[**基于SSM的图书管理系统的设计与实现**](https://www.cnblogs.com/ylxb2539989915/p/16329365.html)

专业： 计算机科学与技术 班级： 4班 学号： 222123240421 姓名： 李程飞

## 一、实验目的

使用的SSM框架技术实现人个基于WEB的图书管理系统。

## 二、实验内容

采用Spring整合springmvc+Mybatis框架实现一个图书管理系统。

## 三、实验要求

参照课本及参考程序，快速熟悉相关内容，编写基于MyBatis持久化技术的对象增删改查等操作相关代码，掌握持久层框架技术的具体过程和基本要素，注重良好的代码风格（如变量命名、代码注释、类中代码的结构等），具备编写对象持久化应用程序的能力。

## 四、实验环境

软件环境：Windows10/11，MySQL8.0，Idea2020；

设备名称 DESKTOP-GV3I9FM

处理器 Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz

机带 RAM 8.00 GB

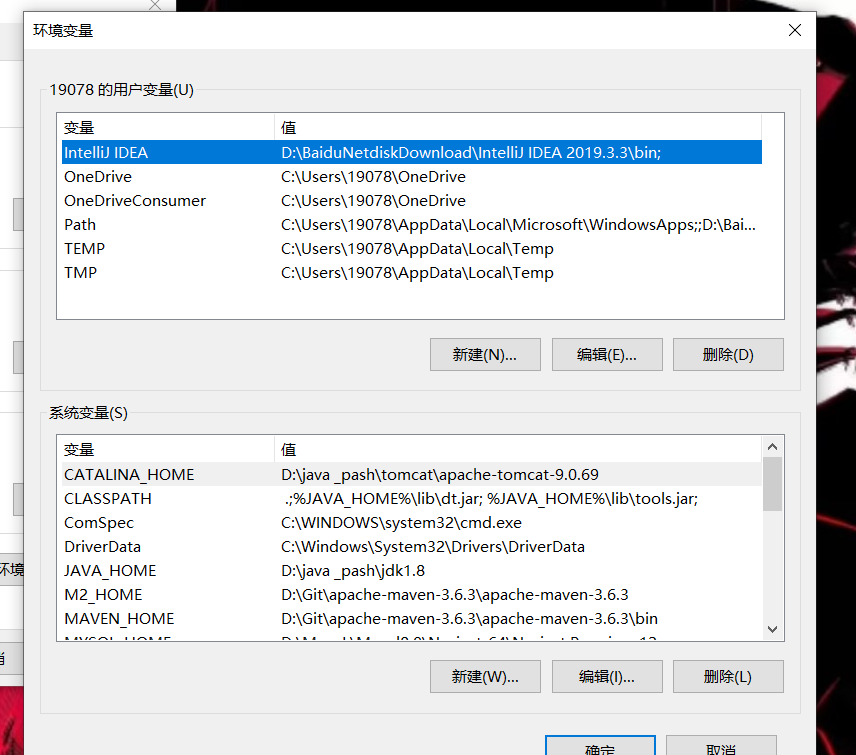
设备 ID 77BD8318-D225-4AB8-A2A7-736F517AED6B

