

SAMS
**Teach
Yourself**

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

每章内容针对初学者精心设计，**1**小时轻松阅读学习，
24小时彻底掌握关键知识

每章**案例与练习题**助你轻松完成常见任务，
通过**实践**提高应用技能，巩固所学知识

Node.js

入门经典

[英] George Ombo 著
傅强 陈宗斌 译

学习如何：

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

24章阶梯教学

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

循序渐进的示例引导读者完成最常见的Node.js开发任务。

问与答、测验和练习帮助读者检验知识的掌握情况。

“注意”、“提示”和“警告”指出捷径和解决方案。



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



人民邮电出版社·信息技术分社
<http://weibo.com/ptpitbooks>

美术编辑 王建国

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



ISBN 978-7-115-31107-8



9 787115 311078

ISBN 978-7-115-31107-8

定价：59.00 元

SAMS
Teach
Yourself

SAMS
Teach
Yourself

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

每章内容针对初学者精心设计，**1**小时轻松阅读学习，
24小时彻底掌握关键知识

每章**案例与练习题**助你轻松完成常见任务，
通过**实践**提高应用技能，巩固所学知识

Node.js

入门经典

[英] George Ormbo 著
傅强 陈宗斌 译

人民邮电出版社
POSTS & TELECOM PRESS

Node.js 入门经典

人民邮电出版社

Node.js

入门经典

[英] George Ornbo 著
傅强 陈宗斌 译

人民邮电出版社
北京

TP312JA
1469

关于作者

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



内容提要

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

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

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

新华书店
PDG

学习 Node.js 的理由

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

本书组织结构

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

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

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

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

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

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

代码示例

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

目 录

第1部分 入门

第1章 Node.js 介绍 2

1.1 什么是 Node.js 2

1.2 使用 Node.js 能做什么 3

1.3 安装并创建第一个 Node.js 程序 3

1.3.1 验证 Node.js 正确安装 4

1.3.2 创建“Hello World”
Node.js 程序 4

1.4 小结 5

1.5 问与答 6

1.6 测验 6

1.6.1 问题 6

1.6.2 答案 7

1.7 练习 7

第2章 npm (Node 包管理器) 8

2.1 npm 是什么 8

2.2 安装 npm 9

2.3 安装模块 9

2.4 使用模块 10

2.5 如何找模块 11

2.5.1 官方来源 11

2.5.2 非官方来源 12

2.6 本地和全局的安装 13

2.6.1 本地安装 13

2.6.2 全局安装 13

2.7 如何找模块文档 14

2.8 使用 package.json 指定依赖
关系 (dependency) 14

2.9 小结 16

2.10 问与答 16

2.11 测验 16

2.11.1 问题 16

2.11.2 答案 17

2.12 练习 17

第3章 Node.js 的作用 18

3.1 设计 Node.js 的目的 18

3.2 理解 I/O 19

3.3 处理输入 19

3.4 联网的 I/O 是不可预测的 22

3.5 人类是不可预测的 24

3.6 处理不可预测性 25

3.7 小结 26

3.8 问与答 26

3.9 测验 27

3.9.1 问题 27

3.9.2 答案 27

3.10 练习 27

第4章 回调 (Callback)	29	6.5.5 routes	58
4.1 什么是回调	29	6.5.6 views	58
4.2 剖析回调	33	6.6 介绍 Jade	59
4.3 Node.js 如何使用回调	34	6.6.1 使用 Jade 定义页面结构	60
4.4 同步和异步代码	36	6.6.2 使用 Jade 输出数据	62
4.5 事件循环	39	6.7 小结	68
4.6 小结	39	6.8 问与答	68
4.7 问与答	39	6.9 测验	68
4.8 测验	40	6.9.1 问题	69
4.8.1 问题	40	6.9.2 答案	69
4.8.2 答案	40	6.10 练习	69
4.9 练习	40		
第2部分 使用 Node.js 的基本网站		第7章 深入 Express	70
第5章 HTTP	44	7.1 Web 应用程序中的路由	70
5.1 什么是 HTTP	44	7.2 在 Express 中路由如何工作	70
5.2 使用 Node.js 的 HTTP 服务器	44	7.3 添加 GET 路由	71
5.2.1 一个基础的服务器	44	7.4 添加 POST 路由	72
5.2.2 加入头 (Header)	45	7.5 在路由中使用参数	73
5.2.3 检查响应头	46	7.6 让路由保持可维护性	74
5.2.4 Node.js 中的重定向	49	7.7 视图渲染	75
5.2.5 响应不同的请求	50	7.8 使用本地变量	76
5.3 使用 Node.js 的 HTTP 客户端	52	7.9 小结	78
5.4 小结	53	7.10 问与答	78
5.5 问与答	53	7.11 测验	78
5.6 测验	54	7.11.1 问题	79
5.6.1 问题	54	7.11.2 答案	79
5.6.2 答案	54	7.12 练习	79
5.7 练习	54		
第6章 Express 介绍	55	第8章 数据的持久化	80
6.1 什么是 Express	55	8.1 什么是持久的数据	80
6.2 为什么使用 Express	55	8.2 将数据写入文件	81
6.3 安装 Express	56	8.3 从文件读取数据	82
6.4 创建一个基础的 Express 站点	56	8.4 读取环境变量	83
6.5 探索 Express	58	8.5 使用数据库	84
6.5.1 app.js	58	8.5.1 关系数据库	84
6.5.2 node_modules	58	8.5.2 NoSQL 数据库	85
6.5.3 package.json	58	8.6 在 Node.js 中使用 MongoDB	85
6.5.4 public	58	8.6.1 安装 MongoDB	86
		8.6.2 连接 MongoDB	87
		8.6.3 定义文档	89
		8.6.4 将 Twitter Bootstrap 包含进来	90

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

第3部分 调试、测试与部署

第9章 调试 Node.js 应用程序 106

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

第10章 测试 Node.js 应用程序 119

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

第11章 部署 Node.js 应用程序 133

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

第4部分 使用 Node.js 的中间站点

第12章 介绍 Socket.IO 148

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

12.9 问与答	163	15.4 从 JavaScript 对象创建 JSON	212
12.10 测验	164	15.5 使用 Node.js 消费 JSON	
12.10.1 问题	164	数据	213
12.10.2 答案	164	15.6 使用 Node.js 创建 JSON API	216
12.11 练习	165	15.6.1 在 Express 中以 JSON	
第 13 章 一个 Socket.IO 聊天服务器	166	发送数据	216
13.1 Express 和 Socket.IO	166	15.6.2 构建应用程序	219
13.2 添加昵称	168	15.7 小结	224
13.2.1 将昵称发送给服务器	169	15.8 问与答	225
13.2.2 管理昵称列表	171	15.9 测验	225
13.2.3 使用回调来验证	174	15.9.1 问题	225
13.2.4 广播昵称列表	178	15.9.2 答案	225
13.2.5 添加消息收发功能	179	15.10 练习	226
13.3 小结	183	第 5 部分 探索 Node.js API	
13.4 问与答	184	第 16 章 进程模块	228
13.5 测验	184	16.1 进程是什么	228
13.5.1 问题	184	16.2 退出进程以及进程中的	
13.5.2 答案	184	错误	230
13.6 练习	185	16.3 进程与信号	230
第 14 章 一个流 Twitter 客户端	186	16.4 向进程发送信号	231
14.1 流 API	186	16.5 使用 Node.js 创建脚本	233
14.2 注册 Twitter	187	16.6 给脚本传递参数	234
14.3 和 Node.js 一起使用 Twitter		16.7 小结	236
的 API	189	16.8 问与答	236
14.4 从数据中挖掘含义	191	16.9 测验	237
14.5 将数据推送到浏览器	194	16.9.1 问题	237
14.6 创建一个实时的爱恨表	197	16.9.2 答案	237
14.7 小结	206	16.10 练习	238
14.8 问与答	206	第 17 章 子进程模块	239
14.9 测验	206	17.1 什么是子进程	239
14.9.1 问题	206	17.2 杀死子进程	241
14.9.2 答案	206	17.3 与子进程通信	242
14.10 练习	207	17.4 集群 (Cluster) 模块	244
第 15 章 JSON API	208	17.5 小结	246
15.1 API	208	17.6 问与答	246
15.2 JSON	209	17.7 测验	246
15.3 使用 Node.js 发送 JSON		17.7.1 问题	246
数据	211	17.7.2 答案	246

17.8 练习	247	20.9 练习	279
第 18 章 事件模块	248	第 6 部分 进一步的 Node.js 开发	
18.1 理解事件	248	第 21 章 CoffeeScript	282
18.2 通过 HTTP 演示事件	251	21.1 什么是 CoffeeScript	282
18.3 用事件玩乒乓	254	21.2 安装与运行 CoffeeScript	284
18.4 动态编写事件侦听器程序	255	21.3 为什么要使用预编译器	285
18.5 小结	258	21.4 CoffeeScript 的功能	286
18.6 问与答	258	21.4.1 最小语法	286
18.7 测验	259	21.4.2 条件和比较	287
18.7.1 问题	259	21.4.3 循环	288
18.7.2 答案	259	21.4.4 字符串	289
18.8 练习	259	21.4.5 对象	290
第 19 章 缓冲区模块	260	21.4.6 类、继承和 super	291
19.1 二进制数据初步	260	21.5 调试 CoffeeScript	294
19.2 从二进制到文本	261	21.6 对 CoffeeScript 的反应	294
19.3 二进制和 Node.js	262	21.7 小结	295
19.4 Node.js 中的缓冲区是什么?	264	21.8 问与答	295
19.5 写入缓冲区	265	21.9 测验	296
19.6 向缓冲区追加数据	266	21.9.1 问题	296
19.7 复制缓冲区	267	21.9.2 答案	296
19.8 修改缓冲区中的字符串	267	21.10 练习	296
19.9 小结	268	第 22 章 创建 Node.js 模块	298
19.10 问与答	268	22.1 为什么创建模块	298
19.11 测验	268	22.2 流行的 Node.js 模块	298
19.11.1 问题	268	22.3 package.json 文件	299
19.11.2 答案	269	22.4 文件夹结构	301
19.12 练习	269	22.5 开发和测试模块	302
第 20 章 流模块	270	22.6 添加可执行文件	304
20.1 流简介	270	22.7 使用面向对象或者基于原型的 编程	305
20.2 可读流	272	22.8 通过 GitHub 共享代码	306
20.3 可写流	275	22.9 使用 Travis CI	307
20.4 通过管道连接流	276	22.10 发布到 npm	309
20.5 流的 MP3	277	22.11 公开模块	310
20.6 小结	278	22.12 小结	310
20.7 问与答	278	22.13 问与答	310
20.8 测验	279	22.14 测验	311
20.8.1 问题	279	22.14.1 问题	311
20.8.2 答案	279		

22.14.2 答案	311	第 24 章 结合使用 Backbone.js 与	
22.15 练习	311	Node.js	326
第 23 章 使用 Connect 创建中间件	312	24.1 什么是 Backbone.js	326
23.1 什么是中间件	312	24.2 Backbone.js 如何工作	327
23.2 Connect 中的中间件	313	24.3 一个简单的 Backbone.js	
23.3 使用中间件的访问控制	317	视图	332
23.4 按 IP 地址限制访问	319	24.4 使用 Backbone.js 创建记录	336
23.5 将用户强制到单个域上	322	24.5 小结	337
23.6 小结	324	24.6 问与答	337
23.7 问与答	324	24.7 测验	338
23.8 测验	324	24.7.1 问题	338
23.8.1 问题	324	24.7.2 答案	338
23.8.2 答案	325	24.8 练习	338
23.9 练习	325		



第1部分 入门

第1章 Node.js 介绍

第2章 npm (Node 包管理器)

第3章 Node.js 的作用

第4章 回调 (Callback)



第 1 章

Node.js 介绍

在本章中你将学到：

- Node.js 是什么，为什么要创建它；
- 使用 Node.js 能创建的应用程序示例；
- 创建并运行第一个 Node.js 程序。

1.1 什么是 Node.js

2009 年 11 月 8 日，Ryan Dahl 在 jsconf.eu 这个关于 JavaScript 的会议上进行了一次演讲，将 Node.js 介绍给了 JavaScript 社区。他对于许多程序设计语言难以实现并发（同时做多件事情）并且经常导致糟糕的性能的问题颇为苦恼。他希望能够更容易地编写出快速的、支持许多用户并且高效地使用内存的联网软件，于是他创建了 Node.js。

在 Ryan Dahl 探索解决该问题的方法的同时，诸如 Apple 和 Google 这样的公司也开始在浏览器技术上大举投入。

鉴于 Google 依赖于浏览器来交付诸如 Gmail 这样的产品，Google 的工程师们创建了 V8，这是个给 Google Chrome 浏览器编写的 JavaScript 引擎，它也是专门为 Web 而设计的经过高度优化的软件。Google 希望 V8 能够发扬光大，于是将其以 BSD (Berkeley Software Distribution) 协议开源。

Ryan Dahl 决定使用 V8 引擎来创建 JavaScript 服务器端环境，这样做具有如下理由。

- V8 引擎极快。
- V8 专注于 Web，所以在处理超文本传输协议 (HTTP)、域名系统 (DNS) 和传输控制协议 (TCP) 等事务上驾轻就熟。

- JavaScript 在 Web 上人人皆知，所以大多数开发人员都能使用它。

从核心上说，Node.js 是个事件驱动的服务器端 JavaScript 环境。也就是说，我们可以像使用 PHP、Ruby 和 Python 语言那样，使用 JavaScript 创建服务器端的应用程序。对于网络以及创建与网络交互的软件，它尤为专注。

1.2 使用 Node.js 能做什么

Node.js 是个程序设计平台，只要有想法和足够的编程技艺，它就无所不能。它既可以创建对文件系统进行操作的小段脚本，也可以创建大规模的 Web 应用程序来运行整个业务。由于 Node.js 的独特设计，它非常适合于多人游戏、实时系统、联网软件和具有上千个并发用户的应用程序。

以下是一些使用 Node.js 的公司。

- LinkedIn
- eBay
- Yahoo!
- Microsoft

能使用 Node.js 创建的应用程序有：

- 实时多人游戏；
- 基于 Web 的聊天客户端；
- 将网络中的数据源进行合并的混搭软件（Mashup）；
- 单页面浏览器应用程序；
- 基于 JSON 的 API。

注意：什么是服务器端的 JavaScript？

大多数 Web 开发人员都熟悉于使用 JavaScript 操纵浏览器中的 Web 页面并与之交互。这就是通常所称的客户端 JavaScript，因为它发生在浏览器或者客户端。服务器端 JavaScript 发生在把页面发送给浏览器之前的服务器上。当然，使用的是同样的语言！

By the Way

1.3 安装并创建第一个 Node.js 程序

说得够多的了！现在来看看运行中的 Node.js 并编写你的第一个 Node.js 程序。首先得安装 Node.js。用于 Windows 和 OSX 的安装程序可以在 Node.js 的主页下载：<http://nodejs.org/>。要想在这些平台上安装 Node.js，只需下载相关文件并双击安装程序即可。如果使用 Linux 或者想手动编译 Node.js，请在 <https://github.com/joyent/node/wiki/installation> 上找操作指南。

1.3.1 验证 Node.js 正确安装

安装 Node.js 之后，应当验证其是否正确安装。我们需要使用终端来与 Node.js 交互。如果是在 OSX 下，可以在 Applications→Utilities→Terminal 找到终端应用程序。如果是在 Windows 下，可以通过按住 Windows 键并同时按 r 键，然后输入 cmd 来启动一个终端。在 Linux 下，终端应用程序通常称为 Terminal。

TRY IT YOURSELF

使用如下步骤来检查 Node.js 是否成功安装。

1. 打开终端并输入 node。
 2. 你应当看到一个提示符。
 3. 输入 1+1，可看到系统返回 2（见图 1.1）。
-

图 1.1

检查 Node.js 是否
成功安装



1.3.2 创建“Hello World” Node.js 程序

现在创建一个能启动 Web 服务器并显示 Hello World 的 Node.js 程序。

TRY IT YOURSELF

按照下列步骤来运行一个 Hello World 服务器。

1. 打开用来编写软件的文本编辑器，创建一个新文件。
2. 将程序清单 1.1 中的代码复制到该文件中（如果下载了本书代码，那么这段代码位于 hour01/example01）。

程序清单 1.1 Hello World 服务器

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

3. 将文件以 `server.js` 为名保存到计算机上。

4. 从终端运行这个程序：

```
node server.js
```

读者应当看到 `Server running at http://127.0.0.1:3000`。这表示服务器已经启动。

5. 打开 Web 浏览器并访问 `http://127.0.0.1:3000`。如果看到 `Hello World`，那么就成功创建了第一个 Node.js 程序（见图 1.2）。

6. 要想停止服务器运行，请返回终端并按 `Ctrl+C`。这将杀死（kill）Node.js 进程并停止服务器。

读者刚刚运行的就是服务器端的 JavaScript！



图 1.2
第一个 Node.js
程序！

1.4 小结

好极了！你刚刚创建并运行了第一个 Node.js 程序。虽然你现在不太可能以 Internet 百万富翁的身份退休，但以这个简单的示例为基础，在将来的几章里你将可以创建出更为复杂的应用程序，包括一个聊天服务器和一个实时的 Twitter 客户端。

除了创建一个简单的服务器以外，在本章我们还学到：Node.js 运行在 V8 引擎之上，这

是一个由 Google 开发的开源的 JavaScript 引擎。我们还了解到 Node.js 有多种用途，且精于创建有上千个并发用户的联网应用程序。

在随后的几章里，读者将探究 Node.js 与其他编程语言和框架的不同，尤其是那些让它运行神速的特性，还要探究编写代码来利用这些特性的方法。

1.5 问与答

问：我能在服务器上使用 JavaScript 吗？JavaScript 不是只能在浏览器上用吗？

答：JavaScript 绝对可以用在服务器上，而且，它的许多特性使其精于此道。编写服务端的 JavaScript 有许多好处，尤其在需要处理并发的时候。如果读者有使用诸如 jQuery 这样的框架编写 JavaScript 的经验，就会在 Node.js 中看到相似的模式。

问：创建 Web 应用程序，Node.js 比 PHP、Python、.NET 或 Ruby 好吗？

答：要评估哪个编程语言最好，就犹如试着说世界上哪个城市最好。这要看情况，要创建的应用程序类型决定你的选择。不过，可以这么说，Node.js 能胜任其他编程语言胜任的大多数事情，并且精于其他语言和平台所不精的领域。

问：是否需要对 Node.js 是个新平台而担心？

答：在编程平台方面，Node.js 相对年轻。开发的速度正在加快，世界各地的开发人员每天都在创建新的库。使用 Node.js 的优势在于，可以访问一个顶尖的并且从其他编程语言中受益良多的平台。涉足 Node.js 社区，现在正是最好的时候。

问：为了使用 Node.js，我是否需要是个 JavaScript 编程专家？

答：你需要使用 JavaScript 来编写 Node.js 应用程序，但完全没必要是个专家。JavaScript 这个语言成功的原因之一就是它的可访问性。就其本质而言，JavaScript 是个简单的语言，所以无需胆怯。此外，Node.js 使用大量模式，你将很快熟悉它们。你将在本书的剩余部分探究这些模式。

问：我是否应当相信与 Node.js 有关的那些夸夸其谈的宣传？

答：如果你购买了本书，那么你可能已经对 Node.js 有所耳闻。或许你在相关的博客中看到过下述描述：“Node.js 是下一个 Ruby on Rails”或者“Node.js 是个新的热门事物”。别全信。Node.js 是个编程平台，虽然它精于许多事情，但其他编程语言和平台还是会继续繁荣下去。读过本书之后，读者对于 Node.js 能提供什么会有自己的判断。

1.6 测验

在更多地了解了 Node.js 之后，这里有几个问题，用来帮助巩固新学的知识。

1.6.1 问题

1. Node.js 基于哪个 JavaScript 引擎？

- A. V8 JavaScript 引擎
 - B. SpiderMonkey
 - C. SquirrelFish
2. 谁创建了 Node.js?
- A. Ryan Adams
 - B. Sophie Dahl
 - C. Ryan Dahl
3. 额外加分：关于 Node.js，并发的含义是什么？
- A. 在同一时间运行多个程序
 - B. 一次做多件事情的能力
 - C. 一次只能执行一个操作

1.6.2 答案

1. A. Node.js 构建在 V8 JavaScript 引擎之上。这个引擎由 Google 创建并且在 BSD 协议下开源。

2. C. 要是 Node.js 是由英国偶像或者美国音乐人创建的，这当然会很惹人爱，但它的创建者是 Ryan Dahl。访问 Ryan 的主页 <http://tinyclouds.org/> 可以对他了解更多。

3. B. 如果你回答正确，应当给你一阵掌声！在接下来的几章中，你将学习更多关于并发的知识，因为这是 Node.js 的关键特性。

1.7 练习

- 1. 修改 Hello World 示例，将显示的文本改为“我在用 Node.js 写程序！”。
- 2. 修改 Hello World 示例，更改服务器的启动端口号。
- 3. 使用 Hello World 示例，熟悉通过输入 `node server.js` 在终端中启动 Node.js 并且使用 `Ctrl+C` 停止程序的方法。

第 2 章

npm (Node 包管理器)

在本章中你将学到:

- 使用 npm 为 Node.js 安装模块;
- 为 Node.js 的应用程序查找模块;
- 在 Node.js 应用程序中使用模块;
- 查找 Node.js 模块的文档;
- 使用 package.json 文件。

2.1 npm 是什么

npm (Node Package Manager, Node 包管理器) 是 Node.js 的包管理器。它允许开发人员在 Node.js 应用程序中创建、共享并重用模块。它也可用于共享完整的 Node.js 应用程序。模块就是可以在不同项目中重用的代码库。如果你使用其他语言写过程序, 那么 npm 就类似于 Ruby 中的 RubyGems、Perl 中的 CPAN、Python 中的 pip 或者 PHP 中的 PEAR。

典型的模块示例包括:

- 用于与数据库交互的库;
- 验证输入数据的库;
- 分析 yaml 文件的库。

对经验不足的开发人员来说, 使用 Node.js 模块, 学习更有经验的开发人员的技能, 是学习 Node.js 的绝好方法。

无论用 Node.js 来做什么, 都应当熟悉 npm 及其所能提供的库。

注意：大多数模块都是开源的

Node.js 社区在开源授权协议下发布了大多数模块。这也就意味着模块可以自由安装、修改和分发。

2.2 安装 npm

如果使用来自 <http://nodejs.org> 的安装程序安装了 Node.js，那么 npm 就已经安装好了。如果从源代码安装编译了 Node.js，那么可以在 <http://npmjs.org> 找到 npm 的操作指南。完成安装之后，打开终端输入 npm（见图 2.1）来检查其是否安装成功。系统应当返回一些帮助文本。

A terminal window titled 'Terminal -- zsh -- 80x25' showing the output of the 'npm' command. The output lists various npm commands and their functions, such as 'adduser', 'apihelp', 'author', etc. It also provides instructions on how to specify configurations in an ini-formatted file or via command line options like '--key value'.

```
Terminal -- zsh -- 80x25
[george@auster ~]$ npm

Usage: npm <command>

where <command> is one of:
  adduser, apihelp, author, bin, bugs, c, cache, completion,
  config, deprecate, docs, edit, explore, faq, find, get,
  help, help-search, home, i, info, init, install, la, link,
  list, ll, ln, ls, outdated, owner, pack, prefix, prune,
  publish, r, rb, rebuild, remove, restart, rm, root,
  run-script, s, se, search, set, show, star, start, stop,
  submodule, tag, test, un, uninstall, unlink, unpublish,
  unstar, up, update, version, view, whoami

npm <cmd> -h      quick help on <cmd>
npm -l            display full usage info
npm faq           commonly asked questions
npm help <term>   search for help on <term>
npm help npm      involved overview

Specify configs in the ini-formatted file:
  /Users/george/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config
[george@auster ~]$
```

图 2.1

检查 npm 是否正确安装

警告：使用 sudo

如果使用 UNIX 系统（Mac OSX 或者 Linux），由于安全原因，强烈建议不要使用 sudo 权限来安装 npm。

2.3 安装模块

安装了 npm 之后，就可以从终端开始安装模块了：

```
npm install [module_name]
```

这个命令向 npm 注册服务器（registry server）发送请求，将某个模块的最新版本下载到计算机上。可看到用于确认文件成功下载的输出：

```
module_name@1.2.0 ./node_modules/module_name
```

这行输出告诉我们以下三件事情。

- 成功下载的模块的名称。
- 模块的版本。
- 模块的下载位置。

By the Way

注意：确认与网络连接！

由于模块 npm 注册 (registry) 和源文件都托管在 Internet 上，所以通过 npm 安装模块时需要与 Internet 连接。

2.4 使用模块

要在 Node.js 应用程序中使用模块，在下载它们之后必须请求 (require) 它们。在应用程序中请求一个模块的方法如下：

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

现在，当应用程序运行的时候，它将在源文件中找库 (library) 并将其包含在应用程序中。通常我们要使用这个模块，所以我们将其赋予一个变量。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour02/example01 找到。要在 Node.js 中使用第三方模块，请遵照如下这些步骤进行。

1. 打开文本编辑器，编写下列代码：

```
var _ = require('underscore');
_.each([1, 2, 3], function(num){
  console.log("underscore.js says " + num);
});
```

2. 将文件保存成：

```
foo.js
```

3. 使用终端安装 underscore 模块：

```
npm install underscore
```

Watch Out!

警告：检查一下目录

为了让 npm 将模块安装在正确的地方，在运行上述命令时必须位于项目文件夹中。

4. 从终端运行程序：

```
node foo.js
```

该程序应当使用 underscore 模块数到 3 (见图 2.2)：

```
underscore.js says 1
underscore.js says 2
underscore.js says 3
```



图 2.2

使用 underscore
库数到 3

2.5 如何找模块

既然可以安装 Node.js 模块，就会想探究有哪些模块是可用的。Node.js 有一个充满生机的开发者社区，模块的建立和维护每天都在发生。

2.5.1 官方来源

你可以在许多地方搜索模块。首先，在 <http://search.npmjs.org/> 上有一个官方的基于 Web 的 npm 搜索工具（见图 2.3）。这是寻找第三方模块的正统资源。请尝试搜索“irc”，读者将看到有许多和 irc 有关的模块以及模块的简要介绍。

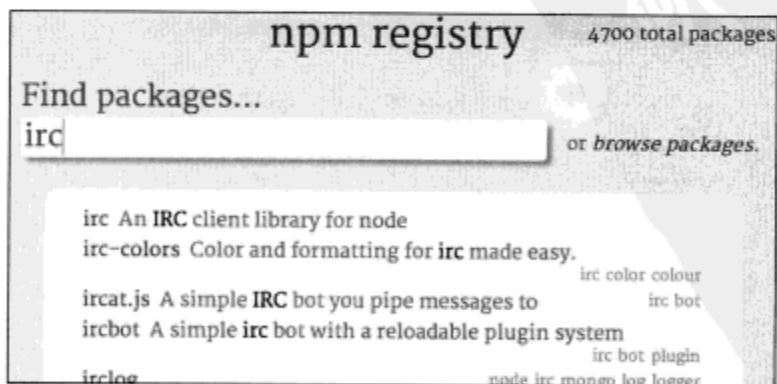


图 2.3

在 <http://search.npmjs.org/> 搜索
模块

一旦找到了要找的模块，就可以按先前的方法在命令行下安装它。

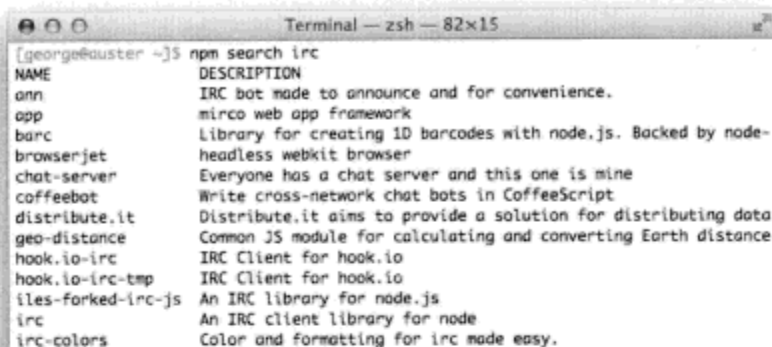
也可以在终端中直接使用 npm 命令行工具来搜索 Node.js 模块（见图 2.4）：

```
npm search irc
```

注意在第一次运行这个命令的时候，可能需要花一些时间来创建索引。

图 2.4

通过命令行来搜索
模块



这样搜索的不仅仅是模块名，还有模块的描述。还可以通过使用空格隔关键词的方法进行跨关键字搜索：

```
npm search socket connect
```

2.5.2 非官方来源

还有许多非官方的来源可用于搜索 npm 模块。这些来源使用官方的注册数据，但在模块上增加了一些额外的信息。

Blagovest Dachev 创建了 <http://blago.dachev.com/modules> 站点，该站点可以提供 GitHub 上某个项目的围观者数量、分叉 (fork) 数量以及问题数量 (见图 2.5)。如果了解项目的活跃程度以及最近的更新时间，这会极其有用。

图 2.5

在 <http://blago.dachev.com/modules> 搜索模块



Eirik Brandtzag 创建了一个类似的工具，地址是 <http://eirikb.github.com/nipster/>，它在 Node.js 社区内也广受欢迎。它根据 GitHub 上的分叉数量对项目评级，而分叉数量是项目流行程度的风向标 (见图 2.6)。

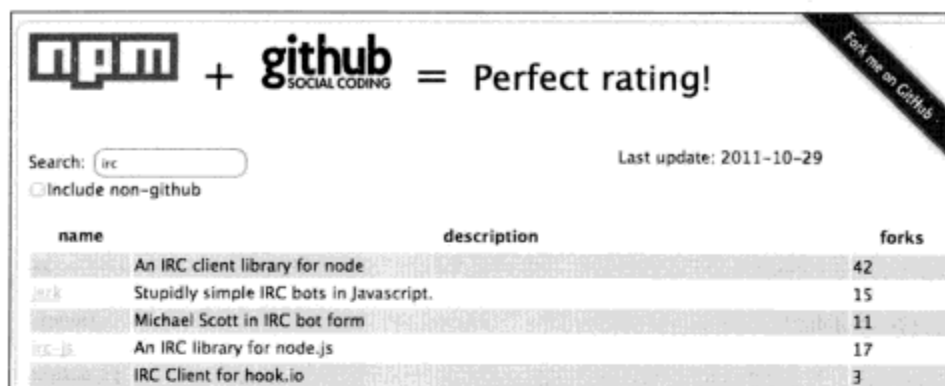


图 2.6

在 Nipster 上搜索
模块

另外一个优秀的搜索工具可在 <http://toolbox.no.de/> 上找到。

警告：使用模块的注意事项

尽管模块可通过 npm 获得，但这并不意味着它是得到良好测试的或者是稳定的。请使用模块的下载量和项目的问题数量作为模块的可靠性和成熟度的大致指南。

**Watch
Out!**

2.6 本地和全局的安装

可以使用 npm 以两种方式安装模块，理解它们的工作方式很重要。

2.6.1 本地安装

本地安装意味着库将安装在项目本地的一个名为 `node_modules` 的文件夹下（在安装 `underscore` 库时就是如此）以便项目使用。这是默认行为，只要运行如下命令，就是如此：

```
npm install [module_name]
```

如果 Node.js 应用程序的名称是 `foo.js`，这将产生如下的文件夹结构：

```
- foo.js
- node_modules/
  - module_name
```

这是最为常见并且推荐的安装 Node.js 模块的方法。

2.6.2 全局安装

有些模块带有可执行文件，你希望能够在文件系统的任何一个位置都能运行这些可执行文件。Express 就是一个可能需要全局安装的模块示例。Express 是 Node.js 的一个 Web 开发框架，带有一个能够创建站点骨架的生成器。

要全局安装模块，只需在安装时加上 `-g` 标记。


```
npm install -g express
```

全局安装一个模块意味着可以在文件系统的任何位置运行它。

**By the
Way**

注意：任何人都可以是 Node.js 模块的作者

npm 注册库对世界上的任何人开放以便提交模块。而且没有审核过程，给注册库提交代码也很简单，因此可以方便地与其他开发人员共享。

2.7 如何找模块文档

在知道如何安装与寻找模块之后，我们还需要知道如何使用它们。通常来讲，Node.js 模块都有编写良好的文档，可以通过运行如下命令在浏览器中查看模块文档：

```
npm docs [module_name]
```

这会打开浏览器并进入模块作者所提供的文档页面。这通常是一个指向 GitHub 上某个 README 文件的链接。要查看 `underscore.js` 的文档，请运行：

```
npm docs underscore
```

通过运行如下命令也可以查看项目的 bug：

```
npm bugs underscore
```

这会打开浏览器并进入模块作者所提供的问题页面。

随着读者越来越有经验，就会发现阅读模块的源代码是理解其作用的最快方法。

```
npm edit underscore
```

这里有个“意外”，那就是这个命令必须在 Node.js 项目文件夹的根目录下运行，而且该模块必须已经被下载到了 `node_modules` 文件夹中。

**By the
Way**

注意：你可以更新文档

大多数 npm 模块都托管在 GitHub 上，模块作者会对模块文档的贡献者极为感激。如果你在文档中发现了某些不足，可以考虑帮助作者改进文档。

2.8 使用 package.json 指定依赖关系 (dependency)

在开发 Node.js 应用程序的时候，毫无疑问要使用模块。一个一个地安装模块会是件耗时的工作，而且容易出错。比如，要是忘了安装某个模块该怎么办？

不必担心。npm 允许开发人员使用 `package.json` 文件来指定在应用程序中要用的模块，并且通过单个命令来安装它们：

```
npm install
```

这样做具有如下优势：

- 无需一个一个地安装模块;
- 其他开发人员可以很容易地安装你的应用程序;
- 应用程序的依赖关系储存在单一的地方。

回到我们安装 `underscore` 模块那个例子上。现在可以给项目添加一个 `package.json` 文件了。`package.json` 文件仅包含以特定格式表示的项目信息。一个最小的 `package.json` 文件会是这样:

```
{
  "name": "example02",
  "version": "0.0.1",
  "dependencies": {
    "underscore": "~1.2.1"
  }
}
```

我们后续会了解 `package.json` 文件的细节, 而目前, 我们要关心的关键部分是依赖关系。我们指定这个应用程序需要 `underscore` 模块。

TRY IT YOURSELF

如果已经下载了本书的代码示例, 那么这段代码位于 `hour02/example02` 中。

请按照如下步骤使用 `package.json` 文件指定依赖关系。

1. 打开文本编辑器并写入以下内容:

```
var _ = require('underscore');
_.each([1, 2, 3], function(num){
  console.log("underscore.js says " + num)
});
```

2. 将文件保存为:

`foo.js`

3. 创建一个名为 `package.json` 的新文件并加入如下内容:

```
{
  "name": "example02",
  "version": "0.0.1",
  "dependencies": {
    "underscore": "~1.2.1"
  }
}
```

4. 将文件保存为:

`package.json`

5. 确认当前位于与 `foo.js` 相同的目录下, 运行:

```
npm install
```

读者应当看到 `underscore` 库安装在了 `node_modules` 文件夹下。

就算只安装一个模块，我们也强烈建议使用 `package.json` 文件来管理 Node.js 模块。

2.9 小结

在本章中，我们学习了如何安装 Node.js 的包管理器 npm。此外，也学习了如何使用 npm 安装模块以及如何在 Node.js 应用程序中使用它们。我们学习了本地安装模块和全局安装模块的区别，以及如何寻找模块文档。最后，我们学习了如何使用 `package.json` 来声明应用程序中的依赖关系。

2.10 问与答

问：我刚刚开始学习使用 Node.js，我应当使用模块吗？

答：是的。通过使用模块可以快速地为应用程序加入许多功能。模块通常可以为开发人员除去常见的困难。比如，Express 模块让使用 Node.js 进行 Web 开发变得简单。

问：有许多模块可以解决我的问题，哪个模块最好？

答：你应当使用社区中最为流行的模块。可以通过使用位于 <http://blago.dachev.com/modules> 和 <http://eirikb.github.com/nipster/> 的搜索工具来评估模块的流行程度。GitHub 上围观者数量最能衡量流行程度。

问：我应该使用第三方模块还是自己编写代码？

答：编写自己的代码是理解问题最好的方式，但在许多时候，你的问题已经有人解决了，此时可以考虑在应用程序中使用第三方模块。许多开发人员最后还会为其使用的模块修复 bug 并贡献新功能。

问：使用 Node.js 模块是否需要付费？

答：不需要。Node.js 模块几乎总是在开源授权协议下发布的，可免费使用。通常可以在模块的主页上确认协议细节。如果有任何疑问，可联系模块作者。

问：我应当手工安装模块，还是使用 `package.json` 文件？

答：只要有可能，就应当使用 `package.json` 文件来管理 Node.js 模块。这样，其他开发人员就可以很容易地安装你的应用程序，而且当你的应用程序越来越大时，你也就不需要一个一个地安装模块。

2.11 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

2.11.1 问题

1. 什么是模块？

2. 模块的本地安装和全局安装有什么区别？
3. 在管理模块时，使用 `package.json` 文件有什么优势？

2.11.2 答案

1. 模块是可重用的代码库。比如用来与数据库交互的模块、支持 Web 开发的模块以及通过 Web 套接字协助通信的模块。

2. 本地安装模块意味着模块会被安装在项目内名为 `node_modules` 的文件夹中，而且它只可在该项目中使用。全局安装模块意味着该模块可在系统的任何一个地方使用。作为一条经验法则，请本地安装 Node.js 模块。

3. 使用 `package.json` 文件意味着我们无需记忆应用程序会依赖于哪些模块。其他开发人员会发现，可以很简单地安装你的应用程序，而且你可以使用 `npm install` 来安装你的应用程序运行时所需的所有模块。

2.12 练习

1. 使用本章所描述的一个工具搜索 “`template engine`”，并尝试评估一下哪个最流行。
2. 使用本章所描述的一个搜索工具搜索并安装 “`coffee-script`” 模块。在一个新的 Node.js 项目中本地安装这一模块。
3. 使用本章所描述的一个工具搜索并安装 “`express`” 模块。在系统上全局安装这一模块。请确认可以在系统的任何位置运行 `express` 命令。
4. 创建带有 `package.json` 文件的一个新 Node.js 项目，并在 `package.json` 文件中指定一个想要安装的依赖模块。请确认可以通过运行 `npm install` 来安装这一模块。

第3章

Node.js 的作用

在本章中你将学到：

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

3.1 设计 Node.js 的目的

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

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

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

3.2 理解 I/O

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

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

I/O 的思想可以通过在终端里运行一个程序来演示（见图 3.1）：

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

在 Mac OSX 或 Linux 下输出会是：

```
each peach pear plum
```

在 Windows 下输出会是：

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



图 3.1
数据流经计算机

TRY IT YOURSELF

我们在这里演示一下输入和输出。

1. 打开终端并输入如下命令：

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

2. 可看到该行内容的回显。
3. 注意在这里键盘是输入。
4. 注意在这里显示器是输出。

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

1. 文本字符串被传递给 `echo` 程序（输入）。
2. 文本字符串流经 `echo` 程序的逻辑。
3. `echo` 程序将其输出到终端（输出）。

3.3 处理输入

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

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

- 姓；
- 名；
- 电子邮件地址。

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

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

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

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

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

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

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

- 两名游戏者；

- 两个 Wii 遥控器，每个各有 8 个数字按钮。

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

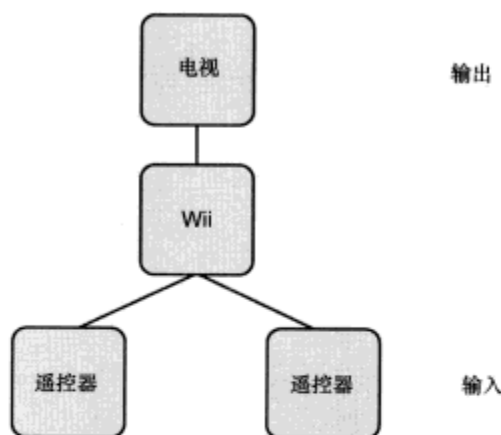


图 3.2
两个输入让事情复杂

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

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

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

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

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

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

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

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

图 3.3

输入和输出变得复杂

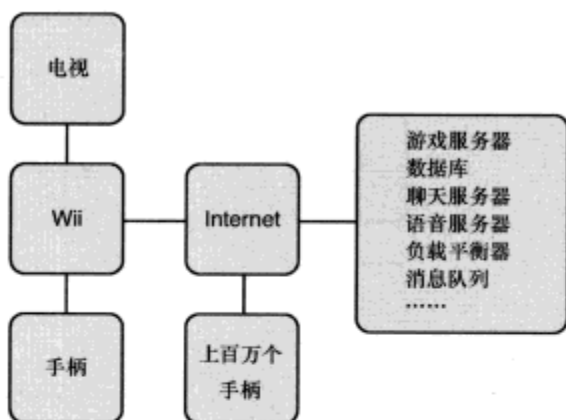


图 3.4

提供 HTML 页面服务的 Web 服务器



图 3.5

增加数据库服务器



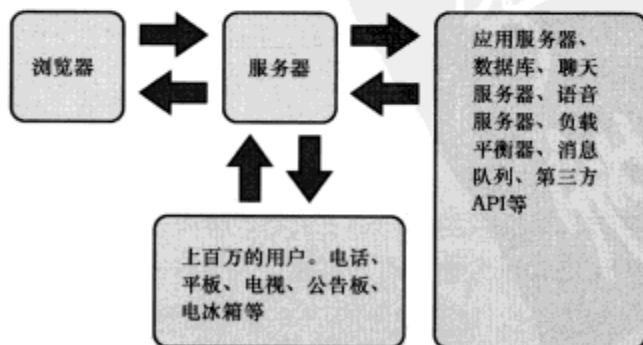
今天，Web 应用程序的设计要复杂得多得多。I/O 是碎片化的，而且 I/O 操作更为频繁。

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

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

图 3.6

复杂的 Web 应用程序



3.4 联网的 I/O 是不可预测的

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

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

程序清单 3.1 演示网络 I/O

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

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

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

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

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

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

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

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

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

➤ 数据要传输多远。

TRY IT YOURSELF

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

1. 将程序清单 3.1 中的代码复制到系统上以 `app.js` 为名的文件中。
2. 从终端运行它：

```
node app.js
```

3. 观察来自服务器的响应。
4. 从终端再次运行它：

```
node app.js
```

5. 检查输出并比较响应时间。
-

这个简单的示例演示了与时间相关时，基于网络的 I/O 是不可预测的。

3.5 人类是不可预测的

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

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

如果读者更熟悉 jQuery，那么以 jQuery 写同样的代码就会是：

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

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

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

TRY IT YOURSELF

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

1. 创建一个名为 `index.html` 的新文件并加入下列内容：

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

2. 使用 Web 浏览器打开这个文件。
3. 单击按钮。注意随着单击事件的触发会有提示框显示。
4. 鼠标移过文本段落。注意随着鼠标移过（mouseover）事件的触发会有提示框显示。

3.6 处理不可预测性

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

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

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

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

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

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

By the Way

注意：有一些对 Node.js 不友好的反应

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

3.7 小结

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

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

3.8 问与答

问：我在开发内容不多的小网站，Node.js 适合吗？

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

问：并发似乎难以理解，如果我不能完全理解它会有问题吗？

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

问：I/O 与我在 Web 上看到的.io 域名有关吗？

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

3.9 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

3.9.1 问题

1. 与 Web 站点只是 HTML 文档那个时候不同，现在的 Web 应用程序有哪些方面的变化？
2. 你觉得 Node.js 适合哪些范畴？
3. 为什么 JavaScript 是一个事件驱动的语言？

3.9.2 答案

1. Web 应用程序变得更为复杂的地方包括了脚本语言和数据库的加入等。越来越多的数据现在分布在 Web 的不同地方，使用 API 和网络粘合在一起。
2. 当应用程序需要在网络上发送和接收数据时 Node.js 最为适合。这可能是第三方的 API、联网设备或者浏览器与服务器之间的实时通信。
3. JavaScript 围绕着最初与文档对象模型（DOM）相关的事件构架。开发人员可以在事件发生时做事情。这些事件有用户单击一个元素、页面完成加载等。使用事件，开发人员可以编写事件的侦听器，当事件发生时被触发。

3.10 练习

1. 如果读者开发过网站或者软件，请选择一个然后画出其输入输出图。理解代表输入和输出的设备或事物。
2. 选一个你最喜欢的计算机游戏并尝试理解该游戏的输入和输出。有多少个输入？是否有会同时发生或者几乎同时发生的事情？

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



第 4 章

回调 (Callback)

本章介绍如下内容：

- 回调是什么，它们在 JavaScript 中如何使用；
- Node.js 中如何使用回调；
- 同步和异步编程的区别；
- 事件循环是什么。

4.1 什么是回调

如果读者熟悉 jQuery，那么使用回调就可能是家常便饭。回调指的是将一个函数作为参数传递给另一个函数，并且通常在第一个函数完成后被调用。在下面的 jQuery 示例中，使用 jQuery 的 `hide()` 方法隐藏了一个段落标记。这个方法可以使用一个可选的回调函数作为参数。如果提供了回调函数作为参数，那么当段落隐藏完成后它就会被调用。这就让我们有可能在隐藏完成后做一些事情——在本例中，我们显示一个提示。

```
$('#p').hide('slow', function() {  
    alert("The paragraph is now hidden");  
});
```

不过，回调是可选的。如果不需要回调，可以这么写代码：

```
$('#p').hide('slow');
```

比较两段代码示例，第一段加入了一个匿名函数作为第二个参数，并且在第一个函数完成后被调用。在第二个示例中没有回调。由于 JavaScript 中函数是第一类（first-class）对象，它们可以以这种方式作为参数传递到其他函数中。于是我们就可以编写这样的代码：执行 A，并且在执行完 B 之后再执行 A。为了演示编写带与不带回调的代码之间的区别，我们在浏览器中看两

段jQuery 示例。

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Callback Example</title>
  </head>
  <body>
    <h1>Callback Example</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer ut augue
    et orcialiquam aliquam. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Pellentesquehabitant morbi tristique senectus et netus et malesuada fames ac turpis
    egestas. Etiamfermentum dictum convallis. In at diam et orci sodales sollicitudin.
    Sed viverra, orcisit amet faucibus condimentum, nibh augue consectetur ipsum, eu
    tristique dolor diaminterdum tellus. Donec in diam nunc. Nulla sollicitudin elit
    sit amet neque elementum accursus nibh lobortis.</p>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.
    js"></script>
    <script>
      $(function () {
        $('p').hide('slow');
        alert("The paragraph is now hidden");
      });
    </script>
  </body>
</html>
```

在这个示例中，提示框会在段落隐藏之前显示。这是因为 JavaScript 在 hide()方法执行之后直接执行 alert，即使段落尚未隐藏好。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour04/example01 找到。下列步骤帮助读者理解 JavaScript 中的代码执行。

1. 创建一个名为 index.html 的文件并将下列代码复制到文件中：

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Callback Example</title>
  </head>
  <body>
    <h1>Callback Example</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer
    ut augue et orcialiquam aliquam. Lorem ipsum dolor sit amet, consectetur
    adipiscing elit. Pellentesquehabitant morbi tristique senectus et netus et
    malesuada fames ac turpis egestas. Etiamfermentum dictum convallis. In at diam
    et orci sodales sollicitudin. Sed viverra, orcisit amet faucibus condimentum,
    nibh augue consectetur ipsum, eu tristique dolor diaminterdum tellus. Donec
    in diam nunc. Nulla sollicitudin elit sit amet neque elementum accursus nibh
    lobortis.</p>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
    min.js"></script>
    <script>
      $(function () {
        $('p').hide('slow');
        alert("The paragraph is now hidden");
      });
    </script>
  </body>
</html>
```

```

    });
  </script>
</body>
</html>

```

2. 在 Web 浏览器中打开这一文件。
3. 可看到提示框显示，并且在过了一会儿之后，段落会被隐藏（见图 4.1）。



图 4.1

提示框显示在
hide()方法完成之前

在上面的示例中，我们看到提示信息在段落隐藏之前显示。这是因为 `alert` 在段落完成隐藏之前被执行。对于使用回调来确保在某件事情完成之后执行另一件事情，这是个绝好的示例。以下这个相同的示例使用的是回调。

```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Callback Example</title>
  </head>
  <body>
    <h1>Callback Example</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer ut augue
    et orcialiquam aliquam. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Pellentesquehabitant morbi tristique senectus et netus et malesuada fames ac turpis
    egestas. Etiamfermentum dictum convallis. In at diam et orci sodales sollicitudin.
    Sed viverra, orcisit amet faucibus condimentum, nibh augue consectetur ipsum, eu
    tristique dolor diaminterdum tellus. Donec in diam nunc. Nulla sollicitudin elit
    sit amet neque elementum accursus nibh lobortis.</p>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.
    js"></script>
    <script>
      $(function () {
        $('p').hide('slow', function() {
          alert("The paragraph is now hidden");
        });
      });
    </script>
  </body>
</html>

```

现在这个示例在段落隐藏之后显示提示，提示框正确地出现在事件发生之后而不是之前。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour04/example02` 找到。按照如下步骤在 jQuery 中使用回调。

1. 创建一个名为 `index.html` 的文件并将下列代码复制到文件中：

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Callback Example</title>
  </head>
  <body>
    <h1>Callback Example</h1>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer
ut augue et orcialiquam aliquam. Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Pellentesquehabitans morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Etiamfermentum dictum convallis. In at diam
et orci sodales sollicitudin. Sed viverra, orcisit amet faucibus condimentum,
nibh augue consectetur ipsum, eu tristique dolor diaminterdum tellus. Donec
in diam nunc. Nulla sollicitudin elit sit amet neque elementum accursus nibh
lobortis.</p>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
min.js"></script>
    <script>
      $(function () {
        $('p').hide('slow', function() {
          alert("The paragraph is now hidden");
        });
      });
    </script>
  </body>
</html>
```

2. 在 Web 浏览器中打开这个文件。
3. 在过一会儿之后，可看到段落被隐藏，然后显示提示框（见图 4.2）。

图 4.2

提示框在 `hide()` 方法完成之后显示



4.2 剖析回调

我们已经看到了在 jQuery 中回调的效果，但在 JavaScript 内部它是如何工作的？这里要掌握的关键概念是：函数可以作为参数传递到另一个函数中，然后被调用。在下面的示例中，创建了一个名为 `haveBreakfast` 的函数，它带有两个参数：“food”和“drink”。第三个参数是“callback”，这个参数必须是个函数。`haveBreakfast` 函数将所吃的东西记录到控制台中然后调用作为参数传递给它的回调函数。

```
function haveBreakfast(food, drink, callback) {
  console.log('Having breakfast of ' + food + ', ' + drink);
  if (callback && typeof(callback) === "function") {
    callback();
  }
}
```

要使用 `haveBreakfast` 函数，我们传递被吃的食物和被饮用的饮料两个变量（在这里是吐司和咖啡），传递一个函数作为第三个参数。这是个回调函数，将会在 `haveBreakfast()` 函数内被执行。

```
haveBreakfast('toast', 'coffee', function() {
  console.log('Finished breakfast. Time to go to work!');
});
```

这样的回调模式在 Node.js 中到处使用，所以要花一些时间来理解这里所发生的一切。在本例中，回调函数在 `haveBreakfast()` 函数完成后做了某些事情。确切地要做的事情定义在作为回调传递进来的函数中。在这里要显示去工作！运行这一脚本会显示如下输出：

```
Having breakfast of toast, coffee
Finished breakfast. Time to go to work!
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour04/example03` 找到。以下步骤阐明了回调。

1. 创建一个名为 `app.js` 的文件并将下列代码复制到文件中：

```
function haveBreakfast(food, drink, callback) {
  console.log('Having breakfast of ' + food + ', ' + drink);
  if (callback && typeof(callback) === "function") {
    callback();
  }
}

haveBreakfast('toast', 'coffee', function() {
  console.log('Finished breakfast. Time to go to work!');
});
```

2. 运行该文件：

```
node app.js
```

3. 首先可看到“Having breakfast of toast, coffee”，然后回调会被触发，打印出“Finished breakfast. Time to go to work!”。
-

4.3 Node.js 如何使用回调

Node.js 到处使用回调,尤其在有 I/O(输入/输出)操作发生的地方。下面是一个在 Node.js 中使用 `filesystem` 模块从磁盘上读入文件内容的示例。

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

fs.readFile('somefile.txt', 'utf8', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

以下是所发生的事情。

1. `fs` (`filesystem`) 模块被请求,以便在脚本中使用。
2. 将文件系统上的文件路径作为第一个参数提供给 `fs.readFile` 方法。
3. 第二个参数是 `utf8`, 表示文件的编码。
4. 将回调函数作为第三个参数提供给 `fs.readFile` 方法。
5. 回调函数的第一个参数是 `err`, 用于保存在读取文件时返回的错误。
6. 回调函数的第二个参数是 `data`, 用于保存读取文件所返回的数据。
7. 一旦文件被读取,回调就会被调用。
8. 如果 `err` 为真,那么就会抛出错误。
9. 如果 `err` 为假,那么来自文件的数据就可以使用。
10. 在本例中,数据会记录到控制台上。

读者将会看到在 Node.js 中以这种方式一次又一次地使用回调。另外一个示例是 `http` 模块。`http` 模块使得开发人员可以创建 `http` 客户端和服务端。读者已经在 Hello World 服务器中看过了 `http` 模块。`http` 模块中的 `http.get()` 方法可用来请求 Web 服务器获得响应数据以便使用。

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

http.get({ host: 'shapedshed.com' }, function(res) {
  console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
  console.log("Got error: " + e.message);
});
```

以下是这段代码的解释。

1. 请求 `http` 模块,以便在脚本中使用。
2. 给 `http.get()` 方法提供两个参数。
3. 第一个参数是选项对象。在本示例中,要求获取 `shapedshed.com` 的主页。
4. 第二个参数是一个以响应作为参数的回调函数。
5. 当远程服务器返回响应时,会触发回调函数。
6. 在回调函数内记录响应状态码,如果有错误的话可以记录下来。

回调函数的调用发生在远程服务器发回响应之后而不是之前。这个示例基本上演示了 Node.js 想要做成的全部事情。由于这段代码必须进入网络 (Internet) 来获取数据, 所以就不可能确切知道什么时候数据能返回, 甚至都不能确认数据会不会返回。尤其在从多个来源和多个网络中获取数据时, 由于数据返回的时间的不可预测本质, 要编写代码对此作出响应将是困难的。Node.js 是对这一问题的响应, 它以提供一个创建联网应用程序的平台为目标。回调是 Node.js 实现网络编程的关键方法, 因为回调让代码在其他事件发生的时候能够运行 (在本例中, 也就是数据从 shapedshed.com 返回的时候)。当事件发生时, 我们称回调被“触发 (fired)”, 从而导致回调函数被调用。

在程序清单 4.1 中, 4 个不同的 I/O 操作都在发生, 它们都使用回调。

程序清单 4.1 演示网络 I/O 和回调

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

http.get({ host: 'shapedshed.com' }, function(res) {
  console.log("Got a response from shapedshed.com");
}).on('error', function(e) {
  console.log("There was an error from shapedshed.com");
});

fs.readFile('file1.txt', 'utf8', function (err, data) {
  if (err) throw err;
  console.log('File 1 read!');
});

http.get({ host: 'www.bbc.co.uk' }, function(res) {
  console.log("Got a response from bbc.co.uk");
}).on('error', function(e) {
  console.log("There was an error from bbc.co.uk");
});

fs.readFile('file2.txt', 'utf8', function (err, data) {
  if (err) throw err;
  console.log('File 2 read!');
});
```

当代码运行的时候, 它做如下事情。

1. 获取 shapedshed.com 的主页。
2. 读取 `file1.txt` 的内容。
3. 获取 bbc.co.uk 的主页。
4. 读取 `file2.txt` 的内容。

看看这个示例, 你能说出哪个操作会先返回吗? 这样的猜测显然不错: 从磁盘上读取的两个文件很可能会先返回, 因为无需进入网络。但仅此而已, 我们难以说出被读取的文件哪个会先返回, 因为我们不知道文件的大小。至于对两个主页的获取, 脚本要进入网络, 而响应时间则依赖于许多难以预测的事情。Node.js 进程在还有已经注册的回调尚未触发之前将不会退出。回调首先是负责解决不可预测性的方法, 它也是处理并发 (或者说一次做超过一件

事情) 的高效方法。

TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour04/example04` 找到。以下步骤演示了回调。

1. 将程序清单 4.1 中的代码复制到系统中名为 `app.js` 的文件里。

2. 创建两个分别名为 `file1.txt` 和 `file2.txt` 的文件并给它们加入一些文本。读者可以从 <http://www.lipsum.com/> 取一些拉丁文本。

3. 运行这个脚本:

```
node app.js
```

4. 可以看到, 来自文件读取的回调会首先触发, 然后是两个 `http.get` 操作。

5. 多运行几次这个脚本。思考一下这些事件是如何异步发生的, 回调是如何用于在操作完成后做其他事情的。

4.4 同步和异步代码

Node.js 运行在单一的进程中并且要求开发人员使用异步编码风格。在上一个示例中, 我们看到 4 个操作如何通过回调模式异步执行。不过以异步方式编程并不是 Node.js 或者 JavaScript 特有的。这是一种编程风格。

同步的代码意味着每一次执行一个操作, 在一个操作完成之前, 代码的执行会被阻塞, 无法移到下一个操作上。程序清单 4.2 演示的是阻塞代码, 它演示两个使用 Node.js 进行的操作: 获取一个 Web 页面并且从第三方 API 获取数据。这个示例模拟了这些花费很多时间的操作及其对代码执行的影响。请注意, 在这个示例当中, `sleep()` 函数只是模拟完成这些操作所需的时间花销的手段。

程序清单 4.2 同步 (或者阻塞) 代码

```
function sleep(milliseconds) {
  var start = new Date().getTime();
  while ((new Date().getTime() - start) < milliseconds){
  }
}

function fetchPage() {
  console.log('fetching page');
  sleep(2000); // simulate time to fetch a web page
  console.log('data returned from requesting page');
}

function fetchApi() {
  console.log('fetching api');
  sleep(2000); // simulate time to fetch from an api
  console.log('data returned from the api');
}

fetchPage();
fetchApi();
```

在这个示例中，`fetchPage()`函数模拟从 Internet 上获取一个页面，而 `fetchApi()`函数模拟从第三方 API 获取数据。它们并不获取实际的数据，而是演示当响应所需的时间是个未知量时使用同步编程风格会发生什么。

如果运行这一示例，将看到如下输出：

```
fetching page
page fetched
fetching api
data returned from the api
```

当脚本运行时，`fetchPage()`函数会被调用，直到它返回之前，脚本的运行是被阻塞的。在 `fetchPage()`函数返回之前，程序是不能移到 `fetchApi()`函数中的。这称为阻塞操作，因为这种风格的代码阻塞了进程的运行直到函数返回。一件事情在另外一件事情之后有效发生。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour04/example05` 找到。以下步骤演示同步或者阻塞代码。

1. 将程序清单 4.2 中的代码复制到系统中名为 `app.js` 的文件里。
2. 运行这一脚本：

```
node app.js
```

3. 可看到从 Internet 上获取页面的模拟，当这一操作完成后做的是从 API 获取数据的模拟过程。这里要理解的要点是，两个操作一个接一个发生，一个函数完成之前，代码的执行是受阻塞的。

Node.js 几乎从不使用这种编码风格，而是异步地调用回调。程序清单 4.3 演示了同样的操作，但是使用了 Node 的异步风格。注意这个示例通过网络对某个 Web Service 进行了请求来模拟慢的响应效果，而没有使用模拟的 `sleep()`函数。

程序清单 4.3 异步（或非阻塞）代码

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

function fetchPage() {
  console.log('fetching page');
  http.get({ host: 'trafficjamapp.herokuapp.com', path: '/?delay=2000' },
    function(res) {
      console.log('data returned from requesting page');
    }).on('error', function(e) {
      console.log("There was an error" + e);
    })
};

function fetchApi() {
  console.log('fetching api');
```

```

    http.get({ host: 'trafficjamapp.herokuapp.com', path: '/?delay=2000' },
      function(res) {
        console.log('data returned from the api');
      }).on('error', function(e) {
        console.log("There was an error" + e);
      })
    );
  }

  fetchPage();
  fetchApi();

```

运行这段代码的时候，就不再等待 `fetchPage()` 函数返回了，`fetchApi()` 函数随之立刻被调用。这是可能的，因为代码通过使用回调，是非阻塞的了。一旦调用了，两个函数都会侦听远程服务器的返回，并以此触发回调函数。注意，这些函数的返回顺序是无法保证的，而是和网络有关。运行这段脚本，可看到如下输出：

```

fetching page
fetching api
page fetched
data returned from the api

```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour04/example06` 找到。以下步骤演示了异步代码。

1. 将程序清单 4.3 中的代码复制到系统中名为 `app.js` 的文件里。
2. 运行这一脚本：

```
node app.js
```

3. 可看到数据库查询模拟以及数据异步地从 API 获取。这里要理解的要点是，由于使用了异步风格，这些操作可以同时执行，当数据返回时，会触发回调。
-

通过在终端上运行这两个示例，我们可以清楚地了解这两种编码风格对运行的是什么以及什么时候运行的影响。Node.js 对如何在网络和 I/O 操作中处理并发（或者一次做超过一件事情）有自己的做法，所推崇的是异步方式。

如果读者是 Node.js 和 JavaScript 的新手，可能会混淆异步、同步、阻塞和非阻塞这几个术语。这些术语在 Node.js 社区和文档中大量使用，所以理解它们就很重要了。

同步和阻塞这两个术语可互换使用，指的是代码的执行会在函数返回之前停止，就如在前面的示例所看到的。如果某个操作阻塞，那么脚本就无法继续。对于最终用户而言，这意味着他们必须得等待。

异步和非阻塞这两个术语也可互换使用，指的是基于回调的、允许脚本并行执行操作的方法。脚本无需等待某个操作的结果才能继续前进，因为操作结果会在事件发生时由回调来处理。使用异步方法，操作无需一个接一个地发生。

4.5 事件循环

现在，读者可能会问，这些神奇的事情都是如何发生的？Node.js 使用 JavaScript 的事件循环来支持它所推崇的异步编程风格。这可能又是一个难以掌握的概念。基本上，事件循环使得系统可以将回调函数先保存起来，而后当事件在将来发生时再运行。这可以是数据库返回数据，也可以是 HTTP 请求返回数据。因为回调函数的执行被推迟到事件发生之后，于是就无需停止执行，控制流可以返回到 Node 运行时的环境，从而让其他事情发生。

事件循环不是 JavaScript 特有的，只是这个语言精于此道。JavaScript 围绕着事件循环设计，作为在有人与 Web 页面交互时对浏览器中所发生的不同事件进行响应的方式。这包括诸如单击鼠标或滚动页面的某个部分的事件。事件循环对于基于浏览器的交互而言是个良好的选择，因为要预测这些事件何时发生并不容易。Node.js 将这一方法应用到服务器端的编程上，尤其是在网络和 I/O 操作的上下文中。Node.js 经常被当成是一个网络编程框架，因为它的设计旨在处理网络中数据流的不确定性。促成这样的设计的是 JavaScript 的事件循环和对回调的使用，它们使得程序员可以编写对网络或 I/O 事件进行响应的异步代码。

就如读者在本章中所看到的阻塞和非阻塞示例那样，使用事件循环是另一种编程方式。有些开发人员称其为编写将里面翻到外面的程序，实际上关键思想是，将代码围绕着事件来构架而不是按照期待中的输入顺序来构架。由于事件循环以单一进程为基础，所以为了确保高性能需要遵循以下的一些规则。

- 函数必须快速返回。
- 函数不得阻塞。
- 长时间运行的操作必须移到另一个进程中。

在这样的上下文中，有些程序不适合于事件循环。如果程序或函数需要长时间运行才能完成处理，那么事件循环就不是个好选择。Node.js 所不适合的地方包括处理大量数据或者长时间运行计算等。Node.js 旨在在网络中推送数据并瞬间完成！

4.6 小结

我们在本章中学了很多！我们学习了回调以及回调在 JavaScript 中如何工作。而后我们看了 Node.js 使用回调的方法并且看了 Node.js 在从磁盘读取文件和从 Web 获取主页时使用回调的例子。我们探究了同步和异步编程的不同，介绍了阻塞和非阻塞代码之间的不同。最后，我们了解了事件循环，它让回调可以先注册而后在事件发生时运行。

本章任务繁重并且偏向理论，不过，在后续的章节中读者将看到，学习 Node.js 使用回调的方法和异步编程的大致哲学，对于 Node.js 这个技术来说是核心内容。

4.7 问与答

问：当使用回调时如果我想控制事情的顺序该如何？

答：可以在回调函数中嵌套回调函数，这样当第一个回调触发后，它就调用下一个函数。

这是个好示例：从数据库读取数据然后通过回调用另一个函数处理这些数据。

问：难道线程不是做这件事情最好的方式吗？

答：许多转向 Node.js 的程序员都会有使用线程来解决 Node.js 想解决的相同问题的经验。Node.js 对解决网络编程、使用回调、事件循环和单一进程有自己的做法。有些第三方模块已经让线程和纤程成为可能，但 Node 的核心哲学是在事件循环和单一进程的上下文中尝试并编程。

问：我可否在 Node.js 中同步编程？

答：许多转向 Node.js 的程序员都习惯于在其他语言中编程，并且发现切换到对异步编程的使用有点转不过弯来。如果你有这种感觉，给自己一些时间并探索使用事件循环和以异步风格编程能带给你的是什么。Node.js 认为，应该使用异步编程风格，所以如果你坚持使用同步风格的话，就不是按其所设计的使用它了。

问：难道这不是更复杂、需要写更多代码么？

答：的确，相对同步编码，使用异步编程风格会导致更多代码和更大的复杂性。不过，Node.js 想要解决的是个复杂问题呢！

4.8 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

4.8.1 问题

1. 在 Node.js 中经常使用的阻塞和非阻塞代码的同义词是什么？
2. 为什么异步代码是解决网络问题的好方法？
3. 在使用回调的时候可能会遇到哪些问题？

4.8.2 答案

1. 阻塞和非阻塞代码的同义词分别是同步和异步。
2. 网络经常不是开发人员所能控制的。我们可能要从不属于我们的远程服务器获取代码并且处理许多超出我们所能控制的元素。通过使用异步风格，可以让脚本在网络事件返回时响应。
3. 在使用回调时可能碰到与控制流有关的问题。在这里需要指定回调的执行顺序。还可能发现回调的嵌套深度会多达 4 到 5 层。本章的目标是理解回调，但随着读者对 Node.js 的经验增长，将会碰到深度嵌套的回调和控制流的问题。

4.9 练习

1. 回到本章中的 jQuery 示例并确认理解了回调的工作方式。接下来，在 Node.js 的主页

(<http://nodejs.org>) 上看看 Node.js Hello World Web 服务器。自己指出回调所使用的地方和方法。

2. 想一想学生上课中的同步和异步问题。想一想为什么参加物理课的学生和老师是同步事件。然后想想为什么远程学习是异步的。远程的学生可观看视频并使用在线资源在任何他们想学习的时间来学习，并且不同的个体可以异步学习。试着想一些编程以外的其他同步和异步示例。

3. 为了对 JavaScript 上下文内的事件循环有更深入的理解，请观看 Douglas Crockford 关于 Loopage 的课程。可在 <http://www.yuiblog.com/blog/2010/08/30/yui-theater-douglas-crockford-crockford-on-javascript-scene-6-loopage-52-min/> 上免费观看。



第2部分 使用 Node.js 的基本网站

第5章 HTTP

第6章 Express 介绍

第7章 深入 Express

第8章 数据的持久化



第 5 章

HTTP

在本章中你将学到：

- 理解 HTTP；
- 使用 Node.js 创建 HTTP 服务器；
- 使用 Node.js 创建 HTTP 客户。

5.1 什么是 HTTP

超文本传输协议（HTTP）是 Internet 上老牌的通信协议。从本质上说，它定义了服务器和客户端在通信的时候应该如何发送和接收数据。我们每天都在使用 HTTP，当浏览器载入 Web 页面时，浏览器是连接到正在浏览的网站服务器的客户端。

通过使用 HTTP 模块的低级应用程序编程接口（API），Node.js 既允许我们创建服务器，也允许我们创建客户端。

在本章中，我们将探究这一模块的工作原理以及使用 Node.js 编写 HTTP 服务器和客户端的方法。

5.2 使用 Node.js 的 HTTP 服务器

Node.js 精于 HTTP。它让开发人员得以使用短短几行代码就创建出服务器和客户端。理解 Node 的 HTTP 模块的基础，能让我们对 Node.js 所能提供什么有更深入的理解。

5.2.1 一个基础的服务器

在第 1 章中，我们学习了如何创建一个简单的 Node.js 服务器将“Hello World”输出到

Web 浏览器上（见程序清单 5.1）。

程序清单 5.1

```
var http = require('http');
http.createServer(function (req, res) {
  res.end('Hello world\n');
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

这个示例包含许多与 HTTP 模块有关的信息。以下是所发生的事情。

1. 从 Node.js 的核心请求 HTTP 模块并赋予一个变量，以便以后在脚本中使用。于是脚本就可以访问一些方法来通过 Node.js 使用 HTTP。
2. 使用 `http.createServer` 创建了新的 Web 服务器对象。
3. 脚本将一个匿名函数传递给 Web 服务器，告诉 Web 服务器对象每当其接收到请求时会发生的是什么。在本例中，当请求来到时，应当响应 ‘Hello World’ 字符串，然后关闭连接。
4. 在脚本的第 4 行上，定义了 Web 服务器的端口和主机。这意味着可以使用 `http://127.0.0.1:3000` 来访问服务器。
5. 脚本将服务器的访问位置信息记录到控制台上。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour05/example01` 找到。以下步骤演示一个基础的 HTTP 服务器。

1. 将程序清单 5.1 中的代码复制到系统中名为 `server.js` 的文件中。
2. 通过运行如下命令启动服务器：

```
node server.js
```

3. 打开浏览器并访问 `http://127.0.0.1:3000`。

5.2.2 加入头 (Header)

对于每个 HTTP 请求和响应，都会发送 HTTP 头。HTTP 头发送的是附加的信息，包括内容类型、服务器发送响应的日期以及 HTTP 状态码。

在简单的 Hello World 服务器中，Node.js 已经发送了一些基本信息：

```
HTTP/1.1 200 OK
Connection: keep-alive
```

这给任何从本服务器请求页面的客户展示了以下内容。

- 所用的 HTTP 版本是 1.1。
- 响应代码 200 表示成功响应。
- 连接是持久的，而且符合 HTTP 1.1 协议。持久连接 (Persistent Connection) 从 HTTP

1.1 开始具备，它让许多实时功能成为可能。

By the Way

注意：HTTP 头有许多信息！

从 HTTP 头可以收集许多信息。以下是来自 BBC 网站的 HTTP 头：

```
HTTP/1.1 200 OK
Date: Sat, 12 Nov 2011 14:27:37 GMT
Server: Apache
Set-Cookie: BBC
-UID=b47e9b3e289245d93524ac809080blaf9b1380f9b010a01c72c993ff1992dc4f0
curl%2f7%2e21%2e4%20%28universal%2daple%2ddarwin11%2e0%29%20libcurl%2f
7%2e21%2e4%20OpenSSL%2f0%2e9%2e8r%20zlib%2f1%2e2%2e5; expires=Wed, 11-
Nov-15 14:27:37 GMT;
path=/; domain=bbc.co.uk;
Vary: X-IP-Is-Advertise-Combined
Etag: "1321108048"
X-Lb-NoCache: true
Cache-Control: private, max-age=60
Age: 9
Content-Length: 136556
Content-Type: text/html
```

从头的信息中我们可以看到 Web 服务器是 Apache，所发送的内容是 HTML，以及发送的日期，等等。

在我们所创建的 Hello World 示例中，可以很容易地发送一个头，表明内容是纯文本（见程序清单 5.2）。

程序清单 5.2 给服务器增加头

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/plain'
  });
  res.end('Hello world\n');
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

如果下载了本书的代码示例，那么这段代码可在 `hour05/example02` 找到。`writeHead` 方法在服务器发送给客户的响应中加入响应代码和头。现在，当服务器收到请求时，服务器就告诉客户其内容是纯文本。

```
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: keep-alive
```

Watch Out!

