第 1 章

Spring Boot基础

在Java开发的世界中，Spring框架以其强大的功能和灵活性成为构建企业级应用的基石。随着技术的发展，Spring Boot作为Spring家族的新成员，以其简化配置和快速开发的特点，引领Java应用开发的新趋势。如今，Spring Boot已经在蓬勃发展的快速应用开发领域（Rapid Application Development）成为领导者。本章将带你从Spring基础平滑过渡到第一个Spring Boot应用的世界。

1.1 Spring Boot简介

Spring Boot是由Pivotal团队提供的框架，是Spring家族的重要成员之一。使用Spring Boot可以做到专注于Spring应用的开发，而无须过多关注XML的配置。Spring Boot使用“习惯优于配置”的理念，简单来说，它提供了一堆依赖打包，并已经按照使用习惯解决了依赖问题。使用Spring Boot可以不用或者只需要很少的Spring配置，就可以让企业项目快速运行起来，因此大大地简化了应用的开发和部署工作。

以下是Spring Boot的核心功能：

（1）独立运行的Spring应用：Spring Boot允许应用以JAR格式独立运行，无须依赖外部的Web服务器。

（2）内嵌Servlet容器：Spring Boot支持内嵌Tomcat、Jetty或Undertow等Servlet容器，使得应用无须打包成WAR文件即可部署。

（3）简化的Maven配置：Spring Boot提供推荐的POM文件模板，以简化Maven项目的配置工作。

（4）自动配置Spring：Spring Boot能够根据项目中的依赖自动配置Spring，大幅减少手动配置的需求。

（5）生产级别的特性：Spring Boot提供了多种生产环境中所需的特性，包括性能监控、应用信息展示和健康检查等。

（6）无须代码生成和XML配置：Spring Boot完全摒弃了代码生成和XML配置，所有配置都可以通过Java注解和属性文件来实现。

总之，Spring Boot的主要目标是：

- 简化配置：通过自动配置和起步依赖（Starters）减少手动配置。
- 独立运行：允许应用打包成单个JAR文件，不需要外部服务器。
- 快速开发：提供快速的反馈循环，加速开发过程。

1.2　Spring Boot的特点和优势

从早期的Java EE到Spring，再到如今的Spring Boot，开发工作得到最大程度的简化。早期的Java EE开发配置是相当复杂的，需要大量的XML配置，甚至过度工程化，开发效率极慢；后续的Spring简化了开发流程，使得依赖注入更加简单。虽然Spring极大地简化了企业Java Web应用的开发，但随着时间的推移和Spring生态系统的扩展，配置和启动一个Spring项目变得越来越复杂，特别是XML配置、依赖管理和各种与特定模块相关的设置使得初学者很难上手。随着敏捷开发和微服务架构的兴起，开发者需要更快、更简便的方法来开发、部署和扩展其应用程序。

为了简化基于Spring的应用程序的创建和开发过程，Spring Boot应运而生。它是Spring生态系统中的一个项目，提供了一系列工具和功能，使开发者能够更轻松地开发、测试和部署Spring应用。其主要特点如下：

1）自动配置

传统的Spring应用通常需要进行大量的配置工作，而Spring Boot遵循“约定优于配置”的原则，通过提供合理的默认设置、自动配置以及简化的属性配置，极大地减少了配置需求。这意味着开发者可以专注于业务逻辑，而不是深陷于烦琐的配置之中。

2）独立运行

Spring Boot应用能够以独立的JAR文件形式运行，无须将应用打包成WAR文件并部署到外部的Servlet容器中。Spring Boot支持内嵌Tomcat、Jetty等服务器，这使得应用部署变得异常简单，开发者可以直接运行JAR文件，而无须额外的部署步骤。

3）生产级应用监控

Spring Boot提供了一套生产级别的服务监控方案，包括安全监控、应用监控以及健康监测等功能。这些工具和特性使得开发者能够更好地监控和管理他们的应用，确保应用在生产环境中的稳定性和可靠性。

4）无代码生成和 XML 配置

不需要编写大量的XML配置文件，也不需要生成代码，只需通过注解和配置文件即可完成配置。

1.3　搭建Spring Boot开发环境

本节将介绍在Windows平台搭建Spring Boot开发环境的步骤，包括安装配置JDK、安装配置Maven以及集成开发工具IDEA的使用方法。

1.3.1　安装配置Java

Spring Boot 2.7是最后一个支持JDK 8的版本。根据官方公告，Spring Boot 2.7.x的维护已于2023年11月结束。因此，未来能够获得官方免费维护的版本只有Spring Boot 3.0及以上。由于Spring Boot 3.5要求Java 17作为最低版本（本书将使用Spring Boot 3.5进行讲解），因此需要安装JDK 17或更高版本来运行这些应用。这里我们统一使用Java 17作为运行环境。

接下来，我们详细介绍在Windows 10平台上安装Java 17的步骤。

01 下载Java 17：访问Oracle官方网站下载页面，根据系统类型选择合适的ZIP文件进行下载，如图1.1所示。

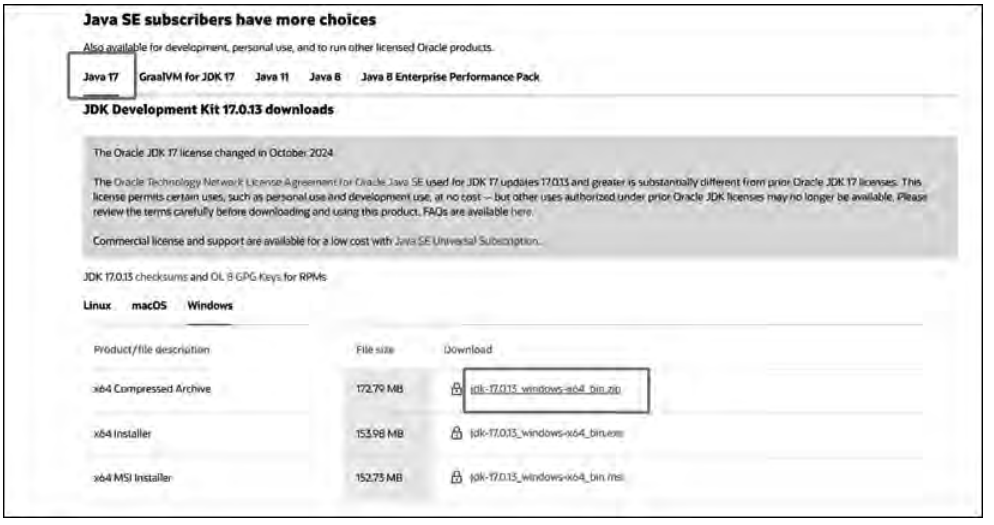

图1.1　Oracle下载页面

02 解压下载文件：将jdk-17.0.13-windows-x64_bin.zip解压到系统的任意文件夹中，这里解压到D:\tools\jdk-17.0.13_ windows-x64_bin目录。

03 在任务栏的搜索栏中搜索“系统环境变量”，然后选择“编辑系统环境变量”，打开“系统属性”，单击“环境变量”按钮，如图1.2所示。

04 在“系统变量”中单击“新建”按钮，在弹出的“新建系统变量”中将“变量名”设置为JAVA_HOME，“变量值”设置为JDK的安装路径，笔者这里为D:\tools\jdk-17.0.13 windows-x64_bin\jdk-17.0.13，读者可根据自己的安装路径进行操作，如图1.3所示。设置完成后单击“确定”按钮。

05 在“系统变量”中找到并选择Path变量，然后单击“编辑”按钮进行编辑，在“编辑环境变量”窗口中，单击“新建”按钮并添加%JAVA_HOME%\bin，再单击“确定”按钮保存更改，如图1.4所示。

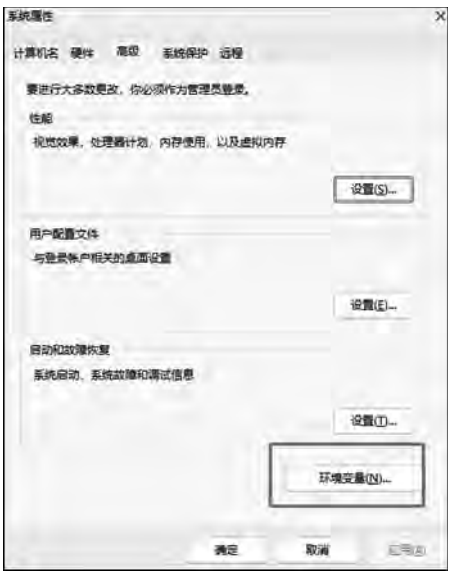

图1.2　环境变量配置

图 1.3　新建系统变量

图 1.4　编辑环境变量

06 验证安装。打开运行窗口，输入cmd命令即可打开一个新的“命令提示符”窗口。在该窗口中输入java -version命令并按回车键，即可看到已安装的Java17的版本信息，如图1.5所示。