产品 ID 00342-35567-06808-AAOEM

系统类型 64 位操作系统, 基于 x64 的处理器

笔和触控 没有可用于此显示器的笔或触控输入

硬件环境：此处写上你自己的电脑的配置。



## 五、实验步骤

### 一、数据库准备

ssm\_mybatis数据库中的ssm\_2图如图1. 1所示

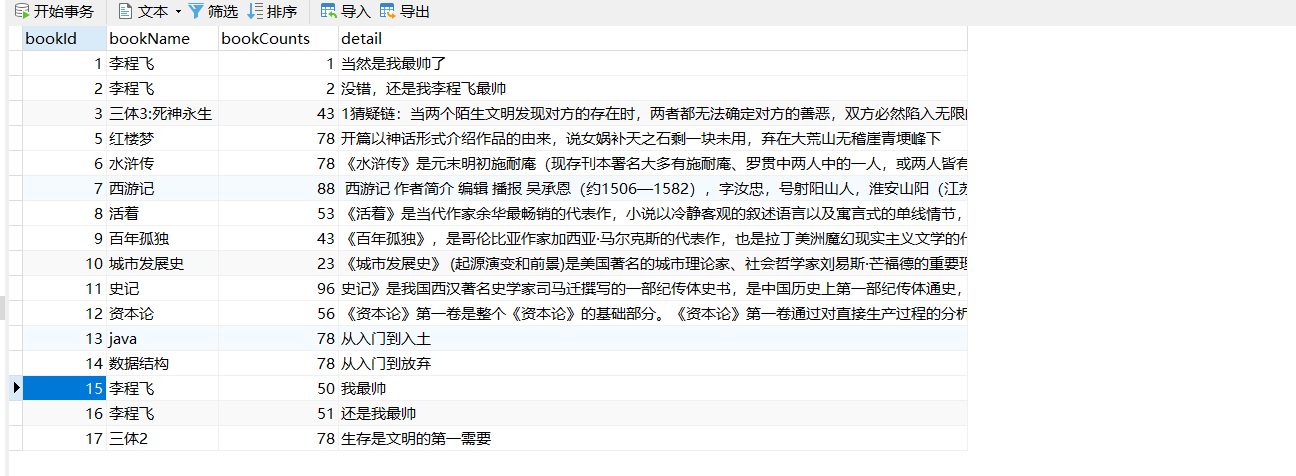
****

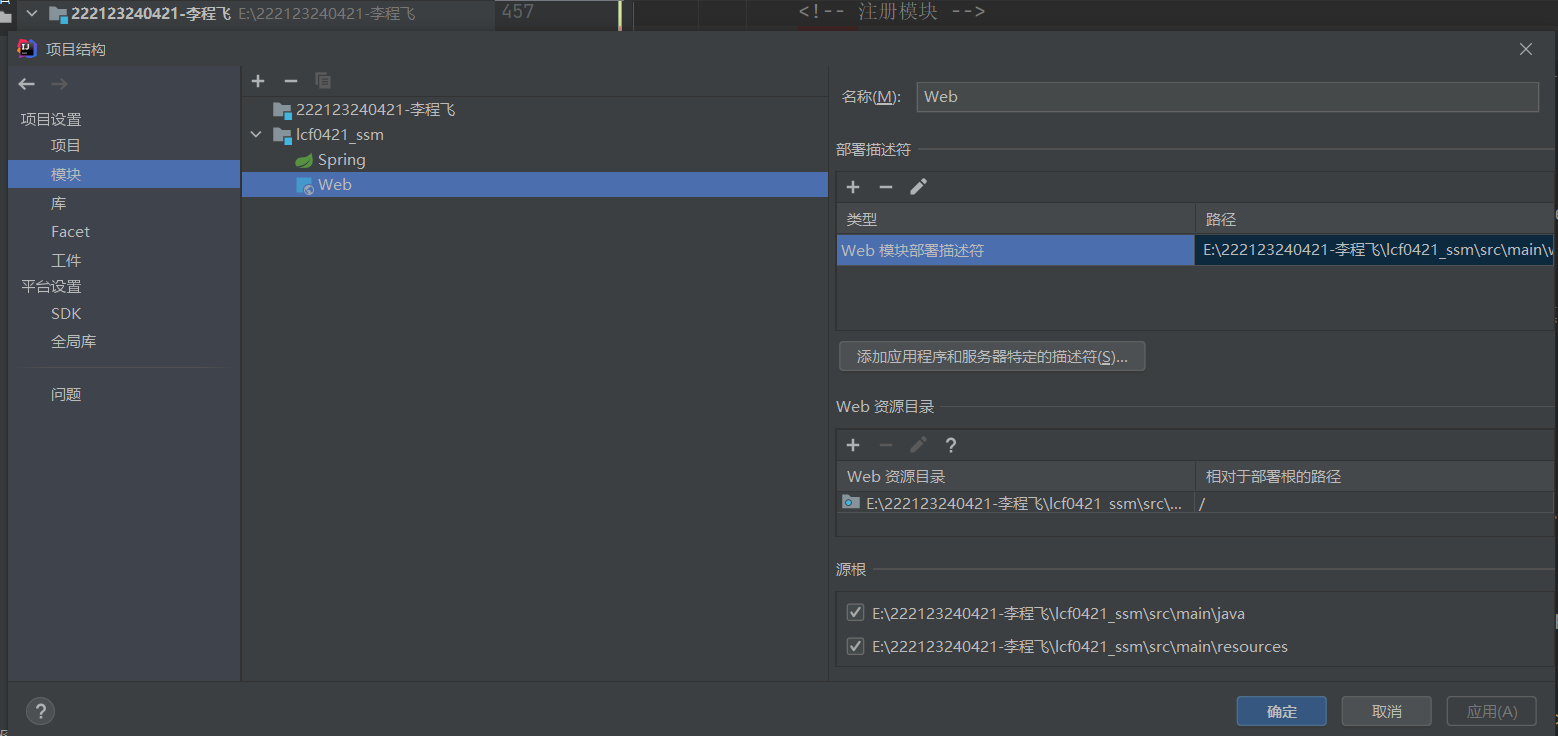
图1. 1

这里写上所使用的数据表等

### 二、创建基于Maven的项目

#### (1) 添加Web支持

**这里注意在pom.xml文件的打包形式，应为war包**

<packaging>war</packaging>

#### (2) 导入依赖

**注意自己的idea的版本，jdk的版本和Tomcat版本，版本太高的话Maven容易冲突**

Pom.xml引入依赖代码1. 1如下所示

|  |
| --- |
| <?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>   <groupId>com.lcf</groupId>  <artifactId>lcf0421\_ssm</artifactId>  <version>1.0-SNAPSHOT</version>  <packaging>war</packaging>  <dependencies>  <!--Junit-->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  <!-- MySQL驱动 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>8.0.16</version>  </dependency>  <!-- 数据库连接池 -->  <dependency>  <groupId>com.mchange</groupId>  <artifactId>c3p0</artifactId>  <version>0.9.5.2</version>  </dependency>   <!--Servlet - JSP -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <version>2.5</version>  </dependency>  <dependency>  <groupId>javax.servlet.jsp</groupId>  <artifactId>jsp-api</artifactId>  <version>2.2</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  <version>1.2</version>  </dependency>  <dependency>  <groupId>taglibs</groupId>  <artifactId>standard</artifactId>  <version>1.1.2</version>  </dependency>   <!--Mybatis-->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.5.2</version>  </dependency>  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>2.0.2</version>  </dependency>   <!--Spring-->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>5.1.9.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>5.1.9.RELEASE</version>  </dependency>   <!--lombok-->  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <version>1.18.12</version>  </dependency>  </dependencies> |
|  |

#### (3)处理Maven静态资源过滤问题

|  |
| --- |
| <!--Maven资源过滤处理--> <build>  <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.xml</include>  </includes>  <filtering>false</filtering>  </resource>  </resources> </build> |

#### (4)配置Web.xml

|  |
| --- |
| <servlet>  <servlet-name>DispatcherServlet</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <!--一定要注意:我们这里加载的是总的配置文件，之前被这里坑了！-->  <param-value>classpath:applicationContext.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup> </servlet> <servlet-mapping>  <servlet-name>DispatcherServlet</servlet-name>  <url-pattern>/</url-pattern> </servlet-mapping>  <!--encodingFilter 乱码过滤器--> <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param> </filter> <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping>  <!--Session过期时间--> <session-config>  <session-timeout>15</session-timeout> </session-config>  <!-- 使用默认servlet处理静态资源请求 --> <!--后缀名为css/gif/jps/png/js的静态资源不拦截--> <servlet-mapping>  <servlet-name>default</servlet-name>  <url-pattern>\*.css</url-pattern>  <url-pattern>\*.js</url-pattern>  <url-pattern>\*.png</url-pattern>  <url-pattern>\*.gif</url-pattern>  <url-pattern>\*.jpg</url-pattern> </servlet-mapping> |

### 构造项目的基本结构

#### 1、整个Mybatis

##### (1).创建Mybatis的核心配置文件mybatis-config.xml

**这里如果自己的数据库中表的属性名与pojo实体类中的属性名不同需要添加以下配置**

**<settings>**

**<!--将下划线映射为驼峰-->**

**<setting name="mapUnderscoreToCamelCase" value="true"/>**

**</settings>**

配置如下

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> <configuration>  <!--日志-->  <settings>  <!--标准的日志工厂STDOUT\_LOGGING-->  <setting name="logImpl" value="STDOUT\_LOGGING"/>  </settings>   <!--别名映射-->  <typeAliases>  <package name="com.lcf.pojo"/>  </typeAliases>   <mappers>  <mapper resource="com/lcf/dao/BooksMapper.xml"/>  </mappers>  </configuration> |
|  |

##### (2).创建属性文件jdbc.properties

**这里要注意两点**

1. **使用mybatis8.0以上的版本必须要设置时区serverTimezone=UTC**
2. **需要在localhost：3306/后边设置自己的数据库**

代码如下

|  |
| --- |
| jdbc.driver=com.mysql.cj.jdbc.Driver # 使用Mybatis8.0+ 必须要设置时区serverTimezone=UTC jdbc.url=jdbc:mysql://localhost:3306/ssm\_2?serverTimezone=UTC jdbc.username=root jdbc.password=1907807382 |

#### 2、整合Spring层

##### (1)Spring整合Mybatis，编写Spring-mapper.xml

**如果需要注意的两点为：**

**一、使用的是Druid连接池，则修改一下的配置**

**<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">**

**<property name="driverClassName" value="${jdbc.driver}"></property>**

**<property name="url" value="${jdbc.url}"></property>**

**<property name="username" value="${jdbc.username}"></property>**

**<property name="password" value="${jdbc.password}"></property>**

**</bean>**

二、若映射文件所在路径和mapper接口所在路径不一致，则需要设置

<property name="mapperLocations" value="classpath:mapper/\*.xml">

代码如下代码1. 3所示

|  |
| --- |
| <!--关联数据库配置文件--> <context:property-placeholder location="classpath:jdbc.properties"/> <!--数据库连接池 C3P0连接池--> <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="${jdbc.driver}"/>  <property name="jdbcUrl" value="${jdbc.url}"/>  <property name="user" value="${jdbc.username}"/>  <property name="password" value="${jdbc.password}"/>   <!-- c3p0连接池的私有属性 -->  <property name="maxPoolSize" value="30"/>  <property name="minPoolSize" value="10"/>  <!-- 关闭连接后不自动commit -->  <property name="autoCommitOnClose" value="false"/>  <!-- 获取连接超时时间 -->  <property name="checkoutTimeout" value="10000"/>  <!-- 当获取连接失败重试次数 -->  <property name="acquireRetryAttempts" value="2"/> </bean>  <!--SQLSessionFactory--> <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  <property name="dataSource" ref="dataSource" />  <!--绑定MyBatis配置文件-->  <property name="configLocation" value="classpath:mybatis-config.xml"/> </bean>  <!--获取SQLSession--> <bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">  <!--SqlSessionTemplate 只能使用构造器注入 因为它没有set方法-->  <constructor-arg index="0" ref="sqlSessionFactory"/> </bean>  <!--配置dao层接口扫描 动态实现Dao层接口注入容器--> <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <!--注入SQLSessionFactory-->  <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>  <!--要扫描的dao包-->  <property name="basePackage" value="com.lcf.dao"/> </bean> |
|  |

##### (2) 配置Spring-service.xml

|  |
| --- |
| <!--配置要扫描的service包--> <context:component-scan base-package="com.lcf.service"/> <!--配置注解支持--> <context:annotation-config/>  <!--将我们所有的业务类，注入Spring，可以通过配置，也可以通过注解--> <bean id="BookServiceImpl" class="com.lcf.service.BooksServiceImpl">  <property name="bookMapper" ref="booksMapper"/> </bean>  <!--配置声明式事务--> <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <!--注入数据源-->  <property name="dataSource" ref="dataSource"/> </bean> |

#### 3、整合Spring-MVC层

|  |
| --- |
| <!--注解驱动--> <mvc:annotation-driven/> <!--静态资源过滤--> <mvc:default-servlet-handler/> <!--扫描包--> <context:component-scan base-package="com.lcf.controller"/>  <!--视图解析器:DispatcherServlet给他的ModelAndView--> <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" id="InternalResourceViewResolver">  <!--前缀-->  <property name="prefix" value="/WEB-INF/jsp/"/>  <!--后缀-->  <property name="suffix" value=".jsp"/> </bean> |

#### 编写总配置文件 applicationContext.xml

**这里一定要注意与web.xml文件中配置总的配置文件，配置的代码如下**

<servlet> <servlet-name>DispatcherServlet</servlet-name> <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class> <init-param> <param-name>contextConfigLocation</param-name> <!--一定要注意:我们这里加载的是总的配置文件，之前被这里坑了！--> <param-value>classpath:applicationContext.xml</param-value> </init-param> <load-on-startup>1</load-on-startup> </servlet>

|  |
| --- |
| <import resource="spring-mapper.xml"/> <import resource="spring-service.xml"/> <import resource="spring-mvc.xml"/> |

ApplicationContext的代码如下

#### 编写pojo层下实体类

**这里实现用Lombok中的@dao等注解**

**如果没有加载这个**<!--lombok-->**依赖,还是要写出它们的有参，无参构造器，和Set、Get方法**

|  |
| --- |
| import lombok.AllArgsConstructor;  import lombok.Data;  import lombok.NoArgsConstructor;  @Data  @AllArgsConstructor  @NoArgsConstructor  public class Books {  private int bookID;  private String bookName;  private int bookCounts;  private String detail;  } |