警告：重启服务器！

在每次修改代码后都请确认重启服务器。如果不重启服务器，那么所做的修改就看不到了。可以在终端中使用 `Ctrl+C` 杀死服务器进程然后再次启动该进程来重启服务器。

5.2.3 检查响应头

使用 Node.js 做开发的时候检查页面的响应头经常有所帮助。有许多工具可帮助我们做

这件事。

Chrome 的 HTTP Headers 扩展 (Extension)

如果读者是 Chrome 用户, HTTP Headers 扩展工作得非常好。读者可以从这里安装该扩展: <http://bit.ly/MolcMd>。

TRY IT YOURSELF

按照如下步骤在 Chrome 中使用 HTTP Headers 扩展。

1. 给 Chrome 安装 HTTP Headers 扩展。
2. 确认 Node.js 服务器运行:
`node server.js`
3. 打开浏览器并访问 <http://127.0.0.1:3000/>。
4. 单击 HTTP Headers 扩展图标。
5. 读者将看到 Node.js 服务器的 HTTP 头 (见图 5.1)。

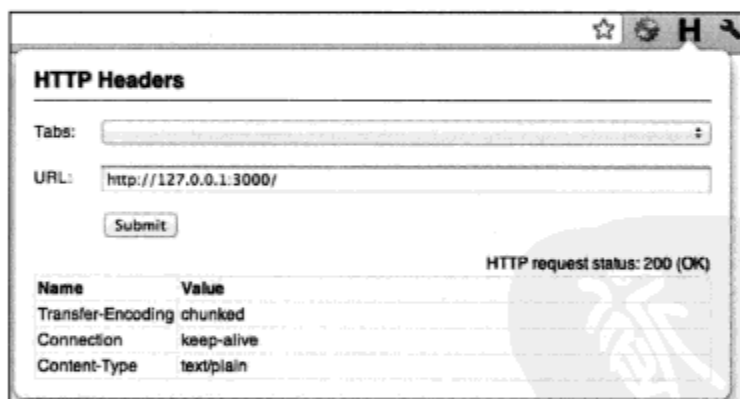


图 5.1

使用 HTTP Headers 扩展来检查响应头

Live HTTP Headers Firefox 外挂 (Add-On)

如果读者是 Firefox 用户, 那么 Firefox Web 浏览器的 Live HTTP Headers 外挂是个有用的工具。可在这里安装这个外挂: <http://bit.ly/LdJBSW>。

TRY IT YOURSELF

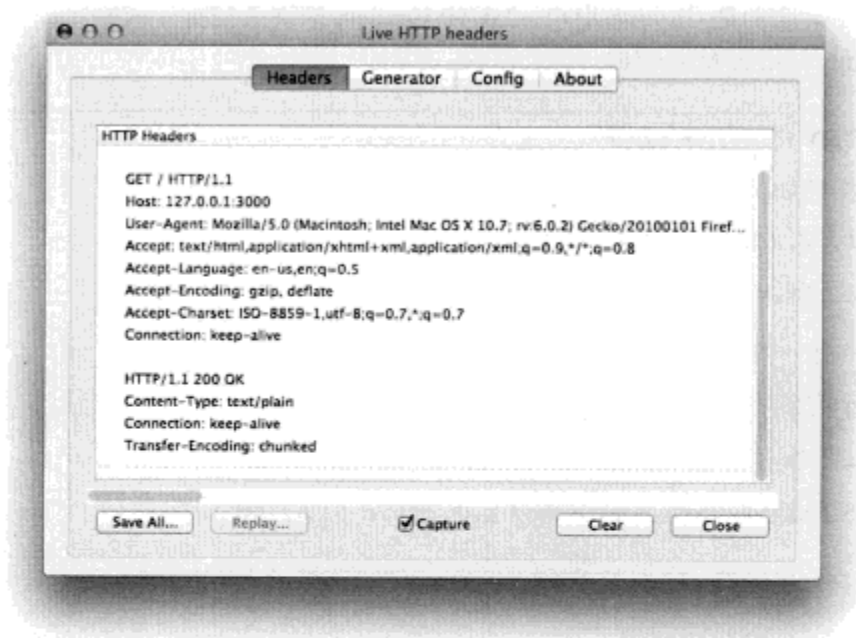
按照如下步骤使用 Live HTTP Headers Firefox 外挂。

1. 打开 Firefox, 在 <http://bit.ly/LdJBSW> 安装 Firefox 的 Live HTTP Headers 外挂。
2. 在 Firefox 中, 选择 Tools 菜单然后选择 Live HTTP Headers。可看到 Live Headers 窗口。
3. 确认 Node.js 服务器运行:
`node server.js`

4. 打开浏览器并访问 `http://127.0.0.1:3000`。
5. 在 Live Headers 窗口中可看到 Node.js 服务器的 HTTP 头（见图 5.2）。

图 5.2

使用 Live HTTP Headers 外挂检查响应头。



cURL

如果读者使用 UNIX 类型的系统并且习惯于使用终端，那么可以使用 cURL 来取得头的内容。cURL 随 OSX 一起发布并且可以在大多数 Linux 发行版中找到。

TRY IT YOURSELF

按照以下步骤使用 cURL 来检查响应头。

1. 打开终端并输入 `curl` 以确认系统中有 cURL。如果安装了 cURL，应当能看到：

```
curl: try 'curl --help' for more information
```

2. 确认 Node.js 服务器运行：

```
node server.js
```

3. 在终端中，运行如下命令：

```
curl -I 127.0.0.1:3000
```

4. 应当看到如下的响应（见图 5.3）：

```
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: keep-alive
```

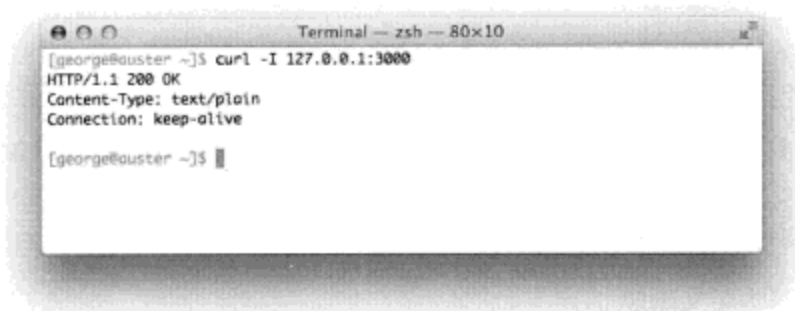


图 5.3

使用 cURL 检查响应头

5.2.4 Node.js 中的重定向

使用本章所学到的技艺，我们可以很容易地创建一个简单的服务器将访问者重定向到另外一个 Web 页面上。

重定向的准则如下所示。

- 给客户发送 301 响应代码，告诉客户：资源已经移到另一个位置。
- 发送一个位置头（Location Header）告诉客户重定向到哪里。在本例中，我们将访问者定向到 Strong Bad 的主页（见程序清单 5.3）。

程序清单 5.3 使用 Node.js 的重定向

```
var http = require('http');  
http.createServer(function (req, res) {  
  res.writeHead(301, {  
    'Location': 'http://www.homestarrunner.com/sbsite/'  
  });  
  res.end();  
}).listen(3000, "127.0.0.1");  
console.log('Server running at http://127.0.0.1:3000/');
```

这个示例再次使用了 `writeHead` 方法，但这一次发送的是 301 响应代码。另外也发送了位置头，告诉客户要重定向到哪里。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour05/example03` 找到。按照下列步骤创建 301 重定向。

1. 将程序清单 5.3 中的代码复制到系统中名为 `server.js` 的文件中。
2. 运行下列命令启动服务器：

```
node server.js
```

3. 打开浏览器并访问 `http://127.0.0.1:3000`。
4. 结果会被重定向到 Strong Bad 的网站。

5.2.5 响应不同的请求

到目前为止，我们用 Node.js 创建的是简单的 HTTP 服务器，返回单一的响应。如果要让服务器响应超过一种类型的请求该怎么做？在这种场景下，我们需要给应用程序加一些路由（route）。Node.js 通过使用 URL 模块让这一切工作直截了当。URL 模块使我们可以读取 URL、分析它然后对输出做一些事情。

By the Way

注意：路由定义响应

路由指的是应用程序要响应的请求。举个例子来说，如果想展示一个位于/about-us 的“关于我们”页面，就需要在应用程序中对这一请求设置一个路由。

TRY IT YOURSELF

以下是使用 URL 模块的方法。

1. 打开终端并启动一个 Node.js 进程：

```
node
```

2. 尝试下列代码来请求 URL 模块并将其赋予一个变量：

```
var url = require('url')
```

3. 输入下列用来在示例中模拟请求 URL 的代码：

```
var requestURL = 'http://example.com:1234/pathname?query=string#hash'
```

4. 现在，可以开始分析请求的 URL 并从中截取内容。要想获得主机名称，输入：

```
url.parse(requestURL).hostname
```

它应该返回“example.com”。

5. 取得端口号。输入下列代码：

```
url.parse(requestURL).port
```

它应该返回“1234”。

6. 取得路径名。输入下列代码：

```
url.parse(requestURL).pathname
```

它应该返回“/pathname”。

使用 URL 模块就有可能对 URL 的所有组成进行分析。完整的清单请查阅 Node.js 的文档。

使用对 URL 模块的新知识，可以修改早先创建的简单服务器，让它通过对所请求的 URL 进行分析来对不同的请求作出不同的响应（见程序清单 5.4）。

程序清单 5.4 给服务器加入路由

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

http.createServer(function (req, res) {
  var pathname = url.parse(req.url).pathname;

  if (pathname === '/') {
    res.writeHead(200, {
      'Content-Type': 'text/plain'
    });
    res.end('Home Page\n');
  } else if (pathname === '/about') {
    res.writeHead(200, {
      'Content-Type': 'text/plain'
    });
    res.end('About Us\n');
  } else if (pathname === '/redirect') {
    res.writeHead(301, {
      'Location': '/'
    });
    res.end();
  } else {
    res.writeHead(404, {
      'Content-Type': 'text/plain'
    });
    res.end('Page not found\n');
  }
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

在程序清单 5.4 中，服务器在第 5 行使用 URL 模块检查请求的路径名。在第 6 行使用 if 语句给所请求的路径名予以相关的响应。应用程序现在可以根据 URL 的路径名对多个请求响应了。如果路径名找不到，那么默认地响应 404 页面。

使用这一技艺，就可以创建对许多不同请求进行响应的服务器。但 Node.js 开发人员很快地意识到这会变得复杂、难以阅读与维护。因此，就有了许多的库来帮助我们对使用 Node.js 创建服务器过程中的一些复杂性做抽象。在第 6 章中我们会介绍 Express，不过无论如何，理解 Node.js 服务器在基础层面上的工作方法有益无害。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour05/example04` 找到。按照下列步骤响应多个请求。

1. 将程序清单 5.4 中的代码复制到系统中名为 `server.js` 的文件中。
2. 运行如下命令启动服务器：

```
node server.js
```

3. 打开浏览器并访问 `http://127.0.0.1:3000`。

4. 可看到“Home Page”。

5. 使用 Chrome 的 HTTP Headers 扩展、Firefox 的 Live HTTP Headers 外挂或者 cURL 来检查 HTTP 头。

6. 可看到:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: keep-alive
```

7. 回到浏览器, 导航到 `http://127.0.0.1:3000/unknown-page`。

8. 可看到“Page not found”。

9. 使用 Chrome 的 HTTP Headers 扩展、Firefox 的 Live HTTP Headers 外挂或者 cURL 来检查 HTTP 头。

10. 可看到:

```
HTTP/1.1 404 Not Found
Content-Type: text/plain
Connection: keep-alive
```

5.3 使用 Node.js 的 HTTP 客户端

可以使用 Node.js 创建 HTTP 服务器, 也可以用它来创建 HTTP 客户端。

By the Way

注意: HTML 客户端不总是浏览器

HTML 客户端可以是任何对服务器请求响应的东西。比如 Web 浏览器、搜索引擎机器人、电子邮件客户端以及 Web 刮取器 (Web Scraper) 都是 HTML 客户端。

我们会需要使用 HTML 客户端的场景有:

- 监控服务器的正常工作时间;
- 刮取不能通过 API 获取的 Web 内容;
- 创建将来自 Web 的两个或更多信息来源组合在一起的混搭 (mashup);
- 对诸如 Twitter 或 Flickr 这样的流行 Web Service 进行 API 调用。

要想使用 Node.js 创建 HTML 客户端, 需要如创建服务器时所做的那样请求 HTTP 模块。HTTP 模块具备方便的 `http.get` 方法来实现对服务器的 GET 请求。要想使用它, 需要指定一个包含了想要获取的页面的细节信息的 `option` 对象。这些细节包括主机、端口号以及路径。然后, 使用 `http.get` 方法完成请求。在程序清单 5.5 中, 我们创建一个客户端来获取一个页面的 HTTP 状态码并检查响应代码。如果响应代码是 200, 就可以认为网站正常工作。如果是任何其他值, 就意味着站点有问题。注意有些网站会将主页重定向到其他位置并以 302 作为响应, 所以读者如果选择使用其他 URL 的时候要小心这样的情况。

程序清单 5.5 以 Node.js 编写的 HTML 客户端

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

var options = {
  host: 'shapeshed.com',
  port: 80,
  path: '/'
};

http.get(options, function(res) {
  if (res.statusCode == 200) {
    console.log("The site is up!");
  }
  else {
    console.log("The site is down!");
  }
}).on('error', function(e) {
  console.log("There was an error: " + e.message);
});
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour05/example05` 找到。按照下列步骤创建 HTTP 客户端。

1. 将程序清单 5.5 中的代码复制到系统中名为 `client.js` 的文件中。
2. 运行如下命令启动服务器：

```
node server.js
```

3. 可看到 `The site is up!` 信息，表示站点返回 200 响应。如果看到任何其他信息，则站点可能有问题。

5.4 小结

在本章中，我们学习了 HTTP 以及如何使用 Node.js 与之交互。我们学习了如何创建简单的 Web 服务器以及如何将头发送回给客户。然后将简单的 Web 服务器扩展了一下，创建了重定向，然后让服务器可以对超过一种请求类型作出响应。最后，我们创建了一个 HTTP 客户端来检查 Web 服务器的状态。

本章展示了 Node.js 在底层使用 HTTP 的功能。在下一章中，我们要介绍 Express，这是一个 Node.js 的 Web 框架，它可让 HTTP 服务器的创建变得对用户更友好。

5.5 问与答

问：HTTP 看起来很复杂，必须得理解 HTTP 才能使用 Node.js 吗？

答：在 Node.js 中即使不完全理解 HTTP 也能应付得过去。本质上，HTTP 只是定义了服务器与客户端之间发送接收信息的交互方法，所以随着你创建越来越多的 HTTP 客户端或服务器，就会越

理解 HTTP。尤其如果创建 Web 应用程序的话，对 HTTP 的理解将给开发工作带来巨大帮助。

问：我听说过 Node.js 中的 Express 框架，我真的得用 HTTP 模块来编写服务器吗？

答：如果你刚刚开始，那么 Express 可降低使用 Node.js 创建 Web 服务器的门槛。它可处理许多常见的场景，比如路由和模板，创建大多数 Web 应用程序时推荐使用它。不过，理解 HTTP 模块的工作原理是有用的。会有这样的情况：你需要创建一个小的、不需要像 Express 这样的框架来实现的 Web 服务器。理解引擎盖下的秘密也总是个好主意。

问：Apache 或 Nginx 如何与 Node.js 配在一起？

答：Apache 和 Nginx 是流行的、成熟的 Web 服务器。将 Web 流量从 Apache 或者 Nginx 代理到 Node.js 服务器是完全可能的。我们同样也可以直接从 Node.js 服务器来服务流量。

问：我已经创建好了我想要部署的服务器！我该如何部署呢？

答：别着急！我们将在第 11 章学习部署。

5.6 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

5.6.1 问题

1. 200 HTTP 响应是什么意思？
2. 在 HTTP 头中可以发送什么信息？
3. Connection: Keep-Alive 头为什么是重要的？

5.6.2 答案

1. 200 响应代码意味着请求成功。可能的响应代码的详细规范可在 <http://bit.ly/KD3qC8> 找到。
2. 我们可以在 HTTP 头中发送大量信息，包括可接受的字符集、特权授权证书、日期以及用户代理等。完整的清单见 <http://bit.ly/KWge56>。
3. Connection: Keep-Alive 头让客户端和服务器可以一直打开持久连接。这会启用客户端和服务器之间的实时通信，于是带来各种各样的可能性。

5.7 练习

1. 使用程序清单 5.1，将响应代码更改为 501。使用本章内讨论过的工具之一来确认 HTTP 响应码是 501。
2. 使用程序清单 5.4，扩展 Web 服务器让其包含更多路由。试验一下对不同的路由发送不同的响应代码和内容。
3. 使用程序清单 5.5，更改 options 对象以检查不同网站或者服务器的状态。

第 6 章

Express 介绍

在本章中你将学到：

- Express 是什么以及如何使用它；
- 用 Express 创建基础站点；
- 在 Express 中使用 Jade 作为模板引擎。

6.1 什么是 Express

Express 是 Node.js 的一个 Web 框架。Web 应用程序有共同的模式，所以使用框架来构建经常事半功倍。不难发现我们可以因此有更快的开发速度，而且是在稳定、已测试的代码之上编写应用程序。

读者可能还熟悉其他的一些 Web 框架，比如：

- Ruby on Rails (Ruby)；
- Sinatra (Ruby)；
- Django (Python)；
- Zend (PHP)；
- CodeIgniter (PHP)。

6.2 为什么使用 Express

Express 是个轻量级的框架，这就意味着它不做太多假定，但却足以避免用户重新发明轮子。

一些使用 Express 能做的事情有：

- 基于 JSON 的 API；
- 单页面 Web 应用程序；
- 实时 Web 应用程序。

一些使用诸如 Express 这样的框架的理由如下。

- 使用框架可以减少创建应用程序所需的时间。
- 诸如路由和视图层（view layer）这样的常用模式在 Express 这样的框架中已经处理，也就是说我们无需为此编写代码。
- 像 Express 这样的框架大家都在积极使用、维护和测试。所以，其代码的稳定性可期。

但是像 Express 这样的框架并不适合所有事情。如果创建命令行脚本的话，显然就不会去使用像 Express 这样的东西了。

By the Way

注意：Express 受 Sinatra 启发

Sinatra 是 Ruby 上的一个流行的轻量级 Web 框架。Express 支持的模板引擎、路由以及传输数据给视图和 Sinatra 在 Ruby 中所做的很像。如果读者用过 Sinatra，就可对 Express 驾轻就熟。如果没有使用过 Sinatra，大多数开发人员会喜欢其设计的简单性。

6.3 安装 Express

可通过 npm 安装 Express：

```
npm install -g express
```

Watch Out!

警告：要确认使用-g 标记

请注意上面的示例中的-g 标记。它表示 Express 将是全局安装的，可以在系统的任何地方使用其命令。如果忘记-g 标记，那么如果不提供可执行文件的完整路径就无法运行生成器。注意根据操作系统的不同，可能需要使用 sudo 来执行这一命令。

6.4 创建一个基础的 Express 站点

安装了 Express 之后，就可以搭起一个基础的站点并运行了。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour06/example01 找到。按照如下步骤来安装一个基础的 Express 站点。

1. 打开终端并运行如下命令生成一个 Express 骨架站点（见图 6.1）：

```
express express_example
```

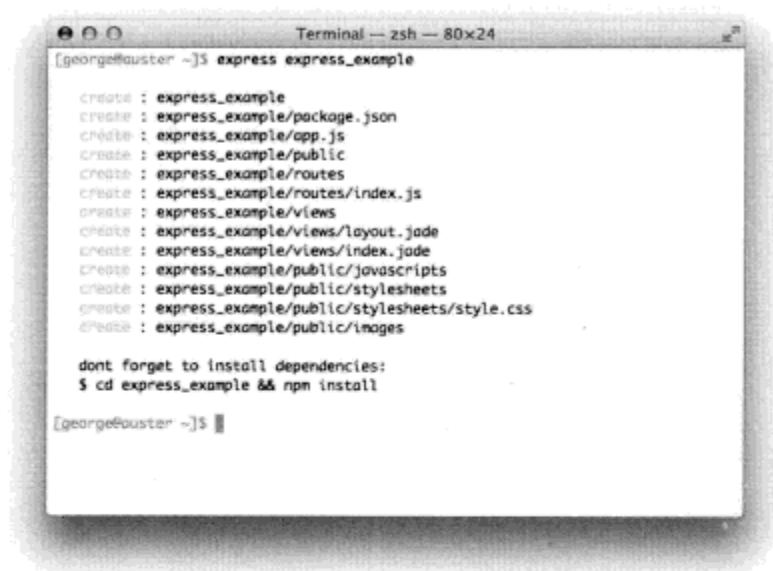


图 6.1

使用 Express 安装
一个基础站点

2. Express 有礼貌地提醒用户安装运行 Express 所需的依赖模块。所以，要确认安装了依赖模块：

```
cd express_example && npm install
```

3. 使用下列命令运行该应用程序：

```
node app.js
```

4. 打开你自己喜爱的 Web 浏览器，浏览 <http://127.0.0.1:3000>，可看到由 Express 服务的一个基础网站（见图 6.2）。

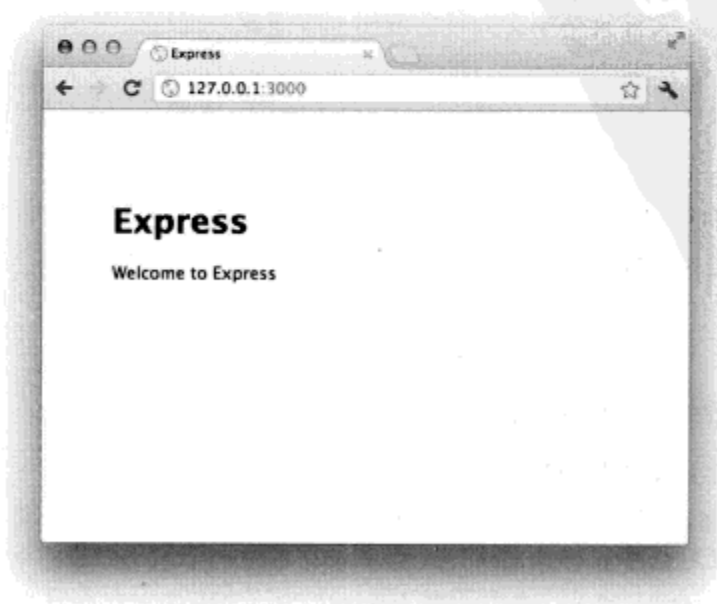


图 6.2

使用 Express 安装
一个基础站点

6.5 探索 Express

如果查看刚刚创建的示例 Express 站点文件夹中的内容，会看到如下结构。

- app.js。
- node_modules。
- package.json。
- public。
- routes。
- views。

6.5.1 app.js

app.js 是用来启动应用程序的应用程序文件夹。它包含应用程序的配置信息。

6.5.2 node_modules

node_modules 用来保存在 package.json 中定义并且已经安装的 Node 模块。

6.5.3 package.json

package.json 提供应用程序的信息，包括运行应用程序所需安装的依赖模块。

6.5.4 public

这是将应用程序提供给 Web 进行服务的公共文件夹。在这个文件夹中我们会看到样式单、javascript 和图片。在这个文件夹中不会有任何应用程序逻辑，这是确保 Web 应用程序的安全性的常用方法。

6.5.5 routes

简单来说，路由定义了应用程序应该响应的页面。比如，如果想在应用程序中有“关于”页面，就需要设置一个“about”路由。路由文件夹保存了这些定义。路由将在第7章深入讲解。

6.5.6 views

视图文件夹定义应用程序的布局（layout）。

**By the
Way**

注意：文件夹结构是可选的

Express 生成器为 Express 项目按建议的布局创建文件夹。这只是个建议，如果应用程序有特殊需求，或者用户有不同的个人选择，可以按自己的需要构架 Express 项目。如果刚对 Express 入门，建议从生成器生成结构。

6.6 介绍 Jade

看看示例项目中的 `views` 文件夹，会看到许多以 `.jade` 为扩展名的文件。Express 利用模板引擎将视图编译成 HTML。默认情况下，Express 使用 Jade 作为模板引擎。

注意：模板引擎生成 HTML

大多数 Web 框架使用模板引擎生成 HTML，而且通常在视图文件夹中使用。它们使开发人员可以从应用程序将数据输出到 HTML 中。典型的功能包括变量和循环。模板引擎也称为模板处理器（template processor）或者过滤器（filter）。Smarty（PHP）和 ERB（Ruby）是两个流行的模板引擎。

**By the
Way**

Jade 是个基于缩进的模板引擎。要理解这句话的意思，我们来比较 HTML 及其在 Jade 中的表示（见程序清单 6.1）。

程序清单 6.1 HTML 和 Jade 比较

```
<div class="wrapper">
  <h1>My holiday in Prague</h1>
  <p>I had a great holiday in Prague where I met some great people.</p>
  
</div>

.wrapper
  h1 My holiday in Prague
  p I had a great holiday in Prague where I met some great people
  img(src='images/photo.jpg', alt='Me on holiday')
```

要注意如下几点。

- Jade 要比 HTML 简洁得多。
- Jade 使用缩进来定义 HTML 文档的层次结构。
- 在 Jade 中无需使用标记，编译模板的时候会自动加入“<>”字符。
- 在 Jade 中无需关闭 HTML 标记，在 Jade 生成 HTML 的时候会为我们关闭标记。

警告：缩进很重要

代码的缩进方式在 Jade 中是重要的，因为它定义了文档的层次结构。有时候在使用 Jade 做开发时会在视图上看到错误生成。这很可能是因为缩进不正确，所以要仔细检查！

**Watch
Out!**

在页面生成之后，Jade 将模板编译成 HTML。在程序清单 6.1 中，输出与普通的 HTML 完全一样。那么为什么还要使用模板引擎呢？答案是：它使得应用程序可以动态输出数据到 HTML 中。使用模板引擎的例子如下。

- 显示一组保存在数据库中的博客帖子。
- 创建单一的模板用于显示许多不同的博客帖子。

- 按变量的值更改页面的<title></title>元素。
- 创建可跨模板重用的页眉和页脚内容。

By the Way

注意：Jade 受的是 Haml 的启发

Jade 受 Haml（HTML 抽象标记层）的启发甚多。Haml 是一个在 Ruby 社区中流行的模板引擎。

6.6.1 使用 Jade 定义页面结构

Jade 通过缩进定义页面结构。如果一行在上一行下面缩进，就认为它是上一行的子行。如果读者刚接触模板语言，会需要一些时间来适应。不过，看了示例就清楚了。

```
html
```

编译后是：

```
<html></html>
```

这里可以使用任何 HTML 标记（body、section、p 等等）。

要给标记使用 id，可加上一个“#”号后跟 id 名称。注意不允许有空格。

```
section#wrapper
```

编译后是：

```
<section id="wrapper"></section>
```

可以添加一个类，方法是加上一个小数点后跟类的名称。

```
p.highlight
```

编译后是：

```
<p class="highlight"></p>
```

如果既需要类也需要 id，那么 Jade 支持串接。

```
section#wrapper.class-name
```

编译后是：

```
<section id="wrapper" class="class-name"></section>
```

Jade 也支持在一个标记上有多个类。

```
p.first.section.third.fourth
```

编译后是：

```
<p class="first second third fourth">
```

为了创建 HTML 结构，要使用缩进。

```
p
  span
```

编译后是:

```
<p><span></span></p>
```

要在标记中加入文本, 只需在标记定义后加入即可。

```
h1 Very important heading
```

编译后是:

```
<h1>Very important heading</h1>
```

Jade 也支持使用管道描述符 (pipe delineator, 也就是 “|” 字符) 组织大的文本主体。

```
p
  | Text can be over
  | many lines
  | after a pipe symbol
```

编译后是:

```
<p>Text can be over many lines after a pipe symbol</p>
```

TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour06/example02` 找到。按照下列步骤在 Jade 中创建一个页面结构。

1. 打开终端并运行如下命令生成一个 Express 骨架站点:

```
express jade_structure
```

2. 进入所创建的目录能够安装所需的依赖模块:

```
cd jade_structure && npm install
```

3. 启动应用程序, 更换到 `jade_structure` 文件夹, 然后运行

```
node app.js
```

4. 打开你所喜爱的 Web 浏览器并浏览 `http://127.0.0.1:3000`, 可看到由 Express 提供服务的一个基础网站。

5. 编辑 `views/index.jade` 中的 `index.jade` 文件, 可看到该页面的一个骨架结构:

```
h1= title
p Welcome to #{title}
```

6. 在页面中加入如下代码:

```
section#wrapper
  h2 Basic Structure
  section
    p
      span This is a span within a p within a div!
```

7. 重新载入页面并通过查看源代码来检查页面上的 HTML。虽然 HTML 会被压缩，但也应该能看到如下 HTML：

```
<section id="wrapper">
  <section>
    <p>
      <span>This is a span within a p within a div!</span>
    </p>
  </section>
</section>
```

By the Way

注意：section 标记是 HTML5 新有的标记

HTML5 是超文本标记语言 (HTML) 的第 5 个修订版。在 HTML5 中有许多新元素，包括 article、header、nav 和 section。它们意在增强 Web 页面的语义含义，这对人类和技术都是好事。如果读者对学习 HTML5 中的新元素有兴趣，可考虑读一读《Introducing HTML5》一书的第 2 版，作者是 Remy Sharp 和 Bruce Lawson (New Riders 出版，0321784421)。

6.6.2 使用 Jade 输出数据

使用 Jade 构建页面结构没有什么问题，但模板语言的真正能力在于操纵数据并输出数据到 HTML 中。

Jade 使用两个特殊字符来决定应该如何翻译代码。第一个字符是减号 (-)，用于告诉随后的代码应当被执行。

第二个字符是等号 (=)。它告诉解释器要对代码进行演算、转义，然后输出。

一开始会有点混淆，但看看示例就清楚了。

变量

在这个示例中，代码要被执行并且不会返回任何输出。Jade 仅仅将变量 foo 设置为 bar：

```
- var foo = bar;
```

变量设置之后可以在以后使用：

```
p I want to learn to use variables. Foo is #{foo}!
```

“#{变量}”这个特殊语法告诉 Jade 用将“变量”替换为字符串值。上述代码编译后是：

```
<p>I want to learn how to use variables. Foo is bar!</p>
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour06/example03 找到。按照下列步骤在 Jade 中使用变量。

1. 打开终端并运行如下命令生成一个 Express 骨架站点：

```
express jade_variables
```

2. 进入所创建的目录并安装所需的依赖模块:

```
cd jade_variables && npm install
```

3. 启动应用程序, 更换到 jade_variables 文件夹, 然后运行:

```
node app.js
```

4. 打开你所喜爱的 Web 浏览器并浏览 <http://127.0.0.1:3000>, 可看到由 Express 提供服务的一个基础网站。

5. 编辑 views/index.jade 中的 index.jade 文件, 加入如下代码:

```
- var name = "Your Name"
h1 Hello #{name}!
```

6. 在浏览器中重新载入页面, 通过查看源代码来检查页面上的 HTML, 可看到下列 HTML:

```
<h1>Hello Your Name</h1>
```

循环

循环用来对数组和对象进行迭代。只要不是创建基础的小站点, 肯定得大量使用循环。循环通常称为迭代, 意思是对数组或对象进行迭代并不断做相同的事情。

可能因为这是非常常见的模式, 所以 Jade 让减号的使用是可选的。应该承认的是, 将减号用在变量赋值上却不用在循环上, 这会造成混淆。所以, 在下列这些示例中, 为了清楚起见, 在每个循环前面都包括了减号, 但这个减号是可选的。

```
- users = ['Sally', 'Joseph ', 'Michael', 'Sanjay']
- each user in users
  p= user
```

编译后是:

```
<p>Sally</p>
<p>Joseph</p>
<p>Michael</p>
<p>Sanjay</p>
```

如果选择使用 for 关键字, 也可以这样写:

```
- for user in users
  p= user
```

也可以对对象进行迭代:

```
- obj = { first_name: 'George', surname: 'Ornbo' }
- each val, key in obj
  li #{key}: #{val}
```

编译后是:

```
<li>first_name: George</li>
<li>surname: Ornbo</li>
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour06/example04` 找到。按照下列步骤在 Jade 中使用循环。

1. 打开终端并运行如下命令生成一个 Express 骨架站点：

```
express jade_loops
```

2. 进入所创建的目录并安装所需的依赖模块：

```
cd jade_loops && npm install
```

3. 启动应用程序，更换到 `jade_loops` 文件夹，然后运行：

```
node app.js
```

4. 打开你所喜爱的 Web 浏览器并浏览 `http://127.0.0.1:3000`，可看到由 Express 提供服务的一个基础网站。

5. 编辑 `views/index.jade` 中的 `index.jade` 文件，加入如下代码：

```
- beatles = ['John', 'Paul', 'Ringo', 'George']
ul
  each beatle in beatles
    li #{beatle}
```

6. 在浏览器中重新载入页面，通过查看源代码来检查页面上的 HTML，可看到下列 HTML：

```
<ul>
  <li>John</li>
  <li>Paul</li>
  <li>Ringo</li>
  <li>George</li>
</ul>
```

条件

如果读者曾经接触过任何编程语言，就不会对条件感到陌生。用平实的语言说，我们可以将条件流描述为：如果某件事情是真的，做一些事情；否则，做其他事情。在 Jade 中，条件的形式如下：

```
- awake = false
- if (awake)
  p You are awake! Make coffee!
- else
  p You are sleeping
```

请再次注意，`if` 和 `else` 关键字之前的减号 (-) 是可选的。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour06/example05` 找到。按照下列步骤在 Jade 中使用条件。

1. 打开终端并运行如下命令生成一个 Express 骨架站点:

```
express jade_conditions
```

2. 进入所创建的目录并安装所需的依赖模块:

```
cd jade_conditions && npm install
```

3. 启动应用程序, 更换到 jade_conditions 文件夹, 然后运行:

```
node app.js
```

4. 打开你所喜爱的 Web 浏览器并浏览 <http://127.0.0.1:3000>, 可看到由 Express 提供服务的一个基础网站。

5. 编辑 views/index.jade 中的 index.jade 文件, 加入如下代码:

```
- raining = true
- if (raining)
  p It is raining. Take an umbrella!
- else
  p No rain. Take the bike!
```

6. 在浏览器中重新载入页面, 通过查看源代码来检查页面上的 HTML, 可看到下列 HTML:

```
<p>It is raining. Take an umbrella!</p>
```

7. 将 views/index.jade 中 raining 的值改为 false:

```
- raining = false
```

8. 在浏览器中重新载入页面。通过查看源代码来检查页面上的 HTML。可看到下列 HTML:

```
<p>No rain. Take the bike!</p>
```

内联 (inline) JavaScript

在 Jade 模板中可以执行内联 JavaScript。要想这样做, 可声明一个脚本块然后在其中加入 JavaScript 代码:

```
script
  alert('You can execute inline JavaScript through Jade')
```

TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 hour06/example06 找到。按照下列步骤在 Jade 中执行内联 JavaScript。

1. 打开终端并运行如下命令生成一个 Express 骨架站点:

```
express jade_inline_javascript
```

2. 进入所创建的目录并安装所需的依赖模块:

```
cd jade_inline_javascript && npm install
```

3. 启动应用程序，更换到 `jade_inline_javascript` 文件夹，然后运行：

```
node app.js
```

4. 打开你所喜爱的 Web 浏览器并浏览 `http://127.0.0.1:3000`，可看到由 Express 提供服务的一个基础网站。

5. 编辑 `views/index.jade` 中的 `index.jade` 文件，加入如下代码：

```
script
  alert('Inline JavaScript in Jade')
```

6. 在浏览器中重新载入页面，可看到一个 JavaScript 提示框。

包含 (include)

大多数网站都会有在站点的每个页面上都会出现的页面组成部分，比如：

- 页眉；
- 页脚；
- 边栏。

Did you know?

提示：包含让网站的维护更简单

包含将网站的公共部分移到了单一的文件中。这就意味着如果顾客要求在页眉中多加一个项目，开发人员只需更改一个文件即可。

Jade 通过使用 `include` 关键字后跟想要包含的模板来支持包含功能。

```
html
  body
    include includes/header
```

上述代码从 `views/includes/header.jade` 文件中包含代码。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour06/example07` 找到。按照下列步骤在 Jade 中使用包含。

1. 打开终端并运行如下命令生成一个 Express 骨架站点：

```
express jade_includes
```

2. 进入所创建的目录并安装所需的依赖模块：

```
cd jade_includes && npm install
```

3. 启动应用程序，更换到 `jade_includes` 文件夹，然后运行：

```
node app.js
```

4. 打开你所喜爱的 Web 浏览器并浏览 `http://127.0.0.1:3000`，可看到由 Express 提供服务的一个基础网站。

5. 创建新的文件夹：`views/includes`。
6. 在 `views/includes` 中添加一个名为 `footer.jade` 的文件。
7. 在 `footer.jade` 文件中添加一些内容：

```
p This is my footer. Get me. I have a footer on my website.
```

8. 将页脚（footer）文件包含到 `views/index.jade` 中：

```
h1 Jade Includes Example
include includes/footer
```

9. 在浏览器中重新载入页面，应该可以看到页面中包含进了页脚内容。

Mixin

Jade 的 `Mixin` 是许多其他模板引擎所不具备的功能。如果发现自己必须一次又一次地重复某些代码块，那么使用 `mixin` 是让代码保持可维护并且整洁的良好方法。将 `mixin` 想象成代码的包含。在循环中输出数据是可以使用 `mixin` 的一个示例。要想定义一个 `mixin`，请使用 `mixin` 关键字：

```
mixin users(users)
  ul
    each user in users
      li= user
```

一旦定义了 `mixin`，就可以使用它并且在模板中重用：

```
- users = ['Krist', 'Kurt', 'Dave']
mixin users(users)
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour06/example08` 找到。按照下列步骤在 Jade 中使用包含。

1. 打开终端并运行如下命令生成一个 Express 骨架站点：

```
express_jade_mixins
```

2. 启动应用程序，更换到 `jade_mixins` 文件夹，然后运行：

```
node app.js
```

3. 打开你所喜爱的 Web 浏览器并浏览 `http://127.0.0.1:3000`，可看到由 Express 提供服务的一个基础网站。

4. 编辑位于 `views/index.jade` 中的 `index.jade` 文件并添加如下代码：

```
mixin users(users)
  ul
    each user in users
      li= user
- users = ['Tanya', 'Jose', 'Kim']
mixin users(users)
- more_users = ['Mark', 'Elena', 'Dave', 'Pete', 'Keiron']
mixin users(more_users)
```

5. 在浏览器中重新载入页面，可看到两个用户列表，它们由同一个 `mixin` 生成。

6.7 小结

在本章，我们学习了 Express，这是一个 Node.js 的 Web 框架。我们学习了选择使用 Web 框架的时机以及 Web 框架能给我们提供什么。

我们学习了 Jade，这是一个 Node.js 的模板引擎，还学习了如何使用它输出变量、循环、条件语句、内联 JavaScript、包含以及 `mixin`。

在本章之后，读者可以使用 Node.js 和 Express 创建基础站点并且开始在应用程序中显示数据了。

6.8 问与答

问：我是应该总在 Node.js 中使用 Express 还是仅仅有些时候使用？

答：Express 是 Node.js 的一个 Web 框架，所以会有它不适合的时候。如果读者的需求是处理视图并在 Web 浏览器中显示数据，那么 Express 是适合的。如果要使用 Node.js 构建一个命令行脚本或者一个微小的 Web 服务，那么或许 Express 并不是正确选择。无论什么情况，都要选择最适合于工作的工具。

问：我对基于缩进的模板引擎不熟悉，还有其他选择吗？

答：虽然 Express 使用 Jade 作为默认的模板引擎，但它并不知道所使用的模板引擎是什么。Express 中另一个流行的选择是 EJS（嵌入式 JavaScript 模板）模板引擎。它不是基于缩进的，与 Ruby 中的 ERB 的工作方式接近。读者也可以在 Jade 中使用 jQuery。新的模板引擎一直都在出现，有许多都支持 Express。请在 Node.js 的 wiki 上查看一份综合清单：<https://github.com/joyent/node/wiki/modules#wiki-templating>。

问：其他模板的性能如何？

答：Jeremy Ashkenas 创建了一个测试，用来展示不同 JavaScript 模板语言的性能。如果读者对模板语言的性能感兴趣，他的帖子给出了基准并做出了性能比较（<http://bit.ly/KKa3nw>）。

问：使用诸如 Jade 这样的模板引擎有何优缺点？

答：使用模板引擎让我们可以更快地进行开发，并且具备内建的代码检查能力。通过使用缩进，Jade 很容易阅读，从而也容易维护。至于缺点，Jade 通过对纯 HTML 的抽象，多增加了一层复杂性，对于简单的项目而言会有些过头。

6.9 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

6.9.1 问题

1. 为什么我们会使用 Express 这样的框架，而不是编写自己的代码？
2. 在 Express 项目中，我们在哪里放置依赖模块？
3. Express 是否有固定的文件夹结构？

6.9.2 答案

1. 对于许多 Web 应用程序而言，会有诸如模板和路由这样的公共模式。虽然可以自己编写代码解决这些问题，但许多开发人员会通过使用诸如 Express 这样的框架来避免这样的工作，并且对之有所贡献。
2. 我们在 package.json 文件中放置 Express 项目的依赖模块，这和任何其他 Node.js 项目一样。
3. 不。虽然 Express 生成器会为我们产生一个文件夹结构，但无需遵循建议的结构。

6.10 练习

1. 使用命令行生成器创建一个 Express 站点。探索每个文件夹并理解其作用。请参阅本章中关于文件夹结构的注解。
2. 在终端中运行 `express—help`。注意在生成新的 Express 站点时如何通过传递参数来改变生成器的工作方式。请尝试在生成 Express 站点时切换所用的模板引擎。
3. 生成一个新的 Express 站点并通过对变量赋值然后输出来实践你对使用 Jade 的理解。创建一个数据数组然后使用循环来显示数据。最后，创建一个页眉包含文件并在模板中使用它。

第7章

深入 Express

在本章中你将学到：

- 在 Express 中路由如何工作；
- 在 Express 中添加路由；
- 在路由中使用参数；
- 将数据传递给视图层；
- 在视图层显示数据。

7.1 Web 应用程序中的路由

路由描述应用程序是否以及如何对特定的超文本传输协议（HTTP）请求进行应答。当用户在与应用程序或者网站交互的时候，浏览器会生成这些请求。

路由只是个用来定义应用程序中 HTTP 请求的最终点的术语。所以如果希望应用程序能做点什么，那么就必须为它设置路由！

7.2 在 Express 中路由如何工作

Express 使用 HTTP 动词（verb）来定义路由。HTTP 动词描述对服务器的请求的类型。最为可能用到的常用动词如下。

- GET——从服务器获取数据。
- POST——将数据发送给服务器。

其他的 HTTP 动词包括了 PUT、DELETE、HEAD、OPTIONS 和 TRACE。

在生成一个基础的 Express 站点时，生成器认为用户会给应用程序定义路由。为了理解这一点，我们来生成一个基础的 Express 应用程序并检查可用的路由。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour07/example01` 找到。要想生成一个基础的 Express 应用程序并检查可用的路由，请按如下步骤进行。

1. 创建一个基础 Express 站点：

```
express express_routing
cd express_routing
npm install
```

2. 现在启动服务器：

```
node app.js
```

3. 打开你喜爱的 Web 浏览器并浏览 `http://127.0.0.1:3000`，可看到由 Express 提供服务的
一个基础网站。

4. 将浏览器地址更改为 `http://127.0.0.1:3000/about/`。

5. 返回的将是页面未找到的 404 头，会看到：

```
Cannot GET / about
```

在这个示例中，`/about` 没有路由可用，于是返回的是页面未找到（404）响应。

注意：你一直都在做 GET 和 POST 请求！

在加载 Web 页面的时候，浏览器使用 GET 请求来获取 HTML、CSS、JavaScript 和图片文件。在提交表单的时候，浏览器很可能在做 POST 请求。HTTP 动词实际上颇为简单，而如果你正在浏览 Web，那么你会一直都在做 GET 和 POST 请求！

**By the
Way**

7.3 添加 GET 路由

在刚刚创建的简单的 Express 应用程序示例中，对于 `/about` 的 GET 请求没有路由。于是，Express 服务的是“404 页面未找到响应”。那么我们要怎么给 GET `/about` 添加路由呢？

为了指定一个 GET 请求，要使用 GET HTTP 动词。

```
app.get('/about', function(req, res){
  res.send('Hello from the about route!');
});
```

从技术上说，这段代码调用一个匿名函数，这个函数将请求和响应作为参数。Express 而后使用 `res.send` 方法返回响应。`res.send` 是 Express 提供的用于发送响应的方法。在这个示例中，用一串文本作为响应来返回。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour07/example02` 找到。要想添加一个

GET 路由，请按如下步骤进行。

1. 返回到刚才创建的基础 Express 站点。
2. 编辑主 app.js 文件并在 “app.get('/', routes.index)” 这一行之后为对/about 的 GET 请求加入定义：

```
app.get('/about', function(req, res){
  res.send('Hello from the about route!');
});
```

3. 现在启动服务器：

```
node app.js
```

4. 打开浏览器访问 <http://127.0.0.1:3000/>。
5. 可看到一个基础的 Express 站点。
6. 将浏览器的值改为 <http://127.0.0.1:3000/about/>。
7. 这次不再是 404 页面了，应该看到的是：

```
Hello from the about route!
```

Did you know?

提示：Express 为我们处理了一些路由

由于有些路由被认为是所有应用程序都需要的，所以 Express 提供了一些中间件（Middleware），为任何加入到/public 文件夹中的东西提供服务。如果使用站点生成器，这就会被包括其中。想象一下，如果每个要显示的图片都需要加入一个路由，该是多么无聊的工作！所以，我们可以安全地认为，只要把东西放入 public 文件夹，就可以得到服务，并且无需路由对此声明。

7.4 添加 POST 路由

我们学习了如何在 Express 中创建 GET 请求，但要是想允许用户提交数据给应用程序的话又要如何处理？

过程是一样的，但要使用 POST 动词而不是 GET 动词。

```
app.post('/', function(req, res){
  res.send(req.body);
});
```

在这个示例中，应用程序接收 POST 请求，送到 index 页面，并且将数据输出到浏览器中。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour07/example03` 找到。要想添加 POST 请求，请按如下步骤进行。

1. 创建一个基础 Express 站点:

```
express express_post
cd express_post
npm install
```

2. 打开 index.jade, 加入下列代码, 以便在 index 视图中添加一个表单:

```
h2 Form example
form(method='post', action='/')
fieldset
  legend Add a user
  p
    label First name
    input(name='user[first_name]')
  p
    label Last name
    input(name='user[surname]')
  p
    input(type='submit', value='Save')
```

3. 在 app.js 中添加一个接收 POST 请求的路由。在本示例中, 它输出寄送 (post) 给浏览器的数据。

```
app.post('/', function(req, res){
  res.send(req.body);
});
```

4. 现在启动服务器:

```
node app.js
```

5. 打开浏览器访问 <http://127.0.0.1:3000/>。

6. 在表单中填入你的姓名并单击 Submit。

7. 可看到寄送的数据中有自己的名字。

当然, 这个示例非常基础。在实际中, 会对数据做一些验证并保存到数据库中。不用担心, 我们很快就要学习如何做这件事!

7.5 在路由中使用参数

在 Web 应用程序中, 通过路由来重用模板并根据传递进来的参数改变内容, 这是个常用模式。读者可能对诸如这样的 URL 感觉熟悉:

```
/users/12
/projects/2345
```

在这里, 应用程序使用整数 (或者数) 来设置要显示的用户或项目。在第一个示例中, id 为 12 的用户将会被显示。在第二个示例中, id 为 2345 的项目将会被显示。

这是个有用的模式, 有如下的许多理由。

➤ 应用程序可以使用单一模板来显示用户记录。

- 应用程序为每个用户创建一个独一无二的 URL 或链接。
- 使用这种方法更容易加入创建、更新和删除方法。

幸运的是, Express 完全支持这一方法。要声明一个捕获参数的路由, 看起来是这样的:

```
app.get('/users/:id', function(req, res){
  res.send('show content for user id' + req.params.id);
});
```

在请求中, 所有正斜杆后面、跟随在 user 之后的东西, 都会被捕获并且通过 req 对象在应用程序中使用。所以如果对 /user/24 做一个 GET 请求, Express 会取出这个数字然后让应用程序可以在 req.params.id 中使用它。在路由中指定的符号成为了 params 键值, 所以如果要使用 '/users/:name', 那么可以通过 req.params.name 来访问。

这时, Web 应用程序通常使用这个 id 为用户获取数据然后返回数据。

TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 hour07/example04 找到。按照下列步骤在路由中使用参数。

1. 创建一个基础 Express 站点:

```
express express_parameters
cd express_parameters
npm install
```

2. 将下列内容添加到 app.js 文件的 routes 节中:

```
app.get('/users/:id', function(req, res){
  res.send('show content for user id ' + req.params.id);
});
```

3. 现在启动服务器:

```
node app.js
```

4. 打开你喜爱的 Web 浏览器并浏览 <http://127.0.0.1:3000/users/12>。

5. 可看到:

```
show content for user id 12
```

6. 将浏览器中的地址改为 <http://127.0.0.1:3000/users/99999>。

7. 可看到:

```
show content for user id 99999
```

7.6 让路由保持可维护性

读者已经看到了通过在 app.js 文件中加入声明来创建 POST 和 GET 请求的方法。但是如果使用这一方法, app.js 将很快充满路由声明, 导致维护上的问题。随着更多路由的加入,

我们会发现如下问题。

- 难以对所有路由有个清晰的视图。
- 要找到一个路由要花很长时间。
- 所有的一切都位于一个真的很大的文件里！

读者如果一直在使用 Express 生成器，可能已经看到建议的解决方法就是将路由移到一个单独的文件或不同的文件中，随后在主 app.js 文件中请求这些路由。

```
var routes = require('./routes')
```

如果使用了 Express 生成器，请往 routes 文件夹里看，这里有个包含路由声明的文件 index.js。

```
/*
 * GET home page.
 */

exports.index = function(req, res){
  res.render('index', { title: 'Express' })
};
```

就如我们已经见过的 GET 和 POST 请求那样，这段代码定义了应用程序应当如何对请求做出响应。不同之处在于，我们将函数赋值给一个可以在别的地方使用的变量。通过使用 require 语句并且将其赋予一个变量，就可以在主应用程序文件以 routes.index 来使用。

在主应用程序文件 app.js 中请求了路由文件后，就可以使用 routes.index 声明对主页的路由。

```
app.get('/', routes.index);
```

路由文件的组织方式并不是强制的，但我们将在第 8 章中见到一种典型的做法：按照资源来分路由。

将路由分割成它们自己的文件是个好主意，原因如下。

- 它让代码更容易维护、更可读。
- 可以使用版本控制来快速查看一个文件的历史。
- 它支持应用程序的未来成长，同时又保持其可维护性。

7.7 视图渲染

读者会注意到，我们一直使用 res.render 和 res.send 来指定要把什么内容作为响应来发送。视图渲染描述应用程序在对 HTTP 请求的响应中要如何渲染、要渲染什么。

可能用于视图渲染的响应示例有：

- 发送 HTML；
- 发送 JSON（JavaScript 对象符号）数据；
- 发送 XML（扩展标记语言）数据。

所有这些都在路由的响应节中指定。

大多数时候，我们将使用 `res.render`。它功能强大，可以：

- 渲染模板，包括不同的模板引擎；
- 将本地变量传递给模板；
- 使用布局（layout）文件；
- 发送 HTTP 响应代码。

常见的模式是在路由内使用 `res.render` 来渲染一个模板并且将本地变量传递给它。

```
app.get('/', function(req, res){
  res.render('index.jade', { title: 'My Site' });
});
```

这段代码定义了如下的许多事情。

- 位于 `views/index.jade` 的 `index.jade` 模板要用于渲染页面。
- 本地变量“title”要被传递给 `view/index.jade` 模板。
- 要使用一个布局文件。虽然这并没有显式声明，但 Express 假定如此，除非禁用它。默认的用于布局的文件是 `views/layout.jade`。

如果想使用不同的布局文件，可像这样指定它：

```
res.render('page.jade', { layout: 'custom_layout.jade' });
```

这将使用位于 `views/custom_layout.jade` 的布局文件。

如果不想使用布局文件，可以这样指定：

```
res.render('page.jade', { layout: false });
```

7.8 使用本地变量

在 `res.render` 函数中，本地变量的使用是要掌握的关键所在。使用本地变量可以：

- 设置要在视图层展示的数据；
- 将数据传递到视图层。

读者第一次见到变量的使用是在示例中使用参数展示不同的用户。当读者理解了使用本地变量传输数据的方法，就可以创建更为复杂的应用程序了。

在应用程序发送响应之前，它通常在别的地方设置数据、获取数据。

这可意味着：

- 从数据库获取数据；
- 从 API 获取一些数据；
- 用一些简单的算术来计算一个数；
- 将两个字符串合并在一起。

无论做的是什，这都需要在 `res.render` 函数之前做好并且储存到变量中。在下面的示例中，我们给用户设置一些数据并储存到一个变量里。在设置好数据之后，我们将它作为用户

的一个本地变量传递给视图。

```
app.get('/', function(req, res){
  // You would probably get this data from a data store
  var user = {
    first_name: 'Lord',
    surname: 'Lucan',
    address: 'I'm not telling!',
    facebook_friends: '1356200'
  };
  // Note how the user object is passed as local variable to the view
  res.render('index.jade', { title: 'User', user: user });
});
```

数据现在可以在视图中使用了，于是就可以在 `views/index.jade` 模板中访问它。

```
h1 Accessing data

h3 First Name
p= user.first_name

h3 Surname
p= user.surname
```

在呈现页面时，它会在 `h3` 标题'First Name'下显示'Lord'，在 `h3` 标题'Surname'下显示 Lucan。该数据已经从本地变量用户传递给视图层（View Layer）。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour07/example05` 找到。要将数据传递给视图层，请按下列步骤进行。

1. 创建一个新的 Express 站点：

```
express express_locals
cd express_locals
npm install
```

2. 打开 `app.js` 文件并移除含有如下内容的行：

```
app.get('/', routes.index);
```

3. 在 `app.js` 文件的 `// Routes` 注释之后加入下列代码：

```
app.get('/', function(req, res){
  var user = {
    first_name: 'Barak',
    surname: 'Obama',
    address: 'The White House',
    facebook_friends: '10000000000000'
  };
  res.render('index.jade', { title: 'User', user: user });
});
```

4. 打开 `views/index.jade` 文件并用如下代码替换其内容：

```
h1 Passing data through to the view layer
```

```

h3 First Name
p= user.first_name

h3 Surname
p= user.surname

h3 Address
p= user.address

h3 Facebook Friends
p= user.facebook_friends

```

5. 现在启动服务器:

```
node app.js
```

6. 打开浏览器浏览 <http://127.0.0.1:3000/>。

7. 可看到数据被传递给了视图。

8. 可看到你的名字在寄送的数据中。

7.9 小结

在本章，我们学习了很多与 Express 有关的知识。尤其是我们学习了路由及其在 Express 中的工作方式。我们学习了如何给 Express 应用程序加入 GET 和 POST 请求以及如何使用参数来动态地显示内容。最后，我们学习了如何将数据传递给视图层。

现在，你已经具有了使用 Express 来构建 Node.js 应用程序的极好基础了！

7.10 问与答

问：我在哪里能学到更多关于 HTTP 动词的知识？

答：如果想阅读 HTTP 动词的规范，可在这里阅读：<http://bit.ly/Opx29E>。

问：我是否应当使用 Express 生成器？

答：在刚开始的时候，使用 Express 生成器是开始进行开发工作极好的方法。随着时间的推移你会对如何安排你的应用程序结构有更多自己的看法。Express 提供了一个建议的结构，但无需依赖于此！

问：其他的框架是否可在 Node.js 下使用？

答：有许多框架可用于 Node.js。包括 Geddy (<http://geddyjs.org/>)、Tako (<https://github.com/mikeal/tako>) 以及 Flatiron (<http://flatironjs.org/>)。

7.11 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

7.11.1 问题

1. 如果创建 POST 路由来接收表单，是否会保存到数据库中？
2. 使用参数设置要在路由中展示的数据有何优点？
3. 什么时候应该将路由移到各自的文件中？

7.11.2 答案

1. 不会。除了能确保应用程序接收 POST 请求外，给 POST 请求创建路由不做任何其他事情。我们需要自己对所接收的数据做好处理。通常是对数据进行验证，然后将其保存到某个地方。通过创建路由，不会将表单所提交的数据保存到任何地方。

2. 使用参数使得我们可以重用模板并按一个或多个参数来更改内容。这会让代码可维护，并且可以简单地显示上千不同的记录。通过使用这样的方法，只需相对小的代码库就可为许许多多不同的页面提供能量。

3. 这个问题没有正确答案，但可维护性是这里要考虑的关键问题。如果应用程序有 3 个或者 4 个路由，就不太可能需要另一个文件来保存路由。一旦超过这个范围，那么将路由移到各自文件中就是合理的。怎么做由你自己决定，只是总要记在脑子里的是：代码对其他开发人员而言的可读和可维护程度如何。

7.12 练习

1. 创建一个包括给/about、/contact 和/products 的 GET 请求的 Express 站点。
2. 创建一个带有单个 / 路由的、不使用布局文件的 Express 站点。
3. 创建一个 Express 站点，使用参数根据 GET 请求参数的内容来展示不同内容。使用下列示例数据，应用程序应当在/users/1 显示 Keyser Soze 的详细信息、在/users/2 显示 Roger Kint 的详细信息。此题的一种解法可以在代码示例的 hour07/example06 中找到。

```
var users = {
  1 : {
    first_name: 'Keyser',
    surname: 'Soze',
    address: 'Next door',
    facebook_friends: '4'
  },
  2 : {
    first_name: 'Roger',
    surname: 'Kint',
    address: 'London, England',
    facebook_friends: '100000000000000'
  }
}
```

第 8 章

数据的持久化

在本章中你将学到：

- 从文件读取数据；
- 将数据写入文件；
- 读取环境变量；
- 使用 MongoDB 存储数据；
- 使用 Express 和 MongoDB 创建 CRUD（Create、Read、Update、Delete，即创建、读取、更新和删除）应用程序。

8.1 什么是持久的数据

在 Web 应用程序中，持久的数据这个术语指的是被保存在某个地方，将来某个时候可以在应用程序中使用的数据。

持久的数据的示例有：

- 购物车中的物品；
- 用户账户细节；
- 订购历史；
- 信用卡细节。

数据的存储有下面多种方法。

- 保存在硬盘或闪盘上。
- 保存在计算机内存中。

- 保存在数据库中。
- 保存在 cookie 或会话中。

在开发 Node.js 应用程序时，很快就会遇到以某种形式对数据进化持久化的需求。本章讲解的就是如何做这件事。

8.2 将数据写入文件

如果应用程序需要储存不经常读写的小块信息，那么使用文本文件是快速的、轻量级的选择。Node.js 有个富文件系统模块，允许你随心所欲地与文件系统进行交互。将数据写入文件的方法如程序清单 8.1 所示。

程序清单 8.1 将数据写入文件

```
var fs = require('fs'),
    data = "Some data I want to write to a file";
fs.writeFile('file.txt', data, function (err) {
  if (!err) {
    console.log('Wrote data to file.txt');
  } else {
    throw err;
  }
});
```

第一行请求了文件系统模块并用一个变量保存要写入文件中的数据。而后使用了 `writeFile` 方法将数据写入文件。

这里需要注意的要点如下所示。

- 如果文件不存在，`writeFile` 方法也创建文件。我们无需创建单独创建文件。
- 文件将会被写入与脚本运行位置相关的位置。如果需要控制文件的确切写入位置，可指定完整路径（比如 `/full/path/to/file.txt`）。
- 可能的错误包括文件不存在或者没有读取该文件的权限。

通过写文件来储存数据的一些场景如下所示。

- 将数据备份到文件中。
- 储存时间戳以便脚本使用。
- 记录进程的进程标识符（PID）。
- 记录脚本或应用程序的日志。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example01` 找到。按照如下步骤将数据写入文件。

1. 将程序清单 8.1 中的代码复制到系统中名为 `app.js` 的文件中。
2. 运行脚本：

```
node app.js
```

3. 在 `app.js` 文件所在的文件夹中，可看到一个新创建的文件 (`file.txt`)。
4. 打开这个文件。
5. 文件中应包含 “Some data I want to write to a file”。

8.3 从文件读取数据

如果将数据写入文件，那么极有可能应用程序在某些时候需要读取这个文件。文件系统模块也提供了读文件的方法。假设有个名为 `file.txt` 的文本文件中包含 “Hello Node.js!”，那么可以这样来读它（见程序清单 8.2）。

程序清单 8.2 从文件读取数据

```
var fs = require('fs');
fs.readFile('file.txt', 'utf8', function (err, data) {
  if (!err) {
    console.log(data);
  } else {
    throw err;
  }
});
```

如果运行该脚本，它会读入文件的内容并输出到控制台上。

```
node app.js
Hello Node.js!
```

如同将数据写入文件的示例那样，这里也在第一行代码请求文件系统模块，但这一次使用的是 `readFile` 方法。它需要一个文件和一个字符编码值作为参数。

这里要注意的要点如下所示。

- `file.txt` 的位置与脚本运行的位置有关。我们也可以指定完整路径，比如 `/full/path/to/file.txt`。
- 如果不提供字符编码，那么会返回原始的缓冲区内容。
- 可能的错误包括没有读取文件的权限或文件不存在。

Watch Out!

