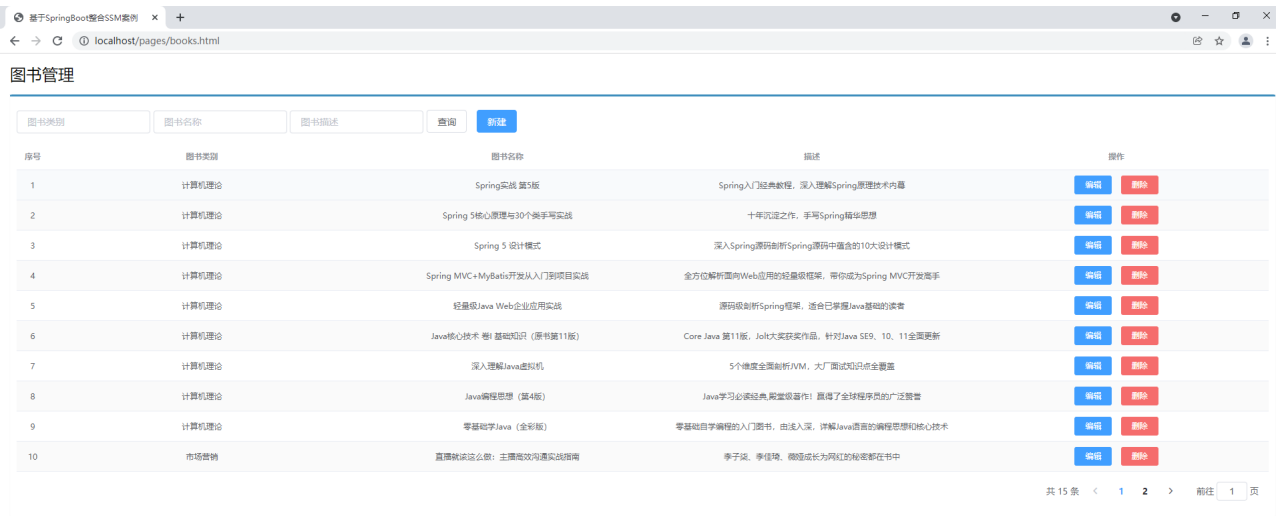


# JC-3-5.SSMP整合综合案例

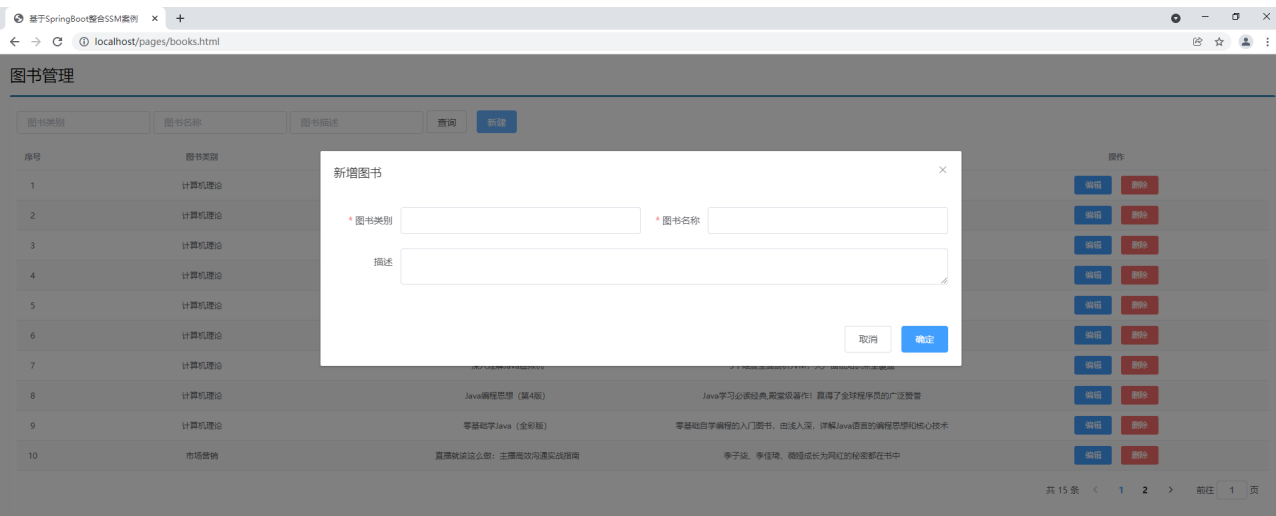
SpringBoot能够整合的技术太多太多了，对于初学者来说慢慢来，一点点掌握。下面就通过一个稍微综合一点的案例，将所有知识贯穿起来，同时做一个小功能，体会一下。不过有言在先，这个案例制作的时候，你可能会有这种感觉，说好的SpringBoot整合其他技术的案例，为什么感觉SpringBoot整合其他技术的身影不多呢？因为这东西书写太简单了，简单到瞬间写完，大量的时间做的不是这些整合工作。

先看一下这个案例的最终效果

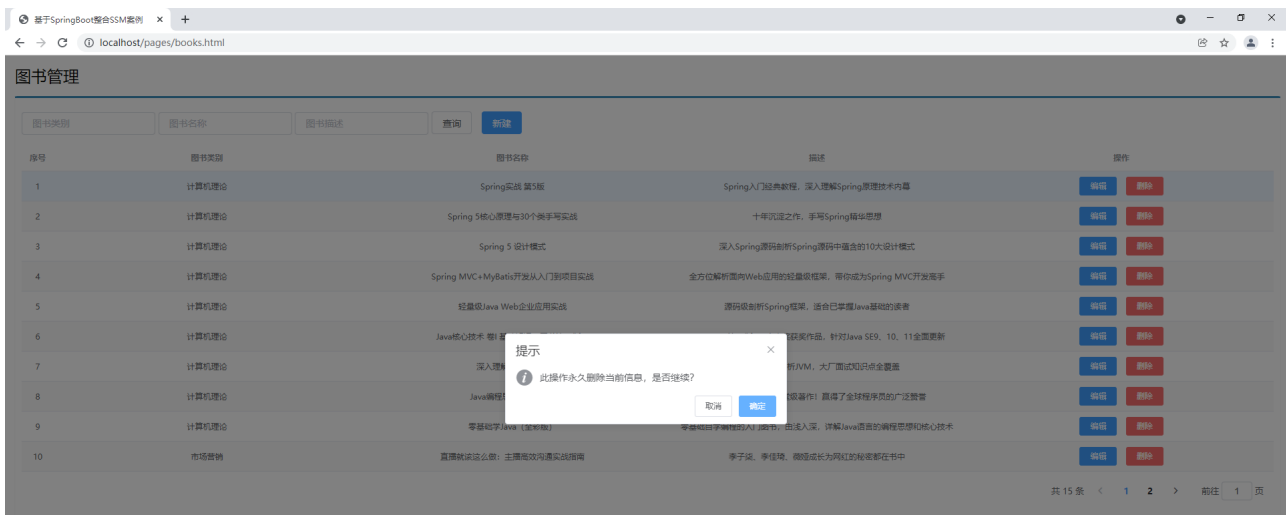
## 主页面



## 添加



## 删除



修改

分页



条件查询



整体案例中需要采用的技术如下，先了解一下，做到哪一个说哪一个

- 1. 实体类开发———使用Lombok快速制作实体类
- 2. Dao开发———整合MyBatisPlus，制作数据层测试
- 3. Service开发———基于MyBatisPlus进行增量开发，制作业务层测试类
- 4. Controller开发———基于Restful开发，使用PostMan测试接口功能