#### 编写dao层

##### Mapper接口-BooksMapper

|  |
| --- |
| public interface BooksMapper {  //增加一个Book  int addBook(Books book);   //根据id删除一个Book  int deleteBookById(@Param("bookId") int id);   //更新Book  int updateBook(Books books);   //根据id查询,返回一个Book  Books queryBookById(@Param("bookId") int id);   //如果不传参 则查询全部Book,返回list集合 如果传参 则查询具体内容  List<Books> queryAllBook(String bookName); } |

##### 编写BookMapper.xml

MyBatis映射文件用于编写SQL，访问以及操作表中的数据

MyBatis映射文件存放的位置是src/main/resources/mappers目录下

**这里注意两点**

一 、MyBatis获取参数值的两种方式：${}和#{}

1、${}的本质就是字符串拼接，#{}的本质就是占位符赋值

2、 ${}使用字符串拼接的方式拼接sql，若为字符串类型或日期类型的字段进行赋值时，需要手动加单引

二、 MyBatis中可以面向接口操作数据，要保证两个一致：

a>mapper接口的全类名和映射文件的命名空间（namespace）保持一致

b>mapper接口中方法的方法名和映射文件中编写SQL的标签的id属性保持一致

|  |
| --- |
| <!--namespace 命名空间 指向 定义一个Dao接口--> <mapper namespace="com.lcf.dao.BooksMapper">  <!--增加一个Book-->  <insert id="addBook" parameterType="books">  insert into books (bookName,bookCounts,detail)  values (#{bookName},#{bookCounts},#{detail})  </insert>  <!--根据id删除一个Book-->  <delete id="deleteBookById" parameterType="int">  delete from books where bookId=#{bookId}  </delete>  <!--更新一个book-->  <update id="updateBook" parameterType="books">  update books set bookName = #{bookName},bookCounts = #{bookCounts},detail = #{detail}  where bookId = #{bookId}  </update>  <!--根据id查询,返回一个Book-->  <select id="queryBookById" parameterType="int" resultType="com.lcf.pojo.Books">  select \* from books where bookID=#{bookId}  </select>  <!--传参则模糊查询 不传参则查全量数据-->  <select id="queryAllBook" parameterType="string" resultType="books">  select \* from books  <where>  <if test="bookName != null">  bookName like concat('%',#{bookName},'%')  </if>  </where>  </select>  </mapper> |

#### 编写service层

##### Service接口-BooksService

|  |
| --- |
| public interface BooksService {   //增加一个Book  int addBook(Books book);  //根据id删除一个Book  int deleteBookById(int id);  //更新Book  int updateBook(Books books);  //根据id查询,返回一个Book  Books queryBookById(int id);  //如果不传参 则查询全部Book,返回list集合 如果传参 则查询具体内容  List<Books> queryAllBook(String bookName); } |

##### 编写接口实现类 BooksServiceImpl

|  |
| --- |
| public class BooksServiceImpl implements BooksService {  //调用dao层的操作，设置一个set接口，方便Spring管理  private BooksMapper bookMapper;   public void setBookMapper(BooksMapper bookMapper) {  this.bookMapper = bookMapper;  }   @Override  public int addBook(Books book) {  return bookMapper.addBook(book);  }   @Override  public int deleteBookById(int id) {  return bookMapper.deleteBookById(id);  }   @Override  public int updateBook(Books books) {  return bookMapper.updateBook(books);  }   @Override  public Books queryBookById(int id) {  return bookMapper.queryBookById(id);  }   @Override  public List<Books> queryAllBook(String bookName) {  List<Books> books = bookMapper.queryAllBook(bookName);  if (books.size()==0){ //判断是否查询到值 如果没查询到 则查全量  books= bookMapper.queryAllBook(null);  }  return books;  } } |

