取一个可读流之后——创建或者获取可读流的方法取决于流的类型——可以对其进行控制。可以通过暂停和恢复来控制数据流动，可以在数据可用时得到通知，并且在流终止时你也会得到通知并能够关闭它。

9.1.1 等待数据

流能够以块为单位发送数据。通过监听“data”事件，在流每次提交一块数据时，你都会得到通知。可以以缓冲区或字符串的形式获取数据，这取决于流的编码设置。

通过绑定到“data”事件上的事件处理程序可以传送数据。根据流的编码格式，也许会用字节缓冲区以原始形式传送数据，或者，如果用 stream.setEncoding()函数定义了流的编码格式，数据就会以某种编码格式的字符串形式进行传送，如下面两个例子所示：

```
var readable stream1 = ...
readable stream1.on('data', function(data) {
  // 数据是缓冲区;
  console.log('got this data:', data);
});

var readable stream2 = ...
readable stream2.setEncoding('utf8');
readable stream2.on('data', function(data) {
  // 数据是 utf-8 编码格式的字符串;
  console.log('got this data:', data);
});
```

在第一个例子中，因为没有指定编码格式，所以数据是作为缓冲区传送的。在第二个例子中，数据是作为 UTF-8 编码格式的字符串传送的。

注意：因为 UTF-8 字符可能是多字节的，所以当你希望获取 UTF-8 字符时，也许需要在两个独立的“data”事件发生之后才能获得一个字符，当你将编码格式设置为 utf8 时，流仅会在一个字符完整的情况下传送它。

9.1.2 暂停与恢复流

可读流就像一个阀门，可以通过暂停流来阻止数据流动，如下所示：

```
stream.pause();
```

执行上面的代码之后，就再也不会接收到“data”事件了。正如读者将在后面看到的那样，这一特性会帮助你规避隐含缓冲区。

注意：暂停流在不同的情况下会被解释成不同的行为。例如，如果是一个文件流，Node 就会停止从该文件中读取数据。如果流是一个 TCP 套接字，Node 就不再会读取新的数据包，这会终止从其他终端来的数据包流。其他对象对 pause 的实现各不相同。

当想恢复被暂停的流时，只需要调用 stream.resume()方法，就会重新打开阀门，数据也就再次被传送出来——直到再次暂停流或者流终止为止。

9.1.3　了解流何时终止

流会终止。例如，如果用一个可读流来传送文件的内容，当到达文件结束位置时，流会发射“end”事件。类似地，如果用一个 HTTP 流传送请求主体，当请求结束时，流也会发射“end”事件。你监听“end”事件，如下所示：

```
var readable stream = ...
readable stream.on('end', function() {
  console.log('the stream has ended');
});
```

可读流终止后，你就再也接收不到“data”事件了。

9.2　使用可写流

可写流是一个抽象概念，可以向其发送数据。它可能是一个文件，或者是一个 TCP 网络连接，甚至可能是一个对象，用来输出被转换的数据，例如，压缩文件。在本节中将学习向可写流写入数据以及清空事件。

9.2.1　将数据写入流

通过向可写流传递缓冲区或字符串，可以将字符串写入其中，如下所示：

```
var writable_stream = ...;
writable_stream.write('this is an UTF-8 string');
```

如果将一个字符串传递给 write 函数，作为其第一个参数，就可以将字符串的编码格式也传递给该函数，作为其第二个参数。否则，流会假定字符串采用的是 UTF-8 编码。

下面是一个向 write 函数传入另一种编码格式字符串的示例：

```
var writable_stream = ...;
writable_stream.write('7e3e4acde5ad240a8ef5e731e644fbd1', 'base64');
```

如前所述，也可以通过向 write 函数传递一个缓冲区来写入数据：

```
var writable_stream = ...;
var buffer = new Buffer('this is a buffer with some string');
writable_stream.write(buffer);
```

一旦向流写入数据，Node 可以立即将数据传递到核缓冲区中，如果此时没有核缓冲区，Node 就会将数据存储到一个队列中，这个队列位于内存中。可以通过观察调用 writable_stream.write()函数的返回值确定发生了何种事件。write 命令将返回一个布尔值，如果数据被传递到核缓冲区中，该值为 true，如果数据被存入队列中，则该值为 false。在本章的后面将看到该信息有何帮助。

9.2.2 等待流被清空

因为 Node 不会在 I/O 操作上产生阻塞，所以它也不会在读或写命令上产生阻塞。正如已经看到的那样，在调用写命令时你会知道缓冲区是否被立即刷新，如果没有被刷新，将其存储在进程内存中。

稍后当流成功刷新挂起的缓冲区时，就会发射 drain 事件，可以监听该事件：

```
var writable stream = ...;
writable stream.on('drain', function() {
  console.log('drain emitted');
});
```

在本章的后面，你将看到具有暂停和恢复能力的清空通知如何限制 Node 进程的内存增长。

9.3 考虑几个流的例子

到目前为止，已经介绍了抽象的流接口。下面来看看几个 Node 实现流接口的例子。

9.3.1 创建文件系统流

可以用文件路径创建一个可读流，如下所示：

```
var fs = require('fs');
var rs = fs.createReadStream('/path/to/file');
...
```

可以向 fs.createReadableStream()函数传递第二个参数，这个参数具有几个选项，在这个参数中，可以指定文件的起始和结束位置、编码格式、标志位以及缓冲区大小。options 参数如下所示：

- encoding——“data”事件发送的字符串的编码格式，如果想使用未经处理的缓冲区，该值为 null。
- fd——如果已经有了一个打开文件描述符，则可以将其传入该选项，流会假定它的默认值为 null。

- bufferSize——要被读取的每个文件块的大小，单位为字节。默认是 64KB。
- start——文件中第一个被读取的字节位置，用来读取一定范围内的数据，而不是读取全部文件。
- end——文件中最后一个被读取的字节位置，用来读取一定范围内的数据，而不是读取全部文件。

注意：如果已经有了一个打开的文件，可以用 fd 选项从文件中创建可读流：

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

var path = '/path/to/my/file';
fs.open(path, 'r', function(err, fd) {
  fs.createReadStream(null, {fd: fd, encoding: 'utf8'});
  fs.on('data', console.log);
});
```

此外，如果只是想读取文件中的一部分，可以使用起始和结束选项。例如，如果从文件的第 10 个字节开始，一直读取到文件结束：

```
var fs = require('fs');
var path = '/path/to/my/file'
var readStream = fs.createReadStream(path, {start: 10});
...
```

或者，让文件流读取到第 20 个字节为止：

```
var fs = require('fs');
var path = '/path/to/my/file'
var readStream = fs.createReadStream(path, {end: 20});
...
```

还可以创建一个文件可写流：

```
var fs = require('fs');
var rs = fs.createWriteStream('/path/to/file', options);
```

fs.createWriteStream()函数也可以接受 option 对象作为其第二个参数，option 对象的默认值如下：

```
{ flags: 'w',
  encoding: null,
  mode: 0666 }
```

- flags 选项包含用于打开文件的标志位，所以所有可被 fs.open()函数接受的标志位均可用于此处。
- 如果要创建文件的话，mode 选项指定了要打开文件的权限。

还可以用 encoding 选项强制指定编码格式。例如，可以创建一个采用 UTF-8 编码格式的可写流，如下所示：

```
var fs = require('fs');
var rs = fs.createWriteStream('/path/to/file', { encoding: 'utf8' });
```

9.3.2 理解网络流

Node 中网络 API 上有几种类型的流。例如，客户端的 TCP 连接既是可写流，又是可读流，服务器连接也是如此。

HTTP 的请求对象是一个可读流。HTTP 响应对象是一个可写流。

在本章后面将介绍每种类型的流。现在只要记住这些流都有一个通用的接口，也就是说可以交替使用这些对象，还可以一次性抽象出一些问题并解决这些问题。慢客户端问题(在下一节中将介绍)就是问题之一。

9.4 避免慢客户端问题以及挽救服务器

当有进程读取数据，并将数据(或者数据的变换形式)发送给另一个进程时，一般就会出现慢客户端问题。

9.4.1 理解慢客户端问题

Node 在进行 I/O 操作时不会产生阻塞，也就是说 Node 不会阻塞数据的读写操作——如果用 write 无法将数据写入核缓冲区，就会缓存数据。想象一下这样的场景：将数据传入一个可写流中(比如到浏览器的 TCP 连接)，而数据源是一个可读流(比如文件可读流)。

```
require('http').createServer(function(req, res) {
  var rs = fs.createReadStream('/path/to/big/file');

  rs.on('data', function(data) {
    res.write(data);
  });

  rs.on('end', function() {
    res.end();
  });
}).listen(8080);
```

如果是本地文件，可读流应该是快速的。然而，如果到客户端的网络连接是慢速的，可写流也就是慢速的。可读流会快速产生“data”事件，同时数据将被发送到可写流。然而，Node 最终不得不缓存数据，因为核缓冲区将被填满。

实际上，对于每个 write 命令而言，文件都会在内存中被缓冲。随着请求的成倍增长，要不了一会就会产生内存增长问题。一般而言，如果既有数据生产者，又有数据消费者，并且数据生产者比数据消费者要快，就要缓存数据。也即要暂停数据生产者，直到数据消

费者赶上来为止。

9.4.2　避免慢客户端问题

在大多数情况下，可以通过暂停数据生产者来避免填满具有未刷新缓冲区的内存——可读流——以便让消费者的数据——可写流——不会被传入核缓冲区中。幸好，Node 的可读流有一些可以满足需求的功能：可以暂停和恢复可读流。这里有一个控制数据流动的例子，如下所示：

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

  var rs = fs.createReadStream('/path/to/big/file');

  rs.on('data', function(data) {
    if (!res.write(data)) {
      rs.pause();
    }
  });

  res.on('drain', function() {
    rs.resume();
  });

  rs.on('end', function() {
    res.end();
  });

}).listen(8080);
```

在上面的例子中，创建了一个 HTTP 服务器，它在端口 8080 上监听，并将文件 /path/to/big/file 的内容提交给所有 HTTP 请求。

如果 write 命令无法将可读流传入核缓冲区中，第 7 行就会暂停可读流。当可写流完成接收数据时，第 11 行就会恢复可读流。

9.4.3　应用 stream.pipe()避免慢客户端问题与使用 pipe()集成可读流和可写流

暂停可读流直到可写流赶上并在之后恢复可读流的过程是一种循环模式，Node 使用 stream.pipe()命令实现这种模式。

stream.pipe()命令是可读流接口的一部分——由传输源调用——并接受目标可写流作为其第一个参数。

如果在上面的例子中使用 pipe 命令，示例代码如下所示：

```
require('http').createServer(function(req, res) {
  var rs = fs.createReadStream('/path/to/big/file');
  rs.pipe(res);
}).listen(8080);
```

默认情况下，end()会在可读流结束时，在可写流上被调用。为了避免这种情况，可以将 end: false 传入 pipe()函数，作为其第二个参数，这个参数是 option 对象，如下所示：

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

  var rs = fs.createReadStream('/path/to/big/file');

  rs.pipe(res, { end: false });

  rs.on('end', function() {
    res.write("And that's all, folks!");
    res.end();
  });

}).listen(8080);
```

在上面的代码中，在文件结束时写入字符串"And that's all, folks!"，然后结束请求，而不是让 stream.pipe()方法进行这项工作。

9.5 本章小结

流是一种极好的抽象，允许利用 Node 的架构来轻松实现任意对象的流，这些对象实现了流 API。

流接口可以让你控制数据流动，以缓解慢客户端问题，即可以停止和恢复可读流。通过使用 stream.pipe()方法可以实现该过程的自动化，这对于任何可读流实例都适用。

第10章

构建 TCP 服务器

本章提要：

- 创建 TCP 服务器
- 关闭服务器端的 TCP 连接
- 处理网络错误
- 通过管道从 TCP 连接读写数据
- 构建一个 TCP 聊天服务器

传输控制协议(Transmission Control Protocol, TCP)是 Internet 的基础协议之一，它位于网际协议(Internet Protocol, IP)之上，为应用层提供了一种传输机制。例如，HTTP 工作在 TCP 之上，其他很多面向连接的应用(诸如 iRC、SMTP 和 IMAP)也工作在 TCP 之上。

Node 以 http.Server 中伪类的形式一流地实现了 HTTP 服务器，该伪类继承自.net Server 中的 TCP 服务器伪类，这意味着本章中叙述的所有内容同样能应用到 Node HTTP 服务器中。

10.1 创建 TCP 服务器

可以使用 net 模块创建 TCP 服务器，如下所示：

```
require('net').createServer(function(socket) {
  // 新连接

  socket.on('data', function(data) {
    // 获取数据
  });
```

```
  socket.on('end', function(data) {
    // 关闭连接
  });

  socket.write('Some string');

}).listen(4001);
```

在第 1 行用到了 net 包中的 createServer()方法，并在第 14 行将该方法绑定到 4001 端口。可以向 createServer 传递一个回调函数，每当有“connection”事件发生时都会调用该回调函数。在回调函数内部会提交一个套接字对象，可以应用该对象从客户端获取数据或者向客户端发送数据。

因为服务器对象也是一个事件发射器，所以也可以在其生命周期内监听事件，.net Server 发射以下事件：

- “listening”——当服务器在指定的端口和地址监听时。
- “connection”——当有新的连接创建时。回调函数会接收对应的套接字对象，可以通过向 net.createServer()函数传递一个回调函数绑定到该事件上，就像你在上一个例子中所做的那样。
- “close”——当服务器被关闭时，即不再绑定到那个端口。
- “error”——当在服务器层面出现错误时。例如当你尝试绑定一个已被占用的端口或者没有权限绑定的端口时，就会发生错误。

程序清单 10-1 是一个输出服务器生命周期的例子。

程序清单 10-1：TCP 服务器生命周期示例

```
var server = require('net').createServer();

var port = 4001;

server.on('listening', function() {
  console.log('Server is listening on port', port);
});

server.on('connection', function(socket) {
  console.log('Server has a new connection');
  socket.end();
  server.close();
});

server.on('close', function() {
  console.log('Server is now closed');
});

server.on('error', function(err) {
  console.log('Error occurred:', err.message);
```

```
});

server.listen(port);
```

启动这个例子，当服务器运行时可以应用 telnet 或者 nc 来连接它，如下所示：

```
$  telnet localhost 4001
```

或者

```
$  nc localhost 4001
```

一旦连接成功，服务器就会断开你的连接并关闭(因为是你在第 11 行告诉它这样做的)。应该在服务器控制台看到如下输出信息：

```
Server is listening on port 4001
Server has a new connection
Server is now closed
```

10.1.1　应用套接字对象

当获取“connection”事件时也获得了一个套接字对象作为回调函数的第一个参数，套接字对象既是可读流也是可写流，这意味着当它获得数据包时会发射“data”事件，当连接关闭时会发射“end”事件。

因为套接字对象也是一个可写流，这意味着可以应用 socket.write()函数向套接字写入缓冲区或者字符串。可以通过调用 socket.end()函数告诉套接字当所有数据都被写入之后应该终止连接。程序清单 10-2 是一个 TCP 服务器简单地重复写入内容的例子：

程序清单 10-2：简单地重复 TCP 服务器

```
var server = require('net').createServer(function(socket) {
  console.log('new connection');

  socket.setEncoding('utf8');

  socket.write("Hello! You can start typing. Type 'quit' to exit.\n");

  socket.on('data', function(data) {
    console.log('got:', data.toString())
    if (data.trim().toLowerCase() === 'quit') {
      socket.write('Bye bye!');
      return socket.end();
    }
    socket.write(data);
  });

  socket.on('end', function() {
    console.log('Client connection ended');
```

```
  });

}).listen(4001);
```

可以像以前那样，再次启动服务器并用 telnet 或者 nc 进行连接，然后。可以输入 quit<回车>来终止连接。

因为 socket 对象是一个可读流，可以通过调用 socket.pause()和 socket.resume()函数控制数据流，甚至用管道将套接字写入一个可写流中，如程序清单 10-3 所示。

程序清单 10-3：从套接字输出

```
var ws = require('fs').createWriteStream('mysocketdump.txt');

require('net').createServer(function(socket) {
  socket.pipe(ws);
}).listen(4001);
```

和前面一样，可以用 telnet 和 nc 来测试程序清单 10-3，如下所示：

```
$  nc localhost 4001
```

你想输入多少就输入多少，至少要按一次<回车键>将数据输入。然后可以打开文件 mysocketdump.txt 找到你输入的文本。

可以反其道而行之，将一个可读流输入到套接字中，如程序清单 10-4 所示。为了达到这一目的，必须在本地文件夹中创建一个名为 hello.txt 的文件，在其中填入一些随机文本，然后运行程序清单 10-4 中的例子。

程序清单 10-4：向套接字输入

```
require('net').createServer(function(socket) {
  var rs = require('fs').createReadStream('hello.txt');
  rs.pipe(socket);
}).listen(4001);
```

如果尝试连接服务器，会看到你插入文件 hello.txt 中的文本。同样连接也会被立即关闭，这是因为正如第 9 章“读写数据流”中所解释的那样，在默认情况下，pipe 操作会在源终端终止之后终止目的终端。如果希望连接一直保持为运行状态，应该将{end : false}传递给 pipe()命令作为其第二个参数。

10.1.2　理解空闲套接字

默认情况下当一条连接在两个对等终端之间建立起来之后，就会一直保持为活动状态，直到其中一个终端关闭它或者其下层的链路丢失为止。然而，在 Node 中因为非活动性可以为 TCP 连接设置超时时间。当一段时间没有发出或者收到流量信息，就可以自动关闭连接。可以通过在连接上调用 setTimeout(millisecond)函数激活和定义超时时间，还可以监听套接字对象上的 timeout 事件。

```
var timeout = 60000; // 1 分钟
socket.setTimeout(timeout);
socket.on('timeout', function() {
  socket.write('idle timeout, disconnecting, bye!');
  socket.end();
});
```

或者也可以使用一种较为简约的方式，即传递事件监听器作为 socket.setTimeout()函数的第二个参数：

```
socket.setTimeout(60000, function() {
  socket.end('idle timeout, disconnecting, bye!');
});
```

10.1.3　设置保持运行

在 Node 中 net.Socket 实现了一种保持运行的机制避免网络或终端上出现超时，Node 通过发送带有被打开的确认(Acknowledgement, ACK)标志的空 TCP 包触发另一终端的空应答来达到这一目的，这一操作会保持两个终端之间的连接处于运行状态。

可以通过以下代码激活保持运行功能：

```
Socket.setKeepAlive(true);
```

还可以指定最后接收到的包与下一个保持运行包之间的延时时间，为此，可以设置 socket.keepAlive()函数的第二个参数，示例代码如下所示：

```
socket.setKeepAlive(true, 10000); // 10 秒
```

注意：保持运行设置和前面讨论的套接字超时设置没有关系，socket.KeepAlive()设置周期性地发送空数据包以保持连接运行，而 socket.setTimeout()则用来定义本地休眠状态的超时时间。

10.1.4　应用延时或非延时

在发送 TCP 数据包之前，内核会缓冲数据，它使用 Nagle 算法决定何时实际发送数据。在应用程序发送小块数据时，该算法被用来减少通过网络发送的包的数量。取决于应用程序，该特性会变得十分有用，但它会导致延时发送数据，这会增加应用程序的反应时间。

如果要使延时失效，在 write 命令之后强制数据立即被发送出去，可以调用如下所示的函数：

```
socket.setNoDelay(true);
```

当然，可以常常恢复该设置，如下所示：

```
socket.setNoDelay(false);
```

10.1.5　监听连接

正如已经看到那样，在服务器被创建之后，可以将其绑定到特定的端口上，如下所示：

```
var port = 4001;
var host = '0.0.0.0';
server.listen(port, host);
```

第二个参数(host)是可选的，如果忽略该参数，则服务器将会接受指向任意 IP 地址的连接：

```
server.listen(port);
```

10.1.6　关闭服务器

如下所示的方法会关闭服务器，不让其再接受新的连接，关闭服务器的函数是异步的，服务器在被关闭时会发射 close 事件：

```
var server = ...
server.close();
server.on('close', function() {
  console.log('server closed!');
});
```

10.1.7　处理错误

在处理客户端上的套接字或者服务器时，可以(也应该)通过监听 error 事件来处理错误，如下所示：

```
require('net').createServer(function(socket) {
  socket.on('error', function(error) {
    // do something
  });
});
```

如果未能捕获错误，那么 Node 会处理一个未捕获异常，并终止当前进程。

注意：可以按照如下所示的那样，选择捕获“未捕获”异常——以防止 Node 进程被终止：

```
process.on('uncaughtException', function(err) {
  // do something
});
```

但是，这样做通常不是一个好主意，因为当出现异常而你又未能对其进行适当地处理时，应用程序就会陷入一种未知状态，这种未知状态也许在后面会导致更多更危险的错误。此外，错误本身也会引起更多错误，导致难以理解错误根源。如果按照上述方法处理，可能还会泄露内存或资源(例如文件描述符)，因为错误在该被处理时却没有得到处理。

10.2　构建一个简单的 TCP 聊天服务器

现在应该准备开始构建一个基于 TCP 的聊天服务器，可以从实例化服务器、记录一些重要事件以及将服务器绑定到端口 4001 着手。

程序清单 10-5：TCP 聊天服务器——第一步

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

var server = net.createServer();

server.on('error', function(err) {
  console.log('Server error:', err.message);
});

server.on('close', function() {
  console.log('Server closed');
});

server.listen(4001);
```

10.2.1　接受连接

下一步需要接受新的客户端连接。

程序清单 10-6：TCP 聊天服务器——第二步

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

var server = net.createServer();

server.on('connection', function(socket) {
  console.log('got a new connection');
});

server.on('error', function(err) {
  console.log('Server error:', err.message);
});

server.on('close', function() {
  console.log('Server closed');
});

server.listen(4001);
```

10.2.2　从连接中读取数据

每当服务器获得新的连接时，需要通过绑定 data 事件，让其监听传入的数据。

程序清单 10-7：TCP 聊天服务器——第三步

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

var server = net.createServer();

server.on('connection', function(socket) {
  console.log('got a new connection');
  socket.on('data', function(data) {
    console.log('got data:', data);
  });
});

server.on('error', function(err) {
  console.log('Server error:', err.message);
});

server.on('close', function() {
  console.log('Server closed');
});

server.listen(4001);
```

10.2.3　聚合所有客户端

因为创建了一个聊天服务器，在其中需要向所有客户端广播用户数据，所以第一步是将所有连接存入某个位置。

程序清单 10-8：TCP 聊天服务器——第四步

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

var server = net.createServer();

var sockets = [];

server.on('connection', function(socket) {
  console.log('got a new connection');

  sockets.push(socket);

  socket.on('data', function(data) {
    console.log('got data:', data);
  });

});

server.on('error', function(err) {
  console.log('Server error:', err.message);
});
```

```
server.on('close', function() {
  console.log('Server closed');
});

server.listen(4001);
```

10.2.4　广播数据

每当一个已连接的用户输入数据，就应该将数据广播给所有其他已连接的用户，如程序清单 10-9 所示。

程序清单 10-9：TCP 聊天服务器——第五步

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

var server = net.createServer();

var sockets = [];

server.on('connection', function(socket) {
  console.log('got a new connection');

  sockets.push(socket);

  socket.on('data', function(data) {
    console.log('got data:', data);

    sockets.forEach(function(otherSocket) {
      if (otherSocket !== socket) {
        otherSocket.write(data);
      }
    });
  });

});

server.on('error', function(err) {
  console.log('Server error:', err.message);
});

server.on('close', function() {
  console.log('Server closed');
});

server.listen(4001);
```

10.2.5　删除被关闭的连接

现在至少还缺少一步——还要在连接被关闭时将其删除，如程序清单 10-10 所示。

程序清单 10-10：简单的 TCP 聊天服务器

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

var server = net.createServer();

var sockets = [];

server.on('connection', function(socket) {
  console.log('got a new connection');

  sockets.push(socket);

  socket.on('data', function(data) {
    console.log('got data:', data);

    sockets.forEach(function(otherSocket) {
      if (otherSocket !== socket) {
        otherSocket.write(data);
      }
    });
  });

  socket.on('close', function() {
    console.log('connection closed');
    var index = sockets.indexOf(socket);
    sockets.splice(index, 1);
  });
  });

server.on('error', function(err) {
  console.log('Server error:', err.message);
});

server.on('close', function() {
  console.log('Server closed');
});

server.listen(4001);
```

10.2.6 使用 TCP 聊天服务器

现在应该准备对服务器进行测试，可以将服务器代码保存为文件 chat_server.js，并用如下所示的命令启动它：

```
$ node chat_server.js
```

现在可以用 nc 或者 telnet 连接服务器：

```
$ nc localhost 4001
```

如果可以，请试着启动几个单独的客户端窗口，可以看到聊天服务器在运行中。

10.3　本章小结

TCP 服务器会在其生命周期内发射一些事件，即在被绑定到某个端口上时会发射“监听”事件，被关闭时会发射“关闭”事件，而出现错误时会发射“错误”事件。还可以监听“连接”事件，该事件会在新客户端连接时出现。连接事件会向你提交一个 socket 对象，socket 对象既是可读流又是可写流，可以用该对象来监听数据、发送数据、终止连接，甚至可以将连接数据传入另一个流中，反过来也可以将一个可读流传入连接。

socket 对象允许你使用 socket.pause()和 socket.resume()控制它的流程，还允许你对它的一些参数进行微调。例如，可以在连接空闲一段时间时关闭它、频繁发送保持活跃数据包，或者打开或关闭 Nagle 算法。

此外，还可以创建处理大量连接的 TCP 服务器，客户端使用这个服务器按照某种方式彼此之间进行通信，例如本章创建的聊天服务器示例。

第11章

构建 HTTP 服务器

本章提要：

- 创建 HTTP 服务器
- 让 HTTP 服务器在某个端口上监听
- 处理 HTTP 请求
- 观察请求头
- 监听和传送请求数据
- 用状态码和头数据进行响应
- 用主体数据进行响应
- 使用 HTTP 分块响应流式传送响应主体

HTTP 是一个用来进行传送内容和应用程序的应用层协议，它将 TCP 用作传输协议，并且是万维网进行数据通信的基础。首选的应用程序部署方案之一就是在互联网上提供 HTTP 服务，来响应 HTTP 客户端的请求。

在 Node 中可以轻松地创建 HTTP 服务器，下面是著名的“Hello World!”HTTP 服务器示例，如程序清单 11-1 所示。

程序清单 11-1：“Hello World!”HTTP 服务器

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

var server = http.createServer();

server.on('request', function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.write('Hello World!');
```

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

server.listen(4000);
```

在第 1 行获得了一个 http 模块，从中可以通过调用 http.createServer()函数创建 server 对象(第 3 行)。

然后监听 request 事件，该事件会在有新客户端连接时发生，所提供的事件回调函数有两个参数：request 对象和 respose 对象。然后可以使用 request 对象了解请求的细节，还可以用 response 对象向客户端写回数据。

在第 6 行写入了头('ContentType': 'text/plain')和 HTTP 状态 200，200 是表示请求成功的状态码。

在第 7 行用字符串“Hello World!”进行响应，并在第 8 行结束请求。

在第 11 行将服务器绑定到 TCP 端口 4000.

然后将上述脚本保存为文件 server.js 并运行，如下所示：

```
$ node server.js
```

接着将浏览器转向 http://localhost:4000，此时应该能够看到“Hello World!”字符串显示在浏览器上。

上面这个完整的示例可以被简写，如程序清单 11-2 所示。

程序清单 11-2：简写的“Hello World！”HTTP 服务器

```
require('http').createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!');
}).listen(4000);
```

在上面的代码中舍弃了存储 http 模块的中间变量(因为只需要调用一次该模块)，也舍弃了存储服务器的变量(因为只需要它在端口 4000 上监听)。此外，作为一种快捷方式，http.createServer()函数接受了一个回调函数，该回调函数会在每个请求上被调用。

上面的代码中还有最后一个快捷方式：respose.end()函数，该函数可以接受一个字符串或者缓冲区，它会在结束请求之前向这个字符串或者缓冲区写入响应。

当客户端发出一个请求时，HTTP 就会发射一个 request 事件，该事件传入 HTTP 请求和 HTTP 响应对象。HTTP 请求对象允许你查询请求的一些属性，而 HTTP 响应对象允许你创建一个 HTTP 响应，这个响应会被发送到客户端。在下面几节中，你将学习这些对象以及如何使用这些对象完成任务。

11.1　理解 http.ServerRequest 对象

在监听 request 事件时，回调函数会得到一个 http.ServerRequest 对象作为第一个参数，

这个对象包含一些属性，可以查看这些属性，这些属性包括 url、method 以及 headers：

- req.url：该属性包含一个字符串型形式的请求 URL。它不包含模式、主机名或者端口，但包括 URL 中在上述内容之后的所有剩余部分，试着运行如下所示的服务器来分析 request 属性：

```
require('http').createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end(req.url);
}).listen(4000);
```

  用浏览器打开 http://127.0.0.1:4000/abc，修改 URL 中的路径看看会发生什么。
- req.method：该属性包含在请求上用到的 HTTP 方法，例如，它有可能是 GET、POST、DELETE 或者 HEAD。
- req.headers：该属性包含一个对象，这个对象拥有请求上所有的 HTTP 头，为了分析该属性，可以运行如下所示的服务器：

```
var util = require('util');
require('http').createServer(function(req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end(util.inspect(req.headers));
}).listen(4000);
```

  将浏览器连接到 http://127.0.0.1:4000 来查看请求头，服务器应该输出如下所示的信息：

```
{ host: 'localhost:4000',
  connection: 'keep-alive',
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_2)
AppleWebKit/535.2
(KHTML, like Gecko) Chrome/15.0.874.121 Safari/535.2',
  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' }
```

  在上面的代码中用到了 util.inspect()，它是一个可以分析任意对象属性的实用函数。header 对象的键是小写的，例如，如果浏览器发送了一个请求头“Cache-Control: max-age=0”，则 req.headers 就会具有一个属性“cache-control”，该属性的值是“max-age=0”，值不会受到影响。

  当在服务器上得到 request 事件时，并不会立即获得请求体，这只是因为请求体还没有到达。但因为 request 对象是可读流，所以只要你想，就可以监听 data 事件，或者将请求体传送到一个可写流中，比如一个文件或者一个 TCP 连接：

```
var writeStream = ...

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

```
  req.on('data', function(data) {
    writeStream.write(data);
  });
}).listen(4001);
```

11.2 理解 http.ServerResponse 对象

响应对象(request 事件回调函数的第二个参数)用来响应客户端，可以用它写入响应头和响应主体。

11.2.1 写入响应头

为了写入响应头，可以使用 res.writeHead(status, headers)函数，其中 headers 是一个可选参数，它是一个包含所有想要发送的响应头属性的对象。考虑下面这个示例：

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

  res.writeHead(200, {
    'Content-Type': 'text/plain',
    'Cache-Control': 'max-age=3600' });
  res.end('Hello World!');

}).listen(4000);
```

上面的示例设置了两个响应头，一个是“Content-Type: text/plain”，另一个是“Cache-Control: max-age=3600”。

如果将前面的源代码保存为 http_server.js 并运行它：

```
$ node http_server.js
```

可以使用浏览器或者像 curl 这样的命令行 HTTP 客户端对响应头进行查询：

```
$ curl -i http://localhost:4000
HTTP/1.1 200 OK
Content-Type: text/plain
Cache-Control: max-age=3600
Connection: keep-alive
Transfer-Encoding: chunked
Hello World!
```

11.2.2 修改或设置响应头

通过下面的语句，可以修改已经设置的响应头或者设置一个新响应头：

```
res.setHeader(name, value);
```

上面的语句只有在还没有用 res.write()或者 res.end()发送响应主体时才会起作用，如果已经在响应对象上使用 res.writeHead()，上述语句也不会起作用，这是因为响应头已经被发

送出去了。

11.2.3　删除响应头

通过调用 res.removeHeader 并提供响应头名称，可以删除已设置的响应头：

```
res.removeHeader('Cache-Control');
```

再次提醒，上面的语句只会在响应头还未被发送出去时起作用。

11.2.4　写入一块响应主体

HTTP 服务器会在发送响应头之后发送响应主体，有两种写入响应主体的方法，一是写入一个字符串：

```
res.write('Hello');
```

另一种是使用已存在的缓冲区：

```
var buffer = new Buffer('Hello World');
res.write(buffer);
```

11.3　以流的形式传送 HTTP 分块响应

Node 的一个极好的特性是能够很容易地从不同的源中消耗和产生流，因为 HTTP 是 Node 中的第一类协议，所以对于 HTTP 响应而言也是如此。

HTTP 分块编码允许服务器持续向客户端发送数据，而不需要发送数据大小。除非指定了 Content-Length 响应头，否则 Node HTTP 服务器会向客户端发送如下所示的响应头：

```
Transfer-Encoding: chunked
```

根据 HTTP 规范，上面的响应头可以让客户端接受若干块数据作为响应主体，并在客户端结束响应之前发送一个具有 0 长度的结束块。这对于向 HTTP 客户端流式传送文本、音频或者视频数据非常有用。

11.3.1　传送文件

可以将任意的可读流传送到响应当中，下面是一个将文件传送到响应中的例子，首先在当前目录下有一个.mp4 文件(电影)test.mp4，然后运行服务器，如程序清单 11-3 所示。

程序清单 11-3：向 HTTP 响应传送一部电影

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

require('http').createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'video/mp4'});
  var rs = fs.createReadStream('test.mp4');
  rs.pipe(res);
}).listen(4000);
```

针对程序清单 11-3 中的每个请求，都写入了头响应状态并设置了内容类型头，以便让浏览器能够正确识别传送给它的流的类型。然后将电影文件作为一个可读流打开并将其传入响应的可写流中。

如果打开任意一个现代浏览器，并转向 http://127.0.0.1:4000，电影都会立即开始播放，即使它还没有完全被加载。

11.3.2 传送其他进程的输出

程序清单 11-4 展示了一个向客户端传送子进程输出的例子。

程序清单 11-4：向 HTTP 响应传送进程的输出

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

require('http').createServer(function(req, res) {
  var child = spawn('tail', ['-f', '/var/log/system.log']);
  child.stdout.pipe(res);
  res.on('end', function() {
    child.kill();
  });
}).listen(4000);
```

在上面的代码中，当出现一个新的请求时，就通过执行 tail -f /var/log/system.log 命令启动一个新的子进程(第 4 行)，然后将子进程的输出传入响应主体当中(第 5 行)。

当响应结束时(例如，因为浏览器窗口被关闭或者网络连接被切断)，就需要结束子进程，使得它不会被在后来被无限期地挂起。

所以程序清单 11-4——在 9 行代码中——创建了能生成进程并传输进程输出的流服务器，并根据需要结束子进程，通过这些工作向你展示了 Node 的能力。

11.4 关闭服务器

可以通过解除 HTTP 服务器与端口的绑定来终止服务器，使其不再接受新的连接，如下所示：

```
server.close();
```

如果希望它能够重新开始监听，就必须再次执行以下代码：

```
server.listen(port[, hostname]);
```

后面是 HTTP 服务器的一些简单示例，在这些例子中构建 HTTP 服务器时不需要任何第三方模块。

11.5　示例 1：构建提交静态文件的服务器

在本例中将要构建一个提交静态文件的服务器(见程序清单 11-5)，文件路径是通过一个 URL 提供的，如下所示：

```
http://localhost:4000/path/to/my/file.txt
```

程序清单 11-5：简单的 HTTP 静态文件服务器

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

require('http').createServer(function(req, res) {
  var file = path.normalize('.' + req.url);
  console.log('Trying to serve', file);

  function reportError(err) {
    console.log(err);
    res.writeHead(500);
    res.end('Internal Server Error');
  }

  path.exists(file, function(exists) {
    if (exists) {
      fs.stat(file, function(err, stat) {
        var rs;

        if (err) {
          return reportError(err);
        }

        if (stat.isDirectory()) {
          res.writeHead(403); res.end('Forbidden');
        } else {
          rs = fs.createReadStream(file);

          rs.on('error', reportError);

          res.writeHead(200);

          rs.pipe(res);
        }
      });
    } else{
      res.writeHead(404);
      res.end('Not found');
    }
  });
}).listen(4000);
```

11.6　示例 2：使用 HTTP 分块响应和定时器

在本例中，将要创建一个输出纯文本的 HTTP 服务器，输出的纯文本每隔 1 秒钟会新增 100 个用换行符分隔的时间戳，(见程序清单 11-6)。

程序清单 11-6：HTTP 分块响应和定时器示例

```
require('http').createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  var left = 10;
  var interval = setInterval(function() {
    for(var i = 0; i< 100; i++) {
      res.write(Date.now() + " ");
    }
    if (-- left === 0) {
      clearInterval(interval);
      res.end();
    }
  }, 1000);
}).listen(4000);
```

11.7　本章小结

Node 的 http 模块允许你创建 HTTP 服务器，并让其在某个特定的端口和主机上监听。然后可以监听新的 request 事件，其中，针对每个请求都有一个回调函数会获得对应的 ServerRequest 对象和 ServerRespnse 对象。

可以观察请求对象的某些属性，例如请求的 URL 和请求头，ServerRequest 对象也是一个可读流，这就意味着除了别的之外，还可以监听包含流式主体数据部分的 data 事件，甚至可以将数据传送到另一个流当中。

可以用 ServerResponse 对象通过设置 HTTP 响应状态和发送响应头对客户端做出响应。还可以向响应主体写入数据，或者将任意可读流传入其中。如果不知道内容长度或者希望有一个连续的响应流，可以使用 HTTP 分块响应协议以流式方式传送内容主体，在内容主体中服务器会发送多块响应主体。

第12章

构建 TCP 客户端

本章提要：

- 连接 TCP 服务器
- 从 TCP 服务器收发数据
- 重启已断开的连接

TCP 位于 IP 的上一层，是应用最广泛的互联网传输协议之一，在其之上就是类似 HTTP 这样的应用层协议。TCP 是一种面向连接的协议，这意味着一个终端需要和另一个终端建立专门的连接，该连接是双向数据流，两个终端可以同时通过该数据流收发数据。TCP 可以保证所收到的消息是有序的，它有一套内置的数据流控制机制，使它成为一个良好的平台，可以支持计算机到计算机以及人到计算机的协议。

正如你将看到的那样，Node 可以非常容易地构建一个 TCP 客户端来建立到另一个终端的连接，该终端通常被称作 TCP 服务器。建立的连接形成了两个数据流，允许你和远程终端进行交互：一个是接收数据的可读流，另一个是写入数据的可写流。

注意：应该牢记在心的一点是：在往 TCP 数据流中写入数据时，不会收到对方收到数据的确认信号。更糟糕的是，因为底层的网络实现也许会随机地切分和路由数据包，所以接收方也许只能收到一部分发送的信息（即使 TCP 内部尝试重发丢失或损坏的数据包）。请记住 TCP 只能保证数据包被应用程序按顺序接收，却不能保证它们被全部接收。

而且，TCP 并不是面向消息的，它只提供了一个连续数据流。如果需要识别单独的消息，就必须实现一个帧协议，指定每个消息的起始位置和结束位置。

例如，HTTP 凭借应用程序解决了上述问题，该应用程序需要一个来自服务器的请求——响应工作流，并且请求和响应消息都被明确定界。

12.1　连接服务器

可以使用 net 模块来连接 TCP 服务器，如下所示：

```
var net = require('net');
var port = 4000;
var conn = net.createConnection(port);
```

注意此处省略了 net.createConnection 函数的第二个参数，它表示想连接的主机名。在缺省该参数的情况下，主机名默认是“localhost”。

```
var net = require('net');
var port = 4000;
var host = "www.acme.com";
var conn = net.createConnection(port, host);
```

可以传递一个回调函数作为 net.createConnection 函数的最后一个参数，这样在连接建立之后就可以得到通知，如下所示：

```
function connectionListener(conn) {
  console.log('We have a new connection');
}

var conn = net.createConnection(port, host, connectionListener);
```

如果尝试连接本地主机，就可以省略 host 参数，而是传递连接监听器回调函数作为 net.createConnection 的第二个参数。

```
var conn = net.createConnection(port, connectionListener);
```

还可以监听 conn 对象发射的 connect 事件：

```
conn.once('connect', connectionListener);
```

12.2　发送和接收数据

调用 net.createConnection()函数的返回值是 net.Socket 类的一个实例，它表示与服务器的连接，它既是可读流也是可写流，允许你发送某些数据：

```
conn.write('here is a string for you!');
```

还可以传递一个字符串，并指定字符串的编码格式：

```
conn.write('SGVsbG8gV29ybGQh', 'base64');
```

还可以向服务器写入一个未经处理的缓冲区：

```
var buffer = new Buffer('Hello World!');
conn.write(buffer);
```

在上述变量之后还可以传递一个回调函数：

```
conn.write('Hey!', function() {
  console.log('data was written out');
});
```

注意这个回调函数并不会在服务器接收数据时被调用，而只会在数据被写入网络时被调用。

当数据可用时，连接都会发射 data 事件，可以通过监听该事件接收来自服务器的数据：

```
conn.on('data', function(data) {
  console.log('some data has arrived:', data);
});
```

如果没有指定流的编码格式，那么传入的数据就是一个缓冲区，如果想要这个缓冲区在发送之前被编码，就需要使用 setEncoding 为其指定编码格式：

```
conn.setEncoding('base64');
```

正如在第 4 章“使用缓冲区处理、编码和解码二进制数据”中介绍的那样，可以使用 ascii、utf8 或者 base64 作为编码标识符。

12.3　终止连接

可以使用 end 方法关闭来自终端的连接：

```
conn.end();
```

这将安排连接在所有数据都被都传送完之后被关闭，在终止连接的同时，也可以发送数据(缓冲区或字符串)：

```
conn.end('Bye bye!', 'utf8');
```

这等效于：

```
conn.write('Bye bye!', 'utf8');
conn.end();
```

警告：当终止连接时，实际上是将该操作放入队列中，这意味着连接不会立刻被关闭。同样，当连接收到“ended”通知时，会发送一个 FIN 数据包，这意味着服务器仍旧可以发送部分数据。

基于以上两点原因，在终止连接之后，依然可以接收“data”事件。

12.4 处理错误

在建立连接时，可能会发生错误——在 DNS 上没有找到主机名、目标主机无法访问或者连接被拒绝——在连接已经建立起来之后，也可能会发生网络错误。可以通过监听 error 事件来捕捉错误，如下所示：

```
conn.on('error', function(err) {
  console.error('this error happened:' + err.message +
    ', code: ' + err.code);
});
```

除了 message 属性外，与网络有关的错误还有一个 code 属性，它是一个字符串，可以查看和测试它。例如，如果请求的连接被拒绝，你会得到一个 ECONNREFUSED 代码。

```
var net = require('net');
var conn = net.createConnection('82837');
conn.on('error', function(err) {
  console.log('I got this error code: ' + err.code);
});
```

最后一段代码应该输出如下所示的信息：

```
I got this error code: ECONNREFUSED
```

12.5 创建命令行 TCP 客户端的示例

下面将要创建一个简单的 TCP 客户端，该客户端尝试连接到服务器上，然后将从其中收发数据。你将用到在第 10 章“构建 TCP 服务器”中构建的 TCP 服务器，并实现连接到其上的客户端。

首先需要在后台启动这个 TCP 服务器，并让其在 4000 号端口上监听，为客户端进行连接做好准备。

12.5.1　连接服务器

在下面的例子中，将连接到本地主机上的 4000 号服务器端口，同时监听任何可能发生的错误。

```
var net  = require('net');
var port = 4000;
var conn = net.createConnection(port);

conn.on('connect', function() {
  console.log('connected to server');
});

conn.on('error', function(err) {
  console.log('Error in connection:', err);
});
```

12.5.2　向服务器发送命令行

当 Node 进程启动时，就准备好了 process.stdin 流来接受用户的键盘输入。这个可读流初始化时处于暂停状态，只有在恢复它之后，才会发射“data”事件，可以调用流上的 resume 方法来恢复流：

```
process.stdin.resume();
```

现在已经有了到服务器的可写流，也有了来自用户输入的可读流，可以使用 pipe 方法将它们一体化，如下所示：

```
process.stdin.pipe(conn);
```

现在，当 process.stdin 流被清空——这通常发生在用户输入一行新字符之后——数据就会通过 conn 可写流传送到服务器。

12.5.3　打印服务器消息

可以将服务器发送给进程标准输出流的每条消息都打印出来，最简单的方法是传送连接输出，如下所示：

```
conn.pipe(process.stdout);
```

然而，上一行命令有一个问题，在默认情况下，sourceStream.pipe(targetStream)会在源流结束时终止目标流，这意味着进程的标准输出流会在连接关闭之后关闭，这不是你希望的结果。幸好，.pipe()方法可以接受一个 end 选项，当将该选项设置为 false 时可以避免这种情况的发生。

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

12.5.4 在连接终止时重新连接

TCP 连接也许会被远程服务器关闭，一旦网络出现问题，它也有可能被关闭，甚至连接处于空闲状态的时间过长也会被关闭。在任何情况下，只要应用程序需要，很容易就可以和服务器重新建立连接，如程序清单 12-1 所示。

程序清单 12-1：立即重新进行连接的 TCP 客户端

```
var net  = require('net');
var port = 4000;
var conn;

process.stdin.resume();

(function connect() {
  conn = net.createConnection(port);

  conn.on('connect', function() {
    console.log('connected to server');
  });

  conn.on('error', function(err) {
    console.log('Error in connection:', err);
  });

  conn.on('close', function() {
    console.log('connection got closed, will try to reconnect');
    connect();
  });

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

在上面的代码中，通过立即调用本地函数 connect 来进行连接。

当连接上发生错误时，你会得到一个 error 事件，然后连接会关闭并触发 close 事件。当 close 事件发生时，上述代码尝试通过调用本地函数 connect 立即进行重新连接(在后面你会看到这是个糟糕的主意)，connect 函数会将全部连接逻辑再次执行一次。

可以试一下上述过程，当连接成功后，关闭服务器后再重启它，然后应该会看到客户端重新连接成功。

然而，并不推荐你在连接断开后立即重新进行连接，也不推荐你不断重试连接，这样做会产生一种循环，使网络上充斥着连接请求，这样一来，服务器在断开连接一段时间后也许会无法访问。取而代之的方法是等一小会，让网络的状态有所改善，或者是等待服务回到在线状态。在几次重新连接失败后，应该停止尝试重新连接——此时的问题也许就可能不是通过重新连接所能解决的了：它也许是永久性的网络问题或者服务器问题。此时用

户应该得到通知，并且程序应该至少在相当一段时间内自动停止尝试重新连接。

在程序清单 12-2 中，将重试间隔变量 retryInterval 设置为 3 秒，并且将最多重试次数限制为 10 次，这两个数值都是随意的，可以基于特定的应用程序进行定义。

程序清单 12-2：立即重新进行连接的 TCP 客户端

```
var net  = require('net');
var port = 4000;
var conn;

var retryInterval = 3000; // 3 seconds
var retriedTimes = 0;
var maxRetries = 10;

process.stdin.resume();

(function connect() {
  function reconnect() {
    if (retriedTimes >= maxRetries) {
      throw new Error('Max retries have been exceeded, I give up.');
    }
    retriedTimes += 1;
    setTimeout(connect, retryInterval);
  }

  conn = net.createConnection(port);

  conn.on('connect', function() {
    retriedTimes = 0;
    console.log('connected to server');
  });

  conn.on('error', function(err) {
    console.log('Error in connection:', err);
  });

  conn.on('close', function() {
    console.log('connection got closed, will try to reconnect');
    reconnect();
  });

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

在上面的代码中，通过在变量 retriedTimes 中存储一个计数器，来跟踪到最后成功连接为止重新尝试了多少次连接，一旦连接成功，retriedTimes 值就被重置为 0。