5. Controller开发———前后端开发协议制作

6. 页面开发———基于VUE+ElementUI制作，前后端联调，页面数据处理，页面消息处理

- 列表
- 新增
- 修改
- 删除
- 分页
- 查询

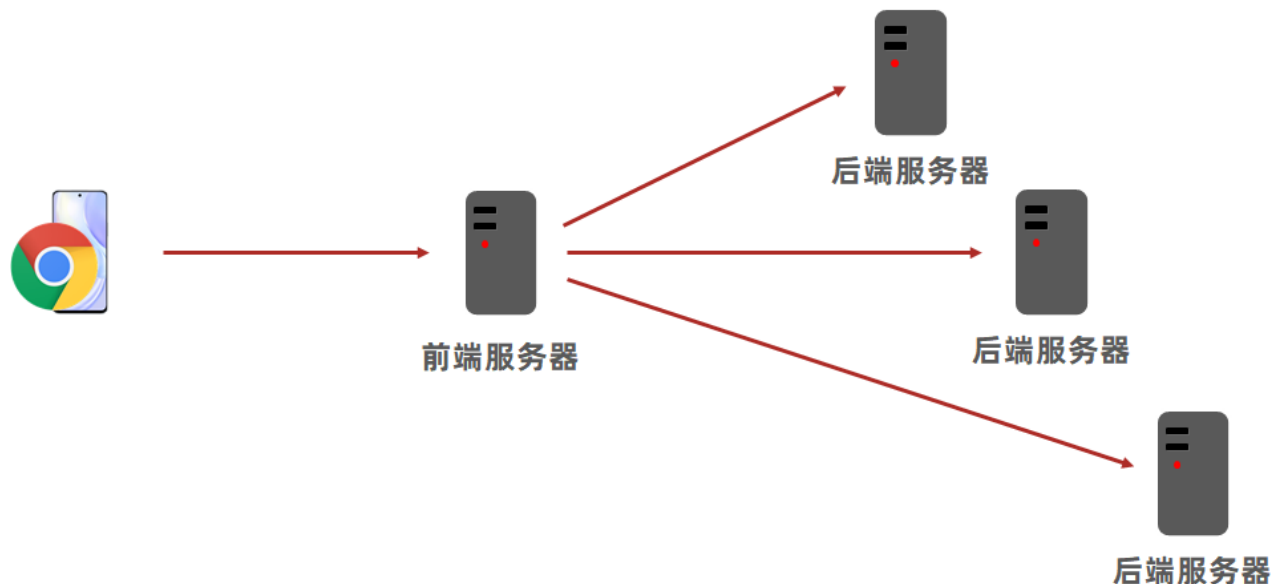
7. 项目异常处理

8. 按条件查询———页面功能调整、Controller修正功能、Service修正功能

可以看的出来，东西还是很多的，希望通过这个案例，各位小伙伴能够完成基础开发的技能训练。整体开发过程采用做一层测一层的形式进行，过程完整，战线较长，希望各位能跟进度，完成这个小案例的制作。

## 0.模块创建

对于这个案例如果按照企业开发的形式进行应该制作后台微服务，前后端分离的开发。



我知道这个对初学的小伙伴要求太高了，咱们简化一下。后台做单体服务器，前端不使用前后端分离的制作了。



一个服务器即充当后台服务调用，又负责前端页面展示，降低学习的门槛。

下面我们就可以创建一个新的模块，加载要使用的技术对应的starter，修改配置文件格式为yml格式，并把web访问端口先设置成80。

### **pom.xml**

```
<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>
```

### **application.yml**

```
server:
  port: 80
```

## 1. 实体类开发

本案例对应的模块表结构如下：

```
-- -----
-- Table structure for tbl_book
-- -----

DROP TABLE IF EXISTS `tbl_book`;
CREATE TABLE `tbl_book` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `type` varchar(20) CHARACTER SET utf8 COLLATE utf8_general_ci
  NULL DEFAULT NULL,
  `name` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci
  NULL DEFAULT NULL,
  `description` varchar(255) CHARACTER SET utf8 COLLATE
  utf8_general_ci NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 51 CHARACTER SET = utf8 COLLATE
= utf8_general_ci ROW_FORMAT = Dynamic;

-- -----
-- Records of tbl_book
-- -----

INSERT INTO `tbl_book` VALUES (1, '计算机理论', 'Spring实战 第5版',
'Spring入门经典教程, 深入理解Spring原理技术内幕');
INSERT INTO `tbl_book` VALUES (2, '计算机理论', 'Spring 5核心原理与30个
类手写实战', '十年沉淀之作, 手写Spring精华思想');
INSERT INTO `tbl_book` VALUES (3, '计算机理论', 'Spring 5 设计模式',
'深入Spring源码剖析Spring源码中蕴含的10大设计模式');
INSERT INTO `tbl_book` VALUES (4, '计算机理论', 'Spring MVC+MyBatis开
发从入门到项目实战', '全方位解析面向Web应用的轻量级框架, 带你成为Spring MVC开发
高手');
INSERT INTO `tbl_book` VALUES (5, '计算机理论', '轻量级Java web企业应用
实战', '源码级剖析Spring框架, 适合已掌握Java基础的读者');
INSERT INTO `tbl_book` VALUES (6, '计算机理论', 'Java核心技术 卷I 基础知
识（原书第11版）', 'Core Java 第11版, Jolt大奖获奖作品, 针对Java SE9、10、
11全面更新');
INSERT INTO `tbl_book` VALUES (7, '计算机理论', '深入理解Java虚拟机', '5
个维度全面剖析JVM, 大厂面试知识点全覆盖');
INSERT INTO `tbl_book` VALUES (8, '计算机理论', 'Java编程思想（第4版）',
'Java学习必读经典, 殿堂级著作! 赢得了全球程序员的广泛赞誉');
INSERT INTO `tbl_book` VALUES (9, '计算机理论', '零基础学Java（全彩
版）', '零基础自学编程的入门图书, 由浅入深, 详解Java语言的编程思想和核心技术');
```

```
INSERT INTO `tbl_book` VALUES (10, '市场营销', '直播就该这么做：主播高效沟通实战指南', '李子柒、李佳琦、薇娅成长为网红的秘密都在书中');  
INSERT INTO `tbl_book` VALUES (11, '市场营销', '直播销讲实战一本通', '和秋叶一起学系列网络营销书籍');  
INSERT INTO `tbl_book` VALUES (12, '市场营销', '直播带货：淘宝、天猫直播从新手到高手', '一本教你如何玩转直播的书，10堂课轻松实现带货月入3w+');
```

