



Spring Boot 讲义

北京动力节点教育科技有限公司

动力节点课程讲义

DONGLJIEDIANKECHENGJIANGYI

www.bjpowernode.com

第1章 Spring Boot 框架入门

1.1 Spring Boot 简介

Spring Boot 是 Spring 家族中的一个全新的框架，它用来简化 Spring 应用程序的创建和开发过程，也可以说 Spring Boot 能简化我们之前采用 SpringMVC + Spring + MyBatis 框架进行开发的过程。

在以往我们采用 SpringMVC + Spring + MyBatis 框架进行开发的时候，搭建和整合三大框架，我们需要做很多工作，比如配置 web.xml，配置 Spring，配置 MyBatis，并将它们整合在一起等，而 Spring Boot 框架对此开发过程进行了革命性的颠覆，完全抛弃了繁琐的 xml 配置过程，采用大量的默认配置简化我们的开发过程。

所以采用 Spring Boot 可以非常容易和快速地创建基于 Spring 框架的应用程序，它让编码变简单了，配置变简单了，部署变简单了，监控变简单了。正因为 Spring Boot 它化繁为简，让开发变得极其简单和快速，所以在业界备受关注。

Spring Boot 在国内的关注趋势图：<http://t.cn/ROQLquP>

1.2 Spring Boot 的特性

- 能够快速创建基于 Spring 的应用程序
- 能够直接使用 java main 方法启动内嵌的 Tomcat 服务器运行 Spring Boot 程序，不需要部署 war 包文件
- 提供约定的 starter POM 来简化 Maven 配置，让 Maven 的配置变得简单
- 自动化配置，根据项目的 Maven 依赖配置，Spring boot 自动配置 Spring、Spring mvc 等
- 提供了程序的健康检查等功能
- 基本可以完全不使用 XML 配置文件，采用注解配置

1.3 Spring Boot 四大核心

1.3.1 自动配置

1.3.2 起步依赖

1.3.3 Actuator

1.3.4 命令行界面

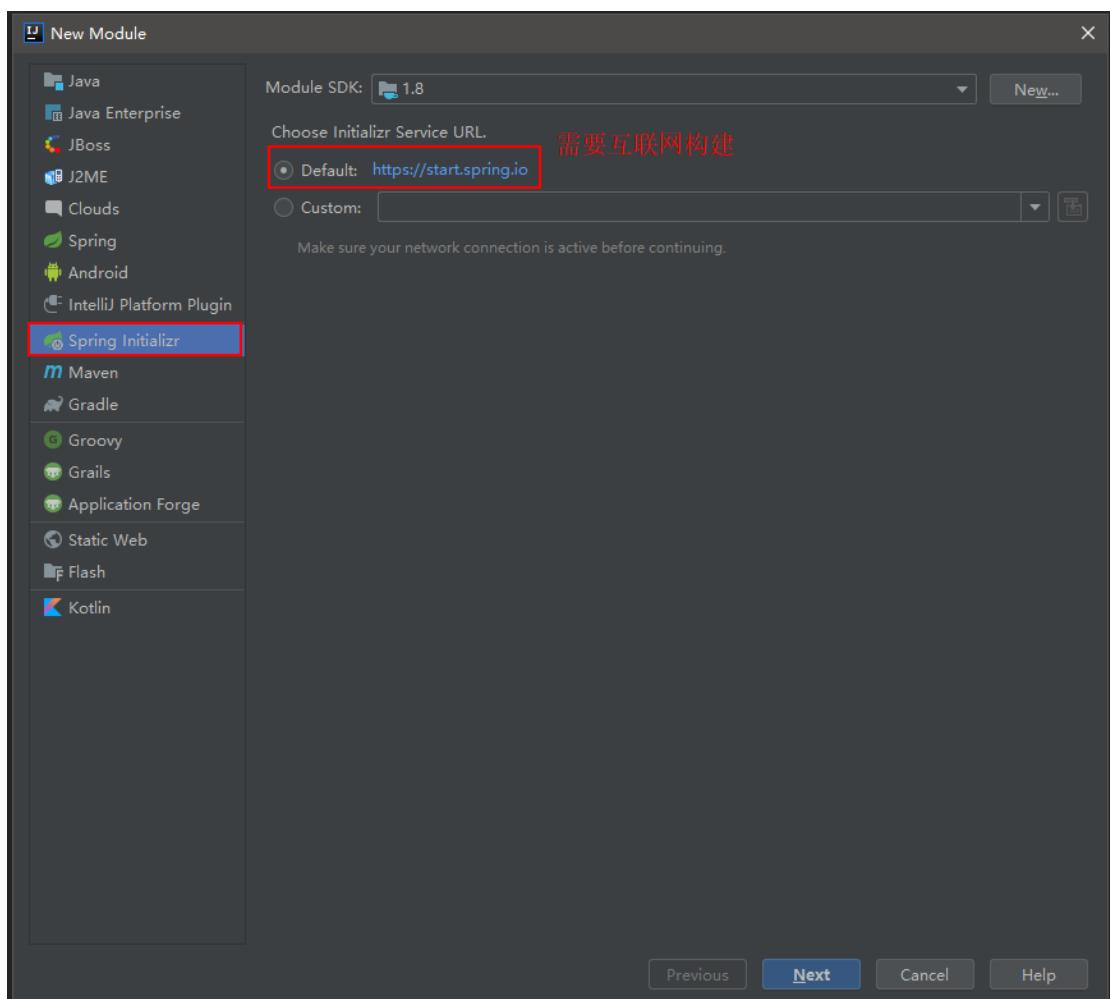
第2章 Spring Boot 入门案例

2.1 第一个 SpringBoot 项目

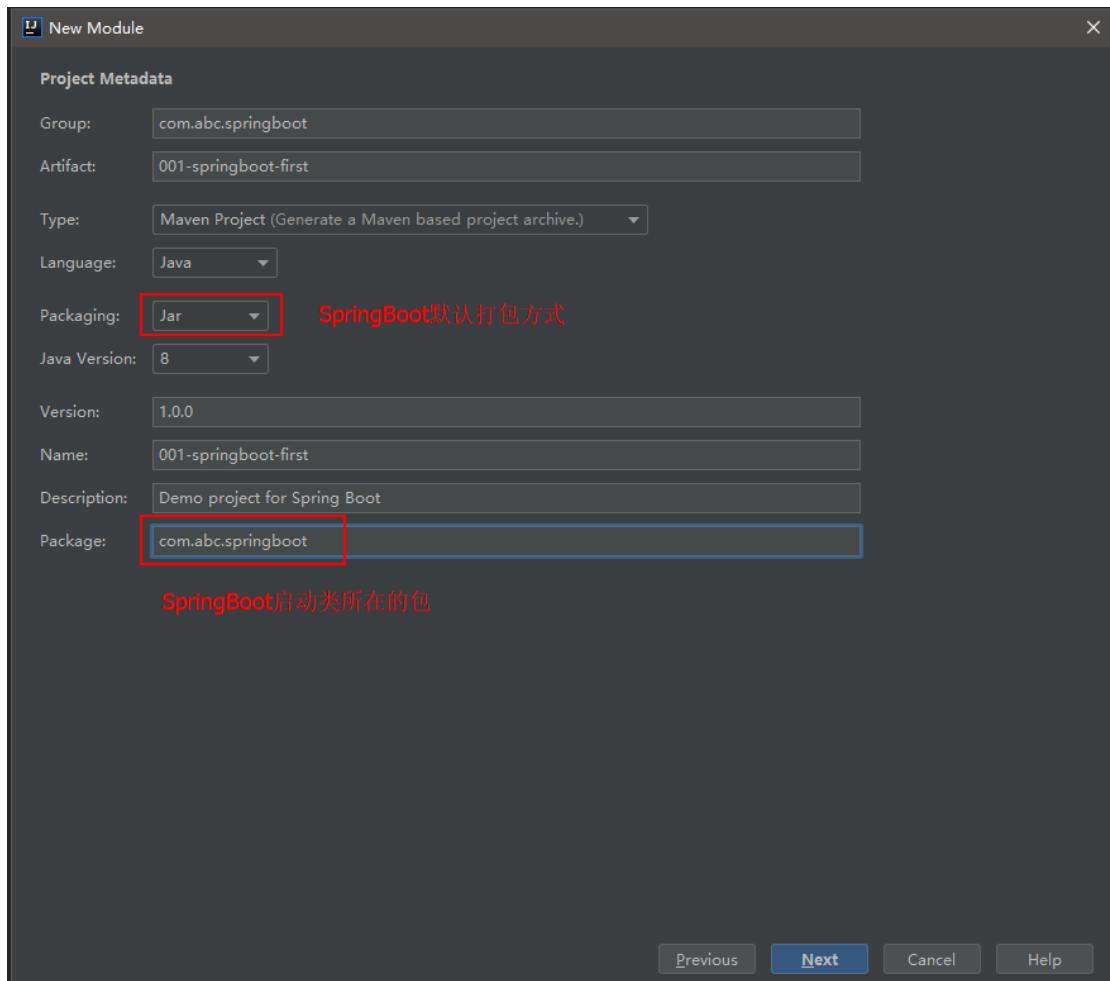
2.1.1 开发步骤

项目名称：001-springboot-first

(1) 创建一个 Module，选择类型为 Spring Initializr 快速构建

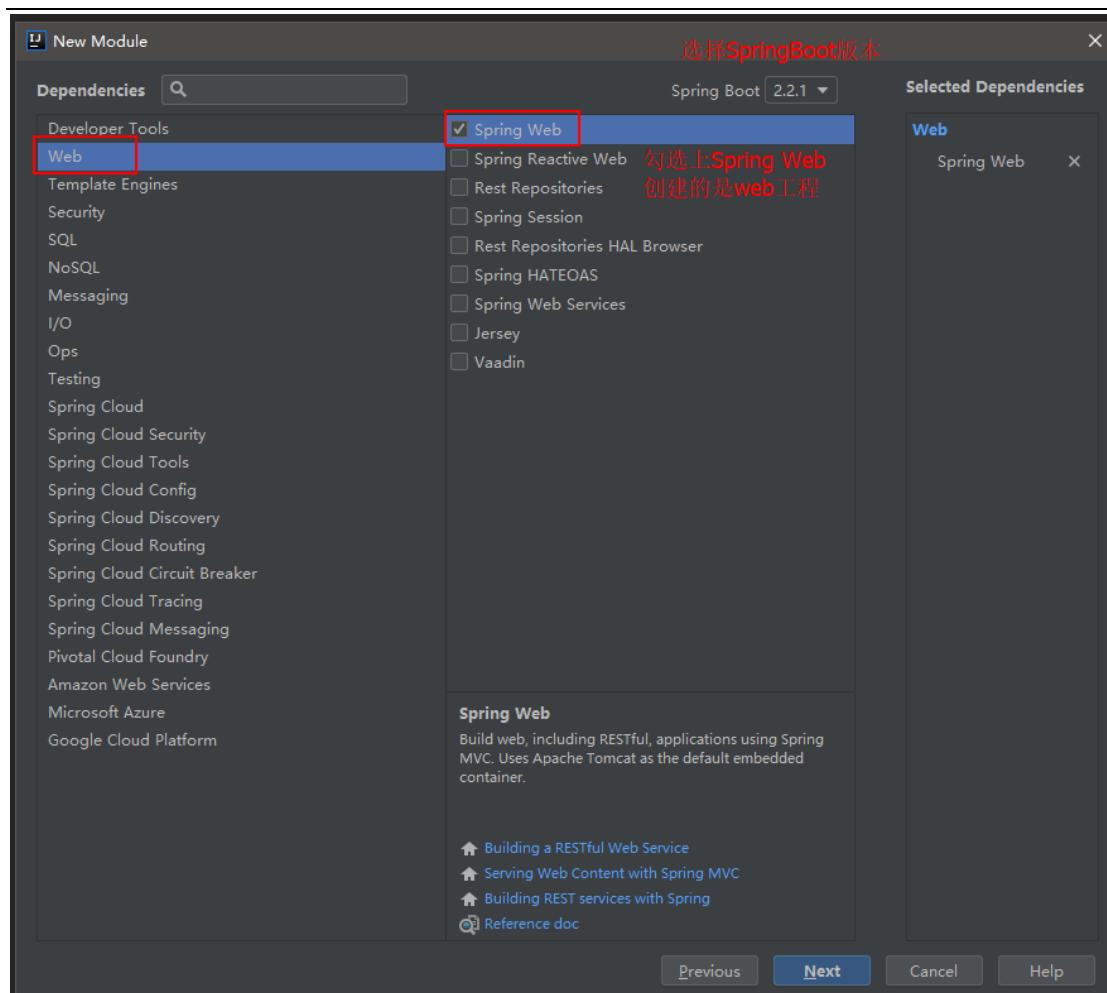


(2) 设置 GAV 坐标及 pom 配置信息

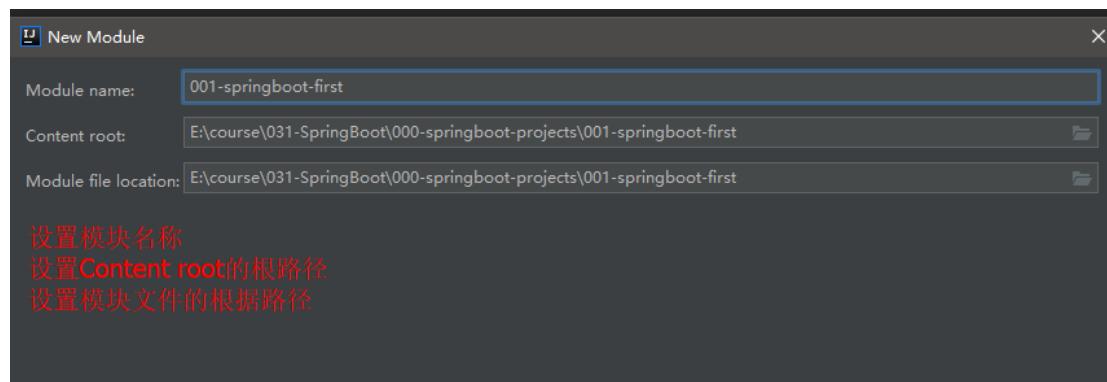


(3) 选择 Spring Boot 版本及依赖

会根据选择的依赖自动添加起步依赖并进行自动配置

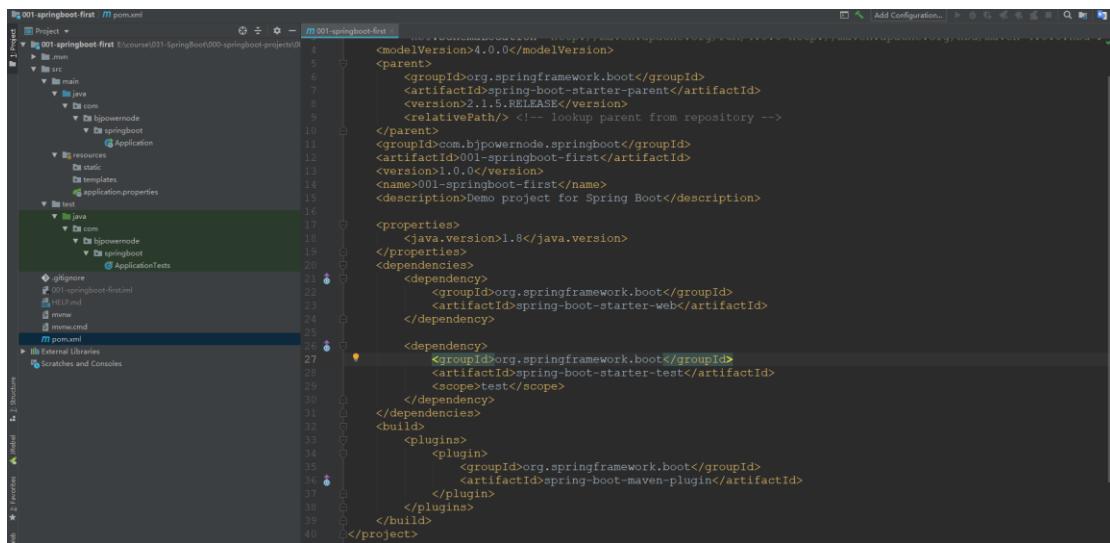


(4) 设置模块名称、Content Root 路径及模块文件的目录



点击 Finish, 如果是第一次创建, 在右下角会提示正在下载相关的依赖

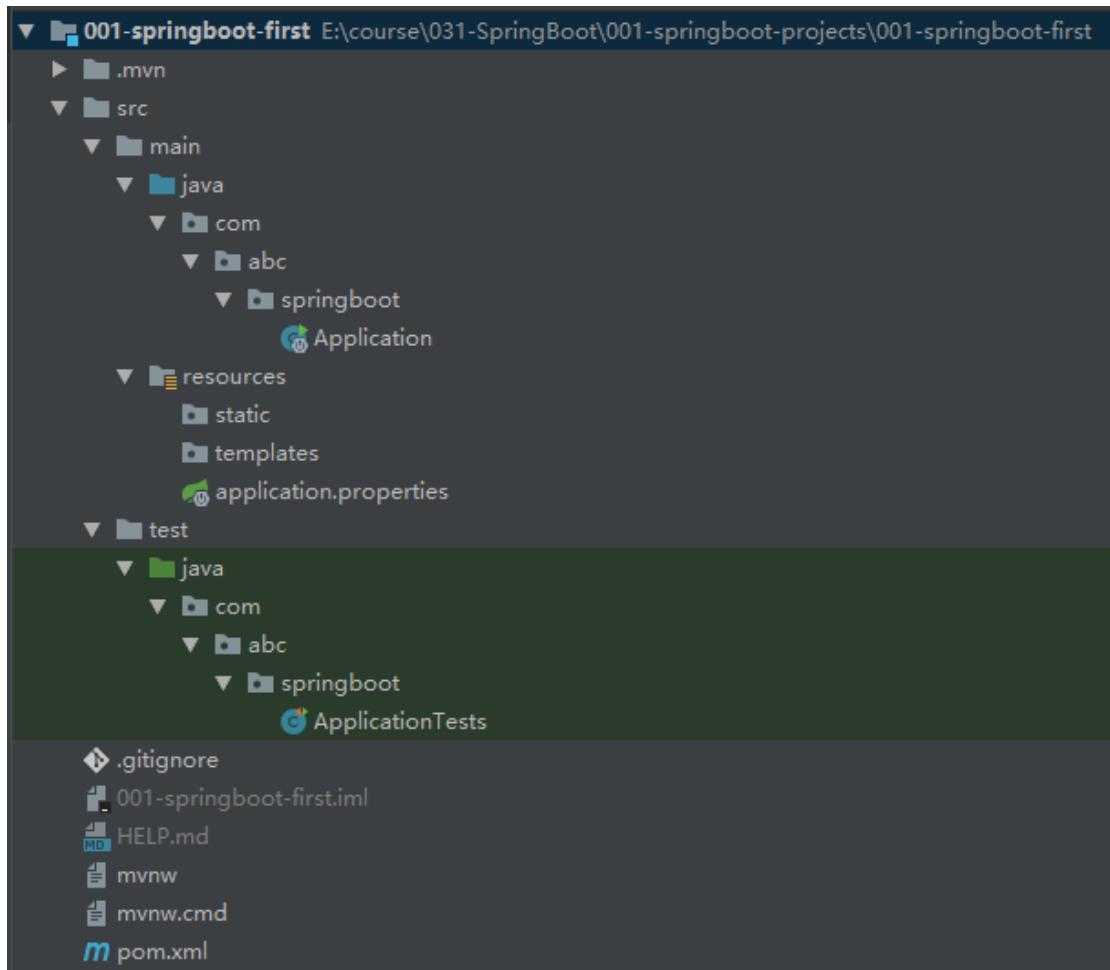
(5) 项目创建完毕，如下



The screenshot shows the IntelliJ IDEA interface with the project '001-springboot-first' open. The left sidebar displays the project structure, including .mvn, src (with main and test subfolders), resources (static, templates, application.properties), and test (with java, com, and springboot subfolders). The right pane shows the content of the pom.xml file:

```
<modelVersion>4.0.0</modelVersion>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.5.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.bnjpownode.springboot</groupId>
<artifactId>001-springboot-first</artifactId>
<version>1.0.0</version>
<name>001-springboot-first</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>1.8</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>
```

(6) 项目结构



static: 存放静态资源, 如图片、CSS、JavaScript 等

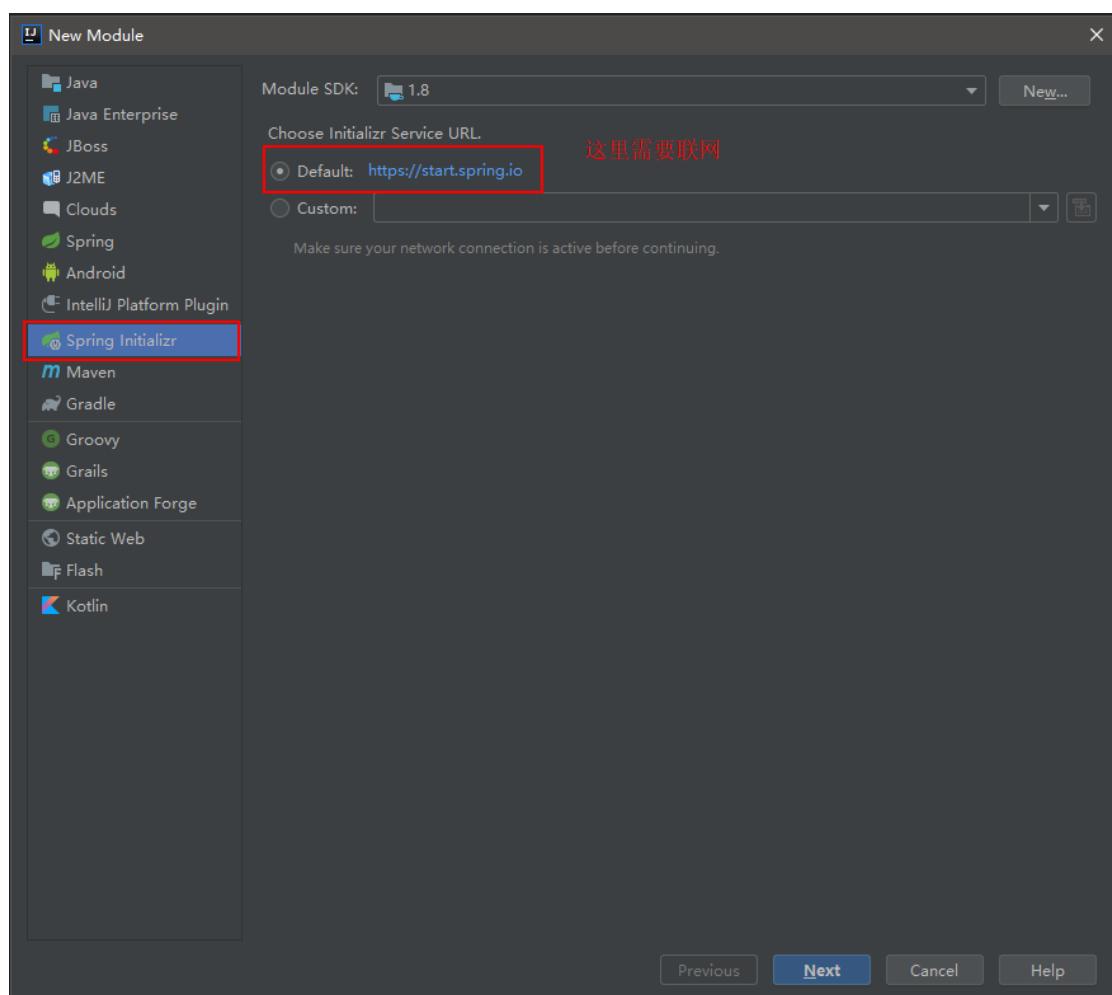
templates: 存放 Web 页面的模板文件

application.properties/application.yml 用于存放程序的各种依赖模块的配置信息, 比如 服务端口, 数据库连接配置等

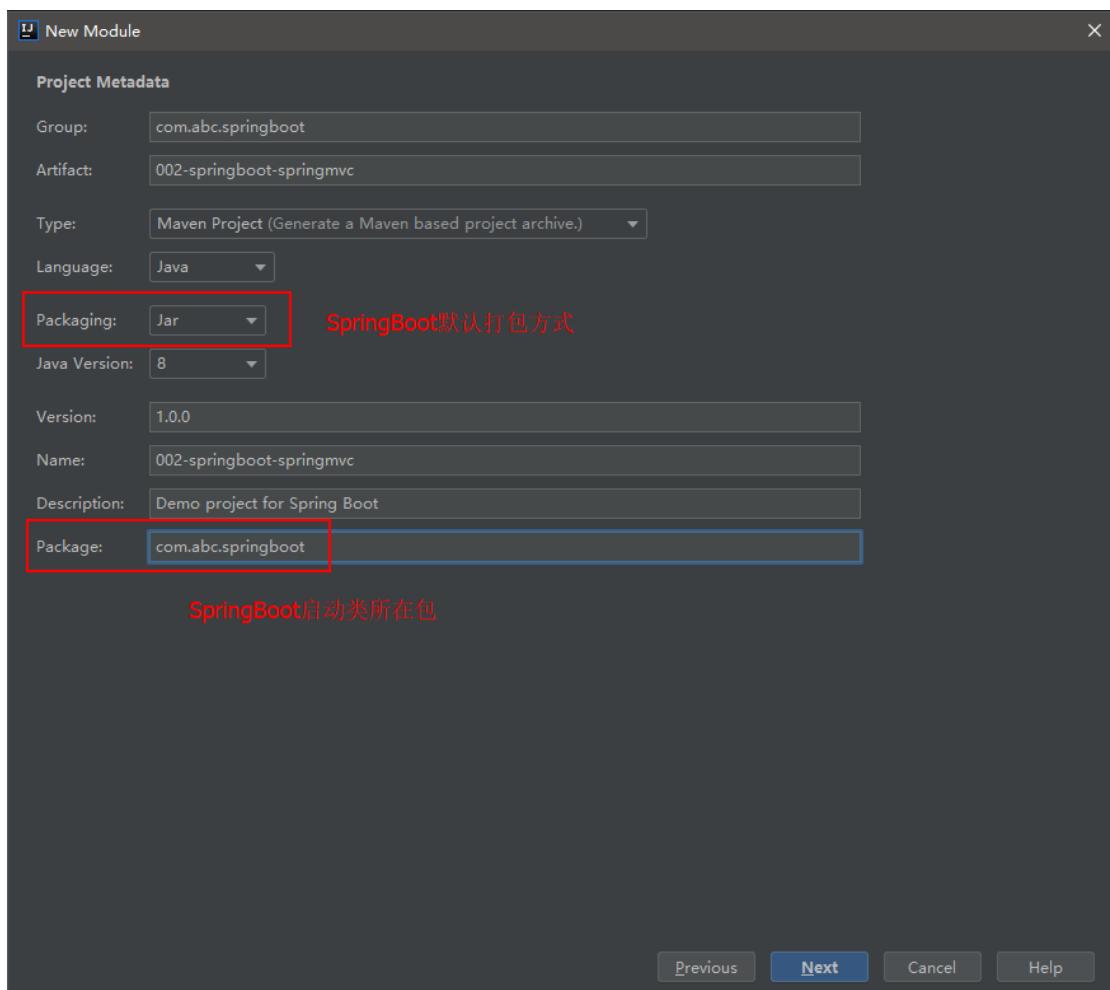
2.2 入门案例

项目名称: 002-springboot-springmvc

2.2.2 创建一个新的 Module, 选择类型为 Spring Initializr

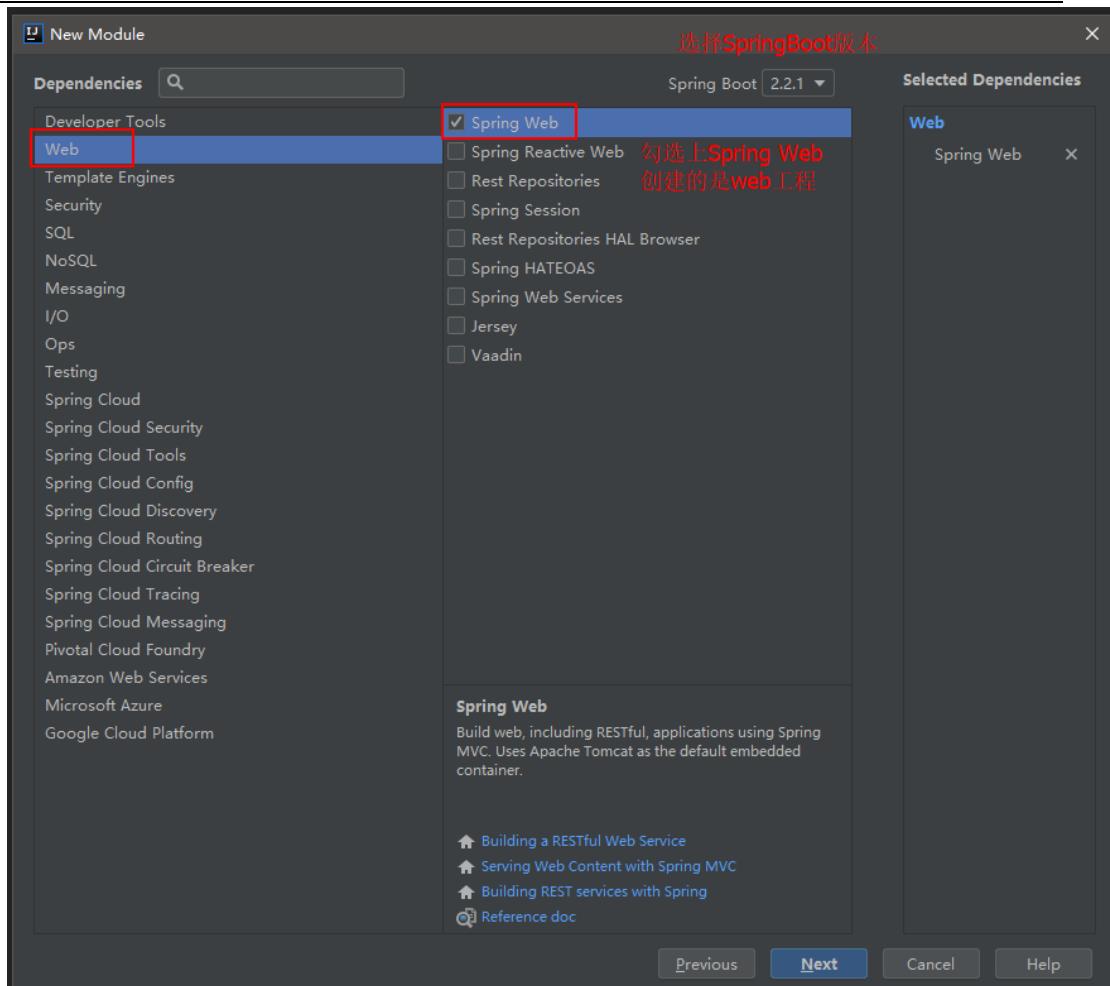


2.2.3 指定 GAV 及 pom 配置信息

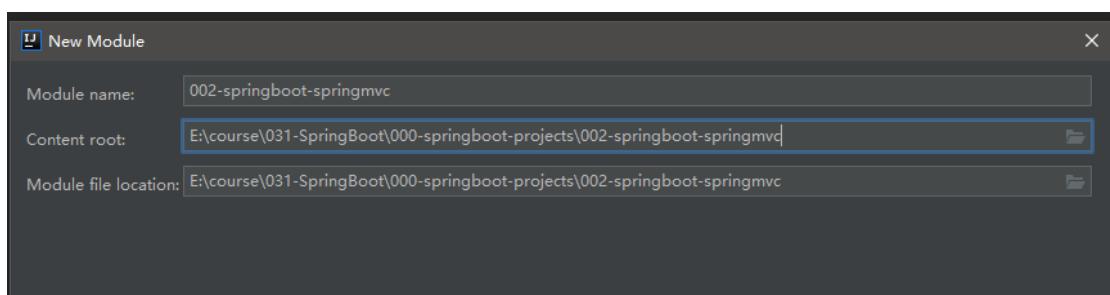


2.2.4 选择 Spring Boot 版本及依赖

会根据选择的依赖自动添加起步依赖并进行自动配置



2.2.5 修改 Content Root 路径及文件所在目录



2.2.6 对 POM.xml 文件进行解释

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<!--继承 SpringBoot 框架的一个父项目，所有自己开发的 Spring Boot 都必须的继承-->

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.1.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<!--当前项目的 GAV 坐标-->
<groupId>com.bjpowernode.springboot</groupId>
<artifactId>002-springboot-springmvc</artifactId>
<version>1.0.0</version>

<!--maven 项目名称，可以删除-->
<name>002-springboot-springmvc</name>
<!--maven 项目描述，可以删除-->
<description>Demo project for Spring Boot</description>

<!--maven 属性配置，可以在其它地方通过${}方式进行引用-->
<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <!--SpringBoot 框架 web 项目起步依赖，通过该依赖自动关联其它依赖，不需要我们一个一个去添加了-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

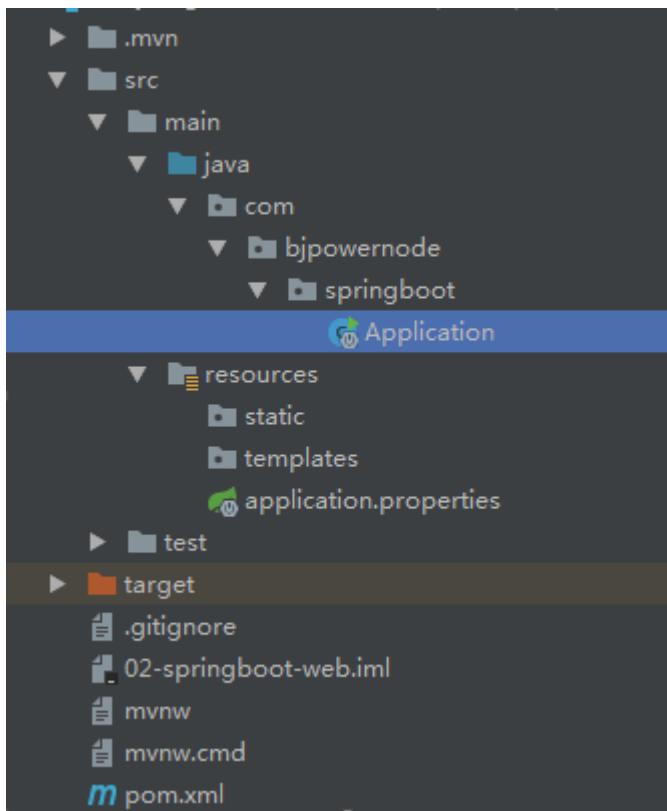
    <!--SpringBoot 框架的测试起步依赖，例如：junit 测试，如果不需要的话可以删除-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
        <exclusions>
            <exclusion>
                <groupId>org.junit.vintage</groupId>
                <artifactId>junit-vintage-engine</artifactId>
            </exclusion>
        </exclusions>
    </dependency>

```

```
</exclusions>
</dependency>
</dependencies>

<build>
    <plugins>
        <!--SpringBoot 提供的打包编译等插件-->
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>
```

2.2.7 对 SpringBoot 项目结构进行说明



- .mvn|mvnw|mvnw.cmd: 使用脚本操作执行 maven 相关命令, 国内使用较少, 可删除
- .gitignore: 使用版本控制工具 git 的时候, 设置一些忽略提交的内容
- static|templates: 后面模板技术中存放文件的目录
- application.properties: SpringBoot 的配置文件, 很多集成的配置都可以在该文件中进行配置, 例如: Spring、springMVC、Mybatis、Redis 等。目前是空的
- Application.java: SpringBoot 程序执行的入口, 执行该程序中的 main 方法, SpringBoot 就启动了

2.2.8 创建一个 Spring MVC 的 Spring BootController

SpringBootController 类所在包: com.bjpowernode.springboot.web

```
package com.bjpowernode.springboot.web;

import org.springframework.stereotype.Controller;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * ClassName:SpringBootController
 * Package:com.bjpowernode.springboot.web
 * Description:<br/>
 */
@Controller
public class SpringBootController {

    @RequestMapping(value = "/springBoot/say")
    public @ResponseBody String say() {
        return "Hello,springBoot!";
    }
}
```

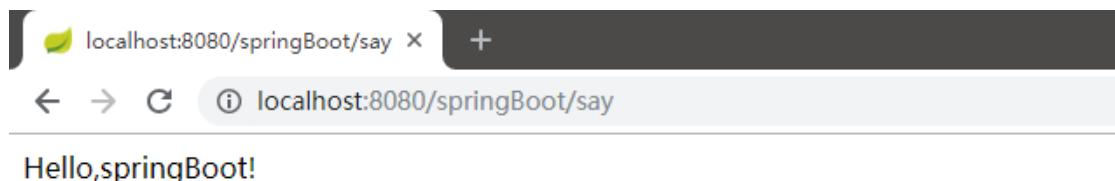
注意：新创建的类一定要位于 **Application** 同级目录或者下级目录，否则 **SpringBoot** 加载不到。

2.2.9 在 IDEA 中右键，运行 **Application** 类中的 **main** 方法

通过在控制台的输出，可以看到启动 **SpringBoot** 框架，会启动一个内嵌的 **tomcat**，端口号为 **8080**，上下文根为空

```
Tomcat started on port(s): 8080 (http) with context path ''
```

2.2.10 在浏览器中输入 <http://localhost:8080/springBoot/say> 访问



2.3 入门案例分析

- Spring Boot 的父级依赖 spring-boot-starter-parent 配置之后，当前的项目就是 Spring Boot 项目
- spring-boot-starter-parent 是一个 Springboot 的父级依赖，开发 SpringBoot 程序都需要继承该父级项目，它用来提供相关的 Maven 默认依赖，使用它之后，常用的 jar 包依赖可以省去 version 配置
- Spring Boot 提供了哪些默认 jar 包的依赖，可查看该父级依赖的 pom 文件
- 如果不想使用某个默认的依赖版本，可以通过 pom.xml 文件的属性配置覆盖各个依赖项，比如覆盖 Spring 版本

```
<properties>
```

```
    <spring-framework.version>5.0.0.RELEASE</spring-framework.version>
```

```
</properties>
```

- **@SpringBootApplication** 注解是 Spring Boot 项目的核心注解，主要作用是开启 Spring 自动配置，如果在 Application 类上去掉该注解，那么不会启动 SpringBoot 程序
- main 方法是一个标准的 Java 程序的 main 方法，主要作用是作为项目启动运行的入口
- @Controller 及 @ResponseBody 依然昰我们之前的 Spring MVC，因为 Spring Boot 的里面依然昰使用我们的 Spring MVC + Spring + MyBatis 等框架

2.4 Spring Boot 的核心配置文件

Spring Boot 的核心配置文件用于配置 Spring Boot 程序，名字必须以 application 开始

2.4.1 核心配置格式

(7) .properties 文件（默认采用该文件）

在 002-springboot-springmvc 项目基础上，进行修改

项目名称: 003-springboot-port-context-path

通过修改 application.properties 配置文件，在修改默认 tomcat 端口号及项目上下文件根键值对的 properties 属性文件配置方式

```
#设置内嵌 Tomcat 端口号  
server.port=9090  
  
#配置项目上下文根  
server.servlet.context-path=/003-springboot-port-context-path
```

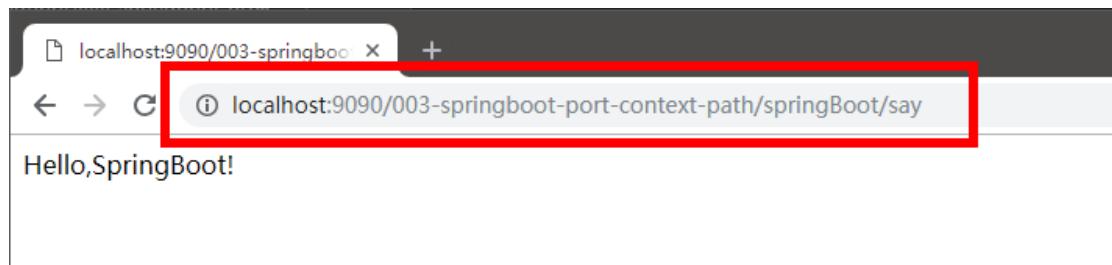
配置完毕之后，启动浏览器测试

```
application.properties
server.port=9090
server.servlet.context-path=/003-springboot-port-context-path

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

    @RestController
    @RequestMapping("/003-springboot-port-context-path")
    public class SpringBootController {
        ...
    }
}
```

页面显示结果



(8) .yml 文件

项目名称：004-springboot-yml，在003项目基础之上

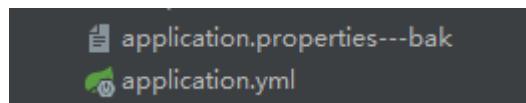
yml是一种yaml格式的配置文件，主要采用一定的空格、换行等格式排版进行配置。

yaml是一种直观的能够被计算机识别的数据序列化格式，容易被人类阅读，yaml类似于xml，但是语法比xml简洁很多，值与前面的冒号配置项必须要有一个空格，yml后缀也可以使用yaml后缀



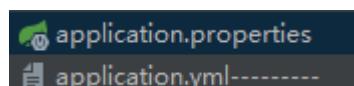
```
server:  
  port: 9090  
  servlet:  
    context-path: /
```

注意：当两种格式配置文件同时存在，使用的是.properties配置文件，为了演示yml，可以先将其改名，重新运行Application，查看启动的端口及上下文根



```
Tomcat started on port(s): 9090 (http) with context path ''
```

我们以后在授课的过程中，使用properties，所以演示完yml效果后，将该配置文件改名

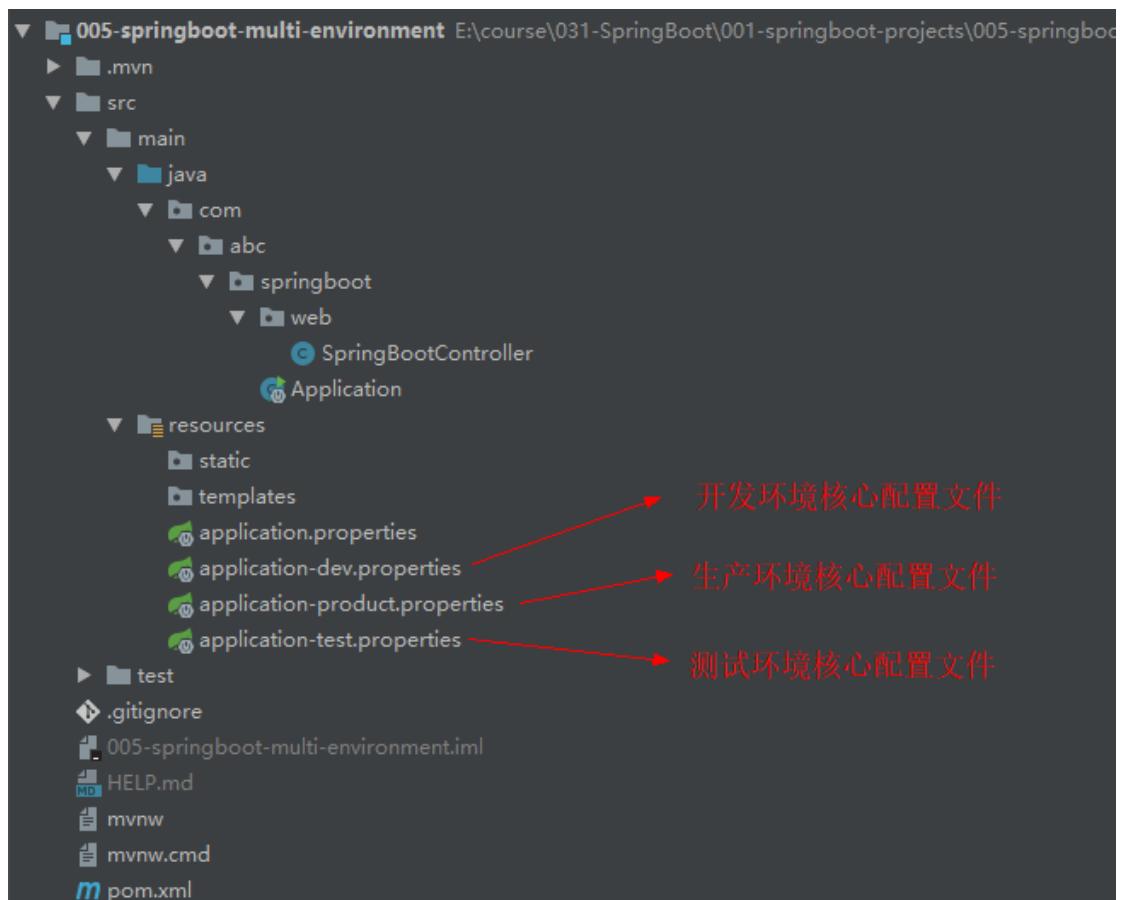


2.4.2 多环境配置

在实际开发的过程中，我们的项目会经历很多的阶段（开发->测试->上线），每个阶段的配置也会不同，例如：端口、上下文根、数据库等，那么这个时候为了方便在不同的环境之间切换，SpringBoot 提供了多环境配置，具体步骤如下

(9) 项目名称：005-springboot-multi-environment

为每个环境创建一个配置文件，命名必须以 **application-环境标识.properties|yml**



application-dev.properties

#开发环境

```
#设置内嵌 Tomcat 默认端口号
server.port=8080

#设置项目的上下文根
server.servlet.context-path=/005-springboot-multi-environment-dev
```

application-product.properties

```
#生产环境

#配置内嵌 Tomcat 默认端口号
server.port=80

#配置项目上下文根
server.servlet.context-path=/005-springboot-multi-environment-product
```

application-test.properties

```
#测试环境

#配置内嵌 Tomcat 端口号
server.port=8081

#配置项目的上下文根
server.servlet.context-path=/005-springboot-multi-environment-test
```

在总配置文件 application.properties 进行环境的激活

```
#SpringBoot 的总配置文件

#激活开发环境
spring.profiles.active=dev

#激活测试环境
spring.profiles.active=test

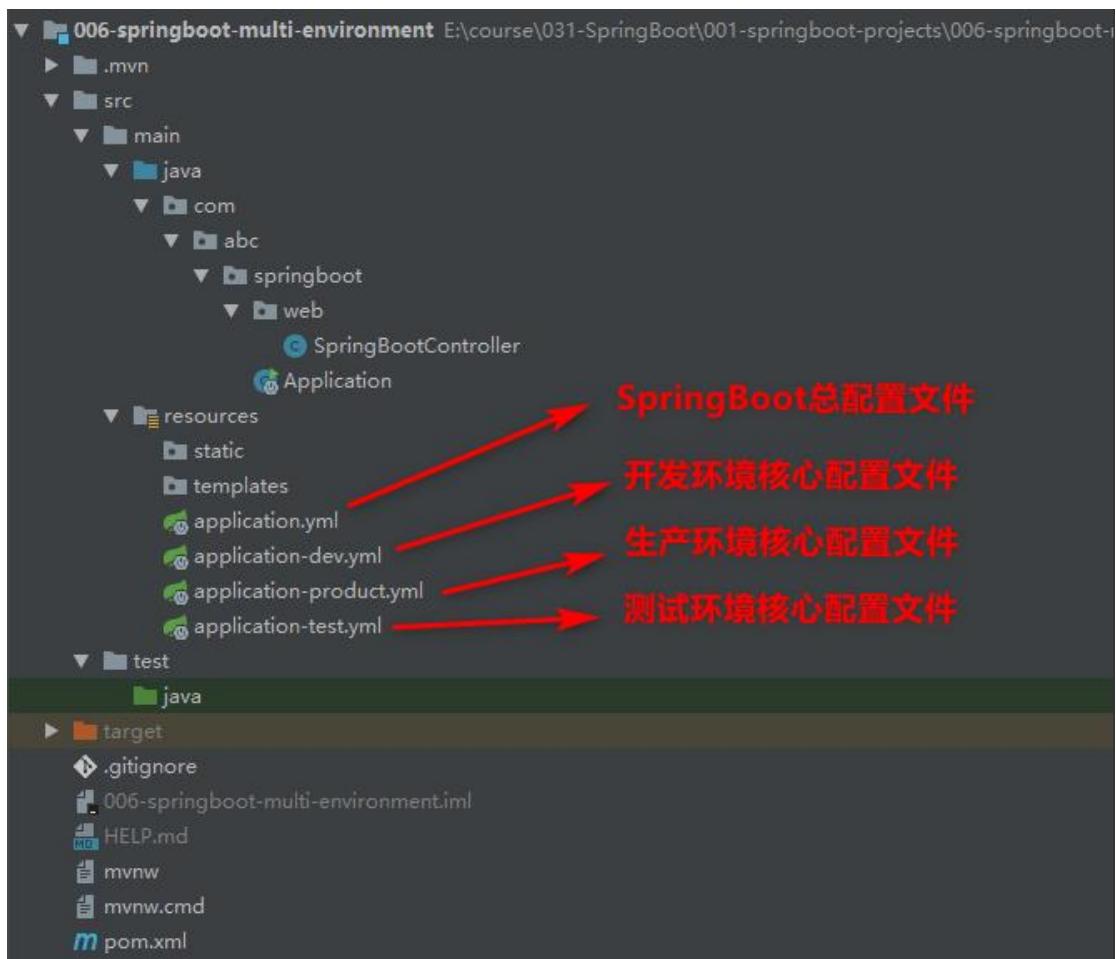
#激活生产环境
spring.profiles.active=product
```

等号右边的值和配置文件的环境标识名一致，可以更改总配置文件的配置，重新

运行 Application，查看启动的端口及上下文根

(10) 项目名称：006-springboot-multi-environment

为每个环境创建一个配置文件，命名必须以 **application-环境标识.properties|yml**



SpringBoot 总配置文件：application.yml

```
#springboot 总配置文件

#激活开发环境
#spring:
#  profiles:
#    active: dev
```

```
#激活测试环境
#spring:
#  profiles:
#    active: test

#激活生产环境
spring:
  profiles:
    active: product
```

开发环境配置文件: application-dev.yml

```
#设置开发环境配置

server:
  port: 8080 #设置 Tomcat 内嵌端口号
  servlet:
    context-path: /dev #设置上下文根
```

测试环境配置文件: application-test.yml

```
#设置测试环境配置

server:
  port: 9090
  servlet:
    context-path: /test
```

生产环境配置文件: application-product.yml

```
#设置生产环境配置

server:
  port: 80
  servlet:
    context-path: /product
```

2.4.3 Spring Boot 自定义配置

在 SpringBoot 的核心配置文件中，除了使用内置的配置项之外，我们还可以在自定义配置，然后采用如下注解去读取配置的属性值

(11) @Value 注解

A、项目名称：007-springboot-custom-configuration

用于逐个读取 application.properties 中的配置

案例演示

- 在核心配置文件 applicatin.properties 中，添加两个自定义配置项 school.name 和 website。在 IDEA 中可以看到这两个属性不能被 SpringBoot 识别，背景是桔色的



#设置内嵌Tomcat端口号
server.port=9090

#设置上下文根
server.servlet.context-path=/

school.name=abc
websit=http://www.baidu.com

在IDEA中可以看到这两个属性不能被SpringBoot识别，背景是桔色的

application.yml 格式配置文件