警告：编码很重要！

计算机上的数据以“位”来储存和传输。位是计算机上的信息单位，是人类不可读的。当计算机输出文本时，它通过编码将位组转换成人类可读的字符。字符编码定义了计算机将位组映射成特定字符集的方法。Web 上用于编码的实际的字符集是 UTF-8，所以要是不知道该用什么编码，那就总是用它吧。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example02` 找到。按下列步骤从文件读入数据。

1. 将程序清单 8.2 中的代码复制到系统中名为 `app.js` 的文件中。
2. 在同一个目录下，创建一个名为 `file.txt` 的文件。
3. 给该文件添加一些文本内容。
4. 运行脚本：

```
node app.js
```
5. 脚本读入该文件并输出写在 `file.txt` 中的文本。

8.4 读取环境变量

如果要做的只是一次性地将数据项目储存起来，以后再取出它，那么环境变量就是个极好的选择。环境变量储存与操作系统相关的数据，比如主（HOME）文件夹所在的位置或者你在系统上的用户名。也可以使用环境变量来储存能被 Node.js 应用程序使用的小数据。

可使用环境变量的示例如下所示。

- 储存数据库连接字符串的细节。
- 给 Amazon S3 这样的服务储存秘密（secret）和键值（key）。
- 为不同的服务储存用户名。

对于特定于应用程序运行位置的数据，环境变量可以有所建树。比如我们有个站点的开发版本，使用的是与真正上线的站点不同的数据库。将数据库设置作为环境变量来保存，然后在 Node.js 应用程序中读入它们，是解决这个问题的一個方法，不过我们得考虑这一方法的安全问题。

注意：环境变量重要！

环境变量为 Node.js 脚本的运行环境设置许多事情，这包括在哪儿找 Node.js 二进制文件，在哪儿找全局安装的 Node.js 模块等。在你使用的任何操作系统中，环境变量通常是自动设置的，但是这些变量扮演重要角色。要查看完整的环境变量清单，请打开终端并在 UNIX 系统下运行 `env` 命令或者在 Windows 下运行 `SET` 命令。

By the
Way

设置环境变量的方法，在 Windows 系统和 UNIX 类型的系统中略有不同。

在 Windows 下，请在终端中运行下列命令：

```
SET SOMETHING='12345678'
```

在 UNIX 下，请在终端中运行下列命令：

```
export SOMETHING='12345678'
```

一旦设置了环境变量，Node.js 只需一行代码就可将其读出：

```
var something = process.env.SOMETHING
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example03` 找到。按照下列步骤读取环境变量。

1. 创建一个名为 `app.js` 的文件并将下列代码复制到该文件中：

```
console.log(process.env.SECRET_KEY)
```

2. 设置 `SECRET_KEY` 环境变量。如果在 UNIX 类的系统中，运行：

```
export SECRET_KEY='c6YFPvdT7Yh3JAFW62EBa5LDe4X'
```

如果在 Windows 类的系统中，运行：

```
SET SECRET_KEY='c6YFPvdT7Yh3JAFW62EBa5LDe4X'
```

3. 运行脚本：

```
node app.js
```

4. 可看到 `c6YFPvdT7Yh3JAFW62EBa5LDe4X` 打印在终端上。

8.5 使用数据库

本章中的示例已经展示了如何使用文件或环境变量设置或获取少量数据。如果使用 Node.js 创建 Web 应用程序，我们肯定要在某一时刻使用数据库或数据存储。需要数据库的一些场景如下所示。

- 让用户保存信息。
- 让用户创建、读取、更新以及删除记录。
- 创建 Web 服务或 API 来使用数据。

Node.js 对大多数主流的数据库都有良好的支持，那么我们该用哪一个呢？

8.5.1 关系数据库

读者可能已经接触过下面这些关系数据库。

- MySQL。
- PostgreSQL。
- Oracle。
- Microsoft SQL Server。
- SQLite。

关系数据库以不同的表储存数据，并使用主键和外键在表之间建立关系。例如，有一个带有数据库的日志应用程序，这个数据库包含了用于记录日志帖子和评论内容的表。通过在表中使用主键和外键，可以很容易地为如下的要求写出查询。

- 以日期排序的最近 10 个日志帖子。
- 某个日志帖子的所有评论。

- 两个时刻之间的所有日志帖子。

将表连接在一起创建出数据视图是关系数据库的一种常见用法。这种用法很强大，因为它仍然保持数据的分离，以便以任何方式对其查询，但它同时提供了对数据的复杂视图。如果读者正在构建的应用程序在数据之间建立了关系，而且数据又能很好地装入表结构，那么关系数据库是个良好的选择。

提示：搜索数据库文件的 npm

可搜索 npm 来寻找用于连接所选择数据库的库文件。这些库文件由世界各地的开发人员编写与维护。大多数主流的数据库都得到了完全的支持。

*Did you
know?*

8.5.2 NoSQL 数据库

NoSQL 是最近几年出现的术语，它涵盖了不符合关系数据库模型要求的大范围的数据库。以下是一些可以归入 NoSQL 数据库的数据库系统。

- Cassandra。
- Redis。
- Memcached。
- MongoDB。
- Hadoop。

这些数据库的功能集会有很大的不同，但通常都不需要有固定的表模式，并避免使用连接，而且是针对水平扩展而设计的。由于 Node.js 是一种颠覆性的技术，所以许多用例（use case）都非常适合使用 NoSQL 数据库。

要选择什么样的数据库，既是一种个人喜好，也由要创建的应用程序类型决定。如果只是想创建基本的应用程序，那么最好选择你熟悉的数据库。

8.6 在 Node.js 中使用 MongoDB

MongoDB 是 Node.js 社区中的一种流行选择。MongoDB 是面向文档的数据库，它不遵照将关系数据连接在一起的关系模型。它可执行关系数据库的大多数功能并且旨在提供高度的可用性和可扩展性。

为了学习如何在 Node.js 中使用 MongoDB，我们使用 Express 创建一个 todo（待办事项）应用程序，用户可用这个程序创建、读取、更新以及删除 todo 条目。

最终的应用程序如图 8.1 所示。

注意：CRUD 的意思是 Create（创建）、Read（读取）、Update（更新）和 Delete（删除）

创建、读取、更新和删除数据是常见的模式，开发人员经常将这些称为 CRUD 操作（Create、Read、Update 和 Delete）。

*By the
Way*

图 8.1

完成后的 todo 应用程序



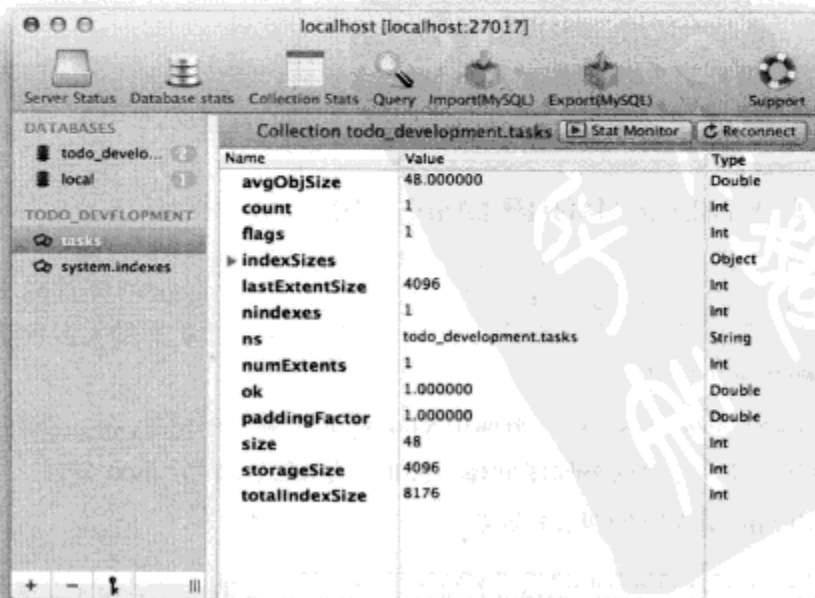
8.6.1 安装 MongoDB

MongoDB 可用于 Mac OSX、UNIX 和 Windows。在这三个平台上安装 MongoDB 的指南可在 <http://www.mongodb.org/display/DOCS/Quickstart> 获得。

除非你很熟悉 MongoDB 命令行界面，否则最好还是使用 GUI（图形用户界面）来开发。MongoDB 有很多可用的 GUI。如果使用 OSX，有 MongoHub（<http://mongohub.todayclose.com/>），它支持连接不同的 MongoDB 实例以及查询、导入、导出数据，还可导入/导出数据到 MySQL（见图 8.2）。MongoHub 是免费的软件。

图 8.2

MongoHub，OSX 上的一个 MongoDB 客户端



对于 Windows，有 MongoVUE（<http://mongovue.com>）。MongoVUE 有一个功能受限的

免费版本，或者也可以按一个用户 35 美元购买。MongoVUE 需要 .NET Framework 支持，所以如果系统上没有 .NET Framework，则需要安装它（见图 8.3）。

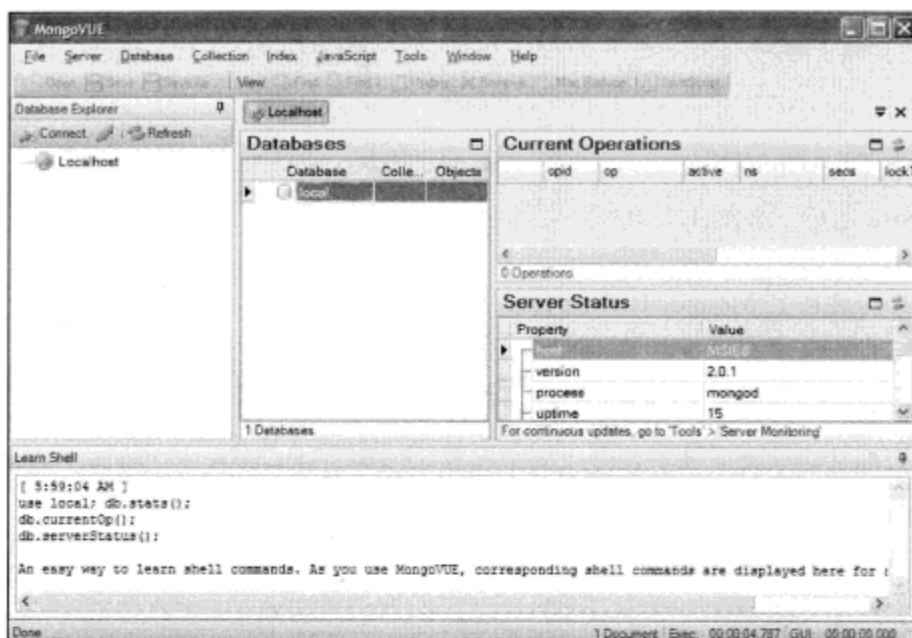


图 8.3

MongoVUE，Windows 下的一个 MongoDB 客户端

对于 Linux，有 JMongoBrowser，这是一个基于 Java 的客户端，所以需要有 Java 运行时（见图 8.4）。

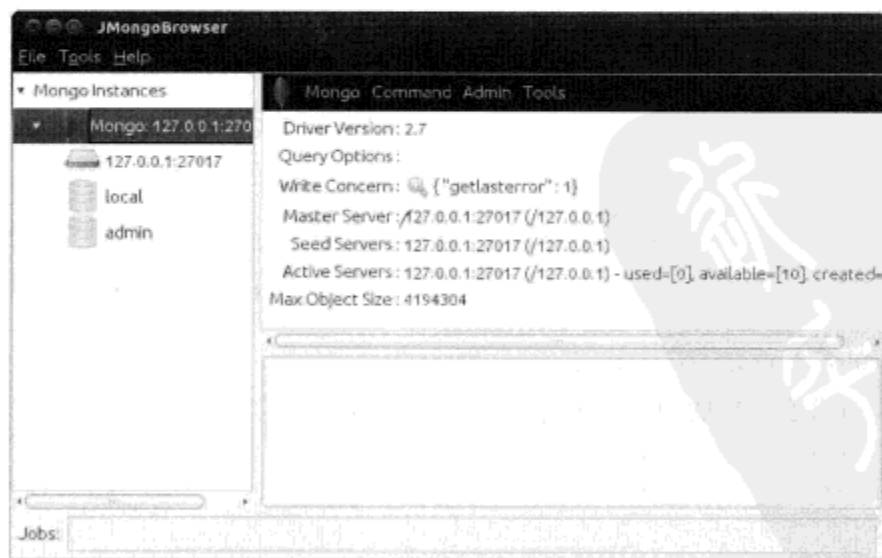


图 8.4

JMongoBrowser，一个 Linux 下的 MongoDB 客户端

8.6.2 连接 MongoDB

Mongoose (<http://mongoosejs.com/>) 是 Node.js 中一个功能齐备的第三方模块，用于处理 MongoDB。我们在 todo 示例应用程序中使用 Mongoose 与 MongoDB 交互，从而保存 todo 条目。

要将 Mongoose 模块加入到项目中，请在项目的 package.json 文件中将其作为依赖模块包含进来：

```
{
  "name": "your-application",
  "version": "0.0.1",
  "dependencies": {
    "mongoose": ">= 2.3.1"
  }
}
```

在加入以后别忘了运行 `npm install`，以将其安装到项目中！

安装完 Mongoose 之后，必须在应用程序文件中请求它。

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

然后，应用程序可连接到 MongoDB 并使用它。

```
mongoose.connect('mongodb://localhost/your_database');
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example04` 找到。按照下列步骤来连接到 MongoDB。

1. 创建一个基本的 Express 站点：

```
express connect_to_mongo
cd connect_to_mongo
npm install
```

2. 打开 `package.json` 文件，将 Mongoose 作为依赖模块加入。这个文件看起来如下所示：

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "2.5.1",
    "jade": ">= 0.0.1",
    "mongoose": ">= 2.3.1"
  }
}
```

3. 运行 `npm install`，将依赖模块安装到应用程序中。
4. 打开 `app.js` 文件并在文件的顶部为应用程序请求 Mongoose。

```
var express = require('express'),
    routes = require('./routes'),
    mongoose = require('mongoose');
```

5. 紧接着上面的代码，加一行代码，以连接到 MongoDB。注意将数据库指定为 “`todo_development`”。无需创建数据库。

```
mongoose.connect('mongodb://localhost/todo_development', function(err) {
  if (!err) {
    console.log('connected to MongoDB');
  } else {
    throw err;
  }
});
```

6. 确认 MongoDB 已经在系统中运行，然后运行以下命令启动服务器：

```
node app.js
```

7. 在控制台中，可看到：

```
Express server listening on port 3000 in development mode
connected to MongoDB
```

8.6.3 定义文档

在 MongoDB 中，没有关系数据中的表的概念。MongoDB 是围绕着文档的思想来组织数据的。一开始这会让人有点摸不着头脑，但很快你就会熟悉它。MongoDB 中的文档具备属性。比如，如果有个狗的文档，那么它就可以有如下属性。

- 名称。
- 血统。
- 颜色。
- 年龄。

对于 todo 应用程序来说，要定义的是一个任务文档。这很简单：只需一个“task”（任务）属性即可。

要想使用 Mongoose 模块在 MongoDB 中定义一个文档，其过程是这样的：通过 Mongoose 提供的 Schema（模式）接口定义，然后声明属性。在使用 Mongoose 的时候，有可能对 MongoDB 的文档和 Mongoose 的模型产生混淆。Mongoose 提供一些额外的功能，比如验证器、getter 和 setter，这些都以模型的概念包装起来。不过，模型最终映射到 MongoDB 文档上。

在属性中可使用不同的数据类型，Mongoose 可声明如下类型。

- String（字符串）。
- Number（数值）。
- Date（日期）。
- Boolean（布尔值）。
- Buffer（缓存）。
- ObjectId（对象 ID）。
- Mixed（混合）。
- Array（数组）。

对于 task 模型而言，只需要一个属性：

```
var Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;

var Task = new Schema({
  task      : String
});

var Task = mongoose.model('Task', Task);
```

注意变量 `Task` 有两个声明。第一个定义了模型的模式 (schema)，第一个定义是使用变量来创建新 task (任务)。必须承认的是，这样的模式 (pattern) 并不优雅，读者可以选择使用不同的变量名。

在模型声明中，`task` 被定义为 `String`。这告诉 MongoDB 数据的类型应该是什么。`String` (字符串) 是字符的一个序列，这对保存像 “Feed the dog” 或者 “Eat Lunch” 这样的任务非常合适。

以上是我们唯一需要做的事情，我们无需运行任何数据库迁移程序，也无需创建结构化查询 (SQL) 脚本来建立数据库。所以，只需少数几行代码，我们就已经定义了数据结构，可以开始在应用程序中构建视图了！

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example05` 找到。按照如下步骤来定义 Mongoose 模型。

1. 回到刚才创建的用来连接 MongoDB 的示例 Express 应用程序。
2. 打开 `app.js` 并声明 `todo` 模型。将以 “`mongo.connect`” 开始的块用以下的代码替换：

```
mongoose.connect('mongodb://localhost/todo_development')
var Schema = mongoose.Schema;
var ObjectId = Schema.ObjectId;

var Task = new Schema({
  task: String
});

var Task = mongoose.model('Task', Task);
```

3. 这样就行了！现在可以开始构建视图了。

Did you Know?

提示 Twitter Bootstrap 很适合用来建立原型

Twitter 的工程团队开源了一个名为 Bootstrap 的框架来引导 (bootstrap) Web 应用程序的布局。它包括涵盖了排版、表单、按钮等内容的基础 (base) HTML 和 CSS。如果读者想找一个能够快速给应用程序加入样式的方法，那么 Bootstrap 是个很好的选择。我们使用 Bootstrap 创建 todo 应用程序。

8.6.4 将 Twitter Bootstrap 包含进来

我们使用 Twitter Bootstrap 来设置这个示例应用程序的视图样式。要想将其包含进来，请更新 `views/layout.jade` 文件，让这个文件包含如下内容：

```
!!
html
  head
    title= title
    link(rel='stylesheet', href='http://twitter.github.com/bootstrap/1.4.0/
➔bootstrap.min.css')
  body
    section.container!= body
```


8.6.5 索引 (Index) 视图

在有了数据的结构之后，可以开始创建视图了。我们按第 7 章中的相同模式来进行：首先创建路由然后加入视图模板。

任务的索引视图包含所有任务的列表。要创建这个视图，需要在 `app.js` 中添加一个新的路由：

```
app.get('/tasks', function(req, res){
  res.render('tasks/index', {
    title: 'Todos index view' });
});
```

接下来，通过在 `views/tasks/index.jade` 这个地方创建一个包含如下内容的新文件来添加一个视图文件：

```
h1 Your Tasks
```

希望读者能够熟悉这个来自第 7 章的模式！下一步是从 MongoDB 获取条目以便进行显示。通过 Mongoose 模块，有一个简单的方法将储存在模型中的全部记录获取出来。

```
YourModel.find({}, function (err, docs) {
  // do something with the data here
});
```

现在可以更新任务索引路由，从而查询 MongoDB 并将结果传递到视图层。

```
app.get('/tasks', function(req, res){
  Task.find({}, function (err, docs) {
    res.render('tasks/index', {
      title: 'Todos index view',
      docs: docs
    });
  });
});
```

注意 `docs` 被传递到了视图层，而且它包含了从 MongoDB 所返回的所有记录。现在可以更新位于 `views/tasks/index.jade` 的视图文件，以便展示这些结果（如果有的话）。

```
h1 Your tasks

- if(docs.length)
  table
    tr
      th Task
      each task in docs
        tr
          td #{task.task}
- else
  p You don't have any tasks!
```

Jade 模板通过调用 `docs` 的 `length` 方法检查是否有任务存在。如果没有任务，它就会以 `false` 返回，而后模板告诉用户没有任务需要做。如果有任务（译者注：原文为文档，疑笔误），就

循环输出每一个任务。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example06` 找到。按照下列步骤显示 MongoDB 中的数据。

1. 回到刚才在处理的示例 `Express todo` 应用程序。
2. 打开 `app.js` 并添加任务路由。这会从 MongoDB 中获取所有任务。

```
app.get('/tasks', function(req, res){
  Task.find({}, function (err, docs) {
    res.render('tasks/index', {
      title: 'Todos index view',
      docs: docs
    });
  });
});
```

3. 在 `views/tasks/index.jade` 处创建一个新文件并添加如下内容：

```
h1 Your tasks

- if(docs.length)
  table
    tr
      th Task
      each task in docs
        tr
          td #{task.task}
- else
  p You don't have any tasks!
```

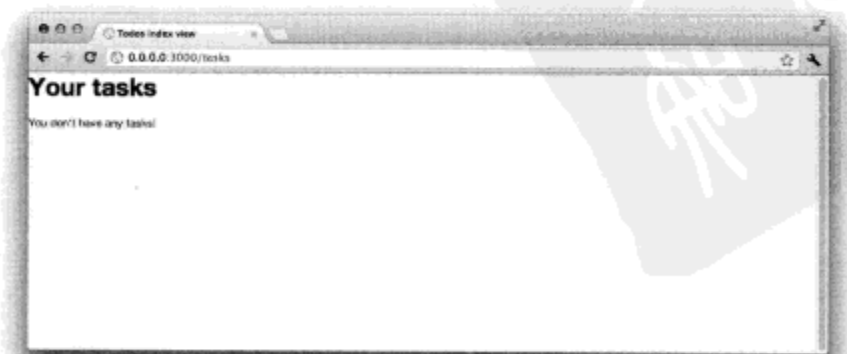
4. 使用如下命令启动服务器：

```
node app.js
```

5. 浏览到 `http://0.0.0.0:3000/tasks`。
6. 可看到一个页面，上面显示没有任务（见图 8.5）。我们需要一个能添加任务的方法！

图 8.5

todo 应用程序的
索引视图



8.6.6 创建 (Create) 视图

到现在为止，我们有一个能告诉我们没有任务需要去做的页面，但却不能添加任务。要让这个应用 (app) 有创建记录的功能，需要创建两个新路由。首先，要有个新路由来让用户输入任务：

```
app.get('/tasks/new', function(req, res){
  res.render('tasks/new.jade', {
    title: 'New Task'
  });
});
```

与之相关的视图文件展示一个表单，以便让用户提交任务。由于我们使用 Twitter Bootstrap 来提供表单的样式，所以在元素上通过类名来应用正确的样式。

```
h1 New task view

form(method='post', action='/tasks')
  fieldset
    legend Add a task
  div.clearfix
    label Task
    div.input
      input(name='task[task]', class='xlarge')
    div.actions
      input(type='submit', value='Save', class='btn primary')
      button(type='reset', class='btn') Cancel
```

在这个表单中，有一个单一的字段将数据作为 POST 请求发送给 /tasks，而这个寄送的数据将会被接收并保存到 MongoDB 中。

为了接收数据，要创建一个新的路由来接收 POST 请求并创建一个任务。使用 Mongoose 可以轻松完成该任务：

```
app.post('/tasks', function(req, res){
  var task = new Task(req.body.task);
  task.save(function (err) {
    if (!err) {
      res.redirect('/tasks');
    }
    else {
      res.redirect('/tasks/new');
    }
  });
});
```

在上述代码示例的第 2 行里，实例化了一个 new Task 模型并将 POST 数据传递给它。在第 3 行，调用了 task 的 save 方法。如果不出错误，那么这会将任务保存到 MongoDB。一旦保存了任务，程序就将用户重新定位到索引视图上。

在用户可以添加任务后，就应该在索引视图上加入指向新视图的链接，以便用户创建任务。

```

P
a(href='/tasks/new', class='btn primary') Add a Task

```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example07` 找到。按照下列步骤增加任务创建功能。

1. 回到刚才在处理的示例 Express todo 应用程序。
2. 打开 `app.js` 并添加新任务路由。

```

app.get('/tasks/new', function(req, res){
  res.render('tasks/new.jade', {
    title: 'New Task'
  });
});

```

3. 在 `views/tasks/new.jade` 处创建一个新文件并添加如下内容：

```

h1 New task view

form(method='post', action='/tasks')
  fieldset
    legend Add a task
    div.clearfix
      label Task
      div.input
        input(name='task[task]', class='xlarge')
      div.actions
        input(type='submit', value='Save', class='btn primary')
        button(type='reset', class='btn') Cancel

```

4. 在 `app.js` 文件中添加第二个新路由，以接收 POST 请求并创建任务：

```

app.post('/tasks', function(req, res){
  var task = new Task(req.body.task);
  task.save(function (err) {
    if (!err) {
      res.redirect('/tasks');
    }
    else {
      res.redirect('/tasks/new');
    }
  });
});

```

5. 添加一个指向 `views/tasks/index.jade` 的链接以使用户创建任务：

```

P
a(href='/tasks/new', class='btn primary') Add a Task

```

6. 使用如下命令启动服务器：
- ```
node app.js
```
7. 浏览到 `http://0.0.0.0:3000/tasks`。
  8. 可看到一个指向添加任务的链接，并且可以创建新的任务（见图 8.6）。



图 8.6

todo 应用程序的创建视图

### 8.6.7 编辑视图

todo 应用程序当前允许用户添加新任务并查看任务清单。但如果他们想编辑某个现有的任务该怎么办？为了添加编辑功能，需要两个路由：一个用于显示可进行编辑的任务的表单，另一个用来接收寄送过来的数据并更新记录。

但首先，需要一个检索可编辑的任务的方法，这样才有可能将其显示在表单中。由于每个 MongoDB 记录都有一个唯一的 ID，所以可用这个 ID 来获取记录。Mongoose 提供了一个简单的方法来做这件事：

```
YourModel.findById(someId, function (err, doc) {
 // do something with the data here
});
```

通过使用 `findById`，可以找到记录，于是就可以给编辑表单创建一个路由。这个路由捕获 id 请求参数，以便从 MongoDB 中检索记录。注意路由将任务传递给视图层的方法，这是为了将现有的数据展示给用户。

```
app.get('/tasks/:id/edit', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 res.render('tasks/edit', {
 title: 'Edit Task View',
 task: doc
 });
 });
});
```

相关的视图将数据展示给用户并允许用户编辑数据。由于这个表单执行的是更新操作，所以要使用 PUT 请求。Express 有个辅助方法，通过将名称指定为“`_method`”将标准的寄送请求转换为其他方法。表单在一个隐藏的输入字段中使用这一功能来确保 Express 将 POST 转换成 PUT。这件事由 Express 来做，因为大多数 HTML 表单只支持 GET 和 POST。

```
h1 Edit task view
form(method='post', action='/tasks/' + task.id)
 input(name='_method', value='PUT', type='hidden')
```

```

fieldset
 legend Editing task
 div.clearfix
 label Task
 div.input
 input(name='task[task]', class='xlarge', value="#{task.task}")
 div.actions
 input(type='submit', value='Save', class='btn primary')
 button(type='reset', class='btn') Cancel

```

为了接收数据，需要添加另一个路由来接收 PUT 请求并更新记录。如果更新成功，就将用户重定向到索引视图上。

```

app.put('/tasks/:id', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 doc.task = req.body.task.task;
 doc.save(function(err) {
 if (!err){
 res.redirect('/tasks');
 }
 else {
 // error handling
 }
 });
 });
});

```

最后，在应用程序支持编辑功能后，就可以在主页上添加一个链接，以便让用户编辑记录。

```

- if(docs.length)
 table
 tr
 th Task
 tr
 th
 each task in docs
 tr
 td #{task.task}
 td
 a.btn(href="/tasks/#{task.id}/edit") Edit
- else
 p You don't have any tasks!

```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example08` 找到。按照下列步骤让任务可以被编辑。

1. 回到刚才在处理的示例 Express todo 应用程序。
2. 打开 `app.js` 并添加编辑任务路由：

```

app.get('/tasks/:id/edit', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 res.render('tasks/edit', {
 title: 'Edit Task View',
 task: doc
 });
 });
});

```

```

 });
 });
});

```

3. 在 `views/tasks/edit.jade` 处创建一个新文件并添加如下内容:

```

h1 Edit task view

form(method='post', action='/tasks/' + task.id)
 input(name='_method', value='PUT', type='hidden')
 fieldset
 legend Editing task
 div.clearfix
 label Task
 div.input
 input(name='task[task]', class='xlarge', value="#{task.task}")
 div.actions
 input(type='submit', value='Save', class='btn primary')
 button(type='reset', class='btn') Cancel

```

4. 在 `app.js` 文件中加入第二个新路由以便接收 PUT 请求并更新任务:

```

app.put('/tasks/:id', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 doc.task = req.body.task.task;
 doc.save(function(err) {
 if (!err){
 res.redirect('/tasks');
 }
 else {
 // error handling
 }
 });
 });
});

```

5. 更新 `tasks/index.jade` 视图, 使其包含指向编辑视图的链接:

```

- if(docs.length)
 table
 tr
 th Task
 th
 each task in docs
 tr
 td #{task.task}
 td
 a.btn(href="/tasks/#{task.id}/edit") Edit
- else
 p You don't have any tasks!

```

6. 使用如下命令启动服务器:

```
node app.js
```

7. 浏览到 `http://0.0.0.0:3000/tasks`。

8. 可看到一个指向编辑任务的链接, 可以编辑任务了 (见图 8.7)。

图 8.7

todo 应用程序的  
编辑视图



### 8.6.8 删除任务

todo 应用程序现在可以创建、阅读、编辑记录了。最后一项功能是加入删除记录的能力。首先，要创建一个路由来接收 DELETE 请求，它使用来自请求的 id 找到要删除的任务。这里同样使用 Mongoose 提供的 findById 方法来查找要删除的任务。如果用传递进来的 id 没能找到任务，那么就返回错误。

```
app.del('/tasks/:id', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 if (!doc) return next(new NotFound('Document not found'));
 doc.remove(function() {
 res.redirect('/tasks');
 });
 });
});
```

为了发送 DELETE 请求，需要有个表单。这个表单要加入到每个任务记录的输出，以便用户删除它。就如编辑动作所用的 PUT 请求那样，我们使用名为“\_method”的隐藏字段，以便 Express 将 POST 请求转换为 DELETE。

```
- if(docs.length)
 table
 tr
 th Task
 th
 each task in docs
 tr
 td #{task.task}
 td
 a.btn(href="/tasks/#{task.id}/edit") Edit
 td
 form(method='post', action="/tasks/" + task.id)
 input(name='_method', value='DELETE', type='hidden')
 button.btn(type='submit') Delete
- else
 p You don't have any tasks!
```



## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour08/example09` 找到。按照下列步骤让任务可以被删除。

1. 回到刚才在处理的示例 Express todo 应用程序。
2. 打开 `app.js` 并添加删除任务路由：

```
app.del('/tasks/:id', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 if (!doc) return next(new NotFound('Document not found'));
 doc.remove(function() {
 res.redirect('/tasks');
 });
 });
});
```

3. 更新 `tasks/index.jade` 视图，使其包含指向删除（译者注：原文为编辑，疑为笔误）视图的链接：

```
- if (docs.length)
 table
 tr
 th Task
 th
 each task in docs
 tr
 td #{task.task}
 td
 a.btn(href="/tasks/#{task.id}/edit") Edit
 td
 form(method='post', action="/tasks/" + task.id)
 input(name='_method', value='DELETE', type='hidden')
 button.btn(type='submit') Delete
- else
 p You don't have any tasks!
```

4. 用如下命令启动服务器：  
`node app.js`
5. 浏览到 `http://0.0.0.0:3000/tasks`。
6. 可看到一个指向删除任务的链接，可以删除任务了。

### 8.6.9 添加闪出消息

todo 应用程序可以创建、阅读、编辑、删除了！如果应用程序能够将操作成功的消息告诉用户岂不是很棒！通过 Express 框架，可以通过首先启用会话支持将这个功能加入到应用程序中。

为了在 Express 中启用会话，以下两行内容需要加入到 `app.js` 文件的 `app.configure` 部分：

```
app.use(express.cookieParser());
app.use(express.session({ secret: "OZhCLfxlGp9TtzSXmJtq" }));
```

第一行代码告诉 Express 启用 cookieParser。这使得 Express 可以保存会话。第二行给 Express 设置一个用于保护会话安全的密码，它可以是任何字符串。在 random.org 有一个工具可生成随机的字符串：<http://www.random.org/strings/>。

通知用户“发生了什么情况”的消息是通过一个常见的术语——闪出消息（flash message）——来描述的：图 8.8 展示了用户创建任务之后展示给用户的消息。

图 8.8

闪出消息的示例



一旦在应用程序中启用了会话支持，就有可能在路由中设置闪出消息了：

```
req.flash('info', 'It worked!');
req.flash('error', 'Something went wrong!');
```

这会把消息加入到用户的会话中，于是就可以在视图之间使用。在将用户从创建视图重定向到索引视图时，就可以给用户展示一个闪出消息。

为了将闪出消息加入到会话中，要在重定向之前将其包含在路由中。

```
app.post('/tasks', function(req, res){
 var task = new Task(req.body.task);
 task.save(function (err) {
 if (!err) {
 req.flash('info', 'Task created');
 res.redirect('/tasks');
 }
 else {
 // Error handling
 }
 });
});
```

为了显示闪出消息，有一个 Jade 模板可检查消息是否存在，如果存在则显示出来。以下示例使用 Twitter Bootstrap 的可用样式来更改提示消息的样式。

```
-if(typeof flash != 'undefined')
- if(flash.warning)
 div.alert-message.warning
 p= flash.warning
- if(flash.info)
 div(data-alert='alert').alert-message.success
 p= flash.info
```

由于闪出消息是可跨模板重用的东西，所以这是将代码抽象成一个 `mixin` 的好例子。我们在第 6 章介绍了 `mixin`，它们让代码段可以在不同的视图之间重用。

闪出消息可以抽象到一个 `mixin` 文件中并保存到 `views/mixins/flash-messages.jade`:

```
mixin flash-messages(flash)
 - if(flash.warning)
 div.alert-message.warning
 p= flash.warning
 - if(flash.info)
 div(data-alert='alert').alert-message.success
 p= flash.info
```

这在以后将可以跨视图重用。要将闪出消息添加到 `tasks/index.jade` 视图中，需要添加如下几行代码。现在要是有了闪出消息，就会显示给用户了。

```
-if(typeof flash != 'undefined')
 include ../mixins/flash-messages
 mixin flash-messages(flash)
```

最后，需要更新一下路由，以便根据操纵数据的输出结果来设置闪出内容。我们通过 `req.flash` 来完成。

```
app.post('/tasks', function(req, res){
 var task = new Task(req.body.task);
 task.save(function (err) {
 if (!err) {
 req.flash('info', 'Task created');
 res.redirect('/tasks');
 }
 else {
 req.flash('warning', err);
 res.redirect('/tasks/new');
 }
 });
});
```

完整的示例请参考本书代码示例中的 `hour08/example10`。

### 8.6.10 验证输入的数据

对用户输入的数据进行验证对任何 Web 应用程序而言都是重要的一部分工作。在我们已经做过的这些示例中，目前还没有验证。用户可以创建一个没有输入任何文本内容的任务！

Mongoose 允许在模型内对示例开始部分定义的属性设置验证。我们也可以创建自定义的函数，使其按自己希望的任意方式来验证数据。要验证字符串是否存在，可以创建一个简单的 JavaScript 函数。

```
function validatePresenceOf(value) {
 return value && value.length;
}
```

现在可以更新模型来验证字符串是否存在了：

```
var Task = new Schema({
 task : { type: String, validate: [validatePresenceOf, 'a task is required'] }
});
```

现在，当用户提交表单时如果字符串不存在，验证就会失败，记录就不会被保存，而且应用程序可以通过闪出消息来告诉用户。

更新寄送路由使其捕获错误并设置一个警告闪出消息（见图 8.9）。注意如果数据没能通过验证，`err` 会是 `true`，用户会被重定向回带有警告信息的表单。

```
app.post('/tasks', function(req, res){
 var task = new Task(req.body.task);
 task.save(function (err) {
 if (!err) {
 req.flash('info', 'Task created');
 res.redirect('/tasks');
 }
 else {
 req.flash('warning', err);
 res.redirect('/tasks/new');
 }
 });
});
```

图 8.9

验证闪出消息



完整的示例请参考本书代码示例中的 `hour08/example11`。

## 8.7 小结

在本章，我们学习了持久的数据以及使用 Node.js 实现数据持久化的一些方法。我们学习了读写文件，并且学习了在应用程序中使用环境变量设置数据值的方法。最后我们完成了一个完整的 CRUD（创建、读取、更新、删除）示例应用程序，并学习了闪出消息和验证。

## 8.8 问与答

问：什么时候应该将数据保存到文件，而什么时候应该使用数据库？

答：如果只是偶尔保存少量数据、执行备份或者记录应用程序的日志数据，那么选择文件就好了。如果 Web 应用程序的用户需要频繁访问数据，通常这是需要将数据移到数据库中的信号。从磁盘的文件上读取数据要比从数据库读取数据慢得多。

问：我应该选择哪种数据库？

答：有许多数据库系统可以选择，而且它们各自有不同的优势。对于一般目的而言，Node.js 社区的许多人倾向使用 MongoDB，而且有许多库文件支持与 MongoDB 的交互。但是，如果读者更熟悉其他数据库，Node.js 也可通过第三方库来对这些数据库给予良好支持。基本上说，这是一种个人选择！

问：在保存数据之前验证数据为什么是重要的？

答：为了创建稳定的软件，应用程序需要知道什么类型的数据是所期望的。如果在 Web 上放一个表单，就会对人们填写的内容感到惊讶。我们应该对输入表单中的意料之外的数据有所预期，并且就此对数据进行验证。好的验证也会带来更好的用户体验，因为应用程序可以通过有帮助的提示信息来帮助用户修复错误。

## 8.9 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

### 8.9.1 问题

1. 可否将敏感数据，比如密码，保存在文本文件中？
2. 用于更新记录的 HTML 动词是什么？
3. 本章中，安全上的考虑出现在示例中是什么时候？

### 8.9.2 答案

1. 不可。将数据存储在文本文件中适合于非敏感数据，比如 PID（进程标识）或者日志。
2. 用于更新记录的动词是 PUT。因为 HTML 表单本身不支持 PUT 请求，所以 Express 提供了一个方便的方法添加这一功能。
3. 对于本章的示例，应改进其安全性。读者应该将任何从应用程序以外的数据都当成是“不洁净的”。这就是说，在接收数据的时候应当对数据进行清洁，以防止像 SQL 注入和 XSS 这样的攻击。与之相似，在显示数据的时候，也应该明智地检查数据。在处理数据的时候，有一句安全箴言值得一说再说，那就是：“过滤输入，转义输出”。

## 8.10 练习

1. 创建一个简单的 Node.js 应用程序，将你的名字写入一个文本文件。
2. 再创建一个应用程序，读入在练习 1 中创建的文本文件内容并将“Hello [你的名字]!”打印到控制台上。
3. 在你创建的示例应用程序中，在 Task 模型中添加一个名为“Details”的新属性，以便用户可以对任务添加细节信息。请更新表单和视图，使其包括这一属性。



## 第3部分 调试、测试与部署

第9章 调试 Node.js 应用程序

第10章 测试 Node.js 应用程序

第11章 部署 Node.js 应用程序



## 第9章

# 调试 Node.js 应用程序

在本章中你将学到：

- 什么是调试；
- 使用 `STDIO` 模块进行调试；
- 使用 Node.js 的调试器进行调试；
- 使用 Node Inspector 进行调试。

## 9.1 调试

开始软件开发之后不用太久，你就会碰到开发人员每天都在学习与之斗法的东西：`bug`。软件中的 `bug` 指的是导致应用程序不正常行为的错误或问题。以下是一些 `bug` 的示例。

- 源代码中的语法错误导致的错误。
- 没有按期望中的行为方式执行的功能。
- 性能低下的代码让应用程序运行缓慢。
- 安全上的脆弱性。

`bug` 可以由开发人员发现，也可以由应用的用户报告。不论哪种方式，研究 `bug`（或者说调试）都是开发人员的职责。许多开发人员使用 `bug` 或问题记录器来记录 `bug` 报告，而一旦软件发布，开发人员的大部分工作就是处理 `bug` 报告了。如果找到一个 `bug` 或者用户报告了一个 `bug`，就需要进行调试。调试的意思就是研究哪儿出错了并解决错误。一旦 `bug` 得到解决，就可以说错误得到修正或者 `bug` 被碾扁了。在本章中，我们学习调试 Node.js 应用程序时可用的一些工具和技艺。



## 9.2 STDIO 模块

Node.js 核心带有 STDIO 模块，这是调试应用程序的一种轻量级的方法，它对许多工作都极为适合。这个模块无需其他依赖模块，也就是说不需要设置就能使用它。它遵循通过使用控制台对象在浏览器中调试 JavaScript 的惯例，所以如果读者在浏览器中调试过 JavaScript 的话，应该对此感到熟悉。虽然并不是浏览器的控制台对象中的所有特性都得到支持，但也八九不离十。

在 STDIO 模块中，信息被记录到终端中，使用这个模块的方法是直白的。只需在脚本中想要往终端上记录信息的地方添加代码即可。当脚本运行到这一行时，就会往终端记录信息：

```
console.log("Look! I'm in your terminal!")
```

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour09/example01` 找到。按照下列步骤使用 STDIO 模块进行调试。

1. 创建一个名为 `app.js` 的文件并将下列代码复制到文件中：

```
console.log('Debugging message')
```

2. 运行脚本：

```
node app.js
```

3. 可在终端上看到“Debugging Message!”。

对下列场景的调试来说，使用 `console.log()` 经常是快速而直接的解决方案。

- 检查变量或字符串的值。
- 记录脚本调用了一个函数。
- 记录来自第三方服务的响应。

如果想记录错误，可使用 `console.warn()`，它又名 `console.error()`，这两个方法都会打印到标准错误流中。

#### 提示：程序往不同的流中写入

在 UNIX 类型的系统中，数据流经常是连接在一起的，或者是接上管道（pipe）的。这就让小的不同的程序可以通过组合在一起创建更为复杂的功能。为了协助程序粘合在一起，程序可往标准输入、输出和错误接口中写入。这些称为标准输入（`stdin`）、标准输出（`stdout`）和标准错误（`stderr`）。测试运行器（test runner）就是实际利用这些标准流的一个示例。当一组自动化的测试通过时，输出就会被发送到标准输出上。当测试失败时，消息就会被送到标准错误上。这就让其他软件可以理解测试的运行结果并依此而动作。Node.js 的 STDIO 模块遵照这一惯例，所以要确认写入的是正确的流！

**Did you  
Know?**

PDG

记录错误产生的原因是 `console.error()` 的用处。的一个示例。比如，如果不检查某个函数是否定义，或者如果某个函数在编写了调用它的代码之后被删除，那么就会有错误抛出。为了让问题显而易见，可在一个 `try catch` 语句中使用 `console.error()` 将错误的发生原因记录下来（见程序清单 9.1）。

#### 程序清单 9.1 使用 `console.error()`

```
function notDefined(){
 try {
 someFunction(); // undefined
 } catch (e) {
 console.error(e);
 }
}
notDefined();
```

如果运行这段代码，就会在终端上看到如下错误：

```
[ReferenceError: someFunction is not defined]
```

#### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour09/example02` 找到。按照下列步骤使用 `STDIO` 模块将错误写入 `stderr`。

1. 将程序清单 9.1 中的代码复制到系统中名为 `app.js` 的文件中。
2. 运行脚本：

```
node app.js
```

3. 可在终端上看到 `[ReferenceError: someFunction is not defined]`，这表示 `someFunction` 未定义。

使用 `console.error()` 很有用处，因为它提供错误的类型信息抛出以及问题的确切所在。在本示例中，有可能以后会定义 `someFunction()` 函数，而在这个函数中有错误存在。这种情况如程序清单 9.2 所示。

#### 程序清单 9.2 用 `console.err()` 捕获错误

```
function someFunction(){
 return undefinedVar; // undefined
}
function notDefined(){
 try {
 someFunction(); // now defined
 } catch (e) {
 console.error(e);
 }
}
notDefined();
```

如果再次运行这段代码，就会看到不同的错误信息，并且立即清楚地给出问题所在：

```
[ReferenceError: undefinedVar is not defined]
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour09/example03` 找到。以下这个示例展示如何理解使用 STDIO 模块显示的错误。

1. 将程序清单 9.2 中的代码复制到系统中名为 `app.js` 的文件中。
2. 运行脚本：

```
node app.js
```

3. 可在终端上看到 `[ReferenceError: undefinedVar is not defined]`，这表示 `undefinedVar` 没有定义。

使用 `console.error()` 是找出错误为何被抛出的轻量级方法。要想探索代码的瓶颈所在或者想快速检测代码的某些部分的性能基准，可使用 STDIO 模块提供的 `console.time()` 和 `console.timeEnd()`。这两个方法通过使用一个标签作为参数，这个参数将两个方法绑在一起。要想启动定时器，可使用 `console.time()`；要停止定时器，使用 `console.timeEnd()`。这两个方法对于检测少量代码的性能基准或者任何与时间相关的基准都是有用的。使用这些方法，可很容易地找出代码运行缓慢的地方或者通过比较执行相同代码的两种方法来优化性能（见程序清单 9.3）。

### 程序清单 9.3 一个简单的基准测试

```
var sum = 0;
var arr = new Array(1000000);

for (var i = 0; i < arr.length; i++) {
 arr[i] = Math.random();
}

console.time('for-loop-1');
for (var i in arr) {
 sum += arr[i];
}
console.timeEnd('for-loop-1');

console.time('for-loop-2');
for (var i = 0; i < arr.length; i++) {
 sum += arr[i];
}
console.timeEnd('for-loop-2');
```

在这个示例中，要做的是探索执行一个对有百万个随机数的数组的简单操作最为高效的方法。有了 STDIO 模块提供的方法，就可以通过对两个循环计时来探索哪种方法更快。这在调试性能问题的时候很有用。运行此脚本显示，第二种方法更快。

```
for-loop-1: 431ms
for-loop-2: 27ms
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour09/example04` 找到。以下步骤展示的是使用 STDIO 模块测试性能。

1. 将程序清单 9.3 中的代码复制到系统中名为 `app.js` 的文件中。
2. 运行脚本：

```
node app.js
```

3. 可在终端上看到两个循环的执行时间：

```
for-loop-1: 431ms
for-loop-2: 27ms
```

如果希望看到在脚本执行过程中任意一点的堆栈内的位置踪迹内容，可使用 `console.trace()` 方法（见程序清单 9.4）。

#### 程序清单 9.4 查看堆栈踪迹

```
function notDefined(){
 console.trace();
 try {
 someFunction();
 } catch (e) {
 console.error(e);
 }
}
notDefined();
```

这会将插入该代码的那一点的堆栈踪迹打印出来：

```
Trace:
 at notDefined (/Users/george/code/nodejsbook.io.examples/hour09/example05/app.
 ↪js:2:11)
 at Object.<anonymous> (/Users/george/code/nodejsbook.io.examples/hour09/
 ↪example05/app.js:10:1)
 at Module._compile (module.js:432:26)
 at Object..js (module.js:450:10)
 at Module.load (module.js:351:31)
 at Function._load (module.js:310:12)
 at Array.0 (module.js:470:10)
 at EventEmitter._tickCallback (node.js:192:40)
```

#### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour09/example05` 找到。以下是使用 `STDIO` 模块打印堆栈踪迹的方法。

1. 将程序清单 9.4 中的代码复制到系统中名为 `app.js` 的文件中。
2. 运行脚本：

```
node app.js
```

3. 应该在终端上看到堆栈踪迹。

#### 注意：堆栈踪迹有助于调试

堆栈踪迹是应用程序中某个点的函数或方法调用清单。通常情况下，错误或者 bug 可能位于从其他文件或模块中包含进来的代码里。堆栈跟踪让开发人员可以了解到脚本如何执行到某个点，并追踪到问题发生的位置。

## 9.3 Node.js 调试器

Node.js 提供对 V8 调试器的访问能力，通过在脚本中包含它，我们可以在代码中设置断点，从而遍历代码。断点用于让开发人员停止脚本的执行并检查所发生的情况。断点是极强大的调试技巧，因为它们可用来检查应用程序在执行当中的某个点的状态。要想使用 Node.js 调试器设置断点，可在希望断点发生的地方加入下列代码行：

```
debugger
```

现在，可以通过附加 `debug` 参数运行 Node.js 的方法来运行带有调试功能的程序了：

```
node debug app.js
```

Node.js 调试器让我们可以做如下事情。

- 按步执行代码。
- 设置断点。
- 在脚本的断点处获取包括堆栈跟踪和已装载的脚本在内的信息。
- 进入 REPL (Read、Evaluate、Print、Loop，读取、演算、打印、循环) 以便在断点处详细调试。
- 运行、重启以及停止脚本。

插入断点然后使用 REPL 来检查对象和变量状态是使用 Node 调试器时的常用模式。这在调试 JavaScript 中的作用域问题时很有用。以下示例摘自 Douglas Crockford 的《JavaScript: The Good Parts》，这是将 JavaScript 中作用域这个很容易让人混淆的问题展示出来的极好示例。在下列脚本的不同点上，a、b 和 c 的值是不同的，也可能根本没有设置（见程序清单 9.5）。

程序清单 9.5 JavaScript 作用域示例

```
var foo = function(){
 var a = 3, b = 5;
 var bar = function() {
 var b = 7, c = 11;
 a += b + c;
 }
 bar ();
};
foo();
```

如果这样的函数没有返回期待中的值作为结果，那么在脚本中放入调试器断点以便调试器能按步执行脚本就是个合理的调试方法。而后就可使用调试器的 REPL 来检查变量的值（见程序清单 9.6）。

程序清单 9.6 使用断点的 JavaScript 作用域示例

```
var foo = function(){
 var a = 3, b = 5;
 debugger;
 var bar = function() {
 var b = 7, c = 11;
```

```

 a += b + c;
 debugger;
 }
 bar ();
 debugger;
};
foo();

```

这段脚本现在可使用 `debug` 参数来运行，调试器会在第 1 行中断。这会冻结脚本的执行过程以及在该点上脚本的状态。

```

node debug app.js
< debugger listening on port 5858
connecting... ok
break in app.js:1
 1 var foo = function(){
 2 var a = 3, b = 5;
 3 debugger;
debug>

```

为了前进到第一个断点，键入 `cont`。这个命令将调试器的运行进度移至脚本的第 3 行。现在可以进入 REPL 并查询 `a`、`b` 和 `c` 的值了：

```

debug> repl
Press Ctrl + C to leave debug repl
> a
3
> b
5
> c
ReferenceError: c is not defined

```

要想退出 REPL，请按 `Ctrl+C` 并键入 `cont` 跳到代码中的下一个断点。在这个示例中，它将移动到代码的第 7 行。这次又可以输入 `repl` 进入 REPL 查询 `a`、`b` 和 `c` 的值了。

```

debug> repl
Press Ctrl + C to leave debug repl
> a
21
> b
7
> c
11

```

要想跳到最后一个断点，请再次按 `Ctrl+C` 并键入 `cont`。这是代码中的最后一个断点，输入 `repl`，又可以检查 `a`、`b` 和 `c` 的值了。

```

debug> repl
Press Ctrl + C to leave debug repl
> a
21
> b
5
> c
ReferenceError: c is not defined

```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour09/example06` 找到。以下步骤展示了使用 V8 调试器调试代码的方法。

1. 将程序清单 9.6 中的代码复制到系统中名为 `app.js` 的文件中。
2. 运行脚本时启用调试器：

```
node debug app.js
```

3. 可看到调试器启动，然后在脚本执行的时候停止。
4. 键入 `cont` 继续执行到第一个断点。
5. 键入 `repl` 并查询 `a`、`b` 和 `c` 的值。
6. 按 `Ctrl+C` 然后键入 `cont` 跳到下一个断点。
7. 键入 `repl` 并查询 `a`、`b` 和 `c` 的值。
8. 按 `Ctrl+C` 然后键入 `cont` 跳到下一个断点。
9. 键入 `repl` 并查询 `a`、`b` 和 `c` 的值。
10. 按 `Ctrl+C` 然后键入 `cont`。
11. 可看到“Program terminated”信息，表示程序已经结束。
12. 按 `Ctrl+D` 退出调试器。

## 9.4 Node Inspector

调试 Node.js 应用程序最为有用的工具可能就是 Node Inspector。这个第三方工具是由 dannycoates (Danny Coates) 所创建。要想使用 Node Inspector，机器上必须有一个 WebKit 浏览器，也就是说需要安装 Chrome 或者 Safari。Node Inspector 让我们可以使用 WebKit JavaScript 调试器来按步执行代码，它支持如下功能。

- 浏览应用程序的源代码。
- 使用终端来与应用程序交互。
- 添加、移除断点。
- 按步执行代码中的函数调用。
- 步入、步出函数。
- 设置观察表达式。
- 查看代码中不同点上的堆栈踪迹。
- 查看作用域变量。

只要计算机上安装有 WebKit 浏览器，就可从 npm 安装 Node Inspector：

```
npm install -g node-inspector
```

启动 Node Inspector 的过程分为两部分。首先，使 `--debug` 或者 `--debug-brk` 标志启动应

用程序，以便启用 JavaScript 调试器。如果使用 `--debug-brk`，那么 Node Inspector 会在应用程序的第一行位置放置一个断点。注意要想开始调试的话，就需要按 Play 来跳到你设置的第一个断点上。

```
node --debug-brk app.js
```

一旦应用程序运行于调试模式下，就必须另启一个 Node Inspector 的进程。在另外一个终端选项卡中，运行如下命令：

```
node-inspector
```

如果成功，会输出一行信息：

```
visit http://0.0.0.0:8080/debug?port=5858 to start debugging
```

现在可以通过打开 WebKit 浏览器浏览 `http://0.0.0.0:8080/debug?port=5858` 来开始应用程序的调试了。而后会有个应用程序的综合视图出现在我们面前！

可以通过单击左边的列中的行号来添加断点。如果使用 `--debug-brk` 启动 Node，那么脚本的执行会在这里暂停。而后可以通过单击右边列顶部的 Play 图标按步执行代码，脚本会停在下一个断点上。如果没有设置任何断点，那么代码将一直运行，所以要确认设置了一些断点！

Node Inspector 与 Node 自带的调试器相似，但更有交互性。使用我们在使用 Node.js 调试器时所用的同一个示例，我们现在可以将 `debugger` 语句从代码中移除了。通常不在代码中遗留 `debugger` 语句是个好主意，而如果使用 Node Inspector 的话就无需担心这个问题了（见程序清单 9.7）。

#### 程序清单 9.7 移除了断点的 JavaScript 作用域示例

```
var foo = function(){
 var a = 3, b = 5;
 var bar = function() {
 var b = 7, c = 11;
 a += b + c;
 }
 bar ();
};
foo();
```

通过下面这个命令，我们可以以调试方式启动这个脚本并在第一行中断：

```
node --debug-brk app.js
```

而后可通过下列命令启动 Node Inspector：

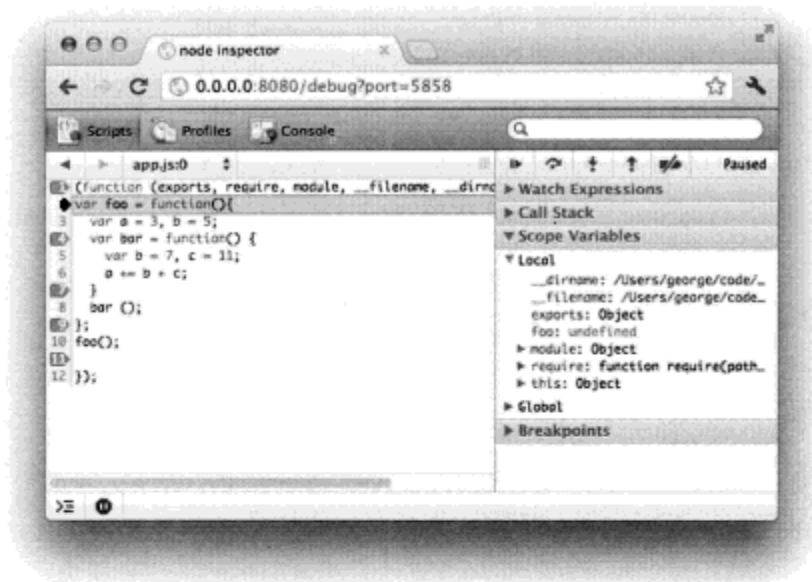
```
node-inspector
```

最后，打开浏览器访问 `http://0.0.0.0:8080/debug?port=5858` 显示 WebKit JavaScript 调试器（见图 9.1）。通过单击行号就可以添加以及移除断点。对于本示例而言，我们在第 4、7、9 和 11 行添加断点。



图 9.1

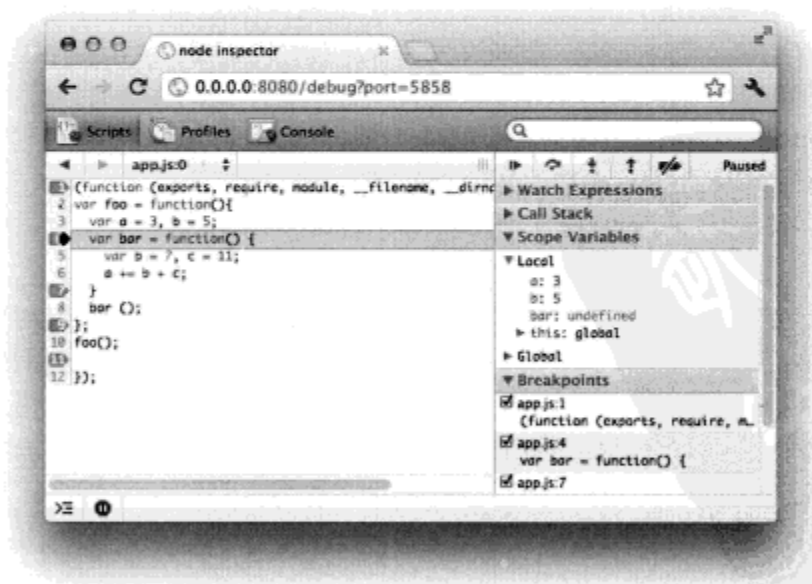
启动 Node Inspector



通过点击右手列上的 Play 按钮可按步执行断点。在按步执行时请注意右手列中信息的改变，这可让我们检查应用程序的当前状态（见图 9.2）。

图 9.2

按步执行断点



### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour09/example07` 找到。以下是使用 Node Inspector 进行调试的方法。

1. 将程序清单 9.7 中的代码复制到系统中名为 `app.js` 的文件中。
2. 运行脚本，启用 JavaScript 调试器并在第一行中断：

```
node --debug-brk app.js
```

3. 可看到 “debugger listening on port 5858.”。
4. 在另外一个终端选项卡或窗口中启动 Node Inspector:  
`node-inspector`
5. 可看到 “visit <http://0.0.0.0:8080/debug?port=5858> to start debugging.”。
6. 打开 WebKit 浏览器 (Chrome 或 Safari) 浏览 <http://0.0.0.0:8080/debug?port=5858>。
7. 可看到脚本。
8. 单击第 4、7、9 和 11 行添加断点。
9. 单击 Play 按钮按步执行断点。
10. 注意在右边列中 Scope Variables 下的变量值如何改变。

### Did you know?

#### 提示: Node Inspector 是学习的好方法

Node Inspector 的另外一个用处是学习代码在 Node.js 中如何工作。Node Inspector 让我们可以按步执行代码并查看代码所引用的文件和模块。通过使用 Node Inspector, 阅读代码就会变得有交互性, 而不只是在文本编辑器中阅读。

## 9.5 关于测试的注释

如何使用调试器依赖于你的开发风格。有些开发人员喜欢有机地构建应用程序, 这意味着他们编写代码并持续刷新页面或者重复期望中的输入直到对其工作结果感到满意。对于这种开发风格而言, Node Inspector 有用, 因为它给我们一个能够直入应用程序心脏的视图。不过, 大多数职业开发人员遵循的是测试驱动的开发过程。这涉及编写能一遍又一遍地运行的、用于确保应用程序按期望的方式工作的自动化测试。如果是这样, 那么就可使用调试器来解决难题和边界案例。

我们鼓励读者给应用程序编写测试而不是用调试器来帮助编写新代码。我们将在第 10 章学习测试以及测试如何帮助开发。不过, 有时候, 当我们在编写需要用来检查数据或检查函数返回的是什么的代码时, 使用调试器来查看通常要比编写测试让代码通过来得快。什么时候编写测试、什么时候使用调试器, 请读者自己决定, 不过, 只要可能就请使用测试。

## 9.6 小结

在本章我们学习了调试以及在 Node.js 中可用于调试应用程序的一些工具。我们学习了 STDIO 模块, 这是一种在脚本中加入调试的快速、简易的方法。而后我们了解了如何访问 V8 调试器并使用断点停止代码的执行并检查变量的值。最后, 我们介绍了 Node Inspector, 这是一个第三方的模块, 提供了对 WebKit JavaScript 调试器的访问。

## 9.7 问与答

问: 我应该使用哪个调试器, 在什么时候使用?

**答：**通常如果问题看起来不大而且容易调查明白，使用 **STDIO** 模块是最快的解决问题的方法。许多开发人员喜欢 **Node Inspector** 所提供的对代码执行时的交互性视图并且只使用 **Node Inspector**。对于开发人员来说，调试是个人的事情，所以要试遍所有的工具并试着理解它们的优势和弱点。调试是我们大量进行的事情，所以值得投入一些时间来理解你的选择。

**问：**我什么时候应该使用调试器，什么时候编写测试？

**答：**开发 **Node.js** 应用程序没有对或者错的方法，只是为应用程序构建一套测试会让应用程序从长远的意义上更稳定。如果创建新软件，那么我们鼓励你编写测试然后让测试通过。我们将在第 10 章学习更多这方面的知识。调试工具应当用于调查问题，而且，如果可能的话，要编写一个测试来验证 bug 被碾碎。

**问：**我如何记录 bug？

**答：**有许多 bug 记录器可用。**GitHub Issues** 就是在许多开发人员中流行的选择，因为它集成了源代码控制（如果使用 **Git** 的话）并且提供简单的基于 **Web** 的界面来管理 bug 和问题。

## 9.8 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

### 9.8.1 问题

1. 在软件开发中，为什么有 bug 这一说？
2. 对于不再成为问题的 bug，我们如何称呼？
3. bug 的最佳类型是什么？

### 9.8.2 答案

1. 许多人认为“bug”一词的发明者是 **Grace Hopper**，她是位计算机科学的先锋。在调查一台计算机原型的问题时，据说她发现在继电器中有个飞蛾陷在里面，于是发明了“bug”这个词。

2. 不再成为问题的 bug 可以说是“修正了的”、“关闭的”或者“碾碎了的”。许多问题记录器提供对 bug 的状态进行标记的支持，并计算还有多少开放的 bug（尚未解决的 bug）。

3. 在计算机学科中只有一种好的 bug 类型：已死的 bug！

## 9.9 练习

1. 如果读者开发过 **Node.js** 应用程序，请尝试在本章所描述的所有 3 种调试工具来检查你的应用程序。思考每种调试工具的优势和弱点。你选择哪种工具，为什么？

2. 使用 **Node.js** 主页上的 **Hello World** 服务器，用 `--debug-brk` 启动，然后启动 **Node**

Inspector。探索 JavaScript 调试器并设置一些断点。探究文件清单并查看哪些文件包含在内。设置一些断点并尝试对脚本如何工作以及 Node Inspector 如何让你访问许多关于脚本如何工作的细节有所了解。

3. 探索 GitHub Issues 页面上位于 <http://github.com/joyent/node/issues> 上的 Node.js 存放处。检查 bug 和问题报告如何被送进来。尝试理解：有些报告因为带有示例所以很不错；而其他一些则没有太多信息，对开发人员来说难以调查。这可帮助你在将来编制更好的 bug 报告文件！



## 第 10 章

# 测试 Node.js 应用程序

---

在本章中你将学到：

- 为什么测试是重要的；
- 使用 assert 模块进行测试；
- 使用 Nodeunit 模块进行测试；
- 使用 Vows 模块进行测试；
- 使用 Mocha 模块进行测试。

### 10.1 为什么测试

在前一章中，我们学习了如何调试 Node.js 应用程序，读者可能会问，既然有了这么强大的调试工具，为什么还需要编写测试呢。给应用程序编写测试是许多开发人员不做的事情。如果代码库小而且项目上只有一个开发人员的话倒也无妨。但随着应用程序的改变，变得越来越复杂，要确保代码的稳定性越来越困难。测试是可重复的代码片段，可对应用程序是否以期望的方式进行工作做出断言（assert）。比如，我们可能编写的测试如下所示。

- 某个 HTTP 响应是否返回 200 代码？
- 某个方法是否返回特定的值？
- 某个方法是否返回一个字符串？某个方法是否接收两个参数、执行计算然后返回正确的数？

具备可重复的测试让开发人员可以确认：当他们对代码做出修改之后，不会给应用程序引入新的 bug 或错误。许多开发人员也使用测试来驱动应用程序的创建，这个过程称为测试驱动开发（TDD）。在这个过程中，开发人员首先编写描述了他们希望应用程序如何工作的

测试，然后编写代码让测试通过。这样的一套测试可帮助驱动开发，也有助于应用程序的长期稳定性，因为在任何时刻开发人员都可运行测试并确认一切都工作如常。我们强烈鼓励编写测试，它有许多优点，如下所示。

- 这是质量保证的简单一层。
- 它让开发人员得以确保应用程序按期待的方式动作。
- 它支持开发人员创建新功能时不破坏现有功能。
- 它可集成到第三方代码质量和测试工具中。
- 它让开发人员团队得以保持对应用程序如何工作的一致理解。
- 它极大地增强了代码稳定性。
- 它让开发人员可以重构代码库并确保它仍旧按期待的方式工作。

## 10.2 Assert（断言）模块

Node.js 在 `assert` 模块中提供了一个简单的测试框架。它提供一组断言方法，让开发人员得以为 Node.js 应用程序创建低级测试。以下断言可在 Node.js 的断言模块中使用。

- 断言一个值是真的。
- 使用 JavaScript 的 `==` 等式运算符断言两个值相等。
- 使用 JavaScript 的 `!=` 等式运算符断言两个值不相等。
- 断言两个值深度相等（`deeply equal`）。
- 断言两个值深度不相等（`deeply not equal`）。
- 使用 JavaScript 的 `===` 等式运算符断言两个值严格相等。
- 使用 JavaScript 的 `!==` 等式运算符断言两个值严格不相等。
- 断言有个错误抛出。
- 断言有个错误没有抛出。
- 断言错误在回调中抛出。

对于深度相等断言，在 CommonJS 规范中的定义如下。

- 所有同等的值都相等，如 `===` 所确定的那样。
- 如果期望的值是个 `Date` 对象，那么实际的值如果也是个 `Date` 对象并且指向同一时间的话就是相等的。
- 对于其他的对，如果不能都通过值类型 `== "object"`（`typeof value == "object"`），则等同与否由 `===` 决定。
- 对于所有其他的对象对，包括数据对象在内，是否等同由是否有相同数量的属性（如使用 `Object.prototype.hasOwnProperty.call` 所校验的）、相同的键值集合（但无需有相同的顺序）、每个相关键值是否等同以及有一样的“原型”属性决定。注意：对于数组，这既适用于命名的属性也适用于索引的属性。

在 JavaScript 中, 比较相等与否有两种方法, 理解这一点很重要。第一种方法是使用 `==` 运算符。这在进行比较的时候不严格, 于是就会有将字符串与数字做比较并得到 `true` 结果的问题。如果不熟悉 JavaScript, 这可能在某些场合造成意料之外的结果:

```
"8" == 8 // true
'' == '0' // false
0 == '' // true
```

第二种比较运算符是 `===`。它检查值是否是相同的值以及是否是相同类型。所以如果想将字符串与数值进行比较的话, 会返回 `false`。许多有经验的 JavaScript 开发人员建议只使用这个比较运算符。

```
"8" === 8 // false
'' === '0' // false
0 === '' // false
```

为了使用 Node.js 的 `assert` 模块, 需要在脚本中请求它。

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

而后可使用 `assert` 模块, 使用这个模块提供的断言方法来比较不同的值。`assert.strictEqual()` 方法让我们可以使用 JavaScript 比较运算符来比较两个值, 我们建议读者使用这一方法。

```
assert.strictEqual("hello", "hello");
```

为了运行测试, 我们使用和运行任何正常的 Node 程序一样的方法运行文件。所以如果将文件保存成 `test.js`, 可按如下方法运行测试:

```
node test.js
```

如果测试通过, 则在终端上看不到任何输出。除非有问题存在, 否则 `assert` 模块会有意保持沉默。

---

## TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour10/example01` 找到。按照下列步骤使用 `assert` 模块比较值。

1. 创建一个名为 `test.js` 的文件并将下列代码复制到该文件中:

```
var assert = require('assert');
assert.strictEqual("hello", "hello");
```

2. 运行脚本:

```
node test.js
```

3. 在终端上应该看不到任何输出, 这表示测试通过。
- 

但是, 如果有测试失败, 那么就会看到异常抛出。在下面的示例中, 两个值不一样:

```
assert.strictEqual("hello", "there");
```

如果运行这个测试, 将会看到一个异常抛出:

```
node.js:201
 throw e; // process.nextTick error, or 'error' event on first tick
 ^
AssertionError: "hello" == "there"
```

默认情况下, `AssertError` 显示的是失败的比较, 不过, 也可以传递第三个参数, 它会在

异常抛出的时候显示出来：

```
assert.equal("hello", "there", "Message to show if an exception is thrown");
```

现在如果运行测试的话，它显示定制的消息：

```
node.js:201
 throw e; // process.nextTick error, or 'error' event on first tick
 ^
AssertionError: Message to show if an exception is thrown
```

可使用它在脚本中记录影响到断言值的其他值。

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour10/example02` 找到。按照下述步骤来演示错误被抛出。

1. 创建一个名为 `test.js` 的文件并将下列代码复制到文件中：

```
var assert = require('assert');
assert.strictEqual("hello", "there");
```

2. 运行脚本：

```
node test.js
```

3. 可在终端上看到有异常抛出。
- 

为了避免被 JavaScript 的比较特质绊倒而造成危险，我们给出之前在 Node 的 `assert` 模块中使用 `assert.equal()` 和 `assert.strictEqual()` 的示例。建议为了避免出现问题，默认情况下应使用 `assert.strictEqual()`。

```
assert.equal("8", 8) // true
assert.equal('', '0') // false
assert.equal(0, '') // true
assert.strictEqual("8", 8) // false
assert.strictEqual('', '0') // false
assert.strictEqual(0, '') // false
```

## 10.3 第三方测试工具

在测试任务中，Node 的 `assert` 模块的功能相当强大，但它是低级的，有许多限制。第三方开发人员创建了许多测试框架，在 `assert` 模块之上添加功能，包括：

- 能够处理测试的运行并报告结果的测试运行器；
- 将不同测试组成组，以便能独立运行；
- 建立与拆卸测试，这样的话开发人员就可编写代码来准备应用程序状态以便测试，也可随之将这个状态清除；
- 在浏览器中运行测试的能力；
- 加入测试报告器，以便测试结果可输出成 HTML、XML 和其他格式；
- 对异步测试的更好支持。



### 10.3.1 Nodeunit

Nodeunit 构建于 Node 的 `assert` 模块之上，添加了建立与拆卸测试的能力、异步测试能力以及模拟（mock）和桩（stub）的功能。Nodeunit 可通过 npm 使用全局标记全局地安装到计算机上。

```
npm install -g nodeunit
```

还可以使用 `package.json` 文件来安装 Nodeunit。测试的模块通常在 `devDependencies` 中声明（见程序清单 10.1）。

程序清单 10.1 带有 Nodeunit 的 `package.json` 文件

```
{
 "name": "nodeunit_example",
 "version": "0.0.0",
 "private": true,
 "devDependencies": {
 "nodeunit": "0.7.4"
 }
}
```

Nodeunit 中的断言与原生的 `assert` 模块紧密映射，而测试的声明如程序清单 10.2 所示。

程序清单 10.2 使用 Nodeunit 的断言

```
exports.firstTest = function(test){
 test.expect(1);
 test.strictEqual("hello", "hello");
 test.done();
};
exports.secondTest = function(test){
 test.expect(1);
 test.strictEqual("hello", "there");
 test.done();
};
```

每个测试都声明成 `exports.testName`，这里的 `testName` 是测试的描述。在测试的开始处声明了 `test.expect(n)`，这里的 `n` 是期望的断言数。这用于避免通过假测试。在完成了测试之后，一定要调用 `test.done()` 来表示测试完成。

如果已经调用了 `test.js` 文件并全局安装了 Nodeunit，就可用如下命令来运行测试：

```
nodeunit test.js
```

如果选择使用 `package.js` 文件安装 Nodeunit，就需要在项目文件夹中直接引用二进制文件。

```
./node_modules/nodeunit/bin/nodeunit test.js
```

前面的示例可看到如下输出（见图 10.1）。

```
test.js
✓ firstTest
✗ secondTest
```

```
AssertionError: 'hello' === 'there'
..stacktrace here..
```

图 10.1

使用 Nodeunit 运  
行测试



## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour10/example03` 找到。以下是使用 Nodeunit 进行测试的方法。

1. 将程序清单 10.1 中的代码复制到系统上名为 `package.json` 的文件中。
2. 将程序清单 10.2 中的代码复制到系统上名为 `test.js` 的文件中。
3. 运行如下命令安装依赖模块：

```
npm install
```

4. 运行测试：

```
./node_modules/nodeunit/bin/nodeunit test.js
```

5. 可看到一个测试通过而另一个测试失败，并带有失败测试的堆栈踪迹。

Nodeunit 也支持测试异步代码。在以下示例中，使用了 Node 的 `fs` 模块从文件系统中读入文件并假设在与测试所在的相同文件夹下有一个名为 `test.txt` 的带有一些文本内容的文件。当回调被调用时，会运行两个测试以测试是否有错误抛出以及文件大小是否为 0。注意，测试都位于回调函数中（见程序清单 10.3）。

### 程序清单 10.3 使用 Nodeunit 的异步测试

```
var fs = require('fs');
exports.asyncTest = function(test){
 fs.stat('test.txt', function(err, stats) {
 test.expect(2);
 test.strictEqual(err, null);
 test.notStrictEqual(stats.size, 0);
 test.done();
 })
};
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour10/example04` 找到。按照下列步骤

使用 Nodeunit 测试回调。

1. 将程序清单 10.1 中的代码复制到系统中名为 `package.json` 的文件中。
2. 将程序清单 10.3 中的代码复制到系统中名为 `test.js` 的文件中。
3. 运行如下命令安装依赖模块：  

```
npm install
```
4. 创建一个用于在测试中使用的测试文件，命名为 `test.txt`，然后添加如下内容：  

```
This is a test file
```
5. 运行测试：  

```
./node_modules/nodeunit/bin/nodeunit test.js
```
6. 可看到测试通过。

## 10.4 行为驱动的开发 (Behavior Driven Development)

许多开发人员喜欢使用行为驱动的开发。行为驱动的开发从外部往内考虑应用程序，而不是从内部测试程序的工作情况。请比较如下两种描述。

当用户注册时，应用程序应当接收 POST 请求，检查字段是否有效，然后将数据库中的用户数量递增 1。

作为一个用户，当我成功注册时，我应该看到“感谢注册”。

第一个示例更接近于测试驱动的开发，开发人员以应用程序应该有的功能为出发点为应用程序代码编写测试。第二个示例描述用户如何与应用程序交互以及他们期望看到的是什么。目前，Node.js 社区中的趋势是使用 BDD（行为驱动的开发）而不是测试驱动的开发（TDD）。这两种都是有效的方法，但使用 BDD 可能更容易让相关人员介入到测试中。在 BDD 中，测试是以应用程序与来自外界的交互为基础，而不需要理解应用程序的内部原理。通常，在编写 BDD 风格的测试时，要使用更为平白的英语并描述正在测试的是什么，然后声明其如何工作。在 Node.js 社区中有许多流行的库可用于 BDD 测试方法。

### 10.4.1 Vows

Vows 是一个第三方模块，用于以 BDD 风格测试 Node.js 应用程序。可使用 npm 全局安装 Vows：

```
npm install -g vows
```

也可通过在项目的 `package.json` 文件的 `devDependencies` 一节中加入 Vows 来安装它（见程序清单 10.4）。

#### 程序清单 10.4 带有 Vows 的 `package.json` 文件

```
{
 "name": "vows_example",
 "version": "0.0.0",
 "devDependencies": {
 "vows": "0.6.2"
 }
}
```

Vows 也是建立在 assert 模块之上的，它加入了 BDD 风格的测试。我们在本章中使用过的同步测试示例，用 Vows 来做的话会是这样（见程序清单 10.5）。

#### 程序清单 10.5 使用 Vows 的断言

---

```
var vows = require('vows'),
 assert = require('assert');

vows.describe('Comparing strings').addBatch({
 'when comparing the same strings': {
 topic: "hello",
 'they should be equal': function (topic) {
 assert.strictEqual (topic, "hello");
 }
 },
 'when comparing different strings': {
 topic: "hello",
 'they should not be equal': function (topic) {
 assert.notStrictEqual (topic, "there");
 }
 }
}).run();
```

注意，Vows 鼓励用户在测试中使用大量平白英语。这不仅可帮助用户从外往内思考应用程序，也有助于测试报告的编写、测试失败的处理。Vows 以如下方式用平白英语描述功能。

- Description（描述）——测试套组的描述。
- Context（上下文）——测试运行的上下文。
- Topic（主题）——要测试的是什么。
- Vow（宣告）——期望在测试中发生的是什么。

以下是使用这些术语做注解的同一个测试（见程序清单 10.6）。

#### 程序清单 10.6 使用 Vows 注解的断言

---

```
var vows = require('vows'),
 assert = require('assert');

vows.describe('Comparing strings').addBatch({ // Description
 'when comparing the same strings': { // Context
 topic: "hello", // Topic
 'they should be equal': function (topic) { // Vow
 assert.strictEqual (topic, "hello");
 }
 },
 'when comparing different strings': { // Context
 topic: "hello", // Topic
 'they should not be equal': function (topic) { // Vow
 assert.notStrictEqual (topic, "there");
 }
 }
}).run();
```

如果安装了 Vows 并且将这些示例测试保存在名为 `test.js` 的文件中, 那么就可以从终端中使用 `node` 命令运行这些测试:

```
node test.js
```

如果测试通过, 输出将列出得到荣耀 (honored) 的宣告数量。这等同于测试通过。

```
.. ✓ OK » 2 honored
```

如果测试失败, 报告中将包括上下文、宣告以及异常的第一行 (见图 10.2)。这在调试的时候会有用处。

```
X·
```

```
when comparing the same strings
 X they should be equal
 » expected 'hello',
 got 'goodbye' (===) // test.js:8
X Broken » 1 honored · 1 broken
```



图 10.2

使用 Vows 运行测试

## TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour10/example05` 找到。按照下列步骤使用 Vows 进行测试。

1. 将程序清单 10.4 中的代码复制到系统中名为 `package.json` 的文件中。
2. 将程序清单 10.5 中的代码复制到系统中名为 `test.js` 的文件中。
3. 运行如下命令安装依赖模块:

```
npm install
```

4. 运行测试:

```
node test.js
```

5. 可看到测试通过:

```
.. ✓ OK » 2 honored
```

至于异步代码的测试, Vows 中的主题就变成了要执行测试的异步函数 (见程序清单 10.7)。

## 程序清单 10.7 使用 Vows 的异步测试

```

var vows = require('vows'),
 assert = require('assert'),
 fs = require('fs');

vows.describe('Async testing').addBatch({
 'When using fs.stat on a file': {
 topic: function () {
 fs.stat('test.txt', this.callback);
 },
 'it should be present': function (err, stat) {
 assert.strictEqual(err, null);
 },
 'it should not be empty': function (err, stat) {
 assert.notStrictEqual(stat.size, 0);
 }
 },
}).run();

