第 1 章

Spring Boot 基础知识

距离 Java 的第一个版本已有 25 个年头，这门语言的生态在这漫长岁月里变得愈发丰富多样。这个 Java 生态中有一个不容忽视的名字，那便是 Spring。Spring 技术极大地提高了开发人员的开发效率，将人们从刀耕火种一下带到了工业时代。本书将介绍 Spring 这个开发利器中最锐利的锋芒——Spring Boot。

本章主要涉及的知识点有：

- Spring 与 Spring Boot 的基本概念
- Spring Boot 开发环境的搭建
- 构建 Spring Boot 项目的基本步骤

1.1 Spring 与 Spring Boot

当学习一门技术时，先对其有一个大概的认识是非常必要的，这样对学习方向的把控很有帮助。本节先来了解一下 Spring 与 Spring Boot 的基本概念，看看它们在开发过程将扮演什么角色，发挥什么作用。

1.1.1 当我们谈论 Spring 时会谈论些什么

在不同的语境中 Spring 蕴含不同的含义。狭义的解释为 Spring 指 Spring Framework，因为这是生态的核心，Spring 起源于此。但随着时间推移，社区基于 Spring Framework 构建了更多其他的项目，这样一来，当人们说到“Spring”时，往往指的是整个 Spring 生态。

Spring 的架构如图 1.1 所示，其核心 Core Container 是一个 IoC（Inversion of Control）容器。

IoC 即控制反转，是一种面向对象的思想，作用在于将对象之间的依赖关系交由框架进行统一管理。具体的实现方式是 DI（Dependency Injection，依赖注入）。简单来说，就是开发人员通过 XML 配置或 JavaConfig 的方式将依赖关系告知容器。容器在“恰当”的时机去创建对象，而不需要开发人员过多的关注。

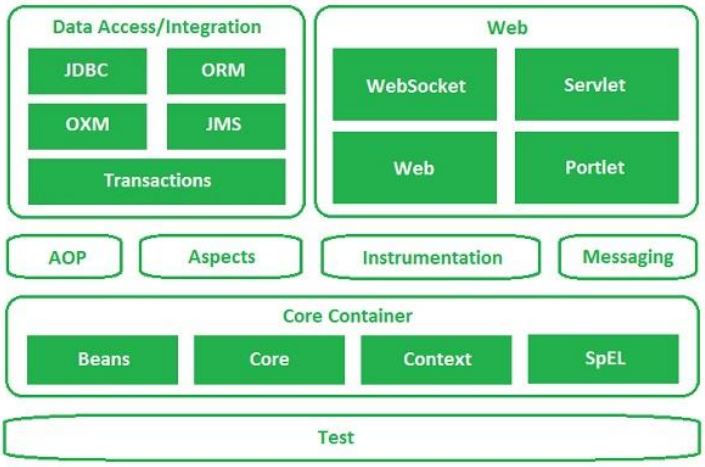

图 1.1　Spring 架构图

Web 模块，指 Web 应用基础功能的集合。其中包含对文件上传的支持、使用 Servlet 监听器初始化 IoC 容器、Web 应用上下文等内容。另外还有对基于 Servlet 开发的支持，这块在 Spring 的体系中又被称作 Spring MVC。在第 2 章将着手构建一个 Web 应用，就离不开 Spring MVC 的支持。

Data Access/Integration，即数据访问与集成方案。JDBC、ORM、OXM 等对于数据库操作的方案被包含其中。在这些模块当中，ORM 将会在之后的章节着重介绍。相较于 JDBC 这样基础的数据库访问方案，使用 ORM 开发起来更为高效。ORM 是对 JDBC 的封装，将字段高效地与对象进行映射，将对数据库的操作转换为对对象的操作。我们将在第 4 章开始学习如何利用这些工具访问数据库。

AOP（Aspect-Oriented Programming，面向切面编程）是通过预编译方式和运行期间动态代理实现程序功能统一维护的一种技术，是 OOP 的延续，也是 Spring Framework 中的一个重要内容，是函数式编程的一种衍生范型。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

Test 模块提供了 Spring 应用使用 JUnit 和 TestNG 进行单元测试和集成测试的支持。在测试过程中能轻松读取到应用上下文，并且它具有可用于隔离测试代码的 Mock 对象。

1.1.2　什么是 Spring Boot

Spring Boot 是在 Spring 的基础上构建起来的一个项目。它基于“约定优于配置”（Convention Over Configuration）的理念，解决了基于 Spring 开发需要繁复配置的痛点。使用 Spring Boot 进行开发可以巧妙地选择项目所需的依赖项，对依赖中涉及的功能进行自动配置，并且能在不依赖 Web 容器的情况下一键启动，大大简化了应用的开发和部署过程。

以下是 Spring Boot 提供的高级功能：

- 自动配置：根据“starter”依赖项进行自动配置。
- 独立：无需将程序部署到另外的 Web 容器，可通过 run 命令直接启动。
- 智能：配置中的默认值会根据依赖项自动调整。

使用 Spring Boot 可以轻松构建一个企业级的应用并且快速上线，而不用担心配置的准确性和安全性。图 1.2 所示是 Spring Boot 与 Spring Cloud、Spring Cloud Data Flow 的关系。

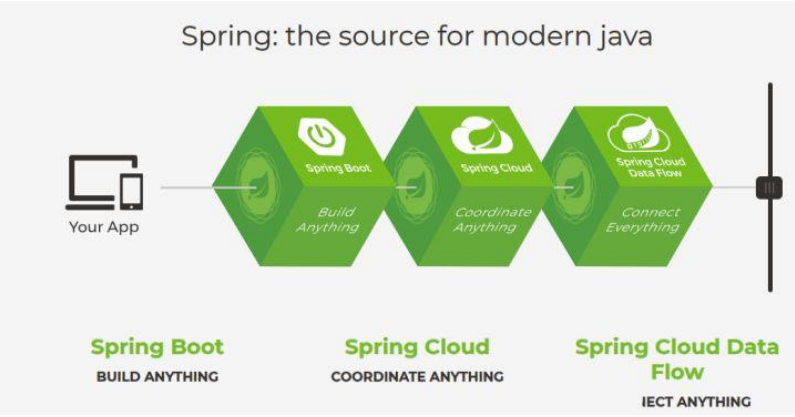

图 1.2　Spring Boot 与 Spring Cloud、Spring Cloud Data Flow 的关系

1.1.3　Spring Boot 的优势

为什么选择 Spring Boot 而不是其他的解决方案？理由有以下几点：

（1）成熟：Spring Boot 基于 Spring Framework。Spring Framework 已经开发超过 15 年，是 J2EE 的轻量级替代方案。

（2）稳定：Spring 生态中的核心模块长期稳定运行，并且它们的更改都向后兼容。开发人员在做版本升级的过程中，不会感到“举步维艰”。

（3）基于 JVM（Java 虚拟机）：Spring 是基于 Java 的，自然依赖于 JVM。JVM 上除了 Java 之外还可以运行其他的语言，例如：Kotlin、Groovy、Scala 等，Spring Boot 同样可以使用这些语言进行开发。

（4）由公司运作的开源项目：这意味着项目可以有规律地更新以及维护有基本的保障。

（5）云原生：Spring Boot 遵循云应用程序的部署原则，并为开箱即用的云做好了准备。它与 Spring Cloud 一起，可以轻松构建分布式系统。

（6）丰富的支持：使用 Spring 可以轻松地将应用连接到不同的关系型数据库、NoSQL、消息队列等中间件。

（7）灵活性：使用 Spring Boot 既可以开发经典的服务端（或称为服务器端，本书统一简称为服务端，以便具有更广义的含义）渲染 Web 应用，也可以开发 RESTful 或者其他形式的 Web-API，甚至可以创建批处理和常规命令行应用程序。

1.2 Spring Boot 2.3 开发环境

在正式编码之前，还需要做一些准备工作。首要任务是挑选并搭建好 Spring Boot 开发环境。一个基础的Spring Boot开发配置包括：JDK（Java Development Kit，Java开发工具集）、IDE（Integrated Development Environment，集成开发环境）以及一款自动化构建工具。得益于开源社区的繁荣，这些配置有不少备选项可供选择。挑选合适的配置是各类开发中绕不开的一个话题。

1.2.1 选择合适的 JDK

目前 Spring Boot 2.3 已对当前最新的 JDK 14 提供了支持，可以在 Spring Boot 中体验到强大又“炫酷”的功能。不过笔者仍推荐使用 JDK 8 或 JDK 11 这两个版本，大多 Java 类库基于这些版本构建，在学习与开发中能够少走很多弯路。除自身的版本之外，JDK 还有发行版之分。

- Oracle JDK。Oracle JDK 称得上是一个经典的选择，Java 归 Oracle 所有，Oracle JDK 自然在市场占有率上占主导地位。在 Oracle 官网即可下载到不同版本的、面向不同操作平台的 JDK 安装程序。
- Liberica JDK。对一般用户而言，Liberica 算得上是最友好的 OpenJDK 发行版了。以 Windows 平台的 JDK 安装为例，许多其他发行版的 JDK 安装过程中都免不了手动配置环境变量，而 Liberica JDK 的 Windows MSI 安装包会自动配置环境变量，并且自动关联 jar 打开方式，省时省力。
- Adopt OpenJDK。一个完全免费的 OpenJDK 版本。这个版本完全免费并且对于 JDK 8 和 JDK 11 提供不超过 4 年的支持。

笔者以 Windows 环境下安装 Liberica JDK 8u252+9 为例，介绍 JDK 的安装步骤：

（1）打开页面 https://bell-sw.com/pages/downloads/#/java-8-lts 下载，如图 1.3 所示。

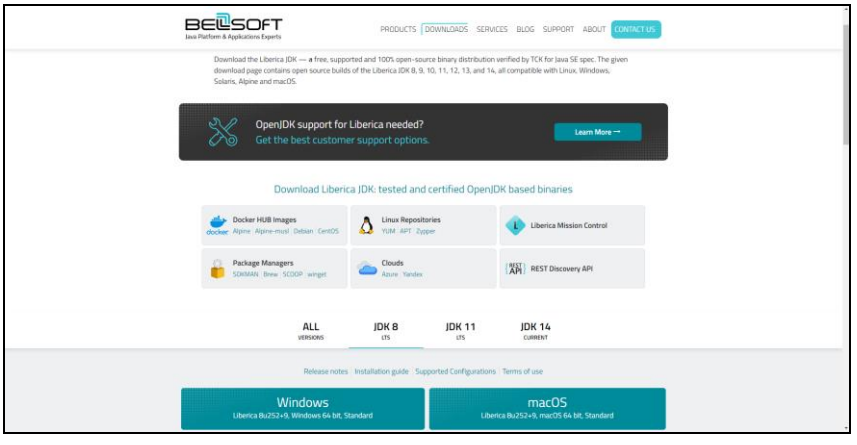

图 1.3 Liberica JDK 下载页面

（2）单击图中左下角的“Windows”按钮，开始 Windows 平台下的 Liberica JDK 安装包下载。

（3）MSI 格式的安装包下载完成之后，并运行安装包。其中没有需要注意的配置项，一路选择默认项即可。

（4）运行命令行程序 cmd.exe，执行命令“java -version”，如果出现版本信息，即说明 JDK 安装成功，如图 1.4 所示。

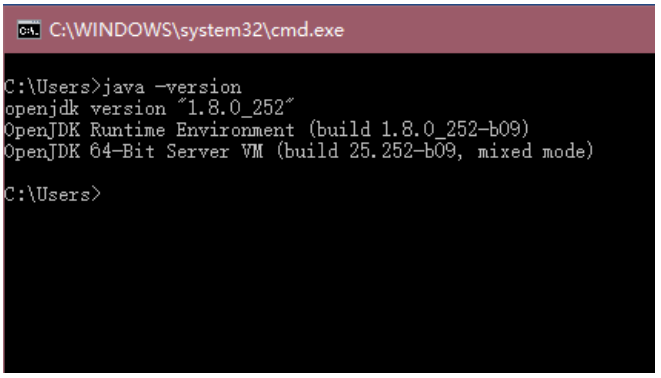

图 1.4　验证 JDK 是否安装成功

1.2.2　选择趁手的 IDE

如果把开发人员比作士兵，那 IDE 就是士兵手中的武器，特别是 Spring Boot 开发过程中对 IDE 的依赖尤为显著。IDE 的选择也是十分丰富，下面列举几款主流的 IDE 供大家选择。

- IntelliJ IDEA（以下简称：IDEA）。目前最流行的 Java IDE 之一，提供诸多功能以提升开发人员的开发体验。笔者在本书介绍的项目构建都是借助 IDEA 来实现的。IDEA 的索引系统是 IDEA 的特色之一，该系统提供更智能的提示以及更便捷的操作。这款 IDE 的优点很多，还需要读者慢慢探索。
- Eclipse。曾经是市场占有率最高的 IDE 之一，具有丰富的插件支持，同样是一款功能强大的 IDE。Spring 社区还在 Eclipse 的基础上提供了 STS 版本（Spring Tool Suite），与 IDEA 相比不遑多让。
- Visual Studio Code（以下简称 VSCode）。严格来说，这款工具虽然称不上是 IDE，但丰富的插件让它与以上两款 IDE 相比也是毫不逊色。基础的插件选择 Java Extension Pack 以及 VS Code 版本的 Spring Tools4 即可开始开发 Spring Boot 应用。

笔者同样以 Windows 平台安装 IDEA 2020.1.3 Community 版本为例，介绍 IDE 的安装及配置流程：

（1）打开官网下载页面：https://www.jetbrains.com/idea/download/#section=windows，如图 1.5 所示。

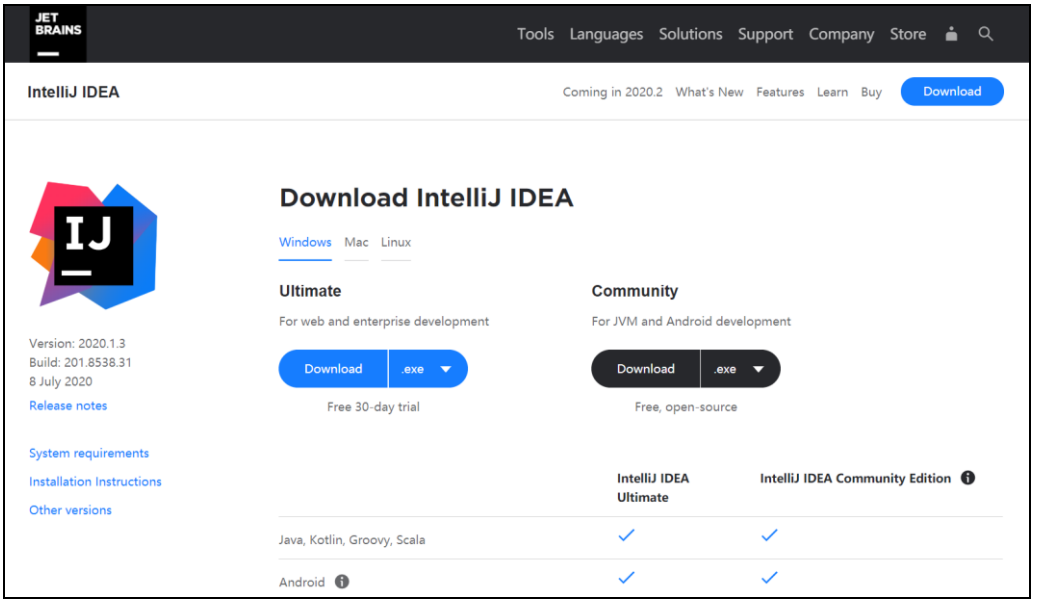

图 1.5　IDEA 下载页面

（2）单击 Community 下的 Download 按钮下载。

（3）下载完成后，运行安装程序。安装过程同样一直选择默认项即可。

（4）打开 IDEA 后即可创建 Java 项目。在窗口的 Project SDK 栏目，可以选择 JDK 版本（见图 1.6）。根据需要可以选择 IDE 自带的 JDK 或者本机安装好的 JDK。

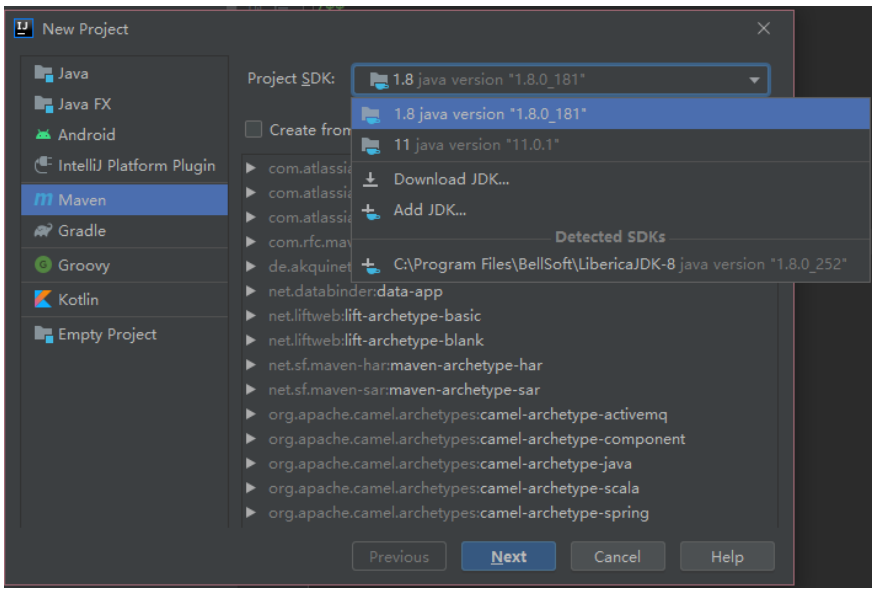

图 1.6　IDEA 创建新项目

1.2.3　选择适用于大型项目的自动化构建工具

当 IDE 准备好之后，理论上来说已经具备了开始编码工作的条件。但实际上，这个过程中经常会遇见很多与编程无关的项目管理工作。比如依赖管理、编译源码、单元测试、项目部署等。当项目的规模并不大的时候，这些工作可以手工实现。等到着手进行大型项目的开发时，特别是使用 Spring Boot 进行企业级的 Web 应用开发，这些工作可能会比编码工作本身更为棘手。这种情况下就需要用到自动化的构建工具来协助我们从前到后地完成编译、连接、打包、依赖管理等工作，一步一步地将源代码打包成可执行的形式。

Java 开发中常见的构建工具有三款：Ant、Maven 和 Gradle。相较于 Maven 与 Gradle，Ant 的历史更为久远。相应地，Ant 不及后来者 Maven 与 Gradle（智能），体验上较为烦琐，渐渐不再流行。因此笔者在这里只着重介绍 Maven 与 Gradle。

Maven 基于 Ant 的项目构建功能解决了构建过程中的两个问题：构建方式及其依赖项。以 Ant 为代表的早期构建工具，在构建过程中需要显式声明各类信息，比如源码路径、输出路径、编译具体步骤等。Maven 在这方面同样使用到“约定优于配置”的理念，对构建方式进行约定，这样一来，配置中需要关注的方面变少许多，相应地提升了开发人员的开发效率。在便利的配置之外，Maven 提出了仓库的概念，可以自动化地管理依赖类库。为了方便 Maven 进行依赖管理以及开发人员对依赖进行配置，Maven 规定在 pom.xml（Maven 配置文件）中需要声明依赖的“坐标”。

```
<project>
  <! -Maven 2.x POM 的模型版本始终为 4.0.0- >
 <modelVersion> 4.0.0 </ modelVersion>
  <! -项目坐标，即一组唯一标识此项目的值->
  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-app </ artifactId>
  <version> 1.0 </ version>
  <! -库依赖->
  <dependencies>
    <dependency>
      <!  -所需库的坐标->
      <groupId> junit </ groupId>
      <artifactId> junit </ artifactId>
      <version>3.8.1 </ version>
      <! -此依赖项仅用于运行和编译测试->
      <scope>测试</ scope>
    </ dependency>
  </ dependencies>
</ project>
```

以上是一个 pom.xml 的代码片段，其中由<dependency>包裹的部分便是代表 junit 依赖的坐标。groupId 代表负责维护类库的组织名，artifactId 代表类库对应的项目名，version 则是类库的版本号。这三项合一构成一个完整的 Maven 依赖坐标。

相较于 Maven，Gradle 就更为“新潮”了。它是一款基于多语言开发的自动化构建工具，基于 Ant 和 Maven 的概念引入了基于 Groovy 语言（一种基于 JVM 的敏捷开发语言）的 DSL（Domain-Specific Language，领域特定性语言），而并非 Maven 和 Ant 一直坚守的 XML 形式。采用 DSL 而非 XML 的形式大大提升了配置文件的可阅读性。例如上文的 junit 依赖，在 Gradle 中只

需要用一行来表示：

```
compile(junit: junit:3.8.1)
```

除此之外，Gradle 还将构建工具的灵活性提升到了一个新高度，使用 Groovy 语言可以轻松地在 Gradle 配置文件 build.gradle 中修改项目构建的生命周期，类似的操作在 Maven 中需要花费不少时间才能得以实现。

Gradle 虽然如此强大，但并不能说明使用 Gradle 是绝对优于 Maven 的选择。其中缘由十分复杂，还需要在实际开发使用中细细体会。另外，这两款构建工具已内置于 IDEA 中，并且这两款工具可以使用“wrapper”模式，无需另外下载安装。具体的操作内容将在本书第 2 章做详细介绍。

1.3 Spring Initializr 初始化项目

在 Spring 官方的开发指南当中有提到，所有 Spring 应用程序都应该从 Spring Initializr 开始。

1.3.1 什么是 Spring Initializr

Spring Initializr 是 Spring 官方提供的一个项目初始化工具。它是一个可扩展的 API，用以生成基于 JVM 的 Spring Boot 项目的“骨架”，并检查用于生成项目的元数据，例如列出可用的依赖项以及版本号。这个工具有不同的形式，可以单独使用，也可以嵌入其他工具（1.2.2 小节中提到的 IDE 中均有 Spring Initializr 的支持），有 Web UI 的应用，也有命令行的程序。

1.3.2 开始吧！start.spring.io

如果读者仍然不大理解什么是 Spring Initializr，那么可以在浏览器中打开链接：https://st-art.spring.io/来一探究竟。图 1.7 所示是 start.spring.io 对应的 Web 页面。

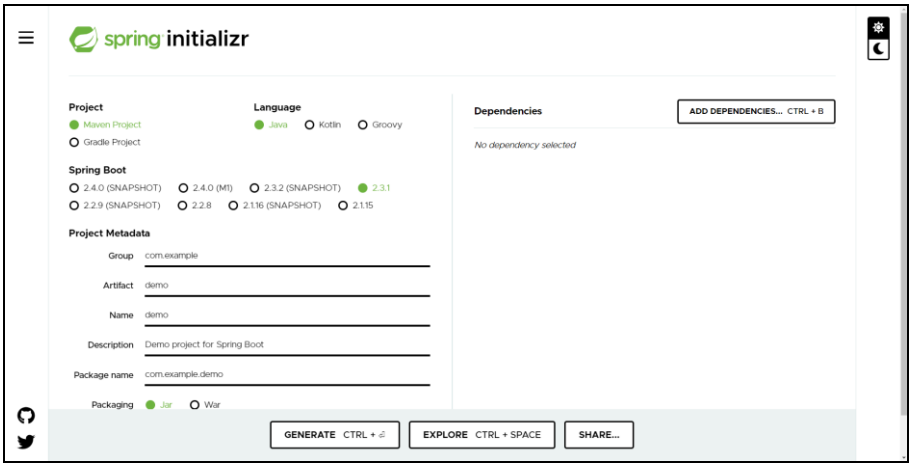

图 1.7 start.spring.io 页面

在页面的 Project 栏中，可以看到了两个熟悉的名词“Maven”以及“Gradle”。用意也十分明显了，在这里可以选择用于构建项目的构建工具。在 Language 栏中，可以选择项目将使用的编程语言。得益于 JVM 的支持，除了经典的 Java 语言，在开发 Spring Boot 时还可以使用 Kotlin 和 Groovy，它们都是非常优秀的 JVM 语言。

