通过学习Srping MVC框架达到设计真实Web应用的专业水准

精通Spring MVC 4

Mastering Spring MVC 4

[美] Geoffroy Warin 著
张卫滨 孙丽文 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

精通Spring MVC 4

[美] Geoffroy Warin 著
张卫滨 孙丽文 译

人 民 邮 电 出 版 社

图书在版编目（CIP）数据

精通Spring MVC 4 / (美) 瓦林 (Geoffroy Warin) 著 ; 张卫滨, 孙丽文译. -- 北京 : 人民邮电出版社, 2017.5
ISBN 978-7-115-44758-6

Ⅰ. ①精… Ⅱ. ①瓦… ②张… ③孙… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第038777号

◆ 著 [美] Geoffroy Warin
译 张卫滨 孙丽文
责任编辑 陈冀康
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：18
字数：350 千字 2017 年 5 月第 1 版
印数：1 – 3 000 册 2017 年 5 月北京第 1 次印刷

著作权合同登记号 图字：01-2016-7045 号

定价：59.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

内容提要

Spring MVC 属于 Spring Framework 的衍生产品，已经融合在 Spring Web Flow 里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。Spring MVC 4 是当前最新的版本，在众多特性上有了进一步的提升。

本书中从头开始构建了一个完整的 Web 应用。全书共 10 章，分别介绍了快速搭建 Spring Web 应用、精通 MVC 结构、处理表单和复杂的 URL 映射、文件上传与错误处理、创建 RESTful 应用、保护应用、单元测试与验收测试、优化请求、将 Web 应用部署到云等内容，循序渐进地讲解了 Spring MVC 4 的开发技巧。

本书最适合已经熟悉 Spring 编程基础知识并迫切希望扩展其 Web 技能的开发人员。通过阅读本书，读者将深度把握 Spring MVC 的各项特性及实用技巧。

译者序

Spring MVC 是 Spring 社区项目的重要组成部分，目前用到了无数规模各异的项目之中。Spring 以简化企业级应用开发为己任，十多年来的发展历史证明，他们确实做到了这一点。不论是 Web 应用开发、数据库访问还是应用集成、大数据处理，我们都能看到 Spring 的相关项目，借助 IoC 与 AOP 的核心理念，Spring 让开发人员的工作更加轻松愉悦。

Spring MVC 本身的发展也是如此，从最初的 XML 文件配置，到后来的注解和 Java 代码方式的配置，都是尽可能地让开发过程更加便利，同时，Spring MVC 还在不断完善其测试功能，推动优秀编码实践能够更容易地落地。

Spring Boot 项目的横空出世更是引起了空前的关注，这是一个改变游戏规则的项目，它能够极大地简化配置，高效地管理依赖，并且与当下流行的微服务架构模式契合良好，得到了广泛地应用。本书组合使用了 Spring Boot 和 Spring MVC，从项目的搭建过程入手，涉及页面开发、文件上传、应用安全、RESTful API 开发、测试等，涵盖了 Web 应用开发的方方面面，最后还介绍了如何将应用部署到云端，不管你关心哪个领域，希望通过本书都能获取到有用的知识。

从《Spring 实战》（第三版）开始，我参与翻译了多本 Spring 相关的书籍，结识了许多志同道合的好友，这也是一种特殊的缘分。技术的发展日新月异，作为从业者的我们自然不能停下前进的脚步，唯有不断学习、不断思考、不断实践，才能保证不被时代所淘汰，希望与诸位读者共勉。

读者在阅读本书的过程中，如果发现错误或问题，请不吝指正，您可以通过 levinzhang1981@126.com 或微博@张卫滨 1895 联系到我。

希望本书能够为您的工作和学习带来帮助。

张卫滨

2017 年 2 月于大连

前言

作为 Web 开发人员，我愿意创建新的东西，将它们快速上线，然后将注意力转移到下一个新的想法上。

如今，所有的应用都互相连接在了一起，我们需要与社交媒体进行交互，促进产品和复杂系统的发展，为用户提供更大的价值。

直到现在，这些对于 Java 开发人员来说都非常遥远和复杂。随着 Spring Boot 的诞生和云平台的平民化，我们可以在有限的时间内创建精彩的应用并让所有的人来访问，而这个过程不需要花一分钱。

在本书中，我们将会从头开始构建一个有用的 Web 应用。这个应用有很多很棒的特性，如国际化、表单校验、分布式会话与缓存、社交登录、多线程编程等。

同时，我们还会对其进行完整的测试。

在本书结束之前，我们还会将这个小应用部署到云端，使它能够通过 Web 进行访问。

如果你觉得这挺有意思的话，那么就别浪费时间了，马上开始着手编码吧！

本书所涵盖的内容

第 1 章“快速搭建 Spring Web 应用”能够让我们非常快速地开始使用 Spring Boot。本章介绍了让我们更具生产效益的工具，如 Spring Tool Suite 和 Git，本章还会帮助我们搭建应用的主体框架，并见识 Spring Boot 背后的魔力。

第 2 章“精通 MVC 架构”指导我们创建一个小的 Twitter 搜索引擎，同时，本章还涵

盖了 Spring MVC 和 Web 架构的基础知识。

第 3 章“处理表单和复杂的 URL 映射”帮助你理解如何创建用户基本信息表单，本章介绍如何在服务端和客户端校验数据，并且让我们的应用支持多语言访问。

第 4 章“文件上传与错误处理”将会指导你为基本信息表单添加文件上传功能，它阐述了如何在 Spring MVC 中恰当地处理错误并展示自定义的错误页面。

第 5 章“创建 RESTful 应用”阐述了 RESTful 架构的理念，它还帮助我们创建了一个可以通过 HTTP 调用的用户管理 API，这个过程中会看到帮助我们设计 API 的工具，并且会讨论如何很简便地实现文档化。

第 6 章“保护应用”将会指导我们如何保护应用，包括如何使用基本 HTTP 认证保护 RESTful API，以及如何保护登录页之后的 Web 页面，它阐述了如何通过 Twitter 进行登录以及如何将会话保存在 Redis 中，从而允许我们的应用进行扩展。

第 7 章“单元测试与验收测试”帮助我们对应用进行测试。它讨论了测试与 TDD，介绍了如何对控制器进行单元测试，如何使用现代的库设计端到端的测试。最后，介绍了 Groovy 如何提升测试的生产效率和可读性。

第 8 章“优化请求”对应用进行了优化。它包括缓存控制和 Gzip，本章将教会我们如何把 Twitter 搜索结果缓存到内存和 Redis 中，以及如何对搜索实现多线程执行。除此之外，还会介绍如何实现 Etag 和使用 WebSocket。

第 9 章“将 Web 应用部署到云中”会指导我们对应用进行部署，通过对比，阐述了不同 PaaS 解决方案的差异。然后，介绍了如何将应用部署到 Cloud Foundry 和 Heroku 中。

第 10 章“超越 Spring Web”在整体上讨论了 Spring 生态系统，介绍了现代 Web 应用的组成部分以及后续的发展方向。

阅读本书所需的前提条件

尽管我们将要构建的是一个很高级的应用，但是并不需要你安装很多的东西。

我们将要构建的应用需要 Java 8。

我们并不强制你使用 Git，不过你绝对应该使用 Git 来对自己的应用进行版本控制。如果你希望将应用部署到 Heroku 上，那么会需要用到它。另外，借助 Git 可以非常容易地回顾你的工作，通过查看代码的差异和历史来了解其演进过程。在第 1 章中包含了很多开始使用 Git 的资源。

我还推荐你使用一个好的 IDE。我们会看到如何使用 Spring Tool Suite（免费）和 IntelliJ Idea（一个月的免费试用）实现快速起步。

如果你使用 Mac 的话，应该了解一下 Homebrew（http://brew.sh）。通过使用这个包管理器，你可以安装本书中提到的所有工具。

本书为谁而作

本书最适合已经熟悉 Spring 编程基础知识并迫切希望扩展其 Web 技能的开发人员。建议你事先掌握一些 Spring 框架的知识。

约定

在本书中，你会看到多种样式的文本，它们用来区分不同类型的信息。如下是这些风格的样例以及对它们的描述。

文本中的代码、数据库表名、文件夹的名称、文件名、文件扩展名、路径名称、伪 URL（dummy URL）、用户输入以及 Twitter 如下所示。

“你会在目录 build/libs 中找到 JAR 文件”。

代码片段如下所示：

```
public class ProfileForm {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();

    // getters and setters
}
```

如果希望你关注特定的代码片段，那么相关的行或条目会粗体显示：

```
public class ProfileForm {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();
```

```
    // getters and setters
}
```

命令行的输入和输出会按照如下所示进行编写：

```
$ curl https://start.spring.io
```

新术语和**关键字**会粗体显示。在屏幕、菜单或对话框中看到的内容将会按照如下的样例显示：“转到新建项目菜单并选择 **Spring Initializr** 项目类型”。

警告或关键的提示按照这种方式显示。

小技巧或窍门按照这种方式显示。

读者反馈

我们欢迎读者的反馈。请告诉我们你觉得这本书怎么样——你喜欢什么，不喜欢什么。读者的反馈是非常重要的，这将有助于我们开发出读者得到真正想要的图书。

如果是一般的反馈，只需发送邮件到 feedback@packtpub.com 即可，并在邮件的主题中包含本书的名称。

如果对于某个话题，你有专业的知识，并且愿意编写一本书或者为某本书做出贡献，那么可以参考我们的读者指南：www.packtpub.com/authors。

客户支持

如果你购买了 Packt 图书，那么我们会从很多方面为你提供帮助，尽可能最大化你的购买所带来的价值。

示例代码下载

你可以通过账号，在 http://www.packtpub.com 下载所有你所购买的 Packt 图书的示例代码文件。如果你是在其他途径购买的本书，那么可以访问 http://www.packtpub.com/support

并进行注册，这些文件会直接邮件发送给你。

对于本书的样例代码，你也可以通过 https://github.com/Mastering-Spring-MVC-4/mastering-spring-mvc4 进行下载。

答疑

如果你对本书有任何的问题，可以通过 questions@packtpub.com 联系我们，我们将尽全力为你解答。

作者简介

Geoffroy Warin 从 10 岁就开始编程了，是软件匠艺（Software Craftsmanship）运动的坚定信奉者和开源的倡导者，他跟随自己的内心选择成为一名开发人员并对其坚定不移。在职业生涯中，他一直致力于使用 Java 和 JavaScript 语言开发企业级的 Web 应用。

Geoffroy 在前端和后端方面都游刃有余，他非常关注代码整洁和可测试性。他深信开发人员应该尽其所能编写出可读性更强的代码，这样的代码能够持续地为用户交付价值。

在推动测试驱动开发和创建伟大的软件设计方面，他的核心工具就是结对编程和导师制。

他还讲授 Java Web 技术栈的课程，是 Groovy 和 Spring 的狂热支持者。

最近，他还担任《Learning Spring Boot》和《Spring Boot Cookbook》等书的审阅者，这两本书都是由 Packt 出版社所出版，涵盖了 Spring 生态系统中新增的主要内容。

如果想学习最新的 Spring 和 JavaScript 编程技巧，请查阅 Geoffroy 的博客（http://geowarin.github.io）和他的 Twitter 账号（https://twitter.com/geowarin）。

技术审校者简介

Raymundo Armendariz 是有超过 10 年经验的软件开发人员，他之前主要致力于为 Java 和.NET 平台构建软件，但是现在他投身于 JavaScript。

他还是一本 JavaScript 微框架图书的作者。

在他的职业生涯中，大部分的工作都是与汽车行业相关，他曾经就职的公司包括 Autozone、Alldata、TRW 和 1A Auto。

他是《Getting Started with Backbone Marionette》（Packt 出版社）一书的作者，这本书可以在 https://www.packtpub.com/web-development/getting-started-backbone-marionette 找到。

我要感谢朋友们对我的支持和帮助。

——Raymundo

Abu Kamruzzaman 是纽约城市大学的 Web 程序员和数据库分析师。在过去的 10 多年以来，他所开发和维护的 Web 应用作为班级教学和注册课程，用于高校的教学。从 2014 年 11 月开始，他担任 CUNY 总部的 PeopleSoft 开发专家，目前，他所从事的项目是与商业智能团队（Business Intelligence）一起使用 OBIEE 为 CUNY 构建数据仓库。在加入总部之前，自 2001 年以来，他曾经在 CUNY 的各个校区工作过，还讲授研究生和本科生的 IT 课程，指导学生们使用他所开发的应用。从 2001 年开始，他所讲授的课程包括 J2EE、DBMS、数据仓库、面向对象编程、Web 设计以及 Web 编程。他是柏鲁克学院（Baruch College）杰克林商学院（Zicklin School of Business）计算机信息系统系的教员之一。他非常热心于教育事业，对开源项目也有着极大的热情，如 Hadoop、Hive、Pig、NoSQL 数据库、Java、云

计算以及移动应用开发。他从布鲁克林学院（Brooklyn College）/CUNY 获得了硕士学位，从宾厄姆顿大学（Binghamton University）/SUNY 计算机科学专业获得了学士学位。他的 Web 站点地址：http://faculty.baruch.cuny.edu/akamruzzaman/。

我要感谢美丽的妻子 Nicole Woods，感谢她对我所从事事业的耐心、支持以及鼓励。感谢我的父母，感谢他们的祝福和祈祷。感谢本书的作者和 Packt 出版团队能够给我参与本书的机会。

——Abu

Jean-Pol Landrain 是软件工程专业的学士，从 1998 年开始就主要从事面向网络、实时和分布式计算的工作。他逐渐成为一名软件架构师，具有超过 17 年的面向对象编程经验，尤其擅长 C++、Java/JEE、各种应用服务器、操作系统（Windows 和 Linux）以及相关的技术。

他目前就职于 Agile Partner，这是一家位于卢森堡的 IT 咨询公司，从 2006 年以来一直致力于敏捷方法论的推广、教育和应用。在过去的 5 年中，他参与了针对欧洲议会（European Parliament）开发团队的工具与技术方案的选择与验证。

他与 Packt 出版社协作，审校了《HornetQ Messaging Developer's Guide》，与 Manning 出版社合作，审校了《Docker in Action》《Git in Practice》《ActiveMq in Action》以及《Spring in Action》（第一版）。

首先，我要感谢我的妻子 Marie Smets 以及 9 岁的女儿 Phoebe，感谢他们理解我对技术的热爱以及在这上面所花费的时间。我还要感谢在 Agile Partner 的朋友和同事，如果一个人单枪匹马地投身于技术会非常乏味的，感谢他们所增添的乐趣。

很不幸的是，在这本书制作期间，我失去了我的祖父 André Landrain 和祖母 Hélène Guffens，我非常悲痛。因为这些私人的事件导致了一些延迟，非常感谢 Packt 出版社编辑团队的耐心，尤其要感谢该项目的协调者 Nidhi Joshi 和本书的作者 GeoffroyWarin。他们做得非常棒，我完全信任本书的品质。在 Spring MVC 方面，这是市面上很好的一本书。

——Jean-Pol

Wayne Lund 是 Pivotal 的 PaaS 和现场工程师。在企业级软件开发和分布式环境方面具有超过 25 年的经验，主要的方向是 Spring、企业级 Java、Groovy 和 Grails，并将这些技术扩展至使用 Smalltalk 和 C++的系统，他非常热衷于个性化和新兴的技术。他的目标是继续享受下一代的技术，这就是 PaaS 以及新的 Spring 工具集，这包括构建在 Spring Boot、Spring

Cloud 和 Spring XD 技术上的原生云应用，这样的技术方案能够启用快数据（Fast Data）、大数据、社交以及移动等特性。

目前，他就职于 Pivotal，关注云、数据以及敏捷的结合，之前曾经在一家财富 500 强的医疗保健公司和一家全球性的大型咨询公司工作多年。

他还参与出版了 Packt 出版社的《Learning Spring Application Development》一书。

目录

第 1 章
快速搭建 Spring Web 应用

在本章中，我们将会直接接触代码并搭建一个 Web 应用，本书的其他章节将会基于该应用进行讲解。

在这里，我们将会使用 Spring Boot 的自动配置功能来构建应用，这样的话，就能完全避免使用样板式的配置文件。

本书将会从整体上介绍 Spring Boot 是如何运行的以及该如何对其进行配置，共有 4 种方式来开启一个 Spring 项目：

- 使用 Spring Tool Suite 生成 Starter 代码；
- 使用 IntelliJ IDEA 14.1，它对 Spring Boot 提供了良好的支持；
- 借助 Spring 站点，从 http://start.Spring.io 上下载可配置的 ZIP 文件；
- 使用到 http://start.Spring.io 站点的 curl 命令来达到相同的效果。

本书中将会使用 Gradle 和 Java 8，但是也不必为此感到担心。如果你还在使用 Maven 和更早版本的 Java 的话，相信你会发现这些技术也是很易于使用的。

很多官方的 Spring 教程同时提供了 Gradle 构建和 Maven 构建，因此，如果你决定继续使用 Maven 的话，也能很容易地找到样例。Spring 4 完全兼容 Java 8，如果你不采用 Lambda 表达式来简化代码库的话，那真的是很遗憾的事情。

本书同时还会为你展示一些 Git 命令。笔者认为，跟踪工作进展并在稳定的状态进行提交是一件好事。另外，这样还能很容易地将你的工作成果与本书提供的源码进行对比。

本书第 9 章将借助 Heroku 部署我们的应用，建议从一开始就使用 Git 对代码进行版本

管理。在本章中，关于如何开始使用 Git，我会给出一些建议。

1.1 Spring Tool Suite 简介

如果要开始学习 Spring 并使用 Spring 社区所提供的指南和 Starter 项目的话，那么最好的起步方式之一就是下载 Spring Tool Suite（STS）。STS 是一个自定义版本的 Eclipse，它被用来与各种 Spring 项目进行协作，它同时还包括 Groovy 和 Gradle 功能。即便如此，你可能像我一样，还会使用其他的 IDE，但是，我强烈建议你给 STS 一个机会，因为它通过“Getting Started”项目，能够让你快速地了解 Spring 广阔的生态系统。

所以，你可以访问 https://Spring.io/tools/sts/all，并下载 STS 的最新发布版。在生成第一个 Spring Boot 项目之前，首先需要安装 Gradle 对 STS 的支持。在 Dashboard 中，可以看到“Manage IDE Extensions”按钮，然后，需要在“Language and framework tooling”区域中选择下载“Gradle Support”。

还推荐你下载“Groovy Eclipse”以及“Groovy 2.4 compiler”，如图 1-1 所示，在本书的后文中，介绍使用 geb 构建验收测试时会用到它们。

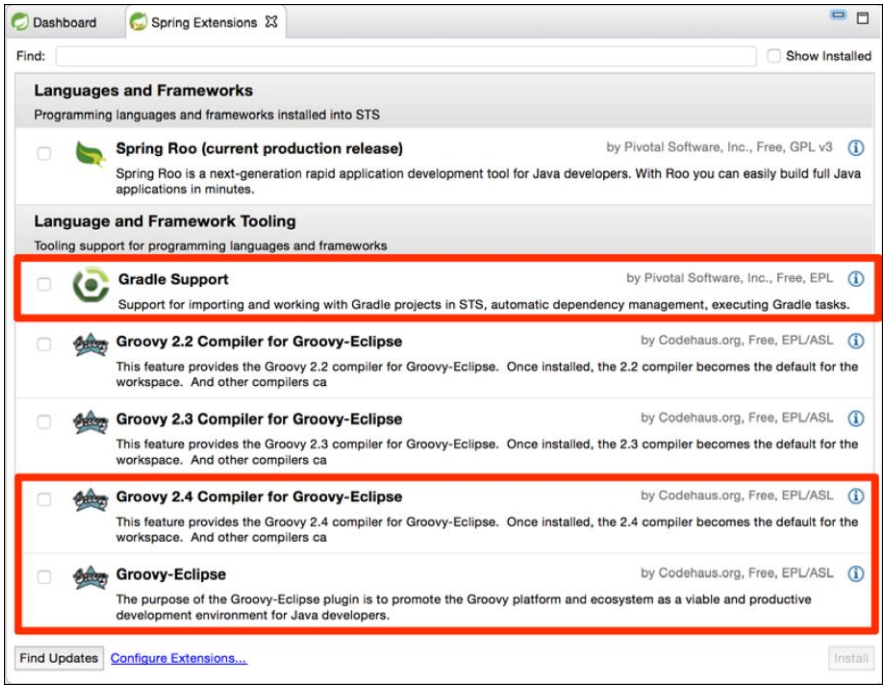

图 1-1

现在，在起步阶段，提供两种可选方案。

下载示例代码

通过你的账号，可以在 http://www.packtpub.com 站点下载购买的所有 Packt 书籍的示例代码文件。如果你通过其他途径购买本书的话，那么可以访问 http://www.packtpub.com/support 并进行注册，这些文件就能通过 Email 直接发送给你了。也可以直接通过 https://github.com/Mastering-Spring-MVC-4/mastering-spring-mvc4 下载本书的示例代码。

第一个方案是使用“File | New | Spring Starter Project”导航菜单，如图 1-2 的截屏所示。这里的可选项是与 http://start.Spring.io 相同的，只不过嵌入到了 IDE 中。

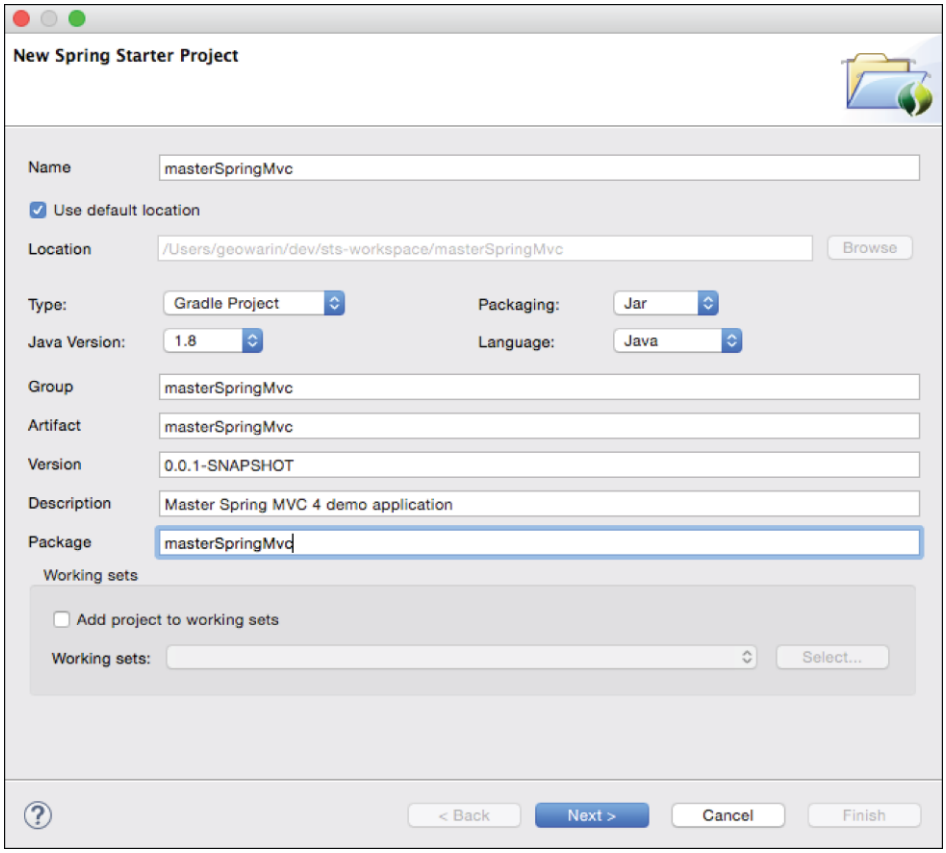

图 1-2

通过使用顶部菜单中的“File | New | Import Getting Started Content”，我们可以看到 http://spring.io 上所有的可配置项，这里可以选择使用 Gradle 或 Maven，如图 1-3 所示。

可以下载 Starter 代码，并按步骤学习本书中的内容，也可以直接下载完整的代码。

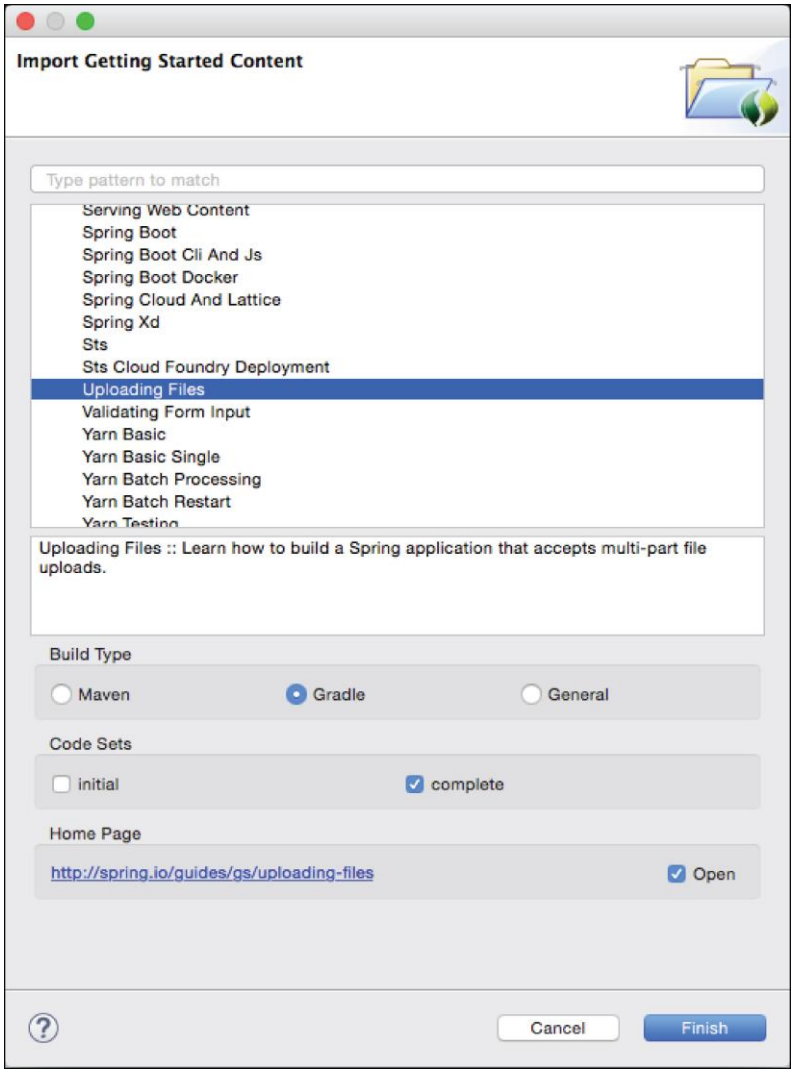

图 1-3

在“Getting Started Content”中有很多有意思的内容，建议读者自行对其进行一下探索。

它阐述了如何将 Spring 与各种读者可能感兴趣的技术进行集成。

此时，将会生成一个 Web 项目，如图 1-3 所示，这是一个 Gradle 应用，会生成 JAR 文件并使用 Java 8。

表 1-1 是我们想要使用的配置。

表 1-1

属性	值
Name	masterSpringMvc
Type	Gradle project
Packaging	Jar
Java version	1.8
Language	Java
Group	masterSpringMvc
Artifact	masterSpringMvc
Version	0.0.1-SNAPSHOT
Description	Be creative!
Package	masterSpringMvc

在第 2 个界面中，将会询问你想要使用的 Spring Boot 版本以及想要添加进工程的依赖。

在编写本书的时候，Spring Boot 的最新版本是 1.2.5，请确保你始终选择最新的版本。

当你阅读本书的时候，可以使用最新的快照版本。如果到那时 Spring Boot1.3 还没有发布的话，那么你可以试一下快照版本。你可以参考 https://spring.io/blog/2015/06/17/devtools-in-spring-boot-1-3 来了解更多细节。

在配置窗口的底部会有一些复选框，代表各种 Spring Boot starter 库。它们是可以添加到构建文件中的依赖项，针对各种 Spring 项目，它们提供了自动配置功能。

现在只关心 Spring MVC，所以只选中 Web 这个复选框。

为 Web 应用生成一个 JAR 文件？将 Web 应用打包为 JAR 文件，这一点你们可能会觉得有些诡异。尽管仍然可以将其打包为 WAR，但这并不是推荐的实践。在默认情况下，Spring Boot 将会创建一个胖 JAR 包（fat JAR），这个 JAR 包中包含了应用所有的依赖，提供了通过“java-jar”命令便捷启动 Web 应用的方法。
我们的应用将会打包为 JAR 文件，如果你想创建 WAR 文件的话，可以参考 http://spring.io/guides/gs/convert-jar-to-war/。

你点击了“**Finish**”按钮了吗？如果已经点击了的话，将会得到如图 1-4 所示的项目结构。

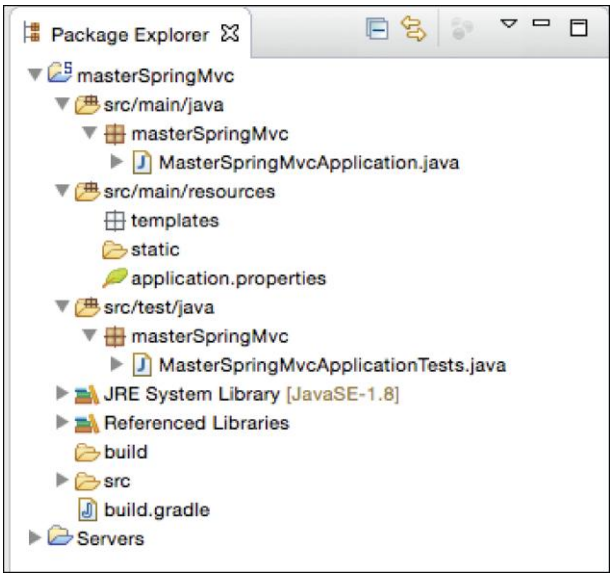

图 1-4

可以看到主类 MasterSpringMvcApplication 及其测试类 MasterSpringMvc ApplicationTests，还可以看到两个空的文件夹——static 和 templates，它们分别用来存放静态 Web 资源（图片、样式文件等）和模板（jsp、freemarker 或 Thymeleaf）。最后一个文件是空的 application.properties，它是 Spring Boot 默认的配置文件。这是一个很便利的文件，在本章中，将会看到 Spring Boot 如何使用它。

对于构建文件 build.gradle，稍后将会详细介绍。

如果你觉得已经准备就绪，那么运行应用的主方法，这样就能会启动一个 Web 服务器。

要做到这一点，切换至应用的主方法，然后右键点击该类，并在工具栏中导航至“Run as | Spring Application”，或者点击工具栏上绿色的 Play 按钮。

遵循上面的步骤，并导航至 http://localhost:8080，此时会产生一个错误，不必担心，请继续往下阅读。

接下来将为读者展示如何不使用 STS 来生成相同的项目，然后再回过头来看这些文件。

1.2　IntelliJ 简介

IntelliJ IDEA 是在 Java 开发人员中非常流行的一个工具。在过去的几年中，因为这个很棒的编辑器，我非常心甘情愿地为 Jetbrains 支付了年费。

IntelliJ 也有快速创建 Spring Boot 项目的方法。

如图 1-5 所示，进入新建项目菜单，默认的择“Spring Initializr”项目。

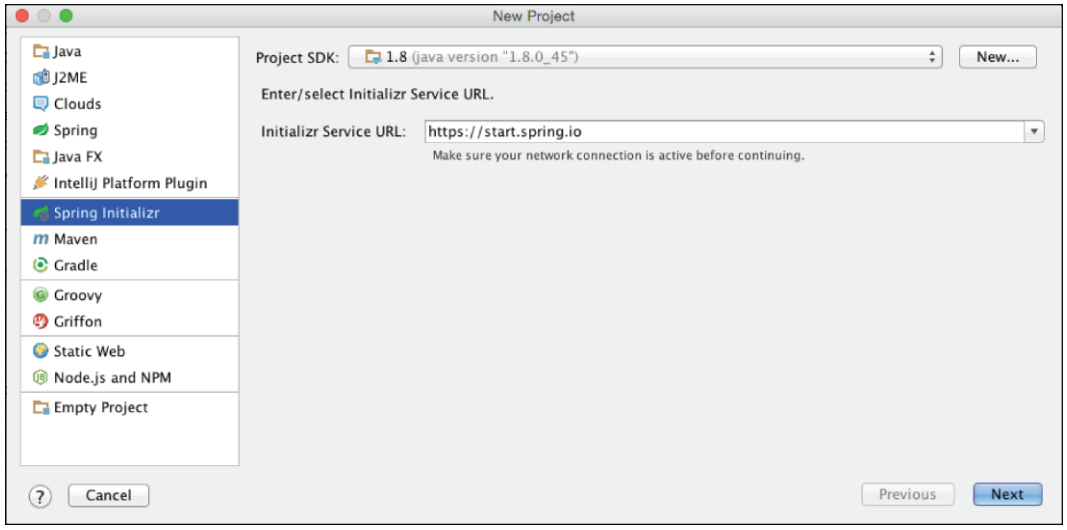

图 1-5

这将会出现与 STS 相同的配置选项，所以请参照之前的小节来了解详细的配置。

你可能需要将 Gradle 项目导入到 IntelliJ 之中。推荐你首先生成 Gradle 包装器（参考 1.5.1 节）。
如果需要的话，通过再次打开项目的 build.gradle 文件，可以重新导入该项目。

1.3 start.Spring.io 简介

请导航至 http://start.Spring.io 站点来开始使用 start.Spring.io，对于这个类似于 Bootstrap 的站点，你可能会感到很熟悉。如果进入上述的链接，那么看到的内容会如图 1-6 的截屏所示。

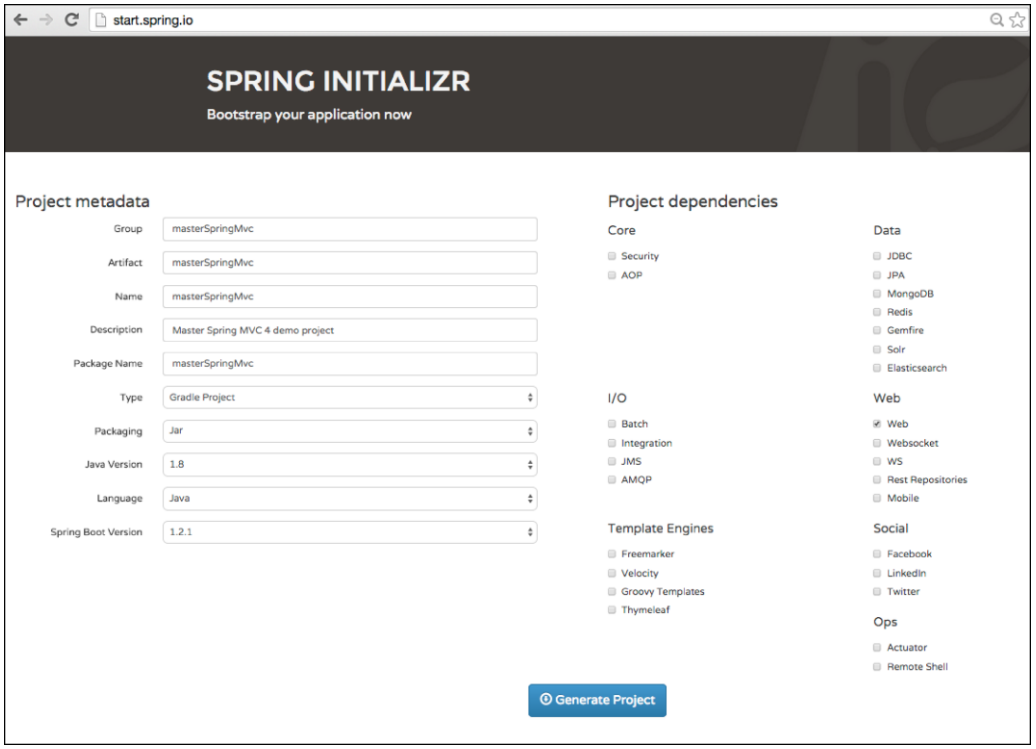

图 1-6

在这里所看到的配置选项与 STS 中是相同的，点击“Generate Project”按钮后将会下载一个 ZIP 文件，这个文件中会包含我们的 Stater 项目。

1.4 命令行方式简介

对于钟情于控制台的读者来说，可以采用“curl http://start.Spring.io”的方式。采用这种方式的话，将会需要一些指令，帮助我们组织 curl 请求。

例如，要生成与之前相同的项目，那么可以输入如下的命令：

```
$ curl http://start.Spring.io/starter.tgz \
-d name=masterSpringMvc \
-d dependencies=web \
-d language=java \
-d JavaVersion=1.8 \
-d type=gradle-project \
-d packageName=masterSpringMvc \
-d packaging=jar \
-d baseDir=app | tar -xzvf -
% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
Dload  Upload   Total   Spent    Left  Speed
100  1255  100 1119  100   136    1014    123  0:00:01  0:00:01 --:--:-
-  1015
x app/
x app/src/
x app/src/main/
x app/src/main/Java/
x app/src/main/Java/com/
x app/src/main/Java/com/geowarin/
x app/src/main/resources/
x app/src/main/resources/static/
x app/src/main/resources/templates/
x app/src/test/
x app/src/test/Java/
x app/src/test/Java/com/
x app/src/test/Java/com/geowarin/
x app/build.Gradle
x app/src/main/Java/com/geowarin/AppApplication.Java
x app/src/main/resources/application.properties
x app/src/test/Java/com/geowarin/AppApplicationTests.Java
```

现在，我们不离开控制台就能开始使用 Spring 了，美梦变成了现实。

可以考虑为上述的命令创建一个别名（alias），这样的话，就能快速地创建 Spring 项目的原型了。

1.5　那就正式开始吧

现在 Web 应用已经准备就绪，先看一下它是如何编写的。在进一步学习之前，我们可

以将工作的成果保存到 Git 上。

如果你还不了解 Git 的话，我推荐下面的两个教程：

- https://try.github.io，这是一个很好的交互式教程，可以引导你一步步地学习基础的 Git 命令；
- http://pcottle.github.io/learnGitBranching，这是一个很棒的教程，它将 Git 以类似于树形的结构进行了可视化，它同时展现了 Git 的基本和高级功能。

安装 Git

在 Windows 下，需要安装 Git bash，这可以在 https://msysgit.github.io 找到。在 Mac 下，如果你使用 homebrew 的话，很可能已经安装过 Git 了，否则的话，使用 brew install git 命令来进行安装。如果有疑问的话，请查阅 https://git-scm.com/book/en/v2/Getting-Started-Installing-Git 上的文档。

如果要使用 Git 版本化我们的工作内容，那么可以在控制台中输入如下的命令：

```
$ cd app
$ git init
```

使用 IntelliJ 的话，要忽略自动生成的文件，即“.idea”和“*.iml”。使用 Eclipse 的话，应该将“.classpath”文件和“.settings”文件夹提交上去。不管是哪种情况，都要忽略“.gradle”文件夹和 build 文件夹。

创建一个包含如下文本内容的“.gitignore“文件：

```
# IntelliJ project files
.idea
*.iml

# gradle
.gradle
build
```

现在，我们可以将其他文件添加到 Git 中：

```
$ git add .
$ git commit -m "Generated with curl start.Spring.io"
```

```
[master (root-commit) eded363] Generated with curl start.Spring.io
4 files changed, 75 insertions(+)
create mode 100644 build.Gradle
create mode 100644 src/main/Java/com/geowarin/AppApplication.Java
create mode 100644 src/main/resources/application.properties
create mode 100644 src/test/Java/com/geowarin/AppApplicationTests.Java
```

1.5.1　Gradle 构建

如果你还不熟悉 Gradle 的话，那么可以将其视为 Maven 的继任者，它是一个现代化的构建工具。与 Maven 类似，它会使用约定，例如如何组织 Java 应用的结构。我们的源码依然会放在“src/main/java”之中，Web 应用的代码放到“src/main/webapp”之中，诸如此类。与 Maven 类似，我们可以使用 Gradle 插件来处理各种构建任务。但是，Gradle 真正的闪光点在于，它允许我们使用 Groovy DSL 编写自己的构建任务。默认库使得管理文件、声明任务之间的依赖以及增量执行 job 都变得非常容易。

安装 Gradle

如果你使用 OS X 的话，那么可以通过 brew install gradle 命令，借助 brew 来安装 Gradle。在任意的*NIX 的系统下（包括 Mac），都可以使用 gvm（http://gvmtool.net/）来进行安装。另外，也可以在 https://Gradle.org/ downloads 下获取二进制分发包。

使用 Gradle 创建应用的第一个最佳实践就是生成 Gradle 包装器（wrapper）。Gradle 包装器是一个小的脚本，它能够在你的代码中进行共享，从而确保会使用相同版本的 Gradle 来构建你的应用。

生成包装器的命令是 gradle wrapper：

```
$ gradle wrapper
:wrapper

BUILD SUCCESSFUL

Total time: 6.699 secs
```

如果我们看一下新创建的文件，可以看到有两个脚本和两个目录：

```
$ git status -s
?? .gradle/
?? gradle/
?? gradlew
?? gradlew.bat
```

在“.gradle”目录中包含了 Gradle 二进制文件，我们不希望将其添加到版本控制之中。

前面已经忽略了这个文件和构建目录，所以可以安全地对其他内容执行 git add 操作：

```
$ git add .
$ git commit -m "Added Gradle wrapper"
```

gradle 目录包含了如何得到二进制文件的信息。另外两个文件是脚本：用于 Windows 的批处理脚本（gradlew.bat）以及用于其他系统的 shell 脚本。

我们可以使用 Gradle 运行应用，替换借助 IDE 来执行应用的方式：

```
$ ./gradlew bootrun
```

执行上面的命令将会运行一个嵌入式的 Tomcat，应用会位于它里面！

如图 1-7 所示，日志提示服务器运行在 8080 端口上，我们检查一下。

图 1-7

可以想象到你内心的失望，因为应用还没有为完全公开做好准备。

换句话说，在工程中，这两个文件所完成的工作内容还是很让人振奋的。我们来看一下。

首先是 Gradle 构建文件，也就是 build.gradle：

```
buildscript {
    ext {
```

```
        springBootVersion = '1.2.5.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
        classpath("io.spring.gradle:dependency-management-
plugin:0.5.1.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'
apply plugin: 'io.spring.dependency-management'

jar {
    baseName = 'masterSpringMvc'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}


dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile("org.springframework.boot:spring-boot-starter-test")
}
eclipse {
    classpath {
         containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
         containers 'org.eclipse.jdt.launching.JRE_CONTAINER/org.
eclipse.jdt.internal.debug.ui.launcher.StandardVMType/JavaSE-1.8'
    }
}
```

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.3'
}
```

在这里，都看到了什么呢？

- 对 Spring Boot 插件的依赖，它分布在 Maven 中央仓库中。
- 我们的项目是 Java 工程。Gradle 可以为 IntelliJ 或 Eclipse 生成 IDE 工程文件。
- 该应用将会生成 JAR 文件。
- 我们的项目的依赖是托管在 Maven 中央仓库中的。
- 类路径在生产环境下包含 spring-boot-starter-web，在测试环境中，还包含 spring-boot-starter-test。
- 针对 Eclipse 的一些额外配置。
- Gradle 包装器的版本是 2.3。

Spring Boot 插件将会产生一个胖（fat）JAR 文件，其中包含了项目的所有依赖。要构建的话，只需输入：

```
./gradlew build
```

我们将会在“build/libs”目录下找到这个 JAR 文件。该目录下会包含两个文件，其中一个为胖 JAR 包，名为 masterSpringMvc-0.0.1-SNAPSHOT.jar，另外一个是普通的 JAR 文件，名为 masterSpringMvc-0.0.1-SNAPSHOT. jar.original，这个文件不包含任何的依赖。

可运行的 JAR

Spring Boot 主要的一个优势在于将应用所需的所有内容都放到一个易于重发布的 JAR 文件中，其中包含了 Web 服务器。如果你运行 java jar masterSpringMvc-0.0.1-SNAPSHOT.jar 的话，Tomcat 将会在 8080 端口上启动，就像在开发期一样。如果要将其部署到生产环境或云中，这都是相当便利的。

在这里，主要的依赖是 spring-boot-starter-web，Spring Boot 提供了很多的 Starter，它们会对应用的很多方面进行自动化配置，这是通过提供典型的依赖和 Spring 配置来实现的。

例如，spring-boot-starter-web 将会包含对 tomcat-embedded 和 Spring MVC 的依赖。它会自动运行 Spring MVC 最为常用的配置并提供一个分发器（dispatcher），使其监听“/”根路径，还会提供错误处理页面，就像之前所看到的 404 页面那样。除此之外，还有一个典型的视图解析器（view resolver）配置。

稍后，我们将会看到更多的内容，首先从下一节开始吧！

1.5.2　让我们看一下代码

这里将会展现运行应用的所有代码，它是一个经典的 main 函数，这种方式有很大的优势，因为我们可以在 IDE 中像运行其他程序那样运行这个应用。我们可以对其进行调试，并且不需要插件就能实现一些类的重新加载。

在开发模式下，当我们在 Eclipse 中保存文件或者在 IntelliJ 中点击“Make Project”就会触发重新加载的过程。只有 JVM 支持切换至新编译版本的类文件时，它才是可行的，如果修改静态变量或配置文件的话，我们必须要重新加载应用。

主类如下所示：

```
package masterSpringMvc;

import org.Springframework.boot.SpringApplication;
import org.Springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AppApplication {

    public static void main(String[] args) {
        SpringApplication.run(AppApplication.class, args);
    }
}
```

需要注意的是@SpringBootApplication 注解，如果看一下这个注解的代码的话，就会发现它实际上组合了 3 个其他的注解，也就是@Configuration、@EnableAutoConfiguration 和@ComponentScan：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
```

```
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {

  /**
  * Exclude specific auto-configuration classes such that they will
never be applied.
  */
  Class<?>[] exclude() default {};
}
```

如果你之前使用 Java 代码配置过 Spring 应用的话，那么你应该对@Configuration 非常熟悉。它表明我们的这个类将会处理 Spring 的常规配置，如 bean 的声明。

@ComponentScan 也是一个比较经典的注解，它会告诉 Spring 去哪里查找 Spring 组件（服务、控制器等）。在默认情况下，这个注解将会扫描当前包以及该包下面的所有子包。

在这里，比较新颖的是@EnableAutoConfiguration 注解，它会指导 Spring Boot 发挥其魔力。如果你将其移除掉的话，就无法从 Spring Boot 的自动配置中收益了。

使用 Spring Boot 来编写 MVC 应用的第一步通常是在代码中添加控制器。将控制器放到 controller 子包中，这样它就能够被@ComponentScan 注解所发现：

```
package masterSpringMvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HelloController {

    @RequestMapping("/")
    @ResponseBody
    public String hello() {
        return "Hello, world!";
    }
}
```

现在，如果打开浏览器并访问 http://localhost:8080 的话，就能看到我们钟爱的“Hello, world!”被输出了出来（见图 1-8）。

图 1-8

1.6　幕后的 Spring Boot

如果你之前搭建过 Spring MVC 应用，那么可能已经习惯于编写相关的 XML 文件或 Java 注解配置类。

一般来讲，初始的步骤如下所示：

1．初始化 Spring MVC 的 DispatcherServlet；

2．搭建转码过滤器，保证客户端请求进行正确地转码；

3．搭建视图解析器（view resolver），告诉 Spring 去哪里查找视图，以及它们是使用哪种方言编写的（JSP、Thymeleaf 模板等）；

4．配置静态资源的位置（CSS、JS）；

5．配置所支持的地域以及资源 bundle；

6．配置 multipart 解析器，保证文件上传能够正常工作；

7．将 Tomcat 或 Jetty 包含进来，从而能够在 Web 服务器上运行我们的应用；

8．建立错误页面（如 404）。

不过，Spring Boot 为我们处理了所有的事情。因为这些配置一般是与应用相关的，所以你可以无限制地将它们进行组合。

在一定程度上来讲，Spring Boot 是带有一定倾向性的 Spring 项目配置器。它基于约定，并且默认会在你的项目中使用这些约定。

1.6.1　分发器和 multipart 配置

接下来，让我们看一下在幕后到底发生了什么。

我们使用默认生成的 Spring Boot 配置文件，并将其设置为 debug 模式。在 src/main/resources/application.properties 中添加下面这一行：

```
debug=true
```

现在，如果重新启动应用的话，就能看到 Spring Boot 的自动配置报告。它分为两部分：一部分是匹配上的（positive matches），列出了应用中，所有的自动配置，另一部分是没有匹配上的（negative matches），这部分是应用在启动的时候，需求没有满足的 Spring Boot 自动配置：

```
=========================
AUTO-CONFIGURATION REPORT
=========================



Positive matches:
-----------------


  DispatcherServletAutoConfiguration
      - @ConditionalOnClass classes found: org.Springframework.web.
servlet.DispatcherServlet (OnClassCondition)
      - found web application StandardServletEnvironment
(OnWebApplicationCondition)


  EmbeddedServletContainerAutoConfiguration
      - found web application StandardServletEnvironment
(OnWebApplicationCondition)


  ErrorMvcAutoConfiguration
      - @ConditionalOnClass classes found: javax.servlet.Servlet,org.
springframework.web.servlet.DispatcherServlet (OnClassCondition)
      - found web application StandardServletEnvironment
(OnWebApplicationCondition)


  HttpEncodingAutoConfiguration
      - @ConditionalOnClass classes found: org.springframework.web.
filter.CharacterEncodingFilter (OnClassCondition)
      - matched (OnPropertyCondition)



<Input trimmed>
```

仔细看一下 DispatcherServletAutoConfiguration：

```
/**
* {@link EnableAutoConfiguration Auto-configuration} for the Spring
* {@link DispatcherServlet}. Should work for a standalone application
where an embedded
* servlet container is already present and also for a deployable
application using
* {@link SpringBootServletInitializer}.
*
* @author Phillip Webb
* @author Dave Syer
*/
@Order(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(EmbeddedServletContainerAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {

    /*
    * The bean name for a DispatcherServlet that will be mapped to the
root URL "/"
    */
    public static final String DEFAULT_DISPATCHER_SERVLET_BEAN_NAME =
"dispatcherServlet";

    /*
    * The bean name for a ServletRegistrationBean for the
DispatcherServlet "/"
    */
    public static final String DEFAULT_DISPATCHER_SERVLET_
REGISTRATION_BEAN_NAME = "dispatcherServletRegistration";
    @Configuration
    @Conditional(DefaultDispatcherServletCondition.class)
    @ConditionalOnClass(ServletRegistration.class)
    protected static class DispatcherServletConfiguration {

        @Autowired
        private ServerProperties server;

        @Autowired(required = false)
        private MultipartConfigElement multipartConfig;
```

```
        @Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
        public DispatcherServlet dispatcherServlet() {
            return new DispatcherServlet();
        }

        @Bean(name = DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_
NAME)
        public ServletRegistrationBean dispatcherServletRegistration()
{
            ServletRegistrationBean registration = new
ServletRegistrationBean(
                    dispatcherServlet(), this.server.
getServletMapping());
            registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_
NAME);
            if (this.multipartConfig != null) {
                registration.setMultipartConfig(this.multipartConfig);
            }
            return registration;
        }

        @Bean
        @ConditionalOnBean(MultipartResolver.class)
        @ConditionalOnMissingBean(name = DispatcherServlet.MULTIPART_
RESOLVER_BEAN_NAME)
        public MultipartResolver multipartResolver(MultipartResolver
resolver) {
            // Detect if the user has created a MultipartResolver but
named it incorrectly
            return resolver;
        }
    }

    @Order(Ordered.LOWEST_PRECEDENCE - 10)
    private static class DefaultDispatcherServletCondition extends
SpringBootCondition {

        @Override
        public ConditionOutcome getMatchOutcome(ConditionContext
context,
                AnnotatedTypeMetadata metadata) {
            ConfigurableListableBeanFactory beanFactory = context.
```

```
    getBeanFactory();
                ConditionOutcome outcome = checkServlets(beanFactory);
                if (!outcome.isMatch()) {
                    return outcome;
                }
                return checkServletRegistrations(beanFactory);
            }

    }
}
```