```

这里要指出的关键点是，回调是特殊的 `this.callback` 函数。这在所有 Vows 主题中都可用，它使回调的结果可以被传递到测试函数中。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour10/example06` 找到。按照下列步骤使用 Vows 测试异步代码。

1. 将程序清单 10.4 中的代码复制到系统中名为 `package.json` 的文件中。
2. 将程序清单 10.7 中的代码复制到系统中名为 `test.js` 的文件中。
3. 创建一个名为 `test.txt` 的测试文件用于测试，并在其中加入如下内容：

```
This is a test file
```

4. 运行如下命令安装依赖模块：

```
npm install
```

5. 运行测试：

```
node test.js
```

6. 可看到测试通过：

```
.. ✓ OK > 2 honored
```

## 10.4.2 Mocha

Mocha 是另外一个第三方模块，用于使用模块化方法测试 Node.js 应用程序。可以使用一系列的库文件，包括 Node 的 `assert` 和许多第三方模块。此外还支持大量报表格式，Mocha 的设计旨在给开发人员尽可能多的选择。

通过在项目的 `package.json` 文件的 `devDependencies` 节中加入 Mocha，然后使用 `npm install` 命令即可安装 Mocha。

```

"devDependencies": {
 "mocha": "0.10.1"
}

```

为了运行 Mocha 测试, 必须更新 `package.json` 文件 (见程序清单 10.8) 并让 `npm` 知道如何运行测试。默认情况下, 这会运行任何位于 `test/` 文件夹中的东西。对于本示例而言, 测试位于名为 `test.js` 的单个文件中。

#### 程序清单 10.8 带有 Mocha 的 `package.json` 文件

```
{
 "name": "mocha_example",
 "version": "0.0.0",
 "devDependencies": {
 "mocha": "0.10.1"
 },
 "scripts": {
 "test": "./node_modules/.bin/mocha test.js"
 }
}
```

在更新了 `package.json` 文件之后, 就可以使用如下命令从终端运行测试:

```
npm test
```

对于同步测试的示例, 在 Mocha 中看起来是这样的 (程序见清单 10.9)。

#### 程序清单 10.9 使用 Mocha 的断言

```
var assert = require('assert');
describe('Comparing strings', function() {
 describe('when comparing the same strings', function() {
 it('should return true', function() {
 assert.strictEqual("hello", "hello");
 })
 })
 describe('when comparing different strings', function() {
 it('should return false', function() {
 assert.notStrictEqual("hello", "there");
 })
 })
})
```

注意就如 Vows 模块那样, Mocha 鼓励用户使用平白的英语以 BDD 风格来描述应用程序。在这个示例中, Mocha 扮演 Node 的 `assert` 模块的简单包装器, 提供 BDD 风格的语法。在运行的时候, Mocha 提供许多报表格式, 其中默认的是 `dots` (见图 10.3)。



图 10.3

使用 Mocha 运行  
测试

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour10/example07` 找到。按照下列步骤使用 Mocha 进行测试。

1. 将程序清单 10.8 中的代码复制到系统中名为 `package.json` 的文件中。
2. 将程序清单 10.9 中的代码复制到系统中名为 `test.js` 的文件中。
3. 安装依赖模块：

```
npm install
```

4. 运行测试：

```
npm test
```

5. 应该看到测试通过：

```
./node_modules/.bin/mocha test.js
```

```
..
```

```
✓ 2 tests complete
```

如果测试失败，Mocha 的输出中也使用了平白英语的描述。在调试的时候这有助于快速了解测试失败处的上下文。

```
✗ 1 of 2 tests failed:
```

```
0) Comparing strings when comparing the same strings should return true:
```

```
AssertionError: "hello" === "goodbye"
```

至于异步测试，Mocha 和 Vows 一样，通过使用 `done()` 回调的方法来表示测试完成（见程序清单 10.10）。

### 程序清单 10.10 使用 Mocha 的异步测试

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

describe('Async testing', function(){
 describe('When using fs.stat on a file', function(){
 it('should not be empty', function(done){
 fs.stat('test.txt', function (err, stat){
 assert.notEqual(stat.size, 0);
 done();
 });
 });
 });
});
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour10/example08` 找到。按照下列步骤使用 Mocha 测试异步代码。

1. 将程序清单 10.8 中的代码复制到系统中名为 `package.json` 的文件中。



2. 将程序清单 10.10 中的代码复制到系统中名为 `test.js` 的文件中。
3. 创建一个名为 `test.txt` 的测试文件用于测试，并在其中加入如下内容：

```
This is a test file
```

4. 运行如下命令安装依赖模块：

```
npm install
```

5. 运行测试：

```
npm test
```

6. 可看到测试通过：

```
./node_modules/.bin/mocha test.js
```

```
✓ 1 tests complete
```

## 10.5 小结

在本章，我们介绍了 Node.js 中的测试。我们了解了如何使用 `assert` 模块对代码运行简单测试的方法。而后我们介绍了一些第三方测试工具，它们既支持测试驱动的开发也支持行为驱动的开发。我们介绍了 Nodeunit、Vows 和 Mocha。我们也学习了使用所有这些模块运行异步代码的方法。读者现在对测试已经有了良好的基础，也就没有借口不测试了！

## 10.6 问与答

问：我真的需要测试代码吗？

答：是的！测试有巨大的益处。它改进代码的稳定性，让其他开发人员更容易介入项目，并让你得以满怀信心地进行代码重构并加入到代码库中。做吧！

问：我该测试什么？

答：应该测试的是与 Node 的核心或者第三方模块无关的代码。那些代码通常已经有了它们自己的测试，所以重新测试它们就不合理了。随着你开发的代码越来越多，你将不断地思考“我是否需要测试这个？”如果你心里有疑问，那就为它写个测试吧。

问：我应该用哪个测试模块？

答：在 Node.js 的生态系统中有许多测试库，所以要选择哪一个就会让人难以决定。读者显然必须对 Node 的 `assert` 模块有良好的理解。你有可能觉得它可满足许多需求，而许多第三方测试工具也依赖于它。除此以外，你最喜欢哪个库、哪个库适合于你的编码风格，就用哪个吧。

问：我可否使用这些库来测试客户端的代码？

答：可以。使用浏览器化的模块 (<https://github.com/substack/node-browserify>)，就可在浏览器中使用 `assert` 模块。在本书编写的时候，Nodeunit 和 Mocha 都可以在浏览器中使用了。

问：我想学更多关于测试的内容。我该从何开始？

答：只需给应用程序编写测试并理解测试库，就是很好的开始。这个主题方面有一本很好的书：Christian Johansen 的《Test-Driven JavaScript Development》。虽然这本书并不是专门给 Node.js 写的，但它包含与开发 Node.js 应用程序相关的信息。关于本书的更多信息请访问 <http://tddjs.com/>。

## 10.7 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

### 10.7.1 问题

1. 为什么使用 JavaScript 的 `===` 相等运算符，而不是用 `==`？
2. 为什么第三方开发人员要在 `assert` 模块上创建其他测试模块？
3. 给代码编写测试有哪些优势？

### 10.7.2 答案

1. JavaScript 有两个相等运算符。第一个：`==`，执行类型强制转换并可导致期望之外的结果。第二个：`===`，检查值的相等也检查类型的相同。通常建议使用三个等号运算符。
2. `assert` 模块给 Node.js 提供了一个基础的测试框架。而在某些领域它有改进的余地，包括测试报告、测试异步代码以及使用 BDD 或者 TDD 风格的测试。
3. 编写测试可让我们确认代码能按期望的方式工作。它也让其他开发人员可理解代码如何工作，并提供一组可重复的测试，在更改或重构代码的时候可以运行。

## 10.8 练习

1. 通读 Node 的 `assert` 模块文档，地址在 <http://nodejs.org/docs/latest/api/assert.html>。使用这个模块可以写出许多好测试！花些时间来理解 JavaScript 的不同，比较方法之间的区别，以及它们与 `assert` 模块的关系。
2. 扩展示例 01，加入更多的能用上 `assert.equal()` 和 `assert.stringEqual()` 的测试。比较 `"0"` 和 `"false"` 以及 `"false"` 和 `"null"` 的区别，以此理解 JavaScript 处理它们的方式。
3. 探究 Github 上 Node.js 的测试文件夹以及如何使用 `assert` 模块测试自己，地址是 <https://github.com/joyent/node/blob/master/test/simple/test-assert.js>。如果不理解所有的测试也没关系，但至少试着理解。

## 第 11 章

# 部署 Node.js 应用程序

---

在本章中你将学到：

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

### 11.1 准备好部署

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

### 11.2 在云上托管

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

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

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

可使用 Internet 来访问数据。云计算有许多优点，如下所示。

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

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

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

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

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

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

## TRY IT YOURSELF

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

1. 下载代码示例并打开 `hour11/example01` 文件夹，可以看到一个简单的 Express 应用程序。
2. 为这个应用程序安装依赖模块：  
`npm install`
3. 打开浏览器并浏览 `http://127.0.0.1:3000`。
4. 可看到这个应用程序示例。这是我们要在本章部署的应用程序（见图 11.1）。

图 11.1

将要在本章部署  
的 Express 应用程  
序示例



## 11.3 Heroku

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

### 11.3.1 注册 Heroku

为了使用 Heroku 服务，必须先注册一个账号。请按如下步骤进行。

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

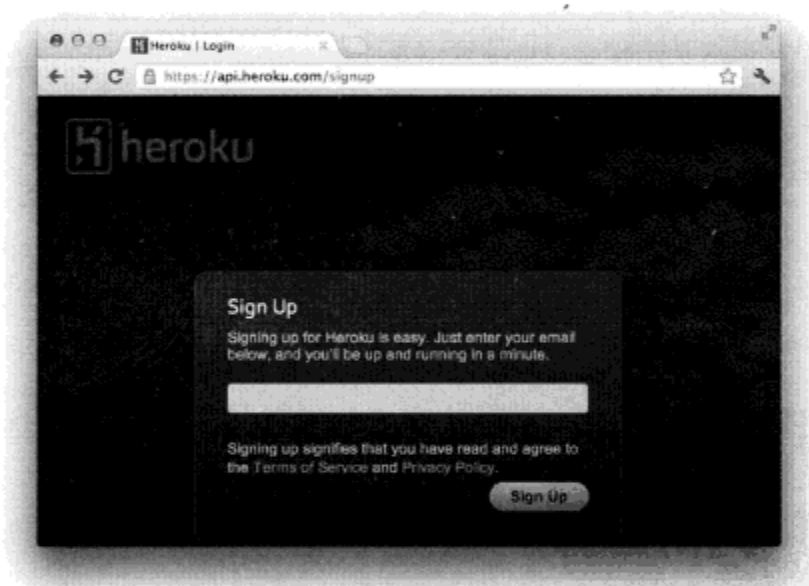


图 11.2

注册 Heroku

2. 一旦成功输入电子邮件地址，网站会邀请你检查邮件，邮件中有确认链接。
3. 打开电子邮件，进入所提供的 Heroku 链接，会邀请你选择密码。
4. 按照 [http://devcenter.heroku.com/articles/quickstart#step\\_2\\_install\\_the\\_heroku\\_toolbelt](http://devcenter.heroku.com/articles/quickstart#step_2_install_the_heroku_toolbelt) 中的指南安装 Heroku 工具条。它提供了能让我们将站点部署到 Heroku 的命令行工具。
5. 一旦完成了适用于自己平台的安装之后，最后要做的事情就是登录账号。为了完成这一过程，请从要求你输入证书的终端上运行 `heroku login`。注意，如果这是你第一次登录，会为你生成一个 SSH 公共密钥。Heroku 以此来管理对服务的访问。

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

搞定！你已经有了一个 Heroku 账号，可以准备部署了！

### By the Way

#### SSH 密钥是密码的替代

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

### TRY IT YOURSELF

按照如下步骤注册 Heroku 账号。

1. 在 <http://api.heroku.com/signup> 注册 Heroku 账号。
2. 访问邮件中的链接并选择密码。
3. 从 [http://devcenter.heroku.com/articles/quickstart#step\\_2\\_install\\_the\\_heroku\\_toolbelt](http://devcenter.heroku.com/articles/quickstart#step_2_install_the_heroku_toolbelt) 为你的平台安装安装程序。
4. 打开终端窗口并登录到 Heroku。

```
heroku login
```

## 11.3.2 为 Heroku 准备应用程序

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

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

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

然后将下列行

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

替换成

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

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

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

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

### TRY IT YOURSELF

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

1. 制作一个 `hour11/example01` 代码示例所提供的 Express 应用程序的副本。
2. 在 `app.js` 文件的顶部，添加如下行：

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

3. 将下列行从 `app.js` 中移除：

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

4. 用下列行替换它：

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

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

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

6. 用如下命令安装依赖模块：

```
npm install
```

7. 启动应用程序并检查其是否运行正常：

```
node app.js
```

### 11.3.3 将应用程序部署到 Heroku

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

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

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

```
node_modules
```

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

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

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

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

这会为我们创建站点并自动设置好需要部署的一切。我们会看到如下输出：

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

最后，可以使用如下命令部署应用程序：

```
git push heroku master
```

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

---

#### TRY IT YOURSELF

为了将 Node.js 应用程序部署到 Heroku，请按如下步骤进行。

1. 返回示例 Express 应用程序。
2. 使用如下命令创建 Git 库：

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

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

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

4. 用如下命令将站点发布到 Heroku:

```
git push heroku master
```

5. 访问之前得到的 URL, 可看到所部署的网站了!

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

```
git push heroku master
```

## 11.4 Cloud Foundry

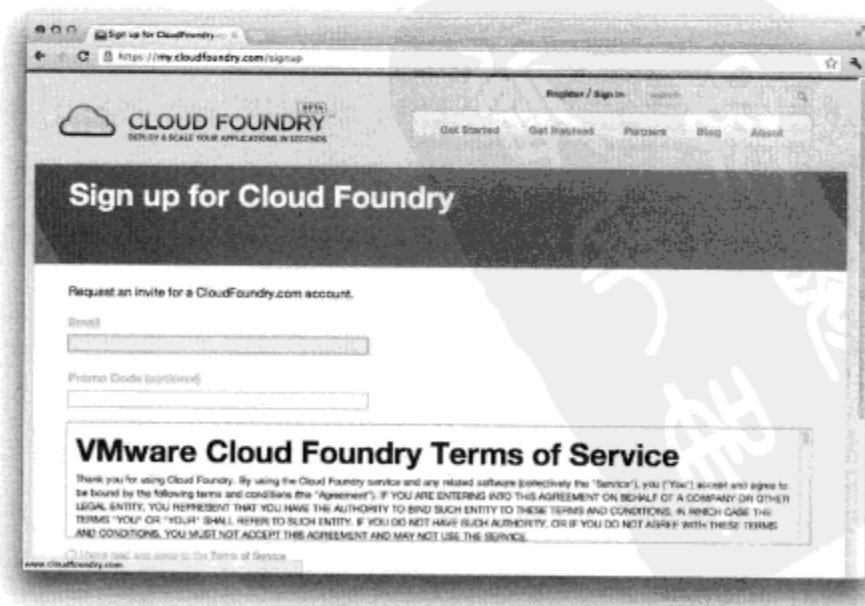
Cloud Foundry 是另外一个支持 Node.js 的 PaaS 提供商。Cloud Foundry 和其他服务的主要不同是它不依赖 Git, 所以如果读者不喜欢将 Git 作为部署过程的一部分, 它可能是个好选择。

### 11.4.1 注册 Cloud Foundry

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

图 11.3

注册 Cloud Foundry



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



[vmc/installing-vmc.html](#)。

### Cloud Foundry 是开源的

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

**Did you  
know?**

### TRY IT YOURSELF

为了创建 Cloud Foundry 账户，请按如下步骤进行。

1. 在 <https://my.cloudfoundry.com/signup> 注册一个 Cloud Foundry 账户。
2. 访问邮件中的链接并添加密码。
3. 按照 <http://start.cloudfoundry.com/tools/vmc/installing-vmc.html> 中的指南安装 vmc 及相关软件。

## 11.4.2 为 Cloud Foundry 准备应用程序

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

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

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

接下来，将内容为

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

的行修改为：

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

### TRY IT YOURSELF

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

1. 制作一个 [hour11/example01](#) 代码示例所提供的 Express 应用程序的副本。
2. 在 app.js 文件的顶部，添加如下行：

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

3. 将下列行从 app.js 中移除：

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

4. 用下列行替换它：

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

5. 用如下命令安装依赖模块：

```
npm install
```

6. 启动应用程序并检查其是否运行正常:

```
node app.js
```

### 11.4.3 将应用程序部署到 Cloud Foundry

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

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

接下来, 必须添加登录证书:

```
vmc login
```

这会提示用户输入用户名和密码并告知是否成功:

```
Attempting login to [http://api.cloudfoundry.com]
```

```
Email: george@shapedshed.com
```

```
Password: *****
```

```
Successfully logged into [http://api.cloudfoundry.com]
```

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

```
npm install
```

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

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

要从项目的根目录部署站点, 运行如下命令:

```
vmc push
```

这会提示一组问题并告知应用是否成功部署:

```
Would you like to deploy from the current directory? [Yn]: y
```

```
Application Name: yourappname
```

```
Application Deployed URL [yourappname.cloudfoundry.com]:
```

```
Detected a Node.js Application, is this correct? [Yn]: Y
```

```
Memory Reservation (64M, 128M, 256M, 512M, 1G) [64M]:
```

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

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

```
vmc update yourappname
```

---

#### TRY IT YOURSELF

以下是将 Node.js 应用程序部署到 Cloud Foundry 的方法。

1. 返回示例 Express 应用程序。
2. 将 `vmc` 的目标设置为 Cloud Foundry 服务器:

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

3. 输入登录证书:

```
vmc login
```

4. 确保所有的依赖模块都已使用 `npm` 安装:

```
npm install
```

5. 用下列命令修补 vmc 的 bug:

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

6. 部署站点:

```
vmc push
```

7. 访问显示在输出中的 URL, 可看到所部署的网站了!

## 11.5 Nodester

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

### 11.5.1 注册 Nodester

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

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



图 11.4

从 Nodester 请求  
一张令牌

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

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

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

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

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

Linux)，那么可以使用如下命令生成一个密钥：

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

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

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

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

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

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

---

## TRY IT YOURSELF

要想注册 Nodester 账号，请执行如下步骤。

1. 从 <http://nodester.com> 请求一张 Nodester 券。
2. 收到券之后，如果需要的话，生成一个 RSA 密钥：

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

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

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

4. 确认在本地计算机上安装了 Nodester CLI：

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

5. 为 Nodester 设置本地计算机：

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

## 11.5.2 为 Nodester 准备应用程序

要在 Nodester 上创建新的应用程序，请运行如下命令：

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

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

```
nodester app init yourapp
```

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

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

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

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

删除有下列内容的行：

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

将其替换为：

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

### 11.5.3 将应用程序部署到 Nodester

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

```
nodester npm install
```

这个命令会读取 `package.json` 文件并安装任何包。

最后，可以将文件添加到 Git 并推送发布站点！

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

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

---

#### TRY IT YOURSELF

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

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

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

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

```
nodester app init yourapp
```

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

4. 在 `app.js` 文件的顶部，添加如下一行：

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

5. 将下行从 `app.js` 中移除：

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

6. 用如下内容替换它：

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

7. 用下列命令安装依赖模块：

```
nodester npm install
```

## 8. 将文件添加到 Git 库中:

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

## 9. 推送文件以便发布站点:

```
git push origin master
```

## 10. 访问显示在输出中的 URL, 可看到所部署的网站了!

## 11.6 其他 PaaS 提供商

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

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

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

## 11.7 小结

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

## 11.8 问与答

问: 我应该使用哪个 PaaS 提供商?

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

问: 我应该使用哪个数据库呢?

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

问：我应该使用哪个 WebSocket 呢？

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

问：我可否在自己的服务器上托管 Node.js 站点？

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

## 11.9 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

### 11.9.1 测验

1. PaaS 代表什么？
2. 本章讨论的 PaaS 提供商中哪些是开源的？
3. 什么时候应该使用你自己的托管服务而不使用 PaaS？

### 11.9.2 答案

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

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

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

## 11.10 练习

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

发布更改。

2. 在 Internet 上对 Node.js 托管提供商做一些研究。试着理解针对每个提供商的发布过程并留意使用其服务的费用。

3. 如果编写了一个 Node.js 应用程序，无论它多么小，请选择一个 Node.js PaaS 提供商并发布你的站点。





## 第4部分 使用 Node.js 的中间站点

第12章 介绍 Socket.IO

第13章 一个 Socket.IO 聊天服务器

第14章 一个流 Twitter 客户端

第15章 JSON API



## 第 12 章

# 介绍 Socket.IO

---

在本章中你将学到：

- SocketIO 是什么以及如何使用它；
- 将客户连接到一个启用了 Socket.IO 的 Node.js 服务器；
- 从服务器发送数据到客户；
- 创建实时应用程序。

### 12.1 现在要开始学习一些完全不同的技术了

到目前为止，我们学习了使用 Node.js 创建基础应用程序的方法。Socket.IO 是 Node.js 的一个模块，它展现出一些让 Node.js 与其他框架和语言不同的地方。在本章，我们将看到 Node.js、Socket.IO 和 WebSocket 如何实现实时 Web 应用程序的创建——这是 Web 开发非常让人兴奋的地方！

### 12.2 动态 Web 简史

从历史上说，Web 在设计上并没有考虑动态。它是围绕着文档设计的，可以在一个时刻阅读单一的文档。用户从浏览器请求一个 Web 页面，获得响应，然后浏览器显示页面——这样的循环是许多为 Web 提供力量的技术的设计基础。随着时间的推移，开发人员想做更多的事情，而不只是显示文档。而 JavaScript 一直都处于开发人员推动 Web 页面功能发展的中心。

在 20 世纪 90 年代后期和本世纪初，DHTML（动态超文本标记语言）这个术语出现了。它描述的是使用 JavaScript、HTML 和 CSS 更改页面的某些部分的能力，从而让 Web 页面更有交互性，或者说更为动态。它引入了诸如 ticker、显示页面的隐藏部分以及简单的

动画等内容。DHTML 原本是个行为术语，它讲解的是用户与 Web 页面交互的方法以及当交互发生时会发生什么。Web 标准社区后来采用 DOM（文档对象模型）脚本，以可响应的方式来描述对 Web 页面的操纵，这样当浏览器中的某些插件不可用时，浏览器可以优雅地降级。

Ajax（异步 JavaScript 和 XML）是动态 Web 页面的下一个巨大进展，它和 DHTML 一样，描述了一组包括 JavaScript、XML、HTML 和 CSS 在内的技术。Ajax 让开发人员可以无需刷新 Web 页面就能从服务器请求数据。这意味着可以在诸如用户单击按钮这样的特定事件发生的时候，从服务器请求新的数据。由于整个 Web 页面无需刷新，就带来了更具有交互性、动态的体验。

使用实时技术创建的应用程序中最为有名的可能当属 Gmail，这是 Google 的基于 Web 的电子邮件服务。它使用一种称为 Comet 的技术，在用户阅读电子邮件的时候刷新页面各个部分的全部，用新数据替换它而无需重新装载页面。这是第一款在多个浏览器中支持实时流（real-time streaming）的应用程序，真正挑战了浏览器在技术上的可能。今天，Ajax 和相关技术继续在 Web 上被大量使用，动态地将数据拉入 Web 页面并加入动态功能。示例如下所示。

- 可嵌入的、展示来自其他网站和服务的构件。
- 基于 Web 的聊天客户端（比如 Campfire）。
- 与服务器验证数据而无需刷新页面。
- Web 上的游戏者之间的简单游戏。

毋庸置疑，将来在创建动态 Web 页面上，Ajax 将继续发挥极大作用；但在某些领域，它却极为不足。如果从服务器请求数据，Ajax 工作得很好；但如果服务器想将数据推送到浏览器呢？Ajax 技术无法很容易地支持将数据推送到客户，虽然诸如 Comet 这样的一些技术已经让其成为可能。使用 Ajax 技术有可能实现双向数据流，但需要跨过许多障碍才行，而且在不同的浏览器上的工作方式也不同。

WebSocket 是对在服务器和客户端之间实现双向实时通信问题的响应。它的思想是，从头开始，设计一个开发人员可以使用的标准以便以一致的方式创建应用程序；而不是通过复杂的、并不总能在所有浏览器中都工作的设置来使用现有技术完成这一任务。WebSocket 的基本思想是在 Web 服务器和浏览器之间保持连接持久打开。这就使得不管是服务器还是浏览器都可以在想要的时候推送数据。由于连接是持久的，所以数据的交换就非常快，也就成就了“实时”这个术语。WebSocket 不支持诸如重新连接处理（reconnection handling）或者心跳（heartbeat）这样的功能，但诸如 Socket.IO 这样的库则提供了这些功能，同时还对某些跨浏览器的问题做了抽象。

一开始，会有许多东西需要学习。不过在本章中，我们会看到 Socket.IO 和 WebSocket 实际的示例，它们可以澄清这些概念。

## 12.3 Socket.IO

Socket.IO 是 Node.js 的一个模块，它提供通过 WebSocket 进行通信的一种简单方式。WebSocket 协议很复杂，从头开始编写一个支持 WebSocket 的应用程序将需花费很多时间。Socket.IO 提供服务器和客户端双方的组件，所以只需一个模块就可给应用程序加入对

WebSocket 的支持。Socket.IO 也解决浏览器支持的问题（不是所有浏览器都支持 WebSocket）并让实时通信可以跨几乎所有常用的浏览器实现。Socket.IO 的设计非常好，将实时通信带入应用程序的过程变得非常简单。如果想做任何涉及在 Web 服务器和浏览器之间通信的事情，那么 Node.js 和 Socket.IO 是极佳的选择！

## 12.4 基础的 Socket.IO 示例

Socket.IO 既能在服务器也能在客户端工作。要使用它，必须将其添加到服务器端的 JavaScript (Node.js) 和客户端的 JavaScript (jQuery 等) 中。这是因为通信会是双向的，所以 Socket.IO 需要能在两边工作。在下面的示例中，我们首先学习构建服务器。在程序清单 12.1 中，使用 Node.js 来提供单独的一个 HTML 文件服务。

程序清单 12.1 使用 Node.js 提供一个文件的服务

---

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

var server = http.createServer(function (req, res) {
 fs.readFile('./index.html', function(error, data) {
 res.writeHead(200, { 'Content-Type': 'text/html' });
 res.end(data, 'utf-8');
 });
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

---

看过了前几章看到的 Hello World 示例，读者应该对这个服务器感到熟悉。这里不同的是在发送 HTML 文件之前使用了 Node.js 的文件系统模块来读该文件。为了在该服务器中加入 Socket.IO 的功能，必须将 Socket.IO 库包括进来，而后将其附加到服务器上。

```
var io = require('socket.io').listen(server);
```

这会将 Socket.IO 绑定到服务器上，于是有效地让任何连接到服务器上的客户端都具备实时通信功能。在启用了服务器的 Socket.IO 之后，需要一些代码来让 Socket.IO 可对特定事件和来自客户端的消息作出响应。Socket.IO 侦听许多事件，包括客户连接和断开连接。在本示例中，只要客户端连接或者断开连接，就会有消息记录到终端上。当启用了 Socket.IO 的页面装载到浏览器时，连接就发生。当同样的浏览器页面关闭时，断开连接就发生。在程序清单 12.2 中，服务器将这些事件记录到控制台。

程序清单 12.2 给服务器添加 Socket.IO 功能

---

```
io.sockets.on('connection', function (socket) {
 console.log('User connected');
 socket.on('disconnect', function () {
 console.log('User disconnected');
 });
});
```

---

Socket.IO 服务器现在已经完成了，可以接收来自客户端的连接了。为了让客户端（或浏览器）可以连接到服务器，它们必须使用 Socket.IO 中带的客户端 JavaScript 库，然后连接到服务器。服务器提供单个 index.html 文件服务，这需要和服务器连接。通过在 index.html 中

添加如下代码即可实现：

```
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect('http://127.0.0.1:3000');
</script>
```

这会包括进单一的一个能让客户端（或浏览器）与服务器通信的 JavaScript 文件。注意这可以是任何服务器，但在本例中，它被连接到运行于 127.0.0.1 端口 3000 上的 Node.js 服务器。

#### 提示 Socket.IO 自动提供客户端的库

读者可能已经注意到了在客户端的代码中，有一个名为 socket.io 的 JavaScript 文件。如果在同一服务器上运行服务器和客户端，那么 Socket.IO 库将会自动为你提供。这包括浏览器连接到服务器并且发送/接收消息所需的所有代码。

**Did you  
know?**

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour12/example01 找到。按照下列步骤连接 Socket.IO 服务器和客户端。

1. 创建一个新的名为 socket.io 的文件夹。
2. 在 socket.io 文件夹中，创建一个名为 package.json 的新文件并加入以下内容以便将 Socket.IO 声明为依赖模块：

```
{
 "name": "socketio_example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "socket.io": "0.8.7"
 }
}
```

3. 在 socket.io 文件夹中，创建一个名为 app.js 的带有如下内容的新文件：

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

var server = http.createServer(function (req, res) {
 fs.readFile('./index.html', function(error, data) {
 res.writeHead(200, { 'Content-Type': 'text/html' });
 res.end(data, 'utf-8');
 });
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');

var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
 console.log('User connected');
 socket.on('disconnect', function () {
 console.log('User disconnected');
 });
});
```

4. 在 socket.io 文件夹中，创建一个名为 index.html 的带有如下内容的新文件：

```
<!DOCTYPE html>
<html>
 <head>
 <title>Socket.IO Example</title>
 </head>
 <body>
 <h1>Socket.IO Example</h1>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect('http://127.0.0.1:3000');
 </script>
 </body>
</html>
```

5. 从终端运行如下命令安装依赖模块：

```
npm install
```

6. 从终端运行如下命令启动服务器：

```
node app.js
```

7. 打开浏览器访问 <http://127.0.0.1:3000>。

8. 可看到一个显示 “Socket.IO Example.” 的页面。

9. 检查服务器记录在终端上的日志，可看到 “User connected.”。

10. 打开另外一个浏览器选项卡，浏览 <http://127.0.0.1:3000>。

11. 可在服务器日志上看到另外一条 “User connected.”。

12. 关闭浏览 <http://127.0.0.1:3000> 的浏览器选项卡。

13. 在服务器日志上可看到 “User disconnected”。

这个示例演示了当客户端做诸如连接和断开连接这样的事情时，Socket.IO 对特定事件的侦听。在下一个示例中我们将走得更远，将这一数据送回给客户端。

## 12.5 从服务器发送数据到客户端

在第一个示例中，我们使用了 Socket.IO，在客户端连接或者断开一个简单的 Socket.IO 服务器的连接时做记录。不过，要是想将数据发送给已经连接的浏览器又该如何？要想从服务器将数据发送给单个客户端，可以这样：

```
io.sockets.on('connection', function (socket) {
 socket.emit('message', { text: 'You have connected!' });
});
```

注意在这个示例中，Socket.IO 发送的是一个 JavaScript 对象，所以如果有更复杂的需求的话，可以使用更复杂的数据结构。

这段代码需要添加到应用程序的服务器端。只要客户端连接，它就将数据发送给每个新的客户端。而如果想给每个当前已连接的客户端发送消息的话，也可以发送广播消息。将数据广播给所有已连接的客户端，可以这样：

```
io.sockets.on('connection', function (socket) {
 socket.broadcast.emit('message', { text: 'A new user has connected' });
});
```

这是相似的，不同的只是将消息发送给所有已连接的客户端而不仅仅是刚刚连接的客户端。注意消息还是以 JavaScript 对象在“message”事件下发送。“message”这个词可以是任何东西，但它是重要的，因为当接收到数据时需要设置客户端侦听“message”事件。不一定必须以 JavaScript 对象来发送数据，如果需要的话也可以发送文本，但使用 JavaScript 对象可以让客户端更容易使用数据。

在客户端（或者浏览器），要包含如下 JavaScript 以便首先连接 Socket.IO 服务器然后侦听在“message”事件上接收的数据。

```
var socket = io.connect('http://127.0.0.1:3000');
socket.on('message', function (data) {
 alert(data.text);
});
```

通过使用这些功能以及一些客户端的 JavaScript，就可能创建实时的计数器来计算已启用 Socket.IO 的服务器上所连接的客户端数量。其思想是这样的：当服务器启动后，计数器从 0 开始。当客户端连接到服务器时，它递增 1。当客户端断开连接时，它递减 1。Socket.IO 的消息机制可用于实时地随着客户端连接与断开连接将服务器上客户端的数量信息更新给所有的客户端。考虑一下，这就是站点访问者的实时统计数据了！

要实现这个功能，可使用一个简单的 JavaScript 变量来保存服务器上的用户数量。当服务器启动时，用户数量会是 0：

```
var counter = 0;
```

而后，可将其与 Socket.IO 的事件一起使用，通过 JavaScript 的递增递减功能，在客户端加入和离开的时候递增递减计数器的值：

```
counter++; // This increases the count by one.
counter--; // This decreases the count by one.
```

服务器既然有了服务器上客户端数量的计数，那么客户端自己就可通过使用 Socket.IO、使用 socket.emit.broadcast 将数据发送到所有的客户端的方式来更新。在本示例中，数据在“users”事件下发送，因为它与用户数量相关。

```
socket.broadcast.emit('users', { number: count });
```

当客户端断开连接时这些代码能良好工作，但当新客户端连接时，却接收不到更新，因为 socket.emit.broadcast 不会送达连接中的客户端。为了解决这个问题，要以相同的“users”事件将 socket.emit 消息多发送一次给连接中的客户端：

```
socket.emit('users', { number: count });
```

Node.js 服务器的 Socket.IO 部分现在如程序清单 12.3 所示。

### 程序清单 12.3 使用 Socket.IO 的实时计数器

```
var io = require('socket.io').listen(server);
io.sockets.on('connection', function (socket) {
 count++;
```

```

socket.emit('users', { number: count });
socket.broadcast.emit('users', { number: count });
socket.on('disconnect', function () {
 count--;
 socket.broadcast.emit('users', { number: count });
});
});

```

服务器将已连接的用户数发送给客户端！但我们如何将此显示给用户呢？需要加入一些客户端的 JavaScript 来侦听“users”事件，从而接收并显示数据。只需少量 Socket.IO 代码和一些 JavaScript 来更新 DOM 即可做到（见程序清单 12.4）。

#### 程序清单 12.4 在客户端处理事件

```

<script src="/socket.io/socket.io.js"></script>
<script>
var socket = io.connect('http://127.0.0.1:3000');
var count = document.getElementById('count');
socket.on('users', function (data) {
 count.innerHTML = data.number
});
</script>

```

当接收到“users”事件时，会使用所收到的数值来更新 HTML 页面中 id 为 count 的段落。方便起见，在客户端和服务端发送和接收消息的语法一模一样。

为了让数值能正确显示，需要在 HTML 中加入 id 为 count 的段落标记。

```
<p id="count"></p>
```

服务器现在已经设置好，可以接收来自客户端的连接和断开连接，并递增递减计数器。它还可以将这个信息实时广播给所有已连接的客户端。只要一点客户端的 JavaScript，已连接的客户端就可使用这一数据为用户更新视图，以便用户看到已连接的用户数量。那么，该是启动浏览器并看看数据来回流动的时候了！

#### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour12/example02 找到。以下代码演示实时 Socket.IO 计数器。

1. 创建一个名为 realtime\_counter 的新文件夹。
2. 在 realtime\_counter 文件夹中，创建一个名为 package.json 的新文件并加入如下内容，以便将 Socket.IO 声明为依赖模块：

```

{
 "name": "socketio_example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "socket.io": "0.8.7"
 }
}

```

3. 在 realtime\_counter 文件夹内，创建一个带有如下内容的、名为 app.js 的新文件：



```

var http = require('http');
var fs = require('fs');
var count = 0;

var server = http.createServer(function (req, res) {
 fs.readFile('./index.html', function(error, data) {
 res.writeHead(200, { 'Content-Type': 'text/html' });
 res.end(data, 'utf-8');
 });
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');

var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
 count++;
 console.log('User connected. ' + count + ' user(s) present. ');
 socket.emit('users', { number: count });
 socket.broadcast.emit('users', { number: count });
 socket.on('disconnect', function () {
 count--;
 console.log('User disconnected. ' + count + ' user(s) present. ');
 socket.broadcast.emit('users', { number: count });
 });
});

```

4. 在 socket.io 文件夹中，创建一个带有如下内容的、名为 index.html 的新文件：

```

<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8" />
 <title>Socket.IO Example</title>
 </head>
 <body>
 <h1>Socket.IO Example</h1>
 <h2>How many users are here?</h2>
 <p id="count"></p>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect('http://127.0.0.1:3000');
 var count = document.getElementById('count');
 socket.on('users', function (data) {
 console.log('Got update from the server');
 console.log('There are ' + data.number + ' users');
 count.innerHTML = data.number
 });
 </script>

 </body>
</html>

```

5. 从终端运行如下命令安装依赖模块：

```
npm install
```

6. 从终端运行如下命令启动服务器：

```
node app.js
```

7. 打开浏览器浏览 `http://127.0.0.1:3000`。

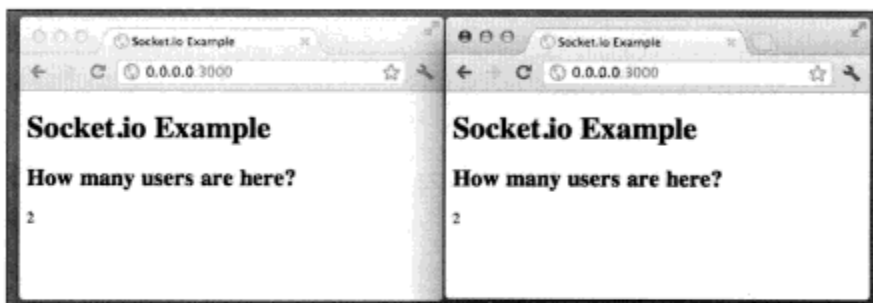
8. 可看到一个页面，显示“Socket.IO Example”并且服务器上有一个用户。

9. 打开另一个浏览器选项卡访问 `http://127.0.0.1:3000`。

10. 可看到现在在服务器上有两个用户。切换到另一个选项卡，可看到这个选项卡上的内容也已经更新为两个用户（见图 12.1）。

图 12.1

使用 Socket.IO 的  
实时计数器



11. 关闭所打开的浏览器选项卡中的一个，可看到服务器上的用户数量少了 1。

## 12.6 将数据广播给客户端

实时计数器展示了客户端数量数据如何实时往外推送给客户端。但如果想让客户端之间做通信呢？有可能会有需要添加聊天功能的场合，或者有需要在客户端之间发送消息的游戏。

为了让客户端可以互相通信，首先需要从客户端发送一条消息给服务器然后从服务器将消息推送给客户端。过程如下。

1. 客户端连接到 Socket.IO Node.js 服务器。
2. 一个客户端发送消息给服务器。
3. 服务器接收此消息。
4. 服务器将消息广播给所有其他客户端。

我们已经在上一个示例中看到，可以从 Socket.IO 服务器将消息广播给客户端：

```
socket.broadcast.emit('message', { text: "This goes to everyone!" });
```

我们还看到，可以从服务器给单个客户端发送消息：

```
socket.emit('message', { text: "This goes to a single client" });
```

从客户端发送消息给服务器的代码完全相同！与实时计数器示例不同的是，消息是在客户端原发，而且来自客户端 JavaScript。于是，在客户端（或者浏览器）上，必须要有能让用户添加消息并提交给服务器的方法。一种实现这个功能的方法是创建一个表单然后使用一些 JavaScript 来捕获用户输入到表单中的内容并将其发送给服务器。在本示例中使用了 jQuery，但如果读者选择使用普通的 JavaScript 来实现也可以。用来捕获消息的表单是一个标准的 HTML 表单：

```
<form id="message-form" action="#">
 <textarea id="message" rows="4" cols="30"></textarea>
 <input type="submit" value="Send message" />
</form>
```

客户端的 JavaScript 会利用它来捕获表单的内容并发送到服务器：

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></
<script>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect('http://127.0.0.1:3000');
 var message = document.getElementById('message');
 $(message.form).submit(function() {
 socket.emit('message', { text: message.value });
 return false;
 });
 socket.on('broadcast', function (data) {
 $('form').after('<p>' + data.text + '</p>');
 });
</script>
```

在这段 JavaScript 中发生了什么呢？

- 将 jQuery 和 Socket.IO 库包含进来。
- 指示浏览器连接位于 `http://127.0.0.1:3000` 的 Socket.IO 服务器。
- 使用 jQuery 的 `submit()` 方法加入侦听器，等候用户提交表单。
- 发送消息给 Socket.IO 服务器，文本区域的内容作为消息发送。
- 添加了一条 `return false` 语句，以防止表单在浏览器窗口中提交。

在实时计数器示例中，加入了一些代码以便在客户端接收消息。在本示例中，必须在服务器上设立一些代码来接收消息。我们希望实现的功能如下所示。

- 接收来自客户端的消息。
- 消息立刻广播到所有其他客户端，但不包括发送该消息的客户端。

读者可能已经知道可以使用 `socket.broadcast.emit` 来实现。这样可将一条消息广播给所有的客户端，除了发送这条消息的客户端以外。只需在服务器端使用一小段代码就可实现：

```
io.sockets.on('connection', function (socket) {
 socket.on('message', function (data) {
 socket.broadcast.emit('push message', data);
 });
});
```

现在，只要服务器收到一条消息，就会实时将其推送到除了发送这条消息的客户端之外的所有的客户端！读者将看到这是在广播事件之下发送的。“`push message`”这个名称可以是任何东西，但它将被用在客户端以便侦听这一类型的新消息。本示例已近乎完成，不过目前客户端还没有将服务器推送过来的消息显示出来的方法。为了接收消息，我们需要创建一些客户端的 JavaScript 以实现如下内容。

- 侦听“`push message`”事件。
- 捕捉接收的数据。
- 将数据写入 DOM 或页面以使用户阅读。

将下列代码加入到客户端就完成了本示例：

```
socket.on('push message', function (data) {
 $('form').after('<p>' + data.text + '</p>');
});
```

只需少量代码，当接收到广播事件时，数据就会被传递到一个使用 jQuery 的 `after()` 方法的匿名函数中。它会将数据插入到 DOM 中给定元素之后的位置上。在本例中，数据插入在表单之后，包含在两个段落标记之间。我们已经有了消息在一个简单的消息服务器上的完整循环。

1. 客户端连接到启用了 Socket.IO 的 Node.js 服务器。
2. 客户端提交信息并单击 Submit。
3. 消息被发送到服务器。
4. 服务器将消息广播给所有其他客户端。
5. 客户端用消息内容更新 Web 页面。

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour12/example03` 找到。以下步骤展示使用 Socket.IO 在客户端间发送消息的方法。

1. 创建一个名为 `simple_messaging` 的新文件夹。
2. 在 `simple_messaging` 文件夹中，创建一个名为 `package.json` 的新文件并添加如下内容将 Socket.IO 声明为依赖模块：

```
{
 "name": "socketio_example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "socket.io": "0.8.7"
 }
}
```

3. 在 `simple_messaging` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件：

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

var server = http.createServer(function (req, res) {
 fs.readFile('./index.html', function(error, data) {
 res.writeHead(200, { 'Content-Type': 'text/html' });
 res.end(data, 'utf-8');
 });
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');

var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
 socket.on('message', function (data) {
 socket.broadcast.emit('push message', data);
 });
});
```

4. 在 `simple_messaging` 文件夹中, 创建一个带有如下内容的、名为 `index.html` 的新文件:

```
<html lang="en">
 <head>
 <meta charset="utf-8" />
 <title>Socket.IO Example</title>
 </head>
 <body>
 <h1>Socket.IO Example</h1>
 <form id="message-form" action="#">
 <textarea id="message" rows="4" cols="30"></textarea>
 <input type="submit" value="Send message" />
 </form>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect('http://127.0.0.1:3000');
 var message = document.getElementById('message');
 $(message.form).submit(function() {
 socket.emit('message', { text: message.value });
 return false;
 });
 socket.on('push message', function (data) {
 $('form').after('<p>' + data.text + '</p>');
 });
 </script>

 </body>
</html>
```

5. 从终端运行如下命令安装依赖模块:
- ```
npm install
```
6. 从终端运行如下命令启动服务器:
- ```
node app.js
```
7. 打开浏览器访问 `http://127.0.0.1:3000`。
8. 打开另外一个浏览器选项卡访问 `Http://127.0.0.1:3000`。
9. 在文本框中输入消息并单击 **Submit**。
10. 在另外一个浏览器选项卡中可看到消息出现。
11. 在文本框中输入消息并单击 **Submit** 往回发送一条消息。
12. 可在另一个窗口中看到消息出现 (见图 12.2)。

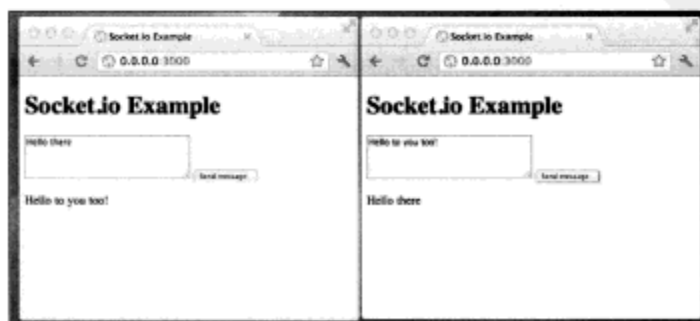


图 12.2

在 Socket.IO 客户端之间发送消息

## 12.7 双向数据

在实时计数器和消息服务器示例中，我们看到了如何从服务器发送数据给客户端以及如何从客户端发送数据给服务器。我们双向发送了数据，而这是需要理解的 **Socket.IO** 和 **WebSockets** 的关键特性。**WebSockets** 带来的是让客户端和服务端能够在需要的时候互相推送数据的方法。这真正改变了 Web 的风景以及我们所能给浏览器所创建的应用程序的类型。当浏览器发送消息给服务器的同时，服务器也可能在给客户端发送消息。其他客户可能在更新服务器，客户端可能正在加入与离开服务器。可以看到，即使在一个小的示例中，也有许多事件在发生！**JavaScript** 对此非常适合。它是一种事件驱动的语言，而且，由于 **Node.js** 将服务器端的联网功能带入到这么一个聚会中，于是 **JavaScript** 就成了这类应用程序的香饽饽了。

为了巩固双向数据发送的理论知识，本章的最后一个示例演示客户端与服务端之间的数据流以及如何使用事件对更多消息进行响应。在计算科学中，**PING** 这个术语是对一个服务或者服务器的请求，要求该服务或者服务器告诉请求者它是否正常工作（或者说活着）。所期望的响应是 **PONG**。想想乒乓球比赛，球在网的两边来回击打——**PING-PONG**！我们使用这个思想来探究 **Socket.IO** 中的消息流——发送一条消息，请求另一端响应一个 **PONG**。

在服务器端，本示例实现如下内容。

1. 在收到 **PING** 时发送 **PONG** 响应。
2. 当收到 **PONG** 响应时在终端上记录一条消息。
3. 每 10 秒发送一条 **PING** 消息给客户端。

在客户端，本示例实现如下内容。

1. 在收到 **PING** 时发送 **PONG** 响应。
2. 当收到 **PONG** 响应时在终端上记录一条消息。
3. 每 10 秒发送一条 **PING** 消息给服务器。

在开始为这些功能写代码之前，需要重点理解的是，这些事件可能在任何时间以任何顺序发生。毕竟这些消息是双向的！为了实现服务器端的功能，**Socket.IO** 的代码如程序清单 12.5 所示。

程序清单 12.5 演示 **Socket.IO** 中的消息收发

```
io.sockets.on('connection', function (socket) {
 socket.on('ping', function (data) {
 console.log('Received PING. Sending PONG..');
 socket.emit('pong', { text: 'PONG' });
 });
 socket.on('pong', function (data) {
 console.log('Received PONG response. PONG!');
 });
 setInterval(function() {
 console.log('Sending PING to client..');
 socket.emit('ping', { text: 'PING' });
 }, 10000);
});
```

这里创建了两个侦听器事件用来侦听“ping”和“pong”事件。如果收到“ping”事件，就响应以 PONG 消息。如果收到“pong”事件，就记录到终端。为了每 10 秒发送一条消息给客户端，使用了 `setInterval()`。它简单地按照所指定的时间间隔重复某些事情（在本例中，是 10 000 毫秒或者 10 秒）。在客户端，功能几乎相同，但“ping”事件是由用户单击 Submit 按钮来触发的。

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect('http://127.0.0.1:3000');
 socket.on('ping', function (data) {
 console.log('Received PING. Sending PONG..');
 socket.emit('pong', { text: 'PONG' });
 });
 socket.on('pong', function (data) {
 console.log('Received PONG response. PONG!');
 });
 $('#ping').click(function() {
 console.log('Sending PING to server..')
 socket.emit('ping', { text: 'PING' });
 });
</script>
```

在这段 JavaScript 中发生了什么呢？

- 将 jQuery 和 Socket.IO 库包含进来。
- 指示浏览器连接位于 `http://127.0.0.1:3000` 的 Socket.IO 服务器。
- 使用 jQuery 的 `submit()` 方法加入侦听器，等候用户提交表单。
- 创建两个侦听器，以响应“ping”和“pong”事件。“ping”事件以“pong”响应。“pong”事件将事件记录到浏览器控制台。
- 为了让用户发送 PING，使用了 jQuery 的 `click()` 方法侦听 id 为 ping 的 HTML 按钮的单击事件。当这个按钮被单击，就会把 ping 消息发送到服务器。

用来捕获单击事件并发送 ping 给服务器的 HTML 按钮很简单：

```
<button id="ping">Send PING to server</button>
```

这个示例完成了 ping-pong 消息收发循环。如果读者能理解其中的原理，就理解了 Socket.IO 的消息收发机制。通过观察消息来回移动更容易理解，所以现在启动浏览器吧！

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour12/example04` 找到。按照下列步骤查看 Socket.IO 中的动态消息发送过程。

1. 创建一个名为 `ping_pong` 的新文件夹。
2. 在 `ping_pong` 文件夹中，创建一个名为 `package.json` 的新文件，并加入如下内容将 Socket.IO 声明为依赖模块：

```
{
 "name": "socketio_example",
 "version": "0.0.1",
```

```

 "private":true,
 "dependencies":{
 "socket.io":"0.8.7"
 }
 }
}

```

3. 在 `ping_pong` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的文件：

```

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

var server = http.createServer(function (req, res) {
 fs.readFile('./index.html', function(error, data) {
 res.writeHead(200, { 'Content-Type': 'text/html' });
 res.end(data, 'utf-8');
 });
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');

var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
 socket.on('ping', function (data) {
 console.log('Received PING. Sending PONG..');
 socket.emit('pong', { text: 'PONG' });
 });
 socket.on('pong', function (data) {
 console.log('Received PONG response. PONG!');
 });
 setInterval(function() {
 console.log('Sending PING to client..');
 socket.emit('ping', { text: 'PING' });
 }, 10000);
});

```

4. 在 `ping_pong` 文件夹中，创建一个带有如下内容的、名为 `index.html` 的新文件：

```

<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8" />
 <title>Socket.IO Example</title>
 </head>
 <body>
 <h1>Socket.IO Example</h1>
 <button id="ping">Send PING to server</button>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect('http://127.0.0.1:3000');
 socket.on('ping', function (data) {
 console.log('Received PING. Sending PONG..');
 socket.emit('pong', { text: 'PONG' });
 });
 socket.on('pong', function (data) {
 console.log('Received PONG response. PONG!');
 });
 </script>
 </body>
</html>

```



```

});
$('#ping').click(function() {
 console.log('Sending PING to server..')
 socket.emit('ping', { text: 'PING' });
});
</script>
</body>
</html>

```

5. 从终端运行如下命令安装依赖模块:

```
npm install
```

6. 从终端运行如下命令启动服务器:

```
node app.js
```

7. 打开 Firefox、Chrome 或者 Safari 浏览 `http://127.0.0.1:3000` 并打开一个控制台窗口。

8. 查看当接收到来自服务器的“ping”事件以及发送“pong”响应的时候,浏览器控制台的内容(见图 12.3)。

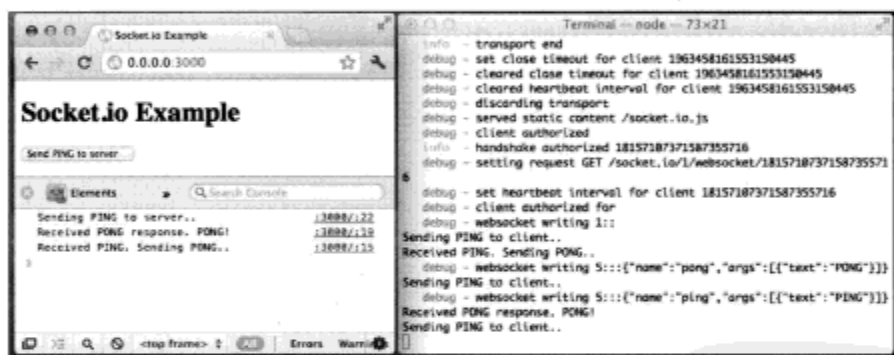


图 12.3

在服务器和客户端之间收发 ping-pong 消息

9. 在浏览器窗口中,单击 Submit 按钮发送“ping”给服务器。

10. 查看终端日志,看“ping”的接收和“pong”的返回(见图 12.3)。

## 12.8 小结

在本章,我们了解了真正让 Node.js 与众不同的所在。我们了解了如何使用 Socket.IO 创建能在服务器和客户端之间实时通信的应用程序!我们学习了如何从服务器发送数据给客户端,然后从客户端发送数据给服务器。我们理解了如何在服务器和客户端上接收数据以及如何使用客户端的 JavaScript 将数据显示给用户。在最后一个示例中,我们理解了消息如何触发其他消息,以及消息流如何可以是双向的。在本章我们学习了许多 Socket.IO 背后的理论,现在,读者有了构建实时 Web 所需的所有工具了!

## 12.9 问与答

问:是否所有浏览器都支持 WebSocket?

答:简而言之,不是。在本书编写的时候,WebSocket 标准仍旧还是草案,该标准还没完成。许多浏览器供应商已经实现了 WebSocket,但有些供应商出于安全考虑禁用了这一功

能。虽然对 WebSocket 的支持有望成为所有浏览器之间的标准，但在直到本书编写之时，对 WebSocket 的支持还是不可期。而好消息是，Socket.IO 会处理浏览器并不完全支持 WebSocket 的问题。如果有 WebSocket 的支持，那么 Socket.IO 就选择这一支持。否则，Socket.IO 会尝试使用 Adobe Flash Socket、Ajax 长轮询、Ajax 多部分流 (multipart streaming)、Forever iFrame 或者 JSONP 轮询。读者无需理解这些是什么，因为 Socket.IO 无缝地处理这些事情，但如果需要的话也可以选择启用哪种传输方式。这就是说，如果使用 Socket.IO，大量浏览器都支持它。在本书编写时，Socket.IO 对浏览器的支持包括：Internet Explorer 5.5+、Safari 3+、Google Chrome 4+、Firefox 3+、Opera 10.61+、iPhone Safari、iPad Safari、Android WebKit 以及 WebOS WebKit。很吸引人啊！

问：我能否跨域 (cross-domain) 连接？

答：是的，可以。如果读者有使用 Ajax 的经验，那么就会知道，要想启用跨域请求，必须使用额外的技巧。在使用 Socket.IO 的时候无需这些。它就是能工作！

问：Ajax 和 Socket.IO 之间有什么区别？

答：Socket.IO 提供的功能要比 Ajax 多得多，比如很容易就能实时地将数据推送给客户端。如果可以使用 WebSocket，那么客户端和服务端之间的连接就会是持久的。而对于 Ajax 而言，每次都需要建立连接。

问：我应该使用 WebSocket 还是 Ajax？

答：WebSocket 和 Ajax 的服务目标不同，所以不好说要使用哪个不使用哪个。Ajax 适合浏览器需要时不时请求数据并且数据有可能需要缓冲的场景。如果经常需要在客户端之间收发数据，而且有许多客户端，那应该使用 WebSocket。如果要构建一个客户端能互相通信的联网应用程序，那么 WebSocket 是个更好的选择。

## 12.10 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

### 12.10.1 问题

1. 推送和拉取消息之间有何不同？
2. WebSocket 当前对浏览器的支持如何？（提示：可使用 <http://caniuse.com/> 来帮助你。）
3. 使用 Socket.IO 比起使用单纯的 WebSocket 有何优势？

### 12.10.2 答案

1. 推送消息意味着新的消息通过持久连接推送到客户端。这意味着客户端无需请求获得新消息。作为对比，拉取消息意味着客户端必须检查服务器上的内容，看看是否有新的消息可用。

2. 如果访问 <http://caniuse.com> 并在搜索框中输入 WebSocket, 就会看到 WebSocket 的支持情况。尤其在移动设备上, 对 WebSocket 的支持不是很好, 于是这就增加了使用 Socket.IO 的筹码了。

3. 如果不能使用 WebSocket, Socket.IO 会使用其他传输方法。这就意味着无需编写更多代码就可以让应用程序支持更多浏览器。

## 12.11 练习

1. 创建一个 Socket.IO 应用程序, 当新客户端连接的时候, 发送 “You have connected!” 消息给客户端。

2. 扩展习题 1, 编写一些客户端的 JavaScript 接收消息并将消息显示给客户端。

3. 探索 Firefox、Safari 和 Chrome 中可用的客户端 JavaScript 调试器。如果需要刷新一下记忆, 请看第 9 章。如果不能访问所有这些浏览器, 那么就选择你的系统上可用的一个或几个。试试本章中的几个示例, 理解使用客户端调试器和 `console.log` 来帮助进行 Socket.IO 应用程序开发的方法。



## 第 13 章

# 一个 Socket.IO 聊天服务器

在本章中你将学到：

- 与 Express 一起使用 Socket.IO；
- 添加昵称并维护昵称列表；
- 与 Socket.IO 一起使用回调；
- 在连接的用户之间发送消息；
- 使用 Socket.IO 创建一个功能齐备的聊天服务器。

### 13.1 Express 和 Socket.IO

在第 12 章中，我们介绍了 Socket.IO 以及如何用它在浏览器和服务器之间实时发送消息。在本章，我们将了解一个实用的 Socket.IO 应用程序——一个基于浏览器的聊天服务器。这是 Node.js 的另一个优秀的使用案例！

聊天服务器有如下功能。

- 它允许用户设置昵称。
- 它显示一组连接的用户。
- 它将聊天信息广播给所有已连接的用户。

在以前的几章，我们介绍了 Express，这是一个 Node.js 上的 Web 框架。Express 使得我们可更简单地使用 Node.js 创建 Web 应用程序，而且，幸运的是，它能很好地与 Socket.IO 集成。我们仅仅使用 Node.js 自带的标准库就能创建 Socket.IO 应用程序，但使用 Express 可以将许多共同的问题抽象开。为了与 Express 一起使用 Socket.IO，需要将 Socket.IO 绑定到 Express 服务器：

```
var app = module.exports = express.createServer(),
 io = require('socket.io').listen(app);
```

一个简单的 Express Socket.IO 服务器如下:

```
var app = module.exports = express.createServer(),
 io = require('socket.io').listen(app);

app.listen(3000);

app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});

io.sockets.on('connection', function (socket) {
 socket.emit('welcome', { text: 'OH HAI! U R CONNECTED!' });
});
```

它为浏览器服务一个单一的 index.html 文件并在客户端连接的时候发送欢迎消息。在客户端 (或者浏览器), 代码与在第 12 章看到的示例一样。在使用 Express 服务的 index.html 文件中, 为 welcome 事件设置了一个侦听器, 接收消息并将其写入到 JavaScript 控制台:

```
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect();
 socket.on('welcome', function (data) {
 console.log(data.text);
 });
</script>
```

## TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour13/example01` 找到。按照下列步骤使用 Express 创建一个基础的 Socket.IO 服务器。

1. 创建一个名为 `socket.io_express` 的新文件夹。
2. 在 `socket.io_express` 文件夹中, 创建一个名为 `package.json` 的新文件并加入下列内容以便将 Socket.IO 和 Express 声明为依赖模块:

```
{
 "name": "socket.io-express-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "socket.io": "0.8.7"
 }
}
```

3. 在 `socket.io_express` 文件夹中, 创建一个带有下列内容的、名为 `app.js` 的新文件:

```
var app = module.exports = express.createServer(),
 io = require('socket.io').listen(app);

app.listen(3000);
```

```

app.get('/', function (req, res) {
 res.sendFile(__dirname + '/index.html');
});

io.sockets.on('connection', function (socket) {
 socket.emit('welcome', { text: 'OH HAI! U R CONNECTED!' });
});

```

4. 在 `socket.io_express` 文件夹中, 创建一个带有下列内容的、名为 `index.html` 的新文件:

```

<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.IO Express Example</title>
 </head>
 <body>
 <h1>Socket.IO Express Example</h1>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect();
 socket.on('welcome', function (data) {
 console.log(data.text);
 });
 </script>
 </body>
</html>

```

5. 从终端运行如下命令安装依赖模块:

```
npm install
```

6. 从终端运行如下命令启动服务器:

```
node app.js
```

7. 打开浏览器浏览 `http://127.0.0.1:3000`。

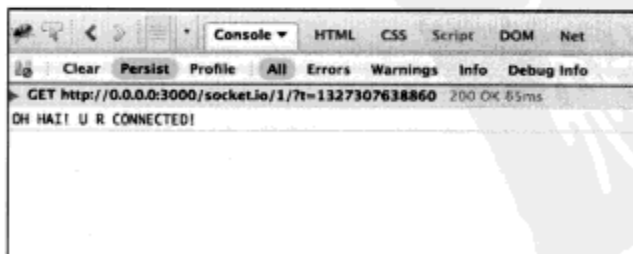
8. 可看到一个显示 “Socket.IO Express Example” 的页面。

9. 打开浏览器的 JavaScript 控制台。

10. 可看到 “OH HAI! U R CONNECTED!” (见图 13.1), 这表明浏览器已经连接到 Socket.IO 服务器并且收到了消息。

图 13.1

与 Express 一起使用 Socket.IO



## 13.2 添加昵称

既然有了一个基础的连接到了 Socket.IO 的 Express 服务器, 现在可以添加功能了。第一

个功能是让用户能够在加入服务器的时候设置昵称。它的工作方式如下。

1. 用户可以将昵称输入到表单中。
2. 当提交表单时将昵称发送给服务器。
3. 服务器检查昵称的唯一性。
4. 如果昵称已经被使用，服务器要通知客户端。
5. 如果昵称还不存在，则将其加入到昵称列表中。
6. 服务器将昵称列表广播给所有已连接的客户端。
7. 客户端接收广播并更新昵称列表。

### 13.2.1 将昵称发送给服务器

第一步是要在 `index.html` 中添加一个表单以使用户输入昵称。

```
<form id="set-nickname">
 <label for="nickname">Nickname:</label>
 <input type="text" id="nickname" />
 <input type="submit" />
</form>
```

有了表单，就可使用客户端的 JavaScript 在用户提交表单时捕获昵称。jQuery 的 `submit()` 方法可用于侦听对表单的提交事件并捕获输入字段中的值然后发送给 Socket.IO 服务器：

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect();
 jQuery(function ($) {
 var nickname = $('#nickname');
 var setNicknameForm = $('#set-nickname');
 setNicknameForm.submit(function(event) {
 event.preventDefault();
 socket.emit('nickname', nickname.val());
 });
 });
</script>
```

以下是在上述 JavaScript 中所发生的一切。

1. 从 Google 的 Content Delivery Network 包含进 jQuery 库。
2. 将 Socket.IO 库包含进来，以便让页面能连接到 Socket.IO 服务器。
3. 使用 `io.connect()` 将页面连接到 Socket.IO 服务器。
4. 创建一个匿名函数用于包装我们在页面上使用的任何 jQuery 函数。这种安全的做法可确保其他可能使用 \$ 的库或函数可以与 jQuery 一起使用。
5. jQuery 的 `submit()` 方法侦听表单上的提交事件。
6. 使用 `event.preventDefault()` 防止表单提交。

7. 使用 jQuery 的 `val()` 方法来捕获用户输入的昵称。
8. Socket.IO 使用 `socket.emit()` 将昵称发送给服务器。

### Did you know?

#### 提示：Socket.IO 可自动发现服务器

如果 Socket.IO 客户端和服务端位于同一个服务器上（如本示例的情况），在使用 `io.connect()` 的时候就无需指定服务器。Socket.IO 将自动发现它。但如果要跨来源进行连接，就要将 URL 传递给 `io.connect()` 函数（比如 `io.connect('http://yourdomain.com/')`）。

读者应该熟悉这一模式，这也是上一章 `hour12/example03` 的代码示例所用的。代码捕获来自用户的输入并使用客户端的 JavaScript 将其发送到 Socket.IO 服务器上。为了接收消息，需要在服务器端代码中加入侦听器：

```
socket.on('nickname', function (data) {
 console.log('The server received the following nickname: ' + data);
});
```

如此就给 `nickname` 事件添加了一个侦听器，可以接收数据，也就是可以使用了。

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour13/example02` 找到。为了使用 Socket.IO 从客户端接收数据，请按如下步骤进行。

1. 创建一个名为 `sending_nickname` 的新文件夹。
2. 在 `sending_nickname` 文件夹中，创建一个名为 `package.json` 的新文件并添加如下内容将 Socket.IO 和 Express 声明为依赖模块：

```
{
 "name": "socket.io-express-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "socket.io": "0.8.7"
 }
}
```

3. 在 `sending_nickname` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件：

```
var app = module.exports = express.createServer(),
 io = require('socket.io').listen(app);

app.listen(3000);

app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});

io.sockets.on('connection', function (socket) {
 socket.on('nickname', function (data) {
```



```

 console.log('The server received the following nickname: ' + data);
 });
});

```

4. 在 socket.io\_express 文件夹中, 创建一个带有如下内容的、名为 index.html 的新文件:

```

<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.IO Express Example</title>
 </head>
 <body>
 <h1>Socket.IO Express Example</h1>
 <form id="set-nickname">
 <label for="nickname">Nickname:</label>
 <input type="text" id="nickname" />
 <input type="submit" />
 </form>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect();
 jQuery(function ($) {
 var nickname = $('#nickname');
 var setNicknameForm = $('#set-nickname');
 setNicknameForm.submit(function(event) {
 event.preventDefault();
 socket.emit('nickname', nickname.val());
 });
 });
 </script>
 </body>
</html>

```

5. 从终端运行如下命令安装依赖模块:  
npm install
6. 从终端运行如下命令启动服务器:  
node app.js
7. 打开浏览器访问 <http://127.0.0.1:3000>。
8. 可看到一个显示“Socket.IO Express Example”的页面。
9. 将昵称输入到表单中单击 Submit 按钮。
10. 在服务器日志中, 可看到服务器已经接收了昵称。

### 13.2.2 管理昵称列表

到目前为止, 本应用程序可以从浏览器发送昵称给服务器了。但是, 这并不是特别有用! 服务器需要有能够维护已连接的昵称列表的方法, 并且能够在用户断开连接时从列表中移除昵称。这里可以使用一些简单的 JavaScript 来设置一个空数组保存昵称, 然后在用户连接和

断开连接时加入或者移除昵称。在服务器端，创建一个空的数组来保存昵称：

```
var nicknames = [];
```

当服务器收到昵称，要将其加入到数组中时，可采用 JavaScript 的 `push()` 方法：

```
io.sockets.on('connection', function (socket) {
 socket.on('nickname', function (data) {
 nicknames.push(data);
 });
});
```

实际上，在从客户端接收数据时，需要做一些清洁工作，这里为了简单起见忽略了。当客户端从服务器断开连接时，要将用户的昵称从列表中移除。最简洁的实现方法就是使用如下的 JavaScript 代码片段：

```
nicknames.splice(nicknames.indexOf("Item to remove"), 1);
```

这会在数组中查找条目并移除。可以在 Socket.IO 服务器中使用它，以便在用户断开连接时移除昵称。Socket.IO 可以给每个客户端设置一个变量以使用户在断开连接事件中访问昵称。`nickname` 事件需要做一些修改，从而设置这一变量以便在以后使用：

```
socket.on('nickname', function (data) {
 nicknames.push(data);
 socket.nickname = data;
});
```

既然客户端的昵称可以在服务器上通过使用 `socket.nickname` 来使用，那么我们就可以用它在用户断开连接时从服务器移除昵称：

```
socket.on('disconnect', function () {
 if (!socket.nickname) return;
 if (nicknames.indexOf(socket.nickname) > -1) {
 nicknames.splice(nicknames.indexOf(socket.nickname), 1);
 }
});
```

在这里，首先检查 `socket.nickname` 的存在会是个好主意，因为如果不存在的话就会抛出异常。还有，如果要移除的昵称不在昵称数组中的话，会导致不需要的结果。于是，通过一些简单的服务器端代码，我们现在可以维护昵称列表了！

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour13/example03` 找到。为了使用管理昵称，请按如下步骤进行。

1. 创建一个名为 `managing_nicknames` 的新文件夹。

2. 在 `managing_nicknames` 文件夹中，创建一个名为 `package.json` 的新文件并添加如下内容将 Socket.IO 和 Express 声明为依赖模块：

```
{
 "name": "socket.io-express-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
```

```

 "express": "2.5.4",
 "socket.io": "0.8.7"
 }
}

```

3. 在 `managing_nicknames` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件：

```

var app = require('express').createServer(),
 io = require('socket.io').listen(app),
 nicknames = [];

app.listen(3000);

app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});

io.sockets.on('connection', function (socket) {
 socket.on('nickname', function (data) {
 nicknames.push(data);
 socket.nickname = data;
 console.log('Nicknames are ' + nicknames);
 });
 socket.on('disconnect', function () {
 if (!socket.nickname) return;
 if (nicknames.indexOf(socket.nickname) > -1) {
 nicknames.splice(nicknames.indexOf(socket.nickname), 1);
 }
 console.log('Nicknames are ' + nicknames);
 });
});

```

4. 在 `socket.io_express` 文件夹中，创建一个带有如下内容的、名为 `index.html` 的新文件：

```

<!doctype html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.IO Express Example</title>
 </head>
 <body>
 <h1>Socket.IO Express Example</h1>
 <form id="set-nickname">
 <label for="nickname">Nickname:</label>
 <input type="text" id="nickname" />
 <input type="submit" />
 </form>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect();
 jQuery(function ($) {
 var nickname = $('#nickname');
 var setNicknameForm = $('#set-nickname');
 setNicknameForm.submit(function(event) {
 event.preventDefault();

```

```

 socket.emit('nickname', nickname.val());
 });
}
</script>
</body>
</html>

```

5. 从终端运行如下命令安装依赖模块：