在 Spring Boot 这一栏，提供了 Spring Boot 版本的选项。简单介绍一下 Spring Boot 的版本号：

- Release：最终版本。Release 不会以单词形式出现在软件封面上，取而代之的是符号(R)。
- GA：General Availability。正式版本，官方推荐使用此版本，在国外都是用 GA 来说明 Release 版本的。
- RC：Release Candidate，发行候选。该版本已经相当成熟了，基本上不存在导致错误的 BUG，与即将发行的正式版相差无几。
- M：Milestone，又叫里程碑版本。表示该版本较之前版本有功能上的重大更新。
- SNAPSHOT：快照版，可以稳定使用，且仍在继续改进版本。
- Beta：该版本相对于 α 版已有了很大的改进，消除了严重的错误，但还是存在着一些缺陷，需要经过多次测试来进一步消除错误。
- Alpha（不建议使用）：主要是以实现软件功能为主，通常只在软件开发者内部交流，BUG 较多，需要继续修改。
- PRE（不建议使用）：预览版，内部测试版。主要是给开发人员和测试人员测试和查找 BUG 用的。

在图 1.8 中可以看到 Spring Boot 部分版本以及对应标识。在实际学习与开发过程中，笔者更推荐使用相对稳定的 Release 版作为候选项。

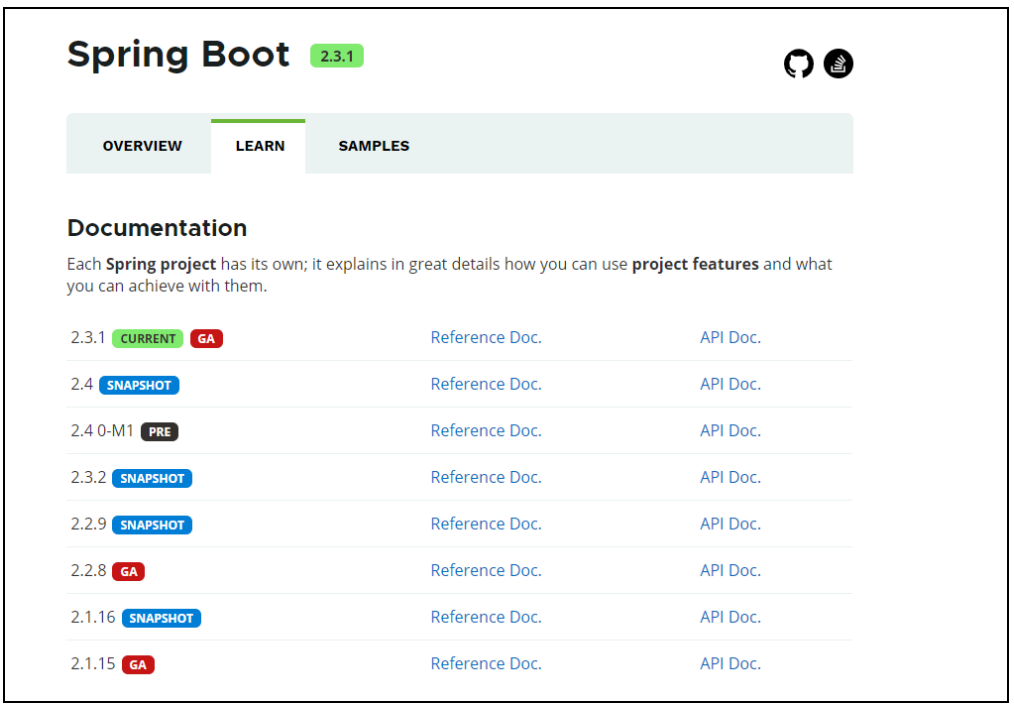

图 1.8　Spring Boot 当前版本以及版本号标识

回到 Spring Initializr 的页面。在 Project Metadata 栏目需要填入将要构建项目的元数据。由上至下依次是：

- Group：项目组织名。
- Artifact：用于构成依赖坐标的项目名。
- Name：用于描述项目的项目名，可以与 Artifact 项保持一致。
- Description：对项目的基本描述内容。
- Package name：包名，通常可以由 Group 与 Artifact 共同组成。
- Packaging：打包方式。
- Java：Java 版本号。

在 Project Metadata 栏中填入以上信息后，Spring Initializr 就能帮助开发人员生成一个基于 Maven 或 Gradle 构建的基本工程了。不过项目难免会需要额外的依赖，那么还需要把目光再投向 Dependencies 这个栏。单击“ADD DEPENDENCIES”按钮，挑选项目所需要的依赖以及开发工具。图 1.9 所示为单击按钮后呈现的页面。

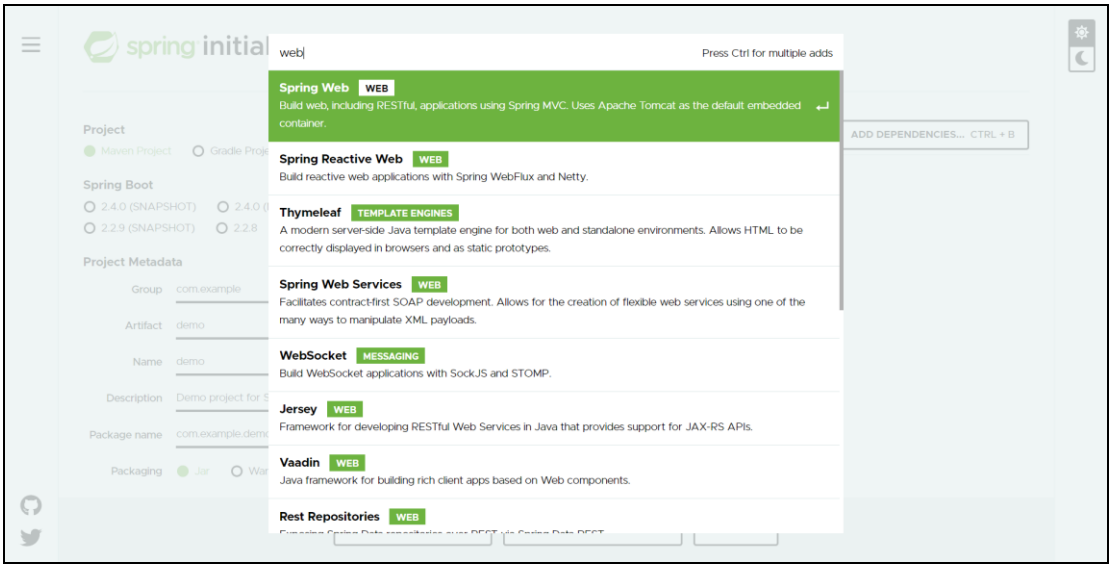

图 1.9　ADD DEPENDENCIES 页面

比如需要 Spring Web 的依赖。在输入框中输入 Web，就可以看到 Spring Web 的候选项。单击 Spring Web 后，这个依赖就被加入到了依赖列表当中。

配置好依赖之后，单击页面左下角的“GENERATE”按钮，一个初始化过后的 Spring Boot 工程就开始下载了。

1.3.3　使用 IDE 初始化 Spring Boot 工程

start.spring.io 固然方便，但并不是人人都乐意于在创建一个新 Spring Boot 项目时都打开浏览器操作一番。不必苦恼，针对 Spring Boot 工程的初始化，各大 IDE 也都有插件提供相关的支持。

下文以 IDEA2020.1.3 Community 版本为例进行讲解，在该版本的 IDEA 中会使用插件“Spring Assistant”来支持 Spring Boot 工程初始化。这个插件需要另行安装，操作步骤如下：

（1）在工具栏中依次选择“File”→“Settings”，打开 IDEA 的设置页面。
（2）在设置页面的左侧选择“Plugins”，打开 IDEA 的插件管理页面。
（3）在搜索栏中输入“Spring Assistant”，即可看到所需的插件。
（4）单击右上角的“Install”按钮安装插件。
（5）待安装完成后，单击“Restart IDE”按钮以重启 IDE。

这样就完成了一个插件的安装。Spring Assistant 的安装页面如图 1.10 所示。

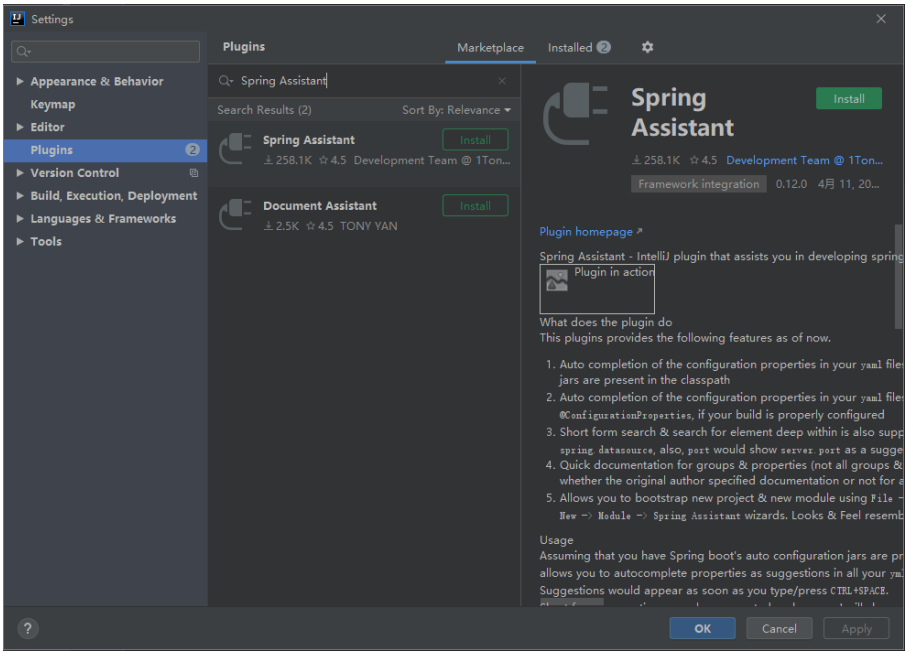

图 1.10　Spring Assistant

在插件安装完成之后，就可以在 IDEA 中初始化一个 Spring Boot 工程了。操作步骤如下：

（1）在工具栏中依次选择“File”→“New”→“Project…”，打开新建项目页面，如图 1.11 所示。

（2）在新建项目页面左侧可以选择“Spring Assistant”选项。对应页面可以配置 Spring Initializr 的服务地址。目前仅需要使用默认的 start.spring.io 即可。单击“Next”按钮进入下一步。

（3）这一步需要配置的内容和在 1.3.2 小节中配置的相同，这里就不再展开了。配置页内容如图 1.12 所示。

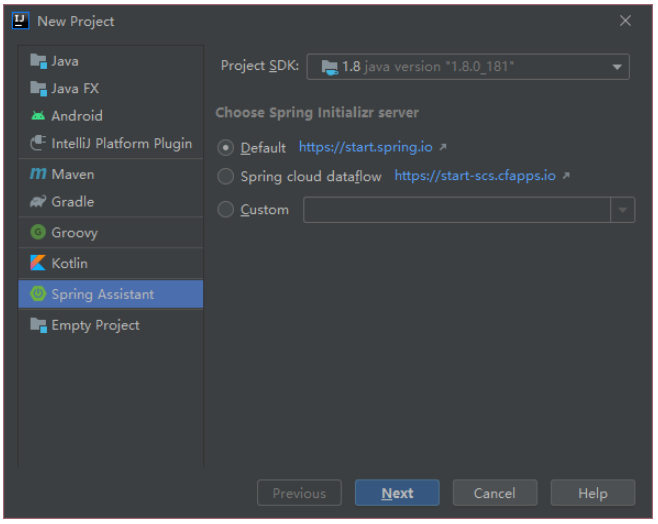

图 1.11　创建新项目页面

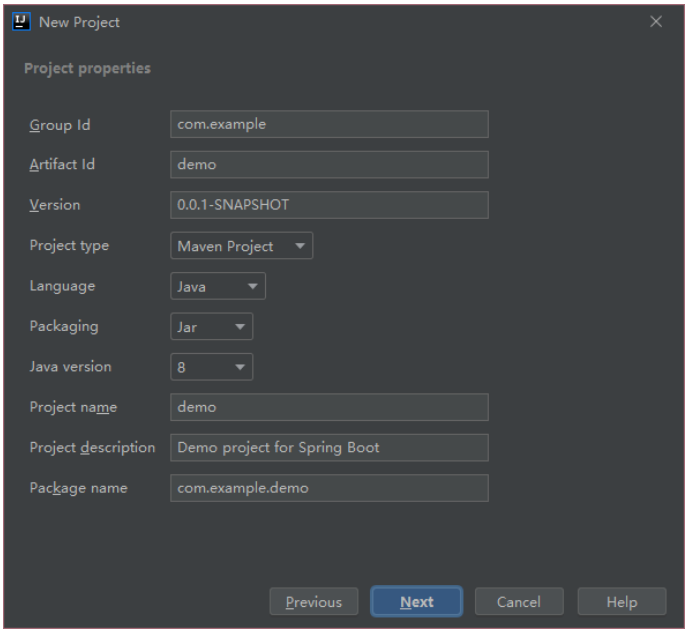

图 1.12　IDE 中配置 Spring Initializr 所需参数

1.3.4　初探 Spring Boot CLI

上文已经介绍了两种方法来初始化 Spring Boot 工程。不过都需要打开页面选择，有没有更高效更“极客”的方法？还真有！Spring 社区为了方便 Spring Boot 开发，推出了一款命令行工具——Spring Boot CLI。当然并不一定需要 CLI 就能着手开发功能强大的 Spring Boot 程序，但这绝对是最快的方法了。

首先，需要安装 Spring Boot CLI，操作步骤如下：

（1）在 Spring Software Repository 里下载 Spring Boot CLI 的发行版。链接地址为 https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/2.3.1.RELEASE/spring-boot-cli-2.3.1.RELEASE-bin.zip。

（2）下载后解压安装包，并把./ spring-2.3.1.RELEASE/bin 路径配置到 Path 环境变量中。

（3）测试是否配置成功。打开 cmd，输入命令“spring --version”。当页面出现 Spring Boot CLI 的版本号时，如“Spring CLI v2.3.1.RELEASE”。这就说明 Spring Boot CLI 已经安装完成了。

使用 Spring Boot CLI 来初始化 Spring Boot 工程，所依赖的是其中的“init”命令，使用 init –list 即可查看可用参数：

```
spring init -list
```

假设要构建一个 Web 应用，其中使用 JPA 实现数据持久化，使用 Spring Security 进行安全管理，可用--dependencies 或者-d 来指定初始依赖：

```
spring init -dweb,jpa,security
```

或者需要用 Gradle 来构建项目。--build 参数将 Gradle 指定为构建工具：

```
spring init -dweb,jpa,security --build gradle
```

无论是 Maven 还是 Gradle 的构建，都会产生一个可执行 jar 文件。但如果需要的是一个 war 格式的文件该怎么办呢？可以通过--packaging 或者-p 参数来解决：

```
spring init -dweb,jpa,security --build gradle -p war
```

细心的读者可能会发现，包括之前用页面初始化项目时，最终的初始化的结果都是一个名为 demo.zip 的压缩包。如果实在不想在之后执行解压操作，Spring Boot CLI 可以帮我们做到这一点吗？答案是肯定的。在命令的最后指定一个目录，文件下载完成后会自动解压这个目录中：

```
spring init -dweb,jpa,security --build gradle -p war myapp
```

至此，一个初始化的 Spring Boot 程序就创建好了。

1.4　Spring Boot 目录结构

根据上一节的介绍，我们已经在本地完成了一个基础 Spring Boot 工程的初始化。一个基础的工程结构包含一个主应用类、一个配置文件以及若干个与构建工具相关的文件。下面将介绍 Spring Boot 工程的目录结构。

1.4.1　初始化的工程结构

首先通过命令“spring init -dweb,jpa,security --build maven -p jar basic-project”创建好一个基础工程。工程的目录结构如图 1.13 所示。

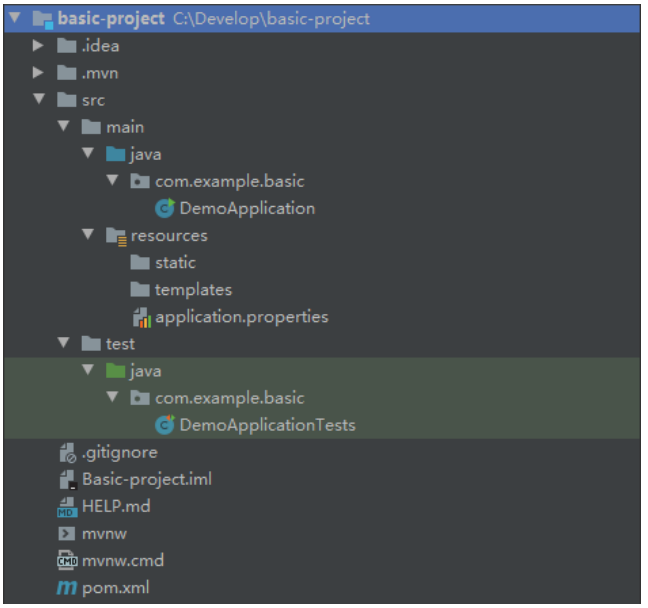

图 1.13　Spring Boot 工程初始的目录结构

由上至下的文件以及目录：

（1）.idea：由 IDEA 而非 Spring Initializr 创建。包含该项目的历史记录、版本控制信息等。

（2）.mvn：Maven 的 Wrapper 目录，对应该项目选用的构建工具。如若选用了 Gradle 为项目构建工具，该目录将会被替代为“gradle”。

（3）src：源码目录。包含源码、测试代码以及资源文件目录。

（4）main：主体目录。在主体程序目录下，需要放置主体程序构建所需的代码文件以及资源文件。

（5）test：测试目录。测试目录存放用于构建测试用例相关的代码文件以及资源文件。

（6）java：java 代码目录。在 java 目录之下存放.java 后缀的代码文件。

（7）resources：资源文件目录。在资源文件目录中，存放除代码文件之外任何构建程序所需的“资源文件”，包括但不限于配置文件、模板文件、图片文件等。

（8）DemoApplication：主体程序入口类。

```
package com.example.basic;

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

@SpringBootApplication
public class DemoApplication {

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

}
```

在这个类中可以看到熟悉的 main 函数，这说明该类是整个 Spring Boot 应用的入口类。

（9）static 目录与 template 目录：由于初始化的过程中引用了“web”模块，Spring Initializr 创建了这两个目录，以便于构建 MVC（Model-View-Controller，一种软件设计模式）模式的应用。static 主要用于存放 js/css 或者图片之类的静态文件，template 用于存放页面模板文件。

（10）application.properties：Spring Boot 项目的配置文件。还记得前文提到的“约定优先于配置”吗？Spring Boot 提供了许多默认的配置。当默认配置不满足需求时，在配置文件中填写需要的配置项是在 Spring Boot 项目中用于覆盖默认配置的方法之一。

（11）DemoApplicationTests：Spring Boot 项目的测试类。

（12）.gitignore：使用 git 做版本管理的话，需要在该文件内列出忽略文件的列表。

（13）Basic-project.iml：iml 后缀的文件为 IDEA 用于存储开发环境相关信息的配置文件。

（14）HELP.md：md 格式的帮助文档。该文件根据应用程序中所选依赖而定制，包含与这些依赖有关的指南以及参考文档，以便开发人员更好地进行开发。

（15）mvnw 与 mvnw.cmd/gradlew 与 gradle.bat：mvnw 与 gradlew 分别为 Maven Wrapper 以及 Gradle Wrapper 附带的脚本，与（2）中提到的目录文结合起来，以方便用户无须手动安装这些构建工具就能享用它们带来的帮助。

（16）pom.xml/build.gradle：pom.xml 与 build.gradle 分别对应 Maven 和 Gradle 的构建配置文件。需要在其中声明构架项目所需的依赖以及其他配置项。

（17）settings.gradle：如果使用 Gradle 构建项目的话，还会看到 settings.gradle 这样一个文件。作用在于定义所有包含的子模块并标记模块树的目录根。

1.4.2　推荐的工程结构

上一小节介绍了一个初始化的 Spring Boot 目录的结构以及所包含的文件。本小节在此基础上进行延伸，提出一些关于 Spring Boot 工程结构的建议。

提　示

Spring Boot 本身并不会对目录格式进行约束，结构完全可以符合个性并且天马行空。但是一个工程化的项目有越多约定俗成的元素，就更容易上手开发以及学习，这是笔者鼓励初学者根据“最佳实践”进行代码编写的理由。

以下是一个比较经典的 Spring Boot 项目包的结构：

```
com
  +- example
    +- myproject
      +- Application.java
      |
      +- domain
      |  +- Customer.java
      |  +- CustomerRepository.java
      |
      +- service
      |  +- CustomerService.java
```

```
      |
      +- web
      |  +- CustomerController.java
      |
```

- root package：com.example.myproject，所有的类以及其他的 package 都在 root 下。
- 应用主类：Application.java，该类大多直接位于 root package 下。通常会在应用主类加入一些注解以实现一些配置的效果。例如 1.4.2 小节（8）列出的主类中可以看到@SpringBootApplication 这个注解。该注解为一个复合注解，作用是用于告诉 Spring Boot 该应用已经开启了自动配置以及组件扫描。
- com.example.myproject.domain 包：用于定义实体映射关系与数据访问相关的接口和实现。
- com.example.myproject.service 包：用于编写业务逻辑相关的接口与实现。
- com.example.myproject.web：用于编写 Web 层相关的实现，比如：Spring MVC 的 Controller。

以上便是一个 Spring Boot 工程的推荐结构。root package 与应用主类的位置是整个结构的关键。由于应用主类在 root package 中，因此按照上面的规则定义的所有其他类都处于 root package 下的其他子包中。应用主类中由@SpringBootApplication 注解开启的自动配置以及扫描，是针对 root package 以及其子包的。如若情况发生变化，例如：

```
com
  +- example
    +- myproject
      +- Application.java
      |
      +- domain
      |  +- Customer.java
      |  +- CustomerRepository.java
      |
      +- service
      |  +- CustomerService.java
      |
    +- web
    |  +- CustomerController.java
|
```

将 web 目录放置于与 root package 同级。这种情况下 CustomerController.java 无法自动被扫描到，需要使用其他注解对 Spring Boot 工程进行配置。具体的方式有如下两种：

（1）使用@ComponentScan 注解指定具体的加载包，比如：

```
@SpringBootApplication
@ComponentScan(basePackages="com.example")
public class Application {

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

}
```

（2）使用@Bean 注解来初始化，比如：

```
@SpringBootApplication
public class Application {

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

    @Bean
    public CustomerController customerController() {
        return new CustomerController();
    }

}
```

这些方式可以帮助实现目录结构的自定义，不过为了开发的便利性和可维护性，推荐参考“最佳实践”来组织代码。

1.4.3　Maven Wrapper 让构建工具随源码分发

Spring Initializr 为开发人员提供了以 Wrapper 模式来使用构建工具，其中包含 Maven Wrapper 以及 Gradle Wrapper。这种方式为开发过程带来了极大的便利，因为进一步提高了对开发环境依赖的配置功能。

Maven Wrapper 实质上是依赖一款插件实现的，插件为 takari-maven-plugin，项目地址为 https://github.com/takari/takari-maven-plugin。要使用到这个功能，需要进入项目的根目录并运行以下命令：

```
mvn -N io.takari:maven:wrapper
```

还可以指定 Maven 版本号：

```
mvn -N io.takari:maven:wrapper -Dmaven=3.5.2
```

选项-N 表示－non-recursive，因此包装程序将仅应用于当前目录的主项目，而不应用于任何子模块。

之后可以使用以下命令进行程序的构建：

```
mvnw clean install
```

在这之后，Maven 将会下载到目录“$USER_HOME/.m2 /wrapper/dists”下。

注　意

使用官方源进行下载将会花费很长的时间。在./mvn/wrapper/maven- wrapper.properties 文件中将配置项 distributionUrl 设置为阿里云的分发源即可解决，例如：

```
distributionUrl=https://maven.aliyun.com/repository/central/org/apache/maven/apache-maven/3.6.3/apache-maven-3.6.3-bin.zip
```