```
#设置端口号及上下文根
server:
  port: 9090
  servlet:
    context-path: /

school:
  name: ssm
websit: http://www.baidu.com
```

- 在 SpringBootController 中定义属性，并使用@Value 注解或者自定义配置值，并对

其方法进行测试

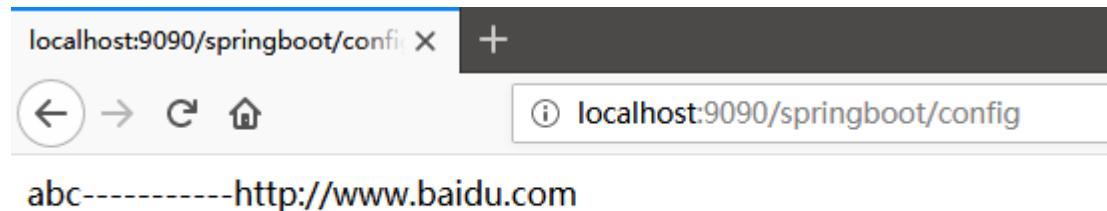
```
@Controller
public class SpringBootController {

    @Value("${school.name}")
    private String schoolName;

    @Value("${websit}")
    private String websit;

    @RequestMapping(value = "/springBoot/say")
    public @ResponseBody String say() {
        return schoolName + "-----" + websit;
    }
}
```

- 重新运行 Application，在浏览器中进行测试



(12) @ConfigurationProperties

项目名称：008-springboot-custom-configuration

将整个文件映射成一个对象，用于自定义配置项比较多的情况

案例演示

- 在 com.abc.springboot.config 包下创建 ConfigInfo 类，并为该类加上 Component 和 ConfigurationProperties 注解，并在 ConfigurationProperties 注解中添加属性 prefix，作用可以区分同名配置

```
@Component
```

```
@ConfigurationProperties(prefix = "school")
public class ConfigInfo {

    private String name;

    private String websit;

    public String getName() {
        return name;
    }

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

    public String getWebsit() {
        return websit;
    }

    public void setWebsit(String websit) {
        this.websit = websit;
    }
}
```

application.properties 配置文件

```
#设置内嵌 Tomcat 端口号
server.port=9090

#设置上下文根
server.servlet.context-path=/config

school.name=ssm
school.websit=http://www.baidu.com
```

application.yml 配置文件

```
server:
  port: 9090
  servlet:
    context-path: /config
```

```
school:  
  name: ABC  
  websit: http://www.baidu.com
```

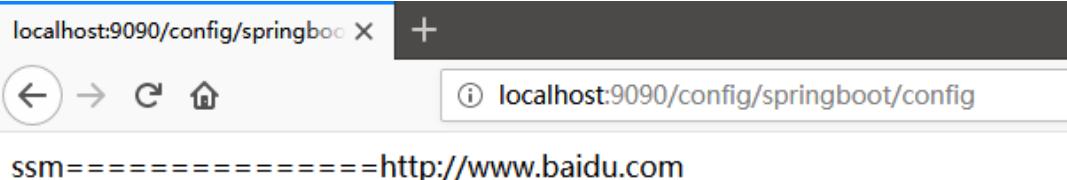
- 在 SpringBootController 中注入 ConfigInfo 配置类

```
@Autowired  
private ConfigInfo configInfo;
```

- 修改 SpringBootController 类中的测试方法

```
@RequestMapping(value = "/springBoot/config")  
public @ResponseBody String say() {  
    return configInfo.getName() + "======" + configInfo.getWebsit();  
}
```

- 重新运行 Application，在浏览器中进行测试

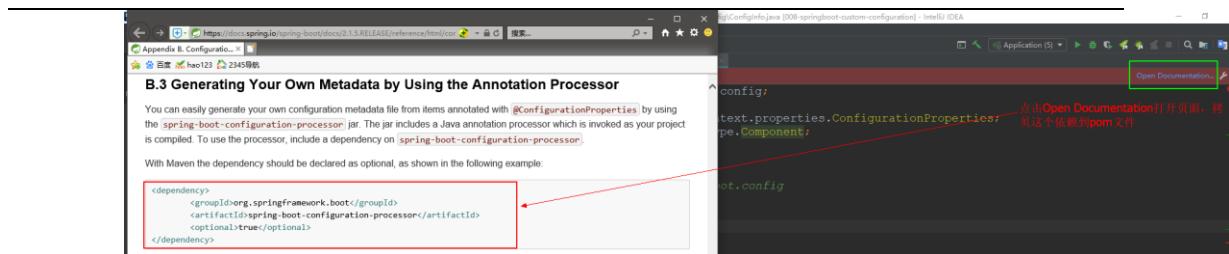


(13) 警告解决

- 在 ConfigInfo 类中使用了 ConfigurationProperties 注解后，IDEA 会出现一个警告，不影响程序的执行



- 点击 open documentation 跳转到网页，在网页中提示需要加一个依赖，我们将这个依赖拷贝，粘贴到 pom.xml 文件中

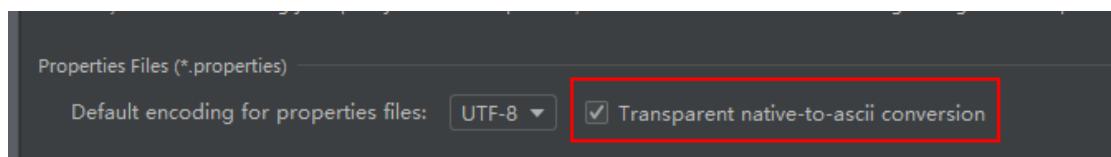


```
<!--解决使用@ConfigurationProperties注解出现警告问题-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

(14) 中文乱码

如果在 SpringBoot 核心配置文件中有中文信息，会出现乱码：

- 一般在配置文件中，不建议出现中文（注释除外）
- 如果有，可以先转化为 ASCII 码



(15) 友情提示

大家如果是从其它地方拷贝的配置文件，一定要将里面的空格删干净

2.5 Spring Boot 前端使用 JSP

项目名称：009-springboot-jsp

2.5.4 在 pom.xml 文件中配置以下依赖项

```
<!--引入 Spring Boot 内嵌的 Tomcat 对 JSP 的解析包，不加解析不了 jsp 页面-->
<!--如果只是使用 JSP 页面，可以只添加该依赖-->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>

<!--如果要使用 servlet 必须添加该以下两个依赖-->
<!-- servlet 依赖的 jar 包-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.3.1</version>
</dependency>

<!--如果使用 JSTL 必须添加该依赖-->
<!--jstl 标签依赖的 jar 包 start-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

2.5.5 在 pom.xml 的 build 标签中要配置以下信息

SpringBoot 要求 jsp 文件必须编译到指定的 META-INF/resources 目录下才能访问，否则访问不到。其实官方已经更建议使用模板技术（后面会讲模板技术）

```
<!--
    SpringBoot 要求 jsp 文件必须编译到指定的 META-INF/resources 目录下才能访问，否则访问不到。
    其它官方已经建议使用模版技术（后面会课程会单独讲解模版技术）
-->
<resources>
    <resource>
```

```
<!--源文件位置-->
<directory>src/main/webapp</directory>
<!--指定编译到 META-INF/resources，该目录不能随便写-->
<targetPath>META-INF/resources</targetPath>
<!--指定要把哪些文件编译进去，**表示 webapp 目录及子目录，*.*表示所有文件-->
<includes>
    <include>**/*.*</include>
</includes>
</resource>
</resources>
```

2.5.6 在 `application.properties` 文件配置 Spring MVC 的视图展示为 jsp，这里相当于 Spring MVC 的配置

```
#SpringBoot 核心配置文件
#指定内嵌 Tomcat 端口号
server.port=8090

#配置 SpringMVC 视图解析器
#其中： / 表示目录为 src/main/webapp
spring.mvc.view.prefix=/
spring.mvc.view.suffix=.jsp
```

集成完毕之后，剩下的步骤和我们使用 Spring MVC 一样

`application.yml` 格式的配置文件

```
#SpringBoot 核心配置文件
#指定内嵌 Tomcat 端口号
server:
  port: 8090
  servlet:
    context-path: /

#配置 SpringMVC 视图解析器
#其中： / 表示目录为 src/main/webapp
spring:
  mvc:
    view:
      prefix: /
      suffix: .jsp
```

2.5.7 在 com.abc.springboot.controller 包下创建 JspController 类，并编写代码

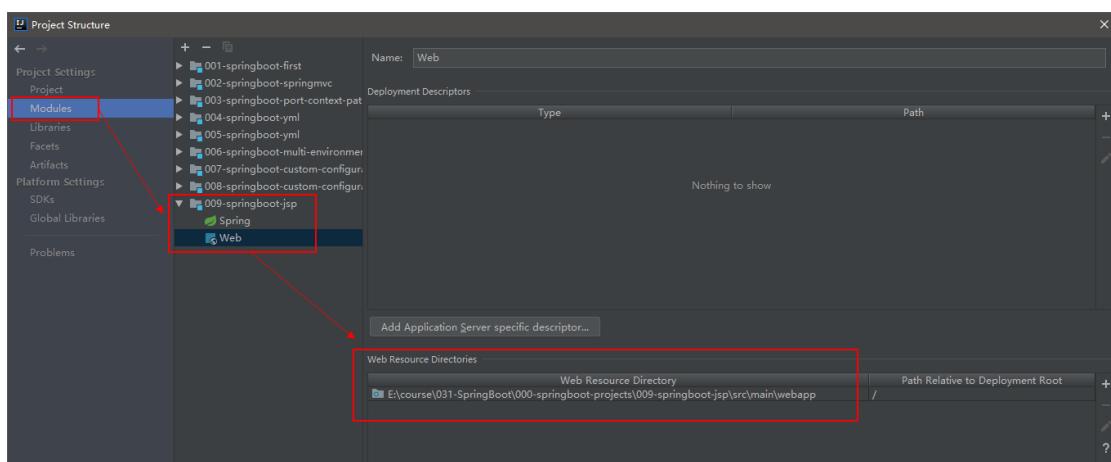
```
@Controller
public class SpringBootController {

    @RequestMapping(value = "/springBoot/jsp")
    public String jsp(Model model) {

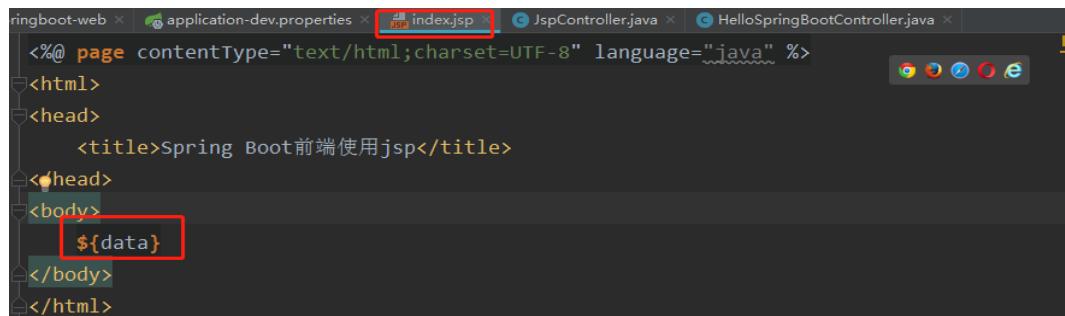
        model.addAttribute("data", "SpringBoot 前端使用 JSP 页面！");
        return "index";
    }
}
```

2.5.8 在 src/main 下创建一个 webapp 目录，然后在该目录下新建 index.jsp 页面

如果在 webapp 目录下右键，没有创建 jsp 的选项，可以在 Project Structure 中指定 webapp 为 Web Resource Directory

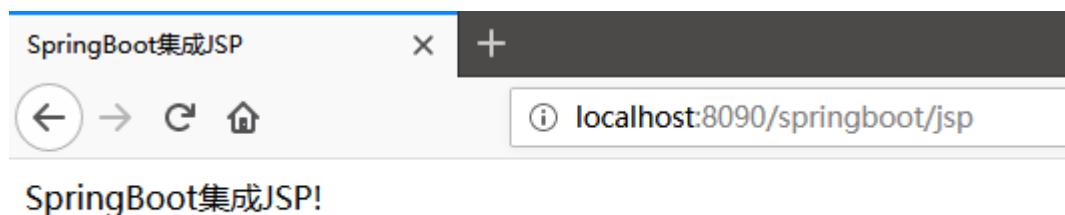


2.5.9 在 jsp 中获取 Controller 传递过来的数据



```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Spring Boot前端使用jsp</title>
</head>
<body>
    ${data}
</body>
</html>
```

2.5.10 重新运行 Application，通过浏览器访问测试



第3章 Spring Boot 框架 Web 开发

3.1 Spring Boot 集成 MyBatis

项目名称：010-springboot-web-mybatis

3.1.1 案例思路

通过 SpringBoot +MyBatis 实现对数据库学生表的查询操作

数据库参考：springboot.sql 脚本文件

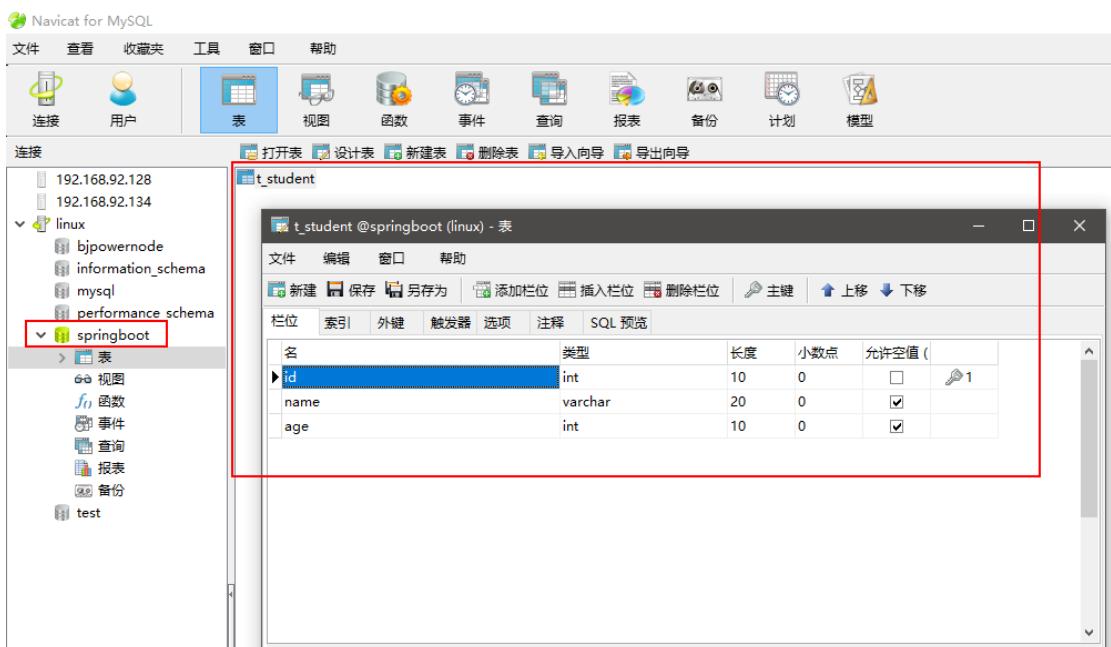
3.1.2 实现步骤

(1) 准备数据库

- 启动 Linux 系统上的 MySQL 服务器，通过 Navicat 连接

```
[root@localhost bin]# pwd  
/usr/local/mysql-5.7.24/bin  
[root@localhost bin]# ./mysqld_safe &
```

- 创建新的数据库 springboot，指定数据库字符编码为 utf-8

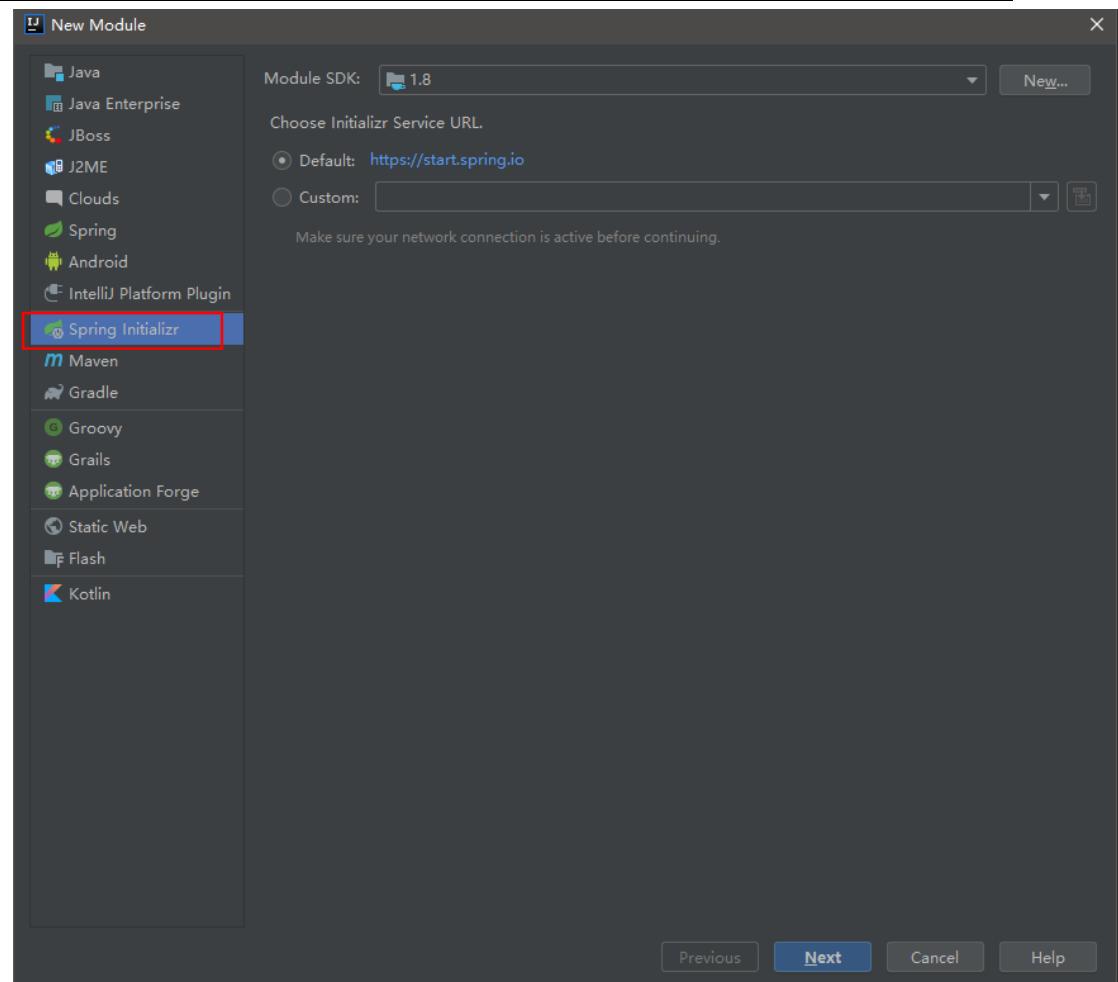


- 向表中插入数据

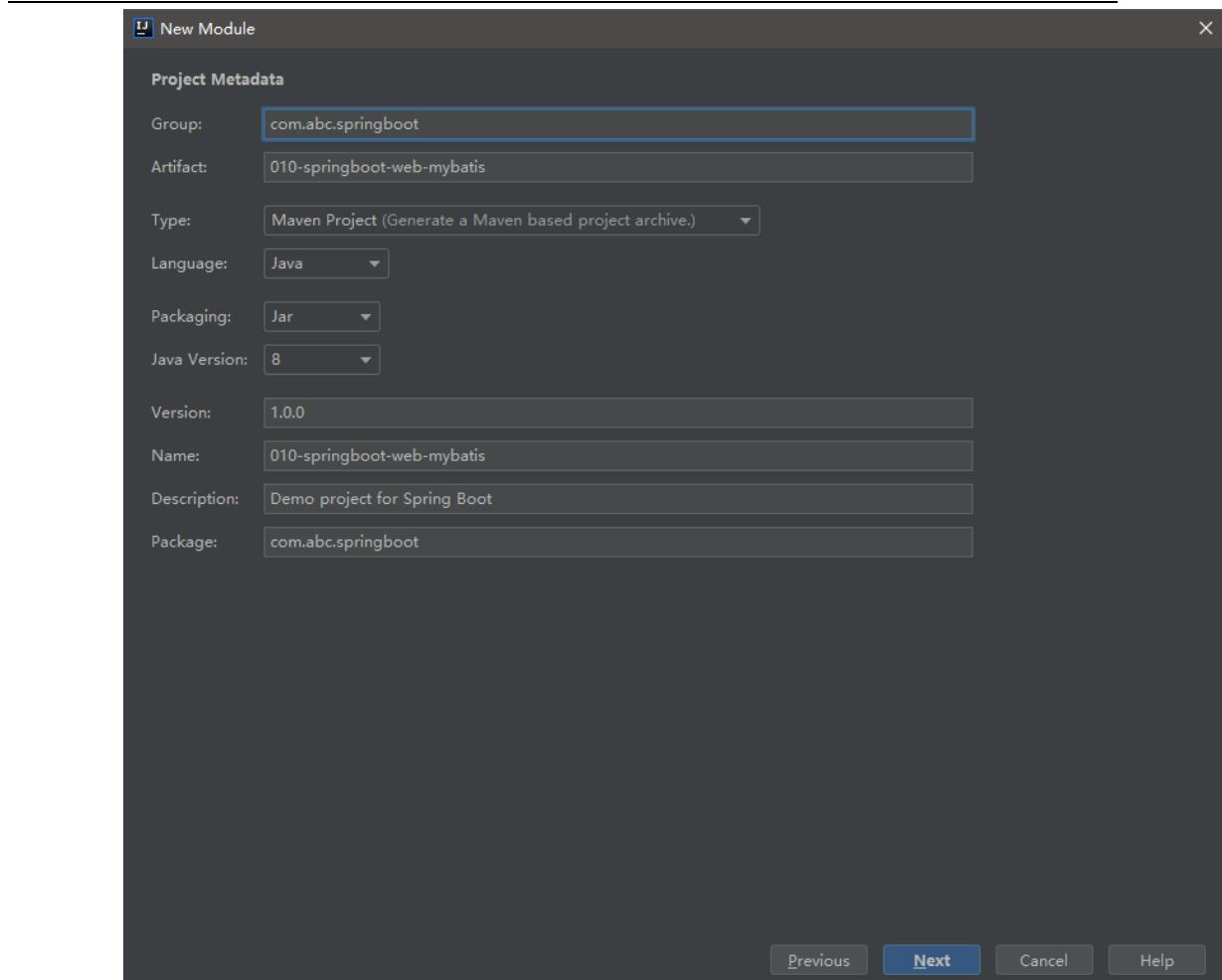
id	name	age
1	zs	15
2	ls	18
3	ww	20

(2) 创建 010-springboot-web-mybatis 项目

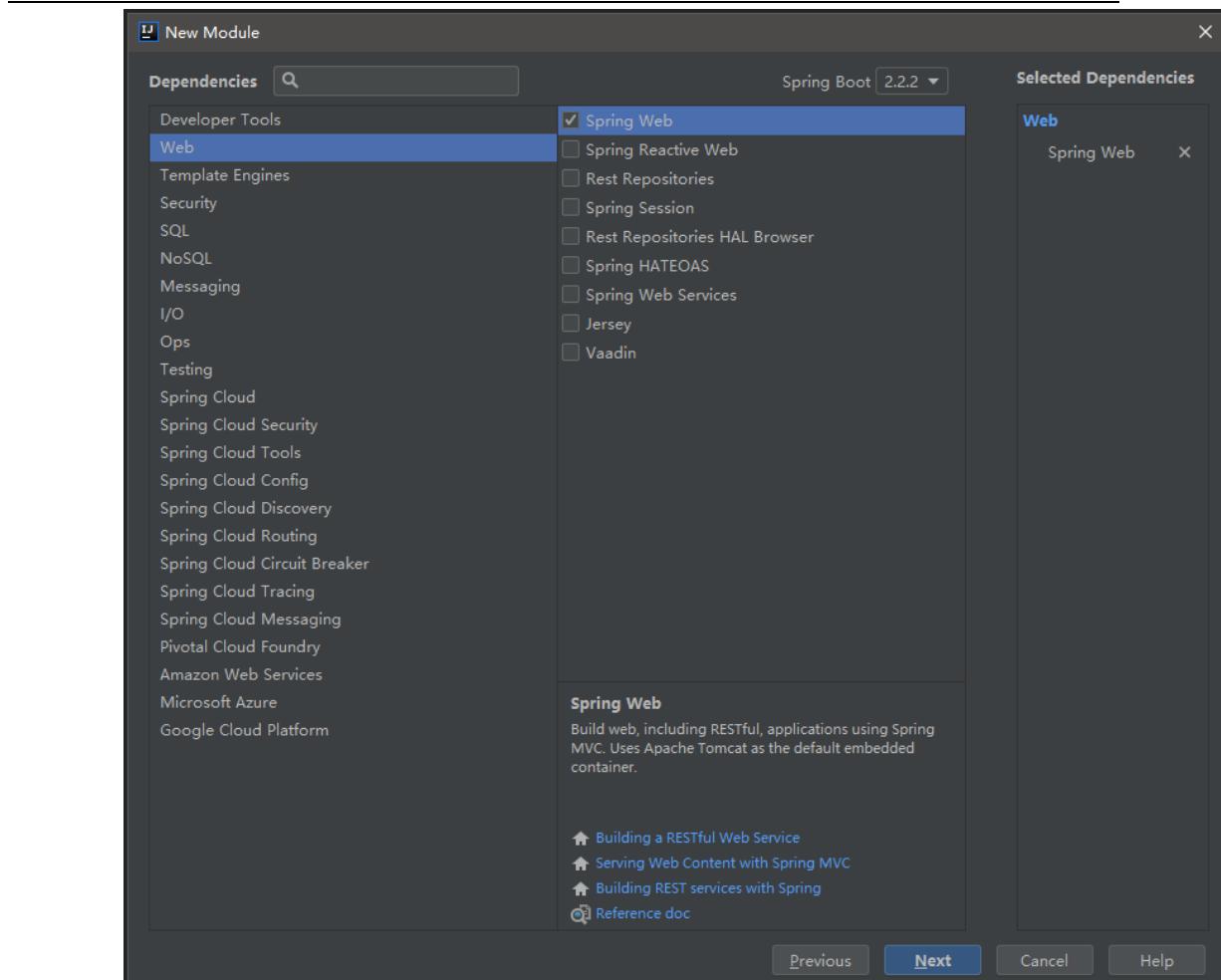
- 创建一个新的 SpringBoot 的 Module



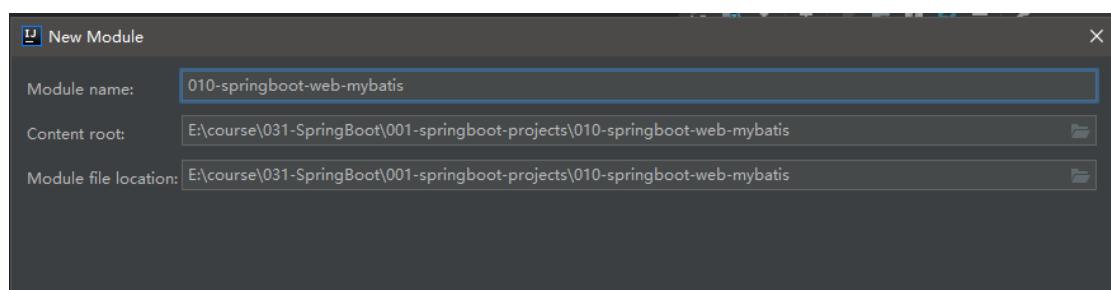
➤ 指定 GAV 坐标



- 选择 SpringBoot 版本以及 web 依赖



➤ 修改 Content root 以及 Module file location



(3) 在 pom.xml 中添加相关 jar 依赖

```
<!--MyBatis 整合 SpringBoot 的起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

```
<!--MySQL 的驱动依赖-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

(4) 在 Springboot 的核心配置文件 application.properties 中配置数据源

注意根据自己数据库的信息修改以下内容

```
#配置内嵌 Tomcat 端口号
server.port=9090

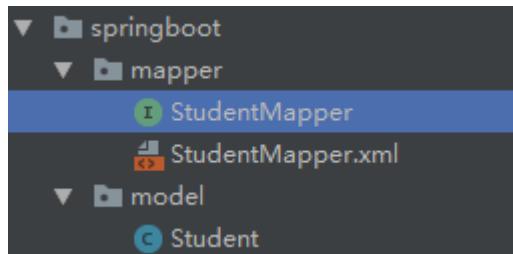
#配置项目上下文根
server.servlet.context-path=/010-springboot-web-mybatis

#配置数据库的连接信息
#注意这里的驱动类有变化
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=GMT%2B8

spring.datasource.username=root
spring.datasource.password=123456
```

(5) 开发代码

- 使用 Mybatis 反向工程生成接口、映射文件以及实体 bean，具体步骤参见附录 1



- 在 web 包下创建 StudentController 并编写代码

```
@Controller
public class StudentController {

    @Autowired
    private StudentService studentService;

    @RequestMapping(value = "/springBoot/student")
    public @ResponseBody Object student() {

        Student student = studentService.queryStudentById(1);

        return student;
    }
}
```

- 在 service 包下创建 service 接口并编写代码

```
public interface StudentService {

    /**
     * 根据学生标识获取学生详情
     * @param id
     * @return
     */
    Student queryStudentById(Integer id);
}
```

- 在 service.impl 包下创建 service 接口并编写代码

```
@Service
public class StudentServiceImpl implements StudentService {
```

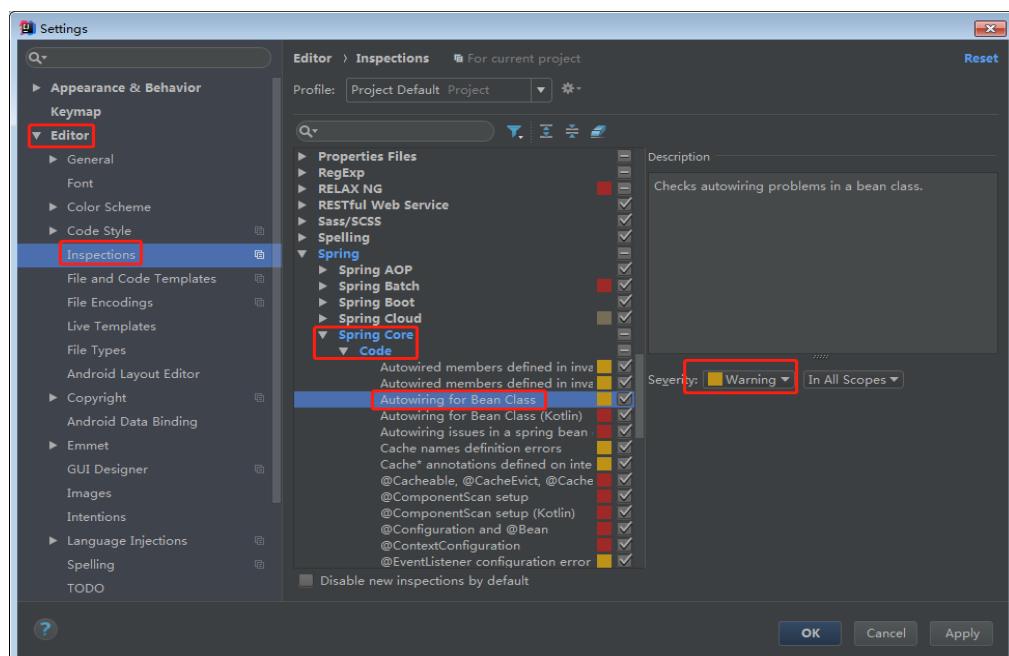
```

    @Autowired
    private StudentMapper studentMapper;

    @Override
    public Student queryStudentById(Integer id) {
        return studentMapper.selectByPrimaryKey(id);
    }
}

```

- 如果在 web 中导入 service 存在报错，可以尝试进行如下配置解决



- 在 Mybatis 反向工程生成的 StudentMapper 接口上加一个 Mapper 注解

@Mapper 作用： mybatis 自动扫描数据持久层的映射文件及 DAO 接口的关系

```

@Mapper
public interface StudentMapper {
}

```

- 注意：默认情况下，Mybatis 的 xml 映射文件不会编译到 target 的 class 目录下，所以我们需要在 pom.xml 文件中配置 resource

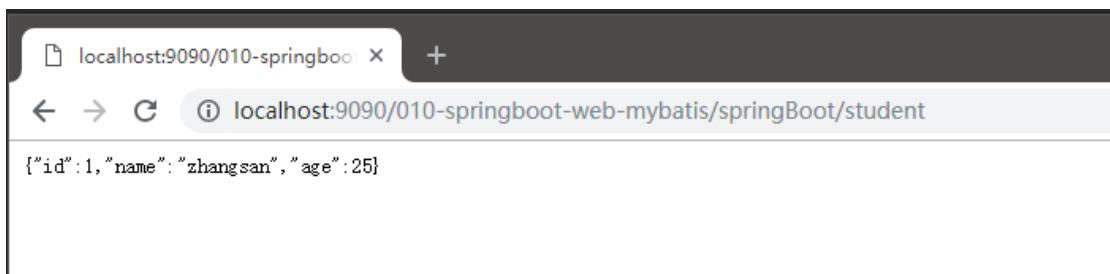
```

<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.xml</include>
        </includes>
    </resource>

```

```
</resource>
</resources>
```

(6) 启动 Application 应用，浏览器访问测试运行



3.1.3 DAO 其它开发方式

(7) 在 运 行 的 主 类 上 添 加 注 解 包 扫 描

@MapperScan("com.abc.springboot.mapper")

注释掉 StudentMapper 接口上的@Mapper 注解

```
//@Mapper
public interface StudentMapper {
```

在运行主类 Application 上加@MapperScan("com.abc.springboot.mapper")

```
@SpringBootApplication
@MapperScan("com.abc.springboot.mapper")
public class Application {
```

或

```
@SpringBootApplication
//Mybatis 提供的注解：扫描数据持久层的 mapper 映射配置文件，DAO 接口上就不用加@Mapper
//basePackages 通常指定到数据持久层包即可
@MapperScan(basePackages = "com.abc.springboot.mapper")
public class Application {
```

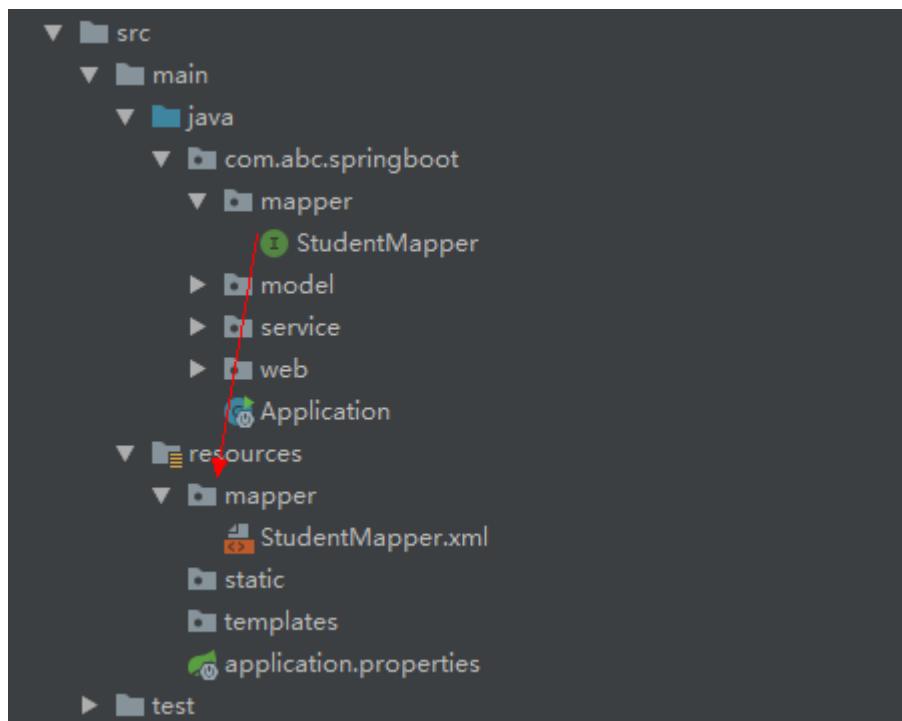
测试运行

(8) 将接口和映射文件分开

A、项目名称：011-springboot-web-mybatis

因为 SpringBoot 不能自动编译接口映射的 xml 文件，还需要手动在 pom 文件中指定，所以有的公司直接将映射文件直接放到 resources 目录下

- 在 resources 目录下新建目录 mapper 存放映射文件，将 StudentMapper.xml 文件移到 resources/mapper 目录下



- 在 application.properties 配置文件中指定映射文件的位置，这个配置只有接口和映射文件不在同一个包的情况下，才需要指定

```
# 指定 Mybatis 映射文件的路径
mybatis.mapper-locations=classpath:mapper/*.xml
```

3.2 Spring Boot 事务支持

Spring Boot 使用事务非常简单，底层依然采用的是 Spring 本身提供的事务管理

- 在入口类中使用注解 @EnableTransactionManagement 开启事务支持
- 在访问数据库的 Service 方法上添加注解 @Transactional 即可

3.2.1 案例思路

通过 SpringBoot +MyBatis 实现对数据库学生表的更新操作，在 service 层的方法中构建异常，查看事务是否生效

项目名称：012-springboot-web-mybatis-transacation

该项目是在 011 的基础上添加新增方法，在新增方法中进行案例的演示

3.2.2 实现步骤

(9) 在 StudentController 中添加更新学生的方法

```
@RequestMapping(value = "/springboot/modify")
public @ResponseBody Object modifyStudent() {

    int count = 0;
    try {
        Student student = new Student();
        student.setId(1);
        student.setName("Jack");
        student.setAge(33);
        count = studentService.modifyStudentById(student);
    } catch (Exception e) {
        e.printStackTrace();
        return "fail";
    }

    return count;
}
```

(10) 在 StudentService 接口中添加更新学生方法

```
/**
 * 根据学生标识更新学生信息
 * @param student
```

```
* @return
*/
int modifyStudentById(Student student);
```

(11) 在 **StudentServiceImpl** 接口实现类中对更新学生方法进行实现，并构建一个异常，同时在该方法上加**@Transactional** 注解

```
@Override
@Transactional //添加此注解说明该方法添加的事务管理
public int update(Student student) {

    int updateCount = studentMapper.updateByPrimaryKeySelective(student);

    System.out.println("更新结果: " + updateCount);

    //在此构造一个除数为 0 的异常，测试事务是否起作用
    int a = 10/0;

    return updateCount;
}
```

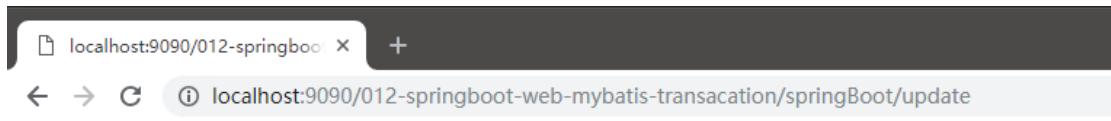
(12) 在 **Application** 类上加**@EnableTransactionManagement** 开启事务支持

@EnableTransactionManagement 可选，但是业务方法上必须添加**@Transactional** 事务才生效

```
@SpringBootApplication
@MapperScan(basePackages = "com.abc.springboot.mapper")
@EnableTransactionManagement //开启事务支持(可选项，但@Transactional 必须添加)
public class Application {
```

(13) 启动 Application，通过浏览器访问进行测试

浏览器



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed May 29 16:53:53 CST 2019

There was an unexpected error (type=Internal Server Error, status=500).
/ by zero

控制台

```
更新结果: 1
2019-05-29 16:53:53.776 ERROR 11720 ---
java.lang.ArithmetricException: / by zero
```

数据库表

id	name	age
1	zs	15
2	ls	18
3	ww	20

通过以上结果，说明事务起作用了

(14) 注释掉 StudentServiceImpl 上的@Transactional 测试

数据库的数据被更新

id	name	age
1	update-boot	99
2	ls	18
3	ww	20

3.3 Spring Boot 下的 Spring MVC

Spring Boot 下的 Spring MVC 和之前的 Spring MVC 使用是完全一样的，主要有以下注解

3.3.1 @Controller

Spring MVC 的注解，处理 http 请求

3.3.2 @RestController

Spring 4 后新增注解，是@Controller 注解功能的增强

是 @Controller 与@ResponseBody 的组合注解

如果一个 Controller 类添加了@RestController，那么该 Controller 类下的所有方法都相当于添加了@ResponseBody 注解

用于返回字符串或 json 数据

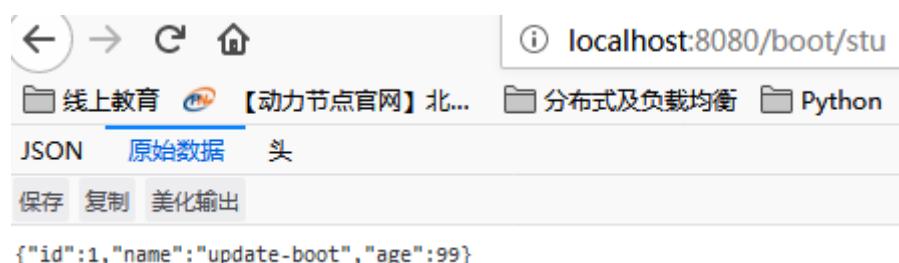
案例：

- 创建 MyRestController 类，演示@RestController 替代@Controller + @ResponseBody

```
@RestController
public class MyRestController {
    @Autowired
    private StudentService studentService;

    @RequestMapping("/boot/stu")
    public Object stu(){
        return studentService.getStudentById(1);
    }
}
```

- 启动应用，浏览器访问测试



3.3.3 @RequestMapping（常用）

支持 Get 请求，也支持 Post 请求

3.3.4 @GetMapping

RequestMapping 和 Get 请求方法的组合

只支持 Get 请求

Get 请求主要用于查询操作

3.3.5 @PostMapping

RequestMapping 和 Post 请求方法的组合

只支持 Post 请求

Post 请求主要用于新增数据

3.3.6 @PutMapping

RequestMapping 和 Put 请求方法的组合

只支持 Put 请求

Put 通常用于修改数据

3.3.7 @DeleteMapping

RequestMapping 和 Delete 请求方法的组合

只支持 Delete 请求

通常用于删除数据

3.3.8 综合案例

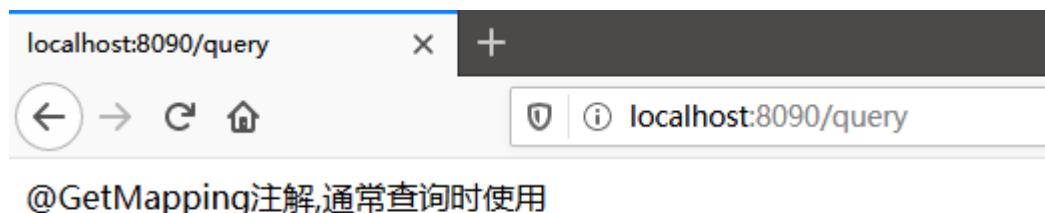
项目名称：013-springboot-springmvc 项目集成 springmvc

项目作用：演示常见的 SpringMVC 注解

(15) 创建一个 MVController，里面使用上面介绍的各种注解接收不同的请求

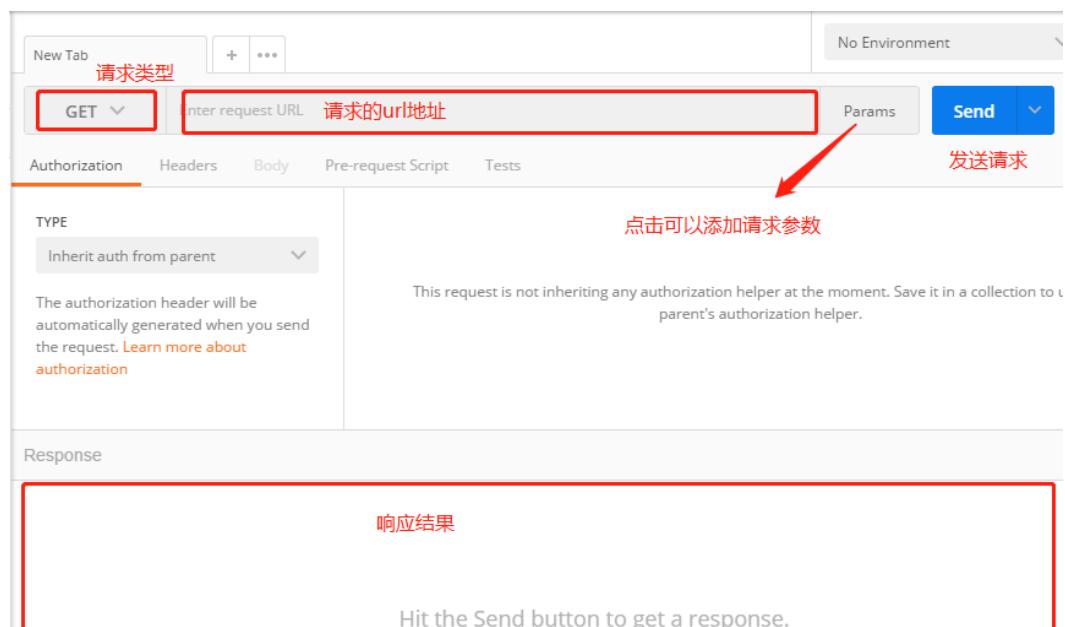
```
/**  
 * 该案例主要演示了使用 Spring 提供的不同注解接收不同类型的请求  
 */  
//RestController 注解相当于加了给方法加了@ResponseBody 注解, 所以是不能跳转页面的,  
只能返回字符串或者 json 数据  
@RestController  
public class MVController {  
  
    @GetMapping(value = "/query")  
    public String get() {  
        return "@GetMapping 注解, 通常查询时使用";  
    }  
  
    @PostMapping(value = "/add")  
    public String add() {  
        return "@PostMapping 注解, 通常新增时使用";  
    }  
  
    @PutMapping(value = "/modify")  
    public String modify() {  
        return "@PutMapping 注解, 通常更新数据时使用";  
    }  
  
    @DeleteMapping(value = "/remove")  
    public String remove() {  
        return "@DeleteMapping 注解, 通常删除数据时使用";  
    }  
}
```

(16) 启动应用，在浏览器中输入不同的请求进行测试



(17) Http 接口请求工具 Postman 介绍

因为通过浏览器输入地址，默认发送的只能是 get 请求，通过 Postman 工具，可以模拟发送不同类型的请求，并查询结果，在安装的时候，有些机器可能会需要安装 MicroSort .NET Framework



(18) 使用 Postman 对其它请求类型做个测试

3.4 Spring Boot 实现 RESTful

3.4.1 认识 RESTful

REST（英文：Representational State Transfer，简称 REST）

一种互联网软件架构设计的风格，但它并不是标准，它只是提出了一组客户端和服务器交互时的架构理念和设计原则，基于这种理念和原则设计的接口可以更简洁，更有层次，REST这个词，是 Roy Thomas Fielding 在他 2000 年的博士论文中提出的。

任何的技术都可以实现这种理念，如果一个架构符合 REST 原则，就称它为 RESTful 架构

比如我们要访问一个 http 接口：<http://localhost:8080/boot/order?id=1021&status=1>

采用 RESTful 风格则 http 地址为：<http://localhost:8080/boot/order/1021/1>

3.4.2 Spring Boot 开发 RESTful

Spring boot 开发 RESTful 主要是几个注解实现

(1) @PathVariable

获取 url 中的数据

该注解是实现 RESTful 最主要的一个注解

(2) @PostMapping

接收和处理 Post 方式的请求

(3) @DeleteMapping

接收 delete 方式的请求，可以使用 GetMapping 代替

(4) @PutMapping

接收 put 方式的请求，可以用 PostMapping 代替

(5) @GetMapping

接收 get 方式的请求

3.4.3 案例：使用 RESTful 风格模拟实现对学生的增删改查操作

项目名称：014-springboot-restful

该项目集成了 MyBatis、spring、SpringMVC，通过模拟实现对学生的增删改查操作

(6) pom.xml 文件添加内容如下

```
<dependencies>
    <!--SpringBoot 框架 web 项目起步依赖-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!--MyBatis 集成 SpringBoot 框架起步依赖-->
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>2.0.1</version>
    </dependency>

    <!--MySQL 驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
</dependencies>

<build>
    <!--指定配置资源的位置-->
    <resources>
```

```
<resource>
    <directory>src/main/java</directory>
    <includes>
        <include>**/*.xml</include>
    </includes>
</resource>
</resources>

<plugins>
    <!--mybatis 代码自动生成插件-->
    <plugin>
        <groupId>org.mybatis.generator</groupId>
        <artifactId>mybatis-generator-maven-plugin</artifactId>
        <version>1.3.6</version>
        <configuration>
            <!--配置文件的位置-->
            <configurationFile>GeneratorMapper.xml</configurationFile>
            <verbose>true</verbose>
            <overwrite>true</overwrite>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
</plugins>
</build>
```

(7) application.yml 核心配置文件

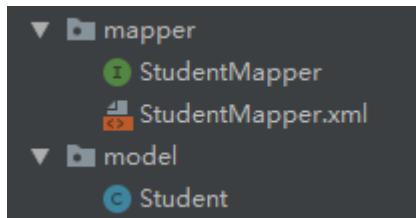
```
server:
  port: 8090 #设置Tomcat 内嵌端口号
  servlet:
    context-path: / #设置上下文根

#配置数据源
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url:
      jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBC
```

```
CompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=GMT%2B8
```

```
username: root  
password: 123456
```

(8) 通过逆向工程生成 DAO



(9) 创建 RESTfulController，并编写代码

```
@RestController  
public class RESTfulController {  
  
    /**  
     * 添加学生  
     * 请求地址:  
     * http://localhost:9090/014-springboot-restful/springBoot/student/wangpeng/23  
     * 请求方式: POST  
     * @param name  
     * @param age  
     * @return  
     */  
  
    @PostMapping(value = "/springBoot/student/{name}/{age}")  
    public Object addStudent(@PathVariable("name") String name,  
                            @PathVariable("age") Integer age) {  
  