这是一个典型的 Spring Boot 配置类。

- 与其他的 Spring 配置类相同，它使用了@Configuration 注解；
- 一般会通过@Order 注解来声明优先等级，可以看到 DispatcherServletAutoConfiguration 需要优先进行配置；
- 其中也可以包含一些提示信息，如@AutoConfigureAfter 或@AutoConfigureBefore，从而进一步细化配置处理的顺序；
- 它还支持在特定的条件下启用某项功能。通过使用@ConditionalOnClass (DispatcherServlet.class)这个特殊的配置，能够确保我们的类路径下包含 DispatcherServlet，这能够很好地表明 Spring MVC 位于类路径中，用户当前希望将其启动起来。

这个文件中还包含了 Spring MVC 分发器 Servlet 和 multipart 解析器的典型配置。整个 Spring MVC 配置被拆分到了多个文件之中。

另外，值得一提的是，这些 bean 会遵循特定的规则，以此来检查是否处于激活状态。在@Conditional(DefaultDispatcherServletCondition.class)条件满足的情况下，ServletRegistrationBean 函数才会启用，这有些复杂，但是能够检查在你的配置中，是否已经注册了分发器 Servlet。

只有在满足@ConditionalOnMissingBean(name=DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)条件的情况下，MultipartResolver 函数才会处于激活状态，例如，当我们自己还没有注册的时候。

这意味着 Spring Boot 仅仅是基于常见的使用场景，帮助我们对应用进行配置。不过，可以在任意的地方覆盖这些默认值，并声明自己的配置。

因此，通过查看 DispatcherServletAutoConfiguration，就了解了为什么我们已经拥有了分发器 Servlet 和 multipart 解析器。

1.6.2 视图解析器、静态资源以及区域配置

另外一个密切相关的配置是 WebMvcAutoConfiguration，它声明了视图解析器、地域解析器（localeresolver）以及静态资源的位置。视图解析器如下所示：

```
@Configuration
@Import(EnableWebMvcConfiguration.class)
@EnableConfigurationProperties({ WebMvcProperties.class,
ResourceProperties.class })
public static class WebMvcAutoConfigurationAdapter extends
WebMvcConfigurerAdapter {

  @Value("${spring.view.prefix:}")
  private String prefix = "";

  @Value("${spring.view.suffix:}")
  private String suffix = "";

  @Bean
  @ConditionalOnMissingBean(InternalResourceViewResolver.class)
  public InternalResourceViewResolver defaultViewResolver() {
    InternalResourceViewResolver resolver = new
InternalResourceViewResolver();
    resolver.setPrefix(this.prefix);
    resolver.setSuffix(this.suffix);
    return resolver;
  }
}
```

视图解析器的配置并没有什么特殊之处，这里真正有意思的是使用了配置属性，从而允许用户对其进行自定义。

它的意思就是说“将会在用户的 application.properties 文件中查找两个变量，这两个变量的名字是 spring.view.prefix 和 spring.view.suffix”。在配置中只需两行代码就能将视图解析器搭建起来了，这是非常便利的。

为了下一章内容的讲解，你需要牢记这一点，不过，我们现在会继续浏览 Spring Boot 的代码。

关于静态资源，配置中包含了如下的内容：

```
private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {
    "classpath:/META-INF/resources/", "classpath:/resources/",
    "classpath:/static/", "classpath:/public/" };

private static final String[] RESOURCE_LOCATIONS;
static {
  RESOURCE_LOCATIONS = new String[CLASSPATH_RESOURCE_LOCATIONS.length
      + SERVLET_RESOURCE_LOCATIONS.length];
  System.arraycopy(SERVLET_RESOURCE_LOCATIONS, 0, RESOURCE_LOCATIONS,
0,
      SERVLET_RESOURCE_LOCATIONS.length);
  System.arraycopy(CLASSPATH_RESOURCE_LOCATIONS, 0, RESOURCE_
LOCATIONS,
      SERVLET_RESOURCE_LOCATIONS.length, CLASSPATH_RESOURCE_LOCATIONS.
length);
}

@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
  if (!this.resourceProperties.isAddMappings()) {
    logger.debug("Default resource handling disabled");
    return;
  }

  Integer cachePeriod = this.resourceProperties.getCachePeriod();
  if (!registry.hasMappingForPattern("/webjars/**")) {
    registry.addResourceHandler("/webjars/**")
        .addResourceLocations("classpath:/META-INF/resources/
webjars/")
        .setCachePeriod(cachePeriod);
}
if (!registry.hasMappingForPattern("/**")) {
  registry.addResourceHandler("/**")
        .addResourceLocations(RESOURCE_LOCATIONS)
        .setCachePeriod(cachePeriod);
  }
}
```

资源位置的声明有点复杂，但是通过它，我们可以了解到以下两点：

- 对带有“webjar”前缀的资源访问将会在类路径中解析。这样的话，我们就能使用 Mavan 中央仓库中预先打包好的 JavaScript 依赖；
- 我们的静态资源需要放在类路径中，并且要位于以下 4 个目录中的任意一个之中，

“/META-INF/resources/”“/resources/”“/static/”或“/public/”。

WebJars 是 JAR 包格式的客户端 JavaScript 库，可以通过 Maven 中央仓库来获取。它们包含了 Maven 项目文件，这个文件允许定义传递性依赖，能够用于所有基于 JVM 的应用之中。WebJars 是 JavaScript 包管理器的替代方案，如 bower 或 npm。对于只需要较少 JavaScript 库的应用来说，这种方案是很棒的。你可以在 www.webjars.org 站点上看到所有可用的 WebJars 列表。

在这个文件中，还专门有一部分用来声明地域管理：

```
@Bean
@ConditionalOnMissingBean(LocaleResolver.class)
@ConditionalOnProperty(prefix = "spring.mvc", name = "locale")
public LocaleResolver localeResolver() {
  return new FixedLocaleResolver(
      StringUtils.parseLocaleString(this.mvcProperties.getLocale()));
}
```

默认的地域解析器只会处理一个地域，并且允许我们通过 spring.mvc.locale 配置属性来进行定义。

1.7 错误与转码配置

还记得在没有添加控制器的时候，第一次启动应用吗？当时看到了一个有意思的“Whitelabel Error Page”输出。

错误处理要比看上去更麻烦一些，尤其是在没有 web.xml 配置文件并且希望应用能够跨 Web 服务器部署时更是如此。好消息是 Spring Boot 将会处理这些事情！让我们看一下 ErrorMvcAutoConfiguration：

```
ConditionalOnClass({ Servlet.class, DispatcherServlet.class })
@ConditionalOnWebApplication
// Ensure this loads before the main WebMvcAutoConfiguration so that
the error View is
// available
@AutoConfigureBefore(WebMvcAutoConfiguration.class)
```

```
@Configuration
public class ErrorMvcAutoConfiguration implements
EmbeddedServletContainerCustomizer,
        Ordered {

    @Value("${error.path:/error}")
    private String errorPath = "/error";

    @Autowired
    private ServerProperties properties;

    @Override
    public int getOrder() {
        return 0;
    }

    @Bean
    @ConditionalOnMissingBean(value = ErrorAttributes.class, search =
SearchStrategy.CURRENT)
    public DefaultErrorAttributes errorAttributes() {
        return new DefaultErrorAttributes();
    }

    @Bean
    @ConditionalOnMissingBean(value = ErrorController.class, search =
SearchStrategy.CURRENT)
    public BasicErrorController basicErrorController(ErrorAttributes
errorAttributes) {
        return new BasicErrorController(errorAttributes);
    }
    @Override
    public void customize(ConfigurableEmbeddedServletContainer
container) {
        container.addErrorPages(new ErrorPage(this.properties.
getServletPrefix()
                + this.errorPath));
    }

    @Configuration
    @ConditionalOnProperty(prefix = "error.whitelabel", name =
"enabled", matchIfMissing = true)
    @Conditional(ErrorTemplateMissingCondition.class)
    protected static class WhitelabelErrorViewConfiguration {
```

```
        private final SpelView defaultErrorView = new SpelView(
                "<html><body><h1>Whitelabel Error Page</h1>"
                        + "<p>This application has no explicit mapping
for /error, so you are seeing this as a fallback.</p>"
                        + "<div id='created'>${timestamp}</div>"
                        + "<div>There was an unexpected error
(type=${error}, status=${status}).</div>"
                        + "<div>${message}</div></body></html>");

        @Bean(name = "error")
        @ConditionalOnMissingBean(name = "error")
        public View defaultErrorView() {
            return this.defaultErrorView;
        }

        // If the user adds @EnableWebMvc then the bean name view
resolver from
        // WebMvcAutoConfiguration disappears, so add it back in to
avoid disappointment.
        @Bean
        @ConditionalOnMissingBean(BeanNameViewResolver.class)
        public BeanNameViewResolver beanNameViewResolver() {
            BeanNameViewResolver resolver = new
BeanNameViewResolver();
            resolver.setOrder(Ordered.LOWEST_PRECEDENCE - 10);
            return resolver;
        }
    }
}
```

这段配置都做了些什么呢？

- 定义了一个 bean，即 DefaultErrorAttributes，它通过特定的属性暴露了有用的错误信息，这些属性包括状态、错误码和相关的栈跟踪信息。
- 定义了一个 BasicErrorController bean，这是一个 MVC 控制器，负责展现我们所看到的错误页面。
- 允许我们将 Spring Boot 的 whitelabel 错误页面设置为无效，这需要将配置文件 application.properties 中的 error.whitelable.enabled 设置为 false。
- 我们还可以借助模板引擎提供自己的错误页面。例如，它的名字是 error.html，ErrorTemplateMissingCondition 条件会对此进行检查。

在本书后面的内容中，我们将会看到如何恰当地处理错误。

至于转码的问题，非常简单的 HttpEncodingAutoConfiguration 将会负责处理相关的事宜，这是通过提供 Spring 的 CharacterEncodingFilter 类来实现的。通过 spring.http.encoding.charset 配置，我们可以覆盖默认的编码（“UTF-8”），也可以通过 spring.http.encoding.enabled 禁用这项配置。

1.8　嵌入式 Servlet 容器（Tomcat）的配置

默认情况下，Spring Boot 在打包和运行应用时，会使用 Tomcat 嵌入式 API（Tomcat embedded API）。

我们来看一下 EmbeddedServletContainerAutoConfiguration：

```
@Order(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@Import(EmbeddedServletContainerCustomizerBeanPostProcessorRegistrar.
class)
public class EmbeddedServletContainerAutoConfiguration {

  /**
  * Nested configuration for if Tomcat is being used.
  */
  @Configuration
  @ConditionalOnClass({ Servlet.class, Tomcat.class })
  @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.
class, search = SearchStrategy.CURRENT)
  public static class EmbeddedTomcat {

    @Bean
    public TomcatEmbeddedServletContainerFactory
tomcatEmbeddedServletContainerFactory() {
      return new TomcatEmbeddedServletContainerFactory();
    }
  }
  /**
  * Nested configuration if Jetty is being used.
  */
  @Configuration
```

```
  @ConditionalOnClass({ Servlet.class, Server.class, Loader.class })
  @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.
class, search = SearchStrategy.CURRENT)
  public static class EmbeddedJetty {

    @Bean
    public JettyEmbeddedServletContainerFactory
jettyEmbeddedServletContainerFactory() {
      return new JettyEmbeddedServletContainerFactory();
    }
  }
  /**
  * Nested configuration if Undertow is being used.
  */
  @Configuration
  @ConditionalOnClass({ Servlet.class, Undertow.class,
SslClientAuthMode.class })
  @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.
class, search = SearchStrategy.CURRENT)
  public static class EmbeddedUndertow {

    @Bean
    public UndertowEmbeddedServletContainerFactory
undertowEmbeddedServletContainerFactory() {
      return new UndertowEmbeddedServletContainerFactory();
    }
  }
}
```

上面的代码非常简单直接，这个代码包含了 3 个不同的配置，哪一个会处于激活状态要取决于类路径下哪些内容是可用的。

可以将 Spring Boot 与 Tomcat、tc-server、Jetty 或者 Undertow 结合使用。服务器可以很容易地进行替换，只需将 spring-boot-starter-tomcat JAR 依赖移除掉，并将其替换为 Jetty 或 Undertow 对应的依赖即可。如果你想这样做的话，请参考相关的文档。

对 Servlet 容器（Tomcat）的所有配置都会在 TomcatEmbeddedServletContainerFactory 中进行。尽管你应该读一下这个类，它为嵌入式 Tomcat 提供一个非常高级的配置（为其查找文档会非常困难），但是在这里我们不会直接查看这个类。

我会为读者介绍配置 Servlet 容器时不同的选项。

1.8.1　HTTP 端口

通过在 application.properties 文件中定义 server.port 属性或者定义名为 SERVER_PORT 的环境变量，我们可以修改默认的 HTTP 端口。

通过将该变量设置为−1，可以禁用 HTTP，或者将其配置为 0，这样的话，就会在随机的端口上启动应用。对于测试，这是很便利的。

1.8.2　SSL 配置

配置 SSL 是一项很麻烦的事情，但是 Spring Boot 有一项很简单的解决方案。我们只需一点属性就能保护服务器了：

```
server.port = 8443
server.ssl.key-store = classpath:keystore.jks
server.ssl.key-store-password = secret
server.ssl.key-password = another-secret
```

不过，为了使上面的例子运行起来，我们需要生成一个 keystore 文件。

我们将会在第 6 章中，深入介绍安全的可选方案。当然，我们还可以通过添加自己 EmbeddedServletContainerFactory 来进一步自定义 TomcatEmbeddedServletContainerFactory 的功能。如果你希望添加多个连接器的话，这会是非常便利的，可以参考 http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html#howto-configure-ssl 来获取更多信息。

1.8.3　其他配置

在配置中，我们可以通过简单地声明@Bean 元素来添加典型的 Java Web 元素，如 Servlet、Filter 和 ServletContextListener。

除此之外，Spring Boot 还为我们内置了 3 项内容：

- 在 JacksonAutoConfiguration 中，声明使用 Jackson 进行 JSON 序列化；
- 在 HttpMessageConvertersAutoConfiguration 中，声明了默认的 HttpMessageConverter；
- 在 JmxAutoConfiguration 中，声明了 JMX 功能。

我们将会在第 5 章中，更详细地了解 Jackson 的配置。关于 JMX 配置，我们可以在本

地通过 jconsole 连接应用之后进行尝试，如图 1-9 所示。

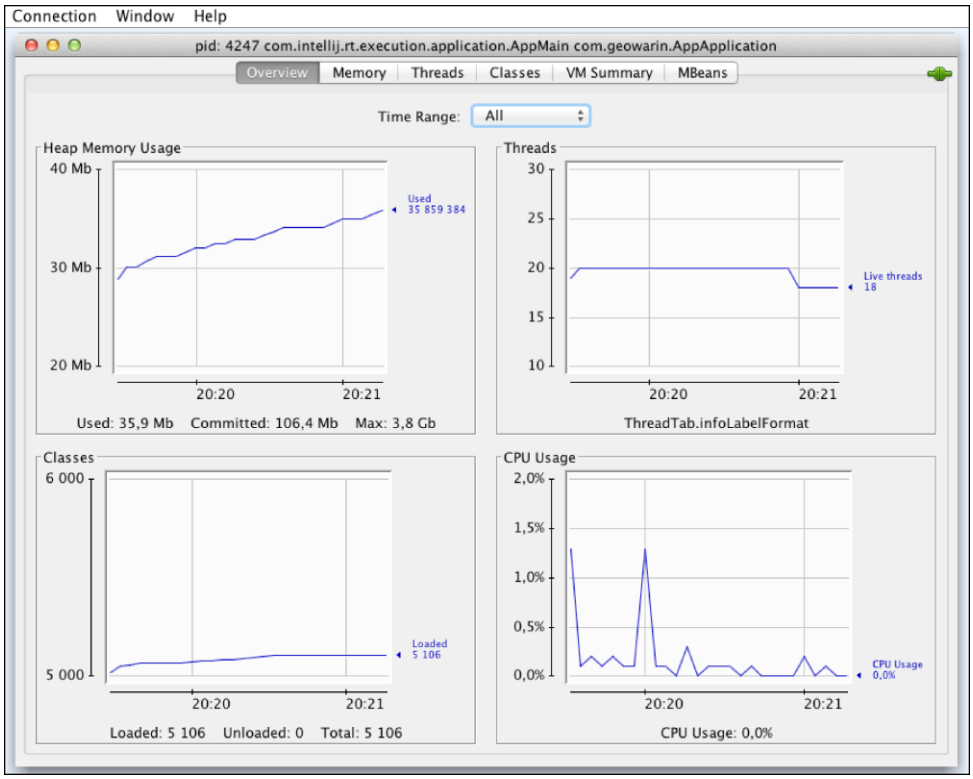

图 1-9

通过将 org.springframework.boot:spring-boot-starter-actuator 添加到类路径下，我们可以添加更多有意思的 MBean。我们甚至可以定义自己的 MBean，并通过 Jolokia 将其暴露为 HTTP。另一方面，我们也可以禁用这些端点，只需在配置中添加 spring.jmx.enabled=false 即可。

参考 http://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-jmx.html 了解更多细节。

1.9 小结

我们现在已经有了一个非常简陋的 Spring Web 应用，虽然我们没有对其进行什么配置，但是它可以输出 RESTful JSON 的“Hello world”。我们已经看到过 Spring Boot 做了什么、

是如何实现的，也了解到如何重写默认的自动配置。

单是详细介绍 Spring Boot 如何运行就够写一本书了。如果你想更深入探究的话，我推荐一本很棒的书，这就是由 Greg Turnquist 编写的《Learning Spring Boot》，它和本书是同一个系列的。

现在，我们已经为下一章做好了准备，在下一章中这个应用将会推进到一个新的阶段，它会真正地提供 Web 页面，我们还会学习 Spring MVC 的更多哲学理念。

第 2 章
精通 MVC 架构

在本章中，我们将会讨论 MVC 架构理念以及 Spring MVC 是如何实现这些理念的。

本章将会继续使用上一章的应用，并且会构建一些有意思的功能。目标是设计一个简单的页面，在这个页面上用户可以根据特定的条件（criteria）查询 Tweet，并将其展现出来。

为了实现该功能，我们将使用 Spring Social Twitter 项目，可以通过该地址 http://projects.spring.io/spring-social-twitter/了解这个项目。

我们会让 Spring MVC 与一个现代化的模板引擎协作，这个引擎也就是 Thymeleaf，并且还会试图理解这个框架的内部机制。引导用户在不同的视图间流转，最后，会借助 WebJars 和 Materialize（http://materializecss.com）让应用在外观上看起来更棒。

2.1 MVC 架构

大多数人对 MVC 这个缩写应该不会感到陌生。它代表的是模型（Model）、视图（View）和控制器（Controller），它会将数据和展现层进行解耦，被视为构建用户界面的一种很流行的方式。

自从在 Smalltalk 领域中提出这个理念，并在 Ruby on Rails 框架中采用之后，MVC 就变得广受欢迎。

如图 2-1 所示，它的架构可以分为 3 层。

- 模型：包含了应用中所需的各种展现数据。
- 视图：由数据的多种表述所组成，它将会展现给用户。

- 控制器：将会处理用户的操作，它是连接模型和视图的桥梁。

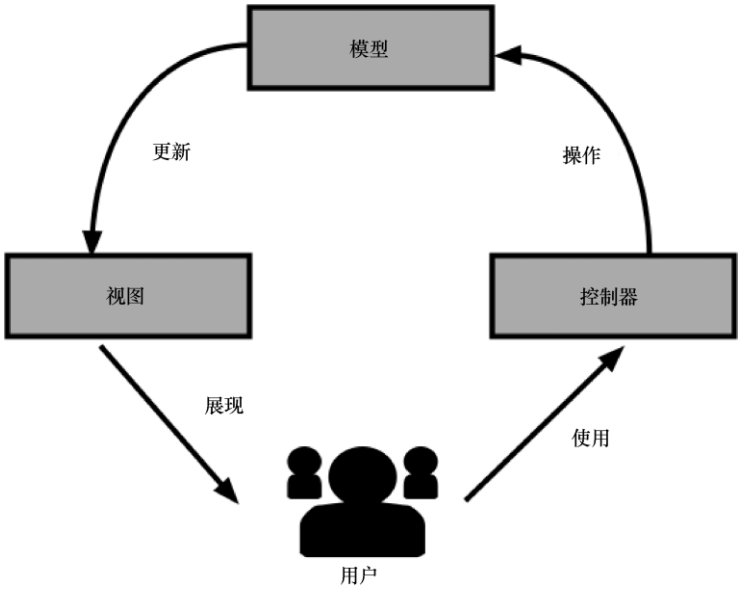

图 2-1

MVC 背后的理念是将视图与模型进行解耦，模型必须是自包含的并且与 UI 无关。这样的话，基本上就可以实现相同的数据跨多个视图重用。其实，这些视图就是以不同的方式来查看数据。通过钻取（Drill down）或使用不同的渲染器（HTML、PDF），可以很好地阐述这一原则。

控制器会作为用户和数据的中间协调者，它的角色就是控制终端用户的可用行为，并引导他们在应用的不同视图间跳转。

2.2　对 MVC 的质疑及其最佳实践

尽管 MVC 依然是当前设计 UI 的首选方案，但是随着它的流行，也有很多对它的批评。实际上，大多数的批评都指向了该模式的错误用法。

2.2.1　贫血的领域模型

Eric Evans 编写过一本很有影响力的书，名为《领域驱动设计》（Domain Driven Design，DDD）。在这本书中，定义了一组架构规则，能够指导我们更好地将业务领域集成到代

码之中。

其中有一项核心的理念就是将面向对象的范式应用到领域对象之中。如果违背这一原则的话，就会被称之为贫血的领域模型（Anemic Domain Model）。Martin Fowler 的博客（http://www.martinfowler.com/bliki/AnemicDomainModel.html）对这一问题进行了很好的定义。

贫血的领域模型通常来讲会具有如下的症状：

- 模型是由简单老式的 Java 对象（plain old Java object，POJO）所构成的，只有 getter 和 setter 方法；
- 所有业务逻辑都是在服务层处理的；
- 对模型的校验会在本模型外部进行，例如在控制器中。

根据业务领域的复杂性不同，这可能是一种较差的实践方式。通常来讲，DDD 实践需要付出额外的努力，将领域从应用逻辑中分离出来。

架构通常都是一种权衡，需要注意的是，设计 Spring 应用的典型方式往往会在这个过程中导致系统在可维护性上变得较为复杂。

避免领域贫血的途径如下：

- 服务层适合进行应用级别的抽象（如事务处理），而不是业务逻辑；
- 领域对象应该始终处于合法的状态。通过校验器（validator）或 JSR-303 的校验注解，让校验过程在表单对象中进行；
- 将输入转换成有意义的领域对象；
- 将数据层按照 Repository 的方式来实现，Repository 中会包含领域查询（例如参考 Spring Data 规范）；
- 将领域逻辑与底层的持久化框架解耦；
- 尽可能使用实际的对象，例如操作 FirstName 类而不是操作 String。

DDD 所涉及的内容远不止上述的规则：实体（Entity）、值类型（value type）、通用语言（Ubiquitous Language）、限界上下文（Bounded Context）、洋葱架构（Onion Architecture）以及防腐化层（anti corruption layer），我强烈建议你自行学习一下这些原则。就我们而言，在构建 Web 应用的过程中，会努力遵循上述的指导原则。随着本书的推进，你会对这些关注点更加熟悉的。

2.2.2　从源码中学习

如果熟悉 Spring 的话，那么很可能你已经访问过 Spring 的 Web 站点，即 http://spring.io。它全部是由 Spring 构建的，而且很棒的一点在于它是开源的。

这个项目的名称为 sagan，它包含了很多有意思的特性：

- 基于 Gradle 的多模块项目；
- 集成了安全；
- 集成了 Github；
- 集成了 Elasticsearch；
- JavaScript 前端应用。

这个项目的 Github wiki 页面非常详尽，能够帮助你非常容易地开始了解该项目。

如果你对这个实际应用的 Spring 架构感兴趣的话，可以访问如下的 URL:
https://github.com/spring-io/sagan

2.3　Spring MVC 1-0-1

在 Spring MVC 中，模型是由 Spring MVC 的 Model 或 ModelAndView 封装的简单 Map。它可以来源于数据库、文件、外部服务等，这取决于你如何获取数据并将其放到模型中。与数据层进行交互的推荐方式是使用 Spring Data 库：Spring Data JPA、Spring Data MongoDB 等。有 10 多个与 Spring Data 相关的项目，推荐你查看一下 http://projects.spring.io/spring-data。

Spring MVC 的控制层是通过使用@Controller 注解来进行处理的。在 Web 应用中，控制器的角色是响应 HTTP 请求。带有@Controller 注解的类将会被 Spring 检索到，并且能够有机会处理传入的请求。

通过使用@RequestMapping 注解，控制器能够声明它们会根据 HTTP 方法（如 GET 或 POST 方法）和 URL 来处理特定的请求。控制器就可以确定是在 Web 响应中直接写入内容，还是将应用路由一个视图并将属性注入到该视图中。

纯粹的 RESTful 应用将会选择第一种方式，并且会在 HTTP 响应中直接暴露模型的

JSON 或 XML 表述，这需要用到@ResponseBody 注解。在 Web 应用中，这种类型的架构通常会与前端 JavaScript 框架关联，如 Backbone.js、AngularJS 或 React。在这种场景中，Spring 应用只需处理 MVC 中的模型层。我们将会在第 4 章中学习这种架构。

在第二种方式中，模型会传递到视图中，视图会由模板引擎进行渲染，并写入到响应之中。

视图通常会与某种模板方言关联，这种模板允许遍历模型中的内容，流行的模板方言包括 JSP、FreeMarker 或 Thymeleaf。

混合式的方式则会在某些方面采用模板引擎与应用进行交互，并将视图层委托给前端框架。

2.4 使用 Thymeleaf

Thymeleaf 是一个模板引擎，在 Spring 社区中，它备受关注。

它的成功在很大程度上要归因于对用户友好的语法（它几乎就是 HTML）以及扩展的便利性。

如表 2-1 所示，现在有各种可用的扩展，并且能够与 Spring Boot 进行集成。

表 2-1

所支持的功能	依赖
布局	nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect
HTML 5 data-*属性	com.github.mxab.thymeleaf.extras:thymeleaf-extras-data-attribute
Internet Explorer 的条件注释	org.thymeleaf.extras:thymeleaf-extras-conditionalcomments
对 Spring Security 的支持	org.thymeleaf.extras:thymeleaf-extras-springsecurity3

关于 Thymeleaf 与 Spring 集成有一个很好的入门指南，参见 http://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html。

闲言少叙，现在我们将 spring-boot-starter-thymeleaf 依赖添加进来，这样就能启动 Thymeleaf 模板引擎了：

```
buildscript {
    ext {
```

```
        springBootVersion = '1.2.5.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
        classpath("io.spring.gradle:dependency-management-
plugin:0.5.1.RELEASE")
    }
}
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'
apply plugin: 'io.spring.dependency-management'

jar {
    baseName = 'masterSpringMvc'
    version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}


dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    compile 'org.springframework.boot:spring-boot-starter-thymeleaf'
    testCompile 'org.springframework.boot:spring-boot-starter-test'
}

eclipse {
    classpath {
         containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
         containers 'org.eclipse.jdt.launching.JRE_CONTAINER/org.
eclipse.jdt.internal.debug.ui.launcher.StandardVMType/JavaSE-1.8'
    }
}
```

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.3'
}
```

第一个页面

现在，我们将第一个页面添加到应用之中，我们将其放到 src/main/resources/templates 中，并将其命名为 resultPage.html：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello thymeleaf</title>
</head>
<body>
    <span th:text="|Hello thymeleaf|">Hello html</span>
</body>
</html>
```

我们首先能够看到 Thymeleaf 与 HTML 结合得非常好，它的语法看上去也非常自然。

th:text 的值放在两个竖线中间，这意味着文本中所有的值都会连接在一起。

看上去，这可能有点怪异，但实际上，我们很少在页面中进行硬编码，因此，Thymeleaf 在这里采用了具有一定倾向性的设计。

对于 Web 设计人员来说，Thymeleaf 有一项很大的优势，那就是在服务器没有运行的时候，模板中所有的动态内容都可以采用一个默认值。资源 URL 可以采用相对的路径来指定，每个标签都可以包含占位符。在前面的样例里面，如果是在应用的上下文中，那么文本“Hello html”将不会显示，但是如果直接在 Web 浏览器中打开这个文件的话，那么它就会显示了。

为了加快开发速度，在 application.properties 文件中添加该属性：

```
spring.thymeleaf.cache=false
```

这将会禁止启用视图缓存，每次访问的时候都会重新加载模板。

当然，在部署到生产环境时，该项配置需要禁用。在第 8 章时，我们会再进行设置。

重新加载视图

如果禁用了缓存，在修改视图之后，只需在 Eclipse 中进行保存或者在 IntelliJ 中使用 Build > Make Project 操作就可以刷新视图。

最后，需要修改 HelloController 类，它必须要导航至我们新建的视图，而不是展现简单的文本。为了完成该功能，需要移除@ResponseBody 注解。这样做完之后，如果再次返回字符串的话，就会告诉 Spring MVC 要将这个字符串映射为视图名，而不是在响应中直接展现特定的模型。

我们的控制器将会变为如下所示：

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String hello() {
        return "resultPage";
    }
}
```

在本例中，控制器会将用户转移到名为 resultPage 的视图中，ViewResolver 接口会将这个名字与我们的视图进行关联。

再次启动应用并转至 http://localhost:8080。

你将会看到如图 2-2 所示的页面。

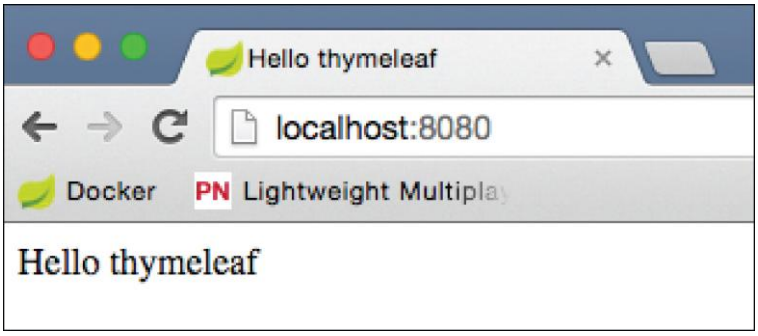

图 2-2

2.5 Spring MVC 架构

让我们从这个新的“Hello World”页面后退一步，尝试去理解在这个 Web 应用中到底发生了什么。为了做到这一点，需要跟踪浏览器所发送的 HTTP 请求的行程以及它是如何从服务器端得到响应的。

2.5.1 DispatcherServlet

每个 Spring Web 应用的入口都是 DispatcherServlet。图 2-3 展现了 DispatcherServlet 的架构。

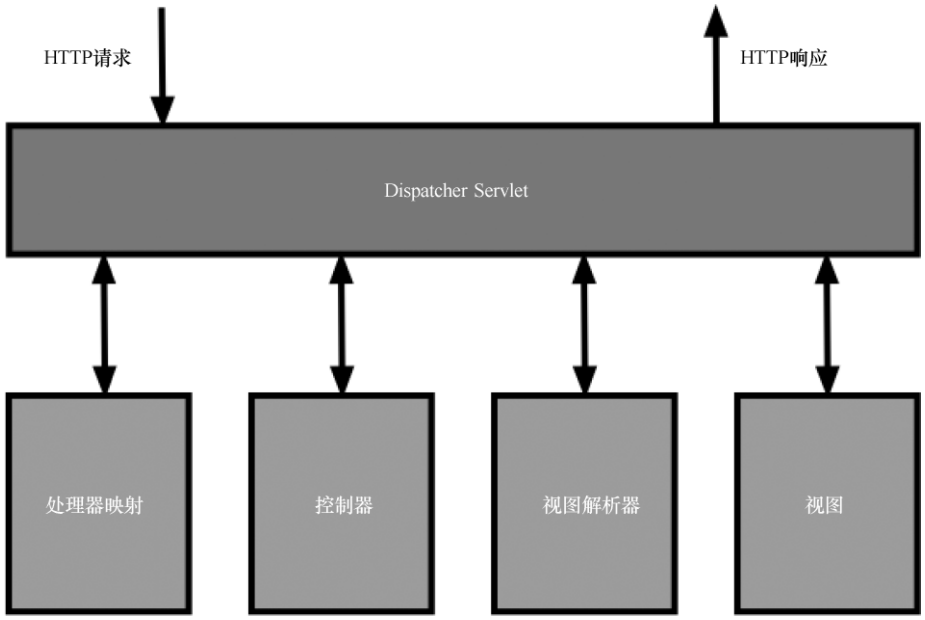

图 2-3

这个一个典型的 HttpServlet 类，它会将 HTTP 请求分发给 HandlerMapping。HandlerMapping 会将资源（URL）与控制器关联起来。

控制器上对应的方法（也就是带有@RequestMapping 注解的方法）将会被调用。在这个方法中，控制器会设置模型数据并将视图名称返回给分发器。

然后，DispatcherServlet 将会查询 ViewResolver 接口，从而得到对应视图的实现。

在样例中，ThymeleafAutoConfiguration 将会为我们搭建视图解析器。

通过查看 ThymeleafProperties 类，可以知道视图的默认前缀是“classpath:/templates/”，后缀是“.html”。

这就意味着，假设视图名为 resultPage，那么视图解析器将会在类路径的 templates 目录下查找名为 resultPage.html 的文件。

在我们的应用中，ViewResolver 接口是静态的，但是更为高级的实现能够根据请求的头信息或用户的地域信息，返回不同的结果。

视图最终将会被渲染，其结果会写入到响应之中。

2.5.2　将数据传递给视图

第一个页面完全是静态的，其实并没有真正发挥出 Spring MVC 的威力。我们现在更进一步，如果“Hello World”这个字符串不是硬编码的，而是来源于服务器，那该怎么实现呢？

你可能会问，还是显示这个无聊的“hello world”吗？是的，不过这种方式会开启更多的可能性。现在，修改 resultPage.html 文件，让它展现来自模型中的信息：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello thymeleaf</title>
</head>
<body>
    <span th:text="${message}">Hello html</span>
</body>
</html>
```

然后，我们修改控制器，将该信息保存到模型中：

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String hello(Model model) {
        model.addAttribute("message", "Hello from the controller");
        return "resultPage";
    }
}
```

我知道，这种悬疑的感觉会让你觉得很受折磨！那么，我们访问 http://localhost:8080，看一下效果是什么样子的（见图 2-4）。

图 2-4

需要注意的第一件事情就是传递了一个新的参数到控制器的方法之中，DispatcherServlet 会提供正确的对象。实际上，很多对象都可以注入到控制器的方法之中，例如 HttpRequest、HttpResponse、Locale、TimeZone 和 Principal，其中 Principal 代表了一个认证过的用户。完整的对象列表可以在文档中查阅，参见 http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-arguments。

2.6 Spring 表达式语言

当使用“${}”语法时，我们实际上使用的是 Spring 表达式语言（Spring Expression Language，SpEL）。关于 EL，有多个不同的变种，而 SpEL 是其中威力强大的一种。

表 2-2 是它主要特性的概览。

表 2-2

特性	语法	描述
访问列表元素	list[0]	
访问 Map 条目	map[key]	
三元运算符	condition ? 'yes' :'no'	
Elvis 运算符	person ?: default	如果 person 的值为空的话，将会返回 default
安全导航	person?.name	如果 person 为空或其 name 为空的话，返回 null
模板	'Your name is #{person.name}'	将值注入到字符串中
投影	${persons.![name]}	抽取所有 persons 的 name，并将其放到一个列表中
选择	persons.?[name =='Bob']'	返回列表中 name 为 Bob 的 person
函数调用	person.sayHello()	

如果要查看完整的指导文档，可以参考其手册：http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html。

SpEL 的用处并不仅限于视图之中，可以将它用在 Spring 框架的各种地方，例如，在通过@Value 注解往 bean 中注入属性时，也可以使用 SpEL。

从请求参数中获取数据

我们已经能够在视图中展现来自服务端的数据，但是，如果想获取用户的输入该怎么办呢？根据 HTTP 协议，有很多方式可以实现这一点，其中最简单的就是传递查询参数到 URL 之中。

查询参数

你肯定早就知道什么是查询参数了吧，它们会位于 URL 的“?”字符后面，是由名称和值所组成的列表，每一项会使用“&”符号进行分割，例如：page?var1=value1&var2=value2。

可以使用这项技术要求用户提供他们的名字，再次修改 HelloController 类，如下所示：

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String hello(@RequestParam("name") String userName, Model
model) {
        model.addAttribute("message", "Hello, " + userName);
        return "resultPage";
    }
}
```

如果此时导航至 localhost:8080/?name=Geoffroy，将会看到如图 2-5 所示的结果。

图 2-5

默认情况下，请求参数是强制要求存在的。这意味着，如果导航至 localhost:8080，那么将会看到一个错误页面。

查阅一下@RequestParam 的代码，可以看到除了 value 属性以外，它还有其他两个可用的属性：required 和 defaultValue。

因此，可以修改一下代码，为其指定一个默认值或者将其设置为非必填项：

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String hello(@RequestParam(defaultValue = "world") String
name, Model model) {
        model.addAttribute("message", "Hello, " + name);
        return "resultPage";
    }
}
```

在 Java 8 中，我们可以不指定 value 参数。如果这样的话，将会使用带有注解的方法参数名称。

2.7 结束 Hello World，开始获取 Tweet

好了，毕竟这本书的名字不是“精通 Hello World”，我们结束这一话题。借助 Spring，使用 Twitter 的 API 进行查询是非常容易的事情。

2.7.1 注册应用

在开始之前，我们需要在 Twitter 的开发者控制台中注册应用。

访问 https://apps.twitter.com，并创建一个新的应用。

根据你喜好为其设定一个名称，在 Website 和 Callback URL 区域中，输入 http://127.0.0.1:8080（见图 2-6）。这样的话，就能在本地机器上，测试开发阶段的应用。

现在，导航至“Keys and Access Token”，并复制 **Consumer Key** 和 **Consumer Secret**，稍后我们会用到它们，参见图 2-7 所示的截图。

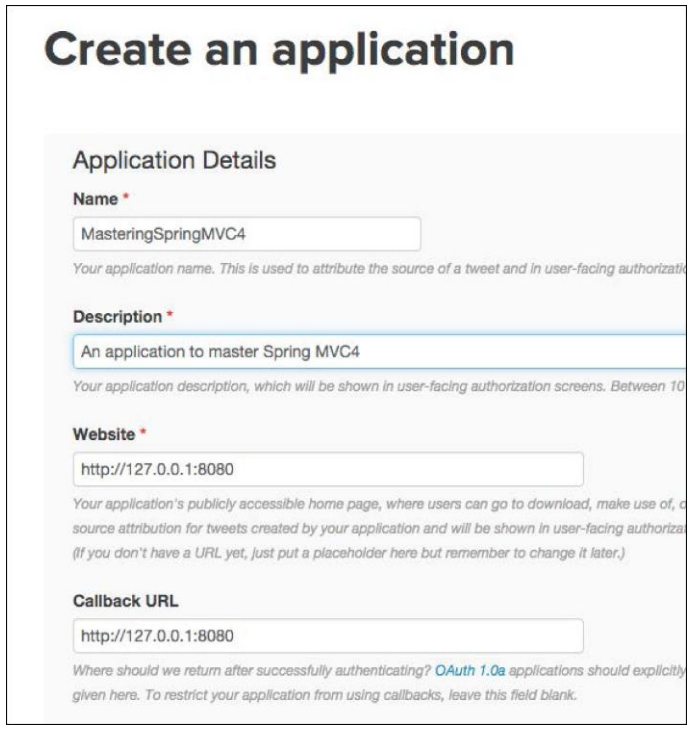

图 2-6

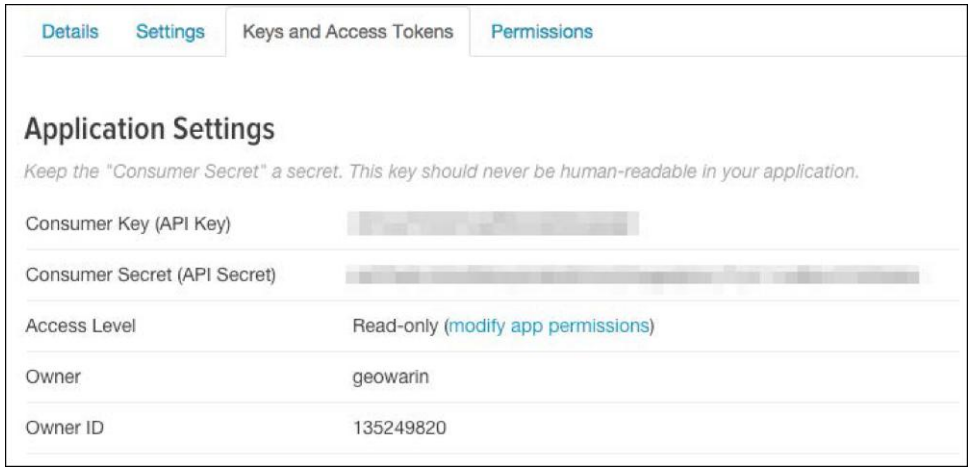

图 2-7

默认情况下，应用会具有只读的权限，对于该应用来说，这就足够了，但是如果你愿

意的话，可以对其进行调整。

2.7.2 搭建 Spring Social Twitter

添加如下的依赖到 build.gradle 文件中：

```
compile 'org.springframework.boot:spring-boot-starter-social-twitter'
```

Spring Social 是一组项目，提供了对各种社交网络公开 API 的访问功能。Spring Boot 内置了对 Twitter、Facebook 和 LinkedIn 的支持。Spring Social 一共包括大约 30 个项目，可以访问 http://projects.spring.io/spring-social/来进一步了解它的情况。

添加如下的两行代码到 application.properties 中：

```
spring.social.twitter.appId= <Consumer Key>
spring.social.twitter.appSecret= <Consumer Secret>
```

这是与刚才所创建的应用相关的 key。

我们将会在第 6 章中详细介绍 OAuth。就现在而言，只是使用这些凭证信息发送请求到 Twitter 的 API 上，以满足我们应用的需要。

2.7.3 访问 Twitter

现在，就可以在控制器中使用 Twitter 了，将它的名字改为 TweetController，从而能够以更好的方式反映其新功能：

```
@Controller
public class TweetController {

    @Autowired
    private Twitter twitter;

    @RequestMapping("/")
    public String hello(@RequestParam(defaultValue =
"masterSpringMVC4") String search, Model model) {
        SearchResults searchResults = twitter.searchOperations().
search(search);
```

```
            String text = searchResults.getTweets().get(0).getText();
            model.addAttribute("message", text);
            return "resultPage";
        }
    }
```

我们可以看到，上面的代码会搜索匹配请求参数的 Tweet。如果一切运行正常的话，结果中第一条记录的文本将会显示出来（见图 2-8）

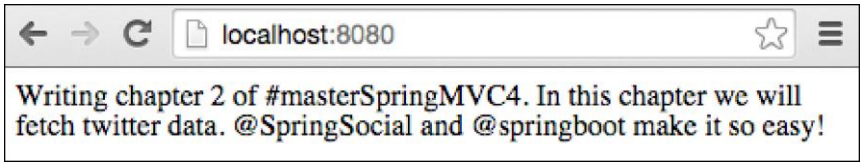

图 2-8

当然，如果搜索没有得到任何结果的话，这段蹩脚的代码将会因为 ArrayOutOfBoundException 异常而导致失败。因此，可以抓紧发一条 Tweet 来解决这个问题！

如果想展现 Tweet 列表的话，那该怎么办呢？让我们修改一下 resultPage.html 文件：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello twitter</title>
</head>
<body>
    <ul>
        <li th:each="tweet : ${tweets}" th:text="${tweet}">Some
tweet</li>
    </ul>
</body>
</html>
```

th:each 是由 Thymeleaf 所定义的标签，它允许我们遍历一个集合并且能够在循环中将集合中的每个值赋给一个变量。

我们也需要修改控制器：

```
@Controller
public class TweetController {

    @Autowired
    private Twitter twitter;

    @RequestMapping("/")
    public String hello(@RequestParam(defaultValue =
"masterSpringMVC4") String search, Model model) {
        SearchResults searchResults = twitter.searchOperations().
search(search);
        List<String> tweets =
                searchResults.getTweets()
                        .stream()
                        .map(Tweet::getText)
                        .collect(Collectors.toList());
        model.addAttribute("tweets", tweets);
        return "resultPage";
    }
}
```

注意，我们在这里使用了 Java 8 的流来收集 Tweet 的信息。Tweet 类包含了很多其他的属性，如发送者、转推的数量等。但是，现在我们尽可能地保持简单，如图 2-9 中的截图所示。

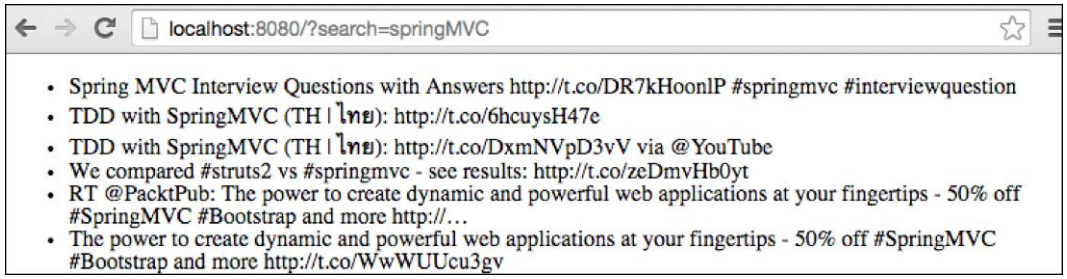

图 2-9

2.8 Java 8 的流和 lambda 表达式

可能你对 lambda 还不太了解，在 Java 8 中，每个集合都会有一个默认的方法 stream()，它能够实现函数式风格的操作。

这些操作可以是中间操作（intermediate_operation），它会返回一个流，这样就能将其连接起来，也可以是终止操作（terminal operation），这样的话会返回一个值。

最著名的中间操作如下所示。

- map：它会为列表中的每个元素都应用某个方法，并返回结果所组成的列表；
- filter：它会返回匹配断言的所有元素；
- reduce：它会借助一个操作和累加器（accumulator）将一个列表聚合到单个值上。

Lambda 是函数表达式的便捷语法，它可以用到单个的抽象方法（Single Abstract Method）之中，也就是只包含一个函数的接口。

例如，我们可以按照如下的方式来实现 Comparator 接口：

```
Comparator<Integer> c = (e1, e2) -> e1 - e2;
```

在 lambda 之中，return 关键字就是最后的表达式。

之前所使用的双冒号操作符是引用类函数的快捷方式：

```
Tweet::getText
```

之前的表达式等价于：

```
(Tweet t) -> t.getText()
```

collect 方法允许我们调用一个终止操作。Collectors 类是一组终止操作，它会将结果放到列表、集合或 Map 之中，允许进行分组（grouping）、连接（joining）等操作。

调用 collect(Collectors.toList())方法将会产生一个列表，其中包含了流中的每一个元素，在我们的例子中，也就是 Tweet 的内容。

2.9　使用 WebJars 实现质感设计

现在，我们的应用已经很棒了，但是在美学方面却差得很多。你可能听说过质感设计（material design），这是 Google 的扁平化设计。

如图 2-10 所示，我们将会使用 Materialize（http://materializecss.com），这是一个非常漂亮的 CSS 和 JavaScript 库，与 Bootstrap 类似。

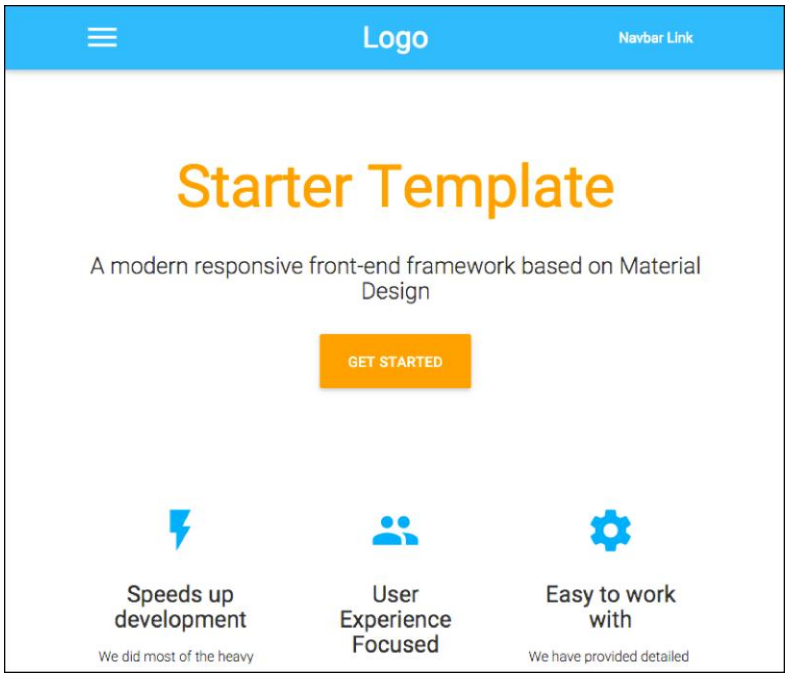

图 2-10

在第 1 章中，我们曾经简单介绍过 WebJars，现在要开始使用它们了。在依赖中，我们要添加 jQuery 和 Materialize CSS：

```
compile 'org.webjars:materializecss:0.96.0'
compile 'org.webjars:jquery:2.1.4'
```

每个 WebJar 的结构都是标准的，每个库的 JS 和 CSS 文件都会位于/webjars/{lib}/{version}/*.js 中。

例如，如果要添加 jQuery 到我们的页面中，那 Web 页面需要如下所示：

```
<script src="/webjars/jquery/2.1.4/jquery.js"></script>
```

接下来修改一下控制器，让它显示所有 Tweet 对象的列表，而不是只显示简单的文本：

```
package masterSpringMvc.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.SearchResults;
import org.springframework.social.twitter.api.Tweet;
```

```
import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.List;

@Controller
public class TweetController {

    @Autowired
    private Twitter twitter;

    @RequestMapping("/")
    public String hello(@RequestParam(defaultValue =
"masterSpringMVC4") String search, Model model) {
        SearchResults searchResults = twitter.searchOperations().
search(search);
        List<Tweet> tweets = searchResults.getTweets();
        model.addAttribute("tweets", tweets);
        model.addAttribute("search", search);
        return "resultPage";
    }
}
```

在视图中，将 Materialize CSS 包含进来：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title>Hello twitter</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
type="text/css" rel="stylesheet" media="screen,projection"/>
</head>
<body>
<div class="row">

    <h2 class="indigo-text center" th:text="|Tweet results for
${search}|">Tweets</h2>
```

```
    <ul class="collection">
        <li class="collection-item avatar" th:each="tweet :
${tweets}">
            <img th:src="${tweet.user.profileImageUrl}" alt=""
class="circle"/>
            <span class="title" th:text="${tweet.user.
name}">Username</span>
            <p th:text="${tweet.text}">Tweet message</p>
        </li>
    </ul>

</div>

<script src="/webjars/jquery/2.1.4/jquery.js"></script>
<script src="/webjars/materializecss/0.96.0/js/materialize.js"></
script>
</body>
</html>
```

如图 2-11 所示结果看起来会好很多。

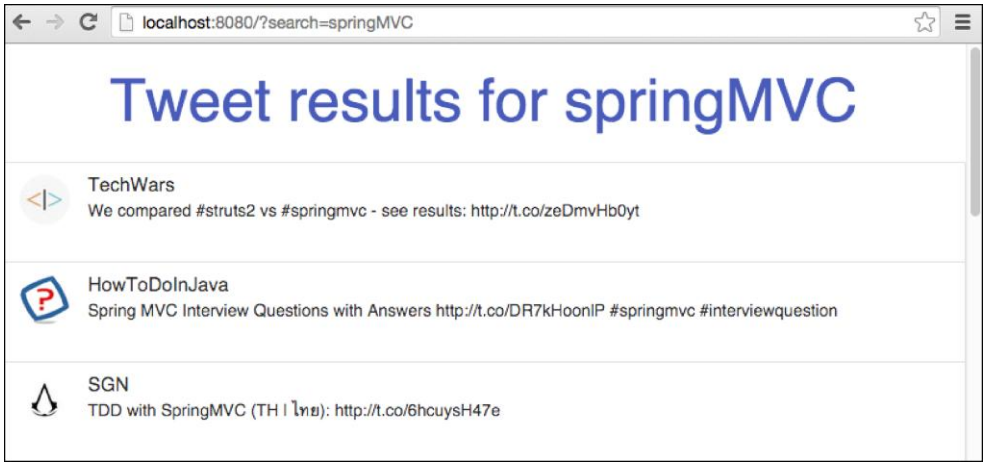

图 2-11

2.9.1 使用布局

我们最后想实现的就是将 UI 中可重用的代码块放到模板之中。为了实现该功能，我们需要使用 thymeleaf-layout-dialect 依赖，它已经包含在项目的 spring-boot-starter-thymeleaf 依赖里面。

我们会创建一个新的文件，名为 default.html，并将其放在 src/main/resources/templates/layout 之中，它包含了每个页面中都重复出现的代码：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-
scale=1, maximum-scale=1.0, user-scalable=no"/>
    <title>Default title</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
type="text/css" rel="stylesheet" media="screen,projection"/>
</head>
<body>
<section layout:fragment="content">
    <p>Page content goes here</p>
</section>

<script src="/webjars/jquery/2.1.4/jquery.js"></script>
<script src="/webjars/materializecss/0.96.0/js/materialize.js"></
script>
</body>
</html>
```