1.5 构建第一个 Spring Boot 项目

本节将介绍如何从零开始构建一个 Spring Boot 项目。主要需求是使用 MVC 模式配合 Mustache 模板（一种模板引擎）实现一个简单的博客程序，为这个程序编写若干个测试用例。最后还需要发布博客相关的数个 API，让该程序除了是一个“单体应用”之外，还可以成为一个前后端分离架构中的“Web 后端程序”。

1.5.1 经典“Hello World”

按照 1.3.4 小节中介绍的 Spring Boot CLI 初始化一个 Spring Boot 项目。

【示例 1-1】

我们最先能预想到的依赖有 web、mustache、jpa、devtools，根目录暂且设定为 com.example.blog，将项目目录设置为 myblog。输入以下指令：

```
    spring init -dweb,mustache,jpa,h2,devtools --package-name=com.example.blog
myblog
```

创建好项目之后，编写第一个 Controller 类。还记得“最佳实践”的工程结构吗？参照那个结构创建目录 com.example.myblog.controller。在该目录下创建“Html-Controller”类。

```
    package com.example.myblog.controller;

    import org.springframework.stereotype.Controller;
    import org.springframework.ui.Model;
    import org.springframework.web.bind.annotation.GetMapping;

    @Controller
    public class HtmlController {

        @GetMapping("/")
        public String blog(Model model) {
            model.addAttribute("title", "Hello World");
            return "blog";
        }
    }
```

在创建好 HtmlController 之后，还需要创建一个对应的模板文件，以展示 Controller 返回的内容。在“resource/templates”目录下创建模板文件“blog.mustache”以及一些子模板。当然，这个名字可以是任何“*.mustache”，只要 controller 中的字符串返回值与文件名保持一致即可。

在路径 src/main/resources/templates/下创建 blog.mustache：

```
    {{> header}}

    <h1>{{title}}</h1>
```

```
{{> footer}}
```

{{> header}}为子模板，这些类似的子模板在运行时才会呈现。

在路径 src/main/resources/templates/下创建 header.mustache：

```
<html>
<head>
  <title>{{title}}</title>
</head>
<body>
```

resources/templates/footer.mustache：

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

在以上文件准备好了之后，通过 mvn spring-boot:run 或者其他方式来运行这个程序。在浏览器中输入：http://localhost:8080/，将会看到一个大大的“Hello World”。

1.5.2　使用 JUnit 5 测试

在完成了一个基础的功能之后，为了保证功能的正确性，需要针对该功能编写若干测试用例。自动化的测试在整个开发流程中是不可或缺的。尽可能多地编写测试用例，不仅可以帮助开发人员减少 Bug，还能促进开发人员对需求的理解。这里需要使用一款经典的测试框架 JUnit 5 来帮助实现对程序的自动化测试。

【示例 1-2】

在路径 src/test/java/com/example/myblog 下创建 IntegrationTests.java：

```
    @SpringBootTest(classes = {BlogApplication.class}, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
    class IntegrationTests{

         //Spring 提供的在测试环境下访问 Rest 服务的客户端
        @Autowired
        TestRestTemplate restTemplate;

        @Test
        void assertBlogPageTitle_Content_And_StatusCode() {
            //访问路径“/”，以 String 类型来解析响应的主体 entity.body
            ResponseEntity<String> entity = restTemplate.getForEntity("/",
String.class);
            //判断响应的状态码为 HttpStatus.OK，即 200
            assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
            //判断 entity.body 包含“<h1>Hello World</h1>”
            assertThat(entity.getBody()).contains("<h1>Hello World</h1>");
        }
    }
```

这是一个借助注解“@SpringBootTest”实现的集成测试的测试用例。方法的命名结合了驼峰

命名和蛇形命名，在方法名过长的情况下有助于提高可读性。在使用 JUnit 5 编写测试用例时，通常会把编写的测试用例分为单元测试和集成测试。这两者的区别在于测试的粒度不同。单元测试的粒度通常测试一个独立的模块甚至单独的一个类。测试过程中，该模块不与依赖项进行任何交互，以确认该模块内部在做正确的事。集成测试的粒度则覆盖多个模块，关注多个模块在协同工作的情况下是否正常。

目前需要测试的内容是控制器、模板文件在 Spring Boot 的协同工作结果，这是一个集成测试，所以需要用到 Spring Boot 提供的“@SpringBootTest”注解在测试环境创建应用程序上下文。

有时候需要在给定类之前或者之后执行一个方法，此时需要依赖到注解“@BeforeAll”和“@AfterAll”。不过，如果把这两个注解在 JUnit 5 中用于修饰常规方法，那么在使用之前需要编写一个配置文件。因为在默认情况下，JUnit 5 要求注解“@BeforeAll”和“@AfterAll”修饰的对象为静态方法。

在路径 src/test/resources 下创建配置文件 junit-platform.properties：

```
    junit.jupiter.testinstance.lifecycle.default = per_class
```

配置完成之后，就可以再更新上文的 IntegrationTests.java 了：

```
    @SpringBootTest(classes = {BlogApplication.class}, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM PORT)
    class IntegrationTests{

         //Spring 提供的在测试环境下访问 Rest 服务的客户端
        @Autowired
        TestRestTemplate restTemplate;

        @BeforeAll
        void setup() {
            System.out.println(">> Setup");
        }

        @Test
        void assertBlogPageTitle_Content_And_StatusCode() {
            //访问路径“/”，以 String 类型来解析响应的主体 entity.body
            ResponseEntity<String> entity = restTemplate.getForEntity("/",
String.class);
            //判断响应的状态码为 HttpStatus.OK，即 200
            assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
            //判断 entity.body 包含“<h1>Hello World</h1>”
            assertThat(entity.getBody()).contains("<h1>Hello World</h1>");
        }

        @Test
    void assertArticlePageTitle Content And StatusCode() {
            System.out.println(">> TODO");
        }

        @AfterAll
        void teardown() {
            System.out.println(">>  Tear down");
        }
    }
```

1.5.3　创建工具类 CommonUtil

在构建博客应用的过程中，需要用到一些工具方法。目前需要的工具方法有：格式化时间和将标题转换为 slug 格式（slug 格式是一种方便构建 URI 命名方式的格式）。例如有一篇文章标题为“This is a title”，那么人们期望的 URI 格式可能为：

```
www.example.com/article/This is a title
```

但事实上这并不是一个有效的 URI。为了成为有效的 URI，空格会被转义为%20：

```
www.example.com/article/This%20is%20a%20title
```

不得不说，转义后的标题可读性变差了许多。为了增强 URI 的可读性，这里的空格可以被替换为“-”字符：

```
www.example.com/article/This-is-a-title
```

这种格式就是 slug 格式，在定义 API 的时候推荐使用该格式来命名超过一个单词的路径名。

【示例 1-3】

在路径 src/main/java/com/example/myblog/util 之下创建 CommonUtil.java：

```
public class CommonUtil {

    private static final DateTimeFormatter englishDateFormatter;
    private static final Map<Long, String> daysLookup;

    public static String format(LocalDateTime localDateTime) {
        return localDateTime.format(englishDateFormatter);
    }

    //将 title 转换为 slug 格式:“this is a title” -> “this-is-a-title”
    public static String toSlug(String title) {
        return String.join("-", title.toLowerCase()
                .replace("\n", " ")
                .replace("[^a-z\\d\\s]", " ")
                .split(" "))
                .replace("-+", "-");
    }

    static {
        daysLookup = buildDaysLookup();
        englishDateFormatter = new DateTimeFormatterBuilder()
                .appendPattern("yyyy-MM-dd")
                .appendLiteral(" ")
                .appendText(ChronoField.DAY OF MONTH, daysLookup)
                .appendLiteral(" ")
                .appendPattern("yyyy")
                .toFormatter(Locale.ENGLISH);
    }

    //创建 appendText 需要用到的参数 daysLookup，用于提供 appendText 时的映射关系
```

```
    private static Map<Long, String> buildDaysLookup() {
        Map<Long, String> ret = new HashMap<>();
        for (int i = 1; i <= 31; i++) {
            ret.put((long) i, getDayOfMonthSuffix(i));
        }
        return ret;
    }

    //根据天数的个位获得对应的后缀
    private static String getDayOfMonthSuffix(int n) {
        switch (n % 10) {
            case 1:
                return "${n}st";
            case 2:
                return "${n}nd";
            case 3:
                return "${n}rd";
            default:
                return "${n}th";
        }
    }
}
```

1.5.4 使用 JPA 进行数据持久化

这里计划使用 JPA（Java Persistence API）配合 H2 数据库进行数据的持久化，将文章以及作者信息写入到数据库中，并提供从数据库中读取对应数据的方法。当然，使用另外的数据库作为数据源也是可以的。

【示例 1-4】

首先需要创建文章以及作者的实体类。在路径 src/main/java/com/example/myblog/entity 下创建 Article.java：

```
@Entity
public class Article {

    @Id
    @GeneratedValue
    //主键
    private Long id;
    //标题
    private String title;
    //摘要
    private String headline;
    //内容
    private String content;
    @ManyToOne
    //作者
    private User author;
    //slug 格式的标题
    private String slug;
    //创建时间
```

```
    private LocalDateTime addedAt;

    public Article() {
        addedAt = LocalDateTime.now();
    }

    public Long getId() {
        return id;
    }

    //chain 风格的 setter
    public Article setId(Long id) {
        this.id = id;
        return this;
    }
   //以下为除 id 字段各个字段的 getter 与 setter，在此省略……
}
```

在路径 src/main/java/com/example/myblog/entity 下创建 User.java：

```
@Entity
public class User {
    @Id
    @GeneratedValue
    //主键
    private Long id;
    //登录名
    private String login;
    //名字
    private String firstName;
    //姓氏
    private String lastName;
    //描述
    private String description;
    //省略了 getter 与 setter 方法
}
```

以上两个类中包含 JPA 中提供的几个注解：

- @Entity：表明所修饰的类是一个实体类，如 Article.java。其默认的数据库表名为“article”，也可以通过注解的 name 属性修改表名，如@Entity(name="article_alias")。
- @Id：指定字段为主键。
- @ GeneratedValue：与@Id 配合使用，指定字段的生成策略为通用的生成策略。
- @ ManyToOne：声明该字段与对应的对象存在一对多的关系。

创建完实体类之后，需要根据需求创建对应的 repository 类，以支持对数据库的增删改查（CURD）。在路径 src/main/java/com/example/myblog/ repository 下创建 ArticleRepository.java：

```
public interface ArticleRepository extends CrudRepository<Article, Long> {
    //通过 slug 找到对应的文章
    Article findBySlug(String slug);
    //查询所有的文章并通过添加时间以及描述进行排序
    Iterable<Article> findAllByOrderByAddedAtDesc();
}
```

在路径 src/main/java/com/example/myblog/repository 下创建 UserRepository.java：

```
public interface UserRepository extends CrudRepository<User, Long> {
    //通过登录名找到对应的用户
    User findByLogin(String login);
}
```

在创建 Repository 类的时候，仅需要创建一个继承 CrudRepository<T, ID> 的 interface 接口即可。泛型标记符 T 为实体类的类型，ID 为实体主键的类型。创建好对应的 interface 接口之后，剩余的工作将交给 JPA 自动完成。或许这样会带来疑惑，对数据库增删改查就不用编写 SQL 语句吗？在这里，JPA 遵循“约定优先于配置”原则，只要根据 Spring 以及 JPQL 的约定来对方法进行命名，对应 SQL 语句的生成就都交给框架来实现了。具体的 JPA 使用技巧在本书第 4 章会展开介绍。

实现了数据持久化之后，免不了对功能做一番测试。在路径 src/test/java/com/exa-mple/myblog 下创建 RepositoriesTests.java：

```
@DataJpaTest
public class RepositoriesTests {

    @Autowired
    TestEntityManager entityManager;

    @Autowired
    UserRepository userRepository;

    @Autowired
    ArticleRepository articleRepository;

    @Test
void whenFindByIdOrNull thenReturnArticle() {
    //创建一个用户对象
        User leili = new
User().setLogin("leili").setFirstName("Lei").setLastName("Li");
        //将用户对象转变为托管状态
        entityManager.persist(leili);
        //创建一个文章对象
        Article article = new Article().setTitle("Spring Framework 5.0 goes
GA").setHeadline("Dear Spring community ...")
                .setContent("Lorem ipsum").setAuthor(leili);
        //将文章对象转变为托管状态
        entityManager.persist(article);
        //将托管状态的对象写入到数据库
        entityManager.flush();
        //根据 Id 查询文章否则返回 null
        Article found =
articleRepository.findById(article.getId()).orElse(null);
        //断言保存前的对象与返回值相等
        assertThat(article).isEqualTo(found);
    }

    @Test
void whenFindByLogin_thenReturnUser() {
    //创建一个用户对象
        User leili = new
```

```
User().setLogin("leili").setFirstName("Lei").setLastName("Li");
            //将对象直接保存至数据库
            entityManager.persistAndFlush(leili);
            //根据登录名查询对象
            User found = userRepository.findByLogin(leili.getLogin());
            //断言找到的对象与保存前的对象相等
            assertThat(found).isEqualTo(leili);
        }
    }
```

在这段测试代码中，可以看到新的注解“@DataJpaTest”。该注解是 Spring Boot 为 JPA 组件测试而提供的支持。使用该注解之后，将会把全局的自动配置禁用，转而开启仅与 JPA 测试有关的配置。并且带有@DataJpaTest 注解的测试都是事务性的，这意味着每个测试结束之后都会进行回滚，避免数据库中数据被测试所影响。

1.5.5　修改控制器以及对应模板文件

对数据库的增删改查已经通过 JPA 实现了，接下来修改之前实现的控制器以及模板文件。

更新 blog.mustache：

```
    {{> header}}

    <h1>{{title}}</h1>

    <div class="articles">

        {{#articles}}
            <section>
                <header class="article-header">
                    <h2 class="article-title"><a
href="/article/{{slug}}">{{title}}</a></h2>
                    <div class="article-meta">By
<strong>{{author.firstName}}</strong>, on <str
    ong>{{addedAt}}</strong></div>
                </header>
                <div class="article-description">
                    {{headline}}
                </div>
            </section>
        {{/articles}}
    </div>

    {{> footer}}
```

在路径 src/main/resources/templates/下创建 article.mustache：

```
    {{> header}}

    <section class="article">
        <header class="article-header">
            <h1 class="article-title">{{article.title}}</h1>
            <p class="article-meta">By
```

```
<strong>{{article.author.firstName}}</strong>, on <strong>{{art
    icle.addedAt}}</strong></p>
       </header>

       <div class="article-description">
          {{article.headline}}

          {{article.content}}
       </div>
    </section>

    {{> footer}}
```

更新完模板文件之后，要着手更新对应的控制器了。在此之前需要创建一个RenderedArticle.java。因为在很多情况下，页面展示的内容并不完全是数据库中存储的内容。这时需要借助业务对象模型（也叫领域模型）来帮助实体进行转换。Rendere-dArticle.java 便是领域模型中的 VO（view object）表现层对象。

在路径 src/main/java/com/example/myblog/domain 下创建 RenderedArticle.java：

```
    public class RenderedArticle {
       //slug 格式的标题
    private String slug;
    //标题
    private String title;
    //摘要
    private String headline;
    //内容
    private String content;
    //作者
    private User author;
    //创建时间
    private String addedAt;
    //省略了 getter 与 setter
    }
```

修改 HtmlController.java：

```
    @Controller
    public class HtmlController {

       private final ArticleRepository repository;

       public HtmlController(ArticleRepository repository) {
          this.repository = repository;
       }

       @GetMapping("/")
       public String blog(Model model) {
          //返回 title 字符串
          model.addAttribute("title", "Blog");
          //返回 articles 列表
          //这里用到了 StreamSupport.stream()方法，将 repository.
findAllByOrderByAddedAtDesc()的结果
          //转换为 stream，依次调用 render 方法进行对象的类型转换
```

```
            model.addAttribute("articles",
StreamSupport.stream(repository.findAllByOrderByAddedAtDesc().spliterator(),
true)
                    .map(this::render)
                    .collect(Collectors.toList()));
            return "blog";
        }

        //根据 slug 查询文章
        @GetMapping("/article/{slug}")
        public String article(@PathVariable String slug, Model model) {
            Article article = repository.findBySlug(slug);
            if (article == null) {
                throw new ResponseStatusException(HttpStatus.NOT_FOUND, "This
article does not exist");
            }
            RenderedArticle renderedArticle = render(article);
            //返回 title 对象
            model.addAttribute("title", renderedArticle.getTitle());
            //返回 article 对象
            model.addAttribute("article", renderedArticle);
            return "article";
        }

        //对象转换
        private RenderedArticle render(Article article) {
            return new RenderedArticle()
                    .setTitle(article.getTitle())
                    .setHeadline(article.getHeadline())
                    .setSlug(article.getSlug())
                    .setContent(article.getContent())
                    .setAuthor(article.getAuthor())
                    .setAddedAt(CommonUtil.format(article.getAddedAt()));
        }
    }
```

还缺少数据的初始化，可以通过 ApplicationRunner 的方式进行配置。在程序启动前，调用数据保存的方法，实现数据的初始化。

在路径 src/main/java/com/example/myblog/config 下创建 BlogConfiguration.java：

```
    @Configuration
    public class BlogConfiguration implements ApplicationRunner{

        private final UserRepository userRepository;
        private final ArticleRepository articleRepository;

        public BlogConfiguration(UserRepository userRepository, ArticleRepository
articleRepository) {
            this.userRepository = userRepository;
            this.articleRepository = articleRepository;
        }

        @Override
    public void run(ApplicationArguments args) throws Exception {
        //依次保存一条作者信息和两条文章信息
```

```
            User meimeihan= userRepository.save(new User()
                    .setLogin("meimeihan ")
                    .setFirstName("meimei")
                    .setLastName("han"));
            articleRepository.save(new Article()
                    .setTitle("the title1")
                    .setHeadline("headline1")
                    .setContent("content1")
                    .setAuthor(meimeihan));
            articleRepository.save(new Article()
                    .setTitle("the title2")
                    .setHeadline("headline2")
                    .setContent("content2")
                    .setAuthor(meimeihan));
        }
    }
```

同样，再编写一条测试用例来测试一下。更新 IntegrationTest.java：

```
    @SpringBootTest(classes = {BlogApplication.class}, webEnvironment =
SpringBootTest.Web
    Environment.RANDOM PORT)
    class IntegrationTest {

        @Autowired
        TestRestTemplate restTemplate;


        @BeforeAll
        void setup() {
            System.out.println(">> Setup");
        }

        @Test
        void assertBlogPageTitle_Content_And_StatusCode() {
            System.out.println(">> Assert blog page title, content and status
code");
            ResponseEntity<String> entity = restTemplate.getForEntity("/",
String.class);
            assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
            assertThat(entity.getBody()).contains("<h1>Blog</h1>", "title1");
        }

        @Test
        void assertArticlePageTitle Content And StatusCode() {
            System.out.println(">> Assert article page title, content and status
code");
            String title = "title1";
            ResponseEntity<String> entity =
restTemplate.getForEntity(String.format("/article/%s",
CommonUtil.toSlug(title)), String.class);
            assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.OK);
            assertThat(entity.getBody()).contains(title, "headline1",
"content1");
        }
```

```
    @AfterAll
    void teardown() {
        System.out.println(">>  Tear down");
    }

}
```

启动应用，跳转到地址 http://localhost:8080/。看到可单击链接的文章列表，在单击后能查看文章了吗？如果成功了的话，就说明以上用例已经精准无误地完成了。

1.5.6　发布 HTTP 接口

现在大多 Web App 都有跨平台访问的需求。为了满足这一需求，前后端分离的架构渐渐成为主流。在前后端分离的架构中，大多选用 RESTful 风格的 Web API。这里将借助注解“@RestController”来实现一个 Web API 服务，用以查询 article 以及 user 的信息。

【示例 1-5】

在路径 src/main/java/com/example/myblog/controller 下创建 ArticleController.java：

```
@RestController
@RequestMapping("/api/article")
public class ArticleController {

    private final ArticleRepository articleRepository;

    public ArticleController(ArticleRepository articleRepository) {
        this.articleRepository = articleRepository;
    }

    @GetMapping("/")
public Iterable<Article> findAll() {
        return articleRepository.findAllByOrderByAddedAtDesc();
    }

    @GetMapping("/{slug}")
    public Article findOne(@PathVariable String slug) throws
HttpStatusCodeException {
        Article result = articleRepository.findBySlug(slug);
        //当查询结果为 null，返回 404
        if (result == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "This
article dose not exit");
        }
        return result;
    }
}
```

在路径 src/main/java/com/example/myblog/controller 下创建 UserController.java：

```
@RestController
@RequestMapping("/api/user")
```

```
    public class UserController {

        private final UserRepository userRepository;

        public UserController(UserRepository userRepository) {
            this.userRepository = userRepository;
        }

        @GetMapping("/")
        public Iterable<User> findAll() {
            return userRepository.findAll();
        }

        @GetMapping("/{login}")
        public User findOne(@PathVariable String login) {
            User result = userRepository.findByLogin(login);
            //当查询结果为 null，返回 404
            if (result == null) {
                throw new ResponseStatusException(HttpStatus.NOT_FOUND, "This user
does not exit");
            }
            return result;
        }
    }
```

实现了 Rest 服务对应的控制器之后，需要编写与之对应的测试用例。对控制器层的测试有些类似于对 JPA 的测试。下文将通过 Mockito（mocking 框架）对 DAO 层进行模拟，以消除 DAO 层的返回结果对测试结果的影响，最终达到仅校验 controller 层的目的。

在路径 src/test/java/com/example/myblog 下创建 HttpControllersTests.java：

```
    @WebMvcTest
    public class HttpControllersTests {

        @Autowired
        private MockMvc mockMvc;

        @MockBean
        private UserRepository userRepository;

        @MockBean
        private ArticleRepository articleRepository;

        @Test
        void listArticles() throws Exception {
            User leili = new
User().setLogin("leili").setFirstName("lei").setLastName("li");
            Article article1 = new Article().setTitle("Spring Framework 5.0 goes
GA")
    .setHeadline("headline1").setContent("content1")
                    .setAuthor(leili);
            Article article2 = new Article().setTitle("Spring Framework 4.3 goes
GA")
    .setHeadline("headline2").setContent("content2")
                    .setAuthor(leili);
```

```
            //模拟 articleRepository.findAllByOrderByAddedAtDesc()方法。当调用该方法
时，返回 article1 与 article2 组成的列表

when(articleRepository.findAllByOrderByAddedAtDesc()).thenReturn(Arrays.asList
(article1, article2));
            //调用接口"/api/article/"
            mockMvc.perform(MockMvcRequestBuilders.get("/api/article/")
    .accept(MediaType.APPLICATION_JSON))
                    //期望调用结果的 statuscode 为 2xx
                    .andExpect(status().is2xxSuccessful())
                    //期望内容类型为 APPLICATION_ JSON
                    .andExpect(content().contentType(MediaType.APPLICATION_JSON))
                    .andExpect(jsonPath("$.[0].author.login").value(leili.getLogi
n()))
                    .andExpect(jsonPath("$.[0].slug").value(article1.getSlug()))
                    .andExpect(jsonPath("$.[1].author.login").value(leili.getLogi
n()))
                    .andExpect(jsonPath("$.[1].slug").value(article2.getSlug()));
        }
    }
```

1.6　Spring Boot 自动配置与外部配置

在之前的章节反复提及 Spring Boot 的“自动配置”，那么自动配置大概在做什么？如果程序有更多的自定义配置的需求，在 Spring Boot 开发的过程中又需要做什么？本节会对这些内容做一个基本介绍。

