

O'REILLY®

TURING

图灵程序设计丛书

第2版

Node与Express 开发

Web Development with Node and Express, Second Edition

巧用JavaScript技术栈，探索Web开发新思路



[美] 伊桑·布朗 著
吴灏栩 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

推荐阅读



《你不知道的JavaScript》系列



《Head First JavaScript程序设计》

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。



Node与Express开发（第2版）

Web Development with Node and Express, Second Edition

[美] 伊桑·布朗 著
吴inston 译

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北京

图书在版编目（CIP）数据

Node与Express开发：第2版 / (美) 伊桑·布朗
(Ethan Brown) 著；吴灏栩译。— 北京：人民邮电出
版社，2021.6
(图灵程序设计丛书)
ISBN 978-7-115-56509-9

I. ①N… II. ①伊… ②吴… III. ①JAVA语言—程序
设计 IV. ①TP312.8

中国版本图书馆CIP数据核字(2021)第086116号

内 容 提 要

本书系统地讲解了使用 Express 开发动态 Web 应用的流程和步骤。作者不仅讲授了开发公共站点及 REST API 的基础知识，还讲解了构建单页、多页及混合 Web 应用的规划方式及最佳实践。为了适应中间件及相关工具在过去几年里的变化，第 2 版更侧重于 Express 作为提供 API 的后端服务器，并新增了单页应用的示例。

本书适合希望使用 JavaScript、Node、Express 构建 Web 应用的开发人员阅读。

-
- ◆ 著 [美] 伊桑·布朗
 - 译 吴灏栩
 - 责任编辑 张海艳
 - 责任印制 周昇亮
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
 - 邮编 100164 电子邮件 315@ptpress.com.cn
 - 网址 <https://www.ptpress.com.cn>
 - 北京 印刷
 - ◆ 开本：800×1000 1/16
 - 印张：18
 - 字数：426千字 2021年6月第1版
 - 印数：1-2500册 2021年6月北京第1次印刷
 - 著作权合同登记号 图字：01-2020-2148号
-

定价：109.80元

读者服务热线：(010)84084456 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东市监广登字 20170147 号

版权声明

Copyright © 2020 Ethan Brown. All rights reserved.

Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2021. Authorized translation of the English edition, 2021 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

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

英文原版由 O'Reilly Media, Inc. 出版，2020。

简体中文版由人民邮电出版社有限公司出版，2021。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc.介绍

O'Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O'Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

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

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

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

——*Linux Journal*

谨以此书献给我的家人：

感谢我的父亲 Tom，他让我爱上了工程学；感谢我的母亲 Ann，她培养了
我对写作的兴趣；感谢我的妹妹 Meris，她一直陪伴着我。

目录

前言	xvii
第 1 章 Express 是什么	1
1.1 JavaScript 的革命	1
1.2 Express 简介	2
1.3 服务器端应用和客户端应用	4
1.4 Express 简史	4
1.5 Node：另一种 Web 服务器	5
1.6 Node 生态系统	6
1.7 开源协议	7
1.8 小结	8
第 2 章 Node 的世界	9
2.1 获取 Node	9
2.2 使用终端	10
2.3 编辑器	11
2.4 npm	11
2.5 用 Node 做一个简单的 Web 服务器	12
2.5.1 Hello world	13
2.5.2 事件驱动编程	14
2.5.3 路由	14
2.5.4 提供静态资源	15
2.6 进入 Express	17

第3章 Express 的方式	18
3.1 脚手架	18
3.2 草地鹨旅游网站	19
3.3 初始工作	19
3.4 视图与布局	22
3.5 静态文件与视图	25
3.6 视图中的动态内容	26
3.7 小结	26
第4章 项目整顿	27
4.1 项目文件与目录结构	27
4.2 最佳实践	28
4.3 版本控制	28
4.4 在本书学习中如何使用 Git	29
4.4.1 亲手录入	29
4.4.2 使用官方版本库	30
4.5 npm 包	31
4.6 项目元数据	32
4.7 Node 模块	32
4.8 小结	34
第5章 质量保证	35
5.1 QA 计划	36
5.2 QA：是否值得	37
5.3 逻辑与表示	38
5.4 测试类型	38
5.5 QA 技术	38
5.6 安装和配置 Jest	39
5.7 单元测试	40
5.7.1 模拟	40
5.7.2 为可测试性而重构应用	40
5.7.3 写第一个测试	41
5.7.4 测试维护	43
5.7.5 代码覆盖率	43
5.8 集成测试	44
5.9 Linting	47

5.10 持续集成.....	50
5.11 小结.....	51
第 6 章 request 和 response 对象.....	52
6.1 URL 的各个组成部分.....	52
6.2 HTTP 请求方法.....	53
6.3 请求头.....	54
6.4 响应头.....	54
6.5 互联网媒体类型.....	55
6.6 请求的 Body.....	55
6.7 request 对象.....	55
6.8 response 对象.....	57
6.9 深入源代码.....	59
6.10 按功能归纳.....	59
6.10.1 渲染内容.....	60
6.10.2 处理表单.....	61
6.10.3 API 服务.....	62
6.11 小结.....	63
第 7 章 视图模板——使用 Handlebars.....	64
7.1 何时使用模板.....	65
7.2 选择模板引擎.....	66
7.3 Pug：另辟蹊径.....	66
7.4 Handlebars 基础.....	67
7.4.1 注释.....	69
7.4.2 代码块.....	69
7.4.3 服务器端模板.....	70
7.4.4 视图和布局.....	71
7.4.5 在 Express 中使用（或不使用）布局.....	73
7.4.6 sections.....	73
7.4.7 partial 模板.....	74
7.4.8 完善模板.....	76
7.5 小结.....	77
第 8 章 表单处理.....	78
8.1 把客户端数据发送到服务器.....	78
8.2 HTML 表单.....	78

8.3 表单的编码.....	79
8.4 处理表单的不同做法.....	80
8.5 使用 Express 处理表单	81
8.6 使用 <code>fetch</code> 发送表单数据	83
8.7 文件上传.....	85
8.8 提升文件上传的 UI.....	88
8.9 小结.....	88
第 9 章 Cookie 和 Session	89
9.1 提取敏感信息.....	91
9.2 Express 中的 Cookie.....	91
9.3 查看 Cookie	93
9.4 Session.....	93
9.4.1 内存存储	94
9.4.2 使用 Session.....	95
9.5 使用 Session 实现 flash 消息.....	95
9.6 Session 的用途.....	97
9.7 小结.....	97
第 10 章 中间件.....	98
10.1 基本原理.....	99
10.2 中间件示例.....	99
10.3 常用中间件.....	102
10.4 第三方中间件.....	104
10.5 小结.....	104
第 11 章 发送邮件	105
11.1 SMTP、MSA 和 MTA	105
11.2 接收邮件	106
11.3 邮件头	106
11.4 邮件格式	106
11.5 HTML 邮件.....	107
11.6 Nodemailer	107
11.6.1 发送邮件.....	108
11.6.2 发送给多个收件人.....	109

11.7 群发邮件更好的选择	110
11.8 发送 HTML 邮件	110
11.8.1 HTML 邮件中的图片	111
11.8.2 使用视图来发送 HTML 邮件	111
11.8.3 封装邮件功能	113
11.9 小结	114
第 12 章 考虑生产环境中的问题	115
12.1 运行环境	115
12.2 特定环境的配置	116
12.3 运行 Node 进程	117
12.4 网站的扩展	118
12.4.1 使用应用集群实现水平扩展	119
12.4.2 处理未捕获的异常	121
12.4.3 使用多台服务器完成水平扩展	123
12.5 监控网站的运行	123
12.6 压力测试	124
12.7 小结	125
第 13 章 持久化	126
13.1 文件系统持久化	126
13.2 云持久化	128
13.3 数据库持久化	129
13.3.1 关于性能的提醒	129
13.3.2 数据库层抽象	130
13.3.3 设置 MongoDB	131
13.3.4 Mongoose	132
13.3.5 使用 Mongoose 连接数据库	132
13.3.6 创建模式和模型	133
13.3.7 使用种子数据初始化	134
13.3.8 获取数据	136
13.3.9 更新数据	138
13.3.10 PostgreSQL	139
13.3.11 新增数据	145
13.4 使用数据库存储 Session	146
13.5 小结	148

第 14 章 路由	149
14.1 路由与 SEO	151
14.2 子域名	151
14.3 路由处理函数也是中间件	152
14.4 路由路径和正则表达式	154
14.5 路由参数	154
14.6 组织路由	155
14.7 在模块中声明路由	156
14.8 合乎逻辑地分组路由	157
14.9 自动化渲染视图	158
14.10 小结	159
第 15 章 REST API 和 JSON	160
15.1 JSON 和 XML	161
15.2 我们的 API	161
15.3 API 错误报告	162
15.4 跨域资源共享	163
15.5 测试	164
15.6 使用 Express 提供 API	166
15.7 小结	167
第 16 章 单页应用	168
16.1 Web 应用开发简史	168
16.2 SPA 技术选择	171
16.3 创建 React 应用	172
16.4 React 基本概念	172
16.4.1 主页	174
16.4.2 路由	176
16.4.3 度假产品页——可视化设计	178
16.4.4 度假产品页——跟服务器端集成	179
16.4.5 向服务器发送信息	181
16.4.6 状态管理	184
16.4.7 部署选择	185
16.5 小结	185

第 17 章 静态内容	187
17.1 性能上的考量	188
17.2 内容分发网络 (CDN)	189
17.3 为 CDN 而设计	189
17.3.1 服务器端渲染的网站	190
17.3.2 单页应用	190
17.4 缓存静态资源	191
17.5 变更静态内容	192
17.6 小结	193
第 18 章 安全	194
18.1 HTTPS	194
18.1.1 生成自己的证书	195
18.1.2 使用免费的证书中心	196
18.1.3 购买证书	196
18.1.4 为 Express 应用启用 HTTPS	198
18.1.5 有关端口的说明	199
18.1.6 HTTPS 与代理	200
18.2 跨站请求伪造	201
18.3 认证	202
18.3.1 认证与授权	202
18.3.2 使用密码认证的问题	203
18.3.3 第三方认证	203
18.3.4 在数据库里存储用户信息	204
18.3.5 认证与注册及用户体验	205
18.3.6 Passport	205
18.3.7 基于角色的授权	214
18.3.8 增加认证提供者	215
18.4 小结	216
第 19 章 集成第三方 API	217
19.1 社交媒体	217
19.1.1 社交媒体插件与网站性能	217
19.1.2 搜索推文	218
19.1.3 展现推文	221

19.2 地理编码.....	223
19.2.1 使用谷歌生成地理编码.....	223
19.2.2 为你的数据做地理编码.....	225
19.2.3 显示地图.....	227
19.3 天气数据.....	228
19.4 小结.....	230
第 20 章 调试.....	231
20.1 调试原则第一条.....	231
20.2 利用 REPL 和控制台.....	232
20.3 使用 Node 的内建调试器.....	233
20.4 Node 调试客户端.....	233
20.5 调试异步函数.....	237
20.6 调试 Express.....	237
20.7 小结.....	239
第 21 章 上线.....	240
21.1 域名注册与托管.....	240
21.1.1 域名系统.....	241
21.1.2 安全.....	241
21.1.3 顶级域名.....	242
21.1.4 子域名.....	243
21.1.5 域名服务器.....	243
21.1.6 托管服务.....	245
21.1.7 部署.....	247
21.2 小结.....	250
第 22 章 维护.....	251
22.1 维护的原则.....	251
22.1.1 长远规划.....	251
22.1.2 使用源代码控制.....	253
22.1.3 使用问题跟踪系统.....	253
22.1.4 保持良好的“卫生习惯”.....	253
22.1.5 不要拖延.....	253
22.1.6 例行 QA 核查.....	254
22.1.7 监控分析.....	254
22.1.8 优化性能.....	255

22.1.9 优先跟踪潜在客户	255
22.1.10 避免“不可见”的故障	256
22.2 代码重用与重构	257
22.2.1 私有 npm 仓库	257
22.2.2 中间件	257
22.3 小结	259
第 23 章 更多资源	260
23.1 在线文档	260
23.2 期刊	261
23.3 Stack Overflow	261
23.4 对 Express 做贡献	263
23.5 小结	265
关于作者	266
关于封面	266

前言

目标读者

本书是为那些想用 JavaScript、Node 和 Express 创建 Web 应用（传统网站，采用了 React、Angular 或 Vue 的单页应用，REST API，或是介于以上任意二者之间的应用）的程序员而写的。Node 开发中一个令人兴奋的方面是已经有一大批新程序员投入其中。世界各地的程序员们深感于 JavaScript 既好理解又十分灵活，纷纷加入了自学的行列。计算机历史上从未有过这样的时候，人人都可以如此方便地编程。网上学习编程的资源以及在你受挫时为你提供帮助的资源，其数量和质量都让人难以置信，并让人深受鼓舞。因此，对于那些编程新人们（也许是自学者），我欢迎你们。

当然，还有像我这样已经编程多年的人。就像我那个年代的很多程序员一样，我从汇编和 BASIC 语言开始，经历过 Pascal、C++、Perl、Java、PHP、Ruby、C、C# 以及 JavaScript。大学期间，我接触过更多的小众语言，比如 ML、LISP 和 PROLOG。其中的很多语言，虽然让我感觉十分亲切，但在发展前景上，没有一个能跟 JavaScript 相比。因此，对于像我一样已经拥有丰富经验，对具体技术也许有了更超然的态度的程序员，本书也是为你们而写的。

读者无须 Node 开发经验，但应该有 JavaScript 开发经验。如果你初学编程，我向你推荐 Codecademy 网站；如果你是中级或有经验的程序员，我向你推荐《JavaScript 学习指南（第 3 版）》，这是我自己的书。本书中的代码在任何能够运行 Node 的系统（Windows、macOS、Linux，等等）上都可以运行。这些代码适合命令行（终端）用户，因此你对所在系统的终端应该有一定的了解。

最重要的是，本书是为那些兴奋的程序员而写的：那些为互联网的未来而兴奋，并想要投身其中的程序员；那些为学习新概念、新技术以及 Web 开发的新思路而兴奋的程序员。如果你尚未感到兴奋，亲爱的读者，希望你读完本书后会兴奋起来……

对第2版的说明

写作本书第1版是一件乐事。读者们热情的回应和切实可行的建议，我今天回想起来还十分愉快。第1版出版时，正当Express 4.0的beta版刚刚发布。虽然现在Express仍旧是4.x版，但与Express配合的中间件和工具已经发生了极大的变化。而且，JavaScript本身也在演化，甚至Web应用的设计方式也发生了转变[从纯服务器端渲染转向单页应用(SPA)]。虽然第1版中的很多原则仍然正确而且有用，但具体的技术和工具已经完全不同了。本书是该出新版了。由于SPA大行其道，Express更多是作为提供API和静态资源的服务器。第2版更强调Express的这种用法，而且会包含一个SPA的示例。

本书结构

第1章和第2章将引领你初识Node和Express，以及贯穿本书会用到的一些工具。在第3章和第4章中，你将开始使用Express构建一个示例网站的框架，后面的章节都用这个网站作为示例并会持续更新。

第5章将讨论测试和质量保证(QA)。第6章将讨论Node的一些更重要的构造，以及Express如何扩展和使用它们。第7章将讨论模板化(使用Handlebars)，这为使用Express构建实用的网站打下了基础。第8章和第9章将讨论Cookie、Session以及表单处理函数，了解了这些，你就可以使用Express构建具有基本功能的网站了。

第10章将介绍中间件，这是Express的一个核心概念。第11章将说明如何使用中间件从服务器端发送电子邮件，并讨论电子邮件固有的安全和内容布局问题。

第12章将讨论生产环境需要关注的问题。本书读到这里，即使你还没掌握构建一个可以投产的网站所需要的所有知识，提前考虑生产问题也可以免去将来若干头痛之事。

第13章将探讨持久化，主要关注MongoDB(卓越的文档数据库之一)和PostgreSQL(一个流行的开源关系型数据库管理系统)。

第14章将详细讨论Express中的路由(如何把URL映射到内容)。第15章将讨论如何在Express中写API。

第16章将运用我们已经学到的Express知识，把当前的示例重构为SPA，Express只用作后端服务器，提供在第15章中创建的API。

第17章将详细讨论静态内容服务，重点关注性能的最大化。

第18章将讨论安全：如何把认证和授权整合进应用中(主要关注一个第三方认证的使用)以及如何通过HTTPS运行网站。

第 19 章将说明如何集成第三方服务，使用的例子包括 Twitter、谷歌地图和全美天气服务。

第 20 章和第 21 章将帮助你为网站上线做好准备，内容包括调试和上线流程。调试可以让你在网站上线前发现所有缺陷。第 22 章将讨论下一个重要的（并且常常被忽视的）阶段：生产维护。

本书终结于第 23 章，如果你想更深入地学习 Node 和 Express，这一章指出了更多的资源以及去哪里寻求帮助。

示例网站

从第 3 章开始，我们使用了一个贯穿全书并持续更新的示例：草地鹨（Meadowlark）旅游网站。写作本书第 1 版时，正值我刚从葡萄牙的里斯本旅游回来。我脑子里想着旅游的事，所以选择的示例网站就是为一个假想的旅游公司而创建的。这个假想的公司位于我的家乡俄勒冈州（西草地鹨是俄勒冈州的州鸟）。草地鹨旅游网站让游客能够联系上当地的业余导游，而且网站跟提供自行车 / 踏板车租赁和当地旅游服务的公司合作，强调生态旅游。

像任何用于教学的示例一样，草地鹨旅游网站是假想出来的，但它涵盖了很多真实网站正面临的挑战：第三方组件集成、地理定位、电子商务、性能以及安全。

由于本书更关注后端设施，所以这个示例网站最终不会很完整，它只是作为真实网站的一个范例，为讲解提供一定的深度和背景。大概你也会开发自己的网站，到时可以用这个示例网站作为模板。

排版约定

本书使用以下排版约定。

黑体

表示新术语或重点强调的内容。

等宽字体（`constant width`）

表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键词等。

加粗等宽字体（`constant width bold`）

表示应该由用户输入的命令或其他文本。

斜体等宽字体（`constant width italic`）

表示应该由用户输入的值或根据上下文确定的值替换的文本。



此图标表示提示或建议。



此图标表示一般性注记。



此图标表示警告或警示。

使用示例代码

本书的补充材料（代码、练习等）可以在 GitHub 网站的 [EthanRBrown/web-development-with-node-and-express-2e](#) 页面下载。¹

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如，“*Web Development with Node and Express, Second Edition* by Ethan Brown (O'Reilly). Copyright 2019 Ethan Brown, 978-1-492-05351-4”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly 在线学习平台 (O'Reilly Online Learning)

O'REILLY® 40 多年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台让你能够按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的

注 1：读者也可到图灵社区本书主页 ituring.cn/book/2779 “随书下载” 处下载示例代码。——编者注

大量文本资源和视频资源。有关的更多信息，请访问 <https://oreilly.com>。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

请访问 https://oreil.ly/web_dev_node_express_2e，到本书页面查看相关勘误²、示例代码以及任何附加信息。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：<http://www.oreilly.com>

我们在 Facebook 的地址如下：<http://facebook.com/oreilly>

请关注我们的 Twitter 动态：<http://twitter.com/oreillymedia>

我们的 YouTube 视频地址如下：<http://www.youtube.com/oreillymedia>

致谢

很多人为本书的出版贡献了力量。要是没有这些人在生活中影响我、让我能够成为今天的我，我是无法出版这本书的。

我想先从 Pop Art 开始，感谢 Pop Art 里的每一个人：不仅是因为在 Pop Art 的时光重新燃起了我作为工程师的激情，而且我从这里每个人的身上都学到了很多，如果没有他们的支持，就不会有这本书。感谢 Steve Rosenbaum 创造了这样一个工作场所，让人灵感不断。感谢 Del Olds 让我加入 Pop Art，并让我感到自己很受欢迎，他是一位受人尊敬的领导者。感谢 Paul Inman 的坚定支持以及在工程上鼓舞人心的态度。感谢 Tony Alferez 的热情支持，以及帮助我挤出时间来写作，同时不影响工作。最后，感谢我有幸一起共事过的所有出色的工程师，你们让我时刻保持警醒：John Skelton、Dylan Hallstrom、Greg Yung、Quinn Michaels、CJ Stritzel、Colwyn Fritze-Moor、Diana Holland、Sam Wilskey、Cory Buckley 和 Damion Moyer。

注 2：本书中文版勘误请到 ituring.cn/book/2779 查看和提交。——编者注

非常感谢我目前所在公司（Value Management Strategies, VMS）的团队成员。我从 Robert Stewart 和 Greg Brink 身上学到了很多关于软件业务方面的知识，从 Ashley Carson 身上学到了很多与团队沟通、团结和效率有关的东西（Scratch Chromatic，感谢你的坚定支持）。Terry Hays、Cheryl Kramer 和 Eric Trimble，感谢你们的努力工作和支持！Damon Yeutter、Tyler Brenton 和 Brad Wells，感谢你们在需求分析和项目管理上至关重要的工作。最重要的，感谢 VMS 公司中和我一起工作的富有才华而又敬业的开发者们：Adam Smith、Shane Ryan、Jeremy Loss、Dan Mace、Michael Meow、Julianne Soifer、Matt Nakatani 和 Jake Feldmann。

感谢“School of Rock”乐队所有的伙伴们！那是多么疯狂的一段经历，一种多么快乐而有创造性的发泄方式。特别感谢 Josh Thoma、Amanda Sloane、Dave Coniglio、Dan Lee、Derek Blackstone 和 Cory West，你们的热情感染了我，你们跟我分享音乐知识，是我的导师。感谢你们所有人，给了我成为一个摇滚明星的机会。

Zach Mason，感谢你给我带来了灵感。本书不能跟你的 *The Lost Books of the Odyssey* 相比，但毕竟是我写的。要是没有你这个榜样，我不知道有没有勇气写作本书。

Elizabeth 和 Ezra，感谢你们给我的礼物，我会永远爱你们。

我对父母的感激之情无以言表。他们给了我最好的、充满爱的教育。我看到他们也将同样的爱给予了我妹妹。

非常感谢 Simon St. Laurent 给了我写作本书的机会。感谢 Angela Rufino（第 2 版）和 Brian Anderson（第 1 版）稳步而又鼓舞人心的编辑工作。感谢 O'Reilly 每个人的奉献和热情。感谢 Alejandra Olvera-Novack、Chetan Karande、Brian Sletten、Tamas Piros、Jennifer Pierce、Mike Wilson、Ray Villalobos 和 Eric Elliot 全面又有建设性的技术评论。

对于我冒昧送上的书稿，Katy Roberts 和 Hanna Nelson 提供了宝贵的反馈和建议，使得本书的面世成为可能。非常感谢你们！感谢 Chris Cowell-Shah 对第 5 章提出的精彩反馈意见。

最后，感谢我亲爱的朋友们：Byron Clayton、Mark Booth、Katy Roberts 和 Kimberly Christensen。没有你们，我肯定会精神失常的。我爱你们！

电子书

扫描如下二维码，即可购买本书中文版电子版。



第1章

Express是什么

1.1 JavaScript的革命

在进入本书的主题之前，非常有必要先提供一点儿历史和背景信息。这意味着我们要谈一谈 JavaScript 和 Node。JavaScript 的时代已经到来了。当初 JavaScript 作为客户端脚本语言，并不受人待见，可到了现在，它不仅完全统治了客户端，而且早已被用作服务器端语言，这一切多亏了 Node。

全 JavaScript 技术栈的前景是显而易见的：再也没有频繁的语境切换了！你再也不需要从 JavaScript 切换到 PHP、C#、Ruby 或 Python（或任何其他服务器端语言）了。而且，全 JavaScript 技术栈提升了前端工程师们的能力，使他们可以转到服务器端编程。这并非说服务器端编程只是语言的事，关于服务器端编程还是有很多东西要学的。但是选择使用 JavaScript，至少语言不会是一个障碍了。

本书适合所有看到了 JavaScript 技术栈广阔前景的人。或许你是个前端工程师，想继续拓展到后端开发。或许你是个像我一样有经验的后端开发者，在目前诸多根深蒂固的服务器端语言之外，把 JavaScript 看作另一个可行的选择。

如果你做软件工程师的年头和我一样长，就会发现很多语言、框架和 API 曾风靡一时。有些的确发展起来了，而有的最终销声匿迹。你或许自豪于自己快速学习新语言、新系统的能力。每一种新语言你都觉得有些熟悉：某块知识，你在大学学习某种语言时就学过了；另一块知识，你在几年前的工作中就学过了。这种感觉当然不错，但也会让人疲惫。有时你不过是想把任务完成，并不想学一种全新的技术，也不想把搁置了几个月甚至几年的技

能重新捡起来。

起初，JavaScript 看起来没有什么胜算。我也是这么看的，毋庸置疑。如果在 2007 年你告诉我说，你不仅会考虑把 JavaScript 作为首选语言，而且会写一本关于 JavaScript 的书，我会说“你疯了”。我也和大家一样对 JavaScript 有过偏见，认为它只是一种“玩具”语言，是只有外行和半吊子才会来捣鼓的东西。平心而论，JavaScript 为编程爱好者降低了门槛，加上那时有问题的 JavaScript 代码又太多了，这些都不利于这门语言的名声。用一句流行的话来说：“可恨的是玩家，不是游戏。”

遗憾的是人们受到这种偏见的影响，没能发现 JavaScript 这门语言是多么强大、灵活和优雅。我们现在知道，JavaScript 早在 1996 年就出现了（尽管它很多吸引人的特性是 2005 年加上的），然而，很多人直到现在才开始认真地看待它。

在你拿起本书时，你可能没有这种偏见：或许像我一样，偏见成了过去式；又或许是从来就没有过偏见。不管怎样，你是幸运的，我很乐意为你介绍 Express。造就 Express 的，正是 JavaScript 这门给人带来惊喜的语言。

到 2009 年，距人们开始认识到 JavaScript 作为浏览器脚本语言的威力和表达力已经有几年了，Ryan Dahl 看到了 JavaScript 作为服务器端语言的潜力，于是 Node.js 诞生了。此时正是互联网技术蓬勃发展的时代。Ruby 以及 Ruby on Rails 将学院派计算机科学的一些优秀思想，同它们自己的一些原创思想融合起来，向世界展示了一种开发网站和 Web 应用的更快的方式。微软公司为了证明自己在互联网时代的不可或缺而不懈努力，一方面大量地从学术殿堂汲取营养，另一方面从 Ruby 和 JavaScript 的成功及 Java 的失误中学到了不少，于是在 .NET 上取得了卓越的成就。

今天，多亏了像 Babel 这样的转译技术，Web 开发者才得以放心地使用 JavaScript 语言的最新特性，不用担心阻隔了使用老浏览器的用户。在管理 Web 应用的依赖和保证性能方面，Webpack 成了通用的解决方案。像 React、Angular 和 Vue 这样的框架，正在改变人们进行 Web 开发的方式，使得声明式 DOM 操作库（例如 jQuery）慢慢地变得不再重要，成了明日黄花。

现在是参与互联网技术发展的激动人心的时刻。让人惊奇的新想法（或重新恢复活力的旧想法）随处可见。很多年以来，创新精神和兴奋感从没有像今天这样强烈。

1.2 Express简介

Express 的官网上把 Express 描述为“最小而又灵活的 Node.js Web 应用框架，为 Web 和移动应用提供了一个健壮的特性集”。可这到底是什么意思？我们把这句话分解一下。

最小

这是 Express 最吸引人的一个方面。很多时候，框架开发者忘记了通常“少即是多”。Express 的哲学就是，在你的大脑与 Web 服务器之间提供最小的一层。这并不意味着它就不够健壮，或是没有太多有用的特性。相反，这意味着它在为你提供有用东西的同时，对你的限制更少，能够让你的想法得到更充分的表达。Express 给你提供一个最小的框架，你可以根据需要加入 Express 的各部分功能，替换任何不符合你需要的功能。这给框架带来了可操作的空间。太多的框架想要包办一切，你甚至还没开始写一行代码，你的项目就已经变得臃肿、神秘而又复杂。项目的第一项任务，往往就是砍掉不需要的功能或替换不符合需求的功能，结果时间都浪费了。Express 反其道而行之，允许你在需要的时候再加入东西。

灵活

最小的结果就是，Express 所做的事情是非常简单的：从客户端（可以是浏览器、移动设备、另一个服务器、桌面应用……只要能理解 HTTP）接收 HTTP 请求，然后返回 HTTP 响应。这个基本模式几乎可以描述任何到互联网的连接，使得 Express 的应用极其灵活。

Web 应用框架

或许更精确的说法是“Web 应用框架的服务器端部分”。时至今日，当你想到“Web 应用框架”这个概念的时候，想到的多半是像 React、Angular 或 Vue 这样的单页应用框架。不过，除少数独立应用外，大多数 Web 应用需要分享数据和集成其他服务。这些工作一般需要一套 Web API 来完成，这套 API 就可以看作 Web 应用框架的服务器端组件。要注意，要构建整个只有服务器端渲染的应用还是可以的（有时也需要如此），这种情况下，Express 就是完整的 Web 应用框架。

关于 Express 的特性，除了官方描述中明确提到的，我还想再加两点。

高性能

随着 Express 成为 Node.js 开发的首选 Web 框架，它也吸引了很多大公司的注意，这些大公司运行着高性能、高流量的网站。这给 Express 团队带来了压力，促使他们特别关注性能。现在 Express 可以为高流量网站提供一流的性能。

中立

JavaScript 生态系统的一个特征就是它的规模和多样性。尽管 Express 常常位于 Node.js Web 开发的中心，还是有数百个（如果在读者阅读时还没有达到数千个的话）来自社区的包可以用到 Express 应用里。Express 团队认识到了这个生态系统的多样性，于是提供了一个极其灵活的中间件体系，使你可以很容易地选择组件来创建自己的应用。你可以看到，在 Express 本身的开发过程中，它摒弃了内建组件的做法，选择了可配置中间件的机制。

前面提到 Express 是 Web 应用框架的“服务器端部分”，因此或许我们应该谈谈服务器端应用和客户端应用之间的关系。

1.3 服务器端应用和客户端应用

服务器端应用就是应用中的页面都在服务器上生成（如 HTML、CSS、图片和其他多媒体资源，以及 JavaScript），然后发送给客户端。而客户端应用的大部分用户界面是从一个资源包渲染的，这个资源包只需要在最开始的时候接收一次。也就是说，一旦浏览器接收到初始的 HTML（通常也非常小），它就使用 JavaScript 来动态地修改 DOM，不需要依靠服务器来显示新页面（虽然原始数据通常还是来自服务器）。

在 1999 年之前，服务器端应用是标准。事实上，Web 应用这个术语是在 1999 年正式引入的。我认为大约 1999 年到 2012 年这段时间是 Web 2.0 时代，在这个时期，最终成为客户端应用的技术正在开发之中。到 2012 年，随着智能手机的稳固发展，通过网络发送尽可能少的信息成为一种普遍做法，这种做法更青睐客户端应用。

服务器端应用常常被称为服务器端渲染（SSR）应用，客户端应用常常被称为单页应用（SPA）。客户端应用完全在诸如 React、Angular 和 Vue 等框架中实现。我总是觉得“单页”的叫法有点儿不妥，因为在用户看来显然有很多个页面。唯一的区别是，页面是服务器发过来的还是在客户端动态渲染的。

在实际中，服务器端应用和客户端应用之间有很多模糊的界线。很多客户端应用有 2 个或 3 个 HTML 资源包可以发给客户端（例如公开可见的界面和登录可见的界面，或普通界面和管理界面）。而且，SPA 常常跟 SSR 结合以便提升第一页加载的性能，这样做也有助于搜索引擎优化（SEO）。

一般来说，如果服务器只发送少量的 HTML 文件（一般 1 至 3 个），而此时用户可以获得基于动态 DOM 操作的丰富的多视图体验，我们就认为这是客户端渲染。各个视图的数据（通常是 JSON 格式）和多媒体资源通常还是要从网络传过来。

当然，对 Express 来说，它并不怎么关心你是在开发一个服务器端应用还是客户端应用，两种角色它都能完成得很好。你要提供 1 个 HTML 资源包还是 100 个，对它来说都一样。

目前 SPA 毫无疑问已经成了统治性的 Web 应用架构。尽管如此，本书还是从符合服务器端应用的示例开始。服务器端应用还是有意义的，而且，提供 1 个 HTML 资源包还是很多 HTML 资源包，在概念上的差异是很小的。第 16 章会提供一个 SPA 的示例。

1.4 Express 简史

Express 的创造者 TJ Holowaychuk 把 Express 描述为一个受 Sinatra 启发的 Web 框架。Sinatra 是

一个基于 Ruby 的 Web 框架。Express 借鉴一个用 Ruby 写的框架并不奇怪：在 Web 开发方面，Ruby 孵化出了大量卓越的新思路、新方法，使 Web 开发变得更快捷、更高效和更好维护。

就像深受 Sinatra 启发一样，Express 还跟 Connect 深度交织在一起，这是 Node 应用的一个插件库。Connect 最初使用中间件这个术语来描述各种能在不同程度上处理 Web 请求的可插拔的 Node 模块。2014 年，4.0 版本的 Node 移除了对 Connect 的依赖，不过仍然将其“中间件”概念归功于 Connect。



从版本 2.x 到 3.0，Express 经历了大规模的重写；从 3.x 到 4.0，亦如此。本书关注的是版本 4.0。

1.5 Node：另一种Web服务器

某种程度上，Node 跟其他几个流行的 Web 服务器，如微软的 IIS 或 Apache，有很多共同之处。不过，更有意思的是它们究竟有哪些不同。让我们从这里开始讲起。

对于 Web 服务器，Node 的做法也是最小化，这一点跟 Express 一样。不像 IIS 或 Apache 那样，需要花上数年时间去掌握，Node 是很容易搭建和配置的。这并不是说在生产环境中为达到性能最大化去调优 Node 服务器是一件很简单的事情，而是说比起那些服务器，Node 的配置选项要简单多了。

Node 跟传统 Web 服务器的另一个显著差异就是，Node 是单线程的。乍一看去，这似乎是一步倒退。而事实上，这是明智之举。单线程极大地简化了 Web 应用的编程，而当需要达到多线程应用的性能时，你只需简单地多开几个 Node 实例，就能获得多线程的好处。敏锐的读者可能会想，这听起来像是糊弄人呢。毕竟，通过服务器并行（跟应用内的并行相对）实现的多线程，只是把复杂性转移了，并没有消除它呀。或许如此，但根据我的经验，这种做法是把复杂性转移到了它本来就该在的地方。况且，随着云计算越来越受欢迎，越来越多的人把服务器资源看作普通商品，这种做法就越发显得合理了。IIS 和 Apache 的确很强大，它们从设计上就是要“榨干”今天强大硬件的最后“一滴”性能。可是有代价的，要达到那样的性能，就必须有相当专业的人员来搭建和调优。

从编程来说，Node 应用更像 PHP 或 Ruby 应用，而不是 .NET 或 Java 应用。虽然 Node 所使用的 JavaScript 引擎（谷歌的 V8）是把 JavaScript 编译成了原生机器代码（像 C 或 C++ 那样），但这是透明地完成的¹，所以从用户的角度来看，它就像纯粹解释型语言一样。由于没有单独的编译步骤，维护和部署更省事了：你只需更新 JavaScript 文件，所做的变更会自动生效。

注 1：常被称作即时（JIT）编译。

Node 应用还有一个很吸引人的好处：Node 是平台独立的。它不是第一项或唯一一项平台独立的服务器技术，然而，平台独立远不只是二进制包的问题。例如，因为有了 Mono，你可以在 Linux 服务器上运行 .NET 应用，但是由于文档的缺失和系统不兼容，这会是一项痛苦的工作。同样，你可以在 Windows 服务器上运行 PHP 应用，但是通常来说搭建它没有在 Linux 服务器上那么容易。而在各大主流操作系统（Windows、macOS 和 Linux）上搭建 Node 都轻而易举，不同操作系统协作起来也很容易。在各个网站设计小组中，混合使用 PC 和 Mac 是寻常事。某些开发平台，如 .NET，就给前端开发和设计人员带来了不少的挑战（因为他们常常使用 Mac），严重影响了他们的协作和工作效率。只要几分钟（甚至几秒钟）就可以在任何操作系统上运行起来一个能正常工作的服务器，这个美梦已经成真了。

1.6 Node 生态系统

自然，Node 位于这个技术栈的核心位置。正是 Node，使得 JavaScript 可以运行于服务器之上，脱离浏览器而存在，从而允许使用用 JavaScript 编写的框架（如 Express）。另一个重要的组件就是数据库，这将在第 13 章深入讨论。除非是最简单的 Web 应用，否则都需要一个数据库。有几种数据库在 Node 中比在其他语言中还要好用。

所有主要的关系型数据库（MySQL、MariaDB、PostgreSQL、Oracle、SQL Server）都有 Node 的接口，这不奇怪。这些数据库经过多年发展已经非常强大，忽略它们是愚蠢的。尽管如此，Node 时代的到来，让一种新数据库存储方式重新焕发了活力：所谓的 NoSQL 数据库。把某个东西定义为不是什么，无助于人们理解，所以我们补充一下，这些 NoSQL 数据库或许叫作“文档数据库”或“键值对数据库”更为合适。对于数据存储，它们提供了一种概念上更简单的方法。这类 NoSQL 数据库有很多，而 MongoDB 是它们的领跑者，所以我们将 MongoDB 作为本书的 NoSQL 数据库。

构建一个实用的网站需要依赖多个技术板块，因此对于网站所基于的技术栈，人们想出了各种缩写词来描绘它们。例如，Linux、Apache、MySQL 和 PHP 的组合就被叫作 LAMP 技术栈。MongoDB 的一位工程师 Valeri Karpov 造了一个缩写词 MEAN：Mongo、Express、Angular 和 Node。它很易记，这是肯定的，但也有局限：对于数据库和应用框架来说，有太多的选择了，“MEAN”并不能体现这个生态的多样性（何况它没有把渲染引擎包含进来，这是我认为很重要的一个组件）。

想出一个具有包容性的缩写词是一项有意思的练习。当然，其中不可或缺的一个组件就是 Node。尽管还有其他的服务器端 JavaScript 容器，但 Node 自问世以来便统治了一切。Express 同样不是唯一可用的 Web 应用框架，但它的统治力也接近 Node 了。对 Web 应用开发来说，还有两个常常至关重要的组件，就是数据库服务器和渲染引擎（或者是模板引擎，如 Handlebars；或者是 SPA 框架，如 React）。对于这两者，没有那么多明显的领跑

者，我认为进行限制是有害无益的。

把所有这些技术捆绑到一起的，就是 JavaScript。所以为了更具包容性，我会把它叫作 **JavaScript 技术栈**。就本书来说，它指的是 Node、Express 和 MongoDB（第 13 章还有关系型数据库的示例）。

1.7 开源协议

开发 Node 应用的时候，你可能会发现，自己必须要比以前更加留心开源协议（我当然也是）。Node 生态系统的一个美妙之处就是有大量的包可用。可是，每个包都有自己的授权方式，更麻烦的是，每个包都可能依赖一些其他的包，这意味着要理解应用各个部分的授权方式会十分困难。

好在有一些好消息。MIT 协议是 Node 包用得最多的协议之一。这是一个非常宽松的协议，你几乎可以做任何事，包括把这个包用到闭源软件里。不过你不能设想你用的每个包都采用 MIT 协议。



在 npm 中有好几个包，可以尝试找出你的项目中每个依赖的授权方式。请在 npm 中搜索 `nlf` 或 `license-report`。

除了 MIT 这个最常见的协议，你可能还会遇到以下开源协议。

GNU 通用公共许可 (GPL)

GPL 是一个很受欢迎的开源协议，为了保持软件的开源，它经过了精心设计。这意味着如果你在项目中使用了 GPL 授权的代码，你的项目必须也要以 GPL 来授权。自然，你的项目就不能是闭源的。

Apache 2.0

像 MIT 一样，这个协议允许你的项目采用一种不同的协议，包括闭源的协议。不过对于那些采用 Apache 2.0 许可的组件，你必须在项目协议中把它们的授权声明都包含进来。

BSD 许可

类似于 Apache 许可，这个协议允许你的项目采用任何你想用的协议，只要你把采用 BSD 许可的组件的授权声明都包含进来。



有时候软件是双重许可的（按两种不同的协议来授权）。这样做的一个常见原因是，让软件既可以在 GPL 项目中，也可以在采用更宽松许可的项目中（如果一个组件要用到 GPL 软件中，那么这个组件必须是 GPL 授权的）。在我自己的项目中，我经常采用这种授权模式：GPL 和 MIT 的双重许可。

最后，如果你在写自己的包，你应该做一个社区的好公民，选择合适的协议，并进行适当的说明。对开发者来说，在使用别人的包的时候，如果需要深挖代码以便确定所采用的协议，甚至找了半天却发现根本没有采用许可协议，就太让人沮丧了。

1.8 小结

对于 Express 是什么以及它在更大的 Node 和 JavaScript 生态系统中处于什么位置，希望本章让你有了更深入的理解。同时，希望本章也阐明了服务器端 Web 应用和客户端 Web 应用的关系。

对于 Express 确切来说是什么，如果你仍旧感到迷惑，也不用担心：有时候直接上手使用一个东西更有助于理解它。本书会帮助你使用 Express 来构建 Web 应用。不过在使用 Express 之前，在下一章我们先游览一番 Node 的世界。要理解 Express 是如何工作的，这是很重要的一步。

第2章

Node的世界

如果你还没有 Node 开发经验，本章就是为你写的。要理解 Express 及其用处，首先要对 Node 有基本的理解。如果你已经有使用 Node 构建 Web 应用的经验，则可以放心地跳过本章。本章将使用 Node 构建一个最小的 Web 应用，下一章将使用 Express 来实现同样的构建。

2.1 获取Node

将 Node 安装到系统上再容易不过了。为确保所有主要系统中的安装过程都简单直接，Node 团队已经付出了很多努力。

打开 Node 官网，点击写着版本号及“LTS (Recommended for Most Users)”的绿色大按钮。LTS 代表长期支持 (Long-Term Support)，某种程度上比 Current 版本更稳定，后者包含更多的最新特性和性能提升。

对于 Windows 和 macOS 用户来说，点击上述按钮就可以下载安装程序，内有安装指导。而对于 Linux 用户来说，使用包管理器安装和运行或许会更快。



如果你是想要使用包管理器的 Linux 用户，确保遵循上述网页中的步骤。不增加合适的包仓库配置的话，很多 Linux 发行版安装的将是非常老的 Node 版本。

如果你想在组织内分发 Node，也可以下载一个独立安装程序。

2.2 使用终端

我是那种迷恋于终端（也叫控制台或命令行）的威力和生产率而无法自拔的人。本书中的所有示例都假设你使用终端。如果还不熟悉所选用的终端，强烈建议你花一点儿时间熟悉一下。本书使用的很多工具有相应的 GUI，如果你实在不想使用终端，也可以选择别的，但就得自己想办法了。

如果你使用的是 macOS 或 Linux，那么就会有好多可贵的 shell（终端的命令解释器）可供选择。目前为止最流行的是 bash，不过 zsh 也有很多拥护者。我更青睐 bash 的主要原因（除了长期的“亲密接触”外）是它的无处不在。如果你坐到一台装有 Unix 系统的计算机前，99% 的情况下，默认 shell 是 bash。

如果你是 Windows 用户，事情就没有那么美好了。因为微软从来没有对提供一种舒服的终端体验上过心，所以你得自己做些工作。Git 体贴地附带了一个叫“Git bash”的 shell，提供了类似 Unix 的终端体验（虽然它只包含 Unix 上可用命令行工具的一个小子集，但很有用）。不过，Git bash 提供的是一个最小的 bash shell，仍然要使用 Windows 内建的控制台应用，所以有时会给你带来一些麻烦（即便是像调整窗口大小、选择文本、剪切和复制这样简单的功能，用起来都有些别扭）。为此，推荐安装一个更高级的终端，例如 ConsoleZ 或 ConEmu。对于 Windows 高级用户，尤其是 .NET 开发者以及系统管理员和网络管理员，还有另一种选择：使用微软自己的 PowerShell。PowerShell 名副其实，它可以把工作完成得非常出色，一个熟练的 PowerShell 用户完全不输于一个 Unix 命令行老手。不过，如果要在 macOS/Linux 和 Windows 之间切换，建议你还是使用比较通用的 Git bash。

如果你使用的是 Windows 10 或更新版本，则可以直接在 Windows 上安装 Ubuntu Linux。Ubuntu Linux 不是双系统或虚拟化的结果，而是微软开源团队所做的卓越工作，给 Windows 带来了 Linux 的体验。可以通过微软应用商店安装 Ubuntu。

Windows 用户的最后选择是虚拟化。得益于现代计算机的强大性能和体系结构，现在虚拟机（VM）的性能跟真实机器相差无几。我就幸运地遇到过免费的 Oracle VirtualBox。

最后，无论在什么系统中，都可以使用非常好用的云平台开发环境，比如 Cloud9（现在是 AWS 的一个产品）。Cloud9 会给你启动一个新的 Node 开发环境，让你可以快速上手 Node。

选定了一个觉得很舒服的 shell 之后，建议你花点儿时间了解它的基本用法。网上有很多非常好的课程材料（“The Bash Guide”是很好的新手指导）。现在学一点儿可以为将来免去很多麻烦。至少，你应该知道如何切换目录，如何复制、移动和删除文件，以及如何中断一个命令行程序（通常是 Ctrl-C）。如果想成为一位终端“忍者”，建议你学一下如何搜索文件中的文本、如何搜索文件和目录、如何把多个命令链接起来（传说中的“UNIX 哲学”），以及如何重定向输出。



在很多类 Unix 系统中，Ctrl-S 有特别的含义——它会“冻结”终端（因此曾被用来暂停快速滚动的输出）。由于它也是常见的“保存”快捷键，因此用户很容易不假思索地按下去，从而导致很多人很困惑（虽然不想承认，但这种事情经常发生在我身上）。要想从“冻结”中恢复，只需按下 Ctrl-Q。因此，如果你的终端好像突然冻结了，你也没明白怎么回事，试着按下 Ctrl-Q 看看能不能恢复。

2.3 编辑器

在程序员中，很少有话题能像编辑器的选择那样引发激烈的争论。这很合理，毕竟编辑器是你首要的工具。我选择的编辑器是 vi（或支持 vi 模式的编辑器）。¹vi 未必适合所有人（每当我对同事们说，要是我用 vi 来完成他们的工作会多么轻而易举，他们总会冲我翻白眼），但寻找一款强大的编辑器并学会使用它，可以大幅提升你的工作效率，而且我敢说，这样也可以大大增加工作的乐趣。我特别喜欢 vi 的一个原因（虽然不是最重要的原因）是，和 bash一样，它无处不在。只要能使用 Unix 系统，就能使用 vi。很多流行的编辑器有“vi 模式”，可以让你使用 vi 的键盘命令。一旦你习惯了 vi，就很难再去使用别的编辑器了。虽然 vi 刚用起来有些难，但为其付出努力是值得的。

如果你像我一样，看到了熟悉一个无处不在的编辑器的好处，还可以选择 Emacs。虽然我不惯 Emacs（人们一般要么习惯使用 Emacs，要么习惯使用 vi），但对于它的强大和灵活，我绝对是信服的。如果 vi 的分模式编辑方式不适合你，推荐你仔细看看 Emacs。

尽管掌握一个控制台编辑器（例如 vi 或 Emacs）会非常方便，你可能还是想要掌握一个更现代的编辑器。一个流行的选择是 Visual Studio Code（不要跟不带 Code 的“Visual Studio”混淆）。诚心推荐 Visual Studio Code，它设计得非常好：运行快速、工作效率高，完全适合 Node 和 JavaScript 开发。另一个流行的选择是 Atom，它在 JavaScript 社区也很受欢迎。这两个编辑器在 Windows、macOS 和 Linux 中均可免费使用，而且都支持 vi 模式。

现在有了编辑代码的好工具，让我们把注意力转移到 npm 上来。npm 可以帮助我们获取别人写好的包，这样就可以利用这个庞大而活跃的 JavaScript 社区了。

2.4 npm

npm 就是那个时常见到的管理 Node 包的包管理器（也用来获取和安装 Express）。npm 继承了 PHP、GNU、WINE 等的幽默传统，它不是一个首字母缩写词（这就是它不用大写的原因），相反，它是“npm is not an acronym”（npm 不是缩写）的递归缩写。

注 1：现今，vi 基本上成了 vim 的同义词。在大部分系统中，命令 vi 是 vim 的别名，但我一般都输入 vim，以确保用的是 vim。

大体来说，包管理器的两个主要职责就是安装包和管理包依赖关系。npm 是一个快速、能干、易用的包管理器，Node 生态能发展得如此迅速和多姿多彩，我想其中有 npm 很大的功劳。



另外还有一个很流行的相竞争的包管理器叫 Yarn，它跟 npm 使用同一个包数据库。我们将在第 16 章使用 Yarn。

如果你是照着先前的步骤安装 Node，npm 就会一起安装了，因此，你已经获得 npm 了。现在可以开始工作了。

你在 npm 中使用的主要命令将是 `install`（一点儿也不奇怪）。比如说要安装 `nodemon`（一个流行的工具，用于修改代码时自动重启 Node 应用），你就在控制台中运行以下命令：

```
npm install -g nodemon
```

`-g` 参数告诉 npm，这个包要全局安装，意味着在系统中全局可用。等到讨论 `package.json` 文件的时候，这个差别就会更清楚了。就现在来说，经验法则是 JavaScript 工具（例如 `nodemon`）一般应该全局安装，而那些特定于 Web 应用或项目的包就不需要这样了。



不像 Python 等语言（Python 从 2.0 到 3.0 经历了大变更，需要一种办法来为切换环境提供便利），Node 平台足够新，很可能你运行的就是最新版的 Node。不过如果你觉得需要支持多版本的 Node，可以看看 `nvm` 或 `n`，它们支持切换环境。想知道你的计算机上安装的是什么版本的 Node，键入 `node --version` 即可。

2.5 用Node做一个简单的Web服务器

如果你曾搭建过静态 HTML 网站，或者有 PHP 或 ASP 背景，很可能熟悉 Web 服务器（例如 Apache 或 IIS）这个概念。Web 服务器提供静态文件服务，以便浏览器通过网络读取。例如，如果你创建了文件 `about.html` 并将其放到合适的目录，那么在浏览器中就可以导航到 `http://localhost/about.html`。取决于你的 Web 服务器配置，你或许能够省去 `.html`，不过 URL 和文件名之间的关系是清楚的：Web 服务器知道这个文件在计算机上的哪个位置，然后提供给浏览器。



正如其名，`localhost` 指的是你所使用的计算机。这是 IPv4 回送地址 `127.0.0.1` 或 IPv6 回送地址 `::1` 的常见别名。虽然 `127.0.0.1` 很常见，但本书中将使用 `localhost`。如果你在使用远程计算机（例如正在使用 SSH），要记得浏览 `localhost` 时不会连到那台计算机上。

比起传统的 Web 服务器，Node 提供了一种不同的范式：你所写的应用，其本身就是 Web 服务器。Node 只是给你提供了构建 Web 服务器的框架。

你可能会说：“可是，我不想写一个 Web 服务器。”这是很自然的反应：你只是想写一个应用，而不是 Web 服务器。但是，Node 使得写 Web 服务器变成了很简单的事情（甚至只需几行代码），而你也获得了对应用的控制，两全其美。

所以就这么干吧。你已经安装了 Node，熟悉了终端，现在可以继续了。

2.5.1 Hello world

常见的编程入门示例都是平平无奇的“Hello world”，我时常为此感到遗憾。不过要是悍然挑战这一沉闷的传统，未免显得十分不敬，因此我们还是先这样开始，之后再创建更有趣的东西。

在你喜欢的编辑器里，创建一个文件 helloworld.js（版本库的 ch02/00-helloworld.js）：

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('Hello world!')
})

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate....'))
```



跟学习 JavaScript 的经历有关，你可能会对示例代码中缺少了分号感到不安。我曾是分号的“顽固分子”，后来做了一些 React 开发后才勉强不再使用分号，毕竟 React 开发的习惯就是不用分号。不久之后，我终于想开了，奇怪为何自己过去对分号那么执着。现在我是坚定的“无分号”阵营的一员，书中的示例代码也会体现这一点。这是个人选择，如果你想用分号，尽管用。

确认你位于 helloworld.js 所在的目录下，然后输入 `node hello world.js`。打开浏览器，导航到 `http://localhost:3000`。好啦！这就是你的第一个 Web 服务器。这个服务器并没有提供 HTML，它只是在浏览器上显示文本消息“Hello world!”。如果你愿意，可以体验一下发送 HTML：只需把 `text/plain` 改为 `text/html`，并把 `'Hello world!'` 改为合法的 HTML 字符串。不过，我没有那么演示，因为我试图避免在 JavaScript 里写 HTML，至于其中的原因，第 7 章会详细讨论。

2.5.2 事件驱动编程

Node 背后的核心哲学就是事件驱动编程。对你（程序员）就意味着，必须要知悉有什么事件可能会发生以及如何响应。很多人是通过实现用户界面才第一次认识事件驱动编程的：用户点击了某个东西，然后你处理一个点击事件。事件驱动编程是很好的比喻，因为很显然程序员没法控制用户是否点击以及何时点击一个东西，因此，事件驱动编程是非常直观的。把这个概念推广到服务器上，会略微有些难以理解，但基本原理一样。

在前面的示例代码中，事件是隐含的：正在处理的事件就是一个 HTTP 请求。`http.createServer` 方法需要一个函数作为参数，每次创建新的 HTTP 请求时，就会调用这个函数。简单示例只是把内容类型设置为普通文本，然后发送字符串“Hello world!”。

一旦你开始从事件驱动编程的角度思考，就会发现到处都是事件。用户使用你的应用时，从一页导航到另一页，或从一块内容导航到另一块内容，就是一个事件。你的应用如何响应这样的导航事件，就叫作路由。

2.5.3 路由

路由指的是对客户端内容请求的满足机制。对基于 Web 的客户端 / 服务器端应用，客户端在 URL 中指定了想要的内容，具体来讲就是 URL 中的路径和查询串（第 6 章将更详细地讨论 URL 的各个部分）。



通常，服务器端的路由依赖于路径和查询串，但还有别的信息可以使用：HTTP 头、域名、IP 地址，等等。这使得服务器端可以把更多的因素纳入考虑，例如用户的大概位置，或是用户首选的语言。

下面对前面的“Hello world!”做一下扩展，让它做点儿更有意思的事情。做一个只包含主页、“关于”页和“页面未找到”页的小网站。继续前面的例子，仍然只发送普通文本而不是 HTML（版本库的 `ch02/01-helloworld.js`）：

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req,res) => {
  // 对URL做规范化处理：移除查询串、末尾可选的斜杠，并转成小写
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('Homepage')
      break
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('About')
```

```

        break
    default:
        res.writeHead(404, { 'Content-Type': 'text/plain' })
        res.end('Not Found')
        break
    }
}

server.listen(port, () => console.log(`server started on port ${port}; ` +
    'press Ctrl-C to terminate....'))

```

把它运行起来，你会发现现在可以浏览主页（<http://localhost:3000>）和“关于”页（<http://localhost:3000/about>）了。任何查询串都会被忽略（因此 <http://localhost:3000/?foo=bar> 会转到主页），任何其他的 URL（<http://localhost:3000/foo>）都会转到“页面未找到”页。

2.5.4 提供静态资源

现在我们的简单路由已经起作用了，再来提供一些实际的 HTML 和一个 logo 图片。这些被称作 **静态资源**，因为它们一般不会变化（与此相反的例子是股票行情页——每次你刷新页面，股票价格都可能会变化）。



由 Node 来提供静态资源，适用于开发阶段和小项目，而对于稍大一些的项目，你可能希望使用代理服务器，例如 NGINX 或 CDN。更多信息参见第 17 章。

如果你用过 Apache 或 IIS，只要创建一个 HTML 文件，然后从浏览器上访问它，文件就会自动发过来，可能你已经习惯了这种方式。但 Node 的工作模式不是这样的，像打开文件、读取文件、把内容发送到浏览器这些工作，必须我们自己来做。在项目下创建一个名为 public 的目录（为什么不称其为 static，下一章将解释）。在 public 目录下，创建 home.html、about.html、404.html、一个名为 img 的子目录，以及一个名为 img/logo.png 的图片。我把这些留给你来做，你在阅读本书的时候应该已经知道如何编写 HTML 文件和找图片了。在 HTML 文件中，这样来引入 logo: 。

现在修改一下 helloworld.js（版本库的 ch02/02-helloworld.js）：

```

const http = require('http')
const fs = require('fs')
const port = process.env.PORT || 3000

function serveStaticFile(res, path, contentType, responseCode = 200) {
    fs.readFile(__dirname + path, (err, data) => {
        if(err) {
            res.writeHead(500, { 'Content-Type': 'text/plain' })
            return res.end('500 - Internal Error')
        }
        res.writeHead(responseCode, { 'Content-Type': contentType })
        res.end(data)
    })
}

```

```

        })
    }

const server = http.createServer((req,res) => {
  // 对URL做规范化处理: 去除查询串、末尾可选的斜杠, 并转成小写
  const path = req.url.replace(/\/?(?:\?.*)?$/,'').toLowerCase()
  switch(path) {
    case '':
      serveStaticFile(res, '/public/home.html', 'text/html')
      break
    case '/about':
      serveStaticFile(res, '/public/about.html', 'text/html')
      break
    case '/img/logo.png':
      serveStaticFile(res, '/public/img/logo.png', 'image/png')
      break
    default:
      serveStaticFile(res, '/public/404.html', 'text/html', 404)
      break
  }
})

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate...'))

```



在这个例子中，我们的路由处理还比较缺乏想象力。如果你导航到 `http://localhost:3000/about`，会得到 `public/about.html` 文件。你可以把路径改成任何想要的样子，也可以把文件换成任何想要的文件。举个例子，如果对于一周中的每一天，你都有一个不同的“关于”页，那么就会有这些文件：`public/about_mon.html`、`public/about_tue.html`，等等。然后你在路由中增加一些逻辑，这样当用户访问 `http://localhost:3000/about` 时，就可以提供合适的 HTML 文件了。

注意，我们创建了一个辅助函数 `serveStaticFile`，它做了大量的工作。`fs.readFile` 是一个读取文件的异步方法，虽然它也有一个同步的版本 `fs.readFileSync`，但我们最好还是早点儿以异步的方式来思考问题。`fs.readFile` 函数采用了一种称为回调的模式。你要提供一个叫回调函数的函数，当工作一完成，这个回调函数就会调用（所以叫“回调”）。在这里，`fs.readFile` 读取指定文件的内容，读完以后就执行这个回调函数。如果该文件不存在或者读取时出现权限问题，就会调用 `err` 这个变量，然后这个回调函数会向客户端返回 HTTP 状态码 500，指示一个服务器端错误。如果能够成功读取文件，就把这个文件发送到客户端，带上指定的响应码和内容类型。关于响应码的详细信息，参见第 6 章。



`_dirname` 会被解析为当前脚本所在的目录。因此如果脚本位于 `/home/sites/app.js`，`_dirname` 将被解析为 `/home/sites`。如果有可能，最好使用这个顺手的全局变量。否则，如果从一个不同的目录运行应用，就会出现难以诊断的错误。

2.6 进入Express

目前为止，Node 可能还没有给你带来多大的震撼，基本上只是重复了 Apache 或 IIS 为你自动完成的工作。不过，现在你对 Node 的工作方式以及如何控制它，应该有了更深入的理解。虽然我们还没有做出特别震撼的东西，但是你应该能看到，可以以此为起点，做一些更高级的东西。如果继续这么做下去，写出越来越高级的 Node 应用，到最后很可能你得到的东西跟 Express 很相像……

幸运的是，不需要那么做：Express 已经存在了，你不再需要花大量时间去实现各种基础设施。现在我们已经获得了一些 Node 的经验，可以开始学习 Express 了。

第3章

Express的方式

前一章学习了如何仅用 Node 来创建一个简单的 Web 服务器。本章将使用 Express 来重新创建该服务器，这一方面可以为你介绍 Express 的基础知识，另一方面也为本书的后续内容提供了一个起点。

3.1 脚手架

脚手架不是什么新想法，但很多人（包括我自己）是首先通过 Ruby 接触到这个概念的。脚手架这个想法很简单：大部分项目需要某些所谓的**样板**代码，而谁会愿意每开始一个项目就重写一遍那些代码呢？一个简单的办法就是创建一个项目的框架，每次你需要一个新项目时，就复制这个框架（或叫模板）。

Ruby On Rails 在这个概念的基础上更进一步，它提供了一个程序，可以自动生成脚手架代码。比起从一组模板中选择，这种做法可以生成更高级的框架，这就是它的优势。

Express 借鉴了 Ruby On Rails，它提供了一个工具来生成脚手架，帮助你开始一个 Express 项目。

虽然 Express 脚手架工具很有用，但我认为学习如何从头搭建一个 Express 项目也很有价值。除了可以学习更多东西，你也可以对项目有更多的控制，包括要安装什么包以及采用什么样的目录结构。此外，Express 的脚手架工具更适合服务器端 HTML 的生成，跟 API 和单页应用关系不大。

虽然我们不使用脚手架工具，但是我鼓励你在读完本书后去看一看它。到那时你已经掌握

了各种必备的知识和技能，可以评估脚手架工具所生成的脚手架对你是否有用了。要了解更多信息，请查看 express-generator 的文档。

3.2 草地鹨旅游网站

贯穿本书，我们将使用一个持续更新的示例网站：为草地鹨旅游公司（Meadowlark Travel）而做的一个假想的网站，这个公司是为到俄勒冈州旅行的人提供服务的。如果你对创建 API 更感兴趣，不用担心：这个网站除了提供功能正常的网页之外，还会暴露一套 API。

3.3 初始工作

先创建一个新目录，这将是你的项目的根目录。在本书中，每当说项目目录、应用目录或应用根目录，都是指这个目录。



你可能想让 Web 应用的文件跟通常属于项目的其他文件（例如会议笔记、文档等）分离。为此，推荐你把应用目录作为项目目录的子目录。例如，对于草地鹨旅游网站，我会把项目放在 `~/projects/meadowlark` 中，把应用放在 `~/projects/meadowlark/site` 中。

npm 在一个名为 `package.json` 的文件里管理项目的依赖关系以及项目的元数据。创建这个文件的最简单方式是运行 `npm init`：它会问你一系列问题，然后生成 `package.json` 文件以开始项目（对于“entry point”这个提问，按照项目名，使用 `meadowlark.js`）。



每次运行 npm，你可能都会收到一条警告信息，提示缺少项目描述或版本库字段。忽略这些警告是没问题的，不过如果你想消除它们，则需要编辑 `package.json` 文件，填上 npm 提示缺少的字段值。关于此文件各个字段的更多信息，请查看 npm 的 `package.json` 文档。

首先安装 Express。运行以下 npm 命令：

```
npm install express
```

运行 `npm install` 会把指定的包安装到 `node_modules` 目录下，并更新 `package.json` 文件。既然任何时候 npm 都可以重新生成 `node_modules` 目录，那么就不应该把这个目录保存进版本仓库。为了确保不会无意地把 `node_modules` 目录加进版本仓库，创建一个名为 `.gitignore` 的文件：

```
# ignore packages installed by npm
node_modules

# put any other files you don't want to check in here, such as .DS_Store
# (OSX), *.bak, etc.
```

接着创建一个名为 meadowlark.js 的文件。这将是项目的入口点 (entry point)。本书中简单地把这个文件叫作应用文件 (版本库的 ch03/00-meadowlark.js)：

```
const express = require('express')

const app = express()

const port = process.env.PORT || 3000

// 定制404页
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 - Not Found')
})

// 定制500页
app.use((err, req, res, next) => {
  console.error(err.message)
  res.type('text/plain')
  res.status(500)
  res.send('500 - Server Error')
})

app.listen(port, () => console.log(
  `Express started on http://localhost:${port}; ` +
  `press Ctrl-C to terminate. `))
```



很多教程以及 Express 脚手架生成器推荐把主文件命名为 app.js (有时候是 index.js 或 server.js)。除非在使用某个托管服务或部署系统，其要求你的项目主文件使用特定的文件名，否则我觉得没有必要如此命名，我更喜欢按项目名来命名这个主文件。在编辑器中写代码时，如果谁有过盯着一排标签页看而它们全都叫“index.html”的经历，那他就会立刻明白我的做法有多么明智。`npm init` 命令默认是 index.js，如果你的应用文件使用不同的文件名，请确保更新 package.json 文件的 `main` 属性。

现在你已经有了一个小的 Express 服务器。你可以启动服务器 (`node meadowlark.js`)，在浏览器中打开 `http://localhost:3000`。结果可能会让你失望：你还没有给 Express 提供路由，因此它只会给你一条泛泛的 404 信息，指示页面不存在。



注意我们是如何选择想让应用运行的端口的：`const port = process.env.PORT || 3000`。这样就可以在启动服务器之前先设置环境变量，覆盖 3000 端口。如果在你启动之后，应用没有在 3000 端口上运行，检查看看是否已经设置了 PORT 环境变量。

让我们给主页和“关于”页加上一些路由。在 404 处理函数前，增加两个路由 (版本库的 ch03/01-meadowlark.js)：

```
app.get('/', (req, res) => {
  res.type('text/plain')
  res.send('Meadowlark Travel');
})

app.get('/about', (req, res) => {
  res.type('text/plain')
  res.send('About Meadowlark Travel')
})

// 定制404页
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 - Not Found')
})
```

`app.get` 就是用来增加路由的方法。在 Express 文档中，你会看到 `app.METHOD`。并不是说实际存在一个名为 `METHOD` 的方法，它只是你的（小写形式的）HTTP 动词（`get` 和 `post` 最常见）的占位符。这个方法接收两个参数：一个路径和一个函数。

这个路径就界定了一个路由。注意，`app.METHOD` 方法为你做了很多事情：默认情况下，它不区分大小写，也不介意末尾有没有斜杠，在匹配路径时也不考虑查询串部分。因此对于“关于”页的路由，`/about`、`/About`、`/about/`、`/about?foo=bar`、`/about/?foo=bar` 等 URL 都可以匹配。

当路由匹配成功时，你提供的函数会得到执行。传入这个函数的参数是 `request` 和 `response` 对象，第 6 章将深入学习它们。目前，我们只是返回一条文本和一个状态码 200（Express 默认的状态码就是 200，你不需要明确指定）。



强烈建议你安装一个浏览器插件，这个插件可以向你显示 HTTP 状态码以及可能发生的重定向。它让你更容易发现代码中重定向的问题或不正确的状态码，这些都是常常被忽视的。对于 Chrome，Ayima 的 Redirect Path 插件十分好用。在大多数浏览器中，你可以在开发者工具中的“网络”部分查看状态码。

我们不再使用 Node 的更低层的 `res.end` 方法，而是切换到了 Express 扩展的 `res.send` 方法。也会把 Node 的 `res.writeHead` 替换为 `res.set` 和 `res.status`。Express 也提供了一个便利的方法，即 `res.type`，它可以设置 `Content-Type` 头信息。仍旧使用 `res.writeHead` 和 `res.send` 也可以，但没必要或不推荐。

注意定制的 404 页和 500 页的处理方式必须有些许不同。对于它们，我们不使用 `app.get`，而是使用 `app.use`。`app.use` 就是 Express 用来增加中间件的方法。第 10 章将深入讨论中间件，但就目前来说，你可以把这些中间件看作一个兜底的处理函数，能够处理任何在前面没有得到路由匹配的请求。由此可以得出一个重要结论：在 Express 中，路由和中间件的

加入顺序至关重要。如果把 404 处理函数放在路由之前，主页和“关于”页将不再正常，那些 URL 都会“走”到 404 页。现在，虽然我们的路由还十分简单，但由于它们也支持通配符，因此也会产生顺序的问题。举个例子，如果想给“关于”页增加子页面，比如 /about/contact 和 /about/directions，该怎么做？写成下面这样是不会按预期工作的：

```
app.get('/about*', (req, res) => {
  // 发送内容……
}) app.get('/about/contact', (req, res) => {
  // 发送内容……
}) app.get('/about/directions', (req, res) => {
  // 发送内容……
})
```

在这个例子中，/about/contact 和 /about/directions 这两个路由永远得不到匹配，因为第一个处理函数在路径中使用了通配符：/about*。

Express 可以区分 404 和 500 这两个处理函数，依靠的是回调函数接收的参数个数。错误处理路由将在第 10 章和第 12 章中进一步讨论。

现在可以重新启动服务器，看看主页和“关于”页是否正常了。

目前为止我们所做的事虽然没有 Express 也很容易完成，但是 Express 正在默默地提供一些帮助。回想前一章：为了确定请求的是什么资源，必须对 req.url 做规范化处理。我们不得不手动去除 URL 的查询串和末尾斜杠，并把它转换成小写。现在 Express 的路由自动处理了这些细节。虽然看起来或许不是什么大事，但这只是 Express 路由能力的冰山一角。

3.4 视图与布局

如果熟悉“模型 – 视图 – 控制器”这个范式，那么你对视图这个概念就不会陌生。从根本上说，视图就是最终发送给用户的东西。对于网站来说，视图通常意味着 HTML，尽管你也可以发送 PNG、PDF 或任何客户端能够渲染的东西，但就我们的目的而言，可以认为视图就是 HTML。

与静态资源（例如图片或 CSS 文件）不同，视图不一定是静态的：对于每个请求，HTML 都可以即时地构造出来，从而提供一个定制的页面。

Express 支持很多视图引擎，这些引擎提供了不同的抽象层次。Express 更偏爱一个名为 Pug 的视图引擎（也难怪，毕竟 Pug 也是 TJ Holowaychuk 的“手笔”）。Pug 采取的方法是最小化：你写的东西一点儿也不像 HTML，这肯定就代表着更少的键盘输入（不再有尖括号或结束标签）。Pug 引擎以此输入，把它转换成 HTML。



Pug 最初叫 Jade，由于商标问题，到 2.0 发布的时候就改名了。

Pug 很吸引人，但那个层次的抽象会带来一些代价。如果你是一个前端开发人员，即使实际是在用 Pug 写视图，也必须要深入理解 HTML。对于我认识的多数前端开发人员来说，把主要的标记语言抽象掉是很不舒服的。为此，推荐使用另一个不那么抽象的模板框架，即 Handlebars。

Handlebars（基于独立于语言的模板引擎 Mustache，非常流行）并不会让你直接抽象掉 HTML：你仍然写 HTML，只是包含专门的标记，以便让 Handlebars 输入内容。



在本书第 1 版出版的几年中，React 席卷了世界……它竟把前端开发人员的 HTML 抽象掉了！现在回头看，我说的“前端开发人员不想把 HTML 抽象掉”的预言，没有经受住时间的考验。不过，JSX（大多数 React 开发人员使用的 JavaScript 语言扩展）跟写 HTML（几乎）一样，所以我也没有全错。

为了支持 Handlebars，我们准备使用 Eric Ferraiuolo 的 `express-handlebars` 包。在项目目录下，执行以下命令：

```
npm install express-handlebars
```

然后在 `meadowlark.js` 文件中，修改前面几行（版本库的 `ch03/02-meadowlark.js`）：

```
const express = require('express')
const expressHandlebars = require('express-handlebars')

const app = express()

// 配置Handlebars视图引擎
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
app.set('view engine', 'handlebars')
```

这里创建了一个视图引擎，配置 Express 默认会使用这个引擎。现在创建一个名为 `views` 的目录，再在其下面创建一个名为 `layouts` 的子目录。如果你有 Web 开发经验，那么很可能已经习惯了布局（有时叫作母页面）。当你构建一个网站时，总会有相当数量的 HTML 代码是一样的，或是几乎一样。为每个页面都编写一遍重复的代码，不仅单调乏味，还制造了潜在的维护噩梦：如果想修改重复的东西，就必须修改所有文件。布局让你高枕无忧，它为你的网站的所有页面提供了一套公共的框架。

现在就为我们的网站创建一个模板。创建文件 `views/layouts/main.handlebars`：

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

可能你以前没见过的只有 {{{body}}}。对于每个视图，此表达式将被替换为 HTML。前面创建 Handlebars 实例的时候，我们指定了默认布局 (`defaultLayout: '\main'`)。这意味着除非你另外指定，否则任何视图都会使用这个布局。

现在创建主页的视图，即 `views/home.handlebars`:

```
<h1>Welcome to Meadowlark Travel</h1>
```

然后是“关于”页，即 `views/about.handlebars`:

```
<h1>About Meadowlark Travel</h1>
```

接下来是“页面未找到”页，即 `views/404.handlebars`:

```
<h1>404 - Not Found</h1>
```

最后是服务器错误页，即 `views/500.handlebars`:

```
<h1>500 - Server Error</h1>
```



你可能希望你的编辑器把 `.handlebars` 和 `.hbs` (Handlebars 另一种常见的文件扩展名) 与 HTML 关联起来，这样便可以启用语法高亮和其他编辑器特性。对于 vim，可以在 `~/.vimrc` 文件中增加一行: `au BufNewFile,BufRead *.handlebars set file type=html`。对于其他编辑器，请查阅相关文档。

创建了这些视图之后，需要更新一下路由，这样才能用上这些视图（版本库的 `ch03/02-meadowlark.js`）：

```
app.get('/', (req, res) => res.render('home'))

app.get('/about', (req, res) => res.render('about'))

// 定制404页
app.use((req, res) => {
  res.status(404)
  res.render('404')
})
```

```
// 定制500页
app.use((err, req, res, next) => {
  console.error(err.message)
  res.status(500)
  res.render('500')
})
```

注意我们不再需要指定内容类型或状态码：视图引擎会默认返回 `text/html` 内容类型和 `200` 状态码。在 `404` 页和 `500` 页的兜底处理函数中，必须明确设置状态码。

如果你启动服务器，访问主页或“关于”页，会看到视图得到了渲染。如果你查看页面源代码，会看到来自 `views/layouts/main.handlebars` 的 HTML 样板代码。