现在，我们要修改 resultPage.html 文件，让它使用布局，这会简化它的内容：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Hello twitter</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center" th:text="|Tweet results for
${search}|">Tweets</h2>

    <ul class="collection">
```

```
            <li class="collection-item avatar" th:each="tweet :
${tweets}">

                <img th:src="${tweet.user.profileImageUrl}" alt=""
class="circle"/>
                <span class="title" th:text="${tweet.user.
name}">Username</span>

                <p th:text="${tweet.text}">Tweet message</p>
            </li>
        </ul>
</div>
</body>
</html>
```

其中，layout:decorator="layout/default"能够表明去哪里查找布局。这样，我们可以将内容注入到布局的不同 layout:fragment 区域中。需要注意的是，每个模板都是合法的 HTML 文件，我们可以非常容易地重写它的标题。

2.9.2 导航

我们现在已经有了一个很棒的用于展现 Tweet 的小应用，但是，我们的用户需要提供一个“search”请求参数，这该如何实现呢（见图 2-12）？

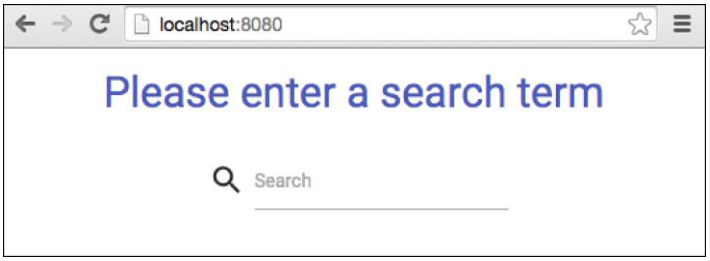

图 2-12

如果我们能够在应用上增加一个小表单的话，那就非常完美了。

我们接下来要做的事情如下所示。

首先，我们需要修改 TweetController，为应用增加第二个视图。访问应用的根目录会展现出搜索页面，在 search 域中点击回车键会展现结果页面：

```
@Controller
public class TweetController {
```

```
    @Autowired
    private Twitter twitter;

    @RequestMapping("/")
    public String home() {
        return "searchPage";
    }

    @RequestMapping("/result")
    public String hello(@RequestParam(defaultValue =
"masterSpringMVC4") String search, Model model) {
        SearchResults searchResults = twitter.searchOperations().
search(search);
        List<Tweet> tweets = searchResults.getTweets();
        model.addAttribute("tweets", tweets);
        model.addAttribute("search", search);
        return "resultPage";
    }
}
```

我们会添加另外一个页面到 templates 文件夹下，并将其命名为 searchPage.html 文件。这个页面会包含一个简单的表单，它会通过 get 方法将要搜索的术语传递到结果页面：

```
<!DOCTYPE html>
<html xmlns:th="http://www.w3.org/1999/xhtml"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Search</title>
</head>
<body>

<div class="row" layout:fragment="content">

    <h4 class="indigo-text center">Please enter a search term</h4>

    <form action="/result" method="get" class="col s12">
        <div class="row center">
            <div class="input-field col s6 offset-s3">
                <i class="mdi-action-search prefix"></i>
                <input id="search" name="search" type="text"
class="validate"/>
```

```
                <label for="search">Search</label>
            </div>
        </div>
    </form>
</div>

</body>
</html>
```

这是一个很简单的 HTML，但是已经可以正常运行了，你现在可以尝试一下。

如果不允许展现某些搜索结果的话，该怎么办呢？假设如果用户输入 struts 的话，我们想展现一个出错页面。

要实现该功能，最好的方式就是修改提交数据的表单。在控制器中，我们可以拦截提交的内容并实现该业务相关的规则。

首先，要修改 searchPage 中的表单，原来的内容如下所示：

```
<form action="/result" method="get" class="col s12">
```

现在，我们将表单改成如下的形式：

```
<form action="/postSearch" method="post" class="col s12">
```

我们还需要在服务器端处理该 POST 请求。在 TweetController 中添加如下的方法：

```
@RequestMapping(value = "/postSearch", method = RequestMethod.POST)
public String postSearch(HttpServletRequest request,
    RedirectAttributes redirectAttributes) {
        String search = request.getParameter("search");
        redirectAttributes.addAttribute("search", search);
        return "redirect:result";
}
```

在这里，有几项比较新鲜的内容：

- 在请求映射的注解中，指定了想要处理的 HTTP 方法，也就是 POST；
- 在方法参数中，直接注入了两个属性，它们是 request 和 RedirectAttributes；
- 检索到请求中 post 提交过来的数据，并将其传递给下一个视图；
- 现在不是直接返回视图的名称，而是重定向到另一个 URL。

RedirectAttributes 是一个 Spring 的模型，专门用于 redirect 场景下传送值。

在 Java Web 应用中，Redirect/Forward 是典型的可选方案。它们都会改变展现给用户的视图，其中的区别在于 Redirect 会发送一个 302 头信息，它会在浏览器内部触发导航，而 Forward 则不会导致 URL 的变化。在 Spring MVC 中，我们可以任意使用这两种方案，只需在方法返回的字符串上添加“redirect:”或“forward:”前缀即可。在这两种场景中，我们所返回的字符串都不会像前面看到的那样解析为视图，而是触发到特定 URL 的导航。

之前的样例有些牵强，在下一章中会进行更加巧妙的处理。如果你在 postSearch 方法上添加一个断点的话，就会发现它会在 post 表单之后进行调用。

那么，该怎样显示错误信息呢？

我们改一下 postSearch 方法：

```
@RequestMapping(value = "/postSearch", method = RequestMethod.POST)
public String postSearch(HttpServletRequest request,
    RedirectAttributes redirectAttributes) {
        String search = request.getParameter("search");
        if (search.toLowerCase().contains("struts")) {
                redirectAttributes.addFlashAttribute("error", "Try
using spring instead!");
                return "redirect:/";
        }
        redirectAttributes.addAttribute("search", search);
        return "redirect:result";
}
```

如果用户的搜索包含“struts”这个术语的话，我们会将其重定向到 searchPage 页面并且使用 flash 属性添加一点错误信息。

这个特殊的属性会在该请求的时间范围内一直存活，直到页面渲染完成才消失。当使用 POST-REDIRECT-GET 模式的时候，它是非常有用的，就像刚刚做的这样。

我们接下来需要在 searchPage 结果页面中展现这个错误信息：

```
<!DOCTYPE html>
<html xmlns:th="http://www.w3.org/1999/xhtml"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
```

```
      layout:decorator="layout/default">
<head lang="en">
    <title>Search</title>
</head>
<body>

<div class="row" layout:fragment="content">

    <h4 class="indigo-text center">Please enter a search term</h4>

    <div class="col s6 offset-s3">
        <div id="errorMessage" class="card-panel red lighten-2"
th:if="${error}">
            <span class="card-title" th:text="${error}"></span>
        </div>
        <form action="/postSearch" method="post" class="col s12">
            <div class="row center">
                <div class="input-field">
                    <i class="mdi-action-search prefix"></i>
                    <input id="search" name="search" type="text"
class="validate"/>
                    <label for="search">Search</label>
                </div>
            </div>
        </form>
    </div>
</div>

</body>
</html>
```

现在，如果用户试图搜索包含“struts2”的 Tweet，将会得到有用且合适的答案（见图 2-13）。

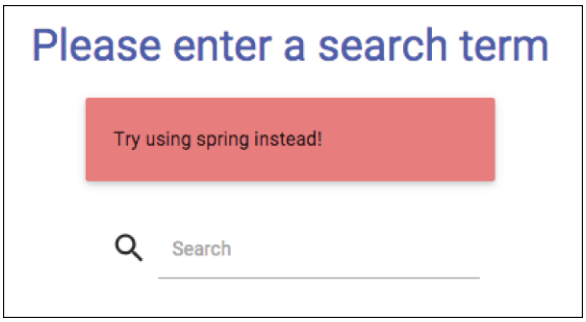

图 2-13

2.10　检查点

在本章结束的时候，你应该有了一个控制器，也就是 TweetController，它会负责处理搜索，同时还有一个没有发生变化的配置类 MasterSpringMvcApplication，它们位于 src/main/java 目录之中（见图 2-14）。

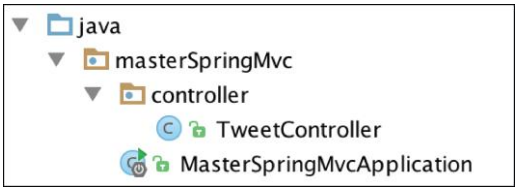

图 2-14

在 src/main/resources 目录中，会有一个默认的布局以及使用该布局的两个页面。

在 application.properties 文件中，我们添加了 Twitter 应用的凭证，还有一个告诉 Spring 禁用模板缓存的属性，从而便于开发的进行，该目录的结构如图 2-15 所示：

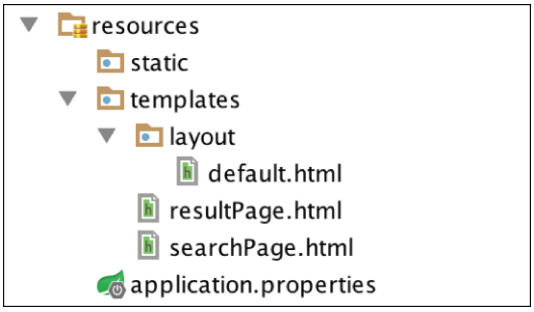

图 2-15

2.11　小结

在本章中，我们学习了如何实现良好的 MVC 架构，看到了 Spring MVC 内部的一些运行机制，并通过非常少量的配置使用了 Spring Social Twitter 的功能。通过使用 WebJars，可以设计非常漂亮的 Web 应用。

在下一章中，我们会要求用户填写其基本信息（profile），这样就能自动获取他们可能感兴趣的 Tweet。这也给了我们一个机会来更深入地学习表单、格式化、校验以及国际化的功能。

第 3 章
处理表单和复杂的 URL 映射

我们的应用看起来已经很漂亮了，如果用户能够提供更多信息的话，那么应用会为他们带来更大的收益。

这样，我们就能提供他们感兴趣的话题了。

在本章中，将会构建一个基本信息（profile）页面。它的特性包括服务端和客户端的校验，并且要上传一个文件作为基本信息的图片。我们会将这些信息存储到用户会话（session）之中，为了保证应用有尽可能多的用户，还会将它翻译为多种语言。最后，将会展现符合用户口味的 Twitter 内容概览。

听起来不错吧？那我们就开始吧，这里有不少的任务需要完成。

3.1 基本信息页——表单

表单是每个 Web 应用的基础。从互联网诞生开始，它就是获取用户输入的主要方式。

第一项任务就是创建一个如图 3-1 所示的基本信息页面。

它允许用户输入一些个人信息，以及个人口味的一个列表。这些口味将用到我们的搜索引擎中。

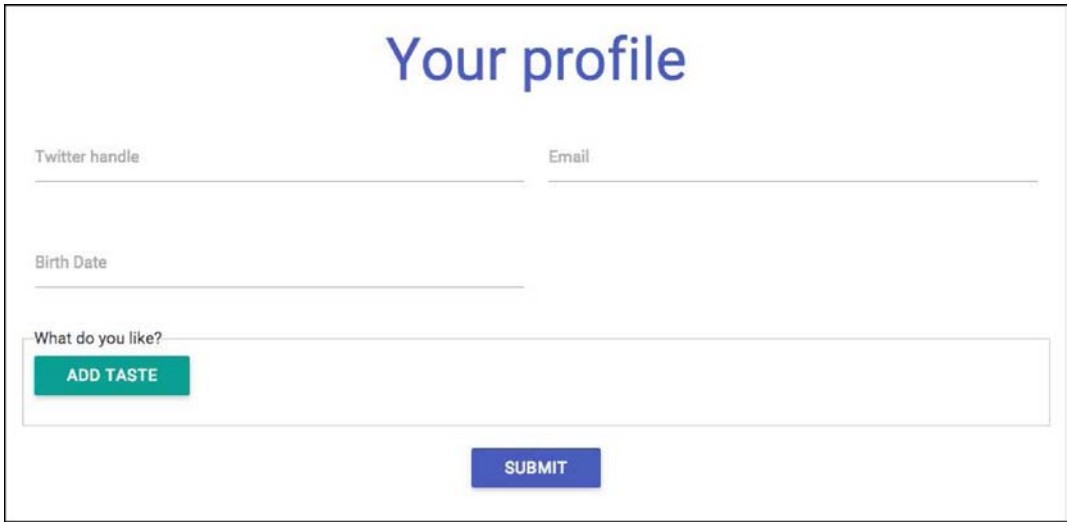

图 3-1

我们在 templates/profile/profilePage.html 中创建一个新的页面：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Your profile</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center">Personal info</h2>

    <form th:action="@{/profile}" method="post" class="col m8 s12
offset-m2">

        <div class="row">
            <div class="input-field col s6">
                <input id="twitterHandle" type="text"/>
                <label for="twitterHandle">Last Name</label>
            </div>
            <div class="input-field col s6">
                <input id="email" type="text"/>
                <label for="email">Email</label>
            </div>
```

```
            </div>
            <div class="row">
                <div class="input-field col s6">
                    <input id="birthDate" type="text"/>
                    <label for="birthDate">Birth Date</label>
                </div>
            </div>
            <div class="row s12">
                <button class="btn waves-effect waves-light" type="submit"
    name="save">Submit
                    <i class="mdi-content-send right"></i>
                </button>
            </div>
        </form>
    </div>
    </body>
    </html>
```

注意，“@{}”语法将会为资源构建完整的路径，它会将服务器上下文路径（在本例中，也就是 localhost:8080）添加到它的参数上。

我们还需要在 profile 包中创建相关的控制器，它的名字是 ProfileController：

```
package masterspringmvc4.profile;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class ProfileController {

    @RequestMapping("/profile")
    public String displayProfile() {
        return "profile/profilePage";
    }
}
```

现在，我们访问 http://localhost:8080 的话，能够看到一个漂亮的表单，但是它什么功能也做不了。这是因为还没有为它的 post URL 映射任何的行为。

我们创建一个数据传输对象（Data Transfer Object，DTO），将它放到与控制器相同的目录中，并将其命名为 ProfileForm。它的作用就是匹配 Web 表单中的域并描述校验规则：

```
package masterSpringMvc.profile;
```

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class ProfileForm {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();

    // getters and setters
}
```

这是一个常规的简单老式 Java 对象（plain old Java object，POJO）。不要忘记为它生成 getter 和 setter 方法，否则的话数据绑定就无法正常运行了。注意，在这里我们有一个口味列表没有进行填充，这个任务我们稍后再完成。

因为我们使用的是 Java 8，用户的出生日期使用的是新的 Java date-time API（JSR 310）。这个 API 要比老的 java.util.Date API 好得多，因为它对人类的日期进行细致的区分，并使用了流畅的 API 和不可变的数据结构。

在示例中，LocalDate 类就是一个简单的日期，没有与其相关联的时间。这与 LocalTime 是有区别的，后者代表了一天中的某个时间，而 LocalDateTime 能包含这两部分的信息，或者是 ZonedDateTime，它还使用了一个时区。

如果你希望学习 Java 8 date-time API 的更多信息，可以参考 Oracle 提供的学习指南，地址是 https://docs.oracle.com/javase/tutorial/datetime/TOC.html。

Google 的建议是为这样的数据对象生成 toString 方法，对于调试这是特别有用的。

为了让 Spring 将表单域绑定到 DTO 上，我们需要在 profilePage 上添加一些元数据：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
```

```
    <title>Your profile</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center">Personal info</h2>

    <form th:action="@{/profile}" th:object="${profileForm}"
method="post" class="col m8 s12 offset-m2">

        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.twitterHandle}"
id="twitterHandle" type="text"/>
                <label for="twitterHandle">Last Name</label>
            </div>
            <div class="input-field col s6">
                <input th:field="${profileForm.email}" id="email"
type="text"/>
                <label for="email">Email</label>
            </div>
        </div>
        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.birthDate}"
id="birthDate" type="text"/>
                <label for="birthDate">Birth Date</label>
            </div>
        </div>
        <div class="row s12">
            <button class="btn waves-effect waves-light" type="submit"
name="save">Submit
                <i class="mdi-content-send right"></i>
            </button>
        </div>
    </form>
</div>
</body>
</html>
```

在这里，有两个需要注意的地方：

- 表单上的 th:object 属性；
- 所有输入域上的 th:field 属性。

在第一项中，会按照类型将一个对象绑定到控制器上。第二项会将实际的输入域绑定到表单 bean 的属性上。

为了让 th:object 能够运行起来，我们需要添加一个 ProfileForm 类型的参数到请求映射方法中：

```
@Controller
public class ProfileController {

    @RequestMapping("/profile")
    public String displayProfile(ProfileForm profileForm) {
        return "profile/profilePage";
    }

    @RequestMapping(value = "/profile", method = RequestMethod.POST)
    public String saveProfile(ProfileForm profileForm) {
        System.out.println("save ok" + profileForm);
        return "redirect:/profile";
    }
}
```

我们还添加了一个对 POST 方法的映射，当表单提交的时候，它就会被调用。此时，如果我们使用一个日期值（如 10/10/1980）提交表单的话，它其实是不能运行的，会提示一个 400 错误，而且没有有用的日志。

Spring Boot 中的日志

借助 Spring Boot，进行日志配置超级简单，只需将 logging.level.{package} = DEBUG 添加到 application.properties 文件中，其中{package}是应用中某个类或包的全限定名称。当然，你可以将 debug 换成自己任意想要的日志级别。你也可以添加传统的日志配置，参考 http://docs.spring.io/spring-boot/docs/current/reference/html/howto-logging.html 了解更多信息。

我们需要调试应用来了解发生了什么状况，那么在 application.properties 中添加如下这行代码：

```
logging.level.org.springframework.web=DEBUG
```

org.springframework.web 包是 Spring MVC 的基础包，这样我们就能看到 Spring Web

所产生的调试信息。如果再次提交表单的话，在日志中将会看到如下的错误：

```
Field error in object 'profileForm' on field 'birthDate':
rejected value [10/10/1980]; codes [typeMismatch.profileForm.
birthDate,typeMismatch.birthDate,typeMismatch.java.time.
LocalDate,typeMismatch]; … nested exception is org.springframework.
core.convert.ConversionFailedException: Failed to convert from type
java.lang.String to type java.time.LocalDate for value '10/10/1980';
nested exception is java.time.format.DateTimeParseException: Text
'10/10/1980' could not be parsed, unparsed text found at index 8]
```

为了理解出现了什么问题，我们需要看一下 Spring 的 DateTimeFormatterRegistrar 类。

在这个类中，可以看到很多用于 JSR 310 的解析器以及输出器（printer）。如果没有指定的话，它们都会使用备用的简短日期格式，如果住在美国的话，就是 MM/dd/yy，否则将是 dd/MM/yy。

在应用启动的时候，这将会让 Spring Boot 创建一个 DateFormatter 类。

在本例中，我们需要完成相同的任务并创建自己的 Formatter，因为使用两位数字来表示年份显得有些怪异。

在 Spring 中，Formatter 类能够用来 print 和 parse 对象。它可以将一个 String 转码为值，也可以将一个值输出为 String。

在 date 包中，我们将会创建一个非常简单的 Formatter，名为 USLocalDateFormatter：

```
public class USLocalDateFormatter implements Formatter<LocalDate> {
    public static final String US_PATTERN = "MM/dd/yyyy";
    public static final String NORMAL_PATTERN = "dd/MM/yyyy";

    @Override public LocalDate parse(String text, Locale locale)
throws ParseException {
        return LocalDate.parse(text, DateTimeFormatter.
ofPattern(getPattern(locale)));
    }
    @Override public String print(LocalDate object, Locale locale) {
        return DateTimeFormatter.ofPattern(getPattern(locale)).
format(object);
    }

    public static String getPattern(Locale locale) {
        return isUnitedStates(locale) ? US_PATTERN : NORMAL_PATTERN;
    }
```

```
    private static boolean isUnitedStates(Locale locale) {
        return Locale.US.getCountry().equals(locale.getCountry());
    }
}
```

这个很简单的类能够让我们根据用户的地域以更加通用的方式来解析日期（使用 4 位数字来表示年份）。

在 config 包中，我们创建名为 WebConfiguration 的新类：

```
package masterSpringMvc.config;

import masterSpringMvc.dates.USLocalDateFormatter;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.config.annotation.
WebMvcConfigurerAdapter;

import java.time.LocalDate;

@Configuration
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Override public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldType(LocalDate.class, new
USLocalDateFormatter());
    }
}
```

这个类扩展了 WebMvcConfigurerAdapter，这是对 Spring MVC 进行自定义配置的一个很便利的类。它提供了很多的扩展点，我们可以重写诸如 addFormatters()这样的方法来访问这些扩展点。

这次再提交表单的话，就不会产生错误了，除非你不按照正确的格式输入日期。

现在，用户看不到要以什么样的方式来输入出生日期，所以，接下来我们在表单上添加上这个信息。

在 ProfileController 中，我们添加 dateFormat 属性：

```
@ModelAttribute("dateFormat")
public String localeFormat(Locale locale) {
```

```
        return USLocalDateFormatter.getPattern(locale);
    }
```

@ModelAttribute 注解允许我们暴露一个属性给 Web 页面，完全类似于我们在上一章中所看到的 model.addAttribute()方法。

现在，我们可以在页面中使用这个信息，只需在日期输入域中添加一个占位符即可：

```
<div class="row">
    <div class="input-field col s6">
        <input th:field="${profileForm.birthDate}" id="birthDate"
type="text" th:placeholder="${dateFormat}"/>
        <label for="birthDate">Birth Date</label>
    </div>
</div>
```

这个信息就会展现给用户了（见图 3-2）。

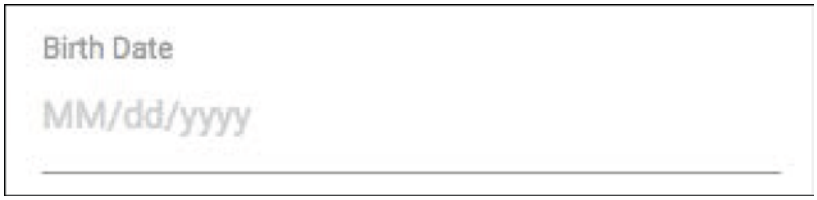

图 3-2

3.2 校验

我们不希望用户输入非法或空的信息，这样的话，就要在 ProfileForm 中增加一些校验逻辑：

```
package masterspringmvc4.profile;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```
public class ProfileForm {
    @Size(min = 2)
    private String twitterHandle;

    @Email
    @NotEmpty
    private String email;

   @NotNull
    private Date birthDate;

    @NotEmpty
    private List<String> tastes = new ArrayList<>();
}
```

可以看到，我们新增了一些校验限制。这些注解来源于 JSR-303 规范，它详细规定了 bean 的校验功能。这个规范的最流行实现是 hibernate-validator，它已经包含在了 Spring Boot 之中。

我们使用了来自 javax.validation.constraints 包中的注解（在 API 中定义的）以及来自 org.hibernate.validator.constraints 包的注解（额外限制）。它们都是可以运行的，我建议你查阅一下 validation-api 和 hibernate-validator 的 jar 包文件中都有哪些可用的注解。

我们还可以查阅 hibernate-validator 的文档来了解它有哪些可用的限制，其地址是 http://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#section-builtin-constraints。

为了让校验功能运行起来，我们还需要添加一些内容。首先，控制器需要声明在表单提交时，它希望得到一个合法的模型。在代表表单的参数上添加 javax.validation.Valid 注解就能实现该功能：

```
@RequestMapping(value = "/profile", method = RequestMethod.POST)
public String saveProfile(@Valid ProfileForm profileForm,
BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "profile/profilePage";
    }

    System.out.println("save ok" + profileForm);
    return "redirect:/profile";
}
```

需要注意的是，如果表单中包含错误的话，我们并没有为用户进行重定向。这样的话，能够在同一个 Web 页面中展现错误信息。

说到这里，我们还需要在 Web 页面上添加一个位置来展现这些错误。

在 profilePage.html 中，表单标签开始的地方添加如下这几行代码：

```
<ul th:if="${#fields.hasErrors('*')}" class="errorlist">
    <li th:each="err : ${#fields.errors('*')}" th:text="${err}">Input
is incorrect</li>
</ul>
```

这将会遍历表单中能够找到的每一项错误并在一个列表中对其进行展现。如果你提交一个空表单的话，将会看到下面这一堆错误（见图 3-3）。

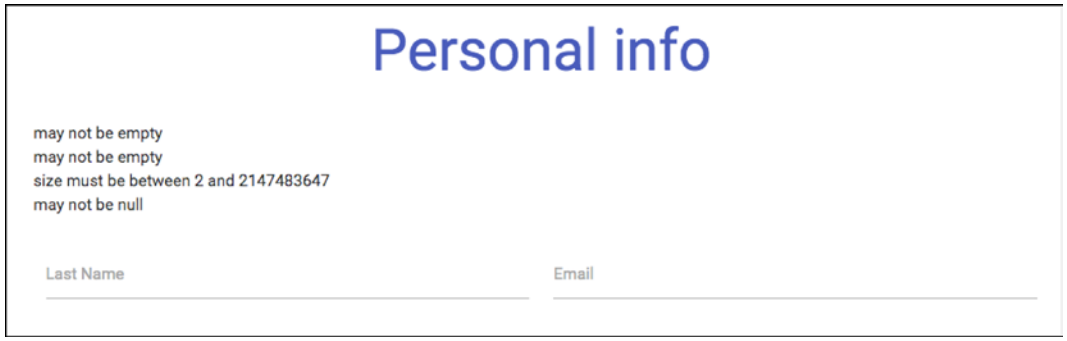

图 3-3

注意，针对用户的@NotEmpty 检查将会阻止表单的提交。实际上，我们现在也没有办法提供这部分信息。

3.2.1 自定义校验信息

目前，这些错误信息对用户来说是没有什么用处的。我们需要做的第一件事情就是将它们与对应的输入域进行关联。按照如下的方式修改 profilePage.html：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Your Profile</title>
</head>
```

```
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center">Personal info</h2>

    <form th:action="@{/profile}" th:object="${profileForm}"
method="post" class="col m8 s12 offset-m2">

        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.twitterHandle}"
id="twitterHandle" type="text" th:errorclass="invalid"/>
                <label for="twitterHandle">Twitter handle</label>

                <div th:errors="*{twitterHandle}" class="red-
text">Error</div>
            </div>
            <div class="input-field col s6">
                <input th:field="${profileForm.email}" id="email"
type="text" th:errorclass="invalid"/>
                <label for="email">Email</label>

                <div th:errors="*{email}" class="red-text">Error</div>
            </div>
        </div>
        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.birthDate}"
id="birthDate" type="text" th:errorclass="invalid" th:placeholder="${
dateFormat}"/>
                <label for="birthDate">Birth Date</label>

                <div th:errors="*{birthDate}" class="red-text">Error</
div>
            </div>
        </div>
        <div class="row s12">
            <button class="btn indigo waves-effect waves-light"
type="submit" name="save">Submit
                <i class="mdi-content-send right"></i>
            </button>
        </div>
    </form>
</div>
```

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

在表单的每个输入域下面，新增了 th:errors 标签，对于每个输入域，还添加了 th:errorclass 属性。如果输入域包含错误的话，相关的 CSS 类将会应用在 DOM 之上。

如图 3-4 所示，校验看起来就会好很多了。

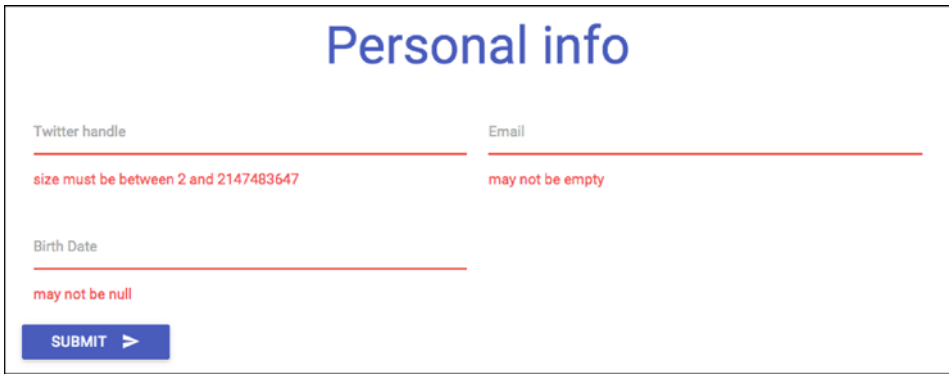

图 3-4

接下来，需要做的事情就是自定义错误信息，让它们能够以更好的方式来反映应用程序中的业务规则。

还记得吧，Spring Boot 会负责为我们创建信息源 bean。信息源默认的位置是 src/main/resources/messages.properties。

创建一个这样的 bundle，并添加如下的文本：

```
Size.profileForm.twitterHandle=Please type in your twitter user name
Email.profileForm.email=Please specify a valid email address
NotEmpty.profileForm.email=Please specify your email address
PastLocalDate.profileForm.birthDate=Please specify a real birth date
NotNull.profileForm.birthDate=Please specify your birth date

typeMismatch.birthDate = Invalid birth date format.
```

在开发期，将信息源配置为每次都重新加载 bundle 是非常便利的。添加如下的属性到 application.properties:

```
spring.messages.cache-seconds=0
```

0 意味着每次都重新加载，而−1 则代表着不进行重新加载。

在 Spring 中负责解析错误信息的类是 DefaultMessageCodesResolver。在进行输入域校验的时候，这个类将会按照如下的顺序来尝试解析信息：

- 编码+“.”+对象名+“.”+输入域；
- 编码+“.”+输入域；
- 编码+“.”+输入域类型；
- 编码。

在前面的规则中，编码部分可能会是两种内容：第一种是注解类型，如 Size 或 Email；第二种是异常码，如 typeMismatch。还记得我们因为数据格式不正确而得到的异常码吗？它相关的异常码的确是 typeMismatch。

有了上面的这些消息，我们接下来让它更为具体。定义默认信息的最佳实践如下所示：

```
Size=the {0} field must be between {2} and {1} characters long
typeMismatch.java.util.Date = Invalid date format.
```

注意，这里的占位符，每个校验错误都有与之关联的一组参数。

声明错误信息的最后一种方式是直接在检验注解中定义错误信息，如下所示：

```
@Size(min = 2, message = "Please specify a valid twitter handle")
private String twitterHandle;
```

但是，这种方式的缺点在于它无法与国际化功能兼容。

3.2.2　用于校验的自定义注解

对于 Java 的日期来说，还有一个名为@Past 的注解，它能够确保某个日期是过去的值。

我们不希望应用程序中的用户将自己伪装成来自未来的人，所以需要校验出生日期。为了实现这一点，在 date 包中，我们将会定义自己的注解：

```
package masterSpringMvc.date;

import javax.validation.Constraint;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import javax.validation.Payload;
```

```
import java.lang.annotation.*;
import java.time.LocalDate;

@Target({ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PastLocalDate.PastValidator.class)
@Documented
public @interface PastLocalDate {
    String message() default "{javax.validation.constraints.Past.
message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    class PastValidator implements ConstraintValidator<PastLocalDate,
LocalDate> {
        public void initialize(PastLocalDate past) {
        }

        public boolean isValid(LocalDate localDate,
ConstraintValidatorContext context) {
            return localDate == null || localDate.isBefore(LocalDate.
now());
        }
    }
}
```

非常简单吧？这段代码将会保证我们的日期的确是一个过去的值。

现在，可以将其添加到基本信息表单的 birthDate 域上：

```
@NotNull
@PastLocalDate
private LocalDate birthDate;
```

3.3 国际化

国际化，通常被称之为 i18n，指的是将应用程序设计为可以翻译成各种语言的过程。

这通常会涉及将翻译文本放到属性 bundle 中，并且要以目标地域作为后缀，例如 messages_en.properties、messages_en_US.properties 和 messages_fr.properties 文件。

属性 bundle 的解析过程是首先尝试最为具体的地域，如果无法找到的话，将会依次使用备用的非具体地域。

例如，对于美国英语来说，如果想要从名为 x 的 bundle 中得到译文，那么应用程序首先会从 x_en_US.properties 文件中进行查找，然后是 x_en.properties 文件，最后会查找 x.properties 文件。

我们所要做的第一件事情就是将错误信息翻译为法语。将已有的 messages.properties 文件重命名为 messages_en.properties。

接下来，创建第二个 bundle，名为 messages_fr.properties：

```
Size.profileForm.twitterHandle=Veuillez entrer votre identifiant
Twitter
Email.profileForm.email=Veuillez spécifier une adresse mail valide
NotEmpty.profileForm.email=Veuillez spécifier votre adresse mail
PastLocalDate.profileForm.birthDate=Veuillez donner votre vraie date
de naissance
NotNull.profileForm.birthDate=Veuillez spécifier votre date de
naissance

typeMismatch.birthDate = Date de naissance invalide.
```

我们曾经在第 1 章中看到，默认情况下，Spring Boot 会使用一个固定的 LocaleResolver 接口。LocaleResolver 非常简单，只有两个方法：

```
public interface LocaleResolver {

    Locale resolveLocale(HttpServletRequest request);
    void setLocale(HttpServletRequest request, HttpServletResponse
response, Locale locale);
}
```

Spring 提供了这个接口的多个实现，如 FixedLocaleResolver。区域解析器非常简单，我们可以通过一个属性来配置应用程序的地域，而且一旦定义之后，就不能进行修改了。要配置我们应用的地域，只需添加如下的属性到 application.properties 文件中：

```
spring.mvc.locale=fr
```

这样，就会添加我们的法语检验信息。

如果我们看一下 Spring MVC 所提供的不同 LocaleResolver 接口实现，会发现它们如下所示。

- FixedLocaleResolver：使用配置中固定的地域，一旦确定之后，不能发生变化。
- CookieLocaleResolver：允许在 Cookie 中检索和保存地域信息。
- AcceptHeaderLocaleResolver：根据用户浏览器所发送的 HTTP 头信息来查找地域。
- SessionLocaleResolver：在 HTTP 会话中查找和存储地域信息。

这些实现类能够涵盖很多的用户场景，但是在更为复杂的应用程序中，我们可以直接实现一个 LocaleResolver，从而允许更复杂的逻辑，例如先从数据库中获取地域信息，如果无法匹配的话，再使用浏览器中的地域。

3.3.1 修改地域

在应用中，地域是与用户相关联的。我们会将用户的基本信息存储在会话中。

我们允许用户通过一个小菜单，修改站点的语言，这样的话，就应该使用 SessionLocaleResolver。我们再次编辑一下 WebConfiguration：

```
package masterSpringMvc.config;

import masterSpringMvc.date.USLocalDateFormatter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.config.annotation.
InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.
WebMvcConfigurerAdapter;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;

import java.time.LocalDate;

@Configuration
public class WebConfiguration extends WebMvcConfigurerAdapter {
```

```
    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldType(LocalDate.class, new
USLocalDateFormatter());
    }

    @Bean
    public LocaleResolver localeResolver() {
        return new SessionLocaleResolver();
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor localeChangeInterceptor = new
LocaleChangeInterceptor();
        localeChangeInterceptor.setParamName("lang");
        return localeChangeInterceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

我们声明了一个 LocaleChangeInterceptor bean 作为 Spring MVC 的拦截器。它将会拦截所有发往控制器的请求，并检查 lang 查询参数。例如，导航至 http://localhost:8080/profile?lang=fr 将会导致地域的修改。

Spring MVC 拦截器可以类比为 Web 应用中的 Servlet 过滤器。拦截器允许进行自定义的预处理、跳过处理器的执行以及执行自定义的事后处理。过滤器会更加强大，例如，对于链中传递下来的请求和响应对象，它们支持进行一些交互操作。过滤器是在 web.xml 中配置的，而拦截器是在 Spring 应用上下文中，以 bean 的方式配置的。

现在，我们可以通过输入正确的 URL 来自行修改地域，但是如果能够添加一个导航栏，允许用户修改语言的话那就更好了。修改默认的布局（templates/layout/default.html），添加一个下拉按钮：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8"/>
    <meta name="viewport" content="width=device-width, initial-
scale=1, maximum-scale=1.0, user-scalable=no"/>
    <title>Default title</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
type="text/css" rel="stylesheet" media="screen,projection"/>
</head>
<body>

<ul id="lang-dropdown" class="dropdown-content">
    <li><a href="?lang=en_US">English</a></li>
    <li><a href="?lang=fr">French</a></li>
</ul>
<nav>
    <div class="nav-wrapper indigo">
        <ul class="right">
            <li><a class="dropdown-button" href="#!" dataactivates="
lang-dropdown"><i class="mdi-action-language right"></i>
Lang</a></li>
        </ul>
    </div>
</nav>

<section layout:fragment="content">
    <p>Page content goes here</p>
</section>

<script src="/webjars/jquery/2.1.4/jquery.js"></script>
<script src="/webjars/materializecss/0.96.0/js/materialize.js"></
script>
<script type="text/javascript">
    $(".dropdown-button").dropdown();
</script>
</body>
</html>
```

这样的话，就允许用户在我们支持的两种语言间进行选择（见图 3-5）。

图 3-5

3.3.2　翻译应用的文本

为了实现支持双语的应用程序，所要做的最后一件事情就是翻译应用中的标题和标签。要实现这一点，我们需要编辑 Web 页面，例如在 profilePage.html 中采用如下的方式来使用 `th:text` 属性：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Your profile</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center" th:text="#{profile.title}">Personal
info</h2>

    <form th:action="@{/profile}" th:object="${profileForm}"
method="post" class="col m8 s12 offset-m2">
        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.twitterHandle}"
id="twitterHandle" type="text" th:errorclass="invalid"/>
                <label for="twitterHandle" th:text="#{twitter.
handle}">Twitter handle</label>

                <div th:errors="*{twitterHandle}" class="red-
text">Error</div>
            </div>
            <div class="input-field col s6">
                <input th:field="${profileForm.email}" id="email"
```

```
type="text" th:errorclass="invalid"/>
                <label for="email" th:text="#{email}">Email</label>

                <div th:errors="*{email}" class="red-text">Error</div>
            </div>
        </div>
        <div class="row">
            <div class="input-field col s6">
                <input th:field="${profileForm.birthDate}"
id="birthDate" type="text" th:errorclass="invalid"/>
                <label for="birthDate" th:text="#{birthdate}" th:place
holder="${dateFormat}">Birth Date</label>

                <div th:errors="*{birthDate}" class="red-text">Error</
div>
            </div>
        </div>
        <div class="row s12 center">
            <button class="btn indigo waves-effect waves-light"
type="submit" name="save" th:text="#{submit}">Submit
                <i class="mdi-content-send right"></i>
            </button>
        </div>
    </form>
</div>
</body>
</html>
```

带有表达式的 th:text 属性将会替换 HTML 元素中的内容。在这里，我们使用了“#{}”语法，它表明我们想要展现的信息来源于属性源文件，例如 messages.properties。

在英文对应的 bundle 中，添加对应的译文：

```
NotEmpty.profileForm.tastes=Please enter at least one thing
profile.title=Your profile
twitter.handle=Twitter handle
email=Email
birthdate=Birth Date
tastes.legend=What do you like?
remove=Remove
taste.placeholder=Enter a keyword
add.taste=Add taste
submit=Submit
```

法文的 bundle 如下所示：

```
NotEmpty.profileForm.tastes=Veuillez saisir au moins une chose
profile.title=Votre profil
twitter.handle=Pseudo twitter
email=Email
birthdate=Date de naissance
tastes.legend=Quels sont vos goûts ?
remove=Supprimer
taste.placeholder=Entrez un mot-clé
add.taste=Ajouter un centre d'intérêt
submit=Envoyer
```

其中有些翻译的文本还没有用到，但是我们在稍后就会使用它们。非常好！Twitter 搜索应用已经打开了法国市场的大门。

3.3.3　表单中的列表

现在，我们希望用户能够输入一个“口味”的列表，实际上，它就是用来搜索 Tweet 的关键字列表。

这里会展现一个新的按钮，允许用户输入新的关键字并将其添加到列表中。如图 3-6 所示，列表中的每一项都是可编辑的文本域，通过移除按钮，可以将其移除掉。

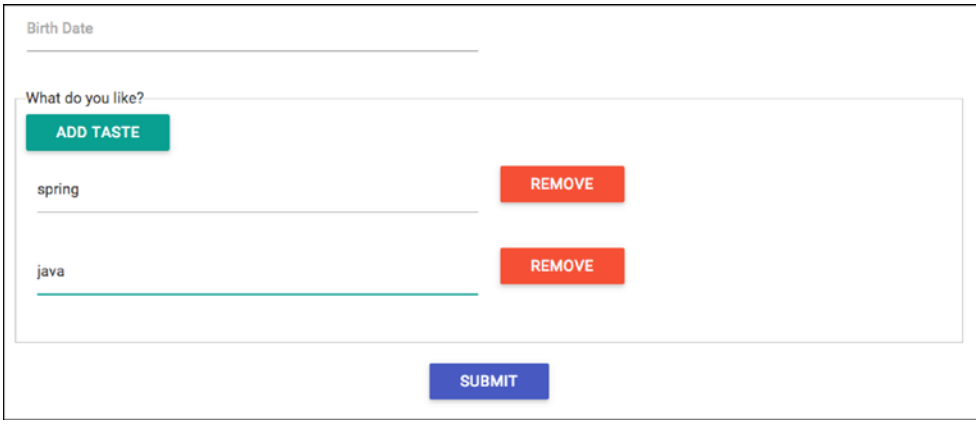

图 3-6

在有些框架中，处理表单里面的列表会比较麻烦。但是，通过使用 Spring MVC 和 Thymeleaf，只要我们理解了其中的理念，这项任务就会变得非常简单。

在 profilePage.html 文件中添加如下的代码行，它的位置要位于出生日期行后面，提交

按钮的前面：

```
<fieldset class="row">
    <legend th:text="#{tastes.legend}">What do you like?</legend>
    <button class="btn teal" type="submit" name="addTaste"
th:text="#{add.taste}">Add taste
        <i class="mdi-content-add left"></i>
    </button>

    <div th:errors="*{tastes}" class="red-text">Error</div>

    <div class="row" th:each="row,rowStat : *{tastes}">
        <div class="col s6">
            <input type="text" th:field="*{tastes[__${rowStat.
index}__]}" th:placeholder="#{taste.placeholder}"/>
        </div>

        <div class="col s6">
            <button class="btn red" type="submit" name="removeTaste"
th:value="${rowStat.index}" th:text="#{remove}">Remove
                <i class="mdi-action-delete right waves-effect"></i>
            </button>
        </div>
    </div>
</fieldset>
```

这个代码片段的目的在于遍历 `LoginForm` 中的 `tastes` 变量。这可以通过 `th:each` 属性来实现，它看上去非常类似于 Java 中的 `for...in` 循环。

与我们之前看到的搜索结果循环相比，迭代存储在两个变量中，而不是一个变量。第一个变量将会包含数据中的每一行，而 `rowStat` 变量将会包含当前迭代状态的额外信息。

在这段代码中，最奇怪的就是下面这一部分：

```
th:field="*{tastes[__${rowStat.index}__]}"
```

这是一个很复杂的语法，你可以自行想出一些更为简单的形式，例如：

```
th:field="*{tastes[rowStat.index]}"
```

但是，这并不能运行。这里的${rowStat.index}变量代表了迭代循环的当前索引，它需要在其他表达式执行之前计算出来。为了实现这一点，需要预处理功能。

在双下划线之间的表达式将会进行预处理，这意味着它能够在正常的处理流程之前进行处理，从而允许执行两次计算。

在我们的表单中，有了两个新的提交按钮，它们分别都有自己的名称。我们之前的全局提交按钮名为 save，两个新的提交按钮名为 addTaste 和 removeTaste。

在控制器中，这能够很容易地区分来自表单的不同行为，接下来，在 ProfileController 中添加两个新的行为：

```
@Controller
public class ProfileController {

    @ModelAttribute("dateFormat")
    public String localeFormat(Locale locale) {
        return USLocalDateFormatter.getPattern(locale);
    }

    @RequestMapping("/profile")
    public String displayProfile(ProfileForm profileForm) {
        return "profile/profilePage";
    }
    @RequestMapping(value = "/profile", params = {"save"}, method =
RequestMethod.POST)
    public String saveProfile(@Valid ProfileForm profileForm,
BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return "profile/profilePage";
        }
        System.out.println("save ok" + profileForm);
        return "redirect:/profile";
    }

    @RequestMapping(value = "/profile", params = {"addTaste"})
    public String addRow(ProfileForm profileForm) {
        profileForm.getTastes().add(null);
        return "profile/profilePage";
    }

    @RequestMapping(value = "/profile", params = {"removeTaste"})
    public String removeRow(ProfileForm profileForm,
HttpServletRequest req) {
        Integer rowId = Integer.valueOf(req.
getParameter("removeTaste"));
```

```
            profileForm.getTastes().remove(rowId.intValue());
            return "profile/profilePage";
        }
    }
```

我们在每个 POST 行为上都添加了一个 `param` 参数，用于对它们进行区分。我们之前的那个操作绑定了 save 参数。

当我们点击某个按钮的时候，这个按钮的名称将会自动添加到表单数据上，由浏览器进行发送。注意，我们为移除按钮指定了一个特殊的值：th:value="${rowStat.index}"。这个属性将会指定相关的参数要带有什么样的值。如果这个属性不存在的话，将会发送空值。这意味着当我们点击移除按钮的时候，会有一个 `removeTaste` 参数添加到 POST 请求中，包含了要移除行的索引。在控制器中，通过如下的代码，可以重新得到这个值：

```
Integer rowId = Integer.valueOf(req.getParameter("removeTaste"));
```

这种方法的唯一缺点在于，每次我们点击按钮的时候，都会发送整个表单的数据，即便有的数据并不是强制要求的。我们的表单比较小，因此这种权衡也是可以接受的。

到此为止，表单已经完成了，我们可以添加一项或更多的口味。

3.4 客户端校验

再补充一点，通过使用 HTML 5 的表单检验规范，如今实现客户端的校验已经非常容易了。如果你的目标浏览器是 Internet Explorer 10 及以上的话，添加客户端校验只需要指定正确的输入域类型，不要再将 type 属性设置为 text。

通过添加客户端校验，我们就可以预先校验表单，避免已知的不正确请求对服务器形成过大的负载。关于客户端校验规范的更多信息可以参考 http://caniuse.com/#search=validation。

我们可以修改输入域来启用简单的客户端校验。之前的输入域如下面的代码所示：

```
<input th:field="${profileForm.twitterHandle}" id="twitterHandle"
type="text" th:errorclass="invalid"/>
<input th:field="${profileForm.email}" id="email" type="text"
th:errorclass="invalid"/>
<input th:field="${profileForm.birthDate}" id="birthDate" type="text"
th:errorclass="invalid"/>
<input type="text" th:field="*{tastes[__${rowStat.index}__]}"
th:placeholder="#{taste.placeholder}"/>
```

将会变成如下所示：

```
<input th:field="${profileForm.twitterHandle}" id="twitterHandle"
type="text" required="required" th:errorclass="invalid"/>
<input th:field="${profileForm.email}" id="email" type="email"
required="required" th:errorclass="invalid"/>
<input th:field="${profileForm.birthDate}" id="birthDate" type="text"
required="required" th:errorclass="invalid"/>
<input type="text" required="required" th:field="*{tastes[__${rowStat.
index}__]}" th:placeholder="#{taste.placeholder}"/>
```

通过这种方法，浏览器能够探测到表单的提交并根据对应的类型校验每个属性。其中，required 属性将会强制用户输入非空的值，email 类型会为对应输入域进行基本的 E-mail 格式校验（见图 3-7）。

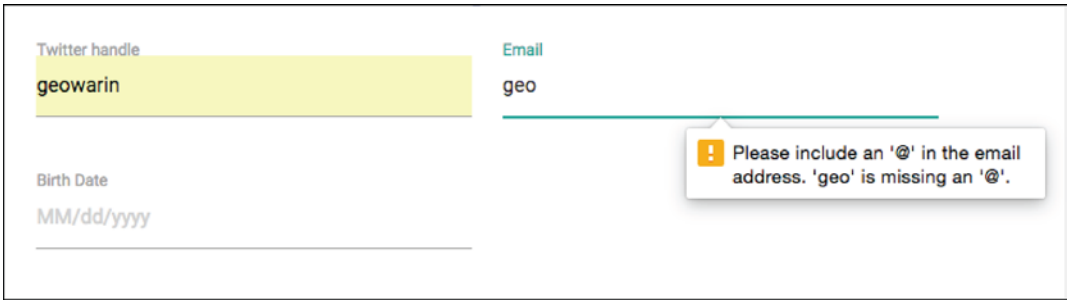

图 3-7

还存在其他类型的校验，请参考 http://www.the-art-of-web.com/html/html5-form-validation。

这种方式的缺点在于添加和移除口味的按钮也会触发校验。为了修正这个问题，我们需要在默认的布局底部包含一个脚本，就放在 jQuery 声明的后面。

但是，最好只在基本信息页面包含这个脚本。我们只需在 layout/default.html 页面上添加一个新的片段，将其放在 body 的结束标签前面：

```
<script type="text/javascript" layout:fragment="script">
</script>
```

这样，在每个页面上，我们就能根据需要添加额外的脚本。

现在，我们可以在基本信息页面添加如下的脚本，将其放在 body 的闭合标签之前：

```
<script layout:fragment="script">
    $('button').bind('click', function(e) {
```

```
        if (e.currentTarget.name === 'save') {
            $(e.currentTarget.form).removeAttr('novalidate');
        } else {
            $(e.currentTarget.form).attr('novalidate', 'novalidate');
        }
    });
</script>
```

如果表单中存在 novalidate 属性的话，表单校验将不会触发。按照这一小段脚本的逻辑，如果操作的输入域名为 save 的话，将会动态移除 novalidate 属性，否则的话，novalidate 属性将会一直保存。因此，只有 save 按钮才会触发检验功能。

3.5 检查点

在进入下一章之前，我们来检查一下所有内容是否都已经就位。

如图 3-8 所示，在 Java 源码中，应该包含如下的内容。

- 新的控制器 ProfileController。
- 两个与日期相关的新类：日期 formatter 和校验 LocalDate 的注解。
- 新的 WebConfiguration，用于存放自定义的 Spring MVC 配置。

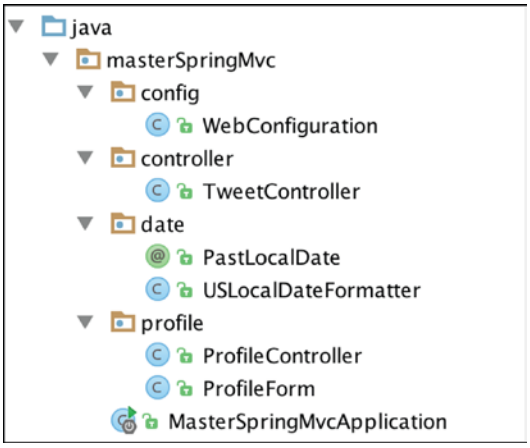

图 3-8

在资源中，profile 目录中应该包含一个新的模板，并且还会有两个新的信息 bundle 文件（见图 3-9）。

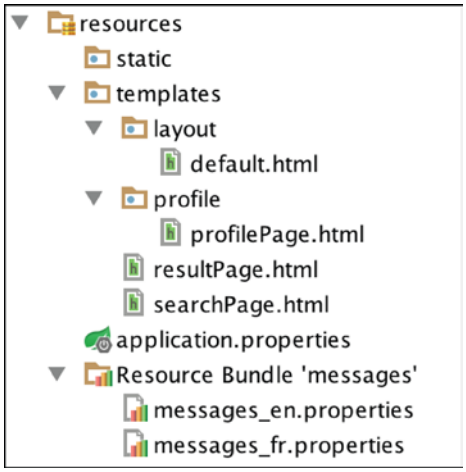

图 3-9

3.6　小结

在本章中，学习到了如何构建完整的表单。我们使用 Java 8 的日期创建了模型，学习了如何格式化来自用户的信息并进行相应的展现。

我们通过校验器注解确保表单填充了合法的信息，这其中也包括自定义的注解。同时，我们使用了一些很简便的客户端检验，确保明显错误的信息根本就不会发送到服务端。

最后，我们将整个应用翻译成了英文和法文，还包括日期的格式。

在下一章中，我们将会构建一项功能，允许用户上传图片，同时还会学习 Spring MVC 应用中错误处理的更多知识。

第 4 章
文件上传与错误处理

在本章中，我们将会允许用户上传基本信息图片，并且还会看到如何在 Spring MVC 中处理错误。

4.1 上传文件

现在，我们希望用户能够上传基本信息图片。稍后我们会将其放到基本信息页面，但现在为了让事情更加简单，在模板目录创建一个新的页面 profile/uploadPage.html：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Profile Picture Upload</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center">Upload</h2>