1.6.1　自动配置

有读者可能会问：自动配置，自动在哪？答案并不复杂。Spring Boot 的自动配置会在没有自定义配置的情况下，尝试根据现有的 jar 包依赖对程序进行配置。以 1.5 节中实现的程序为例，在该程序的实现过程中，这里并没有直接地编写配置文件就使用上了 H2 数据库。原因就在于 Pom.xml 中声明了 H2 依赖项，并且没有其他数据源被人为添加到配置文件当中。Spring Boot 在这种场景下，默认将 H2 配置为该项目的数据库。

在不做任何配置的情况下启动项目。输入以下命令以使用 Spring-Boot Maven 插件运行 Spring Boot 项目：

```
mvn spring-boot:run
```

程序成功启动之后，终端程序上将打印如下日志信息：

```
Using dialect: org.hibernate.dialect.H2Dialect
```

看到了这行日志，说明程序已自动将 H2 数据库作为默认数据源配置在项目中了。

1.6.2 外部配置

自动配置能简化开发流程，但在真实的开发情景下还是需要开发人员手动进行配置才能满足多种多样的需求。Spring Boot 提供了多种途径的外部配置来协助开发人员，以便于在不同环境能使用相同的代码。使用 properties 文件、yaml 文件、环境变量和命令行参数，结合@Value 与@ConfigurationProperties 等注解，将配置的值注入到程序当中。

为了将外部配置和自动配置配合使用，Spring Boot 为这些配置的加载指定了一个顺序：

（1）Devtools 全局配置文件，路径：~/.spring-boot-devtools.properties（当 Spring Boot Devtools 启用）。

（2）测试用例中的“@TestPropertySource”注解。

（3）测试用例中的“properties”属性。在@SpringBootTest 和测试注解上可用，用于测试应用程序的特定部分。

（4）命令行参数。

（5）环境变量“SPRING_APPLICATION_JSON”中的字段。

（6）“ServletConfig”初始化参数。

（7）“ServletContext”初始化参数。

（8）“java:comp/env”中的 JNDI 参数。

（9）Java 系统属性(System.getProperties())。

（10）系统环境变量。

（11）配置文件“RandomValuePropertySource”，该配置文件都是随机的（random.*）。

（12）打包文件外包含于特定 profile 的配置文件（application-{profile}.propertie-s/yaml）。

（13）打包文件内包含于特定 profile 的配置文件（application-{profile}.propertie-s/yaml）。

（14）打包文件外的配置文件（application.properties/yaml）。

（15）打包文件内的配置文件（application.properties/yaml）。

（16）配置在@Configuration 配置文件中的“@PropertySource”注解。

（17）默认配置。通过 SpringApplication.setDefaultProperties()配置的属性。

1.6.3 命令行配置

在上文的介绍中可以了解到，命令行配置的优先级相对来说还是比较高的。通过“--”符号声明配置项，比如--server.port=9000 可以配置服务的端口。在 IDEA 的配置项中，打开启动项配置页“Run/Debug Configuration”，在 Program arguments 栏将配置内容输入即可在启动时附上配置项，如图 1.14 所示。

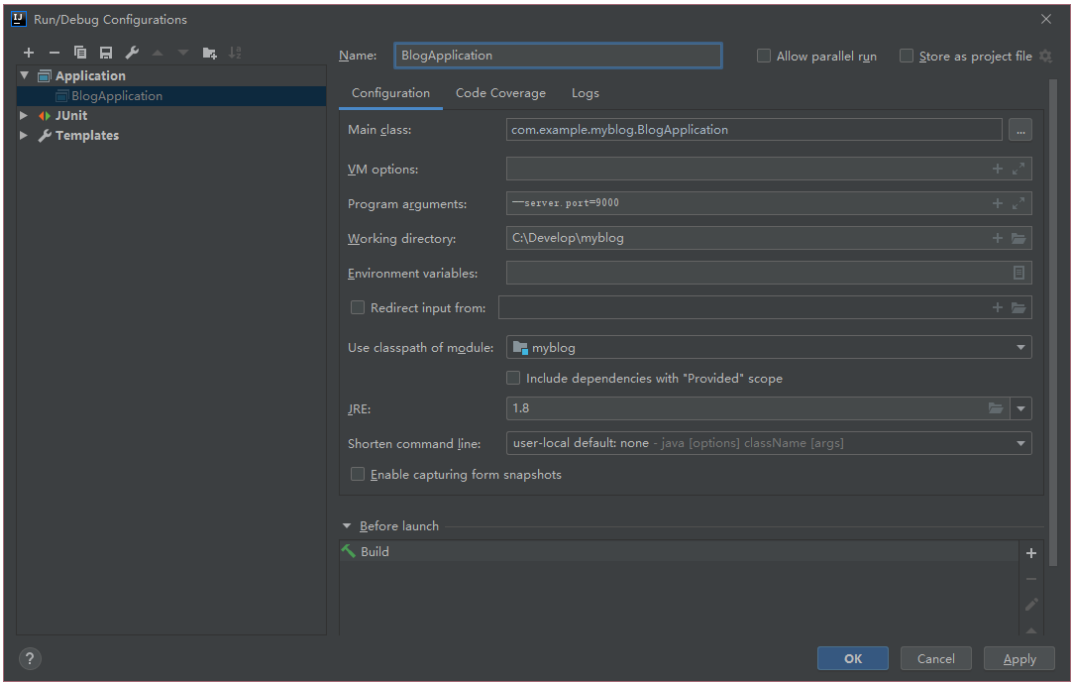

图 1.14　启动配置页

1.6.4　application.yaml/properties 配置文件

application.yaml/properties 配置文件是相对来说更常用的配置方式，配合各类注解可以满足不同需求的场景。application 配置文件支持三种格式：yaml、yml 和 properties，优先级分别为 properties→yml→yaml。在配置文件内部的优先级为自下而上进行覆盖。虽然 Spring Boot 有这样的机制，但不建议创建多个不同格式的 application 配置文件，并且配置文件内的配置项务必保持唯一，否则项目的维护将是一场灾难。

推荐使用 yaml 格式的 application.yml 文件进行配置。相较 application.properties 来说，application.yml 采用的 yaml 格式层级关系更为鲜明，使用起来更为方便。

application.properties 样例：

```
server.port=8080

logging.path=/var/logs
logging.file=myapp.log
logging.config=# 此行填入配置文件路径 (default classpath:logback.xml)
logging.level.*=# 此行填入 loggers 等级，e.g.
"logging.level.org.springframework=DEBUG" (TRACE, DEBUG, INFO, WARN, ERROR, FATAL,
OFF)
```

application.yml 样例：

```
server:
  port: 8080

logging:
```

```
    path: /var/logs
    file: myapp.log
    config:   # 此行填入配置文件路径 (default classpath:logback.xml)
    level.*:  # 此行填入 loggers 等级，e.g.
"logging.level.org.springframework=DEBUG" (TRACE, DEBUG, INFO, WARN, ERROR, FATAL,
OFF)
```

结合@ConfigurationProperties 注解可以将对程序内变量的赋值提取到配置文件中。下文把在 1.5 节实现的程序中做进一步的配置，将博客的主题以及 banner 提取到配置文件中。在路径 src/main/java/com/example/myblog/config 下创建 BlogProperties.java：

```
@ConfigurationProperties("blog")
public class BlogProperties {

    private String title;
    private Banner banner;
    //省略了字段的 getter 与 setter
}
```

在 BlogApplication 中启用该配置：

```
@SpringBootApplication
@EnableConfigurationProperties(BlogProperties.class)
public class BlogApplication {
    //…
}
```

在路径 src/main/resources 路径下删除初始化的 application.properties 文件，并创建 application.yml 以替代它：

```
blog:
  title: Blog
  banner:
    title: Surprise!
content: You made it!!!
```

修改模板以及控制器，以用上这些配置属性。blog.mustache：

```
{{> header}}

<div class="articles">

    {{#banner.title}}
        <section>
            <header class="banner">
                <h2 class="banner-title">{{banner.title}}</h2>
            </header>
            <div class="banner-content">
                {{banner.content}}
            </div>
        </section>
    {{/banner.title}}

    {{#articles}}
        <section>
            <header class="article-header">
```

```
                <h2 class="article-title"><a
href="/article/{{slug}}">{{title}}</a></h2>
                <div class="article-meta">By
<strong>{{author.firstName}}</strong>, on <strong>{{addedAt}}</strong></div>
            </header>
            <div class="article-description">
                {{headline}}
            </div>
        </section>
    {{/articles}}
</div>

{{> footer}}
```

HtmlController.java：

```
@Controller
public class HtmlController {
    //…
    @GetMapping("/")
    public String blog(Model model) {
        model.addAttribute("title", blogProperties.getTitle());
        model.addAttribute("banner", blogProperties.getBanner());
        model.addAttribute("articles", StreamSupport.stream(repository.
findAllByOrderByAddedAtDesc().spliterator(), true)
                .map(this::render)
                .collect(Collectors.toList()));
        return "blog";
    }
    //……
}
```

重启应用后，可以看到新的主页标题和内容了。

第 2 章

使用 Spring Boot 构建 Web 应用程序

在 1.5 节着手实现了第一个 Spring Boot 应用程序，大家在实现的过程中已经领略到 Spring Boot 的强大与便捷。但是，其中接触到的大部分知识因为开篇的缘故一笔带过了，本章将结合之前的示例进一步展开说明这部分内容。

本章主要涉及的知识点有：

- 数据持久化
- MVC 架构的 Web 应用
- 对多媒体数据的处理
- 拦截器与过滤器
- Spring Boot 事件
- Spring Boot 日志

2.1 实体与数据持久化

Web 应用程序终归是离不开数据的。数据在其中好比是泥土，一款 Web 应用程序构建出来的目的便是有机地组织数据，就像果树汲取了泥土的养分之后结出可口的果实。数据持久化技术自然就是绕不开的第一步。

2.1.1 数据持久化框架

数据持久化意味着应用程序可以将数据存储到非易失性的存储设施中，并能在其中执行检索操作。在早期的 Web 开发过程中，这一环节依赖 JDBC（Java DataBase Connectivity，Java 数据库

基本连接）与 RDBMS（Relational Database Management System，关系型数据库管理系统）。如今有诸多数据持久化的方案可供选择，例如：Hibernate、MyBatis、JOOQ、Ebean 以及 JdbcTemplate。下面对其中 3 个相对主流的解决方案进行对比。

（1）Hibernate：一个开源的轻量级 ORM（Object Relational Mapping，对象关系映射）框架。通过将应用程序中的实体映射到关系型数据库，以实现 JPA 的数据持久性，进一步做到与数据库进行交互。这意味着开发人员无须编写 SQL 语句，仅仅通过操作实体以及调用框架所封装的方法即可实现增删改查（CURD），大大简化了应用程序的开发。

为了提高所构建应用的性能，Hibernate 还提供了诸多额外的功能，如缓存、延迟初始化等。其中缓存相当于一个介于应用与数据库之间的存储设施，作用是提升数据的获取速度。Hibernate 支持两级缓存，分别称为一级缓存与二级缓存。因为同样的功能，在不同数据库中所使用的关键字存在一定差异，Hibernate 为了支持多种数据库提供了对应各种类型的“数据库方言”。

（2）MyBatis：一个开源的轻量级持久化框架。与 ORM 框架不同，MyBatis 并不会在 Java 对象与数据库之间建立映射，而将 Java 方法映射到 SQL 语句。这意味着开发过程中所调用的 DAO 层方法，都需要由开发人员编写 SQL 语句以及对应的 Mapper。与 ORM 框架相比，MyBatis 非常强调 SQL 语句。这导致开发过程相对烦琐，却换来很大程度的自由度，为在特殊场景下编写 SQL 语句调优提供了便利。

MyBatis 虽然不是 ORM 框架，但同样提供了映射引擎以帮助 SQL 的执行结果映射至对象。该框架允许开发人员使用所有的数据库功能，例如存储过程、视图等。MyBatis 支持声明式数据缓存，一条语句被标记为可缓存后，其查询结果将被存储至缓存当中。在查询过程的开始，首先访问缓存以查询是否有缓存数据，未命中缓存则转而查询数据库，并将结果存储到缓存中以便下次查询。

（3）JdbcTemplate：由 Spring 提供的持久化解决方案。JdbcTemplate 并非框架，而是对原生 JDBC 的封装。在封装的过程中，JdbcTemplate 解决了数个 JDBC 开发面临的问题：

- 在执行查询之前与之后，开发人员均需要编写大量的模板代码，例如创建或关闭连接。
- 对数据库逻辑执行异常处理（try/catch）。
- 手动提交事务。
- 很难实现不同数据库之间的迁移。

2.1.2　什么是实体

实体（Entity）的概念来源于 ER 模型（Entity–Relationship），由 Peter Chen 提出用于数据库设计。在一个简单的关系型数据库当中，表的每一行对应一个实体类型的实例，表中每一个字段代表实例类型的一个属性。在关系型数据库当中，实体之间的关系是通过将一个实体的主键以外键形式存储在另一实体表中来实现的。

实体在数据持久化中的作用在于使用面向对象的形式表达数据库，操作对象即操作数据库。

2.1.3 浅谈 Spring Data JPA

2.1.1 小节介绍 Hibernate 并提到了 JPA。JPA（Java Persistence API）是由 Sun 公司推出的一套 ORM 规范。Hibernate 之于 JPA 如同 JDBC Driver 之于 JDBC。Hibernate 是 JPA 的具体实现。Spring 为了支持在使用 JPA 的场景下更方便地编码，推出了 Spring Data JPA 这一项目。Spring Data JPA 是对 JPA 的封装。

【示例 2-1】

在 1.5.4 小节我们已初步接触过 Spring Data JPA。对数据库的操作正是依赖 Spring Data JPA 的 CrudRepository。CrudRepository 提供了一组与 CRUD 有关的基础方法：

```
    public interface CrudRepository<T, ID> extends Repository<T, ID> {
        //添加一条数据
        <S extends T> S save(S entity);
        //添加多条数据
        <S extends T> Iterable<S> saveAll(Iterable<S> entities);
        //根据 id 查询一条数据
        Optional<T> findById(ID id);
        //根据 id 判断是否存在
        boolean existsById(ID id);
        //查询所有的数据
        Iterable<T> findAll();
        //根据 id 查询集合
        Iterable<T> findAllById(Iterable<ID> ids);
        //查询数据的条数
        long count();
        //根据 id 删除数据
        void deleteById(ID id);
        //根据给定的数据进行删除
        void delete(T entity);
        //根据给定的集合进行删除
        void deleteAll(Iterable<? extends T> entities);
        //删除所有数据
        void deleteAll();
    }
```

使用以上提供的方法，为 1.5.4 小节中声明的 UserRepository 编写一个 DataJpaTest，以熟悉这些方法的使用：

```
    @DataJpaTest
    public class ArticleRepositoryTests {

        @Autowired
        UserRepository userRepository;

        @Test
        public void saveAUser_thenFindIt() {
            //保存一条用户数据，然后进行查询
            User leili = new
User().setLogin("leili").setFirstName("Lei").setLastName("Li");
```

```
            userRepository.save(leili);
            User found = userRepository.findById(leili.getId()).orElse(null);
            assertThat(found).isNotEqualTo(null);
            assertThat(leili).isEqualTo(found);
        }

        @Test
        public void saveUserList_thenCountThem() {
            //保存一个用户数据集合，然后查询记录的数据量
            User leili = new
User().setLogin("leili").setFirstName("Lei").setLastName("Li");
            User meimeihan = new
User().setLogin("meimeihan").setFirstName("Meimei")
                        .setLastName("Han");
            User taolin = new
User().setLogin("taolin").setFirstName("Tao").setLastName("Lin");
            User jimgreen = new
User().setLogin("jimgreen").setFirstName("Jim").setLastName("Green");
            List<User> toSave = Arrays.asList(leili, meimeihan, taolin, jimgreen);
            userRepository.saveAll(toSave);
            Long countThem = userRepository.count();
            assertThat(countThem).isEqualTo(4);
        }

        @Test
        public void saveAUser_thenDeleteIt() {
            //保存一条用户数据，然后删除它
            User leili = new
User().setLogin("leili").setFirstName("Lei").setLastName("Li");
            userRepository.save(leili);
            userRepository.delete(leili);
            User found = userRepository.findById(leili.getId()).orElse(null);
            assertThat(found).isEqualTo(null);
        }

        @Test
        public void saveUserList_thenDeleteThem() {
            //保存一个用户数据集合，然后删除所有记录
            User leili = new
User().setLogin("leili").setFirstName("Lei").setLastName("Li");
            User meimeihan = new
User().setLogin("meimeihan").setFirstName("Meimei")
                        .setLastName("Han");
            User taolin = new
User().setLogin("taolin").setFirstName("Tao").setLastName("Lin");
            User jimgreen = new
User().setLogin("jimgreen").setFirstName("Jim").setLastName("Green");
            List<User> toSave = Arrays.asList(leili, meimeihan, taolin, jimgreen);
            userRepository.saveAll(toSave);
            userRepository.deleteAll();
            Long countThem = userRepository.count();
            assertThat(countThem).isEqualTo(0);
        }
    }
```

2.1.4 使用 Lombok 简化 POJO

1.5.4 小节中定义了实体类 Article 与 User。实体类的职责在“贫血模式”（领域驱动设计中的概念）下大多很简单，即存储数据。但随着字段的增长，getter 以及 setter 的声明使得类行数变得十分膨胀。本节将推荐一款插件解决这个问题——Lombok。

（1）添加 Maven 依赖。

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.12</version>
    <scope>provided</scope>
</dependency>
```

（2）添加 IDE 对 Lombok 的支持。在工具栏中依次选择“File”→“Settings”，打开 IDEA 的设置页面。在设置页面的左侧选择“Plugins”，打开 IDEA 的插件管理页面。在搜索栏中输入“Lombok”，即可看到所需的插件，如图 2.1 所示。

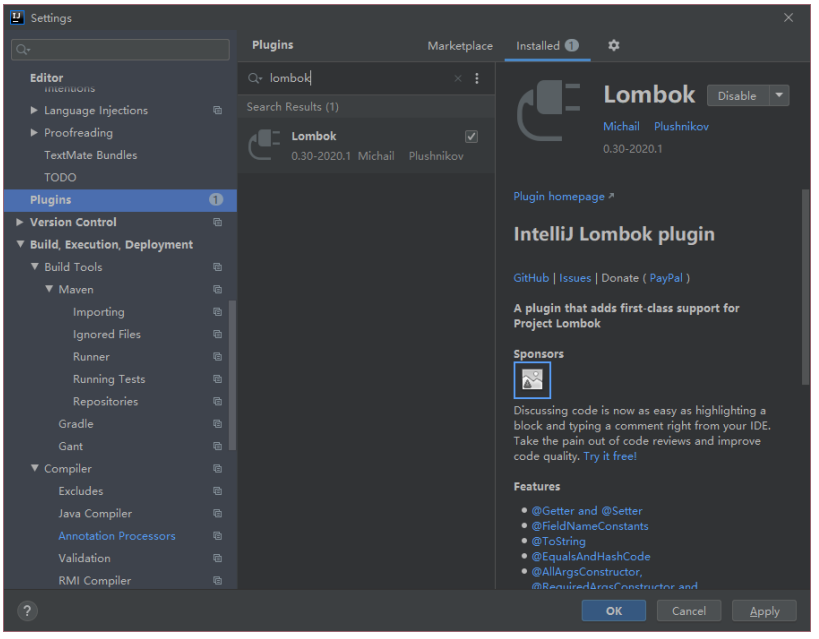

图 2.1 IDEA Lombok 插件

开启 AnnotationProcessor，使 Lombok 在编译阶段生效。操作顺序为“File”→“Settings”→“Build,Execution,Deployment”→“compiler”→“Annotation Processor”，如图 2.2 所示。

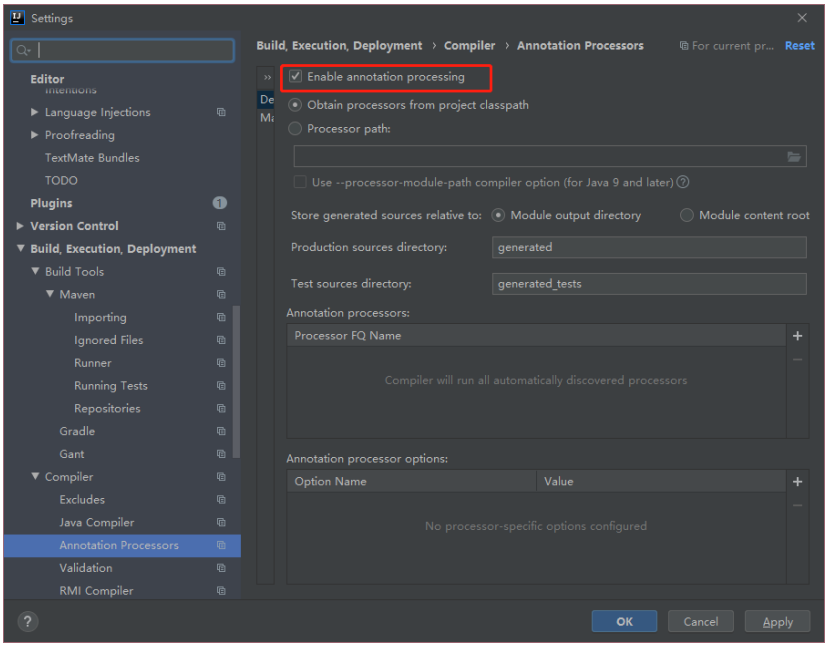

图 2.2　启用 AnnotationProcessor

（3）作用于 POJO，消除不友好的代码。使用 Lombok 注解修改 Article.java：

```
@Accessors(chain = true)
@Setter
@Getter
@Entity
public class Article {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String headline;
    private String content;
    @ManyToOne
    private User author;
    private String slug;
    private LocalDateTime addedAt;

    public Article() {
        addedAt = LocalDateTime.now();
    }

    public Article setTitle(String title) {
        this.title = title;
        this.slug = CommonUtil.toSlug(title);
        return this;
    }
}
```

@Getter/@Setter 注解作用于类上用于生成所有成员变量的 getter/setter，作用于成员变量上用于生成对应的 getter/setter。@ Accessors 用于配置 getter/setter 的生成结果，当设置属性 chain 为 true

时，setter 方法将返回当前对象。

Lombok 的功能不仅限于此，更多的特性可自行参考官网。

2.2 MVC 与模板引擎

在 1.5.1 小节实现的 HelloWorld 示例中，我们已经接触到了 MVC 与模板引擎。这是一种可以迅速构建出一个 Web 应用的技术。本节将继续介绍 MVC 的相关内容，并介绍在基于 MVC 架构的开发过程中组织代码的思路。

2.2.1 MVC 架构

MVC（Model-View-Controller）是一个很经典的软件设计模式。通常用于开发 GUI（Graphical User Interface，图形用户界面）。MVC 设计模式将程序表现层逻辑分为三个模块：Model（模型）、View（视图）以及 Controller（控制器）。这种模式便于将数据的内部表达、数据的输入以及数据的输出拆分开，使代码更易组织与维护。各个模块的职责如下：

- Model：该模块在 MVC 设计模式中的职责在于维护数据的内部表达。
- View：该模块的职责在于数据的展示。对应示例中的 mustache 模板引擎。
- Controller：该模块的职责在于控制来自于 View 的输入数据，操作 Model 并将执行结果返回用于渲染 View。

三者之间的关系如图 2.3 所示。

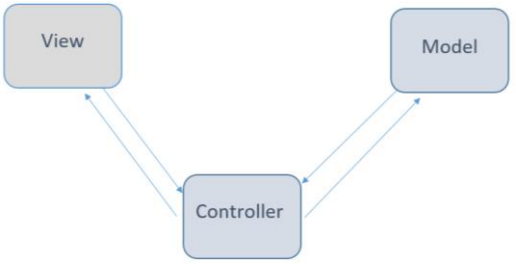

图 2.3 Model-View-Controller

使用 MVC 模式来开发软件的表现层实现了程序的高内聚与松耦合。例如 1.5.1 小节的示例中使用的 View 为 Mustache，可以在基本不修改 Model 与 Controller 的情况下，将 View 更换为 Thymeleaf 或者 Velocity。

2.2.2 Mustache 模板引擎

在之前的示例中，Mustache 已经展现出了它的威力。使用模板引擎，可以将 Web 页面的维护拆分成了动态（内容）和静态（模板）两部分。以 Mustache 的一个经典的模板举例，模板如下：

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

如果传入如下内容：