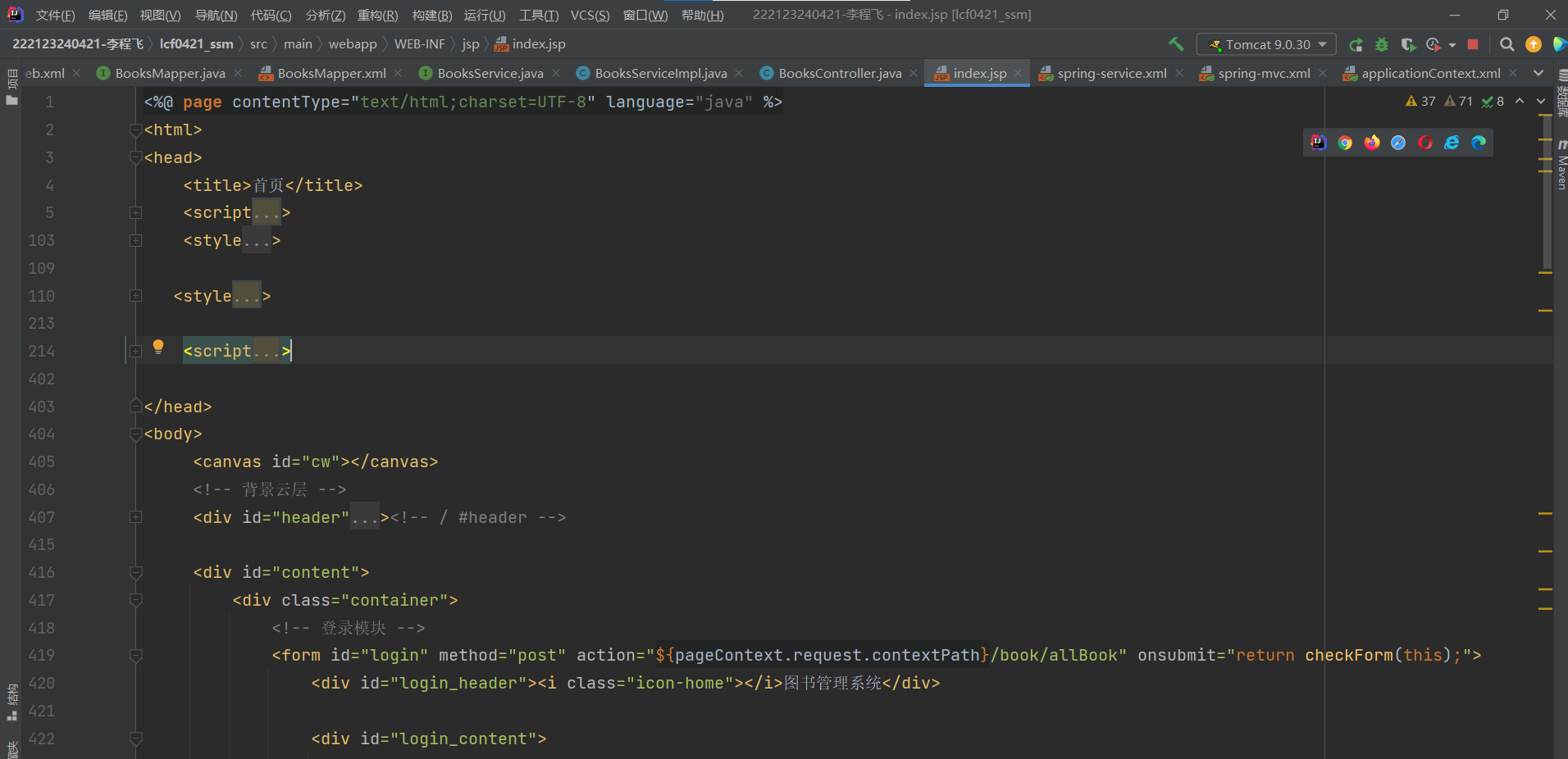
#### 8 、编写Controller层->BooksController类

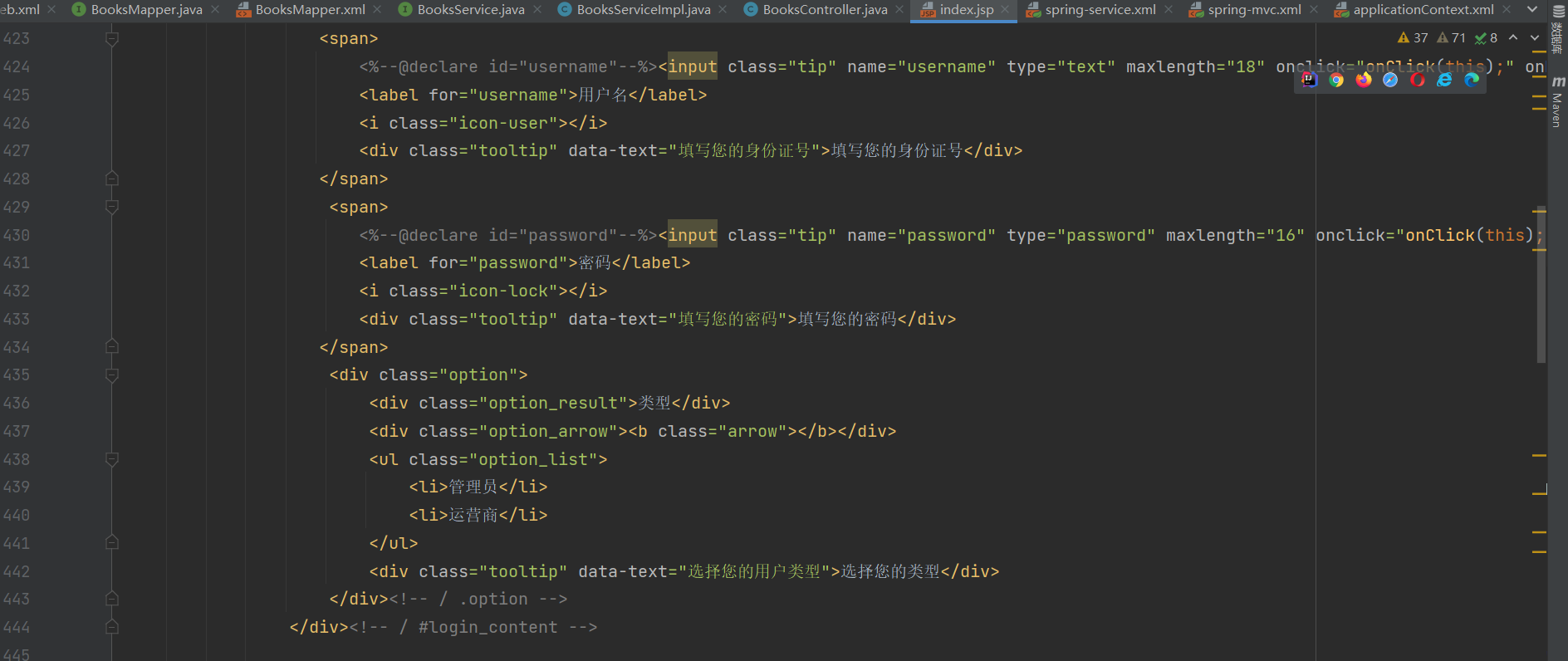
|  |
| --- |
| @Controller @RequestMapping("/book") public class BooksController {  //controller层调service层  @Autowired  private BooksService booksService;  @RequestMapping("/")  public String protal() {  //将逻辑视图返回  return "index";  }  */\*\*  \* 查询全部书籍  \** ***@param*** *model  \** ***@return*** *返回书籍展示页面  \*/* @RequestMapping("/allBook")  public String list(Model model){  List<Books> books = booksService.queryAllBook(null);  model.addAttribute("list",books);  return "allBook";  }  */\*\*  \* 查询指定名字的书籍  \** ***@param*** *queryBookName 要查询的书籍  \** ***@param*** *model 封装查询到的数据  \** ***@return*** *返回页面 注意:此处不能使用重定向  \*/* @RequestMapping("/queryBookName")  public String listall(String queryBookName,Model model){  String trim = queryBookName.trim();  List<Books> books = booksService.queryAllBook(trim);  model.addAttribute("list",books);  return "allBook";  }  */\*\*  \* 新增页面  \** ***@return*** *跳转到新增书籍页面  \*/* @RequestMapping("/toAddBook")  public String toaddBook(Model model){  model.addAttribute("msg","成功跳转页面");  return "addBook";  }  */\*\*  \* 新增书籍数据  \** ***@param*** *books 添加表单值  \** ***@return*** *重定向到查询页  \*/* @RequestMapping("/addBook")  public String addBook(Books books){  booksService.addBook(books);  return "redirect:/book/allBook";  }  */\*\*  \* 基于id查询对应数据 返回修改页面  \** ***@param*** *id 修改数据的id  \** ***@param*** *modle 封装书籍id对应的数据  \** ***@return*** *返回一个修改页面  \*/* @RequestMapping("/toUpdateBook")  public String updateBook(int id,Model modle){  Books books = booksService.queryBookById(id);  modle.addAttribute("Qbooks",books);  return "updateBook";  }  */\*\*  \* 修改表单数据  \** ***@param*** *books 修改数据  \** ***@return*** *重定向到查询页  \*/* @RequestMapping("/updateBook")  public String updateBook(Books books){  int i = booksService.updateBook(books);  return "redirect:/book/allBook";  }  */\*\*  \* 删除书籍  \** ***@param*** *bookId 需要删除的书籍id  \** ***@return*** *重定向到查询页  \*/* @RequestMapping("/del/{bookId}")  public String delBook(@PathVariable int bookId){  booksService.deleteBookById(bookId);  return "redirect:/book/allBook";  } } |

#### 前端页面编写jsp层

##### (1) Index.jsp

这里的代码太多了所以用截图









##### （2）allBooks.jsp

|  |
| --- |
| <div class="row clearfix">  <div class="col-md-12 column">  <div class="page-header">  <h1>  <small>书籍列表 —— 显示所有书籍</small>  </h1>  </div>  </div>  </div>   <div class="row">  <div class="col-md-4 column">  <a class="btn btn-primary" href="${pageContext.request.contextPath}/book/toAddBook">新增</a>  </div>  <div class="col-md-4 column"></div>  <div class="col-md-4 column">  <%--查询书籍--%>  <form class="form-inline" action="${pageContext.request.contextPath}/book/queryBookName" method="post" style="float: right">  <input type="text" name="queryBookName" class="form-control" placeholder="请输入要查询的书籍">  <input type="submit" value="查询" class="btn btn-primary">  </form>  </div>  </div>   <div class="row clearfix">  <div class="col-md-12 column">  <table class="table table-hover table-striped">  <thead>  <tr>  <th>书籍编号</th>  <th>书籍名字</th>  <th>书籍数量</th>  <th>书籍详情</th>  <th>操作</th>  </tr>  </thead>   <tbody>  <**c:forEach** var="book" items="${list}">  <tr>  <td>${book.getBookId()}</td>  <td>${book.getBookName()}</td>  <td>${book.getBookCounts()}</td>  <td>${book.getDetail()}</td>  <td>  <a href="${pageContext.request.contextPath}/book/toUpdateBook?id=${book.getBookId()}">更改</a> |  <a href="${pageContext.request.contextPath}/book/del/${book.getBookId()}">删除</a>  </td>  </tr>  </**c:forEach**>  </tbody>  </table>  </div>  </div>  </div> |

##### （3）addBook.sjp

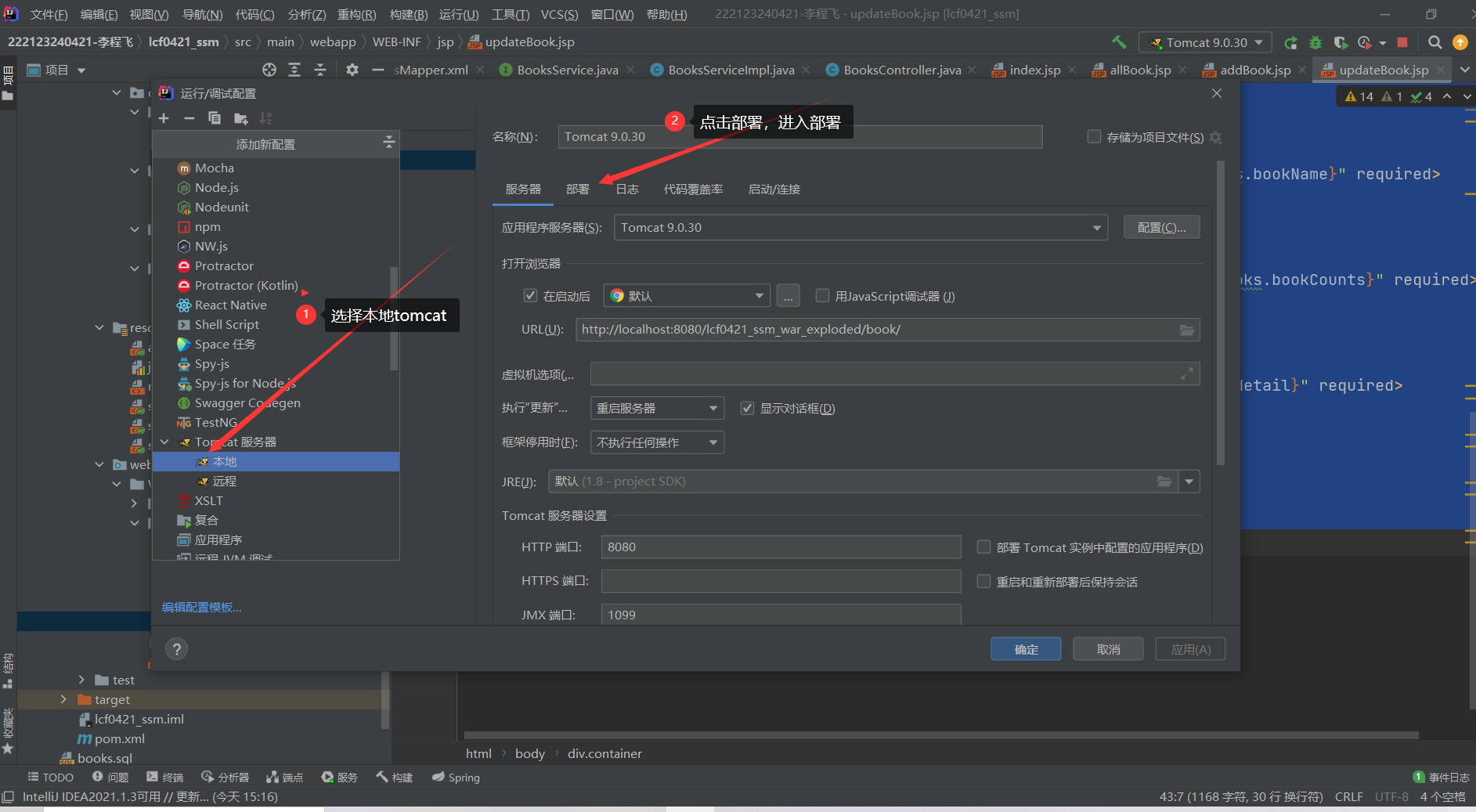
|  |
| --- |
| <div class="row clearfix">  <div class="col-md-12 column">  <div class="page-header">  <h1>  <small>新增书籍</small>  </h1>  </div>  </div>  </div>  <form action="${pageContext.request.contextPath}/book/addBook" method="post">  <div class="form-group">  <lable>书籍名称：</lable>  <input type="text" name="bookName" class="form-control" required>  </div>  <div class="form-group">  <lable>书籍数量：</lable>  <input type="text" name="bookCounts" class="form-control" required>  </div>  <div class="form-group">  <lable>书籍详情：</lable>  <input type="text" name="detail" class="form-control" required>  </div>  <div class="form-group">  <input type="submit" value="添加">  </div>  </form> </div> |