    <form th:action="@{/upload}" method="post" enctype="multipart/
form-data" class="col m8 s12 offset-m2">

        <div class="input-field col s6">
            <input type="file" id="file" name="file"/>
        </div>
        <div class="col s6 center">
            <button class="btn indigo waves-effect waves-light"
```

```
type="submit" name="save" th:text="#{submit}">Submit
                <i class="mdi-content-send right"></i>
            </button>
        </div>
    </form>
</div>
</body>
</html>
```

除了表单中的 enctype 属性以外，并没有太多值得关注的。文件将会通过 POST 方法发送到 upload URL 上。我们将会创建一个新的控制器，它会位于 profile 包中，与 Profile Controller 位于同级目录下：

```
package masterSpringMvc.profile;

import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

@Controller
public class PictureUploadController {
    public static final Resource PICTURES_DIR = new
FileSystemResource("./pictures");

    @RequestMapping("upload")
    public String uploadPage() {
        return "profile/uploadPage";
    }

    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public String onUpload(MultipartFile file) throws IOException {
        String filename = file.getOriginalFilename();
        File tempFile = File.createTempFile("pic",
getFileExtension(filename), PICTURES_DIR.getFile());
```

```
            try (InputStream in = file.getInputStream();
                 OutputStream out = new FileOutputStream(tempFile)) {
                IOUtils.copy(in, out);
            }

            return "profile/uploadPage";
        }

        private static String getFileExtension(String name) {
            return name.substring(name.lastIndexOf("."));
        }
    }
```

这段代码做的第一件事情是在 pictures 目录下创建一个临时文件，这个目录位于项目的根文件夹下，所以要确保该目录是存在的。在 Java 中，临时文件只是用来获取文件系统中唯一的文件标识符的，用户可以自行决定是否要删除它。

在项目的根目录下创建 pictures 目录，并添加名为“.gitkeep”的空文件，从而保证能够将其提交到 Git 上。

Git 中的空目录

Git 是基于文件的，不能提交空目录。常见的变通方案就是在目录中提交一个空文件，如“.gitkeep”，从而强制 Git 对其进行版本控制。

用户提交的文件将会以 MultipartFile 接口的形式注入到控制器中，这个接口提供了多个方法，用来获取文件的名称、大小及其内容。

这个方法中，最令我们感兴趣的应该就是 `getInputStream()`，借助 IOUtils.copy 方法，将这个流复制到 fileOutputStream 上。将输入流复制到输出流的代码是很让人感觉乏味的，所以将 Apache Utils 添加到类路径下，我们就可以非常便捷了（它是 tomcat-embedded-core.jar 文件的一部分）。

这里使用了 Spring 和 Java7 NIO 一些很酷的特性：

- 用字符串表示的 Resource 类是一种工具类，它代表了资源的抽象，而这些资源可以按照不同的方式来获取；
- `try...with` 代码块将会自动关闭流，即便出现异常也会如此，从而移除了 `finally` 这样的样板式代码。

通过前面的代码，用户上传的所有文件都会复制到 pictures 目录中。

在 Spring Boot 中有一些属性可以用来自定义文件上传功能，请参考 `MultipartProperties` 类。

其中最有意思的是：

- `multipart.maxFileSize`：这定义了所允许上传文件的最大容量。尝试上传更大的文件将会出现 MultipartException，其默认值是 1Mb；
- `multipart.maxRequestSize`：这定义了整个 multipart 请求的最大容量，默认值是 10MB。

对于我们的应用来说，默认值就足够了。在上传几个文件之后，我们的 picture 目录将会如图 4-1 所示。

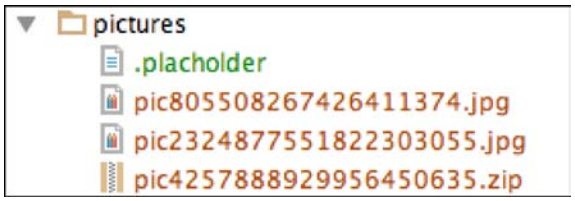

图 4-1

稍等！有人居然上传了 ZIP 文件！真的难以置信，我们最好在控制器中添加一些检查，来确保上传的文件是真正的图片：

```
package masterSpringMvc.profile;

import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;
import java.io.*;

@Controller
public class PictureUploadController {
    public static final Resource PICTURES_DIR = new
FileSystemResource("./pictures");

    @RequestMapping("upload")
```

```
    public String uploadPage() {
        return "profile/uploadPage";
    }

    @RequestMapping(value = "/upload", method = RequestMethod.POST)
    public String onUpload(MultipartFile file, RedirectAttributes
redirectAttrs) throws IOException {

        if (file.isEmpty() || !isImage(file)) {
            redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
            return "redirect:/upload";
        }

        copyFileToPictures(file);

        return "profile/uploadPage";
    }

    private Resource copyFileToPictures(MultipartFile file) throws
IOException {
        String fileExtension = getFileExtension(file.
getOriginalFilename());
        File tempFile = File.createTempFile("pic", fileExtension,
PICTURES_DIR.getFile());
        try (InputStream in = file.getInputStream();
             OutputStream out = new FileOutputStream(tempFile)) {

            IOUtils.copy(in, out);
        }
        return new FileSystemResource(tempFile);
    }

    private boolean isImage(MultipartFile file) {
        return file.getContentType().startsWith("image");
    }
    private static String getFileExtension(String name) {
        return name.substring(name.lastIndexOf("."));
    }
}
```

非常简单！`getContentType()`方法将会返回文件的多用途 Internet 邮件扩展（Multipurpose Internet Mail Extensions，MIME）类型。它将会是 image/png、image/jpg 等。所以，只需要检查 MIME 类型是否以“image”开头即可。

我们在表单上添加了错误信息，所以需要在 Web 页面上新增一个位置来显示它。在 uploadPage 的标题下面添加如下的代码：

```
<div class="col s12 center red-text" th:text="${error}"
th:if="${error}">
    Error during upload
</div>
```

如果你下次再尝试上传 ZIP 文件的话，就会看到一个错误！如图 4-2 所示。

图 4-2

4.1.1　将图片写入到响应中

上传的文件并不是以静态目录的方式来使用的，我们需要采取一些特殊的操作，从而能够在 Web 页面中展现它们。

我们在上传页面的表单上方新增如下的代码行：

```
<div class="col m8 s12 offset-m2">
    <img th:src="@{/uploadedPicture}" width="100" height="100"/>
</div>
```

这将会试图从控制器中获取图片，接下来在 `PictureUploadController` 类中添加对应的方法：

```
@RequestMapping(value = "/uploadedPicture")
public void getUploadedPicture(HttpServletResponse response) throws
IOException {
    ClassPathResource classPathResource = new ClassPathResource("/
images/anonymous.png");
    response.setHeader("Content-Type", URLConnection.guessContentTypeF
romName(classPathResource.getFilename()));
    IOUtils.copy(classPathResource.getInputStream(), response.
getOutputStream());
}
```

这段代码会直接将 src/main/resources/images/anonymous.png 目录下的图片写到响应中，非常棒！

如果我们再次访问页面的话，将会看到如图 4-3 所示的页面。

图 4-3

我是在 iconmonstr（http://iconmonstr.com/user-icon）上找到这个匿名用户的图片的，并下载了 128 像素 × 128 像素的 PNG 文件。

4.1.2 管理上传属性

此时，我们最好通过 application.properties 文件来配置上传目录以及匿名用户图片的路径。

首先，在新创建的 config 包中，我们定义一个 `PicturesUploadProperties` 类：

```
package masterSpringMvc.config;

import org.springframework.boot.context.properties.
ConfigurationProperties;
import org.springframework.core.io.DefaultResourceLoader;
import org.springframework.core.io.Resource;

import java.io.IOException;
@ConfigurationProperties(prefix = "upload.pictures")
public class PictureUploadProperties {
    private Resource uploadPath;
    private Resource anonymousPicture;

    public Resource getAnonymousPicture() {
        return anonymousPicture;
```

```
    }

    public void setAnonymousPicture(String anonymousPicture) {
        this.anonymousPicture = new DefaultResourceLoader().
getResource(anonymousPicture);
    }

    public Resource getUploadPath() {
        return uploadPath;
    }

    public void setUploadPath(String uploadPath) {
        this.uploadPath = new DefaultResourceLoader().
getResource(uploadPath);
    }
}
```

在这个类中，我们使用了 Spring Boot 的 ConfigurationProperties，这将会告诉 Spring Boot 以一种类型安全的方式，自动映射类路径下所发现的属性（默认情况下，位于 application.properties 文件中）。

注意，我们定义的 `setter` 方法接受“String”类型作为参数，这里最为有用的是我们可以让 getter 返回任意的类型。

我们现在需要将 `PicturesUploadProperties` 添加到配置中：

```
@SpringBootApplication
@EnableConfigurationProperties({PictureUploadProperties.class})
public class MasterSpringMvc4Application extends
WebMvcConfigurerAdapter {
  // code omitted
}
```

现在，可以在 application.properties 文件中添加如下的属性值：

```
upload.pictures.uploadPath=file:./pictures
upload.pictures.anonymousPicture=classpath:/images/anonymous.png
```

因为我们使用的是 `DefaultResourceLoader` 类，因此可以使用像“file:”或“classpath:”这样的前缀来指定查找资源的位置。

这等价于分别创建 `FileSystemResource` 和 `ClassPathResource` 类。

这种方式对于代码的文档化也是有好处的，我们可以很容易地看到图片目录是位于应用的根目录下，而匿名用户的图片在位于类路径下。

这样就可以了！我们现在就能在控制器中使用属性了，如下是 PictureUploadController 类中相关的部分：

```
package masterSpringMvc.profile;

import masterSpringMvc.config.PictureUploadProperties;
import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.net.URLConnection;

@Controller
public class PictureUploadController {
    private final Resource picturesDir;
    private final Resource anonymousPicture;

    @Autowired
    public PictureUploadController(PictureUploadProperties
uploadProperties) {
        picturesDir = uploadProperties.getUploadPath();
        anonymousPicture = uploadProperties.getAnonymousPicture();
    }
    @RequestMapping(value = "/uploadedPicture")
    public void getUploadedPicture(HttpServletResponse response)
throws IOException {
        response.setHeader("Content-Type", URLConnection.guessContentT
ypeFromName(anonymousPicture.getFilename()));
        IOUtils.copy(anonymousPicture.getInputStream(), response.
getOutputStream());
}

  private Resource copyFileToPictures(MultipartFile file) throws
```

```
IOException {
       String fileExtension = getFileExtension(file.
getOriginalFilename());
       File tempFile = File.createTempFile("pic", fileExtension,
picturesDir.getFile());
       try (InputStream in = file.getInputStream();
            OutputStream out = new FileOutputStream(tempFile)) {

            IOUtils.copy(in, out);
        }
        return new FileSystemResource(tempFile);
   }
// The rest of the code remains the same
}
```

到此为止，如果你重新启动应用的话，会发现结果并没有什么变化。匿名用户的图片依然会展现，而我们用户所上传的图片将会位于项目根目录的 pictures 文件夹中。

4.1.3　展现上传的图片

如果我们能够展现用户所上传的图片，那就非常棒了，对吧？为了实现这一点，需要在 PictureUploadController 中添加一个模型属性：

```
@ModelAttribute("picturePath")
public Resource picturePath() {
  return anonymousPicture;
}
```

现在，当我们获取上传的图片时，就可以将其注入进来并检索它的值：

```
@RequestMapping(value = "/uploadedPicture")
public void getUploadedPicture(HttpServletResponse response, @
ModelAttribute("picturePath") Path picturePath) throws IOException {
    response.setHeader("Content-Type", URLConnection.guessContentTypeF
romName(picturePath.toString()));
    Files.copy(picturePath, response.getOutputStream());
}
```

`@ModelAttribute` 是一个非常便利的注解，我们可以通过带有该注解的方法来创建模型属性。通过使用相同的注解，它们就可以注入到控制器方法之中。借助上面的这些代码，只要不重定向页面，就可以始终在模型中得到 `picturePath` 参数。它的默认值是在属性中所定义的匿名图片。

当文件上传之后，我们需要更新这个值，修改一下 onUpload 方法：

```
@RequestMapping(value = "/upload", method = RequestMethod.POST)
public String onUpload(MultipartFile file, RedirectAttributes
redirectAttrs, Model model) throws IOException {

    if (file.isEmpty() || !isImage(file)) {
        redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
        return "redirect:/upload";
    }

    Resource picturePath = copyFileToPictures(file);
    model.addAttribute("picturePath", picturePath);

    return "profile/uploadPage";
}
```

通过将模型注入进来，我们就能在上传完成之后，更新 picturePath 参数的值。

现在，问题在于我们的两个方法，即 onUpload 和 getUploadedPicture 是在不同的请求中调用的。而模型属性在不同的请求间会进行重置。

这样的话，我们就需要将 picturePath 参数定义为会话属性，要实现这一点，只需在控制器类上添加另外一个注解即可：

```
@Controller
@SessionAttributes("picturePath")
public class PictureUploadController {
}
```

为了处理一个简单的会话属性，我们添加了好几个注解。现在，能够得到如图 4-4 所示的结果。

图 4-4

这种方式让代码的组合变得非常容易，我们也没有直接使用 HttpServletRequest 或 HttpSession，另外，我们的对象在编码时输入起来会非常简便。

4.1.4　处理文件上传的错误

细心的读者可能已经观察到，我们的代码会抛出两种类型的错误。

- IOException：在将文件写入到磁盘的时候，如果出现问题，将会抛出该错误。
- MultipartException：在上传文件的时候，如果出现错误，将会抛出该异常。例如，超出了最大容量的限制。

这让我们有机会看一下在 Spring 中处理错误的两种方式：

- 在控制器的本地方法中使用`@ExceptionHandler` 注解；
- 在 Servlet 容器级别定义全局的异常处理器。

我们在 PictureUploadController 中使用`@ExceptionHandler` 注解来处理 `IOException`，添加如下的方法：

```
@ExceptionHandler(IOException.class)
public ModelAndView handleIOException(IOException exception) {
    ModelAndView modelAndView = new ModelAndView("profile/
uploadPage");
    modelAndView.addObject("error", exception.getMessage());
    return modelAndView;
}
```

这是一种简单却强大的方式，在我们的控制器中，每当抛出 IOException 的时候，这个方法就会被调用。

让 Java IO 的代码抛出异常可能比较麻烦，为了测试异常处理器，在测试阶段我们将 onUpload 的方法体替换如下的样子：

```
@RequestMapping(value = "/upload", method = RequestMethod.POST)
public String onUpload(MultipartFile file, RedirectAttributes
redirectAttrs, Model model) throws IOException {
    throw new IOException("Some message");
}
```

在修改之后，如果我们再尝试上传图片的话，那么将会在上传页面看到异常的错误信息展现了出来（见图 4-5）。

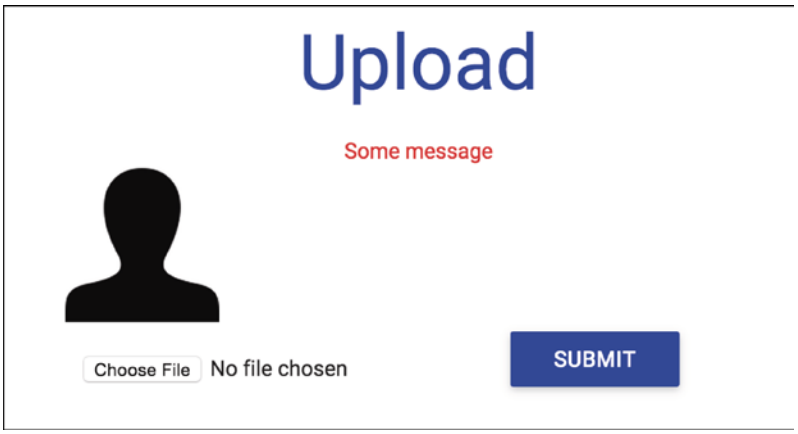

图 4-5

现在，来处理 MultipartException。它需要在 Servlet 容器级别（也就是 Tomcat 级别）来进行，因为这个异常不会由我们的控制器直接抛出。

我们需要添加一个新的 `EmbeddedServletContainerCustomizer` 到配置之中，在 `WebConfiguration` 类中添加如下的方法：

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    EmbeddedServletContainerCustomizer
embeddedServletContainerCustomizer = new
EmbeddedServletContainerCustomizer() {
        @Override
        public void customize(ConfigurableEmbeddedServletContainer
container) {
            container.addErrorPages(new ErrorPage(MultipartException.
class, "/uploadError"));
        }
    };
    return embeddedServletContainerCustomizer;
}
```

这有点冗长，注意 `EmbeddedServletContainerCustomizer` 是只包含一个方法的接口，它可以替换为 lambda 表达式：

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    EmbeddedServletContainerCustomizer
embeddedServletContainerCustomizer
```

```
            = container -> container.addErrorPages(new
ErrorPage(MultipartException.class, "/uploadError"));
    return embeddedServletContainerCustomizer;
}
```

因此，我们只需这样编写：

```
@Bean
public EmbeddedServletContainerCustomizer containerCustomizer() {
    return container -> container.addErrorPages(new
ErrorPage(MultipartException.class, "/uploadError"));
}
```

这段代码创建了一个新的错误页面，当 MultipartException 出现的时候就会调用该页面。它还可以映射到 HTTP 状态上，EmbeddedServletContainerCustomizer 接口有很多其他的特性，通过它们能够自定义应用程序所运行的 Servlet 容器。参考 http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-web-applications.html#boot-features-customizing-embedded-containers 了解更多信息。

现在需要在 PictureUploadController 类中处理这个 uploadError URL：

```
@RequestMapping("uploadError")
public ModelAndView onUploadError(HttpServletRequest request) {
    ModelAndView modelAndView = new ModelAndView("uploadPage");
    modelAndView.addObject("error", request.getAttribute(WebUtils.
ERROR_MESSAGE_ATTRIBUTE));
    return modelAndView;
}
```

在 Servlet 环境中定义的错误页面包含了多项有用的属性，用于错误的调试（见表 4-1）。

表 4-1

属性	描述
javax.servlet.error.status_code	错误的状态码
javax.servlet.error.exception_type	异常类
javax.servlet.error.message	抛出异常的信息
javax.servlet.error.request_uri	异常发生时所对应的 URL
javax.servlet.error.exception	实际的异常
javax.servlet.error.servlet_name	捕获该异常的 Servlet

借助 Spring Web 的 WebUtils 类，可以非常便利地访问这些属性。

如果有人上传的文件太大的话，那么就会得到非常清晰的错误信息（见图 4-6）。

现在，可以通过上传大文件（>1MB）来测试异常得到了正确的处理，或者也可以将 multipart.maxFileSize 属性设置为一个更小的值，如 1KB：

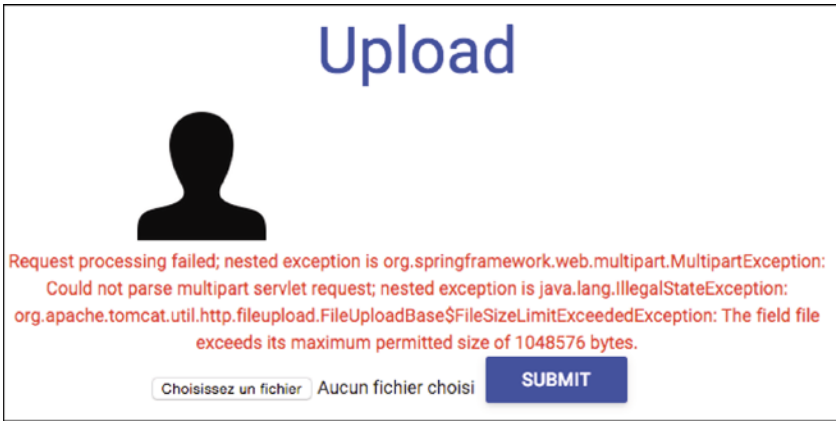

图 4-6

4.2 转换错误信息

对于开发人员来说，看到应用所抛出的异常是非常好的。但是，对用户来说，这就没什么价值了，因此，需要对其进行转换。为了实现这一点，需要将应用的 MessageSource 类注入到控制器的构造函数之中：

```
private final MessageSource messageSource;

@Autowired
public PictureUploadController(PictureUploadProperties
uploadProperties, MessageSource messageSource) {
    picturesDir = uploadProperties.getUploadPath();
    anonymousPicture = uploadProperties.getAnonymousPicture();
    this.messageSource = messageSource;
}
```

现在，我们就可以从信息 bundle 中获取信息了：

```
@ExceptionHandler(IOException.class)
```

```
public ModelAndView handleIOException(Locale locale) {
    ModelAndView modelAndView = new ModelAndView("profile/
uploadPage");
    modelAndView.addObject("error", messageSource.getMessage("upload.
io.exception", null, locale));
    return modelAndView;
}

@RequestMapping("uploadError")
public ModelAndView onUploadError(Locale locale) {
    ModelAndView modelAndView = new ModelAndView("profile/
uploadPage");
    modelAndView.addObject("error", messageSource.getMessage("upload.
file.too.big", null, locale));
    return modelAndView;
}
```

如下是英语的信息：

```
upload.io.exception=An error occurred while uploading the file. Please
try again.
upload.file.too.big=Your file is too big.
```

如下是法文的信息：

```
upload.io.exception=Une erreur est survenue lors de l'envoi du
fichier. Veuillez réessayer.
upload.file.too.big=Votre fichier est trop gros.
```

4.3　将基本信息放到会话中

我们接下来需要做的事情就是将基本信息存储在会话之中，这样的话，就不用在每次访问基本信息页面的时候进行重置了。原来的这种做法肯定会让用户感到厌烦，因此必须要解决这个问题。

HTTP 会话（session）是用来在请求之间存储信息的一种方式。HTTP 是无状态的协议，这就意味着没有办法将同一用户的两个请求关联起来。Servlet 容器最常用的办法是为每个用户关联一个名为 JSESSIONID 的 cookie。这个 cookie 将会通过请求头信息进行传输，借助这项技术允许用户将任意的对象存储在 Map 中，也就是名为 HttpSession 的抽象形式。这样的会话一般情况下会在用户关闭或切换 Web 浏览器，或者预定时间内用户没有活跃的动作时失效。

我们刚刚看到了一个方法通过使用@SessionAttributes 注解，将对象放到了会话中。如果只有一个控制器的话，这种做法还是不错的，但是如果要跨多个控制器的话，就很难共享数据了。我们需要依赖字符串并根据名字来解析属性，这很难进行重构。基于同样的原因，我们不想直接操作 HttpSession。不建议直接使用会话的另外一个原因在于，对依赖于会话的控制器进行单元测试是非常困难的。

在 Spring 中，将内容放到会话中的另外一种流行方式就是为 bean 添加@Scope("session")注解。

这样的话，就能将会话 bean 注入到控制器中，其他的 Spring 组件可以为其设置值，或从中检索值。

我们在 profile 包中创建一个 UserProfileSession 类：

```
package masterSpringMvc.profile;

import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;
import java.io.Serializable;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class UserProfileSession implements Serializable {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();

    public void saveForm(ProfileForm profileForm) {
        this.twitterHandle = profileForm.getTwitterHandle();
        this.email = profileForm.getEmail();
        this.birthDate = profileForm.getBirthDate();
        this.tastes = profileForm.getTastes();
    }


    public ProfileForm toForm() {
        ProfileForm profileForm = new ProfileForm();
        profileForm.setTwitterHandle(twitterHandle);
```

```
            profileForm.setEmail(email);
            profileForm.setBirthDate(birthDate);
            profileForm.setTastes(tastes);
            return profileForm;
        }
    }
```

我们提供了一种便利的方式进行 ProfileForm 对象的转换。这能够帮助我们通过 ProfileController 的构造函数来存储和检索表单数据。我们需要将 UserProfileSession 变量注入到控制器的函数中，并将其存储为一个域，还需要将 ProfileForm 暴露为模型属性，这样的话，就不用将它注入到 displayProfile 方法中了。最终，在基本信息校验之后，我们就能将它存储起来了：

```
    @Controller
    public class ProfileController {

        private UserProfileSession userProfileSession;
        @Autowired
        public ProfileController(UserProfileSession userProfileSession) {
            this.userProfileSession = userProfileSession;
        }

        @ModelAttribute
        public ProfileForm getProfileForm() {
            return userProfileSession.toForm();
        }
    @RequestMapping(value = "/profile", params = {"save"}, method = RequestMethod.POST)
    public String saveProfile(@Valid ProfileForm profileForm,BindingResult
bindingResult) {
            if (bindingResult.hasErrors()) {
                return "profile/profilePage";
            }
            userProfileSession.saveForm(profileForm);
            return "redirect:/profile";
        }

        // the rest of the code is unchanged
    }
```

这就是借助 Spring MVC 将数据存储到会话中所需的所有工作。

现在，如果你完成基本信息表单并刷新页面的话，数据就能在请求之间实现持久化。

在进入下一章之前，我想详细阐述一下这里所使用的理念。

第一个问题是这里使用了构造函数进行注入。ProfileController 的构造函数使用了@Autowired 注解，这意味着在实例化这个 bean 之前，Spring 将会在应用上下文中解析构造器参数。另外一种方案，可能比这个稍微简洁一些，那就是使用域注入：

```
@Controller
public class ProfileController {

    @Autowired
    private UserProfileSession userProfileSession;
}
```

如果我们不使用 spring-test 框架的话，那使用构造函数注入会更好一些，因为它能让控制器的单元测试更容易，另外，它也会让 bean 的依赖在一定程度上更加明确。

关于域注入和构造函数注入的更详细讨论，请参考 Oliver Gierke 有一篇很精彩的博客文章，地址是 http://olivergierke.de/2013/11/why-field-injection-is-evil/。

另外一个需要声明的是 Scope 注解上的 proxyMode 参数：

```
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
```

如果不算默认值的话，Spring 有 3 个可用的 proxyMode 参数。

- TARGET_CLASS：这将会使用 CGLib 代理。
- INTERFACES：这会创建 JDK 代理。
- NO：这样不会创建任何代理。

当我们将一些内容注入到长期存活的组件之中时，例如单例的 bean，那代理的好处就体现出来了。因为注入只会发生一次，bean 创建之后，对被注入 bean 的后续调用不一定能够反应它的实际状态。

在我们的场景中，会话 bean 的实际状态存储在会话之中，并没有直接反应在 bean 上。这就阐明了 Spring 为什么需要创建代理：它需要拦截对 bean 方法的调用，并监听它的变化。通过这种方式，bean 状态的存储和获取，就对底层 HTTP 会话完全透明了。

对于会话 bean，我们必须要使用代理模式。CGLib 代理会对字节码进行 instrument 操作，能够用在任意的类上，而 JDK 的方式可能会更加轻量级，但是需要你实现一个接口。

最后，我们让 UserProfileSession bean 实现了 Serializable 接口。这并非强制要求的，因

为 HTTP 会话能够将任意的对象存储在内存之中，不过，让存储在会话中的对象支持序列化是一种好的实践。

我们的确有可能会修改会话持久化的方式，实际上，我们会在第 8 章中将会话存储到 Redis 中，而 Redis 必须使用 Serializable 的对象。最好将会话想象为通用的数据存储。我们必须提供某种方式往这个存储系统中写入和读取对象。

为了让 bean 能够正常地序列化，还需要将它的每个域都变成可序列化的。在我们的样例中，字符串和日期都是可序列化的，所以不必为此担心了。

4.4　自定义错误页面

Spring Boot 允许我们定义自己的错误视图，替换之前看到的 Whitelabel 错误页面。它的名称必须是 error，其目的是用来处理所有的异常。默认的 BasicErrorController 类会暴露很多有用的模型属性，它们可以展现在页面上。

我们在 src/main/resources/templates 中创建一个自定义的错误页面，称之为 error.html：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head lang="en">
    <meta charset="UTF-8"/>
    <title th:text="${status}">404</title>

    <link href="/webjars/materializecss/0.96.0/css/materialize.css"
type="text/css" rel="stylesheet"
          media="screen,projection"/>
</head>
<body>
<div class="row">
    <h1 class="indigo-text center" th:text="${error}">Not found</h1>

    <p class="col s12 center" th:text="${message}">
        This page is not available
    </p>
</div>
</body>
</html>
```

现在，如果访问一个应用无法处理的 URL，将会看到如图 4-7 的错误页面。

图 4-7

处理错误的更高级方式就是定义自己的 `ErrorController` 实现类，这个控制器负责在全局处理所有的异常。可以参考 `ErrorMvcAutoConfiguration` 类和 `BasicErrorController`，这是默认的实现。

4.5 使用矩阵变量进行 URL 映射

我们已经了解了用户感兴趣的内容是什么。如果能够增强一下 Tweet 控制器，允许用户根据一个关键字列表进行搜索的话，那就更好了。

在 URL 中传递键值对的一种有意思的方式就是矩阵变量（matrix variable）。它非常类似于请求参数。考虑如下的代码：

```
someUrl/param?var1=value1&var2=value2
```

我们可以使用如下的矩阵变量，替换之前的参数：

```
someUrl/param;var1=value1;var2=value2
```

它还支持将每个参数设置为列表：

```
someUrl/param;var1=value1,value2;var2=value3,value4
```

矩阵变量可以映射为控制器中的不同对象类型。

- Map<String, List<?>>：这将会处理多个变量的多个值。
- Map<String, ?>：这会处理一个变量只有一个值的场景。

- List<?>：如果我们只对一个变量感兴趣，并且变量的名称可以配置的话，可以使用这种方式。

在我们的场景中，想处理的参数如下所示：

```
http://localhost:8080/search/popular;keywords=scala,java
```

第一个参数 popular 是 Twitter 搜索 API 能够识别的结果类型。它可以接受如下的值：mixed、recent 或 popular。

URL 的其他部分是关键字的列表，因此我们将其映射到一个简单的 List<String>对象中。

默认情况下，Spring MVC 将会移除 URL 中分号之后的字符。为了在应用程序中启用矩阵变量，我们所要做的第一件事情就是关闭这种默认行为。

在 WebConfiguration 类中添加如下的代码：

```
@Override
public void configurePathMatch(PathMatchConfigurer configurer) {
    UrlPathHelper urlPathHelper = new UrlPathHelper();
    urlPathHelper.setRemoveSemicolonContent(false);
    configurer.setUrlPathHelper(urlPathHelper);
}
```

在 search 包中，我们创建一个新的名为 SearchController 控制器。它的角色是处理如下的请求：

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.MatrixVariable;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.List;

@Controller
public class SearchController {
    private SearchService searchService;
    @Autowired
```

```
    public SearchController(SearchService searchService) {
        this.searchService = searchService;
    }

    @RequestMapping("/search/{searchType}")
    public ModelAndView search(@PathVariable String searchType, @
MatrixVariable List<String> keywords) {
        List<Tweet> tweets = searchService.search(searchType,
keywords);
        ModelAndView modelAndView = new ModelAndView("resultPage");
        modelAndView.addObject("tweets", tweets);
        modelAndView.addObject("search", String.join(",", keywords));
        return modelAndView;
    }
}
```

可以看到，我们能够重用已有的结果页面来展现 Tweet。另外，我们还想要将搜索委托给名为 SearchService 的另外一个类，并将这个类放到与 SearchController 相同的包中：

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class SearchService {
    private Twitter twitter;

    @Autowired
    public SearchService(Twitter twitter) {
        this.twitter = twitter;
    }

    public List<Tweet> search(String searchType, List<String>
keywords) {
        return null;
    }
}
```

现在，我们需要实现 search()方法。

`twitter.searchOperations().search(params)`的搜索操作将会接受一个`searchParameters` 参数用于高级查询。这个对象能够让我们基于多个条件（criteria）执行搜索，我们所关心的是 query、resultType 以及 count 属性。

首先，我们需要创建 ResultType 的一个构造函数，并使用 searchType 路径变量。ResultType 是一个枚举，所以我们可以遍历它不同的值，并找到匹配输入的那一个，这个过程中会忽略大小写：

```
private SearchParameters.ResultType getResultType(String searchType) {
    for (SearchParameters.ResultType knownType : SearchParameters.
ResultType.values()) {
        if (knownType.name().equalsIgnoreCase(searchType)) {
            return knownType;
        }
    }
    return SearchParameters.ResultType.RECENT;
}
```

可以使用如下的方法创建一个`SearchParameters`的构造函数：

```
private SearchParameters createSearchParam(String searchType, String
taste) {

    SearchParameters.ResultType resultType =
getResultType(searchType);
    SearchParameters searchParameters = new SearchParameters(taste);
    searchParameters.resultType(resultType);
    searchParameters.count(3);
    return searchParameters;
}
```

创建 SearchParameters 列表的构造函数就很简单了，只需要执行一个`map`操作（接受关键字的列表，并为每个关键字返回一个`SearchParameters`构造函数）：

```
List<SearchParameters> searches = keywords.stream()
        .map(taste -> createSearchParam(searchType, taste))
        .collect(Collectors.toList());
```

现在，我们想要为每个`SearchParameters`构造函数获取对应的 Tweet。你可能会认为我们要这样做：

```
List<Tweet> tweets = searches.stream()
        .map(params -> twitter.searchOperations().search(params))
        .map(searchResults -> searchResults.getTweets())
```

```
        .collect(Collectors.toList());
```

但是，如果思考一下的话，这将会返回一个 Tweet 的列表。我们想要做的是扁平化所有的 Tweet，使其成为一个简单的列表。调用 map 然后再将结果进行扁平化其实就是 flatMap 操作。所以，我们可以将其写成这样：

```
List<Tweet> tweets = searches.stream()
        .map(params -> twitter.searchOperations().search(params))
        .flatMap(searchResults -> searchResults.getTweets().stream())
        .collect(Collectors.toList());
```

flatMap 函数的语法会接受一个流作为参数，乍看上去，这可能有些难以理解。现在，我们可以进而看一下 SearchService 类的完整代码：

```
package masterSpringMvc.search;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.SearchParameters;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
public class SearchService {
    private Twitter twitter;

    @Autowired
    public SearchService(Twitter twitter) {
        this.twitter = twitter;
    }

    public List<Tweet> search(String searchType, List<String>
keywords) {
        List<SearchParameters> searches = keywords.stream()
                .map(taste -> createSearchParam(searchType, taste))
                .collect(Collectors.toList());

        List<Tweet> results = searches.stream()
                .map(params -> twitter.searchOperations().
search(params))
```

```
                .flatMap(searchResults -> searchResults.getTweets().
stream())
                .collect(Collectors.toList());
        return results;
    }
    private SearchParameters.ResultType getResultType(String
searchType) {
        for (SearchParameters.ResultType knownType : SearchParameters.
ResultType.values()) {
            if (knownType.name().equalsIgnoreCase(searchType)) {
                return knownType;
            }
        }
        return SearchParameters.ResultType.RECENT;
    }
    private SearchParameters createSearchParam(String searchType,
String taste) {
        SearchParameters.ResultType resultType =
getResultType(searchType);
        SearchParameters searchParameters = new
SearchParameters(taste);
        searchParameters.resultType(resultType);
        searchParameters.count(3);
        return searchParameters;
    }
}
```

现在，如果我们导航至 http://localhost:8080/search/mixed;keywords=scala,java，我们就能看到预期的结果，首先是 Scala 关键字的结果，然后是 Java 关键字的结果（见图 4-8）。

图 4-8

4.6 将其组合起来

现在，每项功能都能单独运行了，接下来该将其组合起来了。我们将会通过 3 步来实现：

1．将上传表单添加到个人基本信息页面，并移除原有的上传页面；

2．修改个人基本信息页面，让提交按钮直接触发口味搜索功能；

3．修改应用的主页，它应该直接展现符合用户口味的搜索结果。如果还没有设置口味的话，转到基本信息页面。

我建议读者自己尝试一下这些功能，在这个过程中，你将会遇到一些小问题，不过你所掌握的知识应该足以解决它们。在这一点上，我相信你们。

那么，你已经完成了这些工作（已经完成了，对吧？），现在，看一下我的方案。

第一步是移除已有的 uploadPage。不要往前看，直接做就可以了。

现在，将如下的这些代码行添加到 profilePage 页面的标题下面：

```
<div class="row">

    <div class="col m8 s12 offset-m2">
        <img th:src="@{/uploadedPicture}" width="100" height="100"/>
    </div>

    <div class="col s12 center red-text" th:text="${error}"
th:if="${error}">
        Error during upload
    </div>

    <form th:action="@{/profile}" method="post" enctype="multipart/
form-data" class="col m8 s12 offset-m2">

        <div class="input-field col s6">
            <input type="file" id="file" name="file"/>
        </div>

        <div class="col s6 center">
            <button class="btn indigo waves-effect waves-light"
type="submit" name="upload" th:text="#{upload}">Upload
                <i class="mdi-content-send right"></i>
```

```
            </button>
        </div>
    </form>
</div>
```

它的内容与之前的 uploadPage 非常类似。我们只是修改了标题并修改了提交按钮的文本，因此添加对应的翻译信息到资源信息的 bundle 中。

在英文中：

```
upload=Upload
```

在法文中：

```
Upload=Envoyer
```

我们还将提交按钮的名称改为 upload，这有助于我们在控制器侧识别它的行为。

现在，如果试图提交图片的话，它将会重定向到原有的上传页面。需要在 PictureUpload Controller 类的 onUpload 方法中对其进行修正：

```
@RequestMapping(value = "/profile", params = {"upload"}, method =
RequestMethod.POST)
public String onUpload(@RequestParam MultipartFile file,
RedirectAttributes redirectAttrs) throws IOException {

    if (file.isEmpty() || !isImage(file)) {
        redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
        return "redirect:/profile";
    }

    Resource picturePath = copyFileToPictures(file);
    userProfileSession.setPicturePath(picturePath);

    return "redirect:profile";
}
```

注意，我们修改了处理该 POST 请求的 URL，将“`/upload`”改成了“`/profile`”。GET 和 POST 请求具有相同的 URL 时，表单处理会更加简单一些，尤其是在处理异常的时候，会省去很多的麻烦。通过这种方式，在出现错误的时候，我们就不用对用户进行重定向了。

我们还移除了模型属性 picturePath。因为现在有一个 bean 代表了会话中的用户，也就

是 UserProfileSession，我们决定将其添加到这里。因此，在 UserProfileSession 类中新增 picturePath 属性以及对应的 getter 和 setter 方法。

不要忘记将 UserProfileSession 注入到 PictureUploadController 类中，并使其作为一个可使用的属性。

在会话中，所有的属性均是可序列化的，这与资源不同，我们需要按照不同的方式来存储。URL 看起来会比较合适，它是可序列化的，并且通过 UrlResource 类可以很容易地从 URL 创建资源：

```
@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class UserProfileSession implements Serializable {
    private URL picturePath;
    public void setPicturePath(Resource picturePath) throws
IOException {
        this.picturePath = picturePath.getURL();
    }

    public Resource getPicturePath() {
        return picturePath == null ? null : new
UrlResource(picturePath);
    }
}
```

我需要做的最后一件事情就是在出现错误之后，将 profileForm 作为可访问的模型属性。这是因为当 profilePage 渲染的时候，会需要它。

总结起来，如下就是最终版本的 PictureUploadController 类：

```
package masterSpringMvc.profile;

import masterSpringMvc.config.PictureUploadProperties;
import org.apache.tomcat.util.http.fileupload.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.ModelAndView;
```

```
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import javax.servlet.http.HttpServletResponse;
import java.io.*;
import java.net.URLConnection;
import java.util.Locale;

@Controller
public class PictureUploadController {
    private final Resource picturesDir;
    private final Resource anonymousPicture;
    private final MessageSource messageSource;
    private final UserProfileSession userProfileSession;
    @Autowired
    public PictureUploadController(PictureUploadProperties
uploadProperties,
                                   MessageSource messageSource,
                                   UserProfileSession
userProfileSession) {
        picturesDir = uploadProperties.getUploadPath();
        anonymousPicture = uploadProperties.getAnonymousPicture();
        this.messageSource = messageSource;
        this.userProfileSession = userProfileSession;
    }

    @RequestMapping(value = "/uploadedPicture")
    public void getUploadedPicture(HttpServletResponse response)
throws IOException {
        Resource picturePath = userProfileSession.getPicturePath();
        if (picturePath == null) {
            picturePath = anonymousPicture;
        }
        response.setHeader("Content-Type", URLConnection.guessContentT
ypeFromName(picturePath.getFilename()));
        IOUtils.copy(picturePath.getInputStream(), response.
getOutputStream());
    }

    @RequestMapping(value = "/profile", params = {"upload"}, method =
RequestMethod.POST)
    public String onUpload(@RequestParam MultipartFile file,
RedirectAttributes redirectAttrs) throws IOException {

        if (file.isEmpty() || !isImage(file)) {
```

```
            redirectAttrs.addFlashAttribute("error", "Incorrect file.
Please upload a picture.");
            return "redirect:/profile";
        }


        Resource picturePath = copyFileToPictures(file);
        userProfileSession.setPicturePath(picturePath);

        return "redirect:profile";
    }


    private Resource copyFileToPictures(MultipartFile file) throws
IOException {
        String fileExtension = getFileExtension(file.
getOriginalFilename());
        File tempFile = File.createTempFile("pic", fileExtension,
picturesDir.getFile());
        try (InputStream in = file.getInputStream();
             OutputStream out = new FileOutputStream(tempFile)) {

            IOUtils.copy(in, out);
        }
        return new FileSystemResource(tempFile);
    }


    @ExceptionHandler(IOException.class)
    public ModelAndView handleIOException(Locale locale) {
        ModelAndView modelAndView = new ModelAndView("profile/
profilePage");
        modelAndView.addObject("error", messageSource.
getMessage("upload.io.exception", null, locale));
        modelAndView.addObject("profileForm", userProfileSession.
toForm());
        return modelAndView;
    }

    @RequestMapping("uploadError")
    public ModelAndView onUploadError(Locale locale) {
        ModelAndView modelAndView = new ModelAndView("profile/
profilePage");
        modelAndView.addObject("error", messageSource.
```

```
        getMessage("upload.file.too.big", null, locale));
                modelAndView.addObject("profileForm", userProfileSession.
        toForm());
                return modelAndView;
            }


            private boolean isImage(MultipartFile file) {
                return file.getContentType().startsWith("image");
            }


            private static String getFileExtension(String name) {
                return name.substring(name.lastIndexOf("."));
            }
        }
```

所以，我们现在可以进入基本信息页面并上传图片，同时还能提供基本信息，如图 4-9 所示。

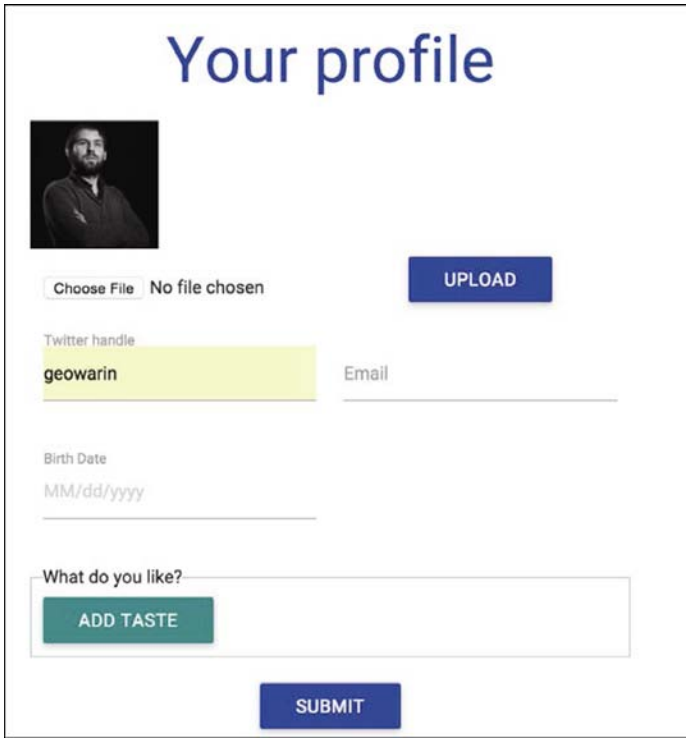

图 4-9

在基本信息补充完整之后，我们可以将用户重定向到搜索结果页面。为了实现该功能，我们需要修改 `ProfileController` 类的 `saveProfile` 方法：

```
@RequestMapping(value = "/profile", params = {"save"}, method =
RequestMethod.POST)
public String saveProfile(@Valid ProfileForm profileForm,
BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return "profile/profilePage";
    }
    userProfileSession.saveForm(profileForm);
    return "redirect:/search/mixed;keywords=" + String.join(",",
profileForm.getTastes());
}
```

我们已经能够通过用户的基本信息来搜索 Tweet，因此就没有必要再保留之前创建的 searchPage 和 TweetController 了，只需将 searchPage.html 页面和 TweetController 删除即可。

最后，我们可以修改主页，如果用户已经完善了其基本信息，将会自动重定向到匹配其口味的搜索结果页面。

在 `controller` 包中创建一个新的控制器，它负责重定向访问站点根目录的用户，如果基本信息不完整的话，就重定向到基本信息页面，如果已经设置了个人口味的话，就重定向到 resultPage 页面：

```
package masterSpringMvc.controller;

import masterSpringMvc.profile.UserProfileSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@Controller
public class HomeController {
    private UserProfileSession userProfileSession;

    @Autowired
    public HomeController(UserProfileSession userProfileSession) {
        this.userProfileSession = userProfileSession;
    }

    @RequestMapping("/")
    public String home() {
```

```
            List<String> tastes = userProfileSession.getTastes();
            if (tastes.isEmpty()) {
                return "redirect:/profile";
            }
            return "redirect:/search/mixed;keywords=" + String.join(",",
    tastes);
        }
    }
```

4.7　检查点

在本章中，我们添加了两个控制器，其中 PictureUploadController 负责将上传的文件写入到磁盘中并处理上传过程中的错误，SearchController 能够根据矩阵参数表示的关键字列表搜索 Tweet。

SearchController 控制器会将搜索功能委托给一个新的服务，即 SearchService。

我们删除了旧的 TweetController。

创建了一个新的会话 bean，UserProfileSession 使用它来存储用户的信息。

最后，我们在 WebConfiguration 中添加了两项内容，也就是 Servlet 容器级别的错误页面以及对矩阵变量的支持功能（见图 4-10）。

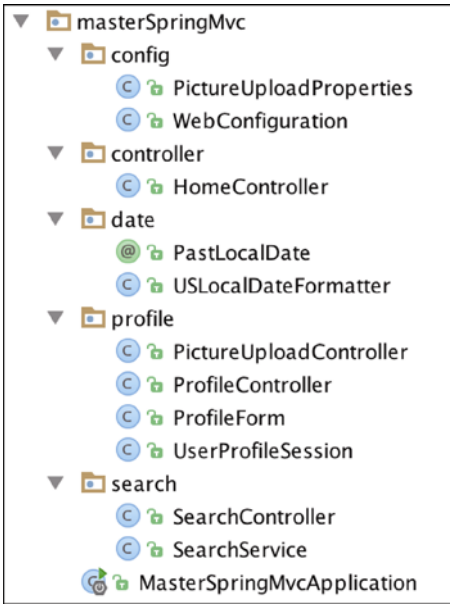

图 4-10

在资源方面，我们添加了一张图片来表示匿名用户以及一个静态页面来处理错误，并且将文件上传功能转移到了 profilePage 中，移除了旧的 searchPage（见图 4-11）。

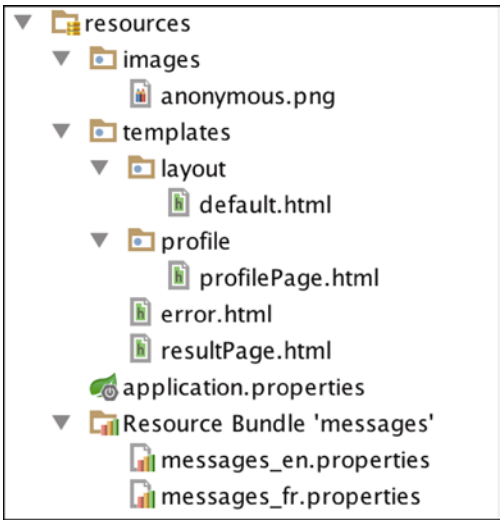

图 4-11

4.8 小结

在本章中，我们讨论了文件上传和错误处理功能。上传文件其实并不复杂，但是如何处理上传的文件则需要进行一番设计。我们可以将图片存储到数据库中，但是在这里将其写入到了磁盘中，并把每个用户的图片地址保存到了他们的会话中。

我们看到了在控制器级别和 Servlet 容器级别处理异常的典型方式。关于 Spring MVC 错误处理的更多资源，可以参考如下这篇博客文章：https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc。

我们的应用看起来已经很不错了，而且到目前为止所编写的代码其实并不太多。

在下一章中，我们将会看到 Spring MVC 框架也是构建 REST 应用的利器。

第 5 章
创建 RESTful 应用

在本章中，我们将会首先看一下 RESTful 架构的主要理念。然后，借助一些便利的工具，我们会设计一个用户友好的 API，借助 Jackson 的功能将我们的模型序列化为 JSON。

我们将会使用 Swagger UI 对应用进行文档化，其中会包含对应的错误码和 HTTP 动作，并且还会为应用生成一个简洁的前端。

最后，我们将会看一下其他形式的序列化，学习 Spring MVC 内容协商机制的更多知识。

5.1 什么是 REST

REST（表述性状态转移，Representational State Transfer）是一种架构风格，它定义了创建可扩展 Web 服务的最佳实践，这个过程中会充分发挥 HTTP 协议的功能。

RESTful Web 服务会天然具备如下的属性。

- 客户端-服务器：UI 是与数据存储分离的。
- 无状态：每个请求会包含服务器所需的足够信息，无需维护状态就能进行操作。
- 可缓存：服务器的响应中包含了足够的信息，客户端能够对数据存储做出合理的决策。
- 统一接口：URL 会唯一识别资源，能够通过超链接发现 API。
- 分层：API 的每个资源都提供了都合理程度的细节。

这种架构的优势在于易于维护以及便于进行服务发现。它的扩展性也很好，因为没有必要在服务器和客户端之间维护持久化的连接，这样就没有必要进行负载均衡或会话粘性。最后，因为服务的内容非常简洁且易于缓存，所以服务会更加高效。

我们通过使用 Richardson 的成熟度模型，看一下如何渐进式地设计更好的 API。

5.2 Richardson 的成熟度模型

Leonard Richardson 定义了著名的 4 个等级，从 0 级到 3 级，它们描述了 Web API 的“RESTful 程度（RESTfulness）”。每个等级都需要开展额外的工作，并在 API 方面进行投入，但是这也会带来额外的收益。

5.2.1 第 0 级——HTTP

第 0 级非常容易实现，我们只需让资源能够在网络上通过 HTTP 协议获取即可。你可以使用任意合适的数据表述形式（XML、JSON 等），只要能够最好地满足你的使用场景即可。

5.2.2 第 1 级——资源

大多数的人在听到 REST 这个术语时，首先想到的是资源。资源是模型中某个元素的唯一标识符，例如这种元素可以是一个用户或一篇 Tweet。借助 HTTP，资源会与一个统一资源标识符 URL 进行关联，如下面的样例所示：

- /users 将会包含所有用户的列表；
- /user/42 将会包含特定的用户；
- /user/42/tweets 包含了特定用户的所有 Tweet 列表。

