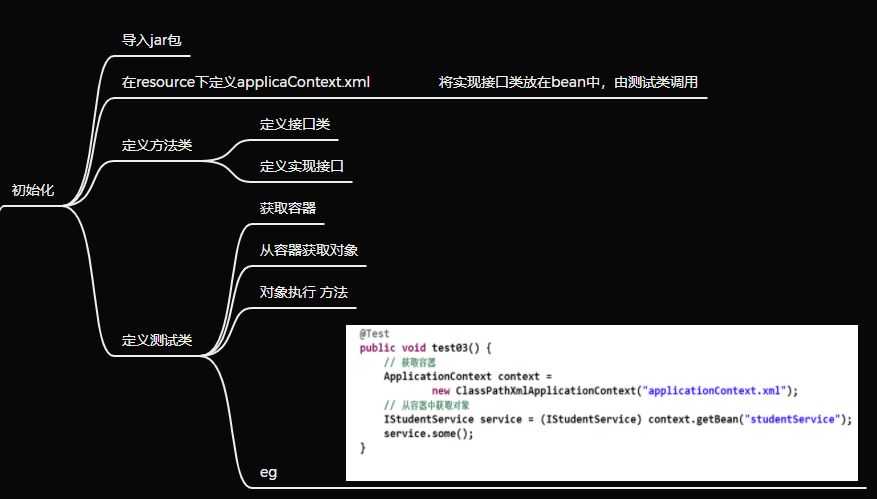
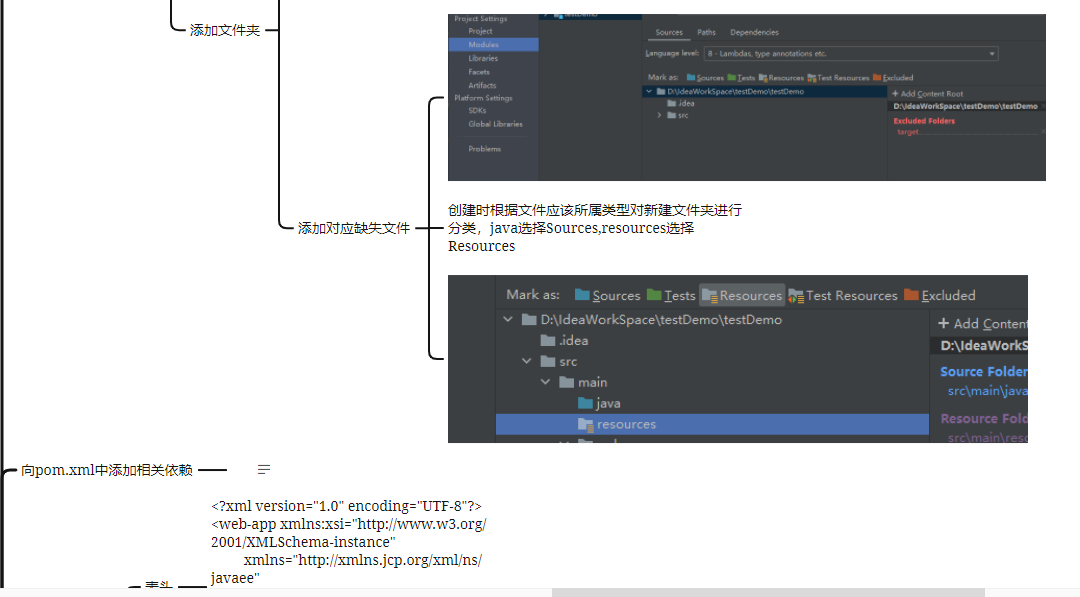
## Spring