        Map<String, Object> retMap = new HashMap<String, Object>();  
        retMap.put("name", name);  
        retMap.put("age", age);  
  
        return retMap;  
    }  
}
```

```
/**
 * 删除学生
 * 请求地址:
 http://localhost:9090/014-springboot-restful/springBoot/student/1
 * 请求方式: Delete
 * @param id
 * @return
 */
@DeleteMapping(value = "/springBoot/student/{id}")
public Object removeStudent(@PathVariable("id") Integer id) {

    return "删除的学生 id 为: " + id;
}

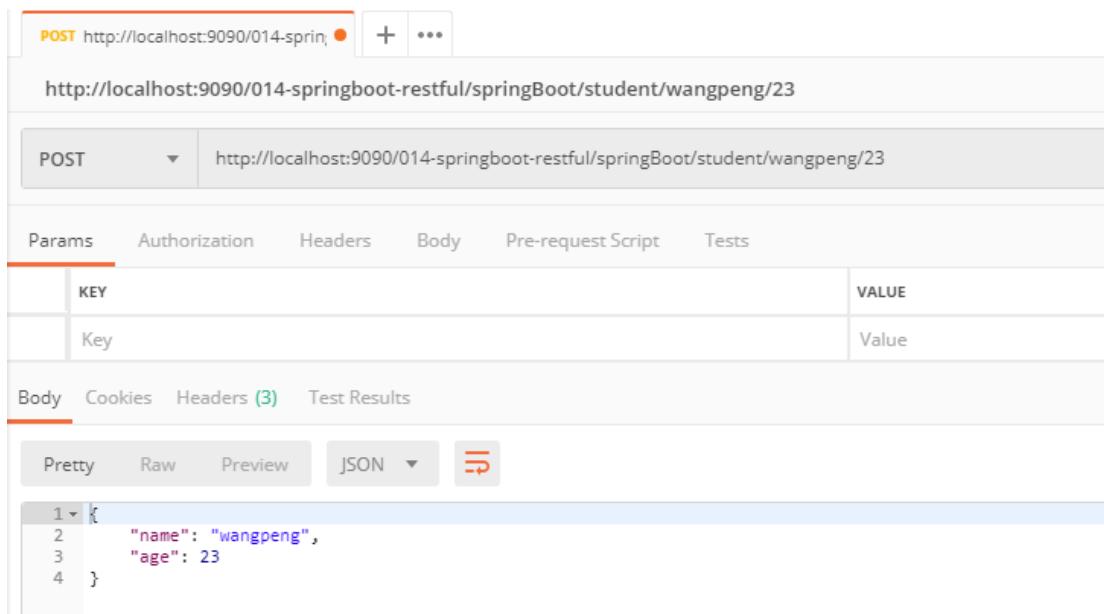
/**
 * 修改学生信息
 * 请求地址:
 http://localhost:9090/014-springboot-restful/springBoot/student/2
 * 请求方式: Put
 * @param id
 * @return
 */
@PutMapping(value = "/springBoot/student/{id}")
public Object modifyStudent(@PathVariable("id") Integer id) {

    return "修改学生的 id 为" + id;
}

/**
 * 查询学生信息
 * 请求地址:
 http://localhost:9090/014-springboot-restful/springBoot/student/1
 * @param id
 * @return
 */
@GetMapping(value = "/springBoot/student/{id}")
public Object queryStudent(@PathVariable("id") Integer id) {

    return "查询学生的 id 为" + id;
}
```

(10) 使用 Postman 模拟发送请求，进行测试



POST http://localhost:9090/014-spring

http://localhost:9090/014-springboot-restful/springBoot/student/wangpeng/23

POST http://localhost:9090/014-springboot-restful/springBoot/student/wangpeng/23

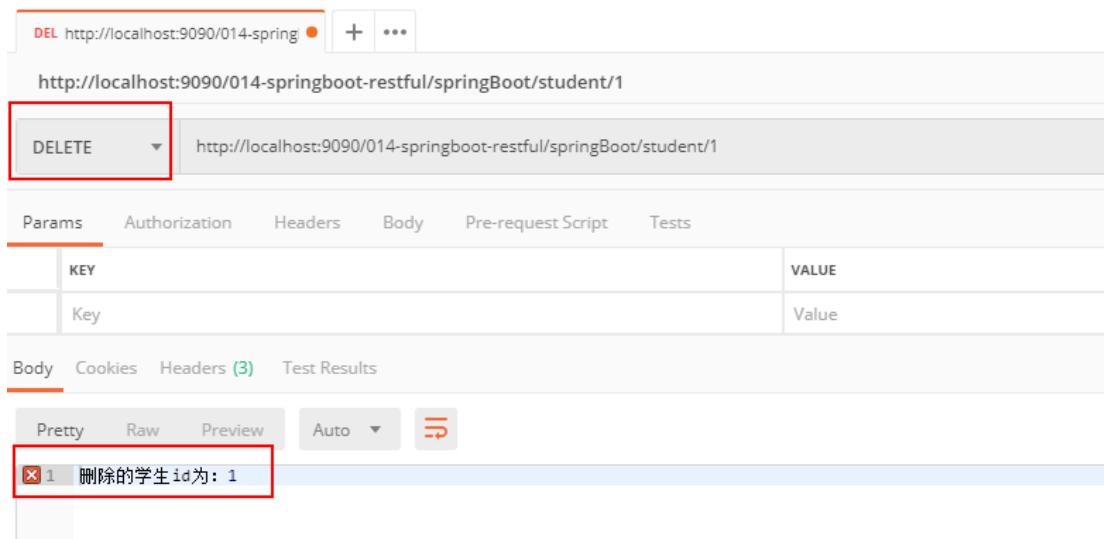
Params Authorization Headers Body Pre-request Script Tests

KEY	VALUE
Key	Value

Body Cookies Headers (3) Test Results

Pretty Raw Preview JSON

```
1 {
2   "name": "wangpeng",
3   "age": 23
4 }
```



DEL http://localhost:9090/014-spring

http://localhost:9090/014-springboot-restful/springBoot/student/1

DELETE http://localhost:9090/014-springboot-restful/springBoot/student/1

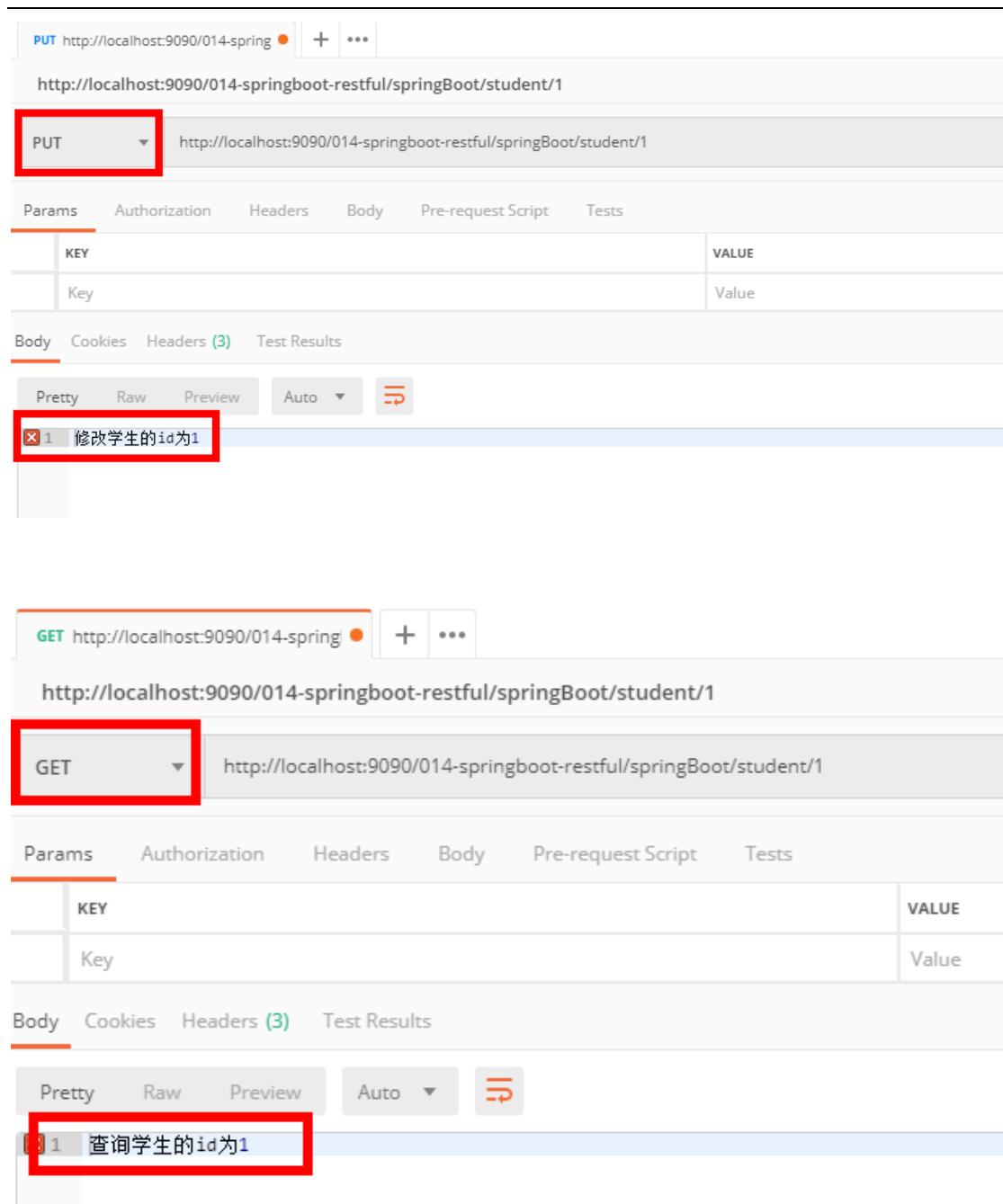
Params Authorization Headers Body Pre-request Script Tests

KEY	VALUE
Key	Value

Body Cookies Headers (3) Test Results

Pretty Raw Preview Auto

☒ 1 删除的学生 id 为：1



The screenshot shows two requests made using the Postman application:

PUT Request:

- Method: PUT (highlighted with a red box)
- URL: <http://localhost:9090/014-springboot-restful/springBoot/student/1>
- Body tab is selected.
- Body content: "修改学生的id为1" (Modify student's id to 1) (highlighted with a red box).

GET Request:

- Method: GET (highlighted with a red box)
- URL: <http://localhost:9090/014-springboot-restful/springBoot/student/1>
- Body tab is selected.
- Body content: "查询学生的id为1" (Query student's id 1) (highlighted with a red box).

(11) 总结：其实这里我们能感受到的好处

- 传递参数变简单了
- 服务提供者对外只提供了一个接口服务，而不是传统的 CRUD 四个接口

3.4.4 请求冲突的问题

项目名称：015-springboot-restful-url-conflict

- 改路径
- 改请求方式

创建 RESTfulController 类，结合 Postman 进行测试说明

```
@RestController
public class RESTfulController {

    /**
     * id: 订单标识
     * status: 订单状态
     * 请求路径:
     http://localhost:9090/015-springboot-restful-url-conflict/springBoot/order/1001
     * @param id
     * @param status
     * @return
     */
    @GetMapping(value = "/springBoot/order/{id}/{status}")
    public Object queryOrder(@PathVariable("id") Integer id,
                             @PathVariable("status") Integer status) {

        Map<String, Object> map = new HashMap<String, Object>();
        map.put("id", id);
        map.put("status", status);

        return map;
    }

    /**
     * id: 订单标识
     * status: 订单状态
     * 请求路径:
     http://localhost:9090/015-springboot-restful-url-conflict/springBoot/1/order/1001
     * @param id
     * @param status
    }
```

```
* @return
*/
@GetMapping(value = "/springBoot/{id}/order/{status}")
public Object queryOrder1(@PathVariable("id") Integer id,
                           @PathVariable("status") Integer status) {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("id", id);
    map.put("status", status);

    return map;
}

/**
 * id: 订单标识
 * status: 订单状态
 * 请求路径:
http://localhost:9090/015-springboot-restful-url-conflict/springBoot/1001
/order/1
 * @param id
 * @param status
 * @return
*/
@GetMapping(value = "/springBoot/{status}/order/{id}")
public Object queryOrder2(@PathVariable("id") Integer id,
                           @PathVariable("status") Integer status) {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("id", id);
    map.put("status", status);

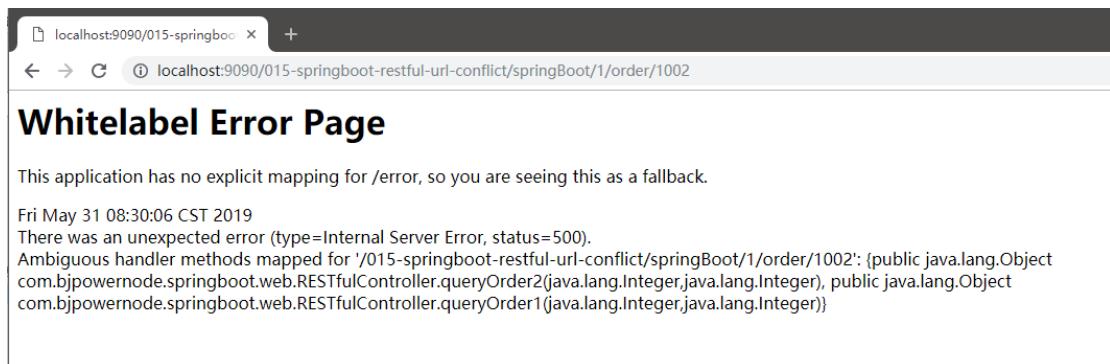
    return map;
}

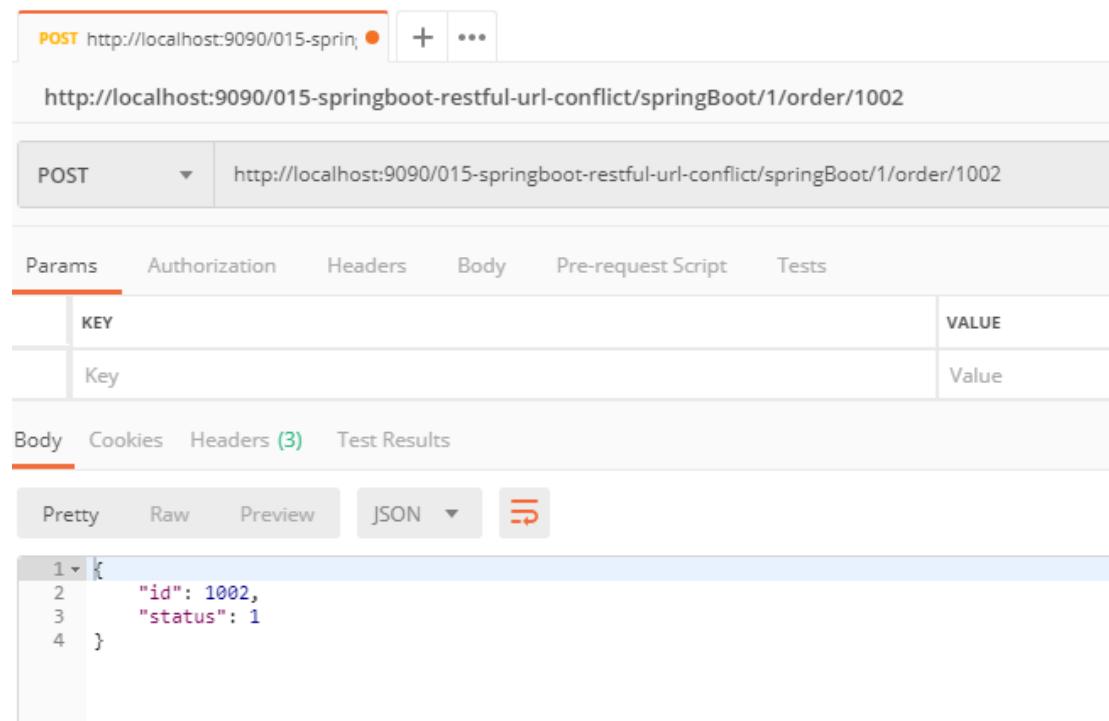
/**
 * id: 订单标识
 * status: 订单状态
 * 请求路径:
http://localhost:9090/015-springboot-restful-url-conflict/springBoot/1001
/order/1
 * @param id
 * @param status
 * @return
*/
```

```
@PostMapping(value = "/springBoot/{status}/order/{id}")
public Object queryOrder3(@PathVariable("id") Integer id,
                           @PathVariable("status") Integer status) {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("id", id);
    map.put("status", status);

    return map;
}

/**
 * query1 和 query2 两个请求路径会发生请求路径冲突问题
 * query3 与 query1 和 query2 发生请求冲突
 * 注意：虽然两个路径写法改变了，但是由于传递的两个参数都是 int 值，所以不知道该交给
哪个请求进行处理
 *      就会出现匹配模糊不清的异常，所以要想解决冲突，有两种方式：
 *      1. 修改请求路径
 *      2. 修改请求方式
 */
}
```





POST http://localhost:9090/015-springboot-restful-url-conflict/springBoot/1/order/1002

POST http://localhost:9090/015-springboot-restful-url-conflict/springBoot/1/order/1002

Params Authorization Headers Body Pre-request Script Tests

KEY	VALUE
Key	Value

Body Cookies Headers (3) Test Results

Pretty Raw Preview JSON

```
1 {
2     "id": 1002,
3     "status": 1
4 }
```

3.4.5 RESTful 原则

- 增 post 请求、删 delete 请求、改 put 请求、查 get 请求
- **请求路径不要出现动词**

例如：查询订单接口

/boot/order/1021/1 (推荐)

/boot/queryOrder/1021/1 (不推荐)

- **分页、排序等操作，不需要使用斜杠传参数**

例如：订单列表接口

/boot/orders?page=1&sort=desc

一般传的参数不是数据库表的字段，可以不采用斜杠

3.5 Spring Boot 集成 Redis

3.5.1 Spring Boot 集成 Redis

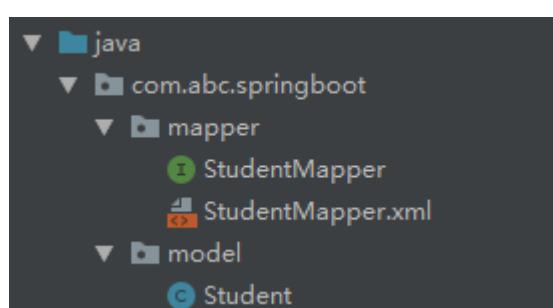
项目名称：016-springboot-redis

(1) 案例思路

完善根据学生 id 查询学生的功能，先从 redis 缓存中查找，如果找不到，再从数据库中查找，然后放到 redis 缓存中

(2) 实现步骤

B、首先通过 MyBatis 逆向工程生成实体 bean 和数据持久层



C、在 pom.xml 文件中添加 redis 依赖

```
<!-- 加载 spring boot redis 包 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

D、Spring Boot 核心配置文件

a、完整 application.properties 配置文件如下

```
#配置内嵌 Tomcat 端口号
server.port=9090

#配置项目上下文根
server.servlet.context-path=/016-springboot-redis

#配置连接 MySQL 数据库信息
spring.datasource.url=jdbc:mysql://192.168.92.134:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=GMT%2B8
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=123456

#配置 redis 连接信息
spring.redis.host=192.168.92.134
spring.redis.port=6379
spring.redis.password=123456
```

b、完整 application.yml 配置文件

```
server:
  port: 8090
  servlet:
    context-path: /
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url:
      jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=GMT%2B8
    username: root
    password: 123456
  redis:
    host: 192.168.92.134
    password: 123456
    port: 6379
```

E、启动 redis 服务

```
● 1192.168.92.134 x [ ]
[root@localhost src]# pwd
/usr/local/redis-4.0.12/src
[root@localhost src]# nohup ./redis-server ../redis.conf &
[1] 2803
[root@localhost src]# nohup: ignoring input and appending output to ' nohup.out'
[root@localhost src]# ps -ef | grep redis
root      2803  2723  0 15:32 pts/2    00:00:00 ./redis-server *:6379
root      2808  2723  0 15:32 pts/2    00:00:00 grep --color=auto redis
[root@localhost src]#
```

F、RedisController 类

```
@RestController
public class RedisController {

    @Autowired
    private StudentService studentService;

    /**
     * 请求地址:
     * http://localhost:9090/016-springboot-redis//springboot/allStudentCount
     * @param request
     * @return
     */
    @GetMapping(value = "/springboot/allStudentCount")
    public Object allStudentCount(HttpServletRequest request) {

        Long allStudentCount = studentService.queryAllStudentCount();

        return "学生总人数: " + allStudentCount;
    }
}
```

G、StudentService 接口

```
public interface StudentService {
```

```
    /**
     * 获取学生总人数
     * @return
     */
    Long queryAllStudentCount();
}
```

H、在 **StudentServiceImpl** 中注入 **RedisTemplate** 并修改根据 **id** 获取学生的方法

配置了上面的步骤，Spring Boot 将自动配置 RedisTemplate，在需要操作 redis 的类中注入 redisTemplate 即可。

注意：**Spring Boot 帮我们注入 RedisTemplate 类，泛型里面只能写 <String, String>、<Object, Object> 或者什么都不写**

```
@Service
public class StudentServiceImpl implements StudentService {

    @Autowired
    private StudentMapper studentMapper;

    @Autowired
    private RedisTemplate<Object, Object> redisTemplate;

    @Override
    public Long queryAllStudentCount() {

        //设置 redisTemplate 对象 key 的序列化方式
        redisTemplate.setKeySerializer(new StringRedisSerializer());

        //从 redis 缓存中获取总人数
        Long allStudentCount = (Long)
        redisTemplate.opsForValue().get("allStudentCount");

        //判断是否为空
        if (null == allStudentCount) {
            //去数据库查询，并存放到 redis 缓存中
        }
    }
}
```

```
        allStudentCount = studentMapper.selectAllStudentCount();

redisTemplate.opsForValue().set("allStudentCount", allStudentCount, 15,
TimeUnit.SECONDS);
    }
    return allStudentCount;
}
}
```

I、 StudentMapper 接口

```
/**
 * 获取学生总人数
 * @return
 */
Long selectAllStudentCount();
```

J、 StudentMapper 映射文件

```
<!--获取学生总人数-->
<select id="selectAllStudentCount" resultType="java.lang.Long">
    select
        count(*)
    from
        t_student
</select>
```

K、 启动类 Application

在 SpringBoot 启动类上添加扫描数据持久层的注解并指定扫描包

```
@SpringBootApplication
@MapperScan(basePackages = "com.abc.springboot.mapper") //扫描数据持久层
public class Application {
```

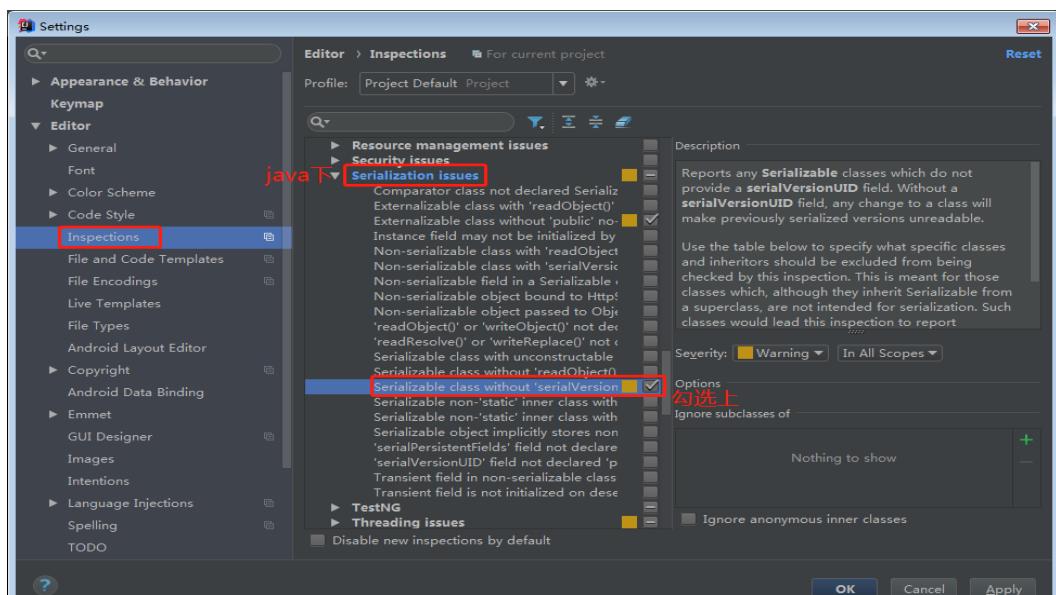
```

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

```

L、让 Student 类实现序列化接口（可选）

在类名上 Alt + 回车，如果没有提示生成序列化 id，那么需要做如下的配置



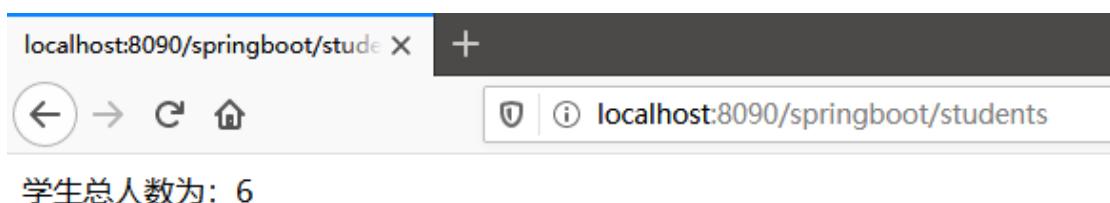
The screenshot shows the IntelliJ IDEA settings window under 'Editor > Inspections'. The 'Serializable issues' section is highlighted with a red box. A specific rule, 'Serializable class without serialVersionUID', has its checkbox checked, indicating it is enabled. The 'Severity' dropdown is set to 'Warning'. Below the checkboxes, there are sections for 'Options' and 'Ignore subclasses of'.

```

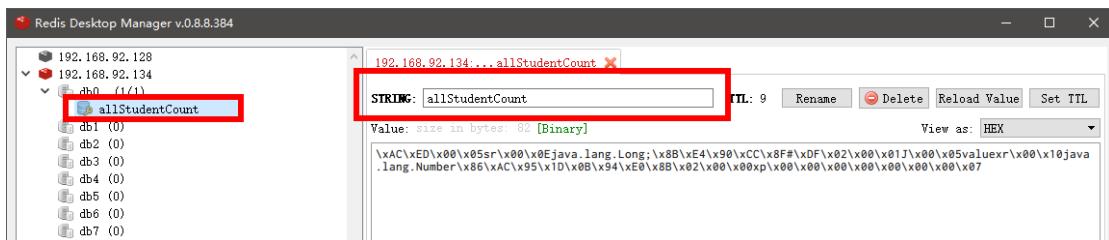
public class Student implements Serializable{
    private static final long serialVersionUID = -2337841098512469167L;
}

```

M、启动 SpringBoot 应用，访问测试



N、 打开 Redis Desktop Manager 查看 Redis 中的情况



3.6 Spring Boot 集成 Dubbo

阿里巴巴提供了 dubbo 集成 springBoot 开源项目，可以到 GitHub 上
<https://github.com/alibaba/dubbo-spring-boot-starter> 查看入门教程

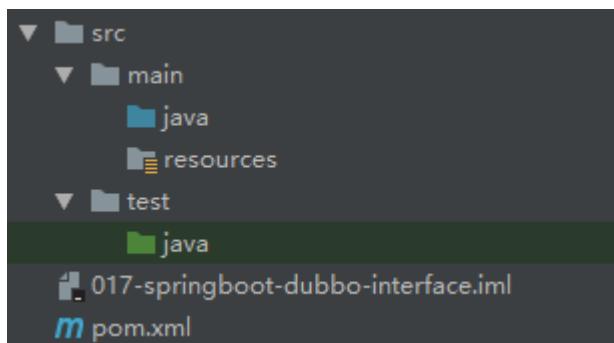
3.6.1 基本集成步骤

(1) 开发 Dubbo 服务接口

按照 Dubbo 官方开发建议，创建一个接口项目，该项目只定义接口和 model 类

项目名称: 017-springboot-dubbo-interface

A、 创建普通 Maven 项目，dubbo 服务接口工程



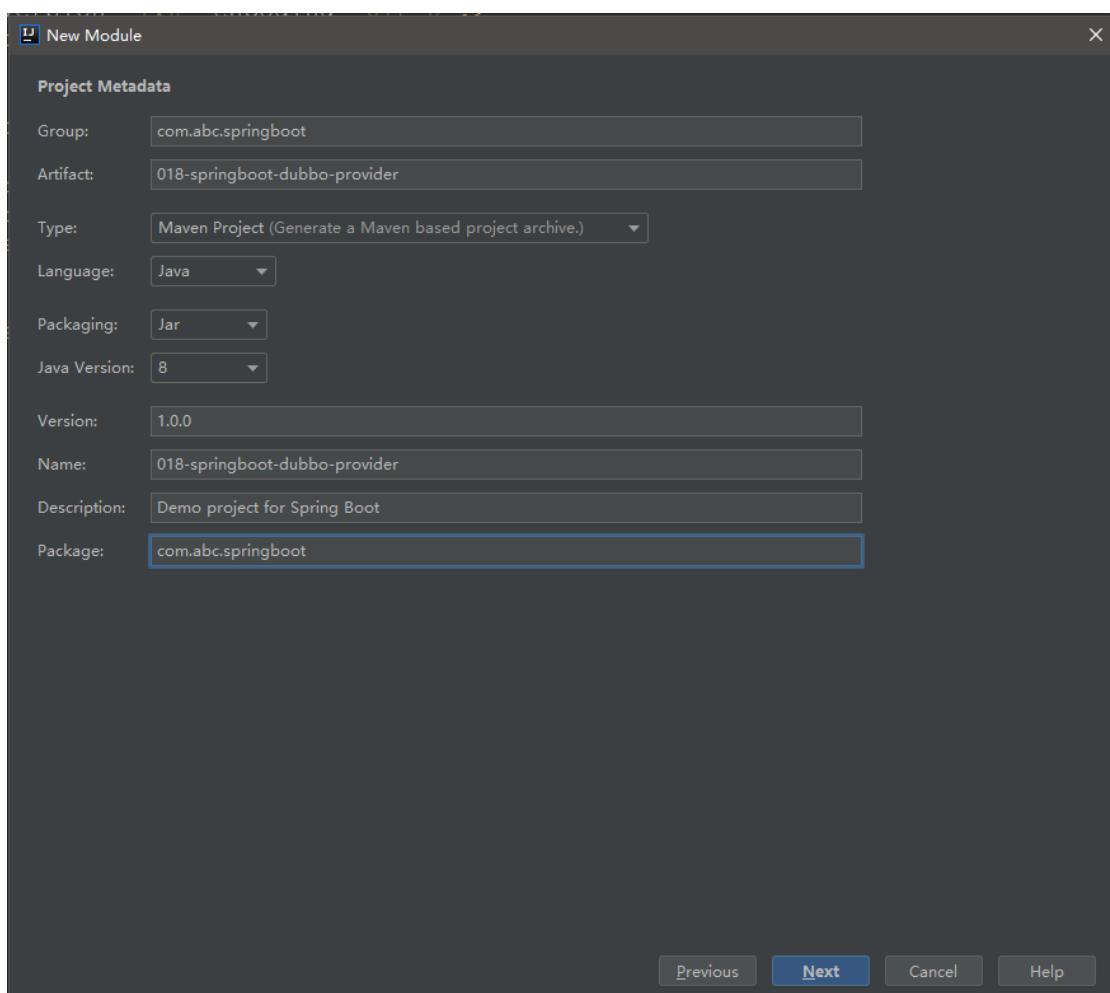
B、 创建 UserService 接口

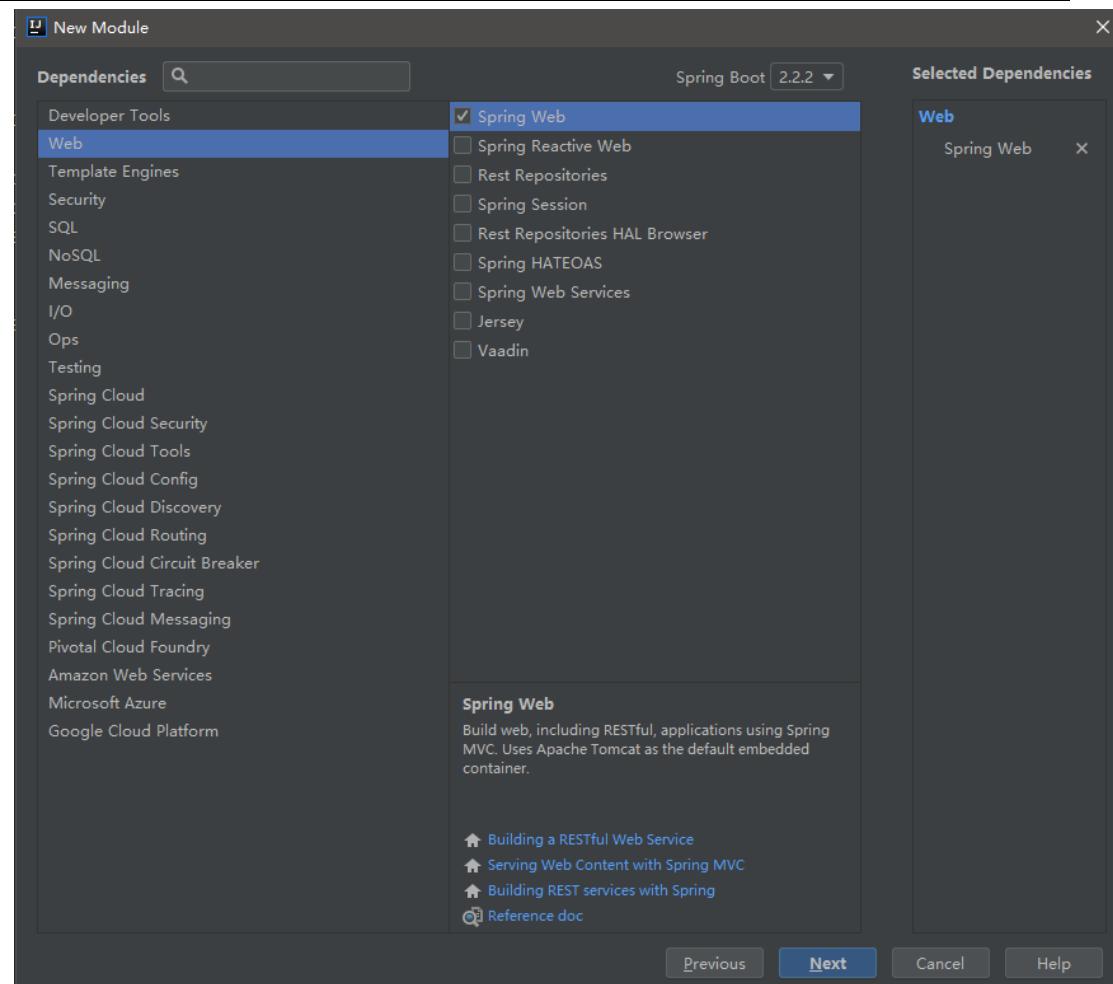
```
public interface UserService {  
    String say(String name);  
}
```

(2) 开发 Dubbo 服务提供者

项目名称: 018-springboot-dubbo-provider

A、 创建 SpringBoot 框架的 WEB 项目





B、加入 Dubbo 集成 SpringBoot 的起步依赖

```
<!--Dubbo 集成 SpringBoot 框架起步依赖-->
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

C、由于使用 **zookeeper** 作为注册中心，需加入 **zookeeper** 的客户端依赖

```
<!--Zookeeper 客户端依赖-->
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
```

D、加入 **MyBatis** 和 **MySQL** 依赖

```
<!--MyBatis 集成 SpringBoot 框架起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.1</version>
</dependency>

<!--MySQL 数据库驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

E、加入 **Dubbo** 接口依赖

```
<!--Dubbo 接口工程-->
<dependency>
    <groupId>com.abc.springboot</groupId>
    <artifactId>017-springboot-dubbo-interface</artifactId>
    <version>1.0.0</version>
</dependency>
```

F、 在 Springboot 的核心配置文件

(1) application.properties 格式-配置 dubbo

```
#设置内嵌 Tomcat 端口号
server.port=8090
#设置上下文根
server.servlet.context-path=/

#配置数据源
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=123456

#配置 dubbo 的服务提供者信息
#服务提供者应用名称(必须写, 且不能重复)
spring.application.name=springboot-dubbo-provider
#设置当前工程为服务提供者
spring.dubbo.server=true
#设置注册中心
spring.dubbo.registry=zookeeper://localhost:2181
```

注意: Dubbo 的注解都是自定义的注解, 由我们添加的 Dubbo 依赖中的类进行处理编写

dubbo 配置是没有提示的

(2) application.yml 格式-配置 dubbo

```
server:
  port: 8090
  servlet:
    context-path: /

spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url:
      jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false
```

```
lease&serverTimezone=GMT%2B8
    username: root
    password: 123456

application:
    name: springboot-dubbo-provider
dubbo:
    server: true
    registry: zookeeper://localhost:2181
```

G、 编写 Dubbo 的接口实现类

```
package com.abc.springboot.service.impl;

import com.abc.springboot.mapper.StudentMapper;
import com.abc.springboot.model.Student;
import com.abc.springboot.service.StudentService;
import com.alibaba.dubbo.config.annotation.Service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
@Service(interfaceName =
"com.abc.springboot.service.StudentService", version = "1.0.0", timeout
= 15000)
public class StudentServiceImpl implements StudentService {

    @Autowired
    private StudentMapper studentMapper;

    @Override
    public Student queryStudent(Integer id) {
        return studentMapper.selectByPrimaryKey(id);
    }
}
```

H、 在 SpringBoot 入口程序类上加开启 Dubbo 配置支持注解

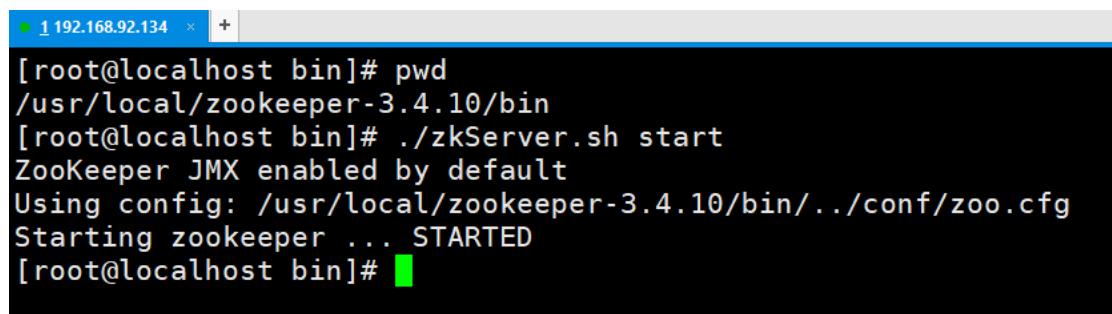
```
@SpringBootApplication
@MapperScan(basePackages = "com.abc.springboot.mapper")
```

```
@EnableDubboConfiguration //开启 Dubbo 配置
public class Application {

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

I、启动 Zookeeper 服务

- 启动 Linux 服务器上的 Zookeeper



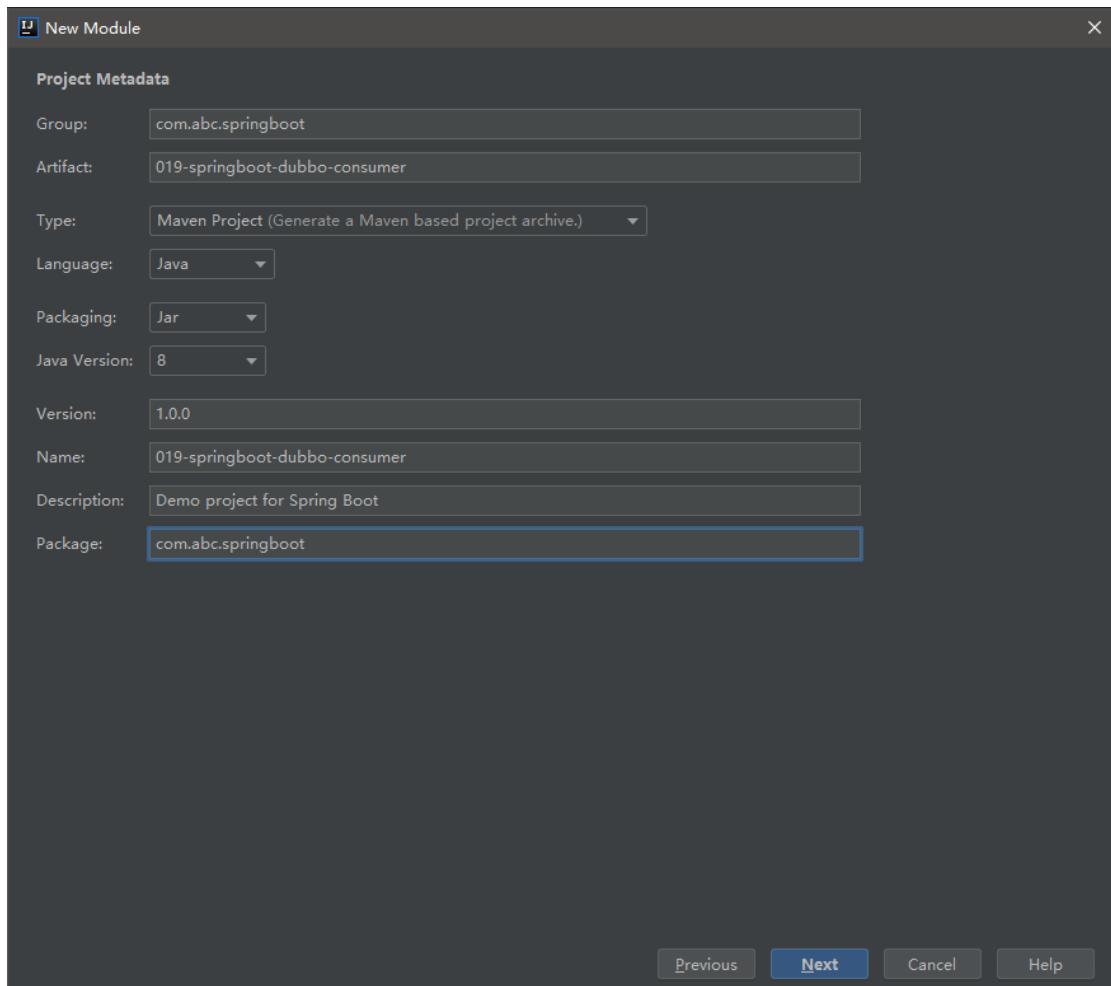
```
1 192.168.92.134 x + 
[root@localhost bin]# pwd
/usr/local/zookeeper-3.4.10/bin
[root@localhost bin]# ./zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /usr/local/zookeeper-3.4.10/bin/..../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@localhost bin]#
```

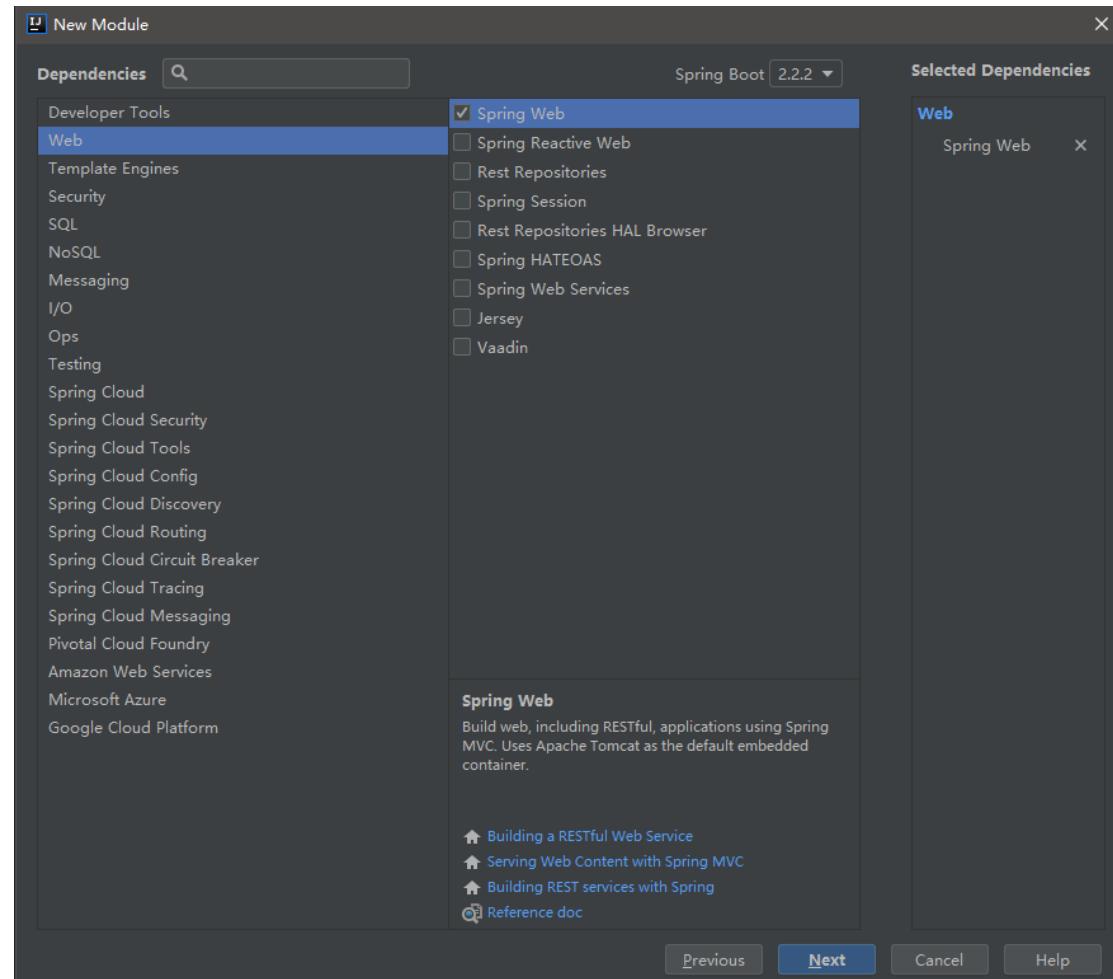
- 启动服务提供者项目主程序

(3) 开发 Dubbo 服务消费者

项目名称: 019-springboot-dubbo-consumer

A、 创建 SpringBoot 框架的 web 项目





B、加入 Dubbo 集成 SpringBoot 框架的起步依赖

```
<!--Dubbo 集成 SpringBoot 框架起步依赖-->
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

C、由于使用 **zookeeper** 作为注册中心，需加入 **zookeeper** 的客户端依赖

```
<!--Zookeeper 客户端依赖-->
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
```

D、加入 **Dubbo** 接口依赖

```
<!--dubbo 接口依赖-->
<dependency>
    <groupId>com.abc.springboot</groupId>
    <artifactId>017-springboot-dubbo-interface</artifactId>
    <version>1.0.0</version>
</dependency>
```

E、**Springboot** 的核心配置

(1) **application.properties** 格式-配置 **dubbo**

```
#设置内嵌 Tomcat 端口号
server.port=8080
#设置上下文根
server.servlet.context-path=/

#设置 dubbo 配置
#设置服务消费者名称
spring.application.name=springboot-dubbo-consumer
#配置 dubbo 注册中心
spring.dubbo.registry=zookeeper://localhost:2181
```

(2) **application.yml** 格式-配置 **dubbo**

```
server:  
  port: 8080  
  servlet:  
    context-path: /  
  
spring:  
  application:  
    name: springboot-dubbo-consumer  
dubbo:  
  registry: zookeeper://localhost:2181
```

F、 编写一个 Controller 类，调用远程的 Dubbo 服务

```
package com.abc.springboot.web;  
  
import com.abc.springboot.model.Student;  
import com.abc.springboot.service.StudentService;  
import com.alibaba.dubbo.config.annotation.Reference;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class StudentController {  
  
    @Reference(interfaceName =  
"com.abc.springboot.service.StudentService", version = "1.0.0", check =  
false)  
    private StudentService studentService;  
  
    @RequestMapping(value = "/student")  
    public Object queryStudent(Integer id) {  
  
        Student student = studentService.queryStudent(id);  
  
        return student;  
    }  
}
```