即使每次访问主页得到的都是同样的 HTML，这些路由也是动态内容，因为每次调用路由时，我们都可以做出不同的决策（在本书后面章节中将看到很多这样的例子）。不过，永远不变的内容（也就是静态内容）也是很常见的，所以接下来讲一下静态内容。

3.5 静态文件与视图

Express 依靠中间件来处理静态文件和视图。第 10 章将详细讨论中间件这个概念。就现在来说，只要知道中间件提供模块化功能，简化了请求处理，就够了。

`static` 这个中间件允许你指定一个或多个包含静态资源的目录，里面的资源会不加任何特别处理，直接发送给客户端。这些目录就是你放入图片、CSS 文件和客户端 JavaScript 文件的地方。

在项目目录下，创建一个名为 `public` 的子目录（称其为 `public` 是因为这个目录下的任何东西都可以不经询问而发给客户端）。然后，在声明任何路由前，加入 `static` 中间件（版本库的 `ch03/02-meadowlark.js`）：

```
app.use(express.static(__dirname + '/public'))
```

这个 `static` 中间件实现的效果，与为你想发送的每一个静态文件创建路由，再在里面把文件发给客户端的做法，是一样的。在 `public` 下创建子目录 `img`，把 `logo.png` 放进去。

现在只要引用 `/img/logo.png`（注意不要加上 `public`，这个目录对客户端不可见），`static` 中间件就会提供这个文件，并适当地设置内容类型。修改一下布局，让 `logo` 出现在每个页面里：

```
<body>
  <header>
    
  </header>
  {{body}}
</body>
```



请记住，中间件是按顺序处理的，static 中间件——常常是第一个或至少是非常早声明的——会覆盖其他路由。举个例子，如果你把一个 index.html 文件放在 public 目录下（可以试试），就会发现得到的是这个文件的内容，而不是配置的路由。因此，如果你得到了让人迷惑的结果，就检查一下静态文件，看看是不是有什么东西错误地匹配了路由。

3.6 视图中的动态内容

视图如果被用来发送静态 HTML，就把操作复杂化了（尽管也可以做），其真正的威力在于包含动态信息。

比如说在“关于”页，我们想发送一块“虚拟幸运饼”，那就在 meadowlark.js 文件中，定义一个“幸运饼”的数组：

```
const fortunes = [
  "Conquer your fears or they will conquer you.",
  "Rivers need springs.",
  "Do not fear what you don't know.",
  "You will have a pleasant surprise.",
  "Whenever possible, keep it simple.",
]
```

修改视图（/views/about.handlebars）来显示一块“幸运饼”：

```
<h1>About Meadowlark Travel</h1>
{{#if fortune}}
  <p>Your fortune for the day:</p>
  <blockquote>{{fortune}}</blockquote>
{{/if}}
```

现在修改路由 /about 并发送随机的“幸运饼”：

```
app.get('/about', (req, res) => {
  const randomFortune = fortunes[Math.floor(Math.random()*fortunes.length)]
  res.render('about', { fortune: randomFortune })
})
```

现在如果重启服务器，加载“关于”页，就会看到一块随机的“幸运饼”。你每刷新一下页面，就会得到一块新的“幸运饼”。模板化太有用了，第 7 章还会深入讨论。

3.7 小结

我们已经使用 Express 创建了一个基本的网站。尽管很简单，但它还是包含了一个全功能网站的核心要素。下一章将继续完善我们的项目，准备增加更多高级功能。

第4章

项目整顿

在前面两章中，我们只是做了些简单的尝试。在开发更复杂的功能之前，我们需要先做些工作调整，并在工作中养成一些好习惯。

在本章，我们将满腔热忱地开始草地鹨旅游项目。在开始实际编码网站之前，先要准备好创建高质量产品所需的几个工具。



你不必完全遵照本书的代码示例。如果很想做一个自己的网站，可以只采用本书的代码框架，然后按照自己的想法修改。这样到学完本书时，你就完成了一个自己的网站。

4.1 项目文件与目录结构

如何安排项目目录结构一直存在争议，还没有什么所谓“正确”的做法。不过，倒是有一些“惯例”有必要了解一下。

一种惯用做法是限制项目根目录下文件的数量。通常，你可以在根目录下看到若干配置文件（如 package.json）、README.md 文件和多个目录。绝大部分代码放在其中一个名为 src 的目录下。为了简明起见，本书不遵循这个惯例（奇怪的是，Express 的脚手架应用也没有遵循）。在项目实战开发中，如果你把代码直接放在根目录下，最后可能会十分杂乱，于是你会想把它们归拢到一个像 src 这样的目录下。

前面提过，我喜欢按照项目名（meadowlark.js）来命名项目主文件（有时叫入口点），而把普通文件命名为 index.js、app.js 或 server.js。

项目的文件与目录该如何安排，很大程度上由你自己决定。推荐在 README.md 文件（或从它链接到的自述文件）里加上项目文件与目录的导航。

建议项目根目录至少包含 package.json 和 README.md 这两个文件。其余的取决于你的想象力。

4.2 最佳实践

最佳实践如今是个热词，你应该有所耳闻。它意味着应该“把事情做对”，并且不能取巧（一会儿再专门讨论其具体含义）。你应该听说过下面这句工程建设格言：可以又快、又省、又好，但只能选择其中两样。我常常对此“教条”感到不满：它没有考虑使用正确方式做事的累积价值。第一次使用正确的方式做某件事，可能要花比“只求快而不计后果”的做法多 4 倍的时间。可是到了第二次，只要花 3 倍的时间，而十几次之后，就可以跟“只求快而不计后果”的做法一样快了。

一位击剑教练曾告诫我说：练习未必达成完美，它只是在固化。也就是说，如果你一遍遍地做某事，最终就会形成无意识的机械行为。这句话是对的，但它忽略了练习的质量。如果你不断练习的是坏习惯，那么坏习惯就会机械化。因此，应该遵守“完美的练习才能达成完美”的法则。我鼓励你在此法则的指引下跟随本书后面的示例代码，就好像在构建一个真实的网站，且你的声誉和收入都依赖这个网站的质量。使用本书的目的，不应该只是学习新技能，而是通过练习，养成一些好习惯。

我们将主要关注版本控制和质量保证（QA）方面的实践。本章会讨论版本控制，QA 将在下一章讨论。

4.3 版本控制

希望不需要我来告诉你版本控制的重要性（否则恐怕得再写一本书）。大体来说，版本控制可以带来以下好处。

历史记录

版本控制能够提供很有价值的历史记录：通过追溯项目的历史，可以回顾项目中做过的决策，并查看各个组件的开发顺序。总之，一份技术历史记录是很有用的。

责任追溯

如果在团队中工作，责任追溯则非常重要。每当你发现代码中有看不懂或存在疑问的地方，知道去问谁能节省好几个小时的时间。或许代码中的注释足以回答你的问题，即便不能，你也知道去问谁。

编码试验

一个好的版本控制系统可以让你放心地进行各种编码试验。你可以放下正在进行的工

作，来试验一些新东西，而无须担心会影响项目的稳定性。如果试验成功，就把代码合并进项目；如果不成功，丢弃即可。

多年前，我把版本控制系统切换成了分布式版本控制系统（DVCS），最后将选择范围缩小到 Git 和 Mercurial，并选择了 Git。选择 Git 是因为它无处不在且设计灵活。当然，Git 和 Mercurial 都是很优秀且免费的版本控制系统，值得推荐。本书中将使用 Git，你要使用 Mercurial（或干脆使用其他的）也可以。

如果不熟悉 Git，推荐你阅读 Jon Loeliger 的精品大作 *Version Control with Git* (O'Reilly 出版)。

4.4 在本书学习中如何使用 Git

首先确认是否安装了 Git。键入 `git --version`，如果没有输出版本号信息，则需要先安装 Git。关于如何安装 Git，请查看其官方文档。

本书中的代码示例有两种使用方式。一种是亲手录入每个示例的代码，然后再输入 Git 命令进行版本管理。另一种是复制用于所有示例的版本库，然后检出每个示例的代码文件。有人自己录入代码可以学得更好；有人则更愿意查看完整代码，做些修改后再运行，免得全部重新录入。

4.4.1 亲手录入

我们的项目已经有了十分粗糙的框架，即包含几个视图文件、一个布局文件、一个图标文件、一个项目主文件和一个 `package.json` 文件。接下来，创建一个 Git 版本库，并把所有这些文件加进去。

首先进入项目目录，在这里初始化一个 Git 版本库：

```
git init
```

在把文件加入版本库之前，为了避免误加文件，在项目目录中创建一个名为 `.gitignore` 的文本文件。在这个文件里，你可以加入任何想要 Git 默认忽略的文件或目录（一行一个）。文件里也支持通配符。例如，如果代码编辑器会创建以波浪线结尾的备份文件（如 `meadowlark.js~`），就可以把 `*~` 加入 `.gitignore` 文件。如果使用 Mac 系统，应该把 `.DS_Store` 加入进去。也应该把 `node_modules` 加入进去（原因一会儿再讨论）。这时，这个文件看起来如下所示：

```
node_modules
*~
.DS_Store
```



.gitignore 文件中的条目也会应用到子目录。因此，如果你把 *~ 加入到项目根目录的 .gitignore 文件里，那么所有子目录中的备份文件都会被忽略。

现在可以把所有文件都加入版本库了。对此，Git 有多种做法。我通常更喜欢用一次能提交全部变更的 `git add -A`。但如果你是 Git 新手，我的建议是，如果只想提交一两个文件，就一个一个地加入（如 `git add meadowlark.js`），否则就使用 `git add -A` 一次性提交全部变更（包括你删除的文件，如果有的话）。不过，既然想提交所有的工作，那么就使用：

```
git add -A
```



Git 的新用户常常搞不清 `git add` 命令，此命令加入的是对文件所做的变更，不是文件。因此，如果修改了 `meadowlark.js`，然后键入 `git add meadowlark.js`，那么实际是把你对这个文件所做的变更加入版本控制。

Git 有一个“暂存区”，当运行 `git add` 时，变更就记录到那里。因此，刚加入的变更实际上还没有提交，只是准备好提交了。使用 `git commit` 提交变更：

```
git commit -m "Initial commit."
```

`-m "Initial commit."` 选项允许写一条关于此次提交的说明。如果没有提交说明，Git 则不允许提交变更。这是合理的。提交说明应当言之有理，能够简洁而完整地描述你做了什么变更。

4.4.2 使用官方版本库

要获取本书的官方版本库，请登录 GitHub 官网，搜索 `EthanRBrown/web-development-with-node-and-express-2e`。

版本库中每个有代码的章节都有一个对应的目录。例如，本章的代码可以在 `ch04` 这个目录中找到。为了便于参考，每章的代码示例通常都加了数字编号。在整个版本库中，我已经增加了很多 `README.md` 文件，包含示例的额外说明。



本书第 1 版的版本库有所不同：它包含随时间变化的历史版本，就好像你正在开发一个越来越高级的项目。虽然这种方式可以理想地反映一个真实的项目开发过程，但给我以及我的读者带来了很多麻烦。随着 `npm` 包的升级，示例代码也会有所变更，我们又不想重写版本库的整个历史，所以没有什么好办法去更新版本库或用文本说明这些变更。虽然现在每章一个目录的做法无法体现项目随时间的自然变化，但使得本书文本跟版本库更加贴近，也更方便大家为社区做贡献。

这个版本库也会随着本书的修订而更新。当版本库更新时，我会打上版本标签，这样当你在读本书的某个版本时，便可以检出对应版本的代码。版本库当前的版本是 2.0.0。我会大致遵循语义版本的原则（本章后面会做更多说明）。补丁增量（最后一个数字）表示微小变更，不太会影响你跟随该版本的书本学习。也就是说，如果版本库处于版本 2.0.15，则依旧对应本书的当前版。可是，如果小版本增量（第二个数字）不同（2.1.0），则说明版本库的内容跟当前你在读的这个版本有所不同了，你应该检出版本以 2.0 开头的版本。

版本库从优使用 README.md 文件对示例代码进行说明。



无论何时你想要做些试验，要记得，是从标签检出的版本使你进入了 Git 所说的“分离头”状态。尽管你可以放心地编辑任何文件，但如果不想创建一个分支，你提交东西就不安全。因此，如果的确想要一个基于某个标签的试验性分支，只需创建一个新的分支并检出。对此，使用一条命令即可：`git checkout -b experiment`（这里 `experiment` 就是你的分支名，想叫什么都可以）。然后在这个分支上，所有编辑和提交都是安全的。

4.5 npm包

项目依赖的 npm 包就位于名为 `node_modules` 的目录下（称其为 `node_modules` 而不是 `npm_packages` 有些遗憾，因为 Node 模块虽然相关却是不同的概念）。你可以放心地探索这个目录，不管是为了满足好奇心还是为了调试程序，但千万不要修改里面的任何代码。随意修改代码本就是个坏习惯，而且你所做的所有修改都可能被 npm 轻易抹去。

如果你需要对项目依赖的某个包做修改，正确的做法是创建自己的分支版本库（fork）。如果你的确这样做了，并且觉得所做的改进对其他人也会有用，祝贺你：你参与了一个开源项目！如果变更符合项目标准，你就可以提交了，这些变更会被包含进官方的包。为一个现有的包做贡献并创建定制构建超出了本书的范畴，不过如果想做贡献，活跃的开发者社区随时都可以帮助你。

`package.json` 文件的两大用途是描述项目和列出项目依赖。现在就打开 `package.json` 文件看看。你应该会看到类似下面这样的代码（具体的版本号很可能不一样，因为包更新总是很频繁）：

```
{  
  "dependencies": {  
    "express": "^4.0.0",  
    "express-handlebars": "^3.0.0"  
  }  
}
```

现在 `package.json` 文件只包含关于依赖的信息。包版本号前的脱字符（`^`）表示从这个指定版本号开始到下一个大版本号的版本之前都可以。比如，这个 `package.json` 文件表示任

从 4.0.0 开始的 Express 版本都可以。因此，4.0.1 和 4.9.9 都没问题，3.4.7 则不行，5.0.0 也不行。执行 `npm install` 时安装的是指定依赖版本，一般比较保险。但这样的结果是，如果想升级到更新的版本，就必须编辑文件并指定这个新版本。通常来说这是好事，因为避免了在你不知道的情况下，依赖发生了变更而破坏项目。npm 中的版本号是由一个叫 semver（表示 semantic versioning）的组件来解析的。如果想了解更多有关 npm 中版本的信息，请查看“语义版本规范”。



“语义版本规范”规定采用语义版本的软件必须声明一个“公开 API”。我总觉得这个规定令人困惑，它其实想表达：一定会有人关心如何跟你的软件实现对接。如果从广义角度考虑，它可以解释成任何东西。因此不要理会规范的这个部分，重在语义版本的格式。

既然 `package.json` 文件已经列出了所有的依赖，那么 `node_modules` 目录实际上只是它的派生物。也就是说，即使你删除了 `node_modules`，要让项目再次正常运行，需要做的也只是运行 `npm install`，这样就会重新创建这个目录，把运行项目所有必需的依赖都放进去。正因如此，我推荐把 `node_modules` 加入 `.gitignore` 文件，但不包含进版本控制里。不过，有些人觉得版本库应该包含任何运行项目所必需的东西，所以把 `node_modules` 包含进了版本控制里。但我觉得这是版本库的“噪音”，宁愿不要它。



npm 版本 5 会创建一个额外的文件：`package-lock.json`。`package.json` 对依赖版本的规范是松散的（如果用了`^`和`~`修饰符），而 `package-lock.json` 记录了已安装的确切版本，如果你需要重建确切的依赖版本，那么它很有用。我推荐你把这个文件加入版本控制，但不要手动改动它。更多信息请查看 npm 官方文档的 `package-lock.json` 部分。

4.6 项目元数据

`package.json` 文件的另一个用途是存储项目元数据，比如项目名、作者、许可信息，等等。如果你最初是用 `npm init` 创建的 `package.json` 文件，它就会给你填上一些必要的字段，随时可以修改。如果你打算把自己的项目放到 npm 或 GitHub 上供他人使用，元数据就十分关键了。想要了解关于 `package.json` 各字段的更多信息，请查看 npm 官方文档的 `package.json` 部分。另外一块重要的元数据是 `README.md` 文件。这个文件很方便：它描述了网站总体架构以及一个人初次接触项目时可能需要的关键信息。`README.md` 是一个纯文本的 wiki 格式，叫作 Markdown。更多信息请查看 Markdown 文档。

4.7 Node 模块

前面已经提到，Node 模块跟 npm 包虽然相关却是不同的概念。正如其名，Node 模块提供

了一个模块化和封装的机制。npm 包提供了存储、版本管理和引用其他项目（并不限于模块）的标准方案。例如，在项目主文件中把 Express 本身作为一个模块导入：

```
const express = require('express')
```

`require` 是用于导入模块的一个 Node 函数。默认情况下，Node 会在 `node_modules` 目录下查找模块（毫无疑问，`node_modules` 下面会有一个 `express` 目录）。不过，Node 也提供了创建你自己的模块的机制（绝对不要把你自己的模块创建到 `node_modules` 目录下）。除了通过包管理器安装到 `node_modules` 下的模块，Node 还提供了 30 多个“核心模块”，比如 `fs`、`http`、`os` 和 `path`。要了解模块的完整列表，可以参考 Node 的官方文档。

对于前一章实现的“幸运饼”的功能，我们来看看怎么把它模块化。

首先创建一个用于存储模块的目录。你把它叫什么都可以，但叫 `lib` (`library` 的缩写) 最常见。在这个目录下，创建一个名为 `fortune.js` 的文件（版本库的 `ch04/lib/fortune.js`）：

```
const fortuneCookies = [
  "Conquer your fears or they will conquer you.",
  "Rivers need springs.",
  "Do not fear what you don't know.",
  "You will have a pleasant surprise.",
  "Whenever possible, keep it simple."
]

exports.getFortune = () => {
  const idx = Math.floor(Math.random()*fortuneCookies.length)
  return fortuneCookies[idx]
}
```

这里最需要注意的是全局变量 `exports` 的使用。如果想让什么东西在这个模块以外可见，就必须把它加到 `exports` 里。在这个例子中，函数 `getFortune` 是模块外可以用的，但数组 `fortuneCookies` 是完全隐藏的。这是个好事：封装有助于减少易错和脆弱的代码。



从一个模块导出功能有好几种方式。本书中会讨论多种方式，并在第 22 章做归纳。

现在，在 `meadowlark.js` 中，可以移除 `fortuneCookies` 数组了（尽管留着也不会有什么害处，它也不可能跟 `lib/fortune.js` 里定义的同名数组相冲突）。在文件顶部导入模块是传统做法（但不是必须的），所以在 `meadowlark.js` 文件的顶部，加入下面这行代码（版本库的 `ch04/meadowlark.js`）：

```
const fortune = require('./lib/fortune')
```

注意，我们在模块名前面加上了 `.` 前缀。这是在告诉 Node 不要在 `node_modules` 目录下查

找这个模块。如果遗漏了这个前缀，就会出错。

现在，在“关于”页的路由里，可以利用模块的 `getFortune` 方法了：

```
app.get('/about', (req, res) => {
  res.render('about', { fortune: fortune.getFortune() })
})
```

如果你是跟着做下来的，就可以提交这些变更了：

```
git add -A git commit -m "Moved 'fortune cookie' into module."
```

你会发现模块是实现封装功能方便而强大的方式，它可以提升项目的总体设计和可维护性，还可以简化测试。更多信息请参考 Node 模块的官方文档。



Node 模块有时被称作 CommonJS (CJS) 模块。这是向一个更旧一些的规范致敬，Node 是从它那里得到的启发。JavaScript 语言采纳了一个正式的包机制，叫 ECMAScript 模块 (ESM)。如果你一直在使用 React 或别的新潮前端语言，可能已经很熟悉 ESM 了：它使用的是 `import` 和 `export` (而不是 `exports`、`module.exports` 和 `require`)。更多信息请查看 Axel Rauschmayer 博士的博文 “ECMAScript 6 modules: the final syntax”。

4.8 小结

现在我们已经掌握了更多关于 Git、npm 和模块的知识，接下来将讨论如何在编码中采纳一些好的 QA 实践，以创造更好的产品。

本章讲述了如下主题，希望你能记住。

- 版本控制使得软件开发过程更安全、更可预期。哪怕是对于小项目，我也建议你使用它，这样有助于培养好习惯。
- 模块化是管理复杂软件的一项重要技术。很多开发者已经通过 npm 开发了大量的模块，形成了丰富的生态系统，你也可以把自己的代码封装成模块，从而更好地组织项目。
- Node 模块（也叫 CJS）与 ECMAScript 模块使用了不同的语法。如果需要在前后端代码之间切换，可能就不得不在这两种语法之间切换。所以，对两者都熟悉是最好的。

第5章

质量保证

质量保证（QA）是一个让开发者脊背发凉的词语。这太令人遗憾了，难道你不想做出高品质的软件吗？你当然想。所以这并非目标问题，而是“政治”问题。在 Web 开发中，存在以下两种常见情形。

规模较大或资金充足的组织

这些组织通常会有一个 QA 部门。不幸的是，QA 与研发部门之间总会存在一种对立关系。最坏的情况是两个部门进入了一个团队。虽然工作目标都一样，但 QA 常常把自己的成功定义为发现更多的 bug，开发则把自己的成功定义为产生更少的 bug，这就奠定了冲突和竞争的基础。

小规模或资金有限的组织

这些组织常常没有 QA 部门，开发人员会承担 QA 和开发的双重角色。这并不荒谬，也避免了利益冲突。然而，QA 与开发是截然不同的工作，对它们感兴趣的分别是具有不同个性和才能的人。虽然肯定存在一些有 QA 头脑的开发人员，但是临近最后期限，QA 常被抛诸脑后，这样反而对项目不利。

现实中大部分职业需要多种技能，而要成为掌握所有所需技能的专家，真是越来越难了。不过，即使你不直接负责某些方面，要是拥有这些方面的能力，不仅对团队更有价值，也更有利于团队的高效运作。一名拥有 QA 技能的开发人员就是很好的例子。这两个领域的联系是如此紧密，以至于跨领域的人才十分珍贵。

把传统上由 QA 完成的工作转移到开发上，让开发人员负责 QA 也是很常见的。在这种工作模式中，有 QA 专长的软件工程师就像是开发人员的顾问，帮助他们把 QA 整合进开发流程。

无论 QA 这个角色是独立的，还是与开发整合，对开发人员来说，理解 QA 显然都是有益的。

本书不是为 QA 从业人员而写，而是为开发人员写的。因此，我的目标不是让你成为 QA 专家，只是帮助你增加一些这方面的经验。如果你的组织有 QA 专员，这些经验可以帮助你更容易地跟他们进行沟通协作；如果没有 QA 专员，有了这些经验，你也可以为项目建立起一个全面的 QA 计划。

本章将学习以下内容：

- 软件质量基本准则和良好的工作习惯
- 测试的类型（单元测试和集成测试）
- 如何使用 Jest 写单元测试
- 如何使用 Puppeteer 写集成测试
- 如何配置 ESLint 以避免常见错误
- 持续集成是什么以及从何处学起

5.1 QA计划

大体而言，开发是一个创造的过程：想象着某个东西并把它变成现实。QA 则更多的是验证，更讲究条理。因而，QA 总的来说是弄清需要做什么和确保完成了什么，很适合使用核对清单、流程图和文档这些工具来完成。可以这样说，QA 的主要活动不是测试软件本身，而是创建一个全面的、可重复执行的 QA 计划。

我建议为每个项目都创建一个 QA 计划，不管项目大小（不错，包括周末为了好玩做的小项目）。QA 计划也不需要做得多大或多精致，你可以将它写进记事本里、Word 文档里或 wiki 里。为了保证你的产品能按预期正常运行，需要采取哪些步骤？QA 计划的目标就是记录所有这些步骤。

无论 QA 计划形式如何，它都是一个动态文档，应该随时更新以适应以下变化：

- 项目的新特性
- 已有特性的变更
- 特性的移除
- 测试方法或技术的变更
- QA 计划缺陷

特别提一下最后一点：无论你的 QA 做得如何完备，缺陷都在所难免。当发现新的缺陷类型时，应该问问自己：“怎么做才能避免这类缺陷？”只要回答了这个问题，你就可以相应地调整 QA 计划，避免将来再出现同类型的缺陷。

现在你应该能感觉到 QA 要花费的精力不少了吧，那么应该投入多少精力呢？

5.2 QA：是否值得

QA 的成本可能会很高——有时会极高，那么这么做是否值得？这很难说。大多数组织会使用某种“投资回报”模型进行评估。我们花一笔钱，必然期望收回成本（多一些更好）。可是对于 QA 来说，这种关系会非常复杂。举个例子，一个已经取得地位和好评的产品可能需要比新的不知名产品花更多时间来解决质量问题。显然，没人想制造劣质产品，但是生产上承受的各种压力是很大的。产品的投产时间非常关键，有时宁可尽快把不完美的产品推向市场，也不能等数月后再推完美的产品。

在 Web 开发中，质量可以划分为 4 个维度。

用户覆盖

用户覆盖指的是产品市场渗透，即浏览网站或使用服务的人数。用户覆盖跟盈利能力直接相关：访问网站的人越多，购买产品或服务的人就越多。从开发的视角来看，搜索引擎优化（SEO）对用户覆盖的影响最大，这就是为什么我们的 QA 计划会包含 SEO。

功能实现

一旦人们开始访问你的网站或使用你的服务，功能实现的质量就会对用户留存率产生极大影响。相比实际与宣传不符的网站，实际与宣传完全相符的网站运行起来更有可能获得用户的回访。自动化测试适用于评估功能实现。

易用性

功能实现关注的是功能的正确性，易用性评估的则是人机交互（HCI）方面。要评估的基本问题就是“这个功能以这种方式来呈现，对目标用户有用吗？”这常常可以转换为“这易用吗？”不过，对易用性的追求往往有悖于软件的灵活性或强大功能：程序员觉得容易的东西跟无技术背景的消费者觉得容易的东西也许完全不同。换句话说，在评估易用性时必须考虑目标用户。既然易用性衡量的基本输入来自用户，那么它通常是可以自动化测试的。不过，QA 计划应该把用户测试包含进去。

审美

审美在这 4 个维度当中主观性最大，因而跟开发的关系也就最小。尽管开发很少关注网站的审美，但是对审美方面进行例行的评审应该是 QA 计划的一部分。把你的网站展示给目标用户，看看用户是否有过时之感，或者有没有唤起想要的用户回应。要记住审美变化有时（随着时代改变审美标准也在变化）也是特定于用户群体的（能吸引这个用户群体但可能对另一用户群体毫无吸引力）。

4 个维度都应该在 QA 计划中有所体现，不过在开发期间，只有功能测试和 SEO 可以进行自动化，因此本章将专注于这些方面。

5.3 逻辑与表示

大体而言，网站有两个层面：逻辑（常被叫作业务逻辑，我不这么叫是因为这种叫法多少有些商业意味）和表示。你可以认为网站的逻辑只存在于人的头脑里。比如说，在草地鹨旅游的场景里，可能会有一条规则：客户必须拥有合法驾照才能租借踏板车。这是一条基于数据的简单规则，说明用户拥有合法的驾照才能完成每一次踏板车预订。而表示跟这条规则是分离的，可能它只是体现在订单页中最终表单的一个复选框，或者是用户必须提供而由网站来验证的合法驾照号码。区分逻辑与表示很重要，因为在逻辑层面，事情应该尽可能清楚和简单；而在表示层面，可以根据需要选择复杂或简单。表示会受易用性和美学方面的制约，逻辑则不会。

你应该尽可能地把逻辑和表示划分清楚。对此有很多种方式，而本书选择把逻辑封装进 JavaScript 模块。表示则是一个组合，包含 HTML、CSS、多媒体、JavaScript，以及像 React、Vue 或 Angular 这样的前端框架。

5.4 测试类型

本书考虑的测试类型可以分为两大类：单元测试和集成测试（我把系统测试也看作一种集成测试）。单元测试相当精细，需要测试单个组件确保其功能正常，集成测试则测试多个组件的交互，甚至会测试整个系统。

一般来说，单元测试对逻辑层面的测试更有用，也更合适。而集成测试对逻辑和表示两个层面都有用。

5.5 QA技术

本书将使用以下技术和软件来完成全面测试。

单元测试

单元测试覆盖应用的最小功能单元，通常是单个函数。单元测试基本都是由开发人员而不是 QA 专员（尽管 QA 专员应该有权评估单元测试的质量和覆盖率）所写。本书中将使用 Jest 做单元测试。

集成测试

集成测试覆盖更大的功能单元，通常涉及应用的多个部分（函数、模块、子系统，等等）。既然我们构建的是 Web 应用，那么“终极”集成测试就是在浏览器上渲染应用、操作应用，并验证应用的行为是否符合预期。这些测试建立和维护起来通常要更复杂，既然本书的重点不是 QA，那么我们就只提供一个简单的例子，这个例子中用到了 Puppeteer 和 Jest。

Linting

Linting 并非用来寻找错误，而是用来寻找潜在错误。Linting 的一般含义是识别出可能的错误或脆弱的结构，而这些脆弱结构会导致将来的错误。我们将使用 ESLint 来做 Linting。

先从测试框架 Jest 开始（单元测试和集成测试都用它来运行）。

5.6 安装和配置Jest

我曾有点儿犹豫本书应该选择哪个测试框架。虽然 Jest 最初是作为测试 React 应用的框架出现的（现在仍是明智的选择），但 Jest 并不限于 React，它是优秀的通用测试框架。当然，它也不是唯一的测试框架：Mocha、Jasmine、Ava 和 Tape 都是很好的选择。

最终我选择了 Jest，因为它提供了最好的综合体验（此观点得到了“JavaScript 状况 2018”调查的支持，Jest 的得分很高）。尽管如此，因为测试框架存在大量相似之处，所以你可以把学到的东西应用到最喜欢的那个上面。

要安装 Jest，只需在项目目录下运行以下命令：

```
npm install --save-dev jest
```

（注意，这里用了 `--save-dev`，这是在告诉 npm 它是开发依赖，应用正常运行并不需要它。在 `package.json` 文件中，它被列在 `devDependencies` 而不是 `dependencies` 一节。）

接着，需要一种方法来运行 Jest（可以运行项目中的任何测试）。传统做法是在 `package.json` 中配置一个脚本。编辑 `package.json`（版本库的 `ch05/package.json`），修改 `scripts` 属性（如果还不存在，就先增加）：

```
"scripts": {  
  "test": "jest"  
},
```

现在要运行项目中所有的测试，只需键入以下命令：

```
npm test
```

如果现在就运行，很可能得到错误，提示说还没有配置任何测试……这是因为还没有加入任何测试。好吧，先来写几个单元测试。



正常来说，如果把一个脚本加入 `package.json` 文件，就要用 `npm run` 来运行这个脚本。例如，如果加入了一个脚本 `foo`，就要用 `npm run foo` 来运行它。`test` 脚本太常见了，即使只是键入 `npm test`，npm 也知道是要运行它。

5.7 单元测试

现在可以把注意力转到单元测试上了。既然单元测试的重点是隔离单个函数或组件，那么就先来学习模拟（mocking），它是实现这种隔离的重要技术。

5.7.1 模拟

如何写出“可测试”的代码是我们经常会遇到的一个挑战。一般来说，试图做太多事情或者设想了太多依赖的代码，比起更专注、设想更少依赖或没有依赖的代码要更难于测试。

每当代码中多了一个依赖，为了保证测试的有效性，就需要多模拟一处。例如，我们主要的依赖是 Express，它已经经过了完整的测试，所以不必（也不想）测试 Express 本身，只需测试使用它的方式。要确定是否在正确地使用 Express，唯一的方法是模拟 Express 本身。

目前，我们的路由（主页、“关于”页、404 页和 500 页）是很难测试的，因为它们设想了 3 个有关 Express 的依赖：Express 的 app 对象（所以才能调用 `app.get`）、request 对象以及 response 对象。幸运的是，要消除对 app 对象的依赖十分容易（request 和 response 对象要难一点儿，后面会详细介绍）。而且，我们并没有使用过多来自 response 对象的功能（只使用了 `render` 方法），因此也很容易模拟，一会儿就会看到。

5.7.2 为可测试性而重构应用

我们的应用还没有太多代码可以测试。目前才只是加了几个路由处理函数和一个 `getFortune` 函数。

为了让应用更易于测试，我们准备把实际的路由处理函数提取到它们自己的库中。创建文件 `lib/handlers.js`（版本库的 `ch05/lib/handlers.js`）：

```
const fortune = require('./fortune')

exports.home = (req, res) => res.render('home')

exports.about = (req, res) =>
  res.render('about', { fortune: fortune.getFortune() })

exports.notFound = (req, res) => res.render('404')

exports.serverError = (err, req, res, next) => res.render('500')
```

现在可以重写 `meadowlark.js` 应用文件了，以便使用下面这些处理函数（版本库的 `ch05/meadowlark.js`）：

```
// 一般在文件顶部
const handlers = require('./lib/handlers')
```

```
app.get('/', handlers.home)

app.get('/about', handlers.about)

// 定制404页
app.use(handlers.notFound)

// 定制500页
app.use(handlers.serverError)
```

现在要测试这些处理函数就更容易了，毕竟它们只是接收 `request` 和 `response` 对象的函数。我们需要验证是否在正确使用这些对象。

5.7.3 写第一个测试

有多种方式可以识别出 Jest 的测试文件，最常见的有两种，一种是把测试文件放到 `_tests_` (test 前后各有两条下划线) 子目录下面，另一种是使用 `.test.js` 作为测试文件的扩展名。我个人喜欢组合使用这两种方式，因为二者各有千秋。把测试放到 `_tests_` 目录下面可以避免将源代码目录弄乱（否则，在源代码目录下，什么东西看起来都好像有两个……每个 `foo.js` 文件都有一个 `foo.test.js`）。使用 `.test.js` 作为扩展名则意味着，如果向编辑器中的一连串标签页看过去，就可以一眼看出哪个是测试文件，哪个是源代码文件。

那么，创建一个名为 `lib/_tests_/handlers.test.js` 的文件（版本库的 `ch05/lib/_tests_/handlers.test.js`）：

```
const handlers = require('../handlers')

test('home page renders', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.home(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('home')
})
```

如果你刚接触测试，很可能会觉得以上代码相当怪异，所以我们把它分解一下。

首先，导入想要测试的代码（在这里就是路由处理函数）。然后每个测试都有一个描述文本，我们想尽量描述清楚所测试的是什么。这里是想确认主页是否得到了渲染。

要调用渲染，就需要 `request` 和 `response` 对象。如果想模拟所有 `request` 和 `response` 对象，就得写一整周的代码，不过好在我们不太需要这两个对象中的东西。我们知道这里不需要 `request` 对象中的任何东西（所以只使用了一个空对象），只需要 `response` 对象中的一个 `render` 方法。注意我们是如何构造 `render` 函数的：只需调用一个名为 `jest.fn()` 的 Jest 方法。之后会创建一个通用的模拟函数，这个函数会记录每次对它自己的调用。

最后，到了测试的重要环节：断言。前面做了那么多，终于调用了要测试的代码，可是如何确认它是否做了本职工作呢？

在这个例子中，要测试的代码应该做的是以字符串 `home` 作为参数调用 `response` 对象的 `render` 方法。Jest 的模拟函数每次会记录对它的调用，所以我们需要做的就是验证它只被调用了一次（要是被调用了两次，就很可能是有问题），这就是第一个 `expect` 所做的，而且被调用的时候是以 `home` 作为第一个参数（第一个数组索引表示是哪次调用，第二个表示是哪个参数）。



每次对代码做出一点儿改动就重新运行，这样不停地重复很乏味。幸运的是，大多数测试框架有一个“watch”模式，可以不间断地监控源代码和测试的变更，然后自动重新运行。要按这个模式运行测试，需键入 `npm test -- --watch`（额外的双横杠必不可少，这是让 `npm` 知道 `--watch` 是传给 Jest 的参数）。

可以试试修改 `home` 处理函数，让它渲染主页以外的东西，这样就会看到测试失败，于是就发现了一个 bug。

现在可以加上其他的路由测试了：

```
test('about page renders with fortune', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.about(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('about')
  expect(res.render.mock.calls[0][1])
    .toEqual(expect.objectContaining({
      fortune: expect.stringMatching(/\W/),
    }))
})
}

test('404 handler renders', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.notFound(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('404')
})

test('500 handler renders', () => {
  const err = new Error('some error')
  const req = {}
  const res = { render: jest.fn() }
  const next = jest.fn()
  handlers.serverError(err, req, res, next)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('500')
})
```

注意，“关于”页的功能和服务器错误页的测试需要比别的页做更多的工作。“关于”页调用 `render` 函数时传入了 `fortune`（幸运饼），因此我们增加了一个断言，即 `render` 函数会获得一个 `fortune`，而 `fortune` 是至少包含一个字符的字符串。完整地阐述 Jest 的功能以及它的 `expect` 方法超出了本书的范畴，你可以从 Jest 的主页找到详尽的文档。注意，服务器错误处理函数接收 4 个而不是 2 个参数，所以需要提供更多的模拟对象。

5.7.4 测试维护

你可能意识到了测试不是一劳永逸的事。例如，如果因为某些原因你重命名了“home”视图，那么测试就会失败，除了修复代码之外，还需要修正这个测试。

因此，团队要花很多精力考虑待测试目标以及测试内容，并设置切合实际的断言。例如，我们不需要检查“关于”页是否获得了一个 `fortune` 变量，如果丢弃“幸运饼”这个特性，就不需要修正这个测试了。

而且，你的代码究竟要测试到什么程度，我也没法提供太多的建议。不过我认为，航空设备或医疗设备的测试代码和市场营销网站的测试代码应该有截然不同的标准。

我可以提供给你的是如何回答“我的代码有多少被测试到了？”这个问题，答案就是接下来要讨论的代码覆盖率。

5.7.5 代码覆盖率

对于你的代码有多少被测试到了这个问题，代码覆盖率提供了一个量化的答案，但是就像编程中的很多问题一样，其实是没有简单答案的。

Jest 提供了一些代码覆盖率自动化的分析，这对我们很有帮助。要查看有多少代码被测试到了，可以运行以下命令：

```
npm test --coverage
```

如果你照着做下来，对于在 `lib` 下的文件，就应该会看到一连串绿色的“100%”覆盖率数值，非常令人宽慰。Jest 会按代码的语句（Stmt）、分支、函数（Func）和行来报告覆盖率百分比。

这里的语句指的是 JavaScript 语句，比如每个表达式、控制流结构等。需要注意的是，可以达到 100% 的代码行覆盖率，但不能达到 100% 的语句覆盖率，因为在 JavaScript 中可以把多条语句放到同一行里。分支覆盖率指的是控制流语句，比如 `if-else`。如果你有一个 `if-else` 语句而只测试到 `if` 部分，那么对于这条语句就只有 50% 的分支覆盖率。

可能你注意到了，`meadowlark.js` 并未达到 100% 的覆盖率，但这未必是个问题。如果看一下重构后的 `meadowlark.js` 文件，就会发现现在里面大部分只是简单的配置……只是把这些

东西拼接了起来。给 Express 配置有关的中间件，然后启动服务。不用说要有效测试这些代码有多困难，就说不必测试也是合理的理由，因为它只是把经过良好测试的代码装配起来了。

甚至可以说，目前写的测试没多大用处，因为它们只是验证是否正确地配置了 Express。

这次同样我也没法提供简单的答案。究竟要写多少测试？归根结底，你正在构建的应用类型、你的经验层次，以及你团队的规模和配置，对此都会有很大的影响。我的建议是，与其不够，还不如多写一点儿，这样随着经验的增长，就会知道多少才是“刚刚好”。

熵功能测试

熵功能（带随机性的功能）测试有它自己的难点。我们可以为幸运饼的生成器再增加一个测试，测试它的确返回了一个随机的幸运饼。可是如何知道某个东西是随机的呢？一个办法是获取大量的幸运饼（响应），比如说 1000 个，然后衡量这些响应的分布。如果这个生成器函数大体是随机的，那就说明没有哪个响应特别突出。这个办法的不足之处在于它不是确定的。有可能（虽然实际不太可能）取到某个幸运饼的次数是取到任何其他幸运饼次数的 10 倍，如果是这样，测试就会失败（取决于对“随机”应该达到的样本容量阈值），然而这或许并不真的就说明所测试的系统失败了，这只是熵系统测试的一种正常结果。在幸运饼生成器的例子中，生成 50 个幸运饼并期望其中至少有 3 种不同的饼，可以认为是合理的。反过来说，如果我们是为某科学仿真或安全组件开发一个随机源，就很可能希望进行更细致的测试。所以，测试熵功能是很困难的，需要更多的思考。

5.8 集成测试

当前我们的应用还没有什么有意思的东西可以测试，只有几个页面，彼此之间也没有交互。所以在写集成测试之前，可以先加一点儿测试的功能。本着能简则简的原则，我们加的这个功能只是一个链接，允许你从主页转到“关于”页。这几乎不能再简单了。而且，虽然在用户看来很简单，但这个链接也是一个真正的集成测试，因为它不仅连接了两个 Express 路由处理函数，还涉及了 HTML 和 DOM 交互（用户点击链接然后导航到结果页面）。在 `views/home.handlebars` 中增加一个链接：

```
<p>Questions? Checkout our  
<a href="/about" data-test-id="about">About Us</a> page!</p>
```

你可能会对 `data-test-id` 属性感到奇怪。测试要进行下去，就需要用某种方法来识别出这个链接，然后才能（虚拟地）点击它。本来可以使用一个 CSS 类，但我更希望 CSS 类只用于样式而数据属性用于自动化处理。也可以搜索文本“About Us”，但这样做 DOM 搜索

会又贵又不好用。还可以查出 href 参数，这也行得通（但那样的话测试则更不容易失败，达不到我们的教学目标）。

现在可以先运行应用，在开始自动化测试之前手动验证一下刚加的链接的确按预期工作了。

在安装 Puppeteer 以及写集成测试之前，需要修改一下应用，让其可以导入为一个模块（当前它只能直接运行）。Node 的做法有点儿不好理解：在 meadowlark.js 的底部，把对 app.listen 的调用取代为以下命令：

```
if(require.main === module) {
  app.listen(port, () => {
    console.log(`Express started on http://localhost:${port} +
      '; press Ctrl-C to terminate.'`)
  })
} else {
  module.exports = app
}
```

我就不做技术性的解释了，因为比较乏味。如果你好奇，仔细阅读一下 Node 的模块文档就清楚了。很重要的一点是，如果用 Node 直接运行一个 JavaScript 文件，require.main 会等于全局变量 module；反过来，如果二者不相等，则说明 require.main 是从另一个模块导入的。

现在已经清除了障碍，可以安装 Puppeteer 了。本质上，Puppeteer 是一个可控制的无头 (headless) 版 Chrome（无头只是表示这个浏览器不实际在屏幕上渲染用户界面也能够正常运行）。运行以下命令安装 Puppeteer：

```
npm install --save-dev puppeteer
```

也可以安装一个小工具以用来寻找没被占用的端口，这样就不会因为应用在指定端口无法运行而导致很多测试错误了：

```
npm install --save-dev portfinder
```

现在可以写一个集成测试，让它实现以下功能。

1. 在未被占用的端口上运行应用。
2. 启动一个无头的 Chrome 浏览器并打开一个页面。
3. 导航到应用主页。
4. 找到带 data-test-id="about" 的链接并点击。
5. 等待导航完成。
6. 验证我们到达了“关于”页。

创建一个名为 integration-tests 的目录（如果你喜欢，命名为别的也可以），在此目录下创建一个文件，命名为 basic-navigation.test.js（版本库的 ch05/integration-tests/basic-navigation.test.js）：

```

const portfinder = require('portfinder')
const puppeteer = require('puppeteer')

const app = require('../meadowlark.js')

let server = null
let port = null

beforeEach(async () => {
  port = await portfinder.getPortPromise()
  server = app.listen(port)
})

afterEach(() => {
  server.close()
})

test('home page links to about page', async () => {
  const browser = await puppeteer.launch()
  const page = await browser.newPage()
  await page.goto(`http://localhost:${port}`)
  await Promise.all([
    page.waitForNavigation(),
    page.click('[data-test-id="about"]'),
  ])
  expect(page.url()).toBe(`http://localhost:${port}/about`)
  await browser.close()
})

```

我们把 Jest 的 `beforeEach` 和 `afterEach` 钩子函数用于在每个测试前后启动和关闭服务（目前只有一个测试，如果增加了更多的测试，这样做则很有意义）。也可以使用 `beforeAll` 和 `afterAll`，这样就不用逐个测试地启动和清理服务器了，从而可以加速测试，但代价是每个测试将不再拥有“干净”的环境。意思就是说，如果你的一个测试做了影响后续测试结果的变更，那么你就引入了难以维护的依赖关系。

实际的测试使用了 Puppeteer 的 API，它包含了大量的 DOM 查询功能。需要注意的是，这里的所有东西都是异步的，而且我们使用 `await` 使得测试更易读、易写（所有的 Puppeteer API 调用都返回一个 promise¹。依据 Puppeteer 的文档，为避免竞争条件，我们把导航和点击包装在一起放在 `Promise.all` 调用里面。

Puppeteer 的 API 非常丰富，远不是本书能够覆盖的。幸运的是，它的文档说得很清晰。

测试是确保产品质量的重要后盾，但它不是你唯一的武器。Linting 可以帮助你从一开始就避免常见的错误。

注 1：如果对 promise 不熟悉，推荐阅读 Tamas Piros 的文章“Async/await in Node.js”。

5.9 Linting

拥有一个好的 linter 就像有了第二双眼睛：它可以发现人眼难以识别或人脑处理不过来的小问题。最初的 JavaScript linter 是 Douglas Crockford 的 JSLint。2011 年，Anton Kovalyov 创建了 JSLint 的分支——JSHint。但 Kovalyov 发现 JSLint 变得越来越不中立，于是他想创建一个更可定制的、社区开发的 JavaScript linter。所以在 JSHint 之后，又出现了 Nicholas Zakas 的 ESLint，它已成为最流行的一种选择（它在“JavaScript 状况 2017”调查中获得了压倒性胜利）。除了被广泛使用以外，ESLint 看起来也是维护最活跃的 Linter，比起 JSHint，我更喜欢它灵活的配置，因此推荐使用 ESLint。

ESLint 可以按项目安装或全局安装。为了防止不经意间破坏了什么，我尽量避免全局安装（例如，如果我全局安装了 ESLint 而又比较频繁地更新它，因为存在破坏性的更新，老的项目可能不再成功地 lint，那么就需要额外做更新项目的工作）。

运行以下命令将 ESLint 安装到项目中：

```
npm install --save-dev eslint
```

ESLint 需要一个配置文件来告诉它要应用什么规则。因为从头创建配置文件很耗费时间，所以 ESLint 提供了创建配置文件的辅助程序。从项目根目录运行以下命令：

```
./node_modules/.bin/eslint --init
```



如果全局安装了 ESLint，则可以只使用 `eslint --init`。要直接运行项目的本地程序，必须基于有点儿奇怪的路径 `./node_modules/.bin`。一会儿就会看到，如果把程序配置到 `package.json` 文件的 `scripts` 这一节，就不需要那样做；如果直接运行比较频繁，则推荐配置到 `scripts`。不过，创建 ESLint 配置文件这件事，每个项目只需要做一次。

ESLint 会询问你一些问题。对其中的大部分，选择默认值是没问题的，但有几个需要多加注意。

项目使用的是哪种类型的模块？（What type of modules does your project use?）

既然使用的是 Node（跟要运行于浏览器的代码相对），那么应该会选择“CommonJS (require(exports))”。可能你的项目中也包含客户端的 JavaScript，但那样的话需要一份独立的 Lint 配置。划分成两个项目是最简单的方法，但同一个项目里有多份 ESLint 配置也是可能的。更多信息请参考 ESLint 官方文档。

项目使用的是哪个框架？（Which framework does your project use?）

除非在列表中看到了 Express（写到这里时我没有看见），否则选择“None of these.”。

你的代码运行在哪里？（Where does your code run?）

选择“Node”。

ESLint 已经设置好了，我们需要一个便利的方式来运行它。将以下内容添加到 package.json 的 scripts 一节：

```
"lint": "eslint meadowlark.js lib"
```

注意，需要明确告诉 ESLint，想要对哪些文件和目录进行 lint。这也是把所有的源代码都归拢到一个目录（通常是 src）下的理由。

现在请振作精神，运行以下命令：

```
npm run lint
```

你可能会看到很多令人不快的错误——这通常是在第一次运行 ESLint 时发生的。不过，如果照着 Jest 测试做下来，则会出现一些跟 Jest 有关的错误，它们看起来是这样的：

```
3:1  error  'test' is not defined    no-undef
5:25 error  'jest' is not defined   no-undef
7:3   error  'expect' is not defined no-undef
8:3   error  'expect' is not defined no-undef
11:1  error  'test' is not defined    no-undef
13:25 error  'jest' is not defined   no-undef
15:3   error  'expect' is not defined no-undef
```

ESLint 不喜欢未识别的全局变量（相当合理）。这些全局变量是 Jest 注入的（比较显眼的有 test、describe、jest 和 expect）。这个问题很容易修正。在项目目录下，打开 .eslintrc.js 文件（这是 ESLint 配置）。在 env 这一节，添加以下命令：

```
"jest": true,
```

现在如果再次运行 npm run lint，看到的错误应该少多了。

那么其余的错误要怎么处理呢？此处我可以给出一些方向，但无法给出专门的指导。大体而言，一个 Linting 错误的根源为以下三者之一。

- 这是一个不符合规则的细节问题，你应该修正它。这种错误并不总是显而易见，可能需要参考 ESLint 文档来了解特定的错误。
- 这是一条你不认可的规则，可以直接禁用它。ESLint 中的很多规则只代表个人的态度。一会儿我会演示如何禁用一条规则。
- 你认可这条规则，但在某些特定情形下，这条规则不太可行或修正它的成本比较高。这时候，可以只对文件中的某些行禁用这条规则，一会儿我们也会看到一个例子。

如果你是一直跟着做下来的，现在应该能看到以下错误：

```
/Users/ethan/wdne2e-companion/ch05/meadowlark.js
27:5  error  Unexpected console statement  no-console

/Users/ethan/wdne2e-companion/ch05/lib/handlers.js
10:39  error  'next' is defined but never used  no-unused-vars
```

ESLint 默认不使用控制台记录日志，因为这未必是输出应用的好方式。控制台会使日志输出杂乱而矛盾，而且可能会因为运行方式不当让日志输出被覆盖。不过对我们来说，控制台不会影响到使用，所以希望禁用这条规则。打开 `.eslintrc2` 文件，找到 `rules` 一节（如果没有，在导出对象的最上一层创建一个），然后添加以下规则：

```
"rules": {
  "no-console": "off",
},
```

现在如果再次运行 `npm run lint`，会看到那条错误消失了。而下一条错误会更难处理一些……

打开 `lib/handlers.js`，考虑下面这行错误：

```
exports.serverError = (err, req, res, next) => res.render('500')
```

ESLint 说得没错，我们提供了 `next` 参数，却没有使用它做任何事情（同样没有使用 `err` 和 `req`，但由于 JavaScript 对待函数参数的方式，必须有参数占据旁边的位置才能取到 `res`，而 `res` 是我们要用的）。

或许你会忍不住想要移除 `next` 参数，也会思考：“有什么损害吗？”确实，运行时没有错误，linter 也会很高兴……不过此时一个难以觉察的损害已经发生了：之前定制的错误处理函数不再起作用了！（如果你想亲眼看看，可以在某个路由中抛出一个异常，并尝试访问它。然后把 `serverError` 处理函数的 `next` 参数移除，再尝试访问。）

Express 在这里做的有些微妙：通过传入的实际参数的个数，它认出了这个语句就是想用作错误处理函数的。如果没有 `next` 参数，那么不管你是否使用，Express 都不再把它认作错误处理函数。



不可否认，在错误处理函数这件事上，Express 团队的做法很“聪明”，但聪明的代码往往是令人糊涂、容易出问题、难以捉摸的。尽管我非常喜欢 Express，但还是认为 Express 团队的这个选择错了：我认为当初应该找一种更自然且更明确的方式来指定错误处理函数。

我们需要这个错误处理函数，从而无法修改这行代码，而且也认可这条规则，不想禁用。虽然可以留着这个错误，但是错误会越积越多，成为一件恼人的事，最终违背了使用 linter 的初衷。幸运的是，可以仅针对那一行代码禁用这条规则，从而修正这个问题。编辑

注 2：根据配置文件的格式，也可能是 `.eslintrc.json` 和 `.eslintrc.js`。——译者注

lib/handlers.js，在错误处理函数前后加上一些东西：

```
// Express根据4个参数来认出它是错误处理函数，  
// 因此我们要禁用ESLint的no-unused-vars规则  
/* eslint-disable no-unused-vars */  
exports.serverError = (err, req, res, next) => res.render('500')  
/* eslint-enable no-unused-vars */
```

Linting 起初会给你带来一些小麻烦——好像总在不停地找碴儿。当然，对于不适合你的规则，可以放心地禁用它。到最后，随着逐渐学会避免那些（Linting 本来要捕获的）常见的人为失误，你的麻烦就会越来越少。

测试和 Linting 无疑都很有用，但是如果从不使用，任何工具都是毫无价值的。花时间和精力去编写单元测试并设置 Linting 似乎很疯狂，但我看到过这种情况，尤其是在压力很大的时候。幸运的是，有一个办法可以保证你不会忘记这些有用的工具，那就是持续集成。

5.10 持续集成

这里我向你提出另一个极其有用的 QA 概念：持续集成（CI）。在团队工作中，它显得尤其重要，即便你是独自工作，它也可以提供一些很有帮助的准则。

从根本上来说，每当你向源代码的版本库提交了代码，CI 就会运行某些或全部测试（你可以控制要应用到版本库的哪些分支）。如果所有这些测试都通过了，一般不会有什么事（你可能会收到一封邮件说“干得不错”，不过这取决于你的 CI 是如何配置的）。

反过来说，如果测试失败了，结果通常就会……人尽皆知了。同样也是取决于你的 CI 如何配置，但通常情况是整个团队都会收到一封邮件，说你们把构建（build）破坏了。如果集成测试的负责人是个“虐待狂”，说不定你的老板也会在收件人列表里。在有些团队中，如果有人把构建破坏了，就会触发灯光和声音警报。某个特别有创造力的团队甚至弄了一个自动控制的泡沫软弹，每当构建被破坏时，软弹就会朝那个惹事的开发人员发射。这是巨大的鞭策，驱使着你在提交代码之前先运行一遍 QA 工具链。

关于 CI 服务器的安装和配置超出了本书的范畴，但作为 QA 的一章，没有提及 CI 服务器是不完整的。

当前，最适合 Node 项目的 CI 服务器是 Travis CI。Travis CI 是一个托管方案，这一点可能很吸引人（省得你从头搭建自己的 CI 服务器了）。如果你在使用 GitHub，那么它也有非常好的集成测试支持。CircleCI 则是另一个选择。

如果你是独自做项目，CI 服务器可能用途有限，但如果你在一个团队或是一个开源项目中工作，强烈建议你研究一下，并把项目的 CI 搭建起来。

5.11 小结

本章涉及很多方面，但我认为这些都是很基本、很实用的技能，跟使用什么开发框架无关。JavaScript 的生态极其庞大，令人眼花缭乱，如果你是一名新人，会很难看清楚应当从哪里入手。希望本章除能为你指出正确的方向。

对这些工具有了一些经验之后，我们将把注意力转到 Node 和 Express 对象的一些基本原理上来。先从 request 和 response 对象开始，毕竟它们可以囊括 Express 应用中所发生的一切。

第 6 章

request和response对象

本章将学习 `request` 和 `response` 对象的重要内容。在 Express 应用中发生的一切，很大程度上由它们开始，也由它们结束。如果使用 Express 构建一个 Web 服务器，则大部分要做的事情会始于一个 `request` 对象，而止于一个 `response` 对象。

这两个对象来自 Node，Express 则对它们做了扩展。在深入探索这些对象能够带来什么之前，先建立一点儿背景知识：客户端（通常指浏览器）是怎样向服务器请求一个页面，这个页面又是怎样返回的。

6.1 URL的各个组成部分

我们每天都会看见各种 URL，但很少会去思考它们是怎么组成的。下面列举了一个 URL 的各个组成部分。

协议（protocol）

协议决定了请求将怎样传输。本书只会涉及 `http` 和 `https`。其他常见协议包括 `file` 和 `ftp`。

主机（host）

主机用来标识服务器。如果是在你的计算机（`localhost`）或本地网络上，那么服务器可以用一个名字或数字 IP 地址简单地标识。在互联网中，主机会以顶级域名（TLD）结尾，比如 `.com` 或 `.net`。另外，可能还会有子域名。子域名放在主机名的前面，怎么命名都可以，比如 `www` 就是常见的子域名。子域名是可选添加的。

端口 (port)

每个服务器都有一套用数字表示的端口。有些端口很特别，比如 80 和 443。如果省去端口，HTTP 的端口就默认为 80，而 HTTPS 的端口就默认为 433。一般来说，如果不使用 80 和 433 端口，就应该使用大于 1023 的端口¹。使用最多的是数字易记的端口，比如 3000、8080 和 8088。如果给定一个端口，则只有一个服务器应用可以关联，所以，虽然可以选择的端口非常多，但如果使用的是常用端口，就可能会冲突，不得不另换一个。

路径 (path)

URL 的各个部分中，路径通常是应用需要最先考虑的部分（基于协议、主机和端口做决策是可以的，但实践效果不好）。路径应当用于唯一标识应用中的页面或其他资源。

查询串 (querystring)

查询串是可选的，是名 / 值对的集合。查询串以一个问号 (?) 开始，名 / 值对用和号 (&) 分隔。名和值两部分都应当是由 URL 编码过的。JavaScript 提供了一个内建函数来实现 URL 编码：`encodeURIComponent`。例如，空格将被取代为加号 (+)。其他特别的字符会被取代为数字形式的字符引用。查询串有时候也被称为搜索串或搜索。

片段 (fragment)

片段（或散列）根本不会传给服务器，它只限于浏览器使用。有些单页应用使用片段来控制应用导航。起初，片段的目的只是让浏览器显示文档的指定部分，而此部分要通过一个锚标签来标记（例如：``）。

6.2 HTTP请求方法

HTTP 协议定义了一批客户端跟服务器通信的请求方法（也常被叫作 HTTP 动词）。最常用的方法无疑是 GET 和 POST。

在浏览器中输入一个 URL（或点击一个链接），浏览器就会向服务器发出一个 HTTP 的 GET 请求，传给服务器的重要信息就是 URL 的路径和查询串。应用要确定如何响应，依据的是请求方法、路径和查询串的组合。

对一个网站来说，大部分页面响应的是 GET 请求。POST 请求通常是用来向服务器发回信息的（比如表单处理）。对一个 POST 请求来说，在服务器处理完请求中包含的信息（比如一个表单）之后，返回给对应的 GET 请求同样的 HTML 是相当常见的。跟服务器通信时，浏览器主要使用 GET 和 POST 方法。不过，应用发出的 Ajax 请求可以使用任何 HTTP 动词。例如，有一个名为 DELETE 的 HTTP 方法对实现删除操作的 API 很有用。

使用 Node 和 Express，完全可以决定自己要对哪些 HTTP 方法做响应。在 Express 中，通常会为不同的方法分别写处理函数。

注 1：端口 0-1023 是“周知端口”，为“常见服务”保留。

6.3 请求头

当跳转到某个页面时，URL 不是唯一被传到服务器的信息。每次访问某网站时，浏览器都会发送很多“不可见”的信息。我说的不是让人担忧的个人信息（不过要是浏览器感染了病毒的话，这可能会发生）。浏览器会告诉服务器，它希望接收的页面是哪种语言的（例如，如果你的 Chrome 是在西班牙下载的，Chrome 就会请求所访问页面的西班牙语版，但前提是有个这个版本）。浏览器还会发送关于用户代理（浏览器、操作系统和硬件）的信息以及其他几项信息。所有这些信息都是以请求头形式发送的，可以通过 `request` 对象的 `headers` 属性访问。如果好奇想看看浏览器发送的这些信息，可以创建一个简单的 Express 路由用以显示（版本库的 `ch06/00-echo-headers.js`）：

```
app.get('/headers', (req, res) => {
  res.type('text/plain')
  const headers = Object.entries(req.headers)
    .map(([key, value]) => `${key}: ${value}`)
  res.send(headers.join('\n'))
})
```

6.4 响应头

正如浏览器以请求头形式向服务器发送隐藏的信息，服务器响应的时候也会发回一些浏览器不需要渲染或显示的信息。响应头中的信息一般包含元数据或服务器信息。我们已经见到了 `Content-Type` 头，它可以告诉浏览器要传输的内容是什么类型的（HTML、图片、CSS、JavaScript，等等）。要知道，浏览器更看重的是 `Content-Type` 头，而不管 URL 路径是什么。因此你可以对路径 `/image.jpg` 响应一个 HTML，或者对路径 `/text.html` 响应一个图片（这么做没有什么理由，重要的是要理解路径是抽象的，而浏览器是根据 `Content-Type` 来确定如何渲染内容的）。除了 `Content-Type`，部分头信息可以指示响应的内容是否被压缩了，使用的编码是什么。响应头中还可以包含给浏览器的提示信息，告诉浏览器当前资源可以缓存多久。对于网站优化来说，缓存是需要着重考虑的一个方面，这些将在第 17 章中详细讨论。

响应头中包含一些服务器的信息也是很常见的，例如用以指示服务器的类型，有时甚至还有操作系统的细节。返回服务器信息的负面作用，就是给黑客提供了危害网站的一个切入点。安全意识非常强的服务器通常不会提供这条信息，甚至会提供虚假的信息。Express 默认通过响应头 `X-Powered-By` 提供服务器信息，禁用它很容易（版本库的 `ch06/01-disable-x-powered-by.js`）：

```
app.disable('x-powered-by')
```

如果你想看响应头，可以从浏览器的开发者工具中看到。例如，以下步骤展示了从 Chrome 中查看响应头。

1. 打开 JavaScript 控制台。
2. 点击“Network”标签页。
3. 重新加载页面。
4. 从请求列表中选择当前 HTML（应该是第一个）。
5. 点击“Headers”标签页，就可以看到所有的响应头。

6.5 互联网媒体类型

`Content-Type` 这个 HTTP 头至关重要，要是没有它，客户端就只能费力地去猜要怎样渲染内容。`Content-Type` 头的格式是一个互联网媒体类型，包含类型、子类型以及可选参数。例如，`text/html; charset=UTF-8` 指定了类型为“text”，子类型为“html”，字符编码为 UTF-8。“互联网号码分配局”（IANA）维护着互联网媒体类型的官方列表。很多时候，你会发现“内容类型”“互联网媒体类型”和“MIME 类型”几个词语被混用。MIME（多用途互联网邮件扩展）曾是互联网媒体类型的前身，如今，二者等价。

6.6 请求的Body

除了请求头，请求还可以包含一个 `body`（正如响应的 `body` 是要返回的实际内容一样）。正常的 GET 请求是不包含 `body` 的，但 POST 请求通常会包含。对于 POST 的 `body` 来说，最常用的媒体类型是 `application/x-www-form-urlencoded`，它只是以“&”分隔的经过编码的名 / 值对（基本上跟查询串的格式一样）。如果 POST 请求需要支持文件上传，那么媒体类型就是 `multipart/form-data`，这是一种更复杂的格式。最后，Ajax 请求的 `body` 可以使用 `application/json`。第 8 章将学习更多关于请求的 `body` 的内容。

6.7 request对象

`request` 对象（`request` 是作为请求处理函数的第一个参数被传进来的，意味着你想怎么命名都可以，常见的命名是 `req` 或 `request`）的生命周期是从作为 Node 核心对象 `http.IncomingMessage` 的一个实例开始的。Express 给这个对象增加了更多功能。我们来看一下 `request` 对象最有用的属性和方法（除了 `req.headers` 和 `req.` 是来自 Node 以外，其他的都是 Express 增加的）。

`req.params`

包含命名的路由参数的一个数组。第 14 章将进一步学习。

`req.query`

包含查询串参数（有时称为 GET 参数）作为名 / 值对的一个对象。

`req.body`

包含 POST 参数的一个对象。把这个参数对象叫作 `body`，是因为 POST 参数是放入 `body` 里传过来的，而不是像查询串参数那样包含在 URL 里。要想使用 `req.body`，需要有能解析 `body` 内容类型的中间件，这部分内容将在第 10 章学习。

`req.route`

有关当前所匹配路由的信息。主要用于路由调试。

`req.cookies/req.signedCookies`

包含客户端传过来的 Cookie 值的对象。参见第 9 章。

`req.headers`

从客户端接收到的请求头信息。这是一个对象，它的键是请求头的名称，值是请求头的值。注意它来自底层的 `http.IncomingMessage` 对象，因此没有在 Express 文档中列出来。

`req.accepts(types)`

一个用于确定客户端是否接受某个（或某几个）媒体类型（可选的 `types` 参数可以是单个 MIME 类型，比如 `application/json`，也可以是一个逗号分隔的列表，或一个数组）的简便方法。对这个方法最感兴趣的是那些写公开 API 的人。一般认为，如果没有指定，浏览器总是会默认接受 HTML。

`req.ip`

客户端的 IP 地址。

`req.path`

请求的路径（不包含协议、主机、端口或查询串）。

`req.hostname`

一个用以返回客户端报告的（服务器）主机名的简便方法。这个信息可能会伪造，不应该用于涉及安全的用途。

`req.xhr`

一个简便属性，如果请求是由 Ajax 调用发起的，就返回 `true`。

`req.protocol`

建立当前请求所使用的协议（对本书来说，不是 `http` 就是 `https`）。

`req.secure`

一个简便属性，如果连接是安全的就返回 `true`。这等价于 `req.protocol === 'https'`。

```
req.url / req.originalUrl
```

名称有点儿误导性，它们返回路径和查询串（不包含协议、主机或端口）。出于内部路由的目的，`req.url` 可以被重写，而 `req.originalUrl` 的设计本意就是为了保存最初的需求和查询串。

6.8 response 对象

`response` 对象（`response` 是作为请求处理函数的第二个参数被传进来的，这意味着可以任意命名，常见的命名是 `res`、`resp` 或 `response`）的生命周期是从作为 Node 核心对象 `http.ServerResponse` 的一个实例开始的。Express 给这个对象增加了更多功能。我们来看一下 `response` 对象最有用的属性和方法（所有这些都是由 Express 增加的）。

```
res.status(code)
```

设置 HTTP 状态码。Express 默认设置为 200（OK），当你想要返回 404（页面未找到）、500（服务器错误）或其他状态码时，就必须使用这个方法。对于重定向来说（状态码 301、302、303 和 307），使用 `redirect` 方法更合适。请注意，`res.status` 返回的是 `response` 对象本身，意味着你可以链式调用：`res.status(404).send('Not found')`。

```
res.set(name, value)
```

设置响应头信息。正常情况是不需要手动设置的。也可以一次设置多个响应头，只需要传入一个参数对象，对象的键是响应头的名称，值是响应头的值。

```
res.cookie(name, value, [options]), res.clearCookie(name, [options])
```

设置或清除在客户端存储的 Cookie。这要求某些中间件的支持，参见第 9 章。

```
res.redirect([status], url)
```

让浏览器重定向。重定向状态码默认是 302（资源已找到）。一般来说应该尽量少用重定向，除非永久性地移动了某个页面，要是那样的话，应该用状态码 301（永久性移动）。

```
res.send(body)
```

向客户端发送一个响应。Express 默认使用内容类型 `text/html`，因此如果想改为 `text/plain`，就必须在调用 `res.send` 之前调用 `res.type('text/plain')`。如果 `body` 是一个对象或数组，响应就以 JSON 的格式发送（会适当地设置内容类型）。不过如果要发送 JSON，我会推荐调用 `res.json`，这样更明确一些。

```
res.json(json)
```

发送 JSON 到客户端。

```
res.jsonp(json)
```

发送 JSONP 到客户端。

```
res.end()
```

不发送响应，结束当前连接。要想了解更多关于 `res.send`、`res.json` 和 `res.end` 的差异，可以参考 Tamas Piros 的这篇文章：“[res.json\(\) vs res.send\(\) vs res.end\(\) in Express](#)”。

```
res.type(type)
```

设置 `Content-Type` 头信息的简便方法。基本上等价于 `res.set('Content-Type', type)`，它也会尝试把文件扩展名映射到互联网媒体类型。要是提供了一个不包含斜杠的字符串的话，比如 `res.type('txt')`，结果就相当于设置内容类型 `text/plain`。这个功能（映射文件扩展名）在有些地方是有用的（例如，支持各种类型多媒体文件的自动文件服务），但一般来说，应该避免用它并选择明确地设置正确的互联网媒体类型。

```
res.format(object)
```

这个方法允许你根据 `Accept` 请求头来发送不同的内容，它主要用于 API 中，第 15 章将进一步讨论。下面是一个简单的例子：`res.format({'text/plain': 'hi there', 'text/html': 'hi there'})`。

```
res.attachment([filename]), res.download(path, [filename], [callback])
```

这两个方法都会把一个名为 `Content-Disposition` 的响应头设置为 `attachment`，这会告诉浏览器响应的内容是要下载的，而不是在浏览器上显示。可以指定 `filename`，用于提示浏览器保存文件时的文件名。使用 `res.download` 可以指定要下载的文件，而 `res.attachment` 仅是设置这个响应头，你还需要给客户端发送内容。

```
res.sendFile(path, [options], [callback])
```

这个方法会读取由 `path` 指定的文件，并把文件内容发送到客户端。这个方法应该不太常用，因为使用 `static` 中间件会更容易，只要把想让客户端访问的文件放进 `public` 目录就可以了。然而，如果想根据某些条件给同一个 URL 提供不同的资源，这个方法就比较方便了。

```
res.links(links)
```

设置 `Links` 响应头。这是个专门的响应头，不太常用。

```
res.locals,res.render(view, [locals], callback)
```

`res.locals` 是一个对象，包含渲染视图的默认上下文。`res.render` 会使用预配置的模板引擎渲染一个视图（传入 `res.render` 的 `locals` 参数不要跟 `res.locals` 相混淆：`locals` 参数会覆盖 `res.locals` 中的上下文，但没被覆盖的部分仍然可用）。注意 `res.render` 会默认使用状态码 200，指定另外的状态码就要使用 `res.status`。第 7 章将深入讨论渲染视图的话题。

6.9 深入源代码

由于 JavaScript 是一种原型继承的语言，要完全理解正在使用的对象有时候是一个挑战。Node 提供的对象会经过 Express 的扩展，而你增加的包也会扩展那些对象。想要清楚地知道对象有什么属性和方法可用有时候并不容易。通常我推荐反向操作：如果要寻找某个功能，首先检查 Express API 文档。Express API 文档相当完整，很多时候我们能从里面找到所需。

如果文档里没有想要的信息，有时候就得深入 Express 源代码。我十分鼓励你深入源代码，因为你很可能会发现，读源代码远没有想象的那般令人头疼。下面是一个快速导航，据此可以从 Express 源代码里查找想要的东西。

lib/application.js

这是 Express 的主要接口。如果想理解中间件是怎么链接起来的或视图是怎么渲染的，就需要读这个文件。

lib/express.js

一个相对短小的文件，主要提供了 `createApplication` 函数（这个文件的默认导出），这个函数用于创建 Express 应用的一个实例。

lib/request.js

扩展 Node 的 `http.IncomingMessage` 对象，从而提供一个健壮的 `request` 对象。这个文件包含 `request` 对象所有的属性和方法。

lib/response.js

扩展 Node 的 `http.ServerResponse` 对象，从而提供 `response` 对象。这个文件包含 `response` 对象的属性和方法。

lib/router/route.js

提供基本的路由支持。尽管路由在应用中处于中心地位，但这个文件才不到 230 行，既简单又优雅。

随着深入到 Express 的源代码，你很可能会想查阅 Node 文档，尤其是文档的 HTTP 模块部分。

6.10 按功能归纳

本章已经对 `request` 和 `response` 这两个对象进行了总体介绍，它们是 Express 应用的最基本的部分。然而，我们通常只会使用到它们功能的一个小子集。所以下面把它们常用的功能归纳一下。

6.10.1 渲染内容

在渲染内容时我们会频繁使用 `res.render`，因为它可以在布局内渲染视图，最具价值。偶尔你可能想写一个快速的测试页，因此如果只是想要一个测试页，可以使用 `res.send`。可以使用 `req.query` 获取查询串的值、使用 `req.session` 获取 Session 值，或使用 `req.cookie`/`req.signedCookies` 获取 cookies。示例 6-1 到示例 6-8 演示了常见的内容渲染任务。

示例 6-1 基本用法 (ch06/02-basic-rendering.js)

```
// 基本用法
app.get('/about', (req, res) => {
  res.render('about')
})
```

示例 6-2 不使用 200 状态码 (ch06/03-different-response-codes.js)

```
app.get('/error', (req, res) => {
  res.status(500)
  res.render('error')
})

// 或写成一行……

app.get('/error', (req, res) => res.status(500).render('error'))
```

示例 6-3 给视图传入一个上下文，上下文包含查询串、Cookie 和 Session 值 (ch06/04-view-with-content.js)

```
app.get('/greeting', (req, res) => {
  res.render('greeting', {
    message: 'Hello esteemed programmer!',
    style: req.query.style,
    userid: req.cookies.userid,
    username: req.session.username
  })
})
```

示例 6-4 渲染一个不需要布局的视图 (ch06/05-view-without-layout.js)

```
// 以下的布局不含布局文件,
// 因此views/no-layout.handlebars必须包含所有必要的HTML
app.get('/no-layout', (req, res) =>
  res.render('no-layout', { layout: null })
)
```

示例 6-5 渲染一个使用定制布局的视图 (ch06/06-custom-layout.js)

```
// 会使用布局文件views/layouts/custom.handlebars
app.get('/custom-layout', (req, res) =>
  res.render('custom-layout', { layout: 'custom' })
)
```

示例 6-6 渲染普通文本输出 (ch06/07-plaintext-output.js)

```
app.get('/text', (req, res) => {
  res.type('text/plain')
  res.send('this is a test')
})
```