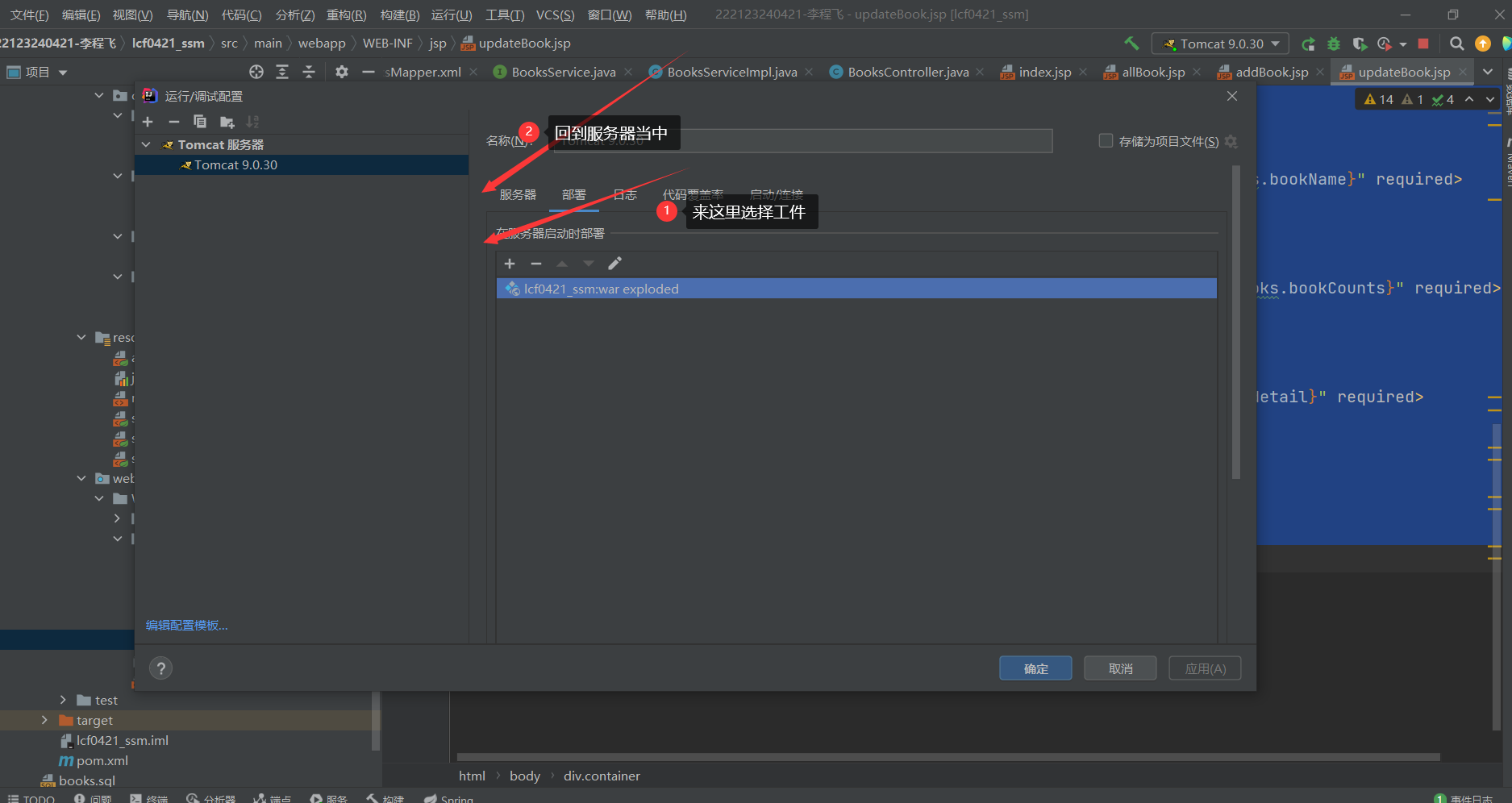
##### (4)Update.jsp

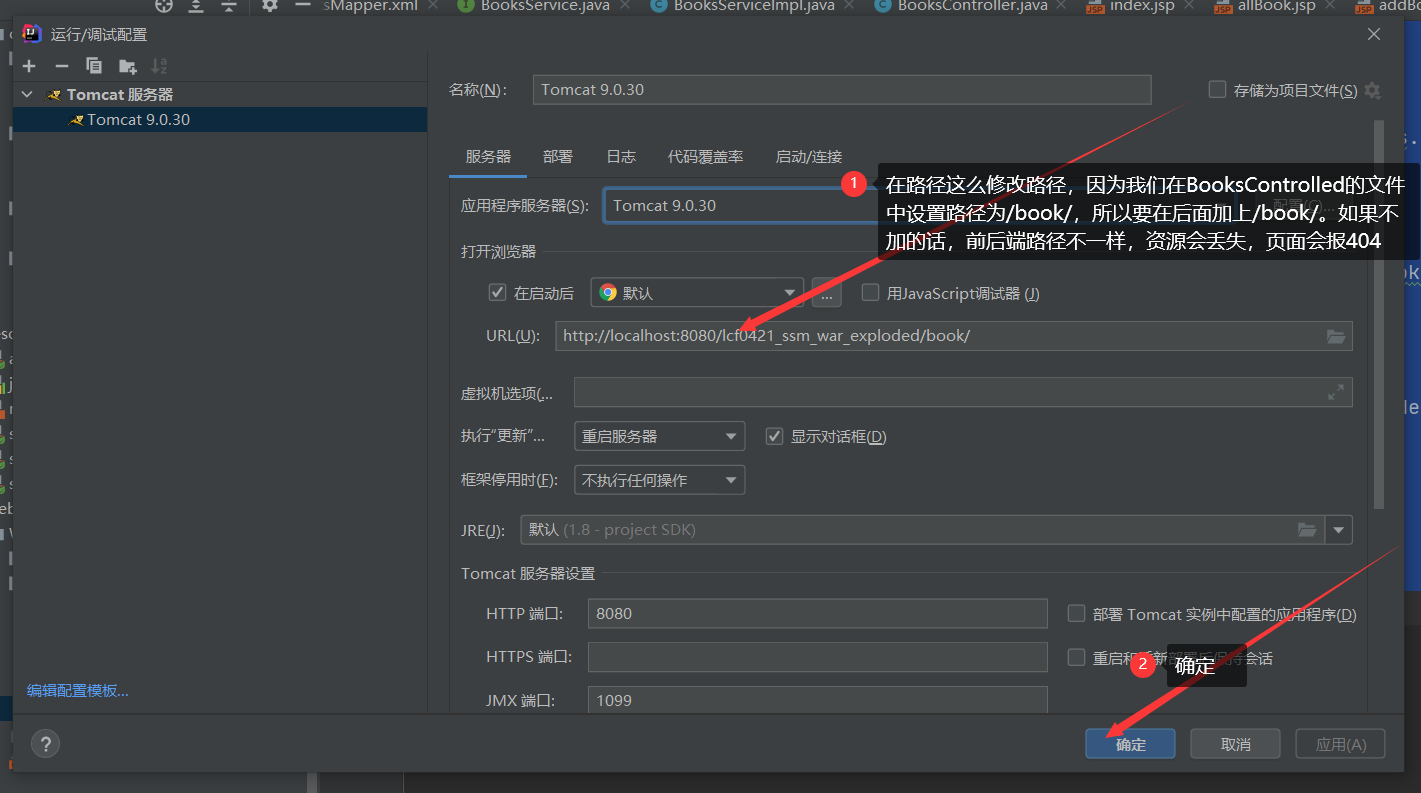
|  |
| --- |
| <div class="row clearfix">  <div class="col-md-12 column">  <div class="page-header">  <h1>  <small>修改书籍</small>  </h1>  </div>  </div>  </div>  <form action="${pageContext.request.contextPath}/book/updateBook" method="post">  <div class="form-group">  <lable>书籍id：</lable>  <input type="text" name="bookId" class="form-control" value="${Qbooks.bookId}" readonly>  </div>  <div class="form-group">  <lable>书籍名称：</lable>  <input type="text" name="bookName" class="form-control" value="${Qbooks.bookName}" required>  </div>  <div class="form-group">  <lable>书籍数量：</lable>  <input type="text" name="bookCounts" class="form-control" value="${Qbooks.bookCounts}" required>  </div>  <div class="form-group">  <lable>书籍详情：</lable>  <input type="text" name="detail" class="form-control" value="${Qbooks.detail}" required>  </div>  <div class="form-group">  <input type="submit" value="修改">  </div>  </form> </div> |