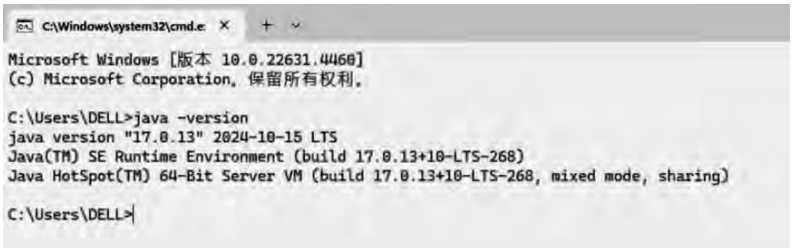

图1.5　验证安装

顺利完成以上步骤后，就已成功在Windows 10上安装了Java 17。

1.3.2　安装Maven构建工具

Apache Maven是一个流行的Java项目管理和构建工具，本书中的所有源码均使用Maven作为项目依赖管理工具。本节将讲解Maven的安装和配置，首先确保已经在系统上安装了Java Development Kit(JDK)。

1. 安装Maven

01 下载Maven。访问Apache Maven官方下载页面https://archive.apache.org/dist/maven/maven-3/3.8.1/binaries，binaries表示可执行版本，即已经编译好可以直接使用。source是源代码版本，需要自己编译成可执行软件才可使用。这里下载可执行版本apache-maven-3.8.1-bin.zip，如图1.6所示。

02 解压下载的Maven文件到本地系统的任意目录。

03 按照1.3.1节配置Java环境变量的步骤，在“系统变量”中单击“新建”按钮，在弹出的“新建系统变量”对话框中，将“变量名”设置为MAVEN_HOME，“变量值”设置为D:\tools\apache-maven-3.8.1，实际变量值以读者自己的安装路径为准，最后单击“确定”按钮，如图1.7所示。

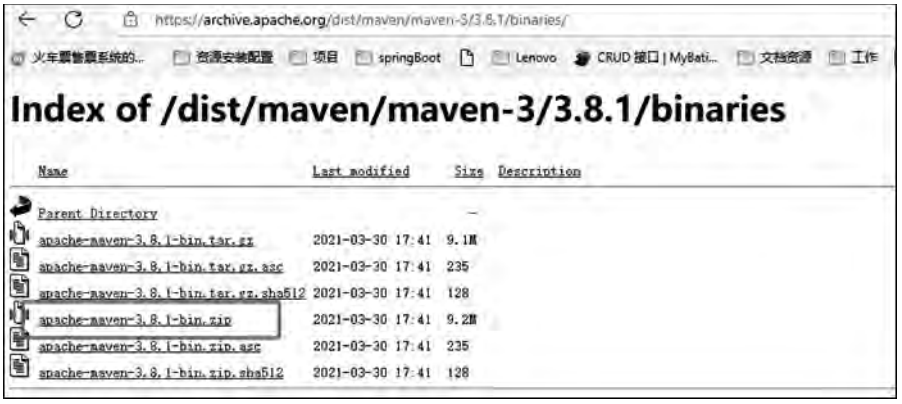

图1.6　Maven下载

04 在“系统变量”中找到并选择Path变量，然后在“编辑环境变量”窗口中单击“新建”按钮，添加%MAVEN_HOME%\bin，如图1.8所示。

图 1.7　新建系统变量

图 1.8　编辑环境变量

05 验证安装。在运行窗口输入cmd命令打开一个新的“命令提示符”窗口，输入mvn -v命令并按回车键，即可看到已安装的Maven的版本信息以及配置的JDK信息，如图1.9所示。

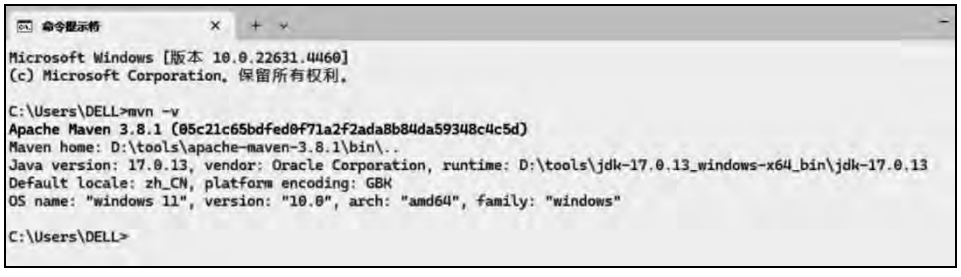

图1.9　验证安装

2．配置国内Maven镜像

配置Maven镜像是为了提高Maven依赖的下载速度，尤其是当默认的Maven中央仓库响应慢或无法访问时，使用镜像站可以帮助用户更快速地下载所需的库和插件，具体操作步骤如下：

01 在Maven的安装目录下，找到conf/settings.xml文件，使用文本编辑器打开这个文件。

02 在settings.xml文件中，找到<mirrors>节点。这里可能已经有一些默认的镜像配置，可以在里面添加新的镜像配置或修改现有的配置。

03 在<mirrors>节点内部，添加一个<mirror>节点。例如，使用阿里云公共仓库的Maven镜像。

04 保存setting.xml文件并关闭文本编辑器。后续如果需要使用Maven下载依赖，就会使用这里配置的镜像站点来下载。

```
<mirror>
<id>aliyunmaven</id>
<mirrorOf>*</mirrorOf>
<name>阿里云公共仓库</name>
<url>https://maven.aliyun.com/repository/public</url>
</mirror>
```

1.4　创建第一个Spring Boot项目

在搭建Spring Boot项目之前，应该确保已经安装Java运行环境。Java开发最常使用的集成开发环境IntelliJ IDEA，通常简称为IDEA。它是由JetBrains公司开发的，主要用于Java开发，同时也支持其他编程语言的开发。

1.4.1　创建Maven项目

创建Spring Boot项目的方式有多种，最常见的是使用官方提供的spring initializr。spring initializr是一个在线工具，用于快速生成一个新的Spring Boot项目。它提供了一个直观的Web界面，使用户能够选择所需的依赖项、项目元数据以及其他配置选项，然后生成一个压缩的项目包，可以直接下载并使用。但是，如果IDEA没有spring initializr，那我们就需要手动创建Maven工程。这里以手动创建Maven工程为例讲解一下相关步骤。

01 打开IDEA，单击New Project按钮，在弹出的New Project窗口左侧选择Maven Archetype，如图1.10所示。右侧的项目信息分别是：

- Name：工程名称。
- Location：工程所存放的位置。
- JDK：选择17以上的版本，Spring Boot 3所支持的JDK最低版本是17。
- Archetype：骨架选择。

02 填写完之后检查最下面的Advanced Settings：

- Group：项目分组名称，如公司、组织或团队的名称，它是项目组织的唯一标识符。对于公司，GroupId可能是org.example。结合ArtifactId，GroupId可确保项目的唯一性。这个设置非常重要，尤其是在将依赖添加到Maven仓库时。

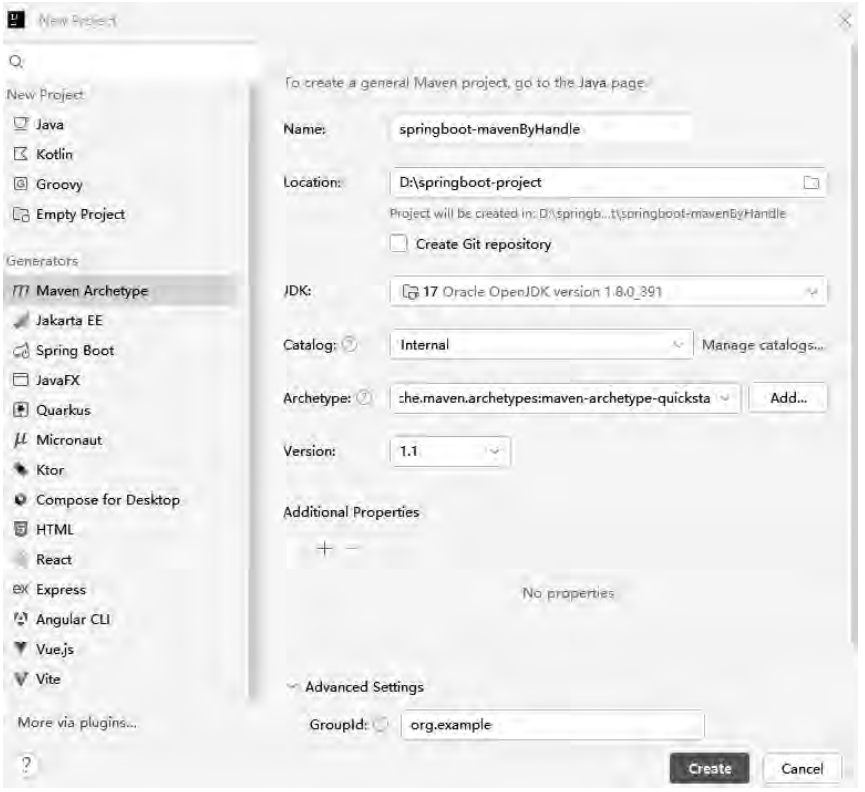

图1.10　创建Maven项目

- ArtifactId: 它表示具体项目或模块的名称（一般就是工程名称）。

以上这些数据通常会被添加到生成的项目的pom.xml（对于Maven项目）或其他配置文件中。这样能够确保目的的唯一性和可识别性，同时也为项目提供了有关其目的和用途的描述性信息。