```
npm install
```

6. 从终端运行如下命令启动服务器：

```
node app.js
```

7. 打开浏览器访问 <http://127.0.0.1:3000>。
8. 可看到一个显示“Socket.IO Express Example”的页面。
9. 将昵称输入到表单中单击 Submit 按钮。
10. 在服务器日志中，可看到服务器当前具有的昵称列表。
11. 打开另一个浏览器选项卡，输入另外一个昵称，单击 Submit。
12. 在服务器日志中，可看到在昵称列表中加入了该昵称。
13. 关闭访问 Socket.IO 示例的一个浏览器选项卡。
14. 在服务器日志中，可看到在昵称列表中移除了该昵称。

### 13.2.3 使用回调来验证

读者可能已经注意到，目前管理服务器上昵称的实现有许多问题。如果你注意到下列问题中的任何一个，就给你加分！

- 用户可以输入同一个昵称超过一次。
- 在服务器端没有验证。
- 一旦成功提交昵称，表单仍旧是可见的，于是理论上应该再提交另一个昵称。

为了让客户端能够将正确的信息显示给用户，这里采用使用 Socket.IO 时的一个关键技巧——回调。回调时让服务器在发送消息后给客户返回一些消息的方法。可以将此想象为：客户端问，“嘿，咋样了啊？是否一切正常？”。这让客户端得以知道昵称是否已经在昵称列表中并按此行动。客户端可以有效地从服务器接收到一切如期的确认信息。以下是期望的逻辑。

1. 用户通过客户端的表单提交昵称。
2. 服务器接收昵称消息。
3. 服务器检查昵称是否已经在已注册的昵称列表中。
4. 如果昵称已经在昵称列表中，服务器发出一个一切不正常的回调。
5. 如果昵称不在昵称列表中，就将其加入到昵称列表并且发出回调告诉客户端一切正常。
6. 如果回调告诉客户端一切正常，就将昵称表单隐藏。

在客户端，用于将昵称发送给服务器的代码如下：

```

var socket = io.connect();
jQuery(function ($) {
 var nickname = $('#nickname');
 var setNicknameForm = $('#set-nickname');
 setNicknameForm.submit(function(event) {
 event.preventDefault();
 socket.emit('nickname', nickname.val());
 });
});

```

这段代码可修改一下，以便通过回调，在服务器接收了消息之后，接收来自服务器端的数据。这样客户端就能知道另一端的情况了：

```

jQuery(function ($) {
 var nickname = $('#nickname');
 var setNicknameForm = $('#set-nickname');
 setNicknameForm.submit(function(event) {
 event.preventDefault();
 socket.emit('nickname', nickname.val(), function (data) {
 if (data) {
 console.log('Nickname set successfully');
 setNicknameForm.hide();
 } else {
 setNicknameForm.prepend('<p>Sorry - that nickname is already taken.</p>');
 }
 });
 });
});

```

如果服务器以真返回回调，那么昵称就已经成功添加。客户端可使用 jQuery 的 `hide()` 方法将表单从用户面前隐藏。另一方面，如果服务器返回假，那么就显示一条消息给用户说昵称已经有人用了。客户端而后使用 jQuery 的 `prepend()` 方法将一个段落插入到 DOM（文档对象模型）或者页面中，告诉用户昵称已经有人用了。在服务器端，需要添加一些逻辑，根据昵称是否已经在昵称列表中来给回调返回真或者假：

```

socket.on('nickname', function (data, callback) {
 if (nicknames.indexOf(data) !== -1) {
 callback(false);
 } else {
 callback(true);
 nicknames.push(data);
 socket.nickname = data;
 console.log('Nicknames are ' + nicknames);
 }
});

```

昵称事件中的匿名函数现在需要两个参数：一个从客户端（数据）接收的数据以及一个函数（回调）。这个函数是会从服务器返回给客户端的回调函数。可使用 JavaScript 的 `indexOf()` 函数来返回某个条目是否已经在数组中。如果昵称不在数组中，就会返回 -1，于是通过这么一个简单的、对昵称是否在数组中的测试就可设置真和假的值。这个值会返回给客户端，以便隐藏表单或者告诉用户昵称已经有人用了。在客户端和服务器之间收发消息时，回调是极为强大的工具，在需要客户端了解所发送的数据有何结果时就应当使用它们。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour13/example04` 找到。按下列步骤演示客户端与服务器之间的回调。

1. 创建一个名为 `callbacks` 的新文件夹。

2. 在 `callbacks` 文件夹中，创建一个名为 `package.json` 的新文件并添加如下内容将 `Socket.IO` 和 `Express` 声明为依赖模块：

```
{
 "name": "socket.io-express-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "socket.io": "0.8.7"
 }
}
```

3. 在 `callbacks` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件：

```
var app = require('express').createServer(),
 io = require('socket.io').listen(app),
 nicknames = [];

app.listen(3000);

app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});

io.sockets.on('connection', function (socket) {
 socket.on('nickname', function (data, callback) {
 if (nicknames.indexOf(data) !== -1) {
 callback(false);
 } else {
 callback(true);
 nicknames.push(data);
 socket.nickname = data;
 console.log('Nicknames are ' + nicknames);
 }
 });
 socket.on('disconnect', function () {
 if (!socket.nickname) return;
 if (nicknames.indexOf(socket.nickname) > -1) {
 nicknames.splice(nicknames.indexOf(socket.nickname), 1);
 }
 console.log('Nicknames are ' + nicknames);
 });
});
```

4. 在 `socket.io_express` 文件夹中，创建一个带有如下内容的、名为 `index.html` 的新文件：

```
<!doctype html>
```

```

<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.io Express Example</title>
 </head>
 <body>
 <h1>Socket.io Express Example</h1>
 <form id="set-nickname">
 <label for="nickname">Nickname:</label>
 <input type="text" id="nickname" />
 <input type="submit" />
 </form>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect();
 jQuery(function ($) {
 var nickname = $('#nickname');
 var setNicknameForm = $('#set-nickname');
 setNicknameForm.submit(function(event) {
 event.preventDefault();
 socket.emit('nickname', nickname.val(), function (data) {
 if (data) {
 console.log('Nickname set successfully');
 setNicknameForm.hide();
 } else {
 setNicknameForm.prepend('<p>Sorry - that nickname is already
taken.</p>');
 }
 });
 });
 });
 </script>
 </body>
</html>

```

5. 从终端运行如下命令安装依赖模块:

```
npm install
```

6. 从终端运行如下命令启动服务器:

```
node app.js
```

7. 打开浏览器访问 <http://127.0.0.1:3000>。

8. 可看到一个显示“Socket.IO Express Example”的页面。

9. 将昵称输入到表单中单击 Submit 按钮。

10. 在服务器日志中, 可看到服务器当前具有的昵称列表。还可看到有一条消息被记录到了浏览器的 JavaScript 控制台, 表明已经成功设置了昵称 (见图 13.2)。

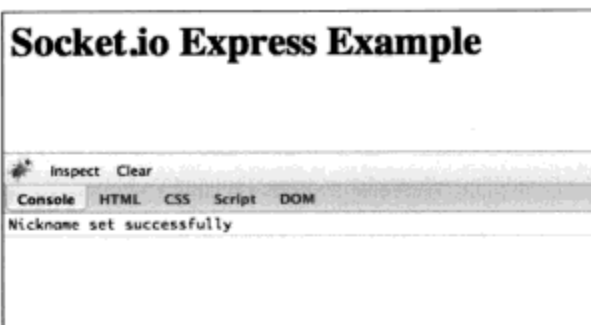
11. 打开另一个浏览器选项卡, 输入另外一个昵称, 单击 Submit。

12. 在服务器日志中, 可看到该昵称已经在服务器当前拥有的昵称列表中。还可看到客户端的表单已经被隐藏, 有一条消息被记录到了 JavaScript 控制台中。

13. 打开另一个浏览器选项卡, 输入同一个昵称, 单击 Submit。

图 13.2

使用回调来设置  
昵称



14. 可看到一条消息表明该昵称已经有人用了（见图 13.3），这表示服务端验证能工作了！

图 13.3

使用回调验证昵称



### 13.2.4 广播昵称列表

既然维护昵称列表的逻辑已经齐备，那么应用程序就需要将列表广播给客户端，以便让用户看到谁在服务器上。就如在第 12 章所看到的那样，在 Socket.IO 中要将消息广播给所有用户很简单，我们可以给昵称事件加一个发送器（emitter）将昵称列表发送给客户端：

```
io.sockets.emit('nicknames', nicknames);
```

在客户端就可以接收列表然后将其写到页面中，将谁在服务器上展示给用户。可使用简单的 JavaScript 循环处理数据，迭代数组并将昵称添加到 DOM 中。为了给 JavaScript 写入昵称列表的地方，需要在 HTML 文档中加入一个空白的节标记，其 id 为“nicknames”，用来包含一个空白的、不排序的列表：

```
var nicknamesList = $('#nicknames ul');
socket.on('nicknames', function (data) {
 var html = '';
 for (var i = 0; i < data.length; i++) {
 html += '' + data[i] + '';
 }
 nicknamesList.empty().append(html);
});
```

当客户端收到“nickname”事件时，使用 jQuery 的 empty() 函数来清空位于节标记中的不排序列表。这样做可解决这么一种场景：有一份前一次更新过的、但现在已经过期的昵称清单。数据显示在不排序的清单中，于是 jQuery 的 append() 函数将数据添加到 DOM 中。由于数据是 JavaScript 数组，所以可以被迭代（或者循环）。当新用户加入服务器时，服务器就



对所有已连接的客户端广播“nickname”事件，昵称列表就可以得到更新。当用户断开连接时，通过再次发送昵称列表，客户端也可以得到更新。

如果想在浏览器中试试这段代码，本示例可在本书的代码示例的 hour13/example05 文件夹中找到。

### 13.2.5 添加消息收发功能

应用程序现在在服务器上有了昵称列表，并且在用户加入和离开服务器时可更新客户端的信息显示。但用户尚不能互相通信！为了给应用程序加入消息收发功能，需要在 index.html 文件中添加另一个表单，以便让用户提交消息。这是个标准 HTML 表单：

```
<form id="send-message">
 <textarea id="message"></textarea>
 <input type="submit" />
</form>
```

这个表单使用 CSS 中的“display: none”来隐藏，直到成功设置了昵称之后。在成功设置了昵称之后，就可使用回调来隐藏昵称表单并显示消息表单。用于提交昵称的客户端 JavaScript 现在是这样子的：

```
var nickname = $('#nickname');
var setNicknameForm = $('#set-nickname');
var messages = $('#messages');
setNicknameForm.submit(function(event) {
 event.preventDefault();
 socket.emit('nickname', nickname.val(), function (data) {
 if (data) {
 console.log('Nickname set successfully');
 setNicknameForm.hide();
 messages.show();
 } else {
 setNicknameForm.prepend('<p>Sorry - that nickname is already taken.</p>');
 }
 });
});
```

如果来自服务器的回调表明昵称已经成功添加，那么就使用 jQuery 的 hide() 方法隐藏添加昵称的表单，使用 jQuery 的 show() 方法显示提交消息的表单，有效地将它们切入切出。既然表单现在对用户可见，我们可以添加一些客户端 JavaScript 来侦听表单的提交事件并将文本区域的内容发送给 Socket.IO 服务器。而后，表单会被清空，焦点回到表单上以使用户可以输入另一条消息。

```
var messageForm = $('#send-message');
var message = $('#message');
messageForm.submit(function(event) {
 event.preventDefault();
 socket.emit('user message', message.val());
 message.val('').focus();
});
```

在服务器端，创建了一个侦听器来接收“user message”事件并将其广播给所有已连接的

客户端。这同时也使用在 `nickname` 事件上所设置的 `socket.nickname` 与消息一起发送昵称：

```
socket.on('user message', function (data) {
 io.sockets.emit('user message', {
 nick: socket.nickname,
 message: data
 });
});
```

回到客户端，添加了一些客户端的 JavaScript 来接收消息广播并显示给用户。这会接受消息然后使用 jQuery 的 `append()` 方法将消息写到页面中。注意无需分析从 Socket.IO 接收的数据，可以直接使用！

```
var messages = $('#messages');
socket.on('user message', function (data) {
 messages.append('<p>' + data.nick + '' + data.message + '</p>');
});
```

最后，需要用上一点 CSS 在页面刚刚装载时隐藏的消息表单。还需添加少许基本的样式，以便在页面的右边显示昵称列表，让文本框大一些并且在昵称和消息之间加入一些空白：

```
<style>
#send-message { display: none; }
#nicknames { width: 300px; float: right; }
#message { width: 300px; height: 100px; }
#messages p strong { margin-right: 5px; }
</style>
```

---

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour13/example06` 找到。按下列步骤在客户端与服务器之间收发消息。

1. 创建一个名为 `messaging` 的新文件夹。
2. 在 `messaging` 文件夹中，创建一个名为 `package.json` 的新文件并添加如下内容将 Socket.IO 和 Express 声明为依赖模块：

```
{
 "name": "socket.io-express-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "socket.io": "0.8.7"
 }
}
```

3. 在 `messaging` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件：

```
var app = require('express').createServer(),
 io = require('socket.io').listen(app),
 nicknames = [];

app.listen(3000);
```

```

app.get('/', function (req, res) {
 res.sendFile(__dirname + '/index.html');
});

io.sockets.on('connection', function (socket) {
 socket.on('nickname', function (data, callback) {
 if (nicknames.indexOf(data) !== -1) {
 callback(false);
 } else {
 callback(true);
 nicknames.push(data);
 socket.nickname = data;
 console.log('Nicknames are ' + nicknames);
 io.sockets.emit('nicknames', nicknames);
 }
 });
 socket.on('user message', function (data) {
 io.sockets.emit('user message', {
 nick: socket.nickname,
 message: data
 });
 });
});

socket.on('disconnect', function () {
 if (!socket.nickname) return;
 if (nicknames.indexOf(socket.nickname) > -1) {
 nicknames.splice(nicknames.indexOf(socket.nickname), 1);
 }
 console.log('Nicknames are ' + nicknames);
 io.sockets.emit('nicknames', nicknames);
});
});

```

4. 在 messaging 文件夹中，创建一个带有如下内容的、名为 index.html 的新文件：

```

<!doctype html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.IO Express Example</title>
 <style>
 #send-message { display: none; }
 #nicknames { width: 300px; float: right; }
 #message { width: 300px; height: 100px; }
 #messages p strong { margin-right: 5px; }
 </style>
 </head>
 <body>
 <h1>Socket.IO Express Example</h1>
 <form id="set-nickname">
 <label for="nickname">Nickname:</label>
 <input type="text" id="nickname" />
 <input type="submit" />
 </form>
 <form id="send-message">
 <textarea id="message"></textarea>
 <input type="submit" />
 </form>
 </body>
</html>

```

```

</form>
<section id="nicknames">

</section>
<section id="messages">
</section>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
min.js"></script>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect();
 jQuery(function ($) {
 var nickname = $('#nickname');
 var setNicknameForm = $('#set-nickname');
 var nicknamesList = $('#nicknames ul');
 var messageForm = $('#send-message');
 var message = $('#message');
 var messages = $('#messages');
 setNicknameForm.submit(function(event) {
 event.preventDefault();
 socket.emit('nickname', nickname.val(), function (data) {
 if (data) {
 console.log('Nickname set successfully');
 setNicknameForm.hide();
 messageForm.show();
 } else {
 setNicknameForm.prepend('<p>Sorry - that nickname is already
taken.</p>');
 }
 });
 });
 messageForm.submit(function(event) {
 event.preventDefault();
 socket.emit('user message', message.val());
 message.val('').focus();
 });

 socket.on('nicknames', function (data) {
 var html = '';
 for (var i = 0; i < data.length; i++) {
 html += '' + data[i] + '';
 }
 nicknamesList.empty().append(html);
 });
 socket.on('user message', function (data) {
 messages.append('<p>' + data.nick + '' + data.
message + '</p>');
 });
 });
</script>
</body>
</html>

```

##### 5. 从终端运行如下命令安装依赖模块:

```
npm install
```

6. 从终端运行如下命令启动服务器：

```
node app.js
```

7. 打开浏览器访问 <http://127.0.0.1:3000>。

8. 可看到一个显示“Socket.IO Express Example”的页面。

9. 将昵称输入到表单中单击 Submit 按钮。可看到昵称显示在了屏幕的右边。

10. 在服务器日志中，可看到服务器当前具有的昵称列表。

11. 打开另一个浏览器选项卡，输入另外一个昵称，单击 Submit。可看到在屏幕的右边有两个昵称。

12. 开始在选项卡之间聊天。可看到两个选项卡中都显示了对话内容！

聊天服务器现在工作良好，但怎么说都有点功能化了些。由于我们在浏览器中做工作，所以对 HTML 和 CSS 有完全的掌控！关于布局 and 设计的细节超出了本书范畴，但如果读者需要有个例子来开开胃口，在本书代码示例中有一个具备更多设计的聊天服务器版本，位置在 `hour13/exmample07`（见图 13.4）。

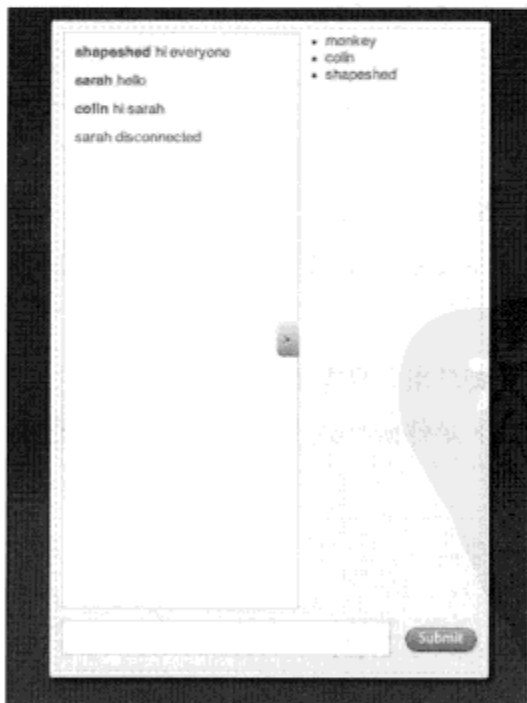


图 13.4

应用了一些样式的  
聊天应用程序

## 13.3 小结

在本章，我们学习了如何使用 Socket.IO 创建一个全功能的聊天服务器。我们学习了如何在使用 Express 框架的 Node.js 应用程序中使用 Socket.IO。我们学习了如何在服务器上管理昵称列表以及如何使用回调将数据发送的结果信息返回给原发送者。我们也看到了如何广播昵称列表以及如何将消息收发功能添加到应用程序中的方法。更为重要的是，我们学习了如何一起使用客户端和服务端端的 JavaScript 来创建丰富的、实时的应用程序！

## 13.4 问与答

问：我注意到在客户端 Socket.IO 应用程序逻辑要比服务器端多，这是个糟糕的设计吗？

答：大多数应用程序逻辑都变成客户端的 JavaScript 是我们所创建的聊天服务器这样的单页面应用程序的特性之一。对于大多数逻辑存在于服务器端并且习惯于将 HTML 发送给浏览器的经典应用程序，这是一种范式上的转变。这样的设计也称为“厚服务器，瘦客户端”，因为大多数处理都发生在服务器上。现代的 JavaScript 应用程序翻转了这一思想，更多的逻辑发生在浏览器上。于是带来一个有许多令人兴奋的可能性的世界，但也不是没有问题。对于服务器，我们可以 100% 确认硬件的能力和所使用的软件。对于浏览器，就必须依赖各种不同的 JavaScript 引擎和 CSS 渲染能力，更不用说还有屏幕分辨率问题。涉足过前端开发的人都了解在不同浏览器上部署的痛苦！像 Socket.IO 和 jQuery 这样的库为我们将浏览器之间的许多不同抽象出来。这是使用库的另外一个理由！这不是糟糕的设计，但公平地说，它的确带来一些新的挑战。

问：我在哪儿能找到放置这种应用程序的主机？

答：由于 Socket.IO 依赖 WebSocket，所以需要找到支持 WebSocket 的主机。不是所有的 Node.js PaaS 主机都支持 WebSocket。在本书编写时，Nodester、Nodejitsu、Nodesocket、Cure 和 Cloudnode 支持 WebSocket。

问：Express 可以和 Socket.IO 共享会话吗？

答：是的，可以。许多方法使用 Redis（一种键-值存储）来储存 Express 会话并且允许 Socket.IO 访问它。此实现超过了本书范畴。

问：可否给本应用程序添加关键词禁用和日志功能？

答：当然可以！可挂入 Socket.IO 事件侦听器来检查消息中的禁用词，或者将消息写入文件或数据存储中。

## 13.5 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

### 13.5.1 问题

1. 使用 Socket.IO 是否必须使用 Express？
2. 如果服务器进程被杀死或者死机，昵称列表会发生什么？
3. 可否将 Socket.IO 服务器和 Web 应用程序托管在不同的地方？

### 13.5.2 答案

1. 不需要。Socket.IO 独立于 Express。另外，Socket.IO 为我们抽象了 WebSocket，只要

愿意，也可以写我们自己的 WebSocket 服务器并处理客户端代码。

2. 在本章的示例里，昵称列表只保持在内存中。如果进程被杀死或者死机的话，昵称列表就会丢失。如果需要，应该使用诸如 Redis 这样的键-值存储机制来保存这个列表。

3. 是的。虽然 Socket.IO 服务器和 Web 应用程序位于同一个进程中，但在这些示例里，如果需要也可以分开。

## 13.6 练习

1. 使用在第 12 章学到的知识给应用程序添加一个新功能：每次当用户加入或离开服务器的时候在聊天窗口上贴一条消息。这道习题的解题方法可参阅本书代码示例中的 `hour13/exmaple07`。

2. 体验一下在聊天应用程序加入一些样式的感觉。如果读者不具备设计能力，可与了解 CSS 和 HTML 的朋友或者同事一起加入一些简单的样式。

3. 要是想找挑战，可在应用程序中加入私有消息功能。这样可以让消息只在两个已连接的客户端之间发送，而不是广播给服务器上的所有人。



## 第 14 章

# 一个流 Twitter 客户端

在本章中你将学到：

- 从 Twitter 的流 API 接收数据；
- 分析来自 Twitter 的流 API 的数据；
- 实时将第三方数据推送给客户；
- 创建实时图；
- 通过使用来自 Twitter 的实时数据来探究这个世界上是爱更多还是恨更多。

### 14.1 流 API

在第 13 章中，我们学习了如何使用 Socket.IO 和 Express 创建一个聊天服务器。它涉及从客户端（或浏览器）发送数据给 Socket.IO 服务器、而后将其广播给其他客户端。在本章，我们学习如何使用 Node.js 和 Socket.IO 直接从 Web 消费数据，然后将数据广播给已连接的客户端。我们将使用 Twitter 的流应用程序接口（API）并实时将数据推送到浏览器上。

使用 Twitter 的标准 API 获取数据的过程如下。

- 打开一个对 API 服务器的连接。
- 发送对数据的请求。
- 从 API 接收所请求的数据。
- 连接关闭。

使用 Twitter 的流 API，这个过程有所不同，如下所示。

- 打开一个对 API 服务器的连接。



- 发送对数据的请求。
- 从 API 推送数据给你。
- 连接保持打开状态。
- 在数据可用时会有更多数据推送给你。

流 API 使得在有新数据可用时，可以将数据从服务提供商推送给用户。在 Twitter 这个例子中，数据会极为频繁而大量。Node.js 极适合于这种随着大量数据的接收事件频繁发生的场景。本章展示了 Node.js 的另一个优秀的使用案例并且凸显了让 Node.js 与其他语言和框架不同的特点。

## 14.2 注册 Twitter

Twitter 通过免费的、公开可用的 API 向开发人员提供数量巨大的数据。许多 Twitter 桌面和移动客户端都以此 API 为基础构建，但这个 API 也对用任何技术的开发人员开放。

如果读者没有 Twitter 账户，那么本章你需要有一个。读者可以在 <https://twitter.com/免费注册> 注册一个账号。只需花费不到一分钟！一旦有了 Twitter 账号，就需要在 <http://dev.twitter.com/使用> 详细个人信息注册到 Twitter 开发人员网站。该站点提供所有一切跟 Twitter API 有关的文档和论坛。该文档很全面，如果需要的话，可以在这里好好了解可以从 API 请求的数据类型有哪些。

在 Twitter 开发人员网站里，还可以注册使用 Twitter API 创建的应用程序。我们要在本章创建一个 Twitter 应用程序，为了注册应用程序，按如下步骤进行。

1. 单击 Create an App 链接。
2. 给应用程序起个名称并填写表单（见图 14.1）。Twitter 上的应用程序名必须是唯一的，所以如果发现名称已经被占用，就请选用另外一个。

The screenshot shows the 'Create an application' page on the Twitter Developers website. The page has a header with 'twitter developers' and navigation links like 'API Health', 'Blog', 'Discussions', 'Documentation', and 'nodejsbook.io'. The main content area is titled 'Create an application' and contains a form with the following fields:

- Name:** lovehatecenter
- Description:** A lovehatecenter to show if there is more love or hate in the world
- Website:** http://nodejsbook.io
- Callback URL:** (empty)

Below the form, there is a 'Create Application' button and a 'Cancel' link. The page also includes a 'My applications' link and a 'Create Application' button.

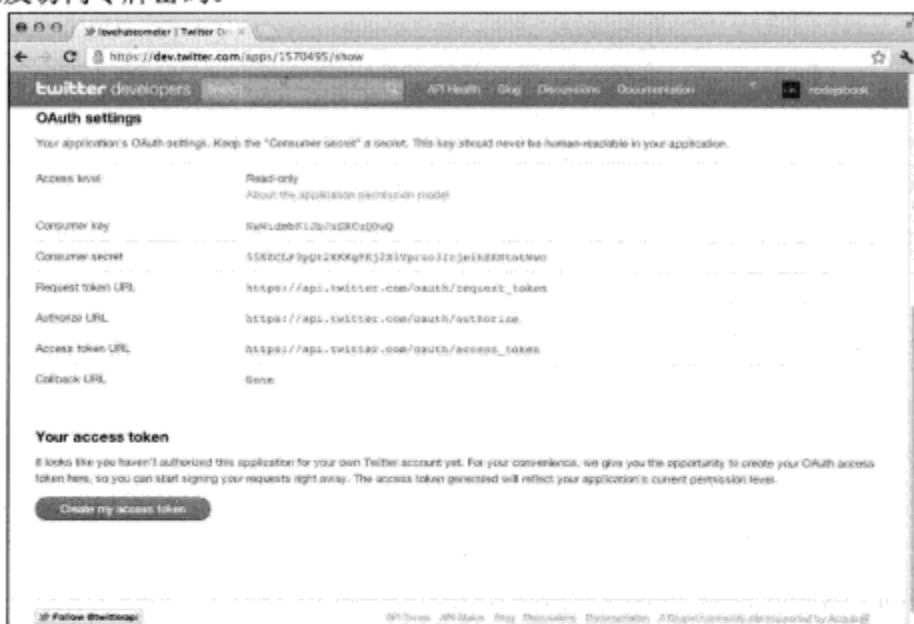
图 14.1  
创建 Twitter 应用程序

一旦创建了应用程序，就需要生成一个访问令牌和一个访问令牌密码（access token secret）来获得从应用程序访问 API 的权限。

3. 在 Details 选项卡的底部是个 Create My Access Token 按钮（见图 14.2）。单击这个按钮创建访问令牌以及访问令牌密码。

图 14.2

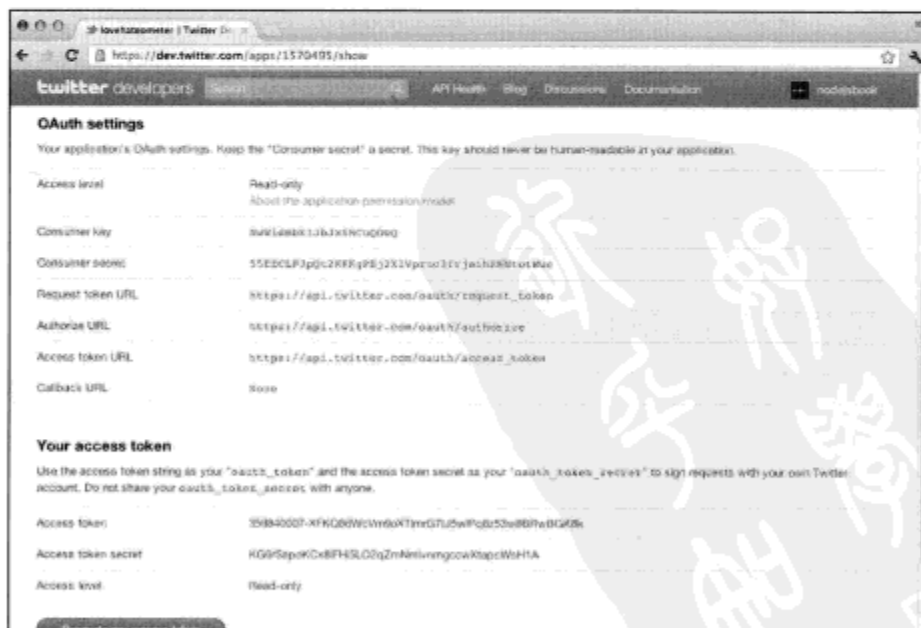
注册一个访问令牌



4. 当页面刷新时，会看到访问令牌和访问令牌密码的值出现（见图 14.3）。现在可以开始使用 API 了！

图 14.3

成功创建访问令牌



**By the Way**

**注意：**OAuth 是允许访问在线账户的一种方法

OAuth 是验证的一种开发标准，典型地用在 Web 应用程序的上下文中。它让用户可以不通过提交用户名或密码就可授予对某个账户的全部或部分访问权。当用户授予应用程序访问其账户的权限时，要生成一个唯一的令牌。第三方服务器可以使用它来访问用户账户的全部或一部分。在任何时候，用户可以撤销访问权限，令牌将不再有效，应用程序也就不再能访问账户了。

## 14.3 和 Node.js 一起使用 Twitter 的 API

一旦在 Twitter 开发人员网站内创建了应用程序并请求了 OAuth 访问令牌，就可以开始使用 Twitter API 了。有一个很优秀的名为 `ntwitter` 的 Node.js 模块可用来与 Twitter API 交互。这个模块最初由 `technoweenie` (Rick Olson) 开发，然后是 `jdub` (Jeff Waugh)，现在由 `AviaFlu` (Charlie McConnell) 维护。这些作者做了出色的工作，将与 Twitter API 交互的复杂性抽象了出来，让操作 Twitter API 获取数据进行工作变成小菜一碟。本章我们继续使用 Express，所以应用程序的 `package.json` 文件将包括 Express 和 `ntwitter` 模块。

```
{
 "name": "socket.io-twitter-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "ntwitter": "0.2.10"
 }
}
```

`ntwitter` 模块使用 OAuth 来验证户，所以必须提供如下的 4 组信息。

- 消费者钥匙 (Consumer key)。
- 消费者密码 (Consumer secret)。
- 访问令牌钥匙 (Access token key)。
- 访问令牌密码 (Access token secret)。

如果在 Twitter 开发人员网站上设置应用程序时请求了这些信息，那么这些信息可以在应用程序的 Details 页面上得到。如果没有在设置应用程序时请求它们，那么现在需要在 Details 选项卡中进行请求。一旦有了这些钥匙和密码，就可以创建一个小 Express 服务器来连接到 Twitter 的流 API：

```
var app = require('express').createServer(),
 twitter = require('ntwitter');

app.listen(3000);

var twit = new twitter({
 consumer_key: 'YOUR_CONSUMER_KEY',
 consumer_secret: 'YOUR_CONSUMER_SECRET',
 access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
 access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});
```

当然，要记得将本示例中的值替换为实际值。开始与 Twitter API 交互所需的一切就是这些了！在本示例中，我们要通过使用来自 Twitter 的实时数据回答这么一个问题：在世界上爱更多还是恨更多？我们从 Twitter 的流 API 中请求提及“love”或者“hate”的推文，通过对数据做一些分析来回答这个问题。用 `ntwitter` 模块很容易就可请求这些数据：

```

 twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function (data) {
 console.log(data);
 });
 });
 });
}

```

这会从“statuses/filter”端点（endpoint）请求数据，它允许开发人员按关键字、位置或特定用户来记录推文。在本例中，我们对关键词“love”和“hate”感兴趣。Express 服务器打开一个对 API 服务器的连接并侦听所收到的新数据。当收到新数据项时，它将数据写入控制台。换言之，我们可以在终端中看到关键字“love”和“hate”的流现场直播。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour14/example01` 找到。要从 Twitter 获得流数据，请按如下步骤进行。

1. 创建一个名为 `express_twitter` 的新文件夹。

2. 在 `express_twitter` 文件夹中，创建一个名为 `package.json` 的新文件并加入如下内容将 `ntwitter` 和 `Express` 声明为依赖模块：

```

{
 "name": "socket.io-twitter-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "ntwitter": "0.2.10"
 }
}

```

3. 在 `express_twitter` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件。记得将钥匙和密码替换为你自己的：

```

var app = require('express').createServer(),
 twitter = require('ntwitter');

app.listen(3000);

var twit = new twitter({
 consumer_key: 'YOUR_CONSUMER_KEY',
 consumer_secret: 'YOUR_CONSUMER_SECRET',
 access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
 access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function (data) {
 console.log(data);
 });
});

```

4. 从终端运行如下命令安装依赖模块：

```
npm install
```

5. 从终端运行如下命令启动服务器:

```
node app.js
```

6. 观察终端, 可看到从 Twitter 的流 API 接收到的数据 (见图 14.4)。会有大量数据, 所以要是移动得很快, 那是意料之中的!



```
Terminal — node — 80x24

notifications: null,
profile_background_tile: false,
follow_request_sent: null,
profile_sidebar_fill_color: '252429',
created_at: 'Sat Dec 11 15:04:33 +0000 2010',
protected: false,
default_profile_image: false,
contributors_enabled: false,
profile_sidebar_border_color: '181A1E',
followers_count: 21,
profile_image_url: 'http://a0.twimg.com/profile_images/1789738168/____norma
l.jpg',
name: 'Noura',
id_str: '225434395',
favourites_count: 0,
id: 225434395,
lang: 'en',
profile_use_background_image: true,
utc_offset: -18000,
url: null },
in_reply_to_screen_name: null,
id: 164010055543435260,
entities: { user_mentions: [], urls: [], hashtags: [[Object]] }
```

图 14.4

将流数据显示到终端

7. 在终端上按 Ctrl+C 杀死服务器。

## 14.4 从数据中挖掘含义

到目前为止, 我们创建了实时从 Twitter 检索数据的方法, 我们看到充满大量数据的、快速滚动的终端窗口。这不错, 但要是不能理解数据, 就不能给出问题的答案。为了朝这方面努力, 我们需要能够对所接收到的推文进行分析并挖掘其中的信息。Twitter 以 JSON 格式提供数据, 这是 JavaScript 的一个子集, 对于使用 Node.js 的人来说, 这是个很棒的消息。对于每个响应, 只需简单地使用点号就可检索感兴趣的数据。所以, 如果想查看用户的昵称及其推文, 很容易就可以实现:

```
twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function (data) {
 console.log(data.user.screen_name + ': ' + data.text);
 });
});
```

从 Twitter 接收的数据的结构, 其完整的文档可在状态元素的文档上在线看到, 其位置是: <https://dev.twitter.com/docs/api/1/get/statuses/show/%3Aid>。在“Example Request”一节中, 可以看到状态响应的数据结构。通过对来自 Twitter 所返回的数据对象使用点号, 就可以访问这些数据点中的任意点。比如, 如果想得到用户的 URL, 可以使用 `data.user.url`。以下是对贴出推文的用户可获取的完整数据:

```

"user": {
 "profile_sidebar_border_color": "eeeeee",
 "profile_background_tile": true,
 "profile_sidebar_fill_color": "efefef",
 "name": "Eoin McMillan ",
 "profile_image_url": "http://a1.twimg.com/profile_images/1380912173/Screen_
 ➤ shot_2011-06-03_at_7.35.36_PM_normal.png",
 "created_at": "Mon May 16 20:07:59 +0000 2011",
 "location": "Twitter",
 "profile_link_color": "009999",
 "follow_request_sent": null,
 "is_translator": false,
 "id_str": "299862462",
 "favourites_count": 0,
 "default_profile": false,
 "url": "http://www.eoin.me",
 "contributors_enabled": false,
 "id": 299862462,
 "utc_offset": null,
 "profile_image_url_https": "https://si0.twimg.com/profile_images/1380912173/
 ➤ Screen_shot_2011-06-03_at_7.35.36_PM_normal.png",
 "profile_use_background_image": true,
 "listed_count": 0,
 "followers_count": 9,
 "lang": "en",
 "profile_text_color": "333333",
 "protected": false,
 "profile_background_image_url_https": "https://si0.twimg.com/images/themes/
 ➤ themel4/bg.gif",
 "description": "Eoin's photography account. See @mceoin for tweets.",
 "geo_enabled": false,
 "verified": false,
 "profile_background_color": "131516",
 "time_zone": null,
 "notifications": null,
 "statuses_count": 255,
 "friends_count": 0,
 "default_profile_image": false,
 "profile_background_image_url": "http://a1.twimg.com/images/themes/themel4/
 ➤ bg.gif",
 "screen_name": "imeoin",
 "following": null,
 "show_all_inline_media": false
}

```

每个响应都有更多的信息可用，包括地理坐标、推文是否有回复，等等。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour14/example02` 找到。要分析来自 Twitter 的数据，请按如下步骤进行。

1. 创建一个名为 `parsing_twitter_data` 的新文件夹。
2. 在 `parsing_twitter_data` 文件夹中，创建一个名为 `package.json` 的新文件并加入如下内容将 `ntwitter` 和 `Express` 声明为依赖模块：

```
{
 "name": "socket.io-twitter-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "ntwitter": "0.2.10"
 }
}
```

3. 在 `parsing_twitter_data` 文件夹中, 创建一个带有如下内容的、名为 `app.js` 的新文件。记得将钥匙和密码替换为你自己的:

```
var app = require('express').createServer(),
 twitter = require('ntwitter');

app.listen(3000);

var twit = new twitter({
 consumer_key: 'YOUR_CONSUMER_KEY',
 consumer_secret: 'YOUR_CONSUMER_SECRET',
 access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
 access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function(data) {
 console.log(data.user.screen_name + ': ' + data.text);
 });
});
```

4. 从终端运行如下命令安装依赖模块:

```
npm install
```

5. 从终端运行如下命令启动服务器:

```
node app.js
```

6. 观察终端, 现在应该看到只有用户的昵称和推文显示出来 (见图 14.5)。

7. 在终端中按 `Ctrl+C` 杀死服务器。

```
Terminal - node - 80x24
accioabbie: RT @justsaypenny: I love @accioabbie and @underapapermo they are n
y life XXXXXXXXXXXXXXXXXXXXXXXXXX
JacobIrsan: RT@monicaaakin: pacaran : love you beb, miss you <3 .. putus : f
uck you! damn you...
NSUCoachJones: No ESPN love, 16-6, 8-0 Conference, preseason player of year and
they show the 5th and 6th place teams..
jockinmyfresh24: Call me a jerk one more time ctfu lmao RT @TassieBaddAss: @jock
inmyfresh24 you freakin love me you lil jerk!!!
Huangaholic: @TheManuMania This, a thousand times this. See, this is why I love
you guys. @foreword55 @jeannette_sites
imaginebojan: Jefferson Airplane Somebody To Love http://t.co/1bl9RH76 兄さん
バレルな! www 空耳?
raulorlandoJ: "Never love someone easily" real man
iWant_GUMMIES: i hate when #Twitter try to tell me i already tweeted something a
nd i didn't
ChoczBB: Lowkey Wizkid RT @DONJAYYYY: Kalamoko Skally RT @ChoczBB: Oti yan yan
Slim joe RT @DONJAYYYY: Dadubule Skally RT @ChoczBB: No Love Eminem
_YummyKellogs: I hate Mondays, and I'm tired |:
Kcruzz2: But jealousy is just love n hate at the same time
hishh: Among the all actress i know @divyaspadana is a sweet heart. Real gem ,
brave , dedicated and the best.. Respect.. Loads of love
LUcKyAcE_103: RT @Mr_Gjasmin5: I love google
JakePena7: RT @BW0RD: You're tacky and I hate you
```

图 14.5  
分析从 Twitter 接  
收的数据

## 14.5 将数据推送到浏览器

既然来自 Twitter 的数据已经处理成易于消化的格式，现在可使用 Socket.IO 将数据推送到已连接的浏览器上，然后使用一些客户端 JavaScript 显示推文了。这与在第 12 章和第 13 章中的模式相似：由 Socket.IO 服务器接收数据然后广播到已连接的客户端。为了使用 Socket.IO，需要将其作为依赖模块加入到 package.json 文件中：

```
{
 "name": "socket.io-twitter-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "ntwitter": "0.2.10",
 "socket.io": "0.8.7"
 }
}
```

然后，必须在主服务器中请求 Socket.IO 模块，并要求它侦听 Express 服务器。这与第 12 章和第 13 章中的示例所用的方法完全一样：

```
var app = require('express').createServer(),
 twitter = require('ntwitter'),
 io = require('socket.io').listen(app);
```

现在，对流 API 的请求可以得到增强，从而在收到新数据事件的时候可以将数据推送到任何已连接的 Socket.IO 客户端。

```
twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function (data) {
 io.sockets.volatile.emit('tweet', {
 user: data.user.screen_name,
 text: data.text
 });
 });
});
```

我们现在在做的是一些有用的事情：通过将数据推送到已连接的顾客，而不只是将数据记录到控制台。我们创建了一个简单的 JSON 结构来保持用户的名称和推文。如果将更多信息发送给浏览器，只需扩展 JSON 对象来保存其他属性即可。

读者可能已经注意到，我们现在使用的是 `io.sockets.volatile.emit`，而不是在第 12 章和第 13 章中使用的 `io.sockets.emit`。这是 Socket.IO 提供的另一种方法，用于可以抛除某些消息的场合。这可归结于网络问题或者用户可能正处于请求-响应循环的中间。当大量消息正在送往客户端的时候尤其可能如此。通过使用 `volatile` 方法，可确保应用程序在某个客户端没有接收消息时不至于经受不住。换言之，客户端是否接收到消息无关紧要。

Express 服务器也被要求为单一 HTML 页面提供服务，于是可以在浏览器中查看数据。

```
app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});
```



在客户端（或者浏览器），在 `index.html` 文件中加入了一些简单的客户端 JavaScript 来侦听正在发送到浏览器的新推文并将它们显示给用户。完整的 HTML 文件位于之后的示例中：

```
<ul class="tweets">
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></
<script>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect();
 jQuery(function ($) {
 var tweetList = $('ul.tweets');
 socket.on('tweet', function (data) {
 tweetList
 .prepend('' + data.user + ': ' + data.text + '');
 });
 });
</script>
```

在 DOM（文档对象模型）中添加了一个空白的不排序列表，我们每次收到新推文时，用包含用户昵称和推文的新列表项来填充它。这使用 jQuery 的 `prepend()` 方法将收到的数据插入到不排序列表的列表项中。如此就有了在页面上创建一个流的效果。

现在，只要 Socket.IO 将新的推文事件推送出去，浏览器就会收到它并立即将其写到页面上。我们现在可以在浏览器上看推文流，而不是在终端上看了！

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour14/example03` 找到。以下是将流 Twitter 数据发送到浏览器的方法。

1. 创建一个名为 `socket.io-twitter-example` 的新文件夹。
2. 在 `socket.io-twitter-example` 文件夹中，创建一个名为 `package.json` 的新文件并加入如下内容将 `ntwitter`、`Express` 和 `Socket.IO` 声明为依赖模块：

```
{
 "name": "socket.io-twitter-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "ntwitter": "0.2.10",
 "socket.io": "0.8.7"
 }
}
```

3. 在 `socket.io-twitter-example` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件。记得将钥匙和密码替换为你自己的：

```
var app = require('express').createServer(),
 twitter = require('ntwitter'),
 io = require('socket.io').listen(app);

app.listen(3000);
```

```

var twit = new twitter({
 consumer_key: 'YOUR_CONSUMER_KEY',
 consumer_secret: 'YOUR_CONSUMER_SECRET',
 access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
 access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function (data) {
 io.sockets.volatile.emit('tweet', {
 user: data.user.screen_name,
 text: data.text
 });
 });
});

app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});

```

4. 在 Socket.IO-twitter-example 中, 创建一个名为 index.html 的文件并加入如下内容:

```

<!doctype html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.IO Twitter Example</title>
 </head>
 <body>
 <h1>Socket.IO Twitter Example</h1>
 <ul class="tweets">
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
 min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect();
 jQuery(function ($) {
 var tweetList = $('ul.tweets');
 socket.on('tweet', function (data) {
 tweetList
 .prepend('' + data.user + ': ' + data.text + '');
 });
 });
 </script>
 </body>
</html>

```

5. 从终端运行如下命令安装依赖模块:

```
npm install
```

6. 从终端运行如下命令启动服务器:

```
node app.js
```

7. 打开浏览器浏览 <http://127.0.0.1:3000>。

8. 可在浏览器中看到推文流 (见图 14.6)。

9. 在终端中按 Ctrl+C 杀死服务器。

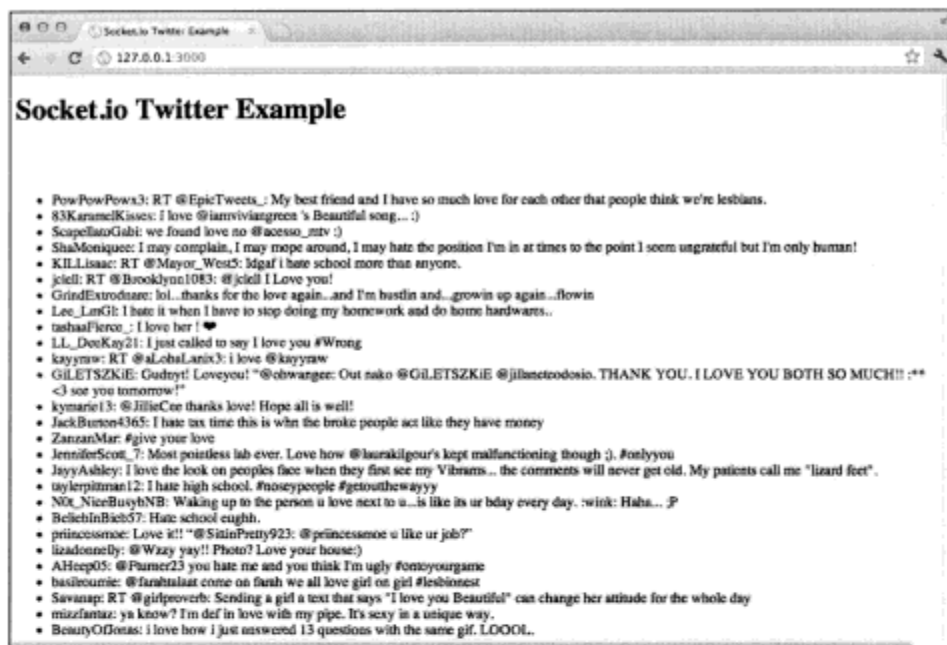


图 14.5

将推文流发送到浏览器

## 14.6 创建一个实时的爱恨表

虽然本应用程序现在可以将流推文发送到浏览器窗口，但还是不太有用。它仍旧无法回答这个世界上爱更多还是恨更多的问题。为了回答这个问题，我们需要能够让数据可视化的方法。假设从 API 收到的推文能够表达人类的情感，我们在服务器上设置几个计数器，当“爱”和“恨”这两个词在所接收到的流数据中出现的时候递增计数器的值。此外，通过维护另外一个对要么有爱要么有恨的推文总数计数的计数器，就可以计算出更常被提到的是爱还是恨。通过这种方法，就有可能以不太科学的术语说，在世界上有 x%的爱和 y%的恨。

为了能够在浏览器中显示数据，需要服务器上的计数器来保存如下数值。

- 包含“爱”或者“恨”的推文总数。
- 包含“爱”的推文总数。
- 包含“恨”的推文总数。

我们可以在 Node.js 服务器上初始化一些变量并将这些计数器设置为 0 来实现：

```
var app = require('express').createServer(),
 twitter = require('ntwitter'),
 io = require('socket.io').listen(app),
 love = 0,
 hate = 0,
 total = 0;
```

每当从 API 收到新数据时，如果找到“爱”这个词，爱的计数器就会递增，如此这般。可使用 JavaScript 的 `indexOf()` 字符串函数来查找推文中的词并提供一个简单的、分析推文内容的方法：

```

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function (data) {

 var text = data.text.toLowerCase();
 if (text.indexOf('love') !== -1) {
 love++;
 total++;
 }
 if (text.indexOf('hate') !== -1) {
 hate++;
 total++;
 }
 });
});

```

因为有些推文可能既包含“爱”又包含“恨”，所以每找到一个词就要递增总数。这就意味着总计数器代表“爱”或者“恨”出现在推文中的总数，而不是推文的总数。

既然应用程序维护了词汇出现次数的计数器，我们就可以将这个数据添加到推文发送器中并实时推送到已连接的客户端。这里需要一些简单的计算工作以便将数值以推文总数的百分数来发送：

```

io.sockets.volatile.emit('tweet', {
 user: data.user.screen_name,
 text: data.text,
 love: (love/total)*100,
 hate: (hate/total)*100
});

```

在客户端，通过使用不排序列表和一些客户端 JavaScript，浏览器可以接收数据并将其显示给用户。在从服务器接收任何数据之前，这些值要被设置为 0：

```

<ul class="percentage">
 <li class="love">0
 <li class="hate">0


```

最后，可以添加一个客户端的侦听器，用来接收推文事件并使用从服务器接收到的数替换所显示的百分数。启动服务器，打开浏览器，现在可以回答问题了！

```

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect();
 jQuery(function ($) {
 var tweetList = $('ul.tweets'),
 loveCounter = $('li.love'),
 hateCounter = $('li.hate');
 socket.on('tweet', function (data) {
 tweetList
 .prepend('' + data.user + ': ' + data.text + '');
 loveCounter

```

```

 .text(data.love + '%');
 hateCounter
 .text(data.hate + '%');
 });
 });
</script>

```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour14/example04` 找到。要分析来自 Twitter 流 API 的数据，请按如下步骤进行。

1. 创建一个名为 `percentage` 的新文件夹。
2. 在 `percentage` 文件夹中，创建一个名为 `package.json` 的新文件并加入如下内容将 `ntwitter`、`Express` 和 `Socket.IO` 声明为依赖模块：

```

{
 "name": "socket.io-twitter-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "ntwitter": "0.2.10",
 "socket.io": "0.8.7"
 }
}

```

3. 在 `percentage` 文件夹中，创建一个带有如下内容的、名为 `app.js` 的新文件。记得将钥匙和密码替换为你自己的：

```

var app = require('express').createServer(),
 twitter = require('ntwitter'),
 io = require('socket.io').listen(app),
 love = 0,
 hate = 0,
 total = 0;

app.listen(3000);

var twit = new twitter({
 consumer_key: 'YOUR_CONSUMER_KEY',
 consumer_secret: 'YOUR_CONSUMER_SECRET',
 access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
 access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function(data) {
 var text = data.text.toLowerCase();
 if (text.indexOf('love') !== -1) {
 love++;
 total++;
 }
 if (text.indexOf('hate') !== -1) {

```

```

 hate++;
 total++;
 }
 io.sockets.volatile.emit('tweet', {
 user: data.user.screen_name,
 text: data.text,
 love: (love/total)*100,
 hate: (hate/total)*100
 });
});
});
});

app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});

```

4. 在 percentage 文件夹中, 创建一个名为 index.html 的文件并加入如下内容:

```

<!doctype html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.IO Twitter Example</title>
 </head>
 <body>
 <h1>Socket.IO Twitter Example</h1>
 <ul class="percentage">
 <li class="love">0
 <li class="hate">0

 <ul class="tweets">
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect();
 jQuery(function ($) {
 var tweetList = $('ul.tweets'),
 loveCounter = $('li.love'),
 hateCounter = $('li.hate');
 socket.on('tweet', function (data) {
 tweetList
 .prepend('' + data.user + ': ' + data.text + '');
 loveCounter
 .text(data.love + '%');
 hateCounter
 .text(data.hate + '%');
 });
 });
 </script>
 </body>
</html>

```

5. 从终端运行如下命令安装依赖模块:

```
npm install
```

6. 从终端运行如下命令启动服务器:

```
node app.js
```

7. 打开浏览器浏览 <http://127.0.0.1:3000>。
8. 可在浏览器中看到推文流，还有动态更新的百分数（见图 14.7）。
9. 在终端上按 **Ctrl+C** 杀死服务器。

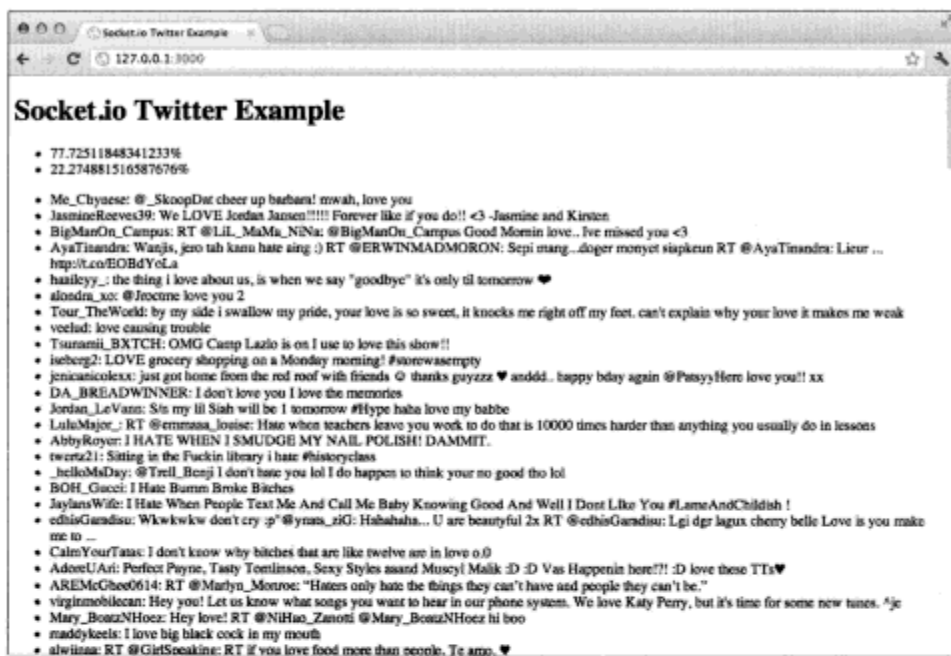


图 14.7  
动态更新百分数值

### 14.6.1 添加实时图形

本应用程序现在可以回答我们的问题了。好棒！只是从视觉上说，它仍旧只是数据。如果应用程序能生成一个小的、能够根据所接收到的数据动态移动的条图，将会是多么棒的事情啊。服务器已经在把数据发送给浏览器了，于是我们完全可以使用客户端的 JavaScript 和一些 CSS 来实现它。应用程序有一个包含百分数的不排序列表，它是用来创建简单条图的完美选择。我们将稍微修改一下不排序列表以便让它更容易调整样式。这里唯一增加的就是将数字包装在一个 `span` 标记中：

```
<ul class="percentage">
 <li class="love">
 0

 <li class="hate">
 0


```

可在 HTML 文档的头部加入一些 CSS，让不排序列表看起来像条图。列表条目表示条图，其中粉色的表示爱，黑色的表示恨：

```
<style>
ul.percentage { width: 100% }
ul.percentage li { display: block; width: 0 }
ul.percentage li span { float: right; display: block}
```

```

 ul.percentage li.love { background: #ff0066; color: #fff}
 ul.percentage li.hate { background: #000; color: #fff}
</style>

```

最后，用一些客户端的 JavaScript 让条（列表项目）的大小可根据从服务器所接收到的百分数值动态调整：

```

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></
<script>
<script src="/socket.io/socket.io.js"></script>
<script>
 var socket = io.connect();
 jQuery(function ($) {
 var tweetList = $('ul.tweets'),
 loveCounter = $('li.love'),
 hateCounter = $('li.hate'),
 loveCounterPercentage = $('li.love span'),
 hateCounterPercentage = $('li.hate span');
 socket.on('tweet', function (data) {
 loveCounter
 .css("width", data.love + '%');
 loveCounterPercentage
 .text(Math.round(data.love * 10) / 10 + '%');
 hateCounter
 .css("width", data.hate + '%');
 hateCounterPercentage
 .text(Math.round(data.hate * 10) / 10 + '%');
 tweetList
 .prepend('' + data.user + ': ' + data.text + '');
 });
 });
</script>

```

每当从 Socket.IO 接收到新的推文事件时，通过使用从服务器接收的百分数值设置列表项目的 CSS 宽度，就可动态更新条图。于是每次接收到新的推文事件时，就有了调整图形的效果。这就创建了一个实时的图形！

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour14/example05` 找到。按下列步骤来可视化实时数据。

1. 创建一个名为 `realtime_graph` 的新文件夹。
2. 在 `realtime_graph` 文件夹中，创建一个名为 `package.json` 的新文件并加入如下内容将 `ntwitter`、`Express` 和 `Socket.IO` 声明为依赖模块：

```

{
 "name": "socket.io-twitter-example",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4",
 "ntwitter": "0.2.10",
 "socket.io": "0.8.7"
 }
}

```



3. 在 `realtime_graph` 文件夹中, 创建一个带有如下内容的、名为 `app.js` 的新文件。记得将钥匙和密码替换为你自己的:

```
var app = require('express').createServer(),
 twitter = require('ntwitter'),
 io = require('socket.io').listen(app),
 love = 0,
 hate = 0,
 total = 0;

app.listen(3000);

var twit = new twitter({
 consumer_key: 'YOUR_CONSUMER_KEY',
 consumer_secret: 'YOUR_CONSUMER_SECRET',
 access_token_key: 'YOUR_ACCESS_TOKEN_KEY',
 access_token_secret: 'YOUR_ACCESS_TOKEN_KEY'
});

twit.stream('statuses/filter', { track: ['love', 'hate'] }, function(stream) {
 stream.on('data', function (data) {

 var text = data.text.toLowerCase();
 if (text.indexOf('love') !== -1) {
 love++
 total++
 }
 if (text.indexOf('hate') !== -1) {
 hate++
 total++
 }

 io.sockets.volatile.emit('tweet', {
 user: data.user.screen_name,
 text: data.text,
 love: (love/total)*100,
 hate: (hate/total)*100
 });
 });
});

app.get('/', function (req, res) {
 res.sendfile(__dirname + '/index.html');
});
```

4. 在 `realtime_graph` 文件夹中, 创建一个名为 `index.html` 的文件并加入如下内容:

```
<!doctype html>
<html lang="en">
 <head>
 <meta charset="utf-8">
 <title>Socket.IO Twitter Example</title>
 <style>
 ul.percentage { width: 100% }
```

```

 ul.percentage li { display: block; width: 0 }
 ul.percentage li span { float: right; display: block}
 ul.percentage li.love { background: #ff0066; color: #fff}
 ul.percentage li.hate { background: #000; color: #fff}
 </style>
</head>
<body>
 <h1>Socket.IO Twitter Example</h1>
 <ul class="percentage">
 <li class="love">
 Love 0

 <li class="hate">
 Hate 0

 <ul class="tweets">
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
 min.js"></script>
 <script src="/socket.io/socket.io.js"></script>
 <script>
 var socket = io.connect();
 jQuery(function ($) {
 var tweetList = $('ul.tweets'),
 loveCounter = $('li.love'),
 hateCounter = $('li.hate'),
 loveCounterPercentage = $('li.love span'),
 hateCounterPercentage = $('li.hate span');
 socket.on('tweet', function (data) {
 loveCounter
 .css("width", data.love + '%');
 loveCounterPercentage
 .text(Math.round(data.love * 10) / 10 + '%');
 hateCounter
 .css("width", data.hate + '%');
 hateCounterPercentage
 .text(Math.round(data.hate * 10) / 10 + '%');
 tweetList
 .prepend('' + data.user + ': ' + data.text + '');
 });
 });
 </script>
</body>
</html>

```

5. 从终端运行如下命令安装依赖模块:

```
npm install
```

6. 从终端运行如下命令启动服务器:

```
node app.js
```

7. 打开浏览器浏览 <http://127.0.0.1:3000>。

8. 可在浏览器中看到推文流，还有根据所接收到的数据更改大小的实时图形（见图 14.8）。

9. 在终端上按 Ctrl+C 杀死服务器。

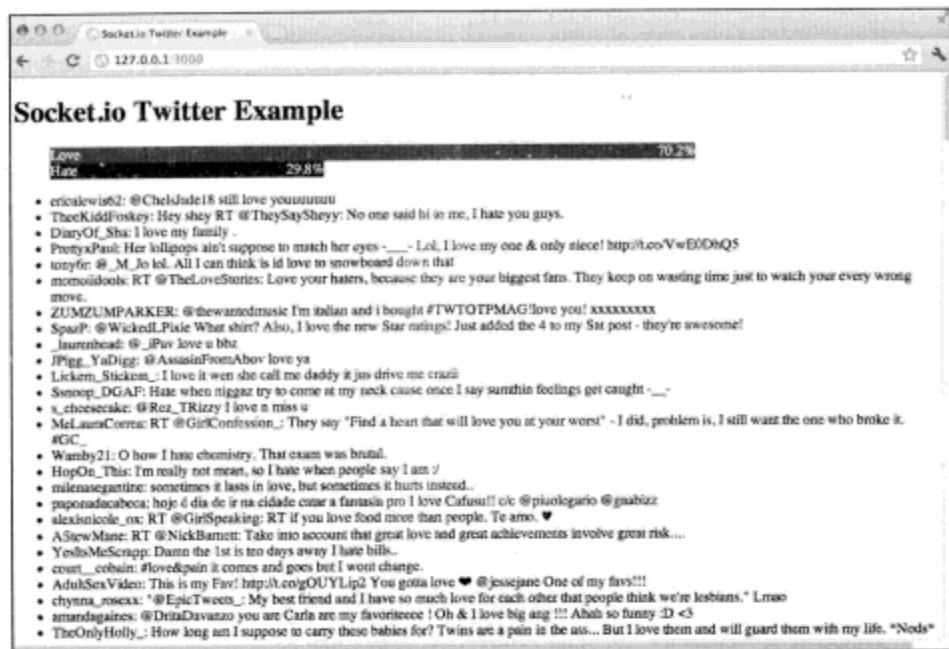


图 14.8

实时图形

我们所创建的这个应用程序根据来自 Twitter 的实时数据，给出了世界上是爱更多还是恨更多的可视化呈现。诚然这完全是不科学的，但它的确展示了 Node.js 和 Socket.IO 接收大量数据并将其推送到浏览器的能力。如果在 CSS 上多做一些的工作，应用程序就可以通过更改样式变得更好看（见图 14.9）。

如果读者想自己运行这一示例，这个版本可在本书代码的 `hour14/example06` 中获得。

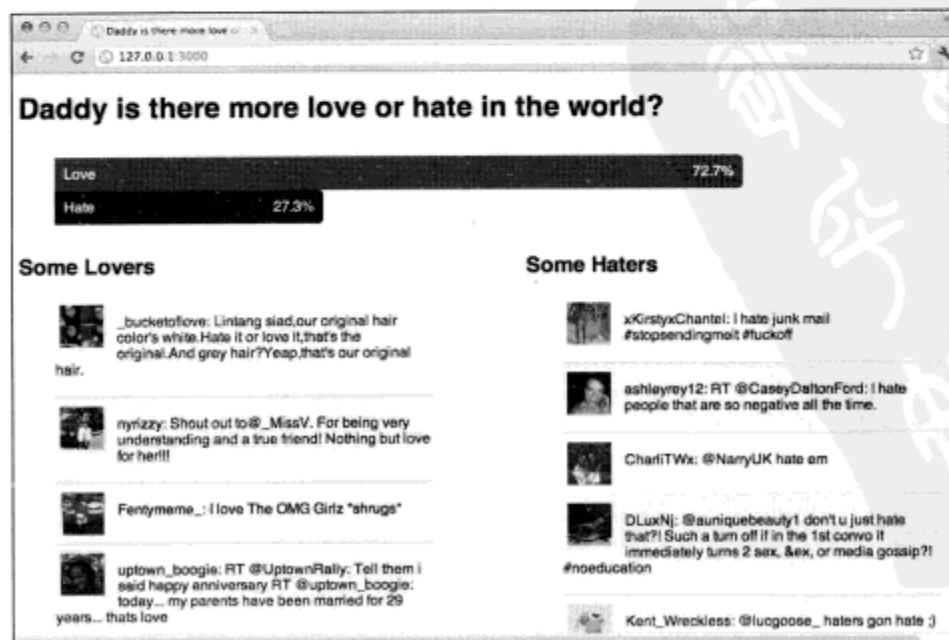


图 14.9

带有更多样式的完成的应用程序

## 14.7 小结

在本章，我们使用 Node.js、Twitter 和 Socket.IO 回答了一个关于人类本性的基本问题。本章的工作还不错！在本书编写的时候，显然世界上的爱更多，所以，本章的内容至少应该让你觉得心情愉快！我们学习了 Node.js 服务器如何从第三方服务接收大量数据并使用 Socket.IO 实时将其推送给浏览器。我们看到了如何操纵数据以便从中挖掘含义，并且对数据执行简单计算来获得百分数的方法。最后，我们添加了一些客户端的 JavaScript 来接收数据并创建实时图形。本章展示了许多 Node.js 的力量所在，包括很容易实现数据在服务器和浏览器之间发送、处理大量数据的能力以及对联网的强大支持。

## 14.8 问与答

问：是否还有其他的流 API 可以用来创建像这样的应用程序？

答：是的。开发人员可使用的流 API 越来越多。在本书编写的时候，一些相关的 API 有 Campfire、Salesforce、Datasift 和 Apigee，还会有许多 API 会被创建。

问：此数据有多精确？

答：不怎么样。此数据建立在 Twitter 流 API 的 “statuses/filter” 方法的基础之上。更多关于在这个源（feed）中会有什么信息请见 <https://dev.twitter.com/docs/streaming-api/methods>。简而言之，别将人类学的研究建立在此基础上。

问：我能将数据保存起来么？

答：本章创建的应用程序不在任何地方保持数据，所以如果服务停止的话，计数器和百分数就会被重置。显然，数据收集的时间越长，结果就越精确。可以扩展本应用程序，让它将计数器储存到可以处理大容量写入操作的数据存储中，比如 Redis。但这超过了本章的范畴了！

## 14.9 测验

本测验包含一些问题，可帮助读者巩固本章所学的知识。

### 14.9.1 问题

1. 流 API 有什么不同？
2. 什么是 OAuth？
3. 为什么 Node.js 很适合流 API？

### 14.9.2 答案

1. 流 API 保持客户端和服务端之间的连接，能够在有新数据的时候将新数据推送给客

户端。这就让应用程序可以是实时的，因为在数据可用时可以尽可能快地被推送到客户端。

2. OAuth 是在无需暴露用户证书的前提下授予应用程序数据访问权的方法。授权按应用程序给予，可以在任何时候撤销。如果读者使用过其他服务连接 Twitter 账户，应该对允许其他服务访问数据感觉熟悉。OAuth 用于实现这个功能。

3. 由于 Node.js 围绕着事件化的 I/O 设计，所以它对接收来自流 API 的新数据可以有非常好的响应。它可处理大量数据而无需大量内存。由于 Node.js 是 JavaScript，它可以很容易地与浏览器这样的能理解 JSON 的客户通信。Node.js 可接收、处理以及发送大量数据事件而无需许多机器来处理它们。

## 14.10 练习

1. 修改本书代码示例中的 `hour14/example02` 示例，让它显示用户的真实名称和 URL。请参阅本章早些时候的数据结构，以便理解做这个事情需要用到哪些属性。

2. 修改服务器，让它按照你所感兴趣的一些关键字从 Twitter 的流 API 接收数据。如果超过两个关键字，请更新应用程序，让它在图形中显示超过两个条图。

3. 思考一下如何创建应用程序来提供不同的流 Twitter 数据集的可视化结果。还记得我们可以按地点、特定用户和关键字来限制查询吗？以下是一些示例。

- 在伦敦，人们谈论更多的是啤酒还是红酒？
- 名人多久使用一次“我”或者“你”这些词？
- 披头士乐队比滚石乐队更流行吗？



## 第 15 章

# JSON API

---

在本章中你将学到：

- API 是什么；
- JSON 是什么，如何在 Node.js 中使用它；
- 从第三方 API 检索数据；
- 创建一个简单的 JSON API。

### 15.1 API

应用程序编程接口（API）是在不同软件组件之间创建接口的方式。在 Web 上，这通常意味着让网站或者服务上的数据可以被第三方或者移动设备、平板设备上的应用程序访问。

Node.js 极为适合于创建 API，原因如下所示。

- 它能够处理大量并发连接，并且内存用得少。
- 它以 JavaScript 写成，所以创建 JSON（JavaScript 对象标记）很容易。
- 在 Node.js 中对 HTTP 一流的支持让创建 API 变得简单。

API 遵循与 Web 浏览器和 Web 服务器所遵循的同样的客户端与服务器模型。考虑一下如下用户在浏览器中装载 Web 页面的情况。

- 用户使用浏览器（或客户端）请求 Web 页面。
- 服务器将 Web 页面以及任何与之有关的媒体（图片、视频等）返回给浏览器。
- 浏览器为用户渲染 Web 页面。

使用 API 的过程是一样的，但用户感兴趣的只有数据，所以服务器将以所请求的格式返

回数据。在使用 API 时，我们经常会听到“客户端”和“服务器”这样的术语。客户端指的是与 API 交互或者从 API 获取数据的东西。它可以是 iPhone 或者 Android 设备上的移动应用。服务器指的是按请求发送数据给客户端的东西。Node.js 既可用于创建 API 客户端也可以用于创建服务器。

创建 API 服务器可能的理由有：

- 公开数据，允许其他开发人员使用；
- 让移动电话、平板和公告牌等设备能够与服务交互。

创建 API 客户端可能的理由有：

- 编写软件与诸如 Twitter 或者 Facebook 这样的第三方服务交互；
- 使用第三方服务自动完成任务；
- 将外部数据集成到产品中。

## 15.2 JSON

JavaScript 对象标记 (JSON) 是种轻量级的数据交换格式。它由 Douglas Crockford 发明，旨在用来作为扩展标记语言 (XML) 的一种替代。历史上，XML 是一种受欢迎的数据交换方式。但对于在 Web 的上下文中进行数据交换的要求，人们觉得 XML 缺乏创建数据结构的能力而且对于手边的工作而言过于重量级了。最初的 XML 被设计用来交换文档。JSON 则特别被设计用来做数据交换。对于许多 Web 开发人员来说，JSON 比起 XML 有极大的优势，比如：

- 容易阅读；
- 容易分析 (parse)，尤其如果使用 JavaScript 的话；
- 无论复杂或简单的数据结构都适合；
- 是数据交换的优秀格式；
- 大多数语言中都有支持良好的分析器 (parser) 可用；
- 通常比 XML 尺寸小。

在最近几年，JSON 以 Web 开发人员首选的数据交换格式出现，许多大型 Web 业务，比如 Twitter、Flickr 和 Facebook，都创建了 JSON API 允许第三方开发人员丰富地访问数据。开发人员开始使用 Ajax 从第三方服务请求数据时，JSON 开始成为他们的选择。使用 JavaScript 将 JSON 作为数据格式来消费要比使用 XML 简单得多，所以 JSON 越来越受欢迎。

今天，单页 Web 应用程序越来越流行。这些应用程序大量使用 JavaScript 和 Ajax 来获取数据并将其显示在页面上。Web 技术的发展方向指向一件事情——JavaScript 将变得更为重要、JSON 将成为数据 API 的首选数据格式。在这样的上下文中，Node.js 是创建 API 的理想选择。由于 Node.js 基于 JavaScript，所以它容易创建、发送以及消费 JSON 数据。如果需要创建基于 JSON 的 API，就应当选择 Node.js！

JSON 看起来像这样：