12.5.5 关闭连接

只需要调用 conn.end()就可以关闭连接，如下所示：

```
conn.end();
```

在上面这个简单的应用程序中，你知道用户在控制台中输入“quit”时希望断开连接，可以检测这个事件并进行相应的操作：

```
process.stdin.on('data', function(data) {
  if (data.toString().trim().toLowerCase() === 'quit') {
    conn.end();
    process.stdin.pause();
  }
});
```

注意同时还结束了标准输入流，让 Node 知道你不再对它感兴趣。

这段代码运行得并不是很好，因为你仍然尝试向服务器传送所有数据，而不管这些数据是不是字符串“quit”，这意味着你也许会向一个已被关闭的连接发送数据。替代的办法是删除下面的传送命令：

```
process.stdin.pipe(conn);
```

你必须在用户没有退出时发送数据：

```
process.stdin.on('data', function(data) {
  if (data.toString().trim().toLowerCase() === 'quit') {
    console.log('quitting...');
    conn.end();
    process.stdin.end();
  } else {
    conn.write(data);
  }
});
```

当连接关闭时，客户端会尝试重新连接，这样就改写了用户指令，因此，必须维持一个变量来说明用户是否想退出，这样就会避免重新连接的出现。

```
var quitting = false;
// ...

process.stdin.on('data', function(data) {
  if (data.toString().trim().toLowerCase() === 'quit') {
    quitting = true;
    console.log('quitting...');
    conn.end();
    process.stdin.pause();
  } else {
    conn.write(data);
  }
```

```
});

// ...

conn.on('close', function() {
  if (! quitting) {
    console.log('connection got closed, will try to reconnect');
    reconnect();
  }
});
```

12.5.6　前述内容综合

程序清单 12-3 展示了 TCP 客户端重新连接和退出的完整源代码。

程序清单 12-3：一个新版的命令行 TCP 客户端，在连接断开时重新连接

```
var net  = require('net');
var port = 4000;
var quitting = false;
var conn;
var retryTimeout = 3000; // 3 seconds

var retriedTimes = 0;
var maxRetries = 10;

process.stdin.resume();

process.stdin.on('data', function(data) {
  if (data.toString().trim().toLowerCase() === 'quit') {
    quitting = true;
    console.log('quitting...');
    conn.end();
    process.stdin.pause();
  } else {
    conn.write(data);
  }
});

(function connect() {
  function reconnect() {
    if (retriedTimes >= maxRetries) {
      throw new Error('Max retries have been exceeded, I give up.');
    }
    retriedTimes += 1;
    setTimeout(connect, retryTimeout);
  }

  conn = net.createConnection(port);

  conn.on('connect', function() {
```

```
    retriedTimes = 0;
    console.log('connected to server');
  });

  conn.on('error', function(err) {
    console.log('Error in connection:', err);
  });

  conn.on('close', function() {
    if (! quitting) {
      console.log('connection got closed, wi.l try to reconnect');
      reconnect();
    }
  });

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

}());
```

12.6 本章小结

TCP 是面向连接的协议，可以为你按序处理消息和控制数据流，TCP 是一个很好的传输层协议，在其上能够创建客户端-服务器协议。

在 Node 中构建 TCP 客户端很容易，当尝试与 TCP 服务器建立连接时，net.createConnection() 会返回一个 net.Socket 实例，它既是可读流，也是可写流，可以用该实例传输数据。然后可以从连接发送和接收数据，这些数据既可以是未经处理的缓冲区，也可以是进行了编码的字符串。

应该监听来自连接的错误事件，并且当连接关闭时也应该检测到。可以使用 close 事件尝试重新建立连接，但是此时必须十分小心。在尝试两次重建连接之间等上片刻，同时设置允许连接失败的最大次数，以避免应用程序陷入无限循环。

第13章

创建 HTTP 请求

本章提要：

- 向 HTTP 服务器发送 GET 请求
- 在 HTTP 请求上使用不同的 HTTP 动词
- 查看请求响应状态和头部的属性
- 使用第三方请求模块

HTTP 已经成为互联网上许多私有和公有服务的底层基础的核心部分，HTTP 不仅被用来提供静态数据，而且也成为提供和使用公共 API 的首选方式。

由于 Node 擅长处理 I/O 操作，所以它不仅适合提供 HTTP 服务，也适合使用这些服务。在接下来的几节里，你将学习使用核心 http 模块和实用的第三方请求模块执行和控制 HTTP 请求。

在 HTTP 协议中，请求有两个重要的属性：URL 和方法。最常见的方法是 GET 方法，它主要用来请求内容，同时还有其他方法，如 POST(主要被用来提交 Web 表单)、PUT、DELETE 和 HEAD，每一种方法都会从服务器获得不同的结果。

13.1 创建 GET 请求

下面这个例子使用 http.get 方法针对 URL www.google.com:80/index.html 创建了一个 HTTP GET 请求。

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

var options = {
  host: "www.google.com",
```

```
  port: 80,
  path: "/index.html"
};

http.get(options, function(res) {
  console.log('Got response:' + res.statusCode);
});
```

在上面这个例子中，使用 TCP 端口 80 向主机名 www.google.com 和路径/index.html 执行 GET 请求。回调函数会在响应到达时被调用来处理响应对象。在本章后面你将学习可以对响应对象所进行的操作。

13.2 使用其他 HTTP 动词

http.get 是通用的 http.request 的快捷方式，其选项如下：

- host——想发送请求的主机名或者 IP 地址。
- port——远程服务器的 TCP 端口号。
- method——指定 HTTP 请求方法的字符串，其可能的值为 GET(默认方法)、POST、PUT、DELETE、HEAD 或者请求接收端能够理解的其他方法。
- path——请求的路径，应该包括一个查询字符串和一个分隔符，例如，/index?page=2。
- headers——表示值-名称对的形式的请求头的对象。下面是 header 对象的一个例子：

```
{
  "Accept"           : "text/json",
  "If-Modified-Since" : "Sat, 28 Jan 2012 00:00:52 GMT"
}
```

http.request 函数会返回一个 http.ClientRequest 对象，它是一个可写流，可以使用这个流发送数据，所发送的数据是请求主体数据的一部分，当完成向请求主体中写入数据时，要结束数据流来终止请求。

下面这个例子中将一些请求主体内容写入到 http://my.server.com/upload 中：

```
var http = require('http');
var options = {
  host: "www.google.com",
  port: 80,
  path: "/upload",
  method: "POST"
};

var request = http.request(options, function(response) {
  console.log('STATUS:', response.statusCode);
  console.log('HEADERS:', response.headers);
  response.setEncoding('utf8');
  response.on('data', function (chunk) {
```

```
    console.log('BODY:', chunk);
  });
});
request.write('This is a piece of data.\n');
request.write('This is another piece of data.\n');
request.end();
```

http.request 函数返回 ClientRequest 对象，可以通过调用 request.write()函数将请求主体部分写入该对象。可以传入一个经过编码的字符串——通过使用 request.write(str, encoding)函数——一个 UTF-8 编码的字符串，或者甚至一个未经处理的缓冲区。必须用 request.end()函数来结束请求，如果没有结束请求，服务器会认为请求不完整，也就不会对其进行响应。

如果服务器可访问并且正在运行，会获得一个响应，这个响应将触发 response 事件。可以将 response 事件绑定到 request 对象上，如下所示：

```
function responseHandler(res) {
  console.log('I got a response:', res);
}
request.on('response', responseHandler);
```

也可以传递一个回调函数给 http.request()调用，作为其第二个参数，当出现来自服务器的响应时，这个回调函数同样会被调用。

回调函数会将响应对象作为第一个也是唯一一个参数，通过查看该对象，可以发现服务器分配给请求的响应状态码以及服务器发送的响应头。

13.2.1　查看响应对象

当 HTTP 服务器的响应返回时，响应事件就起作用了，它会触发所有为其注册的回调函数，并将响应对象传递给这些回调函数，响应对象是 http.ClientResponse 的一个实例，可以立即查看该对象的属性：

- response.statusCode——一个整数，表示 HTTP 响应状态码。
- response.httpVersion——服务器在响应上实现的 HTTP 版本号，可能是 1.1 或者 1.0。
- response.headers——表示响应头名称-值对的普通对象，对象名是小写的，下面是响应头对象的一个示例：

```
{ date: 'Wed, 01 Feb 2012 16:47:51 GMT',
  expires: '-1',
  'cache-control': 'private, max-age=0',
  'content-type': 'text/html; charset=ISO-8859-1',
  'set-cookie':
  [ 'NID=56=rNJVRav-ZW1Sd4f9NPsjLhaybMSzTOfbMPNHCqLeYXwKAvs4_f...],
  p3p: 'CP="This is not a P3P policy! See
  http://www.google.com/support/accou...',
  server: 'gws',
  'x-xss-protection': '1; mode=block',
  'x-frame-options': 'SAMEORIGIN',
  connection: 'close' }
```

13.2.2 获取响应主体

前面提到过，响应主体并不会在请求的 response 事件发生时出现，如果对响应主体感兴趣，就可以在 response 上注册“data”事件，如下所示：

```
http.request(options, function(response) {
  response.setEncoding('utf8');
  response.on('data', function(data) {
    console.log('I have a piece of the body here', data);
  });
});
```

在上面的例子中，一旦获取 response 事件，就在响应上注册“data”事件，当有响应主体到达时，会收到通知。响应主体可以是一个缓冲区，如果在响应对象上指定了编码格式(就如同在上面的例子中所做的那样)，它就是一个编码字符串。

13.2.3 以流的方式传送响应主体

HTTP 响应是一个可读流，表示响应主体数据流。与任何可读流一样，可以将它传送到任意可写流中，例如一个 HTTP 请求或者一个文件。例如，下面的例子展示了如何将响应主体传送到一个打开的文件中：

```
var http = require('http');
var fs   = require('fs');
var options = {
  host: "www.google.com",
  port: 80,
  path: "/",
  method: "GET"
};

var file = fs.createWriteStream('/tmp/test.txt');

http.request(options, function(res) {
  res.pipe(file);
}).end();
```

上面的例子中创建了一个文件可写流，并将响应主体传送到其中，当来自服务器响应的主体数据到达时，响应主体就被写入文件。在响应主体结束时，文件流也会结束，然后就会关闭文件。

13.3 使用 HTTP.Agent 维护套接字池

在创建 HTTP 请求时，Node 在内部使用了一个代理。代理是 Node 中的一种实体，该实体被用来为你创建请求，它负责维护一个活动套接字池，对指定的主机名和端口对，套接字池中包含已被打开但是未被使用的连接。当出现新的 HTTP 请求时，代理要求连接

保持活动状态。当请求结束，并且在套接字上没有额外的请求等待释放时，套接字就会被关闭。这意味着不必手动关闭 HTTP 客户端，并且 TCP 连接在 Node 负载较轻时易于重用。

当创建了一个请求并选择了一个可用的套接字，或者为该请求创建了一个新的连接时，http.ClientRequest 会发射 socket 事件。在请求结束之后，套接字会在发射 close 事件或 agentRemove 事件时从代理的套接字池中被删除。如果想让 HTTP 请求打开一段较长时间，可以将其从套接字池中删除，如下所示：

```
function handleResponseCallback(res) {
  console.log('got response:', res);
}

var req = http.request(options, handleResponseCallback);

req.on('socket', function(socket) {
  socket.emit('agentRemove');
});
```

Node 允许在一个给定线程上的每个主机-端口对上最多可以有 5 个打开的套接字，这意味着在负载较重的情况下，Node 会将请求串行排列到同一个主机-端口对以便重用套接字。如果这种处理对你的应用模式来说并不是最优，可以通过向 option 对象传递 agent: false 来使代理的套接字池失效：

```
var options = {
  host: "my.server.com",
  port: 80,
  path: "/upload",
  method: "POST",
  agent: false
};

http.request(options, handleResponseCallback);
```

还可以通过修改 http.Agent.defaultMaxSockets 来改变套接字池中每个主机-端口对所允许打开的套接字的最大数目，如下所示：

```
var http = require('http');
http.Agent.defaultMaxSockets = 10;
```

对这个默认值的修改是全局性的，这意味当有新的 HTTP 代理被创建时，它依然会使用修改后的默认值。注意，如果已经针对给定的主机名和端口创建了请求，对应的 HTTP 代理实例也会被创建，所以此时对默认值进行修改不会有任何作用。

当创建请求时也可以指定 HTTP 代理，如下所示：

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

```
  maxSockets: 10
};
var agent = new Agent(options);

// 使用该请求的代理:
var requestOptions = {
  host: 'www.google.com',
  port: 80,
  agent: agent
};
var req = http.request(requestOptions);
// ...
req.end();
```

上述代码用给定的代理创建 HTTP 请求，该代理会将套接字池存储在其中，可以共享创建的代理，并将其应用到其他 HTTP 请求中以重用套接字池。

13.4　应用第三方请求模块简化 HTTP 请求

Node 的 HTTP 客户端非常强大，但用起来有点麻烦。首先，必须提供一个对象，该对象要具有所有选项，包括被分割成几个部分的 URL，之后如果希望能够处理响应主体，还需要获取响应主体。如果是重定向响应，还必须人工处理该响应。

通过使用 request 模块可以避免上述问题以及很多其他的问题，request 模块的作者是 Mikeal Rogers，他是 Node 的核心贡献者之一。

13.4.1　安装和应用 request 模块

要在当前目录下安装第三方请求模块，可以使用如下所示的命令：

```
$ npm install request
```

然后在应用程序或者模块中包含它：

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

之后只需要提供一个 URL 和一个回调函数就可以创建请求了，如下所示：

```
request('http://www.acme.com:4001/something?page=2',
  function(error, response, body) {
    // ...
  }
);
```

上述代码会对指定的 URL 执行 HTTP GET 请求，同时接收响应和响应主体缓冲区，并将它们传递给一个简单的回调函数作为参数。

可以使用其他的简便方法修改 HTTP 动词，如下所示：

- request.put(url)——发布一个 PUT 请求。
- request.post(url)——发布一个 POST 请求。
- request.head(url)——发布一个 HEAD 请求。
- request.del(url)——发布一个 DELETE 请求。
- request.get(url)——发布一个 GET 请求，该方法只是追求形式上的统一，可以用默认的 request()代替它。

还可以使用 option 对象来代替包含 URL 的字符串，该对象如下所示：

```
{
  url: 'http://www.acme.com:4001/something',
  method: 'DELETE',
  headers: {Accept: 'application/json'},
  body: new Buffer('Hello world!')
}
```

下面是一些可接受参数的清单：

- uri 或者 url——完全合法的 URI 或者是经 url.parse()解析的 URL 对象。例如：https://my.server.com/some/path?a=1&b=2.
- method——HTTP 方法，默认是 GET。
- headers——HTTP 请求头，默认值是{ }。
- qs——作为查询字符串附加到 URL 后的名称-值对，例如：{a:1, b:2}。
- body——POST 和 PUT 请求主体，必须是缓冲区或字符串。
- form——将请求主体设置为查询字符串的形式，并且在请求头中增加了内容类型 application/x-www-form-urlencoded; charset=utf-8。
- json——将请求主体设置为 JSON 的形式，并且在头部数据中增加了内容类型 application/json。
- followRedirect——跟随具有状态码 3xx 的响应，这类响应表示重定向，默认为 true。
- maxRedirects——跟随重定向的最多次数，默认是 10。
- onResponse——如果 onResponse 为 true，回调函数将会在 response 事件发生时被调用，而不是在 end 事件发生时被调用。
- encoding——Encoding 是 setEncoding 函数用来为响应数据设置编码格式的，如果 encoding 为 null，响应主体将以缓冲区的形式返回。
- pool——表示请求代理的哈希对象，如果忽略该选项，请求会使用全局套接字池，全局套接字池被设置为 Node 中 maxSockets 的默认值。
- pool.maxSockets——一个整数，表示套接字池中套接字的最大数目。
- timeout——一个整数，表示在放弃之前等待对请求做出响应的时间，单位是毫秒。

13.4.2　创建测试服务器

要想全面理解第三方请求模块是如何工作的，可以创建一个简单的 HTTP 服务器来打

印一些与 HTTP 请求有关的重要信息，如程序清单 13-1 所示。

程序清单 13-1：请求内省的 HTTP 服务器

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

  function printBack() {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end(JSON.stringify({
      url: req.url,
      method: req.method,
      headers: req.headers
    }));
  }

  switch (req.url) {

    case'/redirect':
      res.writeHead(301, {"Location": '/'});
      res.end();
      break;

    case'/print/body':
      req.setEncoding('utf8');
      var body = '';
      req.on('data', function(d) {
        body += d;
      });
      req.on('end', function() {
        res.end(JSON.stringify(body));
      });
      break;

    default:
      printBack();
      break;

  }
}).listen(4001);
```

将上面的代码保存为文件 server.js，并用如下所示的命令行运行它：

```
$ node server.js
```

然后要对这个服务器进行简单的调用，如程序清单 13-2 所示。

程序清单 13-2：创建一个简单的请求

```
var request = require('request');
var inspect = require('util').inspect;
```

```
request(
  'http://localhost:4001/abc/def',
  function(err, res, body) {
    if (err) { throw err; }
    console.log(inspect({
      err: err,
      res: {
        statusCode: res.statusCode
      },
      body: JSON.parse(body)
    }))
  }
);
```

不用停止服务器，应该将上述代码保存为文件 simple.js，并且在命令行下运行它：

```
$ node simple.js
```

结果应该是：

```
{ err: null,
  res: { statusCode: 200 },
  body:
   { url: '/abc/def',
     method: 'GET',
     headers:
      { host: 'localhost:4001',
        'content-length': '0',
        connection: 'keep-alive' } } }
```

可以从中看出服务器从请求中得到什么样的请求头、URL 和方法，你能看到这些是因为没有向请求流传入任何数据，也没有向其写入任何数据，因此 request 模块会假定请求主体的长度为 0。

13.4.3　跟随重定向

request 模块的众多特性之一是它在默认情况下能够跟随重定向，可以用运行中的请求内省服务器来观察这个特征，内省服务器进行了编码，以便当 URL 是/redirect 时，可以用 301 重定向响应状态码做出响应。可以用客户端来查看跟随上述 URL 的请求，如程序清单 13-3 所示。

程序清单 13-3：创建会被服务器重定向的请求

```
var request = require('request');
var inspect = require('util').inspect;

request(
  'http://localhost:4001/redirect',
  function(err, res, body) {
```

```
    if (err) { throw err; }
    console.log(inspect({
      err: err,
      res: {
        statusCode: res.statusCode
      },
      body: JSON.parse(body)
    }))
  });
```

将程序清单 13-3 保存为文件 redirect.js 并运行它：

```
$ node redirect.js
```

可以看到如下所示的输出信息：

```
{ err: null,
  res: { statusCode: 200 },
  body:
   { url: '/',
     method: 'GET',
     headers:
      { 'content-length': '0',
        host: 'localhost:4001',
        connection: 'keep-alive' } } }
```

可以看到服务器返回的 URL 是/，而不是原来的/redirect，这意味着跟随服务器的请求重定向为/。

如果你希望在发生重定向时得到通知，就应该将请求选项的 followRedirect 属性设置为 false，以此来禁用跟随重定向属性。

某些服务器可能会发生错误，会使重定向陷入循环中，这也是为什么请求要定义重定向最多次数的原因，重定向最多次数的默认值是 10，但是可以根据需要对该值进行调整。例如，你也许希望每个请求的重定向最多次数为 3，如果重定向超过 3 次，则操作失败。然而，你也许还是愿意在每个请求上都确保一个较大的重定向次数。在这个例子中，可以将选项属性 maxRedirects 设置为希望的值。

```
var options = {
  url: 'http://www.example.com',
  maxRedirects = 3
};
request(options, callback);
```

13.4.4 设置请求选项

除了请求中的 GET 方法外，还可以执行一些其他的 HTTP 方法。例如，可以使用 options 的方法属性并对 HTTP 方法进行编码，这和原生的 Node http.request 方法非常相似。可以使用如下所示的快捷方式：

```
request.get
request.put
request.post
request.del
```

程序清单 13-4 展示了一个使用 POST 请求的例子。

程序清单 13-4：创建 POST 请求

```
var request = require('request');
var inspect = require('util').inspect;
request.post(
  'http://localhost:4001/abc/def',
  function(err, res, body) {
    if (err) { throw err; }
    console.log(inspect({
      err: err,
      res: {
        statusCode: res.statusCode
      },
      body: JSON.parse(body)
    }))
  });
```

将程序清单 13-4 存为文件 post.js 并运行：

```
$ node post.js
```

应该得到如下所示的信息：

```
{ err: null,
  res: { statusCode: 200 },
  body:
   { url: '/abc/def',
     method: 'POST',
     headers:
      { host: 'localhost:4001',
        'content-length': '0',
        connection: 'keep-alive' } } }
```

上面的输出信息证实了服务器能够理解所发出的请求是 POST 请求。

更一般的情况是请求可以接受 options 对象来代替字符串 URL：

```
var request = require('request');
var inspect = require('util').inspect;

var options = {
  url: 'http://localhost:4001/abc/def',
  method: 'PUT'
};
```

```
request(options, function(err, res, body) {
  if (err) { throw err; }
console.log(inspect({
    err: err,
    res: {
      statusCode: res.statusCode
    },
    body: body
  }))
});
options.js
```

通过使用 option 对象，可以发送一些自定义的请求头，如程序清单 13-5 所示。

程序清单 13-5：发送自定义的请求头

```
var request = require('request');
var inspect = require('util').inspect;

var options = {
  url: 'http://localhost:4001/abc/def',
  method: 'PUT',
  headers: {

    'X-My-Header': 'value'
  }
};

request(options, function(err, res, body) {
  if (err) { throw err; }
  console.log(inspect({
    err: err,
    res: {
      statusCode: res.statusCode,
      headers: res.headers
    },
    body: JSON.parse(body)
  }))
});
```

将程序清单 13-5 存入文件 headers.js 并运行：

```
$ node headers.js
```

然后会得到如下所示的输出信息：

```
{ err: null,
  res:
   { statusCode: 200,
     headers:
      { 'content-type': 'text/plain',
```

```
      connection: 'keep-alive',
      'transfer-encoding': 'chunked' } },
body:
 { url: '/abc/def',
   method: 'PUT',
   headers:
    { 'x-my-header': 'value',
      host: 'localhost:4001',
      'content-length': '0',
      connection: 'keep-alive' } } }
```

可以看到服务器得到了请求头。

13.4.5　对请求体进行编码

有时必须在请求主体上发送一些数据，可以使用表单编码对请求主体进行编码，这是浏览器对请求主体字符串进行编码的一种模拟。查看程序清单 13-6。

代码清单 13-6：发送表单编码的请求主体

```
var inspect = require('request');
var inspect = require('util').inspect;

body = {
  a: 1,
  b: 2
};

var options = {
  url: 'http://localhost:4001/print/body',
  form: body
};

request(options, function(err, res, body) {
  if (err) { throw err; }
  console.log(inspect({
    err: err,
    res: {
      statusCode: res.statusCode,
      headers: res.headers
    },
    body: JSON.parse(body)
  }))
});
```

将上面的代码保存为文件 form.js 并运行：

```
$ node form.js
```

可以得到如下所示的输出信息：

```
{ err: null,
 res:
  { statusCode: 200,
   headers: { connection: 'keep-alive', 'transfer-encoding': 'chunked' } },
 body: 'a=1&b=2' }
```

可以看到 qs 对象被编码写入了请求主体。

此外，对请求主体中的某些参数可以采用 JSON 编码，如程序清单 13-7 所示。

程序清单 13-7：发送 JSON 编码的请求主体

```
var request = require('request');
var inspect = require('util').inspect;

body = {
  a: 1,
  b: 2
};

var options = {
  url: 'http://localhost:4001/print/body',
  json: body
};

request(options, function(err, res, body) {
  if (err) { throw err; }
  console.log(inspect({
    err: err,
    res: {
      statusCode: res.statusCode,
      headers: res.headers
    },
    body: JSON.parse(body)
  }))
});
```

将上述代码保存为文件 json.js 并运行：

```
$ node json.js
```

可以得到如下所示的输出信息：

```
{ err: null,
 res:
  { statusCode: 200,
   headers: { connection: 'keep-alive', 'transfer-encoding': 'chunked' } },
 body: { a: 1, b: 2 } }
```

13.4.6 流式传送

请求可以返回一个客户端请求对象，可以将该对象传入可写流中(正如我在本章前面所

展示的 Node 核心 HTTP 请求函数那样)，甚至可以将请求对象传入另一个请求。例如，如果想将请求的响应传入文件流中，可以按如下方式：

```
var fs = require('fs');
var request = require('request');
var file = fs.createWriteStream('/path/to/my/file');

request.get('http://www.example.com/tmp/test.html').pipe(file);
```

还可以将一个请求传入另一个请求中，例如，可以将一个 GET 请求传入另一个 POST 请求中，如下所示：

```
var request = require('request');
var source = request.get('http://my.server.com/images/some_file.jpg');
var target =
request.post('http://other.server.com/images/some_file.jpg');
source.pipe(target);
```

13.4.7　使用 Cookie Jar

默认情况下，请求会解释 HTTP cookies，并将其存入全局的 cookie jar，这意味着对于应用程序而言，所有的请求都会发送来自给定主机的 cookies。

这种行为有时是有用的，但是也许你不想获取每个 cookie，因为这些 cookie 也许对于应用程序并不重要。可以通过选择请求的默认值以便在全局禁用 cookies：

```
request.defaults({jar: false});
```

还可以通过在请求发出之前，将请求选项中的 jar 属性设置为 false 来关闭该行为：

```
var options = {
  url: 'http://www.example.com/',
  jar: false };
request(options, callback);
```

此外，还可以通过在 option 对象中创建和指定一个 cookie jar，使得每个请求都可以使用这个特定的 cookie jar：

```
var jar = request.jar();
var options = {
  url: 'http://www.example.com/',
  jar: jar };
request(options, callback);
```

13.5　本章小结

在 Node 中通过使用 http 模块的 request()函数可以很容易地创建 HTTP 请求，request()函数会返回一个 http.ClientResponse 对象，该对象会在服务器做出响应时发射 response 对

象。只要愿意，可以以流的方式传送响应主体，或者将响应主体传入另一个可写流。此外还可以查看响应头、状态码和 HTTP 的版本。

为了简化创建 HTTP 请求，也许应该使用 Mikeal Roger 的 request 模块，在大多数情况下该模块会让你使用 URL。这个模块有一些很有用的特性，例如跟随重定向、记录 cookies、缓冲响应主体、设置查询字符串的值、设置表单编码或者 JSON 编码的请求主体以及流式传递。

第14章

使用用户数据报

本章提要：

- 何时使用用户数据报？为什么要使用它？
- 接收用户数据报消息
- 发送用户数据报消息
- 多点传送用户数据报消息
- 理解用户数据报的限制

TCP 是一个面向连接的协议，它在网络节点之间提供可靠有序的数据流。然而，用户数据报协议(User Datagram Protocol, UDP)是一种无连接的协议，不具备 TCP 的传输特性。当发送 UFP 数据包时，无法保证数据包的到达顺序，甚至无法保证它能够到达。

除了这些缺陷外，UDP 十分有用，例如在需要广播消息时并不需要严格地传输保证或者消息顺序，此外 UDP 也可用于不知道网络节点地址的场合。

14.1 理解用户数据报

与 TCP 一样，UDP 也位于传输层，在 IP 层之上，它允许你向其他主机发送消息(此处的消息是指数据报)，而不需要事先的通信来建立必要的传输信道或者数据路径。与 TCP 不一样的是，UDP 不是面向连接的，不具有可靠性、消息排序甚至数据完整性等语义。UDP 提供的是一种不可靠的服务，数据报可能不会按顺序到达、重复出现甚至在无察觉的情况下发生丢失。UDP 假定错误检测要不没有必要，要不就将其放在应用层进行。

14.2　理解用户数据报的使用

时间敏感应用程序通常使用 UDP，因为此时丢失数据包比等待延时的数据包更适合，后者并不是实时系统的选择。

UDP 的无状态特性对于服务器响应来自大量客户端的小查询同样有用。与 TCP 不同，UDP 支持数据包的广播和多播(将数据包发送给所有订阅者)。

使用 UDP 的常见网络应用程序包括 DNS(Domain Name System，域名系统)、流媒体应用程序，例如交互式网络电视(Internet Protocol Television, IPTV)、网络语音(Voice over IP, VOIP)以及 IP 隧道、简单文件传输协议(Trivial File Transfer Protocol, TFTP)、简单网络管理协议(Simple Network Management Protocol，SNMP)、动态主机配置协议(Dynamic Host Coniguration Protocol, DHCP)等。其他常见的应用程序包括强可靠性不必要时的消息记录以及清理分布式缓存。

14.3　构建数据报服务器

Node 使用 dgram 模块来支持数据报，所以首先必须导入该模块：

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

然后必须创建服务器：

```
var server = dgram.createSocket('udp4');
```

dgram.createSocket 函数接受套接字类型作为第一个参数，该参数或者是 udp4(IPv4 上的 UDP)，或者是 udp6(IPv6 上的 UDP)，上面的例子中指定的是 udp4。

注意：此外还要注意在 UDP 中不存在真正的服务器，它只是在套接字上监听消息的终端，服务器一般要对收到的消息提供某种类型的反馈，在本章的后面你将看到如何响应 UDP 消息。

为了简化问题和消除歧义，我仍然将你创建的这种软件称作“服务器”。

14.3.1　监听消息

当有新消息传入时，UDP 服务器会发射 message 事件，所以必须为该事件注册一个回调函数：

```
server.on('message', function(message) {
  console.log('server got message: ' + message);
```

```
});
```

现在应该将服务器绑定到一个特定的 UDP 端口上，如下所示：

```
var port = 4000;server.on('listening', function() {
  var address = peer.address();
  console.log('peer listening on ' + address.address + ':' + address.port);
});
server.bind(port);
```

同样也可以监听 listening 事件，该事件会在服务器开始监听消息之后被触发。

14.3.2　测试服务器

程序清单 14-1 包含了完整的 UDP 服务器代码。

程序清单 14-1：一个简单的 UDP 服务器

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

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

server.on('message', function(message) {
  console.log('server got message: ' + message);
});

var port = 4000;

server.on('listening', function() {
  var address = server.address();
  console.log('server listening on ' + address.address + ':' + address.port);
});
server.bind(port);
```

将程序清单 14-1 中的代码保存为文件 udp_server_1.js 并运行：

```
$ node udp_server_1
```

上述脚本应该输出服务器地址和端口，然后只是等待消息：

```
server listening on 0.0.0.0:4000
```

可以通过使用类似 nc 这样的工具发送数据来测试上述脚本：

```
$ echo hello | nc -u 0.0.0.0 4000
```

上面的调用会向主机的 4000 号端口发送“hello”UDP 数据包，然后你会获得服务器的输出信息，如下所示：

```
server got message: hello
```

14.3.3 查看附加的消息信息

由于 UDP 服务器可以获取来自大量发送端的消息，因此根据应用程序获取消息的源地址就十分令人关注。除了发送消息本身外，message 事件还会将远程发送端的一些基本信息传入回调函数，作为其第二个参数，可以查看这些消息，如下所示：

```
server.on('message', function(message, rinfo) {
  console.log('server got message: %s from %s:%d',
              message,
              rinfo.address,
              rinfo.port);
});
```

正如在本章后面将要看到的那样，可以用这些源信息响应远程服务器。

14.4 创建简单的数据报回送服务器

既然已经在 message 事件上获得了源地址，也许你想发回一条消息。例如，也许你想告知源服务器消息已收到，或者将某些状态反馈给源服务器。

将创建一个简单的 UDP 服务器作为示例，将接收到的消息发送回其发送端。

14.4.1 等待消息

可以从创建一个 UDP 套接字并将其绑定到某一端口上开始：

```
var dgram = require('dgram');
var server = dgram.createSocket('udp4');
var port = 4000;

server.on('message', function(message, rinfo) {
  console.log('server got message: %s from %s:%d',
              message,
              rinfo.address,
              rinfo.port);
});

server.bind(port);
```

14.4.2 向发送端发回消息

然后，当服务器收到一条消息时，将它发送回发送端：

```
server.on('message', function(message, rinfo) {
  server.send(message,
              0,
              message.length,
              rinfo.port,
              rinfo.address);
});
```

上面的例子中用到了服务器的 send 函数，该函数会向指定的地址和端口发送消息。

前三个参数对消息进行描述——第一个是一个缓冲区，然后是缓冲区中消息的初始偏移(本例中该值为 0，因为消息是从缓冲区的起始位置开始的)，再然后是消息的长度(即缓冲区的实际长度)，单位为字节。

剩下的两个参数描述了消息的目的地，即端口和地址。在上面的例子中，得到了包含在 rinfo 参数中的远程发送端信息——它包含了远程服务器的地址信息——并且使用该信息指定接收者的地址。

14.4.3 前述内容综合

现在 UDP 回送服务器应该如下所示：

```
var dgram = require('dgram');
var server = dgram.createSocket('udp4');
var port = 4000;

server.on('message', function(message, rinfo) {
  server.send(message, 0, message.length, rinfo.port, rinfo.address);
});

server.bind(port);
```

可以在 shell 命令行中用 nc 实用工具来测试该服务器：

```
$ nc -u 0.0.0.0 4000
```

然后开始输入字符串并按下回车键，应该看到字符串被送回。

注意：在消息被复制到内核之前，注意不要修改传递给 client.send 函数的缓冲区。如果想知道消息在何时被输入到内核中，以及何时能够安全地重用缓冲区，就必须将一个回调函数作为最后一个参数传递给 client.send 函数：

```
var message = new Buffer('blah blah');
client.send(message,
            0,
            message.length,
            4000,
            'localhost',
            function() {
              // 可以重用消息缓冲区了
            });
```

14.5 构建数据报客户端

根据应用程序，你也许只想发送一条或多条 UDP 消息，而并不关心监听消息的响应

或者任何其他 UDP 消息，在这种情况下，可以创建一个只发 UDP 客户端。

14.5.1　创建客户端

要创建一个 UDP 客户端来发送 UDP 数据包，只需要创建一个 UDP 套接字，并用它来发送消息，如下所示：

```
var dgram = require('dgram');
var client = dgram.createSocket('udp4');
```

在上面的例子中，除了没有绑定端口外，使用的是和服务器上一样的 dgram.createSocket 函数。由于没有绑定到特定的端口，因此消息会从一个随机的 UDP 端口发出。如果想从一个特定端口发送消息，可以使用 client.bind(port)函数。

14.5.2　发送消息

下面可以创建一个包含消息的缓冲区，并通过客户端套接字将该缓冲区发送给服务器。

```
var message = new Buffer('this is a message');
client.send(message, 0, message.length, 4000, 'localhost');
```

14.5.3　关闭套接字

当不再需要发送消息时，可以关闭数据报套接字，在上面的例子中，套接字会在消息被全部传送到网络中时被关闭。幸好，client.send 函数会在套接字被关闭时接受一个回调函数，在本例中用到了这个回调函数：

```
client.send(message,
            0,
            message.length,
            4000,
            'localhost',
            function(err, bytes) {
              if (err) { throw err; }
              client.close();
            }
);
```

如你所见，传入 client.send 函数的回调函数有两个参数：一个是发生错误时的错误对象，另一个是发送的字节数。

14.6　创建一个简单的数据报命令行客户端

现在准备用一个基于 Node 的命令行 UDP 客户端来代替 nc 客户端，这个客户端程序应该接受目标主机名和端口作为命令行参数，然后从标准输入数据流中读取数据，并将数

据流作为消息传送给 UDP 目标。

从创建脚本文件和输入如下所示的代码开始：

```
#!/usr/bin/env node
```

将上述代码保存为文件 udp_client.js，并将其修改成可执行文件：

```
$ chmod +x udp_client.js
```

可以从 shell 命令行调用客户端，如下所示：

```
$ udp_client.js <host> <port>
```

然后解析命令行参数，这些参数是 process.argv 中的变量，它是一个包含所有命令行参数的数组，第一个参数是 Node 可执行文件的路径，第二个参数是主脚本文件的路径，第三个参数是主机名，第四个参数是端口。

```
var host = process.argv[2];
var port = parseInt(process.argv[3], 10);
```

14.6.1　读取命令行

为了在 Node 进程启动之后读取命令行，必须使用 process.stdin 可读流。然而，process.stdin 流启动时处于暂停状态，这意味着在使用之前必须恢复它，如下所示：

```
process.stdin.resume();
```

然后就可以用这个标准输入流来获取命令行：

```
process.stdin.on('data', function(data) {
  console.log('user typed:', data);
});
```

14.6.2　向服务器发送数据

现在可以将用户输入传给目标 UDP 服务器：

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

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

process.stdin.on('data', function(data) {
  client.send(data, 0, data.length, port, host);
});
```

14.6.3　从服务器接收数据

要接收从服务器返回的消息，需要订阅 message 事件：

```
client.on('message', function(message) {
  console.log('Got message back:', message.toString());
});
```

14.6.4 前述内容综合

程序清单 14-2 展示了完整的 UDP 命令行客户端。

程序清单 14-2：命令行 UDP 客户端

```
#!/usr/bin/env node
var dgram = require('dgram');

var host = process.argv[2];
var port = parseInt(process.argv[3], 10);
var client = dgram.createSocket('udp4');

process.stdin.resume();

process.stdin.on('data', function(data) {
  client.send(data, 0, data.length, port, host);
});

client.on('message', function(message) {
  console.log('Got message back:', message.toString());
});

console.log('Start typing to send messages to %s:%s', host, port);
```

可以使用 shell 命令行来启动客户端：

```
$ ./udp_client.js 0.0.0.0 4000
```

应该看到如下所示的信息：

```
Start typing to send messages to 0.0.0.0:4000
```

如果输入一些字符，然后按下回车键，应该看到发送的消息被当作响应返回，并显示出来。

14.7 理解和使用数据报多播

UDP 最令人感兴趣的应用之一是能够将一条消息发布给几个网络节点，可以利用这一点来记录日志和清理缓存，在两种情况下，都可能会丢失一些消息。

注意：消息多播在不需要知道所有终端地址时可能会很有用，终端只需要转入接收状态并监听信道即可。

例如，可以让一个事件记录服务器监听多个“信道，当另一个进程需要登录一个“信道”时，它将会向该通道多播包含该事件细节的消息。

还可以让多个服务器监听同一通道，例如，一个服务器负责存储事件，另一个服务器负责向分析引擎发送事件。

节点可以通过转入一个信道公布它对监听该多播信道的兴趣，在分配 IP 地址时，有一个地址范围是被保留用来作为多播地址的，在 IPv4 中，该范围在 224.0.0.0 和 239.255.255.255 之间，但其中的某些地址被保留。

从 224.0.0.0 到 224.0.0.255 之间的地址被保留用于本地用途(比如管理和维护任务)，从 239.0.0.0 到 239.255.255.255 之间的地址同样也是被保留用于管理任务。

14.7.1　接收多播消息

要连接像 230.1.2.3 这样的多播地址，可以编写如下所示的代码：

```
var server = require('dgram').createSocket('udp4');

server.on('message', function(message, rinfo) {
  console.log('server got message: ' + message + ' from ' + rinfo.address
+
    ':' + rinfo.port);
});

server.bind(4000);
server.addMembership('230.1.2.3');
```

第 9 行告诉内核 UDP 套接字应该接收来自多播地址 230.1.2.3 的多播消息，当在服务器上调用 addMembership 方法时，可以将监听接口作为可选的第二个参数传递给该方法，如果这个参数被忽略，Node 就会尝试在每个网络接口上进行监听。

然后可以用新创建的 UDP 客户端来测试这个服务器，如下所示：

```
$ ./udp_client.js 230.1.2.3 4000
```

输入的每一行文本都会出现在服务器的控制台日志中。

14.7.2　发送多播消息

要发送多播消息，只需要指定多播地址即可，如下所示：

```
var dgram = require('dgram');
var client = dgram.createSocket('udp4');

var message = new Buffer('this is a multicast message');

client.bind(4000);
client.setMulticastTTL(10);
client.send(message, 0, message.length, 4000, '230.1.2.3');
client.close();
```

回忆一下，在前面你曾设置多播存活时间(Time to Live, TTL)为 10，这是一个任意值，TTL 通知网络数据报在被抛弃之前可以经过多少周转(路由器)，每经过一次周转，存活时间计数器就递减 1，当为 0 时，数据报就会被抛弃。

14.7.3 理解数据报最大容量

如果应用程序允许数据丢失和无序，UDP 就是一种轻量级的优秀解决方案，但是应该清楚一些附加限制。

数据报的最大容量取决于传输它的网络，UDP 头最多允许 65 535 字节的数据，但是比如说如果数据包是在一个以太网上传输，而该以太网的最大传输单位(Maximum Transmission Unit, MTU)为 1 500 字节，那么它也会将数据报的最大容量限制为 1 500 字节。另外，一些路由器会尝试将大的 UDP 数据包分割成 512 字节的块，所以如果希望数据报避免被分割，就应该将数据报的容量限制为 512。

14.8 本章小结

UDP 是一个轻量级的传输协议，如果可以放松可靠性的限制，它能很好地为你提供服务。因为它不是面向连接的，所以它比 TCP 的负载要小，并且可以只用一个网络数据包进行多播和广播。

如果想发送或者接收 UDP 消息，首先必须创建一个套接字，通过该套接字可以将消息发送给其他 UDP 终端。如果要监听传入的消息，就必须将套接字绑定到一个 UDP 端口上，然后等待 message 事件。

此外，通过将一个多播地址附加到套接字上，还可以监听多播消息。

第15章

用 TLS/SSL 保证服务器的安全性

本章提要：

- 理解公钥基础结构是如何工作的
- 创建 TLS 服务器
- 连接 TLS 服务器
- 验证服务器和客户端证书
- 安全地收发数据

传输层安全(Transport Layer Security, TLS)和安全套接字层(Secure Socket Layer, SSL)允许客户端/服务器应用程序以阻止窃听(其他人看到你的消息)和篡改(别人修改你的消息)的方式通过网络进行通信。TLS/SSL 会在传输层上对网络连接进行加密，这使得隐私和消息都能得到验证。

TLS 是基于 Netscape 公司开发的早期 SSL 规范，实际上，TLS 1.0 也被称为 SSL 3.1，而最新版的 TLS(TLS 1.2)也被称为 SSL 3.3。本章将用 TLS 代替遭轻视的 SSL 命名法。

15.1 理解私钥和公钥

公钥加密是指需要两个独立密钥的加密系统，一个密钥用来加密明文，另一个密钥解密已加密的消息。其中一个密钥是公有的，另一个是私有的，如果明文使用公钥加密，那么只能用私钥对其进行解密，这可以使公钥拥有者和私钥拥有者进行私有通信。如果明文是用私钥加密的，那么公钥就可以对其进行解密，在这种情况下，系统会验证私钥拥有者在文档上的签名。

公钥证书(也被称作数字证书)使用数字签名绑定公钥和身份信息——个人或者是组

织。文档会由认证中心进行签名，认证中心会完成验证匹配公钥的身份信息的过程。认证中心会响应其他证书，无论这个证书是自签名的还是由其他认证中心签名的，这样就会形成一个授权链，这个授权链是公钥基础结构的一部分。

每台计算机都会定义一组根认证中心，在默认情况下用它们来验证认证和认证链。在使用 Node 的 TLS 库时，可以使用这些默认的认证中心或者自定义的认证中心。

15.1.1　产生私钥

注意：Node 中的 TLS 是基于 OpenSSL 库实现的，可能你已经安装了这个库，为了检查安装与否，需要打开一个终端并尝试使用 openssl 命令。如果 OpenSSL 库没有安装，就要查找 openssl 包并安装。

TLS 依赖于公钥/私钥基础结构，在这个基础结构中，每个客户端和服务器都必须有一个私钥对消息签名。openssl 实用程序可以在命令行上创建私钥，如下所示：

```
$ openssl genrsa -out my_key.pem 1024
```

上述命令创建了一个文件 my_key.pem，也即私钥。

15.1.2　产生公钥

涉及 TLS 的服务器和客户端在彼此验证时必须具有一个证书，证书是由认证中心签名或者自签名的公钥。获取证书的第一步是创建一个证书签发请求(Certificate Signing Request, CSR)文件，如下所示：

```
$ openssl req -new -key my_key.pem -out my_csr.pem
```

上述命令会创建一个 CSR 文件 my_csr.pem，但首先你必须回答几个由 openssl 提出的身份认证问题。

为了用 CSR 创建自签名的证书，可以如下所示：

```
$ openssl x509 -req -in my_csr.pem -signkey my_key.pem -out my_cert.pem
```

上述命令会在私钥和认证请求的基础上创建一个自签名文件 my_cert.pem。另一种方法是将 CSR 发送到认证中心进行签名，但是对于研究本章中的示例而言，自签名认证已经足够了。

15.2　构建 TLS 服务器

TLS 服务器是 net.Server 的子集，在第 10 章“构建 TCP 服务器”中我已经介绍过

net.Server。能用 net.Server 创建什么，就能用 TLS 创建什么。但是，在使用 TLS 服务器时，使用的是安全的连接。

15.2.1　初始化服务器

初始化 TLS 服务器比初始化普通的 TCP 服务器要稍微复杂一些，因为必须传入服务器的私钥和证书文件：

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

var serverOptions = {
  key:  fs.readFileSync('./my_key.pem'),
  cert: fs.readFileSync('./my_certificate.pem')
};

var server = tls.createServer(serverOptions);
```

在上面的代码中，用到了同步版本的 fs.readFile，它会将文件全部读入内存，来获取文件系统上的密钥和证书。

注意：在上面的代码中用到了 fs.readFileSync，这是一个同步函数。这不会阻塞事件循环吗？不是，它只是在 Node 的初始化阶段执行，此时事件循环还没有开始。只要不是在事件处理程序中使用阻塞函数，就不会阻塞事件循环。

但是等等，如果这是一个写入数据的模块，而某些人在回调函数中请求该模块会怎么样？不应该在回调函数内部请求模块，这些模块会进行同步的文件系统访问，这样就会阻塞事件循环。

除了 key 选项和 cert 选项外，tls.createServer 还可以接受如下所示的选项：

- requestCert——如果为 true，服务器就会请求连接到其上的客户端的证书，并且会尝试对证书进行验证。默认值是 false。
- rejectUnauthorized——如果为 true，服务器会使用一个认证中心的列表来拒绝任何未经授权的连接。该选项只在 requestCert 为 true 时起作用。默认值是 false。

15.2.2　监听连接

与简单的 TCP 服务器一样，可以将 TLS 服务器绑定到 TCP 端口，如下所示：

```
var port = 4001;
server.listen(port);
```

当有新的连接到达时，服务器会发射 secureConnection 事件，并将其传入套接字来注册回调函数：

```
function connectionListener(stream) {
  console.log('got a new connection');
}

server.on('secureConnection', connectionListener);
```

15.2.3 从客户端读取数据

当有客户端连接时，服务器会发射 secureConnection 事件，并将 tls.CleartextStream 类的实例传入回调函数。与 net.Socket 对象很相似，tls.CleartextStream 对象实现了一个双向流接口，这意味着所有可读流、可写流以及流的事件都可为你所用，只需要绑定 data 事件来获取客户端发送的未加密数据即可，如下所示：

```
function secureConnectionListener(clientStream) {
  clientStream.on('data', function(data) {
    console.log('got some data from the client:', data);
  });
}
server.on('secureConnection', secureConnectionListener);
```

15.2.4 向客户端发送数据

还可以使用 CleartextStream 对象向连接到服务器上的客户端发送数据，如下所示：