```
{
  "name": "Chris",
  "value": 10000,
  "taxed_value": 10000 - (10000 * 0.4),
  "in ca": true
}
```

那么将得到这样的结果：

```
Hello Chris
You have just won 10000 dollars!
Well, 6000.0 dollars, after taxes.
```

这样一来便可以将动态的内容插入模板，让 Web 页面根据需要生成对应的内容。

模板引擎的使用围绕标签展开，不同的标签具有不同的渲染规则。下面介绍一些常用的标签。

1. {{tag}}

该标签被称为变量（Variables），是最基本的标签类型。例如，示例“Hello {{name}}”中的{{name}}，使用该标签之后，模板引擎将会在“内容源”中找到键为“name”对应的值——“Chris”，并将其呈现至结果当中。如果“内容源”中不包含该键值对（Key Value Pair），则不会呈现任何内容。需要注意的是，变量的内容是经过 HTML 转义的。如果需要未转义的内容，则需要三重大括号{{{data}}}。示例如下：

模板：

```
{{name}}
{{age}}
{{company}}
{{{company}}}
```

内容：

```
{
  "name": "Chris",
  "company": "<b>GitHub</b>"
}
```

结果：

```
Chris

&lt;b&gt;GitHub&lt;/b&gt;
"<b>GitHub</b>
```

2. {{#tag}}/{{/tag}}

该标签由{{#tag}}与{{/tag}}两部分组成，被称为区块（Section）。其作用在于当 tag 的内容为

false、null 或者空数组时，tag 中的内容将被隐藏。该标签功能强大，传入不同类型的内容源可以实现不同效果。

（1）当 tag 对应的值为“false”的示例如下：

模板：

```
Shown.
{{#ifShow}}
  Never shown!
{{/ ifShow }}
```

内容：

```
{
  " ifShow ": false
}
```

结果：

```
Shown.
```

（2）当 tag 对应的值为“非空数组”的示例如下：

模板：

```
{{#beatles}} {{firstName}} {{lastName}} <br/>{{/beatles}}
```

内容：

```
{
  "beatles": [{
      "firstName": "John",
      "lastName": "Lennon"
    },
    {
      "firstName": "Paul",
      "lastName": "McCartney"
    },
    {
      "firstName": "George",
      "lastName": "Harrison"
    },
    {
      "firstName": "Ringo",
      "lastName": "Starr"
    }
  ]
}
```

结果：

```
John Lennon
Paul McCartney
George Harrison
Ringo Starr
```

（3）当 tag 对应的值为“Lambda”的示例如下：

模板：

```
{{#wrapped}}
  {{name}} is awesome.
{{/wrapped}}
```

内容：

```
{
  "name": "Willy",
  "wrapped": function() {
    return function(text, render) {
      return "<b>" + render(text) + "</b>"
    }
  }
}
```

结果：

```
<b>Willy is awesome.</b>
```

（4）当 tag 对应的值为“对象”的示例如下：

模板：

```
{{#person?}}
  Hi {{name}}!
{{/person?}}
```

内容：

```
{
  "person?": { "name": "Jon" }
}
```

结果：

```
Hi Jon!
```

3. {{^tag}}/{{/tag}}

该标签被称作反区块（Inverted Section），与区块的作用恰好相反。当 tag 对应的值为 false、null 或者空数组，tag 中的内容将被展示出来。示例如下：

模板：

```
{{#repo}}
  <b>{{name}}</b>
{{/repo}}
{{^repo}}
  No repos :(
{{/repo}}
```

内容：

```
{
  "repo": []
}
```

结果：

```
No repos :(
```

4. {{> tag}}

该标签被称作部分（Partial），用于调用其他以 mustache 结尾的模板。

base.mustache：

```
<h2>Names</h2>
{{#names}}
  {{> user}}
{{/names}}
```

user.mustache：

```
<strong>{{name}}</strong>
```

结果可以当作如下单个模板：

```
<h2>Names</h2>
{{#names}}
  <strong>{{name}}</strong>
{{/names}}
```

5. {{!tag}}：

该标签为注释（Comment）。这里不再举例。

6. 设置定界符

在定界符“Delimiter”中使用等号设定自定义的定界符。示例如下：

```
{{default_tags}}
{{=<% %>=}}
<% erb_style_tags %>
<%={{ }}=%>
{{ default_tags_again }}
```

示例中分别使用了“{{=<% %>=}}”与“<%={{ }}=%>”修改了定界符，避免在模板文件使用大括号引发异常。

注　意
自定义定界符不可包含空格与等号。

2.2.3　构建 MVC 架构的 Web 应用

本节将继续完善 1.5 节中所实现的应用。MVC 架构的应用往往比较注重交互，在构建应用的第一步，往往需要考虑的是页面的输入与输出。这里计划新增一个文章录入功能，需要在页面中选择作者并填写文章标题、正文等信息。

【示例 2-2】

（1）确定页面元素以及数据传输对象的结构。文章录入功能需要传输的信息有：作者名、标

题、副标题、正文。根据需求设计对应的请求对象，在路径 src/main/java/ com/example/myblog/domain 下创建 SubmitArticleQuery.java：

```
@Accessors(chain = true)
@Setter
@Getter
public class SubmitArticleQuery {
    //标题
    private String title;
    //副标题
    private String headline;
    //正文
    private String content;
    //作者名
    private String author;
}
```

（2）设计一个页面模板，满足页面输入的基本要求之外，还需要具有发起 POST 请求的功能。修改模板文件 header.mustache，引入 jquery 脚本，以便编写 ajax 请求相关的脚本：

```
<html>
<head>
    <script src="http://code.jquery.com/jquery-1.12.4.min.js"></script>
    <title>{{title}}</title>
</head>
<body>
```

在同一目录下新增用于录入文章的模板文件 writing.mustache：

```
{{> header}}

<div class="writing">
    <b><a href="/">Home</a></b>
    <form class="to-save">
        <br> title <br>
        <input type="text" name="title">
        <br> headline <br>
        <input type="text" name="headline">
        <br> content <br>
        <textarea rows="10" cols="70" name="content"></textarea>
        <br>Author:
        <select name="author">
            {{#users}}
                <option value="{{login}}">{{login}}</option>
            {{/users}}
        </select>
        <br>
        <input type="submit">
    </form>
</div>

<script type="text/javascript">
$("form").submit(function() {
    //构造请求体
    var formObject = {};
```

```
        var formArray = $("form").serializeArray();
        $.each(formArray,
        function(i, item) {
            formObject[item.name] = item.value;
        });
        //使用 AJAX，创建 POST 请求
        $.ajax({
            type: 'POST',
            url: "/article",
            data: JSON.stringify(formObject),
            contentType: 'application/json',
            success: function(data) {
                alert(data);
            }
        });
    });
    </script>

    {{> footer}}
```

（3）如此一来，MVC 中的“Model”与“View”的部分已经基本实现了。开始着手创建 MVC 最后的“Controller”。该页面分别需要两个 Controller 方法来协助实现最终效果，分别用于展示页面与接收 POST 请求。更新 HtmlController.java：

```
    @RequiredArgsConstructor
    @Controller
    public class HtmlController {

        private final ArticleRepository articleRepository;
        private final UserRepository userRepository;
        private final BlogProperties blogProperties;

        //省略若干方法

        @GetMapping("/writing")
        public String article(Model model) {
            //用于填写页面的展示
            Iterable<User> userList = userRepository.findAll();
            model.addAttribute("title", "writing");
            model.addAttribute("users", userList);
            return "writing";
        }

        @PostMapping("/article")
        @ResponseBody
        public String submitArticle(@RequestBody SubmitArticleQuery query) {
            //用于接收 POST 请求
            User author = userRepository.findByLogin(query.getAuthor());
            if (author == null) {
                //返回 400 错误码
                throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "This
user does not exist");
            }
            Article toSave = new Article().setAuthor(author)
                    .setTitle(query.getTitle())
```

```
                .setHeadline(query.getHeadline())
                .setContent(query.getContent())
                .setSlug(CommonUtil.toSlug(query.getTitle()));
        articleRepository.save(toSave);
        return "success";
    }
    }
```

submitArticle 方法仅返回类型为 String 的请求结果，而非对应名称的视图模板，因此需要在方法上方使用注解“@ResponseBody”对其进行修饰。

注　意

“Controller”为表现层（User Interface Layer）的重要一环，不应处理过多业务逻辑。在开发过程中若存在更多更复杂的代码逻辑，应“下沉”至业务逻辑层（Business Logic Layer）。更多内容请参考“三层架构”理论。

（4）测试环节，编写针对 submitArticle()的测试用例。在 IntegrationTest.java 中新增一个测试方法：

```
    @SpringBootTest(classes = {BlogApplication.class}, webEnvironment =
SpringBootTest.WebEnvironment.RANDOM PORT)
    class IntegrationTest {

        @Autowired
        TestRestTemplate restTemplate;
        //省略若干方法
        @Test
        void submitAnArticle() {
            System.out.println(">> Submit an article");
            SubmitArticleQuery queryFromAnonymous = new SubmitArticleQuery()
                    .setAuthor("anonymous")
                    .setTitle("title")
                    .setHeadline("headline")
                    .setContent("content");
            ResponseEntity<String> entity = restTemplate.postForEntity("/article",
queryFromAnonymous, String.class);
            //若查找不到"anonymous"这位作者，将返回错误码 400

assertThat(entity.getStatusCode()).isEqualTo(HttpStatus.BAD REQUEST);
            SubmitArticleQuery queryFromLeili = new SubmitArticleQuery()
                    .setAuthor("meimeihan")
                    .setTitle("title2")
                    .setHeadline("headline2")
                    .setContent("content2");
            ResponseEntity<String> entity2 =
restTemplate.postForEntity("/article", queryFromLeili, String.class);
            //正常的返回结果
            assertThat(entity2.getStatusCode()).isEqualTo(HttpStatus.OK);
            assertThat(entity2.getBody()).contains("success");
        }
    }
```

2.3 文件上传与下载

在 Web 应用中，对多媒体文件的操作非常常见，文件的上传与下载尤为如此。本节将通过在 Myblog 这个项目中新增图片以及附件的上传下载模块，带领读者熟悉该类型功能的开发流程。

2.3.1 文件上传

首先实现文件上传功能。当文件上传完毕之后对文件进行存储，最终返回对应的下载路径。

【示例 2-3】

（1）编写配置以指定文件的存储路径。在路径 src/main/java/ com/example/myblog/co-nfig 下创建 FileStorageProperties.java：

```
    @Setter
    @Getter
    @ConfigurationProperties(prefix = "file")
    public class FileStorageProperties {
        private String uploadDir;
    }
```

（2）在 BlogApplication.java 中启用该配置：

```
    @SpringBootApplication
    @EnableConfigurationProperties({BlogProperties.class,
FileStorageProperties.class})
    public class BlogApplication {

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

（3）在配置文件 application.yml 中添加对应配置信息，暂且将文件的存储路径设定为当前项目路径下的一个目录中：

```
    file:
      upload-dir: ./assets
```

（4）编写文件存储相关业务逻辑。创建用于容纳业务逻辑代码的路径 src/main/java/com/example/myblog/service，并在此路径下创建 FileStorageService.java：

```
    @Service
    public class FileStorageService {

        private final Path fileStorageLocation;

        public FileStorageService(FileStorageProperties fileStorageProperties)
throws Exception {
```

```
            this.fileStorageLocation =
Paths.get(fileStorageProperties.getUploadDir()).toAbsolutePath().normalize();
            try {
                Files.createDirectories(this.fileStorageLocation);
            } catch (IOException e) {
                e.printStackTrace();
                throw new Exception("Could not create the directory where the uploaded
files will be stored.", e);
            }
        }

        public String uploadFile(MultipartFile file) {
            //文件名
            String originalName = file.getOriginalFilename();
            //扩展名
            String extName = originalName == null || originalName.lastIndexOf(".")
<= 0 ?
                    null : originalName.substring(originalName.lastIndexOf("."));
            String fileName = UUID.randomUUID().toString() + extName;
            if (fileName.contains("..")) {
                throw new RuntimeException("Sorry! Filename contains invalid path
sequence " + fileName);
            }
            //根据文件名获得最终的 Path 对象
            Path target = fileStorageLocation.resolve(fileName);
            try {
                //将文件流复制至目标 Path
                Files.copy(file.getInputStream(), target,
StandardCopyOption.REPLACE_EXISTING);
            } catch (IOException e) {
                throw new RuntimeException("Could not store file " + fileName + ".
Please try again!", e);
            }
            return fileName;
        }

        public Resource loadFile(String fileName) {
            //TODO
            return null;
        }
    }
```

当前步骤提供了一个用于文件存储的方法。对存储路径执行初始化操作，当路径不存在时，则会创建该路径。之后通过将文件流复制至该路径以实现上传文件的需求。

（5）公布文件上传 API。逻辑在 Service 层实现之后，需要通过 Controller 层将对应的服务公开出来。在路径 src/main/java/com/example/myblog/controller 下创建 FileController.java：

```
    @RequiredArgsConstructor
    @RestController
    @RequestMapping("/file")
    public class FileController {

        private final FileStorageService fileStorageService;
```

```
        @PostMapping
        public String uploadFile(@RequestParam("file") MultipartFile file) {
            return String.format("/file/%s",
fileStorageService.uploadFile(file));
        }

        @GetMapping("/{fileName:.+}")
    public ResponseEntity<Resource> download(@PathVariable String fileName,
     HttpServletRequest request) {
            //todo
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Under
construction!");
        }
    }
```

（6）至此，文件上传的功能已基本实现。接下来将要对其做一番测试，以验证代码的准确性，在路径 src/test/java/com/example/myblog 下创建 FileUploadControllerTests.java：

```
    @RunWith(SpringRunner.class)
    @SpringBootTest
    public class FileUploadControllerTests {

        @Autowired
        private WebApplicationContext wac;

        private MockMvc mockMvc;

        @Before
        public void setup() {
            mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
        }

        @Test
        public void whenUploadFile_thenReturnAnUrl() throws Exception {
            String result = mockMvc.perform(
                   MockMvcRequestBuilders
                          .multipart("/file")
                          .file(
                                 new MockMultipartFile("file",
                                        "test.txt",
                                        ",multipart/form-data",
                                        "hello
upload".getBytes(StandardCharsets.UTF_8))
                          )
            ).andExpect(MockMvcResultMatchers.status().isOk())
                   .andReturn().getResponse().getContentAsString();
            //断言返回结果中包含路径名“file”
            assertThat(result).contains("file");
        }

    }
```

2.3.2　文件下载

在实现了文件上传功能之后，本小节将展示文件下载功能的实现过程。操作步骤如下：

【示例 2-4】

（1）在 FileStorageService.java 新增下载相关方法 loadFile，通过路径名获得文件的 Resource 对象：

```
    public Resource loadFile(String fileName) throws FileNotFoundException {
        Path filePath = fileStorageLocation.resolve(fileName).normalize();
        try {
            Resource resource = new UrlResource(filePath.toUri());
            if (resource.exists()) {
                return resource;
            } else {
                throw new FileNotFoundException("file not found" + fileName);
            }
        } catch (MalformedURLException e) {
            throw new FileNotFoundException("file not found" + fileName);
        }
    }
```

（2）在 FileController.java 补全 download 方法，获取对应文件的 Resource 对象之后，将其写入响应体中：

```
    @Slf4j
    @RequiredArgsConstructor
    @RestController
    @RequestMapping("/file")
    public class FileController {

        private final FileStorageService fileStorageService;

        @PostMapping
        public String uploadFile(@RequestParam("file") MultipartFile file) {
            return String.format("/file/%s",
fileStorageService.uploadFile(file));
        }

        @GetMapping("/{fileName:.+}")
    public ResponseEntity<Resource> download(@PathVariable String fileName,
     HttpServletRequest request) {
            Resource ret;
            try {
                //获取文件的 Resource 对象
                ret = fileStorageService.loadFile(fileName);
            } catch (FileNotFoundException e) {
                throw new ResponseStatusException(HttpStatus.NOT_FOUND, "File not
found.");
            }
       //将结果写入响应中
            return downloadFile(ret, request);
```

```
        }

        private ResponseEntity<Resource> downloadFile(Resource resource,
HttpServletRequest request) {
            String contentType = null;
            try {
                contentType =
request.getServletContext().getMimeType(resource.getFile().getAbsolutePath());
            } catch (IOException e) {
                log.error("Could not determine file type.");
            }
            if (contentType == null) {
                contentType = "application/octet-stream";
            }
            return
ResponseEntity.ok().contentType(MediaType.parseMediaType(contentType))
                    .header(HttpHeaders.CONTENT_DISPOSITION,
"attachment;filename=\"" +
     resource.getFilename())
                    .body(resource);
        }

    }
```

（3）编写测试用例。先上传一个测试文件，然后通过 API 下载该文件并比对文件内容：

```
    @RunWith(SpringRunner.class)
    @SpringBootTest
    public class FileUploadControllerTests {

        @Autowired
        private WebApplicationContext wac;

        private MockMvc mockMvc;

        @Before
        public void setup() {
            mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
        }
        //省略若干方法
        @Test
        public void uploadFile AndDownloadIt() throws Exception {
            String content = "hello upload";
            String downloadUrl = mockMvc.perform(
                    MockMvcRequestBuilders
                            .multipart("/file")
                            .file(
                                    new MockMultipartFile("file",
                                            "test.txt",
                                            ",multipart/form-data",
                                            content.getBytes(StandardCharsets.UTF 8))
                            )
            ).andExpect(MockMvcResultMatchers.status().isOk())
                    .andReturn().getResponse().getContentAsString();
            MvcResult downloadResult = mockMvc.perform(
```

```
                MockMvcRequestBuilders
                        .get(downloadUrl)
                        .contentType(MediaType.APPLICATION_OCTET_STREAM)
            ).andExpect(MockMvcResultMatchers.status().isOk()).andReturn();

assertThat(downloadResult.getResponse().getContentAsString()).isEqualTo(conten
t);
        }

    }
```

2.4　Spring Boot 日志

在软件开发过程中，偶尔发生程序没有按照预期方向执行的情况，是在所难免的。为了避免程序出现 Bug 但又不能 Debug 去定位的情况，提供丰富的日志功能就是最“通用”的选择了。好消息是 Spring Boot 集成的日志功能非常强大且容易使用，本节将讲解 Spring Boot 日志相关的内容。

2.4.1　使用预设配置

自动配置在没有特殊的需求时，日志功能是开箱即用的状态。在路径 src/main/java/com/example/myblog/controller 下创建 LogController.java：

【示例 2-5】

```
    import org.slf4j.Logger;
    import org.slf4j.LoggerFactory;
    import org.springframework.web.bind.annotation.GetMapping;
    import org.springframework.web.bind.annotation.RequestMapping;
    import org.springframework.web.bind.annotation.RestController;

    @RequestMapping("/log")
    @RestController
    public class LogController {

        Logger logger = LoggerFactory.getLogger(LogController.class);

        @GetMapping
        public String justShowSomeLog() {
            logger.trace("TRACE Message.");
            logger.debug("DEBUG Message.");
            logger.info("INFO Message.");
            logger.warn("WARN Message.");
            logger.error("ERROR Message.");
            return "Bro,go check your console.";
        }

    }
```

访问 http://localhost:8080/log 并查看控制台，将会看到有日志信息在不断输出。

2.4.2 基础配置

以上示例实现了之后，细心的读者可能会疑惑：为什么代码中编写了五条日志，但控制台中的记录只输出了三条，如图 2.4 所示。

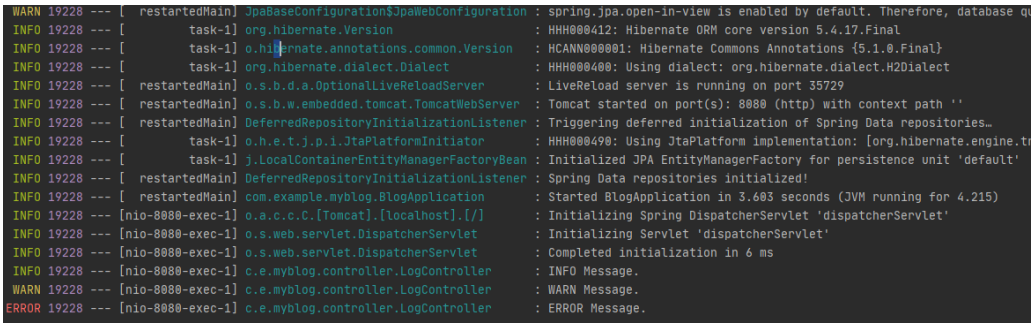

图 2.4 默认日志等级

原因在于这五条日志从上至下分别对应五种日志等级：跟踪（Trace）、调试（Debug）、信息（Info）、警告（Warn）和错误（Error）。其中 Info 为默认的日志等级，代表着仅显示 Info、Warn、Error 这三个等级的日志。如果有改变日志等级的需求，需要做一些基本的配置工作。

以将默认日志等级调整至 Debug 为例，在 application.yml 中新增一条配置：

```
logging:
  level:
    root: debug
```

这条配置的含义为：设置 root 级别，即所有包内的日志等级为 debug。重新启动项目并访问 http://localhost:8080/log，可以观察到增加了许多不同的日志信息，如图 2.5 所示。

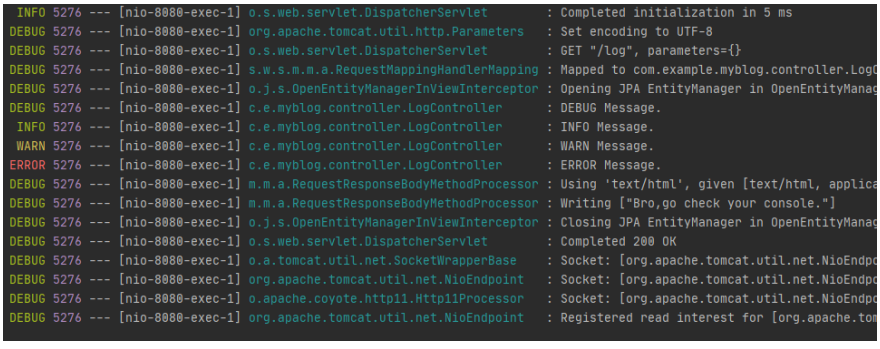

图 2.5 root 日志等级设置为 Debug

该配置项支持以包为单位的日志等级控制，这意味着可以将 Spring Boot 以及其他各种依赖的日志等级提升至 Warn，业务代码的日志等级下调至 Debug：

```
logging:
  level:
    root: warn
com.example.myblog: debug
```

结果如图 2.6 所示。

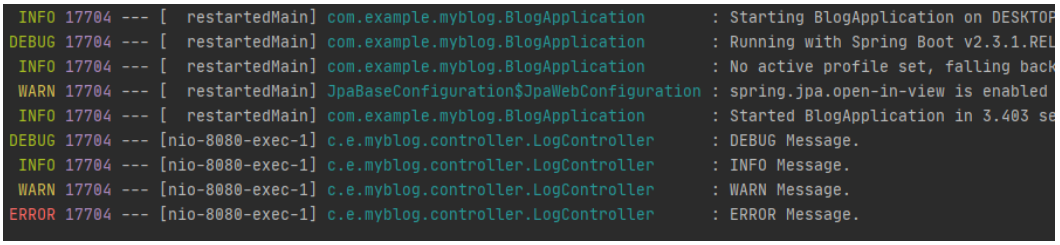

图 2.6　自定义日志等级

除了以上所介绍的 logging.level 配置之外，Spring Boot 还提供了另外一些配置项以改变日志的行为：

- logging.file：指定日志输出的目标文件。为避免用户使用中产生不必要的困惑，该属性在 2.2.x 及以上版本被废弃，转而被替换成 logging.file.name。
- logging.path：指定日志输出的目标路径。与 logging.file 一起使用时，将只有一项生效。该属性同样在 2.2.x 及以上版本被替换，替换后的属性名为 logging.file.path。
- logging.pattern.console：指定日志在控制台输出的格式。
- logging.patter.file：指定日志在文件输出的格式。