03 选择完成之后，单击Create按钮创建Maven项目。

1.4.2　引入起步依赖

项目创建完毕之后，在当前工程中配置Maven环境，使其自动导入依赖。

01 在IDEA工具栏中，选择File→Settings→Build, Execution, Deployment→Build Tools→Maven，如图1.11所示。

02 在Maven的配置界面，配置以下参数：

- Maven home path: 指定Maven的安装目录，此处选择IDEA自带的Maven，也就是Bundled(Maven 3)。
- User settings file: 指向Maven的配置文件，此处选择D:\tools\apache-maven-3.8.1\conf\settings.xml。
- Local repository: 指向本地Maven仓库目录，默认是~/.m2/repository。如果需要，可以更改此路径。

03 单击Apply按钮，再单击OK按钮，保存设置并关闭Settings窗口。需要注意的是，上述配置仅在当前项目中生效，如果需要后续创建的所有项目均采用上述配置，可以单击File→New Projects Setup→Settings for NewProjects，再进行一次相同的配置即可。

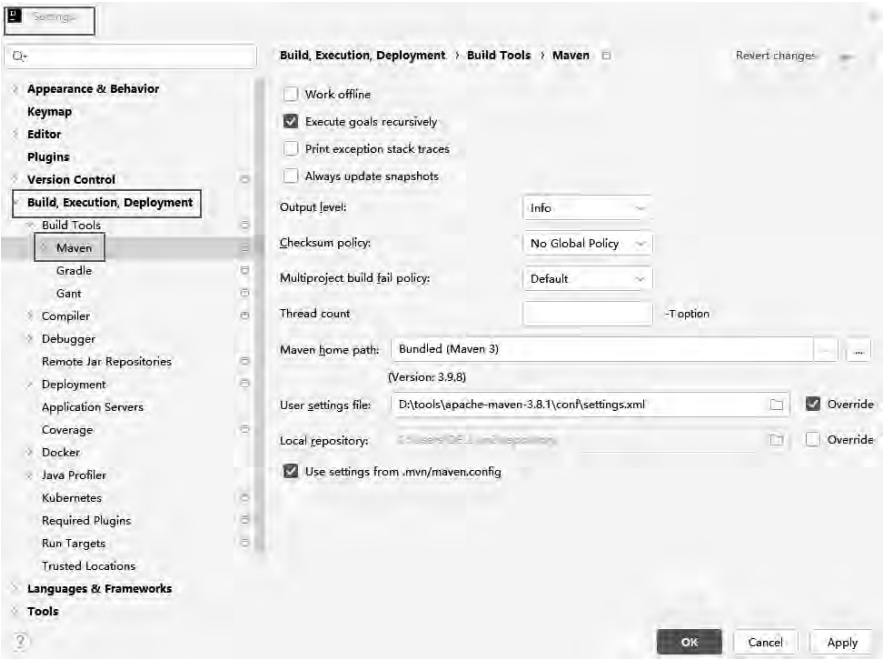

图1.11　配置Maven环境

04 打开pom.xml文件，添加父工程parent标签：

```
<!--父工程，引入父工程 该父工程可以管理下面依赖的版本号-->
  <parent>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-parent</artifactId>
     <version>3.1.9</version>
  </parent>
```

05 在version标签下面添加打包方式为jar：

```
<groupId>org.example</groupId>
    <artifactId>springboot-mavenByHandle</artifactId>
    <version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

06 添加dependencies标签，在该标签内引入Web的起步依赖，同时也引入Spring Boot的测试依赖：