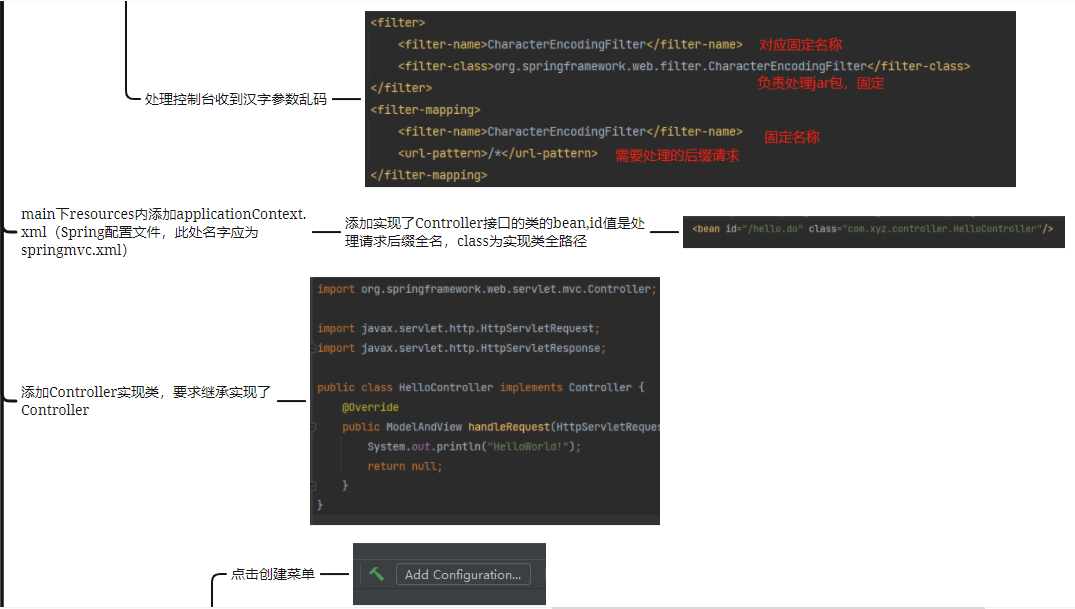
**SpringMVC**

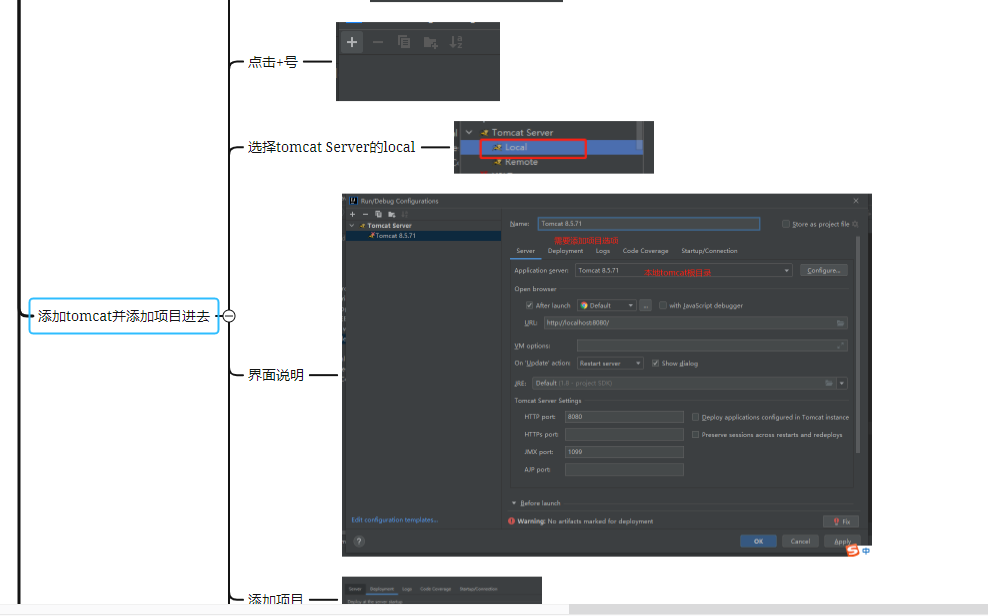






## 





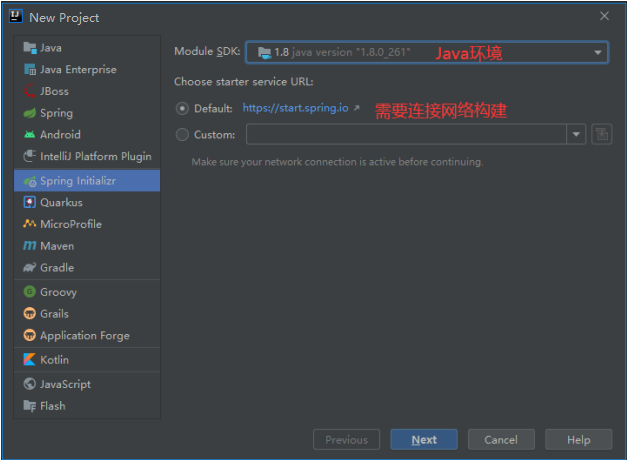




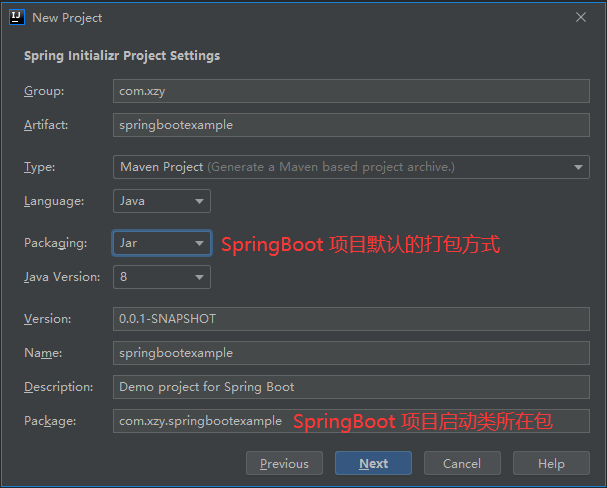
## 第一个SpringBoot项目

**项目名称：springbootexample**

**1、创建一个Project，选择类型为Spring Initializr快速构建：**

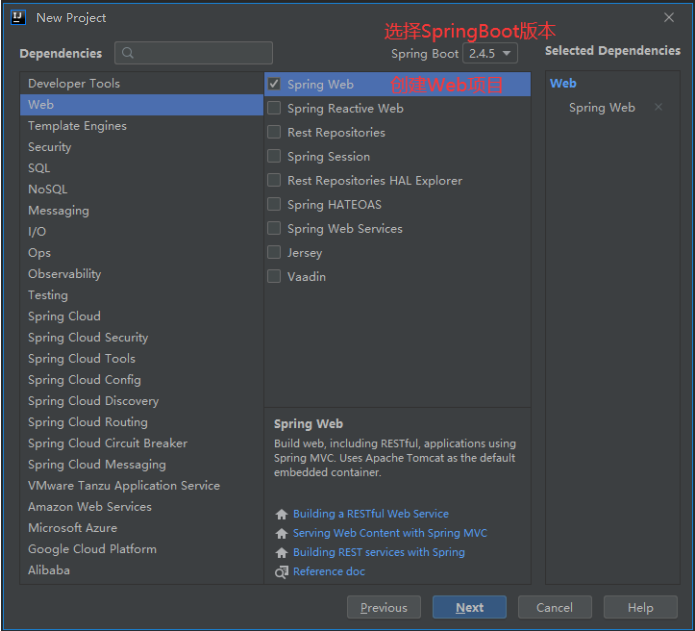


**2、设置GAV坐标及pom配置信息：**

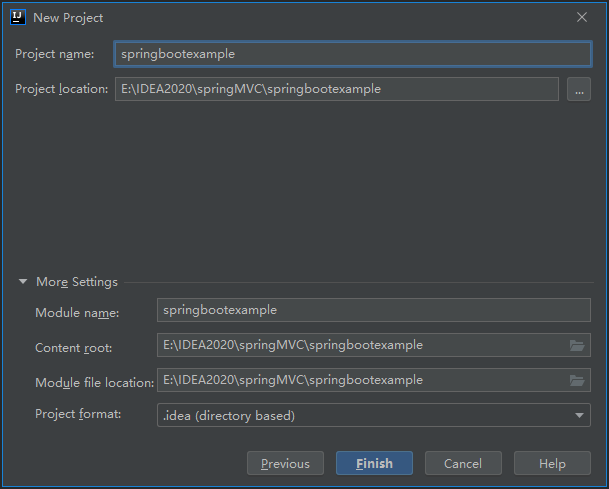


**3、选择SpringBoot版本及依赖：**

SpringBoot会根据选择的依赖自动添加起步依赖并进行自动配置



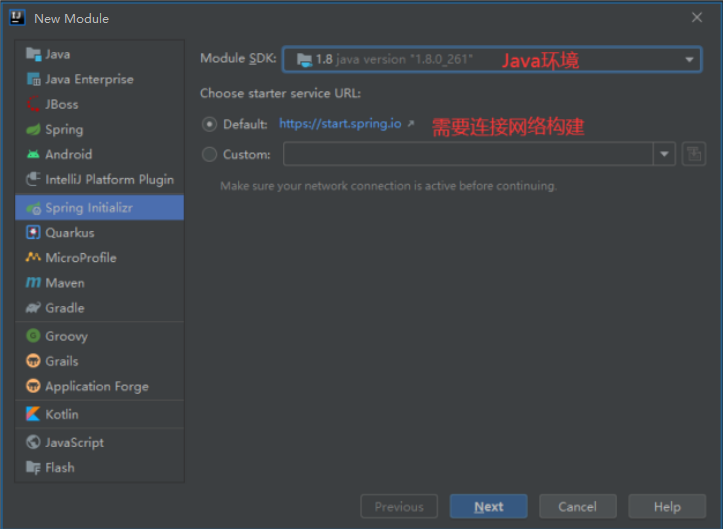
**4、设置模块名称、Content root路径及模块文件的目录：**



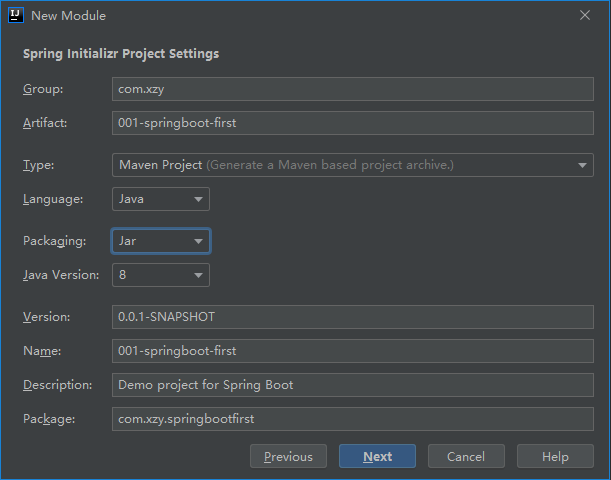
**点击Finish，如果是第一次创建，在右下角会提示正在下载相关的依赖。**

**模块名称：001-springboot-first**

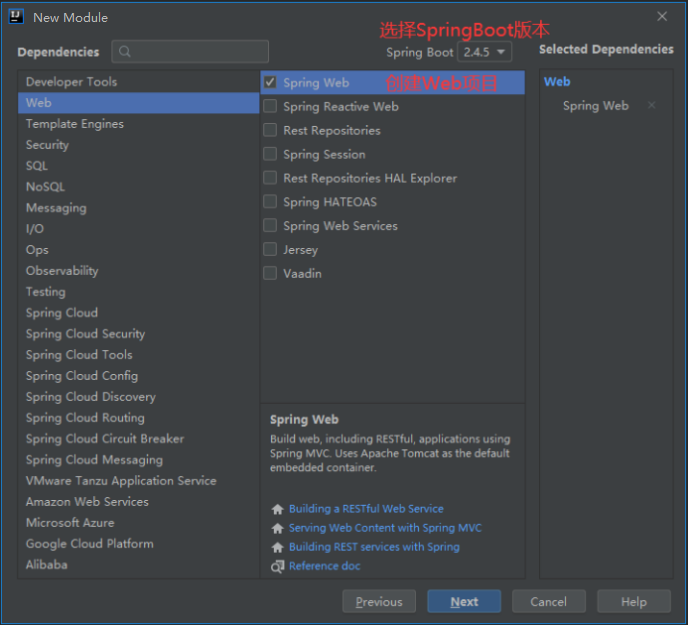
**1、在项目“springbootexample”基础上，创建一个Module，选择类型为Spring Initializr快速构建：**



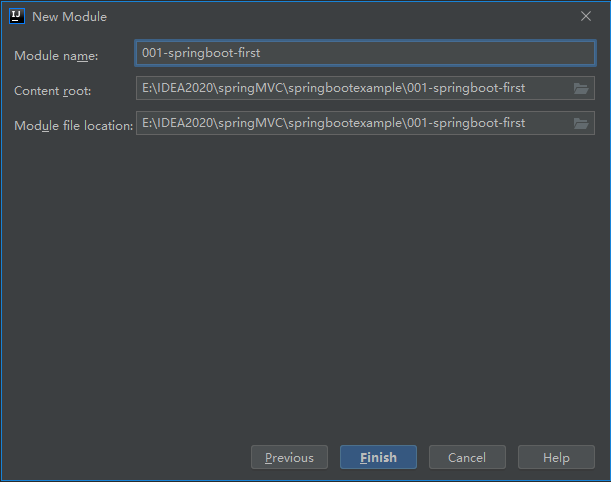
**2、设置GAV坐标及pom配置信息：**



**3、选择SpringBoot版本及依赖**：



**4、设置模块名称、Content root路径及模块文件的目录：**



**5、项目结构说明**

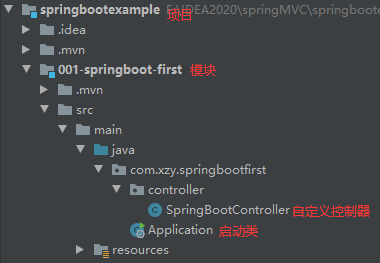
* static：存放静态资源，如图片、CSS、JavaScript等。
* templates：存放Web页面的模板文件。
* application.properties/application.yml：用于存放程序的各种依赖模块的配置信息，比如服务端口，数据库连接配置等。
* .mvn|mvnw|mvnw.cmd：使用脚本操作执行maven相关命令，国内使用较少，可删除。
* gitignore：使用版本控制工具git的时候，设置一些忽略提交的内容。
* static|templates：后面模板技术中存放文件的目录application。
* properties：SpringBoot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。