```
{
 "name": "Darth Vader",
 "occupation": "Dark Lord of the Sith",
 "home": "Tatooine"
}
```

它是 JavaScript 的一个子集，如果读者懂一点 JavaScript 的话应当不陌生。在左括号之后是一个键，然后是与该键相关联的值。值可以是以下类型中的任何类型。

- String（字符串）。
- Number（数值）。
- Object（对象）。
- Array（数组）。
- True or false（真或假）。
- Null。

键/值对之间用逗号分开，键/值对可以嵌套，以便创建更为复杂的数据结构。考虑一下从 Twitter API 响应的 JSON 示例的简化版：

```
{
 "name": "Twitter API",
 "description": "The Real Twitter API. I tweet about API changes, service issues and happily answer questions about Twitter and our API. Don't get an answer? It's on my website.",
 "time_zone": "Pacific Time (US & Canada)",
 "profile_background_image_url": "http://a3.twimg.com/profile_background_images/59931895/twitterapi-background-new.png",
 "friends_count": 20,
 "statuses_count": 2404,
 "status": {
 "coordinates": null,
 "created_at": "Wed Dec 22 20:08:02 +0000 2010",
 "id_str": "17672734540570624",
 "text": "Twitter downtime - Twitter is currently down. We are aware of the problem and working on it. http://t.co/37z12jI",
 "retweet_count": 30,
 "in_reply_to_status_id_str": null,
 },
 "following": true,
 "screen_name": "twitterapi"
}
```

status 键展示了嵌套数据和创建更为复杂的结构的方法。JSON 可按任何想要的方式创建数据结构，而且大多数开发人员发现 JSON 要比 XML 易读得多。

**Watch  
Out!**

**警告：键和字符串必须位于双引号内**

JSON 的键和字符串必须位于双引号内才是有效的。如果使用一个模块来生成 JSON，模块应该已经处理了这个要求。如果手工编写 JSON，那么要记得使用双引号。更多信息请见 <http://simonwillison.net/2006/oct/11/json/>。



## 15.3 使用 Node.js 发送 JSON 数据

为了使用 Node.js 以 JSON 发送数据，必须将头（header）设置为 `application/json`。这个头告诉客户端所发送的是 JSON，这是重要的，因为许多客户端照此决定如何使用数据：

```
var http = require('http');
http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'application/json'});
 res.end('{ "name": "Darth Vader", "occupation": "Dark Lord of the Sith", "home": "Tatooine" }');
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour15/example01` 找到。按照下列步骤使用 Node.js 发送 JSON。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var http = require('http');
http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'application/json'});
 res.end('{ "name": "Darth Vader", "occupation": "Dark Lord of the Sith", "home": "Tatooine" }');
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

2. 运行脚本：

```
node app.js
```

3. 使用 Google Chrome 打开 `http://127.0.0.1:3000`。
4. 单击 HTTP Header 图标（我们在第 8 章中学习了如何安装它），了解这是 JSON。
5. 可看到 Content-Type 正确地设置为了 `application/json`（见图 15.1）。

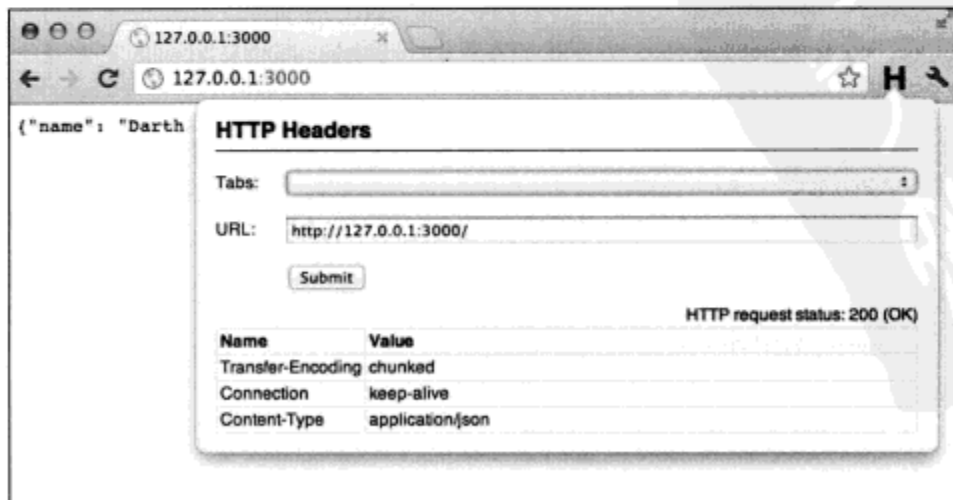


图 15.1  
使用 Node.js 发送 JSON

## 15.4 从 JavaScript 对象创建 JSON

通常情况下，我们可能会使用作为 JSON 发送的 JavaScript 对象。以下是一些我们可能想要返回的东西的示例。

- 数据库记录。
- 在服务器端计算的数据。
- 在服务器端对不同数据来源组合之后的数据组合。

在 Node.js 中，可以使用 `JSON.stringify` 从 JavaScript 对象创建 JSON。它可很容易地用来将 JavaScript 对象转换成 JSON。比如我们有个简单的 JavaScript 对象：

```
var obj = {
 name : "Officer",
 surname : "Dibble"
}
```

使用 `JSON.stringify` 很容易就可将其转换为 JSON：

```
JSON.stringify(obj);
```

通过使用这一方法，而后就可按照先前的方法发送 JSON：

```
var http = require('http');
var obj = {
 name : "Officer",
 surname : "Dibble"
}
http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'application/json'});
 res.end(JSON.stringify(obj));
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

**Did you know?**

**提示：**在大多数浏览器中也可以使用 `JSON.stringify`

在大多数浏览器中也可以使用 `JSON.stringify` 将 JavaScript 对象转换成 JSON。从 Internet Explorer 7 开始就具备这一功能，在移动浏览器中也得到良好支持。完整的浏览器支持矩阵请看 <http://caniuse.com/json>。

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour15/example02` 找到。按照下列步骤将 JavaScript 对象转换为 JSON。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var http = require('http');
var obj = {
 name : "Officer",
 surname : "Dibble"
}
http.createServer(function (req, res) {
```

```
res.writeHead(200, {'Content-Type': 'application/json'});
res.end(JSON.stringify(obj));
}).listen(3000, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3000/');
```

2. 运行脚本:

```
node app.js
```

3. 使用 Google Chrome 打开 `http://127.0.0.1:3000`。

4. 单击 HTTP Header 图标 (我们在第 8 章学习了如何安装它), 了解这是 JSON (见图 15.2)。

5. 可看到数据以 JSON 字符串形式发送给了浏览器。

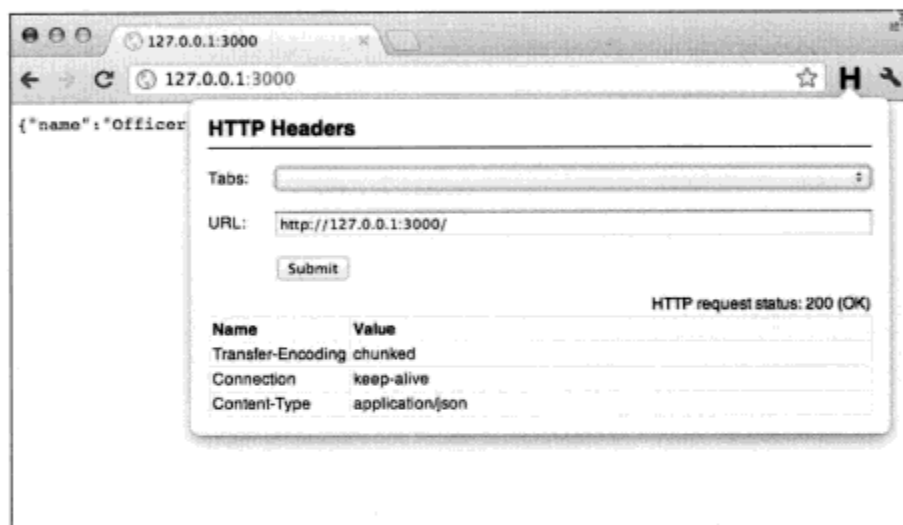


图 15.2

将 JavaScript 对象  
转换成 JSON

## 15.5 使用 Node.js 消费 JSON 数据

既然已经创建了简单的基于 JSON 的 Web 服务, 现在可以编写客户端来消费数据并对数据做一些工作了。我们使用标准的 Node.js HTTP 客户端并使用 `JSON.parse` 来分析我们所接收的数据。

`JSON.parse` 方法分析 JSON 字符串, 重新构建原来的 JavaScript。我们扩展一下早先的示例, 让它先将 JavaScript 对象转换成 JSON 然后再转换回来, 就可以看到这个方法的运行结果了:

```
var obj = {
 name : "Officer",
 surname : "Dibble"
}

console.log('JavaScript object:');
console.log(obj);

var json = JSON.stringify(obj);
console.log('JavaScript object to JSON:');
console.log(json);

var parsedJson = JSON.parse(json);
console.log('JSON to JavaScript object:');
console.log(parsedJson);
```

这个示例表明，将 JavaScript 对象转换成 JSON 然后再转换回来有多么容易。这就是 JavaScript 开发人员喜欢使用 JSON 作为数据格式的原因！

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour15/example03` 找到。按照下列步骤来分析 JSON 数据。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var obj = {
 name : "Officer",
 surname : "Dibble"
}

console.log('JavaScript object:');
console.log(obj);

var json = JSON.stringify(obj);
console.log('JavaScript object to JSON:');
console.log(json);

var parsedJson = JSON.parse(json);
console.log('JSON to JavaScript object:');
console.log(parsedJson);
```

2. 运行脚本：

```
node app.js
```

3. 可看到数据在终端中先被记录为 JavaScript 对象，而后是 JSON，然后回到 JavaScript 对象。
- 

这个示例是理论上的，但读者很快就会看到在使用第三方数据提供商的时候使用 Node.js 来消费 JSON 有多么容易。Node.js 让数据的获取变得容易，而 `JSON.parse` 让数据的使用变得容易。在以下示例中，创建了一个简单的 HTTP 客户端来查询 Twitter 搜索 API 关于 Node.js 的最新推文。它返回 JSON，而后就可以分析、使用：

```
var http = require('http');
var data = "";
var tweets = "";

var options = {
 host: 'search.twitter.com',
 port: 80,
 path: '/search.json?q=%23node.js'
};

var request = http.get(options, function(res){
 res.on('data', function(chunk){
 data += chunk
 });
 res.on('end', function(){
 tweets = JSON.parse(data);
```

```

 for (var i=0; i<tweets.results.length; i++) {
 console.log(tweets.results[i].text)
 }
 });
 res.on('error', function(e){
 console.log("There was an error: " + e.message);
 });
})

```

在本示例中，当来自 Twitter 的响应完成时，就使用 `JSON.parse` 来分析原始数据。而后在循环中将每个推文输出到终端。在实际的应用程序中，我们可能将此数据保存到数据库或者发送到浏览器中。

这段小脚本是展示从 API 检索数据有多么强大的好示例。这段脚本可以扩展一下，将数据保存起来，以便在以后的日子分析或者动态设置搜索条件。只需几行代码，就可以让我们的应用程序用上 Twitter 的搜索结果。

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour15/example04` 找到。按照下列步骤分析来自 Twitter 的 JSON 数据。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```

var http = require('http');
var data = "";
var tweets = "";

var options = {
 host: 'search.twitter.com',
 port: 80,
 path: '/search.json?q=%23node.js'
};

var request = http.get(options, function(res){
 res.on('data', function(chunk){
 data += chunk
 });
 res.on('end', function(){
 tweets = JSON.parse(data);
 for (var i=0; i<tweets.results.length; i++) {
 console.log(tweets.results[i].text)
 }
 });
 res.on('error', function(e){
 console.log("There was an error: " + e.message);
 });
})

```

2. 运行脚本：

```
node app.js
```

3. 在一段短暂停后，可看到终端上输出了最近 15 个关于 Node.js 的推文。
-

**注意：有成千上万个 API！**

Web 上有成千上万个免费的、可公开使用的 API，可以给开发人员提供海量的 JSON 格式数据。有些大型组织通过 API 来发布数据，包括美国和英国政府以及世界银行。我们还可从诸如 Rotten Tomatoes 这样的站点获取数据满足对电影和视频的需要、从 Google Maps 获取地图、从 GitHub 获取源代码，等等等等。我们有无穷无尽的可能性！

## 15.6 使用 Node.js 创建 JSON API

既然对 JSON 有了更多了解，而且知道了如何在 Node.js 中使用它，我们现在要看看如何创建基于 JSON 的 API 以便其他开发人员能与我们的服务交互。我们修改在第 8 章所创建的 Express 应用程序，让它变成基于 JSON 的 API 并且让数据可以在浏览器之外使用。

在第 8 章创建的应用程序具备如下功能。

- 能够创建、更新、删除并且读取任务。
- 任务储存在 MongoDB 中。
- 可以从浏览器更新任务。

这很不错。不过要是有人想创建 Android 或者 iPhone 应用程序该怎么办？也可能你希望有一个第三方软件可以更新你的任务列表？比如，如果你在 GitHub 项目中记录问题，就可能希望任何加入到项目中的问题都会自动加入到你的任务列表中。这就是需要 API 的地方！

### 15.6.1 在 Express 中以 JSON 发送数据

Express 对 JSON 有极佳的支持，并且要比 Node.js 的标准库中的 HTTP 模块方便。使用 Express 发送 JSON 数据有两种方法，它们都可为我们转换好 JSON 并设置好头。第一种方法是 `res.send()`：

```
var ingredients = [
 { name: 'eggs' },
 { name: 'flour' },
 { name: 'milk' }
];

app.get('/', function(req, res, next){
 res.send(ingredients);
});
```

要是在这条路线上执行一次螺旋请求（curl request），将得到如下返回：

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 127
Connection: keep-alive

[{"name": "eggs"}, {"name": "flour"}, {"name": "milk"}]
```

这个响应显示，Express 在这里为我们做了如下的许多工作。

- 响应代码设置为 200。
- Content-Type 头正确地设置为 application/json。
- 将 JavaScript 对象转换为 JSON 字符串。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour15/example05` 找到。按照下列步骤使用 Express 来提供 JSON 数据服务。

1. 创建一个名为 `package.json` 的文件并将如下代码复制到其中：

```
{
 "name": "express-json",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.4"
 }
}
```

2. 创建第二个文件，起名 `app.js` 并将如下代码复制到其中：

```
var express = require('express')

var app = module.exports = express.createServer();

var rebels = [
 { name: 'Han Solo' },
 { name: 'Luke Skywalker' },
 { name: 'C-3PO' },
 { name: 'R2-D2' },
 { name: 'Chewbacca' },
 { name: 'Princess Leia' }
];

app.get('/', function(req, res, next){
 res.send(rebels);
});

app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));

app.listen(3000);
console.log("Express server listening on port %d in %s mode", app.address().
 port, app.settings.env);
```

3. 安装依赖模块：

```
npm install
```

4. 运行脚本：

```
node app.js
```

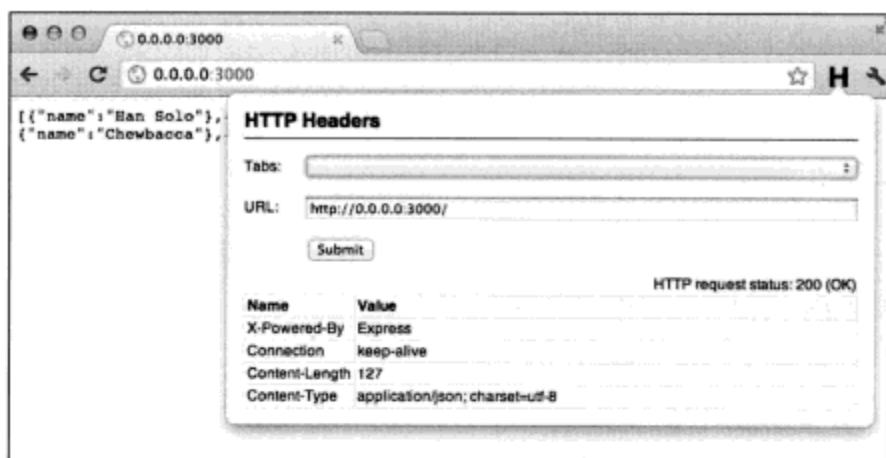
5. 使用 Google Chrome 打开 `http://127.0.0.1:3000`。

6. 可看到以 JSON 表示的 Rebel Alliance 成员列表。可使用 HTTP Headers 扩展插件来检

查头的内容（见图 15.3）。

图 15.3

使用 Express 发送 JSON



告诉 Express 发送 JSON 的第二种方法是使用 `res.json()`。它与 `res.send()` 几乎一模一样：

```
var ingredients = [
 { name: 'eggs' },
 { name: 'flour' },
 { name: 'milk' }
];

app.get('/', function(req, res, next){
 res.json(ingredients);
 res.end();
});
```

它就如 `res.send()` 所做的那样设置响应代码和内容类型并转换成 JSON。那么这两种方法之间的区别在哪儿呢？

在 Express 中，`res.send` 被设计为一个高层的响应工具，它允许我们将所有形式的对象传递给它，这可以是许多东西，比如：

- 空白的响应；
- 一些 JSON；
- 一些 HTML；
- 一些纯文本；
- 什么都没有的 404 响应。

用代码来表示的话就是这样：

```
res.send();
res.send({ greeting: 'OHAI!' });
res.send('<p>some html</p>');
res.send('text', { 'Content-Type': 'text/plain' }, 201);
res.send(404);
```

可以看到，我们可以使用 `res.send` 发送许多不同的东西，而它也足够聪明，能够处理你



想发送的东西。

如果选择使用 `res.json()`，那么就是在显式地说：我想发送 JSON，而不是别的什么东西。如果知道自己要发送的一定是 JSON 或者需要以 JSON 发送一个字符串的话，这会是个好选择。如果给 `res.send()` 方法传递一个字符串，那么它假设你发送的是 HTML。但使用 `res.json()` 可以让我们以 JSON 发送字符串。如果对此感觉困惑，那就只使用 `res.send()`。它几乎总能按你所要的去做！

## 15.6.2 构建应用程序

为了理解使用 Node.js 创建 JSON API 的方法，我们构建一套与在第 8 章中创建的任务应用程序相似的简单 API。用它可以创建、读取、更新和删除记录。按照我们在第 10 章中所学的测试工具的要求，我们遵循行为驱动的开发方法，所以先要编写测试然后编写代码来让测试通过。我们使用和第 8 章中一样的 MongoDB 作为数据存储。

### 警告：确认 MongoDB 的运行

要想按步进行这些示例，需要确认 MongoDB 运行于你的计算机上。如果需要复习一下，请参考第 11 章。

**Watch  
Out!**

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour15/example06` 找到。要使用 Node.js 创建 JSON API，请按如下步骤进行。

1. 创建一个名为 `json_api` 的新文件夹。
2. 在新文件夹中，创建一个名为 `app.js` 的文件并输入如下内容。这会设置好一个使用 Mongoose 模型的基础 Express 应用程序以及开发和测试环境：

```
var express = require('express');
var mongoose = require('mongoose');

var Schema = mongoose.Schema;
var ObjectId = Schema.ObjectId;

var Task = new Schema({
 task : {
 type: String,
 required: true,
 },
 created_at: { type: Date, default: Date.now },
 updated_at : Date
});

var Task = mongoose.model('Task', Task);

var app = module.exports = express.createServer();

app.configure(function() {
```

```

 app.use(express.bodyParser());
 app.use(express.methodOverride());
 app.use(app.router);
 });

 app.configure('development', function() {
 app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
 mongoose.connect('mongodb://localhost/todo_development');
 app.listen(3000);
 });

 app.configure('test', function() {
 app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
 mongoose.connect('mongodb://localhost/todo_test');
 app.listen(3001);
 });

 console.log("Express server listening on port %d in %s mode", app.address().
 ➤ port, app.settings.env);

```

3. 在 `json_api` 文件夹中，创建一个名为 `pacakge.json` 的文件并输入如下内容：

```

{
 "name": "json_api",
 "version": "0.0.1",
 "private": true,
 "dependencies": {
 "express": "2.5.3",
 "mongoose": ">= 2.3.1"
 },
 "scripts": {
 "test": "./node_modules/.bin/mocha"
 },
 "devDependencies": {
 "mocha": "0.3.x"
 }
}

```

4. 在 `json_api` 文件夹中，创建一个名为 `test` 的文件夹。

5. 最后，安装依赖模块，这一切就设置好了！

```
npm install
```

遵循行为驱动的开发方法，首先要为 API 能做的事情编写测试。我们所要的第一件事是一个能返回所有任务的路由。这可在 `/api/v1/tasks` 访问。它的行为可按如下描述清楚。

➤ 它应该返回一个 200 响应。

➤ 它应该以 JSON 返回任务。

在 Mocha 中，应该是这样：

```

describe('api v1', function() {
 describe('GET /api/v1/tasks', function() {
 it('should return at 200 response code')
 it('should return JSON')
 })
})

```

没有回调的话，mocha 测试在有效地通过之后会保持等待状态。在考虑 API 时，这是很有用的做法。通过使用这种技术，我们在编写代码之前首先描述 API 的行为。对于许多开发人员而言，带来的是对这个软件该如何工作的更为清晰的定义。

一旦描述了 API 该如何工作，就可以编写一些测试来确保它的确按期望的方式工作：

```
var http = require('http');
var assert = require('assert');
var app = require('../app.js');

describe('api v1', function(){
 describe('GET /api/v1/tasks', function(){
 it('should return a 200 response', function(done){
 http.get({ path: '/api/v1/tasks', port: 3001 }, function(res){
 assert.equal(res.statusCode,
 200,
 'Expected: 200 Actual: ' + res.statusCode);
 done();
 })
 })
 it('should return JSON', function(done){
 http.get({ path: '/api/v1/tasks', port: 3001 }, function(res){
 assert.equal(res.headers["content-type"],
 "application/json; charset=utf-8",
 'Expected: application/json; charset=utf-8 Actual: ' + res.
 headers["content-type"]);
 done();
 })
 })
 })
})
```

在这两个测试中，我们使用 Node.js 标准库中的 assert 模块首先检查响应代码是否是 200 以及返回的是 JSON。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour15/example07 找到。要使用 Mocha 测试 JSON API，请按如下步骤进行。

1. 返回测试文件夹中的 json\_api 项目，创建一个名为 tasks.js 的文件并加入如下内容：

```
var http = require('http');
var assert = require('assert');
var app = require('../app.js');

describe('api v1', function(){
 describe('GET /api/v1/tasks', function(){
 it('should return a 200 response', function(done){
 http.get({ path: '/api/v1/tasks', port: 3001 }, function(res){
 assert.equal(res.statusCode,
 200,
 'Expected: 200 Actual: ' + res.statusCode);
 done();
 })
 })
 })
})
```

```

 })
 it('should return JSON', function(done){
 http.get({ path: '/api/v1/tasks', port: 3001 }, function(res){
 assert.equal(res.headers["content-type"],
 "application/json; charset=utf-8",
 'Expected: application/json; charset=utf-8 Actual: ' + res.
 headers["content-type"]);
 done();
 })
 })
 })
})

```

2. 运行测试。应该看到测试失败：

```
NODE_ENV=test npm test
```

3. 在 app.js 中，就在最后一行之前，加入如下代码：

```

app.get('/api/v1/tasks', function(req, res, next){
 Task.find({}, function (err, docs) {
 res.send(docs);
 });
});

```

4. 运行测试。应该看到测试成功：

```
✓ 2 tests complete
```

这是行为驱动开发的一个好示例。首先，我们描述希望 API 如何动作，然后为其编写一些测试。我们看到这些测试失败，然后给应用程序加入一些代码以便测试通过。这是一个好方法的理由有很多，比如：

- 在编写代码之前我们清晰地定义了希望应用程序如何动作；
- 我们有一组可重复的测试集合，将来添加新功能时还可以使用；
- 一旦测试通过，就能知道应用程序工作正常！

我们创建了一个返回任务列表的路由，不过我们还需要创建、编辑以及删除任务的路由。简洁起见，以下给出了行为驱动开发的描述以及实现代码。

这是创建新任务的路由：

```

describe('POST /api/v1/tasks', function(){
 it('should return at 200 response code on success')
 it('should return JSON')
 it('should return at 422 response code if there is a validation error')
})

```

这是展示单个任务的路由：

```

describe('GET /api/v1/tasks/:id', function(){
 it('should return at 200 response code on success')
 it('should return JSON')
 it('should return at 404 response code if the task doesn't exist')
})

```

这是更新任务的路由：

```
describe('PUT /api/v1/tasks/:id', function(){
 it('should return at 200 response code on success')
 it('should return JSON')
 it('should return at 404 response code if the task doesn't exist')
 it('should return at 422 response code if there is a validation error')
})
```

这是删除任务的路由：

```
describe('DELETE /api/v1/tasks/:id', function(){
 it('should return at 200 response code on success')
 it('should return JSON')
 it('should return at 404 response code if the task doesn't exist')
})
```

想想 API 该如何实现功能。这里是否有其他东西要添加的？完成这个过程也对文档的建立有帮助。在某些点上，我们会需要与其他开发人员共享我们的 API，完成这些步骤可以让文档更容易编写。

这里正常的过程跟我们之前做过的一样，应该是首先为这些描述编写测试，然后编写代码来让测试通过。但为了简洁起见，我们会对所有这些动作进行测试，但我们鼓励读者在本章之外自己探索。

以下是每个路由的实现。它紧密遵循我们在第 8 章所用的技艺。首先，创建一个新任务：

```
app.post('/api/v1/tasks', function(req, res){
 var doc = new Task(req.body.task);
 doc.save(function (err) {
 if (!err) {
 res.send(doc);
 } else {
 res.send(err, 422);
 }
 });
});
```

如我们在第 8 章看到的那样，这会设置一个接收寄送请求的路由，并按这样的方式期待数据：`task[task] = feed the dog`。如果成功保存了任务，就会带着 200 响应以 JSON 对象返回。

这是按 id 获取单个任务的实现：

```
app.get('/api/v1/tasks/:id', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 if (doc) {
 res.send(doc);
 } else {
 res.send(404);
 }
 });
});
```

这会用所提供的 id 在 MongoDB 中搜索一条记录。如果找到，会以 JSON 对象返回。如果没有找到，则发送 404 响应。

这是按 id 更新单个任务的实现：

```
app.put('/api/v1/tasks/:id', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 if (!doc) {
 res.json(404)
 } else {
 doc.updated_at = new Date();
 doc.task = req.body.task.task;
 doc.save(function (err) {
 if (!err) {
 res.send(doc);
 } else {
 res.send(err, 422);
 }
 });
 }
 });
});
```

这会按 id 搜索记录并且按照以 PUT 请求发送过来的数据更新该记录。如果存在验证错误，就返回 422 响应。如果用该 id 没找到文档，则返回 404 响应。

这是删除任务的实现：

```
app.del('/api/v1/tasks/:id', function(req, res){
 Task.findById(req.params.id, function (err, doc){
 if (doc) {
 doc.remove(function() {
 res.send(200)
 });
 } else {
 res.json(404)
 }
 });
});
```

这会按 id 搜索一条记录，如果找到的话就删除它，如果没找到就返回 404 响应。

如果读者下载了本书的代码示例，此 API 的完整代码位于 `hour15/example08`。

这是个简化的 API（没有授权！），但它说明了使用 Node.js 创建 JSON API 有多么简单！

## 15.7 小结

在本章，我们介绍了 API 的思想以及作为一种数据交换格式的 JSON。我们看到了如何使用 Node.js 发送 JSON 数据；看到了使用 Node.js 消费 JSON 数据之容易。我们还介绍了 API 客户端和服务器的思想。首先，通过 Twitter 示例，我们创建了一个简单的 API 客户端来获取最新的 Node.js 推文；而后我们为任务程序创建了一个简单的 API。

在本章，我们只是对 Node.js 处理 API 的能力做了一些表面上的介绍，但希望这些已经足够让读者确信使用 Node.js 创建 API 客户端和服务器的直白又强大的方法。组合了 Node.js 的优秀性能和可扩展性，API 是这一技术的一个优秀的使用案例。

## 15.8 问与答

问：为什么是 JSON 而不是 XML？

答：JSON 在 Web 开发人员中已经非常流行，尤其是使用 JavaScript 的那些人。在使用 Node.js 和 JavaScript 的时候，使用 JSON 作为数据格式是合情合理的。此外，在 Web 的上下文中，JavaScript 现在既被用在服务器端也被用在客户端。如果使用 JavaScript 的话，让数据交换格式保持在 JavaScript 家族中，是合理的做法。它的分析速度更快，用起来更容易。读者会发现 Node.js 社区中多数人喜欢 JSON。如果想使用 XML 或其他数据格式，当然绝对可以。但这些格式不是本章的内容。

问：JSON 比 XML 小而快吗？

答：通常 JSON 的数据量要比 XML 小。许多服务器在发送数据给客户端前会进行压缩，所以实际上数据大小的差异可以忽略。

问：我可以使用第三方 API 所发布的数据吗？

答：大体上，答案是可以。尤其对于公共数据（比如政府和公共实体），数据有效地由纳税人所拥有，所以属于公共域。许多组织发现，通过将它们的数据以 API 发布，有许多很令人惊奇的东西都被创建了出来。有些 API 有授权限制，尤其是围绕着商业使用的限制。如果有疑问，就审视一下 API 的授权信息，或者直接与 API 提供商接洽。

## 15.9 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

### 15.9.1 问题

1. 我们为什么要创建 API？
2. 相对于 XML，JSON 有哪些优势？
3. 如果手工编写 JSON 的话，如何确保它是有效的？

### 15.9.2 答案

1. 创建 API 是因为想提供对数据的访问能力，以便数据能够得到使用。它可以是私有 API，表示只有你或者你允许访问的人才能使用它。它也可以是公共 API，表示任何人都可以使用它。
2. 如果使用 Node.js 的话 JSON 是个很棒的选择，因为分析 JSON 很容易。它是一种可以处理复杂数据结构的轻量级数据格式，而且也得到浏览器的良好支持。
3. 许多在线工具都可用来快速验证 JSON。这些工具可确保手写的 JSON 不至于弄坏脚

本，所以建议验证手写的 JSON。两个流行的站点是：<http://jsonlint.com/>和 <http://jsonformatter.curiousconcept.com/>。

## 15.10 练习

1. 访问 [www.programmableweb.com/apis/directory](http://www.programmableweb.com/apis/directory) 并浏览 API 清单。将数据格式过滤成 JSON 并探索这一清单。想象一下使用所有这些数据能做出什么！

2. 编写一个客户端从 GitHub 获取一些 Node.js 库。对如下 URL 执行一个 GET 请求：<https://api.github.com/legacy/repos/search/node>。分析并在响应中输出 JSON。

3. 复习我们所创建的任务 API 中 `test/tasks.js` 下的文件。试着理解以这种方式编写测试的意义，并且完成一个检查 200 响应的测试。如果满怀雄心，请完成整套测试！





## 第5部分 探索 Node.js API

第16章 进程模块

第17章 子进程模块

第18章 事件模块

第19章 缓冲区模块

第20章 流模块



## 第 16 章

# 进程模块

在本章中你将学到：

- 进程是什么；
- 给进程发送信号（signal）；
- 使用 Node.js 创建可执行脚本；
- 向进程传递参数。

### 16.1 进程是什么

只要在计算机上运行东西，就是在运行进程。如果打开一个浏览器，就至少有一个进程，甚至是超过一个进程在让浏览器完成其工作。我们已经知道，Node.js 运行于一个进程中，所以当我们运行 Node.js 程序的时候，它运行于单个进程之上。操作系统给进程指派一个 id，这个 id 有时候也称为 pid 或者进程 id。只要进程活着，这就是指派给进程的一个唯一的数。

有了 Process（进程）模块，Node.js 的开发人员可以了解一个脚本的进程 id。在以下示例中，脚本要做的事只是将进程 id 写到终端上。不需要像其他脚本那样请求 Process 模块，因为所有的信息在现有进程中都已经有了！

```
console.log(process.pid);
```

如果从终端运行这一脚本，就会看到进程 id 打印出来：

```
node process.js
32204
```

多次运行这一脚本，可注意到每次运行的时候就会有新的进程 id 指派给它：

```
node process.js
32634
node process.js
32639
node process.js
32643
```

这似乎没什么大不了的，但要理解每当进程运行的时候会被指派一个唯一的进程 id，这很重要。在本示例中，进程 id 仅被指派了一小段时间。当脚本完成了对进程 id 的打印之后，它就退出了，进程就不再存在了。

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour16/example01` 找到。按照下列步骤来探索进程 id。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：  
`console.log(process.pid);`
2. 运行脚本：  
`node process.js`
3. 注意所显示的进程 id 或者 pid。
4. 多运行几次脚本。
5. 可看到每次运行脚本的时候 pid 数值都会变化（见图 16.1）。

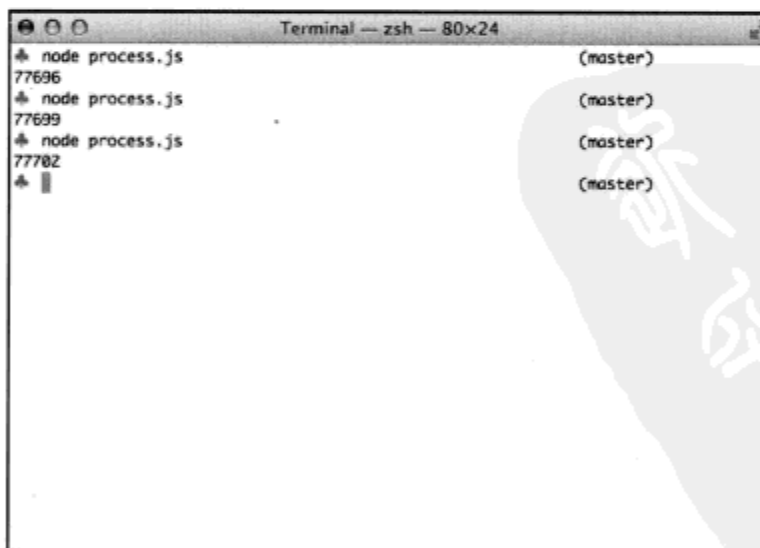


图 16.1

每个进程有一个唯一的进程标识符

**Did you know?**

#### 提示：在计算机上有许多进程在运行

在任何一个时刻，计算机上都有许多进程在运行。如果读者使用 Mac OSX 或者 Linux，打开终端并运行 `ps aux` 命令可看到进程列表。在 Windows 计算机上，打开命令提示符并运行 `tasklist /v` 命令。所有这些在运行的进程都是为了让计算机保持运行以及让你——计算机的用户，保持心情愉快。

## 16.2 退出进程以及进程中的错误

Process 模块给进程的退出提供了一个事件：

```
process.on('exit', function ()
 // Do something when the script exits
});
```

如果当脚本退出时需要执行一些清理操作，比如关闭连接，或者需要记录一些信息的话，这会很有用。Process 模块也提供 “uncaughtException” 事件用于脚本没有处理的异常：

```
process.on('uncaughtException', function (err) {
 console.error(err.stack);
});
```

这个事件的用处之一是，捕获未捕获的异常并将它们发送到某个电子邮件地址上。这样就作为开发人员的你能够了解在应用程序中通过正常的错误处理过程不曾遇到过的问题，于是就能让脚本更稳定健壮。这正是第三方的 node-exception-notifier (node 异常通知) 模块所做的事情 (<https://github.com/saschagehlich/node-exception-notifier/>)，虽然它的源码是以 CoffeeScript 写成的，但它让我们可以了解如何在代码中使用这个事件来发送异常通知。必须提及的是，在这个事件中，代码的执行情况不能完全信赖。

我们也可以选择使用这一事件来通知诸如 Airbrake (<http://airbrake.io/>) 这样的异常日志记录服务。这个服务维护未捕获异常的日志记录以便开发人员审视并解决异常。第三方的 node-airbrake 模块做的就是这件事并且捕获 uncaughtException 事件并将其发送到服务 (<https://github.com/felixge/node-airbrake/>)。

## 16.3 进程与信号

在 UNIX 类型的系统上 (Mac OSX 或者 Linux)，进程可以接收信号。这就让进程可以以某种方式来响应。SIGINT 就是一个最为常见的信号。它是发送中断 (Interrupt) 信号的缩写。我们已经通过按 Ctrl+C 停止 Node.js 进程使用过这个信号了。比如，我们在 Hello World Node.js 服务器中就这么做了。

Process 模块让 Node.js 脚本可以侦听这些信号并据此响应。发出 SIGINT 的意思是要进程停止，但给进程在退出之前清理的机会。

为了演示对 SIGINT 信号的接收，可创建一个短的 Node.js 脚本来展示信号被接收：

```
process.stdin.resume();

process.on('SIGINT', function() {
 console.log('Got a SIGINT. Exiting');
 process.exit(0);
});
```

这段脚本的第一行代码防止脚本在初始化从 stdin 的读取时退出，这样它就不会退出。如果这行代码不存在的话，脚本就会完成，进程结束。在 Node.js 中接收信号遵循我们熟悉的

回调模式：当进程收到信号时调用一个匿名函数。如果运行本脚本然后按键盘上的 Ctrl+C，那么就会给进程发送 SIGINT 信号，脚本在退出前显示接收到 SIGINT：

```
node process.js
[Press Ctrl-C on your keyboard]
Got a SIGINT. Exiting.
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour16/example02` 找到。按照如下步骤在 `Process` 模块中处理 SIGINT 信号。

1. 创建一个名为 `process.js` 的文件并将下列代码复制到其中：

```
process.stdin.resume();

process.on('SIGINT', function() {
 console.log('Got a SIGINT. Exiting');
 process.exit(0);
});
```

2. 运行脚本：

```
node process.js
```

3. 按键盘上的 Ctrl+C。

4. 可看到 “Got a SIGINT. Exiting.”。

在这个简单的示例中，脚本接收到 SIGINT，记录接收到它，然后退出。在进程退出之前清理和进程有关的东西，则是接收 SIGINT 信号更为实际的用法。比如关闭连接、写入日志文件或者甚至对另一个进程做点什么。

## 16.4 向进程发送信号

在 UNIX 类型的系统上，可以使用 `kill` 命令给进程发送信号。需要给 `kill` 命令提供要发送信号的进程 id。默认情况下，它发送 SIGTERM，进程应该立即终止：

```
kill [process_id]
```

`kill` 命令也可发送其他信号。在上一个示例中，我们看到如何通过按键盘上的 Ctrl+C 给进程发送 SIGINT。`kill` 命令也可发送 SIGINT：

```
kill -s SIGINT [process_id]
```

使用 `Process` 模块，就可以设置这些信号的侦听器并据此响应：

```
process.on('SIGINT', function() {
 console.log('Got a SIGINT. Exiting');
 process.exit(0);
});

process.on('SIGTERM', function() {
 console.log('Got a SIGTERM. Exiting');
 process.exit(0);
});
```

```
setInterval(function() {
 // This keeps the process running
}, 10000);

console.log('Run `kill ` + process.pid + ` to send a SIGTERM')
console.log('Run `kill -s SIGINT ` + process.pid + ` to send a SIGINT')
```

在这个示例中，脚本侦听 SIGINT 和 SIGTERM 信号。最后的 `setInterval` 让脚本保持运行。要是没有它，脚本就会退出。如果将此文件保存成 `process.js`，那么就可以用如下命令运行这个脚本：

```
node process.js
```

本脚本输出进程 id，以便在另一个终端中通过使用 `kill` 命令来发送信号。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour16/example03` 找到。以下是在 `Process` 模块中处理其他信号的方法。

1. 创建一个名为 `process.js` 的文件并将如下代码复制到其中：

```
process.on('SIGINT', function() {
 console.log('Got a SIGINT. Exiting');
 process.exit(0);
});

process.on('SIGTERM', function() {
 console.log('Got a SIGTERM. Exiting');
 process.exit(0);
});

setInterval(function() {
 // This keeps the process running
}, 10000);

console.log('Run `kill ` + process.pid + ` to send a SIGTERM')
console.log('Run `kill -s SIGINT ` + process.pid + ` to send a SIGINT')
```

2. 运行脚本：

```
node process.js
```

3. 注意输出的内容并在输出的第一行上运行 `kill` 命令来演示 SIGTERM：

```
kill [process_id]
```

4. 可看到脚本被终止。

5. 在后台再次启动脚本：

```
node process.js &
```

6. 给脚本发送 SIGINT，如运行中的脚本所输出的所示：

```
kill -s SIGINT [process_id]
```

7. 可看到脚本收到 SIGINT 并退出。

**Did you  
know?**

### 提示：退出状态事项

在退出脚本时，给一个正确的退出状态是重要的，因为其他脚本可能与之交互。如果脚本成功执行，它以状态 0 退出。如果以状态 1 退出则说明有错误。`Process` 模块提供了一个简单的方法用于正确退出脚本：`process.exit()`。

## 16.5 使用 Node.js 创建脚本

Process 模块可协助我们使用 Node.js 创建小的脚本。在从终端运行命令时，我们在运行一个可执行文件。被执行的代码可以是二进制代码也可以是用许多脚本语言中的一种编写的脚本。于是，它可以是使用 Node.js 编写的脚本！要创建可以从命令行运行的脚本需要几个步骤。

首先是在脚本的顶部加一个 Node.js 的 shebang。“shebang”这个词读者可能是第一次接触，它就是一行代码，告诉操作系统从哪儿找到用来运行脚本的二进制程序。这也意味着我们可以给文件起任意名称，也不需要.js 扩展名来区分该文件的类型。对于 Node.js 的 shebang 如下：

```
#!/usr/bin/env node
```

在把这个 shebang 放置到脚本的第一行之后，还需要将脚本设置为可执行。在 Mac OSX 或者 Linux 上，可以通过如下命令来实现：

```
chmod +x yourscrip.js
```

准备好了 shebang 并且确认脚本是可执行的之后，就可以这样运行脚本了：

```
./yourscrip.js
```

注意在运行脚本的时候无需加上 node 命令了，因为它封装在了 shebang 中。如果有一段简单的 Node.js 脚本：

```
#!/usr/bin/env node
console.log('my first node script!');
```

如果将这个文件保存成脚本，就可以将其设置为可执行文件，然后运行查看输出：

```
chmod +x script
./script
my first node script!
```

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour16/example04` 找到。按照下列步骤创建一个使用 Node.js 的可执行脚本。

1. 创建一个名为 `script` 的文件并将如下代码复制到其中：

```
#!/usr/bin/env node
console.log('my first node script!');
```

2. 将脚本设置为可执行：

```
chmod +x script
```

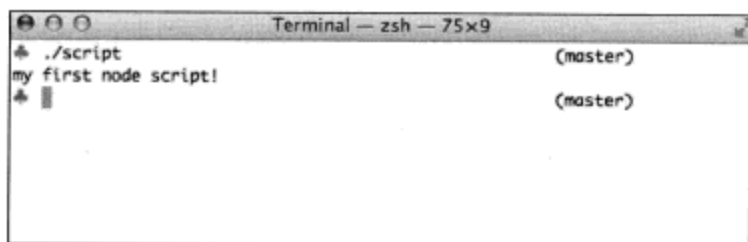
3. 在与脚本相同的目录下运行如下命令执行脚本：

```
./script
```

4. 可看到来自脚本的输出（见图 16.2）。
-

图 16.2

运行一个简单的  
Node.js 脚本



## 16.6 给脚本传递参数

Process 模块支持将参数传递给脚本，参数以 `process.argv` 数组的形式使用。以下脚本将任何参数记录到控制台：

```
#!/usr/bin/env node
console.log(process.argv);
```

如果不带参数运行这段脚本，那么就会看到数组的前两项已经被填为“node”和脚本的路径：

```
['node',
 '/Users/george/script']
```

如果传递参数给脚本，那么这些参数显示在数组中上述元素之后：

```
./script one two three
['node',
 '/Users/george/script'
 'one',
 'two',
 'three']
```

通过使用参数，大大增加了脚本的灵活性和有用性。比如，我们可以编写一个从公共 API 获取数据的脚本。以下示例演示了从 Twitter 的搜索 API 获取搜索结果并在控制台中显示出来的一小段 Node.js 脚本的编写方法：

```
#!/usr/bin/env node

var http = require('http');

if (!process.argv[2]) {
 console.error('A search term is required');
 process.exit(1);
}

var options = {
 host: 'search.twitter.com',
 path: '/search.json?q=' + process.argv[2]
};

http.get(options, function(res) {
 var data = "";
 var json;
 res.on("data", function (chunk) {
 data += chunk;
 });
});
```



```

});
res.on("end", function () {
 json = JSON.parse(data);
 for (i = 0; i < json.results.length; i++) {
 console.log(json.results[i].text)
 }
 process.exit(0);
});
}).on('error', function(e) {
 console.log("Error fetching data: " + e.message);
 process.exit(1);
});
});

```

我们在第 5 章介绍的 HTTP 模块“HTTP”用于从 Twitter 公开的 JSON 搜索 API 获取数据。对于以查询给出的搜索词汇，它以 JSON 数据返回一系列搜索结果。当数据从 Twitter 服务器返回时，脚本分析 JSON 然后循环遍历结果并将文本记录到控制台上。

像这样的脚本很有用，因为计算机可以以规定的时间间隔执行它们，同时人类也可以在终端上执行它们。更多像这样的脚本可以通过其他脚本来粘合，从而带来更为复杂的功能。鉴于 Node.js 在网络编程上的能力，如果需要使用网络来自动完成一些任务，它是一个极佳的选择；而 Process 模块提供了创建脚本所最为需要的工具。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour16/example05` 找到。按照下列步骤在可执行脚本中使用参数。

1. 创建一个名为 `script` 的文件并将如下代码复制到其中：

```

#!/usr/bin/env node

var http = require('http');

if (!process.argv[2]) {
 console.error('A search term is required');
 process.exit(1);
}

var options = {
 host: 'search.twitter.com',
 path: '/search.json?q=' + process.argv[2]
};

http.get(options, function(res) {
 var data = "";
 var json;
 res.on("data", function (chunk) {
 data += chunk;
 });
 res.on("end", function () {
 json = JSON.parse(data);
 for (i = 0; i < json.results.length; i++) {
 console.log(json.results[i].text)
 }
 })
});

```

```

 process.exit(0);
 });
}).on('error', function(e) {
 console.log("Error fetching data: " + e.message);
 process.exit(1);
});

```

## 2. 让脚本可执行:

```
chmod +x script
```

## 3. 从与脚本文件相同的目录下运行脚本:

```
./script
```

## 4. 可看到脚本提示用户输入搜索词汇作为参数。

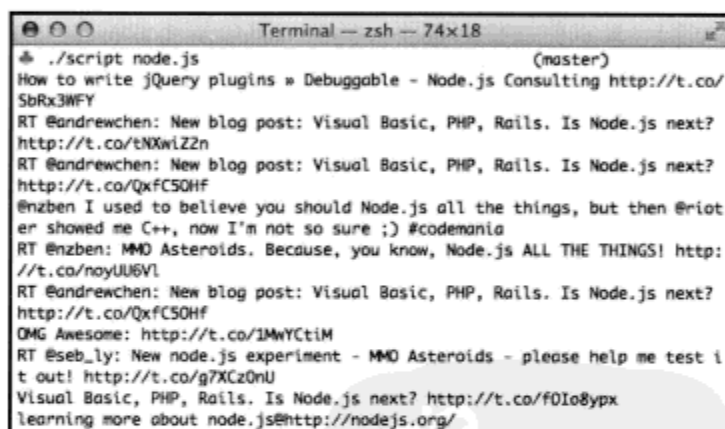
## 5. 带上参数再次运行脚本:

```
./script node.js
```

## 6. 可看到从 Twitter 搜索 API 返回的关于 node.js 的推文 (见图 16.3)。

图 16.3

给 Node.js 脚本传递参数



```

Terminal — zsh — 74x18
❖ ./script node.js (master)
How to write jQuery plugins » Debuggable - Node.js Consulting http://t.co/SbRx3WfY
RT @andrewchen: New blog post: Visual Basic, PHP, Rails. Is Node.js next? http://t.co/tNXwiZZn
RT @andrewchen: New blog post: Visual Basic, PHP, Rails. Is Node.js next? http://t.co/QxfC5OHf
@enzben I used to believe you should Node.js all the things, but then @riot er showed me C++, now I'm not so sure ;) #codemania
RT @enzben: MMO Asteroids. Because, you know, Node.js ALL THE THINGS! http://t.co/nayUU6VL
RT @andrewchen: New blog post: Visual Basic, PHP, Rails. Is Node.js next? http://t.co/QxfC5OHf
OMG Awesome: http://t.co/1MwYctiM
RT @seb_ly: New node.js experiment - MMO Asteroids - please help me test it out! http://t.co/g7XCz0nU
Visual Basic, PHP, Rails. Is Node.js next? http://t.co/F0Io8ypx
learning more about node.js http://nodejs.org/

```

## 16.7 小结

在本章,我们介绍了 Process 模块。我们看到了这个模块给开发人员提供许多用来与 Node.js 进程交互的工具。我们了解了如何发送 UNIX 信号给 Node.js 脚本以及如何使用 Node.js 创建小的、可执行的脚本。我们可以在这些知识的基础之上使用 Node.js 创建出可使用的、自动的解决许多问题的网络脚本。

## 16.8 问与答

问: 我能用 Process 模块来做什么?

答: 在使用 Node.js 创建脚本时, Process 模块尤其有用。它可提供关于脚本运行环境的信息,并可让信号发送给进程。它也让脚本可以设置正确的退出状态,以便让其他脚本可以正确地与之交互。对于大多数使用 Node.js 来做的编程工作,它都有用;毕竟所有一切都是建立在进程的基础上的!

问：我是否应该使用 Node.js 来编写脚本，而不使用诸如 Ruby、Python 或 Bash 这样的？

答：Node.js 精于网络编程，所以如果在脚本中做涉及网络的事情，Node.js 是个极佳的选择。Node.js 并不是作为通用脚本语言来设立的，所以如果读者有其他编程语言的经验，会发现有些目标使用其他语言更合适。要选择最适合于工作的工具，永远都是这样。

问：可以在 Node.js 脚本中使用系统命令吗？

答：可以。我们将在第 17 章了解如何做到。

问：是否有帮助编写 Node.js 脚本的模块？

答：是的。Commander 就是一个旨在让开发命令行脚本变得容易的 Node.js 模块，地址为 <https://github.com/visionmedia/commander.js>。还有一个位于 <https://github.com/chriso/cli> 的 cli 模块。更多可用的模块可以在 <http://search.npmjs.org/> 找到。

## 16.9 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

### 16.9.1 问题

1. 要在脚本中使用 Process 模块需要做点事情吗？
2. 为什么需要给脚本发送信号？
3. 如何确认一个进程是运行着的？

### 16.9.2 答案

1. 不需要。Process 模块是 Node.js 中的一个全局对象，所以无需在脚本中添加任何东西就可使用它。

2. 信号在 UNIX 和 UNIX 类型的系统中用于给进程之间的相互通信提供一种方法。可以使用它们来让进程终止、挂起或者重启。信号是 POSIX 的一个标准，意味着许多其他程序也使用它。在 Node.js 中使用信号让我们可以在需要的时候与 UNIX 工具集成。

3. 读者可能已经意识到，如果要使用进程功能的话，可能要相当依赖于脚本的运行。在 UNIX 类型的系统上，可以在背景运行脚本，也就是说不需要让终端窗口一直开着。在 UNIX 中还有许多工具，比如 monit 和 upstart，可以帮助我们监视进程并在需要的时候重启它们。此外还有使用 Node.js 创建的工具可带来相同的帮助。最流行的一个是 forever (<https://github.com/nodejitsu/forever/>)，它可确认脚本持续运行。

## 16.10 练习

1. 打开一个终端，在 Mac OSX 或者 Linux 下运行 `ps aux` 或者在 Windows 下运行 `tasklist /v`。试着理解这些进程的关联所在。读者应该看到在机器上运行的程序名称。请理解在机器上有许多进程正在运行而 Node.js 只是其中之一。

2. 编写一个接收两个数字作为参数的小命令行脚本。该脚本对两个数做乘法然后把结果输出到终端上。考虑需要对所输入数据进行验证的类型。



## 第 17 章

# 子进程模块

在本章中你将学到：

- 子进程是什么；
- 使用 Node.js 创建子进程；
- 使用子进程运行系统命令；
- 管理子进程。

### 17.1 什么是子进程

在第 16 章中，我们看到了 Node.js 提供了一个能让我们运行并管理 Node.js 进程的模块。我们也可能需要从 Node.js 进程创建子进程，而 Child Process（子进程）模块就是实现这一目标所需的工具。子进程就是由另一个进程创建的进程。创建了子进程的进程称为父进程。一个父进程可以有許多子进程，但一个子进程只能有一个父进程。以下是一些需要考虑使用子进程的场景。

- 需要计算一个复杂的等式。
- 需要使用位于 Node.js 外部的基于系统的工具来操纵一些数据。
- 正在执行任何资源密集型的或者需要花费大量时间来完成的操作。
- 想执行一些清理操作。

要说父进程与子进程之间的接口的话，数据流可以通过 `stdin`（标准输入）、`stdout`（标准输出）或者 `stderr`（标准错误）来交换。一开始的时候，这可能会有些令人困惑，但以后会变得清晰。为了演示 Node.js 如何与子进程通信，请打开终端并输入：

```
ping bbc.co.uk
```

我们会看到一系列的响应，表示服务器响应所花的时间。`ping` 命令与 Node.js 没有任何

关系。它是 Windows 和 UNIX 类型的系统上都有的一个工具，用于通过请求回声响应来查看主机或网关是否工作。ping 命令让我们可以说：“嘿，服务器，你还在吗？”如果正在工作，它就响应，ping 命令就向我们显示一些与这一来回旅程有关的统计信息。就如大多数基于终端的程序一样，ping 期待一些输入并发送一些输出。如果有兴趣在 Node.js 中使用 ping 程序，可以通过生成（spawn）一个子进程并通过侦听来自子进程的标准输出来实现。

在 Node.js 中创建子进程的第一种方法是使用 spawn() 方法。它需要以下几个参数。

- Command——想要运行的命令。
- Arguments——传递给命令的任何参数。
- Options——诸如工作目录和环境变量这样的东西。

在使用 ping 工具的例子中，ping 是命令而“bbc.co.uk”是参数，于是生成使用 ping 工具的子过程的方法如下：

```
var spawn = require('child_process').spawn;
var ping = spawn('ping', ['bbc.co.uk']);
```

这会启动一个子进程，就如我们在终端中运行 ping 命令所看到的那样，它向终端或者 stdout（标准输出）返回一些输出。为了能在 Node.js 脚本中使用这些数据，需要添加一个针对子进程的侦听器以便处理从标准输出接收到的数据。这里要注意的是必须指定数据的编码，否则显示的是原始流：

```
ping.stdout.setEncoding('utf8');
ping.stdout.on('data', function (data) {
 console.log(data);
});
```

现在如果运行脚本，就会生成子进程，父进程可以使用标准输出并将内容显示在终端上：

```
PING bbc.co.uk (212.58.241.131): 56 data bytes
64 bytes from 212.58.241.131: icmp_seq=0 ttl=245 time=16.115 ms

64 bytes from 212.58.241.131: icmp_seq=1 ttl=245 time=396.518 ms
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour17/example01 找到。按照下列步骤在 Node.js 中创建一个子进程。

1. 创建一个名为 app.js 的文件并将下列代码复制到其中：

```
var spawn = require('child_process').spawn;
var ping = spawn('ping', ['bbc.co.uk']);
ping.stdout.setEncoding('utf8');
ping.stdout.on('data', function (data) {
 console.log(data);
});
```

2. 运行脚本：

```
node app.js
```

3. 可在终端中看到来自子进程的输出（见图 17.1）。



图 17.1

生成子进程并输出  
子进程的结果

这个简单但强大的技巧可让我们访问操作系统中的任何软件。比如对上传的视频文件进行编码就是其有用的示例之一。对视频编码的业内标准工具是 **ffmpeg**。这是一个开源软件，可以重新编译并从命令行运行。它不是 Node.js 的一部分，但通过使用 **Child Process** 模块，Node.js 可访问它并使用它来创建一个能接收文件并使用 **ffmpeg** 对其编码的服务器。注意如果生成多个 CPU 密集型进程的话，有可能必须使用作业队列或者在多个计算机上分配各种处理以防止机器资源耗尽。

## 17.2 杀死子进程

父进程可以对子进程发送 **kill** 信号。这与我们在第 16 章中看到的 **Process** 模块的 **kill** 消息相似。为了杀死子进程，**Child Process** 模块提供了 **kill()** 方法。我们可以传进想让 **kill** 消息发送的消息类型。如果不传递任何东西，那么发送的就是 **SIGTERM** 信号。我们刚刚创建的示例现在可以修改成从父进程发送 **kill** 信号给子进程。父进程也可侦听子进程的退出事件并对此做一些处理。在本示例中，由于子进程被杀死，所以就不会显示 **ping** 命令的输出：

```
var spawn = require('child_process').spawn;
var ping = spawn('ping', ['bbc.co.uk']);

ping.stdout.setEncoding('utf8');
ping.stdout.on('data', function (data) {
 console.log(data);
});

ping.on('exit', function (code, signal) {
 console.log('child process was killed with a ' + signal + ' signal');
});

ping.kill('SIGINT');
```

### TRY IT YOURSELF

按如下步骤杀死子进程。

1. 创建一个名为 **app.js** 的文件并将下列代码复制到其中：

```
var spawn = require('child_process').spawn;
var ping = spawn('ping', ['bbc.co.uk']);

ping.stdout.setEncoding('utf8');
```

```
ping.stdout.on('data', function (data) {
 console.log(data);
});

ping.on('exit', function (code, signal) {
 console.log('child process was killed with a ' + signal + ' signal');
});

ping.kill('SIGINT');
```

## 2. 运行脚本:

```
node app.js
```

## 3. 可看到子进程被父进程杀死。

## 17.3 与子进程通信

Child Process 模块还提供一个用于创建一个也是 Node.js 进程的子进程的方法, 并提供让父进程具备通信通道的能力。这个方法称为 `fork()`, 用于创建子 Node.js 进程。如果正在与系统命令交互, 那么就使用 `spawn()`。使用 `fork()` 的开销在于每个子进程是个全新的 V8 实例。Node.js 文档提到, 每一个进程需要花费 30 毫秒来启动并占用 10MB 内存。所以能启动多少个子进程取决于计算机有多少内存!

对于本示例而言, 假设我们有一个父进程和一个需要与之通信的子进程。使用 `fork()` 方法与 `spawn()` 稍微有些不同, 创建子进程的方法现在如下:

```
var fork = require('child_process').fork;
var child = fork(__dirname + '/child.js');
```

现在的子进程应当是一个 Node.js 模块或程序。通过使用 `fork()` 方法, 父进程可以与子进程通信。这对于数据的传递或者让两个进程知道彼此状态的改变都有用处。要从父进程发送消息给子进程, 可以这样:

```
child.send({ message: 'Hello child!' });
```

在子进程中, 可以侦听消息并处理:

```
process.on('message', function(m) {
 console.log('child process received message:', m);
});
```

子进程也可将消息发送回父进程:

```
process.send({ message: 'Hello parent!' });
```

所有这一切加在一起, 父 Node.js 程序看起来就是这样的:

```
var fork = require('child_process').fork;
var child = fork(__dirname + '/child.js');
child.on('message', function(m) {
 console.log('parent process received message:', m);
});

child.send({ message: 'Hello child!' });
```



相关的 `child.js` 程序如下：

```
process.on('message', function(m) {
 console.log('child process received message:', m);
});
```

```
process.send({ message: 'Hello parent!' });
```

现在，当父进程运行的时候，它创建子进程，消息会在它们之间来回传送：

```
child process received message: { message: 'Hello child!' }
parent process received message: { message: 'Hello parent!' }
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour17/example03` 找到。按照下列步骤使用 `fork()` 并在进程间收发消息。

1. 创建一个名为 `parent.js` 的新文件并将如下内容复制到其中：

```
var fork = require('child_process').fork;
var child = fork(__dirname + '/child.js');

child.on('message', function(m) {
 console.log('parent process received message:', m);
});

child.send({ message: 'Hello child!' });
```

2. 创建另外一个名为 `child.js` 的新文件并粘贴如下内容：

```
process.on('message', function(m) {
 console.log('child process received message:', m);
});
```

3. 运行父进程：

```
node parent.js
```

4. 可在终端上看到父子进程之间的通信（见图 17.2）。



图 17.2

父进程和子进程之间的通信

## 17.4 集群 (Cluster) 模块

按计算机上可用的处理器数量扩展一个应用程序，是与子过程相关的一个常见需求。这意味着机器上的每个处理器都会有一个子进程，还会有一个父（或者主）进程来管理它们。在这样的上下文中，子进程经常被称为工作者进程（worker process）。Node.js 专门有一个模块用于让程序以集群创建子进程，并且使用单个父进程先处理到来的请求而后将请求交给一个子进程去处理。Hello World Web 服务器就是一个实际的示例。Cluster 模块提供一个简单的方法实现按计算机上可用的处理器数量扩展一个 Node.js 程序：

```
var cluster = require('cluster');
var http = require('http');
var cpus = 2;

if (cluster.isMaster) {
 console.log('Master process started with PID:', process.pid);

 for (var i = 0; i < cpus; i++) {
 cluster.fork();
 }

 cluster.on('death', function(worker) {
 console.log('worker ' + worker.pid + ' died');
 cluster.fork();
 });
} else {
 console.log('Worker process started with PID:', process.pid);
 http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('Hello World\n');
 }).listen(3000);
}
```

Cluster 模块启动一个主进程（有时候也称为父进程）并可按用户的意愿启动任意多的子进程。在 Cluster 模块中也可使用 fork() 方法，这实际上是建立在 Child Process 模块的 fork() 方法之上的，所以有许多相同的功能。当这个示例运行时，它会根据计算机上可用的 CPU 数量创建子进程——在本例中是 2。这些进程会以子（或者工作者）进程来创建，并且服务于 Hello World HTTP 服务器。主进程侦听工作者进程的死亡并对此事件作出响应（在本例中，是重启一个新的工作者）。

### Watch Out!

#### 警告：要小心会话

如果应用程序使用会话的话，在使用 Cluster 模块时需要小心。如果将会话储存在内存中，是不会在工作者之间共享的。为了解决这个问题，可以将会话储存在诸如 Redis 这样的东西中，从而确保工作者进程可共享会话。

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour17/example04` 找到。按照下列步骤使用 Cluster 模块创建子进程。

1. 创建一个名为 `app.js` 的新文件并粘贴如下内容:

```
var cluster = require('cluster');
var http = require('http');
var cpus = 2;

if (cluster.isMaster) {
 console.log('Master process started with PID:', process.pid);

 for (var i = 0; i < cpus; i++) {
 cluster.fork();
 }

 cluster.on('death', function(worker) {
 console.log('worker ' + worker.pid + ' died');
 cluster.fork();
 });
} else {
 console.log('Worker process started with PID:', process.pid);
 http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('Hello World\n');
 }).listen(3000);
}
```

2. 启动服务器:

```
node app.js
```

3. 可看到一个主进程和两个子进程启动。

4. 注意其中一个子进程的 `pid`, 并在另外一个终端窗口或选项卡中对该子进程发出一个 `kill` 信号:

```
kill [pid number of child process]
```

5. 回到运行服务器的终端窗口, 可看到该进程死亡。主进程启动另外一个子进程 (见图 17.3)。

```
Terminal — node — 80x24
$ node app.js
Master process started with PID: 77378
Worker process started with PID: 77379
Worker process started with PID: 77380
worker 77379 died
Worker process started with PID: 77388
```

图 17.3  
Cluster 模块启动  
一池子的工作者并  
重启死亡的工作者

## 17.5 小结

在本章，我们学习了 `Child Process` 模块。我们了解了创建子进程的方法以及使用子进程在 `Node.js` 中运行系统命令的方法。我们了解了父进程如何杀死子进程以及如何向子进程发送信号。我们学习了 `fork()` 这个可以创建子 `Node.js` 进程的方法以及如何在 `Node.js` 进程之间通信。最后，我们看了给予进程使用 `Cluster` 模块的方法以及按照服务器上可用的处理器内核数量来扩展应用程序的方法。

## 17.6 问与答

问：我应当使用系统命令，而不用尝试使用 `Node.js` 编写 `JavaScript` 吗？

答：是的。有时候，我们遇到的问题已经有成熟、稳定的使用其他编程语言或者是系统工具的解决方案。使用 `ffmpeg` 来处理视频就是个好示例：用其他工具来解决问题会比自己去写代码更稳定并快得多。

问：使用子进程和系统工具有什么问题吗？

答：可移植性是个主要问题。如果为一种操作系统编写代码，就不能保证当代码运行在另一种操作系统上的时候某个系统工具或者软件可用。如果可移植性会成为所写的软件的问题，那么就要仔细考虑使用系统级别的工具。

问：我需要使用子进程吗？

答：子进程应当用于特殊目的，比如可使用系统工具、完成长时间运行的进程或者按计算机上处理器的数量扩展应用程序。对于大多数编程需求而言，使用单个进程写程序足矣。

## 17.7 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

### 17.7.1 问题

1. 什么时候应当使用 `spawn()`，什么时候应当使用 `fork()`？
2. 当使用 `spawn()` 的时候需要注意什么？
3. 什么时候应当使用 `Cluster` 模块而不使用 `Child Process` 模块？

### 17.7.2 答案

1. 如果想使用系统命令的话应当使用 `spawn()`。使用 `fork()` 来创建另外一个 `Node.js` 进程。
2. 如果开发过程中使用系统命令的话，在部署应用程序的时候可能会发现系统命令要

么不可用要么有不同的名称。如果使用 `spawn()` 的话，要确认在部署应用程序时 Node.js 能够访问这一命令。如果使用平台即服务提供商，那么很有可能无法使用系统命令。

3. 如果希望使用主进程来管理一池子的子进程或者工作者进程的话，应当使用 `Cluster` 模块。如果需要利用计算机上可用的处理器核心来扩展应用程序，那么 `Cluster` 模块提供了一个简单的方法来实现。

## 17.8 练习

1. 找另外一个在命令行上使用的系统程序，修改 `hour17/example01` 示例的代码将其结果输出到控制台。尝试理解传递给子进程的参数如何影响输出。

2. 扩展 `hour17/example03` 中的示例，通过让子脚本读取一个文件的内容并将文件内容以消息发送给父脚本的方式，进一步添加 `hour17/example01` 中的父脚本和子脚本之间的消息收发。

3. 体验 `hour17/example04` 的示例，添加更多工作者进程。尝试发送 `kill` 信号给工作者进程。如果给主进程发送 `kill` 信号会发生什么？



## 第 18 章

# 事件模块

---

在本章中你将学到：

- 使用事件模块；
- 发送并侦听事件；
- 事件模块在网络编程中的作用；
- 动态添加与移除事件。

### 18.1 理解事件

我们已经了解，Node.js 被认为是实现并发的最佳方法。它认为使用事件循环（有时候也称为事件队列）是支持异步代码并且解决并发问题的高效方式。对于来自过程化的、命令式的语言的开发人员而言，这会是在范式上的改变；许多开发人员将围绕着事件的编程称为“将里面翻到外面来编程”。

Events（事件）模块是 Node.js 的核心，许多其他模块用它来围绕着事件构架功能。由于 Node.js 运行于单一的线程中，任何同步代码都是阻塞的，所以如果有长时间运行的代码的话事件循环会被阻塞。为了有效地使用 Node.js 编写代码，必须仔细思考自己的编程风格并且遵循一些简单的规则。

- **别阻塞**——Node.js 是单线程的，如果代码阻塞的话所有其他一切都停止了。
- **快速返回**——操作应当快速返回。如果不能快速返回，就应当将其移到另一个进程中。

Events 模块提供了围绕事件构架代码的一个简单方法，无论事件是由你来控制的，或者来自网络的某个地方。Events 模块让开发人员可以为事件设置侦听器和处理器。如果读者已经在使用客户端的 JavaScript，那么应该熟悉侦听器和处理器的创建。在客户端 JavaScript 中，

可以对单击事件设置一个侦听器，然后在事件发生时执行一些事情：

```
var target = document.getElementById("target");
target.addEventListener('click', function () {
 alert('Click event fired. Target was clicked!');
}, false);
```

如果熟悉 jQuery，那么在 jQuery 中的代码就会是这样的：

```
$("#target").click(function() {
 alert('Click event fired. Target was clicked!');
});
```

这两段代码做的是相同的事情。它们对 id 为“target”的 HTML 元素设置单击侦听器。当单击事件发生时，就说事件被触发，事件侦听器处理事件，显示一个警告。事件的处理方式在侦听器内的函数中定义。在这些示例中，事件由人控制鼠标发送。当单击鼠标时，事件触发。

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour18/example01` 找到。按下列步骤在 JavaScript 中触发事件。

1. 创建一个名为 `index.html` 的文件并将下列代码复制到其中：

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <meta charset="utf-8" />
 <title>JavaScript Events</title>
 </head>
 <body>
 <h1>JavaScript Events</h1>
 <button id="target">Click me</button>
 <script type="text/javascript">
 var target = document.getElementById('target');
 target.addEventListener('click', function () {
 alert('Click event fired. Target was clicked!');
 }, false);
 </script>
 </body>
</html>
```

2. 在 Web 浏览器中打开 `index.html`。
  3. 单击按钮触发单击事件。
  4. 注意，随着事件的触发，会有警告框显示出来（见图 18.1）。
- 

Node.js 中的事件更常见的是网络事件，包括如下的事情。

- 来自 Web 服务器的响应。
- 从文件读取数据。
- 从数据库返回数据。

图 18.1

单击事件触发并由  
事件侦听器处理



在代码中包含 `Events` 模块与其他模块类似，都要用到大家熟悉的对模块的请求。不过，为了使用这一模块，必须创建一个新的 `EventEmitter` 实例：

```
var EventEmitter = require('events').EventEmitter;
var ee = new EventEmitter();
```

一旦在代码中加入上述内容，就可以开始添加事件和侦听器了。`Events` 模块实际上很简单，如果读者理解事件的发送和侦听，那么就不难理解其工作方式了。如果请求了 `Events` 模块并且创建了一个新的 `EventEmitter()` 实例，就可按如下方法发送事件：

```
ee.emit('message', 'This emits a message');
```

第一个参数是一个对事件进行描述的字符串，以便用于与侦听器匹配。读者可使用任何其他字符串。一旦给事件标好了标签，就可以添加更多的参数。如果需要，可以有超过一个参数，在收到事件时这些参数将会被传递到侦听器中。在本例中，第二个参数是一个字符串。

为了接收消息，必须添加侦听器。侦听器侦听事件并在事件触发时处理它。在本例中，`emit()` 的第二个参数以 `data` 传递给了匿名函数以便使用：

```
ee.on('message', function(data) {
 console.log(data);
});
```

为了进一步探究这个过程，想象一下自己是詹姆斯邦德，要显示一条能在 5 秒钟内自我毁灭的机密消息。通过发送和侦听事件，用一小段 `Node.js` 脚本很容易就能实现：