```
 <dependencies>
   <!--springBoot起步依赖-->
 <dependency>
     <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
   <!--测试依赖-->
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
   </dependency>
 </dependencies>
```

1.4.3　编写启动类

编写启动类之前先介绍一下正常的目录结构：

- src/main/java/：此目录包含项目的主要Java源代码。
- src/main/java/Package.name(你自己的包名)/SpringBootApplication（main入口文件）：这是Spring Boot应用程序的入口点，通常包含@SpringBootApplication注解，并包含main方法来启动应用程序。
- src/main/resources/：存放项目的资源文件，如配置文件、国际化属性文件、SQL脚本等。
- src/test/：此目录用于存放项目的测试代码和测试资源。
- pom.xml：Maven的配置文件，定义了项目的依赖、插件和其他设置。
- .gitignore：如果使用Git作为版本控制系统，则此文件定义了不应该被加入版本控制的文件和目录。

在resources目录下又有如下两个目录：

- static/：存放静态资源，如HTML、CSS、JavaScript文件和图片。在运行时，这些文件都是直接可访问的。
- application.properties：Spring Boot的主配置文件。也可以选择使用application.yml文件。

可能由于Maven版本号太低，创建的项目工程并没有src目录，这里我们右击项目，在弹出的快捷菜单中依次选择新建→目录（Directory）→src（IDEA会有提示，src/main/java、src/main/resources...），创建src目录，如图1.12所示。

图1.12　新建文件夹

建完文件夹之后，再创建Spring Boot应用程序的入口文件。入口文件是src/main/java/ Package.name（你自己定义的包名）/入口文件名称。

01 右击java→New→Java Class→输入org.example.SpringBootCreateMavenApplication（包名.文件名，具体的按自己的命名来写）。

02 在这个class文件中写上注解@SpringBootApplication，标识这是入口文件。

03 在main方法里面写上固定代码：SpringApplication.run(该入口文件的文件名称, main方法里面的数组参数)，这里是SpringApplication.run(SpringBootCreateMavenApplication.class, args)，如图1.13所示。

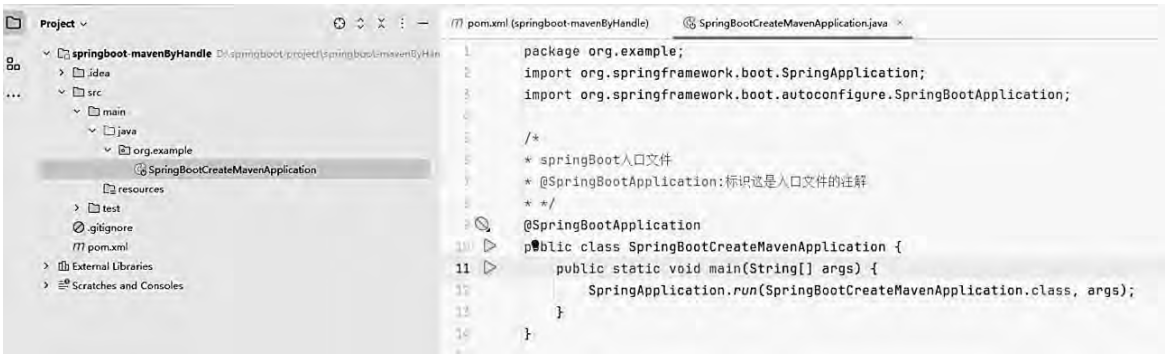

图1.13　入口文件

1.4.4　自定义Controller

前面提到src/main/java/里面存放的是项目的主要Java源代码，所有的Java后端代码都必须与启动类在同级目录下。

（1）右击包名org.example→New→Directory→输入Controller→按回车键。

（2）右击Controller→New→Java Class→输入Controller的名称（HelloController）→按回车键。

（3）要求：编写HelloController，输出“hello，SpringBoot！”。

解释一下注解：

（1）@RestController：是Spring框架中用于定义RESTful控制器的注解。它是一个组合注解，包含了@Controller和@ResponseBody两个注解的功能。

- @Controller：表示该类是一个控制器，用于处理HTTP请求。在Spring MVC中，@Controller注解用于定义一个类为控制器，其中的方法可以处理特定的HTTP请求。
- @ResponseBody：表示该方法的返回值会作为HTTP响应的正文返回，而不是返回一个视图（View）。在Spring MVC中，@ResponseBody注解用于告诉Spring框架，该方法的返回值应该直接写入HTTP响应正文中，而不是返回一个视图。

@RestController注解的组合效果是：

① 标记类为Spring MVC控制器。

② 标记类中方法的返回值作为HTTP响应的正文返回，而不是返回一个视图。

使用@RestController注解可以简化代码，因为它自动将方法的返回值作为HTTP响应的正文，无须在每个方法上添加@ResponseBody注解。这在开发RESTful API时非常有用，因为RESTful API通常返回JSON或XML格式的数据，而不是跳转到一个视图。

（2）@RequestMapping：是Spring MVC中的一个注解，用于将HTTP请求映射到特定的处理方法上。这个注解可以定义在类或者方法上，用于指定类或方法处理的请求类型、请求路径等信息。

以下是@RequestMapping注解的一些常用属性：

- value或path：指定请求的URL路径。可以是一个具体的路径，也可以是路径的模式。
- method：指定请求的类型，如GET、POST、PUT、DELETE等。
- params：指定请求参数的条件，只有当请求参数符合这些条件时，请求才会被映射到对应的方法。
- headers：指定请求头的条件，只有当请求头符合这些条件时，请求才会被映射到对应的方法。
- consumes：指定请求体的媒体类型，用于限制可以处理哪些类型的请求体。
- produces：指定响应的媒体类型，用于指定返回值的格式，如application/json。

@RequestMapping可以与@GetMapping、@PostMapping、@PutMapping、@DeleteMapping等特定HTTP方法的快捷注解一起使用，这些快捷注解自动指定了请求的类型，如图1.14所示。

图1.14　控制层

1.4.5　开始测试

在入口文件SpringBootCreateMavenApplication中右击main方法左边的启动小图标，在弹出的快捷菜单中选择run命令，即可使该工程运行起来，在控制台可以看到Tomcat开启了8080端口，如图1.15所示。

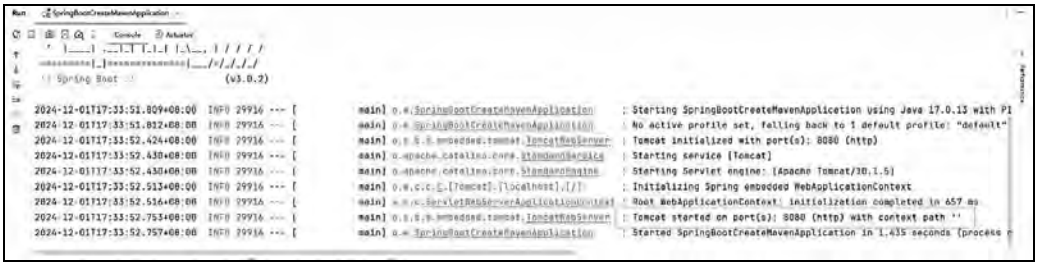

图1.15　启动终端

再次打开浏览器访问http://localhost:8080/hello，即可看到浏览器按照我们的要求输出了“hello，SpringBoot！”，如图1.16所示。这说明项目已成功启动了。

图1.16　运行界面

1.5　Spring Boot的核心配置

Spring Boot采纳了“约定优于配置”的设计理念。例如，在1.4.5节中，当启动项目时，Spring Boot会默认通过其内置的Tomcat在8080端口上启动应用，免去了手动配置的步骤。除此之外，Spring Boot还提供了众多类似的默认约定。当然，如果有特定需求，如更改端口号或指定数据库连接等，也可以通过Spring Boot提供的配置文件进行自定义配置，本节将详细介绍如何在实际项目中进行系统配置。

1.5.1　自动配置

Spring Boot的自动配置是其核心特性之一，它基于类路径中的依赖、环境配置以及自定义代码进行智能化配置，避免开发者手动编写大量的样板代码。以下是自动配置的详细说明。

1）@SpringBootApplication 注解

自动配置的起点通常是@SpringBootApplication注解，它是一个组合注解，包含@SpringBootConfiguration、@EnableAutoConfiguration和@ComponentScan三个重要注解。

- @SpringBootConfiguration标记为一个Spring配置类，类似于@Configuration。
- @EnableAutoConfiguration启用Spring Boot的自动配置机制。
- @ComponentScan用于扫描当前包及其子包下的所有Spring组件。

2）@EnableAutoConfiguration 和 AutoConfigurationImportSelector

@EnableAutoConfiguration注解的作用是告诉Spring Boot启动时自动配置Spring应用。该注解引入了AutoConfigurationImportSelector，这是自动配置的核心处理器。AutoConfigurationImportSelector类会从配置文件中（通常是spring.factories）读取所有的自动配置类，并将它们导入应用上下文中。

3）spring.factories 文件

自动配置类是通过spring-boot-autoconfigure 模块的META-INF/spring.factories文件来配置的。这个文件中列出了所有可以被自动加载的配置类。这些配置类会在Spring Boot启动时根据当前环境条件选择性加载。

4）条件装配（@Conditional 系列注解）

Spring Boot并不是盲目地加载所有的自动配置类。每个自动配置类通常会使用@Conditional系列注解来进行有条件的加载，如@ConditionalOnClass、@ConditionalOnMissingBean、@ConditionalOnProperty等。这些条件注解确保了只有在满足特定条件时，相应的自动配置类才会被加载和应用。

5）自动配置类示例：DataSourceAutoConfiguration

Spring Boot中的DataSourceAutoConfiguration是配置数据源的自动配置类，它展示了如何基于条件注解进行自动配置。如果类路径下存在数据库连接池和数据库驱动，并且没有配置任何数据源，Spring Boot会自动配置内存数据库HSQLDB。

1.5.2　外部化配置

Spring Boot的外部化配置允许应用的配置与代码分离，提高了应用的灵活性和可维护性。以下是外部化配置的详细说明。

1. 配置文件

Spring Boot最常使用配置文件来配置和定制应用程序的行为，支持.properties和.yml/.yaml格式的配置文件。该文件位于src/main/resources/目录中，如application.properties和application.yml，两者的功能是相同的，只是表示和格式有所不同。

1）application.properties

application.properties文件使用键–值对的方式配置应用程序属性。键–值对之间使用“=”进行分隔。例如，指定服务器端口：

```
#设置服务器端口为8082
server.port=8082
```

文件中的“#”表示单行注释，属性之间的层级关系使用点（.）语法表示。下面是一个典型的系统配置内容，包括了服务器端口及数据库连接的相关信息。

```
#设置服务器端口为8082
server.port=8082
#设置数据库的连接地址
spring.datasource.url=jdbc:mysql://localhost:3306/system
#设置数据库账号
spring.datasource.username=root
#设置数据库密码
spring.datasource.password=password
```

2）application.yml

application.yml是Spring Boot中用于配置应用程序属性的YAML格式文件。YAML（YAML Ain’t Markup Language）是一种直观的数据序列化格式，支持数据结构的表示。相对于PROPERTIES文件来说，YML文件更加结构化和简洁。以下是application.yml配置文件的详细讲解。

YAML使用缩进表示层级关系，并且对字母大小写敏感，键–值对之间使用“:”分隔。例如，指定服务器端口：

```
#设置服务器端口为8082
server:
  port: 8082
```

YAML文件非常依赖正确的缩进。一个常见的错误是混淆制表符和空格，或者使用了错误数量的空格进行缩进。在处理YAML文件时，要确保文本编辑器或集成开发环境（IDE）能够正确显示缩进，并避免使用制表符。

下面是一个典型的使用YAML文件的系统配置内容，指定了服务器端口及数据库连接的相关信息。

