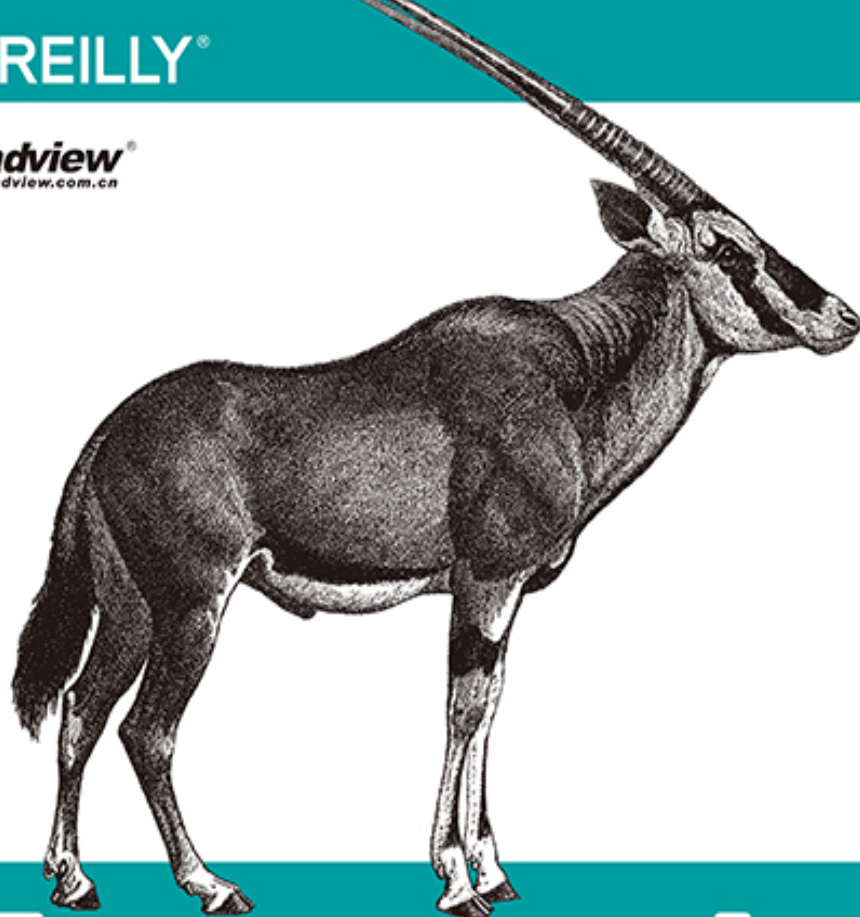


O'REILLY®

Broadview®
www.broadview.com.cn



Laravel

入门与实战

构建主流PHP应用开发框架

Laravel: Up and Running

[美] Matt Stauffer 著
韦玮 译



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

O'REILLY®

Laravel入门与实战

构建主流PHP应用开发框架

Laravel: Up and Running

[美] Matt Stauffer 著
韦玮 译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书对Laravel框架进行了系统的介绍,包括Laravel的背景、Laravel开发环境的搭建、路由与控制器、Blade模板、前端组件、收集和处理用户数据、Artisan和Tinker、数据库和Eloquent、容器、Laravel测试等知识。通过阅读本书,读者可以比较全面地学习并掌握Laravel开发的相关理论知识。另外,书中涵盖大量实例,更有利于读者在学习过程中不断实践。

©2017 by Matt Stauffer

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2018. Authorized translation of the English edition, 2017 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

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

本书简体中文版专有出版权由O'Reilly Media, Inc. 授予电子工业出版社。未经许可,不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字:01-2017-0157

图书在版编目(CIP)数据

Laravel入门与实战:构建主流PHP应用开发框架/(美)马特·斯托弗(Matt Stauffer)著;韦玮译. —北京:电子工业出版社,2018.4

书名原文:Laravel: Up and Running

ISBN 978-7-121-33611-9

I. ①L… II. ①马… ②韦… III. ①网页制作工具—PHP语言—程序设计 IV. ①TP393.092②TP312

中国版本图书馆CIP数据核字(2018)第020315号

策划编辑:孙奇俏

责任编辑:牛 勇

封面设计:Randy Comer 张健

印 刷:

装 订:

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编:100036

开 本:787×980 1/16 印张:27.5 字数:570千字

版 次:2018年4月第1版

印 次:2018年4月第1次印刷

定 价:108.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至zlts@phei.com.cn, 盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始, O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来, 而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者, O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”; 创建第一个商业网站 (GNN); 组织了影响深远的开放源代码峰会, 以至于开源软件运动以此命名; 创立了 Make 杂志, 从而成为 DIY 革命的主要先锋; 公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖, 共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择, O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程, 每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法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、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了: ‘如果你在路上遇到岔路口, 走小路 (岔路) 。’ 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

本书献给我最亲爱的妻子 Tereva、开朗勇敢的儿子 Malachi，还有美丽的女儿 Mia，
她在妈妈的肚子里时，我完成了本书的大部分创作。

译者序

在 Web 开发方面，相信大家对 PHP 语言并不会感到陌生。是的，使用 PHP 语言实现 Web 应用开发确实非常便捷，正因如此，PHP 语言获得了大量 Web 开发者的忠实拥护，译者也是其中之一。

在开发过程中，如果能使用一款优秀的 PHP 框架将会事半功倍。目前使用 PHP 语言开发 Web 项目通常有三种方案：使用原生 PHP 代码（不基于框架）开发、基于框架开发以及基于框架并对框架进行二次开发。对大部分公司来说，开发效率这项指标是非常重要的，又因为框架实际上相当于项目的“半成品”，所以大部分公司可能更倾向于选择一款性能优越的 PHP 框架，基于框架进行具体项目的开发。

目前，可以选择的 PHP 开发框架有很多，比如 Laravel、ThinkPHP、Yii、CakePHP 等，这些框架各有千秋，但如果非要让译者在其中选择出一两款自己更喜欢的框架，那么应该是 Laravel 与 ThinkPHP。Laravel 框架性能较好，运行速度较快，适合开发中大型项目，在国际范围内都比较流行，但由于 Laravel 框架是由国外研发的，所以目前可参考文献与资源在国内都比较匮乏。ThinkPHP 框架是国内研发的，性能同样也非常好，比较适合开发中小型项目，并且国内的可参考文献与资源相对较多。这两款框架都是非常优秀的 PHP 开发框架，如果有时间，建议大家对这两款框架进行深入的研究。

这本书主要对 Laravel 框架进行了系统的介绍。包括 Laravel 背景、Laravel 开发环境的搭建、路由与控制器、Blade 模板、前端组件、收集和處理用户数据、Artisan 和 Tinker、数据库和 Eloquent、用户认证和授权、请求和响应、容器、Laravel 测试、API 编写、存储与检索、邮件和通知、任务、队列、事件、广播及调度程序、助手和集合等。上面已经提到，Laravel 框架的相关学习资料比较匮乏，因此，衷心希望这本书出版后在一定程度上能够弥补这方面的空缺。

有幸得到博文视点编辑孙奇俏老师的邀请，并在孙老师的监督下完成了本书的翻译与审校工作。但由于我公司事务繁忙，精力有限，所以邀请了我的学生黄梦颖与俞渺共同完成了本书的翻译工作。黄梦颖与俞渺都是非常优秀的高材生，都有过国外留学经历，黄梦颖同学现已获得了新加坡国立大学硕士学位，俞渺同学现已获得墨尔本大学硕士学位。

翻译过程中，我负责正文第 1 ~ 6 章翻译以及全书的审校工作，黄梦颖负责正文第 7 ~ 10 章、第 13 和 14 章、前言、封底、词汇表、作者简介、封面简介等的翻译工作，俞渺负责正文第 11 和 12 章、第 15 ~ 17 章的翻译工作。在此由衷感谢两位同学的艰苦付出，若读者在阅读过程中发现翻译文风略有不同，还请理解。

本书关于 Laravel 框架相关知识的介绍非常具体，如果你希望系统地学习并掌握 Laravel 框架，那么本书将非常适合你。当然，由于 Laravel 属于 PHP 语言众多框架中的一种，所以，大家在阅读本书之前最好先学习一些关于 PHP 的基础知识，这样会更容易理解本书的内容。

再次感谢孙奇俏老师的邀请，感谢本书编辑杨嘉媛，孙老师与杨编辑在本书的审稿与编加过程中付出了大量的精力与心血。感谢黄梦颖与俞渺对翻译做出的巨大贡献。感谢本书原作者 Matt Stauffer 在写作过程中付出的巨大努力。感谢所有一直支持我的朋友们。感谢所有 PHP 的忠实拥护者与 Laravel 爱好者，谢谢你们的支持！

韦玮

2018 年 1 月于上海

目录

前言	xxiii
第 1 章 为什么使用 Laravel.....	1
为什么要使用框架	1
“自己动手做”	2
一致性和灵活性	2
Web 和 PHP 框架简史	2
Ruby on Rails	2
PHP 框架的涌入	3
CodeIgniter 的优点与缺点	3
Laravel 1、Laravel 2 和 Laravel 3	3
Laravel 4	4
Laravel 5	4
Laravel 有什么特别之处	4
Laravel 的哲学	4
Laravel 如何让开发者快乐	5
Laravel 社区	6
Laravel 是如何工作的	6
为什么使用 Laravel	9

第 2 章 搭建 Laravel 的开发环境	11
系统要求	11
Composer	11
本地开发环境	12
Laravel Valet.....	12
Laravel Homestead	13
创建一个新的 Laravel 项目	17
使用 Laravel 安装工具来安装 Laravel	17
通过 Composer 的 create-project 来安装 Laravel.....	17
Laravel 的目录结构.....	18
文件夹	18
文件	19
配置	19
启动和运行	20
测试	21
本章小结.....	21
 第 3 章 路由和控制器	23
路由定义	23
路由动词 (Verb)	25
路由处理	25
路由参数	26
路由名称	27
路由组.....	30
中间件	30
路径前缀	31
子域名路由	32
命名空间前缀.....	33
名称前缀	33
视图	33
使用视图 Composer 在每个视图中共享变量	35
控制器.....	35
获取用户输入.....	38
将依赖注入控制器	39

资源控制器	40
路由模型绑定	42
隐式路由模型绑定	42
自定义路由模型绑定	43
路由缓存	44
表单方法欺骗 (Form Method Spoofing)	44
HTTP 动词的介绍	44
Laravel 中的 HTTP 动词	45
在 HTML 表单中进行 HTTP 方法欺骗	45
CSRF 保护	45
重定向	47
redirect()->to()	48
redirect()->route()	48
redirect()->back()	49
其他重定向方法	49
redirect()->with()	49
中止请求	51
自定义响应	51
response()->make()	51
response()->json() 和 ->jsonp()	52
response()->download() 和 ->file()	52
测试	52
本章小结	53
第 4 章 Blade 模板	55
输出数据	56
控制结构	56
条件语句	57
循环语句	57
or	59
模板继承	60
用 @section/@show 和 @yield 定义 Section	60
@parent	62

@include	62
@each.....	63
视图 Composer 和服务注入	63
用视图 Composer 绑定数据到视图	64
Blade 服务注入	67
自定义 Blade 指令	68
自定义 Blade 指令中的参数.....	69
示例：对多租户应用程序（Multitenant App）使用自定义 Blade 指令.....	70
测试	71
本章小结	72
第 5 章 前端组件	73
Elixir	73
Elixir 文件夹结构.....	75
运行 Elixir.....	75
Elixir 提供了什么.....	76
分页	82
分页数据库结果	82
手动创建分页.....	83
消息包.....	84
错误包命名	85
字符串助手、多元化和本地化	86
字符串助手和多元化.....	86
本地化.....	87
测试	89
用 Elixir 进行测试.....	89
测试消息包和错误包.....	90
翻译和本地化.....	90
本章小结	90
第 6 章 收集和处理用户数据	91
注入请求对象	91
\$request->all()	92

\$request->except() 和 \$request->only()	92
\$request->has() 与 \$request->exists()	93
\$request->input()	93
数组输入	94
JSON 输入 (\$request->json())	94
路由数据	96
通过 Request 实现	96
通过路由参数实现	96
上传的文件	96
验证	99
在控制器中使用 ValidatesRequests 的 validate() 方法	99
手动验证	102
显示验证错误信息	102
表单请求	103
创建表单请求	103
使用表单请求	104
Eloquent 模型质量分配	105
{{ 与 {!!	106
测试	106
本章小结	107

第 7 章 Artisan 和 Tinker 109

Artisan 入门	109
Artisan 的基本命令	110
选项	110
组合命令	111
书写常见的 Artisan 命令	113
注册命令	115
示例命令	116
参数和选项	117
输入	118
提示	120
输出	121

在其他代码中调用 Artisan 命令	122
Tinker	123
测试	124
本章小结	125
第 8 章 数据库和 Eloquent.....	127
配置	127
数据库连接	127
其他数据库配置选项.....	129
迁移	129
定义迁移	129
运行迁移	137
填充	138
创建填充器	138
模型工厂	139
查询构造器	143
DB Facade 的基本使用.....	143
原始 SQL 语句	144
查询构造器链.....	145
事务	153
Eloquent 入门.....	154
新建和定义 Eloquent 模型	156
通过 Eloquent 获取数据	157
Eloquent 的插入和更新	159
Eloquent 中的删除	162
作用域	164
自定义与访问器、修改器和属性转换器的字段交互	167
Eloquent 集合	171
Eloquent 序列化	173
Eloquent 关系	175
通过子类更新父类时间戳	186
Eloquent 事件.....	188
测试	189

本章小结	191
第 9 章 用户认证和授权	193
用户模型和迁移	194
使用 auth() 全局助手和认证 Facade	197
Auth 控制器	197
RegisterController	197
LoginController	199
ResetPasswordController	200
ForgotPasswordController	200
Auth::routes()	200
认证脚手架 (Auth Scaffold)	201
“记住我”	202
手动认证用户	203
认证中间件	204
保护	204
修改默认保护	205
在不改变默认情况下使用其他保护	205
添加新的保护	205
创建自定义用户提供器	206
为非关系型数据库自定义用户提供器	207
认证事件	207
授权 (ACL) 和角色	208
定义授权规则	208
Gate facade (和注入 Gate)	209
Authorize 中间件	210
控制器授权	210
检查用户实例	212
Blade 检查	213
插入检查	213
政策	214
测试	216
本章小结	218

第 10 章 请求和响应	221
Laravel 请求的生命周期	221
引导应用程序	222
服务提供者	223
Request 对象	224
在 Laravel 中获取请求对象	225
获取请求的基本信息	225
持久性	228
Response 对象	228
在容器中使用和创建 Response 类	229
特殊的响应类型	230
Laravel 和中间件	233
中间件入门	233
创建自定义中间件	234
绑定中间件	236
向中间件传参	239
测试	240
本章小结	241
 第 11 章 容器	 243
依赖注入简介	243
依赖注入和 Laravel	245
app() 全局助手	245
容器如何连接	246
将类绑定到容器	247
绑定到闭包	247
绑定单例模式、别名和实例	248
将具体实例绑定到接口	249
语境绑定	250
构造器注入	250
方法注入	251
facade 与容器	252
facade 如何工作	252

服务提供者	254
测试	254
本章小结	255
第 12 章 测试	257
测试基础	258
命名测试	260
测试环境	261
测试特性	262
没有中间件	262
数据库迁移	262
数据库事务	263
应用程序测试	263
测试用例	263
“访问”路径	264
自定义应用测试断言	266
JSON 和 Non-visit() 应用测试断言	267
点击和表单	271
任务和事件	273
认证和会话	273
Artisan 和 Seed	274
mock（模拟）	275
Mockery	275
模拟 facade	278
本章小结	279
第 13 章 编写 API	281
类 REST JSON API 基础	281
控制器组织和 JSON 返回	282
读取和发送头	286
在 Laravel 中发送响应头	287
在 Laravel 中读取请求头	287
Eloquent 分页	287

排序和筛选	289
对 API 结果排序	289
过滤 API 结果	291
数据转换	292
编写自己的转换器	292
嵌套和关系	293
使用 Laravel Passport 的 API 认证	295
OAuth 2.0 简介	295
安装 Passport	296
Passport 的 API	297
Passport 可用的授权类型	298
使用 Passport API 和 Vue 组件管理客户端和令牌	305
Passport 作用域	307
Laravel 5.2 以上版本的 API 令牌认证	309
测试	310
本章小结	311

第 14 章 存储和检索313

本地和云端文件管理器	313
配置文件访问	313
使用存储 facade	314
添加额外的 Flysystem 提供商	316
基本的文件上传和操作	317
会话	318
访问会话	318
会话实例的可用方法	319
闪存会话存储	321
高速缓存器 cache	321
访问高速缓存	322
Cache 实例中可用的方法	322
cookie	324
Laravel 中的 cookie	324
访问 cookie 工具	324

基于 Laravel Scout 全文搜索	327
安装 Scout	327
标记索引模型	327
索引检索	328
队列和 Scout	328
执行无索引操作	328
通过代码手动触发索引	329
利用 CLI 手动触发索引	329
测试	329
文件存储	329
会话	331
高速缓存	332
cookie	332
本章小结	334
 第 15 章 邮件和通知	335
邮件	335
“classic” 邮件	336
基本 “mailable” 邮件	336
邮件模版	339
build() 中可用的方法	339
附件和内联图片	340
队列	341
本地开发	342
通知	344
为通知对象定义 via() 方法	346
发送通知	347
排队通知	348
开箱即用的通知类型	348
测试	351
邮件	351
通知	352
本章小结	352

第 16 章 队列，任务，事件，广播及调度程序	353
队列	353
为什么使用队列	354
基本队列配置	354
队列任务	354
运行队列工作者	358
错误处理	358
控制队列	361
支持其他功能的队列	361
事件	362
触发事件	362
监听事件	364
通过 WebSocket 广播事件及 Laravel Echo	367
配置和设置	368
广播事件	368
接收消息	371
高级广播工具	372
Laravel Echo（JavaScript 方面）	376
调度程序	379
可用任务类型	380
可用时间框架	380
阻塞和重叠	382
处理任务输出	382
任务钩子	383
测试	384
本章小结	385
 第 17 章 助手和集合	387
助手	387
数组	387
字符串	389
应用路径	391
URL	392

Misc（宏指令结构技术体系）..... 394

集合 397

 集合的基础 397

 几种方法 399

本章小结 404

词汇表 405

前言

一个很常见的问题是，应该如何学习 Laravel？尽管已经写了很多年的 PHP 程序，但是我不想闭门造车，我想更好地学习 Rails 的强大特性，也想学习其他先进的 Web 框架。Rails 有一个非常活跃的社区，完美结合了默认配置及其灵活性，并且具有 Ruby Gems 的能力来提高预包装的常用代码。

我曾经也很犹豫是否要继续使用 PHP，直到知道了 Laravel，我才坚定信心。Laravel 具备 Rails 所有的特性，但它不仅仅是 Rails 的一个副本，更是一个全新的 Web 框架。Laravel 提供了非常完善的文档、开放的交流社区，还继承了许多现有的语言和框架。

当我在博客和会议中分享 Laravel 的学习经验时，我已经使用 Laravel 开发了几十个应用程序和项目，并且可以在线或者面对面与成千上万名 Laravel 开发者进行交流。我的工具箱中已经有很多种开发工具了，但是当打开命令行终端，输入 `laravel new project` 时，我还是会感到非常激动。

关于本书

这并不是第一本关于 Laravel 的书，也不会是最后一本。我不打算让它成为一本涵盖每一行代码或实现模式的书。我也并不希望它一定要与现在最新版本的 Laravel 保持一致。相反，我编写本书的主要目的是提供一个高阶的概述和具体的例子，帮助开发者更快地学习 Laravel。我希望能帮助读者理解 Laravel 背后的基本概念，而不是单纯讲解文档。

Laravel 是一个强大且灵活的 PHP 框架。它拥有发展迅速的社区、丰富的开发工具，这些特性都让 Laravel 越来越受欢迎。本书旨在帮助已经知道如何开发网站和应用程序的开发者学习如何在 Laravel 中进行开发。

Laravel 的官方文档非常清晰明了，如果读者对一些特定的内容感兴趣，但本书中又没有进行深入讲解，那么建议读者参考 Laravel 的在线文档进行深入学习。

本书不仅对知识点进行概述，也提供了相应的实例。在学习完本书后，相信读者便可以使用 Laravel 从头开始实现一个完整的应用程序了。希望本书能激发大家动手操作的兴趣。

目标读者

读者要具备基本的面向对象编程基础，了解 PHP（或者至少是 C 语言的一般语法），以及模型 - 视图 - 控制器 (MVC) 开发模式和模板的基本概念。如果从来没有开发过网站，那么可能会觉得本书的内容比较难懂。但是只要有一定的编程基础，就不需要在阅读本书之前了解 Laravel——我们会在本书中从“Hello, world!”开始，覆盖所有需要了解的内容。

本书结构

本书按照时间顺序编写：如果正在使用 Laravel 构建第一个 Web 应用程序，那么前面的章节将介绍开始时需要使用的基本组件，后面的章节将更深入地介绍知识点及对应的具体实例。

本书的每一部分都可以单独阅读，但是也力图使章节之间的联系更加紧密，以便刚接触框架的读者能够从头到尾顺畅地进行阅读和学习。

每章节的最后都包括“测试”和“本章小结 (TL;DR)”，TL;DR 表示“长话短说”。这两部分将展示如何测试对应章节中讲过的方法，并对所涵盖的内容进行高阶概述。

本书基于 Laravel 5.3 版本，但是因为 Laravel 5.1 是最新的 LTS 版本，所以 Laravel 5.2 或 Laravel 5.3 中的新功能将被标记出来。

本书使用的排版约定

本书使用如下排版约定：

斜体 (*Italic*)

标志着新词汇、URL、邮箱地址、文件名和文件扩展名。

等宽字体 (Constant width)

用于程序清单（包括段落中的），表示程序元素，如变量名或函数名、数据库、数

据类型、环境变量、语句和关键字。

等宽字体加粗 (**Constant width bold**)

显示字面上被用户输入的命令或其他文本。

等宽字体且斜体 (*Constant width italic*)

显示应该被用户提供的值或者由上下文决定的值所替换的文本。



该图标表示技巧或建议。



该图标表示一般注释。



图标表示警告或者注意事项。

O'Reilly Safari



Safari[®] Safari (以前的 Safari Books Online) 是企业、政府、教育者和个人的会员制培训及参考平台。

订阅者可以从一个完全可搜索的数据库中获得来自 250 多家出版商提供的成千上万的书
籍、培训视频、互动教程，这些出版商包括 O'Reilly Media、Harvard Business Review、
Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、
Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley&Sons、Syngress、
Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、
New Riders、McGraw-Hill、Jones&Bartlett、Course Technology。

若想获得更多资讯，请访问 <http://oreilly.com/safari>。

如何联系我们

请将对本书的评价和发现的问题通过如下地址通知出版社。

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

我们提供了本书网页，上面列出了勘误表、示例和其他信息。请通过 <http://bit.ly/laravel-up-and-running> 访问该页。

要给出本书意见或者询问技术问题，请发送邮件到 bookquestions@oreilly.com。

更多有关书籍、课程、会议和新闻的信息，请见网站 <http://www.oreilly.com>。

在 Facebook 找到我们：<http://facebook.com/oreilly>

在 Twitter 上关注我们：<http://twitter.com/oreillymedia>

在 YouTube 上观看：<http://www.youtube.com/oreillymedia>

致谢

如果没有妻子 Tereva 的全力支持以及儿子 Malachi 的理解（“这是我爸爸的书，哥们儿！”），就不会有这本书。女儿 Mia 几乎见证了这本书的全部创作过程，虽然她自己都不知道。所以，这本书是献给我的家人的。因为编写本书，我经常难以在周末与家人一起享受傍晚时光和星巴克之旅，感谢家人的支持，他们的存在让我的生活更加美好。

另外，Tighten Co. 也为我提供了很大的支持和鼓励，特别是其中有几位负责编辑（杰出的编辑工作者 Keith Damiani）和测试代码的人员（Adam Wathan，信息收集之王），感谢他们。感谢合作伙伴 Dan Sheetz 对我的支持，感谢运营经理 Dave Hicking 帮我安排时间以及明确工作职责。

感谢 Taylor Otwell 开发了 Laravel，他为我们创造了如此多的就业机会，使许多开发人员更加热爱生活。他希望开发者能感受到幸福，并且致力于建立一个活跃的、令人备受

鼓舞的社区。他也是一个善良、励志、喜欢挑战的朋友。谢谢你，Taylor。

感谢 Jeffrey Way, 我仍然认为他是互联网上最好的老师之一。感谢他将 Laravel 介绍给我。他是一位非常出色的工程师，很高兴能和他成为朋友。

感谢 Jess D’Amico、Shawn McCool、Ian Landsman 和 Taylor，给我机会让我作为会议的发言人，并且当了一次老师。感谢 Dayle Rees, 让很多人在早期学习 Laravel 时不至于困惑。

感谢每一位辛苦编写 Laravel 博客的朋友，特别是 Eric Barnes、Chris Fidao、Matt Machuga、Jason Lewis、Ryan Tablada、Dries Vints、Maks Surguy 等，感谢你们。

感谢这些年在 Twitter、IRC 和 Slack 上与我交流的朋友们，我不希望遗漏掉任何一位朋友。你们都非常棒，希望我们能保持联系。

感谢 O’Reilly 的编辑 Ally MacDonald，以及所有的技术编辑：Keith Damiani、Michael Dyrynda、Adam Fairholm 和 Myles Hyson。

最后，感谢在编写本书的过程中所有支持我的家人、朋友。感谢我的父母、兄弟姐妹。感谢 Gainesville 社区、企业老板、作者、会议发言人，以及独一无二的 DCB。感谢在我没有灵感、暂停写作时给予我帮助的星巴克咖啡师。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/33611>



为什么使用Laravel

在动态网页发展前期，写一个 Web 应用程序看起来和今天有很大的不同。开发人员不仅负责编写应用程序中独特的业务逻辑代码，还负责编写站点中各个公共组件的代码，包括用户认证、输入验证、数据库访问和模板等内容。

当今已经有数十个应用程序开发框架可供程序员选择，并且可以轻松获取数千个组件和库。对程序员来说，通常面临一个问题：在学习一款框架的时候，一些新的框架（而且据说是更好的框架）已经出现，并且未来可能会取代正在学习的这款框架。

“因为山在那里”，也许是攀登一座山峰的理由，但是选择使用一款特定的框架（或者某款框架）可以有更好的理由。值得思考的是：为什么使用框架？更准确地说，为什么使用 Laravel 这款框架？

为什么要使用框架

从 PHP 开发人员可以使用的一些组件以及软件包来看，其实很容易得知为什么使用这些组件以及软件包会有好处。使用包，可以让其他人负责开发和维护一个独立的、具有特定功能的代码块，从理论上说，他们更有时间对这些单个的组件进行深入的了解与研究。

Laravel、Symfony、Silex、Lumen 和 Slim 等框架会将第三方组件和自定义框架预先装好，比如配置文件、服务提供者、规定的目录结构，以及应用程序引导等。所以，一般来说，使用框架的好处是，不仅有选择好的单独组件，而且会帮助开发人员决定如何组合这些组件。

“自己动手做”

让我们来分析一下没有框架的好处。假如要创建一个 Web 应用，会从哪里开始？也许会从 HTTP 请求开始，所以现在需要评估所有可用的 HTTP 请求库和 HTTP 响应库并选择其中一个。接下来需要建立路由，可能需要设置好某种形式的路由配置文件。应该使用什么语法？怎么做好控制器？应该把它们放在哪里，以及如何加载它们？可能需要一个依赖注入容器来解决控制器及其依赖问题，但是具体选择哪一个呢？

此外，如果花时间解答了以上所有问题并成功创建了应用程序，会对下一个开发人员产生什么影响？当有四个这样基于自定义框架的应用程序或十几个应用程序时，必须记住控制器在每个应用程序中的位置，以及路由的语法。

一致性和灵活性

“我们应该在这里使用哪个组件？”框架解决了这个问题。它提供了一个深思熟虑后的答案，并确保所选择的特定组件能够很好地协同工作。此外，框架提供了一些约定，可以减少开发人员在创建新项目时必须理解的代码量。例如，如果了解一个 Laravel 项目中的路由是如何工作的，那么也就了解了路由在所有 Laravel 项目中的工作原理。

当有人为每一个新项目制定自己的框架时，他们真正在意的是在不改变应用程序基础部分内容的前提下，还可以去控制什么。这意味着最好的框架不仅可以提供稳定的基础部分内容，而且还可以对核心部分的内容进行个性化定制。正如本书其余部分所展示的一样，这也是 Laravel 框架如此特别的原因之一。

Web 和 PHP 框架简史

如果要回答“为什么使用 Laravel”这个问题，极为重要的一点是需要理解 Laravel 的历史，并了解它之前的内容。在 Laravel 普及之前，PHP 和其他 Web 开发领域就有许多框架以及其他动态。

Ruby on Rails

2004 年，David Heinemeier Hansson 发布了 Ruby on Rails 的第一个版本，从那时起就很难找到一个 Web 应用程序框架完全没有受到 Rails 的影响。

Rails 让 MVC、RESTful JSON API、按约定编程（convention over configuration）、活动记录（Active-Record）以及更多的工具和约定得到了普及，这对 Web 开发人员开发应用

程序的方式产生了深远的影响，尤其是在应用程序快速开发方面。

PHP 框架的涌入

大多数开发人员很清楚，Rails 和类似的 Web 应用程序框架是未来的浪潮，PHP 框架（包括那些直接模仿 Rails 的）开始如雨后春笋般出现。

CakePHP 是 2005 年出现的第一个 PHP 框架，很快就出现了 Symfony、CodeIgniter、Zend Framework 和 Kohana（CodeIgniter 的一个分支）等 PHP 框架。2008 年出现了 Yii，2010 年出现了 Aura 以及 Slim。2011 年，出现了 FuelPHP 以及 Laravel，这两款产品都不是 Code CodeIgniter 分支，而是作为替代方案。

在这些框架中，更多的是 Rails-y 系列，它们重点关注的是数据库对象关系映射（ORMs）、MVC 结构，以及其他针对快速开发的工具。其他的框架，像 Symfony 和 Zend，更多关注于企业设计模式和电子商务。

CodeIgniter 的优点与缺点

CakePHP 和 CodeIgniter 是两个早期的 PHP 框架，它们的灵感来自 Rails。CodeIgniter 很快成名，到了 2010 年，可以说它成为受欢迎的独立 PHP 框架。

CodeIgniter 简单易用，并拥有大量的文档和强大的社区。但是它在现代技术和模式的使用方面发展缓慢，CodeIgniter 在技术和开箱即用（out-of-the-box）特性方面都开始落后了。与其他许多框架不同，CodeIgniter 是由一家公司管理的，他们对 PHP 5.3 的新特性，如命名空间、对 GitHub 的迁移和对后来的 Composer 的支持等，发展比较缓慢。在 2010 年，Laravel 的创始人 Taylor Otwell 对 CodeIgniter 感到不满，他开始写自己的框架。

Laravel 1、Laravel 2 和 Laravel 3

2011 年 6 月，Laravel 1 的第一个测试版本发布了，它完全是从零开始的。它具有一个自定义的 ORM（Eloquent）、基于闭包的路由（灵感来自 Ruby Sinatra）、扩展模块系统，以及用于表单、认证、身份验证等功能的助手组件。

Laravel 早期发展迅速，Laravel 2 和 Laravel 3 分别于 2011 年 11 月和 2012 年 2 月发布。它们引入了控制器、单元测试、命令行工具、控制反转（IoC）容器、Eloquent 关联和迁移等。

Laravel 4

在 Laravel 4 中，Taylor 从头开始重写了整个框架。在这一点上，Composer（如今无处不在的 PHP 软件包管理器）已经显示出成为行业标准的迹象，而泰勒（Taylor）也看到了将框架重新编写为 Composer 分发和打包在一起的组件集的价值。

泰勒开发了一款名为 Illuminate 的组件，于 2013 年 5 月发布了 Laravel 4，并在 Laravel 4 中使用了一种全新的结构。Laravel 的大部分代码不能捆绑下载，而是使大部分组件通过 Symfony（另一个发布组件以供其他用户使用的框架）和 Composer 的 Illuminate 组件获得。

Laravel 4 还引入了队列、邮件组件、外观和数据库种子（database seeding）。由于 Laravel 现在依赖于 Symfony 的组件，所以 Laravel 宣布，Laravel 将会效仿 Symfony 的 6 个月（不是确切的，但很快）的发布时间表。

Laravel 5

Laravel 4.3 计划于 2014 年 11 月发布，但是随着时代的进步与发展，显而易见，其有了重大的改变，所以值得发布更大的消息（译者注：由于变化太大，Laravel 4.3 取消，直接发布 5.0 版本），最后于 2015 年 2 月发布了 Laravel 5。

Laravel 5 具有改进后的目录结构、表单删除和 HTML 助手类，引入了契约机制（contract）、大量新视图、可以用于社交媒体身份验证的 Socialite、可以用于资源编译的 Elixir、可以用于简化调度程度的任务调度（cron）、可以用于简化环境管理的 dotenv、表单请求以及全新的 REPL（“读取 - 求值 - 输出”循环）。

Laravel 有什么特别之处

是什么让 Laravel 与众不同呢？为什么任何时候都要有一个以上的 PHP 框架？它们都使用 Symfony 的组件，对吧？让我们来谈谈是什么让 Laravel 变得特别。

Laravel 的哲学

只需要阅读 Laravel 的营销材料和自述文件（READMEs），就能看到它的价值。Taylor 使用了一些与光相关的词汇，如照明（Illuminate）、火花（Spark）。

然后还有这些词汇：工匠（Artisans）、优雅（Elegant）。除此之外，还有一些短语：呼吸新鲜空气（Breath of fresh air）、新的开始（Fresh start）。最后还有这些词语：快速（Rapid）、异乎寻常的速度（Warp speed）。

这个框架传递出强烈的价值感：提高开发人员的操作速度和开发人员的幸福感。泰勒（Taylor）将“工匠（Artisan）”语言描述为是一种更具有实用价值的语言。在 2011 年的 StackExchange (<http://bit.ly/2dT5kmS>) 问题上，就可以看到这种想法的起源，他说：“有时候我花了大量的时间，只是为了让代码看起来更漂亮，这是比较令人苦恼的事情。而且这只是为了更好地阅读代码。”他经常谈到要让开发人员更容易、更快地实现他们的想法，消除不必要的、创建优秀产品的障碍。

Laravel 的核心是为开发人员提供装备和支持。它的目标是提供清晰、简单、漂亮的代码和特性，帮助开发人员快速学习和开发，并编写出简单、清晰和持久的代码。

针对开发者的概念在 Laravel 材料中写得很清楚，该文档明确写道——开发者最好的代码。同时，“让开发人员从下载到部署都快乐”亦是一段非正式的口号。当然，任何工具或框架都会说它想让开发人员高兴。但是，把开发人员的快乐作为首要关注对象而不是次要的问题，这一点对 Laravel 的风格和决策过程产生了巨大的影响。在其他框架中，可能会把体系结构的纯度（architectural purity）作为首要目标，抑或需要符合企业开发团队的目标和价值观，而 Laravel 的主要目标则是为开发人员服务。

Laravel 如何让开发者快乐

只是说想让开发者快乐是一回事，真正做到则是另一回事，它需要猜测框架中什么地方最有可能让开发人员不开心，什么地方又最有可能让他们开心。Laravel 有几种方法试图让开发人员的生活变得更加轻松。

首先，Laravel 是一个快速应用程序开发框架。这意味着它关注简单的学习过程，并尽量减少一个新的应用程序从启动到发布的步骤。所有在构建 Web 应用程序时最常见的任务，从数据库交互到身份验证、从队列到电子邮件再到缓存，都通过 Laravel 提供的组件简化了。但是，Laravel 的组件并不仅仅局限于此，它们在整个框架中提供了一个统一的 API 和稳定的结构。这意味着，在 Laravel 做出新的尝试时，开发人员很可能作出肯定的评价：“就是这么好用！”

其好处也不会就此结束，Laravel 提供了一个完整的、用于构建和发布应用程序的工具生态系统。可以使用 Laravel 的 Homestead 以及 Valet 工具进行本地开发，使用 Forge 工具进行服务器管理，以及使用 Envoyer 工具进行高级部署。此外，还有一套附加组件：Cashier（用于支付和订购）、Echo（用于 WebSocket 编程）、Scout（用于实现搜索功能）、Passport（用于 API 认证）、Socialite（用于社交登录）以及 Spark（用于 SaaS 引导）等。Laravel 正试图从开发人员的工作中提取出这些重复性的工作，这样开发人员就可以做一些独特的事情了。

接下来要说的是，Laravel 关注的是“约定优于配置”——这意味着，如果愿意使用 Laravel 的默认设置，那么相比于其他框架，就可以减少很多工作。在其他框架中，即使使用了推荐的配置，也需要对所有的设置进行声明。使用 Laravel 构建的项目比其他大多数 PHP 框架花费的时间少。

Laravel 还专注于简约性。如果需要，可以使用依赖注入（injection）、模拟（mocking）、数据映射模式（Data Mapper pattern）、仓库模式（repositories）、命令查询责任分离（Command Query Responsibility Segregation），以及其他更复杂的架构模式。虽然其他框架可能会建议在每个项目中使用这些工具和结构，但是 Laravel 及其文档和社区更倾向于从最简单的开始，比如从一个全局函数、门面（facade）、ActiveRecord 开始。这使得开发人员可以创建最简单的应用程序来解决他们的需求。

有趣的是，Laravel 的创造者以及它的社区与 Ruby、Rails 和函数式编程语言有更多的联系，而与 Java 的联系则较少。在现代 PHP 中，有一个强大的潮流，就是倾向于变得更加复杂，它囊括了更多的 Java 风格的 PHP 代码。但 Laravel 则不同，它会追求更富有表现力的、动态的、简单的编码实践和语言特性。

Laravel 社区

如果第一次接触 Laravel 社区，那么会有些令人期待。Laravel 有一个显著的特点就是它的成长和成功与它受欢迎的教学社区是密不可分的。从 Jeffrey Way 的 Laracasts 视频教程 (<https://laracasts.com/>)，到 Laravel 资讯 (<https://laravel-news.com/>)，到 Slack 和 IRC 频道，从 Twitter 网友到 Laravel 博客，再到 Laravel 的会议。Laravel 拥有一个丰富而充满活力的社区，这里汇聚了从第一天开始就一直在这里的人和那些第一次到来的人。这不是偶然的：

从一开始接触 Laravel，我就有这样的想法，所有的人都希望自己是某种东西的一部分。想要拥有一群志同道合的人并加入其中，这是人类的一种本能。因此，通过将人格注入一个 Web 框架并在社区中真正活跃起来，这种感觉可以在社区中发展起来。

——Taylor Otwell，《产品与支持访谈》

Taylor 在 Laravel 的早期开发工作中就意识到，一个成功的开源项目需要良好的文档和受欢迎的社区。它们现在都是 Laravel 的标志。

Laravel 是如何工作的

到目前为止，我在这里所分享的一切都是抽象的。你可能会问，Laravel 的代码是怎样的呢？让我们深入了解一个简单的应用程序（见示例 1-1），这样就可以看到在日常工作中，Laravel 实际上是什么样的了。


示例1-1 在routes/web.php中实现 “Hello,World”

// 文件： routes/web.php

<?php

```
Route::get('/', function() {  
    return 'Hello, World!';  
});
```

在 Laravel 应用程序中，最简单的操作可能就是定义一条路径，并在任何时间访问该路径时返回结果。如果在计算机上初始化一个全新的 Laravel 应用程序，在示例 1-1 中定义好路由，然后从公共目录中使用该站点，那么你将拥有一个功能齐全的 “Hello, World” 示例程序（见图 1-1）。



Hello, World!

图1-1 通过Laravel返回 “Hello, World!”

该操作与控制器的操作非常相似，如示例 1-2 所示。

示例1-2 通过控制器实现 “Hello, World”

// 文件： routes/web.php

<?php

```
Route::get('/', 'WelcomeController@index');
```

// 文件： app/Http/Controllers/WelcomeController.php

<?php

```
namespace app\Http\Controllers;
```

```
class WelcomeController  
{  
    public function index()  
    {  
        return 'Hello, World!';  
    }  
}
```

如果我们在数据库中存储问候信息，它看起来和控制器的操作也很相似（见示例 1-3）。

示例1-3 使用数据库访问问候信息 “Hello, World”

```
// 文件：routes/web.php
<?php

Route::get('/', function() {
    return Greeting::first()->body;
});
// 文件：app/Greeting.php
<?php

use Illuminate\Database\Eloquent\Model;

class Greeting extends Model {}

// 文件：database/migrations/2015_07_19_010000_create_greetings_table.php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateGreetingsTable extends Migration
{
    public function up()
    {
        Schema::create('greetings', function (Blueprint $table) {
            $table->increments('id');
            $table->string('body');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::drop('greetings');
    }
}
```

如果示例 3-3 有点令人难以理解，那就先跳过它。在后面的章节中，我们将了解这里所发生的一切，但是可以看到，只需几行代码，就已经建立了数据库迁移和模型，并将记录读取出来。

为什么使用 Laravel

那么，为什么使用 Laravel?

因为 Laravel 可以帮助你想法变成现实，并且不会浪费代码（即以极精简的代码实现强大的功能），使用现代编码标准，并拥有一个充满活力的社区，还拥有强大的工具生态系统。

更因为你，亲爱的开发者，使用 Laravel 会让你快乐！

搭建Laravel的开发环境

PHP 成功的部分原因在于其很容易就可以找到一个支持 PHP 服务的 Web 服务器。然而，随着对现代 PHP 工具更加严格的要求，开发 Laravel 应用程序的最佳方法是让代码拥有一致的本地和远程服务器环境。为此，Laravel 生态系统提供了一些工具。

系统要求

本章介绍的所有内容都可以通过 Windows 机器实现，但有可能会遇到数十个页面的自定义说明和注意事项。这些说明和注意事项将留给 Windows 用户自行尝试处理，因此，这里的例子和本书的其他部分内容将主要针对 UNIX/Linux/MAC OS 开发人员进行介绍。

无论是选择直接在本地机器上安装 PHP 还是其他工具来搭建站点开发环境，例如，通过 Vagrant 创建一个虚拟机来搭建站点开发环境，抑或通过 MAMP/WAMP/XAMPP 等工具来搭建站点开发环境，如果希望最终搭建好支持 Laravel 站点服务的环境，那么在开发环境中需要安装以下工具。

- 对于 Laravel 5.3，PHP 版本须高于 5.6.4；对于 Laravel 5.1 及 Laravel 5.2，PHP 版本须高于 5.5.9
- OpenSSL PHP 扩展
- PDO PHP 扩展
- Mbstring PHP 扩展
- Tokenizer PHP 扩展

Composer

无论在什么机器上进行开发，都需要全局安装 Composer (<https://getcomposer.org/>)。Composer，简单地说，它是现代 PHP 开发的基础。Composer 是 PHP 用于管理依赖关系的一款工具，非常像 Node 的 NPM 或 Ruby 的 RubyGems。需要使用 Composer 来安装

Laravel、更新 Laravel，以及引入外部依赖关系。

本地开发环境

对于许多项目来说，使用一些简单的工具集来搭建开发环境就足够了。如果系统上已经安装了 MAMP、WAMP 或 XAMPP，就可以轻而易举地运行 Laravel。如果系统上的 PHP 版本正确，那么也可以用 PHP 内置的 Web 服务器来运行 Laravel。

只要让 PHP 可以运行，就算是准备就绪了。至于是否按照以上步骤去操作，一切由你决定。

Laravel 为本地开发提供了两个工具，即 Valet 和 Homestead，下面进行简单介绍。如果不确定该使用哪一种工具，建议使用 Valet 工具，至于 Homestead 工具，只需要简单了解即可。当然，这两种工具都很有价值，值得掌握。

Laravel Valet

如果使用的是 PHP 内置的 Web 服务器，那么最简单的方法就是直接使用本地 URL 地址访问每个站点。如果在 Laravel 站点的根目录中运行 `php -S localhost:8000 -t public`，将会启动 PHP 内置的 Web 服务器，并且默认可以通过 `http://localhost:8000/` 访问站点。还可以运行 `php artisan serve`，一旦设置好应用程序，同样可以轻松地启动一个服务器。

如果有兴趣将每个站点绑定到各自专属的域名上，则需要使用操作系统上的主机文件以及使用像 dnsmasq (<http://bit.ly/2eNPJ5T>) 这样的工具来实现。让我们尝试一些更简单的操作吧。

对于 Mac 用户（也包括非官方的 Windows 和 Linux），Laravel Valet (<https://laravel.com/docs/5.5/valet>) 会将该用户的域名和应用程序文件夹进行关联。也可以使用 Valet 来安装 dnsmasq 以及一系列的 PHP 脚本，这样输入 `laravel new myapp && open myapp.dev` 就可以正常运行了。需要使用 Homebrew 来安装一些工具，这个过程可以根据文档进行，并且从初始化安装到服务应用程序的整个步骤非常简单。

安装 Valet（请参阅最新的安装指南 (<https://laravel.com/docs/5.5/valet>) ——它在本书写作时处于非常活跃的开发阶段），并将其指向一个或多个网站所在的目录。我在 `~/Sites` 目录中运行 `valet park`，这个目录是我所开发的未完成版应用程序所在的目录。现在，可以将 `.dev` 添加到目录名的末尾，并在浏览器中访问它。

Valet 可以轻松使用 `valet park` 操作类似于“`FOLDERNAME.dev`”形式指定目录中的所有文件夹。比如，可以使用 `valet link` 命令实现提供单个站点服务，也可以使用 `valet open` 命令打开一个 Valet 服务域名，同样可以使用 `valet secure` 命令使项目支

持 HTTPS, 以及打开一个 ngrok 通道, 当然也可以使用 `valet share` 命令与他人分享站点。

Laravel Homestead

Homestead 是用来配置本地开发环境的另一种工具。Homestead 是建立在 Vagrant 上的一个配置工具, 它提供了一个预先配置好的虚拟机镜像, 可以完美地用于 Laravel 开发, 并且拥有许多在运行 Laravel 站点时常见的开发环境。

安装 Homestead

如果选择使用 Homestead, 相比于 MAMP 或 Valet, 则需要花更多的精力进行设置。然而, Homestead 具有非常多的优点: 只要配置正确, 本地环境可以非常接近远程工作环境, 不必担心更新本地机器上的依赖关系; 可以从本地机器的安全性中了解 Ubuntu 服务器的所有结构。

Homestead 提供什么工具?

可以随时升级 Homestead 虚拟机中的各个组件, 但是在默认情况下, Homestead 附带了一些重要工具, 如下所示。

- 服务器运行时需要的工具以及站点支持工具, 如 Ubuntu、PHP 和 Nginx (类似于 Apache 的一种 Web 服务器)。
- 为提供数据库、存储以及消息队列提供支持的工具, 例如 MySQL、Postgres、Redis、Memcached 以及 *beanstalkd* 等。
- 构建步骤中需要的其他工具, 例如 Node。

安装 Homestead 的依赖

首先需要下载并安装 VirtualBox (<https://www.virtualbox.org/wiki/Downloads>) 或者 VMWare。由于 VirtualBox 是免费的, 所以它比较常用。

然后需要下载并安装 Vagrant (<https://www.vagrantup.com/downloads.html>)。

Vagrant 使用起来很方便, 可以从预先创建的 “box” 中创建一个新的本地虚拟机, 实际上, “box” 是虚拟机的模板。接下来在终端上运行 `vagrant box add laravel/homestead`, 这样就可以下载 “box” 了。

安装 Homestead

接下来安装 Homestead。可以安装多个 Homestead 实例 (也许每个项目都有一个不同的

Homestead box)，但是我更喜欢所有的项目只使用一个 Homestead 虚拟机。如果希望每个项目都有各自的 Homestead 虚拟机，则需要项目目录中安装 Homestead，这样可以在线（<https://laravel.com/docs/5.3/homestead>）查看 Homestead 的文档说明。如果希望所有的项目共用一个虚拟机，则可以在用户主目录中使用以下命令安装 Homestead。

```
git clone https://github.com/laravel/homestead.git ~/Homestead
```

现在，在 Homestead 目录所在的位置运行初始化脚本，代码如下。

```
bash ~/Homestead/init.sh
```

这会将 Homestead 的主配置文件 *Homestead.yaml* 移动到一个新的目录 *~/homestead* 中。

配置 Homestead

打开 *Homestead.yaml* 并按照需求进行配置，以下是文件的初始内容。

```
ip: "192.168.10.10"
memory: 2048
cpus: 1
provider: virtualbox

authorize: ~/.ssh/id_rsa.pub
keys:
  - ~/.ssh/id_rsa

folders:
  - map: ~/Code
    to: /home/vagrant/Code

sites:
  - map: homestead.app
    to: /home/vagrant/Code/Laravel/public

databases:
  - homestead

# blackfire:
#   - id: foo
#     token: bar
#     client-id: foo
#     client-token: bar

# ports:
#   - send: 50000
```

```
# to: 5000
# - send: 7777
# to: 777
# protocol: udp
```

这里需要告诉服务提供者（比如 `virtualbox`），配置好 SSH 公钥信息（默认情况下在 `~/.ssh/id_rsa.pub` 中；如果没有，可以到 GitHub（<http://bit.ly/2e7Auof>）中找相应教程，GitHub 上有一些关于如何创建 SSH 密钥的教程）。与本地机器的配置类似，接下来需要对文件夹和站点进行关联映射，并提供一个数据库。

在 Homestead 中进行文件夹映射可以让你编辑本地计算机上的文件，并将这些文件显示在 Vagrant box 中。例如，假设在 `~/Sites` 目录中放置了所有代码，那么可以在 Homestead 中将 `Homestead.yaml` 配置文件中的代码修改为如下所示的代码，从而实现文件夹映射。

```
folders:
  - map: ~/Sites
    to: /home/vagrant/Sites
```

现在已经在 Homestead 虚拟机中创建了一个目录，该目录位于 `/home/vagrant/Sites`，对应着本地计算机的 `~/Sites` 目录。



使用顶级域名进行站点的开发

对于本地开发的网站的 URL，可以使用任何形式，但是 `.app` 和 `.dev` 是最常见的两种形式。Homestead 建议使用 `.app` 的形式。如果正在进行 `symposiumapp.com` 的本地版本的开发，就可以通过 `symposiumapp.app` 网址进行开发以及调试。

现在，建立第一个示例网站。假设实际网站是 `projectName.com`。在 `Homestead.yaml` 中，将映射本地开发目录到 `projectName.app` 中，因此有一个单独的 URL 来访问在本地开发的网站，代码如下。

```
sites:
  - map: projectName.app
    to: /home/vagrant/Sites/projectName/public
```

这里将 URL `projectName.app` 映射到了虚拟机目录 `/home/vagrant/Sites/projectName/public` 中，该文件夹是 Laravel 安装的公共文件夹。

最后，当尝试访问 `projectName.app` 时，需要对本地计算机进行相应配置，比如可以使用本地 IP 地址实现 IP 相关的配置。

Mac 和 Linux 用户可以编辑 `/etc/hosts` 文件，Windows 用户可以编辑 `C:\Windows\System32\drivers\etc\hosts` 文件。在此文件中添加如下所示的一行信息。

```
192.168.10.10 projectName.app
```

一旦设置好 Homestead，就可以通过 `http://projectName.app/` 浏览（在计算机上）网站。

在 Homestead 中创建数据库

就像可以在 `Homestead.yaml` 中定义一个网站一样，还可以定义一个数据库。定义数据库要简单得多，因为只需要告诉系统使用什么名字去创建这个数据库就可以，不需要其他信息。做法如下所示。

```
databases:
  - projectname
```

使用 Homestead

第一次打开 Homestead box 的时候，需要使用 Vagrant 来对它进行初始化。进入 `Homestead` 所在目录并运行 `vagrant up`，如下所示。

```
cd ~/Homestead
vagrant up
```

现在，Homestead box 已经启动并运行了。它将一个本地文件夹与一个 URL 进行映射，可以在电脑上的任何浏览器中访问该 URL。此外，它还创建了一个 MySQL 数据库。现在已经搭建好开发环境，可以建立第一个 Laravel 项目了。在建立项目之前，可以阅读一下关于 Homestead 日常使用的快速说明。

Homestead 的日常使用

通常，Homestead 虚拟机始终保持运行状态，但如果不想要它一直运行，或者如果最近重启了计算机，则需要知道如何开启和关闭 box。

由于 Homestead 是建立在 Vagrant 命令基础上的，所以只需使用基本的 Vagrant 命令就可以进行大多数 Homestead 操作。进入 Homestead 的安装目录（使用 `cd` 命令），然后可以根据需要运行以下命令。

- `vagrant up` 开启 Homestead box。
- `vagrant suspend` 对 box 进行快照（暂停当前环境），然后关闭它，就像台式机的“hibernating（睡眠）”一样。
- `vagrant halt` 关闭整个 box，就像关闭台式机一样。
- `vagrant destroy` 删除整个 box，就像格式化台式机一样。

- `vagrant provision` 有一些预先存在的操作指令，通过 `provision` 任务自动运行这些指令。

在桌面应用程序中连接 Homestead 数据库

如果使用 Sequel Pro 之类的桌面应用程序，则可能会想要连接到 Homestead MySQL 数据库主机中。可以通过以下设置信息实现这个想法。

- **Connection type:** Standard (non-SSH)
- **Host:** 127.0.0.1
- **Username:** homestead
- **Password:** secret
- **Port:** 33060

创建一个新的 Laravel 项目

有两种方法可以创建一个新的 Laravel 项目，但都需要在命令行中运行。第一种方法是全局安装 Laravel 安装工具（使用 Composer），第二种方法是使用 Composer 的 `create-project` 实现。

可以在安装文档页面（<https://laravel.com/docs/5.5/installation>）中更详细地了解这两种方法，但是建议使用 Laravel 安装工具。

使用 Laravel 安装工具来安装 Laravel

如果已经全局安装好 Composer，那么安装 Laravel 安装程序工具就非常简单了，只需要运行以下命令即可。

```
composer global require "laravel/installer=~1.1"
```

一旦安装好 Laravel 安装工具，启动一个新的 Laravel 项目也会变得非常简单。只需在命令行中运行以下命令。

```
laravel new projectName
```

这将会在当前目录下创建一个新的名为 *projectName* 的子目录，并在其中安装一个全新的 Laravel 项目。

通过 Composer 的 create-project 来安装 Laravel

Composer 还提供了一个名为 `create-project` 的命令，用于创建具有特定结构的新项目。要使用此工具来创建一个新的 Laravel 项目，可以输入以下命令。

```
composer create-project laravel/laravel projectName --prefer-dist
```

就像 Laravel 安装工具一样，该操作将会在当前目录下创建一个名为 *projectName* 的子目录，并且其中已经安装好了一个具有特定结构的 Laravel。

Laravel 的目录结构

在打开已经具有特定结构的 Laravel 应用程序的目录时，将看到以下目录和文件。

```
app/  
bootstrap/  
config/  
database/  
public/  
resources/  
routes/  
storage/  
tests/  
vendor/  
.env  
.env.example  
.gitattributes  
.gitignore  
artisan  
composer.json  
composer.lock  
gulpfile.js  
package.json  
phpunit.xml  
readme.md  
server.php
```

让我们逐一进行了解吧。

文件夹

根目录默认包含以下文件夹。

- *app* 应用程序的大部分文件都在这里，比如模型、控制器、路由定义、命令，以及 PHP 域名代码等。
- *bootstrap* 包含了 Laravel 框架每次运行时使用的文件。
- *config* 用于放置所有的配置文件。
- *database* 用于放置数据库迁移和数据库种子文件。

- *public* 是当站点运作时服务器指向的目录，该目录包含 `index.php` 文件，`index.php` 是前端控制器，并且是所有请求的入口文件。该目录下也放置了一些公共（静态资源）文件，如图片、样式表、脚本或下载的文件等。
- *resources* 所需要的非 PHP 的其他脚本文件都会放在这里，比如视图文件、语言文件、Sass / Less（可选）文件，以及 JavaScript 文件等。
- *routes* 是所有路由定义文件所在的位置，其中包括 HTTP 路由、“控制台路由”和 Artisan 命令等。
- *storage* 用于放置缓存、日志和系统编译文件。
- *tests* 用于放置单元测试以及集成测试文件。
- *vendor* 用于放置 Composer 安装的依赖关系文件。它是一个 Git 忽略文件（Git-ignored，忽略掉你在版本控制系统中标记的内容），Composer 将作为部署过程的一部分在任何远程服务器上运行。

文件

根目录还包含以下文件。

- *.env* 和 *.env.example* 是指定环境变量（每个环境中预设着不同的变量，因此不会提交到版本控制）的文件。*.env.example* 是一个模板文件，每个环境中都应该复制该文件内容并创建一个自己的 *.env* 文件，这是一个 Git 忽略（Git-ignored）文件。
- *artisan* 是允许从命令行运行 Artisan 命令（详见第 7 章）的文件。
- *.gitignore* 和 *.gitattributes* 是 Git 配置文件。
- *composer.json* 和 *composer.lock* 是 Composer 的配置文件，*composer.json* 是用户可编辑的，*composer.lock* 是用户不可编辑的。这些文件共享一些与该项目相关的基本信息，并定义其依赖关系。
- *gulpfile.js* 是 Elixir 和 Gulp 的配置文件（可选），用于编译和处理前端资源的。
- *package.json* 类似于 *composer.json* 文件，但是该文件主要用于处理前端资源。
- *phpunit.xml* 是 PHPUnit 的配置文件，PHPUnit 是一款开箱即用的 Laravel 测试工具。
- *readme.md* 是一个 Markdown 文件，这里面主要是一些 Laravel 的基本介绍信息。
- *server.php* 是一个备份服务器文件，它尝试在功能较差的服务器中仍然对 Laravel 应用程序进行预览。

配置

Laravel 应用程序的核心配置信息，比如数据库连接、队列以及邮件设置等，都存放在配置文件夹下的文件中。这里的每一个文件都将返回一个数组，数组中的每个值都可以通

过一个配置键（config key）进行访问，该配置键（config key）由文件名和所有后续的键组成，以点号（.）进行分隔。

所以，可以在 `config/services.php` 中创建如下所示的信息。

```
// config/services.php
return [
    'sparkpost' => [
        'secret' => 'abcdefg'
    ]
];
```

现在可以使用 `config('services.sparkpost.secret')` 访问配置好的变量了。

每个环境中的任何配置变量都应该放在 `.env` 文件中（而不是提交给源代码控制）。如果希望在每个环境使用不同的 Bugsnag API 密钥，可以将配置信息从 `.env` 中提取出来，如下所示。

```
// config/services.php
return [
    'bugsnag' => [
        'api_key' => env('BUGSNAG_API_KEY')
    ]
];
```

这个 `env()` 助手函数可以从 `.env` 文件中提取出该键名所对应的值。现在可以将该键名对应的信息添加到 `.env`（当前环境的设置）和 `.env.example`（适用于所有环境的模板）文件中。

```
BUGSNAG_API_KEY=oinfp9813410942
```

`.env` 文件已包含了很多框架所需的特定环境的变量，例如将会使用到的邮件驱动程序以及基本数据库设置信息等。

启动和运行

现在已经安装好一个全新的 Laravel，并可以启动和运行了。例如可以运行 `git init`，以及通过运行 `git add` 以及 `git commit` 来提交新文件，这样就可以开始编程了。如果使用的是 Valet，那么运行下面的命令，便可以立刻在浏览器中看到网站了。

```
laravel new myProject && cd myProject && valet open
```

每当启动一个新项目的时候，都采取以下这些步骤。

```
laravel new myProject cd myProject
```



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

所有网站都保存在 `~/Sites` 文件夹中，并且将该文件夹设置为 Valet 的主目录，所以在这种情况下，无须进行新增站点操作，便可以立即在浏览器中访问 `myProject.dev` 了。可以编辑 `.env` 并为其设置特定的数据库，同时将该数据库添加到 MySQL 应用程序中，这样一切就绪，就可以开始进行编程了。



Lambo

这是我经常执行的步骤，为此创建了一个简单的全局 Composer 包。它被称为 Lambo，可以通过 GitHub (<https://github.com/tightenco/lambo>) 详细了解。

测试

从第 2 章起，每章末尾的“测试”部分将展示如何为所介绍的功能编写测试文件。由于本章没有可测试的功能，所以先简单介绍一下（如果想了解更多有关在 Laravel 中编写和运行测试的信息，请查看第 12 章）。

开箱即用，Laravel 将 PHPUnit 作为依赖项并配置好，这样在 `tests` 目录下，便可以通过任何以 `Test.php`（例如，`tests/UserTest.php`）结尾的文件实现测试的运行。

所以编写测试的最简单方法是在 `tests` 目录中创建一个文件，并且该文件名称以 `Test.php` 结尾。运行它们的最简单方法是在命令行（在项目根目录中）中运行 `./vendor/bin/phpunit`。

如果某些测试需要通过数据库访问，请确定从数据库服务器所在的机器中进行了测试。所以，如果数据服务器在 Vagrant 中，请确定通过 SSH 进入了 Vagrant box，并在 Vagrant box 中进行测试。本书第 12 章将详细介绍关于测试的更多内容。

本章小结

由于 Laravel 是一款 PHP 开发框架，所以在本地中使用 Laravel 非常简单。此外，Laravel 提供两种用于本地开发管理的工具：一种是名为 Valet 的简单工具，它使用本地机器进行依赖管理；另一种预先配置好 Vagrant 的名为 Homestead 的工具。Laravel 以及相关依赖可以通过 Composer 安装，安装好后，就带有一系列文件夹和文件了。在这些文件夹和文件中体现了一些约定俗成的内容，以及与其他开源工具的关系。

路由和控制器

任何 Web 应用程序框架的基本功能都是接收用户的请求并提供响应，通常这个过程是通过 HTTP 协议进行的。这意味着在学习 Web 框架时，定义好应用程序的路由是第一个也是最重要的一个环节。没有路由，就无法与终端用户进行交互。

在本章中，我们将学习如何查看以及定义 Laravel 中的路由、如何设置路由相关的代码，以及如何使用 Laravel 的路由工具来处理各种路由需求。

路由定义

在一个 Laravel 应用程序中，可以在 *routes/web.php* 中定义 Web 路由，并在 *routes/api.php* 中定义 API 路由。Web 路由是供终端用户进行访问的，如果定义了 API 路由，那么 API 路由主要是用来提供 API 服务的。现在，我们将主要关注 *routes/web.php* 中的路由（即 Web 路由）。



5.3 在 Laravel 5.3 版本之前的项目中，只有一个路由文件，该路由文件位于 *app/Http/routes.php*。

定义路由最简单的方法是将路径（例如 `/`）与闭包结合起来使用，如示例 3-1 所示。

示例3-1 路由的基本定义

```
// routes/web.php
Route::get('/', function () {
    return 'Hello, World!';
});
```

闭包是什么

闭包是 PHP 版本的匿名函数。闭包是一个函数，可以将它作为一个对象传递，并赋值给一个变量，将其作为参数传递给其他的函数和方法，甚至进行序列化。

现在，已经定义好了路由，如果任何人访问 /（域名的根路径），Laravel 的路由就会运行在那里已经定义好的闭包，同时会返回结果。请注意，这里会 `return` 内容，而不是 `echo`（响应）或 `print`（打印）该内容。



关于中间件的快速介绍

读者可能会想：“为什么它会返回 ‘Hello, World!’，而不是输出它”。

这个问题有很多种答案，其中最简单的是，Laravel 的请求和响应过程包含很多封装起来的内容，包括所谓的中间件。仅仅定义好路由闭包以及控制器方法，还不足以将输出发送给浏览器，所以这里采用返回内容的方式，这样返回的内容可以继续 `response stack` 以及中间件中运行（即继续在程序中处理该返回的内容），运行完成后再返回给浏览器供终端用户查看。

许多简单的网站完全可以在 Web 路由文件中进行定义。下面通过一些简单的 GET 路由和一些模板，可以很容易地编写出一个经典的网站，如示例 3-2 所示。

示例3-2 简单的网站

```
Route::get('/', function () {  
    return view('welcome');  
});  
  
Route::get('about', function () {  
    return view('about');  
});  
  
Route::get('products', function () {  
    return view('products');  
});  
  
Route::get('services', function () {  
    return view('services');  
});
```



静态调用

如果有相当丰富的 PHP 的开发经验,可能会惊奇地发现 Route 类中的静态调用。这实际上并不是一个静态的方法,而是使用 Laravel 的 facade 实现的,相关内容将在本书第 11 章中具体讨论。

如果不想使用 facade,那么也可以通过以下方式完成同样的定义。

```
$router->get('/', function () {  
    return 'Hello, World!';  
});
```

HTTP 方法

如果不熟悉 HTTP 方法的概念,则可以在本章中了解更多的信息,但是现在只需要知道每个 HTTP 请求都有一个动词或动作。Laravel 允许根据特定的动词来定义路由,最常见的是 GET 和 POST,然后是 PUT、DELETE 以及 PATCH。每个方法都会将不同的信息传达给服务器,即代码中包含了调用者的意图。

路由动词 (Verb)

你可能已经注意到了,这里一直在使用 `Route::get` 进行路由的定义。这意味着我们会告诉 Laravel,当 HTTP 使用 GET 方法进行请求时,才匹配对应的路由。但是如果对应请求是一个使用 POST 方法的表单请求,或者是一些通过 JavaScript 发送的 PUT 或 DELETE 请求呢?其实,还可以使用一些其他方法来进行路由的定义,如示例 3-3 所示。

示例3-3 路由动词 (Verb)

```
Route::get('/', function () {  
    return 'Hello, World!';  
});  
  
Route::post('/', function () {});  
  
Route::put('/', function () {});  
  
Route::delete('/', function () {});  
  
Route::any('/', function () {});  
  
Route::match(['get', 'post'], '/', function () {});
```

路由处理

将一个闭包传递给路由定义并不是进行路由处理的唯一方法。闭包非常快速、简单,但

是由于闭包会把所有的路由处理信息放在一个文件中，所以应用程序越大，该文件中的路由处理信息就会越多。此外，使用路由闭包的应用程序不能利用 Laravel 的路由缓存，而使用路由缓存可以在每个请求中节省数百毫秒的时间。

另一个常见的方法是将控制器名称和方法作为字符串传递给闭包，如示例 3-4 所示。

示例3-4 路由调用控制器方法

```
Route::get('/', 'WelcomeController@index');
```

这是在告诉 Laravel，将请求传递到应用程序 `index()` 方法中的 `\Http\Controllers\WelcomeController` 控制器。与对应位置上的闭包的处理方式一样，该方法将传递相同的参数，并以相同的方式进行路由处理。

路由参数

如果定义的路由具有参数（可变的 URL 地址段），那么可以在路由中定义它们，并将它们传递给闭包（见示例 3-5）。

示例3-5 路由参数

```
Route::get('users/{id}/friends', function ($id) {  
    //  
});
```

路由参数与闭包 / 控制器方法参数之间的命名关系

从示例 3-5 中可以看到，在路由参数 (`{id}`) 中使用相同的名称，以及将对应的名字添加到路由定义的方法参数中 (`function ($id)`) 是十分常见的。

除非使用路由 / 模型绑定，否则定义的路由参数与哪个方法参数相匹配仅由它们的顺序（从左到右）决定，如以下代码所示。

```
Route::get('users/{userId}/comments/{commentId}', function (  
    $thisIsActuallyTheRouteId,  
    $thisIsReallyTheCommentId  
) {  
    //  
});
```

也就是说，可以让它们使用不同的名称，也可以使用相同的名称。这里建议使它们的名称保持一致，以免未来开发人员在使用的時候可能因为不一致的命名而出現问题。

还可以通过在参数名称后添加一个问号 (?) 来实现路由参数的选择, 如示例 3-6 所示。在这种情况下, 应该为相应的路由变量设置好默认值。

示例3-6 可选路由参数

```
Route::get('users/{id?}', function ($id = 'fallbackId') {  
    //  
});
```

并且可以使用正则表达式来定义一个路由, 这个时候, 只有该参数满足特定的模式时才会匹配, 如示例 3-7 所示。

示例3-7 通过正则表达式来定义路由

```
Route::get('users/{id}', function ($id) {  
    //  
})->where('id', '[0-9]+');
```

```
Route::get('users/{username}', function ($username) {  
    //  
})->where('username', '[A-Za-z]+');
```

```
Route::get('posts/{id}/{slug}', function ($id, $slug) {  
    //  
})->where(['id' => '[0-9]+', 'slug' => '[A-Za-z]+']);
```

如果访问一个与路由字符串匹配的路径, 但正则表达式与该参数不匹配, 那么它将不进行匹配处理。由于路由是从上到下进行匹配的, 在示例 3-7 中, 路径 `users/abc` 将会跳过第一个闭包, 然后由第二个闭包进行匹配, 因此它将会被路由到第二个闭包那里。此外, 路径 `posts/abc/123` 将不与任何闭包匹配, 因此它将会返回 404 Not Found 错误。

路由名称

要在应用程序的其他位置引用这些路由, 最简单的方法就是使用它们的路径。如果需要, 则可以使用 `url()` 助手来简化视图中的链接, 如示例 3-8 所示。`url()` 助手会在路由上补全站点的完整域名。

示例3-8 URL助手

```
<a href="<?php echo url('/'); ?>">  
// 输出 <a href="http://myapp.com/">
```

但是, Laravel 仍然允许为每个路由起一个名字, 这样就可以在不明确引用什么 URL 的情况下引用该路由。这是很有用的, 因为这意味着可以给复杂的路由提供简单的别名,

由于现在是直接通过名称来链接它们，所以意味着即使路径发生变化，也不必重写前端的链接（见示例 3-9）。

示例3-9 定义路由名称

// 在 routes/web.php 中定义具有名称的路由：

```
Route::get('members/{id}', 'MembersController@show')->name('members.show');
```

// 使用 route() 助手在视图中链接路由

```
<a href="<?php echo route('members.show', ['id' => 14]); ?>">
```

这个例子中展示了一些新的内容。首先，使用流畅路由定义（fluent route definition）来添加一个名字，具体方式是通过在 `get()` 方法之后调用 `name()` 方法。这种方式可以给一个路由命名，比如给它起一个简短的别名，这样就可以更方便地在其他位置引用该路由。



在 Laravel 5.1 中自定义路由

5.2 在 Laravel 5.1 中不存在流畅路由定义（fluent route definitions）。如果要实现在 Laravel 5.1 中自定义路由，则需要将数组传递到路由定义的第二参数中，其运作原理可以通过查看 Laravel 文档进行更多了解。以下是在 Laravel 5.1 中实现与示例 3-9 类似功能的程序。

```
Route::get('members/{id}', [
    'as' => 'members.show',
    'uses' => 'MembersController@show'
]);
```

在示例中，给这个路由命名为 `members.show`，Laravel 中路由和视图的命名格式一般是：`resourcePlural.action`。

路由命名规则

可以将路由命名为任何喜欢的名字，但常见的规则是使用复数的资源名称，然后使用一个英文句号（.），最后加上相应的动作。以下是名为 `photo` 的资源最常用的路由命名示例。

```
photos.index
photos.create
```



```
photos.store
photos.show
photos.edit
photos.update
photos.destroy
```

要了解更多关于这些规则的信息，请查看第 40 页关于资源控制器部分的内容。

本书还介绍了 `route()` 方法。就像 `url()` 一样，该方法在视图中使用，目的是简化命名路由的链接。如果路由没有参数，则可以轻松地传递一个路由名称（`route('members.index')`），并且接收路由字符串（`http://myapp.com/members/index`）。如果它有参数，则将这些参数作为一个数组传递给第二个参数，就像我们在这个例子中所做的那样。

一般来说，建议使用路由名称而不是路径来引用路由，使用 `route()`，而不是 `url()`。如果在使用多个子域名的时候，使用这种方式可能会显得有点笨，但这种方式非常灵活，这样后续在更改应用程序的路由结构时，就不会遇到重大问题。

将路由参数传递给 `route()` 助手

当路由有参数的时候（例如 `:users/{id}`），如果使用 `route()` 助手路由绑定一个链接，那么需要定义好这些参数。

传递这些参数有几种不同的方法。假设我们定义一个路由 `users/{userId}/comments/{commentId}`。如果 `user ID` 为 1，`comment ID` 为 2，则以下几种方案可供选择。

方案 1：

```
route('users.comments.show', [1, 2])
// http://myapp.com/users/1/comments/2
```

方案 2：

```
route('users.comments.show', ['userId' => 1, 'commentId' => 2])
// http://myapp.com/users/1/comments/2
```

方案 3：

```
route('users.comments.show', ['commentId' => 2, 'userId' => 1])
// http://myapp.com/users/1/comments/2
```

方案 4 :

```
route('users.comments.show', ['userId' => 1, 'commentId' => 2, 'opt' => 'a'])  
// http://myapp.com/users/1/comments/2?opt=a
```

不是关联数组里面的值会按顺序进行分配；关联数组里面的值会按照对应的键名进行匹配，数组里面其他剩下的数据都会作为查询参数进行添加并使用。

路由组

通常，一组路由会有一些特定的特征：比如，一定的认证要求、路径前缀，或者是控制器与命名空间等。在每个路由上反复定义这些共同的特征不仅会显得很烦琐，还可能会改变路由文件的模型并影响应用程序的某些结构。

路由组允许多个路由组合在一起，并且可以将任何共享的配置应用于整个组，从而减少配置信息的重复。此外，这些路由会被组合在一起，所以路由组也是未来开发者的可视化提示。

要将两个或多个路由组合在一起，需要将路由的定义放在路由组里面，如示例 3-10 所示。实际上，这是将闭包传递给路由组定义，并在闭包中定义分组的路由。

示例3-10 定义一个路由组

```
Route::group([], function () {  
    Route::get('hello', function () {  
        return 'Hello';  
    });  
    Route::get('world', function () {  
        return 'World';  
    });  
});
```

在默认情况下，路由组不会做任何事情。示例 3-10 中的路由组与使用常规代码分隔的路由之间没有区别。但是，路由组中的第一个参数为空数组，该空数组允许传递各种配置信息，这些配置将对该组内的所有路由都生效。

中间件

路由组最常见的功能就是将中间件应用于一组路由中，但是在其他方面，路由组在 Laravel 中也常常被应用在权限控制方面，比如，对用户进行验证以及对访客用户使用网

站的某些内容进行限制等。在第 10 章中会涉及更多中间件的知识。

在示例 3-11 中，将 `dashboard` 与 `account` 组成一个路由组，并且为该路由组建立一个中间件 `auth`，此时该中间件对 `dashboard` 与 `account` 这两者都会生效。在此示例中，表示用户必须登录后才能查看控制中心（`dashboard`）或账户页面（`account`）。

示例3-11 将一组路由限制为只允许登录用户访问

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('dashboard', function () {
        return view('dashboard');
    });
    Route::get('account', function () {
        return view('account');
    });
});
```



中间件在控制器中的应用

通常，在控制器中使用中间件比在路由定义中使用中间件更为清晰和直接。可以在控制器的构造函数中调用 `middleware()` 方法来使用中间件。`middleware()` 方法中的参数代表中间件的名称，可以使用 `modifier` 方法 (`only()` 和 `except()`) 来确定将由哪些方式接收中间件，示例代码如下。

```
class DashboardController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('admin-auth')
            ->only('admin');

        $this->middleware('team-member')
            ->except('admin');
    }
}
```

需要注意的是，如果使用了很多次“only”和“except”方法，这通常意味着应该为个别路由建立一个新的控制器。

路径前缀

如果有一组路由需要共享某个路径段，比如，假设站点的 API 前缀为 `/api`，那么可以使用路由组来简化此结构（见示例 3-12）。

示例3-12 为一组路由设置路径前缀

```
Route::group(['prefix' => 'api'], function () {
    Route::get('/', function () {
        // 设置 path /api
    });
    Route::get('users', function () {
        // 设置 path /api/users
    });
});
```

需要注意的是，每组路径前缀还有一个 / 路由，表示前缀的根，比如示例 3-12 中的路径前缀是 /api。

子域名路由

子域名路由与路由前缀一样，但是作用域不同：子域名路由的作用域是子域名，而不是路由前缀。这主要有两个作用。首先，可以在不同的子域名中展示应用程序中的不同部分（或完全不同的应用程序）。示例 3-13 演示了如何实现这一点。

示例3-13 子域名路由

```
Route::group(['domain' => 'api.myapp.com'], function () {
    Route::get('/', function () {
        //
    });
});
```

其次，可以将子域名的某一部分设置为参数，如示例 3-14 所示。通常，这在多租户技术的案例中用的比较多（例如 Slack 或 Harast，其实每个公司都有自己的子域名，例如 *tighten.slack.co*）。

示例3-14 参数化的子域名路由

```
Route::group(['domain' => '{account}.myapp.com'], function () {
    Route::get('/', function ($account) {
        //
    });
    Route::get('users/{id}', function ($account, $id) {
        //
    });
});
```

需要注意的是，路由组中的任何参数都将通过第一个参数传递到路由组内各路由的方法中。

命名空间前缀

当按照子域名或者路由前缀的方式对路由进行分组时，它们的控制器可能会有相同的 PHP 命名空间。在 API 的示例中，所有的 API 路由的控制器都可能在一个 API 命名空间内。通过使用路由组命名空间前缀（如示例 3-15 所示），就可以避免在群组内使用很长的控制器进行引用，如 "API/ControllerA@index" 以及 "API/ControllerB@index"。

示例3-15 路由组命名空间前缀

```
//App\Http\Controllers\ControllerA
Route::get('/', 'ControllerA@index');

Route::group(['namespace' => 'API'], function () {
    // App\Http\Controllers\API\ControllerB
    Route::get('api/', 'ControllerB@index');
});
```

名称前缀

前缀并不仅局限于此。通常，路由名称会反映路径元素的继承链(inheritance chain)，因此，users/comments/5 将会由名为 users.comments.show 的路由实现。在这种情况下，在 users.comments 资源下面的所有路由中使用路由组是很常见的。

就像可以使用前缀 URL 字段和控制器命名空间一样，也可以在路由名称前面加上前缀字符串。使用路由组名称前缀，可以定义该组中的每个路由，让它们都有一个给定的字符串作为其名称前缀。在这种情况下，我们会将前缀“users.”添加到每个路由名称前，然后添加“comments.”（见示例 3-16）。

示例3-16 路由组名称前缀

```
Route::group(['as' => 'users.', 'prefix' => 'users'], function () {
    Route::group(['as' => 'comments.', 'prefix' => 'comments'], function () {
        // 路由名称将是 users.comments.show
        Route::get('{id}', function () {
            //
        }->name('show'));
    });
});
```

视图

到目前为止，我们已经接触了一些路由闭包，并且看到了 return view('account') 的一些内容。

对于模型 - 视图 - 控制器（MVC）模式，简单地说，视图（或者模板）指的是用于描述某些特定输出应该是什么样的文件。我们看到的视图可能有 JSON、XML 或电子邮件格式的，但 Web 框架中最常见的视图是使用 HTML 进行输出的。

在 Laravel 中，有两种不同的视图可以使用：普通 PHP 或者 Blade 模板（详细的内容可以参考第 4 章）。两种视图的区别在于，使用 PHP 引擎进行处理的文件名形式一般为 *about.php*，使用 Blade 引擎进行处理的文件名的形式一般为 *about.blade.php*。



加载 view() 的三种方法

返回视图有三种不同的方法。第一种方法是直接使用 `view()` 加载；`View::make()` 是加载视图的第二种方法，实际上，这与第一种方法是一样的；还可以使用注入 `Illuminate\View\View Factory` 的方式实现视图的加载，这是第三种方法。

加载视图后，就可以轻松地返回对应的视图了（见示例 3-17），如果视图不依赖于控制器中的任何变量，则可以正常工作。

示例3-17 简单的view()的用法演示

```
Route::get('/', function () {  
    return view('home');  
});
```

此代码可以在 *resources/views/home.blade.php* 或 *resources/views/home.php* 中查找视图、加载其内容，以及解析任何内联 PHP 或控件结构，直到有视图的输出。一旦返回它，它将被传递到响应栈的其余部分，并且最终返回给用户。

但是如果需要传递变量呢？可以看看示例 3-18。

示例3-18 传递变量给视图