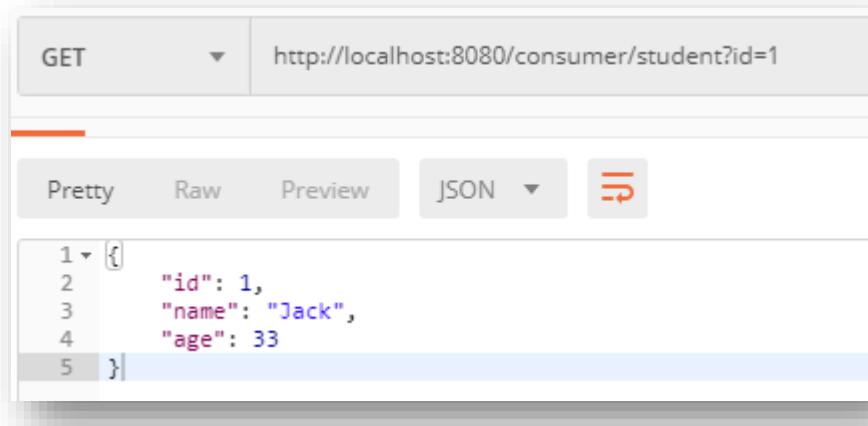
G、在 SpringBoot 入口程序类上加开启 Dubbo 配置支持注解

```
@SpringBootApplication
@EnableDubboConfiguration //开启 Dubbo 配置
public class Application {

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

H、测试

- 启动服务消费者项目主程序

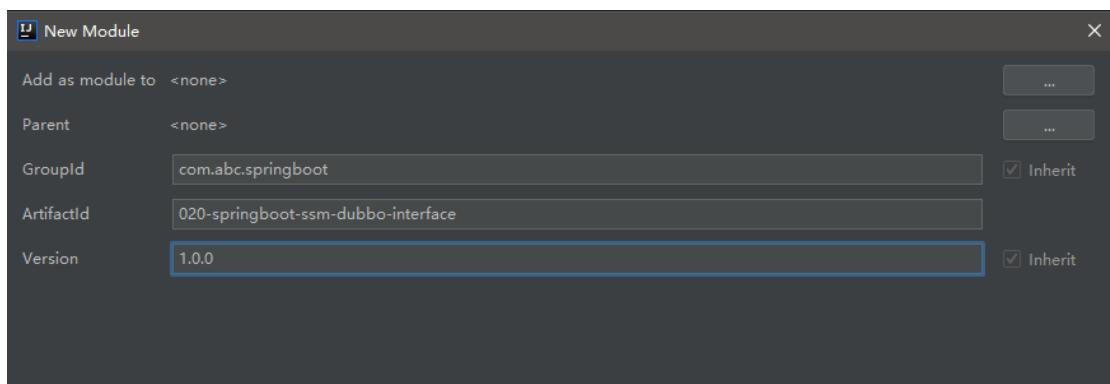


3.6.2 SpringBoot 集成 SSM+Dubbo+Redis

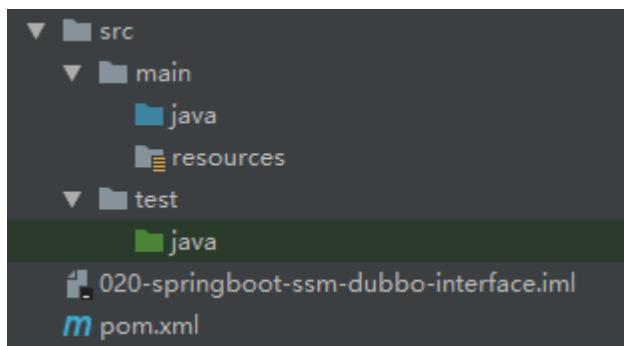
该实例目的是为了让同学们快速学会使用 SpringBoot 搭建基于 Dubbo 的 SSM 分布式框架及集成 Redis 服务。

(1) 创建 Maven Java 工程，Dubbo 接口工程

项目名称：020-springboot-ssm-dubbo-exterface

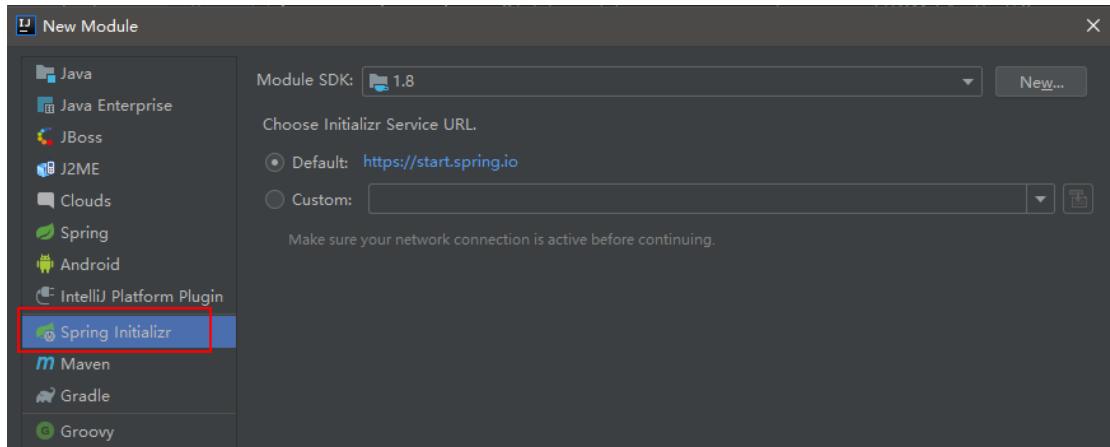


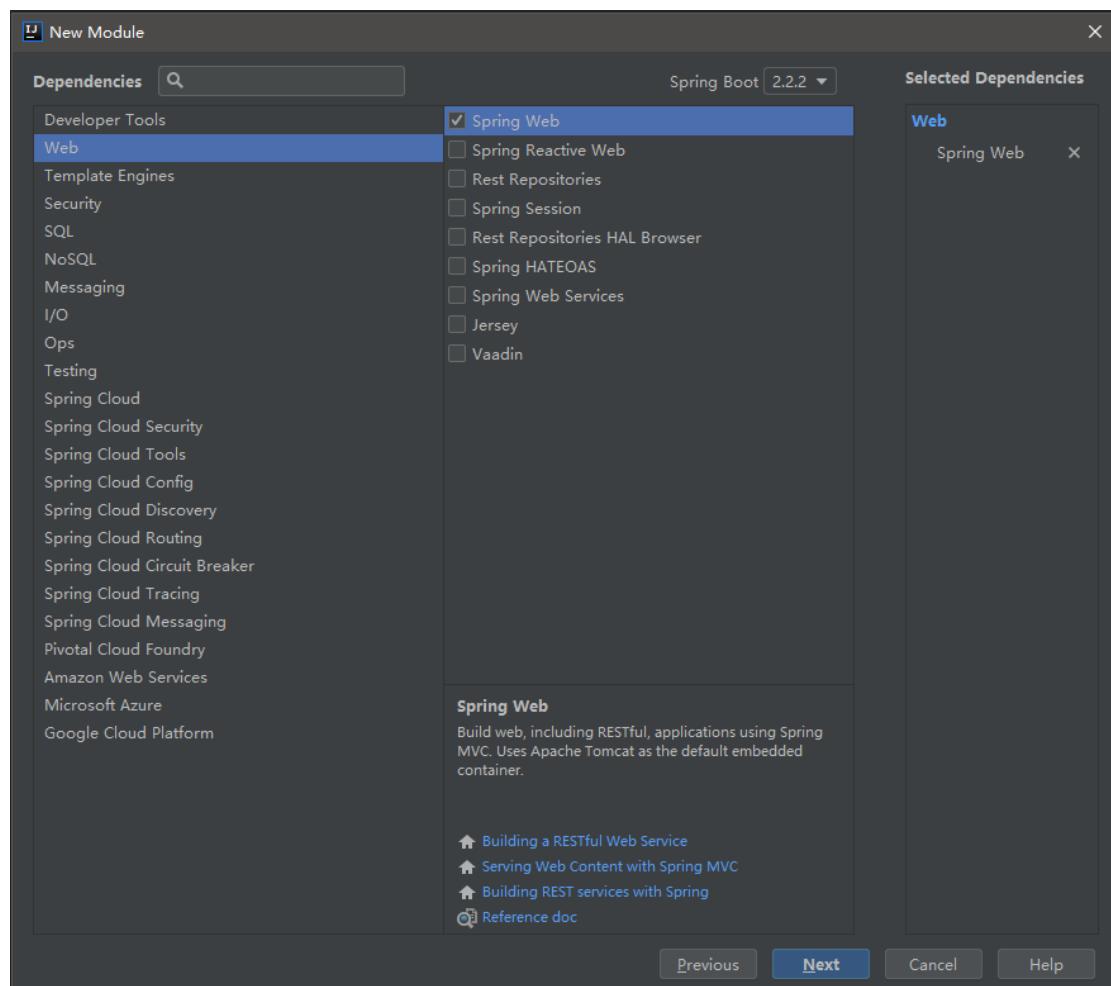
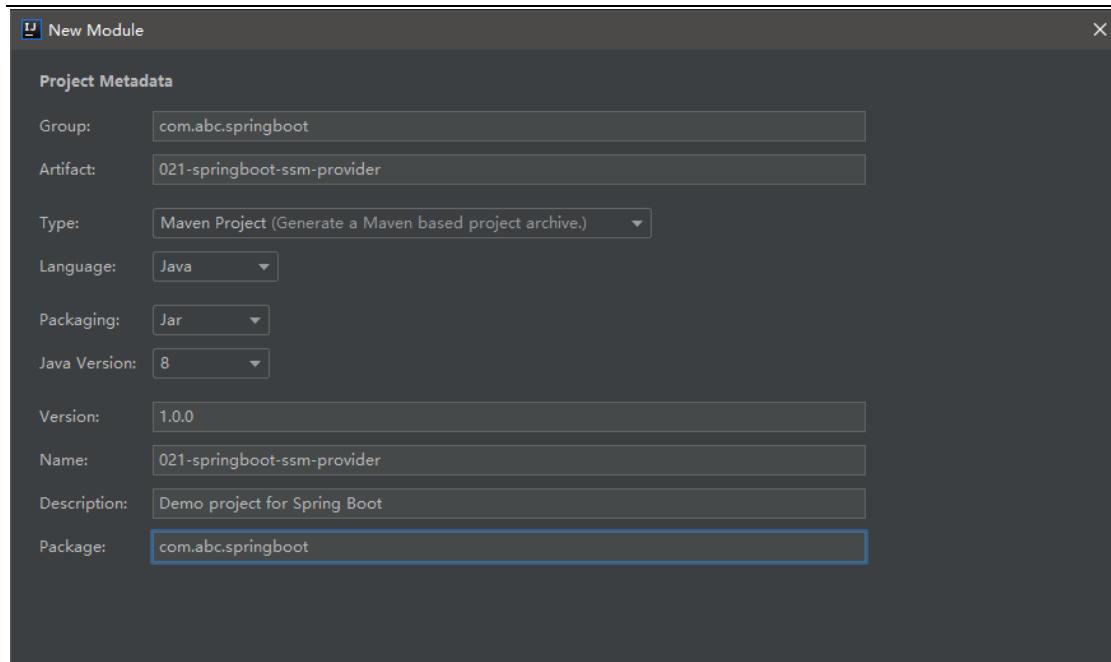
(2) 接口工程项目结构



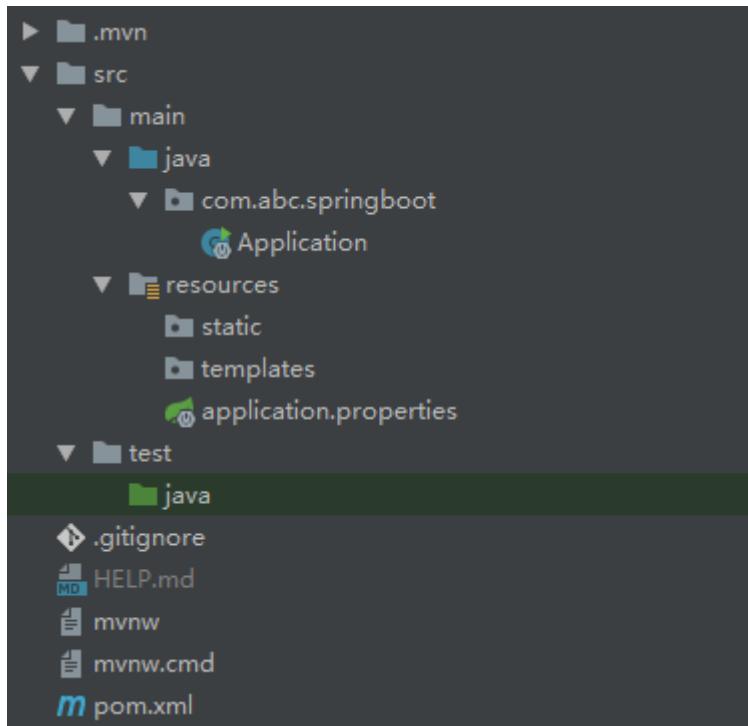
(3) 创建 Dubbo 服务提供者项目

项目名称: 021-springboot-ssm-dubbo-provider





(4) Dubbo 服务提供者项目结构



(5) 给 Dubbo 服务提供者添加依赖

```
<!--MyBatis 集成 SpringBoot 框架起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.1</version>
</dependency>

<!-- MySQL 数据库驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>

<!--Dubbo 集成 SpringBoot 框架起步依赖-->
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
```

```
</dependency>

<!--zookeeper 注册中心-->
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>

<!--SpringBoot 集成-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<!--接口工程-->
<dependency>
    <groupId>com.abc.springboot</groupId>
    <artifactId>020-springboot-ssm-dubbo-interface</artifactId>
    <version>1.0.0</version>
</dependency>
```

(6) 手动指定资源配置文件路径

在 pom 文件中的 build 标签中添加

```
<!--手动指定资源配置文件路径-->
<!--目的：将数据持久层映射文件编译到 classpath 中-->
<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.xml</include>
        </includes>
    </resource>
</resources>
```

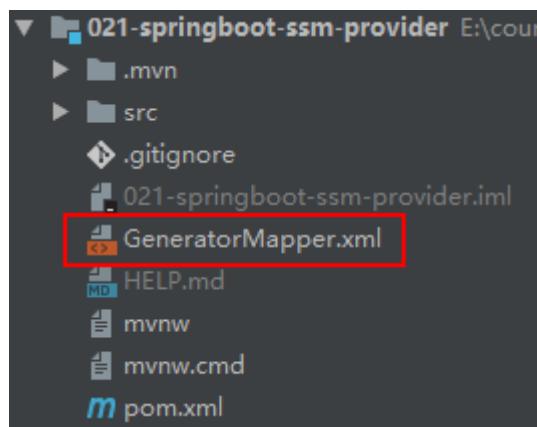
(7) 配置 MyBatis 逆向工程

在 springboot-ssm-dubbo-provider 工程的 pom.xml 文件中添加

A、添加插件

```
<!--mybatis 代码自动生成插件-->
<plugin>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-maven-plugin</artifactId>
    <version>1.3.7</version>
    <configuration>
        <!--配置文件的位置-->
        <configurationFile>GeneratorMapper.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
    </configuration>
</plugin>
```

B、将配置文件存放到项目根据目录



C、GeneratorMapper.xml 内容如下

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration
1.0//EN">
```

```
"http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>

    <!-- 指定连接数据库的 JDBC 驱动包所在位置，指定到你本机的完整路径 -->
    <classPathEntry location="E:\mysql-connector-java-5.1.38.jar"/>

    <!-- 配置 table 表信息内容体，targetRuntime 指定采用 MyBatis3 的版本 -->
    <context id="tables" targetRuntime="MyBatis3">

        <!-- 抑制生成注释，由于生成的注释都是英文的，可以不让它生成 -->
        <commentGenerator>
            <property name="suppressAllComments" value="true"/>
        </commentGenerator>

        <!-- 配置数据库连接信息 -->
        <jdbcConnection driverClass="com.mysql.jdbc.Driver"

connectionURL="jdbc:mysql://localhost:3306/springboot"
                    userId="root"
                    password="123456">
        </jdbcConnection>

        <!-- 生成 model 类，targetPackage 指定 model 类的包名， targetProject 指定生成的 model 放在 eclipse 的哪个工程下面-->
        <javaModelGenerator targetPackage="com.abc.springboot.model"

targetProject="E:\course\031-SpringBoot\001-springboot-projects\020-springboot-ssm-dubbo-interface\src\main\java">
            <property name="enableSubPackages" value="false"/>
            <property name="trimStrings" value="false"/>
        </javaModelGenerator>

        <!-- 生成 MyBatis 的 Mapper.xml 文件，targetPackage 指定 mapper.xml 文件的包名， targetProject 指定生成的 mapper.xml 放在 eclipse 的哪个工程下面 -->
        <sqlMapGenerator targetPackage="com.abc.springboot.mapper"
targetProject="src/main/java">
            <property name="enableSubPackages" value="false"/>
        </sqlMapGenerator>

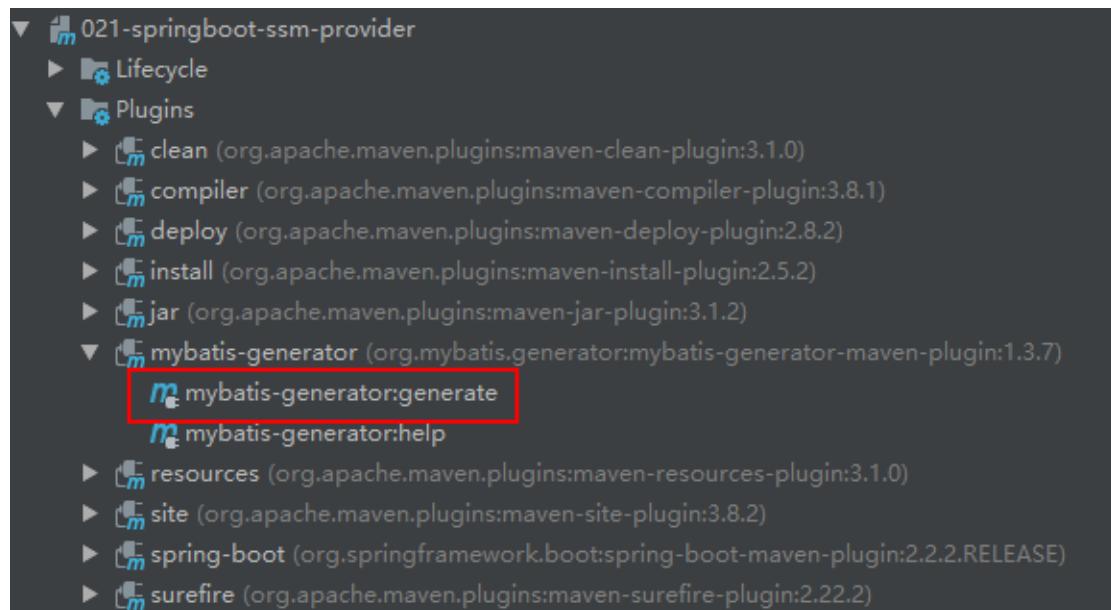
        <!-- 生成 MyBatis 的 Mapper 接口类文件，targetPackage 指定 Mapper 接口类的包名， targetProject 指定生成的 Mapper 接口放在 eclipse 的哪个工程下面 -->
        <javaClientGenerator type="XMLMAPPER"
targetPackage="com.abc.springboot.mapper"
```

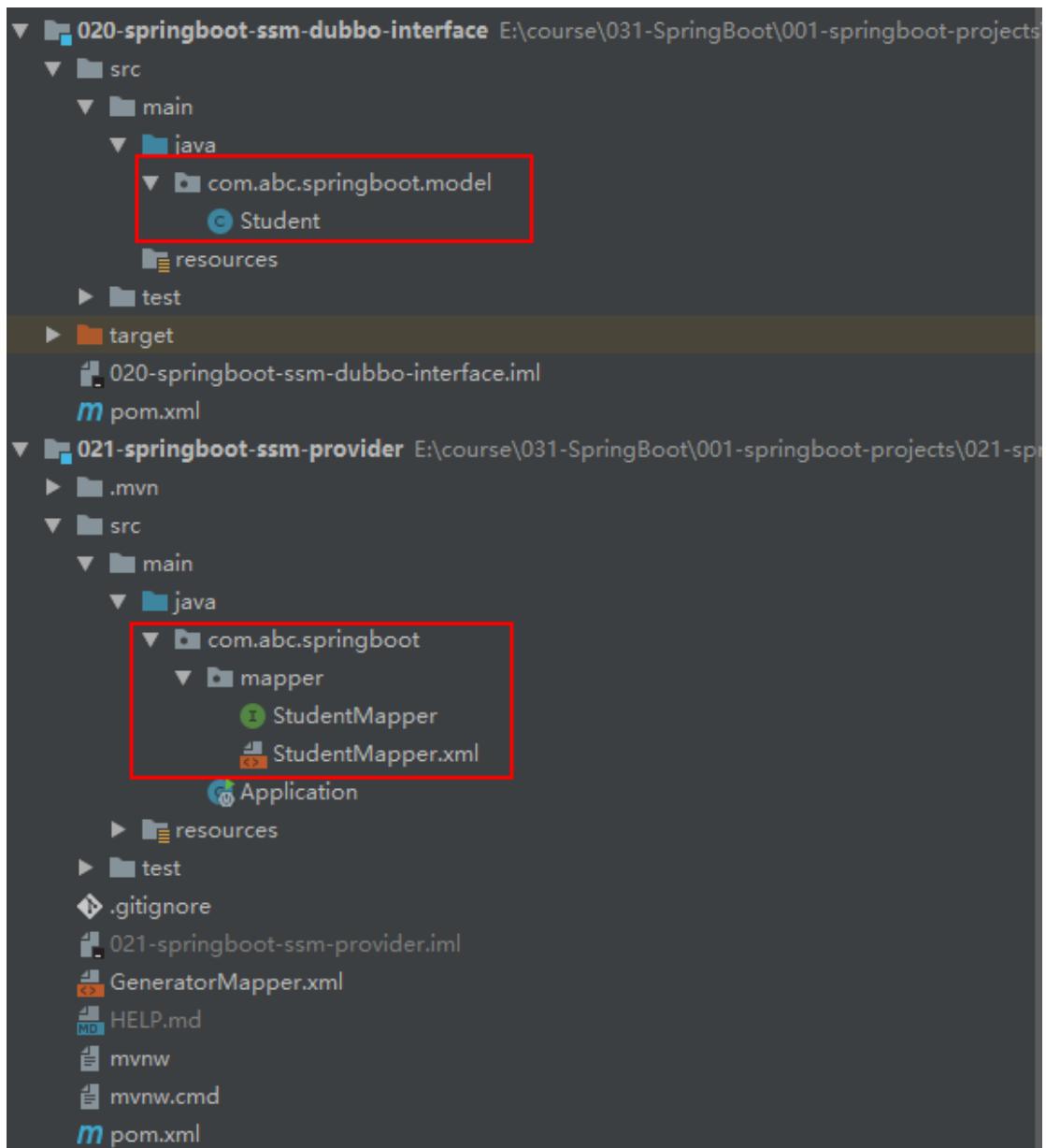
```
targetProject="src/main/java">
    <property name="enableSubPackages" value="false"/>
</javaClientGenerator>

    <!-- 数据库表名及对应的 Java 模型类名 -->
<table tableName="t_student" domainObjectName="Student"
    enableCountByExample="false"
    enableUpdateByExample="false"
    enableDeleteByExample="false"
    enableSelectByExample="false"
    selectByExampleQueryId="false"/>
</context>

</generatorConfiguration>
```

D、 双击生成





E、实体 bean 必须实现序列化

```
public class Student implements Serializable {
```

(8) 配置 021-springboot-ssm-dubbo-provider 核心配置

A、 application.properties

```
#配置内嵌 Tomcat 端口号
server.port=8081
#设置上下文根
server.servlet.context-path=/provider

#配置数据源
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=GMT%2B8
spring.datasource.username=root
spring.datasource.password=123456

#配置 dubbo 服务提供者
spring.application.name=springboot-ssm-dubbo-provider

#表示是服务提供者
spring.dubbo.server=true
#注册中心地址
spring.dubbo.registry=zookeeper://192.168.92.134:2181

#配置 redis 连接信息
spring.redis.host=192.168.92.134
spring.redis.port=6379
spring.redis.password=123456
```

B、 application.yml

```
server:
  port: 8081
  servlet:
    context-path: /provider

spring:
  datasource:
```

```
driver-class-name: com.mysql.cj.jdbc.Driver

url:jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=GMT%2B8

username: root
password: 123456

redis:
  host: 192.168.92.134
  port: 6379
  password: 123456

application:
  name: springboot-ssm-dubbo-provider

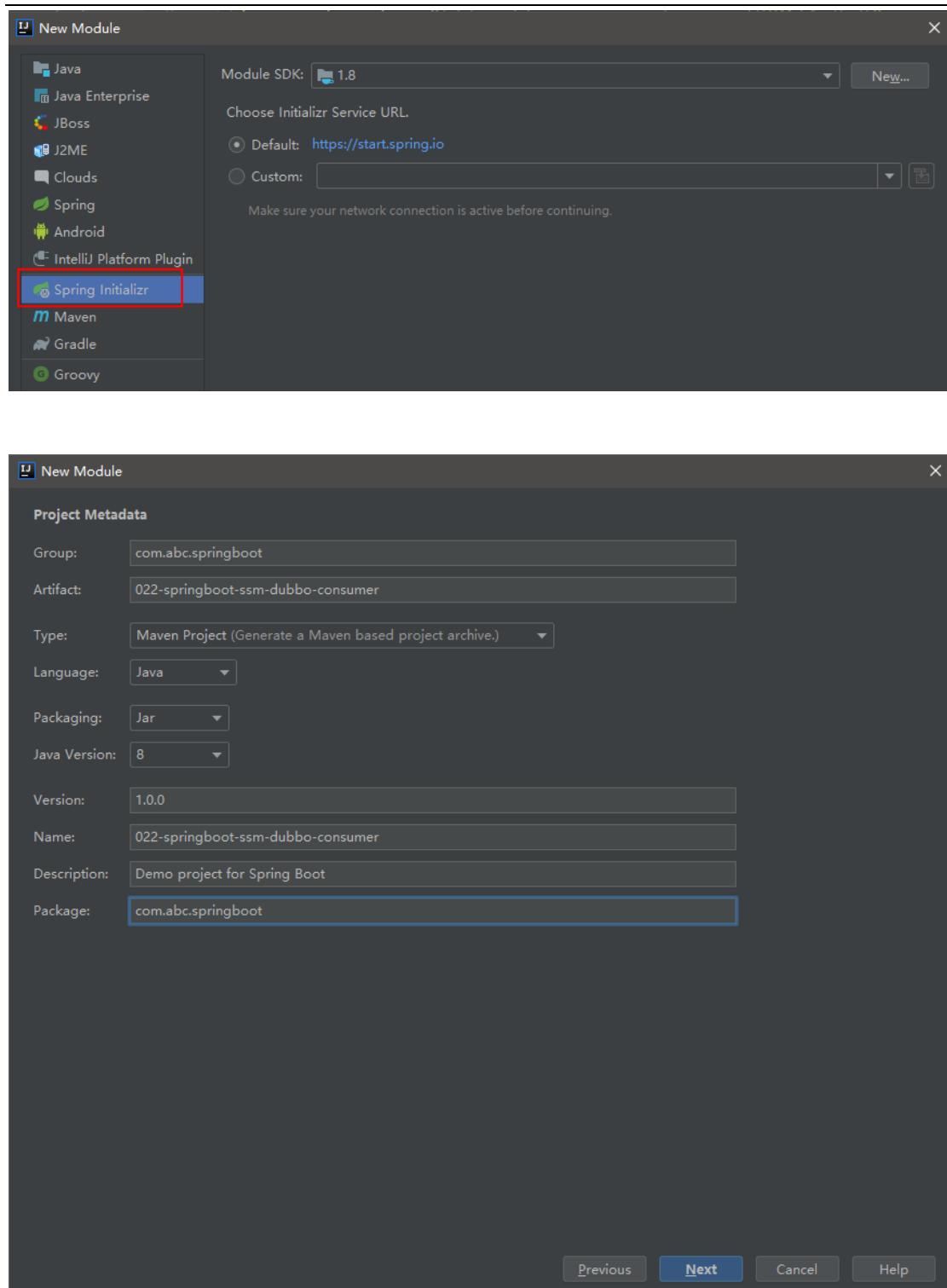
dubbo:
  server: true
  registry: zookeeper://localhost:2181
```

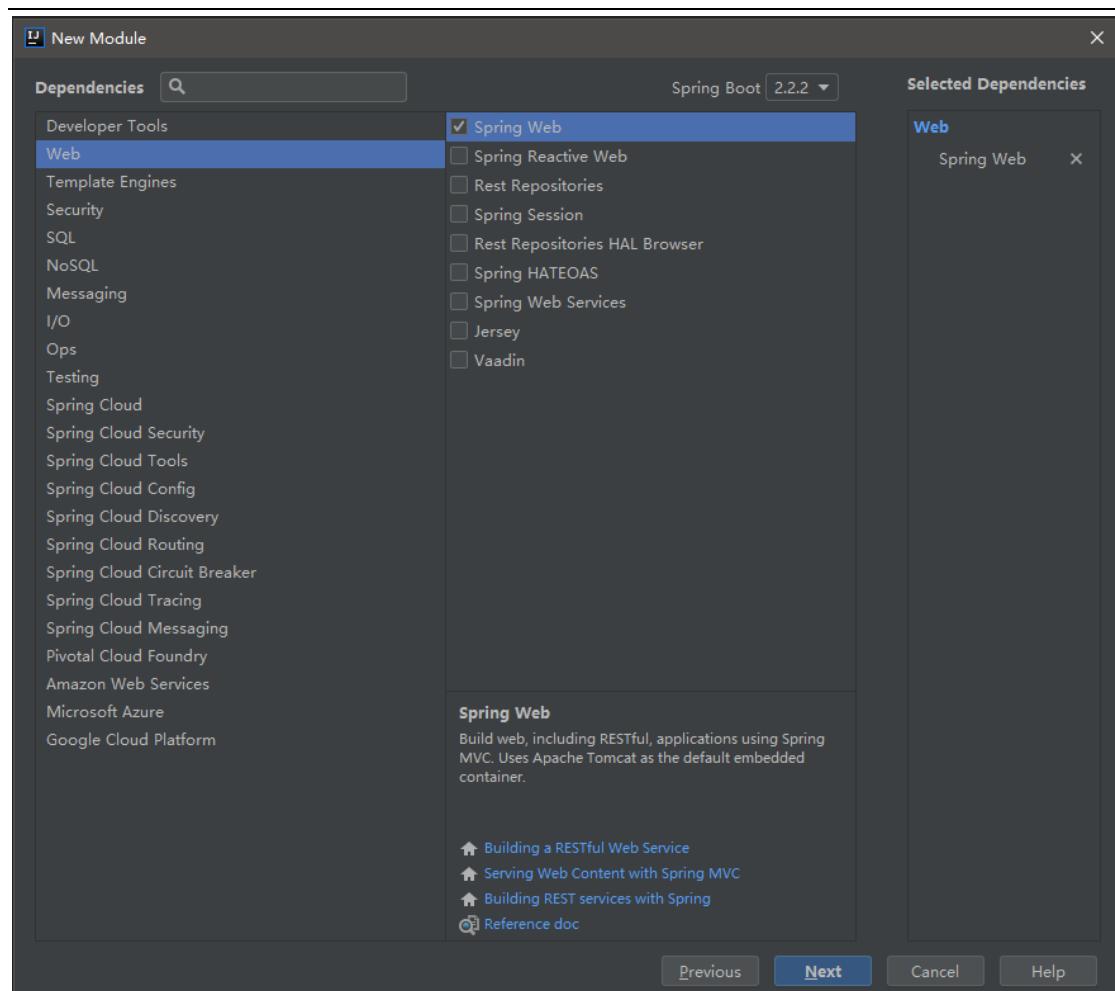
(9) 配置 021-springboot-ssm-dubbo-provider 启动类

```
@SpringBootApplication
@EnableDubboConfiguration //开启 dubbo 支持配置
@MapperScan(basePackages = "com.abc.springboot.mapper") //扫描数据持久层映射文件
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

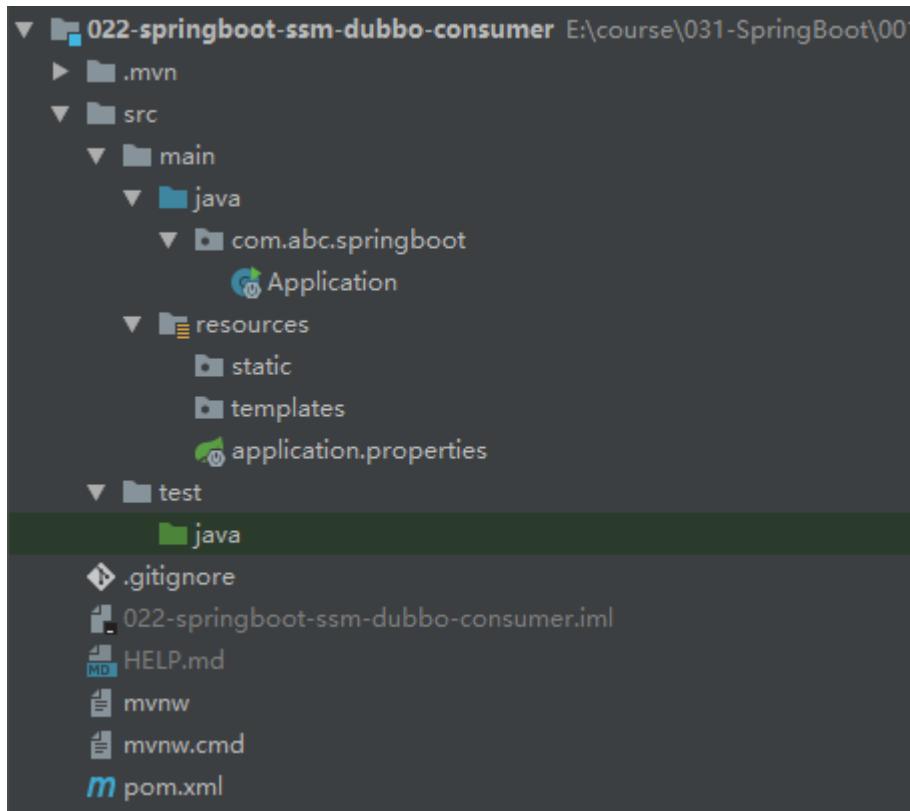
(10) 创建 Dubbo 服务消费者项目

项目名称: 022-springboot-ssm-dubbo-consumer





(11) Dubbo 服务消费者项目结构



(12) 给 Dubbo 服务消费者添加依赖

```
<!--Dubbo 集成 SpringBoot 框架起步依赖-->
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>

<!--zookeeper 注册中心-->
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>

<!--Dubbo 的接口工程-->
<dependency>
```

```
<groupId>com.abc.springboot</groupId>
<artifactId>020-springboot-ssm-dubbo-interface</artifactId>
<version>1.0.0</version>
</dependency>
```

(13) 配置 022-springboot-ssm-dubbo-consumer 核心配置

A、 application.properties

```
#配置内嵌 Tomcat 端口号
server.port=8080
#配置项目上下文根
server.servlet.context-path=consumer

#配置 zookeeper 注册中心
spring.application.name=springboot-ssm-dubbo-consumer
spring.dubbo.registry=zookeeper://192.168.92.134:2181
```

B、 application.yml

```
server:
  port: 8080
  servlet:
    context-path: /consumer

spring:
  application:
    name: springboot-ssm-dubbo-consumer
  dubbo:
    registry: zookeeper://192.168.92.134:2181
```

(14) 配置 022-springboot-ssm-dubbo-consumer 启动类

```
@SpringBootApplication
@EnableDubboConfiguration //开启 dubbo 支持配置
public class Application {
```

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

(15) StudentController 控制层

在服务消费者项目中添加 StudentController 类

```
package com.abc.springboot.web;

import com.abc.springboot.model.Student;
import com.abc.springboot.service.StudentService;
import com.alibaba.dubbo.config.annotation.Reference;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class StudentController {
    @Reference(interfaceName =
"com.abc.springboot.service.StudentService", version = "1.0.0", check =
false)
    private StudentService studentService;

    @GetMapping(value = "/springboot/student")
    public Object queryAllStudents(Integer id) {
        //查询所有学生
        Student student = studentService.queryStudentById(id);
        return student;
    }

    @GetMapping(value = "/springboot/student/count")
    public Object queryAllStudentCount() {
        //获取学生总人数
        Long allStudentCount = studentService.queryAllStudentCount();

        return "学生总人数为: " + allStudentCount;
    }
}
```

(16) 启动 021-springboot-ssm-dubbo-provider

```
Tomcat started on port(s): 8081 (http) with context path '/023-springboot-ssm-dubbo-provider'
```

(17) 启动 022-springboot-ssm-dubbo-consumer

```
Tomcat started on port(s): 8080 (http) with context path '/024-springboot-ssm-dubbo-consumer'
```

(18) 创建 StudentService 业务接口类

在项目 020-springboot-ssm-dubbo-interface 中

```
public interface StudentService {  
    /**  
     * 根据学生标识获取学生信息  
     * @param id  
     * @return  
     */  
    Student queryStudentById(Integer id);  
  
    /**  
     * 获取学生总人数  
     * @return  
     */  
    Long queryAllStudentCount();  
}
```

(19) StudentServiceImpl 业务接口实现类

```
package com.abc.springboot.service;  
  
import com.abc.springboot.mapper.StudentMapper;  
import com.abc.springboot.model.Student;  
import com.alibaba.dubbo.config.annotation.Service;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.stereotype.Component;  
import java.util.concurrent.TimeUnit;
```

```
@Component
@Service(interfaceName =
"com.abc.springboot.service.StudentService",version = "1.0.0",timeout
= 15000)
public class StudentServiceImpl implements StudentService {

    @Autowired
    private StudentMapper studentMapper;

    @Autowired
    private RedisTemplate<Object, Object> redisTemplate;

    @Override
    public Student queryStudentById(Integer id) {
        return studentMapper.selectByPrimaryKey(id);
    }

    @Override
    public Long queryAllStudentCount() {

        //从 redis 中获取学生总人数
        Long allStudentCount = (Long)
redisTemplate.opsForValue().get("allStudentCount");

        //判断是否有值
        if (null == allStudentCount) {

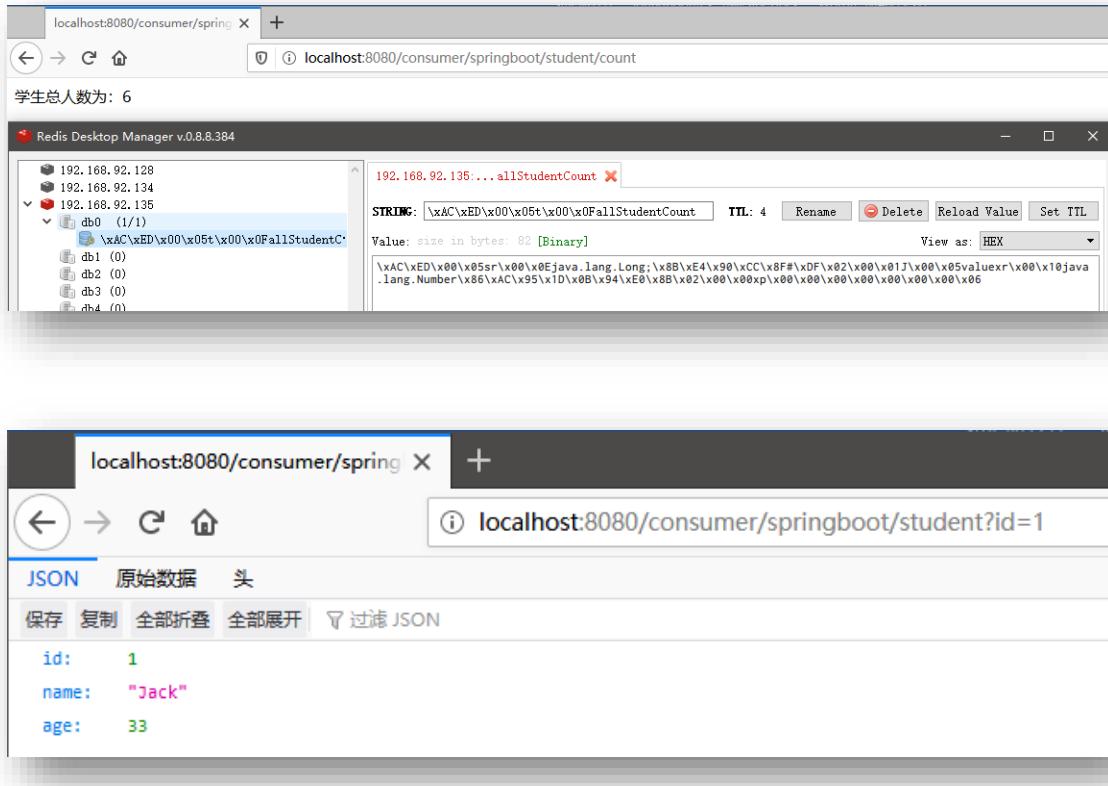
            //从数据库中查询
            allStudentCount = studentMapper.selectAllStudentCount();

            //并存放到 redis 缓存中

            redisTemplate.opsForValue().set("allStudentCount",allStudentCount,10,
TimeUnit.SECONDS);
        }

        return allStudentCount;
    }
}
```

(20) 启动浏览器进行测试



3.6.3 SpringBoot 项目日志中警告信息的处理

在启动项目的时候，控制台出现如下警告信息

```
SLF4J: Class path contains multiple SLF4J bindings.  
SLF4J: Found binding in [jar:file:/D:/repository/ch/qos/logback/logback-classic/1.2.3/logback-classic-1.2.3.jar!/org/slf4j/  
SLF4J: Found binding in [jar:file:/D:/repository/org/slf4j/slf4j-log4j12/1.7.25/slf4j-log4j12-1.7.25.jar!/org/slf4j.impl/st  
SLF4J: See http://www.slf4j.org/codes.html#multiple\_bindings for an explanation.  
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]  
log4j:WARN No appenders could be found for logger (com.alibaba.dubbo.common.logger.LoggerFactory).  
log4j:WARN Please initialize the log4j system properly.  
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
```

主要是因为 Zookeeper 包中，slf4j-log4j12 和 log4j 冲突了，需要处理一下

在服务提供者和消费中的 `pom.xml` 文件的 ZooKeeper 依赖中添加如下内容

```
<!--zookeeper 注册中心-->
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
```

```
<version>0.10</version>
<exclusions>
    <exclusion>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
    </exclusion>
    <exclusion>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
</exclusions>
</dependency>
```

替换后，重启服务进行测试

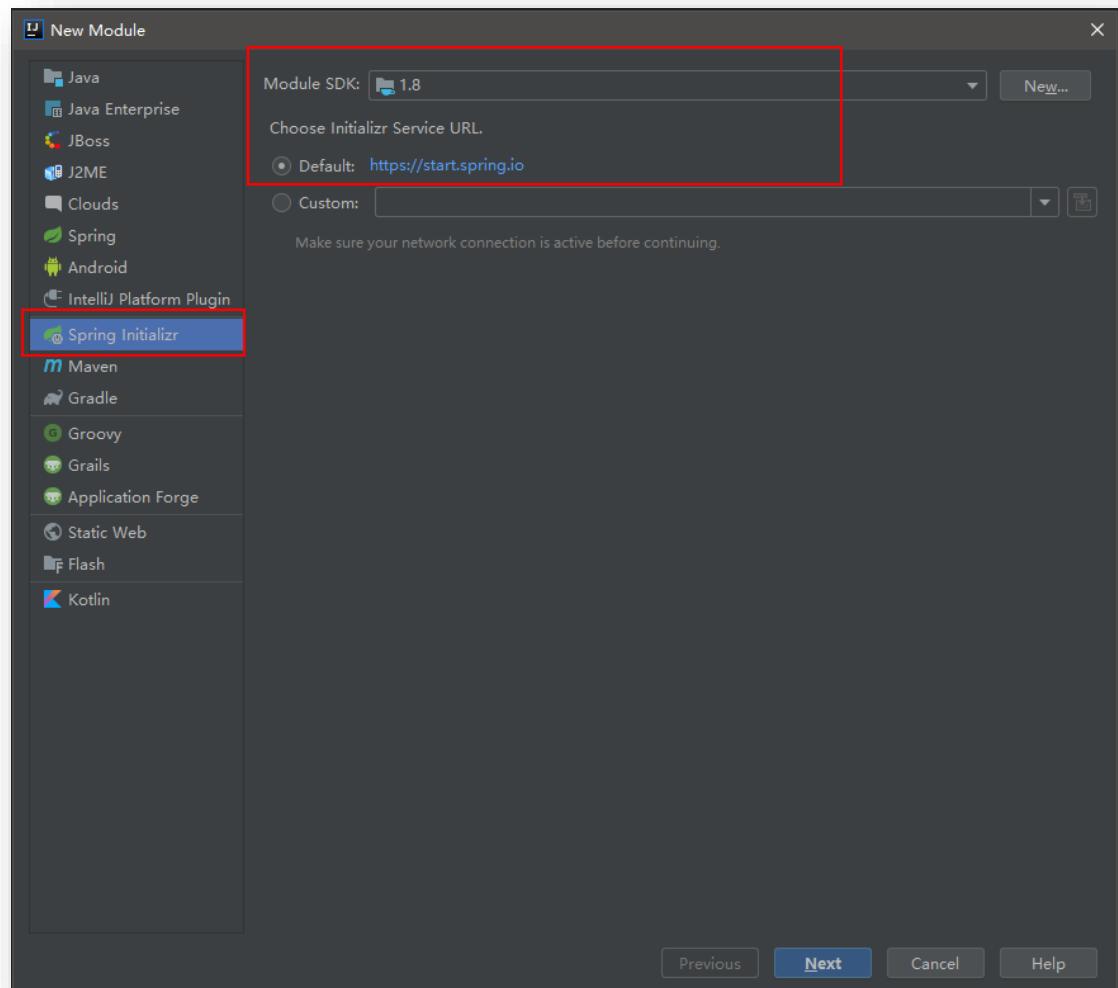
第4章 Spring Boot 非 web 应用程序

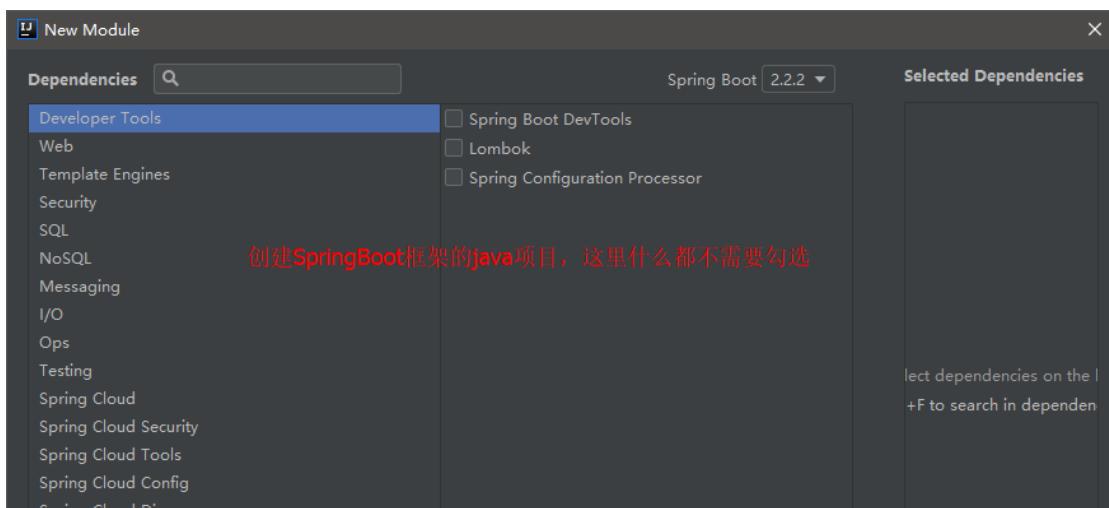
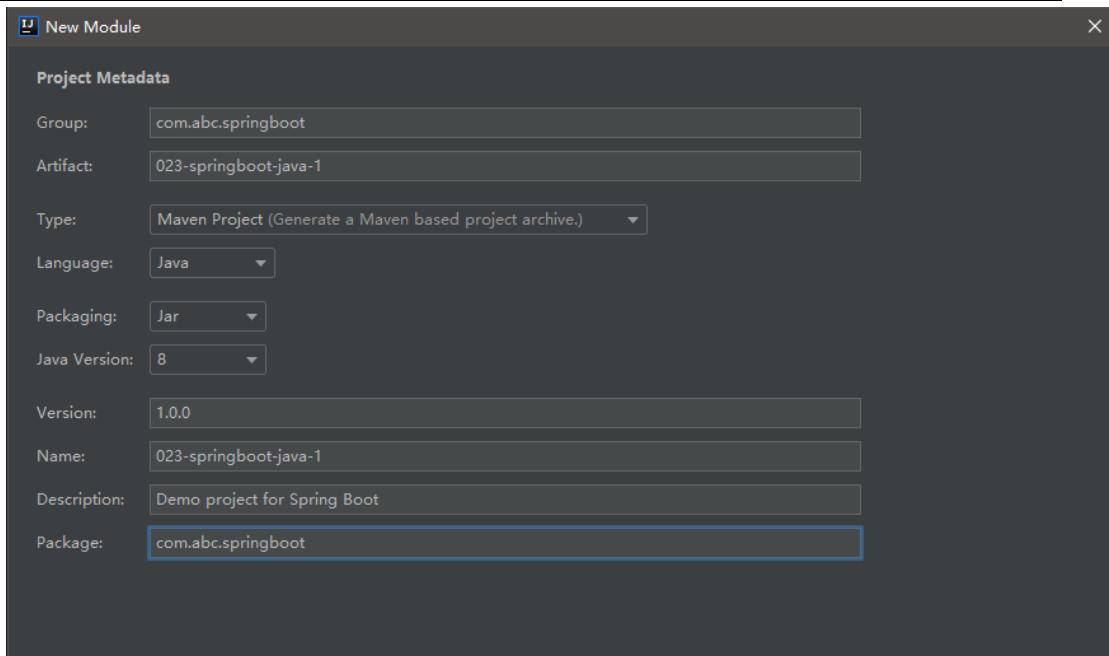
在 Spring Boot 框架中，要创建一个非 Web 应用程序（纯 Java 程序），有两种方式

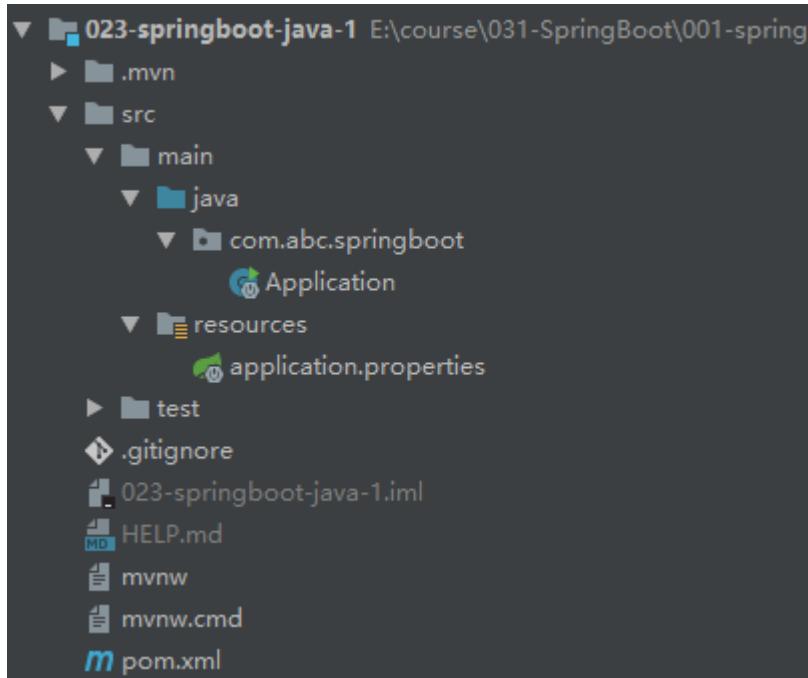
4.1 方式一：直接在 **main** 方法中，根据 **SpringApplication.run()** 方法获取返回的 **Spring** 容器对象，再获取业务 **bean** 进行调用

项目名称：023-springboot-java-01

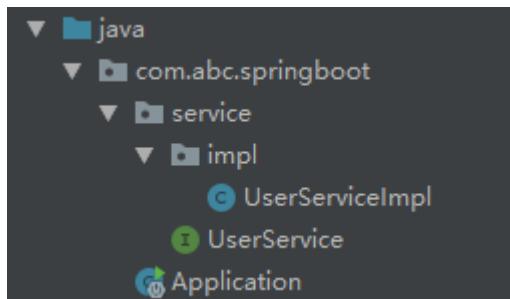
4.1.1 创建一个 SpringBoot Module







4.1.2 创建一个演示 UserService 接口及实现类



UserService.java 接口

```
public interface UserService {  
    String sayHello(String message);  
}
```

UserServiceImpl.java 接口实现类

```
public class UserServiceImpl implements UserService {  
  
    @Override  
    public String sayHello(String message) {  
        return "Hello, SpringBoot Java!";  
    }  
}
```

4.1.3 在 Application 类的 main 方法中，获取容器，调用业务 bean