**6、pom.xml解析**

|  |
| --- |
| <?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 https://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>2.4.5</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <!--当前项目的 GAV 坐标-->  <groupId>com.xzy</groupId>  <artifactId>springbootexample</artifactId>  <version>0.0.1-SNAPSHOT</version>  <!--maven 项目名称，可以删除-->  <name>springbootexample</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>  </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> |

7、创建创建一个Spring MVC的SpringBootController

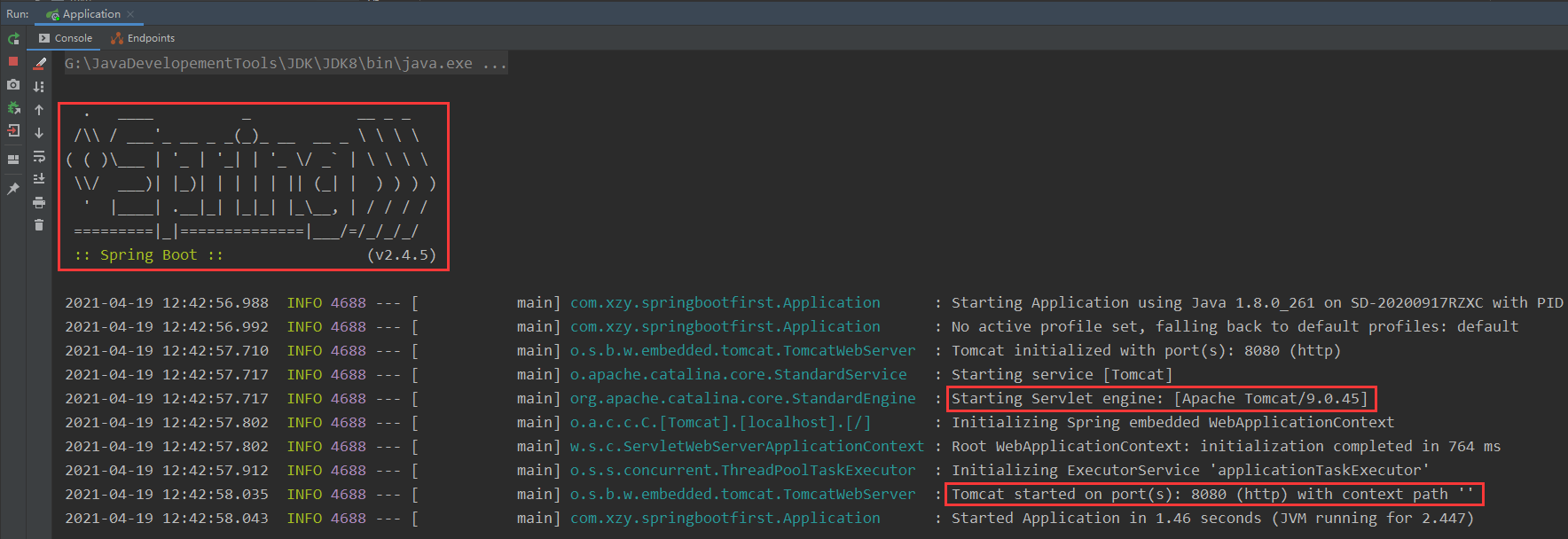
|  |
| --- |
| package com.xzy.springboot.controller; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestBody; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody; @Controller public class SpringBootController {  @RequestMapping("/hello")  public @ResponseBody String hello(){  return "Hello SpringBoot！";  } } |



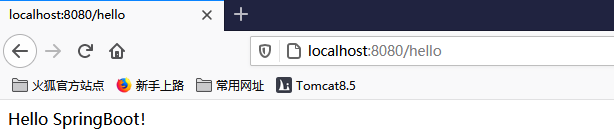
**注意：新创建的类一定要位于Application同级目录或者下级目录，否则SpringBoot加载不到。**

8、在IDEA中右键，运行启动类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。



9、在浏览器地址栏中输入：[http://localhost:8080/hello](http://localhost:8080/hellolai) 来访问。



## 2.2 入门案例分析

* SpringBoot的父级依赖spring-boot-starter-parent配置之后，当前的项目就是SpringBoot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。
* SpringBoot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SpringBoot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依然是我们之前的SpringMVC，因为SpringBoot的里面依然是使用我们的Spring MVC + Spring + MyBatis。

## 2.3 SpringBoot核心配置文件

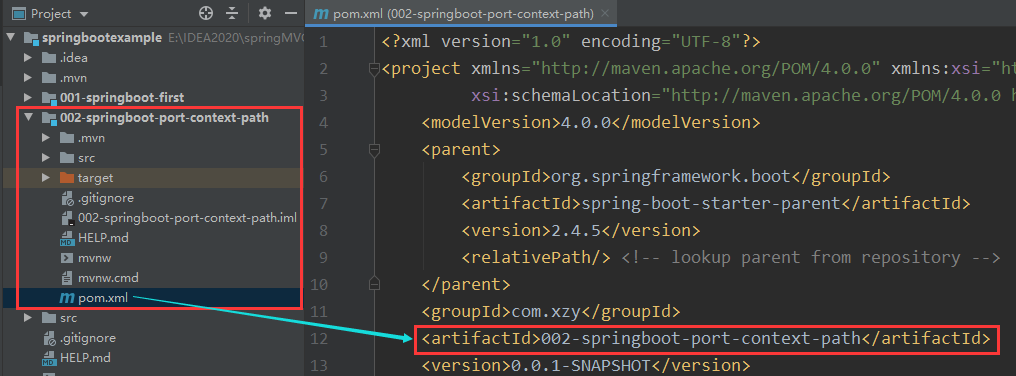
SpringBoot的核心配置文件用于配置SpringBoot程序，名字必须以application开始。

### 2.3.1 .properties文件（默认采用该文件）

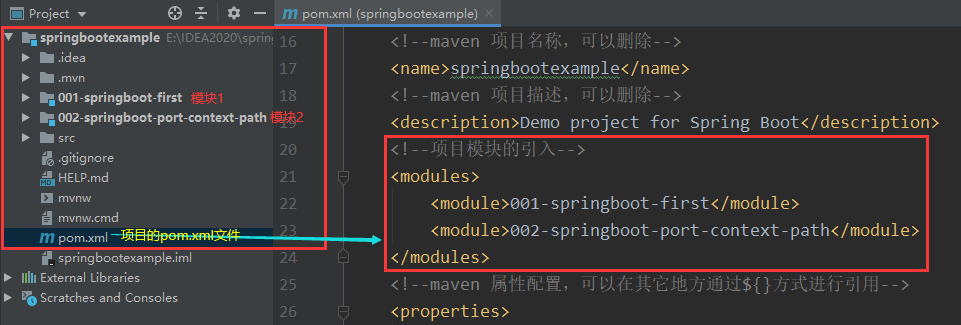
**模块名称：002-springboot-port-context-path**