```
Route::get('tasks', function () {  
    return view('tasks.index')  
        ->with('tasks', Task::all());  
});
```

这个闭包加载 *resources/views/tasks/index.blade.php* 或 *resources/views/tasks/index.php* 的视图，并且会传递一个名为 `tasks` 的单个变量给视图，该变量包含 `Task::all()` 方法所返回的结果。`Task::all()` 是一个优秀的数据库查询方法，本书第 8 章会具体讲解该方法。

使用视图 Composer 在每个视图中共享变量

有时，一次又一次地传递相同的变量是一件很麻烦的事情。你可能希望存在一个共享变量可以访问站点中的每个视图、某个类别的视图或者某个类别包含的子视图，例如，所有与任务相关或 header 部分的视图。

这个时候可以让某些变量与每个模板（或者某些模板）进行共享，示例代码如下。

```
view()->share('variableName', 'variableValue');
```

要了解更多信息，请查看第 64 页“视图 Composer 和服务注入”的相关内容。

控制器

之前已经多次提到过控制器，但到目前为止，大多数示例使用的都是路由闭包。MVC 模式如图 3-1 所示，简单地说，控制器本质上是在一个位置将一个或多个路由逻辑组织在一起的类。控制器往往会将类似的路由组合在一起，特别是如果应用程序是按照传统的 CRUD 格式构造的，在这种情况下，控制器可以处理那些可对特定资源执行的所有操作。

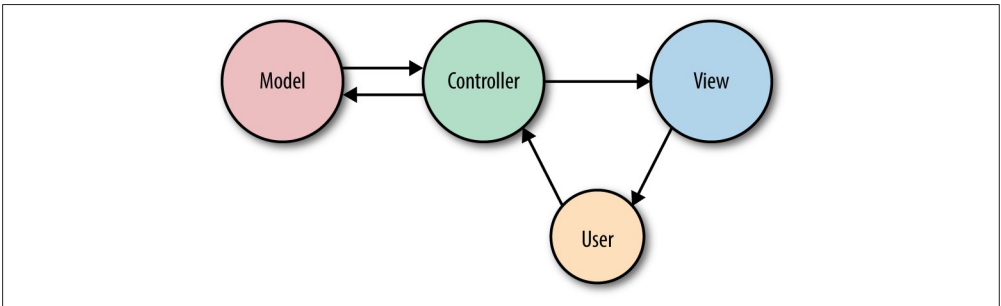


图3-1 MVC的基本说明



CRUD 是什么

CRUD 代表创建、读取、更新、删除，这是 Web 应用程序通常在资源上提供的四个主要操作。例如，我们可以创建一篇新的博客文章，也可以阅读该文章，还可以更新该文章，或者删除该文章。

将所有应用程序的逻辑放到控制器中可能会很有诱惑力，但最好将控制器看成是应用程序路由中的 HTTP 请求流量警报。因为还有其他的方法可以进入应用程序，比如 cron 作业、Artisan 命令行调用、队列作业等，所以明智的做法是不依赖于控制器进行大量的行为处

理（译者注：即业务逻辑处理）。这意味着控制器的主要任务是捕获 HTTP 请求，并将这些请求传递给应用程序的其余部分。

接下来创建一个控制器。创建控制器的一种简单方法是使用工匠（Artisan）命令，因此可以在命令行中运行如下指令。

```
php artisan make:controller TasksController
```



工匠和工匠 generators（Artisan and Artisan generators）

Laravel 附带了一个名为 Artisan 的命令行工具。Artisan 可用于手动运行迁移、创建用户和其他数据库记录，并且可以执行许多其他手动、一次性任务。

在 make 命名空间下，Artisan 提供了为各种系统文件生成框架文件的工具。这就使得我们可以运行 `php artisan make:controller`。

要了解更多关于这方面的知识，可以查看第 7 章。

接下来将在 `app/Http/Controllers` 中创建一个名为 `TasksController.php` 的新文件，内容如示例 3-19 所示。

示例3-19 默认生成的控制器

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

class TasksController extends Controller
{
}
```

修改这个文件，如示例 3-20 所示，创建一个新的名为 `home()` 的公共方法，会在这里返回一些文本。

示例3-20 简单的控制器的例子

```
<?php

use App\Http\Controllers\Controller;

class TasksController extends Controller
```



```
{
    public function home()
    {
        return 'Hello, World!';
    }
}
```

接下来将建立一个路由，如示例 3-21 所示。

示例3-21 为简单的控制器建立路由

```
// routes/web.php
```

```
<?php
```

```
Route::get('/', 'TasksController@home');
```

访问 /，会看到 “Hello, World!” 等信息。

控制器命名空间

例如在示例 3-21 中，我们引用了一个具有完全限定类名的控制器 `App\Http\Controllers\TasksController`，但是只使用了类名。这并不是因为我们可以简单地用类名来引用控制器。相反，当引用控制器时，可以忽略 `App\Http\Controllers\`，默认情况下，Laravel 被设置为在该命名空间中寻找控制器。

这意味着如果有一个具有完全限定类名的控制器 `App\Http\Controllers\API\ExercisesController`，则可以在路由定义中通过 `API\ExercisesController` 将其进行引用。

控制器方法的最常用的方法，如示例 3-22 所示。

示例3-22 通用控制器方法示例

```
// TasksController.php
```

```
...
```

```
public function index()
```

```
{
```

```
    return view('tasks.index')
```

```
        ->with('tasks', Task::all());
```

```
}
```

该控制器方法会加载 `resources/views/tasks/index.blade.php` 或 `resources/ views/tasks/index`。

php 视图，并将其通过一个名为 `tasks` 的变量进行传递，其中包含 `Task::all()` Eloquent 方法的结果。



生成资源控制器

5.3 如果在 Laravel 5.3 发布之前使用过 `php artisan make:controller`，那么可能希望它可以为所有的基本资源路由使用自动生成方法，如 `create()` 与 `update()` 等。如果希望在 Laravel 5.3 中使用这些方法，可以通过在创建控制器时使用 `--resource` 字段来实现，如下所示。

```
php artisan make:controller TasksController --resource
```

获取用户输入

在控制器方法中执行的第二个最常见的操作是从用户那里获取输入内容并对其进行操作。这里介绍了一些新的概念，先看以下示例代码，然后浏览新的内容。

首先，快速地实现绑定，如示例 3-23 所示。

示例3-23 绑定基本的表单操作

```
// routes/web.php
Route::get('tasks/create', 'TasksController@create');
Route::post('tasks', 'TasksController@store');
```

请注意，绑定 `tasks/create` 的 GET 操作（显示表单）和 `tasks/` 的 POST 操作（创建一个新的任务并进行 POST 操作时提交到 `tasks/`）。假设控制器中的 `create()` 方法只显示一个表单，`store()` 方法的使用如示例 3-24 所示。

示例3-24 常见的表单输入控制器方法

```
// TasksController.php
...
public function store()
{
    $task = new Task;
    $task->title = Input::get('title');
    $task->description = Input::get('description');
    $task->save();

    return redirect('tasks');
}
```

该示例使用了 Eloquent 模型和 `redirect()` 函数，稍后再作讨论。这里可以看到现在我们进行了什么操作：创建一个新的 Task，从用户输入中提取数据并进行保存，然后重定

向到显示所有任务 (task) 的页面。

从 POST 中获取用户输入有两种主要方式:这里使用了 Input facade 及请求对象(Request object), 下面将具体讨论这两种情况。



导入 facade

如果使用这些示例中的任何一个, 无论是在控制器中还是任何其他命名空间的 PHP 类中, 可能会发现无法找到 facade 的错误。这是因为它们在根命名空间中可用, 但是在每个命名空间中不一定可用。

因此, 在示例 3-24 中, 我们需要在文件的开始导入 Input facade。有两种方法可以做到这一点: 我们可以导入 `\Input`, 也可以导入 `Illuminate\Support\Facades\Input`。示例代码如下。

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Input;

class TasksController
{
    public function store()
    {
        $task = new Task;
        $task->title = Input::get('title');
        $task->description = Input::get('description');
        $task->save();

        return redirect('tasks');
    }
}
```

可以看到, 这里使用 `Input::get('fieldName')` 获取任何用户提供的信息 (无论是查询参数还是 POST 的值)。所以用户在 “add task” 页面上填写了两个字段: “title” 和 “description”。我们使用 Input facade 检索这两个字段, 将它们保存到数据库中, 然后返回。

将依赖注入控制器

Laravel 的 facade 代表 Laravel 代码库中最有用的类的简单界面, 可以获取当前相关请求、用户输入、会话和缓存等信息。

如果喜欢注入依赖关系，或者想要使用不具有 facade 的服务，则需要找到一些方式将这些类的实例带入控制器。

这是我们第一次接触 Laravel 的服务容器（service container）。如果对它不熟悉，可以把它当成 Laravel 的魔法；如果想了解更多关于实际使用情况的信息，则可以打开第 11 章阅读。

所有的控制器方法（包括构造函数）都是由 Laravel 的容器进行解析的，这意味着所有的指定（type-hint）容器都知道如何解析的信息都将被自动注入。

如果希望拥有一个 Request 对象的实例并且不使用 facade 实现，应该怎么做呢？其实只需要在方法参数中指定（type-hint）Illuminate\Http\Request 就行，如示例 3-25 所示。

示例3-25 通过指定（type-hint）方式实现控制器方法注入

```
// TasksController.php
...
public function store(\Illuminate\Http\Request $request)
{
    $task = new Task;
    $task->title = $request->input('title');
    $task->description = $request->input('description');
    $task->save();

    return redirect('tasks');
}
```

现在已经定义了必须传递给 store() 方法的参数。既然已经指定（type-hint）它了，并且由于 Laravel 知道如何解析这个类名，就会有一个 Request 对象，可以在方法中使用它。没有明确的绑定，没有其他任何信息，它便可以直接通过 \$request 变量进行使用。

此外，这实际上是笔者和其他许多 Laravel 的开发人员更喜欢的、获取用户输入的方式，即注入一个 Request 的实例并从那里读取用户输入，而不是依赖于 Input facade。

资源控制器

有时命名控制器中的方法可能是编写控制器中最困难的部分。幸运的是，Laravel 对传统 REST/CRUD 控制器（在 Laravel 中称为“资源控制器”）的所有路由都有一些约定，另外，它还带有一个开箱即用的开发工具，以及一个方便的路由定义，这样可以一次性绑定好整个资源控制器。

如果要查看 Laravel 资源控制器的一些方法，可以从命令行中创建一个新的控制器，代

码如下所示。

```
php artisan make:controller MySampleResourceController --resource
```

现在打开 `app/Http/Controllers/MySampleResourceController.php`。可以看到它预先填充了很多方法。接下来看看每个方法代表什么。这里将以 Task 为例进行介绍。

Laravel 资源控制器的方法

你可以看到每个控制器方法的 HTTP 动词、URL、控制器方法名称和具体名字。在表 3-1 中展示了 HTTP 动词、URL、控制器方法名称以及其中每个默认方法的具体名字等信息。

表3-1 Laravel资源控制器的方法

HTTP 动词	URL	控制器方法名称	具体名字	描述
GET	tasks	index()	tasks.index	显示所有任务
GET	tasks/create	create()	tasks.create	显示创建任务表单
POST	tasks	store()	tasks.store	从创建任务表单中接受 表单提交
GET	tasks/{task}	show()	tasks.show	显示一个任务
GET	tasks/ {task}/edit	edit()	tasks.edit	编辑一个任务
PUT/PATCH	tasks/{task}	update()	tasks.update	从编辑任务表单中接受 表单提交
DELETE	tasks/{task}	destroy()	tasks.destroy	删除一个任务

绑定资源控制器

这些是在 Laravel 中使用的传统路由名称，而且可以轻易为每个默认路由生成一个资源控制器。幸运的是，我们不必手动为每个控制器方法生成路由；相反，可以使用一种叫作“资源控制器绑定”的方法来解决这个问题，如示例 3-26 所示。

示例3-26 资源控制器绑定

```
// routes/web.php
Route::resource('tasks', 'TasksController');
```

这样可以自动将该资源的所有路由绑定到指定控制器的相应方法上。它也会适当地命名这些路由，例如，tasks 资源控制器上的 `index()` 方法将被命名为 `tasks.index`。



artisan route:list

如果想要知道当前的应用程序有哪些路由，那么有一个工具可以使用，即在命令行中运行 `php artisan route:list`，这样将获得所有可用路由的列表（见图 3-2）。

```
mattstauffer at Cassin in ~/Sites/book-up-and-running
o php artisan route:list
```

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	web
	GET HEAD	api/user		Closure	api,auth:api
	GET HEAD	dogs	dogs.index	App\Http\Controllers\DogsController@index	web
	POST	dogs	dogs.store	App\Http\Controllers\DogsController@store	web
	GET HEAD	dogs/create	dogs.create	App\Http\Controllers\DogsController@create	web
	GET HEAD	dogs/{dog}	dogs.show	App\Http\Controllers\DogsController@show	web
	PUT PATCH	dogs/{dog}	dogs.update	App\Http\Controllers\DogsController@update	web
	DELETE	dogs/{dog}	dogs.destroy	App\Http\Controllers\DogsController@destroy	web
	GET HEAD	dogs/{dog}/edit	dogs.edit	App\Http\Controllers\DogsController@edit	web

图3-2 php artisan route:list 示例

路由模型绑定

最常见的路由模式之一是：任何控制器方法的第一行都会尝试使用给定的 ID 查找资源，如示例 3-27 所示。

示例3-27 获取每个路由的资源

```
Route::get('conferences/{id}', function ($id) {
    $conference = Conference::findOrFail($id);
});
```

Laravel 提供了一个简化这种模式的特性，也就是“路由模型绑定”。这样便可以定义一个特定的参数名称（例如 {conference}），并向路由解析器表明它应该使用该 ID 查找一个 Eloquent 记录，然后将其作为参数传递，而不是仅仅传递该 ID。

有两种路由模型绑定方式：隐式绑定和自定义绑定（或者称之为显式绑定）。

隐式路由模型绑定

使用路由模型绑定的最简单的方法是对该模型唯一的路由参数进行命名操作（例如，将其命名为 \$conference，而不是 \$id），然后在 closure/controller 方法中使用相同的变量名指定（type-hint）该参数。实现起来非常容易，可以看示例 3-28 所示的程序。

示例3-28 使用隐式路由模型绑定

```
Route::get('conferences/{conference}', function (Conference $conference) {
```

```
return view('conferences.show')->with('conference', $conference);
});
```

由于路由参数（{conference}）与方法参数（\$conference）相同，并且方法参数中带有 Conference 模型（Conference \$conference），Laravel 认为这是一种路由模型绑定。每次访问该路由时，应用程序都会假定传入 URL 中 {conference} 部分的任何内容都是一个 ID，然后将生成的模型实例传递给闭包或控制器方法。



为 Eloquent 模型建立路由键

任何时候，通过 URL 段（通常是基于路由模型绑定的）来查找一个 Eloquent 模型，Eloquent 会通过它的主键（ID）查找默认的列。

如果要将 Eloquent 模型更改为使用 URL 查找对应的列，可以向模型中添加一个名为 `getRouteKeyName()` 的方法，代码如下。

```
public function getRouteKeyName()
{
    return 'slug';
}
```

现在，通过类似 `conferences/{conference}` 这样的网址可以得到这个 slug（而不是得到主键 ID），并且执行相应的查找操作。

5.2 在 Laravel 5.2 中添加了隐式路由模型绑定，因此无法在 Laravel 5.1 中访问它。

自定义路由模型绑定

要手动配置路由模型绑定，可以添加如示例 3-29 所示的行到 `App\Providers\RouteServiceProvider` 的 `boot()` 方法中。

示例3-29 添加路由模型绑定

```
public function boot(Router $router)
{
    // 只允许 parent 的 boot() 方法继续运行
    parent::boot($router);

    // 执行绑定
    $router->model('event', Conference::class);
}
```

现在，已经定义好了。当一个路由在名为 {event} 的定义中有一个参数时（见示例 3-30），路由解析器将返回一个带有该 URL 参数 ID 的 `Conference` 类的实例。

示例3-30 使用显式路由模型绑定

```
Route::get('events/{event}', function (Conference $event) {  
    return view('events.show')->with('event', $event);  
});
```

路由缓存

如果希望加快加载时间，则需要查看一下路由缓存。Laravel 的引导程序可以在几十到几百毫秒的时间内解析 `routes/*` 文件，而路由缓存在极大程度上加快了这一过程的速度。

要想对路由文件进行缓存，需要使用所有的控制器和资源路由（没有路由闭包）。如果应用程序没有使用任何路由闭包，则可以运行 `php artisan route:cache`，Laravel 会将 `routes/*` 文件的结果进行序列化。如果想要删除缓存，则可以运行 `php artisan route:clear`。

这里有一个缺点：Laravel 可以将路由与该缓存文件进行匹配，而不会与实际的 `routes/*` 文件进行匹配。可以对这些文件进行很多更改，但是这些更改是不直接生效的，直到再次运行 `route:cache` 时，更改才会生效。这意味着每次做出的更改，都必须重新缓存，这会带来很多困惑。

建议方案：由于 Git 默认会忽略路由缓存文件，所以可以考虑只在生产服务器上使用路由缓存，并在每次部署新代码时运行 `php artisan route:cache` 命令（不管是通过什么部署方式，Git post-deploy、Forge 部署命令、其他部署系统的一部分等都可以）。这样就不会混淆本地开发环境的问题，并且远程环境仍然会从路由缓存中受益。

表单方法欺骗（Form Method Spoofing）

有时，需要手动定义一个表单应该发送的 HTTP 动词。HTML 表单只允许使用 GET 或 POST，所以如果想要使用其他类型的动词，则需要自己指定。

HTTP 动词的介绍

最常见的 HTTP 动词是 PUT 和 DELETE，除此之外，还有 HEAD、OPTIONS、PATCH 等，以及两个在 Web 开发中很少使用的动词：TRACE 和 CONNECT。

通过下面这些简单介绍可以快速了解 HTTP 动词：GET 用于请求一个资源，HEAD 请求相

当于一个头部版本的 GET，POST 用于创建一个资源，PUT 用于覆盖资源，PATCH 用于修改资源，DELETE 用于删除资源，OPTIONS 用于询问服务器在这个 URL 中允许使用哪个动词。

Laravel 中的 HTTP 动词

正如展示的那样，可以使用 `Route::get()`、`Route::post()`、`Route::any()` 或者 `Route::match()` 来决定路由在路由定义中匹配哪些动词，也可以通过 `Route::patch()`、`Route::put()` 及 `Route::delete()` 进行匹配。

但是如何通过 Web 浏览器发送 GET 以外的请求呢？首先，HTML 表单中的 `method` 属性决定了它使用什么 HTTP 动词：比如，表单有一个 GET 方法，它将通过查询参数和 GET 方法进行提交；如果表单有 POST 方法，那么它将通过 post 主体（post body）和 POST 方法进行提交。

JavaScript 框架可以轻松发送其他请求，如 DELETE 和 PATCH 等请求。但是，如果发现自己需要在 Laravel 中使用 GET 或 POST 以外的动词提交 HTML 表单，则需要使用表单方法欺骗（Form Method Spoofing），以 HTML 形式伪装成 HTTP 方法。

在 HTML 表单中进行 HTTP 方法欺骗

为了通知 Laravel 当前正在提交的表单应被视为 POST 以外的其他内容，这时需要添加一个名为 `_method` 的隐藏变量，它的值可以是“PUT”、“PATCH”或“DELETE”，Laravel 将进行匹配和路由。这样，表单提交实际上就好像是使用一个动词进行请求。

由于示例 3-31 中的表单通过 Laravel 的 DELETE 方法进行请求，所以这里将匹配 `Route::delete()` 定义的路由，而不会匹配 `Route::post()` 对应的路由。

示例3-31 表单方法欺骗

```
<form action="/tasks/5" method="POST">
  <input type="hidden" name="_method" value="DELETE">
</form>
```

CSRF 保护

如果尝试在一个 Laravel 应用程序中创建并提交一个表单，包括在示例 3-31 中的表单，那么可能会遇到可怕的 `TokenMismatchException` 错误提示。

在默认情况下，除“只读”路由（使用 GET、HEAD 或 OPTIONS 的路由）外，Laravel 中的所有路由都通过令牌（会存放在名为 `_token` 的输入表单中）进行保护，防止跨站点请求伪造（CSRF）攻击，该令牌会伴随着每个请求进行传递。该令牌在会话开始时生成，每个非只读路由将提交的 `_token` 与会话令牌进行比较，进而确保请求令牌的一致性。



什么是 CSRF

跨站点请求伪造指的是一个网站假装是另一个网站进行请求的方式。跨站点请求伪造的目标是让某个用户可以通过登录用户的浏览器向网站提交表单，从而劫持用户（即该登录用户）访问网站。

防范 CSRF 攻击的最佳方法是保护所有入站路由请求，比如 POST、DELETE 等，同时附带一个令牌信息，这在 Laravel 中是开箱即用的。

这里有两种方法来解决这个问题。首选方法是在每个提交请求中都添加一个 `_token` 输入表单。在 HTML 表单中，这实现起来很简单，如示例 3-32 所示。

示例3-32 CSRF令牌

```
<form action="/tasks/5" method="POST">
  <?php echo csrf_field(); ?>
  <!-- 或者： -->
  <input type="hidden" name="_token" value="<?php echo csrf_token(); ?>" />
</form>
```

在 JavaScript 应用程序中，解决起来可能稍微麻烦些。使用 JavaScript 框架的网站最常见的解决方案是，将令牌存储在每个页面中像 `<meta>` 这样的标签上，代码如下所示。

```
<meta name="csrf-token" content="<?php echo csrf_token(); ?>" id="token">
```

将令牌存储在 `<meta>` 标签中可以轻松地将它绑定到正确的 HTTP 头上，这样便可以在 JavaScript 框架中对所有请求进行全局处理，如示例 3-33 所示。

示例3-33 全局绑定CSRF的header

```
// 在 jQuery 中
$.ajaxSetup({
  headers: {
    'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
  }
});

// 在 Vue 中
Vue.http.interceptors.push((request, next) => {
```

```
request.headers['X-CSRF-TOKEN'] =  
    document.querySelector('#token').getAttribute('content');  
  
    next()  
});
```

Laravel 会根据要求对每个请求的 X-CSRF-TOKEN 进行检查，通过有效令牌（valid token）这种方案可以实现 CSRF 保护。

需要注意的是，如果使用了 Laravel 5.3 的 Vue 引导程序，那么这个示例中 CSRF 的 Vue 部分程序是不需要手动编写的。

重定向

到目前为止，从控制器方法或路由定义中返回的唯一内容就是视图。但是，还可以返回一些其他可以指示浏览器如何操作的信息。

首先介绍重定向。生成重定向有两种常见的方法：重定向全局助手（redirect global helper）和 facade，这里重点介绍重定向全局助手。两者都会创建一个 Illuminate\Http\RedirectResponse 实例，然后在实例上执行一些简单的方法，再返回它。也可以手动进行操作，但是手动操作需要自己做更多的事情。这里可以看到一些可以返回重定向的方法，如示例 3-34 所示。

示例3-34 返回重定向的不同方法

```
// 使用重定向全局助手生成重定向响应  
Route::get('redirect-with-helper', function () {  
    return redirect()->to('login');  
});  
  
// 使用全局助手的快捷方式  
Route::get('redirect-with-helper-shortcut', function () {  
    return redirect('login');  
});  
  
// 使用 facade 生成重定向响应  
Route::get('redirect-with-facade', function () {  
    return Redirect::to('login');  
});
```

请注意，redirect() 助手与 Redirect facade 实现重定向的效果类似，但是重定向助手也有一个快捷方式，也就是将参数直接传递给助手，而不是之后通过链接上相应的方法实现，这是 to() 重定向方法的一种快捷方式。

redirect()->to()

用于重定向的 to() 方法的方法签名 (method signature)，代码如下。

```
function to($to = null, $status = 302, $headers = [], $secure = null)
```

\$to 是一个有效的内部路径，\$status 表示 HTTP 状态（默认为 302 FOUND），\$headers 可以定义要与重定向一起发送的 HTTP 头，而 \$secure 可以选择使用 http 还是 https（通常根据当前的请求 URL 进行设置）。使用 redirect()->to() 的另一个示例，如示例 3-35 所示。

示例3-35 redirect()->to()

```
Route::get('redirect', function () {
    return redirect()->to('home');

    // 或者使用快捷方式

    return redirect('home');
});
```

redirect()->route()

route() 方法与 to() 方法相同，但 route() 方法会指向一个特定的路由名称，而不会指向特定的路径（见示例 3-36）。

示例3-36 redirect()->route()

```
Route::get('redirect', function () {
    return redirect()->route('conferences.index');
});
```

请注意，由于某些路由名称需要参数，因此其参数顺序有所不同。route() 方法第二个参数是新增的，该参数主要对路由参数进行设置，代码如下。

```
function route($to = null, $parameters = [], $status = 302, $headers = [])
```

所以，可以通过示例 3-37 所示的程序使用它。

示例3-37 带有参数的redirect()->route()

```
Route::get('redirect', function () {
    return redirect()->route('conferences.show', ['conference' => 99]);
});
```

redirect()->back()

由于 Laravel 的会话机制实现了一些内置的便利，所以应用程序总会知道用户以前访问过哪些页面。这样会话机制为 `redirect()->back()` 重定向的实现提供了基础保证，该重定向只是将用户重定向到它的来源页面，这里还可以使用一个全局的快捷方式 `:back()` 实现相应的功能。

其他重定向方法

重定向服务还提供一些其他不常用的方法，这些方法仍然是可以用的，如下所示。

- `home()` 重定向到一个名为 `home` 的路由。
- `refresh()` 重定向到当前用户正在使用的页面（类似刷新操作）。
- `away()` 允许不进行默认 URL 验证，并重定向到外部 URL。
- `secure()` 类似于 `to()` 方法，但该方法的 `Secure` 参数默认设置为 `"true"`。
- `action()` 允许链接到控制器和方法，代码：`redirect()->action('MyController@myMethod')`。
- `guest()` 在认证系统内部使用（在第 9 章中详细讨论）。当用户访问路由时，如果没有通过身份验证，则捕获“intended”路由（即目标路由），然后对该用户的访问请求进行重定向（通常重定向到登录页面）。
- `intended()` 也在认证系统内部使用，成功认证后，将会捕获由 `guest()` 方法存储的“intended”URL（即目标 URL），并将该用户的访问请求重定向到目标 URL。

redirect()->with()

在将用户重定向到不同的页面时，通常希望将某些数据与它们一起进行传递。此时可以手动把数据存储到会话中，但是 Laravel 也有一些更方便的实现方法。

最常见的是使用 `with()` 方法来传递关联数组或单个键和值，如示例 3-38 所示。

示例3-38 附带数据的重定向

```
Route::get('redirect-with-key-value', function () {
    return redirect('dashboard')
        ->with('error', true);
});

Route::get('redirect-with-array', function () {
    return redirect('dashboard')
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```



重定向中的方法链（Chaining method）

与其他许多 facade 一样，大多数对重定向 facade（Redirect facade）的调用都可以使用流畅的方法链，如示例 3-38 中 `with()` 方法的调用。可以参考本书第 143 页中的“什么是流畅界面”了解更多的相关内容。

还可以使用 `withInput()` 方法使用户的表单输入重定向，如示例 3-39 所示。在验证错误的情况下，我们通常希望将用户重定向到他刚才输入表单的位置。

示例3-39 表单输入重定向

```
Route::get('form', function () {
    return view('form');
});

Route::post('form', function () {
    return redirect('form')
        ->withInput()
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```

最简单的获得通过 `withInput()` 方法传递 flashed 输入（flashed input）的方法是使用 `old()` 助手，它可以用于获取所有的 old 输入（`old()`），也可以用于获取特定的键值（如 `old('username')`），如果没有对应的值，则第二个参数为默认值。我们通常会在视图中看到 `old()` 方法，可以在“create”和“edit”视图中通过 HTML 来使用 `old()` 方法，代码如下。

```
<input name="username" value="<?=
    old('username', 'Default username instructions here');
?>">
```

对于验证，这里也有一个传递错误及重定向响应的有效方法：`withErrors()`。通过该方法可以传递任何错误的“提供者（provider）”，包括错误字符串、错误数组等，最常见的是本书第 10 章讲解的关于 `Illuminate Validator` 的实例。示例 3-40 是一个 `withErrors()` 使用示例。

示例3-40 带有错误信息的重定向

```
Route::post('form', function () {
    $validator = Validator::make($request->all(), $this->validationRules);

    if ($validator->fails()) {
        return redirect('form')
    }
});
```

```

        ->withErrors($validator)
        ->withInput();
    }
});

```

`withErrors()` 方法会自动将一个名为 `$errors` 的变量在重定向的页面视图中进行共享，以便处理相关需求。



控制器方法中的 `validate()` 快捷方式

示例 3-40 是怎样的呢？如果在控制器中定义路由，就会有一个简单而强大的工具清理代码。读者可以在本书第 99 页的“在控制器中使用 `ValidatesRequests` 的 `validate()` 方法”中了解更多相关信息。

中止请求

除返回视图和重定向外，最常见的退出路由的方法是中止请求。有一些全局可用的方法（`abort()`、`abort_if()` 及 `abort_unless()`），这些方法可以将 HTTP 状态码、消息和请求头数组（`headers array`）作为参数。

如示例 3-41 所示，`abort_if()` 和 `abort_unless()` 会获取第一个参数，该参数会根据访问的真实情况进行判断，并根据结果执行中止操作。

示例 3-41 403 Forbidden 中止

```

Route::post('something-you-cant-do', function (Illuminate\Http\Request) {
    abort(403, 'You cannot do that!');
    abort_unless($request->has('magicToken'), 403);
    abort_if($request->user()->isBanned, 403);
});

```

自定义响应

在学习了视图、重定向和中止之后，再来看看最常见的响应。就像使用重定向一样，也可以使用 `response()` 助手或 `Response facade` 来运行这些方法。

`response()->make()`

如果想要手动创建 HTTP 响应，只需将数据传递给 `response()->make()` 的第一个参数，例如：`return response()->make('Hello, World!')`。此外，第二个参数是 HTTP 状态代码，第三个参数是 header 头信息。

response()->json() 和 ->jsonp()

如果想要手动创建 JSON 编码的 HTTP 响应，则将可以进行 JSON 编码操作的内容（比如数组、集合或其他任何内容）传递给 `json()` 方法，例如：`return response()->json(User::all())`。与 `make()` 方法不同，`json-encodes` 可以将内容编码成 JSON 格式，并设置相应的 header 头信息。

response()->download() 和 ->file()

如果要给终端用户发送需要下载的文件，可以将 `SplFileInfo` 实例或字符串文件名传递给 `download()` 方法，并且第二个参数可以设置文件名信息，例如：`return response()->download('file501751.pdf', 'myFile.pdf')`。

如果要在浏览器中显示相同的文件（这些文件是 PDF、图像或者浏览器可以处理的其他内容），可以使用 `response()->file()` 实现，两者参数的使用方式是一样的。

测试

在其他一些社区中，通常使用单元测试控制器方法进行测试操作，但是在 Laravel（以及大多数 PHP 社区）中，最常见的是依赖应用程序测试来测试路由的功能。

例如，为了验证 POST 路由是否正确，可以编写一个测试，如示例 3-42 所示。

示例3-42 编写一个简单的POST路由测试

```
// AssignmentTest.php
public function test_post_creates_new_assignment()
{
    $this->post('/assignments', [
        'title' => 'My great assignment'
    ]);

    $this->seeInDatabase('assignments', [
        'title' => 'My great assignment'
    ]);
}
```

这里是否直接调用了控制器方法？不是的，但是这里需要确保这个路由的目标得到满足，该路由的目标是接收一个 POST 请求并将它的重要信息保存到数据库中。

还可以使用类似的语法来访问路由，并验证某些文本是否按要求显示在页面上了，或者单击某些按钮执行操作（见示例 3-43）。

示例3-43 编写一个简单的GET路由测试

```
// AssignmentTest.php
public function test_list_page_shows_all_assignments()
{

    $assignment = Assignment::create([
        'title' => 'My great assignment'
    ]);

    $this->visit('assignments')
        ->see(['My great assignment']);
}
```

本章小结

Laravel 的路由会在 *routes/web.php* 和 *routes/api.php* 中进行定义，可以在其中定义每个路由的目标路径、哪些字段是静态的、哪些字段是参数、哪些 HTTP 动词可以访问路由，以及如何进行路由处理。还可以将中间件附加到路由中，并进行分组，也可以给它们命名。

从路由关闭或控制器方法中返回的内容代表了 Laravel 对用户的响应。如果是字符串或视图，则会呈现给用户；如果是其他类型的数据，则会转换为 JSON 并呈现给用户；如果是重定向，则会强制进行重定向。

Laravel 提供一系列工具和便利的方法，以简化与路由相关的常见任务和结构。这一系列的工具和便利的方法包括资源控制器、路由模型绑定和表单方法欺骗等。

Blade模板

与大多数其他后端语言相比，PHP 作为模板语言的能力是相对比较好的。但是它也存在缺点，而且在使用 `<?php` 内嵌的位置也很难看，因此其实可以使用大多数现代框架来提供模板语言。

Laravel 提供了一个定制的模板引擎，叫作 Blade，其设计灵感来自于 .NET 的 Razor 引擎。它具有简洁的语法、清晰的学习路径和强大直观的继承模型，并且该引擎易于扩展。

快速了解如何编写 Blade，可以查看示例 4-1。

示例4-1 Blade示例

```
<h1>{{ $group->title }}</h1>
{!! $group->heroImageHtml() !!}

@forelse ($users as $user)
    • {{ $user->first_name }} {{ $user->last_name }}<br>
@empty
    No users in this group.
@endforelse
```

Blade 引入了一种约定，在这种约定中，称为“指令”的自定义标签前缀为 @。我们可以为所有控件结构、继承和任何想要添加的自定义功能使用指令。

Blade 的语法是非常简洁的，所以它的核心是比替代品更加简洁。但是，当需要使用嵌套继承模板、复杂条件或递归时，Blade 开始真正体现出它的价值。就像最好的 Laravel 组件一样，在应对复杂的应用程序需求的时候，Blade 可以让相关的应用程序更容易实现及访问。

另外，因为所有 Blade 语法都被编译成正常的 PHP 代码然后缓存，所以它的速度很快，并且允许在 Blade 文件中使用原生 PHP 代码（如果需要）。但是在条件允许的情况下，

建议避免在 Blade 中使用原生 PHP 代码，因为如果执行不能使用 Blade 或自定义 Blade 指令实现的操作，那么这些操作实际上并不属于模板。



在 Laravel 中使用 Twig 模板引擎

与许多其他基于 Symfony 的框架不同，Laravel 默认不使用 Twig 模板引擎。如果喜欢使用 Twig，那么可以使用一个名为 Twig Bridge (<https://github.com/rcrowe/TwigBridge>) 的软件包实现，该软件包可以轻松使用 Twig，而不是使用 Blade。

输出数据

在示例 4-1 中可以看到，`{{ 和 }}` 用于包含 PHP 中输出部分的内容。`{{ $variable }}` 与普通 PHP 中的 `<?= $variable ?>` 类似。

但是在某种程度上，二者也存在区别：Blade 在默认情况下会使用 PHP 的 `htmlspecialchars()` 来转义所有输出的信息，以保护用户免受恶意脚本插入的攻击。这意味着 `{{ $variable }}` 在功能上相当于 `<?= htmlspecialchars($variable) ?>`。如果想进行没有转义的输出，则可以使用 `{!! 和 !!}` 实现。

使用前端模板框架时的 `{{ 和 }}`

你可能已经注意到，Blade (`{{ }}`) 的 `echo` 语法与许多前端框架的 `echo` 语法具有相似之处。那么，Laravel 怎么知道写的是“Blade”还是“Handlebars”呢？

Blade 将忽略任何以 `@` 开头的 `{{`。所以，Blade 将解析下面的第一个示例，但第二个将直接输出。

```
// 解析为 Blade，$bladeVariable 的值输出到视图
{{ $bladeVariable }}

// @被删除，"{{handlebarsVariable}}" 直接输出到视图
@{{ handlebarsVariable }}
```

控制结构

大家非常熟悉 Blade 中的大多数控制结构。许多人都会直接输出 PHP 中相同 tag 标签的名称和结构。

可以使用一些方便的助手工具 (helper)，但总的来说，控制结构会比直接在 PHP 中使用助手工具更便捷。

条件语句

首先，来看看控制结构的逻辑。

@if

Blade 的 `@if ($condition)` 会编译成 `<?php if ($condition): ?>`。同样，`@else`、`@elseif` 和 `@endif` 也会编译成与 PHP 具有完全相同语法的语句（见示例 4-2）。

示例4-2 @if、@else、@elseif和@endif

```
@if (count($talks) === 1)
    There is one talk at this time period.
@elseif (count($talks) === 0)
    There are no talks at this time period.
@else
    There are {{ count($talks) }} talks at this time period.
@endif
```

就像使用原生 PHP 的条件结构语句一样，可以自由搭配或组合需要的条件控制结构语句。这些语句没有任何特殊的逻辑，同时会有一个解析器去寻找一些形如 `@if ($condition)` 的代码段，并将其替换成对应的 PHP 代码。

@unless 和 @endunless

`@unless` 是 PHP 中没有直接等价语句的一种新语法结构。它与 `@if` 的含义相反。`@unless ($condition)` 与 `<?php if (!$condition)` 的含义相同吗？具体可以参考示例 4-3。

示例4-3 @unless 和 @endunless

```
@unless ($user->hasPaid())
    You can complete your payment by switching to the payment tab.
@endunless
```

循环语句

接下来了解循环语句。

@for、@foreach 和 @while

在 Blade 中，`@for`、`@foreach` 和 `@while` 的运作方式与原生 PHP 中的运作方式是一样的，

具体可以参考示例 4-4、示例 4-5 和示例 4-6。

示例4-4 @for和@endfor

```
@for ($i = 0; $i < $talk->slotsCount(); $i++)  
    The number is {{ $i }}<br>  
@endfor
```

示例4-5 @for和@endfor

```
@foreach ($talks as $talk)  
    • {{ $talk->title }} ({{ $talk->length }} minutes)<br>  
@endforeach
```

示例4-6 @while和@endwhile

```
@while ($item = array_pop($items))  
    {{ $item->orSomething() }}<br>  
@endwhile
```

@forelse

@forelse 是在迭代的对象为空的时候，允许在编程中进行处理的一种 @foreach 语句结构。在本章开始就看到过它，示例 4-7 是另一个关于 @forelse 应用的例子。

示例4-7 @forelse

```
@forelse ($talks as $talk)  
    • {{ $talk->title }} ({{ $talk->length }} minutes)<br>  
@empty  
    No talks this day.  
@endforelse
```



@foreach 和 @forelse 中的 \$loop

E3 在 Laravel 5.3 中，@foreach 和 @forelse 指令添加了一个在 PHP 的 foreach 循环中没有的特性：\$loop 变量。在 @foreach 或 @forelse 循环中使用此变量，此变量将返回具有以下属性的 stdClass 对象。

index

循环中当前项目基于 0 的索引，0 意味着“第一项”。

iteration

循环中当前项目基于 1 的索引，1 意味着“第一项”。

remaining

循环中剩余的项还有多少；如果一共有 3 项，当前是第 1 项，那么 **remaining** 将是 2。

count

循环中的项目数。

first

一个布尔值，表示这是否是循环中的第一个项目。

last

一个布尔值，表示这是否是循环中的最后一个项目。

depth

这个循环有多少层：1 为一层循环，2 为循环中还有一层循环等。

parent

如果此循环在另一个 `@foreach` 循环中，引用父循环项的 `$loop` 变量，否则为 `null`。

以下是一个使用示例。

```
<ul>
  @foreach ($pages as $page)
    <li>{{ $loop->iteration }}: {{ $page->title }}
      @if ($page->hasChildren())
        <ul>
          @foreach ($page->children() as $child)
            <li>{{ $loop->parent->iteration }}.
              {{ $loop->iteration }}:
              {{ $child->title }}</li>
          @endforeach
        </ul>
      @endif
    </li>
  @endforeach
</ul>
```

or

如果不确定是否设置了一个变量，你可能习惯在使用之前先用 `isset()` 检查它，然后输出该变量；如果没有设置变量，那么它可以输出其他内容。Blade 有一个便利的助手工具，**or**，也可以这样做：在对应位置编写 `{{ $title or "Default" }}`，将会首先尝试输出 `$title` 的值（如果设置了 `$title` 变量）；如果没有设置 `$title` 变量，则输出 “Default”。

模板继承

Blade 为模板继承提供了一种结构，允许扩展、修改视图和包含其他视图。

下面介绍如何使用 Blade 进行继承。

用 @section/@show 和 @yield 定义 Section

首先从一个顶层的 Blade 布局开始介绍，如示例 4-8 所示。这是一个通用页面包装器的定义，稍后会将页面特定内容放入该通用页面包装器中。

示例4-8 Blade布局

```
<!-- resources/views/layouts/master.blade.php -->
<html>
  <head>
    <title>My Site | @yield('title', 'Home Page')</title>
  </head>
  <body>
    <div class="container">
      @yield('content')
    </div>
    @section('footerScripts')
      <script src="app.js"></script>
    @show
  </body>
</html>
```

这看起来像一个普通的 HTML 页面，但是可以看到这里有两处（`title` 和 `content`）使用了 `yield`，并且我们已经在第三处（`footerScripts`）中定义了一个 `section`。

这里有三个 Blade 指令，每个都不同：`@yield('title', 'Home Page')` 单独定义一个变量；`@yield('content')` 定义一个默认名称；`@section ...@show` 会把实际内容放在其中。

这三个函数在本质上是是一样的。这三个函数都定义了一个具有给定名称的部分（即第一个参数）；这三个函数都代表定义的部分可以扩展；这三个函数都定义了在这个部分没有被扩展时的做法：例如可以通过提供一个字符串进行展示（`'Home Page'`）；当然也可以没有展示（如果没有扩展的话，将不会显示任何内容）；也可以展示一个完整的块（比如在本例中，通过 `<script src="app.js"></script>` 展示了一个完整的块）。

有什么不同？很明显，`@yield('content')` 没有定义默认内容（只有默认名称）。此外，`@yield('title')` 中默认的内容只在该部分未被扩展时才会显示出来；如果该部分被扩展，它的子部分将不会访问设置的默认值。而 `@section ... @show` 既定义了默认值，同时也通过 `@parent` 将其默认内容提供了子部分。

一旦建立了这样的父级布局，便可以像示例 4-9 那样进行扩展。

示例4-9 扩展Blade布局

```
<!-- resources/views/dashboard.blade.php -->
@extends('layouts.master')

@section('title', 'Dashboard')

@section('content')
    Welcome to your application dashboard!
@endsection

@section('footerScripts')
    @parent

    <script src="dashboard.js"></script>
@endsection
```



@show 与 @endsection

你可能已经注意到，示例 4-8 使用的是 `@section ... @show`，但示例 4-9 使用的是 `@section ... @endsection`。二者有什么不同呢？

在同一模板中定义 section 的位置时，使用 `@show`；在子模板中定义模板的内容时，则使用 `@endsection`。

这个子视图有一些关于 Blade 继承的新概念，下面将进行具体介绍。

@extends

首先，使用 `@extends('layouts.master')`，这里定义的这个视图不会自己呈现，而是会扩展另一个视图。这意味着它的作用是定义不同部分（section）的内容，而不是一个独立的页面。它更像是一系列内容的桶，而不是一个 HTML 页面。这一行还定义了它会在 `resources/views/layouts/master.blade.php` 上扩展视图。

每个文件只能扩展一个其他文件，如果要使用 `@extends` 进行扩展，那么应该在该文件

的第一行进行定义。

@section 与 @endsection

其次,使用 `@section('title', 'Dashboard')`, 可以为第一部分 (title) 提供内容。由于内容太短, 所以不使用 `@section` 和 `@endsection` 部分, 这里只使用快捷方式进行。可以以 `@section` 的第二个参数传递内容, 然后继续进行后续事项。如果因为看到 `@section` 却没有看到 `@endsection` 而不习惯, 也可以使用正常的语法进行。

第三,使用 `@section('content')` 和 `on`, 以正常语法来定义 `content` 部分的内容。请注意, 当在子视图中使用 `@section` 时, 结束时可以使用 `@endsection` (或其别名, 如 `@stop`) 进行, 而不是使用 `@show` 进行的, `@show` 用于在父视图中定义 `section`。

@parent

第四, 使用 `@section('footerScripts')` 和 `on`, 以正常语法定义 `footerScripts` 部分的内容。

请记住, 这里实际上已经在主布局中定义了内容 (或者至少是它的默认值)。所以这次有两个选择: 从父视图获取内容, 或者自行添加它。

可以看到, 这里通过在 `section` 中使用 `@parent` 指令, 将内容从父项中包含进来。如果不这样做, 那么本部分 (节) 的内容将会完全覆盖在本部分 (节) 的父项中定义的任何内容。

@include

介绍了关于继承的基础知识后, 接下来介绍一些使用技巧。

如果在视图中想要拉出另一个视图, 那么应该怎么做呢? 假设有一个“注册”按钮, 如果想在网站上重新使用该按钮, 则每次使用它的时候都要自定义该按钮文本, 如示例 4-10 所示。

示例4-10 使用 `@include` 包含视图部分

```
<!-- resources/views/home.blade.php -->
<div class="content" data-page-name="{{ $pageName }}">
    <p>Here's why you should sign up for our app: <strong>It's Great.</strong></p>

    @include('sign-up-button', ['text' => 'See just how great it is'])
</div>
```

```
<!-- resources/views/sign-up-button.blade.php -->
<a class="button button--callout" data-page-name="{{ $pageName }}">
    <i class="exclamation-icon"></i> {{ $text }}
</a>
```

`@include` 拉入了 `sign-up-button` 部分，并将数据传递给它（可选）。请注意，我们不仅可以通过 `@include` 的第二个参数将数据显式传递给包含的部分，还可以引用包含文件中可用于包含视图的任何变量（在此示例中为 `$pageName`）。虽然我们可以实现任何想要实现的内容，但是建议明确地传递想要使用的每个变量。

@each

可以想象在某些情况下，需要对数组或集合循环，并且通过 `@include` 包含每个项目 (item) 的部分。这里有一个指令可以实现该要求：`@each`。

假设有一个由模块组成的侧边栏，包含多个模块，每个模块都有不同的标题，如示例 4-11 所示。

示例 4-11 通过 `@each` 进行视图部分循环

```
<!-- resources/views/sidebar.blade.php -->
<div class="sidebar">
    @each('partials.module', $modules, 'module', 'partials.empty-module')
</div>

<!-- resources/views/partials/module.blade.php -->
<div class="sidebar-module">
    <h1>{{ $module->title }}</h1>
</div>

<!-- resources/views/partials/empty-module.blade.php -->
<div class="sidebar-module">
    No modules :(
</div>
```

可以思考一下 `@each` 的语法。第一个参数是视图部分的名称；第二个参数是要遍历的数组或集合；第三个参数是每个项目（在这种情况下，指的是 `$modules` 数组中的每个元素）将被传递到视图的变量名称；可选的第四个参数是显示数组或集合是否为空的视图（也可以在此处传递将要作为模板的字符串）。

视图 Composer 和服务注入

正如本书第 3 章所介绍的，从路由定义传递数据到视图很简单（见示例 4-12）。

示例4-12 关于将数据传递给视图的提示

```
Route::get('passing-data-to-views', function () {  
    return view('dashboard')  
        ->with('key', 'value');  
});
```

有时会发现自己在多个视图中传递相同的数据。或者发现自己使用的是 header 部分或类似的信息，这里需要一些数据。必须从每个可能加载该 header 部分的路由定义中传递这些数据吗？

用视图 Composer 绑定数据到视图

幸运的是，有一个更简单的方法。该解决方案被称为视图编辑器，它允许在定义特定视图加载时（任何时候），将某些数据直接传递给它，而不需要显式地将该数据传递给路由定义。

假设在每个页面上都有一个侧边栏，它在名为 `partials.sidebar` (*resources/views/partials/sidebar.blade.php*) 的部分中定义，然后包含在每个页面上。此侧边栏显示在网站上最近发布的 7 个帖子列表。在每个页面上，每个路由定义通常都必须获取该列表并传入，如示例 4-13 所示。

示例4-13 在每个路由中传递侧边栏数据

```
Route::get('home', function () {  
    return view('home')  
        ->with('posts', Post::recent());  
});  
  
Route::get('about', function () {  
    return view('about')  
        ->with('posts', Post::recent());  
});
```

这看起来很麻烦。如果使用视图共享（composer）该变量与一组给定的视图集，则会比较方便。可以通过几种方法来实现，这里先从简单的方法开始介绍。

全局共享变量

首先，最简单的方法如示例 4-14 所示，只需在应用程序中的每个视图中全局共享一个变量。

示例4-14 全局共享变量

```
// 服务提供者
public function boot()
{
    ...
    view()->share('posts', Post::recent());
}
```

如果使用 `view()->share()`，最好是放在服务提供者的 `boot()` 方法中，以便可以绑定在每个页面加载时运行。也可以创建一个自定义的 `ViewComposerServiceProvider`（有关服务提供者的更多信息，请参阅本书第 11 章），但现在只需将它放在 `boot()` 方法的 `App\Providers\AppServiceProvider` 中即可。

使用 `view()->share()` 可以使整个应用程序中的每个视图都可以访问该变量，所以它可能会被过度使用。

基于闭包的视图 composers (Closure-based view composer)

下一个方案使用基于闭包的视图 composer，以实现单个视图共享变量，如示例 4-15 所示。

示例4-15 创建一个基于闭包的视图composer

```
view()->composer('partials.sidebar', function ($view) {
    $view->with('posts', Post::recent());
});
```

可以看到，这里已经在第一个参数（`partials.sidebar`）中定义了希望共享的视图名称，然后将闭包传递给第二个参数。在这个闭包中，使用 `$view->with()` 来共享一个变量，但是仅限于一个特定的视图。



多个视图的视图 composer

任何一个视图 composer 都要绑定一个特定的视图（在示例 4-15 中，绑定了 `partials.sidebar`），如果有多个视图，则可以传递视图名称数组，而不是绑定多个视图。

也可以在视图路径中使用星号，如 `partials.*`，`tasks.*` 或仅仅只使用 `*`，如下所示。

```
view()->composer(
    ['partials.header', 'partials.footer'],
    function () {
        $view->with('posts', Post::recent());
    }
);
```

```
view()->composer('partials.*', function () {  
    $view->with('posts', Post::recent());  
});
```

基于类的视图 composer

最后，最灵活的并且最复杂的方案是为视图 composer 创建一个专门的类。

首先创建视图 composer 类。这里没有严格规定定义视图 composer 类的位置，但文档推荐在 `App\Http\ViewComposers` 中进行。所以，接下来创建 `App\Http\ViewComposers\RecentPostsComposer`，如示例 4-16 所示。

示例4-16 视图composer

```
<?php  
  
namespace App\Http\ViewComposers;  
  
use App\Post;  
use Illuminate\Contracts\View\View;  
  
class RecentPostsComposer  
{  
  
    private $posts;  
  
    public function __construct(Post $posts)  
    {  
        $this->posts = $posts;  
    }  
  
    public function compose(View $view)  
    {  
        $view->with('posts', $this->posts->recent());  
    }  
}
```

这里注入 `Post` 模型（视图 composer 的指定（type-hint）构造函数的参数将被自动注入，有关容器和依赖注入的更多信息，请参见第 11 章）。请注意，这里可以跳过 `private $posts` 和构造函数注入，需要时可以在 `compose()` 方法中使用 `Post::recent()`。这样在 composer 被调用时，会运行 `compose()` 方法，在该方法中将 `posts` 变量与运行 `recent()` 方法的结果绑定在一起。

像共享变量的其他方法一样，视图 composer 需要在某处有一个绑定。之前可能会创建一个自定义 `ViewComposerServiceProvider`；但是现在可以将其放在 `App\Providers\AppServiceProvider` 的 `boot()` 方法中，如示例 4-17 所示。

示例4-17 在AppServiceProvider中注册视图composer

```
// AppServiceProvider
public function boot()
{
    ...

    view()->composer(
        'partials.sidebar',
        \App\Http\ViewComposers\RecentPostsComposer::class
    );
}
```

请注意，该绑定与基于闭包的视图 composer 相同，但不是传递闭包，而是传递视图 composer 的类名。在每次 Blade 渲染 `partials.sidebar` 视图时，它会自动运行提供者，并将一个 `posts` 变量的视图传递给 `Post` 模型中 `recent()` 方法的结果。

Blade 服务注入

这里主要有三种类型的数据可能注入视图：要迭代的数据集合、在页面上显示的单个对象，以及生成数据或视图的服务。

通过使用服务，该模式很可能类似于示例 4-18，其中将分析服务的实例注入路由定义，并将其指定（type-hint）到路由的方法签名中，然后将其传递到视图中。

示例4-18 通过路由定义构造函数将服务注入视图

```
Route::get('backend/sales', function (AnalyticsService $analytics) {
    return view('backend.sales-graphs')
        ->with('analytics', $analytics);
});
```

就像视图 composers 一样，Blade 的服务注入提供了一个方便的快捷方式，以减少路由定义中的重复。使用分析服务的视图内容如示例 4-19 所示。

示例4-19 在视图中使用注入的导航服务

```
<div class="finances-display">
    {{ $analytics->getBalance() }} / {{ $analytics->getBudget() }}
</div>
```

Blade 服务注入可以很容易地从视图中直接插入一个类的实例，如示例 4-20 所示。

示例4-20 直接向视图中注入服务

```
@inject('analytics', 'App\Services\Analytics')

<div class="finances-display">
    {{ $analytics->getBalance() }} / {{ $analytics->getBudget() }}
</div>
```

可以看到，@inject 指令实际上已经提供了一个 \$analytics 变量，后面将在视图中使用。

@inject 的第一个参数主要表示要注入的变量名称，第二个参数表示要注入实例的类或接口。如果不熟悉工作原理，请参阅第 11 章了解更多信息。

例如视图 composers，Blade 服务注入使得我们可以很容易地为视图的每个实例提供特定的数据或功能，而不必每次都通过路由定义注入它。

自定义 Blade 指令

到目前为止，我们所提到的所有内置的 Blade 语法 (@if、@unless 等) 都被称为指令。每个 Blade 指令 (如 @if (\$condition)) 都会和 PHP 输出 (如 <?php if (\$condition): ?>) 通过一个模式进行映射。

指令不仅包含这些核心指令，实际上也可以自己创造指令。可能有人会认为，指令为较大的代码段提供了一些非常有用的快捷方式，例如使用 @button('buttonName')，并将其扩展到更大的一组按钮 HTML 集。这个注意不赖，但是对于简单的代码扩展来说，可能用包含一个视图部分来解决该问题会更好。

自定义指令在简化某些形式的重复逻辑时是最有用的。例如不想用 @if (auth()->guest()) (检查用户是否登录) 来包装代码，那么可以用一个自定义的 @ifGuest 指令。与视图 composer 一样，可能需要一个自定义服务提供者来注册这些，但现在将它放在 App\Providers\AppServiceProvider 的 boot() 方法中进行。可以通过示例 4-21 看看这个绑定是什么样的。

示例4-21 绑定自定义Blade指令

```
// AppServiceProvider
public function boot()
{
    Blade::directive('ifGuest', function () {
```



```

        return "<?php if (auth()->guest()): ?>";
    });
}

```

现在已经注册了一个自定义指令 `@ifGuest`，它将被替换为 PHP 代码 `<?php if (auth()->guest()): ?>`。

这可能比较奇怪——正在编写一个字符串，它将被返回，然后作为 PHP 执行。但是，这意味着可以利用 PHP 模板代码复杂、难看、不明确或重复的方面，并将它们隐藏在清晰、简单又有表现力的语法后面。



自定义指令结果缓存

也许可以尝试执行一些逻辑以使自定义指令执行速度更快，通过在绑定中执行操作，然后将结果嵌入返回的字符串。

```

Blade::directive('ifGuest', function () {
    // 反模式，不要模仿
    $ifGuest = auth()->guest();
    return "<?php if ({$ifGuest}): ?>";
});

```

这个想法的问题在于，它假定在每个页面加载时都会重新创建该指令。但是 Blade 的缓存十分高效，所以如果尝试这样做，将会发现自己处于不利地位。

自定义 Blade 指令中的参数

如果想要检查自定义逻辑中的条件，该怎么办呢？可以参考示例 4-22。

示例4-22 创建带有参数的Blade指令

```

// 绑定
Blade::directive('newlinesToBr', function ($expression) {
    return "<?php echo nl2br({$expression}); ?>";
});

```

```

// 使用
<p>@newlinesToBr($message->body)</p>

```

闭包接收的 `$expression` 参数表示括号内的内容。这里将生成一个有效的 PHP 代码片段并返回它。



在 Laravel 5.3 之前的版本中 `$expression` 参数的范围

`<?php echo nl2br{$expression}; ?>`。在 Laravel 5.3 之前的版本中，还包括 `$expression` 参数括号。所以，在示例 4-22 中，`$expression`（在 Laravel 5.3 及更高版本中为 `$message->body`）会被改为 `($message->body)`，而且必须要写 `<?php echo nl2br{$expression}; ?>`。

如果发现自己不断在编写相同的条件逻辑，那么应该考虑创建一个 Blade 指令。

示例：对多租户应用程序（Multitenant App）使用自定义 Blade 指令

假设我们正在构建一个支持多租户的应用程序，这意味着用户可能会从 `www.myapp.com`、`client1.myapp.com`、`client2.myapp.com` 或其他地址访问该网站。

假设已经编写了一个类来封装一些多租户逻辑，并将其命名为 `Context`。该类将捕获关于当前访问的上下文信息和逻辑，例如被认证的用户是谁，以及用户访问的是公共网站还是客户端子域。

我们可能会经常在视图中解析这个 `Context` 类并在上面执行相应条件，如示例 4-23 所示。`app('context')` 是在容器中获取类实例的快捷方式，通过本书第 11 章可以了解更多信息。

示例 4-23 不通过自定义 Blade 指令在 `context` 上执行相应条件

```
@if (app('context')->isPublic())
    &copy; Copyright MyApp LLC
@else
    &copy; Copyright {{ app('context')->client->name }}
@endif
```

如果希望将 `@if (app('context')->isPublic())` 简化为 `@ifPublic`，该怎么做？实现方式可参照示例 4-24。

示例 4-24 通过自定义 Blade 指令在 `context` 上执行相应条件

```
// 绑定
Blade::directive('ifPublic', function () {
    return "<?php if (app('context')->isPublic()): ?>";
});

// 使用
@ifPublic
    &copy; Copyright MyApp LLC
```

```
@else
    &copy; Copyright {{ app('context')->client->name }}
@endif
```

由于这里解析了一个简单的 `if` 语句，仍然需要依赖于自带的 `@else` 和 `@endif` 条件。但是如果需要，也可以通过创建一个自定义的 `@elseifClient` 指令或单独的 `@ifClient` 指令，或者希望创建的其他任何自定义指令来实现。

测试

测试视图最常见的方法是应用程序测试，这意味着实际上调用了显示视图的路由，并确保视图具有一定内容（见示例 4-25）。还可以点击按钮或提交表单，以确保被重定向到某个页面，或者看到一个特定的错误（本书第 12 章将详细讲解测试）。

示例4-25 测试视图显示某些内容

```
// EventsTest.php
public function test_list_page_shows_all_events()
{
    $event1 = factory(Event::class)->create();
    $event2 = factory(Event::class)->create();

    $this->visit('events')
        ->see($event1->title)
        ->see($event2->title);
}
```

还可以测试某个已经传递了一组特定的数据的视图，如果它完成了测试目标，那么这种测试方法比在页面上检查某些文本更加容易。示例 4-26 演示了这种测试方法。

示例4-26 测试传递了某些内容的视图

```
// EventsTest.php
public function test_list_page_shows_all_events()
{
    $event1 = factory(Event::class)->create();
    $event2 = factory(Event::class)->create();

    $this->visit('events');

    $this->assertViewHas('events', Event::all());
    $this->assertViewHasAll([
        'events' => Event::all(),
        'title' => 'Events Page'
    ]);
}
```

```
$this->assertViewMissing('dogs');  
}
```

5.3 在 5.3 中，可以将闭包传递给 `$assertViewHas()`，这意味着可以自定义如何检查更复杂的数据结构。使用方式，如示例 4-27 所示。

示例4-27 传递一个闭包到assertViewHas()

```
// EventsTest.php  
public function test_list_page_shows_all_events()  
{  
    $event1 = factory(Event::class)->create();  
  
    $this->visit('events/' . $event1->id);  
  
    $this->assertViewHas('event', function ($event) use ($event1) {  
        return $event->id === $event1->id;  
    });  
}
```

本章小结

Blade 是 Laravel 的模板引擎。其重点是具有强大的继承和可扩展、清晰、简洁而富有表现力的语法。它的“安全输出 (safe echo)”括号（即边界符）是 `{{` 和 `}}`，无保护的 echo 括号是 `{!!` 和 `!!}}`，而且它还有一系列名为指令的自定义标签，全部以 `@`（如 `@if` 和 `@unless`）开始。

用户可以定义父模板，并使用 `@yield` 及 `@section/@show` 为内容留下“holes（即一个区域）”，然后可以让子视图通过 `@extends('parent.view.name')` 去扩展它，并使用 `@section/@endsection` 定义其 section。还可以使用 `@parent` 引用块的父项内容。

视图 composer 可以轻松地进行定义操作，每次加载特定的视图或子视图时，都会有一些可用的信息。服务注入也允许视图本身直接从应用程序容器中请求数据。

前端组件

Laravel 的主要作用是 PHP 框架，但它也有一系列组件专注于生成前端代码。其中一些，如分页和消息包等，面向前端的 PHP 助手。Laravel 还提供了一个基于 Gulp 的构建系统，名为 Elixir，还有一些关于非 PHP（non-PHP）asset 的约定。

由于 Elixir 是非 PHP 前端组件的核心，所以我们从 Elixir 开始介绍。

Elixir

Elixir（避免与函数式编程语言混淆）是一种构建工具，它可以在 Gulp 的基础上提供简单的用户界面和一系列约定（<http://gulpjs.com/>）。Elixir 的核心功能是通过更简捷的 API、一系列命名及应用程序结构约定来简化最常见的 Gulp 任务。

Gulp 的快速介绍

Gulp 是一个 JavaScript 工具，用于编译静态资源并协调构建过程的其他步骤。

Gulp 类似于 Grunt、Rake 或 make，它允许定义一个操作或一系列操作（在 Gulp 中称之为“任务”），并且每次构建应用程序时会执行这些操作。这些操作通常包括运行像 Sass 或 LESS 这样的 CSS 预处理器、复制文件、连接及压缩 JavaScript 等。

由于是在 Gulp 上构建的，因此 Elixir 沿用流的思路。大多数任务始于把一些文件加载到流缓冲区中，然后任务将应用转换为内容预处理，将其缩小，然后把内容保存到新文件中。

关键是，Elixir 只是 Gulp 工具箱中的一个工具，在操作中甚至不会产生一个 Elixir 文件，

并且可以在 *gulpfile.js* 中定义 Elixir 任务。它们看起来与普通的 Gulp 任务有很大的不同，必须做更多的工作才能使其运行。

我们来看一个常见的例子：运行 Sass 来预处理 CSS 样式。在正常的 Gulp 环境中，代码如下示例 5-1 所示。

示例5-1 在Gulp中编译Sass文件

```
var gulp = require('gulp'),
    sass = require('gulp-ruby-sass'),
    autoprefixer = require('gulp-autoprefixer'),
    rename = require('gulp-rename'),
    notify = require('gulp-notify'),
    livereload = require('gulp-livereload'),
    lr = require('tiny-lr'),
    server = lr();

gulp.task('sass', function() {
  return gulp.src('resources/assets/sass/app.scss')
    .pipe(sass({
      style: 'compressed',
      sourcemap: true
    }))
    .pipe(autoprefixer('last 2 version', 'ie 9', 'ios 6'))
    .pipe(gulp.dest('public/css'))
    .pipe(rename({suffix: '.min'}))
    .pipe(livereload(server))
    .pipe(notify({
      title: "Karani",
      message: "Styles task complete."
    }));
});
```

现在，这段代码具有可读性，并且条理清晰。但是发生了很多事情，需要把做的每一个网站都拉进去。一个网站中进行处理的话，则需要大量的工作，这会让人感到困惑而啰唆。而且，笔者还见过更糟糕的情况。

可以试试在 Elixir 中执行的同样的任务（见示例 5-2）。

示例5-2 在Elixir中编译Sass文件

```
var elixir = require('laravel-elixir'); elixir(function (mix) {
  mix.sass('app.scss');
});
```

就是这么简单。这涵盖了所有的基础知识，包括预处理、通知、文件夹结构和

autoprefixing 等。



在 Elixir 6 中的 ES6

5.3 Elixir 6 与 Laravel 5.3 一起出现，改变了很多使用 ES6（ES6 是最新版本的 JavaScript）的语法。在 Elixir 6 中使用 ES6 时，可以参照示例 5-2 所示的代码。

```
const elixir = require('laravel-elixir');

elixir(mix => {
  mix.sass('app.scss')
});
```

别担心，二者其实是一样的。

Elixir 文件夹结构

Elixir 的简单性，来自于许多设定好的目录结构。在新的应用程序中如何确定新的资源和编译资源的位置？只要坚持 Elixir 的惯例就可以了。

每一个新的 Laravel 应用都有一个资源文件夹和一个资源子文件夹，这是 Elixir 存放前端资源的位置：将 Sass 存放在 *resources/assets/sass* 中；将 LESS 存放在 *resources/assets/less* 中；将 JavaScript 存放在 *resources/assets/js* 中。这样，这些资源将会导出到 *public/css* 和 *public/js*。

如果想更改目录结构，也可以通过更改 `elixir.config` 对象上的相应属性（`assetsPath` 及 `public Path`）来更改源和公共路径。

运行 Elixir

由于 Elixir 在 Gulp 上运行，所以需要在使用之前设置以下工具。

1. 首先，需要安装 Node.js。可以访问 Node 官方网站，从而了解如何运行。
2. 接下来，需要在计算机上全局安装 Gulp。只需在计算机上终端的任何位置运行 `npm install --global gulp-cli`，就能实现。

一旦安装了 Node 和 Gulp，就不必再次运行这些命令了。现在，已经准备好安装此项目的依赖项了。

3. 打开终端中的项目根目录，并运行 `npm install` 来安装所需的软件包（Laravel 提供了一个 Elixir-ready *package.json* 文件用于指导 NPM 的使用方法）。

现在已经安装好 Elixir 了。执行 `gulp` 便可以运行一次 Gulp/Elixir，可以使用 `gulp`

`watch` 来监听相关的文件更改和运行的响应，也可以使用 `gulp scripts` 或 `gulp styles` 来运行脚本或样式任务。

Elixir 提供了什么

我们已经知道 Elixir 可以使用 Sass 或 LESS 来预处理 CSS。它可以连接文件、压缩文件、重命名并复制文件，并且可以复制整个目录或单个文件等。

Elixir 还可以处理 ES6/ES2015 的 JavaScript，并在代码上运行 Webpack、Rollup 和 Autoprefixer 等。不仅如此，Elixir 在每个脚本或样式上都涵盖了大多数 JavaScript 和 CSS 的现代编码标准，开箱即用。

Elixir 也可以运行测试。PHPUnit 的方法和 PHPSpec 的方法，都会监听测试文件的更改，并在每次进行更改时重新运行测试套件。

Elixir 文档涵盖了所有相关操作，接下来介绍一些具体的示例。

--production 标志

在默认情况下，Elixir 不会压缩生成的所有文件。但是，如果在“生产”模式下运行构建脚本，希望启用所有分区，只需添加 `--production` 标志。

```
$ gulp --production
```

传递多个文件

大多数 Elixir 方法通常只接收单个文件（例如，`mix.sass('app.scss')`），当然这些方法也可以接收一个文件数组，如示例 5-3 所示。

示例5-3 用Elixir编译多个文件

```
const elixir = require('laravel-elixir'); elixir(mix => {  
  
    mix.sass([  
        'app.scss',  
        'public.scss'  
    ]);  
});
```

source map

在默认情况下，Elixir 会为文件生成 source map，可以将它们视为在每个生成的文件旁边的映射文件，类似于 `.{file-name}.map`。

如果不熟悉 source map，那么可以使用任何预处理器以使浏览器的 Web 检查器展示正在

检查的编译源代码会生成哪些文件。

没有 source map，如果使用浏览器的开发工具检查特定的 CSS 和 Java Script 操作时，会看到编译的代码十分混乱；有 source map，浏览器可以精确查明源文件的每一行，无论是 Sass、Java Script 或其他类型的文件，都会在检查时生成一定的规则。

如果不想用 source map，也可以随时更改 elixir 块配置，如示例 5-4 所示。

示例5-4 禁用Elixir中的source map

```
const elixir = require('laravel-elixir');

elixir.config.sourcemaps = false;

elixir(mix => {
  mix.sass('app.scss');
});
```

预处理 CSS

如果不想通过预处理器处理，那么可以使用一个命令进行，它会获取所有的 CSS 文件，然后连接它们，并将它们输出到 *public/css* 目录，就好像是通过一个预处理器进行的。如果没有指定输出文件名，那么最终将在 *all.css* 中输出。

示例5-5 通过Elixir联结样式表

```
const elixir = require('laravel-elixir');

elixir(mix => {
  // 联结 resources/assets/css 和子文件夹中的所有文件
  mix.styles();

  // 在 resources/assets/css 中联结文件
  mix.styles([
    'normalize.css',
    'app.css'
  ]);

  // 在其他目录中联结所有样式文件
  mix.stylesIn('resources/some/other/css/directory');

  // 在 resources/assets/css 中联结给定的样式文件
  // 输出到自定义目录
  mix.styles([
    'normalize.css',
    'app.css'
  ], 'public/other/css/output.css');
```

```
// 从自定义目录中联结给定的样式文件
// 输出到自定义目录
mix.styles([
  'normalize.css',
  'app.css'
], 'public/other/css/output.css', 'resources/some/other/css/directory');
});
```

连接 JavaScript

普通 JavaScript 文件的设置与普通 CSS 文件的设置类似，如示例 5-6 所示。与 `styles()` 一样，未提供输出文件名的任何命令都将输出到 *public/js/all.js*。

示例5-6 通过Elixir联结JavaScript文件

```
const elixir = require('laravel-elixir');

elixir(mix => {
  // 从 resources/assets/js 中联结文件
  mix.scripts([
    'jquery.js',
    'app.js'
  ]);

  // 从其他文件夹中联结所有 scripts 文件
  mix.scriptsIn('resources/some/other/js/directory');

  // 从 resources/assets/js 中联结给定的 scripts 文件
  // 输出到自定义目录
  mix.scripts([
    'jquery.js',
    'app.js'
  ], 'public/other/js/output.js');

  // 从自定义目录中联结给定的 scripts 文件
  // 输出到自定义目录
  mix.scripts([
    'jquery.js',
    'app.js'
  ], 'public/other/js/output.js', 'resources/some/other/js/directory');
});
```

处理 JavaScript

如果要处理 JavaScript，例如将 ES6 代码编译成纯 JavaScript 代码，可以通过 Elixir 的

Webpack 或 Rollup 轻松实现（见示例 5-7）。

示例 5-7 使用 Webpack 或 Rollup 处理 Elixir 中的 JavaScript 文件

```
elixir(function(mix) {  
    mix.webpack('app.js');  
  
    // 或者  
  
    mix.rollup('app.js');  
});
```

这些脚本将在 `resources/assets/js` 中查找所提供的文件名并输出到 `public/js/all.js` 中。

可以在项目根目录下创建一个 `webpack.config.js` 文件，以便使用 Webpack 更复杂的功能。



在 Elixir 5 中编译 JavaScript

5.2 在介绍 Laravel 5.3/Elixir 6 之前，需要通过 `mix.browserify('app.js')` 编译 JavaScript 脚本。

版本

在 Steve Souders 的 *Even Faster Web Sites*（O'Reilly 出版）这本书中，大部分建议都融入了日常的实际操作。例如将脚本移动到页脚、减少 HTTP 请求的数量等，但是我们常常不会意识到这些想法的来源。

Steve 有一个建议很少能真正得以实现，那就是要求为资源（脚本、样式和图像）设置一个很长的缓存生命周期。这样做意味着将会有更少的请求通过服务器来获取最新版本资源，同时也意味着用户获取的极有可能是缓存版本的资源，这将使这些资源变得过时，因此很快就会崩溃。

解决方案是版本控制。每次运行构建脚本时，为每个资源的文件名添加唯一的哈希值，然后该唯一文件将无限期缓存，或者至少缓存到下一次构建才释放。

这有什么问题吗？首先需要将唯一的哈希值生成并附加到文件名中，还需要更新构建的每个视图，从而引用新的文件名。

Elixir 会处理这些操作，所以这非常简单。这里有两个组件：Elixir 中的版本控制任务和 `elixir()` PHP 助手。首先，可以通过运行 `mix.version()` 对资源进行版本控制，如示例 5-8 所示。

示例5-8 mix.version

```
const elixir = require('laravel-elixir');

elixir(mix => {
  mix.version('public/css/all.css');
});
```

这将生成一个指定文件的版本，在 *public/build* 目录中添加一个唯一的哈希值，就像 *public/build/css/all-84fa1258.css* 一样。

接下来，就可以在视图中使用 PHP `elixir()` 助手来引用这个文件了，如示例 5-9 所示。

示例5-9 在视图中使用 `elixir()` 助手

```
<link rel="stylesheet" href="{{ elixir("css/all.css") }}">
```

// 会输出如下信息

```
<link rel="stylesheet" href="/build/css/all-84fa1258.css">
```

Elixir 版本控制幕后是如何工作的

Elixir 使用了 `gulp-rev`，它负责将哈希值添加到文件名中，并生成名为 *public/build/rev-manifest.json* 的文件。这将存储 `elixir()` 助手查找生成的文件所需的信息。关于 *rev-manifest.json* 的示例内容如下。

```
{
  "css/all.css": "css/all-7f592e49.css"
}
```

测试

每次更改测试文件时，都可以使用 Elixir 运行 PHPUnit 或 PHPSpec 进行测试。

这里有两种方案，`mix.phpUnit()` 或者 `mix.phpSpec()`，每种方案都会直接从 *vendor* 文件夹中运行相应的框架，所以不必进行其他操作来使它们工作。

但是，如果将其中一种方法添加到 Gulp 文件中，那么即便使用 `gulp watch`，也会发现它们只运行一次。那么，如何让它们知道测试文件夹的变化呢？

可以使用一个单独的 Gulp 命令：`gulp tdd`。这样无论是 `phpUnit()` 还是 `phpSpec()`，都可以获取 Gulp 文件中的测试命令、监听适当的文件夹，当任何文件发生更改时，都

会重新运行测试套件。

Elixir 扩展

Elixir 不仅为预先构建的任务提供了简单的语法，同时它也可以很容易地进行自定义。

假设想在特定的时候将文本保存到日志文件中，可以使用一个 shell 命令进行，这个 shell 命令可以是 `echo "message" >> file.log`。通常可以将它定义为 Gulp 任务，并且使用 `shell('echo "message" >> file.log')` 实现，如示例 5-10 所示。

示例5-10 在Elixir中使用Gulp任务

```
// 定义任务
gulp.task("log", function () {
    var message = "Something happened";
    gulp.src("").pipe(shell('echo "' + message + '" >> file.log'));
});

elixir(mix => {
    // 在 Elixir 中使用任务
    mix.task('log');

    // 绑定任务，每当某些文件被更改时运行
    mix.task('log', 'resources/somefiles/to/watch/**/*')
});
```

但是，如果想要进行更多控制，例如传递一些消息，这个时候特定的任务就会变得非常重要。这里可以创建一个 Elixir 扩展，如示例 5-11 所示。

示例5-11 创建Elixir扩展

```
// 在 gulpfile.js 或 gulpfile.js 中需要的外部文件
var gulp = require("gulp"),
    shell = require("gulp-shell"),
    elixir = require("laravel-elixir");
elixir.extend("log", function (message) {
    new Task('log', function() {
        return gulp.src('').pipe(shell('echo "' + message + '" >> file.log'));
    })
    .watch('./resources/some/files/**/*');
});
```

与任何组件一样，这里还没有介绍完有关 Elixir 的所有内容，但希望读者已经学会了目前介绍过的知识，这样基本暂时够用了。如果想要了解更多关于 Elixir 的知识，可以查看文档 (<https://laravel.com/docs/elixir>)。

分页

在 Web 应用程序中，分页非常常见，但是有时实现分页是非常复杂的。值得庆幸的是，Laravel 拥有内置的分页组件，在默认情况下还可以自动与 Eloquent 结果、路由器进行整合。

Eloquent 的简介

我们将在第 8 章中深入介绍 Eloquent、数据库访问和 Laravel 查询构建器相关的知识，这里介绍一些基础知识，给大家一些参考，这样有助于大家对这些知识进行基本的理解。

Eloquent 是 Laravel 的 ActiveRecord 数据库对象关系映射器 (ORM)，它可以使 Post 类（模型）与 posts 数据库表相关联，并通过 `Post::all()` 等获取调用所有记录。

查询构建器是可以进行诸如 `Post::where('active', true)->get()` 或者 `DB::table('users')->all()` 调用的工具。可以通过一个接一个的链接方法构建查询。

分页数据库结果

关于分页，最常需要使用的内容是显示数据库查询的结果，但是单个页面的结果太多。Eloquent 和查询构建器都从当前页面请求中读取页面查询参数，并可以对对应的结果集使用 `paginate()` 方法。这里可以通过设定 `paginate()` 的单个参数，来决定每页需要多少个结果，如示例 5-12 所示。

示例5-12 对查询构建器的响应进行分页

```
// PostsController
public function index()
{
    return view('posts.index', ['posts' => DB::table('posts')->paginate(20)]);
}
```

示例 5-12 定义了该路由每页返回 20 个结果，并且将根据 URL 的 `page` 查询参数（如果有的话）定义当前用户所在的“页面”结果。Eloquent 模型有相同的 `paginate()` 方法。

当在视图中显示结果时，集合有一个 `links()` 方法（在 Laravel 5.1 中是 `render()` 方法），它将输出分页控件，并默认分配给它们的类名（见示例 5-13）。

示例5-13 在模板中呈现分页链接

```
// posts/index.blade.php
<table>
@foreach ($posts as $post)
    <tr><td>{{ $post->title }}</td></tr>
@endforeach
</table>

{{ $posts->links() }}
```

// 在默认情况下，\$posts->links() 将会输出如下信息

```
<ul class="pagination">
    <li class="disabled"><span>&laquo;</span></li>
    <li class="active"><span>1</span></li>
    <li><a href="http://myapp.com/posts?page=2">2</a></li>
    <li><a href="http://myapp.com/posts?page=3">3</a></li>
    <li><a href="http://myapp.com/posts?page=2" rel="next">&raquo;</a></li>
</ul>
```

手动创建分页

如果不使用 Eloquent 或查询构建器，或者复杂查询（例如使用 `groupBy` 查询），则可能需要手动创建分页。幸运的是，用 `Illuminate\Pagination\Paginator` 或者 `Illuminate\Pagination\LengthAwarePaginator` 类也可以做到这一点。

两个类之间的差异在于，`Paginator` 仅提供上一个和下一个按钮，但不提供每个页面的链接，`LengthAwarePaginator` 需要知道完整结果有多少，以便为每个页面生成链接。在大型结果集上使用 `Paginator` 会比较好，因为这样分页器就不必知道所有结果集，也就不需要获得大量结果，故而可以降低运行成本。

`Paginator` 和 `LengthAwarePaginator` 都要求手动提取要传递给视图的内容子集，如示例 5-14 所示。

示例5-14 在Laravel 5.2和Laravel 5.3中手动创建分页

```
use Illuminate\Http\Request;
use Illuminate\Pagination\Paginator;

Route::get('people', function (Request $request) {
    $people = [...]; // 大型人物清单

    $perPage = 15;
```

```

$offsetPages = $request->input('page', 1) - 1;