```
package com.abc.springboot;

import com.abc.springboot.service.UserService;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {

        /**
         * SpringBoot 程序启动后，返回值是 ConfigurableApplicationContext，它也是一个
         Spring 容器对象
         * 它其它相当于原来 Spring 中启动容器 ClassPathXmlApplicationContext context =
         new ClassPathXmlApplicationContext("");
         */

        //获取 SpringBoot 程序启动后的 Spring 容器
        ConfigurableApplicationContext context =
        SpringApplication.run(Application.class, args);

        //从 Spring 容器中获取指定 bean 的对象
        UserService userService = (UserService)
        context.getBean("userServiceImpl");

        //调用业务 bean 的方法
        String sayHello = userService.sayHello();

        System.out.println(sayHello);
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:62832', transport: 'socket'

. \ \ / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
( ( ) \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ \ \ \ 
\ \ \ \ \ \ \ 
\ \ \ \ \ 
\ \ \ 
\ \ 
\ 

:: Spring Boot ::      (v2.1.5.RELEASE)

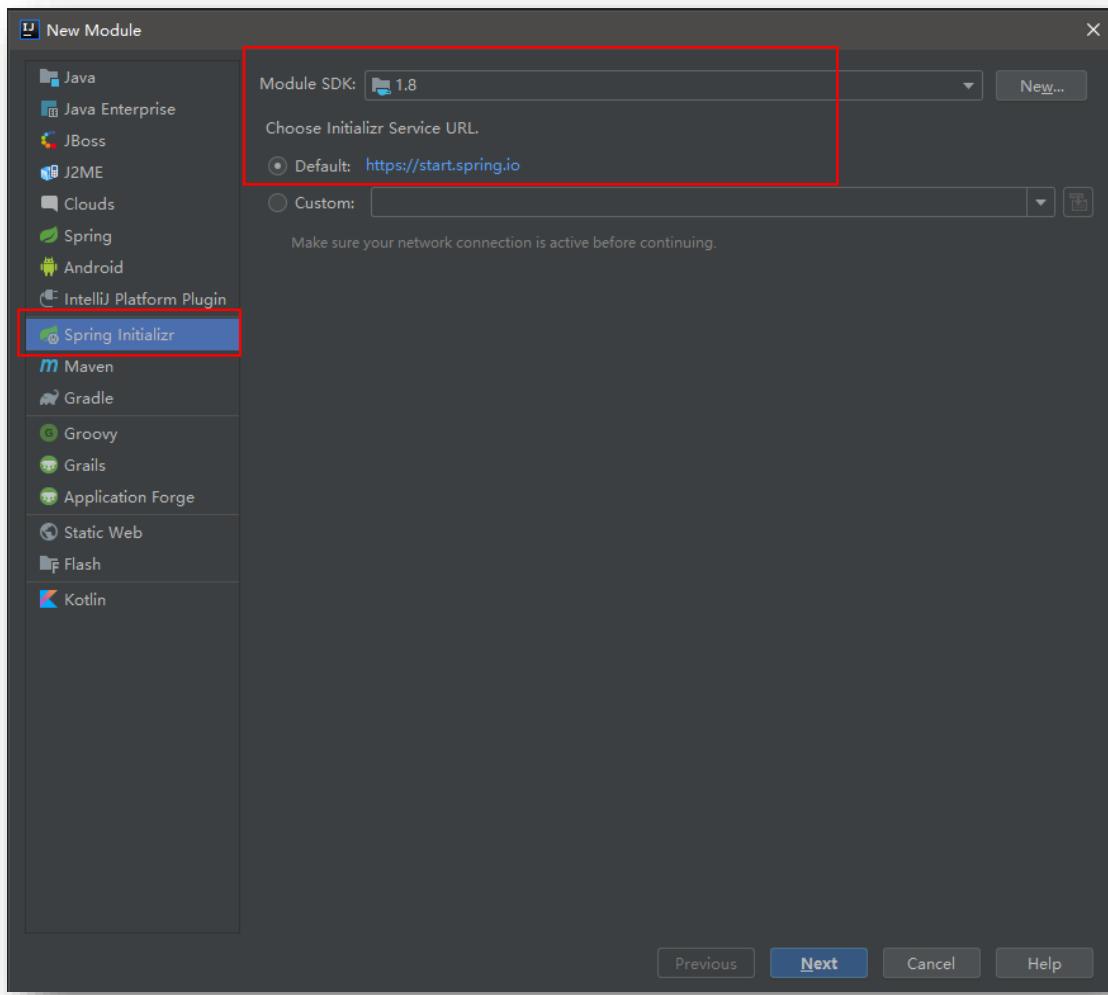
2019-06-11 08:49:01.485  INFO 11128 --- [           main] com.abc.springboot.Application      : Hello,SpringBoot Java!
Disconnected from the target VM, address: '127.0.0.1:62832', transport: 'socket'

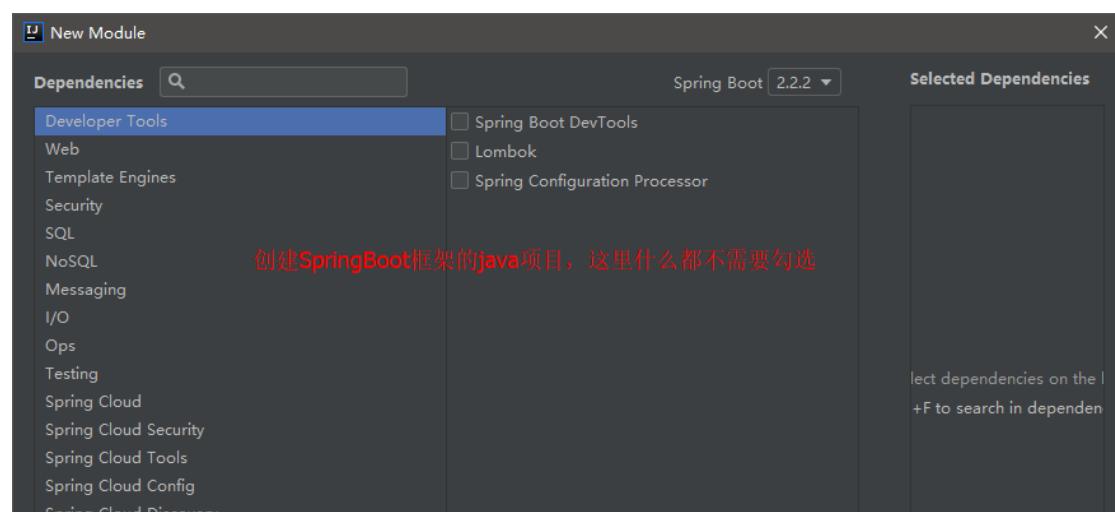
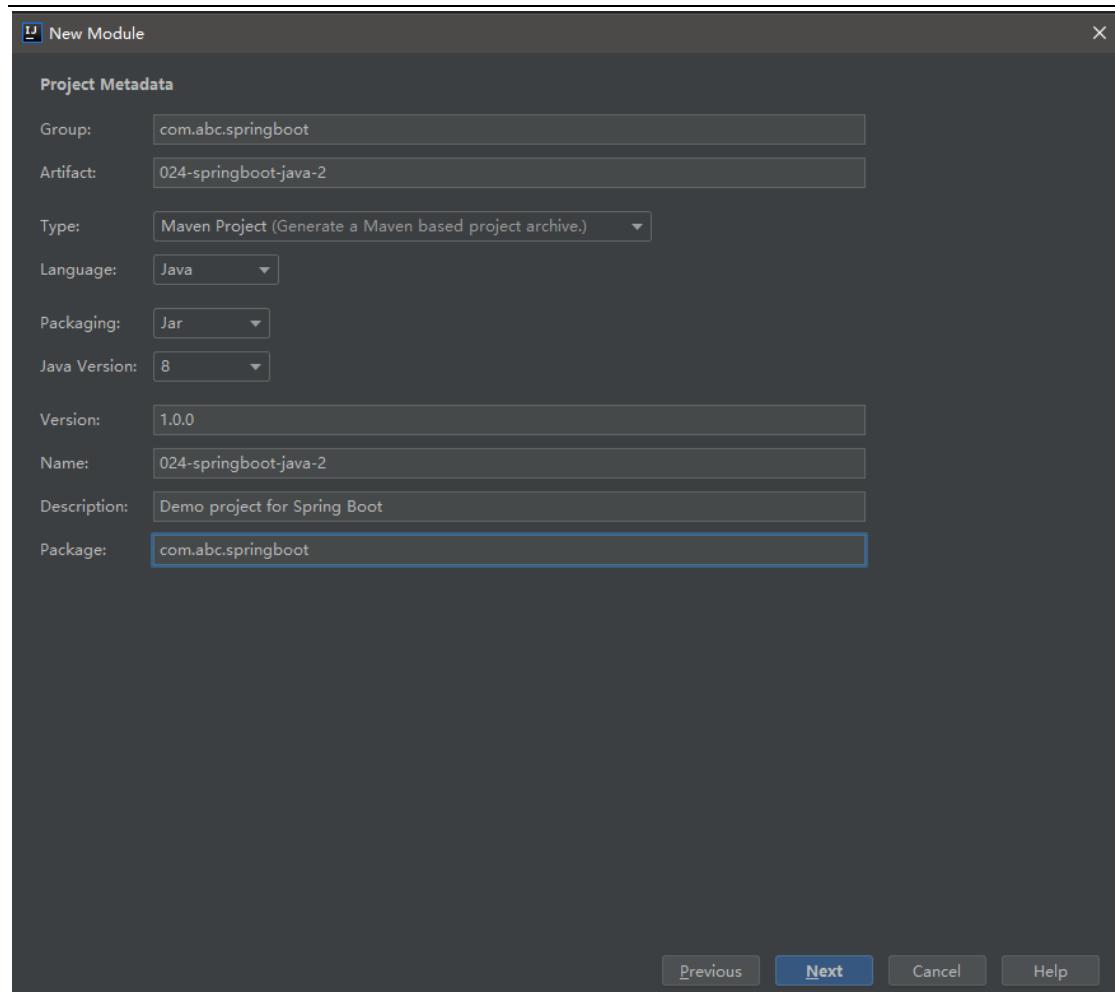
Process finished with exit code 0
```

4.2 方式二：Spring boot 的入口类实现 CommandLineRunner 接口

项目名称：024-springboot-java-2

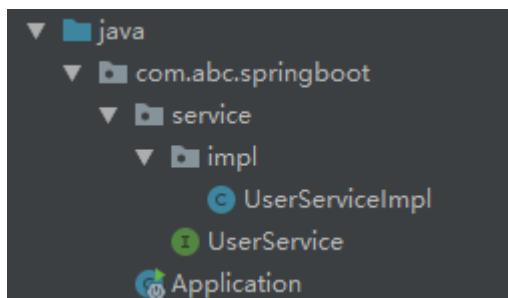
4.2.4 创建一个 SpringBoot Module







4.2.5 创建一个演示 UserService 接口及实现类



UserService.java 接口

```
public interface UserService {  
    String sayHello(String message);  
}
```

UserServiceImpl.java 接口实现类

```
public class UserServiceImpl implements UserService {  
  
    @Override  
    public String sayHello(String message) {  
        return "Hello, SpringBoot Java!";  
    }  
}
```

4.2.6 修改启动类

将原有 Application 类的@SpringBootApplication 注解注释掉，复制一个新的 Application 取名为 Application2，实现 CommandLineRunner 接口

```
package com.abc.springboot;

import com.abc.springboot.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application2 implements CommandLineRunner {

    //第二步：通过容器获取 bean，并注入给 userService
    @Autowired
    private UserService userService;

    public static void main(String[] args) {
        //第一步：SpringBoot 的启动程序，会初始化 spring 容器
        SpringApplication.run(Application2.class,args);
    }

    //覆盖接口中的 run 方法
    @Override
    public void run(String... args) throws Exception {

        //第三步：容器启动后调用 run 方法，在该方法中调用业务方法
        String sayHello = userService.sayHello();

        System.out.println(sayHello);
    }
}
```

4.3 小 Tip

4.3.7 关闭 SpringBoot Logo 图标及启动日志

项目名称：027-springboot-logo-01

```
package com.abc.springboot;

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

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication springApplication = new SpringApplication(Application.class);
        //关闭启动 logo 的输出
        springApplication.setBannerMode(Banner.Mode.OFF);
        springApplication.run(args);
    }
}
```

4.3.8 修改启动的 logo 图标

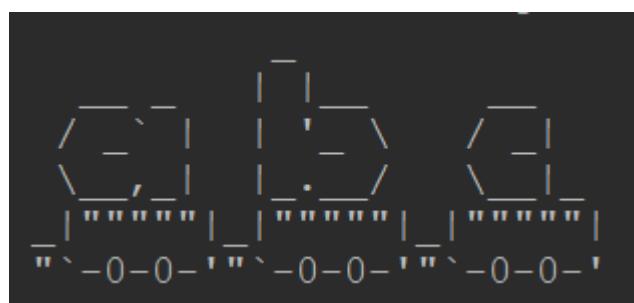
项目名称：028-springboot-logo-02

修改前 LOGO



在 src/main/resources 放入 banner.txt 文件，该文件名字不能随意，文件中的内容就是要输出的 logo；可以利用网站生成图标：<https://www.bootschool.net/ascii> 或者 <http://patorjk.com/software/taag/>，将生成好的图标文字粘贴到 banner.txt 文件中，然后将关闭 logo 输出的语句注释，启动看效果

修改后 LOGO



第5章 Spring Boot 使用拦截器

5.1 回顾 SpringMVC 使用拦截器步骤

- 自定义拦截器类，实现 HandlerInterceptor 接口
- 注册拦截器类

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/loan/**"/>
        <mvc:exclude-mapping path="/loan/loan"/>
        <mvc:exclude-mapping path="/loan/loanInfo"/>
        <mvc:exclude-mapping path="/loan/checkPhone"/>
        <mvc:exclude-mapping path="/loan/checkCaptcha"/>
        <mvc:exclude-mapping path="/loan/register"/>
        <mvc:exclude-mapping path="/loan/login"/>
        <mvc:exclude-mapping path="/loan/loadStat"/>
        <mvc:exclude-mapping path="/loan/message/register"/>
        <mvc:exclude-mapping path="/loan/messageCode"/>
        <mvc:exclude-mapping path="/loan/wxpayNotify"/>
        <bean class="com.bjpowernode.p2p.interceptor.UserInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

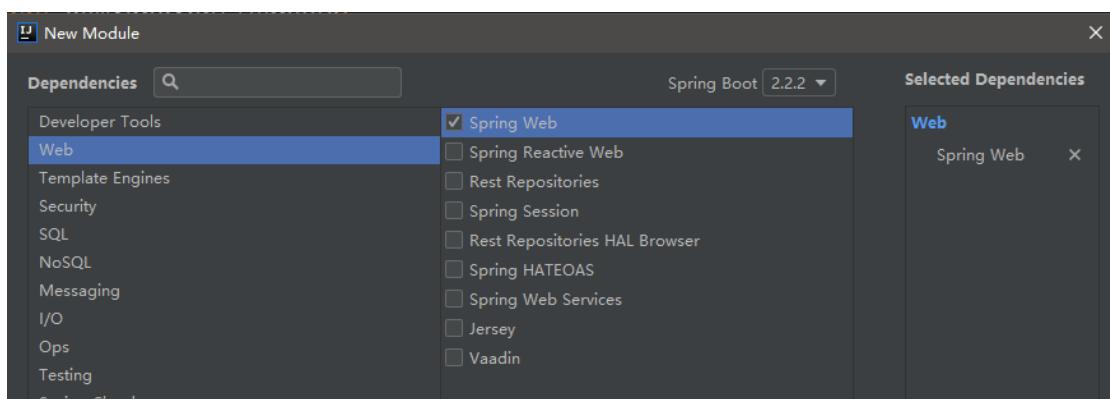
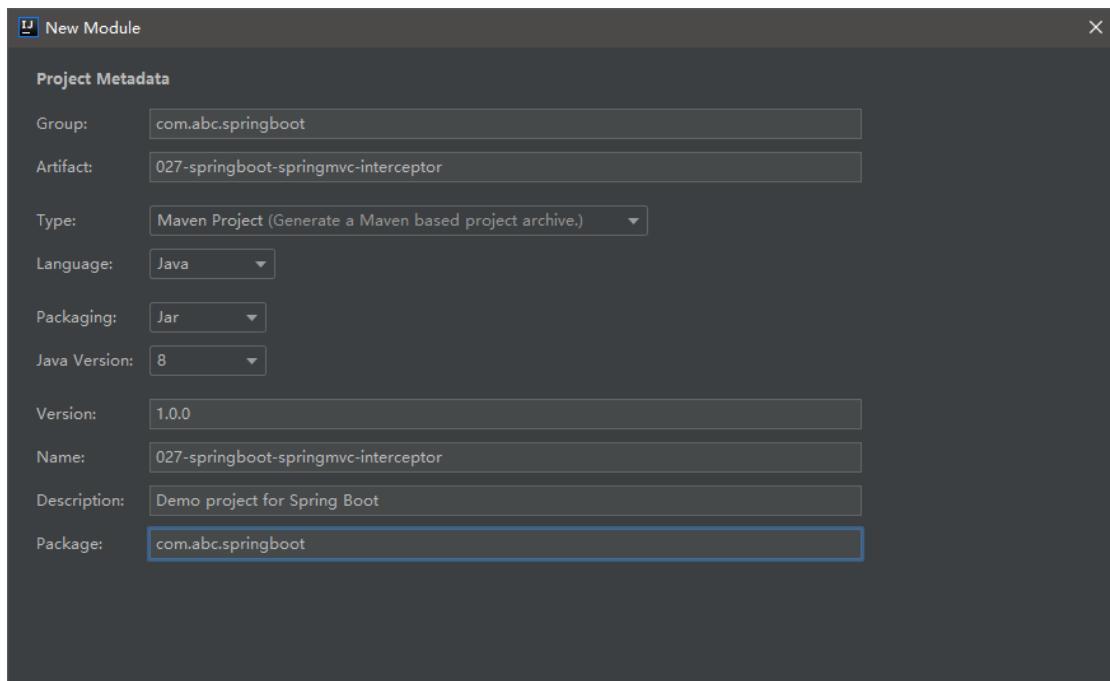
- 按照 SpringMVC 方式编写一个拦截器类，实现 HandlerInterceptor 接口

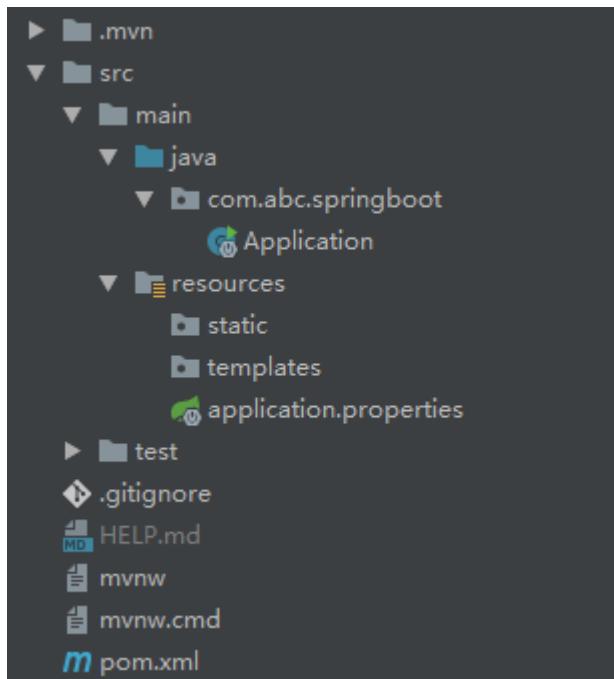
```
public class UserInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {
        //编写登录拦截业务逻辑
        //返回 true 通过
        //返回 false 被拦截
        System.out.println("-----登录拦截器-----");
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler, ModelAndView modelAndView)
    throws Exception {
    }
    @Override
    public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex) throws
    Exception {
    }
}
```

5.2 Spring Boot 使用拦截器步骤

项目名称：027-springboot-springmvc-interceptor

5.2.1 创建一个 SpringBoot 框架 Web 项目





5.2.2 实现一个登录拦截器

```
public class UserInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        System.out.println("-----编写拦截规则-----");
        //编写拦截规则
        //true: 通过
        //false: 不通过

        //从 session 中获取结果
        Integer code = (Integer) request.getSession().getAttribute("code");
        if (null == code) {
            response.sendRedirect(request.getContextPath() + "/user/error");
            return false;
        }
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
    }
}
```

```
@Override
public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) throws Exception {
    }
}
```

5.2.3 创建一个控制层

```
@RestController
public class UserController {

    @RequestMapping(value = "/user/account")
    public Object queryAccount() {
        return "帐户可用余额: 900 元";
    }

    @RequestMapping(value = "/user/verifyRealName")
    public Object verifyRealName(HttpServletRequest request) {
        request.getSession().setAttribute("code", 0);
        return "用户实名认证成功";
    }

    @RequestMapping(value = "/user/error")
    public Object error() {
        return "用户没有实名认证";
    }
}
```

5.2.4 @Configuration 定义配置类-拦截器

在项目中创建一个 config 包，创建一个配置类 InterceptorConfig，并实现 WebMvcConfigurer 接口，覆盖接口中的 addInterceptors 方法，并为该配置类添加 @Configuration 注解，标注此类为一个配置类，让 Spring Boot 扫描到，这里的操作就相当于 SpringMVC 的注册拦截器，@Configuration 就相当于一个 applicationContext-mvc.xml

```
@Configuration //用于定义配置类，可替换 xml 文件；定义一个拦截器，相当于之前的
mvc 里的配置
public class InterceptorConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {

        //定义需要拦截的路径
        String[] addPathPatterns = {
            "/user/**",
        } ;

        //定义不需要拦截的路径
        String[] excludePathPatterns = {
            "/user/error",
            "/user/verifyRealName"
        } ;

        registry.addInterceptor(new UserInterceptor()) //添加要注册的拦截
器对象
            .addPathPatterns(addPathPatterns)           //添加需要拦截的路径
            .excludePathPatterns(excludePathPatterns); //添加不需要拦截的路径
    }
}
```

5.2.5 测试

第6章 Spring Boot 中使用 Servlet（了解）

6.1 方式一 通过注解扫描方式实现

项目名称：028-springboot-servlet-01

6.1.1 通过注解方式创建一个 Servlet

创建 MyServlet.class

```
@WebServlet(urlPatterns = "/myServlet")
public class MyServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().print("My SpringBoot Servlet");
        response.getWriter().flush();
        response.getWriter().close();
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

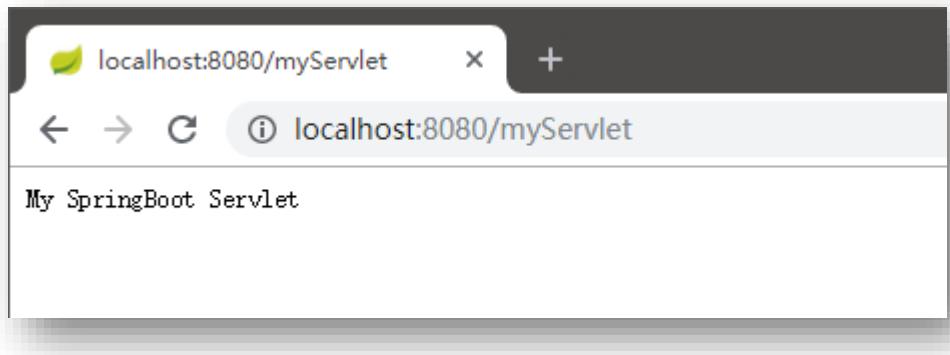
6.1.2 在 主 应 用 程 序 Application 类 上 添 加

@ServletComponentScan("com.abc.springboot.servlet")

```
@SpringBootApplication
@ServletComponentScan(basePackages = "com.abc.springboot.servlet")
public class Application {

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

6.1.3 启动应用 SpringBoot，浏览器访问测试



6.2 方式二 通过 SpringBoot 的配置类实现（组件注册）

项目名称：029-springboot-servlet-2

6.2.1 创建一个普通的 Servlet

创建 MyServlet 类

```
public class MyServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.getWriter().print("Hello, SpringBoot Servlet!");
        resp.getWriter().flush();
        resp.getWriter().close();
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        doGet(req, resp);
    }
}
```

6.2.2 编写一个 Spring Boot 的配置类，在该类中注册 Servlet

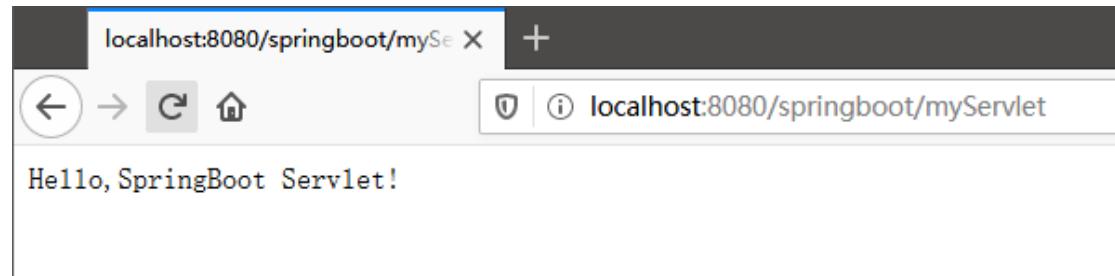
创建 ServletConfig 配置类

```
@Configuration //将此类作为配置类
public class ServletConfig {

    // @Bean 是一个方法级别上的注解，主要用在配置类里
    /*
     * 相当于一个<beans>
     *         <bean id="" class="" />
     *     </beans>
     */
    @Bean
    public ServletRegistrationBean myServletRegistrationBean() {
        //将自定义 servlet 注册到注册 Servlet 类中，并指定访问路径
        ServletRegistrationBean servletRegistrationBean = new
        ServletRegistrationBean(new MyServlet(), "/springboot/myServlet");

        return servletRegistrationBean;
    }
}
```

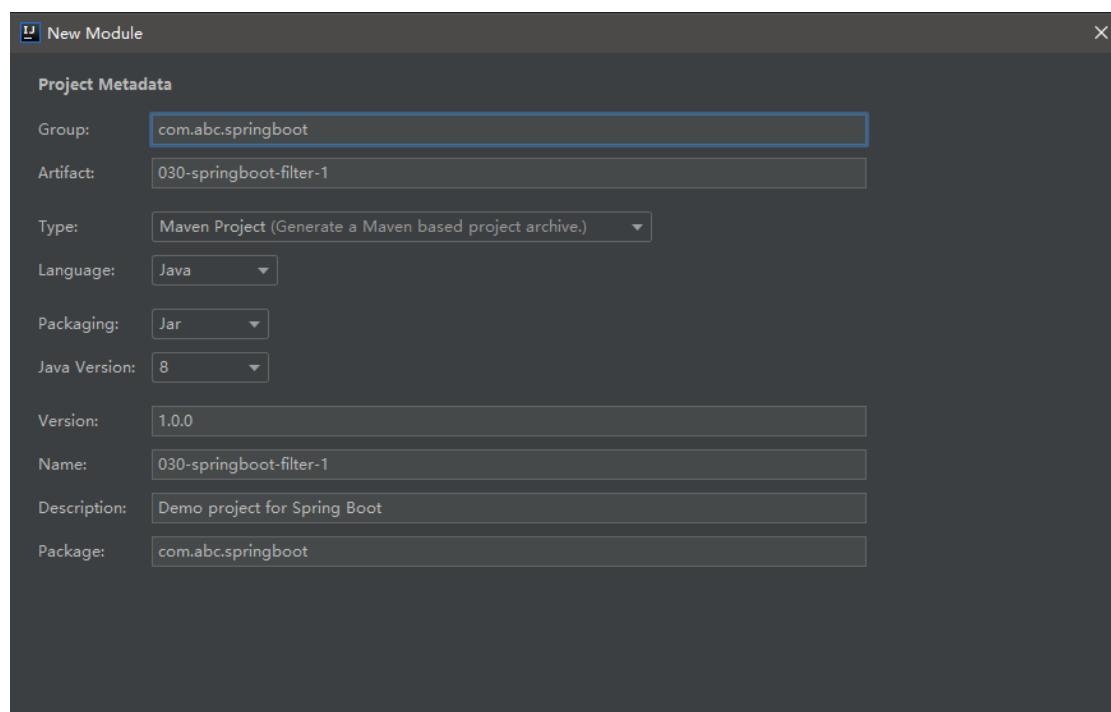
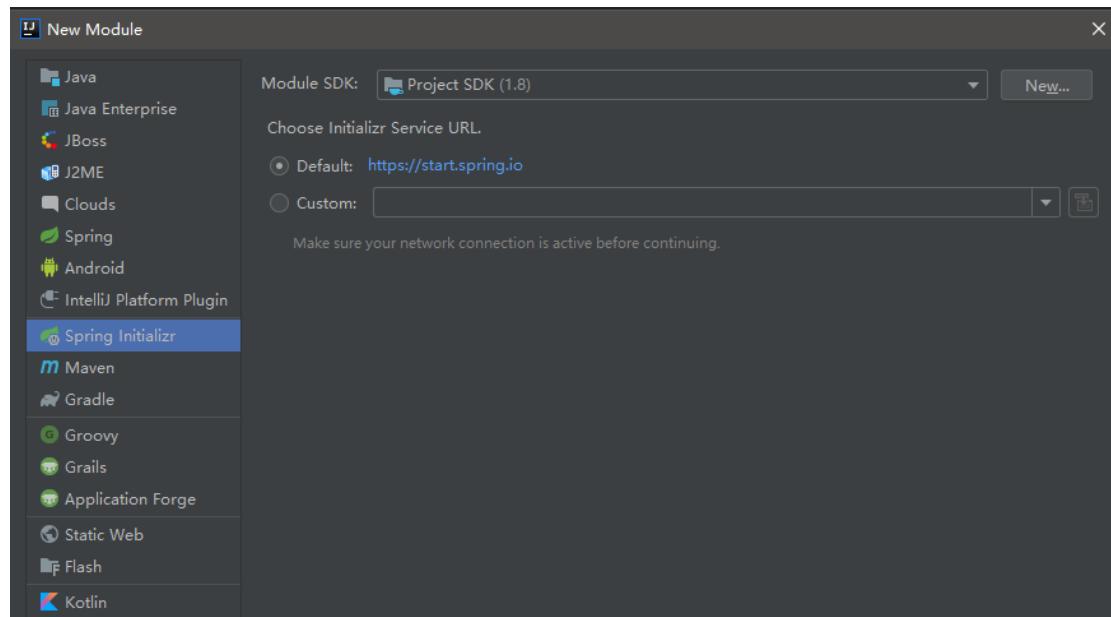
6.2.3 启动应用 SpringBoot，浏览器访问测试

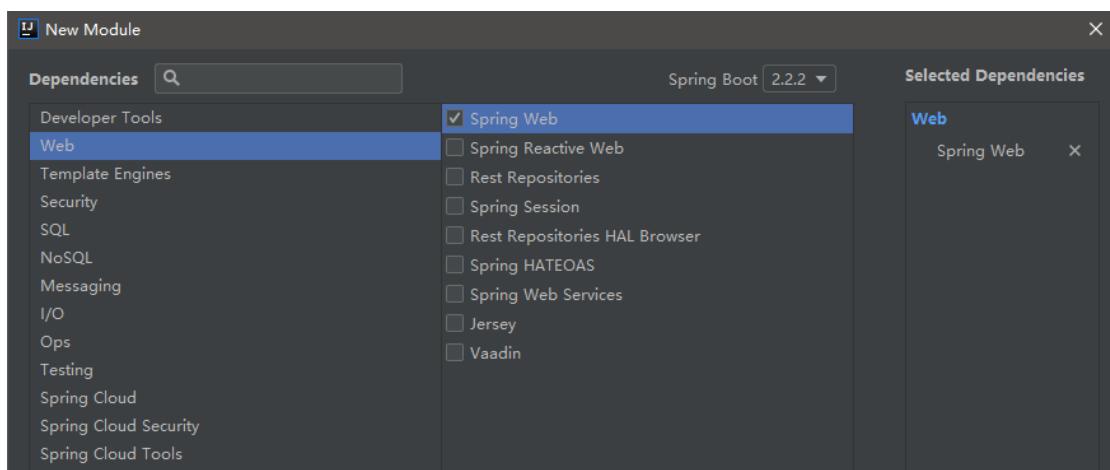


第7章 Spring Boot 中使用 Filter（了解）

7.1 方式一 通过注解方式实现

7.1.1 项目名称：030-springboot-filter-1





7.1.2 通过注解方式创建一个 Filter

创建 MyFilter

```
@WebFilter(urlPatterns = "/springboot/myFilter")
public class MyFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
FilterChain filterChain) throws IOException, ServletException {

        System.out.println("-----您已进入过滤器-----");

        filterChain.doFilter(servletRequest,servletResponse);
    }
}
```

7.1.3 在 主 应 用 程 序 Application 类 上 添 加

```
@ServletComponentScan("basePackages = "com.abc.springboot.filter")
```

```
package com.abc.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.ServletComponentScan;
```

```
@ServletComponentScan(basePackages = "com.abc.springboot.filter")
@SpringBootApplication
public class Application {

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

}
```

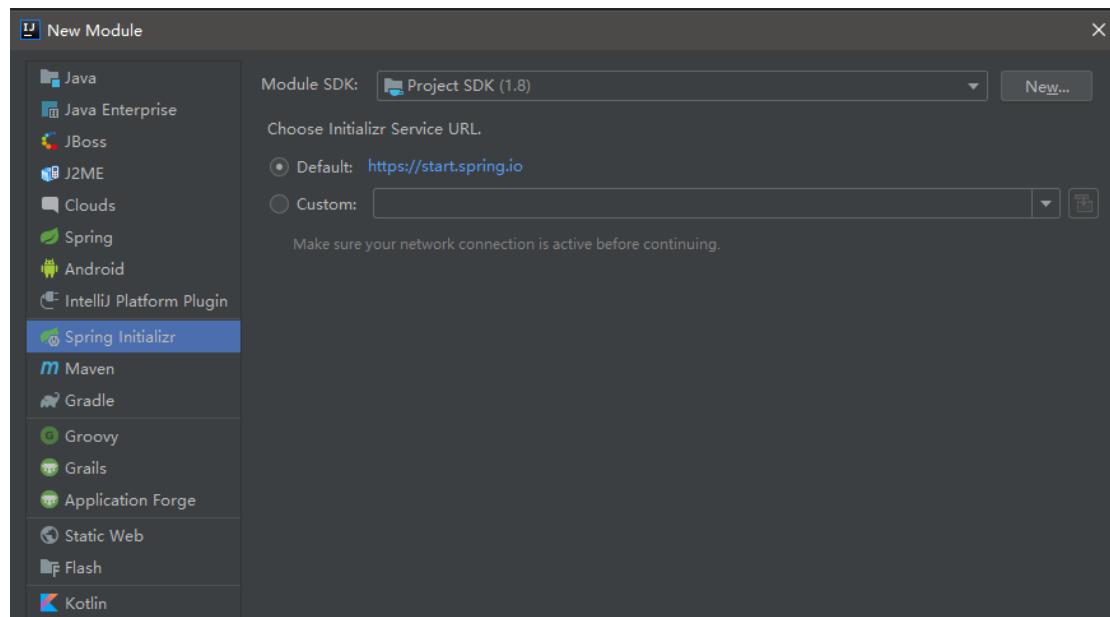
7.1.4 启动应用 SpringBoot，浏览器访问测试

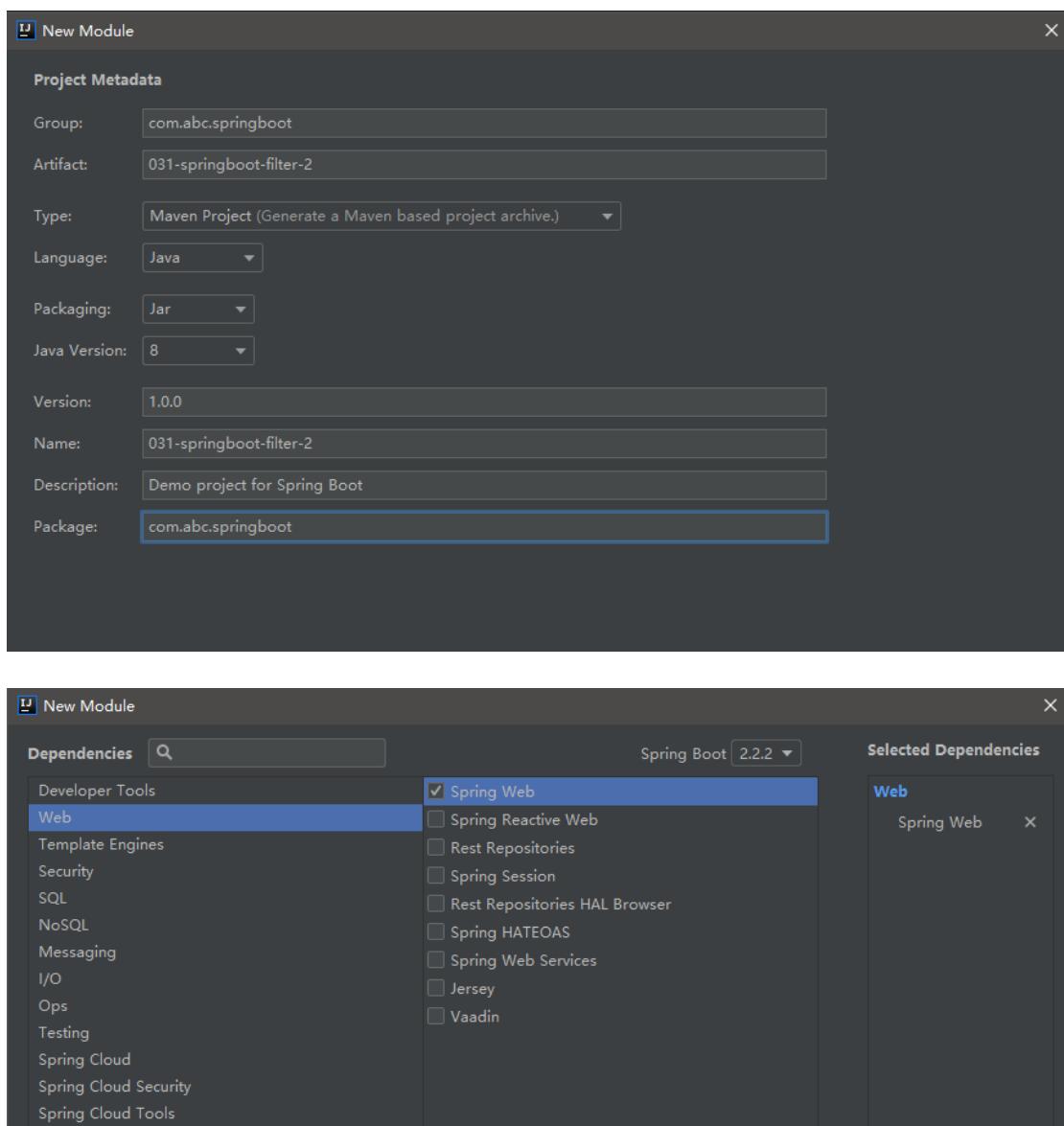
只过滤/springboot/myFilter 请求



7.2 方式二 通过 Spring Boot 的配置类实现

7.2.5 项目名称：031-springboot-filter-2





7.2.6 创建一个普通的 Filter

创建 MyFilter 类

```
public class MyFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
            FilterChain filterChain) throws IOException, ServletException {
        System.out.println("-----通过配置类注册过滤器-----");
        filterChain.doFilter(servletRequest, servletResponse);
    }
}
```

7.2.7 编写一个 Spring Boot 的配置类，在该类中注册 Filter

创建 FilterConfig 配置类

```
@Configuration //定义为配置类
public class FilterConfig {

    @Bean
    public FilterRegistrationBean myFilterRegistration() {
        //注册过滤器
        FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(new MyFilter());
        //添加过滤路径
        filterRegistrationBean.addUrlPatterns("/springboot/*", "/user/*");
        return filterRegistrationBean;
    }
}
```

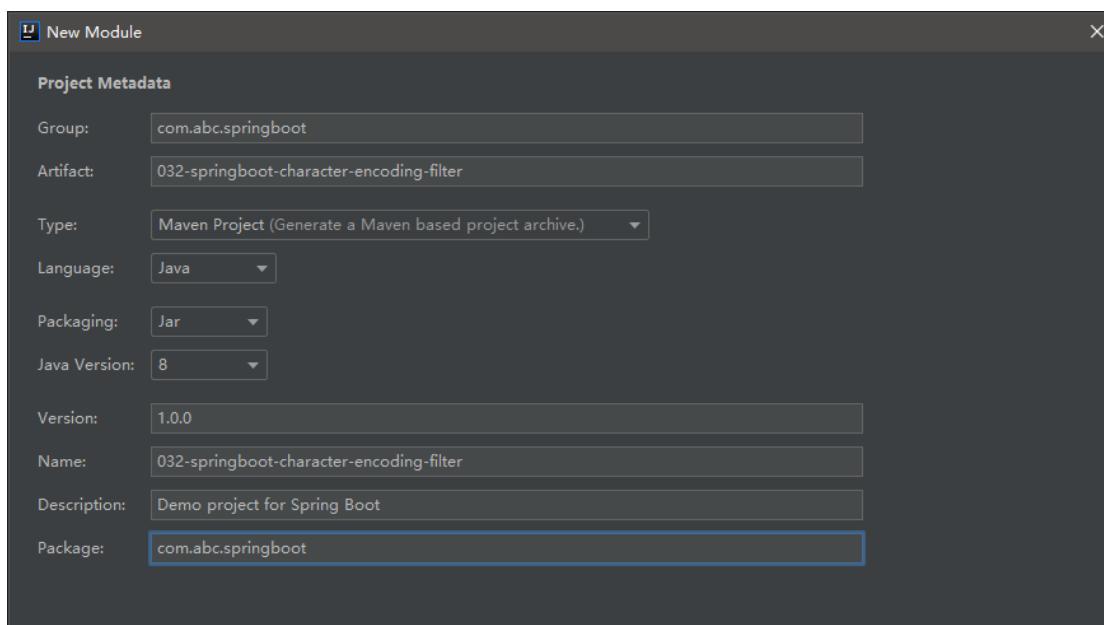
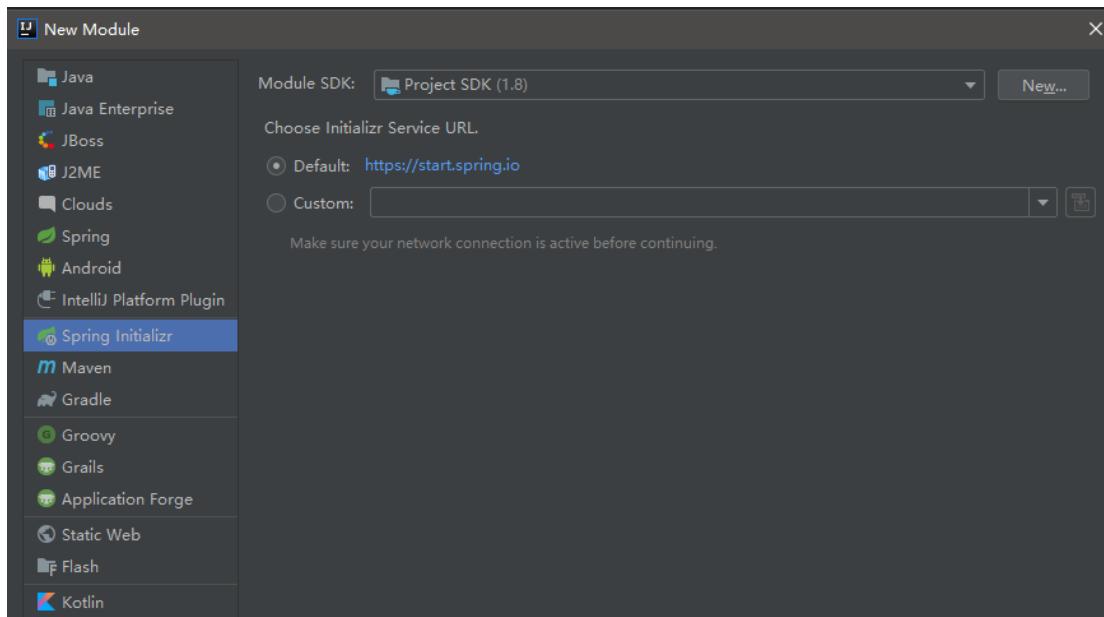
7.2.8 启动应用 SpringBoot，浏览器访问测试

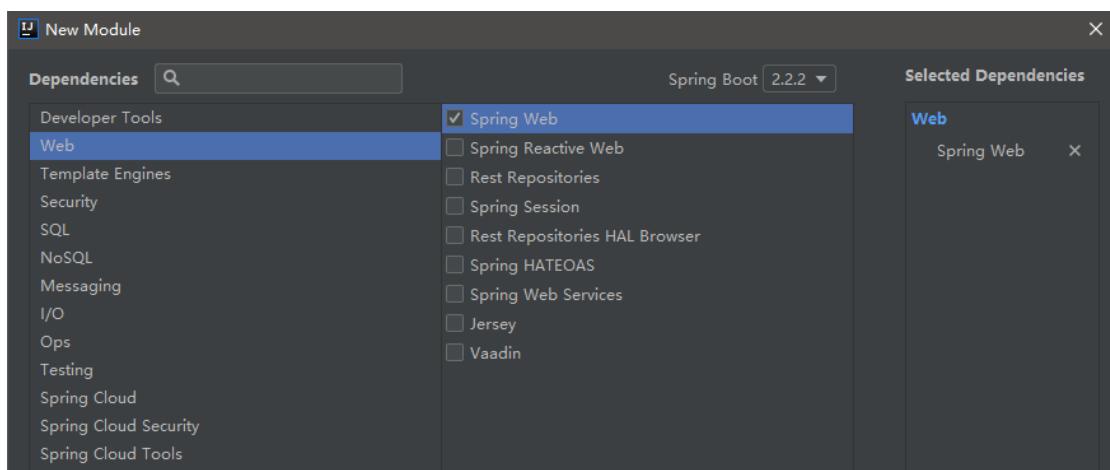
-----通过配置类注册过滤器-----

第8章 Spring Boot 项目配置字符编码

8.1 方式一 使用传统的 Spring 提供的字符编码过滤器

8.1.1 项目名称：032-springboot-character-encoding-filter





8.1.2 创建一个 Servlet

```
public class MyServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
    ServletException, IOException {
        resp.getWriter().print("Hello World, 世界您好! ");
        //设置浏览器编码格式
        resp.setContentType("text/html;character=utf-8");
        resp.getWriter().flush();
        resp.getWriter().close();
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
        doGet(req, resp);
    }
}
```

8.1.3 创建配置类 SystemConfig

```
@Configuration //设置为配置类
public class SystemConfig {

    @Bean
    public ServletRegistrationBean myServletRegistration() {
        ServletRegistrationBean servletRegistrationBean =
            new ServletRegistrationBean(new MyServlet(),
"springboot/myServlet");

        return servletRegistrationBean;
    }

    @Bean
    public FilterRegistrationBean characterFilterRegistration() {
        //设置字符编码过滤器
        //CharacterEncoding 是由 Spring 提供的一个字符编码过滤器，之前是配置在
        web.xml 文件中
        CharacterEncodingFilter characterEncodingFilter = new
        CharacterEncodingFilter();
        //强制使用指定字符编码
        characterEncodingFilter.setForceEncoding(true);
        //设置指定字符编码
        characterEncodingFilter.setEncoding("UTF-8");

        //创建过滤器注册 bean
        FilterRegistrationBean filterRegistrationBean = new
        FilterRegistrationBean();
        //设置字符编码过滤器
        filterRegistrationBean.setFilter(characterEncodingFilter);
        //设置字符编码过滤器路径
        filterRegistrationBean.addUrlPatterns("/*");