```
server.on('secureConnection', function(clientStream) {
  clientStream.write('Hey Hello!\n');
});
```

15.2.5 终止连接

只要调用客户端流上的.end()函数就可以终止安全连接，例如，如果客户端输入了“quit”，就可以终止连接，如下所示：

```
server.on('secureConnection', function(clientStream) {
  clientStream.on('data', function(data) {
    if (data.toString().trim().toLowerCase() === 'quit') {
      clientStream.end('Bye bye!');
    }
  });
});
```

与 TCP 下的情况一样，可以将一个字符串或者缓冲区传入 clientStream.end 函数，它们将在连接关闭前最后被发送到客户端。

15.3 构建 TLS 客户端

要创建到服务器的连接，首先必须发布密钥和证书，如果没有，请按照本章开始的讲

解创建它们。

15.3.1　初始化客户端

必须用几个选项来初始化客户端连接，首先必须获得这些选项。事先应该已经创建了私钥和数字证书，并将它们存入了本地文件，可以用 fs.readFile 的本地版本 fs.readFileSync 来获取私钥和数字证书。

```
var fs = require('fs');
var options = {
  key: fs.readFileSync('/path/to/my/private_key.pem'),
  cert: fs.readFileSync('/path/to/my/certificate.pem')
};
```

15.3.2　连接服务器

拥有了私钥和数字证书之后，就可以使用 tls.connect 函数连接 TLS 服务器：

```
var tls = require('tls');
var host = 'localhost';
var port = 4001;

var client = tls.connect(port, host, options, function() {
  console.log('connected');
});
```

15.3.3　验证服务器证书

私钥和公钥允许你在客户端和服务器之间建了一条安全的信道，也就是说该信道可以免受监听和消息篡改。然而，在建立连接之后，你也许还想确认所连接的确实是所希望连接的服务器，为此可以查看服务器证书。

为了能够查看服务器证书，需要学习一些未曾见过的 TLS 连接选项，其中之一就 ca，它是授权中心的缩写。如果忽略该选项，会使用默认的一组根证书。

注意：除非内部的认证中心为你的自定义操作颁布了证书，否则就要忽略该选项并使用默认的证书。

如果想用自定义的证书来改写默认的证书，必须传入一个包含所有认证中心证书的缓冲区或者字符串数组。

默认的证书集合被用来对服务器进行认证，服务器会发送它的身份信息，而 TLS 客户端会检验身份信息的签名，看其是否与可信任的认证中心的签名相符。

如果证书得到认可，CleartextStream 实例(将其赋值给 client 变量)会将属性 authorized 设置为 true。如果没有得到认可，流会将 authorized 属性设置为 false，并且流的 authorizationError

属性值会包含出错的原因。如果根据根认证中心，服务器证书没有得到授权，就可以拒绝连接，如下所示：

```
var client = tls.connect(port, host, options, function() {
  console.log('authorized: ' + client.authorized);
  if (! client.authorized) {
    console.log('client denied access:', client.authorizationError);
  } else {
    // ...
  }
});
```

15.3.4 向服务器发送数据

回忆一下，tls.connect 函数返回的对象是 tls.CleartextStream 的一个实例，它也是一个可写流。这意味着可以使用 write()函数向服务器发送数据：

```
var client = tls.connect(port, host, options, function() {
  console.log('connected');
  client.write('Hey, hello!');
});
```

15.3.5 从服务器读取数据

一旦连接到服务器上，并且服务器已经开始发送数据，就可以得到 CleartextStream 对象发射的“data”事件。

```
var client = tls.connect(port, host, options, function() {
  client.on('data', function(data) {
    console.log('got some data from the server:', data);
  });
});
```

15.3.6 终止连接

基于应用程序以及服务器与客户端之间达成一致的协议，客户端也许在不再需要连接时关闭它。

由于只是关闭了连接的终端，因此仍然可以继续从服务器获得消息。

```
var client = tls.connect(port, host, options, function() {
  client.end('Bye bye!');
});
```

15.4 创建几个示例

是时候将前述内容整合一下了，现在要创建一个 TLS 聊天服务器和客户端。必须为客户端和服务器分别创建目录，以便可以轻松地分开脚本、密钥和证书文件。

15.4.1　创建 TLS 聊天服务器

创建一个项目目录 chat_server，并将密钥和证书文件放入其中，首先创建服务器密钥：

```
$ openssl genrsa -out server_key.pem 1024
```

然后创建证书请求：

```
$ openssl req -new -key server_key.pem -out server_csr.pem
```

为了用 CSR 创建自签名的证书，可以这样做，如下所示：

```
$ openssl x509 -req -in server_csr.pem -signkey server_key.pem \
-out server_cert.pem
```

既然 chat_server 目录中已经有了 server_cert.pem 文件，那么可以将服务器脚本存入文件 server.js，见程序清单 15-1。

程序清单 15-1：TLS 聊天服务器

```
var tls = require('tls');
var fs  = require('fs');
var port = 4001;

var clients = [];

var options = {
  key:  fs.readFileSync('server_key.pem'),
  cert: fs.readFileSync('server_cert.pem')
};

function distribute(from, data) {
  var socket = from.socket;
  clients.forEach(function(client) {
    if (client !== from) {
      client.write(socket.remoteAddress + ':' + socket.remotePort +
        ' said: ' + data);
    }
  });
}

var server = tls.createServer(options, function(client) {

  clients.push(client);

  client.on('data', function(data) {
    distribute(client, data);
  });

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

```
    clients.splice(clients.indexOf(client), 1);
  });

});

server.listen(port, function() {
  console.log('listening on port', server.address().port);
});
```

本例中的服务器会在一个模块全局数组中维护一个包含所有连接到其上的客户端列表，这个数组存储在变量 clients 中。当有客户端进行连接时，就会被添加到该数组中。当客户端发送数据时，数据都会被发送到所有其他的客户端，并且在所发数据之前会添加源 IP 地址和 TCP 端口号作为前缀。同样，当一个客户端断开连接时，就会从列表中删除。

现在只需要运行脚本：

```
$ node server.js
```

15.4.2　创建 TLS 命令行聊天客户端

在创建了 cht_client 目录之后，就可以开始创建客户端密钥和证书：

```
$ openssl genrsa -out client_key.pem 1024
$ openssl req -new -key client_key.pem -out client_csr.pem
$ openssl x509 -req -in client_csr.pem -signkey client_key.pem -out
  client_cert.pem
```

用客户端密钥和证书文件就可以创建客户端代码，并将代码存入文件 client.js，如程序清单 15-2 所示。

程序清单 15-2：TLS 聊天客户端

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

var port = 4001;
var host = '0.0.0.0';

var options = {
  key:  fs.readFileSync('client_key.pem'),
  cert: fs.readFileSync('client_cert.pem')
};

process.stdin.resume();

var client = tls.connect(port, host, options, function() {
  console.log('connected');
  process.stdin.pipe(client, {end: false});
  client.pipe(process.stdout);
});
```

这个客户端非常简单——它连接到本地主机上的 TLS 服务器，然后将进程的标准输入传入服务器，也即将输入的每一行传入服务器。它还会将服务器流传入到一个标准输出，它会输出服务器发送的所有字符。

可以通过在多个窗口中启动客户端来进行测试：

```
$ node client.js
```

你会看到在某个客户端窗口中输入的内容也会显示在所有其他客户端上。

15.4.3　验证客户端证书

服务器还有一件事没做，就是验证客户端的证书。如果打开 requestCert 选项，会看到客户端流上的 authorized 选项被设置为 false：

```
var options = {
  key:  fs.readFileSync('server_key.pem'),
  cert: fs.readFileSync('server_cert.pem'),
  requestCert: true
};

var server = tls.createServer(options, function(client) {
  console.log('client.authorized:', client.authorized);
  // ...
});
```

如果对服务器进行上述修改并运行客户端，会看到服务器输出如下所示的信息：

```
client.authorized: false
```

通过将 rejectUnauthorized 设置为 true，还可以决定不接受任何未经授权的客户端：

```
var options = {
  key:  fs.readFileSync('server_key.pem'),
  cert: fs.readFileSync('server_cert.pem'),
  requestCert: true,
  rejectUnauthorized: true
};
```

上述代码保证未通过认证链的客户端不会得到服务，如果尝试一下，你会发现这样的客户端无法进入服务器回调函数。

上述情况会发生是因为客户端证书是自签名的，而服务器使用的是默认的认证中心，默认的认证中心无法验证客户端上自签名的证书。

可以在服务器上耍点小把戏来测试一下这个特性——复制客户端证书并将其当作认证中心：

```
$ cd chat_server
$ cp ../chat_client/client_cert.pem fake_ca.pem
```

然后改写服务器上的认证中心选项：

```
var options = {
  key: fs.readFileSync('server_key.pem'),
  cert: fs.readFileSync('server_cert.pem'),
  ca: [fs.readFileSync('fake_ca.pem')],
  requestCert: true
};
```

此后如果启动服务器并将其与某个客户端进行连接，将会看到客户端已被授权：

```
client.authorized: true
```

15.5　本章小结

TLS 在 TCP 之上提供了一个加密的通信流，并且使用公钥/私钥对客户端和服务器进行验证。要进行通信，必须要有私钥和签名证书。证书可以是自签名的，如果服务器或客户端需要的话，也可以由认证中心对证书进行签名。

TLS 服务器和客户端遵循 net 模块中已有的模式，但是具有一些扩展，即允许指定重要的私钥和证书，私钥和证书允许你进行一些设置，决定是否进行验证。

第16章

用 HTTPS 保证 HTTP 服务器的安全性

本章提要：

- 创建 HTTPS 服务器
- 创建面向 HTTPS 服务器的请求
- 验证客户端和服务器证书

HTTPS 为标准的 HTTP 协议增加了 TLS 的安全性，在 Node 中 HTTPS 被实现为一个有别于 HTTP 的独立模块，除了一些细微的差别外，HTTPS API 和 HTTP API 十分相似。

https Node 核心模块扩展了核心 http 模块，并将 tls 模块作为传输机制。例如，https.Server 伪类继承自 http.Server 伪类，改写了对应的 Agent 类中创建连接的方法，它用实例化 TLS 连接来代替实例化普通的 TCP 连接。

16.1 构建安全的 HTTP 服务器

在本节中你将创建一个 HTTP 服务器，该服务器通过一个安全加密的信道与客户端交互，创建的 HTTP 服务器可以向客户端提供自认证，也可以对客户端身份进行认证。

首先，必须创建服务器私钥和自签名证书，这和在第 15 章中所做的相同：

```
$ openssl genrsa -out server_key.pem 1024
$ openssl req -new -key server_key.pem -out server_csr.pem
$ openssl x509 -req -in server_csr.pem -signkey server_key.pem
        -out server_cert.pem
```

第二步是提出一些问题——只要愿意就可以回答。

16.1.1 设置服务器选项

为了创建服务器，可以这样做，如下所示：

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

var options = {
  key: fs.readFileSync('server_key.pem'),
  cert: fs.readFileSync('server_cert.pem')
};

var server = https.createServer(options, function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!');
});
```

https.createServer 的第一个参数是一个 options 对象，这个对象和 TLS 模块很相似，也提供私钥和证书字符串。正如在第 15 章“用 TLS/SSL 保证 TCP 服务器的安全性”所做的那样，要使用 fs.readFileSync 函数从文件系统加载私钥和证书，顾名思义，fs.readFileSync 函数会同步加载文件。

注意：正如在第 15 章中所提到的，同步加载私钥和证书不会产生阻塞事件循环的危险，因为它是在模块加载阶段完成的，而模块加载阶段发生在 Node 的事件循环启动之前。

16.1.2 监听连接

然后可以——和原生的 HTTP 服务器模块一样——注册服务器，让其监听某个特定端口：

```
var port = 4001;
server.listen(port);
```

还可以指定希望监听哪个接口：

```
port = 4001;
var address = '192.168.1.100';
server.listen(port, address);
```

如果没有指定地址，那么服务器就会监听所有网络接口。

listen 命令是异步的，这意味着服务器不会在接收该命令后立刻监听连接。但是可以提供一个回调函数，该回调函数在服务器开始监听时就会被调用，可以将这个回调函数传入 server.listen 函数，作为其第二个或者第三个参数(不管是否指定了一个网络接口绑定到其上)：

```
server.listen(port, address, function() {
  console.log('Server is listening on port', server.address().port);
});
```

16.1.3　验证 HTTPS 客户端证书

如果正在创建 HTTPS 服务器，你的主要关注点也许是服务器与客户端之间的信道是否被加密。也许需要每个客户端都能用认证的方式来识别服务器。也许需要服务器验证客户端的真实性。最后一个例子不是最常见的用例，但是它在希望构建具有高安全性的计算机到计算机的通信方案时非常有用，尤其在向客户端发布身份证书时(类似于一些家庭银行服务所做的那样)十分有用。

可以通过指定服务器的一些启动选项来让其验证客户端的真实性：

```
var options = {
  key: fs.readFileSync('server_key.pem'),
  cert: fs.readFileSync('server_cert.pem'),
  requestCert: true,
  rejectUnauthorized: true
};
```

现在用到了两个新选项—— requestCert 和 rejectUnauthorized—— 两个选项被设置为 true。第一个选项要求客户端发送证书，第二个选项要求服务器拒绝那些未通过认证中心链验证的证书。

注意：要了解更多关于认证中心的信息，请参见第 15 章“用 TLS/SSL 保证 TCP 服务器的安全性”。

此外，如果忽略了 rejectUnauthorized 选项，仍然可以通过查看客户端流上的 authorized 属性来确定客户端是否可信：

```
var server = https.createServer(options, function(req, res) {
  console.log('authorized:', req.socket.authorized);
});
```

也可以获取客户端证书，但是注意只能在 socket.authorized 为 true 时才能相信证书信息的真实性。获取客户端证书信息的方法如下所示：

```
var server = https.createServer(options, function(req, res) {
  console.log('authorized:', req.socket.authorized);
  console.log('client certificate:', req.socket.getPeerCertificate());
});
```

通过查询流是否可信(req.socket.authorized)以及获取网络端点的证书(req.socket.getPeer-Certificate())，可以识别和证实客户端连接的可信性。

16.2 创建 HTTPS 客户端

在 Node 中，创建 HTTPS 请求与创建 HTTP 请求非常相似，实际上很可能就是相同的，因为并不需要传入客户端私钥或者客户端证书，除非服务器需要它们。

16.2.1 初始化客户端

可以针对 HTTPS 服务器创建 HTTPS 请求，如下所示：

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

var options = {
  host: '0.0.0.0',
  port: 4001,
  method: 'GET',
  path: '/'
};
```

同初始化 HTTP 客户端的代码很相似，需要创建一个包含请求选项的对象，该对象应该包含主机、端口和路径的值。如果想创建一个使用 GET 之外的方法的请求，还可以指定方法。

16.2.2 创建请求

必须将最后几个指定主机、端口、方法和路径的选项传递到 https 模块中的 request 函数中，request 函数会接受 options 对象和一个回调函数，并返回一个 http.ClientReques 实例。

```
var request = https.request(options, function(response) {
  console.log('response.statusCode:', response.statusCode);

  response.on('data', function(data) {
    console.log('got some data back from the server:', data);
  });
});

request.write('Hey!\n');
request.end();
```

上面创建的客户端请求是一个可写流，可以向其中写入数据，或者将一个可读流传入其中，然后就应该结束请求以便让服务器做出响应。

一旦接收到来自服务器的响应——就会触发 request 对象上的 response 事件——回调函数就会获得 response 对象。如果对响应主体的内容感兴趣，因为 response 对象是一个可读

流，所以这意味着可以将其传入一个可写流或者只是为任意 data 事件注册一个回调函数。

16.2.3　验证 HTTPS 服务器证书

除了安全信道外，客户端对服务器证书的验证是在网络上使用 HTTPS 最常见的理由。大多数网络浏览器希望确保与它们交互的服务器与 URL 栏中显示的一致。通过使用内置的 Node HTTPS 客户端就可以执行这种验证。

当进行请求时，在获得了 response 事件以及 response 对象后，就可以查看 response 对象。response 对象在其 socket 属性内有一个 stream 对象，由于它是 TLS 流，因此可以在其上调用 getPeerCertificate 函数来获取服务器证书，如下所示：

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

var options = {
  host: 'google.com',
  method: 'GET',
  path: '/'
};

var req = https.request(options, function(res) {
  console.log('res.socket.authorized:', res.socket.authori
  console.log('peer certificate:');
  console.log(res.socket.getPeerCertificate());
});

req.end();
```

如果运行上面这个小型 HTTPS 客户端，就会获得与如下所示类似的输出信息：

```
res.socket.authorized: true
peer certificate:
{ subject:
   { C: 'US',
     ST: 'California',
     L: 'Mountain View',
     O: 'Google Inc',
     CN: '*.google.com' },
  issuer: { C: 'US', O: 'Google Inc', CN: 'Google Internet Authority' },
  subjectaltname: 'DNS:*.google.com, DNS:google.com,
    DNS:*.youtube.com, ...',
  modulus:
    'CAB66CCEFADF0EF63AC7B5DC1647EB9F31E196929391D28DAB0747C42276...',
  exponent: '10001',
  valid_from: 'Jul 26 00:34:27 2012 GMT',
  valid_to: 'Jun  7 19:43:27 2013 GMT',
  fingerprint:
    '56:7E:72:30:90:DB:16:E1:A9:65:8F:5A:7B:18:59:52:01:F7:64:D7',
  ext_key_usage: [ '1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2' ] }
```

上面这段信息中的验证数据以及 res.socket.authorized 为 true 的事实，可以让你有把握断定主题名与主机名或者作为请求目标的域相匹配。

16.3　本章小结

HTTPS 是保证信道安全和在通信节点间提供认证框架的一种方法。可以用 HTTPS 对客户端或者服务器进行认证，Node 的 https 模块拓展了 HTTP 模块的功能，它使用 TLS 模块提供位于 HTTP 之上的底层传输。

可以通过进入底层的 TLS 套接字以及抽取节点证书识别数据来验证服务器和客户端。

第Ⅳ部分

构建与调试模块及应用程序

- 第 17 章：测试模块及应用程序
- 第 18 章：调试模块及应用程序
- 第 19 章：控制回调流程

第 17 章

测试模块及应用程序

本章提要：

- 使用 node-tap 进行测试
- 使用 Node 的 assert 模块
- 测试一个示例模块

为了最小化用户在使用软件时遇到的错误数量，必须测试所编写的代码，可以用人工的方式进行测试，但是最好的办法还是实现测试过程的自动化。

自动化测试是提供代码质量的最重要实践之一，目的是创建覆盖全部代码的一系列测试，包括软件运行的主要情形和边缘情形。一些准则，例如测试驱动开发甚至鼓励编写测试并让其遭遇失败，然后再实现可让测试成功运行的代码。

无论你的方法是什么，编写自动化测试是一项不错的投资，本章展示如何在 Node 的模块和程序中进行自动化测试。

17.1 应用测试运行工具

要创建自动化测试，首先需要一种方式来定义测试，然后需要一个工具来编译和运行这些测试，并输出结果。

有很多面向 Node.js 和 JavaScript 的工具，本章主要选取了 node-tap 工具，选择它基于几点原因：简单，很容易就支持异步测试，并且输出格式是基于某个标准的。

将 node-tap 包含进 package.json 文件的 dev-dependencies 节，就可以安装它：

```
{
  "name": "MyApp",
  "version": "0.1.0",
  "devDependencies": {
    "tap": "*"
  }
}
```

然后执行以下命令：

```
$ npm install
```

当 NPM 下载和安装 node-tap 以及它的依赖模块后，就可以开始使用它。

注意：如果在 package.json 文件的 devDependencies 属性中声明了一个依赖模块，实际就是通知程序如果它是在产品模式下运行，就不要这个依赖模块。

devDependencies 属性是存储所有只用于开发和测试应用程序的依赖模块的好地方。

当在产品模式下安装应用程序时，可以使用-production 选项跳过安装 devDependencies 节中所列的模块：

```
$ npm install -production
```

17.1.1　编写测试

要创建测试，首先需要建立一个目录来放置测试脚本，一般说来该目录被命名为 tests，每个要测试的模块都包含一个文件。

在该目录中创建你的第一个测试脚本，为达到这一目的，要像下面这样有赖于 tap 模块，并获得 test 构造函数：

```
var test = require('tap').test;
```

然后就可以定义第一个测试了，此时需要提供一个测试名和一个包含测试代码的函数：

```
test("addition of 2 and 2 works", function(t) {
  t.equal(2+2, 4, "2+2 should be 4");
  t.end();
});
```

在上面的代码中，声明了一个测试 addition of 2 and 2 works，然后向其传递一个包含测试代码的函数，该函数接收一个参数 t，该参数包含面向测试的控制对象，利用该对象

可以测试是否符合你的期望，并结束当前测试。显式地结束测试非常重要，Node-tap 支持异步测试，这意味着如果进行 IO 操作，测试在测试函数返回之后仍旧会运行。只有在测试上调用 end 方法时，Node-tap 才会结束测试。

17.1.2　运行测试

保存在上一节中创建的文件，将其命名为 my_test.js，可以直接调用测试脚本来执行测试：

```
$ node my_test
```

然后应该获得如下所示的输出信息：

```
# addition of 2 and 2 works
ok 1 2+2 should be 4

1..1
# tests 1
# pass  1

# ok
```

17.2　使用断言测试模块

要创建可用的测试，需要一种验证结果与期望是否匹配的方法，因此，前面的例子用到了 node-tap 内置的断言函数——你将在后面学到——但是目前可以使用 Node 内置的 assert 模块。

17.2.1　使用断言模块

Node 在其自身的测试中用到了内置 assert 模块，此外，它还向外暴露了 assert 模块，这意味着可以用它在自己的测试中验证假设。

为了使用 assert 模块，需要将该模块导入到测试脚本中：

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

现在就可以使用断言函数了，每一个断言函数都会设置一个期望条件，如果期望条件不满足，会抛出一个异常。

首先，可以测试值是否为真：

```
var a = true;
assert(a);
```

注意：在 JavaScript 中，可以成功用于 if 语句中的值就为真，真值通常由所有可能的值定义，以下这些假值除外：

- false
- null
- undefined
- 空字符串“”
- 数字 0
- 数字 NaN

在上面这个断言函数以及其他断言函数中，还可以提供一个消息，作为函数的最后一个参数，以此来在测试失败时提供更多的显示信息：

```
assert(a, 'a should be truthy');
```

还可以使用别名来判断值是否为真：

```
assert.ok(a, 'a should be truthy');
```

此外，assert 模块也可以按两种方式测试相等性：浅和深，浅相等性只是对提供的两个对象进行测试，而不会测试这两个对象中内嵌对象的相等性。相反，深相等性递归测试所有嵌套的对象。浅相等性通常用来比较标量对象，例如字符串和数字，而深相等性则通常用于比较两个对象。

可以按照如下所示的方法测试浅相等性：

```
assert.equal(a, 10, 'a should be 10');
```

这个操作符和 JavaScript 中的等于操作符(==)是等效的，这意味着下面的断言是有效的：

```
assert.equal('10', 10);
```

还可以使用它的对立面——不相等断言——它和 JavaScript 中的强制不相等操作符是等效的，如下所示的断言会失败：

```
assert.notEqual('10', 10);
```

与 JavaScript 中的严格相等操作符(===)类似，assert 模块同样也支持严格相等语义，下面的断言会失败：

```
assert.strictEqual('10', 10, 'string 10 should  equal to number 10');
```

而下面的断言是有效的：

```
assert.notStrictEqual('10', 10, 'string 10 should not be equal to number
10');
```

到目前为止，已经测试了浅相等性，在浅相等性中，如果想测试两个对象，这两个对象的引用必须指向同一个对象，相等性断言才会有效：

```
assert.equal({a:1}, {a:1});
```

出于如下所示的原因，前述断言会失败：

```
AssertionError: {"a":1} == {"a":1}
```

上面的浅比较之所以失败是因为即使这两个对象看上去是相同的，但是实际上它们是两个不同的对象。

如果想比较对象的属性，就要用到深比较。

可以用 assert.deepEqual 函数来进行深比较，如下所示的断言是有效的：

```
var obj = {b:1, a:[1,2,3,4]};
assert.deepEqual(obj, {a:[1,2,3,4], b:1});
```

还可以用它来比较日期：

```
var a = new Date(),
    b = new Date();
assert.deepEqual(a, b);
```

从 Node 0.7 版开始，还可以用它来比较正则表达式，如果用的是 Node 0.7 或者更高版本，下面的断言是有效的：

```
assert.deepEqual(/a/ig, /a/gi);
```

deepEqual 函数验证的仅仅是对象的可枚举属性，这意味着如果两个对象具有不同的原型，但是它们的所有可枚举属性是相同的，它们就不会表现出不同。例如，如下所示的断言是有效的：

```
var EventEmitter = require('events').EventEmitter;
var ee = new EventEmitter();
assert.deepEqual(ee, {});
```

上述断言有效是因为下面的断言是有效的：

```
assert.strictEqual(Object.keys(ee).length, 0);
```

17.2.2　使用 Node-Tap 中的内置断言函数

node-tap 模块同样也提供了一组断言测试函数，用它们来代替定义在 Node 核心模块中的断言测试函数，就可以利用 node-tap 中内置的测试计数、错误报告以及合计报告。测试函数类似于 assert 模块，但值得单独查看以查找不同点。

可以用 t.ok()和 t.notOk 测试值是否为真：

```
test("truthyness of numbers", function(t) {
  t.ok(1, '1 should be truthy');
  t.notOk(0, '0 should not be truthy');
  t.end();
});
```

还可以用 t.equal 函数测试严格相等性：

```
test("sum works", function(t) {
  var a = 2 + 2;
  t.equal(a, 4, '2 + 2 should be equal to 4');
  t.notEqual(a, '4', '2 + 2 should not be equal to the string "4"');
  t.end();
});
```

同样，还可以用 t.equal 来测试浅相等性，如下所示的测试会失败：

```
test("object equality", function(t) {
  var a = {a:1};
  t.equal(a, {a:1});
  t.end();
});
```

在上面的情况下，应该使用等价性测试，如下所示测试会成功：

```
test("object equivalency", function(t) {
  var a = {a:1, b:2};
  t.equivalent(a, {b:2, a:1});
  t.end();
});
```

还可以测试对象的相似性，例如，你也许想判断一个对象是否包含一系列属性，然后对属性值做出断言，为此需要使用 t.similar 函数，如下所示：

```
test("object similarity", function(t) {
  a = {a:1, b:2};
  t.similar(a, {a:1});
  t.similar('abc', 'abc');
  t.similar(10, 10);
  t.end();
});
```

对象的相似性对于数字和字符串对象同样有效：

```
test("object similarity", function(t) {
  t.similar('abc', 'abc');
  t.similar(10, 10);
  t.end();
});
```

可以使用 t.type()函数测试对象类型，如下所示：

```
test("object type", function(t) {

  t.type(1, "number");

  t.type('abc', 'string');

  t.type({}, Object);

  var EventEmitter = require('events').EventEmitter;
  var ee = new EventEmitter();
  t.type(ee, EventEmitter);
  t.type(ee, Object);

  t.end();
});
```

可以从前面的例子中看到，你能通过向 t.type()断言函数传递一个构造函数来判断一个对象是否继承自某个构造函数。

17.3　测试异步模块

既然你已经知道了如何使用 node-tap 模块，就可以创建一个有趣的模块示例进行测试。为了让它显得更有趣些，该模块可能要执行一些 I/O 操作，所以你能进行一些异步测试。

该模块通过调用远程 HTTP 服务将两个整数相加，首先，创建一个 HTTP 服务器，让其接受两个整数，并将这两个整数相加，如程序清单 17-1 所示。

程序清单 17-1：将两个整数相加的 HTTP 服务器

```
var parse = require('url').parse;

require('http').createServer(function(req, res) {
  params = parse(req.url, true).query;
  var a = parseInt(params.a, 10);
  var b = parseInt(params.b, 10);
  var result = a + b;
  res.end(JSON.stringify(result));
}).listen(8080);
```

创建两个目录：一个命名为“client”，另一个命名为“server”。然后将程序清单 17-1 保存为文件 server/sum_server.js 并运行，如下所示：

```
$ node server/sum_server.js
```

以上操作就启动了整数相加的 HTTP 服务。

然后需要创建一个客户端模块来使用该服务，并向外暴露客户端 API。首先，要在“client”目录下的 package.json 清单中声明依赖模块，如程序清单 17-2 所示。

程序清单 17-2：客户端的 package.json 声明

```
{
  "name": "sum-client",
  "version": "1.0.0",
  "dependencies": {
    "request": "*"
  }
}
```

现在，需要安装依赖模块，在本例中只需要安装“request”模块：

```
$ npm install
request@2.10.0 ./node_modules/request
```

现在，将程序清单 17-3 中的代码复制到客户端目录中名为 sum_client.js 的文件里，这样就将客户端模块放置到客户端目录中的 index.js 文件中。

程序清单 17-3：求和客户端模块

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

function sum(a, b, callback) {
  var options = {
    uri: 'http://localhost:8080/',
    qs: {
      a: a,
      b: b
    }
  };

  request(options, function(err, res, body) {
    var result;
    if (err) {
      return callback(err);
    }
    try {
      result = JSON.parse(body);
    } catch(err) {
      return callback(err);
    }
    return callback(null, result);
  });
}

module.exports = sum;
```

该模块只是导出一个提供求和服务的函数。

现在，准备测试这个模块，创建一个与“client”和“Server”目录同级的“tests”目录。

然后，必须在该目录中创建一个 package.json 程序清单，在程序清单中列出“tap”作为开发依赖项：

```
{
  "name": "sum-tests",
  "version": "1.0.0",
  "devDependencies": {
    "tap": "*"
  }
}
```

进入“tests”目录，安装开发依赖项：

```
$ cd tests
$ npm install
tap@0.3.0 node_modules/tap
├── buffer-equal@0.0.0
├── deep-equal@0.0.0
├── mkdirp@0.3.3
├── slide@1.1.3
├── nopt@2.0.0 (abbrev@1.0.3)
├── difflet@0.2.1 (charm@0.0.8, traverse@0.6.3)
└── runforcover@0.0.2 (bunker@0.1.2)
$ cd ..
```

然后，将程序清单 17-4 中的内容存入文件 tests/sum.js。

程序清单 17-4：求和客户端测试

```
var sum = require('../client/sum_client');
var test = require('tap').test;
test('sums 1 and 2', function(t) {
  sum(1, 2, function(err, result) {
    t.notOk(err, 'no error');
    t.equal(result, 3, '1 + 2 should be equal to 3');
    t.end();
  });
});

test('sums 5 and 0', function(t) {
  sum(5, 0, function(err, result) {
    t.notOk(err, 'no error');
    t.equal(result, 5, '5 + 0 should be equal to 3');
    t.end();
  });
});

test('sums 5 and -2', function(t) {
  sum(5, -2, function(err, result) {
    t.notOk(err, 'no error');
```

```
  t.equal(result, 3, '5 + -2 should be equal to 3');
  t.end();
});
```

在测试文件中，首先导入了求和客户端模块，然后进行一系列测试，用不同的参数来调用求和客户端模块函数，观察是否有错误以及结果是否与期望值相符，最后结束测试。

确定求和服务器仍然在运行，执行如下所示的测试：

```
$ node tests/sum.js
```

然后将得到测试成功的输出信息：

```
...
1..6
# tests 6
# pass  6

# ok
```

17.4　本章小结

测试应用程序和模块对于最小化代码中错误来说不仅是最佳实践，也是必不可少的。在 Node 中除了别的模块以外，可以使用 node-tap 模块来进行测试。

为了测试程序中的期望，既可以使用 Node 的原生 assert 模块，也可以使用 node-tap 模块提供的函数。

第18章

调试模块及应用程序

本章提要：

- 格式化并输出变量
- 使用变量插值
- 使用 Node 调试器
- 使用可视化 Node 调试器

创建实用的软件不是一项微不足道的任务，将应用程序代码划分成小的模块并对每个模块进行测试会十分有帮助，但是仍然必须得找出问题。如果发现自己面临需要检查 Node 应用程序或模块代码内部运行的情况，几个工具会有所帮助。本章将介绍几个调试工具——console.log、Node 内置调试器以及 Node 检查器。

18.1 使用 console.log

Node 提供了一些不必显式声明就可以使用的全局对象，其中之一就是 console 对象，该对象可以让你输出格式化的字符串。

可供使用的最简单的调试工具就是 console.log，调用该函数会完成两项工作——将你的对象序列化并连接成一个字符串，并向标准输出流输出结果，可以像下面这样用它来检查对象。

```
var obj = {a: 1, b: 2};
console.log(obj);
```

最后一段代码打印出如下信息：

```
{ a: 1, b: 2 }
```

这个简单的例子传入了一个对象，该对象被转换成一个字面量字符串，可以传入任意类型的对象，console.log 将检查这些对象的属性和值。

注意：实际上 console.log 没有进行任何格式化工作，所有格式化都是由 util.format 函数完成的。无论你传入 console.log 的参数是什么，都会交给 util.format 处理，并且结果都会输出到标准输出流中，这意味着任何调用 console.log 的例子都可以直接应用 util.format。

console.log 函数只会检查对象的自有可枚举属性，也就是说处于原型链上的属性以及不可枚举的属性都无法显示。下面是一个例子，其中的代码在 Node 控制台上实例化了一个 EventEmitter 对象：

```
> var EventEmitter = require('events').EventEmitter;
undefined
> var emitter = new EventEmitter();
undefined
> console.log(emitter);
{}
undefined
> emitter.on('data', function() { });
{ _events: { data: [Function] } }
> console.log(emitter);
{ _events: { data: [Function] } }
undefined
```

在前面这个交互式的例子中可以看到，首次在 emitter 变量值上调用 console.log 时将打印一个空对象，这是因为 emitter 对象在构造时没有设置任何属性，在附加一个事件监听器后，它将获得一个名为_events 的属性，该属性会在紧接着的 console.log 调用中被打印出来。

console.log 函数也能以易理解的方式记录数组：

```
> var arr = [1, 2, 3];
undefined
> console.log(arr);
[ 1, 2, 3 ]
Undefined
```

console.log 函数也能通过 util.format 函数提供向字符串中插入数值的能力，可以像下面这样在字符串中指定以%为前缀的占位符：

```
> console.log('an object: %j and a number: %d', {a:1, b:2}, 0xfa);
an object: {"a":1,"b":2} and a number: 250
undefined
```

可以用%j 对一个对象进行 JSON 编码，用%d 输出一个数(整数或者浮点数)，用%s 输出另一个字符串。

如果向 console.log 提供在模板字符串中不具有占位符的额外参数，或者只提供不带有模板字符串的参数列表， console.log 会简单地对它们进行格式化和连接，并将每个参数用空格分开。

```
> console.log({a:1, b:2}, 0xfa);
{ a: 1, b: 2 } 250
undefined
> console.log('an object: %j and a number:', {a:1, b:2}, 0xfa);
an object: {"a":1,"b":2} and a number: 250
undefined
```

注意：console.log 函数只输出 util.format 的结果字符串，并将结果字符串附加一个换行符后推入到标准输出流中。然而，在 Node 中向进程的输出流中写入数据是一种阻塞操作，这意味着在你写入记录时会阻塞事件循环。取决于你的应用程序以及使用该函数的频率，(这样做)可能会削弱(程序的)响应性，这意味着应该在产品代码中避免使用 console.log。

18.2　使用 Node 内置调试器

使用 console.log 检查变量会很快变得难以驾驭，特别是在你尝试修正的问题非同一般时更是如此。在定位难以发现的问题时，也许不得不增加很多条记录指令，同时程序的输出将会变得冗长啰嗦。最糟糕的是当引入新的记录指令时，都必须运行程序并设法重现问题。

幸好 JavaScript V8 虚拟机导出了一个可受 Node 支持的调试接口，如果需要暂停执行来仔细检查程序，可以使用 Node 内置调试器，虽然它既简单又基础，但是足够帮助你不必安装其他工具就可以调试程序。

首先，创建一个带有错误的简单示例应用程序以便尝试调试器(见程序清单 18-1)。

程序清单 18-1：一个带有错误的应用程序

```
var a = 0;

function init() {
  a = 1;
}

function incr() {
```

```
  var a = a + 1;
}

init();
console.log('a before: %d', a);

incr();
console.log('a after: %d', a);
```

如果运行这个应用程序，会看到如下所示的输出：

```
a before: 1
a after: 1
```

可以看到变量 a 并没有如你所愿的那样递增，下一步要以调试模式启动 Node 来查看发生了什么。

```
$ node debug my_app.js
```

上面的代码片段以调试模式启动了 Node，同时也启动了调试器。这个操作会启动应用程序，但是会在主模块的第 1 行中断，现在应该能获得调试器的提示信息以及当前指令的位置：

```
< debugger listening on port 5858
connecting... ok
break in my_app.js:1
  1 var a = 0;
  2
  3 function init() {
```

可以用提示信息来列出所有命令：

```
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
```

现在可以继续下一条指令：

```
debug> next
```

这将继续执行程序并在同一作用域的下一条指令处中断执行：

```
break in my_app.js:11
  9 }
 10
 11 init();
 12 console.log('a before: %d', a);
 13
```

现在不用 next 命令继续执行到同一作用域的下一指令处，而是进入 init()函数调用中：

```
debug> step
break in my_app.js:4
  2
  3 function init() {
  4   a = 1;
  5 }
  6
```

现在会看到调用堆栈的状态像下面这样：

```
debug> backtrace
#0 my_app.js:4:3
#1 my_app.js:11:1
```

还可以像下面这样通过添加一个 watch 来监视一个变量：

```
debug> watch('a')
```

该命令会向监视列表增加一个监视，可以按如下方式来检查监视列表：

```
debug> watchers
  0: a = 0
```

在此处可以看到变量 a 在局部作用域中的值是 0。

现在可以进入下一条命令，如下所示：

```
debug> next
break in my_app.js:5
Watchers:
  0: a = 1

  3 function init() {
  4   a = 1;
  5 }
  6
  7 function incr() {
```

调试器指令执行了下一条命令并显示监视列表以及执行的下一行。可以看到变量 a 现在的值是 1。

现在从 init 函数的结尾处准备返回，可以用 out 命令跳至父作用域的下一指令处，或者——因为已经到达了当前作用域的结尾处——所以也可以使用 next 命令：

```
debug> next
break in my_app.js:12
Watchers:
  0: a = 1
10
```

```
11 init();
12 console.log('a before: %d', a);
13
14 incr();
```

要是想跳过第 12 行而直接进入 incr()函数调用的执行中该怎么办呢？为此，要在 my_app.js 文件的第 8 行设置一个断点：

```
debug> sb ('my_app.js', 8)
  7 function incr() {
* 8   var a = a + 1;
  9 }
 10
 11 init();
 12 console.log('a before: %d', a);
 13
 14 incr();
 15 console.log('a after: %d', a);
 16 });
```

可以看到在第 8 行有一个断点，并且当前指令在第 12 行，现在准备好迅速前进至断点处：

```
debug> cont
< a before: 1
break in my_app.js:8
Watchers:
  0: a = undefined

  6
  7 function incr() {
* 8   var a = a + 1;
  9 }
 10
```

通过检查调试器的输出，可以检测到应用程序中的问题，调试器是在说变量 a 的前一个值是 1 而新值未定义。这是因为在第 8 行中用到了关键字 var，它会创建一个局部变量来代替全局变量。既然你知道了做错了什么，就可以按下 Ctrl+D 快捷键退出调试器，并从第 8 行删除 var 关键字来修正该问题：

```
var a = 0;

function init() {
  a = 1;
}

function incr() {
```

```
  a = a + 1;
}

init();
console.log('a before: %d', a);

incr();
console.log('a after: %d', a);
```

再次运行应用程序：

```
$ node my_app.js
a before: 1
a after: 2
```

现在的输出和你希望的一致，错误被修复了。

18.3　使用 Node 检查器

另一个可能有帮助的调试工具是 Node 检查器，该工具通过浏览器将成熟的 Google Chrome 检查器加入到应用程序中，借此它提供了一个图形界面，而不仅仅是纯文本形式的调试器。

可以通过下面的命令以全局模式安装 Node 检查器：

```
$ npm install -g node-inspector
```

Node 检查器作为一个守护程序默认运行在 8080 端口，可以通过下面的命令启动它：

```
$ node-inspector &
```

这会向后台发送 node-inspector 进程。

下一步需要在 node 可执行文件上使用--debug-brk 或--debug 选项来启动应用程序，如下所示：

```
$ node --debug-brk my_app.js
```

--debug-brk 选项让应用程序在第 1 行中断，而--debug 选项只是启动了调试功能。

注意：当调试服务器时应该使用--debug，当调试其他脚本时也许想用--debug--brk 选项在第 1 行中断。

现在可以打开浏览器并转向 http://localhost:8080，应该可以得到如图 8-1 所示的屏幕信息。

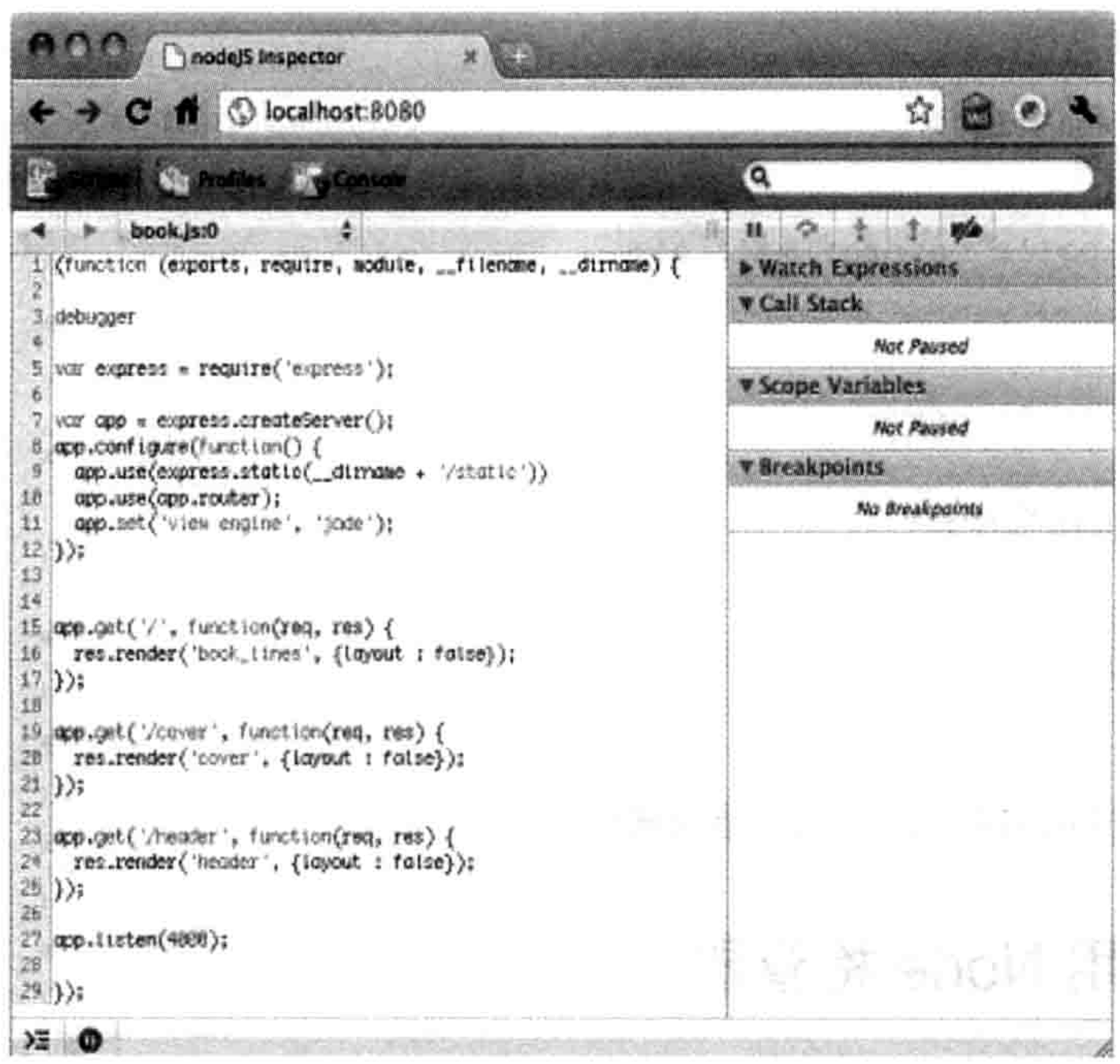

图 18-1　Node 检查器的启动屏幕

可以单击行号来设置或者取消设置断点。

当到达断点时，你的应用程序会暂停，同时 Node 检查器将显示上下文，如图 18-2 所示。

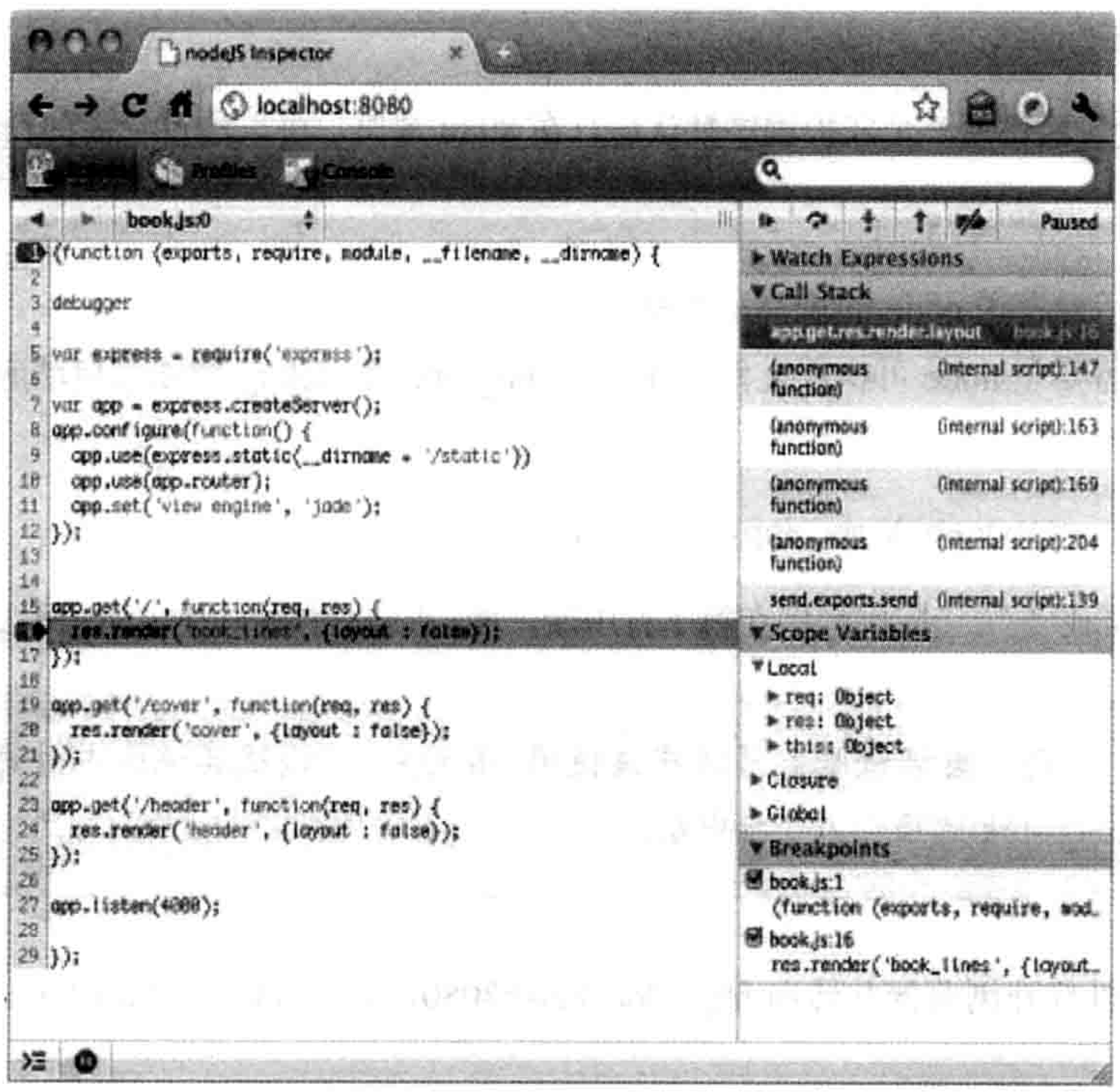

图 18-2　Node 检查器中到达断点的位置

可以在右边看到两个非常有趣的窗格：

- 调用堆栈，在该窗格中可以看到到达断点时是哪些函数被调用。
- 作用域变量，在该窗格中可以检查局部变量、全局变量以及闭包变量，闭包变量是在更高一级的函数作用域中定义的。

在这两个窗格的上方有几个按钮，可以用这些按钮控制被执行的指令，如图 18-3 到图 18-6 所示。

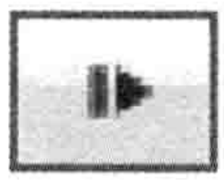

图 18-3　快速向前执行直到遇到一个断点

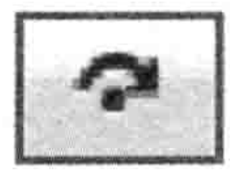

图 18-4　执行函数并在第一行停止

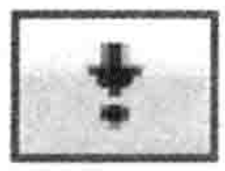

图 18-5　进入函数内部

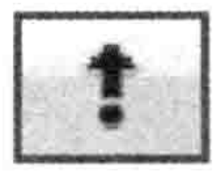

图 18-6　继续，当函数结束时返回到函数调用者

注意：Live Edit 描述了在调试时修改代码的过程，你只要双击一行代码并编辑它，修改代码并单击 Tab 键或者在编辑区域外单击，就修改了正在运行的代码。

但是，注意用 Live Edit 修改代码不能保存修改结果——使用它只能对快速修改进行测试。

18.4　本章小结

可以使用 console.log()输出应用程序的一些状态，以辅助调试。使用这个函数可以格式化并输出变量，以此暴露应用程序的状态。

如果有更为复杂的问题，可以使用 Node 的内置控制台调试器。如果愿意，也可以使用 Node 检查器，它是一个具有图形界面的调试器，它使用浏览器并提供了更为复杂的选项，Node 检查器也允许在调试时编辑代码。

第19章

控制回调流程

本章提要：

- 理解飞去来器效应
- 避免深度嵌套回调
- 创建异步流程控制例程
- 使用 async 控制回调流程
- 使用 async 进行异步迭代

当 Node 执行异步操作时，操作的结果并不是通过返回函数来反馈。相反，异步程序依赖于回调函数，回调函数通常作为参数传入。

如果需要离开进程并进行一些 I/O 操作，大多数时候需要指定一个在操作结束时被调用的回调函数。

如果任务涉及大量的 I/O 操作，要把这些回调函数的流程安排得井井有条就很棘手，也许还会导致一种被称作“回调汤”或“飞去来器效应”的模式。

19.1 理解飞去来器效应

“飞去来器”效应发生在一组回调按序执行时——当一个 I/O 操作结束时，下一个操作就开始。“飞去来器”的名称来源于代码中有很多嵌套回调函数时的形状，递增的左缩进以及左缩进的回退构成了一个类似飞去来器的形状。

为了说明飞去来器效应，需要创建一个脚本，向文件 a.txt 附加 0~10 个字节，并存入文件 b.txt。这两个文件都存在于当前目录下，程序清单 19-1 展示了一个飞去来器的例子。

程序清单 19-1：飞去来器效应演示

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

function append_some_a_to_b(callback) {
  fs.open(__dirname + '/a.txt', 'r', function(err, aFd) {

    if (err) {
      return callback(err);
    }

    var buffer = new Buffer(10);

    fs.read(aFd, buffer, 0, buffer.length, 0, function(err) {

      if (err) {
        return callback(err);
      }

      fs.close(aFd, function(err) {

        if (err) {
          return callback(err);
        }

        fs.open(__dirname + '/b.txt', 'a', function(err, bFd) {

          if (err) {
            return callback(err);
          }

          fs.fstat(bFd, function(err, bStats) {

            if (err) {
              return callback(err);
            }

            fs.write(bFd, buffer, 0, buffer.length, bStats.size,
                     function(err) {
              if (err) {
                return callback(err);
              }

              fs.close(bFd, callback);
            });
          });

        });
      });
    });
  });
```

```
  });
}

console.log('starting...');

append_some_a_to_b(function(err) {

  if (err) {
    throw err;
  }

  console.log('done');
});
```