// 分页不会切换序列
$people = array_slice(
    $people,
    $offsetPages * $perPage,
    $perPage
);

return new Paginator(
    $people,
    $perPage
);
});

```

Paginator 语法在 Laravel 的最后几个版本中有所改变，所以如果使用 Larvel 5.1，请查看文档以找到正确的语法。

消息包

Web 应用程序中的另一个常见但令人头疼的功能是在应用程序的各个组件之间传递消息，并最终与用户共享该消息。例如控制器可能需要发送验证消息“email 字段必须是有效的电子邮件地址”，但是该特定消息不仅需要发送到视图层，实际上还需要在重定向中生存，最终出现在不同页面的视图层中。如何构建这种消息传递逻辑呢？

`Illuminate\Support\MessageBag` 是一个负责存储、分类和返回最终用户消息的类。它通过键（key）分组所有消息，其中键类似于 `errors`、`messages` 等，同时该类会为获取的所有存储消息（或者仅对特定键提供的消息）提供便利方法，并以各种格式输出这些消息。

这里手动启动一个新的 `MessageBag`，如示例 5-15 所示。

示例 5-15 手动创建和使用消息包（`MessageBag`）

```

$messages = [
    'errors' => [
        'Something went wrong with edit 1!'
    ],
    'messages' => [
        'Edit 2 was successful.'
    ]
];
$messagebag = new \Illuminate\Support\MessageBag($messages);

```



```
// 检查错误；如果有的话，进行 decorate 和 echo
if ($messagebag->has('errors')) {
    echo '<ul id="errors">';
    foreach ($messagebag->get('errors', '<li><b>:message</b></li>') as $error) {
        echo $error;
    }
    echo '</ul>';
}
```

消息包也与 Laravel 的验证器密切相关（参见第 99 页的“验证”，了解更多信息）：在验证程序返回错误时，实际上返回一个 `MessageBag` 实例，然后可以将其传递给视图或添加到重定向中。这里可以使用 `redirect('route')->withErrors($messagebag)` 实现。

Laravel 将 `MessageBag` 的一个空实例传递给每个视图，分配给变量 `$errors`，如果在重定向上使用 `withErrors()` 闪存了一个消息包，则会将其分配给变量 `$errors`。这意味着每个视图总是可以假设有一个名为 `$errors` 的 `MessageBag`，可以在执行验证的任何位置检查它。开发人员可以在每个页面上都使用这些代码片段，这些代码片段可以看作是通用代码片段，如示例 5-16 所示。

示例5-16 错误包 (Error bag) 代码片段

```
// partials/errors.blade.php
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```



缺少 \$errors 变量

对于任何不在 Web 中间件组下的路由，它们将不会有会话中间件，这意味着它们的 `$errors` 变量不可用。

错误包命名

有时，我们需要区分消息包，而且不仅仅是通过键（`notices` 或者 `errors`）进行区分，同时还需要通过组件来区分消息包。比如在同一页上有登录表单和注册表单，那么这个

时候如何区分他们呢？

当使用 `withErrors()` 发送错误以及重定向时，其实第二个参数是包的名称：`redirect('dashboard')->withErrors($validator,'login')`。然后，在仪表板上，可以使用 `$errors->login` 来调用之前看到的所有方法：`any()`、`count()` 等。

字符串助手、多元化和本地化

作为开发人员，倾向于将文本块视为大占位符，等待客户将真实内容放入其中。在这些占位区块内，很少会涉及任何业务逻辑。

在以下几种情况下，如果使用 Laravel 为字符串操作提供的工具，将非常方便。

字符串助手和多元化

Laravel 有一系列操作字符串的助手函数。它们可以作为 `Str` 类的方法（例如 `Str::plural()`），此外其中大多数也会有一个全局助手函数（例如 `str_plural()`）。

Laravel 文档（<https://laravel.com/docs/5.3/helpers>）详细介绍了所有的字符串助手函数，这里介绍一些最常用的助手。

e

`html_entities` 的快捷方式。

`starts_with`、`ends_with`、`str_contains`

检查一个字符串（第一个参数），看看它是否以另一个字符串（第二参数）为开始或结束。

`str_is`

检查字符串（第二个参数）是否与特定模式（第一个参数）匹配。例如，`foo*` 将匹配 `foobar` 和 `foobaz`。

`str_slug`

将字符串转化为 URL 类型的 slug。

`str_plural (word, num)`、`str_singular`

将一个复数化或单数化，仅限于英文（例如，`str_plural('dog')` 会返回 `dogs`）。

本地化

通过本地化，可以定义多种语言，并将任何字符串标记为转换目标。这样便可以设置一个备用语言，甚至可以处理多种语言。

在 Laravel 中，需要在页面加载的某个点设置应用程序语言环境，以便本地化助手知道要从哪个页面开始翻译。可以使用 `App::setLocale($localeName)` 来执行此操作，将其放在服务提供者中；还可以将其放在 `AppServiceProvider` 的 `boot()` 方法中。如果最终想得到的不仅仅是与这个本地语言相关的绑定，则可能需要创建一个 `LocaleServiceProvider`。

为每个请求设置本地化

首先，弄清楚 Laravel 是如何“知道”用户的语言环境并且提供翻译的。其中大部分的工作都是开发人员完成的。接下来设想以下场景。

可能会有一些功能允许用户选择一个本地化设置，或者尝试自动检测。无论哪种方式，应用程序最终将确定本地化设置，然后将其存储在 URL 参数或会话 cookie 中。那么服务提供者就像 `LocaleServiceProvider` 一样，也许会获取这个键并将其设置为 Laravel 引导程序的一部分。

也许有的用户在访问 `http://myapp.com/es/contacts`。此时 `LocaleServiceProvider` 将获取该 `es` 字符串，然后运行 `App::setLocale('es')`。这样，在每次要求翻译字符串时，Laravel 会在需要翻译的位置寻找西班牙语（es）版本的字符。

可以在 `config/app.php` 中定义备用语言环境。这里可以找到一个名为 `fall back_locale` 的键。它可以为应用程序定义一个默认语言，如果 Laravel 无法为请求的语言环境找到翻译，则使用默认语言。

基本本地化

那么如何调用一个翻译的字符串呢？可以使用助手函数 `trans($key)` 实现，它将为传递的键获取当前语言环境的字符串，如果不存在，则从默认语言环境中获取。示例 5-17 演示了基本翻译的运作过程，并且以详细页面顶部的“返回信息中心”链接作为示例。

示例5-17 trans()的基本使用方法

```
// 普通 PHP
<?php echo trans('navigation.back'); ?>
```

```
// Blade
{{ trans('navigation.back') }}
```

```
// Blade 指令
@lang('navigation.back')
```

假设我们现在正在使用 es 语言环境。Laravel 将在 *resources/lang/es/navigation.php* 中查找一个文件，并返回一个数组。它将在该数组中寻找一个名为 **back** 的键，如果存在，返回其值，如示例 5-18 所示。

示例5-18 使用翻译

```
// resources/lang/es/navigation.php
return [
    'back' => 'Volver al panel'
];

// routes/web.php
Route::get('/es/contacts/show/:id', function () {
    // 手动设置，而不是放在服务提供者中
    App::setLocale('es');
    return view('contacts.show');
});

// resources/views/contacts/show.blade.php
<a href="/contacts">{{ trans('navigation.back') }}</a>
```

本地化参数

前面的例子比较简单。接下来研究一些比较复杂的内容。如果定义要返回的仪表板，该怎么办呢？可以看一下示例 5-19。

示例5-19 翻译参数

```
// resources/lang/en/navigation.php
return [
    'back' => 'Back to :section dashbaord'
];

// resources/views/contacts/show.blade.php
{{ trans('navigation.back', ['section' => 'contacts']) }}
```

可以看到，用冒号加上一个单词 (`:section`) 表示将其标记为可以替换的占位符。`trans()` 的第二个可选参数表示用于替换占位符的值的数组。

多元本地化

我们已经介绍了多元化，所以现在需要知道如何自定义多元化规则。有两种方法可以做到，下面以一个简单的示例开始介绍，如示例 5-20 所示。

示例5-20 定义一个简单的翻译与多元选择

```
// resources/lang/en/messages.php
return [
    'task-deletion' => 'You have deleted a task|You have successfully deleted
tasks'
];

// resources/views/dashboard.blade.php
@if ($numTasksDeleted > 0)
    {{ trans_choice('messages.task-deletion', $numTasksDeleted) }}
@endif
```

这里有一个 `trans_choice()` 方法，它将受影响的项目数作为第二个参数，并从中确定使用哪个字符串。

还可以使用与 Symfony 更复杂的 `translation` 组件兼容的任何翻译定义，相关的示例程序，可以参见示例 5-21。

示例5-21 使用Symfony的翻译组件

```
// resources/lang/es/messages.php
return [
    'task-deletion' => "{0} You didn't manage to delete any tasks." .
        "[1,4] You deleted a few tasks." .
        "[5,Inf] You deleted a whole ton of tasks."
];
```

测试

在本章中，主要关注了 Laravel 的前端组件。这些前端组件一般不是单元测试的对象，但它们有时可以进行集成测试。

用 Elixir 进行测试

尽管我们不会在 Elixir 任务上编写任何测试，但是 Elixir 提供了一些帮助进行测试的功能，

所以这里简单介绍一下。

如果将 `mix.phpunit()` 或 `mix.phpspec()` 添加到 `gulpfile.js` 中，那么每次运行 `gulp`，它会将运行异常测试和内联，作为构建脚本的一部分。

每次运行 `gulp watch` 时，Elixir 将会随时对测试文件或任何其他核心文件（如 `routes/web.php`）进行一些更改，并在每次对这些文件进行更改时重新运行 PHPUnit 或 PHPSpec。

测试消息包和错误包

测试传递消息包和错误包主要有两种方式。其一，可以在应用程序测试中执行一个行为，设置最终将显示在某处的消息，然后重定向到该页面，并声明显示相应的消息。

其二，对于错误（这是最常见的一个例子），可以使用 `$this->assertSessionHasErrors($bindings = [])` 断言会话有错误。具体的实现方法可以通过示例 5-22 进行了解。

示例5-22 断言会话有错误

```
public function test_missing_email_field_errors()
{
    $this->post('person/create', ['name' => 'Japheth']);
    $this->assertSessionHasErrors(['email']);
}
```

翻译和本地化

测试本地化，最简单的方法是应用程序测试。设置适当的条件（无论是通过 URL 还是会话），在页面使用 `visit()` 方法，并断言得到相应内容。

本章小结

作为一个全栈框架，Laravel 为前端和后端提供了工具和组件。

Elixir 是围绕常见的 Gulp 构建任务的一个封装，它让使用前沿的构建步骤变得简单。Elixir 可以轻松添加 CSS 预处理器、JavaScript 透视、连接和压缩，并且可以实现更多的功能。

Laravel 还提供针对前端的其他内部工具，包括分页、消息包和错误包及本地化等。

收集和处理用户数据

使用 Laravel 这样的框架搭建的网站通常不仅处理静态内容，还会处理复杂和混合数据源，其中最常见（也是最复杂）的数据源是用户多种形式的输入：URL 路径、查询参数、POST 数据和文件上传等。

Laravel 提供了一组工具，其用于收集、验证、归一化和过滤用户提供的数据。

注入请求对象

在 Laravel 中访问用户数据的最常见工具是注入 `Illuminate\Http\Request` 对象的实例。它可以轻松访问用户能够在站点中输入的所有形式的数据：POST、posted JSON、GET（查询参数）和 URL 片段等。



访问请求数据的其他方式

还可以使用 `request()` 全局助手和 `Request facade` 实现，这两种方式都有相同的方法。这些方式都使用了完整的 `Illuminate Request` 对象，但目前我们只讨论涉及用户数据的方法。

如果想要注入一个 `Request` 对象，则需要了解如何获取 `$request` 对象，可以通过如下代码调用所有相关的方法。

```
Route::post('form', function (Illuminate\Http\Request $request) {  
    // $request->etc()  
});
```

\$request->all()

就像其名字一样，`$request->all()` 会提供一个包含用户所有输入的数组。比方说，出于某种原因，我们决定使用查询参数的 `POST` 表单（例如，发送 `POST` 到 `http://myapp.com/post?utm=12345`）。可以通过示例 6-1 了解从 `$request->all()` 获得了什么。

示例6-1 `$request->all()`

```
<!-- GET route form view at /get-route -->
<form method="post" action="/post-route?utm=12345">
    {{ csrf_field() }}
    <input type="text" name="firstName">
    <input type="submit">
</form>

Route::post('/post-route', function (Request $request) {
    var_dump($request->all());
});

// 输出
/**
 * [
 *     '_token' => 'CSRF token here',
 *     'firstName' => 'value',
 *     'utm' => 12345
 * ]
 */
```

\$request->except() 和 \$request->only()

`$request->except()` 与 `$request->all()` 具有相同的输出，但 `$request->except()` 可以设置一个或多个要排除的字段，例如 `_token`。可以将其传递为字符串或字符串数组。

示例 6-2 展示了如果使用 `$request->except()`，与示例 6-1 中的相同表单的应用将会变成什么形式。

示例6-2 `$request->except()`

```
Route::post('/post-route', function (Request $request) {
    var_dump($request->except('_token'));
});

// 输出
/**
 * [
```



```
*      'firstName' => 'value',
*      'utm' => 12345
* ]
*/
```

`$request->only()` 与 `$request->except()` 的含义相反，如示例 6-3 所示。

示例6-3 `$request->only()`

```
Route::post('/post-route', function (Request $request) {
    var_dump($request->only(['firstName', 'utm']));
});
```

```
// 输出
/**
 * [
 *      'firstName' => 'value',
 *      'utm' => 12345
 * ]
 */
```

`$request->has()` 与 `$request->exists()`

使用 `$request->has()` 可以检测特定的用户输入是否可用。使用示例 6-3 中的 `utm` 字符串请求参数进行分析，如示例 6-4 所示。

示例6-4 `$request->has()`

```
// 通过 /post-route 传递路由
if ($request->has('utm')) {
    // 进行分析
}
```

`$request->exists()` 和 `$request->has()` 的区别在于它们处理空值的方式不同：如果该键存在且为空，`has()` 会返回 `FALSE`；如果键存在，即使它为空，`exists()` 也会返回 `TRUE`。

`$request->input()`

`$request->all()`、`$request->except()` 和 `$request->only()` 都会对用户的完整输入数组进行操作，但是 `$request->input()` 只允许获取单个字段的值，如示例 6-5 所示。请注意，第二个参数是默认值，因此如果用户没有传入值，则可以有一个合理的（和不中断的）回退。

示例6-5 \$request->input()

```
Route::post('/post-route', function (Request $request) {  
    $userName = $request->input('name', '(anonymous)');  
});
```

数组输入

Laravel 还为访问数组类型的输入数据提供了便利助手函数。使用 “.” 符号表示下层数组结构的标识，如示例 6-6 所示。

示例6-6 通过点标记法在用户数据中访问数组值

```
<!-- GET route form view at /get-route -->  
<form method="post" action="/post-route">  
    {{ csrf_field() }}  
    <input type="text" name="employees[0][firstName]">  
    <input type="text" name="employees[0][lastName]">  
    <input type="text" name="employees[1][firstName]">  
    <input type="text" name="employees[1][lastName]">  
    <input type="submit">  
</form>  
  
// 通过 /post-route 传递路由  
Route::post('/post-route', function (Request $request) {  
    $employeeZeroFirstName = $request->input('employees.0.firstName');  
    $allLastNames = $request->input('employees.*.lastName');  
    $employeeOne = $request->input('employees.1');  
});  
  
// 如果表单填写为 "Jim" "Smith" "Bob" "Jones"  
// $employeeZeroFirstName = 'Jim';  
// $allLastNames = ['Smith', 'Jones'];  
// $employeeOne = ['firstName' => 'Bob', 'lastName' => 'Jones']
```

JSON 输入 (\$request->json())

到目前为止，已经介绍了查询字符串（GET）和表单提交（POST）的输入。但是，随着 JavaScript 单页应用程序（SPA）的出现，另一种形式的用户输入变得越来越普遍：JSON 请求。它本质上只是一个 POST 请求，但是其主体设置为 JSON 形式的数据，而不是传统的 POST 形式。

接下来了解将 JSON 提交给 Laravel 路由会如何，以及如何使用 \$request->input() 来提取数据（见示例 6-7）。

示例6-7 使用\$request->input()从JSON获取数据

POST /post-route HTTP/1.1

Content-Type: application/json

```
{
  "firstName": "Joe",
  "lastName": "Schmoe",
  "spouse": {
    "firstName": "Jill",
    "lastName": "Schmoe"
  }
}
// post-route
Route::post('post-route', function (Request $request) {
    $firstName = $request->input('firstName');
    $spouseFirstname = $request->input('spouse.firstName');
});
```

`$request->input()` 可以足够灵活地从 GET、POST 或 JSON 中提取用户数据，为什么 Laravel 还提供 `$request->json()`？可能有两个原因。首先，更加明确地对项目的程序员指出期望数据来自哪里；其次，如果 POST 没有正确的 `application/json` 头，那么 `$request->input()` 不会将其替换为 JSON，但 `$request->json()` 会。

Facade 命名空间、request() 全局助手函数与 \$request 注入

任何时候在命名空间的类中（例如控制器）使用 facade，必须将完整的 facade 路径添加到文件顶部的导入部分（例如 `use Illuminate\Support\Facades\Request` 等）。

正因如此，一些 facade 有一个配套的全局助手函数。如果这些助手函数在运行时不带参数，那么它们将与 facade 具有相同的语法（例如 `request()->has()` 与 `Request::has()` 相同）。当传递一个参数时，它们还有一个默认的规则（例如 `request('firstName')` 是 `request()->input('firstName')` 的快捷方式）。

对于 Request，我们已经介绍了一个注册 Request 对象的实例，其实也可以使用 Request facade 或 `request()` 全局助手函数实现。如果想要了解更多相关知识，可以查看第 10 章。

路由数据

当提到“用户数据”时，你可能不会第一时间想到 URL，但是在本章中，关于 URL 的介绍与关于其他用户数据的介绍一样多。

可以通过以下三种方式从 URL 中获取数据：Request facade、路由参数和 Request 对象。我们将在第 10 章中更详细地介绍 Request 对象的相关知识。

通过 Request 实现

注入的 Request 对象（包括 Request facade 和 request() 助手函数）有几种可用的方法来表示当前页面的 URL 状态，我们主要了解如何获取与 URL 片段相关的信息。

对于 URL 片段，简单来说，在域名之后的每组字符都称为片段。所以，*http://www.myapp.com/users/15/* 有两个片段：*users* 和 *15*。

我们有两种可用的方法：`$request->segments()` 会返回所有片段的数组，`$request->segment($segmentId)` 会获取单个片段的值。请注意，这里会在一个基于 1 的索引上返回片段，因此在前面的示例中，`$request->segment(1)` 将返回 *users*。

Request 对象、Request facade 和 request() 助手函数提供了不少其他更多的方法帮助我们 从 URL 中获取数据。要了解更多的知识，可以参考第 10 章内容。

通过路由参数实现

获取关于 URL 的数据的另一个主要方式是路由参数，这些参数被注入控制器方法或者正在服务于当前路由的闭包，如示例 6-8 所示。

示例6-8 从路由参数获取URL详细信息

```
// routes/web.php
Route::get('users/{id}', function ($id) {
    // 如果用户访问 myapp.com/users/15/, $id 等同于 15
});
```

如果想了解有关路由和路由绑定的更多信息，可以参考第 3 章内容。

上传的文件

我们已经讨论了与用户的文本输入进行交互的不同方法，但是还有一些关于文件上传的问题需要考虑。Request facade 使用的 `Request::file()` 方法可以访问任何上传的文

件，该方法将文件的输入名称作为参数，并返回 `Symfony\Component\HttpFoundation\File\UploadedFile` 的实例。

我们来看一个相关例子，如示例 6-9 所示。

示例6-9 上传文件的表单

```
<form method="post" enctype="multipart/form-data">
    {{ csrf_field() }}
    <input type="text" name="name">
    <input type="file" name="profile_picture">
    <input type="submit">
</form>
```

现在，我们来看看如何运行 `$request->all()`，如示例 6-10 所示。注意 `$request->input('profile_picture')` 会返回 `null`，而使用 `$request->file('profile_picture')` 才能正常返回。

示例6-10 提交示例6-9中的表单后的输出

```
Route::post('form', function (Request $request) {
    var_dump($request->all());
});

// 输出
// [
//     "_token" => "token here"
//     "name" => "asdf"
//     "profile_picture" => UploadedFile {}
// ]

Route::post('form', function (Request $request) {
    if ($request->hasFile('profile_picture')) {
        var_dump($request->file('profile_picture'));
    }
});

// 输出
// UploadedFile (details)
```

验证文件上传

如示例 6-10 所示, 可以通过 `$request->hasFile()` 来查看用户是否上传了文件; 还可以通过在文件自身上使用 `isValid()` 来检查文件上传是否成功, 示例如下。

```
if ($request->file('profile_picture')->isValid()) {  
    //  
}
```

因为 `isValid()` 在文件自身上被调用, 如果用户没有上传文件, 它将会出错。所以, 要检查文件是否上传成功及用户是否上传了文件, 需要首先检查文件是否存在, 示例如下。

```
if (  
    $request->hasFile('profile_picture') &&  
    $request->file('profile_picture')->isValid()  
) {  
    //  
}
```

Symfony 的 `UploadedFile` 类扩展自 PHP 的原生 `SplFileInfo`, 通过这些方法可以轻松实现检查和操作文件。以下列表并不详尽, 但它可以作为参考。

- `guessExtension()`
- `getMimeType()`
- `store($path, $storageDisk = default disk)`
- `storeAs($path, $newName, $storageDisk = default disk)`
- `storePublicly($path, $storageDisk = default disk)`
- `storePubliclyAs($path, $newName, $storageDisk = default disk)`
- `move($directory, $newName = null)`
- `getClientOriginalName()`
- `getClientOriginalExtension()`
- `getClientMimeType()`
- `guessClientExtension()`
- `getClientSize()`
- `getError()`
- `isValid()`

5.3 可以看到，大多数方法与获取关于上传文件的信息有关，但有一个方法可能使用得更多：`store()`（Laravel 5.3 中的新功能），它将使用 `request` 上传的文件存储在服务器上的指定目录中。它的第一个参数表示目标目录，是可选的；第二个参数表示存储文件的存储磁盘（如 `s3.local` 等）。

可以在示例 6-11 中看到一个通用的工作流。

示例6-11 通用文件上传工作流

```
if ($request->hasFile('profile_picture')) {  
    $path = $request->profile_picture->store('profiles', 's3');  
    auth()->user()->profile_picture = $path;  
    auth()->user()->save();  
}
```

如果需要指定文件名，可以使用 `storeAs()` 替代 `store()`。第一个参数仍然表示路径，第二个参数表示文件名，第三个参数（可选的）表示要使用的存储磁盘。



文件上传的正确编码格式

如果在尝试从 `request` 请求中获取文件的内容时，结果为 `null`，那么可能是忘记在表单上设置编码类型了。需要确保在表单上添加了 `enctype="multipart/form-data"` 属性。

```
<form method="post" enctype="multipart/form-data">
```

验证

Laravel 还有很多方法可以验证传入的数据。对于表单请求，这里主要有两种实现方式：手动验证或者在控制器中使用 `validate()` 方法。从简单的开始，首先介绍常见的方法 `validate()`。

在控制器中使用 `ValidatesRequests` 的 `validate()` 方法

开箱即用，是 `ValidatesRequests` 的一大特点，它提供了一个便捷的 `validate()` 方法。我们可以看看示例 6-12 中的内容。

示例6-12 控制器验证的基本用法

```
// routes/web.php  
Route::get('recipes/create', 'RecipesController@create');  
Route::post('recipes', 'RecipesController@store');
```

```
// app/Http/Controllers/RecipesController.php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class RecipesController extends Controller
{
    public function create()
    {
        return view('recipes.create');
    }

    public function store(Request $request)
    {
        $this->validate($request, [
            'title' => 'required|unique:recipes|max:125',
            'body' => 'required'
        ]);

        // Recipe 有效，继续保存
    }
}
```

在这里运行验证只使用了四行代码，实际上却做了很多。

首先，明确地定义了我们期望的字段，并将规则（这里用管道字符“|”进行分隔）分别应用到每个字段。

接下来，用 `validate()` 方法检查来自 `$request` 传入的数据（这意味着它可以使用 `$request->all()` 或 `$request->input()`），并确定数据是否有效。

如果数据有效，则 `validate()` 方法结束，我们可以继续使用控制器方法，保存数据或其他任何内容。

但如果数据无效，则抛出 `ValidationException` 异常，它包含了关于如何处理这个异常路由的说明。如果请求是 Ajax（或请求 JSON 作为响应）的方式，该异常将创建包含验证错误的 JSON 响应。如果不是，该异常将返回重定向到前一页，包括所有用户输入和验证错误，它会完美地重新填写失败的表单并显示一些错误。

更多关于 Laravel 的验证规则

在我们的示例中（如在文档中），使用了“管道”语法：`'fieldname': 'rule|otherRule|anotherRule'`。也可以使用数组语法实现：`'fieldname': ['rule', 'otherRule', 'anotherRule']`。

另外，还可以验证嵌套属性。可以使用 HTML 的数组语法，这一点很重要，因为这样就可以在 HTML 表单上拥有多个“用户（user）”，每个用户都有相关联的名称。如何进行验证，如下所示。

```
$this->validate($request, [
    'user.name' => 'required',
    'user.email' => 'required|email',
]);
```

这里无法介绍所有可能的验证规则，但是会介绍一些最常见的规则及其功能，如下所示。

需要的字段

```
required; required_if:anotherField,equalToThisValue;
required_unless:anotherField,equalToThisValue
```

字段必须包含特定类型的字符

```
alpha, alpha_dash, alpha_num, numeric, integer
```

字段必须包含特定的模式

```
email, active_url, ip
```

日期

```
after:date, before:date (date 可以是 strtotime() 可以处理的任何有效字符串)
```

数字

```
between:min,max, min:num, max:num, size:num (对字符串长度的 size 测试,值为整数,
数组的 count, 文件大小以 KB 为单位)
```

图像尺寸

```
dimensions:min_width=XXX; 也可以使用 and 或 or 组合 max_width, min_height, max_
height, width, height, 以及 ratio
```

数据库

`exists:tableName, unique:tableName` (希望查看与字段名相同的表列, 请参阅文档 (<http://bit.ly/2eMLZDI>) 了解如何自定义)

手动验证

如果不在控制器中运行, 或者由于某种其他原因, 以前描述的流程并不适合, 那么可以手动创建 `Validator` 实例并检查是否成功, 如示例 6-13 所示。

示例6-13 手动验证

```
Route::get('recipes/create', function () {
    return view('recipes.create');
});

Route::post('recipes', function (Illuminate\Http\Request $request) {
    $validator = Validator::make($request->all(), [
        'title' => 'required|unique:recipes|max:125',
        'body' => 'required'
    ]);

    if ($validator->fails()) {
        return redirect('recipes/create')
            ->withErrors($validator)
            ->withInput();
    }

    // Recipe 通过, 继续保存
});
```

我们把输入作为第一个参数, 把验证规则作为第二个参数, 然后传递给创建验证器的实例。验证器有一个可以检查的 `fails()` 方法, 并可以将其传递给重定向的 `withErrors()` 方法。

显示验证错误信息

我们已经在第 5 章中介绍了很多这方面的内容, 这里快速回顾一下如何从验证中显示错误。

控制器中的 `validate()` 方法 (以及它依赖的重定向 `withErrors()` 方法) 会将任何错误

都闪存到会话中。这些错误可以在 `$errors` 变量定义重定向到的视图中使用。请记住，作为 Laravel 的神奇之处，每当加载视图时，即使 `$errors` 变量是空的，它也可以使用，所以不必通过 `isset()` 检查它是否存在。

这意味着在每个页面上都可以运行如示例 6-14 所示的代码。

示例6-14 输出验证错误

```
@if ($errors->any())
    <ul id="errors">
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
        @endforeach
    </ul>
@endif
```

表单请求

在构建应用程序时，你可能会注意发现器方法中的某些模式，其中有一些重复的模式，例如输入验证、用户认证和授权及可能的重定向等。如果想要一个结构来进行规范化，并将这些常见行为从控制器方法中提取出来，那么可能需要用到 Laravel 的表单请求。

表单请求是一个自定义请求类，它的目的是映射到提交表单，负责验证请求、授权用户，并将用户重定向到失败的验证（可选）。每个表单请求通常都会显式地映射到单个 HTTP 请求，但不是全都这样，如“创建注释（Create Comment）”。

创建表单请求

可以使用 Artisan 创建新的表单请求。

```
php artisan make:request CreateCommentRequest
```

现在可以在 `app/Http/Requests/CreateCommentRequest.php` 中找到一个表单请求对象了。

每个表单请求类都提供一到两个公共方法。第一个方法是 `rules()`，它会返回一个该请求的验证规则数组。第二个方法（可选）是 `authorize()`，如果它返回 `true`，那么用户会被授权执行该请求；如果返回 `false`，那么该用户会被拒绝。可以通过示例 6-15 了解表单请求。

示例6-15 表单请求示例

```
<?php
```

```

namespace App\Http\Requests;

use App\BlogPost;
use App\Http\Requests\Request;

class CreateCommentRequest extends Request
{
    public function rules()
    {
        return [
            'body' => 'required|max:1000'
        ];
    }

    public function authorize()
    {
        $blogPostId = $this->route('blogPost');

        return auth()->check() && BlogPost::where('id', $blogPostId)
            ->where('user_id', auth()->user()->id)->exists();
    }
}

```

示例 6-15 的 `rules()` 部分是很容易理解的，让我们来看看 `authorize()` 部分。

这里从名为 `blogPost` 的路由中获取片段。这意味着这个路由的路由定义类似于这种形式：`Route::post('blog Posts/{blogPost}', function () { // Do stuff })`。这里命名了路由参数 `blogPost`，因此请求可以访问 `$this->route('parameter name')`。

接下来查看用户是否登录。如果是，则要考虑是否存在由当前登录用户所拥有的标识符。这里的重点是返回 `true` 意味着用户被授权执行指定的操作（在这种情况下会创建一个 `comment` 注释），而如果返回 `false` 则表示用户没有被授权。

使用表单请求

现在已经创建了一个表单请求对象，那么如何使用它呢？任何指定（`type-hint`）表单请求的路由（闭包或控制器方法）将受益于该表单请求的定义。

让我们试一下，以示例 6-16 为例。

示例6-16 使用表单请求

```

Route::post('comments', function (App\Http\Requests\CreateCommentRequest

```

```
$request) {  
    // Store comment  
});
```

你可能想知道调用表单请求的位置，但是 Laravel 已经为我们做好了。它会验证用户输入并授权请求。如果输入无效，它将像控制器中的 `validate()` 方法一样，将用户重定向到上一页并保留输入，同时传递相应的错误消息。如果用户未经授权，Laravel 将返回 403 Forbidden 错误，而不执行路由代码。

Eloquent 模型质量分配

到目前为止，我们一直在控制器级别中进行验证，这绝对是最好的开始，但是也可以在模型级别过滤传入的数据。

将整个表单的输入直接传递给数据库模型是一种常见的模式。在 Laravel 中，相关实现如示例 6-17 所示。

示例6-17 将整个表单传递给Eloquent模型

```
Route::post('posts', function (Request $request) {  
    $newPost = Post::create($request->all());  
});
```

这里假设终端用户是善良而不是恶意的，并且只保留了我们希望用户编辑的字段，也许是标题或正文。

但是如果终端用户可以猜测或辨别出在该 `posts` 表上有一个 `author_id` 字段，该怎么办呢？如果用户使用浏览器工具添加一个 `author_id` 字段并将该 ID 设置为别人的 ID，并且冒充他人并且通过创建与别人相关的虚假博客帖子来模仿对方，该怎么办呢？

Eloquent 有一种叫作“大规模分配 (mass assignment)”的方法，可以以这种方式（例如使用模型的 `$fillable` 属性）或者以不可填充的黑名单字段的方式（例如使用模型的 `$guarded` 属性）将可填充的字段列入白名单。如果要了解更多，则可以查看第 8 章。

我们可以填写如示例 6-18 所示的模型，使应用程序变得安全。

示例6-18 从恶意质量分配中保护Eloquent模型

```
<?php  
  
namespace App;  
  
use Illuminate\Database\Eloquent\Model;
```

```
class Post extends Model
{
    // 在 author_id 字段上禁用质量分配
    protected $guarded = ['author_id'];
}
```

通过设置 `author_id`，我们确保恶意用户将不能够手动添加该字段的值并将其发送到应用程序的表单的内容中来覆盖该字段的值。



使用 `$request->only()` 进行双重保护

尽管通过质量分配来保护模型的工作很重要，但在分配的终端也需要小心。与其使用 `$request->all()`，不如考虑使用 `$request->only()`，这样可以指定要传递给模型的字段。

```
Route::post('posts', function (Request $request) {
    $newPost = Post::create($request->only([
        'title',
        'body'
    ]));
});
```

{{ 与 {!!

每次在用户创建的网页上显示内容时，都需要防范恶意输入，例如脚本注入。

假设允许用户在网站上写博文，但是不想让他们在毫无戒心的访客浏览器中注入恶意的 JavaScript，那么需要转义在页面上显示的任何用户输入，从而避免这种情况。

幸运的是，使用 Laravel 几乎完全可以满足要求。如果使用 Laravel 的 Blade 模板引擎，默认的“echo”语法（`{{ $stuffToEcho }}`）会通过 `htmlentities()` 运行输出（PHP 是使用户内容安全 echo 的最佳方法）。实际上必须做一些额外的工作，以避免转义输出，这里可以使用 `{!! $stuffToEcho !!}` 语法实现。

测试

如果想要测试与用户输入的交互，则很可能想要模拟有效和无效的用户输入，并且确保如果输入无效，则用户被重定向；如果输入有效，则最终会出现在适当的位置（如数据库）。

Laravel 的端到端应用测试使这个步骤变得简单。这里从一个无效的路由开始，假设此时期望被拒绝，如示例 6-19 所示。

示例6-19 测试无效输入并被拒绝

```
public function test_input_missing_a_title_is_rejected()
{
    $this->post('posts', ['body' => 'This is the body of my post']);
    $this->assertRedirectedTo('posts/create');
    $this->assertSessionHasErrors();
    $this->assertHasOldInput();
}
```

这里可以断言，无效输入后，用户被重定向，出现错误，旧输入正确传回。可以看到这里使用 Laravel 添加了几个自定义 PHPUnit 声明。

如果路由成功，那么该如何测试呢？可以查看示例 6-20。

示例6-20 测试有效输入并对输入进行处理

```
public function test_valid_input_should_create_a_post_in_the_database()
{
    $this->post('posts', ['title' => 'Post Title', 'body' => 'This is the body']);
    $this->seeInDatabase(['title' => 'Post Title']);
}
```

请注意，如果正在使用数据库测试某些内容，则需要了解有关数据库迁移和事务的更多信息。更多的信息可以参阅第 12 章。

本章小结

有很多方法可以获得相同的数据：Request facade、request() 全局助手函数，以及注入 Illuminate\Http\Request 的实例。每种方法都可以获取所有输入、部分输入或特定的数据；对于文件和 JSON 输入，有时可以使用一些特殊的方法。

URL 路径片段也是用户输入的可能来源，也可以通过请求工具进行访问。

可以使用 Validator::make() 手动执行验证，或者使用 \$this->validate() 控制器方法及表单请求自动进行。在验证失败后，自动化工具将用户重定向到上一页，存储所有旧输入并传递错误。

视图和 Eloquent 模型也需要保护，免受恶意的用户输入。可以使用双大括号语法 ({{ }}) 保护 Blade 视图，该语法转义用户输入，并通过 \$request->only() 实现仅将特定字段传递到批量方法的方式来保护模型，也可以通过对模型本身定义质量分配规则实现。

Artisan和Tinker

新版本的 PHP 框架希望通过更多的交互来取代命令行。Laravel 主要提供了 3 种实现命令行交互的工具：Artisan, 它有一套内置的命令行操作可以自定义进行扩展；Tinker, 为应用提供了 REPL 或者交互的 shell；安装器, 在第 2 章中我们已经详细介绍过了。

Artisan 入门

如果详细阅读了这本书每一章的内容, 那么应该已经知道怎么使用 Artisan 命令了。其类似下面的格式。

```
php artisan make:controller PostsController
```

如果查看应用的根目录, 那么会发现 *artisan* 已经在 PHP 的文件夹下了。这也是需要用 `php artisan` 来调用的原因, 这样会用 PHP 对 *artisan* 文件进行解析。之后所有的文件都将作为参数传入 Artisan。



Symfony 控制台语法

Artisan 实际上位于 Symfony 控制台组件 (<http://symfony.com/doc/current/components/console.html>) 的顶层, 所以如果对 Symfony 控制台命令非常熟悉, 就可以在家办公了。

由于应用程序的 Artisan 命令列表可以通过程序包或应用程序的特定代码进行更改, 所以最好能对每一个新的应用检查可用的命令。

当得到了所有可用的 Artisan 命令之后, 便可以在项目的根目录下运行 `php artisan list` (如果没有参数, 那么只运行 `php artisan` 效果也是一样的)

Artisan 的基本命令

由于篇幅有限，这里没办法介绍所有的 Artisan 命令，但是会尽量覆盖更多的命令。下面介绍 Artisan 的基本命令。

- `help` 帮助命令，例如 `php artisan help commandName`。
- `clear-compiled` 删除 Laravel 的编译文件（就像一个内部缓存）。当遇到问题却不知道原因的时候，可以先尝试运行这个命令。
- `down` 把应用切换到“维护模式”以解决错误、迁移或者其他运行方式。`up` 可以在“维护模式”里恢复应用。
- `env` 显示当时 Laravel 的运行环境。它等效于在应用中输入 `app()->environment()`。
- `migrate` 迁移数据库。
- `optimize` 通过把重要的 PHP 类缓存到 `bootstrap/cache/compile.php` 来优化应用。
- `serve` 部署一个 PHP 服务器到 `localhost:8000`（可以通过 `--host` 和 `--port` 自定义修改主机名和端口号）。
- `tinker` 打开 Tinker 的 REPL，相关内容会在接下来的章节中具体介绍。

选项

在介绍其他 Artisan 命令之前，先来了解一些好用的 Artisan 命令选项。这些选项在任何时刻都可以使用。

- `-q` 隐藏所有的输出。
- `-v`、`-vv` 和 `-vvv` 对应三种不同形式的输出（正常、详细、调试）。
- `--no-interaction` 不会询问任何有关交互的问题，所以使用这个命令不会中断自动化流程。
- `--env` 可以定义 Artisan 命令操作的环境，如 `local`、`production` 等。
- `--version` 显示应用运行的 Laravel 版本。

Artisan 命令和常用的 shell 命令的选项非常相似：可以手动运行它们，但是在某些情况下它们也可以作为自动化过程的函数。

有很多自动化部署过程可以得益于 Artisan 命令。每次部署一个应用程序的时候，可以运行 `php artisan optimize`。例如 `-q` 和 `--no-interaction` 之类的选项，这样可以保证用来部署的脚本不会被人人为地中断，从而可以一直顺利运行下去。

组合命令

其他可以开箱即用的命令可以按照上下文进行分组。这里没有包含所有命令，但会尽可能多的扩展每一个命令的内容。

app

这个命令只包含 `app:name`，允许用选定的命名空间替换每一个默认的、顶层的 `App\` 命名空间实例。例如 `php artisan app:name MyApplication`。

auth

这里只涉及 `auth:clear-resets`，它将刷新所有过期的密码，然后通过数据库中的令牌重置。

cache

`cache:clear` 会清除缓存 `cache:table`，如果想使用数据库缓存驱动，则可以用这个命令迁移数据库。

config

为了更快地查阅，`config:cache` 会缓存所有的配置。使用 `config:clear` 可以清理缓存。

db

如果已经配置了数据库的 `seeder`，便可以用 `db:seed` 命令来填充数据库。

event

根据 `EventServiceProvider` 的定义，`event:generate` 会建立缺失事件和事件监听文件。更多相关知识，请参阅第 16 章。

key

`key:generate` 会在 `.env` 文件中创建一个随机的应用加密密钥。



重新运行 `artisan key:generate` 会丢失原有的密钥

只需要运行一次 `php artisan key:generate`，也就是第一次在新环境里设置应用程序运行，因为这个密钥是用来对数据进行加密的。如果在数据存储之后对其进行修改，这个数据就不能访问了。

make

`make:auth` 为着陆界面用户仪表盘及登录注册界面提供视图和对应的路由器。

其他所有的 `make: action` 创建了单独的条目和对应的参数。如果想要学习更多命令参数，可以使用 `help` 命令查看文档。

例如可以运行 `php artisan help make:migration` 命令，也可以通过添加 `--create=tableNameHere` 参数创建一个数据迁移（注意，创建该迁移需要文件中已经具有创建表的语法）来学习这个命令。完整的代码是 `php artisan make:migration create_posts_table --create=posts`。

migrate

之前提到可以通过 `migrate` 命令来运行迁移，但是这里可以运行所有跟迁移相关的命令。通过 `migrate:install` 可以创建迁移及追踪执行的迁移；通过 `migrate:reset` 可以重置迁移；通过 `migrate:refresh` 可以进行迁移的重置和再次运行；通过 `migrate:rollback` 可以进行迁移回滚；通过 `migrate:status` 可以查看迁移的状态。

notifications

`notifications:table` 用于生成一个能产生数据库通知表的迁移。

queue

本书第 16 章会进一步讲解 Laravel 的队列，它的基本思想是可以将作业推送到远程队列中，以便工作人员一个接一个地执行。下面这些组合命令提供了所有需要跟队列交互的工具。比如 `queue:listen`，开始监听一个队列；`queue:table`，为数据库支持队列创建一个迁移 `queue:flush`，刷掉所有失败的队列任务。更多相关内容请参阅第 16 章。

route

如果运行 `route:list`，那么可以看到应用中每一个路由器的定义，包括每个路由器的方法、路径、名字、控制器 / 闭包动作和中间件。可以通过 `route:cache` 来缓存路由器的定义，以便更快地查阅，也可以用 `route:clear` 清理控制器。

schedule

我们会在第 16 章介绍 Laravel 中类似 cron 的调度程序，但是为了使其能正确工作，需要先设置系统的 cron：每分钟运行一次 `schedule:run`。

```
* * * * * php /home/myapp.com/artisan schedule:run >> /dev/null 2>&1
```

这个 Artisan 命令用于实现定时控制执行 Laravel 的核心服务。

session

`session:table` 通过数据库支持的会话来为应用创建一个迁移。

storage

`storage:link` 创建一个从 `public/storage` 到 `storage/app/public` 的符号链接。这是 Laravel 应用程序中的常见约定，其目的是更容易地把用户上传的内容（或者其它在 `storage/app` 中常见的终端应用文件）放到一个能通过公共 URL 访问的位置。

vendor

一些 Laravel 特定的包需要“公开”一些内容，这样既可以在 `public` 目录下提供服务也可以对其进行修改。无论哪种方式，这些包在 Laravel 中注册“可公开的内容”，当运行 `vendor:publish` 命令的时候，这些包就会在特定的位置上显示。

view

Laravel 的视图渲染引擎会自动缓存视图。它通常会处理自己的无效缓存，但是如果发现它卡住了，则可以运行 `view:clear` 来清理缓存。

书写常见的 Artisan 命令

到目前为止，已经介绍了 Laravel 可以直接使用的命令，下面让我们动手写一下。

首先需要先运行一行 Artisan 命令。运行 `php artisan make:command YourCommandName`，在 `app/Console/Commands/{YourCommandName}.php` 下生成一个新的 Artisan 命令。



`php artisan make:command`

5.2 `make:command` 的命令签名已经变更过很多次了。一开始是 `command:make`，但是在 Laravel 5.2 版本中变成了 `console:make`，然后是 `make:console`。

最后在 Laravel 5.3 版本中终于确定下来了。所有的生成器都是在 `make:` 命名空间下的，现在生成一个新的 Artisan 命令则通过 `make:command` 实现。

第一个参数应该是命令的类名，也可以选择性地添加一个 `--command` 参数，用来定义终端命令（例如：`appname:action`）。

下面输入命令。

```
php artisan make:console WelcomeNewUsers --command=email:newusers
```

下面通过示例 7-1 来看一下执行效果。

示例7-1 Artisan命令的默认架构

```
<?php
```

```
namespace App\Console\Commands;

use Illuminate\Console\Command;

class WelcomeNewUsers extends Command
{
    /**
     * 控制台命令名称和签名
     *
     * @var string
     */
    protected $signature = 'email:newusers';

    /**
     * 控制台命令描述
     *
     * @var string
     */
    protected $description = 'Command description';

    /**
     * 创建一个新的命令实例
     *
     * @return void
     */
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * 执行控制台命令
     *
     * @return mixed
     */
}
```

```

        */
        public function handle()
        {
            //
        }
    }
}

```

可以看到这里可以很容易地定义命令签名、显示在命令列表的帮助文档、作用于实例（__construct()）及执行（handle()）的命令。

注册命令

为了使应用真正能够使用，还差最后一步，也就是注册这个命令。

打开 `app/Console/Kernel.php`，可以在 `$commands` 属性下看到一个命令类名的数组。为了注册新命令，需要把它的类添加到这个数组里。这里可以把它写出来，或者只是使用类名访问器 `::class`，如示例 7-2 所示。

示例7-2 在控制台内核中注册新命令

```

class Kernel extends ConsoleKernel
{
    /**
     * 应用提供的 Artisan 命令
     *
     * @var array
     */
    protected $commands = [
        \App\Console\Commands\WelcomeNewUsers::class,
    ];
}

```



基于闭包的命令

如果希望命令的定义过程更加简单，那么可以把它写到闭包里而不是在 `routes/console.php` 中定义类。在本章里，所有讨论的内容都采用了这样的方式，也可以只在 `routes/console.php` 文件里定义，然后注册命令。

```

// routes/console.php
Artisan::command(
    'password:reset {userId} [--sendEmail]',
    function ($userId, $sendEmail) {
        // 进行某些操作 ...
    }
);

```

示例命令

邮件和 Eloquent 分别会在第 8 章和第 15 章中介绍，而这里使用简单的 `handle()` 方法，如示例 7-3 所示。

示例7-3 Artisan命令中的handle方法示例

```
...
class WelcomeNewUsers extends Command
{
    public function handle()
    {
        User::signedUpThisWeek()->each(function ($user) {
            Mail::send(
                'emails.welcome',
                ['name' => $user->name],
                function ($m) use ($user) {
                    $m->to($user->email)->subject('Welcome!');
                }
            );
        });
    }
}
```

现在只要运行 `php artisan email:newusers` 命令，就会自动获取这周新注册的用户，然后给这些用户发送欢迎邮件。

如果想要在邮件注入用户依赖，那么可以在实例化命令的时候修改命令的构造函数和 Laravel 容器中的内容。

如果在例 7-3 中注入了依赖然后从服务类中提取行为，那么效果如示例 7-4 所示。

示例7-4 重构的代码

```
...
class WelcomeNewUsers extends Command
{
    public function __construct(UserMailer $userMailer)
    {
        parent::__construct();

        $this->userMailer = $userMailer
    }

    public function handle()
    {
        $this->userMailer->welcomeNewUsers();
    }
}
```


保持简单性

之后的代码中可能还会调用 Artisan 命令，可以使用这些命令来封装应用逻辑。这是一种在 Laravel 社区中很常见的应用方法。

但是可以看到 Laravel 官方文档建议用服务类来代替封装应用逻辑的包，然后把服务注入命令。控制台命令和容器类似，它们都没有作用域类将传入的请求分配到正确的行为中。

参数和选项

新的命令中的 `$signature` 属性看起来只包含了命令名称，但是可以在里面定义命令的任何参数和选项，它有固定的语法。

在研究语法之前，我们先看一个例子，如下所示。

```
protected $signature = 'password:reset {userId} [--sendEmail]';
```

必选参数、可选参数和默认参数

用大括号把必须要有的参数包起来。

```
password:reset {userId}
```

对于可选的参数，在其后面加一个问号。

```
password:reset {userId?}
```

默认的可选参数如下。

```
password:reset {userId=1}
```

选项、待定值和默认值

选项和参数类似，但是选项需要加前缀 `--`，它也可以不赋值。同样的，添加一个基本的选项也需要用大括号把它包起来。

```
password:reset {userId} [--sendEmail]
```

如果想为选项赋值，那么在它的 `signature` 属性后跟一个等号 (`=`)。

```
password:reset {userId} [--password=]
```

如果想要传递一个默认值，直接把它加到等号后面就可以了。

```
password:reset {userId} {--queue=default}
```

参数数组和选项数组

在参数和选项中，如果想使用数组则需要使用符号 *，代码如下。

```
password:reset {userIds*}
```

```
password:reset {--ids=*}
```

这种使用数组的方式与示例 7-5 很相似。

示例7-5 在Artisan命令中使用数组

```
// 参数
```

```
php artisan password:reset 1 2 3
```

```
// 选项
```

```
php artisan password:reset --ids=1 --ids=2 --ids=3
```



参数数组必须是最后一个参数

参数数组包含了每个定义参数，然后把它们添加到数组元素中，所以在 Artisan 命令的属性中参数数组必须位于最后。

输入描述

前面提到过可以使用 `artisan help` 命令来查看内置的 Artisan 命令，该方式同样也可以获得自定义命令的信息。使用时只需要在大括号里加一个冒号和描述文字，详见示例 7-6。

示例7-6 为Artisan参数和选项定义描述文字

```
protected $signature = 'password:reset
                        {userId : The ID of the user}
                        {--sendEmail : Whether to send user an email}';
```

输入

在命令的 `handle()` 方法中具体如何操作呢？这里有两种方式可以用于检索参数和选项的值。

argument()

没有参数的 `$this->argument()` 会返回一个包含所有参数的数组（数组第一个元素是命

令名)；当传入参数时将会返回指定参数的值，代码如下。

```
// 定义 "password:reset {userId}":
php artisan password:reset 5

// $this->argument() 返回数组
[
    "command": "password:reset",
    "userId": "5",
]

// $this->argument('userId') 返回字符串
"5"
```

option()

没有参数的 `$this->option()` 会返回一个包含所有选项的数组，包括默认为 `false` 或者 `null` 的选项；当传入参数时，将会返回指定选项的值，代码如下。

```
// 定义 "password:reset [--userId=]":
php artisan password:reset --userId=5

// $this->option() 返回数组
[
    "userId" => "5"
    "help" => false
    "quiet" => false "verbose" => false
    "version" => false
    "ansi" => false
    "no-ansi" => false
    "no-interaction" => false
    "env" => null
]
// $this->option('userId') 返回字符串
"5"
```

示例 7-7 展示了在 Artisan 命令的 `handle()` 方法中使用 `argument()` 和 `option()`。

示例7-7 从Artisan命令中获取输入

```
public function handle()
{
    // 所有参数，包括命令名称
    $arguments = $this->argument();

    // 仅 'userId' 参数
```

```

    $userid = $this->argument('userId');

    // 所有选项，包括默认选项，如 'no-interaction' 和 'env'
    $options = $this->option();

    // 仅 'sendEmail' 选项
    $sendEmail = $this->option('sendEmail');
}

```

提示

在 `handle()` 代码中还有一些获取用户输入的方法，它们都会在执行命令的时候提示用户进行输入。

`ask()`

提示用户输入文本，代码如下。

```
$email = $this->ask('What is your email address?');
```

`secret()`

提示用户输入文本，但是会用星号 * 来隐藏输入内容，代码如下。

```
$password = $this->ask('What is the DB password?');
```

`confirm()`

提示用户回复是 / 否，返回一个布尔值，代码如下。

```

if ($this->confirm('Do you want to truncate the tables?')) {
    //
}

```

除 `y` 或者 `Y` 的输入外，其余都会被当做 “no”。

`anticipate()`

提示用户输入任何文本并提供自动补全建议，用户也可以输入其他内容。

```

$album = $this->anticipate('What is the best album ever?', [
    "The Joshua Tree", "Pet Sounds", "What's Going On"
]);

```

`choice()`

提示用户选择一个选项，如果用户没有选择，那么最后一个参数就会使用默认值。

```
$winner = $this->choice(
    'Who is the best football team?',
    ['Gators', 'Wolverines'],
    0
);
```

需要注意的是，最后一个参数是数组形式的键。因为我们传入了一个不相关的数组，所以 Gators 的键就是零，也可以为数组定义其他的键。

```
$winner = $this->choice(
    'Who is the best football team?',
    ['gators' => 'Gators', 'wolverines' => 'Wolverines'],
    'gators'
);
```

输出

在执行命令的时候，可能想给用户传递一些信息。最常见的用法就是用 `$this->info()` 来输出基本的提示文字。

```
$this->info('Your command has run successfully.');
```

同样也可以用其他命令来输出 `comment()` (橙色)、`question()` (高亮青色)、`error()` (高亮红色) 和 `line()` (没有颜色)。

值得注意的是，每台电脑的颜色会有一定的差别，但是它们会尽量与本地计算机的标准保持一致以便和终端用户进行交互。

输出表

表组件将有助于更方便地在数据中完全创建 ASCII 表，见示例 7-8。

示例7-8 在 Artisan 命令中输出表

```
$headers = ['name', 'email'];

$data = [
    ['Dhriti', 'dhriti@amrit.com'],
    ['Moses', 'moses@gutierrez.com']
];

// 或者可以从数据库中得到类似的数据
// $data = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $data);
```

在示例 7-8 中有两组数据，一个是数据头，一个是数据本身。它们中的每一行都包含两个单元格：第一个单元格是名称，第二个单元格是邮件。这样在 Eloquent 中调用的数据（限定只提取名称和电子邮件信息）就可以与数据头进行匹配。

表的输出如示例 7-9 所示。

示例7-9 Artisan表输出

```
+-----+-----+
| Name   | Email               |
+-----+-----+
| Dhriti | dhriti@amrit.com    |
| Moses  | moses@gutierrez.com |
+-----+-----+
```

进度条

如果之前运行过 `npm install` 命令，那么应该已经看到过命令行的进度条了。让我们创建一个进度条，如示例 7-10 所示。

示例7-10 Artisan进度条

```
$totalUnits = 10;
$this->output->progressStart($totalUnits);

for ($i = 0; $i < $totalUnits; $i++) {
    sleep(1);

    $this->output->progressAdvance();
}

$this->output->progressFinish();
```

这段代码具体作用是什么？首先告诉系统，我们需要 unit 的个数。假设一个 unit 对应一个用户，如果有 350 个用户，那么这个进度条会在屏幕中被分成 350 份。每次运行 `processAdvance()` 的时候，进度条会增加 1/350。当应用执行完之后，可以通过 `progressFinish()` 命令来显示完整的进度条。

在其他代码中调用 Artisan 命令

尽管 Artisan 命令一开始被用来在命令行中运行，但是也可以在其他常见的代码中调用它。

最简单的办法就是用 `Artisan facade`。也可以用 `Artisan::call()`（返回命令退出码）

来调用命令，或者用 `Artisan::queue()` 把命令放到队列中。

这两种方法都需要两个参数：第一个是终端命令 (`password:reset`)；第二个是传进去的参数数组。下面通过示例 7-11 来看一下它是如何和参数及选项一起工作的。

示例7-11 在其他代码中调用Artisan命令

```
Route::get('test-artisan', function () {
    $exitCode = Artisan::call('password:reset', [
        'userId' => 15, '--sendEmail' => true
    ]);
});
```

可以看到的参数是通过参数名称传入的，没有值的选项可以传入 `true` 或者 `false`。

在其他命令中调用 Artisan 命令，可以使用 `$this->call`（等效于 `Artisan::call()`）或者 `$this->callSilent`（等效于输出，但是会覆盖所有的输出），详见示例 7-12。

示例7-12 在其他命令中调用Artisan命令

```
public function handle()
{
    $this->callSilent('password:reset', [
        'userId' => 15
    ]);
}
```

最后可以注入一个 `Illuminate\Contracts\Console\Kernel` 规约的实例，并且使用它的 `call()` 方法。

Tinker

Tinker 是一种 REPL，也被叫作读取 - 执行 - 输出循环。如果之前在 Ruby 中用过 IRB，那么应该会对 REPL 非常熟悉。

REPL 会弹出一个提示，和命令行中的提示很相似，类似于应用中的“等待”状态。在 REPL 中输入命令后点击 Return，然后程序会根据输入的内容响应并打印结果。

示例 7-13 提供了一个简单的示例，从而可以 Tinker 的基本工作流程和有效性进行快速的了解。我们通过 `php artisan tinker` 来打开 REPL，它会显示空格提示 (`>>>`)，每次响应输入命令的内容都会在 `=>` 后面打印出来。

示例7-13 使用Tinker

```
php artisan tinker
```

```
>>> $user = new App\User;
=> App\User: {}
>>> $user->email = 'matt@mattstauffer.co';
=> "matt@mattstauffer.co"
>>> $user->password = bcrypt('superSecret');
=> "$2y$10$TWPGB7e8d1bvJ1q5kv.VDUGfYDnE9gANL4mleuB3htIY2dxcQfQ5"
>>> $user->save();
=> true
```

在上面的代码中创建了一个新的用户、设置一些数据，然后把数据存到数据库中，这些都是真实的。如果这是一个产品应用，那我们就已经在系统中创建了一个新的品牌。

这个特性让 Tinker 成为一个非常有效的、用于简单数据库交互的工具，并且可以用于尝试新的想法。对于不知道应该放在源码哪个位置的代码片段，也可以用 Tinker 来运行。

Tinker 是由 Psy Shell (<http://psysh.org/>) 提供支持的。

测试

到目前为止，我们已经知道了怎么在代码中调用 Artisan 命令，这样更便于测试也能保证程序能够达到期望的效果，如示例 7-14 所示。

示例7-14 在测试中调用Artisan命令

```
public function test_empty_log_command_emptyies_logs_table()
{
    DB::table('logs')->insert(['message' => 'Did something']);
    Artisan::call('logs:empty');
    $this->assertCount(0, DB::table('logs')->get());
}
```

通常来说，facade 很容易置换。但是如果不这样做，也可以在 Artisan 命令的构造器中注入依赖，这样就能在测试的时候更容易置换。

Artisan 的 facade 类为 Illuminate\Contracts\Console\Kernel 规约提供了入口，所以如果想在代码中避免使用 facade，可以注入一个 Kernel 实例，然后使用 call() 方法，如示例 7-15 所示。

示例7-15 不使用Artisan的facade，而是注入一个Kernel

```
use Illuminate\Contracts\Console\Kernel;
...
class NightlyCleanup extends Job
```



```
{  
  
    ...  
    public function handle(Kernel $kernel)  
    {  
        // ... 进行其他操作  
        $kernel->call('logs:empty');  
    }  
}
```

本章小结

Artisan 命令是 Laravel 的命令行工具，Laravel 一般可以直接使用它。创建自己的 Artisan 命令，然后在命令行或者其他代码中调用也很简单。

Tinker 作为 REPL 能够更简单地访问应用环境以及与真实的代码和数据交互。

数据库和Eloquent

Laravel 提供了很多用于与数据库交互的工具，其中最好用的就是 Eloquent，也就是 Laravel 的 ActiveRecord ORM（object-relational-mapper）关系型对象映射器。

Eloquent 是 Laravel 最流行并且最有影响力的一个特性，它是 Laravel 不同于大多数 PHP 框架的一个典型例证。在 DataMapper ORM 中，这个功能很有用但是也很复杂，Eloquent 会简化中间的操作，在每张表中会有一个类用于在表中获取、表示和持久化数据。

无论是否使用 Eloquent，都会感受到 Laravel 提供的其他数据库工具的很多好处。所以在深入研究 Eloquent 之前，让我们先从 Laravel 数据库的基本功能开始介绍：迁移 (migration)、填充 (seeder) 和查询构造器 (query builder)。

接着我们会更多地讨论 Eloquent：定义模型；插入、更新和删除；使用访问器转换器和属性转换实现自定义响应以及关系。本章的内容比较多，可能一下完全接受有些困难，让我们一步一步来。

配置

在讨论怎样使用 Laravel 的数据库工具之前，先看看怎样配置数据库的证书和连接。

可以通过访问 `config/database.php` 来配置数据库。和 Laravel 中很多其他配置一样，这里可以定义多个“连接”，然后设定默认使用的代码。

数据库连接

默认每一个连接类型只有一个连接，如示例 8-1 所示。

示例8-1 数据库默认连接列表

```
'connections' => [  
  
  'sqlite' => [  
    'driver' => 'sqlite',  
    'database' => database_path('database.sqlite'),  
    'prefix' => '',  
  ],  
  
  'mysql' => [  
    'driver' => 'mysql',  
    'host' => env('DB_HOST', 'localhost'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'charset' => 'utf8',  
    'collation' => 'utf8_unicode_ci',  
    'prefix' => '',  
    'strict' => false,  
    'engine' => null,  
  ],  
  
  'pgsql' => [  
    'driver' => 'pgsql',  
    'host' => env('DB_HOST', 'localhost'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'charset' => 'utf8',  
    'prefix' => '',  
    'schema' => 'public',  
  ],  
  
  'sqlsrv' => [  
    'driver' => 'sqlsrv',  
    'host' => env('DB_HOST', 'localhost'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'charset' => 'utf8',  
    'prefix' => '',  
  ],  
]
```

可以创建新命名的连接，也可以在这些连接中设置驱动（MySQL、Postgres 等）。所以这里并不强制限制每个驱动默认只能有一个连接。

在每个连接中都可以定义连接及自定义每一个连接类型相关的属性。

这里使用多个驱动有以下理由。首先，“连接”部分是一个简单的模板，它可以直接使用，同时它可以轻松开启使用任何数据库连接类型的应用程序。在很多应用中选择数据库连接填写信息，甚至删除其他想要的内容。笔者一般都会保持这些信息以免后面可能会用到它们。

但在某些情况下，对于同一个应用可能需要多个连接。例如，使用不同的数据库连接两个不同类型的数据，或者读取一种数据类型，但是写入另一种数据类型。多个连接使这种操作成为可能。

其他数据库配置选项

`config.database` 的配置部分有一些其他设置，例如可以配置 Redis 入口修改用于迁移的表名、设置默认连接及决定非 Eloquent 的调用是否返回 `stdClass` 或者数组实例。

在 Laravel 中允许多个连接的服务都可以备份到数据库或者文件中，缓存器可以使用 Redis 或者 Memcached、数据库可以用 MySQL 或者 PostgreSQL，这样既可以定义多个连接，也可以选择一个默认的连接（也就是当没有明确指定一个连接的时候，Laravel 会默认使用的连接）。下面的代码用于请求一个特定的连接

```
$users = DB::connection('secondary')->select('select * from users');
```

迁移

现在的框架，比如 Laravel，能更容易地通过代码驱动的迁移来定义数据库结构，每一个新的表、列、索引和键都可以在代码中定义，在任何新的环境中都可以在数秒内实现将裸机数据库完整地与应用程序进行数据同步。

定义迁移

迁移是一个单独的文件，它定义了两项内容，分别是在运行迁移 *up* 和 *down* 时所需的修改。

迁移中的“up”和“down”

迁移总是按照日期顺序运行的。迁移文件的命名都类似于这种形式：*2014_10_12_000000_create_users_table.php*。当一个新系统被迁移到这个系统，就会从最早的日期开始抓取每一个迁移然后运行它的 `up()` 方法，也就是在这个时间点“往上”进行迁移。迁移系统也允许“回滚”最近的一系列迁移。它会抓取每一个迁移然后运行 `down()` 方法，然后撤销任何向上迁移所做的改动。

所以，可以把迁移的 `up()` 方法理解为“执行”迁移，`down()` 就是“撤销”这个迁移。

示例 8-2 展示了如何在 Laravel 中进行默认“创建用户表”迁移操作。

示例8-2 Laravel中的默认“创建用户表”迁移

<?php

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUsersTable extends Migration
{
    /**
     * 运行迁移
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table){
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password', 60);
            $table->rememberToken();
            $table->timestamps();
        });
    }
    /**
     * 反转迁移
     *
     * @return void
     */
    public function down()
```

```

    {
        Schema::drop('users');
    }
}

```

可以看到的这里有 `up()` 和 `down()` 方法。`up()` 表示迁移创建一个名为 `users` 的新表，并且在新表里面包含几个字段；`down()` 表示删除 `users` 表。

创建迁移

在第 7 章中我们已经介绍了在 Artisan 命令中创建迁移文件。命令 `php artisan make:migration` 只包含一个参数，就是迁移的名字。例如我们创建一个示例中的表，就可以通过执行命令 `php artisan make:migration create_users_table` 实现。

可以选择向命令中传入两个标志性内容：`--create = table_name` 可以创建名为 `table_name` 的表进行迁移；`--table = _table_name_` 会将预修改迁移到现有的表中。下面是几个例子。

```

php artisan make:migration create_users_table
php artisan make:migration add_votes_to_users_table --table=users
php artisan make:migration create_users_table --create=users

```

创建表

在默认的 `create_users_table` 迁移中，我们已经看到了迁移取决于 `Schema facade` 及其方法。所有能在迁移中进行的操作都依赖于 `Schema` 中的方法。

为了在迁移中创建一个新表，我们需要使用 `create()` 方法。第一个参数表示表名，第二个参数表示定义列的闭包，示例代码如下。

```

Schema::create('tablename', function (Blueprint $table) {
    // 创建列
});

```

创建列

为了在表中创建一个新的列，无论是创建表调用还是修改表调用，都可以使用传递到闭包中的 `Blueprint`，示例代码如下。

```

Schema::create('users', function (Blueprint $table) {
    $table->string('name');
});

```

让我们来看一下在 `Blueprint` 实例中创建列的不同方法。以 `MySQL` 为例，但是如果使

用的是其他数据库，那么在 Laravel 中也不会有太多差别。

以下是简单的 Blueprint 方法。

```
integer(colName), tinyInteger(colName), smallInteger(colName), mediumInteger(colName), bigInteger(colName)
```

添加一个 INTEGER 类型列，或者其他变体之一。

```
String(colName, OPTIONAL length)
```

添加一个 VARCHAR 类型列。

```
binary(colName)
```

添加一个 BLOB 类型列。

```
boolean(colName)
```

添加一个 BOOLEAN 类型列 (在 MySQL 中是 TINYINT(1))。

```
char(colName, length)
```

添加一个 CHAR 类型列。

```
datetime(colName)
```

添加一个 DATETIME 类型列。

```
decimal(colName, precision, scale)
```

添加一个 DECIMAL 类型列，指定精度和数值范围，例如 `decimal('amount',5,2)` 指定精度为 5、数值范围为 2。

```
double(colName, total digits, digits after decimal)
```

添加一个 DOUBLE 类型列，例如 `double('tolerance', 12, 8)` 指定数字长度为 12 位、小数点后保留 8 位数字，如 7204.05691739。

```
enum(colName, [choiceOne, choiceTwo])
```

添加一个 ENUM 类型列，包括提供的选择。

```
float(colName)
```

添加一个 FLOAT 类型列 (等同于 MySQL 中的 double)。

`json(colName)` 和 `jsonb(colName)`

添加一个 JSON 或 JSONB 类型列 (或者是 Laravel 5.1 中的 TEXT 列)。

`text(colName)`, `mediumText(colName)`, `longText(colName)`

添加一个 TEXT 类型列 (大小可以不同)。

`time(colName)`

添加一个 TIME 类型列。

`timestamp(colName)`

添加一个 TIMESTAMP 类型列。

`uuid(colName)`

添加一个 UUID 类型列 (在 MySQL 中是 CHAR(36))

在 Blueprint 中还有一些特殊的 (join) 方法，如下所示。

`increments(colName)` 和 `bigIncrements(colName)`

添加一个无符号递增 INTEGER 或 BIG INTEGER 主键 ID。

`timestamps()` 和 `nullableTimestamps()`

添加一个 `created_at` 和 `updated_at` 时间戳列

`rememberToken()`

为用户 “remember me” 的令牌添加一个 `remember_token` 类型列 (VARCHAR(100))。

`softDeletes()`

为包含软删除的用户添加一个 `deleted_at` 时间戳。

`morphs(colName)`

对于一个给定的 `+ colName +`，添加一个整数 `colName_id` 和一个字符串 `colName_type` (例如 `morphs('tag')` 会添加整数 `tag_id` 和 string `tag_type`)，用于多态关系。

建立额外的属性

通过一个字段定义的大多数属性（例如，长度可以通过在字段创建方法中设置第二个参数的方式得以实现）。但是在创建列之后，我们还可以通过链接更多方法来设置一些其他属性，例如 email 字段为空（在 MySQL 中），将其放置在 last_name 字段后。

```
Schema::table('users', function (Blueprint $table) {  
    $table->string('email')->nullable()->after('last_name');  
});
```

下面介绍设置字段的其他属性的方法。

nullable()

允许在列中插入 NULL 值。

default('default content')

在没有设置值时指定列中默认内容。

unsigned()

标记整型列为无符号型。

first() (仅用于 MySQL)

按列顺序把列放置到首列。

after(colName) (仅用于 MySQL)

按列顺序把列放到另外一列的后面。

unique()

添加一个 UNIQUE 索引。

primary()

添加一个主键索引。

index()

添加一个基本索引。

需要注意的是，unique()、primary() 和 index() 方法也可以在 Fluent 列创建环境以外使用。

删除表

如果希望删除一张表, 在 Schema 中有一个 `drop` 方法, 该方法只带一个参数 (即该表名), 如下所示。

```
Schema::drop('contacts');
```

修改列

如果想要修改列, 只需要像创建一个新的列那样编写代码, 然后在 `change()` 方法中添加一个调用。



修改列的必要依赖

在修改任何列 (或者在 SQLite 中删除列) 时, 都需要在 *composer.json* 中添加 *doctrine/dbal* 包作为条件, 然后执行 `composer update` 命令使其生效。

所以对于名为 `name` 的字符串, 列长度为 255, 如果想要把它的列长度改为 100, 可以使用如下代码。

```
Schema::table('users', function ($table) {
    $table->string('name', 100)->change();
});
```

如果我们想要调整任何在方法名中没有定义的属性, 操作也是一样的。可以设置字段为空, 代码如下。

```
Schema::table('contacts', function ($table) {
    $table->string('deleted_at')->nullable()->change();
});
```

下面代码可以对列进行重命名。

```
Schema::table('contacts', function ($table)
{
    $table->renameColumn('promoted', 'is_promoted');
});
```

删除列的代码如下。

```
Schema::table('contacts', function ($table)
{
    $table->dropColumn('votes');
});
```



在 SQLite 中一次修改多列

如果想在一次迁移中删除或修改多列，而正好使用的是 SQLite，那么会报错。如果在第 12 章中，笔者会建议使用 SQLite 来测试数据库，而如果使用的是传统的数据库，那么刚才所提到的问题可能会在测试的时候限制操作。

如果不想每次都创建一个新的迁移，那么可以在迁移的 `up()` 方法中调用 `Schema::table()` 来创建多个调用。

```
public function up()
{
    Schema::table('contacts', function (Blueprint $table)
    {
        $table->dropColumn('is_promoted');
    });
    Schema::table('contacts', function (Blueprint $table)
    {
        $table->dropColumn('alternate_email');
    });
}
```

索引和外键

我们已经介绍了怎样创建、修改和删除列，接下来继续学习索引和关联。

添加索引。示例 8-3 展示了怎样为列添加索引。

示例8-3 在迁移中添加列索引

```
// 创建列之后
$table->primary('primary_id'); // 主键；如果使用了 increments(), 则不是必须有的
$table->primary(['first_name', 'last_name']); // 复合键
$table->unique('email'); // 唯一索引
$table->unique('email', 'optional_custom_index_name'); // 唯一索引
$table->index('amount'); // 基本索引
$table->index('amount', 'optional_custom_index_name'); // 基本索引
```

需要注意的是，第一个例子中的主键不是必须的，如果在创建索引的时候使用了，那么 `increments()` 方法数据库会自动添加一个主键索引。

删除索引。示例 8-4 展示了怎样删除索引。

示例8-4 在迁移中删除列索引

```
$table->dropPrimary('contacts_id_primary');
$table->dropUnique('contacts_email_unique');
$table->dropIndex('optional_custom_index_name');
```

```
// 如果上传了一个列名的数组给 dropIndex, 那么它会根据生成规则猜测索引名称
$table->dropIndex(['email', 'amount']);
```

添加和删除外键。为了添加一个外键也就是定义了一个特殊的列, 这里引用了其他表中的一列, Laravel 的语法简单明了。

```
$table->foreign('user_id')->references('id')->on('users');
```

这里为 `user_id` 列添加了一个外索引, 也就是它引用了 `users` 表中的 `id` 这一列。

如果想要声明一个外键限制, 则可以通过 `onDelete()` 和 `onUpdate()` 方法实现, 代码如下。

```
$table->foreign('user_id')
    ->references('id')
    ->on('users')
    ->onDelete('cascade');
```

为了删除一个索引, 可以通过引用它的索引名 (通过连接列名和引用的表名自动生成的) 的方式来实现, 代码如下。

```
$table->dropForeign('contacts_user_id_foreign');
```

或者通过传入一个引用本地表的字段数组的方式来实现, 代码如下。

```
$table->dropForeign(['user_id']);
```

运行迁移

现在已经定义了迁移, 那么要怎么运行呢? Artisan 有专门的命令来实现这个效果, 代码如下。

```
php artisan migrate
```

这行命令会运行所有的“outstanding”迁移。Laravel 会追踪每一个正在运行和没有运行的迁移。每一次运行上面的命令, 它会检查是否已经运行了所有可用的迁移, 如果没有, 它就会运行那些还没有运行的迁移。

在命名空间中还有一些选项。首先可以运行迁移和填充, 代码如下。

```
php artisan migrate --seed
```

也可以运行下面的任何命令。

- `migrate:install` 创建数据库表来监测运行和没有运行的迁移, 它会在你运行迁移时自动运行。

- `migrate:reset` 回滚运行当前安装下的每一个数据库迁移。
- `migrate:refresh` 回滚运行在当前安装下的每一个数据库迁移，然后运行每一个可用的迁移。这和运行 `migrate:reset` 后再运行 `migrate` 是一样的。
- `migrate:rollback` 回滚刚刚运行 `migrate` 的迁移或者添加选项 `--step = 1` 且回滚指定的迁移次数。
- `migrate:status` 显示列出每次迁移的列表，通过 Y 或者 N 来显示它是否在当前环境中运行。



包含 Homestead/Vagrant 的迁移

如果在本地运行迁移而且 `.env` 文件指向 Vagrant 盒子中的一个数据库，那么迁移将会失败。需要通过 `ssh` 来访问 Vagrant 盒子，然后运行迁移。对于填充及其他任何从数据库读取或受影响的 Artisan 命令来说，也是如此。

填充

Laravel 中的填充非常简单。作为常见开发工作流程的一部分，它在以前的 PHP 框架中没有被广泛采用。文件夹 `database/seeds` 该文件夹带有一个 `DatabaseSeeder` 类，该类中有一个可以在调用填充器时使用的 `run()` 方法。

运行填充器主要有两种方法：一种是和迁移一起运行；另一种是分开执行。

和迁移一起运行的话，需要在调用迁移时加入 `--seed`。

```
php artisan migrate --seed
php artisan migrate:refresh --seed
```

单独运行的代码如下。

```
php artisan db:seed
php artisan db:seed --class=VotesTableSeeder
```

这样将会运行每一个 seeder 类的 `run()` 方法中定义的任何内容（或只是传递给 `--class` 的类）。

创建填充器

通过 Artisan 命令 `make:seeder` 来创建一个填充，代码如下。

```
php artisan make:seeder ContactsTableSeeder
```

现在可以在 `database/seeds` 目录下看到 `ContactsTableSeeder` 类。在对它进行编辑之前，

让我们先把它加到 DatabaseSeeder 类中,以便在运行填充的时候能正常执行,代码如下。

```
// database/seeds/DatabaseSeeder.php
...
    public function run ()
    {
        $this->call(ContactsTableSeeder::class);
    }
}
```

接下来编辑填充器,这里最简单的操作是使用 DB facade 手动插入一套记录,代码如下。

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class ContactsTableSeeder extends Seeder
{
    public function run()
    {
        DB::table('contacts')->insert([
            'name' => 'Lupita Smith'
            'email' => 'lupita@gmail.com',
        ]);
    }
}
```

得到一条记录是一个好的开始。但是对于真正的功能性填充来说,可能想要循环一些随机生成器,然后多次运行 insert()。

模型工厂

为了在数据库表中创建假条目,模型工厂定义了一个或多个模式。默认情况下它们是以一个 Eloquent 的类命名的,但是如果不打算使用 Eloquent,也可以在表名后面自行进行命名。下面是对同一个表进行设置两种方式。

```
$factory->define(User::class, function (Faker\Generator $faker){
    return [
        'name' => $faker->name,
    ];
});

$factory->define('users', function (Faker\Generator $faker){
    return [
        'name' => $faker->name,
```

```
    ];  
});
```

理论上来说可以对这些工厂进行任何命名，但是最常用的方式是在 Eloquent 类后命名工厂。

创建一个模型工厂

模型工厂可以在 `database/factories/ModelFactory.php` 中定义。每个工厂包含名称和指明如何在定义类中创建一个新实例的定义。`$factory->define()` 方法把工厂名作为第一个参数，把运行每一代模型的闭包作为第二个参数。

我们可以定义的、最简单的工厂，如下所示。

```
$factory->define(Contact::class, function (Faker\Generator $faker) {  
    return [  
        'name' => 'Lupita Smith',  
        'email' => 'lupita@gmail.com',  
    ];  
});
```

现在可以通过 `factory()` 全局助手在填充和测试里创建一个 `Contact` 实例，代码如下。

```
// 创建一个  
$contact = factory(Contact::class)->create();  
  
// 创建多个  
factory(Contact::class, 20)->create();
```

但是如果通过工厂来创建 20 个联系人 (contact)，那么这 20 个联系人所有的信息都相同，这并不是我们想要的结果。

当利用传入闭包的 `Faker` 实例 (<https://github.com/fzaninotto/Faker>) 时，则可以更加体会到模型工厂的强大；`Faker` 可以轻松地使创建的结构化的假数据变得随机化。前面一个示例现在可以转变为如下形式。

```
$factory->define(Contact::class, function (Faker\Generator $faker) {  
    return [  
        'name' => $faker->name,  
        'email' => $faker->email,  
    ];  
});
```

现在每次通过模型工厂创建一个假的联系人时，它们各自的属性将是唯一值。

使用模型工厂

模型工厂的使用主要包含两个方面:测试(第 12 章中具体介绍)、填充(本节的主要内容)。我们先用模型工厂写一个填充,如示例 8-5 所示。

示例8-5 使用模型工厂

```
factory(Post::class)->create([
    'title' => 'My greatest post ever'
]);

factory(User::class, 20)->create()->each(function ($u) use ($post) {
    $post->comments()->save(factory(Comment::class)->make([
        'user_id' => $u->id
    ]));
});
```

在使用工厂时,可以使用 `factory()` 全局助手,以及传入工厂名(就像刚才那样,传入的名字就是在 Eloquent 类生成实例的名字)。它会返回一个工厂,然后就可以运行工厂里的 `make()` 或者 `create()` 了。

这两个方法都会通过 *modelFactory.php* 里的定义,为当前类生成一个实例。不同之处在于 `make()` 创建了实例但是不会把它保存到数据库中,而 `create()` 会马上保存实例。

在本章后面的内容中,我们会讨论 Eloquent 中的关系,然后再来看第一个例子就更容易理解了。

调用模型工厂时覆盖属性。如果往 `make()` 或 `create()` 方法中传入一个数组,那么可以覆盖指定的键,就像我们在示例 8-5 中为评论设置 `user_id` 以及手动为 `post` 设置标题一样。

在模型工厂中生成多个实例。如果把数字当作第二个参数传入 `factory()` 助手中,那么就相当于指定创建多个实例。不同于返回单个实例,这样将会返回实例的集合。可以把返回结果当作一个数组,也可以把任何一个实例同其他实体联系起来,或者可以在每一个实例上使用其他实体的方法,就像我们在示例 8-5 中通过 `each()` 方法为每一个新创建的用户添加一条评论。

定义和访问多个模型工厂类型

再看看 *modelFactory.php*, 我们定义了一个 `Contact` 工厂。

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
```

```
];
});
```

但是有时需要不止一个工厂来处理同一类对象，如果想添加一些很重要的联系人 (VIP)，该怎么办呢？可以像示例 8-6 一样为模型定义第二个工厂类型。

示例8-6 为同一个模型定义多个工厂类型

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});

$factory->defineAs(Contact::class, 'vip', function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'vip' => true,
    ];
});
```