```
#设置服务器端口为8082
 server:
  port: 8082
 #设置数据库的连接信息
```

```
spring:
 datasource:
   url: jdbc:mysql://localhost:3306/system
   username: root
   password: password
```

application.properties和application.yml两者的区别：

- YAML支持列表配置，而PROPERTIES不支持。
- 使用@PropertySource注解可以加载自定义的PROPERTIES配置文件，但无法加载自定义的YAML文件。
- application.yml的优先级低于application.properties，如果两个文件都存在且配置了同一个属性，会以application.properties中的配置为准。

2. @ConfigurationProperties注解

@ConfigurationProperties注解用于将配置文件中的属性值和配置类中的属性进行映射，以达到自动配置的目的。该注解通过@EnableConfigurationProperties来触发整个流程，参数是Properties配置类。

3. Environment类

Environment是Spring Boot外部化配置的核心类，存储了所有的外部化配置资源，且其他获取外部化配置资源的方式也都依赖该类。

4. 配置加载流程

Spring Boot在启动时会加载外部配置到Environment对象中，然后通过@ConfigurationProperties 将这些配置绑定到相应的配置类中。

1.5.3　命令行配置

Spring Boot允许通过命令行参数来覆盖配置文件中的值。这些参数在应用启动时作为JVM属性传递，或者使用--标志后跟属性名称和值传递。例如：java -jar myapp.jar --my.property=value这种方式可以用于快速测试和临时覆盖配置，而无须修改配置文件。

1.5.4　YAML配置文件

YAML是一种简洁的配置文件格式，Spring Boot支持YAML格式的配置文件，提供了以下特点：

- 层级结构：YAML支持层级结构，使得配置文件更加清晰和易于管理。
- 数据类型丰富：YAML支持包括列表、映射、布尔值等多种数据类型。
- 配置合并：Spring Boot允许使用多个YAML文件，并且支持配置的合并和覆盖。
- 配置绑定：与.properties文件一样，可以使用@ConfigurationProperties将YAML中的配置绑定到Java对象中。

YAML配置文件提供了一种更加直观和灵活的方式来组织和管理应用配置，具体的配置规则参见1.5.2节。

1.6　小　　结

在本章中，我们深入探讨了Spring Boot的基础知识，从其简介、特点与优势，到实际搭建第一个Spring Boot项目，再到核心配置机制的详细解析。Spring Boot能简化Spring应用开发，提供了一种快速、高效的方式来构建独立、生产级别的基于Spring的应用程序。

Spring Boot的核心优势在于其自动配置、独立运行的特性，以及对微服务架构的支持。这些特点使得开发者能够专注于业务逻辑，而无须深陷于复杂的配置之中。通过内嵌容器、生产就绪的特性和跨平台兼容性，Spring Boot大大提升了开发效率和应用的可移植性。

在实践操作方面，我们介绍了如何安装Maven构建工具，创建Maven项目，并引入起步依赖来搭建Spring Boot项目。通过自定义Controller和编写启动类，我们讲解了Spring Boot应用的基本结构和运行机制。此外，还探讨了如何进行应用测试，确保开发的应用能够按预期工作。

核心配置部分，我们详细讨论了自动配置、外部化配置、命令行配置和YAML配置文件。这些配置机制不仅提高了配置的灵活性，也使得应用能够更好地适应不同的环境和需求。

总的来说，本章为读者提供了一个全面的Spring Boot入门指南，从理论到实践，帮助读者建立起对Spring Boot的深刻理解，并为进一步的学习打下坚实的基础。通过本章的学习，读者应该能够熟练地使用Spring Boot来构建和配置Java应用程序。

第 2 章

使用Spring Boot进行Web开发

在第1章中，我们介绍了Spring Boot的基础知识，并搭建了第一个Spring Boot项目。随着互联网技术的飞速发展，Web应用已经成为我们日常生活中不可或缺的一部分。它们不仅改变了我们获取信息、沟通和交易的方式，也为企业提供了展示自身、拓展市场和服务客户的重要平台。Spring Boot以其简洁和强大的特性，成为开发现代Web应用的首选框架之一。现在，我们将深入探讨如何使用Spring Boot进行Web开发。本章将引导你从Web开发的基础知识开始，逐步深入到Spring Boot在构建Web应用中的高级特性和最佳实践。通过本章的学习，你将掌握Spring Boot在Web开发中的强大功能，并能够构建出响应速度快、易于维护且安全的Web应用。

2.1 实体与数据持久层

在Web开发中，数据持久化是核心需求之一，它涉及将数据存储到数据库中以及从数据库中检索数据。Spring Boot通过提供对数据访问抽象的支持，简化了数据持久层（Data Persistence Layer）的实现。本节将详细介绍数据持久层框架、实体（Entity）、Spring Data JPA，以及如何使用Lombok来简化实体类的创建。

2.1.1 数据持久层框架

数据持久层是应用程序中负责数据持久化的部分，包括数据的存储、检索、更新和删除。在Java Web应用中，常用的数据持久层框架包括：

- JDBC（Java Database Connectivity）：Java提供的原生数据库连接机制，通过SQL语句直接与数据库交互。
- JPA（Java Persistence API）：Java EE的规范，提供了一种对象-关系映射（ORM）的解决方案，允许开发者以面向对象的方式处理数据库操作。
- Hibernate：一个强大的ORM框架，实现了JPA规范，提供了额外的功能，如缓存、事务管理等。
- MyBatis：另一种流行的ORM框架，提供了SQL映射和对象关系映射功能，允许更细致的SQL控制。

Spring Boot通过自动配置和起步依赖简化了这些框架的集成和使用。

2.1.2　实体

在JPA中，实体是指被映射到数据库表的Java类。实体类通常包含一系列属性，这些属性对应数据库表的列。实体类通过使用JPA注解与数据库表建立映射关系。例如：

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String email;
    // getters and setters
}
```

在这个例子中，User类是一个实体，它映射到数据库中的users 表。@Entity注解标记这个类为一个JPA实体，@Table注解指定了对应的数据库表名。

2.1.3　Spring Data JPA

Spring Data JPA是Spring提供的一个用于简化数据库操作的数据访问工具集，它在JPA的基础上增加了更多的便利性。Spring Data JPA的核心是Repository接口，它允许开发者自定义数据访问方法而无须编写实现类。例如：

```
public interface UserRepository extends JpaRepository<User, Long>
{
    List<User> findByName(String name);
}
```

在这个例子中，UserRepository接口继承了JpaRepository接口，提供了基本的CRUD操作和查询方法。CRUD是指在做计算处理时的增加（Create）、读取（Read）、更新（Update）和删除（Delete）操作。findByName方法将自动生成对应的查询实现，并返回所有名字匹配的用户列表。

2.1.4　使用Lombok简化POJO

Lombok是一个Java库，它通过注解的方式简化Java类的编写，特别是在创建POJO（Plain Old Java Objects）时。使用Lombok，可以避免编写重复的模板代码，如getter和setter、构造函数、toString、equals和hashCode方法等。

例如，对于User实体类，使用Lombok可以这样写：

```
@Entity
@Table(name = "users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
10      private String name;
11      private String email;
12  }
```

在这个例子中，@Data注解为类生成了所有必要的布尔方法，包括getter、setter、toString、equals和hashCode。@NoArgsConstructor和@AllArgsConstructor注解分别生成了无参和全参构造函数。通过使用Lombok，开发者可以减少模板代码的编写，从而提高开发效率，使代码更加简洁和易于维护。

综上所述，本节内容涵盖了数据持久层的基础知识、实体的概念、Spring Data JPA的强大功能，以及如何使用Lombok来简化实体类的创建。这些知识为后续的Web开发奠定了坚实的基础，使得开发者可以更专注于业务逻辑的实现。

2.2　MVC与模板引擎

随着Web技术的不断演进，用户对于Web应用的体验要求越来越高。一个优秀的Web应用不仅要有强大的后端逻辑，更要有直观、响应式的前端界面。MVC（Model-View-Controller）架构和模板引擎正是为了满足这些需求而生。

MVC架构是一种设计模式，它将应用分为3个核心组件：模型（Model）、视图（View）和控制器（Controller）。这种分离使得开发者可以专注于单个组件的开发，而不必担心其他部分的实现细节。在Spring Boot 3中，MVC架构得到了进一步的增强和简化，使得开发者可以更加轻松地构建和维护Web应用。

模板引擎则是一种服务器端技术，用于渲染视图。它允许开发者使用模板文件（通常包含HTML代码和特定的模板语法）来动态生成页面。在Spring Boot中，常用的模板引擎包括Thymeleaf、FreeMarker和Velocity等。这些模板引擎与Spring MVC紧密集成，使得在Spring应用中渲染HTML页面变得异常简单。在Spring Boot 3中，模板引擎的使用进一步简化，提供了更多的功能和更好的性能。它们不仅支持复杂的页面渲染逻辑，还支持国际化、CSS/JS集成和服务器端页面分割等高级特性。

2.2.1　MVC框架

MVC框架是Web应用开发中常用的设计模式，其架构如图2.1所示。

MVC的业务流程图

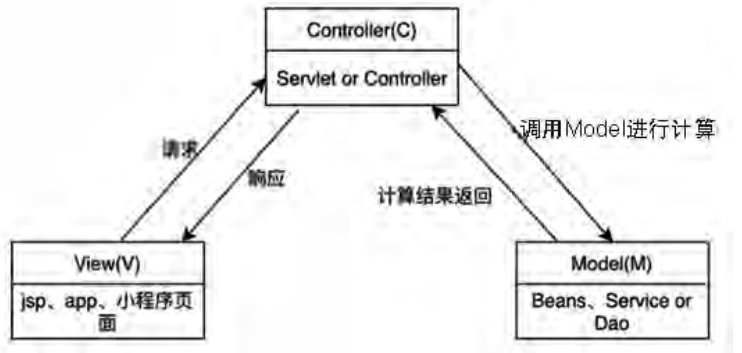

图2.1　MVC三层架构

- 模型（Model）：负责业务数据和业务逻辑，即数据的处理和存储。
- 视图（View）：负责展示数据，即用户界面。
- 控制器（Controller）：接收用户请求，调用模型和视图完成对用户请求的处理。

Spring MVC是Spring框架中实现MVC模式的模块，提供了一种灵活的方式来构建Web应用。它通过注解和配置文件来简化MVC组件的定义和使用。接下来我们详细讲解一下SpringBoot的MVC模式。