复制模块“001-springboot-first”，并重命名为：002-springboot-port-context-path。在此基础做如下修改：

（1）修改该模块的pom.xml文件

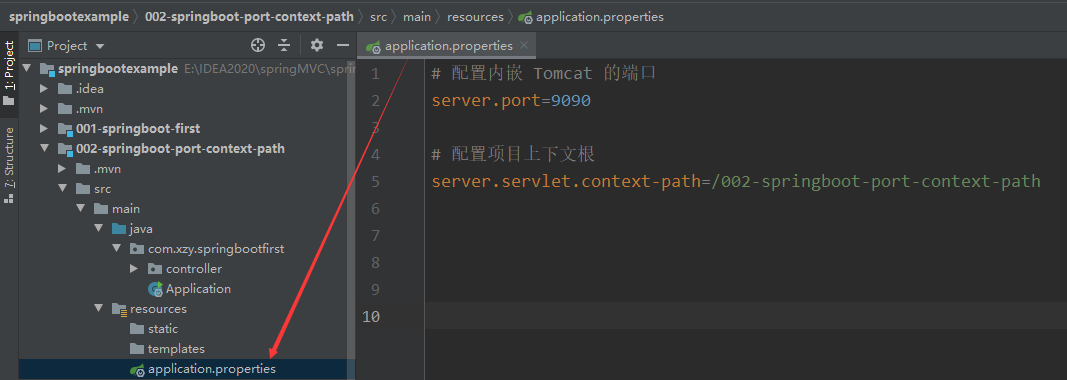


（2）修改项目的pom.xml文件，在项目pom文件中引入所有的module名。

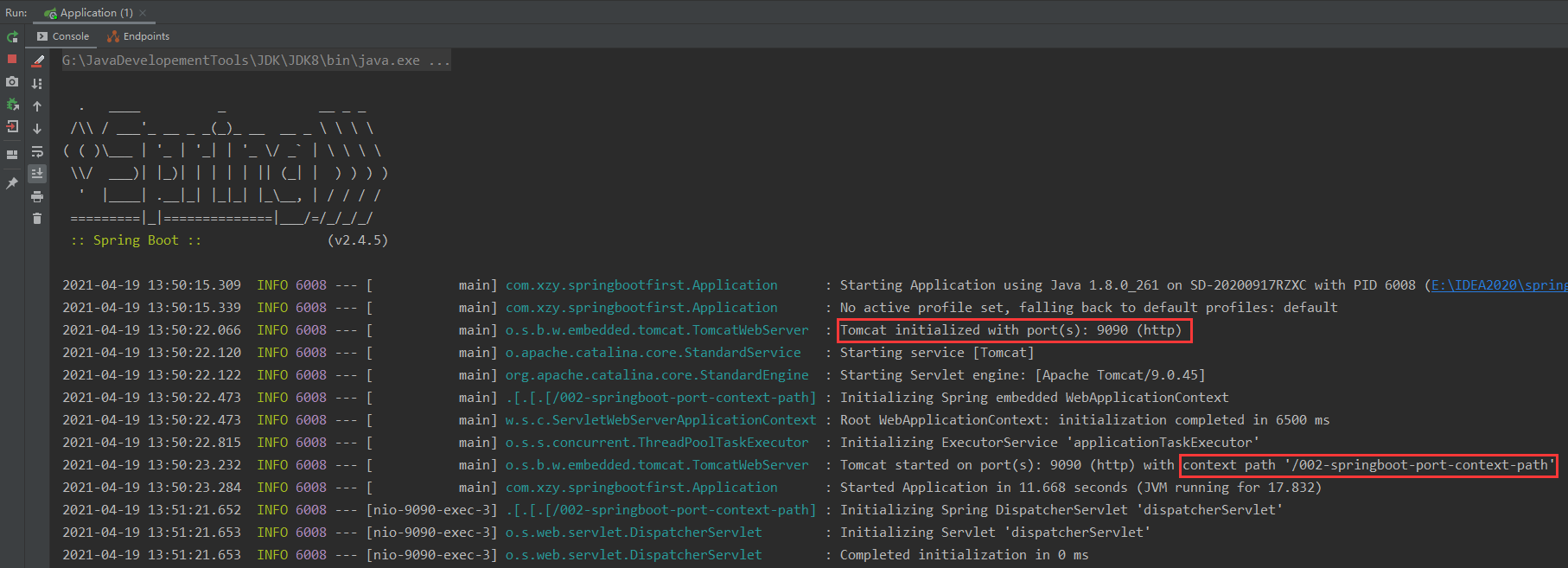


（3）将该模块中的“001-springboot-first.iml”文件，重命名为“002-springboot-port-context-path.iml”。

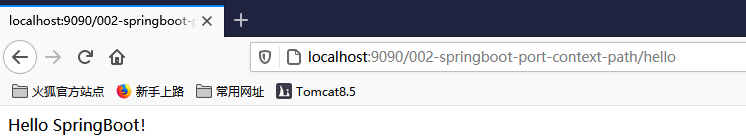
（4）修改该模块的“application.properties”文件。



（5）运行该模块中Application类中的main方法。



（6）在浏览器地址栏中输入：<http://localhost:9090/002-springboot-port-context-path/hello> 来访问。



### 2.3.2 .yml文件

**模块名称：003-springboot-yml**

复制模块“002-springboot-port-context-path”，并重命名为：003-springboot-yml，并参照2.3.1修改相应的配置。

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。



**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，重新运行启动类Application，查看启动的端口及上下文根。**

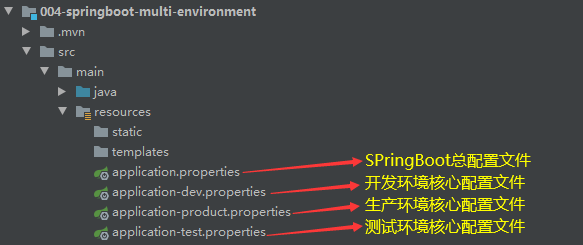
### 2.3.3 多环境配置

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：

**模块名称：004-springboot-multi-environment**

复制模块“002-springboot-port-context-path”，并重命名为：004-springboot-multi-environment，并参照2.3.1修改相应的配置。

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



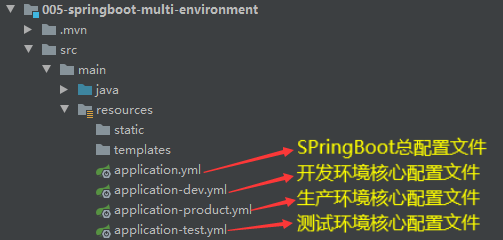
|  |  |
| --- | --- |
| **application-dev.properties文件内容：**  # 开发环境 # 配置内嵌 Tomcat 的端口 server.port=8080 # 配置项目上下文根 server.servlet.context-path=/004-springboot-multi-environment | **application-product.properties文件内容：**  # 生产环境 # 配置内嵌 Tomcat 的端口 server.port=80 # 配置项目上下文根 server.servlet.context-path=/004-springboot-multi-environment |
| **application-test.properties文件内容：**  # 测试环境 # 配置内嵌 Tomcat 的端口 server.port=8081 # 配置项目上下文根 server.servlet.context-path=/004-springboot-multi-environment | a**pplication.properties文件内容：**  # SpringBoot 的总配置文件 # 激活开发环境 spring.profiles.active=dev # 激活生产环境 # spring.profiles.active=product # 激活测试环境 # spring.profiles.active=test |

**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。**

**模块名称：005-springboot-multi-environment**

复制模块“004-springboot-multi-environment”，并重命名为：005-springboot-multi-environment，并参照2.3.1修改相应的配置。

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



|  |  |
| --- | --- |
| **application-dev.yml文件内容：**  *# 开发环境配置文件* server:  port: 8080 *# 配置内嵌 Tomcat 的端口* servlet:  context-path: /dev *# 设置上下文根* | **application-product.yml文件内容：**  *# 生产环境配置文件* server:  port: 80 *# 配置内嵌 Tomcat 的端口* servlet:  context-path: /product *# 设置上下文根* |
| **application-test.yml文件内容：**  *# 测试环境配置文件* server:  port: 8081 *# 配置内嵌 Tomcat 的端口* servlet:  context-path: /test *# 设置上下文根* | a**pplication.yml文件内容：**  *# SpringBoot 的总配置文件*  *# 激活开发环境* spring:  profiles:  active: dev *# 激活生产环境 # spring: # profiles: # active: product  # 激活测试环境 # spring: # profiles: # active: test* |

### 2.3.4 SpringBoot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。

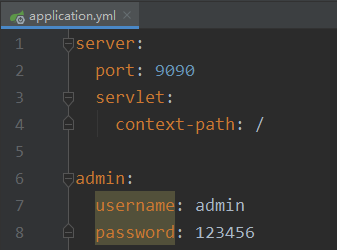
#### 1、@Value注解

用于逐个读取SpringBoot核心配置文件（application.properties或application.yml）中的配置。

**模块名称：006-springboot-custom-configuration**

复制模块“002-springboot-port-context-path”，并重命名为：006-springboot-custom-configuration，并参照2.3.1修改相应的配置。

（1）在核心配置文件application.properties或application.yml中，添加两个自定义配置项。可以看到这两个属性不能被SpringBoot识别，背景是桔色的。

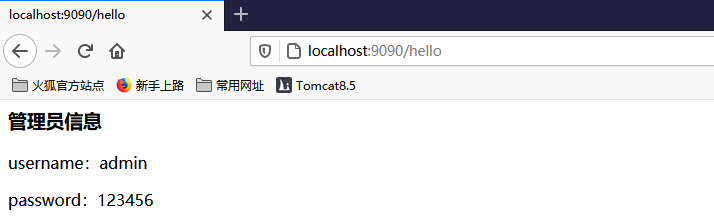
 

（2）测试

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。

|  |
| --- |
| package com.xzy.springboot.controller; import org.springframework.beans.factory.annotation.Value; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestBody; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody; @Controller public class SpringBootController {  @Value("${admin.username}")  private String username;  @Value("${admin.password}")  private String password;   @RequestMapping("/hello")  public @ResponseBody String hello(){  String data = "<h3>管理员信息</h3>" + "<p>username：" + username + "</p>" + "<p>password：" + password + "</p>";  return data;  } } |

（3）在浏览器地址栏中输入：[http://localhost:9090/hello](http://localhost:8080/hellolai) 来访问。



#### 2、@ConfigurationProperties

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。

**模块名称：007-springboot-custom-configuration**

复制模块“006-springboot-custom-configuration”，并重命名为：007-springboot-custom-configuration，并参照2.3.1修改相应的配置。

（1）添加依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-configuration-processor</artifactId>  <optional>true</optional> </dependency> <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId> </dependency> |

（2）创建ConfigInfo类

|  |
| --- |
| package com.xzy.springboot.config; import lombok.Data; import org.springframework.boot.context.properties.ConfigurationProperties; import org.springframework.stereotype.Component; @Component @ConfigurationProperties(prefix = "admin") @Data public class ConfigInfo {  private String username;  private String password; } |

（3）修改SpringBootController类

|  |
| --- |
| package com.xzy.springboot.controller; import com.xzy.springboot.config.ConfigInfo; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody; import javax.annotation.Resource; @Controller public class SpringBootController {  @Resource  private ConfigInfo configInfo;   @RequestMapping("/config")  public @ResponseBody String hello(){  String data = "<h3>管理员信息</h3>" + "<p>username：" + configInfo.getUsername() + "</p>" + "<p>password："  + configInfo.getPassword() + "</p>";  return data;  } } |