这样会有很多重复，不过我们可以让任何指定的模型工厂继承其他工厂，然后重写一个或多个属性。接下来通过 `$factory->raw()` 继承前一个工厂，添加“VIP”联系人，如示例 8-7 所示。

示例8-7 继承工厂类型

```
$factory->define(Contact::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
    ];
});

$factory->defineAs(
    Contact::class,
    'vip',
    function (Faker\Generator $faker) use ($factory) {
        $contact = $factory->raw(Contact::class);

        return array_merge($contact, ['vip' => true]);
    });
```

使用指定的类型，代码如下。

```
$vip = factory(Contact::class, 'vip')->create();  
  
$vips = factory(Contact::class, 'vip', 3)->create();
```

查询构造器

现在已经学会了连接迁移和填充表，下面学习怎样使用数据库工具。Laravel 的数据库核心功能是查询构造器，也就是与数据库进行交互的流畅界面。

什么是流畅界面

流畅界面是一种界面，其主要使用方法链来为终端用户提供更简单的 API。流畅界面不会将所有的相关数据都传递到构造函数或方法调用中，它可以逐渐构建调用链并连续调用。来看一下非流畅与流畅的差异。

```
// 非流畅  
$users = DB::select(['table' => 'users', 'where' => ['type' => 'donor']]);  
// 流畅  
$users = DB::table('users')->where('type', 'donor')->get();
```

对于 Laravel 的数据库结构而言，只需要修改一部分配置，就可以通过一个交互界面连接 MySQL、Postgres、SQLite 以及 SQL Server。

如果以前使用过 PHP 框架，为了安全性可能会使用一个允许运行“raw”的 SQL 查询与基本转义的工具。而查询构建器会有很多方便的层次和助手。

DB Facade 的基本使用

在通过流畅的方法链来创建复杂的查询语句之前，先来看一些简单的 DB facade 命令。DB facade 用于查询构造链和简单的原始查询语句，如示例 8-8 所示。

示例8-8 原始SQL和查询构造器使用示例

```
// 基本语句  
DB::statement('drop table users')  
  
// 原始查询和参数绑定  
DB::select('select * from contacts where validated = ?', [true]);  
  
// 选择使用流畅构造器  
$users = DB::table('users')->get();
```

```
// joins 和其他调用
DB::table('users')
    ->join('contacts', function ($join) {
        $join->on('users.id', '=', 'contacts.user_id')
        ->where('contacts.type', 'donor');
    })
    ->get();
```

原始 SQL 语句

像示例 8-1 中那样，可以用 DB facade 以及 `statement()` 方法 `DB::statement('SQL statement here')` 来实现任何原始调用。

但是对于不同的常见操作：`select()`、`insert()`、`update()` 和 `delete()`，这里也有一些特定的方法。它们依然是原始调用，但是有一些不同之处。首先，使用 `update()` 和 `delete()` 方法会返回受影响的行数，但是 `statement()` 不会；其次，对未来的开发人员来说，使用这些方法能让他们更清楚地知道之前使用了什么语句。

原始查询

DB 的特殊方法中最简单的就是 `select()`，不用添加任何参数就可以运行，代码如下。

```
$users = DB::select('select * from users');
```

它将会返回一个 `stdClass` 对象的集合。

Illuminate 集合

在 Laravel 5.3 之前的版本中，DB facade 为返回一行的方法返回一个 `stdClass` 对象（如 `first()`），为返回多行的方法返回一个数组（如 `all()`）。在 Laravel 5.3 中，DB facade 和 Eloquent 一样为任何返回（或可以返回）多列的方法返回一个集合。DB facade 返回一个 `Illuminate\Support\Collection` 实例；Eloquent 返回一个 `Illuminate\Database\Eloquent\Collection` 实例，它继承了 `Illuminate\Support\Collection` 中的一些与 Eloquent 有关的方法。

集合就像有超能力的 PHP 数组，可以运行 `map()`、`filter()`、`reduce` 和 `each()`，使人们更专注于数据。本书将会在第 17 章中更深入地介绍集合。

参数和命名绑定

Laravel 的数据库结构允许使用 PDO 参数绑定，它可以保护查询不受潜在的 SQL 攻击。

向语句中传递一个参数就像用 ? 在语句中替换值一样简单，然后把具体的值作为第二参数加到调用中，代码如下。

```
$usersOfType = DB::select(
    'select * from users where type = ?',
    [$type]
);
```

为了进行更清晰的说明，也可以为这些参数取名字，代码如下。

```
$usersOfType = DB::select(
    'select * from users where type = :type',
    ['type' => $userType]
);
```

原始插入

这里可以看出原始的命令看起来都很类似。原始插入代码如下。

```
DB::insert(
    'insert into contacts (name, email) values (?, ?)',
    ['sally', 'sally@me.com']
);
```

原始更新

原始更新代码如下。

```
$countUpdated = DB::update(
    'update contacts set status = ? where id = ?',
    ['donor', $id]
);
```

原始删除

原始删除代码如下。

```
$countDeleted = DB::delete(
    'delete from contacts where archived = ?',
    [true]
);
```

查询构造器链

到现在为止，我们还没有真正使用查询构造器，只是使用了 DB facade 中简单的方法调用。下面实际构造一些查询。

查询构造器能把方法都链接起来,然后构造一个查询。在连接的最后,可以使用一些方法,比如 `get()`, 来触发执行之前构造的查询。

让我们看一个简单的示例,代码如下。

```
$usersOfType = DB::table('users')
    ->where('type', $type)
    ->get();
```

这里构造了查询 `users` 表、`$type` 类型,然后执行这个查询得到结果。

来看看查询构造器允许链接使用的方法。这些方法包括限制方法、修改方法和结束 / 返回方法。

限制方法

以下这些方法可以直接执行查询并限制可能返回的数据中较小子集。

`select()`

允许指定想选择的列。

```
$emails = DB::table('contacts')
    ->select('email', 'email2 as second_email')
    ->get();
// 或
$emails = DB::table('contacts')
    ->select('email')
    ->addSelect('email2 as second_email')
    ->get();
```

`where()`

允许限制使用 `WHERE` 子句返回的范围。默认情况下 `where()` 方法的特点是它需要三个参数——列、比较运算符以及值。

```
$newContacts = DB::table('contact')
    ->where('created_at', '>', Carbon::now()->subDay())
    ->get();
```

然而,如果比较运算符 `=` 也较为常用,那么就可以省略第二个操作符:

```
$vipContacts = DB::table('contacts')->where('vip',true)->get();
```

如果想要连接 `where()` 子句,可以把他们连接起来或者传入一个数组集中的数组。

```
$newVips = DB::table('contacts')
```

```

->where('vip', true)
->where('created_at', '>', Carbon::now())
->subDay());
// 或
$newVips = DB::table('contacts')->where([
    ['vip', true],
    ['created_at', '>', Carbon::now()->subDay()],
]);

```

orWhere()

创建简单的 OR WHERE 子句，代码如下。

```

$priorityContacts = DB::table('contacts')
->where('vip', true)
->orWhere('created_at', '>', Carbon::now()->subDay())
->get();

```

如果想要创建更加复杂、多条件的 OR WHERE 子句,则需要传入一个 orWhere() 闭包。

```

$contacts = DB::table('contacts')
->where('vip', true)
->orWhere(function ($query) {
    $query->where('created_at', '>', Carbon::now()->subDay())
    ->where('trial', false);
})
->get();

```



多条 where 语句和 orWhere 调用的区别

如果同时使用 orWhere() 调用和多条 where() 语句调用，那么需要非常小心以保证查询结果满足预期的效果。这不是 Laravel 的错误，就像下面的这个查询就可能没有实现预期的效果。

```

$scanEdit = DB::table('users')
->where('admin', true)
->orWhere('plan', 'premium')
->where('is_plan_owner', true)
->get();
SELECT * FROM users
WHERE admin = 1
OR plan = 'premium'
AND is_plan_owner = 1;

```

如果想要 SQL 语句达到“if 这样（或者 if A 和 B）”的效果，也就是前面的例子中的预期效果，则需要向 orWhere() 调用中传入一个闭包，代码如下。

```

$scanEdit = DB::table('users')

```

```

->where('admin', true)
->orWhere(function ($query) {
    $query->where('plan', 'premium')
        ->where('is_plan_owner', true);
})
->get();
SELECT * FROM users
WHERE admin = 1
OR (plan = 'premium' AND is_plan_owner = 1);

```

`whereBetween(colName, [low, high])`

允许指定查询范围来返回列位于两个值之间（包括两个值）所在的行，代码如下。

```

$mediumDrinks = DB::table('drinks')
->whereBetween('size', [6, 12])
->get();

```

`whereNotBetween()` 遵循一样的原理，但是它返回列大于或小于两个值的行。

`whereIn(colName, [1,2,3])`

允许指定查询范围来返回列满足指定选项列表的行，代码如下。`$closeBy = DB::table('contacts')->whereIn('state', [FL, GA, AL])->get()`。

```

$closeBy = DB::table('contacts')->whereIn('state', ['FL', 'GA', 'AL'])->get()

```

`whereNotIn()` 则返回列不满足指定选项列表的行。

`whereNull(colName)` 和 `whereNotNull(colName)`

分别选择指定列是 NULL 或者 NOT NULL 的行。

`whereRaw()`

允许传入一行非转义字符串会被加到 WHERE 子句后面：`$goofs = DB::table('contacts')->whereRaw('id = 12345')->get()`。



警惕 SQL 注入

任何传入 `whereRaw()` 的 SQL 查询都是不能转义的。注意尽量少使用这个方法，因为它将会是应用程序中最可能被 SQL 注入攻击的位置。

whereExists()

当传入一个指定的次级查询时，如果该查询至少返回一行数据，那么该行可以被 whereExists 方法选中。想象一下如果只希望得到超过一条评论的用户信息，则代码如下。

```
$commenters = DB::table('users')
    ->whereExists(function ($query) {
        $query->select('id')
            ->from('comments')
            ->whereRaw('comments.user_id = users.id');
    })
    ->get();
```

distinct()

仅选择不同的行。通常它与 select() 成对出现，因为如果想使用一个主键，就要保证不会有重复的行：`$lastNames = DB::table('contacts')->select('last_name')->distinct()->get()`。

修改方法

这些方法改变了查询的输出结果，而不仅对结果有所限制。

orderBy(colName, direction)

把结果排序。第二个参数可以是 asc（默认）或 desc，代码如下。

```
$contacts = DB::table('contacts')
    ->orderBy('last_name', 'asc')
    ->get();
```

groupBy() 和 having() 或者 havingRaw()

根据列组合输出结果。having() 和 havingRaw() 是可选项，它们可以通过组合的属性过滤输出结果。例如只查询有 30 个人以上的城市，代码如下。

```
$populousCities = DB::table('contacts')
    ->groupBy('city')
    ->havingRaw('count(contact_id) > 30')
    ->get();
```

skip() 和 take()

大多数时候用于分页，这样便可以用它们来定义返回的行数以及在返回之前跳过的

行数——就像分页系统中的页数和界面大小。

```
$page4 = DB::table('contacts')->skip(30)->take(10)->get();
```

`latest(colName)` 与 `oldest(colName)`

将传入的列按降序 (最新的) 或升序 (最旧的) 排序, 如果传入的列没有名字会默认为 `created_at`。

`inRandomOrder()`

将输出结果随机排序。

结束 / 返回方法

这两个方法会结束查询链, 以及触发 SQL 的执行。

`get()`

得到构造的查询的所有结果, 代码如下。

```
$contacts = DB::table('contacts')->get();  
$vipContacts = DB::table('contacts')->where('vip', true)->get();
```

`first()` 和 `firstOrFail()`

只得到第一个结果——跟 `get()` 类似, 但是多了一个 `LIMIT 1` 条件, 代码如下。

```
$newestContact = DB::table('contacts')  
    ->orderBy('created_at', 'desc')  
    ->first();
```

`first()`

如果没有输出结果不会有任何提示, 但是 `firstOrFail()` 会抛出异常。

如果为每一个方法传入一个列名数组, 这两个方法会为指定列返回数据而不是所有的列。

`find(id)` 和 `findOrFail(id)`

跟 `first()` 类似, 但是需要指定一个 ID, 也就是主键对应的 ID 值。如果哪一行不存在 ID, `find()` 会失败, 而 `findOrFail()` 则会抛出异常。

```
$contactFive = DB::table('contacts')->find(5);
```

value()

仅从第一行的单个字段中提取值。类似于 `first()`，但是如果只想要一个列，则需要使用以下代码。

```
$newestContactEmail = DB::table('contacts')
    ->orderBy('created_at', 'desc')
    ->value('email');
```

count()

返回一个整型数，其代表所有符合条件结果，代码如下。

```
$countVips = DB::table('contacts')
    ->where('vip', true)
    ->count();
```

min() 和 max()

返回指定列的最小或最大值，代码如下。

```
$highestCost = DB::table('orders')->max('amount');
```

sum() 和 avg()

返回指定列中所有值的和或平均值，代码如下。

```
$averageCost = DB::table('orders')
    ->where('status', 'completed')
    ->avg('amount');
```

通过 DB::raw 在查询构造器方法中编写原始查询

我们已经看到了原始语句中的一些自定义方法——比如 `select()` 有对应的 `selectRaw()` 方法。它可以向查询构造器中传入一串字符串并把它放到 `WHERE` 子句后面；也可以向查询构造器的大多数方法传入 `DB::raw()` 调用的结果来得到同样的输出结果。

```
$contacts = DB::table('contacts')
    ->select(DB::raw('*', (score * 100) AS integer_score'))
    ->get();
```

join 连接

有时候定义 `join` 连接很难，但是它可以通过框架使定义变得更加简单。让我们来看一个示例，如下所示。

```

$users = DB::table('users')
->join('contacts', 'users.id', '=', 'contacts.user_id')
->select('users.*', 'contacts.name', 'contacts.status')
->get();

```

`join()` 方法会创建一个内连接。也可以把多个 `join` 连接一个接一个地连接起来或者使用 `leftJoin()` 来得到一个左连接。

最后可以通过向 `join()` 传入一个闭包来创建更加复杂的 `join` 连接，代码如下。

```

DB::table('users')
->join('contacts', function ($join) {
    $join
        ->on('users.id', '=', 'contacts.user_id')
        ->orOn('users.id', '=', 'contacts.proxy_user_id');
})
->get();

```

union 连接

可以先创建两个查询，然后使用 `union()` 或者 `unionAll()` 方法来连接这两个查询。

```

$first = DB::table('contacts')
->whereNull('first_name');

$contacts = DB::table('contacts')
->whereNull('last_name')
->union($first)
->get();

```

插入

`insert()` 方法非常简单。传入一个数组来插入一行或者传入一个数组的数组来插入多行。或者也可以使用 `insertGetId()` 方法来得到一个自增的主键 ID 作为返回值。

```

$id = DB::table('contacts')->insertGetId([
    'name' => 'Abe Thomas',
    'email' => 'athomas1987@gmail.com',
]);
DB::table('contacts')->insert([
    ['name' => 'Tamika Johnson', 'email' => 'tamikaj@gmail.com'],
    ['name' => 'Jim Patterson', 'email' => 'james.patterson@hotmail.com'],
]);

```

更新

Update 也很简单。创建更新查询不需要使用 `get()` 或者 `first()`，它只需要使用

`update()` 方法，然后传入一个参数的数组，代码如下。

```
DB::table('contacts')
    ->where('points', '>', 100)
    ->update(['status' => 'vip']);
```

也可以通过 `increment()` 和 `decrement()` 方法来快速递增或递减列。第一个参数是列名，第二个参数是可选的对应递增 / 递减的数量，代码如下。

```
DB::table('contacts')->increment('tokens', 5);
DB::table('contacts')->decrement('tokens');
```

删除

删除也很简单，构造查询，然后用 `delete()` 来结束，代码如下。

```
DB::table('users')
    ->where('last_login', '<', Carbon::now()->subYear())
    ->delete();
```

还可以使这两个表删除每一行并重置自动递增的 ID，代码如下。

```
DB::table('contacts')->truncate();
```

JSON 操作

如果数据库中有 JSON 列，那么可以根据 JSON 的结构，使用 `arrow` 语法遍历子节点来更新或选择行，代码如下。

```
// 选择 "options" 列中属性为 "isAdmin" 的记录
// JSON 列设为 true

DB::table('users')->where('options->isAdmin', true)->get();
// 更新所有记录设置，将 "options"JSON 列中属性为 "verified" 的记录设置为 true
DB::table('users')->update(['options->isVerified', true]);
```

53 这是 Laravel 5.3 中的新特性。

事务

数据库的事务是用来批处理一系列数据库查询，而且允许进行回滚撤销全部的查询。事务通常用来保证完整执行或者不执行（而不是部分执行）一系列相关的查询——如果其中一个失败了，ORM 会回滚所有的查询。

Laravel 查询构造器的事务特性表现在：不论在任何时候，如果事务闭包中抛出了异常，那么事务中的所有查询都会回滚。如果事务闭包成功结束，那么所有的查询都会被提交

并且不会回滚。

下面来看一下示例 8-9。

示例8-9 一个简单的数据库事务

```
DB::transaction(function () use ($userId, $numVotes)
{
    // 可能失败的 DB 查询
    DB::table('users')
        ->where('id', $userId)
        ->update(['votes' => $numVotes]);

    // 当上面的查询失败时，缓存不想执行的查询
    DB::table('votes')
        ->where('user_id', $userId)
        ->delete();
});
```

在上面的示例中总结了 `votes` 表中的投票数。我们想要将该数字缓存在 `users` 表上，然后把这些投票从 `votes` 表中删除。但是很明显，这里不希望在成功更新 `users` 表之前就删除这些投票，而且如果这些投票删除失败了，也不希望在 `users` 表里保留更新后的投票数。

如果任何一个查询出了问题，那么其他的查询也不会生效——这就是数据库事务的奇妙之处。

也可以手动开启或终止事务——而且该操作会同时在查询构造器和 Eloquent 查询中生效。这里用 `DB::beginTransaction()` 来开启；`DB::commit()` 来终止；`DB::rollBack()` 来回滚。

Eloquent 入门

Eloquent 是一个 ActiveRecord ORM，也就是说它是一个数据库抽象层，它提供了一个与多个数据库类型进行交互的界面。“ActiveRecord”表示一个 Eloquent 类不仅与表格进行整体交互（例如 `User::all()`），还可以与表里的单独一行进行交互（例如 `$sharon = new User`）。此外，每一个实例都可以自己进行持久化管理，可以调用 `$sharon->save` 或 `$sharon->delete`。

Eloquent 主要关注简捷性。和其他很多框架一样，它依赖于“公约配置”，能以最少的代码建立强大的模型。

例如可以在示例 8-10 中定义模型里实现示例 8-11 中的所有方法。

示例8-10 最简单的Eloquent模型

```
<?php
```

```
use Illuminate\Database\Eloquent\Model;
```

```
class Contact extends Model {}
```

示例8-11 最简单的Eloquent模型中实现的操作

```
public function save(Request $request)
```

```
{  
    // 根据用户输入创建和保存一个新的联系人  
    $contact = new Contact();  
    $contact->first_name = $request->input('first_name');  
    $contact->last_name = $request->input('last_name');  
    $contact->email = $request->input('email');  
    $contact->save();  
  
    return redirect('contacts');  
}
```

```
public function show($contactId)
```

```
{  
  
    // 根据网址片段返回联系人的 JSON 表示形式  
    // 如果联系人不存在，则抛出异常  
    return Contact::findOrFail($contactId);  
}
```

```
public function vips()
```

```
{  
    // 在基本的 Eloquent 中可能会出现更复杂的例子  
    // 类为每一个 VIP 条目添加一个 "formalName" 属性  
    return Contact::where('vip', true)->get()->map(function($contact) {  
        $contact->formalName = "The exalted {$contact->first_name} of the  
            {$contact->last_name}s";  
  
        return $contact;  
    });  
}
```

如何实现公约？Eloquent 假设表名 `Contact` 变成了 `contacts`，而通过这种方式可以得到一个完整的、功能强大的 Eloquent 模型。

下面来看看怎样使用 Eloquent 模型。

新建和定义 Eloquent 模型

首先使用 Artisan 命令创建一个模型，代码如下。

```
php artisan make:model Contact
```

下面是我们将在 `app/Contact.php` 看到的内容。

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    //
}
```



为模型创建一个迁移

如果想在新建模型时自动创建一个迁移，则可以在命令中加上 `-m` 或者 `--migration` 标签，如下所示。

```
php artisan make:model Contact --migration
```

表名

表名的默认行为是 Laravel 的 “snake case”，并且它会将类名复数化，所以 `SecondaryContact` 会访问一个名为 `secondary_contacts` 的表。如果想要自定义表名，则可以在模型中显示设置 `$table` 属性。

```
protected $table = 'contacts_secondary';
```

主键

Laravel 默认认为，每一个表都有一个整型的自增主键并且命名为 `id`。

如果想要修改主键的名字，则需要修改 `$primaryKey` 属性，代码如下。

```
protected $primaryKey = 'contact_id';
```

如果不想让它自增，则可以使用以下代码。

```
public $incrementing = false;
```


时间戳

Eloquent 希望每张表都包含 `created_at` 和 `updated_at` 时间戳列。如果不想在表中包含这两列，可以关闭 `$timestamps` 这个功能，代码如下。

```
public $timestamps = false;
```

也可以把 `$dateFormat` 类的属性设置为一个自定义的字符串，修改 Eloquent 用来保存时间戳的格式。这个字符串会按照 PHP 的 `date()` 语法来解析，因此下面这个示例会按 UNIX 版本的秒来存储日期。

```
protected $dateFormat = 'U';
```

通过 Eloquent 获取数据

在大多数情况下，在从 Eloquent 中获取数据时，会在 Eloquent 模型中使用静态调用。

我们先获取所有数据，代码如下。

```
$allContacts = Contact::all();
```

让我们再过滤掉一些数据，代码如下。

```
$vipContacts = Contact::where('vip', true)->get();
```

可以看到 Eloquent 的 facade 能够进行链式约束，如下所示。

```
$newestContacts = Contact::orderBy('created_at', 'desc')
    ->take(10)
    ->get();
```

结果表明，一旦改变了最初的 facade 的名字，很多人就会使用 Laravel 的查询构造器。其实还可以进行更多操作，但是需要注意的是，任何使用数据库 facade 的查询构造器可以进行的操作，都可以在 Eloquent 对象上进行处理。

获取单个数据

我们在本章前面的内容中提到可以用 `first()` 来返回查询的第一条记录或者用 `find()` 得到指定 ID 的记录。对这两种方法来说，如果在方法名后添加了“orFail”，那么在没有任何记录匹配的时候系统会抛出异常。所以在 URL 片段中查询一个实体时，`findOrFail()` 是一个非常常用的工具（可以匹配的实体不存在时抛出异常），如示例 8-12 所示。

示例8-12 在控制器方法中使用Eloquent的OrFail()方法

```
// Contact 控制器
```

```
public function show($contactId)
{
    return view('contacts.show')
        ->with('contact', Contact::findOrFail($contactId));
}
```

任何返回单一记录的方法 (如 `first()`、`firstOrFail()`、`find()` 或者 `findOrFail()`)，都会返回一个 Eloquent 类的实例。所以 `Contact::first()` 将返回一个 `Contact` 类的实例 (与第一行的数据填充)。



异常

就像示例 8-12 一样，不需要在控制器中抓取 Eloquent 模型的 not found 异常 (`Illuminate\Database\Eloquent\ModelNotFoundException`)，Laravel 的路线系统会自动抓取然后抛出 404 异常。

当然也可以根据自己的喜好抓取特定的异常，然后进行解决。

获取多个数据

在 Eloquent 中，`get()` 就跟普通的查询构造器调用一样创建一个查询，然后调用 `get()` 方法获取结果，代码如下。

```
$vipContacts = Contact::where('vip', true)->get();
```

不过 Eloquent 也有自己独有的方法 `all()`，当人们从表中获取没有过滤的数据时，经常使用这个方法。

```
$contacts = Contact::all();
```



多使用 `get()` 而不是 `all()`

任何使用 `all()` 的时候，都可以使用 `get()`。`Contact::get()` 与 `Contact::all()` 的返回结果相同。但是如果修改了查询语句，比如添加 `where()` 条件，就不能使用 `all()` 了，但是还可以继续使用 `get()`。

所以尽管 `all()` 很常用，仍然建议大家尽量选择使用 `get()`。

Laravel 5.3 之前版本中 Eloquent 的 `get()` 方法，不同之处在于它会返回一个数组而不是集合。在 Laravel 5.3 和之后的版本中，`get()` 都会返回一个集合。

用 chunk() 进行响应分块

如果处理过大量 (上千或者更多的) 记录, 那么应该遇到过内存或者锁定的问题。Laravel 可以把请求分成很多、很小的部分 (分块), 然后进行批量处理, 这样能让内存分别加载每一小部分的请求。示例 8-13 介绍了 chunk() 的使用方法。

示例8-13 对Eloquent查询进行分块来限制内存使用

```
Contact::chunk(100, function ($contacts) {  
    foreach ($contacts as $contact) {  
        // 对 $contact 进行操作  
    }  
});
```

聚合

在查询构造器上, 可用的聚合在 Eloquent 的查询中同样可以使用, 示例代码如下。

```
$countVips = Contact::where('vip', true)->count();  
$sumVotes = Contact::sum('votes');  
$averageSkill = User::avg('skill_level');
```

Eloquent 的插入和更新

插入和更新是 Eloquent 与常用的查询构造器语法不同的地方之一。

插入

Eloquent 中主要有两种方法来插入一条新的记录。

首先可以新建一个 Eloquent 类的实例, 手动设置属性, 然后在实例中调用 save() 方法, 如示例 8-14 所示。

示例8-14 通过创建一个新的实例插入Eloquent记录

```
$contact = new Contact;  
$contact->name = 'Ken Hirata';  
$contact->email = 'ken@hirata.com';  
$contact->save();
```

// 或

```
$contact = new Contact([  
    'name' => 'Ken Hirata',  
    'email' => 'ken@hirata.com'  
]);  
$contact->save();
```

在调用了 `save()` 方法后, `Contact` 实例代表了所有的联系人——除那些从来没有保存到数据库的数据外。这代表它没有 `id`, 如果应用退出了, 那么它不会持久地保存, 也没有 `created_at` 和 `updated_at` 值的集合。

也可以通过向 `Model::create()` 传入一个数组得到相同的输出, 如示例 8-15 所示。

示例8-15 向`create()`传入一个数组插入Eloquent记录

```
$contact = Contact::create([
    'name' => 'Keahi Hale',
    'email' => 'halek481@yahoo.com'
]);
```

同时需要注意的是, 当传入一个数组(向 `Model()`, `Model::create()` 或 `Model::update()`)时, 每一个通过 `Model::create()` 设置的属性都需要满足“批量赋值”的要求。

如果使用的是 `Model::create()`, 那么可以不用 `save()` 实例——`create()` 方法会自动进行保存。

更新

更新和插入操作非常类似。可以得到一个指定的实例, 改变它的属性然后保存; 或者可以单独进行调用, 然后传入一个修改了属性的数组。例 8-16 中使用的就是第一种方法。

例8-16通过更新实例的方式更新Eloquent记录并保存

```
$contact = Contact::find(1);
$contact->email = 'natalie@parkfamily.com';
$contact->save();
```

因为这条记录已经存在了, 它已经有 `created_at` 时间戳和 `id`, 这两个属性不会改变, 但是 `update_at` 字段会根据当前日期和时间进行改变。示例 8-17 对应的就是第二种方法。

示例8-17 向`update()`传入数组以更新一条或多条Eloquent记录

```
Contact::where('created_at', '<', Carbon::now()->subYear())
    ->update(['longevity' => 'ancient']);
// 或
$contact = Contact::find(1);
$contact->update(['longevity' => 'ancient']);
```

使用这种方法需要保证数组中的每一个键是列名, 每一个值是列中对应的值。

批量赋值

在前面的很多例子中已经看过了怎样往 Eloquent 类的方法中传入数组。但是除非在方法中定义了“可填充”字段，否则这些传输都不会生效。

这样做的目的是防止（恶意的）用户不小心在不想修改的字段上设置新的值。示例 8-18 展示了常见的一种使用情况。

示例8-18 使用整个请求输入更新Eloquent模型

```
// Contact 控制器
public function update(Contact $contact, Request $request)
{
    $contact->update($request->all());
}
```

如果对 Illuminate Request 对象不熟悉，那么通过示例 8-18 可以看到它会获取用户的每一次输入，然后把输入传入 update() 方法。all() 方法包括 URL 参数和表单输入，所以恶意用户就可以轻松地在里面添加一些内容，比如 id 和 owner_id，这些都是我们不想修改的。

不过除非在模型中定义了可填充字段，否则这些操作都不会生效。我们可以把可填充字段加入白名单，也可以把“防护”字段加入黑名单，然后决定字段能不能通过“批量赋值”来编辑。也就是把包含需要修改值的数组传到 create() 或 update() 方法中。而对于不能填充的属性，可以通过直接赋值的方式来修改（如 \$contact->password = 'abc'；）。示例 8-19 展示了这两种方法。

示例8-19 通过Eloquent的“可填充”或“防护”属性定义批量赋值的字段

```
class Contact
{
    protected $fillable = ['name', 'email'];

    // 或
    protected $guarded = ['id', 'created_at', 'updated_at', 'owner_id'];
}
```



在 Eloquent 批量赋值中使用 Request::only()

在示例 8-18 中，我们需要 Eloquent 的批量赋值保护，因为在请求对象中使用了 all() 方法把所有的用户输入传到 Eloquent 对象中。

Eloquent 的批量赋值保护机制是非常重要的工具，但是这里也有一个很有帮助的小技巧，那就是不接收用户的任何输入。

Request 类的 `only()` 方法能提取出用户输入的一些关键字，所以可以使用以下代码。

```
Contact::create($request->only('name', 'email'));
```

firstOrCreate() 和 firstOrCreateNew()

有的时候，我们希望应用程序“根据属性返回一个实例，如果实例不存在就新建一个”，这就是使用 `firstOrCreate()` 方法的主要目的。

`firstOrCreate()` 和 `firstOrCreateNew()` 方法的第一个参数是由键和值组成的数组，如下所示。

```
$contact = Contact::firstOrCreate(['email' => 'luis.ramos@myacme.com']);
```

它们都是用于查询和获取第一个与参数匹配的记录，如果都没有匹配的记录，那么这两个方法会根据它们的属性创建一个实例，`firstOrCreate()` 会把实例持久化到数据库，然后返回；而 `firstOrCreateNew()` 会返回但不会保存实例。

如果把值的数组作为第二参数，它们将会被添加到创建好的条目中（如果已经创建），但是不会用来查询条目是否存在。

Eloquent 中的删除

Eloquent 中的删除和更新操作类似，但是对于软删除来说，可以把删除的项目归档以供以后检查甚至恢复。

普通删除

删除一个实例，最简单的方法就是在实例中调用 `delete()` 方法，如下所示。

```
$contact = Contact::find(5);  
$contact->delete();
```

如果只知道 ID，就不能通过查找 ID 的方式来删除实例，这时可以把 ID 或者多个 ID 的数组传入模型的 `destroy()` 方法，直接删除。

```
Contact::destroy(1);  
// 或  
Contact::destroy([1, 5, 7]);
```

最后可以删除查询的所有结果，如下所示。

```
Contact::where('updated_at', '<', Carbon::now()->subYear()->delete();
```

软删除

软删除会记录删除的行，但是不会真的从数据库中删除对应的记录。这样以后可以审阅

它们,而不是在历史信息中看见“没有记录已删除”而束手无策。软删除也允许用户(或管理员)恢复部分或全部数据。

使用软删除进行编码的难点在于,写的每个查询都需要排除软删除的数据。如果使用了 Eloquent 的软删除,那么查询会默认忽略软删除,除非显示声明包含这些数据。

在功能上, Eloquent 的软删除需要在表中添加 `deleted_at` 列。只要在 Eloquent 模型中开启了软删除功能,那么进行任何查询都会默认忽略被软删除的数据,除非我们希望把它们包含进来。

什么时候用软删除

软删除只是 Eloquent 的一个特性,并不代表需要一直使用它。Laravel 社区里很多人都会在他们的每一个项目里使用软删除,这是在滥用软删除。就像只是把数据库当作一个类似于 Sequel Pro 的工具,这样就会忘记检查 `deleted_at` 列。而如果不对软删除以前的记录进行整理,那么数据库就会变得越来越大。

所以建议大家不要默认使用软删除,而是在需要的时候使用它们。即便如此,使用它们的时候也尽量先整理之前旧的记录。软删除是非常强大的工具,但是当不用它的时候,它也是毫无用处的。

开启软删除。为了开启 Eloquent 的软删除,需要做三件事情:在迁移中添加 `deleted_at` 列;在模型中导入 `SoftDeletes` 特征;最后把 `deleted_at` 列添加到 `$dates` 属性中。在查询构造器中有一个 `softDeletes()` 方法可以用来在表中添加 `deleted_at` 列,如示例 8-20 所示。在示例 8-21 中展示了怎样开启 Eloquent 模型中的软删除。

示例8-20 在迁移中添加软删除列

```
Schema::table('contacts', function (Blueprint $table){
    $table->softDeletes();
});
```

示例8-21 在Eloquent模型中开启软删除

```
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Contact extends Model
{

    use SoftDeletes; // use the trait
```

```
protected $dates = ['deleted_at']; // mark this column as a date
}
```

一旦做好了上面的这些设置，每次调用 `delete()` 和 `destroy()` 方法就会把当前行上 `deleted_at` 列的值设置为当前日期和时间，而不是直接删除该行，并且将来所有的查询将排除该行。

基于软删除的查询。所以要怎样得到软删除的条目呢？

首先，可以在查询中添加软删除条目，代码如下。

```
$allHistoricContacts = Contact::withTrashed()->get();
```

然后可以使用 `trashed()` 方法来查看指定的实例是否已经被软删除，代码如下。

```
if ($contact->trashed()) {
    // 操作
}
```

最后也可以只获取软删除条目，代码如下。

```
$deletedContacts = Contact::onlyTrashed()->get();
```

从软删除中恢复实体。如果想要恢复已经被软删除的条目，则可以在实例或查询中执行 `restore()` 方法，代码如下。

```
$contact->restore();

// 或

Contact::onlyTrashed()->where('vip', true)->restore();
```

强制删除软删除实体。可以在实体或查询中调用 `forceDelete()` 方法来删除一个软删除的实体，代码如下。

```
$contact->forceDelete();

// 或

Contact::onlyTrashed()->forceDelete();
```

作用域

我们已经介绍了“过滤的”查询，这些查询不是每次都返回表中所有记录。但是在本章中，我们写过的这些查询都是用查询构造器手动编写的。

Eloquent 的本地和全局作用域允许预先构造“带作用域的（过滤的）”查询，可以在每次查询模型“全局”时或每次使用特定方法链（“本地”）查询模型时使用。

本地作用域

本地作用域是最容易理解的，让我们来看一个示例，代码如下。

```
$activeVips = Contact::where('vip', true)->where('trial', false)->get();
```

如果一遍一遍地编写这种查询方法的组合，就太烦琐了，而且在应用程序中如何将某个人定义为“活跃 VIP”是亟需考虑的问题。要如何解决这个问题呢？

```
$activeVips = Contact::activeVips()->get();
```

可以采用本地作用域，它在 Contact 类中很容易定义。

```
class Contact
{
    public function scopeActiveVips($query)
    {
        return $query->where('vip', true)->where('trial', false);
    }
}
```

为了定义本地作用域，我们向以“scope”开头的 Eloquent 类添加一个方法，然后把标题版本的作用域名称包含进去。这个方法传递一个查询构造器，也会返回一个查询构造器，但是可以在返回之前修改这个查询，这一点极为重要。

也可以定义接收参数，代码如下。

```
class Contact
{
    public function scopeStatus($query, $status)
    {
        return $query->where('status', $status);
    }
}
```

使用它们的方法也是一样的，需要向作用域中传入参数。

```
$friends = Contact::status('friend')->get();
```

全局作用域

还记得之前讨论的关于软删除的问题吗？如果在一个范围内，每个查询模型都忽略软删除项目，此时才适用于软删除。这就是全局作用域，我们可以自定义全局作用域，这样它们将作用于模型中的每一次查询。

定义全局作用域有两种方法：使用闭包或者整个类。无论哪种方法，都需要在模型的 `boot()` 方法中注册这个定义的作用域。先看看闭包的方法，如示例 8-22 所示。

示例8-22 使用闭包添加全局作用域

```
...
class Contact extends Model
{
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('active', function (Builder $builder) {
            $builder->where('active', true);
        });
    }
}
```

在示例 8-22 中，我们添加了一个名为 `active` 的全局作用域，这个模型上的所有查询只会作用于 `active` 为 `true` 的行。

下面尝试另一种方法，创建一个 `Illuminate\Database\Eloquent\Scope` 类，该类中的 `apply()` 方法分别包含了查询构造器和模型的实例，如示例 8-23 所示。

示例8-23 创建全局作用域类

```
<?php

namespace App\Scopes;
use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class ActiveScope implements Scope
{
    public function apply(Builder $builder, Model $model)
    {
        return $builder->where('active', true);
    }
}
```

要将作用域应用到模型中，需要再次覆盖父类的 `boot()` 方法并使用 `static` 调用模型中的 `addGlobalScope()` 方法，如示例 8-24 所示。

示例8-24 应用一个基于类的全局作用域

```
<?php
```

```

use App\Scopes\ActiveScope;
use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new ActiveScope);
    }
}

```



没有命名空间的 Contact 类

在前面的一些例子中都使用了 `Contact` 类,但是没有命名空间。这是不正常的,在本书中,笔者只是为了节省空间省略了这个部分。一般即便是最高级的模型都应该包含如 `App\Contact` 这样的命名空间。

删除全局作用域。有三种方法可以删除全局作用域,它们都使用了 `withoutGlobalScope()` 或 `withoutGlobalScopes()` 方法。如果希望删除一个基于闭包的作用域,那么需要将 `addGlobalScope()` 注册方法的第一个参数作为键,代码如下。

```
$allContacts = Contact::withoutGlobalScope('active')->get();
```

如果只是想删除一个基于类的全局作用域,那么可以把类名传给 `withoutGlobalScope()` 或 `withoutGlobalScopes()`,代码如下。

```
Contact::withoutGlobalScope(ActiveScope::class)->get();
```

```
Contact::withoutGlobalScopes([ActiveScope::class, VipScope::class])->get();
```

或者可以关闭查询的所有全局作用域,代码如下。

```
Contact::withoutGlobalScopes()->get();
```

自定义与访问器、修改器和属性转换器的字段交互

现在已经知道了怎样用 Eloquent 从数据库中获取记录,让我们来继续探讨怎样装饰和操作 Eloquent 模型中的单独的属性。

访问器、修改器和属性转换器都允许自定义 Eloquent 实例单独属性的输入和输出方式。如果不使用这些工具,那么 Eloquent 实例中的每一个属性都会被当做一个字符串而且不能获取任何在数据库中不存在的属性。

访问器

在模型实例中读取数据时，访问器能在模型中自定义属性。例如改变特定列的输出方式，或者创建一个在数据库表中不存在的属性。

在模型中通过 `get{PascalCasedPropertyName}Attribute` 结构，就可以定义一个访问器了。所以如果属性名为 `first_name`，那么访问器方法就应该被命名为 `getFirstNameAttribute`。

下面来尝试一下，首先装饰一个已经存在的列，如示例 8-25 所示。

示例8-25 用Eloquent访问器装饰一个已存在的列

```
// 模型定义
class Contact extends Model
{
    public function getNameAttribute($value)
    {
        return $value ?: '(No name provided)';
    }
}
```

```
// 使用访问器
$name = $contact->name;
```

也可以用访问器来定义数据库中不存在的属性，如示例 8-26 所示。

示例8-26 使用Eloquent访问器为不存在的backing列定义属性

```
// 模型定义
class Contact extends Model
{
    public function getFullNameAttribute()
    {
        return $this->first_name . ' ' . $this->last_name; }
}
```

```
// 使用访问器
$fullName = $contact->full_name;
```

修改器

修改器的工作原理和访问器类似，二者唯一的区别在于修改器用来设置数据而非获取数据。和访问器一样，也可以用它修改过程（即向已存在的列写入数据的过程），也可以设置数据库中不存在的列。

通过在模型中按如下结构编写方法，我们便可以轻松定义一个修改器——

set{PascalCasedPropertyName}Attribute。所以如果属性名是 first_name，那么修改器方法就应该被命名为 setFirstNameAttribute。

让我们来尝试一下。首先为已存在列的更新操作添加一个约束，如示例 8-27 所示。

示例8-27 通过Eloquent修改器对属性值的装饰设置

```
// 定义修改器
class Order extends Model
{
    public function setAmountAttribute($value)
    {
        $this->attributes['amount'] = $value > 0 ? $value : 0;
    }
}

// 使用修改器
$order->amount = '15';
```

在示例 8-27 中，修改器希望设置模型中数据的方式是把数据设置到列名为 key 的 \$this->attributes 中。

添加一个代理列，进行设置，如示例 8-28 所示。

示例8-28 通过Eloquent修改器设置不存在属性的值

```
// 定义修改器
class Order extends Model
{
    public function setWorkgroupNameAttribute($workgroupName)
    {
        $this->attributes['email']="{ $workgroupName}@ourcompany.com";
    }
}

// 使用修改器
$order->workgroup_name = 'jstott';
```

对于不存在的列，一般不会创建一个修改器，因为这样可能会误认为设置一个属性，然后把它变成一个不同的列。

属性转换器

也许你已经思考过通过访问器把所有整型类型的字段变成整数，把 JSON 格式的数据编码解码，然后存入 TEXT 列；或者把 TINYINT 0 和 1 转换成布尔值。

Eloquent 中有这样的功能，也就是属性转换器。属性转换器允许定义任何列在读和写的

时候的类型，就好像它们有特定的数据类型一样。具体如表 8-1 所示。

表8-1 可能的属性转换器列类型

| 类型 | 描述 |
|-------------------|--------------------------------------|
| int integer | 通过 PHP 转换 (int) |
| real float double | 通过 PHP 转换 (float) |
| string | 通过 PHP 转换 (string) |
| bool boolean | 通过 PHP 转换 (bool) |
| object | 作为一个 stdClass 对象，从 JSON 解析或被解析为 JSON |
| array | 作为一个数组，从 JSON 解析或被解析为 JSON |
| collection | 作为一个集合，从 JSON 解析或被解析为 JSON |
| date datetime | 从数据库 DATETIME 解析为 Carbon 类型，然后返回 |
| timestamp | 从数据库 TIMESTAMP 解析为 Carbon 类型，然后返回 |

示例 8-29 展示了怎样在模型中使用属性转化器。

示例8-29 在Eloquent模型中使用属性转化器

```
class Contact
{
    protected $casts = [
        'vip' => 'boolean',
        'children_names' => 'array',
        'birthday' => 'date',
    ];
}
```

日期转换

可以选中一个特定的列，然后把它们加入日期数组，这样它们就会被转换成 timestamp 列，如示例 8-30 所示。

示例8-30 定义会被转换为timestamps类型的列

```
class Contact
{
    protected $dates = [
        'met_at'
    ];
}
```

默认这个数组包含 `creaed_at` 和 `updated_at`，所以添加 `dates` 条目也只是将它们对应的不同值添加到列表中。

5.2 但是把这两列加到列表中和作为时间戳加到 `$this->casts` 中没什么区别，所以在属性转换器中转换时间戳的特性就变得有些不必要了（Laravel 5.2 中的更新）。

Eloquent 集合

当在 Eloquent 中调用一些可能返回多行而不是一个数组的查询时，它们会被封装成一个 Eloquent 集合，这是集合的一种特殊类型。让我们来看一下集合和 Eloquent 集合，以及它们为什么会比普通的数组更好。

集合简介

Laravel 的集合对象 (Illuminate\Support\Collection) 有点像类里的数组。把类似于数组的对象应用到方法中是很有用的，在做非 Laravel 的项目时，可能为了使用集合而在项目中添加 Illuminate 对象。这里可以通过 Tightly Coupled (https://github.com/tightenco/collect) 包来实现。

为了创建一个集合，可以将一个数组传入集合的构造器中，也可以创建一个空的集合，然后把条目写到集合中。Laravel 也有 `collect()` 助手，这是最简单的、新建集合的办法。让我们来试一下，代码如下。

```
$collection = collect([1, 2, 3]);
```

如果想要得到所有偶数，则代码如下。

```
$odds = $collection->reject(function ($item) {  
    return $item % 2 === 0;  
});
```

如果想要得到每一个条目都是 10 的倍数的数组，则代码如下。

```
$multiplied = $collection->map(function ($item) {  
    return $item * 10;  
});
```

也可以只是得到条目，然后把所有值都乘以 10，然后求和，代码如下。

```
$sum = $collection  
->filter(function ($item) {  
    return $item % 2 == 0;  
})->map(function ($item) {  
    return $item * 10;  
})->sum();
```

可以看到集合提供了一系列方法，可以把它们结合起来，然后应用到数组中。集合也提供了与常见 PHP 方法相似的功能，如 `array_map()` 和 `array_reduce()`。这里不需要为

PHP 不可预知的参数顺序预留内存，而且这些方法链接的语法也更加容易读取。

在 `Collection` 类中有 60 多种方法，包括 `max()`、`whereIn()`、`flatten()` 和 `flip()`。因为篇幅有限，这里就不为大家一一介绍了。在第 397 页，会进一步地介绍“集合”，同学们也可以参考集合（<https://laravel.com/docs/master/collections>）的官方文档以了解更多方法。



数组中的集合

只要可以使用数组，就可以使用集合（除了类型提示）。集合可以被迭代，可以在 `foreach` 语句中使用它们时也允许数组访问，因此也可以通过指定键来获取值，如 `$a = $collection['a']`。

Eloquent 集合的扩展功能

Eloquent 集合依旧是一个普通的集合，但是我们会在此基础上对 Eloquent 集合结果的特殊需求进行功能扩展。

由于篇幅有限，我们不能在书里把所有的扩展内容都覆盖到，但它们主要的特别之处在于通过集合进行交互而不仅是通用的对象，对象则是用于表示数据库行的。

例如每一个 Eloquent 集合都有一个 `modelKeys()` 方法，它会返回一个包含集合里所有实例主键的数组。`find($id)` 会查找一个主键为 `$id` 的实例。

Eloquent 集合还有一个特性就是它可以定义任何给定的模型，这个模型将返回的结果封装在集合里的一个特定的类里。所以如果想要为 `Order` 类对象的任何集合添加特殊的方法（比如可能是与汇总订单的财务细节有关），可以创建一个自定义的 `OrderCollection`，它会继承 `Illuminate\Database\Eloquent\Collection` 类，然后在模型中注册，如示例 8-31 所示。

示例8-31 自定义Eloquent模型中的集合类

```
...
class OrderCollection extends Collection
{
    public function sumBillableAmount()
    {
        return $this->reduce(function ($carry, $order) {
            return $carry + ($order->billable ? $order->amount : 0);
        }, 0);
    }
}
```



```
...
class Order extends Model
{
    public function newCollection(array $models = [])
    {
        return new OrderCollection($models);
    }
}
```

现在起, 无论任何 Orders 集合 (如通过 `Order::all()`), 都是 `OrderCollection` 类的实例。

```
$orders = Order::all();
$billableAmount = $orders->sumBillableAmount();
```

Eloquent 序列化

序列化会出现在处理一些复杂数据的时候——数组或对象, 需要将它转换成一个字符串。在网页中, 字符串通常是 JSON 格式的, 但也可能是其他格式的。

对复杂的数据库记录进行序列化, 可能会非常复杂, 这也是很多 ORM 的缺点。但是 Eloquent 免费提供两个非常强有力的方法: `toArray()` 和 `toJson()`。集合也有 `toArray()` 和 `toJson()` 方法, 所以下面这些代码都是有效的。

```
$contactArray = Contact::first()->toArray();
$contactJson = Contact::first()->toJson();
$contactsArray = Contact::all()->toArray();
$contactsJson = Contact::all()->toJson();
```

也可以将一个 Eloquent 实例或集合转换成字符串 (例如 `$string=(string)$contact;`), 但是模型和集合都只会运行 `toJson()` 方法, 然后返回结果。

直接从路由方法返回模型

Laravel 的路由器最终会将所有路由返回成一个字符串, 这里有一个小技巧。如果在控制器中返回一个 Eloquent 调用的结果, 它会自动被转换成一个字符串, 即会在网页中返回 JSON 格式。也就是说, 返回 JSON 的路由就会像示例 8-32 一样简单了。

示例8-32 从路由中直接返回JSON

```
// routes/web.php
Route::get('api/contacts', function () {
    return Contact::all();
});
```

```
Route::get('api/contacts/{id}', function ($id) {
    return Contact::findOrFail($id);
});
```

从 JSON 中隐藏属性

用 JSON 返回 API 的方式非常常见，很多时候我们同样希望在这些内容中隐藏一些特定的属性。每次将数据转换成 JSON 的时候，Eloquent 能非常简单地隐藏任何属性。

可以将属性加入黑名单，把列出来的这些属性隐藏起来，代码如下。

```
class Contact extends Model
{
    public $hidden = ['password', 'remember_token'];
```

或者加入白名单，显示指定的属性，代码如下。

```
class Contact extends Model
{
    public $visible = ['name', 'email', 'status'];
```

这种方法对关系来说同样有效，代码如下。

```
class User extends Model
{
    public $hidden = ['contacts'];

    public function contacts()
    {
        return $this->hasMany(Contact::class);
    }
}
```



加载关系的内容

默认为获得数据库记录时，关系的内容不会被加载，所以是否隐藏这些数据并没有什么影响。但是就像刚才介绍的那样，我们有可能需要获得与这些隐藏的关系相关的记录。在这种情况下，如果隐藏了该关系那些记录，这些关系将不会被包含在该记录的序列化副本中。

如果还没有理解，则可以假设已经正确设置好了所有的关系，然后通过下面的调用得到一个包含所有联系人的 User。

```
$user = User::with('contacts')->first();
```

如果希望通过一条调用让某条属性可见，则需要通过 Eloquent 的 `makeVisible()` 方法实现。

```
$array = $user->makeVisible('remember_token')->toArray();
```



添加生成列到数组并输出 JSON

如果已经为一个不存在的列创建了访问器——比如示例 8-26 中的 `full_name` 列，把它添加到模型的 `$appends` 数组中，然后输出成 JSON 格式，代码如下。

```
class Contact extends Model
{
    protected $appends = ['full_name'];

    public function getFullNameAttribute()
    {
        return "{$this->first_name} {$this->last_name}";
    }
}
```

Eloquent 关系

在关系型数据库模型中，往往希望表与表之间有一定的关系，这也是其名称的由来。Eloquent 提供了简单但更强有力的工具，使得在数据库表中设置表关系的过程变得非常容易。

在本章中，大部分的示例都是围绕包含一个用户对应多个联系人的用户表进行的，这也是一种相对比较常见的情况。

在 ORM 中，比如 Eloquent，一个用户对应多个联系人的情况一般被称为一对多关系。

如果是一个 CRM，那么一个联系人可以被赋值给多个用户，这就是一种多对多关系，即多个用户可以对应同一个联系人，同时每一个用户又可以对应多个联系人。也就是说，一个用户包含或属于多个联系人。

如果每个联系人有多个电话号码，用户想要一个用于 CRM 的、包含每个电话号码的数据库，我们就可以说这个用户通过联系人拥有了多个电话号码。也就是用户对应多个联系人，每个联系人又对应多个电话号码，可以把联系人当作一个中间媒介。

如果每个联系人都包含一个住址，而这里只想要其中一个住址呢？我们可能已经拿到了 `Contact` 中的所有住址字段，但是可能也创建了一个 `Address` 模型，意味着一个联系人对应一个住址。

最后，如何标注喜欢的联系人甚至事件？这就涉及多态的关系了，一个用户有多个评分，有的评分对应的是联系人，而有的评分则对应的是事件。

下面进一步了解怎样定义和访问这些关系。

一对一

首先来看一个简单的例子，一个 `Contact` 对应一个 `PhoneNumber`，如示例 8-33 所示。

示例8-33 定义一对一关系

```
class Contact extends Model
{
    public function phoneNumber()
    {
        return $this->hasOne(PhoneNumber::class);
    }
}
```

在 Eloquent 模型自身的方法 (`$this->hasOne()`) 中，定义关系（至少在模型的实例中）所关联的类是被完全限定类名的。

在数据库中，应该怎样定义关系呢？在前面已经定义了一个 `Contact` 只有一个 `PhoneNumber`，Eloquent 就会为 `PhoneNumber` 类（也可能是 `phone_numbers`）在表中添加一个 `contact_id` 列。如果想要自定义这个列名（比如 `owner_id`），则需要修改定义，代码如下。

```
return $this->hasOne(PhoneNumber::class, 'owner_id');
```

下面的代码能让我们在联系人中访问电话号码。

```
$contact = Contact::first();
$contactPhone = $contact->phoneNumber;
```

在示例 8-33 中已经定义了 `phoneNumber()` 方法，但是它是通过 `->phoneNumber` 来访问的，这就是 Eloquent 神奇的地方。也可以通过 `->phone_number` 访问，这样会返回一个与 `PhoneNumber` 记录相关的、完整的 Eloquent 实例。

但是如果想要从 `PhoneNumber` 中访问 `Contact` 呢？示例 8-34 可以实现这个想法。

示例8-34 定义反向的一对一关系

```
class PhoneNumber extends Model
{
    public function contact()
    {
        return $this->belongsTo(Contact::class);
    }
}
```

然后用同样的方式访问，代码如下。

```
$contact = $phoneNumber->contact;
```



插入关联条目

每个关系类型都有自己关联模型的模式，但是它的核心是将实例传入 `save()` 或者将实例数组传入 `saveMany()`。也可以将属性传入 `create()` 方法，这样将会创建一个实例，代码如下。

```
$contact = Contact::first();

$phoneNumber = new PhoneNumber;
$phoneNumber->number = 8008675309;
$contact->phoneNumbers()->save($phoneNumber);

// 或

$contact->phoneNumbers()->saveMany([
    PhoneNumber::find(1),
    PhoneNumber::find(2),
]);

// 或

$contact->phoneNumbers()->create([
    'number' => '+13138675309'
]);
```

一对多

一对多关系是最为常见的关系。在示例 8-35 中定义了 `User` 对应多个 `Contact`。

示例8-35 定义一个一对多关系

```
class User extends Model
{
    public function contacts()
    {
        return $this->hasMany(Contact::class);
    }
}
```

同样的 Eloquent 会在 `Contact` 模型的备份表(可能是 `contacts`)中创建一个 `user_id` 列。如果没有，则可以将列名作为 `hasMany()` 的第二参数传入，从而重写列名。

下面的代码可以得到一个用户的联系人。

```
$user = User::first();
$usersContacts = $user->contacts;
```

类似于一对一关系，使用关系方法的名字并且调用它就好像它只是一个属性而不是方法。

然而这个方法会返回一个集合而不是模型的实例。它是一个 Eloquent 的集合，所以可以对它进行任何类似 Eloquent 集合的操作。

```
$donors = $user->contacts->filter(function ($contact) {  
    return $contact->status == 'donor';  
});  
  
$lifetimeValue = $contact->orders->reduce(function ($carry, $order) {  
    return $carry + $order->amount;  
}, 0);
```

和一对一关系一样，这里也可以定义反向关系，如示例 8-36 所示。

示例8-36 定义一对多反向关系

```
class Contact extends Model  
{  
    public function user()  
    {  
        return $this->belongsTo(User::class);  
    }  
}
```

也可以通过 Contact 来访问 User，代码如下。

```
$userName = $contact->user->name;
```



从附加内容中附加或分离关联内容

在大多数情况下，我们通过在上级关系上运行 `save()` 并传递相关内容来设置关联关系，如 `$user->contacts()->save($contact)`。但是如果想在关联的（“child”）执行行为，则可以在返回 `belongsTo()` 的方法上使用 `associate()` 和 `dissociate()`。

```
$contact = Contact::first();  
  
$contact->user()->associate(User::first());  
$contact->save();  
  
// 然后  
  
$contact->user()->dissociate();  
$contact->save();
```

将关系用作查询构造器。到目前为止，已经使用了方法名称（例如 `contacts()`），并且把它当作属性进行了调用（例如 `$user->contacts`）。如果我们把它当作方法进行调用

呢？那它就不会处理关系，而是返回一个处理中的查询构造器。

如果拿到了 User 1 并且调用了它的 `contacts()` 方法，那么现在就会得到一个被预先设置为“字段 `user_id` 值为 1 的所有联系人”的查询构建器。然后就可以从中创建一个功能性查询，代码如下。

```
$donors = $user->contacts()->where('status', 'donor')->get();
```

仅查询有关联条目的记录。可以用 `has()` 方法选择满足关联条目特定标准的记录，代码如下。

```
$postsWithComments = Post::has('comments')->get();
```

也可以进一步修改这个标准，代码如下。

```
$postsWithManyComments = Post::has('comments', '>=', 5)->get();
```

还可以嵌套标准，代码如下。

```
$usersWithPhoneBooks = User::has('contacts.phoneNumbers')->get();
```

最后可以在关联条目中自定义查询条件，代码如下。

```
// 得到所有联系人，它们的电话号码中包含字符串 "867-5309"
$jennyIGotYourNumber = Contact::whereHas('phoneNumbers', function ($query) {
    $query->where('number', 'like', '%867-5309%');
});
```

远程一对多关系

“远程一对多关系”是一种非常方便的、建立关系的方法。在前面的示例中也用到了这个方法，即一个 User 有多个 Contacts 而每个 Contacts 对应多个电话号码。如果想要用户的联系人 PhoneNumbers 列表，就需要用到远程一对多关系。

这个结构假定 Contacts 表的 `user_id` 列用于与用户表进行关联，而 `phone_numbers` 表的 `contact_id` 用于与联系人表关联。在示例 8-37 中定义了 User 的关系。

示例8-37 定义远程一对多关系

```
class User extends Model
{
    public function phoneNumbers()
    {
        return $this->hasManyThrough(PhoneNumber::class, Contact::class);
    }
}
```

可以通过 `$user->phone_numbers` 来访问关系，也可以在中间模型（通过 `hasmanyThrough()` 的第三参数）和远程模型（第四参数）自定义关系的键。

多对多

下面的内容会比之前的内容复杂一些。如示例 8-38 所示，CRM 允许 `User` 包含多个 `Contacts`，同时每个 `Contact` 又关联多个用户。

首先在示例 8-38 中，定义了 `User` 上的关系。

示例8-38 定义多对多关系

```
class User extends Model
{
    public function contacts()
    {
        return $this->belongsToMany(Contact::class);
    }
}
```

对于多对多关系来说，反向关系也是一样的，如示例 8-39 所示。

示例8-39 定义反向多对多关系

```
class Contact extends Model
{
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

因为单独的 `Contact` 表不能包含 `user_id` 列，单独的 `User` 表也不能包含 `contact_id` 列，所以多对多关系依赖于连接它们的数据透视表。表的常规命名方式是将两个单数表名称放在一起按字母顺序排列，然后用下划线分隔。

因此连接 `user` 和 `contacts` 的透视表应该命名为 `contacts_users`（如果想要修改表名，只需要将表名作为第二参数传入 `belongsToMany()` 方法），该透视表需要两列：`contact_id` 和 `user_id`。

类似于 `hasMany()`，在多对多关系中可以访问关联内容的集合，但是这里可以实现双向访问，如示例 8-40 所示。

示例8-40 在多对多关系中双向访问关联内容

```
$user = User::first();
```



```

$user->contacts->each(function ($contact) {
    // 进一步操作
});

$contact = Contact::first();

$user->contacts->each(function ($contact) {
    // 进一步操作
});

$donors = $user->contacts()->where('status', 'donor')->get();

```

多对多关系内容中附加和分离的独特性

因为透视表有自己的属性，在添加多对多关系的内容时，就需要设置这些属性。可以通过将数组作为第二参数传入 `save()` 方法进行设置，代码如下。

```

$user = User::first();
$contact = Contact::first();
$user->contacts()->save($contact, ['status' => 'donor']);

```

此外，不要传入一个关联项目的实例，而要使用 `attach()` 和 `detach()` 方法，传入一个 ID 就可以了。它们的工作原理和 `save()` 是一样的，但是它们可以接收一个 ID 数组而不需要将该方法重命名为 `attachMany()`，代码如下。

```

$user = User::first();
$user->contacts()->attach(1);
$user->contacts()->attach(2, ['status' => 'donor']);
$user->contacts()->attach([1, 2, 3]);
$user->contacts()->attach([
    1 => ['status' => 'donor'],
    2,
    3
]);

$user->contacts()->detach(1);
$user->contacts()->detach([1, 2]);
$user->contacts()->detach(); // 分离所有联系人

```

也可以使用 `updateExistingPivot()` 方法来修改透视表中的内容，代码如下。

```

$user->contacts()->updateExistingPivot($contactId, [

```

```
        'status' => 'inactive'
    ]);
```

如果想要替换现在的关系、解除之前所有的关系并添加新的关系，那么可以将数组传入 `sync()`，代码如下。

```
$user->contacts()->sync([1, 2, 3]);
$user->contacts()->sync([
    1 => ['status' => 'donor'],
    2,
    3
]);
```

从透视表中获取数据。多对多与其他关系最大的不同之处在于，它是这些关系中唯一一个有透视表的。透视表中的数据越少越好，但是也有些时候值得把一些信息存到透视表——例如通过存储 `created_at` 字段来记录关系创建的时间。

为了存储这些字段，需要将它们添加到关系的定义中，如示例 8-41 所示。这里可以用 `withPivot()` 方法定义指定的字段或用 `withTimestamps()` 添加 `created_at` 和 `updated_at` 时间戳。

示例8-41 在透视表中添加字段

```
public function contacts()
{
    return $this->belongsToMany(Contact::class)
        ->withTimestamps()
        ->withPivot('status', 'preferred_greeting');
}
```

在通过关系得到一个模型实例时，它可能会包含 `pivot` 属性。这个属性代表了它在透视表中的位置，也就是获取数据的位置，从而可以进行如示例 8-42 所示的操作。

示例8-42 从关联内容的透视条目中获取数据

```
$user = User::first();
$user->contacts->each(function ($contact) {
    echo sprintf(
        'Contact associated with this user at: %s',
        $contact->pivot->created_at
    );
});
```

多态

需要记住的是,当有多个 Eloquent 类对应同一种关系时就会有多态关系。现在以 Stars(类似收藏)为例,一个用户可以给 Contacts 和 Events 打分,这也就是多态的由来:单个接口对应多个类型的对象。

所以我们需要三张表以及三个模型:Star、Contact 和 Event(还有 User,但是这个很容易得到)。contacts 和 events 就是普通的表,stars 表会包含 id 字段、starrable_id 和 starrable_type。对于每一个 star 来说,需要定义类型(如 Contact 或 Event)及该类型的 ID(如 1)。

先来创建模型,如示例 8-43 所示。

示例8-43 从多态评分系统中创建模型

```
class Star extends Model
{
    public function starrable()
    {
        return $this->morphsTo();
    }
}

class Contact extends Model
{
    public function stars()
    {
        return $this->morphMany(Star::class, 'starrable');
    }
}

class Event extends Model
{
    public function stars()
    {
        return $this->morphMany(Star::class, 'starrable');
    }
}
```

要怎样新建一个 Star 呢?代码如下。

```
$contact = Contact::first();
$contact->stars()->create();
```

Contact 已经创建好了。

为了查找一个给定 `Contact` 的所有 `Star`, 这里可以调用 `starts()` 方法, 如示例 8-44 所示。

示例8-44 从多态关系中获取实例

```
$contact = Contact::first();

$contact->stars->each(function ($star) {
    // 自定义操作
});
```

如果有一个 `Star` 的实例, 那么可以调用它来定义 `morphTo()` 的方法, 也就是 `starrable()`, 从而获取这个实例的目标, 如示例 8-45 所示。

示例8-45 从多态实例中获取目标

```
$stars = Star::all();

$stars->each(function ($star) {
    var_dump($star->starrable()); // 联系人或事件的实例
});
```

最后你可能想问, “如果我想知道是谁给这个联系人评分了呢?” 只需要在评分表中添加 `user_id` 字段, 然后设置这个 `User` 有多个 `Stars`, 每个 `Star` 对应一个 `User`——一对多关系, 如示例 8-46 所示。那么这个 `stars` 表就类似于 `User`、`Contacts` 以及 `Events` 之间的透视表了。

示例8-46 继承多态系统来区分用户

```
class Star extends Model
{
    public function starrable()
    {
        return $this->morphsTo;
    }

    public function user()
    {
        return $this->belongsTo(User::class);
    }
}

class User extends Model
{
    public function stars()
    {
        return $this->hasMany(Star::class);
    }
}
```

```
}  
}
```

现在可以运行 `$star->user` 或 `$user->stars` 来得到 User 的 Star 列表或通过 Star 找到打分的 User。同样当创建一个新的评分时，可能想要忽略这个用户，代码如下。

```
$user = User::first();  
$event = Event::first();  
$event->stars()->create(['user_id' => $user->id]);
```

多对多的多态

多对多的多态关系是最复杂的，但是它也是最常见的一种关系。它和多态关系类似，但是多态关系里是一对多的多态，而这里是多对多。

最常见的关系类型就是标签，所以这里用标签类型作为例子进行讲解。如果要给之前例子中的 Contacts 和 Events 加标签，那么这里多对多的多态，其唯一性就在于它的多对多：每一个标签可能会被用于多个条目，而每一个被标注的条目又可能有多个标签。需要注意的是，这是一个多态关系，这些标签可以对应多种不同类型的条目。在数据库中，这里从包含透视表多态关系的普通结构开始讲解。

需要事先准备 contacts 表、events 表、tags 表、所有包含 ID 的常规形状和任何希望具备的属性，以及一个新的 taggables 表，表里包括 tag_id、taggable_id 和 taggable_type 列。在 taggable 表里，每一个条目都将标签与其中一个可标记内容类型相关联。

接下来在模型中定义多对多的多态关系，如示例 8-47 所示。

示例8-47 定义多对多的多态关系

```
class Contact extends Model  
{  
    public function tags()  
    {  
        return $this->morphToMany(Tag::class, 'taggable');  
    }  
}  
  
class Event extends Model  
{  
    public function tags()  
    {  
        return $this->morphToMany(Tag::class, 'taggable');  
    }  
}
```

```

}

class Tag extends Model
{
  public function contacts()
  {
    return $this->morphedByMany(Contact::class, 'taggable');
  }

  public function events()
  {
    return $this->morphedByMany(Event::class, 'taggable');
  }
}

```

创建第一个标签，代码如下。

```

$tag = Tag::firstOrCreate(['name' => 'likes-cheese']);
$contact = Contact::first();
$contact->tags()->attach($tag->id);

```

我们得到了如 normal 这样的关系，结果如示例 8-48 所示。

示例8-48 从多对多多态关系双向访问关联条目

```

$contact = Contact::first();

$contact->tags->each(function ($tag) {
    // 进一步操作
});

$tag = Tag::first();

$tag->contacts->each(function ($contact){
    // 进一步操作
});

```

通过子类更新父类时间戳

任何 Eloquent 模型默认都具有 `created_at` 和 `updated_at` 时间戳。当你对记录进行任何修改时，Eloquent 会自动设置 `updated_at` 时间戳。

当一个条目通过 `belongsTo` 或 `belongsToMany()` 与其他条目关联时，在关联条目被更新时也需要对被关联条目进行更新。例如如果更新了一个 `PhoneNumber`，那么与它关联的 `Contact` 也应该同时更新。

可以通过将该关系的方法名称添加到子类的 `$touches` 数组属性中来实现，如示例 8-49 所示。

示例8-49 同步更新父类和子类记录

```
class PhoneNumber extends Model
{
    protected $touches = ['contact'];
    public function contact()
    {
        return $this->belongsTo(Contact::class);
    }
}
```

预载入

Eloquent 默认采用“懒惰载入”来加载关系，也就是第一次载入模型实例时，它的关联模型不会一起被加载，而只在单独调用的时候才会被加载。

在迭代一个模型实例的列表并且每一个元素都有关联条目时，可能会遇到问题，因为“懒惰载入”会导致严峻的数据库加载问题（这是一个 $N+1$ 问题，如果不熟悉就先跳过）。例如在示例 8-50 中，在每一次循环中都会执行一个新的查询来获取指定联系人的电话号码。

示例8-50

```
$contacts = Contact::all();

foreach ($contacts as $contact){
    foreach ($contact->phone_numbers as $phone_number) {
        echo $phone_number->number;
    }
}
```

如果在加载一个模型实例时知道要使用该模型实例的关系，那么便可以采用“预载入”的方式预先加载一个或多个关联条目，代码如下。

```
$contacts = Contact::with('phoneNumbers')->get();
```

在查询时，使用 `with()` 方法可以获得查询条目的所有关联条目。在示例 8-50 中可以看到我们传入了定义关系的方法名称。

只有当请求（通过 `foreach` 循环每次获得一个电话号码）获取关联条目时才会与默认的加载不同，当我们使用预载入时，单词查询会获得初始条目（查询所有联系人），而二级查询会返回所有关联条目（查询所有初始条目中的联系人包含的所有电话号码）。

可以向 `with()` 调用传入多个参数来预载入多个关系，代码如下。

```
$contacts = Contact::with('phoneNumbers', 'addresses')->get();
```

同样也可以将预载入用于加载关系的关系，代码如下。

```
$authors = Author::with('posts.comments')->get();
```

带条件的的预载入。如果只希望预载入一个关系中的部分条目，那么可以将一个闭包传入 `with()`，从而定义希望预载入的条目，代码如下。

```
$contacts = Contact::with(['addresses' => function ($query){
    $query->where('mailable', true);
}])->get();
```

懒惰的预载入。我们定义的预载入方法仿佛与懒惰载入完全相反，但是有的时候你并不确定自己想要预载入，直到获取初始实例之后。所以可以先采用懒惰载入，在得到实例之后再使用预载入，代码如下。

```
$contacts = Contact::all();

if ($showPhoneNumbers) {
    $contacts->load('phoneNumbers');
}
```

仅预载入计数。如果希望预载入关系，但是权限只能获取关系里的条目数量，那么可以使用 `withCount():`，代码如下。

```
$authors = Author::withCount('posts')->get();

// 为每位作者添加一个 "posts_count" 整数代表该作者的相关帖子数量
```

Eloquent 事件

无论是否监听事件，Eloquent 模型都会把应用程序每次发生的动作记录下来。如果对 `pub/sub` 熟悉，那么 Eloquent 事件也是类似的（在第 16 章中可以进一步学习 Laravel 其他的事件系统）。

下面是一个快速将监听器绑定到新创建的联系人上的方法。我们会将它绑定到 `AppServiceProvider` 的 `boot()` 方法上，然后每次创建一个新的联系人都会通知第三方服务，如示例 8-51 所示。

示例8-51 为Eloquent事件绑定监听器

```
class AppServiceProvider extends ServiceProvider
```



```

{
    public function boot()
    {
        $thirdPartyService = new SomeThirdPartyService;

        Contact::creating(function ($contact) use ($thirdPartyService) {
            try {
                $thirdPartyService->addContact($contact);
            } catch (Exception $e) {
                Log::error('Failed adding contact to ThirdPartyService; cancelled.');

```

在示例 8-51 中，首先使用了 `Modelname::eventName()` 方法，然后将它传入一个闭包，这样这个闭包就有权访问操作的模型实例了。其次需要在服务中定义这个监听器，如果返回的是 `false`，操作就会被取消，而 `save()` 或者 `update()` 方法同样也会被取消。

下面列出了 Eloquent 模型会记录的事件。

- `creating`
- `created`
- `updating`
- `updated`
- `saving`
- `saved`
- `deleting`
- `deleted`
- `restoring`
- `restored`

除了 `restoring` 和 `restored`，大多数操作都很直观，它们是用于记录回复软删除的行。另外，`creating` 和 `updating` 都会触发保存，`created` 和 `updated` 也会。

测试

Laravel 完整的应用测试框架能让你很容易地测试数据库——不是对 Eloquent 进行单元测试，而是进行系统测试。

也许你希望达到的效果是特定界面显示特定的联系人而不显示别的，其中一些逻辑与 URL、控制器和数据库之间的相互作用有关，所以最好的测试方法是系统测试。可能你之前想过模拟 Eloquent 调用来避免数据库冲突，不要这样做，建议尽量采用示例 8-52 中的方法。

示例8-52 通过简单的系统测试来测试数据库交互

```
public function
test_active_page_shows_active_and_not_inactive_contacts()
{
    $activeContact = factory(Contact::class, 'active')->create();
    $inactiveContact = factory(Contact::class, 'inactive')->create();

    $this->visit('active-contacts')
        ->see($activeContact->name)
        ->dontSee($inactiveContact->name);
}
```

模型工厂和 Laravel 的应用程序测试功能都非常适合测试数据库调用。

或者，也可以直接在数据库中查找该记录，如示例 8-53 所示。

示例8-53 使用seeInDatabase()查找数据库中的某些记录

```
public function test_contact_creation_works()
{
    $this->post('contacts', [
        'email' => 'jim@bo.com'
    ]);

    $this->seeInDatabase('contacts', [
        'email' => 'jim@bo.com'
    ]);
}
```

Eloquent 和 Laravel 的数据库框架已经经过了广泛测试，不需要再测试它们，也不需要模拟它们。如果真的想避免与数据库的冲突，可以使用仓库然后返回一个没有保存的 Eloquent 模型的实例。注意，最重要的事情是测试应用的数据库逻辑。

如果应用中包括自定义的访问器、转换器、作用域或者其他自定义内容，那么也可以直接对它们进行测试，如示例 8-54 所示。

示例8-54 测试访问器、转换器和作用域

```
public function test_full_name_accessor_works()
{
```

```

        $contact = factory(Contact::class)->make([
            'first_name' => 'Alphonse',
            'last_name' => 'Cumberbund'
        ]);

        $this->assertEquals('Alphonse Cumberbund', $contact->fullName);
    }

    public function test_vip_scope_filters_out_non_vips()
    {
        $vip = factory(Contact::class, 'vip')->create();
        $nonVip = factory(Contact::class)->create();

        $vips = Contact::vips()->get();
        $this->assertTrue($vips->contains(['id' => $vip->id]));
        $this->assertFalse($vips->contains(['id' => $nonVip->id]));
    }

```

在编写测试的时候，需要避免创建复杂的“Demeter 链”来声明在某些数据库模拟中调用了特定堆栈。如果测试在数据库层面上特别复杂，那是因为测试了一些不必要的、复杂的系统。建议尽量编写简单的测试。

本章小结

Laravel 包括很多功能强大的数据库工具，包括迁移、填充、查询构造器，以及 Eloquent 这个强大的 ActiveRecord ORM。Laravel 的数据库工具也不需要使用 Eloquent——可以很方便地在很窄的层面上访问和操作数据库，甚至也不需要直接编写 SQL 语句。只需要添加一个 ORM，无论是 EloquentDoctrine 还是其他的 ORM，就可以和 Laravel 主要的数据库工具一起工作了。

Eloquent 遵循 Active Record 模式，这样能简单地定义一类支持数据库的对象，包括存储的表列、访问器和转换器等。Eloquent 可以处理任何常规的 SQL 操作以及复杂的关系，甚至多态的多对多关系。

Laravel 还拥有强大的测试数据库的系统，如模型工厂。

用户认证和授权

设置基本的用户认证系统，包括注册、登录、会话、重置密码，以及访问权限——可能是在实现一个应用时最花时间的部分。这个过程需要从库中提取相应的功能，这里有很多这样的库。

在不同的项目中，其需要的认证系统差别很大，并且大多数认证系统很快就不能用了。不过不用担心，Laravel 能让认证系统很容易被理解和使用，也能非常灵活地适用于多种多样的设置。

对于 Laravel 每一个新的安装，除默认的配置外还包含一个 `create_users_table` 迁移和内置的用户模型。Laravel 提供一行 Artisan 命令：`make:auth`，用于填充与认证相关的视图和路由的集合。每一个安装还包含一个 `RegisterController`、`LoginController`、`ForgotPasswordController` 和 `ResetPasswordController`。这些 API 的功能都很清晰明了，它们协同工作并为用户提供了简单易懂的认证和授权系统。



Laravel 5.3 中不同的验证结构

在 Laravel 5.1 和 5.2 中，大多数功能都依赖于 `AuthController`；而在 Laravel 5.3 中，这个功能分散到了多个控制器。这里的讲解内容包括如何自定义重定向路由、验证保护，以及它们与 Laravel 5.1 和 Laravel 5.2 版本的不同之处（所有的核心功能都是一样的）。所以对于 Laravel 5.1 或者 Laravel 5.2 版本的用户，如果希望改变一些默认的认证行为，则可以花时间研究一下 `AuthController`，了解如何进行自定义。

用户模型和迁移

在创建一个 Laravel 应用时,首先看到的就是 create_users_table 迁移和 App\User 模型。在示例 9-1 中,我们可以看到从迁移的 Users 表里可以获取的字段。

示例9-1 Laravel默认的用户迁移

```
Schema::create('users', function (Blueprint $table) {  
    $table->increments('id');  
    $table->string('name');  
    $table->string('email')->unique();  
    $table->string('password');  
    $table->rememberToken();  
    $table->timestamps();  
});
```

该表里包括自增的主键 ID、用户名、唯一的邮箱地址、密码、“记住我”令牌,以及创建和修改的时间戳字段。它包含了在大多数应用中需要进行的基本用户认证信息。



认证和授权的区别

认证是为了验证某个人的身份,允许他在系统中扮演角色。认证包括登录、登出过程,以及任何允许用户在使用应用过程中证明自身身份的工具。

授权用于决定这个认证用户是否可以进行特定的操作,例如授权系统能让我们禁止任何非管理员查看网站收入情况。

User 模型会更复杂一些,如示例 9-2 所示。可以看到 App\User 类本身并不复杂,但是它继承了 Illuminate\Foundation\Auth\User 类,因此它还有一些其他特性。

示例9-2 Laravel默认的用户模型

```
<?php  
// App\User  
  
namespace App;  
  
use Illuminate\Notifications\Notifiable;  
use Illuminate\Foundation\Auth\User as Authenticatable;  
  
class User extends Authenticatable  
{  
    use Notifiable;
```

```

/**
 * 批量赋值的属性
 *
 * @变量数组
 */
protected $fillable = [
    'name', 'email', 'password',
];

/**
 * 从模型的 JSON 表单中排除的属性
 *
 * @变量数组
 */
protected $hidden = [
    'password', 'remember_token',
];
}

<?php
// Illuminate\Foundation\Auth\User

namespace Illuminate\Foundation\Auth;

use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Auth\Passwords\CanResetPassword;
use Illuminate\Foundation\Auth\Access\Authorizable;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\Access\Authorizable as AuthorizableContract;
use Illuminate\Contracts\Auth\CanResetPassword as CanResetPasswordContract;

class User extends Model implements
    AuthenticatableContract,
    AuthorizableContract,
    CanResetPasswordContract
{
    use Authenticatable, Authorizable, CanResetPassword;
}

```



Eloquent 模型更新器

如果对这个概念感到很陌生,则建议先阅读第 8 章中关于 Eloquent 模型的内容。

我们可以从模型中学到什么？首先，用户的信息存储在 `user` 表里，Laravel 可以从类名推断出具体的用户。我们也可以在创建新的用户时填写 `name`、`email` 和 `password` 属性，在把用户信息输出成 JSON 格式时，`password` 和 `remember_token` 属性是不会输出的。

我们还可以从 `Illuminate\Foundation\Auth` 中的合约和特性中看到这个框架中有一些功能（验证授权和重置密码），理论上该框架可以应用于其他模型而不仅仅是 `user` 模型。我们可以单独或一起使用这些功能。

规约和接口

你可能已经注意到了，本书有时候用的是“规约”，有时候用的则是“接口”，因为 Laravel 中大多数的接口都是在 `Contracts` 的命名空间下的。

PHP 接口就是在两个类之间的合约中，其中一个类会以某种方式产生“行为”。这就像两个类之间达成了某种规约，规约更像是有特定意义的名称，而不只是被称为“接口”。

最后，类的规约会通过特定的签名，提供特定的方法。

值得一提的是，`Illuminate\Contracts` 命名空间包含了一组用于实现和指定（type-hint）Laravel 组件的接口。这样我们就能很容易地使用相同的接口来开发相似的组件并且将它们放到应用程序中，从而代替 `Illuminate` 组件。例如当 Laravel 的核心组件需要提示邮件时，Laravel 不会提示 `Mailer` 类而是提示 `Mailer` 规约（接口），这样就能很容易地提示提供邮件信息。第 11 章中会进一步地介绍相应的知识。

`Authenticatable` 规约要求方法 (`getAuthIdentifier()` 等) 允许框架给认证系统的模型实例进行认证，`Authenticatable` 特性包括一些必需的方法，这些方法用于满足一个平均 Eloquent 模型的规约。

`Authorizable` 规约要求方法 (`can()`、`cannot()`) 允许框架为模型的实例授权，让这些实例能够有权访问不同的内容。`Authorizable` 特性提供了用于满足一个平均 Eloquent 模型中 `Authorizable` 规约的一些必要方法。

最后，`CanResetPassword` 规约要求方法 (`getEmailForPasswordReset()`) 允许框架为任何满足规约的实体重置密码。这个特性为一个平均的 Eloquent 模型提供了相应的方法。

到目前为止，在一个数据库（包括迁移）中，我们已经可以很容易地表示一个单独的用户，

并且可以把它从模型实例中提取出来,这个实例是被认证的(登录和登出)、被授权的(检查是否有权访问特定的资源),以及可以发送重置密码的邮件。

使用 auth() 全局助手和认证 Facade

`auth()` 全局助手是最简单的、与认证用户状态进行交互的方式。也可以注入 `Illuminate\Auth\AuthManager` 实例或者使用 `Auth facade` 来实现同样的功能。

最常见的是检查用户是否已经登录(如果用户已经登录, `auth()->check()` 会返回 `true`, 否则 `auth()->guest()` 返回 `true`), 以及获得已登录用户信息(`auth()->user` 或 `auth()->id()` 来获得用户 ID, 如果用户没有登录, 则两个方法都会返回空值)。

示例 9-3 是在控制器中使用全局助手的一个简单示例。

示例9-3 在控制器中使用auth()全局助手

```
public function dashboard()
{
    if (auth()->guest()) {
        return redirect('sign-up');
    }

    return view('dashboard')
        ->with('user', auth()->user());
}
```

Auth 控制器

要怎样让用户登录, 触发密码的重置?