根据上述表结构，制作对应的实体类

实体类

```
public class Book {  
    private Integer id;  
    private String type;  
    private String name;  
    private String description;  
}
```

实体类的开发可以自动通过工具手工生成get/set方法，然后覆盖toString()方法，方便调试，等等。不过这一套操作书写很繁琐，有对应的工具可以帮助我们简化开发，介绍一个小工具，lombok。

Lombok，一个Java类库，提供了一组注解，简化POJO实体类开发，SpringBoot目前默认集成了lombok技术，并提供了对应的版本控制，所以只需要提供对应的坐标即可，在pom.xml中添加lombok的坐标。

```
<dependencies>  
    <!--lombok-->  
    <dependency>  
        <groupId>org.projectlombok</groupId>  
        <artifactId>lombok</artifactId>  
    </dependency>  
</dependencies>
```

使用lombok可以通过一个注解@Data完成一个实体类对应的getter，setter，toString，equals，hashCode等操作的快速添加

```
import lombok.Data;
@Data
public class Book {
    private Integer id;
    private String type;
    private String name;
    private String description;
}
```

alt+7: 查看该注解的方法

到这里实体类就做好了，是不是比不使用lombok简化好多，这种工具在Java开发中还有N多，后面课程中遇到了能用的东西时，在不增加各位小伙伴大量的学习时间的情况下，尽量多给大家介绍一些

总结

1. 实体类制作
2. 使用lombok简化开发
  - 导入lombok无需指定版本，由SpringBoot提供版本
  - @Data注解

## 2.数据层开发——基础CRUD

数据层开发本次使用MyBatisPlus技术，数据源使用前面学习的Druid，学都学了都用上

步骤①：导入MyBatisPlus与Druid对应的starter，当然mysql的驱动不能少

```
<dependencies>
  <dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.3</version>
  </dependency>
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.2.6</version>
```

```

</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
</dependencies>

```

步骤②：配置数据库连接相关的数据源配置

```

server:
    port: 80

spring:
    datasource:
        druid:
            driver-class-name: com.mysql.cj.jdbc.Driver
            url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
            username: root
            password: root

```

步骤③：使用MP的标准通用接口BaseMapper加速开发，别忘了@Mapper和泛型的指定

```

@Mapper
public interface BookDao extends BaseMapper<Book> {
}

```

步骤④：制作测试类测试结果，这个测试类制作是个好习惯，不过在企业开发中往往都为加速开发跳过此步，且行且珍惜吧

```

package com.itheima.dao;

import
com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.itheima.domain.Book;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

```



```
@SpringBootTest
public class BookDaoTestCase {

    @Autowired
    private BookDao bookDao;

    @Test
    void testGetById(){
        System.out.println(bookDao.selectById(1));
    }

    @Test
    void testSave(){
        Book book = new Book();
        book.setType("测试数据123");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookDao.insert(book);
    }

    @Test
    void testUpdate(){
        Book book = new Book();
        book.setId(17);
        book.setType("测试数据abcdefg");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookDao.updateById(book);
    }

    @Test
    void testDelete(){
        bookDao.deleteById(16);
    }

    @Test
    void testGetAll(){
        bookDao.selectList(null);
    }
}
```

## 温馨提示

MP技术默认的主键生成策略为雪花算法，生成的主键ID长度较大，和目前的数据库设定规则不相符，需要配置一下使MP使用数据库的主键生成策略，方式嘛还是老一套，做配置。在application.yml中添加对应配置即可，具体如下

```
server:
  port: 80

spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/ssm_db?serverTimezone=UTC
      username: root
      password: root

mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_          #设置表名通用前缀
      id-type: auto               #设置主键id字段的生成策略为参照数据库设定的策略，当前数据库设置id生成策略为自增
```

## 查看MP运行日志

在进行数据层测试的时候，因为基础的CRUD操作均由MP给我们提供了，所以就出现了一个局面，开发者不需要书写SQL语句了，这样程序运行的时候总有一种感觉，一切的一切都是黑盒的，作为开发者我们啥也不知道就完了。如果程序正常运行还好，如果报错了，这个时候就很崩溃，你甚至都不知道从何下手，因为传递参数、封装SQL语句这些操作完全不是你干预开发出来的，所以查看执行期运行的SQL语句就成为当务之急。