## 2.4 SpringBoot前端使用JSP

**模块名称：008-springboot-jsp**

复制模块“007-springboot-custom-configuration”，并重命名为：008-springboot-jsp，并参照2.3.1修改相应的配置。

（1）添加依赖

|  |
| --- |
| <?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.4.5</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <groupId>com.xzy</groupId>  <artifactId>008-springboot-jsp</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>008-springboot-jsp</name>  <description>Demo project for Spring Boot</description>  **<!--JavaWeb项目的打包方式，必须修改-->**  **<packaging>war</packaging>**   <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>   <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  </dependency>  <!--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>  <scope>provided</scope>  </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>  </dependencies> |

（2）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。其实官方已经建议使用模板技术。

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins> <!--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-->  <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> </build> |

（3）在application.properties文件配置SpringMVC的视图展示为jsp，这里相当于SpringMVC的配置。

|  |
| --- |
| server.port=9090 server.servlet.context-path=/ # 配置 SpringMVC 的视图解析器 # 其中：/ 表示目录为 src/main/webapp spring.mvc.view.prefix=/WEB-INF/jsp/ spring.mvc.view.suffix=.jsp |

（4）修改SpringBootController类

|  |
| --- |
| package com.xzy.springboot.controller; import org.springframework.stereotype.Controller; import org.springframework.ui.Model; import org.springframework.web.bind.annotation.RequestMapping; @Controller public class SpringBootController {  @RequestMapping("/indexJsp")  public String indexJsp(Model model){  model.addAttribute("msg", "SpringBoot 前端使用 JSP 页面！");  return "index";  } } |

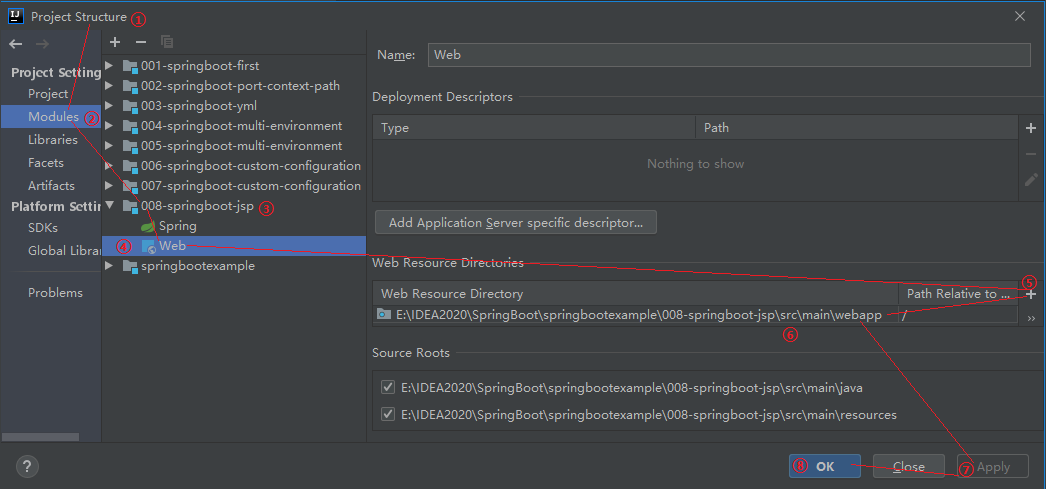
（5）完善项目的目录结构，具体操作如下：

① 在src/main目录下创建webapp目录

② 在src/main/webapp目录下创建WEB-INF目录

③ 在src/main/webapp/WEB-INF目录下创建jsp目录

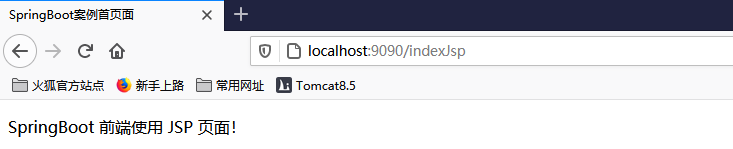
④ 配置web项目的根目录，如下图所示：



（6）在src/main/webapp/WEB-INF/jsp下新建index.jsp文件。

|  |
| --- |
| <%@ **page** language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %> <!DOCTYPE html> <html> <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>SpringBoot案例首页面</title> </head> <body>  <p> ${msg} </p> </body> </html> |

（7）运行启动类Application类中的main方法，在浏览器地址栏中输入：[http://localhost:9090/indexJsp](http://localhost:8080/hellolai) 来访问



# 第3章 SpringBoot框架Web开发

## 3.1 SpringBoot集成MyBatis框架

**模块名称：009-springboot-web-mybatis**

复制模块“008-springboot-jsp”，并重命名为：009-springboot-web-mybatis，并参照2.3.1修改相应的配置，删除webapp目录。

### 3.1.1 案例一

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。

Mapper接口与Mapper映射文件在同一个包下。

**（1）添加依赖**

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!--lombok依赖-->  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  </dependency>  <!--MyBatis 整合 SpringBoot 的起步依赖-->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>2.1.4</version>  </dependency>  <!-- MySQL驱动的依赖 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.5</version>  </dependency> </dependencies> |

**（2）编写实体类、Mapper接口**

|  |  |
| --- | --- |
| package com.xzy.springboot.model; import lombok.Data; @Data public class Student {  private Integer studentId;  private String username;  private String password;  private String name;  private String sex;  private Integer age;  private Integer score;  private String clazz;  private String tel;  private String qq; } | package com.xzy.springboot.mapper; import com.xzy.springboot.model.Student; import org.apache.ibatis.annotations.Mapper; import java.util.List;  **// @Mapper 作用：mybatis 自动扫描数据持久层的映射文件及 Mapper 接口的关系** @Mapper public interface StudentMapper {  // 根据专业班级查询  List<Student> selectStudents(String clazz); } |

**（3）在com.xzy.springboot.mapper包下编写MyBatis映射文件：StudentMapper.xml**

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> <mapper namespace="com.xzy.springboot.mapper.StudentMapper">  <!-- 根据专业班级查询 -->  <select id="selectStudents" resultType="com.xzy.springboot.model.Student">  select \* from t\_student where clazz=#{clazz}  </select> </mapper> |

1. **修改pom.xml文件**

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

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

（5）编写service层接口和实现类

|  |  |
| --- | --- |
| package com.xzy.springboot.service; import com.xzy.springboot.model.Student; import java.util.List; public interface StudentService {  // 根据专业班级查询  List<Student> selectStudents(String clazz); } | package com.xzy.springboot.service.impl; import com.xzy.springboot.mapper.StudentMapper; import com.xzy.springboot.model.Student; import com.xzy.springboot.service.StudentService; import org.springframework.stereotype.Service; import javax.annotation.Resource; import java.util.List; @Service public class StudentServiceImpl implements StudentService {  @Resource  private StudentMapper studentMapper;  @Override  public List<Student> selectStudents(String clazz) {  return studentMapper.selectStudents(clazz);  } } |

（6）修改SpringBootController类

|  |
| --- |
| package com.xzy.springboot.controller; import com.xzy.springboot.model.Student; import com.xzy.springboot.service.StudentService; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody; import javax.annotation.Resource; import java.util.List; @Controller  @RequestMapping("/student") public class SpringBootController {  @Resource  private StudentService studentService;  @RequestMapping("/selectStudentByClazz")  public @ResponseBody Object selectStudentByClazz(String clazz){  List<Student> studentList = studentService.selectStudents(clazz);  return studentList;  }  } |

1. 在SpringBoot的配置文件中application.properties配置数据源

|  |
| --- |
| server.port=9090 server.servlet.context-path=/ # 配置数据库连接信息 spring.datasource.driver-class-name=com.mysql.jdbc.Driver spring.datasource.url=jdbc:mysql://localhost:3306/springboot?characterEncoding=UTF-8 spring.datasource.username=root spring.datasource.password=123456 |

1. 运行启动类Application类中的main方法，在浏览器地址栏中输入：[http://localhost:9090/student/selectStudentByClazz?clazz=物联网工程1701班](http://localhost:8080/hellolai) 来访问。
2. 修改该项目，具体如下：

① 删除Mapper接口上的@Mapper注解

② 在启动类Application类上加注解@MapperScan

|  |
| --- |
| package com.xzy.springboot; import org.mybatis.spring.annotation.MapperScan; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; @SpringBootApplication  **// Mybatis 提供的注解：扫描数据持久层的 mapper 映谢配置文件，Mapper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.xzy.springboot.mapper" )  // 或 // @MapperScan("com.xzy.springboot.mapper") public class Application {  public static void main(String[] args) {  SpringApplication.*run*(Application.class, args);  } } |

### 3.1.2 案例二

Mapper接口与Mapper映射文件分开。

**模块名称：010-springboot-web-mybatis**

复制模块“009-springboot-web-mybatis”，并重命名为：010-springboot-web-mybatis，并参照2.3.1修改相应的配置。

（1）在src/main/resources目录下创建mapper目录，并将com.xzy.springboot.mapper包下的映射文件StudentMapper.xml剪切到mapper目录下。

（2）在SpringBoot配置文件application.properties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）删除pom.xml文件中<resources>标签的所有配置。

（4）运行启动类Application类中的main方法，在浏览器地址栏中输入：[http://localhost:9090/student/selectStudentByClazz?clazz=物联网工程1702班](http://localhost:8080/hellolai) 来访问。