```
var EventEmitter = require('events').EventEmitter;
var secretMessage = new EventEmitter();

secretMessage.on('message', function(data) {
 console.log(data);
});

secretMessage.on('self destruct', function() {
 console.log('BANG!! The message is destroyed!');
});

secretMessage.emit('message', 'This is a secret message. It will self destruct in 5
seconds..');

setTimeout(function(){
 secretMessage.emit('self destruct');
}, 5000);
```



在这段脚本中，发送了两个事件，有两个侦听器。当脚本运行的时候消息事件就发生并由“message”处理器处理。setTimeout 在 5 秒之后发送另一个“self destruct”事件。注意这里没有额外的参数——因为额外参数是可选的。如果运行这段脚本，可看到如下结果：

```
This is a secret message. It will self destruct in 5 seconds..
[5 seconds later ...]
BANG!! The message is destroyed!
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour18/example02` 找到。按下列步骤发送和侦听事件。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var EventEmitter = require('events').EventEmitter;
var secretMessage = new EventEmitter();
secretMessage.on('message', function(data) {
 console.log(data);
});

secretMessage.on('self destruct', function() {
 console.log('BANG!! The message is destroyed!');
});

secretMessage.emit('message', 'This is a secret message. It will self destruct
 in 5 seconds..');

setTimeout(function(){
 secretMessage.emit('self destruct');
}, 5000);
```

2. 运行脚本：

```
node app.js
```

3. 可看到如下输出，其中第二行在 5 秒暂停后出现。

```
This is a secret message. It will self destruct in 5 seconds..
BANG!! The message is destroyed!
```

### 提示：在 Node.js 中到处都用 EventEmitter

EventEmitter 也在许多其他模块中用来处理 Node.js 中的事件。读者可能还记得在读取文件时、创建 HTTP 服务器时或者使用 Stream（流）时使用了 EventEmitter。代码相对较小，如果读者知道如何阅读 JavaScript，可以在这里找到：<https://github.com/joyent/node/blob/master/lib/events.js>。代码虽小，却是 Node.js 中的大拿！

**Did you  
know?**

## 18.2 通过 HTTP 演示事件

Node.js 以事件的方式来接近在网络中传输或者通过 I/O 操作获得的数据并广泛使用

Events 模块来支持异步编程。从联网事件上说, Twitter 的流 API 就是一个很好的示例。它让开发人员得以保持与 Twitter API 连接的打开状态并且当数据产生时接收数据。

在 Node.js 中使用 HTTP 或者 HTTPS 模块时, 我们实际上在使用 EventEmitter 的实例, 而且为了让我们能够与 HTTP 服务器交互, 系统事先定义了许多事件。

思考一下从 Twitter 的流 API 获取数据的示例, 就会了解围绕事件来构架是合理的。假设你馋巧克力, 想实时侦听有关巧克力的推文。由于流 API 允许我们保持连接打开, 所以我们可以实现这个目标, API 会在新数据可用时将新数据推送给我们。由于 Node.js 中的 HTTP 和 HTTPS 模块是 EventEmitter 的实例, 所以可以侦听数据事件, 当接收到数据事件时立刻做一些处理:

```
var https = require('https');
var username = 'YOUR_TWITTER_USERNAME';
var password = 'YOUR_TWITTER_PASSWORD';
var json;

var options = {
 host: 'stream.twitter.com',
 auth: username + ':' + password,
 path: '/1/statuses/filter.json?track=chocolate',
 method: 'POST'
};

var req = https.request(options, function(res) {
 res.setEncoding('utf8');
 res.on('data', function(data) {
 json = JSON.parse(data);
 console.log('New data event!');
 console.log(json.text);
 });
});

req.end();
```

这是个简单的 HTTPS 客户端, 与在第 5 章的 HTTP 模块中的相似。这里需要关注的是客户端中的如下代码行:

```
res.on('data', function (data) {
```

这是一个直接来自 Events 模块的事件侦听器, 它让客户端可以在事件发生的时候接收数据并对其做一些处理。在本例中, 触发侦听器的事件是个网络事件。当收到新数据时, 触发这一事件的是 Twitter API。在这样的上下文中, 我们可以看到, Events 模块可用于处理可能在任何时间发生的网络事件并且在事件抵达的时候立即响应。处理经常是脚本控制范围之外的网络事件和 I/O 操作并使用事件作为管理这些事情的一种方法, 这是 Node.js 的核心原理之一。

---

## TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour18/example03` 找到。按下列步骤演示流数据的事件。

## 1. 创建一个名为 app.js 的文件并将下列代码复制到其中:

```

var https = require('https');
var username = 'YOUR_TWITTER_USERNAME';
var password = 'YOUR_TWITTER_PASSWORD';
var json;

var options = {
 host: 'stream.twitter.com',
 auth: username + ":" + password,
 path: '/1/statuses/filter.json?track=chocolate',
 method: 'POST'
};

var req = https.request(options, function(res) {
 res.setEncoding('utf8');
 res.on('data', function (data) {
 json = JSON.parse(data);
 console.log('New data event!');
 console.log(json.text);
 });
});

req.end();

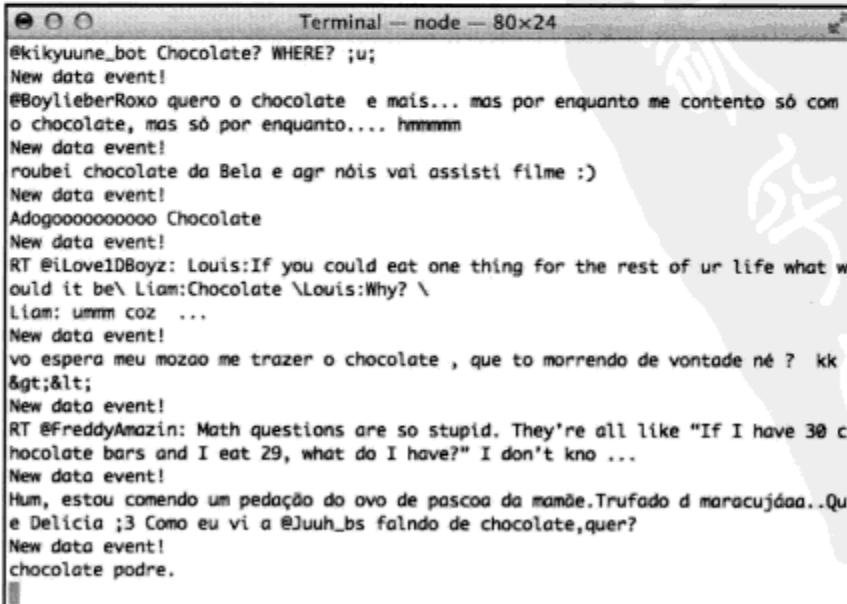
```

## 2. 将 username 和 password 变量更改为读者自己的 Twitter 用户名和密码。

## 3. 运行脚本:

```
node app.js
```

## 4. 可看到新的数据事件到达服务器并由事件侦听器处理 (见图 18.2)。



```

Terminal — node — 80x24
@kikyune_bot Chocolate? WHERE? ;u;
New data event!
@BoyllieberRoxo quero o chocolate e mais... mas por enquanto me contento só com
o chocolate, mas só por enquanto.... hmmm
New data event!
roubei chocolate da Bela e agr nós vai assisti filme :)
New data event!
Adogoooooooooooo Chocolate
New data event!
RT @iLove1DBoyz: Louis:If you could eat one thing for the rest of ur life what w
ould it be\ Liam:Chocolate \Louis:Why? \
Liam: umm coz ...
New data event!
vo espera meu mozao me trazer o chocolate , que to morrendo de vontade né ? kk
>>
New data event!
RT @FreddyAmazin: Math questions are so stupid. They're all like "If I have 30 c
hocolate bars and I eat 29, what do I have?" I don't kno ...
New data event!
Hum, estou comendo um pedaço do ovo de pascoa da mamãe.Trufado d maracujáaa..Qu
e Delícia ;3 Como eu vi a @Juuh_bs falndo de chocolate,quer?
New data event!
chocolate podre.

```

图 18.2

侦听来自网络事件  
的事件

## 18.3 用事件玩乒乓

为了对用事件来构建程序的方式有更全面的理解，我们创建一个 ping-pong（乒乓）示例。在计算术语中，ping 指的是对其他一些事务的请求（比如对服务器），向其询问是否活着。通过使用 Events 模块，我们可以创建一个自我引用的脚本用来向自己来回发送 ping 和 pong 消息。我们的目的是演示事件侦听器也可以是事件发送器，而且这样的技巧可用于让事件侦听器触发其他事件：

```
var EventEmitter = require('events').EventEmitter;
var pingPong = new EventEmitter();

setTimeout(function() {
 console.log('Sending first ping');
 pingPong.emit('ping');
}, 2000);

pingPong.on('ping', function() {
 console.log('Got ping');
 setTimeout(function() {
 pingPong.emit('pong');
 }, 2000);
});

pingPong.on('pong', function() {
 console.log('Got pong');
 setTimeout(function() {
 pingPong.emit('ping');
 }, 2000);
});
```

这段脚本大量使用了 setTimeout，它在一定时间之后执行其他一些事情。在现实中，这可以是一个网络操作或者不能确切知道会返回什么的事情。在这段脚本中，第一个 setTimeout 启动游戏并发送 ping 消息。对于 ping 和 pong 有两个侦听器，它们的响应都是在 2 秒之后发送一条消息。当脚本运行时，事件创建一个无限循环并来回发送消息直到用户停止脚本：

```
Sending first ping
Got ping
Got pong
Got ping
Got pong
```

使用一个事件来触发另一个事件的简单思想使复杂的功能得以创建，尤其在事件层叠发生的地方。在这样的上下文中，我们可以看到，Events 模块可以用来构建复杂的网络程序，对网络或者 I/O 事件作出响应。

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour18/example04` 找到。按照下列步骤用事件侦听器发送事件。

1. 创建一个名为 `parent.js` 的文件并将如下内容粘贴到其中：

```
var EventEmitter = require('events').EventEmitter;
var pingPong = new EventEmitter();

setTimeout(function(){
 console.log('Sending first ping');
 pingPong.emit('ping');
}, 2000);

pingPong.on('ping', function() {
 console.log('Got ping');
 setTimeout(function(){
 pingPong.emit('pong');
 }, 2000);
});

pingPong.on('pong', function() {
 console.log('Got pong');
 setTimeout(function(){
 pingPong.emit('ping');
 }, 2000);
});
```

2. 运行脚本：

```
node app.js
```

3. 查看消息在事件处理器之间如何来回发送并理解事件处理器也是发送器。这让 ping-pong 持续不断进行下去（见图 18.3）。

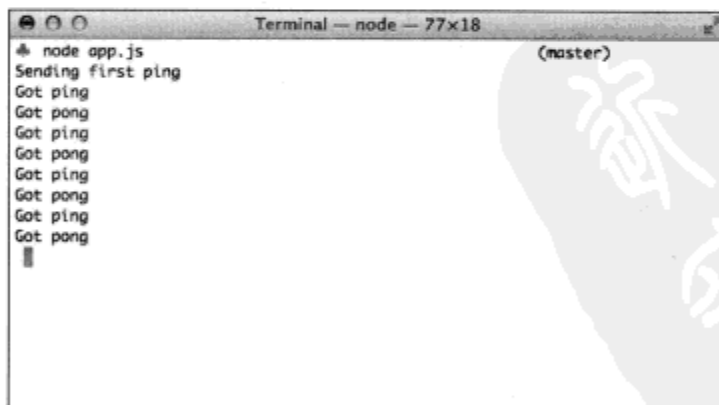


图 18.3

使用 Events 模块  
玩乒乓

## 18.4 动态编写事件侦听器程序

ping-pong 示例演示了事件侦听器也可成为事件发送器，但所有这些逻辑都是在脚本首次运行时创建的。Events 模块也允许我们在脚本运行当中依据某个事件或者脚本中的其他一些逻辑的发生动态地添加或者移除侦听器。

通过使用 Events 模块，我们可以在脚本的任何点上添加并移除侦听器，于是我们可以选择让脚本在特定事件发生时响应，或者分析在某个事件中的数据，从而依此添加或删除其他

侦听器。

在 ping-pong 示例中，我们在 4 秒之后添加第二个侦听器，其中带有另外一个 `setTimeout` 函数。这会运行回调函数并记录：每当接收到一个 ping 事件时，第二个侦听器已经接受到了一个事件：

```
var logPing = function() {
 console.log("Second ping listener got ping");
}
setTimeout(function(){
 console.log('Added a second ping listener');
 pingPong.on('ping', logPing);
}, 4000);
```

现在当脚本运行 4 秒之后，第二个侦听器将会被动态添加并且会将它已经接收到的一个 ping 消息记录到终端。在现实中，我们会想使用这样的机制来编写一些逻辑在另一个事件的基础上对特定的场景响应。事件侦听器也可通过提供对消息的引用和回调函数来移除：

```
pingPong.removeListener('ping', logPing);
```

于是如果我们想移除动态添加的事件侦听器的话，可添加另一个 `setTimeout` 在一段时间，比如 12 秒之后移除它：

```
setTimeout(function(){
 console.log('Removed second ping listener');
 pingPong.removeListener('ping', logPing);
}, 12000);
```

于是，现在当脚本运行时，会有是发送器的事件，也会有在进程的生命周期中动态添加和移除的侦听器：

```
Sending first ping
Got ping
Added a second ping listener
Got pong
Got ping
Second ping listener got ping
Got pong
Got ping
Second ping listener got ping
Removed second ping listener
Got pong
Got ping
Got pong
```

以演示为目的，这个示例使用 `setTimeout` 来触发事件，但在现实中我们会使用 `Events` 模块对 I/O 和网络操作做出响应。`Events` 模块是个小而强大的瑞士军刀，它让我们以事件为切入点解决问题。

以下是发送事件的侦听器以及动态添加删除侦听器的完整示例：

```
var EventEmitter = require('events').EventEmitter;
var pingPong = new EventEmitter();

setTimeout(function(){
```

```

 console.log('Sending first ping');
 pingPong.emit('ping');
 }, 2000);

pingPong.on('ping', function() {
 console.log('Got ping');
 setTimeout(function(){
 pingPong.emit('pong');
 }, 2000);
});

pingPong.on('pong', function() {
 console.log('Got pong');
 setTimeout(function(){
 pingPong.emit('ping');
 }, 2000);
});

var logPing = function() {
 console.log("Second ping listener got ping");
}

setTimeout(function(){
 console.log('Added a second ping listener');
 pingPong.on('ping', logPing);
}, 4000);

setTimeout(function(){
 console.log('Removed second ping listener');
 pingPong.removeListener('ping', logPing);
}, 12000);

```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 [hour18/example05](#) 找到。按照下列步骤来动态管理事件。

1. 创建一个名为 `app.js` 的新文件并粘贴如下内容：

```

var EventEmitter = require('events').EventEmitter;
var pingPong = new EventEmitter();

setTimeout(function(){
 console.log('Sending first ping');
 pingPong.emit('ping');
}, 2000);

pingPong.on('ping', function() {
 console.log('Got ping');
 setTimeout(function(){
 pingPong.emit('pong');
 }, 2000);
});

pingPong.on('pong', function() {
 console.log('Got pong');
 setTimeout(function(){
 pingPong.emit('ping');

```

```

 }, 2000);
 });

 var logPing = function() {
 console.log("Second ping listener got ping");
 }

 setTimeout(function(){
 console.log('Added a second ping listener');
 pingPong.on('ping', logPing);
 }, 4000);

 setTimeout(function(){
 console.log('Removed second ping listener');
 pingPong.removeListener('ping', logPing);
 }, 12000);

```

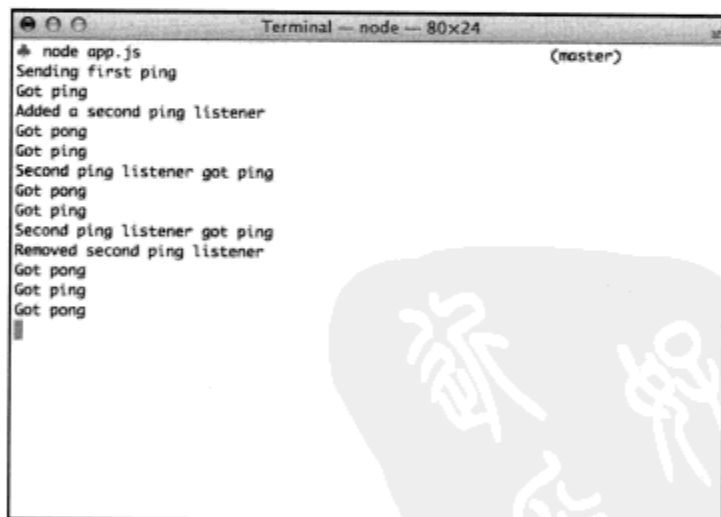
## 2. 启动服务器:

```
node app.js
```

## 3. 查看消息如何来回发送以及侦听器如何被动态添加而后移除（见图 18.4）:

图 18.4

动态添加与移除事件侦听器



## 18.5 小结

在本章，我们介绍了 Events 模块以及如何在 Node.js 中使用事件。我们扮演了一下詹姆斯邦德创建一个自我销毁的消息，然后了解了 Node.js 的许多其他模块如何使用事件。我们看到 HTTP 和 HTTPS 模块如何使用事件来接收数据，并且看了一个使用 Twitter 流 API 的示例。我们使用 Node.js 脚本玩了一回乒乓，看到了事件侦听器也可发送事件并可让脚本更智能和复杂。最后，我们了解了事件侦听器可动态添加和移除。

## 18.6 问与答

问：对于一个事件是否有最大侦听器数量的限制？



答：默认情况下如果事件有超过 10 个侦听器，它会发出警告。不过，可以按每个事件通过使用 `emitter.setMaxListeners(n)` 来更改这个数量。

问：为什么事情不以代码中出现的顺序发生？

答：如果围绕着事件编程，就是在将脚本从内翻到外。我们不再在代码中围绕着事情出现的顺序编程，而是围绕着事件发生的顺序编程。这可是件大事！

问：是否有能够侦听所有发送出来的事件的方法？

答：没有。我们需要给每个想要响应的事件创建侦听器。

## 18.7 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

### 18.7.1 问题

1. 围绕事件的编程有何不同？
2. 为何事件是网络编程的好方法？
3. 为什么说 JavaScript 非常适合于 Node.js 要解决的问题？

### 18.7.2 答案

1. 使用事件的时候，脚本不再以编写的顺序执行，这与简单的顺序化的脚本不同。我们使用事件侦听器响应所发生的事件，而无论事件在何时发生。这是解放性的，不过一开始需要对编码方法的思维方式有所改变。

2. 当应用程序包括网络调用时，我们经常无法准确预测结果什么时候会返回。随着应用程序越来越复杂，围绕着事件来构架代码就变得越来越可管理和高效。当事件发生时，侦听器可对此做出响应！

3. 由于 JavaScript 是事件驱动的编程语言，所以它非常适合于 Node.js 所采用的解决网络编程的方法。JavaScript 让 Node.js 得以提供许多开发人员已经熟悉的事件驱动 API。

## 18.8 练习

1. 阅读位于 <http://nodejs.org/docs/latest/api/http.html> 上的 Node.js 的 HTTP 模块的文档和示例。读者是否能认出使用 Events 模块的地方以及事件的使用方法？

2. 创建一个短的 Node.js 脚本，使用 Events 模块来发送两个不同的事件。创建两个事件侦听器来接收事件。

3. 扩展 `hour18/example05` 示例，给 `pong` 事件动态添加然后移除第二个侦听器。

## 第 19 章

# 缓冲区模块

在本章中你将学到：

- 什么是二进制数据；
- 什么是字符编码；
- 如何使用二进制数据；
- 如何使用缓冲区。

### 19.1 二进制数据初步

为了理解 Buffer（缓冲区）模块，我们需要先理解二进制数据。如果读者已经很好掌握了这一主题，可跳过本节。如果没有，那么继续阅读吧！二进制数字可以是两个值之一：0 或者 1。它也称为比特（二进制数字的简写），它是计算机系统中数据的原子单位。0 和 1 这两个值是表示信息的最简单方式，因为它们只能是两种可能之一——0 或者 1。我们也可用真和假、开和关来解释它们。虽然可以认为这是一个简单而且微不足道的事情，但如果使用多个比特，就可以表达更为复杂的事务。

关于人类对数的处理方法，我们有一个用 10 个不同的值来表达数值的系统：

0 1 2 3 4 5 6 7 8 9

有了这些值，我们通过使用多个这样的值并将它们组合在一起就可以表达数值。由于它们有 10 个值，所以这个系统称为以 10 为基数。数值也可以二进制来表达，与使用 10 个值来表达一个数值不同，它们只使用两个——1 和 0。因为只有两个值，所以称其为以 2 为基数。

**By the  
Way**

**注意：**十进制系统是以 10 为基数的

我们用来表达数值的系统通常称为十进制，因为它们有 10 个数，但也可称为以 10 为基数。

就如以 10 为基数的系统那样，二进制（或者以 2 为基数）中数的值由从右边开始的位置来决定。随着一个比特从右到左转移，它的值按 2 的次方增长：

1	1	1	1	1	1	1	1
128	64	32	16	8	4	2	1

就如以 10 为基数的系统那样，以 2 为基数的二进制数由从右开始的位置来决定。它使用与十进制系统相同的方式对值进行组合来表示一个值。下表给出一个数的以 2 为基数的系统中的值及其以 10 为基数的值：

以 2 为基数	以 10 为基数
0	0
1	1
00	0
01	1
10	2
11	3

在以 2 为基数的系统中，8 个比特都是 1（或者 11111111）的值是 255。8 个比特组成一组称为字节，它可用来储存一个表示字母或数字的值。如你所见，8 个比特最大可以声明的数是 255。如果 8 个都是 0 值，可以声明一个 0，于是一个字节可以表示 256 个不同的值。字节有时候也称为八位组（octets），因为它有 8 个比特；它们构成保存文本字符的基础。

#### 注意：4 个比特（Nibble）

4 比特或者半个字节称为 Nibble。由于一个 Nibble 是 4 比特，所以它可以有 16 个可能的值。这也可以称为十六进制（hexadecimal 或简称 hex）或者以 16 为基数，就是因为它有 16 个可能的值！

By the  
Way

## 19.2 从二进制到文本

计算机是在一个由许多 0 和 1 构成的系统上操作，但人类不是这样。因此，有许多编码系统用于将二进制数据转换成文本。ASCII（美国标准信息交换码）就是其中一种。它是以英文字母为基础并且提供了用来对英语中常用的字母、数字和标点符号编码的一种方法。ASCII 中可能的不同字符数为 128（0~127）。读者学过以 2 为基数的系统，应该会想起 128 这个数所表示的意义是显然的：它是我们能 7 个比特能表示的最大数字个数。这就提供了一个声明英文字符的系统。比如，在 ASCII 中，“A” 这个数字是十进制的 65 或者二进制的 1000001。

ASCII 是为英语设计的，但显然这个世界不仅仅使用英语或者英文字符来交流。在最近几年，Web 上选择的编码方式是 UTF-8。它可表达 Unicode 这一包括了当今事件上大多数语言和字符集合的字符集中的每个字符。UTF-8 使用 1 到 4 个字节来表达字符并且完全后向兼容 ASCII。对于被认为是会经常使用的字符就使用少一些的字节数（通常是一个）来编码，而不常用的字符则使用 4 个字节来编码。

## 19.3 二进制和 Node.js

JavaScript 最初是为浏览器设计的，所以能很好处理 unicode 编码的字符串（人类可以阅读的文本），但不能很好处理二进制数据。这是 Node.js 的一个问题，因为 Node.js 旨在网络上发送和接收经常是以二进制格式传输的数据。需要使用二进制数据传输的示例有：

- 通过 TCP 连接发送和接收数据；
- 从图像或者压缩文件读取二进制数据；
- 从文件系统读写数据；
- 处理来自网络的二进制数据流。

Buffer 模块给 Node.js 带来一种存储原始数据的方法，于是可以在 JavaScript 的上下文中使用二进制数据。每当需要在 Node.js 中处理在 I/O 操作中移动的数据时，就有可能使用 Buffer 模块。比如，在读取一个文件时，Node.js 使用 Buffer 对象来保存来自文件的数据。我们可以通过创建一个名为 file.txt 的文本文件然后对其写入一些文本并创建一个 Node.js 脚本读取文件来看一看：

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

fs.readFile('file.txt', function(err,data){
 if (err) throw err;
 console.log(data);
});
```

如果运行这段脚本，可看到所记录的数据实际上是一个 Buffer 对象：

```
<Buffer 23 23 0a 23 20 55 73 65 72 20 44 61..
```

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour19/example01 找到。按照下列步骤来展示 Buffer 对象。

1. 创建一个名为 app.js 的文件并将下列内容复制到其中：

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

fs.readFile('file.txt', function(err,data){
 if (err) throw err;
 console.log(data);
});
```

2. 创建一个名为 file.txt 的文件并加入如下内容：

```
It is the future,
The distant future
It is the distant future,
The year 2000

We are robots
```

## 3. 运行脚本:

```
node app.js
```

## 4. 可看到原始 Buffer 对象打印在了控制台中 (见图 19.1)。

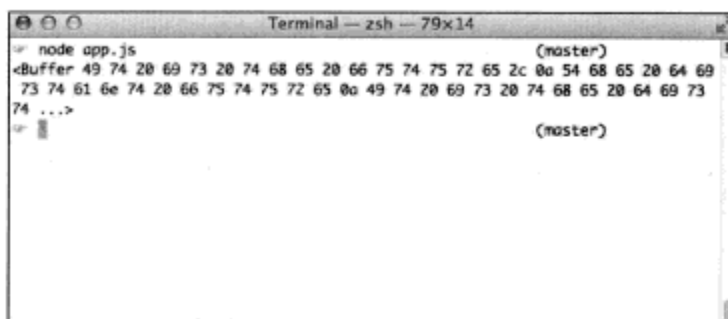


图 19.1

通过读取文件来展示原始缓冲区数据

在大多数模块中, 必须设置编码以便告诉 Node.js 将数据转换成人类可读的格式。为了展示数据, 必须将编码作为参数之一提供给 `fs.readFile()` 方法:

```
var fs = require('fs');
fs.readFile('file.txt', 'utf8', function(err,data){
 if (err) throw err;
 console.log(data);
});
```

现在, 文件的内容将会显示出来, 而不再是原始缓冲区了:

```
It is the future,
The distant future
It is the distant future,
The year 2000

We are robots
```

## TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour19/example02` 找到。按照下列步骤来给缓冲区设置编码。

1. 创建一个名为 `app.js` 的文件并将下列内容复制到其中:

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

fs.readFile('file.txt', 'utf8', function(err,data){
 if (err) throw err;
 console.log(data);
});
```

2. 创建一个名为 `file.txt` 的文件并加入如下内容:

```
It is the future,
The distant future
It is the distant future,
The year 2000

We are robots
```

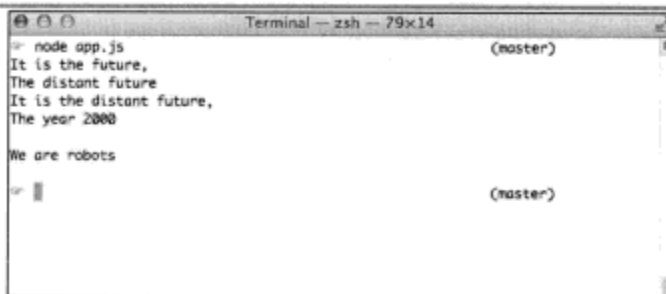
## 3. 运行脚本:

```
node app.js
```

## 4. 可看到文件的内容而不是原始 Buffer 对象打印在了控制台中 (见图 19.2)。

图 19.2

在声明编码之后  
展现出来的是文  
件内容



### By the Way

#### 注意: 字符有许多种编码方式

字符的编码与解码有许多方式。最常见的是 UTF-8, 它支持当今世界上几乎所有在用的字符。在 Node.js 中可使用的其他编码格式包括 `ascii`、`ucs3`、`base64` 和 `hex`。

## 19.4 Node.js 中的缓冲区是什么?

Node.js 中的缓冲区是处理二进制数据的一种方式。由于 JavaScript 语言不能很好处理二进制数据, 所以缓冲区实际上是对原始内存的分配, 以便 Node.js 对此读写数据。这就让 Node.js 可以以健全的方式来处理二进制数据, 而不必依赖在二进制数据的处理方面有所不足的 JavaScript 语言。

缓冲区是 Buffer 类的实例, 由于 Buffer 模块是全局的, 所以无需请求该模块即可使用。依据实例化的方式不同, 缓冲区有许多选项。它们可以用一个代表要分配给缓冲区的字符数 (或者八位位组数) 的整型值作为参数。也可指定要使用的编码。默认编码是 UTF-8, 所以如果不指定编码的话, 使用的将是 UTF-8。

创建一个带有 8 个字节的新缓冲区方法如下:

```
var buffer = new Buffer(8);
```

这会实例化一个带有 8 个字节内存的新缓冲区:

```
buffer
<Buffer cc cc cc cc cc cc ff 75>
```

如果将其作为字符串写到控制台, 那么它将展示的是一个 UTF-8 字符串。Node.js 提供 `toString()` 方法将缓冲区以字符串方式写出:

```
buffer.toString()
'???????u'
```

注意读者可能看到与本示例不同的字符, 因为缓冲区所代表的是在读者计算机上所分配出来的原始内存。

也可创建一个新 Buffer 对象然后将字节或者八位位组的数据传递进来。如前所述,

85 等同于字符“U”。于是，创建一个值为 85 的缓冲区，如果使用 UTF-8 编码的话其内容就是字符 U：

```
var buffer = new Buffer([85])
buffer.toString('utf-8')
'U'
```

## TRY IT YOURSELF

按下列步骤创建一个带有字节数组的缓冲区。

1. 在控制台上输入 node 启动 Node.js REPL。
2. 用一个八位位组初始化一个缓冲区：

```
var buffer = new Buffer([85])
```

3. 将缓冲区转换为字符串：

```
buffer.toString('utf-8')
```

4. 可看到 UTF-8 编码的缓冲区内容为“U”。

也可给八位位组数组加入多个值来构建一个文本字符串：

```
var buffer = new Buffer([79, 72, 65, 73, 33])
buffer.toString('utf-8')
'OHAI!'
```

如果知道要编码的字符串，可以实例化一个新 Buffer 对象然后将字符串传递给它：

```
var buffer = new Buffer('Look at me mum!')
buffer.toString('utf-8')
'Look at me mum!'
```

一旦初始化完成，缓冲区就不能改变尺寸了，所以要确保缓冲区尺寸足够保存要处理的数据。在本书编写的时候，最大缓冲区尺寸是 1 GB。

## 19.5 写入缓冲区

目前，我们已经创建了一个分配了 8 个字节内存的缓冲区。除非你有盯着空值的嗜好，否则应该往缓冲区里写一些数据。

```
var buffer = new Buffer(8)
buffer.write("a", "utf8")
1
```

这会将字符“a”写入缓冲区，node 返回经过编码以后写入缓冲区的字节数量。字母“a”的 UTF-8 编码占用 1 个字节，但其他字符使用超过 1 个字节。比如，UTF-8 也有许多诸如黑色电话机这样的符号（☎）。

对这个字符编码需要 3 个字节：

```
var buffer = new Buffer(8)
buffer.write('☎', "utf8")
3
```

**TRY IT YOURSELF**

按照下列步骤用字符串来初始化一个缓冲区。

1. 在控制台中输入 `node` 启动 Node.js REPL。
2. 初始化一个 8 个字节的缓冲区：  

```
var buffer = new Buffer(8)
```
3. 以 UTF-8 编码向缓冲区写入一个字符 “c”：  

```
buffer.write('c', 'utf8')
```
4. 可看到这一编码占用 1 个字节。

## 19.6 向缓冲区追加数据

`Buffer` 对象经常用于作为接收到的数据的缓冲区，以便让数据在全部收到后再使用。这意味着在对缓冲区写入的时候我们得能向缓冲区追加数据。默认情况下，在写入缓冲区的时候，它写入第一个字节，或者说缓冲区的开始。这可用下列代码来说明：

```
var buffer = new Buffer(8);
buffer.write('hi', 'utf8');
buffer.toString();
'hi\u0003H?E?H'
buffer.write(' there', 'utf8');
buffer.toString();
' thereas'
```

注意读者可能见到与示例不一样的字符，因为 `Buffer` 代表的是在读者计算机上分配的原始内存。

这显然不是我们想要的！在对缓冲区追加数据时，我们可以传递一个可选的偏移量参数，这样就能在给定的字节或者八位位组上写入缓冲区：

```
buffer.write(' there', 2, 'utf8')
```

**Watch Out!****警告：偏移量值从 0 开始**

在计算科学中，经常从 0 开始计数。对于 `Buffer` 偏移量也是如此。如果想将字符串追加在第 3 个字节，那么偏移量会是 2。

对缓冲区追加数据现在可以写成如下的正确代码了：

```
var buffer = new Buffer(8);
buffer.write('hi', 'utf8');
buffer.toString();
'hi\u0000\u0000\u0000H?'
buffer.write(' there', 2, 'utf8');
buffer.toString();
'hi there'
```

**TRY IT YOURSELF**

按照下列步骤对缓冲区追加数据。



1. 在控制台输入 `node` 启动 Node.js REPL。
2. 初始化一个 8 个字节的缓冲区：  

```
var buffer = new Buffer(8)
```
3. 以 UTF-8 编码向缓冲区写入一个字符串：  

```
buffer.write('hi', 'utf8')
```
4. 检查缓冲区的内容：  

```
buffer.toString()
```
5. 带偏移量向缓冲区写入另一个字符串：  

```
buffer.write(' there', 2, 'utf8')
```
6. 检查缓冲区内容：  

```
buffer.toString()
```
7. 可看到第 2 个字符串追加在第 1 个之后。

## 19.7 复制缓冲区

Node.js 提供了一个将 Buffer 对象整体内容复制到另外一个 Buffer 对象中的方法。我们只能在已经存在的 Buffer 对象之间复制，所以必须先创建它们。要从一个缓冲区复制到另一个缓冲区的方法如下，其中 `buffer` 是要被复制的 Buffer 对象而 `bufferToCopyTo` 是要复制的目标 Buffer 对象：

```
buffer.copy(bufferToCopyTo)
```

以下是将缓冲区的整个内容复制到另一个与之尺寸相同的缓冲区中的最简单的示例：

```
var buffer1 = new Buffer(8)
buffer1.write('hi there', 'utf8')
var buffer2 = new Buffer(8)
buffer1.copy(buffer2)
buffer2.toString()
'hi there'
```

我们可能不想将整个 Buffer 对象复制到另一个对象中，这样的话可以指定想要从源缓冲区复制到目标缓冲区中的内容。Buffer 模块允许我们指定往目标缓冲区中复制的位置：

```
buffer.copy(bufferToCopyTo, 2, 3, 8)
```

第 2 个参数指定在复制时目标缓冲区中应当写入哪个字节。第 3 个和第 4 个参数指定复制时源缓冲区中的开始和停止位置。这可用如下代码来说明：

```
var buffer1 = new Buffer(8)
buffer1.write('hi there', 'utf8')
var buffer2 = new Buffer(8)
buffer2.write('hi', 'utf8');
buffer1.copy(buffer2, 2, 2, 8)
buffer2.toString()
'hi there'
```

## 19.8 修改缓冲区中的字符串

将字符串数据储存在缓冲区中而后需要修改，是开发人员经常碰到的麻烦之一。Buffer

模块没有提供修改字符串的方法。这是合理的，因为缓冲区旨在保存原始数据。如果想修改保存在缓冲区中的字符串，可按如下步骤进行。

- 使用 `toString()` 方法读缓冲区。
- 对 `String` 对象执行任何修改。
- 将修改后的字符串写回缓冲区。

## 19.9 小结

在本章，我们了解了缓冲区，这是 `Node.js` 处理网络和数据的低级 API 之一。我们获得了对二进制数据以及它如何给计算机提供动力的初步知识。我们了解了比特可以组成字节（或者八位位组）并通过字符编码来表示字符。我们介绍了 `ASCII` 和 `UTF-8` 这两种常见的编码系统。我们而后学了如何使用 `Buffer` 模块创建、修改以及复制缓冲区。

## 19.10 问与答

问：缓冲区似乎非常底层，我需要知道这个吗？

答：如果只是用 `Node.js` API 中的更高层次的那些部分，就不会直接接触 `Buffer` 模块，即使它用得频繁。如果必须使用 `Stream`（流）（见第 20 章）的话，就需要使用缓冲区。鉴于缓冲区对 `Node.js` 极基础，至少对它们有工作上的理解是有用的。

问：我应该使用哪种编码？

答：除非有充分的理由，否则不应该不使用 `UTF-8`。如果不提供编码，`Node.js` 也为用户假设使用 `UTF-8`。

问：我应该将缓冲区初始化为多大尺寸？

答：要理解准备添加到缓冲区中的是什么数据及其大小。缓冲区在计算机上分配原始内存，所以用得太多会让系统变慢。如果分配得太少，则可能丢失数据。

问：我是否必须清理对缓冲区对象的引用？

答：不需要。如果对缓冲区引用，系统会自动对它进行垃圾回收。

## 19.11 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

### 19.11.1 问题

1. 用二进制表示 134 这个数。
2. 在 `Node.js` 缓冲区中，偏移量从哪个数开始？

3. 默认情况下，建议使用哪种数据编码格式？

### 19.11.2 答案

1. 10000110 (128+4+2)
2. 在 Node.js 中偏移量从 0 开始。
3. 默认情况下，使用 UTF-8，世界上几乎所有语言都可用它编码。

## 19.12 练习

1. 研究一下 <http://web.cs.mun.ca/~michael/c/ascii-table.html> 中的 ASCII 字符列表。思考十进制值如何与二进制的表示相关。写下一个或多个字符的二进制值。
2. 在终端窗口中输入 `node` 并初始化一个缓冲区。试着理解在创建了缓冲区之后你能使用的是什么样的内存。在缓冲区中加入一些文本然后再次读出。在缓冲区中追加一些文本。
3. 创建一个缓冲区并添加字符串 “Thas as ancorrect”。修改缓冲区，将字符 “a” 更改为字符 “i”。回忆一下我们所学过的关于修改缓冲区中的字符串的方法。



## 第 20 章

# 流模块

---

在本章中你将学到：

- 流是什么；
- 创建一个可读的流；
- 创建一个可写的流；
- 在可读和可写流之间通过管道传输数据。

### 20.1 流简介

在 UNIX 类型的操作系统中，流是个标准概念。有如下三个主要的流。

- 标准输入。
- 标准输出。
- 标准错误。

程序可以从这些流中读写，而因为它们是标准接口，所以可以很容易地将小的、离散的程序连接在一起。在 UNIX 类型的操作系统中，可以对这些流重定向。比如，如果需要的话我们可以将标准输入重定向到标准输出。理解数据在流之间如何流动会有帮助，所以，让我们探索得更深入一点。假设我们有一个名为 `names.txt` 的文本文件，其中有如下的名字清单：

```
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

UNIX 中的 `sort` 工具接收文本行，对其排序，然后返回已排序的版本。它从标准输入获取将要操作的数据，将结果发送到标准输出。于是，要想对上面这个文件排序，可以将标准

输入重定向到 `sort` 命令，如下所示：

```
sort < names.txt
```

#### 提示：Windows 中也有 `sort` 命令

如果读者是 Windows 用户，`sort` 命令在 Windows 中也可用，输入和输出重定向也是一样的。所以这些示例在 Windows 中完全一样工作。

**Did you  
know?**

在 shell 中使用的时候，`<`号（也称为单书名号）表示其右边的无论是什么都应该读入并作为标准输入传递给其左边的任何东西。在本例中，它意味着：

- 以标准输入读入 `names.txt` 的内容；
- 将标准输入重定向到 `sort` 命令。

如果运行这一命令，可看到 `names.txt` 的内容被作为标准输入发送给 `sort` 命令的结果：

```
sort < names.txt
Bailey, Bill
Carr, Jimmy
Smith, Arthur
```

在终端中查看结果实际上就是以标准输入接收的数据经过 `sort` 程序之后以标准输出进行输出的结果。在本例中，标准输出是终端。不过我们也可以重定向标准输出。假设我们要将 `sort` 命令的结果输出到一个文件中：

```
sort < names.txt > sorted_names.txt
```

现在这里多出一个单书名号，它将标准输出重定向到一个文件中。如果文件不存在的话会创建一个，然后标准输出将写入文件中。由于标准输出被重定向，所以在终端上就看不到任何东西了。这里需要思考的重要事情是数据流。数据从 `names.txt` 文件中取出并以标准输入读入。而后被重定向到 `sort` 程序执行排序，然后以标准输出返回结果。最后标准输出被重定向到一个文件上。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour20/example01` 找到。以下步骤演示流的思想。

1. 创建一个名为 `names.txt` 的文件并将下列内容复制到其中：

```
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

2. 打开一个终端窗口并使用如下命令对文件排序：

```
sort < names.txt
```

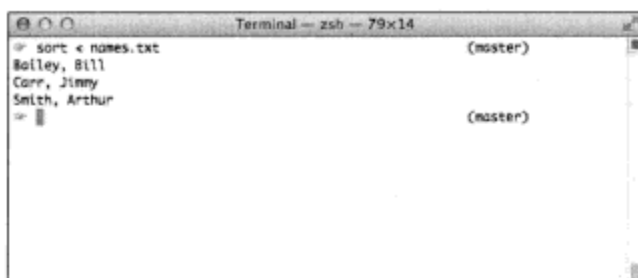
3. 可在终端上看到文件的排序结果（见图 20.1）。
4. 再次排序文件，但这次将输出重定向到一个文件中。

```
sort < names.txt > sorted_names.txt
```

5. `sorted_names.txt` 文件应当已被创建并包含 `names.txt` 的排序结果。

图 20.1

将标准输入重定向  
到 sort 命令



## 20.2 可读流

在第 19 章中，我们了解了缓冲区并知道它是 Node.js 架构的一个基本组成部分。如果缓冲区是 Node.js 处理原始数据的方式的话，那么流通常是 Node.js 移动数据的方式。Node.js 中的流可以是可读的和/或者可写的，它与早先我们看到的标准输入和标准输出松散关联：标准输入是可读的而标准输出是可写的。Node.js 中的许多其他模块使用了流，包括 HTTP 和文件系统。

我们可以在文件系统模块中看到通过使用流来读写文件数据的实例。假设我们想要从早先我们看到的一个名为 `names.txt` 的文件中读入姓名清单，以便使用这些数据。由于数据是流，这就意味着在完成文件读取之前，从收到最初几个字节开始，就可以对数据动作。这是 Node.js 中的一个常见模式：

```
var fs = require('fs');
var stream = fs.ReadStream('names.txt');
stream.setEncoding('utf8');
stream.on('data', function(chunk) {
 console.log('read some data')
});
stream.on('close', function () {
 console.log('all the data is read')
});
```

可读流侦听事件（它实际上是我们在第 18 章中学到的 `EventEmitter` 的一个实例）。在本示例中，在收到新数据时触发事件数据。当文件读取完成后触发关闭事件。如果运行这一脚本，可看到如下输出：

```
read some data
all the data is read
```

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour20/example02` 找到。按如下步骤使用流来读文件。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var fs = require('fs');
var stream = fs.ReadStream('names.txt');
stream.setEncoding('utf8');
```

```
stream.on('data', function(chunk) {
 console.log('read some data')
});
stream.on('close', function () {
 console.log('all the data is read')
});
```

2. 创建一个名为 `names.txt` 的文件并将如下内容复制到其中：

```
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

3. 运行脚本：

```
node app.js
```

4. 可在终端窗口中看到如下内容：

```
read some data
all the data is read
```

这显示，两个事件得到了触发。读取了一些数据，文件读取完成。但姓名在哪儿呢？这里需要指出的重要一点是，在流中，我们要负责按自己想要的方式使用数据，所以我们必须在数据事件接收到数据的时候处理它。如果想读入所有数据，必须将其连接到一个变量中：

```
var fs = require('fs');
var stream = fs.ReadStream('names.txt');
var data = '';
stream.setEncoding('utf8');
stream.on('data', function(chunk) {
 data += chunk;
 console.log('read some data')
});
stream.on('close', function () {
 console.log('all the data is read')
 console.log(data);
});
```

现在如果运行这一脚本的话，可看到事件发生，而姓名也同时显示出来：

```
read some data
all the data is read
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour20/example03` 找到。按下列步骤连接来自流的数据。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var fs = require('fs');
var stream = fs.ReadStream('names.txt');
var data = '';
stream.setEncoding('utf8');
```

```
stream.on('data', function(chunk) {
 data += chunk;
 console.log('read some data')
});
stream.on('close', function () {
 console.log('all the data is read')
 console.log(data);
});
```

2. 创建一个名为 `names.txt` 的文件并将如下内容复制到其中:

```
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

3. 运行脚本:

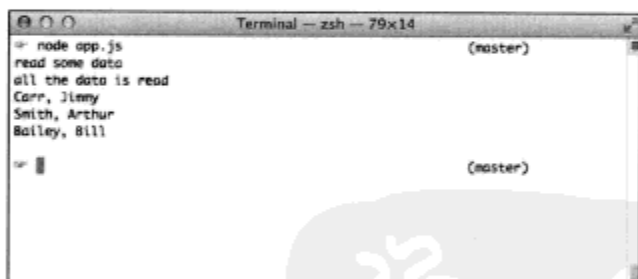
```
node app.js
```

4. 可在终端窗口中看到如下内容 (见图 20.2):

```
read some data
all the data is read
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

图 20.2

使用可读流读取  
数据



对于只有三个姓名的清单, 文件读取时速度很快, 而且只有一个数据事件。但有可能文件是一个大文件并且事件超过一个。这就让开发人员可以在一接收到数据的时候就做一些事情, 而不是等到整个文件都读取完成。比如, 如果将一个更大的文本文件读入可读流, 它生成超过一个数据事件。网站 <http://www.lipsum.com/> 可用来生成拉丁文本。在下一个示例中, 我们使用这个网站生成 1 000 个段落然后将这些段落复制到名为 `latin.txt` 的文件中。和前面一样, 我们通过一个短小的 Node.js 脚本, 使用可读流来读这个文件:

```
var fs = require('fs');
var stream = fs.ReadStream('latin.txt');
stream.setEncoding('utf-8');
stream.on('data', function(chunk) {
 console.log('read some data')
});
stream.on('close', function () {
 console.log('all the data is read')
});
```

这次当脚本运行时, 可见到超过一个数据事件。由于这个文件更大, 所以在收到数据的时候就将数据读入, 触发了超过一个数据事件, 也就意味着我们可以直接使用它。流使得在



接收到数据的时候就开始处理数据成为可能，无论文件有多大：

```
read some data
read some data
read some data
all the data is read
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour20/example04` 找到。下列步骤演示数据事件。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var fs = require('fs');
var stream = fs.ReadStream('latin.txt');
stream.setEncoding('utf-8');
stream.on('data', function(chunk) {
 console.log('read some data')
});
stream.on('close', function () {
 console.log('all the data is read')
});
```

2. 访问 `http://www.lipsum.com/` 并生成 1 000 段拉丁文本。将文本行复制到名为 `latin.txt` 的文件中。

3. 运行脚本：

```
node app.js
```

4. 可在终端窗口中看到如下内容（见图 20.3）：

```
read some data
read some data
read some data
all the data is read
```



图 20.3

有多个数据事件触发的可读流

## 20.3 可写流

显然，我们也可创建可写流以便写数据。这意味着，只要一段简单的脚本，就可以使用流读入文件然后写入另一个文件：

```
var fs = require('fs');
var readableStream = fs.ReadStream('names.txt');
var writableStream = fs.WriteStream('out.txt');
```

```
readableStream.setEncoding('utf8');
readableStream.on('data', function(chunk) {
 writableStream.write(chunk);
});
readableStream.on('close', function () {
 writableStream.end();
});
```

现在，当接收到数据事件时，数据会被写入可写流。这极高效，因为只要从可读文件接收到数据事件，数据就会被写入文件，于是尤其对于大文件而言，这一操作要比不得不等到所有数据读完才能写入文件的方式执行得更快。大体上，这就是流。它们是在网络和文件系统中移动数据的高效方式。

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour20/example05` 找到。以下步骤展示使用可写流的方法。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var fs = require('fs');
var readableStream = fs.ReadStream('names.txt');
var writableStream = fs.WriteStream('out.txt');

readableStream.setEncoding('utf8');
readableStream.on('data', function(chunk) {
 writableStream.write(chunk);
});
readableStream.on('close', function () {
 writableStream.end();
});
```

2. 创建一个名为 `names.txt` 的文件然后将如下内容复制到其中：

```
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

3. 运行脚本：

```
node app.js
```

4. `names.txt` 中的内容应当已经被复制到了 `out.txt` 中。
- 

## 20.4 通过管道连接流

由于在输入和输出之间通过管道传输数据在 Node.js 中很常见，所以它也提供了连接两个可读和可写流并在它们之间通过管道传输数据的方法。以下是 `pipe()` 方法：

```
var fs = require('fs');
var readableStream = fs.ReadStream('names.txt');
var writableStream = fs.WriteStream('out.txt');
readableStream.pipe(writableStream);
```

`pipe()` 方法会仔细处理事件，在需要的时候会暂停流并恢复流操作，所以除非需要对事件

的发生有完全的控制，否则应该使用 `pipe()`。

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour20/example06` 找到。按照下列步骤在可读和可写流之间通过管道传输数据。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```
var fs = require('fs');
var readableStream = fs.ReadStream('names.txt');
var writableStream = fs.WriteStream('out.txt');
readableStream.pipe(writableStream);
```

2. 创建一个名为 `names.txt` 的文件然后将如下内容复制到其中：

```
Carr, Jimmy
Smith, Arthur
Bailey, Bill
```

3. 运行脚本：

```
node app.js
```

4. `names.txt` 中的内容应当已经被复制到了 `out.txt` 中。

#### 注意：缓冲区被用于读写流中的数据

在第 19 章，我们了解了 Node.js 中的缓冲区。在 Node.js 中缓冲区被用于与流一起读写数据，所以对流是如何工作的有一个良好的理解是很值得的。

**By the  
Way**

## 20.5 流的 MP3

流可以与在许多对象上也使用流的 HTTP 模块一起使用。比如这个示例：我们创建一个流的 MP3 服务器。我们所用的 MP3 是来自 `http://www.danosongs.com/` 的一个文件。Dan-O 是一个在 Creative Commons 授权下在线发布音乐的音乐人。我们要使用的文件可以从 `http://www.danosongs.com/music/danosongs.com-rapidarc.mp3` 下载，它也包含在本书的代码示例中。

在 HTTP 模块中，响应对象实际上是一个可写流。它可让文件以可读流的方式读入，然后经过管道成为进入响应对象的可写流，这就和我们看到的复制文件的示例一样。由于 `pipe()` 处理所有需要的暂停和恢复，所以只需几行代码，流 MP3 服务器就建成了：

```
var http = require('http'),
 fs = require('fs');
http.createServer(function(request, response) {
 var mp3 = 'danosongs.com-rapidarc.mp3';
 var stat = fs.statSync(mp3);

 response.writeHead(200, {
 'Content-Type': 'audio/mpeg',
 'Content-Length': stat.size
 });

 var readableStream = fs.createReadStream(mp3);
```

```
readableStream.pipe(response);

}).listen(3000);
```

读者对于 HTTP 模块应当觉得熟悉，这在第 5 章已经介绍过。使用流使得 MP3 文件可以先读然后通过管道输送到响应中。如果启动服务器并浏览端口 3000——如果读者的浏览器支持 MP3 播放的话，将可听到经由流传输播放的 MP3。

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour20/example07` 找到。以下步骤展示了创建流 MP3 服务器的方法。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

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

http.createServer(function(request, response) {
 var mp3 = 'danosongs.com-rapidarc.mp3';
 var stat = fs.statSync(mp3);

 response.writeHead(200, {
 'Content-Type': 'audio/mpeg',
 'Content-Length': stat.size
 });

 var readableStream = fs.createReadStream(mp3);
 readableStream.pipe(response);

}).listen(3000);
```

2. 运行脚本：

```
node app.js
```

3. 打开浏览器访问 `http://127.0.0.1:3000/`。如果浏览器支持 MP3 播放，可听到经由流传输播放的 MP3。

## 20.6 小结

在本章，我们学习了 Node.js 中的另一个高级概念：流。我们看到了 Node.js 流的工作方式与 UNIX 类型的系统的流的工作方式的相似性，以及流在 Node.js 中的使用方法。我们介绍了流在文件系统模块中的使用方法并且学习了如何使用可读流读取文件。而后我们看了在文件系统模块中创建可写流的方法并且看了如何使用流在 Node.js 中复制文件。我们学习了如何使用 `pipe()` 方法作为在可读流和可写流之间移动数据的一种简要方法。最后，我们看了以区区几行代码就实现的一个流 MP3 服务器并且了解了 Node 中有许多对象都是可写流，包括 `http.ServerResponse`，可让可读流直接通过管道与它们连接。

## 20.7 问与答

问：使用流能做什么？

答：流可用于创建代理服务器、流服务器或者操作文件上传的服务，比如图像尺寸调整或者视频编码转换。

问：我什么时候应该自己处理事件，什么时候使用 `pipe()`？

答：可以使用 `pipe()` 将可读流和可写流连接在一起，它会为我们处理暂停和恢复。如果需要对暂停和恢复有更大的控制权，可自己处理这些事件，但对于大多数情况而言，`pipe()` 可为我们处理所有的事情。

问：流似乎比缓冲区更复杂！

答：流是 Node.js 所采用的实现网络编程的方法的一部分，所以理解它们以及它们如何工作是重要的。对于我们的日常编程工作而言，总会有可能使用到流（以及随之的缓冲区）却不知它们的存在。由于流是 Node.js 工作方式的基础，至少对它们的工作原理有一个大致的了解是必须的。

## 20.8 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

### 20.8.1 问题

1. 为什么流是移动数据的高效方式？
2. 是否有可能通过管道连接到超过一个可写流？
3. 流用在哪些类型的事物上？

### 20.8.2 答案

1. 因为来自流的数据在其备妥的同时就可读取，所以数据在整个操作完成之前就可以使用了。这意味着数据可以更快地从 A 到 B 而且开发人员可以在数据块一准备好的时候就开始操作它。

2. 是的。可以将数据通过管道连接到多个流上。这是一个说明这样做的用处的示例：对于上传的文件，可以通过管道连接到一个可写的流，该流将其上传到 Amazon S3；同时将其连接到另一个流将文件保存到磁盘上。

3. 在需要移动数据并且需要尽可能快地操作数据的时候，流是有用的。比如图像尺寸调整服务器、处理大的文本文件或者 HTTP 服务器。

## 20.9 练习

1. 使用 HTTP 模块创建一个脚本下载以下文件：<http://releases.ubuntu.com/lucid/ubuntu-10.04.4-desktop-i386.iso>。使用数据事件在每次收到数据事件时在控制台上记录一条消息。注意数据是如何在一接收到的时候就可使用的。

2. 创建两个 HTTP 服务器，使用流从一个服务器代理另一个服务器的流量。



## 第6部分 进一步的 Node.js 开发

第21章 CoffeeScript

第22章 创建 Node.js 模块

第23章 使用 Connect 创建中间件

第24章 结合使用 Backbone.js 与 Node.js

新学网  
PDG

## 第 21 章

# CoffeeScript

在本章中你将学到：

- CoffeeScript 是什么以及如何使用它；
- CoffeeScript 所提供的超越原生 JavaScript 的是什么；
- 在 CoffeeScript 中使用类；
- 使用 CoffeeScript 的一些潜在问题。

### 21.1 什么是 CoffeeScript

CoffeeScript 是 Jeremy Ashkenas 所编写的 JavaScript 预编译器。它有如下设计目标。

- 清理 JavaScript 语法。
- 给开发人员提供许多能给 JavaScript 添加新功能的、摘取自 Ruby 和 Python 的“好东西”。
- 让 JavaScript 更容易使用并且展现 JavaScript 编程语言的强大功能而无需完全理解 JavaScript 的各种怪癖。

预编译器这个术语的意思是，CoffeeScript 是位于 JavaScript 上面的一层，必须通过编译输出成 JavaScript。过程是这样的：编写 CoffeeScript，编译它，然后得到 JavaScript 输出。如果读者编写过像 C 这样的编程语言（译者注：作者原意显然是“如果读者使用 C 语言编写过程序”），就会熟悉对源代码编译的思想。如果不知道编译的意思，那么在这一章之内就会明白。

我们来快速介绍一下 CoffeeScript。比较一下 Node.js Hello World 服务器。

首先是 JavaScript 的：



```
var http = require('http');
http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/plain'});
 res.end('Hello World!\n');
}).listen(3000, '127.0.0.1');
console.log('Server running at http://127.0.0.1:3000/');
```

然后是 CoffeeScript 的：

```
http = require 'http'
http.createServer (req, res) ->
 res.writeHead 200, 'Content-Type': 'text/plain'
 res.end 'Hello, World!\n'
.listen 3000, '127.0.0.1'
console.log 'Server running at http://127.0.0.1:3000/'
```

从第一眼的印象看，我们看到 CoffeeScript 移除了 JavaScript 的版本中花括号和分号这样的东西。

我们在这一章学习 CoffeeScript 的特性。要想开始使用 CoffeeScript，可在浏览器中访问 <http://jashkenas.github.com/coffee-script/> 来尝试它。

## TRY IT YOURSELF

按下类步骤开始使用 CoffeeScript。

1. 打开浏览器访问 <http://jashkenas.github.com/coffee-script/>。
2. 单击 Try CoffeeScript。现在我们就处于一个交互式编辑器中，CoffeeScript 在左边而 JavaScript 在右边。在左边输入 CoffeeScript 的时候，等价的 JavaScript 显示在右边。
3. 在左栏中输入：  

```
breakfast = 'eggs'
```
4. 在右栏中，等价的 JavaScript 显示了出来（见图 21.1）。可看到：



图 21.1  
CoffeeScript

```
var breakfast;

breakfast = 'eggs';
```

祝贺！你刚刚编写了 CoffeeScript 并且将其编译成了 JavaScript！

## 21.2 安装与运行 CoffeeScript

可以通过 npm 安装 CoffeeScript。因为我们可能希望在文件系统的任何位置上使用 CoffeeScript，所以应该使用 -g 标志将其安装成全局的：

```
npm install -g coffee-script
```

一旦安装了 CoffeeScript，就可以在命令行上使用 coffee 可执行文件。为了检查所有一切是否正确安装，可在终端上运行如下代码：

```
coffee --help
```

应该见到 coffee 命令的帮助文本。

CoffeeScript 使用 .coffee 扩展名。有两种使用 coffee 命令运行 CoffeeScript 的方法。第一种是直接运行 coffee 文件。如果有一个名为 app.coffee 的 CoffeeScript 文件，可以这样运行它：

```
coffee app.coffee
```

运行 CoffeeScript 文件的第二种方法是先将其编译成原生 JavaScript 然后以正常方式运行程序。要编译 CoffeeScript 文件，要使用 -c 选项。这会从 .coffee 文件创建一个 JavaScript 文件：

```
coffee -c app.coffee
node app.js
```

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour21/example01 找到。要运行 CoffeeScript，按下列步骤进行。

1. 创建一个名为 app.coffee 的文件并将下列内容复制到其中：

```
http = require 'http'
http.createServer (req, res) ->
 res.writeHead 200, 'Content-Type': 'text/plain'
 res.end 'Hello, World!\n'
.listen 3000, '127.0.0.1'
console.log 'Server running at http://127.0.0.1:3000/'
```

2. 保存文件。

3. 假设已经使用 npm 全局安装了 CoffeeScript (npm install -g coffee-script)，使用如下命令启动服务器：

```
coffee app.coffee
```

4. 打开浏览器浏览 http://127.0.0.1:3000。

5. 可见到 “Hello, World!”。

在这个示例中，我们使用了 coffee 命令来直接运行 CoffeeScript 而无需将其预编译成 JavaScript。在 Node.js 社区中，这是使用 CoffeeScript 的最常见方式。

也可以将 CoffeeScript 文件预编译成 JavaScript 然后按正常的方式执行 JavaScript 文件。

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example02` 找到。按照如下步骤将 CoffeeScript 编译成 JavaScript。

1. 回到已经创建好的 `app.coffee` 文件。
2. 在终端中运行如下命令：  

```
coffee -c app.coffee
```
3. 注意在同一个目录下会创建一个名为 `app.js` 的新文件。
4. 检查 `app.js` 文件的内容。
5. 使用 JavaScript 文件启动服务器：  

```
node app.js
```
6. 打开浏览器访问 `http://127.0.0.1:3000`。
7. 可见到 “Hello, World!”。

在这个示例中，文件首先被编译成了 JavaScript，而不是直接从 CoffeeScript 运行脚本。如果检查所生成的 JavaScript 文件，可注意到脚本看起来和我们在第 1 章中的基础 Hello World JavaScript 示例不同。这是因为 CoffeeScript 应用了一些关于如何编写出最好的 JavaScript 的观点。总得来说这是件好事，但随着之后在本章中的学习，就会知道这是 CoffeeScript 会惹恼开发人员的许多事情之一。

## 21.3 为什么要使用预编译器

此时，读者一定会想，“我要这个东西做什么？弄得这么复杂！”如果是这样，你应该读完本章，看看 CoffeeScript 能提供的是什麼，然后再做决定！

使用预编译器的一些理由如下所示。

- 避免常见的语法错误。
- 给编程语言加入一些非原生的功能。
- 改进代码语法与可读性。
- 利用预编译器中包含的最优代码编写方法。

如果读者写过 CSS，就可能接触过诸如 Sass 或者 LESS 这样的 CSS 预编译器。这两个工具所做的是与 CoffeeScript 相似的工作。它们抹平了在使用 CSS 时的一些崎岖不平。CSS 中的变量就是一个极好的示例。CSS 不支持声明变量，于是如果有个调色板的话，在每次使用的时候都必须声明颜色值。理论上这没有问题，但当你决定更改调色板并且需要在许多地方更改颜色的时候就不是这样的了。通过使用诸如 Sass 这样的预编译器，可以设置一个变量然后一次一次地使用它，而设置颜色的只有一个地方。

在第 7 章中，我们介绍了 Jade 这个模板引擎。可以对 HTML 进行预编译是模板引擎的一个功能。如果曾经手写过 HTML，那么忘记关闭一个 HTML 标记是不可避免的情况，而这

会导致布局的破坏。通过使用 HTML 预编译器就可以避免这样的问题。

诸如 Sass 和 Haml 这样的工具都能让开发人员更容易地创建 CSS 和 HTML。我们或许不需要它们，但我们可以选择使用它们来改进生产力。CoffeeScript 也是相似的，它可用一点点的复杂性换来生产力的改进。

## 21.4 CoffeeScript 的功能

本节带读者探索 CoffeeScript 的功能并尝试一些示例。

### 21.4.1 最小语法

提供一个 JavaScript 的最小语法版本是 CoffeeScript 的主要功能之一。JavaScript 的语法经常因其过于冗长复杂而受到批评。经常碰到的具体问题有：

- 分号到处都是，忘记它们就会引入错误；
- 必须一遍一遍地声明 `function` 关键字；
- 拖尾的逗号会引入许多意想不到的问题；
- JavaScript 中的变量作用域很容易搞错。

有经验的 JavaScript 程序员会说许多这样的指责都是不公平的，但对于来自诸如 Python 或 Ruby 这样的面向对象语言的程序员，这些问题则是家常便饭。所以 CoffeeScript 从 Python 和 Ruby 继承许多语法上的特性就不足为怪了。

为了了解其工作方式，考虑如下 JavaScript：

```
var cube = function cube(x) {
 return x * x * x;
};
console.log(cube(3));
```

这段简单的脚本将一个函数赋予一个变量，它计算传给它的数的立方。为了展示其工作结果，它将 3 的立方的结果记录到控制台上。

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example03` 找到。以下是在 JavaScript 中计算立方的方法。

1. 创建一个名为 `app.js` 的新文件并粘贴如下代码：

```
var cube = function cube(x) {
 return x * x * x;
};
console.log(cube(3));
```

2. 从终端运行如下脚本：

```
node app.js
```

3. 可看到 27。
-

CoffeeScript 如何表示这一脚本：

```
cube = (x) ->
 x * x * x
console.log cube 3
```

与 JavaScript 比短了吧！这里都发生什么了呢？

- 分号在 CoffeeScript 中不需要。
- 无需用 var 声明变量。它由=号自动指派。
- 无需 function 关键字。函数通过使用箭头 (->) 将参数指向结果来声明。
- 无需花括号。CoffeeScript 中空格是有意义的，通过缩进来推导出花括号。缩进的代码等同于 {}。
- 无需 return 语句。函数的最后一行自动返回（如 Ruby）。
- 无需括号来包裹参数。函数后面的任何东西都被认为是参数。

如果想写得极为简要，甚至可以将函数放在一行内：

```
cube = (x) -> x * x * x
console.log cube 3
```

---

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example04` 找到。以下是在 CoffeeScript 中计算立方的方法。

1. 创建一个名为 `app.coffee` 的新文件并粘贴如下代码：

```
cube = (x) -> x * x * x
console.log cube 3
```

2. 从终端运行如下脚本：

```
coffee app.coffee
```

3. 可看到 27。
- 

## 21.4.2 条件和比较

除了让 JavaScript 的语法变得简短得多以外，CoffeeScript 也给比较添加了许多新方法。考虑在 JavaScript 中是常见模式的用来检查一个变量是否存在以及是否是非空的如下 JavaScript 语句：

```
if (typeof happiness !== "undefined" && happiness !== null){
 alert("I am happy!");
}
```

CoffeeScript 中可以写成这样：

```
alert "I am happy!" if happy?
```

CoffeeScript 允许条件放置在应当返回的语句之前和之后，这是 Ruby 开发人员所熟悉的一个语言特性。它称为前缀和后缀条件，旨在让代码读起来更像自然语言。以下是更多

CoffeeScript 示例:

```
if audience
 sing song

sing song if audience
```

这两段代码都编译成如下 JavaScript:

```
if (audience){
 sing(song);
}
```

CoffeeScript 为比较操作提供了许多别名, 增进了让 CoffeeScript 的语法写起来更像自然语言的能力。表 21.1 展示了 CoffeeScript 和 JavaScript 运算符的对比。

表 21.1 CoffeeScript 和 JavaScript 运算符的对比

CoffeeScript	JavaScript
is	===
isnt	!==
not	!
and	&&
or	
true, yes, on	true
false, no, off	false
@, this	this
of	in
in	无 JavaScript 等同运算符

唯一的一个在 JavaScript 中无等同运算符的是 in。它可用于检查数组中是否有某个元素。通过使用这些别名, 就可以将代码写成这样:

```
start() if light is on
stop() if light isnt on
```

这代码非常接近自然语言, 使用这种编程风格的论点是:

- 更容易阅读与理解代码;
- 代码更简明更容易维护。

读者当然可以不对这些原因买账, 以下是相等的 JavaScript 用于比较:

```
if (light === true) start();
if (light === false) stop();
```

### 21.4.3 循环

在 JavaScript 中, 相对于其他语言而言 for 循环过于冗长。考虑如下 JavaScript:

```
var cheeses = ['Maroilles', 'Brie de Meaux', 'Stinking Bishop'];
for (var i=0; i<cheeses.length; i++) {
 console.log(cheeses[i]);
}
```

这里，声明了一个变量 *i* 用于作为迭代的计数器。迭代的每次循环它都递增 1，当 *i* 比数组的长度小的时候循环重复。这是冗长的！CoffeeScript 用这样的代码表达相同的功能：

```
for cheese in ['Maroilles', 'Brie de Meaux', 'Stinking Bishop']
 console.log cheese
```

它在循环语句中使用缩进将每个奶酪输出到控制台。如果将其编译成 JavaScript，结果的代码会是相似的，但这是将 JavaScript 不太美丽的部分抽象开来的良好示例之一。

实际上，这个示例可以只需一行 CoffeeScript：

```
console.log food for food in ['Maroilles', 'Brie de Meaux', 'Stinking Bishop']
```

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example05` 找到。按照下列步骤展示 CoffeeScript 中的循环。

1. 创建一个名为 `app.coffee` 的文件并将如下代码复制到其中：

```
console.log food for food in ['Maroilles', 'Brie de Meaux', 'Stinking Bishop']
```

2. 保存文件。

3. 从终端运行脚本：

```
coffee app.coffee
```

4. 可见到在终端上打印了奶酪清单。

## 21.4.4 字符串

CoffeeScript 增加了一些字符串功能，从而使得对字符串的处理更简单。通常情况下，在构建一个字符串的时候，我们会想以变量使用字符串。读者可能见过像这样的 JavaScript：

```
var beer, order;
beer = "Greene King IPA"
order = "I would like a " + beer;
console.log(order);
```

CoffeeScript 支持字符串插值。这意味着变量将可在双引号中自动扩展（注意它们不在单引号内扩展）。用 CoffeeScript 重写上一个示例的结果代码如下：

```
beer = "Greene King IPA"
order = "I would like a #{beer}"
console.log order
```

尤其当有许多不同变量需要包括进来的时候，这会成为有用的技巧。

### TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example06` 找到。以下是在

CoffeeScript 中使用字符串插值的方法。

1. 创建一个名为 `app.coffee` 的文件并将下列代码复制到其中：

```
movie = "Willy Wonka & The Chocolate Factory"
string = "My favorite Movie is #{movie}"
console.log string
```

2. 保存文件。
3. 将电影更改为你最喜欢的电影。

4. 从终端运行脚本：

```
coffee app.coffee
```

5. 可看到你最喜欢的电影打印在了终端上。

CoffeeScript 也支持 Heredocs 风格的字符串声明。在原生 JavaScript 中，必须转义引号和撇号。通过 CoffeeScript 带来的 Heredocs 风格会让这样的操作变得方便得多。

在 JavaScript 中，可以这样写 HTML：

```
var html;
html = "<p>\n My awesome HTML\n</p>";
```

注意必须使用特殊的 `\n` 字符来表示换行。在 CoffeeScript 中，可以按在 HTML 中看到的那样写：

```
html = """
 <p>
 My awesome HTML
 </p>
 """
```

## TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example07` 找到。以下是在 CoffeeScript 中使用 Heredocs 的方法。

1. 创建一个名为 `app.coffee` 的文件并将如下代码复制到其中：

```
html = """
 <p>
 Hello World!
 </p>
 """
console.log html
```

2. 保存文件。
  3. 从终端运行脚本：
- ```
coffee app.coffee
```
4. 可看到 HTML 以正确的缩进和新行输出。

21.4.5 对象

CoffeeScript 支持通过 YAML (Yet Another Markup Language) 风格的语法创建对象。在

JavaScript 中，对象的写法如下：

```
var kids;
kids = {
  brother: {
    name: "Max",
    age: 11
  },
  sister: {
    name: "Ida",
    age: 9
  }
}
```

在 CoffeeScript 中，用如下方式表达：

```
kids =
  brother:
    name: "Max"
    age: 11
  sister:
    name: "Ida"
    age: 9
```

CoffeeScript 中的对象语法移除了花括号和逗号，其他的都接近于原生 JavaScript。

21.4.6 类、继承和 super

原型继承 (prototypal inheritance) 是 JavaScript 的难点之一。对于更习惯于经典的类实现编程方法的开发人员而言，学起来会有困难。提供一种创建类结构的简单方法是 CoffeeScript 的好处之一。