你的 API 可能还允许访问某个用户相关的特定 Tweet，那就可以使用“/user/42/tweet/3”，或者每篇 Tweet 都是唯一标识的，如果是这样的话，可能会使用这种“/tweet/3”的形式。

这个级别的目的在于通过暴露多个特定的资源，处理应用程序的复杂性。

至于服务器可以返回的响应类型，并没有相关的规则。当你通过“/users”列出所有资源的时候，你可能只想包含相关的基本信息，而在请求特定的资源时，再给出更多的细节。有些 API 甚至能够在请求某些域之前，提前将你所感兴趣的域列出来。

API 形式的定义完全取决于你，只需记住一条简单的规则：最小惊讶原则。给用户提供所预期的内容，这样你的 API 就已经很好了。

5.2.3 第 2 级——HTTP 动作

这个级别是使用 HTPP 动作（verb）来识别资源可能的行为。这种方式能够很好地描

述你的 API 能够完成什么功能，因为在开发人员中，HTTP 动作是众所周知的标准。

主要的动作如下所示。

- GET：在一个特定的 URL 上读取数据。
- HEAD：它的行为与 GET 相同，但是不包含响应体。在获取资源的元数据（如缓存信息等等）时，这种方式是有用的。
- DELETE：这将会删除某个资源。
- PUT；这会更新或创建资源。
- POST：这会更新或创建资源。
- PATCH：这会部分更新资源。
- OPTIONS：这会返回服务端针对特定资源所支持方法列表。

大多数的应用都会有创建、读取、更新和删除（Create Read Update Delete，CRUD）操作，它们通过 3 个动作就可以实现：GET、DELETE 和 POST。你所实现的动作越多，你的 API 就会越丰富越具有语义性。这样能够帮助第三方与服务进行交互，他们只需输入几条命令并查看一下发生了什么即可。

OPTIONS 和 HEAD 很少见到，因为它们是在元数据级别上的，对于应用来说并不是那么至关重要。

乍看上去，PUT 和 POST 动作做了相同的事情。主要的区别在于 PUT 动作被认为是幂等的，这意味着多次发送同一个请求将会导致相同的服务器状态。该规则的含义就是 PUT 动作要针对给定的 URL 进行操作，并且其中要包含足够的信息以保证请求能够成功。

例如，客户端可以使用“/user/42”来 PUT 数据，其结果可能是更新也可能是新建，这取决于在请求之前，该实体是否已经存在。

而另一方面，当你无法精确地知道该往哪个 URL 写入数据的话，就应该使用 POST。我们可以发送 POST 请求到“/users”，在请求中不指定 ID，这样预期会创建一个新的用户；我们也可以发送 POST 请求到相同的“/users”资源，并且在请求体中指定一个用户 ID，这样的话，就可以预期服务器端会更新对应的用户。

可以看到，这两种方式都是可行的。一种常用的场景是使用 POST 进行创建（因为，大多数情况下，会由服务端来决定 ID 如何生成），而使用 PUT 动作来更新资源，此时资源

的 ID 是已知的。

服务器端可能还允许资源进行部分修改（客户端不需要发送资源的全部内容），那么在这种情况下，应该对应于 PATCH 方法。

在这个等级，提供响应的时候，我建议你使用有意义的 HTTP 代码，稍后，我们将会看到最为通用的代码。

5.2.4 第 3 级——超媒体控制

超媒体控制（Hypermedia control）也被称为超媒体即应用状态引擎（Hypertext As The Engine Of Application State，HATEOAS）。在这个复杂的缩写词背后，蕴含着 RESTful 服务最重要的特性：通过使用超文本链接，可以进行服务的发现。这实际上就是服务器端通过响应头或响应体，告诉客户端其可选的功能。

例如，在通过使用 PUT 方法创建资源之后，服务端应该返回代码为“201 CREATED”的响应，并且在响应头信息中使用 Location 属性中包含新创建资源的 URI。

在如何定义到 API 其他部分的链接方面，并没有什么标准。Spring Data REST 是一个 Spring 项目，它允许使用最少的配置来创建 RESTful 后端，典型的输出如下所示：

```
{
  "_links" : {
    "people" : {
      "href" : "http://localhost:8080/users{?page,size,sort}",
      "templated" : true
    }
  }
}
```

然后，访问“/users”：

```
{
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/users{?page,size,sort}",
      "templated" : true
    },
    "search" : {
      "href" : "http://localhost:8080/users/search"
    }
```

```
  },
  "page" : {
    "size" : 20,
    "totalElements" : 0,
    "totalPages" : 0,
    "number" : 0
  }
}
```

在如何处理 API 方面，这给了我们一个不错的建议，对吧？

5.3　API 版本化

如果第三方客户端使用你的 API 的话，你可以考虑对你的 API 进行版本化，从而避免更新应用的时候，带来破坏性的变更。

版本化 API 通常会在子域下面提供稳定的资源访问功能。例如，GitLab 维护了 3 个版本的 API，它们可以通过类似这样的 https://example/api/v3 URL 进行访问。与很多软件架构决策类似，版本化也是一种权衡。

在 API 的设计上，这会需要更多的工作，还需要识别出 API 中破坏性的变更。通常来讲，相对于移除或转换 API 的实体结果或请求，添加新内容所造成的问题并不会更少。

大多数情况下，你会同时负责 API 和客户端，因此，完全可以移除这种复杂性。

关于 API 版本化的更深入讨论，可以参考：
http://www.troyhunt.com/2014/02/your-api-versioning-is-wrong-which-is.html。

5.4　有用的 HTTP 代码

好的 RESTful API 还有另外一个很重要的方面，那就是合理地使用 HTTP 编码。HTTP 规范定义了很多标准的编码。在设计与用户交互的良好 API 方面，它们能够涵盖 99%的需求。如下的表格列出了最重要的编码，它们是每个 API 都会用到的，也是每个开发人员都应该掌握的，参见表 5-1。

表 5-1

编码	含义	用途
2xx –成功	当所有的事情运行正常的时候，会使用这些编码	
200	所有的事情都运行正常	请求成功
201	某个资源已经成功创建	资源创建成功。响应应该包含所创建资源位置的列表
204	没有要返回的内容	服务器已经成功处理请求，但是没有可返回的内容
3xx –重定向	如果客户端需要进一步操作才能完成请求的话，就会使用这些编码	
301	永久移除	资源的 URL 已经发生了变化，它的新位置应该使用 Location 头信息来表示
304	资源没有发生修改	从上次请求以来，资源没有发生变化。这个响应必须包含日期、ETag 以及缓存信息
4xx –客户端错误	因为客户端的错误，导致请求没有成功执行	
400	Bad request	客户端发送的数据无法理解
403	Forbidden	请求能够理解，但是不允许访问。其详情可以通过错误的描述信息来完善
404	Not found	没有发现匹配该 URI 的内容。如果不想暴露安全信息的话，可以使用它来替代 403
409	Conflict	请求与其他的修改相冲突。响应中应该包含如何解决这个冲突的信息
5xx – 服务端错误	在服务端一侧发生了错误	
500	服务器端的内部错误	服务器在处理请求时，出现了意料之外的错误

更为详细的列表，可以参见:
http://www.restapitutorial.com/httpstatuscodes.html。

5.5 客户端为王

我们将会允许第三方客户端通过 REST API 来获取搜索结果。这些结果可以使用 JSON

或 XML 的形式来展现。

我们希望处理“/api/search/mixed;keywords=springFramework”形式的请求。这与我们之前的搜索形式是非常类似的，唯一区别在于请求路径是以 api 作为开头。所有在这个命名空间中的 URI 都应该返回二进制结果。

我们在 search.api 包中创建一个新的 SearchApiController 类：

```
package masterSpringMvc.search.api;

import masterSpringMvc.search.SearchService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.twitter.api.Tweet;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/search")
public class SearchApiController {
    private SearchService searchService;

    @Autowired
    public SearchApiController(SearchService searchService) {
        this.searchService = searchService;
    }

    @RequestMapping(value = "/{searchType}", method = RequestMethod.
GET)
    public List<Tweet> search(@PathVariable String searchType, @
MatrixVariable List<String> keywords) {
        return searchService.search(searchType, keywords);
    }
}
```

这与我们之前的控制器非常类似，但是有 3 个小的差异：

- 控制器类使用了@RequestMapping 注解，这会作为我们的基础地址，会以前缀的形式添加到控制器的其他映射声明之中；
- 我们在 search 方法中没有重定向到一个视图，而是返回一个简单的对象；
- 控制器上使用了@RestController 注解来代替@Controller。

RestController 注解是一种快捷方式，它所声明的控制器在返回响应时，就如同使用了@ResponseBody 注解一样。它会告诉 Spring 将返回类型序列化为合适的格式，默认情况下为 JSON 格式。

当编写 REST API 的时候，有个最佳实践就是指定想要响应的 HTTP 方法。请求采用相同的方式来处理 GET 和 POST 方法是不太可能出现的。

如果你访问 http://localhost:8080/api/search/mixed;keywords=springFramework 的话，将会得到一个非常冗长的结果，如图 5-1 所示。

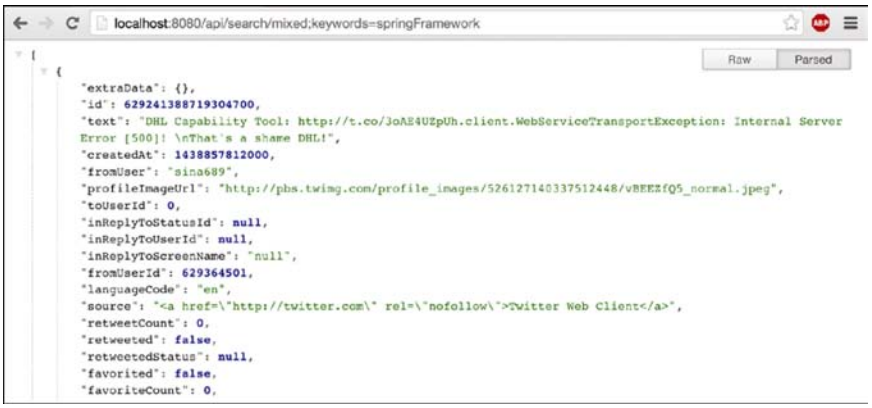

图 5-1

Spring 将会使用 Jackson 自动地将 Tweet 类的所有属性进行序列化。

5.6 调试 RESTful API

在浏览器中，我们只能针对特定的 API 执行 GET 操作。好的工具将会让开发过程更加简单。有很多测试 RESTful API 的工具，我这里只列出我使用和喜爱的工具。

5.6.1 JSON 格式化扩展

通常我们只是测试 GET 方法，此时我们的第一反应就是将地址复制到浏览器中，并检查结果。在这种情况下，通过使用 Chrome 下的 JSON Formatter 或 Firefox 下的 JSONView，我们所看到的结果将会进行格式化，而不是纯文本。

5.6.2 浏览器中的 RESTful 客户端

对于处理 HTTP 请求来说，浏览器是最自然的工具。但是，使用地址栏，我们很难细

致地测试 API。

Postman 是针对 Chrome 的一个扩展，RESTClient 是实现同等功能的 Firefox 扩展。它们具有类似的特性，例如创建和共享查询、修改头信息以及处理认证（基本认证、摘要认证以及 OAuth 认证）。当编写本书的时候，只有 RESTClient 能够处理 OAuth2。

5.6.3 httpie

httpie 是一个命令行工具，类似于 curl，但是它更加面向 REST 查询。它允许我们按照如下的方式输入命令：

```
http PUT httpbin.org/put hello=world
```

这要比 curl 的丑陋版本友好得多：

```
curl -i -X PUT httpbin.org/put -H Content-Type:application/json -d
'{"hello": "world"}'
```

5.7 自定义 JSON 输出

通过使用我们的工具，能够非常便利地查看服务器所产生的响应。它非常庞大复杂，默认情况下，Spring Boot 所使用的 JSON 序列化库 Jackson 会将所有能够通过 getter 方法访问的内容都进行序列化。

我们想要的内容可能更加轻量级，例如：

```
{
  "text": "original text",
  "user": "some_dude",
  "profileImageUrl": "url",
  "lang": "en",
  "date": 2015-04-15T20:18:55,
  "retweetCount": 42
}
```

自定义哪些域要进行序列化的最简单方式就是为我们的 bean 添加注解。我们可以在类级别使用@JsonIgnoreProperties 注解，定义想要忽略的一组属性，也可以在想要忽略的属性所对应的 getter 方法上，添加@JsonIgnore 注解。

在该样例中，Tweet 并不是我们自己的类。它是 Spring Social Twitter 的一部分，我们

并没有办法为其添加注解。

直接序列化模型类通常来讲并不是好的方案，它会将你的模型类与序列化库关联在一起，所使用的序列化库是具体的实现细节，所以不应该将其与模型进行关联。

当处理不可修改的代码时，Jackson 提供了两个方案：

- 创建专门用于序列化功能的新类；
- 使用混入（mixin），这是与模型相关联的简单类。它们会在你自己的代码中声明，能够使用任意的 Jackson 注解。

因为我们只是对模型类中的域进行简单的转换（一些会隐藏，还有个别的要重命名），我们可以选择混入的方案。

这是一种很好的非侵入式的方式，可以通过简单的类或接口在运行时重命名或排除相关的域。

另外一种指定域子集的方式就是使用@JsonView 注解，这样这些域的子集就可以用到应用程序的其他地方了。在本章中，将不会讨论这种方式，不过我推荐你阅读这篇很棒的博客文章：https://spring.io/blog/2014/12/02/latest-jackson-integration-improvements-in-spring。

我们想要控制 API 的输出，所以只需创建一个简单的新类，名为 LightTweet，它可以通过 Tweet 来进行构建：

```
package masterSpringMvc.search;

import org.springframework.social.twitter.api.Tweet;
import org.springframework.social.twitter.api.TwitterProfile;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Date;

public class LightTweet {
    private String profileImageUrl;
    private String user;
    private String text;
    private LocalDateTime date;
    private String lang;
    private Integer retweetCount;

    public LightTweet(String text) {
```

```
        this.text = text;
    }

    public static LightTweet ofTweet(Tweet tweet) {
        LightTweet lightTweet = new LightTweet(tweet.getText());
        Date createdAt = tweet.getCreatedAt();
        if (createdAt != null) {
            lightTweet.date = LocalDateTime.ofInstant(createdAt.
toInstant(), ZoneId.systemDefault());
        }
        TwitterProfile tweetUser = tweet.getUser();
        if (tweetUser != null) {
            lightTweet.user = tweetUser.getName();
            lightTweet.profileImageUrl = tweetUser.
getProfileImageUrl();
        }
        lightTweet.lang = tweet.getLanguageCode();
        lightTweet.retweetCount = tweet.getRetweetCount();
        return lightTweet;
    }

  // don't forget to generate getters
  // They are used by Jackson to serialize objects
}
```

我们现在需要让 SearchService 类返回 LightTweet 类，用它来代替之前的 Tweet 类：

```
    public List<LightTweet> search(String searchType, List<String>
keywords) {
        List<SearchParameters> searches = keywords.stream()
                .map(taste -> createSearchParam(searchType, taste))
                .collect(Collectors.toList());

        List<LightTweet> results = searches.stream()
                .map(params -> twitter.searchOperations().
search(params))
                .flatMap(searchResults -> searchResults.getTweets().
stream())
                .map(LightTweet::ofTweet)
                .collect(Collectors.toList());

        return results;
    }
```

这也会影响到 SearchApiController 类的返回类型，以及 SearchController

类中的 tweets 模型属性，对这两个类进行必要的修改。

我们还需要修改 resultPage.html 文件的代码，因为有些属性发生了变化（我们没有嵌套的 user 属性了）：

```
<ul class="collection">
    <li class="collection-item avatar" th:each="tweet : ${tweets}">
        <img th:src="${tweet.profileImageUrl}" alt="" class="circle"/>
        <span class="title" th:text="${tweet.user}">Username</span>

        <p th:text="${tweet.text}">Tweet message</p>
    </li>
</ul>
```

至此，基本上已经完成了。如果你重新启动应用并访问 http://localhost:8080/api/search/mixed;keywords=springFramework 的话，你会发现日期格式并不是我们所预期的那样，如图 5-2 所示。

图 5-2

这是因为 Jackson 没有提供对 JSR-310 日期的内置支持，幸好，这个问题也很容易解决。我们只需将如下的库添加到 build.gradle 文件的依赖中：

```
compile 'com.fasterxml.jackson.datatype:jackson-datatype-jsr310'
```

这确实能改变日期格式，但是它现在的输出是一个数组而不是格式化的日期。

为了改变这种行为，我们需要理解这个库是做什么的，它包含了一个名为 JSR-310

Module 的 Jackson 新模块。Jackson 模块是自定义序列化和反序列化的扩展点。按照 JacksonAutoConfiguration 类的定义，Spring Boot 会在启动的时候自动注册这个模块，它会创建一个默认的 JacksonObjectMapper，对一些知名的模块提供功能支持。

前面提到的模块会为 JSR-310 定义的新类添加一些序列化器和反序列化器，这样的话，它们会试图将每一个日期均尝试转换为 ISO 格式，参见 https://github.com/FasterXML/jackson-datatype-jsr310。

例如，如果我们仔细看一下 LocalDateTimeSerializer 的话，会发现实际上它有两个模式，可以通过名为 WRITE_DATES_AS_TIMESTAMPS 的序列化特性在它们之间进行切换。

要定义该属性，我们需要自定义 Spring 默认的对象映射器。从自动配置功能可以看出，Spring MVC 提供了一个工具类来创建我们使用的 ObjectMapper。在 WebConfiguration 中添加如下的 bean：

```
@Bean
@Primary
public ObjectMapper objectMapper(Jackson2ObjectMapperBuilder builder)
{
   ObjectMapper objectMapper = builder.createXmlMapper(false).build();
   objectMapper.configure(SerializationFeature.WRITE_DATES_AS_
TIMESTAMPS, false);
   return objectMapper;
}
```

这一次，日期就能正常格式化了，如图 5-3 所示。

图 5-3

5.8 用户管理 API

现在，搜索 API 已经非常好了，接下来我们做一点更有意思的事情。像许多 Web 应用一样，我们需要有一个用户管理模块来识别用户。为了实现该功能，我们将会创建一个新的 user 包，在这个包中，添加如下的模型类：

```
package masterSpringMvc.user;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class User {
    private String twitterHandle;
    private String email;
    private LocalDate birthDate;
    private List<String> tastes = new ArrayList<>();

    // Getters and setters for all fields
}
```

因为我们现在还不想使用数据库，所以可以在相同的包下创建 `UserRepository` 类，它的后端是一个简单的 Map：

```
package masterSpringMvc.user;

import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Repository
public class UserRepository {
    private final Map<String, User> userMap = new
ConcurrentHashMap<>();

    public User save(String email, User user) {
        user.setEmail(email);
```

```
        return userMap.put(email, user);
    }
    public User save(User user) {
        return save(user.getEmail(), user);
    }

    public User findOne(String email) {
        return userMap.get(email);
    }


    public List<User> findAll() {
        return new ArrayList<>(userMap.values());
    }

    public void delete(String email) {
        userMap.remove(email);
    }

    public boolean exists(String email) {
        return userMap.containsKey(email);
    }
}
```

最后，在 `user.api` 包中，创建一个非常简单的控制器实现：

```
package masterSpringMvc.user.api;

import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class UserApiController {

    private UserRepository userRepository;

    @Autowired
    public UserApiController(UserRepository userRepository) {
        this.userRepository = userRepository;
```

```
    }
    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> findAll() {
        return userRepository.findAll();
    }

    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public User createUser(@RequestBody User user) {
        return userRepository.save(user);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
PUT)
    public User updateUser(@PathVariable String email, @RequestBody
User user) {
        return userRepository.save(email, user);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
DELETE)
    public void deleteUser(@PathVariable String email) {
        userRepository.delete(email);
    }
}
```

我们通过 RESTful repository 实现了所有典型的 CRUD 操作，这里使用用户的 E-mail 地址作为唯一标识。

在这个场景中，你很快就会遇到问题，因为 Spring 会略去点号后面的内容。它的解决方案非常类似于在第 4 章中，我们解决矩阵变量 URL 映射时，为了支持 URL 中的分号所采用的方案，

添加 useRegisteredSuffixPatternMatch 属性，在我们之前定义的 `WebConfiguration` 类的 `configurePathMatch()`方法中，它被设置成了 `false`：

```
@Override
public void configurePathMatch(PathMatchConfigurer configurer) {
    UrlPathHelper urlPathHelper = new UrlPathHelper();
    urlPathHelper.setRemoveSemicolonContent(false);
    configurer.setUrlPathHelper(urlPathHelper);
    configurer.setUseRegisteredSuffixPatternMatch(true);
}
```

现在，我们的 API 已经准备好了，可以和它进行交互了。

如下是通过 httpie 发送的几条示例命令：

```
~ $ http get http://localhost:8080/api/users
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Mon, 20 Apr 2015 00:01:08 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

[]

~ $ http post http://localhost:8080/api/users email=geo@springmvc.com
birthDate=2011-12-12 tastes:='["spring"]'
HTTP/1.1 200 OK
Content-Length: 0
Date: Mon, 20 Apr 2015 00:02:07 GMT
Server: Apache-Coyote/1.1


~ $ http get http://localhost:8080/api/users
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Mon, 20 Apr 2015 00:02:13 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

[
    {
        "birthDate": "2011-12-12",
        "email": "geo@springmvc.com",
        "tastes": [
            "spring"
        ],
        "twitterHandle": null
    }
]

~ $ http delete http://localhost:8080/api/user/geo@springmvc.com
HTTP/1.1 200 OK
Content-Length: 0
Date: Mon, 20 Apr 2015 00:02:42 GMT
```

```
Server: Apache-Coyote/1.1


~ $ http get http://localhost:8080/api/users
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Mon, 20 Apr 2015 00:02:46 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

[]
```

这样已经很不错了，但还可以更进一步。状态码还没有进行处理，我们需要更多的 RESTful 特性，沿着 Richardson 的成熟度模型继续前进。

5.9 状态码与异常处理

我们想要做的第一件事就是正确地处理响应状态。默认情况下，Spring 会自动处理一些状态。

- 500 Server Error：这表明在处理请求的时候出现了错误。
- 405 Method not Supported：当你在已有的处理器上使用错误的方法时，会出现这个错误。
- 404 Not Found：当处理器不存在时，会出现该错误。
- 400 Bad Request：这表明请求体或参数不能匹配服务器端的预期。
- 200 OK：如果请求处理没有遇到任务错误的话，将会对应该状态。

在 Spring MVC 中，有两种方式来返回状态码：

- 在 REST 控制器中，返回 `ResponseEntity` 类；
- 所抛出的异常由专门的处理器来捕获。

5.9.1 带有状态码的 ResponseEntity

HTTP 协议规定当我们新建一个用户的时候，需要返回 201 Created 状态。在我们的 API 中，可以通过 POST 方法来实现。当操作不存在的实体时，需要抛出 404 错误。

Spring MVC 有一个类能够将 HTTP 状态与响应实体关联，这个类叫做 ResponseEntity，

我们更新 UserApiController 类，可以增加对错误码的处理：

```
package masterSpringMvc.user.api;

import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class UserApiController {

    private UserRepository userRepository;

    @Autowired
    public UserApiController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }


    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> findAll() {
        return userRepository.findAll();
    }
    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public ResponseEntity<User> createUser(@RequestBody User user) {
        HttpStatus status = HttpStatus.OK;
        if (!userRepository.exists(user.getEmail())) {
            status = HttpStatus.CREATED;
        }
        User saved = userRepository.save(user);
        return new ResponseEntity<>(saved, status);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
PUT)
    public ResponseEntity<User> updateUser(@PathVariable String email,
@RequestBody User user) {
        if (!userRepository.exists(user.getEmail())) {
```

```
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        User saved = userRepository.save(email, user);
        return new ResponseEntity<>(saved, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
DELETE)
    public ResponseEntity<User> deleteUser(@PathVariable String email)
{

        if (!userRepository.exists(email)) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        userRepository.delete(email);
        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

可以看到我们在向 RESTful 的第一等级演化，但是这里会有很多的样板式代码。

5.9.2 使用异常来处理状态码

在 API 中处理错误的另外一种方法就是抛出异常。使用 Spring MVC，有两种方法来匹配异常：

- 在类级别使用@ExceptionHandler，类似于我们在第 4 章中，在上传控制器上处理 IOException 的方式；
- 使用@ControllerAdvice，捕获所有控制器或控制器的一个子集所抛出的全局异常。

这两个方案能够帮助你做一些面向业务的决策，并且在应用中定义一组实践。

要将这些处理器与 HTTP 状态码进行关联，我们可以将 response 注入到带有注解的方法中并使用 `HttpServletResponse.sendError()`方法，或者为这些方法添加@ResponseStatus 注解。

我们将会定义自己的异常，即 EntityNotFoundException。当要操作的用户实体不存在的时候，业务 repository 将会抛出该异常。这会有助于完善 API 代码。

如下是异常的代码，我们可以将它放到一个名为 error 的新包中：

```
package masterSpringMvc.error;

public class EntityNotFoundException extends Exception {
```

```
    public EntityNotFoundException(String message) {
        super(message);
    }

    public EntityNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

现在，我们的 repository 就可以在各种地方抛出异常了，同时，代码中也区分了保存用户和更新用户的操作：

```
package masterSpringMvc.user;

import masterSpringMvc.error.EntityNotFoundException;
import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Repository
public class UserRepository {
    private final Map<String, User> userMap = new
ConcurrentHashMap<>();

    public User update(String email, User user) throws
EntityNotFoundException {
        if (!exists(email)) {
            throw new EntityNotFoundException("User " + email + "
cannot be found");
        }
        user.setEmail(email);
        return userMap.put(email, user);
    }

    public User save(User user) {
        return userMap.put(user.getEmail(), user);
    }

    public User findOne(String email) throws EntityNotFoundException {
        if (!exists(email)) {
            throw new EntityNotFoundException("User " + email + "
cannot be found");
```

```
            }
            return userMap.get(email);
        }

        public List<User> findAll() {
            return new ArrayList<>(userMap.values());
        }
        public void delete(String email) throws EntityNotFoundException {
            if (!exists(email)) {
                throw new EntityNotFoundException("User " + email + "
    cannot be found");
            }
            userMap.remove(email);
        }

        public boolean exists(String email) {
            return userMap.containsKey(email);
        }
    }
```

控制器变得更简单了，因为它不用处理 404 状态了，现在我们在控制器方法中抛出 EntityNotFound 异常：

```
package masterSpringMvc.user.api;

import masterSpringMvc.error.EntityNotFoundException;
import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class UserApiController {

    private UserRepository userRepository;

    @Autowired
    public UserApiController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
```

```
    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> findAll() {
        return userRepository.findAll();
    }
    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public ResponseEntity<User> createUser(@RequestBody User user) {
        HttpStatus status = HttpStatus.OK;
        if (!userRepository.exists(user.getEmail())) {
            status = HttpStatus.CREATED;
        }
        User saved = userRepository.save(user);
        return new ResponseEntity<>(saved, status);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
PUT)
    public ResponseEntity<User> updateUser(@PathVariable String email,
@RequestBody User user) throws EntityNotFoundException {
        User saved = userRepository.update(email, user);
        return new ResponseEntity<>(saved, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/user/{email}", method = RequestMethod.
DELETE)
    public ResponseEntity<User> deleteUser(@PathVariable String email)
throws EntityNotFoundException {
        userRepository.delete(email);
        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

如果我们不处理这个异常的话，Spring 默认将会抛出 500 错误。为了处理这个异常，我们需要在 error 包中创建一个很小的类，与 EntityNotFoundException 类位于同一个包中，名称是 EntityNotFoundMapper，它将会负责处理异常：

```
package masterSpringMvc.error;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
public class EntityNotFoundMapper {
```

```
    @ExceptionHandler(EntityNotFoundException.class)
    @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Entity
could not be found")
    public void handleNotFound() {
    }
}
```

通过为 bean 添加@ControllerAdvice 注解，我们能够为一组控制器添加额外的行为。这些控制器通知可以用来处理异常，也可以通过@ModelAttribute 来声明模型属性，或通过@InitBinder 声明校验策略。

借助刚刚编写的代码，我们就能处理控制器抛出的所有 EntityNotFoundException 异常，并将其与 404 状态进行关联。我们可以抽象这种理念并确保应用的所有控制器按照一致的方式来进行处理。

在当前的级别下，我们还没有在 API 处理中处理超链接。我建议你去了解一下 Spring HATEOAS 和 Spring Data REST，它们提供了非常优雅的方案，能够让你的资源更容易被发现。

5.10 通过 Swagger 实现文档化

Swagger 是一个非常棒的项目，它能够允许我们在一个 HTML5 Web 页面中，对 API 进行文档化和交互。图 5-4 展现了 API 的文档。

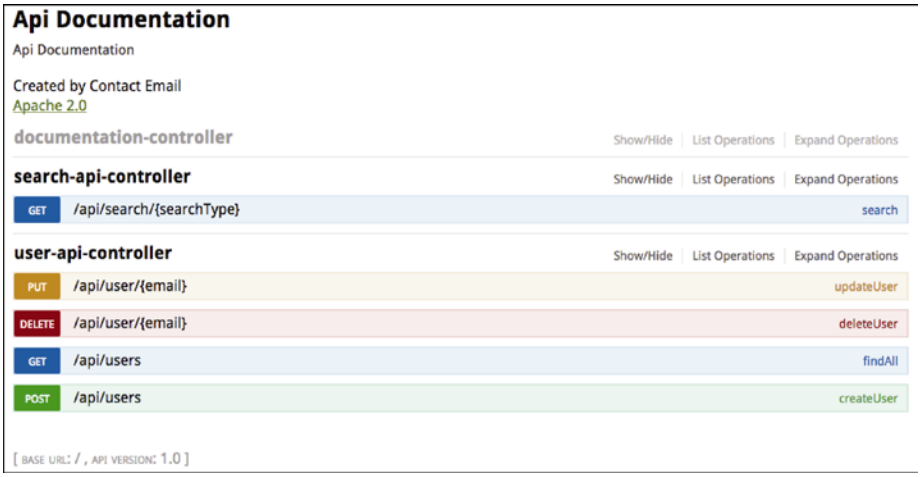

图 5-4

Swagger 曾经非常庞大（使用 Scala 编写的），在一定程度上来讲与 Spring 集成的配置也很

复杂。从 2.0 版本开始，这个库进行了重写，名为 spring-fox 的库能够很容易地实现集成。

spring-fox 之前被称之为 swagger-springmvc，它已经存在超过 3 年了，目前，依然是很活跃的项目。

在构建文件中，添加如下的依赖：

```
compile 'io.springfox:springfox-swagger2:2.1.2'
compile 'io.springfox:springfox-swagger-ui:2.1.2'
```

第一项依赖提供了一个注解，借助它能够在我们的应用中启用 Swagger 功能，它还提供了一个 API，用来通过注解描述资源。Swagger 会生成 API 的 JSON 格式表述。

第二项依赖是一个 WebJar，其中包含了静态资源，这些资源会以 Web 客户端的形式使用前面所生成的 JSON。

我们唯一需要做的事情就是将@EnableSwagger2 注解添加到 WebConfiguration 类上：

```
@Configuration
@EnableSwagger2
public calass WebConfiguration extends WebMvcConfigurerAdapter {
 }
```

我们刚刚添加了 swagger-ui.jar 包，在它的“META-INF/ resources”中会包含一个 HTML 文件。

在当你访问 http://localhost:8080/swagger-ui.html 的时候，Spring Boot 会自动为其提供服务。

在默认情况下，Springfox 会扫描整个类路径并展示应用中所有声明的请求映射。

在我们的场景下，我们只想暴露这一个 API：

```
@Bean
public Docket userApi() {
    return new Docket(DocumentationType.SWAGGER_2)
        .select()
        .paths(path -> path.startsWith("/api/"))
        .build();
}
```

Springfox 会与一组 Docket 协同工作，这些 Docket 需要在配置类中定义为 bean。它们是 RESTful 资源的逻辑分组，一个应用可以包含很多组。

读者可以参考它的文档（http://springfox.github.io/springfox）来了解所有不同的设置方式。

5.11 生成 XML

RESTful API 有时会返回不同媒体类型的响应（JSON、XML 等 。在 Spring 中，负责选择正确媒体类型的机制被称为内容协商（content negotiation）。

默认情况下，在 Spring MVC 中，ContentNegotiatingViewResolver bean 将会按照应用所定义的内容协商策略解析正确的内容。

你可以看一下 ContentNegotiationManagerFactoryBean 类，了解这些策略在 Spring MVC 中是如何运用的。

内容类型可以通过如下的策略来进行解析：

- 按照客户端发送的 Accept 头信息；
- 借助类似于“?format=json”这样的参数；
- 借助路径扩展名，如“/myResource.json”或“/myResource.xml”。

你可以在 Spring 配置中自定义这些策略，这需要重写 `WebMvcConfigurerAdapter` 类的 `configureContentNegotiation()`方法。

默认情况下，Spring 会使用 Accept 头信息和路径扩展名。

要在 Spring Boot 中启用 XML 序列化，可以在类路径中添加如下的依赖：

```
compile 'com.fasterxml.jackson.dataformat:jackson-dataformat-xml'
```

如果你通过浏览器来探测 API，并访问 http://localhost:8080/api/users 的话，将会看到如图 5-5 所示的 XML 结果。

图 5-5

这是因为浏览器通常不会请求 JSON，而 XML 是仅次于 HTML 的第二选择，如图 5-6 所示。

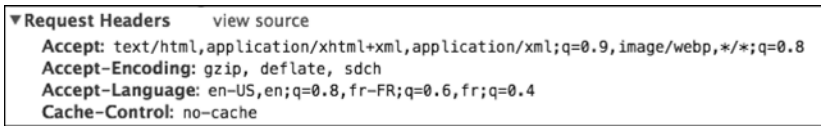

图 5-6

如果想得到 JSON 结果的话，你可以访问 http://localhost:8080/api/users.json，或者通过 Postman 或 httpie 发送适当的 Accept 头信息。

5.12　检查点

在本章中，我们添加了一个搜索 `ApiController` 类。因为 Twitter API 返回的 Tweet 内容并不适合我们使用，所以引入了 `LightTweet` 类，能够将它们转换为更为友好的格式。

我们还开发了用户 API，`User` 类是模型。用户会通过 `UserRepository` 类进行存储和检索，`UserApiController` 类暴露了 HTTP 端点，用于执行针对用户的 CRUD 操作。我们还添加了一个通用的异常和一个映射器，将异常与 HTTP 状态关联了起来。

在配置中，我们添加了一个 bean 来实现 API 的文档化，这要归功于 Swagger，同时还自定义了 JSR-310 日期的序列化。我们的代码结构应该会如图 5-7 所示。

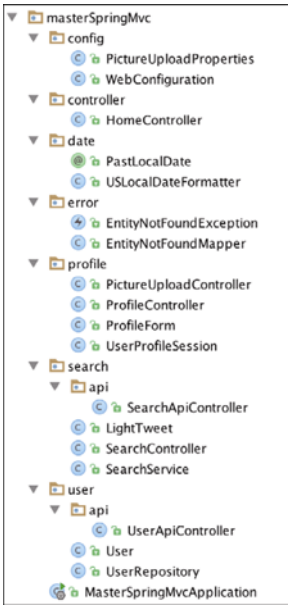

图 5-7

5.13 小结

在本章中，我们看到了如何通过 Spring MVC 创建 RESTful API。这种类型的后端能够在性能和可维护性上带来很大的收益，在与像 Backbone、Angular JS 或 React.js 这样的 JavaScript MVC 框架协作时，能够实现超乎想象的效果。

我们看到了如何恰当地处理错误和异常，并学习了利用 HTTP 状态码实现更好的 API。

最后，我们使用 Swagger 实现了自动化文档的功能，并添加了同时生成 XML 和 JSON 的能力。

在下一章中，我们将会看到如何保护应用，以及如何使用 Twitter API 让我们的用户进行登录。

第 6 章
保护应用

在本章中，我们将会学习如何保护 Web 应用，以及如何应对现代分布式 Web 应用所面临的安全挑战。

本章将会分为 5 个部分：

- 首先，我们会利用几分钟的时间快速搭建基本 HTTP 认证；
- 然后，我们会为 Web 页面设计基于表单的认证，让基本认证继续用于 RESTful API 之中；
- 我们将会允许用户使用 Twitter OAuth API 进行登录；
- 然后，借助 Spring Session 确保应用程序能够扩展到分布式的会话机制中；
- 最后，将会配置 Tomcat 通过 SSL 使用安全的连接。

6.1 基本认证

最为简单的认证机制就是基本认证（http://en.wikipedia.org/wiki/Basic_access_authentication）。简而言之，如果没有用户名和密码的话，我们的应用将不允许访问。

服务器会将我们的资源视为要进行安全保护的，会发送 401 Not Authorized HTTP 状态码并生成 WWW-Authenticate 头信息。

为了成功通过安全检查，客户端必须发送 Authorization 头信息，其中包含 Basic 这个值，然后紧接着是 user:password 格式字符串的 base 64 编码。浏览器窗口将会提示用户输入用户名和密码，如果认证成功的话，将会允许他们访问受保护的页面。

在我们的依赖中添加 Spring Security：

```
compile 'org.springframework.boot:spring-boot-starter-security'
```

重新启动应用并访问应用中的任意 URL，我们会被提示输入用户名和密码，如图 6-1 所示。

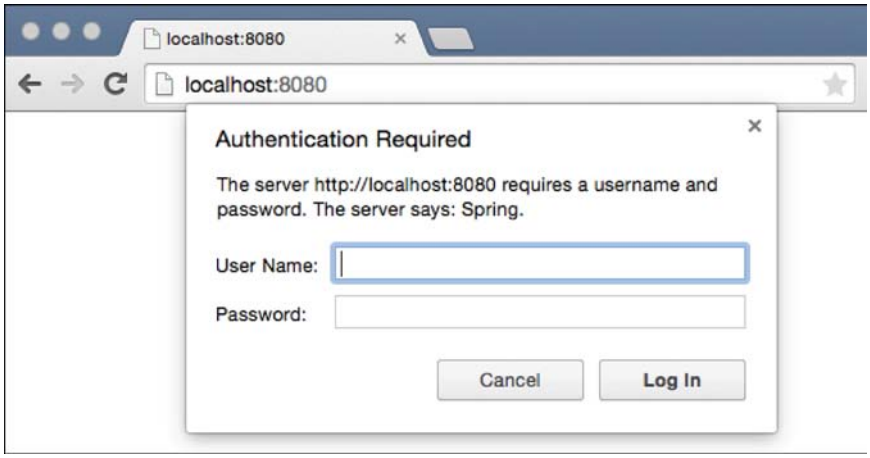

图 6-1

如果认证失败的话，将会看到有 401 错误抛出。默认的用户名是 user，而进行认证的正确密码是应用在每次启动的时候随机生成的，将会展现在服务器的日志中：

```
Using default security password: 13212bb6-8583-4080-b790-103408c93115
```

在默认情况下，Spring Security 将会保护每一项资源，不过，有一些特定的路由会排除在外，如“/css/”“/js/”“/images/”和“**/favicon.ico”。

如果你想配置默认凭证的话，可以添加如下的属性到 application.properties 文件中：

```
security.user.name=admin
security.user.password=secret
```

6.1.1 用户授权

如果我们的应用中只有一个用户的话，那将无法实现细粒度的安全控制。要想在用户的安全凭证方面进行更多控制的话，我们可以在 config 包中添加如下的 SecurityConfiguration 类：

```
package masterSpringMvc.config;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.
builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.method.
configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.
configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    public void configureAuth(AuthenticationManagerBuilder auth)
            throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user").password("user").roles("USER").and()
                .withUser("admin").password("admin").roles("USER",
"ADMIN");
    }
}
```

这个代码片段将会搭建一个内存系统，其中包含了应用程序中的用户及其角色。它会重写之前应用属性中所定义的安全用户名和密码。

@EnableGlobalMethodSecurity 注解将会允许我们为应用中的类和方法添加注解，从而定义它们的安全级别。

例如，假设只有应用中的管理员才能访问用户 API，那么，我们只需在资源上添加 @Secured 注解即可，这样的话，只有具备 ADMIN 角色的用户才能进行访问：

```
@RestController
@RequestMapping("/api")
@Secured("ROLE_ADMIN")
public class UserApiController {
  // ... code omitted
}
```

我们借助 httpie 可以很容易地进行测试，这里使用“-a”开关来启用基本认证，并且使用了“-p=h”开关，这样的话就能只显示响应的头信息。

我们先试一下不具备管理员（admin）角色的用户：

```
> http GET 'http://localhost:8080/api/users' -a user:user -p=h
HTTP/1.1 403 Forbidden
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json;charset=UTF-8
Date: Sat, 23 May 2015 17:40:09 GMT
Expires: 0
Pragma: no-cache
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=2D4761C092EDE9A4DB91FA1CAA16C59B; Path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

然后，再试一下管理员角色的用户：

```
> http GET 'http://localhost:8080/api/users' -a admin:admin -p=h
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json;charset=UTF-8
Date: Sat, 23 May 2015 17:42:58 GMT
Expires: 0
Pragma: no-cache
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=CE7A9BF903A25A7A8BAD7D4C30E59360; Path=/; HttpOnly
Transfer-Encoding: chunked
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

你可能也注意到了，Spring Security 会自动添加一些通用的安全头信息。

- Cache-Control：防止用户缓存受保护的资源。
- X-XSS-Protection：告知浏览器阻止类似于 XSS 的攻击。
- X-Frame-Options：不允许站点嵌入到 IFrame 之中。
- X-Content-Type-Options：这会禁止浏览器猜测恶意资源的 MIME 类型，这些资源会用来进行 XSS 攻击。

关于这些头信息的完备列表，可以参考：
http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#headers。

6.1.2　URL 授权

为控制器添加注解是非常简单的，但这往往并不是最可行的方案。有时候，我们会想要完全控制授权功能。

移除@Secured 注解，我们将会采用一种更好的方案。

通过修改`SecurityConfiguration`类，我们看一下 Spring Security 允许实现什么功能：

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    public void configureAuth(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("user").roles("USER").and()
            .withUser("admin").password("admin").roles("USER",
"ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .httpBasic()
            .and()
            .csrf().disable()
            .authorizeRequests()
            .antMatchers("/login", "/logout").permitAll()
            .antMatchers(HttpMethod.GET, "/api/**").hasRole("USER")
            .antMatchers(HttpMethod.POST, "/api/**").hasRole("ADMIN")
            .antMatchers(HttpMethod.PUT, "/api/**").hasRole("ADMIN")
            .antMatchers(HttpMethod.DELETE, "/api/**").
hasRole("ADMIN")
            .anyRequest().authenticated();
    }
}
```

在前面的示例代码中，我们使用 Spring Security 流畅的 API，配置了应用程序的安全策略。

这个 API 通过调用相关安全功能的方法，能够在全局上配置 Spring Security，这些方法可以通过`and()`连接起来。

我们刚刚定义的是一个基本认证，没有 CSRF 保护功能。任何用户都允许发送针对“/login”和“/logout”的请求。对 API 的 GET 请求只允许具有 USER 角色的用户访问，而

对 API 的 POST、PUT 和 DELETE 请求则只允许具有 ADMIN 角色的用户访问。最后，对其他内容的访问则要求用户进行过认证，不管角色是什么，都可以进行访问。

CSRF 所代表的含义是跨站请求伪造（Cross Site Request Forgery），指的是一种攻击形式，恶意的 Web 站点将会在他们的网站上展示一个表单，但是会将表单的数据 POST 提交到我们的站点上。如果我们站点的用户没有退出的话，那么 POST 请求能够获取用户的 cookie，这样的话，就会被认为是认证过的用户。

针对 CSRF 的防护会生成短期存活的 token，它会随着表单数据一起进行提交。我们会在下面的章节看到如何启用该功能，现在我们先将其禁用。参考 http://docs.spring.io/spring-security/site/docs/current/ reference/htmlsingle/#csrf 来了解更多细节。

如果想了解针对 API 请求进行认证的更多知识，可以参考：http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#authorize-requests。

6.1.3 Thymeleaf 安全标签

有时候，我们需要展现来自认证层的数据，例如用户的名称和角色，或者根据用户的权限隐藏或显示 Web 页面的部分内容。借助 thymeleaf-extras-springsecurity 模块，我们就能实现这些功能。

在 build.gradle 文件中添加如下的依赖：

```
compile 'org.thymeleaf.extras:thymeleaf-extras-springsecurity3'
```

借助这个库，我们可以在“layout/ default.html”页面的导航栏下面添加一段代码，展现登录的用户：

```
<!DOCTYPE html>
<html xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extrasspringsecurity3">
<head>
  <!-- content trimmed -->
</head>
<body>

<!-- content trimmed -->
<nav>
    <div class="nav-wrapper indigo">
```

```
            <ul class="right">
            <!-- content trimmed -->
            </ul>
        </div>
    </nav>
    <div>
        You are logged as <b sec:authentication="name" /> with roles <span
    sec:authentication="authorities" />
        -
        <form th:action="@{/logout}" method="post" style="display: inlineblock">
            <input type="submit" value="Sign Out" />
        </form>
        <hr/>
    </div>

    <section layout:fragment="content">
        <p>Page content goes here</p>
    </section>

    <!-- content trimmed -->
    </body>
    </html>
```

注意，在 HTML 声明中的新命名空间以及 sec:authentication 属性。它允许我们访问 org.springframework.security.core.Authentication 对象的属性，这个对象代表了当前登录的用户，如图 6-2 所示。

图 6-2

现在不要点击退出按钮，因为在基本认证中，它是无法正常运行的。我们将会在下一部分使其能够发挥作用。

这个标签库还有很多其他可用的标签，例如用来检查用户权限的：

```
<div sec:authorize="hasRole('ROLE_ADMIN')">
    You are an administrator
</div>
```

请参考 https://github.com/thymeleaf/thymeleaf-extras-springsecurity 上的文档，来了解关于这个库的更多细节。

6.2 登录表单

对于 RESTful API 来说，基本认证是很好的方式，但是我们更愿意使用一个团队精心设计的登录页面，从而提升 Web 的用户体验。

Spring Security 允许定义任意数量的 `WebSecurityConfigurerAdapter` 类，我们会将 `SecurityConfiguration` 类拆分为两部分。

- ApiSecurityConfiguration：这个类会优先配置，它会使用基本认证来保护 RESTful 端点。
- WebSecurityConfiguration：这个类会接着为应用的其他功能配置登录表单。

你可以移除或重命名 SecurityConfiguration，并创建 ApiSecurityConfiguration 来替代它：

```
@Configuration
@Order(1)
public class ApiSecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Autowired
    public void configureAuth(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("user").roles("USER").and()
            .withUser("admin").password("admin").roles("USER",
"ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .antMatcher("/api/**")
            .httpBasic().and()
            .csrf().disable()
            .authorizeRequests()
            .antMatchers(HttpMethod.GET).hasRole("USER")
```

```
                .antMatchers(HttpMethod.POST).hasRole("ADMIN")
                .antMatchers(HttpMethod.PUT).hasRole("ADMIN")
                .antMatchers(HttpMethod.DELETE).hasRole("ADMIN")
                .anyRequest().authenticated();
        }
    }
```

请注意@Order(1)注解，它会确保这个配置的执行会早于其他配置。然后，为 Web 功能创建第二个配置，名为 WebSecurityConfiguration；

```
package masterSpringMvc.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.
HttpSecurity;
import org.springframework.security.config.annotation.web.
configuration.WebSecurityConfigurerAdapter;

@Configuration
public class WebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .formLogin()
                .defaultSuccessUrl("/profile")
                .and()
                .logout().logoutSuccessUrl("/login")
                .and()
                .authorizeRequests()
                .antMatchers("/webjars/**", "/login").permitAll()
                .anyRequest().authenticated();
    }
}
```

这个代码的结果就是所有匹配“/api/**”的 URL 将会通过基本认证进行保护，并且没有 CSRF 防护功能。然后，将会加载第二项配置，它会保护其他的内容。除了对 WebJars 和登录页面（这会避免重定向循环）的请求以外，应用中的其他内容都会要求客户端进行认证。

如果没有经过认证的用户试图访问受保护的资源，他们将会自动重定向到登录页面。

默认情况下，登录 URL 是 GET 方法的“/login”。默认的登录将会通过 POST 方法的

“/login”请求进行提交，这个请求中将会包含 3 个值：用户名（username）、密码（password）以及 CSRF token（_csrf）。如果登录不成功的话，用户将会重定向到“/login?error”。默认的退出页面是对“/logout”的 POST 请求，会带有 CSRF token。

现在，如果你试图访问应用的话，将会自动生成该表单，如图 6-3 所示。

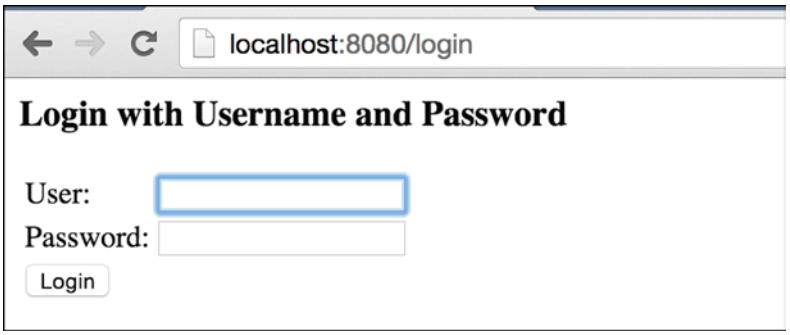

图 6-3

如果已经登录过的话，关闭浏览器，将会清空会话。

现在，我们可以登录和退出应用了。

这看起来已经很不错了，但是我们稍微再下点功夫就能做得更好。首先，我们将会定义一个映射到“/login”的登录页，这需要在 WebSecurityConfiguration 类中配置：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .formLogin()
        .loginPage("/login") // <= custom login page
        .defaultSuccessUrl("/profile")
        // the rest of the configuration stays the same
}
```

这样的话，我们就能创建自己的登录页了。为了实现该功能，需要一个非常简单的控制器，用它来处理 GET 方法的 login 请求。我们在 authentication 包中创建该控制器：

```
package masterSpringMvc.authentication;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class LoginController {
```

```
    @RequestMapping("/login")
    public String authenticate() {
        return "login";
    }
}
```

这样就会展现位于模板目录下的 login.html 页面。现在，我们来创建这个页面：

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head>
    <title>Login</title>
</head>
<body>
<div class="section no-pad-bot" layout:fragment="content">
    <div class="container">

        <h2 class="header center orange-text">Login</h2>

        <div class="row">
            <div id="errorMessage" class="card-panel red lighten-2"
th:if="${param.error}">
                <span class="card-title">Invalid user name or
password</span>
            </div>

            <form class="col s12" action="/login" method="post">
                <div class="row">
                    <div class="input-field col s12">
                        <input id="username" name="username"
type="text" class="validate"/>
                        <label for="username">Username</label>
                    </div>
                </div>
                <div class="row">
                    <div class="input-field col s12">
                        <input id="password" name="password"
type="password" class="validate"/>
                        <label for="password">Password</label>
                    </div>
                </div>
```

```
                <div class="row center">
                    <button class="btn waves-effect waves-light"
type="submit" name="action">Submit
                        <i class="mdi-content-send right"></i>
                    </button>
                </div>
                <input type="hidden" th:name="${_csrf.parameterName}"
th:value="${_csrf.token}"/>
            </form>
        </div>
    </div>
</div>
</body>
</html>
```

注意，我们在这里处理了错误信息，并且还会提交一个 CSRF token。用户名和密码的输入域名称都使用了默认值，但是如果需要的话，它们都是可配置的。如图 6-4 所示，这样页面看起来已经好多了！