示例 6-7 增加一个错误处理函数 (ch06/08-error-handler.js)

```
// 应该放在所有的路由之后。
// 注意，即使不需要next函数，也必须包含它，
// 这样Express才能把这个函数识别为错误处理函数
app.use((err, req, res, next) => {
  console.error('** SERVER ERROR: ' + err.message)
  res.status(500).render('08-error',
    { message: "you shouldn't have clicked that!" })
})
```

示例 6-8 增加一个 404 处理函数 (ch06/09-custom-404.js)

```
// 应该放在所有的路由之后
app.use((req, res) =>
  res.status(404).render('404')
)
```

6.10.2 处理表单

处理表单时，通常会把表单的信息存放在 `req.body` 里（偶尔存放在 `req.query` 里）。可以使用 `req.xhr` 来确定当前请求是 Ajax 请求还是浏览器请求（第 8 章将深入讨论这些内容）。请看示例 6-9 和示例 6-10。对于后面的示例，需要把解析 `body` 的中间件链进来：

```
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: false }))
```

第 8 章将进一步学习 `body` 解析中间件。

示例 6-9 基本表单处理 (ch06/10-basic-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  console.log(`received contact from ${req.body.name} <${req.body.email}>`)
  res.redirect(303, '/thank-you')
})
```

示例 6-10 更健壮的表单处理 (ch06/11-more-robust-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  try {
    // 在这里，要把联系信息保存到数据库或其他持久化设施中……
    // 现在，先模拟一个错误
    if(req.body.simulateError) throw new Error("error saving contact!")
    console.log(`contact from ${req.body.name} <${req.body.email}>`)
    res.format({
      'text/html': () => res.redirect(303, '/thank-you'),
      'application/json': () => res.json({ success: true }),
    })
  } catch (err) {
    res.status(500).send(`Sorry, ${err.message}`)
  }
})
```

```

        })
    } catch(err) {
        // 在这里处理持续运行失败
        console.error(`error processing contact from ${req.body.name} ` +
            `<${req.body.email}>`)
        res.format({
            'text/html': () => res.redirect(303, '/contact-error'),
            'application/json': () => res.status(500).json({
                error: 'error saving contact information' })
        })
    }
}
)

```

6.10.3 API服务

当你提供 API 服务时，通常会把参数存放在 `req.body` 里，就像处理表单一样，虽然也可以使用 `req.query`，但 API 服务的不同之处就在于，通常会返回 JSON、XML，甚至普通文本，而不是 HTML，而且会更多地使用不太常用的 HTTP 方法，比如 PUT、POST 和 DELETE。API 服务将在第 15 章讨论。示例 6-11 和示例 6-12 将使用下面的“产品”数组（正常情况下应该从数据库获取）：

```

const tours = [
    { id: 0, name: 'Hood River', price: 99.99 },
    { id: 1, name: 'Oregon Coast', price: 149.95 },
]

```



术语“端点”常用来描述一套 API 中的单个功能。

示例 6-11 只返回 JSON 的简单的 GET 端点 (ch06/12-api.get.js)

```
app.get('/api/tours', (req, res) => res.json(tours))
```

示例 6-12 使用了 Express 的 `res.format` 方法，以根据客户端的偏好响应不同类型的内容。

示例 6-12 返回 JSON、XML 或文本的 GET 端点 (ch06/13-api-json-xml-text.js)

```

app.get('/api/tours', (req, res) => {
    const toursXml = '<?xml version="1.0"?><tours>' +
        tours.map(p =>
            `<tour price="${p.price}" id="${p.id}">${p.name}</tour>`).
            join('') + '</tours>'
    const toursText = tours.map(p =>
        `${p.id}: ${p.name} (${p.price})`).
        join('\n')
    res.format({
        'application/json': () => res.json(tours),
        'application/xml': () => res.type('application/xml').send(toursXml),
    })
})

```

```
'text/xml': () => res.type('text/xml').send(toursXml),
'text/plain': () => res.type('text/plain').send(toursXml),
})
})
```

示例 6-13 中的 PUT 端点会更新一个产品然后返回 JSON，参数会通过请求的 body 传过来（路由字符串中的 :id 告诉 Express，向 `req.params` 增加一个 `id` 属性）。

示例 6-13 PUT 端点更新 (ch06/14-api-put.js)

```
app.put('/api/tour/:id', (req, res) => {
  const p = tours.find(p => p.id === parseInt(req.params.id))
  if(!p) return res.status(410).json({ error: 'No such tour exists' })
  if(req.body.name) p.name = req.body.name
  if(req.body.price) p.price = req.body.price
  res.json({ success: true })
})
```

最后，示例 6-14 展示了一个 DELETE 端点。

示例 6-14 做删除的 DELETE 端点 (ch06/15-api-del.js)

```
app.delete('/api/tour/:id', (req, res) => {
  const idx = tours.findIndex(tour => tour.id === parseInt(req.params.id))
  if(idx < 0) return res.json({ error: 'No such tour exists.' })
  tours.splice(idx, 1)
  res.json({ success: true })
})
```

6.11 小结

通过本章这些微型示例，希望你能对 Express 应用中常见的功能和用法有一个总体印象。这些示例本意是作为快速索引，供你将来回过头来查阅。

在前面有关视图渲染的例子中，我们已经接触过视图的模板化，在下一章中，将深入研究这个主题。

第 7 章

视图模板——使用Handlebars

本章要讨论模板化，它是一项构建和格式化用户显示的内容的技术。可以把模板化想象成以下邮件的某种演变：“亲爱的 [昵称]：我们很遗憾地通知你，[过时技术 A] 已经没有人用了，但模板化技术如日中天！”要把这封邮件发给一大批人，只需替换 [昵称] 和 [过时技术 A]。



这种替换字段的过程有时候叫作插值，听起来有些花哨，其实就是“补充缺失的信息”的意思。

服务器端的模板化虽然正在快速地被像 React、Angular 和 Vue 这样的前端框架取代，但仍然在使用，比如创建 HTML 邮件。而且，Angular 和 Vue 也是采用类似模板的方式来写 HTML，因此我们学到的服务器端模板化技术也可以用在那些前端框架上。

如果你有 PHP 开发背景，可能就见怪不怪了，因为 PHP 可以说是模板化语言的第一批语言之一。大多数适应了 Web 的语言会提供某种类型的模板化支持。不过今时不同往日，模板引擎正在跟语言解耦。

那么，模板化是什么呢？先来看看模板化取代了什么。考虑一下如何用一种语言来生成另一种语言（具体来说，需要用 JavaScript 来生成 HTML），下面是最明白、最直接的做法：

```
document.write('<h1>Please Don\'t Do This</h1>')
document.write('<p><span class="code">document.write</span> is naughty,<br>')
document.write('and should be avoided at all costs.</p>')
document.write('<p>Today\'s date is ' + new Date() + '</p>')
```

下面这种做法看起来很“明白”，大概是因为编程课一直都这样教：

```
10 PRINT "Hello world!"
```

在命令式语言中，我们都习惯于说“先这样做，再那样做，然后再那样做”。对于某些事情，这种处理方式还不错。如果需要使用 500 行 JavaScript 代码执行某个复杂的计算并得出一个结果数值，而中间的每一步都依赖于前一步，那么这种做法没有什么不对。可是反过来呢？如果你有 500 行 HTML 代码和 3 行 JavaScript 代码，那么此时写 500 遍 `document.write` 有什么意义吗？毫无意义。

事实上，以上问题可以归结为：上下文切换带来了问题。如果写了大量的 JavaScript，却要混入一些 HTML，这么做既不方便又令人困惑。反过来则会好得多。我们都习惯于在 `<script>` 块里写 JavaScript，但我希望你能看出差异：上下文切换仍然存在，要么是在写 HTML，要么是在 `<script>` 块里写 JavaScript，让 JavaScript 输出 HTML 问题重重：

- 总是担心什么字符需要转义以及如何转义；
- 使用 JavaScript 生成 HTML，而 HTML 本身又包含 JavaScript 的话，很快就会让人抓狂；
- 通常会失去编辑器语法高亮的好处，以及其他的语言特定特性；
- 要发现 HTML 的格式问题变得更困难了；
- 肉眼很难看出代码的结构；
- 其他人理解你的代码会更困难。

模板化解决了这些问题，它允许你使用目标语言来写，同时提供插入动态数据的功能。回顾前一个例子，用 Mustache 模板来重写：

```
<h1>Much Better</h1>
<p>No <span class="code">document.write</span> here!</p>
<p>Today's date is {{today}}.</p>
```

现在需要给 `{{today}}` 提供一个值，而这就是模板语言的核心。

7.1 何时使用模板

并不是说永远都不要在 JavaScript 中写 HTML，只是建议你尽可能避免这样做。相对来说，在前端代码中写 HTML 更容易让人接受，尤其是在使用一个健壮的前端框架时。比如，下面这个例子我没有什么异议：

```
document.querySelector('#error').innerHTML =
'Something <b>very bad</b> happened!'
```

可是，如果它后来变成了这样：

```
document.querySelector('#error').innerHTML =  
  '<div class="error"><h3>Error</h3>' +  
  '<p>Something <b><a href="/error-detail/' + errorNumber +  
  '">very bad</a></b> ' +  
  'happened. <a href="/try-again">Try again<a>, or ' +  
  '<a href="/contact">contact support</a>. </p></div>'
```

我就会建议采用模板了。关键在于，当决定如何在 HTML 字符串和使用模板之间划清界限时，我期望你能培养出良好的判断力。不过对我来说，宁可选择模板，也要避免使用 JavaScript 生成 HTML，除非是最简单的情形。

7.2 选择模板引擎

在 Node 的世界中，有很多模板引擎可以选择。那么该如何选择呢？这是一个很复杂的问题，很大程度上取决于你的需求。不过，下面给出了一些考量标准。

性能

显然，你希望模板引擎尽可能地快，不想看到它拖慢了网站。

用于客户端、服务器端，还是两者中均用？

大多数（但不是全部）模板引擎可用于客户端和服务器端。如果需要使用前后两端均可的模板（你需要使用），则建议选择两端都可以处理的模板。

抽象程度

你是想要一个熟悉一些（例如，就像正常的 HTML，只是多加了一些花括号）的，还是有些讨厌 HTML，更喜欢摆脱那些尖括号的模板呢？模板化（尤其是服务器端模板化）让你可以根据偏好来选择。

以上只是选择模板语言时比较重要的几条标准。目前，各种模板化技术都是相当成熟的，所以不管怎么选都不会太差。

Express 允许你使用任何想用的模板，如果 Handlebars 不合你意，则可以轻易地换掉它。如果想浏览一下候选列表，可以使用 Github 上有用又有趣的“模板引擎选择器”（尽管已经不再更新了，但仍然很有用）。

在讨论 Handlebars 之前，先来看一个抽象程度相当高的模板引擎。

7.3 Pug：另辟蹊径

大多数模板引擎以 HTML 为中心，Pug 却为你抽离了 HTML 的细节，这正是它的独特之处。值得注意的是，Pug 同样出自 Express 开发者 TJ Holowaychuk 之手。那么，说 Pug 跟 Express 集成得非常好也就一点儿也不奇怪了。Pug 认定，HTML 手写起来既啰唆又无趣，

所以它要另辟蹊径。下面来看一看 Pug 模板长什么样，以及模板输出的 HTML 是什么样子（取自 Pug 官网主页，为适合本书版面做了细微修改）：

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script.
      if (foo) {
        bar(1 + 5)
      }
  body
    h1 Pug
    #container
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
    p.
      Pug is a terse and
      simple templating
      language with a
      strong focus on
      performance and
      powerful features.

<!DOCTYPE html>
<html lang="en">
<head>
<title>Pug Demo</title>
<script>
  if (foo) {
    bar(1 + 5)
  }
</script>
<body>
<h1>Pug</h1>
<div id="container">
<p>You are amazing</p>
<p>
  Pug is a terse and
  simple templating
  language with a
  strong focus on
  performance and
  powerful features.
</p>
</body>
</html>
```

Pug 无疑代表着简洁的键盘输入（没有尖括号和关闭标签了）。依靠缩进和几条常识性的规则，Pug 让你可以更容易表达想法。Pug 还有一个优势。理论上，当 HTML 本身有什么变化时，只需重新调整 Pug 的目标为最新版本的 HTML，内容就永不过时了。

尽管我十分欣赏 Pug 的核心哲学和它优雅的行事风格，但是在我看来，并不希望 HTML 的细节都被抽离掉。作为一名 Web 开发人员，我做的所有事情都围绕着 HTML，如果使用 HTML 付出的代价是我键盘上的尖括号键都要被按坏，那就由它坏掉吧。我跟很多前端开发人员聊过，他们也有同样的感觉。所以，也许现在大家还没准备好接受 Pug。

到这里我们就要跟 Pug 分道扬镳了，后文中你将不会再看到它。不过，如果这样的抽象方式很吸引你，那么在 Express 中使用 Pug 是完全没问题的，而且也有大量的资源可以提供帮助。

7.4 Handlebars 基础

Handlebars 是另一个流行的模板引擎 Mustache 的扩展。我推荐 Handlebars，一是它易于集成进 JavaScript（不管是前端还是后端），二是它的语法我们比较熟悉。对我来说，它各

个方面的折中都做到了恰到好处，因此本书主要关注 Handlebars。不过，我们将要涉及的很多概念基本上可以在其他模板引擎中应用，所以即使你不是非常喜欢 Handlebars，也为后续尝试各种不同的模板引擎做好了准备。

理解模板化的关键是要理解上下文的概念。当你渲染一个模板时，给模板引擎传入一个“上下文对象”，有了这个对象，替换才得以进行。

例如，如果上下文对象是：

```
{ name: 'Buttercup' }
```

模板是：

```
<p>Hello, {{name}}!</p>
```

那么 {{name}} 将被 Buttercup 替换。如果想把 HTML 传入模板会怎样？例如，如果上下文是：

```
{ name: '<b>Buttercup</b>' }
```

还是使用前面那个模板，则结果是 <p>Hello, Buttercup</p>，这可能不是你想要的。要解决这个问题，只需使用 3 个花括号，而不是 2 个：{{{{name}}}}。



虽然我们已经认同了应该避免在 JavaScript 里写 HTML，但是，如果能够使用 3 个花括号关闭 HTML 转义，那么这样做还是有一些重要用途的。例如，如果你在构建一个内容管理系统（CMS），且使用了“所见即所得”（WYSIWYG）编辑器，你很可能会希望给视图传入 HTML。而且，渲染上下文里面的属性时，不转义 HTML 对布局和 sections 也十分关键，一会儿我们就会学到。

图 7-1 展示了 Handlebars 引擎是如何使用上下文（用一个椭圆表示）并结合模板来渲染 HTML 的。

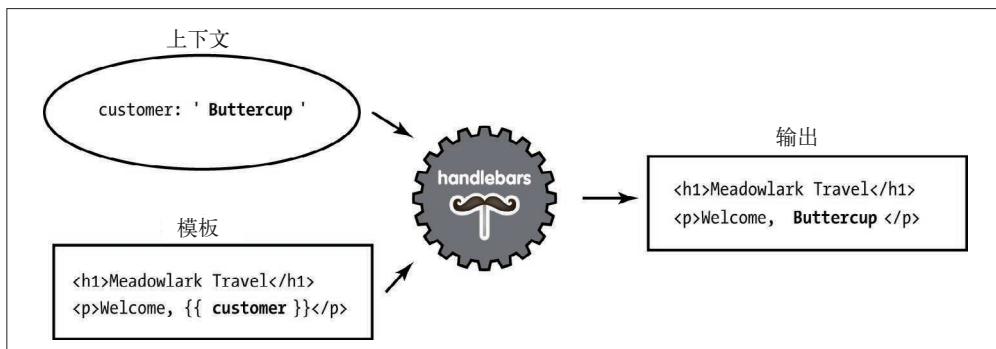


图 7-1：使用 Handlebars 渲染 HTML

7.4.1 注释

Handlebars 中的注释是这样的: `{{! comment goes here }}`。理解 Handlebars 注释和 HTML 注释的区别很重要。考虑下面的模板:

```
 {{! super-secret comment }}  
 <!-- not-so-secret comment -->
```

假设这是服务器端模板, `super-secret` 这条注释是绝对不会发送到浏览器的, 而如果用户查看 HTML 源代码, 就会看到 `not-so-secret` 这条注释。对于任何会暴露实现细节或不想公开的地方, 都应该选择 Handlebars 注释。

7.4.2 代码块

当你考虑代码块的时候, 事情就开始变得复杂了。代码块提供了流程控制、条件执行和扩展性。考虑下面的上下文对象:

```
{  
  currency: {  
    name: 'United States dollars',  
    abbrev: 'USD',  
  },  
  tours: [  
    { name: 'Hood River', price: '$99.95' },  
    { name: 'Oregon Coast', price: '$159.95' },  
  ],  
  specialsUrl: '/january-specials',  
  currencies: [ 'USD', 'GBP', 'BTC' ],  
}
```

把这个上下文传入下面的模板:

```
<ul>  
  {{#each tours}}  
    {{! 现在一个新的代码块里了, 上下文已经改变了 }}  
    <li>  
      {{name}} - {{price}}  
      {{#if ./currencies}}  
        ({{./currency.abbrev}})  
      {{/if}}  
    </li>  
  {{/each}}  
</ul>  
{{#unless currencies}}  
  <p>All prices in {{currency.name}}.</p>  
{{/unless}}  
{{#if specialsUrl}}  
  {{! 现在一个新的代码块里了, 但上下文基本上没有改变 }}  
  <p>Check out our <a href="{{specialsUrl}}>specials!</a></p>  
{{else}}
```

```

<p>Please check back often for specials.</p>
{{/if}}
<p>
  {{#each currencies}}
    <a href="#" class="currency">{{.}}</a>
  {{else}}
    Unfortunately, we currently only accept {{currency.name}}.
  {{/each}}
</p>

```

这个模板里有很多东西，我们一个一个来说。它从一个 `each` 辅助函数开始，这个辅助函数可以迭代一个数组。重要的是要理解在 `{{#each tours}}` 和 `{{/each tours}}` 之间上下文发生了变化。在第一轮，上下文变为 `{ name: 'Hood River', price: '$99.95' }`，在第二轮，上下文变为 `{ name: 'Oregon Coast', price: '$159.95' }`。因此，可以在代码块内引用 `{{name}}` 和 `{{price}}`。不过，如果想访问 `currency` 对象，则必须使用 `..` 来访问父级上下文。

如果上下文中的一个属性本身也是一个对象，则可以像平常一样用句点来访问它的属性，比如 `{{currency.name}}`。

`if` 和 `each` 都有一个可选的 `else` 块（对 `each` 来说，如果数组中没有元素，`else` 块就会得到执行）。我们也使用了 `unless` 辅助函数，它本质上跟 `if` 相反，即仅当参数为 `false` 时才会执行。

关于这个模板，最后需要注意的是 `{{.}}` 在 `{{#each currencies}}` 块中的使用。`{{.}}` 引用当前上下文，在这里，当前上下文只是数组中的一个我们想要打印的字符串元素。



使用单独一个句点访问当前上下文还有一个用途：它可以区分当前上下文中的辅助函数（一会儿就会学到）和属性。例如，如果当前上下文中有一个叫作 `foo` 的辅助函数和一个叫作 `foo` 的属性，那么 `{{foo}}` 引用的是辅助函数，而 `{{./foo}}` 引用的是属性。

7.4.3 服务器端模板

服务器端模板允许先渲染好 HTML 再发送到客户端。对于那些知道如何查看 HTML 源代码的“好奇用户”来说，客户端模板是可以看得到的，而服务器端模板不同，用户永远看不到用于生成最终 HTML 的服务器端模板或上下文对象。

除了可以隐藏实现细节，服务器端模板还支持模板缓存，缓存对提升性能很重要。模板引擎会缓存编译后的模板（仅当模板本身有变更时才会重新编译和重新缓存），从而提升模板化的视图的性能。如果没有设置，视图缓存在开发模式下是禁用的，而在产品模式下是启用的。如果想明确地启用视图缓存，可以这样实现：

```
app.set('view cache', true)
```

Express 只内建了对 Pug、EJS 和 JSHTML 的支持。前面已经讨论过 Pug，而且我也找不到什么理由来推荐 EJS 或 JSHTML（在我看来，它们在语法上的建树都不够）。因此，我们需要增加一个在 Express 里提供 Handlebars 支持的 Node 包：

```
npm install express-handlebars
```

然后把它链进 Express 中（版本库的 ch07/meadowlark.js）：

```
const expressHandlebars = require('express-handlebars')
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
app.set('view engine', 'handlebars')
```



express-handlebars 期望 Handlebars 模板的扩展名为 .handlebars。我对这个扩展名已经习惯了，但如果你觉得有些长，可以在创建 express-handlebars 实例的时候，把扩展名改成同样常用的 .hbs：`app.engine('.hbs', expressHandlebars({ extname: '.hbs' }))`；`app.set('view engine', '.hbs')`。

7.4.4 视图和布局

视图通常代表网站上独立的一个页面（尽管它也可以代表由 Ajax 加载的页面、邮件或其他东西的一部分）。默认条件下，Express 会在 views 子目录下查找视图。布局是特殊类型的视图——本质上，它是模板的模板。布局是非常必要的，因为大部分（即使不是全部）网站的页面有着相似的结构。例如，它们必须要有一个 `<html>` 元素和一个 `<title>` 元素，通常都加载同样的 CSS 文件，等等。你并不希望每个页面都重复那些代码，这时候布局文件就该出场了。下面来看一个最简化的布局文件：

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
    <link rel="stylesheet" href="/css/main.css">
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

请注意 `<body>` 标签里的文本：`{{{body}}}`，视图引擎由这个文本知道要在此处渲染视图内容。这里使用了 3 个而不是 2 个花括号很重要，因为视图很可能包含 HTML，而我们不希望 Handlebars 尝试对 HTML 转义。不过，把 `{{{body}}}` 放到什么位置是没有限制的。例如，如果你正在用 Bootstrap 构建响应式的布局，很可能会把视图放入某个容器 `<div>` 里。而且，公共的页面元素，比如页眉和页脚，通常放在布局里，而不是视图里。下面是一个例子：

```

<!-- ... -->
<body>
  <div class="container">
    <header>
      <div class="container">
        <h1>Meadowlark Travel</h1>
        
      </div>
    </header>
    <div class="container">
      {{body}}
    </div>
    <footer>&copy; 2019 Meadowlark Travel</footer>
  </div>
</body>

```

图 7-2 展示了模板引擎是如何编译视图、布局和上下文的。这个图还厘清了一件很重要的事情，即操作的顺序。在布局之前，**最先渲染的是视图**。乍一看似乎有些反直觉，毕竟视图是渲染到布局里面的，难道不是应该先渲染布局吗？尽管技术上可以实现，但是这样调换操作顺序有其好处，尤其是先渲染视图可以允许视图本身进一步去定制布局这一点。到本章后面讨论 `sections` 的时候，你就会知道定制布局的用处了。

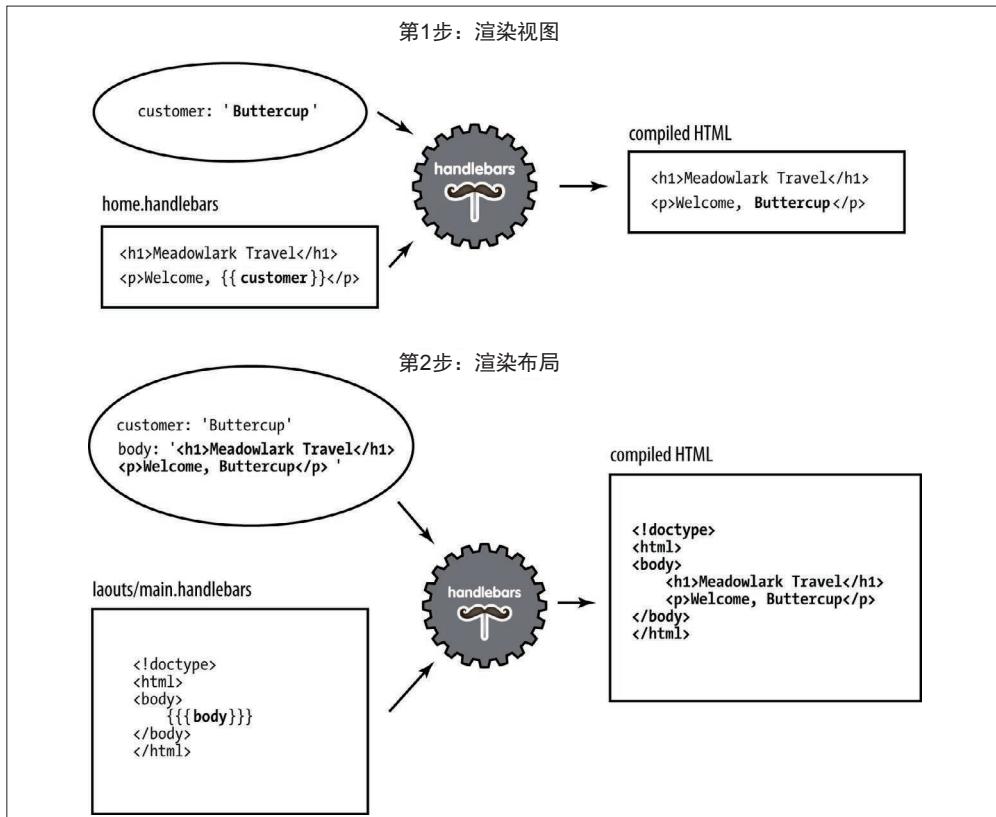


图 7-2：使用布局渲染视图



正是因为这样的操作顺序，你才能把一个叫 `body` 的属性传入视图，使其被正确渲染。然而，到了渲染布局的时候，`body` 的值就被渲染后的视图覆盖了。

7.4.5 在Express中使用（或不使用）布局

很可能大部分（即使不是全部）页面需要使用同一个布局，所以每次渲染视图都指定这个布局没什么意义。你会注意到，我们在创建视图引擎的时候，指定了默认布局的名字：

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
```

如果没有特别设置，Express 会在 `views` 子目录下查找视图，在 `views/layouts` 下查找布局。所以如果有一个视图 `views/foo.handlebars`，你可以这样渲染：

```
app.get('/foo', (req, res) => res.render('foo'))
```

它会使用 `views/layouts/main.handlebars` 作为布局。如果根本不想使用布局（意味着你的视图会包含所有的页面框架代码），可以在上下文对象中指定 `layout: null`：

```
app.get('/foo', (req, res) => res.render('foo', { layout: null }))
```

或者，如果想使用一个不同的模板，可以指定模板名：

```
app.get('/foo', (req, res) => res.render('foo', { layout: 'microsite' }))
```

这样就会使用布局 `views/layouts/microsite.handlebars` 来渲染视图。

要记住的是，拥有的模板越多，需要维护的 HTML 布局就越简单。而如果你的诸多页面在布局上大不相同，那么拥有更多的模板也许是值得的。你需要找到符合项目需求的平衡点。

7.4.6 sections

微软有一个非常棒的模板引擎叫 `Razor`，我从中借用了一个技术点子，并将其命名为 `sections`。如果所有的视图都可以整整齐齐地放入布局的某个元素里面，那么布局当然可以正常工作，但当视图需要放入布局的不同部分时会怎么样？一个常见的例子是，视图需要在 `<head>` 里增加东西，或插入一个 `<script>`，考虑到性能，有时候这些东西会被放到布局文件的最后。

Handlebars 和 `express-handlebars` 都没有内建这个功能。幸运的是，Handlebars 辅助函数可以使这些操作变得相当容易。我们在实例化 Handlebars 对象的时候，增加一个名为 `section` 的辅助函数（版本库的 `ch07/meadowlark.js`）：

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
  helpers: {
    section: function(name, options) {
      if(!this._sections) this._sections = {}
      this._sections[name] = options.fn(this)
      return null
    },
  },
}))
```

现在可以在视图里使用这个辅助函数 `section` 了。增加一个视图 (`views/section-test.handlebars`)，在 `<head>` 里增加一些东西，并增加一个 `script`:

```
{{#section 'head'}}
  <!-- 希望谷歌忽略本页 -->
  <meta name="robots" content="noindex">
{{/section}}

<h1>Test Page</h1>
<p>We're testing some script stuff.</p>

{{#section 'scripts'}}
  <script>
    document.querySelector('body')
      .insertAdjacentHTML('beforeEnd', '<small>(scripting works!)</small>')
  </script>
{{/section}}
```

现在，可以在布局文件里放入这些 `sections` 了，就像放入 `{{{body}}}` 一样：

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
    {{{_sections.head}}}
  </head>
  <body>
    {{{body}}}
    {{{_sections.scripts}}}
  </body>
</html>
```

7.4.7 partial模板

很多时候，你希望跨页面重用一些页面组件（在前端圈子里有时候叫作 `widget`）。要用模板来达到这个目的，一种办法就是使用 `partial`（局部）模板（这样叫是因为它们并不渲染整个视图或整个页面）。假设我们需要一个“当前天气”组件，用来显示 Portland、Bend 和 Manzanita 的当前天气。我们希望这个组件可以简单地跨页面使用，于是选择使用 `partial` 模板。首先，创建一个 `partial` 文件，即 `views/partials/weather.handlebars`：

```

<div class="weatherWidget">
  {{#each partials.weatherContext}}
    <div class="location">
      <h3>{{location.name}}</h3>
      <a href="{{location.forecastUrl}}">
        
        {{weather}}, {{temp}}
      </a>
    </div>
  {{/each}}
  <small>Source: <a href="https://exl.ptpress.cn:8442/ex/l/be4512da">
    National Weather Service</a></small>
</div>

```

注意上下文的命名空间，我们在开头把它设定为 `partials.weatherContext`。既然想让每个页面都可以使用这个 partial，把每个视图都传入这个上下文就不太切合实际了，那么我们使用 `res.locals`（每个视图都可以使用它）。但又因为不想干扰到视图各自指定的上下文，所以干脆把所有 partial 的上下文都放入 `partials` 对象。



`express-handlebars` 允许把 partial 模板作为上下文的一部分来传入。例如，如果把 `partials.foo = "Template!"` 加入上下文对象，就可以用 `{{> foo}}` 来渲染这个 partial 模板。这个用法会优先于任何 `.handlebars` 视图文件，这就是为什么前面使用 `partials.weatherContext` 而不是 `partials.weather`，因为后者优先于 `views/partials/weather.handlebars`。

在第 19 章中，我们会看到如何从公开的“全国天气服务 API”获取当前天气信息。但就目前来说，我们打算先使用从函数 `getWeatherData` 返回的模拟数据。

在本例中，我们希望天气数据对每个视图都适用，对此最好的机制就是中间件（第 10 章将深入学习）。这个中间件会把天气数据注入 `res.locals.partials` 对象，这样天气数据就可以作为上下文在 partial 模板里访问到。

为了让这个中间件更容易测试，把它放入它自己的文件 `lib/middleware/weather.jsz` 中（版本库的 `ch07/01/lib/middleware/weather.js`）：

```

const getWeatherData = () => Promise.resolve([
  {
    location: {
      name: 'Portland',
      coordinates: { lat: 45.5154586, lng: -122.6793461 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PQR/112,103/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra,40?size=medium',
    weather: 'Chance Showers And Thunderstorms',
    temp: '59 F',
  },
])

```

```

location: {
  name: 'Bend',
  coordinates: { lat: 44.0581728, lng: -121.3153096 },
},
forecastUrl: 'https://api.weather.gov/gridpoints/PDT/34,40/forecast',
iconUrl: 'https://api.weather.gov/icons/land/day/tsra_sct,50?size=medium',
weather: 'Scattered Showers And Thunderstorms',
temp: '51 F',
},
{
  location: {
    name: 'Manzanita',
    coordinates: { lat: 45.7184398, lng: -123.9351354 },
  },
  forecastUrl: 'https://api.weather.gov/gridpoints/PQR/73,120/forecast',
  iconUrl: 'https://api.weather.gov/icons/land/day/tsra,90?size=medium',
  weather: 'Showers And Thunderstorms',
  temp: '55 F',
},
])
)

const weatherMiddleware = async (req, res, next) => {
  if(!res.locals.partials) res.locals.partials = {}
  res.locals.partials.weatherContext = await getWeatherData()
  next()
}

module.exports = weatherMiddleware

```

现在一切都已就绪，只需在视图中使用这个 partial 模板。例如，要在这个部件加到主页，编辑 views/home.handlebars：

```
<h2>Home</h2>
{{> weather}}
```

`{{> partial_name}}` 这个语法是用来实现在视图中使用一个 partial 的：express-handlebars 将知道在 views/partials 下查找一个名为 `partial_name.handlebars` 的视图（在本例中就是 `weather.handlebars`）。



express-handlebars 支持子目录，所以如果你有大量的 partial，就可以使用目录分层整理。例如，如果你有一些社交媒体的 partial，就可以把它们放入 views/partials/social 目录，然后使用 `{{> social/facebook}}` 或 `{{> social/twitter}}` 等把它们加入进来。

7.4.8 完善模板

模板是网站的核心。好的模板结构可以节省开发时间，提升整个网站的一致性，并减少布局隐藏的突发状况。不过要获得这些好处，就得花一些时间，仔细雕琢模板。决定所需模板数量也是一门艺术，一般来说越少越好，但是需要根据页面的一致性程度酌情增减，过

犹不及。在跨浏览器兼容性和合法 HTML 方面，模板也是你的第一道防线。这些模板应该由精通前端开发的人员细心地编写和维护。一个很好的入手点（尤其对新手来说）是使用 HTML5 样板。在前面的示例中，为适合排版，我们一直都是使用最小化的 HTML5 模板，但对于实际的项目，则会使用 HTML5 样板。

另一个入手点就是第三方主题，很多人会采用这种方式。像 Themeforest 和 WrapBootstrap 这样的网站有数百个直接可用的 HTML5 主题，你可以把它们当作模板起点。使用第三方主题，先要采纳它的主文件（通常是 index.html），把它重命名为 main.hanidlebars（或者自定义的布局文件名），把所有资源文件（CSS、Javascript、图片）放入用以存放静态文件的 public 目录。然后必须编辑这个模板文件，并确定要把 {{{body}}} 表达式放到哪里。

你可能想把一部分模板中的元素移入 partial 模板。主页横幅（一种高高的横幅广告，用来抓住用户的眼球）就是一个很好的例子。如果这个主页横幅需要在每一页出现（可能不是个好的选择），就让它留在模板文件里；如果只在某一页出现（通常是主页），那就把它放在对应的视图里；如果需要在好几页（但不是全部）出现，则可以考虑把它放进一个 partial 里。如何选择你自己决定，而这正是打造一个独特而吸引人的网站的艺术。

7.5 小结

我们已经看到模板化是如何使得代码更易写、易读、易维护的。多亏了模板，我们才不需要痛苦地在 JavaScript 语言中胡乱拼凑 HTML，才可以在喜欢的编辑器中写 HTML，然后使用一个紧凑而易读的模板语言实现动态化。

我们已经知道了如何格式化要显示的内容，可以转到下一章了，即如何使用 HTML 表单，把来自客户端的数据收入系统。

第8章

表单处理

要采集用户提供的信息，通常做法就是使用 HTML 表单。不管是让浏览器正常地提交表单、使用 Ajax，还是采用高级的前端控件，底层的机制通常仍然是 HTML 表单。本章将讨论表单处理、表单验证和文件上传的各种方法。

8.1 把客户端数据发送到服务器

大体而言，把客户端数据发送到服务器有两种选择，即使用查询串和请求 body。正常情况下，如果使用查询串，就是发起 GET 请求；如果使用请求 body，就是使用 POST 请求（HTTP 协议并不会阻止你反其道而行之，但这样做没什么意义，最好还是遵循标准实践）。

常见的一种误解认为 POST 是安全的，而 GET 是不安全的。但实际上，如果使用 HTTPS，两者都是安全的；如果不使用 HTTPS，则两者都不安全。如果没有使用 HTTPS，那么侵入者查看一个 POST 请求的 body 数据就像查看 GET 请求的查询串一样容易。不过，如果使用 GET 请求，你的用户就可以从查询串看到他们所有的输入（包括隐藏域），而这个查询串又丑又乱。而且，通常浏览器对查询串的长度施加了限制（对请求的 body 则没有这样的限制）。出于这些原因，我一般推荐使用 POST 做表单提交。

8.2 HTML 表单

本书主要关注服务器端，但了解一下 HTML 表单的构造也很有必要。下面是一个简单的例子：

```
<form action="/process" method="POST">
  <input type="hidden" name="hush" val="hidden, but not secret!">
  <div>
    <label for="fieldColor">Your favorite color: </label>
    <input type="text" id="fieldColor" name="color">
  </div>
  <div>
    <button type="submit">Submit</button>
  </div>
</form>
```

需要注意的是，`<form>` 标签里的方法明确指定为 POST，如果没有指定，默认就是 GET。`action` 属性指定了（表单提交后）接收表单数据的 URL。如果省略这个字段，那么表单将被提交到加载这个表单的同一个 URL。建议最好提供一个合法的 `action`，即便你使用的是 Ajax（这是为了避免数据丢失，更多信息参见第 22 章）。

从服务器的角度来看，`<input>` 字段中的 `name` 是个重要属性：这就是服务器标识字段的方式。`name` 属性跟 `id` 属性不一样，这一点一定要理解，`id` 属性一般只用于样式和前端功能（它不会被传到服务器）。

注意那个隐藏域：它并不会显示在浏览器上。但是你也不能将其用于密码或敏感信息，因为用户只需查看页面源代码，隐藏域就一览无余了。

HTML 并不限制一个页面里有多少个表单（很遗憾，早期一些服务器端框架就有这个限制，比如 ASP）。建议保持表单逻辑上的一致性，一个表单应当包含你想一次性提交的所有字段（有可选字段或空字段没关系），不想提交的一个也不要。如果一个页面里有两个不同的操作，那么就使用两个不同的表单，比如将一个表单用作全站搜索，另一个表单用作订阅邮件列表。尽管用一个大表单并根据用户点了哪个按钮来确定要做什么操作可行，但这样做会让人头疼，而且对残障人士也常常是不友好的（由于无障碍浏览器对表单的渲染方式）。

在上面的例子中，当用户提交表单时，/process 这个 URL 会得到激活，字段的值会被放入请求的 body 里传输给服务器。

8.3 表单的编码

提交表单时（不管是通过浏览器提交还是通过 Ajax 提交）必须以某种方式编码。如果没有明确指定编码，默认就是 `application/x-www-form-urlencoded`（这只是“URL encoded”冗长媒体类型名）。这是一个基本而易用的编码，由 Express 提供内建的支持。

如果需要上传文件，事情就更加复杂了，因为使用 URL 编码来发送文件没有简单的办法，所以只好使用 `multipart/form-data` 编码，而 Express 无法直接处理这个编码。

8.4 处理表单的不同做法

如果不使用 Ajax，就只能通过浏览器提交表单了，这样做的结果是会刷新页面。不过，页面要怎样刷新由你决定。当处理表单时，有两件事情需要考虑：由哪个路径处理这个表单（表单中的 `action`）以及给浏览器返回什么样的响应。

如果使用 `method="POST"`（这是推荐做法），那么显示表单和处理表单都使用同一个路径也是相当常见的。不过，它们是可以区分的，因为显示表单是一个 GET 请求，而处理表单是一个 POST 请求。如果采取这种做法，就可以忽略表单的 `action` 属性。

另一个选择是使用另外的路径来处理表单。例如，如果联系信息页使用了路径 `/contact`，可以使用路径 `/process-contact` 来处理表单（通过指定 `action="/process-contact"`）。如果采取这种做法，就可以选择使用 GET 来提交表单（我不推荐这样做，因为在 URL 上暴露表单的字段毫无必要）。如果有多个 URL 是同样的提交途径（例如，网站的多个页面都有使用邮箱地址的注册框），那么为表单提交提供单独的端点或许会更好。

不管用什么路径来处理表单，都需要决定给浏览器返回什么样的响应。你有以下几种选择。

直接 HTML 响应

处理完表单之后，可以直接把 HTML 发回浏览器（比如说一个视图）。如果试图刷新页面，浏览器就会给出一条警告信息。而且这样做还会影响到浏览器的书签和返回按钮。所以，不推荐这种做法。

302 重定向

尽管这是常见的做法，但它是对 302（已找到）状态码最初含义的误用。HTTP 1.1 增加了 303（另见）状态码，这个状态码用起来更合适。除非你要适配 1996 年以前的浏览器，否则应该用 303。

303 重定向

303（另见）状态码是在 HTTP 1.1 中加入的，为了解决对 302 重定向的误用。HTTP 规范特别指出，浏览器应当用 GET 请求来跟随 303 重定向，而不使用最先提到的 HTTP 方法。这是响应表单提交的推荐做法。

既然响应表单提交的推荐做法是 303 重定向，那么下一个问题是，“重定向要指向哪里？”答案由你来决定。以下是几种常见的做法。

重定向至专门的成功 / 失败页

这种做法要求分配专门的 URL，以用于显示成功或失败信息。例如，如果用户订阅促销邮件，但是发生了数据库错误，则可以重定向至 `/error/database`；如果用户的邮箱地址不合法，则可以重定向至 `/error/invalid-email`；如果一切顺利，就可以重定向至 `/promo-email/thank-you`。使用专门的成功 / 失败页的优势之一是有利于统计分析。

/promo-email/thank-you 页的访问量大体上跟促销邮件的订阅用户数相关。这种做法实现起来也很直观，但有一些不足。这种做法意味着你要为每一种可能分配一个 URL，这意味着要设计页面、写页面代码并维护。还有一处不足是它的用户体验也不是最优的。用户愿意看到感谢的话，但他们为此还得返回到原来的页面，或者转到想要的页面。目前先采用这种做法，第 9 章再切换到使用 flash 消息（不要与 Adobe Flash 混淆）。

重定向至最初页面，包含一条 flash 消息

对于遍布全网的小表单（例如输入邮箱订阅），最好的用户体验是不打断用户的页面导航。也就是说，需要提供一种提交邮箱地址而不离开当前页面的方法。当然，一种方法就是使用 Ajax。但如果不想使用 Ajax（或者为了保证用户体验，你需要一种回退机制），则可以重定向到用户原先所在的页面。最简单的做法就是使用表单中的一个隐藏域，这个隐藏域存放了当前的 URL。既然你希望在原先的页面给出反馈，告知用户提交已经接收了，就可以使用 flash 消息。

重定向至新页面，包含一条 flash 消息

大表单通常都有自己的页面，提交表单之后还停留在表单页面就没有什么意义了。这种情况下，你需要判断出用户可能想去哪里并相应地重定向。例如，如果你在开发后台管理功能并创建了一个新的度假套餐表单，那么可以合理推测，在提交这个表单后，用户会想跳转到所有套餐列表的管理页面。即使如此，你还是应该提供一条 flash 消息，告知用户表单提交的结果。

如果你在使用 Ajax，推荐使用专门的 URL 来处理表单。可能你会想给 Ajax 路由的 URL 增加一个统一的前缀（例如 /ajax/enter），但我不鼓励这样做，因为这样就给 URL 关联上了实现的细节。而且，我们一会儿就会看到，作为一个容错机制，你的 Ajax 路由应该能处理常规的浏览器表单提交。

8.5 使用 Express 处理表单

如果表单处理使用的是 GET 方法，就可以从 `req.query` 对象访问到表单字段。例如，如果表单有一个名为 `email` 的 HTML 输入字段，那么它的值将作为 `req.query.email` 传到处理函数。这种方式十分简单，没有太多需要讲的。

如果你使用的是 POST 方法（这是我推荐的），就要引入一个解析（以 URL 编码的）请求的 body 的中间件。首先，安装 `body-parser` 中间件 (`npm install body-parser`)，然后，把它链进来（版本库的 `ch08/meadowlark.js`）：

```
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: true }))
```

把 `body-parser` 链进来之后，你就可以看到 `req.body` 可用了，而且通过它可以访问到所有

的表单字段。注意 `req.body` 并不会阻止你使用查询串。下面来给草地鹨旅游增加一个表单，让用户可以订阅邮件列表。出于演示的目的，在 `/views/newsletter-signup.handlebars` 文件中，将使用查询串、一个隐藏域以及几个可见字段：

```
<h2>Sign up for our newsletter to receive news and specials!</h2>
<form class="form-horizontal" role="form"
      action="/newsletter-signup/process" method="POST">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label">Name</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
             id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
    <div class="col-sm-4">
      <input type="email" class="form-control" required
             id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-primary">Register</button>
    </div>
  </div>
</form>
```

我们使用了 Bootstrap 的样式，本书剩余部分也会使用它，所以如果你对 Bootstrap 还不熟悉，请参考 Bootstrap 官方文档。

我们已经把解析 `body` 的中间件链进来了，现在需要为邮件订阅页、表单处理和“感谢订阅”页增加处理函数（版本库的 `ch08/lib/handlers.js`）：

```
exports.newsletterSignup = (req, res) => {
  // 关于CSRF，后面会学到，目前先提供一个模拟的值
  res.render('newsletter-signup', { csrf: 'CSRF token goes here' })
}

exports.newsletterSignupProcess = (req, res) => {
  console.log('CSRF token (from hidden form field): ' + req.body._csrf)
  console.log('Name (from visible form field): ' + req.body.name)
  console.log('Email (from visible form field): ' + req.body.email)
  res.redirect(303, '/newsletter-signup/thank-you')
}

exports.newsletterSignupThankYou = (req, res) =>
  res.render('newsletter-signup-thank-you')
```

（如果还没创建好“感谢订阅”页的视图，就先创建 `views/newsletter-signup-thank-you.handlebars` 文件。）

最后，把这些处理函数链进应用里（版本库的 ch08/meadowlark.js）：

```
app.get('/newsletter-signup', handlers.newsletterSignup)
app.post('/newsletter-signup/process', handlers.newsletterSignupProcess)
app.get('/newsletter-signup/thank-you', handlers.newsletterSignupThankYou)
```

这些就是所有需要做的了。注意在处理函数里，我们重定向到了一个“感谢订阅”页，在里面可以渲染一个视图。但要是那样做的话，访问者浏览器中的 URL 字段还是保持 /process，这会令人困惑。在这种情况下，让浏览器发出一个重定向请求就可以解决问题。



在这种情形中，你要使用 303（或 302）重定向而不是 301 重定向，这很重要。301 重定向是“永久重定向”，意味着浏览器可能会缓存重定向的目标。如果你使用了 301 重定向并且尝试再次提交这个表单，浏览器就可以完全绕过 /process 这个表单处理，直接进入 /thank-you，因为它正确地理解了这个重定向是永久的。另外，303 重定向就相当于告诉浏览器，“好的，你的请求没问题，你可以从这里获取响应”，并且不要缓存重定向的目标。

对于大多数前端框架来说，更常见的是使用 `fetch` API（一会儿就会看到），把表单数据以 JSON 格式发送出去。不过，理解浏览器默认如何处理表单提交仍然有好处，因为你还会看到以这种方式实现的表单。

现在来看看如何使用 `fetch` 来提交表单。

8.6 使用 `fetch` 发送表单数据

使用 `fetchAPI` 来发送 JSON 编码的表单数据是相当现代的技术，可以让你对客户端 / 服务器通信有更多的控制，而且让你的页面刷新更少。

既然表单提交不再跳转到一个页面，就不用再操心重定向和多个页面 URL 的问题（表单处理还是使用独立的 URL）了，因此，我们就把整个“邮件列表订阅”统一在一个名为 /newsletter 的 URL 下。

先从前端代码开始。HTML 表单的内容本身不需要修改（表单字段和布局都一样），但我们不需要指定 `action` 或 `method`，同时要把表单包含在一个 `<div>` 里，便于显示“感谢”信息：

```
<div id="newsletterSignupFormContainer">
  <form class="form-horizontal" role="form" id="newsletterSignupForm">
    <!-- 其余部分是一样的 -->
  </form>
</div>
```

然后用脚本拦截表单提交事件并取消它（使用 `Event#preventDefault`），以便可以自己处理表单（版本库的 ch08/views/newsletter.handlebars）：

```

<script>
  document.getElementById('newsletterSignupForm')
    .addEventListener('submit', evt => {
      evt.preventDefault()
      const form = evt.target
      const body = JSON.stringify({
        _csrf: form.elements._csrf.value,
        name: form.elements.name.value,
        email: form.elements.email.value,
      })
      const headers = { 'Content-Type': 'application/json' }
      const container =
        document.getElementById('newsletterSignupFormContainer')
      fetch('/api/newsletter-signup', { method: 'post', body, headers })
        .then(resp => {
          if(resp.status < 200 || resp.status >= 300)
            throw new Error(`Request failed with status ${resp.status}`)
          return resp.json()
        })
        .then(json => {
          container.innerHTML = '<b>Thank you for signing up!</b>'
        })
        .catch(err => {
          container.innerHTML = `<b>We're sorry, we had a problem ` +
            `signing you up. Please <a href="/newsletter">try again</a>`
        })
    })
</script>

```

现在转到服务器端文件（meadowlark.js），先确认我们链进了可以解析 JSON 格式的请求 body 的中间件，然后指定两个端点：

```

app.use(bodyParser.json())

//...

app.get('/newsletter', handlers.newsletter)
app.post('/api/newsletter-signup', handlers.api.newsletterSignup)

```

注意，我们将表单处理端点放在了以 `api` 开头的 URL 上。这是常见的做法，可以区分用户（浏览器）端点跟本意是通过 `fetch` 访问的 API 端点。

现在把这些端点加入 `lib/handlers.js` 文件中：

```

exports.newsletter = (req, res) => {
  // 关于CSRF，后面会学到，目前先提供一个模拟的值
  res.render('newsletter', { csrf: 'CSRF token goes here' })
}

exports.api = {
  newsletterSignup: (req, res) => {
    console.log('CSRF token (from hidden form field): ' + req.body._csrf)
    console.log('Name (from visible form field): ' + req.body.name)
    console.log('Email (from visible form field): ' + req.body.email)
  }
}

```

```
        res.send({ result: 'success' })
    },
}
```

可以在表单处理函数里做任何需要的操作，通常是把数据保存到数据库。如果存在问题，就发送一个包含 `err` 属性的 JSON 对象（而不是 `result: success`）。



在这个示例中，假设所有的 Ajax 请求都预期 JSON 响应，但是并不存在 Ajax 必须要用 JSON 通信这样的要求（事实上，Ajax 曾经是首字母缩写，其中“X”代表了“XML”）。使用 JSON 通信对 JavaScript 十分友好，因为 JavaScript 十分擅长处理 JSON。如果你要开放普遍可以访问的 Ajax 端点，或者你知道 Ajax 请求可能使用 JSON 以外的格式，就应该完全基于 `Accepts` 请求头来返回适当的响应（`Accepts` 请求头可以方便地通过 `req.accepts` 来访问）。如果仅基于 `Accepts` 请求头来做响应，那么或许应该看看 `res.format`，它是一个简便方法，让你可以轻松地根据客户端的预期发回适当的响应。如果是那样，当你用 JavaScript 发起 Ajax 请求时，就要确保设置了 `dataType` 或 `accepts` 属性。

8.7 文件上传

我们已经提到过，文件上传会使复杂性剧增。幸运的是，借助一些非常优秀的项目，文件上传就成了小事一桩。

对于多部分表单的处理，有 4 个既流行又健壮的项目：`busboy`、`multiparty`、`formidable` 和 `multer`。这 4 个我都用过，都很不错，但我感觉 `multiparty` 是维护得最好的，因此这里就使用它。

让我们为草地鹨旅游的度假摄影比赛创建一个文件上传的表单（views/contest/vacation-photo.handlebars）：

```
<h2>Vacation Photo Contest</h2>

<form class="form-horizontal" role="form"
      enctype="multipart/form-data" method="POST"
      action="/contest/vacation-photo/{{year}}/{{month}}">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label">Name</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
            id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
```

```
<div class="col-sm-4">
  <input type="email" class="form-control" required
    id="fieldEmail" name="email">
</div>
</div>
<div class="form-group">
  <label for="fieldPhoto" class="col-sm-2 control-label">Vacation photo</label>
  <div class="col-sm-4">
    <input type="file" class="form-control" required accept="image/*"
      id="fieldPhoto" name="photo">
  </div>
</div>
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-4">
    <button type="submit" class="btn btn-primary">Register</button>
  </div>
</div>
</form>
```

请注意，必须指定 `enctype="multipart/form-data"` 才能启用文件上传。我们也通过 `accept` 属性（这是可选属性）限制了可以上传的文件类型。

现在就要创建路由处理函数了，不过有点儿“进退两难”。我们希望能够保持轻松测试路由处理函数的能力，却会因多部分表单处理而变得复杂（就像在到达处理函数之前使用中间件处理其他类型的主体编码一样）。既然并不想自己测试多部分表单的解码（可以假设 `multiparty` 已经完全做好了），干脆就让路由处理函数更纯粹些。直接给它们传入已经处理好的信息。既然还不知道那些处理好的信息什么样，那就先从 Express 的路由开始 (`meadowlark.js`)：

```
const multiparty = require('multiparty')

app.post('/contest/vacation-photo/:year/:month', (req, res) => {
  const form = new multiparty.Form()
  form.parse(req, (err, fields, files) => {
    if(err) return res.status(500).send({ error: err.message })
    handlers.vacationPhotoContestProcess(req, res, fields, files)
  })
})
```

我们使用 `multiparty` 的 `parse` 方法把请求数据解析成了数据字段和文件。这个方法会把文件存储到服务器上的临时目录，文件的信息将以 `files` 数组的形式传回来。

因此，我们要向（可测试的）路由处理函数传入额外的信息：表单字段（不会像前面的例子那样存在于 `req.body` 中，因为使用了不同的 `body` 解析器）和收集到的文件信息。现在知道了处理好的信息是什么样的，就可以写路由处理函数了：

```
exports.vacationPhotoContestProcess = (req, res, fields, files) => {
  console.log('field data: ', fields)
  console.log('files: ', files)
```

```
    res.redirect(303, '/contest/vacation-photo-thank-you')
}
```

(年和月已经作为路由参数了，此参数将在第 14 章中学习。) 运行这个示例，检查控制台输出。你会看到表单字段如预期般出现了：它是一个对象，属性对应表单的字段名。`files` 对象包含了更多数据，但比较直白。对每个上传的文件，你可以看到的属性有文件大小、存储的路径（通常是一个临时目录下的一个随机文件名），以及用户上传时的原始文件名（只有文件名，不是完整的路径，这是由于安全和隐私的原因）。

如何操作这个文件现在完全取决于你：你可以存储进数据库、复制到一个更持久的位置，或是上传至云端的文件系统。要记住，如果你依靠本地存储来保存文件，应用的伸缩性就会受影响，对于云端的应用托管来说，这是一个糟糕的选择。第 13 章将再次审视这个示例。

使用 `fetch` 做文件上传

让人高兴的是，使用 `fetch` 做文件上传几乎跟使用浏览器来做一样。上传文件最难的一部分就是编码，而编码的工作中间件已经为我们做了。

考虑以下用 `fetch` 发送表单内容的脚本：

```
<script>
  document.getElementById('vacationPhotoContestForm')
    .addEventListener('submit', evt => {
      evt.preventDefault()
      const body = new FormData(evt.target)
      const container =
        document.getElementById('vacationPhotoContestFormContainer')
      const url = '/api/vacation-photo-contest/{{year}}/{{month}}'
      fetch(url, { method: 'post', body })
        .then(resp => {
          if(resp.status < 200 || resp.status >= 300)
            throw new Error(`Request failed with status ${resp.status}`)
          return resp.json()
        })
        .then(json => {
          container.innerHTML = '<b>Thank you for submitting your photo!</b>'
        })
        .catch(err => {
          container.innerHTML = `<b>We're sorry, we had a problem processing ` +
            `your submission. Please <a href="/newsletter">try again</a>`
        })
    })
</script>
```

要注意的重要细节就是，我们把 `form` 元素转换为了一个 `FormData` 对象，而 `fetch` 可以直接接受这个对象作为请求的 `body`。这就是全部工作了。由于这个编码跟让浏览器来处理的编码是一样的，因此我们的路由处理函数几乎也是一样的。只需要把原来的重定向改为返回一个 JSON 响应：

```
exports.api.vacationPhotoContest = (req, res, fields, files) => {
  console.log('field data: ', fields)
  console.log('files: ', files)
  res.send({ result: 'success' })
}
```

8.8 提升文件上传的UI

浏览器内建的用于文件上传的 `<input>` 控件从 UI 的角度来看可以说很简陋。你可能见过外观更吸引人的拖放界面和上传按钮。

好消息是，你在这里所学到的东西可以应用到大部分流行的“高级”文件上传组件上。归根结底，大部分组件的表单上传机制是一样的，只是在其基础上加上了漂亮的外观。

下面列出了几个最流行的文件上传前端组件：

- jQuery File Upload
- Uppy（它有个好处，就是支持多个上传目标）
- file-upload-with-preview（它给了你完全的控制。你可以访问文件对象数组，可以用它构造用于 `fetch` 的 `FormData` 对象）

8.9 小结

本章学习了表单处理的各种技术。我们探索了传统的由浏览器来处理表单的方式（让浏览器发起一个 `POST` 请求，把表单内容发送到服务器，并渲染服务器返回的响应，这响应通常是一个重定向），也探索了采用得越来越多的方式，即避免浏览器提交，自己用 `fetch` 来处理表单。

我们了解了表单编码的以下几种常见方式。

`application/x-www-form-urlencoded`

默认且易用的编码，通常与传统的表单处理相关联。

`application/json`

常见于使用了 `fetch` 的（非文件）数据发送。

`multipart/form-data`

需要传输文件时使用的编码。

我们已经讨论了如何把用户数据导入服务器，接下来让我们转到 `Cookie` 和 `Session`，它们同样有助于在服务器端和客户端之间同步数据。

第9章

Cookie和Session

本章将学习如何使用 Cookie 和 Session，通过跨页面甚至跨浏览器会话记住用户的偏好设置，为他们提供更好的体验。

HTTP 是一个无状态协议。这意味着，如果你在浏览器上加载一个页面，再导航到同一个网站的另一个页面，那么不管是服务器还是浏览器，都无法得知这是同一个浏览器在访问同一个网站。换句话说，每个 HTTP 请求都包含着服务器要满足这个请求所需的全部信息，这也正是 Web 的工作方式。

这的确是个问题。如果这个问题不能解决，我们将永远无法登入任何网页，流媒体将无法工作，而网站也将不能跨页面记住你的偏好设置。因此，需要有一种在 HTTP 之上建立状态的办法，这就是 Cookie 和 Session 出现的背景。

不幸的是，有人曾利用 Cookie 做了一些坏事，让 Cookie 落得了个坏名声。这着实不幸，因为 Cookie 对于“现代网站”的正常运行确实至关重要（尽管 HTML5 已经引入了像本地存储这样的新特性来达到相同的目的）。

Cookie 的想法很简单：服务器发送一小段信息，然后浏览器存储一段时间（可配置）。这一小段信息是什么实际由服务器决定，很多时候它只是一个唯一标识特定浏览器的 ID，使得状态得到维护。

关于 Cookie，需要重点知道以下几点。

从用户角度来说，Cookie 是没有秘密的

客户端可以查看所有服务器发送到客户端的 Cookie。你当然可以通过加密来保护内容，但基本上不需要这样做（如果你没做什么坏事的话）。我们稍后将简略讨论签名 Cookie，它可以混淆 Cookie 的内容，但在专家看来，这种方式达不到密码学意义上的安全。

用户可以删除或停用 Cookie

用户可以完全控制 Cookie，浏览器也让你能够将其逐个或批量删除。除非你在做坏事，否则没有什么合理的理由要删除 Cookie，不过在测试期间这么做是很有用的。用户也可以停用 Cookie，但这么做问题更大，因为没有了 Cookie，就只能凑合着使用最简单的 Web 应用了。

常规 Cookie 可以被篡改

每当浏览器向关联了 Cookie 的服务器发出请求，你都盲目相信 Cookie 的内容，那就为攻击提供了方便。其中最愚蠢的行为，就是执行 Cookie 中包含的代码了。要保证 Cookie 不被篡改，就要使用签名 Cookie。

Cookie 可以用于攻击

在近年悄然兴起的攻击中，有一类叫作跨站脚本（XSS）攻击。XSS 攻击有一项技术，就是利用恶意 JavaScript 脚本修改 Cookie 的内容。所以，这又是一条不要信任返回服务器的 Cookie 内容的理由。使用签名 Cookie 有助于防止 XSS 攻击（不管是用户自己还是恶意脚本对签名 Cookie 做出的修改，都是可以轻易发现的）。此外，还可以设定只有服务器才能修改的 Cookie。这些签名 Cookie 和只有服务器才能修改的 Cookie 在用途上有些限制，但它们确实更安全。

如果你滥用 Cookie，用户就会注意到

如果你在用户的计算机上设置了太多的 Cookie 条目或者存储了大量数据，那么可能会惹恼用户，这是应该避免的。要尽量少使用 Cookie。

优先使用 Session 而不是 Cookie

大多数情况下可以使用 Session 来维护状态，而且通常来说这样做更为明智。使用 Session 更简单，不仅不用担心会滥用用户的存储空间，还可以更安全。当然，Session 要依靠 Cookie，但使用 Session 的话，可以让 Express 替你做大量的工作。



Cookie 没有什么神奇的。当服务器想要客户端存储一条 Cookie 的时候，它会发送一个名为 `Set-Cookie` 的 HTTP 头，其中包含 Cookie 的名 / 值对；而当客户端向服务器发送请求的时候，它会把多个包含 Cookie 值并命名为 `Cookie` 的请求头发送过去。

9.1 提取敏感信息

要想让 Cookie 更安全，必须要有一个 Cookie 密钥。这个 Cookie 密钥是一个只有服务器端才知道的字符串，在把 Cookie 发给客户端之前先用它来加密。Cookie 密钥不是需要记住的密码，因此也可以是一个随机的字符串。我经常使用受 xkcd 启发的随机密码生成器（xkcd-password-generator）来生成 Cookie 密钥或一个随机数字。

把第三方的敏感信息提取到外部文件是很常见的操作。这些敏感信息包括 Cookie 密钥、数据库密码以及第三方 API 的 token（Twitter、Facebook 等）。这样做不仅可以减轻维护的工作量（定位和更新这些信息更容易了），而且允许版本控制系统忽略这个敏感信息文件，这对于托管在 GitHub 或其他开放的源码仓库的开源项目来说尤为重要。

既然如此，我们就把敏感信息提取到一个 JSON 文件。创建一个名为 `.credentials.development.json` 文件：

```
{  
  "cookieSecret": "……你的Cookie密钥"  
}
```

这将是我们用于开发的敏感信息配置文件。按照这种方式，可以对产品、测试，或其他环境分别准备不同的配置文件，各有各的用处。

我们准备在这个配置文件之上增加一层抽象，这样随着应用数量的增长，管理依赖可以更容易。这一层抽象非常简单。创建一个名为 `config.js` 的文件：

```
const env = process.env.NODE_ENV || 'development'  
const credentials = require(`./.credentials.${env}`)  
module.exports = { credentials }
```

现在，为了确保不会误把信息配置文件加入版本库，将 `.credentials.*` 添加到 `.gitignore` 文件中。要把配置文件导入应用，只需像下面这样做：

```
const { credentials } = require('./config')
```

这个文件以后还会存储其他敏感信息，但目前我们只需要 Cookie 密钥。



如果你是使用版本库跟着做下来的，那么需要创建自己的配置文件，因为它没包含在版本库里。

9.2 Express中的Cookie

为了能够在应用中设置和访问 Cookie，需要加入 `cookie-parser` 这个中间件。首先安装 `npm install cookie-parser`，然后执行以下操作（版本库的 `ch09/meadowlark.js`）：

```
const cookieParser = require('cookie-parser')
app.use(cookieParser(credentials.cookieSecret))
```

做完了这些，就可以在任何能够访问 `response` 对象的地方设置常规 Cookie 或签名 Cookie 了：

```
res.cookie('monster', 'nom nom')
res.cookie('signed_monster', 'nom nom', { signed: true })
```



签名 Cookie 优先于未签名 Cookie。如果你把一个签名 Cookie 命名为 `signed_monster`，就不能再让一个未签名 Cookie 使用这个名字（它会返回 `undefined`）。

要获取客户端发过来的 Cookie 值（如果有的话），只需访问 `request` 对象的 `cookie` 或 `signedCookie` 属性：

```
const monster = req.cookies.monster
const signedMonster = req.signedCookies.signed_monster
```



可以使用任何字符串命名 Cookie。例如，可以使用 `\'signed monster\'` 来代替 `\'signed_monster\'`，但这样的话就得使用方括号获取 Cookie：`req.signedCookies[\'signed monster\']`。所以，建议 Cookie 名不要包含特殊字符。

要删除一个 Cookie，可以使用 `req.clearCookie`：

```
res.clearCookie('monster')
```

在设置 Cookie 的时候，可以指定下面的选项：

domain

控制这个 Cookie 所关联的域名，允许你把 Cookie 分配给特定的子域名。注意，设置一个 Cookie 时，不能把它关联到当前服务器以外的域名，否则设置无效。

path

控制这个 Cookie 所应用的路径。注意，路径后面有隐含的通配符：如果使用路径 /（默认），它就会应用到网站的所有页面；如果使用路径 /foo，它就会应用到路径 /foo、/foo/bar，等等。

maxAge

指定客户端应该在删除 Cookie 前保存多久，单位为毫秒。如果省略此选项，浏览器关闭的时候这个 Cookie 就会被删除。（你也可以使用 `expires` 选项指定过期日期，但日期的格式会让人糊涂，推荐使用 `maxAge`。）

`secure`

指定仅通过安全连接（HTTPS）发送此 Cookie。

`httpOnly`

如果设为 `true`，则指定这个 Cookie 只能由服务器端修改。也就是说，客户端 JavaScript 不能修改它。这样配置有助于避免 XSS 攻击。

`signed`

如果设为 `true`，则指定这个 Cookie 进行签名，它会出现在 `res.signedCookies` 中，而不是 `res.cookies` 中。签名 Cookie 如果被篡改，就会被服务器丢弃，Cookie 值也会被重置为初始值。

9.3 查看Cookie

出于测试的需要，很可能你希望有办法来检查系统中的 Cookie。大多数浏览器可以查看各个网站有哪些 Cookie 以及这些 Cookie 的值。在 Chrome 中，打开开发者工具，选择“Application”标签页。在左侧列表中，你可以看到“Cookies”，展开它，就可以看到关联当前网站的所有 Cookie 了。可以右键单击域名，清除所有 Cookie，或者右键单击某一条 Cookie，只移除这一条。

9.4 Session

Session 其实是一种更方便的维护状态的方式。要实现 Session，就必须在客户端存储一些信息，否则，服务器将不能从连续的请求中识别出这是同一个客户端。通常的做法是使用一条包含唯一标识的 Cookie，而服务器使用这个标识来获取相应的 Session 信息。

Cookie 不是实现 Session 的唯一办法，在“Cookie 恐慌”达到高峰的时候（那时 Cookie 滥用严重），很多用户只不过是关掉了 Cookie，并想出了其他办法来替代，例如给 URL 附加 Session 信息。但是，这些技术都比较杂乱、难用且低效，所以最好还是让它们成为历史。HTML5 提供了实现 Session 的另一个选择，叫**本地存储**（local storage），如果需要存储大量数据，那么它比 Cookie 更有优势。更多信息请参考 MDN 关于 `window.localStorage` 的文档。

大体而言，实现 Session 有两种方式：把所有信息都存储到 Cookie 中，或者只把唯一标识存储到 Cookie 中，其余的存储到服务器上。前者叫作**基于 Cookie 的 Session**，只是代表着使用 Cookie 的一种习惯。然而，前者还意味着你在 Session 中加入的所有信息都会存储到客户的浏览器上，我并不推荐这种方式。但如果你知道只需存储少量的信息，你也并不介意用户访问这些信息，而且这些信息不会随时间推移而失去控制，那么我会推荐这种方式。如果你想采用这种方式，请查看 `cookie-session` 这个中间件。

9.4.1 内存存储

如果你更愿意把 Session 信息存储到服务器上（这也是我推荐的），就需要一个存储它的地方。入门级的选择就是内存存储。内存存储的 Session 很容易建立起来，但有一个很大的不足：只要重启服务器（在学习本书的过程中你会重启很多次的！），Session 信息就会消失。甚至还有更糟的，如果你的服务横向扩展到多台服务器上（参见第 12 章），每次请求都可能由不同的服务器来提供服务，那么结果是 Session 数据有时候存在，有时候却不存在。这显然是用户不能接受的使用体验。不过就开发和测试来说，内存存储的 Session 已经可以满足需要了。在第 13 章将看到如何持久化 Session 的存储信息。

首先，安装 `express-session` (`npm install express-session`)。然后，在链入 Cookie 解析器后链入 `express-session` (版本库的 `ch09/meadowalk.js`)：

```
const expressSession = require('express-session')
// 要确保在链入Session中间件之前链入了Cookie中间件!
app.use(expressSession({
  resave: false,
  saveUninitialized: false,
  secret: credentials.cookieSecret,
}))
```

`express-session` 中间件接收一个配置对象，这个配置对象包含以下选项。

`resave`

即使没有对 `request` 对象做修改，也要强制保存 Session 信息。通常将其设置为 `false` 更合适。更多信息请查看 `express-session` 文档。

`saveUninitialized`

将其设置为 `true` 将导致新的（未初始化的）Session 被保存，即使它们还没有被修改过。通常而言，将其设置为 `false` 更合适，尤其是当你需要先获得用户的允许才能设置 Cookie 时。更多信息请查看 `express-session` 文档。

`secret`

用于对 Session ID 这个 Cookie 进行签名的键（一个或多个），跟用于 `cookie-parser` 的键一样。

`key`

用于存储 Session 唯一标识的 Cookie 名称。默认为 `connect.sid`。

`store`

Session 存储的一个实例。默认是 `MemoryStore` 的一个实例，对我们当前的要求来说已经足够了。第 13 章将介绍如何使用数据库存储。

cookie

Session 所使用的 Cookie 的设置 (`path`、`domain`、`secure`, 等等)。默认使用常规 Cookie 的设置。

9.4.2 使用Session

一旦把 Session 设置好，使用起来就再简单不过了，直接使用 `request` 对象的 `session` 变量中的属性：

```
req.session.userName = 'Anonymous'  
const colorScheme = req.session.colorScheme || 'dark'
```

注意对于 Session，不需要分别使用 `request` 对象获取值，使用 `response` 对象设置值，都是在 `request` 对象上操作的（`response` 对象并没有 `session` 属性）。要删除一条 Session 数据，可以使用 JavaScript 的 `delete` 操作：

```
req.session.userName = null          // 把userName设置为null，但并不移除它  
delete req.session.colorScheme      // 移除colorScheme
```

9.5 使用Session实现flash消息

`flash` 消息（不要跟 Adobe 的 Flash 相混淆）只是向用户提供反馈的一种方式，不会影响用户的页面导航。要实现 `flash` 消息，最简单的办法就是使用 Session（你也可以使用查询串，但这样会让 URL 更难看，而且当用户添加书签的时候，这个消息也会包含在书签里，这些应该是你不想看到的）。先从 HTML 开始。我们将使用 Bootstrap 的 `alert` 消息来显示 `flash` 消息，所以要先保证你已经把 Bootstrap 加入进来了（请查看 Bootstrap 的“getting started”文档；你可以在主模板引入 Bootstrap 的 CSS 和 JavaScript 文件——本书版本库有一个例子）。在模板文件里某个显眼的地方（一般是网站标题栏的正下方），增加以下代码：

```
{{#if flash}}  
  <div class="alert alert-dismissible alert-{{flash.type}}">  
    <button type="button" class="close"  
      data-dismiss="alert" aria-hidden="true">&times;</button>  
    <strong>{{flash.intro}}</strong> {{{flash.message}}}  
  </div>  
&{{/if}}
```

注意我们对 `flash.message` 使用了三重花括号，这样处理就可以在消息里提供一些简单的 HTML（可能会希望凸显某些词或加入超链接）。现在我们需要一个中间件，如果 Session 中有 `flash` 对象的话，就把它加入上下文。一条 `flash` 消息只要显示了一次，我们就把它从 Session 中移除，这样它就不会在下一个请求中显示了。下面创建这个中间件，检查 Session 中有没有 `flash` 消息，如果有，就把它传给 `res.locals` 对象，让它在视图中可用。

我们把它放入名为 lib/middleware/flash.js 的文件中：

```
module.exports = (req, res, next) => {
  // 如果Session中有一条flash消息，那么将它传给上下文对象，并清除
  res.locals.flash = req.session.flash
  delete req.session.flash
  next()
})
```

然后在 meadowalk.js 文件中，需要在所有视图路由之前，将这个 flash 消息中间件链进来：

```
const flashMiddleware = require('./lib/middleware/flash')
app.use(flashMiddleware)
```

现在来看看 flash 消息究竟是怎么使用的。假设我们想要在用户完成邮件列表订阅后，将订阅页面重定向到历史邮件页面。表单处理函数大致是这样的：

```
// 根据W3C HTML5的电子邮件正则表达式稍作修改：
const VALID_EMAIL_REGEX = new RegExp('^[a-zA-Z0-9.#!#$%&\'*+\/\=?^`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9])?' +
  '(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9]{0,61}[a-zA-Z0-9])?)+$')

app.post('/newsletter', function(req, res){
  const name = req.body.name || '', email = req.body.email || ''
  // 输入验证
  if(VALID_EMAIL_REGEX.test(email)) {
    req.session.flash = {
      type: 'danger',
      intro: 'Validation error!',
      message: 'The email address you entered was not valid.',
    }
    return res.redirect(303, '/newsletter')
  }
  // NewsletterSignup只是你可能会创建的对象的一个例子；
  // 由于每个项目的实现都不一样，这些项目特定的接口要怎么写由你来决定。
  // 这里只是演示一种典型的通过Express来实现的写法
  new NewsletterSignup({ name, email }).save((err) => {
    if(err) {
      req.session.flash = {
        type: 'danger',
        intro: 'Database error!',
        message: 'There was a database error; please try again later.',
      }
      return res.redirect(303, '/newsletter/archive')
    }
    req.session.flash = {
      type: 'success',
      intro: 'Thank you!',
      message: 'You have now been signed up for the newsletter.',
    };
    return res.redirect(303, '/newsletter/archive')
  })
})
```