2.4.3　详细配置

尽管无配置或者在 application.yml 文件中进行配置的方式很有用，但它很可能不能满足我们的日常需求。为了应对更复杂的需求，Spring Boot 支持独立配置的方式。当 Classpath 中包含以下几种配置文件时，Spring Boot 将会默认自动加载它们：

- logback-spring.xml
- logback.xml
- logback-spring.groovy
- logback.groovy

其中使用“-spring”形式的配置文件是官方更为推荐的形式。以下是一个 logback-spring.xml 的示例：

```
    <?xml version="1.0" encoding="UTF-8"?>
    <configuration>

        <property name="LOGS" value="./logs" />

        <appender name="Console"
            class="ch.qos.logback.core.ConsoleAppender">
            <layout class="ch.qos.logback.classic.PatternLayout">
                <Pattern>
                    %black(%d{ISO8601}) %highlight(%-5level)
[%blue(%t)] %yellow(%C{1.}):
    %msg%n%throwable
```

```
            </Pattern>
         </layout>
      </appender>

      <appender name="RollingFile"
         class="ch.qos.logback.core.rolling.RollingFileAppender">
         <file>${LOGS}/spring-boot-logger.log</file>
         <encoder
            class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <Pattern>%d %p %C{1.} [%t] %m%n</Pattern>
         </encoder>

         <rollingPolicy
            class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- rollover daily and when the file reaches 10 MegaBytes -->

<fileNamePattern>${LOGS}/archived/spring-boot-logger-%d{yyyy-MM-dd}.%i.log
            </fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
               class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
               <maxFileSize>10MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
         </rollingPolicy>
      </appender>

      <!-- 设置全局日志等级为 info -->
      <root level="info">
         <appender-ref ref="RollingFile" />
         <appender-ref ref="Console" />
      </root>

      <!-- 将 com.example 包下的日志等级设置为 trace -->
      <logger name="com.example" level="trace" additivity="false">
         <appender-ref ref="RollingFile" />
         <appender-ref ref="Console" />
      </logger>

   </configuration>
```

2.4.4 Lombok 注解：@Sl4j 和@Commonslog

在编写日志相关信息时，总是需要借助 LoggerFactory 获取一个 Logger 实例。为了消除这一类模板代码，Lombok 同样提供了两个注解：@Sl4j 以及@Commonslog。使用@Sl4j 修改 LogController.java：

```
   @RequestMapping("/log")
   @Slf4j
   @RestController
   public class LogController {

      @GetMapping
      public String justShowSomeLog() {
         log.trace("TRACE Message.");
```

```
            log.debug("DEBUG Message.");
            log.info("INFO Message.");
            log.warn("WARN Message.");
            log.error("ERROR Message.");
            return "Bro,go check your console.";
        }

    }
```

使用@Commonslog 替代@Sl4j，结果相同。

2.4.5　在 Windows 平台输出彩色日志的 JANSI

分别在 Windows 平台与类 Unix 平台启动 Spring Boot 项目，可以观察到控制台上日志输出形式存在差异。在类 Unix 平台的控制台输出的日志默认是以彩色显示的，而在 Windows 平台输出的日志是以单色调显示的。原因在于类 Unix 平台默认支持 ANSI 色彩，而 Windows 却不能。因此，在 Windows 平台的控制台上需要输出彩色日志的话，需要用到 JANSI 这个库。在 pom.xml 中新增 JANSI 的依赖：

```
    <dependency>
      <groupId>org.fusesource.jansi</groupId>
      <artifactId>jansi</artifactId>
      <version>1.18</version>
    </dependency>
```

并在 logback-spring.xml 中启用它：

```
    <configuration debug="true">
        <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
            <withJansi>true</withJansi>
            <encoder>
                <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS}
[%thread] %highlight(%-5level) %green
    (%logger{15}) - %msg %n</pattern>
            </encoder>
        </appender>
        <root level="INFO">
            <appender-ref ref="STDOUT" />
        </root>
    </configuration>
```

这样一来，在 Windows 平台上运行 Spring Boot 项目，其控制台的日志输出也会以彩色的形式显示出来了。

2.5　过滤器与拦截器

在实际的开发过程中，可能会遇到这样一类需求：统计在线用户、敏感词过滤或者基于 URL 进行访问控制。这些需求有一个共同点——在每个接口被请求时都需要进行该类操作。换而言之，

如果编写了对应以上需求的代码，在每一个接口的某处都需要对这些代码进行调用。不使用一些技巧的话，这个开发过程会变得异常烦琐。本节介绍的 Filter（过滤器）与 Interceptor（拦截器）将合理解决这类需求。

2.5.1 过滤器

Filter（过滤器）这一概念来源于“Servlet 规范”，具体的功能实现由 Servlet 容器（即 Spring Boot 内容的 Tomcat）提供。过滤器的主要职责在于对资源的请求与响应的过滤，对从客户端向服务端发送的请求进行过滤，也可以对服务端返回的响应进行处理。Filter 与 Servlet 是有区别的。Filter 虽然可以对请求与响应做出处理，但其本身并不可以产生响应，如图 2.7 所示。

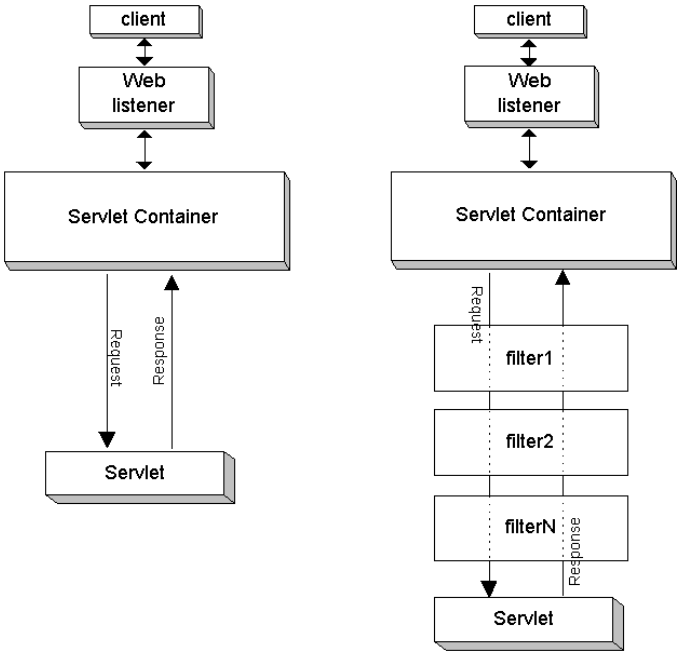

图 2.7 Filter 工作流程示意图

2.5.2 使用过滤器实现访问控制

【示例 2-6】

（1）创建一个需要实施访问控制的控制器。在路径 src/main/java/com/example/myb-log/controller 下创建 SecretController.java：

```
@RequestMapping("/secret")
@RestController
public class SecretController {

    @GetMapping
public String secret() {
    //以下代码可以替换成任意需要被保护的内容
```

```
            return "secret";
        }

    }
```

（2）创建用于身份认证的控制器。如果通过了认证，控制器将在 Cookie 中写入作为身份凭证的内容。在 controller 路径下创建 SessionController.java：

```
    @RestController
    @RequestMapping("/session")
    public class SessionController {

        @PostMapping
        public String login(@RequestBody SessionQuery sessionQuery,
HttpServletResponse response) {
            if (authenticate(sessionQuery)) {
                certificate(response);
                return "success";
            }
            //登录失败返回错误
            return "failed";
        }

        private boolean authenticate(SessionQuery sessionQuery) {
            //简单的验证逻辑，仅用作演示
            return Objects.equals(sessionQuery.getUsername(), "admin") &&
                   Objects.equals(sessionQuery.getPassword(), "password");
        }

        private void certificate(HttpServletResponse response) {
            //将登录凭证以 Cookie 的形式返回给客户端
            Cookie credential = new Cookie("sessionId", "test-token");
            response.addCookie(credential);
        }

    }
```

（3）创建用于检查凭证的过滤器。过滤器通过检查请求中附带的 Cookie 内容，以确认用户的身份。在路径 src/main/java/com/example/myblog/filter 下创建 SessionFilter.java：

```
    @Slf4j
    @WebFilter(urlPatterns = "/secret/*")
    public class SessionFilter implements Filter {
        @Override
        public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
            //读取 Cookie
            Cookie[] cookies = Optional.ofNullable(((HttpServletRequest)
servletRequest).getCookies())
                    .orElse(new Cookie[0]);
            boolean unauthorized = true;
            for (Cookie cookie : cookies) {
                if ("sessionId".equals(cookie.getName()) &&
"test-token".equals(cookie.getValue())) {
                    //验证 Cookie 中的凭证内容，如果通过验证则继续执行，否则返回 401 错误
```

```
                unauthorized = false;
            }
        }
        if (unauthorized) {
            log.error("UNAUTHORIZED");
            unauthorizedResp(servletResponse);
        }else {
            filterChain.doFilter(servletRequest, servletResponse);
        }
    }

    //向响应中写入 401 错误
    private void unauthorizedResp(ServletResponse response) throws IOException
{
        HttpServletResponse httpResponse = (HttpServletResponse) response;
        httpResponse.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
        httpResponse.setHeader("Content-type", "text/html;charset=UTF-8");
        httpResponse.setCharacterEncoding("UTF-8");
        httpResponse.getWriter().write("UNAUTHORIZED");
    }
}
```

（4）启用过滤器。需要在主类中使用@ServletComponentScan 注解，以启用被注解@WebFilter 修饰的过滤器。

```
@SpringBootApplication
@EnableConfigurationProperties({BlogProperties.class,
FileStorageProperties.class})
@ServletComponentScan(basePackages = {"com.example.myblog.filter"})
public class BlogApplication {
    public static void main(String[] args) {
        SpringApplication.run(BlogApplication.class, args);
    }
}
```

（5）分别在请求登录接口前后访问“受保护”的路径，以确认访问控制是否生效。

2.5.3 拦截器

Interceptor（拦截器）这一功能由 Spring 提供。Interceptor 与 Filter 类似，操作粒度更小，但整体功能不如 Filter 强大。Interceptor 支持自定义预处理（preHandle）可以在此过程中决定是否禁止程序继续进行，自定义后续处理（postHandle）。其处理流程如图 2.8 所示。

使用拦截器的前提是需要实现 HandlerInterceptor 接口。该接口包含三种主要方法：

- preHandle()：在执行实际的处理程序之前调用，但尚未生成视图。
- postHandle()：处理程序执行后调用。
- afterCompletion()：在请求已经响应并且视图生成完毕之后调用。

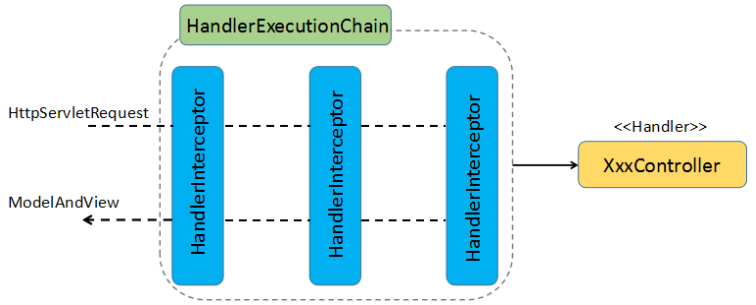

图 2.8　Interceptor 工作流程示意图

2.5.4　使用拦截器记录请求参数

首先创建一个继承 HandlerInterceptorAdapter 的拦截器类，使用日志打印请求中的参数。在路径 src/main/java/com/example/myblog/interceptor 下创建 LogRequestInterceptor.java：

【示例 2-7】

```
    @Slf4j
    @Component
    public class LogRequestInterceptor extends HandlerInterceptorAdapter {

        @Override
        public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
            log.info(String.format("[preHandle][%s][%s]%s%s", request,
request.getMethod(), request
    .getRequestURI(), getParameters(request)));
            return true;
        }

        @Override
        public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
            log.info("[postHandle]");
        }

        @Override
        public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws Exception {
            if (ex != null) {
                ex.printStackTrace();
            }
            log.info(String.format("[afterCompletion][%s][exception:%s]",
request, ex));
        }

        //提取请求中的参数
        private String getParameters(HttpServletRequest request) {
            StringBuilder parameterBuilder = new StringBuilder();
            Enumeration<String> names = request.getParameterNames();
            if (names != null) {
```

```
                parameterBuilder.append("?");
                while (names.hasMoreElements()) {
                    if (parameterBuilder.length() > 1) {
                        parameterBuilder.append("&");
                    }
                    String pointer = names.nextElement();
                    parameterBuilder.append(pointer).append("=").
append(request.getParameter(pointer));
                }
            }
            return parameterBuilder.toString();
        }

    }
```

然后将拦截器配置到 Spring 上下文。在路径 src/main/java/com/example/myblog/config 下创建 InterceptorConfig.java：

```
    @Configuration
    @RequiredArgsConstructor
    public class InterceptorConfig implements WebMvcConfigurer {

        private final LogRequestInterceptor logRequestInterceptor;

        @Override
        public void addInterceptors(InterceptorRegistry registry) {
            registry.addInterceptor(logRequestInterceptor).
addPathPatterns("/**");
        }
    }
```

最后，任意访问一个已发布过的接口。如果拦截器已生效，将会看到上文编写的日志打印到控制台当中。

2.6 Spring Boot 事件

当业务变得繁杂，模块与模块之间的耦合变得愈发严重的时候，很自然的会想要去对该部分代码进行解耦。“事件驱动”这一架构在解耦方面是一把好手。Java 中已经对事件驱动提供了支持，这种架构在 Swing 的 GUI 编程中比较常见。Spring Boot 在此基础上扩展出了自己的事件机制。

2.6.1 事件驱动模型

事件驱动模型由三个核心部分组成，如图 2.9 所示。

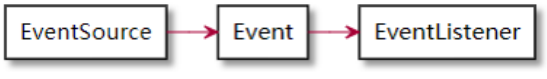

图 2.9 事件驱动模型

- EventSource（事件源）：事件发生的场所（对象）。
- Event（事件）：对事件信息的封装。将其视为一种通知会更易理解一些。
- EventListener（事件监听器）：负责监听事件（事件通知）并对其做出反应的组件。

当业务领域内有状态发生变化时，可以通过发送事件通知的方式通知其他模块。这种模式的特点在于事件源并不关心外部系统的实现，也并不期待通知的发送会带来何种的结果。换而言之，事件源相关的代码与事件监听器相关的代码没有“直接调用”的这种关系。

没有直接调用，这种方式换来了非常容易实现的“低耦合”。例如，一种爆款商品正在热卖中，这个售卖的业务中有“营销”、“售后”以及“库存管理”各种不同的部门参与其中。商品的状态由“销售中”转变为“售罄”，这一事件被发送出去并被各部门的“监听器”监听到后，便可以开始执行各个部门自有的业务逻辑。其中，这个业务若发生变化，例如有部门在业务逻辑中新增或者删除，只需修改对应的监听器即可。

不过，这样的松耦合也是一把双刃剑。因为没有直接调用，代码中缺少对流程的显式描述。如果业务流程变得复杂，整个过程会变得难以调试与修改，最终在系统中留下隐患。

2.6.2　内置事件

Spring 内置了五种标准上下文事件以及四种 ApplicationContextEvent，以便于开发人员进入应用程序以及上下文的生命周期并执行一些自定义操作。当然，即便我们很少手动使用这些事件，该框架内也会大量使用它们：

- ContextRefreshedEvent：上下文更新事件。该事件会在 ApplicationContext 被初始化或者更新时发布。也可以在调用 ConfigurableApplicationContext 接口中的 refresh()方法时被触发。
- ContextStartedEvent：上下文开始事件。当调用 ConfigurableApplicationContext 的 Start()方法开始或重启容器时触发该事件。
- ContextStoppedEvent：上下文停止事件。当调用 ConfigurableApplicationContext 的 Stop()方法停止容器时触发该事件。
- ContextClosedEvent：上下文关闭事件。当 ApplicationContext 被关闭时触发该事件。容器被关闭时，其管理的所有单例 Bean 都被销毁。
- RequestHandledEvent：请求处理事件。在 Web 应用中，当一个 http 请求结束时触发该事件。
- ApplicationStartedEvent：应用启动事件 Spring Boot 启动开始时触发该事件。
- ApplicationEnvironmentPreparedEvent：应用环境就绪事件。该事件在环境（Environment）准备就绪而上下文还没准备就绪的情况下触发。
- ApplicationPreparedEvent：应用就绪事件。该事件在上下文准备就绪而 Bean 还没加载完成时触发。
- ApplicationFailedEvent：应用异常事件。该事件在 Spring Boot 启动异常时触发。

2.6.3　监听内置事件

监听内置事件仅需要创建事件对应的监听器并注册即可，以监听 ApplicationPreparedEvent 为

例。

（1）在路径 src/main/java/com/example/myblog/event 下创建 CustomApplicationPreparedEvent Listener.java：

```
    @Slf4j
    public class CustomApplicationPreparedEventListener implements
ApplicationListener<ApplicationPreparedEvent> {

       @Override
       public void onApplicationEvent(ApplicationPreparedEvent
applicationPreparedEvent) {
         log.info("Gotya! ApplicationPreparedEvent.");
    }

    }
```

（2）注册监听器。该过程需要对 BlogApplication 进行改动：

```
    @SpringBootApplication
    @EnableConfigurationProperties({BlogProperties.class,
FileStorageProperties.class})
    @ServletComponentScan(basePackages = {"com.example.myblog.filter"})
    public class BlogApplication {

       public static void main(String[] args) {
          SpringApplication application = new
SpringApplication(BlogApplication.class);
          application.addListeners(new
CustomApplicationPreparedEventListener());
      application.run(args);
       }

    }
```

（3）修改完成之后重启应用，即可看到对应日志成功输出至控制台。

2.6.4 自定义事件

使用 Spring Boot 事件进行代码解耦，离不开自定义事件这一功能。下面来介绍自定义事件的实现过程：

（1）创建自定义事件类。在路径 src/main/java/com/example/myblog/event 下创建 Message Event.java：

```
    @Getter
    public class MessageEvent extends ApplicationEvent {

       private final String message;

       public MessageEvent(Object source, String message) {
          super(source);
          this.message = message;
```

```
        }
    }
```

（2）创建事件源。在 event 路径创建 MessageEventPublisher.java：

```
    @Slf4j
    @Component
    @RequiredArgsConstructor
    public class MessageEventPublisher {

        private final ApplicationEventPublisher applicationEventPublisher;

        public void publishAnEvent(String message) {
            log.info("Publishing an event.Message:" + message);
            //发送事件通知
            applicationEventPublisher.publishEvent(new MessageEvent(this,
message));
        }

    }
```

（3）创建事件监听器。事件监听器的创建有两种方式，使用实现 ApplicationListener<Eextends ApplicationEvent>接口的方式创建 MessageEventListener.java：

```
    @Slf4j
    @Component
    public class MessageEventListener implements
ApplicationListener<MessageEvent> {
        @Override
        public void onApplicationEvent(MessageEvent messageEvent) {
            //处理接收到事件通知后的业务逻辑
            log.info("Some business……Message:" + messageEvent.getMessage());
        }
    }
```

使用@EventListener 这一注解的方式创建 AnotherMessageEventListener.java：

```
    @Slf4j
    @Component
    public class AnotherMessageEventListener {

        @EventListener
        public void onApplicationEvent(MessageEvent messageEvent) {
            //处理接收到事件通知后的业务逻辑
            log.info("Other business……Message:" + messageEvent.getMessage());
        }
    }
```

（4）使用测试代码测试自定义事件监听相关逻辑，在路径 src/test/java/co-m/example/myblog 下创建 EventTests.java：

```
    @SpringBootTest(classes = {BlogApplication.class},
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM PORT)
    public class EventTests {

        @Autowired
```

```
    MessageEventPublisher publisher;

    @Test
    public void publishAnEvent_thenCheckConsole() {
        publisher.publishAnEvent("Bada bing,bada boom……");
    }

}
```

在测试代码启动之后，可以在控制台观察到事件相关的日志被打印到控制台。这意味着相关逻辑已成功执行。

2.6.5 异步事件

在默认情况下，事件的发布与监听是同步执行的。当要用到异步事件时，需要进行额外的配置。具体方式在于创建 ApplicationEventMulticaster 的 JavaBean。在路径 src/main/java/com/example/myblog/config 下创建 AsynchronousEventsConfig.java：

```
@Configuration
public class AsynchronousEventsConfig {
    @Bean(name = "applicationEventMulticaster")
    public ApplicationEventMulticaster simpleApplicationEventMulticaster() {
        SimpleApplicationEventMulticaster eventMulticaster = new
SimpleApplicationEventMulticaster();
        eventMulticaster.setTaskExecutor(new SimpleAsyncTaskExecutor());
        return eventMulticaster;
    }
}
```

这样一来监听器将会在单独的进程中执行，如图 2.10 所示。

```
INFO 26596 --- [           main] c.e.myblog.event.MessageEventPublisher   : Publishing an event.Message:Bada bing,bada boom…
INFO 26596 --- [TaskExecutor-38] c.e.m.event.AnotherMessageEventListener  : Other business……Message:Bada bing,bada boom……
INFO 26596 --- [TaskExecutor-39] c.e.myblog.event.MessageEventListener    : Some business……Message:Bada bing,bada boom……
```

图 2.10　监听器在不同监听器中执行

第3章

创建 RESTful Web 服务

随着移动互联网的发展，Web 开发技术的更迭，前后端分离的软件设计架构在 Web 应用开发中被广泛应用。为了能构建出更为可靠易用的服务端程序，HTTP 规范制定人之一 Dr. Roy Thomas Fielding 设计了一套 REST 规范（Representational State Transfer）。这套规范一言以蔽之，就是 URL 用以定位资源，HTTP 动词用以描述操作。后端接口的设计，就这样被安排得明明白白的。作为 Web 开发界的“扛把子”，Spring Boot 自然也提供了 REST 相关的一系列支持。接下来请跟着笔者一起来创建一个 RESTful Web 服务吧。

本章主要涉及的知识点有：

- HTTP 动词以及相关接口设计的概念
- 如何进行前后端的数据交互
- 在后端接口进行参数验证以及错误处理
- 使用 Swagger 生成方便测试与对接的接口文档

3.1 HTTP 动词

使用 REST 规范来构建后端应用的关键点，在于需要使用 URL 和 HTTP 动词来描述“调用方”与“资源”的交互。首先来认识一下常见的 HTTP 动词。

- GET：从服务端取出资源。
- POST：在服务端新建资源。
- PUT：在服务端更新资源（指客户端提供更改后完整的资源）。
- PATCH：在服务端更新资源（指客户端提供改变的属性）。
- DELETE：从服务端删除资源。

使用以上 HTTP 动词，结合合适的 URL 定义，基本上可以覆盖对于资源的各种操作。例如：

- GET /vehicle/list：获取 Vehicle 记录列表。
- GET / vehicle /{id}：根据 id 获取某 Vehicle 的信息。
- POST / vehicle：新建 Vehicle 记录。
- PUT / vehicle /{id}：整个地更新/替换某 Vehicle 的信息。
- PATCH / vehicle /{id}：修改 Vehicle 记录的某片段。
- DELTE / vehicle /{id}：删除对应的 Vehicle 记录。