### 项目发布

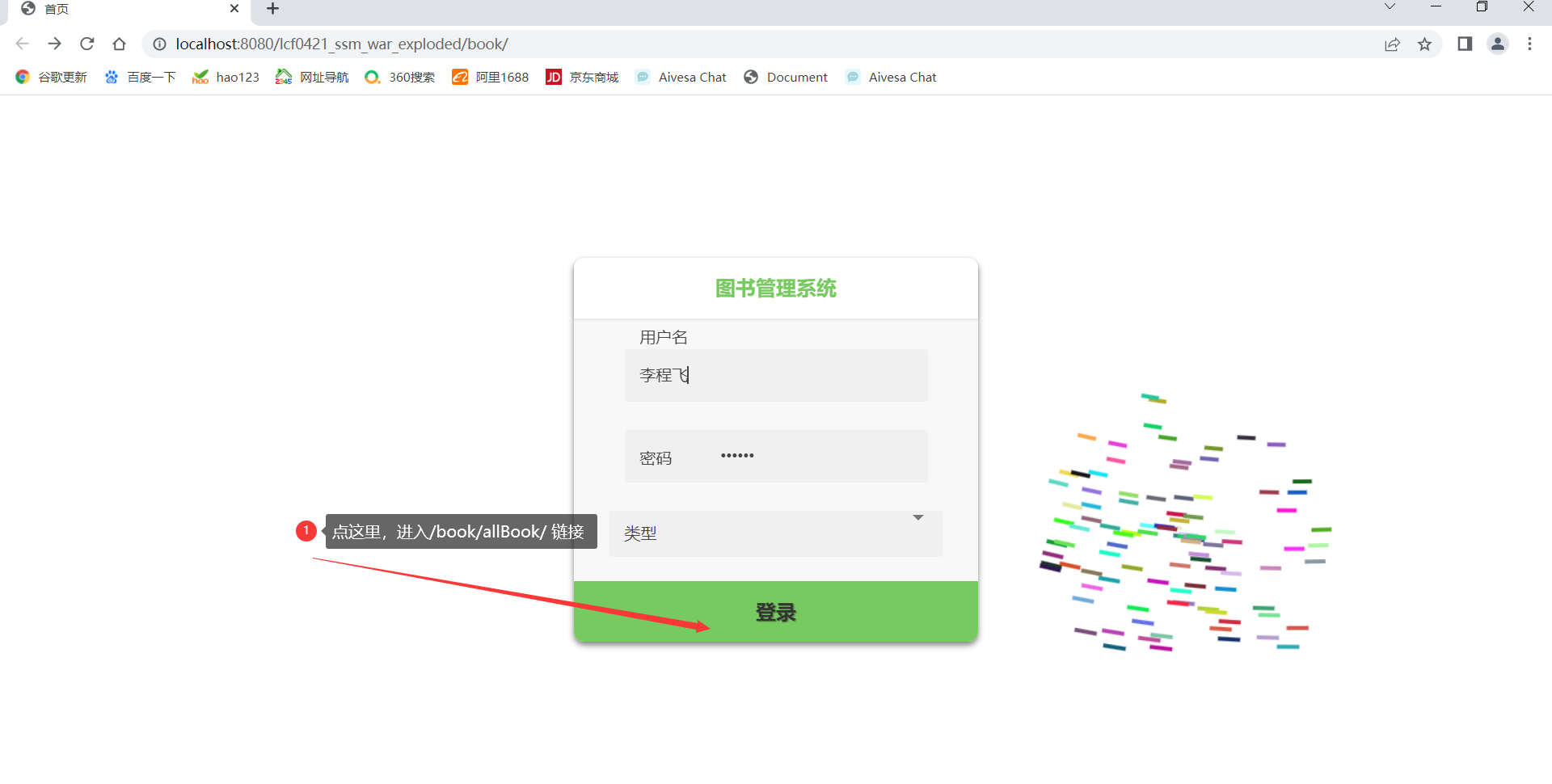
#### 配置Tomcat

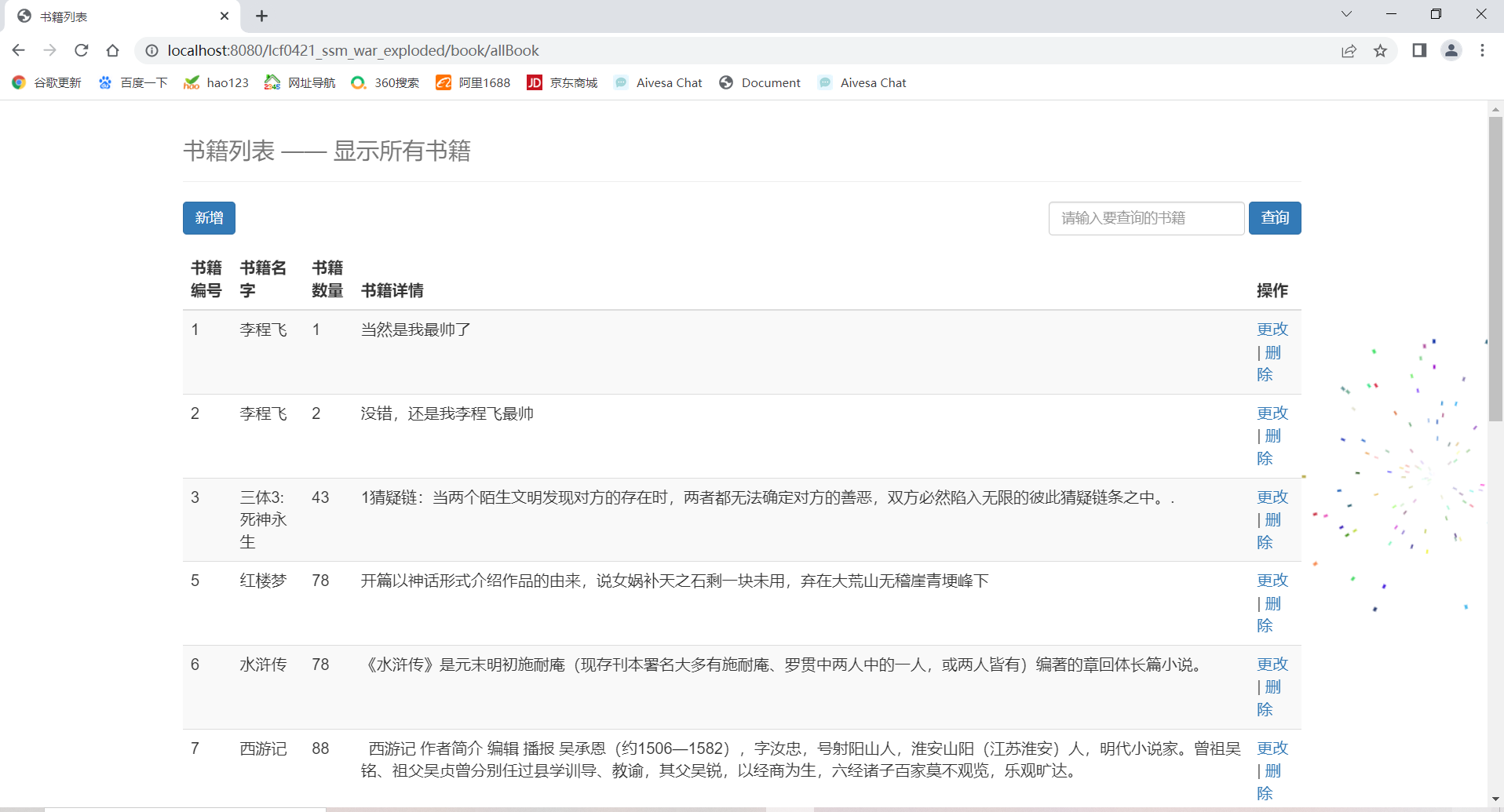


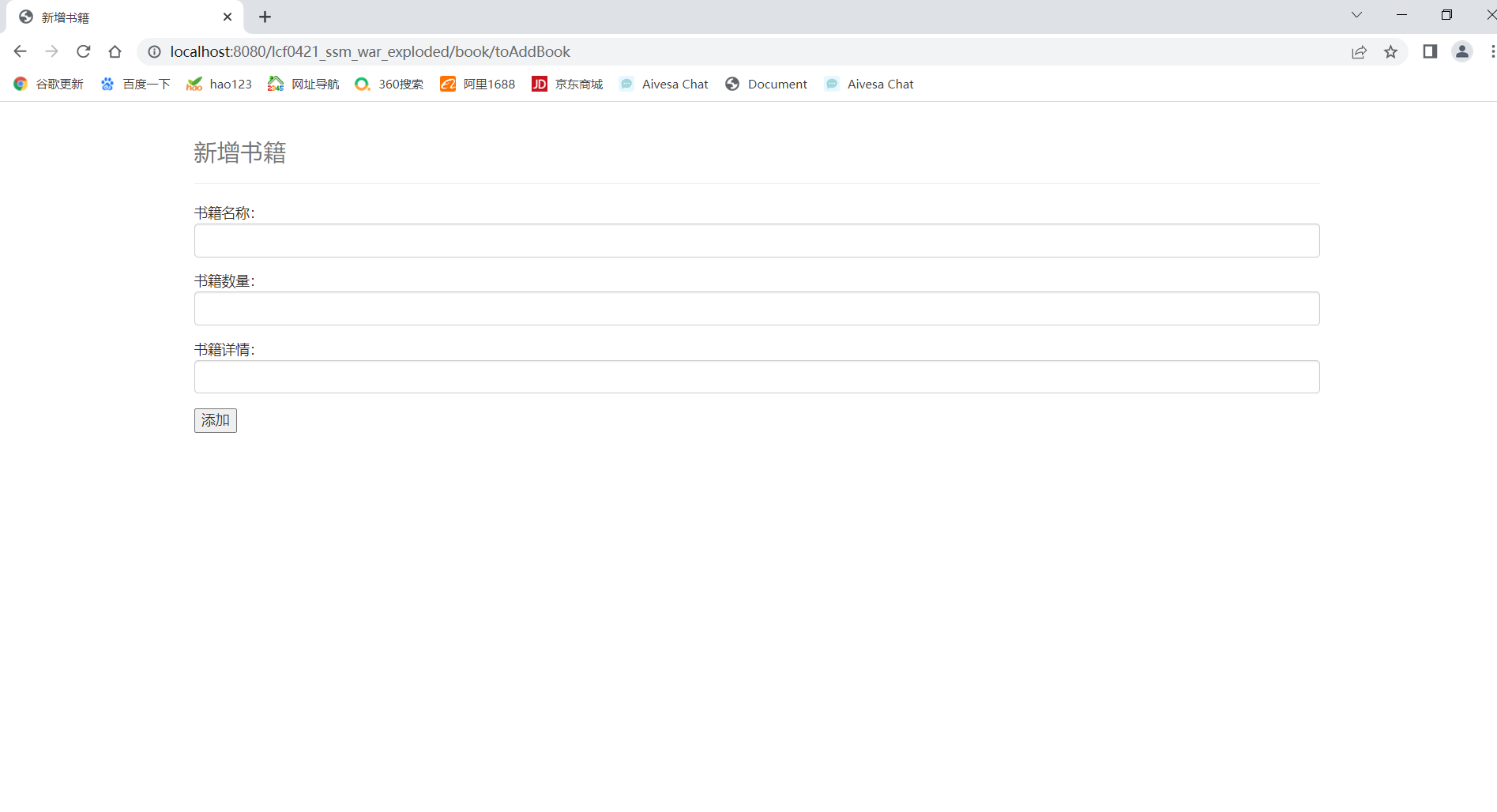


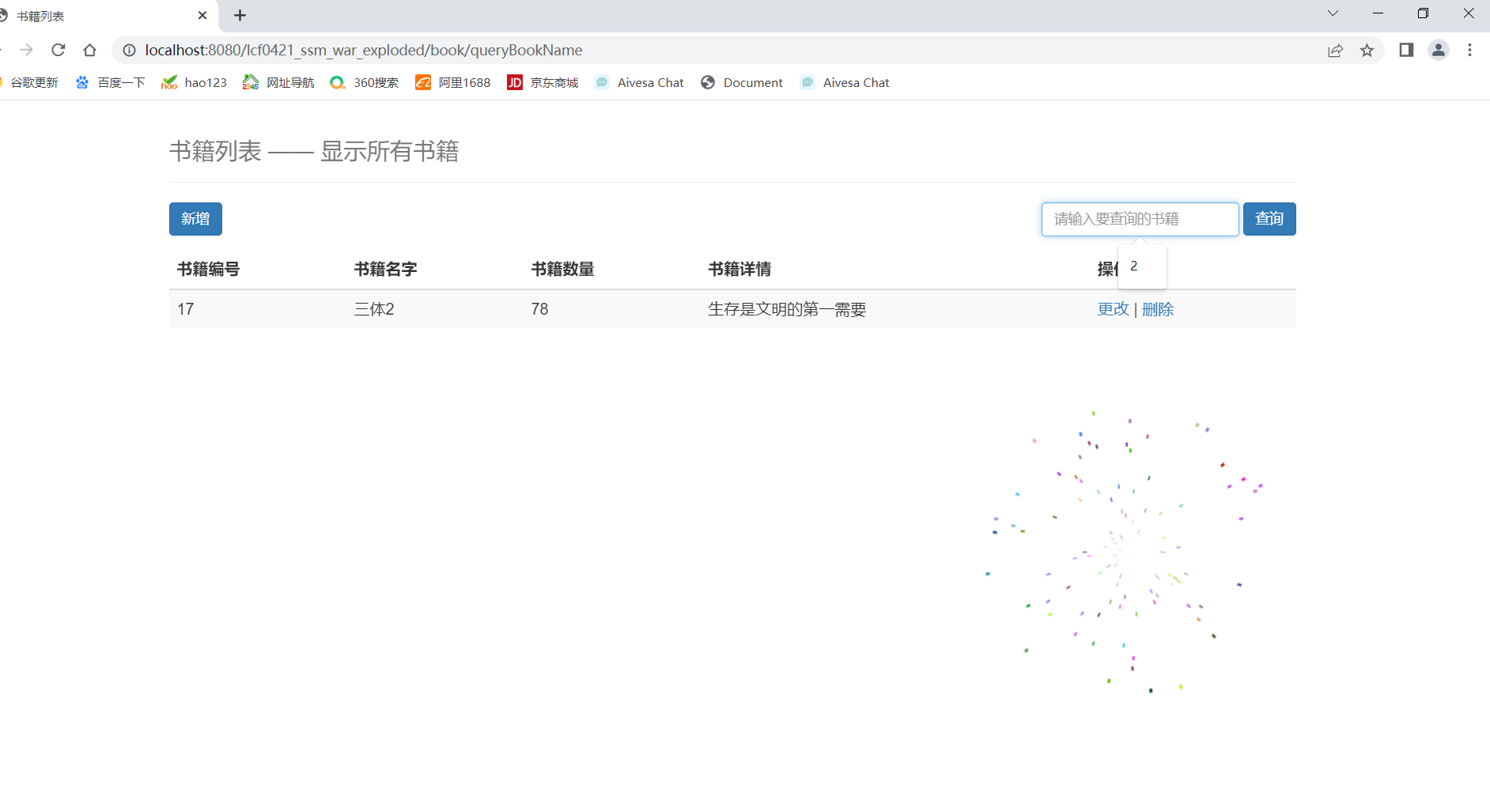


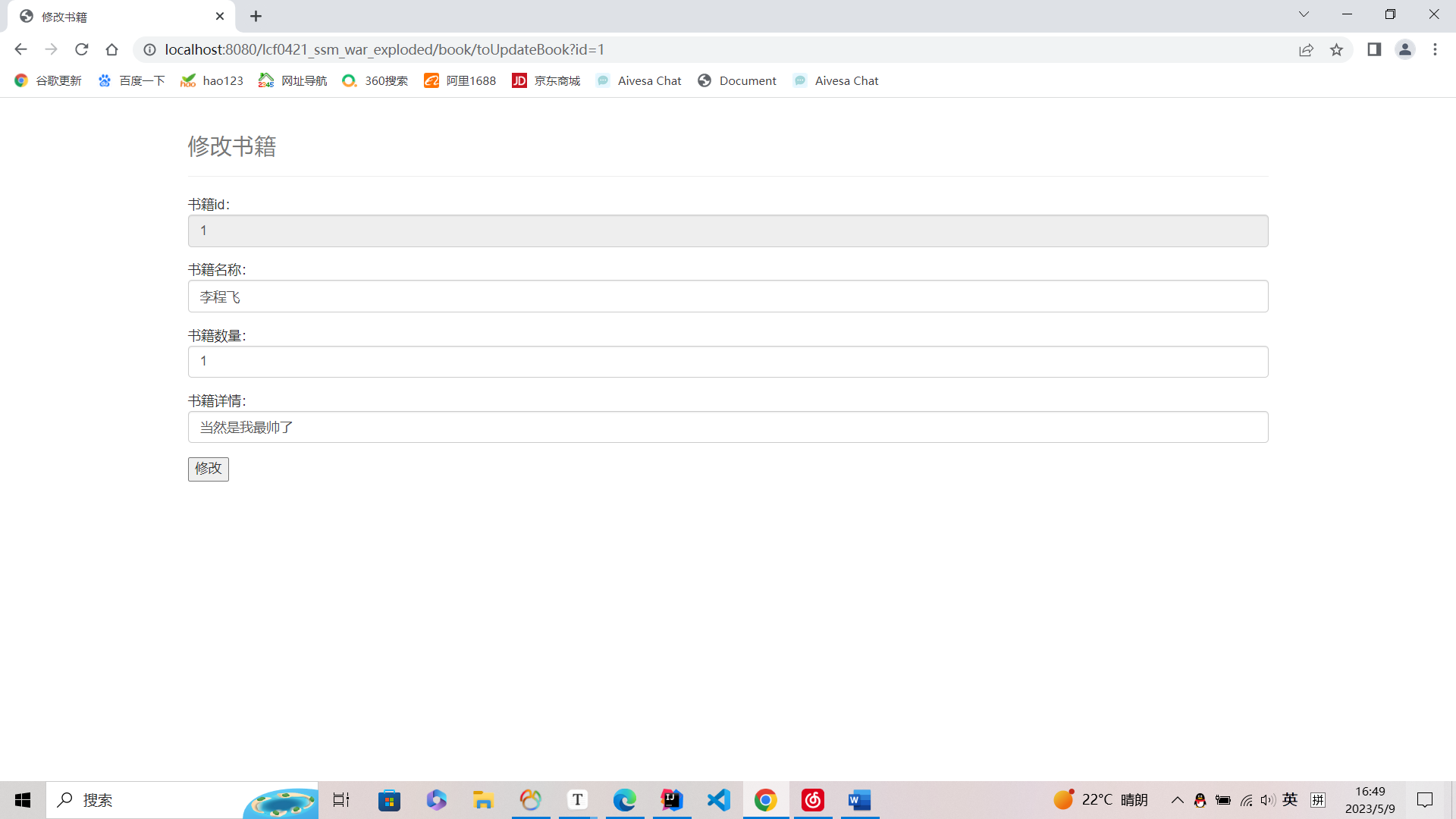
#### 运行整个项目











## 六、实验总结与心得

1、、SpringMVC的执行流程为

1) 用户向服务器发送请求，请求被SpringMVC 前端控制器 DispatcherServlet捕获。

2) DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI），判断请求URI对应的映射：

a) 不存在

i. 再判断是否配置了mvc:default-servlet-handler

ii. 如果没配置，则控制台报映射查找不到，客户端展示404错误



iii. 如果有配置，则访问目标资源（一般为静态资源，如：JS,CSS,HTML），找不到客户端也会展示404

b) 存在则执行下面的流程

3) 根据该URI，调用HandlerMapping获得该Handler(这个时候为控制器的控制方法)配置的所有相关的对象（包括Handler对象以及

Handler对象对应的拦截器），最后以HandlerExecutionChain执行链对象的形式返回。

4) DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。

5) 如果成功获得HandlerAdapter，此时将开始执行拦截器的preHandler(…)方法【正向】

6) 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)方法，处理请求。

在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

a) HttpMessageConveter(报文信息转换器)： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定

的响应信息