我们仔细区分了输入错误和数据库错误。要记住，即使已经在前端做了输入验证（前端是应该做的），在后端还是要做，因为恶意用户是可以绕过前端验证的。

flash 消息是网站可选择的、非常好的机制，尽管在某些情景中会有更合适的方法（例如，对于多表单“向导”或购物车结账流程，flash 消息往往不合适）。在开发期间，flash 消息也很有用，因为它是一种提供反馈的简便方法，即使以后你要把它替换为别的技术。在搭建一个网站时，增加 flash 消息的支持是我最先做的事情之一。在本书后面的章节中，也会使用这项技术。



因为 flash 消息是在中间件中从 Session 传到 `res.locals.flash` 的，你需要做一次重定向 flash 消息才能显示。如果你想不经重定向就显示，直接设置 `res.locals.flash`，而不是 `req.session.flash`。



本章的示例使用浏览器提交表单并重定向，因为像这样使用 Session 来控制 UI 的做法，通常是不会用在由 Ajax 提交表单的应用中的。在 Ajax 提交表单的应用中，你会希望错误信息能够在后端表单处理返回的 JSON 中体现出来，然后前端修改 DOM 以动态地显示错误信息。这并不是说 Session 不适用于前端渲染的应用，而是说 Session 很少这样使用。

9.6 Session的用途

当你想跨越多个页面保存用户的偏好设置时，Session 就很有用了。最常见的是使用 Session 来提供用户认证信息：当你登录的时候，一个 Session 就创建了。之后，每次你重新加载页面都不需要再次登录。即使不需要用到用户账号，Session 也是很有用的。通过 Session，网站可以记住你喜欢的排序方式，记住你偏爱的日期格式，这些都只是基本的功能，而且都不需要你登录。

尽管我推荐你优先选择 Session 而不是 Cookie，理解 Cookie 的运行机制还是很重要的（特别是因为它是 Session 运行的基础）。这有助于你诊断问题，并考虑应用中安全和隐私方面的一些问题。

9.7 小结

Web 的底层协议（HTTP）本来是无状态的，理解了 Cookie 和 Session 之后，就能更好地理解 Web 应用是如何让我们产生它是有状态的错觉的。我们也学习了管理 Cookie 和 Session 的一些技术，用以提升用户体验。

尽管对中间件还没有做太多的解释，我们也一直在使用它。下一章将深入中间件，把需要知道的一切都弄个明白。

第10章

中间件

目前为止，我们已经接触了不少中间件。我们已经使用过现有的中间件（`body-parser`、`cookie-parser`、`static` 和 `express-session`，稍作列举），甚至还写了自己的中间件（把天气数据加到模板的上下文、配置 flash 消息和 404 处理函数）。可是，中间件究竟是什么呢？

从概念上讲，中间件是封装功能的一种方式，具体来说这种功能就是操作向应用发起的 HTTP 请求。在实际应用中，中间件只是一个接收 3 个参数的函数，这 3 个参数分别为：`request` 对象、`response` 对象和 `next()` 函数（`next` 函数一会儿再解释）。（还有一种用于错误处理的接收 4 个参数的形式，本章最后再讨论。）

中间件是在所谓的管线中执行的。你可以将其想象成现实中的一根水管。水从管子的一端泵上来，在流到最终目的地之前会经过水压计和阀门。这个类比的关键点是顺序很重要：如果你把水压计放在阀门前面，那么跟放在阀门后面相比效果完全不同。与此类似，如果你有一个阀门可以向水里注入某种东西，那么这个阀门“下游”的水都会包含加入的这种物质。在 Express 应用中，通过调用 `app.use` 可以把中间件插入管线中。

Express 4.0 之前的版本必须明确地链入 `router`¹，这就使管线复杂化了。链入 `router` 的位置，可能会导致路由的顺序错乱，当你混用中间件和路由处理函数时，管线的序列就不那么清楚了。到了 Express 4.0，中间件和路由处理函数是按照它们链入的顺序调用的，这个序列就清楚多了。

对于未匹配到其他任何路由的请求，使用一个兜底函数来处理，并把这个函数作为管线的最后一个中间件，这是常见的操作。这个中间件通常返回状态码 404（页面未找到）。

注 1：即 `app.use(app.router)` 这行代码，4.0 之后不需要。——译者注

那么在管线中，一个请求是如何“终结”的呢？这就轮到传入每个中间件的 `next` 函数发挥作用了：如果你没有调用 `next()`，请求就终结于这个中间件。

10.1 基本原理

要理解 Express 如何工作，关键是要学习灵活地思考中间件和路由处理函数。以下几点需要牢记。

- 路由处理函数（`app.get`、`app.post` 等，常常统称为 `app.METHOD`）可以被认为是只处理特定 HTTP 动词（GET, POST 等）的中间件。反过来，中间件也可以被认为是处理所有 HTTP 动词的路由处理函数（基本上等同于 `app.all`，它也处理所有 HTTP 动词。对于某些少见的动词，如 PURGE，中间件跟 `app.all` 的处理效果有微小的差异，但对于常用的动词，二者处理效果一样）。
- 路由处理函数需要一个路径作为第一个参数。如果你想让这个路径匹配任何路由，就使用 `*`。中间件也可以接收一个路径作为第一个参数，但这是可选的（如果略去，它将匹配任何路径，就像指定了 `*` 一样）。
- 路由处理函数和中间件接收一个回调函数，这个回调函数可以接收 2 个、3 个或 4 个参数（从技术上来说，你也可以让它没有或只有一个参数，但这种形式没有什么用）。如果有 2 个或 3 个参数，前 2 个参数就是 `request` 和 `response` 对象，第三个参数就是 `next` 函数。如果有 4 个参数，这个回调函数就成了错误处理中间件，第一个参数就成了错误对象，紧接着是 `request`、`response` 和 `next` 这些对象。
- 如果你没有调用 `next()`，管线就会终结，不再有路由处理函数和中间件的处理。如果是这样，你应该发送一个响应到客户端（`res.send`、`res.json`、`res.render` 等），否则客户端会一直等待，直到超时。
- 如果你的确调用了 `next()`，那么发送响应到客户端通常是不妥当的。如果你发送了响应，管线下游的中间件或路由处理函数就会被执行，但这些函数发送的任何客户端响应都会被忽略。

10.2 中间件示例

如果你想实际操作一下，可以尝试几个简单的例子（版本库的 `ch10/00-simple-middleware.js`）：

```
app.use((req, res, next) => {
  console.log(`processing request for ${req.url}...`)
  next()
})

app.use((req, res, next) => {
  console.log('terminating request')
  res.send('thanks for playing!')
  // 注意这里没有调用next()，这样就终结了这个请求
```

```
})

app.use((req, res, next) => {
  console.log(`whoops, i'll never get called!`)
})
```

这里举了3个中间件的例子。第一个中间件在调用`next()`把请求传给管线中下一个中间件之前，只是在控制台记录了一条消息。第二个中间件实际处理了请求。注意如果省略了`res.send`，就不会有响应发送到客户端，最终，客户端会超时。最后一个中间件永远不会得到执行，因为所有请求都在前一个中间件终结了。

现在来看一个更复杂且完整的例子（版本库的`ch10/01-routing-example.js`）：

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
  console.log('\n\nALWAYS')
  next()
})

app.get('/a', (req, res) => {
  console.log('/a: route terminated')
  res.send('a')
})
app.get('/a', (req, res) => {
  console.log('/a: never called');
})
app.get('/b', (req, res, next) => {
  console.log('/b: route not terminated')
  next()
})
app.use((req, res, next) => {
  console.log('SOMETIMES')
  next()
})
app.get('/b', (req, res, next) => {
  console.log('/b (part 2): error thrown' )
  throw new Error('b failed')
})
app.use('/b', (err, req, res, next) => {
  console.log('/b error detected and passed on')
  next(err)
})
app.get('/c', (err, req) => {
  console.log('/c: error thrown')
  throw new Error('c failed')
})
app.use('/c', (err, req, res, next) => {
  console.log('/c: error detected but not passed on')
  next()
})
```

```

app.use((err, req, res, next) => {
  console.log('unhandled error detected: ' + err.message)
  res.send('500 - server error')
})

app.use((req, res) => {
  console.log('route not handled')
  res.send('404 - not found')
})

const port = process.env.PORT || 3000
app.listen(port, () => console.log(`Express started on http://localhost:${port}` +
  `; press Ctrl-C to terminate.`))

```

在试运行这个例子之前，先想象一下结果会是什么。这些不同的路由分别是什么？客户端会看到什么？控制台上会记录什么？如果你能正确回答所有这些问题，就算是掌握了 Express 的路由。要特别注意到 /b 和到 /c 这两个请求的区别，这两种情形都产生了错误，但一个导致了 404，另一个导致了 500。

注意中间件必须是一个函数。要知道在 JavaScript 中，从一个函数返回另一个函数很容易（并且很常见）。例如，你会注意到 `express.static` 是一个函数，但我们实际调用了它，因此它必须返回另一个函数。思考下面的代码：

```

app.use(express.static)      // 不会按预期工作

console.log(express.static()) // 会打印出"function", 指示express.static是一个函数,
                            // 它本身也会返回一个函数

```

而且也要注意，模块可以导出一个函数，这个函数可以直接用作中间件。例如，这里有一个名为 `ib/tourRequiresWaiver.js` 的模块（草地鹨旅游的攀岩套餐要求一个免责声明）：

```

module.exports = (req,res,next) => {
  const { cart } = req.session
  if(!cart) return next()
  if(cart.items.some(item => item.product.requiresWaiver)) {
    cart.warnings.push('One or more of your selected ' +
      'tours requires a waiver.')
  }
  next()
}

```

可以像下面这样链入这个中间件（版本库的 `ch10/02-item-waiver.example.js`）：

```

const requiresWaiver = require('./lib/tourRequiresWaiver')
app.use(requiresWaiver)

```

不过，更常见的是导出一个包含中间件属性的对象。例如，我们把购物车的所有校验代码都放入 `lib/cartValidation.js` 中：

```

module.exports = {

  resetValidation(req, res, next) {
    const { cart } = req.session
    if(cart) cart.warnings = cart.errors = []
    next()
  },

  checkWaivers(req, res, next) {
    const { cart } = req.session
    if(!cart) return next()
    if(cart.items.some(item => item.product.requiresWaiver)) {
      cart.warnings.push('One or more of your selected ' +
        'tours requires a waiver.')
    }
    next()
  },

  checkGuestCounts(req, res, next) {
    const { cart } = req.session
    if(!cart) return next()
    if(cart.items.some(item => item.guests > item.product.maxGuests )) {
      cart.errors.push('One or more of your selected tours ' +
        'cannot accommodate the number of guests you ' +
        'have selected.')
    }
    next()
  },
}

```

然后可以像下面这样链入中间件（版本库的 ch10/03-more-cart-validation.js）：

```

const cartValidation = require('./lib/cartValidation')

app.use(cartValidation.resetValidation)
app.use(cartValidation.checkWaivers)
app.use(cartValidation.checkGuestCounts)

```



在前面的例子中，我们使用了语句 `return next()` 让中间件提前终止。Express 并不期望中间件返回一个值（而且它不会对返回值做任何操作），所以这是 `next(); return` 更简略的写法。

10.3 常用中间件

npm 上有数以千计的中间件项目，其中只有一小部分是基础而常用的，而在这部分中，有一些在每个正规的 Express 项目中都能见到。有些中间件实在太常用了，以至于过去它们实际是随着 Express 一起发布的，不过现在早已经移入独立的包里了。到现在仍然随着 Express 一起发布的中间件只剩下 `static` 了。

下面的列表基本涵盖了最常用的中间件。

`basicauth-middleware`

提供 basic 访问授权。要记住 basic 授权只能提供最基本的安全保障，而且应该仅通过 HTTPS 使用它（否则，用户名和密码就是明文传输的）。只有当你需要快速、简易地做出东西，并且需要使用 HTTPS 时，才应该使用 basic 授权。

`body-parser`

提供对 HTTP 请求 body 的解析。这个库提供了多个中间件，分别解析 URL 编码、JSON 编码和其他编码的 body。

`busboy`、`multiparty`、`formidable`、`multer`

这些中间件都可以解析编码为 `multipart/form-data` 的请求 body，从中选择一个使用。

`compression`

使用 gzip 或 deflate 压缩响应数据。这可是个好东西，你的用户一定会感谢你的，尤其是那些网速慢或使用手机流量的用户。它应该尽早链入，在任何可能会发送响应的中间件之前链入。我只推荐在 `compression` 之前链入调试或日志中间件（它们不会发送响应）。注意在大多数生产环境中，压缩是由像 NGINX 这样的代理来做的，因此这个中间件就没有必要了。

`cookie-parser`

提供 Cookie 支持。参见第 9 章。

`cookie-session`

提供 Cookie 存储的 Session 支持。我通常不推荐这种方式的 Session。它应该在 `cookie-parser` 之后链入。参见第 9 章。

`express-session`

提供 Session ID（作为一条 Cookie）的 Session 支持。默认使用内存存储，但内存存储不适合生产环境，因此可以配置为使用数据库存储。参见第 9 章和第 13 章。

`csurf`

提供针对跨站请求伪造（CSRF）攻击的保护。它要使用 Session，因此必须在 `express-session` 中间件之后链入。不过，仅仅链入这个中间件并不能神奇地保护网站免受 CSRF 攻击。更多信息参见第 18 章。

`serve-index`

为静态文件提供目录列表支持。除非你需要目录列表，否则不需要包含这个中间件。

`errorhandler`

为客户端提供栈跟踪和错误信息。不建议在生产服务器上链入，因为它暴露了实现细节，也可能会带来安全或隐私方面的不利后果。更多信息参见第 20 章。

`serve-favicon`

提供 favicon 文件服务（显示在浏览器标题栏上的图标）。它不是必需的，你也可以简单地把 favicon.ico 放入静态资源的根目录，但这个中间件可以提升性能。如果要使用它，就应该把它链入中间件栈靠前的位置。它也允许使用 favicon.ico 以外的文件名。

`morgan`

提供自动化的日志支持，所有请求都会记录在日志中。更多信息参见第 20 章。

`method-override`

提供 `x HttpMethod-Override` 请求头的支持，这个请求头允许浏览器“伪装”成使用 GET 和 POST 以外的 HTTP 方法，对于调试来说很有用。仅当你写 API 的时候才会需要这个中间件。

`response-time`

把 `X-Response-Time` 头加到响应里，提供以毫秒记的响应时间。除非你在做性能调优，否则通常不需要这个中间件。

`static`

提供静态文件（公开文件）服务的支持。你可以多次链入这个中间件，以便指定多个不同的目录。更多信息参见第 17 章。

`vhost`

虚拟主机（vhost），这是从 Apache 借过来的术语，在 Express 中可以使子域名管理更加容易。更多信息参见第 14 章。

10.4 第三方中间件

对于第三方中间件，当前并没有一个完整的“仓库”或索引。不过，可以在 npm 上访问到大部分 Express 中间件。因此，如果你在 npm 上搜“Express”和“middleware”，那么得到的列表会很长。Express 的官方文档也包含一个中间件列表，对我们很有帮助。

10.5 小结

本章深入学习了中间件是什么、怎样写自己的中间件，以及中间件作为 Express 应用的一部分是怎样运行的。如果你认为一个 Express 应用只不过是一系列中间件的集合，那你就开始理解 Express 了。即便是我们一直在使用的路由处理函数，也不过是一种特殊类型的中间件。

下一章，我们的目光将投向另一个常用的基础设施需求：发送邮件（你最好能想到，肯定又会涉及一些中间件）。

第 11 章

发送邮件

电子邮件是应用跟外界通信的主要途径之一。从用户注册到密码重置指引再到促销邮件，发送邮件是网站的一个重要功能。本章将学习如何在 Node 和 Express 中格式化和发送邮件，从而帮助你更好地跟用户沟通。

不管是 Node 还是 Express，都没有内建邮件发送功能，所以需要使用第三方模块。我要推荐的是 Andris Reinman 的杰作 Nodemailer。在开始配置 Nodemailer 之前，先来了解一下电子邮件的一些基础知识，以扫清障碍。

11.1 SMTP、MSA和MTA

发送邮件的通用语言是简单邮件传输协议（SMTP）。尽管使用 SMTP 直接发送邮件到接收者的邮件服务器是可行的，但除非是像谷歌或雅虎这样的受信任发送者，否则这么做通常并不明智，因为你的邮件很可能会被直接扔进垃圾箱。最好是使用一个邮件发送代理（MSA），它会通过受信任的通道投递邮件，从而减小邮件被标记为垃圾邮件的可能性。除了保证邮件送达，MSA 也可以处理诸如临时故障和邮件退回这些麻烦事。最后一个邮件投递代理（MTA），它提供将邮件投递到最终目的地的服务。但对本书而言，MSA、MTA 和 SMTP 服务器本质上是等效的。

接下来，你需要访问一个 MSA。尽管也可以直接使用 Gmail、Outlook 或雅虎这样的免费邮件服务，但是这些服务对自动化邮件已经没有过去那么友好了（为了尽量减少滥用）。幸运的是，有两个非常出色的邮件服务（SendGrid 和 Mailgun）可供选择，它们对低使用量提供免费服务。这两个服务我都使用过，并且很喜欢。本书的示例将使用 SendGrid。

如果你现在在一个组织内工作，组织本身可能已经有 MSA 了，那么可以联系 IT 部门，询问他们是否有适用于发送自动化邮件的 SMTP 转发服务可用。

如果准备使用 SendGrid 或 Mailgun，那么现在就去设置一下你的账号。使用 SendGrid 前，你需要先创建一个 API key（它将是你的 SMTP 密码）。

11.2 接收邮件

大多数网站只要求能够发送邮件，诸如发送密码重置指引和促销邮件。有些应用也要求能够接收邮件。问题跟踪系统就是一个很好的例子。每当有人提出一个问题，系统就会发出一封邮件，如果你回复了这封邮件，系统就自动以你回复的内容来回复这个问题。

不幸的是，接收邮件比发送邮件要复杂很多，所以本书并不打算涉及这个部分。如果你的確需要接收邮件，就应该让你的邮件服务商维护收件箱，然后使用一个 IMAP 代理（如 imap-simple）定期访问收件箱。

11.3 邮件头

一封电子邮件由两部分组成：邮件头和邮件主体（跟 HTTP 请求非常相像）。邮件头包含关于邮件的信息：发件人、收件人、收件日期、主题，等等。在邮件应用中，这些头信息通常都会显示给用户，但除了这些还有很多其他的头信息。大多数邮件客户端允许你查看邮件头，如果从来没看过，建议你看一看。这些邮件头可以告诉你这封邮件是如何送到你这里的，邮件所经过的每个服务器和 MTA 都会在邮件头中列出来。

很多人会感到奇怪：为什么有些邮件头可以由发送者任意设置，比如发件人地址“from”？当把发件人地址指定为不同于发件人账号的地址时，就是常说的邮件欺诈。谁也无法阻止你发送一封发件人地址是比尔·盖茨 “billg@microsoft.com”的邮件。不过我可不建议你这么尝试，这里只是为了说明你可以对邮件头设置任意的值。有时候这么做是有正当理由的，但绝不该滥用。

然而，你发送的邮件必须包含这个“from”地址。当发送自动化的邮件时，“from”地址有时会引发问题，这就是为什么你常常看到邮件的回复地址是“请勿回复 do-not-reply@meadowlarktravel.com”。是使用这样的发件地址，还是让自动化邮件使用像“Meadowlark Travel info@meadowlarktravel.com”这样的地址，由你来决定。不过，如果要使用后一种地址，就要做好回应那些发送到“info@meadowlarktravel.com”的邮件的准备。

11.4 邮件格式

当互联网还是新鲜事物时，邮件只有 ASCII 文本。但这个世界变化很快，人们开始希望

能以各种语言发送邮件，希望邮件能做更多高级的事情，比如包含格式化文本、图片和附件。这时候事情就开始变得难办了。邮件格式和编码使得各种技术和标准陷入一片混乱。

幸运的是，我们不需要解决这些复杂问题，Nodemailer 可以帮我们解决。你只需要知道，你的邮件既可以是普通文本（Unicode），也可以是 HTML。

所有现代的邮件应用都支持 HTML 邮件，因此把邮件格式化为 HTML 通常来说是相当安全的。不过，目前还是有一些排斥 HTML 邮件的“纯文本主义者”，因此，建议总是同时包含纯文本邮件和 HTML 邮件。如果你不喜欢写两种，那么 Nodemailer 可以提供便利，它支持由 HTML 自动生成纯文本版本。

11.5 HTML邮件

关于 HTML 邮件的内容足够写一本书了。遗憾的是，HTML 邮件并不像为网站写 HTML 那么简单，大多数邮件客户端仅支持 HTML 的一个小子集。这意味着大多数时候你写 HTML 邮件的感觉就好像还在 1996 年一样，没有什么乐趣。尤其是你不得不像过去那样，使用 table 做布局。

如果你处理过 HTML 的浏览器兼容性问题，就知道那是一件令人头痛的事。但电子邮件的兼容性问题更令人头痛。好在有一些好东西可以帮助处理这个问题。

首先，建议你读读 MailChimp 的优秀文章 “About HTML Email”。它阐述了一些基本原则以及在写 HTML 邮件时需要牢记的东西。

接着，我会给你一个节省时间的利器：HTML 邮件样板。这里面基本都是写得很好并经过严格测试的 HTML 邮件模板。

最后是测试。你已经了解了如何写 HTML 邮件，也使用了 HTML 邮件样板，但是只有通过测试，你才能确认你的邮件会不会在 Lotus Notes 7（是的，人们还在用它）上一塌糊涂。为了测试一封邮件，是否需要安装 30 个不同的邮件客户端呢？我觉得不需要。幸运的是，有一个非常好的服务可以帮你做这些工作，它就是 Litmus。这个服务不便宜，费用大约是每月 100 美元起。但是如果你要发送很多促销邮件，则非常划算。

另外，如果你的邮件格式很简单，就没有必要使用像 Litmus 这样价格不菲的测试服务了。如果严格限定了诸如标题、粗体 / 斜体文本、水平分隔线以及图片链接这样的格式，那你大可不必花这个钱。

11.6 Nodemailer

首先，需要安装 Nodemailer 包：

```
npm install nodemailer
```

然后，导入 Nodemailer 包，并创建一个 Nodemailer 实例（以 Nodemailer 的说法叫 transport）：

```
const nodemailer = require('nodemailer')

const mailTransport = nodemailer.createTransport({
  auth: {
    user: credentials.sendgrid.user,
    pass: credentials.sendgrid.password,
  }
})
```

请注意，我们使用了在第 9 章创建的 credentials 模块。你需要相应地更新 .credentials.development.json 文件：

```
{
  "cookieSecret": "your cookie secret goes here",
  "sendgrid": {
    "user": "your sendgrid username",
    "password": "your sendgrid password"
  }
}
```

SMTP 的常用配置项是端口、认证类型和 TLS 选项。不过，大多数主要的邮件服务使用默认配置。想要知道有什么设置，请参考邮件服务的文档（试着搜索“sending SMTP email”“SMTP configuration”或“SMTP relay”）。如果发送 SMTP 邮件时遇到了麻烦，或许应该检查一下配置项。要了解完整的配置项列表，请查阅 Nodemailer 文档。



如果你是跟着本书版本库操作的，就会发现在 credentials 文件中并没有任何设置。很多读者曾联系我，问为什么没有这个文件或为什么文件里面是空的。你应当小心你的 credentials 文件！我有意不提供合法的 credentials 设置，理由也一样。亲爱的读者，虽然我非常信任你，但也不至于把邮箱密码给你。

11.6.1 发送邮件

现在我们有了一个 Nodemailer 实例，可以发送邮件了。先从一个简单的例子开始，发送只有一个收件人的纯文本邮件（版本库的 ch11/00-smtp.js）：

```
try {
  const result = await mailTransport.sendMail({
    from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
    to: 'joecustomer@gmail.com',
    subject: 'Your Meadowlark Travel Tour',
    text: 'Thank you for booking your trip with Meadowlark Travel. ' +
      'We look forward to your visit!',
  })
}
```

```
        })
        console.log('mail sent successfully: ', result)
    } catch(err) {
        console.log('could not send mail: ' + err.message)
    }
}
```



在本节的代码示例里，我使用了虚构的邮件地址，比如 joecustomer@gmail.com。为了能够验证，你或许应该把它换成真实的邮件地址，这样才能看到发生了什么。否则，可怜的 joecustomer@gmail.com 就要收到大量无意义的邮件了，而你什么也看不到。

你应该会注意到这里做了错误处理。但是没有错误并不意味着邮件已经成功发送至收件人的邮箱，理解这一点很重要。回调的 `error` 参数仅在跟 MSA 通信出现问题时才会设置（比如网络错误或身份认证失败）。如果这封邮件 MSA 发送不出去（比如邮件地址不合法或用户不存在），你就得在邮件服务中检查一下账户活动，这种检查可以通过管理界面或 API 进行。

如果你想让系统自动确认邮件是否已经成功发送，就需要使用邮件服务的 API。详情请查阅邮件服务的 API 文档。

11.6.2 发送给多个收件人

Nodemailer 支持发送邮件给多个收件人，收件人用逗号分隔（版本库的 `ch11/01-multiple-recipients.js`）：

```
try {
    const result = await mailTransport.sendMail({
        from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
        to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
            'fred@hotmail.com',
        subject: 'Your Meadowlark Travel Tour',
        text: 'Thank you for booking your trip with Meadowlark Travel. ' +
            'We look forward to your visit!',
    })
    console.log('mail sent successfully: ', result)
} catch(err) {
    console.log('could not send mail: ' + err.message)
}
```

请注意，这个例子中混合使用了普通邮件地址（`joe@gmail.com`）和指定收件人名字的邮件地址（“Jane Customer” `jane@yahoo.com`）。后者也是合理的语法。

当发送给多个收件人时，你必须注意 MSA 的限制。例如，SendGrid 会建议限制收件人的数量（SendGrid 建议一封邮件的收件人数不要超过 1000）。如果你要群发邮件，可能要多次发送，每次包含多个收件人（版本库的 `ch11/02-many-recipients.js`）：

```

// largeRecipientList是一个邮件地址的数组
const recipientLimit = 100
const batches = largeRecipientList.reduce((batches, r) => {
  const lastBatch = batches[batches.length - 1]
  if(lastBatch.length < recipientLimit)
    lastBatch.push(r)
  else
    batches.push([r])
  return batches
}, [[]])
try {
  const results = await Promise.all(batches.map(batch =>
    mailTransport.sendMail({
      from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
      to: batch.join(', '),
      subject: 'Special price on Hood River travel package!',
      text: 'Book your trip to scenic Hood River now!',
    })
  ))
  console.log(results)
} catch(err) {
  console.log('at least one email batch failed: ' + err.message)
}

```

11.7 群发邮件更好的选择

毫无疑问，你可以使用 Nodemailer 和一个合适的 MSA 来群发邮件，但是这么做之前需要深思熟虑。负责任的邮件群发必须提供一种让用户可以取消订阅促销邮件的方法，这可不是一项简单的任务。尤其是当你要维护多个订阅列表的时候（例如，你有一个每周通讯的订阅和一个特别公告的订阅），复杂性就成倍增加了。所以，在这方面最好还是不要浪费时间。像 Emma、Mailchimp 和 Campaign Monitor 这些服务提供了你想要的所有东西，已经包括了一套很好的监控群发是否成功的工具。这些服务都不太贵，强烈推荐使用它们来发送促销、业务通讯等群发邮件。

11.8 发送HTML邮件

目前为止，我们还是只能发送纯文本的邮件，但是大多数人希望邮件能够更美观一些。Nodemailer 允许你在一封邮件里同时发送 HTML 和纯文本版本，由客户端来选择要显示哪个版本（通常会选 HTML）（版本库的 ch11/03-html-email.js）：

```

const result = await mailTransport.sendMail({
  from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
  to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
    'fred@hotmail.com',
  subject: 'Your Meadowlark Travel Tour',
  html: '<h1>Meadowlark Travel</h1><n>p>Thanks for book your trip with ' +
    'Meadowlark Travel. <b>We look forward to your visit!</b>',

```

```
    text: 'Thank you for booking your trip with Meadowlark Travel. ' +
          'We look forward to your visit!',
  })
}
```

同时提供 HTML 和纯文本工作量太大了，尤其是当只有少数用户选择纯文本邮件时，同时提供两种形式似乎有些多余。如果你想节省一些时间，可以只写 HTML，然后使用像 html-to-formatted-text 这样的包从 HTML 自动生成文本。（不过要记住，HTML 转换的文本质量跟手写出来的比不了，不会像手写那样整齐、清晰。）

11.8.1 HTML邮件中的图片

虽然可以在 HTML 邮件中嵌入图片数据，但是我非常不赞成这么做。它们会使邮件变得臃肿，通常不认为这是一种好做法。你应该把邮件中要使用的图片放到你的 Web 服务器上，然后在邮件中适当地链接到这些图片。

在静态资源目录下，最好为邮件的图片创建专用的目录。你也应该单独存放那些同时用于网站和邮件的资源，这样便可以降低对邮件布局造成不利影响的可能性。

让我们在草地鹨旅游项目中增加一些邮件的资源。在 public 目录下创建子目录 emai。你可以把 logo.png 放进去，也可以把其他想用于邮件的图片放进去。然后，就可以直接在邮件里使用那些图片了：

```

```



在给其他人发邮件时，你显然不会再使用 localhost，因为他们的机器可能没有运行中的服务器，更别说在 3000 端口运行了。邮件能否运行取决于邮件客户端，或许你能够在测试中使用 localhost，但离开你的计算机就无效了。第 17 章将讨论从开发平稳过渡到生产的一些技术。

11.8.2 使用视图来发送HTML邮件

目前为止，我们都是把 HTML 放入 JavaScript 的字符串中，但这种做法是你应当避免的。我们这么做是因为涉及的 HTML 都十分简单，但是看一看前面介绍的“HTML 邮件样板”，你还想把那样的样板代码整个放入一个字符串中吗？你绝对不想。

幸运的是，可以利用视图来解决。回想“感谢你从草地鹨旅游预订了一次旅行”邮件的例子，我们对此稍微解释一下。设想我们有一个包含订单信息的购物车对象，这个对象会被存放在 Session 里。假设我们下订单的最后一个步骤是一个表单，这个表单在由 /cart/checkout 处理之后会发送一封确认邮件。让我们先来创建这个感谢页的视图，即 views/cart-thank-you.handlebars：

```

<p>Thank you for booking your trip with Meadowlark Travel,
{{cart.billing.name}}!</p>
<p>Your reservation number is {{cart.number}}, and an email has been
sent to {{cart.billing.email}} for your records.</p>

```

然后为这封邮件创建一个模板。下载 HTML 邮件样板并放到 views/email/cart-thank-you.handlebars 下。编辑这个文件，修改 body 部分的内容：

```

<table cellpadding="0" cellspacing="0" border="0" id="backgroundTable">
<tr>
<td valign="top">
<table cellpadding="0" cellspacing="0" border="0" align="center">
<tr>
<td width="200" valign="top"></td>
</tr>
<tr>
<td width="200" valign="top"><p>
    Thank you for booking your trip with Meadowlark Travel,
    {{cart.billing.name}}.</p><p>Your reservation number
    is {{cart.number}}.</p></td>
</tr>
<tr>
<td width="200" valign="top">Problems with your reservation?
    Contact Meadowlark Travel at
    <span class="mobile_link">555-555-0123</span>.</td>
</tr>
</table>
</td>
</tr>
</table>

```



因为在邮件中不能使用 localhost 作为地址，所以如果你的网站还未上线，则可以先使用一个占位服务来显示图片。例如，可以使用 http://placeholder.it/100x100 来动态提供一个 100 像素的矩形图像。这项技术经常被用于占位 (FPO) 图片和布局设计。

现在可以为感谢页创建路由了（版本库的 ch11/04-rendering-html-email.js）：

```

app.post('/cart/checkout', (req, res, next) => {
  const cart = req.session.cart
  if(!cart) next(new Error('Cart does not exist.'))
  const name = req.body.name || '', email = req.body.email || ''
  // 输入验证
  if(!email.match(VALID_EMAIL_REGEX))
    return res.next(new Error('Invalid email address.'))
  // 分配一个随机订单ID，正常情况下会使用数据库的ID
  cart.number = Math.random().toString().replace(/^0\.\d*/,'')
  cart.billing = {

```

```
        name: name,
        email: email,
    }
    res.render('email/cart-thank-you', { layout: null, cart: cart },
      (err,html) => {
        console.log('rendered email: ', html)
        if(err) console.log('error in email template')
        mailTransport.sendMail({
          from: '"Meadowlark Travel": info@meadowlarktravel.com',
          to: cart.billing.email,
          subject: 'Thank You for Book your Trip with Meadowlark Travel',
          html: html,
          text: htmlToFormattedText(html),
        })
        .then(info => {
          console.log('sent! ', info)
          res.render('cart-thank-you', { cart: cart })
        })
        .catch(err => {
          console.error('Unable to send confirmation: ' + err.message)
        })
      }
    )
  })
```

注意，我们调用了 `res.render` 两次。正常情况下，我们只会调用 `res.render` 一次（调用两次也只会显示第一次调用的结果）。然而在这个情形中，我们规避了第一次调用的正常渲染过程。注意，我们提供了一个回调函数。如此就防止了渲染后的视图被发送到浏览器，取而代之的是，它作为 `html` 参数被传入了回调函数。我们只需用提供的这个 HTML 发送邮件就可以了。我们指定了 `layout: null` 来避免使用布局文件，因为邮件模板中包含了一切（另一种做法是创建一个用于邮件的单独布局文件并使用它）。最后，再次调用 `res.render`。这次，视图结果会正常作为 HTML 响应发送到浏览器。

11.8.3 封装邮件功能

如果你的网站有很多地方要使用邮件，也许你会希望对邮件的功能做一下封装。假设你想让网站总是用同一个发送者（“Meadowlark Travel” info@meadowlarktravel.com）发送邮件，并且总是发送 HTML 格式，同时带上自动生成的纯文本。那么，创建一个名为 lib/email.js 的模块（版本库的 ch11/lib/email.js）：

```
const nodemailer = require('nodemailer')
const htmlToFormattedText = require('html-to-formatted-text')

module.exports = credentials => {

  const mailTransport = nodemailer.createTransport({
    host: 'smtp.sendgrid.net',
    auth: {
```

```
        user: credentials.sendgrid.user,
        pass: credentials.sendgrid.password,
    },
})

const from = '"Meadowlark Travel <info@meadowlarktravel.com>"'
const errorRecipient = 'youremail@gmail.com'

return {
  send: (to, subject, html) =>
    mailTransport.sendMail({
      from,
      to,
      subject,
      html,
      text: htmlToFormattedText(html),
    }),
}
}
```

如果现在要发送一封邮件，那么只需要这么做（版本库的 ch11/05-email-library.js）：

```
const emailService = require('./lib/email')(credentials)

emailService.send(email, "Hood River tours on sale today!",
  "Get 'em while they're hot!")
```

11.9 小结

本章学习了如何在互联网上发送电子邮件。如果你跟着操作了，那么应该已经建立起了免费邮件服务（很可能是 SendGrid 或 Mailgun），并使用这个服务发送了文本和 HTML 邮件。你也学习了如何使用模板来渲染邮件的 HTML，这跟在 Express 应用中渲染 HTML 使用的是同样的机制。

电子邮件仍然是你的应用跟用户沟通的重要方式。但是注意不要过度使用。如果你和我一样，都有一个被自动发送的邮件占据的收件箱，就会明白这些邮件大多数会被忽略。对自动化邮件来说，少即是多。即使有正当理由向用户发送邮件，你也应该问问自己：“用户真的需要这封邮件吗？没有第二种方式可以传达这些信息了吗？”

我们已经讨论了创建应用所需要的一些基础设施，下面花些时间来讨论一下应用的最终发布，以及为了成功发布所需考虑的问题。

第 12 章

考慮生产环境中的问题

或许你觉得在目前这个节点就开始讨论生产环境中的问题为时过早，不过如果你提前考虑，就可以为将来节省大量时间，免去很多麻烦。何况发布到生产环境的日子往往不知不觉就到了。

本章将学习 Express 如何为不同运行环境提供支持、扩展网站的方式以及如何监控网站的运行状态。你会了解如何在测试和开发中模拟生产环境，也会学到如何通过压力测试提前发现生产环境中的问题。

12.1 运行环境

Express 支持运行环境的概念，让你的应用可以在生产、开发或测试模式下运行。实际上，你想要多少种不同的环境都可以。比如说，你可以有一个准生产环境或一个培训环境。然而要记住，开发环境、生产环境和测试环境都属于“标准”环境，Express 和第三方中间件常常会根据这些环境做决策。换句话说，如果你有一个“准生产”环境，那么是没有办法让它自动继承生产环境的属性的。为此，建议你还是使用标准的生产、开发和测试环境。

尽管可以通过调用 `app.set('env', 'production')` 来指定运行环境，但这很不明智，因为这意味着不论是什么情况，你的应用都会在之前指定的环境中运行。更糟糕的是，应用启动时是一个环境，而后又切换到了另一个环境。

比较好的做法是使用环境变量 `NODE_ENV` 来指定运行环境。让我们通过调用 `app.get('env')` 修改一下应用，让它报告它所在的模式：

```
const port = process.env.PORT || 3000
app.listen(port, () => console.log(`Express started in ` +
  `${app.get('env')} mode at http://localhost:${port}` +
  `; press Ctrl-C to terminate.`))
```

如果你现在启动应用，就会看到它正在开发模式下运行，这是默认模式。尝试一下让它在生产模式下运行：

```
$ export NODE_ENV=production
$ node meadowlark.js
```

如果使用 Unix/BSD，就有一种便捷的写法，可以让你仅在这个命令的运行期间修改环境：

```
$ NODE_ENV=production node meadowlark.js
```

这会让服务器仅在生产模式下运行，当服务器终止时，`NODE_ENV` 环境变量将不会被修改。我特别喜欢这种便捷的写法，它降低了偶然误设环境变量的可能性，毕竟误设的值未必需要全部应用。



如果你以生产模式启动 Express，可能会出现一些警告，提示有些组件不适用于生产模式。如果你是跟着本书示例操作的，就会发现 `connect.session` 使用的是内存存储，不适用于生产环境。等到第 13 章我们把它切换为数据存储后，这条警告就会消失。

12.2 特定环境的配置

虽然单独修改运行环境好像没有太大作用，但是在生产模式下，Express 会打印更多的警告信息（比如通知你某些模块已经淘汰了，并且将被移除），而视图缓存也是默认开启的（参见第 7 章）。

大体而言，运行环境是你的工具，能够帮你轻松地决定应用在不同环境中的行为。特别提醒，应该尽量让开发环境、测试环境和生产环境的差异最小化。也就是说，你应该少用环境支持这个特性。如果开发或测试环境跟生产环境差异很大，就会增大在生产环境中产生不同行为的可能性，从而产生更多（或更难查找）的缺陷。不过，有些差异不可避免。例如，如果你的应用是由数据库高度驱动的，你就不会希望在开发期间混用生产数据库，这就很适合采用特定环境的配置。另一个例子是更详尽的日志，对于生产其影响就没有那么大了。你希望在开发环境中记录的很多东西，到了生产环境中就不必记录了。

下面给服务器增加一些日志输出。与以往不同，我们想让生产和开发具有不同的行为。对于开发，我们使用默认配置，但对于生产，我们希望能够记录到文件中。我们将使用 `morgan`（别忘了执行 `npm install morgan`），它是最常用的日志中间件（版本库的 `ch12/00-logging.js`）：

```
const morgan = require('morgan')
const fs = require('fs')

switch(app.get('env')) {
  case 'development':
    app.use(morgan('dev'))
    break
  case 'production':
    const stream = fs.createWriteStream(__dirname + '/access.log',
      { flags: 'a' })
    app.use(morgan('combined', { stream }))
    break
}
```

如果你像往常那样启动服务器（`node meadowlark.js`）并访问网页，就可以看到记录到控制台的活动。要想看看日志输出在生产模式下是什么样子，只需在启动服务器时带上 `NODE_ENV=production`。这时如果再访问网页，在终端上就不会看到任何活动了（对于生产服务器，这很可能是我们所期望的），所有的活动都会以 Apache 的组合日志格式记录到文件中（很多服务器工具会生成这种格式的日志）。

这是怎么实现的？我们先创建了一个可追加的（`{ flags: 'a'}`）写入流，然后把它传入 `morgan` 的配置对象中。`morgan` 有很多配置项，更多信息请参考它的官方文档。



在前面的例子中，我们使用了 `__dirname` 把请求日志存储到项目本身的子目录下。如果选择这种做法，记得把 `log` 加入 `.gitignore` 文件。或者你也可以采用更符合 Unix 习惯的做法，就像 Apache 的默认做法一样，把日志存储到 `/var/log` 的一个子目录下。

我想再次强调，在决定哪些是特定环境的配置时，你应该根据实际情况自己做出最佳的判断。始终要记住，你的网站在线时将（或应该）在产品模式下运行。所以每当你忍不住想做出一些特定于开发模式的修改时，应该始终首先考虑在生产环境中这些修改可能会对 QA 产生什么样的影响。关于特定环境的配置，第 13 章中将介绍一个更健壮的示例。

12.3 运行Node进程

目前为止，我们都是直接使用 `node`（比如 `node meadowlark.js`）来启动应用的。对于开发和测试来说，这样做很好，但对于生产来说，其存在一定的不足。很显然，如果应用崩溃或终止了，它不会采取什么保护措施。而使用一个健壮的进程管理可以解决这个问题。

对于应用的托管服务，如果它本身已经提供了进程管理服务，你可能就不需要再考虑了。也就是说，托管服务会给你提供一个配置项，指向你的应用文件，然后它会负责进程管理。

如果你需要自己管理进程，有以下两个主流的选择：

- Forever
- PM2

由于生产环境各种各样，因此我们不会深入讨论初始化和配置进程管理的种种细节。Forever 和 PM2 有很好的参考文档，你可以在开发机器上安装并学习如何配置它们。

二者我都使用过，而且没有特别倾向于哪一个。初次上手的话，Forever 更为直接和易用一些，但 PM2 支持更多的特性。

如果你想体验一下进程管理而又不想投入太多时间，建议试试 Forever。只需要两步。第一步，安装 Forever：

```
npm install -g forever
```

第二步，通过 Forever 启动应用（在应用根目录下运行）：

```
forever start meadowlark.js
```

现在应用开始运行了。即使你把终端窗口关掉，它也仍在运行。你可以使用 `forever restart meadowlark.js` 来重启进程，使用 `forever stop meadowlark.js` 来结束进程。

上手 PM2 稍微麻烦一些，但如果生产环境需要使用自己的进程管理，那么深入研究一下是很值得的。

12.4 网站的扩展

目前而言，扩展通常分为两种：垂直扩展和水平扩展。垂直扩展指的是使服务器更强大：更快的 CPU、更好的架构、更多的核心、更大的内存，等等。而水平扩展只是意味着更多的服务器。随着云计算越来越流行，虚拟化越来越普遍，服务器的计算能力也逐渐变得不那么重要了。要想根据需求来扩展网站，水平扩展通常是最经济的办法。

当你开发 Node 网站的时候，应该一直考虑水平扩展的可能性。即使你的网站很小（可能只是一个用户有限的内网应用），而且你从没有想过需要水平扩展，多考虑水平扩展的可能性也是一个好习惯。毕竟，说不定你接下来的 Node 项目会成为下一个 Twitter 呢。那么水平扩展将至关重要。幸运的是，Node 支持水平扩展，我们在编码时留意一下扩展问题也并不费力。

当你构建一个最初就要水平扩展的网站时，最需要注意的是持久化。如果你习惯于依靠文件存储来实现持久化，那么赶紧停止吧，这是愚蠢的做法。

我第一次处理这个问题的经历几乎是灾难性的。我们的一位客户曾在网上举办了一个竞赛，网站本应当通知前 50 名获胜者，告诉他们将收到奖品。由于客户公司 IT 条件的限制，要使用数据库不太容易，因此大多数的持久化是直接通过写文件来实现的。我像往常

那样把获胜者都写入一个文件里。一旦文件已经记录了 50 个获胜者，就不能再增加了。但问题在于，服务器做了负载均衡，因此有一半的请求会被其中一台服务器处理，另一半则会被另一台服务器处理。结果就是这台服务器通知了 50 个人说他们获胜了，而另一台也通知了 50 个人……不过好在这些奖品价值不高，不是 iPad 之类的贵重物品。客户也欣然当了冤大头，拿出了 100 份而不是 50 份奖品（我提出要自掏腰包为多出的 50 条毯子买单，但他们大度地拒绝了）。

我从这次经历中得到的教训是，除非你有一个所有服务器都能访问的文件系统，否则不应该依靠本地文件系统来实现持久化。不过，只读数据（比如日志和备份）是个例外。例如，我就经常把表单提交数据备份到本地文件，以防数据库连接失败。毕竟，当数据库中断时，要从每一台服务器收集这些文件是很麻烦的，至少这样做不会有什么损失。

12.4.1 使用应用集群实现水平扩展

Node 本身支持应用集群，这是一种简单的、单服务器的水平扩展形式。使用应用集群，你可以为机器的每个核心（CPU）创建一个独立的应用服务器实例（服务器实例超过核心数并不会提升应用的性能）。应用集群有两大好处：首先，它有助于最大化给定服务器（实体机器或虚拟机器）的性能；其次，这是在并行条件下测试应用的低成本方式。

现在给我们的网站增加集群支持。尽管我们通常在主应用文件里完成所有工作，还是要创建第二个应用文件，它将使用之前一直使用的非集群应用文件在集群中运行应用。为此，需要先对 meadowlark.js 做一些修改（版本库的 ch12/01-server.js 是一个简化的例子）：

```
function startServer(port) {
  app.listen(port, function() {
    console.log(`Express started in ${app.get('env')} mode on http://localhost:${port} + ` +
      `; press Ctrl-C to terminate.`)
  })
}

if(require.main === module) {
  // 应用直接运行，启动应用服务器
  startServer(process.env.PORT || 3000)
} else {
  // 应用要通过"require"导入为一个模块。这里导出创建服务器的函数
  module.exports = startServer
}
```

回想一下，在第 5 章中，`require.main === module` 表示脚本是直接运行的；如果二者不相等，则表示是在其他的脚本中通过 `require` 调用的。

然后，创建一个名为 meadowlark-cluster.js 的新脚本（版本库的 ch12/01-cluster 是一个简化例子）：

```

const cluster = require('cluster')

function startWorker() {
  const worker = cluster.fork()
  console.log(`CLUSTER: Worker ${worker.id} started`)
}

if(cluster.isMaster){

  require('os').cpus().forEach(startWorker)

  // 记录所有断开连接的worker。
  // 如果一个worker断开了连接，接着就会退出，
  // 所以我们就等待退出事件，之后生成一个新的worker来取代它
  cluster.on('disconnect', worker => console.log(
    `CLUSTER: Worker ${worker.id} disconnected from the cluster.`
  ))

  // 当一个worker死亡（退出）时，创建一个新的worker来取代它
  cluster.on('exit', (worker, code, signal) => {
    console.log(
      `CLUSTER: Worker ${worker.id} died with exit ` +
      `code ${code} (${signal})`
    )
    startWorker()
  })
}

} else {

  const port = process.env.PORT || 3000
  // 在worker进程上运行我们的应用，参见01-server.js
  require('./01-server.js')(port)

}

```

如果要执行这个 JavaScript 脚本，要么是在 master 进程的上下文中执行（如果它是通过 `node meadowlark-cluster.js` 直接运行的话），要么是在 worker 进程的上下文中执行，此时是 Node 的集群系统在工作。通过属性 `cluster.isMaster` 和 `cluster.isWorker` 可以确定我们正处于哪个上下文中。运行的这个脚本是在 master 模式下执行的。对系统中的每个 CPU，也使用 `cluster.fork` 来启动一个 worker。同时，我们也在监听 worker 的 `exit` 事件，一旦发现有任何 worker 退出，就创建一个新的。

最后，在 `else` 子句中，我们处理的是 worker 的情形。既然已经将 `meadowlark.js` 配置为一个模块，那么只需简单地导入并直接调用它（别忘了，我们已把它导出为一个用来启动服务器的函数了）。

现在启动新的集群服务器：

```
node meadowlark-cluster.js
```



如果你在使用虚拟机（比如 Oracle 的 VirtualBox），则可能需要给它配置多个 CPU，因为虚拟机往往默认只有一个 CPU。

假设你在使用一个多核系统，你应该能看到有几个 worker 进程启动了。如果想看到不同 worker 在处理不同请求的证据，就在路由之前加上以下中间件：

```
const cluster = require('cluster')

app.use((req, res, next) => {
  if(cluster.isWorker)
    console.log(`Worker ${cluster.worker.id} received request`)
  next()
})
```

现在你可以通过浏览器连接应用了。重新加载几次，就可以看到每次请求都从 worker 池里获得了一个不同的 worker。（Node 就是用来处理大基数的连接的，简单地刷新浏览器是不能给它充分施压的。后面我们会深入研究压力测试，那时就能看到这个集群在实战中的表现了。）

12.4.2 处理未捕获的异常

在 Node 的异步世界中，未捕获的异常需要特别关注。我们先从一个不会引发太大问题的简单例子开始（建议你也跟着这些小例子试一试）：

```
app.get('/fail', (req, res) => {
  throw new Error('Nope!')
})
```

Express 在执行路由处理函数时会把它们包含在 try/catch 块里，因此这个例子实际上不是一个未捕获的异常，也不会引起太大的问题：Express 会在服务器端记录这个异常，而网页访问者会看到一个丑陋的异常栈。然而服务器是稳定的，其他的请求也会继续得到正确的处理。如果你想提供一个好看一些的错误页，就创建一个名为 views/500.handlebars 的文件，然后在所有的路由之后增加一个错误处理：

```
app.use((err, req, res, next) => {
  console.error(err.message, err.stack)
  app.status(500).render('500')
})
```

提供定制错误页的做法很好。当错误发生时，定制的错误页不仅会让你的用户觉得更专业，而且也允许你采取一些措施。例如，可以通过错误处理函数通知你的开发团队错误发生了。不幸的是，这仅对那些 Express 能够捕获的异常适用。来看一下更糟糕的情形：

```
app.get('/epic-fail', (req, res) => {
  process.nextTick(() =>
    throw new Error('Kaboom!')
  )
})
```

试着运行一下，结果绝对是灾难性的：它让服务器崩溃了！不仅未能向用户显示一条温馨的错误提示，连服务器都终止了，无法再响应任何请求。这是因为 `setTimeout` 是异步执行的，执行这个异常的函数会被推迟到 Node 空闲的时候。问题在于，到了 Node 空闲并开始执行这个函数时，当初要响应的请求对应的上下文已经不复存在了。Node 已经处在未定义的状态，因此只好简单粗暴地关闭整个服务器。（Node 并不知道这个函数或其调用者的意图，所以它不会认为后面的函数还能正常工作。）



`process.nextTick` 类似于 `setTimeout` 传入参数 0，但它更高效。这里只是为了演示，通常不会在服务器端代码中使用它。然而后面的章节中会涉及很多异步执行的函数，比如数据库访问、文件系统访问、网络访问等，执行时都会遭受未捕获的异常这个问题的困扰。

就算我们有处理未捕获的异常的方式也无济于事，因为如果 Node 不能确定应用的稳定性，那么你也不能。换句话说，如果出现了未捕获的异常，唯一能做的就是关闭服务器。这种情形下，我们能做的也只是尽可能关闭得优雅一些，再使用某种故障转移机制。最简单的故障转移机制是使用集群。如果应用在集群模式下运行，每当其中一个 worker 退出了，master 就会再创建一个来取代它。（你甚至不需要多个 worker，因为对于故障转移来说，一个 worker 的集群就够了，尽管会稍微慢一点儿。）

既然如此，当遇到未捕获的异常时，怎样做到尽可能关闭得优雅一些呢？对此，Node 的机制就是 `uncaughtException` 事件（Node 也有一个名为 `domains` 的机制，但这个模块已经被淘汰了，不再推荐使用）。

```
process.on('uncaughtException', err => {
  console.error('UNCAUGHT EXCEPTION\n', err.stack);
  // 在这里做一些必要的清理工作，例如关闭数据库连接
  process.exit(1)
})
```

期望你的应用永远不会出现未捕获的异常并不现实，因此应该准备好一种应对机制并认真对待，当未捕获的异常发生时，这种机制可以进行记录并通知开发者。找出未捕获异常的发生原因后，你才能修正它。像 Sentry、Rollbar、Airbrake 和 New Relic 这样的服务，是记录这些类型的错误以用于日后分析的良好选择。比如说使用 Sentry，首先要注册一个免费的账号，注册时你会获得一个数据源名（DSN），然后就可以修改异常处理函数了。

```
const Sentry = require('@sentry/node')
Sentry.init({ dsn: '**YOUR DSN GOES HERE **' })
```

```
process.on('uncaughtException', err => {
  // 在这里做一些必要的清理工作，例如关闭数据库连接
  Sentry.captureException(err)
  process.exit(1)
})
```

12.4.3 使用多台服务器完成水平扩展

尽管通过集群来实现水平扩展可以最大化单台服务器的性能，但如果你需要更多的服务器呢？这个时候事情就会变得更复杂一些。为了实现这种并行，你需要一个代理服务器。（它常被叫作反向代理或面向转发的代理，以便与通常用于访问外网的代理区分开，但我觉得这种叫法既让人糊涂又没必要，所以就简单地把它叫作代理。）

当今两大主流代理服务器是 NGINX（发音“engine X”）和 HAProxy。尤其是 NGINX，用户增长非常快。最近我给我们公司做了一个竞品分析，发现我们 80% 以上的竞争对手在使用 NGINX。NGINX 和 HAProxy 都是稳定的高性能代理服务器，即便是最苛刻的应用，它们都能支持。（如果你想要证据，Netflix 就是一个很好的例子，它占了整个互联网流量的 15%，而它使用的正是 NGINX。）

还有几个小一些的基于 Node 的代理服务器，比如 node-http-proxy。如果你的要求不是太高，或者是用于开发，轻量级代理服务器是很好的选择。对于生产来说，我还是推荐使用 NGINX 或 HAProxy（两者都是免费的，不过它们也提供付费服务）。

安装和配置代理服务器超出了本书范畴，但这些可能并不像你想象中那么困难（尤其是使用 node-http-proxy 或别的轻量级代理时）。目前我们已经使用了集群，说明我们的网站已经做好水平扩展的准备了。

如果你要配置代理服务器，记得告诉 Express 你在使用代理服务器，并且这个代理服务器是可以信任的：

```
app.enable('trust proxy')
```

这行代码可以保证 `req.ip`、`req.protocol` 和 `req.secure` 体现的是客户端和这个代理服务器之间的连接，而不是客户端和应用之间的连接。而且，它也保证了 `req.ips` 会是一个数组，指示最初客户端的 IP 和所有中间代理服务器的主机名或 IP。

12.5 监控网站的运行

监控网站的运行是你可以采取的最重要的 QA 措施之一，也是最常被忽视的措施。凌晨 3 点起床修复网站 bug 已经够糟糕的了，但比它还糟糕的是凌晨 3 点被老板叫醒说网站崩溃了（或者还有更糟糕的，比如早上你来到公司才得知，网站停机一整晚，但谁都没有注意

到，而你们的客户为此损失了 10 000 美元的销售额)。

网站发生故障是你控制不了的事，它们就像死亡和税收一样不可避免。然而，你力所能及的、能让老板和客户信服的方式是：永远先于他们知道故障发生了。

第三方在线监控

在网站服务器上运行一个在线监控，就相当于给没有人居住的屋子安装一个烟雾报警器，十分有用。如果是某个页面不正常，你要捕获错误或许还有可能，但如果是整个服务器停机了，那么它甚至都不能向外界发出一个“求救信号”，更不用说捕获错误了。这就是为什么你的第一道防线应该是第三方的在线监控。UptimeRobot 可以免费支持多达 50 个监控，而且配置起来很简单。警告方式有邮件、手机短信、Twitter 和 Slack (以及其他办公产品)。对于一个网页，你可以检查它的返回码 (除了 200，其他返回码都被认为是错误的)，或者通过确认某个关键字是否存在来实现监控。不过要记住，如果你使用关键字监控，就可能会影响到网站的流量分析 (在大多数流量分析服务中，你可以把来自在线监控的流量排除)。

12.6 压力测试

压力测试（或负载测试）的作用就在于给予你一定的信心，让你看到你的服务器能在数百个或数千个并行请求之下运转如常。这是另一个很深奥的领域，可以写一本书了。压力测试可以很复杂，而复杂程度很大程度上取决于项目本质。如果你确信你的网站将极其火爆，就需要在压力测试上投入更多的时间。

让我们使用 Artillery 增加一个简单的压力测试。首先安装 Artillery：`npm install -g artillery`，然后编辑 package.json 文件，在 scripts 一节添加以下代码：

```
"scripts": {  
  "stress": "artillery quick --count 10 -n 20 http://localhost:3000/"  
}
```

这将模拟 10 个“虚拟用户” (`--count 10`)，每个将向服务器发送 20 个请求 (`-n 20`)。

确保应用正在运行（例如在另一个终端窗口运行），然后运行 `npm run stress`。你会看到像下面这样的统计信息：

```
Started phase 0, duration: 1s @ 16:43:37(-0700) 2019-04-14  
Report @ 16:43:38(-0700) 2019-04-14  
Elapsed time: 1 second  
  Scenarios launched: 10  
  Scenarios completed: 10  
  Requests completed: 200  
  RPS sent: 147.06
```

```
Request latency:  
  min: 1.8  
  max: 10.3  
  median: 2.5  
  p95: 4.2  
  p99: 5.4  
Codes:  
  200: 200  
  
All virtual users finished  
Summary report @ 16:43:38(-0700) 2019-04-14  
  Scenarios launched: 10  
  Scenarios completed: 10  
  Requests completed: 200  
  RPS sent: 145.99  
  Request latency:  
    min: 1.8  
    max: 10.3  
    median: 2.5  
    p95: 4.2  
    p99: 5.4  
  Scenario counts:  
    0: 10 (100%)  
Codes:  
  200: 200
```

这个测试是在我的开发笔记本计算机上运行的。你可以看到，Express 响应任何请求的时间都不超过 10.3 毫秒，其中 99% 的请求不超过 5.4 毫秒。至于你的预期应该是什么样的数值，我没法提供具体的参考，但是为了保证应用比较精悍，你的预期总连接时间应该保持在 50 毫秒以下。（别忘了这只是服务器要把数据发送到客户端所花的时间，客户端还需要时间渲染它，因此数据传输的时间越少越好。）

如果你定期对应用进行压力测试，并跟基准数据对照，就能够发现问题。如果你刚刚完成了一个新特性，结果发现连接时间飙升到了 3 倍，或许就要对这个新特性做一点儿性能调优了。

12.7 小结

当你的应用临近发布时，需要考虑哪些问题？希望本章除能给你带来一些思考，并能帮助你深入理解这些问题。对于一个产品级应用来说，细节问题很多，而产品发布时会发生什么，你也不能全部预想到，但是预想得越多，结果就会越好。毕竟 Louis Pasteur 说过，机会是留给有准备的人的。

第13章

持久化

除了最简单的网站和 Web 应用，其他所有网站和 Web 应用都需要某种形式的持久化。也就是说，只有拥有比易失的内存存储更持久的方式，你的数据才能在经历服务器崩溃、停电、系统升级和数据迁移之后仍然完好无损。本章将首先讨论持久化的各种选择，然后分别对文档数据库和关系型数据库做一些演示。不过在进入数据库之前，我们先从最基本的持久化方式，即文件系统持久化开始。

13.1 文件系统持久化

实现持久化的一种方式是直接把数据保存到所谓的扁平文件里（之所以用扁平来形容，是因为文件内并没有结构，它只是字节的序列）。在 Node 中实现文件系统持久化要使用 `fs`（文件系统）模块。

文件系统持久化有一些不足，尤其是它的伸缩性不太好。随着网站流量增长，当你需要更多的服务器时，除非所有的服务器都能访问到一个共享的文件系统，否则持久化就会暴露问题。此外，因为扁平文件本身没有结构，所以数据的定位、排序和筛选都要在你的应用上进行，这对应用造成了负担。由于这些原因，你应该首先考虑用数据库而不是文件系统来存储数据。但有一个例外是存储二进制文件，比如图片、音频或视频。尽管很多数据库能处理二进制数据，但很少能比文件系统更高效（不过为了能够搜索、排序和筛选，这些二进制文件的相关信息通常是存储在数据库里的）。

如果确实需要存储二进制数据，那么要记得文件系统存储始终存在伸缩性不好的问题。如果应用的托管服务不能访问到一个共享的文件系统（经常会这样），就应该考虑把二进制

文件存储到数据库（通常需要做一点儿配置，这样数据库就不会停止运行了），或者使用云存储服务，比如亚马逊的 S3 或微软的 Azure 存储。

说完了注意事项，现在来看看 Node 对文件系统的支持。继续第 8 章写的度假摄影比赛的代码。在名为 lib/handlers.js 的文件中补全那个表单处理函数（版本库的 ch13/00-mongodb/lib/handlers.js）：

```
const pathUtils = require('path')
const fs = require('fs')

// 创建保存度假照片的目录（如果还没创建的话）
const dataDir = pathUtils.resolve(__dirname, '..', 'data')
const vacationPhotosDir = pathUtils.join(dataDir, 'vacation-photos')
if(!fs.existsSync(dataDir)) fs.mkdirSync(dataDir)
if(!fs.existsSync(vacationPhotosDir)) fs.mkdirSync(vacationPhotosDir)

function saveContestEntry(contestName, email, year, month, photoPath) {
    // TODO……很快就会加上
}

// 我们会在后面使用这些promise风格的fs函数
const { promisify } = require('util')
const mkdir = promisify(fs.mkdir)
const rename = promisify(fs.rename)

exports.api.vacationPhotoContest = async (req, res, fields, files) => {
    const photo = files.photo[0]
    const dir = vacationPhotosDir + '/' + Date.now()
    const path = dir + '/' + photo.originalFilename
    await mkdir(dir)
    await rename(photo.path, path)
    saveContestEntry('vacation-photo', fields.email,
        req.params.year, req.params.month, path)
    res.send({ result: 'success' })
}
```

这里面有很多东西，我们分开来讲。首先创建一个目录以存储上传的文件（如果还没创建的话）。也许你想把 data 目录加入 .gitignore 文件中，这样就不会意外提交上传的文件了。回想一下第 8 章，我们在 meadowlark.js 里处理文件上传并用已经解码的文件调用了处理函数，得到的是关于上传文件的信息的一个对象（files）。为了避免文件名冲突，不能直接使用用户上传时的文件名（万一两个用户都上传了名为 portland.jpg 的文件呢）。因此，我们根据时间戳创建一个唯一目录。毕竟，两个用户都在 1 毫秒内上传名为 portland.jpg 的文件是极不可能的。然后把上传的文件（文件上传模块会给它一个临时文件名，可以从 path 属性获得）重命名（移动）为我们构造的文件名。

最后，还需要把用户上传的文件跟用户的邮箱地址（以及提交的年份、月份）关联起来。当然，把这些信息编码进文件名或目录名也可以，但我们还是选择把它们存储到数据库里。由于还没有学习数据库操作，因此先把这个功能封装进 vacationPhotoContest 函数，

到本章后面再完成它。



一般来说，你永远都不应该信任用户上传的东西，因为它可能会成为攻击你的网站的媒介。例如，恶意用户可以轻易地获取有害的可执行文件，将其以.jpg 后缀重命名，并上传上去，作为攻击的第一步（企图以后能找到机会再执行它）。与此类似，这里直接使用浏览器提供的 name 属性来命名文件也是要冒一点儿风险的，因为可能会有人利用这一点，在文件名里面插入一些特殊字符。要让代码完全安全，应该给文件随机的文件名，并且只接受一个扩展名（确保只包含字母和数字）。

即使文件系统持久化有其不足之处，但是它频繁被用于中间文件的存储，因此知道如何使用 Node 的文件系统库是很有用的。不过，为了弥补文件系统持久化的缺陷，让我们来看看云持久化。

13.2 云持久化

云持久化越来越流行，其提供的服务既便宜又稳定，强烈建议你将它们利用起来。

要使用这些云服务，需要做一些预备工作。创建账号自不用说，你还要理解应用跟云服务是如何认证身份的，另外理解一些基本术语也是很有用的（例如，AWS 把文件存储的单元叫作 bucket，而 Azure 将它们叫作 container）。详细阐述这些信息超出了本书的范畴，何况官方也有很好的参考文档。

- AWS: Getting Started in Node.js
- Azure for JavaScript and Node.js Developers

好消息是，一旦你完成初始的设置，使用云持久化就很简单了。下面是把文件保存到亚马逊的 S3 的例子，看看有多么简单：

```
const filename = 'customerUpload.jpg'

s3.putObject({
  Bucket: 'uploads',
  Key: filename,
  Body: fs.readFileSync(__dirname + '/tmp/' + filename),
})
```

更多细节请查看 AWS 的 SDK 文档。

下面是使用微软的 Azure 执行相同操作的例子：

```
const filename = 'customerUpload.jpg'

const blobService = azure.createBlobService()
```

```
blobService.createBlockBlobFromFile('uploads', filename, __dirname +  
    '/tmp/' + filename)
```

更多细节请查看微软的 Azure 的文档。

我们已经了解了文件存储的几项技术，现在考虑结构化数据的存储：使用数据库。

13.3 数据库持久化

除了最简单的网站和 Web 应用，其他的网站和 Web 应用都需要一个数据库。即使你的数据是二进制的，并且已经在使用共享文件系统或者云存储，很可能你也希望有一个数据库来帮助归类这些二进制数据。

传统上，**数据库**这个词是**关系型数据库管理系统（RDBMS）**的简称。关系型数据库，比如 Oracle、Mysql、PostgreSQL 或 SQL Server，都是以几十年的研究和形式化的数据库理论为基础的，目前数据库这项技术已经非常成熟，这些数据库的威力也是无可置疑的。然而，现在我们可以扩展对数据库组成的概念。近年来 NoSQL 数据库成了新风尚，它们正在改变互联网数据存储的方式。

声称 NoSQL 数据库要优于关系型数据库是愚蠢的，但是 NoSQL 数据库也有其优势（关系型数据库也有）。尽管在 Node 应用中整合关系型数据库已经相当容易，但 NoSQL 数据库简直就是为 Node 而生的。

两种最流行的 NoSQL 数据库类型是**文档数据库**和**键值数据库**。文档数据库擅长对象的存储，这使得它们天然适合于 Node 和 JavaScript。键值数据库，正如其名，非常简单，对于那些数据模式很容易映射成键值对的应用来说是非常好的选择。

在关系型数据库的约束和键值数据库的简单性之间，我认为文档数据库代表着两者的平衡点。既然如此，我们的第一个示例就先使用文档数据库。MongoDB 是领先的文档数据库，已经考验，十分稳定。

第二个示例，我们将使用 PostgreSQL，这是一个广受欢迎的、健壮的开源关系型数据库。

13.3.1 关于性能的提醒

NoSQL 数据库的简单性是一把双刃剑。对关系型数据库进行仔细调优是一项非常复杂的工作，但可以带来出色的性能。如果认为 NoSQL 数据库通常更简单，对它们的调优没有什么技术含量可言，那就错了。

传统上，关系型数据库的高性能得益于它们严格的数据结构以及数十年的优化研究。而 NoSQL 数据库和 Node 一样加入了互联网的分布式特性，而且为了高伸缩性，它更专注于并行性能（关系型数据库也支持并行，但这通常只用在对数据库性能要求极高的应用上）。

为数据库的高性能和高伸缩性做规划和调优是一个既庞大又复杂的专题，超出了本书讨论的范畴。如果你的应用要求极高的数据库性能，建议从《MongoDB 权威指南（第 3 版）》¹入手。

13.3.2 数据库层抽象

本书将演示使用两个数据库来实现同样的功能（不仅是两个名字不同的数据库，还是两种根本不同的数据库架构）。尽管目的是讨论两种主流的数据库架构选择，但也是真实的场景：在 Web 应用开发过程中切换某个主要组件。这样做的原因有很多种，不过通常可以归结为：在开发过程中发现另一项技术成本收益更大，或可以更快速地实现必要的特性。

尽可能把各种技术抽象出来是很有价值的。在这里把技术抽象出来指的是写某种类型的 API 层，以实现各种底层技术一般化。如果做得好，就可以大幅降低更换有问题的组件的成本。不过，它也是有成本的，因为这个抽象层本身要写出来并维护。

好在我们的抽象层很小，因为从本书的目的出发，只需要支持有限几个特性。目前而言，支持的特性如下。

- 从数据库返回已上线的度假产品的列表。
- 存储用户的电子邮件地址，这些用户希望在产品上市时得到通知。

虽然看起来简单，但还是有很多细节的。一款度假产品看起来是什么样的？我们是每次都从数据库获取全部的度假产品，还是希望能够对它们进行筛选或分页？该如何唯一标识一款度假产品呢？诸如此类。

从本书的目的出发，我们需要让这个抽象层尽可能地简单。把抽象层放进名为 db.js 的文件中，并导出两种方法。对这两种方法先提供模拟实现：

```
module.exports = {
  getVacations: async (options = {}) => {
    // 定义模拟的度假数据：
    const vacations = [
      {
        name: 'Hood River Day Trip',
        slug: 'hood-river-day-trip',
        category: 'Day Trip',
        sku: 'HR199',
        description: 'Spend a day sailing on the Columbia and ' +
          'enjoying craft beers in Hood River!',
        location: {
          // 在本书后面，我们要对它们进行地理编码
          search: 'Hood River, Oregon, USA',
        },
      },
    ];
    return vacations;
  }
};
```

注 1：此书即将由人民邮电出版社出版，详情请见 ituring.cn/book/2043。

```
        price: 99.95,
        tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
        inSeason: true,
        maximumGuests: 16,
        available: true,
        packagesSold: 0,
    },
]
// 如果指定了available，那么只返回匹配的度假产品
if (options.available !== undefined)
    return vacations.filter(({ available }) => available === options.available)
return vacations
},
addVacationInSeasonListener: async (email, sku) => {
    // 我们就认为它做完了。
    // 因为这是一个异步函数，所以会自动返回一个新的promise,
    // 这个promise会被解析为undefined
},
}
}
```

从应用来看，数据库应该怎样实现？对此，上面的接口给出了一个预期。我们需要做的就是让数据库符合这个接口。请注意，我们引入了度假产品“可用”的概念，这样是为了便于临时停用度假产品，而不必从数据库删除该产品。举个例子，一个提供住宿加早餐的小旅馆联系你说，他们因装修需要停业几个月，因此需要停用他们的度假产品。把“可用”跟“上市”的概念区分开是因为我们可能会希望在网站上列出未上市的度假产品，毕竟人们喜欢提前计划。

我们的数据库中也包含了一些泛泛的位置信息，到第 19 章会具体使用它们。

现在我们已经有了一个抽象的数据库层，再来看看如何使用 MongoDB 实现数据库存储。

13.3.3 设置MongoDB

在不同操作系统上建立起一个 MongoDB 实例难度各不相同。因此，我们干脆完全避开这个问题，使用一个非常出色的免费 MongoDB 托管服务——mLab。



mLab 并非唯一可用的 MongoDB 服务。现在 MongoDB 公司本身也有提供免费和低成本的数据库托管服务的 MongoDB Atlas。不过，免费账号不建议用于生产的目的。mLab 和 MongoDB Atlas 都提供生产级的账号，可以先看看它们的定价再做选择。当你要切换到生产时，使用同一个托管服务麻烦会少一些。

使用 mLab 很简单，只需打开它的主页，点击“Sign Up”按钮，填好注册表单然后登录，就进入你的主页了。在 Databases 下面，你会看到“no databases at this time”。点击“Create new”，将跳转到创建新数据库的页面，其中包含一些选项。第一个选项是云提供商。对

于免费（沙箱）账号来说，基本上怎么选都没关系，不过你应该找距离比较近的数据中心（然而并非每个数据中心都支持沙箱账号）。选择 SANDBOX，并选择一个地区，然后选择一个数据库名，点击“Submit Order”（即使它是免费的，也会有一个订单）。页面将跳转到你的数据库列表，几秒之后，你的数据库就可以使用了。

建立数据库只是完成了一半工作，还要知道如何在 Node 中访问它。这便是 Mongoose 发挥作用的地方了。

13.3.4 Mongoose

虽然 MongoDB 有比较底层的驱动可以使用，但很可能你还是希望使用一个对象文档映射（ODM）。适合 MongoDB 的主流 ODM 就是 Mongoose。

JavaScript 的一个优势是其对象模型极其灵活。你如果想给一个对象增加属性或方法，只管去做，无须担心会因此而修改对象所属的类。不幸的是，这种“随心所欲”会对数据库产生负面影响，数据会因此变得零散，难以优化。因此，Mongoose 选择通过引入模式和模型（两者结合起来，就类似于传统的面向对象编程中的类）达成一个平衡。它的模式是灵活的，但仍然可以为你的数据库提供必要的结构。

在开始之前，要先安装 Mongoose 模块：

```
npm install mongoose
```

然后把数据库的认证信息加入名为 .credentials.development.json 的文件中：

```
"mongo": {  
    "connectionString": "your_dev_connection_string"  
}
```

你可以在 mLab 的数据库页面找到连接字符串。在主界面上点击这个数据库，就会看到一个方框，里面是 MongoDB 的连接 URI（以 mongodb:// 开头）。你还需要为数据库创建用户。先点击 Users，然后点击“Add database user”。

也可以创建文件 .credentials.production.js，为生产系统建立另一套认证信息（运行时使用环境变量 NODE_ENV=production）——可以等网站上线时再做。

现在已经完成配置，让我们实际连接到数据库，做点儿有用的事情。

13.3.5 使用 Mongoose 连接数据库

先从创建数据库连接开始。把数据库初始化代码放到 db.js 中，里面还有之前创建的模拟 API（版本库的 ch13/00-mongodb/db.js）：