注意 append_some_a_to_b 函数，该函数为本任务实现了回调流程，内联声明回调函数将导致深层嵌套代码，对于未经过训练的程序员，用这种方式编写的代码难以阅读。

更糟糕的是，如果需要在最终操作之前有条件地执行一些 I/O 操作，用这种方式构造的代码会造成大量重复。

注意：虽然以内联形式声明回调函数至少有一个优点，但是，如果将回调函数代码直接放在 I/O 调用之后，你更能感受到操作是串行进行的。

与编程中遇到的大多数情况一样，这并不是非此即彼的建议——只要不滥用嵌套的回调函数，它们有时也是说得过去的。

19.2　通过声明函数避免飞去来器效应

可以通过给函数命名，并且在同一作用域声明它来避免飞去来器效应，程序清单 19-2 展示了前面的例子，但是这一次，函数在同一作用域中进行了命名和声明。

程序清单 19-2：声明命名函数

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

function append_some_a_to_b(callback) {

  var aFd, bFd,
     buffer = new Buffer(10);

  function open_a() {
    fs.open(__dirname + '/a.txt', 'r', read_from_a);
  }
```

```
function read_from_a(err, fd) {

  if (err) {
    return callback(err);
  }

  aFd = fd;
  fs.read(aFd, buffer, 0, buffer.length, 0, close_a);
}

function close_a(err) {

  if (err) {
    return callback(err);
  }

  fs.close(aFd, open_b);
}

function open_b(err) {

  if (err) {
    return callback(err);
  }

  fs.open(__dirname + '/b.txt', 'a', stat_b);
}

function stat_b(err, fd) {

  if (err) {
    return callback(err);
  }

  bFd = fd;
  fs.fstat(bFd, write_b);
}

function write_b(err, bStats) {

  if (err) {
    return callback(err);
  }

  fs.write(bFd, buffer, 0, buffer.length, bStats.size, close_b);
}

function close_b(err) {

  if (err) {
```

```
      return callback(err);
    }

    fs.close(bFd, callback);
  }

  open_a();

}

console.log('starting...');

append_some_a_to_b(function(err) {

  if (err) {
    throw err;
  }

  console.log('done');
});
```

程序清单 19-2 声明了所有处于同一作用域的函数，应用这项技术摆脱了闭包嵌套效应，现在必须在处于每个函数上层的作用域中存储必要的值。

你也许已经注意到了，尽管这样做了，代码至少还存在两个问题：

- 每个函数都必须知道流程中的下一个函数名。
- 必须总是处理错误，这意味着存在大量的重复代码。

因为将所有回调函数都串接起来了，所以可以让代码变得更加通用和灵活，如程序清单 19-3 所示。

程序清单 19-3：使用通用流程控制方法

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

function cascade(callbacks, callback) {

  // 复制数组
  var functions = callbacks.slice(0);

  function processNext(err) {

    if (err) {
      return callback(err);
    }

    var args = Array.prototype.slice.call(arguments);

    var func = functions.shift();
    if (func) {
```

```
      // 删除包含错误的第一个参数
      args.shift();
    } else {
      func = callback;
    }

    args.push(processNext);

    func.apply(this, args);
  }

  processNext.call(this);
}

function append_some_a_to_b(callback) {

  var aFd, bFd,
      buffer = new Buffer(10);

  cascade([
    function open_a(next) {
      fs.open(__dirname + '/a.txt', 'r', next);
    },

    function read_from_a(fd, next) {
      aFd = fd;
      fs.read(aFd, buffer, 0, buffer.length, 0, next);
    },

    function close_a(bytesRead, buf, next) {
      fs.close(aFd, next);
    },

    function open_b(next) {
      fs.open(__dirname + '/b.txt', 'a', next);
    },

    function stat_b(fd, next) {
      bFd = fd;
      fs.fstat(bFd, next);
    },

    function write_b(bStats, next) {
      fs.write(bFd, buffer, 0, buffer.length, bStats.size, next);
    },

    function close_b(bytesWritten, buf, next) {
      fs.close(bFd, next);
    }
  ], callback);
```

```
};

console.log('starting...');

append_some_a_to_b(function done(err) {

  if (err) {
    throw err;
  }

  console.log('done');
});
```

程序清单 19-3 声明了一个带两个参数的函数 cascade，第一个参数包含所有要被调用的回调函数，第二个参数包含最终的回调函数，该回调函数在所有回调都完成时被调用，或者用来预防错误。

cascade 函数将会逐个调用回调函数，并向每个回调函数传入 processNext 函数，作为其最后一个参数。通过这种方法，每个回调函数不必知道下一个调用的究竟是哪个函数。

然后，append_some_a_to_b 函数用回调函数列表和最终回调函数来调用 cascade 函数。

19.3　使用 ASYNC 流程控制库

如前面的例子所示，有时可以回滚自定义的通用流程，然而，很多第三方库能够简化这项工作以及其他异步模式，其中一个就是 async 模块。

为了安装 async 模块，要将它包含进 package.json 清单，或者在应用程序的根目录下安装它，如下所示：

```
$ npm install async
```

async 模块具有一组辅助函数，这些辅助函数允许你进行异步迭代以及流程控制。

例如，程序清单 9-4 实现了一个返回给定数值平方的 HTTP 服务器。

程序清单 19-4：一个简单的平方运算服务器

```
var port = process.argv[2] && parseInt(process.argv[2], 10) || 8080;

require('http').createServer(function(req, res) {

  var body = '';

  req.setEncoding('utf8');

  req.on('data', function(data) {
    body += data;
  });
```

```
    req.once('end', function() {
      var number = JSON.parse(body);
      var squared = Math.pow(number, 2);
      res.end(JSON.stringify(squared));
    });

  }).listen(port, function() {
    console.log('Squaring Server listening on port %d', port);
  });
```

将上述脚本保存为 server.js 文件并启动它，如下所示：

```
$ node server.js
Squaring Server listening on port 8080
```

下面几节将要介绍 async 库中几个有用的辅助函数。

19.3.1 串行执行

可以执行一组异步返回结果的函数，利用程序清单 19-4 来学习如何使用 async 来串行执行异步函数。

为了运行这些例子，首先必须安装 request 第三方模块，如下所示：

```
$ npm install request
```

程序清单 19-5 展示了一个使用 async.series 辅助函数将两个 I/O 操作执行串接起来的例子。

程序清单 19-5：async.series 示例

```
var async   = require('async'),
    request = require('request');

function done(err, results) {
  if (err) {
    throw err;
  }
  console.log('results: %j', results);
}

async.series([

  function(next) {
    request.post({uri: 'http://localhost:8080', body: '4'},
      function(err, res, body) {
        next(err, body && JSON.parse(body));
      }
    );
  },
```

```
    function(next) {
      request.post({uri: 'http://localhost:8080', body: '5'},
        function(err, res, body) {
          next(err, body && JSON.parse(body));
        }
      );

    }

    ], done);
```

程序清单 19-5 执行了两次调用，这两次调用十分相似，但是一次传递的 body 值是 4，另一次传递的 body 值是 5。

可以将上述脚本保存为文件 series.js 并运行：

```
$ node series.js
results: [16,25]
```

所有函数都结束后，会调用回调函数 done，此时 done 函数带有每个回调函数的异步结果，在本例中，异步结果包含每次调用平方运算服务器后的输出结果，即包含 4 和 5 的平方。

19.3.2　并行执行

在前面的例子中，回调函数逐个被执行，然而这些回调函数也可以并行执行，为此，只需要将 async.series 调用修改为 async.parallel 调用，如程序清单 19-6 所示。

程序清单 19-6：Async. parallel 示例

```
var async   = require('async'),
    request = require('request');

function done(err, results) {
  if (err) {
    throw err;
  }
  console.log('results: %j', results);
}

async.parallel([

  function(next) {
    request.post({uri: 'http://localhost:8080', body: '4'},
      function(err, res, body) {
        next(err, body && JSON.parse(body));
      }
    );
  },
```

```
    function(next) {
      request.post({uri: 'http://localhost:8080', body: '5'},
        function(err, res, body) {
          next(err, body && JSON.parse(body));
        }
      );

    }

  ], done);
```

如果将上面的代码保存为文件 parallel.js 并运行，将会得到和前面的例子一样的结果：

```
$ node parallel.js
results: [16,25]
```

19.3.3 连续传递

同样还可以执行一串函数，其中下一个回调函数的执行取决于前一个回调函数的结果。例如，如果想用平方运算服务器计算 3 的 4 次方，用 async.waterfall 函数就能轻松进行这项工作，如程序清单 19-7 所示。

程序清单 19-7：用 async.waterfall 进行连续传递

```
var async   = require('async'),
    request = require('request');

function done(err, res, body) {
  if (err) {
    throw err;
  }

  console.log("3^4 = %d", body);
}

async.waterfall([

  function(next) {
    request.post({uri: 'http://localhost:8080', body: "3"}, next)
  },

  function(res, body, next) {
    request.post({uri: 'http://localhost:8080', body: body}, next);
  }

], done);
```

可以看到，除了第一个错误参数之外，每个回调函数都会得到最后一个操作的回调参数。如果出现错误，就会调用最后一个回调函数（本例中为 done），此时其错误参数的值

就是出现的错误。

19.3.4　排队

如果需要进行一些重复性的异步作业并想控制并发数——在给定时刻同时存在的正在进行的作业的最大数量——就可以使用 async.queue 函数。

该函数创建一个队列，该队列将基于一个函数处理其中的元素，队列的客户端可以将作业和并发作业的最大数量传入这个函数，按照顺序执行每个作业。

程序清单 19-8 中创建了一个队列，用平方运算服务器来计算数值的平方。

程序清单 19-8：使用 async.queue

```
var async   = require('async'),
    request = require('request');

function done(err, results) {
  if (err) {
    throw err;
  }
  console.log('results: %j', results);
}

var maximumConcurrency = 5;

function worker(task, callback) {
  request.post({uri: 'http://localhost:8080',
                body: JSON.stringify(task)},
    function(err, res, body) {
      callback(err, body && JSON.parse(body));
    }
  );
}

var queue = async.queue(worker, maximumConcurreny);

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].forEach(function(i) {

  queue.push(i, function(err, result) {

    if (err) {
      throw err;
    }

    console.log(i + '^2 = %d', result);

  });
});
```

程序清单 19-8 中声明了一个函数 worker，用这个函数来接受和处理任务，本例中任务

就是一个数值，worker 函数调用平方运算服务器来获取这个数值的平方值。

然后基于 worker 函数创建了一个队列，并定义了最大并发数，本例中最大并发数被设置为 5，但是它可以是任意值，可以根据自己的情况来定义它。

然后用 queue.push 将 10 个作业，也即数值 1~10 传入队列，queue.push 函数接受了一个回调函数，这个回调函数会在一个作业完成时接受其结果。

将上面的脚本保存为文件 queue.js 并运行：

```
$ node queue.js
5^2 = 25
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
6^2 = 36
10^2 = 100
8^2 = 64
9^2 = 81
7^2 = 49
```

注意：在上面的输出信息中可以看到，输出结果的顺序与传入数值的顺序并不一致，这是异步的特性：结果是按照计算完成的顺序进行显示的。如果要对结果进行排序，那么应该使用 async.foreach 函数或者类似的函数——在本章的后面将会介绍。

使用队列的优点之一是它限制了并发请求的数量，可以用它限制对类似数据库等资源的并发访问、避免出现泛滥。但是如何才能确定和适应最优的最大并发数呢？最大并发数是根据应用程序确定的，例如，它可能是具有可察觉延迟的函数，或者由外部设置。不管怎么做，知道在队列被创建之后，可以通过修改其 concurrency 属性来修改最大并发数都是有好处的：

```
queue.concurrency = 10;
```

队列还会发射一些与其状态有关的事件，可以通过将一个函数插入队列的 saturated 属性中，知道何时队列会到达最大并发数限制——此时后面的任务必须等待，如下所示：

```
queue.saturated = function() {
  console.log('queue is saturated');
};
```

同样，也可以通过监听 empty 事件，了解到何时队列中的最后一项被分配给了一个作业，如下所示：

```
queue.empty = function() {
  console.log('queue is empty');
};
```

当返回队列中的最后一项作业时，队列会发射 drain 事件：

```
queue.drain = function() {
  console.log('queue is drained, no more work!');
};
```

drain 事件和其他事件在队列的生命周期内会被发射多次，因为有可能在未来某个时间点会向队列添加作业。

19.3.5　迭代

JavaScript 有一些精巧的函数，用来在集合上进行同步迭代，async 提供了其中的一些函数，但是这些函数具有异步语义。例如，如果在 JavaScript 中有一个数组，就可以使用 array.foreach 来对其进行迭代：

```
[1, 2, 3, 4].forEach(function(i) {
  var squared = Math.pow(i, 2);
  console.log('%d ^ 2 is %d', i, squared);
});
```

如果不使用同步函数，就必须在每个元素值的基础上执行一些 I/O 操作，这种方法不会产生任何效果。

可以使用 async.forEach 在对象集合上进行异步迭代，如程序清单 19-9 所示。

程序清单 19-9：用 async.forEach 迭代集合

```
var async   = require('async'),
    request = require('request');

var results = {};

function done(err) {
  if (err) {
    throw err;
  }
  console.log('done! results: %j', results);
}

var collection = [1, 2, 3, 4];

function iterator(value, callback) {
  request.post({
      uri: 'http://localhost:8080',
      body: JSON.stringify(value)
    },
```

```
    function(err, res, body) {
      if (err) {
        return callback(err);
      }
      results[value] = JSON.parse(body);
      callback();
    });
}

async.forEach(collection, iterator, done);
```

这个例子定义了一个 iterator 函数，在 async.forEach 中用到了该函数。iterator 函数利用集合中的元素来执行必要的 I/O 操作，并且在 I/O 操作结束之后会调用回调函数。当在所有元素上完成了异步迭代时，就会调用最后一个回调函数(本例中为 done 函数)。

如果用到了 async.forEach 函数，就会针对每个元素并行调用相应的迭代器。如果希望 async 模块在一个迭代结束之后，再开始下一个迭代，可以使用 async.forEachSeries 函数，在本例中，只需要用如下所示的代码替换掉程序清单 19-9 中的最后一行即可：

```
async.forEachSeries(collection, iterator, done);
```

你也可以在进行并行迭代的同时控制最大并发数，通过使用 async.forEachLimit，就可以在设置任意时刻同时运行的迭代器的数量。在本例中，只需要用如下所示的代码替换掉程序清单 19-9 中的最后一行即可：

```
var maximumConcurrency = 5;
async.forEachLimit(collection, maximumConcurrency, iterator, done);
```

19.3.6 映射

JavaScript 还提供了一个函数 array.map()，用来同步迭代所有元素，并获得结果。

可以用 async.map 函数完成同样的任务，不同的是迭代是异步进行的，如程序清单 19-10 所示。

程序清单 19-10：用 async.map 映射集合

```
var async   = require('async'),
    request = require('request');

var collection = [1, 2, 3, 4];

function done(err, results) {
  if (err) {
    throw err;
  }
  console.log('done! results: %j', results);
}
```

```
function iterator(value, callback) {
  request.post({
      uri: 'http://localhost:8080',
      body: JSON.stringify(value)
    },

    function(err, res, body) {
      callback(err, body && JSON.parse(body));
    });
}

async.map(collection, iterator, done);
```

除了不必手工获取结果外，这个例子和前面的例子很相似——只要将每个结果传递给迭代函数中的回调函数即可。每次迭代都可能耗费任意时间完成，但是 async.map 在获得所有结果之后，可以按照正确的顺序将结果提交给回调函数 done。

19.3.7　规约

JavaScript 还有一个 array.reduce 函数，它将一个初始值和一个迭代函数作为参数，并返回一个新值，并将这个新值和新元素一并传入下一次迭代。

例如，reduce 函数的典型应用是对元素求和：

```
var collection = [1, 2, 3, 4];
function iterator(sum, elem) {
  return sum + elem;
}
var sum = collection.reduce(iterator, 0);
console.log('sum is %d', sum);
```

async.reduce 函数提供了相同的语义，但它是异步进行迭代的。

例如，如果想用平方运算服务器来计算一个集合中所有值的平方和，可以使用程序清单 19-11 中的代码。

程序清单 19-11：用 async.reduce 异步缩减集合

```
var async   = require('async'),
    request = require('request');

var collection = [1, 2, 3, 4];

function done(err, result) {
  if (err) {
    throw err;
  }
  console.log('The sum of the squares of %j is %d', collection, result);
}
```

```
function iterator(memo, item, callback) {
  request.post({
      uri: 'http://localhost:8080',
      body: JSON.stringify(item)
    },

    function(err, res, body) {
      callback(err, body && (memo + JSON.parse(body)));
    });
}

async.reduce(collection, 0, iterator, done);
```

与同步版本的 array.reduce 函数类似，async.reduce 函数将初始值作为其第二个参数，本例中为 0。然后在集合中的每个元素上进行迭代，当一次迭代结束时，都将一个新 memo 值传入回调函数，在本例中，这个新 memo 值是集合元素的平方值，再加上 memo 原来的值。

19.3.8 过滤

在 JavaScript 中，可以基于一个过滤函数对集合进行过滤，如下所示：

```
var collection = [1, 2, 3, 4];
function isEven(value) {
  return value % 2 === 0;
}
var evenElements = collection.filter(isEven);
console.log('even elements of %j are: %j', collection, evenElements);
```

上面这个例子对一个数值集合进行了过滤，并返回了奇数值。

async.filter 函数可以异步执行相似的操作，例如，可以用平方运算服务器过滤那些平方值大于 10 的元素，如程序清单 19-12 所示。

程序清单 19-12：使用 async.filter

```
var async   = require('async'),
    request = require('request');

var collection = [1, 2, 3, 4, 5];

function done(results) {
  console.log('These are the elements of %j whose ' +
              'square value is greater than 10: %j', collection, results);
}

function test(value) {
  return value > 10;
}
```

```
function filter(item, callback) {
  request.post({
      uri: 'http://localhost:8080',
      body: JSON.stringify(item)
    },

    function(err, res, body) {
      if (err) {
        throw err;
      }
      callback(body && test(JSON.parse(body)));
    });
}

async.filter(collection, filter, done);
```

filter 函数用一个布尔状态调用回调函数，无论这个值存在与否。还可以看到 filter 函数必须将这个布尔值作为第一个参数来调用回调函数，如果在过滤函数中存在错误，必须对错误进行人工处理。

async 还提供了一个作用相反的函数 reject，将程序清单 19-12 最后一行上的 async.filter 修改为 async.reject，就可以将那些平方值小于等于 10 的数过滤掉。

async.filter 和 async.reject 会针对每个元素并行调用 filter 函数，async 还提供了这两个函数的串行版本：

```
async.filterSeries(collection, filter, done);
```

和

```
async.rejectSeries(collection, filter, done);
```

使用这两个函数中的任何一个，在任意给定时刻都只会调用一个 filter 函数。

19.3.9　检测

当满足某个条件时，也许你想停止迭代。例如，也许想检测集合中第一个平方大于 10 的数，程序清单 19-13 展示了如何完成这项任务。

程序清单 19-13：使用 async.detect

```
var async   = require('async'),
    request = require('request');

var collection = [1, 2, 3, 4, 5];

function done(result) {
  console.log('The first element on %j whose square value '+
              'is greater than 10: %j', collection, result);
}
```

```
function test(value) {
  return value > 10;
}

function detect(item, callback) {
  request.post({
      uri: 'http://localhost:8080',
      body: JSON.stringify(item)
    },

    function(err, res, body) {
      if (err) {
        throw err;
      }
      callback(body && test(JSON.parse(body)));
    });
}

async.detect(collection, detect, done);
```

在程序清单 19-13 中，async.detect 针对每个元素并行调用 detect 函数，也就是说有可能无法得到满足条件的第一个元素——这取决于异步操作完成的顺序。

如果顺序很重要，或者在任意给定时刻只想调用一个 detect 函数，可以使用 async.detect 函数的串行版本：async.detectSeries。可以用如下所示的代码替换程序清单 19-13 的最后一行：

```
async.detectSeries(collection, detect, done);
```

在上面的代码中，async.detectSeries 调用了 detect 函数，并等待函数执行结束。如果元素检测失败，async.detectSeries 会针对下一个元素进行检测，直到检测到一个元素或者所有的元素都已经检测完毕。

19.4 本章小结

异步流程难以控制，它容易导致缺乏结构的代码，避免该问题的方法之一是事先将回调声明为命名函数，并避免对它们进行嵌套。

可以创建一个通用的流程控制例程，或者使用第三方模块 async 提供的众多辅助函数，使用这些辅助函数可以串行或并行地执行函数调用，或者以异步方式在集合上进行遍历，本章介绍了一些使用最频繁的函数，但是 async 提供的函数远不止这些。如果感兴趣的话，可以到 https://github.com/caolan/async#readme 上查看 async 的官方文档。

第V部分

构建Web应用程序

- 第 20 章：构建和使用 HTTP 中间件
- 第 21 章：使用 Express.js 创建 Web 应用程序
- 第 22 章：使用 Socket.IO 创建通用的实时 Web 应用程序

第20章

构建和使用 HTTP 中间件

本章提要：

- 理解 Connect 框架如何工作
- 创建与 Connect 框架兼容的中间件
- 创建处理错误的中间件组件
- 使用捆绑在 Connect 框架中的几个内置中间件组件
- 使用 Connect 中间件提交静态文件
- 解析查询字符串、请求主体和 cookies
- 操作会话

Node 特别适用于 HTTP 服务器，它具有极佳的 HTTP 解析器，能够在单个进程中有效处理大量的并发连接。

在构建 HTTP 应用服务器时，通常需要它执行一些常规任务，诸如解析 cookies 头部、解析请求 URL 上的查询字符串、维护与关联会话、持久化会话数据、提交静态文件、解析请求主体、记录请求与响应以及其他等等。当对应用程序的核心逻辑进行编码时，不必显式地执行这些任务，它们应该由 HTTP 服务器应用程序逻辑进行处理。

一些任务涉及检查请求——请求头或请求主体——另一些涉及检查和修改响应，还有一些任务以上两者都涉及了。两者都需要执行的这类任务的一个例子是：想在记录每个到达请求细节的同时，也记录其对应响应的 HTTP 状态码，在这种情况下既需要检查请求对象，也需要检查响应对象。你也许还希望能够设置或修改响应头，比如在使用浏览器 cookies 维护会话 ID 时。

在类似于 Django、Cake、Rails 或者 Sinatra 等典型的 HTTP 应用程序框架中，执行这些任务的组件通常被称作中间件(middleware)。这些中间件组件对请求-响应循环进行了包

装，在到达应用程序代码之前扩展请求对象或者在响应前后对其进行修改。

20.1 理解Connect HTTP中间件框架

Node还包含几个框架，这些框架可以简化基于HTTP的应用程序的开发，其中最流行的框架之一是Connect。Connect针对中间件组件定义了一个模型，同时还定义了一个引擎来运行中间件组件。

一般而言，程序员能够以堆栈形式将中间件组件组织到一起，当Connect接收到HTTP请求时，中间件引擎就会按顺序调用每一个组件。任何组件都有权访问请求对象，这意味着组件可以解析请求头、进行缓冲以及解析请求主体，它们也能修改响应头、向响应主体写入部分数据或者结束响应。

Connect是和一套中间件组件捆绑在一起的，但是可以很容易地创建自定义中间件。

20.2 构建自定义HTTP中间件

正如将要看到的那样，中间件组件只是一个接收请求和响应对象的函数，它与标准的Node HTTP服务器的request监听器非常相似。为了运行一个简单的中间件组件示例，首先需要在所选择的工作目录中安装connect模块：

```
$ npm install connect
```

程序清单20-1展示了一个简单的中间件组件，该组件用字符串“Hello World!”进行响应并结束响应。

程序清单20-1：一个简单的HelloWorld中间件

```
function helloWorld(req, res) {
  res.end('Hello World!');
}

module.exports = helloWorld;
```

将程序清单20-1保存为本地文件hello_world.js，以备后用。

该中间件组件只是导出了一个函数，该函数会被Connect调用来处理新的请求，它将字符串“Hello World”写入响应主体并结束请求。

可以创建一个Connect服务器来测试一下这个中间件组件，如程序清单20-2所示。

程序清单20-2：使用HelloWorld中间件的Connect服务器

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

// 导入中间件
```

```
var helloWorld = require('./hello_world');

var app = connect.createServer(helloWorld);
app.listen(8080);
```

上面的代码中创建了一个 Connect HTTP 服务器，并用 HelloWorld 中间件组件对其进行了初始化。将程序清单 20-2 保存为文件 hello_world_app.js 并运行：

```
$ node hello_world_app
```

将浏览器转向 http://localhost:8080，将会看到显示出“Hello World!”响应。

还可以创建一个更为通用的组件，用任意可选文本进行响应，如程序清单 20-3 所示。

程序清单 20-3：更为通用的 ReplyText 中间件

```
function replyText(text) {
  return function(req, res) {
    res.end(text);
  };
}

module.exports = replyText;
```

replyText 中间件不是直接导出一个处理请求的函数，而是导出了这样一个函数：它在被调用时会返回一个函数，这个函数会用定制文本字符串响应请求。

可以修改服务器代码，使用这个更为通用的组件输出字符串“Hello World!”，如程序清单 20-4 所示。

程序清单 20-4：使用 ReplyText 中间件的 Connect 服务器

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

//导入中间件
var replyText = require('./reply_text');

var app = connect.createServer(replyText('Hello World!'));
app.listen(8080);
```

将程序清单 20-4 保存为文件 hello_world_app_v2.js 并运行它：

```
$ node hello_world_app_v2
```

将浏览器转向 http://localhost:8080，将会看到再次显示出响应“Hello World!”。

20.2.1　创建异步中间件

中间件函数可以是异步的：当工作结束时，它需要调用一个回调函数，以便让中间件引擎继续工作。前面的例子没有用到这个特性是因为它结束了响应，而没有按顺序将控制权移交给下一个中间件。

为了实践一下，可以创建一个中间件组件，将头部数据引入响应头中，如程序清单20-5所示。

程序清单20-5：写入头部数据的Connect中间件

```
function writeHeader(name, value) {
  return function(req, res, next) {
    res.setHeader(name, value);
    next();
  };
}

module.exports = writeHeader;
```

这个中间件组件用到了第三个参数，该参数是一个在组件结束时被调用的回调函数。

将程序清单20-5保存为文件write_header.js。

现在将这个中间件组件包含到传入服务器构造函数的中间件堆栈中，并将其放置在ReplyText中间件之前，使其成为HelloWorld应用程序的一部分，如程序清单20-6所示。

程序清单20-6：将WriteHeader中间件包含进HelloWorld应用程序

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

// import middlewares
var writeHeader = require('./write_header');
var replyText = require('./reply_text');

var app = connect.createServer(
  writeHeader('X-Powered-By', 'Node'),
  replyText('Hello World!')
);

app.listen(8080);
```

将程序清单20-6存为文件hello_world_v3.js并运行：

```
$ node  hello_world_app_v3
```

然后运行命令行实用工具，创建一个HTTP请求，在请求中可以查看响应头，比如可以使用curl实用工具：

```
$ curl -i http://localhost:8080HTTP/1.1 200 OK
X-Powered-By: Node
Connection: keep-alive
Transfer-Encoding: chunked

Hello World!
```

通过以上信息，可以看出响应头被写入响应。

将写入头部数据的中间件(write_header)插入到写入和结束响应的中间件(reply_text)之前十分重要，如果将 ReplyText 中间件放到了前面，第二个中间件就永远不会被调用，因为 ReplyText 没有调用“紧接其后的”回调函数。

20.2.2　在中间件内部注册回调函数

除了检查请求和修改响应外，中间件组件还可以注册一些回调函数，这些回调函数随后会被执行。例如，可以创建一个中间件将每个请求的所有数据存入文件(见程序清单 20-7)。

程序清单 20-7：将每个请求存入文件的中间件

```
var fs = require('fs'),
  path = require('path'),
  util = require('util');

function saveRequest(dir) {

  return function(req, res, next) {

    var fileName = path.join(dir, Date.now().toString() + '_' +
      Math.floor(Math.random() * 100000) + '.txt');
    var file = fs.createWriteStream(fileName);
    file.write(req.method + ' ' + req.url + '\n');
    file.write(util.inspect(req.headers) + '\n');
    req.pipe(file);
    next();
  }
}

module.exports = saveRequest;
```

该中间件对于获得的每个请求，都会在指定的目录中创建一个新文件并将所有请求头存入该文件，然后它将请求传入该文件，此时还没有任何请求主体被写入。设置数据和其他事件监听器，当将来有数据到达时，就会被送入文件可写流。

现在，可以修改服务器来加入中间件组件，如程序清单 20-8 所示：

程序清单 20-8：包含 SaveRequest 中间件的 HelloWorld 服务器

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

// 导入中间件
var saveRequest = require('./save_request');
var writeHeader = require('./write_header');
var replyText = require('./reply_text');

var app = connect.createServer(
  saveRequest(__dirname + '/requests'),
```

```
  writeHeader('X-Powered-By', 'Node'),
  replyText('Hello World!')
);

app.listen(8080);
```

程序清单20-8让SaveRequest中间件将所有请求存入本地目录下的requests目录中，可以将程序清单20-8保存为文件hello_world_app_v4.js，同时创建目录来存储这些请求：

```
$ mkdir requests
```

现在可以启动服务器：

```
$ node hello_world_app_v4
```

可以用curl向服务器提供一个本地文件：

```
$ curl -T hello_world_app_v4.js http://localhost:8080
```

该命令告诉curl将文件上传到指定的URL中。该命令执行完毕之后，可以查看一下reuqests目录，应该会有一个类似“1340373961878_15062.txt”的文件，它包含如下所示的信息：

```
PUT /hello_world_app_v4.js
{ 'user-agent': 'curl/7.21.4 (universal-apple-darwin11.0) libcur
OpenSSL/0.9.8r zlib/1.2.5',
  host: 'localhost:8080',
  accept: '*/*',
  'content-length': '349',
  expect: '100-continue' }

var connect = require('connect');

// 导入中间件
var saveRequest = require('./save_request');
var writeHeader = require('./write_header');
var replyText = require('./reply_text');
var app = connect.createServer(
  saveRequest(__dirname + '/requests'),
  writeHeader('X-Powered-By', 'Node'),
  replyText('Hello World!')
);
```

正如预料的那样，该文件包含HTTP方法、URL、请求头以及请求主体。

20.2.3 在中间件内处理错误

中间件组件也可以处理错误，为了测试它的这种能力，首先必须创建一个“有错误的”中间件组件，该组件将抛出异常，如程序清单20-9所示：

程序清单 20-9：抛出错误的中间件组件

```
function errorCreator() {
  return function(req, res, next) {
    throw new Error('This is an error');
  }
}

module.exports = errorCreator;
```

该组件将一个异常抛回堆栈，之所以能这样做是因为 Connect 会捕获到异常并将其交给紧接其后的中间件错误处理程序。但如果在一个回调函数中捕获到错误，就应该将该错误作为第一个参数来调用“紧接其后的”回调函数，如程序清单 20-10 所示。

程序清单 20-10：另一个抛出错误的中间件组件

```
function errorCreator() {
  return function(req, res, next) {
    next(new Error('This is an error'));
  }
}

module.exports = errorCreator;
```

默认情况下，如果没有中间件被设置用来处理错误，Connect 就会以明文的形式显示错误结果，为了试验一下，可以假如创建错误中间件，如程序清单 20-11 所示。

程序清单 20-11：带有能够抛出错误的中间件的服务器

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

// 导入中间件
var errorCreator = require('./error_creator');
var saveRequest  = require('./save_request');
var writeHeader  = require('./write_header');
var replyText    = require('./reply_text');

var app = connect.createServer(
  errorCreator(),
  saveRequest(__dirname + '/requests'),
  writeHeader('X-Powered-By', 'Node'),
  replyText('Hello World!')
);

app.listen(8080);
```

将程序清单 20-11 保存为文件 hello_world_app_v5.js 并运行：

```
$ node  hello_world_app_v5
```

刷新浏览器，应该看到类似如下所示的错误信息：

```
Error: This is an error
    at Object.handle
(/Users/pedroteixeira/projects/wiley-node/code/20/error_creato
r.js:3:10)
    at next
(/Users/pedroteixeira/projects/wiley-node/code/20/node_modules/connect/
lib/proto.js:190:15)
    at Function.handle
(/Users/pedroteixeira/projects/wiley-node/code/20/node_modul
es/connect/lib/proto.js:198:3)
    at Server.app
(/Users/pedroteixeira/projects/wiley-node/code/20/node_modules/co
nnect/lib/connect.js:66:31)
    at Server.emit (events.js:70:17)
    at HTTPParser.onIncoming (http.js:1514:12)
    at HTTPParser.onHeadersComplete (http.js:102:31)
    at Socket.ondata (http.js:1410:22)
    at TCP.onread (net.js:374:27)
```

现在可以创建自定义的错误处理程序，Connect中的错误处理程序必须带有四个参数——错误、请求、响应以及“紧接其后的”回调函数(见程序清单20-12)。

程序清单20-12：错误处理中间件

```
function errorHandler() {
  return function(err, req, res, next) {
    if (err) {
      res.writeHead(500, {'Content-Type': 'text/html'});
      res.end('<h1>Oh no! We have an error!</h1>\n<pre>' + err.stack +
        '</pre>');
    } else {
      next();
    }
  }
}

module.exports = errorHandler;
```

将程序清单20-12保存为文件error_handler.js。

该错误处理程序对错误进行格式化处理并将其以HTML格式输出，现在可以通过让该中间件作为服务器构造函数的最后一个参数来将其整合到服务器中，如程序清单20-13所示。

程序清单20-13：带有定制错误处理程序的应用程序

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

```
// 导入中间件
var errorCreator = require('./error_creator');
var saveRequest  = require('./save_request');
var writeHeader  = require('./write_header');
var replyText    = require('./reply_text');
var errorHandler = require('./error_handler');

var app = connect.createServer(
  errorCreator(),
  saveRequest(__dirname + '/requests'),
  writeHeader('X-Powered-By', 'Node'),
  replyText('Hello World!'),
  errorHandler()
);

app.listen(8080);
```

将程序清单 20-13 保存为文件 hello_world_app_v6.js 并运行它：

```
$ node  hello_world_app_v6
```

如果刷新浏览器，将会得到一个 HTML 格式的错误输出，也可以使用 curl 命令行实用工具来检查响应：

```
$ curl http://localhost:8080
HTTP/1.1 500 Internal Server Error
Content-Type: text/html
Connection: keep-alive
Transfer-Encoding: chunked

<h1>Oh no! We have an error!</h1>
<pre>Error: This is an error
    at Object.handle
(/Users/pedroteixeira/projects/wiley-node/code/20/error_creato
r.js:3:10)
    at next
(/Users/pedroteixeira/projects/wiley-node/code/20/node_modules/connect/
lib/proto.js:190:15)
    at Function.handle
(/Users/pedroteixeira/projects/wiley-node/code/20/node_modul
es/connect/lib/proto.js:198:3)
    at Server.app
(/Users/pedroteixeira/projects/wiley-node/code/20/node_modules/co
nnect/lib/connect.js:66:31)
    at Server.emit (events.js:70:17)
    at HTTPParser.onIncoming (http.js:1514:12)
    at HTTPParser.onHeadersComplete (http.js:102:31)
    at Socket.ondata (http.js:1410:22)
    at TCP.onread (net.js:374:27)</pre>
```

可以看到请求头不包含定制的“X-Powered-By”条目，这是因为当有错误发生时，Connect 会执行 errorHandler 后面所有中间件组件中的第一个错误处理中间件，而跳过所有其他“普通的”中间件组件。

实际上，当出现错误时，Connect 只会执行错误处理中间件组件，这意味着可以减少错误处理中间件的代码，如程序清单 20-14 所示。

程序清单 20-14：简化的错误处理中间件组件

```
function errorHandler() {
  return function(err, req, res, next) {
    res.writeHead(500, {'Content-Type': 'text/html'});
    res.end('<h1>Oh no! We have an error!</h1>\n<pre>' + err.stack +
      '</pre>');
  }
}

module.exports = errorHandler;
```

20.3 使用捆绑在 Connect 中的 HTTP 中间件

Connect 不仅可以创建和使用 HTTP 中间件组件，而且它原本就带有一组常用的中间件组件。可以选择这样的一些中间件来满足应用程序的需求，即定制中间件堆栈，也许还可以将自定义的中间件和这些通用中间件组件混合使用。

20.3.1 记录请求

Connect 有一个记录某些请求和响应数据的中间件组件，它的用法很简单，如程序清单 20-15 所示。

程序清单 20-15：使用记录器中间件

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

var app = connect();

// 设置记录中间件
app.use(connect.logger());

// 实际响应
app.use(function(req, res) {
  res.end('Hello World!');
});

app.listen(8080);
```

注意：程序清单 20-15 中使用了另一种不同的语法来创建服务器，并将中间件附加其中。该清单中通过直接调用模块函数创建服务器，并使用.use()方法将中间件添加到服务器中，这是前面所看到的那种语法的一种替代语法，它更为方便，因为不需要在创建服务器时传递所有中间件组件。

将上述程序清单保存为文件 logger.js 并运行，如下所示：

```
$ node logger
```

现在如果将浏览器转向 http://localhost:8080，将会看到服务器输出中包含与如下所示类似的内容：

```
127.0.0.1 - - [Sat, 23 Jun 2012 15:23:07 GMT] "GET / HTTP/1.1" 200 - "-"
"Mozilla/5
.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5
(KHTML, like Gecko) Chrome/19.
0.1084.56 Safari/536.5"
127.0.0.1 - - [Sat, 23 Jun 2012 15:23:08 GMT] "GET /favicon.ico HTTP/1.1"
200 - "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/536.5
(KHTML, like Gecko
) Chrome/19.0.1084.56 Safari/536.5"
```

可以看到，服务器输出了请求和响应的一些细节，也许会看到服务器输出了两个请求，这是因为浏览器还会请求/favicon.ico 这个 URL，用来查找喜欢的图标并将其显示在地址栏中。

还可以看到，记录中间件输出了远程客户端地址，紧接其后的是当前日期、HTTP 方法、HTTP 版本、响应状态码、响应内容长度、HTTP 来源地址以及用户代理字符串。这是记录声明的默认格式，可以对其进行任意修改。可以选择记录请求上的每个元素(很快就会看到如何实现这一点)，但是针对简短微小的格式有一些预先定义的快捷方式。在创建中间件组件时需要指定格式：

```
app.use(connect.logger('tiny'));
```

如果将格式修改为 tiny 格式并重启服务器，就会输出如下信息：

```
GET / 200 - - - ms
GET /favicon.ico 200 - - 2 ms
```

tiny 格式将输出 HTTP 方法、URL、HTTP 状态码、内容长度头部数据(本例中没有显示)以及响应时间。

可以选择比这个“简短”输出更为冗长的输出格式，如下所示：

```
127.0.0.1 - GET / HTTP/1.1 200 - - - ms
127.0.0.1 - GET /favicon.ico HTTP/1.1 200 - - 1 ms
```

该格式在输出中增加了远程地址和HTTP版本。

还可以通过指定包含以下标志的字符串来自定义输出格式：

- :req[header]——可以指定任意请求头，例如：:req[Accept]，同样可以用这个格式化的字符串输出任意多的请求头。
- :http-version——HTTP版本。
- :response-time——请求到达记录器服务器的时间与请求结束时间之间的差值，以毫秒为单位。
- :remote-addr——发出请求的客户端的远程IP地址。
- :date——被格式化的创建记录的日期。
- :method——HTTP方法(GET、PUT、DELETE、HEAD等等)。
- :url——请求的URL。
- :referrer——HTTP来源地址头部信息。
- :user-agent——用户代理字符串。
- :status——数值形式的HTTP响应状态码。

例如，如果只想输出HTTP方法、URL、状态码和响应时间，可以使用下面的格式：

```
var format = ':method :url - :status - :response-time ms';
app.use(connect.logger(format));
```

20.3.2 处理错误

Connect带有一个设计非常精巧的错误处理程序。它的功能之一是使用内容协商来了解什么输出才是最佳的。例如，如果客户端接受text/html内容类型，它就输出HTML形式的内容。如果客户端接受application/json内容类型，它将以包含error和stack属性的JSON编码对象的形式输出错误。如果这些内容类型都不被客户端所支持，错误处理程序将以明文的形式输出错误。

可以通过将错误处理程序中间件添加到服务器来使用它，如程序清单20-16所示。

程序清单20-16：使用Connect错误处理程序

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

var app = connect();

app.use(function(req, res, next) {
  next(new Error('Hey!'));
});

// 实际响应
app.use(function(req, res) {
  res.end('Hello World!');
```

```
});

app.use(connect.errorHandler());

app.listen(8080);
```

程序清单 20-16 中引发了一个错误并将错误处理程序中间件放置到中间件堆栈的末尾。如果将程序清单 20-16 保存为文件 errorHandler.js，可以通过如下命令来运行它：

```
$ node errorHandler
```

如果刷新浏览器，将会看到一个包含标题 Connect、错误消息以及错误跟踪的格式化的 HTML 页面。可以通过设置 errorHandler 的标题属性并引入下面这样一行代码来修改标题：

```
connect.errorHandler.title = 'My Application';
```

20.3.3　提交静态文件

HTTP 应用程序通常需要提交一些静态资源，如 CSS、JavaScript、图像以及 HTML 文件，Connect 已经包含了提供这项功能的中间件组件(见程序清单 20-17)。

程序清单 20-17：使用 Connect 的静态服务器中间件

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

var app = connect();

// 设置静态文件服务器
app.use(connect.static(__dirname + '/public'));

// 实际响应
app.use(function(req, res) {
  res.end('Hello World!');
});

app.listen(8080);
```

在程序清单 20-17 中，静态文件服务器应该在 public 目录下查找文件，如果找到一个文件，文件就被提交，中间件链也会被中断。

现在可以在当前文件夹下创建 public 目录，选择任意 JPEG 图像文件将其放置到该目录中，并将其重命名为 pic.jpg，然后在父文件夹中将程序清单 20-17 保存为文件 static.js 并运行它，如下所示：

```
$ node static
```

然后将浏览器转向 URL http://localhost:8080/pic.jpg，应该能看到浏览器中所呈现的图片。

现在将浏览器转向其他的URL，比如http://localhost:8080/abc，应该能看到所呈现的字符串“Hello World”，这意味着静态文件中间件组件将控制权传递给了它后面的一个组件，本例中是“Hello World!”呈现器。

注意：有一个有趣的实验，如果删除“Hello World!”呈现器并尝试将浏览器转向一个静态服务器无法响应的URL，将会看到Connect会用以下信息做出响应：

```
Cannot GET <URL>
```

这种情况会在Connect无法找到对请求做出响应的中间件时发生。

20.3.4 解析查询字符串

向服务器端页面生成器传递参数的一个典型惯用方法是使用查询字符串。在查询字符串中可以对一个包含名称-值对的字符串进行编码。它位于URL的末尾、？字符的后面。下面是一个带有查询字符串的URL的例子：

```
/search?q=node.js&where=mailinglist
```