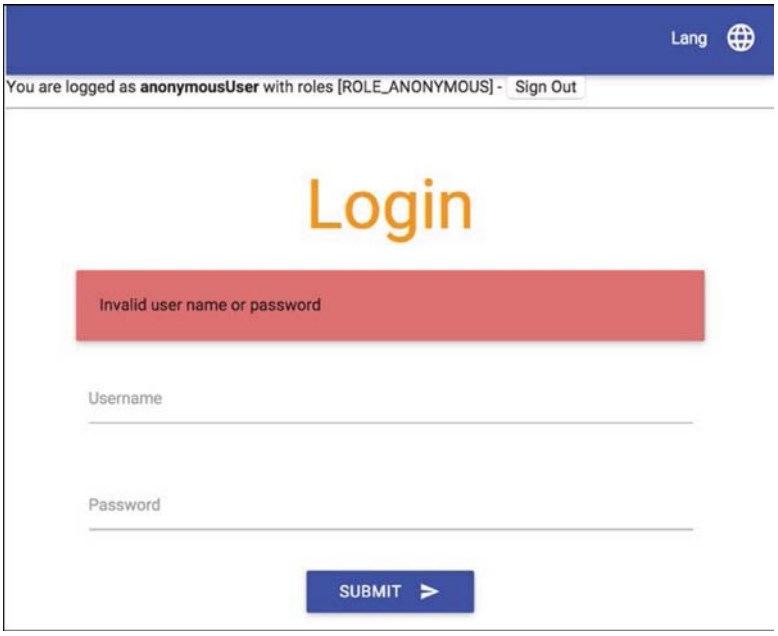

图 6-4

我们从这里可以看到，Spring Security 默认将会为未认证的用户分配一个匿名凭证。

对于匿名用户来说，不应该显示退出按钮，所以我们可以将对应的 HTML 部分用 sec:

authorize="isAuthenticated()"包装起来，这样的话，只有认证过的用户才会显示，代码如下所示：

```
<div sec:authorize="isAuthenticated()">
    You are logged as <b sec:authentication="name"/> with roles <span
sec:authentication="authorities"/>
    -
    <form th:action="@{/logout}" method="post" style="display: inlineblock">
        <input type="submit" value="Sign Out"/>
    </form>
    <hr/>
</div>
```

6.3　Twitter 认证

我们的应用与 Twitter 是紧密集成的，如果能够通过 Twitter 来实现登录的话，是完全合情合理的。

在采取进一步操作之前，确保已经在 Twitter 上（https://apps.twitter.com/）为我们的应用启用了 Twitter 登录功能，如图 6-5 所示。

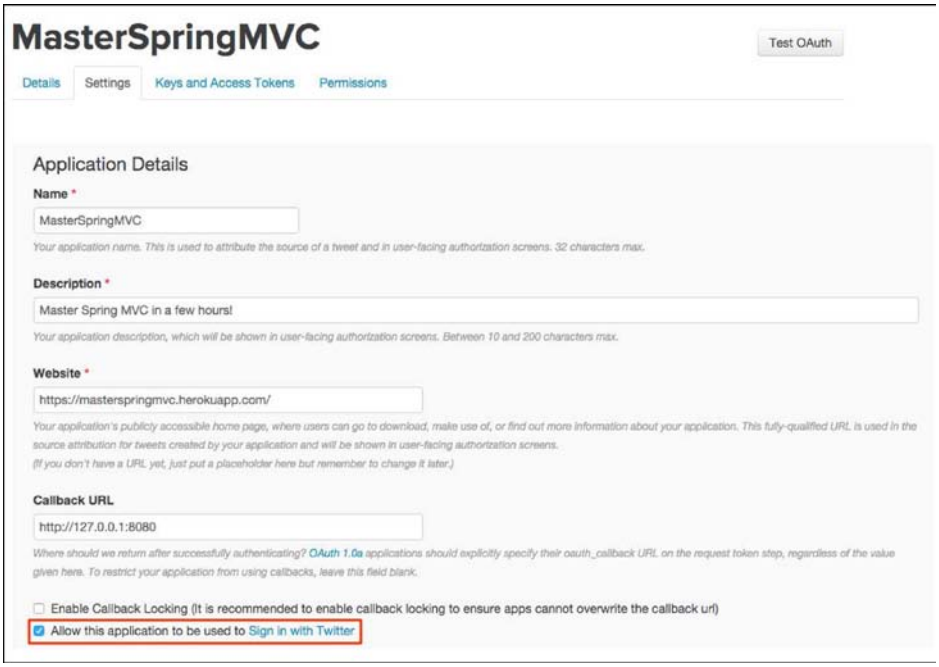

图 6-5

6.3.1 搭建社交认证环境

Spring Social 能够借助 Twitter 这样的 OAuthprovider 实现认证功能，这需要使用一个登录（signin）/注册（signup）场景来完成。它将会拦截针对“/signin/twitter”的 POST 请求。如果用户还没有记录到 UsersConnectionRepository 接口中，将会调用 signup 端点。它允许我们采取一些必要的措施实现用户在我们系统中的注册，在这个过程中，可能会询问用户一些额外的细节信息。

那就开工吧，我们需要做的第一件事就是将“signin/**”和“/signup”这两个 URL 作为公开可访问的资源，只需修改 WebSecurityConfiguration 类的 permitAll 一行：

```
.antMatchers("/webjars/**", "/login", "/signin/**", "/signup").
permitAll()
```

为了启用登录/注册场景，我们需要添加一个 SignInAdapter 接口，这是一个简单的监听器，当已知的用户再次登录时，它将会被调用。

我们在 LoginController 控制器所在的包下创建 AuthenticatingSignInAdapter 类：

```
package masterSpringMvc.authentication;

import org.springframework.security.authentication.
UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.
SecurityContextHolder;
import org.springframework.social.connect.Connection;
import org.springframework.social.connect.UserProfile;
import org.springframework.social.connect.web.SignInAdapter;
import org.springframework.stereotype.Component;
import org.springframework.web.context.request.NativeWebRequest;

@Component
public class AuthenticatingSignInAdapter implements SignInAdapter {

    public static void authenticate(Connection<?> connection) {
        UserProfile userProfile = connection.fetchUserProfile();
        String username = userProfile.getUsername();
        UsernamePasswordAuthenticationToken authentication = new Usern
amePasswordAuthenticationToken(username, null, null);
        SecurityContextHolder.getContext().setAuthentication(authenti
cation);
```

```
        System.out.println(String.format("User %s %s connected.",
    userProfile.getFirstName(), userProfile.getLastName()));
    }

    @Override
    public String signIn(String userId, Connection<?> connection,
    NativeWebRequest request) {
        authenticate(connection);
        return null;
    }
}
```

可以看到，这个处理器会在用户通过 Spring Security 进行认证之时调用。我们稍后还会继续看这个类。不过，我们现在需要在相同的包中定义 `SignupController` 类，这个类会用于第一次进行访问的用户：

```
package masterSpringMvc.authentication;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.social.connect.Connection;
import org.springframework.social.connect.ConnectionFactoryLocator;
import org.springframework.social.connect.UsersConnectionRepository;
import org.springframework.social.connect.web.ProviderSignInUtils;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.context.request.WebRequest;

@Controller
public class SignupController {
    private final ProviderSignInUtils signInUtils;

    @Autowired
    public SignupController(ConnectionFactoryLocat
    or connectionFactoryLocator, UsersConnectionRepository
    connectionRepository) {
        signInUtils = new ProviderSignInUtils(connectionFactoryLocat
    or, connectionRepository);
    }

    @RequestMapping(value = "/signup")
    public String signup(WebRequest request) {
        Connection<?> connection = signInUtils.getConnectionFromSessi
    on(request);
        if (connection != null) {
```

```
                AuthenticatingSignInAdapter.authenticate(connection);
                signInUtils.doPostSignUp(connection.getDisplayName(),
request);
            }
            return "redirect:/profile";
        }
    }
```

首先，控制器会从会话中获取当前连接。然后，它会采用与之前相同的方法来认证用户。最后，会触发 doPostSignUp 事件，这样 Spring Social 会将用户的信息存储到之前所述的 UsersConnectionRepository 接口中。

我们需要做的最后一件事情就是添加一个“Connect With Twitter”按钮到登录页面中（见图 6-6），就在之前的表单下面：

```
<form th:action="@{/signin/twitter}" method="POST" class="center">
    <div class="row">
        <button class="btn indigo" name="twitterSignin"
type="submit">Connect with Twitter
            <i class="mdi-social-group-add left"></i>
        </button>
    </div>
</form>
```

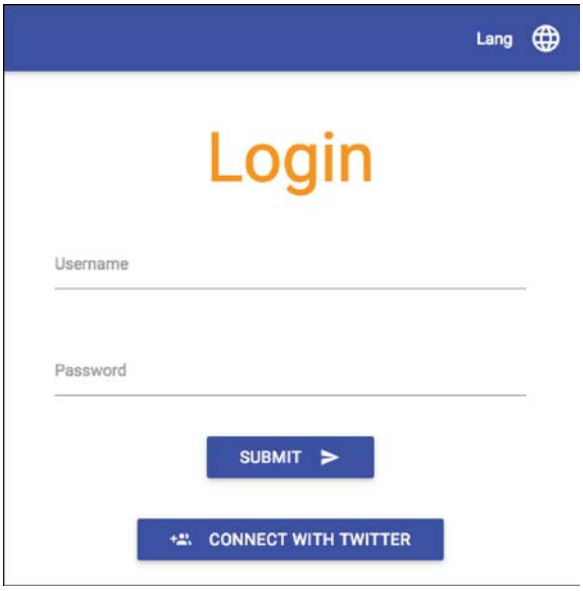

图 6-6

当用户点击“CONNECT WITH TWITTER”按钮的时候，他们将会被重定向到一个 Twitter 登录页面，如图 6-7 所示。

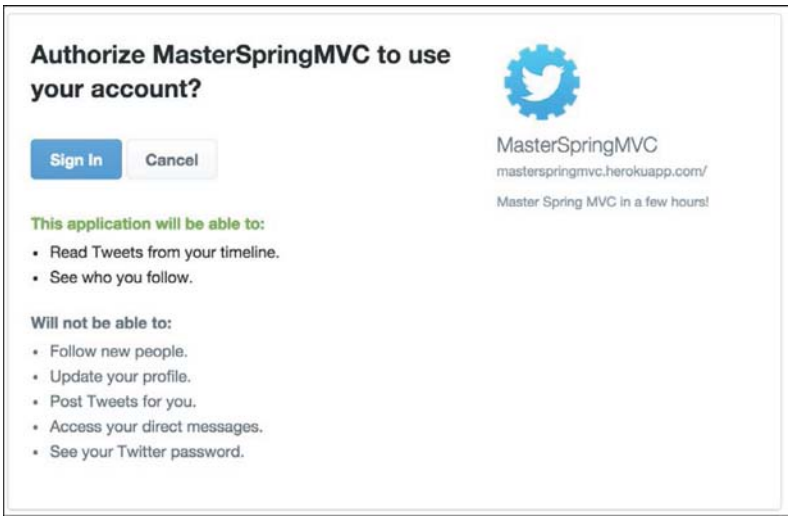

图 6-7

6.3.2　详解

这里并没有太多的代码，但是要理解所有的组成部分还是需要点技巧。我们的第一步是看一下 Spring Boot 的 `SocialWebAutoConfiguration` 类。

这个类中所定义的 SocialAutoConfigurationAdapter 包含如下的 bean：

```
@Bean
@ConditionalOnBean(SignInAdapter.class)
@ConditionalOnMissingBean(ProviderSignInController.class)
public ProviderSignInController signInController(
        ConnectionFactoryLocator factoryLocator,
        UsersConnectionRepository usersRepository, SignInAdapter
signInAdapter) {
    ProviderSignInController controller = new
ProviderSignInController(
            factoryLocator, usersRepository, signInAdapter);
    if (!CollectionUtils.isEmpty(this.signInInterceptors)) {
        controller.setSignInInterceptors(this.signInInterceptors);
    }
    return controller;
}
```

如果在配置中能够探测到 ProviderSignInController 类的话，将会自动创建 ProviderSignInController 类。这个控制器是登录过程的核心。下面看一下它都完成了什么工作（我在这里只会概述重要的部分）：

- 它将会处理连接按钮所发起的针对“/signin/{providerId}”的 POST 请求；
- 它会将用户重定向到认证 provider 所对应的登录 URL 上；
- 它会通过对“/signin/{providerId}”的 GET 请求得到认证 provider 的 OAuth token；
- 然后，它会处理登录；
- 如果没有在 UsersConnectionRepository 接口中找到用户的话，它会使用一个 SessionStrategy 接口来存储正在进行中的登录请求，并且会重定向到 signupUrl 页面；
- 如果能够找到用户的话，会调用 SignInAdapter 接口，用户会被重定向到 postSignupUrl 页面。

这个识别过程有两个重要的组件，其中 UsersConnectionRepository 接口负责从特定类型的存储中保存和检索用户信息，SessionStrategy 接口会临时保存用户连接，这样通过 SignupController 就能获取到相关的信息。

默认情况下，Spring Boot 会为每个认证 provider 创建一个 InMemoryUsersConnectionRepository，这就意味着我们的用户连接是保存在内存中的。如果重启服务器的话，用户的状态就会变成未知的，需要再走一遍登录的流程。

ProviderSignInController 类默认使用的是 HttpSessionSessionStrategy，它会将连接保存在 HTTP 会话中。我们在 SignupController 类中所使用的 ProviderSignInUtils 类默认也采用了这种策略。如果我们将应用分布式部署到多台服务器上的话，这就会产生问题，因为可能不会每台服务器上都能使用该会话。

我们可以通过为 ProviderSignInController 和 ProviderSignInUtils 提供自定义的 SessionStrategy 接口来改变这种默认行为，从而让数据存储到其他位置，而不是存储在 HTTP 会话中。

与之类似，我们也可以提供另外一个 UsersConnectionRepository 接口的实现，从而将用户连接数据保存到其他类型的存储中。

Spring Social 提供了一个 JdbcUsersConnectionRepository，它会自动将认证过的用户保存到数据库的 UserConnection 表中。在本书中不会展开讨论这个话题，不过你可以非常容易地对其进行配置，只需在配置类中添加如下的 bean 即可：

```
@Bean
@Primary
public UsersConnectionRepository getUsersConnectionRepository(
  DataSource dataSource, ConnectionFactoryLocator
connectionFactoryLocator) {
    return new JdbcUsersConnectionRepository(
      dataSource, connectionFactoryLocator, Encryptors.noOpText());
}
```

最佳方法

读者可以参考我的博客中这篇文章来了解更多细节：http://geowarin.github.io/spring/2015/08/02/social-login-with-spring.html。

6.4　分布式会话

正如我们在前面的小节所描述的，在很多地方 Spring Social 都会将内容保存在 HTTP 会话中。我们的用户基本信息也是保存在会话中的。这是一种典型的实现方式，只要用户还在访问站点，其相关的内容就会保存在内存之中。

但是，如果我们想要对应用进行扩展，并将负载分布到后端的多台服务器的话，这就会带来一定的问题。我们现在已经步入了云时代，在第 9 章中，将会讨论如何将应用部署到云端。

为了让会话能够适应分布式环境，有以下几种方案：

- 我们可以使用粘性会话（sticky session），它能够确保特定用户会被转移到同一台服务器上，并保持其会话。这种方案需要部署时的额外配置，不是特别优雅的方式；
- 重构我们的代码，将数据保存在数据库中，而不是保存在会话之中。这样的话，如果需要将其与客户端请求中的 cookie 或 token 相关联，我们可以每次都从数据库中加载数据；
- 使用 Spring Session 项目，从而能够透明地使用像 Redis 这样的分布式数据库作为底层的会话 provider。

在本章中，我们将会看一下如何使用第三种方式。搭建过程很简单，不过所带来的效益是非常可观的，而且我们可以在不影响应用功能的情况下将其关闭。

我们所要做的第一件是安装 Redis，如果要在 Mac 下安装的话，可以使用 brew 命令：

```
brew install redis
```

对于其他的平台，可以遵循 http://redis.io/download 站点的安装指南。

通过使用如下的命令，我们可以启动 Redis 服务器：

```
redis-server
```

添加如下的依赖到 build.gradle 文件中：

```
compile 'org.springframework.boot:spring-boot-starter-redis'
compile 'org.springframework.session:spring-session:1.0.1.RELEASE'
```

在与 application.properties 相同的目录下，新创建一个名为 application-redis.properties 的配置文件：

```
spring.redis.host=localhost
spring.redis.port=6379
```

Spring Boot 提供了一种便利的方式将配置文件与 profile 关联起来，只有当 redis profile 激活的时候，application-redis.properties 才会被加载。

然后，在 config 包中，我们创建 RedisConfig 类：

```
package masterSpringMvc.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.session.data.redis.config.annotation.web.
http.EnableRedisHttpSession;

@Configuration
@Profile("redis")
@EnableRedisHttpSession
public class RedisConfig {
}
```

可以看到，只有 redis profile 激活的时候，这个配置才会生效。

我们已经准备就绪了，现在可以通过如下的标记来启动应用：

```
-Dspring.profiles.active=redis
```

你也可以通过 gradlew build 生成 JAR 文件，并使用如下的命令启动应用：

```
java -Dserver.port=$PORT -Dspring.profiles.active=redis -jar app.jar
```

另外，我们还可以在 Bash 中通过 Gradle 来启动，如下所示：

```
SPRING_PROFILES_ACTIVE=redis ./gradlew bootRun
```

我们也可以在 IDE 的运行配置中将其作为 JVM 参数，从而在 IDE 中启动应用。

这样就可以了，现在我们有了一个服务器来存储登录用户的详细信息。这意味着我们可以进行横向扩展，为 Web 资源提供更多的服务器，而我们的用户根本不会感知到这一切。就我们而言，根本不需要编写任何的代码。

这同时意味着即便我们重启服务器，也能继续保持用户的会话。

如果要看一下它是如何运行的，可以通过 redis-cli 命令来连接 Redis。开始的时候，它不包含任何的 key：

```
> redis-cli
127.0.0.1:6379> KEYS *
(empty list or set)
```

切换至我们的应用并将一些内容保存到会话之中，然后再看 Redis 中的数据：

```
127.0.0.1:6379> KEYS *
1) "spring:session:expirations:1432487760000"
2) "spring:session:sessions:1768a55b-081a-4673-8535-7449e5729af5"
127.0.0.1:6379> HKEYS spring:session:sessions:1768a55b-081a-4673-8535-
7449e5729af5
1) "sessionAttr:SPRING_SECURITY_CONTEXT"
2) "sessionAttr:org.springframework.security.web.csrf.
HttpSessionCsrfTokenRepository.CSRF_TOKEN"
3) "lastAccessedTime"
4) "maxInactiveInterval"
5) "creationTime"
```

我们可以查阅如下站点的列表，了解所有可用的命令：http://redis.io/commands。

6.5　SSL

安全套接字层（Secure Sockets Layer，SSL）是一个安全协议，根据该协议数据会通过

证书进行加密并发往可信任的参与方。在这一部分中，我将会阐述使用 Spring Boot 创建安全连接的不同方式。对于后续的章节来说，这一部分并不是必需的。包含这些内容更多的是出于完整性的考虑，如果你急切地想看到如何将应用部署到云端的话，大可以略过这一节进入后面的内容。

在第 9 章中我们将会看到大多数的云平台已经处理了 SSL 相关的事宜，所以没有必要在我们这一端进行配置。

6.5.1 生成自签名的证书

正常情况下，X.509 证书是由数字证书认证机构（Certificate Authority）来颁发的。他们一般会对这种服务进行收费，作为测试来讲，我们可以创建自签名的 keystore 文件。

JDK 自带了名为 keytool 的二进制文件，它是用来管理证书的。借助这个工具，我们可以创建 keystore 并将证书导入已有的 keystore 中。我们在项目的根目录下，执行如下的命令来创建 keystore：

```
$ keytool -genkey -alias masterspringmvc -keyalg RSA -keystore src/main/
resources/tomcat.keystore
Enter keystore password: password
Re-enter new password: password
What is your first and last name?
  [Unknown]: Master Spring MVC
What is the name of your organizational unit?
  [Unknown]: Packt
What is the name of your organization?
  [Unknown]: Packt
What is the name of your City or Locality?
  [Unknown]: Paris
What is the name of your State or Province?
  [Unknown]: France
What is the two-letter country code for this unit?
  [Unknown]: FR
Is CN=Master Spring MVC, OU=Packt, O=Packt, L=Paris, ST=France, C=FR
correct?
  [no]: yes
Enter key password for <masterspringmvc>
        (RETURN if same as keystore password): password2
Re-enter new password: password2
```

这样就会基于 RSA 算法生成一个名为 masterspringmvc 的 keystore，它会保存到

src/main/resources 下的一个 keystore 中。

不要将 keystore 放到仓库中，它可能会被暴力破解，从而使 Web 站点的安全保护失效。我们在生成 keystore 的时候，应该采用随机生成的强密码。

6.5.2　单一模式

如果你希望只有安全的 https 通道而没有 http 通道的话，那么这非常容易实现：

```
server.port = 8443
server.ssl.key-store = classpath:tomcat.keystore
server.ssl.key-store-password = password
server.ssl.key-password = password2
```

不要将密码放到仓库中，要使用“${}”符号来导入环境变量。

6.5.3　双通道模式

如果你想要在应用中同时使用 http 和 https 通道的话，那么需要在应用中增加如下的配置：

```
@Configuration
public class SslConfig {

    @Bean
    public EmbeddedServletContainerFactory servletContainer() throws
IOException {
        TomcatEmbeddedServletContainerFactory tomcat = new
TomcatEmbeddedServletContainerFactory();
        tomcat.addAdditionalTomcatConnectors(createSslConnector());
        return tomcat;
    }

    private Connector createSslConnector() throws IOException {
        Connector connector = new Connector(Http11NioProtocol.class.
getName());
        Http11NioProtocol protocol =
                (Http11NioProtocol) connector.getProtocolHandler();
        connector.setPort(8443);
```

```
            connector.setSecure(true);
            connector.setScheme("https");
            protocol.setSSLEnabled(true);
            protocol.setKeyAlias("masterspringmvc");
            protocol.setKeystorePass("password");
            protocol.setKeyPass("password2");
            protocol.setKeystoreFile(new ClassPathResource("tomcat.
    keystore").getFile().getAbsolutePath());
            protocol.setSslProtocol("TLS");
            return connector;
        }
    }
```

这样的话，除了 8080 端口以外，还会加载前文生成的 keystore 并在 8443 创建另外一个通道。

我们可以通过如下的配置，使用 Spring Security 自动将连接从 http 重定向到 https：

```
@Configuration
public class WebSecurityConfiguration extends
WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requiresChannel().anyRequest().requiresSecure()
            .and()
            /* rest of the configuration */;
    }
}
```

6.5.4 置于安全的服务器之后

通过 SSL 保护应用的最简便方式通常是将应用部署到启用 SSL 功能的 Web 服务器后面 如 Apache 或 CloudFlare。这样的话，通常会使用头信息来表明这个连接最初是使用 SSL 初始化的。

如果我们在 application.properties 中指定正确的头信息是什么样子的，那么 Spring Boot 就能够理解这种协议

```
server.tomcat.remote_ip_header=x-forwarded-for
server.tomcat.protocol_header=x-forwarded-proto
```

读者可以参考如下的文档来了解更多的细节：
http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html#howto-use- tomcat-behind-a-proxy-server。

6.6　检查点

在本章中，我们添加了 3 项配置：ApiSecurityConfiguration 负责配置 REST API 使用基本 HTTP 认证；WebSecurityConfiguration 为 Web 用户搭建了一个登录表单，用户从而可以使用账号或 Twitter 来进行登录；RedisConfig 允许我们将会话信息存储到 Redis 服务器中。

如图 6-8 所示，在 authentication 包中，我们添加了 `LoginController` 类，它会重定向到登录页面；当用户第一次通过 Twitter 登录的话，将会调用 SignupController；用户每次通过 Twitter 登录的时候，都会调用 AuthenticatingSignInAdapater。

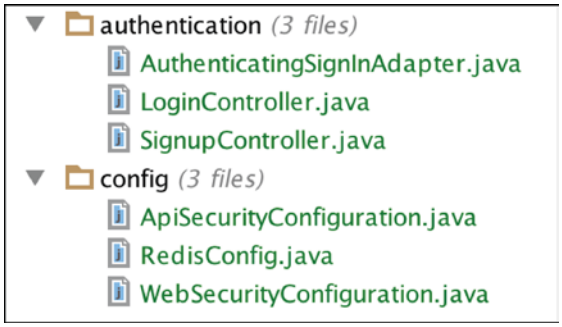

图 6-8

6.7　小结

通过 Spring 来保护 Web 应用是非常简单的，这里面的可能性是无限的，例如社交登录方面的高级配置，读者可以自行探索。分布式会话以及扩展性也值得花点时间去研究。

在下一章中，我们将会对应用进行测试，保证它不会出现问题。

第 7 章
单元测试与验收测试

在本章中，将会看到为何要对应用进行测试，以及该如何开展测试，会了解单元测试与验收测试的区别，并学习这两种测试方式。

本章分为两部分。在第一部分中，将会使用 Java 来编写测试并学习测试的不同方式，第二部分会简短一些，将会使用 Groovy 来编写完全相同的测试，看一下如何使用这个很酷的语言来提升代码的可读性。

如果按照本章的进度来完成每项内容的话，会拥有两份测试，所以完全可以只保留自己最易读的那一份测试。

7.1 为什么要测试我的代码

Java 领域的工作使得很多的开发人员意识到了测试的重要性。一系列好的测试能够让我们提前发现问题，当签发产品的时候，我们就会更加地自信。

很多人现在已经非常熟悉持续集成（http://www.thoughtworks.com/continuous-integration）的理念了。按照这种实践，当源码控制系统发生变更的时候，会有一台服务器来对应用程序进行构建。

构建要越快越好，而且要能够进行测试。这种实践的主要理念在于尽快得到反馈，当系统出现问题的时候，你能够尽可能快速地了解哪里出现了故障，其详情是什么。

那究竟为什么要关心这些呢？毕竟，对应用进行测试会带来额外的成本，设计和维护测试所耗费的时间会侵占开发的时间。

实际上，bug 发现得越晚，它的成本就越高。我们可以想一下，即便 bug 是由 QA 团队发现的，它的成本也要比自己发现要高。这样的话，需要我们切换到编写代码的上下文

中：我为什么要编写这行代码？这个函数的底层业务规则是什么？

如果你事先编写测试并且能够在几秒的时间就将其运行起来的话，在解决代码中潜在的 bug 方面，这种方式所消耗的时间肯定会更少。

测试的另外一项好处在于它可以作为代码的实时文档。丰富的文档甚至代码注释都可能会失效，因为它们很容易过期，我们要为极限情况和意外行为编写良好的测试，形成这样的习惯能够为未来编织一道安全网。

这行代码有什么用处？你曾经问过自己这样的问题吗？如果你有良好的单元测试集的话，你可以将其移除掉，然后看看哪里出错了！在代码质量和重构能力方面，测试能够给我们前所未有的自信。软件是非常脆弱的，如果你停止关注的话，它就会慢慢地腐烂和消亡。

负起你的责任——别让你的代码消亡！

7.2　该如何测试自己的代码

针对一款软件，会有不同的测试方式，如安全测试、性能测试等。作为开发人员，将会关注如何能够自动化并且有助于提升代码质量的测试。

测试分为两大类：单元测试和验收测试。测试金字塔（http://martinfowler.com/bliki/TestPyramid.html）展现了这些测试应该占据的比例。

在金字塔的底部，是单元测试（能够快速执行且相对易于维护），在顶部是 UI 测试（成本更高并且执行更慢）。集成测试位于二者之间：它们可以视为更大的单元测试，需要单元之间的复杂交互。

这个金字塔的理念是提醒我们要将焦点放在最有影响力并且能够得到最佳反馈的地方。

7.3 测试驱动开发

很多开发人员保持了测试驱动开发（Test-driven Development，TDD）的好习惯。这种实践来源于极限编程（Extreme Programming，XP），它会将每个开发阶段拆分为很小的步骤，并为每个步骤编写一个会失败的测试。然后对代码进行修改，让测试再次通过（测试变成绿色）。你可以对代码进行重构，只要能够保证测试是绿色的就可以。图 7-1 阐述了 TDD 的生命周期。

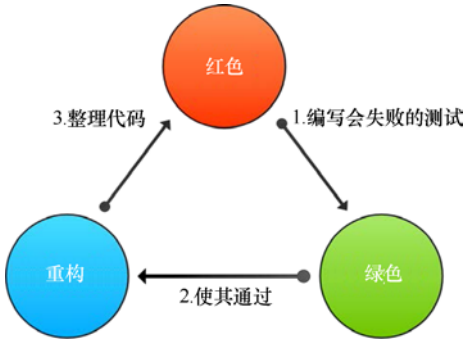

图 7-1

开发人员可以通过非常短的反馈循环最终将特性完成，这样能够确保不会出现问题，所编写的代码从一开始就是经过测试的。

TDD 也受到了一些批评，其中最有意思的如下所示：

- 编写测试所消耗的时间比编写实际实现所消耗的时间还多；
- 它能够导致设计很差的应用程序。

实际上，要成为良好的 TDD 从业者需要一定的时间。一旦读者能够感知到哪些内容需要测试并且对工具有了很好的了解，这并不会耗费太多的时间。

我们需要有经验的开发人员使用 TDD（或其他的方法论）对应用程序进行恰当的设计。如果局限于很小的步骤，而没有看到整体大局的话，TDD 会导致糟糕的设计。TDD 不会像有魔法那样自动产生设计优良的应用，所以要多加小心，在完成每一项特性之后，都要回过头来审视一遍。

从本书开始至今，代码中只有一个自动生成的单元测试。这太糟糕了！这是因为没有遵循最佳实践，本章就会解决这个问题。

7.4　单元测试

我们所能编写的较低层次的测试被称为单元测试。单元测试应该测试很小的一部分代码，因此符合单元的概念。至于单元该如何进行定义则取决于你：它可以是一个类也可以是关系密切的一些类。单元的定义决定了哪些内容需要进行 mock 处理（用虚拟的对象来进行替换）。你会使用轻量级的备选方案来替换数据库吗？你会替换与外部服务的交互吗？对于与被测功能无关，但是与被测对象具有一定联系的对象，是不是要采用 mock 的方式？

我的建议是采用一种平衡的方式。保持你的测试整洁和快捷，其他的问题都会随之得到解决。

我很少完全使用 mock 数据层，更倾向于使用嵌入式数据库来进行测试。在测试的时候，它们提供了一种便利的方式来加载数据。

作为一项准则，我通常会基于如下这两个原因才会采用 mock 方案模拟与外部服务的协作：

- 测试的速度，另外，我们能够不连接网络就能运行测试；
- 当与这些服务交互的时候，能够测试出错的场景。

另外，mock 与 stub 之间有一些细微的差异。我们将会尝试使用这两种方式，从而了解它们之间的关联关系。

7.4.1　完后工作的趁手工具

对于测试的初学者来说，第一道障碍就是缺乏对相关工具和库的了解，这些工具能够帮助我们编写明确和可维护的测试。

表 7-1 列出了几项工具，这并非详尽的列表，但包含了我们将会用到的工具，它们能够很容易地与 Spring 集成。

表 7-1

JUnit	最为广泛采用 Java 测试运行器，在所有的构建工具中都会自动启用
AssertJ	一个非常流畅的断言库，使用起来要比 Hamcrest 更容易
Mockito	很易于使用的 mock 框架
DbUnit	通过 XML 数据集来 mock 和断言数据库内容
Spock	一个优雅的 Groovy DSL，可以使用行为驱动开发（Behaviour Driven Development ，BDD）的风格（Given/When/Then）编写测试

Groovy 在我的测试工具集中占有一席之地。即便你不准备将 Groovy 代码放到生产环境中，也可以在测试中使用这门非常简便的语言。通过使用 Gradle，要实现这一点非常容易，不过我们会在几分钟之后再看这一话题。

7.5 验收测试

在 Web 应用中，“验收测试”通常是指浏览器中的端到端测试。在 Java 领域中，Selenium 是最为可靠和成熟的库之一。

而在 JavaScript 领域，我们可以找到其他的替代方案，如 PhantomJS 或 Protractor。PhantomJS 非常适合我们的场景，因为借助它能够在一个没有界面的浏览器中，通过 Web Driver 来运行 Selenium 测试，这样就提升了启动速度并且无需模拟 X 服务器或启动单独的 Selenium 服务器。详细的对比分析，如表 7-2 所示。

表 7-2

Selenium 2	通过 WebDriver 探测浏览器，从而进行自动化测试
PhantomJS	无界面的浏览器（没有 GUI），可能是最快的浏览器了
FluentLenium	用于执行 Selenium 测试的一个 API 很流畅的库
Geb	用于执行 Selenium 测试的一个 Groovy 库

7.6 第一个单元测试

现在，应该编写第一个单元测试了。

我们主要关注控制器层的测试，因为在我们的应用中，业务代码或服务实在太少了。为 Spring MVC 编写测试的关键是在类路径下添加 org.springframework.boot:spring-boot-starter-test 依赖。这将会添加多个非常有用的库，如下所示。

- `hamcrest`：JUnit 的断言库。
- `mockito`：一个 Mock 库。
- `spring-test`：Spring 的测试库。

我们将要测试当用户没有创建基本信息的时候，系统将会重定向到基本信息页面。

现在，已经有了一个自动生成的测试，名为 MasterSpringMvc4ApplicationTests。这应

该是 Spring 测试框架所能编写出的最基本的测试了：它什么事情都没有做，只是在上下文无法加载的时候，宣告失败：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvc4Application.
class)
@WebAppConfiguration
public class MasterSpringMvc4ApplicationTests {

    @Test
    public void contextLoads() {
    }
}
```

我们可以删除这个测试并创建一个新的测试，确保没有基本信息的用户默认将会重定向到基本信息页面。它实际上会测试 HomeController 的代码，所以将其命名为 `HomeControllerTest` 类，并将其放到 src/test/java 中，位于 HomeController 相同名称的包下。所有的 IDE 都有创建 JUnit 测试用例的快捷方式，现在，可以查找一下自己所使用的 IDE 如何创建 JUnit 测试用例。

如下就是测试的代码：

```
package masterSpringMvc.controller;

import masterSpringMvc.MasterSpringMvcApplication;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.
```

```
MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
public class HomeControllerTest {
    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).
build();
    }
    @Test
    public void should_redirect_to_profile() throws Exception {
        this.mockMvc.perform(get("/"))
                .andDo(print())
                .andExpect(status().isFound())
                .andExpect(redirectedUrl("/profile"));
    }
}
```

我们使用 MockMvc 模拟与 Spring 控制器的交互，而不必使用实际的 Servlet 容器。

我们还使用 Spring 所提供的一些匹配器来断言结果，它们实际上是实现了 Hamcrest 的匹配器。

语句“`.andDo(print())`”将会产生一个整洁的调试输出，能够反映该测试的请求和响应情况。如果你觉得它太冗长的话，可以为其添加注释。

这就是我们所需的所有工作！开始的时候，可能会觉得语法有点怪异，但是具有代码完成功能的 IDE 会对你有一定的帮助。

现在，我们想要测试如果用户填充了其口味信息，系统是否能够重定向到正确的搜索界面。为了实现这一点，我们需要使用 `MockHttpSession` 类来 stub 会话：

```
import org.springframework.mock.web.MockHttpSession;
import masterSpringMvc.profile.UserProfileSession;
```

```
// put this test below the other one
@Test
public void should_redirect_to_tastes() throws Exception {
    MockHttpSession session = new MockHttpSession();
    UserProfileSession sessionBean = new UserProfileSession();
    sessionBean.setTastes(Arrays.asList("spring", "groovy"));
    session.setAttribute("scopedTarget.userProfileSession",
sessionBean);

    this.mockMvc.perform(get("/").session(session))
        .andExpect(status().isFound())
        .andExpect(redirectedUrl("/search/mixed;keywords=spring,groo
vy"));
}
```

为了让这个测试运行起来，需要为 `UserProfileSession bean` 添加 `setTastes()` 的 setter 方法。

在 `org.springframework.mock.web` 包中有很多针对 Servlet 环境的 Mock 工具类。

需要注意的是，在会话中代表 bean 的属性会添加 scopedTarget 前缀。这是因为会话 bean 是由 Spring 进行代理的。因此，在 Spring 上下文中实际会有两个对象，分别是我们定义的实际 bean 以及会话中它的代理。

Mock 会话是一个很整洁的类，不过我们可以对测试进行重构，使用构造器来隐藏实现细节并且便于在后续进行重用：

```
@Test
public void should_redirect_to_tastes() throws Exception {

    MockHttpSession session = new SessionBuilder().
userTastes("spring", "groovy").build();
    this.mockMvc.perform(get("/")
        .session(session))
        .andExpect(status().isFound())
        .andExpect(redirectedUrl("/search/mixed;keywords=spring,groo
vy"));
}
```

构造器的代码如下所示：

```
public class SessionBuilder {
    private final MockHttpSession session;
```

```
    UserProfileSession sessionBean;

    public SessionBuilder() {
        session = new MockHttpSession();
        sessionBean = new UserProfileSession();
        session.setAttribute("scopedTarget.userProfileSession",
sessionBean);
    }

    public SessionBuilder userTastes(String... tastes) {
        sessionBean.setTastes(Arrays.asList(tastes));
        return this;
    }

    public MockHttpSession build() {
        return session;
    }
}
```

当然，在重构之后，我们测试应该依然能够通过。

7.7 Mock 与 Stub

如果我们希望测试 SearchController 类处理请求所得到的搜索结果，那么肯定要 mock Search Service。

有两种方法来实现该功能：使用 mock 或 stub。

7.7.1 使用 Mockito 进行 mock

首先，我们通过使用 Mockito 创建一个 mock 对象：

```
package masterSpringMvc.search;

import masterSpringMvc.MasterSpringMvcApplication;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
```

```
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;

import java.util.Arrays;

import static org.hamcrest.Matchers.*;
import static org.mockito.Matchers.*;
import static org.mockito.Mockito.*;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
public class SearchControllerMockTest {
    @Mock
    private SearchService searchService;

    @InjectMocks
    private SearchController searchController;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
        this.mockMvc = MockMvcBuilders
                .standaloneSetup(searchController)
                .setRemoveSemicolonContent(false)
                .build();
    }

    @Test
    public void should_search() throws Exception {

        when(searchService.search(anyString(), anyListOf(String.
```

```
class)))
                .thenReturn(Arrays.asList(
                        new LightTweet("tweetText")
                ));

        this.mockMvc.perform(get("/search/mixed;keywords=spring"))
                .andExpect(status().isOk())
                .andExpect(view().name("resultPage"))
                .andExpect(model().attribute("tweets", everyItem(
                        hasProperty("text", is("tweetText"))
                )));

        verify(searchService, times(1)).search(anyString(),
anyListOf(String.class));
    }
}
```

可以看到，我们并没有使用 Web 应用上下文来搭建 `MockMvc`，而是创建了一个独立的上下文。这个上下文只会包含控制器，这意味着我们能够完全掌控这些控制器及其依赖的实例化和初始化。这样的话，就可以很容易地为控制器注入 mock 实例。

这种方式的不足之处在于需要重新声明一些配置，例如我们通过配置声明不希望移除 URL 中分号后面的字符。

通过使用 Hamcrest 匹配器断言视图模型中最终的属性。

mock 方式有它的优点，例如能够在运行时检验与 mock 的交互和创建预期值。

同时，这种方式还能将测试与对象的实际实现进行解耦。例如，如果你修改了控制器中获取 Tweet 的方式，可能会破坏与该控制器相关的测试，因为它们依然试图 mock 我们不再依赖的服务。

7.7.2 在测试时 Stub bean

另外一种方式就是在测试中替换 `SearchService` 类的实现。

我们之前有点偷懒，没有为 `SearchService` 定义接口。要始终针对接口编程，而不是针对实现编程。在这句众所周知的谚语背后隐藏着“四人组（Gang of Four）”最重要的经验。

使用控制反转（Inversion of Control）的好处之一就是允许我们很容易地在测试或真实系统中替换实现类。为了实现这一点，我们需要将所有使用 `SearchService` 的地方换成

新的接口。好的 IDE 都会有一个名为“抽取接口（extract interface）”的重构功能，这个功能就是做这件事情的。这样的话，会创建一个接口，该接口包含了 SearchService 类的公开方法 search()：

```
public interface TwitterSearch {
    List<LightTweet> search(String searchType, List<String> keywords);
}
```

当然，我们的两个控制器 SearchController 和 SearchApiController 现在必须要使用接口，而不是使用实现类。

这样，我们就能够创建一个测试类，让它来针对测试场景替换 TwitterSearch 类。为了实现该功能，我们需要声明一个新的 Spring 配置类，名为 StubTwitterSearchConfig，这个配置类包含了 TwitterSearch 的另外一个实现。我将其放到 search 包中，与 SearchControllerMockTest 放到一起：

```
package masterSpringMvc.search;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

import java.util.Arrays;

@Configuration
public class StubTwitterSearchConfig {
    @Primary @Bean
    public TwitterSearch twitterSearch() {
        return (searchType, keywords) -> Arrays.asList(
                new LightTweet("tweetText"),
                new LightTweet("secondTweet")
        );
    }
}
```

在这个配置中，我们通过@Primary 注解重新声明了 TwitterSearch bean，这会告诉 Spring，如果在类路径下还有其他实现类的话，将会优先使用该实现。

因为 TwitterSearch 接口只包含一个方法，所以我们可以通过 lambda 表达式来实现。

如下就是完整的测试代码，它通过 SpringApplicationConfiguration 注解同时

使用了 StubConfiguration 类和我们的主配置：

```
package masterSpringMvc.search;

import masterSpringMvc.MasterSpringMvcApplication;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = {
        MasterSpringMvcApplication.class,
        StubTwitterSearchConfig.class
})
@WebAppConfiguration
public class SearchControllerTest {
    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).
build();
    }

    @Test
    public void should_search() throws Exception {
```

```
        this.mockMvc.perform(get("/search/mixed;keywords=spring"))
                .andExpect(status().isOk())
                .andExpect(view().name("resultPage"))
                .andExpect(model().attribute("tweets", hasSize(2)))
                .andExpect(model().attribute("tweets",
                                hasItems(
                                        hasProperty("text",
is("tweetText")),
                                        hasProperty("text",
is("secondTweet"))
                                ))
                );
    }
}
```

7.7.3　该使用 Mock 还是 Stub

这两种方式各有其优点，关于更为详细的阐述，参考 Martin Fowler 这篇很棒的文章：http://martinfowler.com/articles/mocksArentStubs.html。

在我的测试中会更多地使用 Stub，因为我喜欢的理念是测试对象的输出，而不是其内部的工作机制。但是这完全取决于你，Spring 的核心是一个依赖注入的框架，这意味着你可以很容易地选择自己所喜欢的方式。

7.8　对 REST 控制器进行单元测试

我们已经测试了一个传统的控制器，校验它能够重定向到视图上。REST 控制器的测试在原则上与其非常类似，但是也有点细微的差别。

因为我们测试的是控制器的 JSON 输出，因此需要一个 JSON 断言库。在 build.gradle 文件中添加如下的依赖：

```
testCompile 'com.jayway.jsonpath:json-path'
```

我们为 SearchApiController 编写一个测试，这个控制器允许对 Tweet 进行搜索并返回 JSON 或 XML 格式的结果：

```
package masterSpringMvc.search.api;

import masterSpringMvc.MasterSpringMvcApplication;
```

```
import masterSpringMvc.search.StubTwitterSearchConfig;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = {
        MasterSpringMvcApplication.class,
        StubTwitterSearchConfig.class
})
@WebAppConfiguration
public class SearchApiControllerTest {
    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).
build();
    }

    @Test
    public void should_search() throws Exception {

        this.mockMvc.perform(
```

```
                get("/api/search/mixed;keywords=spring")
                        .accept(MediaType.APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaTy
pe.APPLICATION_JSON))
                .andExpect(jsonPath("$", hasSize(2)))
                .andExpect(jsonPath("$[0].text", is("tweetText")))
                .andExpect(jsonPath("$[1].text", is("secondTweet")));
    }
}
```

请留意一下针对 JSON 输出的简洁和优雅断言。测试我们的用户控制器需要一点额外的工作。

首先，我们需要将 assertj 添加到类路径中，它能够帮助我们编写更整洁的测试：

```
testCompile 'org.assertj:assertj-core:3.0.0'
```

然后，为了简化测试，为 UserRepository 类添加 reset()方法，它会为我们的这个测试提供帮助：

```
void reset(User... users) {
        userMap.clear();
        for (User user : users) {
                save(user);
        }
}
```

在现实中，我们应该抽取一个接口并为这个测试创建 Stub，我将其作为一个练习留给读者。

如下就是获取用户列表的第一个测试：

```
package masterSpringMvc.user.api;

import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.hamcrest.Matchers.*;
   import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
public class UserApiControllerTest {

    @Autowired
    private WebApplicationContext wac;

    @Autowired
    private UserRepository userRepository;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).
build();
        userRepository.reset(new User("bob@spring.io"));
    }

    @Test
    public void should_list_users() throws Exception {
        this.mockMvc.perform(
                get("/api/users")
                        .accept(MediaType.APPLICATION_JSON)
        )
                .andExpect(status().isOk())
                .andExpect(content().contentTypeCompatibleWith(MediaTy
```

```
pe.APPLICATION_JSON))
                .andExpect(jsonPath("$", hasSize(1)))
                .andExpect(jsonPath("$[0].email", is("bob@spring.
io")));
    }
}
```

为了让它运行起来，我们需要为 User 类添加一个构造器，并接受 E-mail 属性作为参数。注意：为了让 Jackson 正常运行，我们还需要一个默认的构造器。

这个测试与之前的测试非常类似，只是多了 `UserRepository` 的搭建过程。

接下来，我们测试创建用户的 POST 方法：

```
import static org.assertj.core.api.Assertions.assertThat;

// Insert this test below the previous one
@Test
public void should_create_new_user() throws Exception {
        User user = new User("john@spring.io");
        this.mockMvc.perform(
                post("/api/users")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(JsonUtil.toJson(user))
        )
                .andExpect(status().isCreated());

        assertThat(userRepository.findAll())
                .extracting(User::getEmail)
                .containsOnly("bob@spring.io", "john@spring.io");
}
```

这里有两点需要注意一下。第一，在测试之后，我们使用 AssertJ 来断言 repository 中内容，因此需要添加如下的静态导入：

```
import static org.assertj.core.api.Assertions.assertThat;
```

第二就是我们使用了一个工具方法，在将数据发送到控制器之前，将其从对象转换成了 JSON。为了实现该功能，我在 utils 包下，创建了一个简单的工具类，如下所示：

```
package masterSpringMvc.utils;

import com.fasterxml.jackson.annotation.JsonInclude;
```

```
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;

public class JsonUtil {
    public static byte[] toJson(Object object) throws IOException {
        ObjectMapper mapper = new ObjectMapper();
        mapper.setSerializationInclusion(JsonInclude.Include.NON_
NULL);
        return mapper.writeValueAsBytes(object);
    }
}
```

针对 DELETE 方法的测试如下所示：

```
@Test
public void should_delete_user() throws Exception {
        this.mockMvc.perform(
                delete("/api/user/bob@spring.io")
                        .accept(MediaType.APPLICATION_JSON)
        )
                .andExpect(status().isOk());
        assertThat(userRepository.findAll()).hasSize(0);
}

@Test
public void should_return_not_found_when_deleting_unknown_user()
throws Exception {
        this.mockMvc.perform(
                delete("/api/user/non-existing@mail.com")
                        .accept(MediaType.APPLICATION_JSON)
        )
                .andExpect(status().isNotFound());
}
```

最后是针对 PUT 方法的测试，该方法会更新一个用户：

```
@Test
public void put_should_update_existing_user() throws Exception {
        User user = new User("ignored@spring.io");
        this.mockMvc.perform(
                put("/api/user/bob@spring.io")
                        .content(JsonUtil.toJson(user))
```

```
                        .contentType(MediaType.APPLICATION_JSON)
        )
                .andExpect(status().isOk());

        assertThat(userRepository.findAll())
                .extracting(User::getEmail)
                .containsOnly("bob@spring.io");
}
```

一旦最后一个测试没有通过，我们只要检查一下 `UserApiController` 的实现，就能很容易地知道这是为什么：

```
   @RequestMapping(value = "/user/{email}", method = RequestMethod.
PUT)
   public ResponseEntity<User> updateUser(@PathVariable String email,
@RequestBody User user) throws EntityNotFoundException {
        User saved = userRepository.update(email, user);
        return new ResponseEntity<>(saved, HttpStatus.CREATED);
   }
```

我们在控制器中返回了错误的状态码，将其修改为 `HttpStatus.OK` 之后，测试应该就能变成绿色了！

通过使用 Spring，我们可以借助与应用本身相同的配置，很容易地编写测试，不过我们也可以在测试的搭建过程中非常高效地重写或变更某些配置元素。

另外一个有意思的地方在于，运行所有的测试期间，应用上下文只会加载一次，这意味着它的开销是很小的。

我们的应用本身很小，因此并没有将配置拆分为可重用的代码块。有一项很好的实践就是不要在每个测试中都加载完整的应用上下文。我们实际上可以通过`@ComponentScan` 注解，将组件扫描功能划分到不同的单元中。

这个注解有多个属性，可以通过 `includeFilter` 和 `excludeFilter` 来声明过滤器（例如，只加载控制器），并且还可以使用 `basePackageClasses` 和 `basePackages` 来指定要扫描的具体的包。

我们还可以将配置划分到多个带有`@Configuration` 注解的类中。例如，可以将该应用中用户相关的代码和 Tweet 相关的代码划分到两个不同的组成部分中。

稍后看一下验收测试，这是一种完全不同的测试形式。

7.9 测试认证

如果你想要在 MockMvc 测试中搭建 Spring Security 的话，那么可以紧挨着前面的测试代码编写如下的测试功能：

```
package masterSpringMvc.user.api;

import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.user.User;
import masterSpringMvc.user.UserRepository;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.http.MediaType;
import org.springframework.security.web.FilterChainProxy;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import java.util.Base64;

import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.status;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MasterSpringMvcApplication.
class)
@WebAppConfiguration
public class UserApiControllerAuthTest {

    @Autowired
    private FilterChainProxy springSecurityFilter;
    @Autowired
    private WebApplicationContext wac;
```

```
    @Autowired
    private UserRepository userRepository;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).
addFilter(springSecurityFilter).build();
        userRepository.reset(new User("bob@spring.io"));
    }

    @Test
    public void unauthenticated_cannot_list_users() throws Exception {
        this.mockMvc.perform(
                get("/api/users")
                        .accept(MediaType.APPLICATION_JSON)
        )
                .andExpect(status().isUnauthorized());
    }

    @Test
    public void admin_can_list_users() throws Exception {
        this.mockMvc.perform(
                get("/api/users")
                        .accept(MediaType.APPLICATION_JSON)
                        .header("Authorization", basicAuth("admin",
"admin"))
        )
                .andExpect(status().isOk());
    }

    private String basicAuth(String login, String password) {
        byte[] auth = (login + ":" + password).getBytes();
        return "Basic " + Base64.getEncoder().encodeToString(auth);
    }
}
```

在上面的样例中，我们在配置中添加了 `SpringSecurityFilter`，这将会激活 Spring Security 检查。为了测试认证是否起作用，我们只需在想要执行的请求中带上正确的头信息。

基本认证的优势在于它非常易于模拟。如果是更为复杂的配置，我们就需要针对认证端点执行一个 mock 请求了。

在编写本书的时候，Spring Boot 的版本号是 1.2.3，它还依赖于 Spring Security 3。

不久，Spring Boot 1.3.0 将会发布，它会升级 Spring Security，使用 Spring Security 4 版本。

这是一个好消息，因为 Spring Security 4 能够通过很简单的注解，非常容易地搭建用户认证功能，参见 http://docs.spring.io/spring-security/site/docs/4.0.x/reference/htmlsingle/#test 来了解更多细节。

7.10 编写验收测试

单元测试只能测试应用程序中各组件之间交互的一个子集。为了更进一步，我们需要进行验收测试，这种测试会启动完整的应用并且允许与界面进行交互。

7.10.1 Gradle 配置

在为项目添加集成测试的时候，我们想要做的第一件事情就是将其放到与单元测试不同的位置中。

实际上，这样做的原因在于验收测试要比单元测试慢得多。它们可以作为不同的集成任务的一部分，例如夜间构建，我们希望开发人员能够从 IDE 中非常容易地运行不同类型的测试。为了在 Gradle 中实现该功能，我们需要添加名为 `integrationTest` 的新配置。对于 Gradle 来说，配置就是一组制件（artifact）及其依赖。在我们的项目中，已经有了多项配置：`compile`、`testCompile` 等。

你可以查看一下项目中的配置，或者在项目的根目录下输入“./gradlew properties”。

在 build.gradle 文件的结尾处添加如下的新配置：