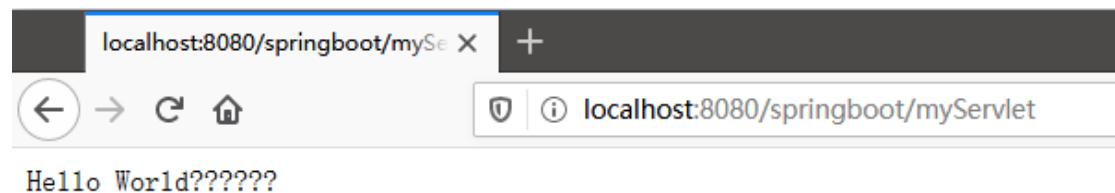
        return filterRegistrationBean;
    }
}
```

8.1.4 关闭 SpringBoot 的 http 字符编码支持

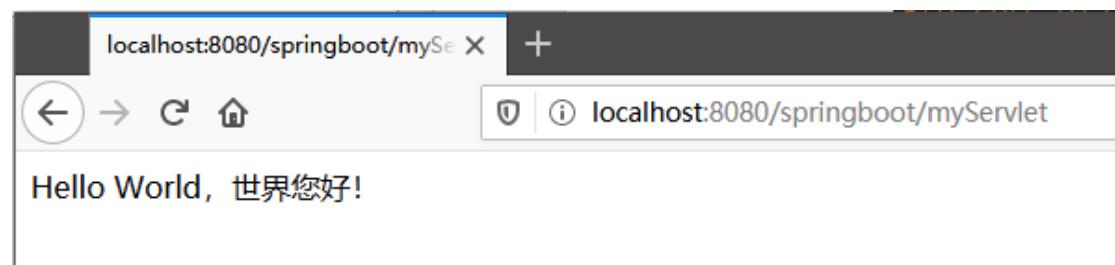
```
#关闭 springboot 的 http 字符编码支持  
#只有关闭该选项后, spring 字符编码过滤器才生效  
spring.http.encoding.enabled=false
```

8.1.5 测试

无字符编码过滤器



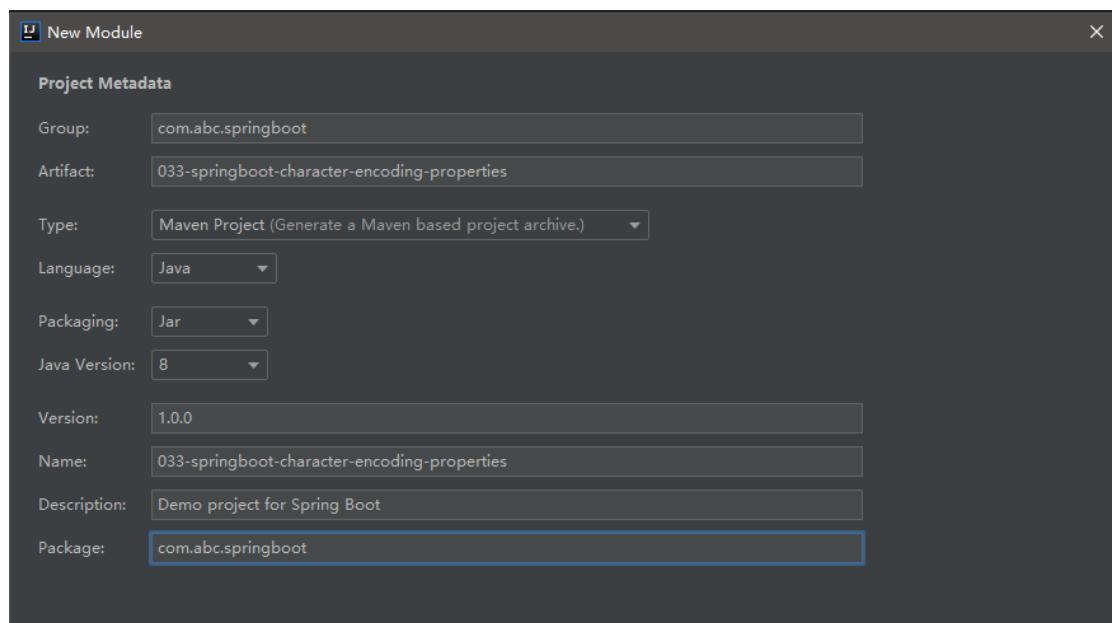
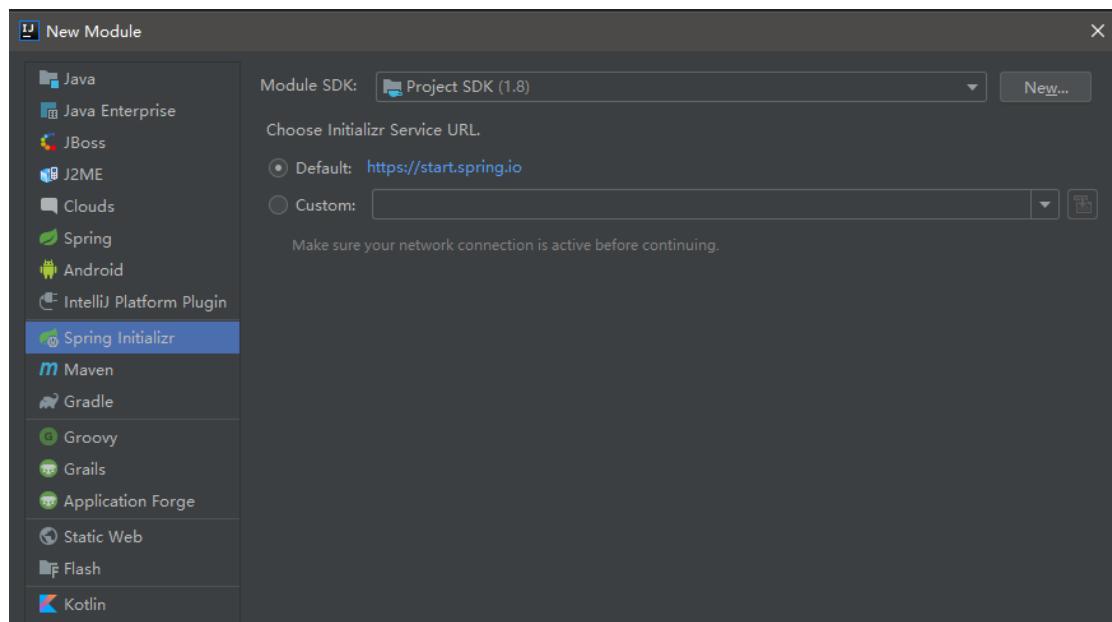
使用字符编码过滤器

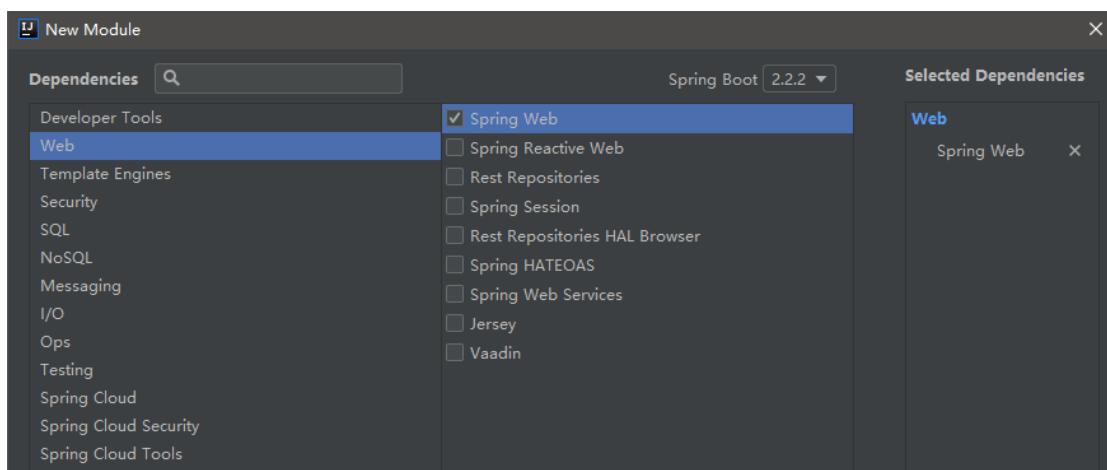


在 servlet 中添加 `response.setContentType("text/html;charset=utf-8")` 指定浏览器编码方式

8.2 方式二 在 application.properties 中配置字符编码（推荐）

8.2.6 项目名称：033-springboot-character-encoding-properties





8.2.7 SpringBoot 核心配置文件添加字符编码设置

从 springboot 1.4.2 之后开始新增的一种字符编码设置

```
#设置请求响应的字符编码
spring.http.encoding.enabled=true
spring.http.encoding.force=true
spring.http.encoding.charset=UTF-8
```

8.2.8 创建 Servlet

```
public class MyServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.getWriter().print("Hello World!世界您真好！");
        resp.setContentType("text/html;character=utf-8");
        resp.getWriter().flush();
        resp.getWriter().close();
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        doGet(req, resp);
    }
}
```

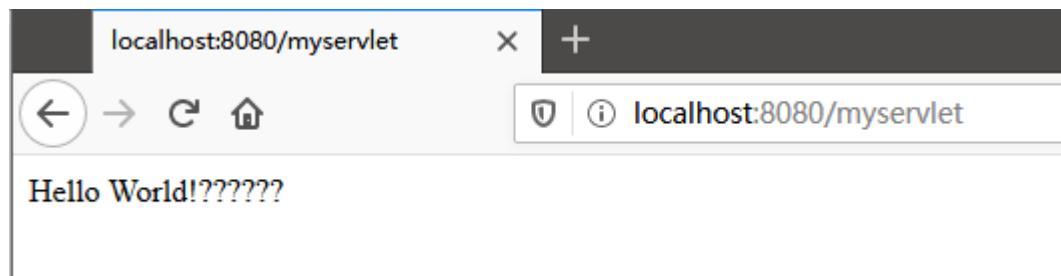
8.2.9 创建配置类 ServletConfig

```
@Configuration
public class ServletConfig {

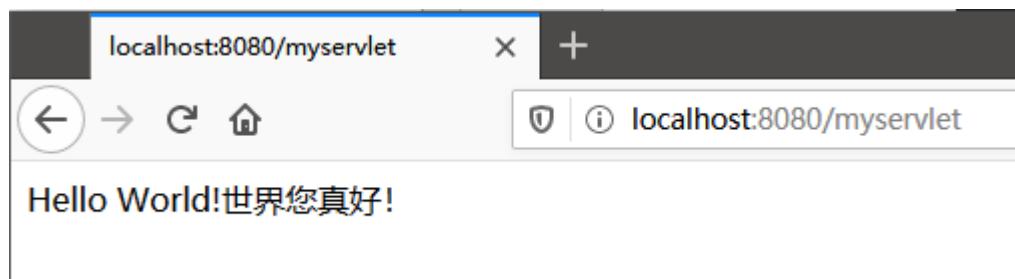
    @Bean
    public ServletRegistrationBean myServletRegistration() {
        ServletRegistrationBean servletRegistrationBean =
            new ServletRegistrationBean(new MyServlet(), "/myservlet");
        return servletRegistrationBean;
    }
}
```

8.2.10 测试

未添加字符编码设置



已添加字符编码设置



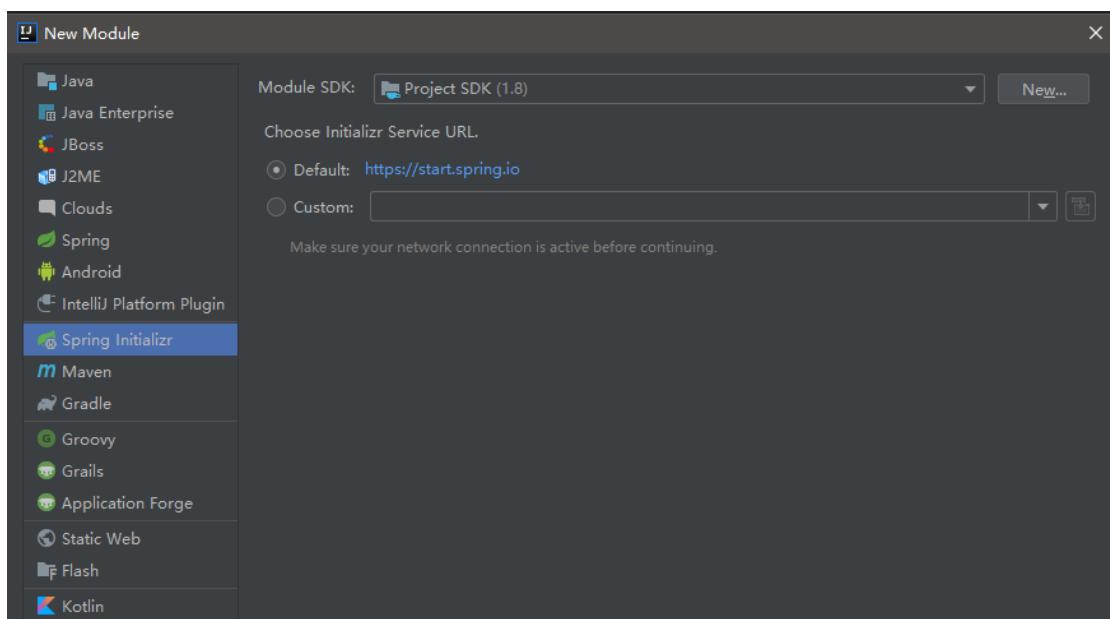
第9章 Spring Boot 打包与部署

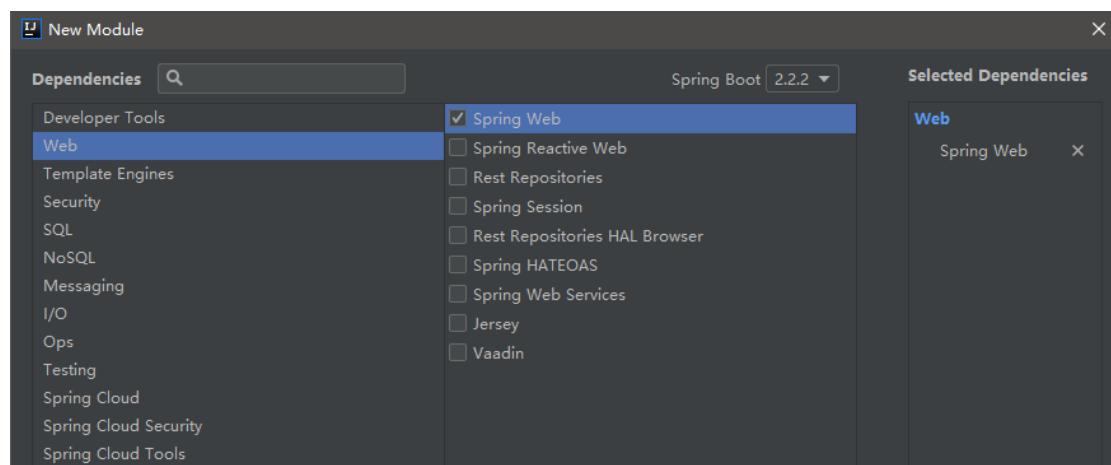
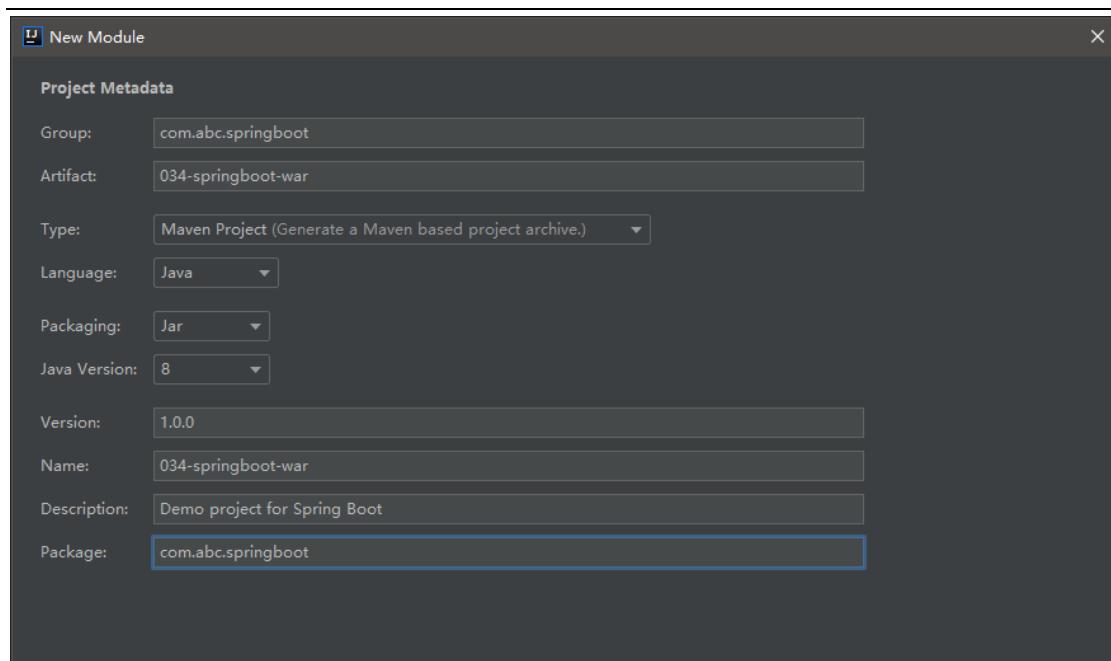
9.1 Spring Boot 程序 war 包部署

项目名称: 034-springboot-war

9.1.1 创建 Spring Boot Web 项目

(1) 创建一个新的 Module





(2) 添加 SpringBoot 解析 jsp 依赖

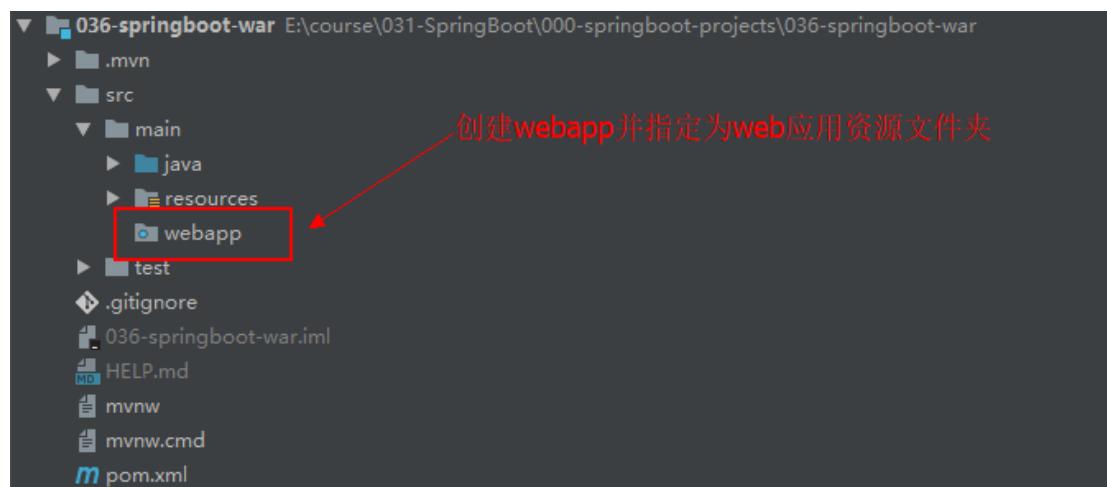
```
<!--SpringBoot 只解析 JSP 页面依赖-->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

(3) 在 pom.xml 文件中配置 jsp 文件解析目录

```
<!--
    SpringBoot 要求 jsp 文件必须编译到指定的 META-INF/resources 目录下，否则不能访问
-->

<resources>
    <resource>
        <!--源文件位置-->
        <directory>src/main/webapp</directory>
        <!--指定编译到 META-INF/resources 目录下，该目录不能随便编写-->
        <targetPath>META-INF/resources</targetPath>
        <!--指定包含文件-->
        <includes>
            <include>**/*.*</include>
        </includes>
    </resource>
</resources>
```

(4) 创建 webapp 并指定为 web 资源文件夹



(5) 在 application.properties 配置文件中配置 jsp 的前后缀

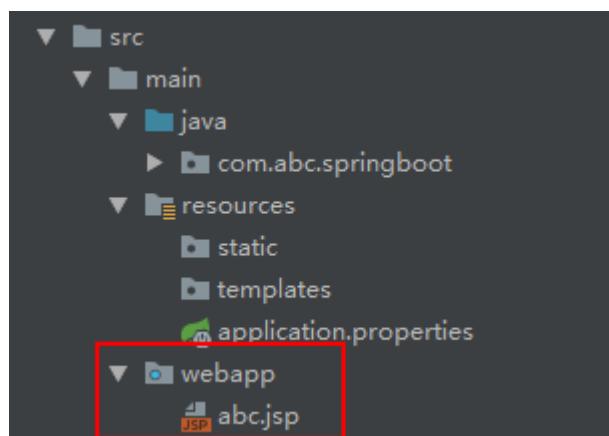
```
#设置 jsp 的前/后缀
spring.mvc.view.prefix=/
spring.mvc.view.suffix=.jsp
```

(6) 创建 IndexController 提供方法分别返回字符串及跳转页面

```
@Controller
public class WarController {

    @RequestMapping(value = "/abc")
    public String index(Model model) {
        model.addAttribute("data", "SpringBoot");
        return "abc";
    }
}
```

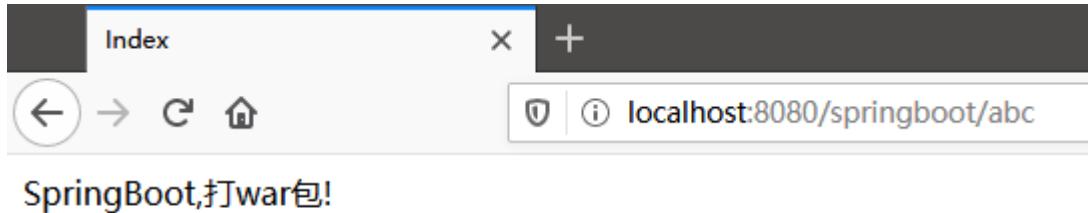
(7) 在 src/main/webapp 目录下创建 index.jsp



abc.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Index</title>
</head>
<body>
    ${data}, 打 war 包!
</body>
</html>
```

(8) 浏览器输入地址访问测试



9.1.2 打 War 包

(9) 程序入口类需扩展继承 `SpringServletInitializer` 类并覆盖 `configure` 方法

```
@SpringBootApplication
public class Application extends SpringServletInitializer {

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

    @Override
    protected SpringApplicationBuilder
    configure(SpringApplicationBuilder builder) {
        //参数为当前 SpringBoot 启动类
        return builder.sources(Application.class);
    }
}
```

(10) 在 `pom.xml` 中添加（修改）打包方式为 war

```
<packaging>war</packaging>
```

(11) 在 pom.xml 中配置 springboot 打包的插件(默认自动加)

```
<!--SpringBoot 打包插件-->
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

(12) 在 pom.xml 中配置将配置文件编译到类路径

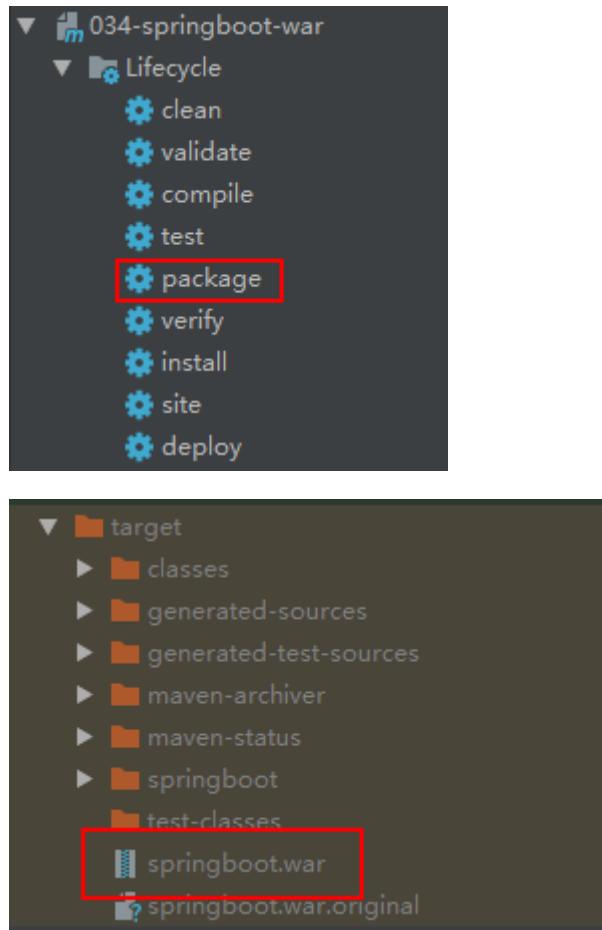
```
<resource>
    <!--源文件夹-->
    <directory>src/main/webapp</directory>
    <!--目标文件夹-->
    <targetPath>META-INF/resources</targetPath>
    <!--包含的文件-->
    <includes>
        <include>**/*.*</include>
    </includes>
</resource>

<!--mybatis 的 mapper.xml-->
<resource>
    <directory>src/main/java</directory>
    <includes>
        <include>**/*.xml</include>
    </includes>
</resource>
<!--src/main/resources 下的所有配置文件编译到 classes 下面去-->
<resource>
    <directory>src/main/resources</directory>
    <includes>
        <include>**/*.*</include>
    </includes>
</resource>
```

(13) 在 pom.xml 的 build 标签下通过 finalName 指定打 war 包的名字

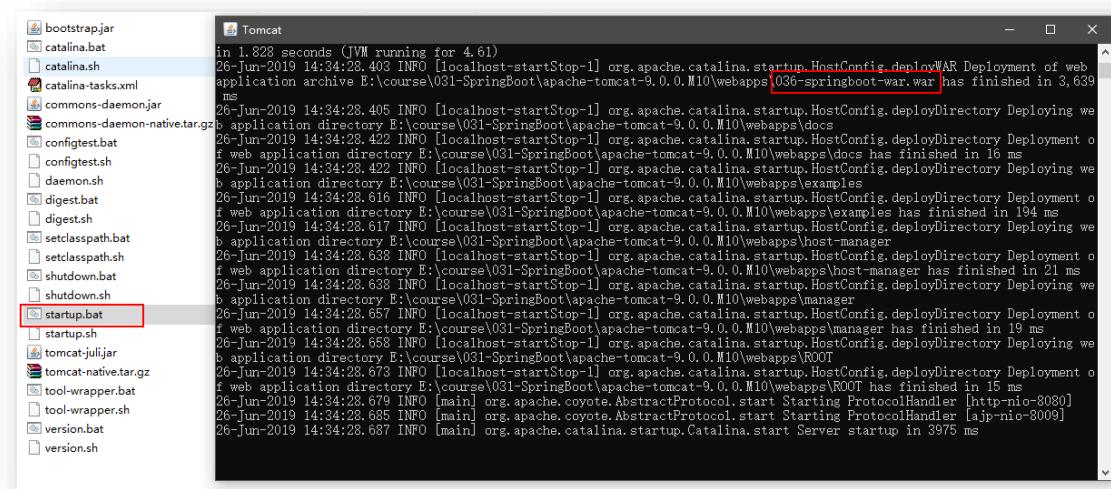
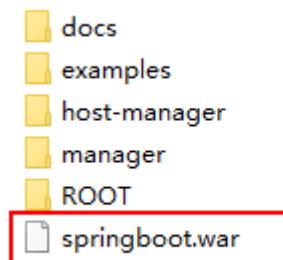
```
<!--指定打war包的名字-->
<finalName>springboot</finalName>
```

(14) 通过 Maven package 命令打 war 包到 target 目录下

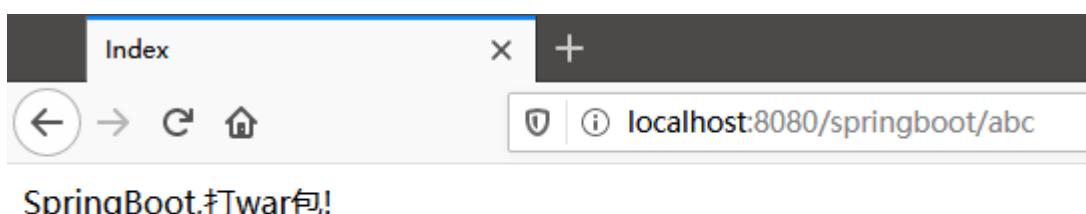


9.1.3 部署到 Tomcat 服务器上测试运行

(15) 将 target 目录下生成的 war 包拷贝到 tomcat 的 webapps 目录，并启动 tomcat



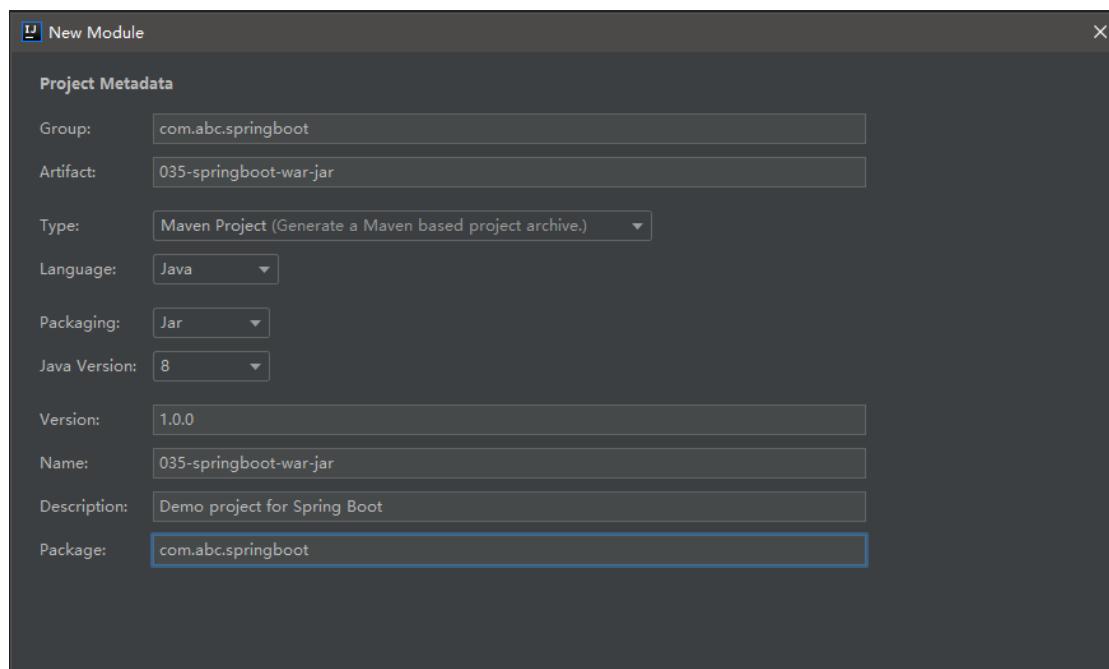
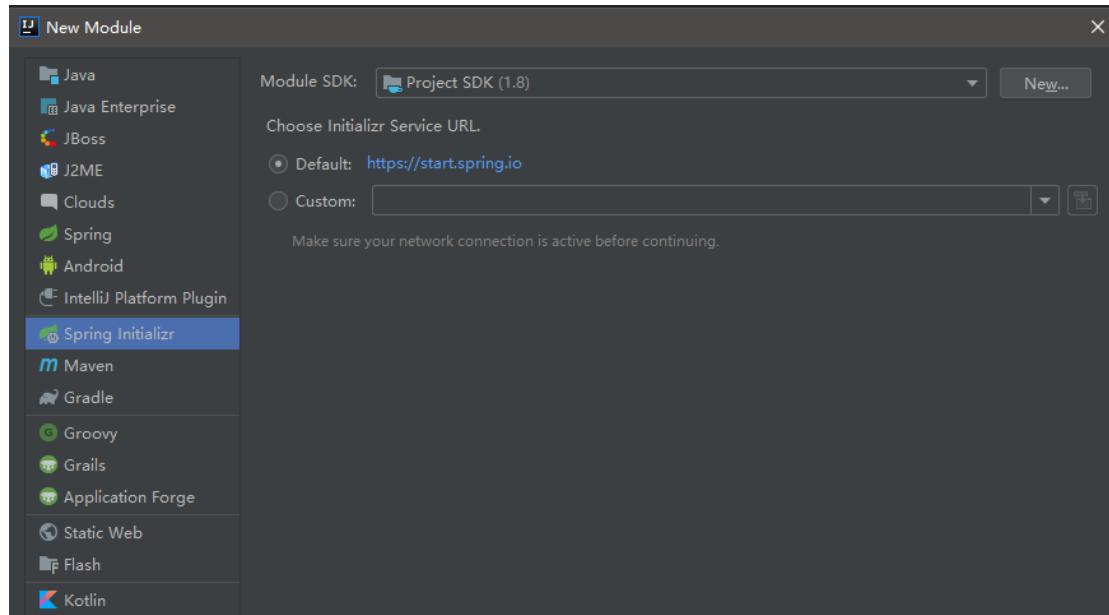
(16) 通过浏览器访问

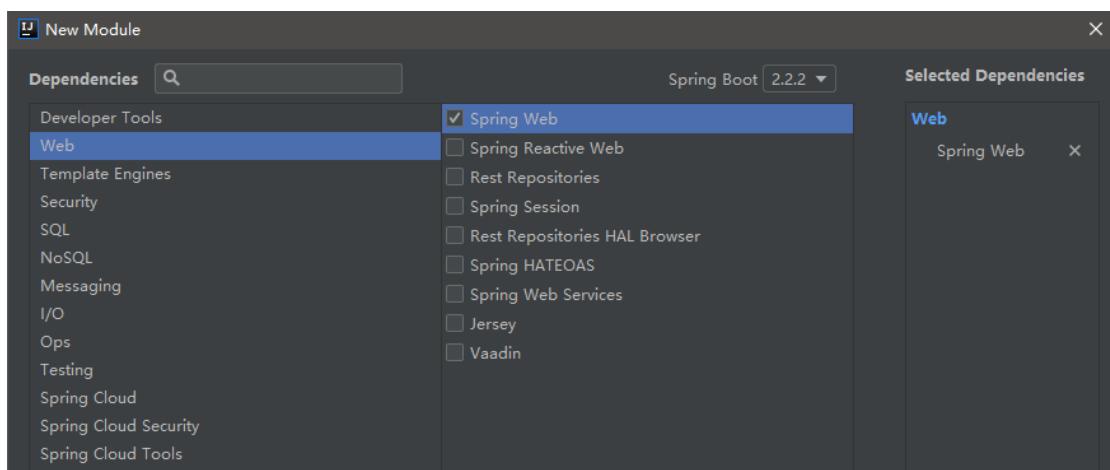


9.2 Spring Boot 程序打 Jar 包与运行

项目名称: 035-springboot-web-jar

因为 SpringBoot 默认打包方式就是 jar 包,所以我们直接执行 Maven 的 package 命令就行了





9.2.1 在 pom.xml 文件中添加 Tomcat 解析 jsp 依赖

```
<!--SpringBoot 项目内嵌 tomcat 对 jsp 的解析包-->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

9.2.2 在 pom.xml 文件中添加 resources 配置，以后为了保险起见，大家在打包的时候，建议把下面的配置都加上

```
<resource>
    <!--源文件夹-->
    <directory>src/main/webapp</directory>
    <!--目标文件夹-->
    <targetPath>META-INF/resources</targetPath>
    <!--包含的文件-->
    <includes>
        <include>**/*.*</include>
    </includes>
</resource>

<!--mybatis 的 mapper.xml-->
<resource>
    <directory>src/main/java</directory>
```

```
<includes>
    <include>**/*.xml</include>
</includes>
</resource>
<!--src/main/resources 下的所有配置文件编译到 classes 下面去-->
<resource>
    <directory>src/main/resources</directory>
    <includes>
        <include>**/*.*</include>
    </includes>
</resource>
```

9.2.3 修改 pom.xml 文件中打包插件的版本

默认 SpringBoot 提供的打包插件版本为 2.2.2.RELEASE，这个版本打的 jar 包 jsp 不能访问，
我们这里修改为 1.4.2.RELEASE（其它版本测试都有问题）

```
<!-- SpringBoot 提供打包编译插件 -->
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>1.4.2.RELEASE</version>
</plugin>
```

9.2.4 修改 application.properties 配置文件

```
#设置内嵌 Tomcat 端口号
server.port=9090
#设置项目上下文根
server.servlet.context-path=/

#配置 jsp 的前/后缀
spring.mvc.view.prefix=/
spring.mvc.view.suffix=.jsp
```

9.2.5 在 com.abc.springboot.web 包下创建 IndexController

```
@Controller
public class IndexController {

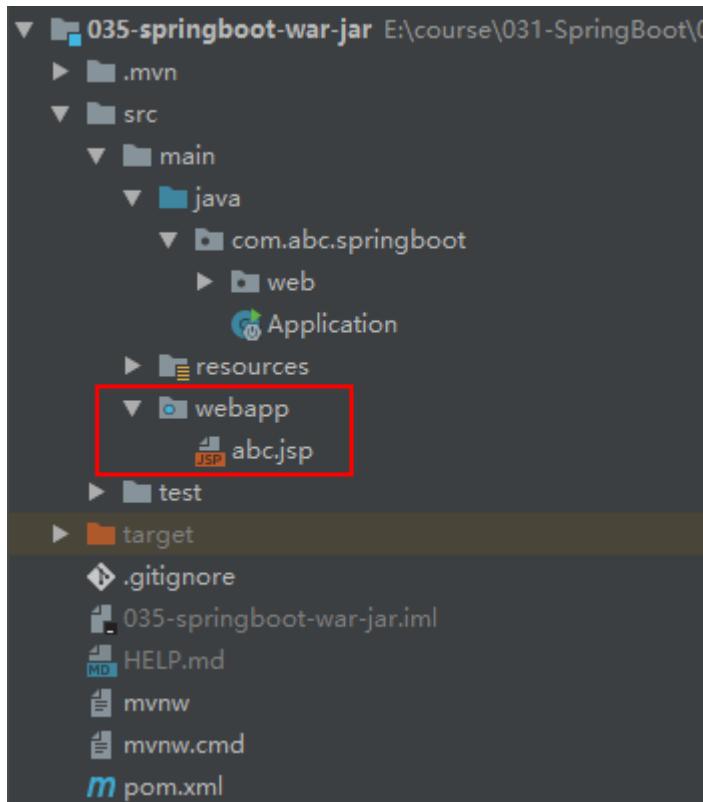
    @RequestMapping(value = "/abc")
    public String abc(Model model) {
        model.addAttribute("data", "SpringBoot 框架打 jar 运行");
        return "abc";
    }

    @RequestMapping(value = "/abc/json")
    public @ResponseBody Object json() {

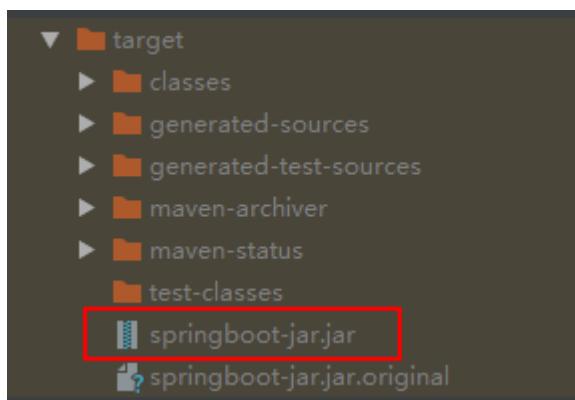
        Map<String, Object> paramMap = new HashMap<String, Object>();
        paramMap.put("code", "10000");

        return paramMap;
    }
}
```

9.2.6 创建 webapp 并指定为 web 资源目录

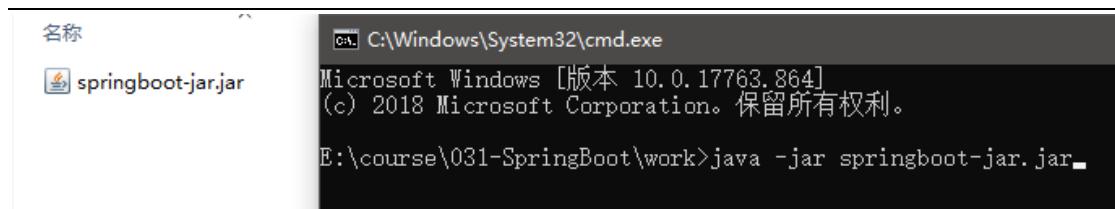


9.2.7 通过 maven package 打包



9.2.8 通过 java 命令执行 jar 包，相当于启动内嵌 tomcat

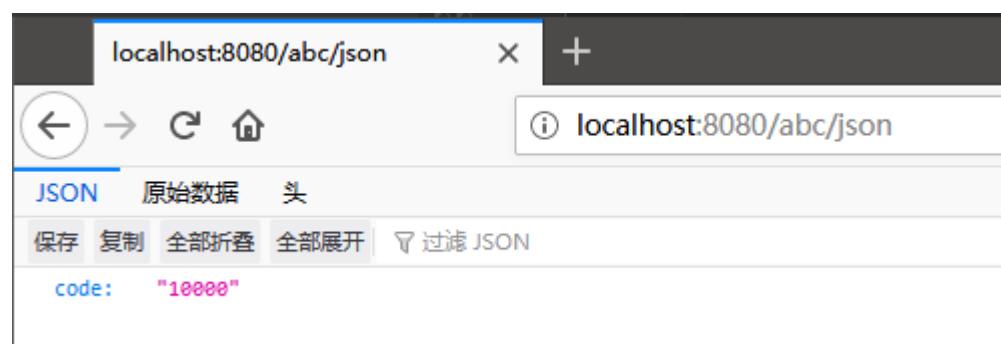
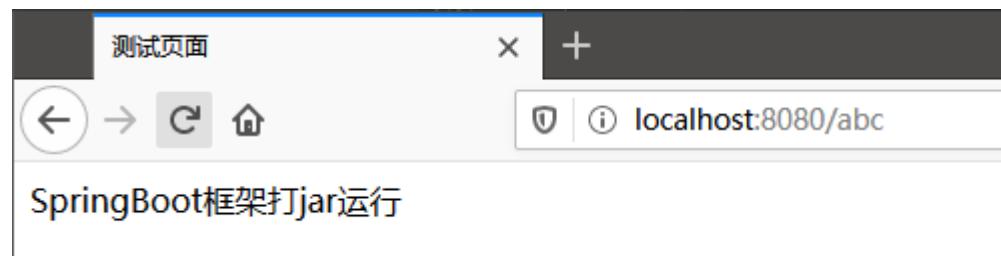
将 target 下的 jar 包拷贝到某一个目录，在该目录下执行 `java -jar jar 包名称`



```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.17763.864]
(c) 2018 Microsoft Corporation. 保留所有权利。

E:\course\031-SpringBoot\work>java -jar springboot-jar.jar
```

9.2.9 浏览器访问测试



9.3 Spring Boot 部署与运行方式总结

- 在 IDEA 中直接运行 Spring Boot 程序的 main 方法（开发阶段）
- 用 maven 将 Spring Boot 安装为一个 jar 包，使用 Java 命令运行

java -jar springboot-xxx.jar

可以将该命令封装到一个 Linux 的一个 shell 脚本中（上线部署）

- 写一个 shell 脚本(run.sh):

```
#!/bin/sh
```

```
java -jar xxx.jar
```

- 赋权限 chmod 777 run.sh
- 启动 shell 脚本： ./run.sh

```
-rw-r--r--. 1 root root 20239259 Jun 26 15:06 037-springboot-web-jar-1.0.0.jar
-rwxrwxrwx. 1 root root      53 Jun 26 15:18 run.sh
[root@localhost springboot-projects]# rm -rf *
[root@localhost springboot-projects]# ll
total 0
[root@localhost springboot-projects]# rz
[]z waiting to receive.*$B0100000023be50
[root@localhost springboot-projects]# ll
total 20596
-rw-r--r--. 1 root root 21088238 Dec 12 10:46 springboot-jar.jar
[root@localhost springboot-projects]# vim run.sh
[root@localhost springboot-projects]# ll
total 20600
-rw-r--r--. 1 root root      39 Dec 12 10:51 run.sh
-rw-r--r--. 1 root root 21088238 Dec 12 10:46 springboot-jar.jar
[root@localhost springboot-projects]# chmod 777 run.sh
[root@localhost springboot-projects]# ll
total 20600
-rwxrwxrwx. 1 root root      39 Dec 12 10:51 run.sh
-rw-r--r--. 1 root root 21088238 Dec 12 10:46 springboot-jar.jar
[root@localhost springboot-projects]# ./run.sh
```



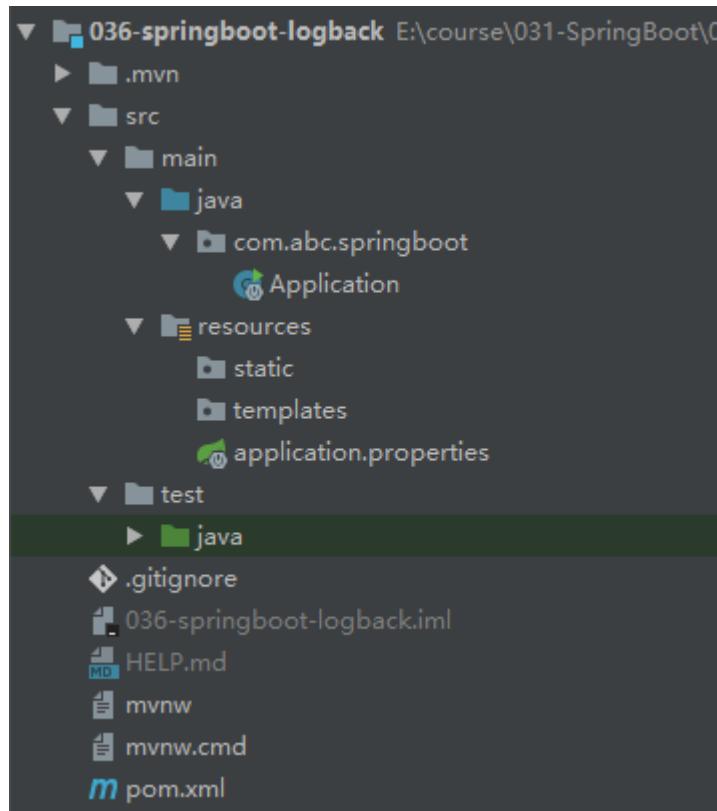
使用 Spring Boot 的 maven 插件将 Springboot 程序打成 war 包，单独部署在 tomcat 中运行(上线部署常用)

第10章 SpringBoot 集成 logback 日志

10.1 使用该功能步骤

项目名称：036-springboot-logback

10.2 创建 SpringBoot 框架 web 项目



10.3 在项目 pom.xml 中添加 SSM 需要的依赖

```
<!--MyBatis 集成 SpringBoot 框架的起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>

<!--连接 MySQL 的驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>

<!--SpringBoot 项目内嵌 tomcat 对 jsp 的解析包-->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
```

```
<artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

10.4 编写集成 SSM 在 application.properties 的配置

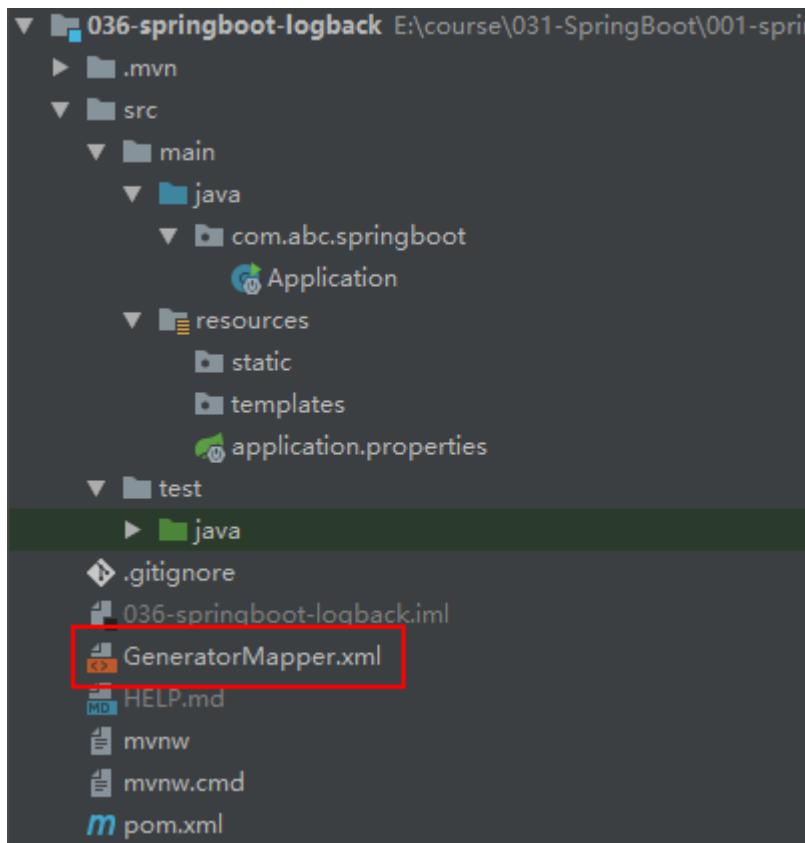
```
#设置内嵌 Tomcat 端口号
server.port=9090
#设置项目上下文根
server.servlet.context-path=/

#配置 jsp 的前/后缀
spring.mvc.view.prefix=/
spring.mvc.view.suffix=.jsp

#配置连接 MySQL 数据库信息
spring.datasource.url=jdbc:mysql://192.168.92.134:3306/springboot?useUnicode
=true&characterEncoding=UTF-8&useJDBCCompliantTimezoneShift=true&useLegacyDa
tetimeCode=false&serverTimezone=GMT%2B8
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=123456
```

10.5 通过 MyBatis 逆向工程生成 DAO

将逆向生成文件放到项目根目录



内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration
1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>

    <!-- 指定连接数据库的 JDBC 驱动包所在位置，指定到你本机的完整路径 -->
    <classPathEntry location="E:\mysql-connector-java-5.1.38.jar"/>

    <!-- 配置 table 表信息内容体，targetRuntime 指定采用 MyBatis3 的版本 -->
    <context id="tables" targetRuntime="MyBatis3">

        <!-- 抑制生成注释，由于生成的注释都是英文的，可以不让它生成 -->
        <commentGenerator>
            <property name="suppressAllComments" value="true" />
        </commentGenerator>

        <!-- 配置数据库连接信息 -->
```

```
<jdbcConnection driverClass="com.mysql.jdbc.Driver"

connectionURL="jdbc:mysql://localhost:3306/springboot"
    userId="root"
    password="123456">
</jdbcConnection>

<!-- 生成 model 类, targetPackage 指定 model 类的包名, targetProject 指定生成的 model 放在 eclipse 的哪个工程下面-->
<javaModelGenerator targetPackage="com.abc.springboot.model"
targetProject="src/main/java">
    <property name="enableSubPackages" value="false" />
    <property name="trimStrings" value="false" />
</javaModelGenerator>

<!-- 生成 MyBatis 的 Mapper.xml 文件, targetPackage 指定 mapper.xml 文件的包名, targetProject 指定生成的 mapper.xml 放在 eclipse 的哪个工程下面 -->
<sqlMapGenerator targetPackage="com.abc.springboot.mapper"
targetProject="src/main/java">
    <property name="enableSubPackages" value="false" />
</sqlMapGenerator>

<!-- 生成 MyBatis 的 Mapper 接口类文件, targetPackage 指定 Mapper 接口类的包名, targetProject 指定生成的 Mapper 接口放在 eclipse 的哪个工程下面 -->
<javaClientGenerator type="XMLMAPPER"
targetPackage="com.abc.springboot.mapper"
targetProject="src/main/java">
    <property name="enableSubPackages" value="false" />
</javaClientGenerator>

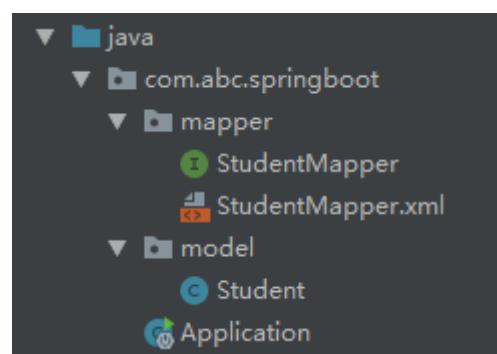
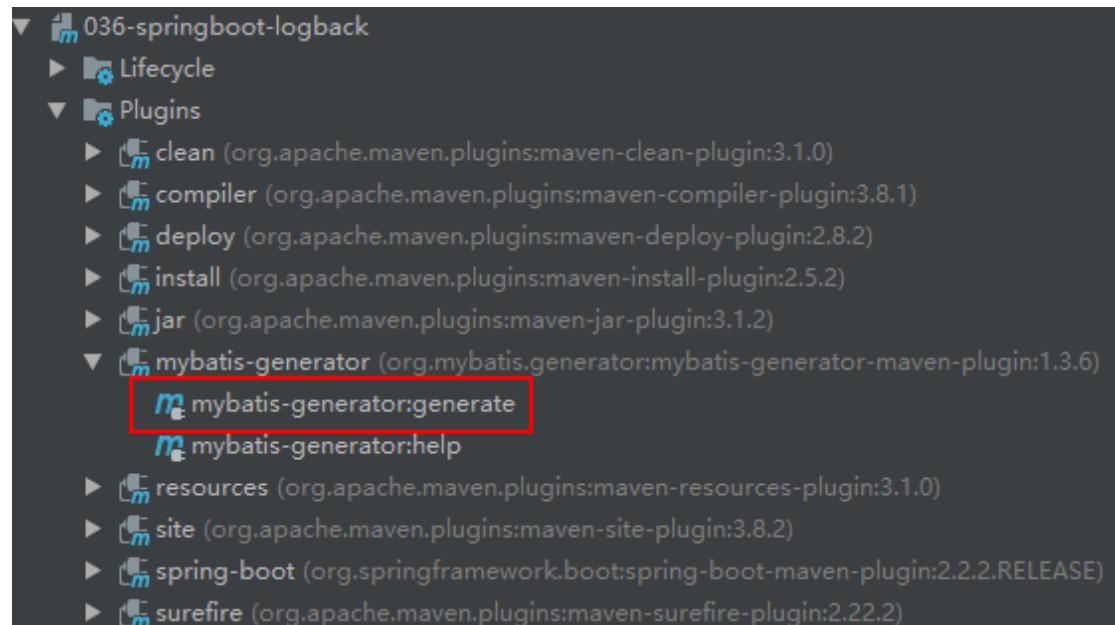
<!-- 数据库表名及对应的 Java 模型类名 -->
<table tableName="t_student" domainObjectName="Student"
    enableCountByExample="false"
    enableUpdateByExample="false"
    enableDeleteByExample="false"
    enableSelectByExample="false"
    selectByExampleQueryId="false"/>
</context>