可以在 `Auth-namespaced` 的控制器中进行这些操作, 包括: `RegisterController`、`LoginController`、`ResetPasswordController` 和 `ForgotPasswordController`。

RegisterController

`RegisterController` 也具备 `RegistersUsers` 的特性, 包括怎样显示新用户的注册表、怎样验证输入内容、怎样根据用户的输入创建用户, 以及注册后的重定向。

控制器本身只包含能在特定时间点调用这些特性的钩子 (hook)。这样能让我们在不深入研究代码的情况下, 能很方便地自定义一些常见的行为。

`$redirectTo` 属性定义了用户在注册之后重定向的位置。`validator()` 方法定义了怎样

验证注册信息。`create()` 方法定义了怎样根据用户的注册信息创建一个新的用户。通过示例 9-4 来看看默认的 `RegisterController`。

示例 9-4 Laravel 默认的注册控制器 (`RegisterController`)

```
...
class RegisterController extends Controller
{
    use RegistersUsers;

    protected $redirectTo = '/home';

    ...

    protected function validator(array $data)
    {
        return Validator::make($data, [
            'name' => 'required|max:255',
            'email' => 'required|email|max:255|unique:users',
            'password' => 'required|min:6|confirmed',
        ]);
    }

    protected function create(array $data)
    {
        return User::create([
            'name' => $data['name'],
            'email' => $data['email'],
            'password' => bcrypt($data['password']),
        ]);
    }
}
```

RegistersUsers 特性

`RegisterController` 导入的 `RegistersUsers` 特性实现了注册过程中的几个主要功能。首先，它通过 `showRegistrationForm()` 方法展示了用户注册表单视图。如果不想用 `auth.register` 默认的视图，则可以在 `RegisterController` 中重写 `showRegistrationForm()` 方法。

其次，它通过 `register()` 方法来处理注册表单中的 `POST` 请求。这个方法将用户的注册输入传给 `RegisterController` 中 `validator()` 方法的验证器，然后将它作用于 `create()` 方法。

最后，`redirectPath()` 方法（继承了 `RedirectsUsers` 特性）定义了用户成功注册后重定向的网址。可以在控制器中通过修改 `redirectTo` 属性定义这个重定向网址，或者通过重写 `redirectPath()` 方法来返回想要的输出结果。

如果希望这个特性不适用默认认证保护（可以在第 204 页中进一步学习“保护”），则可以重写 `auth()` 方法。

LoginController

`LoginController` 允许用户登录。它引入了 `AuthenticatesUsers` 特性，以 `RedirectsUsers` 和 `ThrottlesLogins` 特性。

与 `RegisterController` 相似，`LoginController` 也有一个 `$redirectTo` 属性，它可以让自定义用户成功登录后的重定向地址。其他所有的功能都依赖于 `AuthenticatesUsers` 特性。

AuthenticatesUsers 特性

`AuthenticatesUsers` 特性用于显示用户的登录表格、验证登录、处理失败的登录、处理登出，以及重定向成功的登录。

`showLoginForm()` 方法默认显示用户的 `auth.login` 视图，也可以对它进行重写。

`login()` 方法接受从登录表单的 `POST` 请求，它会通过 `validateLogin()` 方法验证请求。如果想要自定义验证方式，则可以重写 `validateLogin()` 方法。这个方法会接着调用 `ThrottlesLogins` 特性中的功能来拒绝多次登录失败的用户。最后它会将用户重定向到用户希望的网址（在访问应用界面时，用户想要被重定向到登录界面）或者在 `redirectPath()` 方法中定义网址，从而返回 `$redirectTo` 属性。

在成功登录之后，`AuthenticatesUsers` 特性会调用空的 `authenticated()` 方法，所以可以重写 `LoginController` 中的这个方法，自定义成功登录后的任何响应效果。

`username()` 方法定义了 `users` 中名为“username”的列，其默认为 `email`，但是也可以通过在控制器中重写 `username()` 方法来返回 `username` 列的名称。

同 `RegistersUsers` 特性一样，也可以在控制器中重写 `guard()` 方法，从而定义控制器的认证保护。

ThrottlesLogins 特性

`ThrottlesLogins` 特性是 Laravel 的 `Illuminate\Cache\RateLimiter` 类的一个接口，

它的作用是利用缓存来限制事件的速率。这个特性将速率限制应用到用户的登录过程中，如果用户在一定时间内多次登录失败，它会限制用户使用登录表格。在 Laravel 5.1 版本中没有这个功能。

如果引入了 `ThrottlesLogins` 特性，那么所有的方法都是受到保护的，也就是说不能像访问路由那样访问它。`AuthenticatesUsers` 特性会检查是否引入了 `ThrottlesLogins` 特性，如果已经引入，它会把这些所有的功能自动附加到登录中。因为默认 `LoginController` 引入了这两个特性，所以如果使用认证框架，就可以直接得到这些功能。

`ThrottlesLogins` 限制了每 60 秒，任何给定的用户名和 IP 地址的组合只能尝试登录 5 次。缓存，会增加用户名或 IP 地址的失败登录次数。如果任何用户在 60 秒内失败了 5 次，那么 `ThrottlesLogins` 会把用户重定向到登录界面，并显示错误信息：60 秒之后才能再次尝试登录。

ResetPasswordController

`ResetPasswordController` 只是简单地引入了 `ResetPasswords` 特性，它为密码重置视图提供了验证和访问功能，然后利用 Laravel 的 `PasswordBroke` 类的实例（或者其他任何实现了 `PasswordBroker` 的接口）来发送重置密码的邮件，以及实现密码重置功能。

`ResetPasswords` 特性显示了重置密码视图（`showResetForm()` 显示 `auth.passwords.reset` 视图），并且会发送 POST 请求（`reset()` 方法用于验证和发送响应）。`resetPassword()` 方法用于重置密码，同样可以通过 `broker()` 和 `guard()` 来分别自定义 `broker` 和 `guard`。

如果想要自定义任何行为，只需要在对应的控制器中重写特定的方法。

ForgotPasswordController

`ForgotPasswordController` 引入了 `SendsPasswordResetEmails` 特性。它通过 `showLinkRequestForm()` 方法显示 `auth.passwords.email` 表单，并且通过 `sendResetLinkEmail()` 方法处理表单的 POST 请求。还可以通过 `broker()` 方法自定义 `broker`。

Auth::routes()

认证控制器为一系列预定义的路由提供了一些方法，如果希望用户能够访问这些路由，

则可以将这些路由手动加到 routes/web.php 下但是更方便的工具是 :Auth::routes()

```
// routes/web.php
Auth::routes();
```

Auth::routes() 包含路由文件里的预定义路由。在示例 9-5 中, 可以看到实际被定义的路由。

示例9-5 Auth::routes()提供的路由

```
// 认证路由
$this->get('login', 'Auth\LoginController@showLoginForm');
$this->post('login', 'Auth\LoginController@login');
$this->get('logout', 'Auth\LoginController@logout');

// 注册路由
$this->get('register', 'Auth\RegisterController@showRegistrationForm');
$this->post('register', 'Auth\RegisterController@register');

// 重置密码路由
$this->get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm');
$this->post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail');
$this->get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm');
$this->post('password/reset', 'Auth\ResetPasswordController@reset');
```

一般来说, Auth::routes() 包括认证路由、注册路由和重置密码路由。

Laravel 的控制器 / 方法参考语法

在给定控制器时, Laravel 提供了参照特定方法的会话 *ControllerName@methodName*。有时候它只是普通的交流习惯, 但是也会用于实际的绑定中, 如示例 9-5 所示。Laravel 会解析 @ 符号之前和之后的内容, 然后使用这些片段识别控制器和方法。

认证脚手架 (Auth Scaffold)

现在在认证系统中, 已经有了迁移模型控制器和路由, 那么视图呢?

Laravel (在 Laravel 5.2 版本中的新功能) 提供了一个认证脚手架, 它在应用中运行时, 提供了更多的框架代码使认证系统的运行速度更快。

认证脚手架往路由文件中添加了 `Auth::routes()`，并为每个路由添加了一个视图；创建了一个 `HomeController` 作为登录用户的登录后界面；同时还把 `HomeController` 的 `index()` 方法指向了 `/home` 的 URL。

只需要运行 `artisan` 的 `make:auth` 命令，就可以使用下面的文件了。

```
app/Http/Controllers/HomeController.php
resources/views/auth/login.blade.php
resources/views/auth/register.blade.php
resources/views/auth/passwords/email.blade.php
resources/views/auth/passwords/reset.blade.php
resources/views/layouts/app.blade.php
resources/views/home.blade.php
```

这里可以用 `/` 返回 `welcome` 视图，`/home` 返回 `home` 视图，以及指向认证控制器的一系列登录、登出、注册和密码重置的认证路由。每个填充视图都有基于 `Bootstrap` 的布局和用于登录、注册和密码重置的表单字段，然后这些字段都被指向了正确的路由。

到现在为止，已经了解了一般用户注册和认证流程中每一步所需要的组件，并且可以进行用户注册和认证，接下来可以根据自己的需求进行调整。

让我们快速回顾一下完整的认证系统的步骤，代码如下。

```
laravel new MyApp
cd MyApp
php artisan make:auth
php artisan migrate
```

运行这些命令后，就会得到一个适用于所有认证用户的、包含登录界面以及基于 `Bootstrap` 布局的注册、登录、登出和密码重置系统。

“记住我”

认证脚手架现在已经可以直接使用了，但仍然需要学习它是如何工作的，以及如何使用它。如果想要实现“记住我”类型的长期有效访问令牌，则需要在 `users` 表里包含 `remember_token` 列（如果使用的是默认的迁移）。

当一个用户正常登录时（这也是 `LoginController` 通过 `AuthenticatesUsers` 特性实现的功能），需要根据用户提供的信息“尝试”进行认证，如示例 9-6 所示。

示例9-6 尝试一次用户认证

```
if (auth()->attempt([
    'email' => request()->input('email'),
    'password' => request()->input('password')
])) {
    // 处理成功的登录
}
```

这样提供了一个可以持续保持的用户会话。如果希望 Laravel 能通过 cookie，无限延长登录（只要用户在一台电脑上不登出），那么只需要将一个布尔值型的 `true` 作为第二参数传入 `auth()->attempt()` 方法。让我们通过示例 9-7 来看看如何实现。

示例9-7 勾选“记住我”复选框的用户认证

```
if (auth()->attempt([
    'email' => request()->input('email'),
    'password' => request()->input('password') ]),
    request()->has('remember')) {
    // 处理成功的登录
}
```

我们检查了输入是否有 `remember` 属性时，它会返回一个布尔值。这允许我们的用户决定是否要在登录表单中使用复选框来记住他们。

如果想要手动检查当前用户是否已经被记住，可以通过令牌认证实现：返回一个布尔值的 `auth()->viaRemember()`。这将允许通过记住令牌来阻止用户访问某些敏感信息，同时也可以要求用户重新输入密码。

手动认证用户

在用户认证中最常见的就是用户提供验证信息，然后系统通过 `auth()->attempt()` 方法验证信息，以及判断是否匹配实际用户信息。如果匹配，则用户可以登录。

但是有的时候，我们需要登录其中一个用户，比如希望管理员可以切换用户登录。

有两种方法可以实现这个功能。第一种，可以传入一个用户 ID，代码如下。

```
auth()->loginUsingId(5);
```

第二种，传入一个用户对象（或者其他实现 `Illuminate\Contracts\Auth\Authenticatable` 规约的对象），代码如下。

```
auth()->login($user);
```

认证中间件

在示例 9-3 中可以看到，如果想要检查访问者是否已经登录和重定向，可以在程序中每一个路由中都进行同样的检查，但是这样操作起来太烦琐了。这时就用到了中间件（第 10 章中会进行深入学习），它非常适合对访问或者认证用户进行重定向。Laravel 的中间件是可以直接使用的，下面查看在 `App\Http\Kernel` 中定义的路由中间件。

```
protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::
class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
];
```

三个路由中间件都是认证相关的，其中 `auth` 限制了认证用户访问路由；`auth.basic` 限制了认证用户使用 HTTP 基本认证的访问；`guest` 限制了未认证用户。而 `can` 用于指定认证用户是否能访问指定路由。

对于认证用户部分，最常见的是使用 `auth` 路由中间件，`guest` 表示不希望认证用户查看的路由（比如登录表单），`auth.basic` 一般用于通过请求表头的认证。

示例 9-8 显示了一些认证中间件保护的路由。

示例9-8 认证中间件保护的路由。

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('account', 'AccountController@dashboard');
});

Route::get('login', 'Auth\LoginController@getLogin')->middleware('guest');
```

保护

Laravel 认证系统的每个部分都通过“保护”来传递。保护分为两个部分：驱动，定义了如何执行和获取认证状态（例如会话）：提供者，它允许通过某些标准（例如用户）来获取用户。

Laravel 有两种直接可以使用的保护：`web` 和 `api`。`web` 代表一种比较传统的认证风格，它使用会话驱动和基本的用户提供者。`api` 使用同样的用户提供者，但是它为每个请求

使用令牌驱动认证。

如果想要处理不同用户的身份识别和持久性，则可以修改驱动（例如将一个长期运行的会话修改为每一页特定的令牌）；如果希望修改存储类型或者获取数据的方法（例如将用户存储到 Mongo 而不是 MySQL 中），则也可以修改提供者。

修改默认保护

保护被定义在 `config/auth.php` 中，并且可以修改、添加新的保护和定义的默认保护。

默认的保护可以用于任何使用认证的时候。`auth()->user()` 会使用默认保护来获取当前认证的用户。可以在 `config/auth.php` 的 `auth.defaults.guard` 设置中修改默认保护，代码如下。

```
'defaults' => [
    'guard' => 'web', // Change the default here
    'passwords' => 'users',
],
```

如果使用的是 Laravel 5.1 版本，则可能发现它的认证信息结构有一点不同。但是不用担心，它们只是结构不同，但是依旧可以正常工作。



配置公约

对于配置部分（比如 `auth.defaults.guard`），在 `config/auth.php` 中，数组部分键入的默认值中应该有一个键入了保护的属性，也就是 `auth.defaults.guard`。

在不改变默认情况下使用其他保护

如果想使用其他的保护但是又不想改变默认值，那么可以通过 `guard()` 启用 Auth 调用，代码如下。

```
$apiUser = auth()->guard('api')->user();
```

这个调用可以通过 `api` 保护得到当前用户。

添加新的保护

无论任何时候，都可以在 `config/auth.php` 中的 `auth.guards` 设置中添加一个新的保护，代码如下。

```
'guards' => [
    'trainees' => [
        'driver' => 'session',
        'provider' => 'trainees',
    ],
],
```

这样就创建了一个新的保护（除了 `web` 和 `api`），名为 `trainees`。想象一下，在应用的其他部分中，我们的用户是一些物理教练，他们每个人都有自己的用户（学员）可以登录到他们的子域，这时需要将这些用户的保护分开。

`driver` 的两个选项是 `token` 和 `session`，这里可以直接使用的 `provider` 是 `users`，也可以很轻松地创建自己的提供者。

创建自定义用户提供者

在 `config/auth.php` 定义的保护下面，`auth.providers` 部分是用来定义提供者的。接下来创建一个名为 `trainees` 的提供者，代码如下。

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\User::class,
    ],

    'trainees' => [
        'driver' => 'eloquent',
        'model' => App\Trainee::class,
    ],
],
```

`driver` 的两个选项是 `eloquent` 和 `database`：如果使用 `eloquent`，则需要有一个包含 Eloquent 类名的模型属性（这个模型用于 `User` 类）；如果使用 `database`，则需要有一个定义哪个表可以被认证的表属性。

在这个示例中，应用包含 `User` 和 `Trainee`，它们需要分开认证。所以 `auth()->guard(users)` 和 `auth()->guard(trainees)` 的代码是不同的。

最后，认证路由中间件可以带一个指定保护名字的参数。可以通过特殊的保护来保护特定的路由，代码如下。

```
Route::group(['middleware' => 'auth:trainees'], function () {
    // Trainee-only routes here
});
```

为非关系型数据库自定义用户提供者

用户提供者创建流程也依赖于 `UserProvider` 类，也就是说它希望能从关系型数据库中获取身份信息。但是如果用的是 `Mongo`、`Riak` 或者其他非关系型数据库，则需要创建自己的类。

这里需要创建一个实现了 `Illuminate\Contracts\Auth\UserProvider` 接口的类，然后将它绑定到 `AuthServiceServiceProvider@boot` 上，代码如下。

```
auth()->provider('riak', function ($app, array $config){
    // 返回一个 Illuminate\Contracts\Auth\UserProvider 实例 ...
    return new RiakUserProvider($app['riak.connection']);
});
```

认证事件

我们会在第 16 章中更详细地介绍这个知识点。Laravel 的事件系统是一个基本的发布 / 订阅框架。它包含系统以及用户生成的事件，这些事件都是广播事件，用户可以创建事件监听来对特定的事件进行特定的响应。

如果一个用户因为多次登录失败尝试后账户被锁，而开发者想要向特定的安全服务发一个 ping 请求，该怎么办呢？或许这个安全服务可以对特定的地理区域的特定数量的失败登录进行监听等。当然可以在合适的控制器中拒绝调用，但是对于事件，只需要创建一个事件监听器，然后监听“用户被锁”事件并注册这个监听。

下面通过示例 9-9 来了解认证系统发出的所有事件。

示例9-9 框架生成的认证事件

```
protected $listen = [
    'Illuminate\Auth\Events\Attempting' => [],
    'Illuminate\Auth\Events>Login' => [],
    'Illuminate\Auth\Events\Logout' => [],
    'Illuminate\Auth\Events\Lockout' => [],
];
```

在示例 9-9 中可以看到，包括对“用户尝试登录”“成功登录”“登出”“被锁”等事件监听。对于更多为事件创建监听的知識，请查阅第 16 章。

授权（ACL）和角色

最后，学习一下授权系统。它能决定用户是否有权限进行某些操作，可以通过几个主要的动词进行检查：can、cannot、allows，以及 denies。访问控制列表系统（ACL）是 Laravel 5.2 中的新功能。

绝大多数的授权控制都是通过 Gate facade 实现的，但是控制器也有很便利的助手。它们在 User 模型中授权作为中间件，可用作 Blade 指令。让我们先来看一个示例，代码如下。

```
if (Gate::denies('edit', $contact)) {
    abort(403);
}

if (! Gate::check('create', Contact::class)) {
    abort(403);
}
```

定义授权规则

定义授权规则的默认位置是 AuthServiceProvider 中的 boot() 方法。它应该已经有一个 Illuminate\Contracts\Auth\Access\Gate（别名为 GateContract）类型的实例，并注入了 \$gate。

授权规则也叫作 *ability*(能力)，它由两部分组成：一个字符串键（如 update-contact）和一个返回布尔值的闭包。示例 9-10 展示一个更新联系人的授权规则。

示例9-10 更新联系人的授权规则

```
class AuthServiceProvider extends ServiceProvider
{
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);

        $gate->define('update-contact', function ($user, $contact){
            return $user->id === $contact->user_id;
        });
    }
}
```

让我们来看看定义 ability 的步骤。

首先，需要定义一个键。对于键的名字，应该考虑什么字符串能让这个 ability 在代码中通俗易懂。这里可以参考示例中的公约 {verb}-{modelName}：create-contact、

update-contact 等。

第二，定义闭包。第一个参数应该是当前的认证用户，后面的所有参数都是在实例中（contact 实例）中需要验证权限的对象。

所以给定两个对象，可以检查这个用户是否有权限更新联系人，也可以编写自定义的逻辑。但是我们现在看到的这个应用授权取决于 contact 行的创建器。如果当前用户创建了联系人，闭包会返回 true（授权），否则返回 false（未授权）。

和路由定义一样，这里也可以使用类和方法，而不是闭包，代码如下。

```
$gate>define('updatecontact','ContactACLChecker@updateContact');
```

Gate facade（和注入 Gate）

现在已经定义了 ability，接下来测试一下。最简单的办法就是使用 Gate facade，如示例 9-11 所示（也可以注入一个 Illuminate \Contracts\Auth\Access\Gate 实例）。

示例9-11 基本的Gate facade使用

```
if (Gate::allows('update-contact', $contact)) {  
    // 更新 contact  
}  
  
// 或 ...  
if (Gate::denies('update-contact', $contact)) {  
    abort(403);  
}
```

也可以使用多个参数来定义一个 ability——对某个组里的联系人授权，然后希望联系人能够向组里添加其他联系人，如示例 9-12 所示。

示例9-12 多参数的ability

```
// 定义  
$gate->define('add-contact-to-group', function ($user,$contact, $group) {  
    return $user->id === $contact->user_id && $user->id === $group->user_id;  
});  
  
// 使用  
if (Gate::denies('add-contact-to-group', [$contact, $group])){  
    abort(403);  
}
```

这里需要用 forUser() 来尝试检查用户的权限，而不是当前的认证用户，如示例 9-13 所示。

示例9-13 为Gate指定用户

```
if (Gate::forUser($user)->denies('create-contact')){
    abort(403);
}
```

Authorize 中间件

如果想要对整个路由授权，则可以使用 `Authorize` 中间件（它包含 `can` 的快捷方式），如示例 9-14 所示。

示例9-14 使用Authorize中间件

```
Route::get('people/create', function () {
    // 创建 person...
})->middleware('can:create-person');

Route::get('people/{person}/edit', function () {
    // 创建 person...
})->middleware('can:create,person');
```

这里的 `{person}` 参数（不管它是一个字符串还是绑定的模型路由），会作为一个附件参数传入 `ability` 方法。

控制器授权

在 Laravel 中，父类 `App\Http\Controllers\Controller` 引入了 `AuthorizesRequests` 特性，它提供了三种授权方法，分别是 `authorize()`、`authorizeForUser()` 和 `authorizeResource()`。

`authorize()` 的参数包括一个 `ability` 键和一个对象（或一组对象），如果授权失败它就会退出应用并显示 403（未授权）状态码。也就是说这个特性可以把三行授权码转换为一行，如示例 9-15 所示。

示例9-15 通过authorize()简化控制器授权

```
// 从
public function show($contactId)
{
    $contact = Contact::findOrFail($contactId);

    if (Gate::cannot('update-contact', $contact)) {
        abort(403);
    }
}
```

```
// 到
public function show($contactId)
{
    $contact = Contact::findOrFail($contactId);

    $this->authorize('update-contact', $contact);
}
```

`authorizeForUser()` 的原理相同，但是它允许传入一个 `User` 对象而不是默认传入当前的授权用户，代码如下。

```
$this->authorizeForUser($user, 'update-contact', $contact);
```

可以在控制器构造器中调用 `authorizeResource()`，它可以把一组预定义的授权规则映射到控制器中每一个 RESTful 控制器方法中，如示例 9-16 所示。

示例9-16 `authorizeResource()`的授权-方法映射

```
...
class ContactsController extends Controller
{
    public function __construct()
    {
        // 这个调用会完成下面所有的方法
        // 如果把这个调用放到这里，就可以删除 resource 方法中所有的授权调用
        $this->authorizeResource(Contact::class);
    }

    public function index()
    {
        $this->authorize('view', Contact::class);
    }

    public function create()
    {
        $this->authorize('create', Contact::class);
    }

    public function store(Request $request)
    {
        $this->authorize('create', Contact::class);
    }

    public function show(Contact $contact)
    {

```

```

        $this->authorize('view', $contact);
    }

    public function edit(Contact $contact)
    {
        $this->authorize('update', $contact);
    }

    public function update(Request $request, Contact $contact)
    {
        $this->authorize('update', $contact);
    }

    public function destroy(Contact $contact)
    {
        $this->authorize('delete', $contact);
    }
}

```

检查用户实例

如果现在还没有在控制器里，那么可能需要检查一下特定用户的容量。这里可以使用 Gate facade 的 `forUser()` 方法来实现，但是需要注意语法。

User 类的 Authorizable 特性提供了三种方法，让授权特性更易读：`$user->can()`、`$user->cant()` 和 `$user->cannot()`。`cant()` 和 `cannot()` 是一样的，而 `can()` 具有相反的效果。

这里还可以进行如示例 9-17 所示的操作。

示例9-17 检查用户实例的授权

```

$user = User::find(1);

if ($user->can('create-contact')) {
    // 进一步操作
}

```

此外，这些方法只是将参数传给了 Gate。在以上示例中，可以使用 `Gate::forUser($user)->can('create-contact')`

Blade 检查

Blade 也有比较方便的助手，也就是 @can 指令。它的用法如示例 9-18 所示。

示例9-18 使用Blade的@can指令

```
<nav>
    <a href="/">Home</a>
    @can('edit-contact', $contact)
        <a href="{{ route('contacts.edit', [$contact->id]) }}">Edit This Contact</a>
    @endcan
</nav>
```

也可以在 @can 和 @endcan 之间使用 @else，或者像示例 9-19 一样使用 @cannot 和 @endcannot。

示例9-19 使用Blade的@cannot指令

```
<h1>{{ $contact->name }}</h1>
@cannot('edit-contact', $contact)
    LOCKED
@endcannot
```

插入检查

如果之前创建过带管理员用户类的应用，那么应该已经了解了简单的授权闭包，以及怎样添加一个超用户类来重写这些检查。别担心，Laravel 已经提供了这个工具。

在 AuthServiceProvider 中，已经定义了 ability，这里也可以添加一个 before() 检查，它会在所有的检查之前运行，而且可以重写 before() 检查，如示例 9-20 所示。

示例9-20 重写Gate的before()检查

```
$gate->before(function ($user, $ability) {
    if ($user->isOwner()) {
        return true;
    }
});
```

需要注意的是，这个 ability 的字符串名字也被传入了，所以可以根据 ability 的命名方案区分 before 钩子。

政策

到现在为止，所有的访问控制都要求手动将 ability 的名字与 Eloquent 模型进行关联。可以创建一个名为类似 visit-dashboard 的 ability，它与特定的 Eloquent 模型无关，但是我们大多数的例子都是从做某事到某事，而且在多数情况下某事是一个 Eloquent 模型，是动作的接受者。

授权政策是帮助根据需要控制访问权限的资源，从而组合授权逻辑的组织结构。它让我们能更容易地为指向特定 Eloquent 模型的行为定义授权规则（或者其他 PHP 类），能在一个单独的位置进行管理。

生成政策

政策是 PHP 类，可以通过 Artisan 命令生成，代码如下。

```
php artisan make:policy ContactPolicy
```

一旦生成了政策，就需要注册这些政策。AuthServiceProvider 具备 \$policies 属性，它是数组类型的。每个元素的键值是保护资源的类名（在多数情况下是 Eloquent 类，但不是绝对的），数组的值是政策的类名，代码如下。

```
class AuthServiceProvider extends ServiceProvider
{
    protected $policies = [
        Contact::class => ContactPolicy::class,
    ]
}
```

政策类是通过 Artisan 指令生成的，不包含任何特殊的属性或方法。但是添加的每个方法都会映射为此对象的功能键。

接下来定义 update() 方法，然后看看它是怎么工作的，如示例 9-21 所示。

示例9-21 update()政策方法示例

```
<?php

namespace App\Policies;

class ContactPolicy
{
    public function update($user, $contact)
    {
        return $user->id === $contact->user_id;
    }
}
```

注意，这个方法与 Gate 的定义类似。



Laravel 5.3 版本之前的政策方法不包含实例

5.2 在 Laravel 5.2 中，如果想要定义一个关联到类但是不关联实例的政策方法（例如“这个用户可以创建联系人吗”，而不只是“这个用户可以查看特定的联系人吗”），则需要创建一个方法，然后在名字末尾加一个“Any”，代码如下。

```
...
class ContactPolicy
{
    public function createAny($user)
    {
        return $user->canCreateContacts();
    }
}
```

在 Laravel 5.3 版本中，可以不加 Any 后缀，只需要把它当作一个普通方法。

检查政策

为了定义资源类型政策，Gate 会通过第一参数指出需要检查的方法。如果运行 `Gate::allows('update', $contact)`，那么它会为 `ContactPolicy@update` 方法检查授权。

同样，这也可以用于检查 Authorize 中间件 User 模型和 Blade，如示例 9-22 所示。

示例9-22 检查政策的授权

```
// Gate
if (Gate::denies('update', $contact)) {
    abort(403);
}

// 不包含明确实例的 Gate
if (! Gate::check('create', Contact::class)) {
    abort(403);
}

// User
if ($user->can('update', $contact)) {
    // Do stuff
}

// Blade
@can('update', $contact)
    <!-- show stuff -->
</can>
```

@endcan

此外，`policy()` 也有一个助手，允许获取政策类及运行它的方法，代码如下。

```
if (policy($contact)->update($user, $contact)) {  
    // 操作  
}
```

重写政策

与普通的 `ability` 定义类似，政策也可以定义 `before()` 方法，它允许在处理任何调用之前重写调用，如示例 9-23 所示。

示例9-23 通过`before()`方法重写政策

```
public function before($user, $ability)  
{  
    if ($user->isAdmin()) {  
        return true;  
    }  
}
```

Passport 和 OAuth

Laravel 有一个 Passport 包，它把 OAuth 服务器当作 Laravel 应用的一部分，从而可以很容易地对其进行配置。具体请参考第 295 页“使用 Laravel Passport 的 API 认证”。

测试

应用程序测试通常用于测试特定用户的特定行为。因此需要能够在应用程序测试中以一个用户的身份进行身份验证，然后测试授权规则和身份验证路由，再测试应用的授权规则和路由。

当然也可以编写一个应用测试来手动访问登录界面，然后填写表格并提交，但是没有必要这样做。最简单的办法是使用 `->be()` 方法来模拟登录一个用户，如示例 9-24 所示。

示例9-24 在应用测试中作为一个用户被认证

```
public function test_it_creates_a_new_contact()  
{
```

```

$user = factory(User::class)->create();
$this->be($user);

$this->post('contacts', [
    'email' => 'my@email.com'
]);

$this->seeInDatabase('contacts', [
    'email' => 'my@email.com',
    'user_id' => $user->id,
]);
}

```

也可以像示例 9-25 这样测试授权。

示例9-25 测试授权规则

```

public function test_non_admins_cant_create_users()
{
    $user = factory(User::class)->create([
        'admin' => false
    ]);
    $this->be($user);
    $this->post('users', ['email' => 'my@email.com']);

    $this->dontSeeInDatabase('users', [
        'email' => 'my@email.com'
    ]);
}

```

还可以像示例 9-26 这样，测试 403 响应。

示例9-26 通过检查状态码测试授权规则

```

public function test_non_admins_cant_create_users()
{
    $user = factory(User::class)->create([
        'admin' => false
    ]);
    $this->be($user);

    $this->post('users', ['email' => 'my@email.com']);

    $this->assertResponseStatus(403);
}

```

还需要测试认证路由的工作情况（注册和登录），如示例 9-27 所示。

示例9-27 测试认证路由

```
public function test_users_can_register()
{
    $this->post('register', [
        'name' => 'Sal Leibowitz',
        'email' => 'sal@leibs.net',
        'password' => 'abcdefg123',
        'password_confirmation' => 'abcdefg123',
    ]);

    $this->followRedirects()->assertResponseOk();

    $this->seeInDatabase('users', [
        'name' => 'Sal Leibowitz',
        'email' => 'sal@leibs.net',
    ]);
}

public function test_users_can_log_in()
{
    $user = factory(User::class)->create([
        'password' => bcrypt('abcdefg123')
    ]);

    $this->post('login', [
        'email' => $user->email,
        'password' => 'abcdefg123',
    ]);

    $this->followRedirects()->assertResponseOk();

    $this->assertTrue(auth()->check());
}
```

也可以使用集合测试功能，然后通过“点击”认证字段和“提交”字段来测试全部流程。

本章小结

在默认的用户模型、`create_users_table` 迁移、认证控制器和认证脚手架之间，Laravel 有一个可以开箱即用的、完整的用户认证系统。`RegisterController` 用于处理用户注册；`LoginController` 用于处理用户认证、`ResetPasswordController` 和 `ForgotPasswordController` 用于处理密码。每一个控制器都有一些可用于重写某些默认行为的属性和方法。

Auth facade 和 `auth()` 全局助手为当前用户提供访问 (`auth()->user()`)，它们也能轻易地检查出用户是否已经登录 (`auth()->check()` 和 `auth()->guest()`)。

Laravel 内置的认证系统可以定义特定的 ability (`create-contact`、`visit-secret-page`) 或者为用户与模型进行交互的政策。

可以通过 Gate facade 来检查授权，包括 User 类中的 `->can()` 和 `->cannot()` 方法、Blade 中的 `@can` 和 `@cannot` 指令，以及控制器或 can 中间件中的 `->authorize()` 方法。

请求和响应

在前面的内容中，我们提到过 Illuminate 的 `Request` 对象——怎样在构造器中输入类型以获取实例，然后通过这个实例获取用户输入信息。第 3 章介绍了怎样在构造器中输入类型来获取实例；第 6 章介绍了怎样使用这个实例获取用户的输入信息。

在本章中，我们会学习什么是 `Request` 对象、怎样生成它、该对象代表什么，以及它在应用程序的生命周期中扮演的角色。我们还会学习响应对象以及 Laravel 中间件模式的实现。

Laravel 请求的生命周期

不管是从 HTTP 还是命令行交互界面生成的 Laravel 请求，都会立刻被转换成 Illuminate `Request` 对象，它会经过很多层直到被应用解析。然后应用会生成一个 Illuminate `Response` 对象，它会被发送回来，经过很多层直到返回终端用户。

请求 / 响应生命周期，如图 10-1 所示。

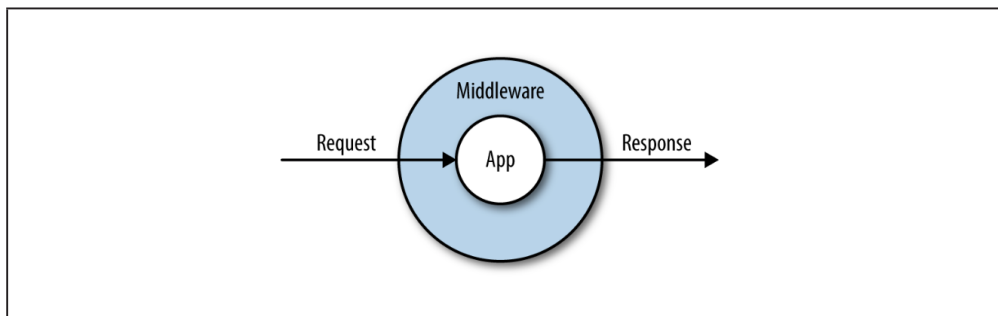


图10-1 请求/响应生命周期

每个 Laravel 应用程序都包含很多 Web 服务器级别的部署，它们位于 `.htaccess` 文件或 Nginx 配置设置或其他类似的位置。不论是什么 URL，它们都会捕获每个 Web 请求并将其指向 Laravel 应用目录（app）下的 `public/index.php`。

引导应用程序

`index.php` 中并没有很多代码，里面主要包括三个方法。

第一，它加载了 Composer 的自动加载文件和 `bootstrap/cache/compiled.php` 中 Laravel 的应用编译缓存。在运行 `php artisan optimize` 的时候，就会生成这个文件。而且为了加快加载速度，该文件预加载了大多数常用的类。

Composer 和 Laravel

Laravel 的核心功能分为一系列在 `Illuminate` 命名空间下的组件，它们通过 Composer 被放到每个 Laravel 应用下。Laravel 还从 `Symfony` 和其他几个社区开发的软件包中引用了数量不少的包。这样，便可以把 Laravel 视为一个由各种组件集合起来的框架。

接下来，`index.php` 启动了 Laravel 的引导，创建了应用容器（在第 11 章中我们会深入学习），然后注册一些主要的服务（包括核，我们会在后面的内容中进一步介绍）。

最后它新建了核的实例，创建了一个代表当前用户 Web 请求的请求，然后把该请求传入核中进行处理。核通过 `Illuminate Response` 对象进行响应，然后 `index.php` 将响应返回给终端用户并结束界面请求。

Laravel 的核

核是每个 Laravel 应用的主要路由，它会得到一个用户请求并通过中间件处理请求，还会处理异常，将请求传给界面路由，然后返回最后的响应。其实在 Laravel 中有两个核，但是只有一个用于处理每个界面的请求。一个路由处理 Web 请求（HTTP 核），另一个处理控制台、cron 和 Artisan 请求（控制台核）。每个核都有一个 `handle()` 方法，它负责接收一个 `Illuminate Request` 对象，然后返回一个 `Illuminate Response` 对象。

核会运行所有需要在请求之前运行的引导程序，包括决定当前请求的运行环境（线上测试、本地、产品等），以及运行所有的服务提供商。HTTP 核还定义了包装每个请求的中间件列表，包括负责会话和 CSRF 保护的核心中间件。

服务提供者

在这些引导程序中有一些过程码，几乎所有的 Laravel 引导码都会被分成 Laravel 中的服务提供者。服务提供者是一个封装了应用程序各个部分需要运行以启动其核心功能的逻辑的类。

例如 `AuthServiceProvider`，引导了 Laravel 认证系统中所有注册必须用到的程序；`RouteServiceProvdrer`，引导启动了路由系统。

服务提供者的概念，一开始可能有点难理解。大家可以这样想，启动应用的时候需要运行包含引导码的组件。服务提供者就是把引导码与相关的类组合起来的工具。如果为了应用能够正常工作而需要提前运行任何代码，就需要用到服务提供者。

例如发现在处理的一些特征时需要用到在容器里注册的类（我们会在第 11 章中进一步学习），就可以为这个功能创建一个服务提供者。可能的代码：`GitHubServiceProvider` 或者 `MailerServiceProvider`。

服务提供者包含两个重要的方法：`boot()` 和 `register()`。这里还可能用到 `$defer` 属性。下面介绍它是怎样工作的。

首先会调用服务提供者的 `register()` 方法：因为我们希望将类和别名绑定到容器上，从而避免在 `register()` 方法中进行任何依赖于整个引导启动应用的设置。

然后，调用服务提供者的 `boot()` 方法。在里面可以进行其他所有的引导设置，比如绑定事件监听器或定义路由——任何依赖于整个引导启动 Laravel 应用的设置。

如果服务提供者只是在容器中注册绑定（也就是告诉我们怎样解决给定的类或接口），而不需要执行任何其他引导，那么可以“延长”它的注册过程，也就是说除非在容器中明确要求其中一个绑定，否则引导不会启动。这样可以加快引导应用的平均时间。

如果想要延长服务提供者的注册，首先设置一个 `protected $defer` 属性并把它设为 `true`，然后设置返回值是一个服务提供商的绑定列表的 `provides()` 方法，如示例 10-1 所示。

示例10-1 延长服务提供者的注册

```
...
class GitHubServiceProvider extends ServiceProvider
{
    protected $defer = true;

    public function provides()
```

```
{
    return [
        GitHubClient::class
    ];
}
```



服务提供者的使用群体

服务提供者还提供一套方法和配置选项，当服务提供者作为 Composer 软件包的一部分发布时，它可以向终端用户提供高级功能。如果大家想要进一步学习服务提供者的定义并且了解它是怎样工作的，可以看看 Laravel source (<http://bit.ly/2eQtW0s>) 中的服务提供者定义。

现在已经介绍了应用启动，下面看看 `Request` 对象，它是引导中最重要的输出。

Request 对象

Illuminate `Request` 请求类是 Laravel 特定的扩展（Symfony 的 `HttpFoundation\Request` 对象）。

Symfony `HttpFoundation`

如果对这些内容不熟悉，Symfony 的 `HttpFoundation` 套件几乎支持现有的每个 PHP 框架。这是 PHP 中最流行和最强大的抽象概念，用于表示 HTTP 请求、响应、表头、cookie 等。

每个 `Request` 对象都用于表示想要了解的用户 HTTP 请求的相关信息。

在本地 PHP 代码中，可能会通过 `$_SERVER`、`$_GET`、`$_POST`，以及全局变量和处理逻辑的其他组合来获取有关当前用户的请求信息。用户上传了什么文件、用户的 IP 地址是什么、用户提交了什么字段，所有这些问题都取决于语言和代码——它们都是很难被模拟的。

Symfony 的 `Request` 对象会将需要表示 HTTP 请求的所有信息存入一个单独的对象，然后通过便利的方式来获取有用的信息。Illuminate `Request` 对象还提供了更便利的方法来获取信息。



捕获请求

在 Laravel 应用中，可能不需要捕获请求。需要时，则直接从 PHP 全局变量中捕获 `Illuminate Request`，可以使用 `capture()` 方法，代码如下。

```
$request = Illuminate\Http\Request::capture();
```

在 Laravel 中获取请求对象

实际上，我们不会捕获自己的请求。Laravel 可以在引导程序中进行捕获，这里有几种方法来访问它。

我们会在第 11 章中进一步学到请求捕获——可以通过容器解析的任何构造函数或方法打开类。也就是说，可以在控制器方法或服务提供者那里打开一个类，进行请求捕获，如示例 10-2 所示。

例10-2 通过打开容器解析的方法接收Request对象

```
...
use Illuminate\Http\Request;

class PeopleController extends Controller
{
    public function index(Request $request)
    {
        $allInput = $request->all();
    }
}
```

可以使用 `request()` 全局助手在里面直接调用方法（如 `request()->input()`），以及调用助手来获得 `$request` 实例，代码如下。

```
$request = request();
$allInput = request()->all();
```

还可以使用 `app()` 全局助手来获取 `Request` 实例。传入完全限定的类名或快捷方式 `request`，代码如下。

```
$request = app(Illuminate\Http\Request::class);
$request = app('request');
```

获取请求的基本信息

现在我们已经知道了怎样获取一个请求实例，那么能用这个实例做什么呢？一个主要的目的就是用来表示当前的 HTTP 请求，所以请求类的主要功能就是能更容易地获取当前请求中的有用信息。

这里对这些方法进行了分类，但是需要注意的是，这些分类之间会有重叠，而且这样的分类方式比较随意——例如参数查询可以像在“基本用户输入”中一样简单地分为“用户和请求状态”。希望这些分类能帮助大家学习，然后就可以扔掉这些分类了。

另外在 `Request` 对象中还有很多其他的方法，上面提到的只是一些常用的方法。

基本用户输入

通过基本的用户输入方法，用户能简单地获取他们提供的信息——比如通过表格或 Ajax 组件。这里提到的“用户提供的输入”是指通过查询语句（GET）提交表达（POST）或 JSON 得到的输入，如下所示。

- `all()` 返回所有用户输入内容的数组。
- `input(fieldName)` 返回单个用户提供的输入字段的值。
- `only(fieldName|[array, of, filed, names])` 返回所有用户指定字段名的数组。
- `except(fieldName|[array, of, filed, names])` 返回所有用户（除了指定字段名）的数组
- `exists(fieldName)` 返回表示字段是否存在的布尔值。
- `has(fieldName)` 返回表示该字段是否存在于输入中并且不为空（具有值）的布尔值。
- `json()`，如果页面收到 JSON 的输入，则返回 `ParameterBag`。
- `json(keyName)` 根据键值返回页面接收的 JSON 数据的值。

ParameterBag

在 Laravel 中可能会遇到 `ParameterBag`。这个类就像一个关联数组，可以使用 `get()` 方法得到一个指定的键值，代码如下。

```
echo $bag->get('name')
```

也可以用 `has()` 方法判断键值是否存在，`all()` 用来获取所有的键和值，`count()` 用来获得条目数，`key()` 用来获得只包含键的数组。

示例 10-3 展示了从请求中使用用户提供信息的方法。

示例10-3 从请求中获取用户提供的基本信息

// 表单

```
<form method="POST" action="/form">
    {{ csrf_field() }}
```

```

    <input name="name"> Name<br>
    <input type="submit">
</form>

// 从表单获取的路由
Route::post('form', function (Request $request) {
    echo 'name is ' . $request->input('name');
    echo 'all input is ' . print_r($request->all());
    echo 'user provided email address: ' . $request->has('email') ? 'true' : 'false';
});

```

用户和请求状态

用户和请求状态方法，包括用户未通过表单明确提供的输入，如下所示。

- `method()` 返回访问路由的方法（GET、POST、PATCH 等）。
- `path()` 返回访问界面的路径（不包括域名），例如 `http://www.myapp.com/abc/def` 就会返回 `abc/def`。
- `url()` 返回访问页面的 URL（包括域名），例如 `http://www.myapp.com/abc` 会返回 `http://www.myapp.com/abc`。
- `is()` 返回表示当前页面请求是否能模糊匹配用户提供字符串的布尔值（比如 `/a/b/c` 会匹配 `$request->is('*b*')`，`*` 代表任何字符）。这里使用了 `Str::is` 中自定义的正则表达式解析。
- `ip()` 返回用户的 IP 地址。
- `header()` 返回表头的数组（比如 `['accept-language'=> ['en-US,en;q=0.8']]`），如果将表头名指定为参数就可以只返回特定的表头。
- `server()` 返回传统的、存储在 `$_SERVER`（如 `REMOTE_ADDR`）中的变量数组；如果传入一个 `$_SERVER` 变量名，则只返回变量的值。
- `secure()` 返回表示页面是否通过 HTTP 加载的布尔值。
- `pjax()` 返回表示页面请求是否通过 Pjax 加载的布尔值。
- `wantsJson()` 返回表示在请求的 `Accept` 表头中是否包含 `/json` 类型内容的布尔值。
- `isJson()` 返回表示在请求的 `Content-Type` 表头中是否包含 `/json` 类型内容的布尔值。
- `accepts()` 返回表示页面请求是否接受指定内容类型的布尔值。

文件

到目前为止，本书介绍的所有输入都是明确的（通过方法获取，`all()`，`input()` 等）：要么通过浏览器定义，要么参照网址（通过方法获取，如 `pjax()`）。文件输入与明确的

用户输入类似，但是具体的操作过程还是有很多不同的地方，如下所示。

- `file()` 返回一个所有上传文件的数组；如果传入了一个键值（上传文件的字段名）则只返回单个文件。
- `hasFile()` 返回一个表示文件是否通过指定键值上传的布尔值。

每个上传的文件都是一个 `Symfony\Component\HttpFoundation\File\UploadedFile` 实例，它提供了一套用于验证、处理和存储上传文件的工具。

关于更多处理上传文件的示例请参考第 14 章。

持久性

请求还提供了与会话交互的功能。大多数会话的功能都存在别的地方，但是有一些特殊方法是与当前界面请求相关的，如下所示。

- `flash()` 将当前请求的用户输入暂存到稍后获取的会话中。
- `flashOnly()` 为提供数组的任何键值暂存当前请求用户的输入。
- `flashExcept()` 为除提供数组外的任何键值暂存当前请求的用户输入
- `old()` 返回所有之前暂存的用户输入的数组；如果传入了一个键值，则返回暂存数据里对应该键值的值。
- `flush()` 删除所有之前暂存的用户输入。
- `cookie()` 从请求中获取所有的 `cookie`；如果指定了键值，则只获取对应的 `cookie`。
- `hasCookie()` 返回表示请求在给定键值时是否包含 `cookie` 的布尔值。

`flash*()` 和 `old()` 方法是用来存储用户输入和稍后获取用户输入的，其通常位于通过验证和拒绝输入后。

Response 对象

与 `Request` 对象类似，`Illuminate Response` 对象代表了应用传递给终端用户的响应，包括表头、`cookie`、`content`，以及其他任何用于加载页面时发送给终端用户的浏览器指令的内容。

就像 `Request` 一样，`Illuminate\Http\Response` 对象继承了 `Symfony` 的 `Symfony\Component\HttpFoundation\Response` 类。这是一个基本类，它包括一系列属性，以及表示和渲染响应的方法；`Illuminate Response` 类还可以通过一些有用的快捷方式对它进行装饰。

在容器中使用和创建 Response 类

在介绍怎样自定义响应对象之前，让我们先来看看最常用的 Response 对象。

最后，任何从路由定义返回的 Response 对象都会被转换成一个 HTTP 响应。它可能会定义指定的表头或 content、设置 cookie，或者进行其他相关的设置，但是无论如何，最终它会被转换成浏览器能解析的响应。

先来看一个简单的响应，如示例 10-4 所示。

示例10-4 最简单的HTTP响应

```
Route::get('route', function () {
    return new Illuminate\Http\Response('Hello!');
});

// 类似地，使用全局函数
Route::get('route', function () {
    return response('Hello!');
});
```

我们创建了一个响应，并向它提供了核心数据，然后返回这个 Response 对象。同时还可以自定义 HTTP 状态、表头、cookie 或其他内容，如示例 10-5 所示。

示例10-5 包含自定义状态和表头的简单HTTP响应

```
Route::get('route', function () {
    return response('Error!', 400)
        ->header('X-Header-Name', 'header-value')
        ->cookie('cookie-name', 'cookie-value');
});
```

设置表头

我们已经通过 header() 在响应中定义了表头，如示例 10-5 所示。第一个参数是表头名，第二个参数是对应的值。

添加 cookie

我们还可以直接在 Response 对象中添加 cookie。在第 14 章中会进一步介绍 Laravel 的 cookie，先通过示例 10-6 来看看如何在响应中添加 cookie 的简单应用。

示例10-6 为响应添加cookie

```
return response($content)
    ->cookie('signup_dismissed', true);
```

特殊的响应类型

Laravel 中还有一些用于视图、下载、文件和 JSON 的特殊响应类型。它们每一个都是预定义的宏，所以我们能很容易将这些模板重新用于表头或 content 的结构。

视图响应

在第 4 章中使用了 `view()` 全局助手来显示怎样返回一个模板，例如 `view(view.name, here)` 或其他类似的方法。但是如果希望在返回视图时能够自定义表头、HTTP 状态等，则就可以像示例 10-7 这样使用 `view()` 响应类型。

示例10-7 使用`view()`响应类型

```
Route::get('/', function (XmlGetterService $xml) {
    $data = $xml->get();
    return response()
        ->view('xml-structure', $data)
        ->header('Content-Type', 'text/xml');
});
```

下载响应

有时候，我们希望应用能强制用户浏览器下载某个文件，不管是在 Laravel 里创建还是从数据库或受保护的位置中获取。`download()` 响应类型就能简单地实现这个功能。

第一个参数是必选的，它指定了存储该文件的路径。如果它是一个生成文件，则需要先将它存到一个暂定的位置。

第二个参数是可选的，也就是下载文件的名称（如 *export.csv*）。如果没有指定字符串，它就会自动生成一个名字。第三个可选参数，允许传入一个表头的数组。示例 10-8 详细展示了 `download()` 响应类型的使用方式。

示例10-8 使用`download()`响应类型

```
public function export()
{
    return response()
        ->download('file.csv', 'export.csv', ['header' => 'value']);
}

public function otherExport()
{
    return response()->download('file.pdf');
}
```

文件响应

文件响应与下载响应类似，其不同之处在于文件响应允许浏览器显示文件而不是强制下载文件。文件响应最常用于图片和 PDF 的显示。

第一个参数是必选的，第二个参数是可选的，可以是一个表头的数组，如示例 10-9 所示。

示例10-9 使用file()响应类型

```
public function invoice($id)
{
    return response()->file("./invoices/{$id}.pdf", ['header' => 'value']);
}
```

JSON 响应

JSON 响应非常常见，尽管对程序来说并不是特别复杂，但是其同样包括自定义响应。

JSON 响应将传入的数组转换成 JSON（通过 `json_encode()`），并将 `Content-Type` 设置为 `application/json`。同时还可以使用 `setCallback()` 方法来创建一个 JSONP 响应，而不是 JSON 响应，如示例 10-10 所示。

示例10-10 使用json()响应类型

```
public function contacts()
{
    return response()->json(Contact::all());
}

public function jsonpContacts(Request $request)
{
    return response()
        ->json(Contact::all())
        ->setCallback($request->input('callback'));
}

public function nonEloquentContacts()
{
    return response()->json(['Tom', 'Jerry']);
}
```

重定向响应

在 `response()` 助手中，重定向不常被调用，所以它和其他一些自定义响应类型不太一样。但是它只是一个不同的响应而已。重定向响应从 Laravel 路由中返回，然后将用户重定向（一般是 301）到另一个页面或返回前一个页面。

从技术上来说，可以从 `response()` 中调用一个重定向，就像在 `response()->redirectTo('/')` 中返回一样。但更为常用的是使用重定向特定的全局助手。

`redirect()` 全局方法用于创建重定向响应，`back()` 全局方法是 `redirect()->back()` 的快捷方式。

与大多数全局助手一样，`redirect()` 全局方法既可以传入参数，也可以用来获取类的实例，然后将方法调用连接到该实例。如果没有连接，只是传入了参数，那么 `redirect()` 实现的功能与 `redirect()->to()` 一样：获取一个字符串 URL，然后将它进行重定向。示例 10-11 展示了一些使用示例。

示例10-11 使用`redirect()`全局助手示例

```
return redirect('account/payment');
return redirect()->to('account/payment');
return redirect()->route('account.payment');
return redirect()->action('AccountController@showPayment');

// 如果命名路由或控制器需要参数
return redirect()->route('contacts.edit', ['id' => 15]);
return redirect()->action('ContactsController@edit', ['id' => 15]);
```

也可以“往回”重定向到前一个界面，这在处理和验证用户输入时非常有用。示例 10-12 展示了在验证内容中的常见模式。

示例10-12 通过输入往回重定向

```
public function store()
{
    // 如果验证失败 ...
    return back()->withInput();
}
```

最后还可以同时将暂存数据重定向到会话中。这在错误和成功的信息中很常见，如示例 10-13 所示。

示例10-13 根据暂存数据重定向

```
Route::post('contacts', function () {
    // 存储联系人

    return redirect('dashboard')->with('message', 'Contact created!');
});

Route::get('dashboard', function () {
```

```
// 从会话获取闪存的数据——通常在 Blade 模板中处理
echo session('message');
});
```

自定义响应宏

也可以通过“宏”创建自定义的响应类型，这允许定义对响应及其提供的内容做出一系列修改。

接下来重新创建 `json()` 自定义响应类型，来看看它是怎样工作的。通常来说，对于这些绑定需要创建自定义的服务提供商，但是在这里只是把它放到了 `AppServiceProvider` 中，如示例 10-14 所示。

示例10-14 创建自定义响应宏

```
...
class AppServiceProvider
{
    public function boot()
    {
        Response::macro('myJson', function ($content) {
            return response(json_encode($content))
                ->headers(['Content-Type'=>'application/json']);
        });
    }
}
```

然后可以像在预定义的 JSON 宏中一样使用，代码如下。

```
return response()->myJson(['name' => 'Sangeetha']);
```

这将返回一个正文（以 JSON 的格式进行编码的数组）的响应，并带有适合 JSON 的 `Content-Type` 头。

Laravel 和中间件

回顾一下本章开始的图 10-1。

我们已经介绍了图中的请求和响应，但是还没有讲到中间件。它并不是 Laravel 中独有的特性，而是一个广泛使用的架构模式。

中间件入门

中间件是围绕在应用中的一系列层次，就像多层蛋糕或者洋葱。如示例 10-1 所示，每个请求通过中间件传递到应用中，然后响应也通过中间件传回终端用户。

通常认为中间件与应用程序的逻辑分离，并且在理论上适用于任何应用程序，而不仅仅是正在开发的应用程序。

中间件可以审查并装饰请求，或者拒绝它，这取决于中间件在请求中发现的内容。也就是说，中间件对于速率限制是非常有用的，它可以审查 IP 地址、检查在过去一分钟内访问资源的次数以及当阈值被超过时，返回 429 状态码（请求次数过多）。

因为中间件也可以访问应用的响应，这对于装饰响应非常有用。例如 Laravel 利用中间件从给定的请求 / 响应周期中，在响应送回给终端用户前将所有队列中的 cookie 添加到响应中。

但是中间件最强大的用途来自于它可能是与请求 / 响应周期交互的第一个和最后一个组成部分。这对于类似于启动会话的操作非常有用——在 PHP 中，如果需要很早打开会话，而很晚关闭它时，中间件就能提供很大的帮助。

创建自定义中间件

如果希望在中间件拒绝了每一个使用 DELETE HTTP 方法的请求同时返回 cookie，那么可以使用 Artisan 命令来创建中间件，代码如下。

```
php artisan make:middleware BanDeleteMethod
```

现在可以打开文件 `app/Http/Middleware/BanDeleteMethod.php`，默认内容如示例 10-15 所示。

示例10-15 默认的中件内容

```
...
class BanDeleteMethod
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

`handle()` 方法代表输入请求和输出请求的过程，这是理解中间件的困难之处。下面一起来研究这个问题。

理解中间件的 `handle()` 方法

首先中间件比其他的层都要高，最后位于应用的最高层。第一个注册的中间件可以第一

个访问传入的请求，然后请求继续传入其他的中间件，最后到达应用程序；接着在响应传回的时候也要经过每一个中间件，第一个注册的中间件最后一个访问响应。

假设已经注册了 `BanDeleteMethod` 作为第一个中间件，也就是说进入它的 `$request` 是原始请求，没有经过其他任何中间件的处理。

接下来将请求传入 `$next()`，代表将这个请求传入其他的中间件进行处理，`$next()` 闭包只接收 `$request`，然后将它传入栈中下一个中间件的 `handle()` 方法。再重复这个过程，直到栈中没有更多中间件，最后该请求就可以到达应用程序了。

那响应又是怎样的流程呢？应用返回一个响应，它会被传入中间件的链中——因为每个中间件都会返回它的响应。所以在同样的 `handle()` 方法中，中间件可以装饰 `$request` 并将它传到 `$next()` 闭包中，然后在将输出内容返回给终端用户之前对输出进行操作。具体如示例 10-16 所示。

示例10-16 中间件调用过程伪代码

```
...
class BanDeleteMethod
{
    public function handle($request, Closure $next)
    {
        // 此时 $request 是来自用户的原始请求
        // 进行一些处理
        if ($request->ip() === '192.168.1.1') {
            return response('BANNED IP ADDRESS!', 403);
        }

        // 决定接受请求，来把它传给栈中下一个中间件
        // 把它传给 $next() 闭包，返回的是这个请求经过栈中的每一个中间件
        // 到达应用之后，应用返回栈中的响应
        $response = $next($request);
        // 在响应返回给用户前，可以再次与它进行交互
        $response->cookie('visited-our-site', true);

        // 最后将该响应返回给终端用户
        return $response;
    }
}
```

最后让中间件来实现之前提到的，希望它实现的功能，如示例 10-17 所示。

示例10-17 中间件禁止delete方法

```
...
```

```

class BanDeleteMethod
{
    public function handle($request, Closure $next)
    {
        // 测试 DELETE 方法
        if ($request->method() === 'DELETE') {
            return response(
                "Get out of here with that delete method",
                405
            );
        }

        $response = $next($request);

        // 指定 cookie
        $response->cookie('visited-our-site', true);

        // 返回响应
        return $response;
    }
}

```

绑定中间件

工作还没有结束，还需要注册这个中间件全局或指定路由。

全局中间件作用于每一个路由；路由中间件则基于路由 - 路由的标准。

绑定全局中间件

两种绑定都发生在 *app/Http/Kernel.php* 中，将中间件的类名添加到 `$middleware` 属性中就可以将它添加为全局中间件，如示例 10-18 所示。

示例10-18 绑定全局中间件

```

// app/Http/Kernel.php
Protected $middleware =[
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\BanDeleteMethod::class,
];

```

绑定路由中间件

面向路由的中间件可以被添加为路由中间件或者作为中间件组的其中一部分。先来看看前一种。

路由中间件被添加到 `app/Http/Kernel.php` 中的 `$routeMiddleware` 数组中。这与添加到 `$middleware` 中类似，其不同之处在于需要为每一个中间件指定一个键值，这个键值用来将中间件应用到指定路由，如示例 10-19 所示。

示例10-19 绑定路由中间件

```
// app/Http/Kernel.php
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    ...
    'nodelete' => \App\Http\Middleware\BanDeleteMethod::class,
];
```

现在可以在路由定义中使用这个中间件了，如示例 10-20 所示。

示例10-20 在路由定义中使用路由中间件

```
// 这个例子不具有实际意义
Route::get('contacts', [
    'middleware' => 'nodelete',
    'uses' => 'ContactsController@index'
]);

// 具有实际意义
Route::group(['prefix' => 'api', 'middleware' => 'nodelete', function () {
    // 所有路由与一个 API 关联
}]);
```

使用中间件组

5.2 Laravel 5.2 引入了中间件组的概念。它们基本上是预包装的中间件包，在特定的情况下是有意义的。



Laravel 5.2 中的中间件组

在 Laravel 5.2 的早期版本中，默认的路由文件是 `routes.php`。它包括三个不同的部分：根路由 (`/`)，不在任何中间件组下；另外还包括 `web` 和 `api` 中间件组。这对于新用户来说可能比较困惑，根路由没有权限访问任何会话或在中间件中启用其他功能。

在 Laravel 5.2 后面发布的版本中，这些功能都被简化了，`routes.php` 中的每个路由都在 `web` 中间件组中。在 Laravel 5.3 版本中，`web` 路由都在 `routes/web.php` 文件中，API 路由都在 `routes/api.php` 文件中。接下来介绍如何在其他组中添加路由。

Laravel 自带两个组：`web`和`api`。`web`包括几乎每一个 Laravel 页面请求的中间件，包括用于 cookie、会话、CSRF 保护等的中间件。`api`中不包括任何 `web` 中间件——它只包含一个 `throttle` 和路由模型绑定中间件。这些都在 `app/Http/Kernel.php` 中被定义。

可以像将路由中间件应用到路由中一样，通过 `middleware()` 流畅方法将中间件组应用到路由上，代码如下。

```
Route::get('/', 'HomeController@index')->middleware('web');
```

也可以创建自己的中间件组，然后将它们添加到已经存在的中间件组中或从之前的中间件组中移除自定义的组。这跟普通的添加中间件类似，其不同之处在于需要将它们添加到 `$middlewareGroups` 数组的键组中。

这些中间件组怎样与两个默认的路由文件匹配？`routes/web.php` 文件被 `web` 中间件组包装，并且 `routes/api.php` 文件被 `api` 中间件组包装。

`routes/*` 文件在 `RouteServiceProvider` 中被加载。接下来介绍 `map()` 方法，如示例 10-21 所示，里面包含 `mapWebRoutes()` 和 `mapApiRoutes()` 方法。它们分别加载已经包装了合适的中间件组的文件。

示例10-21 Laravel 5.3中默认的路由服务提供者

```
// App\Providers\RouteServiceProvider
public function map()
{
    $this->mapApiRoutes();
    $this->mapWebRoutes();
}

protected function mapApiRoutes()
{
    Route::group([
        'middleware' => 'api',
        'namespace' => $this->namespace,
        'prefix' => 'api',
    ], function ($router) {
        require base_path('routes/api.php');
    });
}

protected function mapWebRoutes()
{
    Route::group([
        'middleware' => 'web',
```

```

        'namespace' => $this->namespace,
    ], function ($router) {
        require base_path('routes/web.php');
    });
}

```

在示例 10-21 中，可以看到用路由 web 中间件组和另一个在 api 中间件组下的中间件组来加载在默认命名空间 (App\Http\Controllers) 下的路由组。

向中间件传参

这虽然并不常见，但是有的时候还是需要向路由中间件传参。例如对于是否保护 member 用户类型或者 owner 用户类型，可能有的认证中间件需要有不同的功能，代码如下。

```

Route::get('company', function () {
    return view('company.admin');
})->middleware('auth:owner');

```

为了能够正常工作，需要为中间件的 handle() 方法提供一个或多个参数，然后根据情况更新方法逻辑，代码如下。

```

public function handle($request, $next, $role)
{
    if (auth()->check() && auth()->user()->hasRole($role)) {
        return $next($request);
    }

    return redirect('login');
}

```

另外，也可以为 handle() 方法提供不止一个参数，这里可以用逗号分隔多个参数。

```

Route::get('company', function () {
    return view('company.admin');
})->middleware('auth:owner,view');

```

构造 Request 对象

本章介绍了怎样注入一个 Illuminate Request 对象，这是基本的、常见的请求对象。

也可以继承 `Request` 对象，然后注入它。在第 11 章中，我们会进一步学习怎样绑定以及注入自定义的类，而在这里有一个特殊的类型叫作表单请求，它有自己独有的一套行为。

在第 103 页“表单请求”中，具体介绍了怎样创建及使用表单请求。

测试

作为一个开发人员，在测试程序的时候也会用到请求、响应和中间件。除此之外，Laravel 本身也经常使用这些功能。

在应用测试中——如 `$this->visit('/')` 调用、点击等，可以指示 Laravel 的应用测试框架来生成代表交互作用的请求对象。然后当这些对象真正进行访问时，它们就会被传到应用中。这也是应用测试这么准确的原因：应用并不“知道”与它进行交互的不是真正的用户。

在这种情况下，我们可以做出很多断言，比如 `assertResponseOk()`，就是对应用程序测试框架生成的响应对象的断言。`assertResponseOk()` 方法仅查看响应对象，然后断言它的 `isOk()` 方法返回 `true`，也就是状态码为 200。最后所有应用测试中的过程和行为都是在模拟真实的界面请求。

在测试中需要一个请求的时候，可以用 `$request=request()` 从容器中获取，也可以创建自己的请求——使用 `Request` 类的构造器参数，这些参数都是可选的，如下所示。

```
$request = new Illuminate\Http\Request(
    $query,          // 获取数组
    $request,        // 提交数组
    $attributes,     // "attributes" 数组可以为空
    $cookies,        // cookie 数组
    $files,          // file 数组
    $server,         // server 数组
    $content         // 原始数据
);
```

如果想看具体的示例，那么可以参考 Symfony 用来创建新请求的方法（该方法在全局 PHP 中提供）：`Symfony\Component\HttpFoundation\Request@createFromGlobals()`。

手动创建响应更简单，可以根据需要选择一些参数，代码如下。

```
$response = new Illuminate\Http\Response(  
    $content, // 响应内容  
    $status, // HTTP 状态默认是 200  
    $headers // 数组的表头数组  
);
```

最后，如果想要在系统测试中停用中间件，只需要将 `WithoutMiddleware` 特性导入测试。

本章小结

所有进入 Laravel 应用的请求都会被转换成 `Illuminate Request` 对象，然后这个对象会经过所有的中间件，最后的应用中处理。应用生成一个 `Response` 对象，这个对象经过所有的中间件（与请求的传输方向相反），最后到达终端用户。

请求和响应对象负责封装和表示与传入用户请求和传出服务器响应相关的信息。

服务提供者用于为应用程序收集和绑定与注册类相关的行为。

中间件用于包裹应用程序，可以拒绝或装饰任何请求和响应。

Laravel 的服务容器及依赖注入容器几乎是其所有功能的核心。容器是一个简单的工具，使用它可以绑定及解析类与接口的具体实例，同时，它还是一个强大且细致的管理器，可以用于管理相互依赖的网络。在这一章中，我们将详细地介绍容器，介绍它是如何工作的，以及如何使用它。



命名与容器

可以发现，在这本书以及其他文档或教育资源中，容器有相当多的命名方式，具体如下。

- 应用程序容器
- IoC（控制反转）容器
- 服务容器
- DI（依赖注入）容器

以上名称都是有用且有效的，只要知道这些名词都表示同样的事物即可，它们都指服务容器。

依赖注入简介

依赖注入是指，每个类的依赖项都从外部注入，而不是在一个类中被实例化（“更新”）。最常见的情况是构造函数注入，具体是指对象的依赖项在创建时被注入。也有 setter 注入，其中类暴露了一种专门用于注入给定依赖关系的方法，其中的一个或多个方法期望在调用它们的依赖关系时被注入。下面来看一个构造函数注入的简单例子，示例 11-1 是依赖注入中最常见的类型。

示例11-1 基本的依赖注入

<?php

```
class UserMailer
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function welcome($user)
    {
        return $this->mailer->mail($user->email, 'Welcome!');
    }
}
```

如你所见，UserMailer 类期望在实例化时注入 Mailer 类型的对象，随后引用该实例。

依赖注入的主要好处是，我们能够自由地改变注入的内容，模拟依赖性测试，并且仅实现一次共享的依赖关系。

控制反转

大家也许听说过“控制反转”与“依赖注入”配合使用，有时 Laravel 的容器也被称为 IoC 容器。

这两个概念非常相似。控制反转指的是，在传统编程中，最低级别的代码（特定的类、实例和过程代码）“控制”要使用的特定模式或接口实例的程序代码。例如，如果要在每个类中实例化邮件程序，则每个类都可以决定是否使用 Mailgun、Mandrill 或 Sendgrid。

控制反转的思想是，翻转“控制”，使其在应用程序的另一端。这样一来，使用哪种邮件程序取决于应用程序的最高、最抽象级别是什么，通常在配置中可以查看。每一个实例，每一段低级别的代码，都可以“请求”高级配置：“能给我一封邮件吗？”但它们“不知道”自己正在获取哪封邮件。

依赖注入，尤其是 DI 容器，它为控制反转提供了极好的机会——可以定义 Mailer 接口的具体实例，例如，何时将邮件注入需要它们的类中。

依赖注入和 Laravel

如示例 11-1 所示, 依赖注入中最常见的模式是构造函数注入, 或者当它被实例化(“构建”)时注入对象的依赖关系。

以示例 11-1 中的 Usermailer 类为例, 示例 11-2 显示了创建和使用该类的实例的具体方法。

示例11-2 简单的手动依赖注入

```
$mailer = new MailgunMailer($mailgunKey, $mailgunSecret, $mailgunOptions);  
$userMailer = new UserMailer($mailer);  
  
$userMailer->welcome($user);
```

想象一下, 我们希望 Usermailer 类能够记录信息, 同时每次发送消息时都能向 Slack 通道发送通知。如示例 11-3 所示, 如果每次创建一个新实例时都必须做这些工作, 尤其是必须从某个地方获取所有参数, 那么代码就会变得非常复杂。

示例11-3 更复杂的手动依赖注入

```
$mailer = new MailgunMailer($mailgunKey, $mailgunSecret, $mailgunOptions);  
$logger = new Logger($logPath, $minimumLogLevel);  
$slack = new Slack($slackKey, $slackSecret, $channelName, $channelIcon);  
$userMailer = new UserMailer($mailer, $logger, $slack);  
  
$userMailer->welcome($user);
```

如果每次想要实现一个 Usermailer 类时都必须重写这段代码, 那么即使依赖注入的效果是很好的, 这种方式也很糟糕。

app() 全局助手

在深入研究容器的实际工作方式之前, 先以最简单的方式快速从容器中取出一个对象: 使用 app() 助手。

将任意字符串传递给该助手, 无论是一个完全限定类名 (FQCN) 还是 Laravel 快捷方式 (下文详细介绍), 都将返回该类的实例。

```
$logger = app(Logger::class);
```

这绝对是与容器进行交互的最简单的方式。该方式创建类的实例并返回, 简单且效果显著。

创建具体实例的不同语法

创建一个具体实例的最简单的方法是使用全局助手并通过 `app('FQNC')` 直接将类或接口名称传递给辅助程序。

但是，只要有容器的实例，无论它是在某个地方注入，还是在服务提供者中使用 `$this->app` 获得，或是通过运行 `$container = app()` 方法得到(鲜为人知的方法)，就有若干种方法可以从中生成实例。

最常见的方法是运行 `make()` 方法，`$app->make('FQCN')` 的效果更好，有时也可以看到其他开发人员或文档中使用 `$app['FQCN']` 语法。不必担心，这些方法实现的功能相同，只是写法不同罢了。

创建 `Logger` 实例看起来很简单，但你可能已经注意到，在示例 11-3 中，`$logger` 类有两个参数：`$logPath` 和 `$minimumLogLevel`。容器如何知道此处该怎样操作呢？

简短的回答：它不知道。我们可以使用 `app()` 全局助手创建一个在其构造函数中没有参数的实例，然后运行 `new Logger`。当构造函数具有一些复杂特性时，我们就要查看容器究竟如何通过构造函数来构造类了。

容器如何连接

在深入学习 `Logger` 类之前，先来看一看示例 11-4。

示例11-4 Laravel自动装配

```
class Bar
{
    public function __construct() {}
}

class Baz
{
    public function __construct() {}
}

class Foo
{
    public function __construct(Bar $bar, Baz $baz) {}
}
```

```
$foo = app(Foo::class);
```

以上示例与示例 11-3 中邮件的例子类似。不同的是，这些依赖关系（`Bar` 和 `Baz`）都非常简单，容器不需要任何进一步的信息就可以处理它们。容器在构造函数中读取类型提示（`typehint`），解析每个实例，然后注入新的 `Foo` 实例。这就是所谓的自动装配：基于类型提示来解析实例，无须显式绑定容器中的类。



PHP 的类型提示（`typehint`）

PHP 中的类型提示（`typehint`）是指在方法签名中的变量前加一个类或接口的名称。

```
public function __construct(Logger $logger) {}
```

类型提示（`typehint`）可以通知 PHP，无论传递到何处，方法的类型必须是 `Logger`，它可以是一个接口或一个类。

自动装配表示，如果一个类在没有显式绑定到容器（如 `Foo`、`Bar`、`Baz`）的情况下，容器依然可以找出解决它的方法，那么就意味着容器可以解决它。任何理论上可以由容器解决的没有构造函数依赖关系的类（如 `Bar` 和 `Baz`）以及具有构造函数依赖关系的类（如 `Foo`），都可以在容器中解决。

我们只需要绑定含无法解析的构造函数参数的类——例如，示例 11-3 中的 `$logger` 类，其中包含与日志路径和日志级别相关的参数。

我们需要学习如何明确地将这些类与参数绑定到容器中。

将类绑定到容器

将类绑定到 Laravel 的容器中，实质上就是告诉容器，如果开发人员请求一个 `Logger` 实例，便需要实例化一段具有正确参数和依赖关系的代码，然后正确返回。

我们正在试图实现：当有人访问这个特定的字符串（通常是一个类的 FQCN）时，容器应该用上述方法解决问题。

绑定到闭包

先来看一下如何绑定到容器。注意，绑定到容器适合在服务提供者的 `register()` 方法（见示例 11-5）中进行。

示例11-5 基本容器绑定

```
// 在服务提供者中
public function register()
{
    $this->app->bind(Logger::class, function ($app) {
        return new Logger('\log\path\here', 'error');
    });
}
```

在这个例子中，有几个重要的点需要注意。首先，我们正在运行的 `$this->app->bind()` 方法是每个服务提供者都可以使用的容器实例。容器的 `bind()` 方法是用来绑定容器的。

`bind()` 方法的第一参数是绑定过程中所需的“关键”，在这里，我们使用该类的 FQCN。第二个参数根据所进行操作的不同而有所不同，但本质上都是显示容器如何处理上述的绑定关键的实例。

在这个例子中，我们传递了一个闭包。每当有人运行 `app(Logger::class)` 时，就会得到这个闭包的结果。闭包会通过容器本身的一个实例 (`$app`)，因此，如果正在处理的类中存在依赖关系，则当我们希望从容器中解析该依赖时，便可以在定义中使用它。

```
$this->app->bind(UserMailer::class, function ($app) {
    return new UserMailer(
        $app->make(Mailer::class),
        $app->make(Logger::class),
        $app->make(Slack::class)
    );
});
```

注意，每次请求类的新实例时，此闭包将再次运行，返回新的输出。

绑定单例模式、别名和实例

将绑定闭包的输出缓存，以便每次请求实例时该闭包无须重新运行，这便是单例模式。可以运行 `$this->app->singleton()` 来实现。

```
public function register()
{
    $this->app->singleton(Logger::class, function () {
        return new Logger('\log\path\here', 'error');
    });
}
```

如果已经有一个要返回的对象的实例，也可以得到类似的结果。

```
public function register()
{
    $logger = new Logger('\log\path\here', 'error');
    $this->app->instance(Logger::class, $logger);
}
```

最后，如果要将一个类命名为另一个类，可以将类绑定到快捷方式，或将快捷方式绑定到类，只需传递两个字符串即可。

```
$this->bind(Logger::class, FirstLogger::class);
// 或者
$this->bind('log', FirstLogger::class);
// 或者
$this->bind(FirstLogger::class, 'log');
```

请注意，这些快捷方式在 Laravel 中是非常常见的，Laravel 提供了一个快捷方式系统，该系统提供核心功能类，使用易于记忆的键，如 log。

将具体实例绑定到接口

如同可以将一个类绑定到另一个类或快捷方式一样，也可以将其绑定到一个接口。这个功能非常强大，有了这个功能，我们就可以指定接口而非类的名称了，如示例 11-6 所示。

示例11-6 指定并绑定到接口

```
...
use Interfaces\Mailer;

class UserMailer
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }
}
// 服务提供者
public function register()
{
    $this->app->bind(\Interfaces\Mailer::class, function () {
        return new MailgunMailer(...);
    });
}
```

现在可以在代码中打开 **Mailer** 或 **Logger** 接口，然后在服务提供者中选择一个要使用的特定邮件程序或日志记录器，这些程序可以随处使用，这就是控制反转。

语境绑定

有时需要根据上下文来更改解析接口的方法。例如，我们可能想将事件从某个地方记录到本地的系统日志和外部服务中，这时需要告诉容器区别对待，如示例 11-7 所示。

示例11-7 语境绑定

```
// 在服务提供者中
public function register()
{
    $this->app->when(FileWrangler::class)
        ->needs(Interfaces\Logger::class)
        ->give(Loggers\Syslog::class);

    $this->app->when(Jobs\SendWelcomeEmail::class)
        ->needs(Interfaces\Logger::class)
        ->give(Loggers\PaperTrail::class);
}
```

构造器注入

我们已经介绍了构造函数注入的概念，并且了解了容器是如何轻松解析类或接口的。我们也了解了使用 `app()` 全局助手来创建实例是多么容易，以及容器在创建类时将如何解析类的构造函数依赖关系。

我们还没有介绍容器如何解析应用程序中的许多核心操作类。例如，每个控制器都由容器实例化。这意味着如果想要在控制器中使用日志记录器实例，可以简单地在控制器构造函数中指定日志记录器类。当 Laravel 创建控制器时，它将日志记录器从容器中解析出来，这样日志记录器实例便可用于控制器中，如示例 11-8 所示。

示例11-8 将依赖关系注入控制器