b) 数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

c) 数据格式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

d) 数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

7) Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象。

8) 此时将开始执行拦截器的postHandle(...)方法【逆向】。

9) 根据返回的ModelAndView（此时会判断是否存在异常：如果存在异常，则执行

HandlerExceptionResolver进行异常处理）选择一个适合的ViewResolver进行视图解析，根据Model

和View，来渲染视图。

10) 渲染视图完毕执行拦截器的afterCompletion(…)方法【逆向】。

11) 将渲染结果返回给客户端。

1. 在配置tomcat中一定要注意路径的问题，如果配置的路径不同则会使前后段搭不起来，资源会丢失，报404的错误。
2. 如果在运行当中报505，这要注意是不是resource文件下Spring配置文件的名字与SpringMVC中的名字是否一致。
3. 在配置jdbc.properties中要注意使用mybatis8.0以上的版本必须要设置时区serverTimezone=UTC，还有需要在localhost：3306/后边设置自己的数据库
4. 浏览器发送请求，若请求地址符合前端控制器的url-pattern，该请求就会被前端控制器DispatcherServlet处理。前端控制器会读取SpringMVC的核心配置文件，通过扫描组件找到控制器，
5. 将请求地址和控制器中@RequestMapping注解的value属性值进行匹配，若匹配成功，该注解所标识的控制器方法就是处理请求的方法。处理请求的方法需要返回一个字符串类型的视图名称，该视图名称会被视图解析器解析，加上前缀和后缀组成视图的路径，通过Thymeleaf对视图进行渲染，最终转发到视图所对应页面。
6. Spring提供了监听器ContextLoaderListener，实现ServletContextListener接口，可监听ServletContext的状态，在web服务器的启动，读取Spring的配置文件，创建Spring的IOC容器。