</generatorConfiguration>
```

在 pom.xml 文件 -> build -> plugins 中添加插件

```
<!--mybatis 代码自动生成插件-->
<plugin>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-maven-plugin</artifactId>
    <version>1.3.6</version>
    <configuration>
        <!--配置文件的位置-->
        <configurationFile>GeneratorMapper.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
    </configuration>
</plugin>
```

双击生成



10.6 手动指定资源文件夹

```
<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.xml</include>
        </includes>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.*</include>
        </includes>
    </resource>
    <resource>
        <directory>src/main/webapp</directory>
        <targetPath>META-INF/resources</targetPath>
        <includes>
            <include>**/*.*</include>
        </includes>
    </resource>
</resources>
```

10.7 编写 StudentController

```
@Slf4j
@Controller
public class StudentController {
    @Autowired
    private StudentService studentService;

    @RequestMapping(value = "/student/count")
    public @ResponseBody Object allStudentCount() {
        log.info("-----查询开始-----");
        Long allStudentCount = studentService.queryAllStudentCount();
        log.info("-----查询结束-----");

        return "学生总人数为: " + allStudentCount;
    }
}
```

```
    }  
}
```

10.8 编写 StudentService 接口及实现类

```
public interface StudentService {  
  
    /**  
     * 获取学生总人数  
     * @return  
     */  
    Long queryAllStudentCount();  
}
```

```
@Service  
public class StudentServiceImpl implements StudentService {  
  
    @Autowired  
    private StudentMapper studentMapper;  
  
    @Override  
    public Long queryAllStudentCount() {  
        return studentMapper.selectAllStudentCount();  
    }  
}
```

10.9 数据持久层

StudentMapper.java

```
/**  
 * 获取学生总人数  
 * @return  
 */  
Long selectAllStudentCount();
```

StudentMapper.xml

```
<!--获取学生总人数-->
<select id="selectAllStudentCount" resultType="java.lang.Long">
    select count(*) from t_student
</select>
```

10.10 日志文件

Spring Boot 官方推荐优先使用带有 `-spring` 的文件名作为你的日志配置（如使用 `logback-spring.xml`，而不是 `logback.xml`），命名为 `logback-spring.xml` 的日志配置文件。

默认的命名规则，并且放在 `src/main/resources` 下如果你即想完全掌控日志配置，但又不想用 `logback.xml` 作为 Logback 配置的名字，`application.yml` 可以通过 `logging.config` 属性指定自定义的名字：

```
logging.config=classpath:logging-config.xml
```

(1) 我们一般针对 DAO 的包进行 DEBUG 日志设置：

```
<logger name="com.abc.springboot.mapper" level="DEBUG" />
```

这样的话，只打印 SQL 语句：

```
[BaseJdbcLogger.java : 143] ==> Preparing: select count(*) from t_student
[BaseJdbcLogger.java : 143] ==> Parameters:
[BaseJdbcLogger.java : 143] <== Total: 1
```

(2) 代码里打印日志

之前我们大多数时候自己在每个类创建日志对象去打印信息，比较麻烦：

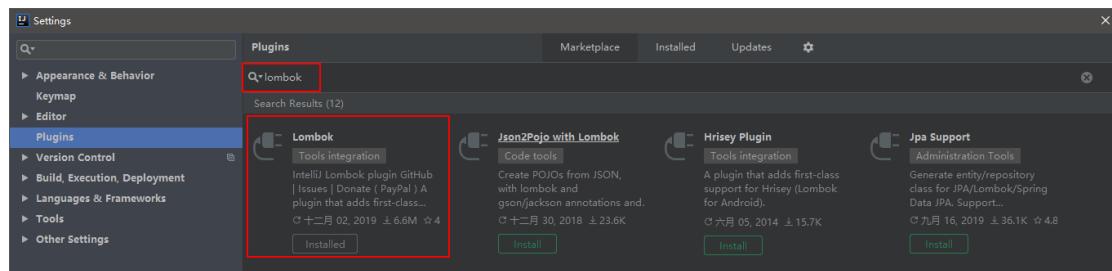
```
private static final Logger logger = LoggerFactory.getLogger(StudentServiceImpl.class);
logger.error("xxx");
```

现在可以直接在类上通过 `@Slf4j` 标签去声明式注解日志对象

A、在 pom.xml 中添加依赖

```
<!--@Slf4j 自动化日志对象-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.2</version>
</dependency>
```

B、添加 lombok 插件



(3) logback-spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 日志级别从低到高分为 TRACE < DEBUG < INFO < WARN < ERROR < FATAL，如果设置为 WARN，则低于 WARN 的信息都不会输出 -->
<!-- scan:当此属性设置为 true 时，配置文件如果发生改变，将会被重新加载，默认值为 true -->
<!-- scanPeriod:设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒。当 scan 为 true 时，此属性生效。默认的时间间隔为 1 分钟。 -->
<!-- debug:当此属性设置为 true 时，将打印出 logback 内部日志信息，实时查看 logback 运行状态。默认值为 false。通常不打印 -->
<configuration scan="true" scanPeriod="10 seconds">

    <!--输出到控制台-->
    <appender name="CONSOLE"
    class="ch.qos.logback.core.ConsoleAppender">
        <!--此日志 appender 是为开发使用，只配置最底级别，控制台输出的日志级别是大于或等于此级别的日志信息-->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>debug</level>
```

```
</filter>
<encoder>
    <Pattern>%date [%-5p] [%thread] %logger{60}
[%file : %line] %msg%n</Pattern>
    <!-- 设置字符集 -->
    <charset>UTF-8</charset>
</encoder>
</appender>

<appender name="FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
    <!--<File>/home/log/stdout.log</File>-->
    <File>D:/log/stdout.log</File>
    <encoder>
        <pattern>%date [%-5p] %thread %logger{60}
[%file : %line] %msg%n</pattern>
    </encoder>
    <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- 添加.gz 历史日志会启用压缩 大大缩小日志文件所占空间 -->

<!--<fileNamePattern>/home/log/stdout.log.%d{yyyy-MM-dd}.log</fileNamePattern>-->

<fileNamePattern>D:/log/stdout.log.%d{yyyy-MM-dd}.log</fileNamePattern>
        <maxHistory>30</maxHistory><!-- 保留 30 天日志 -->
    </rollingPolicy>
</appender>

<logger name="com.abc.springboot.mapper" level="DEBUG" />

<root level="INFO">
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
</root>
</configuration>
```

10.11 Application 启动类

```
@SpringBootApplication
@MapperScan(basePackages = "com.abc.springboot.mapper")
public class Application {

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

}
```

10.12 启动测试

```
[xServlet.java : 347] Completed initialization in 4 ms
:.java : 25] -----查询开始-----
110] HikariPool-1 - Starting...
123] HikariPool-1 - Start completed.
[BaseJdbcLogger.java : 143] ==> Preparing: select count(*) from t_student
[BaseJdbcLogger.java : 143] ==> Parameters:
[BaseJdbcLogger.java : 143] <==      Total: 1
:.java : 29] -----查询结束-----
```

第11章 SpringBoot 集成 Thymeleaf 模板

11.1 认识 Thymeleaf

Thymeleaf 是一个流行的模板引擎，该模板引擎采用 Java 语言开发。模板引擎是一个技术名词，是跨领域跨平台的概念，在 Java 语言体系下有模板引擎，在 C#、PHP 语言体系下也有模板引擎，甚至在 JavaScript 中也会用到模板引擎技术，Java 生态下的模板引擎有 Thymeleaf、Freemarker、Velocity、Beetl（国产）等。

Thymeleaf 对网络环境不存在严格的要求，既能用于 Web 环境下，也能用于非 Web 环境下。在非 Web 环境下，他能直接显示模板上的静态数据；在 Web 环境下，它能像 Jsp 一样从后台接收数据并替换掉模板上的静态数据。它是基于 HTML 的，以 HTML 标签为载体，Thymeleaf 要寄托在 HTML 标签下实现。

SpringBoot 集成了 Thymeleaf 模板技术，并且 Spring Boot 官方也推荐使用 Thymeleaf 来替代 JSP 技术，Thymeleaf 是另外的一种模板技术，它本身并不属于 Spring Boot，Spring Boot 只是很好地集成这种模板技术，作为前端页面的数据展示，在过去的 Java Web 开发中，我们往往会选择使用 Jsp 去完成页面的动态渲染，但是 jsp 需要翻译编译运行，效率低

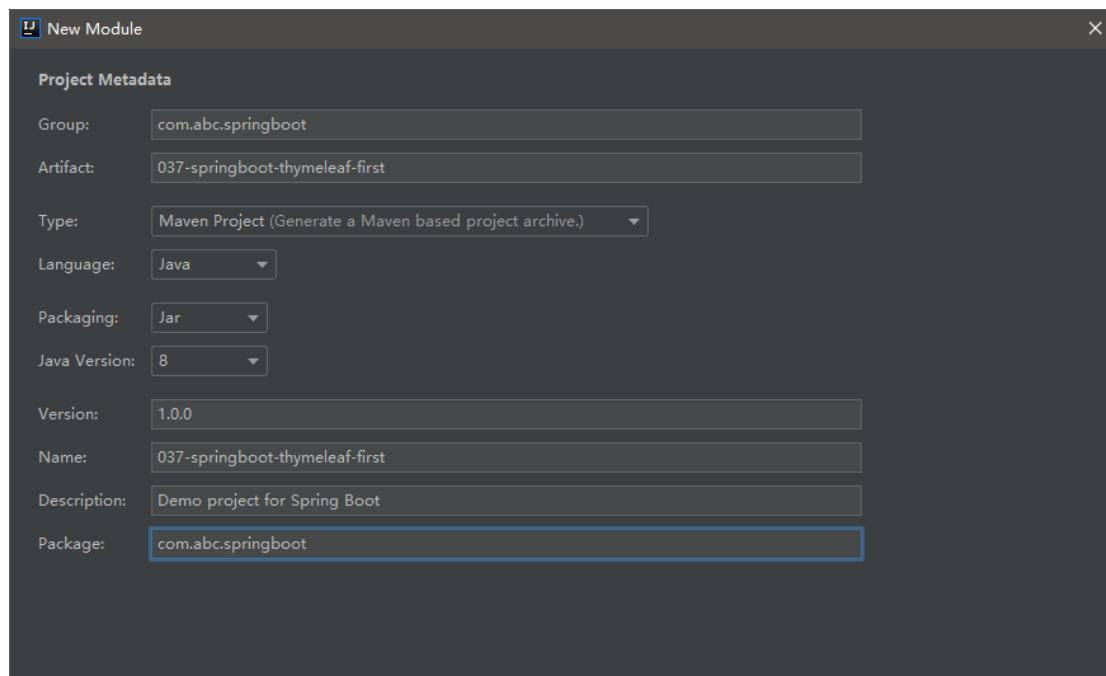
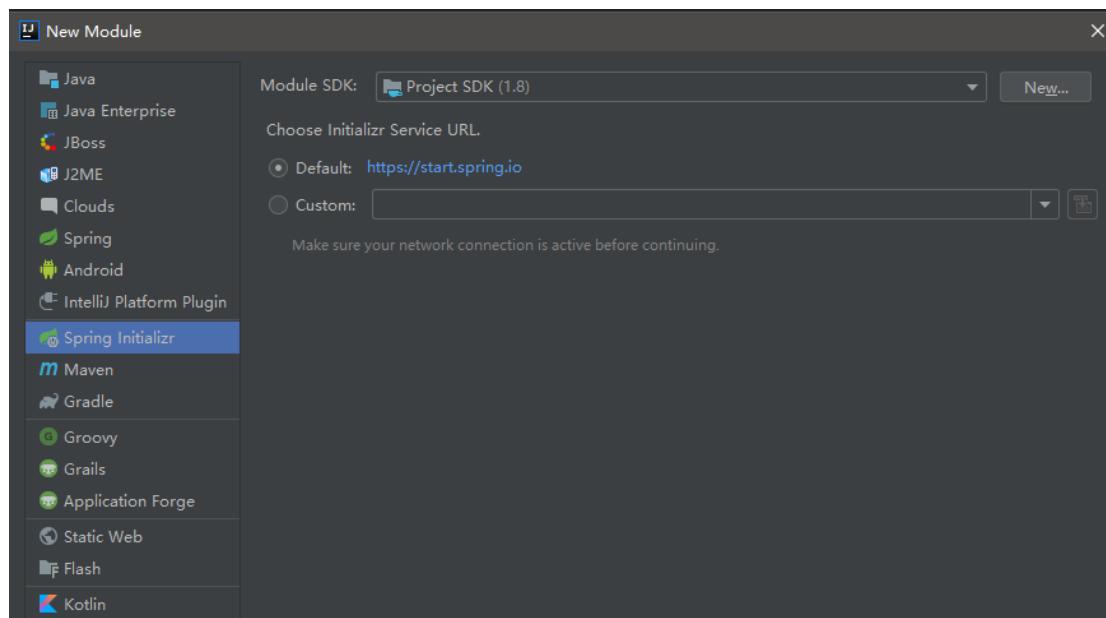
Thymeleaf 的官方网站: <http://www.thymeleaf.org>

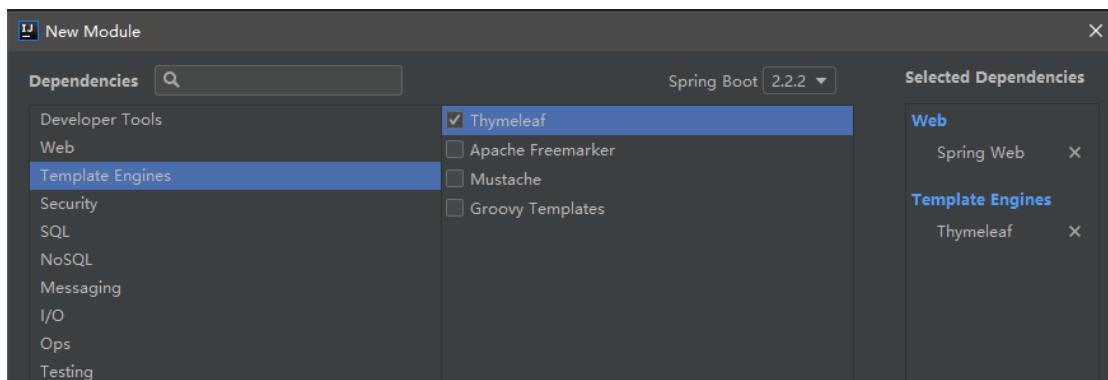
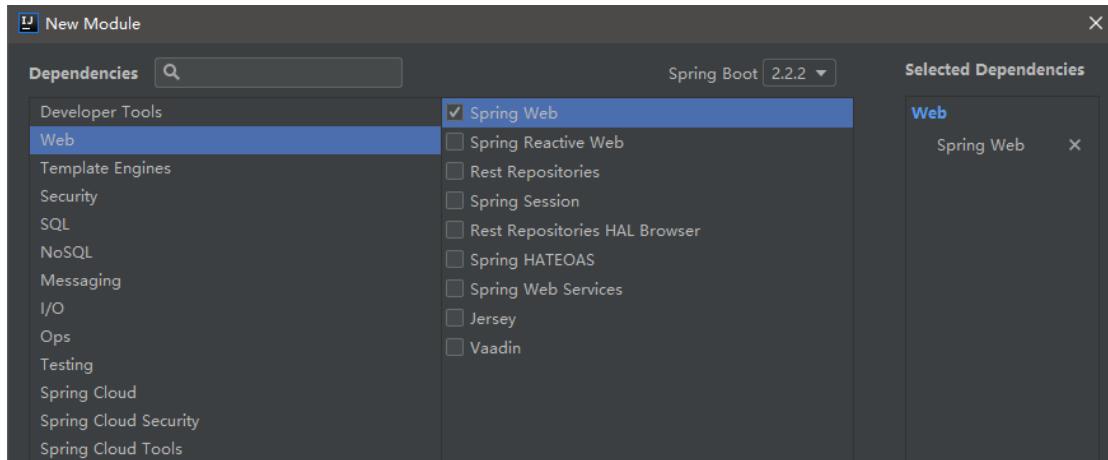
Thymeleaf 官方手册: <https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

11.2 Spring Boot 集成 Thymeleaf

11.2.1 项目名称: 037-springboot-thymeleaf-first

创建 Spring Boot 项目,添加 web 和 Thymeleaf 依赖





按照这种方式创建后，`pom.xml` 文件下会自动添加如下依赖

```
<!--SpringBoot 集成 Thymeleaf 的起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<!--SpringBoot 开发 web 项目的起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

11.2.2 在 Spring boot 的核心配置文件 `application.properties` 中对 Thymeleaf 进行配置



```
#thymeleaf 页面的缓存开关，默认 true 开启缓存
#建议在开发阶段关闭 thymeleaf 页面缓存，目的实时看到页面
spring.thymeleaf.cache=false
```

其实什么都不用配置就可以工作，因为基本 Thymeleaf 的配置都有默认值

前缀：

```
#thymeleaf 模版前缀，默认可以不写
spring.thymeleaf.prefix=classpath:/templates/
```

后缀：

```
#thymeleaf 模版后缀，默认可以不写
spring.thymeleaf.suffix=.html
```

11.2.3 创建 ThymeleafController 去映射到模板页面（和 SpringMVC 基本一致）

```
@Controller
public class ThymeleafController {

    @RequestMapping(value = "/thymeleaf/index")
    public String index(Model model) {

        model.addAttribute("data", "SpringBoot 成功集成 Thymeleaf 模版！");
        return "index";
    }
}
```

11.2.4 在 src/main/resources 的 templates 下新建一个 index.html 页面用于展示数据

HTML 页面的<html>元素中加入以下属性：

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

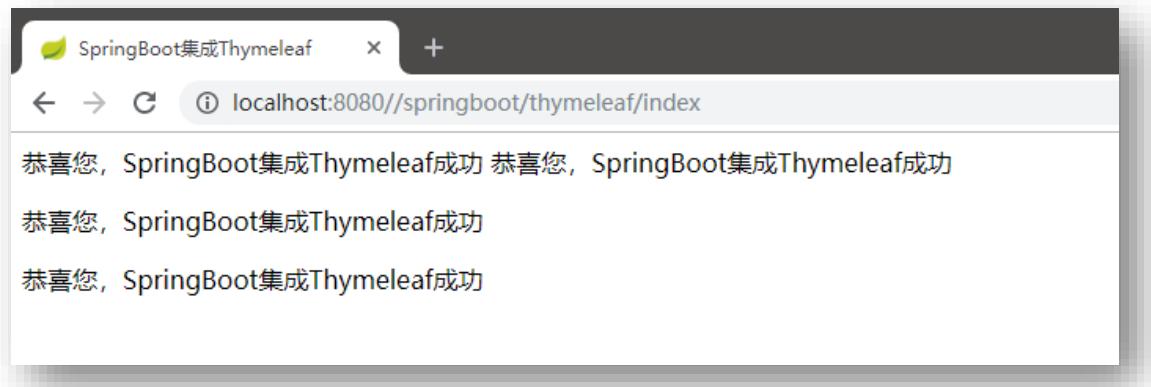
```
<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>SpringBoot 集成 Thymeleaf</title>
</head>
```

```
<body>
    <!--Thymeleaf 前端框架以 Html 为载体--&gt;
    &lt;span th:text="${data}"&gt;&lt;/span&gt;
    &lt;span th:text="${data}"&gt;&lt;/span&gt;
    &lt;p th:text="${data}"&gt;&lt;/p&gt;
    &lt;div th:text="${data}"&gt;&lt;/div&gt;

&lt;/body&gt;
&lt;/html&gt;</pre>
```

11.2.5 启动程序，浏览器访问 <http://localhost:8080/index>



右键->查看页面源代码



The screenshot shows a browser window with two tabs: "SpringBoot集成Thymeleaf" and "view-source:localhost:8080//springboot/thymeleaf/index". The main content area displays the source code of a Thymeleaf template (index.html). The code includes standard HTML tags like DOCTYPE, head, and body, along with Thymeleaf-specific directives. A red arrow points from the text "通过th:text获取的数据是存放到标签体中的" to the line of code where a span element contains the text "恭喜您，SpringBoot集成Thymeleaf成功".

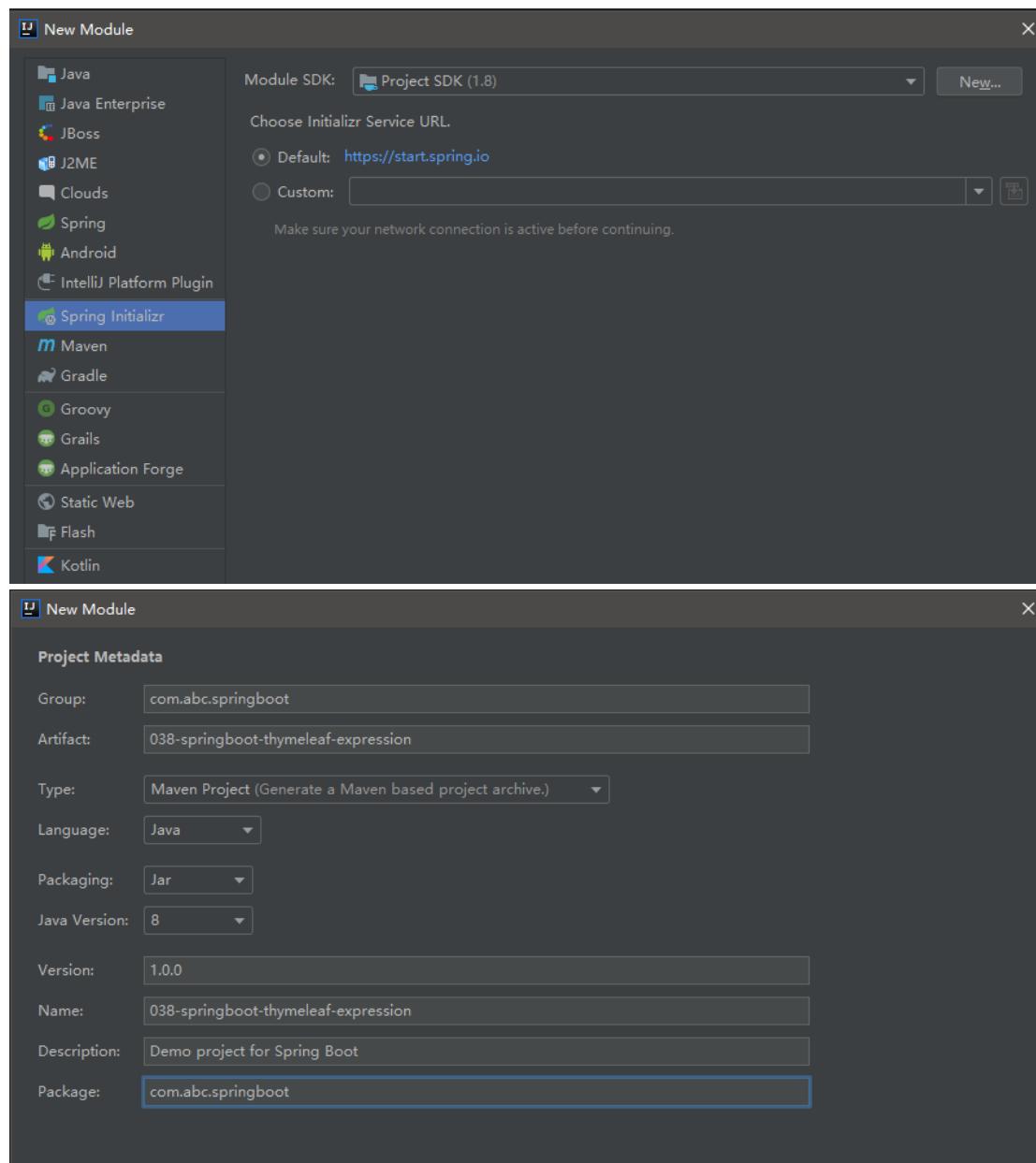
```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>SpringBoot集成Thymeleaf</title>
6 </head>
7 <body>
8   <!--Thymeleaf前端框架以Html为载体-->
9   <span>恭喜您，SpringBoot集成Thymeleaf成功</span>
10  <span>恭喜您，SpringBoot集成Thymeleaf成功</span>
11  <p>恭喜您，SpringBoot集成Thymeleaf成功</p>
12  <div>恭喜您，SpringBoot集成Thymeleaf成功</div>
13
14 </body>
15 </html>
```

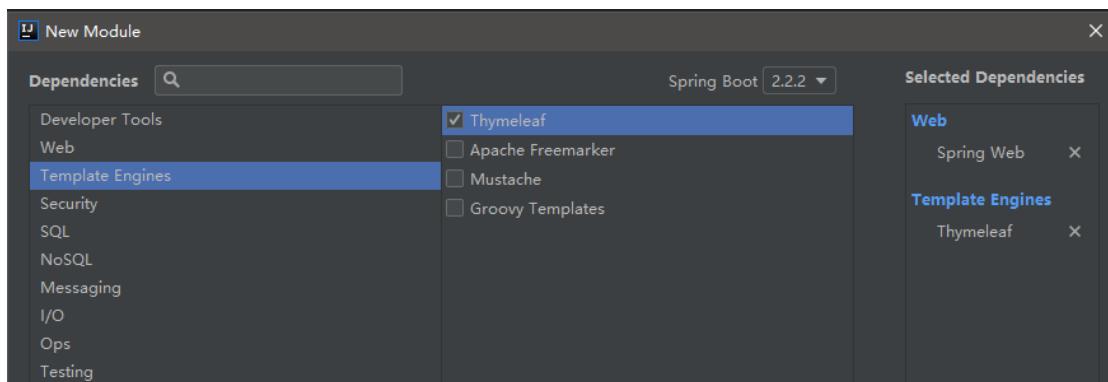
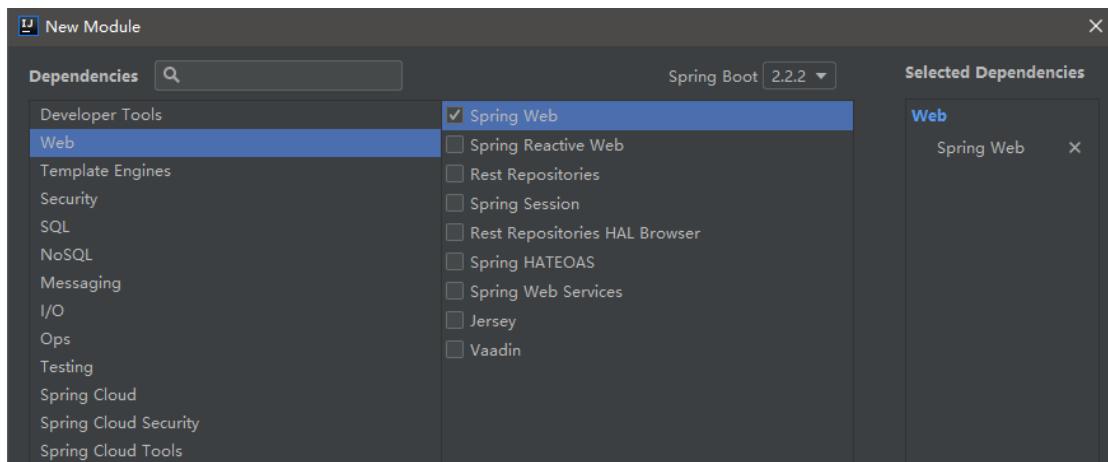
注意：Springboot 使用 thymeleaf 作为视图展示，约定将模板文件放置在 src/main/resource/templates 目录下，静态资源放置在 src/main/resource/static 目录下

11.3 Thymeleaf 的表达式

11.3.6 项目名称：038-springboot-thymeleaf-expression

(1) 创建 SpringBoot 的 web 项目并使用模版引擎





(2) pom.xml 中应该有如下两个依赖

```
<!--SpringBoot 集成 Thymeleaf 模版引擎的起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<!--SpringBoot 的 web 项目起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

(3) 在 application.properties 中设置 thymeleaf 参数

```
#设置 thymeleaf 页面缓存失效
spring.thymeleaf.cache=false

#thymeleaf 模版前缀, 默认值, 可选项
spring.thymeleaf.prefix=classpath:/templates/
#thymeleaf 模版后缀, 默认值, 可选项
spring.thymeleaf.suffix=.html
```

(4) 创建实体 User 实体类

创建 User 实体类, 为后续演示提供数据

```
@Data
public class User {

    private Integer id;

    private String name;

    private String phone;

    private String address;
}
```

(5) 创建 ThymeleafController 类

```
@Controller
public class ThymeleafController {

    @RequestMapping(value = "/thymeleaf/index")
    public String index(Model model) {

        model.addAttribute("data", "SpringBoot 集成 Thymeleaf 模版!");

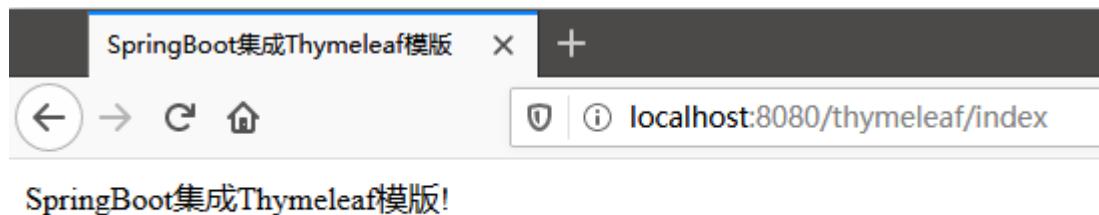
        return "index";
    }
}
```

(6) 在 src/main/resources/templates 在创建 html 页面

```
<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Thymeleaf'Index</title>
</head>
<body>
<span th:text="${data}"></span>
</body>
</html>
```

(7) 测试



11.3.7 标准变量表达式

注意： `th:text=""` 是 **Thymeleaf** 的一个属性，用于文本的显示

(8) 语法 \${...}

(9) 说明

标准变量表达式用于访问容器（tomcat）上下文环境中的变量，功能和 EL 中的 `$()` 相同。Thymeleaf 中的变量表达式使用 `${变量名}` 的方式获取 Controller 中 model 其中的数据

(10) 案例演示

创建一个方法，将用户信息存放到 model 中，thymeleaf 模版页面获取对象信息

- 1) 在 ThymeleafController 中添加方法，向 model 放入 User 对象

```
@RequestMapping(value = "/thymeleaf/user")
public String user(Model model) {

    User user = new User();
    user.setId(1);
    user.setName("张三");
    user.setPhone("13700000000");
    user.setAddress("北京市亦庄经济开发区");

    model.addAttribute("user", user);

    return "user";
}
```

- 2) 在 templates 目录下创建 user.html 页面获取 User 对象数据

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>展示用户对象信息</title>
</head>
<body>
    <h2>展示 User 用户信息: </h2>
    用户编号: <span th:text="${user.id}"></span><br/>
    用户姓名: <span th:text="${user.name}"></span><br/>
    用户手机号: <span th:text="${user.phone}"></span><br/>
    用户地址: <span th:text="${user.address}"></span><br/>
</body>
</html>
```

- 3) 浏览器访问 <http://localhost:8080/thymeleaf/user> 测试



The screenshot shows a web browser window with the title "展示用户对象信息". The address bar indicates the URL is "localhost:8080/thymeleaf/user". The main content area displays the following text:

展示User用户信息：

用户编号：1
用户名：张三
用户手机号：13700000000
用户地址：北京市亦庄经济开发区

11.3.8 选择变量表达式（了解，不推荐使用）

(11) 语法： *{...}

(12) 说明

选择变量表达式，也叫星号变量表达式，使用 `th:object` 属性来绑定对象

选择表达式首先使用 `th:object` 来绑定后台传来的 User 对象，然后使用 * 来代表这个对象，后面 {} 中的值是此对象中的属性。

选择变量表达式 *{...} 是另一种类似于标准变量表达式 \${...} 表示变量的方法

选择变量表达式在执行时是在选择的对象上求解，而 \${...} 是在上下文的变量 Model 上求解，这种写法比标准变量表达式繁琐，只需要大家了解即可

(13) 案例演示

1) 在 user.html 通过选择变量表达式（星号表达式）获取用户数据

```
<h2>展示 User 用户信息（星号表达式，仅在 div 范围内有效）：</h2>
<div th:object="${user}">
    用户编号：<span th:text="*{id}"></span><br/>
    用户姓名：<span th:text="*{name}"></span><br/>
    用户手机号：<span th:text="*{phone}"></span><br/>
    用户地址：<span th:text="*{address}"></span><br/>
</div>
```

- 2) 浏览器访问 <http://localhost:8080/thymeleaf/user> 测试



The screenshot shows a browser window with the title "展示用户对象信息：文本表达式" and the URL "localhost:8080/thymeleaf/user". The page content is divided into two sections:

- 展示User用户信息(文本表达式):**

用户编号: 1
用户名: 李四
用户手机号: 13700000000
用户地址: 北京市亦庄经济开发区
- 展示User用户信息 (星号表达式, 仅在div范围内有效) :**

用户编号: 1
用户名: 李四
用户手机号: 13700000000
用户地址: 北京市亦庄经济开发区

11.3.9 标准变量表达式和选择变量表达式混合使用

- 1) 标准变量和选择变量表达式可以混合使用，也可以不混合使用，使用 `th:object` 进行对象的选择，也可以直接使用 `*{...}` 获取数据

在 `user.html` 模版中添加如下代码：

```
<h2>标准变量表达式和选择变量表达式混用</h2>
<h3>=====标准变量表达式=====</h3>
用户编号: <span th:text="${user.id}"></span><br/>
用户名: <span th:text="${user.name}"></span><br/>
用户手机号: <span th:text="${user.phone}"></span><br/>
用户地址: <span th:text="${user.address}"></span><br/>

<h3>=====选择变量表达式=====</h3>
用户编号: *{user.id} ==> <span th:text="*{user.id}"></span><br/>
用户名: * {user.name} ==> <span th:text="*{user.name}"></span><br/>
用户手机号: *{user.phone} ==> <span th:text="*{user.phone}"></span><br/>
用户地址: *{user.address} ==> <span
th:text="*{user.address}"></span><br/>
```

- 2) 测试查看结果

标准变量表达式和选择变量表达式混用

===== 标准变量表达式 =====

用户编号：1

用户姓名：李四

用户手机号：13700000000

用户地址：北京市亦庄经济开发区

===== 选择变量表达式 =====

用户编号： *{user.id} ==> 1

用户姓名： * {user.name} ==> 李四

用户手机号： *{user.phone} ==> 13700000000

用户地址： *{user.address} ==> 北京市亦庄经济开发区

11.3.10 URL 表达式

(14) 语法@{...}

(15) 说明

主要用于链接、地址的展示，可用于

<script src="...">、<link href="...">、、<form action="...">、等，可以在 URL 路径中动态获取数据

(16) 案例演示

1) 创建 url.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>URL 路径表达式 -> @{}</title>
</head>
```

```
<body>
<h1>URL 路径表达式: @{...}</h1>
<h2>绝对路径（没有参数）</h2>
<a th:href="@{http://localhost:8080/thymeleaf/info}">查看: 绝对路径</a>

<h2>绝对路径（路径中有参数）</h2>
<a th:href="@{'http://localhost:8080/thymeleaf/user/info?id=' +
${user.id}}">查看用户信息: 绝对路径(带参数)</a>

<h2 style="color: red">实际开发推荐使用: 相对路径（没有参数）</h2>
<a th:href="@{/thymeleaf/info}">查看: 相对路径</a>

<h2 style="color: red">实际开发推荐使用: 相对路径（路径中有参数）</h2>
<a th:href="@{'/thymeleaf/user/info?id=' + ${user.id}}">查看用户信息: 相
对路径（带参数）</a>
<a th:href="@{/thymeleaf/info(id=${user.id})}">推荐使用: 优雅的带参数路径
写法</a>
</body>
</html>
```

2) 为了演示加上下文的效果，在 application.properties 中配置项目上下文

```
#设置上下文根
server.servlet.context-path=/url-expression
```

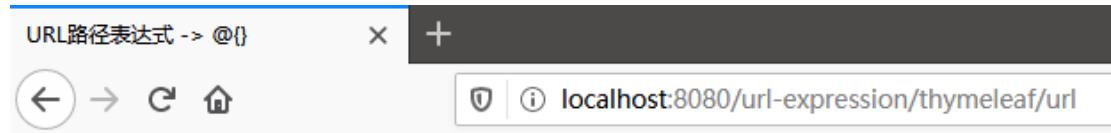
3) 在 ThymeleafController 中添加如下方法：

```
@RequestMapping(value = "/thymeleaf/url")
public String url(Model model) {
    User user = new User();
    user.setId(2);
    user.setName("赵六");
    user.setPhone("13800000000");
    user.setAddress("上海市");

    model.addAttribute("user", user);
    return "url";
}

@RequestMapping(value = "/thymeleaf/info")
public String info(Integer id) {
    System.out.println("用户编号: " + id);
    return "info";
}
```

-
- 4) 浏览器访问，右键查看源代码



URL路径表达式：@{...}

绝对路径（没有参数）

[查看：绝对路径](#)

绝对路径（路径中有参数）

[查看用户信息：绝对路径\(带参数\)](#)

实际开发推荐使用：相对路径（没有参数）

[查看：相对路径](#)

实际开发推荐使用：相对路径（路径中有参数）

[查看用户信息：相对路径（带参数）](#)

推荐使用：优雅的带参数路径写法

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>URL路径表达式 -> @{}</title>
</head>
<body>
<h1>URL路径表达式：@{...}</h1>
<h2>绝对路径（没有参数）</h2>
<a href="http://localhost:8080/url-expression/thymeleaf/info">查看：绝对路径</a>

<h2>绝对路径（路径中有参数）</h2>
<a href="http://localhost:8080/url-expression/thymeleaf/info?id=2">查看用户信息：绝对路径(带参数)</a>

<h2 style="color: red">实际开发推荐使用：相对路径（没有参数）</h2>
<a href="/url-expression/thymeleaf/info">查看：相对路径</a>

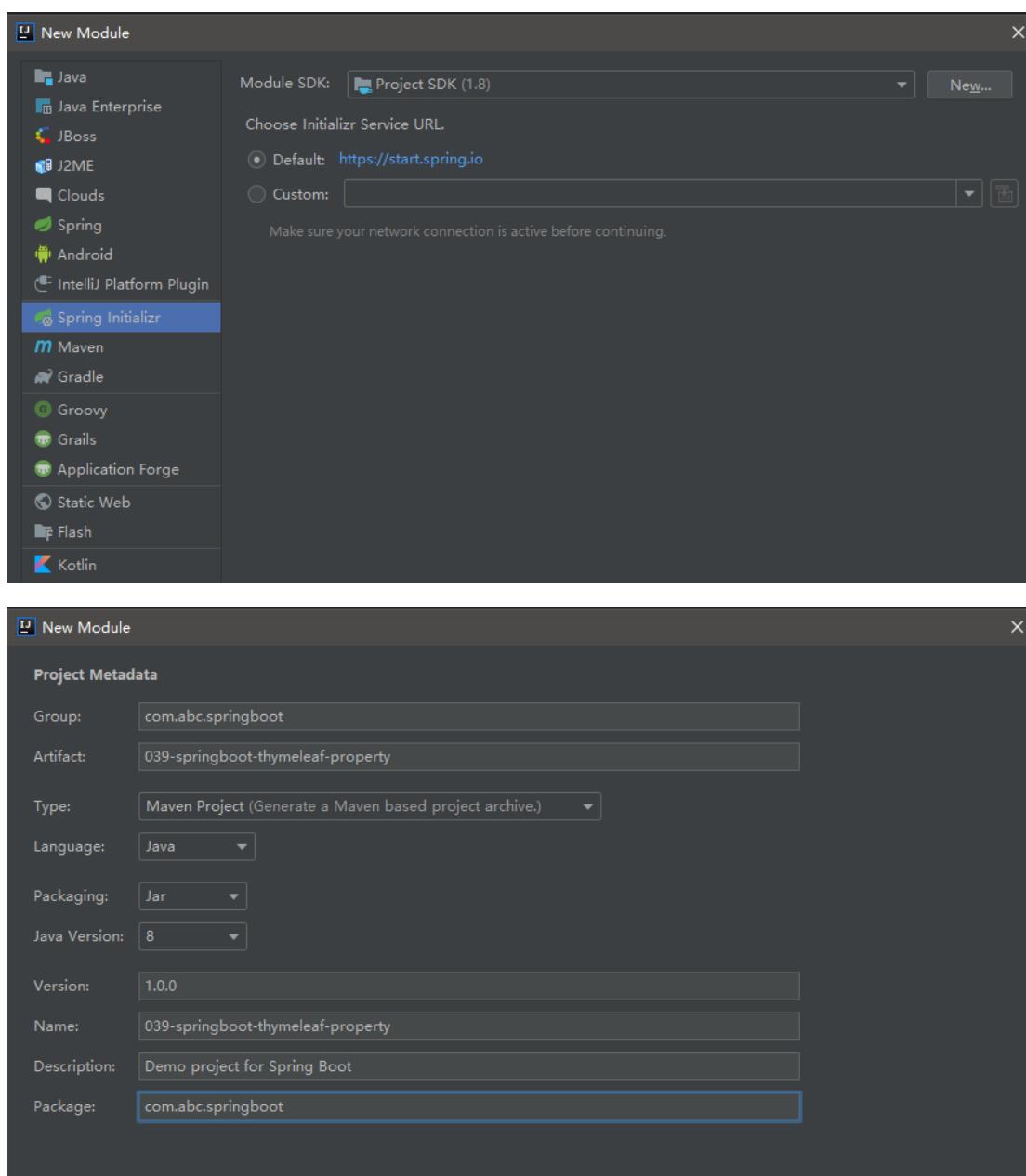
<h2 style="color: red">实际开发推荐使用：相对路径（路径中有参数）</h2>
<a href="/url-expression/thymeleaf/info?id=2">查看用户信息：相对路径（带参数）</a><br/>
<a href="/url-expression/thymeleaf/info?id=2">推荐使用：优雅的带参数路径写法</a>
</body>
</html>
```

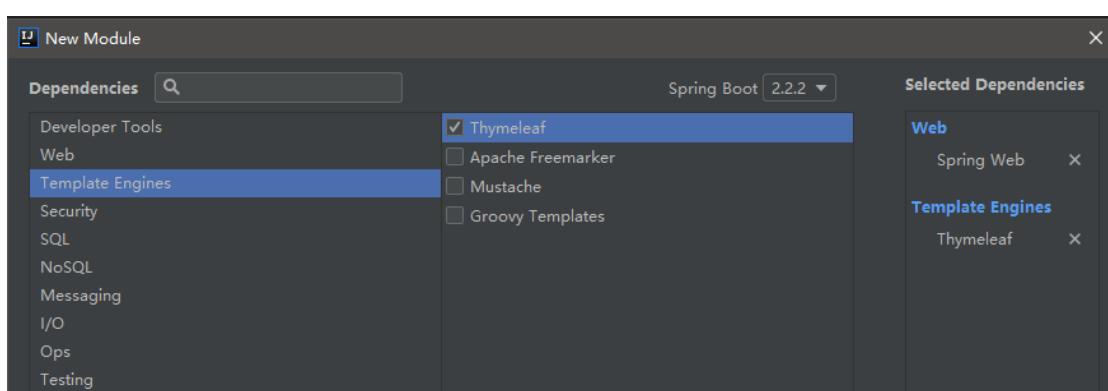
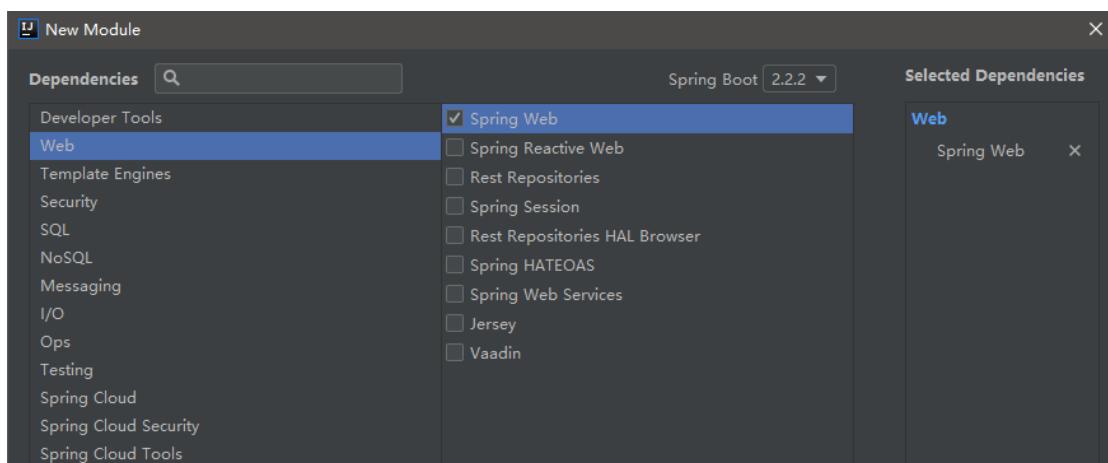
11.4 Thymeleaf 的常见属性

大部分属性和 html 的一样，只不过前面加了一个 th 前缀

11.4.1 项目名称：041-springboot-thymeleaf-property

(17) 创建 SpringBoot 的 web 项目并使用模版引擎





(18) pom.xml 中应该有如下两个依赖

```
<!--SpringBoot 框架集成 Thymeleaf 模版起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<!--SpringBoot 框架 web 项目起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

(19) 在 application.properties 中设置 thymeleaf 参数

```
#设置端口号
server.port=8090
```

```
#设置上下文根
server.servlet.context-path=/property

#thymeleaf 缓存默认开启，开发阶段关闭
spring.thymeleaf.cache=false

#设置 thymeleaf 前缀
spring.thymeleaf.prefix=classpath:/templates/
#设置 thymeleaf 后缀
spring.thymeleaf.suffix=.html
```

(20) 创建实体 User 实体类

```
@Data
public class User {
    private Integer id;

    private String name;

    private String phone;

    private String address;
}
```

(21) 创建 ThymeleafController 类

```
@Controller
public class ThymeleafController {

    @RequestMapping(value = "/index")
    public String index(Model model) {

        model.addAttribute("data","Hello, SpringBoot 集成 Thymeleaf 模版");

        return "index";
    }
}
```

(22) 在 src/main/resources/templates 在创建 Index 页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1 th:text="${data}"></h1>
</body>
</html>
```

(23) 测试



Hello, SpringBoot 集成 Thymeleaf 模版

11.4.2 th:action

th:action 定义后台控制器的路径，类似<form>标签的 action 属性，主要结合 URL 表达式，获取动态变量

(24) index.html 页面上添加以下内容

```
<h1>th:action 属性的使用</h1>
<h2>请求路径中需要动态获取变量数据时，必须添加 th 前缀</h2>
<form th:action="@{'/user/login?id=' + ${user.id}}"></form>

<h2>以下两种方式获取不到用户 id</h2>
```

```
<form action="/user/login?id=' + ${user.id}"></form>
<form action="/user/login"+${user.id}></form>
```

(25) ThymeleafController 中 index 方法添加

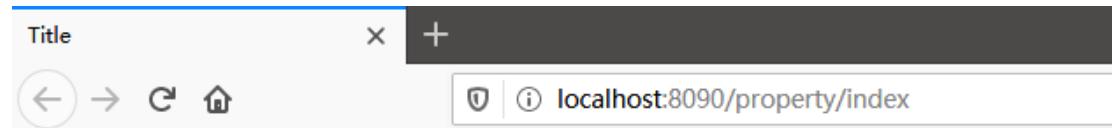
```
@RequestMapping(value = "/index")
public String index(Model model) {

    model.addAttribute("data", "Hello, SpringBoot 集成 Thymeleaf 模版");

    User user = new User();
    user.setId(10001);
    user.setName("王五");
    user.setPhone("13888880000");
    user.setAddress("天津市");
    model.addAttribute("user", user);

    return "index";
}
```

(26) 测试



Hello, SpringBoot 集成 Thymeleaf 模版

th:action 属性的使用

请求路径中需要动态获取变量数据时，必须添加 th 前缀

以下两种方式获取不到用户 id

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>Hello, SpringBoot集成Thymeleaf模版</h1><br/>

<h1>th:action属性的使用</h1>
<h2>请求路径中需要动态获取变量数据时，必须添加th前缀</h2>
<form action="/property/user/login?id=10001"></form>

<h2>以下两种方式获取不到用户id</h2>
<form action="/user/login?id=' + ${user.id}"></form>
<form action="/user/login"+${user.id}></form>
</body>
</html>
```

(27) 思考：为什么后两个中\${user.id} 获取不到数据？

因为我们 Thymeleaf 是以 html 为载体的，所以 html 不会认识\${}语法。

我们请求的流程是，发送请求给服务器，服务器接收请求后，处理请求，跳转到指定的静态 html 页面，在服务器端，Thymeleaf 模板引擎会按照它的语法，对动态数据进行处理，所以如果要是 th 开头，模板引擎能够识别，会在服务器端进行处理，获取数据；如果没有以 th 开头，那么 Thymeleaf 模板引擎不会处理，直接返回给客户端了。

11.4.3 th:method

设置请求方法

```
<form id="login" th:action="@{/login}" th:method="post">.....</form>
```

```
<h1>th:method 属性的使用</h1>
<form th:action="@{/user/login}" th:method="post"></form>
```

11.4.4 th:href

定义超链接，主要结合 URL 表达式，获取动态变量

```
<h1>th:href 使用</h1>
```

```
<a href="http://www.baidu.com">超链接百度</a><br/>
<a th:href="'http://www.baidu.com?id=' + ${user.id}">th:href 链接</a>
```

11.4.5 th:src

用于外部资源引入，比如`<script>`标签的`src`属性，``标签的`src`属性，常与`@{}`表达式结合使用，在 **SpringBoot 项目的静态资源都放到 resources 的 static 目录下。**