SpringBoot整合MP的时候充分考虑到了这点，通过配置的形式就可以查阅执行期SQL语句，配置如下

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
      id-type: auto
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
```

再来看运行结果，此时就显示了运行期执行SQL的情况。

Creating a new SqlSession

SqlSession

[org.apache.ibatis.session.defaults.DefaultSqlSession@2c9a6717] was not registered for synchronization because synchronization is not active

JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@6ca30b8a] will not be managed by Spring

==> Preparing: SELECT id,type,name,description FROM tbl\_book

==> Parameters:

<== Columns: id, type, name, description

<== Row: 1, 计算机理论, Spring实战 第5版, Spring入门经典教程, 深入理解Spring原理技术内幕

<== Row: 2, 计算机理论, Spring 5核心原理与30个类手写实战, 十年沉淀之作, 手写Spring精华思想

<== Row: 3, 计算机理论, Spring 5 设计模式, 深入Spring源码剖析  
Spring源码中蕴含的10大设计模式

<== Row: 4, 计算机理论, Spring MVC+MyBatis开发从入门到项目实战, 全方位解析面向Web应用的轻量级框架, 带你成为Spring MVC开发高手

<== Row: 5, 计算机理论, 轻量级Java Web企业应用实战, 源码级剖析  
Spring框架, 适合已掌握Java基础的读者

<== Row: 6, 计算机理论, Java核心技术 卷I 基础知识（原书第11版）, Core Java 第11版, Jolt大奖获奖作品, 针对Java SE9、10、11全面更新

<== Row: 7, 计算机理论, 深入理解Java虚拟机, 5个维度全面剖析JVM, 大厂面试知识点全覆盖

<== Row: 8, 计算机理论, Java编程思想（第4版）, Java学习必读经典, 殿堂级著作！赢得了全球程序员的广泛赞誉

<== Row: 9, 计算机理论, 零基础学Java（全彩版）, 零基础自学编程的入门图书, 由浅入深, 详解Java语言的编程思想和核心技术

<== Row: 10, 市场营销, 直播就该这么做: 主播高效沟通实战指南, 李子柒、李佳琦、薇娅成长为网红的秘密都在书中

<== Row: 11, 市场营销, 直播销讲实战一本通, 和秋叶一起学系列网络营销书籍

<== Row: 12, 市场营销, 直播带货: 淘宝、天猫直播从新手到高手, 一本教你如何玩转直播的书, 10堂课轻松实现带货月入3W+

<== Row: 13, 测试类型, 测试数据, 测试描述数据

<== Row: 14, 测试数据update, 测试数据update, 测试数据update

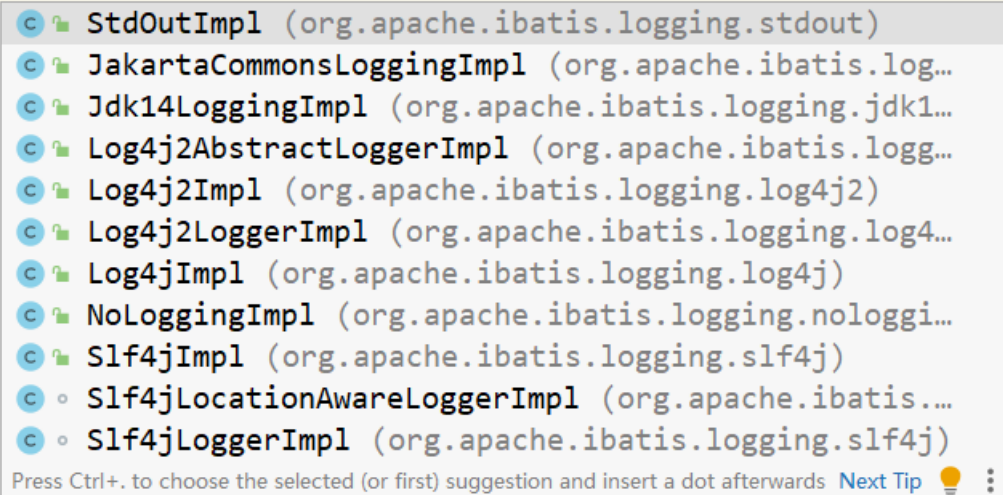
<== Row: 15, -----, 测试数据123, 测试数据123

<== Total: 15

其中清晰的标注了当前执行的SQL语句是什么，携带了什么参数，对应的执行结果是什么，所有信息应有尽有。

此处设置的是日志的显示形式，当前配置的是控制台输出，当然还可以由更多的选择，根据需求切换即可

```
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
  configuration:
    log-impl: |
```



Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards [Next Tip](#)

## 总结

1. 手工导入starter坐标（2个），mysql驱动（1个）
2. 配置数据源与MyBatisPlus对应的配置
3. 开发Dao接口（继承BaseMapper）
4. 制作测试类测试Dao功能是否有效
5. 使用配置方式开启日志，设置日志输出方式为标准输出即可查阅SQL执行日志

## 3.数据层开发——分页功能制作

前面仅仅是使用了MP提供的基础CRUD功能，实际上MP给我们提供了几乎所有的基础操作，这一节说一下如果实现数据库端的分页操作

MP提供的分页操作API如下

```

@Test
void testGetPage(){
    IPage page = new Page(2,5);
    bookDao.selectPage(page, null);
    System.out.println(page.getCurrent());
    System.out.println(page.getSize());
    System.out.println(page.getTotal());
    System.out.println(page.getPages());
    System.out.println(page.getRecords());
}

```

其中selectPage方法需要传入一个封装分页数据的对象，可以通过new的形式创建这个对象，当然这个对象也是MP提供的，别选错包了。创建此对象时就需要指定分页的两个基本数据

- 当前显示第几页
- 每页显示几条数据

可以通过创建Page对象时利用构造方法初始化这两个数据

```

IPage page = new Page(2,5);

```

将该对象传入到查询方法selectPage后，可以得到查询结果，但是我们会发现当前操作查询结果返回值仍然是一个IPage对象，这又是怎么回事？

```

IPage page = bookDao.selectPage(page, null);

```

原来这个IPage对象中封装了若干个数据，而查询的结果作为IPage对象封装的一个数据存在的，可以理解为查询结果得到后，又塞到了这个IPage对象中，其实还是为了高度的封装，一个IPage描述了分页所有的信息。下面5个操作就是IPage对象中封装的所有信息了

```

@Test
void testGetPage(){
    IPage page = new Page(2,5);
    bookDao.selectPage(page, null);
    System.out.println(page.getCurrent());           //当前页码值
    System.out.println(page.getSize());              //每页显示数
    System.out.println(page.getTotal());              //数据总量
    System.out.println(page.getPages());              //总页数
    System.out.println(page.getRecords());            //详细数据
}

```

到这里就知道这些数据如何获取了，但是当你去执行这个操作时，你会发现并不像我们分析的这样，实际上这个分页当前是无效的。为什么这样呢？这个要源于MP的内部机制。

对于MySQL的分页操作使用limit关键字进行，而并不是所有的数据库都使用limit关键字实现的，这个时候MP为了制作的兼容性强，将分页操作设置为基础查询操作的升级版，你可以理解为iPhone6与iPhone6S-PLUS的关系。

基础操作中有查询全部的功能，而在这个基础上只需要升级一下（PLUS）就可以得到分页操作。所以MP将分页操作做成了一个开关，你用分页功能就把开关开启，不用就不需要开启这个开关。而我们现在没有开启这个开关，所以分页操作是没有的。这个开关是通过MP的拦截器的形式存在的，其中的原理这里不分析了，有兴趣的小伙伴可以学习MyBatisPlus这门课程进行详细解读。具体设置方式如下

定义MP拦截器并将其设置为Spring管控的bean

```
@Configuration
public class MPConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        MybatisPlusInterceptor interceptor = new
MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new
PaginationInnerInterceptor());
        return interceptor;
    }
}
```

上述代码第一行是创建MP的拦截器栈，这个时候拦截器栈中没有具体的拦截器，第二行是初始化了分页拦截器，并添加到拦截器栈中。如果后期开发其他功能，需要添加全新的拦截器，按照第二行的格式继续add进去新的拦截器就可以了。

总结

1. 使用IPage封装分页数据
2. 分页操作依赖MyBatisPlus分页拦截器实现功能
3. 借助MyBatisPlus日志查阅执行SQL语句

## 4.数据层开发——条件查询功能制作

除了分页功能，MP还提供有强大的条件查询功能。以往我们写条件查询要自己动态拼写复杂的SQL语句，现在简单了，MP将这些操作都制作成API接口，调用一个又一个的方法就可以实现各种套件的拼装。这里给大家普及一下基本格式，详细的操作还是到MP的课程中查阅吧

下面的操作就是执行一个模糊匹配对应的操作，由like条件书写变为了like方法的调用