如果该字符串可用，那么Connect有一个中间件组件可以解析它，它将query属性输入到请求上，使其可为后继的中间件组件所用。

程序清单20-18展示了一个例子。

程序清单20-18：使用Connect的查询字符串解析中间件

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

var app = connect();

// 设置查询中间件
app.use(connect.query());

// 实际响应
app.use(function(req, res) {
  res.end(JSON.stringify(req.query));
});

app.listen(8080);
```

程序清单20-18先设置了一个查询字符串解析中间件，然后以req.query的内容做出响应，可以将程序清单20-18保存为文件query.js并运行它：

```
$ node query
```

如果将浏览器转向 URL http://localhost:8080/?a=b&c=d，应该看到如下所示的响应：

```
{"a":"b","c":"d"}
```

现在可以修改 URL 的查询字符串部分，看看它是如何给 req.query 对象带来变化的。

20.3.5　解析请求主体

在 HTTP 中，请求主体是紧接着请求头的请求组成部分，它可以包含应用程序选择的任意内容，通常用于上传文件、提交表单以及传递远程 API 参数。

当提交表单时，有两种类型的请求主体编码格式：URL 编码和多部件编码，URL 编码格式和查询字符串格式相似，多部件编码是一种更为复杂的格式，主要用来上传文件以及传递更为复杂的数据类型。

当向服务器上运行的 API 调用传递参数时，远程客户端可以将调用参数编码成 JSON 编码体。

所有主要的编码类型都由请求内容类型字符串指定，并且需要根据请求内容类型字符串在请求主体解码器间进行转换。幸好，Connect 附带了一个通用的请求主体解析中间件组件来完成这项工作，该中间件组件在请求对象上提供了一个不可更改的 body 属性，该属性可在稍后被请求所用。

可以像程序清单 20-19 那样使用请求主体解析中间件。

程序清单 20-19：使用 Connect 的请求主体解析中间件

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

var app = connect();

// 设置中间件
app.use(connect.logger(':method :req[content-type]'));
app.use(connect.bodyParser());

// 实际响应
app.use(function(req, res) {
  res.end(JSON.stringify(req.body));
});

app.listen(8080);
```

程序清单 20-19 中添加了一个记录中间件来检查 HTTP 方法和请求内容类型头。

将程序清单 20-19 保存为文件 body.js 并运行它，如下所示：

```
$ node body
```

现在可以用 curl 实用工具向服务器发出一些请求，首先用简单的 URL 编码发出请求：

```
$ curl -X POST --data-urlencode a=b --data-urlencode c=d
  http://localhost:8080
```

上面这条命令的意思是a参数将具有值b，c参数将具有值d，可以看到响应和预料的一致：

```
{"a":"b","c":"d"}
```

还可以看到服务器记录了如下信息：

```
POST application/x-www-form-urlencoded
```

现在可以测试一下多部件编码：

```
$ curl -X POST -F a=b -F c=d http://localhost:8080
```

响应应该如下所示：

```
{"a":"b","c":"d"}
```

服务器应该包含一条新的记录消息，如下所示：

```
POST multipart/form-data;
boundary=----------------------------cfb73467cea3
```

现在可以测试一下JSON编码，curl本身并不支持该编码，必须自己完成这项工作：

```
$ curl -d '{"a":"b","c":"d"}' http://localhost:8080
{"{\"a\":\"b\",\"c\":\"d\"}":""}
```

可以看到，req.body包含一个字符串，Connect无法解析请求主体，而是提交了一个字符串作为代替。这是因为没有指定内容类型是application/json。使用curl时应该设置请求头，如下所示：

```
$ curl -d '{"a":"b","c":"d"}' -H 'Content-Type: application/json' \
http://localhost:8080
{"a":"b","c":"d"}
```

现在可以看到，bodyParse中间件能够识别内容类型并对其进行正确地解码。

20.3.6 解析Cookies

Cookies是Web浏览器上将数据存储为短字符串的一个标准，服务器可以发送一个或者多个cookie，每一个cookie都有一个名称。服务器通过使用Set-Cookie响应头在任意类型请求的响应中发送一个或多个cookie，浏览器必须存储每一个cookie，并将每一个cookie作为后续请求上的单个Cookie头进行发送，直到cookie过期为止。Cookie头拥有自己的编码机制，因为很多名称-值对都能在此处进行编码。用已经存储在浏览器上的cookie名称设置cookie可以重写原来的cookie。

Connect 附带有一个负责解析请求 cookies 的 cookie 中间件，即在请求对象上设置 cookies 属性。可以像程序清单 20-20 中所示使用该中间件。

程序清单 20-20：使用 Connect 的主体解析中间件

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

var app = connect();

// 设置中间件
app.use(connect.cookieParser());

// 实际响应
app.use(function(req, res) {
  res.end(JSON.stringify(req.cookies));
});

app.listen(8080);
```

程序清单 20-20 以 req.cookies 的内容作为响应，它应该包含一个对象，该对象具有客户端发送的每一个 cookie 的属性。为了进行测试，将程序清单 20-20 保存为文件 cookie.js 并运行它：

```
$ node cookie
```

可以通过-b 选项使用 curl 命令行实用工具来测试服务器，这会在 cookie 头上设置内容：

```
$ curl -b 'a=b; c=d' http://localhost:8080
{"a":"b","c":"d"}
```

可以看到，cookies 被正确地解析到 req.cookies 对象。

20.3.7　使用会话

Connect 有一个中间件组件允许在服务器端保持会话，即允许保存会话值，为将来的请求所用。当客户端首次连接时，就创建了一个会话标识符，并且 Connect 会发送包含会话标识符的浏览器 cookie 来关联浏览器和创建的会话。在以后的请求中，浏览器发送此 cookie，该 cookie 允许服务器标识会话并获取存储的会话对象以备进一步使用和修改。

在服务器端，会话就是一个允许存储名称-值对的对象。默认情况下，Connect 将会话存储在内存中，但是它有一个可插拔接口允许定义不同的存储器。有几个第三方会话存储器允许使用 Memcache、Radis 以及其他数据库服务器作为 Connect 的会话存储器。

注意：如果使用内存储器，会话无法在进程重启之后还继续存在。如果使用持久型会话存储器，将得到一个外部服务，会话在这个外部服务中将得到保持，并能在Node进程重启之后继续存在。同样，使用外部会话存储器允许强制两个或者更多Node进程共享会话状态，这意味着如果使用负载均衡器，任何请求都可以在任意指定时刻攻击运行应用程序的进程，并且总可以获得会话数据。

虽然对于本例而言，将要使用默认的内置内存存储器，但是如果应用程序产品的设置涉及不止一个Node进程时，就应该使用外部持久型存储器。

程序清单20-21展示了如何设置会话中间件。

程序清单20-21：使用Connect的会话中间件

```
var connect = require('connect');
var format = require('util').format;

var app = connect();

// 设置中间件
app.use(connect.query());

app.use(connect.cookieParser('this is my secret string'));

app.use(connect.session({
  cookie: { maxAge: 24 * 60 * 60 * 1000 }
}));

// 实际响应
app.use(function(req, res) {
  for (var name in req.query) {
    req.session[name] = req.query[name];
  }
  res.end(format(req.session) + '\n');
});

app.listen(8080);
```

在程序清单20-21中设置了一些中间件组件。

首先用到了查询字符串解析中间件，这仅仅是因为例子需要，正如后面将看到的那样。

然后设置了cookie解析中间件，但是此处将一个字符串传递给了中间件构造函数，这个字符串包含一个密文，服务器使用这个密文为cookies中的数据进行签名，以此确保数据不会被客户端修改。

这之后设置了会话中间件，会话中间件中定义了cookie.maxAge选项，该选项定义了

会话 cookie 在过期之前能够在浏览器中存在多久。

然后，当请求到达时循环遍历 req.query 对象上的每一个属性并将这些属性复制到会话中，这是一种简单(但不是十分安全)的方法，可以让你稍后很容易地通过修改 URL 的查询字符串部分来测试会话对象。

最后，通过序列化 req.session 属性的内容对请求做出响应。

将程序清单 20-21 保存为文件 session.js 并运行：

```
$ node session
```

然后将浏览器转向 http://localhost:8080，应该能够看到如下所示的信息：

```
{ cookie:
  { path: '/',
    _expires: Sun, 24 Jun 2012 10:58:24 GMT,
    originalMaxAge: 60000,
    httpOnly: true } }
```

这显示了会话有一个 cookie 属性，该属性用来存储 cookie 信息，存储的信息将为会话中间件所用。

现在可以使用查询字符串来设置会话，尝试在浏览器中加载下面的 URL：

```
http://localhost:8080/?a=b
```

将看到如下所示的信息：

```
{ cookie:
  { path: '/',
    _expires: Sun, 24 Jun 2012 11:02:22 GMT,
    originalMaxAge: 60 * 60 * 1000,
    httpOnly: true },
 a: 'b' }
```

在上面的信息中，可以看到会话获得了查询字符串上显示的名称和值。

现在可以尝试传递一个不带查询字符串的 URL(比如 http://localhost:8080)来观察会话数据，应该能看到如下所示的信息：

```
{ cookie:
  { path: '/',
    _expires: Sun, 24 Jun 2012 11:04:17 GMT,
    originalMaxAge: 60000,
    httpOnly: true },
 a: 'b' }
```

现在可以看到会话被成功地记录下来，并在随后的请求上被恢复。

现在可以设置不同的变量名(比如 http://localhost:8080/?c=d)来执行其他请求，应该能看到前面的会话数据加上了刚刚提供的新名称-值对：

```
{ cookie:

  { path: '/',
    _expires: Sun, 24 Jun 2012 12:06:44 GMT,
    originalMaxAge: 3599999,
    httpOnly: true },
 a: 'b',
 c: 'd' }
```

20.3.8 其他可用的中间件

本章描述了 Connect 中一些最为流行和实用的可用中间件组件，但是其他一些中间件组件还能提供另外一些功能，像跨站请求伪造(cross-site request forgery, CSRF)保护、Gzip 压缩、静态文件缓存、虚拟主机支持、请求主体大小限制等。

你已经看到了使用 Redis 或者 Memcache 的会话存储器，但是还有其他一些会话存储器使用诸如 MongoDB、CouchDB 以及 MySQL 等不同的数据库。

而且很多第三方中间件组件和 Connect 兼容，允许提供 JSONP 支持、显示模板、支持请求超时、整合不同的认证方法、支持动态样式语言(LESS、CSS DSL)以及其他等等。在 https://github.com/senchalabs/connect/wiki 上有一个第三方中间件组件的列表，该列表由 Connect 项目的赞助人维护。

20.4 本章小结

Connect 已经成为定义和运行中间件组件的标准，可以很容易地创建组件来进行响应、记录数据、传输请求主体、处理错误以及以可重用的方式做些其他的事情。

还可以使用 Connect 提供的中间件组件，这些中间件包含这样一些模块：解析查询字符串、解析请求主体、解析 cookies、提交静态文件、处理错误和维护会话。

同样，很多第三方模块允许你很容易就能扩展你的 Node HTTP 服务器提供的功能。

第21章

用 Express.js 创建 Web 应用程序

本章提要：

- 安装 Express
- 使用 Express 启动 Web 应用程序
- 定义 Express 中间件
- 显示 Jade 模板
- 定义 URL 路由监听器
- 创建路由中间件

对于一些语言和平台而言，可以用一些框架来简化创建基于 HTTP 的应用程序的工作，也许最知名框架的就是 Rails(针对 Ruby 语言)，而不计其数的其他框架包括 Django、Sinatra 和 Cake。大多数框架的目标是简化和架构 HTML 应用程序的开发，以及解决一些一般性问题，比如将 HTTP 请求发送给正确的控制代码、提交静态资源、显示 HTML 模板。

在 Node 的世界里有一些模块和框架同样可以解决上述这些问题以及其他一些问题，其中一些是可以整合到一起的独立模块，而其他一些——像 Express.js——提供了一个可以快速实现的完整解决方案。

Express.js 基于 Connect 中间件引擎：任意 Connect 指定的中间件也可以和 Express 一起进行工作。除了允许创建中间件来响应 HTTP 请求外，Express.js 还允许基于 HTTP 方法和 URL 执行不同的任务——通过定义路由表。它还基于向模板提供参数来动态显示 HTML 文档。

21.1 初始化 Express.js 应用程序

本节展示如何从头开始创建 Express 应用程序，但首先必须用 npm 全局安装 Express：

```
$ npm install -g express@2.5.x
```

注意：如果上述命令因为不具有足够的权限（取决于系统）而失败，也可以使用 sudo 来执行该命令，如下所示：

```
$ sudo npm install -g express@2.5.x
```

如果无法这样做，就尝试调整文件系统中的权限。

现在可以使用 express 二进制模块在本地初始化应用程序：

```
$ express my_app
   create : my_app
   create : my_app/package.json
   create : my_app/app.js
   create : my_app/public
   create : my_app/public/javascripts
   create : my_app/public/images
   create : my_app/public/stylesheets
   create : my_app/public/stylesheets/style.css
   create : my_app/routes
   create : my_app/routes/index.js
   create : my_app/views
   create : my_app/views/layout.jade
   create : my_app/views/index.jade
```

Express 创建了一个目录 my_app，然后在其中创建了一组文件，可以在这组文件的基础上新建应用程序。进入 my_app 目录，并打开代码编辑器来分析一下 Express 到底创建了什么。

首先，有一个清单文件 package.json，它包含如下所示的一些内容：

```
{
  "name": "application-name"
, "version": "0.0.1"
, "private": true
, "dependencies": {
    "express": "2.5.11"
  , "jade": ">= 0.0.1"
  }
}
```

这个清单文件的主要目的是包含模块依赖列表，NPM 可以根据这个列表来安装所依赖的模块，现在就让我们进行安装：

```
$ npm install
...
jade@0.26.2 ./node_modules/jade
├── commander@0.5.2
└── mkdirp@0.3.0

express@2.5.11 ./node_modules/express
├── qs@0.4.2
├── mime@1.2.4
├── mkdirp@0.3.0
└── connect@1.9.0 (formidable@1.0.11)
```

上述代码将会下载 Express 模块、Jade 模板语言模块以及它们的依赖模块，并将它们安装到本地 node_modules 目录中，可以看到 Express 依赖的模块中包括 Connect 模块。

下一步浏览 Express 创建的 public 目录，可以看到该目录包含应用程序可能需要的静态文件的目录，从某种程度上说，这些目录是标准目录，用来分隔样式表、图像以及客户端 JS 文件的存储位置，但是可以随意修改这些目录。

```
$ ls public
images    javascripts    stylesheets
```

还可以看到Express创建了一个文件app.js，该文件包含一些针对服务器的初始化代码，如程序清单 21-1 所示。

程序清单 21-1：Express 创建的 app.js 文件

```
/**
 * 模块依赖
 */

var express = require('express')
  , routes = require('./routes');

var app = module.exports = express.createServer();

// 配置

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

```
app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});

// 路由

app.get('/', routes.index);

app.listen(3000, function(){
  console.log("Express server listening on port %d in %s mode",
    app.address().port,
     app.settings.env);
});
```

在载入初始化依赖模块之后，用 express.crearteServer()实例化 Express 服务器，该服务器继承自 Connect HTTP 服务器，而 Connect HTTP 服务器继承自 Node HTTP 服务器。

这之后的代码是进行应用程序的一些设置，首先将 views 设置为模板文件的根，用来创建 HTML 页面。然后将 view engine 设置为 jade，jade 是一种 HTML 模板语言，可以执行希望模板语言所能执行的任务——它允许显示 HTML 块、显示动态值、执行循环以及包含其他模板。可以查看一些简单的 Jade 模板，这些模板都位于 Express 刚才创建的 views 目录中。

在下一节中，读者将学习如何为应用程序定义中间件。

21.2 在应用程序中设置中间件

继续 app.js 文件中的代码，可以看到一些中间件组件被添加到应用程序中：

```
app.configure(function(){
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

上述代码由回调函数进行包装，这个回调函数被传递给 app.configure。这个回调函数是被同步调用的，将这些代码写在此处可以让代码与前面由环境指定的配置部分一致。

在回调函数中创建了主体解析中间件组件——在第 20 章中介绍过——用来根据请求中定义的内容类型解析请求主体。

注意： Connect 框架与 Express 对外暴露的一组有用的中间件组件捆绑在一起，例如，express.bodyParser 向外暴露了 Connect 主体解析中间件，这可以让你不用安装 Connect 模块就可以使用 Express 的中间件。

然后，方法重载中间件被添加到应用程序中，该中间件组件可以让浏览器模拟使用包括 GET 和 POST 在内的 HTTP 方法。HTTP 标准定义了很多方法，比如 GET、POST、DELETE、PUT 等等，但是 Web 浏览器无法使用它们。在提交一份表单或者发出一个 AJAX 请求时，大多数浏览器都只能使用 GET 或者 POST 请求。但是将伪 HTTP 方法放置在 _method 隐藏参数中，浏览器就可以在服务器和方法重写中间件的帮助下，貌似能够使用其他方法。

注意： 使用方法重载中间件有几个优点，首先，能够基于标准的 HTTP 方法做出更丰富的路由决策。其次，能够像处理浏览器一样处理请求该方法的 HTTP 的客户端，这会简化服务器端代码。

程序清单 21-1 的后面是路由器中间件，根据 HTTP 方法和 URL，路由器中间件组件将请求分发给定义在路由表中合适的监听器，在后面将会回到这个主题上来。

然后，设置静态文件服务器中间件，它将在 public 目录中提交静态文件。

```
app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});
```

在 Express 中，应用程序运行的每个环境都有独立的初始化代码块，例如，也许在开发环境中使用内存会话存储器，但是在产品环境中希望使用类似 Memcache 或 Redis 这样的外部服务作为会话存储器。

在本例中，希望错误处理程序在开发环境中抛出异常并显示堆栈跟踪，但是在产品环境下运行时又希望将这些信息隐藏起来。

注意：可以通过设置NODE_ENV环境变量来改变运行环境，例如，当在产品环境下启动Node进程时，就要将运行环境设置为production：

```
$ NODE_ENV=production node my_app
```

Express中默认的运行环境是development。

你还可以指定多个运行环境，如果希望在这些运行环境中执行某个代码块，只需要在configure函数的最后一个参数之前列出这些运行环境即可：

```
app.configure('production', 'staging', function(){
  app.use(express.errorHandler());
});
```

21.3 路由请求

在定义HTTP服务器请求监听器或者中间件组件时，针对每个请求都会调用相应的代码段，但是，显然应用程序的不同部分需要触发不同的服务器行为，例如，GET请求应该和POST请求区别处理，并且不同的URL也应该触发不同的响应。例如，如果想要创建的应用程序的某一部分显示用户简介，就希望下列方法和URL的组合具有不同的行为：

- GET/users——显示用户列表或者查找用户。
- GET/users/:username——通常显示给定用户名的用户简介，如下一行所示。
- GET/users/:joe——显示用户名为“joe”的用户简介。
- POST/users——创建用户简介。
- PUT /users/:username——根据指定的用户名更新用户简介，如下一行所示。
- PUT /users/joe——更新用户名为“joe”的用户简介。

上面这个列表定义了路由表，路由表将HTTP方法和URL模式映射成行为，下一节展示了在Express中如何定义这些方法和模式。

21.3.1 处理路由

在Express创建的app.js文件的第32行中(程序清单21-1)，

```
app.get('/', routes.index);
```

上面的代码告诉应用程序，如果向URL / 发送GET请求，就会调用routes.index处理程序，在第7行中导入了routes对象，并且在文件routes/index.js中对该对象进行了定义，可以对该模型做一点改动来适应新的路由器。

修改文件routes/index.js，如程序清单21-2所示。

程序清单 21-2：新版 routes/index.js

```
/*
 * GET 主页
 */

module.exports = function(app) {
  app.get('/', function(req, res){
    res.render('index', { title: 'Express' })
  });
};
```

修改 app.js 的第 32 行，如下所示：

```
require('./routes/index')(app);
```

然后，在文件 routes/users.js 中定义新的路由器，如程序清单 21-3 所示。

程序清单 21-3：用户路由器

```
/*
 * 用户路由器
 */

module.exports = function(app) {

  app.get('/users/:name', function(req, res){
    res.render('users/profile', {title: 'User profile'});
  });

};
```

然后，需要在 app.js 的第 32 行之后增加一行，如下所示：

```
require('./routes/users')(app);
```

注意：上述设置路由器的模型将路由设置提交给每个路由组，这会避免文件 app.js 被路由弄得凌乱不堪。

在程序清单 21-3 中，为应用程序定义了两个新的路由处理程序，第一个路由处理程序会在请求具有 GET 方法和 URL /users 时被激活，在本例中会显示模板 users/index，同时向模板传递了一个标题，将这个标题作为模板的变量参数。

另一个路由处理程序是动态路由，它匹配的路由以/users/起始，并在其后紧跟着一个非空字符串，这个字符串可为 req.params.name 中的路由处理程序所用。

在 Express 路由中，具有:前缀的字符串被当作占位符处理，除了/字符外，获得的值可

以匹配任意字符串。

现在启动服务器：

```
$ node app
```

然后，可以将浏览器转向 http://localhost:3000，应该看到显示出默认的索引视图。如果将浏览器转向 http://localhost:3000/users，将会看到显示一个错误消息，并且服务器的输出应该包含如下所示的信息：

```
failed to locate view "users/index", tried:
  - /Users/myusername/projects/my_app/views/users/index.jade
  - /Users/myusername/projects/my_app/views/users/index/index.jade
  - /Users/myusername/projects/my_app/views/users/../index/index.jade
```

Express 用缺失的模板文件的位置来帮助你，现在必须在 views/users/index.jade 中创建模板文件，如程序清单 21-4 所示。

程序清单 21-4：用户列表视图

```
h1 Users

p User list should be here
```

将文件保存并刷新页面之后，将会看见一个包含用户列表占位符的简单 HTML 页面，为了简化显示，现在要在 data/users.json 下创建一个模拟用户列表。创建项目根目录 data，然后创建一个包含相同内容的文件 users.json，如程序清单 21-5 所示。

程序清单 21-5：模拟的用户

```
{

  "frank": {
    "username": "frank",
    "name": "Frank Sinatra",
    "bio": "Singer"
  },

  "jobim": {
    "username": "jobim",
    "name": "Antonio Carlos Jobim",
    "bio": "Composer"
  },

  "fred": {
    "username": "fred",
    "name": "Fred Astaire",
    "bio": "Dancer and Actor"
  }

}
```

现在可以完成用户路由监听器，如程序清单 21-6 所示。

程序清单 21-6：完成后的用户路由监听器

```
/*
 * 用户路由
 */

var users = require('../data/users');

module.exports = function(app) {

  app.get('/users', function(req, res){
    res.render('users/index', {title: 'Users', users: users});
  });

  app.get('/users/new', function(req, res) {
    res.render('users/new', {title: "New User"});
  });

  app.get('/users/:name', function(req, res, next){
    var user = users[req.params.name];
    if (user) {
      res.render('users/profile', {title: 'User profile', user: user});
    } else {
      next();
    }

  });

  app.post('/users', function(req, res) {
    if (users[req.body.username]) {
      res.send('Conflict', 409);
    } else {
      users[req.body.username] = req.body;
      res.redirect('/users');
    }
  });

  app.del('/users/:name', function(req, res, next) {
    if (users[req.params.name]) {
      delete users[req.params.name];
      res.redirect('/users');
    } else {
      next();
    }
  });

};
```

上面的代码实现了所有基本的函数，这些函数用于列出用户、显示用户、创建用户和删除用户。让我们逐个仔细地看看每一块代码。

最开始，列表导入了用户的模拟数据库，它只是一个经过解析并载入内存的 JSON 结构。

```
app.get('/users', function(req, res){
  res.render('users/index', {title: 'Users', users: users});
});
```

GET /users 路由的监听器只是将用户“数据库”传入 views/users/index.jade 模板，现在需要修改这个模板来显示用户，如程序清单 21-7 所示。

程序清单 21-7：用户列表模板

```
h1 Users

p
  a(href="/users/new") Create new profile

ul
  - for(var username in users) {
    li
      a(href="/users/" + encodeURIComponent(username))=
users[username].name
  - };
```

这个列表迭代所有用户对象，针对每个用户显示一个列表项，每个列表项包含指向一个详细 URL 的链接。在这里可以看到 Jade 模板的一个示例，以及学习如何用标记点缀代码。

现在重启服务器来测试新代码。

当将浏览器转向 http://localhost:3000/users 时，应该能看到用户列表显示出来。如果单击某个用户链接，就激活了如下所示的路由监听器：

```
app.get('/users/:name', function(req, res, next){
  var user = users[req.params.name];
  if (user) {
    res.render('users/profile', {title: 'User profile', user: user});
  } else {
    next();
  }
});
```

这个路由监听器用到了包含用户名的请求参数 param，它用这个参数来查找用户对象。如果找到一个用户，就被应用于模板 users/profile.jade。如果没有找到用户，控制权就会返回给中间件引擎，本例中会显示“Not found”消息。

如果单击列表中的某个用户，将看到一个错误显示，告知 Express 无法找到模板

users/profile，必须在 views/users/profile.jade 下创建这个模板，如程序清单 21-8 所示。

程序清单 21-8：用户简介模板

```
h1= user.name

h2 Bio
p= user.bio

form(action="/users/" + encodeURIComponent(user.username), method="POST")
  input(name="_method", type="hidden", value="DELETE")
  input(type="submit", value="Delete")
```

在第一小节中会显示用户名和个人简介，然后提供了一个带有按钮的简单表单，可以用这个按钮将用户从数据库中删除。还可以利用该方法重载中间件组件，传入一个请求主体参数_method，该参数包含字符串“DELETE”。当方法重载中间件设置这个参数时，会将请求方法从 POST 修改为 DELETE，这样就会激活如下所示的路由监听器：

```
app.del('/users/:name', function(req, res, next) {
  if (users[req.params.name]) {
    delete users[req.params.name];
    res.redirect('/users');
  } else {
    next();
  }
});
```

这个监听器会在数据库中查找用户，如果用户存在，监听器就将其从数据库中删除，并将请求重定向到用户列表。

通过单击链接 Create new profile，还可以在用户列表中新建用户，这样会激活路由监听器 GET /users/new：

```
app.get('/users/new', function(req, res) {
  res.render('users/new', {title: "New User"});
});
```

这个监听器只是用于模板 views/users/new.jade，需要创建该模板，如程序清单 21-9 所示。

程序清单 21-9：新用户表单模板

```
h1 New User

form(method="POST", action="/users")
  p
    label(for="username") Username<br />
    input#username(name="username")
  p
    label(for="name") Name<br />
    input#name(name="name")
```

```
p
  label(for="bio") Bio<br />
  textarea#bio(name="bio")
p
  input(type="submit", value="Create")
```

上面这个模板包含一个简单的表单，用于创建用户，当点击 Create 按钮时，就会向 URL /users 发出一个请求，从而激活如下所示的路由监听器：

```
app.post('/users', function(req, res) {
  if (users[req.body.username]) {
    res.send('Conflict', 409);
  } else {
    users[req.body.username] = req.body;
    res.redirect('/users');
  }
});
```

监听器函数的第一行检查用户名是否已存在于“数据库”中，如果存在，就会以 409 状态码做出响应，这是向客户端表示发生冲突，同时也会用 res.send 显示一个字符串“Conflict”，这个字符串就是响应主体，而 res.send 是由 Express 提供的一个实用工具，允许用字符串进行响应，以及设置响应状态码。

如果没有冲突，就将用户对象存入数据库中，然后，用 res.redirect 实用函数将用户重定向到 URL /users。

注意：在这个例子中，将用户数据库存储在内存中，但是在实际的应用程序中，显然想保留这些数据，这意味着要进行异步的 I/O 操作。如果有一个简单通用的键-值对存储模块来访问数据库，那么最后有关路由监听器的代码就应该如下所示：

```
app.post('/users', function(req, res, next) {
  db.get(req.body.username, function(err, user) {
    if (err) {
      return next(err);
    }
    if (user) {
      res.send('Conflict', 409);
    } else {
      db.set(req.body.username, req.body, function(err) {
        if (err) {
          return next(err);
        }
        res.redirect('/users');
      });
    }
  }
```

在第 23、24 和 25 章中，将学习如何用 Node 来连接一些流行的数据库。

21.3.2　使用会话

现在已经有了一个基本的基于 HTTP 的用户管理应用程序(或者是某个应用程序的一部分)，这个应用程序可以用多种方法改进。一种方法就是提供某种验证机制和会话管理，允许用户通过用户名和密码进行登录。首先需要在文件 app.js 中设置会话中间件：

```
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser('my secret string'));

  app.use(express.session({
    secret: 'my secret string',
    maxAge: 3600000
  }));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

在上面的代码中，设置了一个带有加密字符串(可以修改)的 cookie 解析器中间件，因为就像前面介绍的那样，因为会话中间件需要用 cookie 来维护会话，所以它需要 cookie 解析器中间件。

此外，还设置了会话中间件，并定义了共享加密字符串和最长会话期，最长会话期定义了在没有用户的 HTTP 活动时，会话能够存在的最长时间。

然后需要创建一个会话路由监听器，可以将其存入文件 routes/session.js 中(见程序清单 21-10)。

程序清单 21-10：会话路由

```
/*
 * 会话路由
 */

var users = require('../data/users');

module.exports = function(app) {

  app.dynamicHelpers({
    session: function(req, res) {
      return req.session;
    }
  });

  app.get('/session/new', function(req, res) {
    res.render('session/new', {title: "Log in"});
```

```
});

app.post('/session', function(req, res) {
  if (users[req.body.username] &&
    users[req.body.username].password === req.body.password) {
      req.session.user = users[req.body.username];
      res.redirect('/users');
    } else {
      res.redirect('/session/new')
    }
  });

  app.del('/session', function(req, res, next) {
    req.session.destroy();
    res.redirect('/users');
  });

};
```

仔细查看程序清单21-10中的每块代码，可以看到如下所示的一段代码：

```
app.dynamicHelpers({
  session: function(req, res) {
    return req.session;
  }
});
```

这是针对视图中传入局部变量的替代方法，不是显式地将参数插入每个res.render，而是定义一个动态辅助函数来求每个请求的值，结果就被用作一个局部变量。在上面这段代码中，对外暴露了会话，它在稍后会为任意的视图模板所用。

现在可以修改页面布局，并为其创建一个头，这个头会显示用户的登录状态。为此，要在views/session/user.jade下新建一个文件，如程序清单21-11所示。

程序清单21-11：Express创建的app.js文件

```
- if (session.user) {

  p
    span Hello 
    span= session.user.name
    span !
  p
    form(method="POST", action="/session")
      input(type="hidden", name="_method", value="DELETE")
      input(type="submit", value="Log out")
- } else {

  p
    a(href="/session/new") Login
```

```
    span  or 
    a(href="/users/new") Register

- }
```

这部分代码会用两种模式进行显示：会话中有用户以及会话中没有用户。在第一种情况中，会显示一些用户数据和一个按钮来结束会话。在后一种情况中，会显示一个链接让用户登录。

现在需要将上面这部分代码包含进文件 views/layout.jade，如程序清单 21-12 所示。

程序清单 21-12：修改后的布局文件，在页头上包含用户信息

```
!!!
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    header!= partial('session/user')
    section#main!= body
```

在上面的代码中，将 views/session/user.jade 中的部分内容分配给了 header 标记。

注意：在 Jade 中，分配给标记的内容可以是 HTML 编码的，也可以不是。如果想分配一个字符串，使其能够正确地在浏览器中显示(甚至是 HTML 特定的字符，例如<and>)，就要使用=操作符，它将会对这些特殊字符进行编码：

```
span= session.user.name
```

如果内容中包含 HTML，并希望将 HTML 显示为一个标记（正如本例中所包含的那部分代码的结果），就应该使用!=：

```
header!= partial('session/user')
```

如果单击链接 link，就会激活如下所示的路由：

```
app.get('/session/new', function(req, res) {
  res.render('session/new', {title: "Log in"});
});
```

这个路由只是显示模板文件 views/session/new.jade，这个模板文件是一个登录表单，如程序清单 21-13 所示。

程序清单21-13：登录表单

```
h1 Log in

form(method="POST", action="/session")
  p
    label(for="username") User name:<br />
    input#username(name="username")
  p
    label(for="password") Password:<br />
    input#password(type="password", name="password")
  p
    input(type="submit", value="Log in");
```

这个表单会将数据提交给URL /session，从而激活如下所示的路由：

```
app.post('/session', function(req, res) {
  if (users[req.body.username] &&
    users[req.body.username].password === req.body.password) {
      req.session.user = users[req.body.username];
      res.redirect('/users');
    } else {
      res.redirect('/session/new')
    }
});
```

上面这个路由监听器会根据模拟的用户数据库检查用户名和密码，如果匹配，就将用户对象存入会话，然后将用户重定向到URL /users。

当用户登录成功，页头会显示用户名和一个用于退出登录的按钮，这个按钮会向URL /session发出一个DELETE请求(再次提醒，可以使用方法重载中间件)，从而激活如下所示的路由：

```
app.del('/session', function(req, res, next) {
  req.session.destroy();
  res.redirect('/users');
});
```

上面这个路由监听器只是销毁会话，并重定向到URL /users。

下一步，要向app.js增加这些新路由的初始化代码，为此，要在app.js中的路由部分增加如下所示的一行：

```
require('./routes/session')(app);
```

现在可以登录了，但是模拟数据库中还没有一个用户拥有密码，可以为这些用户添加密码，并重启应用程序。

为了添加密码，需要修改data/users.json文件中每个用户的记录，如下所示：

```
{
  "frank": {
    "username": "frank",
    "name": "Frank Sinatra",
    "bio": "Singer",
    "password": "theladyisatramp"
  },

  "jobim": {
    "username": "jobim",
    "name": "Antonio Carlos Jobim",
    "bio": "Composer",
    "password": "garotadeipanema"
  },

  "fred": {
    "username": "fred",
    "name": "Fred Astaire",
    "bio": "Dancer and Actor",
    "password": "tophat"
  }

}
```

此外，还需要在签名表单(views/users/new.jade)中增加一个密码域，如程序清单 21-14 所示：

程序清单 21-14：新版的签名表单

```
h1 New User

form(method="POST", action="/users")
  p
    label(for="username") Username<br />
    input#username(name="username")
  p
    label(for="name") Name<br />
    input#name(name="name")
  p
    label(for="password") Password<br />
    input#password(type="password", name="password")
  p
    label(for="bio") Bio<br />
    textarea#bio(name="bio")
  p
    input(type="submit", value="Create")
```

现在可以创建一个拥有密码的用户来进行登录和退出登录。

现在应该将用户访问重定向到一些路由，例如，用户应该只能在登录成功后才能创建

其简介，他也只能删除自己的简介。为此，要用到一个新特性：路由中间件。

21.3.3 使用路由中间件

也许想基于会话信息限制某些行为，例如，要限制执行一些到成功登录的用户的路由，尤其要限制执行一些到具有内部简介特征的用户的路由(比如成为管理员)。

可以让每个有关的路由监听器执行上述任务，但是这会导致重复的、不优雅的代码。另一种方法是在所选择的路由上定义要执行的路由中间件。

例如，如果想要针对没有登录的用户，将对数据的访问重定向到 GET /users/new、POST /users、GET /sessions/new 和 POST /session，可以创建如下所示的通用路由中间件，并将这个中间件存入文件 routes/middleware/not_logged_in.js(见程序清单 21-15)。

程序清单 21-15：未登录中间件

```
function notLoggedIn(req, res, next) {
  if (req.session.user) {
    res.send('Unauthorized', 401);
  } else {
    next();
  }
}

module.exports = notLoggedIn;
```

上面这个中间件只是检查用户是否在会话中，如果在，就发送状态 401 Unauthorized；如果不在，就处理下一个中间件。

然后可以将这个中间件导入文件 routes/session.js，并将其放置在文件的最上方，在每个需要该中间件的路由中都要进行这种包含处理(见程序清单 21-16)。

程序清单 21-16：新版的会话路由

```
/*
 * 会话路由
 */

var users = require('../data/users');
var notLoggedIn = require('./middleware/not_logged_in');

module.exports = function(app) {

  app.dynamicHelpers({
    session: function(req, res) {
      return req.session;
    }
  });

  app.get('/session/new', notLoggedIn, function(req, res) {
```

```
  res.render('session/new', {title: "Log in"});
});

app.post('/session', notLoggedIn, function(req, res) {
  if (users[req.body.username] &&
    users[req.body.username].password === req.body.password) {
      req.session.user = users[req.body.username];
      res.redirect('/users');
    } else {
      res.redirect('/session/new')
    }
  });

  app.del('/session', function(req, res, next) {
    req.session.destroy();
    res.redirect('/users');
  });

};
```

在上面的代码中，将新的路由中间件回调函数放在了最后一个监听器之前，Express 会将路由中间件的调用串接起来。

此外，还必须使用用户路由中的函数，如程序清单 12-17 所示。

程序清单 21-17：新版的用户路由

```
/*
 * 用户路由
 */

var users = require('../data/users');
var notLoggedIn = require('./middleware/not_logged_in');

module.exports = function(app) {

  app.get('/users', function(req, res){
    res.render('users/index', {title: 'Users', users: users});
  });

  app.get('/users/new', notLoggedIn, function(req, res) {
    res.render('users/new', {title: "New User"});
  });

  app.get('/users/:name', function(req, res, next){
    var user = users[req.params.name];
    if (user) {
      res.render('users/profile', {title: 'User profile', user: user});
    } else {
      next();
    }
```

```
});

app.post('/users', notLoggedIn, function(req, res) {
  if (users[req.body.username]) {
    res.send('Conflict', 409);
  } else {
    users[req.body.username] = req.body;
    res.redirect('/users');
  }
});

app.del('/users/:name', function(req, res, next) {
  if (users[req.params.name]) {
    delete users[req.params.name];
    res.redirect('/users');
  } else {
    next();
  }
});

};
```

可以利用这种策略和机会删除一些加载用户的冗余代码，为此，要在文件 routes/middleware/load_user.js 中新建一个路由中间件组件，这个路由中间件基于用户名参数加载用户，如程序清单 21-18 所示。

程序清单 21-18：加载用户路由中间件

```
var users = require('../../data/users');

function loadUser(req, res, next) {
  var user = req.user = users[req.params.name];
  if (! user) {
    res.send('Not found', 404);
  } else {
    next();
  }
}

module.exports = loadUser;
```

上面这个中间件将用户载入属性 req.user，使其可为下一个要执行的中间件所用。

现在要包含这个中间件，并重用文件 routes/users.js 中的一些冗余代码，如程序清单 21-19 所示。

程序清单 21-19：新版的用户路由

```
/*
 * 用户路由
 */

var users = require('../data/users');
var notLoggedIn = require('./middleware/not_logged_in');
var loadUser = require('./middleware/load_user');

module.exports = function(app) {

  app.get('/users', function(req, res){
    res.render('users/index', {title: 'Users', users: users});
  });

  app.get('/users/new', notLoggedIn, function(req, res) {
    res.
/*
 * 用户路由
 */

var users = require('../data/users');
var notLoggedIn = require('./middleware/not_logged_in');
var loadUser = require('./middleware/load_user');

module.exports = function(app) {

  app.get('/users', function(req, res){
    res.render('users/index', {title: 'Users', users: users});
  });

  app.get('/users/new', notLoggedIn, function(req, res) {
    res.render('users/new', {title: "New User"});
  });

  app.get('/users/:name', loadUser, function(req, res, next){
    res.render('users/profile', {title: 'User profile', user: req.user});
  });

  app.post('/users', notLoggedIn, function(req, res) {
    if (users[req.body.username]) {
      res.send('Conflict', 409);
    } else {
      users[req.body.username] = req.body;
      res.redirect('/users');
    }
  });

  app.del('/users/:name', loadUser, function(req, res, next) {
```

```
    delete users[req.user.username];
    res.redirect('/users');
  });

};
```

现在，需要保护用户简历不被其他用户操纵，为此，需要新建一个中间件检查加载的用户是否与登录的用户相同，如 routes/middleware/restrict_user_to_self.js 中的程序清单21-20所示。

程序清单21-20：将用户限制为会话用户的新中间件

```
function restrictUserToSelf(req, res, next) {
  if (! req.session.user || req.session.user.username !== req.user.username) {
    res.send('Unauthorized', 401);
  } else {
    next();
  }
}

module.exports = restrictUserToSelf;
```

现在可以导入这个新建的路由中间件，并将其放置在用户路由的上面：

```
var restrictUserToSelf = require('./middleware/restrict_user_to_self');
```

并且使用位于用户路由中最后一个监听器中的中间件：

```
app.del('/users/:name', loadUser, restrictUserToSelf, function(req,
  delete users[req.user.username];
  res.redirect('/users');
});
```

然后重启服务器，确认登录成功后只能删除自己的简介。

注意：这个应用程序还需要进行大量改进才能保证其安全性，这些改进包括要对用户输入进行清理和验证、当产生动作时要反馈消息、当出现错误时显示更多友好的消息。

还应该使用持久化数据库来存储用户数据，这在本书的后面将会介绍。

21.4 本章小结

Express 是用来构建基于 HTML 的网站的最流行的 Web 框架之一，它建立在 Connect

提供的中间件功能之上，允许你基于它查看显示信息和路由。

可以使用 Connect 命令行开始项目，并以此为起点，增加或者删除中间件组件，以及定义路由表。

路由用到了一个简单的针对方法和请求 URL 的模式-匹配表单，不应将所有路由都放置在主模块中，相反，应该针对不同的功能集合使用单独的模块。

在 Express 中，路由的运行类似于全局中间件，在路由中，可以定义一个监听某个特定路由的监听器函数堆栈，使用这个特征可以保证最终的路由监听器更为安全，并能减少其冗余度。

第22章

使用 Socket.IO 创建通用的实时 Web 应用程序

本章提要：

- 理解与实时应用程序相关的 HTTP 历史
- 理解 WebSockets 如何使实时应用程序在网络上变得可行
- 使用 Socket.IO 为 Node.js 编写实时应用程序
- 使用针对 Node.js 的 Socket.IO 编写基于网络的实时聊天应用程序
- 使用名称空间分离 Socket.IO 应用程序
- 使用 Redis 扩展到几个 Node 进程

多年来，HTTP 和实时应用程序无法真正很好地共存，原因在于 HTTP 协议的特殊性质，这个特殊性质是指 HTTP 协议是无状态的，并且是基于请求-响应循环的——HTTP 服务器只会响应来自客户端的请求，但是，它与客户端之间不具备有状态的持续连接。

基于 HTML 和 JavaScript 的应用程序可以响应客户端事件，并将这些事件发送到服务器上，但是反其道而行之却并不容易。如果在服务器端发生一个事件，服务器无法将有关该事件的信息实时主动地通知它的客户端。只有在客户端查询服务器的当前状态时，所发生事件的信息才会从服务器传递到客户端。

基于浏览器的聊天应用程序是基于 web 的实时应用程序的一个很好的示例，它展示了 HTTP 的局限性。多个用户共享一间聊天室，这从技术上意味着任何一个参与聊天的用户所发送的消息都必然会传递给同一聊天室中的其他用户。

为了获得实时性(或至少是近实时性)，针对传统网络聊天应用程序中的行为采用一些

变通方法十分普遍，形式最简单的变通方法是采用传统的请求-响应流程：每个聊天用户的浏览器会每隔一定时间对 HTTP 服务器进行轮询，看是否有新的消息，只要轮询速度足够快，例如每隔 1 秒或更少时间，就会给人造成交互是实时进行的印象。

然而，上述技术极其低效，客户端必须每隔 1 秒就要进行一次轮询，服务器也必须每隔 1 秒对来自所有连接到其上的聊天用户的请求进行处理，即使没有新消息也是如此。在一个每隔 10 秒钟才会发布新消息的聊天室中，客户端和服务器都必须进行大量计算，而其中 90%的计算都是没有必要的。

现在已经开发出了更为灵活的变通方法，但是这些变通方法仍然是基于请求-响应循环的，只是在使用请求-响应循环的效率方面略微有所提高而已。长轮询是这样一种技术：客户端只向服务器发出一次请求，然后服务器就会将连接保持为打开状态，并且只要有新数据可用就将其发送出去。但是因为 HTTP 不是被设计用于这一目的，所以长轮询的实现仍然会造成一些问题——浏览器在面对连续的请求响应时会表现出不同的行为，此外，将连接保持为打开状态通常会在服务器端导致低效的加载行为。

如今的情况要好得多，WebSocket 协议已被开发出来(最近成为了标准)，用于克服与实时应用程序有关的 HTTP 的缺陷，现在它不需要所谓的变通方式，就可以让 HTTP 客户端与浏览器以高效、全双工的方式进行交互。

实时网络应用程序首次能够只使用网络技术，而不需要诸如 Java Applets、Flash 或者 ActiveX 这样的外部技术。

22.1 理解 WebSockets 如何工作

WebSocket 连接在其核心只不过是 HTTP 服务器和 HTTP 客户端之间传统的 TCP 连接，WebSocket 连接是用一种握手协议创建的，该握手协议和 HTTP 握手十分相似。

客户端会初始化连接，为此，浏览器会向服务器发送一个特殊的 HTTP/1.1 请求，要求服务器将特殊请求的连接转换为 WebSocket 连接：

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

虽然是从常规的 HTTP 连接开始，但是客户端要求将这个常规连接“升级”为 WebSocket 连接。如果服务器支持 WebSocket 协议，会做出如下所示的响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

```
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

上面的响应标志着握手的结束，并且连接已经切换到数据传输模式。客户端和服务器不需要 HTTP 开销或者额外的握手就可以来回传递消息，也就是说创建了一个双向全双工的通信连接，在这个连接中，客户端和服务器不必互相等待，在任意时刻都可以向对方发送消息。

22.2　使用 Socket.IO 创建 WebSocket 应用程序

虽然为 Node.js 实现自定义的 WebSocket 服务器是可能的，但是没有必要这样做。在实现一个实际的应用程序之前，有很多底层细节需要处理，而应用程序就建立在这些底层细节之上，这样一来使用库就要实用得多。

创建 WebSocket Node.js 应用程序事实上的标准库是 Socket.IO，它不仅是一个包装器库，使得创建 WebSocket 服务器十分方便，而且还为不支持 WebSocket 协议的客户端提供了类似长轮询的透明回调机制，此外它还带有一套客户端的库，这套库为开发应用程序的浏览器部分提供了一组方便的 API。

通过使用 Soket.IO，就不需要解决 WebSocket 服务器或客户端的实现细节问题。无论是在服务器端还是在客户端，都能获得一组纯净和富于表现力的 API，这可以让你很轻松地编写出实时应用程序。

22.2.1　在服务器上安装和运行 Socket.IO

与很多其他的 Node.js 库相似，Socket.IO 也是一个 NPM 模块，因此，可以按照如下所示的方式来安装 Socket.IO：

```
$ npm install socket.io
```

在安装 Socket.IO 之后，就可以实现一个非常基本的 WebSocket 服务器，如程序清单 22-1 所示。

程序清单 22-1：创建 Socket.IO 服务器

```
var io = require('socket.io').listen(4000);

io.sockets.on('connection', function (socket) {
  socket.on('my event', function(content) {
    console.log(content);
  });
});
```

将上面的代码保存为文件 server.js。

对应的客户端部分如程序清单 22-2 所示。

程序清单 22-2：基本的 Socket.IO 客户端