放到 static 路径下的内容，写路径时不需要写上 static

```
<h1>th:src 属性的使用</h1>
<!--以下方式无法引入 js-->
<script src="/static/js/jquery-1.7.2.min.js"></script>
<!--该方法是常用方法-->
<script type="text/javascript"
th:src="@{/jquery-1.7.2.min.js}"></script>
<script>
    $(function () {
        alert("引入 js 文件");
    });
</script>
```

这种方式比传统方式的好处是，在 URL 表达式前加`/`，会自动加上上下文根，避免 404 找不到资源的情况。

11.4.6 th:id

类似 html 标签中的`id`属性

```
<span th:id="${hello}">aaa</span>
```

11.4.7 th:name

设置名称

```
<input th:type="text" th:id="userName" th:name="userName">
```

11.4.8 th:value

类似 html 标签中的 value 属性，能对某元素的 value 属性进行赋值

```
<input type="hidden" id="userId" name="userId" th:value="${userId}">
```

11.4.9 th:attr

该属性也是用于给 HTML 中某元素的某属性赋值，好处是可以给 html 中没有定义的属性动态的赋值。

```
<h1>th:attr 属性的使用</h1>
<span zhangsan="${user.name}"></span>
<!--通过 th:attr 对自定义的属性赋值--&gt;
&lt;span th:attr="zhangsan=${user.name}"&gt;&lt;/span&gt;</pre>
```

```
<h1>th:attr 属性的使用</h1>
<span zhangsan="${user.name}"></span>
<!--通过 th:attr 对自定义的属性赋值--&gt;
&lt;span zhangsan="王五"&gt;&lt;/span&gt;</pre>
```

11.4.10 th:text

用于文本的显示，该属性显示的文本在标签体中，如果是文本框，数据会在文本框外显示，要想显示在文本框内，使用 th:value

```
<input type="text" id="realName" name="realName" th:text="${realName}">
```

11.4.11 th:object

用于数据对象绑定

通常用于选择变量表达式（星号表达式）

11.4.12 th:onclick

```
<h1>th:onclick 的使用</h1>
<!--目前 thymeleaf 版本要求只能传递数字和布尔值--&gt;
&lt;a th:onclick="`show(' + ${user.id} + ')`"&gt;点击：显示学生编号&lt;/a&gt;
&lt;script type="text/javascript"&gt;
    function show(id) {
        alert("用户编号为：" + id);
    }
&lt;/script&gt;</pre>
```

11.4.13 th:style

设置样式

```
<a th:onclick="`show(' + ${user.id} + ')`"
th:style="`font-size:40px;color:red;`">点击：显示学生编号</a>
```

11.4.14 *th:each

这个属性非常常用，比如从后台传来一个对象集合那么就可以使用此属性遍历输出，它与 JSTL 中的<c:forEach>类似，此属性既可以循环遍历集合，也可以循环遍历数组及 Map

(28) 遍历 List 集合

A、在 ThymeleafController 中添加 eachList 方法，准备集合数据

```
@RequestMapping("/each/list")
public String eachList(Model model) {
    List<User> userList = new ArrayList<User>();

    for (int i = 0; i < 10; i++) {
        User user = new User();
        user.setId(100 + i);
        user.setNick("张" + i);
        user.setPhone("1361234567" + i);
        user.setAddress("北京市大兴区" + i);
        userList.add(user);
    }
    model.addAttribute("userList", userList);
```

```
        return "each";
    }
```

B、 创建 eachList.html 对 List 集合进行遍历

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>循环遍历 List 集合</title>
</head>
<body>
<h1>th:each 循环遍历 List 集合</h1>
<div style="color: red">
    1.user:当前对象的变量名<br/>
    2.userStat:当前对象的状态变量名<br/>
    3.${ userList }:循环遍历的集合<br/>
    4.变量名自定义
</div>
<div th:each="user, userStat: ${ userList }">
    <span th:text="${ userStat.index }"></span>
    <span th:text="${ user.id }"></span>
    <span th:text="${ user.name }"></span>
    <span th:text="${ user.phone }"></span>
    <span th:text="${ user.address }"></span>
</div>
</body>
</html>
```

代码说明

th:each="user, iterStat : \${userlist}" 中的 **\${userList}** 是后台传过来的集合

■ **user**

定义变量，去接收遍历\${userList}集合中的一个数据

■ **iterStat**

\${userList} 循环体的信息

- 其中 user 及 iterStat 自己可以随便取名
- iterStat 是循环体的信息，通过该变量可以获取如下信息

index: 当前迭代对象的 index (从 0 开始计算)

count: 当前迭代对象的个数（从 1 开始计算）这两个用的较多

size: 被迭代对象的大小

current: 当前迭代变量

even/odd: 布尔值，当前循环是否是偶数/奇数（从 0 开始计算）

first: 布尔值，当前循环是否是第一个

last: 布尔值，当前循环是否是最后一个

注意：循环体信息 interStat 也可以不定义，则默认采用迭代变量加上 Stat 后缀，即 userStat

C、 浏览器访问测试



th:each循环遍历List集合

1.user:当前对象的变量名

2.userStat:当前对象的状态变量名

3.\${ userList }:循环遍历的集合

4.变量名自定义

0 0	zhangsan0	13300000000	北京市0
1 1	zhangsan1	13300000001	北京市1
2 2	zhangsan2	13300000002	北京市2
3 3	zhangsan3	13300000003	北京市3
4 4	zhangsan4	13300000004	北京市4
5 5	zhangsan5	13300000005	北京市5
6 6	zhangsan6	13300000006	北京市6
7 7	zhangsan7	13300000007	北京市7
8 8	zhangsan8	13300000008	北京市8
9 9	zhangsan9	13300000009	北京市9

(29) 遍历 Map 集合

D、在 ThymeleafController 中添加 eachMap 方法

```
@RequestMapping(value = "/each/map")
public String eachMap(Model model) {

    Map<Integer, Object> userMaps = new HashMap<Integer, Object>();

    for (int i = 0; i < 10; i++) {
        User user = new User();
        user.setId(i);
        user.setName("李四"+i);
        user.setPhone("1390000000"+i);
        user.setAddress("天津市"+i);

        userMaps.put(i, user);
    }

    model.addAttribute("userMaps", userMaps);

    return "eachMap";
}
```

E、添加 eachMap.html 页面对 Map 集合进行遍历

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>循环遍历 Map 集合</title>
</head>
<body>
    <h1>th:each 循环遍历 Map 集合</h1>
    <div th:each="userMap, userMapStat : ${userMaps}">
        <span th:text="${userMapStat.count}"></span>
        <span th:text="${userMap.key}"></span>
        <span th:text="${userMap.value}"></span>
    </div>
</body>

```

```
<span th:text="${userMap.value.id}"></span>
<span th:text="${userMap.value.name}"></span>
<span th:text="${userMap.value.phone}"></span>
<span th:text="${userMap.value.address}"></span>
</div>
</body>
</html>
```

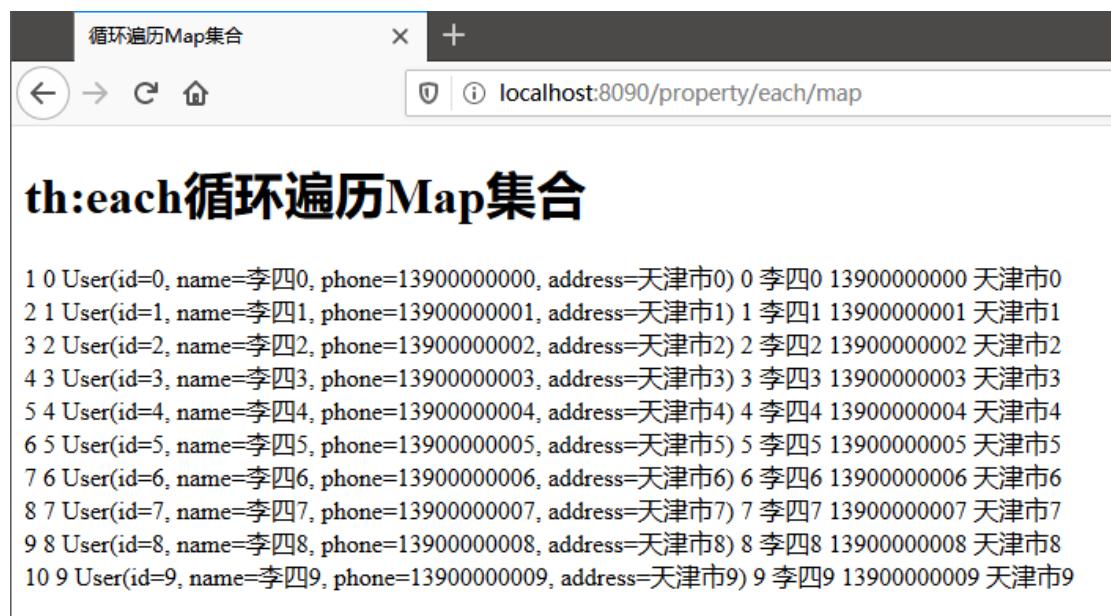
代码说明

th:each="userMap,userMapStat:\${userMaps}"，用 userMap 变量接收每次遍历的结果，封装为一个键值对，userMapStat 状态

userMap.key：获取当前键值对中的 key

userMap.value：获取当前键值对中的 value

F、浏览器访问测试



(30) 遍历 Array 数组

G、在 ThymeleafController 中的 eachArray 方法中准备数组数据

```
@RequestMapping(value = "/each/array")
public String eachArray(Model model) {
    User[] userArray = new User[10];

    for (int i = 0; i < 10; i++) {
        User user = new User();
        user.setId(i);
        user.setName("赵六"+i);
        user.setPhone("1380000000"+i);
        user.setAddress("深圳市"+i);
        userArray[i] = user;
    }

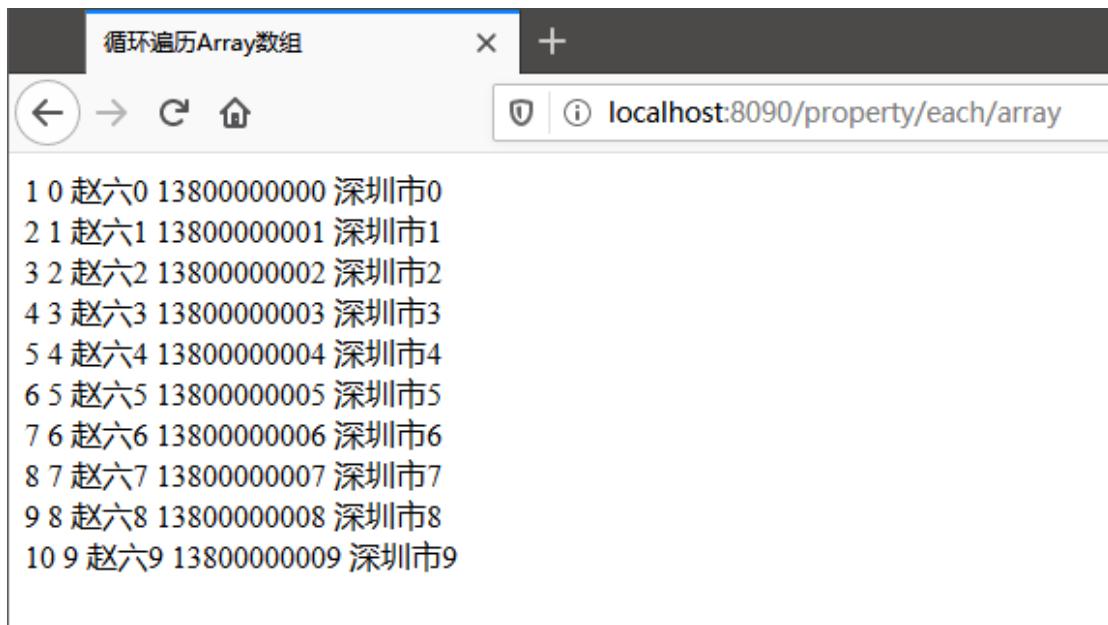
    model.addAttribute("userArray", userArray);

    return "eachArray";
}
```

H、 在 **eachArray.html** 页面对数组进行遍历（和 **List** 一样）

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>循环遍历 Array 数组</title>
</head>
<body>
<div th:each="user, userStat:${userArray}">
    <span th:text="${userStat.count}"></span>
    <span th:text="${user.id}"></span>
    <span th:text="${user.name}"></span>
    <span th:text="${user.phone}"></span>
    <span th:text="${user.address}"></span>
</div>
</body>
</html>
```

I、浏览器访问测试

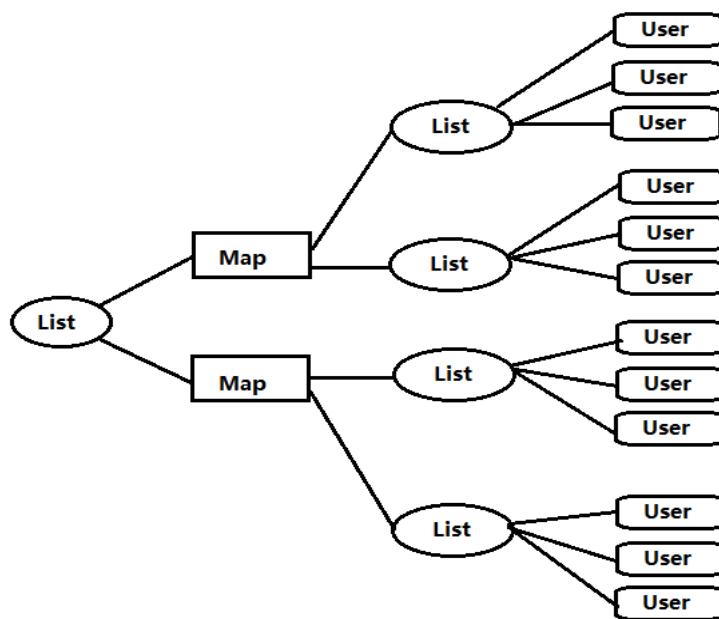


The screenshot shows a browser window with the title "循环遍历Array数组". The address bar displays "localhost:8090/property/each/array". The page content lists 10 entries, each consisting of a number from 1 to 10 followed by a name and a phone number, all ending with "深圳市":

```
1 0 赵六0 13800000000 深圳市0
2 1 赵六1 13800000001 深圳市1
3 2 赵六2 13800000002 深圳市2
4 3 赵六3 13800000003 深圳市3
5 4 赵六4 13800000004 深圳市4
6 5 赵六5 13800000005 深圳市5
7 6 赵六6 13800000006 深圳市6
8 7 赵六7 13800000007 深圳市7
9 8 赵六8 13800000008 深圳市8
10 9 赵六9 13800000009 深圳市9
```

(31) 比较复杂的循环案例

需求：List 里面放 Map，Map 里面又放的是 List



J、在 ThymeleafController 的 each 方法中构造数据

```
@RequestMapping(value = "/each/all")
public String eachAll(Model model) {

    //list -> Map -> List -> User
    List<Map<Integer, List<User>>> myList = new ArrayList<Map<Integer,
List<User>>>();

    for (int i = 0; i < 2; i++) {

        Map<Integer, List<User>> myMap = new HashMap<Integer,
List<User>>();

        for (int j = 0; j < 2; j++) {

            List<User> myUserList = new ArrayList<User>();

            for (int k = 0; k < 3; k++) {

                User user = new User();
                user.setId(k);
                user.setName("张三"+k);
                user.setPhone("1350000000"+k);
                user.setAddress("广州市"+i);

                myUserList.add(user);

            }
            myMap.put(j, myUserList);

        }
        myList.add(myMap);

    }

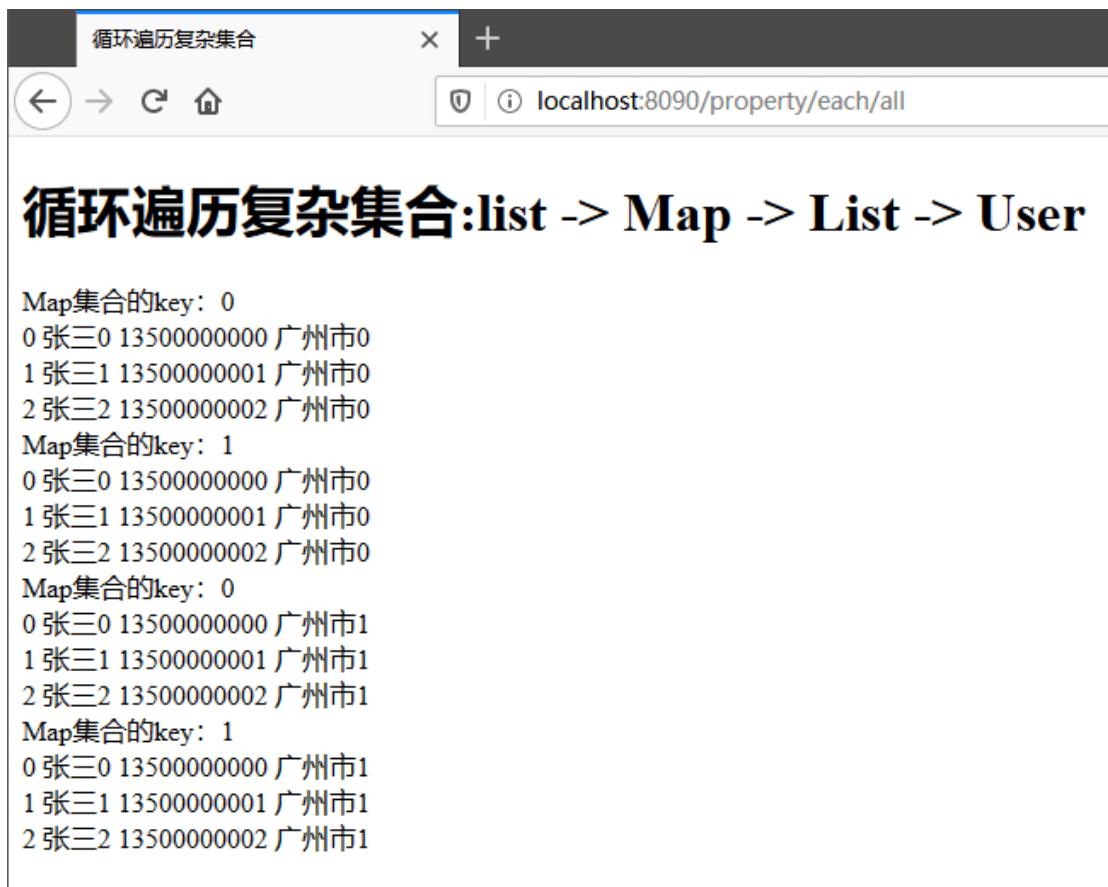
    model.addAttribute("myList", myList);

    return "eachAll";
}
```

K、 在 eachAll.html 页面对复杂集合关系进行遍历

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>循环遍历复杂集合</title>
</head>
<body>
<h1>循环遍历复杂集合:list -> Map -> List -> User</h1>
<div th:each="myListMap:${myList}">
    <div th:each="myListMapObj:${myListMap}">
        Map 集合的 key: <span th:text="${myListMapObj.key}"></span>
        <div th:each="myListMapObjList:${myListMapObj.value}">
            <span th:text="${myListMapObjList.id}"></span>
            <span th:text="${myListMapObjList.name}"></span>
            <span th:text="${myListMapObjList.phone}"></span>
            <span th:text="${myListMapObjList.address}"></span>
        </div>
    </div>
</div>
</body>
</html>
```

L、浏览器访问测试



循环遍历复杂集合

localhost:8090/property/each/all

循环遍历复杂集合:list -> Map -> List -> User

```
Map集合的key: 0
0 张三0 13500000000 广州市0
1 张三1 13500000001 广州市0
2 张三2 13500000002 广州市0
Map集合的key: 1
0 张三0 13500000000 广州市0
1 张三1 13500000001 广州市0
2 张三2 13500000002 广州市0
Map集合的key: 0
0 张三0 13500000000 广州市1
1 张三1 13500000001 广州市1
2 张三2 13500000002 广州市1
Map集合的key: 1
0 张三0 13500000000 广州市1
1 张三1 13500000001 广州市1
2 张三2 13500000002 广州市1
```

11.4.15 条件判断

(32) **th:if**

(33) **th:unless**

```
@RequestMapping(value = "/condition")
public String condition(HttpServletRequest request, Model model) {

    User user1 = null;
    model.addAttribute("user1", user1);

    User user2 = new User();
    user2.setId(1001);
```

```
user2.setName("小岳岳");
user2.setPhone("13900000000");
user2.setAddress("北京市");
model.addAttribute("user2", user2);

model.addAttribute("sex", 1);

User user3 = new User();
user3.setId(1002);
user3.setName("孙悦");
user3.setPhone("13200000000");
user3.setAddress("北京市");
model.addAttribute("user3", user3);
request.getSession().setAttribute("user3", user3);

return "condition";
}
```

condition.html 页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>条件判断</title>
</head>
<body>
<h1>th:if 用法:如果满足条件显示, 否则相反</h1>
<div th:if="${sex eq 1}">
    男: <input type="radio" name="sex" th:value="1"/>
</div>
<div th:if="${sex eq 0}">
    女: <input type="radio" name="sex" th:value="0"/>
</div>

<h1>th:unless 用法: 与 th:if 用法相反, 即对条件判断条件取反</h1>
<div th:unless="${sex == 1}">
    男: <input type="radio" name="sex" th:value="1"/>
</div>
<div th:unless="${sex eq 0}">
    女: <input type="radio" name="sex" th:value="0"/>
</div>
```

```
<div th:if="${user1 eq null}">
    <h3 style="color: red">用户未登录</h3>
</div>

<div th:unless="${user2 == null}">
    用户姓名: <span th:text="${user2.name}"></span>
</div>

<h1>从 session 中获取值</h1>
<div th:if="${user3 != null}">
    <span th:text="${user3.name}"></span>
</div>
</body>
</html>
```

(34) th:switch/th:case

switch, case 判断语句

```
<h1>th:switch/th:case 用法</h1>
<div th:switch="${sex}">
    <span th:case="1">性别: 男</span><br/>
    <span th:case="2">性别: 女</span><br/>
    <span th:case="*>性别: 保密</span>
</div>
```

一旦某个 case 判断值为 true, 剩余的 case 默认不执行, “*” 表示默认的 case, 前面的 case 都不匹配时候, 执行默认的 case

(35) 浏览器访问测试

th:switch/th:case 用法

性别: 男

11.4.16 th:inline

th:inline 有三个取值类型 (text, javascript 和 none), 值为 none 什么都不做, 没有效果

(1) 内敛文本 (th:inline="text")

内敛文本表达式不依赖于 html 标签, 直接使用内敛表达式[[表达式]]即可获取动态数据, 但必须要求在父级标签上加 th:inline = "text" 属性

A、在 ThymeleafController 类中添加方法

```
@RequestMapping(value = "/inline")
public String inlineText(Model model) {

    User user = new User();
    user.setId(1003);
    user.setName("杰克");
    user.setPhone("13899990000");
    user.setAddress("天津市");
    model.addAttribute("user", user);

    return "inline";
}
```

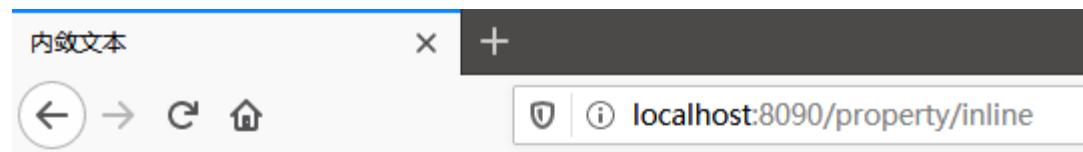
B、创建 inline.html 页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>内敛文本</title>
</head>
<body>
    <h1>标准变量表达式(展示数据)</h1>
    用户编号: <span th:text="${user.id}"></span><br/>
```

```
用户姓名: <span th:text="${user.name}"></span><br/>
用户手机号: <span th:text="${user.phone}"></span><br/>
用户地址: <span th:text="${user.address}"></span><br/>

<h1>内嵌文本 th:inline="text"</h1>
<div th:inline="text">
    用户编号: <div>[$user.id]</div><br/>
    用户姓名: [$user.name]<br/>
    用户手机号: [$user.phone]<br/>
    用户地址: [$user.address]<br/>
</div>
</body>
</html>
```

C、 浏览器访问测试



标准变量表达式(展示数据)

用户编号: 1003
用户姓名: 杰克
用户手机号: 13899990000
用户地址: 天津市

内嵌文本th:inline="text"

用户编号:
1003

用户姓名: 杰克
用户手机号: 13899990000
用户地址: 天津市

注意: 一般我们将 th:inline="text" 放到<body th:inline="text">标签中

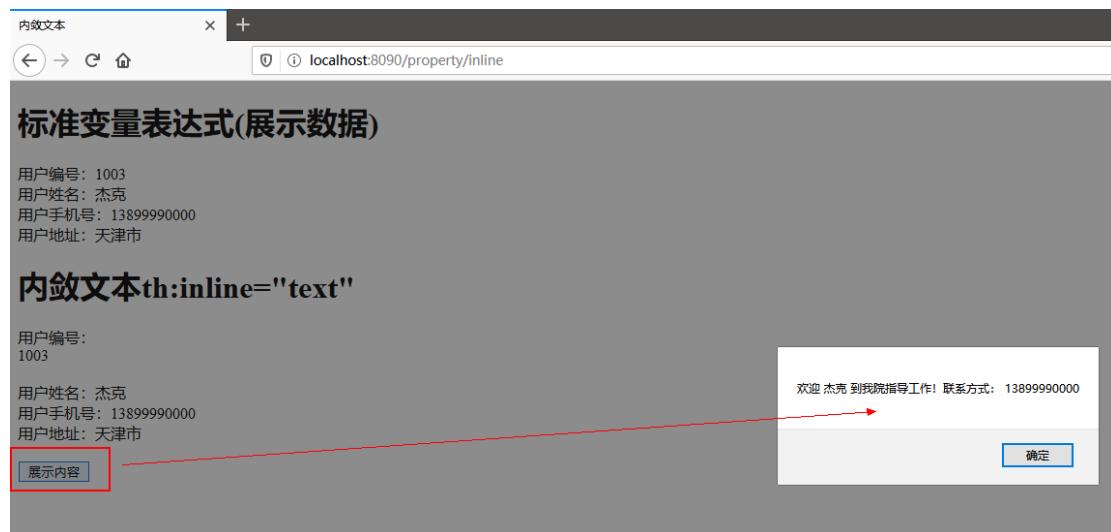
(2) 内嵌脚本 (`th:inline="javascript"`)

`th:inline="javascript"` 在 js 代码中获取后台的动态数据

D、在 `inline.html` 页面上，添加如下代码

```
<script type="text/javascript" th:inline="javascript">
    function showInlineJavaScipt() {
        alert("欢迎 " + [[${user.name}]] + " 到我院指导工作！联系方式：" +
[[${user.phone}]]);
    }
</script>
<button th:onclick="showInlineJavaScipt()">展示内容</button>
```

E、浏览器访问测试



11.5 Thymeleaf 字面量

字面量：对应数据类型的合法取值，可以在 `html` 页面直接使用，不需要后台传递

1. 在 `ThymeleafController` 类中添加方法并准备数据

```
@RequestMapping(value = "/literal")
public String literal(Model model) {
```

```
model.addAttribute("success",true);
model.addAttribute("flag",false);

User user = new User();
user.setId(1004);
user.setName("贾玲");
user.setPhone("13777777777");
user.setAddress("北京市");
model.addAttribute("user",user);

}

return "literal";
}
```

2. 创建 literal.html 页面

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

</body>
</html>
```

11.5.1 文本字面量

用单引号'...'包围的字符串为文本字面量

```
<h1>文本字面量：用单引号'....'包围的字符串</h1>
<a th:href="@{'/user/info?id=' + ${user.id}}>查看用户：文本字面的路径使用</a><br/>
<span th:text="您好"></span>
```

11.5.2 数字字面量

```
<h1>数字字面量</h1>
今年是<span th:text="2019">1949</span>年<br/>
20年后，将是<span th:text="2019 + 20">1969</span>年<br/>
```

11.5.3 boolean 字面量

```
<h1>boolean 字面量</h1>
<div th:if="${success}">执行成功</div>
<div th:unless="${flag}">执行不成功</div>
```

11.5.4 null 字面量

```
<h1>null 字面量</h1>
<span th:if="${user ne null}">用户不为空</span><br/>
<span th:unless="${user eq null}">用户不为空（使用 th:unless 取反）</span><br/>
```

11.6 Thymeleaf 字符串拼接

```
<h1>文本字面量使用 "+" 拼接字符串</h1>
<span th:text=""共${totalRows}条${totalPage}页, 当前第${currentPage}+页"></span>

<h1>另一种更优雅的方式: 使用 " | " 要拼接的内容 | " 减少字符串拼接的加号</h1>
<span th:text="|共${totalRows}条${totalPage}页, 当前第${currentPage}页|"></span>
```

11.7 Thymeleaf 运算符

三元运算: 表达式?"正确结果":"错误结果"

算术运算: + , - , * , / , %

关系比较:: > , < , >= , <= (gt , lt , ge , le)

相等判断: == , != (eq , ne)

```
@RequestMapping(value = "/operator")
public String operator(Model model) {

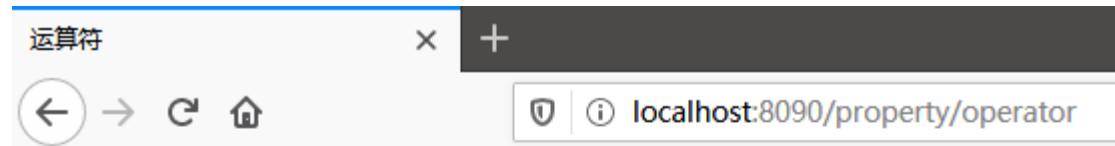
    model.addAttribute("sex", 1);

    return "operator";
}
```

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>运算符</title>
</head>
<body>
<h1>三元运算符</h1>
<span th:text="${sex eq 1 ? '男':'女'}"></span><br/>
<span th:text="${sex == 1 ? '男':'女'}"></span><br/>

20*8=<span th:text="20 * 8"></span><br/>
20/8=<span th:text="20 / 8"></span><br/>
20+8=<span th:text="20 + 8"></span><br/>
20-8=<span th:text="20 - 8"></span><br/>

<div th:if="5 > 2">5>2 是真的</div>
<div th:if="5 gt 2">5 gt 2 是真的</div>
</body>
</html>
```



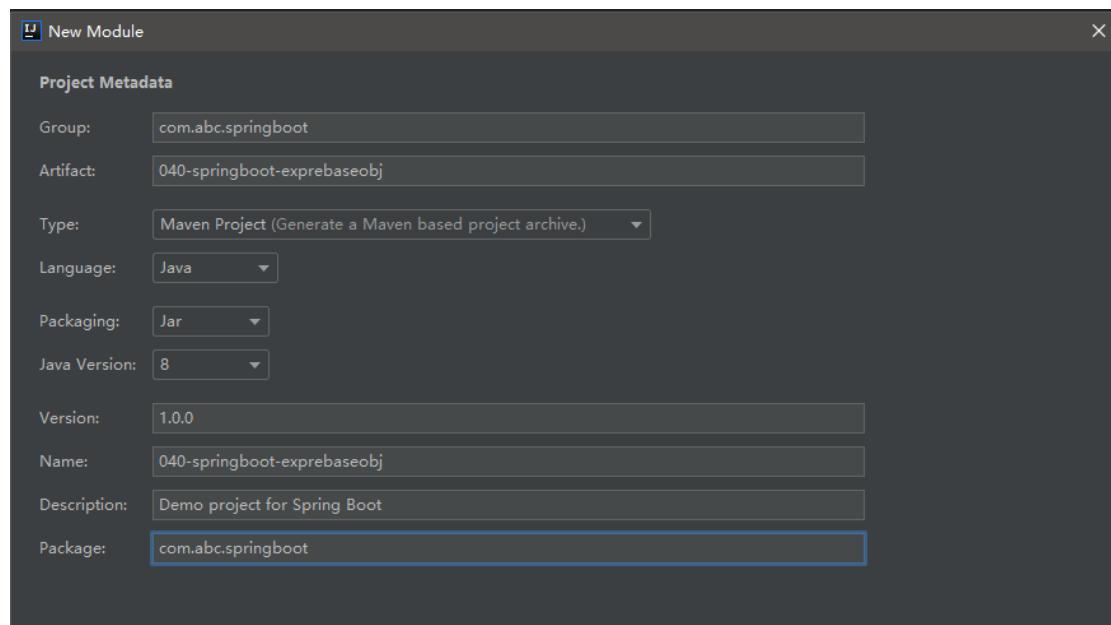
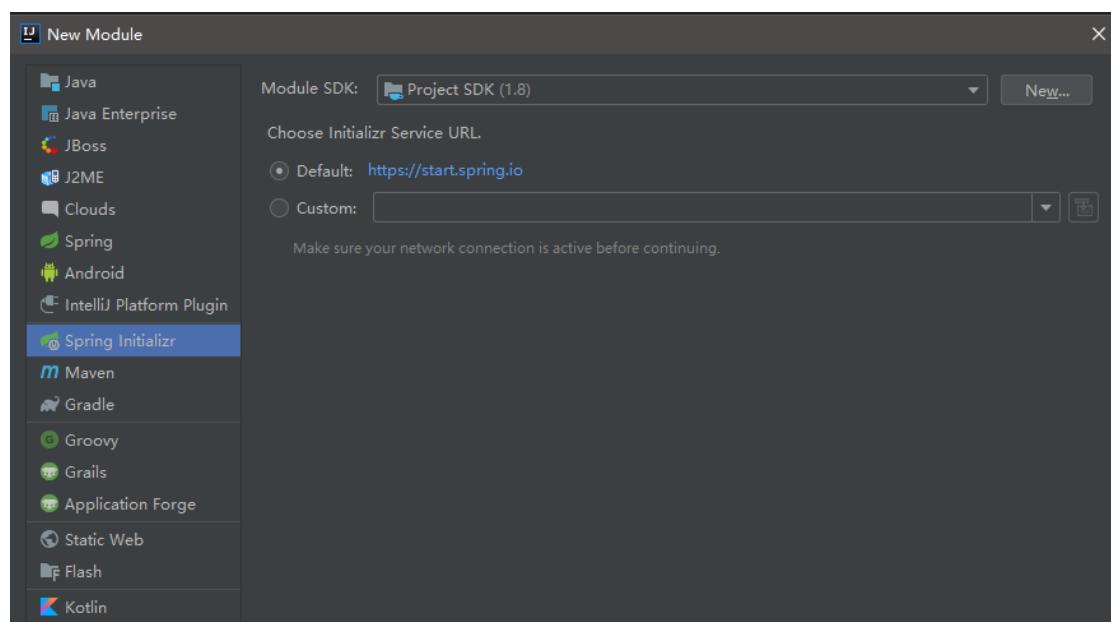
三元运算符

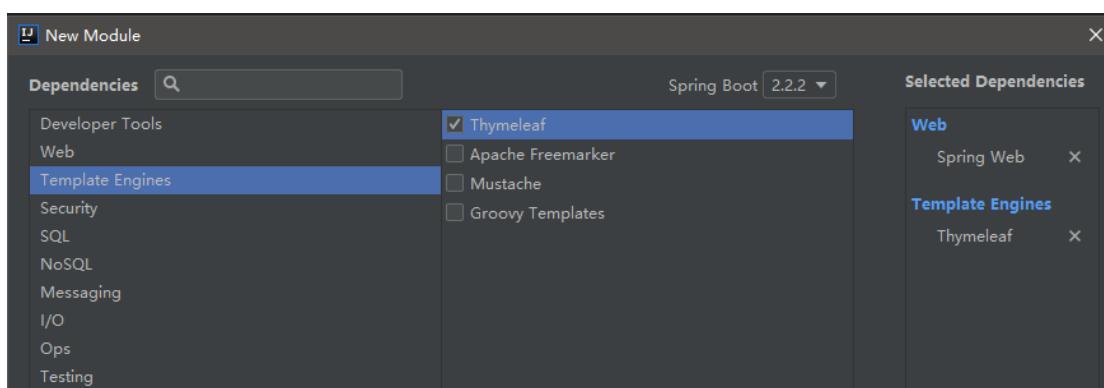
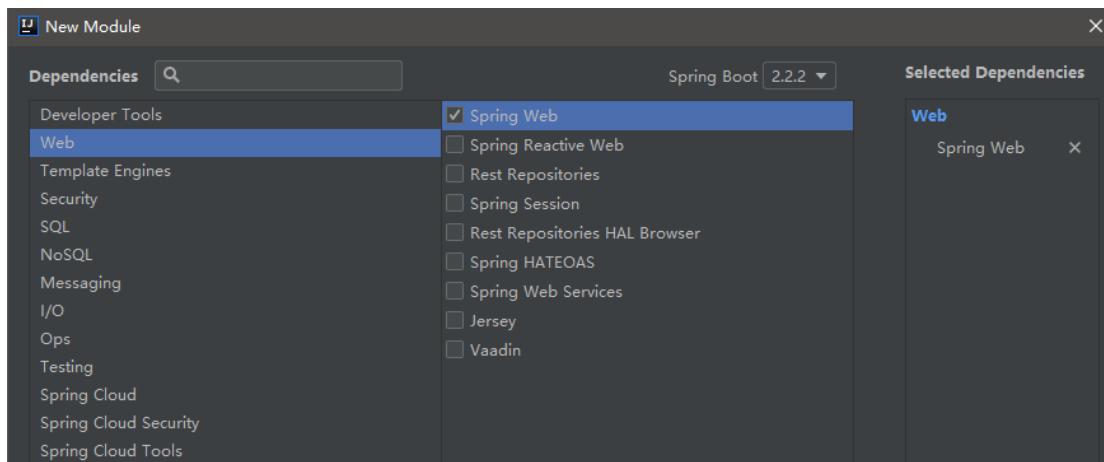
男
男
 $20*8=160$
 $20/8=2.5$
 $20+8=28$
 $20-8=12$
5>2是真的
5 gt 2是真的

11.8 Thymeleaf 表达式基本对象

模板引擎提供了一组内置的对象，这些内置的对象可以直接在模板中使用，这些对象由#号开始引用，我们比较常用的内置对象

11.8.1 创建 SpringBoot 项目并集成 thymeleaf 框架





11.8.2 #request

#request 相当于 `httpServletRequest` 对象，这是 3.x 版本，若是 2.x 版本使用 `#httpServletRequest`，在页面获取应用的上下文根，一般在 js 中请求路径中加上可以避免 404

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Thymeleaf 表达式基本对象</title>
</head>
<body>
<script type="text/javascript" th:inline="javascript">
    var basePath = ['${'#httpServletRequest.getScheme()' + "://" +
    '#httpServletRequest.getServerName()' + ":" +
    '#httpServletRequest.getServerPort()' +
    '#httpServletRequest.getContextPath()' }];
    //http://localhost:8080/springboot/user/login
</script>
</body>
</html>
```

```
//获取协议名称
var scheme = [ ${#request.getScheme()} ] ;
//获取服务 IP 地址
var serverName = [ ${#request.getServerName()} ] ;
//获取服务端口号
var serverPort = [ ${#request.getServerPort()} ] ;
//获取上下文根
var contextPath = [ ${#request.getContextPath()} ] ;

var allPath = scheme+"://"+serverName+":"+serverPort+contextPath;

alert(allPath);
</script>
</body>
</html>
```

11.8.3 #session

相当于 HttpSession 对象，这是 3.x 版本，若是 2.x 版本使用#httpSession

在后台方法中向 session 中放数据

```
@RequestMapping(value = "/index")
public String index(HttpServletRequest request) {

    request.getSession().setAttribute("username", "zhangsan");

    return "index";
}
```

从页面获取数据

```
<h1>从 SESSION 中获取用户名</h1>
<span th:text="${#session.getAttribute('username')} "></span><br/>
<span th:text="${#httpSession.getAttribute('username')} "></span>
```

11.9 Thymeleaf 表达式功能对象（了解）

模板引擎提供的一组功能性内置对象，可以在模板中直接使用这些对象提供的功能方法

工作中常使用的数据类型，如集合，时间，数值，可以使用 Thymeleaf 的提供的功能性对象来处理它们

内置功能对象前都需要加#号，内置对象一般都以 s 结尾

官方手册：<http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

#dates: java.util.Date 对象的实用方法：

```
<span th:text="#{#dates.format(curDate, 'yyyy-MM-dd HH:mm:ss')}"></span>
```

#calendars: 和 dates 类似，但是 java.util.Calendar 对象；

#numbers: 格式化数字对象的实用方法；

#strings: 字符串对象的实用方法： contains, startsWith, prepending/appending 等；

#objects: 对 objects 操作的实用方法；

#bools: 对布尔值求值的实用方法；

#arrays: 数组的实用方法；

#lists: list 的实用方法，比如

#sets: set 的实用方法；

#maps: map 的实用方法；

#aggregates: 对数组或集合创建聚合的实用方法；

第12章 总结及综合案例

12.1 总结

采用 Spring Boot 开发实质上也是一个常规的 Spring 项目开发，只是利用了 SpringBoot 启动程序和自动配置简化开发过程，提高开发效率。

SpringBoot 项目开发代码的实现依然是使用 SpringMVC+ Spring + MyBatis 等，当然能集成几乎所有的开源项目， SpringBoot 是极速 web 开发框架。

采用 SpringBoot 开发，需要掌握大量的注解，所以日常开发中注意对注解的积累。

12.2 综合案例

通过上面内容的学习，我们完成一个综合案例：

采用 SpringBoot + Dubbo + MyBatis + Redis + Thymeleaf 实现对数据库的增删改查、缓存操作。

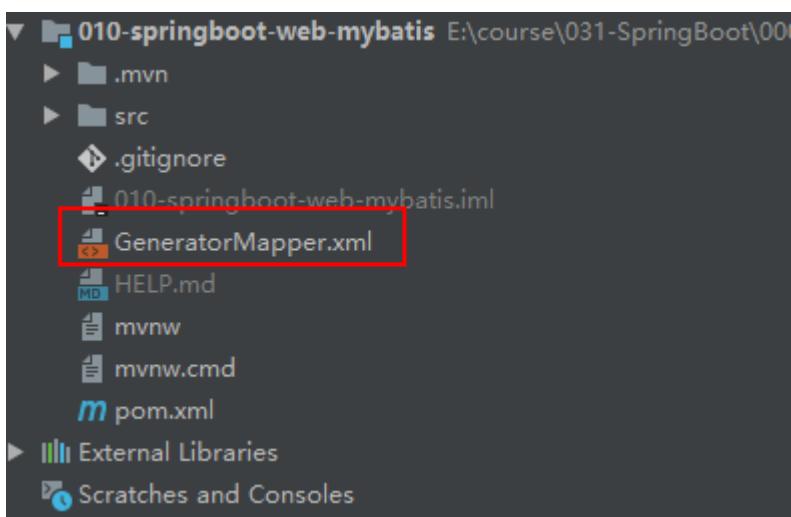
主要目的是练习 Springboot 如何集成各类技术进行项目开发

第13章 附录

13.1 SpringBoot 工程下使用 Mybatis 反向工程

13.1.1 拷贝 Mybatis 反向工程配置文件到项目的根目录下

获取目录: GeneratorMapper.xml



13.1.2 根据项目及表的情况，修改 GeneratorMapper.xml 配置

红色标注的地方是需要确认修改的地方，尤其注意

- 如果使用高版本，驱动类变为： com.mysql.cj.jdbc.Driver
- url 后面应该加属性 nullCatalogMeansCurrent=true，否则生成有问题

当前版本 MySQL 数据库为 5.7.18

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

<!DOCTYPE generatorConfiguration
    PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
    "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>

    <!-- 指定连接数据库的 JDBC 驱动包所在位置，指定到你本机的完整路径 -->
    <classPathEntry location="E:\mysql-connector-java-5.1.38.jar"/>
```

```
<!-- 配置 table 表信息内容体，targetRuntime 指定采用 MyBatis3 的版本 -->
<context id="tables" targetRuntime="MyBatis3">

    <!-- 抑制生成注释，由于生成的注释都是英文的，可以不让它生成 -->
    <commentGenerator>
        <property name="suppressAllComments" value="true" />
    </commentGenerator>

    <!-- 配置数据库连接信息 -->
    <jdbcConnection driverClass="com.mysql.jdbc.Driver"

        connectionURL="jdbc:mysql://localhost:3306/springboot"
            userId="root"
            password="123456">
    </jdbcConnection>

    <!-- 生成 model 类，targetPackage 指定 model 类的包名， targetProject 指定
生成的 model 放在 eclipse 的哪个工程下面-->
    <javaModelGenerator targetPackage="com.abc.springboot.model"
targetProject="src/main/java">
        <property name="enableSubPackages" value="false" />
        <property name="trimStrings" value="false" />
    </javaModelGenerator>

    <!-- 生成 MyBatis 的 Mapper.xml 文件，targetPackage 指定 mapper.xml 文件的
包名， targetProject 指定生成的 mapper.xml 放在 eclipse 的哪个工程下面 -->
    <sqlMapGenerator targetPackage="com.abc.springboot.mapper"
targetProject="src/main/java">
        <property name="enableSubPackages" value="false" />
    </sqlMapGenerator>

    <!-- 生成 MyBatis 的 Mapper 接口类文件，targetPackage 指定 Mapper 接口类的包名， targetProject 指定生成的 Mapper 接口放在 eclipse 的哪个工程下面 -->
    <javaClientGenerator type="XMLMAPPER"
targetPackage="com.abc.springboot.mapper" targetProject="src/main/java">
        <property name="enableSubPackages" value="false" />
    </javaClientGenerator>

    <!-- 数据库表名及对应的 Java 模型类名 -->
    <table tableName="t_student" domainObjectName="Student"
        enableCountByExample="false"
        enableUpdateByExample="false"
        enableDeleteByExample="false"
```

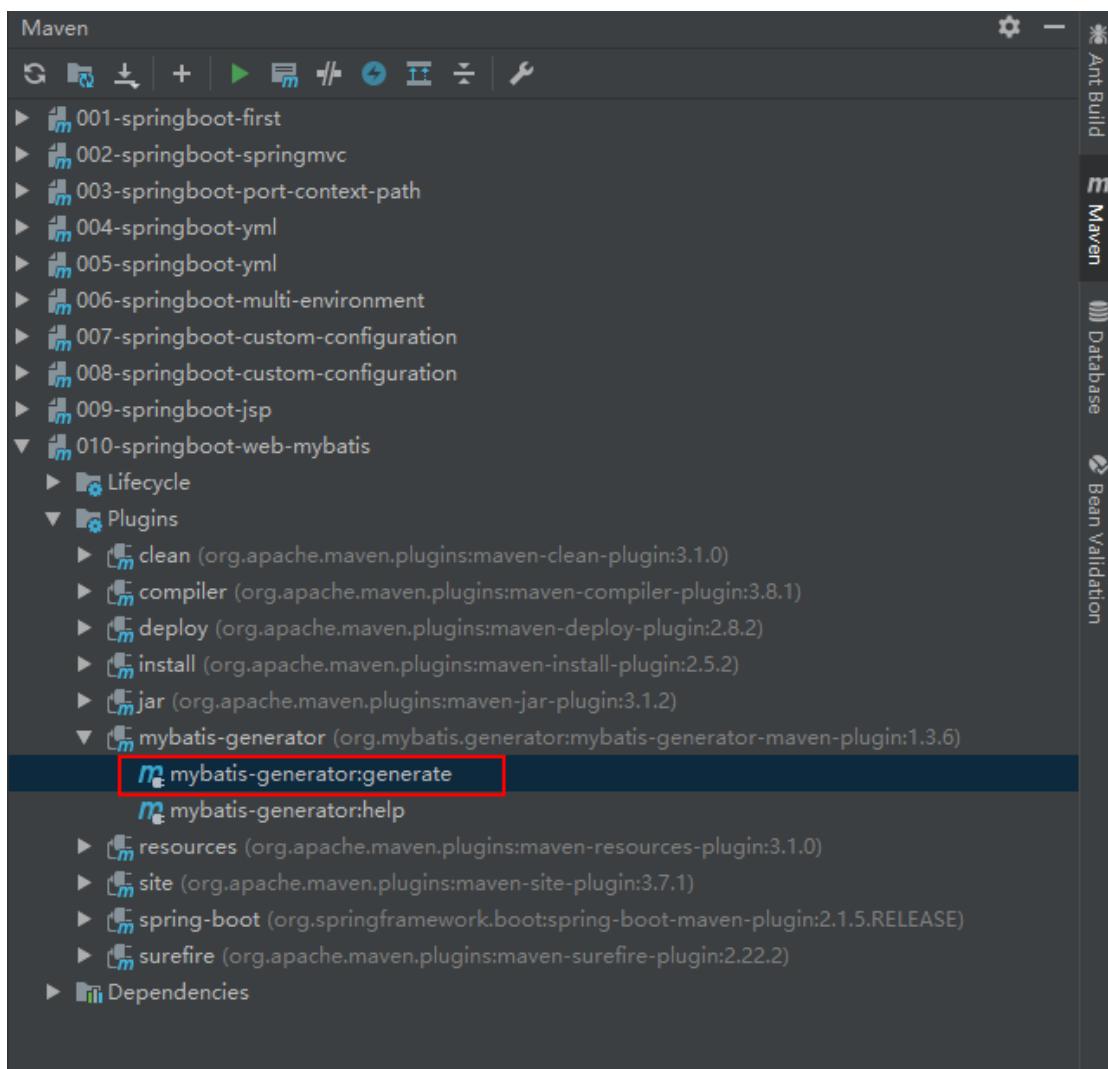
```
        enableSelectByExample="false"
        selectByExampleQueryId="false"/>
    </context>
</generatorConfiguration>
```

13.1.3 在 pom.xml 文件中添加 mysql 反向工程依赖

```
<!--mybatis 代码自动生成插件-->

<plugin>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-maven-plugin</artifactId>
    <version>1.3.6</version>
    <configuration>
        <!--配置文件的位置-->
        <configurationFile>GeneratorMapper.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
    </configuration>
</plugin>
```

13.1.4 双击红色选中命令，生成相关文件



13.2 SpringBoot 项目热部署

在实际开发中，我们修改某些代码逻辑功能或页面都需要重启应用，这无形中降低了开发效率，热部署是指当我们修改代码后，服务能自动重启加载新修改的内容，而不需要重启应用，这样大大提高了我们开发的效率。通常适用于修改页面之后不需要重启服务。

Spring Boot 热部署通过在 pom.xml 中添加一个 spring-boot-devtools 实现。

```
<!--SpringBoot 热部署依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-devtools</artifactId>
<optional>true</optional>
</dependency>
```

该热部署依赖在实际使用中会有一些小问题，明明已经重启，但没有生效，这种情况下，手动重启一下程序；特别是分布式开发，比如 **dubbo** 开发框架，有点问题，需要手动重启。修改完毕后，需要选中项目，在 Build 选项中选择 Build Module。