```

const mongoose = require('mongoose')
const { connectionString } = credentials.mongo
if(!connectionString) {
  console.error('MongoDB connection string missing!')
  process.exit(1)
}
mongoose.connect(connectionString)
const db = mongoose.connection
db.on('error' err => {
  console.error('MongoDB error: ' + err.message)
  process.exit(1)
})
db.once('open', () => console.log('MongoDB connection established'))

module.exports = {
  getVacations: async () => {
    // .....返回模拟的度假产品数据
  },
  addVacationInSeasonListener: async (email, sku) => {
    // .....什么也不做
  },
}

```

任何文件想要访问数据库，只要导入 db.js 就可以了。不过，我们希望一开始就做好这些初始化工作，不用等到需要这些 API 的时候。因此，在 meadowlark.js 中（在这里不使用这个 API）导入这个文件：

```
require('./db')
```

现在已经连接上了数据库，接下来要考虑，如何定义要存取的数据的结构呢？

13.3.6 创建模式和模型

让我们为草地鹨旅游创建一个度假套餐的数据库。先定义一个模式，然后基于它创建一个模型。创建文件 models/vacation.js（版本库的 ch13/00-mongodb/models/vacation.js）：

```

const mongoose = require('mongoose')

const vacationSchema = mongoose.Schema({
  name: String,
  slug: String,
  category: String,
  sku: String,
  description: String,
  location: {
    search: String,
    coordinates: {
      lat: Number,
      lng: Number,
    },
  },
})

```

```
    price: Number,
    tags: [String],
    inSeason: Boolean,
    available: Boolean,
    requiresWaiver: Boolean,
    maximumGuests: Number,
    notes: String,
    packagesSold: Number,
  })

const Vacation = mongoose.model('Vacation', vacationSchema)
module.exports = Vacation
```

以上代码声明了度假产品模型的属性以及它们的类型。可以看到其中有几个字符串属性、一些数值属性、两个布尔属性以及一个字符串数组（用 `[String]` 表示）。我们也可以给模式定义方法。每个商品都有一个库存单位（SKU），虽然我们认为度假产品没有什么“库存”可言，但从记账的角度来看，SKU 是一个相当标准的概念，即便卖的不是有形的商品。

有了模式之后，使用 `mongoose.model` 来创建一个模型。在这一点上，`Vacation` 跟传统的面向对象编程中的类很相似。请注意，如果要定义方法，就必须在创建模型之前定义。



因为浮点数的本质，用 JavaScript 做财务计算时总是要十分小心。我们可以把价格保存为美分而不是美元，这么做当然会有所帮助，但并不能完全解决问题。对我们这个要求不高的旅游网站来说，倒是不用担心，但如果的应用涉及非常大或非常小的金额（例如，由利息或交易量带来的含小数的美分值），就应该考虑使用像 `currency.js` 或 `decimal.js-light` 这样的库。JavaScript 的内建对象 `BigInt` 也可以用于这种用途，从 Node 10 这个版本开始就能使用（在我写作本书时支持的浏览器还很有限）。

我们导出了由 Mongoose 创建的 `Vacation` 模型对象。虽然可以直接使用，但这样会破坏数据库抽象层的努力结果，因此，我们只在 `db.js` 文件中导入它，在应用的其他地方则使用 `db.js` 的方法。在 `db.js` 中加入 `Vacation` 模型：

```
const Vacation = require('./models/vacation')
```

现在已经定义了数据的结构，但我们的数据库还是没什么意思，因为里面还没有数据。我们给它加入一些种子数据，让它更有用。

13.3.7 使用种子数据初始化

鉴于数据库里还没有度假产品数据，我们准备往里面增加几条以便开展工作。到最后，你可能会希望创建一种管理产品的方法，但从本书的目的出发，这些工作就在代码里做了（版本库的 `ch13/00-mongodb/db.js`）：

```

Vacation.find((err, vacations) => {
  if(err) return console.error(err)
  if(vacations.length) return

  new Vacation({
    name: 'Hood River Day Trip',
    slug: 'hood-river-day-trip',
    category: 'Day Trip',
    sku: 'HR199',
    description: 'Spend a day sailing on the Columbia and ' +
      'enjoying craft beers in Hood River!',
    price: 99.95,
    tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
    inSeason: true,
    maximumGuests: 16,
    available: true,
    packagesSold: 0,
  }).save()

  new Vacation({
    name: 'Oregon Coast Getaway',
    slug: 'oregon-coast-getaway',
    category: 'Weekend Getaway',
    sku: 'OC39',
    description: 'Enjoy the ocean air and quaint coastal towns!',
    price: 269.95,
    tags: ['weekend getaway', 'oregon coast', 'beachcombing'],
    inSeason: false,
    maximumGuests: 8,
    available: true,
    packagesSold: 0,
  }).save()

  new Vacation({
    name: 'Rock Climbing in Bend',
    slug: 'rock-climbing-in-bend',
    category: 'Adventure',
    sku: 'B99',
    description: 'Experience the thrill of climbing in the high desert.',
    price: 289.95,
    tags: ['weekend getaway', 'bend', 'high desert', 'rock climbing'],
    inSeason: true,
    requiresWaiver: true,
    maximumGuests: 4,
    available: false,
    packagesSold: 0,
    notes: 'The tour guide is currently recovering from a skiing accident.',
  }).save()
})

```

这里使用的 Mongoose 方法有两个。第一个是 `find`，正如其名，它能查出数据库中 `Vacation` 的所有实例，以得到的实例列表调用回调函数。我们调用 `find` 是因为不想不停地

重复增加种子数据：如果数据库中已经有了度假产品数据，那么说明已经做了初始化了。第一次执行的时候，`find`会返回一个空列表，于是我们创建几个度假对象并调用模型上的`save`方法，就把新对象保存到数据库里了。

现在数据已经在数据库里了，可以把它们取出来。

13.3.8 获取数据

我们在上面已经见到了`find`方法，接下来将用它来显示一个度假列表。不过这次要给`find`传入一个选项以筛选数据。具体来讲，我们希望仅显示当前可用的度假产品。

为产品列表页创建视图 `views/vacations.handlebars`：

```
<h1>Vacations</h1>
{{#each vacations}}
  <div class="vacation">
    <h3>{{name}}</h3>
    <p>{{description}}</p>
    {{#if inSeason}}
      <span class="price">{{price}}</span>
      <a href="/cart/add?sku={{sku}}" class="btn btn-default">Buy Now!</a>
    {{else}}
      <span class="outOfSeason">We're sorry, this vacation is currently
      not in season.
      {{! "这个产品上市时通知我" 页面将是我们的下一个任务 }}
      <a href="/notify-me-when-in-season?sku={{sku}}">Notify me when
      this vacation is in season.</a>
    {{/if}}
  </div>
{{/each}}
```

现在可以创建路由处理函数将所有这些都串起来。在 `lib/handlers.js` 文件中（别忘了导入 `../db`）创建这个处理函数：

```
exports.listVacations = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  const context = {
    vacations: vacations.map(vacation => ({
      sku: vacation.sku,
      name: vacation.name,
      description: vacation.description,
      price: '$' + vacation.price.toFixed(2),
      inSeason: vacation.inSeason,
    }))
  }
  res.render('vacations', context)
}
```

在 `meadowlark.js` 里增加一个路由，调用这个处理函数：

```
app.get('/vacations', handlers.listVacations)
```

如果把这个示例运行起来，就会看到仅有一条数据来自模拟的数据库实现。这是因为虽然我们已经初始化了数据库，加入了种子数据，但是还没有把模拟的实现替换为实际连接数据库的实现。那么现在就替换吧。打开 db.js 文件，修改 `getVacations`：

```
module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    //...
  },
}
```

很简单，只需要一行就能实现，部分是因为 Mongoose 为我们做了大量的工作，也因为我们设计的 API 很符合 Mongoose 的工作方式。稍后适配 PostgreSQL 时，就会发现可能需要多做一点儿工作。



敏锐的读者可能会担心数据库抽象层没有特别“维护”技术中立这个目标。例如，读到这块代码的开发者可能会看出来，他们可以给 `Vacation` 模型传入任何 Mongoose 的选项，然后应用就会使用特定于 Mongoose 的特性，这使得切换数据库变得更难了。可以多采取一些步骤来避免这种情况。在把参数直接传给 Mongoose 之前，可以找出特别的选项，明确地处理它们，并确认每个实现都必须提供这些选项。但为了让示例代码简单一些，我们不准备深究这个。

大多数代码看着很熟悉，但其中应该会有一些让你感到奇怪的地方。比如说，我们是怎样为度假产品列表视图准备上下文的看着就有些奇怪。我们为什么要把从数据库返回的产品映射为几乎一样的对象？原因之一是想让价格显示得更好看一些，所以需要把它转换为一定格式的字符串。

我们本来可以像下面这样少录入一些代码的：

```
const context = {
  vacations: products.map(vacations => {
    vacation.price = '$' + vacation.price.toFixed(2)
    return vacation
  })
}
```

这样做当然可以减少不少代码，但就我的经验而言，不把数据库返回的对象未经映射就直接传给视图也是很合理的。视图可能会得到很多它不需要的属性，而它需要的属性格式也可能跟视图不兼容。目前我们的示例还相当简单，可一旦开始变得复杂之后，很可能你会希望对传入视图的数据做更多的定制。未经映射就直接传给视图，会导致敏感信息或危害网站安全的信息更容易在无意中暴露。出于这些原因，我建议对从数据库返回的数据先做

映射，只把需要的数据传给视图（根据需要转换，就像前面的 `price` 一样）。



MVC 架构的某些变体中引入了称为视图模型的第三个组件。视图模型基本上是对一个模型（或多个模型）进行提取和转换，以便在视图中显示。我们在这里做的工作就相当于即时创建一个视图模型。

我们已经做了很多工作，并成功地使用数据库存储了度假产品信息。但如果不能更新数据，那么数据库就没有什么用。所以，现在把注意力转移到数据库更新上来。

13.3.9 更新数据

我们已经看到了如何新增数据（给度假产品集合增加了种子数据）以及如何更新数据（在预订度假产品时会更新售出的商品数量），现在来看一个更复杂一些的场景，其更能体现文档数据库的灵活性。

如果一款度假产品尚未上市，那么我们就显示一个链接，并邀请用户收取这个产品再次上市的通知。让我们把这个功能串联起来。首先，创建模式和模型 (`models/vacationInSeasonListener.js`)：

```
const mongoose = require('mongoose')

const vacationInSeasonListenerSchema = mongoose.Schema({
  email: String,
  skus: [String],
})
const VacationInSeasonListener = mongoose.model('VacationInSeasonListener',
  vacationInSeasonListenerSchema)

module.exports = VacationInSeasonListener
```

然后，创建视图 `views/notify-me-when-in-season.handlebars`：

```
<div class="formContainer">
  <form class="form-horizontal newsletterForm" role="form"
    action="/notify-me-when-in-season" method="POST">
    <input type="hidden" name="sku" value="{{sku}}">
    <div class="form-group">
      <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
      <div class="col-sm-4">
        <input type="email" class="form-control" required
          id="fieldEmail" name="email">
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-4">
        <button type="submit" class="btn btn-primary">Submit</button>
      </div>
    </div>
  </form>
</div>
```

```
</div>
</form>
</div>
```

接下来，创建路由处理函数：

```
exports.notifyWhenInSeasonForm = (req, res) =>
  res.render('notify-me-when-in-season', { sku: req.query.sku })

exports.notifyWhenInSeasonProcess = (req, res) => {
  const { email, sku } = req.body
  await db.addVacationInSeasonListener(email, sku)
  return res.redirect(303, '/vacations')
}
```

最后，在 db.js 里加上实际的实现：

```
const VacationInSeasonListener = require('./models/vacationInSeasonListener')

module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    await VacationInSeasonListener.updateOne(
      { email },
      { $push: { skus: sku } },
      { upsert: true }
    )
  },
}
```

这是什么魔法呢？如何才能在 `VacationInSeasonListener` 集合中的记录存在之前“更新”它呢？答案就在于使用了一个名为 `upsert`（结合了 `update` 和 `insert`）的 Mongoose 便利方法。基本上，如果给定电子邮件地址的记录尚未存在，就会创建相应的记录。如果记录已存在，就会被更新。然后使用神奇的 `$push` 变量表示我们想要往数组中增加一个值。



如果用户多次填了表单，上面的代码就会不可避免地把多个 SKU 加到记录中。
当一款度假产品要上市时，我们需要找出所有希望得到通知的用户，这时要小心不要重复通知用户。

到这里，各个重要的操作都已经涉及了。我们已经学习了如何连接到 MongoDB 实例、加入种子数据、把数据读取出来以及把更新写进去。然而，也许你更喜欢使用关系型数据库，因此再来看看如何使用 PostgreSQL 做同样的事情。

13.3.10 PostgreSQL

像 MongoDB 这样的对象数据库非常不错，通常来说也可以更快上手，但是如果你要构建一个健壮的应用，那么规划对象数据库结构需要做的工作至少会跟规划一个传统关系型数

数据库一样多，甚至会更多。此外，你可能已经拥有关系型数据库的开发经验，或者已经有了一个关系型数据库，只要连上就可以了。

幸运的是，在 JavaScript 生态中，对每个主要的关系型数据库都提供了强大的支持。如果你希望或必须使用关系型数据库，应该不会有任何问题。

下面我们使用关系型数据库重新实现一下度假产品数据库。这个示例将使用 PostgreSQL，这是一个广受欢迎的成熟的开源关系型数据库。我们要用到的技术和原理跟任何一个关系型数据库都类似。

类似用于 MongoDB 的 ODM，同样存在着用于关系型数据库的对象关系映射（ORM）工具。然而，由于大多数对 ORM 感兴趣的读者很可能已经熟悉了关系型数据库和 SQL，因此我们将直接使用 PostgreSQL 的 Node 客户端库。

像 MongoDB 一样，我们将使用一个免费的在线 PostgreSQL 服务。当然，如果你愿意安装并配置自己的 PostgreSQL 数据库，也可以那么做，只需改变代码中的数据库连接串。如果你使用自己的 PostgreSQL，请确保使用的是 9.4 版或更新版本，因为会用到 JSON 数据类型，而它是在 9.4 版中引入的（写作本书时，我用的是 11.3 版）。

在线 PostgreSQL 服务有很多选择，比如说我将要使用的 ElephantSQL。使用 ElephantSQL 再简单不过了：创建一个账号（可以使用 GitHub 账号登录），然后点击“Create New Instance”。你需要做的只是给它一个名字（比如 meadowlark）和选择一个 plan（你可以使用免费的 plan）。你也要选择一个地区（尽量选择离你最近的那个）。全部设置好之后，你会看到一个列出了实例信息的 Details 区块。复制那个 URL，其中包含用户名、密码和实例位置。

把那个连接串加入 `.credentials.development.json` 文件中：

```
"postgres": {  
    "connectionString": "your_dev_connection_string"  
}
```

关系型数据库与对象数据库之间的一个差异是，在能够存入和取出数据之前，关系型数据库通常要做更多的准备工作：先定义模式，然后使用数据定义 SQL 来创建这些模式。为了跟这个范式一致，我们将在一个单独的步骤中做这些工作，而不是像使用 MongoDB 时一样选择 ODM 或 ORM 来处理。

可以创建数据定义的 SQL 脚本，然后使用命令行客户端来执行这些脚本，从而创建数据库表。或者也可以使用 PostgreSQL 的客户端 API 在 JavaScript 中完成这些工作，但是要把它作为一个单独的步骤，只做一次。既然本书是关于 Node 和 Express 的，我们就采用后一种做法。

首先，需要安装 pg 客户端库 (`npm install pg`)。然后，创建文件 db-init.js，它只是用于初始化数据库，不同于每次服务器启动都要使用的 db.js 文件（版本库的 ch13/01-postgres/db.js）：

```
const credentials = require('./credentials')

const { Client } = require('pg')
const { connectionString } = credentials.postgres
const client = new Client({ connectionString })

const createScript = `
CREATE TABLE IF NOT EXISTS vacations (
    name varchar(200) NOT NULL,
    slug varchar(200) PRIMARY KEY,
    category varchar(50),
    sku varchar(20),
    description text,
    location_search varchar(100) NOT NULL,
    location_lat double precision,
    location_lng double precision,
    price money,
    tags jsonb,
    in_season boolean,
    available boolean,
    requires_waiver boolean,
    maximum_guests integer,
    notes text,
    packages_sold integer
);
`;

const getVacationCount = async client => {
    const { rows } = await client.query('SELECT COUNT(*) FROM VACATIONS')
    return Number(rows[0].count)
}

const seedVacations = async client => {
    const sql = `
    INSERT INTO vacations(
        name,
        slug,
        category,
        sku,
        description,
        location_search,
        price,
        tags,
        in_season,
        available,
        requires_waiver,
        maximum_guests,
        notes,
        packages_sold
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14)
`;
}
```

```

await client.query(sql, [
  'Hood River Day Trip',
  'hood-river-day-trip',
  'Day Trip',
  'HR199',
  'Spend a day sailing on the Columbia and enjoying craft beers in Hood River!',
  'Hood River, Oregon, USA',
  99.95,
  `["day trip", "hood river", "sailing", "windsurfing", "breweries"]`,
  true,
  true,
  false,
  16,
  null,
  0,
])
// 可以使用同样的方式，继续插入其他度假产品数据……
}

client.connect().then(async () => {
  try {
    console.log('creating database schema')
    await client.query(createScript)
    const vacationCount = await getVacationCount(client)
    if(vacationCount === 0) {
      console.log('seeding vacations')
      await seedVacations(client)
    }
  } catch(err) {
    console.log('ERROR: could not initialize database')
    console.log(err.message)
  } finally {
    client.end()
  }
})

```

先从文件的最后说起。我们对数据库客户端（`client`）调用它的 `connect()` 方法，这样会建立一个数据库连接并返回一个 promise。当这个 promise 得到解析的时候，就可以对数据库进行操作了。

对数据库的第一个操作是调用 `client.query(createScript)`，这会创建 `vacations` 表（或者叫作关系）。如果看一下 `createScript`，就会看到它是数据定义 SQL。深入讨论 SQL 超出了本书的范畴，但是在读本节内容时，我假设你对 SQL 已经有了大致了解。你可能会注意到，命名字段的时候使用的是小写下划线的风格，而不是驼峰式。也就是说，原来叫“`inSeason`”的现在成了“`in_season`”。尽管在 PostgreSQL 中使用驼峰式来命名表结构也可以，但那样的话，你必须把包含大写字母的标识符引起来，到最后还会产生更多麻烦。我们过一会儿再回到这个问题。

你会发现，我们不得不对数据的模式投入更多的思考。度假产品的名字最长能有多长？（这里姑且定为最多 200 个字符。）类别名和 SKU 最长能有多长？这里使用了 PostgreSQL

的 `money` 类型来存放价格，并且使用 `slug` 作为主键（而不是再增加一个 ID 字段）。

如果你已经熟悉了关系型数据库，那么对这个简单的模式应该不会感到奇怪。然而，对 `tags`（标签）字段的处理可能会让你觉得新鲜。

在传统的数据库设计中，我们很可能会创建一个新表以关联 `vacations` 和 `tags`（这叫作规范化）。这里当然也可以那样做，但我们决定在传统关系型数据库设计和遵循“JavaScript 之道”之间折中一下。如果使用了两个表（比如 `vacations` 和 `vacation_tags`），为了创建能包含一款度假产品所有信息的单个对象（就像在 MongoDB 的示例中一样），就必须从两个表中查询数据。这额外的复杂性或许会是影响性能的因素，但我们假设它不会，只是希望能够快速地确定某一条度假数据的标签。我们也可以使用一个文本类型字段，以逗号来分隔标签，但是这样的话就得从文本解析出标签来。而 PostgreSQL 给我们提供了更好的方式：JSON 数据类型。一会儿就会看到，通过将字段指定为 JSON 类型（在内部其实是 `jsonb`，这是 JSON 的二进制表示形式，通常性能更高），可以将其存储为一个数组，读取时得到的也是一个数组，就像使用 MongoDB 存储、读取 JavaScript 数组一样。

最后，把种子数据插入数据库中，基本逻辑跟以前一样：如果 `vacations` 表是空的，就增加初始数据；如果表中有数据，就不需要再操作了。

你会注意到，插入数据比用 MongoDB 时略微麻烦一些。要解决这个问题有很多办法，但就这个例子而言，我想更清楚地展示对 SQL 的使用。可以写一个函数，让插入语句更自然一些，也可以使用 ORM（稍后会讨论）。但就目前而言，SQL 已经把事情做了，而且对于熟悉 SQL 的人来说，应该很习惯这样做。

请注意，尽管这个脚本的本意是只运行一次以初始化数据库并增加种子数据，但我们把它写成了可以安全运行多次的形式。我们包含了 `IF NOT EXISTS` 选项，并且在增加种子数据之前先检查了 `vacations` 表是否是空的。

现在可以运行这个脚本来初始化数据库了：

```
$ node db-init.js
```

数据库建立起来了，我们可以写一些代码，在网站上使用这个数据库。

一般数据库服务器在同一时刻只能处理有限数量的连接，因此 Web 服务器通常会采取连接池的策略。建立数据库连接的开销是巨大的，如果让连接保持太长时间，就会有让 Web 服务器阻塞的危险，而连接池可以在两者之间寻求一个平衡。幸运的是，关于连接池的细节，PostgreSQL 的 Node 客户端都为你处理了。

这次对于 `db.js` 文件，将采取略微不同的做法。原来一导入这个文件就建立了数据库连接，现在让它返回我们写的 API，我们通过这个 API 来处理与数据库通信的细节。

我们还要做出一个关于 Vacation 模型的决定。回忆一下，我们创建模型的时候，数据库模式使用的是小写下划线形式，但是我们的 JavaScript 代码用的都是驼峰式。大体来讲，这里有 3 个选择。

- 把数据库模式重构为驼峰式。这样做会让 SQL 更难看，因为必须要记得正确地引用属性名。
- 在 JavaScript 中使用小写下划线形式。这远不是理想做法，因为我们喜欢符合标准的代码。
- 在数据库中使用小写下划线形式，在 JavaScript 中转换为驼峰式。这样的话我们要做的工作会多一些，但 SQL 和 JavaScript 都不用改动。

幸运的是，第三个选择可以自动完成。我们与其自己写函数做转换，不如依靠一个流行的工具库 Lodash，它使得这件事极为容易。只需运行 `npm install lodash` 即可完成安装。

目前我们的数据库需求还比较简单，需要做的只是查出所有可用的度假产品套餐。因此，`db.js` 文件大概是这样的（版本库的 `ch13/01-postgres/db.js`）：

```
const { Pool } = require('pg')
const _ = require('lodash')

const credentials = require('./credentials')

const { connectionString } = credentials.postgres
const pool = new Pool({ connectionString })

module.exports = {
  getVacations: async () => {
    const { rows } = await pool.query('SELECT * FROM VACATIONS')
    return rows.map(row => {
      const vacation = _.mapKeys(row, (v, k) => _.camelCase(k))
      vacation.price = parseFloat(vacation.price.replace(/^\$/ , ''))
      vacation.location = {
        search: vacation.locationSearch,
        coordinates: {
          lat: vacation.locationLat,
          lng: vacation.locationLng,
        },
      }
      return vacation
    })
  }
}
```

短小精悍！我们只导出了一个名为 `getVacations` 的方法，功能是获取度假产品列表。它使用了 Lodash 的 `mapKeys` 和 `camelCase` 函数把数据库属性名转换为驼峰式。

有一个地方需要注意，必须小心处理 `price` 属性，因为 PostgreSQL 的 `money` 类型会被 pg 库转换成一个格式化的字符串。这是有原因的：正如前面讨论过的，JavaScript 最近才加入对任意精度数值类型（`BigInt`）的支持，但能利用这一特性的 PostgreSQL 适配器还不存在

(况且，恐怕它也未必总是最高效的数据类型)。当然可以修改数据库模式，使用一个数值类型而不是 `money` 类型，但是不应该让前端选择左右模式设计。也可以干脆就使用 `pg` 返回的格式化字符串，但那样的话就得修改所有代码，这些代码都是按 `price` 是一个数值类型写的。此外，这种做法会对我们从前端进行的数值计算（比如说计算购物车中各项产品的总价）产生不利影响。由于这些原因，我们倾向于从数据库取出时就把字符串解析为数字。

我们还获取了在表中以“扁平”形式存在的位置信息 (`location`)，把它变成了更符合 JavaScript 风格的结构。做这个变换是为了跟 MongoDB 的示例一致，也可以不做变换按原本那样使用（或者修改 MongoDB 示例让它使用扁平结构）。

关于 PostgreSQL 的使用，还有最后一样东西需要学习，就是数据更新。那么下面就来完成“度假产品上市通知”这个特性。

13.3.11 新增数据

就像 MongoDB 的示例一样，我们也用“度假产品上市通知”来演示数据的新增。先把下面的数据定义加入 `db-init.js` 的 `createScript` 字符串中：

```
CREATE TABLE IF NOT EXISTS vacation_in_season_listeners (
    email varchar(200) NOT NULL,
    sku varchar(20) NOT NULL,
    PRIMARY KEY (email, sku)
);
```

别忘了，我们已经小心地把 `db-init.js` 写成非破坏性的了，因此可以随时运行它。那么就直接再运行一次，让它创建 `vacation_in_season_listeners` 表。

现在可以修改 `db.js` 并增加一个方法来更新这个表：

```
module.exports = {
  // ...
  addVacationInSeasonListener: async (email, sku) => {
    await pool.query(
      'INSERT INTO vacation_in_season_listeners (email, sku) ' +
      'VALUES ($1, $2) ' +
      'ON CONFLICT DO NOTHING',
      [email, sku]
    )
  },
}
```

PostgreSQL 的 `ON CONFLICT` 子句相当于启用了 `upsert`。在这个例子中，如果 `email` 和 `SKU` 的组合已经存在，说明用户已经订阅了通知，那就不需要再做什么了。要是这个表中还有其他的字段（比如上次订阅的日期），也许就需要使用更高级的 `ON CONFLICT` 子句了（更多信息请查看 PostgreSQL 文档的 `INSERT` 部分）。而且要注意，这个行为要依赖于定义表的方式。我们使用 `email` 和 `SKU` 作为组合主键，意味着不能存在重复，这时 `ON CONFLICT` 子句就十分有必

要了（否则，当用户试图再次对同一个度假订阅通知的时候，`INSERT` 命令就会出错）。

现在我们已经看到了使用对象数据库和关系型数据库来实现相同目的的完整例子。应该清楚的是，数据库的功能都是以一致并可扩展的方式来存储、获取和更新数据。因为功能一样，所以可以创建一个抽象层，从而选择不同的数据库技术。最后一个可能会用到数据库的地方是持久化 Session 存储，这个第 9 章已经提到过了。

13.4 使用数据库存储Session

正如第 9 章所讨论的，在生产环境中使用内存作为 Session 数据的存储是不合适的。幸运的是，使用数据库作为 Session 存储是很容易的。

对于 Session 存储，虽然可以使用现有的 MongoDB 或 PostgreSQL 数据库，但是它们功能太强大了，恐怕会过犹不及。这里其实是键值数据库的完美应用场景。当我写到这里时，可以用于 Session 存储的主流键值数据库是 Redis 和 Memcached。为了跟本章的其他示例一致，我们将使用一个 Redis 数据库的免费在线服务。

先去 Redis Labs 创建一个账号。然后创建一个免费的订阅和计划。在计划中选择 Cache，并给数据库起个名字，其余设置按默认的就可以。

接下来会转到数据库详情页面，当我写到这里时，关键信息需要若干秒才能出来，所以需要一点儿耐心。你需要的是 Endpoint 字段和 Access Control & Security 下面的 Redis 密码（默认是隐藏的，但紧跟着它有一个小按钮，点击即可见）。把这些信息放入 `.credentials.development.json` 文件中：

```
"redis": {  
  "url": "redis://:<YOUR PASSWORD>@<YOUR ENDPOINT>"  
}
```

这个 URL 有点儿奇怪，正常来说在冒号前面应该有一个用户名，但是 Redis 允许仅使用密码来连接，不过用来分隔用户名和密码的冒号仍旧不可省略。

我们将使用一个名为 `connect-redis` 的包来提供 Redis 的 Session 存储。安装了它之后 (`npm install connect-redis`)，就可以在主应用文件中把它配置起来了。仍旧使用 `express-session`，但现在给它传入一个新属性 `store`，用来配置使用数据库。要注意，对导入 `connect-redis` 返回的函数，需要给它传入 `expressSession` 才能得到构造函数，这是 Session 存储一种相当常见的用法（版本库的 `ch13/00-mongodb/meadowlark.js` 或 `ch13/01-postgres/meadowlark.js`）：

```
const expressSession = require('express-session')  
const RedisStore = require('connect-redis')(expressSession)  
  
app.use(cookieParser(credentials.cookieSecret))  
app.use(expressSession({  
  resave: false,
```

```
saveUninitialized: false,
secret: credentials.cookieSecret,
secret: credentials.cookieSecret,
store: new RedisStore({
  url: credentials.redis.url,
  logErrors: true, // 强烈推荐!
}),
}))
```

让我们使用新配置好的 Session 存储做点儿有用的事情。设想一下，我们希望能够以不同的币种显示度假产品的价格，并希望网站记住用户的币种偏好设置。

先在度假产品列表的底部加上币种选择器：

```
<hr>
<p>Currency:
  <a href="/set-currency/USD" class="currency {{currencyUSD}}">USD</a> |
  <a href="/set-currency/GBP" class="currency {{currencyGBP}}">GBP</a> |
  <a href="/set-currency/BTC" class="currency {{currencyBTC}}">BTC</a>
</p>
```

还有少许 CSS（可以把它们放入文件 views/layouts/main.handlebars 中，或者链到 public 目录下的一个 CSS 文件中）：

```
a.currency {
  text-decoration: none;
}
.currency.selected {
  font-weight: bold;
  font-size: 150%;
}
```

最后，增加一个路由处理函数来设置币种，并且修改路由 /vacations 的处理函数，以便显示当前币种的价格（版本库的 ch13/00-mongodb/lib/handlers.js 或 ch13/01-postgres/lib/handlers.js）：

```
exports.setCurrency = (req, res) => {
  req.session.currency = req.params.currency
  return res.redirect(303, '/vacations')
}

function convertFromUSD(value, currency) {
  switch(currency) {
    case 'USD': return value * 1
    case 'GBP': return value * 0.79
    case 'BTC': return value * 0.000078
    default: return NaN
  }
}

exports.listVacations = (req, res) => {
  Vacation.find({ available: true }, (err, vacations) => {
    const currency = req.session.currency || 'USD'
    const context = {
      currency: currency,
```

```

vacations: vacations.map(vacation => {
  return {
    sku: vacation.sku,
    name: vacation.name,
    description: vacation.description,
    inSeason: vacation.inSeason,
    price: convertFromUSD(vacation.price, currency),
    qty: vacation.qty,
  }
})
}
switch(currency){
  case 'USD': context.currencyUSD = 'selected'; break
  case 'GBP': context.currencyGBP = 'selected'; break
  case 'BTC': context.currencyBTC = 'selected'; break
}
res.render('vacations', context)
})
}

```

还要在 meadowlark.js 里给设置币种增加一个路由：

```
app.get('/set-currency/:currency', handlers.setCurrency)
```

当然，这里转换币种的做法并不好。我们更希望利用一个第三方的币种转换 API，以保证汇率是最新的。不过这样用作演示已经足够了。现在你可以在不同币种之间切换了。试着停止再重启服务器，就会看到网站已经记住了你的币种偏好设置。如果清除了 Cookie，币种偏好就会被忘记。你会注意到，现在币种金额的格式已不再整齐，而代码也比以前更复杂了，我把这个问题留作读者的练习。

另一个适合留作读者练习的是把 `set-currency` 路由改成通用的，让它更有用。目前它总会重定向到度假产品列表页，可是如果你想在购物车页使用该怎么办？看看能不能想出一两个办法来解决这个问题。

如果看一下数据库，你会发现有一个新的名为 `sessions` 的集合。如果浏览一下这个集合，就会找到一个包含你的 session ID（属性 `sid`）和币种偏好的文档。

13.5 小结

本章的确讨论了很多方面的知识。对大多数的 Web 应用而言，应用之所以有用，核心还是在于数据库。数据库的设计和调优是一个庞大的主题，需要很多本书才能讨论完整。但我希望本章除能给你带来一些基本工具，帮助你连接到两种不同的数据库，并能够对数据进行各种操作。

现在我们已经熟悉了持久化这个基础组件，接下来让我们重新审视一下路由，并理解它在 Web 应用中的重要角色。

第 14 章

路由

路由是你的网站或 Web 服务最重要的方面之一。幸运的是，Express 中的路由十分简单、灵活且稳定。通过路由这种机制，就可以让请求（由 URL 和 HTTP 方法来指定）与处理它们的代码相匹配。正如前面已经提到的，路由过去是基于文件的，而且很简单。例如，如果你把文件 `foo/about.html` 放到你的网站上，就可以在浏览器上通过路径 `/foo/about.html` 访问它。不过，这种方式虽然简单但不够灵活。而且，不管你是否已注意到，在 URL 中带上 `html` 这种方式如今已经非常过时了。

在深入到 Express 路由的技术层面之前，应该先讨论一下信息架构（IA）这个概念。IA 指的是内容的概念组织方式。在开始考虑路由之前，先规划好一套可扩展（但又不过度复杂）的 IA，将来可以带来巨大的好处。

在众多关于 IA 的优秀文章中有一篇文章充满智慧而又永不过时，它就是“Cool URIs don’t change”，作者是 Tim Berners-Lee，实际上他是发明互联网的人。现在你就可以（也应该）读一读。这篇关于互联网技术的文章写于 1998 年，那时的结论到今天还能适用的已经不多了，让我们好好体会一下。

这篇文章向我们提出了应该承担起来的崇高责任：

分配一套 URI，让它们能够坚持到 2 年后、20 年后、200 年后，这是作为网站管理员的职责。这需要思索、需要组织、需要决心。

——Tim Berners-Lee

我总是想象，如果 Web 设计也像其他的工程领域一样要求职业许可，那么我们就会做类

似乎上面的宣誓。（敏锐的读者会发现这篇文章有一个很幽默的地方，就是它的 URL 竟是以 .html 结尾的。）

如果要做一个类比（年轻一代的读者可能不懂这个了，令人遗憾），可以想象一下你最喜欢的图书馆所采用的杜威十进制图书分类系统每两年就大改一次。总有一天你会发现在这个图书馆里什么图书都找不到了。如果你重新设计了 URL 结构，就会发生同样的事情。

请对 URL 多投入一些严肃的思考。20 年后它们仍然会有效吗？（200 年或许有点儿夸张了，谁知道到那时还用不用 URL。不过我还是对考虑得那么长远的人表示敬意。）要仔细组织你的内容，分类要符合逻辑，而且要尽量留有余地。这项工作既是科学，也是艺术。

要跟人合作设计 URL，这可能是最需要注意的一点。即使你是方圆数英里¹最好的信息架构师，也会惊奇于人们看待同样的内容竟有着如此不同的视角。并不是说你应该努力成为符合每个人视角的信息架构师，而是说你应该从多个视角来看待问题，这样才能得出更好的想法，并暴露出你自身的 IA 的缺陷。

为了能得出一套持久的 IA，下面是几条建议。

永远不要在 URL 中暴露技术细节

你有没有过这样的经历：当你访问一个网站，发现它的 URL 是以 .asp 结尾的，然后就认为这个网站已经老旧不堪、无可救药了。要知道，曾几何时 ASP 也是一把利刃。虽然这么说会让我难过，但是 JavaScript、JSON、Node 和 Express 都会不可避免地走向没落。虽然我希望这种事情在很多很多年的辉煌之后再发生，但这个时代对技术往往是无情的。

在 URL 中避免无意义的信息

仔细考虑 URL 中的每一个词，如果没有任何意义，就去掉它。举个例子，只要看到网站把 home 这个词用在 URL 中，我就说不出的难受。因为你的根 URL 就是主页，不需要 /home/directions 和 /home/contact 这样的 URL。

避免不必要的长 URL

如果其他方面都差不多，短 URL 就要比长 URL 好。然而，你也不应该为了让 URL 短一些而牺牲清晰性，或是牺牲 SEO 效果。缩写很“诱人”，但要小心考虑。除非的確是很常见的缩写，才可以用在 URL 中。

使用一致的词间分隔符

使用连字符来分隔词汇很常见，而使用下划线比较少见。通常认为连字符要比下划线更有美感一些，大多数 SEO 专家推荐使用连字符。不过，无论选择连字符还是下划线，务必要保持一致。

注 1：1 英里 =1.609 344 千米。

绝不要使用空格和不可打印的字符

并不推荐在 URL 中包含空格。因为空格通常会被转换为一个加号 (+)，从而引起混淆。显而易见，你也应该避免使用不可打印的字符。我要严肃地告诫你，不要使用任何除字母、数字、连字符和下划线以外的字符。否则即使一时显得很“聪明”，这种“聪明”也是经不住时间考验的。如果你的网站不是面向英语用户的，当然可以使用非英语字符（使用百分号编码），可是如果想给你的网站做本地化，这些字符就是你头疼的根源。

在 URL 中使用小写字母

这条建议会引起一些争议。有些人不仅可以接受混合大小写，而且认为这么做更好。对此，我不想卷入一种教徒式的争论，但我想指出，都用小写的优势是它可以通过代码自动生成。要是你曾经不得不遍历整个网站来剔除数千个链接或做字符串比较，就会赞成这个理由。我个人感觉小写的 URL 更有美感，但最终还是由你来决定。

14.1 路由与SEO

如果你希望你的网站可以被人发现（大多数人会希望），那就需要考虑一下 SEO 以及 URL 是如何影响它的。尤其是，如果有某些关键字很重要，并且本身是有意义的，那就考虑把它们放入 URL 里。例如，草地鹨旅游提供了几个俄勒冈海岸的度假产品。为了让这些产品在搜索引擎中获得高权重，我们在网页 title、标题栏、内容主体、网页元信息描述中都使用了“Oregon Coast”，而且让 URL 都以 /vacations/oregon-coast 开头。俄勒冈的海滨小镇 Manzanita，它的度假产品包位于 /vacations/oregon-coast/manzanita。如果为了缩短 URL 而只使用 /vacations/manzanita，那么在重要的 SEO 上就失分了。

尽管如此，我们还是要抵御为了提升权重就轻率地把关键字塞入 URL 这种诱惑，因为这样是不会如你所愿的。例如，把 Manzanita 度假的 URL 改成 /vacations/oregon-coast-portland-and-hood-river/oregon-coast/manzanita，企图多提及“Oregon Coast”一次，并把“Portland”和“Hood River”也插进去，就是大错特错的做法。完全违背了好 IA 的原则，将事与愿违。

14.2 子域名

跟路径一样，作为 URL 的一部分，子域名也常被用于请求的路由。子域名最适合留给应用中差异显著的部分，比如用于 REST API (api.meadowlarktravel.com) 或后台管理 (admin.meadowlarktravel.com)。有时会出于技术原因而使用子域名。例如，如果我们要使用 WordPress（网站的其余部分则使用 Express）来构建博客，那么使用 blog.meadowlarktravel.com 会更容易（更好的方案是使用代理服务器，比如 NGINX）。使用子域名来划分内容通常会对 SEO 有影响，这就是为什么应该把子域名留给网站中那些对 SEO 来说不那么重要的部分，比如后台管理和 API。这一点要牢记，而且记得在把子域名用于

那些对你的 SEO 方案很重要的内容之前，先确认一下是不是没有别的选择了。

Express 的路由机制默认没有把子域名纳入考虑：`app.get('/about')` 会处理请求 `http://meadowlarktravel.com/about`、`http://www.meadowlarktravel.com/about` 和 `http://admin.meadowlarktravel.com/about`。如果你希望单独处理一个子域名，可以使用一个名为 `vhost`（代表“virtual host”，来自 Apache 的一个常用于处理子域名的机制）的包。首先，安装这个包 (`npm install vhost`)。为了能够在你的开发机器测试基于域名的路由，你需要某种方法来“伪装”域名。幸运的是，你的 hosts 文件就是用来做这个的。在 macOS 和 Linux 机器上，它位于 `/etc/hosts` 中。在 Windows 上，它位于 `c:\windows\system32\drivers\etc\hosts` 中。在 hosts 文件中增加以下内容（你需要管理员权限才能编辑它）：

```
127.0.0.1 admin.meadowlark.local  
127.0.0.1 meadowlark.local
```

这会告诉你的计算机，把 `meadowlark.local` 和 `admin.meadowlark.local` 视为常规的互联网域名，但是映射到 `localhost` (127.0.0.1)。我们使用 `.local` 这个顶级域名是为了让自己不混淆（可以使用 `.com` 或任何其他互联网域名，但它会覆盖实际域名，可能会给你带来麻烦）。

然后你就可以通过 `vhost` 中间件使用可感知域名的路由了（版本库的 `ch14/00-subdomains.js`）：

```
// 创建admin子域名……这应该出现在所有其他路由之前  
var admin = express.Router()  
app.use(vhost('admin.meadowlark.local', admin))  
  
// 创建admin路由，可以定义在任何位置  
admin.get('*', (req, res) => res.send('Welcome, Admin!'))  
  
// 常规路由  
app.get('*', (req, res) => res.send('Welcome, User!'))
```

`express.Router()` 本质上是创建 Express 路由器的一个新实例。你可以认为这个实例跟最初的实例（`app`）一样。然而，要是你不把它加入 `app` 中，它是什么也不会做的。我们通过 `vhost` 把它加入，`vhost` 就把这个路由和域名绑定起来了。



`express.Router` 也可以用来划分路由，十分有用。你可以一次链接多个路由处理函数。更多信息参见 Express 文档的路由部分。

14.3 路由处理函数也是中间件

我们已经看到了基本匹配一个特定路径的路由。但是 `app.get('/foo', ...)` 实际上做了些什么呢？正如在第 10 章中看到的，它只不过是一个特殊化的中间件，使用 `next` 方法传

了进来。再来看一些更复杂的例子（版本库的 ch14/01-fifty-fifty.js）：

```
app.get('/fifty-fifty', (req, res, next) => {
  if(Math.random() < 0.5) return next()
  res.send('sometimes this')
})
app.get('/fifty-fifty', (req, res) => {
  res.send('and sometimes that')
})
```

在之前的例子中，对同一个路由我们使用了两个处理函数。正常来说，第一个会胜出，但在这种情形下，大约有一半的次数会跳过第一个函数，给第二个函数一次机会。我们甚至不需要使用 `app.get` 两次。只要调用一次 `app.get`，使用多少个处理函数都可以。下面这个例子对 3 个不同的响应都给予了近乎同等的机会（版本库的 ch14/02-red-green-blue.js）：

```
app.get('/rgb',
  (req, res, next) => {
    // 大约1/3的请求会返回red
    if (Math.random() < 0.33) return next()
    res.send('red')
  },
  (req, res, next) => {
    // 剩下2/3的请求中有一半（所以也是1/3）会返回green
    if (Math.random() < 0.5) return next()
    res.send('green')
  },
  function (req, res) {
    // 最后的1/3会返回blue
    res.send('blue')
  },
)
```

尽管这起初看起来并没有起到什么特别的作用，但是它允许你创建可以用于任何路由的一般化的函数。比方说我们有一个机制，可以在某些页面上显示若干特别优惠。特别优惠会频繁变更，而且不是每个页面都有。我们可以创建一个函数，把特别优惠注入 `res.locals` 属性中（回忆一下第 7 章）（版本库的 ch14/03-specials.js）：

```
async function specials(req, res, next) {
  res.locals.special = await getSpecialsFromDatabase()
  next()
}

app.get('/page-with-specials', specials, (req, res) =>
  res.render('page-with-specials')
)
```

也可以按这种方式实现一个授权机制。不妨说我们的用户授权的代码设置了一个名为 `req.session.authorized` 的 Session 变量。可以用以下内容来创建一个可重复使用的认证过滤器（版本库的 ch14/04-authorizer.js）：

```
function authorize(req, res, next) {
  if(req.session.authorized) return next()
  res.render('not-authorized')
}

app.get('/public', () => res.render('public'))

app.get('/secret', authorize, () => res.render('secret'))
```

14.4 路由路径和正则表达式

当你在路由中指定一个路径（比如 /foo）时，它最终会被 Express 转换为一个正则表达式。某些正则表达式元字符是可以用在路由路径中的，比如 +、?、*、(和)。下面来看几个例子。比如说你希望 /user 和 /username 这两个 URL 可以用同一个路由来处理：

```
app.get('/user(name)?', (req, res) => res.render('user'))
```

我最喜欢的一个新奇网站的网址为 <http://khaaan.com>（很遗憾现在已经不存在了）。网站上面只是大家都喜欢的星际飞船船长喊着他最具标志性的台词。虽然没什么用，但每次都会让我发笑。假设我们想做出自己的“KHAaaaaaaAN”页，但不希望用户必须记住究竟是 2 个 a、3 个 a 还是 10 个 a。下面的代码就可以做到：

```
app.get('/khaa+n', (req, res) => res.render('khaaan'))
```

然而，在路由路径中，并非所有的正则表达式元字符都有意义——只有前面列出的那几个有意义。这点很重要，因为句点通常是一个可以表示“任何字符”的正则表达式元字符，在路由中使用而无须转义。

最后，如果你的路由的确需要正则表达式的全部威力，那也是支持的：

```
app.get(/crazy|mad(ness)?|lunacy/, (req,res) =>
  res.render('madness')
)
```

我还没有找到一个好理由在路由路径中使用正则表达式元字符，更不用说发挥正则表达式的全部威力了，不过知道有这个功能的存在也不错。

14.5 路由参数

在你的 Express 工具箱中，正则表达式路由可能不太常用，但路由参数极有可能会频繁使用。简单地说，它是把路由的一部分变为变量参数的一个机制。假设我们希望在网站中为每个职员做一个网页。我们有一个包含个人简介和照片的关于职员的数据库。随着公司规模的增长，为每个职员增加一个新的路由就会变得越来越难以操作。下面来看看路由参数是如何解决这个问题的（版本库的 ch14/05-staff.js）：

```

const staff = {
  mitch: { name: "Mitch",
    bio: 'Mitch is the man to have at your back in a bar fight.' },
  madeline: { name: "Madeline", bio: 'Madeline is our Oregon expert.' },
  walt: { name: "Walt", bio: 'Walt is our Oregon Coast expert.' },
}

app.get('/staff/:name', (req, res, next) => {
  const info = staff[req.params.name]
  if(!info) return next() // 最终会“走”到404处理函数
  res.render('05-staffer', info)
})

```

注意看我们是怎样在路由中使用`:name`的。它会匹配任何字符串（不包括斜杠），然后把匹配到的字符串放入`req.params`对象中，键就是`name`。这个特性以后会经常使用，尤其是在创建REST API的时候。你可以在路由中使用多个参数。例如，如果希望将职员按城市划分，可以像下面这样做：

```

const staff = {
  portland: {
    mitch: { name: "Mitch", bio: 'Mitch is the man to have at your back.' },
    madeline: { name: "Madeline", bio: 'Madeline is our Oregon expert.' },
  },
  bend: {
    walt: { name: "Walt", bio: 'Walt is our Oregon Coast expert.' },
  },
}

app.get('/staff/:city/:name', (req, res, next) => {
  const cityStaff = staff[req.params.city]
  if (!cityStaff) return next() // 未识别城市 -> 404
  const info = cityStaff[req.params.name]
  if (!info) return next() // 未识别职员 -> 404
  res.render('staffer', info)
})

```

14.6 组织路由

可能你已经很清楚，在主应用文件中定义所有路由会很不方便。不仅这个文件会随着时间不断变大，而且从功能隔离的角度来说，它也不是一个好方案，因为已经往这个文件里面放太多东西了。一个小网站也许只有十来个路由，或者更少，但一个较大的网站可以有数百个路由。

所以要如何组织你的路由呢？或者，你想要如何组织你的路由呢？Express对于你要如何组织路由没有什么限制，所以要如何做，只受限于你的想象力。

下一节将讨论路由处理的一些主流做法，但在本节的最后，我要推荐4个组织路由需参考的指导原则。

使用命名函数作为路由处理函数

直接在增加路由的地方定义路由处理函数把它内联进去，对很小的应用或原型开发而言是可以的，但随着网站的增长，很快就会变得臃肿和难以维护。

路由不应该神秘化

这条原则是有意说得含混一些，因为比起只有 10 个网页的网站，一个庞大而复杂的网站不可避免地会导致更复杂的组织模式。简单地把网站所有的路由都放入一个文件中，这样就知道它们都在哪里是一种做法。但对于庞大的网站来说，这样做并不可行，所以你要把路由按功能来划分。即使如此，具体的某个路由要到哪里去找你也要清楚。当需要修正某些问题的时候，你绝对不想先花上一个小时来找出这个路由是在哪里处理的。我曾参与过一个 ASP.NET MVC 项目，在这方面简直是一场噩梦。至少有 10 个不同的地方在处理路由，它们也没有什么逻辑性或一致性，常常自相矛盾。即使我十分熟悉那个（非常庞大的）网站，还是要花相当长的时间来捋出某个 URL 是怎么处理的。

路由的组织方式应该是可扩展的

如果你目前有 20 或 30 个路由，那么把它们都定义在一个文件里可能是没问题的。可是 3 年后当你有 200 个路由时会怎么样？这是有可能发生的。不管你采用的是什么方法，都应该保证有增长的空间。

不要忽视基于视图的自动化路由处理函数

如果组成你的网站的很多网页是静态的或有固定的 URL，你所有的路由最终都会像这样：`app.get('/static/thing', (req, res) => res.render('static/thing'))`。为了减少不必要的代码重复，请考虑使用一个基于视图的自动化路由处理函数。这种方式本章后面会讨论，并且可以结合定制路由使用。

14.7 在模块中声明路由

组织路由的第一步是把它们放入自己的模块。实现这一步有多种方法。一种方法是让模块返回包含方法和处理函数属性的一个对象数组。然后就可以在应用文件中这样定义路由了：

```
const routes = require('./routes.js')

routes.forEach(route => app[route.method](route.handler))
```

这种方法有它的优势，很适合动态地存储路由，比如存储到数据库或 JSON 文件中。但是，如果你不需要动态存储路由，那么我推荐把 `app` 实例传入模块里，直接在 `app` 上增加路由。我们的示例应用将采取这种方法。创建一个名为 `route.js` 的文件，把现有的路由都移进去：

```
module.exports = app => {

  app.get('/', (req, res) => app.render('home'))
```

```
//...  
}
```

如果只是剪切和粘贴，很可能会遇到一些问题。例如，如果一些内联的处理函数中使用的变量或方法在新的上下文中不可用，这些引用就损坏了。可以把必要的导入加进来，但先别急，我们一会儿会把这些处理函数移进它们自己的模块，到那时再解决这个问题。

那么如何把这些路由链进来呢？很简单：在 `meadowlark.js` 里，只需导入路由：

```
require('./routes')(app)
```

或者可以更明确一些，增加一个命名的导入（我们把它命名为 `addRoutes`，以便更好地体现它是一个函数；如果愿意，也可以这么命名这个文件）：

```
const addRoutes = require('./routes')  
addRoutes(app)
```

14.8 合乎逻辑地分组路由

为了符合第一条指导原则（使用命名函数作为路由处理函数），需要找个地方来放置这些处理函数。一个相当极端的做法是给每个处理函数都创建一个单独的文件。我想不出来这样做会有什么好处。把相关的功能以某种方式归为一组会更好一些，不仅提取和使用公共函数会更容易，而且修改相关的方法也更容易。

就现在来说，让我们把功能划分到各个文件中：`handlers/main.js` 将放置主页处理函数、“关于”页处理函数以及任何没有另外的逻辑主页的处理函数，而 `handlers/vacations.js` 将放置度假产品相关的处理函数，等等。

`handlers/main.js` 文件大概是这样的：

```
const fortune = require('../lib/fortune')  
  
exports.home = (req, res) => res.render('home')  
  
exports.about = (req, res) => {  
  const fortune = fortune.getFortune()  
  res.render('about', { fortune })  
}  
  
//...
```

现在修改 `routes.js`，让它使用 `handlers/main.js`：

```
const main = require('./handlers/main')

module.exports = function(app) {

  app.get('/', main.home)
  app.get('/about', main.about)
  //...

}
```

这样做满足我们所有的指导原则。`/routes.js` 文件很直白，一眼就可以看出你的网站有哪些路由以及路由都是在哪里处理的。我们也留出了很大的增长空间。可以按相关功能来划分，只要需要，划分成多少个文件都可以。而且，如果 `routes.js` 变得比较笨重了，可以“故技重施”，把 `app` 对象传入另一个模块，在那个模块里注册更多的路由（不过那样的话又有走向“过度复杂”的危险——请先确认复杂化的做法的确是合理的）。

14.9 自动化渲染视图

过去，你都是直接把一个 HTML 文件放入一个目录中，网站就可以展示了。如果你曾经怀念过那些日子，那你并不孤单。如果你的网站是以内容为主而没有太多功能的，你就会觉得为每个视图都加一个路由是不必要的麻烦事。幸运的是，有办法解决这个问题。

比如说我们想增加文件 `views/foo.handlebars`，希望能神奇地通过路由 `/foo` 访问到它。下面来看看要怎么做。在应用文件里，就在 `404` 处理函数之前，添加以下中间件（版本库的 `ch14/06-auto-views.js`）：

```
const autoViews = {}
const fs = require('fs')
const { promisify } = require('util')
const fileExists = promisify(fs.exists)

app.use(async (req, res, next) => {
  const path = req.path.toLowerCase()
  // 检查缓存，如果视图存在，就渲染视图
  if(autoViews[path]) return res.render(autoViews[path])
  // 如果不在缓存里，看看是否有一个匹配的.handlebars文件
  if(await fileExists(__dirname + '/views' + path + '.handlebars')) {
    autoViews[path] = path.replace(/\//, '')
    return res.render(autoViews[path])
  }
  // 未找到视图，往下传递到404处理函数
  next()
})
```

现在只需在 `views` 目录下增加一个 `.handlebars` 文件，就可以在合适的路径上把它神奇地渲染出来。注意常规的路由不会触发这个机制（因为我们把这个自动化视图处理加到所有其他的路由后面了），因此如果对路由 `/foo` 还有一个渲染不同视图的路由，那这个渲染不同

视图的路由会优先。

要注意，如果你删除了一个访问过的视图，这种方法就会出现问题。这个视图将被添加到 `autoViews` 对象中，因此即使它已经被删除，后续的请求也会试图渲染它，这样就出错了。不过，这个问题是可以解决的，只要使用 `try/catch` 块把渲染的代码包装起来，当发现错误时把视图从 `autoViews` 中移除即可。我把这个功能增强留作读者的练习。

14.10 小结

路由是项目的重要组成部分，而路由处理函数可能的组织方式很多，比在这里概括的要多得多，所以你大可放心地试验，并找出一种适合你以及你的项目的技术。我鼓励你优先采用那些既清晰又易于追踪的技术。在外部世界（客户端，通常是浏览器）看来，路由就像是一张地图，导向响应它的服务器端代码。如果地图错综复杂，就很难追踪应用的信息流，对开发和调试都将是障碍。

第 15 章

REST API 和 JSON

虽然第 8 章给出过一些 REST API 的例子，但是目前为止，我们的做事模式几乎是“在服务器端处理这些数据，然后给客户端发送格式化处理的 HTML”。渐渐地，这不再是 Web 应用的默认操作模式。取而代之的是单页应用（SPA），大多数现代 Web 应用属于 SPA。SPA 的操作模式是把所有的 HTML 和 CSS 都打成一个静态资源包，客户端接收整个资源包，随后依靠接收 JSON 格式的零散数据直接操作 HTML。与此同时，提交表单到服务器来通知变更的重要性降低了，被直接使用 HTTP 请求来跟 API 通信替代。

以前我们使用 Express 提供格式化处理的 HTML，现在把注意力转移到使用 Express 提供 API 端点上来。这些内容在第 16 章将派上大用场，到时会演示我们提供的 API 是如何用于动态应用渲染的。

本章会将应用的视图都剥离掉，只留下一个“coming soon”的 HTML 界面，到第 16 章再填充视图。本章将专注于 API，这个 API 将提供对度假产品数据库的访问，同时也支持对产品上市通知的订阅。

Web 服务是一个宽泛的术语，可以指任何通过 HTTP 访问的 API。Web 服务的理念已经出现相当长一段时间了，但最近，实现 Web 服务的技术都太死板、神秘化，而且过度复杂。现在仍有一些系统在使用这些技术（比如 SOAP 和 WSDL），你也可以使用一些 Node 包来跟这些系统打交道。不过本章不会讨论这些。我们将专注于提供所谓的 RESTful 风格的服务，跟它们打交道要简单直白得多。

缩写词 REST 代表 representational state transfer，而不符合语法的 RESTful 用作一个形容词，描述了一种满足 REST 原则的 Web 服务。准确描述 REST 很复杂，且流于计算机科

学语言的形式，但其本质无非就是客户端和服务器端之间的无状态连接。REST 的正式定义也指出，服务既可以被缓存，也可以被分层（也就是说，当你在使用一个 REST API 时，在它之下可能还有另外的 REST API）。

从现实出发，由于 HTTP 本身的限制，创造一个不是 RESTful 的 API 反倒很困难，比方说你还要费很大劲去建立状态。因此，我们的 REST API 的工作非常省事。

15.1 JSON和XML

要提供一套 API，最重要的是双方要有“共同语言”。沟通方式的一部分已经规定好了，就是必须使用 HTTP 方法来跟服务器通信。但在此之后，可以自由选择要使用什么样的数据表示方式。过去，XML 是主流选择，现在，它仍是重要的标记语言。尽管 XML 不算特别复杂，但 Douglas Crockford 还是觉得某种更轻量的东西很有必要，于是 JavaScript Object Notation (JSON) 就诞生了。它除了对 JavaScript 很友好之外（但并不意味着它是 JavaScript 专有的，对任何语言它都是易于解析的格式），还有通常而言更易于手写的优势。

对大多数应用而言，我更偏好 JSON 而不是 XML，因为 JSON 对 JavaScript 有更好的支持，而且它的格式更简单、更紧凑。我建议主要关注 JSON，仅当有系统要求用 XML 跟你的应用通信时才提供 XML。

15.2 我们的API

在开始实行 API 之前需要先规划好它。除了列出度假产品和订阅产品“上市”通知以外，还需要增加一个“删除度假产品”的端点。鉴于这是个公开的 API，我们不会真的删除度假产品，只是把它标记为“请求删除”状态，以便管理员审查。例如，我们使用这个不安全的端点，允许供应商提出把度假产品从网站删除的请求，这个请求随后会被管理员审查。下面是我们的 API 端点。

GET /api/vacations

获取度假产品。

GET /api/vacation/:sku

根据 SKU 返回一个度假产品。

POST /api/vacation/:sku/notify-when-in-season

接收查询串参数 email，对指定的度假产品增加一个上市通知订阅。

DELETE /api/vacation/:sku

提请删除一个度假产品，接收查询串参数为（提请者的）email 和 notes。

[NOTE]

例 15-1

可以使用的 HTTP 动词有很多，其中 GET 和 POST 最常用，其次是 DELETE 和 PUT。创建东西使用 POST，更新（或修改）东西使用 PUT，这已成为标准了。这些词的英语含义不支持这样的任意区分，所以你还要考虑使用路径来区分两个操作，以免混淆。如果想了解关于 HTTP 动词的更多信息，推荐从 Tamas Piros 的这篇文章开始：“RESTful API Design-POST vs PUT vs PATCH”。

可以用多种方式来描述 API。这里选择使用 HTTP 方法和路径的组合来区分 API 调用，组合使用查询串和 body 参数来传输数据。另外，还可以使用不同的路径（比如 /api/vacations/delete），但使用同一个 HTTP 方法¹。也可以使用一致的方式来传输数据。例如，可以选择把所有必要的信息作为路径参数放在 URL 中，而不是使用查询串 DEL /api/vacation/:id/:email/:notes。为了避免 URL 过长，如果要传输较大的数据块（比如数据删除请求的说明信息），推荐使用请求 body。



对于 JSON 格式的 API，有一个主流的重要规范被创造性地命名为 JSON:API。它有点儿啰嗦和重复，不太适合我。不过我认为，不完美的标准总比没有标准好。即使本书中不使用 JSON:API，你也会学到一切必要的知识，帮助你理解和采纳其中的种种约定。更多信息请访问 JSON:API 的主页。

15.3 API 错误报告

在 HTTP API 中，错误报告通常是通过 HTTP 状态码送达的。如果请求返回 200 (OK)，客户端就会知道请求已经成功了。如果请求返回 500 (服务器内部错误)，就是失败了。然而，在大多数应用中，并非所有情况都能（或应该）粗糙地归类为“成功”或“失败”。例如，如果你根据 ID 请求某个东西，而这个 ID 是不存在的，会怎么样呢？这并不代表一个服务器错误，而是客户端请求了不存在的东西。通常，错误可以划分为以下几类。

灾难性错误

错误导致服务器进入了不稳定的状态。通常这是由于某个异常未得到处理。要从一个灾难性的错误中恢复，唯一安全的方式是重启服务器。比较理想的情况是，任何在等待的请求都能收到一个 500 响应码，但如果错误十分严重，服务器也许根本就不能响应，请求最终会超时。

可恢复的服务器错误

可恢复的错误不需要重启服务器或采取其他比较极端的行动。错误是由服务器上某个未预想到的错误条件导致的（比如某个数据库连接失效了）。问题或者是临时的或者是持

注 1：如果你的客户端不能使用不同的 HTTP 方法，可以看看 method-override 这个模块，它允许你“伪装”成不同的 HTTP 方法。

久的。对这种情形，返回一个 500 响应码是合适的。

客户端错误

客户端错误是由客户端差错导致的，这种差错往往是缺少参数或参数非法造成的。这时使用 500 响应码并不合适，毕竟服务器并没有出错。一切都是正常的，只是客户端没有正确使用 API。这时你有两种选择：返回状态码 200，并在响应的 body 中描述错误，或者返回一个合适的 HTTP 状态码，尝试通过状态码来描述错误。我推荐使用后一种方法。在这种情形下，最有用的响应码是 404（未找到）、400（请求错误）和 401（未授权）。而且，响应 body 应当包含对这个错误的详细解释。如果你希望精益求精，那么错误信息甚至要包含文档的链接。要注意，如果用户请求了某个东西的一个列表，而实际没有东西可返回，这就并非是一个错误条件，只要返回一个空列表就是合适的。

我们的应用中将同时使用 HTTP 响应码和 body 中的错误信息。

15.4 跨域资源共享

如果你要发布一个 API，很可能会希望别人能够使用它，这就出现了跨站点的 HTTP 请求。跨站点的 HTTP 请求已经成了很多攻击的入手点，因而也受到了同源策略的约束。同源策略对脚本加载进行了约束，具体来讲，就是要求协议、域名和端口都必须匹配。但如此一来，你就没法把你的 API 给另一个网站使用了。于是，跨域资源共享（CORS）就应运而生了。CORS 允许你根据具体情况解除这个约束，甚至允许具体列出允许访问当前脚本的域名。CORS 是通过 `Access-Control-Allow-Origin` 头实现的。要在 Express 应用中实现 CORS，最容易的方式是使用 `cors` 包 (`npm install cors`)。在应用中启用 CORS，只需使用以下代码：

```
const cors = require('cors')

app.use(cors())
```

因为 API 的同源约束是有原因的（为了避免攻击），所以建议只在必要的时候才使用 CORS。在示例中，我们希望暴露整个 API（但仅限于 API），所以对以 /api 开头的路径应用了 CORS：

```
const cors = require('cors')

app.use('/api', cors())
```

关于 CORS 的更高级的用法，请查阅 `cors` 包的文档。

15.5 测试

如果使用了 GET 以外的 HTTP 动词，测试 API 就会变得很麻烦，因为浏览器只知道发出 GET 请求（以及通过表单发出 POST 请求）。对此倒是有一些办法，比如使用著名的 Postman。然而，不论你是否会使用这样的工具，重要的是要有自动化测试。在为 API 写测试之前，我们需要有某种方法来实际调用 REST API。为此，使用一个名为 node-fetch 的 Node 包，它复制了浏览器上的 Fetch API：

```
npm install --save-dev node-fetch@2.6.0
```

将调用 API（尚未实现）的测试放入 tests/api/api.test.js 中（版本库的 ch15/test/api/api.test.js）：

```
const fetch = require('node-fetch')

const baseUrl = 'http://localhost:3000'

const _fetch = async (method, path, body) => {
  body = typeof body === 'string' ? body : JSON.stringify(body)
  const headers = { 'Content-Type': 'application/json' }
  const res = await fetch(baseUrl + path, { method, body, headers })
  if(res.status < 200 || res.status > 299)
    throw new Error(`API returned status ${res.status}`)
  return res.json()
}

describe('API tests', () => {

  test('GET /api/vacations', async () => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    expect(vacation0.name).toMatch(/\w/)
    expect(typeof vacation0.price).toBe('number')
  })

  test('GET /api/vacation/:sku', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    const vacation = await _fetch('get', '/api/vacation/' + vacation0.sku)
    expect(vacation.name).toBe(vacation0.name)
  })

  test('POST /api/vacation/:sku/notify-when-in-season', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    // 此时，除了确保HTTP请求成功以外，我们没有什么能做的了
    await _fetch('post', `/api/vacation/${vacation0.sku}/notify-when-in-season` ,
      { email: 'test@meadowlarktravel.com' })
  })
})
```

```
    test('DELETE /api/vacation/:sku', async() => {
      const vacations = await _fetch('get', '/api/vacations')
      expect(vacations.length).not.toBe(0)
      const vacation0 = vacations[0]
      // 此时，除了确保HTTP请求成功以外，我们没有什么能做的了
      await _fetch('delete', `/api/vacation/${vacation0.sku}`)
    })
  })
}
```

测试套件的开头是个辅助函数 `_fetch`，首先由它来完成一些公共的工作。它会对 `body` 进行 JSON 编码（如果还没有编码的话），给请求加上合适的请求头，如果响应状态码不是在 200 至 299 范围内，就抛出适当的错误。

我们对 API 的每一个端点都有相应的测试。并不是说这些测试已经是健壮的或完整的，即使对于这样简单的 API，对每个端点也可以（而且应该）进行多个测试。这里只是最基本的代码，更多的只是用来演示 API 测试的技术。

值得一提的是，这些测试有两个比较重要的特征。一个特征是，要求 API 已经启动并且运行在 3000 端口。更健壮一些的测试套件应该找一个未被占用的端口，因为作为测试初始化工作的一个部分，需要在此端口启动 API，并在运行完测试后停止 API。第二个特征是，测试要求数据已经存在于 API 中。例如，第一个测试期望至少有一条度假产品数据，这条产品数据应包含名称和价格属性。在实际应用中，你也许不能做这些假设（比方说，开始的时候并没有数据，你也希望测试缺失数据的情形）。跟端口的问题类似，一个更健壮的测试框架应该有一种方法来设置和重设 API 中的初始化数据，以便每次都可以从一个已知的状态开始。例如，你应该有几个脚本可以用于在每次运行测试时设置测试数据库并加入种子数据、把 API 关联到数据库以及清理数据库。正如第 5 章所述，测试是一个庞大而复杂的主题，这里只能触及皮毛。

第一个测试涉及 GET `/api/vacations` 端点。它取出所有的度假产品信息，验证至少存在一条度假产品数据，并检查第一条看是否包含名称和价格属性。也可以测试其他的数据属性，我把这个留作读者的练习，请思考一下哪些属性对测试来说最重要。

第二个测试涉及 GET `/api/vacation/:sku` 端点。因为没有完全一致的测试数据，所以需要先取出所有的度假产品，然后取出第一条的 SKU，如此才能测试这个端点。

最后两个测试涉及 POST `/api/vacation/:sku/notify-when-in-season` 和 DELETE `/api/vacation/:sku` 端点。遗憾的是，就目前的 API 和测试框架而言，难以验证这些端点是否按预期工作。所以，如果调用它们后没有返回错误，则相信 API 正确地完成了工作。如果希望这些测试更健壮一些，就得采取以下两种措施之一：要么增加若干端点允许我们验证这些操作（比如增加一个端点，用以确认给定的邮箱地址是否已经订阅了某个特定的度假产品），要么给测试增加某种能够访问到数据库的“后门”。

不过，如果现在运行这些测试，它们就会超时然后失败，因为我们还没有实现 API，也没有启动服务器。那么现在就“动手”吧！

15.6 使用Express提供API

Express 足以提供 API 支持。虽然有各种各样的 npm 模块（比如 `connect-rest` 和 `json-api`）提供有效支持，但我发现 Express 本身就非常适合，因此我们将完全使用 Express 来实现。

先在 `lib/handlers.js` 文件里创建这些处理函数（也可以创建一个单独的文件，比如 `lib/api.js`，但目前还是让事情简单一点儿）：

```
exports.getVacationsApi = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  res.json(vacations)
}

exports.getVacationBySkuApi = async (req, res) => {
  const vacation = await db.getVacationBySku(req.params.sku)
  res.json(vacation)
}

exports.addVacationInSeasonListenerApi = async (req, res) => {
  await db.addVacationInSeasonListener(req.params.sku, req.body.email)
  res.json({ message: 'success' })
}

exports.requestDeleteVacationApi = async (req, res) => {
  const { email, notes } = req.body
  res.status(500).json({ message: 'not yet implemented' })
}
```

然后在 `meadowlark.js` 里把这些 API 串联起来：