```
@Test
void testGetBy(){
    QueryWrapper<Book> qw = new QueryWrapper<>();
    qw.like("name","Spring");
    bookDao.selectList(qw);
}
```

其中第一句QueryWrapper对象是一个用于封装查询条件的对象，该对象可以动态使用API调用的方法添加条件，最终转化成对应的SQL语句。第二句就是一个条件了，需要什么条件，使用QueryWrapper对象直接调用对应操作即可。比如做大于小于关系，就可以使用lt或gt方法，等于使用eq方法，等等，此处不做更多的解释了。

这组API使用还是比较简单的，但是关于属性字段名的书写存在着安全隐患，比如查询字段name，当前是以字符串的形态书写的，万一写错，编译器还没有办法发现，只能将问题抛到运行器通过异常堆栈告诉开发者，不太友好。

MP针对字段检查进行了功能升级，全面支持Lambda表达式，就有了下面这组API。由QueryWrapper对象升级为LambdaQueryWrapper对象，这下就变了上述问题的出现

```
@Test
void testGetBy2(){
    String name = "1";
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
    lqw.like(Book::getName,name);
    bookDao.selectList(lqw);
}
```

为了便于开发者动态拼写SQL，防止将null数据作为条件使用，MP还提供了动态拼装SQL的快捷书写方式



```

@Test
void testGetBy2(){
    String name = "1";
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<Book>();
    //if(name != null) lqw.like(Book::getName,name);           //方式
一： JAVA代码控制
    lqw.like(name != null,Book::getName,name);               //方式
二： API接口提供控制开关
    bookDao.selectList(lqw);
}

```

其实就是个格式，没有区别。关于MP的基础操作就说到这里吧，如果这一块知识不太熟悉的小伙伴还是去完整的学习一下MP的知识吧，这里只是蜻蜓点水的用了几个操作而已。

## 总结

1. 使用QueryWrapper对象封装查询条件
2. 推荐使用LambdaQueryWrapper对象
3. 所有查询操作封装成方法调用
4. 查询条件支持动态条件拼装

## 5.业务层开发

数据层开发告一段落，下面进行业务层开发，其实标准业务层开发很多初学者认为就是调用数据层，怎么说呢？这个理解是没有大问题的，更精准的说法应该是**组织业务逻辑功能，并根据业务需求，对数据持久层发起调用**。有什么差别呢？目标是为了组织出符合需求的业务逻辑功能，至于调不调用数据层还真不好说，有需求就调用，没有需求就不调用。

一个常识性的知识普及一下，业务层的方法名定义一定要与业务有关，例如登录操作

```
login(String username,String password);
```

而数据层的方法名定义一定与业务无关，是一定，不是可能，也不是有可能，例如根据用户名密码查询

```
selectByUsernameAndPassword(String username,String password);
```



我们在开发的时候是可以根据完成的工作不同划分成不同职能的开发团队的。比如一个哥们制作数据层，他就可以不知道业务是什么样子，拿到的需求文档要求可能是这样的

接口：传入用户名与密码字段，查询出对应结果，结果是单条数据

接口：传入ID字段，查询出对应结果，结果是单条数据

接口：传入离职字段，查询出对应结果，结果是多条数据

但是进行业务功能开发的哥们，拿到的需求文档要求差别就很大

接口：传入用户名与密码字段，对用户名字段做长度校验，4-15位，对密码字段做长度校验，8到24位，对喵喵喵字段做特殊字符校验，不允许存在空格，查询结果为对象。如果为null，返回BusinessException，封装消息码INFO\_LOGON\_USERNAME\_PASSWORD\_ERROR

你比较一下，能是一回事吗？差别太大了，所以说业务层方法定义与数据层方法定义差异化很大，只不过有些入门级的开发者手懒或者没有使用过公司相关的ISO标准化文档而已。

多余的话不说了，咱们做案例就简单制作了，业务层接口定义如下：

```
public interface BookService {
    Boolean save(Book book);
    Boolean update(Book book);
    Boolean delete(Integer id);
    Book getById(Integer id);
    List<Book> getAll();
    IPage<Book> getPage(int currentPage, int pageSize);
}
```

业务层实现类如下，转调数据层即可

```
@Service
public class BookServiceImpl implements BookService {

    @Autowired
    private BookDao bookDao;

    @Override
    public Boolean save(Book book) {
        return bookDao.insert(book) > 0;
    }

    @Override
```

```

    public Boolean update(Book book) {
        return bookDao.updateById(book) > 0;
    }

    @Override
    public Boolean delete(Integer id) {
        return bookDao.deleteById(id) > 0;
    }

    @Override
    public Book getById(Integer id) {
        return bookDao.selectById(id);
    }

    @Override
    public List<Book> getAll() {
        return bookDao.selectList(null);
    }

    @Override
    public IPage<Book> getPage(int currentPage, int pageSize) {
        IPage page = new Page(currentPage, pageSize);
        bookDao.selectPage(page, null);
        return page;
    }
}

```

别忘了对业务层接口进行测试，测试类如下

```

@SpringBootTest
public class BookServiceTest {
    @Autowired
    private IBookService bookService;

    @Test
    void testGetById(){
        System.out.println(bookService.getById(4));
    }

    @Test
    void testSave(){
        Book book = new Book();
        book.setType("测试数据123");
    }
}

```

```

        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.save(book);
    }

    @Test
    void testUpdate(){
        Book book = new Book();
        book.setId(17);
        book.setType("-----");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.updateById(book);
    }

    @Test
    void testDelete(){
        bookService.removeById(18);
    }

    @Test
    void testGetAll(){
        bookService.list();
    }

    @Test
    void testGetPage(){
        IPage<Book> page = new Page<Book>(2,5);
        bookService.page(page);
        System.out.println(page.getCurrent());
        System.out.println(page.getSize());
        System.out.println(page.getTotal());
        System.out.println(page.getPages());
        System.out.println(page.getRecords());
    }
}

```

## 总结

1. Service接口名称定义成业务名称，并与Dao接口名称进行区分
2. 制作测试类测试Service功能是否有效

## 业务层快速开发

其实MP技术不仅提供了数据层快速开发方案，业务层MP也给了一个通用接口，个人观点不推荐使用，凑合能用吧，其实就是一个封装+继承的思想，代码给出，实际开发慎用

### 业务层接口快速开发