```
...
class MyController extends Controller
{
    protected $logger;

    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
    }
}
```

```

    }

    public function index()
    {
        // 完成某些功能
        $this->logger->error('Something happened');
    }
}

```

容器负责解析控制器、中间件、队列工作、事件监听器，以及在应用程序的生命周期中由 Laravel 自动生成的其他类——所以任何类都可以在其构造函数中键入依赖关系，并期望它们能够自动注入。

方法注入

在应用程序的某些地方，Laravel 不只读取构造函数签名，它还读取方法签名，并注入依赖关系。

最常见的方法注入是控制器方法注入。如果有依赖关系，只需要使用单个控制器方法，就可以将依赖注入该方法，如示例 11-9 所示。

示例11-9 将依赖关系注入单个控制器方法

```

...
class MyController extends Controller
{
    // 方法依赖关系可以在路由参数前后注入
    public function show(Logger $logger, $id)
    {
        // 完成某些功能
        $logger->error('Something happened');
    }
}

```

以上方法可以在服务提供者的 `boot()` 方法中使用，也可以使用容器任意调用其他类的方法，为该方法注入其他方法（见示例 11-10）。

示例11-10 使用容器的`call()`方法手动调用类的方法

```

class Foo
{
    public function bar($parameter1) {}
}

```

```
$foo = new Foo;
```

```
// 在 $foo 中利用 "value" 的第一个参数来调用 "bar" 方法  
app()->call($foo, 'bar', ['parameter1' => 'value']);
```

facade 与容器

到目前为止，本书中已经介绍了许多与 facade 相关的内容，但还没有介绍它是如何工作的。

Laravel 中的 facade 是专门用于简单访问 Laravel 的核心功能件的类。facade 有两个特征：首先，它在全局命名空间中有效（\Log 是 \Illuminate\Support\Facades\Log 的别名）；其次，它使用静态方法访问非静态资源。

我们已经在本章中了解了日志记录，下面来看 Log facade。在控制器或视图中，可以使用以下调用。

```
Log::alert('Something has gone wrong!');
```

以下是在没有 facade 的情况下进行相同调用的代码。

```
$logger = app('log');  
$logger->alert('Something has gone wrong!');
```

facade 将静态调用（对类本身进行的方法调用，使用 :: 而不是实例）转换为实例上的常规方法调用。



导入 facade 命名空间

如果是命名空间的类，需要确保在顶部导入 facade。

```
...  
use Illuminate\Support\Facades\Log;  
  
class Controller extends Controller  
{  
    public function index()  
    {  
        // ...  
        Log::error('Something went wrong!');  
    }  
}
```

facade 如何工作

下面来看一下 Log facade 是如何工作的。

首先，打开类 `Illuminate\Support\Facades\Log`，可以看到如示例 11-11 所示的内容。

示例11-11 Log facade类

```
<?php

namespace Illuminate\Support\Facades;

class Log extends facade
{
    protected static function getFacadeAccessor()
    {
        return 'log';
    }
}
```

每个 facade 中都有一个单一的方法 `getFacadeAccessor()`。该方法明确了 Laravel 应该从容器中查找这个 facade 的后台实例的关键。

在这个例子中，我们可以看到，每一次对 `Log` facade 的调用都被代理为从容器重调用 `log` 快捷方式的实例。当然，这不是一个真正的类或接口的名称，这是前面提到的快捷方式之一。

以下代码是一个真实的例子。

```
Log::error('Help!');

// 和……一样

app('log')->error('Help!');
```

有多种方法可以用于查看每个 facade 访问者究竟指向什么类，其中检查文件是最简单的。在 facade 文档页面 (<https://laravel.com/docs/facades>) 上有一个表格，其中显示了每个 facade 的容器绑定（快捷方式，如 `log`）以及返回的类，如下所示。

| facade | 类 | 服务容器绑定 |
|--------|-----------------------------------|--------|
| App | Illuminate\Foundation\Application | app |
| ... | ... | ... |
| Log | Illuminate\Log\Writer | log |

有了上面的参考，我们可以做以下三件事。

第一，可以随时找出 facade 上面的可用方法。只需找到方法的支持类并查看该类的定义就可以知道，所有公共方法都可以在 facade 上调用。

第二，可以了解如何使用依赖注入来注入 facade 的支持类。如果想要在保留 facade 功能的同时又使用依赖注入，只需键入 facade 的支持类，或使用 `app()` 得到一个实例，并调用与 facade 相同的调用方法。

第三，可以学会如何创建自己的 facade。为 facade 创建一个类，扩展 `Illuminate\Support\Facades\Facade`，并且提供 `getFacadeAccessor()` 方法，返回一个字符串。该字符串可用于从容器中解析支持类——也许得到的只是该类的 FQCN。最后，通过 `config/app.php` 将它添加到 `aliases` 数组中来注册 facade。完成了！这样我们就实现了自己的 facade。

服务提供者

在前一章中，我们已经介绍过服务提供者的相关内容。关于容器，最重要的是，一定要在某些服务提供者提供的 `register()` 方法中进行注册绑定。

可以将松散绑定转储到包罗万象的 `App\Providers\AppServiceProvider` 中。但一般来说，为正在开发的每一组功能创建一个单独的服务提供者，并将其绑定到 `register()` 方法中，这是更好的做法。

测试

使用控制反转和依赖注入使 Laravel 的测试变得非常灵活。例如，我们可以绑定不同的日志记录器，具体取决于应用程序处于工作状态还是测试状态。或者为了方便查询，我们可以将 Mailgun 的交易电子邮件服务更改为本地电子邮件日志记录器。这两种交换实际上是非常普遍的，这样做更容易使用 Laravel 的 `.env` 配置文件，也可以使用任何其他接口或类进行类似的交换。

当需要返回时，最简单的办法是在测试中显示重新绑定的类和接口。具体做法如示例 11-12 所示。

示例11-12 在测试中重写绑定

```
public function test_it_does_something()
{
    app()->bind(Interfaces\Logger, function () {
        return new DevNullLogger;
    });

    // 完成某些功能
}
```

如果需要在测试的全局范围内返回某些类或接口（这种情况不是特别常见），可以在测试类的 `setUp()` 方法，以及 Laravel 的 `TestCase` 基础测试中的 `setUp()` 方法中执行以下操作，如示例 11-13 所示。

示例11-13 覆盖所有测试的绑定

```
class TestCase extends \Illuminate\Foundation\Testing\TestCase
{
    public function setUp()
    {
        parent::setUp();

        app()->bind('whatever', 'whatever else');
    }
}
```

当使用如 `Mockery` 一类的对象时，通常会创建一个类的 `mock`、`spy` 或 `stub`，然后将其重新绑定到容器中来替代其指示对象。

本章小结

Laravel 的服务容器有很多名称，但最终目标是使确定如何将某些字符串名称解析为具体实例这一操作更容易。这些字符串名将是类或接口的完全限定类名，或者类似 `log` 的快捷方式。

给定一个字符串键（例如 `app('log')`），每个绑定都会指导应用程序如何解析一个具体的实例。

容器非常智能，可以执行递归依赖性解析。因此，如果试图解析具有构造函数依赖关系的实例，容器将会根据它们的类型来解析依赖关系，然后传递到类中，最后返回一个实例。

有多种方法可以绑定到容器，但是最后它们都定义给定字符串所返回的内容。

`facade` 是一种简单的快捷方式，可以很轻松地使用根命名空间类中的静态调用从容器解析的类中调用非静态方法。

大多数开发人员都认同，对代码进行测试非常有好处，而且应该这样做。我们想知道为什么这样做是有好处的，为此甚至会阅读一些关于应该如何进行测试的教程。

但是了解为什么要测试和掌握如何测试，这之间的差距很大。值得庆幸的是，如 PHPUnit、Mockery、PHPSpec 这样的工具为 PHP 测试提供了数量惊人的选项，同时依然可以很好地进行选项设置。

Laravel 与 PHPUnit（单元测试）、Behat（行为驱动开发）、Mockery 和 Faker（为播种和测试创建假数据）进行了整合。它也具有自己简单且强大的应用程序测试工具套件，允许我们获得网站的 URI，点击按钮，提交表单，检查 HTTP 状态码，并对 JSON 进行验证和断言。

Laravel 的测试装置中包含一个可以在创建新的应用程序时成功运行的示例应用程序测试。这意味着我们无须花费时间来配置测试环境，减少了编写测试的障碍。

测试条件

令所有的程序员都认同“用同一种术语来定义不同类型的测试”，这是很难的一件事。

在这本书中，我们将用以下三个基本术语来定义测试。

单元测试

单元测试通常针对规模较小、相对独立的单元——类或方法。

集成测试

集成测试用于测试单个单元间协同工作并传递消息的方式。

应用测试

通常被称为接收 (acceptance) 或功能测试 (functional test)，应用测试对应用程序的完整行为进行测试，通常采用类似文档对象模型的爬虫——这正是 Laravel 应用测试套件所提供的。

测试基础

Laravel 中的测试存在于测试文件中，可以看到，默认情况下有两个文件：*TestCase.php*，通过应用测试进行扩展的基础类；*ExampleTest.php*，将在新的应用程序中返回绿色的即将运行的应用测试。

如示例 12-1 所示，*ExampleTest* “爬取”应用程序根路径返回页面的 DOM，并查找单词“Laravel”。如果找到它，程序将通过；如果找不到，程序运行将失败。

示例12-1 *tests/ExampleTest.php*

```
<?php
```

```
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;
```

```
class ExampleTest extends TestCase
{
    /**
     * 一个基本的功能测试实例
     *
     * @return void
     */
    public function testBasicExample()
    {
        $this->visit('/')
            ->see('Laravel');
    }
}
```

要运行这个测试，需要转到命令行并从应用程序的根文件夹中运行 `./vendor/bin/phpunit`，得到如示例 12-2 所示的输出。

示例12-2 示例测试输出

```
PHPUnit 5.5.2 by Sebastian Bergmann and contributors.
```

```
.
```

```
Time: 139 ms, Memory: 12.00Mb
OK (1 test, 2 assertions)
```

我们刚刚运行了第一个 Laravel 应用测试！可以看到，我们不仅可以使使用功能齐全的 PHPUnit 实例，还可以使用完整的 DOM 爬虫程序完成应用测试。

下面我们像示例 12-3 一样改变测试来查找单词“Applesauce”，看看会出现什么样的错误。大家会通过这个变化更加熟悉 PHPUnit。

示例12-3 `test/exampletest.php`，编辑失败

```
public function testBasicExample()
{
    $this->visit('/')
        ->see('Applesauce');
}
```

以上示例的输出结果看上去可能有点像示例 12-4。

示例12-4 输出未使用样本

```
PHPUnit 5.5.2 by Sebastian Bergmann and contributors.
```

```
F
```

```
Time: 115 ms, Memory: 12.00Mb
```

```
There was 1 failure:
```

```
1) ExampleTest::testBasicExample
```

```
<source of page here>
```

```
Failed asserting that the page contains the HTML [Applesauce].
```

```
Please check the content above.
```

```
/path-to-your-app/vendor/.../Foundation/Testing/Constraints/PageConstraint.php:90
/path-to-your-app/vendor/.../Foundation/Testing/Concerns/InteractsWithPages.
php:271
```

```
/path-to-your-app/vendor/.../Foundation/Testing/Concerns/InteractsWithPages.php:287
/path-to-your-app/tests/ExampleTest.php:21
```

FAILURES!

Tests: 1, Assertions: 2, Failures: 1

分解来看以上示例。首先,我们得到一个 F 而不是顶部的 “.” (在 PHPUnit 参数信息下)。然后,每个错误都表明了测试名称,这里为 `1)ExampleTest::testBasicExample`, 错误消息 (断言失败) 以及错误的完整堆栈跟踪,所以我们能知道什么被调用。由于这是一个应用测试,所有堆栈跟踪只是告诉我们它是通过 `InteractsWithPages` 被调用的,但是,如果这是一个单元测试,我们将看到完整的调用堆栈的测试。

JSON 测试示例

正如以下例子中所显示的,JSON 测试是简单明了的,可能比任何其他类型的应用测试都简单。

```
public function test_people_list_shows_person_after_creation()
{
    $this->json('post', 'people', ['name' => 'matt']);
    $this->json('get', 'people');
    $this->seeJson(['name' => 'matt']);
}
```

仅运行 POST、GET、DELETE 或其他某些操作,然后断言数据库或额外的 GET 响应,或者执行特定动作后所期望得到的返回。

下面来更加深入地了解一下关于 Laravel 测试环境的内容。

命名测试

默认情况下,Laravel 的测试系统将运行 `tests` 目录下所有名字以 “*Test*” 结尾的测试文件。这就是默认运行 `tests/ExampleTest.php` 的原因。

如果不熟悉 PHPUnit,你可能不知道,在测试中只有名字以 `test` 开头的方法或者使用 `@test` docblock 的方法才会运行。参见示例 12-5,来看看哪些方法会运行,哪些方法不会运行。

示例12-5 命名PHPUnit方法

```
class Naming
{
    public function test_it_names_things_well()
    {
        // 运行 "test it names things well"
    }

    public function testItNamesThingsWell()
    {
        // 运行 "It names things well"
    }

    /** @test */
    public function it_names_things_well()
    {
        // 运行 "it names things well"
    }

    public function it_names_things_well()
    {
        // 不运行
    }
}
```

测试环境

每当有 Laravel 应用程序运行时，都有显示当前的“环境”名称表示其正在运行的环境。这个名称可以设置为 `local`、`staging`、`production` 或任何我们想要设置的。可以通过运行 `app()->environment()` 来进行检索，也可以运行如 `if (app()->environment('local'))` 这样的方法来测试当前环境是否与传递的名称匹配。

当运行测试时，Laravel 会自动将环境设置为 `testing`。这意味着我们可以测试 `if (app()->environment('testing'))` 来启用或禁用测试环境中的某些行为。

另外，Laravel 不会从 `.env` 中加载正常的环境变量进行测试。如果要为测试设置环境变量，请编辑 `phpunit.xml`，对于要传递的每一个环境变量，需要在 `<php>` 部分添加一个新的 `<env>`，例如 `<env name="DB_CONNECTION" value="sqlite"/>`。

使用 .env.test 从版本控制中排除测试环境变量

如果要为测试设置环境变量，可以按上面所描述的在 `phpunit.xml` 中进行操作。但是，如果希望测试的环境变量在每个测试环境中都有所不同，该怎么办呢？或者，如果希望它们被排除在源代码控制之外，该怎么办呢？

幸运的是，实现这些要求相当容易。首先，创建一个 `.env.test.example` 文件——就像 Laravel 的 `.env.example` 文件——并将 `.env.test` 添加到 `.gitignore` 文件中。接下来，将特定环境的变量添加到 `.env.test.example` 文件中，就像在 `.env.example` 中进行设置一样。然后，复制 `.env.test.example` 并命名为 `.env.test`。

最后将该文件加载到测试中。在 `tests/TestCase.php` 的 `createApplication()` 方法中，将下面的代码粘贴到 `$app = require(...)` 一行。

```
if (file_exists(dirname(__DIR__) . '/.env.test')) {  
    (new \Dotenv\Dotenv(dirname(__DIR__), '.env.test'))->load();  
}
```

现在可以通过加载 `.env.test` 为每一次测试提供环境变量了。

测试特性

在正式介绍可用于测试的方法前，大家需要先了解适用于所有测试类的三个测试特性。

没有中间件

如果将 `Illuminate\Foundation\Testing\WithoutMiddleware` 导入测试类，那么它将禁用该类中所有测试的中间件。这意味着，我们不必担心中间件认证或 CSRF 保护，以及其他可能在实际应用中有用但分散在测试中的事务。

数据库迁移

Laravel 提供了两种开箱即用的工具，可以让数据库在测试时处于正确的状态：`DatabaseMigrations trait` 和 `DatabaseTransactions trait`。

如果导入 `DatabaseMigrations trait`，它将在每次测试之前及之后分别运行整个数据库迁移。在每次测试运行之前，Laravel 都会通过在 `setUp()` 方法中运行 `php artisan migrate` 来实现上述操作，每次测试完成后，`php artisan migrate:rollback` 将在

`tearDown()` 方法中运行。

数据库事务

另一方面，`DatabaseTransactions` 期望数据库在测试开始之前被正确迁移。然后，它将每个测试包装在一个数据库事务中，在每次测试结束时回滚。这意味着，每次测试结束时，数据库将回到与测试之前完全相同的状态。

应用程序测试

现在我们已经确定了 Laravel 测试环境的基本框架，下面来看看它是如何工作的。

可以看到，在 Laravel 的默认 `ExampleTest` (`tests/ExampleTest.php`) 中，仅需几行代码，我们就可以获取应用程序中特定的 URI，并实际检查某些单词的输出。但是 `PHPUnit` 如何像浏览器一样浏览页面呢？

测试用例

任何应用测试都应该扩展 Laravel 默认包含的 `TestCase` 类 (`tests/TestCase.php`)。应用程序的 `TestCase` 类将扩展抽象类 `Illuminate\Foundation\Testing\TestCase`，进而带来许多好处。

首先，这两个 `TestCase` 类 (`TestCase` 类和它的抽象父类) 执行的第一步操作是处理引导 `Illuminate` 应用程序实例，使完全引导的应用程序可用。它们还“刷新”每个测试之间的应用程序，这意味着它们不是完全在测试之间重新创建应用程序的，而是确保没有任何数据延迟。

父 `TestCase` 类还设置一个允许在创建应用程序之前及之后回调运行的钩子系统，并导入一系列方法特性，为我们提供与应用程序的方方面面进行交互的方法。这些特性包括 `InteractsWithContainer`、`MakesHttpRequests`、`InteractsWithConsole` 等，它们还引入了各种各样的自定义断言和测试方法。

因此，应用测试获得了访问完全自举的程序实例的权限，一系列简单且强大的包装器使应用测试自定义断言和 DOM 爬虫变得更易于使用。

我们可以编写一个 `$this->visit('/')->see('Laravel')`，知道应用程序实际上表现得像响应了一个普通的 HTTP 请求一样，并且响应被传递给一个正在检查该文本的 DOM 爬虫。这是一个非常强大的功能。

下面来看一看公开的基本方法。



Laravel 5.1 中的不同特性结构

在 Laravel 5.1 中，测试特性的结构以及测试框架的组织方式与这里描述的不同，但是功能是相同的。

“访问”路径

这是最复杂的 Laravel 应用测试功能同时也最简单且最强大的。使用这些方法，测试可以在应用程序中与（“访问”）页面进行交互，这是前所未有的。

```
$this->visit($uri)
```

访问路径是 Laravel 应用测试的核心。调用 `$this->visit('dashboard')` 时，我们会模仿框架在 Web 请求进入同一路径时所采取的操作。应用程序将为该请求创建一个请求对象，并将相应对象（`Illuminate\Http\Response` 实例）存储在 `$this->response` 中。

这是通常会返回且显示在浏览器中的响应对象，但它只缓存并检查 Laravel 的应用测试断言（或者想要与响应交互的代码）。

就其本身而言，访问的作用不大，因为我们已经在 `$this->response` 中缓存了响应，所以可以针对它编写断言。

使 visit() 与其他访问方法不同的因素

我们将介绍 `call()` 和 `get()`，以及与访问路径相关的其他方法。它们比 `visit()` 简单得多，值得注意的是使得它们和 `visit()` 不同的因素。

以下是一个缩短定义的 `visit()` 方法。

```
public function visit($uri)
{
    $uri = $this->prepareUrlForRequest($uri);

    $this->call($method, $uri, $parameters, $cookies, $files);

    $this->clearInputs()->followRedirects()->assertPageLoaded($uri);
}
```

```

        $this->currentUri = $this->app->make('request')->fullUrl();

        $this->crawler = new Crawler(
            $this->response->getContent(),
            $this->currentUri
        );

        return $this;
    }

```

这段代码中，`visit()` 完成了很多工作。当想要检查页面加载情况时，或者想要获取一个页面时，使用 `visit()` 几乎可以完成所有的应用测试，近乎神奇。

如果只是想得到响应，或者使用传统的检查来 POST 页面并断言会发生什么行为，而没有别的需求，则可以使用更简单的方法，如 `call()`。

```

$this->call($method,$uri,$params=[],$cookies=[],$files=[],
$server=[],$content=null)

```

如果要对服务器进行调用，而无须担心爬虫返回的 DOM（例如，要断言给定的 POST 有一定的效果），有一种方法可以实现。`visit()` 实际上是基于 `call()` 的，但是我们也可以直接使用 `call()`。

从方法定义中可以看到，当使用 `call()` 时，有很多选择——HTTP 方法、URI、参数、cookie 消息块和文件——都假装与调用一起被发送。

就像使用 `visit()` 时一样，这些请求将发出请求并将响应存储在 `$this->response` 上，但不会启用任何基于 DOM 爬虫的断言，如 `see()`。

```

$this->get($uri, $headers = []), ->post($uri, $data = [], $headers = []),
->put($uri, $data = [], $headers = []), ->patch(), 和 ->delete()

```

这是一系列包装 `call()` 的助手，它们是将特定字符串传递给 `call()` 的第一个参数的 HTTP 快捷方式。我们也可以像使用 `call()` 一样使用它们。

```

$this->json($method, $uri, $data = [], $headers = [])

```

就像 `get()`、`post()` 等刚刚提到的方法一样，`json()` 是 `call()` 的包装器。它将传递的数据转换为 JSON 数据，并添加 JSON 请求头，然后将其全部传递给 `call()`。

`json()` 对于测试 JSON API 来讲是非常有用的，因为我们甚至可以通过它来自定义

头文件和数据，所以可以使用这个方法在测试中与 REST API 进行充分交互。

```
$this->followRedirects()
```

实际上 `visit()` 还具有一个 `call()` 所不具备的特点：它告诉 Laravel 遵循重定向方式使用 `followRedirects()`，然后使用 `assertPageLoaded()` 检查最终的登录页面是否加载。

没有 `followRedirects()`，在调用重定向页面后，得到的响应仅仅是重定向的内容，而不是被重定向的页面。

自定义应用测试断言

我们获得了哪些新的应用测试断言呢？实际上有许多，先从简单的开始介绍，逐步提升难度。

```
$this->assertPageLoaded()
```

`assertPageLoaded()` 检查在加载页面时获得的 HTTP 状态码 200。

```
$this->see() 和 ->dontSee()
```

就像本章前面所介绍的，`see()` 获取一个字符串并使用正则表达式来检查字符串是否存在于目前呈现的页面上。`dontSee()` 是它的反函数。

```
$this->seeLink() 和 ->dontSeeLink()
```

`seeLink()` 具有两个参数：第一个是要查找的链接文本，第二个是可选的 URL。`dontSeeLink()` 是它的反函数。

```
$this->seeHeader()
```

`seeHeader()` 具有两个参数：第一个是头部名称，第二个是可选的头部值。

```
$this->seeCookie()
```

`seeCookie()` 具有两个参数：第一个是 cookie 的名称，第二个是可选的 cookie 值。

```
$this->seeInField() 和 ->dontSeeInField()
```

`seeInField()` 具有两个参数：第一个是要查看的输入或文本区域的名称及 ID，第二个是要查找的值。`dontSeeInField()` 是它的反函数。

`$this->seeIsChecked()` 和 `->dontSeeIsChecked()`

`seeIsChecked()` 只具有一个参数：复选框输入的名称或 ID。`dontSeeIsChecked()` 是它的反函数。

`$this->seeIsSelected()` 和 `->dontSeeIsSelected()`

`seeIsSelected()` 具有两个参数：第一个是要检查的复选框的名称或 ID，第二个是需要正确设置的值。`dontSeeIsSelected()` 是它的反函数。

`$this->seePageIs()`

`seePageIs()` 断言当前加载的页面 URI 与传递给它的参数相同。

`$this->seeInDatabase()` 和 `->dontSeeInDatabase()`

若要检查数据库表中的记录，须将表名作为 `seeInDatabase()` 的第一个参数，将要查找的数据作为第二个参数。

```
public function test_database_has_user_after_registration()
{
    $this
        ->visit('register')
        ->fillForm([
            'email' => 'matt@mattstauffer.co'
        ])
        ->submitForm();
    $this->seeInDatabase('emails', ['email' => 'matt@mattstauffer.co']);
}
```

如上面的代码所示，`seeInDatabase()` 的第二个“数据”参数是结构化的，类似于一个 SQL WHERE 语句——传递一个键或一个值（或者多个键和值），然后 Laravel 会在指定的数据库表中查找与上述键和值匹配的记录。

`dontSeeInDatabase()` 是它的反函数。

JSON 和 Non-visit() 应用测试断言

其余的应用程序断言与 `visit()` 方法的关系并不密切，但与应用程序的实现细节关系非常密切，其中一些也经常用于测试 JSON API。

`$this->seeJson()`，`->dontSeeJson()`，`->seeJsonEquals()`

`seeJson()` 没有参数检查，以确保响应的内容是有效的 JSON，其可选参数表示正

在检查的数据。例如，在下面的示例中，我们收到一个响应，检查其是否为有效的 JSON，以及是否包含 `username/mattstauffer` 键 / 值对。

```
public function test_api_returns_certain_json()
{
    $this->json('get', 'users');
    $this->seeJson(['username' => ,mattstauffer']);
}
```

`seeJson()` 有一个反函数 `dontSeeJson()`。`dontSeeJson()` 仍然希望得到有效的 JSON，但不希望有任何数据传入。

如果想检查 JSON 是否完全映射到了数据中，可以尝试使用 `seeJsonEquals()` 方法，它将数据数组与 JSON 响应进行比较，如果它们不完全匹配，就抛出异常。

`$this->assertResponseOK()` 和 `->assertResponseStatus($status)`

在 `call()` 或 `visit()` 之后，有价值的断言就是页面加载了我们期望的 HTTP 状态。尽管 `assertResponseOK()` 断言页面返回了 200 HTTP 状态码，但是也可以传递我们期望的特定状态。

```
public function test_pages_load_the_way_we_want()
{
    $this->get('people');
    $this->assertResponseOK();

    $this->call('post', 'owners');
    $this->assertResponseStatus(405); // 方法不允许
}
```

可以通过断言提供 401（未经授权的）状态码来检查特定路由的授权设置，然后进行身份验证并断言它给出了 200 状态码。

`$this->assertViewHas($key, $value = null)`，`->assertViewHasAll(array $bindings)`，`->assertViewMissing($key)`

有时候，唯一的情况就是断言我们使用 `see()` 在页面上看到了一个特定的短语，但更有可能的是检查正确的数据是否被传递到了视图中。幸运的是，我们可以直接使用这些方法进行检查。

`assertViewHas()` 检查具有特定密钥的数据是否被发送到最近检索的视图中，如果将断言传递给第二个参数，则它将断言该数据与之相同。

```
// 路由
```



```
Route::get('test', function () {
    return view('test')->with('foo', 'bar');
});

// 测试
public function test_view_gets_data()
{
    $this->get('test');
    $this->assertViewHas('foo'); // 正确
    $this->assertViewHas('foo', 'bar'); // 正确
    $this->assertViewHas('foo', 'baz'); // 不正确
}
```

还可以同时使用断言 `viewhasall()` 检查多个视图变量,期望得到一组键/值对数组。

```
// 路由
Route::get('test', function () {
    return view('test')
        ->with('foo', 'bar')
        ->with('baz', 'qux');
});

// 测试
public function test_view_gets_data()
{
    $this->get('test');
    $this->assertViewHasAll([
        'foo' => 'bar',
        'baz' => 'qux'
    ]); // 正确
}
```

可以通过将该密钥传递给 `assertViewMissing()` 来确保视图尚未与特定的密钥匹配。

5.3 如下例所示, 在 Laravel 5.3 版本中, 我们可以向 `assertViewHas()` 的第二个参数传递一个闭包, 这样可以对视图所提供的数据进行更细微的检查。

```
public function test_events_are_owned_by_user_1()
{
    $this->get('events');
    $this->assertViewHas('events', function ($events) {
        return $events->reject(function ($event) {
            return $event->user_id == 1;
        })->isEmpty();
    });
}
```

```
$this->assertRedirectedTo(), ->assertRedirectedToRoute(),  
->assertRedirectedToAction())
```

如果希望用户不仅能在特定的页面上结束任务，而且还能作为重定向进行发送，则可以使用以下方法。通过 URL(`to()`)、路由名称(`toRoute()`)、控制器以及方法(`toAction()`)来进行检查。

```
// 路由  
Route::get('redirector', function () {  
    return redirect('/');  
});  
  
Route::get('/', 'HomeController@index')->name('home');  
  
// 测试  
public function test_redirector_works()  
{  
    $this->get('redirector');  
    $this->assertRedirectedTo('/');  
    $this->assertRedirectedToRoute('home');  
    $this->assertRedirectedToAction('HomeController@index');  
}  
$this->assertSessionHas($key, $value = ''),  
  
->assertSessionHasAll($bindings, ->assertSessionHasErrors($bindings =  
[], $format = null), ->assertHasOldInput())
```

使用这些方法可以更方便地检查会话中的特定值。`assertSessionHas()`和`assertSessionHasAll()`的用法与`assertViewHas()`和`assertViewHasall()`相似。

当仅传递一个参数的时候，`assertSessionHas()`断言该键值有一个对应的会话值；如果要传递两个参数，则会断言该会话键值和会话参数相等。`assertSessionHasAll()`接收一个键值对的数组，断言每个键值都存在于会话中，并且被设置为其所对应的值。

```
public function test_session_has_stuff()  
{  
    Session::put('foo', 'bar');  
    Session::put('baz', 'qux');  
    $this->assertSessionHas('foo');  
    $this->assertSessionHas('foo', 'bar');  
    $this->assertSessionHasAll([  
        'foo' => 'bar',
```

```

        'baz' => 'qux'
    });
}

```

没有任何参数声明的 `assertSessionHasErrors()` 断言在 Laravel 的 `errors` 会话容器中至少存在一个错误设置。它的第一个参数可以是一组键 / 值对，用于定义应该被设置的错误，第二个参数可以是已检查的错误应该被格式化的字符串形式，如下所示。

```

public function test_posting_empty_errors_out()
{
    // 假设 "/form" 路由需要一个电子邮件域
    // 并且我们向它投递一个空的表格来触发错误
    $this->post('form', []);
    $this->assertSessionHasErrors();
    $this->assertSessionHasErrors(['email' => 'The email field is
required.']);
    $this->assertSessionHasErrors(
        ['email' => '<p>The email field is required.</p>'],
        '<p>:message</p>'
    );
}

```

最后，`assertHasOldInput()` 断言一些旧的输入已经从提交的表单中被保存，可能使用了 `redirect()->withOldInput()` 方法。

点击和表单

下面，让我们移步到一些神奇却又可怕的力量中：浏览表单、点击、填写、取消选择、添加文件附件。Laravel 为表单的使用提供了如下方法。

`$this->click($name)`

给定的 `$name` 的链接作为链接主体、名称或 ID，`click()` 从该链接中获取 URI 并访问。

`$this->type($text,$element)`

给定页面中提供的 `$element` 作为名称或 ID 输入，使用 `type()` 方法向其中“输入”提供的文本。

操作表单

通过点击链接和输入表单字段的会话，Laravel 似乎运行了某种基于 JavaScript 的驱动浏览器和页面交互的应用程序测试。但事实并非如此。

它存储了我们正在创建的“输入”，如果在某些时候“提交”表单，它会将输入收集到一起，并发布到表单的目标地。从理论上讲，这与用户的普遍经验有很大的不同，但实际上它确实是一个语法精美的编写模拟表单提交测试的方法。

```
$this->check($element)
```

在页面上给定一个以 *\$element* 为名称或 ID 的复选框，用 `check()` 来“检查”。

```
$this->unchecked($element)
```

在页面上给定一个以 *\$element* 为名称或 ID 的复选框，用 `unchecked()` 来“取消检查”。

```
$this->select($option, $element)
```

在页面上给定一个以 *\$element* 为名称或 ID 的复选框，用 `select()` 将其值设置为 *\$option*。

```
$this->attach($filePath, $element)
```

在页面上给定一个以 *\$element* 为名称或 ID 的复选框，`attach()` 用来将文件从给定的本地路径传送到表单，并在表单提交时标记为上传。

```
$this->press($buttonText)
```

给定页面上含有所提供文本的按钮，`press()` 提交按钮是其中一部分表单。

```
$this->submitForm($buttonText, $inputs = [], $uploads = [])
```

给定页面上含有所提供文本的按钮，`submitForm()` 提交按钮是表单的一部分。还可以使用第二个或第三个参数来选择设置或覆盖所有输入及上传的文件。

```
$this->fillForm($buttonText, $inputs = [])
```

给定页面上含有所提供文本的按钮，`fillForm()` 按钮是其中一部分表单，并将所有值设置为提供的值。

```
$this->clearInputs()
```

`clearInputs()` 将清除之前设置的所有输入和上传的文件。

任务和事件

我们将在第 16 章中更加深入地讨论与任务和事件相关的测试，现在先来快速了解一下它们是如何工作的。

```
$this->expectsEvents($eventClassName)
```

如果想要在测试过程中断言某种特定类型的事件将被触发，可以将类名传递到 `expectsEvents()` 中。

```
public function test_usersubscribed_event_fires_when_subscribing() {
    $this->expectsEvents(App\Events\UserSubscribed::class);

    $this->visit('subscribe')->type('me@me.com', 'email')
        ->press('Subscribe');
}
```

```
$this->withoutEvents()
```

`withoutEvents()` 实际上不是断言，相反，它会禁用 Laravel 的事件处理系统，因此，在测试期间，我们不必担心任何事件影响——例如发送任何电子邮件或写入任何日志。

```
$this->expectsJobs()
```

如果想要在测试过程中断言某种特定类型的任务将被触发，则可以将该类名传递给 `expectsJobs()`。

```
public function test_number_of_subscriptions_crunches_reports()
{
    $this->expectsJobs(App\Jobs\CrunchReports::class);

    $this->visit('subscribe')->type('me@me.com', 'email')
        ->press('Subscribe');
}
```

认证和会话

Laravel 可以轻松地为测试设置一个测试环境，甚至可以轻松控制会话并作为给定用户进行身份验证。

```
$this->session(['key' => 'value'])
```

`session()` 启动会话，并将所提供的数组中的键值对数据对保存到会话中。如果有需要，可以在测试期间多次运行会话来添加不同的会话数据。

```
$this->flushSession()
```

`flushSession()` 清除当前会话中所有的数据。

```
$this->be($authenticatable)
```

`be()` 接收符合 `Illuminate\Contracts\Auth\Authenticatable` 接口（包括基础的 `App\User` 类）的所有对象，并以该用户身份来验证测试中的每个页面请求或交互。这意味着我们可以编写如下测试。

```
public function test_members_cant_see_admin_dashboard()
{
    $member = factory(\App\User::class, 'member')->create();
    $this->be($member);

    $this->get('admin-dashboard');
    $this->assertResponseStatus(403);
}
public function test_admins_can_see_admin_dashboard()
{
    $admin = factory(\App\User::class, 'admin')->create();
    $this->be($admin);
    $this->get('admin-dashboard');
    $this->assertResponseOK();
}
```

Artisan 和 Seed

还有两个测试方法需要看一下。

```
$this->artisan($command, $parameters = [])
```

如果想在测试中使用 Artisan 命令，`artisan()` 可以轻松实现。只需将命令名称作为第一个参数传递，并且可选地将其他参数作为数组传递给第二个参数。

这样可以将响应代码保存到 `$this->code` 中。因此，这和 `Artisan::call()` 功能相同，另外还添加了对 `$this->code` 的响应保存。

```
public function test_returns_certain_code()
{
    $this->artisan('do:thing', ['--flagOfSomeSort' => true]);
    $this->assertEquals(0, $this->code); // 0 表示“没有错误返回”
}
```

```
$this->seed($seederClassName = 'DatabaseSeeder')
```

如果想要填充数据库，`seed()` 方法可以实现；如果要传递一个参数，可以选择只运行一个 seeder。

`seed()` 提供了和 `$this->artisan('db:seed')` 相同的功能。



模型工厂

模型工厂是一种令人惊叹的工具。它使得随机播种以及测试结构良好的数据库中的数据（或其他功能）变得很简单。

我们已经深入地了解了它们，请参照前文中关于“模型工厂”的内容掌握更多信息。

mock（模拟）

mock（包括 spy、stub、dummie、fake 以及其他相关工具等）是测试的常用工具。此处不会详细介绍，但无论什么规模的应用程序，如果没有进行至少一次 mock，一定无法进行完全的测试。

从本质上讲，mock 和其他相似的工具一样，可以创建一个以某种方式模仿真实类但又不是真实类的对象，该对象仅用于测试。有时这样做是因为，将真正的类引入测试或与外部服务进行通信非常难以实例化。

从这些例子中可以看出，Laravel 鼓励尽可能多地使用真正的应用程序——这样可以避免对 mock 依赖过多。然而 mock 的存在还是很有必要的，这就是为什么 Laravel 包含了一个开箱即用的 mock 库——Mockery。

Mockery

Mockery 可以快速轻松地在应用程序的任何 PHP 类中创建 mock。想象一下，假如我们有一个依赖于 Slack 客户端的类，但是不想在外面调用 Slack。这时 Mockery 可以简单地在测试中创建一个假的 Slack 客户，如示例 12-6 所示。

示例12-6 在Laravel中使用Mockery

```
// app/SlackClient.php
class SlackClient
{
    ...
    public function send($message, $channel)
    {
        // 实际上向 Slack 发送了一条消息
    }
}

// app/Notifier.php
class Notifier
{
    private $slack;

    public function __construct(SlackClient $slack)
    {
        $this->slack = $slack;
    }

    public function notifyAdmins($message)
    {
        $this->slack->send($message, 'admins');
    }
}

// tests/NotifierTest.php
public function test_notifier_notifies_admins()
{
    $slackMock = Mockery::mock(SlackClient::class)->shouldIgnoreMissing();

    $notifier = new Notifier($slackMock);
    $notifier->notifyAdmins('Test message');
}
```

此处涉及很多配件，我们分解来看。我们正在测试一个名为 **Notifier** 的类，其中有一个名为 **SlackClient** 的依赖，这个依赖可以完成实际测试中不必由它来完成的事：发送真正的 Slack 通知。因此我们要对它进行模拟。

使用 Mockery 可以得到 **SlackClient** 类的 mock。如果我们不在意该类中发生什么——假设它的存在只是简单地使测试不发生错误——便可以只使用 `shouldIgnoreMissing()`。

```
$slackMock = Mockery::mock(SlackClient::class)-shouldIgnoreMissing();
```


无论 `Notifier` 在 `$slackMock` 上调用了什么，它都会接收并且返回空值。

但是，看一下 `test_notifier_notifies_admins()`。此处，实际上它并没有测试任何内容。

我们可以保留 `shouldIgnoreMissing()` 并且在它下面编写一些断言。这通常是通过 `shouldIgnoreMissing()` 方法来完成的，使得该对象变成一个“fake”或“stub”。

然而，如果我们想要真正地声明对 `SlackClient()` 中的 `send()` 方法进行了调用，该怎么办呢？我们应该放弃 `shouldIgnoreMissing()`，并且使用 `should*` 方法（参见示例 12-7）。

示例12-7 在Mockery中使用shouldReceive方法

```
public function test_notifier_notifies_admins()
{
    $slackMock = Mockery::mock(SlackClient::class);
    $slackMock->shouldReceive('send')->once();

    $notifier = new Notifier($slackMock);
    $notifier->notifyAdmins('Test message');
}
```

`shouldReceive('send')->once()` 与“断言 `$slackMock` 将会仅调用一次 `send()` 方法”是一样的。因此，我们现在断言，`Notifier` 在调用 `notifyAdmins()` 时必须单独调用 `SlackClient` 的 `Send` 方法。

也可以使用 `shouldReceive('send')->times(3)` 或 `shouldReceive('send')->never()` 之类的方法。

如果想要使用 IoC 容器来解决 `Notifier` 的实例该怎么办呢？当 `Notifier` 中含有几个不需要 mock 的依赖时，这是很有用的，并且不难实现。

如示例 12-8 所示，使用容器中的 `instance()` 方法，告诉 Laravel 向任意请求它的类提供一个模拟实例（在这个例子中，是指 `Notifier`）。

示例12-8 绑定Mockery实例和容器

```
public function test_notifier_notifies_admins()
{
    $slackMock = Mockery::mock(SlackClient::class);
    $slackMock->shouldReceive('send')->once();

    app()->instance(SlackClient::class, $slackMock);
    $notifier = app(Notifier::class);
}
```

```
$notifier->notifyAdmins('Test message');
}
```

Mockery 的用途还有很多，比如可以使用 spy，或者部分使用 spy 等。更多的关于如何使用 Mockery 的内容已经超出了本书的范围，但是仍然鼓励大家去自学关于 Mockery 库的知识以及使用它的方法。

模拟 facade

假设我们有一个调用 facade 的控制器方法。现在，想要测试这个控制器方法，并且断言应该进行 facade 调用，该怎么做呢？值得庆幸的是，做法很简单：将 facade 当成测试中的 Mockery 实例即可，如示例 12-9 所示。

示例12-9 模拟facade

```
// PeopleController (控制器)
public function index()
{
    return Cache::remember('people', function () {
        return Person::all();
    });
}

// PeopleTest (测试)
public function test_all_people_route_should_be_cached()
{
    $person = factory(Person::class)->make();

    Cache::shouldReceive('remember')
        ->once()
        ->andReturn(collect([$person]));
    $this->visit('people')->seeJson(['name' => $person->name]);
}
```

可以看到，我们可以在 facade 上使用类似 `shouldReceive()` 一样的方法，就像对 Mockery 对象进行的操作一样。

5.3 从 Laravel 5.3 版本起，我们可以把 facade 当作 spy 来用，这意味着可以在最后设置断言，并且使用 `shouldHaveReceived()` 而不是 `shouldReceive()`，示例 12-10 说明了这一点。

示例12-10 facade spy

```
public function test_queue_job_should_be_pushed_after_regisration()
{
    Cache::spy();
```

```
$this->post('register', ['email' => 'joaquin@me.com']);

Cache::shouldReceive('push')
->with(SendWelcomeEmail::class, ['email' => 'joaquin@me.com']);
}
```

本章小结

Laravel 可以使用任何现代的 PHP 测试框架，但是当使用 PHPUnit，并且测试扩展了 Laravel 的 `TestCase` 时，则会产生很多框架特定的功能。Laravel 的应用测试框架让通过应用程序发送 fake 请求并且检查结果这一行为变得简单，甚至提交表单之前的“键入”和“点击”按钮也变得简单。

`TestCase` 类提供了一组方法，可以轻松自定义测试用例与数据库互动的方法、禁用事件的影响，以及对框架级结构（如任务及 facade）做出断言。

如果需要 mock、stub、spy、dummie 或其他内容，Laravel 会提供 Mockery 库来满足需求。但 Laravel 的测试理念是尽可能多地使用 collaborator（一款代码审查工具）。除非万不得已，不然不要弄虚作假。

Laravel 开发者最常见的任务之一就是创建 API，一般是 JSON 和 REST，或类 REST 的格式，API 允许第三方与 Laravel 应用数据进行交互。

Laravel 处理 JSON 非常简单，并且它的资源控制器已经围绕 REST 动词和模式进行了结构化。本章会介绍一些在编写第一个 Laravel API 时用到的基本 API 编写概念，在 Laravel 中编写 API 的外部工具和组织系统。

类 REST JSON API 基础

表现层状态转化（REST）是创建 API 时的一个架构类型。从技术上说，广义的 REST 可以用于整个互联网，所以不要被它的定义吓到了。当我们在 Laravel 中谈到 REST 风格或类 REST API 时一般都是在说具备一些共同特点的 API，如下所示。

- 围绕仅由 URI 表示的“资源”结构化，如 `/cats` 表示所有的 cats，`/cats/15` 表示 ID 为 15 的单个 cat 等。
- 主要使用 HTTP 动词与资源进行交互（Get `/cats/15` 对比 DELETE `/cats/15`）。
- 无状态，也就是说在请求之间不存在持久化会话认证，每个请求必须唯一认证。
- 可缓存性和一致性，也就是说无论请求者是谁，每个请求（除了少数认证的特定用户请求）的返回结果都应该相同。
- 返回 JSON。

最常见的 API 模式就是每一个作为 API 资源的 Eloquent 模型都有唯一的 URL 结构，并且允许用户通过特殊的动词与资源进行交互并得到 JSON 返回值。示例 13-1 提供了一些常见的例子。

示例13-1 常见的REST API端点结构

GET /api/cats

```
[
  {
    id: 1,
    name: 'Fluffy'
  },
  {
    id: 2,
    name: 'Killer'
  }
]
```

GET /api/cats/2

```
{
  id: 2,
  name: 'Killer'
}
```

DELETE /api/cats/2

deletes cat

POST /api/cats with body:

```
{
  name: 'Mr Bigglesworth'
}
(creates new cat)
```

PATCH /api/cats/3 with body:

```
{
  name: 'Mr. Bigglesworth'
}
(updates cat)
```

可以看到，我们可以与 API 进行一些基本的交互。下面来深入学习如何让 API 在 Laravel 中工作。

控制器组织和 JSON 返回

Laravel 的资源控制器与 REST 风格的 API 控制器结构类似，所以先来看看这个知识点。首先为资源创建一个新的控制器，将它指向 `/api/dogs`，如下所示。

```
php artisan make:controller Api/\DogsController --resource
```

记住，`--resource` 标志会生成一个资源控制器而不是普通的控制器。



在 Artisan 命令中转义反斜杠

为了将 `DogsController` 放到 `Api` 命名空间下，我们需要用正斜杠 `\` 来转义，命名空间反斜杠。

从示例 13-2 中可以看出控制器是如何工作的。

示例13-2 一个已生成的资源控制器

```
<?php

namespace App\Http\Controllers\Api;

use Illuminate\Http\Request;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class DogsController extends Controller
{
    /**
     * 显示资源列表
     *
     * @return \Illuminate\Http\Response
     */
    public function index() {}

    /**
     * 显示创建新资源的格式
     *
     * @return \Illuminate\Http\Response
     */
    public function create() {}

    /**
     * 存储新创建的资源
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request) {}
```

```

/**
 * 显示特定的资源
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function show($id) {}

/**
 * 显示编辑特定资源的格式
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function edit($id) {}

/**
 * 在内存中更新特定资源
 *
 * @param \Illuminate\Http\Request $request
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id) {}

/**
 * 从内存中删除特定资源
 *
 * @param int $id
 * @return \Illuminate\Http\Response
 */
public function destroy($id) {}
}

```

docblock 实现了很多功能。index() 里显示了所有 dog，show() 列出了单个 dog，create() 显示新建视图，store() 存储 dog，edit() 显示编辑视图，update() 更新 dog 以及 destroy() 删除 dog。

因为这是一个 API，我们可以删除 bat 的 create() 和 edit() 方法，在这里我们不会显示视图。

下面来快速创建一个模型和迁移。


```
php artisan make:model Dog --migration
php artisan migrate
```

接下来就可以补充我们的控制器方法了。

可以利用 Eloquent 强大的特性，如果输出一个 Eloquent 的结果集合，它会自动被转化成 JSON 格式（使用 `__toString()` 魔术方法）。也就是说如果从路由中返回一个结果集合，实际上会返回 JSON。

如示例 13-3 所示，这可能是最简单的代码。

示例13-3 dog实体的资源控制器示例

```
...
class DogsController extends Controller
{
    public function index()
    {
        return Dog::all();
    }

    public function store(Request $request)
    {
        Dog::create($request->all());
    }

    public function show($id)
    {
        return Dog::findOrFail($id);
    }

    public function update(Request $request, $id)
    {
        $dog = Dog::findOrFail($id);
        $dog->update($request->all());
    }

    public function destroy($id)
    {
        $dog = Dog::findOrFail($id);
        $dog->delete();
    }
}
```

示例 13-4 展示了怎样将其连接到路由文件中。

示例13-4 为资源控制器绑定路由

```
// Routes.php
Route::group(['prefix' => 'api', 'namespace' => 'Api',function() {
    Route::resource('dogs', 'DogsController');
}]);
```

现在已经在 Laravel 中创建了第一个 REST 风格的 API。

当然我们还需要更多个性化的区分：分页、排序、认证以及更多定义的响应头。这些是其他一切功能的基础。

读取和发送头

REST 的 API 一般通过头读取和发送非内容信息。例如对 GitHub 的 API 的任何请求都会返回包含当前用户速率限制状态的头，如下所示。

```
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4987
X-RateLimit-Reset: 1350085394
```



X-* 头

你可能会好奇为什么 GitHub 速率限制头的前缀是 X-，特别是在使用相同的请求返回了其他头时，如下所示。

```
HTTP/1.1 200 OK
Server: nginx
Date: Fri, 12 Oct 2012 23:33:14 GMT
Content-Type: application/json;charset=utf-8
Connection: keep-alive
Status: 200 OK
ETag: "a00049ba79152d03380c34652f2cb612"
X-GitHub-Media-Type: github.v3
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4987
X-RateLimit-Reset: 1350085394
Content-Length: 5
Cache-Control: max-age=0, private, must-revalidate
X-Content-Type-Options: nosniff
```

在头中，任何前缀为 X- 的头都不符合 HTTP 规范。它可能完全是由单词组成的（例如 X-How-Much-Matt-Loves-This-Page），或者尚未成为规范的常见公约的一部分（例如 X-Requested-With）。

同样地，很多 API 允许开发者使用请求头来自定义请求。例如 GitHub 的 API 允许使用 `Accept` 头来定义 API 的版本，如下所示。

```
Accept: application/vnd.github.v3+json
```

如果想要将版本从 v3 修改到 v2，GitHub 就会将请求发送到 API 的版本 2 中。

下面介绍怎样在 Laravel 中实现这两个功能。

在 Laravel 中发送响应头

本书在第 10 章中已经介绍了很多相关的内容，这里先来快速回顾一下。当获得一个响应对象时，可以使用 `header($headerName, $headerValue)` 来添加头，如示例 13-5 所示。

示例13-5 在Laravel中添加响应头

```
Route::get('dogs', function () {  
    return response(Dog::all())  
        ->header('X-Greatness-Index', 9);  
});
```

非常简单。

在 Laravel 中读取请求头

如果有一个传入的请求，也可以很简单地读取任何给定的头，如示例 13-6 所示。

示例13-6 在Laravel中读取请求头

```
Route::get('dogs', function (Request $request) {  
    echo $request->header('Accept');  
});
```

现在可以读取传入的请求头，以及在 API 响应中设置该头了。下面让我们来看看怎样自定义 API。

Eloquent 分页

分页是绝大多数 API 需要首先考虑的事情。Eloquent 本身就包含分页系统，可以获取任何界面请求中的查询参数。在本书第 5 章中已经介绍过分页器组件，这里快速回顾一下。

任何 Eloquent 调用都提供了 `paginate()` 方法，可以传入希望从每个界面中返回的条目数。然后 Eloquent 会为界面查询参数检索 URL，如果 URL 已经设置好，那么 Eloquent 会将其视为用户进入分页列表的页面数量的指标。

为了让 API 路由实现自动分页，只需要在路由中使用 `paginate()` 方法而不是 `all()` 或 `get()`，如示例 13-7 所示。

示例13-7 一个分页的API路由

```
Route::get('dogs', function () {  
    return Dog::paginate(20);  
});
```

这里定义了 Eloquent 需要从数据库中获取 20 条数据。根据查询参数指定的页面，Laravel 就能知道应该获取哪 20 条查询结果，如下所示。

```
GET /dogs - Return results 1-20  
GET /dogs?page=1 - Return results 1-20  
GET /dogs?page=2 - Return results 21-40
```

注意 `paginate()` 方法也适用于查询构造器调用，如示例 13-8 所示。

示例13-8 在查询构造器调用中使用paginate()方法

```
Route::get('dogs', function () {  
    return DB::table('dogs')->paginate(20);  
});
```

很有趣的是，当你将返回结果转化为 JSON 时就不止 20 条数据了。它会创建一个响应对象，自动将一些有用的、与分页相关的信息与分页数据一起传给终端用户。示例 13-9 显示了调用的可能响应，为了节省篇幅，这里只截取了 3 条记录。

示例13-9 分页数据库调用的输出

```
{  
    "total": 50,  
    "per_page": 3,  
    "current_page": 1,  
    "last_page": 17,  
    "next_page_url": "http://myapp.com/api/dogs?page=2",  
    "prev_page_url": null,  
    "from": 1,  
    "to": 3,  
    "data": [  
        {  
            'name': 'Fido'  
        },  
        {  
            'name': 'Pickles'
```

```
    },  
    {  
        'name': 'Spot'  
    }  
]  
}
```

排序和筛选

尽管 Laravel 有用于分页的公约和一些内置工具，但是并没有能用于排序的工具，所以需要手动实现排序效果。这里提供一个简单的代码示例，并且将查询参数的类型设置为满足 JSON API 规范的类型，如下所示。

JSON API 规范

JSON API (<http://jsonapi.org/>) 是构建基于 JSON 的 API 中处理常见任务的标准，包括过滤、排序、分页、认证、嵌入、链接和元数据等。

根据 JSON API 规范，Laravel 默认的分页不能正常工作，但是它指明了方向。而对于其他大多数 JSON API 规范，只需要手动选择实现或者不实现。

例如下面的 JSON API 规范处理的是当遇到错误返回时如何结构化数据。

一个文件必须包含至少一个下述顶级成员。

- **data**: 文件的“主要数据”
- **errors**: 错误对象数组
- **meta**: 包含非标准元信息的元对象

data 和 **errors** 成员不能同时存在于一个相同的文件中。

需要注意的是，将 JSON API 作为规范是非常好的，但运行它时也需要满足一些基本条件。在示例中我们不会完全使用它，但是会介绍一些通用的用法。

对 API 结果排序

首先我们需要让 API 具备排序功能。在示例 13-10 中，将结果仅根据单列进行单方向排序。

示例13-10 最简单的API排序

```
// Handles /dogs?sort=name
Route::get('dogs', function (Request $request) {
    // 获取排序查找参数（或默认根据 "name" 排序）
    $sortCol = $request->input('sort', 'name');
    return Dog::orderBy($sortCol)->paginate(20);
});
```

在示例 13-11 中，可以进行反向排序（例如 ?sort=-weight）。

示例13-11 指定排序方向的单列API排序

```
// Handles /dogs?sort=name 和 /dogs?sort=-name
Route::get('dogs', function (Request $request) {
    // 获取排序查找参数（或默认根据 "name" 排序）
    $sortCol = $request->input('sort', 'name');

    // 使用 Laravel 的 starts_with() 帮助方法，根据键值是否以 - 开头设置排序方向
    $sortDir = starts_with($sortCol, '-') ? 'desc' : 'asc';
    $sortCol = ltrim($sort, '-');

    return Dog::orderBy($sortCol, $sortDir)
        ->paginate(20);
});
```

最后我们将同样的方法应用到多列中（例如 ?sort=name, -weight），如示例 13-12 所示。

示例13-12 JSON API风格的排序

```
// Handles ?sort=name,-weight
Route::get('dogs', function (Request $request) {
    // 获取查询参数并存入用逗号分隔的数组中
    $sorts = explode(',', $request->input('sort', ''));

    // 创建查询
    $query = Dog::query();

    // 按顺序添加排序
    foreach ($sorts as $sortCol) {
        $sortDir = starts_with($sortCol, '-') ? 'desc' : 'asc';
        $sortCol = ltrim($sort, '-');

        $query->orderBy($sortCol, $sortDir);
    }
});
```

```
// 返回
return $query->paginate(20);
});
```

上面这个过程就不那么简单了，也许你还想要在重复进程周围构建一些帮助工具。不用担心，我们会使用逻辑和简单的特性，一步一步地构建自定义的 API。

过滤 API 结果

在创建 API 时，另一个常见的工作是过滤数据的指定子集。例如用户只想要获取雌性狗的列表。

JSON API 并没有提供任何对应的语法，所以需要使用 `filter` 查询参数。让我们对比一下排序的语法，在排序的时候，把所有的数据都放到一个单独的键里，如 `?filter=sex:female`。示例 13-13 展示了怎样完成此设置。

示例13-13 API结果的单次筛选

```
Route::get('dogs', function (Request $request) {
    $query = Dog::query();

    if ($request->has('filter')) {
        list($criteria, $value) = explode(':', $request->input('filter'));
        $query->where($criteria, $value);
    }

    return $query->paginate(20);
});
```

在示例 13-14 中进行了多条件筛选，如 `?filter=sex:female,color:brown`。

示例13-14 API结果的多条件筛选

```
Route::get('dogs', function (Request $request) {
    $query = Dog::query();

    if ($request->has('filter')) {
        $filters = explode(',', $request->input('filter'));
        foreach ($filters as $filter) {
            list($criteria, $value) = explode(':', $filter);
            $query->where($criteria, $value);
        }
    }
});
```

```
return $query->paginate(20);
});
```

数据转换

我们已经了解了怎样对 API 结果进行排序和筛选。但是现在所有的操作都是基于 Eloquent 的 JSON 序列的，也就是说必须从每个模型中获取每一个字段。

Eloquent 提供了一些便利的工具来定义序列化数组时显示的字段。更多内容可以查阅第 8 章。而这里的要点是，如果在 Eloquent 类中设置了一个 `$hidden` 数组属性，则该数组中的任何字段都不会在序列化的模型输出中显示出来。也可以设置 `$visible` 属性以便定义允许显示的字段。此外还可以在模型上重写或者模拟 `toArray()` 方法来自定义输出格式。

另一个常见的模式是为每一个数据类型创建转换器。有一个非常强大的包 Fractal (<http://bit.ly/2fEt8Nr>)，它包含一系列非常便利的、用于数据转换的结构和类。让我们先来看一个简单的转换器的实现过程。

编写自己的转换器

转换器的大体思路是在另一个将数据转换成不同状态的类中运行模型里的每一个实例。它可能会添加字段、重命名字段、删除字段、操作字段、添加嵌套子字段等。让我们先来看一个简单的例子，如示例 13-15 所示。

示例13-15 简单的转换器

```
Route::get('users/{id}', function ($userId) {
    return (new UserTransformer(User::findOrFail($userId)));
});
```

```
class UserTransformer
{
    protected $user;

    public function __construct($user)
    {
        $this->user = $user;
    }

    public function toArray()
    {
        return [
```



```

        'id' => $this->user->id,
        'name' => sprintf(
            "%s %s",
            $this->user->first_name,
            $this->user->last_name
        ),
        'friendsCount' => $this->user->friends->count()
    ];
}

public function toJson()
{
    return json_encode($this->toArray());
}

public function __toString()
{
    return $this->toJson();
}
}

```



经典转换器

经典的转换器一般都包含可以带有 `$user` 参数的 `transform()` 方法。它可能会直接返回一个数组或 JSON。

我使用这个模式已经几年了，有时候我们也把它叫作“API 对象”。我非常喜欢它强大的功能和灵活性。

如示例 13-15 所示，转换器接收需要转换的模型参数，然后操作这个模型以及它内部的关系生成希望发送给 API 的最终输出。

将 API 相关的逻辑与模型本身分离开，这样就可以对 API 进行更好的控制，即使模型还有它的关系发生了变化，依旧可以提供具备一致性的 API。

嵌套和关系

能不能以及怎样在 API 中嵌套关系，一直是大家热议的话题。推荐大家阅读 Phil Sturgeon 所著的 *Build APIs You Won't Hate* (<https://apisyouwonthate.com/>) (Leanpub)，学习 API 中的嵌套以及 REST API。

关系嵌套有几个主要的办法。在下面的例子中，假设主资源是用户 (`user`)，关联资源

是朋友 (friend)。

- 将关联资源直接嵌入主资源（如可以将关联资源嵌套在 users/5 资源中）
- 在主资源中仅嵌入外键（如 users/5 资源嵌套了一组朋友 ID）
- 允许用户查询被原始资源过滤的关联资源（如 /friends?user=5 或者“返回所有与第五个用户相关联的朋友”）
- 新建一个次级资源（如 /users/5/friends）
- 允许选择性嵌入（如 /users/5 没有被嵌入，但是 /users/5?embed=friends 被嵌入，另外还有 /users/5?embed=friends,dogs 也会被嵌入）

假设想要（选择性地）嵌入关系，要怎样做呢？示例 13-15 中的转换器实例是一个很好的启发。让我们来看看示例 13-16 中添加的选择性嵌入。

示例13-16 在转换器中允许资源的选择性嵌入

```
Route::get('users/{id}', function ($userId, Request $request){
    // 获取嵌入查询参数并用逗号隔开
    $embeds = explode(',', $request->input('embed', ''));
    // 将用户和 embeds 传给用户转换器
    return (new UserTransformer(User::findOrFail($userId), $embeds));
});

class UserTransformer
{
    protected $user;
    protected $embeds;

    public function __construct($user, $embeds = [])
    {
        $this->user = $user;
        $this->embeds = $embeds;
    }

    public function toArray()
    {
        $append = [];

        if (in_array('friends', $this->embeds)) {
            // 如果有不止一个嵌入，你可能会想要生成这个
            $append['friends'] = $this->user->friends->map(function ($friend) {
                return (new FriendTransformer($friend))->toArray();
            });
        }
    }
}
```

```

        return array_merge([
            'id' => $this->user->id,
            'name' => sprintf(
                "%s %s",
                $this->user->first_name,
                $this->user->last_name
            )
        ], $append);
    }
    ...

```

在第 17 章学习集合的时候会进一步介绍 `map()` 功能，但是首先需要熟悉本章中的内容。

在路由中用逗号将 `embed` 查询参数分开，然后将它传给转换器。现在转换器只能处理嵌入的 `friends`，以及抽象处理其他信息。如果用户已经请求了 `friends` 嵌入，那么转换器会映射每一个 `friend` (`user` 模型中有多个 `friend` 的关系)，将这个 `friend` 传给 `FriendTransformer`，然后将所有转换的 `friends` 数组嵌入用户响应。

使用 Laravel Passport 的 API 认证

大多数 API 需要不同形式的认证来访问部分或全部数据。Laravel 5.2 中引进了简单的“令牌”认证方案，稍后会介绍这个知识点。Laravel 5.3 中有了一个新工具，叫作 Passport（通过单独的包 `Composer` 引入），这样就可以在应用中简单设置一个功能齐全的 OAuth 2.0 服务器。该服务器包含一个 API 和 UI 组件，用于管理客户端和令牌。Passport 和它相关的特性只适用于 Laravel 5.3 及以上的版本。

OAuth 2.0 简介

OAuth 是 REST 风格 API 中最常见的认证系统。但是在这里我们没办法深入进行讲解，如果大家有兴趣的话，可以翻阅由 `MattFrost` 编写的一本非常著名的书 *Integrating Web Services with OAuth and PHP*。

5.2 如果使用的是 Laravel 5.1 或 Laravel 5.2，则可以使用为 Laravel 提供的 OAuth 2.0 Server 包 (<http://bit.ly/2e2IFYi>)，它能在 Laravel 应用中很容易地添加一个基本的 OAuth 2.0 认证服务器。Laravel 还提供了一个与 PHP 连接的包，PHP OAuth 2.0 Server (<http://bit.ly/2f1dUyP>)。

5.3 如果使用的是 Laravel 5.3，Passport 会通过一个简单但强大的 API 和交互界面提供刚才提到的包中所有的功能，并且还有别的扩展。

安装 Passport

Passport 是一个分离的包，所以需要先安装它。这里总结了简单的步骤，你还可以查阅官方文档了解更详细的安装流程。

首先通过 Composer 引入，如下所示。

```
composer require laravel/passport
```

然后将 `Laravel\Passport\PassportServiceProvider::class` 添加到提供商数组 `config/app.php` 中。这样每次加载应用时，Passport 就会启动。

Passport 导入了一系列迁移，所以可以通过 `php artisan migrate` 命令运行这些迁移，创建用于 OAuth 客户端作用域和令牌的必备的表。

接着使用 `php artisan passport:install` 命令运行安装器。命令会为 OAuth 服务器创建加密 (`storage/oauth-private.key` 和 `storage/oauth-public.key`)，然后将 OAuth 客户端插入个人和密码授权类型令牌的数据库（稍后会讲到）。

这里需要将 `Laravel\Passport\HasApiTokens` 特性导入 `User` 模型，这样就为每个用户添加了 OAuth 客户端和与令牌相关的关系，以及与令牌相关的帮助方法。然后在 `AuthServiceProvider` 的 `boot()` 方法中添加一个调用到 `Laravel\Passport\Passport::routes()` 中。这个过程会为我们添加下面的路由。

- `oauth/authorize`
- `oauth/clients`
- `oauth/clients/{client_id}`
- `oauth/personal-access-tokens`
- `oauth/personal-access-tokens/{token_id}`
- `oauth/scopes`
- `oauth/token`
- `oauth/token/refresh`
- `oauth/tokens`
- `oauth/tokens/{token_id}`

最后在 `config/auth.php` 中找到 `api` 保护。默认情况下，这个保护会用于令牌驱动（马上会介绍这部分内容），但是我们需要将它改为 `passport` 驱动。

现在已经有了功能齐全的 OAuth 2.0 服务器，便可以使用命令 `php artisan passport:client` 来创建新的客户端了，而且还有一个 API 来管理 `/oauth` 路由前缀下可用的客户端和令牌。

为了保护 Passport 认证系统中的路由，可以在路由或路由组中添加 `auth:api` 中间件，如示例 13-17 所示。

示例13-17 通过Passport认证中间件保护API路由

```
// routes/api.php
Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:api');
```

为了认证这些受保护的路由，客户端应用程序需要在 `Authorization` 头中传入一个令牌（后面会介绍如何获得该令牌），作为 `Bearer` 令牌。示例 13-18 展示了使用 Guzzle HTTP 库发送请求的效果。

示例13-18 使用Bearer令牌发送实例API请求

```
$http = new GuzzleHttp\Client;
$response = $http->request('GET', 'http://speakr.dev/api/user', [
    'headers' => [
        'Accept' => 'application/json',
        'Authorization' => 'Bearer ' . $accessToken,
    ],
]);
```

下面了解一下它是如何运行的。

Passport 的 API

Passport 在应用中有一个公开的 API 位于 `/oauth` 路由前缀下。这个 API 主要提供了两个功能：第一个是使用 OAuth 2.0 授权流程对用户进行授权（`/oauth/authorize` 和 `/oauth/token`）；第二个是允许用户管理他们的客户端和令牌（其他所有的路由）。

这是一个重要的区别，特别是在对 OAuth 不是很熟悉的情况下。每一个 OAuth 服务器都需要能够在服务器中对用户进行认证，这是整个服务中最核心的点。Passport 还提供了一个 API 用于管理 OAuth 服务器的客户端和令牌的状态。也就是说，这样就可以很容易地构建前端，以使用户在 OAuth 应用中管理信息。Passport 实际上是由基于 Vue 的管理组件组成的，可以直接使用它们，也可以用来激发其他灵感。

后面会继续介绍允许管理客户端和令牌的 API 路由，以及 Passport 附带的 Vue 组件，便于更容易地进行管理。先让我们来看看有哪些方法可以通过受 Passport 保护的 API 来对用户进行认证。

Passport 可用的授权类型

Passport 提供了四种不同的用户认证方法。两种是传统的 OAuth 2.0 授权方法（密码授权和授权码授权），另外两种是 Passport 特有的、比较方便的方法（个人令牌和同步器令牌）。

密码授权

密码授权虽然没有授权码授权那么常见，但是它更为简单。如果希望 API 能使用用户名和密码来对用户直接进行认证——例如公司有自己的移动应用程序，它使用的也是已有的 API，这时候就可以使用密码授权。



创建一个密码授权客户端

为了使用密码授权流程，数据库中需要包含一个密码授权客户端。执行命令 `php artisan passport:install` 时会添加一个客户端，但是如果因为某些原因需要生成一个新的密码授权客户端，则可以尝试以下方法。

```
php artisan passport:client --password
```

怎样为该密码授权客户端命名？

```
[My Application Password Grant Client]:
```

```
> SpaceBook_internal
```

成功创建密码授权客户端

当获得密码授权类型之后，只需要一步就可以得到一个令牌将用户凭据发送给 `/oauth/token` 路由，如示例 13-19 所示。

示例13-19 使用密码授权类型发送请求

```
// 假设 SpaceBook 不是一个外部应用（但它实际上是）
// 受信任的内部应用，此处是 SpaceBook 的 routes/web.php
Route::get('speakr/password-grant-auth', function () {
    $http = new GuzzleHttp\Client;

    $response = $http->post('http://speakr.dev/oauth/token', [
        'form_params' => [
            'grant_type' => 'password',
            'client_id' => config('speakr.id'),
            'client_secret' => config('speakr.secret'),
            'username' => 'matt@mattstauffer.co',
            'password' => 'my-speakr-password',
        ],
    ]);
});
```

```

    $thisUsersTokens = json_decode((string) $response->getBody(), true);
    // 对令牌进行操作
});

```

这个路由会返回一个 `access_token` 和 `refresh_token`。现在可以保存这些令牌，通过 API（access token）进行认证，然后请求更多的令牌（refresh token）。

注意用于密码授权类型的 ID 和密码是 Passport 应用程序的 `clients` 数据库表中名为 `Space Book_internal` 的行。

授权码授权

OAuth 2.0 认证流程中最常见同时也是最复杂的就是 Passport。想象一下，假设我们正在开发一个类似 Twitter 的应用，叫作 `Speakr`。再想象另外一个针对科技小说迷的社交网站，叫作 `SpaceBook`。`SpaceBook` 的开发者希望用户能将他们的 `Speakr` 数据嵌入 `SpaceBook` 的新闻推送中。在这种情况下我们就可以在应用中安装 Passport，这样其他的应用，如 `SpaceNBook` 就可以允许他们的共同用户通过 `Speakr` 信息进行认证。

在授权码授权类型中，每个消费性网站（在本例中是 `SpaceBook`）都需要在启用 Passport 的应用中创建一个“客户端”。在大多数情况下，其他网站的管理员会在 `Speakr` 中保存用户账户，然后我们会为它们创建工具使其可以创建客户端。但是对于初学者来说，也可以为 `SpaceBook` 管理员手动创建一个客户端。

php artisan passport:client

为客户端赋的用户 ID 值

> 1 ❶

怎样命名客户端

> SpaceBook

授权后怎样重定向请求

[http://passport.dev/auth/callback]:

> http://spacebook.dev/auth/callback

成功创建客户端

客户端 ID: 3

客户端密码: RiQstsWDqd9SqY3lQhiZF50ulKdw4iPhPAcke03

❶ 在应用中每个客户端都需要指定一个用户。比如 Jill，ID 是 #1，她正在编写 `SpaceBook`，那么她将会成为正在创建的客户端的“所有者”。

现在 SpaceBook 客户端已经有了 ID 和密码。SpaceBook 可以使用这个 ID 和密码来创建工具，当 SpaceBook 想要代表该用户在 Speakr 中调用 API 时，这个工具就可以允许 SpaceBook 的用户（同时也是 Speakr 用户）从 Speakr 中获取认证令牌。示例 13-20 解释了这个原理。（在这个示例以及之后的示例中，都假设 SpaceBook 是一个 Laravel 应用，同时假设已经在返回创建的 ID 和密码的 *config/speakr.php* 中创建了一个文件。）

示例13-20 将用户重定向到OAuth服务器的消费性应用中

```
// 在 SpaceBook 的 routes/web.php 中
Route::get('speakr/redirect', function () {
    $query = http_build_query([
        'client_id' => config('speakr.id'),
        'redirect_uri' => url('speakr/callback'),
        'response_type' => 'code',
    ]);

    // 创建一个类似的字符串
    // client_id={$client_id}&redirect_uri={$redirect_uri}&response_type=code
    return redirect('http://speakr.dev/oauth/authorize?' . $query); });
```

当用户在 SpaceBook 中触发这个路由时，他们会被重定向到 Speakr 应用中的 `/oauth/authorize` Passport 路由。然后他们会看到一个确认界面，可以通过运行下面的命令来使用默认的 Passport 确认界面。

```
php artisan vendor:publish --tag=passport-views
```

这样会把视图发布到 *resources/views/vendor/passport/authorize.blade.php* 中，用户都会看到如图 13-1 所示的界面。

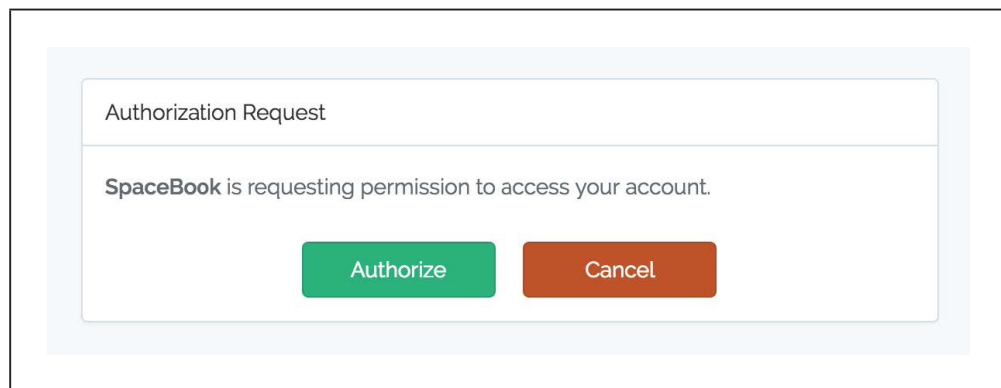


图13-1 OAuth认证码通过界面

一旦用户选择同意或拒绝授权，Passport 会将用户重定向到提供的 `redirect_uri` 中。在示例 13-20 中设置了 `url('speakr/callback')` 的 `redirect_uri`，所以用户会被重定向到 `http://spacebook.dev/speakr/callback` 中。

通过的请求会包含一个码，应用的回调路由就可以使用这个码从启动了 Passport 的应用中返回一个令牌 Speakr。拒绝的请求则会包含一个错误。SpaceBook 的回调路由如示例 13-21 所示。

示例13-21 应用中的授权回调路由

```
// 在 SpaceBook 的 routes/web.php 中
Route::get('speakr/callback', function (Request $request) {
    if ($request->has('error')) {
        // 处理错误情况
    }

    $http = new GuzzleHttp\Client;

    $response = $http->post('http://speakr.dev/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => config('speakr.id'),
            'client_secret' => config('speakr.secret'),
            'redirect_uri' => url('speakr/callback'),
            'code' => $request->code,
        ],
    ]);

    $thisUsersTokens = json_decode((string) $response->getBody(), true);
    // 操作令牌
});
```

这里在 Speakr 中创建一个指向 `/oauth/token` Passport 路由的 Guzzle HTTP 请求。当用户同意访问时，我们发送一个包含接收到的授权码的 POST 请求，Speakr 会返回包含一些键值的 JSON 请求。

- `access_token` 是 SpaceBook 希望为用户保存的令牌。用户在给 Speakr 发送请求时用这个令牌来进行认证。
- `refresh_token` 是当用户决定将令牌设为过期时，SpaceBook 需要使用的令牌。默认 Passport 的请求令牌永远不需要刷新，所以可以忽略它。
- `expires_in` 是 `access_token` 过期（需要被刷新）前的秒数。

使用刷新令牌

如果想强制用户更频繁地进行重认证，那么需要在令牌中设置一个较短的刷新时间，然后使用 `refresh_token` 去请求一个新的 `access_token`。

设置一个较短的刷新时间。

```
// AuthServiceProvider 的 boot() 方法
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    // 令牌需要刷新前的有效时间
    Passport::tokensExpireIn(
        Carbon::now()->addDays(15)
    );
    // 重认证前一个刷新令牌的持续时间
    Passport::refreshTokensExpireIn(
        Carbon::now()->addDays(30)
    );
}
```

想要用刷新令牌来请求一个新的令牌，首先需要像示例 13-21 那样从初始的认证响应中保存 `refresh_token`。当需要刷新的时候，就可以像示例那样进行调用，当然还可以进行微小的修改。

```
// 在 SpaceBook 的 routes/web.php 中
Route::get(
    'speakr/request-refresh',
    function (Request $request) {
        $http = new GuzzleHttp\Client;

        $params = [
            'grant_type' => 'refresh_token',
            'client_id' => config('speakr.id'),
            'client_secret' => config('speakr.secret'),
            'redirect_uri' => url('speakr/callback'),
            'refresh_token' => $theTokenYouSavedEarlier,
```

```
];

$response = $http->post(
    'http://speakr.dev/oauth/token',
    ['form_params' => $params,]
);
$thisUsersTokens = json_decode(
    (string) $response->getBody(),
    true
);

// 操作令牌
}

);
在响应中将收到一组新的令牌，保存到 user 中。
```

现在已经获得了所有在基本的授权码流程中使用的工具。接下来讲解怎样为客户端和令牌创建一个管理员面板。首先来看看其他授权类型。

个人访问令牌

授权码授权适用于用户的应用，密码授权适用于自身的应用，但是如果用户希望能够自己创建令牌来测试 API 或者在开发他们自己的应用时使用令牌，应该采用什么授权类型呢？这时候就应该使用个人令牌了。



创建个人访问客户端

为了创建个人令牌，需要先在数据库中创建一个个人访问客户端。可以通过执行 `php artisan passport:install` 命令来添加该客户端，但是如果需要生成一个新的个人访问客户端，你可以执行命令 `php artisan passport:client --personal`。

```
php artisan passport:client --personal
```

怎样命名密码授权客户端？

```
[My Application Personal Access Client]:
> My Application Personal Access Client
```

成功创建个人访问客户端。

个人访问令牌不是标准的“授权”类型，这里没有 OAuth 规定的流程。可以理解为，个人访问令牌是 Passport 添加的便利方法，它可以轻松地在系统中注册一个客户端，这样

是为了让开发人员能轻松地创建令牌。

例如有一个用户正在开发 SpaceBook 网络的竞争者——RaceBook（为马拉松赛跑者服务），他想先研究一下 Speakr API 以便在开始编码之前弄清楚它是如何工作的。这个开发人员是否具有使用授权码流创建令牌的能力呢？还没有——他还没有写任何代码！这也是使用个人访问令牌的目的。

可以通过 JSON API 创建个人访问令牌，还可以使用下面的代码直接为用户创建一个令牌。

```
// 不通过作用域创建令牌
$token = $user->createToken('Token Name')->accessToken;

// 通过作用域创建令牌
$token = $user->createToken('My Token', ['place-orders'])->accessToken;
```

用户可以像使用通过授权码授权流程创建的令牌一样去使用这些令牌。

Laravel 会话认证的令牌（同步器令牌）

用户获取访问 API 的令牌还有最后一个方法，那就是 Passport 中添加的但常规的 OAuth 服务器中没有提供的便捷方法。此方法适用于用户已经通过了身份验证的情况，因为他们已经正常登录了 Laravel 应用程序。如果希望 Laravel 应用中的 JavaScript 可以访问 API，使用授权码或密码授权流程重新认证用户是一件很困难的事情，所以 Laravel 为此提供了一个助手。

如果在 web 中间件组中添加了 `Laravel\Passport\Http\Middleware\CreateFreshApiToken` 中间件（在 `app/Http/Kernel.php` 中），每一个返回认证用户的 Laravel 响应中都会含有一个叫作 `laravel_token` 的 cookie。这个 cookie 是一个 JSON Web Token (JWT)，它包含关于 CSRF 令牌的编码信息。现在如果通过 JavaScript `diamante` 发送常规的 CSRF 令牌并发送它在 `X-CSRF-TOKEN` 标题中的 API 请求，这个 API 就会将 CSRF 令牌与这个 cookie 进行比较，然后它会像其他令牌一样向 API 验证用户。

JSON Web Token (JWT)

JWT 是一种新的格式。JSON Web Token 是一个包含所有决定用户认证状态和访问权限信息的 JSON 对象。该 JSON 对象使用键值 - 哈希消息认证码 (HMAC) 或 RSA 进行数字签名，这使得它值得信赖。

这个令牌通常都是被编码的，然后通过 URL、POST 请求或者头进行发送。一旦系统认证了某个用户，之后所有的 HTTP 请求都会包含一个描述用户身份和授权的令牌。

JSON Web Token 包括三个用点号 (.) 分隔的 Base64 编码的字符串，例如 xxx.yyy.zzz。第一部分是 Base64 编码的 JSON 对象，它包含了所用哈希算法的信息；第二部分是一系列关于用户授权和身份的“声明”；第三部分是签名，或第三部分依赖于前两部分，使用第一部分中指定的算法进行加密和签名。

更多内容请查阅 JWT.io (<https://jwt.io/>) 或 jwt-auth Laravel 包 (<https://github.com/tymondesigns/jwt-auth>)。

Laravel 附带的默认 Vue 设置为你设置的头，但是如果使用的是不同的框架，那就需要进行手动设置了。示例 13-22 展示了如何在 jQuery 中使用该令牌。

示例13-22 设置jQuery以便传递所有AJAX请求的Laravel CSRF令牌

```
$.ajaxSetup({
  headers: {
    'X-CSRF-TOKEN': "{{ csrf_token() }}"
  }
});
```

如果为 web 中间件组添加了 CreateFreshApiTokens 中间件，并且通过每个 JavaScript 请求来传递该头，那么 JavaScript 请求就可以访问 Passport 保护的 API 路由，而不必担心任何授权码或密码授权的复杂性。

使用 Passport API 和 Vue 组件管理客户端和令牌

到现在为止已经介绍了怎样手动创建客户端和令牌，以及怎样作为消费者进行授权。接下来看看 Passport API 建立用户界面元素的特点，这样可以允许用户管理他们的客户端和令牌。

路由

研究 API 路由的最简单方法是查看提供的样本 Vue 组件的工作原理，以及它们所依赖的路由。下面简单介绍一下。

```
/oauth/clients (GET, POST)
/oauth/clients/{id} (DELETE, PUT)
/oauth/personal-access-tokens (GET, POST)
```

```
/oauth/personal-access-tokens/{id} (DELETE)
/oauth/scopes (GET)
/oauth/tokens (GET)
/oauth/tokens/{id} (DELETE)
```

这里有一些实体（客户端、个人访问令牌、作用域和令牌），可以将它们列出来也可以新建（不能新建作用域，因为它是在代码中定义的；也不能新建令牌，因为它是在授权流程中新建的），还可以删除和更新这些实体。

Vue 组件

Passport 中有一套开箱即用的 Vue 组件，可以让用户很容易地管理他们的客户端（已经创建的）、授权客户端（有权限访问它们账户的）和个人访问令牌（用于自身测试）。

执行下面的命令可以将这些组件发布到应用中。

```
php artisan vendor:publish --tag=passport-components
```

现在 *resources/assets/js/components/pass-port* 中已经有了三个新的 Vue 组件。将它们添加到 Vue 的引导程序中，然后就可以在模板中访问这些组件了。示例 13-23 展示了怎样在 *resources/assets/js/app.js* 文件中注册这些组件。

示例13-23 将Passport的Vue组件导入app.js

```
require('./bootstrap');

Vue.component(
  'passport-clients',
  require('./components/passport/Clients.vue')
);

Vue.component(
  'passport-authorized-clients',
  require('./components/passport/AuthorizedClients.vue')
);

Vue.component(
  'passport-personal-access-tokens',
  require('./components/passport/PersonalAccessTokens.vue')
);

const app = new Vue({
  el: 'body'
});
```

现在可以在应用中的任意位置使用这三个组件，如下所示。

```
<passport-clients></passport-clients>
<passport-authorized-clients></passport-authorized-clients>
<passport-personal-access-tokens></passport-personal-access-tokens>
```

`<passport-clients>` 向用户展示创建的所有客户端。也就是说，当 SpaceBook 的创建者登录 Speakr 时就可以看到列出的 SpaceBook 客户端。

`<passport-authorized-clients>` 向用户展示授权访问其账户的所有客户端。也就是说，SpaceBook 和 Speakr 中的任何用户，只要授权给 Speakr 账户，访问 SpaceBook 时就可以看到列出的 SpaceBook。

`<passport-personal-access-tokens>` 向用户展示创建的所有个人访问令牌。也就是说，RaceBook（SpaceBook 的竞争对手）的创建者可以看到自己用来测试 Speakr API 的个人访问令牌。

如果是刚安装了 Laravel，想要测试一下效果，那么需要遵循下面几个步骤。

- 按照本章前面介绍的方法保证 Passport 成功安装
- 在终端运行下列命令

```
— php artisan vendor:publish --tag=passport-components
— npm install
— gulp
— php artisan make:auth
```
- 打开 `resources/views/home.blade.php` 文件并在 `<div class="panel">` 下添加 Vue 组件引用（如 `<passport-clients>`）

如果愿意，可以直接使用这些组件。也可以将组件作为参考，从而了解如何使用 API，然后创建自定义的前端组件。

Passport 作用域

如果对 OAuth 很熟悉，那么你可能已经注意到我们还没有讨论作用域。

到目前为止，我们所涵盖的一切内容都可以通过作用域进行个性化修改，首先快速介绍一下作用域。

在 OAuth 中，范围是定义的一组特权，也就是说它们可以“执行任何操作”。比如如果之前见过 GitHub API 令牌，那么应该注意到有的应用只想访问名称和邮件地址，有的想访问所有的 repo，还有的想访问 gist。刚才提到的每个内容都是“作用域”，它允许用户和应用程序定义应用程序执行其工作所需的访问权限。

在示例 13-24 中可以看到，可以在 `AuthServiceProvider` 的 `boot()` 方法中为应用定义作用域。

示例13-24 定义Passport作用域

```
// AuthServiceProvider
use Laravel\Passport\Passport;
...
public function boot()
{
    ...