现在我们可以依据这个 API 设计着手构建一个基础的 RESTful 后端应用。

3.1.1 构建一个基础的 RESTful Web 服务

在编写代码之前，需要做一下准备工作。通过 Spring Initializr（https://start.spring.io/）创建一个初始化工程，在这个页面选择项目将用到的依赖项，比如当前项目将会使用到 Spring Web、H2 Database 和 Spring Data JPA 这三项。选择完毕后在页面下方单击 Generate 按钮即可获取一个空的初始化工程，如图 3.1 所示。

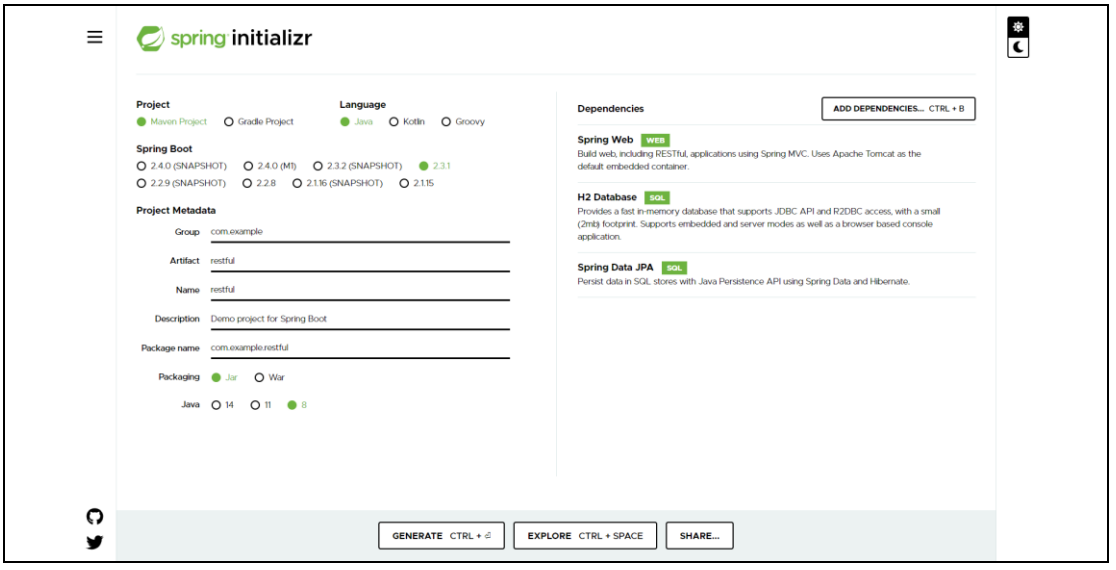

图 3.1　Spring Initializr

【示例 3-1】

打开初始化工程，开始着手编写代码。上文提到的接口被设计用于管理一些 Vehicle 记录，所以首先需要定义代表这些信息的实体类 Vehicle。

```
package com.example.restful;

import javax.persistence.*;

@Entity
@Table(name = "t_vehicle")
```

```
public class Vehicle {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String description;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Override
    public String toString() {
        return "Vehicle{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", description='" + description + '\'' +
                '}';
    }
}
```

定义好实体类之后，需要定义一个 JpaRepository 接口 VehicleRepository 用于操作数据库。Spring Data JPA 的具体使用方法会在后面的章节详细介绍。这里了解声明对应的 JpaRepository 可以做基础的 CURD 操作即可。

```
package com.example.restful;

import org.springframework.data.jpa.repository.JpaRepository;

public interface VehicleRepository extends JpaRepository<Vehicle, Long> {
}
```

声明好了实体类以及对应的 JpaRepository 类，我们可以自由地在代码中对数据库进行操作了。现在需要将这些功能根据之前的接口设计实现成 REST Web 服务。最后再实现一个 RestController。

```
    package com.example.restful;

    import org.springframework.beans.BeanUtils;
    import org.springframework.beans.BeanWrapper;
    import org.springframework.beans.BeanWrapperImpl;
    import org.springframework.http.HttpStatus;
    import org.springframework.http.ResponseEntity;
    import org.springframework.web.bind.annotation.*;

    import java.beans.FeatureDescriptor;
    import java.util.List;
    import java.util.Objects;
    import java.util.Optional;
    import java.util.stream.Collectors;
    import java.util.stream.Stream;

    @RestController
    @RequestMapping("/vehicle")
    public class VehicleController {

       private final VehicleRepository vehicleRepository;

       public VehicleController(VehicleRepository vehicleRepository) {
          this.vehicleRepository = vehicleRepository;
       }

       @GetMapping("/list")
    public ResponseEntity<List<Vehicle>> vehicleList() {
       //列出所有 Vehicle 记录
          return new ResponseEntity<>(vehicleRepository.findAll(),
HttpStatus.OK);
       }

       @GetMapping("/{id}")
    public ResponseEntity<Vehicle> selectOne(@PathVariable Long id) {
       //根据 Id 查出一条 Vehicle 记录
          return vehicleRepository.findById(id).map(v -> new ResponseEntity<>(v,
HttpStatus.OK))
                .orElse(new ResponseEntity<>(null, HttpStatus.BAD_REQUEST));
       }

       @PostMapping("/")
    public ResponseEntity<Vehicle> createOne(@RequestBody Vehicle vehicle) {
       //创建一条 Vehicle 记录
          return new ResponseEntity<>(vehicleRepository.save(vehicle),
HttpStatus.OK);
       }

       @PutMapping("/")
    public ResponseEntity<Vehicle> replaceOne(@RequestBody Vehicle vehicle) {
       //替换一条 Vehicle 记录
          Optional<Vehicle> oldOne =
```

```
vehicleRepository.findById(vehicle.getId());
           if (!oldOne.isPresent()) {
              new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
           }
           return new ResponseEntity<>(vehicleRepository.save(vehicle),
HttpStatus.OK);
       }

       @PatchMapping("/")
    public ResponseEntity<Vehicle> modifyOne(@RequestBody Vehicle vehicle) {
       //修改 Vehicle 记录
           Optional<Vehicle> findById =
vehicleRepository.findById(vehicle.getId());
           Vehicle oldOne;
           if (!findById.isPresent()) {
              return new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
           } else {
              oldOne = findById.get();
           }
           Vehicle newOne = new Vehicle();
           List<String> nullProperties = getNullProperties(oldOne);
           BeanUtils.copyProperties(newOne, oldOne, nullProperties.toArray(new
String[0]));
           return new ResponseEntity<>(vehicleRepository.save(newOne),
HttpStatus.OK);
       }

       @DeleteMapping("/{id}")
    public ResponseEntity<Vehicle> deleteOne(@PathVariable Long id) {
       //根据 Id 删除一条记录
           vehicleRepository.deleteById(id);
           return new ResponseEntity<>(null, HttpStatus.OK);
       }

    private List<String> getNullProperties(Object source) {
       //获取空属性对应的属性名
           final BeanWrapper wrappedSource = new BeanWrapperImpl(source);
           return Stream.of(wrappedSource.getPropertyDescriptors())
                  .map(FeatureDescriptor::getName)
                  .filter(propertyName ->
Objects.isNull(wrappedSource.getPropertyValue(propertyName)))
                  .collect(Collectors.toList());
       }

    }
```

每一个 RestController 类都需要被@RestController 这个注解修饰。这个注解等同于@ResponseBody+@Controller。最后呈现出的效果就是，该控制器下暴露出来的所有接口都是会自动序列化成JSON格式，并放进HttpResponse中。@GetMapping@PostMapping这些注解即对应HTTP动词。

3.1.2 是 GetMapping 吗？是 RequestMapping

从以上的例子可以观察到，通过@GetMapping、@PostMapping、@PutMapping、@PatchMapping以及@DeleteMapping 可以定义一个接口的 URL 并限定它能接受的 HTTP 方法。那它还有别的功能吗？让我们打开其中一个注解@GetMapping 的源码来一窥究竟。（使用 IDEA 的读者，可以通过Ctrl+鼠标左键选中@GetMapping 的方式打开 GetMapping 反编译后的代码。单击“Download Source”按钮即可下载并查看源码。）

【示例 3-2】

```
package org.springframework.web.bind.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.core.annotation.AliasFor;

/**
 * Annotation for mapping HTTP {@code GET} requests onto specific handler
 * methods.
 *
 * <p>Specifically, {@code @GetMapping} is a <em>composed annotation</em> that
 * acts as a shortcut for {@code @RequestMapping(method = RequestMethod.GET)}.
 *
 * @author Sam Brannen
 * @since 4.3
 * @see PostMapping
 * @see PutMapping
 * @see DeleteMapping
 * @see PatchMapping
 * @see RequestMapping
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.GET)
public @interface GetMapping {

 /**
  * Alias for {@link RequestMapping#name}.
  */
 @AliasFor(annotation = RequestMapping.class)
 String name() default "";

 /**
  * Alias for {@link RequestMapping#value}.
  */
 @AliasFor(annotation = RequestMapping.class)
```

```
    String[] value() default {};

    /**
     * Alias for {@link RequestMapping#path}.
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] path() default {};

    /**
     * Alias for {@link RequestMapping#params}.
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] params() default {};

    /**
     * Alias for {@link RequestMapping#headers}.
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] headers() default {};

    /**
     * Alias for {@link RequestMapping#consumes}.
     * @since 4.3.5
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] consumes() default {};

    /**
     * Alias for {@link RequestMapping#produces}.
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] produces() default {};

}
```

打开源码，可以看到源码中的注释。其实这些“Mapping”本质上就是 RequestMapping。可以观察到这个注解包含有非常多的属性，这里挑选几个相对实用的属性来简单介绍一下。

- path：指定接口的 URL 访问路径。
- params：指定请求中必须包含的参数。
- headers：指定请求中必须包含的请求头。
- consumes：指定请求的内容类型（Content-Type）。
- produces：指定响应的内容类型（Content-Type）。

现在请来设想一个场景。出现了一个新的需求，需要在查询到一条 Vehicle 记录之后，将这条信息复制并再次插入到数据库当中。单纯依靠 URL 以及 HTTP 动词，好像并不能很好地完成这项任务。不要担心，这时候 params 可以派上用场了。

```
    @PostMapping(path = "/{id}", params = "method")
    public ResponseEntity<Vehicle> dependOnMethod(@PathVariable Long id, String
method) {
        switch (method) {
            case "select": {
```

```
            return selectOne(id);
        }
        case "delete": {
            return deleteOne(id);
        }
        case "duplicate": {
            return duplicateOne(id);
        }
        default: {
            return new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
        }
    }
}

private ResponseEntity<Vehicle> duplicateOne(Long id) {
    //复制一条记录并写入到数据库当中
    Optional<Vehicle> findById = vehicleRepository.findById(id);
    if (!findById.isPresent()) {
        return new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
    } else {
        Vehicle oldOne = findById.get();
        Vehicle newOne = new Vehicle();
        newOne.setName(oldOne.getName());
        newOne.setDescription(oldOne.getDescription());
        return new ResponseEntity<>(vehicleRepository.save(newOne),
HttpStatus.OK);
    }
}
```

在这里扩展了一个 method 参数，通过传入不同的 method 参数实现不同的需求。当参数为“duplicate”的时候，复制对应的 Vehicle 实体中的内容，并插入到数据库中。

这样一来，一个基础的 REST 服务就构建出来了。程序成功运行之后，可以通过 curl 或者 Postman 这类工具简单测试与检验。在 3.4 节将介绍另外一个非常酷的方式来与接口交互。当然了，一个稳定可靠的 REST 服务只有这些是远远不够的。不用着急，我们在后续的章节会慢慢完善这个程序。

3.2 请求与响应

作为信息系统的一员，Web 服务的数据交互无论在开发或使用中，都是备受关注的方面。其中与用户（客户端）的数据交互大多需要依赖 HTTP 协议中的请求与响应。本节将对此展开介绍。

3.2.1 HTTP 报文

请求（Request）与响应（Response）属于 HTTP 报文的两种形式，由客户端传递至服务端就称为请求，由服务端返回客户端则称为响应。同为 HTTP 报文，它们都有如图 3.2 所示的结构。

- 起始行：用于描述请求或者响应的状态。
- Header：HTTP 头信息。
- 空行：由 CRLF 字符组成的空行，用于分隔 HTTP 头与 HTTP 报文体。
- Body：报文体，用于搭载请求或响应中的实体。

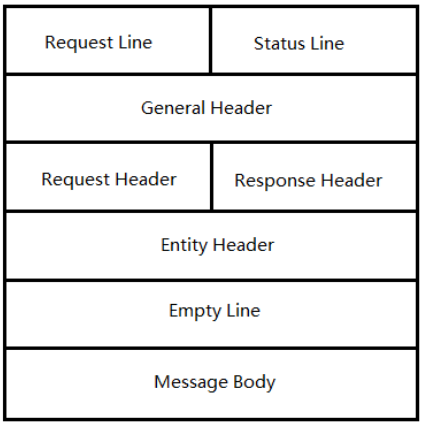

图 3.2　HTTP 报文结构

3.2.2　简单请求与@RequestParam

当请求内容相对简单时，可以考虑使用普通参数用于提取请求中的内容。代码示例如下：

【示例 3-3】

```
    @GetMapping("/no-param")
    public String noParam() {
        //无参
        return "No parameter.";
    }

    @GetMapping(path = "/single-param")
    public String singleParam(String param) {
        //单个可选参数
        return "The parameter is :" + param;
    }

    @GetMapping("/single-param-with-annotation")
    public String singleParamWithAnnotation(@RequestParam("parameter") String
param) {
        //单个必传参数
        return "The parameter is :" + param;
    }

    @PostMapping("/single-param-with-default-value")
    public String singleParamWithDefaultValue(@RequestParam(defaultValue =
"default") String param) {
        //单个必传参数
        return "The parameter is :" + param;
```

```
    }

    @PostMapping("/few-params-with-annotation")
    public String fewParamsWithAnnotation(@RequestParam String paramA,
@RequestParam(required = false) Integer paramB) {
        //paramA 必传，paramB 可选
        return String.format("paramA is :%s,paramB is :%s", paramA, paramB);
    }
```

@RequestParam 注解在该场景下可以很好地协助开发人员。虽然该注解为可选项，不使用也不会影响参数的映射，不过它提供了三个非常实用的属性：name（Web 参数名）、required（是否必传）以及 defaultValue（默认值）。使用这些属性，可以使接口定义这一开发环节变得灵活方便。

注 意

@RequestParam 的 required 属性默认为 true。这意味着如果未接收到对应参数或对应参数值为 null 的话，接口将会报错。

一个引发报错的请求示例：

```
    POST http://localhost:8080/api/simple/few-params-with-annotation HTTP/1.1
    User-Agent: PostmanRuntime/7.26.3
    Accept: */*
    Postman-Token: da5ca131-f063-444a-8195-f1433a31974e
    Host: localhost:8080
    Accept-Encoding: gzip, deflate, br
    Connection: keep-alive
    Content-Type: application/x-www-form-urlencoded
    Content-Length: 9

    paramB=13
```

响应结果：

```
    HTTP/1.1 400
    Content-Type: application/json
    Transfer-Encoding: chunked
    Date: Wed, 19 Aug 2020 15:23:19 GMT
    Connection: close

    14e3
    {"timestamp":"2020-08-19T15:23:19.337+00:00","status":400,"error":"Bad
Request"…
```

不过有一种情况是例外——使用 Optional 作为入参。修改过后的代码：

```
    @PostMapping("/few-params-with-annotation")
    public String fewParamsWithAnnotation(@RequestParam Optional<String> paramA,
@RequestParam(required = false) Integer paramB) {
        //paramA 必传，paramB 可选
        return String.format("paramA is :%s,paramB is :%s", paramA.orElse(null),
paramB);
    }
```

使用相同请求，返回不同的响应结果：

```
    HTTP/1.1 200
    Content-Type: text/plain;charset=UTF-8
    Content-Length: 29
    Date: Wed, 19 Aug 2020 15:32:32 GMT
    Keep-Alive: timeout=60
    Connection: keep-alive

    paramA is :null,paramB is :13
```

不过这并不是一个好的实现，此处仅用作简要说明。

3.2.3　使用@PathVariable 获取 URL 中的参数

根据 RESTful 风格设计出来的 API 接口，“URL 中包含查询所需的元素”这种情况也是常见的。例如“/user/{username}”，该 URL 对应的接口根据占位符中内容的不同，展示不同的用户信息。相比于另一种形式“/user?username=…”，这种处理参数的方式会让 URL 的设计更加简洁直观。要实现这种风格的接口就需要借助@PathVariable 注解。

事实上，在之前的 UserController 中已经对该注解有所接触。接下来将继续之前的示例，以展示@PathVariable 的多种使用方法。

【示例 3-4】

```
    @GetMapping("/{login}")
    public User findOne(@PathVariable String login) {
        //单个参数绑定单个入参
        User result = userRepository.findByLogin(login);
        if (result == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "This user does
not exit");
        }
        return result;
    }

    @GetMapping(value = "/{firstName}/{lastName}", params = MULTI)
    public User findOne(@PathVariable String firstName, @PathVariable String
lastName) {
        //多个参数绑定多个入参
        User result = userRepository.findByFirstNameAndLastName(firstName,
lastName);
        if (result == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "This user does
not exit");
        }
        return result;
    }

    @GetMapping(value = "/{firstName}/{lastName}", params = IN_MAP)
    public User findOne(@PathVariable Map<String, String> paramMap) {
        //多个参数绑定单个 Map
        User result =
userRepository.findByFirstNameAndLastName(paramMap.get("firstName"),
paramMap.get("lastName"));
```

```
        if (result == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "This user does
not exit");
        }
        return result;
    }
```

注 意

在两个多参数的演示代码中，@GetMapping 中的 params 属性并非必传项，仅在路径重复时作区分用。

除了以上展示的@PathVariable 使用方式之外，该注解结合正则表达式还可以实现参数过滤的功能：

```
    @GetMapping("/{login:[\\D]+}")
    public User findOne(@PathVariable String login) {
        //login 只匹配内容为非数字的参数
        //单个参数绑定单个入参
        User result = userRepository.findByLogin(login);
        if (result == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "This user does
not exit");
        }
        return result;
    }
```

3.2.4 借助@RequestHeader 读取请求头

在介绍请求体与传入参数的绑定以及 URL 中参数的获取之后，本小节将介绍如何借助@Req-uestHeader 实现对请求头内容的访问。

【示例 3-5】

（1）读取单个头属性

如果要读取某个特定的属性，可以使用对应头属性名配置@RequestHeader 的方式：

```
    @GetMapping("/greeting")
    public String greeting(@RequestHeader("accept-language") String language) {
        //根据 Header 中的不同属性返回不同问候语
        switch (language) {
            case "zh":
                return "你好";
            case "jp":
                return "こんにちは";
            case "en":
                default:
                return "Hello";
        }
    }
```

（2）将所有头属性绑定至一个 Map 实例

如果不通过声明获取特定的内容，@RequestHeader 默认获取所有头属性。具体实现方式有三种，其中通过 Map 接收参数最为常见：

```
    @GetMapping("/header-map")
    public String headerMap(@RequestHeader Map<String, String> headers) {
        //返回一个由所有头属性拼接而成的字符串
        return headers
                .entrySet()
                .stream()
                .map(entry -> String.format("key=%s,value=%s", entry.getKey(),
entry.getValue()))
                .collect(Collectors.joining("\r\n"));
    }
```

（3）将所有头属性绑定至一个 MultiValueMap 实例

当一个属性拥有多个值时，可以考虑使用 MultiValueMap 实例来接收所有头属性：

```
    @GetMapping("/multi-value-map")
    public String headerMap(@RequestHeader MultiValueMap<String, String> headers)
{
        //返回所有头属性拼接而成的字符串，使用“|”分隔有多个值的属性
        return headers
                .entrySet()
                .stream()
                .map(entry -> String.format("key=%s,value=%s", entry.getKey(),
String.join("|", entry.getValue())))
                .collect(Collectors.joining("\r\n"));
    }
```

（4）将所有头属性绑定至一个 HttpHeaders 实例

除了以上两种选择之外，还可以通过 HttpHeaders 实例的形式获取所有头属性：

```
    @GetMapping("/http-headers")
    public String httpHeaders(@RequestHeader HttpHeaders httpHeaders) {
        //获取“Accept-Encoding”属性，以逗号分隔多个值
        return String.join(",",
Optional.ofNullable(httpHeaders.get("Accept-Encoding"))
                .orElse(new ArrayList<>()));
    }
```

3.2.5　@RequestBody 与@ResponseBody

这一对注解分别代表着将请求体的内容反序列化为对应类实例（@RequestBody）以及将返回实例序列化为 JSON 字符串（@ResponseBody），在 Restful 服务的开发过程中非常常见。比如 HtmlController.java 中提交文章相关的功能就使用到了这两个注解：

【示例 3-6】

```
    @PostMapping("/article")
    @ResponseBody
    public Article submitArticle(@RequestBody SubmitArticleQuery query) {
```

```
        //用于接收 POST 请求
        User author = userRepository.findByLogin(query.getAuthor());
        if (author == null) {
            //返回 400 错误码
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "This user
does not exist");
        }
        Article toSave = new Article().setAuthor(author)
                    .setTitle(query.getTitle())
                    .setHeadline(query.getHeadline())
                    .setContent(query.getContent())
                    .setSlug(CommonUtil.toSlug(query.getTitle()));
        articleRepository.save(toSave);
        return toSave;
    }
```

返回值的默认 Content-Type 为 application/json。如果需要将其调整为 application/xml，需要引入额外的依赖：

```
    <dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
    </dependency>
```

微调对应的方法：

```
    @PostMapping(value = "/article", headers = "Accept=application/xml",
    produces = MediaType.APPLICATION_XML_VALUE)
    @ResponseBody
    public Article submitArticleAndGetXml(@RequestBody SubmitArticleQuery query)
{
        //用于接收 POST 请求
        User author = userRepository.findByLogin(query.getAuthor());
        if (author == null) {
            //返回 400 错误码
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "This user
does not exist");
        }
        Article toSave = new Article().setAuthor(author)
                    .setTitle(query.getTitle())
                    .setHeadline(query.getHeadline())
                    .setContent(query.getContent())
                    .setSlug(CommonUtil.toSlug(query.getTitle()));
        articleRepository.save(toSave);
        return toSave;
    }
```

3.2.6 使用 ResponseEntity 处理 HTTP 响应

一个 Web 服务的返回值，大多数情况下所关注的仅仅是响应体这一部分，因此响应头以及状态码都是默认状态。如果要对默认的响应头以及状态码进行修改，就需要用到 ResponseEntity 作为返回值去实现。

（1）基础使用方式：

```
    @GetMapping("/greeting")
    public ResponseEntity<String> greeting() {
        return new ResponseEntity<>("Hello there.", HttpStatus.OK);
    }
```

（2）添加自定义的 HTTP header：

```
    @GetMapping("/custom-header")
    public ResponseEntity<String> customHeader() {
        HttpHeaders headers = new HttpHeaders();
        headers.add("Custom-Header", "customHeader");
        return new ResponseEntity<>("Hello there.", headers, HttpStatus.OK);
    }
```

（3）返回不同的状态码：

```
    @GetMapping("/next-birth-day/{year}/{month}/{day}")
    public ResponseEntity<Long> nextBirthday(@PathVariable int year,@PathVariable
int month,@PathVariable int day) {
        LocalDate birthDate = LocalDate.of(year, month, day);
        if (birthDate.isAfter(LocalDate.now())) {
            return new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
        }
        LocalDate nextBirthDay = LocalDate.of(LocalDate.now().getYear(),
birthDate.getMonth(), birthDate.getDayOfMonth());
        if (nextBirthDay.isBefore(LocalDate.now())) {
            nextBirthDay = nextBirthDay.plusYears(1);
        }
        return new ResponseEntity<>(DAYS.between(LocalDate.now(),
nextBirthDay),HttpStatus.OK);
    }
```

3.3　参数验证

在构建任何程序的过程中，参数验证这一步骤都是难以避免的。比较传统的解决方式通常是在函数或者方法中编写验证相关的业务逻辑。为了将验证从业务代码中抽离，Spring 提供了一种方式——Spring Validation。本节将介绍如何借助 Spring Validation，使参数验证变得简洁通用。