## 3.2 SpringBoot事务支持

SpringBoot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。具体操作如下：

* 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。

**模块名称：011-springboot-web-mybatis-transacation**

复制模块“010-springboot-web-mybatis”，并重命名为：011-springboot-web-mybatis-transacation，并参照2.3.1修改相应的配置。

**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。**

（1）在StudentMapper接口中添加如下代码：

|  |
| --- |
| // 增加学生 int insertStudent(Student student); |

（2）映射文件StudentMapper.xml中添加SQL语句

|  |
| --- |
| <!--增加学生--> <insert id="insertStudent" >  insert into t\_student (studentId,username,password) values(#{studentId},#{username},#{password}) </insert> |

（3）StudentService接口中增加方法

|  |
| --- |
| // 增加学生 Integer insertStudent(Student student); |

（4）StudentService层接口StudentServiceImpl实现类中增加方法

|  |
| --- |
| // 添加学生 @Override @Transactional // 添加此注解说明该方法添加的事务管理 public Integer insertStudent(Student student) {  int updateCount = studentMapper.insertStudent(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; } |

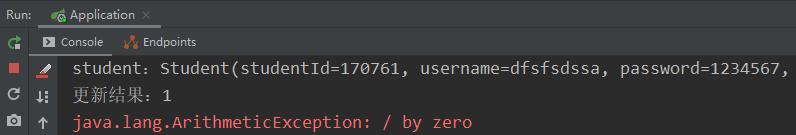
（5）控制类SpringBootController中增加方法

|  |
| --- |
| // 增加学生 @RequestMapping("/addStudent") public @ResponseBody Object insertStudent(Student student){  System.*out*.println("student：" + student);  Integer count = 0;  try{  count = studentService.insertStudent(student);  }catch (Exception e){  e.printStackTrace();  return "fail";  }  return count; } |

（6）运行启动类Application类中的main方法，在浏览器地址栏中输入：

[http://localhost:10086/student/addStudent?studentId=170761&username=dfsfsdssa&password=1234567](http://localhost:8080/hellolai) 来访问。

运行结果：



查看数据库中对应的表，发现数据库未添加成功。

（7）删除StudentServiceImpl实中insertStudent方法方法上的事务注解 @Transactional，重新运行启动类Application类中的main方法，在浏览器地址栏中输入：

[http://localhost:10086/student/addStudent?studentId=170761&username=dfsfsdssa&password=1234567](http://localhost:8080/hellolai) 来访问。

查看数据库中对应的表，发现数据库添加成功。通过以上结果，说明事务起作用了

## 3.3 SpringBoot下的Spring MVC

SpringBoot下的SpringMVC和之前的SpringMVC使用是完全一样的，主要有以下注解：

### 3.3.1 @Controller注解

Spring MVC的注解，在SpringMVC中，控制器Controller负责处理由DispatcherServlet分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个Model，然后再把该Model返回给对应的View进行展示。在SpringMVC中提供了一个非常简单的定义Controller的方法，你无需继承特定的类或实现特定的接口，只需使用@Controller标记一个类是Controller，然后使用@RequestMapping和@RequestParam等一些注解用以定义URL请求和Controller方法之间的映射，这样的Controller就能被外界访问到。此外Controller不会直接依赖于HttpServletRequet和HttpServletResponse等HttpServlet对象。

### 3.3.2 @RestController注解

Spring4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。

### 3.3.3 @RequestMapping注解（常用）

**（1）@RequestMapping参数说明：**

value：定义处理方法的请求的URL地址。（重点）

　　method：定义处理方法的http method类型，如GET、POST等。（重点）

　　params:定义请求的URL中必须包含的参数。或者不包含某些参数。（了解）

headers:定义请求中Request Headers必须包含的参数。或者不包含某些参数。（了解）

**（2）@RequestMapping的用法：**

@RequestMapping有两种标注方式，一种是标注在类级别上，一种是标注在方法级别上。标注在方法上时，value表示访问该方法的URL地址。标注在类上时，value相当于一个命名空间，即访问该Controller下的任意方法都需要带上这个命名空间。

**（3）@RequestMapping(method)：指定页面请求方式**

|  |
| --- |
| @RequestMapping(value = "/register", method = RequestMethod.*GET*) public String register() {  return "example\_register\_page"; } |

method的值一旦指定，那么，处理方法就只对指定的http method类型的请求进行处理。 这里方法/register只能使用get请求，使用post请求无法访问。

|  |
| --- |
| @RequestMapping(value = "/register", method = RequestMethod.*GET*) public String register1(){  return "example\_register\_get\_page"; } @RequestMapping(value = "/register", method = RequestMethod.*POST*) public String register2(){  return "example\_register\_post\_page"; } |

可以为多个方法映射相同的URI，不同的http method类型，SpringMVC根据请求的method类型是可以区分开这些方法的。当/example/register是以GET的方式提交的时候，SpringMVC调用register1()来处理请求；若是以POST的方式提交，则调register2()来处理提交的请求。

method若是缺省没指定，并不是说它默认只处理GET方式的请求，而是它可以处理任何方式的http method类型的请求。指定method是为了细化映射 ( 缩小处理方法的映射范围 )，在method没有指定的情况下，它的映射范围是最大的。

**（4）@RequestMapping(params)：**

　　与method相类似，作用是为了细化映射。只有当URL中包含与params值相匹配的参数的请求，处理方法才会被调用。

|  |
| --- |
| @RequestMapping(value = "/find", params = "target") public String find1(){  return "example\_find1\_page"; } @RequestMapping(value = "/find", params = "!target") public String find2(){  return "example\_find2\_page"; } @RequestMapping(value = "/search", params = "target=product") public String search1(){  return "example\_search1\_page"; } @RequestMapping(value = "/search", params = "target!=product") public String search2(){  return "example\_search2\_page"; } |

find1()：请求的URL中必须要有target参数，才能够到达此方法。如 /example/find.action?target或/example/find.action?target=x等。

　　find2()：请求的URL中必须不能有target参数，才能够到达此方法。如 /example/find.action或/example/find.action?q=x等。

　　search1()：请求的URL中必须要有target=product参数，才能够到达此方法。如/example/search.action?target=product等。

search2()：请求的URL中必须不能有target=product参数，才能够到达此方法。如/example/search.action?target=article等。

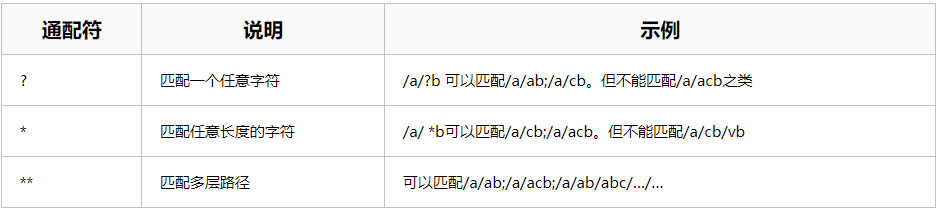
**（5）@RequestMapping(headers)：**

headers的作用也是用于细化映射。只有当请求的Request Headers中包含与heanders值相匹配的参数，处理方法才会被调用。

|  |
| --- |
| @RequestMapping(value = "/specify", headers = "accept=text/\*") public String specify() {  return "example\_specify\_page"; } |

请求的 Request Headers 中 Accept 的值必须匹配 text/\* ( 如 text/html )，方法才会被调用。

**（6）@RequestMapping支持Ant风格的通配符：**



### 3.3.4 @GetMapping注解

RequestMapping和Get请求方法的组合，只支持Get请求。@GetMapping是一个组合注解，是@RequestMapping(method = RequestMethod.GET)的缩写。

Get请求主要用于查询操作。

### 3.3.5 @PostMapping注解

RequestMapping和Post请求方法的组合，只支持Post请求，@PostMapping是一个组合注解，是@RequestMapping(method = RequestMethod.POST)的缩写。

Post请求主要用户新增数据。

### 3.3.6 @PutMapping注解

RequestMapping和Put请求方法的组合，只支持Put请求，@PutMapping是一个组合注解，是@PutMapping(method = RequestMethod.PUT)的缩写。

Put通常用于修改数据。

### 3.3.7 @DeleteMapping

RequestMapping和Delete请求方法的组合，@DeleteMapping是一个组合注解，是@DeleteMapping(method = RequestMethod.DELETE)的缩写。

只支持Delete请求，通常用于删除数据。

### 3.3.8 综合案例

**案例作用：演示常见的SpringMVC注解**

**模块名称：012-springboot-springmvc**

复制模块“011-springboot-web-mybatis-transacation”，并重命名为：012-springboot-springmvc，并参照2.3.1修改相应的配置。

（1）修改StudentMapper接口：

|  |
| --- |
| public interface StudentMapper {  // 增加学生  int insertStudent(Student student);  // 删除学生  int deleteStudent(Integer studentId);  // 修改学生信息  int updateStudent(Student student);  // 查询学生信息（根据学号studentId查询）  Student selectStudentByStudentId(Integer studentId); } |