```
app.get('/api/vacations', handlers.getVacationsApi)
app.get('/api/vacation/:sku', handlers.getVacationBySkuApi)
app.post('/api/vacation/:sku/notify-when-in-season',
  handlers.addVacationInSeasonListenerApi)
app.delete('/api/vacation/:sku', handlers.requestDeleteVacationApi)
```

目前为止，这里没有什么特别奇怪的。注意，我们正在使用数据库抽象层，因此不管使用 MongoDB 实现还是使用 PostgreSQL 实现都是可以的（虽然你会发现有的实现存在一些无关紧要的额外字段，但如果必要的话这些额外字段也是可以去掉的）。

我把 `requestDeleteVacationsApi` 留作了读者的练习，主要是因为这个功能可以有很多种实现方式。最简单的方式应该是修改度假产品的模式，让它包含“已请求删除”的有关字段，当调用这个 API 的时候，使用邮件地址和说明信息更新这些字段。更高级的做法是

使用一个单独的表，类似于一个审核队列，单独地记录这些删除请求，引用涉及的度假产品。不过，这个表会更适合管理员使用。

假设你在第 5 章正确地设置了 Jest，那么现在应该能够运行测试了（`npm test`），而且这些 API 测试将被选中执行（Jest 会查找任何以 `.test.js` 结尾的文件）。你会看到有 3 个测试通过了，1 个没通过，即未完成的 `DELETE /api/vacation/:sku`。

15.7 小结

读完本章，希望你会问：“就这些了吗？”在这个时候，你很可能已经认识到了，Express 的主要作用就是响应 HTTP 请求。至于这些请求是做什么的以及如何来响应完全由你决定。不管你需要响应 HTML、CSS、普通文本，还是 JSON，使用 Express 都能轻而易举实现。你甚至可以响应二进制文件类型。例如，动态构造和返回图片并不会多难。从这个意义上来说，API 不过是 Express 支持的多种响应方式中的一种。

下一章将通过构建一个单页应用把这些 API 都使用起来。而且，会换一种方式重新实现前面章节所完成的页面功能。

第 16 章

单页应用

术语单页应用（SPA）多少有些命名不当，或者至少混淆了“页”的两种含义。从用户的角度看来，SPA 也可以（实际上通常也是）表现为多个页面：主页、度假产品列表页、“关于”页，等等。事实上，的确可以创建出一个让用户无法分辨是传统服务器端渲染的还是 SPA 的应用。

“单页”更多的是体现在 HTML 在什么位置以及如何构造，而不是用户的体验。在一个 SPA 中，当用户第一次加载应用的时候，服务器会发送单个的 HTML 资源包¹，而且，任何 UI 的变化都是 JavaScript 操作 DOM 的结果，而这是为了响应用户活动或网络事件。

虽然 SPA 也需要频繁地跟服务器通信，但 HTML 通常只在第一次请求时作为响应的一部分发送过来。之后，在客户端和服务器之间只会传输 JSON 数据和静态资源文件。

这就是现今统治性的 Web 应用开发方法，要理解其中的原因，首先需要了解一点儿历史。

16.1 Web 应用开发简史

虽然在过去 10 年中，Web 开发方法已经发生了翻天覆地的变化，但有些东西还是保留下来了，那就是网站或 Web 应用所涉及的几个组件。

- HTML 和文档对象模型（DOM）
- JavaScript
- CSS

¹注 1：出于性能考虑，资源包也许会被分割为按需加载（称为懒加载）的多个“组块”，但原理是一样的。

- 静态资源（通常是多媒体，比如图片和视频等）

浏览器把这些组件整合到一起，从而提供了用户体验。

然而，用户体验的结构在 2012 年左右发生了剧烈的变革。到了今天，统治性的 Web 开发范式已经变成了 SPA。

要理解 SPA，先要理解它的对照物。因此在时间上还要往前追溯，回到 1998 年。这一年是“Web 2.0”的说法被提出的前一年，距 jQuery 的出现还有 8 年。

1998 年，Web 应用提供服务的统治性方法是服务器每次都要发送 HTML、CSS、JavaScript 和多媒体文件来响应每一个请求。打个比方，假如你正在看电视，然后想换一个频道。当时 Web 应用提供服务的方法就是让你把电视机扔掉，买一个新的搬回家中并安装好——而这一切只是为了换一个频道（导航到另一个页面，即使页面位于同一个网站内）。

这种方法的问题在于不必要的开销太多了。有时候 HTML（或其中一大部分）根本就没有变化，而 CSS 的变化甚至更少。浏览器可以通过缓存资源文件来降低开销，但是 Web 应用开发变革的节奏太快，使得即使有缓存也难以应付了。

1999 年，“Web 2.0”的说法被创造出来，用以描绘人们开始期待网站能够带来的丰富体验。从 1999 到 2012 年，时间见证了技术的提升，这些提升为 SPA 打下了基础。

聪明的 Web 开发者认识到，如果想要留住用户，那么用户每次换频道时（前面的比喻）都将整个网站发送过去的方式并不可取。开发者们认识到，并非应用的每次变化（UI、数据）都需要来自服务器的信息。即使需要，也不必为了获得一点儿细小的变化，每次都发送整个应用。

1999 到 2012 年间，多页面还是通常意义上的多页面，当你第一次打开网站的时候，会得到这个页面的 HTML、CSS 和静态资源。当你导航到另一个页面，则会得到不同的 HTML、不同的静态资源，有时还有不同的 CSS。然而，为了响应用户的交互，每个页面本身会发生变化，而且不需要请求服务器发送一个完整的新应用，JavaScript 就可以直接改变 DOM。如果需要从服务器获取信息，那就以 XML 或 JSON 格式发送过来，并不需要发送与之相关的 HTML。而解释这些数据并相应地更新用户界面的同样也是 JavaScript。2006 年，jQuery 出现了，它显著地减轻了 DOM 操作的负担，并可以发送网络请求。

这些技术上的革新，很多是由越来越强大的计算机和浏览器（通过扩展）驱动的。Web 开发者们发现，越来越多网站或 Web 应用的美化工作可以直接在用户的计算机上完成，而不需要先在服务器上完成再发送给用户。

在 21 世纪 00 年代的后几年，当智能手机出现以后，Web 开发方法的变化就开始加速了。现在，不仅浏览器有能力完成更多的工作，人们也希望通过无线网络访问 Web 应用。突然间，发送数据的开销上升了，于是，“尽可能少地通过网络传输数据，让浏览器做尽可能

多的工作”的想法就变得更有吸引力了。

到了 2012 年，“尽可能少地通过网络传输数据、让浏览器做尽可能多的工作”的想法已经融入日常的开发之中。正如原始汤孕育出最初的生命一样，这样欣欣向荣的技术环境也为 SPA 这类技术的自然演化提供了条件。

SPA 的理念十分简单：任何一个 Web 应用的 HTML、JavaScript 和 CSS（如果有的话）只会被发送一次。一旦浏览器获得了 HTML，之后对 DOM 所有的修改都由 JavaScript 来完成，让用户感觉就像导航到了另一个页面。比方说，当你从主页导航到度假列表页的时候，服务器就不需要再发送不同的 HTML 了。

当然，服务器还会继续参与：它还要负责提供更新的数据，而且在多用户的应用中，它也是“单一的事实来源”。但在 SPA 的架构中，应用要如何呈现给用户不再是服务器端需要关心的，而是 JavaScript 和 SPA 框架要关注的内容。正是 SPA 框架造就了应用多页面的奇妙错觉。

通常认为，Angular 是第一个 SPA 框架。后来 SPA 阵营又增加了很多成员，其中最著名的是 React、Vue 和 Ember。

如果你刚接触开发，SPA 也许是唯一的参考标准，这段历史对你来说可能只是好玩而已。但如果你是开发老手，则可能会对技术变化带来的困惑和震撼感同身受。不管你属于哪一类，本章的目标都是帮助你理解 Web 应用是如何发布为 SPA 的，以及理解 Express 在 SPA 中承担什么样的角色。

这段历史跟 Express 也有关系，因为在 Web 开发技术的变化中，Express 的角色也有所改变。本书第 1 版出版的时候，Express 一般用于提供多页应用（加上支持 Web 2.0 功能的 API），但现在几乎已经完全用于 SPA、开发服务器和 API 了，这也体现了 Web 开发是不断变化发展的。

有意思的是，仍旧有一些合理的理由，需要 Web 应用能够提供“特定”的网页（而不是那个“通用”的网页，因为它会被浏览器重新排版）。看起来我们是兜了个圈，或者抛弃了 SPA 的好处，但是正是 SPA 的架构，让我们可以更好地实现“特定”的网页。这个“特定”网页的技术叫**服务器端渲染**（SSR），它允许服务器使用跟浏览器一模一样的代码，创建一些专门的网页，来提升第一页的加载速度。这里的关键是服务器端无须过多思索，只需使用跟浏览器端一样的技术就可以生成一个特定网页。这种 SSR 通常用来增强第一页的加载体验，同时支持搜索引擎优化。这是个更高级的主题，这里不再讨论了，你只需知道有这种实践就好了。

现在我们已经明白了 SPA 是如何出现的以及为什么出现，接下来看一看目前可以使用的 SPA 框架都有哪些。

16.2 SPA技术选择

现在有很多种 SPA 技术可以选择。

React

目前，React 似乎是 SPA 的“掌权者”，尽管一边是赫赫有名的前霸主（Angular），另一边是虎视眈眈的后来者（Vue）。大约在 2018 年，React 在使用量统计上超过了 Angular。React 虽然是一个开源的库，但最初是作为 Facebook 的一个项目起步的，现在 Facebook 仍是它的活跃贡献者。对草地鹨旅游的重构将使用 React。

Angular

大多数人认为，谷歌的 Angular 是“最初”的 SPA，它虽然曾经风靡一时，但后来还是被 React 拉下了宝座。2014 年年底，Angular 发布了版本 2，其对第 1 版进行了大幅度的重写，此举疏离了很多已有的用户，也让新用户望而却步。我相信这次版本迁移是 React 最终能超过 Angular 的其中一个原因。另一个原因是，Angular 是一个比 React 庞大得多的框架。框架庞大既有优势也有劣势：对于构建大而全的应用，Angular 提供了更完整的架构，而且不管做什么，总会有清楚的“Angular 方式”，但也正因如此，它不像 React 和 Vue 等其他框架中很多东西可以取决于用户的个人选择和创造力。不管哪种 SPA 框架更好，更大的框架演化起来就会更笨重、更缓慢一些，相比之下 React 就有了创新上的优势。

Vue.js

Vue 是 React 的“后进者”，最初由一名叫尤雨溪（Evan You）的开发者创建。在极短的时间内，Vue 就收获了无数的追随者，他们将其视若珍宝。不过 Vue 在流行度上仍与 React 相去甚远。我曾使用过 Vue，也欣赏它清晰的文档和轻量化的方法，但已经习惯 React 的架构和哲学了。

Ember

如同 Angular，Ember 提供了一个全面的应用框架。它拥有庞大而活跃的开发社区，尽管在创新性上不如 React 和 Vue，但它支持大量的功能，清晰性也很好。我发现自己特别偏爱更轻量的框架，坚持使用 React 也是这个原因。

Polymer

我没有用过 Polymer，但它背后有谷歌的支持，想来比较可靠。人们似乎都好奇 Polymer 带来的东西，但我还没有见到有很多人真正采用它。

如果你在寻找一个健壮的开箱即用的框架，并且不介意在它的规则之内行事，那就应该考虑 Angular 或 Ember。如果你希望有表达和创新的空间，我推荐 React 或 Vue。我还不清楚 Polymer 适用于什么地方，但仍会对它保持关注。

既然这些“选手”都已经亮相，那么让我们与 React 一起继续前行，把草地鹨旅游重构成为一个 SPA。

16.3 创建React应用

开始一个 React 应用最好的方式是使用 `create-react-app` (CRA) 这个工具，它会创建所有的项目样板代码、开发者工具，并提供一个最小的应用让你作为起点。而且，`create-react-app` 会让它的配置保持更新，因此我们可以专注于应用的构建而无须再为框架工具链而分心。尽管这么说，如果真的需要配置工具链，也可以把应用从工具链中分离出来，这样虽然不能再跟最新的 CRA 工具链保持一致，但也获得了对应用配置完全的控制。

一直以来，我们应用开发的一切工作成果都是跟 Express 应用合为一体的，而 SPA 最好是被看作一个分离的独立应用。如此一来，就有了两个应用根目录。清晰起见，当我提及 Express 应用所在的目录时，会说服务器根目录；当提及 React 应用所在的目录时，会说客户端根目录。而应用根目录是这两个目录所在的目录。

既然如此，那就进入你的应用根目录，创建一个名为 `server` 的目录，这就是 Express 应用服务器所在的地方。不用创建客户端应用的目录，CRA 会为我们创建。

运行 CRA 之前需要先安装 Yarn。Yarn 是一个包管理器，跟 npm 类似。实际上，Yarn 基本上完全替代了 npm。虽然 React 开发并未强制使用 Yarn，但它已经成为事实标准了，不使用它无异于要“逆流而上”。虽然 Yarn 和 npm 在用法上有一些微小差异，但是很可能你只会注意到，你要运行的是 `yarn add` 而不是 `npm install`。要安装 Yarn，只需遵照 Yarn 官方文档的安装指引即可。

安装好 Yarn 之后，在应用根目录下运行以下代码：

```
yarn create react-app client
```

现在进入 `client` 目录，输入 `yarn start`。数秒之后，会弹出一个新的浏览器窗口，里面运行着 React 应用。

就让这个终端窗口保持运行。CRA 对于“热加载”有着相当好的支持，因此只要对源代码做了修改，源代码就会快速得到构建，浏览器也会自动重新加载。一旦习惯，你就再也离不开 CRA 了。

16.4 React基本概念

React 的官方文档非常出色，这里就不照搬文档内容了。因此，如果你刚接触 React，请先从“入门教程”读起，然后是文档的“核心概念”部分。

你会发现，React 是围绕着组件组织的，这是 React 主要的构建块。用户在 React 中看到的或与之交互的通常都是组件。我们看一下文件 `client/src/App.js`（你的文件内容或许会有一点儿差异，因为 CRA 也会随着时间而变更）：

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://exl.ptpress.cn:8442/ex/l/f74974f4"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

React 的一个核心概念是 UI 由函数生成。如上所见，最简单的 React 组件就是一个返回 HTML 的函数。你也许会想，这不是合法的 JavaScript，看起来像是混入了 HTML。实际情况要复杂一些。React 默认启用了 JavaScript 的一个名为 JSX 的超集。JSX 允许你写看起来像 HTML 的东西，但实际上它不是 HTML，而是在创建 React 元素，目的是要（最终）对应到一个 DOM 元素。

不过归根结底，你可以把这些看起来像 HTML 的东西看作 HTML。在上面的代码中，`App` 是一个函数，它将渲染一个 HTML，这个 HTML 对应函数所返回的 JSX。

有几件事情需要注意，尽管 JSX 很接近（但并不是）HTML，但还是存在一些微妙差异。你可能已经注意到了，我们使用的是 `className`，而不是 `class`，因为 `class` 是 JavaScript 的保留关键字。

要指定 HTML，只需在任何一个期望表达式的地方创建 HTML 标签。在 HTML 内，还可以使用花括号“返回”到 JavaScript。例如：

```
const value = Math.floor(Math.random()*6) + 1
const html = <div>You rolled a {value}!</div>
```

在这个例子中，`<div>` 开启了 HTML，包住 `value` 的花括号跳转到了 JavaScript，提供存放于 `value` 的数值。要把计算内联进去也很容易：

```
const html = <div>You rolled a {Math.floor(Math.random()*6) + 1}!</div>
```

JSX 中的花括号可以包含任何合法的 JavaScript 表达式，甚至可以包含其他 HTML 元素。渲染列表就是一个常见的用例：

```
const colors = ['red', 'green', 'blue']
const html = (
  <ul>
    {colors.map(color =>
      <li key={color}>{color}</li>
    )}
  </ul>
)
```

这个例子有几点需要注意。首先，注意我们把 `colors` 映射成了返回 `` 元素。这点很关键，因为 JSX 完全依靠表达式运算。因此，`` 必须要包含一个表达式或表达式的一个数组。如果你把 `map` 改成了 `forEach`，就会发现那些 `` 元素没有被渲染。其次，注意那些 `` 元素接收了一个属性 `key`：这是性能上的一个让步。为了让 React 知道何时需要重新渲染元素，对每个元素它都需要一个唯一键。既然我们的数组元素是唯一的，就可以直接使用元素的值，但通常你会使用一个 ID，或者（如果没有其他属性可用的话）使用元素在数组中的索引。

继续阅读之前，鼓励你体验一下 `client/src/App.js` 中的一些 JSX 例子。如果一直运行 `yarn start`，每次保存修改时，修改的内容都会自动地在浏览器中反映出来，如此无疑会为你的学习周期加速。

在结束本节内容之前，还有一个话题需要提及，那就是状态（state）。每个组件都可以有自己的状态。状态可以解释为“组件关联的可以修改的数据”。购物车就是一个很好的例子。购物车组件可以包含商品项目的一个列表，当你增加或移除购物车中的商品时，这个组件的状态就改变了。这看起来好像是过于简单或显而易见的概念，但是它造就了一个 React 应用的绝大部分细节，归根结底，都是为了高效地设计和管理组件的状态。到制作度假产品列表页的时候，我们会看到一个有关状态的例子。

下面继续创建草地鹨旅游的主页。

16.4.1 主页

回想一下我们的 Handlebars 视图，我们有一个主布局文件，负责为网站建立主要的外观。下面先来检查一下，它的 `<body>` 里面有什么（除了脚本）：

```
<div class="container">
  <header>
    <h1>Meadowlark Travel</h1>
    <a href="/"></a>
  </header>
  {{body}}
</div>
```

将这些重构为 React 组件很容易。首先，把 logo 复制到 client/src 目录。为什么不是 public 目录呢？对于很小或很常用的图片文件来说，把它们内联在 JavaScript 资源包里可能会更高效一些，而且，CRA 可以为你打的这个包做出智能判断。你从 CRA 得到的示例应用把它的 logo 直接放到 client/src 目录里了，但我还是喜欢把图片资源统一放到一个子目录里，所以把 logo (logo.png) 移到了 client/src/img/logo.png 中。

只剩下一个小问题了，但比较棘手，那就是如何处理 {{{body}}}。在视图中，这是要渲染另一个视图的地方，即渲染你所在的特定页面的内容。在 React 中，我们的基本理念也是一样的：既然所有内容都是以组件形式来渲染的，那只在这里渲染另一个组件就可以了。先从一个空的 Home 组件开始，一会儿再扩充：

```
import React from 'react'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (<i>coming soon</i>)
}

function App() {
  return (
    <div className="container">
      <header>
        <h1>Meadowlark Travel</h1>
        <img src={logo} alt="Meadowlark Travel Logo" />
      </header>
      <Home />
    </div>
  )
}
export default App
```

我们对 CSS 采用的做法跟示例应用一样：简单地创建一个 CSS 文件并导入它。因此，可以直接编辑那个 CSS 文件，应用任何想要的样式。在这个例子中，我们尽量不把事情搞得太复杂。而使用 CSS 对 HTML 进行样式化的方式并没有什么根本性的改变，因此那些熟悉的各种工具依旧可以使用。



CRA 会为你建立好 Lint。在跟随本章示例的过程中，很可能看到一些警告（既会出现在 CRA 终端的输出中，也会出现在浏览器的 JavaScript 控制台中）。这只是因为我们增量地加入了东西，到本章结束，应该就不会再有警告了。如果还有，那么确认一下是不是漏了某个步骤。也可以跟配套的版本库对照一下。

16.4.2 路由

第 14 章中学习的路由的核心概念现在并没有改变。我们仍旧使用 URL 路径来决定用户看到的是哪部分的界面。有所不同的是，在这里路由由客户端应用来处理。客户端应用现在负责基于路由来改变 UI。如果页面导航要求来自服务器的新数据或数据更新，那么就由客户端应用来向服务器发出请求。

React 应用的路由有非常多的选择，每个人的偏好也不同。不过，有一个路由库是统治性的：React Router。对于 React Router，我不喜欢的地方不少，但它太常用了，你一定会碰到它的。而且，要想快速运行一个简单的东西，它是个不错的选择。基于这两个原因，这里选择使用 React Router。

先安装 DOM 版本的 React Router（还有一个适合移动开发的 React Native 版本）：

```
yarn add react-router-dom
```

现在把路由连接起来，并增加“关于”页和“页面未找到”页，然后给网站 logo 加上链接，让它链接到主页：

```
import React from 'react'
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (
    <div>
      <h2>Welcome to Meadowlark Travel</h2>
      <p>Check out our "<Link to="/about">About</Link>" page!</p>
    </div>
  )
}

function About() {
  return (<i>coming soon</i>)
}

function NotFound() {
  return (<i>Not Found</i>)
}

function App() {
  return (
    <Router>
      <div className="container">
```

```

<header>
  <h1>Meadowlark Travel</h1>
  <Link to="/"><img src={logo} alt="Meadowlark Travel Logo" /></Link>
</header>
<Switch>
  <Route path="/" exact component={Home} />
  <Route path="/about" exact component={About} />
  <Route component={NotFound} />
</Switch>
</div>
</Router>
)
}

export default App

```

需要注意的第一件事是，我们把整个应用都包装进了一个 `<Router>` 组件。如此就如你所愿启用了路由。在 `<Router>` 内部，可以使用 `<Route>` 基于 URL 路径有条件地渲染一个组件。把路由内容放进一个 `<Switch>` 组件里，就保证了其中仅有一个组件能够得到渲染。

我们已经使用过的 Express 路由与 React Router 之间存在一些微妙的差异。我们会在 Express 中渲染第一个成功匹配的页面（如果没有匹配，就是 404 页）。而使用 React Router，路径只是一个“提示”，用以确定要显示哪些组件的组合。这样看来，它要比 Express 中的路由更灵活。因此，React Router 路由的默认行为就好像有一个星号 (*) 被放在了路径的末尾。也就是说，路由 / 默认会匹配所有页面（因为所有页面都以一个斜杠开头）。由于这个原因，我们要使用 `exact` 属性，让这个路由更像一个 Express 路由。同理，要是没有 `exact` 属性，/about 路由会把 /about/contact 也匹配了，这很可能不是你所希望的。对于主要内容的路由，可能你会希望所有的路由（除了“Not Found”路由）都包含 `exact` 属性。否则，你就得确保在 `<Switch>` 里正确地排列了它们，以便按正确的顺序逐一匹配。

需要注意的第二件事是 `<Link>` 的使用。你也许会奇怪为什么不直接使用 `<a>` 标签。`<a>` 标签的问题在于，浏览器会坚持把它们看作“转到别的页面”，即使要转到的页面位于同一个网站，结果就是浏览器向服务器发出一个新的 HTTP 请求，HTML 和 CSS 将再次被下载，但这样就破坏 SPA 的工作机制了。`<a>` 标签所做的工作也就仅限于此了。仅从页面加载而言，虽然使用 `<a>` 标签可以正常工作，React Router 也没有做错，但发出了不必要的网络请求，难言快速或高效。理解其中的差异是很有必要的，将有助于理解 SPA 的本质。可以试验一下，创建两个导航元素，一个使用 `<Link>`，一个使用 `<a>`：

```

<Link to="/">Home (SPA)</Link>
<a href="/">Home (reload)</Link>

```

然后打开开发者工具（在 Chrome 上），再打开“Network”标签页，勾选“Preserve log”。现在点击“Home (SPA)”链接可以看到并没有网络流量产生，而点击“Home (reload)”可以观察到网络流量。简而言之，这就是 SPA 的本质。

16.4.3 度假产品页——可视化设计

目前为止，我们还只是在构建一个纯前端的应用。那么，Express 处于哪个位置呢？我们的服务器仍然是单一的数据源。尤其是它维护着度假产品的数据库，而这些度假产品正是要在网站上显示的。幸运的是，我们在第 15 章已经做了绝大部分的工作：暴露了一个可以以 JSON 格式返回度假产品的 API，React 应用可以直接使用了。

不过，在把前后端连接起来之前，让我们先把度假产品页做出来。虽然目前还没有度假产品可以渲染，但是不要让度假产品页成为后续开发的障碍。前一节把所有的内容页面都包含进了 client/src/App.js，但通常认为这不是一个好实践：让每个组件都放入它自己的文件中是更符合习惯的做法。所以让我们花一点儿时间，把 Vacations 组件放入它自己的文件。创建文件 client/src/Vacations.js：

```
import React, { useState, useEffect } from 'react'
import { Link } from 'react-router-dom'

function Vacations() {
  const [vacations, setVacations] = useState([])
  return (
    <>
      <h2>Vacations</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <div key={vacation.sku}>
            <h3>{vacation.name}</h3>
            <p>{vacation.description}</p>
            <span className="price">{vacation.price}</span>
          </div>
        )}
      </div>
    </>
  )
}

export default Vacations
```

目前我们做的还十分简单：只是返回一个 `<div>`，里面还包含了若干 `<div>` 元素，每个都代表一个度假产品。那么 `vacations` 变量是从哪来的呢？在这个例子中，我们使用了 React 的一个较新的特性，叫 React hooks（React 钩子），在钩子方法出现之前，如果一个组件希望拥有自己的状态（在这种情形中，就是度假产品列表），就必须得使用一个类实现。而钩子的出现让我们可以使用拥有独立状态的函数组件。在 `Vacations` 函数中，我们调用了 `useState` 来建立状态。注意，我们给 `useState` 传入了一个空数组，这将是状态中 `vacations` 的初始值（一会儿再讨论怎么填充这个数组）。`useState` 返回的是一个数组，包含这个状态值本身（`vacations`）和更新这个状态的方式（`setVacations`）。

你也许会好奇为什么不能直接修改 `vacations` 的状态：它不过是一个数组，难道不能调用 `push` 把度假产品加进去吗？能是能，但这样就违背了 React 状态管理系统的本意：保证一

致性、保证性能以及保证组件间的通信。

你大概也会好奇环绕着度假产品的看起来像空组件的东西 (`<>...</>`)。它被称为 **fragment**。这个 fragment 是必要的，因为每个组件都必须渲染成单个元素。在这种情形中有两个元素，即 `<h2>` 和 `<div>`。这个 fragment 只是提供一个能容纳这两个元素的“透明”的根元素，这样我们才能将其渲染成单个元素。

把 `Vacations` 组件加入应用，即便它还没有度假产品可以显示。在 `client/src/App.js` 中，首先导入度假产品页：

```
import Vacations from './Vacations'
```

然后在 `<Switch>` 路由组件中为它创建一个路由：

```
<Switch>
  <Route path="/" exact component={Home} />
  <Route path="/about" exact component={About} />
  <Route path="/vacations" exact component={Vacations} />
  <Route component={NotFound} />
</Switch>
```

保存文件，应用应该会自动重新加载，然后就可以导航到 `/vacations` 页了，尽管现在还没有什么东西可看。现在，我们的客户端已经有了大部分的基础设施，接下来看看如何跟 Express 集成起来。

16.4.4 度假产品页——跟服务器端集成

我们已经完成了建立度假产品列表页的大部分工作，也有了一个从数据库获取度假产品并返回 JSON 格式的 API 端点，现在需要确定如何实现服务器跟客户端互相通信。

可以从第 15 章完成的工作开始。不需要增加什么，但可以去掉一些不再需要的东西。以下内容可以去掉。

- Handlebars 和视图支持（不过，我们将保留 static 中间件，原因一会儿就会看到）。
- Cookie 和 Session（SPA 仍会使用 Cookie，但不再需要服务器端的支持……我们将以完全不同的方式来思考 Session）。
- 所有渲染视图的路由（显然，我们会保留 API 路由）。

这样就得到了一个简化得多的服务器。那么现在需要做些什么工作呢？首先要解决以下问题：服务器在使用 3000 端口，而 CRA 开发服务器默认也是使用 3000 端口。我们可以修改两个端口中的任意一个，那就修改 Express 端口吧。我通常使用 3033 端口——只是觉得这个数字好听。回想一下在 `meadowlark.js` 里是怎么设置默认口号的，只需这么修改：

```
const port = process.env.PORT || 3033
```

当然可以使用环境变量来控制端口号，但既然要频繁地把它跟 SPA 开发服务器一起使用，还不如修改代码。

现在只要两个服务器都在运行，就可以实现在两者之间通信了。但如何实现呢？只需在 React 中做类似这样的事情：

```
fetch('http://localhost:3033/api/vacations')
```

不过这个做法的问题在于，如果整个应用都像这样发出请求，那么就把 http://localhost:3033 嵌入到了所有的地方……这在生产环境是无效的；而且在你的同事的计算机上也未必能工作，因为可能需要使用不同的端口；还有，在测试服务器上，也许要求使用不同的端口……诸如此类。上面的做法在配置上就是个头疼事。没错，你的确可以把 URL 基地址存储为一个变量，并在各处使用这个变量，但还有更好的做法。

在理想世界中，从应用的角度来看，任何东西都是从一个地方来的：从同样的协议、主机和端口来获取 HTML、静态资源和 API。这样就简化了很多东西，并保证了源代码的一致性。如果任何东西都是从一个地方来的，那么你可以简单地省略它的协议、主机和端口，只调用 `fetch('/api/vacations')`。这是个不错的做法，而且幸运的是，这么做也很容易。

CRA 的配置包含对 proxy 的支持，会帮助你把 Web 请求传输给你的 API。编辑 `client/package.json` 文件，并添加以下内容：

```
"proxy": "http://localhost:3033",
```

将以上代码放在哪里都没关系。我通常把它放在 `"private"` 和 `"dependencies"` 之间，只是因为喜欢在文件的顶头看见它。现在，只要你的 Express 服务器运行在 3033 端口，你的 CRA 开发服务器就会把 API 请求传送到 Express 服务器。

既然已经加上了配置，那么让我们使用 `effect`（另一个 React 钩子）来获取和更新度假数据。下面是使用了 `useEffect` 钩子的整个 `Vacations` 组件：

```
function Vacations() {
  // 初始化状态
  const [vacations, setVacations] = useState([])

  // 获取初始数据
  useEffect(() => {
    fetch('/api/vacations')
      .then(res => res.json())
      .then(setVacations)
  }, [])

  return (
    <>
    <h2>Vacations</h2>
    <div className="vacations">
```

```

    {vacations.map(vacation =>
      <div key={vacation.sku}>
        <h3>{vacation.name}</h3>
        <p>{vacation.description}</p>
        <span className="price">{vacation.price}</span>
      </div>
    )}
  </div>
</>
)
}

```

跟前面一样，`useState` 配置了组件的状态数据，让它包含一个 `vacations` 数组，并提供属性设置方法。现在我们增加了 `useEffect`，它会调用 API 获取度假数据，然后异步调用这个属性设置方法。注意我们给 `useEffect` 传入了一个空数组作为第二个参数，这是给 React 的一个信号，告诉它这个 effect 钩子只应该在组件刚被挂载的时候运行一次。表面上，以这种方式给出信号似乎有些怪异，一旦对钩子有了更多了解，你就会发现实际上它是始终如一的。更多信息请查阅 React 文档中的钩子部分。

钩子是相对较新的概念，是在 2019 年 2 月份的版本 16.8 中才加入的，因此，即使你已经有了一些 React 的开发经验，也可能对它们并不熟悉。我坚持认为，钩子是在 React 架构上一项卓越的创新，而且，尽管开始看起来有些怪异，最终你还是会发现它们实际上简化了你的组件，并且可以减少那些人们常犯的更棘手的、与状态相关的错误。

既然已经了解了如何从服务器获取数据，那么再来关注一下另一个方向的信息传输。

16.4.5 向服务器发送信息

我们已经有了一个做服务器变更的 API 端点，可以设置当某个度假产品重新上市时接收邮件通知。修改 `Vacations` 组件，为已下线的度假产品显示一个注册表单。为遵循 React 的模式，我们将创建两个新的组件，即把单独的度假产品视图拆分成 `Vacation` 和 `NotifyWhenInSeason` 两个组件。虽然可以在一个里完成，但是对 React 开发来说，推荐的做法是使用多个特定用途的组件，而不是巨大的多用途组件（不过为了简洁起见，就不把这些组件放进它们自己的文件里了，我把这个留作读者的练习）：

```

import React, { useState, useEffect } from 'react'

function NotifyWhenInSeason({ sku }) {
  return (
    <>
      <i>Notify me when this vacation is in season:</i>
      <input type="email" placeholder="(your email)" />
      <button>OK</button>
    </>
  )
}

```

```

function Vacation({ vacation }) {
  return (
    <div key={vacation.sku}>
      <h3>{vacation.name}</h3>
      <p>{vacation.description}</p>
      <span className="price">{vacation.price}</span>
      {!vacation.inSeason &&
        <div>
          <p><i>This vacation is not currently in season.</i></p>
          <NotifyWhenInSeason sku={vacation.sku} />
        </div>
      }
    </div>
  )
}

function Vacations() {
  const [vacations, setVacations] = useState([])
  useEffect(() => {
    fetch('/api/vacations')
      .then(res => res.json())
      .then(setVacations)
  }, [])
  return (
    <>
      <h2>Vacations</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <Vacation key={vacation.sku} vacation={vacation} />
        )}
      </div>
    </>
  )
}

export default Vacations

```

现在，如果有哪些度假产品的 `inSeason` 属性是 `false` 的（应该会有，除非你改过数据库或初始化脚本），表单就显示出来了。把按钮激活，让它调用 API。修改 `NotifyWhenInSeason`：

```

function NotifyWhenInSeason({ sku }) {
  const [registeredEmail, setRegisteredEmail] = useState(null)
  const [email, setEmail] = useState('')
  function onSubmit(event) {
    fetch(`'/api/vacation/${sku}/notify-when-in-season`, {
      method: 'POST',
      body: JSON.stringify({ email }),
      headers: { 'Content-Type': 'application/json' },
    })
    .then(res => {
      if(res.status < 200 || res.status > 299)
        return alert('We had a problem processing this...please try again.')
      setEmail(res.data)
      setRegisteredEmail(res.data)
    })
  }
}

```

```
        event.preventDefault()
    }
    if(registeredEmail) return (
      <i>You will be notified at {registeredEmail} when
      this vacation is back in season!</i>
    )
    return (
      <form onSubmit={onSubmit}>
        <i>Notify me when this vacation is in season:</i>
        <input
          type="email"
          placeholder="(your email)"
          value={email}
          onChange={({ target: { value } }) => setEmail(value)}
        />
        <button type="submit">OK</button>
      </form>
    )
  }
}
```

这里选择让组件跟踪两个不同的值：用户正在键入的邮箱地址以及用户按下 OK 按钮之后的最终值。前者是被称作受控组件的技术，你可以从 React 文档的表单部分读到更多关于它的内容。而跟踪后者是因为需要知道用户何时按下了 OK 按钮，以便可以相应地更新 UI。本来也可以使用一个简单的布尔属性“registered”，但是现在的做法允许 UI 提醒用户他们当初是用什么邮箱注册的。

在跟 API 的通信上，也不得不多做一点儿工作：必须指定 HTTP 方法（POST），把请求 body 编码为 JSON，并指定内容的类型。

注意我们是怎样决定到底返回哪个 UI 的。如果用户已经注册了邮件通知，就返回一条简单的消息；如果还没有注册，就渲染这个表单。这是在 React 中十分常见的模式。

看起来我们好像为这一点儿功能做了很多工作，而且还做得很粗糙。如果 API 调用出现了什么错误，我们的错误处理虽然正常，但还不能说是“用户友好”。而且，尽管这个组件可以记住我们注册过哪些度假产品的上线通知，但也是在这个页面上才能记住。如果导航到别处再回来，就又会看到表单了。

为了让这些代码更看得过去，可以采取几个步骤。首先，写一个 API 包装器，让它来处理编码输入和辨别错误这些杂乱的细节。随着使用的 API 端点越来越多，毫无疑问，它带来的回报就会越来越大。现在也有很多流行的 React 表单处理框架，可以大幅减轻表单处理的负担。

“记住”用户已经注册了哪些度假产品，这个问题要更棘手一些。真正对我们有用的，是从度假产品对象中获得注册信息的方法（不管用户是否已经注册过）。然而，这个专门组件并不知道任何关于度假产品的信息，它得到的只是度假产品的 SKU。下一节将讨论状态管理，它给出了这个问题的解决方案。

16.4.6 状态管理

规划和设计一个 React 应用的大部分架构性工作是围绕着状态管理进行的——通常不是单个组件的状态管理，而是在多个组件之间如何共享和协同状态。我们的示例应用确实共享了一些状态：`Vacations` 组件把一个度假产品对象传入 `Vacation` 组件，而 `Vacation` 组件把度假产品的 SKU 传入 `NotifyWhenInSeason` 订阅组件。但目前为止，我们的信息只能在组件树上往下走，如果需要信息往上走该怎么办？

最常见的做法是在组件之间传递负责更新状态的函数。例如，`Vacations` 组件会有一个修改度假产品的函数，可以把它传入 `Vacation` 组件，而在 `Vacation` 里面，也可以把这个函数传入 `NotifyWhenInSeason` 组件。当 `NotifyWhenInSeason` 调用它来修改度假产品时，组件树顶层的 `Vacations` 就会发现有东西被修改了，于是 `Vacations` 组件会得到重新渲染，进而所有下层组件都得到了重新渲染。

这听起来既累人又复杂，有时候确实如此，不过一些技术可以提供帮助。它们多种多样，有时候也比较复杂，这里不能完整地覆盖这部分内容（何况本书也不是关于 React 的），不过我可以给你指出进一步阅读的方向。

Redux

在考虑 React 应用中全面的状态管理方案时，Redux 通常是人们第一个会想到的。它是最先出现的正式状态管理架构之一，现在仍广受欢迎。它的概念极其简单，到现在仍是我最偏爱的状态管理框架。即便最终没有选择 Redux，我还是推荐你看看它的创建者 Dan Abramov 主讲的一系列免费视频课程。

MobX

MobX 紧随 Redux 之后出现。它曾在很短的时间之内就收获了大量的追随者，很可能是第二流行的状态容器，仅次于 Redux。MobX 无疑让代码可以显得更好写，但我还是觉得，随着应用扩展，即使应用的非业务代码会随之增加，Redux 也会在提供一个好框架方面比 MobX 有更大的优势。

Apollo

Apollo 本身并非一个状态管理库，但常常使用它取代状态管理库。它本质上是 GraphQL (REST API 的另一种选择) 的一个前端接口，对于与 React 的集成提供了大量的支持。如果你在使用 GraphQL (或对它感兴趣)，Apollo 就很值得深入研究。

React Context

React 本身通过提供 Context API 也加入了“角逐”，现在已经内建在 React 中了。它实现了一些跟 Redux 一样的东西，但使用起来无关代码会更少。不过我感觉 React Context 还不够健壮，随着应用的增长，Redux 会是更好的选择。

刚开始使用 React 的时候，基本上可以忽略整个应用范围错综复杂的状态管理，但是很快，你会认识到需要更有条理的方式来管理状态。到了那个时候，就会希望深入研究其中一些候选技术，并找出认同的那个。

16.4.7 部署选择

目前为止，我们一直在使用 CRA 内建的开发服务器。它的确是开发时的最好选择，推荐你使用。然而，到了部署的时候，它就不是合适的选择了。幸运的是，CRA 本身包含着构建脚本，可以创建为生产而优化的资源包，然后你就有很多的选择了。当你做好创建部署包的准备时，只需运行 `yarn build`，就会创建出一个 `build` 目录。`build` 目录下的资源文件都是静态的，可以部署到任何地方。

我目前的部署选择是把 CRA 构建放到 AWS S3 的一个 bucket 上，并启用静态网站托管。这当然不是唯一的选择，每个主要的云服务提供商和 CDN 都提供类似的服务。

在这个构建配置中，我们必须创建路由，以便将 API 调用路由到你的 Express 服务器，并且从 CDN 访问到静态资源。就我的 AWS 部署来说，我使用了 AWS 的 CloudFront 来实施这个路由，静态资源从前面提到的 S3 的 bucket 上提供，而 API 请求会被路由到一个 EC2 实例或是 Lambda 上。

另一个选择是让 Express 把所有这些事情都做了。这样做的优势是，可以把你的整个应用都合并进一个服务器里，这样部署起来简单，管理也容易。从伸缩性或性能的角度来说，这么做可能不会很理想，但对于小型的应用来说是一个合理的选择。

要从 Express 提供整个应用的服务，只需把运行 `yarn build` 之后生成的 `build` 目录的内容复制进 Express 应用的 `public` 目录。只要把 `static` 中间件链接进来，它就会自动提供 `index.html` 文件服务，这正是你所需要的。

不妨自己尝试一下：如果你的 Express 服务器仍旧运行在 3033 端口，你就能够访问 `http://localhost:3033`，并可以看到跟你的 CRA 开发服务器提供的一模一样的应用。



想知道 CRA 开发服务器是如何工作的吗？它使用了一个名为 `webpack-dev-server` 的包，这个包在内部使用了 Express。所以到最后又回到 Express 了。

16.5 小结

本章只触及了 React 的皮毛以及它周边的几项技术。如果想深入研究 React，Alex Banks 和 Eve Porcello 合著的 *Learning React* (O'Reilly 出版) 是一个很好的开始。这本书也讨论了

Redux 的状态管理（不过，当前版本没有涉及钩子²）。另外，React 的官方文档也很全面且写得很好。

毫无疑问，SPA 已经改变了我们思考和发布 Web 应用的方式，并且带来了显著的性能提升，尤其是在移动端。即使 Express 是在绝大多数 HTML 还在服务器上渲染的时代编写的，它也从来都没有因此而过时，这是可以确定的。如果说有什么影响，单页应用要求提供的 API 倒是让 Express 又焕发了蓬勃生机。

读完本章你应该清楚了，Web 应用其实并没有改变：无非是让数据在浏览器和服务器之间来回传输。改变的只是所传输的数据的本质，我们要逐渐习惯通过动态的 DOM 操作来修改 HTML。

注 2：2020 年出版的第 2 版包含了钩子。——译者注

第 17 章

静态内容

静态内容指的是应用提供的那些不会在不同请求之间变化的资源。通常包括以下内容。

多媒体

图片、视频和音频文件。当然，也可能即时生成图片文件（视频和音频也是如此，不过要少见得多），但是大多数多媒体资源是静态的。

HTML

如果 Web 应用在使用视图来渲染动态的 HTML，一般就不能认为是静态 HTML 了（虽然出于性能考虑，可以动态生成 HTML，之后缓存它，并作为静态资源提供）。正如所见，SPA 应用通常是发送单个静态 HTML 文件到客户端，这就是把 HTML 看作静态资源最常见的原因。注意，要求客户端使用 .html 扩展名是有些过时的做法，因此现在大多数服务器允许提供不带扩展名的静态 HTML 资源（所以 /foo 和 /foo.html 会返回同样的内容）。

CSS

即使使用的是经过抽象的 CSS 语言，比如 LESS、Sass 或 Stylus，浏览器最终需要的还是普通的 CSS，这是一种静态资源。¹

JavaScript

如果服务器端正在运行 JavaScript，并不意味着就不能有客户端 JavaScript。客户端 JavaScript 被认为是静态资源。当然，现在这个界线开始有些模糊了：要是有一些公共

注 1：在浏览器上使用未编译的 LESS 是可能的，但需要一些 JavaScript “魔法”。而且，这样做对性能有一些影响，所以我不推荐。

代码既想用在后端又想用在客户端该怎么办？有多种方法可以解决这个问题，但无论如何，发往客户端的 JavaScript 通常都是静态的。

二进制下载文件

这个分类可以包含一切：PDF、压缩文件、Word 文档、安装程序，等等。



如果只是在构建一个 API，可能就没有静态资源。如果是这样，则可以跳过本章。

17.1 性能上的考量

静态资源的处理方式可以显著影响网站的实际性能，尤其是网站主要提供多媒体服务的时候。两个最主要的性能考量，一是要降低请求的数量，二是要缩减内容的大小。

在两者之中，降低（HTTP）请求的数量更为关键，尤其是对移动端来说（通过蜂窝网络建立 HTTP 连接的开销要大很多）。减少请求的数量可以通过两种途径来达到：合并资源和浏览器缓存。

合并资源主要从架构和前端方面考虑：小图片应该尽可能地合并成单一文件。合并之后再使用 CSS 来设置图片的偏移和大小，只显示想要的部分。要创建合并的单一文件，我强烈推荐一个免费服务——SpritePad。它使得合并图片简单得不可思议，同时也会生成 CSS。没有比这更容易操作的了。SpritePad 的免费功能对你来说很可能已经够用了，但如果要创建大量的合并文件，也许你会觉得它的付费服务是很值得的。

浏览器缓存也可以通过把常用的静态资源存储在客户端浏览器上的方式来减少 HTTP 请求。尽管浏览器在很大程度上已经把缓存自动化，但它并不是什么“魔法”，而在让浏览器缓存静态资源这件事上，你还有很多可以做也是应该做的。

最后，缩减静态文件的大小也可以提升性能。有些技术是无损的（缩减大小但不损失任何数据），有些技术是有损的（缩减大小的同时也降低了静态资源的品质）。无损技术包括 JavaScript 和 CSS 最小化，以及 PNG 图片优化。有损技术包括提高 JPEG 图片和视频的压缩等级。本章将讨论最小化和打包（打包可以减少 HTTP 请求数量）。



随着 HTTP/2 变得越来越普遍，减少 HTTP 请求的重要性会逐渐降低。HTTP/2 的一个主要的提升就是请求响应复用技术，它降低了并行获取多个资源文件的开销。更多信息请参阅 Ilya Grigorik 的“Introduction to HTTP/2”。

17.2 内容分发网络 (CDN)

当你要把网站推上生产的时候，网站的静态资源必须存放于互联网上的某个地方。你也许习惯于把它们存放在生成所有动态 HTML 的同一台服务器上，目前为止我们的示例应用也是这么做的：使用 `node meadowlark.js` 运行起来的 Node/Express 服务器，除了提供 HTML 服务之外，还提供静态资源服务。不过，如果你想最大化网站的性能（或者希望在将来能够做到），就会希望把静态资源托管到一个 CDN 上。CDN 就是把分发静态资源优化到极致的服务器。它通过专门的响应头（一会儿就会学习）启用了浏览器缓存。

CDN 也可以启用基于地理位置的优化（常称为边缘缓存），也就是说，CDN 可以先找出地理上距当前客户端最近的一台服务器，然后从这台服务器发送静态内容。虽然互联网传输数据很快（虽不是以光速运行，但也很接近），但是传输 100 英里还是比 1000 英里速度要快。虽然单次节省的时间或许很少，但如果考虑到用户数、请求数、资源数，累积起来就显得快了。

大多数静态资源是在 HTML 视图中引用的（通过 `<link>` 标签引用 CSS 文件、`<script>` 引用 JavaScript 文件、`` 引用图片文件，再加上多媒体嵌入标签）。CSS 文件中包含静态引用也是很常见的，比如通常使用的 `background-image` 属性。最后，有时候在 JavaScript 中也会引用静态资源，比如有的 JavaScript 代码会动态地修改或插入 `` 标签或 `background-image` 属性。



在使用 CDN 时，通常不需要担心跨域资源共享（CORS）。在 HTML 中加载的外部资源并不受 CORS 策略的影响，仅需要对通过 Ajax 加载的资源启用 CORS（参见第 15 章）。

17.3 为CDN而设计

网站的架构会影响到 CDN 的使用方式。大多数 CDN 允许配置路由规则，以决定把进入的请求转发到何处。尽管路由规则可以做得任意复杂，但通常划分为两类：一是把访问静态资源的请求转发到一个位置（通常由 CDN 提供），二是把访问动态端点的请求（比如动态页面或 API 端点）转发到其他位置。

选用配置 CDN 是一个很大的主题，这里不准备深入讨论，但会提供一些背景知识，帮助你配置所选择的 CDN。

为了让 CDN 路由的规则尽可能简单，就要让动态资源跟静态资源更易于区分，这样组织应用也是最容易的。尽管使用子域名可以做到，但其带来了额外的难题，使得本地开发更困难了。更容易的做法是使用请求路径，比如说，任何开始于 `/public/` 的都是静态资源，除此之外都是动态资源。如果使用 Express 来生成内容，或使用 Express 来提供单页应用的

API，请求路径的做法就会有所不同。

17.3.1 服务器端渲染的网站

如果你在使用 Express 来渲染动态 HTML，那么“任何开始于 /static/ 的都是静态资源，除此之外都是动态资源。”按这种做法，把所有的（动态生成的）URL 指定成什么都可以（当然，不能是以 /static/ 开头的），而所有的静态资源都会加上 /static/ 前缀：

```
  
Welcome to <a href="/about">Meadowlark Travel</a>.
```

目前为止，本书一直在使用 Express 的 static 中间件，就好像所有这些静态资源都托管在服务器的根目录中一样。也就是说，如果把一个静态资源 foo.png 放到 public 目录下，就要使用 URL 路径 /foo.png 而不是 /static/foo.png 来引用它。当然，也可以在已有的 public 目录下创建 static 子目录，那么 /public/static/foo.png 的 URL 就是 /static/foo.png，但创建 static 子目录的做法似乎显得有点儿愚蠢。幸运的是，如果使用 static 中间件，就可以不这样做。只需要在调用 app.use 时指定一个不同的路径：

```
app.use('/static', express.static('public'))
```

现在就可以在开发环境中使用跟将来的生产环境一样的 URL 结构了。如果小心地保持 public 目录跟 CDN 中的同步，就可以在开发和生产两个地方引用到一样的静态资源，并且能够在开发和生产之间无缝切换。

当为 CDN 配置路由时（为此，你将不得不查阅 CDN 的文档），路由应该如表 17-1 所示。

表17-1：CDN路由配置

URL路径	路由目标/文件源
/static/*	静态 CDN 文件存储
/* (除上面以外的任何路径)	Node/Express 服务器、代理，或负载均衡

17.3.2 单页应用

单页应用通常与服务器端渲染的网站相反：只有 API 会被路由到服务器（例如，任何前缀为 /api 的请求），除此之外所有的请求都会被路由到静态文件存储。

正如在第 16 章中所看到的，可以通过某种方式来为应用创建一个生产包，这个包将包含所有的静态资源，并且需要上传到 CDN 上。然后只需要保证 API 对应的路由得到了正确的配置。因此路由应该如表 17-2 所示。

表17-2：正确的CDN路由配置

URL路径	路由目标/文件源
/api/*	Node/Express 服务器、代理，或负载均衡
/* (除上面以外的任何路径)	静态 CDN 文件存储

我们已经看到了，要想无缝地从开发切换到生产应该怎样组织应用，现在来看看缓存究竟是怎么做的，以及它是怎样提升性能的。

17.4 缓存静态资源

不管你是使用 Express 来提供静态资源，还是使用 CDN，浏览器都会使用以下几个 HTTP 响应头来确定何时以及如何缓存静态资源，理解它们是很有帮助的。

`Expires/Cache-Control`

这两个响应头会告诉浏览器这个资源可以被缓存的最长期限。而浏览器会认真地对待它们的命令：如果它们告诉浏览器某个文件需要缓存一个月，那么在一个月之内，只要文件还在缓存里，浏览器就不会重新下载它。但是出于某些无法控制的原因，浏览器可能会从缓存里永久地删除图片文件，明白这一点很重要。例如，用户可以手动清理缓存，又或者，浏览器清理掉某个资源是为了给用户访问得更频繁的其他资源腾出空间。这两个响应头只需要用一个，其中 `Expires` 适用范围更广，所以应该优先使用。如果资源还在缓存里且没有过期，那么浏览器根本不会发出 GET 请求，因而性能就得到了提升，尤其是在移动端。

`Last-Modified/ETag`

这两个标记提供了某种版本化机制：浏览器如果需要获取某个资源，就会在实际下载之前先检查这些标记。它仍然会向服务器发送一个 GET 请求，但如果返回的这两个响应头告诉浏览器这个资源并没有变更，浏览器就不会开始文件的下载。正如其名，`Last-Modified` 允许指定这个资源上次修改的日期。`ETag` 则可以使用任意的字符串，通常是版本字符串或内容散列。

当提供静态资源服务时，应该使用 `Expires` 头，或者使用 `Last-Modified` 和 `ETag` 二者之一。Express 内建的 `static` 中间件设置了 `Cache-Control`，但并没有处理 `Last-Modified` 和 `ETag`。因此，虽然 `static` 中间件比较适合于开发，但对于生产部署来说不是很好的选择。

如果选择将静态资源托管在 CDN 上，比如亚马逊的 CloudFront、微软的 Azure、Fastly、Cloudflare、Akamai，或者 StackPath，好处是它们会处理大部分的细节。你可以对这些细节进行微调，但以上服务提供的默认设置通常都是开箱即用的。

17.5 变更静态内容

缓存显著提升了网站的性能，但也有不利的一面，尤其是当你修改了某些静态资源时，在浏览器缓存的版本过期之前，客户端可能看不到修改。谷歌推荐至少缓存一个月，缓存一年则更好。假设有一个用户每天都在同一个浏览器上使用你的网站，那么他得整整一年才能看到更新。

这种情形显然不是我们所希望的，而且你也不能逐一通知你的用户去清理缓存。解决方案是使用缓存破坏。缓存破坏这项技术可以控制用户的浏览器何时重新下载某个资源。通常缓存破坏是在资源名上带上一个版本号（main.2.css 或 main.css?version=2），或者带上某种散列（main.e16b7e149dccfcc399e025e0c454bf77.css）。不管是哪一种做法，变更这个资源时资源名也会随之改变，而浏览器便知道了需要去下载它。

对多媒体资源也可以这样处理。不妨拿 logo (/static/img/meadowlark_logo.png) 做例子。为了最大化性能，我们把它托管在 CDN 上，指定一年后过期，之后如果修改这个 logo，那么用户在满一年之前可能都不会看到变更的 logo。然而，如果把它重命名为 static/img/meadowlark_logo-1.png（并且在 HTML 中体现出这个文件名修改），浏览器就只能下载它，因为它看起来是一个新的资源。

如果使用一个单页应用的框架，类似 `create-react-app` 这样的工具就会提供一个构建步骤，用以创建适合生产的资源包文件，并且会在这些资源包的文件名上追加一个散列。

如果是从零开始的项目，可以深入探究一下打包工具，这是 SPA 框架内部需要使用的。打包工具把 JavaScript、CSS 和其他类型的静态资源合并成尽可能少的文件，并把合并后的文件最小化。虽然打包工具的配置是个很大的主题，但好在有大量的优质文档可以查阅。目前而言，最流行的打包工具有以下几种。

Webpack

Webpack 是第一批火起来的打包工具之一，现在仍然追随者众多。它非常高级，但这种高级是有代价的：它的学习曲线很陡峭。不过，对它有一些基础的了解还是很有用的。

Parcel

Parcel 是一个后来者，但引起了极大的关注。它的文档做得相当好，速度也极快，不过它最突出的优势是学习曲线最短。如果你追求的是快速完成工作，又不想小题大做，就从它开始吧。

Rollup

Rollup 介于 Webpack 和 Parcel 之间。它很像 Webpack，十分稳定并且特性很多。但要比 Webpack 更易于上手，也不像 Parcel 那么简单。

17.6 小结

即便是像静态资源这样看起来很简单的东西，也可能带来很多的麻烦。然而，静态资源几乎代表着实际给用户传输的数据块，因此花一些时间来优化它们，收益将相当可观。

还有一个前面没有提到的可行的静态资源方案是，一开始就把它们托管到 CDN 上，并且在你的视图和 CSS 中始终使用资源的完整 URL 路径。这样做好处是简单，但如果你周末想去参加一个在没有互联网连接的林中小屋举办的“黑客马拉松”，那就麻烦了。

如果精细的资源打包和文件最小化工作对你的应用来说并不值得，则可以简化它们，以便节省一些时间。尤其是如果网站只包含一两个 JavaScript 文件，而所有的 CSS 都在一个文件里，则很可能就会完全跳过打包这一步。不过真实世界的应用都有随着时间而增长的趋势。

不管选择什么技术来提供静态资源，我都强烈建议把它们跟动态资源分离开，最好是托管到 CDN 上。如果你觉得这样很麻烦，那我可以向你保证它远没有听起来那么难，尤其是如果在部署系统上花上一点儿时间，就会发现把静态资源部署到一个位置，把应用部署到另一个位置，整个过程都是自动化的。

如果担心托管到 CDN 上的成本，建议先看看目前的托管费用。大部分的托管服务提供商按带宽收费，即使你并不知道这条计费规则。然而，如果突然间网站被 Slashdot 提到了，可能就会收到意料之外的托管服务的账单。CDN 服务通常是按用多少付多少来设置的。下面这个例子可作为参考：我曾经为一个中等规模的地区性公司管理一个网站，这个网站每个月使用 20GB 的流量，而花在托管静态资源上的费用只有几美元（它是一个偏重多媒体资源的网站）。

通过把静态资源托管到 CDN 上所获得的性能提升很显著，而花费和不便非常小，因此强烈建议这样做。

第18章

安全

今天，大多数网站和应用会有某种类型的安全需求。如果允许用户登录网站，或者要存储用户的个人识别信息（PII），就需要为网站实现某种安全机制。本章将讨论 HTTP 安全（HTTPS）和认证机制。HTTPS 为构建一个安全的网站打下了坚实的基础。对于认证机制，本章主要关注第三方的认证。

安全是一个很大的主题，足以写一本书了。因此，本书准备把关注点放在如何利用现有的认证模块上。当然可以自己写认证系统，但这会是一项艰巨的任务。况且，有几个很好的理由支持优先使用第三方的登录方法，本章稍后会讨论。

18.1 HTTPS

提供安全服务的第一步就是使用 HTTPS。互联网的本质使得某个第三方要拦截客户端跟服务器之间传输的包是可能的。HTTPS 对这些包进行了加密，攻击者想要获得正在传输的信息极其困难。（我说的是“极其困难”，而不是“不可能”，因为“完美安保”是不存在的。不过对于银行业、企业安全和医护领域来说，HTTPS 足够安全了。）

可以把 HTTPS 看作网站安全加固的某种基础设施。它并不提供认证，但为认证做了基础性的工作。例如，认证系统很可能需要传输密码，如果密码未经加密就传输，无论认证做得多么高级，系统都不安全。安全的强度是链条中最弱的一环，而链条中的第一环就是网络协议。

在 HTTPS 协议中，服务器有一个公钥证书，有时候也叫 SSL 证书，这个证书是 HTTPS 协议运行的基础。目前 SSL 证书的标准格式为 X.509。证书的背后是负责颁发证书的证书中

心（CA），它们先创建出受信任的根证书以供浏览器提供商们使用。当你安装一个浏览器时，就会把这些受信任的根证书包含进来，这就建立了 CA 跟浏览器之间的信任链。为了让这条信任链起作用，你的服务器必须使用由某个 CA 颁发的证书。

所以为了提供 HTTPS，需要一个来自 CA 的证书，那么怎么做才能得到这样一个证书呢？大体来讲，你可以自己生成、从免费的 CA 获取，或者从商业 CA 购买。

18.1.1 生成自己的证书

生成自己的证书很容易，但通常只适用于开发和测试（而且还可能是局域网内的开发）。由于证书中心建立了层级结构，浏览器只会信任由已知 CA（你应该不在其中）生成的证书。如果网站使用了浏览器未知 CA 的证书，浏览器就会用非常严肃的话语警告你，你正在跟一个未知的（因而也是不可信任的）实体建立安全连接。在开发和测试中，这是没问题的：你和你的团队都知道证书是自己生成的，因此浏览器的行为也在意料之中。但如果把这样一个网站部署到生产上让大众访问，那么大众都会离它而去的。



如果控制了浏览器的分发和安装，就可以在安装浏览器时自动安装自己的根证书。这样的话，当使用这个浏览器的人连接到你的网站时，就可以避免收到警告信息。可是，控制浏览器的分发和安装并不是一件容易的事，而且这仅适用于你能控制这个浏览器的环境。除非有非常过硬的理由，否则这种做法通常并不值得。

要生成自己的证书，需要一个 OpenSSL 实现。表 18-1 说明了如何获取 OpenSSL。

表18-1：不同的平台下获取OpenSSL的方法

平 台	方 法
macOS	<code>brew install openssl</code>
Ubuntu、Debian	<code>sudo apt-get install openssl</code>
其他 Linux	从 OpenSSL 官网下载，然后提取 tarball 并遵照说明
Windows	搜索 OpenSSL for Windows 下载



如果是 Windows 用户，可能需要先知道 OpenSSL 配置文件的位置，由于 Windows 的路径名的原因，事情可能会有些麻烦。保险的做法是先找到 `openssl.cnf` 文件（通常在安装目录下面的 `share` 目录里），然后在运行 `openssl` 命令之前，先设置 `OPENSSL_CONF` 环境变量：`SET OPENSSL_CONF=openssl.cnf`。

安装完 OpenSSL 之后，就可以生成私钥和公开证书了：

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout meadowlark.pem  
-out meadowlark.crt
```

这个命令会向你询问各种细节，比如国家（地区）代码、城市、州、全限定域名（FQDN，也叫通用名或全限定主机名）和邮件地址。既然这个证书是用于开发 / 测试的目的，提供的值就不是特别重要了（事实上，这些值都是可选的，但是如果都不填，得到的这个证书在浏览器看来就更可疑了）。证书的 FQDN 是浏览器用来识别域名的。因此，如果你在使用 localhost，FQDN 就可以使用它，你也可以使用服务器的 IP 地址，或者服务器名（要是有的话）。如果证书的通用名跟在 URL 中使用的域名不匹配，那么加密仍然会正常工作，但对于这个不匹配，浏览器会发出额外的警告信息。

如果你对这个命令的细节感到好奇，可以从 OpenSSL 的官方文档页上详细了解。值得指出的是，`-nodes` 选项跟 Node 没有任何关系，实际上只表示“no DES”，意味着私钥并非是 DES 加密的。

这个命令的结果是得到了 meadowlark.pem 和 meadowlark.crt 这两个文件。其中“隐私增强邮件”（PEM）文件就是私钥，不该被客户端访问到。而 CRT 文件是自签名的证书，它将被发送到浏览器，用以建立安全连接。

除了自己运行命令，有些网站也会提供免费的自签名证书，读者可自行查找。

18.1.2 使用免费的证书中心

HTTPS 以信任为基础，但一个不幸的事实是，如果要在互联网上获取信任，最容易的办法就是花钱购买。不过，这也并非是糊弄人的办法：毕竟建立起安全基础设施、为证书做担保，以及维护跟浏览器提供商的关系的工作都成本高昂。

对于生产级的证书来说，购买证书并非是唯一合法选择：作为一个基于开源的免费自动化 CA，Let's Encrypt 已经成了非常好的选择。事实上，除非你已经在某个设施上有了投资，而这个设施把提供免费或低价的证书作为托管服务（例如 AWS）的一部分，否则 Let's Encrypt 就是一个非常好的选择。Let's Encrypt 唯一的不足是其证书的最大生命周期只有 90 天。但是这个不足可以弥补，因为 Let's Encrypt 使自动刷新证书变得非常容易，而且它会推荐建立一个自动化过程，每 60 天运行一次，保证证书不会过期。

所有主要的证书供应商（比如 Comodo 和 Symantec）都可以提供免费的体验性证书，有效期从 30 天到 90 天不等。如果想测试一个商业证书，这是合理的选择，但为了保证业务的连续性，需要在体验性证书过期之前购买一个新证书。

18.1.3 购买证书

当前，每个主要浏览器分发的根证书大约有 50 个，其中 90% 由 4 家公司持有：Symantec（收购了 VeriSign）、Comodo Group、Go Daddy 和 GlobalSign。直接从 CA 购买证书相当昂贵：通常都要 300 美元 / 年起（尽管也有少于 100 美元 / 年的）。而不那么昂贵的选择是通

过零售商，能够以低至 10 美元 / 年甚至更低的价格获得一个 SSL 证书。

花了这些钱究竟得到了什么？为什么要为一个证书花上 10 美元、150 美元或 300 美元（甚至更多）呢？把这些搞明白很重要。首先要理解的是，不管是 10 美元还是 1500 美元的证书，它们所提供的加密级别并无差别。这是那些昂贵的证书中心宁愿你不知道的，尽管它们的市场营销在不遗余力地模糊这个事实。

如果准备选择一个商业的证书提供商，建议你在做决定时考虑以下 3 点。

客户支持

如果你的证书遇到过什么问题，不管是浏览器支持（如果在你的客户的浏览器上证书被标记为不受信任，客户会告诉你的）、安装问题，还是证书延期上的麻烦，就会知道好的客户支持有多么重要。这也是你愿意购买一个更昂贵的证书的原因之一。通常，托管服务的供应商会零售证书，而且根据我的经验，他们会提供更高级别的客户支持，因为他们也想让你成为托管服务的客户。

单域名证书、多子域名证书、通配符证书和多域名证书

最便宜的证书通常是**单域名证书**。这听起来并不坏，但要知道，它意味着如果为 meadowlarktravel.com 购买了一个证书，这个证书则不能用于 www.meadowlarktravel.com，反之亦然。因为这个原因，我倾向于避免单域名证书，尽管它在极端的预算意识下也是一个好选择（可以随时设置重定向，把请求导向合适的域名）。**多子域名证书**的好处则在于，只需购买一个证书就覆盖了 meadowlarktravel.com、www.meadowlarktravel.com、blog.meadowlarktravel.com 和 shop.meadowlarktravel.com 等子域名。但它的不足之处是需要提前知道要使用哪些子域名。

如果觉得在一年之中需要不停地增加或使用不同的子域名（为了支持 HTTPS），可能使用**通配符证书**更好一些，当然通常会更昂贵。通配符证书可以用于任何子域名，而且永远不需要指定这些子域名是什么。

最后，还有**多域名证书**。与通配符证书一样，多域名证书通常会更昂贵。这些证书支持完整的多个域名，举例来说，使用一个证书就可以支持 meadowlarktravel.com、meadowlarktravel.us、meadowlarktravel.ca 这些域名以及它们的 www 子域名。

域名证书、组织证书和扩展验证证书

域名证书可以给你信心——正在操作的域名的确是自己认为的那个。**组织证书**可以让你放心——自己的确是在跟这个组织打交道。域名证书和组织证书要更难获得：通常会涉及一些文书，你必须提供州和（或）联邦的商业名称记录、实际地址，等等。不同的证书供应商会要求不同的文件，因此请务必询问你的证书提供商，要获取某个证书需要提供什么材料。最后是**扩展验证证书**，它们是 SSL 证书中最昂贵的。跟组织证书类似，扩展验证证书也验证组织是否存在，但是要求更高标准的证据，甚至要求有成本高昂的

审计（不过这个要求似乎越来越少见了），以便建立数据安全实践。它们的价格可以低至 150 美元一个域名。

我建议要么使用不那么昂贵的域名证书，要么使用扩展验证证书。尽管组织证书能验证组织的确存在，但在浏览器上并不会显示出不同之处，因此根据我的经验，除非用户实际去检查这个证书（很少会这样），否则组织证书跟域名证书外表上不会有差异。而扩展验证证书通常会给用户显示一些线索（比如把地址栏显示成绿色，并且组织的名字紧跟着 SSL 图标显示），以提示他们正在处理的是一个合法的业务。

如果你以前处理过 SSL 证书，也许会奇怪我为什么不提证书保险。证书保险是价格差异的一个因素，我忽略它是因为它所担保的事情几乎不可能发生。这个保险是说，如果有人在你的网站上做了一笔交易而遭受了财务损失，并且能够证明损失是因不充分的加密而导致，保险就会给这个人赔偿。如果你的应用涉及了财务交易，那么有人因为他们的损失而试图对你采取法律行动当然是有可能的，但是要说损失是因不充分的加密而导致，则根本不可能。如果因为一个在线服务导致了财务损失，从而需要向这个公司寻求赔偿，我绝对不会试图去证明该服务的 SSL 加密是有问题的。如果两个证书的差别只体现在价格和承保范围上，那就买便宜的那个。

购买一个证书首先从创建私钥开始（就如同前面对自签名证书所做的那样）。然后生成一个证书签名请求（CSR），在购买过程中把它上传上去（证书颁发者对此会给出具体的步骤）。要注意证书颁发者永远不会访问你的私钥，你的私钥也永远不会在互联网上传输，因此就保护了私钥的安全性。证书颁发者会把你的证书发送给你，证书扩展名是 .crt、.cer 或 .der [.der 扩展名比较少见，它的证书是“可区分编码规则”（DER）的格式]。你也会收到这条证书链上的所有其他证书。使用邮件传输证书是安全的，因为如果没有私钥，证书将无法正常工作。

18.1.4 为 Express 应用启用 HTTPS

可以修改 Express 应用，让它通过 HTTPS 访问网站。但在实践和生产中，这是极不寻常的，我们在下一节会了解到这些。不过，从对 HTTPS 的高级应用、测试以及理解的角度，知道如何提供 HTTPS 服务是很有用的。

获得了私钥和证书之后，在应用中使用是很容易的。再来看一下，我们一直是如何创建服务器实例的：

```
app.listen(app.get('port'), () => {
  console.log(`Express started in ${app.get('env')} mode ` +
    `on port ${app.get('port')}.`)
})
```

把它切换为 HTTPS 很简单。我推荐你把私钥和 SSL 证书放入一个名为 ssl 的子目录里（尽

管把它们放在项目的根目录下也很常见)。然后用 https 模块来取代 http 模块，并把一个 options 对象传入 createServer 方法：

```
const https = require('https')
const fs = require('fs') // 通常放在文件顶部

// ……其余的应用配置

const options = {
  key: fs.readFileSync(__dirname + '/ssl/meadowlark.pem'),
  cert: fs.readFileSync(__dirname + '/ssl/meadowlark.crt'),
}

const port = process.env.PORT || 3000
https.createServer(options, app).listen(port, () => {
  console.log(`Express started in ${app.get('env')} mode ` +
    `on port ${port}.`)
})
```

这样就可以了。假设你的服务器仍旧在 3000 端口运行，那现在就可以连接到 `https://localhost:3000` 了。如果试图连接到 `http://localhost:3000`，连接就会超时。

18.1.5 有关端口的说明

不论你是否知道，当你访问某个网站时，即使在 URL 中并没有指定，也总是要连接到某个特定的端口。如果没有指定端口，对 HTTP 来说就认为是 80 端口。事实上，即使明确指定了 80 端口，大多数浏览器也不会显示这个端口号。

与此类似，对 HTTPS 来说，标准端口是 443。浏览器的行为也是类似的，不会显示这个端口号。

如果 HTTP 不使用端口 80 或 HTTPS 不使用端口 443，就需要明确指定端口和协议，这样才能正确地连接。在同一个端口上无法同时运行 HTTP 和 HTTPS (尽管技术上是可能的，但没有理由这么做，而且实现起来也非常复杂)。

还有另一件事情需要知道，在大多数操作系统中，端口 1-1023 需要提升权限才能打开。例如，在 Linux 或 macOS 机器上，如果试图在端口 80 上启动应用，那么很可能得到一个 EACCES 错误。为了打开端口 80 或 443 (或任何低于 1024 的端口)，需要使用 `sudo` 命令来提升权限。所以，如果没有管理员权限，就不能在端口 80 或 443 上直接启动服务器。

除非管理的是自己的服务器，否则很可能不能以 root 用户身份来访问托管账户，那么当你想要在端口 80 或 443 上运行时该怎么办呢？一般来说，托管服务提供商都会提供某种类型的代理服务，以较高的权限运行并会把请求转发到应用，而应用是在非特权端口上运行的。下一节将继续了解。

18.1.6 HTTPS与代理

我们已经看到了，在 Express 中使用 HTTPS 非常容易，而且对于开发环境来说，运行也很正常。可是，等到你想要扩展网站以便应付更大的流量的时候，就会希望使用一个像 NGINX 这样的代理服务器（参见第 12 章）。如果网站是在共享托管环境中运行，那么几乎可以肯定的是，会存在一个代理服务器，把请求路由到你的应用。

如果你在使用一个代理服务器，那么客户端（用户的浏览器）会跟这个代理服务器而不是服务器通信。而这个代理服务器跟应用的通信很有可能是基于常规 HTTP（因为应用和代理服务器会共同运行于一个受信任的网络）。你会常常听到人们说“**HTTPS 终结于代理服务器**”，或者“**代理服务器做的就是终结 SSL 的工作**”。

大多数情况下，不管是你还是托管服务提供商，一旦正确地配置了处理 HTTPS 请求的代理服务器，就不需要做任何额外的工作了，除非你的应用需要同时处理安全和不安全的请求，不过这是特例。

对于需要同时处理安全和不安全的请求这个问题，有两个解决方案。第一个方案是直接配置代理服务器，让它把所有的 HTTP 流量都重定向到 HTTPS，从根本上强制所有跟应用的通信都基于 HTTPS。现在这种方法比以前普及多了，也的确是一个比较容易的解决方案。

第二个方案是把用于客户端跟代理之间通信的协议传送到应用服务器。通常的传送方式是通过 `X-Forwarded-Proto` 请求头。例如，在 NGINX 中设置以下请求头：

```
proxy_set_header X-Forwarded-Proto $scheme;
```

然后，可以在应用中测试，看看协议是否是 HTTPS：

```
app.get('/', (req, res) => {
  // 以下基本等价于: if(req.secure)
  if(req.headers['x-forwarded-proto'] === 'https') {
    res.send('line is secure')
  } else {
    res.send('you are insecure!')
  }
})
```



在 NGINX 中，HTTP 和 HTTPS 都有各自的 `server` 配置块。如果忘了在 HTTP 的 `server` 配置块里设置 `X-Forwarded-Protocol`，就为客户端欺诈打开了方便之门。很可能连接本来是不安全的，但客户端通过设置这个头信息，让应用误以为连接是安全的。如果采用上面的方案，始终要记得设置 `X-Forwarded-Protocol` 请求头。

使用代理时，Express 可以提供一些便利属性，让代理更加“透明”（获得了代理的好处，但好像又没有使用代理）。为了达到这一点，需要告诉 Express 信任这个代理：使用

`app.enable('trust proxy')`。如此调用之后，`req.protocol`、`req.secure` 和 `req.ip` 呈现的将是客户端到代理而不是到应用的连接。

18.2 跨站请求伪造

跨站请求伪造（CSRF）攻击利用了这样一个事实：用户通常会信任他们的浏览器，并且会在一个浏览器会话里访问多个网站。在 CSRF 攻击中，来自恶意网站的脚本会向目标网站发出请求，而如果你已经登录了目标网站，恶意网站就可以成功地访问到目标网站的敏感数据。

为了防止网站遭受 CSRF 攻击，必须要有一种办法来确认请求的确是从你的网站发出的。我们确认的办法就是给浏览器传一个唯一的令牌，当随后浏览器提交表单时，服务器会检查并确认令牌是否匹配。中间件 `csurf` 可以创建令牌和验证，你需要做的就是确保令牌包含在发到服务器的请求中。先安装 `csurf` 中间件 (`npm install csurf`)，然后把它链进来，并在 `res.locals` 中加入一个令牌。请确保是在 `body-parser`、`cookie-parser` 和 `express-session` 之后链入 `csurf` 中间件：

```
// 必须在body-parser、cookie-parser和express-session之后链入
const csrf = require('csurf')

app.use(csrf({ cookie: true }))
app.use((req, res, next) => {
  res.locals._csrfToken = req.csrfToken()
  next()
})
```

`csurf` 中间件给 `request` 对象增加了 `csrfToken` 方法。并非一定要把它赋给 `res.locals`，也可以明确地把 `req.csrfToken()` 传给每个需要它的视图，但赋给 `res.locals` 通常可以减少工作量。



注意，这个包叫作 `csurf`，但涉及大部分变量和方法的是 `csrf`，并没有“u”。很容易在这里出错，所以要小心这个元音字母。

现在，所有的表单（以及 AJAX 调用）里都需要提供一个名为 `_csrf` 的字段，它必须要跟生成的令牌一致。看看如何把它加入表单里：

```
<form action="/newsletter" method="POST">
  <input type="hidden" name="_csrf" value="{{_csrfToken}}>
  Name: <input type="text" name="name"><br>
  Email: <input type="email" name="email"><br>
  <button type="submit">Submit</button>
</form>
```

`csurf` 中间件会处理其余的事情：如果 body 包含了一些字段，但没有合法的 `_csrf` 字段，它就会抛出一个错误（请确保你的中间件里有一个处理错误的路由）。也可以试着去掉 `_csrf` 隐藏域，看看会发生什么。



如果你有一个 API，很可能就不希望 `csurf` 中间件妨碍到它。如果希望限制从其他网站到 API 的访问，就应该研究一下某个 API 库（比如 `connect-rest`）的“API key”功能。要想避免 `csurf` 妨碍到某个中间件，就在链入 `csurf` 之前链入这个中间件。

18.3 认证

认证是一个庞大且复杂的话题。而且，对大多数出色的 Web 应用来说，它也是至关重要的一个部分。我能够给你传授的最重要的一条智慧是“不要想着自己去做认证”。对于设计一个安全的认证系统来说，涉及的种种考量太多、太复杂了。请看一下自己的名片，如果上面没有印着“安全专家”，那么很可能你还没有做好这个准备。

我并非说不应该尝试去理解应用中的安全系统，只是建议不要尝试自己去建立它。对于下面我推荐的认证技术，你可以放心地去研究它们的开源代码。这些开源代码无疑可以加深理解，并且让你明白，为什么不应该贸然地独立承担这项任务。

18.3.1 认证与授权

虽然这两个词经常被互换着用，但它们是有差别的。认证指的是核对用户的身份。也就是说，要核对他们的确是其所声称的身份。授权指的是要确定一个用户对什么有访问、修改或查看的权限。例如，客户应该有访问他们账户信息的权限，而草地鹨旅游的一个雇员应该有访问他人账户信息或售货单的权限。



认证（authentication）常常被缩写为 authN，而授权（authorization）常常被缩写为 authZ。

通常是先做认证（但并不总是如此），然后再确定授权。授权可以是非常简单（已授权 / 未授权）或非常宽泛的（一般用户 / 管理员），也可以是非常精细的，分别针对不同的账号类型指定读、写、删除和更新的权限。授权系统的复杂性由应用的类型决定。

正因为授权如此依赖于应用的细节，所以本书中只给出了一个大概的实现，只使用一个非常宽泛的认证方案（客户 / 雇员）。我会频繁使用缩略词“auth”，但仅限于在上下文中没有歧义的时候，即它的意思是“认证”还是“授权”很明确，或者意思是哪一个不重要。