7. web应用中必须在web.xml中配置tomcat服务器启动—>监听器—>diapatcherservlet初始化;diapatcherservlet初始化时创建springmvc的ioc，所以在监听器中创建spring的ioc容器，由于springmvc的controller依赖于spring的service，所以spring的ioc要早于springmvc的ioc创建

* DispatcherServlet：**前端控制器**，不需要工程师开发，由框架提供作用：统一处理请求和响应，整个流程控制的中心，由它调用其它组件处理用户的请求HandlerMapping：**处理器映射器**，不需要工程师开发，由框架提供作用：根据请求的url、method等信息查找Handler，即控制器方法Handler：
* **处理器**，需要工程师开发作用：在DispatcherServlet的控制下Handler对具体的用户请求进行处理

HandlerAdapter：**处理器适配器**，不需要工程师开发，由框架提供

作用：通过HandlerAdapter对处理器（控制器方法）进行执行

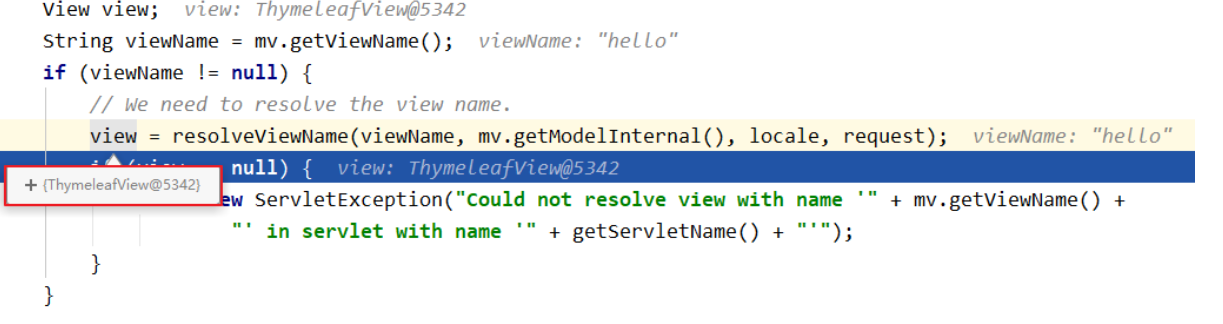
ViewResolver：**视图解析器**，不需要工程师开发，由框架提供

作用：进行视图解析，得到相应的视图，例如：ThymeleafView、InternalResourceView、

RedirectView

View：**视图**

作用：将模型数据通过页面展示给用户

1. 当控制器方法中所设置的视图名称没有任何前缀时，此时的视图名称会被SpringMVC配置文件中所配置的视图解析器解析，视图名称拼接视图前缀和视图
2. 

使用了axios发送ajax请求之后，浏览器发送到服务器的请求参数有两种格式：

1、name=value&name=value...，此时的请求参数可以通过request.getParameter()获取，对应

SpringMVC中，可以直接通过控制器方法的形参获取此类请求参数

2、{key:value,key:value,...}，此时无法通过request.getParameter()获取，之前我们使用操作

json的相关jar包gson或jackson处理此类请求参数，可以将其转换为指定的实体类对象或map集

合。在SpringMVC中，直接使用@RequestBody注解标识控制器方法的形参即可将此类请求参数

转换为java对象