```
<html>
  <head>
    <title>Socket.IO example application</title>
    <script src="http://localhost:4000/socket.io/socket.io.js"></script>
    <script>
      var socket = io.connect('http://localhost:4000');
      socket.emit('my event', 'Hello world.');
    </script>
  </head>
  <body>
  </body>
</html>
```

将上面的代码保存为文件 index.html，并将其与 server.js 放置在同一目录下。

如你所见，服务器端和客户端用到了相似的词汇表。在服务器上，第一步是绑定一个端口，在本例中是 4000。只要服务器运行起来，它就会监听传入的 WebSocket 连接，只要有新客户端连接到其上，就会触发 connection 事件，然后服务器就会监听该连接上传入的消息，并记录收到的消息内容。

注意：如你所见，Socket.IO 提供了一种面向消息的通信机制，这是对底层 WebSockets 协议的一个改进，WebSockets 协议不提供消息分帧。

还可以看到，Socket.IO 对客户端是采用 WebSockets 进行连接还是采用其他类型的机制进行连接并不进行区分：它提供了一组统一的 API，这对实现细节进行了抽象。

事件名 my event 是随便取的——它只不过是客户端赋予所发送消息的标签而已，用来在应用程序内部区分不同类型的消息。

在客户端，通过连接到服务器来创建 WebSocket 连接，然后套接字被用来向服务器发射(或发送)my event 消息，这会触发服务器代码中的监听器，结果是将消息内容输出到控制台。

现在，如果通过 file:///path/to/index.html 以本地方式打开 HTML 文件会非常顺利，但是由于打开本地文件时所应用的安全性限制，使得这个操作无法在所有浏览器中都能可靠地进行。

因此，需要通过 Node.js 服务器来提交 HTML 文件，它可以让你请求 http://localhost:4000/，这就不会强加任何安全性限制，因为这是一个“真正的” HTTP 请求。

为此，就要将 Socket.IO 服务器架设在常规的 Node.js HTTP 服务器之上，并且通过这个 HTTP 服务器来提交 HTML 文件。听起来复杂吗？不，如程序清单 22-3 所示。

程序清单 22-3：用 Node.js HTTP 服务器提交 HTML 文件

```
var httpd = require('http').createServer(handler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');

httpd.listen(4000);

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.on('my event', function(content) {
    console.log(content);
  });
});
```

在上面的代码中，WebSocket 服务器链接到了 HTTP 服务器中，并监听某些请求。当客户端在/socket.io 之下进行任意请求，Socket.IO 都会处理这些请求，并返回 socket.io.js 脚本或者创建 WebSocket 连接，其他请求则会接收到 index.html 文件作为响应。

现在将浏览器转向 http://localhost:4000/——字符串“Hello World”就会输出在服务器的控制台上。

下面将介绍如何重写示例代码，使其成为一个完整的应用程序。

22.2.2　使用 Socket.IO 创建实时网络聊天应用程序

Node.js 为网络上的实时应用程序提供了基础构件：它基于网络标准的 JavaScript，在本质上并不是非阻塞的，这对实现实时行为十分重要。

因此，实时 web 应用程序的典型代表，即基于浏览器的聊天室成为展示 Node.js 性能的最流行的示例应用程序并不奇怪。

这个示例应用程序对于基本的聊天应用程序而言是一个很好的起点，它需要在服务器端和客户端都进行扩展，下面几节将会逐步讲解如何进行这种扩展。

1. 应用程序结构和用例

基于浏览器的聊天应用程序非常简单，用户通过一个十分简约的界面与聊天系统进行

交互，交互界面只有两个部件——一个文本输入域以及一个显示所有接收到的聊天信息的区域。

当用户在输入域中输入文本并按下回车键时，输入的文本就会通过 Socket.IO 发送到服务器。当在 Socket.IO 连接上接收到消息时，客户端就会将其添加到消息区域中。

每个客户端都会接收到其他客户端发送的所有消息，并且这些消息也会返还给发送端。

2. 聊天服务器

首先要重写程序清单 22-3 中创建的服务器应用程序，需要监听新连接和新消息。当有来自客户端的新消息到达时，都要将其返还给发送它的客户端，然后将其广播给连接到服务器上的其他客户端。可以通过维护一个包含所有现在已连接到服务器上的客户端队列来实现广播部分，但是没有必要这样做——Socket.IO 已经提供了一种方便的广播机制，可以将消息发送给所有已连接到服务器上的客户端(发送消息的客户端除外)。

因为聊天服务器的文件提交部分不要修改，所以主要重写处理套接字连接的部分：

```
io.sockets.on('connection', function (socket) {
  socket.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);
    socket.broadcast.emit('serverMessage', socket.id + ' said: ' +
      content);
  });
});
```

现在需要监听每个连接套接字上的 clientMessage 事件，并通过将事件内容返还给其发送端以及广播给其他客户端的方式来处理该事件。为了区分客户端上的聊天消息，在每条聊天消息之前加上了前缀 You said:或者<client id> said:，client id 是当前内部的客户端 ID 号。

重申一下，传入的事件名称并不是特殊的关键词，不具备任何与 Socket.IO 相关的含义——它可以是一个任意选择的标签。当然，需要保证客户端在套接字上发射消息时采用的是同一标签。

程序清单 22-4 展示了此时完整的服务器代码。

程序清单 22-4：一个简单的 Socket.IO 聊天应用程序服务器

```
var httpd = require('http').createServer(handler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');

httpd.listen(4000);

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.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);
    socket.broadcast.emit('serverMessage', socket.id + ' said: ' +
      content);
  });
});
```

3. 聊天客户端

因为在聊天应用程序服务器端上的消息处理仍然很初级，所以大部分繁重的消息处理实际是发生在客户端上。

因此，需要重写前述示例代码的大部分内容。客户端需要一个部件让客户输入聊天消息，还需要一个区域显示所有传入的聊天消息。首先，要在页面的<head>节中使用级联样式表，来对页面上的这两个部件进行样式处理和定位处理。

```
<style type="text/css">
  #input {
    width: 200px;
  }
  #messages {
    position: fixed;
    top: 40px;
    bottom: 8px;
    left: 8px;
    right: 8px;
    border: 1px solid #EEEEEE;
    padding: 8px;
  }
</style>
```

input 部件是一个简单的文本输入域，位于页面的顶端，紧接其后是一个 div 标签，它可以让消息占满页面的剩余部分。

```
Your message:
<input type="text" id="input">

<div id="messages"></div>
```

在body结束标签的前面需要加载Socket.IO客户端库，并且增加客户端应用程序逻辑。将JavaScript放在文档末尾十分重要，这可以保证所有的DOM元素都可以为JavaScript代码所用。

首先，需要加载Socket.IO客户端库：

```
<script src="http://localhost:4000/socket.io/socket.io.js"></script>
```

之后再用<script type="text/javascript">为代码打开另一个脚本块后，需要处理两个用例——来自服务器的消息到达客户端，以及消息必须被发送到服务器。让我们从一个函数开始，该函数知道如何将新消息添加到消息域中：

```
var messagesElement = document.getElementById('messages');
var lastMessageElement = null;
function addMessage(message) {
  var newMessageElement = document.createElement('div');
  var newMessageText = document.createTextNode(message);
  newMessageElement.appendChild(newMessageText);
  messagesElement.insertBefore(newMessageElement, lastMessageElement);
  lastMessageElement = newMessageElement;
}
```

新消息通过Socket.IO连接到达，所以必须连接到后台，并为服务器代码发射的serverMessage事件增加一个处理程序：

```
var socket = io.connect('http://localhost:4000');
socket.on('serverMessage', function(content) {
  addMessage(content);
});
```

最后一步是发射clientMessage事件，该事件将用户消息作为其内容，为了达到这一目的，可以绑定输入文本域的onkeydown事件，并且当用户按下回车键时就能将用户消息写入套接字：

```
var inputElement = document.getElementById('input');

inputElement.onkeydown = function(keyboardEvent) {
  if (keyboardEvent.keyCode === 13) {
    socket.emit('clientMessage', inputElement.value);
    inputElement.value = '';
    return false;
  } else {
    return true;
  }
};
```

将前述内容组合到一起就形成了文件index.html，如程序清单22-5所示。

程序清单 22-5：一个简单的 Socket.IO 聊天应用程序客户端

```
<html>
  <head>
    <title>Node.js WebSocket chat</title>
    <style type="text/css">
      #input {
        width: 200px;
      }
      #messages {
        position: fixed;
        top: 40px;
        bottom: 8px;
        left: 8px;
        right: 8px;
        border: 1px solid #EEEEEE;
        padding: 8px;
      }
    </style>
  </head>

  <body>

    Your message:
    <input type="text" id="input">

    <div id="messages"></div>

    <script src="http://localhost:4000/socket.io/socket.io.js"></script>
    <script type="text/javascript">
      var messagesElement = document.getElementById('messages');
      var lastMessageElement = null;

      function addMessage(message) {
        var newMessageElement = document.createElement('div');
        var newMessageText = document.createTextNode(message);

        newMessageElement.appendChild(newMessageText);
        messagesElement.insertBefore(newMessageElement,
          lastMessageElement);
        lastMessageElement = newMessageElement;
      }

      var socket = io.connect('http://localhost:4000');
      socket.on('serverMessage', function(content) {
        addMessage(content);
      });

      var inputElement = document.getElementById('input');
```

```
    inputElement.onkeydown = function(keyboardEvent) {
      if (keyboardEvent.keyCode === 13) {
        socket.emit('clientMessage', inputElement.value);
        inputElement.value = '';
        return false;
      } else {
        return true;
      }
    };
  </script>

  </body>
</html>
```

上面的代码实现了最简单的聊天用例，当用 node server.js 命令启动聊天服务器并打开 http://localhost:4000/时，就会看到一个精简的界面，通过该界面可以向其他用户发送消息，也可以接收来自其他用户的消息。打开另一个浏览器窗口或选项卡来模拟实际中的多用户交互。

22.2.3 扩展聊天应用程序

此时，客户端和服务器之间的聊天交互协议还不够丰富——客户端和服务器都会发射某种类型的消息，并会对某种类型的消息做出响应。当应用程序中的用例变得更为复杂时，上述情形将发生变化。

例如，由内部套接字 ID 来识别用户并不方便，如果能在开始聊天之前，由用户名来识别用户就比较合适。

这可以通过新增一种客户端到服务器的 Socket.IO 消息类型来实现，这个消息的类型是 login。

服务器需要通过关联用户名与套接字连接来处理该消息，用户名就是 login 消息的内容。

```
socket.on('login', function(username) {
  socket.set('username', username, function(err) {
    if (err) { throw err; }
    socket.emit('serverMessage', 'Currently logged in as ' + username);
    socket.broadcast.emit('serverMessage', 'User ' + username +
      ' logged in');
  });
});
```

上述服务器端代码在套接字上设置了用户名，针对当前连接，采用 soket.set 在套接字上设置任意的键-值对，同时保留了你觉得有用的任意信息。

然后，需要从客户端发送 login 消息，为此，浏览器会向用户询问用户名，现在要用某个事件来触发该问题，这个事件可能是 login 服务器事件。

```
socket.on('login', function() {
  var username = prompt('What username would you like to use?');
  socket.emit('login', username);
});
```

之后，服务器需要在用户连接时发送 login 事件：

```
socket.on('login', function(username) {
  socket.set('username', username, function(err) {
    if (err) { throw err; }
    socket.emit('serverMessage', 'Currently logged in as ' + username);
    socket.broadcast.emit('serverMessage', 'User ' + username +
      ' logged in');
  });
});
```

并且服务器在用户连接成功时会发射 login 事件：

```
socket.emit('login');
```

现在，登录流程是这样的：用户进行连接时，服务器发送 login 事件，浏览器一旦接收到该事件，就会将向用户提示用户名，并通过 login 事件向服务器发送用户名。服务器对该事件进行处理，将用户名和套接字关联起来以备后用。

程序清单 22-6 中包含完整的服务器代码，程序清单 22-7 中包含完整的客户端代码。

程序清单 22-6：支持用户登录的 Socket.IO 聊天服务器

```
var httpd = require('http').createServer(handler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');

httpd.listen(4000);

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.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);
    socket.get('username', function(err, username) {
```

```
    if (! username) {
      username = socket.id;
    }
    socket.broadcast.emit('serverMessage', username + ' said: ' +
      content);
  });
});

socket.on('login', function(username) {
  socket.set('username', username, function(err) {
    if (err) { throw err; }
    socket.emit('serverMessage', 'Currently logged in as ' + username);
    socket.broadcast.emit('serverMessage', 'User ' + username +
      ' logged in');
  });
});

socket.emit('login');

});
```

程序清单 22-7：支持用户登录的 Socket.IO 聊天客户端

```
<html>
  <head>
    <title>Node.js WebSocket chat</title>
    <style type="text/css">
      #input {
        width: 200px;
      }
      #messages {
        position: fixed;
        top: 40px;
        bottom: 8px;
        left: 8px;
        right: 8px;
        border: 1px solid #EEEEEE;
        padding: 8px;
      }
    </style>
  </head>

  <body>
    Your message:
    <input type="text" id="input">

    <div id="messages"></div>

    <script src="http://localhost:4000/socket.io/socket.io.js"></script>
    <script type="text/javascript">
```

```
    var messagesElement = document.getElementById('messages');
    var lastMessageElement = null;

    function addMessage(message) {
      var newMessageElement = document.createElement('div');
      var newMessageText = document.createTextNode(message);

      newMessageElement.appendChild(newMessageText);
      messagesElement.insertBefore(newMessageElement,
        lastMessageElement);
      lastMessageElement = newMessageElement;
    }

    var socket = io.connect('http://localhost:4000');
    socket.on('serverMessage', function(content) {
      addMessage(content);
    });

    socket.on('login', function() {
      var username = prompt('What username would you like to use?');
      socket.emit('login', username);
    });

    var inputElement = document.getElementById('input');

    inputElement.onkeydown = function(keyboardEvent) {
      if (keyboardEvent.keyCode === 13) {
        socket.emit('clientMessage', inputElement.value);
        inputElement.value = '';
        return false;
      } else {
        return true;
      }
    };
  </script>

  </body>
</html>
```

现在将客户端代码保存为文件 index.html，同时将服务器代码保存为文件 app.js，然后运行服务器代码，如下所示：

```
$ node app.js
```

然后，打开几个浏览器窗口，并转向 http://localhost:4000，应该可以运行不同的聊天会话，这些会话彼此之间可以进行通信。

22.2.4　检测连接断开

现在，用户离开聊天的唯一方法就是关闭浏览器窗口，这会在服务器端的套接字上触

发 disconnect 事件。某个用户离开聊天时，能够让其他用户得到通知会非常有趣。

```
socket.on('disconnect', function() {
  socket.get('username', function(err, username) {
    if (! username) {
      username = socket.id;
    }
    socket.broadcast.emit('serverMessage', 'User ' + username + '
      disconnected');
  });
});
```

现在，当某个用户断开连接时，服务器会将该信息向其他连接到其上的客户端进行广播，程序清单 22-8 包含完整的服务器代码。

程序清单 22-8：检测连接断开的 Socket.IO 聊天服务器

```
var httpd = require('http').createServer(handler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');

httpd.listen(4000);

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.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);
    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.broadcast.emit('serverMessage', username + ' said:' +
        content);
    });
  });

  socket.on('login', function(username) {
    socket.set('username', username, function(err) {
```

```
    if (err) { throw err; }
    socket.emit('serverMessage', 'Currently logged in as ' + username);
    socket.broadcast.emit('serverMessage', 'User ' + username +
      ' logged in');
  });
});

socket.on('disconnect', function() {
  socket.get('username', function(err, username) {
    if (! username) {
      username = socket.id;
    }
    socket.broadcast.emit('serverMessage', 'User ' + username +
      ' disconnected');
  });
});

socket.emit('login');

});
```

可以将上面的修改保存为文件 app.js 并重启 Node 服务器，然后打开几个浏览器窗口进入聊天应用程序进行测试。如果关闭其中一个窗口，其他窗口都会得到通知。

22.2.5　将用户分隔到聊天室中

还可以提供几个聊天室让用户加入，即将聊天消息限制在加入聊天室的用户范围之内，可以通过记录哪个套接字在哪个聊天室中来实现这个目标，但是还好，Socket.IO 恰好支持这项功能。

但是，客户端首先必须允许用户加入一个聊天室，为此，浏览器命令行必须能够接受一些类似 iRC 协议中的命令，在 iRC 协议中，用户可以输入一条以字符/开头的命令，字符/后紧接着一个动词和一些参数，程序清单 22-9 中是允许这种命令的客户端代码。

程序清单 22-9：接受/j<room>命令的 Socket.IO 聊天客户端

```
<html>
  <head>
    <title>Node.js WebSocket chat</title>
    <style type="text/css">
      #input {
        width: 200px;
      }
      #messages {
        position: fixed;
        top: 40px;
        bottom: 8px;
        left: 8px;
        right: 8px;
```

```
      border: 1px solid #EEEEEE;
      padding: 8px;
    }
  </style>
</head>

<body>

  Your message:
  <input type="text" id="input">

  <div id="messages"></div>

  <script src="http://localhost:4000/socket.io/socket.io.js"></script>
  <script type="text/javascript">
    var messagesElement = document.getElementById('messages');
    var lastMessageElement = null;

    function addMessage(message) {
      var newMessageElement = document.createElement('div');
      var newMessageText = document.createTextNode(message);

      newMessageElement.appendChild(newMessageText);
      messagesElement.insertBefore(newMessageElement,
        lastMessageElement);
      lastMessageElement = newMessageElement;
    }

    var socket = io.connect('http://localhost:4000');
    socket.on('serverMessage', function(content) {
      addMessage(content);
    });

    socket.on('login', function() {
      var username = prompt('What username would you like to use?');
      socket.emit('login', username);
    });

    function sendCommand(command, args) {
      if (command === 'j') {
        socket.emit('join', args);
      } else {
        alert('unknown command: ' + command);
      }
    }

    function sendMessage(message) {
      var commandMatch = message.match(/^\/(\w*)(.*)/);
      if (commandMatch) {
        sendCommand(commandMatch[1], commandMatch[2].trim());
```

```
    } else {
      socket.emit('clientMessage', message);
    }
  }

  var inputElement = document.getElementById('input');

  inputElement.onkeydown = function(keyboardEvent) {
    if (keyboardEvent.keyCode === 13) {
      sendMessage(inputElement.value);
      inputElement.value = '';
      return false;
    } else {
      return true;
    }
  };
</script>

</body>
</html>
```

在上面的代码中，sendMessage 函数用于用户输入，如果用户的输入和某个命令匹配，就会调用 sendCommand 函数，否则就向服务器发送一条普通的文本消息。接下来，sendMessage 函数会对命令进行解析，并产生合适的服务器消息。在本例中只需要支持/j<room>命令，该命令可以让用户加入给定的聊天室。

然后，服务器必须解释 join 命令，并让用户加入聊天室。在加入聊天室时，用户可能已经在另一个聊天室中，在这种情况下用户需要离开先前加入的聊天室。同样，可以通过将聊天室名与套接字对象关联起来，以存储用户当前所处的聊天室。程序清单 22-10 中是修改后的服务器代码，修改后的代码可以进行上述处理。

程序清单 22-10：允许用户加入聊天室的 Socket.IO 聊天服务器

```
var httpd = require('http').createServer(handler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');

httpd.listen(4000);

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.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);

    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.get('room', function(err, room) {
        if (err) { throw err; }
        var broadcast = socket.broadcast;
        var message = content;
        if (room) {
          broadcast.to(room);
        }
        broadcast.emit('serverMessage', username + ' said: ' + message);
      });
    });
  });

  socket.on('login', function(username) {
    socket.set('username', username, function(err) {
      if (err) { throw err; }
      socket.emit('serverMessage', 'Currently logged in as ' + username);
      socket.broadcast.emit('serverMessage', 'User ' + username +
        ' logged in');
    });
  });

  socket.on('disconnect', function() {
    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.broadcast.emit('serverMessage', 'User ' + username +
        ' disconnected');
    });
  });

  socket.on('join', function(room) {
    socket.get('room', function(err, oldRoom) {
      if (err) { throw err; }

      socket.set('room', room, function(err) {
        if (err) { throw err; }
        socket.join(room);
```

```
        if (oldRoom) {
          socket.leave(oldRoom);
        }
        socket.get('username', function(err, username) {
          if (! username) {
            username = socket.id;
          }
        });

        socket.emit('serverMessage', 'You joined room ' + room);
        socket.get('username', function(err, username) {
          if (! username) {
            username = socket.id;
          }
          socket.broadcast.to(room).emit('serverMessage', 'User ' +
            username + ' joined this room');
        });
      });
    });
  });

  socket.emit('login');

});
```

在上面的代码中，当收到一条消息时，服务器就会检查用户是否已经加入聊天室。如果用户已经加入了聊天室，服务器就会通过 broadcast.to(room)来修改 broadcast 对象，将广播范围限制在给定的聊天室中。如果检查到该用户还没有进入聊天室，他发送的消息就会在所有尚未进入聊天室的用户中间进行广播。

可以将所做的修改保存为客户端和服务器文件，然后重启 Node 服务器，同时重新加载可能已经打开的聊天应用程序窗口。现在可以用类似/j chatroom1 的命令来加入聊天室。

22.2.6　使用名称空间

上述这个简单的聊天应用程序也许是大型网站的一部分，如果该网站提供了几个使用 Socket.IO 的近实时交互程序，就要十分小心，同时要分离所有的服务器及客户端事件名，以免它们发生冲突。

幸运的是，Socket.IO 允许使用名称空间来分离服务器和客户端上不同的应用程序，首先必须在客户端上连接到一个已被名称空间化的 Socket.IO URL：

```
var socket = io.connect('http://localhost:4000/chat');
```

然后，聊天服务器必须将套接字的范围限制为仅使用了该名称空间的套接字：

```
var chat = io.of('/chat');

chat.on('connection', function (socket) {
  socket.on('clientMessage', function(content) {
  // …
```

现在，可以轻松地使用相同的技术向客户端和服务器添加另一个应用程序，而不用担心事件名会发生冲突。

22.2.7 使用Redis分布运行服务器端应用程序

使用Socket.IO可以让所有用户采用一个Node进程进行连接，但是在实际的应用程序产品中，也许需要有多个Node进程来为应用程序提供服务。为了冗余和/或负载分配，也许在负载均衡器后要有一个计算机群集来运行应用程序服务器。

注意：如果在一组运行Socket.IO服务器的Node.js进程之前运行负载均衡器，需要使用的负载均衡器支持WebSockets。

如果想运行基于Node的Socket.IO服务器，我推荐你考虑node-http-proxy模块，该模块适应性很强，并且具有丰富的特性，可以在https://github.com/nodejitsu/node-http-proxy上查找到关于更多关于该模块的信息。

运行一组分布式进程的问题在于任意用户都可以在任意给定时刻连接到任意指定的Node.js进程，用户只能与连接到该服务器的其他用户进行通信，而连接到不同服务器的用户则无法访问。

幸好，Socket.IO具有可插拔的存储器，默认情况下它使用本地进程内存，但是可以将其设置成符合给定协议的任意存储器，例如，一个主流的选择是采用Redis数据库作为其存储器。

Redis是一个强大的内存键-值数据库，它具有一些先进的特性，其中之一就是支持发布-订阅，即客户端可以发布事件，而订阅者可以得到这些事件的通知。Socket.IO也内置了对Redis存储器的支持，但是需要安装redis模块：

```
$ npm install redis
```

现在需要安装的只是Redis服务器自身。

注意：如何安装Redis服务器超出了本书的范围，但是到官方网站http://redis.io/topics/quickstart上查阅相关文档就可以入门。

一旦安装并运行Redis服务器，就可以准备配置Socket.IO服务器以便让其使用Redis存储器：

```
var redis = require('redis'),
    RedisStore = require('socket.io/lib/stores/redis'),
```

```
    pub    = redis.createClient(),
    sub    = redis.createClient(),
    client = redis.createClient();

io.set('store', new RedisStore({
  redisPub : pub,
  redisSub : sub,
  redisClient : client
}));
```

即只要重启服务器就可以使用 Redis 了。如果需要安装多个服务器，那么 Redis 就会成为聊天消息的中心服务器，但是由于默认情况下 redis 模块是连接到本地主机上的，因此需要为 Redis 服务器配置服务器名：

```
var redisPort = 6379;
var redisHostname = 'my.host.name';

var redis = require('redis'),
    RedisStore = require('socket.io/lib/stores/redis'),
    pub    = redis.createClient(redisPort, redisHostname),
    sub    = redis.createClient(redisPort, redisHostname),
    client = redis.createClient(redisPort, redisHostname);
```

如果将 my.host.name 替换为运行 Redis 的服务器主机名，那么程序清单 22-11 包含了完整的服务器代码。

程序清单 22-11：使用 Redis 存储器的 Socket.IO 聊天服务器

```
var httpd = require('http').createServer(handler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');

var redisPort = 6379;
var redisHostname = 'my.host.name';

var redis = require('redis'),
    RedisStore = require('socket.io/lib/stores/redis'),
    pub    = redis.createClient(redisPort, redisHostname),
    sub    = redis.createClient(redisPort, redisHostname),
    client = redis.createClient(redisPort, redisHostname);

io.set('store', new RedisStore({
  redisPub : pub,
  redisSub : sub,
  redisClient : client
}));

httpd.listen(4000);

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);
  }
  );
}

var chat = io.of('/chat');

chat.on('connection', function (socket) {
  socket.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);

    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.get('room', function(err, room) {
        if (err) { throw err; }
        var broadcast = socket.broadcast;
        var message = content;
        if (room) {
          broadcast.to(room);
        }
        broadcast.emit('serverMessage', username + ' said: ' + message);
      });
    });
  });

socket.on('login', function(username) {
  socket.set('username', username, function(err) {
    if (err) { throw err; }
    socket.emit('serverMessage', 'Currently logged in as ' + username);
    socket.broadcast.emit('serverMessage', 'User ' + username +
      ' logged in');
  });
});

socket.on('disconnect', function() {
  socket.get('username', function(err, username) {
    if (! username) {
      username = socket.id;
    }
    socket.broadcast.emit('serverMessage', 'User ' + username +
```

```
      ' disconnected');
  });
});

socket.on('join', function(room) {
  socket.get('room', function(err, oldRoom) {
    if (err) { throw err; }

    socket.set('room', room, function(err) {
      if (err) { throw err; }
      socket.join(room);
      if (oldRoom) {
        socket.leave(oldRoom);
      }
      socket.get('username', function(err, username) {
        if (! username) {
          username = socket.id;
        }
      });
      socket.emit('serverMessage', 'You joined room ' + room);
      socket.get('username', function(err, username) {
        if (! username) {
          username = socket.id;
        }
        socket.broadcast.to(room).emit('serverMessage', 'User ' +
            username + ' joined this room');
      });
    });
  });
});

socket.emit('login');

});
```

22.3　本章小结

Socket.IO 是一个优秀的跨浏览器解决方案，为浏览器和服务器之间提供了近实时的双向通信。

可以通过对所有连接起来的套接字进行广播来将若干客户端连接起来，此外，还可以让客户端加入和离开特定的聊天室，并且可以选择只对特定的聊天室进行广播。

通过使用名称空间，可以让几个基于 Socket.IO 的应用在同一客户端和服务器应用程序中安全地共存。

此外，还可以通过使用 Redis 存储器，将一个 Socket.IO 应用程序分布到多个 Node.js 进程上运行，Redis 存储器采用 Redis 服务器作为通信中心。

第VI部分

连接数据库

- 第 23 章：使用 node-mysql 连接 MySQL 数据库
- 第 24 章：使用 Nano 连接 CouchDB 数据库
- 第 25 章：使用 Mongoose 连接 MongoDB 数据库

第23章

使用 node-mysql 连接 MySQL 数据库

本章提要：

- 从 Node 中连接 MySQL 数据库
- 借助 Node 的异步和非阻塞风格与数据库交换数据
- 避免安全和性能陷阱

与在其他很多编程环境中一样，迟早都要持久化应用程序产生的某些类型的数据，对于多用户的服务器-客户端应用程序更是如此，在这类应用程序中，一个用户提供的信息需要在稍后提交给另外一个或者多个用户。

虽然与在其他环境中一样，在 Node 中将数据存入普通文件中或者内存中能够运行得很好，但是当需要对大量数据进行结构化访问时，上述方法就不具备可伸缩性。

类似 MySQL 这样的关系数据库，通过将数据存储在硬盘上并用标准的 SQL 语言访问数据，为以结构化的方式存储和获取信息提供了可靠快速的解决方案。

如果 Node 应用程序需要能跟 MySQL 数据库通信并与其交换数据，下面几节有关使用 node-mysql 库、从 MySQL 数据库中高效读取数据以及以安全的方式传递数据的内容能够让你完成这些任务。

23.1 应用库与 MySQL 数据库进行连接和通信

因为安装和设置 MySQL 数据库并不在本书的学习范围之内，所以假设在本地主机或者其他可访问的主机上已经有一个正在运行的 MySQL 实例。

要能够连接到数据库，必须用如下所示的命令在本地目录中安装 node-mysql 包：

```
$ npm install mysql
```

一旦安装完成，就可以编写一个简单的应用程序来连接服务器，如程序清单 23-1 所示。

程序清单 23-1：连接和查询 MySQL 数据库

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

var client = mysql.createClient({
  host: 'localhost',
  user: 'root',
  password: 'root'
});

client.query(
  'SELECT "Hello, world!"',
  function (err, results, fields) {
    console.log(results);
    console.log(fields);
    client.end();
  }
);
```

在存储和运行这个程序清单之前，需要修改第 4 行、第 5 行和第 6 行，为访问 MySQL 服务器指定主机和身份凭证。如果 node-mysql 模块是以本地模式安装的，那么连接 MySQL 服务器时就不需要任何身份凭证，此时可以从 createClient 函数的参数选项中删除用户和密码字段。

然后可以将上述代码保存为文件 hello_world.js 并运行它：

```
$ node hello_world
[ { 'Hello, world!': 'Hello, world!' } ]
{ 'Hello, world!':
   { length: 35,
     received: 35,
     number: 2,
     type: 4,
     catalog: 'def',
     name: 'Hello, world!',
     charsetNumber: 192,
     fieldLength: 39,
     fieldType: 253,
     flags: 1,
     decimals: 31 } }
```

甚至通过这个基本的示例，都能看到典型的异步回调流程如何再次被用来使得数据库请求非阻塞化。在处理数据库请求时这一点尤其重要，因为——取决于查询和结果集合的大小——数据库请求有点慢，所以如果采用同步处理方式会导致应用程序的响应性存在风险隐患。在之后的章节中你将学习一种更为有效的从数据库获取结果的方法。

当使用 mysql.createClient()函数连接数据库时，可以提供以下一组选项：

- host：要连接的服务器的 IP 地址或者域名，如果忽略的话默认是 localhost。
- port：要连接的 TCP 端口，如果忽略的话默认是 3306。
- user：服务器用来进行身份验证的用户名，如果忽略的话是 root。
- password：服务器用来进行身份验证的密码，如果忽略的话，则尝试无密码的连接。
- database：成功连接上所使用的数据库，如果忽略的话就没有选择数据库，那么就必须用 SELECT 数据库查询语句选择一个数据库。
- debug：如果将该参数设置为 true 的话，*node-mysql* 模块就会将所有的输入和输出的数据包打印到控制台上，默认是 false。
- flags：MySQL 连接协议支持几个影响底层连接握手细节的标志位，例如使用的协议版本、是否采用压缩等等。因为 *node-mysql* 模块中的这些标志位具有合理的默认值，所以，除非服务器是以非标准的方式配置的，否则就不必设置这些标志位。在 http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol#Handshake_Initialization_Packet 可以看到可用标志位的列表。

你还没有查询真正的数据，为此需要创建一个带一张数据表的数据库，然后向数据表中添加数据行：

```
client.query('CREATE DATABASE node', function(err) {
  if (err) {
    client.end();
    throw err;
  }
});
```

上面的代码展示了错误如何触发回调函数，在本例中，如果出现了错误，就会关闭服务器的连接。可以通过多次运行该脚本来触发一个错误实例。因为数据库在代码第一次运行之后就已经存在了，所以它不能再次被创建，而且第二次查询会失败。

23.2　向数据库添加数据时请记住安全性

现在创建一张数据表并向其中插入两行数据：

```
client.query('USE node');

client.query('CREATE TABLE test ' +
             '(id INT(11) AUTO_INCREMENT, ' +
             ' content VARCHAR(255), ' +
             ' PRIMARY KEY(id))'
);

client.query('INSERT INTO test (content) VALUES ("Hello")');
client.query('INSERT INTO test (content) VALUES ("World")');

client.end();
```

上面的代码会运行得很好，但是存在一个问题，如果查询不是用这种方式创建的，而又没有对查询所使用的数据值进行直接的控制(也许是因为 content 的值是用户通过网页表单输入提供的)，应用程序就容易遭受 SQL 注入攻击。恶意用户会向应用程序提交数据，如果这些数据不经过滤就传递给数据库，就会破坏想要执行的 SQL 语句，并允许攻击者执行任意可能造成危害的语句。

下面是执行 SQL 注入的示例：

```
client.query('USE node');

var userInput = '"); DELETE FROM test WHERE id = 1; -- ';

client.query('INSERT INTO test (content) VALUES ("' + userInput + '")');

client.end();
```

要看出实际发生了什么有些复杂，但是它确实发生了。

编写上述代码的目的是提供一个由用户输入的文本字符串来查询 content 字段的值，当然，实际的用户输入只能被模拟，但是可以假定 userInput 的值是来自于某个 HTTP 表单中的输入域。

在这个例子中，攻击者并不提供简单的文本字符串，而代之以用结尾处的引号和括号来");提前结束查询，然后开始执行自定义查询，删除 id 为 1 的数据行。由于原来查询的第二部分——结尾的引号和括号——将导致无效的 SQL，所以攻击者在其前面加上了两个破折号作为前缀，结果这两个破折号之后的语句都被当作是注释而被忽略。

因此，服务器接收到的完整语句是：

```
INSERT INTO test (content) VALUES (""); DELETE FROM test WHERE id = 1; -- ")
```

这将导致会向表中插入新的一行，而 id 为 1 的那一行数据则会被删除。

你能够也应该在几个层次上保护应用程序免受这种类型的攻击，要做的最重要的事就是总是将传入的数据当作潜在有害的，尽早拒绝任何不能匹配“无害”严格定义的数据。

在应用程序中，有一个额外的预防措施，来保护由外部数据组成数据库查询部分免受攻击。通过使用可赋值的占位符，就可以保证查询中任何应该被视为值的部分确实被数据库解释为值，而绝不会被解释成其他东西，例如 DELETE 命令。

占位符能够起到这一作用，是因为不再将自定义命令与用户输入连接到一起组成最终的 SQL 语句(这样做会允许攻击者破坏 SQL 语句的结构)，这样一来，所编写的 SQL 语句只包含计划之中的命令，而不包含任何值。占位符只表示值的位置，而值本身是单独提供的，数据库将两者结合起来形成可执行的语句。

这样，查询就不会被当作一个整体处理，而是变得语义丰富——数据库知道查询的哪一部分应被当作值处理，而哪一部分仅仅就是值本身。看一下第一个例子，缺少这种语义就会允许攻击者执行一条命令，而不是输入一个值。

下面可以发布一条包含占位符和值替换的语句：

```
client.query('USE node');

var userInput = '"); DELETE FROM test WHERE id = 1; -- ';

client.query('INSERT INTO test (content) VALUES (?)', [userInput]);

client.end();
```

虽然用户的输入仍旧是恶意的，但是它再也不会对数据库造成危害，因为服务器知道 userInput 中的任何内容都应被视为 content 字段的值，所以查询不是以删除 id 为 1 的数据行结束，而是以插入一个新数据行结束，新数据行的 content 字段值为"); DELETE FROM test WHERE id = 1; --。

尽管也许不是你所希望的内容，但是这阻止了攻击行为。

要在 node-mysql 中进行赋值，就需要编写带引号的 SQL 语句，其中引号是可赋值的占位符，此外还要提供一个值数组来按顺序替换每个占位符。只要按照键值从低到高的顺序遍历值数组，就可以在多个占位符和值之间进行匹配。因此，下面的语句会导致写入相同的数据。

```
var data = [100, "the content"];

client.query('INSERT INTO test (id, content) VALUES (?, ?)', data);
```

这等同于

```
var data = [];
data[1] = "the content";
data[0] = 100;

client.query('INSERT INTO test (id, content) VALUES (?, ?)', data);
```

有时，在插入或者更新数据行时，需要获取某些信息，比如最后插入记录的 ID 或者查询影响的数据行数目，node-mysql 将这类信息提交给一个可选的回调函数：

```
client.query('INSERT INTO test (content) VALUES (?)', ['the content'],
function(err, info) {
if (err) {
return handle_error(err);
}
console.log(info.insertId);
}
);

client.query('UPDATE test SET content = ?', ['new content'], function(err,
  info) {
if (err) {
return handle_error(err);
}
console.log(info.affectedRows);
});
```

可以看到，像往常一样，上面代码中的回调函数所提供的第一个参数包含一个错误对象，以便处理出现的错误。如果没有出现错误，那么第二个参数包含一个对象，即查询的结果，如果向数据表中插入一个自动增长字段，那么该对象将包含一个 insertId 属性，该属性即自动增长字段的值。如果查询执行的是删除、更新或者移除数据行，那么 affectedRows 属性就包含受影响的数据行数。

23.3 高效读取数据

还可以用占位符从数据库中读取数据，node-mysql 模块用两种不同的方式提供结果集，其中，向查询调用提供回调函数是最简单的方式：

```
query = client.query(
  'SELECT id, content FROM test WHERE id IN (?, ?)',
  [1, 100], function(err, results, fields) {
    if (err) {
      throw err;
    }
    console.log(results);
  }
);
```

如下所示的输出就是查询结果：

```
[ { id: 1, content: 'the content' },
  { id: 100, content: 'the content' } ]
```

如你所见，在上面的代码中一口气接收了所有数据行，并将其当作一个对象数组，当查询返回大的结果集合时，这样就可能产生问题，因为 Node.js 必须在内存中缓存整个结果集合。

为了避免这个问题，可以通过应用 node-mysql 模块的数据流 API 逐行处理结果集合。

当使用 client.query 函数向服务器发送一个查询时，会返回一个 Query 对象，它会发射一些事件：error、field、row 和 end，可以监听这些事件。

可以利用这些事件，如下所示：

```
query = client.query('SELECT id, content FROM test WHERE id IN (?, ?)', [1,
100]);

query.on('error', function(err) {
  throw err;
});

query.on('field', function(field) {
  console.log('Received field:');
  console.log(field);
});
```

```
query.on('row', function(row) {
  console.log('Received row:');
  console.log(row);
});

query.on('end', function(result) {

  console.log('Finished retrieving results');

});
```

上述代码运行的结果如下：

```
Received field:
{ length: 38,
  received: 38,
  number: 2,
  type: 4,
  catalog: 'def',
  db: 'node',
  table: 'test',
  originalTable: 'test',
  name: 'id',
  originalName: 'id',
  charsetNumber: 63,
  fieldLength: 11,
  fieldType: 3,
  flags: 16899,
  decimals: 0 }
Received field:
{ length: 48,
  received: 48,
  number: 3,
  type: 4,
  catalog: 'def',
  db: 'node',
  table: 'test',
  originalTable: 'test',
  name: 'content',
  originalName: 'content',
  charsetNumber: 192,
  fieldLength: 765,
  fieldType: 253,
  flags: 0,
  decimals: 0 }
Received row:
{ id: 1, content: 'the content' }
Received row:
{ id: 100, content: 'the content' }
Finished retrieving results
```

为什么要将 result 参数传递给 end 事件的回调函数？只有在查询不会返回任何结果，并且包含 MySQL OK 数据包的内容时，result 参数才会被传递。

学习了流式查询 API 以及用占位符创建查询之后，就可以将前述内容组合到一起，形成一个完整的应用程序，如程序清单 23-2 所示。

程序清单 23-2：一个 MySQL 应用程序

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

var client = mysql.createClient({
  host: 'localhost',
  user: 'root',
  password: 'root',
});

client.query('DROP DATABASE IF EXISTS node');
client.query('CREATE DATABASE node');
client.query('USE node');

client.query('CREATE TABLE test ' +
           '(id INT(11) AUTO_INCREMENT, ' +
           ' content VARCHAR(255), ' +
           ' PRIMARY KEY(id))'
);

for (var i = 0; i < 10000; i++) {
  client.query('INSERT INTO test (content) VALUES (?)', ['content for row ' +
    (i + 1)]);
}

client.query('UPDATE test SET content = ? WHERE id >= ?', ['new content',
  9000],
  function(err, info) {
    console.log('Changed content of ' + info.affectedRows + ' rows');
  }
);

query = client.query('SELECT id, content FROM test WHERE id >= ? AND id <= ?',
  [8990, 9010]);

query.on('error', function(err) {
  throw err;
});

query.on('row', function(row) {
  console.log('Content of row #' + row.id + ' is: "' + row.content + '"');
});

query.on('end', function(result) {
```

```
  console.log('Finished retrieving results');
});

client.end();
```

再次重申，要在文件的开始处修改主机名和身份认证，然后将其保存为文件 app.js 并且运行：

```
$ node app.js
```

之后会输出如下信息：

```
Content of row #8990 is: "content for row 8990"
Content of row #8991 is: "content for row 8991"
Content of row #8992 is: "content for row 8992"
Content of row #8993 is: "content for row 8993"
Content of row #8994 is: "content for row 8994"
Content of row #8995 is: "content for row 8995"
Content of row #8996 is: "content for row 8996"
Content of row #8997 is: "content for row 8997"
Content of row #8998 is: "content for row 8998"
Content of row #8999 is: "content for row 8999"
Content of row #9000 is: "new content"
Content of row #9001 is: "new content"
Content of row #9002 is: "new content"
Content of row #9003 is: "new content"
Content of row #9004 is: "new content"
Content of row #9005 is: "new content"
Content of row #9006 is: "new content"
Content of row #9007 is: "new content"
Content of row #9008 is: "new content"
Content of row #9009 is: "new content"
Content of row #9010 is: "new content"
Finished retrieving results
```

警告：MySQL 协议是串行的，这意味着在使用一个连接时，是逐个执行每个查询的，如果创建的应用程序要要执行大量的 MySQL 操作，那么让它们互相等待并不是一个好的解决方案。

如果遇到上述情况，就需要创建多个到 MySQL 的连接，你可以为每个应用程序用户选择一个连接，但是根据并发用户的数量以及 MySQL 服务器的容量，这样做也许效果并不好。

此外，还可以创建一个连接池，通过连接池就会有多个到 MySQL 服务器的连接，并且能在这些连接之间分配查询。

23.4 本章小结

幸好有了 node-mysql 模块，与 MySQL 数据库交换数据才变得简单、高效。然而，在处理外部潜在恶意数据时，必须谨慎小心和密切关注。通过监听异步事件，大型结果集合可以被高效地逐行处理。

第24章

使用 Nano 连接 CouchDB 数据库

本章提要：

- 创建 CouchDB 数据库
- 创建和更新 CouchDB 视图
- 插入或更新文档
- 查询视图并显示结果
- 将二进制文件附加到文档上

在过去的几年里，一系列项目尝试通过提供不同的数据库管理系统的设计方法，来解决关系数据库的一些问题，这种发展趋势被称为 NoSQL，它不仅舍弃了查询语言，而且舍弃了关系模型，此外，NoSQL 在大多数情况下简化了模型，放宽了假设条件。

注意：例如，在大多数关系数据库中，当在事务内部进行一些修改时，可以保证所有的修改都是原子的，也就是说修改要不起作用，要不就都没有作用。在大多数 NoSQL 数据库中，由于更为简单和更少出错的并发模型，在插入或者更新几个文档时，无法在交换时保证原子性。

在 NoSQL 的“革新”大潮中，矗立着一个成功典范，即开源项目。CouchDB 是一个符合 Web 标准的数据库，它用 JSON 存储对象，并且用 JavaScript 进行查询。

注意：安装和学习如何全面使用 CouchDB 数据库超出了本书的范围，如果选择使用提供 CouchDB 虚拟主机(比如 IrisCouch)的互联网服务，就不需要在本地计算机上安装 CouchDB。

如果需要安装 CouchDB，可以在 http://couchdb.apache.org/上了解更多关于它的信息。

CouchDB 也采用 HTTP 进行客户端-服务器的通信，实际上，当在 Node 中使用 CouchDB 时，需要的不过是 Node HTTP 客户端 API 而已，最终可能要创建几种库代码来减少代码重复。

Nano 正是这样的一种库——它是一个第三方模块，建立在 Mikeal Roger 的 request 模块之上(第 13 章“创建 HTTP 请求”中有介绍)。Nano 中有一些辅助函数，这些辅助函数针对的是 CouchDB 中的大多数常用功能，并且这些辅助函数将抽象概念减至最少，这样有助于简化问题。

本章建立在第 22 章“使用 Socket.IO 创建通用的实时 Web 应用程序”中开发的基于 Socket.IO 的聊天应用程序之上，并且提供了消息持久化以及一些其他出色的应用程序。

24.1　安装 Nano

可以用 NPM 安装命令来安装 Nano，或者也可以将其添加到 package.json 清单中，如下所示：

```
{
  "name": "couch-socketio-chat",
  "version": "1.0.0",
  "dependencies": {
    "nano": "3.1.x",
    "socket.io": "0.9.x"
  }
}
```

除了 Nano 模块外，清单中还要有 Socket.IO 的依赖项，以便在 Socket.IO 聊天应用程序的基础上创建应用程序。

现在可以安装依赖项，如下所示：

```
$ npm install
```

注意：在第 22 章的最后，得到了一个有关 Redis 模块的依赖项，但是现在可以忽略该模块，因为本章仅会利用内存存储器来创建应用程序。如果以后希望将 Redis 作为 Socket.IO 的存储器，可以参考第 22 章。

可以利用第 22 章中的代码，复制包含客户端代码的 index.html 文件和包含服务器代码的 app.js 文件，本章正是要在这些代码的基础上创建应用程序。程序清单 24-1 展示了一个服务器示例。

程序清单 24-1：基于 Socket.IO 的服务器代码(本章创建应用程序的基础)

```
var httpd = require('http').createServer(handler);
var io = require('socket.io').listen(httpd);
var fs = require('fs');

httpd.listen(4000);

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);
    }
  );
}

var chat = io.of('/chat');

chat.on('connection', function (socket) {
  socket.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);

    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.get('room', function(err, room) {
        if (err) { throw err; }
        var broadcast = socket.broadcast;
        var message = content;
        if (room) {
```

```
        broadcast.to(room);
      }
      broadcast.emit('serverMessage', username + ' said: ' + message);
    });
  });
});

socket.on('login', function(username) {
  socket.set('username', username, function(err) {
    if (err) { throw err; }
    socket.emit('serverMessage', 'Currently logged in as ' + username);
    socket.broadcast.emit('serverMessage', 'User ' +
      username + ' logged in');
  });
});

socket.on('disconnect', function() {
  socket.get('username', function(err, username) {
    if (! username) {
      username = socket.id;
    }
    socket.broadcast.emit('serverMessage', 'User ' +
      username + ' disconnected');
  });
});

socket.on('join', function(room) {
  socket.get('room', function(err, oldRoom) {
    if (err) { throw err; }

    socket.set('room', room, function(err) {
      if (err) { throw err; }
      socket.join(room);
      if (oldRoom) {
        socket.leave(oldRoom);
      }
      socket.get('username', function(err, username) {
        if (! username) {
          username = socket.id;
        }
      });
      socket.emit('serverMessage', 'You joined room ' + room);
      socket.get('username', function(err, username) {
        if (! username) {
          username = socket.id;
        }
        socket.broadcast.to(room).emit('serverMessage', 'User ' +
         username + '
         joined this room');
      });
```

```
        });
      });
    });

    socket.emit('login');