18.3.2 使用密码认证的问题

使用密码认证的问题在于，要求用户凭空想出一个密码，而每个安全系统的强度都会取决于密码设置中最弱的一环。众所周知，人类很难想出安全的密码。2018年的一个安全漏洞分析显示，使用最多的密码是“123456”，其次是“password”。即使是在安全意识深受重视的2018年，人们还是在选择极其糟糕的密码。如果要求某种密码格式，比如说，至少有一个大写字母、一个数字和一个标点符号，人们就会使用“Password1!”这样的密码。

即使系统对照着一个常用密码清单做核查，也无法避免这个问题。就算人们不得不使用高质量的密码，也会把这些密码写进文本文件里，未经加密就存放在计算机中，或者通过电子邮件发送给别人。

总之，作为应用的设计者，对于这个问题你没有太多可以做的。不过，从督促人们使用更安全的密码的角度，你还是可以采取一些措施的。一是依靠一个第三方的认证，由第三方的认证来处理这个问题。二是让登录系统更适应一些密码管理服务，比如1Password、Bitwarden和LastPass。

18.3.3 第三方认证

第三方认证得益于这样一个现实：在当今的互联网上，几乎每个人会拥有至少一个主要服务的账号，这些主要服务包括谷歌、Facebook、Twitter或LinkedIn。所有这些服务都提供一个机制，让你得以通过它们来认证并识别用户。



第三方认证常常被称作联合认证或委托认证，这些术语很大程度上可以互换，而联合认证通常跟“安全断言标记语言”(SAML)和OpenID关联起来，委托认证常常跟OAuth关联起来。

第三方认证有3大优势。第一个优势是减轻了认证的负担。你不必费心地去认证用户个人，只需跟受信任的第三方交互。第二个优势是减轻了密码疲劳，即拥有太多账号所带来的压力。我使用的是LastPass这个密码管理工具，我的密码库有将近400个密码。作为软件技术的专业人员，可能我会比一般的互联网用户拥有更多的账号，但即便是一个寻常的互联网用户，拥有几十个乃至几百个互联网账号也不足为奇。最后一个优势是，第三方认证是无障碍的：用户可以使用他们已有的身份信息，从而更快速地开始使用你的网站。通常，如果用户发现必须重新创建一组用户名和密码，就会干脆不用了。

如果没有使用密码管理器，那么很有可能你在大多数网站上使用的是同一个密码（很多人对于银行等网站使用一个“安全”的密码，对于其余的网站则使用一个“不安全”的密码）。这样做问题是，如果使用这个密码的网站之一被攻破了，那么你的密码就被人知道了，然后黑客就可以在别的服务上尝试这个密码。这就相当于把所有的鸡蛋都放在了一个篮子里。

不过，第三方认证也有其不足之处。尽管难以置信，还是存在着一些没有谷歌、Facebook、Twitter 或 LinkedIn 账号的人。就算拥有这些账号，但出于不信任（或隐私）的原因，也有可能有些人不愿意使用这些账号登录你的网站。对于第三方认证的不足这个问题，很多网站会鼓励用户使用一个已有的账号，同时对于那些没有账号（或者不想使用这些账号来访问你的服务）的人，允许他们创建一个新的账号。

18.3.4 在数据库里存储用户信息

不管是否依靠一个第三方来认证用户，你都希望在自己的数据库里记录用户的信息。例如，如果使用 Facebook 认证，那么验证的只是用户的 ID。但如果需要保存这个用户的一些设置，就不能通过 Facebook 认证做到，而是必须把这个用户的信息存储到你自己的数据库里。还有，很可能你会希望给用户关联一个电子邮件地址，而用户也许不希望使用已经用于 Facebook（或任何你使用的第三方认证服务）的那个电子邮件地址。最后，把用户信息存储进数据库，再自己执行认证，前提是愿意提供这个选择。

既然如此，先创建一个用户模型 models/user.js：

```
const mongoose = require('mongoose')

const userSchema = mongoose.Schema({
  authId: String,
  name: String,
  email: String,
  role: String,
  created: Date,
})

const User = mongoose.model('User', userSchema)
module.exports = User
```

然后修改 db.js，在这个抽象层加上合适的方法（如果你在使用 PostgreSQL，我就把它留作一个练习，你自己实现一下）：

```
const User = require('./models/user')

module.exports = {
  //...
  getUserId: async id => User.findById(id),
  getUserByAuthId: async authId => User.findOne({ authId }),
  addUser: async data => new User(data).save(),
}
```

回忆一下，MongoDB 数据库中的每个对象都有自己的唯一 ID，存储于对象的 `_id` 属性中。不过，这个 ID 是由 MongoDB 控制的，而我们需要把用户记录跟第三方 ID 映射起来，所以需要给用户记录加上我们自己的名为 `authId` 的 ID 属性。由于将使用多种认证策

略，因此为了避免冲突，这个 ID 属性将是认证策略类型和第三方 ID 的组合。例如，一个 Facebook 用户的 authId 可能为 facebook:525764102，而一个 Twitter 用户的 authId 可能为 twitter:376841763。

在示例中，我们将使用两个角色：“客户”和“雇员”。

18.3.5 认证与注册及用户体验

认证指的是核对用户的 ID，要么是使用某个受信任的第三方来核对，要么是通过提供给用户的身份凭证（比如用户名和密码）来核对。而注册指的是用户在网站上获得一个账号的过程（从我们的视角来看，如果数据库里创建了一条用户记录，那么注册就完成了）。

当用户第一次加入你的网站时，应该让他们明白自己正在注册。如果我们在使用一个第三方认证系统，当用户成功地通过认证之后，就可以给他们完成注册，而无须让他们知道。不过，这通常不是一个好做法。如果用户是在注册，就应该让他们知道（不管他们是否是通过第三方来认证的）。如果以后用户想注销账号，也应该提供一个明确的机制。

在用户体验上需要考虑的一种情形是“第三方混淆”。如果用户在一月份使用 Facebook 注册了你的服务，然后在七月份又来了，这时屏幕上显示着各种登录方式可供选择，包括 Facebook、Twitter、谷歌或 LinkedIn，用户可能会忘记当初是用哪个账号注册的，这很正常。这是第三方认证带来的一个隐患，对此你所能做的很有限。因此让用户提供电子邮件地址又有了一个合适的理由：如果用户留了电子邮件地址，你就可以给用户一个选择，即是否通过邮件查看他们的账号，如果要查看，系统就会给那个地址发一封邮件，告诉用户当初是使用哪个服务认证的。

如果你觉得已经对用户会使用哪些社交网络有了充分的了解，就可以让其中一个成为主要的认证服务，从而减少“第三方混淆”这个问题。例如，如果相信大多数用户会有一个 Facebook 账号，就可以放一个“使用 Facebook 登录”的大按钮，然后，使用一个更小的按钮，甚至一个文本链接说“使用谷歌/Twitter/LinkedIn 登录”。这样就可以减少“第三方混淆”的出现。

18.3.6 Passport

Passport 是用于 Node/Express 的一个认证模块，广受欢迎，而且非常健壮。Passport 并没有限定于任何单一的认证机制，相反，它基于可插拔认证策略这样一种思想（包括本地策略，如果你不想使用第三方认证的话）。一下子把认证的信息流程都弄明白压力会有点儿大，所以先从一种认证机制开始，以后再增加别的。

有一个很重要的细节要明白，如果使用了第三方认证，那么应用将永远不会接收到密码，密码完全由第三方来处理。这是好事，这样就把处理和存储密码的负担转移到了第三方。¹

如果使用了第三方认证，整个认证过程就会依赖于重定向（如果你的应用从未接收到用户在第三方服务的密码，那么重定向是必须的）。起初，你可能会感到困惑，为什么可以把 localhost 的 URL 传给第三方，而且还能成功认证（毕竟，处理了请求的第三方服务器并不会知道你的 localhost）。这是因为第三方只是指示你的浏览器进行重定向，而浏览器位于自己的网络中，因此可以重定向到本地的地址。

图 18-1 展示了基本流程及重要的功能流程，从图中可以看出，认证实际发生在第三方网站上。好好享受这个简明的图吧——过一会儿事情会变得更加复杂。

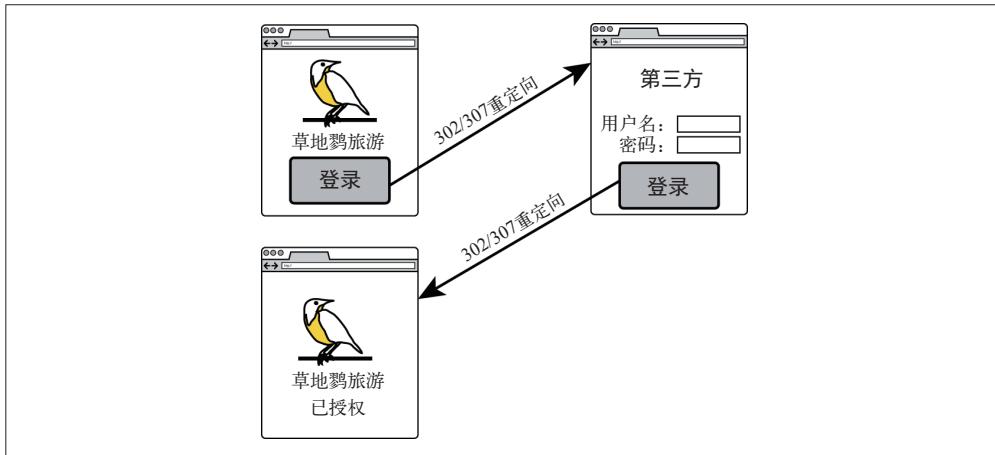


图 18-1：第三方认证流程

在使用 Passport 时，你的应用需要负责 4 个步骤。图 18-2 是第三方认证流程的更详细的视图，让我们来研究一下。

简明起见，我们使用草地鹨旅游来代表你的应用，使用 Facebook 来代表所使用的第三方认证机制。图 18-2 阐明了用户是怎样从登录页来到受控的账户信息页的（账户信息页只是出于阐述的目的，它可以是网站上任何要求认证的页面）。

图 18-2 显示了通常你不会考虑的细节，但在这个上下文中，理解这些细节很重要。特别是要明白，当你访问一个 URL 时，其实不是在向服务器发出请求，实际上请求是浏览器发出的。也就是说，浏览器可以做 3 件事情：发出 HTTP 请求、显示响应和执行重定向（重定向本质上就是发出另一个请求和显示另一个响应……而这个响应，可能又是一个重定向）。

注 1：不过，第三方也不太可能存储密码。数据库存储的是一个叫加盐散列（salted hash）的东西，密码校验用的就是它。这个加盐散列是对密码的一个单向变换。也就是说，在密码生成散列之后，就不能再把这个散列恢复成密码了。在生成散列时“加盐”，对某些类型的攻击提供了额外的保护。

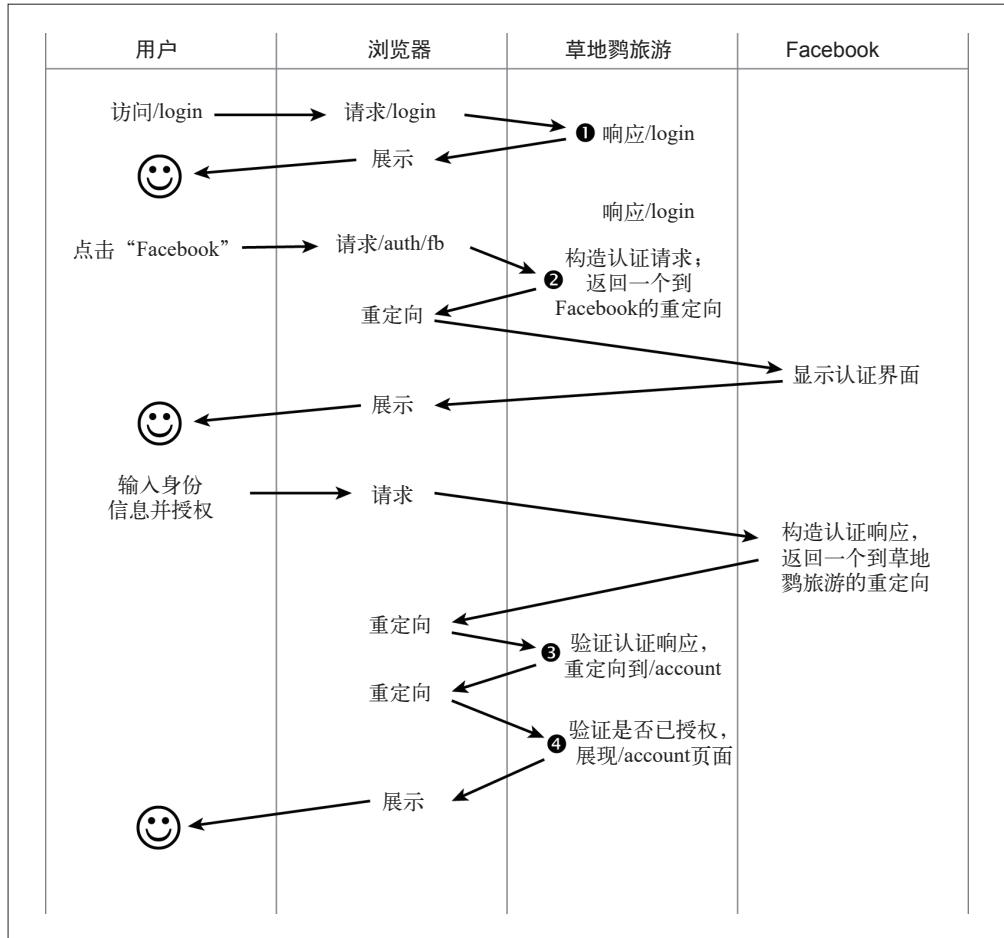


图 18-2：第三方认证流程的详细视图

在 Meadowlark 一栏，可以看到应用实际要负责的 4 个步骤。幸运的是，我们可以利用 Passport（以及各种认证策略插件）来完成这些步骤的细节，否则，本书的篇幅就要长出很多了。

在深入实现细节之前，先仔细考虑一下上面的每一个步骤。

登录页

登录页是用户可以选择登录方法的地方。如果在使用第三方认证，登录页通常就只有一个按钮或链接。如果使用了本地认证，登录页还会包含用户名和密码输入框。如果用户试图访问一个要求认证的 URL（比如示例中的 /account）又还未登录，那么这个页面就很可能是你希望的重定向目标（另一种做法是，可以重定向到“未授权”页，在里面包含到登录页的链接）。

构造认证请求

在这一步，你要构造一个发送到第三方的请求（通过一个重定向发送）。这个请求的细节比较复杂，并且跟所使用的认证策略相关。Passport（以及所使用的认证策略插件）会帮你把所有的“重活儿”都干了。认证请求会包含某种保护机制，可以防止“中间人”以及其他途径的攻击。通常认证请求的有效时间很短，因此不能把它存储起来以备后用。而且，这使得攻击者做出响应的时间窗口也很小，从而有助于防止攻击。在认证请求中，可以从第三方认证机制请求额外的信息。例如，请求用户名，可能还有电子邮件地址，都是很常见的。不过要记住，你向用户请求的信息越多，他们愿意授权应用的可能性就越小。

验证认证响应

假设用户授权了你的应用，你就会从第三方得到一个合法的响应，这是用户身份的证据。跟构造认证请求一样，验证响应的细节很复杂，而且也是由 Passport（以及所使用的认证策略插件）来为你处理。如果认证响应指示用户尚未授权（或许是用户输入了不合法的身份凭证，或许是用户并没有给你的应用授权），就要重定向到合适的页面（要么回到登录页，要么转到“未授权”页或“授权失败”页）。认证响应中应包含用户的 ID 以及步骤 2 中请求的所有额外信息，这个 ID 在特定的第三方中是唯一的。为了完成步骤 4，必须“记住”用户已经得到授权了。通常是设置一个 `Session` 变量，变量值包含用户的 ID，指示这个 `Session` 已经授权过了（也可以使用 `Cookie`，但我推荐使用 `Session`）。

验证授权

在步骤 3 中，我们把用户 ID 存储到了 `Session` 里。有了这个用户 ID，就可以从数据库获取一个用户对象，这个对象里面包含着对用户的授权信息。这样就不必每个请求都通过第三方做用户认证（这会导致速度非常慢，用户体验非常差）。这个工作很简单，不需要 Passport 的参与：我们有自己的用户对象，对象中包含的是自己的认证规则。（如果没有得到用户对象，那么说明这个请求尚未授权，可以重定向到登录页或者“未授权”页。）



使用 Passport 做认证是一项不小的工作，在本章后面就会看到。不过，认证是应用中的一个重要部分，多投入一些时间把它做好，我觉得是很明智的。现在有一些项目，比如 LockIt，正在尝试提供某种更完备的方案。另一个越来越流行的选择是 Auth0，它非常健壮，但不像 LockIt 那么好搭建。然而，要想最高效地使用 LockIt 或 Auth0（或类似的解决方案），理解认证和授权的细节很有必要，而这正是本章的目的。而且，如果哪天需要定制一个认证方案，Passport 就是一个非常好的起点。

搭建 Passport

简明起见，先从单个的认证服务提供者开始。不妨选择 Facebook。在能够设置 Passport 和 Facebook 认证策略之前，需要先在 Facebook 里做一些配置。对于 Facebook 认证，需要一个 Facebook 应用。如果已经有了合适的 Facebook 应用，则可以直接使用，也可以专门创建一个用于认证。如果可能，应该使用所在组织的官方 Facebook 账号来创建这个应用。也就是说，如果你在为草地鹨旅游工作，就要用草地鹨旅游的 Facebook 账号来创建这个应用（为了便于管理，你可以随时以管理员身份添加个人的 Facebook 账号）。如果是用于测试，那么使用你自己的 Facebook 账号是没问题的，但把个人账号用于生产是不专业的表现，而且会导致用户的不信任。

Facebook 应用的管理细节似乎经常在变，因此我不打算在这里详细解释。如果需要了解创建和管理应用的细节，请参考 Facebook 的开发者文档。

对开发和测试来说，你需要给应用关联上开发 / 测试的域名。Facebook 允许使用 localhost（以及端口号），这非常适合于测试。除了 localhost，也可以使用一个本地网络的 IP 地址，如果你的测试使用的是虚拟服务器或本地网络的另一台服务器，这就很有用了。在测试这个应用（例如，<http://localhost:3000>）时，输入浏览器的 URL 是跟这个 Facebook 应用关联起来的，这点才是关键。目前，只可以给 Facebook 应用关联一个域名，如果需要使用多个域名，就必须创建多个应用（例如，可以有草地鹨开发、草地鹨测试和草地鹨准生产的应用，而生产应用就直接叫草地鹨旅游）。

配置好 Facebook 应用之后，需要使用它的 ID 和密钥，两者都可以在这个应用的管理页上看到。



你很可能会遇到一个大难题，就是收到一条 Facebook 的消息说“根据应用配置，给定的 URL 是不允许的。”这表明，回调 URL 中的主机名和端口跟在应用中配置的不匹配。如果看一下浏览器上的 URL，就会看到经过编码的 URL，从中应该能得到线索。例如，如果我使用的是 192.168.0.103:3443，并且收到了那条消息，就会查看一下 URL。如果在查询串中看到 `redirect_uri=https%3A%2F%2F192.68.0.103%3A3443%2Fauth%2Ffacebook%2Fcallback`，我就可以迅速地发现失误：在主机名中用了 68，而不是 168。

现在来安装 Passport 和 Facebook 认证策略：

```
npm install passport passport-facebook
```

到做完时，项目中会有大量的认证代码（尤其是如果支持多个认证策略的话）。我们不想这些代码把 `meadowlark.js` 弄得一团糟，因此将创建一个名为 `lib/auth.js` 的模块。这个文件将变得很大，所以我们准备各个击破（最终示例请看配套版本库的 `ch18` 目录）。先从 Passport 所需要的模块导入和两个方法（`serializeUser` 和 `deserializeUser`）开始：

```
const passport = require('passport')
const FacebookStrategy = require('passport-facebook').Strategy

const db = require('../db')

passport.serializeUser((user, done) => done(null, user._id))

passport.deserializeUser((id, done) => {
  db.getUserById(id)
    .then(user => done(null, user))
    .catch(err => done(err, null))
})
```

Passport 使用 `serializeUser` 和 `deserializeUser` 把请求映射成已认证的用户，并允许使用任何想用的存储方法。在示例中，我们只打算把数据库里的 ID（`_id` 属性）存储到 Session 里。这样使用 ID，使得 `serialize`（序列化 / 串行化）和 `deserialize`（反序列化）有点儿名不副实了，但实际上只是把用户 ID 存储到了 Session 里。以后需要的时候，可以从数据库里直接查找这个 ID，并得到用户对象。

实现了这两个方法之后，只要有活跃的 Session 且用户已经成功认证，`req.session.passport.user` 就是从数据库获取到的那个用户对象。

接下来将确定这个文件要导出什么。为了启用 Passport 的功能，我们需要做截然不同的两件事：初始化 Passport 和注册路由。注册的路由会处理认证的请求及来自第三方认证服务的重定向回调。我们并不想把这两者组合成一个函数，因为在主应用文件里，可以选择何时把 Passport 链入中间件链（记住，增加中间件时，顺序很重要）。因此，我们不准备让模块导出只做某件事的函数，而是让它导出返回一个对象的函数，而这个对象包含着我们需要的方法。为什么不让模块直接返回一个对象呢？因为需要加入一些配置值。而且，由于需要把 Passport 中间件链入应用，因此使用一个函数传入 Express 应用对象会更容易：

```
module.exports = (app, options) => {
  // 如果成功和失败的重定向没有指定，就设置合理的默认值
  if(!options.successRedirect) options.successRedirect = '/account'
  if(!options.failureRedirect) options.failureRedirect = '/login'
  return {
    init: function() { /* TODO */ },
    registerRoutes: function() { /* TODO */ },
  }
}
```

在进入 `init` 和 `registerRoutes` 方法的细节之前，先来看看要怎样使用这个模块（我希望通过了解如何使用，能有助于对这个模块导出的理解）：

```
const createAuth = require('./lib/auth')

// 应用的其他配置
```

```

const auth = createAuth(app, {
  // baseUrl是可选的；如果省略，默认就是localhost。
  // 如果不是在本地机器上运行应用，就有必要设置它。
  // 例如，如果你在使用准生产服务器，可能就会把BASE_URL
  // 环境变量设置为 https://staging.meadowlark.com
  baseUrl: process.env.BASE_URL,
  providers: credentials.authProviders,
  successRedirect: '/account',
  failureRedirect: '/unauthorized',
})
// auth.init()把Passport中间件链进来了
auth.init()

// 现在可以指定认证路由了
auth.registerRoutes()

```

请注意，除了指定成功和失败的重定向路径，我们也指定了一个名为 `providers` 的属性，并把它提取到敏感信息文件（参见第 13 章）里了。我们需要把 `authProviders` 属性加入 `.credentials.development.json` 中：

```

"authProviders": {
  "facebook": {
    "appId": "your_app_id",
    "appSecret": "your_app_secret"
  }
}

```



像这样把认证代码都汇集到一个模块里还有一个原因，即我们可以在其他项目中重用它。事实上，现在已经有了一些认证包，所做的事情跟案例基本一样。然而，理解其中的运行细节还是很重要的，因此即使最终使用的是别人写的模块，自己这个模块也有助于理解认证流程中所发生的一切。

现在，该“关照”一下 `init` 方法了（先前在 `auth.js` 里还只是一个 TODO）：

```

init: function() {
  var config = options.providers

  // 配置Facebook认证策略
  passport.use(new FacebookStrategy({
    clientID: config.facebook.appId,
    clientSecret: config.facebook.appSecret,
    callbackURL: (options.baseUrl || '') + '/auth/facebook/callback',
  }, (accessToken, refreshToken, profile, done) => {
    const authId = 'facebook:' + profile.id
    db.getUserByAuthId(authId)
      .then(user => {
        if(user) return done(null, user)
        db.addUser({
          authId: authId,
          name: profile.displayName,
        })
      })
  }))
}

```

```

        created: new Date(),
        role: 'customer',
    })
    .then(user => done(null, user))
    .catch(err => done(err, null))
})
.catch(err => {
    if(err) return done(err, null);
})
}),
),
app.use(passport.initialize())
app.use(passport.session())
},

```

上面的代码量不少，但实际上大部分只是 Passport 的样板代码。我们向 `FacebookStrategy` 实例传入了一个函数，代码中重要的部分就在这个函数里。当这个函数被调用时（在用户成功认证之后），`profile` 参数就包含了这个 Facebook 用户的信息。其中最重要的是，它包含着一个 Facebook ID：我们要用这个 ID 来关联一个 Facebook 账号与自己的用户对象。注意我们给 `authId` 属性加上了一个前缀 `facebook:` 作为它的命名空间。这样就避免了 Facebook 的 ID 跟 Twitter 或谷歌的 ID 发生冲突的可能，尽管这种可能性非常小（而且，可以通过检查用户 Model 得知用户使用的是哪种认证方法，这也许很有用）。对这个带命名空间的 ID，如果数据库中已经包含一条记录，只要返回就可以了（这时正是 `serializeUser` 被调用的时候，`serializeUser` 会把我们自己的用户 ID 放入 Session 里）。如果没有用户记录可以返回，就创建一个新的用户对象，然后保存进数据库里。

我们要做的最后一件事是创建 `registerRoutes` 方法（别担心，这个要短小得多）：

```

registerRoutes: () => {
    app.get('/auth/facebook', (req, res, next) => {
        if(req.query.redirect) req.session.authRedirect = req.query.redirect
        passport.authenticate('facebook')(req, res, next)
    })
    app.get('/auth/facebook/callback', passport.authenticate('facebook',
        { failureRedirect: options.failureRedirect }),
        (req, res) => {
            // 只有认证成功之后才会到达这里
            const redirect = req.session.authRedirect
            if(redirect) delete req.session.authRedirect
            res.redirect(303, redirect || options.successRedirect)
        }
    )
},

```

现在有了路径 `/auth/facebook`，访问这个路径将自动为访问者重定向到 Facebook 的认证界面（这是由 `passport.authenticate('facebook')` 来完成的），即图 18-1 的步骤 2。注意，先检查查询串中是否有一个参数 `redirect`，如果有，就把它保存进 Session 里。这样才能在完成认证后，自动重定向到想要的目标。一旦用户通过 Facebook 授权之后，浏览器将重

定向回你的网站——具体来说，就是重定向到 /auth/facebook/callback（可能会带着可选的查询串参数 `redirect`，指示用户是从哪个路径来的）。

查询串中还会有令牌的相关参数，Passport 会对其进行验证。如果验证失败，Passport 会让浏览器重定向到 `options.failureRedirect`。如果验证成功，Passport 会调用 `next`，从而回到应用。要注意在 /auth/facebook/callback 的路由处理函数中，中间件是如何串接的：`passport.authenticate` 会先得到调用。如果 `passport.authenticate` 调用了 `next`，控制将交给你的函数²，这个函数要么会重定向到最初访问的路径，要么会重定向到 `options.successRedirect`（如果查询串参数 `redirect` 没有指定的话）。



如果省略查询串参数 `redirect`，认证路由就可以得到简化。如果只有一个 URL 需要认证，你可能就想这么做了。然而，保留查询串参数 `redirect` 最终还是会带来便利的，而且会提供更好的用户体验。你肯定有过这样的经历：找到了自己想要的页面，然后被引导到登录页面，完成登录却被重定向到一个默认页面，还得导航回到你想要的那个页面。这样的用户体验是不尽如人意的。

在这个过程中，Passport 使用的“魔法”是把用户信息（在我们的示例中，用户信息只是数据库中的一个用户 ID）保存到 Session 中。这是好事，因为浏览器在做重定向，这意味着一个不同的 HTTP 请求：如果 Session 中没有用户信息，就无法知道这个用户已经认证过了。只要用户认证成功，`req.session.passport.user` 就会被设置，因而后续的请求就可以知道这个用户已经认证过了。

来看一下 /account 路由处理函数，看它如何确认用户已经被认证（这个路由处理函数会在主应用文件或一个独立的路由模块里，而不是在 /lib/auth.js 里）：

```
app.get('/account', (req, res) => {
  if(!req.user)
    return res.redirect(303, '/unauthorized')
  res.render('account', { username: req.user.name })
})

// 还需要一个“未授权”页
app.get('/unauthorized', (req, res) => {
  res.status(403).render('unauthorized')
})

// 需要一个退出登录的方式
app.get('/logout', (req, res) => {
  req.logout()
  res.redirect('/')
})
```

现在，只有已认证的用户可以看到账号信息页（/account），其他的用户都会被重定向到“未授权”页。

注 2：即上面代码中的 `(req, res) => { ... }`。——译者注

18.3.7 基于角色的授权

目前为止，从技术上来说，我们并没有做任何授权（只是区分了已授权和未授权的用户）。可是，如果只想让客户看到他们自己账号的视图该怎么办？（雇员可以有一个完全不同的视图，在其中他们可以看到所有用户的账号信息。）

还记得吗？在单个路由里，可以有多个函数，它们会按顺序得到调用。创建一个名为 `customerOnly` 的函数，只允许客户通过：

```
const customerOnly = (req, res, next) => {
  if(req.user && req.user.role === 'customer') return next()
  // 让只允许客户访问的页面知道自己是需要登录的
  res.redirect(303, '/unauthorized')
}
```

再创建一个名为 `employeeOnly` 的函数，它跟上面略有不同。比如说，我们希望只有雇员才可以访问路径 `/sales`，而且甚至不想让非雇员知道这个页面的存在，即便他们是误打误撞才访问的。如果某个潜在的攻击者进入了 `/sales`，看到一个“未授权”页，这也会作为给他提供的信息，并可能会使他的攻击更容易一些（知道了这个页面的存在）。因此，为了增加一点儿安全性，希望非雇员在访问 `/sales` 页时看到的是一个常规的 404 页，从而不给潜在的攻击者提供任何头绪：

```
const employeeOnly = (req, res, next) => {
  if(req.user && req.user.role === 'employee') return next()
  // 我们希望把“雇员专属页”的授权失败信息“隐藏”掉,
  // 甚至都不让潜在的黑客知道这些页面的存在
  next('route')
}
```

调用 `next('route')` 并非简单地执行路由的下一个处理函数，而是完全跳过了这个路由。如果后续不再有处理 `/account` 的路由，最终它就会到达 404 路由处理函数那里，而这正是我们想要的结果。

下面把前面两个函数用起来，看看有多简单：

```
// 客户的路由

app.get('/account', customerOnly, (req, res) => {
  res.render('account', { username: req.user.name })
})
app.get('/account/order-history', customerOnly, (req, res) => {
  res.render('account/order-history')
})
app.get('/account/email-prefs', customerOnly, (req, res) => {
  res.render('account/email-prefs')
})

// 雇员的路由
```

```
app.get('/sales', employeeOnly, (req, res) => {
    res.render('sales')
})
```

很明显，你想把基于角色的授权做得多么简单或多么复杂都可以。例如，如果你想授权多个角色，应该怎么办？可以使用下面的函数和路由：

```
const allow = roles => (req, res, next) => {
    if(req.user && roles.split(',').includes(req.user.role)) return next()
    res.redirect(303, '/unauthorized')
}
```

对于基于角色的授权，你可以充分发挥创造性，希望这个例子能提供一个思路。还可以基于其他的属性授权，比如说用户成为会员的时长，或者用户在你这里已经订购了多少度假产品。

18.3.8 增加认证提供者

既然框架已经搭建好，增加更多的认证提供者就会变得很容易。不妨说我们希望能使用谷歌来认证。在开始增加新代码之前，需要先在谷歌账号上设置一个项目。

进入谷歌开发者控制台，从导航栏上选择一个项目（如果你还没有项目，点击“New Project”并遵照指示操作）。选择一个项目之后，点击“Enable APIs and Services”（启用 API 和服务），并启用 Cloud Identity API。点击“Credentials”（凭据），然后点击“Create Credentials”（创建凭据），选择“OAuth client ID”，并选择“Web application”。然后为应用输入合适的 URL：对于测试来说，可以使用 `http://localhost:3000` 作为授权入口 URL，并使用 `http://localhost:3000/auth/google/callback` 作为授权后重定向的 URI。

在谷歌把一切都设置好了之后，可以运行 `npm install passport-google-oauth20`，并在 `lib/auth.js` 里增加以下代码：

```
// 配置谷歌认证策略
passport.use(new GoogleStrategy({
    clientID: config.google.clientID,
    clientSecret: config.google.clientSecret,
    callbackURL: (options.baseUrl || '') + '/auth/google/callback',
}, (token, tokenSecret, profile, done) => {
    const authId = 'google:' + profile.id
    db.getUserByAuthId(authId)
        .then(user => {
            if(user) return done(null, user)
            db.addUser({
                authId: authId,
                name: profile.displayName,
                created: new Date(),
                role: 'customer',
            })
        })
}))
```

```
        })
        .then(user => done(null, user))
        .catch(err => done(err, null))
    })
    .catch(err => {
        console.log('whoops, there was an error: ', err.message)
        if(err) return done(err, null);
    })
}))
```

并在 `registerRoutes` 方法里增加以下代码：

```
app.get('/auth/google', (req, res, next) => {
    if(req.query.redirect) req.session.authRedirect = req.query.redirect
    passport.authenticate('google', { scope: ['profile'] })(req, res, next)
})

app.get('/auth/google/callback', passport.authenticate('google',
    { failureRedirect: options.failureRedirect }),
    (req, res) => {
        // 只有认证成功之后才会到达这里
        const redirect = req.session.authRedirect
        if(redirect) delete req.session.authRedirect
        res.redirect(303, req.query.redirect || options.successRedirect)
    }
})
```

18.4 小结

祝贺你完成了本书最错综复杂的一章！如此重要的特性（认证与授权）竟是这样复杂，真是令人遗憾。然而，在一个安全威胁无处不在的世界里，这是不可避免的。幸运的是，像 Passport 这样的项目（以及基于它的各种优秀的认证方案）在某种程度上减轻了我们的负担。无论如何，我鼓励你多关注一下应用中的安全方面。在安全方面勤勤恳恳，会让你成为更好的互联网公民。如果做得好，或许用户从不会感谢你，但是，如果由于做得很差而导致用户数据泄露，那应用的“主人”就有麻烦了。

第19章

集成第三方API

现在越来越多的成功网站不再完全封闭了。为了留住已有的用户以及寻找新的用户，跟社交网络进行集成就成了必然的选择。而为了提供店铺定位或其他需要定位的服务，位置搜索和地图服务也就必不可少。不止如此，越来越多的组织已经认识到，开放一个 API 有助于服务的拓展以及让服务更加有效。

本章将讨论两个最常见的集成需求：社交媒体和地理定位。

19.1 社交媒体

社交媒体是你的产品或服务的非常好的推广途径。如果推广正是你的目标，那么让用户能够轻而易举地在社交媒体网站上分享你的内容是非常必要的。在本书写作之时，主流的社交网络服务有 Facebook、Twitter、Instagram 和 YouTube。像 Pinterest 和 Flickr 这样的网站也有一席之地，但通常更适合于特定的用户群体（例如，如果你的网站是关于手工制作的，那么绝对会希望支持 Pinterest）。也许会被你笑话，但我认为 MySpace 总有一天会王者归来。它的网站重新设计了，很有创意，而且值得注意的是，MySpace 是基于 Node 构建的。

19.1.1 社交媒体插件与网站性能

大多数社交媒体的集成只涉及前端。在页面里引用合适的 JavaScript 文件，就同时启用了内容的输入（例如，Facebook 页的前 3 个故事）和输出（例如，针对当前页面发一条推特）。单独引用 JavaScript 文件来集成社交媒体尽管很多时候是最简单的做法，但也有代

价。我见过页面的加载时间因为额外的 HTTP 请求而变成了原来的 2 倍甚至 3 倍的情况。如果页面性能对你来说很重要（应该是这样的，尤其是从移动端用户的角度考虑），就需要仔细考虑应该如何集成社交媒体。

虽然这么说，但是实现了 Facebook 的“喜欢”按钮或“发推”按钮的代码还是需要利用浏览器内的 Cookie 并以用户的名义提交。要把这个功能移到后端，可能会很困难（而且，在有些情形中是不可能的）。因此，如果你需要的是“喜欢”按钮或“发推”按钮那样的功能，那么在页面中引用合适的第三方库是你最好的选择，即使它会影响页面的性能。

19.1.2 搜索推文

如果希望展现最近 10 条包含主题标签“#Oregon #travel”的推文，的确可以使用一个前端组件来实现，但这样做会产生额外的 HTTP 请求。而如果在后端实现，则可以为了满足性能的需要而缓存这些推文。而且，如果搜索是在后端进行的，就可以把一些很刻薄的推文“拉黑”，但在前端实现就会相对困难一些。

跟 Facebook 一样，Twitter 允许你创建应用。说是应用，其实有些不恰当，因为一个 Twitter 应用并不做任何事情（在传统意义上）。它更像是在网站上可以用来创建实际应用的“凭证”。要访问 Twitter API，最容易且最可移植的方式是创建一个应用，再通过这个应用来获取访问令牌。

要想创建 Twitter 应用，首先需要进入它的开发者中心。先确认登录，然后点击导航栏中你的用户名，再点击 Apps。接着点击“Create an app”，并遵照说明操作。创建了一个 Twitter 应用之后，你就能看到现在你有了一个消费者 API key 和一个 API 密钥。正如其名，这个 API 密钥是不可公开的，绝对不能把它包含在响应中发送到客户端。如果某个第三方能访问到这个密钥，他们就能以你的应用的名义发出请求，如果是出于恶意，就会对你造成不良的后果。

现在有了消费者 API key 和密钥，就可以跟 Twitter 的 REST API 通信了。

为了让代码整齐一些，把有关 Twitter 的代码放入一个名为 lib/twitter.js 的模块：

```
const https = require('https')

module.exports = twitterOptions => {

  return {
    search: async (query, count) => {
      // TODO
    }
  }
}
```

你应该越来越熟悉上面代码的结构了。我们的模块导出了一个函数，调用者会把一个配置对象传给这个函数。这个函数返回的是一个对象，而对象中包含着若干方法。按这种方式，可以往这个模块中增加功能。目前而言，只提供一个 `search` 方法，但将来会这样来使用这个库：

```
const twitter = require('./lib/twitter')({
  consumerApiKey: credentials.twitter.consumerApiKey,
  apiSecretKey: credentials.twitter.apiSecretKey,
})

const tweets = await twitter.search('#Oregon #travel', 10)
// 推文会在tweets.statuses里
```

[别忘了把 `twitter` 属性（里面包含 `consumerApiKey` 和 `apiSecretKey`）加入 `.credentials.development.json` 文件中。]

在实现 `search` 方法之前，必须增加一些功能，即向 Twitter 证明身份。这个过程很简单：使用 HTTPS，拿着消费者 Key 和消费者密钥去向 Twitter API 请求一个访问令牌。我们只需要做一次，因为目前而言，Twitter 并不会让访问令牌过期（虽然可以手动让它失效）。既然不希望每次都请求一个访问令牌，就把它缓存起来，以便重用。

前面组织模块（`lib/twitter.js`）的方式，允许创建一些模块私有的功能，而不让调用者访问到。具体来说，调用者唯一能访问到的就是 `module.exports`。既然返回的是一个函数，调用者能访问到的就只有这个函数。调用这个函数会得到一个对象，而调用者只能使用这个对象的属性。既然如此，我们就准备定义一个变量 `accessToken`，用来缓存访问令牌，并创建一个函数 `getAccessToken`，用来获取访问令牌。第一次调用 `getAccessToken` 时，它会发出一个 Twitter API 请求来获取访问令牌，而后续的调用会直接返回 `accessToken` 的值：

```
const https = require('https')

module.exports = function(twitterOptions) {
  // 在模块外，这个变量是不可见的
  let accessToken = null

  // 在模块外，这个函数是不可见的
  const getAccessToken = async () => {
    if(accessToken) return accessToken
    // TODO: 获取access token
  }

  return {
    search: async (query, count) => {
      // TODO
    }
  }
}
```

把 `getAccessToken` 标记为异步是因为可能需要向 Twitter API 发出一个 HTTP 请求（如果还不存在缓存的 token 的话）。我们已经建立起了基本的代码框架，现在就来实现 `getAccessToken`：

```
const getAccessToken = async () => {
  if (accessToken) return accessToken

  const bearerToken = Buffer(
    encodeURIComponent(twitterOptions.consumerApiKey) + ':' +
    encodeURIComponent(twitterOptions.apiSecretKey)
  ).toString('base64')

  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'POST',
    path: '/oauth2/token?grant_type=client_credentials',
    headers: {
      'Authorization': 'Basic ' + bearerToken,
    },
  }

  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data =
        res.on('data', chunk => data += chunk)
        res.on('end', () => {
          const auth = JSON.parse(data)
          if (auth.token_type !== 'bearer')
            return reject(new Error('Twitter auth failed.'))
          accessToken = auth.access_token
          return resolve(accessToken)
        })
      }).end()
    )
  )
}
```

构造这个请求的细节可以从 Twitter 的开发者文档的应用认证部分找到。基本上，必须构建一个“持有人”令牌，它是消费者 key 和消费者密钥的组合，并以 Base64 格式编码。构造出这个令牌之后，可以把它放在 `Authorization` 请求头里，调用 `/oauth2/token` 这个 API，请求一个访问令牌。要注意，必须使用 HTTPS：如果你试图通过 HTTP 发出这个请求，就相当于未把私钥加密就在网络上传输了，而且 API 也会直接把连接中断。

从 API 获得完整的响应之后（我们监听响应流的 `end` 事件），可以解析响应 JSON，确保令牌的类型为 `bearer`。缓存得到的访问令牌，然后激活 `Promise` 的回调。

现在已经有了获取 access token 的机制，可以发出 API 调用了。那么，就让我们来实现 `search` 方法：

```
search: async (query, count) => {
  const accessToken = await getAccessToken()
  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/search/tweets.json?q=' +
      encodeURIComponent(query) +
      '&count=' + (count || 10),
    headers: {
      'Authorization': 'Bearer ' + accessToken,
    },
  }
  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  ),
},
```

19.1.3 展现推文

现在已经能够搜索推文了，那么，该如何在网站上显示出来呢？很大程度上由你决定，但还是有几点需要考虑。在数据的使用上，Twitter 总是想确保人们能跟其本身的品牌保持一致。因此，它提出了一个“推文展现要求”，其中包括一些功能性元素的使用，并要求在展现推文时必须包含这些元素。

尽管这些规定并不是毫无回旋余地（比如说，在一个不支持图片的设备上展现推文，就不必包含头像图片），但很多时候你最终做出来的结果会跟直接把推文嵌入进来非常相像。展现推文要做的工作不少。尽管也有一条捷径，但是需要引入 Twitter 的组件库，这会导致我们试图避免的额外的 HTTP 请求。

如果需要显示推文，最好的做法就是使用 Twitter 的组件库，即使它会引起额外的 HTTP 请求。对组件库的更高级的使用，你仍然需要从后端访问 Twitter 的 REST API，因此到最后，很可能会跟前端脚本相配合使用 REST API。

继续示例：我们想要显示最近的 10 条提到主题标签“#Oregon #travel”的推文。使用 REST API 来搜索这些推文，并使用 Twitter 的组件库来显示它们。由于不想超出使用次数限制（或拖慢服务器），因此我们会缓存这些推文和相应的 HTML，在 15 分钟内不会更新。

先修改一下我们的 Twitter 库，加入一个方法 `embed`，用来生成展现一条推文的 HTML。注意我们准备使用一个 npm 库 `querystringify` 从一个对象构造一个查询串，所以别忘了运行 `npm install querystringify`，并把它导入进来（`const qs = require('querystringify')`）。然后，把以下函数加入 `lib/twitter.js` 的导出里：

```

embed: async (url, options = {}) => {
  options.url = url
  const accessToken = await getAccessToken()
  const requestOptions = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/statuses/oembed.json?' + qs.stringify(options),
    headers: [
      'Authorization': 'Bearer ' + accessToken,
    ],
  }
  return new Promise((resolve, reject) =>
    https.request(requestOptions, res => {
      let data =
        res.on('data', chunk => data += chunk)
        res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  ),
},

```

现在可以搜索、缓存推文了。在主应用文件里，创建函数 `getTopTweets`：

```

const twitterClient = createTwitterClient(credentials.twitter)

const getTopTweets = ((twitterClient, search) => {
  const topTweets = {
    count: 10,
    lastRefreshed: 0,
    refreshInterval: 15 * 60 * 1000,
    tweets: [],
  }
  return async () => {
    if(Date.now() > topTweets.lastRefreshed + topTweets.refreshInterval) {
      const tweets =
        await twitterClient.search('#Oregon #travel', topTweets.count)
      const formattedTweets = await Promise.all(
        tweets.statuses.map(async ({ id_str, user }) => {
          const url = `https://exl.ptpress.cn:8442/ex/l/6ee39484/${user.id_str}
          /statuses/${id_str}`
          const embeddedTweet = await twitterClient.embed(url, { omit_script: 1 })
          return embeddedTweet.html
        })
      )
      topTweets.lastRefreshed = Date.now()
      topTweets.tweets = formattedTweets
    }
    return topTweets.tweets
  }
})(twitterClient, '#Oregon #travel')

```

本质上，`getTopTweets` 函数不仅搜索了指定主题标签的推文，还会在合理的时间长度中缓存这些推文。注意我们创建了一个立即执行的函数表达式（IIFE），这是因为想让 `topTweets` 在闭包里安全地缓存，不受干扰。从这个 IIFE 返回的异步函数如果有必要就会

刷新缓存，然后返回缓存的内容。

最后，创建一个视图 `views/social.handlebars` 作为社交媒体的主视图（目前，视图只包含所选择的推文）：

```
<h2>Oregon Travel in Social Media</h2>

<script id="twitter-wjs" type="text/javascript"
  async defer src="//platform.twitter.com/widgets.js"></script>

{{{tweets}}}
```

并增加一个路由，渲染这个视图：

```
app.get('/social', async (req, res) => {
  res.render('social', { tweets: await getTopTweets() })
})
```

注意我们引用了一个外部的脚本，即 Twitter 的 `widgets.js`。对于嵌入到页面里的推文，要用这个脚本来给它们增加格式和功能。默认情况下，`oembed` API 会在 HTML 里包含对这个脚本的引用，但由于是显示 10 条推文，如果按默认情况，就会额外多引用这个脚本 9 次。所以，回想一下，我们在调用 `oembed` API 的时候，传入了选项 `{ omit_script: 1 }`，并让它忽略了脚本的引用。既然如此，就必须在某个地方提供这个脚本，正如在视图中所做的那样。可以试着从视图中移除这个脚本，你将仍然可以看到推文，但它们已没有任何格式或功能了。

现在我们有了一个不错的社交媒体内容聚合。接下来把注意力转到社交媒体的另一个重要应用上来：在应用中展现地图。

19.2 地理编码

地理编码指的是将一个街道地址或地点名（英格兰米尔顿·凯恩斯 MK3-6EB 布莱奇利镇舍伍德大道布莱奇利公园）转换成地理坐标（纬度 51.997 659 7，经度 -0.740 686 3）的过程。如果你的应用准备做任何有关地理的计算（距离或方位），或展现一个地图，就需要地理坐标。



你可能更习惯地理坐标以度、分、秒（DMS）的形式来指定，不过地理编码 API 以及地图服务是使用浮点数来表示经纬度的。如果需要显示度分秒形式的坐标，请查看维基百科的“Geographic coordinate conversion”条目。

19.2.1 使用谷歌生成地理编码

对于地理编码，谷歌和 Bing 都提供了非常好的 REST 服务。我们的示例将使用谷歌的服务，但 Bing 的服务也是很相似的。

如果不给谷歌账号关联一个结算账号，你的地理编码请求会被限制为一天一次，这样就会导致测试周期非常缓慢。本书会尽量避免推荐免费使用量都不能满足开发阶段的服务。我也尝试了一些免费的地理编码服务，但是发现在使用上有相当大的差距，因此将继续推荐谷歌的服务。不过，在写作本书时，对于开发级别的使用量，使用谷歌的地理编码是免费的。你的账号每个月会收到 200 美元的赠金，需要发出 40 000 个请求才能用完。如果你想照着本章做下去，就进入谷歌 API 控制台，从主菜单选择“结算”，然后输入支付信息。

设置完结算账号，你需要一个 API key 以用于谷歌的地理编码 API。进入谷歌 API 控制台，从导航栏中选择你的项目，然后点击 APIs。如果地理编码 API 不在已启用 API 的列表里，就在其他 API 的列表里找到它并将其加进来。大多数谷歌 API 共用同一套 API 凭据，所以点击左上角的导航菜单，回到你的仪表板页。点击“凭据”（Credentials），如果还没有合适的 API key，就创建一个新的。要注意，API key 可能会被限制以避免滥用，因此要确认你的 API key 是否能够在应用里使用。如果你需要一个 API key 用于开发，则可以用 IP 地址来限制这个 key，并选择你的 IP 地址（如果不知道自己的 IP 地址，可以直接在谷歌中输入“What's my IP address?”）。

有了 API key 之后，把它加入 .credentials.development.json 中：

```
"google": {  
    "apiKey": "<你的 API Key>"  
}
```

然后创建模块 lib/geocode.js：

```
const https = require('https')  
const { credentials } = require('../config')  
  
module.exports = async query => {  
  
    const options = {  
        path: '/maps/api/geocode/json?address=' +  
            encodeURIComponent(query) + '&key=' +  
            credentials.google.apiKey,  
    }  
  
    return new Promise((resolve, reject) =>  
        https.request(options, res => {  
            let data = ''  
            res.on('data', chunk => data += chunk)  
            res.on('end', () => {  
                data = JSON.parse(data)  
                if(!data.results.length)  
                    return reject(new Error(`no results for "${query}"`))  
                resolve(data.results[0].geometry.location)  
            })  
        }).end()  
    )  
}
```

现在我们就有了一个函数，它可以连接到谷歌 API 并为地址生成地理编码。如果 API 未能找到某个地址（或因为任何原因失败了），就会返回一个错误。API 也有可能返回多个地址。例如，如果你搜索“10 Main Street”而没有指定一个城市、州或邮编，就会返回几十条结果。我们的实现只是简单地取出第一条。这个 API 可以返回大量的信息，但目前我们只对地理坐标感兴趣。你可以很容易地修改接口，让它返回更多的信息。如果想了解这个 API 所返回数据的更多信息，请查阅谷歌地理编码的 API 文档。

使用限制

目前，谷歌地理编码 API 有一个每月使用的限制，但你可以为每个额外编码请求支付 0.005 美元。因此如果你在某一个月里发出了 1 000 000 个请求，那么就会从谷歌收到一张 5000 美元的账单……所以对你来说很可能会有一个现实的限制。



如果你忘记了停止服务，或某个坏家伙得到了你的登录凭据，就可能会导致费用失控。如果担心这样的事情发生，可以设置一个预算以及几项警告，当达到警告条件时会发出通知。进入你的谷歌开发者控制台，从结算菜单里选择“预算与警告”。

写作本书时，为了避免滥用，谷歌限制在 100 秒内只能发出 5000 个请求，这个限制应该是很难超过的。谷歌的 API 还要求，如果你的网站需要使用地图，就要使用谷歌地图。也就是说，如果使用了谷歌的服务来为你的数据生成地理编码，就不能转而使用 Bing 地图来显示这些位置信息，这是违反服务条款的。一般而言，这不是什么过分的限制，因为如果你不打算在地图上显示位置，就不太可能会做地理编码了。不过，如果你更喜欢 Bing 的地图而不是谷歌的，或者反过来，就要留意服务条款并使用合适的 API。

19.2.2 为你的数据做地理编码

我们已经有了一个不错的俄勒冈州度假产品套餐数据库，现在想要显示一个地图，在上面使用大头针标出这些度假地点都在哪里。这时就需要做地理编码了。

在数据库中，我们已经有了度假产品数据，并且每个度假产品都有一个可以用于地理编码的位置搜索字符串，但是还没有位置的经纬度坐标。

现在的问题是何时以及如何做地理编码。大体来讲，有以下 3 个选择。

- 在把新的度假产品加入数据库时就生成地理编码。如果给系统增加一个管理界面，允许供应商动态地把度假产品加入数据库，这很可能是一个非常好的选择。不过，既然不准备增加管理界面，就得放弃这个选择。
- 从数据库取出度假产品时再做地理编码。这样的话，每次从数据库取到度假产品时，都要做一次检查，如果其中有任何缺少坐标的，就为它们生成地理编码。这个做法听起来很吸引人，而且很可能是 3 个选择中最容易的，但有几大缺陷，使得它并不是合适的选择。

择。第一个是性能。如果你往数据库里增加了 1000 个新的度假产品，那么第一个查看度假产品列表的人，就得等待所有这些地理编码请求都调用成功并把结果完全写入数据库。而且，你可以想象一下这种情形：一个负载测试套件往数据库里增加了 1000 个度假产品，然后发出了 1000 个产品列表请求。由于这些请求都是并行的，因此每个请求又会衍生出 1000 个地理编码请求（因为地理编码的结果数据还没有写入数据库）……结果就是 1 000 000 个地理编码请求以及来自谷歌的 5000 美元的账单。因此我们把这个选择排除掉。

- 写一个脚本，找出缺少坐标数据的度假产品，然后对它们做地理编码。就目前的情况而言，这是最好的方案了。出于开发的目的，我们对度假数据库将只做一次运算，况且还没有管理界面来增加新的度假产品。此外，如果后续决定增加一个管理界面，这个脚本也是有用的。事实上，在增加一条新的度假产品之后，只要运行这个脚本，事情就完成了。

首先，需要在 db.js 中增加一个方法，用以更新已有的度假产品（也会增加一个关闭数据库连接的方法，便于在脚本中使用）：

```
module.exports = {
  //...
  updateVacationBySku: async (sku, data) => Vacation.updateOne({ sku }, data),
  close: () => mongoose.connection.close(),
}
```

然后写一个脚本 db-geocode.js：

```
const db = require('./db')
const geocode = require('./lib/geocode')

const geocodeVacations = async () => {
  const vacations = await db.getVacations()
  const vacationsWithoutCoordinates = vacations.filter(({ location }) =>
    !location.coordinates || typeof location.coordinates.lat !== 'number')
  console.log(`geocoding ${vacationsWithoutCoordinates.length} ` +
    `of ${vacations.length} vacations:`)
  return Promise.all(vacationsWithoutCoordinates.map(async ({ sku, location }) => {
    const { search } = location
    if(typeof search !== 'string' || !/\w/.test(search))
      return console.log(` SKU ${sku} FAILED: does not have location.search`)
    try {
      const coordinates = await geocode(search)
      await db.updateVacationBySku(sku, { location: { search, coordinates } })
      console.log(` SKU ${sku} SUCCEEDED: ${coordinates.lat}, ${coordinates.lng}`)
    } catch (err) {
      return console.log(` SKU ${sku} FAILED: ${err.message}`)
    }
  }))
}

geocodeVacations()
.then(() => {
```

```
        console.log('DONE')
        db.close()
    })
    .catch(err => {
        console.error('ERROR: ' + err.message)
        db.close()
    })
}
```

当你运行完这个脚本（`node db-geocode.js`），应该能看到，所有的度假产品都已经成功地完成地理编码了。既然已经有了地理位置信息，接下来学习如何在地图中显示出来……

19.2.3 显示地图

在地图上显示这些度假地点虽然说可以归类为“前端”工作，但是我们已经做了这么多，如果看不到成果，无疑很让人失望。因此我们准备稍微偏离一下本书主要关注的后端，看看如何在地图上显示合作商户的位置，这些位置是刚刚做完地理编码得到的。

我们在做地理编码时已经创建了一个谷歌的 API key，现在还需要启用地图 API。进入你的谷歌控制台，点击 APIs，找到地图 JavaScript API，如果还没有启用的话就启用它。

现在创建一个视图 `views/vacations-map.handlebars`，用来显示度假产品地图。一开始只是显示这个地图，下一步再增加度假地点：

```
<div id="map" style="width: 100%; height: 60vh;"></div>
<script>
  let map = undefined
  async function initMap() {
    map = new google.maps.Map(document.getElementById('map'), {
      // 俄勒冈州大概的地理中心
      center: { lat: 44.0978126, lng: -120.0963654 },
      // 这个缩放级别可以看到这个州的大部分
      zoom: 7,
    })
  }
</script>
<script src="/maps/api/js?key={{googleApiKey}}&callback=initMap"
  async defer></script>
```

现在可以依据度假产品在地图上放置一些大头针了。在第 15 章，我们创建了一个 API 端点 `/api/vacations`，现在这个端点就会包含地理位置的数据了。我们将使用这个端点来获取度假产品，然后在地图上放置大头针。修改 `views/vacations-map.handlebars.js` 中的 `initMap` 函数：

```
async function initMap() {
  map = new google.maps.Map(document.getElementById('map'), {
    // 俄勒冈州大概的地理中心
    center: { lat: 44.0978126, lng: -120.0963654 },
    // 这个缩放级别可以看到这个州的大部分
    zoom: 7,
```

```

        })
    const vacations = await fetch('/api/vacations').then(res => res.json())
    vacations.forEach(({ name, location }) => {
        const marker = new google.maps.Marker({
            position: location.coordinates,
            map,
            title: name,
        })
    })
}

```

现在就有了一张可以把所有的度假地点都显示出来的地图了！完善这个页面有很多种方法。最好的起点，大概就是给这些大头针标记加上到度假产品详情页的链接，这样点击一个标记就可以转到详情页。也可以实现定制的标记或提示信息。谷歌地图 API 有大量的特性，你可以通过它的官方文档来进一步学习。

19.3 天气数据

还记得第 7 章中的“当前天气”组件吗？现在，使用动态数据让它运行起来。我们将使用“全美天气服务”（NWS）API 来获取天气预报。正如 Twitter 集成和对地理编码的使用，我们会把预报信息缓存起来，避免每次访问网站都向 NWS 发出请求（如果网站变得很受欢迎，那么这样向 NWS 发送请求会让网站进入黑名单）。创建一个名为 lib/weather.js 的文件：

```

const https = require('https')
const { URL } = require('url')

const _fetch = url => new Promise((resolve, reject) => {
    const { hostname, pathname, search } = new URL(url)
    const options = {
        hostname,
        path: pathname + search,
        headers: {
            'User-Agent': 'Meadowlark Travel'
        },
    }
    https.get(options, res => {
        let data = ''
        res.on('data', chunk => data += chunk)
        res.on('end', () => resolve(JSON.parse(data)))
    }).end()
})

module.exports = locations => {

    const cache = {
        refreshFrequency: 15 * 60 * 1000,
        lastRefreshed: 0,
        refreshing: false,
        forecasts: locations.map(location => ({ location })),
    }
}

```

```

const updateForecast = async forecast => {
  if(!forecast.url) {
    const { lat, lng } = forecast.location.coordinates
    const path = `/points/${lat.toFixed(4)},${lng.toFixed(4)}`
    const points = await _fetch('https://api.weather.gov' + path)
    forecast.url = points.properties.forecast
  }
  const { properties: { periods } } = await _fetch(forecast.url)
  const currentPeriod = periods[0]
  Object.assign(forecast, {
    iconUrl: currentPeriod.icon,
    weather: currentPeriod.shortForecast,
    temp: currentPeriod.temperature + ' ' + currentPeriod.temperatureUnit,
  })
  return forecast
}

const getForecasts = async () => {
  if(Date.now() > cache.lastRefreshed + cache.refreshFrequency) {
    console.log('updating cache')
    cache.refreshing = true
    cache.forecasts = await Promise.all(cache.forecasts.map(updateForecast))
    cache.refreshing = false
  }
  return cache.forecasts
}

return getForecasts
}

```

可以看到，我们不满于直接使用内建的 `https` 库，取而代之的是创建一个辅助函数 `_fetch`，让天气功能更具可读性一些。我们把 `User-Agent` 请求头设置为了 `Meadowlark Travel`，你也许会对此感到奇怪。这是 NWS 天气 API 的一个怪异之处，它要求有一个 `User-Agent` 字符串，并声称最终会把这个请求头取代为一个 API key，但就目前来说，只需给它提供一个值就可以了。

从 NWS API 获取天气数据在这里是两部分工作。有一个 API 端点叫 `points`，负责接收一对经纬度（都包含 4 位小数）并返回关于这个地点的信息，其中包含着获取这个地点的天气预报的 URL。一旦获得了某一组坐标相应的 URL，就不需要再次获取了，只需要调用那个 URL 就可以获得最新的天气预报。

要注意，从天气预报所获得的数据比要使用的多很多，因此以后可以把这个功能做的高级很多。尤其是，这个天气预报 URL 会返回各个时段的数组 (`periods`)，其中第一个元素是当前时段（例如，下午或晚上），后面的时段一直延伸到下一周。可以自由查看 `periods` 数组中的数据，看看有没有对你有用的。

值得一提的一个细节是缓存中有一个叫作 `refreshing` 的属性。它是有必要的，因为更新缓

存需要一段时间，并且是异步完成的。如果在第一次缓存刷新完成前进来了多个请求，要是没有这个属性，这些请求就又会开始缓存刷新的工作。确实，这样也不会有什么害处，但它会发出更多 API 调用，而这些调用是不必要的。这个布尔变量就是一个标志，就好像在对所有后续请求说：“我们正在忙这事儿呢。”

第 7 章创建了一个 `getWeatherData` 函数，当时使用的是模拟数据，这里把它整个替换掉。需要做的是打开 `lib/middleware/weather.js`，替换 `getWeatherData` 函数：

```
const weatherData = require('../weather')

const getWeatherData = weatherData([
  {
    name: 'Portland',
    coordinates: { lat: 45.5154586, lng: -122.6793461 },
  },
  {
    name: 'Bend',
    coordinates: { lat: 44.0581728, lng: -121.3153096 },
  },
  {
    name: 'Manzanita',
    coordinates: { lat: 45.7184398, lng: -123.9351354 },
  },
])
```

现在，我们的天气组件就有实时数据了。

19.4 小结

对于第三方 API 集成能够做到什么，实际上我们只是触及了其皮毛。环顾四周，不断有新的 API 冒出来，提供着你所能想象的各种类型的数据（甚至波特兰市现在也把很多公开数据通过 REST API 开放了）。即使只是想要覆盖所有能访问的 API 中的一小部分，也是不可能的，但是使用这些 API 的必要基础知识本章都已经讨论到了，包括 `http.request`、`https.request` 和 JSON 解析。

目前为止，我们已经讨论了很多方面，也掌握了大量的知识。不过当事情出问题时该怎么办呢？下一章将讨论调试技术。如果事情未能按期望运行，那么调试技术可以帮助解决问题。

第20章

调试

“调试”或许是一个“不吉利”的词，因为它与缺陷关联在一起。但实际上，我们所说的“调试”是一种你一直在做的活动，不管是在实现一种新特性，还是在学习某个东西的运行机制，又或者是在修正一个 bug。或许“探索”是一个更好的词，但我们将继续使用“调试”，因为“调试”所指的活动不是模棱两可的，也不管活动的动机如何。

调试是一项常常被忽视的技能。似乎大家都觉得，绝大多数程序员生来就知道如何调试。也可能是计算机系的教授和技术图书的作者觉得调试很简单，以至于对其都不重视。

事实上，调试是可以进行教学的技能，而且调试是理解一个东西的重要途径。通过调试，程序员得以理解他们所工作的框架，也得以理解他们自己以及团队的代码。本章将讨论一些工具和技术，你可以把它们有效地用在 Node 和 Express 应用的调试中。

20.1 调试原则第一条

正如其名，“调试”常常是指查找和清除缺陷的过程。在开始讨论工具之前，先来考虑一些一般的调试原则。

“我跟你说过多少次了，当你排除了所有的不可能，剩下的无论有多么离奇，也必然是真相。”

——Sir Arthur Conan Doyle

调试原则第一条，同时也是最重要的一条是排除的过程。现代计算机系统复杂得不可思议，如果必须把整个系统都装进脑袋里，并在这巨大的空间中搜寻某一个问题的来源，那

么你很可能不知道从哪里开始。不管什么时候面对一个一眼看不穿的问题，你第一个想法都会是：“这个问题的来源我要怎么排除呢？”毕竟你排除得越多，需要看的地方就越少。

排除法可以有多种形式，以下是一些常见的例子。

- 有步骤地注释掉或禁用一些代码块。
- 编写能够被单元测试覆盖到的代码。单元测试本身提供了一个排除问题来源的框架。
- 分析网络流量，确定问题是产生在客户端还是服务器端。
- 找出系统中与发现问题的部分有相似之处的部分，并对其进行测试。
- 使用先前正常输入的数据，之后一次修改一点点，直到问题显现。
- 使用版本控制系统，来回切换版本直到问题消失，你就可以定位出特定的代码变更（更多信息请查看 `git bisect` 命令）。
- “模拟”某些功能，以便排除复杂的子系统。

不过，排除法并非是万能的。很多时候，问题是由于两个甚至更多组件之间的复杂交互造成的。只要排除（或模拟）其中任何一个组件，问题就可能会消失，但是并不能定位到其中的某个组件。不过即便是在这种情形下，排除也有助于缩减问题来源的范围，尽管它还没有直接给你指明问题所在的位置。

如果能做到细致而有条理，排除法是其中最成功的。如果只是漫无目的地排除一些组件，而不考虑它们是如何影响整体的，就很容易错过一些东西。你可以自己试试：当考虑要不要排除某个组件时，梳理一下如果移除这个组件会对系统产生怎样的影响。这样就会知道应该期望什么样的结果，以及移除这个组件能否告诉你有用的信息。

20.2 利用REPL和控制台

不管是 Node 还是浏览器，都给你提供了一个读取 – 运算 – 打印循环（REPL）。从根本上来说，这是写 JavaScript 的一种交互式方式。你键入一些 JavaScript，按回车键，然后立即看到输出。它非常适合做一些代码演练，如果要在一小块代码中定位错误，那么它也常常是最快速且最直观的方式。

在浏览器中，只需打开 JavaScript 控制台，你就有了一个 REPL。在 Node 中，只需键入 `node`，不用带任何参数，你就进入了 REPL 模式。你也可以引入包、定义变量和函数以及做任何想在代码中正常做的事情（除了创建包，在 REPL 中创建包是没有意义的）。

控制台日志也是你的伙伴。它确实是很粗糙的调试方式，但很易用（既易于理解也易于实现）。在 Node 中调用 `console.log` 会以易读的格式输出一个对象的内容，因而就能轻易地发现问题了。不过要记得，有些对象太大了，如果把它们记录到控制台，就会产生大量的输出，便很难找出有用的信息。例如，在你在一个路由处理函数里尝试一下 `console.log(req)`。

20.3 使用Node的内建调试器

Node 有一个内建的调试器，让你可以一步步地运行一遍自己的应用，就好像骑着 JavaScript 解释器兜风一般。只需增加一个 `inspect` 参数，就可以开启对应用的调试：

```
node inspect meadowlark.js
```

执行之后，你立即就会注意到一些东西。首先，会在控制台上看到一个 URL。这是因为这个 Node 调试器需要创建一个自己的 Web 服务器，这是它的工作方式。通过这个 Web 服务器，就可以控制执行所要调试的应用了。Web 服务器的这一点也许还没有让你“眼前一亮”，不过讨论到调试客户端的时候，它的用处就很清楚了。

当你进入了控制台调试器，就可以通过键入 `help` 来获得一个命令列表。这些命令当中你用得最多的是 `n`（下一语句）、`s`（步进）和 `o`（步出）。`n` 会单步执行当前代码行：它会执行这行代码，但如果这行代码调用了其他函数，那么待这些函数执行完毕控制才会交回给你。相对而言，`s` 命令会进入当前代码行：如果这行代码调用了其他函数，你就能够进入这些函数。`o` 命令让你可以从当前正在执行的函数中跳出来。（要注意，“步进”和“步出”仅仅是针对函数而言，它们并不会进入或跳出 `if` 和 `for` 代码块以及其他控制流语句。）

这个命令行调试器还有更多的功能，但很有可能你并不想过多地使用它。虽然对很多事情来说命令行非常好，但调试不在其中。虽然在紧要关头还有命令行可用也是个好事（例如，如果只能通过 SSH 访问服务器，或者服务器上没有安装图形用户界面），但更多的时候，你会更愿意使用图形化的调试客户端。

20.4 Node调试客户端

除非万不得已，很可能你不愿意使用命令行调试器。既然如此，Node 通过 Web 服务开放了调试控制，给了你其他的选择。

最简单直接的调试器是 Chrome。它所使用的调试界面跟前端代码的调试界面完全一样。因此你只要使用过前端代码的调试界面，就会感到很自然，上手很简单。使用 `--inspect` 选项启动你的应用（跟刚才提到的 `inspect` 参数不同）：

```
node --inspect meadowlark.js
```

现在事情开始有意思了：在浏览器的地址栏里，键入 `chrome://inspect`，就会看到 DevTools 页。在 Devices 部分，点击“Open dedicated DevTools for Node”，打开一个新窗口，这就是你的调试器了，如图 20-1 所示。

如图 20-2 所示，点击 Sources 标签页，然后在最左面一栏中点击 Node.js 展开它，再点击 `file://`，你会看到你的应用所在的目录。展开这个目录，就会看到你所有的 JavaScript 源代

码（只会看到 JavaScript；如果在哪里引入了 JSON 文件，那么也会看到它们）。从这里，你可以点击任何文件查看源代码，并设置断点。

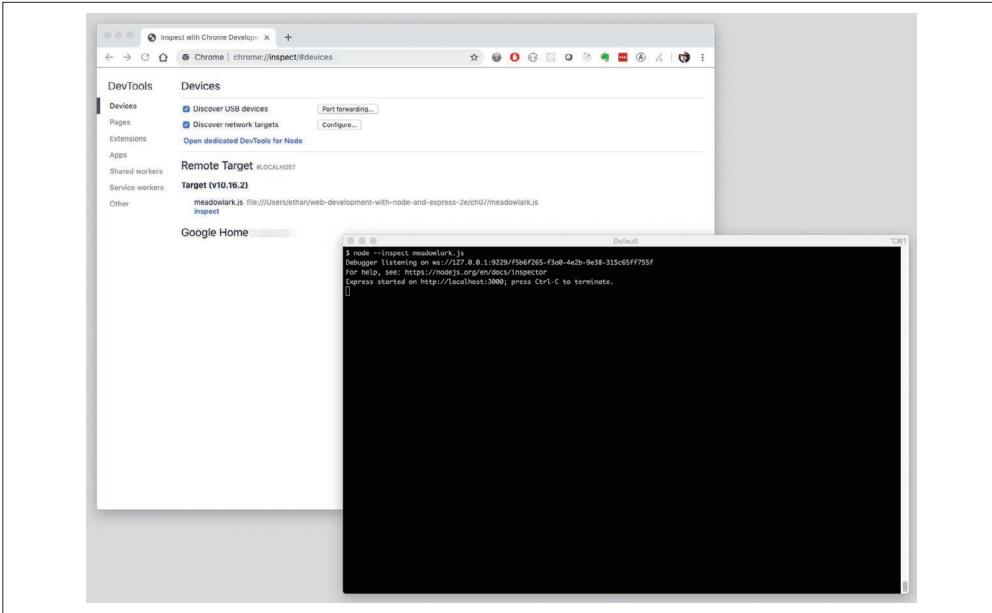


图 20-1：创建 Node 调试器

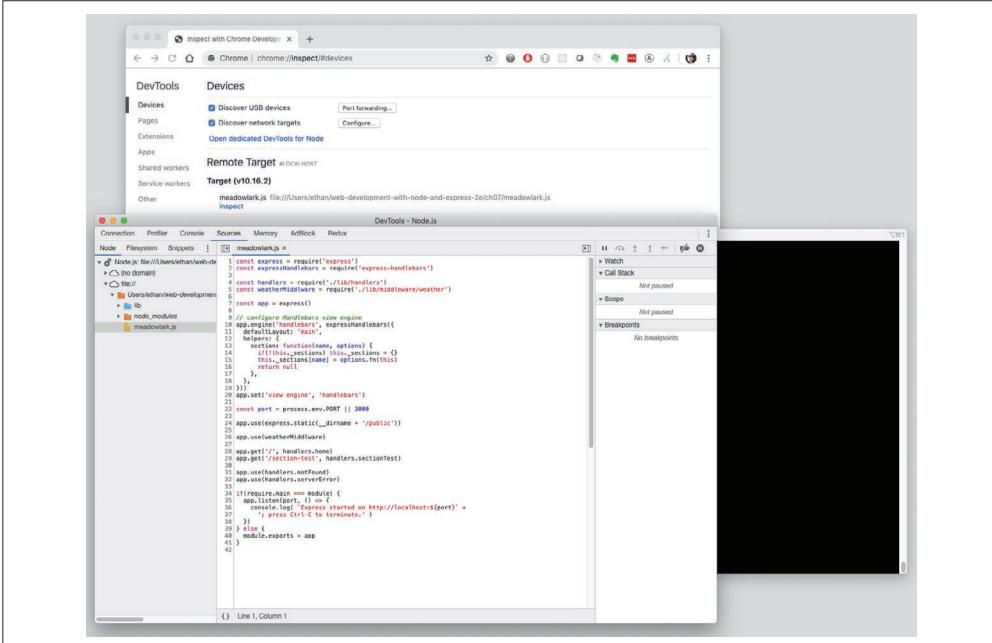


图 20-2：点击文件查看源代码

与前面使用命令行调试器的经历不同，现在你的应用已经在运行了：所有的中间件已经加载进来了，应用正在监听着端口。那么如何进入代码呢？最容易的（很可能也是你用得最多的）方式就是设置断点。设置断点是要告诉调试器，当执行到特定的一行时停下来，以便你可以单步执行代码。

要设置一个断点，你所需要做的只是在调试器的 file:// 文件浏览面板中打开一个源代码文件，然后点击文件的行号（代码左边的一栏），点了以后会出现蓝色的小箭头，指示在这一行上有一个断点（再点一次可以取消）。你可以尝试选一个路由处理函数，并在其中设置一个断点。然后在另一个浏览器窗口里访问这个路由。如果你使用的是 Chrome，浏览器就会自动切换到调试窗口，而访问路由的窗口只是在那儿旋转（因为服务器已经暂停了，还没有响应那个请求）。