```
configurations {
    integrationTestCompile.extendsFrom testCompile
    integrationTestRuntime.extendsFrom testRuntime
}
```

这样我们就可以为 `integrationTestCompile` 和 `integrationTestRuntime` 声明依赖，更为重要的是通过继承 test 配置，我们也能访问它们的依赖。

我不建议声明像integrationTestCompile这样的集成测试依赖。只要有 Gradle 它就可以运行，但是并不支持 IDE 中的使用。我通常的做法是将集成测试的依赖声明为 testCompile 依赖。这只是一个很小的不便之处。

我们现在已经有了新的配置，接下来必须要创建与之关联的 sourceSet 类。sourceSet 类代表的是 Java 源码和资源的逻辑分组。它们也需要继承自测试和主类，参见如下的代码：

```
sourceSets {
    integrationTest {
        compileClasspath += main.output + test.output
        runtimeClasspath += main.output + test.output
    }
}
```

最后，我们需要添加一项任务，在构建的时候来运行，如下所示：

```
task integrationTest(type: Test) {
    testClassesDir = sourceSets.integrationTest.output.classesDir
    classpath = sourceSets.integrationTest.runtimeClasspath
    reports.html.destination = file("${reporting.baseDir}/
integrationTests")
}
```

要运行测试的话，我们可以输入“./gradlewintegrationTest”。除了配置类路径以及去哪里寻找测试类，我们还定义了存放测试报告的目录。

这个配置允许我们在 src/integrationTest/java 或 src/integrationTest/groovy 目录下编写测试，这样的话就能够很容易地识别它们，并且可以独立于单元测试来运行。

默认情况下，报告会生成在 build/reports/tests 目录下。如果我们不进行重写并且通过 gradle clean test integrationTest 命令同时运行单元测试和集成测试的话，它们会互相覆盖。

另外值得一提的是，Gradle 生态系统中有一个很新的插件，它致力于简化测试配置的声明，参见 https://plugins.gradle.org/plugin/org.unbroken-dome.test-sets 以了解更多信息。

7.10.2 第一个 FluentLenium 测试

FluentLenium 是一个非常棒的库，它致力于简化 Selenium 方面的测试。在我们的构建脚本中，添加如下的几项依赖：

```
testCompile 'org.fluentlenium:fluentlenium-assertj:0.10.3'
testCompile 'com.codeborne:phantomjsdriver:1.2.1'
testCompile 'org.seleniumhq.selenium:selenium-java:2.45.0'
```

默认情况下，`fluentlenium` 会级联依赖 `selenium-java`。我们进行重新声明只是明确使用最新的版本。我们还添加了对 PhantomJS driver 的依赖，Selenium 官方并没有提供对它的支持。`selenium-java` 库的问题在于它会将所有支持的 Web Driver 都关联进来。

通过输入 gradle dependencies 命令，我们可以看到项目的依赖树。在底部，会看到类似如下的内容：

```
+--- org.fluentlenium:fluentlenium-assertj:0.10.3
|    +--- org.fluentlenium:fluentlenium-core:0.10.3
|    |    \--- org.seleniumhq.selenium:selenium-java:2.44.0 -> 2.45.0
|    |         +--- org.seleniumhq.selenium:selenium-chromedriver:
2.45.0

|    |         +--- org.seleniumhq.selenium:selenium-htmlunitdriver:
2.45.0

|    |         +--- org.seleniumhq.selenium:selenium-firefoxdriver:
2.45.0

|    |         +--- org.seleniumhq.selenium:selenium-ie-driver:2.45.0

|    |         +--- org.seleniumhq.selenium:selenium-safaridriver:
2.45.0

|    |         +--- org.webbitserver:webbit:0.4.14 (*)
|    |         \--- org.seleniumhq.selenium:selenium-leg-rc:2.45.0
|    |              \--- org.seleniumhq.selenium:selenium-remotedriver:
2.45.0 (*)
|    \--- org.assertj:assertj-core:1.6.1 -> 3.0.0
```

将这么多的依赖都放到类路径下是完全没有必要的，因为我们只会用到 PhantomJS driver。为了排除掉不需要的依赖，我们可以在构建脚本中添加如下的代码，就将其放在依

赖声明的前面：

```
configurations {
    testCompile {
        exclude module: 'selenium-safari-driver'
        exclude module: 'selenium-ie-driver'
        //exclude module: 'selenium-firefox-driver'
        exclude module: 'selenium-htmlunit-driver'
        exclude module: 'selenium-chrome-driver'
    }
}
```

我们只临时保留了 firefox driver，PhantomJS driver 是一个没有界面的浏览器，如果没有 GUI 的话，试图理解发生了什么状况是很困难的，所以，我们可以切换到 Firefox 上来调试复杂的测试。

类路径已经配置好了，我们现在就可以编写第一个集成测试。Spring Boot 提供了很便利的注解来支持这种测试：

```
import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.search.StubTwitterSearchConfig;
import org.fluentlenium.adapter.FluentTest;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriver;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.boot.test.WebIntegrationTest;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = {
        MasterSpringMvcApplication.class,
        StubTwitterSearchConfig.class
})
@WebIntegrationTest(randomPort = true)
public class FluentIntegrationTest extends FluentTest {
    @Value("${local.server.port}")
    private int serverPort;
```

```
    @Override
    public WebDriver getDefaultDriver() {
        return new PhantomJSDriver();
    }

    public String getDefaultBaseUrl() {
        return "http://localhost:" + serverPort;
    }

    @Test
    public void hasPageTitle() {
        goTo("/");
        assertThat(findFirst("h2").getText()).isEqualTo("Login");
    }
}
```

FluentLenium 提供了非常整洁的 API 来请求 DOM 元素。借助 AssertJ，我们可以针对页面内容编写非常易读的断言。

访问 https://github.com/FluentLenium/FluentLenium 了解更多信息。

我们这里使用了`@WebIntegrationTest` 注解，所以 Spring 实际上会创建嵌入式的 Servlet 容器（Tomcat），并且会在一个随机的端口上启动我们的 Web 应用！我们需要在运行时得到这个端口数，这样就能够为测试提供一个基础的 URL，这个基础的 URL 会作为前缀用在我们测试的所有 URL 导航上。

如果你现在运行测试的话，将会看到如下的错误信息：

```
java.lang.IllegalStateException: The path to the driver executable must
be set by the phantomjs.binary.path capability/system property/PATH
variable; for more information, see https://github.com/ariya/phantomjs/
wiki. The latest version can be downloaded from http://phantomjs.org/
download.html
```

要让测试正常运行起来，我们需要在机器上安装 PhantomJS。在 Mac 上，只需使用 `brew install phantomjs` 命令即可。对于其他的平台，请参考 http://phantomjs.org/download.html 页面上的文档。

如果你不想在机器上安装新的二进制文件的话，那么可以将 `new PhantomJSDriver()`

替换为 `new FirefoxDriver()`。这样的话，测试会慢一些，但是会带有 GUI 界面。

我们的第一个测试加载了基本信息页面，对吧？现在我们需要想办法登录进去。

采用 Stub 来模拟登录怎么样？

将如下的类放到测试的源码中（`src/test/java`）：

```
package masterSpringMvc.auth;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.security.authentication.
UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.
SecurityContextHolder;
import org.springframework.social.connect.ConnectionFactoryLocator;
import org.springframework.social.connect.UsersConnectionRepository;
import org.springframework.social.connect.web.
ProviderSignInController;
import org.springframework.social.connect.web.SignInAdapter;
import org.springframework.web.context.request.NativeWebRequest;
import org.springframework.web.servlet.view.RedirectView;

@Configuration
public class StubSocialSigninConfig {

    @Bean
    @Primary
    @Autowired
    public ProviderSignInController signInController(ConnectionFactory
Locator factoryLocator,

UsersConnectionRepository usersRepository,
                                                    SignInAdapter
signInAdapter) {
        return new FakeSigninController(factoryLocator,
usersRepository, signInAdapter);
    }
    public class FakeSigninController extends ProviderSignInController
{
        public FakeSigninController(ConnectionFactoryLocator
```

```
connectionFactoryLocator,
                                    UsersConnectionRepository
usersConnectionRepository,
                                    SignInAdapter signInAdapter) {
            super(connectionFactoryLocator, usersConnectionRepository,
signInAdapter);
        }

        @Override
        public RedirectView signIn(String providerId, NativeWebRequest
request) {
            UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken("geowar
in", null, null);
            SecurityContextHolder.getContext().setAuthentication(auth
entication);
            return new RedirectView("/");
        }
    }
}
```

这样会将所有点击 Twitter 登录按钮的用户均认证为 geowarin。

我们接下来编写第二个测试，这个测试会填充基本信息表单并断言展现了搜索结果：

```
import masterSpringMvc.MasterSpringMvcApplication;
import masterSpringMvc.auth.StubSocialSigninConfig;
import masterSpringMvc.search.StubTwitterSearchConfig;
import org.fluentlenium.adapter.FluentTest;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.phantomjs.PhantomJSDriver;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.boot.test.WebIntegrationTest;
import org.springframework.test.context.junit4.
SpringJUnit4ClassRunner;

import static org.assertj.core.api.Assertions.assertThat;
import static org.fluentlenium.core.filter.FilterConstructor.withName;
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = {
        MasterSpringMvcApplication.class,
```

```
        StubTwitterSearchConfig.class,
        StubSocialSigninConfig.class
})
@WebIntegrationTest(randomPort = true)
public class FluentIntegrationTest extends FluentTest {

    @Value("${local.server.port}")
    private int serverPort;

    @Override
    public WebDriver getDefaultDriver() {
        return new PhantomJSDriver();
    }

    public String getDefaultBaseUrl() {
        return "http://localhost:" + serverPort;
    }

    @Test
    public void hasPageTitle() {
        goTo("/");
        assertThat(findFirst("h2").getText()).isEqualTo("Login");
    }

    @Test
    public void should_be_redirected_after_filling_form() {
        goTo("/");
        assertThat(findFirst("h2").getText()).isEqualTo("Login");

        find("button", withName("twitterSignin")).click();
        assertThat(findFirst("h2").getText()).isEqualTo("Your
profile");

        fill("#twitterHandle").with("geowarin");
        fill("#email").with("geowarin@mymail.com");
        fill("#birthDate").with("03/19/1987");

        find("button", withName("addTaste")).click();
        fill("#tastes0").with("spring");
        find("button", withName("save")).click();

        takeScreenShot();
        assertThat(findFirst("h2").getText()).isEqualTo("Tweet results
```

```
for spring");
        assertThat(findFirst("ul.collection").find("li")).hasSize(2);
    }
}
```

我们可以非常容易地使用 Web Driver 获取当前浏览器的截屏。这将会产生如图 7-2 所示的输出。

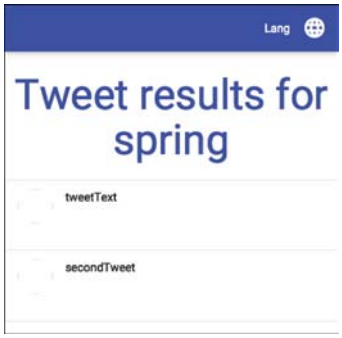

图 7-2

7.10.3 使用 FluentLenium 创建页面对象

前面的测试看上去有些凌乱，我们在测试中硬编码了所有的选择器。如果使用同一个元素编写大量的测试的话，会有很大的风险，因为当页面布局发生变化的时候，所有的测试都会损坏无法使用了。除此之外，这个测试读起来也很困难。

为了解决这个问题，通用的实践是使用使用页面对象（page object），它代表了应用程序中某个特定的页面。在使用 FluentLenium 的时候，页面对象必须继承 FluentPage 类。

我们将会创建 3 个页面，每个对应 GUI 中的一个元素。第一个将会是登录页面，它能够点击 `twitterSignin` 按钮；第二个是基本信息页面，包含了填写基本信息的便利方法；最后一个是结果页面，在这个页面上我们可以断言结果能够正常展现。

首先，我们创建登录页。我将这 3 个页面都放在了 pages 包中：

```
package pages;

import org.fluentlenium.core.FluentPage;
import org.fluentlenium.core.domain.FluentWebElement;
import org.openqa.selenium.support.FindBy;
```

```
import static org.assertj.core.api.Assertions.assertThat;

public class LoginPage extends FluentPage {
    @FindBy(name = "twitterSignin")
    FluentWebElement signinButton;

    public String getUrl() {
        return "/login";
    }

    public void isAt() {
        assertThat(findFirst("h2").getText()).isEqualTo("Login");
    }

    public void login() {
        signinButton.click();
    }
}
```

接下来，创建基本信息页面：

```
package pages;

import org.fluentlenium.core.FluentPage;
import org.fluentlenium.core.domain.FluentWebElement;
import org.openqa.selenium.support.FindBy;

import static org.assertj.core.api.Assertions.assertThat;

public class ProfilePage extends FluentPage {
    @FindBy(name = "addTaste")
    FluentWebElement addTasteButton;
    @FindBy(name = "save")
    FluentWebElement saveButton;

    public String getUrl() {
        return "/profile";
    }

    public void isAt() {
        assertThat(findFirst("h2").getText()).isEqualTo("Your
profile");
    }
```

```
    public void fillInfos(String twitterHandle, String email, String
birthDate) {
        fill("#twitterHandle").with(twitterHandle);
        fill("#email").with(email);
        fill("#birthDate").with(birthDate);
    }

    public void addTaste(String taste) {
        addTasteButton.click();
        fill("#tastes0").with(taste);
    }

    public void saveProfile() {
        saveButton.click();
    }
}
```

最后，创建搜索结果的页面：

```
package pages;

import com.google.common.base.Joiner;
import org.fluentlenium.core.FluentPage;
import org.fluentlenium.core.domain.FluentWebElement;
import org.openqa.selenium.support.FindBy;

import static org.assertj.core.api.Assertions.assertThat;

public class SearchResultPage extends FluentPage {
    @FindBy(css = "ul.collection")
    FluentWebElement resultList;
    public void isAt(String... keywords) {
        assertThat(findFirst("h2").getText())
                .isEqualTo("Tweet results for " + Joiner.on(",").
join(keywords));
    }

    public int getNumberOfResults() {
        return resultList.find("li").size();
    }
}
```

现在，我们就可以使用页面对象来重构之前的测试：

```
@Page
```

```
    private LoginPage loginPage;
    @Page
    private ProfilePage profilePage;
    @Page
    private SearchResultPage searchResultPage;

    @Test
    public void should_be_redirected_after_filling_form() {
        goTo("/");
        loginPage.isAt();

        loginPage.login();
        profilePage.isAt();

        profilePage.fillInfos("geowarin", "geowarin@mymail.com",
    "03/19/1987");
        profilePage.addTaste("spring");

        profilePage.saveProfile();

        takeScreenShot();
        searchResultPage.isAt();
        assertThat(searchResultPage.getNumberOfResults()).isEqualTo(2);
    }
```

这样看起来，是不是可读性要高得多了呢？

7.10.4　用 Groovy 实现测试

如果你不了解 Groovy 的话，可以将其视为 Java 的近亲，只不过它消除了不必要的繁琐代码。Groovy 是一个动态语言，具有可选类型（optional typing）功能。这意味着，如果你在意类型系统的话，在这方面会有所保证，如果你确切地了解自己正在做的事情，也可以体验到 duck 类型的灵活性。

通过使用该语言，在编写 POJO 的时候，我们不需要编写 getter、setter、equals 以及 hashcode 方法。这些事情都已经为你处理好了。

当我们使用“==”的时候，它实际上就会调用 equals 方法。操作符可以进行重载，这样的话，我们就可以使用箭头字符定义出整洁的语法，例如使用“<<”将文本写入到文件之中。这也意味着我们可以将一个整型加到 `BigInteger` 上，并得到正确的结果。

Groovy 开发工具集（Groovy Development Kit，GDK）为典型的 Java 对象添加了很多

有意思的方法。它还将正则表达式和闭包作为了一等公民。

如果你想详细了解 Groovy 的话，可以参考 Groovy 编码风格指南，地址是 http://www.groovy-lang.org/style-guide.html，还可以观看 Peter Ledbrook 的精彩演讲 http://www.infoq.com/presentations/groovy-for-java。

如我之前所述，我一直在应用的测试中使用 Groovy。它确实能够提升代码的可读性和开发人员的生产率。

7.10.5 使用 Spock 进行单元测试

为了在项目中使用 Groovy 来编写测试，我们需要使用 Groovy 插件来替代 Java 插件。

我们之前的构建脚本是这样的：

```
apply plugin: 'java'
```

将其修改为：

```
apply plugin: 'groovy'
```

这个修改是完全没有什么损害的，Groovy 插件扩展了 Java 插件，所以它带来的唯一差异就是我们能够在 `src/main/groovy`、`src/test/groovy` 和 `src/integrationTest/groovy` 中添加 Groovy 源码了。

显然，我们需要将 Groovy 添加到类路径中，同时，还要添加 Spock，这是最为流行的 Groovy 测试库。添加 `spock-spring` 即可，它提供了对 Spring 的兼容性：

```
testCompile 'org.codehaus.groovy:groovy-all:2.4.4:indy'
testCompile 'org.spockframework:spock-spring'
```

现在，我们可以使用不同的方式来重写 `HomeControllerTest`。我们 在 `src/test/groovy` 中创建一个 `HomeControllerSpec` 类，我将其放在了 `masterSpringMvc.controller` 包中，与第一个 `HomeControllerTest` 实例所在的包相同：

```
package masterSpringMvc.controller

import masterSpringMvc.MasterSpringMvcApplication
import masterSpringMvc.search.StubTwitterSearchConfig
```

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.SpringApplicationContextLoader
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.web.WebAppConfiguration
import org.springframework.test.web.servlet.MockMvc
import org.springframework.test.web.servlet.setup.MockMvcBuilders
import org.springframework.web.context.WebApplicationContext
import spock.lang.Specification

import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;

@ContextConfiguration(loader = SpringApplicationContextLoader,
        classes = [MasterSpringMvcApplication,
StubTwitterSearchConfig])
@WebAppConfiguration
class HomeControllerSpec extends Specification {
    @Autowired
    WebApplicationContext wac;

    MockMvc mockMvc;

    def setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).
build();
    }
    def "User is redirected to its profile on his first visit"() {
        when: "I navigate to the home page"
        def response = this.mockMvc.perform(get("/"))

        then: "I am redirected to the profile page"
        response
                .andExpect(status().isFound())
                .andExpect(redirectedUrl("/profile"))
    }
}
```

我们的测试瞬间变得更加易读，它能够使用字符串作为方法名并且用到了 Spock 所提供的一些 BDD 领域特定语言（Domain Specific Language，DSL）。尽管这里不太明显，但实际上 then 代码块中的每个语句都是一个隐含的断言。

在编写本书的时候，Spock 不能读取元数据注解，@SpringApplicationConfiguration 注解无法使用，所以我们将其替换为@ContextConfiguration(loader = SpringAp_plicationContextLoader)，它实际上做的事情是一样的。

我们现在有了同一个测试的两个版本，一个是用 Java 编写的，另外一个是使用 Groovy 编写的。你可以选择一个最适合你编码风格的进行保留，将另外一个移除掉。如果你选择使用 Groovy 的话，需要重写 Groovy 测试中的 should_redirect_to_tastes()，这是非常简单的。

Spock 对 mock 也提供了强大的支持。我们可以采用稍微不同的方式来重写 SearchControllerMockTest 类：

```
package masterSpringMvc.search

import masterSpringMvc.MasterSpringMvcApplication
import org.springframework.boot.test.SpringApplicationContextLoader
import org.springframework.test.context.ContextConfiguration
import org.springframework.test.context.web.WebAppConfiguration
import org.springframework.test.web.servlet.setup.MockMvcBuilders
import spock.lang.Specification

import static org.hamcrest.Matchers.*;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.
MockMvcResultMatchers.*;
@ContextConfiguration(loader = SpringApplicationContextLoader,
        classes = [MasterSpringMvcApplication])
@WebAppConfiguration
class SearchControllerMockSpec extends Specification {
    def twitterSearch = Mock(TwitterSearch)
    def searchController = new SearchController(twitterSearch)

    def mockMvc = MockMvcBuilders.standaloneSetup(searchController)
            .setRemoveSemicolonContent(false)
            .build()

    def "searching for the spring keyword should display the search
page"() {
        when: "I search for spring"
        def response = mockMvc.perform(get("/search/
mixed;keywords=spring"))
```

```
        then: "The search service is called once"
        1 * twitterSearch.search(_, _) >> [new
LightTweet('tweetText')]

        and: "The result page is shown"
        response
                .andExpect(status().isOk())
                .andExpect(view().name("resultPage"))

        and: "The model contains the result tweets"
        response
                .andExpect(model().attribute("tweets", everyItem(
                hasProperty("text", is("tweetText"))
        )))
    }
}
```

所有繁琐的 Mockito 均消失了。then 代码块实际上会断言 `twitterSearch` 方法被调用了一次（1 *）并且接受任意的参数（_,_）。与 Mockito 类似，我们也可以指定预期的参数。

双箭头“>>”语法用来从 mock 方法中返回一个对象。在我们的例子中，这是只包含一个元素的列表。

我们只是在类路径中多加了几个依赖，现在就能编写更加易读的测试了，不过这还没有完工。接下来，我们会使用 Geb 来重构验收测试，这是一个简化 Selenium 测试的 Groovy 库。

7.10.6　使用 Geb 进行集成测试

Geb 是在 Grails 框架中编写测试的事实标准。尽管它的版本是 0.12.0，但是它很稳定并且非常易于使用。

它提供了类似 jQuery 的选择器 API，这样即便是对于前端开发人员来说，测试也非常易于编写。Groovy 语言本身也受到了 JavaScript 的影响，这对他们来说也具有一定的吸引力。

现在，我们将 Geb 对 Spock 规范的支持添加到类路径中：

```
testCompile 'org.gebish:geb-spock:0.12.0'
```

Geb 可以通过一个 Groovy 脚本来进行配置，这个脚本要位于 `src/integrationTest/groovy` 根目录下，名称为 `GebConfig.groovy`：

```
import org.openqa.selenium.Dimension
import org.openqa.selenium.firefox.FirefoxDriver
import org.openqa.selenium.phantomjs.PhantomJSDriver

reportsDir = new File('./build/geb-reports')
driver = {
        def driver = new FirefoxDriver()
    // def driver = new PhantomJSDriver()
    driver.manage().window().setSize(new Dimension(1024, 768))
    return driver
}
```

在这个配置中，我们指定了要将 Geb 生成的报告放在什么地方以及要使用哪一个 driver。Geb 中的报告是截屏方案的一个增强版本，它还以 HTML 的形式包含了当前页面。通过在 Geb 测试中调用 report 函数，报告可以在任意的时间点生成。

我们接下来使用 Geb 重写第一个集成测试：

```
import geb.Configuration
import geb.spock.GebSpec
import masterSpringMvc.MasterSpringMvcApplication
import masterSpringMvc.search.StubTwitterSearchConfig
import org.springframework.beans.factory.annotation.Value
import org.springframework.boot.test.SpringApplicationContextLoader
import org.springframework.boot.test.WebIntegrationTest
import org.springframework.test.context.ContextConfiguration
@ContextConfiguration(loader = SpringApplicationContextLoader,
        classes = [MasterSpringMvcApplication,
StubTwitterSearchConfig])
@WebIntegrationTest(randomPort = true)
class IntegrationSpec extends GebSpec {

    @Value('${local.server.port}')
    int port

    Configuration createConf() {
        def configuration = super.createConf()
        configuration.baseUrl = "http://localhost:$port"
        configuration
    }

    def "User is redirected to the login page when not logged"() {
```

```
        when: "I navigate to the home page"
        go '/'
//        report 'navigation-redirection'
        then: "I am redirected to the profile page"
        $('h2', 0).text() == 'Login'
    }
}
```

现在，它看上去和 FluentLenium 非常类似，我们已经看到了“$”函数，它允许我们通过其选择器获取 DOM 元素。在这里，我们还使用 0 这个索引来表明想要寻找页面上的第一个 h2 元素。

7.10.7　在 Geb 中使用页面对象

通过 Geb 来使用页面对象是非常令人愉悦的。我们将会创建与前面一样的页面对象，这样的话，你就能够看出它们之间的差异。

在使用 Geb 的时候，页面对象必须要继承自 geb.Page 类。首先，我们创建 LoginPage。建议不要将其放到与之前的测试相同的包里面，因此我们创建了名为 geb.pages 的包：

```
package geb.pages

import geb.Page

class LoginPage extends Page {
    static url = '/login'
    static at = { $('h2', 0).text() == 'Login' }
    static content = {
        twitterSignin { $('button', name: 'twitterSignin') }
    }

    void loginWithTwitter() {
        twitterSignin.click()
    }
}
```

然后，我们创建 ProfilePage：

```
package geb.pages

import geb.Page
```

```
class ProfilePage extends Page {

    static url = '/profile'
    static at = { $('h2', 0).text() == 'Your profile' }
    static content = {
        addTasteButton { $('button', name: 'addTaste') }
        saveButton { $('button', name: 'save') }
    }

    void fillInfos(String twitterHandle, String email, String
birthDate) {
        $("#twitterHandle") << twitterHandle
        $("#email") << email
        $("#birthDate") << birthDate
    }

    void addTaste(String taste) {
        addTasteButton.click()
        $("#tastes0") << taste
    }

    void saveProfile() {
        saveButton.click();
    }
}
```

这基本上与前面的页面是相同的。注意，我们使用“<<”为输入元素赋值，其实你也可以使用它们的 setText 方法。

at 方法则完全是由框架提供的，当我们导航至对应的页面时，Geb 将会自动对其进行断言。

接下来，我们创建 SearchResultPage：

```
package geb.pages

import geb.Page

class SearchResultPage extends Page {
    static url = '/search'
    static at = { $('h2', 0).text().startsWith('Tweet results for') }
    static content = {
        resultList { $('ul.collection') }
```

```
        results { resultList.find('li') }
    }
}
```

它非常简短，这要归功于能够重用之前为搜索结果所定义的内容。

页面对象构建完成之后，我们可以按照如下的方式来编写测试：

```
import geb.Configuration
import geb.pages.LoginPage
import geb.pages.ProfilePage
import geb.pages.SearchResultPage
import geb.spock.GebSpec
import masterSpringMvc.MasterSpringMvcApplication
import masterSpringMvc.auth.StubSocialSigninConfig
import masterSpringMvc.search.StubTwitterSearchConfig
import org.springframework.beans.factory.annotation.Value
import org.springframework.boot.test.SpringApplicationContextLoader
import org.springframework.boot.test.WebIntegrationTest
import org.springframework.test.context.ContextConfiguration

@ContextConfiguration(loader = SpringApplicationContextLoader,
        classes = [MasterSpringMvcApplication,
StubTwitterSearchConfig, StubSocialSigninConfig])
@WebIntegrationTest(randomPort = true)
class IntegrationSpec extends GebSpec {

    @Value('${local.server.port}')
    int port
    Configuration createConf() {
        def configuration = super.createConf()
        configuration.baseUrl = "http://localhost:$port"
        configuration
    }

    def "User is redirected to the login page when not logged"() {
        when: "I navigate to the home page"
        go '/'

        then: "I am redirected to the login page"
        $('h2').text() == 'Login'
    }

    def "User is redirected to its profile on his first visit"() {
```

```
            when: 'I am connected'
            to LoginPage
            loginWithTwitter()

            and: "I navigate to the home page"
            go '/'

            then: "I am redirected to the profile page"
            $('h2').text() == 'Your profile'
        }

        def "After filling his profile, the user is taken to result
    matching his tastes"() {
            given: 'I am connected'
            to LoginPage
            loginWithTwitter()

            and: 'I am on my profile'
            to ProfilePage

            when: 'I fill my profile'
            fillInfos("geowarin", "geowarin@mymail.com", "03/19/1987");
            addTaste("spring")

            and: 'I save it'
            saveProfile()
            then: 'I am taken to the search result page'
            at SearchResultPage
            page.results.size() == 2
        }
    }
```

哇，看上去非常棒！你当然可以直接使用 Geb 来编写自己的用户故事！

我们这些简单的测试只是接触到了 Geb 的皮毛，还有非常多的功能可以使用，我建议你阅读一下“Book of Geb”，这是一个非常棒的文档，可以通过 http://www.gebish.org/manual/current/来获取。

7.11 检查点

在本章中，我们在 src/test/java 中添加了很多的测试，我选择了使用 Groovy，因此删除

了重复的测试内容，如图 7-3 所示。

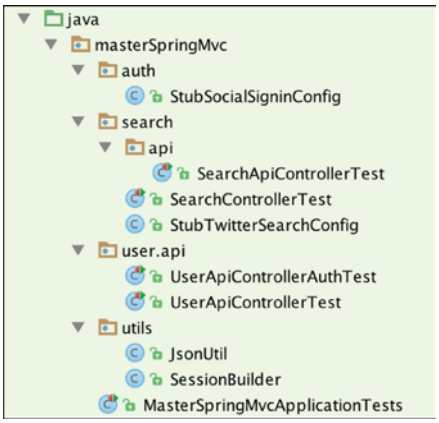

图 7-3

在 src/test/groovy 目录下，我重构的两个测试如图 7-4 所示。

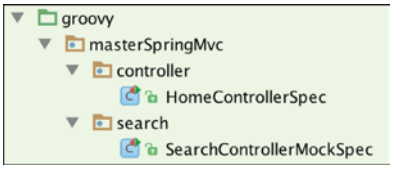

图 7-4

在 src/integrationTest/groovy 目录下，包含了使用 Geb 编写的集成测试，如图 7-5 所示。

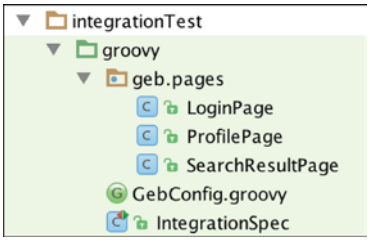

图 7-5

最后，我们在 Gradle 构建中新增了一个名为 `integrationTest` 的任务。运行 `gradle clean test` 和 `gradle clean integrationTest` 可以确保你的测试都能通过。

如果构建成功的话，我们就可以准备学习下一章了。

7.12 小结

在本章中，我们学习了单元测试和集成测试的区别。

我们讨论了测试为什么是一个好的习惯，它能够让我们对所构建和交付的程序更有信心。长期来看，它能够节省金钱并且不会让我们陷入令人头疼的问题之中。

Spring 能够与 Java 编写的经典 JUnit 测试很好地协作，并对集成测试提供了良好的支持。但是，我们还可以非常容易地使用其他语言，如 Groovy，从而能够让测试的可读性更高，并且易于编写。

测试是 Spring 框架最强的功能之一，这也是采用依赖注入的一个重要原因。

在下一章中，我们将会对应用进行优化，这样它能够更好地部署在云端！

第 8 章
优化请求

在本章中，我们将会看一些提升应用性能的技术。

我们将会实现优化 Web 应用的典型技术：缓存控制头信息、Gzip、应用缓存和 ETag，还会看一些反应型（reactive）相关的技术，如异步方法调用和 WebSocket。

8.1 生产环境的 profile

在前面的章节中，我们看到了如何让应用的属性文件仅在特定 profile 激活的情况下才能发挥作用。我们将会采用相同的方式，并在 src/main/ resources 中创建 application-prod.properties 文件，这个文件会与已有的 application.properties 位于相同的目录中。这样的话，我们就可以通过可选的设置项配置生产环境了。

我们首先会在这个文件中添加一些属性。在第 3 章中，我们禁用了 Thymeleaf 缓存并强制翻译 bundle 每次访问都会进行重新加载。

对于开发过程来说，这是很好的，但是对于生成环境来说，这没有什么用处，而且还耗费时间。所以，我们来解决这个问题：

```
spring.thymeleaf.cache=true
spring.messages.cache-seconds=-1
```

将缓存时间设置为-1 意味着永远缓存 bundle。

现在，如果我们使用“prod”profile 启动应用的话，模板和 bundle 将会永远进行缓存。

来自“prod”profile 的属性将会重写 `application.properties` 文件中所声明的属性。

8.2 Gzip

Gzip 是一种能够被浏览器广泛理解的压缩算法。服务器会提供压缩的响应，这耗费的 CPU 周期会稍微多一点，但是能够节省带宽。

客户端浏览器要负责解压资源并将其展现给用户。

要使用 Tomcat 的 Gzip 功能，只需在 application-prod.properties 文件中添加如下的几行配置：

```
server.tomcat.compression=on
server.tomcat.compressableMimeTypes=text/html,text/xml,text/css,text/
plain,\
  application/json,application/xml,application/javascript
```

当所请求的文件匹配列表中的 MIME 类型并且其长度大于 2048 字节，将会启用 Tomcat 的 Gzip 压缩。你可以将 server.tomcat.compression 设置为 force，这样的话，会强制启用压缩功能，如果你想要修改 Gzip 资源的最小长度的话，也可以将其设置为一个数字值。

如果你想对压缩进行更多的控制，例如压缩级别或者对某些用户终端不启用压缩功能，那么可以使用 Jetty 中的 GzipFilter 类，这需要将 org.eclipse.jetty:jetty-servlets 依赖添加到项目之中。

这将会自动触发 GzipFilterAutoConfiguration 类，这个类可以通过一些带有 spring.http.gzip 前缀的属性来进行配置。你可以参考 GzipFilterProperties 类以了解它能进行哪些自定义的设置。

读者可以参考 http://docs.spring.io/spring-boot/docs/current/reference/html/howto-embedded-servlet-containers.html#how-to-enable-http-response- compression 以了解更多信息。

8.3 缓存控制

缓存控制由服务端发送的一组 HTTP 头信息，它会控制用户的浏览器如何缓存资源。

在前面的章节中，我们看到 Spring Security 会自动禁用安全资源的缓存。

如果我们想从缓存控制中受益的话，首先就要关闭 Spring Security 的这个特性：

```
security.headers.cache=false

# Cache resources for 3 days
spring.resources.cache-period=259200
```

现在，启动应用，访问主页并打开 Chrome 的开发人员控制台，能够看到我们的 JavaScript 文件进行了 Gzip 压缩和缓存，如图 8-1 的截屏所示：

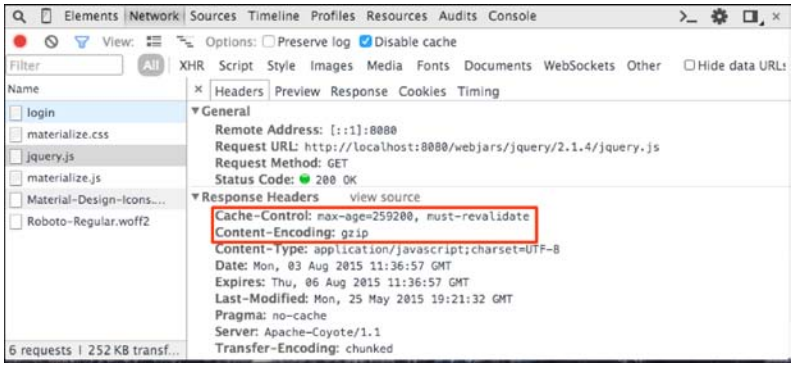

图 8-1

如果你希望对缓存进行更多控制的话，那么应该在配置中为你的资源添加自己的处理器：

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    // This is just an example
    registry.addResourceHandler("/img/**")
            .addResourceLocations("classpath:/static/images/")
            .setCachePeriod(12);
}
```

我们也可以重写 Spring Security 的默认设置。如果我们希望禁用 API 的“无缓存控制”策略，那么可以按照如下的方式来修改 ApiSecurityConfiguration 类：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .antMatcher("/api/**")
// This is just an example – not required in our case
        .headers().cacheControl().disable()
        .httpBasic().and()
```

```
            .csrf().disable()
            .authorizeRequests()
            .antMatchers(HttpMethod.GET).hasRole("USER")
            .antMatchers(HttpMethod.POST).hasRole("ADMIN")
            .antMatchers(HttpMethod.PUT).hasRole("ADMIN")
            .antMatchers(HttpMethod.DELETE).hasRole("ADMIN")
            .anyRequest().authenticated();
    }
```

8.4 应用缓存

现在我们的 Web 请求已经进行了压缩和缓存，在减少服务器负载方面我们所能做的下一件事情就是将耗时操作的结果放到缓存之中。Twitter 搜索会耗费一定的时间，并且会占用我们请求 Twitter API 的比例。通过使用 Spring，我们能够很容易地缓存结果，当使用相同的参数调用搜索功能时，都会返回相同的结果。

我们需要做的第一件事就是使用@EnableCache 注解激活 Spring 的缓存功能，还需要创建一个 CacheManager 来处理缓存。我们在 config 包中创建 `CacheConfiguration` 类：

```
package masterSpringMvc.config;

import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.concurrent.ConcurrentMapCache;
import org.springframework.cache.support.SimpleCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Arrays;
@Configuration
@EnableCaching
public class CacheConfiguration {

    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager simpleCacheManager = new
SimpleCacheManager();
        simpleCacheManager.setCaches(Arrays.asList(
                new ConcurrentMapCache("searches")
        ));
```

```
            return simpleCacheManager;
        }
    }
```

在这个样例中，使用了最简单的缓存抽象。其实，还有其他可用的实现，如 EhCacheCacheManager 或 GuavaCacheManager，稍后我们就会用到它们。

现在，我们配置了缓存，接下来就可以在方法上使用@Cacheable 注解了。这样使用了之后，Spring 会自动缓存方法的结果，并将其与当前的参数进行关联，以便于后续的查询。

对于方法被缓存的 bean，Spring 会创建一个代理。这通常意味着，如果我们在同一个 bean 中调用被缓存方法时，将会无法使用 Spring 缓存功能。

在我们的样例中，执行搜索功能的 SearchService 类能够从缓存功能中受益很多。

作为初始的准备环节，我们最好将负责创建 SearchParameters 的代码放到一个单独的对象中，将其命名为 SearchParamsBuilder：

```
package masterSpringMvc.search;

import org.springframework.social.twitter.api.SearchParameters;

import java.util.List;
import java.util.stream.Collectors;

public class SearchParamsBuilder {

    public static SearchParameters createSearchParam(String
searchType, String taste) {
        SearchParameters.ResultType resultType =
getResultType(searchType);
        SearchParameters searchParameters = new
SearchParameters(taste);
        searchParameters.resultType(resultType);
        searchParameters.count(3);
        return searchParameters;
    }

    private static SearchParameters.ResultType getResultType(String
searchType) {
        for (SearchParameters.ResultType knownType : SearchParameters.
ResultType.values()) {
            if (knownType.name().equalsIgnoreCase(searchType)) {
```

```
                return knownType;
            }
        }
        return SearchParameters.ResultType.RECENT;
    }
}
```

这会帮助我们在服务中创建搜索参数。

现在，我们想要为搜索结果创建缓存，希望每次对 Twitter API 的调用能够缓存起来。Spring 的缓存注解依赖于代理机制，借助代理，能够为带有`@Cacheable`注解的方法提供缓存功能。因此，我们需要有一个新类，其方法要带有`@Cacheable`注解。

当我们使用 Spring 抽象 API 的时候，并不了解缓存的底层实现是什么。有很多的实现都需要返回类型和缓存方法的参数都是可序列化的。

`SearchParameters` 不是可序列化的，这就是为什么我会将搜索类型和关键字（都是 String 类型）传递给缓存方法。

因为我们希望将 `LightTweets` 对象放在缓存中，因此需要将其变为可序列化的，这样它就能够从任意的缓存抽象中进行写入和读取操作：

```
public class LightTweet implements Serializable {
    // the rest of the code remains unchanged
}
```

接下来，我们创建 SearchCache 类，并将其放到 search.cache 包中：

```
package masterSpringMvc.search.cache;

import masterSpringMvc.search.LightTweet;
import masterSpringMvc.search.SearchParamsBuilder;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.social.
TwitterProperties;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.social.twitter.api.SearchParameters;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.social.twitter.api.impl.TwitterTemplate;
import org.springframework.stereotype.Service;
```

```
import java.util.List;
import java.util.stream.Collectors;

@Service
public class SearchCache {
    protected final Log logger = LogFactory.getLog(getClass());
    private Twitter twitter;

    @Autowired
    public SearchCache(TwitterProperties twitterProperties) {
        this.twitter = new TwitterTemplate(twitterProperties.
getAppId(), twitterProperties.getAppSecret());
    }

    @Cacheable("searches")
    public List<LightTweet> fetch(String searchType, String keyword) {
        logger.info("Cache miss for " + keyword);
        SearchParameters searchParam = SearchParamsBuilder.
createSearchParam(searchType, keyword);
        return twitter.searchOperations()
                .search(searchParam)
                .getTweets().stream()
                .map(LightTweet::ofTweet)
                .collect(Collectors.toList());
    }
}
```

这实在是太简单了！我们使用@Cacheable 注解来指定所使用的缓存名称。不同的缓存可能会有不同的策略。

注意，我们这里手动创建了 TwitterTemplate，而不是像之前那样将其注入进来。这是因为我们稍后会在其他的线程中访问缓存。在 Spring Boot 的 `TwitterAutoConfiguration` 类中，Twitter bean 是绑定在 request 作用域的，因此在 Servlet 线程以外，无法访问到它。

有了这两个新对象，`SearchService` 类就简化成了如下的形式：

```
package masterSpringMvc.search;

import masterSpringMvc.search.cache.SearchCache;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;
```

```
import java.util.List;
import java.util.stream.Collectors;

@Service
@Profile("!async")
public class SearchService implements TwitterSearch {
    private SearchCache searchCache;

    @Autowired
    public SearchService(SearchCache searchCache) {
        this.searchCache = searchCache;
    }

    @Override
    public List<LightTweet> search(String searchType, List<String>
keywords) {
        return keywords.stream()
                .flatMap(keyword -> searchCache.fetch(searchType,
keyword).stream())
                .collect(Collectors.toList());
    }
}
```

需要注意的是，我们为这个服务添加了@Profile("!async")注解，这意味着只有 async profile 没有激活的情况下，才会创建这个 bean。

稍后，我们将会创建 `TwitterSearch` 类的另外一个实现，这样能够在它们之间进行切换。

好了！重新启动应用并尝试如下的请求：

```
http://localhost:8080/search/mixed;keywords=docker,spring,spring%20
boot,spring%20mvc,groovy,grails
```

它第一次会耗费一些时间，然后在控制台上能够展现出如下的日志：

```
2015-08-03 16:04:01.958 INFO 38259 --- [nio-8080-exec-8] m.search.cache.
SearchCache              : Cache miss for docker
2015-08-03 16:04:02.437 INFO 38259 --- [nio-8080-exec-8] m.search.cache.
SearchCache              : Cache miss for spring
2015-08-03 16:04:02.728 INFO 38259 --- [nio-8080-exec-8] m.search.cache.
SearchCache              : Cache miss for spring boot
2015-08-03 16:04:03.098 INFO 38259 --- [nio-8080-exec-8] m.search.cache.
SearchCache              : Cache miss for spring mvc
2015-08-03 16:04:03.383 INFO 38259 --- [nio-8080-exec-8] m.search.cache.
```

```
SearchCache              : Cache miss for groovy
2015-08-03 16:04:03.967 INFO 38259 --- [nio-8080-exec-8] m.search.cache.
SearchCache              : Cache miss for grails
```

在此之后，如果我们点击刷新的话，结果将会瞬间展现，并且控制台上不会有缓存缺失的日志。

我们的缓存功能到此为止就可以了，但是缓存 API 并不限于这些。我们可以使用如下的注解来标注方法。

- `@CachEvict`：将会从缓存中移除一个条目。
- `@CachePut`：会将方法的结果放到缓存中，而不会影响到方法调用本身。
- `@Caching`：将缓存注解进行重分组。
- `@CacheConfig`：指向不同的缓存配置。

`@Cacheable` 注解也可以配置为缓存特定条件下的结果。

关于 Spring 缓存的更多信息，请参考如下的文档：http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html。

8.4.1 缓存失效

目前，搜索结果将会永远保留在缓存之中。默认的简化缓存管理器并没有为我们提供太多的可选项。我们还要做另外一件事情来提升应用的缓存功能。鉴于 Guava 已经存在于项目的类路径之中，我们可以在缓存配置中使用如下的代码来替换已有的缓存管理器：

```
package masterSpringMvc.config;

import com.google.common.cache.CacheBuilder;
import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.cache.guava.GuavaCacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.concurrent.TimeUnit;

@Configuration
```

```
@EnableCaching
public class CacheConfiguration {

    @Bean
    public CacheManager cacheManager() {
        GuavaCacheManager cacheManager = new
GuavaCacheManager("searches");
        cacheManager
                .setCacheBuilder(
                        CacheBuilder.newBuilder()
                                .softValues()
                                .expireAfterWrite(10, TimeUnit.
MINUTES)
                );
        return cacheManager;
    }
}
```

通过这种方式我们构建了 10 分钟过期的缓存，并且使用了软引用的值，这样的话当 JVM 内存不足的时候会将缓存条目清理掉。

你可以试着探索一下 Guava 缓存构造器的功能，可以为测试设置更小的时间单元，甚至指定不同的缓存策略。

参考如下地址的文档以了解更多信息：https://code.google.com/p/guava-libraries/wiki/CachesExplained。

8.4.2 分布式缓存

我们已经有了一个 Redis profile。如果 Redis 可用的话，我们可以将其作为缓存提供者。这样的话就能跨多台服务器，实现分布式的缓存。首先，修改一下 `RedisConfig` 类：

```
package masterSpringMvc.config;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cache.CacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.Profile;
import org.springframework.data.redis.cache.RedisCacheManager;
```

```
import org.springframework.data.redis.connection.
RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.session.data.redis.config.annotation.web.
http.EnableRedisHttpSession;

import java.util.Arrays;

@Configuration
@Profile("redis")
@EnableRedisHttpSession
public class RedisConfig {

    @Bean(name = "objectRedisTemplate")
    public RedisTemplate objectRedisTemplate(RedisConnectionFactory
redisConnectionFactory) {
        RedisTemplate<Object, Object> template = new
RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
    @Primary @Bean
    public CacheManager cacheManager(@Qualifier("objectRedisTemplate")
RedisTemplate template) {
        RedisCacheManager cacheManager = new
RedisCacheManager(template);
        cacheManager.setCacheNames(Arrays.asList("searches"));
        cacheManager.setDefaultExpiration(36_000);
        return cacheManager;
    }
}
```

通过该配置，如果我们使用“Redis”profile 的话，将会启用 Redis 缓存管理器来代替 `CacheConfig` 类中所定义的缓存管理器，因为这里使用了`@Primary` 注解。

如果我们希望扩展到多台服务器的话，通过这种方式就能实现分布式的缓存。Redis 模板会用来序列化方法的返回值及其参数，因此需要这些对象实现 `Serializable` 接口。

8.5　异步方法

在我们的应用中，依然还有一个瓶颈。当用户搜索 10 个关键字的时候，每个搜索都会

串行执行。我们可以通过使用不同的线程来快速提升应用的速度，让所有的搜索同时开始执行。

要启用 Spring 的异步功能，必须要使用@EnableAsync 注解。这样将会透明地使用 java.util.concurrent.Executor 来执行所有带@Async 注解的方法。

通过实现 AsyncConfigurer 接口，可以自定义默认的执行器（executor）。我们在 config 包中创建一个名为 AsyncConfig 的新配置类：

```
package masterSpringMvc.config;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.interceptor.
AsyncUncaughtExceptionHandler;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.AsyncConfigurer;
import org.springframework.scheduling.annotation.EnableAsync;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

@Configuration
@EnableAsync
public class AsyncConfiguration implements AsyncConfigurer {

    protected final Log logger = LogFactory.getLog(getClass());

    @Override
    public Executor getAsyncExecutor() {
        return Executors.newFixedThreadPool(10);
    }

    @Override
    public AsyncUncaughtExceptionHandler
getAsyncUncaughtExceptionHandler() {
        return (ex, method, params) -> logger.error("Uncaught async
error", ex);
    }
}
```

借助这个配置，我们能够确保在应用中，用来处理异步任务的线程不会超过 10 个。这对 Web 应用来说是很重要的，因为每个客户端都会有一个专用的线程。你所使用的线程越

多，阻塞时间越长，那么能够处理的客户端就会越少。

接下来，我们为搜索方法添加注解，使其变成异步方法。我们需要让它返回 `Future` 的一个子类型，这是 Java 的一个并发类，代表了异步的结果。

我们创建 TwitterSearch 类的一个新版本实现，它会在不同的线程中调用搜索 API。实现类会有点复杂，因此我将其拆分为很小的组成部分来进行讲解。

首先，我们要为调用 API 的方法添加@Async 注解，这样就会告诉 Spring 要使用我们的执行器来调度该任务。同样，Spring 会使用代理来完成这项神奇的任务，因此这个方法要放到单独的一个类中，不能放到调用它的服务中。如果这个组件能够使用我们的缓存也是很不错的。所以，我们会按照如下的方式来创建这个组件：

```
@Component
private static class AsyncSearch {
    protected final Log logger = LogFactory.getLog(getClass());
    private SearchCache searchCache;

    @Autowired
    public AsyncSearch(SearchCache searchCache) {
        this.searchCache = searchCache;
    }

    @Async
    public ListenableFuture<List<LightTweet>> asyncFetch(String
searchType, String keyword) {
        logger.info(Thread.currentThread().getName() + " - Searching
for " + keyword);
        return new AsyncResult<>(searchCache.fetch(searchType,
keyword));
    }
}
```

先不要创建这个类，我们首先看一些服务的需求。

`ListenableFuture` 抽象允许我们在 Future 完成的时候添加回调，不管得到正确的结果还是出现异常均会对它进行调用。

等待一批异步任务的算法可以写成如下的形式：

```
@Override
public List<LightTweet> search(String searchType, List<String>
keywords) {
```

```
        CountDownLatch latch = new CountDownLatch(keywords.size());
        List<LightTweet> allTweets = Collections.synchronizedList(new
    ArrayList<>());
        keywords
                .stream()
                .forEach(keyword -> asyncFetch(latch, allTweets,
    searchType, keyword));

        await(latch);
        return allTweets;
    }
```

如果你不了解 `CountDownLatch` 的话，可以将其理解为简单的阻塞队列。

`await()`方法将会一直等待，直到 latch 达到 0，从而解锁该线程。

前面的代码会为每个 `asynFetch` 方法设置一个回调。这个回调会将结果添加到 `allTweets` 列表中，并对 latch 的计数进行递减。当所有的回调均被调用之后，这个方法就会返回所有的 Tweet。

明白了吗？如下是完整的最终代码：

```
package masterSpringMvc.search;

import masterSpringMvc.search.cache.SearchCache;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.scheduling.annotation.Async;
import org.springframework.scheduling.annotation.AsyncResult;
import org.springframework.social.twitter.api.SearchParameters;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;
import org.springframework.util.concurrent.ListenableFuture;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;

@Service
@Profile("async")
public class ParallelSearchService implements TwitterSearch {
```

```
    private final AsyncSearch asyncSearch;

    @Autowired
    public ParallelSearchService(AsyncSearch asyncSearch) {
        this.asyncSearch = asyncSearch;
    }

    @Override
    public List<LightTweet> search(String searchType, List<String>
keywords) {
        CountDownLatch latch = new CountDownLatch(keywords.size());
        List<LightTweet> allTweets = Collections.synchronizedList(new
ArrayList<>());
        keywords
                .stream()
                .forEach(keyword -> asyncFetch(latch, allTweets,
searchType, keyword));

        await(latch);
        return allTweets;
    }

    private void asyncFetch(CountDownLatch latch, List<LightTweet>
allTweets, String searchType, String keyword) {
        asyncSearch.asyncFetch(searchType, keyword)
                .addCallback(
                        tweets -> onSuccess(allTweets, latch, tweets),
                        ex -> onError(latch, ex));
    }

    private void await(CountDownLatch latch) {
        try {
            latch.await();
        } catch (InterruptedException e) {
            throw new IllegalStateException(e);
        }
    }

    private static void onSuccess(List<LightTweet> results,
CountDownLatch latch, List<LightTweet> tweets) {
        results.addAll(tweets);
        latch.countDown();
    }
```

```
    private static void onError(CountDownLatch latch, Throwable ex) {
        ex.printStackTrace();
        latch.countDown();
    }

    @Component
    private static class AsyncSearch {
        protected final Log logger = LogFactory.getLog(getClass());
        private SearchCache searchCache;

        @Autowired
        public AsyncSearch(SearchCache searchCache) {
            this.searchCache = searchCache;
        }

        @Async
        public ListenableFuture<List<LightTweet>> asyncFetch(String
searchType, String keyword) {
            logger.info(Thread.currentThread().getName() + " -
Searching for " + keyword);
            return new AsyncResult<>(searchCache.fetch(searchType,
keyword));
        }
    }
}
```