3.3.1　基础验证 Bean Validation

Bean Validation 是 Spring Validation 的基础一环，是由 JCP（Java Community Process）定义的一个标准化的 JavaBean 验证 API。这个 API 提供一组注解，用以标注对应元素的验证形式，但并未提供具体实现。对应功能需要依赖相应的框架工具来实现。其提供的注解如下所示：

- @Null：被标注的元素必须为 Null。
- @NotNull：被标注的元素必须不为 Null。

- @AssertTrue：被标注的元素必须为 True。
- @AssertFalse：被标注的元素必须为 False。
- @Min(value)：被标注的元素必须是一个数字，其值必须大于等于指定的最小值。
- @Max(value)：被标注的元素必须是一个数字，其值必须小于等于指定的最大值。
- @DecimalMin(value)：被标注的元素必须是一个数字，其值必须大于等于指定的最小值。
- @DecimalMax(value)：被标注的元素必须是一个数字，其值必须小于等于指定的最大值。
- @Size(max, min)：被标注的元素的大小必须在指定的范围内。
- @Digits (integer, fraction)：被标注的元素必须是一个数字，其值必须在可接受的范围内。
- @Past：被标注的元素必须是一个过去的日期。
- @Future：被标注的元素必须是一个将来的日期。
- @Pattern(value)：被标注的元素必须符合指定的正则表达式。

3.3.2 高级验证 Spring Validation

Spring Validation 包含 Bean Validation 的实现——Hibernate Validation，并且提供了更多与 Spring 相关的功能。

【示例 3-7】

（1）为了使用到以上功能，需要引入 Spring Validation 依赖。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

（2）使用注解标注需要验证的元素。以 SubmitArticleQuery.java 为例：

```
@Accessors(chain = true)
@Setter
@Getter
public class SubmitArticleQuery {
    //标题
    @NotNull(message = "Title must not be null.")
    private String title;
    //副标题
    @NotNull(message = "Headline must not be null.")
    private String headline;
    //正文
    @NotNull(message = "Content must not be null.")
    private String content;
    //作者名
    @NotNull(message = "Author must not be null.")
    private String author;
}
```

（3）使用@Validated 标注需要验证的入参。以 HtmlController.java 为例：

```
@PostMapping(value = "/article", headers = "Accept=application/xml",
produces = MediaType.APPLICATION_XML_VALUE)
```

```
    @ResponseBody
    public Article submitArticleAndGetXml(@RequestBody @Validated
SubmitArticleQuery query) {
        return submitArticle(query);
    }

    @PostMapping("/article")
    @ResponseBody
    public Article submitArticleAndGetJson(@RequestBody @Validated
SubmitArticleQuery query) {
        return submitArticle(query);
    }

    private Article submitArticle(SubmitArticleQuery query) {
        //用于接收 POST 请求
        User author = userRepository.findByLogin(query.getAuthor());
        if (author == null) {
            //返回 400 错误码
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "This user
does not exist");
        }
        return articleService.saveArticle(query, author);
    }
```

该注解适用于程序的任意一层。在路径 src/main/java/com/example/myblog/service 下创建 ArticleService.java：

```
    @Service
    @RequiredArgsConstructor
    public class ArticleService {

        private final ArticleRepository articleRepository;

        public Article saveArticle(@Validated SubmitArticleQuery query, User
author) {
            Article toSave = new Article().setAuthor(author)
                    .setTitle(query.getTitle())
                    .setHeadline(query.getHeadline())
                    .setContent(query.getContent())
                    .setSlug(CommonUtil.toSlug(query.getTitle()));
            articleRepository.save(toSave);
            return toSave;
        }

    }
```

如此一来，一个基础的 Spring Validation 就实现了。可以观察到，使用不符合验证规则的内容进行请求，接口将会返回 400 错误以及验证相关的信息。在路径 src/main/te-st/com/example/myblog 下创建 ValidationTests.java：

```
    @RunWith(SpringRunner.class)
    @SpringBootTest
    public class ValidationTests {

        @Autowired
```

```
    private WebApplicationContext wac;

    @Autowired
    private ObjectMapper mapper;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    @Test
    public void whenSubmitWrongArgument_thenReturn4xx() throws Exception {
        SubmitArticleQuery submitArticleQuery = new SubmitArticleQuery()
                .setTitle("title")
                .setHeadline("headline")
                //设置 content 为 Null 用于测试接口验证结果
                .setContent(null)
                .setAuthor("meimeihan");
        mockMvc.perform(
                MockMvcRequestBuilders
                        .post("/article")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(mapper.writeValueAsBytes(submitArticleQuery))
        ).andExpect(MockMvcResultMatchers.status().is4xxClientError())
                .andDo(print());
    }

}
```

3.3.3 自定义校验

Spring Validation 提供了自定义校验的途径。通过实现 Validator 接口并定义对应的注解即可完成自定义校验。下面以验证 author 字段为例，实现一组自定义校验。

【示例 3-8】

（1）创建自定义注解。在路径 src/main/java/com/example/myblog/validator 下创建 Author.java：

```
@Target({FIELD})
@Retention(RUNTIME)
@Constraint(validatedBy = AuthorValidator.class)
@Documented
public @interface Author {

    String message() default "Author is not allowed.";

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

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

}
```

（2）创建 Author 所需的 Validator。在路径 src/main/java/com/example/myblog/validator 下创建 AuthorValidator.java：

```
    public class AuthorValidator implements ConstraintValidator<Author, String>
{

        private final List<String> VALID_AUTHORS = Arrays.asList("meimeihan",
"leili");

        @Override
        public boolean isValid(String value, ConstraintValidatorContext context)
{
            //判断验证是否通过的业务逻辑
            return VALID AUTHORS.contains(value);
        }
    }
```

（3）使用 Author 注解。

```
    @Accessors(chain = true)
    @Setter
    @Getter
    public class SubmitArticleQuery {
        //标题
        @NotNull(message = "Title must not be null.")
        private String title;
        //副标题
        @NotNull(message = "Headline must not be null.")
        private String headline;
        //正文
        @NotNull(message = "Content must not be null.")
        private String content;
        //作者名
        @Author
        @NotNull(message = "Author must not be null.")
        private String author;
    }
```

（4）测试自定义注解。

```
    @RunWith(SpringRunner.class)
    @SpringBootTest
    public class ValidationTests {
        //省略若干字段与方法
        @Test
        public void whenSubmitWrongAuthor thenReturn4xx() throws Exception {
            SubmitArticleQuery submitArticleQuery = new SubmitArticleQuery()
                    .setTitle("title")
                    .setHeadline("headline")
                    //设置 content 为 Null 用于测试接口验证结果
                    .setContent("content")
                    .setAuthor("anonymous");
            mockMvc.perform(
                    MockMvcRequestBuilders
                            .post("/article")
                            .contentType(MediaType.APPLICATION_JSON)
```

```
                .content(mapper.writeValueAsBytes(submitArticleQuery))
        ).andExpect(MockMvcResultMatchers.status().is4xxClientError())
                .andDo(print());
    }

}
```

3.4 错误处理

人非圣贤，孰能无过。再精明强干的程序员编写的程序也会出现错误。在 Java 中，程序出现错误会抛出“不正常信息”（Throwable）。Throwable 又被分为“错误”（Error）和“异常”（Exception）。有别于人为失误造成的“故障”（Bug），异常在程序中代表的是出现了当前代码无法处理的状况。例如：在一个对象不存在（值为 Null）的情况下，调用该对象的某个方法引发了空指针；用户输入了一段 URL，但并没有找到对应的资源；一段计算过程中，0 被当作除数……完善的错误处理，使程序不会意外崩溃甚至能友好地提示用户进行正确操作，这是让程序变得愈发健壮的重要处理步骤。图 3.3 所示为 Spring Boot 的 Whitelabel Error Page。

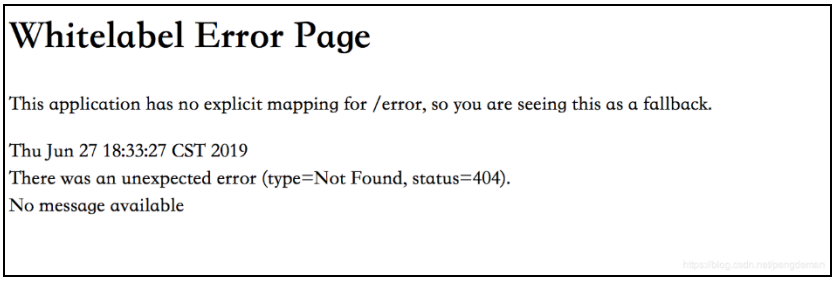

图 3.3 Whitelabel Error Page

在 Java 开发中，异常特别是检查型异常（Checked Exception），通常需要进行 try/catch 处理。而在基于 Spring Boot 的开发过程中，异常处理有了更多处理方式。

3.4.1 使用@ExceptionHandler 处理异常

首先要介绍的解决方案是使用注解@ExceptionHandler。该注解主要用于在 Controller 层面进行相同类型的异常处理。在对应 Controller 类中定义异常处理方法，并为其使用@ExcptionHandler 注解。Spring 将检测到该注解，把该方法注册为对应异常类及其子类的异常处理程序。异常处理的示例代码如下：

```
@ExceptionHandler({MyException.class})
public void handleException(MyException e) {
    //这里可以任意编写异常处理逻辑
    log.info("got an exception" + e.toString());
}
```

使用该注解的方法可以拥有非常灵活的签名，包括以下类型：

- 异常类型（Throwable）：可以选择一个大概的异常类型。例如，示例里的签名可以改为“Throwable e”或者“Exception e”，或者一个具体的异常类型。
- 请求与响应对象（Request/Response）：比如 ServletRequest/HttpServletRequest。
- InputStream/Reader：用于访问请求的内容。
- OutputStream/Writer：用户访问响应的内容。
- Model：作为从该方法返回 Model 的替代方案。

在返回类型方面也有灵活的选择：

- ModelAndView 对象。
- Model 对象，其对应视图由 RequestToViewNameTranslator 隐式确定。
- Map 对象，其对应视图同样由 RequestToViewNameTranslator 隐式确定。
- 值为某视图名的 String 对象，用于指定视图。
- 另外，在使用@ResponseBody 注解标识的情况下，将返回值使用转换器转换为最终结果。
- 使用 HttpEntity<?>或 ResponseEntity<?>对象包装的结果。
- 使用 void 作为返回类型，然后用签名中的 Response、OutputStream 或者 Writer 编写响应内容。

3.4.2　使用 HandlerExceptionResolver 处理异常

@ExceptionHandler 功能足够强大，但在不进行特殊处理的前提之下只能处理单个 Controller 的异常。面对多个 Controller 抛出的异常，还需要依赖 HandlerExceptionResolver 这一手段进行处理。使用 HandlerExceptionResolver 可以解决应用程序内的任何异常，并且依赖它可以实现 RESTful 服务的统一异常处理机制。

HandlerExceptionResolver 是一个公共接口。常见的使用方式是实现一个自定义的处理类。在自定义处理类之前，可以了解一下现有的部分实现：

- ExceptionHandlerExceptionResolver：该处理类就是帮助@ExceptionHandler 生效的组件。
- DefaultHandlerExceptionResolver：用于将标注的 Spring 异常解析为对应的 HTTP 状态码。
- ResponseStatusExceptionResolver：与注解@ResponseStatus 一起使用，将自定义异常与 HTTP 状态码进行对应，示例代码如下：

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class MyException extends Exception{

   public MyException() {
   }

   public MyException(String message) {
      super(message);
   }
}
```

之所以需要自定义处理类，原因在于以上实现无法控制响应体的内容。而大多数情况下，REST 服务的响应都需要有 JSON 或者 XML 格式的响应内容。

自定义处理类的示例代码如下：

```
    @Component
    @Slf4j
    public class RestResponseStatusExceptionResolver extends
AbstractHandlerExceptionResolver {

        @Override
        protected ModelAndView doResolveException(
                HttpServletRequest request,
                HttpServletResponse response,
                Object handler,
                Exception ex) {
            try {
                if (ex instanceof IllegalArgumentException) {
                    return handleIllegalArgument((IllegalArgumentException) ex,
response, request);
                }
                //异常处理逻辑
            } catch (Exception handlerException) {
                log.warn("Handling of [" + ex.getClass().getName() + "]resulted in
Exception", handlerException);
            }
            return null;
        }

        private ModelAndView
        handleIllegalArgument(IllegalArgumentException ex, HttpServletResponse
response, HttpServletRequest request)
                throws IOException {
            response.sendError(HttpServletResponse.SC_CONFLICT);
            String accept = request.getHeader(HttpHeaders.ACCEPT);
            //处理响应内容
            return new ModelAndView();
        }
    }
```

3.4.3 使用@ControllerAdvice 处理异常

在 Spring 3.2 版本引入了@ControllerAdvice 这一注解，为全局的@ExceptionHandler 提供了支持。将这个注解批注在一个处理类上，即可让该类下由@ExceptionHandler 批注的方法在全局层面对异常进行处理。

还记得上一小节参数验证失败的结果吗？请求中包含不符合要求的内容将抛出异常 MethodArgumentNotValidException。默认的响应内容如下：

```
    {
        "timestamp": "2020-08-27T14:24:38.927+00:00",
        "status": 400,
        "error": "Bad Request",
        "trace":
"org.springframework.web.bind.MethodArgumentNotValidException: ……",
        "message": "Validation failed for object='submitArticleQuery'. Error count:
```

```
1",
        "errors": [
            {
                "codes": [
                    "NotNull.submitArticleQuery.content",
                    "NotNull.content",
                    "NotNull.java.lang.String",
                    "NotNull"
                ],
                "arguments": [
                    {
                        "codes": [
                            "submitArticleQuery.content",
                            "content"
                        ],
                        "arguments": null,
                        "defaultMessage": "content",
                        "code": "content"
                    }
                ],
                "defaultMessage": "Content must not be null.",
                "objectName": "submitArticleQuery",
                "field": "content",
                "rejectedValue": null,
                "bindingFailure": false,
                "code": "NotNull"
            }
        ],
        "path": "/article"
    }
```

其中 trace 属性将会输出大段落的堆栈信息，在此处做了省略处理。对于调用方而言，返回的信息或许存在冗余。可以根据需求使用该方案对其进行调整。

在路径 src/main/java/com/example/myblog/controller 下创建 MyBlogControllerAdvice.java：

```
    @ControllerAdvice
    @Slf4j
    public class MyBlogControllerAdvice {
        @ResponseBody
        @ExceptionHandler(value = MethodArgumentNotValidException.class)
        public Result<String> errorHandler(MethodArgumentNotValidException e) {
            String errorMsg =
e.getBindingResult().getAllErrors().get(0).getDefaultMessage();
            log.error("未处理异常" + errorMsg);
            return new Result<String>()
                    .setMessage("参数错误：" + errorMsg);
        }
    }
```

响应结果如下：

```
    {
        "code": 0,
        "data": null,
        "message": "参数错误：Content must not be null."
```

```
    }
```

3.4.4 抛出 ResponseStatusException 异常

上文介绍的方法大多适用于解决一个切面上的问题。如果仅需要针对少数接口进行异常处理，控制其返回给客户端的 HTTP 状态码、错误指引以及报错原因，那么 ResponseStatusException 将会是一个不错的选择。示例代码如下：

```
    @GetMapping(value = "/{id}")
    public Foo findById(@PathVariable("id") Long id, HttpServletResponse response)
{
        try {
            Foo resourceById = RestPreconditions.checkFound(service.findOne(id));
            eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
response));
            return resourceById;
         }
        catch (MyResourceNotFoundException exc) {
            throw new ResponseStatusException(
              HttpStatus.NOT_FOUND, "Foo Not Found", exc);
        }
    }
```

3.5 Swagger 文档

在前后端分离的软件架构模式下，前后端程序的开发人员大多不是同一个人。这时，接口文档的重要性便体现出来了。接口文档在项目初期帮助前端开发人员快速理解接口功能，在项目后期方便维护人员对服务进行查看与维护。一份内容详实的接口文档，可以为开发带来巨大的便利，相应地也要耗费不少心血，毕竟单是为了保持接口与文档的版本一致，所付出的努力都是难以忽视的。

相信不少前后端开发工程师都或多或少被接口文档“折磨”过。前端开发抱怨文档不够友好，后端开发烦恼于文档工作过于耗时、耗力。不过在建立了规范并将流程自动化之后，接口文档将不再是难题。本节将介绍如何在 Spring Boot 中集成一款自动生产 API 文档的工具——Swagger。

3.5.1 Swagger/OpenAPI 规范

Swagger 是一个开源项目，主要用于 RESTful API 的描述与调试。集成了一组 HTML、JavaScript 和 CSS 前端资源，从符合 Swagger 规范的 API 中动态生成可以交互的接口文档。在过去几年中，Swagger2 已经成为了定义或记录 API 的一种规范。之后，该规范被移至 Linux 基金会，并重新命名为 OpenAPI 规范。下文对 Swagger/OpenAPI 的描述，事实上指代的是同一事物。

为了在项目中整合 Swagger，首先需要添加 Swagger 的 starter 依赖。该依赖由 Spring 社区内一个非官方组织 Springfox 所维护，使用该 starter 可以方便地整合 Swagger。依赖配置如下：

```
    <dependency>
```

```
        <groupId>io.springfox</groupId>
        <artifactId>springfox-boot-starter</artifactId>
        <version>3.0.0</version>
    </dependency>
```

3.5.2　生成接口文档

接口文档的生成，依赖于 Docket。为了生成一份接口文档，需要在 Spring Boot 程序中配置一个 Docket 的 Java Bean。得益于 Swagger 的合理设计，需要配置的内容并不复杂。在路径 /src/main/java/com/example/myblog/config 下创建 SwaggerConfig.java：

【示例 3-9】

```
    @EnableOpenApi
    @Configuration
    public class SwaggerConfig {
        @Bean
        public Docket createRestApi() {
            return new Docket(DocumentationType.OAS_30)
                    .apiInfo(apiInfo())
                    .select()
                    //使用@ApiOperaion 的 Controller 将被添加至接口文档当中
                    .apis(RequestHandlerSelectors.withMethodAnnotation(ApiOperati
on.class))
                    .paths(PathSelectors.any())
                    .build();
        }

        private ApiInfo apiInfo() {
            return new ApiInfoBuilder()
                    .title("Swagger3 接口文档")
                    .description("Swagger 整合示例")
                    .version("1.0")
                    .build();
        }
    }
```

创建好 Swagger 配置之后，可以分别在路径 http://localhost:8080/v2/api-docs 与 http://localhost:8080/v3/api-docs 下访问到 Swagger2 规范的接口文档与 OpenAPI3 的接口文档。

http://localhost:8080/v2/api-docs 初始内容：

```
    {
      "swagger": "2.0",
      "info": {
        "description": "Swagger 整合示例",
        "version": "1.0",
        "title": "Swagger3 接口文档"
      },
      "host": "localhost:8080",
      "basePath": "/"
    }
```

http://localhost:8080/v3/api-docs 初始内容：

```
{
  "openapi": "3.0.3",
  "info": {
    "title": "Swagger3 接口文档",
    "description": "Swagger 整合示例",
    "version": "1.0"
  },
  "servers": [
    {
      "url": "http://localhost:8080",
      "description": "Inferred Url"
    }
  ],
  "components": {
  }
}
```

另外，在 http://localhost:8080/swagger-ui/index.html 下可以访问到用来与后端交互的接口文档界面。不过文档暂时还是空空如也，等待开发人员通过后面的操作将文档渐渐丰富起来。Swagger3 的接口文档界面如图 3.4 所示。

图 3.4　Swagger3 的接口文档界面

3.5.3　使用注解生成文档内容

无论是 api-doc 还是 swagger-ui，这两者的内容都依赖于开发者在代码中通过 Swagger 提供的注解进行完善。

（1）@Api：用于 Controller 类上，将该类标记为 Swagger 的资源。常用参数如下：

- tags：说明该类的作用，参数类型为 String 数组。

（2）@ApiOperation：用于接口方法上，描述针对特定路径下的操作。常用参数如下：

- value：方法的用途和作用。
- notes：方法的注意事项和备注。

（3）@ApiModel：用于实体类上，描述实体作用。常用参数如下：

- description：描述实体的作用。

（4）@ApiModelProperty：用于实体属性上，描述实体的属性。常用参数如下：

- value：对属性的简要描述。
- name：属性名。
- required：参数是否是必选的。

（5）@ApiImplicitParam：用于方法，描述隐含的参数。常用参数如下：

- name：参数名。
- value：参数说明。
- dataType：数据类型。
- paramType：用于描述参数的类型（参数所处位置）。可选内容包括：path、query、body、header、form。

（6）@ApiImplicitParams：用于方法上，包含多个@ApiImplicitParam。

（7）@ApiParam：用于方法、参数，用于描述请求的要求和说明。常用参数如下：

- name：参数名。
- value：对参数的简要描述。
- defaultValue：参数默认值。
- required：参数是否是必选的。

（8）@ApiResponse：用于请求的方法上，描述不同的响应。常用参数如下：

- code：表示响应的状态码。
- message：描述状态码对应的响应信息。

（9）@ ApiResponses：用于方法上，包含多个@ ApiResponse。

示例代码 SimpleRestController.java：

```
@Api(tags = "RESTful 服务传参示例")
@RestController
@RequestMapping("/api/simple")
public class SimpleRestController {

    @ApiOperation(value = "无参 GET 请求", notes = "用于演示通过无参 GET 请求的形式对接口进行请求")
    @GetMapping("/no-param")
    public Result<String> noParam() {
        //无参
        return Result.ok("No parameter.");
    }

    @ApiOperation(value = "单个参数的 GET 请求", notes = "用于演示通过单参数 GET 请求的形式对接口进行请求")
    @ApiImplicitParam(name = "implicit", value = "提供隐含参数的输入方式")
    @GetMapping(path = "/single-param", params = "implicit")
    public Result<String> singleParam(@ApiParam(name = "单参数") String param) {
        //单个可选参数
        return Result.ok("The parameter is :" + param);
    }
```

```
        @ApiOperation(value = "下一个生日", notes = "输入出生年、月、日，计算到下一个生
日的天数")
        @ApiResponses({
                @ApiResponse(code = 400, message = "输入日期大于当前日期"),
                @ApiResponse(code = 200, message = "成功")
        })
        @GetMapping("/next-birth-day/{year}/{month}/{day}")
        public ResponseEntity<Result<Long>> nextBirthday(@PathVariable int year,
@PathVariable int month, @PathVariable int day) {
            LocalDate birthDate = LocalDate.of(year, month, day);
            if (birthDate.isAfter(LocalDate.now())) {
                return new ResponseEntity<>(null, HttpStatus.BAD_REQUEST);
            }
            LocalDate nextBirthDay = LocalDate.of(LocalDate.now().getYear(),
birthDate.getMonth(), birthDate.getDayOfMonth());
            if (nextBirthDay.isBefore(LocalDate.now())) {
                nextBirthDay = nextBirthDay.plusYears(1);
            }
            return new ResponseEntity<>(Result.ok(DAYS.between(LocalDate.now(),
nextBirthDay)), HttpStatus.OK);
        }
        //省略若干方法……
    }
```

示例代码 Result.java：

```
    @ApiModel(description = "用于统一 RESTful 服务返回内容")
    @Accessors(chain = true)
    @Setter
    @Getter
    public class Result<T> {
        @ApiModelProperty("业务状态码")
        private int code;
        @ApiModelProperty("响应内容")
        private T data;
        @ApiModelProperty("错误信息")
        private String message;

        private final static int SUCCESS = 0;
        private final static int FAIL = -1;

        public static <T> Result<T> ok(T data) {
            return new Result<T>()
                    .setCode(SUCCESS)
                    .setData(data);
        }

        public static <T> Result<T> failed(String message) {
            return new Result<T>()
                    .setCode(FAIL)
                    .setMessage(message);
        }
    }
```