你可以在调试器窗口里单步执行这个程序。比起先前使用命令行调试器，现在的可视化效果要好得多。你会看到设置了断点的那一行现在已经是蓝色高亮了。这意味着那一行已经成为当前的执行行了（其实是将要执行的下一行）。命令行调试器中能使用的那些命令在这里同样能使用。跟命令行调试器类似，我们可以采取以下操作。

恢复执行 (F8)

这相当于把它“放飞”了，不能再单步执行，直到它遇到另一个断点停下来。通常在你已经看到了需要看的，或者想直接跳到另一个断点时，就会使用这个命令。

执行但不进入函数调用 (F10)

如果当前行调用了一个函数，那么调试器将不会进入这个函数里。也就是说，这个函数会被执行，但执行之后，调试器会前进到下一行。如果你正处于一个函数调用，但对这个函数的细节不感兴趣，就可以使用这个命令。

进入函数调用 (F11)

将进入函数调用里，这个函数会对你暴露无遗。如果你一直如此操作，那么最终，任何被执行的东西你都能看到——初听起来很有意思，但如果你照那样研究上一个小时，则对于 Node 与 Express 为你所做的工作会有新的认识。

跳出当前函数 (Shift-F11)

将执行当前函数的剩余部分，并把调试恢复到这个函数的调用者的下一行。最常见的情况是，当你不经意地进入了一个函数，或者在这个函数里已经得到了需要的东西，都会使用这个命令。

除了可以做所有这些控制，你还可以访问一个控制台。这个控制台运行在应用当前上下文中。因此你可以查看甚至修改当前上下文中的变量，也可以调用函数。如果只是试验一些确实很简单的东西，那么还是非常方便的，但很快就会导致混乱，因此我不鼓励你过多地以这种方式动态修改运行中的应用，这样做很容易被搞糊涂。

在调试器的右边部分中，会有一些对你有用的数据。从上往下，先是观察表达式（watch expression），你可以在这里定义 JavaScript 表达式，在单步执行应用的时候，它们的值会被实时更新。例如，你可以在这里输入希望跟踪的变量。

观察表达式之下是调用栈（call stack），它向你展示了你是如何到达现在的位置的。也就是说，当前所在的函数是被某个函数调用的，而那个函数又是被另一个函数调用的，这个调用栈列出了所有这些函数。在高度异步的 Node 世界里，调用栈可能很难梳理和理解，尤其是涉及匿名函数的时候。列表的最顶部就是我们目前所在的函数。往下的一条，就是调用我们所在函数的函数，以此类推。如果你点击列表中的任何一个条目，就会神奇地切换到那个上下文，而所有观察表达式和控制台的上下文都会变成那个上下文。

调用栈之下是作用域变量。正如其名，这些是当前作用域中的变量（也包括那些对我们可见的父级作用域中的变量）。如果代码中有一些你第一眼就看中的变量，那么作用域变量常常能给你提供很多信息。如果变量有很多，变量列表将变得很庞大，那么还不如把看中的那些变量定义成观察表达式。

再往下是所有断点的一个列表，实际上只是断点的登记。当你调试一个比较棘手的问题并且设置了很多断点时，有这个列表很方便。点击其中一个，就会转到这个断点（但它并不像点击调用栈一样会改变上下文。这是合理的，因为并非每个断点都会代表一个活跃的上下文，而调用栈中的一个条目就代表一个活跃的上下文）。

有些时候，你需要调试的是应用的设置过程（例如把中间件链接进来的过程）。如果还像前面那样启动调试器，在能够设置断点之前，这个过程就已经一闪而过了。我们所做的，就是在启动调试器时指定 `--inspect-brk` 选项，而不是 `--inspect` 选项：

```
node --inspect-brk meadowlark.js
```

调试器会在应用的第一行中断下来，然后你就可以单步执行，或者在认为合适的地方设置断点了。

Chrome 并非调试客户端的唯一选择，特别是使用 Visual Studio Code (VSC) 时，因为 VSC 有非常好的内建调试器。在 VSC 中，启动应用时无须带上 `--inspect` 或 `--inspect-brk` 选项，只需在窗口左边沿菜单上点击调试图标（一只虫子被一根线穿过）。在边栏的顶部，你会看到一个小齿轮图标，点击它就可以打开某个调试配置。唯一需要注意的设置是“program”，确保它指向你的入口点（例如 `meadowlark.js`）。



你可能还需要设置当前工作目录或 "cwd"。例如，如果是在 `meadowlark.js` 所在目录的父目录中打开的 VSC，就需要设置 "`cwd`"（这就跟先让 `cd` 进入正确的目录，然后再运行 `node meadowlark.js` 是一样的）。

设置好了之后，只需点击调试栏上的绿色“播放”箭头，调试器就开始运行了。它的界面跟 Chrome 中有些许不同，但如果开发使用的是 VSC，那么你就会很习惯。更多信息请参见 VSC 的官方文档中“USER GUIDE”的 Debugging 部分。

20.5 调试异步函数

当人们初次接触异步编程时，最常见的困难之一就是调试。以下面的代码为例：

```
1 console.log('Baa, baa, black sheep;');
2 fs.readFile('yes_sir_yes_sir.txt', (err, data) => {
3     console.log('Have you any wool?');
4     console.log(data);
5 })
6 console.log('Three bags full.')
```

如果你是异步编程的新手，那么可能会期望这样的结果：

```
Baa, baa, black sheep,
Have you any wool?
Yes, sir, yes, sir,
Three bags full.
```

但是你会失望的，因为看到的会是这样的结果：

```
Baa, baa, black sheep,
Three bags full.
Have you any wool?
Yes, sir, yes, sir,
```

如果对此感到疑惑，那么调试大概也不会对你有帮助。先从第 1 行开始，然后单步执行它，进入第 2 行。接下来执行“步进”操作，期望进入函数，从而停在第 3 行，但实际上你到了第 5 行。这是因为，`fs.readFile` 只有在读取完文件之后，才会调用这个函数，而读取文件要等到你的应用空闲时才会开始。如果你在第 5 行单步执行，就会停留在第 6 行……你可以继续单步执行，但始终到不了第 3 行（最终会到达，但要过好久）。

不过如果你想调试第 3 或第 4 行，那么只需在第 3 行设置断点，然后让调试器运行。当文件读取完并且这个函数得到调用的时候，就中断在那一行上了，这样就有望把问题搞清楚了。

20.6 调试 Express

如果你像我一样，在职业生涯中见过无数过度设计的框架，那么进入框架单步执行的想法，在你看来无异于疯了（或者是一种折磨）。浏览 Express 的源代码不是儿戏，然而，对于一个对 JavaScript 和 Node 有不错的理解的人来说，Express 的源代码是完全可以掌握的。每当你的代码存在问题时，如果进入 Express 的源代码（或第三方的中间件）进行调试，问题就会得到更好的解决。

本节将对 Express 的源代码做一个简短的讲解，帮助你更有效地调试 Express 应用。在讲解的各个部分，我会给出相对于 Express 的根目录（可以在 node_modules/express 目录下找到）的文件名以及相应的函数。我不使用源代码中的行号，因为很显然，鉴于所使用的 Express 的具体版本不同，我们之间的行号可能会有差异。

Express 应用创建 (lib/express.js, function createApplication)

这里是 Express 应用生命周期开始的地方。当你在代码中调用 `const app = express()` 时，被调用的就是这个函数。

Express 应用初始化 (lib/application.js, app.defaultConfiguration)

这里是 Express 初始化的地方。要想查看 Express 启动时所有的默认设置，这里也是不错的地方。几乎没有必要在这里设置断点，但至少在这里单步执行一次还是很有用的，因为可以让你对 Express 的默认设置有一个大致感觉。

加入中间件 (lib/application.js, app.use)

每次 Express 链入中间件时（不管是显式链入的，还是 Express 或任何第三方显式链入的），这个函数都会得到调用。它看起来简单，但简单得很有欺骗性，要真正理解它需要下一些功夫。有时在这里加个断点很有用（运行应用时应该使用 `--debug-brk`，否则，在你能够设置断点前，所有的中间件都已经被链入了），但可能会让你目不暇接：你会奇怪，在一个普通的应用中竟然有这么多中间件被加入进来。

渲染视图 (lib/application.js, app.render)

这是另一个比较复杂的函数，但如果需要调试跟视图有关的棘手问题，就很有用了。如果你进入这个函数单步执行，就会看到你的视图引擎是怎样被选择并调用的。

request 扩展 (lib/request.js)

你可能会奇怪，这个文件代码如此稀少且容易理解。Express 往 request 对象中添加的方法大部分是使用起来很简便的函数。由于代码简单，几乎没必要在这里单步执行或者设置断点。不过，如果想要理解 Express 的一些简便方法是如何工作的，查阅这里的代码通常很有帮助。

发送响应 (lib/response.js, res.send)

无论你如何构造一个响应，是使用 `.send`、`.render`，还是使用 `.json`、`.jsonp`，几乎差不多，因为最终都会到达这个函数（`.sendFile` 是个例外）。因此这里是适合设置断点的位置，因为每个响应都会调用到这里。你可以从调用栈看到响应是如何到达这里的。当你思索问题出现在哪里时，调用栈就很有用了。

response 扩展 (lib/response.js)

尽管 `res.send` 内容丰富，但 response 对象的大部分方法是十分简单直白的。有些时候，为了看清楚应用响应一个请求的过程，在这些函数里设置断点很有用。

static 中间件 (node_modules/serve-static/index.js, function staticMiddleware)

一般来说，如果静态文件未按期望提供，问题则是出在你的路由，而不是这个 static 中间件，因为路由优先于 static 中间件。如果你有一个文件 public/test.jpg 以及一个路由 /test.jpg，那么由于这个路由的存在，static 中间件将永远都不会得到调用。不过，如果需要设置不同于静态文件的响应头，那么深入 static 中间件的源代码应该会有所帮助。

如果你绞尽脑汁想知道所有中间件都在哪里，那是因为 Express 本身带的中间件很少 (static 中间件和路由是特别的例外)。

就跟你在努力解决一个难题时深入 Express 的源代码会有所帮助一样，你可能也不得不深入中间件的源代码里面。这些中间件确实多到看不过来，但有 3 个我想提一下，因为对于理解在一个 Express 应用中会发生什么它们十分关键。

session 中间件 (node_modules/express-session/index.js, function session)

要让 session 正常工作，涉及的东西很多，但是它的代码相当直白。如果遇到跟 session 有关的问题，就可以在这个函数里设置断点。要记住，为 session 中间件提供存储引擎由你决定。

logger 中间件 (node_modules/morgan/index.js, function logger)

logger 中间件的存在其实是为了辅助调试，而不是为了调试本身。然而，日志的工作方式存在一些微妙之处，只有深入 logger 中间件的源代码一两次才能够理解。我第一次进入源代码单步执行时，有过很多次“原来如此”的时刻，之后发现，在应用中自己对日志的使用更高效了。所以我推荐你至少深入阅读一次这个中间件。

对 URL-encoded 编码的 body 的解析 (node_modules/body-parser/lib/types/urlencoded.js, function urlencoded)

人们常常会觉得请求 body 的解析方式很神秘。其实并没有那么复杂。进入这个中间件单步执行会有助于你理解 HTTP 请求的工作方式。除了对它本身进行学习，你会发现调试时常常是不需要进入这个中间件的。

20.7 小结

本书已经讨论了大量的中间件。在探索 Express 内部机制的过程中，对于你要看些什么，我无法逐个列出所有要点，但希望本书特别提到的内容，能够在一定程度上为你消除 Express 的神秘感，而且在必要的时候，希望你能够有勇气探索 Express 的源代码。中间件各式各样，它们在质量上和可理解性上都差异极大：有的极其难以理解，有的则可以一望到底。不管是哪一种情形，都不妨看一看。如果太过复杂，你可以不予理会（当然，除非你的确需要理解它）；反之，你可以从中学到一些东西。

第21章

上线

大日子就要来临了。在此之前你已经勤勤恳恳地工作了几周甚至几个月，现在，你的网站或服务已经做好了发布的准备。不过，发布并不是那么简单的，至少不是按一下开关然后网站就上线了，是吧？

通过本章（其实，你应该在发布前几周就读本章的内容，而不是等到发布的那一天），你将学到：几个域名注册与托管服务、从准生产环境向生产环境迁移的技术、部署生产的技术，以及选择生产服务时需要考虑的一些东西。

21.1 域名注册与托管

人们常常会混淆域名注册与域名托管。如果你在阅读本书，很可能就不会混淆，但我敢肯定你认识几个会混淆二者的人，比如你的经理或你的客户。

互联网上的每个网站和服务都可以用一个（或多个）IP（Internet Protocol）地址来识别。这些IP数字对人类来说并不是特别友好（随着IPv6得到广泛应用，不友好的程度只会增加），但你的计算机为了展现网页，从根本上还是需要这些数字的。这就是域名起作用的地方了，它们把一个对人类友好的名字跟一个IP地址（74.125.239.13或2601:1c2:1902:5b38:c256:27ff:fe70:47d1）相映射了。

域名跟IP地址的差别用现实世界的例子做类比，就相当于公司名跟公司地址的差别。一个域名就相当于你的公司名（Apple），一个IP地址就相当于公司的实际地址（美国加利福尼亚州库比蒂诺市Apple园区）。如果你需要驾车拜访Apple公司的总部，就需要先知道它的实际地址。幸运的是，如果你已经知道了公司名，就大概能够获得它的实际地址。从公

司名到公司地址的这一层抽象是有用的，其另一个原因是，一个组织可能会搬家（得到一个新的公司地址），但就算它搬家了，人们也依旧能够找到（事实上，在本书第1版和第2版出版之间，Apple公司的总部就搬了一次）。

而托管，说的是运行网站的计算机。继续以实际事物作类比，托管就相当于你到达实际地址后所看到的建筑。域名注册跟托管其实没有什么关系，这一点人们常常会混淆。况且，通常你并不是从托管商那里购买域名（与之类似，你从一个人手里买到了一块土地，但通常需要向另外一个人付费，让他来建造和装修房子）。

尽管可以运行网站而不使用域名，但是这样很不友好，因为IP地址并不适合做市场推广。通常在购买托管服务时，系统会自动给你分配一个子域名（一会儿就要讨论），可以认为这个子域名是介于适合市场推广的域名与IP地址之间的。

一旦有了域名，并且网站上线之后，你就可以通过多个URL访问到你的网站。例如：

- <http://meadowlarktravel.com/>
- <http://www.meadowlarktravel.com/>
- <http://ec2-54-201-235-192.us-west-2.compute.amazonaws.com/>
- <http://54.201.235.192/>

多亏了域名映射，所有这些地址才都指向了同一个网站。一旦请求到达了你的网站，基于请求所使用的URL来采取不同的操作也是可以做到的。比如说，如果有人使用IP地址访问到了你的网站，你就可以自动地重定向到域名，不过这样做并不常见，因为没有什么意义（更常见的是从<http://meadowlarktravel.com/>重定向到<http://www.meadowlarktravel.com/>）。

大多数域名注册商会提供托管服务（或者跟提供托管服务的公司合作）。除了AWS以外，我还没有找到在托管方面做得特别好的域名注册商，而且，域名注册和托管可以分离。

21.1.1 域名系统

域名系统(DNS)负责把域名映射到IP地址。这个系统相当复杂，但作为网站的所有者，有一些关于DNS的东西你应该知道。

21.1.2 安全

时刻都要记住，域名是有价值的。假使有一个黑客完全破坏了你的托管服务，并控制了你的网站，但如果域名仍旧掌握在自己手里，你就可以找一个新的托管服务器，并把域名指向它。反过来说，如果你的域名本身遭到了破坏，那就真的遇到麻烦了。你的声誉是跟域名紧密联系在一起的，好的域名应该得到小心的保护。失去对自己域名的控制的那些人知道结果可能是毁灭性的。这个世界上有一些人正在对你的域名（特别是又短又好记的域名）蠢蠢欲动，企图转卖你的域名、毁坏你的声誉，或者以此来勒索你。所以，你应该十

分严肃地对待域名安全这个问题，可能你的域名比数据还重要（取决于数据有多大价值）。我就看到过有一些人，他们投入到托管安全上面的时间和金钱多得离谱，使用的域名注册服务却是所能找到的最廉价、最简陋的。千万别犯这样的错误。（幸运的是，优质的域名注册服务并非特别昂贵。）

既然对域名所有权的保护如此重要，就应该为域名注册采用一些好的安全实践。至少，你应该使用高强度的、独特的密码，并保持良好的密码使用习惯（不会把密码记在贴纸上然后贴到你的显示器上）。最好选用一个支持两段式认证的域名注册商。不要害怕向域名注册商提出尖锐的问题，比如变更账号都需要些什么授权之类的。

注册域名时，必须提供一个跟这个域名关联的第三方邮箱地址（例如，如果你在注册 meadowlarktravel.com，就不能使用 admin@meadowlarktravel.com 作为注册的邮箱地址）。由于安全系统的安全强度取决于它最薄弱的一环，因此你应该使用一个安全性很好的邮箱。使用 Email 或 Outlook 账号是很常见的，如果你这样做了，就应该跟域名注册账号采取同样的安全标准（良好的密码使用习惯和两段式认证）。

21.1.3 顶级域名

你的域名末尾的部分（比如 .com 和 .net）称为顶级域名（TLD）。一般来说，有两种类型的顶级域名：国家（地区）代码顶级域名和通用顶级域名。国家（地区）代码顶级域名（比如 .us、.es 和 .uk）本意是为了地理上的划分。然而，对于谁能够取得这些顶级域名，其实没有什么限制（毕竟，互联网是真正的全球网络），因此，国家（地区）代码顶级域名常常被用于一些“精巧”的域名，比如 placeholder.it 和 goo.gl。

通用顶级域名包括我们熟悉的 .com、.net、.gov、.fed、.mil 和 .edu。任何人都可以取得某个 .com 或 .net 域名，但对于其他一些域名，则存在限制。要了解更多信息，请看表 21-1。

表21-1：受限的通用顶级域名

TLD	更多信息
.gov、.fed	dotgov.gov
.edu	net.educause.edu
.mil	军事方面的人员和承包商应当跟他们的 IT 部门联系，或访问“美国国防部统一注册系统”

顶级域名的管理最终由互联网名称与数字地址分配机构（ICANN）来负责，虽然它的很多管理工作实际委派给了其他组织。近来，ICANN 又批准了一大批新的通用顶级域名，比如 .agency、.florist、.recipes，甚至还有 .ninja。在可预见的将来，.com 很有可能仍是稀缺的顶级域名以及最抢手的不动产。在互联网发展早期，那些买入了 .com 域名的幸运儿（或者精明的人）因为最优质的域名而收获了大量的回报。

由于 .com 域名稀缺，人们只好寻求其他的顶级域名，或者使用 .com.us，力求获得一个能

够精确体现其组织的域名。选择域名时，应该考虑域名要怎么用。如果计划主要用于在线的市场推广（人们更有可能点击链接而不是手动输入域名），就应该更看重易记的或有意义的域名，而不是短域名。如果你关注的是推送广告，或者有理由相信人们会把你的 URL 手动输入到设备里，或许应该考虑其他的顶级域名，以便获得一个更短小的域名。使用两个域名也是很常见的做法：一种是短小的、易于输入的；一种是更长一些的、更适合市场推广的。

21.1.4 子域名

顶级域名是域名的末尾部分，子域名则是开头部分。目前为止，最常见的子域名是 www。我从来没有特别注意过这个子域名，毕竟，你就在计算机上使用着万维网。可以很肯定地说，即使没有 www 提醒你自己正在干什么，你也不会感到疑惑。因此，对你的主域名，我不推荐使用子域名。直接使用 <http://meadowlarktravel.com/> 而不是 <http://www.meadowlarktravel.com/>，这样域名就可以变得更简短，而且多亏了重定向，对于那些不管访问什么都自动地在前面加上 www 的人，他们的访问也不会有丢失的危险。

子域名也被用于其他的目的。我经常见到类似于 blogs.meadowlarktravel.com、api.meadowlarktravel.com 和 m.meadowlarktravel.com（用于移动版网站）这样的域名。这样做常常是出于技术的原因。例如，如果跟网站的其他部分相比，你的博客使用的是完全不同的服务器，那么让博客使用一个子域名就会更容易。不过，不管是基于子域名还是路径，一个好的代理都能够妥当地重定向网站的流量，因此，选择使用子域名还是路径，应该更关注内容而不是技术（别忘了 Tim Berners-Lee 所说的，URL 表示你的信息架构，而不是技术架构）。

我推荐将子域名用于划分你的网站或服务中截然不同的各部分。例如，使用 api.meadowlarktravel.com 来开放你的 API 就是子域名一个很好的应用。微站点（跟网站其余部分外观不同的站点，通常用来凸显某一个产品或专题）也是不错的子域名候选应用。子域名的另一个合理应用是把管理界面从大众界面中分离出来（例如 admin.meadowlarktravel.com，只限雇员访问）。

除非特别指定，你的域名注册商会把所有的流量都重定向到服务器，不管子域名是什么。接下来，要基于子域名采取怎样的适当操作由你的服务器（或代理）来决定。

21.1.5 域名服务器

让域名起作用的“胶水”是域名服务器。在为网站建立托管的时候，你会被要求提供域名服务器。一般来说，这都是十分简单且好操作的，因为你的托管服务会做大部分的工作。不妨举个例子，我们选择了在 DigitalOcean 上托管网站 meadowlarktravel.com。在 DigitalOcean 上设置托管账户时，你会得到 DigitalOcean 的域名服务器的主机名（为了有

冗余，会有多个）。像大多数托管服务供应商一样，DigitalOcean 会把它的域名服务器称作 ns1.digitalocean.com、ns2.digitalocean.com，等等。之后在你的域名注册商系统中，为托管的域名设置几个域名服务器，如此就大功告成了。

在这个例子中，域名映射的工作机制如下。

1. 用户访问 <http://meadowlarktravel.com/>。
2. 浏览器把请求发送到用户计算机的网络系统。
3. 用户计算机的网络系统（互联网接入供应商已经给它分配了 IP 地址和 DNS 服务器）请求 DNS 服务器解析域名 meadowlarktravel.com。
4. 被请求的 DNS 服务器注意到，meadowlarktravel.com 是由 ns1.digitalocean.com 处理的，因此它向 ns1.digitalocean.com 请求 meadowlarktravel.com 的 IP 地址。
5. ns1.digitalocean.com 的服务器接收到了请求，认出 meadowlarktravel.com 的确是一个活跃的账号，于是返回它所关联的 IP 地址。

尽管以上是最常见的情形，但它并非配置域名映射的唯一方式。由于支持你的网站的服务器（或代理）有一个 IP 地址，因此可以去掉中间步骤，直接把这个 IP 地址注册到 DNS 服务器上（这样就有效地去掉了前面例子中的中间步骤，即域名服务器 ns1.digitalocean.com）。这种方式可行的前提是，托管服务供应商给你分配的 IP 地址必须是静态的。通常情况下，托管服务供应商给你的服务器分配的 IP 地址是动态的，这意味着 IP 地址随时都有可能变更，但你不会得到通知，那么，去掉中间步骤的方案就无效了。想要获得一个静态 IP 而不是动态 IP，有时花费会更高，请跟你的托管服务供应商确认一下。

如果你想把域名直接映射到网站（略过网站主机的域名服务器），那么要么增加一条 A 记录，要么增加一条 CNAME 记录。A 记录会直接把域名映射到 IP 地址，而 CNAME 记录会把一个域名映射到另一个域名。一般来说 CNAME 记录灵活性会低一些，因此 A 记录通常更受青睐。



如果你正在使用 AWS 的域名服务器，那么除了 A 记录和 CNAME 记录，还有一种记录叫作 alias，如果你让它指向在 AWS 上托管的一个服务，这种记录就有很多优势了。更多信息请查看 AWS 的文档“Choosing between alias and non-alias records”。

不管使用的是什么技术，通常域名映射都会得到积极的缓存，这意味着如果你修改了域名记录，那么最多花费 48 小时就能让域名关联到新服务器。要记住这是跟地理位置有关的。如果你在洛杉矶看到域名生效了，那么在纽约的客户看到的域名可能还是关联着前一个服务器。根据我的经验，要让域名在北美大陆内得到正确解析，一般 24 小时就够了，但是想要在全世界范围内生效，最多需要 48 小时。

如果需要精确地在某个时间上线某个东西，就不能依赖于域名的变更。你应该配置你的服务器，让它重定向到“即将上线”站点或页面，并且提前设置好 DNS。然后在预定的时刻，可以让服务器切换到新上线的站点，这样，不管你的用户在世界的哪个角落，就都能立刻看到变更了。

21.1.6 托管服务

起初，托管服务可能会让你眼花缭乱，不知如何选择。Node 已经大获成功，广受欢迎，很多人高嚷着要求提供 Node 托管服务来匹配需求。而如何选择一个服务供应商，很大程度上取决于你的需求。如果你有理由相信你的网站将是下一个亚马逊或 Twitter，那么需要考虑的问题就跟为当地的集邮俱乐部搭建一个网站要考虑的问题完全不一样了。

1. 传统托管还是云托管

“云”是近年来突然出现的、最模糊的技术术语之一。没错，它不过是“互联网”或“互联网的一部分”的一种花哨说法。然而，这个术语也并非毫无用处。云端托管，通常隐含着对计算资源的某种商品化，尽管这不是“云”这个术语技术定义的一部分。也就是说，我们不再把一台服务器看作一个独特的物理实体，它只是云端某个地方的一个同质的资源，服务器之间没有好坏之分。当然，我想的有些简单化了，毕竟计算资源是根据它们的内存、CPU 数等配置进行区分（和定价）的。传统托管和云托管的区别在于，传统托管知道（并关心）你的应用实际托管在哪一台服务器上，而云托管知道你的应用托管在云端的某一台服务器上，并且在你不知道（也不会关心）的情况下，它可以轻易地转移到另一台服务器。

云托管也是高度虚拟化的。也就是说，运行着你的应用的服务器通常不是物理机器，而是运行在物理机器上的虚拟机。虽然虚拟化的思想并非云托管引入，但是云托管基本上已经跟虚拟化同义了。

尽管云托管起初并不受人瞩目，但现在，它已远不只“同质化的服务器”这么简单了。主要的云供应商提供了很多基础性的服务，（在理论上）给你减轻了维护的负担，并带来了更高的伸缩性。这些服务包括数据库存储、文件存储、网络队列、认证、视频处理、通讯服务、AI 引擎，等等。

云托管刚开始可能会有一点儿令人不安，因为你对实际运行你的服务的物理机器一无所知，所以只好相信你的服务不会受到运行在同一台机器上的其他服务的影响。不过，托管服务其实并没有什么变化。当托管账单到来的时候，你仍然是为同样的服务买单：物理硬件和网络的工作自有专人打理，以保证你的 Web 应用能够正常运行。要说有什么变化，就是你更加远离硬件了。

我相信“传统”托管（还没有更好的叫法）最终会完全消失。并非说这些提供托管服务的公司会倒闭（虽然其中一些不可避免），而是说它们也要开始提供云托管服务了。

2. XaaS

在考虑云托管时，你会遇到这些缩写词：SaaS、PaaS、IaaS 和 FaaS。

软件即服务（SaaS）

SaaS 通常描述的是给你提供的软件（网站、应用），比如说谷歌文档或 Dropbox，只管使用就可以了。

平台即服务（PaaS）

PaaS 给你提供了所有的基础设施（操作系统、网络，所有这些都为你解决了），你只管写应用就可以了。PaaS 与 IaaS 之间的界线常常很模糊（作为开发者，你常常会发现自己处于这条界线之上），但总体来讲，PaaS 就是本书所讨论的服务模型。如果你是在运行一个网站或 Web 服务，那么很可能 PaaS 就是你要找的。

基础设施即服务（IaaS）

IaaS 为你提供了最大的灵活性，但成本也随之上升。你所获得的只是一批虚拟主机以及把它们连接起来的基本网络，然后负责安装和维护操作系统、数据库以及网络策略。除非你需要对环境达到这个级别的控制，否则通常都会选择 PaaS。（注意，PaaS 也允许你选择操作系统和网络配置，只不过不需要你亲自去做罢了。）

函数即服务（FaaS）

FaaS 描述的是类似于 AWS Lambda、谷歌 Functions 和 Azure Functions 这样的服务，它提供了在云端单独运行一些函数的一种方式，不需要你自己去配置运行环境。通常所说的“无服务器”架构核心就是 FaaS。

3. 云服务领域的巨头

实质上，互联网运营公司（或者至少在互联网运营上投入很大的公司）都已经认识到，由于计算资源的商品化，他们拥有了另一个可以销售的商品。亚马逊、微软和谷歌都在提供云计算服务，而且服务都非常好。

所有这些服务定价都很接近，如果你对托管的要求不高，那他们 3 家的价格相差无几。但是根据你的需求，如果你对带宽或者存储的要求很高，就得更仔细地评估这些服务了，因为价格差异会更大。

虽然说在考虑开源平台时一般不会第一时间想到微软，但我是不会忽略 Azure 的。不仅是因为 Azure 已经久经考验，而且微软已经为此付出了很大的努力，让 Azure 对 Node 乃至整个开源社区更加友好。微软提供了一个月的 Azure 试用，这非常有用，你可以确定它是否符合你的需要。如果你在考虑以上三巨头之一，那么我肯定会推荐你免费试用 Azure。微软为所有主要的服务都提供了 Node 的 API，包括它的云存储服务。除了出色的 Node 应用托管，Azure 还提供了卓越的云存储系统（带有 JavaScript API），而且对 MongoDB 也有很好的支持。

亚马逊提供了最全面的资源整合，包括短信服务、云存储、电子邮件、支付服务（电子商务）、DNS，等等。此外，亚马逊也提供了各种免费套餐，使我们可以很容易地对这些服务做评估。

谷歌的云平台发展很快，现在可以提供健壮的 Node 托管服务了，而且正如你的期望，它的云平台跟服务集成得非常好（它的地图、认证和搜索服务尤其吸引人）。

除了这三巨头，Heroku 也值得考虑。Heroku 已经为那些希望托管快速、敏捷的 Node 应用的用户提供相当长时间的服务了。此外，还有给我带来过很多好运的 DigitalOcean，它更侧重于提供应用容器和有限数量的服务，并且对用户非常友好。

4.“精品店”托管

我准备把更小一些的托管服务叫作“精品店”托管服务，或许没有像微软、亚马逊和谷歌那样的基础设施或资源，但这并不意味着它们就不能提供有价值的东西。

由于精品店托管服务在基础设施方面没有竞争力，因此它们通常更关注客户服务和支持。如果你需要很多的支持，可能就会考虑一个精品店托管服务。如果你有一个相处愉快的托管服务商，不要犹豫，赶紧询问是否提供（或计划提供）Node 托管服务。

21.1.7 部署

很让我惊讶的是，2019 年人们还在使用 FTP 来部署应用。如果你是这样做的，那么赶快停止吧。FTP 根本就不安全。它传输的所有文件都是未加密的，连你的用户名和密码都未加密。如果托管服务供应商不给你另外的选择，就干脆再找一个供应商。如果实在没有别的选择了，那么一定要使用一个独特的密码，并且不在别的地方使用这个密码。

至少应该使用 SFTP 或 FTPS（不要把两者混淆了），但其实你应该考虑使用持续交付（CD）服务。

持续交付背后的思想是你不会距离可以发布的版本太远（只有几周甚至几天）。持续交付通常跟持续集成（CI）密不可分。持续集成指的是将多名开发人员的工作集成到一起，并对这些工作成果进行测试的自动化过程。

总体而言，这些过程自动化程度越高，部署就越容易。想象一下，从合并代码到收到单元测试通过而自动发出的通知，到集成测试通过的通知，再到看见代码变更已经上线……这些不过是几分钟的事。这不是个小目标。要把这一套建立起来，一开始就需要做相当多的工作，并且随着时间的推移，还会有维护的工作。

尽管这些 CI/CD 过程步骤本身相似（运行单元测试、运行集成测试、部署到准生产服务器、部署到生产服务器），但是建立 CI/CD Pipeline（即管线。在讨论 CI/CD 时，你常常会听到这个词）的做法多种多样。

你应该看看 CI/CD 的候选方案，并选择符合需求的一个。

AWS 的 CodePipeline

如果是托管在 AWS 上，那么 CodePipeline 是首选，因为对你来说，它是最容易达成 CI/CD 的路径。CodePipeline 非常健壮，但我认为它没有其他的选择那么用户友好。

微软 Azure 的 Web Apps

如果是托管在 Azure 上，那么 Web Apps 就是你的最佳选择。（你注意到这样一个趋势了吗？）我还没有太多的使用经验，但看起来它在社区里十分受欢迎。

Travis CI

Travis CI 出现已经有相当长的时间了，它有大量的忠诚用户，所做的文档也很不错。

Semaphore

Semaphore 易于搭建和配置，但并没有提供太多特性。它的基本（低成本）套餐并不快。

谷歌 Cloud Build

我还没有用过谷歌 Cloud Build，它看起来很健壮，与 CodePipeline 和 Azure 的 Web Apps 也类似，如果托管在谷歌云端，那么它很可能是最好的选择。

CircleCI

CircleCI 是另一个出现了很长时间的 CI，而且很受人喜爱。

Jenkins

Jenkins 有着庞大的社区。虽然并没有像其他一些方案一样紧跟现代的部署实践，但它刚刚发布了一个版本。就我的经验而言它看起来前景不错。

最后，CI/CD 服务会将你创建的这些活动自动化。但你仍旧需要编写代码、确定版本化方案、写高质量的单元测试和集成测试并且把这些测试运行起来，以及理解部署的基础设施。本书中的示例都可以轻易地实现自动化，因为它们都可以部署到运行着一个 Node 实例的单台服务器上。然而，随着基础设施的增长，CI/CD 管线的复杂性也会随之增长。

1. Git 在部署中的角色

Git 最强大的地方（也是最薄弱的地方）就在于它的灵活性。它可以适应近乎所有可以想象的工作流程。为了便于部署，我推荐创建一个或多个专门用于部署的分支。例如，或许你应该有一个 `production`（生产）分支和一个 `staging`（准生产 / 生产预览）分支。这些分支要如何使用，很大程度上取决于你的工作流程。

主流的流程是从 `master` 分支到 `staging` 分支，再到 `production` 分支。因此，一旦 `master` 的某些变更上线了，就可以把它们合并进 `staging` 里。在生产预览服务器上通过之后，就可以把 `staging` 合并进 `production` 里。尽管逻辑上这样做很合理，我还是不喜欢它的杂乱

(到处都要合并)。而且,如果你有大量的特性需要加入 `staging` 分支并推到 `production` 分支,顺序还各不相同,那么很快就会弄得一团糟。

我认为更好一些的工作流程是从 `master` 合并到 `staging`, 并且在变更做好上线的准备后, 就从 `master` 合并到 `production`。按这种做法, `staging` 跟 `production` 关联就更少了, 你甚至可以创建多个 `staging` 分支, 在上线前同时体验不同特性(而且除了 `master` 分支, 也可以把其他分支的东西合并进这些 `staging` 分支)。一旦有什么变更得到生产批准, 你就可以把它合并进 `production` 分支了。

如果需要回滚一些变更该怎么办呢? 这时事情就变得复杂了。有多种技术可以撤销变更, 例如可以对一个提交采取反向操作, 撤销前面的提交 (`git revert`)。这些技术不仅复杂, 还会带来后续的问题。解决变更回滚的典型做法是在每次部署时创建一些版本标签(例如, 在 `production` 分支上创建标签 `v1.2.0: git tag v1.2.0`)。如果需要回滚到特定的版本, 总有标签可以使用。

Git 的工作流程最终还是由你和你的团队来决定。比起工作流程的选定, 更重要的是使用时要保持一致性, 并进行与之相关的培训和沟通。



我们已经讨论过让你的静态资源(多媒体资源和文档)跟代码仓库分离的价值。基于 Git 的部署对此又提供了一个佐证。如果你的仓库里有 4GB 的多媒体数据, 仓库克隆起来就会无休无止, 而且在每台生产服务器上都会没有必要地包含这份数据的一份副本。

2. 基于 Git 的手动部署

如果还没有做好采取行动建立 CI/CD 的准备, 可以先从基于 Git 的手动部署开始。这样做的好处是, 可以逐渐熟悉部署相关的步骤和挑战, 当走到自动化这一步时, 这些会给你带来很大的帮助。

部署每一台服务器都需要克隆代码仓库、检出 `production` 分支, 然后建立起启动 / 重启应用的必要设施(这些会依赖于你所选择的平台)。当你更新了 `production` 分支, 就需要进入每一台服务器, 运行 `git pull --ff-only`, 再运行 `npm install --production`, 然后重启应用。如果你的部署并不频繁, 服务器并不多, 这或许不是什么困难。但如果更新得比较频繁, 这些操作很快就没有新鲜感了, 然后你会寻求某种自动化的方案。



`git pull` 命令的 `--ff-only` 参数只允许快进式拉取, 避免自动合并或 `rebase`(衍合)。如果你知道这次拉取只是快进式的, 就可以放心地省略这个参数。但如果你养成了加上这个参数的习惯, 就永远不会意外地导致代码合并或 `rebase`。

本质而言，这里所做的只是重复了你在开发中所做的工作，只不过是在一台远程服务器上做罢了。手动的过程总会有人为错误的风险，所以我推荐只把手动部署作为一个中间台阶，以此为基础走向更加自动化的开发。

21.2 小结

部署网站应当是激动人心的（尤其是第一次），此时与你相伴的应当是香槟和欢呼声。然而，冷汗直冒、咒骂声以及直至深夜的加班却更为常见。我见过太多直到半夜 3 点才上线的网站，搞得每个团队成员都暴躁不已且疲惫不堪。幸运的是，多亏了云部署，这种情况正在改观。

不管选择什么样的部署策略，需要做的最重要的事情都是尽早开始生产部署，而无须等到一切皆已就绪的时候。不必把域名关联上，普通大众也不需要知道这个网站。到了网站发布的日子，如果你已经多次将网站部署到生产服务器了，那么网站发布成功的可能性就要高很多。理想的情况是，早在发布之前，网站就已经在生产服务器上正常运行了，而你只需轻按开关，旧网站就切换到新网站了。

第22章

维护

网站已经发布了！祝贺你，永远不用再想它了。什么？还得继续想着它？好吧，要是那样的话，请继续读下去。

尽管在我的职业生涯中遇到过几个“一劳永逸”的网站，但例外总是会有的（即使你真的不用再去管它，通常也是因为其他人在管，而不是真的不用管）。我清楚地记得有一个网站开了“终结”会议。我大声地说：“我们不是应该叫它产后会议吗？”¹发布网站是一次新生，而不是终结。网站一发布，你就会盯着流量分析工具，紧张地等待用户的反应，还会半夜3点起床查看网站是否依然正常。网站就是你的“宝宝”。

描绘网站、设计网站以及构建网站，这些活动都可以规划得非常长远。但是对网站的维护规划常常被忽视。就这个方面，本章将提出一些建议。

22.1 维护的原则

22.1.1 长远规划

我总是很奇怪为什么客户在同意出资构建一个网站时，对于“期望这个网站运行多久”这个问题绝不提及。我的经验是，如果工作完成得很好，客户会很乐意为此买单，但他们不喜欢额外的支出：过了3年你告诉他们网站需要重建，而他们原本期望网站至少运行5年，只是当时没有说出来。

注1：实际上，产后这个词有点儿太直接了。现在叫它回顾会议。

互联网技术日新月异。如果你使用所能找到的最好、最新的技术构建了一个网站，两年之后你就会感觉网站过时。或许它能坚持 7 年，虽然在老化，但是平缓而优雅（这种情况不常见）。

给网站寿命设定预期涉及技巧、推销和科学。其中的科学部分涉及所有科学家都会做、但 Web 开发人员很少会做的工作：做记录。假想你拥有这些数据：你的团队发布过的每一个网站、每一次维护请求和网站故障的历史记录、网站所采用的技术以及每个网站坚持了多久才开始重建。显然，变量太多了，从涉及的团队成员、社会经济形势，再到技术潮流的变迁，但并不意味着不能从这些数据中发现有意义的趋势。你可能会发现某些开发方法或某些平台或技术更适合于你的团队。我几乎可以肯定你会发现“拖延”跟缺陷的相关性。如果基础设施或平台已经给你带来了痛苦，那么越是推迟对它的更新或升级，后果就越严重。如果有一个好的问题跟踪系统，并且坚持一丝不苟地记录问题，那么对于客户项目的生命周期，你就可以给他们提供一个好得多（也更切合实际）的前景。

设定预期寿命的推销部分自然可以归结到钱的问题。如果客户负担得起每 3 年重建一次的费用，就不太可能会受到基础设施老化的困扰（不过也会遇到其他的问题）。另外，也有一些客户需要把钢镚儿一个掰成两半使，并且希望他们的网站能够支撑 5 年甚至 7 年。（我知道有的网站支撑的时间甚至还要更长，但我觉得，如果希望网站还能继续使用，从实际来说，7 年就是极限了。）你需要对这两种客户负责，他们也有各自的挑战。对于资金宽裕的客户，不能因为他们有钱你就毫不客气，你需要用额外的钱给他们带来更不寻常的东西；对于预算紧张的客户，你要找出创造性的办法，为他们设计出生命周期更长的网站来面对不断变迁的技术潮流。这两种极端情形有自己的挑战，但都可以解决。重要的是需要知道客户的期望是什么。

最后，还有设定预期寿命的技巧部分。正是这部分把各方面都联系了起来。你需要理解什么东西是客户负担得起的，以及通过坦诚交流来说服客户在什么地方应多花点儿钱来获得更大价值。你还需要理解技术的前景，并且能够预测什么技术会在 5 年后黯然退场，什么技术会愈加强大，这也是一种技巧。

当然，我们没法绝对地预言什么东西。在技术上你可能会下错注，人员变动可能会完全改变组织的技术文化，技术的提供商也可能会停业（不过在开源世界，这通常不是什么问题）。你以为自己所选择的技术在产品的整个生命周期中都会地位稳固，结果却是昙花一现而已，然后你就要比预期更早面临是否要重建这个问题。反过来说，有时候在合适的时间，汇聚合适的团队并选择合适的技术做出来的东西，其生命周期比任何理论预期都要长。然而，尽管有这些不确定因素，你也应该有一个计划，就算计划会出偏差，也比一头莽撞地干要好。

我认为已经将自己的观点说得够清楚了：JavaScript 和 Node 还会存在相当长的时间。Node 社区生机勃勃，人们友好热情，而且 Node 是基于一种已经明显取得成功的语言，这是很

明智的。最重要的是，JavaScript 是一种多范式语言：面向对象、函数式、过程式、同步、异步——都包括了。多范式语言使得 JavaScript 对于来自不同背景的开发者来说，都是很有吸引力的平台，而且在 JavaScript 生态系统的变革中，它也起了很大的促进作用。

22.1.2 使用源代码控制

很可能这对你来说显而易见，但这里不是要说使用源代码控制，而是要说使用好它。你使用源代码控制的理由是什么？理解这些理由，并确保你的工具支持这些理由。使用源代码控制的理由有很多，但我总觉得最大的好处就是责任人追踪。你可以清楚地知道，源代码都做了哪些变更，是谁在什么时候做的这些变更，在必要的时候也就知道找谁去询问了。版本控制对于理解项目历史、团队文化来说是一个非常好的工具。

22.1.3 使用问题跟踪系统

问题跟踪系统回到开发的科学方面。要是没有系统性的方法记录一个项目的历史，想要深入透视它是不可能的。你可能也听到过这个说法，精神失常就是“一遍又一遍地做着同样的事而期望不同的结果”。一次又一次地重复着你的错误，看起来的确是很蠢，但要是不知道自己犯了什么错误的话，又如何能够避免它呢？

任何东西都要记录，哪怕是客户报告的一个缺陷、在客户看见之前你自己发现的一个缺陷、用户的一次抱怨、一次提问，抑或是一星半点儿的表扬。记录一个缺陷修复所花的时间、谁修复的它、涉及哪些 Git 提交，以及谁批准了这次修复。此处的技巧就是要寻找既能节省时间又不容易出错的工具。一个很差的问题跟踪系统使用起来既痛苦又没有好处，甚至比无用的系统还要糟糕。一个很好的问题跟踪系统能够带来对于业务、团队以及客户来说至关重要的深刻见解。

22.1.4 保持良好的“卫生习惯”

我说的不是刷牙这样的卫生习惯，而是版本控制、测试、代码评审和问题跟踪。你所使用的工具只有在正确使用时才有用。代码评审是提升“卫生习惯”的很好的方法，因为在代码评审中，从讨论问题跟踪系统的使用（这是代码评审发起的地方），到为了验证缺陷修正而必须加入的测试，再到版本控制的代码提交的评论，各个方面都涉及了。

你从问题跟踪系统收集到的数据应该定期进行评审，并跟团队讨论。从这些数据中，你可以深入了解系统中哪些部分工作正常、哪些部分工作不正常。你可能会有出乎意料的发现。

22.1.5 不要拖延

习惯性的拖延也许是最难克服的一件事情。通常这件事情并没有那么糟糕：你的团队在每周更新上总是要耗上很多个小时，而你会注意到只要做一点点重构，情况就会得到显著的

改观。如果有一周推迟了重构，其代价则是下一周低效率的工作。²更糟糕的是，有些成本会随着时间推移而增加。

依赖软件的更新就是一个很好的例子。随着依赖的软件愈加老化以及团队成员的变更，很难找到能够记得（或理解）这个“老掉牙”的软件的合适人选了。这个软件的支持社区在慢慢消失，不久之后，这项技术就会被淘汰，就再也找不到任何类型的支持了。你常常会听到把这个形容为**技术债**，这是真真切切的。在避免拖延的同时，也要了解这个网站的寿命，因为网站的寿命是你做决策的一个考虑因素。如果你正要重新设计整个网站，消除目前的这些技术债就没什么价值了。

22.1.6 例行QA核查

对你的每一个网站，你都应该有一个**文档化**的例行 QA 核查。这个核查应当包含超链接检查、HTML 与 CSS 验证，以及运行测试。这里的关键在于文档化。如果组成 QA 核查的条目并没有相应的文档，就不可避免地会漏掉一些东西。一个网站的**文档化**的 QA 核查清单不仅有助于避免忽视一些核查，还可以让新团队成员立即进入高效率的状态。理想情况下，QA 核查清单应当由团队的非技术成员来执行。这样可以给你的（有可能是）非技术背景的经理以信心，并且如果你们没有一个专门的 QA 部门，这样也有助于把 QA 的责任分散开。根据你们跟客户的关系，可以选择是否与他们分享 QA 核对清单（或者其中的一部分）。这个做法很好，可以提醒他们是在为了什么而买单，而你们一直在以他们的最大利益为重。

作为例行 QA 核查的一部分，我推荐使用谷歌的 Webmaster 工具和 Bing 的 Webmaster 工具。它们都很容易设置，也给了你一个看待网站的非常重要的视角，即主流搜索引擎的视角。如果你的 robots.txt 文件有什么问题，或者 HTML 有什么影响到好的搜索结果的问题，或者安全性问题等，它就会发出警告。

22.1.7 监控分析

如果你还没有在网站上运行分析，现在就要开始了。分析让你能够深入洞察，不仅是洞察网站的受欢迎程度，还有你的用户使用这个网站的方式。Google Analytics (GA) 是非常出色的（而且免费），即使你还有其他的分析服务，也没有什么理由不把它包含进你的网站里。

很多时候，只要你留意一下分析服务，就能够发现微妙的用户交互问题。是否有一些页面没有达到预期的访问量？这可能暗示着导航问题、产品推广问题，或者 SEO 问题。你的用户跳出率是否很高？这可能暗示着页面的内容需要做一些裁剪（用户是通过搜索来到你的页面的，可是到了以后，才意识到这不是他们要找的）。除了 QA 核查清单，你还应该有

注 2：Fuel 公司的 Mike Wilson 的经验法则：“某件事你做到第三次时，就花点儿时间把它自动化了。”

一个网站分析核查清单（它甚至可以成为 QA 核查清单的一部分）。这个核查清单应该是一个“活文档”，因为在网站的生命周期中，你或你的客户可能会不断地调整清单中条目的优先级来体现什么内容最重要。

22.1.8 优化性能

一个又一个研究表明，网站的性能对流量有显著的影响。这是一个快节奏的世界，人们都期望他们的内容快速地到达，尤其是在移动平台上。在性能调优中，首要的原则是先剖析，再优化。“剖析”意味着找出到底是什么实际拖慢了你的网站。如果你花了数天时间来加快内容渲染，结果问题却是出在社交媒体插件上，那就浪费了宝贵的时间和金钱。

谷歌的 PageSpeed Insights 是非常好的衡量网站性能的方法（而且现在 PageSpeed 的数据是记录在谷歌的 Analytics 里的，因此你可以监控到网站的性能趋势）。它不仅会针对移动和桌面给出总体的性能评分，而且提出了按优先级排序提升性能的建议。

除非你当前就有性能问题，否则周期性的性能核查（监控 Google Analytics，看看性能评分有没有显著的变化，这应该就够了）可能是不必要的。不过，在提升性能之后，就能看到网站的流量增长，还是很让人高兴的。

22.1.9 优先跟踪潜在客户

在互联网世界，访问者如果对你的产品或服务感兴趣，能给你的最强信号就是联系信息。你应当十分认真地对待这些信息。收集邮箱地址或电话号码的表单应当经过例行的测试，而这个测试应当成为 QA 核对清单的一个部分。而且在收集信息的时候，总是应当有一个补充的存储方案。最糟糕的事情莫过于，你收集到了一个潜在客户的联系信息却又把它弄丢了。

意向客户追踪对网站的成功至关重要，既然如此，我推荐以下 5 条收集信息的原则。

准备一个回退方案，以免 JavaScript 失效

通过 Ajax 来收集客户信息很不错，它的用户体验往往更好。可是如果不管是什么原因导致 JavaScript 失败了（用户可以禁用它，或者你的网站的某个脚本存在错误，导致 Ajax 表单不能正常工作），表单提交都应该正常。一个很好的测试方式是禁用 JavaScript，使用表单来提交。用户体验不够理想没关系，重点是用户数据没有丢失。为了能回退到表单提交，即使正常使用的是 Ajax，也始终给 `<form>` 标签设置一个正确的 `action` 参数。

如果使用 Ajax，从表单的 `action` 参数中取出 URL

虽然严格来说并不必要，但这样做可以防止你无意中忘了 `<form>` 标签的 `action` 参数。如果 Ajax 所关联的表单本身在没有 JavaScript 的情况下也能正常提交，就不容易丢失客

户的数据了。比如说你的表单标签是 `<form action="/submit/email" method="POST">`, 然后在 Ajax 代码里, 通过 DOM 操作可以从表单中获得 `action`, 并在 Ajax 表单提交的代码中使用。

至少提供一层冗余

很可能你会想把意向客户的信息保存到一个数据库或者是像 Campaign Monitor 这样的外部服务中去。但如果数据库出现了故障, 或是 Campaign Monitor 服务停机了, 或是出现了网络问题, 那么该怎么办呢? 如果你不想丢失这些信息, 就需要增加冗余。存储信息的同时发送一封邮件, 这是增加冗余的常见办法。如果这样做, 就不要使用个人邮箱, 而应该使用共享邮箱 (比如 `dev@meadowlarktravel.com`)。如果你是把邮件发送给某一个人, 而这个人离开了你们组织, 那么这个信息冗余就没有什么用处了。你也可以把意向客户的信息存储到备用数据库, 甚至 CSV 文件里。不过, 不管任何时候你的主存储出现了故障, 都应该有某种警告机制。而从冗余存储里收集信息只是事情的一半, 意识到故障并采取适当的措施是事情的另一半。

万一存储全部出了问题, 要通知用户

比如说你有 3 层冗余: 你的主存储是 Campaign Monitor 服务, 如果它失败了, 就保存到 CSV 文件, 并发送一封邮件到 `dev@meadowlarktravel.com`。如果所有这些存储通道都失败了, 那么用户应该能收到一条类似这样的消息: “很抱歉, 我们遇到了技术问题。请过一会儿再试, 或者联系 `support@meadowlarktravel.com`。”

要看到成功确认, 而不是没看到错误就算成功

发生故障的时候, 你的 Ajax 后端程序会返回一个对象, 其中包含一个 `err` 属性, 然后客户端代码这样写: `if(data.err){ /* 通知用户发生了故障 */ } else { /* 提交成功, 感谢用户 */ }`。虽然很常见, 但要避免这种做法。设置一个 `err` 属性并没有错, 但如果你的 Ajax 后端程序存在错误, 导致服务器返回一个 500 响应码, 或者返回的并非一个合法的 JSON, 这种做法就会不声不响地失效。这个用户的线索就这样凭空消失了, 而用户和网站方都不会知道。应该换种做法, 提供一个 `success` 属性来指示成功的提交 (即使主存储出现故障了, 如果用户的信息在别的地方得到了记录, 也可以返回 `success`)。于是客户端代码就变成了 `if(data.success){ /* 提交成功, 感谢用户 */ } else { /* 通知用户发生了故障 */ }`。

22.1.10 避免“不可见”的故障

因为开发人员太匆忙了, 所以就算他们记录了错误, 过后也不会做核查。这样的情形屡见不鲜。不管是把错误记录在一个日志文件里、数据库中的一个表里、客户端的控制台上, 或者是发送到一个无人接收的邮箱地址, 最终结果都一样。你的网站存在不为人知的质量问题。

对此，你能采取的首要措施是为记录错误提供一个易用的标准化方法，并形成文档。这个方法不要设计得太复杂，也不要太难懂。要确保每个接触到你的项目的开发人员都理解。它可以简单到仅暴露一个 `meadowlarkLog` 函数（`log` 这个名称常常被其他包使用）。至于这个函数是记录到数据库里、普通文件里、电子邮件里，还是它们的某种组合里都没关系，重要的是它必须是标准化的。标准化也让你得以提升日志机制（例如，对服务器做横向扩展时，普通文件就没那么有用了，于是你就可以修改 `meadowlarkLog` 函数，让它记录到数据库里）。一旦有了这个日志机制、已经文档化了，而且团队中每个人也都了解它，就可以把“检查日志”加入 QA 核对清单中，并且针对如何检查日志设置相应的步骤。

22.2 代码重用与重构

浪费时间一次又一次地做无用功，这是我时常看到的悲剧。通常这些“无用功”都是很小的东西，比如那些与其从数月前的某个项目中挖出一段还不如重新写来得容易的代码。这些重写的代码片段会越积越多。更糟糕的是，它破坏了 QA 的原则，因为你很可能就不打算去为这些小片段写测试了（本来没有重用已有的代码就已经浪费了时间，如果写了测试，就相当于浪费双倍时间）。即使做的事情一样，每个代码片段也可能有不同的 bug。这是一个坏习惯。

在 Node 和 Express 的开发中，有一些非常好的机制，可以克服浪费时间做无用功这个问题。Node 带来了命名空间（通过模块）和包（通过 npm）的机制，而 Express 带来了中间件的概念。运用这些工具，开发可重用的代码就容易多了。

22.2.1 私有npm仓库

npm 仓库是存储共享代码的好地方，毕竟这正是 npm 设计的初衷。除了简单的存储，你还获得了版本管理的功能，以及在其他项目中引入仓库中的包的便利。

不过也有“美中不足”。除非你是工作在一个完全开源的组织，否则可能不会想为所有可重用的代码都创建 npm 包。（除知识产权保护以外，可能还有别的原因。如果你的包是特定于组织 / 项目的，把它放到公共的仓库中就没有意义。）

对此的解决办法就是私有 npm 仓库。npm 现在提供了面向组织的服务 Orgs，让你可以发布私有包，付费让开发者登录，允许他们访问这些私有包。关于 Orgs 和私有包的更多信息，请查看 npm 网站的产品页。

22.2.2 中间件

正如在本书中看到的，写中间件并不是一件巨大的、复杂的、让人犯怵的事情。在本书中我们已经写过几十次了，而且用不了多久，你就可以不假思索地写下一个中间件了。然后

下一步，就是把可重用的中间件放入一个包里，并把这个包放入 npm 仓库中。

如果你觉得你的中间件跟项目过于相关，不适合放入一个可重用的包里，就可以考虑把这个中间件重构一下，使其配置得更加通用。别忘了可以在中间件中传入配置对象，让它适用于各种情形。有多种方式能在一个 Node 模块中把中间件暴露出来，下面概述一下最常见的几种。所有示例都假设你把这些模块用作一个名为 meadowlark-stuff 的包。

1. 模块直接暴露了中间件函数

如果你的中间件不需要传入配置对象，就使用以下这种方式：

```
module.exports = (req, res, next) => {
  // 这里就是你的中间件……记得调用next() 或next('route'),
  // 除非你的中间件要作为一个端点
  next()
}
```

使用这个中间件：

```
const stuff = require('meadowlark-stuff')

app.use(stuff)
```

2. 模块暴露了一个返回中间件的函数

如果你的中间件需要传入配置对象或其他信息，就使用以下这种方式：

```
module.exports = config => {
  // 如果没有传入配置对象，就创建一个，这很常见：
  if(!config) config = {}

  return (req, res, next) => {
    // 这里就是你的中间件……记得调用next() 或next('route'),
    // 除非你的中间件要作为一个端点
    next()
  }
}
```

使用这个中间件：

```
const stuff = require('meadowlark-stuff')({ option: 'my choice' })

app.use(stuff)
```

3. 模块暴露了一个包含中间件的对象

如果你想暴露多个相关的中间件，就使用以下这种方式：

```
module.exports = config => {
  // 如果没有传入配置对象，就创建一个，这很常见：
  if(!config) config = {}
```

```
    return {
      m1: (req, res, next) => {
        // 这里就是你的中间件……记得调用next() 或next('route'),
        // 除非你的中间件要作为一个端点
        next()
      },
      m2: (req, res, next) => {
        next()
      },
    },
}
```

使用这个中间件：

```
const stuff = require('meadowlark-stuff')({ option: 'my choice' })

app.use(stuff.m1)
app.use(stuff.m2)
```

22.3 小结

构建一个网站时，主要关注的往往是网站的上线发布。这也情有可原，毕竟网站发布的前前后后太让人激动了。然而，如果网站疏于维护，一个刚刚还在为这个新网站的发布而欢喜的用户，也许很快就会变成一个不满的客户。制订好维护计划，像关注发布一样关注维护，就能够保持网站的良好体验，从而留住你的用户。

第23章

更多资源

关于如何使用 Express 来构建网站，本书已经做了全面的介绍。也讨论了多个领域的内容，但可以使用的各种包、技术和框架非常多，我们所触及的只是九牛一毛。本章将讨论可以从哪里获取更多的资源。

23.1 在线文档

就 JavaScript、CSS 和 HTML 的文档来说，Mozilla 开发者网络（MDN）无可匹敌。如果需要 JavaScript 的文档，那么或者直接在 MDN 上搜索，或者在搜索引擎上加上“mdn”关键字。如不加“mdn”关键字，不出意料 w3schools 就会出现在搜索结果中。为 w3schools 做 SEO 的人真是个天才，但我建议你别看这个网站，因为我发现它严重缺乏文档。

MDN 是非常好的 HTML 参考，但如果你是 HTML5 新手（即便不是），应该读一读 Mark Pilgrim 的“Dive Into HTML5”。WHATWG 维护着一个出色的 HTML5 规范“活标准”，通常遇到确实很难回答的 HTML 问题时，我就会求助于它。最后是 HTML 与 CSS 的正式规范，它们位于 W3C 网站上，十分晦涩难懂，但面对那些最棘手的问题时，W3C 网站有时是你唯一的资源。

JavaScript 遵循着 ECMA-262 ECMAScript 语言规范。要想追踪在 Node（以及各种浏览器）中各种 JavaScript 特性的可用性，可以查看由 @kangax 维护的出色的指南。

Node 的官方文档非常好，十分全面，如果要找 Node 模块（比如 http、https 和 fs）的权威文档，它应该是你的第一选择。Express 的官方文档也很好，但可能没有你期望的那么全面。npm 的官方文档既全面又好用。

23.2 期刊

下面 3 个免费期刊绝对是你应该订阅的，并且每周都应该好好读一读：

- *JavaScript Weekly*
- *Node Weekly*
- *HTML5 Weekly*

这 3 个期刊可以帮助你了解最新的技术动态、服务、博客和教程，只要它们一出来你就知道。

23.3 Stack Overflow

你很可能已经在使用 Stack Overflow (SO) 了。这个网站始于 2008 年，已经成了统治性的在线问答网站，如果你有 JavaScript、Node 和 Express（以及本书中提到的任何其他技术）方面的问题，它就是最好的资源。SO 是由社区维护的基于个人声誉的问答网站。这个网站的内容质量以及它的持续成功依靠的正是其声誉模型。如果用户的提问或回答得到“顶”，或者回答被采纳，就会获得声誉。在 SO 上不是先有声誉才能提问题，它的注册也是免费的。然而，你可以做一些事情来提高问题回答的质量，本节将对此进行讨论。

声誉是 SO 的通货。虽然说很多人真诚地希望能帮助你，但随之而来的声誉，未尝不是得到一个好回答的因素。在 SO 上有很多很聪明的人，他们都在为你的提问提供第一个 / 最佳的回答而竞争（幸而，这个网站对于快速但很差的回答有很强的控制机制）。为了提高问题回答的质量，以下是可以做的事情。

做一个明白的 SO 用户

看一看 SO 的导览，然后读一读 “How do I ask a good question?”。如果你比较赞同，可以继续完整地读一遍帮助文档——如果全部读完，你就会获得一枚徽章。

不要重复提问已经回答过的问题

先做一些应有的调查，努力找找看有没有人已经问过你提的问题了。如果你提的问题已经在 SO 上有了答案并且可以轻易找到，你的提问很快就会因为重复而被关闭，而人们常常会因此给你点“踩”，这会对你的声誉产生负面影响。

不要请求别人帮你写代码

如果只是简单地问 “某件事我应该怎么做？”，很快你会发现你的问题被“踩”了，然后被关闭了。SO 社区希望你能先努力试着解决自己的问题，最后再求助 SO。在提问中描述清楚你都做过什么尝试以及为什么无效了。

一次提一个问题

不要一连问好几个事情。像“这事我要怎么做呢？那个呢？另外一个呢？这些事最好的做法是什么？”这样的问题很难回答，因而并不鼓励提问。

为你的问题制作一个最简短的例子

我回答过大量的 SO 提问，对于那些动辄有 3 页长（或更多）的代码的问题，几乎都是自动跳过的。仅仅打开一个 5000 行的文件把代码粘贴进 SO 提问里，不是让你的提问得到回答的好办法（可人们还总是会这么做）。这是懒惰的做法，通常是不会得到回应的。这么做不太可能会得到有用的回答，而且剔除那些与问题无关的东西的过程正是解决问题的过程，也许你自己就把这个问题给解决了（甚至都不需要再去 SO 提问）。制作一个最简短的例子，有助于提升调试技能、批判性思维能力，也有助于你成为一个 SO 好公民。

学习 Markdown

Stack Overflow 使用 Markdown 来为提问和回答排版。一个格式良好的提问得到回答的可能性会更高，因此你应该投入一点儿时间，学习一下 Markdown 这个很有用的、应用越来越广泛的标记语言。

“顶”和接受回答

如果有谁的答案很令人满意，你应该给这个回答点“顶”并接受它，这能提升回答者的声誉，也正是声誉在驱动着 SO 运转。如果有多人都提供了可接受的回答，就应该选出你认为最好的并接受它，对于其他觉得有用的回答，都给它们点“顶”。

如果在其他人之前想出答案了，就回答你自己的问题

SO 是一个社区资源，如果你有一个问题，很有可能别人也会有。如果你想出答案了，为了帮助他人，就去回答你自己的问题吧。

如果能从帮助社区中感到快乐，那么考虑自己去回答问题。回答问题既有趣又有收获，得到的好处比那些随意的声誉值实际多了。如果你的提问过了两天还没得到有用的回答，就可以使用自己的声誉值为这个提问发起悬赏。声誉值会直接从你的账户上扣除，且不会退还。如果有人回答了你的提问并且能让你满意，就接受这个回答，而他也会得到这份奖金。当然，你必须有声誉值（最少 50 点声誉）才能发起悬赏。虽然可以从高质量的提问中获得声誉，但是通过提供高质量的回答获得声誉，通常要快得多。

回答别人的问题对自己来说也是非常好的学习过程，这也是回答问题的一个好处。比起自己的问题得到回答，我常常感觉回答别人的问题学到的东西要更多。如果你希望透彻地学习某个技术，在学习了基本概念之后，就可以试着解决一下 SO 上别人的问题。最初你也许会一次又一次地落后于那些已经成为专家的用户，但用不了多久，不知不觉间你就成为那些专家中的一员了。

最后，不必犹豫，你可以携着你的声誉继续职业道路。一个好的声誉绝对值得写进简历里。我就曾把它写进简历里，并且行之有效。现在我亲自面试开发者，对良好的 SO 声誉总是印象深刻（声誉值超过 3000 我就认为是好的，有 5 位数那就是极好的了）。一个好的 SO 声誉告诉我，这个人不仅擅长这个领域，还善于沟通，乐于助人。

23.4 对Express做贡献

Express 和 Connect 都是开源项目，任何人都可以给它们提 pull request (PR, GitHub 的用词，表示你做的那些希望包含进项目里的变更)。这不容易，因为工作在这些项目上的开发者都是项目本身的专家和最高权威。并不是说不鼓励你做贡献，而是说要成为成功的贡献者，必须倾注相当多的精力，而且不能轻率地提交。

如何对项目做贡献，Express 官网的文档已经写得很清楚了。过程包括：把项目 fork 到你自己的 GitHub 账号里、把 fork 出来的项目复制到本地、做自己的变更、把变更推回 GitHub，以及创建一个 PR，这个 PR 会被项目的一个或多个成员评审。如果你的提交比较小或者只是 bug 修复，也许只要提交 PR 就可以了。如果你想做比较大的变更，就需要先跟主要开发者沟通并阐述你要做的东西。你并不希望看到花了数个小时或数天时间实现的一个复杂特性跟项目维护者的规划不符，或者已经有人在做了。

为 Express 和 Connect 的开发做贡献的另一种方式（间接的）是发布 npm 包——具体来说，就是中间件。发布你自己的中间件不需要任何人的批准，但这并不意味着就可以随意用低质量的中间件来扰乱 npm 仓库。只要将规划、测试、实现以及文档这几个环节都做好，你的中间件就会获得更大的成功。

如果你要发布自己的包，最少需准备以下这些东西。

包名

虽然说包名由你来决定，但显然不能选用已经被占用的包名，因此有时候选包名就成了一个挑战。npm 包现在支持以账号作为名空间，因此你就不需要满世界争抢包名了。如果写的是中间件，按照习惯，就应该在你的包名前面加上 `connect-` 或 `express-` 前缀。如果从一个包名看不出它是做什么的，但是很易记，那么也不是不可以，但如果能够体现它的功能，就更好了（易记而又很适合的包名，`zombie` 就是一个非常好的例子，用于无头浏览器的模拟）。

包描述

包描述应当简明、准确。当人们搜索包时，包描述是主要的几个索引字段之一，所以最好提供描述性的信息，而不是精巧的语言（别担心，在你的文档里有你展现精巧和幽默的地方）。

作者 / 贡献者

这是你该得到的荣誉。

许可

这个常常被忽视。没有比找到一个包却发现并无许可更让人沮丧的了（你不确定是否能在自己的项目中使用它）。不要做那样的人。如果不想对代码的使用加以任何限制，MIT 许可就是最简单的选择。如果你希望代码开源（并保持下去），那么 GPL 许可是另一个流行的选择。把许可文件包含在项目的根目录下很明智（文件名应该以 LICENSE 开头）。要想让最多的人使用，就选择 MIT 和 GPL 双许可。关于 package.json 和 LICENSE 文件的示例，可以看看我的 `connect-bundle` 包。

版本

为了让版本系统正常工作，你需要给自己的包打上版本号。要注意，npm 的版本化跟代码仓库的提交号是分离的，所以你想怎样更新代码仓库都可以，因为用户使用 npm 安装你的包所获取到的内容并不会受影响。要把更新体现到 npm 仓库中，就需要升级版本号，然后重新发布。

依赖

在包的依赖上应该努力做到保守一些。我不是建议不断地浪费时间做无用功，但是依赖增加了包的大小和许可的复杂性。至少，你应该确认自己有没有把不需要的依赖加进去。

关键字

跟包描述一样，用户在查找你的包时，关键字也是要使用的主要元数据，因此要选择适当的关键字。

代码仓库

应该有一个仓库。GitHub 是最常用的，但另外几个也很受欢迎。

README.md

GitHub 和 npm 的标准文档格式都是 Markdown。Markdown 语法很简单，类似于 wiki，你很快就能掌握。如果想让别人使用你的包，那么高质量的文档极其重要。如果我打开了一个 npm 包却没有文档，一般就会跳过了，不会继续研究。至少，你应该描述一下基本的用法（带上示例）。要是所有的选项参数都有说明，那就更好了，甚至可以描述一下如何运行测试。

只要做好了发布包的准备，发布过程就很简单了。注册一个免费的 npm 账号，然后按以下步骤进行。

1. 键入 `npm adduser`，用你的 npm 身份登录。
2. 键入 `npm publish`，发布你的包。

就这么简单。现在可以从头创建一个项目，然后使用 `npm install` 来测试你发布的包了。

23.5 小结

Express 是个令人激动的技术栈，我真诚地希望本书给你带来了所有必要的工具，让你可以开始自己新的技术之旅。在我的职业生涯中，还从来没有为一项新技术而振奋到如此地步（尽管这奇怪的主角是 JavaScript），希望我能向你传递一些这个技术栈的优雅和前景。虽然多年来我一直都在做专业网站构建的工作，但多亏了 Node 和 Express，使我感觉比起以前，对互联网运行方式的理解又深入了一个层次。我相信它是能真正增进你对互联网运行方式理解的技术，而不是试图把各种细节都隐藏起来，况且它还为你提供了快速而高效地构建网站的框架。

不管你是一名 Web 开发的新手，或者仅是 Node 和 Express 的新手，都欢迎你进入 JavaScript 开发者的行列。我期盼着能在用户组和研讨会上看到你，更重要的是，能看到你做出来的东西。

关于作者

伊桑·布朗 (Ethan Brown)，美国 VMS 公司技术总监，负责公司旗舰产品的架构和实现。他拥有 20 多年的编程经验，在 Node、Express、JavaScript、React 的实践方面拥有独到见解。

关于封面

本书封面上的动物是一只黑百灵 (*Melanocorypha yeltoniensis*) 和一只白翅百灵 (*Melanocorypha leucoptera*)。这两种鸟都是部分迁徙性的，它们以活动范围极其广阔而著称，从哈萨克斯坦的草原到俄罗斯中部，都是它们最适合的栖息地。除了在哈萨克草原上哺育后代，雄性黑百灵还会在那里过冬，雌性黑百灵则会往南迁徙。而在冬季的几个月里，白翅百灵会一直飞到黑海以西和以北更远的地方。在全球范围内，这些鸟的足迹还要更远：对白翅百灵来说，欧洲包含其全球活动范围的 1/4 到一半；对黑百灵来说，欧洲只包含其全球活动范围的 5% 到 1/4。

黑百灵之所以如此命名，是因为它的雄鸟几乎全身覆盖着黑色。雌鸟则不同，除了黑色的腿和翅下的黑色羽毛跟雄鸟相像以外，它的其余部分是黑色和灰白色相间。

白翅百灵的翅膀羽毛上有黑色、白色和红棕色组成独特图案。白翅百灵背部的灰色条纹与其肚子上的灰白色相互映衬。在外观上，雄鸟跟雌鸟仅有的差异是雄鸟有红棕色的鸟冠。

黑百灵和白翅百灵的叫声都很动听，在百灵所有的种类当中，它们最受作家和音乐家们喜爱，多个世纪以来一直如此。两种鸟都吃昆虫和种子，它们的雏鸟也是如此。两种鸟都会在地面上筑巢。黑百灵也会往巢里搬粪球，用来“筑墙”或者“铺地板”之类的，不过这种行为的原因还未找到。

O'Reilly 图书封面上的许多动物濒临灭绝，它们对世界很重要。

封面图为 Karen Montgomery 所作，以 Lydekker 的 *The Royal Natural History* 的一个黑白雕刻为底板。



微信连接



回复“Web”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I: 218139230

图灵读者官方群II: 164939616

图灵社区 iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈



Node与Express开发 (第2版)

Express是Node/JavaScript技术栈的一个关键组件，可用于高效构建动态Web应用。正所谓“少即是多”，Express以极简的框架赋予Web应用高性能、灵活、健壮等优点。

本书将带你体验构建Web应用的每一步，并在此过程中阐释Express的诸多概念和原理。第2版着重展示Express在构建单页应用方面的潜力，以及近几年来Express中间件和相关工具的发展变化。快快拿起本书，一起探索Web开发的新思路吧！

- 为渲染动态数据创建模板系统
- 深入探究request和response对象、中间件以及URL路由
- 为测试模拟生产环境
- 数据持久化，涉及MongoDB和PostgreSQL
- 开放API，让其他程序可以访问你的资源
- 构建包含认证、授权并支持HTTPS的安全应用
- 集成社交媒体、地理位置及其他功能
- 实现应用的启动和维护计划
- 学习重要的调试技能

“伊桑的过人之处在于，他并不假定读者知道什么或不知道什么。我在这本书中欣喜地看到，他既出色地介绍了Node/Express生态系统，又特地为初学者阐明了诸如持久化、中间件、Git等Web开发的相关概念。”

——Alejandra Olvera-Novack
AWS开发者事务部

伊桑·布朗 (Ethan Brown)，美国VMS公司技术总监，负责公司旗舰产品的架构和实现。他拥有20多年的编程经验，在Node、Express、JavaScript、React的实践方面拥有独到见解。

WEB PROGRAMMING / JAVASCRIPT

封面设计：Karen Montgomery 张健

图灵社区：[iTuring.cn](#)

分类建议 计算机/Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社有限公司出版

此简体中文版仅限于在中华人民共和国境内（不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of the People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-56509-9



扫码领取
随书代码资料



9 787115 565099 >

定价：109.80元