```
public interface IBookService extends IService<Book> {  
    //添加非通用操作API接口  
}
```

业务层接口实现类快速开发，关注继承的类需要传入两个泛型，一个是数据层接口，另一个是实体类

```
@Service  
public class BookServiceImpl extends ServiceImpl<BookDao, Book>  
implements IBookService {  
    @Autowired  
    private BookDao bookDao;  
    //添加非通用操作API  
}
```

如果感觉MP提供的功能不足以支撑你的使用需要，其实是一定不能支撑的，因为需求不可能是通用的，在原始接口基础上接着定义新的API接口就行了，此处不再说太多了，就是自定义自己的操作了，但是不要和已有的API接口名冲突即可。

## 总结

1. 使用通用接口（IService）快速开发Service
2. 使用通用实现类（ServiceImpl<M,T>）快速开发ServiceImpl
3. 可以在通用接口基础上做功能重载或功能追加
4. 注意重载时不要覆盖原始操作，避免原始提供的功能丢失

## 6.表现层开发

终于做到表现层了，做了这么多都是基础工作。其实你现在回头看看，哪里还有什么SpringBoot的影子？前面1,2步就搞完了。继续完成表现层制作吧，咱们表现层的开发使用基于Restful的表现层接口开发，功能测试通过Postman工具进行

表现层接口如下：

```
@RestController
@RequestMapping("/books")
public class BookController2 {

    @Autowired
    private IBookService bookService;

    @GetMapping
    public List<Book> getAll(){
        return bookService.list();
    }

    @PostMapping
    public Boolean save(@RequestBody Book book){
        return bookService.save(book);
    }

    @PutMapping
    public Boolean update(@RequestBody Book book){
        return bookService.modify(book);
    }

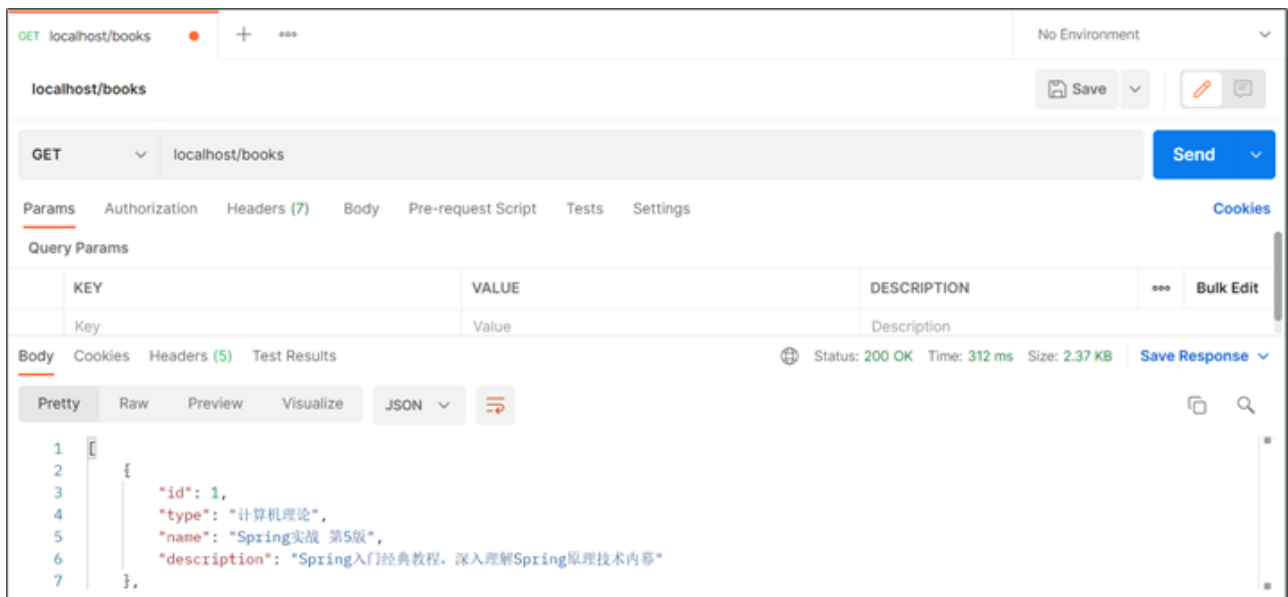
    @DeleteMapping("{id}")
    public Boolean delete(@PathVariable Integer id){
        return bookService.delete(id);
    }

    @GetMapping("{id}")
    public Book getById(@PathVariable Integer id){
        return bookService.getById(id);
    }

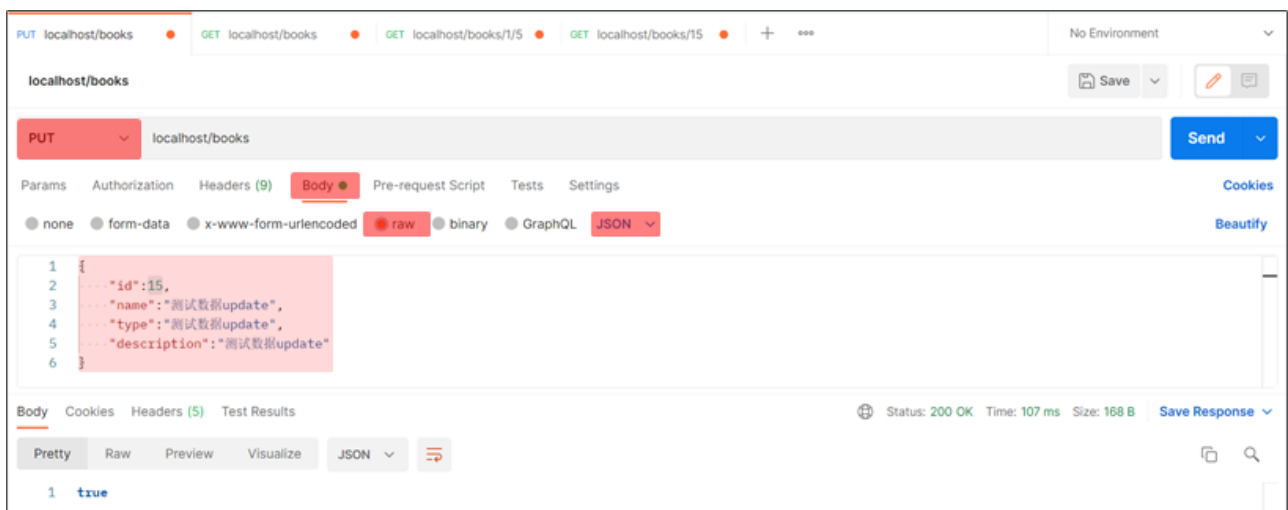
    @GetMapping("{currentPage}/{pageSize}")
    public IPage<Book> getPage(@PathVariable int
currentPage,@PathVariable int pageSize){
        return bookService.getPage(currentPage,pageSize, null);
    }
}
```

在实用Postman测试时关注提交类型，对应上即可，不然就会报405的错误码了

## 普通GET请求



## PUT请求传递json数据，后台实用@RequestBody接收数据



## GET请求传递路径变量，后台实用@PathVariable接收数据

## 总结

### 1. 基于Restful制作表现层接口

- 新增: POST
- 删除: DELETE
- 修改: PUT
- 查询: GET

### 2. 接收参数

- 实体数据: @RequestBody
- 路径变量: @PathVariable

## 7.表现层消息一致性处理

目前我们通过Postman测试后业务层接口功能时通的，但是这样的结果给到前端开发者会出现一个小问题。不同的操作结果所展示的数据格式差异化严重

增删改操作结果