1. Spring Boot MVC自动配置

Spring Boot MVC通过自动配置简化了Spring MVC的使用。当你在Spring Boot项目中添加spring-boot-starter-web依赖时，Spring Boot会自动配置Spring MVC，包括：

- DispatcherServlet：这是Spring MVC的核心组件，负责处理所有的HTTP请求。
- ViewResolver：默认的视图解析器，用于解析视图名称到具体的视图模板。
- ContentNegotiationManager：用于处理客户端的内容协商。
- MessageConverters：用于转换HTTP请求和响应体。

2. 控制器（Controller）

在Spring Boot中，可以使用@Controller或@RestController注解来创建控制器：

- @Controller：用于创建返回视图名称的传统控制器。
- @RestController：是@Controller和@ResponseBody的组合，用于创建RESTful Web服务，其方法的返回值会自动作为HTTP响应体。

3. 请求映射（@RequestMapping）

使用@RequestMapping注解可以将HTTP请求映射到特定的控制器方法上。它可以指定请求类型（如GET、POST）、请求路径等。在controller中写一个“/hello”的请求示例如下：

```
@RestController
public class MyController {

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String hello() {
        return "Hello, Spring MVC!";
    }
}
```

4. 请求参数和响应

Spring MVC提供了多种方式来处理请求参数和响应：

- 请求参数：可以使用@RequestParam、@PathVariable和@RequestBody注解来获取请求参数。
- 响应：可以使用@ResponseBody注解将方法返回值作为响应体，或者返回视图名称让视图解析器解析。

示例如下：

```
    // 处理注册表单提交
    @PostMapping("/register")
    public String registerUser(@RequestParam("username") String username,
```

```
                            @RequestParam("password") String password) {
        // 这里添加将用户信息保存到数据库的逻辑
        System.out.println("Registering user: " + username);
        return "success";
    }
    // 返回一个简单的响应消息
    @GetMapping("/greeting")
    public @ResponseBody Greeting getGreeting() {
        Greeting greeting = new Greeting("Hello, World!");
        return greeting;
    }
```

在上面这个例子中，registerUser方法处理表单提交，@RequestParam注解用于将请求参数username和password绑定到控制器方法的参数上，getGreeting方法返回一个Greeting对象，@ResponseBody注解告诉Spring MVC将对象直接序列化为JSON格式的响应体。

5. 异常处理

Spring MVC提供了异常处理机制，可以使用@ControllerAdvice和@ExceptionHandler注解全局处理控制器中的异常。例如：

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception e) {
    return new ResponseEntity<>(e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

6. 数据绑定和验证

Spring MVC支持自动数据绑定和验证，例如：

- 数据绑定：可以使用@ModelAttribute注解将请求参数绑定到控制器方法的参数上。
- 验证：可以使用JSR-303/JSR-380注解（如@Valid）来验证数据。

示例如下：

```
@PostMapping("/users")
public String createUser(@Valid @ModelAttribute User user, BindingResult result) {
    if (result.hasErrors()) {
        return "error";
    }
    // 保存用户信息
    return "success";
}
```

7. 国际化

Spring MVC支持国际化，可以通过MessageSource组件来解析国际化消息。

8. 拦截器

Spring MVC提供了拦截器机制，可以使用HandlerInterceptor接口或@ControllerAdvice注解来实现请求的前置和后置处理。

最后，Spring Boot MVC框架通过自动配置和简化配置，使得开发者可以更专注于业务逻辑的实现，而不是配置的细节。这让构建Web应用程序变得更加快速和简单。

2.2.2 Thymeleaf模板引擎

Thymeleaf是一个现代服务器端的Java模板引擎，用于Web和独立环境。它能够处理HTML、XML、JavaScript、CSS甚至纯文本，使得开发者可以快速上手。Thymeleaf的主要特点包括：

- 自然模板：Thymeleaf的模板在浏览器中看起来像正常的HTML，即使没有Thymeleaf处理，也能正常显示静态内容。
- 灵活的语法：提供了丰富的表达式语法，支持变量表达式、选择表达式、消息表达式等。
- 集成Spring：Thymeleaf与Spring框架紧密集成，特别是与Spring MVC，提供了对Spring表达式语言（SpEL）的支持。
- 国际化支持：Thymeleaf支持文本的国际化，方便多语言应用的开发。

在Spring Boot 3中使用Thymeleaf模板引擎，可以遵循以下步骤进行集成和使用。

1. 添加Thymeleaf依赖

首先，需要在项目的pom.xml文件中添加Thymeleaf的依赖。如果使用Maven构建项目，可以添加如下依赖：

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

这将引入Spring Boot Thymeleaf Starter，它包含了Thymeleaf的所有必要依赖。

2. 配置Thymeleaf属性（可选）

Spring Boot的自动配置已经提供了一个合理的默认配置，但仍可以通过application.properties或application.yml文件自定义Thymeleaf的一些属性。例如通过application.properties来配置Thymeleaf：

```
#设置Thymeleaf模板文件的前缀位置（默认是src/main/resources/templates）
spring.thymeleaf.prefix=classpath:/templates/
#设置模板文件的后缀（默认是.html）
spring.thymeleaf.suffix=.html
#设置模板模式（默认是HTML5，Thymeleaf 3中为HTML）
spring.thymeleaf.mode=HTML
#开启模板缓存（开发时建议关闭，生产时开启）
spring.thymeleaf.cache=false
```

这些配置可以帮助你根据项目需求调整Thymeleaf的行为。

3. 创建Thymeleaf模板

接下来，在src/main/resources/templates目录下创建Thymeleaf模板文件。例如，创建一个名为greeting.html的模板：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
```

```
<head>
  <title>Greeting</title>
</head>
<body>
  <h1 th:text="'Hello, ' + ${name} + '!'">Hello, World!</h1>
</body>
</html>
```

在这个模板中，th:text属性用于动态替换文本内容。

4. 创建一个控制器

现在，创建一个Spring MVC控制器，用于处理用户请求并返回Thymeleaf模板视图：

```
@Controller
public class GreetingController {
  @GetMapping("/greeting")
  public String greeting(@RequestParam(name="name", required=false,
                                       defaultValue="World")
String name, Model model) {
      model.addAttribute("name", name);
      return "greeting";
  }
}
```

在这个控制器中，greeting方法处理/greeting路径的GET请求，接收一个名为name的请求参数，并将它添加到模型中。然后，它返回greeting作为视图的名称，Spring Boot会自动使用Thymeleaf解析器解析greeting.html模板。

5. 运行应用并访问页面

启动你的Spring Boot应用，并在浏览器中访问http://localhost:8080/greeting。你应该能看到基于提供的name参数（如果没有提供，则默认为"World"）渲染的问候消息。

通过这些步骤，你就可以在Spring Boot 3项目中使用Thymeleaf模板引擎来渲染动态Web页面了。这种方式提供了一种强大而灵活的方法来构建交互式的Web应用。

2.2.3　构建MVC架构的Web应用

构建一个基于Spring Boot 3和Thymeleaf的MVC架构Web应用，通常遵循以下具体步骤：

01 添加依赖：在项目的pom.xml文件中添加Thymeleaf的起步依赖spring-boot-starter-thymeleaf。
02 创建控制器：定义一个控制器类，使用@Controller注解，并定义处理HTTP请求的方法。
03 定义视图：在src/main/resources/templates目录下创建Thymeleaf模板文件，编写HTML代码，并使用Thymeleaf的语法来动态展示数据。
04 模型和视图数据绑定：在控制器方法中，通过Model对象添加属性，这些属性可以在Thymeleaf模板中被访问和展示。
05 返回视图名称：控制器方法返回的字符串通常对应于Thymeleaf模板的名称，Spring Boot会自动解析这个名称为实际的模板文件路径。

通过这些步骤，你可以构建一个完整的MVC架构Web应用，其中模型负责处理业务逻辑，视图负责

展示，控制器负责接收请求和返回响应。Thymeleaf作为模板引擎，使得视图的开发变得更加简单和直观。

2.3　文件的上传和下载

上传和下载文件是Web应用中常见的功能，它们允许用户将文件从客户端传输到服务器（上传）以及从服务器传输到客户端（下载）。这些功能在多种场景中都非常关键，例如，数据交换中用户需要上传文档、图片、视频等文件到服务器，以便进行存储、处理或分享；企业内部系统可能需要上传日志、报表等数据文件，并允许用户下载分析结果。

2.3.1　上传文件

在Spring Boot 3中，文件上传功能允许用户将本地文件发送到服务器。在技术实现上，这通常涉及HTTP的multipart/form-data编码类型，它允许多个数据部分在单个请求中发送。文件上传可以通过MultipartFile接口实现，该接口提供了必要的方法来处理上传的文件数据，通常涉及以下步骤：

01 配置文件上传属性。

在application.properties或application.yml中设置文件上传的相关属性，如最大文件大小和请求大小限制。例如，在.properties文件中配置：