  });
```

注意：如果在客户端连接服务器的过程中出现问题，原因可能在于要进行少量的修改，修改的方法已经在第 22 章的最后提及，即连接到 http://localhost:4000/chat，而不是 http://localhost:4000/：

```
var socket = io.connect('http://localhost:4000/chat');
```

24.2　连接和创建数据库

现在需要创建一个数据库连接，与大多数数据库管理服务器不同，CouchDB 采用的不是面向连接的协议，而是普通的 HTTP 协议，HTTP 是一种无状态协议，这意味着每个事务都会在单独的 HTTP 请求中执行。

从这个意义上来说，Nano 中不存在到数据库的真正“连接”，只有指向 CouchDB 服务器和数据库的引用，这意味着任何连接问题都只会在执行个别请求时发生。

注意：安装和管理 CouchDB 服务器超出了本书的范围，但是一些互联网服务可用来提供远程 CouchDB 服务器，其中一种服务就是 IrisCouch，可以在 www.iriscouch.com/上访问它。

可以用 IrisCouch 来免费创建和使用小型数据库(如果用途是正当的)，注册之后，可以得到如下所示的 URL：

```
http://myiriscouchserver.iriscouch.com
```

一旦 CouchDB 服务器运行起来，就需要将 nano 模块导入到 app.js 文件中：

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

然后需要创建一个数据库服务器引用：

```
var couchdb = nano('https://myiriscouchserver.iriscouch.com');
```

应该用 CouchDB 数据库的 URL 替换掉默认的 URL，如果数据库需要身份认证才能创建数据库以及插入或者修改文档，就应该在 URL 中添加身份认证，如下所示：

```
var couchdb =
 nano('https://username:password@myiriscouchserver.iriscouch.com');
```

下面要保证数据库 chat 存在，为此要进行如下所示的操作：

```
couchdb.db.create('chat', function(err) {
 if (err) { console.error(err); }
 var db = couchdb.use('chat');
 // ...
});
```

现在只要数据库创建成功，就会调用一个回调函数，可以将其余的服务器初始化代码放到这个回调函数中。

注意，你只是对错误进行了记录，但不管如何代码还会继续执行下去，这是因为如果数据库已经存在，CouchDB 就会抗议并产生一个错误，而你会在回调函数中获得这个错误。如果将程序清单 24-2 保存为文件 app.js，然后结束应用程序并重新启动它，就会看到这个错误的出现。这样做会强制代码试着创建一个已经存在的数据库，从而产生相应的 CouchDB 错误：

程序清单 24-2：基于 Socket.IO 的服务器在初始化时创建 CouchDB 客户端

```
var nano = require('nano');
var couchdb = nano('https://myiriscouchserver.iriscouch.com');

couchdb.db.create('chat', function(err) {
 if (err) { console.error(err); }
 var chatDB = couchdb.use('chat');

 var httpd = require('http').createServer(handler);
 var io = require('socket.io').listen(httpd);
 var fs = require('fs');

 httpd.listen(4000);

 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);
   }
```

```
  );
}

var chat = io.of('/chat');

chat.on('connection', function (socket) {
  socket.on('clientMessage', function(content) {
    socket.emit('serverMessage', 'You said: ' + content);

    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.get('room', function(err, room) {
        if (err) { throw err; }
        var broadcast = socket.broadcast;
        var message = content;
        if (room) {
          broadcast.to(room);
        }
        broadcast.emit('serverMessage', username + ' said: ' + message);
      });
    });
  });

  socket.on('login', function(username) {
    socket.set('username', username, function(err) {
      if (err) { throw err; }
      socket.emit('serverMessage', 'Currently logged in as ' + username);
      socket.broadcast.emit('serverMessage', 'User ' + username + ' logged
        in');
    });
  });

  socket.on('disconnect', function() {
    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.broadcast.emit('serverMessage', 'User ' + username +
        ' disconnected');
    });
  });

  socket.on('join', function(room) {
    socket.get('room', function(err, oldRoom) {
      if (err) { throw err; }

      socket.set('room', room, function(err) {
        if (err) { throw err; }
```

```
        socket.join(room);
        if (oldRoom) {
          socket.leave(oldRoom);
        }
        socket.get('username', function(err, username) {
          if (! username) {
            username = socket.id;
          }
        });
        socket.emit('serverMessage', 'You joined room ' + room);
        socket.get('username', function(err, username) {
          if (! username) {
            username = socket.id;
          }
          socket.broadcast.to(room).emit('serverMessage', 'User ' +
            username + ' joined this room');
        });
      });
    });
  });

  socket.emit('login');

});

});
```

再次启动应用程序，将获得如下所示的错误信息：

```
$ node app.js
{ [Error: Unspecified error]
  name: 'Error',
  scope: 'couch',
  status_code: 412,
  'status-code': 412,
  request:
   { method: 'PUT',
     headers:
      { 'content-type': 'application/json',
        accept: 'application/json' },
     uri: 'https://myiriscouchserver.iriscouch.com/chat',
     jar: false,
     callback: [Function] },
  headers:
   { date: 'Fri, 13 Jul 2012 14:21:55 GMT',
     'content-type': 'application/json',
     'cache-control': 'must-revalidate',
     'status-code': 412,
     uri: 'https://myiriscouchserver.iriscouch.com/chat' },
  errid: 'non_200',
  error: 'file_exists',
```

```
  reason: 'The database could not be created, the file already exists.',
  description: 'Unspecified error',
  stacktrace:
   [ 'Error: Unspecified error',
     '    at Request._callback (/Users/pedroteixeira/projects/wiley-node/
code/24/no de_modules/nano/nano.js:290:39)',
     '    at Request.callback (/Users/pedroteixeira/projects/wiley-node/
code/24/nod e_modules/nano/node_modules/request/main.js:120:22)',
     '    at Request.<anonymous> (native)',
     '    at Request.emit (events.js:70:17)',
     '    at Request.<anonymous> (/Users/pedroteixeira/projects/wiley-
node/code/24/node_modules/nano/node_modules/request/main.js:555:16)',
     '    at Request.emit (events.js:67:17)',
     '    at IncomingMessage.<anonymous> (/Users/pedroteixeira/projects/
wiley-node/code/24/node_modules/nano/node_modules/request/main.js:517:14)',
     '    at IncomingMessage.emit (events.js:88:20)',
     '    at HTTPParser.onMessageComplete (http.js:137:23)' ] }
```

可以看到该错误有一个 HTTP 状态码 412 以及一个句子："The database could not be created, the file already exists."。然后要更进一步，检查错误的状态码是否和期望的一致：

```
if (err && err.status_code !== 412) {
   throw err;
}
```

现在得到了一个指向 chat 数据库的引用，该引用在第 9 行被发送：

```
var chatDB = couchdb.use('chat');
```

可以使用这个数据库引用发送查询和命令。

24.3　存储文档

下面要将消息存入数据库，当用户发送一条消息时，需要将其存储在 chat 数据库中。

```
chat.on('connection', function (socket) {
   socket.on('clientMessage', function(content) {
     socket.emit('serverMessage', 'You said: ' + content);

     socket.get('username', function(err, username) {
       if (! username) {
         username = socket.id;
       }
       socket.get('room', function(err, room) {
         if (err) { throw err; }
         var broadcast = socket.broadcast;
         var message = content;
         if (room) {
           broadcast.to(room);
```

```
    }

    var messageDoc = {
      when: Date.now(),
      from: username,
      room: room,
      message: content
    };

    chatDB.insert(messageDoc, function(err) {
      if (err) { console.error(err); }
    });

    broadcast.emit('serverMessage', username + ' said: ' + message);
  });
});
});
```

在上面的代码中，创建了一个简单的文档，该文档包含消息以及一些元数据，元数据是指这样一些信息——消息是何时发送的？是谁发送的？在哪个聊天室中？

注意：虽然在本例中只是创建了一个简单的普通文档，但是在 CouchDB 中，你想让文档有多复杂它就有多复杂，需要它多少层嵌套它就有多少层嵌套。

24.4 创建和使用 CouchDB 视图

既然已经能够让所有消息都被持久化保存，那么如果用户在每次连接或者进入聊天室时能够收到最新消息就好了。CouchDB 只是一个简单的键-值存储器，只要给出正确的 ID，就能用如下所示的代码获得一个文档：

```
chatDB.get(messageID, function(err, doc) {
  console.log('got document!', doc);
});
```

但是这没什么作用，也许需要获得指定的聊天室中最新的 10 条消息，为此，必须创建更为复杂的查询。

创建复杂查询的方法是用 map-reduce 函数创建索引，map-reduce 函数是驻留在 CouchDB 服务器中的 JavaScript 函数。为了创建映射函数，需要考虑将每个文档映射成什么键值，以便让 CouchDB 提交查询。在这种情况下，也许想用聊天室对消息编制索引，所以如下所示的映射函数会很有效：

```
function(doc) {
  emit([doc.room, doc.when], doc);
}
```

当有新文档出现或者文档被修改，就会调用这个映射函数，并且每次它都会发送一个文档，发送的文档将一个键作为索引，键包含聊天室的标识符和时间戳，这样就允许 CouchDB 按照聊天室和创建时间对索引进行排序，你不仅可以获得某个聊天室内的所有消息，还可以得到两个给定时间点之间所创建的消息。

现在，可以将服务器的初始化过程封装为一个函数 startServer，该函数会在数据库初始化操作结束时被调用，如果确定已成功创建了 chat 数据库，可以增加一些初始化代码。

现在要将映射函数插入到设计文档中，以便让 CouchDB 创建对应的视图。

```
couchdb.db.create('chat', function(err) {
  if (err && err.status_code !== 412) {
    throw err;
  }

  var designDoc = {
    language: "javascript",
    views: {
      by_room: {
        map: couchMapReduce.toString()
      }
    }
  };

  var chatDB = couchdb.use('chat');
(function insertOrUpdateDesignDoc() {
    chatDB.insert(designDoc, '_design/designdoc', function(err) {
      if (err) {
        if (err.status_code === 409) {
          chatDB.get('_design/designdoc', function(err, ddoc) {
            if (err) { return console.error(err); }
            designDoc._rev = ddoc._rev;
            insertOrUpdateDesignDoc();
          });
        } else {
          return console.error(err);
        }
      }
      startServer();
    });
  }());
});
```

在上面的代码中创建了一个设计文档，设计文档是 CouchDB 中定义的一种特殊的文档类型。在设计文档中，除了别的外，还要定义视图，每一个视图都由一个映射函数(以及一个可选的累加函数，这里不需要它)组成。然后将该设计文档插入到路径_design/designdoc 中，在 CouchDB 中，路径_design 是存储针对某个数据库的所有设计文档的场所，这里创建的设计文档被命名为 designdoc。

注意插入设计文档可能会失败，尤其当 CouchDB 中已经存在设计文档时更是如此，在这种情况下，错误用 409 Conflict HTTP 响应状态码来表示。此时需要更新设计文档的版本，以便消除版本冲突，然后再次尝试插入。

注意：尽管本书并不打算详细讲授 CouchDB，但是需要理解一些基本原理，其中之一就是 CouchDB 版本管理的方式。

在 CouchDB 中，每个文档都有一个版本号，为了更新某个文档，需要传入最新的版本号。如果没有这样做，CouchDB 就会返回一个 409 Conflict HTTP 响应状态码，这样可以避免两个客户端在互不知情的情况下重复更新同一个文档。

例如，假如客户端 A 和 B 都有一个版本为 1 的文档 D，A 和 B 都对文档进行了修改，并在同一时刻进行更新，文档 D 具有一个._rev 属性，该属性包含的内容类似 1-76e555f270676c7f20a73726875c50b8。

服务器首先处理 A 的请求，并创建了第 2 版的 D 文档，版本号是 2-676c7f20a73726875c50b876e555f270。当客户端 B 试图更新文档 D 时，引用的仍然是版本 1(加上一个校验和)。而现在数据库中的文档 D 的版本号已被修正为 2，所以客户端 B 的更新操作会被拒绝，并返回 409 Conflict 状态码。

程序清单 24-3 包含了完整的服务器代码，在代码中针对视图增加了必要的设计文档。

程序清单 24-3：基于 Socket.IO 的服务器按需创建数据库和视图

```
var nano = require('nano');
var couchdb = nano('https://myiriscouchserver.iriscouch.com');

var couchMapReduce = function (doc) {
  emit([doc.room, doc.when], doc);
};

couchdb.db.create('chat', function(err) {
  if (err && err.status_code !== 412) {
    throw err;
  }

  var designDoc = {
    language: "javascript",
    views: {
      by_room: {
        map: couchMapReduce.toString()
      }
```

```
    }
  };

  var chatDB = couchdb.use('chat');

(function insertOrUpdateDesignDoc() {
    chatDB.insert(designDoc, '_design/designdoc', function(err) {
      if (err) {
        if (err.status_code === 409) {
          chatDB.get('_design/designdoc', function(err, ddoc) {
            if (err) { return console.error(err); }
            designDoc._rev = ddoc._rev;
            insertOrUpdateDesignDoc();
          });
        } else {
          return console.error(err);
        }
      }
      startServer();
    });
  }());
});

function startServer() {
  var chatDB = couchdb.use('chat');

  var httpd = require('http').createServer(handler);
  var io = require('socket.io').listen(httpd);
  var fs = require('fs');

httpd.listen(4000);

  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);
      }
    );
  }

  var chat = io.of('/chat');

  chat.on('connection', function (socket) {
    socket.on('clientMessage', function(content) {
```

```
socket.emit('serverMessage', 'You said: ' + content);

socket.get('username', function(err, username) {
  if (! username) {
    username = socket.id;
  }
  socket.get('room', function(err, room) {
    if (err) { throw err; }
    var broadcast = socket.broadcast;
    var message = content;
    if (room) {
      broadcast.to(room);
    }

    var messageDoc = {
      when: Date.now(),
      from: username,
      room: room,
      message: content
    };

    chatDB.insert(messageDoc, function(err) {
      if (err) { console.error(err); }
    });

    broadcast.emit('serverMessage', username + ' said: ' + message);
  });
});
});

socket.on('login', function(username) {
  socket.set('username', username, function(err) {
    if (err) { throw err; }
    socket.emit('serverMessage', 'Currently logged in as ' + username);
    socket.broadcast.emit('serverMessage', 'User ' + username + ' logged
      in');
  });
});

socket.on('disconnect', function() {
  socket.get('username', function(err, username) {
    if (! username) {
      username = socket.id;
    }
    socket.broadcast.emit('serverMessage', 'User ' + username +
      ' disconnected');
  });
});

socket.on('join', function(room) {
```

```
socket.get('room', function(err, oldRoom) {
  if (err) { throw err; }

  socket.set('room', room, function(err) {
    if (err) { throw err; }
    socket.join(room);
    if (oldRoom) {
      socket.leave(oldRoom);
    }
    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
    });
    socket.emit('serverMessage', 'You joined room ' + room);
    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.broadcast.to(room).emit('serverMessage', 'User ' +
        username + ' joined this room');
    });
  });
 });
});

  socket.emit('login');

 });
}
```

现在可以用如下所示的 URL 来查询聊天室 room1 中最新的 10 条消息：

```
/chat/_design/designdoc/_view/by_room?end_key=[%22room1%22,0]&start_key
=[%22room1%22,9999999999999]&limit=10&descending=true
```

上面的 URL 通过 chat 数据库中的_room 视图对 CouchDB 进行查询，它可以按照升序获取键值在["room1", 0]和["room1", 9999999999999]之间的所有对象，本例中限制为 10 条。这是视图上的一个范围查询，通过这个查询所获得的文档都有一个键，键的第一部分是 room1，第二部分可以从 0(起始时间)变化到 9999999999999 (将来的某个时间)。对视图的调用可以放在 sendBackLog 函数中，该函数会创建查询并将其发送到客户端套接字：

```
function sendBackLog(socket, room) {
  var getOptions = {
    start_key: JSON.stringify([room, 9999999999999]),
    end_key: JSON.stringify([room, 0]),
    limit: 10,
    descending: true
  };
```

```
  chatDB.get('_design/designdoc/_view/by_room', getOptions,
   function(err, results) {
    var messages = results.rows.reverse().map(function(res) {
      return res.value;
    });
    socket.emit('backlog', messages);
  });
}
```

sendBackLog 函数在用户登录和进入聊天室时就会被调用，一旦来自 CouchDB 的响应到达时，就会得到一个对象，这个对象包括文档数目和时间的偏移值，在本例中并不需要这些值，在响应对象中，还可以得到 rows 属性中的文档，进而汇集这类文档，并在 backlog 事件中将这类文档发送给套接字。

你也许注意到不再发送字符串形式的消息，而是发送包含若干属性的消息文档，这些属性包含消息文本、消息作者、所在聊天室和时间戳。

下一步需要修改前端代码来接收文档格式的消息，而不是字符串格式的消息。通过这种方式，就可以通过分别对消息的组成部分进行风格化处理，以增强消息的显示效果。 现在，输出消息的前端代码如下所示：

```
function addMessage(message) {
      var newMessageElement = document.createElement('div');

      newMessageElement.className = "message";

      if (message.when) {
        message.when = new Date(message.when);
      }

      ['when', 'from', 'message'].forEach(function(prop) {
        var newPart = document.createElement('span');
        newPart.className = prop;
        var messageText = message[prop];
        if (prop !== 'message') {
          messageText = '[' + messageText + ']';
          if (prop === 'from' && message[prop] === username) {
            newPart.className += ' self';
          }
        }
        messageText += ' ';
        var newPartText = document.createTextNode(messageText);
        newPart.appendChild(newPartText);
        newMessageElement.appendChild(newPart);
      });
```

同样，为了对消息内容进行风格化处理，需要增加一些 CSS 定义，并用不同的颜色来表明消息的各组成部分：

```
.message .when {
  color: grey;
}
.message .from {
  color: red;
}
.message .from.self {
  color: green;
}
```

现在，程序清单 24-4 包含完整的前端代码。

程序清单 24-4：对消息进行格式化的基于 Socket.IO 的聊天客户端的新版本

```
<html>
  <head>
    <title>Node.js WebSocket chat</titl
    <style type="text/css">
      #input {
        width: 200px;
      }
      #messages {
        position: fixed;
        top: 40px;
        bottom: 8px;
        left: 8px;
        right: 8px;
        border: 1px solid #EEEEEE;
        padding: 8px;
      }
      .message .when {
        color: grey;
      }
      .message .from {
        color: red;
      }
      .message .from.self {
        color: green;
      }

    </style>
  </head>
  <body>

    Your message:
    <input type="text" id="input">

    <div id="messages"></div>

    <script src="http://localhost:4000/socket.io/socket.io.js"></script>
```

```
<script type="text/javascript">
  var messagesElement = document.getElementById('messages');
  var lastMessageElement = null;
  var username;

  function addMessage(message) {
    var newMessageElement = document.createElement('div');

    newMessageElement.className = "message";

    if (message.when) {
      message.when = new Date(message.when);
    }

    ['when', 'from', 'message'].forEach(function(prop) {
      var newPart = document.createElement('span');
      newPart.className = prop;
      var messageText = message[prop];
      if (prop !== 'message') {
        messageText = '[' + messageText + ']';
        if (prop === 'from' && message[prop] === username) {
          newPart.className += ' self';
        }
      }
      messageText += ' ';
      var newPartText = document.createTextNode(messageText);
      newPart.appendChild(newPartText);
      newMessageElement.appendChild(newPart);
    });

    messagesElement.insertBefore(newMessageElement, lastMessageElement);
    lastMessageElement = newMessageElement;
  }

  var socket = io.connect('http://localhost:4000/chat');
  socket.on('serverMessage', function(content) {
    addMessage(content);
  });

  socket.on('login', function() {
    username = prompt('What username would you like to use?');
    socket.emit('login', username);
  });

  socket.on('backlog', function(messages) {
    messages.forEach(addMessage);
  });

  function sendCommand(command, args) {
    if (command === 'j') {
```

```
    socket.emit('join', args);
  } else {
    alert('unknown command: ' + command);
  }
}

function sendMessage(message) {
  var commandMatch = message.match(/^\/(\w*)(.*)/);
  if (commandMatch) {
    sendCommand(commandMatch[1], commandMatch[2].trim());
  } else {
    socket.emit('clientMessage', message);
  }
}

var inputElement = document.getElementById('input');

inputElement.onkeydown = function(keyboardEvent) {
  if (keyboardEvent.keyCode === 13) {
    sendMessage(inputElement.value);
    inputElement.value = '';
    return false;
  } else {
    return true;
  }
  };
</script>

</body>
</html>
```

现在，需要修改服务器代码，针对服务器发送的每条消息应用消息格式，这意味着必须针对每个发送消息的函数都进行精细的修改，由此形成的服务器端代码如程序清单 24-5 所示。

程序清单 24-5：发送格式化消息文档的基于 Socket.IO 的聊天服务器的新版本

```
var nano = require('nano');
var couchdb = nano('https://myiriscouchserver.iriscouch.com');

var couchMapReduce = function (doc) {
  emit([doc.room, doc.when], doc);
};

couchdb.db.create('chat', function(err) {
  if (err && err.status_code !== 412) {
    throw err;
  }

var designDoc = {
```

```
    language: "javascript",
    views: {
      by_room: {
        map: couchMapReduce.toString()
      }
    }
  };

  var chatDB = couchdb.use('chat');

  (function insertOrUpdateDesignDoc() {
    chatDB.insert(designDoc, '_design/designdoc', function(err) {
      if (err) {
        if (err.status_code === 409) {
          chatDB.get('_design/designdoc', function(err, ddoc) {
            if (err) { return console.error(err); }
            designDoc._rev = ddoc._rev;
            insertOrUpdateDesignDoc();
          });
        } else {
          return console.error(err);
        }
      }
      startServer();
    });
  }());
});

function startServer() {
  var chatDB = couchdb.use('chat');

  var httpd = require('http').createServer(handler);
  var io = require('socket.io').listen(httpd);
  var fs = require('fs');

  httpd.listen(4000);

  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);
      }
    );
  }
```

```
function sendBackLog(socket, room) {
  var getOptions = {
    start_key: JSON.stringify([room, 99999999999999]),
    end_key: JSON.stringify([room, 0]),
    limit: 10,
    descending: true
  };

  chatDB.get('_design/designdoc/_view/by_room', getOptions,
    function(err, results) {
      var messages = results.rows.reverse().map(function(res) {
        return res.value;
      });
      socket.emit('backlog', messages);
    });
}

var chat = io.of('/chat');

chat.on('connection', function (socket) {
  socket.on('clientMessage', function(content) {

    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.get('room', function(err, room) {
        if (err) { throw err; }
        var broadcast = socket.broadcast;
        var message = content;
        if (room) {
          broadcast.to(room);
        }

        var messageDoc = {
          when: Date.now(),
          from: username,
          room: room,
          message: content
        };

        socket.emit('serverMessage', messageDoc);

        chatDB.insert(messageDoc, function(err) {
          if (err) { console.error(err); }
        });

        broadcast.emit('serverMessage', messageDoc);
      });
```

```
  });
});

socket.on('login', function(username) {
  socket.set('username', username, function(err) {
    if (err) { throw err; }
    var message = {
      from: username,
      message: 'Logged in',
      when: Date.now()
    };
    socket.emit('serverMessage', message);
    socket.broadcast.emit('serverMessage', message);

    sendBackLog(socket, null);

  });
});

socket.on('disconnect', function() {
  socket.get('username', function(err, username) {
    if (! username) {
      username = socket.id;
    }
    var message = {
      from: username,
      message: 'disconnected',
      when: Date.now()
    };
    socket.broadcast.emit('serverMessage', message);
  });
});

socket.on('join', function(room) {
  socket.get('room', function(err, oldRoom) {
    if (err) { throw err; }

    socket.set('room', room, function(err) {
      if (err) { throw err; }
      socket.join(room);
      if (oldRoom) {
        socket.leave(oldRoom);
      }
      socket.get('username', function(err, username) {
        if (! username) {
          username = socket.id;
        }
        var message = {
          from: username,
          message: 'joined room ' + room + '. Fetching backlog...',
```

```
            when: Date.now()
          };
          socket.emit('serverMessage', message);
          socket.broadcast.to(room).emit('serverMessage', message);
        });

        sendBackLog(socket, room);
      });
    });
    });

    socket.emit('login');

  });
}
```

24.5　将文件附加到 CouchDB 文档上

假设允许所有用户定义一个在消息中显示的头像，用户在登录之后上传一个应用网络格式的简单图像文件，服务器就会将该文件存入 CouchDB。

第一个修改之处就是要提供给用户一个上传文件的表单，为简单起见，表单位于屏幕底部，程序清单 24-6 包含修改后的前端代码。

程序清单 24-6：A new version of the Socket.IO-based chat client that presents avatar

```
<html>
  <head>
    <title>Node.js WebSocket chat</title>
    <style type="text/css">
      #input {
        width: 200px;
      }
      #messages {
        position: fixed;
        top: 40px;
        bottom: 50px;
        left: 8px;
        right: 8px;
        border: 1px solid #EEEEEE;
        padding: 8px;
        overflow: hidden;
      }
      form {
        position: fixed;
        bottom: 0;
      }
      .message .when {
```

```
    color: grey;
  }
  .message .from {
    color: red;
  }
  .message .from.self {
    color: green;
  }
  .message .avatar {
    width: 80px;
  }

  </style>
</head>

<body>

  Your message:
  <input type="text" id="input" />

  <div id="messages"></div>

  <form action="/avatar" enctype="multipart/form-data"
    method="POST" target="avatar-target" id="avatar-form">
    <label for="avatar">Set avatar:</label>
    <input type="file" id="avatar" name="avatar" />
    <input type="submit" />
  </form>

  <iframe style="width:0;height:0;visibility:hidden"
    id="avatar-target"></iframe>

  <script src="http://localhost:4000/socket.io/socket.io.js"></script>
  <script type="text/javascript">
    var messagesElement = document.getElementById('messages');
    var lastMessageElement = null;
    var username;

    function addMessage(message) {
      var newMessageElement = document.createElement('div');

      newMessageElement.className = "message";

      if (message.when) {
        message.when = new Date(message.when);
      }

      message.avatar = '/avatar?username=' +
        encodeURIComponent(message.from);
```

```
    ['avatar', 'when', 'from', 'message'].forEach(function(prop) {
      var newPart = document.createElement('span');
      newPart.className = prop;
      var messageText = message[prop];
      if (prop !== 'message') {
        messageText = '[' + messageText + ']';
        if (prop === 'from') {
          if (message[prop] === username) {
            newPart.className += ' self';
          }
        } else if (prop === 'avatar') {
          var avatarImage = document.createElement('img');
          avatarImage.src = message.avatar;
          avatarImage.className = 'avatar';
          newPart.appendChild(avatarImage);
          messageText = '';
        }
      }
      messageText += ' ';
      var newPartText = document.createTextNode(messageText);
      newPart.appendChild(newPartText);
      newMessageElement.appendChild(newPart);
    });

    messagesElement.insertBefore(newMessageElement,
      lastMessageElement);
    lastMessageElement = newMessageElement;
  }

  var socket = io.connect('http://localhost:4000/chat');
  socket.on('serverMessage', function(content) {
    addMessage(content);
  });

  socket.on('login', function() {
    username = prompt('What username would you like to use?');
    socket.emit('login', username);
  });

  socket.on('backlog', function(messages) {
    messages.forEach(addMessage);
  });

  function sendCommand(command, args) {
    if (command === 'j') {
      socket.emit('join', args);
    } else {
      alert('unknown command: ' + command);
    }
  }
```

```
function sendMessage(message) {
  var commandMatch = message.match(/^\/(\w*)(.*)/);
  if (commandMatch) {
    sendCommand(commandMatch[1], commandMatch[2].trim());
  } else {
    socket.emit('clientMessage', message);
  }
}

var inputElement = document.getElementById('input');

inputElement.onkeydown = function(keyboardEvent) {
  if (keyboardEvent.keyCode === 13) {
    sendMessage(inputElement.value);
    inputElement.value = '';
    return false;
  } else {
    return true;
  }
};
</script>

<script>
  var form = document.getElementById('avatar-form');
  form.onsubmit = function() {
    if (! username) {
      alert('You must first log in')
    } else {
      form.action = '/avatar?username=' + encodeURIComponent(username);
      form.submit();
      console.log('submitted form');
    }
    return false;
  }
</script>

</body>
</html>
```

在上面的代码中，创建了一个表单，用这个表单将头像文件提交进文档中隐藏的内嵌框架中，如果不想离开页面，又想提交几部分数据时(例如有一个或者多个文件要上传时)，就可以使用该技巧。

最后一部分代码会在提交表单之前修改 URL，将用户名传入查询字符串，以便让服务器知道在和哪个用户进行交互。

注意：表单很容易被黑客攻击，用户可以轻松地将图像提交给其他用户，因为站点没有受到密码保护，所以安全性需求非常宽松。

如果想查看基于会话的身份认证的示例，请参见第 21 章“使用 Express.js 创建 Web 应用程序”。

服务器需要接收上传的文件，并将其插入 CouchDB 里的用户数据库中，首先，需要确认 users 数据库已被创建，创建 users 数据库的方法和创建 chat 数据库类似，如下所示：

```
couchdb.db.create('chat', function(err) {
  if (err && err.status_code !== 412) {
    throw err;
  }

  couchdb.db.create('users', function(err) {
    if (err && err.status_code !== 412) {
      throw err;
    }

  //...
});
```

现在接收和解析上传的文件，为此要用 Felix Geisendörfer 提供的第三方库 formidable，将其添加到 package.json 清单文件中：

```
{
  "name": "couch-socketio-chat",
  "version": "1.0.0",
  "dependencies": {
    "nano": "3.1.x",
    "socket.io": "0.9.x",
    "formidable": "1.0.x",
    "bufferedstream": "1.0.x"
  }
}
```

你还需要 bufferedstream 模块，该模块会缓存请求数据，并在需要数据时使数据可用。既然已完成对 package.json 文件的修改，就可以安装这两个缺失的模块，如下所示：

```
$ npm install
```

现在需要修改 handler 函数来处理新的请求：

```
function handler(req, res) {
  var username;
  if (req.method === 'POST' && req.url.indexOf('/avatar') === 0) {
```

```
var currentUserDocRev;

console.log('got avatar');

var bufferedRequest = new BufferedStream();
bufferedRequest.headers = req.headers;
bufferedRequest.pause();
req.pipe(bufferedRequest);

// parse username
console.log(url.parse(req.url).query);
username = url.parse(req.url, true).query.username;

userDB.insert({username: username}, username, function(err, user) {
  if (err) {
    if (err.status_code === 409) {
      userDB.get(username, function(err, user) {
        if (err) {
          console.error(err);
          res.writeHead(500);
          return res.end(JSON.stringify(err));
        }
        currentUserDocRev = user._rev;
        bufferedRequest.resume();

      });
      return;
    } else {
      res.writeHead(500);
      return res.end(JSON.stringify(err));
    }
  }
  console.log('username inserted, rev = ', user.rev);
  currentUserDocRev = user.rev;
  console.log('currentUserDocRev:', currentUserDocRev);
  bufferedRequest.resume();

});

// handle avatar upload

var form = new formidable.IncomingForm();
form.encoding = 'utf8';
form.parse(bufferedRequest);

form.onPart = function(part) {
  if (part.name !== 'avatar') {
    return;
  }
```

```
var attachment = userDB.attachment.insert(
   username, 'avatar', null, part.mime, {rev: currentUserDocRev});

part.pipe(attachment);

attachment.on('error', function(err) {
  console.error(err);
  res.writeHead(500);
  return res.end(JSON.stringify(err));
});

  attachment.on('end', function() {
    res.end();
  });
};

} else if (req.url.indexOf('/avatar') === 0){
  // serve the avatar
  username = url.parse(req.url, true).query.username;
  userDB.attachment.get(username, 'avatar').pipe(res);
} else {
  // serve the index page
  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);
    }
  );
}
}
```

当出现针对 URL /avatar 的 POST 请求时，应该实例化 formidable 表单来解析请求的内容，但是首先需要暂停请求——bufferedstream 正是在此处派上用场。将请求数据传入缓冲流，目前缓冲流被设置为暂停状态，这可以保证缓冲流在恢复流之前将保持缓冲数据，而不会将数据发送出去。

然后，确定在 users 数据库中针对用户名已经创建了 CouchDB 文档，如果已经存在了这样一个用户，会得到 409 Conflict 状态码，并且一定会得到用户文档的版本号。如果已创建了数据库，就可以恢复请求数据流，这就会激活前面创建的 formidable 处理程序来上传文件。

只要 formidable 模块开始解析请求中包含头像的部分，就会激活如下所示的代码：

```
form.onPart = function(part) {
  if (part.name !== 'avatar') {
```

```
    return;
  }
  var attachment = userDB.attachment.insert(
      username, 'avatar', null, part.mime, {rev: currentUserDocRev});

  part.pipe(attachment);

  attachment.on('error', function(err) {
    console.error(err);
    res.writeHead(500);
    return res.end(JSON.stringify(err));
  });

  attachment.on('end', function() {
    res.end();
  });
};
```

在上面的代码中，用 nano db.attachment.insert(documentID, attachment name, mime_type, options)创建了一个附件，但是该语句其实并没有进行任何工作，而是返回了一个可写流，让你能够用如下所示的代码将文件传入其中：

```
part.pipe(attachment);
```

一旦附件流结束，头像附件就完成了上传，同时也结束了响应。

现在，前端代码会以/avatar?username=abc 的形式创建服务器的头像 URL，浏览器希望通过这种 URL 返回图像，所需要做的只是从 CouchDB 中传送图像，如下所示：

```
// 提交头像
username = url.parse(req.url, true).query.username;
userDB.attachment.get(username, 'avatar').pipe(res);
```

程序清单 24-7 包含服务器代码的新版本，需要重启服务器并刷新聊天页面：

程序清单 24-7：presents avatars 的新版基于 Socket.IO 的聊天服务器

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

var nano = require('nano');
var couchdb = nano('https://myiriscouchserver.iriscouch.com');

var formidable = require('formidable');
var BufferedStream = require('bufferedstream');

var couchMapReduce = function (doc) {
  emit([doc.room, doc.when], doc);
};

couchdb.db.create('chat', function(err) {
  if (err && err.status_code !== 412) {
```

```
  throw err;
}

couchdb.db.create('users', function(err) {
  if (err && err.status_code !== 412) {
    throw err;
  }

  var designDoc = {
    language: "javascript",
    views: {
      by_room: {
        map: couchMapReduce.toString()
      }
    }
  };

  var chatDB = couchdb.use('chat');

  (function insertOrUpdateDesignDoc() {
    chatDB.insert(designDoc, '_design/designdoc', function(err) {
      if (err) {
        if (err.status_code === 409) {
          chatDB.get('_design/designdoc', function(err, ddoc) {
            if (err) { return console.error(err); }
            designDoc._rev = ddoc._rev;
            insertOrUpdateDesignDoc();
          });
          return;
        } else {
          return console.error(err);
        }
      }
      startServer();
    });
  }());
});
});

function startServer() {
  console.log('starting server');
  var chatDB = couchdb.use('chat');
  var userDB = couchdb.use('users');

  var httpd = require('http').createServer(handler);
  var io = require('socket.io').listen(httpd);
  var fs = require('fs');

  httpd.listen(4000);
```

```
function handler(req, res) {
  var username;
  if (req.method === 'POST' && req.url.indexOf('/avatar') === 0) {

    var currentUserDocRev;

    console.log('got avatar');

    var bufferedRequest = new BufferedStream();
    bufferedRequest.headers = req.headers;
    bufferedRequest.pause();
    req.pipe(bufferedRequest);

    // parse username
    console.log(url.parse(req.url).query);
    username = url.parse(req.url, true).query.username;

    userDB.insert({username: username}, username, function(err, user) {
      if (err) {
        if (err.status_code === 409) {
          userDB.get(username, function(err, user) {
            if (err) {
              console.error(err);
              res.writeHead(500);
              return res.end(JSON.stringify(err));
            }
            currentUserDocRev = user._rev;
            bufferedRequest.resume();

          });
          return;
        } else {
          res.writeHead(500);
          return res.end(JSON.stringify(err));
        }
      }
      console.log('username inserted, rev = ', user.rev);
      currentUserDocRev = user.rev;
      console.log('currentUserDocRev:', currentUserDocRev);
      bufferedRequest.resume();

    });
    // handle avatar upload

    var form = new formidable.IncomingForm();
    form.encoding = 'utf8';
    form.parse(bufferedRequest);

    form.onPart = function(part) {
      if (part.name !== 'avatar') {
```

```
      return;
    }
    var attachment = userDB.attachment.insert(
       username, 'avatar', null, part.mime, {rev: currentUserDocRev});

    part.pipe(attachment);

    attachment.on('error', function(err) {
      console.error(err);
      res.writeHead(500);
      return res.end(JSON.stringify(err));
    });

    attachment.on('end', function() {
      res.end();
    });
  };

} else if (req.url.indexOf('/avatar') === 0){
  // serve the avatar
  username = url.parse(req.url, true).query.username;
  userDB.attachment.get(username, 'avatar').pipe(res);
} else {
  // serve the index page
  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);
    }
  );
 }
}


function sendBackLog(socket, room) {
  var getOptions = {
    start_key: JSON.stringify([room, 99999999999999]),
    end_key: JSON.stringify([room, 0]),
    limit: 10,
    descending: true
  };

  chatDB.get('_design/designdoc/_view/by_room', getOptions,
    function(err, results) {
      var messages = results.rows.reverse().map(function(res) {
        return res.value;
```

```
      });
      socket.emit('backlog', messages);
    });
}

var chat = io.of('/chat');

chat.on('connection', function (socket) {
  socket.on('clientMessage', function(content) {

    socket.get('username', function(err, username) {
      if (! username) {
        username = socket.id;
      }
      socket.get('room', function(err, room) {
        if (err) { throw err; }
        var broadcast = socket.broadcast;
        var message = content;
        if (room) {
          broadcast.to(room);
        }

        var messageDoc = {
          when: Date.now(),
          from: username,
          room: room,
          message: content
        };

        socket.emit('serverMessage', messageDoc);

        chatDB.insert(messageDoc, function(err) {
          if (err) { console.error(err); }
        });

        broadcast.emit('serverMessage', messageDoc);
      });
    });
  });

  socket.on('login', function(username) {
    socket.set('username', username, function(err) {
      if (err) { throw err; }
      var message = {
        from: username,
        message: 'Logged in',
        when: Date.now()
      };

      socket.emit('serverMessage', message);
```

```
    socket.broadcast.emit('serverMessage', message);

    sendBackLog(socket, null);

  });
});

socket.on('disconnect', function() {
  socket.get('username', function(err, username) {
    if (! username) {
      username = socket.id;
    }
    var message = {
      from: username,
      message: 'disconnected',
      when: Date.now()
    };
    socket.broadcast.emit('serverMessage', message);
  });
});

socket.on('join', function(room) {
  socket.get('room', function(err, oldRoom) {
    if (err) { throw err; }

    socket.set('room', room, function(err) {
      if (err) { throw err; }
      socket.join(room);
      if (oldRoom) {
        socket.leave(oldRoom);
      }
      socket.get('username', function(err, username) {
        if (! username) {
          username = socket.id;
        }
        var message = {
          from: username,
          message: 'joined room ' + room + '. Fetching backlog...',
          when: Date.now()
        };
        socket.emit('serverMessage', message);
        socket.broadcast.to(room).emit('serverMessage', message);
      });

      sendBackLog(socket, room);
    });
  });
});
```

```
    socket.emit('login');

  });
}
```

24.6 本章小结

可以使用 Nano 连接 CouchDB 服务器，Nano 是一个第三方模块，通过 NPM 安装。

可以用 Nano 创建数据库，还可以通过插入包含映射函数和累加函数的设计文档，来创建文档上的视图。

还可以利用创建的视图插入文档，并对文档进行查询。

CouchDB 还允许文档拥有二进制文件附件，并且 Nano 可以让你对这些附件进行写入和读取操作。

第25章

使用 Mongoose 连接 MongoDB 数据库

本章提要：

- 安装和使用 Mongoose
- 定义模式和模型
- 理解 Mongoose 的数据类型
- 使用模式验证
- 查找、插入和更新文档
- 给查询结果标记分页
- 定义索引
- 应用取值器、赋值器和虚属性
- 引用记录
- 用高级方法丰富模型和文档对象

面向文档的数据库在最近几年重获流行，并处于 NoSQL 运动的前锋和中心。这种类型的数据库与关系型数据库不同，它不需要事先指定数据结构，相反你可以根据项目需要有组织地改变数据结构。与关系数据库的另一个重要的不同之处在于文档是原子单元，可以根据需要保证其完整性，因此不再需要跨表联合和事务。

MongoDB 是一个很流行的面向文档的数据库，可以让你使用强大的查询语言执行动态查询。

在 MonogoDB 中，数据被组织起来以便 MonogoDB 能保持一组集合，每个集合中都有一组文档，而每个文档又有一组字段。字段是一个键-值对，其中键是字符串，而值是一种基本类型(比如字符串、整数、浮点数、时间戳、二进制等等)，或者是嵌入式文档，或者是任何数据类型的集合。

MonogeDB 查询可以用 JavaScript 文档描述，比如说有一个用户集合，想在其中查找所有名为“John”、姓为“Doe”的用户，就可以使用如下所示的查询语句：

```
{name: {first: 'John', last: 'Doe'}}
```

还可以使用如下所示的查询语句查找所有姓为“Doe”的用户：

```
{'name.last': 'Doe'}
```

还可以使用正则表达式来代替静态值。例如，要查找所有姓以“D”开头的用户，可以使用：

```
{'name.last': /^D/}
```

这对于数组值也同样有效，例如，如果在一个文章集合中查找那些关键词为 node.js 的文章，可以使用如下所示的查询语句：

```
{keywords: 'node.js'}
```

还可以通过使用如下所示的查询语句，匹配至少有一个关键词与关键词集合相符的所有文章：

```
{keywords: {'$in' : ['node.js', 'javascript']}
```

有若干与此类似的操作符，可以在查询中使用这些操作符，以便查找所有字段 count 的值大于 10 并小于等于 100 的所有文档：

```
{count: {'$gt' : 10, '&lte': 100}
```

注意：本书并不打算让你精通安装和查询 MongoDB，这是其他图书的目标。

如果还没有在系统中安装 MongoDB，并且希望安装它，可以利用 MongoDB 的主页 http://www.mongodb.org/上提供的信息。

另一个选择是利用 MongoDB 的外部服务，比如 https://mongohq.com 上 MongoHQ 提供的服务。

25.1　安装 Mongoose

在本章中，将创建访问 MongoDB 服务器的代码，为此，在 Node 中将会用到第三方模块 MongoDB，Mongoose 允许采用对象模型来访问 MongoDB。

在第 21 章“使用 Express.js 创建 Web 应用程序”中，用到了几个路由监听器来管理用户，这些路由监听器就是管理用户数据库的公共接口。这个数据库是一个伪内存对象，显然无法使其持久化，仅能使其适用于原型。但是，既然可以通过 Mongoose 来使用 MongoDB 数据库，就可以用一个真正的持久化数据访问层来代替伪内存对象。

让我们从将第 21 章中的应用程序代码复制到新的项目目录中开始，现在需要修改应用程序的清单文件 pakage.json，向其中添加 Mongoose 依赖项，如程序清单 25-1 所示。

程序清单 25-1：应用程序的 package.json 清单文件

```
{
   "name": "application-name"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
     "express": "2.5.11"
    , "jade": ">= 0.0.1"
    , "mongoose": ">=2.7.0"
  }
}
```

修改后的 pakage.json 文件表明正在创建的脚本需要 Mongoose 版本高于或者等于 2.7.0。现在可以用如下所示的命令来安装 Mongoose：

```
$ npm install
...
mongoose@2.7.1 ./node_modules/mongoose
├── hooks@0.2.1
└── mongodb@1.0.2 (bson@0.0.6)
```

25.2　理解 Mongoose 如何使用模型封装对数据库的访问

Mongoose 是一个对数据库交互对象建模的工具，一个数据库有多个集合，每个集合又有多个文档。同一集合中的两个文档不必具有相似的数据结构，但是如果说有一个“用户”集合，这个集合中包含用户记录，集合中的所有文档都是相似的。

你要在 Mongoose 中定义模式，每个模式都包含一个字段以及字段约束列表，所有字段都有类型以及由类型限定的约束，比如最小值、字段值能否为空、字段值是否在集合的所有文档中唯一、是否需要验证规则等等。

模型表示数据库到使用模式的集合的连接。

25.3 连接 MongoDB 数据库

首先，需要连接到 MongoDB 服务器，无论这个服务器是在本地安装的还是在互联网上订阅的外部服务。为此，需要指定一个指向 MongoDB 数据库的 URL，如下所示：

```
mongodb://username:password@hostname:port/database
```

如果是连接本地计算机上的 MongoDB，而服务器不需要密码，在这种情况下，只要使用如下所示的 URL：

```
mongodb://localhost/database
```

然后，就可以用 Mongoose 连接数据库，为此，要向文件 app.js 中添加如下代码：

```
var dbURL = 'mongodb://localhost/database';
var db = require('mongoose').connect(dbURL);
```

如果还没有为应用程序创建 MongoDB 数据库，首先需要创建一个数据库。

25.4 定义模式

现在，需要创建模式模块来定义用户文档的外观，可以用 data/schemas 目录来放置模式模块，在该目录中要创建一个文件 user.js，如程序清单 25-2 所示：

程序清单 25-2：data/schemas/user.js 中的用户模式

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

var UserSchema = new mongoose.Schema({
  username: String,
  name: String,
  password: String
});

module.exports = UserSchema;
```

在上面的代码中定义了一个基本模式，这个模式有三个字段，这三个字段都是字符串类型。在本章的后面，将学习如何改进该模式，使其更有用，并且能够支持更复杂的文档。

25.5 定义模型

现在，需要定义一个模型，将用户集合映射到数据库中，同样可以将这个模型定义为文件 data/models/user.js 下的一个模块，如程序清单 25-3 所示。

程序清单 25-3：data/models/user.js 中的用户模型

```
var mongoose = require('mongoose');
var UserSchema = require('../schemas/user');

var User = mongoose.model('User', UserSchema);

module.exports = User;
```

现在，要在路由监听器中使用该模型，但是，首先需要更新路由中间件，使其支持新的用户文档模型。首先，在 routes/middleware/load_user.js 中创建一个模块，该模块定义了一个加载用户的中间件，如程序清单 25-4 所示。

程序清单 25-4：使用用户模型加载用户路由中间件

```
var User = require('../../data/models/user');

function loadUser(req, res, next) {
  User.findOne({username: req.params.name}, function(err, user) {
    if (err) {
      return next(err);
    }
    if (! user) {
      return res.send('Not found', 404);
    }
    req.user = user;
    next();
  });
}

module.exports = loadUser;
```

在上面的代码中，用到了用户模型对象中提供的 findOne 方法，该方法的第一个参数是查询参数，在本例中该参数要和请求参数 name 严格匹配。第二个参数是一个回调函数，这个回调函数会在出现错误或者判断操作结束时被调用。如果出现了错误，就会转而调用下一个回调函数。如果在用户集合中没有发现匹配的用户，就会回应一个状态码 404 Not Found，并停止执行。如果找到了匹配的用户，就将其赋值给 req.user 属性以备路由监听器能在稍后使用。

现在可以修改用户的路由行为，如程序清单 25-5 所示。

程序清单 25-5：使用用户 Mongoose 模型的用户路由