    Passport::tokensCan([
        'list-clips' => 'List sound clips',
        'add-delete-clips' => 'Add new and delete old sound clips',
        'admin-account' => 'Administer account details',
    ]);
}
```

在定义了作用域之后，应用程序就可以定义它想要访问的作用域。只需要从作用域字段的“令牌”字段中选取一个用空格分隔的令牌列表添加到初始化的重定向中。如示例 13-25 所示。

示例13-25 请求授权访问特定作用域

```
// 在 SpaceBook 的 routes/web.php 文件中
Route::get('speakr/redirect', function () {
    $query = http_build_query([
        'client_id' => config('speakr.id'),
        'redirect_uri' => url('speakr/callback'),
        'response_type' => 'code',
        'scope' => 'list-clips add-delete-clips'
    ]);

    return redirect('http://speakr.dev/oauth/authorize?' . $query);
});
```

当用户尝试对应用进行授权时，它会显示请求的作用域列表。这个时候用户就可以发现“SpaceBook 请求查看你的邮件地址”和“SpaceBook 正在请求访问，删除帖子并向朋友发送消息”这两项的不同之处。

可以查看中间件或者 `User` 实例中的作用域。

示例 13-26 展示了怎样查看 `User` 实例。

示例13-26 检查一个用户认证的令牌是否能执行给定的操作

```
Route::get('/events', function () {
    if (auth()->user()->tokenCan('add-delete-clips')) {
        //
    }
});
```

`scope` 和 `scopes` 是两个可以使用的中间件。为了使用它们，需要先将它们添加到 `app/Http/Kernel.php` 文件的 `$routeMiddleware` 中。

```
'scopes'=>\Laravel\Passport\Http\Middleware\CheckScopes::class,
'scope'=>\Laravel\Passport\Http\Middleware\CheckForAnyScope::class,
```

现在可以像示例 13-27 这样使用中间件了。`scopes` 要求所有定义的作用域都要在用户的令牌上,这样用户可以访问路由。而 `scope` 要求至少有一个定义的作用域在用户的令牌上。

示例13-27 使用中间件来限制基于令牌作用域的访问

```
// routes/api.php
Route::get('clips', function () {
    // 访问包含 "list-clips" 和 "add-delete-clips" 作用域的令牌
})->middleware('scopes:list-clips,add-delete-clips');
```

// 或者

```
Route::get('clips', function () {
    // 访问包含至少一个所列作用域的令牌
})->middleware('scope:list-clips,add-delete-clips')
```

如果还没有定义任何作用域,那么可能会认为它们不存在。实际上使用作用域的时候,应用就需要明确地定义以上请求访问的作用域。有种例外情况是,如果使用的是密码授权类型,应用就可以请求 `*` 作用域,这个作用域可以让它的令牌访问所有内容。

Laravel 5.2 以上版本的 API 令牌认证

5.2 Laravel 5.2 中引入了一个简单的 API 令牌认证机制。这与用户名和密码差别不大:有一个单独的令牌分配给每个用户,客户端可以传递一个请求来验证该用户的请求。

但是这个 API 令牌机制没有 OAuth 2.0 那么安全,所以在决定使用这个令牌之前一定要确保它适合你的应用。而且即使确保它很安全,实现起来也并不容易。

首先在 `users` 表中添加一个字符 60 的唯一 `api_token` 列,如下所示。

```
$table->string('api_token', 60)->unique();
```

接着更新所有创建新用户的方法，并确保为每个新用户设置此字段的值。Laravel 为生成随机字符串提供了助手，如果想要使用它，只需要为每个用户将字段设置为 `str_random(60)`。如果想要将其添加到实际应用程序中，还需要对预先存在的用户执行同样的操作。

想要使用这个认证方法包装路由，需要使用 `auth:api` 路由中间件，如示例 13-28 所示。

示例13-28 在路由组中使用API认证中间件

```
Route::group(['prefix' => 'api', 'middleware' => 'auth:api'], function () {  
    //  
});
```

注意：因为这里使用的是认证保护而不是标准保护，所以需要在 `auth()` 方法时声明这个保护，如下所示。

```
$user = auth()->guard('api')->user();
```

测试

幸运的是，实际上在 Laravel 中测试 API 几乎比其他任何测试 API 的方法都更简单。

我们在第 12 章中已经进行了详细的介绍，此处还有一系列方法可以对 JSON 进行断言。再加上全栈应用程序测试的简单性，这些都让 API 测试变得简单。下面通过示例 13-29 来看一个常见的 API 测试模式。

示例13-29 一个常见的API测试模式

```
...  
class DogsApiTest extends TestCase  
{  
    use WithoutMiddleware, DatabaseMigrations;  
  
    public function test_it_gets_all_dogs()  
    {  
        $this->be(factory(User::class)->create());  
        $dog1 = factory(Dog::class)->create();  
        $dog2 = factory(Dog::class)->create();  
  
        $this->visit('api/dogs');  
        $this->seeJson([  
            'name' => $dog1->name  
        ]);  
        $this->seeJson([
```

```
        'name' => $dog2->name
    });
}
}
```

需要注意的是，这里使用了 `WithoutMiddleware` 来避免认证问题（更多关于认证的内容请参考第 9 章）。

这段代码生成了一个用户并使用 `$this->be()` 进行身份验证。接着将两个 dog 插入数据库，然后访问 API 路由列出所有的 dog 并且保证在输出中正常显示这两个 dog。

使用该方法可以简单轻松地覆盖所有 API 路由，包括修改 POST 和 PATCH 等操作。

本章小结

Laravel 面向构建 API，它使 JSON 和类 REST 的 API 工作变得简单。Laravel 中包括一些会话（如分页），但是大部分定义主要用于定义 API 将如何排序、验证，或者实现其他自定义功能。

Laravel 提供了很多工具，可以进行认证和测试，完成简单的头操作和读取 JSON 操作，甚至当 Eloquent 结果直接从路由返回时，Laravel 会自动将其编码成 JSON 格式。

Laravel Passport 是一个单独的包，能让你更加简单地在 Laravel 应用程序中创建和管理 OAuth 服务器。

存储和检索

本书的第 8 章讲解了怎样在关系型数据库中存储数据，但是这里还有更多的存储方式，包括本地存储和远程存储。本章会讲解文件系统、内存存储、文件上传操作、非关系型数据存储、会话、缓冲 cookie 和全文本搜索。

本地和云端文件管理器

Laravel 通过 `storage facade` 提供了一系列文件操作工具，以及一些帮助方法。

Laravel 的文件系统访问工具可以连接本地文件系统以及 S3、Rackspace 和 FTP。S3 和 Rackspace 文件驱动器由 Flysystem 提供，我们可以很容易地在 Laravel 应用中添加额外的 Flysystem 提供商，如 Dropbox 或 WebDAV。

配置文件访问

Laravel 的文件管理器定义位于 `config/filesystems.php` 中。每个连接都被称作“磁盘”，示例 14-1 列出了 Laravel 自带的、可用的磁盘。

示例14-1 默认可用的存储磁盘

```
...
'disks' => [
    'local' => [
        'driver' => 'local',
        'root' => storage_path('app'),
    ],
    'public' => [
        'driver' => 'local',
        'root' => storage_path('app/public'),
        'visibility' => 'public',
    ],
],
```

```
's3' => [
    'driver' => 's3',
    'key' => 'your-key',
    'secret' => 'your-secret',
    'region' => 'your-region',
    'bucket' => 'your-bucket',
],
],
```



storage_path() 助手

示例 14-1 中用到的 `storage_path()` 助手链接了 Laravel 中已配置的存储目录 `storage /`。任何传入 `storage_path()` 的内容都会被添加到目录名的末尾，所以 `storage_path('public')` 会返回字符串 `storage/public`。

`local` 磁盘与本地存储系统相连，并假设它将与存储路径的 `app` 目录，也就是 `storage/app`，进行交互。

`public` 磁盘也是一种本地磁盘（尽管可以根据需要进行更改），它适用于应用程序的任何文件。该磁盘默认位于 `storage/app/public` 目录下，如果想要将这个目录用于向公共磁盘提供文件，则需要在 `public/directory` 中添加一个符号链接（symlink）。Laravel 提供了一行完成此操作的 Artisan 命令，如下所示。

```
# 映射 public/storage, 从 storage/app/public 中提供文件
php artisan storage:link
```

`s3` 磁盘展示了 Laravel 怎样连接基于云端的文件存储系统。如果曾经连接过 S3 或者其他的云端存储提供商，你应该对这个知识点很熟悉：传入密钥、密码，以及一些定义正在使用的“文件夹”的信息，也就是 S3 的区域和存储桶。

使用存储 facade

可以在 `config/filesystem.php` 中设置默认磁盘，在调用 Storage facade 而不指定磁盘时就会使用默认磁盘。在 facade 中调用 `disk('diskname')` 可以指定一个磁盘，代码如下。

```
Storage::disk('s3')->get('file.jpg');
```

每个文件系统都提供了下列方法。

```
get('file.jpg')
```

检索文件 `file.jpg`。

```
put('file.jpg', $contentsOrStream)
```

将给定的文件内容放到 *file.jpg* 中。

```
putFile('myDir', $file)
```

将给定文件的内容（Illuminate\Http\File 或 Illuminate\Http\UploadedFile 实例的格式）放到 *myDir* 目录中，Laravel 管理整个流程并命名文件。

```
exists('file.jpg')
```

返回表示 *file.jpg* 是否存在的布尔值。

```
copy('file.jpg', 'newfile.jpg')
```

将 *file.jpg* 复制到 *newfile.jpg* 中。

```
move('file.jpg', 'newfile.jpg')
```

将 *file.jpg* 移动到 *newfile.jpg* 中。

```
prepend('my.log', 'log text')
```

将内容添加到 *my.log* 的开始位置。

```
append('my.log', 'log text')
```

将内容添加到 *my.log* 的结尾位置。

```
delete('file.jpg')
```

删除 *file.jpg*。

```
deleteDirectory('myDir')
```

删除 *myDir*。

```
size('file.jpg')
```

返回 *file.jpg* 的字节大小。

```
lastModified('file.jpg')
```

返回最近一次修改 *file.jpg* 的 UNIX 时间戳。

```
files('myDir')
```

在 *myDir* 目录中返回一个文件名称数组。

```
allFiles('myDir')
```

在 *myDir* 目录及其子目录中返回一个文件名称数组。

```
directories('myDir')
```

在 *myDir* 目录中返回一个目录名称数组。

```
allDirectories('myDir')
```

在 *myDir* 目录及其子目录中返回一个目录名称数组。



注入实例

如果希望注入一个实例而不使用 `File facade`、类型提示或注入 `Illuminate\Filesystem\Filesystem`，那么同样可以得到所有的方法。

添加额外的 Flysystem 提供商

如果想要添加一个额外的 Flysystem 提供商，则需要“继承”Laravel 的本地存储系统。服务提供商中包含了 `AppServiceProvider` 的 `boot()` 方法，但是使用 `Storage facade` 来添加新的存储系统为每个新的绑定创建一个唯一的服务提供商，这样更为合适，如示例 14-2 所示。

示例14-2 添加额外的Flysystem提供商

```
// 某些服务提供商
public function boot()
{
    Storage::extend('dropbox', function ($app, $config){
        $client = new DropboxClient(
            $config['accessToken'], $config['clientIdentifier']
        );

        return new Filesystem(new DropboxAdapter($client));
    });
}
```


基本的文件上传和操作

Storage facade 中最常见的一个应用就是从用户那里接受文件上传。下面通过示例 14-3 来看看常见的工作流程。

示例14-3 常见的文件上传流程

```
...
class DogsController
{
    public function updatePicture(Request $request, Dog $dog)
    {
        Storage::put(
            'dogs/' . $dog->id,
            file_get_contents($request->file('picture')->getRealPath())
        );
    }
}
```

这里用 `put()` 方法把内容放到一个名为 `dogs/{id}` 的文件中并且从上传的文件中获取内容。每个上传的文件都继承自 `SplFileInfo` 类，这个类提供了一个返回文件路径的 `getRealPath()` 方法。所以可以得到用户上传文件暂时的上传路径，通过 `file_get_contents()` 方法读取，然后将它传入 `Storage::put()`。

现在得到了这个可用的文件，在存储之前我们可以对其进行任意操作——如果是一张图片，则可以使用图片操作包来重新设置图片大小、验证该文件；如果不符合我们的标准，则可以拒绝该文件的上传以及其他任何操作。

在 `config/filesystems.php` 中已经存储了证书的情况下，如果想要将同样的文件上传到 S3，则只需要调整示例 14-3 来调用 `Storage::disk('s3')->put()`，这样就可以将文件上传到 S3。我们来看一个更复杂的例子，如示例 14-4 所示。

示例14-4 使用Intervention进行更复杂的文件上传

```
...
class DogsController
{
    public function updatePicture(Request $request, Dog $dog)
    {
        $original = $request->file('picture');

        // 将图像大小调整为最大宽度 150
        $image = Image::make($original)->resize(150, null, function ($constraint) {
            $constraint->aspectRatio();
        }->encode('jpg', 75);
    }
}
```

```

        Storage::put(
            'dogs/thumbs/' . $dog->id,
            $image->getEncoded()
        );
    }

```

在示例 14-4 中使用了一个名为 Intervention 的图片包，也可以使用其他软件包。大家需要记住的一点就是，在存储文件之前，可以对其进行任何操作。



对上传文件使用 `store()` 和 `storeAs()` 方法

5.3 Laravel 5.3 引入了使用文件本身存储上传文件的功能，参见示例 6-11。

会话

在 Web 应用中，会话存储是用于在界面请求之间存储状态的主要工具。Laravel 的会话管理器通过文件、cookie、数据库、Memcached、Redis 或内存中的数组（页面请求后到期，并且仅适用于测试）来支持会话驱动。

可以在 `config/session.php` 中配置所有的会话设置和驱动器。也可以选择是否加密会话数据、使用哪个驱动器（默认是 `file` 驱动器），以及指定更多特定连接的详细信息，如会话存储长度以及要使用的文件或数据库表。查看会话文档（<https://laravel.com/docs/master/session>），可以了解需要选择使用驱动器准备的、特定的依赖和设置。

会话工具的常用 API，允许通过唯一键值来保存和检索数据。例如，`session()->put('user_id')` 和 `session()->get('user_id')`。因为 Laravel 仅在内部使用闪存（仅适用于下一页请求）进行会话存储，所以应当避免将内容保存到闪存会话密钥中。

访问会话

使用会话 facade 是最常见的访问会话的方法，如下所示。

```
session()->get('user_id');
```

也可以在给定的 Illuminate 响应对象上使用 `session()` 方法，如示例 14-5 所示。

示例 14-5 在响应对象上使用 `session()` 方法

```

Route::get('dashboard', function (Request $request){
    $request->session()->get('user_id');
});

```

或者也可以注入一个 Illuminate\Session\Store 实例，如示例 14-6 所示。

示例14-6 为会话注入支持类

```
Route::get('dashboard', function (Illuminate\Session\Store $session) {  
    return $session->get('user_id');  
});
```

最后可以使用全局 session() 助手。使用它时不添加任何参数便可以得到一个会话实例，可以通过单字符串参数从会话中“获取”数据，或者通过一个数组将数据“放入”会话中，如示例 14-7 所示。

例14-7 使用全局session()帮手

```
// get  
$value = session()->get('key');  
$value = session('key');  
// put  
session()->put('key', 'value');  
session(['key', 'value']);
```

如果刚开始使用 Laravel 不清楚应该使用哪一种办法，那么推荐使用全局助手。

会话实例的可用方法

会话中最常见的两个方法就是 get() 和 put()，首先来看看每一个方法以及它们的参数，如下所示。

`session()->get($key, $fallbackValue)`

get() 从会话中获取提供的键值。如果该键中不含值，则会返回回调值（如果没有提供回调值，则 get() 会返回 null）。回调值可以是一个字符串或一个闭包，如下所示。

```
$points = session()->get('points');  
  
$points = session()->get('points', 0);  
  
$points = session()->get('points', function () {  
    return (new PointGetterService)->getPoints();  
});
```

`session()->put($key, $value)`

put() 将会话中提供的值存储到对应的键值中。

```
session()->put('points', 45);
```

```
$points = session()->get('points');
```

```
session()->push($key, $value)
```

如果会话的值是数组，则可以使用 `push()` 方法向数组中添加元素，如下所示。

```
session()->put('friends', ['Saúl', 'Quang', 'Mechteld']);
```

```
session()->push('friends', 'Javier');
```

```
session()->has($key)
```

`has()` 检查给定键值中是否包含值。

```
if (session()->has('points')) {  
    // 进行操作  
}
```

也可以传入一个键值数组，如果所有键值都已经存在，则会返回 `true`。



`session()->has()` 和 `null`

如果一个会话值已经被设为 `null`，那么 `session()->has()` 将会返回 `false`。

```
session()->all()
```

`all()` 返回会话中所有内容的数组，包括在框架中设置的值。这里可能会看到键下的值，如 `_token` (CSRF 令牌)、`_previous` (`back()` 重定向) 和 `flash` (用于闪存存储) 等。

```
session()->forget($key) 和 session()->flush()
```

`forget()` 删除之前设置的会话的值。`flush()` 删除每个会话的值，包括在框架里设置的值，如下所示。

```
session()->put('a', 'awesome');  
session()->put('b', 'bodacious');
```

```
session()->forget('a');  
// a 不再是一个集合，b 仍是一个集合  
session()->flush();
```

```
// 会话不再为空
```

```
session()->pull($key, $fallbackValue)
```

`pull()` 和 `get()` 类似，不同在于 `pull()` 获取值之后删除该值。

```
session()->regenerate()
```

这个方法并不常见，用于重新生成会话 ID。

闪存会话存储

我们还有三个方法没有讲解，这三个方法都用于处理叫作“闪存”的会话存储。

会话存储中非常常见的模式，是设置一个只能用于下一页加载的值。比如可能想要存储如“成功更新帖子”这样的信息。可以手动获取该消息，然后在下一次加载时擦除它。但是如果经常使用该模式，则可能造成很多浪费。若输入闪存会话存储，键（key）预计只能持续一个页面请求。

Laravel 实现了这个功能，需要使用 `flash()` 方法而不是 `put()` 方法，如下所示。

```
session()->flash($key, $value)
```

`flash()` 将下一个界面请求提供的值设置为会话的键值。

```
session()->reflash() 和 session()->keep($key)
```

如果需要前一个界面的 `flash()` 闪存数据依附于多个请求，则可以使用 `reflash()` 来为下一个请求存储全部闪存内容或者使用 `keep($key)` 为下一个请求存储单个闪存值。`keep()` 也可以接收一个键值数组从而重新保存闪存数据。

高速缓存器 cache

`cache` 的结构与会话非常类似。如果提供一个键值，那么 Laravel 会保存该值。二者最大的不同在于，高速缓存中的数据是为每个应用缓存的，而会话是为每个用户缓存的。也就是说，在进行存储大量的数据库结果、API 调用或者其他慢查询时更常使用 `cache`。

`cache` 的配置设置位于 `config/cache.php` 中。和会话一样，可以为任何驱动器进行特定的配置，也可以指定默认驱动。Laravel 默认使用 `file` 高速缓存驱动器，也可以使用 `Memcached`、`Redis`、API 数据库或者编写自己的高速缓存驱动器。可以通过查看缓存文档（<https://laravel.com/docs/master/cache>）了解驱动程序所需的特定依赖项和设置。

访问高速缓存

跟会话一样，访问高速缓存器也有一些不同的方法。可以使用 facade，如下所示。

```
$users = Cache::get('users');
```

可以从容器中获得一个实例，如示例 14-8 所示。

示例14-8 注入一个高速缓存实例

```
Route::get('users', function (Illuminate\Contracts\Cache\Repository $cache) {  
    return $cache->get('users');  
});
```

也可以使用全局 `cache()` 助手（Laravel 5.3 的新特性），如示例 14-9 所示。

示例14-9 使用全局`cache()`助手

```
// 从 cache 中获取  
$users = cache('key', 'default value');  
$users = cache()->get('key', 'default value');  
// 放置持续 $minutes 时间  
$users = cache(['key' => 'value'], $minutes);  
$users = cache()->put('key', 'value', $minutes);
```

如果刚开始使用 Laravel，不清楚应该使用哪一种办法，那么推荐使用全局助手。

Cache 实例中可用的方法

来看看在 Cache 实例中可以使用的方法，如下所示。

`cache()->get($key,$fallbackValue)` 和 `cache()->pull($key,$fallbackValue)`

`get()` 能更容易地从给定的键中获取值。`pull()` 和 `get()` 相似，不同在于 `pull()` 在获取高速缓存的值后会将其删除。

`cache()->put($key,$value,$minutesOrExpiration)`

`put()` 设置给定分钟数的指定键值。如果想要设置过期日期 / 时间而不是分钟数，则可以将 Carbon 对象作为第三个参数传入。

```
cache()->put('key', 'value', Carbon::now()->addDay());
```

`cache()->add($key,$value)`

`add()` 与 `put()` 类似，不同之处在于，如果值已经存在，`add()` 不会再次设置该值。

`add()` 同样会返回一个表示值是否已经被添加的布尔值。

```
$someDate = Carbon::now();  
cache()->add('someDate', $someDate); // 返回 true  
$someOtherDate = Carbon::now()->addHour();  
cache()->add('someDate', $someOtherDate); // 返回 false
```

`cache()->forever($key,$value)`

`forever()` 将一个值保存到特定键的缓存中，这与 `put()` 类似，但是 `forever()` 中的值永远不会过期（直到使用 `forget()` 方法删除）。

`cache()->has($key)`

`has()` 返回一个表示提供的键是否有值的布尔值。

`cache()->remember($key,$minites,$closure)` 和 `cache()->rememberForever ($key,$closure)`

`remember()` 提供了一个处理常规流程的方法：查询缓存中的指定键是否有值，如果不含有值就获取该值并保存到缓存中，再将其返回。

`remember()` 需要提供一个查询用的键、需要保存的分钟数和定义如何查询的闭包，以防键中不包含需要设置的值。`rememberForever()` 与 `remember()` 类似，但是它不需要设置过期的分钟数。通过下面的例子来看看常见的用于 `remember()` 的使用场景。

```
// 返回 "users" 缓存中的值或获取 "User::all()"  
// 在 "users" 中缓存并返回该值  
$users = cache()->remember('users', 120, function () {  
    return User::all();  
});
```

`cache()->increment($key,$amount)` 和 `cache()->decrement($key,$amount)`

`increment()` 和 `decrement()` 方法允许在缓存中增加或者减少整型值。如果给定的键中不含值，则该值会被当作 0；如果通过第二个参数指定增加或者减少的值，该方法会根据指定的操作加 1 或者减 1。

`cache()->forget($key)` 和 `cache()->flush()`

`forget()` 与会话的 `forget()` 方法类似：首先传入一个键，然后它会删除对应的值。`flush()` 会删除所有的缓存。

cookie

你可能希望 cookie 也和会话、缓存一样。cookie 同样可以使用 facade 和全局助手里面的方法，也可以以同样的方式获取或设定值。

但是因为 cookie 本身就是附加在请求和响应上的，所以与 cookie 的交互方式有所不同。接下来看看为什么不同。

Laravel 中的 cookie

在 Laravel 中，cookie 可以存在于三个地方。首先是在请求中，也就是在访问界面时，用户会获得一个 cookie。可以通过 `Cookie` facade 读取该 cookie，也可以从请求对象中读取。

其次，cookie 也可以发出响应，也就是说响应会指示用户的浏览器保存 cookie 以便将来访问。也可以在返回 cookie 之前，将它添加到响应对象中。

最后，cookie 可以组成队列。如果使用 `Cookie` facade 来设置 cookie，则可以将其放入“CookieJar”队列中，然后 cookie 将会被移除并通过 `AddQueuedCookiesToResponse` 中间件添加到响应对象中。

访问 cookie 工具

可以在三个地方获取和设置 cookie：`Cookie` facade、`cookie()` 全局助手，以及请求和响应对象。

Cookie facade

`Cookie` facade 是功能最齐全的选项，它不仅允许读取和创建 cookie，还能将 cookie 通过队列添加到响应。它提供了下列方法。

`Cookie::get($key)`

获取请求中出现的 cookie 值，可以仅运行 `Cookie::get('cookie-name')`。这是最简单的选项。

`Cookie::has($key)`

可以通过 `Cookie::has('cookie-name')` 检查 cookie 是否与请求一起使用，该 cookie 返回一个布尔值。

Cookie::make(...params)

如果想创建一个 cookie 但不排队，可以使用 `Cookie::make()`。最常见的用法就是创建一个 cookie，然后手动将它附加到响应对象。

以下是 `make()` 的参数。

- `$name` 是 cookie 的名字
- `$value` 是 cookie 的内容
- `$minutes` 指定了 cookie 留存的分钟数
- `$path` 是 cookie 的有效路径
- `$domain` 列出了 cookie 工作的域名
- `$secure` 表示 cookie 是否应该仅通过安全（HTTPS）连接传输
- `$httpOnly` 表明了 cookie 是否只能通过 HTTP 协议访问

Cookie::make()

返回一个 `Symfony\Component\HttpFoundation\Cookie` 实例。



cookie 的默认设置

`Cookie facade` 实例使用 `CookieJar` 从会话配置中读取默认设置。所以如果要在 `config/session.php` 中为会话 cookie 改变任何配置值，同样的默认设置也会被应用到用 `Cookie facade` 创建的所有 cookie 中。

Cookie::queue(Cookie//...params)

如果使用 `Cookie::make()`，依旧需要将 cookie 附加到响应中的这个部分，我们稍后会讲到。`Cookie::queue()` 与 `Cookie::make()` 的语法一样，但是它会将创建的 cookie 排入队列，然后自动附加到中间件的响应中。

如果需要，也可以将创建的 cookie 传入 `Cookie::queue()` 中。

下面是一个在 Laravel 中将 cookie 添加到响应的简单示例。

```
Cookie::queue('dismissed-popup', true, 15);
```



将没有设置的 cookie 进行排队

cookie 只能作为响应的一部分返回。所以如果将 cookie 排队到 `Cookie facade` 中，响应便不能正确返回——例如使用 PHP 的 `exit()` 方法或某些操作停止脚本的执行，那么 cookie 将不会被设置。

cookie() 全局助手

如果不使用参数调用 `cookie()` 全局助手，那么它将会返回一个 `CookieJar` 实例。然而 `Cookie facade` 中最方便的两个方法 `has()` 和 `get()`，只存在于 `facade` 而不包含 `CookieJar`。因此在这种情况下，全局助手实际上比其他选项更有用。

`cookie()` 全局助手的一个重要任务是创建一个 `cookie`。如果将参数传入 `cookie()`，它们会被直接传给 `Cookie::make()`，所以这是创建 `cookie` 最快的方法，如下所示。

```
$cookie = cookie('dismissed-popup', true, 15);
```



注入一个实例

也可以在应用程序的任意位置注入一个 `Illuminate\Cookie\CookieJar` 实例，但是还会遇到以下提到的限制。

请求和响应对象的 cookie

由于 `cookie` 被设置为响应的一部分并且作为请求的一部分被传入，这些 `Illuminate` 对象就会位于它实际应该存在的位置。`Cookie facade` 的 `get()`、`has()` 和 `queue()` 方法只是与请求和响应对象交互的代理。

所以与 `cookie` 交互的最简单方式就是从请求中获取 `cookie`，然后将它们设置到响应中。

从请求对象中读取 cookie

一旦获得一个请求对象的副本——如果不知道怎样获取一个请求对象，可以尝试使用 `app('request')`——便可以使用请求对象的 `cookie()` 方法来读取它的 `cookie`，如示例 14-10 所示。

示例14-10 从请求对象中读取一个cookie

```
Route::get('dashboard', function (Illuminate\Http\Request $request) {  
    $userDismissedPopup = $request->cookie('dismissed-popup', false);  
});
```

正如这个示例一样，`cookie()` 方法包含两个参数：`cookie` 名称和可选参数回调值。

在响应对象中设置 cookie

当响应对象可用的时候，可以使用 `cookie()` 方法（或者 `Laravel 5.3` 之前版本的 `withCookie()` 方法）将一个 `cookie` 添加到响应中，如示例 14-11 所示。

示例14-11 在响应对象中设置cookie

```
Route::get('dashboard', function () {  
    $cookie = cookie('saw-dashboard', true);  
  
    return Response::view('dashboard')  
        ->cookie($cookie);  
});
```

如果刚开始使用 Laravel，不太清楚应该使用哪个选项，那么推荐在请求和响应对象中设置 cookie。虽然这样工作量会大一点，但是如果以后的开发者不了解 CookieJar 队列，这样的设置就可以节省很多时间。

基于 Laravel Scout 全文搜索

Laravel Scout 是一个独立的包，可以将它引入 Laravel 应用来为 Eloquent 模型添加全文搜索。Scout 可以很容易地搜索 Eloquent 模型的内容，它随 Algolia 和 Elasticsearch 驱动程序一起提供，但也有其他服务提供商提供了社区版的软件包。这里假设读者使用的是 Algolia。

安装 Scout

首先从 Laravel 5.3 以上版本的应用中获取软件包，如下所示。

```
composer require laravel/scout
```

然后将 `Laravel\Scout\ScoutServiceProvider::class` 添加到 `config/app.php` 的提供商部分。

为了配置 Scout，需要先运行 `php artisan vendor:publish` 命令，然后将 Algolia 证书添加到 `config/scout.php` 中。

最后安装 Algolia SDK，如下所示。

```
composer require algolia/algoliasearch-client-php
```

标记索引模型

在模型中（这里用 Review 模型作为示例）导入 `Laravel\Scout\Searchable` 特性。

可以定义 `toSearchableArray()` 方法（默认为镜像 `toArray()`）和搜索的属性，然后使用 `searchableAs()` 方法（默认是表名）定义模型的索引名称。

Scout 在标记的模型上订阅了 create/delete/update 事件。在创建更新或者删除任意行时，Scout 会将这些修改同步到 Algolia 中。它既能使更新同步更改，也可以在 Scout 配置为使用队列时对排队进行更新。

索引检索

Scout 的语法非常简单。例如检索包含词语 Llew 的 Review，代码如下所示。

```
Review::search('Llew')->get();
```

也可以像修改常规 Eloquent 调用一样修改查询，如下所示。

```
// 获取 Review 中所有与 "Llew" 匹配的记录
// 像 Eloquent 分页一样限制每页只显示 20 条记录读取界面查询参数
Review::search('Llew')->paginate(20);
// 获取 Review 中所有与 "Llew" 匹配的记录
// 并把 account_id 字段设置为 2
Review::search('Llew')->where('account_id', 2)->get();
```

我们会检索出什么内容呢？结果是从数据库中获取的一个 Eloquent 模型的集合。Algolia 中存储的 ID 返回一个匹配的 ID 列表，然后 Scout 从数据库中获取记录并将它们作为 Eloquent 对象返回。

这里并没有完全用到复杂的 SQL WHERE 命令，但就像在代码示例中看到的那样，它比较检索提供了一个基本的框架。

队列和 Scout

到现在为止，应用将在修改数据库记录的每个请求时向 Algolia 发出 HTTP 请求。这样可以快速减缓应用程序进程。这就是 Scout 可以轻松将其所有动作推送到队列中的原因。

在 *config/scout.php* 中，将 `queue` 设为 `true` 就可以将这些更新设置为异步索引。全文本索引目前会保持“最终一致性”，也就是说数据库记录将立即接收到更新，并且搜索索引的更新将会尽可能快地按照队列允许的速度进行排队和更新。

执行无索引操作

如果想要执行一系列操作同时又不希望在响应中触发索引，那么可以在模型的 `withoutSyncingToSearch()` 方法中进行封装操作，代码如下。

```
Review::withoutSyncingToSearch(function () {
    // 如添加一些评论
```

```
factory(Review::class, 10)->create();
});
```

通过代码手动触发索引

如果希望在模型中手动触发索引，则可以在应用中使用代码或者通过命令行触发。

在使用代码手动触发索引时，需要将 `searchable()` 添加到任意 Eloquent 查询的末尾，然后它将会为该查询里的所有记录添加索引。

```
Review::all()->searchable();
```

也可以在查询中选择希望添加索引的作用域。Scout 非常智能，可以查询新的记录、更新旧的记录，所以我们可以选择仅对模型数据库表里的内容重新索引。

也可以在关系方法中指定 `searchable()`，如下所示。

```
$user->reviews()->searchable();
```

如果想要取消查询链中记录的索引，可以使用 `unsearchable()`，如下所示。

```
Review::where('sucky', true)->unsearchable();
```

利用 CLI 手动触发索引

还可以通过 Artisan 命令触发索引，如下所示。

```
php artisan scout:import App\\Review
```

该命令会将 Review 模型分块，并为这些块添加索引。

测试

测试大多数特性与在测试中使用它们一样简单，不需要模拟或者存根，默认配置已经有效——例如查看 *phpunit.xml* 可以看到会话驱动和缓存驱动已经被设置为适合测试的值了。

但是在尝试测试所有这些方法之前，应该了解一些简便的方法。

文件存储

测试文件上传可能有一点麻烦，可以按照下面的步骤进行操作。

上传假文件

首先通过示例 14-12 来看看怎样在系统测试中手动创建一个 Symfony UploadedFile 对象。注意，假设我们已经创建了 *storage/tests* 目录，存放用于测试的 *for-tests.jpg* 文件。

示例14-12 为测试创建一个假的UploadedFile

```
public function test_file_should_be_stored()
{
    $path = storage_path('tests/for-tests.jpg');
    $file = new UploadedFile(
        $path, // 文件路径
        'for-tests.jpg', // 原始文件名
        'image/jpg', // MIME 类型
        filesize($path), // 文件大小，最好硬编码到测试中
        test, null, // 错误代码
        true // 是否处于测试模式下
    );
    $this->call('post', 'upload-route', [], [], ['upload' => $file]);

    $this->assertResponseOk();
}
```

我们已经创建了一个参照测试文件的 UploadedFile 实例，现在可以用它来测试路由。

返回假文件

如果希望路由测试真实的文件，最好的办法就是该文件真的存在。假设每个用户都必须拥有个人资料图片。

首先，设置模型工厂供用户使用 Faker 制作图片的副本，如示例 14-13 所示。

示例14-13 通过Faker返回假文件

```
$factory->define(User::class, function (Faker\Generator $faker) {
    return [
        'picture' => $faker->file(
            storage_path('tests'), // 源目录
            storage_path('app'), // 目标目录
            false // 仅返回文件名，不返回完整路径
        ),
        'name' => $faker->name,
    ];
});
```

Faker 的 `file()` 方法从源目录中随机选取一个文件，将它复制到目标目录，然后返回文件名。我们只需要从 `storage/tests` 目录中随机选择一个文件将它复制到 `storage/app` 目录下，然后将它的文件名设置为 User 的 `picture` 属性即可。这样我们便可以在测试路由时使用这个包含图片的 User，如示例 14-14 所示。

示例14-14 断言图片的URL被输出

```
public function test_user_profile_picture_echoes_correctly()
{
    $user = factory(User::class)->create();

    $this->visit( "users/{$user->id}");
    $this->see($user->picture);
}
```

当然在多数情况下，甚至都不需要复制文件，只需要生成一个随机字符串。但是如果路由需要检查文件的存在或对文件执行某些操作，最好还是选择复制文件的方式。

会话

如果需要对会话中设置的某些内容进行断言，则可以使用 Laravel 为每个测试提供一些便利方法。所以这些方法都可以在测试的 `$this` 对象中使用，如下所示。

```
assertSessionHas($key,$value=null)
```

在断言会话具有特定键值的同时，如果传入了第二个参数，则该键具有特定值。

```
public function test_some_thing()
{
    // 进行操作
    $this->assertSessionHas('key', 'value');
}
```

```
assertSessionHasAll(array $bindings)
```

如果传入了一个键值对数组，则断言所有的键与值相同。如果一个或多个数组元素只是一个值（使用 PHP 默认的数值键），那么它只在会话中被检索是否存在，如下所示。

```
$check = [
    'has',
    'hasWithThisValue' => 'thisValue',
]

$this->assertSessionHasAll($check);
```

```
assertSessionMissing($key)
```

断言该会话不具有特定键的值。

```
assertSessionHasErrors($bindings=[], $forma=null)
```

断言该会话包含一个 `errors` 值,这是 Laravel 用于从失败的验证中返回错误的键值。

如果数组中只包含键,它将检查会话是否使用这些键设置错误。

```
$this->post('test-route', ['failing' => 'data']);  
$this->assertSessionHasErrors(['name', 'email']);
```

可以为这些键传入值,也可以选择性地设置 `$format` 来检查返回的错误信息是否与期待的内容相符。

```
$this->post('test-route', ['failing' => 'data']);  
$this->assertSessionHasErrors([  
    'email' => '<strong>The email field is required.</strong>'  
, '<strong>:message</strong>'];
```

```
assertHasOldInput()
```

这里可以将之前界面的输入闪存到会话中,如果想要断言输入被正确闪存,如下所示。

```
$this->post('test-route', ['failing' => 'data']);  
$this->assertHasOldInput();
```

高速缓存

缓存的测试并没有什么特别,直接测试即可。

```
Cache::put('key', 'value', 15);  
  
$this->assertEquals('value', Cache::get('key'));
```

在测试环境中,Laravel 默认使用“数组”缓存驱动器,将缓存值存在内容中。

cookie

如果在系统测试中测试路由之前需要设置 cookie,那么可以手动将 cookie 作为一个参数传入 `call()` 方法。可以查阅第 12 章学习更多的内容。



在测试期间将 cookie 从加密中排除

如果将 cookie 从 Laravel 的 cookie 加密中间件中移除，那么它们在测试过程中就不会工作。这里可以在 EncryptCookies 中间件中暂时停用该 cookie。

```
use Illuminate\Cookie\Middleware\EncryptCookies;
...

$this->app->resolving(
    EncryptCookies::class,
    function ($object) {
        $object->disableFor('cookie-name');
    }
);
// .....运行测试
```

这意味着可以设置和检索一个像示例 14-15 这样的 cookie。

示例14-15 为cookie运行单元测试

```
public function test_cookie()
{
    $this->app->resolving(EncryptCookies::class, function ($object){
        $object->disableFor('my-cookie');
    });

    $this->call('get', 'route-echoing-my-cookie-value', [], ['my-cookie' => 'baz']);
    $this->see('baz');
}
```

如果不想停用加密，则可以像示例 14-16 一样为 cookie 设置加密值。

示例14-16 在设置前手动加密cookie

```
use Illuminate\Contracts\Encryption\Encrypter;
...

public function test_cookie()
{
    $encryptedBaz = app(Encrypter::class)->encrypt('baz');

    $this->call(
        'get',
        'route-echoing-my-cookie-value',
        [],
        ['my-cookie' => $encryptedBaz]
```

```
);  
$this->see('baz');  
}
```

如果想测试一个已经设置了 cookie 的响应，还可以使用 `seeCookie()` 测试该 cookie，如下所示。

```
$this->visit('cookie-setting-route');  
$this->seeCookie('cookie-name');
```

或者使用 `seePlainCookie()` 来测试，然后断言该 cookie 没有被加密。

本章小结

Laravel 为许多常见的存储操作提供了简单的接口：文件系统、访问会话、cookie 缓存和检索。无论使用的是哪个提供商，每一个 API 都是一样的，Laravel 允许多个“驱动程序”提供相同的公共接口，这使得根据环境来切换所提供程序，或者根据应用程序的需求变化而变化变得更简单。

邮件和通知

通过电子邮件、Slack、SMS 或其他通知系统来发送应用程序的用户通知，这是一个十分常见但却复杂到令人惊讶的要求。Laravel 的邮件和通知功能提供了一个统一的 API，它可以抽象出任何需要格外注意的特别的系统提供商。就像第 14 章中提到的，我们需要编写代码，然后在配置的时候选择使用哪个系统来发送邮件或通知。

邮件

Laravel 的邮件功能在 SwiftMailer (<http://swiftmailer.org/>) 顶部的一个便利层，除此之外，Laravel 中也含有 Mailgun、Mandrill、Sparkpost、SES、SMTP、PHP 邮件和 Sendmail 驱动程序。

对于所有的云服务来说，我们可以在 `config.service.php` 中设置自己的身份验证信息。但是，如果仔细观察，就会发现其中已经包含了一些关键字，并且在 `config/mail.php` 中，可以使用如 `MAIL_DRIVER` 和 `MAILGUN_SECRET` 一样的变量在 `.env` 中自定义应用的邮件功能。



基于云的 API 驱动程序依赖关系

如果正在使用基于云的 API 驱动程序，则需要使用 Composer¹ 将 Guzzle 插入。运行以下命令可以实现。

```
composer require guzzlehttp/guzzle: "~5.3|~6.0"
```

如果使用 SES 驱动程序，则需要运行以下命令。

```
composer require aws/aws-sdk-php:~3.0
```

¹ Composer 指 PHP 中用来管理依赖关系的工具。——译者注

“classic” 邮件

5.2 Laravel 中有两种不同的用于发送邮件的语法：classic（经典）语法和 mailable（可邮寄）语法。mailable 是 Laravel 5.3 版本之后的首选语法，因此我们将在本书中重点介绍此语法。对于使用 Laravel 5.1 或 5.2 版本的用户来说，我们会大致介绍一下 classic 语法是如何工作的，如示例 15-1 所示。

示例15-1 基本“classic”邮件语法

```
Mail::send(
    'emails.assignment',
    ['trainer' => $trainer, 'trainee' => $trainee],
    function ($m) use ($trainer, $trainee) {
        $m->from($trainer->email, $trainer->name);
        $m->to($trainee->email, $trainee->name)->subject('A New Assignment!');
    }
);
```

`Mail::send()` 的第一个参数是视图的名称。需要注意的是，其中的 `emails.assignment` 是指 `resources/views/emails/assignment.blade.php` 或者 `resources/ views/emails/assignment.php`。

第二个参数是要传递给视图的数据数组。

第三个参数是一个闭包，我们可以在这个闭包中定义如何发送邮件，在哪里发送邮件，以及发件人、收件人、抄送、密送、主题及任何其他的元数据。需要确保在闭包中使用想要访问的变量，并且要注意这个闭包会传递一个名为 `$m` 的参数，这是一个消息对象。

可以查看一些旧的文档来了解 classic 邮件语法。

基本“mailable”邮件

5.3 Laravel 5.3 版本引入了一种新的邮件语法，叫作 mailable 邮件语法。它和 classic 邮件语法的工作原理相同，不同的是，classic 语法在一个闭包中定义邮件信息，而 mailable 语法可以创建一个特定的 PHP 类用来表示每个邮件。

通过使用 `make:mail` Artisan 命令来实现 mailable 语法的代码如下。

```
php artisan make:mail Assignment
```

示例 15-2 对上述 PHP 类进行了详细描述。

示例15-2 一个自动生成的使用mailable语法的PHP类

```
<?php

namespace App\Mail;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class Assignment extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * 创建一个新的消息实例
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * 构建消息
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('view.name');
    }
}
```

这个类看起来可能很熟悉——它和 `Job` 几乎一模一样。这个类甚至为邮件排队引入了 `Queueable trait`¹ 和 `SerializesModels trait`。因此，任何传递给构造函数的优秀的模型都将被正确序列化。

那么这个类是如何工作的呢？基于 `mailable` 的 `build()` 方法是用来定义视图、主题的，同时可以调整除发送内容外的其他想要调整的内容。构造函数可以传递所有数据以及 `mailable` 类的所有可用于模板的公共属性。

示例 15-3 显示了我们如何为实例更新自动生成的 `mailable` 语法。

¹ `trait` 是为 PHP 的单继承语言提供了一种代码复用机制。——译者注

示例15-3 mailable语法

```
<?php

namespace App\Mail;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class Assignment extends Mailable
{
    use Queueable, SerializesModels;

    public $trainer;
    public $trainee;

    public function __construct($trainer, $trainee)
    {
        $this->trainer = $trainer;
        $this->trainee = $trainee;
    }

    public function build()
    {
        return $this->subject('New assignment from ' . $this->trainer->name)
            ->view('emails.assignment');
    }
}
```

示例 15-4 显示了如何发送 mailable 语法。

示例15-4 发送mailable语法的方法

```
// 简单发送
Mail::to($user)->send(new Assignment($trainer, $trainee));

// 含有抄送、密送等内容的发送
Mail::to($user1)
    ->cc($user2)
    ->bcc($user3)
    ->send(new Assignment($trainer, $trainee));

// 含有收集的发送
Mail::to('me@app.com')
    ->bcc(User::all())
```

```
->send(new Assignment($trainer, $trainee))
```

邮件模板

邮件模板与其他模板一样。它们可以扩展其他模板，使用分段、解析变量、包含条件或循环指令，并且可以实现普通 Blade 视图中能够实现的任何功能。

示例 15-5 是为示例 15-3 补充的一个可行的 `emails.assignments` 模板。

示例15-5 任务邮件示例

```
<!-- resources/views/emails/assignment.blade.php -->
<p>Hey {{ $trainee->name }}!</p>

<p>You have received a new training assignment from <b>{{ $trainer->name }}</b>.
Check out your <a href="{{ route('training-dashboard') }}">training
dashboard</a> now!</p>
```

在示例 15-3 中，`$trainer` 和 `$trainee` 都是 `mailable` 的公共属性，可用于模板。

如果想要明确定义传递到模板的变量，可以将 `with()` 方法链接到 `build()` 调用中，如示例 15-6 所示。

示例15-6 自定义模板变量

```
{
    return $this->subject('You have a new assignment!')
        ->view('emails.assignment')
        ->with(['assignment' => $this->event->name]);
}
```



HTML 与纯文本电子邮件

到目前为止，我们已在 `build()` 调用堆栈中使用了 `view()` 方法，这是基于我们引用的模板会回传 HTML 的设想。如果想要传送一个纯文本版本，可以用 `text()` 方法来定义纯文本视图。

```
public function build()
{
    return $this->view('emails.reminder')
        ->text('emails.reminder_plain');
}
```

build() 中可用的方法

以下是一些可以在 `mailable` 的 `build()` 方法中自定义信息的方法。

```
from($address, $name = null)
```

设置“from”的名字和地址，表示作者。

```
subject($subject)
```

设置电子邮件主题。

```
attach($pathToFile, array $options = [])
```

附加文件，符合 MIME 类型的 mime 和显示名称的 as 才是有效的选项。

```
attachData($data, $name, array $options = [])
```

从原始字符串中附加文件，和 attach() 的选择相同。

```
priority($priority)
```

设置电子邮件的优先级，1 是最高级，5 是最低级。

如果要对底层的 Swift 信息进行手动修改，需要使用 withSwiftMessage() 方法，如示例 15-7 所示。

示例15-7 修改底层SwiftMessage对象

```
public function build()
{
    return $this->subject('Howdy!')
        ->withSwiftMessage(function ($swift) {
            $swift->setReplyTo('noreply@email.com');
        })
        ->view('emails.howdy');
}
```

附件和内联图片

示例 15-8 显示了向电子邮件中添加附加邮件或原始数据的两种方法。

示例15-8 向mailables中附加文件或数据

```
// 使用本地文件名添加文件
public function build()
{
    return $this->subject('Your whitepaper download')
        ->attach(storage_path('pdfs/whitepaper.pdf'), [
            'mime' => 'application/pdf', // 可选
        ])
}
```



```

        'as' => 'whitepaper-barasa.pdf' // 可选
    ])
    ->view('emails.whitepaper');
}

// 传递元数据来添加文件
public function build()
{
    return $this->subject('Your whitepaper download')
        ->attachData(
            file_get_contents(storage_path('pdfs/whitepaper.pdf')),
            'whitepaper-barasa.pdf',
            [
                'mime' => 'application/pdf' // 可选
            ]
        )
        ->view('emails.whitepaper');
}

```

示例 15-9 显示了如何在邮件中直接嵌入图像。

示例15-9 内联图像

```

<!-- emails/image.blade.php --!>
Here is an image:

<img src=>{{ $message->embed(storage_path('embed.jpg')) }}">

Or, the same image embedding the data:

<img src=>{{ $message->embedData(
    file_get_contents(storage_path('embed.jpg')), 'embed.jpg'
) }}">

```

队列

发送电子邮件是一项耗时的任务，可能会导致应用程序运行减慢，因此，通常将发送电子邮件的任务转移到后台队列中。事实上，Laravel 有一套内置的工具可以使邮件排队更加容易，不必为每封邮件编写队列任务。



配置队列

此处提到的一切都以队列正确配置为前提。通过第 16 章可以了解到更多关于队列如何工作以及如何让队列在应用程序中运行的内容。

queue()

让邮件对象排队而不是立即发送，只要简单地将要发送的对象传递到 `Mail::queue()` 中即可，而不是 `Mail::send()` 中。

```
Mail::queue(new Assignment($trainer, $trainee));
```

later()

`Mail::later()` 和 `Mail::queue()` 的工作原理相同，但是它允许添加一个延时——既可以在几分钟内添加，也可以通过传递一个 `DateTime`¹ 或 `Carbon`² 示例来设置指定的时间——电子邮件会在这个时间从队列中拉取并发送。

```
$when = Carbon::now()->addMinutes(30);  
Mail::later($when, new Assignment($trainer, $trainee));
```

指定队列或连接

对于 `queue()` 和 `later()` 来说，如果想要指定将邮件添加到哪一个队列或队列连接中，就需要对即将传递的对象使用 `onConnection()` 和 `onQueue()` 方法。

```
$message = (new Assignment($trainer, $trainee))  
    ->onConnection('sqs')  
    ->onQueue('emails');  
  
Mail::to($user)->queue($message);
```

本地开发

在实际生产环境中发送邮件没有任何问题。但是，如何对它们进行全测试呢？有三个主要工具可供参考：Laravel 的日志驱动程序、名为 Mailtrap 的软件服务 (SaaS) 应用程序以及“universal to”配置选项。

日志驱动程序

Laravel 提供了一个日志驱动程序，记录了我们试图发送到本地 `laravel.log` 文件（默认情况下在 `storage/logs` 中）的每一封电子邮件。

如果想使用这个日志，需要编辑 `.env` 并且设置 `MAIL_DRIVER` 来进行登录。现在打开或拖放 `storage/logs/laravel.log`，从应用程序中发送一封电子邮件，可以看到如下所示内容。

¹ 日期时间。——译者注

² 一组被称为应用程序编程接口 (API) 的工具。——译者注

```
Message-ID: <04ee2e97289c68f0c9191f4b04fc0de1@localhost>
Date: Tue, 17 May 2016 02:52:46 +0000
Subject: Welcome to our app!
From: Matt Stauffer <matt@mattstauffer.co>
To: freja@jensen.no
MIME-Version: 1.0
Content-Type: text/html; charset=utf-8
Content-Transfer-Encoding: quoted-printable
```

Welcome to our app!

Mailtrap.io

Mailtrap (<https://mailtrap.io>) 是一个在开发环境中捕捉和检查电子邮件的服务。通过 SMTP 将邮件发送到 Mailtrap 服务器，而不是直接发送给预期的收件人，无论这些电子邮件的地址是否在收件人区域，Mailtrap 都会将它们全部捕捉到并提供一个基于网络的电子邮件客户端对其进行检查。

设置 Mailtrap 需要先注册一个免费的 Mailtrap 账户，并且访问演示的基本信息板。从 SMTP 列中复制用户名和密码。

编辑应用程序的 `.env` 文件并且将 `mail` 部分设置为以下值。

```
MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=your_username_from_mailtrap_here
MAIL_PASSWORD=your_password_from_mailtrap_here
MAIL_ENCRYPTION=null
```

现在，由应用程序发送的所有电子邮件都会在 Mailtrap 收件箱中显示出来。

Universal to

如果想要在首选客户端中检查电子邮件，可以使用“`universal to`”配置设置来覆盖每条信息中的 `to` 字段。要进行此设置，需要在 `config/mail.php` 文件中添加一个“`to`”关键字，代码如下。

```
'to' => [
    'address' => 'matt@mattstauffer.co',
    'name' => 'Matt Testing My Application'
],
```

需要注意的是，要实际设置一个像 Mailgun 或 Sendmail 一样的真正的电子邮件驱动程序，之后才能使用。

通知

网络应用程序发送的大多数邮件具有以下目的：确实通知用户一个特定的动作已经发生或需要发生。由于用户的沟通倾向越来越多元化，因此我们通过 Slack、SMS 和其他途径收集到了越来越多的不同的包来进行通信。

Laravel 5.3 版本中引入了一个新概念：通知。就好像邮件一样，通知是一个 PHP 类，代表我们想要发送给用户的单一通信内容。现在，来想象一下，假设我们正在通知体能训练应用软件的用户，告诉他们软件中有一个新的锻炼项目。

每个类都代表了给使用一个或多个通知频道的用户发送通知时需要的所有消息。单个通知具备发送电子邮件、通过 Nexmo 发送 SMS、发送 WebSockets ping、向数据库添加记录、向 Slack 频道发送消息等诸多功能。

创建通知的命令如下。

```
php artisan make:notification WorkoutAvailable
```

示例 15-10 显示了这个命令的返回情况。

示例15-10 一个自动生成的通知类

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;

class WorkoutAvailable extends Notification
{
    use Queueable;

    /**
     * 创建一个新的通知实例
     *
     * @return void
     */
    public function __construct()
    {
        //
    }
}
```

```

/**
 * 获取通知的交付渠道
 *
 * @param mixed $notifiable
 * @return array
 */
public function via($notifiable)
{
    return ['mail'];
}

/**
 * 获取通知的邮件表示
 *
 * @param mixed $notifiable
 * @return \Illuminate\Notifications\Messages\MailMessage
 */
public function toMail($notifiable)
{
    return (new MailMessage)
        ->line('The introduction to the notification.')
        ->action('Notification Action', 'https://laravel.com')
        ->line('Thank you for using our application!');
}

/**
 * 获取通知的数组表示形式
 *
 * @param mixed $notifiable
 * @return array
 */
public function toArray($notifiable)
{
    return [
        //
    ];
}
}

```

我们可以从以上代码中学到一些知识。首先，要把相关数据传递到构造函数。其次，对于给定的用户，可以通过 `via()` 方法来定义应该使用哪一个通知频道（`$notifiable` 代表我们想要通知的系统中的实体，在大多数应用中，它表示一个用户，但并不总是）。最后，每个通知频道都有各自的方法，允许我们专门定义如何在该频道中发送通知。



什么时候 \$notifiable 不表示一个用户

虽然最常见的通知目标是用户，但我们仍有可能通知其他实体。因为应用程序中有多个用户类型——因此，我们还可能想要通知“培训者”和“学员”，甚至可能想要通知某个“群组”，某个“公司”，或者某个“服务器”。

若要实现上述目标，需要修改类的 `WorkoutAvailable` 示例。请参考示例 15-11。

示例15-11 `WorkoutAvailable`通知类

```
...
class WorkoutAvailable extends Notification
{
    use Queueable;

    public $workout;

    public function __construct($workout)
    {
        $this->workout = $workout;
    }

    public function via($notifiable)
    {
        // 这个方法目前在用户端不存在，我们正在努力实现这个功能
        return $notifiable->preferredNotificationChannels();
    }

    public function toMail($notifiable)
    {
        return (new MailMessage)
            ->line('You have a new workout available!')
            ->action('Check it out now', route('workout', [$this->workout]))
            ->line('Thank you for training with us!');
    }

    public function toArray($notifiable)
    {
        return [];
    }
}
```

为通知对象定义 `via()` 方法

如示例 15-11 所示，我们要决定对于不同的通知以及通知对象，到底应该采用哪个通知频道。

可以把所有的通知都当作邮件来发送，也可以把它们当作 SMS 来发送（见示例 15-12）。

示例 15-12 最简单可行的via()方法

```
public function via($notifiable)
{
    return 'nexmo';
}
```

也可以让每一个用户选择一种偏好方法，并将其保存在 User 内部。

示例 15-13 每个用户自定义via()方法

```
public function via($notifiable)
{
    return $notifiable->preferred_notification_channel;
}
```

或者，参照示例 15-11，为每一个通知对象创建一个允许复杂通知逻辑的方法。例如，可以在工作时间内利用某些频道通知用户，而在非工作时间采用别的频道。需要注意的是，via() 是一个 PHP 类，因此我们可以通过它实现任何复杂逻辑。

发送通知

有两种发送通知的方式：使用 Notification facade，或在一个 Eloquent 类（类似于 User 类）中添加 Notifiable 特性。

使用 Notifiable trait 发送通知

任何导入 Laravel\Notifications\Notifiable trait（默认情况下是 App\User 类）的模型都具有一个可传递通知的 notify() 方法，如示例 15-14 所示。

示例15-14 使用Notifiable trait发送通知

```
use App\Notifications\WorkoutAvailable;
...
$user->notify(new WorkoutAvailable($workout));
```

使用 Notification facade 发送通知

Notification facade 发送通知是两种方法中较为“笨拙”的一种，因为我们要同时传递通知对象和通知，好处在于可以同时传递多个通知对象，如示例 15-15 所示。

示例15-15 使用Notification facade发送通知

```
use App\Notifications\WorkoutAvailable;
```

```
...
Notification::send($users, new WorkoutAvailable($workout));
```

排队通知

大多数通知驱动程序需要发送 HTTP 请求从而发送通知，这有可能会降低用户体验，因此我们将通知进行排队，从而提升用户体验。所有通知都会默认被导入 `Queueable` trait，因此我们需要做的仅仅是在通知中添加一个 `implements ShouldQueue`，Laravel 便会立即把该通知移到队列中去。

和其他所有的排队功能一样，需要保证队列都正确地设置配置，队列工作者也在运行中。

如果想要延迟通知的传送，可以运行 `delay()` 方法。

```
$delayUntil = Carbon::now()->addMinutes(15);

$user->notify((new WorkoutAvailable($workout))->delay($delayUntil));
```

开箱即用的通知类型

开箱即用，即 Laravel 为电子邮件、数据库、广播、Nexmo SMS 和 Slack 提供的通知驱动软件。下面我们简单介绍一下开箱即用，建议大家参考官方文档（<https://laravel.com/docs/notifications>），以便全面深入理解。

创建通知驱动软件非常容易，可以在 Laravel Notification Channels 网站（<http://laravel-notification-channels.com/>）找到详细方法。

电子邮件通知

来看一下示例 15-11 中的电子邮件是如何创建的。

```
public function toMail($notifiable)
{
    return (new MailMessage)
        ->line('You have a new workout available!')
        ->action('Check it out now', route('workout', [$this->workout]))
        ->line('Thank you for training with us!');
}
```

结果如图 15-1 所示。电子邮件通知系统将应用名放置于电子邮件的头部，也可以在 `config/app.php` 的 `name` 关键字中自定义应用程序的名称。

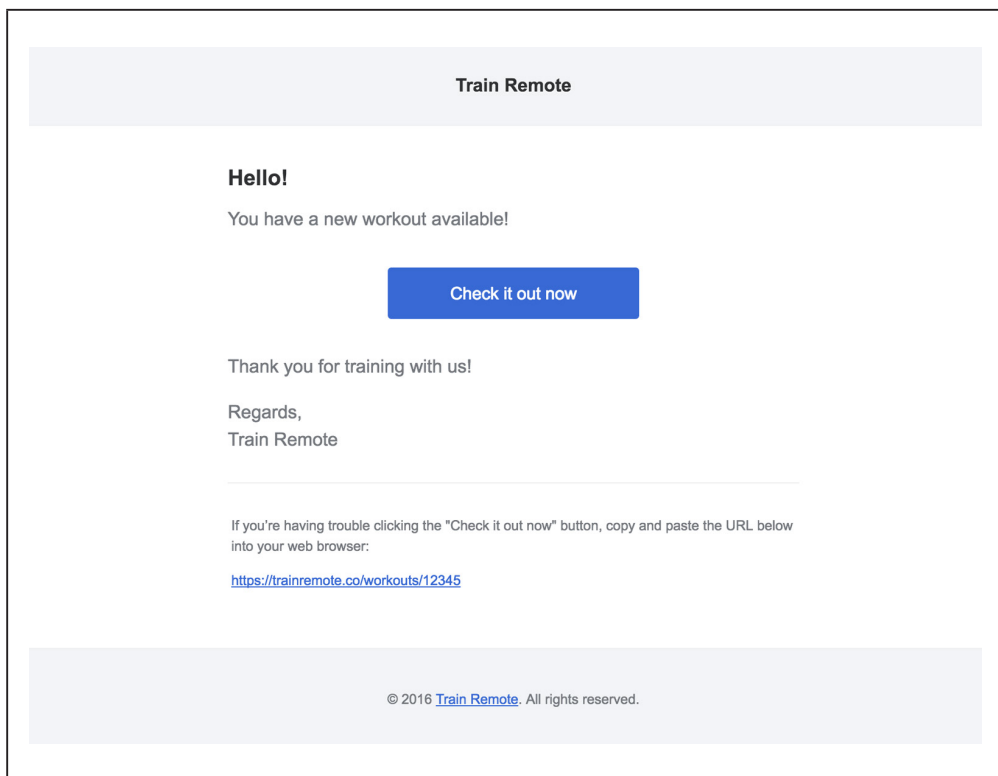


图 15-1 使用默认通知模板发送电子邮件

这封电子邮件会自动发送给通知对象的 `email` 属性，我们也可以通过向一个名为 `routeNotificationForMail()` 的类添加方法来自定义上述操作，这个类会返回我们想要发送电子邮件通知的目标地址。

电子邮件的主题是通过解析通知类名并转换成文字来设定的。因此，`WorkoutAvailable` 通知将会具有 `Workout Available` 的默认主题。也可以通过将 `subject()` 方法链接到 `toMail()` 方法中的 `MailMessage` 上来自定义主题。

如果想要修改模板，需要发布并编辑核心内容，方法如下。

```
php artisan vendor:publish --tag=laravel-notifications
```

也可以将默认模板的样式更改为“错误”消息，该消息使用不同的编程语言，并且将主按钮的颜色变为红色。只需在 `toMail()` 方法中的 `MailMessage` 调用链中添加一个 `error()` 方法调用即可实现。

数据库通知

可以通过使用数据库通知频道向数据库表发送通知。首先,使用 `php artisan notifications:table` 语句创建一个新表。然后,在通知中创建一个 `toDatabase()` 方法,该方法会针对通知返回一个数据数组,该数据数组以 JSON 格式编码存储到数据库表的数据列中。

`Notifiable` trait 会向导入的所有模型中都添加一个通知关系,这样就可以轻松地访问通知表中的记录了。如果使用数据库通知,则可以像下面这样进行操作。

```
User::first()->notifications->each(function ($notification) {  
    // 完成某些功能  
});
```

数据库通知频道也有一个表示通知是否“已读”的标识。可以试着只标记“未读”通知,代码如下。

```
User::first()->unreadNotifications->each(function ($notification) {  
    // 完成某些功能  
});
```

也可以将某个或所有的通知都标记为已读。

```
// 某个  
User::first()->notifications->each(function ($notification) {  
    if ($condition) {  
        $notification->markAsRead();  
    }  
});  
  
// 所有  
User::first()->unreadNotifications->markAsRead();
```

广播通知

广播频道使用 Laravel 的事件广播功能 (Echo) 来发送通知。

在通知上创建一个 `toBroadcast()` 方法并返回数据数组,如果应用程序正确配置为事件广播,则数据将在名为 `{notifiable}.{id}` 的私有频道上广播。`{id}` 就是通知对象的 ID,`{notifiable}` 是通知对象的完全限定类名称,其中“斜杠”可以用“句点”来代替——例如, ID 为 1 的 `App\User` 的私有频道可以表示为 `App.User.1`。

SMS 通知

SMS 通知是通过 Nexmo 发送的，因此，如果想要发送 SMS 通知，则需要注册一个 Nexmo 账户，然后遵循文件 (<https://laravel.com/docs/notifications>) 中的说明进行操作。与其他频道一样，我们需要设置一个 `toNexmo()` 方法然后自定义消息内容。

Slack 通知

slack 通知频道支持自定义通知形式，甚至可以向通知中添加附件。和其他频道一样，我们需要设置一个 `toSlack()` 方法然后自定义消息内容。

测试

下面介绍如何测试邮件和通知。

邮件

Laravel 中有两种测试邮件的选项。如果使用传统邮件语法，建议使用由 Adam Wathan 为 Tighten 编写的 MailThief (<https://github.com/tightenco/mailthief>) 工具。一旦使用 Composer 将 MailThief 引入应用程序中，就可以在测试中使用 `MailThief::hijack()`，使 MailThief 捕捉到所有对 Mail facade 或邮件发件人类的调用。

然后，MailThief 就能对发件人、收件人、抄送、密件抄送，甚至邮件的内容及附件做出断言。想要了解更多内容请参考 GitHub，或在应用程序中进行验证。

```
composer require tightenco/mailthief --dev
```

如果使用 mailable，则有一个简单的语法可用于为发送的邮件编写断言（见示例 15-16）。

示例15-16 针对mailable编写断言

```
public function test_signup_triggers_welcome_email()
{
    ...

    Mail::assertSent(WelcomeEmail::class, function ($e) {
        return $e->subject == 'Welcome!';
    });

    // 也可以使用 assertSentTo() 方法直接测试收件人
}
```

通知

Laravel 提供了一组内置的断言来测试通知，见示例 15-17。

示例15-17 发送断言通知

```
public function test_new_signups_triggers_admin_notification()
{
    ...

    Notification::assertSentTo($user, NewUsersSignedup::class,
        function ($n, $channels) {
            return $n->user->email == 'user-who-signed-up@gmail.com'
                && $channels == ['mail'];
        });

    // 也可以使用 assertNotSentTo() 方法
}
```

本章小结

Laravel 的邮件和通知功能为多种不同的通知系统提供了简单、一致的界面。Laravel 的邮件系统使用“mailable”表示电子邮件的 PHP 类，为不同的邮件驱动程序提供了一致的语法。通知系统可以轻易实现构建单个通知的功能，该通知可以在多种不同媒体中传送——无论通过电子邮件、SMS，还是实际生活中的邮寄。

队列，任务，事件，广播及 调度程序

到目前为止，我们已经介绍了驱动网络应用程序中最常见的一些结构：数据库、邮件、文件系统等。这些在大多数应用程序和框架中都很常见。

Laravel 还为一些不太常见的架构模式和应用程序结构提供了可用工具。在这一章中，我们将介绍 Laravel 中用于实现队列、队列任务、事件和 WebSocket¹ 事件发布的工具。我们还将讨论 Laravel 的调度程序，调度程序的出现使 cron² 逐渐淡出人们的视线。

队列

要理解队列是什么，可以参考在银行中排队的场景。即使有很多行（队列），但每一个队列中只有一个人在享受服务，每个人最终都将到达前端并享受服务。一些银行有严格的“先入先出”的原则，但在其他的一些银行中，并没有严格保障，保证不会有人在你前方某处插队。大体上来说，有的人可以插入队列并提前离开，或者成功地办理业务，然后离开。有的人甚至可能已经到达队列的前端但没有得到想要的服务，然后回到队列中再次等待办理。

编程中的队列与上述场景是十分相似的。应用程序执行特定行为的大段代码作为“任务”被添加到队列中，然后通常作为“队列工作者”的其他独立的应用程序结构负责从队列中拉取一个任务并执行适当的行为。队列工作者可以删除任务，将它们延迟返回队列，或将其标记为处理成功的任务。

1 WebSocket 是一种在单个 TCP 连接上进行全双工通信的协议。——译者注

2 cron 是一种常见于 UNIX 和类 UNIX 操作系统之中的命令，用于设置周期性执行的指令。——译者注

Laravel 可以通过使用 Redis、beanstalkd、Amazon 的 SQS(简单队列服务) 或数据库表单使得为队列提供服务变得容易。还可以选择 sync 驱动使任务在应用程序中正确运行但无须排队, 否则任务的 null 驱动程序将被丢弃。以上两种情况常用于本地开发或测试环境中。

为什么使用队列

使用队列可以很轻松地从中删除昂贵或缓慢的进程。最常见的例子是发送邮件——这样做可能比较慢, 而且我们不希望用户等待邮件发送来响应他们的操作。相反, 触发“发送邮件”排队任务, 可以让使用者继续做他们自己的事情。有时我们不只是希望节省用户的时间, 也可能会有一个 cron 任务或 webhook 进程需要处理, 我们不希望它从头到尾只运行一次(可能超时), 因此可以选择对这些单个部分进行排队, 并让队列工作者逐个对其进行处理。

此外, 如果有一些超出服务器处理范围的重大进程, 则可以让多个队列工作者加快转发速度, 比正常应用程序服务器独自完成队列工作更快。

基本队列配置

像许多抽象了多个程序提供者的其他 Laravel 的功能一样, 队列有自己的专用配置文件 (`config/queue.php`), 允许设置多个驱动程序并定义默认的驱动程序。这也是存储 SQS、Redis 或 beanstalkd 身份验证信息的地方。



Laravel Forge 上的简单 beanstalkd 队列

我们尚未深入了解 Laravel Forge (<http://forge.laravel.com/>), 这是由 Laravel 的创始人 Taylor Otwell 提供的一项托管服务。我们创建的每个服务器都会自动配置 beanstalkd, 所以当访问任何站点的 Forge 控制台时, 都可以直接进入“Queue Workers”标签, 点击 Start Worker, 使用 beanstalkd 作为队列驱动程序。通过这种方式, 我们可以保留所有默认设置, 并且不需要进行其他操作。

队列任务

还记得前面提到的银行排队的类比吗? 在编程术语中, 银行队列(行)中的每个人都是一项待办任务。这项任务可以任意改变, 它可以只是一个字符串, 一个数组, 或者一个对象。在 Laravel 中, 它是包含任务名称、数据有效载荷、到目前为止所操作的次数, 以及其他一些简单元数据的系列信息。

我们无须担心与 Laravel 的互动。Laravel 提供了一个名为 `Job` 的结构，它的目的是封装单个任务——可以命令应用程序执行的行为——并且允许将其添加到队列中或从队列中提取。还有一些简单的帮助程序，可以方便地对 Artisan 命令和邮件进行排队。

下面来看一个具体示例：用户使用我们的 SaaS 应用程序改变其计划，我们想要返回关于整体利润的计算。

创建任务

和之前一样，Artisan 命令如下。

```
php artisan make:job CrunchReports
```

参照示例 16-1，查看运行结果。

示例16-1 Laravel中任务的默认模板

```
<?php

use Illuminate\Bus\Queueable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class CrunchReports implements ShouldQueue
{
    use InteractsWithQueue, Queueable, SerializesModels;

    /**
     * 创建一个新的任务实例
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * 执行任务
     *
     * @return void
     */
    public function handle()
    {
        //
    }
}
```

```

    }
}

```

可以看到，这个模板引入了 `Queueable`、`InteractsWithQueue` 和 `SerializesModels` 的特性，并实现了 `ShouldQueue` 接口。在 Laravel 5.3 版本之前，这些功能是通过父类 `App\Jobs` 来实现的。

我们还从这个模板中获得了两种方法：构造函数法和 `handle()` 方法。使用构造函数法可以将数据附加到任务中，`handle()` 方法则是任务逻辑归属的地方（也是用来引入依赖项的方法签名）。

特性和接口为类提供了加入队列以及与其进行互动的功能。`Queueable` 允许我们指定 Laravel 如何将任务推送到队列中；`InteractsWithQueue` 允许每个任务在处理过程中控制其与队列的关系，包括删除队列或重新排队；`SerializesModels` 为任务提供序列化和反序列化 Eloquent 模型的能力。



序列化模型

`SerializesModels` 特性使任务能够对引入的模型进行序列化，以便 `handle()` 方法可以访问它们。然而，因为可靠地序列化整个 Eloquent 对象很难，所以该特性可以确保在将任务推送到队列时，任何附加的 Eloquent 对象的主键都可以被序列化。当任务被反序列化处理时，该特性通过其主键从数据库中提取新的 Eloquent 模型。这意味着，当任务运行时，会显示这个模型的新实例，而不是在其队列中排队时的状态。

下面来编写样本类的方法，如示例 16-2 所示。

示例16-2 任务实例

```

...
use App\ReportGenerator;
use Illuminate\Log\Writer as Logger;

class CrunchReports implements ShouldQueue
{
    use InteractsWithQueue, SerializesModels;

    protected $user;

    public function __construct($user)
    {
        $this->user = $user;
    }
}

```



```

    }

    public function handle(ReportGenerator $generator, Logger $logger)
    {
        $generator->generateReportsForUser($this->user);

        $logger->info('Generated reports.');
```

我们期待在创建该任务时引入 `User` 实例，然后在处理该实例时输出 `ReportGenerator` 类和一个 `Logger`（Laravel 提供的）。Laravel 会读取这两种类型提示，并自动引入这些依赖项。

将任务推送至队列

有两种主要方法可用于将任务推送至队列：全局 `dispatch()` 助手和 `DispatchesJobs` trait 提供的方法，它在每个控制器中是默认导入的。

将每一个被创建的任务实例传递给构造函数，可以附加上其他必要的的数据，并将其传递给 `dispatch()` 方法（见示例 16-3）。

示例16-3 Dispatching jobs

```
// 在控制器中
public function index()
{
    $user = auth()->user();
    $this->dispatch(new \App\Jobs\CrunchReports($user));
}

// 在其他地方
dispatch(new \App\Jobs\CrunchReports($user));
```

我们可以控制三种设置，以便精确地自定义如何分派任务：连接、队列和延迟。

自定义连接。如果同时进行多个队列连接，则可以通过在实例化的任务上运行 `onConnection()` 来自定义连接。

```
dispatch((new DoThingJob)->onConnection('redis'));
```

自定义队列。在队列服务器中，可以指定将某个任务推送到特定的命名队列中。例如，可以根据它们的重要性对队列进行区分，并将其命名为 `low` 或 `high`。

可以使用 `onQueue()` 方法自定义正在推送任务的队列。

```
dispatch((new DoThingJob)->onQueue('high'));
```

自定义延迟。可以使用 `delay()` 方法自定义队列工作人员在处理任务之前等待的时间，该方法接收一个表示任务延迟秒数的整数为参数。

```
// 向队列工作者释放工作之前，延迟 1 分钟
dispatch((new DoThingJob)->delay(60));
```

注意，Amazon SQS 不允许延迟超过 15 分钟。

运行队列工作者

那么什么是队列工作者，它又是如何工作的呢？在 Laravel 中，这是一个能够永久运行（直到手动停止）的 Artisan 命令，负责从队列中提取任务并运行它们。

```
php artisan queue:work
```

该命令启动一个守护进程来“监听”队列，每次队列中有任务时，它都会拉取第一个任务，处理它，删除它，然后转到下一个任务。如果任何时候都没有任务，那么在重新检查是否有更多的任务之前，它会“休眠”一段可配置的时间。

我们可以定义一个任务在队列监听器停止之前能够运行多少秒 (`--timeout`)，没有任务的时候监听器应该“休眠”多少秒 (`--sleep`)，在删除之前应该允许每个任务有多少次尝试 (`--tries`)，工作者应该监听哪个连接 (`queue:work` 后面的第一个参数)，以及应该监听哪个队列 (`--queue=`)。

```
php artisan queue:work redis --timeout=60 --sleep=15 --tries=3
--queue=high,medium
```