现在，要使用这个实现的话，我们需要基于 async profile 来运行应用。

我们也可以同时使用多个 profile，只需将它们用逗号分隔就可以了，如下所示：

```
--spring.profiles.active=redis,async
```

如果我们搜索多个关键词的话，将会看到如下的效果：

```
pool-1-thread-3 - Searching groovy
pool-1-thread-1 - Searching spring
pool-1-thread-2 - Searching java
```

这表示不同的搜索在并行执行。

Java 8 实际上引入了一个名为 CompletableFuture 的新类型，它是操作 Future 的更好的 API。CompletableFuture 的主要问题在于，与它协作的所有执行器都要编写一些代码。这超

出了本书的范围。关于这个话题，你可以查看我的博客：http://geowarin.github.io/spring/2015/06/12/completable-futures-with-spring-async.html。

免责声明

下面的章节包含了许多的 JavaScript 代码。我认为你应该看一下这些代码，如果你之前不怎么喜欢 JavaScript 的话，更应如此，现在该学一下这门语言了。话又说回来，尽管 WebSocket 非常酷，但它并不是强制要求掌握的。到这里，你可以安心地越过本章，看一下如何对应用进行部署。

8.6 ETag

我们对 Twitter 结果已经进行了缓存，所以当用户刷新结果页面的时候不会对 Twitter API 执行额外的搜索。但是，即便结果没有发生变化，但结果本身还是会多次发送给用户，这会浪费带宽。

ETag 是 Web 响应数据的一个散列（hash），并且会在头信息中进行发送。客户端可以记住资源的 ETag，并且通过 `If-None-Match` 头信息将最新的已知版本发送给服务器。如果在这段时间内请求没有发生变化的话，服务器就可以回应 `304 Not Modified`。

Spring 有一个特殊的 Servlet 过滤器来处理 ETag，名为 `ShallowEtagHeaderFilter`。我们只需将其作为一个 bean 添加到 `MasterSpringMvc4Application` 配置类中：

```
@Bean
public Filter etagFilter() {
    return new ShallowEtagHeaderFilter();
}
```

只要响应没有缓存控制头信息的话，系统将会自动为你的响应生成 ETag。

如果我们访问 RESTful API 的话，可以看到 ETag 会与服务器端的响应一起发送：

```
> http GET 'http://localhost:8080/api/search/mixed;keywords=spring' -a
admin:admin
HTTP/1.1 200 OK
Content-Length: 1276
Content-Type: application/json;charset=UTF-8
```

```
Date: Mon, 01 Jun 2015 11:29:51 GMT
ETag: "00a66d6dd835b6c7c60638eab976c4dd7"
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=662848E4F927EE9A1BA2006686ECFE4C; Path=/; HttpOnly
```

如果我们再次请求相同的资源，并且将已知的最新 ETag 放到 `If-None-Match` 头信息中，服务器将会自动响应 `304 Not Modified` 状态：

```
> http GET 'http://localhost:8080/api/search/mixed;keywords=spring' If-
None-Match:'"00a66d6dd835b6c7c60638eab976c4dd7"' -a admin:admin
HTTP/1.1 304 Not Modified
Date: Mon, 01 Jun 2015 11:34:21 GMT
ETag: "00a66d6dd835b6c7c60638eab976c4dd7"
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=CA956010CF268056C241B0674C6C5AB2; Path=/; HttpOnly
```

因为我们的搜索具有并行的特点，所以针对不同关键字所获取到的 Tweet 在顺序上可能会有所差异，这会导致 ETag 的变化。如果你希望针对多项搜索依然能够使用该技术的话，可以考虑在将搜索结果发送给客户端之前先进行一下排序。

如果我们想利用这一特性的话，显然需要重写客户端的代码。我们将会看到一个使用 jQuery 的简单方案，它会将用户最新的查询结果放到浏览器的本地存储中。

首先，从模型中移除 `tweets` 变量，我们不再从服务端进行查询。为了反映这种变化，你可能需要修改一两个测试用例的代码。

在进行下一步之前，我们将 lodash 添加到 JavaScript 库中。如果你不了解 lodash 的话，可以将其想象为 JavaScript 领域的 Apache Utils 库。你可以按照如下的方式将其添加到项目依赖中：

```
compile 'org.webjars.bower:lodash:3.9.3'
```

将其添加到 default.html 布局中，放到 Materialize JavaScript 脚本的后面：

```
<script src="/webjars/lodash/3.9.3/lodash.js"></script>
```

我们修改 resultPage.html，将应该展现 Tweet 的区域留空：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
```

```
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
      layout:decorator="layout/default">
<head lang="en">
    <title>Hello twitter</title>
</head>
<body>
<div class="row" layout:fragment="content">

    <h2 class="indigo-text center" th:text="|Tweet results for
${search}|">Tweets</h2>
    <ul id="tweets" class="collection">
    </ul>
</div>
</body>
</html>
```

然后，在页面的底部添加如下的脚本，加到 body 闭合标签的前面：

```
<script layout:fragment="script" th:inline="javascript">
    /*<![CDATA[*/
    var baseUrl = /*[[@{/api/search}]]*/ "/";
    var currentLocation = window.location.href;
    var search = currentLocation.substr(currentLocation.
lastIndexOf('/'));
    var url = baseUrl + search;
    /*]]>*/
</script>
```

这段脚本会负责构建请求的 URL，我们会在一个简单的 jQuery AJAX 调用中用到它：

```
$.ajax({
    url: url,
    type: "GET",
    beforeSend: setEtag,
    success: onResponse
});
```

借助 beforeSend 回调，我们在调用发起之前能够有机会改变请求的头信息：

```
function getLastQuery() {
    return JSON.parse(localStorage.getItem('lastQuery')) || {};
}
```

```
function storeQuery(query) {
    localStorage.setItem('lastQuery', JSON.stringify(query));
}

function setEtag(xhr) {
    xhr.setRequestHeader('If-None-Match', getLastQuery().etag)
}
```

可以看到，我们能够非常容易地读取和写入本地存储。这里的问题在于本地存储只能使用字符串，所以我们必须将查询对象解析并序列化为 JSON。

如果 HTTP 状态是 `304 Not Modified` 的话，我们就可以从本地缓存中获取内容，以此来处理响应：

```
function onResponse(tweets, status, xhr) {
  if (xhr.status == 304) {
      console.log('Response has not changed');
      tweets = getLastQuery().tweets
  }

  var etag = xhr.getResponseHeader('Etag');
  storeQuery({tweets: tweets, etag: etag});

  displayTweets(tweets);
}

function displayTweets(tweets) {
  $('#tweets').empty();
  $.each(tweets, function (index, tweet) {
      addTweet(tweet);
  })
}
```

在下面的 addTweet 函数中，我会用到 lodash，这是一个非常有用的 JavaScript 工具库，用来生成模板。为页面添加 Tweet 的函数如下所示：

```
function addTweet(tweet) {
    var template = _.template('<li class="collection-item avatar">' +
        '<img class="circle" src="${tweet.profileImageUrl}" />' +
        '<span class="title">${tweet.user}</span>' +
        '<p>${tweet.text}</p>' +
        '</li>');
```

```
        $('#tweets').append(template({tweet: tweet}));
    }
```

这里有很多的 JavaScript 脚本。如果在一个单页面应用程序（Single Page Application）中，采用像 Backbone.js 这样的库，那么这种模式会更好一些。不过，这可以作为一个简单的样例，用来阐述如何在应用中实现 ETag 功能。

如果你多次刷新搜索页面的话，将会看到内容没有什么变化，并且会马上展现出来，如图 8-2 所示。

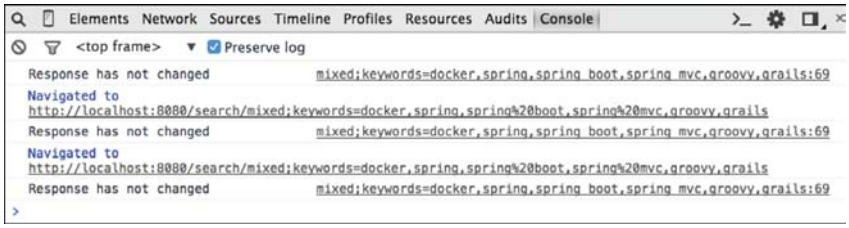

图 8-2

ETag 还有其他的用途，例如事务的乐观锁（它能够让你在任意时间都知道客户端该使用哪个版本的对象）。在发送数据之前要对其进行散列操作，这会给服务端带来额外的工作，但这种方式会节省带宽。

8.7　WebSocket

我们能够想到的另外一种优化方式就是在服务器端有可用数据时，就立即将其发送到客户端。我们是通过多线程的方式来获取搜索结果的，所以数据会分为多个块。我们可以一点点地进行发送，而不必等待所有的结果。

Spring 为 WebSocket 提供了良好的支持，WebSocket 协议允许客户端维持与服务器的长连接。数据可以通过 WebSocket 在这两个端点之间进行双向传输，因此消费数据的一方能够实时获取数据。

我们将会使用名为 SockJS 的 JavaScript 库，从而确保在所有浏览器中的兼容性。如果用户使用过时的浏览器，Sockjs 将会透明地切换为备用策略。

我们还会使用 StompJS 来连接消息代理（message broker）。

在构建过程中，添加如下的库：

```
compile 'org.springframework.boot:spring-boot-starter-websocket'
```

```
compile 'org.springframework:spring-messaging'

compile 'org.webjars:sockjs-client:1.0.0'
compile 'org.webjars:stomp-websocket:2.3.3'
```

在默认的 Thymeleaf 模板中，添加如下的 WebJars：

```
<script src="/webjars/sockjs-client/1.0.0/sockjs.js"></script>
<script src="/webjars/stomp-websocket/2.3.3/stomp.js"></script>
```

为了在应用中使用 WebSocket 功能，我们还需要添加一点配置：

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfiguration extends
AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/ws");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry)
{
        registry.addEndpoint("/twitterSearch").withSockJS();
    }
}
```

这会为我们的应用配置不同的可用通道。SockJS 客户端将会连接到 twitterSearch 端点，将数据推送至服务端的“/ws/”通道，并且还能监听“/topic/”来跟踪变化。

这样我们就能将 SimpMessagingTemplate 注入到新的控制器中，并使用它在“/topic/searchResult”通道上给客户端推送数据，如下所示：

```
@Controller
public class SearchSocketController {
    private CachedSearchService searchService;
    private SimpMessagingTemplate webSocket;

    @Autowired
    public SearchSocketController(CachedSearchService searchService,
```

```
SimpMessagingTemplate webSocket) {
        this.searchService = searchService;
        this.webSocket = webSocket;
    }
    @MessageMapping("/search")
    public void search(@RequestParam List<String> keywords) throws
Exception {
        Consumer<List<LightTweet>> callback = tweet -> webSocket.
convertAndSend("/topic/searchResults", tweet);
        twitterSearch(SearchParameters.ResultType.POPULAR, keywords,
callback);
    }

    public void twitterSearch(SearchParameters.ResultType resultType,
List<String> keywords, Consumer<List<LightTweet>> callback) {
        keywords.stream()
            .forEach(keyword -> {
                searchService.search(resultType, keyword)
                    .addCallback(callback::accept,
Throwable::printStackTrace);
            });
    }
}
```

在 resultPage 页面，JavaScript 的代码也很简单：

```
var currentLocation = window.location.href;
var search = currentLocation.substr(currentLocation.lastIndexOf('=') +
1);

function connect() {
  var socket = new SockJS('/hello');
  stompClient = Stomp.over(socket);
  // stompClient.debug = null;
  stompClient.connect({}, function (frame) {
      console.log('Connected: ' + frame);

      stompClient.subscribe('/topic/searchResults', function (result)
{
          displayTweets(JSON.parse(result.body));
      });

      stompClient.send("/app/search", {}, JSON.stringify(search.
```

```
split(',')));
  });
}
```

displayTweets 函数本质上是与之前完全相同的：

```
function displayTweets(tweets) {
    $.each(tweets, function (index, tweet) {
        addTweet(tweet);
    })
}

function addTweet(tweet) {
    var template = _.template('<li class="collection-item avatar">' +
        '<img class="circle" src="${tweet.profileImageUrl}" />' +
        '<span class="title">${tweet.userName}</span>' +
        '<p>${tweet.text}</p>' +
        '</li>');

    $('#tweets').append(template({tweet: tweet}));
}
```

到此为止就完成了！客户端能够接受应用中所有搜索的结果——这个过程是实时的！

在将其用在生产环境之前，还有一些额外的工作。基本的思路如下所示：

- 为客户端创建子通道，私密地监听变更；
- 客户端完成使用之后，关闭通道；
- 为新的 Tweet 添加 CSS 转换效果，让用户感觉到它是实时的；
- 使用像 RabbitMQ 这样真正的代理，允许后端能够随着连接的增加进行扩展。

WebSocket 有很多的内容，不是这个简单样例所能涵盖的。别忘了参考 http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html 上的文档来了解更多的信息。

8.8 检查点

在本章中，我们创建了两个新的配置：AsyncConfiguration 能够让我们使用@Async 注解将任务提交到执行器中；CacheConfiguration 会创建 CacheManager 接口的实现，并且能

够让我们使用@Cacheable 注解。因为我们可以使用 Redis 作为缓存管理器，因此修改了 RedisConfig 类。

我们创建了 SearchCache 类，它包含了 Tweet 的缓存。另外，现在有两个 TwitterSearch 实现可供选择：旧的 SearchService 会同步获取结果；ParallelSearchService 会将每个查询放到不同的线程中，如图 8-3 所示。

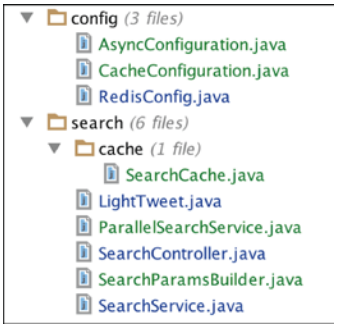

图 8-3

8.9　小结

在本章中，我们看到了提升性能的两种不同的哲学。首先，我们试图减少客户端所使用的带宽，这是通过缓存数据以及使用尽可能少的连接来实现的。

不过，在第二部分中，我们使用了一些更高级的方法，这包括允许搜索并行执行，通过 WebSocket 为每个客户端保持一个同步的持久化连接。这样客户端能够实时获取更新，应用的响应性会更好，但是这样使用的线程会更多。

我强烈建议你在进入下一章之前掌握好这些内容，这有利于应用的部署！

第 9 章
将 Web 应用部署到云中

在本章中，我们将会看到不同的云提供商，理解分布式架构的挑战和收益，学习如何将 Web 应用部署到 Pivotal Web Services 和 Heroku 中。

9.1 选择主机

云主机有多种类型，但是对于开发人员来说，可选的方案主要在平台即服务（Platform as a Service，PaaS）和基础设施即服务（Infrastructure as a Service，IaaS）之间。

如果使用后者的话，你通常得到的就是可管理的裸机，在上面可以安装应用所需的各种服务。

如果你撇开像 Docker（非常惊艳的技术，你绝对应该尝试一下）这样的技术，这非常类似于典型的主机，你的运维团队需要搭建和维护应用运行所需的环境。

而另一方面，PaaS 使得应用的部署非常简单，就像使用了推送即部署（push-to-deploy）工作流进行开发一样。

在这个领域，最知名的提供商为：

- Pivotal 支持的 Cloud Foundry；
- Red Hat 的 OpenShift；
- Salesforce 在 2010 年收购的 Heroku。

这 3 个提供商各有其优缺点，我会其进行概要地介绍。

9.1.1 Cloud Foundry

由 Pivotal 公司提供支持，这也是 Spring 背后的公司，Pivotal Web Service 运行在 Cloud

Foundry 之上，这是由一个基金会维护的开源 PaaS，并且提供了一个很有意思的包格式。

他们提供了 60 天的免费试用，其定价策略基于实例所分配的内存以及实例的数量。

它的价格区间是最小的实例（128Mb）每月 2.70 美元，2 GB 的实例每月 43.20 美元。

如果你想尝试一下的话，免费试用并不需要信用卡。我们能够通过一个市场来非常简便地安装服务，如 Redis 或 Postgre SQL，其免费版本的可选项比较有限。他们有一个非常好的命令行工具，可以通过控制台来管理应用。你可以使用构建打包（buildpacks）或直接推送 JAR 文件的方式来进行部署。

构建打包会猜测你所使用的技术栈并按照最标准的方式来进行构建（如 Maven 的 mvn package 或 Gradle 的“./gradlew stage”等）。

参考如下 URL 的操作指南来了解如何将应用部署到 Cloud Foundry 上：http://docs.cloudfoundry.org/buildpacks/java/gsg-spring.html。

9.1.2 OpenShift

OpenShift 由 Red Hat 来进行维护的，由 OpenShift Origin 项目来提供支持，这是一个开源的设施，它在 Google 的 Kubernetes 之上运行 Docker 容器。

它的价位合理并提供了很高的自由度，因为它既是PaaS又是IaaS。它的定价策略基于Gear[①]、运行应用的容器或像Jenkins、数据库这样的服务。

OpenShift 有一个免费计划，能够提供 3 个小的 Gear。你的应用每月必须要空闲 24 小时，否则的话，就要输入账单信息。

更多或更大的 Gear 需要进行付费，最小的大约是每月 15 美元，而最大的是每月 72 美元。

要将 Spring Boot 应用部署到 OpenShift 上，你需要自己动手。这会比其他基于构建打包的 PaaS 稍微麻烦一点，但是也非常易于配置。

请参考在 OpenShift 中使用 Spring Boot 的博客文章，地址是 http://blog.codeleak.pl/2015/02/openshift-diy-build-spring-boot.html。

[①] 用于存放代码的安全容器——译者注

9.1.3 Heroku

Heroku 是一个知名的 PaaS，它具有丰富的文档，并且提供了以代码为中心的构建打包。它能够连接很多的服务，称之为 add-on，但使用它们的话需要你提供支付信息。

对于免费项目来说，它是非常有意思的，并且能够快速上手。其不足之处在于，如果你想进行扩展的话，它每月至少会直接消费 25 美元。如果超过 30 分钟没有活动的话，免费的实例将会进入睡眠状态，这意味着免费的 Heroku 应用会耗费 30 秒的时间来进行加载。

Heroku 有一个很棒的管理 dashboard 和命令行的工具。在本章中，我会选择 Heroku，因为它非常简单易用。这里介绍的理念适用于大多数的 PaaS。

在本章中，只要你不使用 Redis add-on，就可以跟着执行大多数的功能并且无需提供信用卡信息也能进行应用部署。如果你选择免费计划的话，是不会被计费的。

9.2 将 Web 应用部署到 Pivotal Web Services 中

如果你想要将应用部署到 Pivotal Web Services（PWS），那么就可以遵循本章节中的步骤。

9.2.1 安装 Cloud Foundry CLI 工具

要创建 Cloud Foundry 应用，我们需要做的第一件事就是创建 PWS 账号。参考该地址 http://docs.run.pivotal.io/starting/中的文档。你需要创建一个组织，每个新建的组织都会有一个默认的空间（development），如 图 9-1 所示。

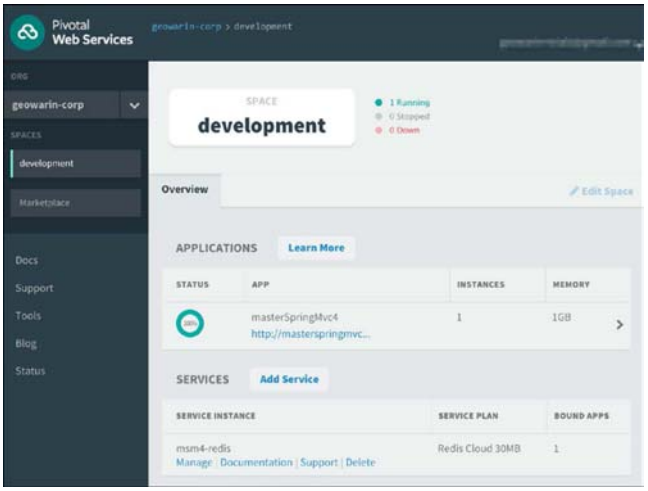

图 9-1

在左侧的导航栏中，你可以看到一个“Tools”的超链接，通过它可以下载 CLI，它也可以通过开发人员控制台来下载。为你的操作系统选择合适的包，如图 9-2 所示。

图 9-2

9.2.2　装配应用

我们的应用只需进行一下组装就可以部署了。

使用 PWS 的好处在于不需要推送源码来进行部署。我们可以生成 JAR、推送，然后所有的事情都会自动探测到。

我们可以执行如下的命令进行打包，从而得到部署所需的 JAR 文件：

```
./gradlew assemble
```

这会在 build/libs 目录中创建一个 JAR 文件。此时，可以执行如下的命令，下面的命令会将待部署的内容发送到 PWS（run.pivotal.io）的空间中：

```
$ cf login -a api.run.pivotal.io -u <account email> -p <password> -o
<organization> -s development

API endpoint: api.run.pivotal.io
Authenticating...
OK

Targeted org <account org>

Targeted space development


API endpoint: https://api.run.pivotal.io (API version: 2.33.0)
User:         <account email>
Org:          <account organization>
Space:        <account space>
```

登录成功之后，就可以通过如下的命令来推送 JAR 文件，你需要带有一个可用的应用名称：

```
$ cf push your-app-name -p build/libs/masterSpringMvc-0.0.1-SNAPSHOT.jar

Creating app msmvc4 in org Northwest / space development as wlund@
pivotal.io...
OK
Creating route msmvc4.cfapps.io...
OK
Binding msmvc4.cfapps.io to msmvc4...
OK
Uploading msmvc4...
Uploading app files from: build/libs/masterSpringMvc-0.0.1-SNAPSHOT.jar
Uploading 690.8K, 108 files
Done uploading
OK
Starting app msmvc4 in org <Organization> / space development as <account
email>
-----> Downloaded app package (15M)
-----> Java Buildpack Version: v3.1 | https://github.com/cloudfoundry/
java-buildpack.git#7a538fb
-----> Downloading Open Jdk JRE 1.8.0_51 from https://download.run.
pivotal.io/openjdk/trusty/x86_64/openjdk-1.8.0_51.tar.gz (1.5s)
       Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.4s)
-----> Downloading Open JDK Like Memory Calculator 1.1.1_RELEASE from
https://download.run.pivotal.io/memory-calculator/trusty/x86_64/memoryca
lculator-1.1.1_RELEASE (0.1s)
       Memory Settings: -Xmx768M -Xms768M -XX:MaxMetaspaceSize=104857K
-XX:MetaspaceSize=104857K -Xss1M
-----> Downloading Spring Auto Reconfiguration 1.7.0_RELEASE
from https://download.run.pivotal.io/auto-reconfiguration/autoreconfiguration-
1.7.0_RELEASE.jar (0.0s)
-----> Uploading droplet (59M)
0 of 1 instances running, 1 starting
1 of 1 instances running

App started
OK
App msmvc4 was started using this command `CALCULATED_MEMORY=$($PWD/.
java-buildpack/open_jdk_jre/bin/java-buildpack-memory-calculator-1.1.1_
RELEASE -memorySizes=metaspace:64m.. -memoryWeights=heap:75,metaspace:10
,stack:5,native:10 -totMemory=$MEMORY_LIMIT) && SERVER_PORT=$PORT $PWD/.
java-buildpack/open_jdk_jre/bin/java -cp $PWD/.:$PWD/.java-buildpack/
spring_auto_reconfiguration/spring_auto_reconfiguration-1.7.0_RELEASE.
jar -Djava.io.tmpdir=$TMPDIR -XX:OnOutOfMemoryError=$PWD/.java-buildpack/
open_jdk_jre/bin/killjava.sh $CALCULATED_MEMORY org.springframework.boot.
loader.JarLauncher`
```

```
Showing health and status for app msmvc4 in org <Organization> / space
development as <Account Email>
OK

requested state: started
instances: 1/1
usage: 1G x 1 instances
urls: msmvc4.cfapps.io
last uploaded: Tue Jul 28 22:04:08 UTC 2015
stack: cflinuxfs2
buildpack: java-buildpack=v3.1-https://github.com/cloudfoundry/javabuildpack.
git#7a538fb java-main open-jdk-like-jre=1.8.0_51 open-jdk-likememory-
calculator=1.1.1_RELEASE spring-auto-reconfiguration=1.7.0_RELEASE

     state     since                    cpu    memory          disk
details
#0   running   2015-07-28 03:05:04 PM   0.0%  450.9M of 1G   137M of 1G
```

在这里，平台会为你做很多的事情。它会提供一个容器，并探测需要哪种构建包，在我们的场景下，所需的就是 Java。

然后它会安装所需的 JDK 并下载我们所指向的应用，它会为应用创建一个路由，并报告给我们，最后，它会启动应用。

现在，我们可以在开发人员控制台中看到该应用，如图 9-3 所示。

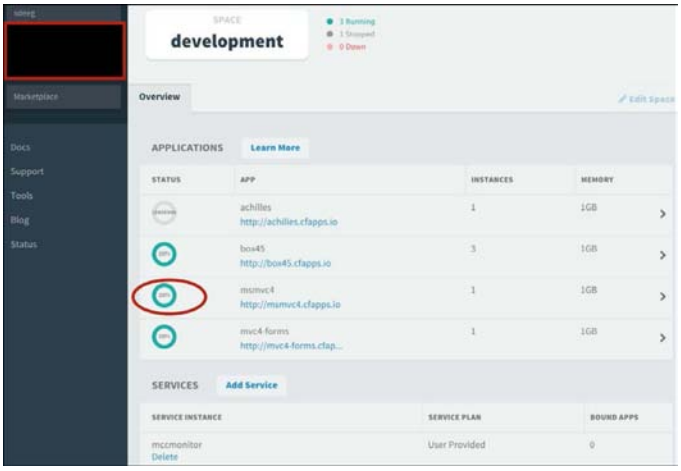

图 9-3

通过使用图 9-3 里面选中的路由，我们就可以访问应用了，导航至 http://msmvc4.cfapps.io，

可以看到如图 9-4 所示。

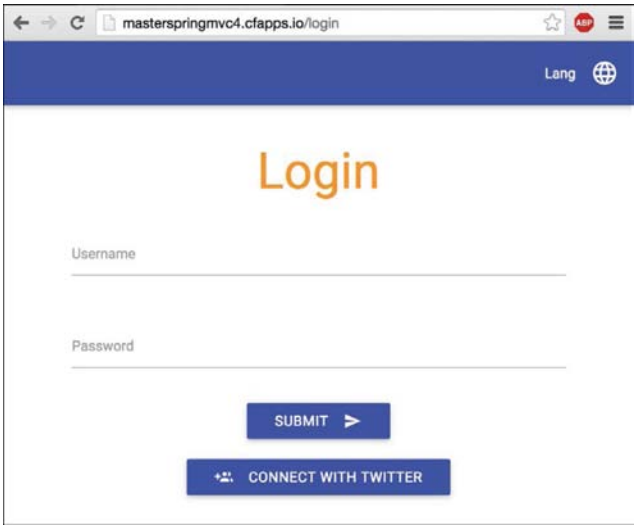

图 9-4

太棒了！

唯一无法正常运行的功能就是文件上传，不过，我们稍后会解决这个问题。

9.2.3 激活 Redis

在应用服务方面，我们可以有很多的服务可供选择。其中有一个就是 Redis Cloud，它有一个免费计划，带有 30MB 的存储空间，你可以选择使用这个计划。

通过这种方式，为其选择一个你喜欢的名字然后将其绑定到应用中。默认情况下，Cloud Foundry 将会注入一些服务相关的属性到环境当中：

- `cloud.services.redis.connection.host`
- `cloud.services.redis.connection.port`
- `cloud.services.redis.connection.password`
- `cloud.services.redis.connection.uri`

这些属性会遵循相同的命名约定，所以如果你添加了多项服务的话，可以很容易地对其进行跟踪。

默认情况下，Cloud Foundry 会启动 Spring 应用并激活 Cloud profile。

我们可以利用这一点，在 src/main/resources 中创建名为 application-cloud.properties 的属性文件，当我们的应用在 PWS 中运行的时候，就会使用这个文件：

```
spring.profiles.active=prod,redis

spring.redis.host=${cloud.services.redis.connection.host}
spring.redis.port=${cloud.services.redis.connection.port}
spring.redis.password=${cloud.services.redis.connection.password}

upload.pictures.uploadPath=file:/tmp
```

这会将 Redis 实例与应用进行绑定并激活另外两个 profile：prod 和 redis。

我们还修改了上传文件的路径。注意，在云中使用文件系统要遵循不同的规则，参考如下的链接以了解更多信息：http://docs.run.pivotal.io/devguide/deploy-apps/prepare-to-deploy.html#filesystem。

最后一件需要做的事情就是禁用 Spring Session 的一项特性，在我们的主机实例上是无法使用该特性的：

```
@Bean
@Profile({"cloud", "heroku"})
public static ConfigureRedisAction configureRedisAction() {
    return ConfigureRedisAction.NO_OP;
}
```

更多信息请参考：http://docs.spring.io/spring-session/docs/current/reference/html5/#api-redisoperationssessionrepository-sessiondestroyedevent。

这个配置也会用到 Heroku 中。

这样就可以了！你可以重新组装应用并进行推送，这样的话，会话和应用缓存就能存到 Redis 中了。

你可能想要看一下它的市场，以便了解其他可用的特性，如数据绑定或消息服务、应用扩展以及对应用健康状况的管理，但这已经超出本书的范围了。

希望你能享受平台带来的生产率提升和其中的乐趣！

9.3　将 Web 应用部署到 Heroku 中

在本节中，我们会将应用免费部署到 Heroku 中。我们甚至还会使用免费的 Redis 实例，

存储会话和缓存信息。

9.3.1 安装工具

要创建 Heroku 应用，我们要做的第一件事情就是下载命令行工具，可以从该地址 https://toolbelt.heroku.com 获取。

在 Mac 中，还可以使用 brew 命令进行安装：

```
> brew install heroku-toolbelt
```

在 Heroku 创建账号，并使用 heroku login 命令将账号与 toolbelt 进行关联：

```
> heroku login
Enter your Heroku credentials.
Email: geowarin@mail.com
Password (typing will be hidden):
Authentication successful.
```

然后，切换到应用的根目录下并输入 heroku create appName --region eu，将 appName 换成你所选择的应用名称。如果你不指定应用名称的话，它将会自动生成：

```
> heroku create appname --region eu
Creating appname... done, region is eu
https://appname.herokuapp.com/ | https://git.heroku.com/appname.git
Git remote heroku added
```

如果你已经通过 UI 创建了应用，那么到应用的根目录下并通过如下的命令创建 Git remote：heroku git:remote -a yourapp。

这些命令所做的就是为 Git repository 添加一个名为 heroku 的 Git remote。在 Heroku 上部署的过程只是将我们的一个分支推送到 Heroku 上。remote 上安装的 Git hook 将会处理好剩余的事情。

如果你输入 `git remote -v` 命令的话，就能看到 heroku 版本：

```
> git remote -v
heroku    https://git.heroku.com/appname.git (fetch)
heroku    https://git.heroku.com/appname.git (push)
origin    https://github.com/Mastering-Spring-MVC-4/mastering-springmvc4-
code.git (fetch)
origin    https://github.com/Mastering-Spring-MVC-4/mastering-springmvc4-
code.git (push)
```

9.3.2　搭建应用

要在 Heroku 上运行 Gradle 应用，我们需要两个组成部分：构建文件中名为 stage 的任务以及包含运行应用所需命令的一个小文件，名为 ProcFile。

1．Gradle

Gradle 构建打包会自动在应用的根目录下尝试运行“./gradlew stage”命令。

你可以通过该链接，获取 Gradle 构建打包的更多信息：https://github.com/heroku/heroku-buildpack-gradle。

我们还没有名为“stage”的任务，添加如下的代码到 build.gradle 文件中：

```
task stage(type: Copy, dependsOn: [clean, build]) {
    from jar.archivePath
    into project.rootDir
    rename {
        'app.jar'
    }
}
stage.mustRunAfter(clean)

clean << {
    project.file('app.jar').delete()
}
```

这将会定义名为 stage 的任务，它将会复制 Spring Boot 在应用的根目录下所生成的 JAR 包并将其命名为 `app.jar`。

按照这种方式，这个 JAR 会很容易找到。`stage` 任务依赖于 `clean` 和 `build` 这两项任务，这就意味着在 stage 任务启动之前会首先执行这两项任务。

默认情况下，Gradle 会尝试优化任务的依赖图，所以我们必须提供一个提示，强制 `clean` 任务在 `stage` 之前执行。

最后，我们为已有的 clean 任务添加一条新的指令，用来删除生成的 `app.jar`。

现在，如果你运行“`./gradlew stage`”的话，它应该会运行测试并将生成的打包应用放在项目的根目录下。

2．Procfile

当 Heroku 探测到 Gradle 应用的时候，它会自动运行一个带有 Java 8 的容器。所以，还需要做一点配置。

我们需要一个文件，其中包含了运行应用所需的 shell 命令，在应用的根目录下创建名为 Procfile 的文件：

```
web: java -Dserver.port=$PORT -Dspring.profiles.active=heroku,prod
-jar app.jar
```

在这里有几点需要注意的事情。首先，我们将应用声明为 Web 应用，同时使用环境变量重新定义了应用运行所需的端口。这一点非常重要，因为你的应用将会与其他的应用共存，所以每个只能分配一个端口。

然后，你会发现我们的应用会基于两个 profile 来运行。第一个是用来优化性能的 prod profile，这个我们在前面的章节已经创建过了，另外还有一个新的 heroku profile，稍后会创建它。

9.3.3 Heroku profile

我们不希望将敏感信息，如 Twitter 的 app key，放到版本控制中。所以，我们需要创建一些属性，并从应用的环境中读取这些值：

```
spring.social.twitter.appId=${twitterAppId}
spring.social.twitter.appSecret=${twitterAppSecret}
```

为了使其正常运行，我们需要在 Heroku 上配置两个环境变量。通过 toolbelt，可以这样实现：

```
> heroku config:set twitterAppId=appId
```

另外一种方案就是到 dashboard 的页面，在 settings 标签页配置环境变量，如图 9-5 所示。

图 9-5

参考 https://devcenter.heroku.com/articles/config-vars 了解这方面的更多信息。

9.3.4　运行应用

现在，我们该在 Heroku 上运行应用了！

如果你还没有提交的话，将所有的变更提交到 master 分支上。现在，通过使用 git push heroku master 命令将你的 master 分支推送到 heroku remote。这样就会从头下载所有依赖并构建应用，所以会耗费一点时间：

```
> git push heroku master
Counting objects: 1176, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (513/513), done.
Writing objects: 100% (1176/1176), 645.63 KiB | 0 bytes/s, done.
Total 1176 (delta 485), reused 1176 (delta 485)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Gradle app detected
remote: -----> Installing OpenJDK 1.8... done
remote: -----> Building Gradle app...
remote:        WARNING: The Gradle buildpack is currently in Beta.
remote: -----> executing ./gradlew stage
remote:        Downloading https://services.gradle.org/distributions/
gradle-2.3-all.zip

...

remote:        :check
remote:        :build
remote:        :stage
remote:
remote:        BUILD SUCCESSFUL
remote:
remote:        Total time: 2 mins 36.215 secs
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing... done, 130.1MB
```

```
remote: -----> Launching... done, v4
remote:        https://appname.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy.... done.
To https://git.heroku.com/appname.git
* [new branch]  master -> master
```

应用构建完成之后，将会自动运行，你可以输入 `heroku logs` 查看最新的日志，也可以使用 `heroku logs –t` 来跟踪实时日志。

这时，能够在控制台上看到应用正在运行，如果所有的步骤都按预期进行的话，就能访问 http://yourapp.herokuapp.com 了，如图 9-6 所示。

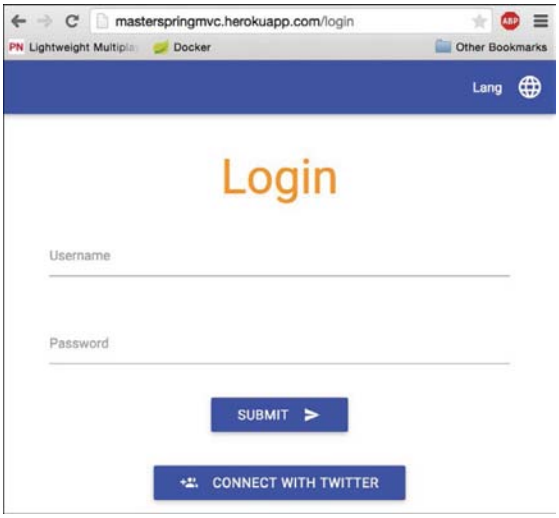

图 9-6

我们可以实时访问了！现在，可以将它分享给你的朋友了！

9.3.5 激活 Redis

要在应用中激活 Redis，我们有多个可选的方案，HerokuRedis add-on 目前是 beta 版本。完全免费的服务涵盖了 20MB 的存储以及分析和日志功能。

访问 https://elements.heroku.com/addons/heroku-redis 来了解更多细节。

在这个阶段，我们需要提供信用卡信息。

为了给应用安装 Redis add-on，输入下面的命令：

```
heroku addons:create heroku-redis:test
```

现在，我们已经激活了这个 add-on，当应用在 Heroku 运行的时候，会有一个名为 REDIS_URL 的环境变量可供使用。

你可以使用 heroku config 命令来检查这个变量是否已经定义：

```
> heroku config
=== masterspringmvc Config Vars
JAVA_OPTS:        -Xmx384m -Xss512k -XX:+UseCompressedOops
REDIS_URL:        redis://x:xxx@ec2-xxx-xx-xxx-xxx.eu-west-1.compute.
amazonaws.com:6439
```

因为 RedisConnectionFactory 类不能使用 URL，所以我们需要进行一些调整：

```
@Configuration
@Profile("redis")
@EnableRedisHttpSession
public class RedisConfig {

    @Bean
    @Profile("heroku")
    public RedisConnectionFactory redisConnectionFactory() throws
URISyntaxException {
        JedisConnectionFactory redis = new JedisConnectionFactory();

        String redisUrl = System.getenv("REDIS_URL");
        URI redisUri = new URI(redisUrl);
        redis.setHostName(redisUri.getHost());
        redis.setPort(redisUri.getPort());
        redis.setPassword(redisUri.getUserInfo().split(":", 2)[1]);

        return redis;
    }

    @Bean
        @Profile({"cloud", "heroku"})
    public static ConfigureRedisAction configureRedisAction() {
        return ConfigureRedisAction.NO_OP;
    }
}
```

现在，在 RedisConfig 类中有了两个 Heroku 特有的 bean。只有 redis 和 heroku 同时激活的时候，这两个 bean 才会生效。

注意，我们还禁用了一些 Spring Session 的配置。

Spring Session 通常会通过 Redis 的 Pub/Sub 接口监听已销毁会话 key 相关的事件。

在启动的时候，它会自动尝试配置 Redis 环境来激活监听器。在我们这样的安全环境下，除非具有管理员访问权限，否则是禁止添加监听器的。

这些监听器对我们的场景也不是非常重要，所以可以安全地禁用这种行为。关于它的更多信息，请参考 http://docs.spring.io/spring-session/docs/current/reference/html5/#api-redisoperationssessionrepository-sessiondestroyedevent。

我们需要修改 Procfile 文件，这样 Heroku 会在 redis profile 下运行应用：

```
web: java -Dserver.port=$PORT -Dspring.profiles.
active=heroku,redis,prod -jar app.jar
```

现在，你可以提交变更并将代码推送至 Heroku。

9.4 改善应用的功能

现在，我们已经有了一个很不错的应用，并将其部署上线了，但是它还没有太大的用途和新颖性，除非我们想办法改造它。

你可以尝试让它变得更好，更符合你的个人偏好。在为自己的成就感到自豪的时候，你就可以将应用的 URL 以#masterspringmvc 作为标签发布到 Tweet 上。

为了让应用做到最好，我们还有一些遗留的工作。如下是一些改善的想法：

- 删除用户的旧图片，避免保存没有用处的图片；
- 使用 Twitter 的认证信息来填充用户基本信息；
- 与用户的账号进行交互；
- 提供 WebSocket 通道，查看应用中实时的搜索结果。

我们可以尽可能发挥自己的想象力！

我的版本的应用已经部署在 http://masterspringmvc.herokuapp.com 上。我会对应用进行改善，使其更具有交互性，你可以尝试发现其中的差异！

9.5　小结

将应用部署在云提供商中是非常简单直接的，这归因于 Spring Boot 所生成的可运行的 JAR 文件。如今，云部署是非常划算的，部署 Java 应用也变得非常容易。

通过使用 Redis 来支撑会话功能，我们实现了基本的可扩展性应用。我们可以轻松地在负载均衡器后面增加服务器，从而应对高访问量的需求。

唯一没有实现可扩展性的是 WebSocket 功能，这需要额外的工作使其运行在消息代理上，例如 RabbitMQ。

我还清楚地记得，在以前寻找主机运行 Tomcat 是很罕见和昂贵的。那些日子已经一去不复返了，未来属于 Web 开发人员，所以采取行动让这一切变成现实吧！

在下一章中，我们将会了解到如何让应用变得更好，讨论目前为止还没有提到的技术，了解 Spring 整体的生态系统以及现代 Web 应用所面临的挑战。

第 10 章
超越 Spring Web

在本章中，我们将会看到已经完成了哪些功能、解决了哪些问题以及还有什么问题需要解决。

我们会从整体上介绍 Spring 生态系统以及持久化、部署以及单页面应用（Single Page Application）的话题。

10.1 Spring 生态系统

从 Web 到数据，Spring 是一个综合的生态系统，致力于以模块化的方式解决各种问题，如图 10-1 所示。

图 10-1

参考如下地址 https://spring.io/platform 了解 Spring IO 平台。

10.1.1　核心

Spring 框架的核心显然是依赖注入机制。

我们只接触到了安全特性的皮毛，学习了该框架与 Groovy 的良好集成。

10.1.2　执行

我们详细了解了 Spring Boot 是什么——它能够简化和内聚众多的子项目。

它能够让我们关注真正重要的事情，也就是你的业务代码。

Spring XD 项目也非常有意思，它的目标是提供工具来处理、分析以及转换或导出数据，它的关注点是大数据。关于它的更多信息，参考 http://projects.spring.io/spring-xd/。

10.1.3　数据

在开发应用的过程中，我们没有涉及到的一个事情就是如何将数据存储到数据库中。在 Pivotal 的参考架构中，有一层是专门用于关系型数据和非关系型（NoSQL）数据的。

在 spring-data 下，Spring 生态系统提供了很多有趣的解决方案，参见 http://projects.spring.io/spring-data/。

在构建缓存的时候，我们简单了解了 Spring Data Redis，但是 Spring Data 所涉及的内容要比这多得多。

所有的 Spring Data 具有相同的基本理念，例如模板 API，这是从持久化系统中检索和存储对象的一个抽象概念。

Spring Data JPA（http://projects.spring.io/spring-data-jpa/）和 Spring Data Mongo（http://projects.spring.io/spring-data-mongodb/）是 Spring Data 中最为知名的项目。通过它们，我们能够使用 repository 来操作实体，这是一些简单的接口，提供了创建查询、持久化对象等功能。

Petri Kainulainen（http://www.petrikainulainen.net/spring-data-jpa-tutorial/）有很多关于 Spring Data 的完整样例，它们没有使用 Spring Boot 所提供的基础设施，但是通过指南，你应该能够快速上手，其中一个样例就是 https://spring.io/guides/gs/accessing-data-jpa/。

Spring Data REST 也是一个很有意思的项目，它能够半自动化地将你的实体暴露为 RESTful API，参考 https://spring.io/guides/gs/accessing-data-rest/来获取更详细的使用指导。

10.1.4 其他值得关注的项目

Spring Integration（http://projects.spring.io/spring-integration）和 Spring Reactor（http://projectreactor.io）是我非常喜欢的两个 Spring 项目。

Spring Reactor 是由 Pivotal 实现的反应式流（reactive stream），它的理念是在服务器端提供完全非阻塞 IO。

Spring Integration 则关注于企业级集成模式（Enterprise Integration Pattern），能够让我们设计通道以加载和转换来自异构系统的数据。

通道所能实现的功能可以参考该地址的一个不错的简单样例：http://lmivan.github.io/contest/#_spring_boot_application。

如果你的应用需要与异构和复杂的子系统进行交互，那么它绝对值得了解一下。

在 Spring 生态系统中，另外一个我们没提到的项目就是 Spring Batch，对于企业级系统中日常操作的大量数据来说，这是对这种操作的一个很有用的抽象。

10.2 部署

Spring Boot 能够以 JAR 文件的形式运行和分布式部署 Spring 应用，在这方面，它是很成功的。

毫无疑问，这是正确的方向，但有时候我们需要部署的并不仅仅是 Web 应用。

在处理与多个服务器和数据源交互的复杂系统时，运维团队的工作可能会非常令人头疼。

Docker

现在，还有人没听说过 Docker 吗？在容器领域这是一个新的产品，它已经非常成功了，这样归功于它活跃的社区。

Docker 背后的理念并不新鲜，它使用了 LinuX Containers （LXC）以及 cgroups 来提供完全隔离的环境，让应用程序在这个隔离的环境中运行。

在 Spring 的 Web 站点上，你可以看到一个教程，它能够指导你从头开始学习 Docker：https://spring.io/guides/gs/spring-boot-docker。

Pivotal Cloud Foundry 已经使用容器技术多年了，他们的容器管理器被称为 Warden。最近他们迁移到了 Garden，这是一个功能抽象，不仅支持 Linux 容器，也支持 Windows 容器。

Garden 是最近发布的 Cloud Foundry（称为 Diego）的一部分，它允许将 Docker 镜像作为部署单元。

开发版本的 Cloud Foundry 也以 Lattice 的名字进行了发布，相关的信息可以参考 https://spring.io/blog/2015/04/06/lattice-and-spring-cloud-resilient-sub-structure-for-your-cloud-native-spring-applications。

如果你想测试容器，而不想被命令行困扰的话，那么你可以看一下 Kitematic。通过它，不用在系统中安装二进制文件，你就运行 Jenkins 容器或 MongoDB。访问 https://kitematic.com/ 了解 Kitematic 的更多信息。

在 Docker 生态系统中，另外值得一提的工具就是 Docker Compose。它允许你通过一个配置文件运行和连接多个容器。

参考 http://java.dzone.com/articles/spring-session-demonstration 上的样例，这个 Spring Boot 应用包含了两个 Web 服务器、存储用户会话的一个 Redis 还有一个进行负载均衡的 Nginx 容器。当然，Docker Swarm 也有很多值得学习的东西，它允许我们通过一个简单的命令就能扩展应用，还有 Docker Machine，它能够在任何机器上为我们创建 Docker 主机，包括在云提供商上面。

Google Kurbernetes 和 Apache Mesos 也是分布式系统中很好的样例，它们从 Docker 容器技术上受益匪浅。

10.3　单页面应用

如今大多数的 Web 应用都是使用 JavaScript 编写的。Java 则负责后端，在处理数据和业务规则中依然扮演着重要的角色。但是，大多数的 GUI 内容都是在客户端完成的。

在响应性和用户体验方面来说，这样做都是有道理的，但是这样的应用会带来额外的复杂性。

开发人员需要同时精通 Java 和 JavaScript 以及很多的框架，这可能会有些挑战。

10.3.1　参与者

如果你想深入学习 JavaScript 的话，我强烈推荐 Dave Syer 的 Spring 和 AngularJS 教程，

地址是 https://spring.io/guides/tutorials/spring-security-and-angular-js。

选择 JavaScript MVC 框架可能也是件很困难的事情。在 Java 社区，AngularJS 多年来一直很受欢迎，但是最近人们开始转向其他的方案，更多信息，请参考 https://gist.github.com/tdd/5ba48ba5a2a179f2d0fa。

其他的可选方案包括如下几种。

- **BackboneJS**：这是一个非常简单的 MVC 框架，它构建在 Underscore 和 jQuery 之上。
- **Ember**：这是一个综合性的系统，它提供了与数据和其他内容进行交互的很多基础设施。
- **React**：这是由 Facebook 推出的最新项目。在处理视图方面，它有很新很有意思的哲学，它的学习曲线很陡峭，但是在设计 GUI 框架方面，它确实是一个非常有意思的系统。

目前，React 是我最喜欢的项目，它能够让我们关注于视图，它的单向数据流功能使得处理应用的状态变得非常容易。但是，它的版本依然是 0.13，这意味着它会非常有趣，因为活跃的社区总会有新的方案和想法，这也会有些令人不安，尽管它已经开源超过两年了，但是前面依然还有很长的路要走。参考 https://facebook.github.io/react/blog/2014/03/28/the-road-to-1.0.html，了解其“1.0 之路”的相关信息。

10.3.2 未来的前景

我看到过很多 Java 开发人员为 JavaScript 的随意性而抓狂。它不是强类型的语言，这一点也让很多开发人员很痛苦。

其实还有其他的替代方案，如 **Typescript**（http://www.typescriptlang.org/），它非常有意思并且提供了 Java 开发人员经常用到的特性，这样能够让我们接受起来更简单一些，如接口、类、IDE 的支持以及自动补全功能。

很多人寄希望于下一个版本（2.0）的 Angular，但是它会破坏已有的功能，这一点引起了很多的争议。我认为这是出于好意，他们与 Microsoft 团队的协作，将会使得 Typescript 更加重要。

当听到 ECMAScript 最大的一个新特性的时候，很多的 JEE 开发人员可能会露出会心的微笑，因为这个新框架允许在开发中使用装饰器（decorator），这是一种类似注解的机制。

要理解注解与装饰器的差异，请参考 http://blog.thoughtram.io/angular/2015/05/03/the-difference-between-annotations-and-decorators.html。

JavaScript 在快速演化，ECMAScript 6 有很多有意思的特性，使其成为了一门很高级和复杂的语言。不要坐失良机，参考 https://github.com/lukehoban/es6features 了解一下它的新特性，要不然就太迟了。

Web 组件规范也会是游戏规则的改变者，它的目标是提供可重用的 UI 组件，React 团队和 Angular 2 团队均计划与其进行结合。Google 在 Web 组件之上开发了一个很有意思的项目，名为 Polymer，目前已经是 1.0 版本了。

参考 http://ng-learn.org/2014/12/Polymer/了解这些项目的最新状态。

10.3.3　实现无状态

当处理 JavaScript 客户端的时候，依赖于会话 cookie 并不是最佳方案。很多的应用选择使用完全无状态的方案，通过一个 token 来识别客户端。

如果你想继续使用 Spring Session 的话，可以看一下 HeaderHttpSessionStrategy 类。它实现了通过 HTTP 头信息发送和获取会话。这么做的样例可以参考 https://drissamri.be/blog/2015/05/21/spring-security-and-spring-session/。

10.4　小结

Spring 生态系统是非常庞大的，为现代的 Web 开发人员提供了很多的内容。

我们能够遇到的所有问题，Spring 几乎都有相应的项目来解决。

该说再见了！我希望你能享受这段 Spring MVC 的简短旅程，希望它能够让你在工作或业余时间的开发更有乐趣，帮助你创建出酷炫的项目。

精通Spring MVC 4

本书带领我们展开一次有意思的旅行，从开发自己的Web应用开始，到将其部署到云中。首先，我们会使用Spring Tool Suite和Spring Boot生成自己的Spring项目。在开发高级的交互应用时，涉及处理文件上传和复杂的URL，此时我们会深入研究Spring MVC的内部运行原理以及现代Web架构的理念。随后，我们将会测试、保护和优化Spring Web应用，并且还会设计可由前端访问的RESTful服务。最后，所有的事情都已准备就绪，我们会将应用部署到云提供商的服务上，邀请所有的人来访问它 。

本书适合已经熟悉Spring编程基础知识并迫切希望扩展其Web技能的开发人员阅读。

通过本书，你将学会：

- 使用Spring Boot和Spring Tool Suite搭建自己的Web应用；
- 探索Spring MVC的架构，了解在视图间实现导航的不同工具；
- 设计复杂的高级表单并对模型进行校验；
- 创建RESTful应用，实现有意义的API，其中会带有相关的错误信息；
- 创建可维护的单元测试和验收测试；
- 保护应用，同时支持可扩展；
- 通过缓存、ETags和异步响应来优化请求；
- 将应用部署到云中。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

ISBN 978-7-115-44758-6

分类建议：计算机／软件开发／Java
人民邮电出版社网址：www.ptpress.com.cn