```
true
```

查询单个数据操作结果

```
{
  "id": 1,
  "type": "计算机理论",
  "name": "Spring实战 第5版",
  "description": "Spring入门经典教程"
}
```

查询全部数据操作结果

```
[
  {
    "id": 1,
    "type": "计算机理论",
    "name": "Spring实战 第5版",
    "description": "Spring入门经典教程"
  },
  {
    "id": 2,
    "type": "计算机理论",
    "name": "Spring 5核心原理与30个类手写实战",
    "description": "十年沉淀之作"
  }
]
```

每种不同操作返回的数据格式都不一样，而且还不知道以后还会有什么格式，这样的结果让前端人员看了是很容易让人崩溃的，必须将所有操作的操作结果数据格式统一起来，需要设计表现层返回结果的模型类，用于后端与前端进行数据格式统一，也称为前后端数据协议

```
@Data
public class R {
    private Boolean flag;
    private Object data;
}
```

其中flag用于标识操作是否成功，data用于封装操作数据，现在的数据格式就变了

```
{
    "flag": true,
    "data": {
        "id": 1,
        "type": "计算机理论",
        "name": "Spring实战 第5版",
        "description": "Spring入门经典教程"
    }
}
```

表现层开发格式也需要转换一下

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @PostMapping
    public R save(@RequestBody Book book){
        Boolean flag = bookService.insert(book);
        return new R(flag);
    }
    @PutMapping
    public R update(@RequestBody Book book){
        Boolean flag = bookService.modify(book);
        return new R(flag);
    }
}
```



```

@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @DeleteMapping("/{id}")
    public R delete(@PathVariable Integer id){
        Boolean flag = bookService.delete(id);
        return new R(flag);
    }
    @GetMapping("/{id}")
    public R getById(@PathVariable Integer id){
        Book book = bookService.getById(id);
        return new R(true,book);
    }
}

```

```

@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private IBookService bookService;
    @GetMapping
    public R getAll(){
        List<Book> bookList = bookService.list();
        return new R(true ,bookList);
    }
    @GetMapping("/{currentPage}/{pageSize}")
    public R getAll(@PathVariable Integer currentPage,@PathVariable Integer pageSize){
        IPage<Book> page = bookService.getPage(currentPage, pageSize);
        return new R(true,page);
    }
}

```

结果这么一折腾，全格式统一，现在后端发送给前端的数据格式就统一了，免去了不少前端解析数据的麻烦。

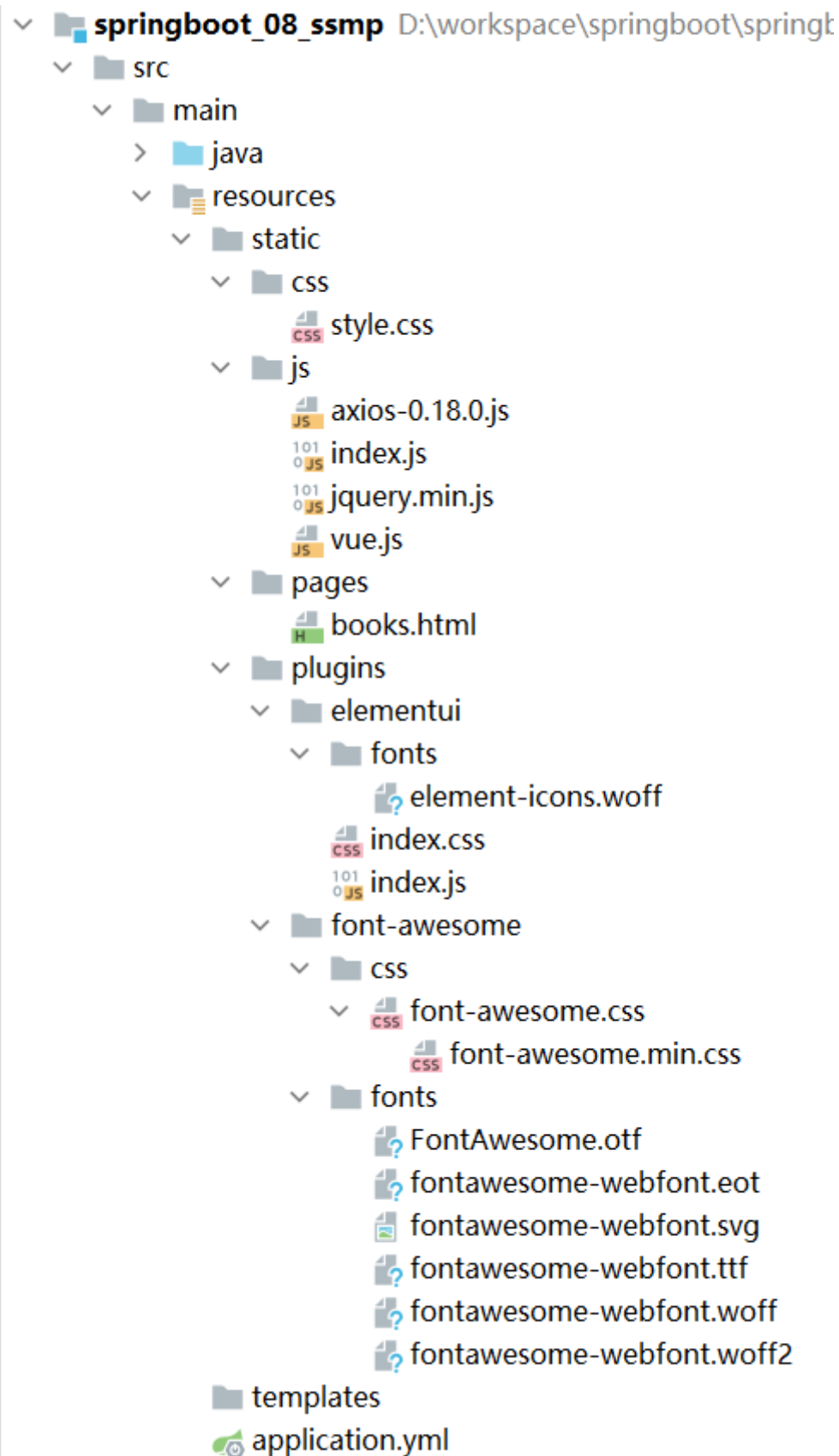
## 总结

1. 设计统一的返回值结果类型便于前端开发读取数据
2. 返回值结果类型可以根据需求自行设定，没有固定格式
3. 返回值结果模型类用于后端与前端进行数据格式统一，也称为前后端数据协议