```
/*
 * 用户路由
 */

var User = require('../data/models/user');
```

```
var notLoggedIn = require('./middleware/not_logged_in');
var loadUser = require('./middleware/load_user');
var restrictUserToSelf = require('./middleware/restrict_user_to_self');

module.exports = function(app) {

  app.get('/users', function(req, res,next){
    User.find({}, function(err, users) {
      if (err) {
        return next(err);
      }
      res.render('users/index', {title: 'Users', users: users});
    });
  });

  app.get('/users/new', notLoggedIn, function(req, res) {
    res.render('users/new', {title: "New User"});
  });

  app.get('/users/:name', loadUser, function(req, res, next){
    res.render('users/profile', {title: 'User profile', user: req.user});
  });

  app.post('/users', notLoggedIn, function(req, res, next) {
    User.findOne({username: req.body.username}, function(err, user) {
      if (err) {
        return next(err);
      }
      if (user) {
        return res.send('Conflict', 409);
      }
      User.create(req.body, function(err) {
        if (err) {
          return next(err);
        }
        res.redirect('/users');
      });
    });
  });

  app.del('/users/:name', loadUser, restrictUserToSelf,
    function(req, res, next) {
      req.user.remove(function(err) {
        if (err) { return next(err); }
        res.redirect('/users');
      });

  });

};
```

现在，要逐个分析路由监听器上所做的修改，首先是针对列表中用户的路由监听器：

```
app.get('/users', function(req, res){
  User.find({}, function(err, users) {
    if (err) {
      return next(err);
    }
    res.render('users/index', {title: 'Users', users: users});
  });
});
```

在上面的代码中，用到了模型中的 find 方法，并向其传递一个空查询对象，这意味着 MongoDB 应该查找每一个用户并返回他/她的信息，这里至少存在两个问题：首先是没有对结果进行排序，MongoDB 采用的是自然排序，而不是指定顺序，自然排序是指文档在内部存储的顺序，这很可能是用户所不希望的。可以按照如下所示的方法对查询结果进行排序。

```
User.find({}).sort('name', 1).exec(function(err, users) {
  if (err) {
    return next(err);
  }
  res.render('users/index', {title: 'Users', users: users});
});
```

在上面的代码中用到了查询链式 API，在这个 API 中可以定义约束、指定排序顺序，然后在单独的创建者方法调用中执行查询，在这里指定应该根据 name 属性对结果进行升序排序(因此值为 1)，如果用-1 替换 1，则对结果进行降序排序。

还可以根据其他字段进行排序，只要在执行查询之前将字段添加到查询对象中即可。比如说，如果想根据年龄进行降序排序，然后根据名字进行升序排序，则应使用如下所示的代码：

```
User.find({})
  .sort('age', -1)
  .sort('name', 1)
  .exec(function(err, users) { ...
```

监听器代码中存在的第二个问题是因为要求 MongoDB 取回集合中的所有对象，所以查询耗费的时间会随着用户集合的增长而增长。

用户界面列表需要分页，这意味着需要对用户列表进行切分，可以使用 skip 和 limit 选项只获取一页检索数据：

```
app.get('/users', function(req, res, next){
  var page = req.query.page && parseInt(req.query.page, 10) || 0;
  User.find({})
    .sort('name', 1)
    .skip(page * maxUsersPerPage)
    .limit(maxUsersPerPage)
```

```
    .exec(function(err, users) {
      if (err) {
        return next(err);
      }
      res.render('users/index', {title: 'Users', users: users, page:
        page});
    });
});
```

在上面的代码中，可以根据请求的 URL 中的查询字符串得到当前页面，如果没有查询字符串，就使用默认值 0。然后让 MongoDB 跳过最前面的 x 个元素——x 是之前页面上的用户数目——并且将结果限制为所希望的每个页面上的最多用户数。

注意：为了能让代码运行起来，需要在模块范围内定义 maxElementsPerPage 配置变量，并让该变量的值符合你的需求：

```
var maxUserPerPage = 5;
```

然后，需要将当前页面传递给显示模板，以便让用户界面知道是用户位于哪个页面，并可以创建页面链接“前一页”和“后一页”。现在，需要修改模板 views/users/index.jade 来接纳分页结果，如程序清单 25-6 所示。

程序清单 25-6：views/users/index.jade 中具有分页的用户列表

```
h1 Users

p
  a(href="/users/new") Create new profile

ul
  - for(var username in users) {
    li
      a(href="/users/" + encodeURIComponent(username))=
        users[username].name - };

- if (page > 0) {
  a(href="?page=" + (page - 1)) Previous
   
- }

a(href="?page=" + (page + 1)) Next
```

但是上面的代码中仍然存在一个小问题——下一页上不存在用户时仍然会显示“下一页”页面链接，为修正这个问题，要统计集合中所有用户的数目，并指出还剩下多少页。可以在 routes/users.js 中的路由监听器中完成这个修改：

```
app.get('/users', function(req, res, next){
  var page = req.query.page  && parseInt(req.query.page, 10) || 0;
  User.count(function(err, count) {
    if (err) {
      return next(err);
    }

    var lastPage = (page + 1) * maxUsersPerPage >= count;

    User.find({})
      .sort('name', 1)
      .skip(page * maxUsersPerPage)
      .limit(maxUsersPerPage)
      .exec(function(err, users) {
        if (err) {
          return next(err);
        }
        res.render('users/index', {
          title: 'Users',
          users: users,
          page: page,
          lastPage: lastPage
        });
      });
  });
```

在上面的代码中，通过模型的 count 方法来请求文档的数目，一旦这个操作结束，就要判断是否还有剩余的页面，然后将判断结果传递到模板中。

需要修改用户索引视图模板，如程序清单 25-7 所示：

程序清单 25-7：views/users/index.jade 中具有分页的用户列表的新版本

```
h1 Users

p
  a(href="/users/new") Create new profile

ul
  - users.forEach(function(user) {
    li
      a(href="/users/" + encodeURIComponent(user.username))= user.name
  - });

- if (page > 0) {
  a(href="?page=" + (page - 1)) Previous
   
- }
```

```
- if (! lastPage) {
  a(href="?page=" + (page + 1)) Next
- }
```

在路由监听器代码中，还要避免一种情况——没有理由让两个查询(计数和获得用户)串行执行，而应该让它们并行执行，从而减少响应用户请求耗费的时间。

注意：使用 Node.js 的众多优势之一是能轻松地创建并行 I/O 请求，好好利用这一点！

为此，可以利用在第 19 章“控制回调流”中学习的 async 模块，首先需要将该模块添加到清单 package.json 中，如程序清单 25-8 所示。

程序清单 25-8：包含 async 依赖项的新版 package.json 清单

```
{
    "name": "application-name"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.11"
    , "jade": ">= 0.0.1"
    , "mongoose": ">=2.7.0"
    , "async": "0.1.22"
  }
}
```

然后，安装这个缺失的依赖项：

```
$ npm install
async@0.1.22 ./node_modules/async
```

准备导入该模块并在路由监听器中使用它，将模块的导入放在 routes/users.js 的最上面：

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

现在，可以使用 async.parallel 来并行化请求，如下所示：

```
app.get('/users', function(req, res, next){
  var page = req.query.page  && parseInt(req.query.page, 10) || 0;
  async.parallel([
      function(next) {
        User.count(next);
      },

      function(next) {
        User.find({})
          .sort('name', 1)
```

```
      .skip(page * maxUsersPerPage)
      .limit(maxUsersPerPage)
      .exec(next);
    }
  ],

  // 最后一个回调
  function(err, results) {

    if (err) {
      return next(err);
    }

    var count = results[0];
    var users = results[1];

    var lastPage = (page + 1) * maxUsersPerPage >= count;

    res.render('users/index', {
      title: 'Users',
      users: users,
      page: page,
      lastPage: lastPage
    });

  }
 );
});
```

最后一个回调函数(async.parallel 的最后一个参数)会在两个查询都完成时或者发生错误时被调用，同时能获得一个数组，数组中的元素是各个查询的结果。如果没有发生错误，那么数组中的第一个元素包含的是用户数目，第二个元素包含的是用户列表。

下面，来看一看用户创建路由监听器：

```
app.post('/users', notLoggedIn, function(req, res, next) {
  User.findOne({username: req.body.username}, function(err, user) {
    if (err) {
      return next(err);
    }
    if (user) {
      return res.send('Conflict', 409);
    }
    User.create(req.body, function(err) {
      if (err) {
        return next(err);
      }
      res.redirect('/users');
    });
  });
});
```

在上面的代码中，检查了用户名是否被占用，如果没有被占用，就在用户模型上用 create 方法创建用户，这段代码中至少存在一个问题——另外一个请求可能会在检查给定的用户名是否存在与数据库实际创建用户的时间点之间用相同的用户名创建用户，这样就可能会导致具有相同用户名的多个用户被插入到数据库，这种情况不允许发生，因为你正是基于用户名来查找用户的。

要避免该问题，就要向用户名模式添加一个选项，声明 username 字段的唯一性，如程序清单 25-9 所示。

程序清单 25-9：data/schemas/user.js 中具有唯一索引的新版用户模式

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

var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: String,
  password: String
});

module.exports = UserSchema;
```

当采用该选项时，Mongoose 会确保在 username 字段上创建了唯一的索引，现在可以删除检查用户名是否存在的代码，只需要依赖 MongoDB 返回文件 routes/user.js 中的错误：

```
app.post('/users', notLoggedIn, function(req, res, next) {
  User.create(req.body, function(err) {
    if (err) {
      if (err.code === 11000) {
        res.send('Conflict', 409);
      } else {
        next(err);
      }
      return;
    }
    res.redirect('/users');
  });
});
```

当 MongoDB 输出错误时，Mongoose 会捕获错误码并将其传入错误对象，在上面的代码中对错误码做出了解释，当错误码表示键值重复错误时，就会显示 409 Conflict HTTP 状态码。

注意：在网站 mongodb.org 上，可以查找到有关 MongoDB 错误代码的更多信息。

为了能让登录起作用，必须将登录过程整合到 Mongoose 当中，为此，要修改文件 routes/session.js，如下所示：

```
/*
 * 会话路由
 */

var User = require('../data/models/user');
var notLoggedIn = require('./middleware/not_logged_in');

module.exports = function(app) {

  app.dynamicHelpers({
    session: function(req, res) {
      return req.session;
    }
  });

  app.get('/session/new', notLoggedIn, function(req, res) {
    res.render('session/new', {title: "Log in"});
  });

  app.post('/session', notLoggedIn, function(req, res) {
    User.findOne({username: req.body.username, password:
      req.body.password},
      function(err, user) {
        if (err) {
          return next(err);
        }

        if (user) {
          req.session.user = user;
          res.redirect('/users');
        } else {
          res.redirect('/session/new');
        }
      });
  });

  app.del('/session', function(req, res, next) {
    req.session.destroy();
    res.redirect('/users');
  });

};
```

既然用户路由可以和 MongoDB 协调工作了，就可以继续完善用户模式来整合验证器、文档嵌套等等。

25.5.1 使用验证器

当用户在网站上注册新的用户帐号时，他们可以在输入域中输入任意数据——Mongoose没有验证机制。但是，可以将约束引入用户模式。例如，可以引入一个新的字段email，要求其必须与一个正则表达式匹配，如程序清单25-10所示。

程序清单25-10：data/schemas/user.js中带有新的受验证的e-mail字段的新版用户模式

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

// simple but incomplete email regexp:
var emailRegexp = /.+\@.+\..+/;

var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: String,
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp

  }
});

module.exports = UserSchema;
```

现在看看验证器是否有效，需要向views/users/new.jade中的用户创建表单添加email输入域：

```
h1 New User

form(method="POST", action="/users")
  p
    label(for="username") Username<br />
    input#username(name="username")
  p
    label(for="name") Name<br />
    input#name(name="name")
  p
    label(for="email") Email<br />
    input#email(name="email")
  p
    label(for="password") Password<br />
    input#password(type="password", name="password")
  p
    label(for="bio") Bio<br />
    textarea#bio(name="bio")
  p
    input(type="submit", value="Create")
```

现在可以重启服务器测试网络应用程序：

```
$ node app
```

将浏览器转向 http://localhost:3000/users，尝试分别用正确和错误的电子邮件地址创建用户简介。将会看到错误验证会触发一个回调函数，该回调函数有一个错误队形参数。在出现验证错误时，如果希望将 Mongoose 错误信息打印出来，可以如下所示：

```
{ message: 'Validation failed',
  name: 'ValidationError',
  errors:
   { email:
     { message: 'Validator "regexp" failed for path email',
       name: 'ValidatorError',
       path: 'email',
       type: 'regexp' } } }
```

可以对这种类型的错误进行测试，并在文件 routes/users.js 中处理该错误：

```
app.post('/users', notLoggedIn, function(req, res, next) {
  User.create(req.body, function(err) {
    if (err) {
      if (err.code === 11000) {
        res.send('Conflict', 409);
      } else {
        if (err.name === 'ValidationError') {
          return res.send(Object.keys(err.errors).map(function(errField) {
            return err.errors[errField].message;
          }).join('. '), 406);
        } else {
          next(err);
        }

      }
      return;
    }
    res.redirect('/users');
  });
});
```

上面的代码会向用户显示一条简陋的消息，但是可以轻松地完善代码，以便显示合适的上下文消息。

还可以枚举某个字段的允许值。例如，如果想在 gender 字段中存储用户的性别，可以对用户模式进行修改，如程序清单 25-11 所示。

程序清单 25-11：文件 data/schemas/user.js 中带有新增可枚举性别字段的新版用户模式

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

var emailRegexp = /.+\@.+\..+/;

var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: String,
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp
  },
  gender: {
    type: String,
    required: true,
    uppercase: true,
    'enum': ['M', 'F']
  }
});

module.exports = UserSchema;
```

在上面的代码中，规定 gender 字段的值是一个字符串，字符串必须为 M 或者 F。同时要求在验证 gender 字段的值之前将其转换成大写，但是小写的 f 或者 m 也同样有效。

为测试所做的修改，需要重启服务器，并向用户创建表单添加 gender 输入域，如程序清单 25-12 所示。

代码清代 25-12：views/users/new.jad 中带有性别字段的新版用户创建表单

```
h1 New User

form(method="POST", action="/users")
  p
    label(for="username") Username<br />
    input#username(name="username")
  p
    label(for="name") Name<br />
    input#name(name="name")
  p
    label(for="email") Email<br />
    input#email(name="email")
  p
    label(for="password") Password<br />
    input#password(type="password", name="password")
  p
    label(for="gender") Gender<br />
```

```
    input#gender(name="gender")
  p
    label(for="bio") Bio<br />
    textarea#bio(name="bio")
  p
    input(type="submit", value="Create")
```

除了枚举可能值或者提供正则表达式之外，还可以提供一个普通函数来验证字段。例如，由于某种原因，也许你想将生日添加到用户简介中，验证用户是否已满 18 岁。

首先，因为 JavaScript 是以毫秒形式存储日期的，所以要将 18 岁转换成毫秒，应该在 data/schemas/user.js 中的用户模块开头进行这种转换：

```
var TIMESPAN_YEAR = 31536000000;
var TIMESPAN_18_YEARS = 18 * TIMESPAN_YEAR;
```

然后，要提供一个验证函数，这个验证函数同样也要放在 data/schemas/user.js 中用户模式的开头：

```
function validate_18_years_old_or_more(date) {

  return (Date.now() - date.getTime()) > TIMESPAN_18_YEARS;
}
```

这个函数有一个日期参数并返回一个布尔值，该布尔值表明是否大于 18 岁，下面是对模式进行的必要修改：

```
var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: String,
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp
  },
  gender: {
    required: true,
    type: String,
    uppercase: true,
    'enum': ['M', 'F']
  },
  birthday: {
    type: Date,
    validate: [
      validate_18_years_old_or_more,
      'You must be 18 years old or more']
  }
});
```

注意，在验证字段选项中传递了一个包含两个元素的数组——验证函数和消息。通过这种方法可以指定自定义的消息，用自定义消息改写由 Mongoose 自动生成的消息。

为了进行测试，必须重启服务器并向模板 views/users/new.jade 添加一个日期输入域，如程序清单 25-13 所示。

程序清单 25-13：文件 views/users/new.jade 中带有生日字段的新版用户创建表单

```
h1 New User
form(method="POST", action="/users")
  p
    label(for="username") Username<br />
    input#username(name="username")
  p
    label(for="name") Name<br />
    input#name(name="name")
  p
    label(for="email") Email<br />
    input#email(name="email")
  p
    label(for="password") Password<br />
    input#password(type="password", name="password")
  p
    label(for="gender") Gender<br />
    input#gender(name="gender")
  p
    label(for="birthday") Birthday<br />
    input#birthday(name="birthday")
  p
    label(for="bio") Bio<br />
    textarea#bio(name="bio")
  p
    input(type="submit", value="Create")
```

注意：Mongoose 接受任何与 Date.parse()兼容的日期格式，这意味着它可以接受 RFC2822 格式和 ISO 8601 语法。也就是说可以用很多可选格式引入日期，但是为了简化测试，可以采用通用格式 YYYY/MM/DD。

如果验证函数需要进行一些 I/O 操作的话，还可以指定异步验证器。例如，你也许想通过与外部服务进行交互，来进行某种类型的验证。为简单起见，要添加一个包含 Twitter 句柄的字段，并且只需使用 Twitter API 来验证该句柄是否存在。

为此，只要向 Twitter 上的用户页面发送 HTTP 请求，然后再观察响应的 HTTP 状态。如果响应状态是 404 Not Found，可以断定这个 Twitter 句柄是无效的。

为了简化 HTTP 请求的创建过程，可以使用 Mikeal Rogers 的第三方模块 request(在第

13 章“创建 HTTP 请求”中有介绍)，该模块必须包含在清单文件 package.json 中(见程序清单 25-14)。

程序清单 25-14：包含 request 依赖项的新版清单文件 package.json

```
{
    "name": "application-name"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.11"
    , "jade": ">= 0.0.1"
    , "mongoose": ">=2.7.0"
    , "async": "0.1.22"
    , "request": "2.10.0"
  }
}
```

安装缺失的包：

```
$ npm install
...
request@2.10.0 ./node_modules/request
```

在文件 data/schemas/user.js 中的用户模式的最上方导入该模块：

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

然后在同一个文件中定义一个验证函数，这个验证函数有一个值参数和一个回调函数参数，当有来自 Twitter 的响应到达时，就会用一个布尔值进行回调：

```
function twitterHandleExists(handle, done) {
  request('http://twitter.com/' + encodeURIComponent(handle),
    function(err, res) {
    if (err) {
      console.error(err);
      return done(false);
    }
    if (res.statusCode > 299) {
      done(false);
    } else {
      done(true);
    }
  });
}
```

在上面的代码中，对用户 URL twitter.com 进行了一次 HTTP 调用，如果出现错误或者 HTTP 的状态码不是 2xx OK，就足以断定字段值是无效的。在上面的代码中用到了异步验证器，而不是用一个布尔值调用回调函数，并根据回调函数的返回值判断字段是否有效。

25.5.2 使用修改器

有时要在进行验证和保存文档之前修改字段值。例如，性别字段有一个修改器uppercase，该修改器用来在输入验证和保存步骤之前将字符串转换为大写。

对于字符串类型的字段而言，可以使用 uppercase 修改器、lowercase 修改器以及 trim 修改器。trim 修改器会使用 string.trim()方法删除字符串中头部和尾部的空格。

还可以通过定义一个转换和返回值的函数来使用自定义的修改器。例如，用户可以用也可以不用@作为 Twitter 句柄的前缀：

```
var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: String,
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp
  },
  gender: {
    required: true,
    type: String,
    uppercase: true,
    'enum': ['M', 'F']
  },
  birthday: {
    type: Date,
    validate: [validate_18_years_old_or_more, 'You must be 18 years old or
      more'
  },
  twitter: {
    type: String,
    validate: [twitterHandleExists, 'Please provide a valid twitter
      handle'],
    set: function(handle) {
      if (! handle) {
        return;
      }

      handle = handle.trim();
      if (handle.indexOf('@') === 0) {
        handle = handle.substring(1);
      }
      return handle;
    }
  }
});
```

在上面的代码中，首先对句柄进行裁剪(如果用户没有这样做)，如果句柄的第一个字

符是@，将该字符删除。通过这种方法，用户既可以输入@mytwitterhandle，也可以输入 mytwitterhandle，两者都会被映射为 mytwitterhandle。

25.5.3　使用取值器

通过定义取值器，可以在 MongoDB 向 Mongoose 提交数据之后但在对象可用之前过滤对象，从而决定哪些对象可被存入数据库。通过提供一个自定义函数，可以使用取值器对值进行过滤，或者将其转换为想要的形式。这在有些时候也许很有用，例如在执行数据迁移时。

假如允许用户存储以@作为前缀 Twitter 的句柄，之后决定用一个赋值器将这些句柄过滤出去。在用户集合中有几个以@作为前缀的 Twitter 句柄，其余 Twitter 句柄则没有这个前缀。可以进行一次完整的数据迁移，虽然根据用户集合的规模，这可能要耗费很长一段时间，并对服务的性能产生影响。还可以通过修改视图代码，允许这两种不同的变化，从而学习容忍这种不一致性，但是这可能会使数据过滤代码在应用程序中的位置不正确。

可以按照如下所示的方式定义过滤函数，并将其放置在 data/schemas/user.js 的最上面：

```
function filterTwitterHandle(handle) {
  if (! handle) {
    return;
  }
  handle = handle.trim();
  if (handle.indexOf('@') === 0) {
    handle = handle.substring(1);
  }
  return handle;
}
```

这个过滤函数既可用作 twitter 字段的赋值器，也可用作其取值器，如下所示：

```
var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: String,
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp
  },
  gender: {
    required: true,
    type: String,
    uppercase: true,
    'enum': ['M', 'F']
  },
  birthday: {
    type: Date,
    validate: [validate_18_years_old_or_more, 'You must be 18 years old or
```

```
      more']
    },
    twitter: {
      type: String,
      validate: [twitterHandleExists, 'Please provide a valid twitter
       handle'],
      set: filterTwitterHandle,
      get: filterTwitterHandle
    }
  });
```

25.5.4 使用虚拟属性

还可以让一些属性不在文档中出现，而是由 Mongoose 在加载文档时动态计算出来。例如，前端代码必须基于 Twitter 句柄构建 Twitter URL 字符串，但 Twitter URL 是用户的一个属性，就其本身而言应该属于用户文档。然而，由于该属性可以基于 Twitter 句柄很容易地被计算出来，所以可以创建一个 virtual 属性。在 data/schemas/user.js 中的用户模式模块导出之前，紧接着该模块添加如下所示的代码，就可以创建 virtual 属性：

```
//...
UserSchema
  .virtual('twitter_url')
  .get(function() {
    if (this.twitter) {
      return 'http://twitter.com/' + encodeURIComponent(this.twitter);
    }
  });
```

注意：虚拟属性不会被存入文档。

现在可以修改用户简介视图模板以使用所有的新属性，如程序清单 25-15 所示。

程序清单 25-15：在 views/users/profile.jade 中使用 twitter_url 虚拟属性的新版用户简介模板

```
h1= user.name

h2 Bio
p= user.bio

h2 Email
p= user.email

h2 Gender
p= user.gender

h2 Birthday
```

```
p= user.birthday

- if (user.twitter) {
  h2 Twitter
  p
    a(href= user.twitter_url)="@" + user.twitter
- }


form(action="/users/" + encodeURIComponent(user.username), method="POST")
  input(name="_method", type="hidden", value="DELETE")
  input(type="submit", value="Delete")
```

还可以为虚拟属性定义赋值器，例如，也许你想在文档中分割用户的名和姓，但是需要在用户界面中保留姓名作为单个的输入域。此时可以重定义文件 data/schemas/user.js 中的用户模式部分，如下所示：

```
var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: {
    first: String,
    last: String
  },
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp
  },
  gender: {
    required: true,
    type: String,
    uppercase: true,
    'enum': ['M', 'F']
  },
  birthday: {
    type: Date,
    validate: [validate_18_years_old_or_more, 'You must be 18 years old or
      more']
  },
  twitter: {
    type: String,
    validate: [twitterHandleExists, 'Please provide a valid twitter
      handle'],
    set: filterTwitterHandle,
    get: filterTwitterHandle
  }
});
UserSchema
  .virtual('full_name')
```

```
  .get(function() {
    return [this.name.first, this.name.last].join(' ');
  })
  .set(function(fullName) {
    var nameComponents = fullName.split(' ');
    this.name.last = nameComponents.pop();
    this.name.first = nameComponents.join(' ');
  });

UserSchema
  .virtual('twitter_url')
  .get(function() {
    if (this.twitter) {
      return 'http://twitter.com/' + encodeURIComponent(this.twitter);
    }
  });
```

在上面的代码中，采用一个内嵌对象定义了一个包含 name 字段的用户文档，这个嵌入式对象既包含名又包含姓。然后规定任何一个用户文档都有一个虚拟属性 full_name，它的值是名和姓通过空格连接起来的结果。该属性同样可被赋值，在赋值器中进行的是相反的操作，即将全名分割成名和姓。

现在需要对视图模板进行少许修改，将属性 name 修改为 full_name，如程序清单 25-16 所示：

程序清单 25-16：views/users/new.jade 中采用虚拟属性 full_name 的新版用户创建表单

```
h1 New User

form(method="POST", action="/users")
  p
    label(for="username") Username<br />
    input#username(name="username")
  p
    label(for="full_name") Full name (first and last)<br />
    input#full_name(name="full_name")
  p
    label(for="email") Email<br />
    input#email(name="email")
  p
    label(for="password") Password<br />
    input#password(type="password", name="password")
  p
    label(for="gender") Gender<br />
    input#gender(name="gender")
  p
    label(for="birthday") Birthday<br />
    input#birthday(name="birthday")
  p
```

```
    label(for="twitter") Twitter Handle<br />
    input#twitter(name="twitter")
  p
    label(for="bio") Bio<br />
    textarea#bio(name="bio")
  p
    input(type="submit", value="Create")
```

需要在用户简介视图模板中进行同样的修改，如程序清单 25-17 所示：

程序清单 25-17：views/users/profile.jade 中采用虚拟属性 full_name 的新版用户简介模板

```
h1= user.full_name

h2 Bio
p= user.bio

h2 Email
p= user.email

h2 Gender
p= user.gender

h2 Birthday
p= user.birthday

- if (user.twitter) {
  h2 Twitter
  p
    a(href= user.twitter_url)="@" + user.twitter
- }


form(action="/users/" + encodeURIComponent(user.username), method="POST")
  input(name="_method", type="hidden", value="DELETE")
  input(type="submit", value="Delete")
```

还需要更新 views/users/index.jade 中的用户列表模板，在这个模板中抓住机会将 for 循环转换成 foreach 迭代，如程序清单 25-18 所示。

程序清单 25-18：使用 full_name 虚拟属性的新版用户列表模板

```
h1 Users

p
  a(href="/users/new") Create new profile

ul
  - users.forEach(function(user) {
    li
```

```
      a(href="/users/" + encodeURIComponent(user.username))=
        user.full_name - });

  - if (page > 0) {
    a(href="?page=" + (page - 1)) Previous
     
  - }

  - if (! lastPage) {
    a(href="?page=" + (page + 1)) Next
  - }
```

此外，还需要修改用户登录时在页面顶端显示用户名的视图部分，即要将文件 views/session/user.jade 修改为：

```
  - if (session.user) {

    p
      span Hello 
      span= session.user.name.first
      span !
    p
      form(method="POST", action="/session")
        input(type="hidden", name="_method", value="DELETE")
        input(type="submit", value="Log out")
  - } else {

    p
      a(href="/session/new") Login
      span  or 
      a(href="/users/new") Register
  - }
```

也许你还拥有之前的文档，在这些文档中，name 字段仍旧是字符串值，而不是一个嵌入式的 name 文档。这样的文档不具有 name.first 属性或者 name.last 属性。相反，name 属性包含一个表示全名的字符串。

为了解决这种不一致性，需要进行三处修改。首先，需要在文件 data/schemas/user.js 中声明 name 字段是混合类型。这样，Mongoose 就允许 name 字段既可以被赋值为字符串，也可以被赋值为一个对象(实际上可以是任何值)。

```
var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: mongoose.Schema.Types.Mixed,
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp
  },
```

```
  gender: {
    required: true,
    type: String,
    uppercase: true,
    'enum': ['M', 'F']
  },
  birthday: {
    type: Date,
    validate: [validate_18_years_old_or_more, 'You must be 18 years old or
      more']
  },
  twitter: {
    type: String,
    validate: [twitterHandleExists, 'Please provide a valid twitter
      handle'],
    set: filterTwitterHandle,
    get: filterTwitterHandle
  }
});
```

另外，还需要修改 full_name 虚拟函数的取值器函数，使得原有文档中的 name 字段仍然可以是字符串。在这种情况下，只要让取值器函数返回字符串即可：

```
UserSchema
  .virtual('full_name')
  .get(function() {
    if (typeof this.name === 'string') {
      return this.name;
    }
    return [this.name.first, this.name.last].join(' ');
  })
  .set(function(fullName) {
    var nameComponents = fullName.split(' ');
    this.name = {
      last: nameComponents.pop(),
      first: nameComponents.join(' ')
    };
  });
```

还需要修改 full_name 虚拟属性的赋值器，以便在给该属性赋值时可以创建内嵌的 name 文档。之所以要这样做，是因为现在 Mongoose 没有假定一个前提——在前面的例子中 Mongoose 知道内嵌文档必定存在，所以才能预先将内嵌文档对象赋值给 name 属性。 既然还没有定义这种对象类型，就只能亲力而为了。

现在按下 Ctrl+C 键重启应用程序并重新启动 Node：

```
$ node app.js
```

现在可以刷新最初的 URL(http://localhost:3000/users)，尝试创建和显示用户的名，同时查看数据库的相应内容。

25.5.5 使用默认值

也许你想跟踪用户记录创建的日期和时间，此时可以插入一个日期类型的 create_at 字段，该字段必须能够自动赋值——你不希望用户提交甚至修改该字段的值。可以将 create_at 字段插入到用户创建路由监听器上的用户文档中，但是对于代码而言这并不是合适的插入位置。如果想在应用程序中别的地方创建用户，就必须再次执行插入操作，这样会导致重复代码，有可能还会造成一些不一致性。

正确的位置是文件 data/schemas/user.js 中的用户模式，这需要进行一些修改：

```
var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: mongoose.Schema.Types.Mixed,
  password: String,
  email: {
    type: String,
    required: true,
    match: emailRegexp
  },
  gender: {
    required: true,
    type: String,
    uppercase: true,
    'enum': ['M', 'F']
  },
  birthday: {
    type: Date,
    validate: [validate_18_years_old_or_more, 'You must be 18 years old or
      more']
  },
  twitter: {
    type: String,
    validate: [twitterHandleExists, 'Please provide a valid twitter
      handle'],
    set: filterTwitterHandle,
    get: filterTwitterHandle

  meta: {
    created_at: {
      type: Date,
      'default': Date.now
    },
    updated_at: {
      type: Date,
      'default': Date.now
    }
  }
});
```

在上面的代码中，meta.created_at 和 meta.updated_at 字段在默认情况下被赋值为当前时间戳(在保存之前会被转换成合适的日期值)。

但是上述方法存在一个问题：如果用户提交的表单修改了 meta.created_at 记录，修改后的值会被存入文档中，而且会覆盖初始值，但是我们并不希望初始值发生改变。为了不改变该值，需要在保存文档之前删除 meta.created_at 和 meta.updated_at 的值，并插入默认值。为此，在模块导出之前，需要将如下所示的一段代码添加到文件 data/schemas/user.js 的末尾：

```
UserSchema.pre('save', function(next) {
  if (this.isNew) {
    this.meta.created_at = undefined;
  }

  this.meta.updated_at = undefined;
  next();
});
```

用户模式会在保存或删除对象的前后发射事件。本例中添加了一个监听器，该监听器会在保存对象之前被调用，监听器函数中的关键字 this 是指文档。

通过观察 isNew 属性，还可以确定文档是否是首次被保存。本例只是想在创建文档对象时删除 meta.create_at 的值。

注意：如果在更新文档对象时用户用某种方法成功改写了 meta.created_at 值，上面的代码仍旧起不了什么作用。为了避免这种情况，可以创建一个属性 setter，这个属性不允许 meta.created_at 的值发生任何改变：

```
//...
meta: {
  created_at: {
    type: Date,
    'default': Date.now,
    set: function(val) {
      return undefined;
    }
  },
  updated_at: {
    type: Date,
    'default': Date.now
  }
}
```

25.5.6　定义索引

与一些其他的面向文档的数据库不同，MongoDB 可以让你执行一些特殊的查询，在

这些查询中，数据库会创建一个计划来执行查询，并尝试对查询进行优化。为了加速查询，需要创建索引。既可以在单个字段上创建索引，也可以结合两个或者多个字段创建复合索引。

索引可以是唯一的，也可以不是。在用户模式中，用户名有一个选项 unique 被设置为 true，这就是告诉 Mongoose，在查询执行之前，必须保证已经为用户名字段创建了一个唯一性索引。恰好在用户的用户名属性上已经有了一个唯一性索引：

```
// ...
var UserSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  name: mongoose.Schema.Types.Mixed,
// ...
```

正如前面所讨论的那样，唯一性索引确保所有文档中的用户名字段值都不相同。

此外，索引既可以是稀疏的，也可以不是。如果在某个字段上定义了索引，普通索引会包含一个针对所有文档的入口：这些文档既可包含该字段，也可以不包含该字段。但是，如果将字段上的 sparse 选项设置为 true，Mongoose 只能保证创建的索引只包含那些具有指定字段的文档。

使用稀疏索引会对利用对应字段的查询结果造成影响。例如，如果在用户模式的 Twitter 句柄上定义稀疏索引，然后根据 Twitter 句柄进行查询，并对查询结果进行排序，那么查询结果只会包含那些定义了 Twitter 句柄的文档。试着修改 data/schemas/user.js 中的用户模式模块，将 sparse 属性设置为 true：

```
  // ...
  twitter: {
    type: String,
    sparse: true,
    validate: [twitterHandleExists, 'Please provide a valid twitter
      handle'],
    set: filterTwitterHandle,
    get: filterTwitterHandle
  },
// …
```

执行如下所示的查询，只会获得包含 twitter 字段的文档：

```
User.find({}).sort({twitter: 1}, callback);
```

可以结合稀疏索引和唯一性索引来创建唯一性约束，唯一性约束会忽略那些缺少对应字段的文档。例如，如果希望 email 字段是唯一的，但不强制要求文档中必须有该字段，可以通过设置文件 data/schemas/user.js 中的用户模式内的下列属性，为其创建一个稀疏的唯一性索引：

```
// ...
email: {
```

```
    type: String,
    sparse: true,
    unique: true,
    match: emailRegexp
  },
  // ...
```

在模块导出之前，如果通过将下面的代码插入到文件 data/users/schema.js 的结尾处，还可以在模式层面定义结合了两个或多个字段值的索引：

```
UserSchema.index({username: 1, 'meta.created_at': -1});
```

上面的代码创建了一个结合了 username 和 meta.created_at 值的索引，其中，用户名按照升序编制索引，而 meta.created_at 按照降序编制索引。当采用这两个字段进行过滤或排序的查询操作时，这个索引会加快查询速度。

注意：如果现在尝试重启应用程序，也许会遇到下面的问题：

```
MongoError: E11000 duplicate key error index: my_mongoose
.users.$email_1  dup key: { : "myemail@mail.com" }
```

如果在数据库中的多个用户具有相同的 e-mail 地址，就会出现该问题。要顺利启动应用程序，必须修改复制次序。

25.5.7　使用 DB Refs 引用其他文档

在 Mongoose 中，文档可能还包含指向其他文档的引用。例如，可能想要有一个文章集合，除了其他字段之外，它还有一个作者字段，作者可以是一个用户。此时，可以规定作者字段的值是一个对象 ID，而不是用户名。在 data/schemas/article.js 下定义如下模式，如程序清单 25-19 所示。

程序清单 25-19：论文模式

```
var Schema = require('mongoose').Schema;

var ArticleSchema = new Schema({
  title: {
    type: String,
    unique: true
  },
  body: String,
  author: {
    type: Schema.ObjectId,
    ref: 'User',
    required: true
  },
```

```
  created_at: {
    type: Date,
    'default': Date.now
  }
});

module.exports = ArticleSchema;
```

在上面的代码中，定义了一个简单的文章模式，该模式包含一个作者字段，类型为Schema.ObjectId，并引用了User模型。

在文件data/models/article.js下定义模型，如程序清单25-20所示。

程序清单25-20：文章模型

```
var mongoose = require('mongoose');
var ArticleSchema = require('../schemas/article');

var Article = mongoose.model('Article', ArticleSchema);

module.exports = Article;
```

现在，需要在文件routes/articles.js中创建一组简单的监听器，以响应路径/articles下的URL(见程序清单25-21)。

程序清单25-21：文章路由监听器

```
/*
 * 文章路由
 */

var async = require('async');

var Article = require('../data/models/article');
var notLoggedIn = require('./middleware/not_logged_in');
var loadArticle = require('./middleware/load_article');
var loggedIn = require('./middleware/logged_in');

var maxArticlesPerPage = 5;

module.exports = function(app) {

  app.get('/articles', function(req, res, next){
    var page = req.query.page  && parseInt(req.query.page, 10) || 0;
    async.parallel([

        function(next) {
          Article.count(next);
        },
```

```
      function(next) {
        Article.find({})
          .sort('title', 1)
          .skip(page * maxArticlesPerPage)
          .limit(maxArticlesPerPage)
          .exec(next);
      }
    ],

    // final callback
    function(err, results) {

      if (err) {
        return next(err);
      }

      var count = results[0];
      var articles = results[1];

      var lastPage = (page + 1) * maxArticlesPerPage >= count;

      res.render('articles/index', {
        title: 'Articles',
        articles: articles,
        page: page,
        lastPage: lastPage
      });

    }
  );
});

app.get('/articles/new', loggedIn, function(req, res) {
  res.render('Articles/new', {title: "New Article"});
});

app.get('/articles/:title', loadArticle, function(req, res, next){
  res.render('articles/article', {title: req.article.title,
    article: req.article});
});

app.post('/articles', loggedIn, function(req, res, next) {
  var article = req.body;
  article.author = req.session.user._id;
  Article.create(article, function(err) {
    if (err) {
      if (err.code === 11000) {
        res.send('Conflict', 409);
      } else {
        if (err.name === 'ValidationError') {
```

```
            return res.send(Object.keys(err.errors).map(function(errField) {
              return err.errors[errField].message;
            }).join('. '), 406);
          } else {
            next(err);
          }

        }
        return;
      }
        res.redirect('/articles');
      });
    });

    app.del('/articles/:title', loggedIn, loadArticle, function(req, res,
      next) {
      req.article.remove(function(err) {
        if (err) { return next(err); }
        res.redirect('/articles');
      });

    });

  };
```

上述代码主要用于创建、移除、更新和删除应用程序(除了为简单起见，没有提供更新方法)，所以这些路由监听器在某种程度上与 User 中的路由监听器类似，但有一点除外，即在新建文章时，会根据如下所示的代码，将作者的用户 ID 赋值给作者：

```
article.author = req.session.user._id;
```

注意：在 MongoDB 中，每个对象在创建之后都有一个_id 字段，该字段包含对象在集合中的唯一 ID。与一些数据库不同，_id 字段不是连续的数字，而是随机字符串。

这些路由监听器用到了两个新的路由中间件组件——loggedIn 和 loadArticle，后者会加载在 routes/middleware/load_article.js 下创建的文章，如程序清单 25-22 所示。

程序清单 25-22：加载 routes/middleware/load_article.js 中文章的中间件组件

```
var Article = require('../../data/models/article');

function loadArticle(req, res, next) {
  Article.findOne({title: req.params.title})
    .populate('author')
```

```
  .exec(function(err, article) {
    if (err) {
      return next(err);
    }
    if (! article) {
      return res.send('Not found', 404);
    }
    req.article = article;
    next();
  });
}

module.exports = loadArticle;
```

上面这个新的路由中间件组件与 load_user.js 中的路由中间件组件非常相似，但有一点除外，即它会使用如下代码，让 Mongoose 加载作者：

```
Article.findOne({title: req.params.title})
  .populate('author')
  //...
```

以上代码使得用户对象可用于属性 article.author，这样一来就可以在 views/articles/article.jade 下的文章显示模板中使用该对象，如程序清单 25-23 所示。

程序清单 25-23：views/articles/article.jade 中的文章细节视图模板

```
h1= article.title

div!= article.body

hr

p
  span Author:
   
  a(href="/users/" + encodeURIComponent(article.author.username))=
   article.author.full_name

p
  a(href="/articles") Back to all articles
```

此外，还必须在 routes/middleware/logged_in.js 下添加 loggedIn 中间件组件，如程序清单 25-24 所示。

程序清单 25-24：views/articles/article.jade 中的文章细节视图模板

```
function loggedIn(req, res, next) {
  if (! req.session.user) {
    res.send('Forbidden. Please log in first.', 403);
  } else {
```

```
    next();
  }
}

module.exports = loggedIn;
```

为了能够让上述代码起作用，还需要向文件 app.js 中添加几个新的路由监听器，如下所示：

```
// 路由

require('./routes/session')(app);
require('./routes/index')(app);
require('./routes/users')(app);
require('./routes/articles')(app);
```

此外还需要两个新的视图模板：

- views/articles/index.jade：包含文章列表
- views/articles/new.jade：包含文章创建表单

查看一下程序清单 25-25 和程序清单 25-26。

程序清单 25-25：views/articles/index.jade 中的文章列表模板

```
h1 Articles

p
  a(href="/articles/new") Create new article

ul
  - articles.forEach(function(article) {
    li
      a(href="/articles/" + encodeURIComponent(article.title))=
        article.title - });

- if (page > 0) {
  a(href="?page=" + (page - 1)) Previous
   
- }

- if (! lastPage) {
  a(href="?page=" + (page + 1)) Next
- }
```

程序清单 25-26：The article creation form in views/articles/new.jade.

```
h1 New Article

form(method="POST", action="/articles")
```

```
p
  label(for="title") Title<br />
  input#title(name="title")
p
  label(for="body") Body<br />
  textarea#body(name="body")
p
  input(type="submit", value="Create")
```

25.5.8　定义实例方法

现在，也许你想列出某个指定用户最近的一些文章。为此，可以在用户路由监听器中直接创建查询，但用户路由监听器并不是完成这项任务的合适场所。另一种方法在文件 data/schemas/user.js 末尾的模块导出之前，将一个方法附加到用户模式中，从而创建一个实例方法，如下所示：

```
UserSchema.methods.recentArticles = function(callback) {
  return this.model('Article')
    .find({author: this._id})
    .sort('created_at', 1)
    .limit(5)
    .exec(callback);
};
```

现在可以用上述实例方法获得某个指定用户最近 5 篇文章的列表。修改文件 routes/users.js 中的路由监听器 GET '/users/:name'，以便获得该用户最新的文章，并列出它们：

```
app.get('/users/:name', loadUser, function(req, res, next){
  req.user.recentArticles(function(err, articles) {
    if (err) {
      return next(err);
    }
    res.render('users/profile', {
      title: 'User profile',
      user: req.user,
      recentArticles: articles
    });
  });

});
```

现在需要修改模板 views/users/profile.jade，将如下所示的一小段代码添加到末尾：

```
h2 Recent Articles:
p!= partial('articles/list', {articles: recentArticles })
```

上述代码用到了必须创建的部分模板 views/articles/list.jade，如程序清单 25-27 所示：

程序清单 25-27：The article list partial template

```
ul
  - articles.forEach(function(article) {
    li
      a(href="/articles/" + encodeURIComponent(article.title))=
        article.title
  - });
```

由于已经决定让这个部分模板显示列表，因此可以用对这个部分模板的调用来替换模板 views/articles/index.jade 中的显式列表，如程序清单 25-28 所示：

程序清单 25-28：使用文章列表 partial 的新版文章列表模板

```
h1 Articles

p
  a(href="/articles/new") Create new article

p!= partial('articles/list', {articles: articles})

- if (page > 0) {
  a(href="?page=" + (page - 1)) Previous
   
- }

- if (! lastPage) {
  a(href="?page=" + (page + 1)) Next
- }
```

现在是测试文章创建和显示的好时机，重启应用程序并转向 http://localhost:3000/articles，尝试登录、创建文章并浏览。

25.5.9 定义静态方法

还可以在模型上对外暴露方法，提供更高级的操作或者查询。例如，也许你想提供一个查找函数，它能返回标题或者正文中包含某个字符串的文章，为此，可以在 data/schemas/article.js 中的模块导出之前，在文章模式模块的末尾定义该函数：

```
ArticleSchema.statics.search = function(str, callback) {
  var regexp = new RegExp(str, 'i');
  return this.find({'$or': [{title: regexp}, {body: regexp}]}, callback);
};
```

现在可以在 routes/articles.js 文件中创建一个路由监听器来暴露该功能：

```
app.get('/articles/search', function(req, res, next) {
  console.log('searching for', req.query.q);
  Article.search(req.query.q, function(err, articles) {
    if (err) {
```

```
    return next(err);
  }
  res.render('articles/index', {
    title: 'Article search results',
    articles: articles,
    page: 0,
    lastPage: true
  });
});
});
```

注意：应该将上述代码放在 app.get('/articles/:title')路由之前，以便使其优先执行。

可以重启服务器、登录、创建一些文章，并尝试使用类似 http://localhost:3000/articles/search?q=somestring 的 URL 来测试上述代码，其中 somestring 可以用某篇文章的标题或者正文的一部分代替。

25.6　本章小结

Mongoose 是一个数据库访问库，允许将对象映射成 MongoDB 数据库中的集合和文档。虽然面向文档的数据库允许非结构化数据，但是 Monogoose 需要为每个集合定义一个模式，在模式中指定数据类型、索引、默认值、多种验证函数以及其他操作。

然后就可以在集合模型中使用这些模式，这些模式提供了查找、插入和修改文档的方法。此外，还可以在已有方法的基础上，定义新的方法，以丰富集合模型和返回的文档对象。

此外，还可以使用 Schema.ObjectId 类型，并定义目标集合，以便在一个文档中引用位于相同或者不同集合中的其他文档。

学习使用Node.js构建基于JavaScript的可伸缩性软件

Node.js是一种主流框架，它允许你使用JavaScript快速构建具有高度可伸缩性的网络程序。可是，它有自己的学习曲线，这本较为深入的指南性图书首先介绍了Node.js平台的安装，然后重点关注：创建和加载模块；使用缓冲区对二进制数据进行处理、编码和编码；用事件发射器模式简化事件绑定等。本书还涵盖了从使用定时器制定函数执行计划到创建和控制外部进程等内容，可以让你为运行、构建和测试自定义模块做好准备。

主要内容

- 介绍了查询和读写文件
- 研究了流、文件系统、网络和自动化单元测试
- 详细描述了如何读写数据流
- 从零开始构建TCP和HTTP服务器与客户端
- 介绍使用数据报、测试和调试模块以及控制回调流程
- 展示了如何用Connect、Express和Socke.IO开发实时Web应用程序
- 引导你连接到MySQL、CouchDB和MongoDB数据库

作者简介

Pedro Teixeria是一位高产的开源项目程序员，同时也是众多Node.js模块的构建者。他是Node公司的创始人之一，同时也是Nodejitsu公司的高级程序员，Nodejitsu公司是以Node.js平台作为服务的领头羊。

自从在14年前获得软件工程学位后，他从事过的职业包括咨询师和程序员，并且他还是世界知名的Node.js社区活跃成员。此外，他还是广受欢迎的Node Tuts视频的制作者。

源代码下载及技术支持

http://www.wrox.com
http://www.tupwk.com.cn/downpage

wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

清华大学出版社数字出版网站
WQBook 书文局泉
www.wqbook.com

Wrox
An Imprint of
WILEY

ISBN 978-7-302-34441-4
9 787302 344414 >
定价：58.00元