要声明一个类，语法如下：

```
class Car
  constructor: (@name) ->
```

这个类可以按如下方法实例化（或者创建）：

```
car = new Car("Audi")
console.log "Car is a #{car.name}"
```

在类上也可以设置实例属性：

```
class Car
  constructor: (@name) ->
  mileage: 81000
```

在类实例化之后就可以通过点号来读取类属性：

```
car = new Car("Audi")
console.log "The #{car.name} has mileage of #{car.mileage}"
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example08` 找到。按照下列步骤

展示 CoffeeScript 中类的构造器。

1. 创建一个名为 `app.coffee` 的文件并将如下代码复制到其中：

```
class Bird
  constructor: (@name) ->
```

2. 在同一个 `app.coffee` 文件中加入如下代码行：

```
bird = new Bird("Robin")
console.log "The bird is a #{bird.name}!"
```

3. 从终端运行脚本：

```
coffee app.coffee
```

4. 可看到如下输出：

```
The bird is a Robin!
```

在面向对象的编程中经常使用子类，也就是一个类从另外一个类继承而来。CoffeeScript 支持继承，意味着可以创建一个从父类继承属性的类。要从另一个类继承，使用 `extends` 关键字：

```
class Human
  constructor: (@legs = 2) ->
  growLeg: ->
    @legs++
```

```
class Horse extends Human
```

在这个示例中，类描述一个人类，构造了两条腿。这个类有一个函数让人类长一条新腿（你总想多长几条腿的吧？）。

使用这个类，我们从 `Human` 继承并创建了第二个类 `Horse`。默认情况下，它会有两条腿。但它可以访问 `growLeg` 函数，所以很容易就可以加腿：

```
horse = new Horse
horse.growLeg()
horse.growLeg()
```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example09` 找到。按下列步骤展示在 CoffeeScript 中使用类的方法。

1. 创建一个名为 `app.coffee` 的文件并将如下代码复制到其中：

```
class Human
  constructor: (@legs = 2) ->
  growLeg: ->
    @legs++

class Horse extends Human

horse = new Horse
horse.growLeg()
horse.growLeg()
console.log "A horse has #{horse.legs} legs"
```

2. 保存文件。
3. 从终端运行脚本：

```
coffee app.coffee
```

4. 可看到如下输出：

```
A horse has 4 legs
```

关于类，最后需要注意的事情是 `super` 关键字。它使得函数可以在子类中修改的同时也可以使用超类中的实现。

```
class Robot
  makeTea: ->
    console.log 'Making tea.'

class Marvin extends Robot
  makeTea: ->
    console.log 'I have a brain the size of a planet'
    super
```

在这个示例中，子类中修改了 `makeTea` 函数，但 `super` 关键字调用了 `super` 类中的函数。注意子类在 `super` 类之前被调用。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour21/example10` 找到。按下列步骤展示在 CoffeeScript 中使用子类的方法。

1. 创建一个名为 `app.coffee` 的文件并将如下代码复制到其中：

```
class Robot
  makeTea: ->
    console.log 'Making tea.'

class Marvin extends Robot
  makeTea: ->
    console.log 'I have a brain the size of a planet'
    super

marvin = new Marvin
marvin.makeTea()
```

2. 保存文件。
3. 从终端运行脚本：

```
coffee app.coffee
```

4. 可看到如下输出：

```
I have a brain the size of a planet
Making tea.
```

CoffeeScript 实现了恰好足够的类结构来提供有用的编程功能。它避免让开发人员不得不理解原型继承并提供了使用类的清晰接口。

如果读者对 JavaScript 还算新人，可在终端中运行如下代码来编译刚刚创建的示例：

```
coffee -c app.coffee
```

将 app.js 的结果作为示例，来看看 CoffeeScript 为我们所抽象的是什么！

21.5 调试 CoffeeScript

难以调试是针对 CoffeeScript 的批评之一。如果只使用 JavaScript，那么调试一个错误的过程如下。

1. 获得错误所在的行号。
2. 编辑在该行号上的代码。
3. 再次运行代码。
4. 重复。

对于 CoffeeScript，过程如下。

1. 获得错误所在的行号。
2. 找到 CoffeeScript 编译的 JavaScript 中的行号。如果读者不是有经验的 JavaScript 开发人员这会更困难，因为 CoffeeScript 按自己的做法创建 JavaScript。
3. 进入 .coffee 文件找到与 JavaScript 行号相关的 CoffeeScript。没有 CoffeeScript 行号可用。
4. 编辑 .coffee 文件中的代码。
5. 重新从 CoffeeScript 编译 JavaScript。
6. 再次运行代码。
7. 重复。

这项显然要比调试原生 JavaScript 复杂得多。由于这个原因，有些开发人员不将 CoffeeScript 预编译成 JavaScript 而是直接从 coffee 命令运行代码。如果将 CoffeeScript 预编译成 JavaScript，过程如下：

```
coffee -c app.coffee
node app.js
```

但是，也可以直接运行脚本，跳过编译成 JavaScript 的过程：

```
coffee app.coffee
```

虽然这样可以捕获语法错误，但我们仍旧必须直接在 CoffeeScript 中调试其他问题，有时候会需要编译成 JavaScript 来看看确切发生的是什么。

关于性能，有几个测试基准可用。但来自 CoffeeScript 创建者的非正式报告认为，不将 CoffeeScript 编译成 JavaScript 对性能的影响不大。

21.6 对 CoffeeScript 的反应

在英国，有一种称为 Marmite 的人们撒在早餐面包上的酵母膏。Marmite 引发强烈的反应——你要么爱它要么恨它。在 Node.js 社区中，对 CoffeeScript 的反应也是如此。如果读者

想试试，可拜访一下#node.js IRC 通道问问大家对 CoffeeScript 怎么看。要健康发言！

对于那些从 Python 和 Ruby 这样的语言切换过来的人而言，CoffeeScript 大体上是受欢迎的。语法相似，无需花费许多年来学习 JavaScript 所有的怪癖。CoffeeScript 用一套熟悉的语法做了所有的重活，输出的是一个经过策划的 JavaScript 版本，避免落入许多常见的编程陷阱。

对于那些编程经验不多的人而言，CoffeeScript 可能不流行。它是令人发指的软件工具清单中的另外一个需要学习的东西。尤其对于那些通过 jQuery 对 JavaScript 有些经验的人，编译到 JavaScript 的额外步骤会是慢吞吞并且不必要的。

对于有经验的 JavaScript 程序员而言，什么反应都有。这些程序员花费了许多年的时间来理解错综复杂的 JavaScript 并花费数小时时间来完成痛苦的调试工作！虽然 CoffeeScript 会避免许多常见的 JavaScript 问题，但它输出的是带有自己的看法的 JavaScript，可能并不与有经验的 JavaScript 开发人员的风格一致。此外，有些人将 CoffeeScript 看作是不相干的编译，如果他们已经是技术高超的 JavaScript 开发人员的话。它经常被看作是用来帮助经验不足的开发人员的玩具。

除了调试更复杂以外，大家还认为用 CoffeeScript 编写 Node.js 项目的话，和其他开发人员共事会难得多。使用 CoffeeScript，就假定项目中的其他所有人都理解 CoffeeScript 并且预编译 JavaScript。大家的指责是这样的：CoffeeScript 给想要在项目上协同工作的人添加了另一道障碍。

JavaScript 的创建者 Brendan Eich 在他的网站上写了如下文字 (<http://brendaneich.com/2011/01/harmony-of-my-dreams/>):

CoffeeScript 做得不错，要比 JS 更方便，只要你对 Python 式的有意义的空格和从另外一个源语言生成 JS 带来的开销买账。

这是一个生动的引述。CoffeeScript 要比 JavaScript 更方便，但其代价就是要学习一种新语法并且需要额外的编译步骤。

21.7 小结

在本章，我们了解了 CoffeeScript 这一 JavaScript 的预编译器。我们了解到可以在 Node.js 项目中使用它，并且检查了诸如语法、循环、对象和类这样的功能。我们了解了使用 CoffeeScript 与编写原生 JavaScript 之间的优缺点。我们了解了 CoffeeScript 所能提供的生产力提升、调试和协作上的问题以及 CoffeeScript 如何绕开 JavaScript 编程语言中更为困难的那些部分。

21.8 问与答

问：别当骑墙派了！我到底该不该用 CoffeeScript？

答：我还是骑墙吧！简而言之就是要看情况。要看你的 JavaScript 技艺水平如何，是否与其他知道 CoffeeScript 的开发人员一起协作，以及是否喜欢 CoffeeScript。CoffeeScript 当然可以带来许多方便，但它也不是没有潜在的问题。

问: CoffeeScript 是否只针对 Node.js?

答: 不是。CoffeeScript 可用在浏览器和许多其他编程语言中。比如, Ruby on Rails 的 3.1 发布版引入 CoffeeScript 作为默认的在 Rails 应用程序中编写 JavaScript 的方法。只要是使用 JavaScript 的地方, 就可以(可能)使用 CoffeeScript。

问: 为什么围绕着 CoffeeScript 有那么多争议?

答: CoffeeScript 是有巨大破坏性的但却是微小的语言。它有可能为一些开发人员除去重写 JavaScript 的需要。对于那些将自己的职业构建在编写 JavaScript 上并且理解该语言所有的难点的人而言, 这是个挑战。这些开发人员理所当然地会指出 CoffeeScript 的瑕疵。相反, 对于那些没有高级 JavaScript 经验的人, CoffeeScript 是一种解放。它让他们可以进入只有专家级程序员才能理解的 JavaScript 的混沌部位。这些 CoffeeScript 的用户将用 CoffeeScript 写所有东西。这通常就是争论所在——如果你喜欢 CoffeeScript, 就会用它来做一切事情。如果将 CoffeeScript 看成是一个玩具和可能挑战你多年学习成果的东西, 那么你就会恨它。

21.9 测验

本测验包含一些问题和习题, 可帮助读者巩固本章所学的知识。

21.9.1 问题

1. 创建 CoffeeScript 的原因有哪些?
2. 使用 CoffeeScript 的优势有哪些?
3. 在项目中使用 CoffeeScript 的潜在问题有哪些?

21.9.2 答案

1. CoffeeScript 的创建是为了将使用 JavaScript 编写应用程序时更为困难的一些部分抽象化。它的创建是为了给开发人员提供更清晰的语法并给 JavaScript 添加一些来自其他编程语言的功能。

2. 通过使用 CoffeeScript, 源代码将更为简明。有些人认为这会更容易理解。可以使用在 JavaScript 语言中不存在的一些功能, 所以我们有可能更快地解决问题。如果选择使用 CoffeeScript 而不使用 JavaScript, 你会是一个更愉快的开发人员!

3. 使用 CoffeeScript 要求在项目中使用预编译器。如果与其他开发人员一起工作的话, 会发现有些人厌恶甚至拒绝使用 CoffeeScript。对于经验比较少的开发人员, 加入一个预编译器等于加入了另一个需要学习的层。

21.10 练习

1. 回到 <http://jashkenas.github.com/coffee-script> 并单击 Try CoffeeScript。试着在左面板上

输入 CoffeeScript 并观察右面板上的输出。试着设置一个变量然后创建一个简单的代表一种动物的类。如果有困难，在主页上有示例可参考。

2. 使用 Node.js HTTP 模块用 CoffeeScript 编写一个小的 Web 服务器。创建 3 个不同的 GET 请求路由。使用 `coffee` 命令运行脚本。

3. 为了熟悉更大型的 CoffeeScript 项目，请访问 <https://github.com/github/hubot>。检查项目中 `src` 文件夹中的 CoffeeScript 文件。试着理解其中的一些源代码并指出其中所使用的类。如果不能理解所有的代码也不要担心——这是一个复杂的项目！如果想深入学习 CoffeeScript，有一本免费的书籍可以阅读：<http://arcturo.github.com/library/coffeescript/>。



第 22 章

创建 Node.js 模块

在本章中你将学到：

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

22.1 为什么创建模块

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

- 为了方便起见并避免一次又一次编写同样的代码。
- 提供在 Node.js 核心中不存在的特定功能。这可以是和第三方 API 交互或者使用 WebSockets。在第 14 章中所创建的流 Twitter 客户端中，我们在 nTwitter 中使用了两个这样的模块来与 Twitter API 和 WebSockets 的 Socket.IO 交互。

22.2 流行的 Node.js 模块

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

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

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

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

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

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

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

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

注意：Node.js 模块本身经常依赖于其他模块

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

**By the
Way**

22.3 package.json 文件

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

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

```
npm init
```

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

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

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

Did you know?

提示：我们可以将选项设置保存在 `~/.npmrc`

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

TRY IT YOURSELF

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

1. 打开终端并输入如下命令：

```
npm init
```

2. 将包名称设为 `ohaihere`。
3. 输入所有能输入的信息。
4. 检查添加的信息并回应说你乐于创建文件。

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

提示：Node.js 模块是 CommonJS 模块

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

Did you Know?

22.4 文件夹结构

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

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

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

提示：.npmignore 和 .gitignore 是互补的

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

Did you Know?

22.5 开发和测试模块

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

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

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

既然有了模块的链接，就可以按下列步骤创建模块了。

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

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

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

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

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

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

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

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

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

6. 这些完成后，就可以使用 `npm test` 来运行测试：

```
npm test

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

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

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

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

8. 如果再次运行这个测试，可看到测试通过！

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

TRY IT YOURSELF

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

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

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

3. 在 `package.json` 文件中添加一行主声明，如下所示：

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

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

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

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

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

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

6. 在终端上通过运行如下命令从模块的根目录下运行测试。应该看到测试失败:

```
npm test
```

7. 打开 `lib/ohaithere.js` 文件并添加如下代码:

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

8. 再次运行测试。可看到测试通过。

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

22.6 添加可执行文件

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

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

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

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

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

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

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

```
npm link
```

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

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

TRY IT YOURSELF

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

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

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

3. 修改 `package.json` 文件，加入可执行文件的声明：
`"bin": "./bin/ohaithere.js"`
4. 从模块的根目录运行 `npm link` 命令将可执行文件链接到系统中：
`npm link`
5. 从终端运行 `ohaithere` 命令：
`ohaithere`
6. 可看到 “Hello from the ohaithere module.”

22.7 使用面向对象或者基于原型的编程

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

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

如果以基于原型的风格来编写这段代码，则它应该成为对象内的一个方法：

```
module.exports = new Ohaithere;

function Ohaithere() {}

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

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

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

```
module.exports = new Ohaithere;

function Ohaithere() {
```

```

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

```

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

TRY IT YOURSELF

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

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

```

module.exports = new Ohaithere;

function Ohaithere(){}

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

```

2. 从模块的根目录运行 `test tests`。可看到测试通过。

22.8 通过 GitHub 共享代码

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

```

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

```

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

```

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

```


读者可在 <http://github.com/shapeded/ohaihere> 看到本例的 GitHub repository (见图 22.1)。

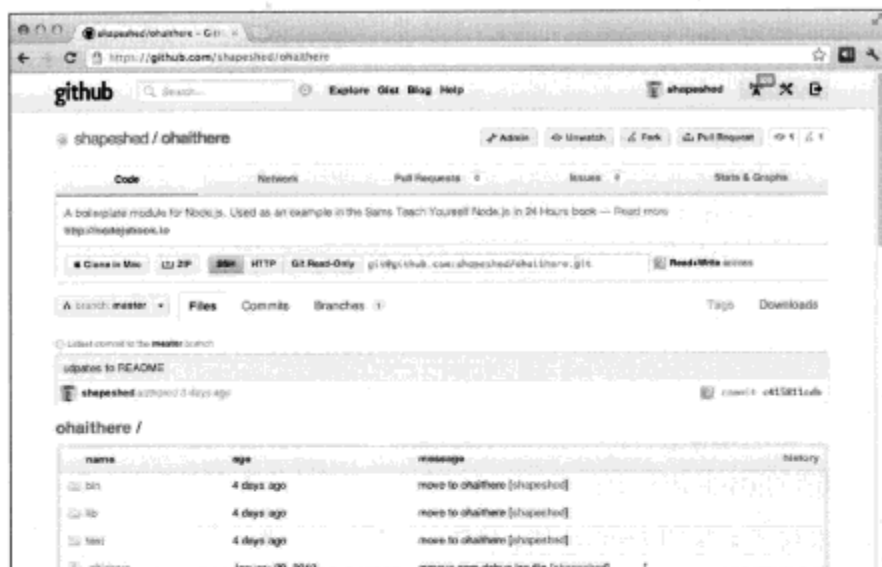


图 22.1

将模块发布到
GitHub

22.9 使用 Travis CI

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

为了使用 Travis CI, 需要如下东西。

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

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



图 22.2

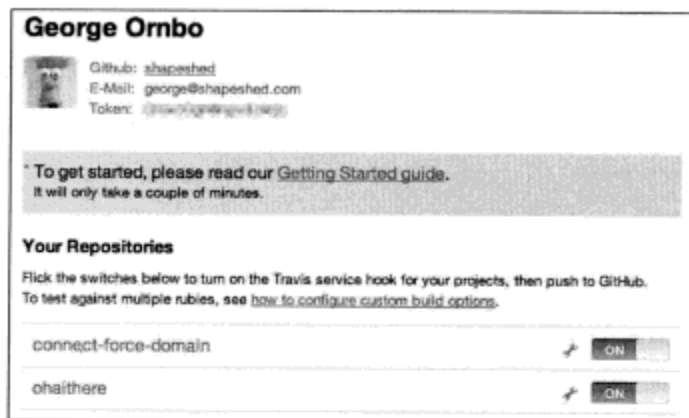
登录 Travis CI

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

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

图 22.3

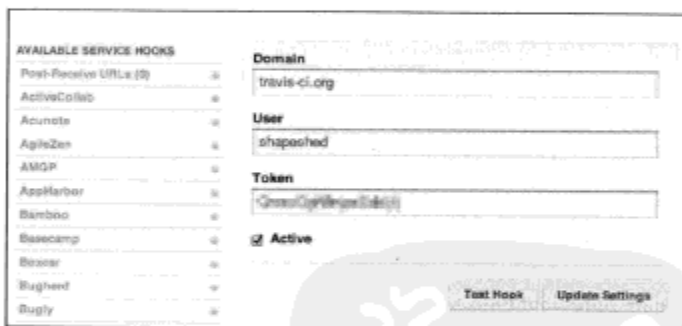
启用库以便 Travis
CI 使用



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

图 22.4

将 Travis CI 添加
到 GitHub 库



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

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

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

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

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

图 22.5)。



图 22.5

正在 Travis CI 上运行的测试

22.10 发布到 npm

到现在，我们已经对模块做了许多工作了！比如：

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

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

```
npm adduser
```

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

```
npm publish
```

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

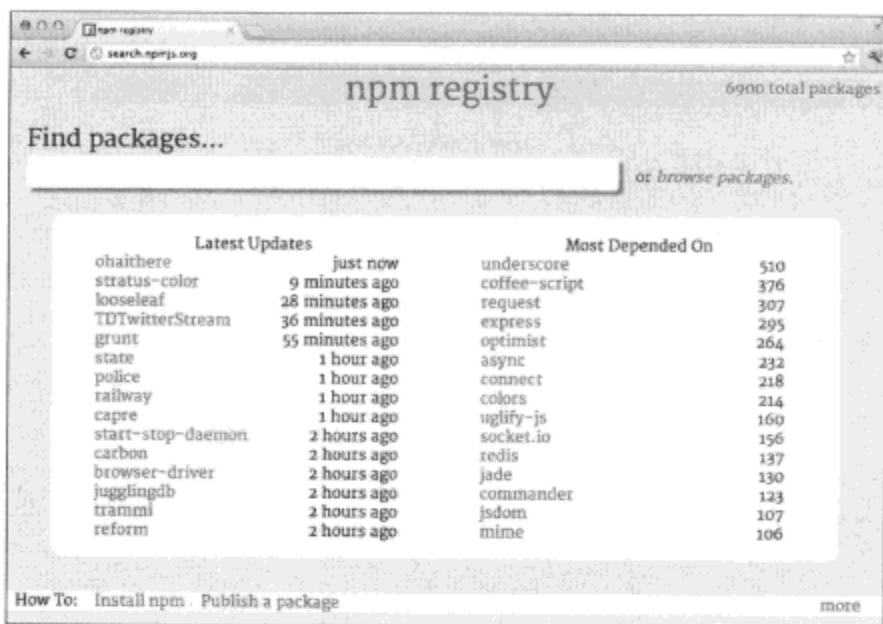
```
npm version 0.0.2
```

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

```
npm publish
```

图 22.6

将模块发布到 npm
注册库



22.11 公开模块

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

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

22.12 小结

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

22.13 问与答

问：我刚刚开始使用 Node.js。我是否必须考虑创建模块的事情？

答：如果在开发中一次又一次遇到相同的问题，首先要做的是检查一下 npm 注册库。可能已经有其他开发人员创建了一个模块。如果没有模块可用，而且你觉得自己编码技艺良好，那么当然可以创建模块！这是给社区做出贡献的绝佳方式！

问：实现抽象的模块是否减低性能？

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

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

问：如果模块已经存在但不完全能实现我想要的功能，我是否应该创建自己的模块？

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

问：我是否必须对模块开源？

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

22.14 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

22.14.1 问题

1. 如何搜索现有的 npm 模块？
2. 在开发一个模块的时候，用来将模块全局地安装在计算机上的命令是什么？
3. 使用 npm 的时候如何取得帮助？

22.14.2 答案

1. 可通过在命令行上使用 `npm search` 或者通过使用 <http://search.npmjs.org/> 上的 Web 界面来搜索现有 npm 模块。
2. 这个命令是 `npm link`，它在计算机上全局安装我们自己的模块。如果想反链接（`unlink`）模块，可运行 `npm unlink` 命令。
3. 可通过运行 `man npm` 来寻找 npm 的帮助。对于各个命令，可运行 `npm help link` 来获得想要运行的命令的全面信息。

22.15 练习

1. 给模块添加另一个名为“goodbye”的方法。它应打印说再见的消息。
2. 在 npm 注册库中浏览一些流行模块的源代码。尝试理解其编码风格并指出使用 `exports` 和 `module.exports` 的地方。
3. 将你的模块以项目发布在 GitHub 上。
4. 将你的项目挂接到 Travis CI 上。

第 23 章

使用 Connect 创建中间件

在本章中你将学到：

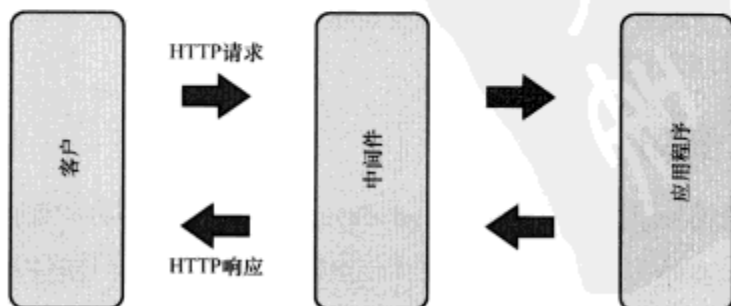
- 中间件是什么；
- 如何与 Connect 和 Express 一起使用中间件；
- 在 Connect 和 Express 应用程序中使用中间件有什么用处；
- 中间件能给开发人员带来什么。

23.1 什么是中间件

在 Node.js 的上下文中，中间件是通过在客户和应用程序逻辑之间添加一个瘦层来过滤应用程序中的请求和响应的一种方法（见图 23.1）。它提供简单的、将应用程序内的所需要考虑的问题分开来的方法，可以带来更可维护的代码、更好的安全模型以及跨项目的代码重用。这个思想相对简单，但它给开发人员带来巨大的能量和灵活性。

图 23.1

中间件如何工作



以下是中间件的一些示例。

- 通过缩进和重新格式化内容来清理 HTML 标记。

- 自动给 HTTP 响应添加 Google Analytics 代码。
- 全功能的授权库。
- 能将错误和异常贴到第三方服务上或者发送成邮件的提示器。
- 在 Web 应用程序上记录日志的日志记录器。
- 监控应用程序的性能。

在 Node.js 中，Web 服务器和应用程序在同一个进程中，所以中间件并不被 Node.js 原生支持，但一个第三方的名为 Connect 的模块给 Node 的 HTTP 模块添加了对中间件的支持。这就让开发人员得以在 Connect 或 Express 应用程序之上添加一个中间件层，以便操纵 HTTP 请求和响应。

23.2 Connect 中的中间件

Connect 是 Node.js 的一个模块，提供中间件框架，让开发人员可以为使用 Connect 创建的应用程序创建中间件。Express，这个我们在第 6 章中介绍的 Web 框架，就是在 Connect 模块之上创建的，所以我们也可在 Express 应用程序中使用中间件。

Connect 包装了来自 Node 的 HTTP 模块的 Server、ServerRequest 和 ServerResponse 对象，以便添加中间件功能。Connect 中的中间件只是一个以请求、响应对象和下一个回调作为参数的函数。我们可以将中间件想成一个函数清单，一个请求在到达应用程序逻辑之前必须流过这个清单中的每个函数。最为基础的中间件示例是一个仅仅将请求往前传递的函数：

```
function nothingMiddleware(req, res, next) {
  next()
}
```

当 next() 回调函数被调用时，中间件就完成了工作，请求被传递到了下一个中间件或者在没有其他中间件的情况下进入应用层。这一示例无甚用处，因为它除了传递请求外不做任何事情；但这里要指出的是一旦位于中间件函数之中，就有可能操作请求的过程。为了演示这一功能，以下是一个 Connect 中的 Hello World 服务器。它与我们在第 1 章中看到的服务器一样，但该服务器现在使用 Connect 以便添加中间件：

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

var app = connect()
  .use(helloWorld);

function helloWorld(req, res){
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hi! Hello World');
}

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

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour23/example01` 找到。以下步骤展示

了一个基础的中间件示例。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

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

var app = connect()
    .use(helloWorld)

function helloWorld(req, res){
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hi! Hello World');
}

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

2. 用下列内容创建一个 `package.json` 文件：

```
{
  "name": "middleware",
  "version": "0.0.0",
  "dependencies": {
    "connect" : "2.0.1"
  }
}
```

3. 安装依赖模块：

```
npm install
```

4. 启动服务器：

```
node app.js
```

5. 打开 Web 浏览器访问 `http://0.0.0.0:3000`。

6. 可在浏览器中看到 `Hi! Hello World`。

在这点上，从功能上说与 Node 的 HTTP 模块没有任何区别。但使用 Connect 包装 HTTP 模块可以将中间件添加到标准 HTTP 模块上。

注意，在本示例中使用了 Node 的 HTTP 模块，而且将 Connect 应用程序传递给了 `http.Server`。Hello World 这一响应实际上也是中间件本身的一部分。

通过在 Connect 服务器中挂接 (mount) 中间件，中间件被添加到了 Connect 应用程序中。可以给整个应用程序挂接中间件，也可以对特别的路由挂接，按照中间件的声明顺序串接在一起。在下面的示例中，在 Connect 应用程序中挂接了中间件并用于对所有请求作出响应：

```
var app = connect()
    .use(connect.favicon());
    .use(connect.logger());
    .use(connect.static(__dirname + '/public'));
```

通过使用中间件，将请求传递了下去，直到返回响应为止。在下面的示例中，第一个中间件 `nothingMiddleware` 仅仅做传递到 `helloWorld` 的工作，而 `helloWorld` 返回响应。通过使用 `res.end`，当响应发送之后即完成了请求的处理。如果读者可理解本示例中请求在中间件中的流向，就可以理解中间件能给开发人员带来什么了：


```

var connect = require('connect'),
    http = require('http');

var app = connect()
  .use(nothingMiddleware)
  .use(helloWorld);

function nothingMiddleware(req, res, next) {
  next();
}

function helloWorld(req, res){
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hi! Hello World');
}

http.Server(app).listen(3000);

```

中间件可用于操纵请求和响应。在下面的示例中，使用了一些中间件来给应用程序所发送的任何响应添加自定义头：

```

var connect = require('connect'),
    http = require('http');

var app = connect()
  .use(addHeader)
  .use(helloWorld);

function addHeader(req, res, next) {
  res.setHeader('X-Custom-Header', 'My header content');
  next();
}

function helloWorld(req, res){
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hi! Hello World');
}

http.Server(app).listen(3000);

```

在 `addHeader` 函数中，更改了响应对象，在将其传递给 `helloWorld` 之前添加了额外的头。这是使用中间件无需进入到应用程序代码就能修改响应的一个简单示例。由于也可将中间件提取到模块中，这就让开发人员可以跨不同的项目重用特定的功能块。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour23/example02` 找到。按照下列步骤使用中间件添加一个自定义头。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```

var connect = require('connect'),
    http = require('http');

var app = connect()
  .use(addHeader)
  .use(helloWorld);

function addHeader(req, res, next) {

```

```

    res.setHeader('X-Custom-Header', 'My header content')
    next();
  }
  function helloWorld(req, res){
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hi! Hello World');
  }

  http.Server(app).listen(3000);

```

2. 创建一个带有如下内容的 package.json 文件:

```

{
  "name": "middleware",
  "version": "0.0.0",
  "dependencies": {
    "connect": "2.0.1"
  }
}

```

3. 安装依赖模块:

```
npm install
```

4. 启动服务器:

```
node app.js
```

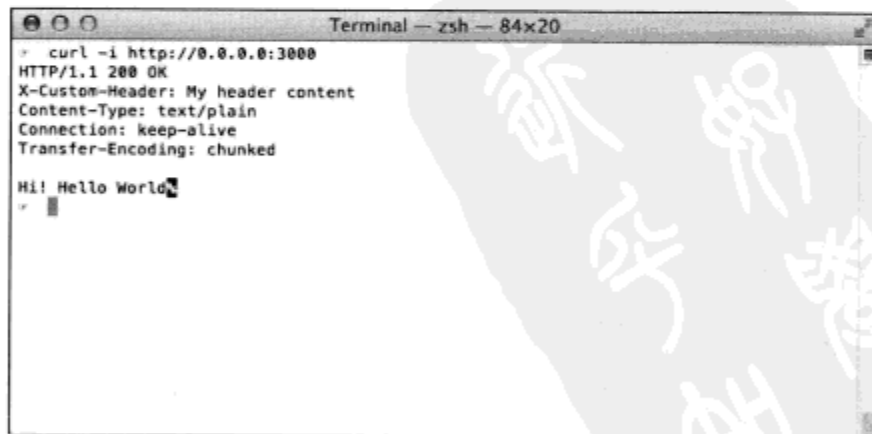
5. 向服务器发送一个 cURL 请求并检查头。如果系统上没有 cURL, 可以在 Google Chrome 中打开该页面并使用 HTTP 头扩展:

```
curl -i http://0.0.0.0:3000
```

6. 可看到中间件在响应中添加了一个自定义头 (见图 23.2)。

图 23.2

使用中间件添加一个自定义头



既然已经对中间件有了更多一些理解, 我们现在可以开始思考中间件可能有哪些用处了。虽然 Connect 是个中间件框架, 但它也带有一些可在常见场景中使用的中间件, 可以让用户对中间件所擅长的工作有很好的洞察。这些包括:

- 用于将信息记录到文件的日志记录器;
- 为页面或站点提供用户名和密码保护的一套基本授权功能;
- 一个 Cookie 分析器;

➤ 跨站点请求伪造保护。

对中间件的需求是对请求或者响应做一些独立于应用程序代码的事情，这些都是良好的示例。由于中间件可以封装在 Node.js 模块中，于是这就可以让应用程序更模块化、更可维护。以下是来自 Connect 源代码的一个示例，它是将 HTTP 请求从一种类型转换成另一种类型的中间件。它通常用于制作来自 Web 浏览器中表单的 PUT 请求。大多数 Web 浏览器中的表单只支持 GET 或 POST 请求，但许多 API 需要 PUT 请求。这是如何将中间件用于在请求到达应用程序代码之前操纵请求的一个良好示例：

```
module.exports = function methodOverride(key) {
  key = key || "_method";
  return function methodOverride(req, res, next) {
    req.originalMethod = req.originalMethod || req.method;

    // req.body
    if (req.body && key in req.body) {
      req.method = req.body[key].toUpperCase();
      delete req.body[key];
    } // check X-HTTP-Method-Override
    } else if (req.headers['x-http-method-override']) {
      req.method = req.headers['x-http-method-override'].toUpperCase();
    }

    next();
  };
};
```

这个中间件与我们在本章早先看到的示例相似，因为 methodOverride 函数以请求、响应和下一个回调作为参数。中间件操作原生 Node.js 方法 request.method 来更改请求的类型。代码检查 req.body.method，如果找到的话就使用它交换出 HTTP 请求方法。这就让开发人员可以创建 RESTful Web 表单。只需几行代码，这个中间件就对 Web 表单的限制问题提供了一个优秀的解决方法，并且让 RESTful Web 服务得以与 HTML 表单一起使用。

23.3 使用中间件的访问控制

使用中间件是管理访问控制的良好选择。许多应用程序将访问控制逻辑直接放在应用程序代码中。通过将访问控制提取到中间件层，可在进入应用程序逻辑之间控制对应用程序的访问。许多开发人员认为这是一个更为安全和优雅的解决方案。在下列示例中，假设我们想按照一天中的时间来限制对网站的访问。如果时间是在早晨 9 点到夜里 5 点之间，则网站可以打开。如果在这些时间以外，则网站关闭。给这个场景编写中间件的一种方法如下：

```
function nineToFive(req, res, next) {
  var currentHour = new Date().getHours();
  if (currentHour < 9 || currentHour > 17) {
    res.writeHead(503, {'Content-Type': 'text/plain'});
    res.end('We are closed. Come back between 0900 and 1700.');
```

```
  } else {
    next();
  }
}
```

随着请求流经中间件，会对当前的时间进行检查，确认是否在 9 点到 5 点的网站工作时间之外。如果在这些时间之外，则返回 503 HTTP 响应表示服务暂时不可用。对 `res.end` 的使用结束了这一过程并将响应返回给客户。如果当前时间不在 9 点到 5 点之外，则将请求传递到下一个中间件。以下是这一中间件的完整示例：

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

var app = connect()
  .use(nineToFive)
  .use(helloWorld);

function nineToFive(req, res, next) {
  var currentHour = new Date().getHours();
  if (currentHour < 9 || currentHour > 17) {
    res.writeHead(503, {'Content-Type': 'text/plain'});
    res.end('We are closed. Come back between 0900 and 1700.');
```

```
  } else {
    next();
  }
}

function helloWorld(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hi! We are open!');
```

```
}

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

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour23/example03` 找到。按照下列步骤使用中间件来实现访问控制。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

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

var app = connect()
  .use(nineToFive)
  .use(helloWorld);

function nineToFive(req, res, next) {
  var currentHour = new Date().getHours();
  if (currentHour < 9 || currentHour > 17) {
    res.writeHead(503, {'Content-Type': 'text/plain'});
    res.end('We are closed. Come back between 0900 and 1700.');
```

```
  } else {
    next();
  }
}

function helloWorld(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hi! We are open!');
```

```

}

http.Server(app).listen(3000);

```

2. 创建一个带有如下内容的 `package.json` 文件:

```

{
  "name": "middleware",
  "version": "0.0.0",
  "dependencies": {
    "connect" : "2.0.1"
  }
}

```

3. 安装依赖模块:

```
npm install
```

4. 启动服务器:

```
node app.js
```

5. 打开浏览器访问 `http://0.0.0.0:3000`。

6. 如果在早晨 9 点到晚上 5 点之间查看该网页, 可看到 “Hi! We are open!” (见图 23.3)。如果不是, 可见到 “We are closed. Come back between 0900 and 1700.”。

7. 尝试停止服务器, 修改时间, 然后重启服务器, 再打开页面, 这样就能看到其工作结果。



图 23.3

用中间件实现基于时间的访问

23.4 按 IP 地址限制访问

当客户对 Node.js 服务器发出请求时, 可以在 `req.connection.remoteAddress` 中得到请求源于哪个 IP 地址。这里的 `req` 是请求对象。这可以用来创建一些中间件, 按照请求来源的 IP 地址来过滤对应用程序的访问。在本示例中, 我们要看如何通过允许参数的传递来让中间件更为灵活:

```

function filterByIp(ips){
  var ips = ips || [];
  return function (req, res, next){
    if (ips.indexOf(req.connection.remoteAddress) == -1) {

```

```

    res.writeHead(401, {'Content-Type': 'text/plain'});
    res.write('Sorry. You are not allowed to access this server');
    res.end();
  } else {
    next();
  }
};
};
};

```

如果请求 IP 地址不在允许的 IP 地址数组中，则返回 401 响应给客户表示他们不允许访问该服务器。调用 `res.end()` 确保请求完成。如果 IP 地址在 IP 地址白名单列表中找到，则请求通过。在下面的示例中，只能从 127.0.0.1 访问 Node.js 服务器：

```

var connect = require('connect'),
    http = require('http');

var app = connect()
  .use(filterByIp(['127.0.0.1']))
  .use(helloWorld);

function filterByIp(ips){
  var ips = ips || [];
  return function (req, res, next){
    if (ips.indexOf(req.connection.remoteAddress) == -1) {
      res.writeHead(401, {'Content-Type': 'text/plain'});
      res.write('Sorry. You are not allowed to access this server');
      res.end();
    } else {
      next();
    }
  };
};

function helloWorld(req, res){
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('You can view this!');
}

http.Server(app).listen(3000);

```

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour23/example04` 找到。按照下列步骤使用中间件按 IP 地址进行过滤。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```

var connect = require('connect'),
    http = require('http');

var app = connect()
  .use(filterByIp(['127.0.0.1']))
  .use(helloWorld);

function filterByIp(ips){
  var ips = ips || [];
  return function (req, res, next){
    if (ips.indexOf(req.connection.remoteAddress) == -1) {

```

```

    res.writeHead(401, {'Content-Type': 'text/plain'});
    res.write('Sorry. You are not allowed to access this server');
    res.end();
  } else {
    next();
  }
};

function helloWorld(req,res){
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('You can view this!');
}

http.Server(app).listen(3000);

```

2. 创建一个如下内容的 package.json 文件:

```

{
  "name": "middleware",
  "version": "0.0.0",
  "dependencies": {
    "connect" : "2.0.1"
  }
}

```

3. 安装依赖模块:

```
npm install
```

4. 启动服务器:

```
node app.js
```

5. 打开浏览器访问 <http://0.0.0.0:3000>。

6. 可看到我们可以访问服务器。

7. 停止服务器, 将第 5 行的 IP 地址更改为 127.0.0.2。

8. 重新启动服务器:

```
node app.js
```

9. 打开浏览器访问 <http://0.0.0.0:3000>。

10. 可看到我们不能再访问服务器了 (见图 23.4)。

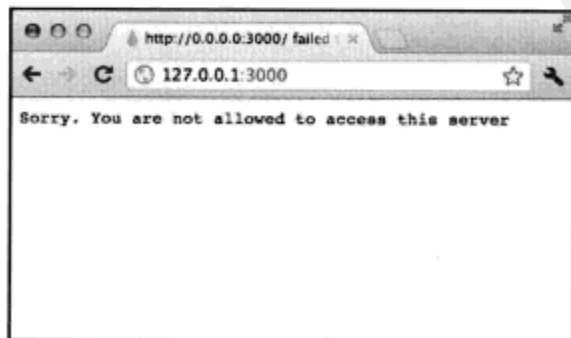


图 23.4

使用中间件实现基于 IP 的访问

23.5 将用户强制到单个域上

许多平台即服务 (PaaS) 提供商都可以给应用程序添加自定义域 (custom domain)。比如, 托管在 Heroku 平台上的一个应用程序可能会被赋予 `http://smooth-light324.herokuapp.com` 这样的 URL。我们可能会需要让站点以我们自己的域名 (比如 `fooboo.com`) 来提供服务, 于是我们会使用自定义域外挂并且为 `fooboo.com` 设置一条 CNAME 记录以便从我们自己的域为站点提供服务。如此, 站点将可以从如下的两个域访问。

- `http://smooth-light324.herokuapp.com`。
- `http://fooboo.com`。

如果只是在做测试的话, 这不会有问题; 但如果这是个产品站点, 那么应该真正地将所有流量都导向单个域, 因为这对 SEO (搜索引擎优化) 不利。Google 已经公开表示重复的内容会被处罚并鼓励 Web 开发人员使用 301 重定向对存在于不同域上的相同内容的请求进行重定向。通过使用中间件, 写一小段代码就可以解决这个问题:

```
function forceDomain(domain){
  var domain = domain || false;
  return function (req, res, next){
    if (domain && (req.headers.host !== domain)){
      res.writeHead(301, {"Location": 'http://' + domain + req.url});
      res.end();
    } else {
      next();
    }
  };
};
```

在这个示例中, 中间件接收一个域作为参数然后对请求头进行检查, 查看这是否匹配。如果匹配, 则传递请求。如果不匹配, 则使用 301 响应代码重定向请求到正确的域。该中间件的完整示例如下:

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

var app = connect()
  .use(forceDomain('127.0.0.1:3000'))
  .use(helloWorld);

function forceDomain(domain){
  domain = domain || false;
  return function (req, res, next){
    if (domain && (req.headers.host !== domain)){
      res.writeHead(301, {"Location": 'http://' + domain + req.url});
      res.end();
    } else {
      next();
    }
  };
};

function helloWorld(req, res){
```



```

res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World');
}

http.Server(app).listen(3000);

```

By the Way

注意：301 重定向用户和搜索引擎

301 状态码表示一个永久的重定向并让搜索引擎知道一个资源现在已经永久地被重定向到另一个资源。对于用户而言，它们会被无缝地重定向到新资源上。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour23/example05` 找到。按照下列步骤使用中间件强制单个域。

1. 创建一个名为 `app.js` 的文件并将如下代码复制到其中：

```

var connect = require('connect'),
    http = require('http');

var app = connect()
  .use(forceDomain('127.0.0.1:3000'))
  .use(helloWorld);

function forceDomain(domain){
  domain = domain || false;
  return function (req, res, next){
    if (domain && (req.headers.host !== domain)){
      res.writeHead(301, {"Location": 'http://' + domain + req.url});
      res.end();
    } else {
      next();
    }
  };
};

function helloWorld(req,res){
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World');
}

http.Server(app).listen(3000);

```

2. 创建一个带有如下内容的 `package.json` 文件：

```

{
  "name": "middleware",
  "version": "0.0.0",
  "dependencies": {
    "connect" : "2.0.1"
  }
}

```

3. 安装依赖模块：

```
npm install
```

4. 启动服务器:

```
node app.js
```

5. 打开浏览器访问 `http://0.0.0.0:3000`。6. 可在地址栏看到被重定向到 `http://127.0.0.1:3000`。

23.6 小结

在本章，我们介绍了中间件。我们看到了与 `Connect` 一起使用中间件在无需触碰应用程序代码的情况下操纵请求和响应的可能性。我们了解了中间件是客户和应用程序之间的一个瘦层，它带来许多能力。我们看到如何给响应添加头、按时间限制访问、按 IP 地址限制访问以及如何将所有用户强制到单个域上。通过这些示例，我们学习了如何给中间件传递参数并了解了请求如何流过中间件。

23.7 问与答

问：我是否可以总是使用中间件？

答：中间件极适合于操纵请求和响应。应该把它与应用程序分开来考虑，于是，处理数据或者查询数据库的事情就不应该放在中间件中。

问：中间件的顺序是否重要？

答：是的。一个请求要按中间件的声明顺序流过中间件。随着在应用程序中添加更多的中间件，我们需要对此知晓，因为如果中间件以不正确的顺序声明的话，会带来意料之外的结果。

问：我可以在 `Connect` 和 `Express` 之外使用中间件吗？

答：`JSGI` 是另外一种在 `Node.js` 中使用中间件的方法，有一个模块可以在 `Node.js` 中使用它。更多信息请看 <https://github.com/persvr/jsgi-node>。读者也可不通过框架编写中间件，但这超过了本书范畴。

23.8 测验

本测验包含一些问题和习题，可帮助读者巩固本章所学的知识。

23.8.1 问题

1. 中间件用来做哪些类型的事情？
2. 停止一个中间件有哪些可用的选项？
3. `Connect` 是否是在 `Node.js` 中使用中间件的唯一方式？

23.8.2 答案

1. 中间件用来做诸如验证、日志记录、缓存异常通知器 (notifier) 以及重定向这样的事情。它是个瘦层，位于应用程序逻辑之前，可操纵请求和响应。
2. 可调用 `next()` 将请求和响应对象传递到下一个中间件 (或者应用程序逻辑——如果是最后一个中间件的话)。可使用 `res.end()` 来返回响应。
3. 不是。Connect 让中间件的创建极为容易，但我们也可使用 JSGI 或者自己编写中间件。

23.9 练习

1. 为 Connect 创建一个中间件函数用于将浏览器版本记录到控制台。提示：可通过 `req.headers['user-agent']` 得到。
2. 阅读位于 <https://github.com/senchalabs/connect/tree/master/lib/middleware> 上的 Connect 库中的中间件源代码。试着理解如何操纵请求。
3. 探究列在 Node.js Wiki 上的中间件，地址是 <https://github.com/joyent/node/wiki/modules#wiki-middleware>。试着理解这些模块所做的是什麼，并阅读一个或多个项目的源代码。



第 24 章

结合使用 Backbone.js 与 Node.js

在本章中你将学到：

- 什么是 Backbone.js;
- 如何结合使用 Backbone.js 与 Node.js;
- 使用 Node.js 和 Backbone.js 创建一个简单的单页面应用程序;
- 创建一个简单的 Backbone.js 视图。

24.1 什么是 Backbone.js

Backbone.js 是前端 JavaScript 框架，用于创建现代的、基于浏览器的 Web 应用程序。它旨在帮助开发人员在浏览器中创建桌面类型的应用程序。它可用于创建单页面应用程序或者有许多 URL 的应用程序。它旨在与 jQuery 或者 Zepto 集成并让开发人员可以创建在用户更新数据时无需刷新页面的应用程序。

随着浏览器的能力越来越强，开发人员已经将更多的应用程序逻辑推给了浏览器，依靠 Ajax 发送并接收来自服务器的更新。这让用户可以享受到非常快速的体验，与桌面应用程序齐头并进。在服务器端，开发人员已经可以使用诸如 Rails 或者 Django 这样的成熟框架来创建 Web 应用程序，但直到最近，能用来创建富客户端 Web 应用程序的框架少之又少。

Backbone.js 是以帮助开发人员使用 JavaScript 创建富客户端 Web 应用程序为目的的一个框架。它通过提供一个重 JavaScript 应用程序的结构并使开发人员极易于构建和维护客户端的 JavaScript 应用程序来实现这一点。

将 JavaScript 带到服务器端并将事件驱动的动态语言的所有能力带到网络编程中，是使用 Node.js 的一个令人兴奋的事情之一。能使用 JavaScript 创建客户端和服务器的思想让 JavaScript 的开发人员得以成为“全栈”开发人员。这就意味着一个 JavaScript 开发人员可以创建服务器和

富客户端。如果读者是一个 JavaScript 开发人员这真是个好消息了！此外，通过使用诸如 MongoDB 或者 CouchDB 这样的数据存储，我们可使用 JavaScript 来储存并与数据交互。在这一上下文中，整个应用程序——从服务器到数据库再到客户端，都可使用 JavaScript 来创建。

如果读者正在创建 Node.js 应用程序，很有可能在某些时候会需要给 Node.js 服务器创建一个富的基于 JavaScript 的客户端。Backbone.js 作为一个客户端应用程序框架，它所期望的就是会和服务器端的应用程序通信。默认情况下，它期望的是一个基于 JSON 的 API，与我们在第 15 章中所学习的非常类似。Node.js 是 Backbone.js 应用程序服务器端的一个优秀选择，因为它是轻量级的、快速的并且是 JavaScript 的！

在本章，我们学习如何创建 Backbone.js 应用程序的服务器端元素并且对 Backbone.js 如何工作有一个概要的了解。

注意：Backbone.js 的创建者也创建了 Underscore.js 和 CoffeeScript

Backbone.js 由 Jeremy Ashkenas 创建。他也创建了 Underscore.js，这是一个增强 JavaScript 语言本身的库，它也在 Backbone.js 中使用。Jeremy Ashkenas 也创建了 CoffeeScript，这是我们在第 21 章介绍的 JavaScript 预处理器。

**By the
Way**

24.2 Backbone.js 如何工作

为了理解如何与 Backbone.js 一起使用 Node.js，首先需要理解 Backbone.js 如何靠近客户端 Web 应用程序。Backbone.js 围绕着模型（Model）和集合（Collection）架构应用程序中的数据。如果读者使用过诸如 Rails 或者 Django 这样的 Web 应用程序框架，应该熟悉这些概念。如果没有使用过这些框架的话，可以这样看待一个简单的 Backbone.js 应用程序：一个待办事宜列表。

在待办事宜列表中，Backbone.js 将任务（或者待办事宜项）当成模型来考虑，任务的列表当成任务的集合来考虑。简单地说，模型是单个项目，而集合是一组项目。在一个待办事宜类型的应用程序中，单个任务是一个模型，而所有任务就是一个集合。

Backbone.js 期望与使用 HTTP 的服务器端 JSON API 以及和第 15 章所介绍过的同样的 GET、POST、PUT 和 DELETE 请求交互。默认情况下，Backbone.js 期望路由遵循 RESTful 惯例。对于一个 `/api/tasks` 的 API 终端，路由如下。

- GET `/api/tasks`: 获得所有任务。
- GET `/api/tasks/:id`: 获得单个任务。
- POST `/api/tasks`: 创建一个新任务。
- PUT `/api/tasks/:id`: 更新一个任务。
- DELETE `/api/tasks/:id`: 删除一个任务。

警告：

这些示例要求机器上的 MongoDB 必须处于运行中。如果需要复习这些知识，请参阅第 15 章。

**Watch
Out!**

通过使用 MongoDB 和 Express，我们可以创建一个简单的 API 让 Backbone.js 与之交互。Express 应用程序将有如下文件夹结构：

```

├─ app.js
├─ package.json
├─ public
│   └─ javascripts
│       └─ application.js
│   └─ stylesheets
│       └─ style.css
└─ views
    └─ index.jade

```

这个应用程序使用 Mongojs 来与 MongoDB 交互，所以 package.json 文件如下：

```

{
  "name": "backbone_example",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "2.5.8",
    "mongojs": "0.4.3",
    "jade": "0.22.0"
  }
}

```

主 app.js 文件是一个简单的 Express 应用程序，它为 Backbone.js 客户端提供 API 服务的同时也提供单个 HTML 页面的服务。这与我们在第 15 章看到的基于 JSON 的 API 相似：

```

var express = require('express'),
    db = require("mongojs").connect('backbone_tasks', ['tasks']);

var app = module.exports = express.createServer();
app.use(express.bodyParser());
app.use(express.static(__dirname + '/public'));
app.use(express.errorHandler({dumpExceptions: true, showStack: true}));

app.get('/', function(req, res){
  res.render('index.jade', {
    layout: false
  });
});

app.get('/api/tasks', function(req, res){
  db.tasks.find().sort({ $natural: -1 }, function(err, tasks) {
    res.json(tasks);
  });
});

app.get('/api/tasks/:id', function(req, res){
  db.tasks.findOne( { _id: db.ObjectId(req.params.id) } , function(err, task) {
    res.json(task);
  });
});

app.post('/api/tasks', function(req, res){
  db.tasks.save(req.body, function(err, task) {
    res.json(task, 201);
  });
});

```

```

app.put('/api/tasks/:id', function(req, res){
  db.tasks.update( { _id: db.ObjectId(req.params.id) }, { $set: { title: req.body.
title } }, function(err, task) {
    res.json(200);
  });
});

app.del('/api/tasks/:id', function(req, res){
  db.tasks.remove( { _id: db.ObjectId(req.params.id) }, function(err) {
    res.send();
  });
});

app.listen(3000);

```

这包括 Backbone.js 所需的路由以及应用程序所需的作为 Backbone.js 客户端的单个 HTML 页面。在 index.jade 文件中添加了如下内容：

```

!!! 5
html
  head
    title Node.js / Backbone.js Example
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    h1 Tasks
    div#tasks
      script(src='https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js')
      script(src='http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.1/
➤ underscore-min.js')
      script(src='http://cdnjs.cloudflare.com/ajax/libs/backbone.js/0.9.1/backbone-
➤ min.js')
      script(src='/javascripts/application.js')

```

这包括 jQuery、Underscore.js、Backbone.js 和一个对 application.js 文件的链接，Backbone.js 代码从这个文件开始执行。最后，application.js 文件的内容如下：

```

var App = {};

App.Task = Backbone.Model.extend({
  idAttribute: "_id",
});

App.Tasks = Backbone.Collection.extend({
  model: App.Task,
  url: '/api/tasks'
});

```

在这段代码中，声明了一个空 App 对象。这是为了给 Backbone.js 应用程序声明一个命名空间以减少与其他库的变量冲突的风险。接下来，声明了一个 Backbone.js 模型，而且由于使用着 MongoDB，所以将 idAttribute 正确地设置成了“_id”。最后，声明了一个 Backbone.js 集合，引用着刚刚创建的模型并设置要发送和接收数据的 URL。Backbone.js 在背后处理所有的 GET、POST、PUT 和 DELETE 方法，所以我们只需做上面这些工作了。

现在，一个 Backbone.js 骨架已经具备，假设 MongoDB 已经安装并运行于我们的计算机上，我们现在可以启动 Node.js 服务器并打开浏览器访问 http://0.0.0.0:3000。在这一步，除了一个占位符页面以外，应用程序还没有视图，但为了获得对 Backbone.js 如何工作的理解，我

们可以开始在浏览器中从 JavaScript 控制台与 Node.js 交互。

有了 Node.js 服务器在运行, 可在浏览器中打开一个 JavaScript 控制台, 这样就可以开始与 Node.js API 交互。我们可以通过运行如下代码从 JavaScript 控制台创建一个新的任务:

```
var tasks = new App.Tasks;
tasks.create({ title: "Test Task" });
```

我们将可在控制台中看到 Backbone.js 为我们处理邮递数据 (它使用 jQuery 来实现) 并将其寄送给 Node.js API。Backbone.js 也处理响应。

TRY IT YOURSELF

如果下载了本书的代码示例, 那么这段代码可在 `hour24/example01` 找到。以下步骤展示了一个基础的 Backbone 应用程序。

1. 创建一个带有如下结构的 Express 项目:

```
.
├─ app.js
├─ package.json
├─ public
│   └─ javascripts
│       └─ application.js
│   └─ stylesheets
│       └─ style.css
└─ views
    └─ index.jade
```

2. 在 `package.json` 文件中, 添加如下内容:

```
{
  "name": "backbone_example",
  "version": "0.0.1",
  "private": true,
  "dependencies": {
    "express": "2.5.8",
    "mongojs": "0.4.3",
    "jade": "0.22.0"
  }
}
```

3. 在 `app.js` 文件中, 添加如下内容:

```
var express = require('express'),
    db = require("mongojs").connect('backbone_tasks', ['tasks']);

var app = module.exports = express.createServer();
app.use(express.bodyParser());
app.use(express.static(__dirname + '/public'));
app.use(express.errorHandler({dumpExceptions: true, showStack: true}));

app.get('/', function(req, res){
  res.render('index.jade', {
    layout: false
  });
});

app.get('/api/tasks', function(req, res){
  db.tasks.find().sort({ $natural: -1 }, function(err, tasks) {
```



```

        res.json(tasks);
    });
});

app.get('/api/tasks/:id', function(req, res){
    db.tasks.findOne( { _id: db.ObjectId(req.params.id) } , function(err, task)
    {
        res.json(task);
    });
});

app.post('/api/tasks', function(req, res){
    db.tasks.save(req.body, function(err, task) {
        res.json(task, 201);
    });
});

app.put('/api/tasks/:id', function(req, res){
    db.tasks.update( { _id: db.ObjectId(req.params.id) }, { $set: { title: req.
body.title } }, function(err, task) {
        res.json(200);
    });
});

app.del('/api/tasks/:id', function(req, res){
    db.tasks.remove( { _id: db.ObjectId(req.params.id) }, function(err) {
        res.send();
    });
});

app.listen(3000);

```

4. 在 index.jade 文件中, 添加如下内容:

```

!!! 5
html
  head
    title Node.js / Backbone.js Example
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    h1 Tasks
    div#tasks
      script(src='https://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.
js')
      script(src='http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.3.1/
underscore-min.js')
      script(src='http://cdnjs.cloudflare.com/ajax/libs/backbone.js/0.9.1/
backbone-min.js')
      script(src='/javascripts/application.js')

```

5. 在 application.js 文件中, 添加如下内容:

```

var App = {};

App.Task = Backbone.Model.extend({
  idAttribute: "_id",
});

App.Tasks = Backbone.Collection.extend({

```

```

    model: App.Task,
    url: '/api/tasks'
  });

```

6. 安装所需的依赖模块:

```
npm install
```

7. 通过从终端运行如下命令来启动服务器:

```
node app.js
```

8. 打开带有 JavaScript 控制台的浏览器访问 <http://127.0.0.1:3000>。

9. 打开 JavaScript 控制台并输入如下内容; 然后按 Return (Enter) 运行代码:

```

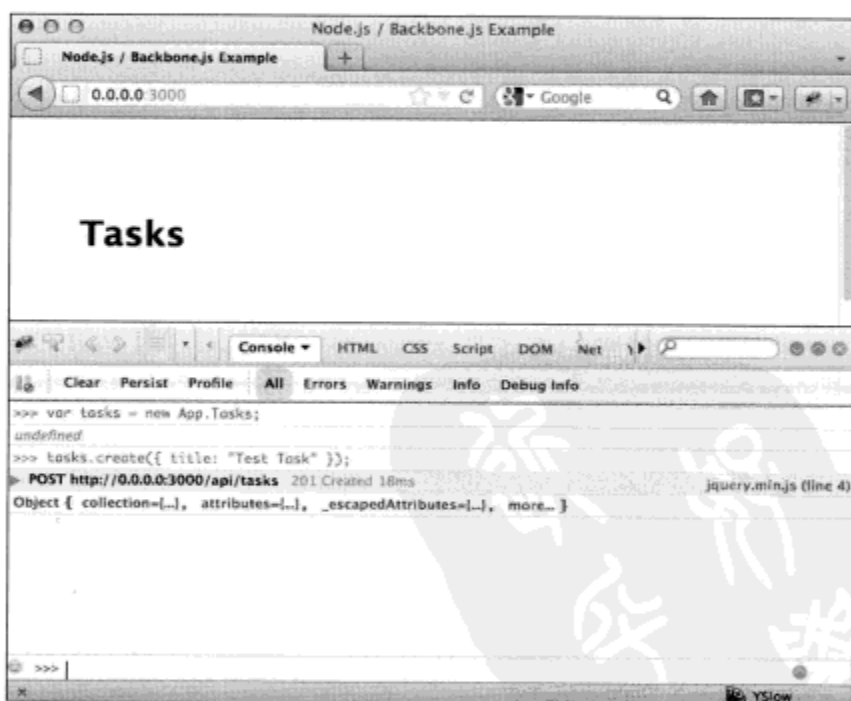
var tasks = new App.Tasks;
tasks.create({ title: "Test Task" });

```

10. 可看到 Backbone.js 向服务器发送数据并且创建了一个新任务 (见图 24.1)。

图 24.1

从 JavaScript 控制台与 Backbone.js 交互



24.3 一个简单的 Backbone.js 视图

具备让用户可以看见应用程序中的数据并与之交互的视图, 是 Backbone.js 的概念。Backbone.js 用与服务器端的框架所用的几乎相同的方法使用模板。这些是将数据动态写入的代码片段。这个概念与我们在 Express 中的 Jade 所看到的相似, 但这一次一切都在浏览器中! 我们可以在 Backbone.js 上使用许多模板语言。对于本示例而言, 使用的是 Underscore.js 模板。为了显示视图, 需要如下内容。

- 一个让用户可以提交任务的表单。

- 一个不排序的列表用于保存任务列表（或集合）。
- 一个列表项模板用于保存每个任务。

通过使用 Underscore.js 模板，可以按如下方式创建这些内容并添加到 index.jade 文件中：

```
script(id='task_form', type='text/template')
| <form action='' id='task-form'>
|   <fieldset>
|     <legend>Add a task</legend>
|     <input type='text' name='title' class='task-title' />
|     <input type='submit' value='Submit' />
|   </fieldset>
| </form>
script(id='task_template', type='text/template')
| <li data-id="<%= task.id %>">
|   <%= task.get('title') %>
| </li>
script(id='tasks_template', type='text/template')
| <%= task_form() %>
| <ul>
|   <% _.each(tasks, function(task) { %>
|     <%= task_template({ task: task }) %>
|   <% }); %>
| </ul>
```

注意“text/template”的类型被用于允许 Backbone.js 找到这些模板并且使用。与服务器的模板非常类似，数据应该被传递到这些模板中，然后动态地显示出来。为了让任何来自 Node.js 服务器的数据都可以传递到 Backbone.js，当页面首次装载时对 MongoDB 做了一个查询。虽然这可以在客户端完成，但建议的方法是在服务器端执行这一操作。app.js 中用于服务 index 页面的 Express 路由现在经过需改，包含了查询并将数据传递到 index.jade 模板：

```
app.get('/', function(req, res){
  db.tasks.find().sort({ $natural: -1 }, function(err, tasks) {
    res.render('index.jade', {
      tasks: JSON.stringify(tasks),
      layout: false
    });
  });
});
```

在 index.jade 文件中，通过在文件的末尾加入一个带有从服务器接收到的数据的脚本标记，Backbone.js 就可以使用这些数据了。

```
script
  var tasks = new App.Tasks;
  tasks.reset(!{tasks});
```

现在可以在 application.js 文件中创建 Backbone.js 视图了：

```
App.TasksView = Backbone.View.extend({
  el: $('#tasks'),
  initialize: function() {
    this.task_form = _.template($('#task_form').html());
    this.tasks_template = _.template($('#tasks_template').html());
    this.task_template = _.template($('#task_template').html());
    this.render();
  }
});
```

```

    },
    render: function() {
      $(this.el).html(this.tasks_template({
        task_form: this.task_form,
        tasks: this.collection.models,
        task_template: this.task_template
      }));
    }
  });
};

```

Backbone.js 视图必须设置一个带有 HTML 结构的元素为目标，所以将一个 id 为 “tasks” 的空白 div 声明为目标。这就意味着 Backbone.js 生成输出然后将其注入到 HTML 元素。

而后在 initialize 函数中初始化视图。当视图实例化的时候任何在 initialize 函数中的东西都会被运行，于是可以将它当成是视图的设立阶段。在 initialize 函数中，引用了刚刚创建的模板来构建视图。注意在 initialize 函数的末尾，调用了 render() 方法。这导致 Backbone.js 将模板渲染成视图并在 HTML 中展示它们。

render 方法在指定的页面元素中写入模板。现在视图已经设立，但为了使用视图，必须对其实例化。Backbone.js 的一个常见模式是，用 init 函数来完成应用程序启动所需做的任何事情。在本例中，所需的一切就是实例化 TasksView 并且让任务传递进来以便显示数据。这可以在 application.js 文件的末尾添加：

```

App.init = function() {
  new App.TasksView({ collection: tasks });
}

```

最后，可以修改 index.jade 文件末尾的 script 标记，让它调用 init 函数并启动 Backbone.js 应用程序：

```

script
  var tasks = new App.Tasks;
  tasks.reset(!{tasks});
  $(function() {
    App.init();
  });

```

Watch Out!

警告：注意浏览器缓存

在读者依次尝试这些示例的时候，可能会发现浏览器缓存了示例之间的客户端 JavaScript。在每个示例之间清理浏览器缓存会有用处。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 hour24/example02 找到。按照下列步骤给 Backbone.js 添加视图。

1. 在上一个示例的基础上，打开 index.jade 文件并在 div#tasks 标记后面添加如下内容：

```

script(id='task_form', type='text/template')
  <form action='' id='task-form'>
  <fieldset>
    <legend>Add a task</legend>
    <input type='text' name='title' class='task-title' />

```

```

|     <input type='submit' value='Submit' />
|   </fieldset>
| </form>
script(id='task_template', type='text/template')
|   <li data-id="<%= task.id %>">
|     <%= task.get('title') %>
|   </li>
script(id='tasks_template', type='text/template')
|   <%= task_form() %>
|   <ul>
|     <% _.each(tasks, function(task){ %>
|       <%= task_template({ task: task }) %>
|     <% }); %>
|   </ul>

```

2. 修改提供 index 页面服务的 Express 路由，让其成为下面这个样子：

```

app.get('/', function(req, res){
  db.tasks.find().sort({ $natural: -1 }, function(err, tasks) {
    res.render('index.jade', {
      tasks: JSON.stringify(tasks),
      layout: false
    });
  });
});

```

3. 给 application.js 文件添加 Backbone 视图和 init 函数：

```

App.TasksView = Backbone.View.extend({
  el: $("#tasks"),
  initialize: function() {
    this.task_form = _.template($('#task_form').html());
    this.tasks_template = _.template($('#tasks_template').html());
    this.task_template = _.template($('#task_template').html());
    this.render();
  },
  render: function() {
    $(this.el).html(this.tasks_template({
      task_form: this.task_form,
      tasks: this.collection.models,
      task_template: this.task_template
    }));
  }
});

App.init = function() {
  new App.TasksView({ collection: tasks });
}

```

4. 为了让 Backbone.js 可以访问来自服务器的数据，在 index.jade 文件末尾添加下列内容：

```

script
  var tasks = new App.Tasks;
  tasks.reset(!{tasks});
  $(function() {
    App.init();
  });

```

5. 启动 Node.js 服务器：

```
node app.js
```

6. 打开带有 JavaScript 控制台的浏览器访问 <http://127.0.0.1:3000>。
7. 可看到在上一个示例中所添加的任何任务都显示在了页面上。

24.4 使用 Backbone.js 创建记录

目前的示例有一个表单，但这需要连接到 Backbone.js 上，这样的话数据才能发送到服务器并写入页面。在 Backbone.js 视图中，可以设立事件和在事件发生时调用的函数。在本例中，我们期望的功能是当表单提交的时候将所提交的任务寄送到服务器并且也写入 HTML。Backbone 对数据正被写入服务器的数据持乐观态度，所以默认情况下，它假设数据已经被成功地送到了 API。触发任务创建的事件是表单的提交，这会调用 `createTask` 函数：

```
events : {
  'submit form' : 'createTask'
},
createTask: function (event) {
  event.preventDefault();
  var taskTitleInput = $('#task-title');
  var taskTitle = taskTitleInput.val();
  tasks.create({ title: taskTitle }, {
    success: function(task){
      $('#tasks ul')
        .prepend("<li>" + taskTitle + "</li>");
      taskTitleInput.val('');
    }
  });
}
```

`createTask` 函数首先捕获提交事件并防止表单被提交。接下来，使用 jQuery 选择器 (selector) 捕获了用户输入的值，然后使用 Tasks 集合上的 `create` 方法在 Backbone.js 中创建一个任务。如果成功，则使用 jQuery 将任务写入 HTML。

TRY IT YOURSELF

如果下载了本书的代码示例，那么这段代码可在 `hour24/example03` 找到。按照下列步骤通过 Backbone.js 创建记录。

1. 在上一个示例的基础上，在 `application.js` 中的 `TasksView` 块添加如下内容：

```
events : {
  'submit form' : 'createTask'
},
createTask: function (event) {
  event.preventDefault();
  var taskTitleInput = $('#task-title');
  var taskTitle = taskTitleInput.val();
  tasks.create({ title: taskTitle }, {
    success: function(task){
      $('#tasks ul')
        .prepend("<li data-id=" + task.id + ">" + taskTitle + "
➡ <button>Done!</button></li>");
      taskTitleInput.val('');
    }
  });
}
```

2. 运行脚本：
`node app.js`
3. 打开带有 JavaScript 控制台的浏览器访问 `http://127.0.0.1:3000`。
4. 在输入框中输入一个任务然后单击 Submit 按钮。
5. 可看到任务写入了页面中。
6. 刷新浏览器窗口。
7. 可看到刚刚创建的任务仍旧在页面上（见图 24.2）。



图 24.2

通过 Backbone.js
视图添加记录

Backbone.js 带有一系列功能，包括数据验证和错误处理，这有些超过了本书介绍 Node.js 的范畴。一个更为完整的示例，包括任务的删除，读者可找到本书示例代码中的 `hour24/example04` 探索并运行。

24.5 小结

在本章，我们学习了如何使用 Node.js 作为 Backbone.js 的 API，Backbone.js 是在浏览器中创建富 JavaScript 应用程序的一个客户端框架。我们了解了整个应用程序栈都可以是 JavaScript 的——Node.js 在服务器，MongoDB 作为数据存储，而 Backbone.js 在客户端。我们介绍了 Backbone.js 并学习了创建模型、集合和视图的方法并看到了使用 Backbone.js 创建一个基础的单页面应用程序的方法。

24.6 问与答

问：我什么时候应该使用单页面应用程序？

答：单页面应用程序意味着一旦页面装载，用户就永远不需要刷新页面。如果能让应用程序可搜索并且用户可以设置书签的话，那么单页面应用程序可能不是一个好的选择。

问: Backbone.js 可以使用 URL 吗?

答: Backbone.js 也支持在应用程序中使用 URL 并使用 HTML5 History API 来允许浏览器窗口中的 URL 被更新。对于更老一些的浏览器而言, Backbone.js 优雅地退却到 URL 的分段版本。

问: 整个应用程序都使用 JavaScript 是否是个好主意?

答: 对于许多基于浏览器的应用程序而言, 整个应用程序使用 JavaScript 是一个很棒的主意。但凡事皆是如此: 它得看应用程序要做的是做什么。如果想给浏览器创建桌面类型的应用程序, 那么在服务器、数据存储和客户端使用 JavaScript 就是个极佳的选择。

问: 我在哪儿可以学到更多关于 Backbone.js 的知识?

答: Backbone.js 文档是着手开始的好地方, 可以在 <http://documentcloud.github.com/backbone/> 找到。在 PeepCode 有许多商业的视频可用, 地址是 <https://peepcode.com/products/backbone-js> 和 <http://backbonescreencasts.com/>。

24.7 测验

本测验包含一些问题和习题, 可帮助读者巩固本章所学的知识。

24.7.1 问题

1. Backbone.js 是否只用于单页面应用程序?
2. Backbone.js 是否是创建富客户端应用程序的唯一框架?
3. 是否需要使用 Node.js 作为 Backbone.js 应用程序的服务器?

24.7.2 答案

1. 不是。虽然 Backbone.js 精于创建单页面应用程序, 但它提供许多让客户端应用程序与服务器交互的功能。而根据应用程序和需求的不同, jQuery 或者纯 JavaScript 可能就足够。
2. 不是。有许多用于在浏览器中创建桌面类型应用程序的客户端框架。比如 Spine.js 和 Ember.js 等。在 <http://codebrief.com/2012/01/the-top-10-javascript-mvc-frameworks-reviewed/> 对客户端框架做了很好的比较。
3. 不是。Backbone.js 独立于服务器应用程序。不过, Node.js 是一个好选择!

24.8 练习

1. 给 Node.js API 加入一些验证, 让任务不能不带标题而提交。
2. 给 Node.js API 编写一些测试, 以确保 Backbone.js 可以按要求与 API 交互。
3. 加入删除任务的能力。解决方案可在书中代码示例的 [hour24/example04](#) 找到。