（2）修改映射文件StudentMapper.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> <mapper namespace="com.xzy.springboot.mapper.StudentMapper">  <!--增加学生-->  <insert id="insertStudent" >  insert into  t\_student (username,password,name,sex,age,score,clazz,tel,qq)  values  (#{username},#{password},#{name},#{sex},#{age},#{score},#{clazz},#{tel},#{qq})  </insert>  <!-- 删除学生 -->  <delete id="deleteStudent" >  delete from t\_student where studentId=#{studentId}  </delete>  <!-- 修改学生信息（密码、成绩） -->  <update id="updateStudent" >  update  t\_student  set  password=#{password}, score=#{score}  where  studentId=#{studentId}  </update>  <!-- 查询学生信息 -->  <select id="selectStudentByStudentId" resultType="com.xzy.springboot.model.Student">  select \* from t\_student where studentId=#{studentId}  </select> </mapper> |

（3）修改StudentService接口

|  |
| --- |
| public interface StudentService {  // 增加学生  int insertStudent(Student student);  // 删除学生  int deleteStudent(Integer studentId);  // 修改学生信息  int updateStudent(Student student);  // 查询学生信息（根据学号studentId查询）  Student selectStudentByStudentId(Integer studentId); } |

（4）StudentService层接口StudentServiceImpl实现类中增加方法

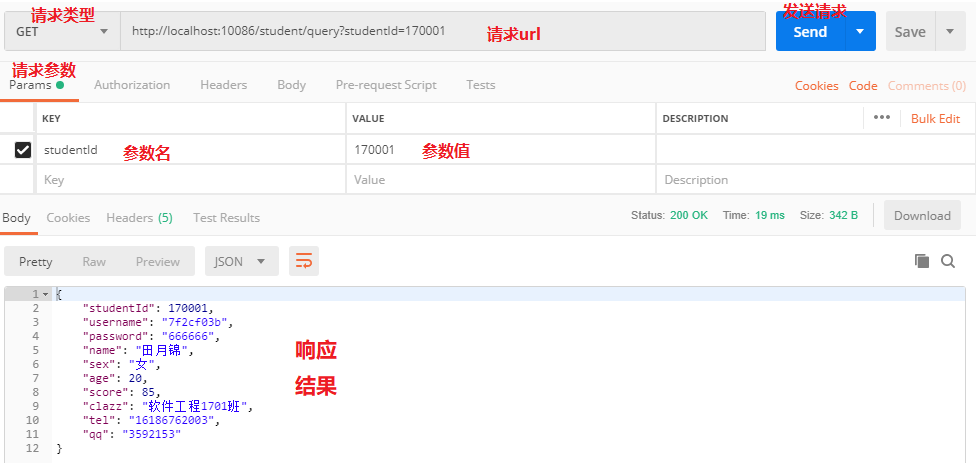
|  |
| --- |
| @Service public class StudentServiceImpl implements StudentService {  @Resource  private StudentMapper studentMapper;  @Override  public int insertStudent(Student student) {  return studentMapper.insertStudent(student);  }   @Override  public int deleteStudent(Integer studentId) {  return studentMapper.deleteStudent(studentId);  }  @Override  public int updateStudent(Student student) {  return studentMapper.updateStudent(student);  }  @Override  public Student selectStudentByStudentId(Integer studentId) {  return studentMapper.selectStudentByStudentId(studentId);  } } |

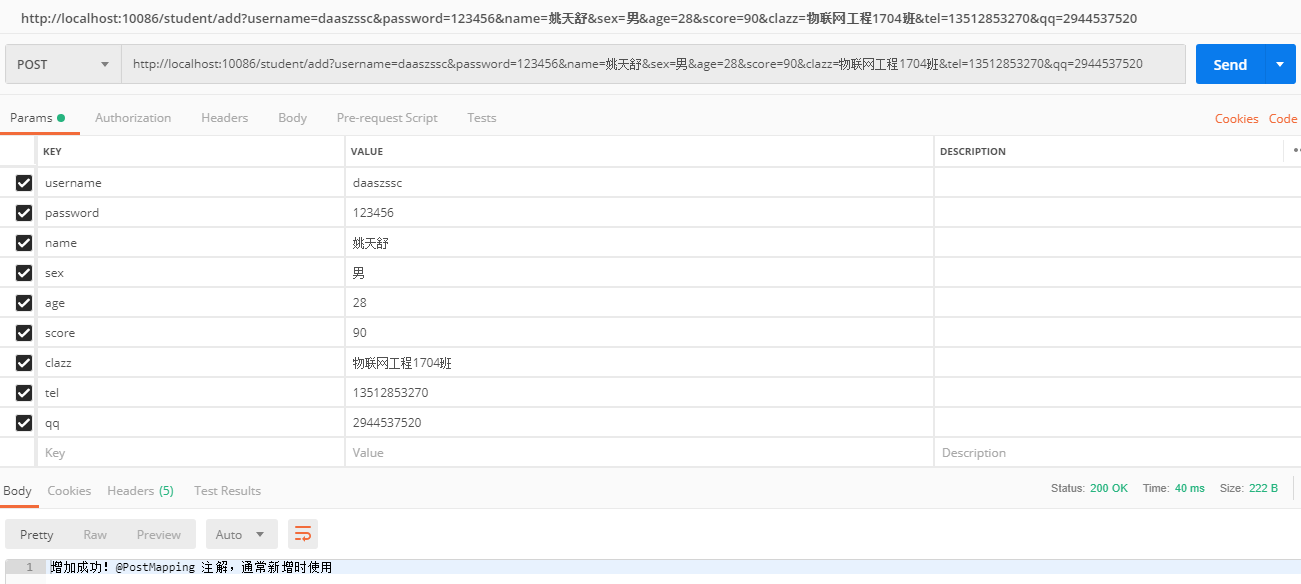
（5）修改控制类SpringBootController

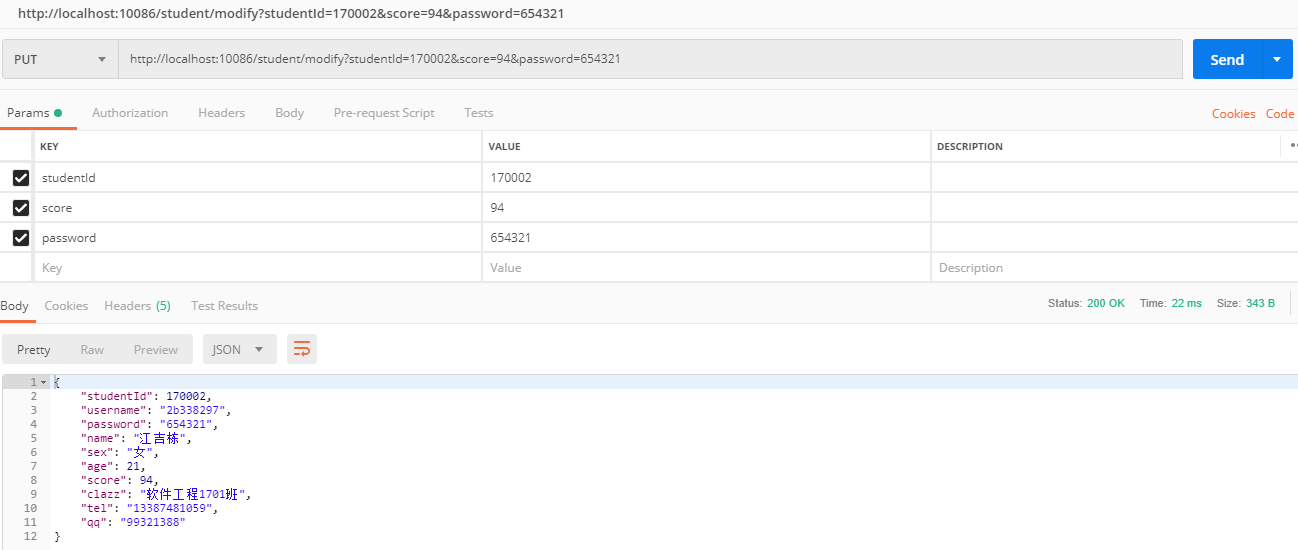
|  |
| --- |
| @RestController @RequestMapping("/student") public class SpringBootController {  @Resource  private StudentService studentService;  @PostMapping(value = "/add")  public String insertStudent(Student student) {  int flag = studentService.insertStudent(student);  if(flag==1)  return "增加成功！@PostMapping 注解，通常新增时使用";  else  return "增加失败！@PostMapping 注解，通常新增时使用";  }  @DeleteMapping(value = "/delete")  public String deleteStudent(Integer studentId) {  int flag = studentService.deleteStudent(studentId);  if(flag==1)  return "删除成功！@DeleteMapping 注解，通常删除数据时使用";  else  return "删除失败！@DeleteMapping 注解，通常删除数据时使用";  }  @PutMapping(value = "/modify")  public Object updateStudent(Student student) {  int flag = studentService.updateStudent(student);  if(flag==1)  return studentService.selectStudentByStudentId(student.getStudentId());  else  return "修改失败！@PutMapping 注解，通常更新数据时使用";  }  @GetMapping(value = "/query")  public Object selectStudentByStudentId(Integer studentId) {  Student student = studentService.selectStudentByStudentId(studentId);  if(student != null)  return student;  else  return "查询失败！@GetMapping 注解,通常查询时使用";  } } |