```
spring.servlet.multipart.max-file-size=10MB
spring.servlet.multipart.max-request-size=10MB
```

这确保了应用能够处理大文件的上传。

02 处理文件上传请求。

创建一个Controller来处理文件上传请求，使用@RequestParam注解来接收MultipartFile对象，然后使用其transferTo()方法将文件保存到目标位置：

```
@RestController
public class FileUploadController {
  @PostMapping("/upload")
  public String handleFileUpload(@RequestParam("file") MultipartFile file) {
      if (file.isEmpty()) {
          return "Please select a file to upload.";
      }
      try {
          byte[] bytes = file.getBytes();
          String uploadDir = "/path/to/upload/directory/";
          File uploadedFile = new File(uploadDir + file.getOriginalFilename());
          file.transferTo(uploadedFile);
          return "Success";
      } catch (IOException e) {
          e.printStackTrace();
          return "Error";
      }
  }
}
```

在这个示例中，上传的文件被保存在服务器的指定目录下，具体路径由变量uploadDir指定。

2.3.2　下载文件

在Spring Boot 3中，文件下载功能允许用户从服务器请求文件，并将其保存到本地。在Web应用中，这通常涉及设置正确的HTTP响应头，告诉浏览器发送的是文件内容，以及文件的名称。后端创建一个Controller来处理文件下载请求，并根据文件名找到对应的文件，再将其内容以流的形式返回给客户端。可以使用ResponseEntity来封装文件的响应，并设置适当的HTTP头部，使浏览器能够下载文件：

```
@RestController
@RequestMapping("/files")
public class FileDownloadController {
  private static final String FILE_DIRECTORY = "/path/to/your/files/directory/";
    @GetMapping("/{fileName:.+}")
  public ResponseEntity<Resource> downloadFile(@PathVariable String fileName) {
     try {
        Path filePath = Paths.get(FILE_DIRECTORY).resolve(fileName).normalize();
        Resource resource = new UrlResource(filePath.toUri());
        if (!resource.exists()) {
           throw new FileNotFoundException("File not found " + fileName);
        }
        return ResponseEntity.ok()
             .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" +
resource.getFilename() + "\"")
              .body(resource);
     } catch (Exception e) {
        return ResponseEntity.internalServerError().build();
     }
  }
}
```

在这个示例中，downloadFile方法处理下载请求，根据请求的文件名构建文件路径，并创建一个Resource对象来表示要下载的文件。然后，设置Content-Disposition头部，指定文件作为附件下载。

通过这些步骤，我们可以在Spring Boot 3中实现文件的上传和下载，并在前端使用控制器作为映射路径，从而建立起清晰的路由结构。这些功能在许多Web应用中是基础需求之一，例如文件管理系统、个人资料上传等。通过Spring Boot的自动配置和简洁的API，可以让文件的上传和下载变得更为简便和高效。

2.4　日　　志

日志（Log）是系统或应用程序在运行过程中记录的事件信息，用于追踪操作流程、监控运行状态、排查错误和分析行为。

2.4.1　使用预设配置

Spring Boot 3默认使用SLF4J作为日志门面，配合Logback作为日志实现。这种预设配置允许开发

者通过简单的配置文件调整日志级别和输出格式，而无须深入了解具体的日志框架实现。Spring Boot 3的spring-boot-starter-logging模块包含了日志系统的基本配置，这些配置通常位于application.properties或application.yml文件中。

2.4.2　基础配置

在基础配置中，可以通过修改application.properties或application.yml文件来设置全局日志级别，或者指定特定包或类的日志级别。例如：

```
# application.properties
logging.level.root=INFO
logging.level.org.springframework.web=DEBUG
```

这将设置全局日志级别为INFO，并将org.springframework.web包的日志级别设置为DEBUG。

2.4.3　详细配置

对于更详细的日志配置，如日志格式和日志输出位置，可以在外部配置文件中指定。例如，对于Logback，可以创建一个logback-spring.xml或logback.xml文件，并在其中定义日志的appender、encoder和日志级别。对于Log4j2，可以创建一个log4j2-spring.xml或log4j2.xml文件进行类似配置，配置XML的方法参见2.4.5节。

2.4.4　Lombok注解

在使用Spring Boot 3 进行日志记录时，Lombok提供了一种简化日志处理的方式。通过在类上添加@Slf4j注解，Lombok会自动为类生成一个日志对象，省去了手动获取Logger对象的步骤。这样，开发者可以直接使用log.info()、log.debug()等方法记录日志，从而简化了代码编写。例如：

```
@Slf4j
public class MyService {
    public void doSomething() {
        log.info("Doing something");
    }
}
```

通过上述配置，可以实现日志的控制台输出和文件输出，并且可以根据需要调整日志级别和输出格式。

2.4.5　在Windows平台输出彩色日志的Jansi

Jansi是一个用于格式化控制台输出的轻量级Java库，能够通过ANSI（American National Standards Institute，美国国家标准协会）转义序列为输出添加色彩。在Windows平台上，由于默认的控制台不支持ANSI颜色代码，Jansi提供了一个桥接，使得日志框架能够在Windows控制台上输出彩色文本。要在Spring Boot 3中使用Jansi输出彩色日志，可以在项目的依赖中添加Jansi库，并在日志配置文件中启用ANSI支持。以下是具体的配置步骤。

1. 添加Jansi依赖

在项目的pom.xml文件中添加Jansi的依赖：

```
<dependency>
 <groupId>org.fusesource.jansi</groupId>
<artifactId>jansi</artifactId>
<version>2.4.1</version> <!-- 请使用最新的版本 -->
</dependency>
```

2. 配置Logback以使用Jansi

在application.properties或application.yml中添加以下配置来启用Jansi：

```
# application.properties
logging.config=classpath:logback-spring.xml
```

然后，在src/main/resources目录下创建logback-spring.xml文件，并添加以下彩色日志的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="10 seconds">
  <contextName>logback</contextName>
  <!-- 彩色日志依赖的渲染类 -->
  <conversionRule conversionWord="clr"
                  converterClass="org.springframework.boot.logging.logback.
                                  ColorConverter"/>
  <conversionRule conversionWord="wex"

converterClass="org.springframework.boot.logging.logback.
                WhitespaceThrowableProxyConverter"/>
  <conversionRule conversionWord="wEx"

converterClass="org.springframework.boot.logging.logback.
ExtendedWhitespaceThrowableProxyConerter"/>

  <!-- 彩色日志格式 -->
  <property name="CONSOLE_LOG_PATTERN"
          value="${CONSOLE_LOG_PATTERN:-%clr(%d{yyyy-MM-dd
                  HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:-%5p})
                  %clr(${PID:- }) {magenta} %clr(--
){faint} %clr([%15.15t]){faint} %clr(%
40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD: -%wEx}}"/>

  <!-- 输出到控制台 -->
  <appender name="CONSOLE"
          class="ch.qos.logback.core.ConsoleAppender">       <encoder>
          <Pattern>${CONSOLE_LOG_PATTERN}</Pattern>
          <!--设置字符集 -->
        <charset>UTF-8</charset>
     </encoder>
  </appender>

  <!--设置全局日志级别和Appender -->
  <root level="INFO">
     <appender-ref ref="CONSOLE"/>
  </root>
</configuration>
```

3. 启用Jansi

对于Log4j2，在2.10版本以后，默认关闭了Jansi。要启用Jansi，需要设置系统属性log4j.skipJansi为false。这可以在IDEA中设置，单击右上角的Edit Configurations，在VM options中添加：

```
-Dlog4j.skipJansi=false
```

或者在application.properties或application.yml中设置：

```
# application.properties
log4j.skipJansi=false
```

通过上述步骤，就可以在Windows平台上实现Spring Boot应用输出彩色日志了。这样配置后，日志输出将更加直观，并且更易于区分不同级别的日志信息。

通过这些配置和工具，Spring Boot 3提供了灵活且强大的日志系统，帮助开发者有效地管理和监控应用程序的日志输出。

2.5　过滤器和拦截器

在Spring Boot应用中，过滤器（Filter）和拦截器（Interceptor）都是处理请求和响应的强大工具。它们可以帮助我们在请求到达控制器之前（预处理）或响应发送给客户端之后（后处理）执行特定的逻辑。

2.5.1　过滤器

过滤器是Servlet规范的一部分，用于在请求到达Servlet或过滤器链中的其他过滤器之前或之后执行任务。过滤器可以用于日志记录、身份验证、授权、请求内容修改等。

（1）创建过滤器：

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

public class MyFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // 初始化过滤器
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        //在请求处理之前执行
        System.out.println("Before request processing");