还可以使用 `php artisan queue:work` 处理单个任务。

错误处理

那么处理任务时会出现什么问题呢？

处理异常

如果抛出异常，队列监听器会将该任务释放到队列中。该任务将被重新执行，直到它能被成功地完成或尝试了队列监听器允许的最多次数为止。

限制尝试次数

尝试的最多次数是由 `--tries` 开关传递到 `queue:listen` 或 `queue:work` 的 Artisan 命令定义的。



无限重试的危险

如果没有设置 `--tries`，或者将其设置为 `0`，则队列监听器将允许无限重试。这意味着，如果存在任务永远无法完成的情况——例如，如果它依赖于已被删除的推文——那么应用程序便会慢慢地停止工作，因为它要不断地重新尝试完成不完整的任务。

官方文档和 Laravel Forge 都显示，`3` 为最大重试次数的默认值。所以，一旦发生混乱，便可以以此为参考进行调整。

```
php artisan queue:listen --tries=3
```

如果想要在某个时刻检查一下尝试运行的次数，可以对任务本身使用 `attempts()` 方法，如示例 16-4 所示。

示例16-4 检查一项任务已经尝试运行了多少次

```
public function handle()
{
    ...
    if ($this->attempts() > 3) {
        //
    }
}
```

处理失败的任务

一旦尝试次数超过允许重试次数的上限，就被认为是“失败”的任务。在进行其他任务之前——即使只是想限制任务可以尝试的次数——需要创建“失败任务”数据库表。

使用 Artisan 命令可以创建迁移。

```
php artisan queue:failed-table
php artisan migrate
```

任何超过允许尝试次数上限的任务都将被转存到那里。对于失败的任务，可以进行以下处理。

首先，可以为任务本身定义一个 `fail()` 方法，该方法在任务失败时运行（参见示例 16-5）。

示例16-5 任务失败时运行的方法

```
...
class CrunchReports implements ShouldQueue
{

```

```

...

public function failed()
{
    // 执行任何操作
}
}

```

接下来，可以为失败的任务注册一个全局处理程序。在应用程序引导中的某处——如果不知道在哪里，只需将其放在 `AppServiceProvider` 的 `boot()` 方法中即可——将代码放在示例 16-6 的代码段中，定义监听器。

示例16-6 注册一个全局处理程序来处理失败的任务

```

// 一些服务提供商
use Illuminate\Support\Facades\Queue; .

..
public function boot()
{
    Queue::failing(function ($connection, $job, $data) {
        // 实现任何想要的功能
    });
}

```

还有一套用于与失败的任务表进行交互的 Artisan 工具。

`queue:failed` 显示失败任务的列表。

```
php artisan queue:failed
```

列表形式如下。

```

+-----+-----+-----+-----+-----+
| ID | Connection | Queue  | Class                               | Failed At           |
+-----+-----+-----+-----+-----+
| 9  | database   | default | App/Jobs/AlwaysFails               | 2016-01-26 03:42:55 |
+-----+-----+-----+-----+-----+

```

在列表中，我们可以捕获所有失败任务的 ID，并用 `queue:retry` 进行重试。

```
php artisan queue:retry 9
```

如果不想重试所有的任务，请传递 `all` 而非单个 ID。

```
php artisan queue:retry all
```

可以使用 `queue:forget` 删除单个失败任务。

```
php artisan queue:forget 5
```

可以使用 `queue:flush` 删除所有失败任务。

```
php artisan queue:flush
```

控制队列

有时候，在处理任务的过程中需要添加一些条件，这些条件可能会使任务重新启动，或者将任务永远删除。

要将任务返回到队列中，可以使用 `release()` 命令，见示例 16-7。

示例16-7 将任务返回队列中

```
public function handle()
{
    ...
    if (condition) {
        $this->release($numberOfSecondsToDelayBeforeRetrying);
    }
}
```

如果想在处理过程中删除某项任务，可以随时返回，见示例 16-8，这是向队列发出的信号，任务应被适当地处理，不应该返回队列。

示例16-8 删除任务

```
public function handle()
{
    ...
    if ($jobShouldBeDeleted) {
        return;
    }
}
```

支持其他功能的队列

队列的主要用途是推送任务，同时也可以使用 `Mail::queue` 功能对邮件进行列队。可以在第 15 章关于 `queue()` 的内容中了解更多相关信息，也可以对 Artisan 命令进行排列，这些我们在第 7 章中介绍过。

事件

对于任务，调用代码通知应用程序执行 `CrunchReports` 或 `NotifyAdminOfNewSignup` 操作。

对于事件，调用代码通知应用程序发生了以下事情：`usersubscribed`、`usersignedup` 或 `contactwasadded`。事件是某事发生时的通知。

其中一些事件可能会被框架本身“解绑”。例如，Eloquent 模型在保存、创建或删除时触发事件。也有些事件是由应用程序的代码手动触发的。

被解绑的事件不会再执行任何操作。但是，我们可以绑定事件监听器，其唯一目的是监听特定事件的广播并做出响应。任何事件都可以有零到多个事件监听器。

Laravel 事件的结构类似于观察者或“pub / sub”模式。许多事件被触发到应用程序中，有些可能永远不会被监听，而另一些可能有十几个监听器。事件本身不知道这些情况，也不关心这些。

触发事件

触发事件的方法有三种。可以使用 Event facade，引入 Dispatcher，或使用 `event()` 全球助手。

```
Event::fire(new UserSubscribed($user, $plan));  
// 或者  
$dispatcher = app(Illuminate\Contracts\Events\Dispatcher);  
$dispatcher->fire(new UserSubscribed($user, $plan));  
// 或者  
event(new UserSubscribed($user, $plan));
```

如果有疑问，建议使用全局助手函数。

创建一个要触发的事件，可以使用 `make:event` Artisan 命令。

```
php artisan make:event UserSubscribed
```

执行上述命令将会创建一个如示例 16-9 所示的文件。

示例16-9 Laravel事件的默认模板

```
<?php
```

```
namespace App\Events;
```

```

use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

```

```

class UserSubscribed
{
    use InteractsWithSockets, SerializesModels;

    /**
     * 创建一个新的事件实例
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * 获取事件频道
     *
     * @return Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}

```

通过以上代码可知，`SerializesModels`的工作方式和任务一样，允许接收 Eloquent 模型作为参数。`InteractsWithSockets`、`Should Broadcast` 和 `broadcastOn()` 方法提供了使用 WebSockets 广播事件的后台功能，稍后将详细介绍。

这里没有 `handle()` 或 `fire()` 方法似乎很奇怪。但是请记住，此对象不是为了确定一个特定的动作，而是为了封装一些有用的数据。第一个数据是它的名称，`UserSubscribed` 告诉我们发生了特定的事件（用户订阅）。其余的数据是我们传递给构造函数并与此实体关联的数据。

示例 16-10 显示了我们可能会对 `UserSubscribed` 事件执行的操作。

示例16-10 将数据引入事件

```
...
class UserSubscribed
{
    use InteractsWithSockets, SerializesModels;

    public $user;
    public $plan;
    public function __construct($user, $plan)
    {
        $this->user = $user;
        $this->plan = $plan;
    }
}
```

现在便可以使用一个对象恰当地表示发生的事件了——`$event->user` 订阅了 `$event->plan` 计划。

监听事件

在有能力触发事件后，我们来看看如何监听事件。

首先，创建一个事件监听器。假设我们想要在每次有新用户订阅事件时通过电子邮件发送应用的所有者，代码如下。

```
php artisan make:listener EmailOwnerAboutSubscription --event=UserSubscribed
```

执行代码，将得到示例 16-11 中的文件。

示例16-11 用于laravel事件监听的默认模板

```
<?php

namespace App\Listeners;

use App\Events\UserSubscribed;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EmailOwnerAboutSubscription
{
    /**
     * 创建事件监听器
     *
     * @return void
```



```

    */
    public function __construct()
    {
        //
    }

    /**
     * 处理事件
     *
     * @param UserSubscribed $event
     * @return void
     */
    public function handle(UserSubscribed $event)
    {
        //
    }
}

```

这就是 `handle()` 方法所在的位置，也是操作发生的地方。该方法期望收到一个类型为 `UserSubscribed` 的事件并做出响应。

所以，我们要为上述方法发送一封电子邮件，见示例 16-12。

示例16-12 事件监听器实例

```

...
use Illuminate\Contracts\Mail\Mailer;

class EmailOwnerAboutSubscription
{
    protected $mailer;

    public function __construct(Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function handle(UserSubscribed $event)
    {
        $this->mailer->send(
            new OwnerSubscriptionEmail($event->user, $event->plan)
        );
    }
}

```

非常好！还有最后一个任务：设置监听器来监听 `UserSubscribed` 事件。我们将在 `EventServiceProvider` 类的 `$listen` 属性中设置它，见示例 16-13。

示例16-13 将监听器绑定到 `EventServiceProvider` 中的事件

```
class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        \App\Events\UserSubscribed::class => [
            \App\Listeners\EmailOwnerAboutSubscription::class,
        ],
    ];
}
```

可以看到，每个数组项的键是事件的类名，该值是一个监听器类名的数组。我们可以在 `Usersubscribed` 键下添加许多类名，它们会监听并响应每个 `Usersubscribed` 事件。

事件订阅者

还有一个结构可以用于定义事件及其监听器之间的关系。Laravel 中有一个称为事件订阅者的概念，它是类的集合，其中包含一系列方法，这些方法可以作为单独的监听器用来处理特定的事件，而且还明确了哪个方法应该处理哪个事件，见示例 16-14。

示例16-14 事件订阅者

```
<?php

namespace App\Listeners;

class UserEventSubscriber
{
    public function onUserSubscription($event)
    {
        // 处理 UserSubscribed 事件
    }

    public function onUserCancellation($event)
    {
        // 处理 UserCancelled 事件
    }

    public function subscribe($events)
    {
        $events->listen(
            \App\Events\UserSubscribed::class,
            'App\Listeners\UserEventSubscriber@onUserSubscription'
        );
    }
}
```

```

        );

        $events->listen(
            \App\Events\UserCancelled::class,
            'App\Listeners\UserEventSubscriber@onUserCancellation'
        );
    }
}

```

订阅者需要定义一个 `subscribe()` 方法，该方法传递一个事件调度器的实例，我们将使用它对事件及其监听器进行配对。但这种情况是该类特有的，不适用于所有类。大家可以将这种情况作为一个规律来记住，任何时候看到“@”这样的标识，就知道类名在@的左边，而方法名在右边。

因此，在示例 16-14 中，我们定义了订阅者的 `onUserSubscription()` 方法，该方法将监听所有的 `UserSubscribed` 事件。

最后需要在 `App\Providers\EventServiceProvider` 中将订阅者的类名添加到 `$subscribe` 属性，如示例 16-15 所示。

示例16-15 注册事件订阅者

```

...
class EventServiceProvider extends ServiceProvider
{
    ...
    protected $subscribe = [
        \App\Listeners\UserEventSubscriber::class
    ];
}

```

通过 WebSocket 广播事件及 Laravel Echo

WebSocket（通常表示为 WebSockets）是一种由 Pusher 推广的协议，它可以简化网络设备间的实时通信。WebSocket 库打开了客户端和服务端之间的直接连接，不依靠 HTTP 请求传递的信息。WebSocket 是后端工具，就像 Gmail 和 Facebook 中的聊天框一样。

WebSocket 最适合处理 pub / sub 结构中传递的小数据——就像 Laravel 的事件一样。Laravel 具有一套内置工具，可以轻松定义一个或多个应该广播到 WebSocket 服务器的事件。例如，将 `MessageWasReceived` 事件发布到某个用户或用户组的通知框中，消息会立刻在应用程序中出现。

Laravel Echo

Laravel 中还有一个更强大的工具，用于广播更复杂的事件。如果需要保存通知，或者使丰富的前端数据模型与 Laravel 的应用程序保持同步，请查看 Laravel Echo，本章末尾将详细介绍。Echo 的大部分内容都被内置到了 Laravel 核心中，第 372 页的“高级广播工具”将介绍其中的一些内容，另外涉及 JavaScript Echo 库的部分，我们将在第 376 页“Laravel Echo（JavaScript 方面）”中介绍。

配置和设置

查看 `config/broadcasting.php`，找到事件广播的配置设置。Laravel 支持三种广播驱动程序：Pusher，一种付费的 SaaS 产品；Redis，用于在本地运行 WebSocket 服务；log，用于本地开发和调试。



队列监听器

为了使事件广播迅速移动，Laravel 将推送广播到队列中。这意味着，我们需要运行队列工作者（或使用同步队列驱动程序进行本地开发）。第 358 页的“运行队列工作者”介绍了具体的运行方法。

Laravel 在队列工作者查找新的任务之前建议保持 3 秒的延迟。但是，在事件广播中，某些事件需要花费一两秒的时间来进行广播。为了加快速度，更新队列设置，可以将延迟更改为 1 秒，然后再查找新的任务。

广播事件

要广播一个事件，需要通过 `Illuminate\Contracts\Broadcasting\ShouldBroadcast` 接口将该事件标记为广播事件。此接口需要添加 `broadcastOn()` 方法，该方法将返回字符串或频道对象的数组，每个数组表示一个 WebSocket 频道。

WebSocket 事件结构

使用 WebSocket 发送的每个事件都有三个主要特征：名称、频道和数据。

事件的名称一般类似于 `user-was-subscribed`，但 Laravel 的默认设置是使用事件的完全限定类名称，例如 `App\Events\UserSubscribed`。可以通过将名称传递给事件类中的可选 `broadcastAs()` 方法来进行自定义。

频道用于描述客户端收到此消息的方式。为每个用户（如 `users.1`、`users.2` 等）以及所有用户（如 `users`）设置频道是非常常见的模式，有时也可能只针对某个账户用户（如 `accounts.1`）。如果所针对的频道是一个私有频道，则需要将频道名称前面加上 `private-` 前缀；如果它是一个 `presence` 频道，则需要将频道名称前面加上 `presence-` 前缀。因此，一个名为 `groups.5` 的私有 Pusher 频道应表示为 `private-groups.5`。如果在 `broadcastOn()` 方法中使用 Laravel 的 `PrivateChannel` 和 `PresenceChannel` 对象，那么它们将负责在频道名称前面自动补充前缀。

数据是与事件相关的信息的有效载荷（通常为 JSON）。消息（也许是关于用户或计划的）可以通过 JavaScript 来执行。

示例 16-16 显示了将 `UserSubscribed` 事件修改为在两个频道上进行广播的过程。一个频道用于用户（确认用户订阅），一个频道用于管理员（通知他们有新的订阅）。

示例16-16 在多个频道广播事件

```
...
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class UserSubscribed extends Event implements ShouldBroadcast
{
    use InteractsWithSockets, SerializesModels;

    public $user;
    public $plan;

    public function __construct($user, $plan)
    {
        $this->user = $user;
        $this->plan = $plan;
    }
    public function broadcastOn()
    {
        // 字符串语法
        return [
            'users.' . $this->user->id,
            'admins'
        ];

        // 频道对象语法
    }
}
```

```

        return [
            new Channel('users.' . $this->user->id),
            new Channel('admins'),
            // 如果是私有频道 : new PrivateChannel('admins'),
            // 如果是 presence 频道 : new PresenceChannel('admins'),
        ];
    }
}

```

默认情况下，事件的公共属性将被序列化为 JSON，并作为广播事件的数据发送。广播 UserSubscribed 事件的数据如示例 16-17 所示。

示例16-17 广播事件数据示例

```

{
    'user': {
        'id': 5,
        'name': 'Fred McFeely',
        ...
    },
    'plan': 'silver'
}

```

可以使用从事件的 broadcastWith() 方法中返回的数据数组来覆盖此值，如示例 16-18 所示。

示例16-18 自定义广播事件数据

```

public function broadcastWith()
{
    return [
        'userId' => $this->user->id,
        'plan' => $this->plan
    ];
}

```

最后，可以使用 onQueue() 方法来指定事件被推送到哪个队列，如示例 16-19 所示。这样做可以使其他队列项目减慢事件广播的速度，如果队列中长时间运行的任务使事件不能及时退出，那么实时 WebSocket 的作用将被削弱。

示例16-19 指定任务运行的队列

```

public function onQueue()
{
    return 'websockets-for-faster-processing'
}

```

接收消息

如果要托管自己的 Redis WebSocket 服务器，Laravel 文档 (<http://bit.ly/2f5lmce>) 中对于如何设置和使用 socket.io 和 ioredis 有非常详细的介绍。

然而，使用 Pusher 更为常见。超过一定规模的计划需要一定花费，但使用 Pusher 有一个免费计划。Pusher 设置了一个简单的 WebSocket 服务器，其中的 JavaScript SDK 可以处理所有的身份验证和频道管理。SDK 可用于 iOS、Android 以及更多其他平台、语言、框架中。



使用 Echo 还是不使用 Echo

下一节将介绍如何编写 JavaScript 前端，以便与有或没有 Echo 的 WebSocket Laravel 进行交互。在没有 Echo 的情况下，这样做还是很有帮助的。但是，如果使用 Echo，编写 JavaScript 前端的大部分代码是没有必要的。建议在编写代码前先阅读下面的内容，同时参考第 376 页的“Laravel Echo (JavaScript 方面)”，选择自己喜欢的方式，然后编写适合的代码。

使用 Pusher 库前，先从 Pusher 账户获取 API 密钥，并在所有频道上订阅，代码如示例 16-20 所示。

示例16-20 Pusher JS的基本用法

```
...
<script src="https://js.pusher.com/3.1/pusher.min.js"></script>
<script>
// 或许这只是一个演示如何在全局情况下将数据导入的例子
var App={
  'userId': 5,
  'pusherKey': 'your-pusher-api-key-here'
};

// 本地
var pusher = new Pusher(App.pusherKey);

var pusherChannel = pusher.subscribe('users.' + App.userId);

pusherChannel.bind('App\\Events\\UserSubscribed', (data) => {
  console.log(data.user, data.plan);
});
</script>
```



转义 JavaScript 中的反斜线

由于 `\` 是 JavaScript 中的控制字符，因此需要写入 `\\` 来表示字符串中的一个反斜线符号，这就是示例 16-20 中每个命名空间段之间有两个反斜线的原因。

要从 Laravel 发布到 Pusher，需要从 Pusher 账户信息中心获取 Pusher 密钥、密码和应用 ID，然后将它们设置在 `.env` 文件中的 `PUSHER_KEY`、`PUSHER_SECRET` 和 `PUSHER_APP_ID` 键下。

如果程序服务在一个窗口中访问嵌入其中的如示例 16-20 所示的 JavaScript 页面，在另一个窗口或终端推送广播事件，运行队列监听器或使用同步驱动程序，并且所有身份验证信息都正确设置，那么我们应该可以实时看到事件日志在 JavaScript 窗口的控制台弹出。

有了这个功能，我们就可以让用户轻松了解他们的数据在应用中的实时状态。我们可以通知用户其他用户的操作，刚刚完成运行了很长时间的进程，或者应用程序对外部操作（如收到的电子邮件或 Webhooks）的响应。



要求

如果想用 Pusher 或 Redis 进行广播，需要引入以下依赖。

- Pusher: `pusher/pusher-php-server:~2.0`
- Redis: `premis/premis:~1.0`

高级广播工具

Laravel 中还有一些其他工具，可以在事件广播中执行更复杂的交互。这些工具是框架特性和 JavaScript 库的组合，称为 Laravel Echo。

在 JavaScript 前端使用 Laravel Echo 时，这些框架功能非常有效，我们将在第 376 页的“Laravel Echo（JavaScript 方面）”中介绍。我们也可以单独使用 Echo，不结合 JavaScript 组件。Echo 将与 Pusher 或 Redis 结合起来工作，下面以 Pusher 为例。

从广播事件中排除当前用户

与 Pusher 结合每个连接都被分配了唯一的一个“Socket ID”，用于标识该套接字连接。另外，任何应被排除在指定接收广播事件之外的给定的套接字（用户）都很容易定义。

若想定义某些事件不应该广播给使用它们的用户时，使用这个特性就会非常简单。假设其他用户创建一项任务时，团队中的每个用户都会收到通知。试想一下，用户想收到刚

刚创建任务的通知吗？不！因此，我们可以使用 `toOthers()` 方法。

实现上述方法，要遵循两个步骤。首先，当 WebSocket 连接初始化时，我们需要设置 JavaScript，将一个特定的 POST 发送到 `/broadcast/socket`。这样做会将 `socket_id` 附加到 Laravel 会话中。Echo 可以完成这个操作，我们也可以手动完成。查看一下 Echo 源代码 (<https://github.com/laravel/echo/>)，看看它的工作原理。

接下来，更新 JavaScript 的每个请求，使其代码中包含该 `socket_id` 的 XSocket-ID 头。示例 16-21 显示了如何在 Vue 或 jQuery 中执行此操作。

示例16-21 在Vue或jQuery中发送套接字ID以及每个AJAX请求

```
// 初始化 echo 之后立刻运行这个程序
// Vue
Vue.http.interceptors.push((request, next) => {
    request.headers['X-Socket-Id'] = Echo.socketId();

    next();
});

// jQuery
$.ajaxSetup({
    headers: {
        'X-Socket-Id': Echo.socketId()
    }
});
```

按照上述方法处理，我们可以排除触发广播事件的用户，这些用户使用 `broadcast()` 方法而非 `event()` 方法，然后链接到 `toOthers()` 方法。

```
broadcast(new UserSubscribed($user, $plan))->toOthers();
```

广播服务提供商

Echo 提供的所有功能都要求 JavaScript 对服务器进行身份验证才能实现。看一下 `App\Providers\BroadcastServiceProvider`，在这里可以定义如何授权用户访问私有频道和 presence 频道。

我们可以采取两个主要措施：定义将用于广播验证路径的中间件；定义频道的授权设置。

如果要使用这些功能，需要先取消 `config/app.php` 中的 `App\Providers\BroadcastServiceProvider::class`。

如果没有使用 Laravel Echo 的功能，则需要手动发送 CSRF 令牌以及身份验证请求，

或者将其添加到 `VerifyCrftToken` 中间件的 `$except` 属性中，从 CSRF 保护中排除 `/broadcasting/auth` 和 `/broadcast/socket`。

为 `WebSocket` 频道绑定授权。私有及 `presence` `WebSocket` 频道需要 `ping`（因特网包探索器）应用程序来了解当前用户是否被授权使用该频道。我们将使用 `Broadcast :: channel()` 方法来定义此授权的规则。



公有频道、私有频道和 `presence` 频道

`WebSocket` 中有三种类型的频道：公有、私有和 `presence`。

公有频道可以被任何用户订阅，无论是否经过身份验证。

私有频道需要终端用户的 `JavaScript` 才能对应用程序进行身份验证，证明用户是经过身份验证的，并被授权加入这个频道。

`presence` 频道是私有频道的一种，但不允许进行消息传递，只记录用户加入或离开频道，并把这些信息提供给应用程序的前端。

`Broadcast::channel()` 有两个参数：第一个表示希望匹配的频道的字符串；第二个表示一个定义了如何为匹配该字符串的频道进行用户授权的闭包。该闭包将传递一个当前用户的 `Eloquent` 模型，将其作为第一个参数，并将其他匹配的 `*` 段作为附加参数来传递。例如当一个带有 `teams.*` 字符串的频道授权定义与频道 `teams.5` 匹配时，将以 `$user` 作为第一个参数，以 `5` 作为第二个参数。

如果正在为私有频道定义规则，则 `Broadcast :: channel()` 闭包将需要返回一个布尔值：该用户是否被授权使用这个频道？如果正在为 `presence` 频道定义规则，则闭包应该为频道中显示的用户返回他们期望得到的可用于 `presence` 频道的数据数组。示例 16-22 说明了两种频道的定义规则。

示例16-22 为私有及`presence` `WebSocket`频道确定授权规则

```
...
class BroadcastServiceProvider extends ServiceProvider
{
    public function boot()
    {
        ...

        // 定义如何认证一个私有频道
        Broadcast::channel('teams.*', function ($user, $teamId) {
            return $user->team_id == $teamId;
        });
    }
}
```

```
// 定义如何认证一个 presence 频道，返回所有数据
// 想要软件知道频道中用户的信息
Broadcast::channel('rooms.*', function ($user, $roomId) {
    if ($user->rooms->contains($roomId)) {
        return [
            'name' => $user->name
        ];
    }
});
```

大家可能想知道这些信息是如何从 Laravel 应用程序中传递到 JavaScript 前端的。Pusher 的 JavaScript 库会向应用程序发送一个 POST，在默认情况下，它将攻击 /pusher/auth，但是，我们可以自定义这一功能（Echo 可提供自定义）去匹配 Laravel 的认证路线 /broadcast/auth，代码如下。

```
var pusher = new Pusher(App.pusherKey, {
    authEndpoint: '/broadcasting/auth'
});
```

示例 16-23 显示了如何在没有 Echo 前端组件的情况下对示例 16-20 中的私有和 presence 频道进行调整。

例16-23 用于私有和presence频道的Pusher JS的基本用法

```
...
<script src="https://js.pusher.com/3.1/pusher.min.js"></script>
<script>
    // 或许这只是一个演示如何在全局情况下将数据导入的例子
    var App={
        'userId': {{ auth()->user()->id }},
        'pusherKey': 'your pusher key here'
    };

    // 本地
    var pusher = new Pusher(App.pusherKey, {
        authEndpoint: '/broadcasting/auth'
    });

    // 私有频道
    var privateChannel = pusher.subscribe('private-teams.1');

    privateChannel.bind('App\\Events\\UserSubscribed', (data) => {
        console.log(data.user, data.plan);
    });
```

```
// presence 频道
var presenceChannel = pusher.subscribe('presence-rooms.5'); console.

    log(presenceChannel.members);
</script>
```

我们现在有能力根据用户是否通过给定频道的授权规则来向用户发送 WebSocket 消息。我们还可以跟踪哪些用户在网站的特定组或其他部分中处于活动状态，并向每个用户显示同一组中其他用户的相关信息。

Laravel Echo（JavaScript 方面）

Laravel Echo 由两部分组成：高级框架功能，以及 JavaScript 软件包，该软件包利用上面提到的功能，极大程度上减少了编写高级 WebSocket-based 前端所需的样板代码量。Echo JavaScript 软件包可以轻松处理身份验证、授权及私人订阅和 presence 频道等问题。Echo 可用于 Pusher JS（pusher）或 socket.io（Redis）的 SDK。

在项目中引入 Echo

要在项目的 JavaScript 中使用 Echo，需要用 `npm install --save` 将它添加到 *package.json*（请务必带上适当的 Pusher 或 socket.io SDK）中。

```
npm install pusher-js --save
npm install laravel-echo --save
```

假设有一个基本的文件，通过 Webpack 的文件运行应用程序，如示例 16-24 所示。

示例16-24 通过Webpack编译app.js

```
const elixir = require('laravel-elixir');
elixir(mix => {
    mix.webpack('app.js');
});
```

现在，创建一个基本的 *resources/assets/js/app.js* 文件（参见示例 16-25）来引入依赖关系并初始化 Echo。

示例16-25 在app.js中初始化Echo

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
```

```
    key: 'your-pusher-key'  
  });
```

// 在此处添加 Echo 绑定

对于 CSRF 保护，还需要在 HTML 模板中添加 csrf-token <meta> 标记，如下所示。

```
<meta name="csrf-token" content="{ {{ csrf_token() }}">
```

当然，请记住链接到 HTML 模板中编译的 *app.js*。

```
<script src="/js/app.js"></script>
```

现在可以开始了。

使用 Echo 进行基本事件广播

这与已经使用过的 Pusher JS 有所不同，示例 16-26 是一个简单的代码示例，用于显示如何使用 Echo 来监听公有频道的基本事件信息。

示例16-26 监听Echo中的公有频道

```
var currentTeamId = 5; // 可能在别处设置
```

```
Echo.channel('teams.' + currentTeamId)  
  .listen('UserSubscribed', (data) => {  
    console.log(data);  
  });
```

Echo 提供了一些订阅各种频道的方法：`channel()` 将订阅公有频道。注意，当监听带有 Echo 的事件时，可以忽略完整的事件名称空间，只需要监听该事件的唯一名称即可。现在可以访问与事件一起传递的公共数据了。

也可以链接 `listen()` 处理程序，如示例 16-27 所示。

示例16-27 在Echo中链接事件监听器

```
Echo.channel('teams.' + currentTeamId)  
  .listen('UserSubscribed', (data) => {  
    console.log(data);  
  })  
  .listen('UserCanceled', (data) => {  
    console.log(data);  
  });
```



记住要编译

你是否运作过这些代码示例，并且在浏览器中没有看到任何更改呢？确保运行 `gulp`（如果运行一次）或 `gulp` 监视（运行监听器）来编译代码。如果还没有，请确保模板中包含 `app.js`。

私有频道和基本身份验证

Echo 还有一种订阅私有频道的方法：`private()`。它与 `channel()` 的工作方式相同，但它需要在 `BroadcastServiceProvider` 中设置频道授权定义。此外，与 SDK 不同，这里不需要将私有信息放在频道名称的前面。

示例 16-28 展示了如何监听名为 `privateteams.5` 的私有频道。

示例16-28 用Echo监听私有频道

```
var currentTeamId = 5; // 可能在别处设置

Echo.private('teams.' + currentTeamId)
  .listen('UserSubscribed', (data) => {
    console.log(data);
  });
```

presence 频道

Echo 使加入和监听 `presence` 频道的事件变得更加简单。使用 `join()` 方法绑定此频道，如示例 16-29 所示。

示例16-29 加入presence频道

```
var currentTeamId = 5; // 可能在别处设置

Echo.join('teams.' + currentTeamId)
  .here((members) => {
    console.log(members);
  });
```

`join()` 订阅 `presence` 频道，`here()` 允许定义用户连接时的行为以及其他用户连接或离开 `presence` 频道时的行为。

试想，一个 `presence` 频道就像一个“在线用户”的侧边栏。当首次加入 `presence` 频道时，将会调用 `here()` 回调，并提供当时所有成员的列表。任何时候任何成员加入或离开，该回调将再次被调用更新的列表。这里没有消息传递，但是可以播放声音、更新成员的页面列表，或者在响应这些动作时进行任何操作。

还有其他可以单独或链接使用个别事件的具体方法（参见示例 16-30）。

示例16-30 监听特定的presence事件

```
var currentTeamId = 5; // 可能在别处设置
```

```
Echo.join('teams.' + currentTeamId)
  .then((members) => {
    // 加入时运行
    console.table(members);
  })
  .joining((joiningMember, members) => {
    // 当另一个成员加入时运行
    console.table(joiningMember);
  })
  .leaving((leavingMember, members) => {
    // 当另一个成员离开时运行
    console.table(leavingMember);
  });
```

不包括当前用户

我们已经在前面的章节中讨论过这个问题，如果想要排除当前的用户，要使用 `broadcast()` 全局助手替代 `event()` 全局助手，然后在广播调用后链接 `toOthers()` 方法。

可以看到，Echo JavaScript 库自己不会进行任何操作——但是它使许多常见的任务更加简单，并且为常见的 WebSocket 任务提供了一种更简洁、更富表现力的语法。

订阅 Echo 通知

Laravel 通知带有一个广播驱动程序开箱即用，作为广播事件推送通知。可以通过使用 Echo 中的 `Echo.notification` 订阅这些通知，如示例 16-31 所示。

示例16-31 使用Echo订阅通知

```
Echo.private('App.User.' + userId)
  .notification((notification) => {
    console.log(notification.type);
  });
```

调度程序

如果曾经编写过 cron 任务，那么你一定希望得到一个更好的工具。因为当前的程序不仅语法烦琐且难以记忆，对于应用程序而言也无法存储到版本控制。

Laravel 调度程序使处理调度任务变得简单。可以使用代码编写调度中的任务，然后在应用程序中指定一个 cron 定时任务：每分钟运行一次 `php artisan schedule:run`。每当这个 Artisan 命令运行时，Laravel 就会检查调度定义，以找出是否有任何调度任务应该运行。

下面通过 cron 任务来定义该命令。

```
* * * * * php /home/myapp.com/artisan schedule:run >> /dev/null 2>&1
```

这里可以调度许多任务类型，并可以应用许多时间框架来调度这些任务。

`app/Console/Kernel.php` 有一个名为 `$schedule` 的方法，可以在其中定义任何要调度的任务。

可用任务类型

首先来看最简单的选项：闭包，每分钟运行一次（参见示例 16-32）意味着，每次 cron 任务都会触发 `schedule:run` 命令，它会调用此闭包。

示例16-32 调度一个闭包，每分钟运行一次

```
// app/Consoles/Kernel.php
public function schedule($schedule)
{
    $schedule->call(function () {
        dispatch(new CalculateTotals);
    })->everyMinute();
}
```

这里还有两种类型的任务可以安排：Artisan 和 shell 命令。

可以像从命令行调用 Artisan 命令一样，通过传递其语法来调度 Artisan 命令。

```
$schedule->command('scores:tally --reset-cache')->everyMinute();
```

可以运行所有使用 PHP 的 `exec()` 的方法运行的 shell 命令。

```
$schedule->exec('/home/myapp.com/bin/build.sh')->everyMinute();
```

可用时间框架

调度程序的优点在于，不仅可以在代码中定义任务，也可以在代码中对任务进行调度。Laravel 会根据时间因素评估给定任务的运行时刻。对于 `everyminute()` 来说这很容易，对于 Laravel 也很简单，即使是最复杂的请求也是如此。这里介绍一个简单的 Laravel 的

怪异定义。

```
$schedule->call(function () {  
    // 在每个星期天的 23:50 运行一次  
})->weekly()->sundays()->at('23:50');
```

请注意，这里可以将时间结合在一起：定义频率、指定星期几和具体时间。当然我们还可以进行更多操作。

表 16-1 展示了在调度任务时使用的潜在日期 / 时间修改器列表。

表16-1 调度程序的日期/时间修改命令

| 命令 | 描述 |
|----------------------------------|---|
| ->timezone('America/Detroit') | 为日程设置时区 |
| ->cron('* * * * *') | 使用传统 cron 符号来定义日程 |
| ->everyMinute() | 每分钟运行一次 |
| ->everyFiveMinutes() | 每 5 分钟运行一次 |
| ->everyTenMinutes() | 每 10 分钟运行一次 |
| ->everyThirtyMinutes() | 每 30 分钟运行一次 |
| ->hourly() | 每小时运行一次 |
| ->daily() | 每天午夜时运行一次 |
| ->dailyAt('14:00') | 每天 14:00 运行一次 |
| ->twiceDaily(1, 14) | 每天 1:00 和 14:00 各运行一次 |
| ->weekly() | 每星期运行一次（星期天的午夜） |
| ->weeklyOn(5, '10:00') | 每个星期五 10:00 运行一次 |
| ->monthly() | 每个月运行一次（1 号的午夜） |
| ->monthlyOn(15, '23:00') | 每个月 15 号的 23:00 运行一次 |
| ->quarterly() | 每个季度运行一次（1 月 1 日、4 月 1 日、7 月 1 日、10 月 1 日的午夜） |
| ->yearly() | 每年运行一次（1 月 1 号午夜） |
| ->when(closure) | 将任务限制在当闭包返回 true 的时候 |
| ->skip(closure) | 将任务限制在当闭包返回 false 的时候 |
| ->between('8:00', '12:00') | 将任务限制在给定时间之间 |
| ->unlessBetween('8:00', '12:00') | 将任务限制在给定时间之外的任意时间 |
| ->weekdays() | 限定在工作日 |
| ->sundays() | 限定在星期天 |

| 命令 | 描述 |
|----------------|--------|
| ->mondays() | 限定在星期一 |
| ->tuesdays() | 限定在星期二 |
| ->wednesdays() | 限定在星期三 |
| ->thursdays() | 限定在星期四 |
| ->fridays() | 限定在星期五 |
| ->saturdays() | 限定在星期六 |

大多数命令都可以被一个接一个地结合起来，但是没有意义的组合不能被结合。

示例 16-33 展示了一些可以考虑的组合。

示例16-33 一些调度事件示例

```
// 均在每个星期天的 23:50 运行一次
$schedule->command('do:thing')->weeklyOn(0, '23:50');
$schedule->command('do:thing')->weekly()->sundays()->at('23:50');

// 工作日上午 8 点到下午 5 点之间，每小时运行一次
$schedule->command('do:thing')->weekdays()->hourly()->when(function () {
    return date('H') >= 8 && date('H') <= 17;
});

// 使用 Laravel 5.3 中的新关键词 "between" 定义
// 工作日上午 8 点到下午 5 点之间每小时运行一次
$schedule->command('do:thing')->weekdays()->hourly()->between('8:00', '17:00');

$schedule->command('do:thing')->everyThirtyMinutes()->skip(function () {
    return app('SkipDetector')->shouldSkip();
});
```

阻塞和重叠

如果想要避免任务间彼此重叠——比如希望一个任务每分钟都运行，这可能会导致运行时间长于一分钟——可以通过 `withoutOverlapping()` 方法来结束调度链。如果之前的任务实例依然在运行，那么使用这种方法可以跳过该实例。

```
$schedule->command('do:thing')->everyMinute()->withoutOverlapping();
```

处理任务输出

有时，调度任务的输出是非常重要的，无论是用于日志记录、通知，还是仅仅确保任务

运行。

如果想要编写一个将任务返回的输出结果传递到是个文件中的示例，则可以使用 `sendOutputTo()`，如下所示。

```
$schedule->command('do:thing')->daily()->sendOutputTo($filePath);
```

如果要将其附加到文件，则可以使用 `appendOutputTo()`。

```
$schedule->command('do:thing')->daily()->appendOutputTo($filePath);
```

如果要将输出结果的电子邮件发送给指定的收件人，请先写入文件，然后添加 `emailOutputTo()`。

```
$schedule->command('do:thing')
->daily()
->sendOutputTo($filePath)
->emailOutputTo('me@myapp.com');
```

确保电子邮件设置在 Laravel 的基本电子邮件配置中处于正确配置状态。



闭包调度事件不能发送输出结果

`sendOutputTo()`、`appendOutputTo()` 和 `emailOutputTo()` 方法只用于 `command` 调度任务。不能把它们用作闭包。

如果想要发送一些输出结果，以确保任务正确运行，可以采用一些提供种正常运行时间监控的服务，效果最显著的是 Laravel Envoyer（零停机部署服务，也提供正常运行时间监控）和 Dead Man's Snitch，这是纯粹用于监控 cron 任务正常运行时间的工具。

这些服务并不希望通过电子邮件发送给它们，而是希望使用 HTTP “ping”，所以 Laravel 可以通过 `pingBefore()` 和 `thenPing()` 来轻松实现。

```
$schedule->command('do:thing')
->daily()
->pingBefore($beforeUrl)
->thenPing($afterUrl);
```

如果想要使用 ping 功能，则需要使用 Composer 在 Guzzle 中输入 `"guzzlehttp/guzzle": "~5.3|~6.0"`。

任务钩子

说到在任务之前和之后运行的内容，就要提到使用 `before()` 和 `after()` 的钩子。

```

$schedule->command('do_thing')
    ->daily()
    >before(function () {
        // 准备
    })
    ->after(function () {
        // 清理
    });

```

测试

测试队列任务（或者队列中的其他内容）十分容易。在测试的配置文件 *phpunit.xml* 中，默认情况下，`QUEUE_DRIVER` 环境变量设置为同步。这意味着测试将直接在代码中同步运行任务或其他队列任务，而无须依赖任何队列系统，可以像其他代码一样测试它们。

但是，如果只想检查任务是否被触发，则可以使用 `expectsJobs()` 方法来执行此操作，如示例 16-34 所示。

示例16-34 断言特定类的任务发送

```

public function test_changing_number_of_subscriptions_crunches_reports()
{
    $this->expectsJobs(App\Jobs\CrunchReports::class);

    ...
}

```

或者，在 Laravel 5.3 及更高版本中，可以针对具体工作使用断言，如示例 16-35 所示。

示例16-35 使用闭包验证分派任务是否符合给定的条件

```

use Illuminate\
public function test_changing_subscriptions_triggers_crunch_job()
{
    ...

    Bus::assertDispatched(CrunchReports::class, function ($e) {
        return $e->subscriptions->contains(5);
    });

    // 也可以使用 assertNotDispatched() 方法
}

```

要触发测试事件，有三个选择。首先，可以测试所期望的行为发生的情况，而不必考虑事件本身。

5.2 其次，可以明确断言事件触发，如示例 16-36 所示，这在 Laravel 5.2 中有效。

例16-36 断言指定类的事件被触发

```
public function test_usersubscribed_event_fires()
{
    $this->expectsEvents(App\Events\UserSubscribed::class);
    ...
}
```

5.3 最后，可以针对被触发的事件运行测试，如示例 16-37 所示，这是 Laravel 5.3 中的新功能。

示例16-37 使用闭包验证一个触发事件是否符合给定的条件

```
public function test_usersubscribed_event_fires()
{
    ...

    Event::assertFired(UserSubscribed::class, function ($e) {
        return $e->user->email = 'user-who-subscribed@mail.com';
    });

    // 也可以使用 assertNotFired() 方法
}
```

另一个常见的场景是，在测试偶然触发事件的代码时，希望在测试期间禁用事件监听器。这时可以使用 `noEvents()` 方法禁用事件系统，如示例 16-38 所示。

示例16-38 测试期间禁用事件监听器

```
public function test_something_subscription_related()
{
    $this->withoutEvents();
    ...
}
```

本章小结

队列允许将应用程序的代码块从用户交互的同步流中分离出来，置于“队列工作者”处理的命令列表中。这样允许用户恢复与应用程序的交互，而较慢的进程则在后台异步处理。

任务是结构化的类，其目的是封装应用程序行为，以便将其推入队列。

Laravel 的事件系统遵循 pub / sub 或观察者模式，允许从应用程序的一部分中发送事

件的通知，其他部分将监听器绑定到那些通知中，以定义应该发生什么行为。使用 WebSocket，事件也可以被广播到前端客户端。

Laravel 的调度程序简化了调度任务，将每分钟的 cron 任务指向 `php artisan schedule:run`，然后甚至可以对最复杂的时间要求进行任务调度，Laravel 将处理所有的时间要求。

助手和集合

我们已经在本书中介绍了许多全局函数：使得执行普通任务更容易的助手，如任务的 `dispatch()`、事件的 `event()` 和依赖性解析的 `app()` 等。我们还在第 8 章中介绍了 Laravel 的集合或加强版数组。

在本章中，我们将介绍一些更常见、功能更强大的助手以及使用集合进行编程的基础知识。

助手

读者可以在帮助文档（<https://laravel.com/docs/helpers>）中找到 Laravel 提供的完整助手列表，而这里将介绍一些更有用的功能。

数组

PHP 的原生数组操作功能十分强大，但有时我们需要进行一些常见的操作，这些操作涉及庞杂的循环和逻辑检查。这时，Laravel 的数组助手可以提供一些常见的数组操作，如下所示。

```
array_first($array, $closure, $default = null)
```

返回通过测试的、闭包中定义的第一个数组值，可以选择将默认值设置为第三个参数，代码如下。

```
$people = [  
    [  
        'email' => 'm@me.com',  
        'name' => 'Malcolm Me'  
    ],  
]
```

```

        [
            'email' => 'j@jo.com',
            'name' => 'James Jo'
        ]
    ];

    $value = array_first($people, function ($key, $person) {
        return $person['email'] == 'j@jo.com';
    });

```

`array_get($array, $key, $default = null)`

该方法使得从数组中获取值变得容易，并且有两个附加的好处：请求一个不存在的键（可以使用第三个参数提供默认值时），不会抛出错误；可以使用点号来遍历嵌套的数组。如下所示。

```

$array = ['owner' => ['address' => ['line1' => '123 Main St.']]];

$line1 = array_get($array, 'owner.address.line1', 'No address');
$line2 = array_get($array, 'owner.address.line2');

```

`array_has($array, $key)`

使用点号遍历嵌套的数组可以很容易地检查出数组中是否具有特定的值，如下所示。

```

$array = ['owner' => ['address' => ['line1' => '123 Main St.']]];

if (array_has($array, 'owner.address.line2')) {
    // 完成某些功能
}

```

`array_pluck($array, $key, $indexKey)`

返回与所提供的键相对应的值的数组，代码如下。

```

$array = [
    ['owner' => ['id' => 4, 'name' => 'Tricia']],
    ['owner' => ['id' => 7, 'name' => 'Kimberly']],
];

$array = array_pluck($array, 'owner.name');

// 返回 ['Tricia', 'Kimberly'];

```

如果希望将返回的数组作为源数组的另一个值来键入，则可以将该值的点号引用作为第三个参数，如下所示。


```
$array = array_pluck($array, 'owner.name', 'owner.id');

// 返回 [4 => 'Tricia', 7 => 'Kimberly'];
```

字符串

与数组一样，有一些字符串的处理和检查也可能与本地 PHP 函数有关，但处理起来可能比较麻烦。使用 Laravel 助手使一些常见的字符串操作变得更快、更简单，如下所示。

`e($string)`

`htmlentities()` 的别名。准备一个（通常是用户提供的）在 HTML 页面上能够安全响应的字符串，如下所示。

```
e('<script>do something nefarious</script>');

// 返回 &lt;script&gt;，实现某些功能 nefarious&lt;/script&gt;
```

`starts_with($haystack, $needle)`, `ends_with($haystack, $needle)` 和

`str_contains($haystack, $needle)`

返回一个布尔值，表示提供的“haystack”字符串是否以“needle”字符串开始、结束，或是否包含“needle”字符串，如下所示。

```
if (starts_with($url, 'https')) {
    // 完成某些功能
}

if (ends_with($abstract, '...')) {
    // 完成某些功能
}

if (str_contains($description, '1337 h4x0r')) {
    // 完成某些功能
}
```

`str_limit($string, $numCharacters, $concatenationString = '...')`

将字符串限制为所提供的字符数。如果字符串小于限制，则返回字符串；如果超出限制，则修剪为提供的字符数，然后追加 ... 或提供的连接字符串，代码如下所示。

```
$abstract = str_limit($loremIpsum, 30);
```

```
// 返回 "Lorem ipsum dolor sit amet, co..."

$abstract = str_limit($loremIpsum, 30, "&hellip;");

// 返回 "Lorem ipsum dolor sit amet, co&hellip;"
```

`str_is($pattern, $string)`

无论给定的字符串是否匹配给定的模式，都返回一个布尔值。该模式可能是一个正则表达式，或者我们也可以使用通配符星号表示位置，如下所示。

```
str_is('*.dev', 'myapp.dev'); // 正确
str_is('*.dev', 'myapp.dev.co.uk'); // 不正确
str_is('*dev*', 'myapp.dev'); // 正确
str_is('*myapp*', 'www.myapp.dev'); // 正确
str_is('my*app', 'myfantasticapp'); // 正确
str_is('my*app', 'myapp'); // 正确
```



如何将正则表达式传递给 `str_is()`

如果对可以传递给 `str_is()` 的正则表达式模式感到好奇，请查看此处的功能定义（缩短为空格），查看它的工作原理。请注意，它是 `Illuminate\Support\Str::is` 的别名。

```
public function is($pattern, $value)
{
    if ($pattern == $value) return true;

    $pattern = preg_quote($pattern, '#');
    $pattern = str_replace('\*', '.*', $pattern);
    return (bool) preg_match(
        ,#^' . $pattern . '\z#u',
        $value
    );
}
```

`str_random($length)`

返回指定长度的字母与数字混合的随机字符串，如下所示。

```
$hash = str_random(64);

// 示例：J40uNwAvY60wE4BPewxu7BZFQEmxEHmGiLmQncj0ThMGJK705Kfgptyb9uIwspmh
```

```
str_slug($string, $separator = '-')
```

从一个字符串中返回一个 URL-friendly slug（通常用于创建名称或标题的 URL 段）。

```
str_slug('How to Win Friends and Influence People');
```

```
// 返回 'how-to-win-friends-and-influence-people'
```

应用路径

在处理文件系统时，获取和保存文件目录的操作通常很乏味。使用助手可以快速访问并找到应用程序中最重要目录的完全限定路径。

请注意，每个助手都可以无参数调用。但是如果传递一个参数，它将被附加到正常的目录字符串中，并作为一个整体返回。

```
app_path($append = '')
```

返回应用程序目录的路径，如下所示。

```
app_path();
```

```
// 返回 /home/forge/myapp.com/app
```

```
base_path($append = '')
```

返回应用程序根目录的路径，如下所示。

```
base_path();
```

```
// 返回 /home/forge/myapp.com
```

```
config_path($append = '')
```

返回应用程序中配置文件的路径，如下所示。

```
config_path();
```

```
// 返回 /home/forge/myapp.com/config
```

```
database_path($append = '')
```

返回应用程序中数据库文件的路径，如下所示。

```
database_path();
```

```
// 返回 /home/forge/myapp.com/database
```

```
storage_path($append = '')
```

返回应用程序中存储目录的路径，如下所示。

```
storage_path();
```

```
// 返回 /home/forge/myapp.com/storage
```

URL

部分前端文件路径是一致的，但有些类型也会令人费解（例如资源路径），我们将在这里介绍一些方便使用的方式。有些前端文件路径可以根据路径定义进行移动，或根据新文件使用 Elixir 来进行版本化，因此对于确保所有链接和资源正常工作而言，助手的作用至关重要。

```
action('Controller@method', $params = [], $absolute = true)
```

假设一个控制器方法有一个单独映射的 URL，给定一个控制器和方法名对（用 @ 符号分开），以返回正确的 URL，如下所示。

```
<a href="{ action('PeopleController@index' )}">See all People</a>
```

```
// 返回 <a href="http://myapp.com/people">See all People</a>
```

如果控制器方法需要参数，则可以将它们作为第二个参数传递（如果有多个必需的参数，则将其作为数组进行传递）。如果想要看起来清晰一些，则可以键入它们，但是要确保它们的顺序是正确的，如下所示。

```
<a href="{ action('PeopleController@show', ['id' => 3])}">See Person #3</a>
```

```
// 或者
```

```
<a href="{ action('PeopleController@show', [3] )}">See Person #3</a>
```

```
// 返回 <a href="http://myapp.com/people/3">See Person #3</a>
```

如果将 `false` 传递给第三个参数，那么链接将生成相对路径的形式（如 `/people/3`），此时不会生成绝对路径的形式（如 `http://myapp.com/people/3`）。

```
route($routeName, $params = [], $absolute = true)
```

如果路径具有名称（使用路径定义中的名称），则将返回该路径的 URL。

```
<a href="{ route('people.index' )}">See all People</a>
```

```
// 返回 <a href="http://myapp.com/people">See all People</a>
```

如果路由定义需要参数，则可以将它们作为第二个参数传递（如果需要多个必需的参数，则将其作为数组）。同样，如果想要清晰一些，则可以键入它们，但是要确保它们的顺序是正确的，如下所示。

```
<a href="{{ action('people.show', ['id' => 3]) }}">See Person #3</a>
```

```
// 或者
```

```
<a href="{{ action('people.show', [3]) }}">See Person #3</a>
```

```
// 返回 <a href="http://myapp.com/people/3">See Person #3</a>
```

如果将错误传递给第三个参数，则链接将生成相对路径形式而不是绝对路径形式。

`url($string)` 和 `secure_url($string)`

给定任意路径字符串，将其转换为完全限定的 URL。`secure_url()` 与 `url()` 相同，但强制作用于 HTTPS。

```
url('people/3');
```

```
// 返回 http://myapp.com/people/3
```

如果没有传递任何参数，则会提供一个 `Illuminate\Routing\UrlGenerator` 实例，这样可以使这些方法链接起来。

```
url()->current();
```

```
// 返回 http://myapp.com/abc
```

```
url()->full();
```

```
// 返回 http://myapp.com/abc?order=reverse
```

```
url()->previous();
```

```
// 返回 http://myapp.com/login
```

```
// 在 UrlGenerator 中还有很多别的方法可用
```

`elixir($filePath)`

如果使用 Elixir 的版本系统对资源进行版本控制，需要给定非版本化路径名称，然后为版本化的文件返回完全限定的 URL。

```
<link rel="stylesheet" href="{{ elixir('css/app.css') }}">
```

// 返回一些如 `/build/css/app-eb555e38.css` 之类的内容

Misc（宏指令结构技术体系）

建议读者自学一些其他全局助手的相关知识。最好查看整个列表，但是这里提到的部分列表也值得一看。

`abort($code, $message, $headers)`, `abort_unless($boolean, $code, $message, $headers)`, and `abort_if($boolean, $code, $message, $headers)`

抛出 HTTP 异常。`abort()` 抛出定义的异常：如果第一个参数为 `false`，则 `abort_unless()` 抛出异常；如果第一个参数为 `true`，则 `abort_if()` 抛出异常。

```
public function controllerMethod(Request $request)
{
    abort(403, 'You shall not pass');
    abort_unless($request->has('magicToken'), 403);
    abort_if($request->user()->isBanned, 403);
}
```

`auth()`

返回 Laravel 认证器的一个实例。例如 `Auth facade`，可以使用它来获取当前用户信息，检查登录状态等，如下所示。

```
$user = auth()->user();

if (auth()->check()) {
    // 完成某些功能
}
```

`back()`

生成“重定向返回（redirect back）”响应，将用户发送到以前的位置，如下所示。

```
Route::get('post', function () {
    ...
    if ($condition) {
        return back();
    }
});
```

`collect($array)`

获取数组并返回相同的数据，然后将其转化为集合，如下所示。

```
$collection = collect(['Rachel', 'Hototo']);
```

这里只介绍少许关于集合的内容。

```
config($key)
```

返回点号配置项的值，如下所示。

```
$defaultDbConnection = config('database.default');
```

```
csrf_field() 和 csrf_token()
```

返回一个完整的 HTML 隐藏输入 (`csrf_field()`) 或者适当的 token 值 (`csrf_token()`)，用于将 CSRF 验证添加到表单提交中，如下所示。

```
<form>
    {{ csrf_field() }}
</form>

// 或者

<form>
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

```
dd($variable...)
```

“备份文件系统和停止系统 (dump and die)” 的缩写，在所有提供的参数上运行 `var_dump()`，然后在 `exit()` 中运行以退出应用程序（此处用于调试），如下所示。

```
...
dd($var1, $var2, $state);
```

```
env($key, $default = null)
```

返回给定键的环境变量。

```
$key = env('API_KEY', '');
```



在 config 文件外使用 env()

如果在配置文件之外的其他地方使用 `env()` 调用，那么 Laravel 中的某些功能（包括一些缓存和优化功能）将都不可用。获取环境变量的最好方法是为我们想要的环境特定事务设置配置项，使这些配置项读取环境变量，然后可以在应用程序的任何地方引用配置变量，如下所示。

```
// config/services.php
```

```

        return [
            bugsnag' => [
                'key' => env('BUGSNAG_API_KEY')
            ]
        ];

        // 在控制器中
        $bugsnag = new Bugsnag(config('services.bugsnag.key'));

```

`dispatch($job)`

分派任务，如下所示。

```
dispatch(new EmailAdminAboutNewUser($user));
```

`event($event)`

触发事件，如下所示。

```
event(new ContactAdded($contact));
```

`factory($entityClass)`

返回给定类的工厂构建器（factory builder）的实例，如下所示。

```
$contact = factory(App\Contact::class)->make();
```

`old($key, $default = null)`

如果该表单存在，则返回旧值（来自最后一个用户的表单提交），如下所示。

```
<input name="name" value="{{ old('value', 'Your name here') }}"
```

`redirect($path)`

返回给定路径的重定向响应，如下所示。

```
Route::get('post', function () {
    ...

    return redirect('home');
});
```

没有参数，生成一个 `Illuminate\Routing\Redirector` 实例类。

`response($body, $status, $headers)`

如果通过参数传递，则返回一个预构建的 `Response` 实例；如果没有响应传递，则

返回 Response 工厂实例，如下所示。

```
return response('OK', 200, ['X-Header-Greatness' => 'Super great']);
```

```
return response()->json(['status' => 'success'], 200);
```

```
view($viewPath)
```

返回一个视图实例，如下所示。

```
Route::get('home', function () {  
    return view('home'); // 得到 /resources/views/home.blade.php  
});
```

集合

集合是 Laravel 中提供的最强大但尚未得到认可的工具之一。我们在第 8 章的“Eloquent Collection”中介绍了相关的内容，这里进行简单的回顾。

集合本质上由功能强大的数组组成。数组遍历时通常要进行数组传递（通过 `array_walk()`、`array_map()`、`array_reduce()` 等），上述方法往往具有容易使人混淆的不一致的方法签名，但在每个集合中却是一致的、干净的、可链接的方法。可以尝试函数式编程，并将其映射、减少及过滤为更干净的代码。

这里将介绍 Laravel 的集合与集合管道编程的一些基础知识，对于更深入的讲解，请查看 Adam Wathan 所著的 *Refactoring to Collections* 一书。

集合的基础

使用集合不是 Laravel 中的独有的。许多语言都可以在数组中使用集合式编程，但是非常遗憾，原生的 PHP 是不支持集合式编程的。

使用 PHP 的 `array*()` 功能，可以将如示例 17-1 所示的复杂代码简化成如示例 17-2 中的样子。

示例17-1 常见但复杂的foreach循环

```
$users = [...];
```

```
$admins = [];
```

```
foreach ($users as $user) {  
    if ($user['status'] == 'admin') {
```

```

        $user['name'] = $user['first'] . ' ' .
        $user['last'];
        $admins[] = $user;
    }
}

return $admins;

```

例17-2 使用本地PHP函数对foreach循环进行重构

```

$users = [...];

return array_map(function ($user) {
    $user['name'] = $user['first'] . ' ' . $user['last'];
    return $user;
}, array_filter($users, function ($user) {
    return $user['status'] == 'admin';
}));

```

这里去掉了一个临时变量 (+\$admins+), 并将一个混淆的 foreach 循环转化为两个不同的操作: map 和 filter。

问题是, PHP 的数组操作函数非常令人困惑。看看以下例子: array_map() 先使用闭包, 然后使用数组, 但 array_filter() 先使用数组, 后使用闭包。此外, 如果增加复杂性, 将会产生函数之间的层层嵌套, 造成混乱。

Laravel的集合利用了PHP数组操作方法的强大功能, 并提供了一种干净、流畅的语法——甚至还添加了在PHP数组操作工具箱中不存在的许多新方法。现在使用 collect() 助手将数组转化为Laravel集合, 如示例17-3所示。

示例17-3 通过Laravel的集合重构foreach循环

```

$users = collect([...]);
return $users->filter(function ($user) {
    return $user['status'] == 'admin';
})->map(function ($user) {
    $user['name'] = $user['first'] . ' ' . $user['last'];
    return $user;
});

```

这并不是最简化的示例。代码的简化改善了其使用体验, 这是很常见的。

再看原始的例子, 语句是多么粗糙。直到理解了所有的代码, 才会明白其中的内容。集合的最大好处是将所需的操作分解成简单的、离散的、可理解的任务。具体如下。

```

$users = [...]

```

```
$countAdmins = collect($users)->filter(function ($user) {  
    return $user['status'] == 'admin'  
})->count();
```

或类似下面这样。

```
$users = [...];  
$greenTeamPoints = collect($users)->filter(function ($user) {  
    return $user->team == 'green';  
})->sum('points');
```

几种方法

我们能做的比这里所讲的要更多。参见 Laravel 集合文档 (<https://laravel.com/docs/collections>)，了解更多可以使用的方法。下面为大家介绍一些核心方法。

all() 和 toArray()

如果要将集合转化为数组，可以使用 `all()` 或者 `toArray()`。`toArray()` 不仅能处理集合，而且还能将所有 Eloquent 对象都转化为数组。`all()` 仅将集合转化为数组：包含在集合中的对象将被保存为 Eloquent 对象。示例如下。

```
$users = User::all();  
  
$users->toArray();  
  
/* 返回  
    [  
        ['id' => '1', 'name' => 'Agouhanna'],  
        ...  
    ]  
*/  
  
$users->all();  
  
/* 返回  
    [  
        Eloquent Object { id : 1, name: 'Agouhanna' },  
        ...  
    ]  
*/
```

filter() 和 reject()

当你想要通过检查一个闭包的每个条目来获取原始集合的一个子集时，可以使用

`filter()` (当闭包返回 `true` 时保留一个条目) 或 `reject()` (当闭包返回 `false` 时保留一个条目), 如下所示。

```
$users = collect([...]);
$admins = $users->filter(function ($user) {
    return $user->isAdmin;
});

$paidUsers = $users->reject(function ($user) {
    return $user->isTrial;
});
```

`where()`

`where()` 在给定的键与给定的值相等的情况下, 可以轻松提供原始集合的子集。所有 `where()` 能完成的, 也都可以使用 `filter()` 来完成, 它是一个常见场景的快捷方式。

```
$users = collect([...]);
$admins = $users->where('role', 'admin');
```

`first()` 和 `last()`

如果只是想从集合中获得一个单独的项目, 可以使用 `first()` 从列表的起始位置进行拉取, 或者使用 `last()` 从列表的尾端进行拉取。

如果调用没有参数的 `first()` 或 `last()` 方法, 那么只会得到集合中的第一项或最后一项。但是如果传递任一闭包, 则不仅会得到集合中的第一项或最后一项, 而且当其传递到闭包时会返回 `true`。

当你想得到集合中的第一或最后一项时, 也许会这样做。即使只期望得到一项, 有时这也是最简单的方法, 如下所示。

```
$users = collect([...]);
$owner = $users->first(function ($user) {
    return $user->isOwner;
});

$firstUser = $users->first();
$lastUser = $users->last();
```

还可以向每个方法传递第二个参数, 这是一个默认值, 它将作为一个备选结果, 在闭包不提供任何结果时生效。

each()

如果想对一个集合中的每一项都进行一些操作，但不修改项目或集合本身，则可以使用 `each()`，如下所示。

```
$users = collect([...]);
$users->each(function ($user) {
    dispatch(new EmailUserAThing($user));
});
```

map()

如果想对集合中的所有项目进行迭代，对它们进行更改，并返回一个包含所有更改的新集合，那么需要使用 `map()`，如下所示。

```
$users = collect([...]);
$users = $users->map(function ($user) {
    return [
        'name' => $user['first'] . ' ' . $user['last'],
        'email' => $user['email']
    ];
});
```

reduce()

如果想从集合中得到一个单一的结果，如一个计数或一个字符串，则可以使用 `reduce()`。这样可以为“carry”定义初始值，以及一个接受当前状态“carry”并将每个项目作为参数的闭包，如下所示。

```
$users = collect([...]);

$points = $users->reduce(function ($carry, $user) {
    return $carry + $user['points']
}, 0); // 从 carry 为 0 开始
```

pluck()

如果想在集合的每个项目下提取给定键的值，可以使用 `pluck()`（原 `lists()`），如下所示。

```
$users = collect([...]);

$emails = $users->pluck('email')->toArray();
```

Chunk() 和 take()

`chunk()` 可以轻松将集合分成预定义大小的组，并且 `take()` 仅提取所提供的项目个数，如下所示。

```
$users = collect([...]);

$rowsOfUsers = $users->chunk(3); // 3 个一组

$topThree = $users->take(3); // 提取前 3 个
```

groupBy()

如果要通过其属性之一的值对集合中的所有条目进行分组，可以使用 `groupBy()`，如下所示。

```
$users = collect([...]);

$usersByRole = $users->groupBy('role');

/* Returns:
[
    'member' => [...],
    'admin' => [...]
]
*/
```

也可以传递一个闭包，无论闭包返回什么，都可以用来将记录分组，如下所示。

```
$heroes = collect([...]);

$heroesByAbilityType = $heroes->groupBy(function ($hero) {
    if ($hero->canFly() && $hero->isInvulnerable()) {
        return 'Kryptonian';
    }

    if ($hero->bitByARadioactiveSpider()) {
        return 'Spidermanesque';
    }

    if ($hero->color === 'green' && $hero->likesSmashing()) {
        return 'Hulk-like';
    }

    return 'Generic';
});
```

reverse() 和 shuffle()

reverse() 逆转集合中项目的顺序，而 shuffle() 用于将其顺序随机化，如下所示。

```
$numbers = collect([1, 2, 3]);

$numbers->reverse()->toArray(); // [3, 2, 1]
$numbers->shuffle()->toArray(); // [2, 3, 1]
```

sort()、sortBy() 和 sortByDesc()

如果项目是简单的字符串或整数，那么可以使用 sort() 进行分类，如下所示。

```
$sortedNumbers = collect([1, 7, 6, 4])->sort()->toArray(); // [1, 4, 6, 7]
```

如果项目较为复杂，那么可以传递一个字符串（表示该属性）或闭包给 sortBy() 或 sortByDesc()，用于定义分类行为，如下所示。

```
$users = collect([...]);

// 根据用户的 'email' 属性对用户数组进行排序
$users->sort('email');

// 根据用户的 'email' 属性对用户数组进行排序
$users->sort(function ($user, $key) {
    return $user['email'];
});
```

count() 和 isEmpty()

可以使用 count() 或 isEmpty() 查看集合中有多少项，如下所示。

```
$numbers = collect([1, 2, 3]);

$numbers->count(); // 3
$numbers->isEmpty(); // 不正确
```

avg() 和 sum()

如果正在处理一个数字集合，那么 avg() 和 sum() 会执行它们的方法名称所表示的内容，并且不需要任何参数，如下所示。

```
collect([1, 2, 3])->sum(); // 6
collect([1, 2, 3])->avg(); // 2
```

如果正在使用数组，则可以传递我们希望从数组中获得的属性的键，并进行后续操作，代码如下。

```
$users = collect([...]);  
  
$sumPoints = $users->sum('points');  
$avgPoints = $users->avg('points');
```



使用外部 Laravel 集合

你是否对集合感兴趣，并且想在非 Laravel 项目中使用它们了呢？这里将 Laravel 的集合功能分解成一个名为 Collect (<http://bit.ly/2f1It7n>) 的独立项目，公司的开发人员将其与 Laravel 的版本保持同步。

只需使用 `composer require tightenco/collect` 命令，便可以在代码中使用 `Illuminate\Support\Collection` 类和 `collect()` 助手。

本章小结

Laravel 提供了一整套全局助手功能，可以简化各种任务。使用全局助手可以使操作和检查数组、字符串更简单，也更容易生成路径和 URL，并且提供了访问一致且重要的功能的途径。

Laravel 集合是功能强大的工具，可以为 PHP 提供集合并道的可能性。

词汇表

Accessor（访问器）

Eloquent 模型中定义的方法，可以自定义给定属性的返回方法。通过访问器可以从模型中获取给定的属性，返回与该属性在数据库中存储的值不同（或格式不同）的值。

ActiveRecord

常见的数据库 ORM 模式，同时也是 Laravel 的 Eloquent 使用的模式。在 ActiveRecord 中，同一个模型类定义了如何检索、持久化，以及如何表示数据库记录。此外，每一条数据库记录都是通过应用中的一个实例来表示的，每个实例会映射到数据库中的一条记录上。

Application test（应用测试）

通常被称为验收或功能测试，应用测试通常在外部边界进行，采用类似于 DOM 爬虫的方式来测试应用的整体行为，这也是 Laravel 采用的方法。

参数（Artisan）

参数可以传入 Artisan 的控制台命令。它不是接在 -- 或 = 后面，而是用于接收一个单一的值。

Artisan

支持在命令行与 Laravel 应用程序进行交互的工具。

Assertion（断言）

在测试过程中，断言是最核心的部分：需要预先断言某个值应该等于（小于或大于）另外一个值，或者它应该有一个给定的数值等。断言既可以通过，也可以不通过。

Authentication（认证）

认证是为了识别应用程序的成员 / 用户。认证只是确认身份，而不是授权发生某种行为。

Authorization (授权)

假设已经认证成功或者认证失败了，授权将定义可以发生的行为。授权与访问、控制有关。

Autowiring (自动装配)

在程序员没有明确指出解析类的方法时，依赖注入容器注入了一个解析类的实例，这就是自动装配。如果一个容器不能进行自动装配，那么在明确地将依赖绑定到容器之前，甚至不能注入一个普通的 PHP 对象。有了自动装配之后，只需要将依赖显式绑定到容器即可，如果依赖于过于复杂或依赖关系比较模糊，那么容器将无法自己解决。

beanstalkd

beanstalk 是一个工作队列。它可以运行多个异步任务，这使得它成为 Laravel 队列中常用的驱动器。

Blade

Laravel 的模板引擎。

Carbon

一个 PHP 包，能更容易地表示和处理与日期相关的信息。

Cashier

使用 Stripe 或者 Braintree 计费的 Laravel 包，特别是在处理订阅操作时，其更简单，更一致，也更强大。

Closure (闭包)

闭包是 PHP 版本的匿名方法。闭包是一个函数，也可以把它作为一个对象传递，或者赋值给一个变量，或者作为参数传递给其他函数和方法，甚至还可以将它序列化。

Codelgniter

一个对 Laravel 有启发作用的较老的 PHP 框架。

Collection (集合)

开发模式的名称，也是 Laravel 实现它的工具。与 steroid 上的数组类似，集合提供了 map、reduce、filter 和更多的 PHP 本地数组所不具备的强大功能。

Command (命令)

自定义 Artisan 控制台任务的名称。

Composer

PHP 的依赖管理器，类似 Ruby Gems 或 NPM。

Container (容器)

Laravel 的“容器”，是指负责依赖注入的应用程序容器。可以通过 `app()` 访问，也负责解析对控制器、事件、作业和命令的调用，容器是将每个 Laravel 应用程序放在一起的黏合剂。

Contract

接口的别名。

Controller (控制器)

负责将用户请求指向应用程序的服务和数据的类，并将有效的响应返回给用户。

CSRF (跨站伪造请求)

外部网站往往通过劫持用户浏览器（比如通过 JavaScript）来进行恶意攻击，但是它们依旧需要在网站上进行登录，为网站中的每一个表单添加一个令牌（以及在 POST 端对该令牌进行验证）来进行保护。

Dependency injection (依赖注入)

从外部注入（一般通过构造器），而不是在类中进行初始化。

Directive

Blade 的语法选项，如 @if、@unless 等。

Dot notation (点符号)

显示继承关系。例如有以下数组：
`['owner'=> ['address'=>['line1'=> '123 Main St.']]]`，它有三层嵌套。使用点符号，就可以将“123 Main St.”表示为“owner.address.line1”。

Eager loading (贪婪加载)

通过向一级查询添加一个二级智能查询来获取一组相关项目，从而避免 $N + 1$ 问题。通常情况下，一级查询可以获得集合 A，但是每个 A 都包括多个 B，所以每次从 A 中获取 B 都需要一次新的查询。贪婪加载也是两次查询：第一次获取所有的 A，接着用一次查询获取每个

与 A 关联的所有的 B。在贪婪查询中，只需要两次查询就可以获取所有内容。

Echo

简化 WebSocket 身份验证和数据同步的 Laravel 产品。

Elixir

Laravel 的构建工具，Gulp 周围的包装。

Eloquent

Laravel 的 ActiveRecord ORM。用来定义如用户模型这样的工具。

Environment variable (环境变量)

在 .env 文件中定义的变量，不会包含在版本控制工具中。也就是说它们不会在环境之间同步，而且是安全的。

Envoyer

用于零时差部署的 Laravel 产品。

Event (事件)

Laravel 实现 pub / sub 或 observer 模式的工具。每个事件代表了有事件发生：事件名称描述发生了什么（例如 User Subscribed），有效载荷允许附加相关信息。它被设定为“fired”，然后被设定为“listened”（或者发布和订阅，如果喜欢 pub / sub 的概念）。

facade

Laravel 中简化访问复杂工具的工具。facade 为 Laravel 提供了对于核心服务的静态访问。因为每个 facade 都在容器

中的类中有备份，可以对任何调用进行替换，例如 `Cache::put()`；或者使用两行代码，例如 `¥cache=app('cache'); $cache->put()`。

Flag（标志）

表示开或关的参数（布尔型）。

Fluent

可以连续链接的方法被称为 fluent。为了提供流畅的语法，每个方法都必须返回一个实例，并且能够进行重新链接。这个特性保证了某些功能的实现，如 `People::where('age', '>', 14)->orderBy('name')->get()`。

Flysystem

Laravel 使用此包来促进本地和云文件访问。

Forge

用于启动和管理 DigitalOcean 和 AWS 等主要云提供商提供的虚拟服务器的 Laravel 软件。

FQCN（完整的类名）

任何给定类、特征或接口的完整命名空间名。控制器是类名，`Illuminate \ Routing \ Con troller` 是 FQCN。

Gulp

基于 JavaScript 的开发工具。

Helper（助手）

一个全局可访问的 PHP 函数，使一些其他功能更容易实现。例如 `array_get()`，

简化了从数组查询结果的逻辑。

Homestead

一个包装了 Vagrant 的 Laravel 工具，使得 Forge 并行虚拟服务器可以轻松地在本地 Laravel 开发环境中运行。

Illuminate

所有 Laravel 组件的顶级命名空间。

Integration test（集成测试）

集成测试主要测试各个单元一起工作并传递消息的方式。

IoC（反向控制）

就是给予“控制”，它不是更低级别的代码，而是实现将一个接口的具体实例提供给包的更高级的代码。如果没有 IoC，每个单独的控制器和类可能需要决定要创建哪个 Mailer 实例。IoC 使得低级代码（控制器和类）能够请求一个 Mailer，并且使一些高级配置代码可以在每个应用程序中定义应该提供哪个实例来满足请求。

Job（任务）

封装单个任务的类，为了将任务加入队列并异步运行。

JSON

JavaScript 对象表示法。表示数据的语法。

JWT（JSON 网页令牌）

包含所有决定用户认证状态和访问权限信息的 JSON 对象。该 JSON 对象使用

HMAC 或 RSA 进行数字签名，这使得它较为可信。其通常在头部进行交付。

Mass assignment（批量赋值）

一次传递多个参数，以使用键值数组创建或更新 Eloquent 模型的功能。

Middleware（中间件）

一系列应用程序的包装，用来过滤和装饰程序的输入和输出。

Memcached

内存数据存储，旨在提供简单且快速的数据存储。Memcached 只支持基本的键/值存储。

Migration（迁移）

对数据库状态的操作，存储在代码中并通过代码运行。

Mockery

Laravel 中支持在测试中模拟 PHP 类的库。

Model factory（模型工厂）

如果需要测试或填充，用来定义应用程序如何为模型生成实例的工具。通常与 Faker 这样的虚拟数据生成器同时出现。

Multitenancy（多用户）

支持多个客户端的单个应用程序，每个客户端有自己的用户。多用户通常表明，应用程序的每个客户端都有自己的主题和域名，从而将客户服务与其他客户的潜在服务区分开。

Mutator（修改器）

Eloquent 中的工具，允许在将数据保存到数据库之前对数据进行处理。

Nginx

类似 Apache 的网络服务器。

选项（Artisan）

与参数类似，选项也可以传入 Artisan 命令。它们以 -- 开头，可用作标记（--force）或提供数据（--userId=5）。

ORM（关系对象映射器）

一种以编程语言中的对象为中心的设计模式，用于在关系型数据库中表示数据及其关系。

Passport

一个 Laravel 软件包，可以轻松地将 OAuth 身份验证服务器添加到 Laravel 应用程序中。

PHPSpec

一个 PHP 测试框架。

PHPUnit

一个 PHP 测试框架。最为通用，并且与 Laravel 的测试代码最为相关。

Polymorphic（多态）

数据库中，能够与具有相似特征的多个数据库表进行交互。多态关系允许以相同的方式附加多个模型的实体。

Preprocessor（预处理器）

一种构建工具，采用特殊的语言形式（对于 CSS，即 LESS），并使用普通语言（CSS）生成代码。预处理器构建了不属于核心语言的工具和功能。

Primary key（主键）

大多数数据库表都会用单独的一列来分别代表一整行。也就是常说的主键，通常叫作 id。

Queue（队列）

用于添加任务的栈。通常与队列 worker 相关联，队列 worker 每次从队列中抽取一个任务，对其进行处理，然后删除。

Redis

与 Memcached 类似，Redis 是一个数据存储比大多数关系型数据库简单，但功能强大、速度更快的数据库。Redis 支持的结构和数据类型有限，但在速度和可扩展性上弥补了这一缺陷。

REST

Representational State Transfer（具象状态传输）是现在最常见的 API 格式。通常认为与 API 的交互应该分别进行认证，并且应该是“无状态的”，通常建议使用 HTTP 动词来区分请求的基本差异。

Route（路由）

定义了用户访问网络应用的方式。+ 一个路由就是一个模式定义。它可以表示为如下形式：`/users/5`、`/users` 或 `users/{id}`。

SaaS

软件即服务。需要付费使用的网络应用。

Scope（作用域）

在 Eloquent 中，定义如何连续而且简单地缩小查询范围的工具。

Scout

Laravel 软件包，用于在 Eloquent 模型上进行全文搜索。

Serialization（序列化）

将复杂数据（通常是一个 Eloquent 模型）简化（在 Laravel 中，一般简化为数组或 JSON）的过程。

Service provider（服务提供者）

Laravel 中注册和引导类和容器绑定的结构。

Soft delete（软删除）

将数据库中的某行标记为“已删除”，但并没有真正删除其内容。ORM 通常会默认隐藏所有的“已删除”行。

Spark

一个 Laravel 工具，可以轻松启动一个新的基于订阅的 SaaS 应用程序。

Symfony

一个 PHP 框架，专注于构建优秀的组件并使其他用户可以访问。Symfony 的 HTTP 基础是 Laravel 和其他所有现代 PHP 框架的核心。

Tinker

Laravel 的 REPL 或“读 - 评估 - 打印”循环。这是一个工具，允许在命令行应用程序的完整上下文中执行复杂的 PHP 操作。

TL;DR

本章小结。

Typehint

在类或接口名称的方法签名中预先给定变量名称。告诉 PHP（以及 Laravel 和其他开发人员），允许在该参数中传递的唯一内容是具有给定类或接口的对象。

Unit test（单元测试）

单元测试通常针对小的、相对独立的单元——一个类或一个方法。

Vagrant

一个命令行工具，可以使用预定义的图片轻松地在本地上构建虚拟机。

Valet

Laravel 软件包（主要适用于 Mac OS 用户，也有 Linux 和 Windows 版本），使用户可以轻松地从选择的开发文件夹中为应用程序提供服务，无须担心 Vagrant 或虚拟机。

Validation（验证）

确保用户输入符合预期模式。

View composer（视图组件）

用于定义“每当一个给定的视图被加载，它将提供一个特定的数据集”的工具。

View（视图）

定义要发送给终端用户的 HTML 模板文件，通常包括接收来自控制器的数据，并将其格式化为 HTML 的一部分。

关于作者

Matt Stauffer, 既是一名开发者, 也是一位教师。他是 Tighten Co. 的合伙人兼技术总监, 主办过 The Five-Minute Geek Show 和 Laravel Podcast。他的博客地址是 <http://mattstauffer.co/>。

封面介绍

《Laravel 入门与实战：构建主流 PHP 应用开发框架》封面上的动物是一只羚羊（南非剑羚）。这种大型的羚羊原产于南非、博茨瓦纳、津巴布韦和纳米比亚的沙漠，它是南非的标志。

剑羚的肩高约 5 英尺 7 英寸，体重可达 250 至 390 英磅。它们通常呈淡灰或棕色，面部黑白相间，有又长又黑的尾巴。身上有一道从下巴延伸到颈部下边缘的黑色条纹。剑羚最让人印象深刻的一点是它用于防御的角，平均长度为 33 英寸，在许多文化中作为魅力的象征。在中世纪的英格兰，剑羚的角经常被当做独角兽的角来售卖。

虽然剑羚的角让它备受欢迎，但是它在南非的数量一直很稳定。1969 年，新墨西哥州的南部地区引进了剑羚，目前数量大约有 3000 只。

剑羚非常适合在沙漠地区生存，因为一年中的大多数时间，它都不需要喝水。它不会喘气或出汗：天气炎热的时候，它的体温会比平常高出几度。剑羚一般可以在野外生活 18 年。

大多数 O'Reilly 封面的许多动物都是濒临灭绝的，这些动物对世界来说很重要。了解更多关于如何帮助濒危动物的信息，请访问 animals.oreilly.com。

封面图片来源于 *Riverside Natural History*。