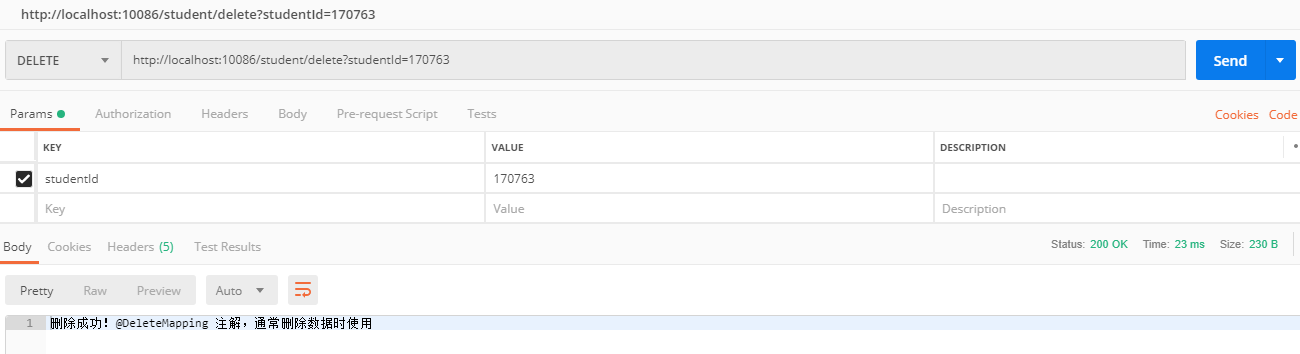
（6）Http接口请求工具Postman测试

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。









## 3.4 SpringBoot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 SpringBoot开发RESTFul

SpringBoot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风格模拟实现增删改查操作

该案例集成了MyBatis、spring、SpringMVC，通过模拟实现对学生的增删改查操作。

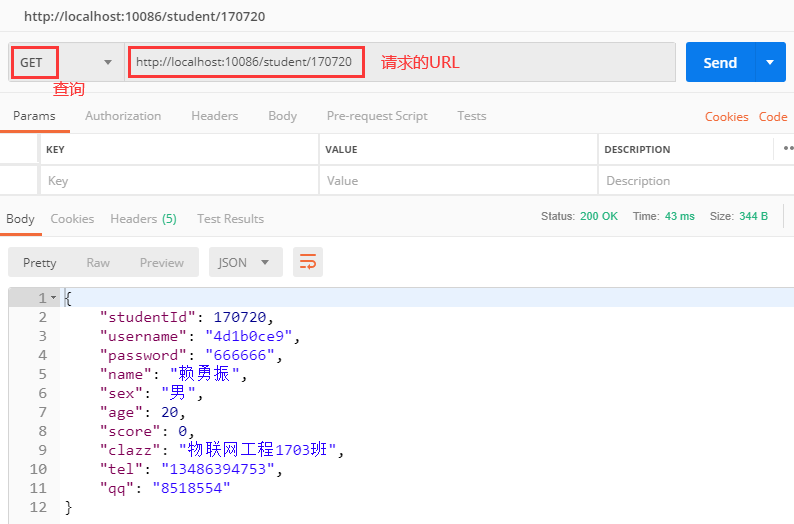
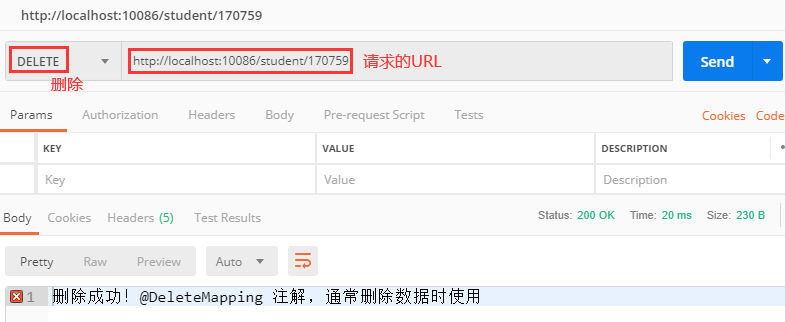
**模块名称：013-springboot-restful**

复制模块“012-springboot-springmvc”，并重命名为：013-springboot-restful，并参照2.3.1修改相应的配置。

1. 修改控制类SpringBootController中的删除、查询方法。

|  |
| --- |
| @GetMapping(value = "/query/{studentId}") public Object selectStudentByStudentId(@PathVariable("studentId") Integer studentId) {  Student student = studentService.selectStudentByStudentId(studentId);  if(student != null)  return student;  else  return "查询失败！@GetMapping 注解,通常查询时使用"; }  @DeleteMapping(value = "/delete/{studentId}") public String deleteStudent(@PathVariable("studentId") Integer studentId) {  int flag = studentService.deleteStudent(studentId);  if(flag==1)  return "删除成功！@DeleteMapping 注解，通常删除数据时使用";  else  return "删除失败！@DeleteMapping 注解，通常删除数据时使用"; } |

1. 通过Postman工具进行测试。

（3）RESTFUL风格的优点

* 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 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 SpringBoot集成MyBatisPlus

**案例作用：演示MyBatisPlus对单表的增删改查**

**模块名称：014-springboot-mybatisPlus**

复制模块“012-springboot-springmvc”，并重命名为：014-springboot-mybatisPlus，并参照2.3.1修改相应的配置。

1. 引入依赖

|  |
| --- |
| <?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  https://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.4.5</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <groupId>com.xzy</groupId>  <artifactId>014-springboot-mybatisPlus</artifactId>  <version>0.0.1-SNAPSHOT</version>  <name>014-springboot-mybatisPlus</name>  <description>Demo project for Spring Boot</description>  <packaging>war</packaging>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!--lombok依赖-->  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  </dependency>  <!--MyBatisPlus 整合 SpringBoot 的起步依赖-->  <dependency>  <groupId>com.baomidou</groupId>  <artifactId>mybatis-plus-boot-starter</artifactId>  <version>3.3.2</version>  </dependency>  <!-- MySQL驱动的依赖 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.5</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build> </project> |

（1）修改StudentMapper接口：

|  |
| --- |
| package com.xzy.springboot.mapper; import com.baomidou.mybatisplus.core.mapper.BaseMapper; import com.xzy.springboot.model.Student; import org.springframework.stereotype.Repository; @Repository public interface StudentMapper extends BaseMapper<Student> { } |

（2）删除映射文件StudentMapper.xml，并修改SpringBoot配置文件application.properties

|  |
| --- |
| server.port=10086 server.servlet.context-path=/ # 配置数据库连接信息 spring.datasource.driver-class-name=com.mysql.jdbc.Driver spring.datasource.url=jdbc:mysql://localhost:3306/springboot?characterEncoding=UTF-8 spring.datasource.username=root spring.datasource.password=123456 # 指定 MyBatis 映射文件的路径 # mybatis.mapper-locations=classpath:mapper/\*Mapper.xml # mybatis-plus 日志 mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl |

（3）修改StudentService接口

|  |
| --- |
| public interface StudentService {  // 增加学生  int insertStudent(Student student);  // 删除学生  int deleteStudent(Integer studentId);  // 修改学生信息  int updateStudent(Student student);  // 查询学生信息（根据学号studentId查询）  Student selectStudentById(Integer studentId); } |

（4）修改StudentService层接口StudentServiceImpl实现类

|  |
| --- |
| @Service public class StudentServiceImpl implements StudentService {   @Resource  private StudentMapper studentMapper;   @Override  public int insertStudent(Student student) {  return studentMapper.insert(student);  }   @Override  public int deleteStudent(Integer studentId) {  return studentMapper.deleteById(studentId);  }   @Override  public int updateStudent(Student student) {  return studentMapper.updateById(student);  }   @Override  public Student selectStudentById(Integer studentId) {  return studentMapper.selectById(studentId);  } } |

（5）修改控制类SpringBootController

|  |
| --- |
| @RestController @RequestMapping("/student") public class SpringBootController {   @Resource  private StudentService studentService;   @PostMapping(value = "/add")  public String insertStudent(Student student) {  int flag = studentService.insertStudent(student);  if(flag==1)  return "增加成功！@PostMapping 注解，通常新增时使用";  else  return "增加失败！@PostMapping 注解，通常新增时使用";  }   @DeleteMapping(value = "/delete")  public String deleteStudent(Integer id) {  int flag = studentService.deleteStudent(id);  if(flag==1)  return "删除成功！@DeleteMapping 注解，通常删除数据时使用";  else  return "删除失败！@DeleteMapping 注解，通常删除数据时使用";  }   @PutMapping(value = "/modify")  public Object updateStudent(Student student) {  int flag = studentService.updateStudent(student);  if(flag==1)  return studentService.selectStudentById(student.getId());  else  return "修改失败！@PutMapping 注解，通常更新数据时使用";  }   @GetMapping(value = "/query")  public Object selectStudentByid(Integer id) {  Student student = studentService.selectStudentById(id);  if(student != null)  return student;  else  return "查询失败！@GetMapping 注解,通常查询时使用";  } } |

（6）修改实体类

|  |
| --- |
| @Data @TableName("t\_student") public class Student {  @TableId(value="studentId", type= IdType.*AUTO*)  private Integer id;  // 省略其他属性 } |

（7）Http接口请求工具Postman测试