        // 继续过滤器链中的下一个过滤器/servlet
```

```
        chain.doFilter(request, response);

        //在请求处理之后执行
        System.out.println("After request processing");
    }

    @Override
    public void destroy() {
        // 销毁过滤器
    }
}
```

（2）注册过滤器：

```
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;

public class FilterConfig {
    @Bean
    public FilterRegistrationBean<MyFilter> myFilter() {
        FilterRegistrationBean<MyFilter> registrationBean =
         new FilterRegistrationBean<>();
        registrationBean.setFilter(new MyFilter());
        registrationBean.addUrlPatterns("/api/*");
        return registrationBean;
    }
}
```

2.5.2　使用过滤器实现访问控制

过滤器非常适合实现访问控制，如IP白名单、用户认证等。示例如下：

```
@Override
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
    HttpServletRequest req = (HttpServletRequest) request;
    String token = req.getHeader("Authorization");
    if (token == null || !token.equals("SECRET_TOKEN")) {
        ((HttpServletResponse) response).setStatus(HttpServletResponse.
          SC_UNAUTHORIZED);
        return;
    }
    chain.doFilter(request, response);
}
```

2.5.3　拦截器

拦截器是Spring框架提供的一个组件，用于在请求的多个点执行自定义逻辑。拦截器可以用于日志记录、权限检查、请求修改等。

（1）创建拦截器：

```
import org.springframework.web.servlet.HandlerInterceptor;

public class MyInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) {
        //在请求处理之前执行
        System.out.println("Before controller");
        return true; // 继续流程
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) {
        // 请求处理之后，视图渲染之前执行
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) {
        // 请求处理和视图渲染之后执行
    }
}
```

（2）注册拦截器：

```
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.context.annotation.Configuration;

@Configuration
public class WebConfig {
    @Autowired
    private MyInterceptor myInterceptor;

    @Bean
    public WebMvcConfigurerAdapter webConfigurer() {
        return new WebMvcConfigurerAdapter() {
            @Override
            public void addInterceptors(InterceptorRegistry registry) {
                registry.addInterceptor(myInterceptor).addPathPatterns("/api/*");
            }
        };
    }
}
```

2.5.4　使用拦截器记录请求参数

拦截器可以用来记录请求的参数，这对于调试和监控非常有用。示例如下：

```
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
```

```
03   Object handler) {
04       Enumeration<String> paramEnum = request.getParameterNames();
05       while (paramEnum.hasMoreElements()) {
06           String paramName = paramEnum.nextElement();
07           System.out.println("Parameter: " + paramName + " = " +
                                request.getParameter(paramName));
08       }
09       return true;
10   }
```

通过使用过滤器和拦截器，Spring Boot应用能够更灵活地处理请求和响应，实现诸如日志记录、身份验证、授权等重要功能。

2.6　Spring Boot事件

Spring Boot中的事件机制允许在应用的不同部分之间进行松耦合的交互。事件发布者不必知道谁是监听者，监听者也不必知道事件从何而来，这种机制提高了应用的模块化和灵活性。

事件机制在以下场景中非常有用：

- 解耦合组件：当不同的应用组件需要通信，但又不想直接调用对方的方法时。
- 实现横切关注点：如日志记录、事务管理、安全性等。
- 实现插件架构：允许插件响应应用核心功能生成的事件。
- 实现工作流和状态机：事件可以触发状态转换。

Spring Boot事件机制提供了一种强大的方式来构建松耦合、灵活且易于维护的应用程序。通过合理使用事件，开发者可以构建出更加模块化和响应式的系统。

2.6.1　事件驱动模型

事件驱动模型是一种设计模式，允许组件通过事件来通信，而不是直接调用彼此的方法。在Spring框架中，事件的发布和监听是通过ApplicationEvent类和ApplicationListener接口实现的。

- 事件（Event）：事件是应用在特定时间点发生的事情，它可以是任何Java对象，但通常是继承自ApplicationEvent的自定义类。
- 事件发布（Event Publisher）：任何组件都可以发布事件，只需获取ApplicationEventPublisher实例并调用其publishEvent方法。
- 事件监听（Listener）：任何组件都可以监听事件，只需实现ApplicationListener接口或使用@EventListener注解。

2.6.2　内置事件

Spring Boot提供了一系列的内置事件，这些事件在应用的生命周期中的关键点被触发。一些常见的内置事件包括：

- ContextRefreshedEvent：在Spring上下文初始化或刷新后发布。

- ContextStartedEvent：在Spring上下文启动后发布。
- ContextStoppedEvent：在Spring上下文停止后发布。
- ContextClosedEvent：在Spring上下文关闭后发布。

2.6.3　监听内置事件

要监听内置事件，可以使用@EventListener注解。以下是一个监听ContextRefreshedEvent的示例：

```
01  import org.springframework.context.event.EventListener;
02  import org.springframework.stereotype.Component;
03  import org.springframework.context.event.ContextRefreshedEvent;
04
05  @Component
06  public class MyEventListener {
07
08      @EventListener
09      public void handleContextRefresh(ContextRefreshedEvent event) {
10          System.out.println("Context refreshed!");
11      }
12  }
```

2.6.4　自定义事件

自定义事件可以通过创建一个继承ApplicationEvent的类来实现。以下是一个自定义事件的示例：

```
01  import org.springframework.context.ApplicationEvent;
02
03  public class MyCustomEvent extends ApplicationEvent {
04      private String message;
05
06      public MyCustomEvent(Object source, String message) {
07          super(source);
08          this.message = message;
09      }
10
11      public String getMessage() {
12          return message;
13      }
14  }
```

然后，可以创建一个监听器来监听这个自定义事件：

```
01  import org.springframework.context.event.EventListener;
02  import org.springframework.stereotype.Component;
03  @Component
04  public class MyCustomEventListener {
05      @EventListener
06      public void handleCustomEvent(MyCustomEvent event) {
07          System.out.println("Received custom event - " + event.getMessage());
08      }
09  }
```

2.6.5　异步事件

Spring还支持异步事件处理，这意味着事件的处理不会阻塞发布者。异步事件处理通过@Async注解和SimpleAsyncTaskExecutor实现，允许事件监听方法在单独的线程中执行。以下是一个异步事件监听器的示例：

```
import org.springframework.scheduling.annotation.Async;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
public class AsyncEventListener {

    @Async
    @EventListener
    public void handleAsyncEvent(MyCustomEvent event) {
        // 异步处理事件
        System.out.println("Async handling custom event - " + event.getMessage());
    }
}
```

为了使@Async注解生效，需要在配置中启用异步操作：

```
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableAsync;

@Configuration
@EnableAsync
public class AsyncConfig {
    // 配置类内容
}
```

通过使用事件机制，Spring Boot应用可以实现更加灵活和响应式的架构，同时提高代码的模块化和可测试性。

2.7　小　　结

在完成关于使用Spring Boot进行Web开发的学习后，相信读者对构建企业级Web应用有了更深入的理解。以下是本章学习的重点小结：

（1）实体与数据持久层：

- 介绍了如何将业务对象映射到数据库表中，使用JPA和Hibernate作为ORM工具来简化数据库操作。
- 讲解了Spring Data JPA的使用，包括定义Repository接口、实现自定义查询方法以及事务管理。

（2）MVC与模板引擎：

- 介绍了Spring MVC的核心组件：模型、视图、控制器，以及如何配置和使用它们。
- 探索了Thymeleaf等模板引擎的使用，包括动态内容生成、数据绑定和国际化。

（3）文件的上传和下载：

- 介绍了如何在Spring Boot应用中处理文件的上传和下载，包括配置文件大小限制和实现文件存储逻辑。
- 讲解了使用MultipartFile接口上传文件和使用Resource下载文件的方法。

（4）日志：

- 介绍了Spring Boot的日志系统，包括配置Logback或Log4j2等日志框架。
- 讲解了如何通过日志级别、日志模式和日志文件管理来优化应用的日志记录。

（5）过滤器和拦截器：

- 介绍了在Spring Boot中使用过滤器和拦截器来处理请求和响应前后的逻辑。
- 讲解了如何实现自定义过滤器和拦截器，以及如何将它们注册到Spring Boot应用中。

（6）Spring Boot事件：

- 介绍了Spring事件机制，包括发布事件、监听事件以及异步事件处理。
- 讲解了如何创建自定义事件和监听器，以及如何利用事件机制进行应用内部通信。

通过本章的学习，读者不仅能掌握Spring Boot Web开发的关键技术点，而且能够将这些知识点应用到实际的项目开发中，构建出功能丰富、结构清晰、易于维护的Web应用。这些知识为进一步的微服务架构学习、前后端分离开发以及云原生应用开发奠定了坚实的基础。随着实践的深入，我们可以更加灵活地运用Spring Boot来解决复杂的业务问题，并构建出更加健壮和可扩展的Web服务。