## 8.前后端联通性测试

后端的表现层接口开发完毕，就可以进行前端的开发了。

将前端人员开发的页面保存到resources目录下的static目录中，建议执行maven的clean生命周期，避免缓存的问题出现。



在进行具体的功能开发之前，先做联通性的测试，通过页面发送异步提交（axios），这一步调试通过后再进行进一步的功能开发

```
//列表
getAll() {
  axios.get("/books").then((res)=>{
    console.log(res.data);
  });
},
```

只要后台代码能够正常工作，前端能够在日志中接收到数据，就证明前后端是通的，也就可以进行下一步的功能开发了

## 总结

1. 单体项目中页面放置在resources/static目录下
2. created钩子函数用于初始化页面时发起调用
3. 页面使用axios发送异步请求获取数据后确认前后端是否联通

## 9. 页面基础功能开发

### F-1. 列表功能（非分页版）

列表功能主要操作就是加载完数据，将数据展示到页面上，此处要利用VUE的数据模型绑定，发送请求得到数据，然后页面上读取指定数据即可

#### 页面数据模型定义

```
data:{
  dataList: [],//当前页要展示的列表数据
  ...
},
```

#### 异步请求获取数据

```
//列表
getAll() {
  axios.get("/books").then((res)=>{
    this.dataList = res.data.data;
  });
},
```

这样在页面加载时就可以获取到数据，并且由VUE将数据展示到页面上了

总结：

1. 将查询数据返回到页面，利用前端数据绑定进行数据展示

## F-2.添加功能

添加功能用于收集数据的表单是通过一个弹窗展示的，因此在添加操作前首先要进行弹窗的展示，添加后隐藏弹窗即可。因为这个弹窗一直存在，因此当页面加载时首先设置这个弹窗为不可显示状态，需要展示，切换状态即可

默认状态

```
data:{
  dialogFormVisible: false,//添加表单是否可见
  ...
},
```

切换为显示状态

```
//弹出添加窗口
handleCreate() {
  this.dialogFormVisible = true;
},
```

由于每次添加数据都是使用同一个弹窗录入数据，所以每次操作的痕迹将在下一次操作时展示出来，需要在每次操作之前清理掉上次操作的痕迹

定义清理数据操作

```
//重置表单
resetForm() {
  this.formData = {};
},
```

切换弹窗状态时清理数据

```
//弹出添加窗口
handleCreate() {
    this.dialogFormVisible = true;
    this.resetForm();
},
```

至此准备工作完成，下面就要调用后台完成添加操作了

添加操作

```
//添加
handleAdd () {
    //发送异步请求
    axios.post("/books",this.formData).then((res)=>{
        //如果操作成功，关闭弹层，显示数据
        if(res.data.flag){
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        }else {
            this.$message.error("添加失败");
        }
    }).finally(()=>{
        this.getAll();
    });
},
```

1. 将要保存的数据传递到后台，通过post请求的第二个参数传递json数据到后台
2. 根据返回的操作结果决定下一步操作
  - 如果是true就关闭添加窗口，显示添加成功的消息
  - 如果是false保留添加窗口，显示添加失败的消息
3. 无论添加是否成功，页面均进行刷新，动态加载数据（对getAll操作发起调用）

取消添加操作

```
//取消
cancel(){
    this.dialogFormVisible = false;
    this.$message.info("操作取消");
},
```

## 总结

1. 请求方式使用POST调用后台对应操作
2. 添加操作结束后动态刷新页面加载数据
3. 根据操作结果不同，显示对应的提示信息
4. 弹出添加Div时清除表单数据

### F-3.删除功能

模仿添加操作制作删除功能，差别之处在于删除操作仅传递一个待删除的数据id到后台即可

删除操作

```
// 删除
handleDelete(row) {
  axios.delete("/books/"+row.id).then((res)=>{
    if(res.data.flag){
      this.$message.success("删除成功");
    }else{
      this.$message.error("删除失败");
    }
  }).finally(()=>{
    this.getAll();
  });
},
```

删除操作提示信息

```
// 删除
handleDelete(row) {
  //1.弹出提示框
  this.$confirm("此操作永久删除当前数据，是否继续？","提示",{
    type: 'info'
  }).then(()=>{
    //2.做删除业务
    axios.delete("/books/"+row.id).then((res)=>{
      if(res.data.flag){
        this.$message.success("删除成功");
      }else{

```

```
        this.$message.error("删除失败");
    }
}).finally(()=>{
    this.getAll();
});
}).catch(()=>{
    //3.取消删除
    this.$message.info("取消删除操作");
});
},
```

## 总结

1. 请求方式使用Delete调用后台对应操作
2. 删除操作需要传递当前行数据对应的id值到后台
3. 删除操作结束后动态刷新页面加载数据
4. 根据操作结果不同，显示对应的提示信息
5. 删除操作前弹出提示框避免误操作

## F-4.修改功能

修改功能可以说是列表功能、删除功能与添加功能的合体。几个相似点如下：

1. 页面也需要有一个弹窗用来加载修改的数据，这一点与添加相同，都是要弹窗
  2. 弹出窗口中要加载待修改的数据，而数据需要通过查询得到，这一点与查询全部相同，都是要查数据
  3. 查询操作需要将要修改的数据id发送到后台，这一点与删除相同，都是传递id到后台
  4. 查询得到数据后需要展示到弹窗中，这一点与查询全部相同，都是要通过数据模型绑定展示数据
  5. 修改数据时需要将被修改的数据传递到后台，这一点与添加相同，都是要传递数据
- 所以整体上来看，修改功能就是前面几个功能的大合体
- 查询并展示数据

```
//弹出编辑窗口
handleUpdate(row) {
  axios.get("/books/"+row.id).then((res)=>{
    if(res.data.flag){
      //展示弹层，加载数据
      this.formData = res.data.data;
      this.dialogFormVisible4Edit = true;
    }else{
      this.$message.error("数据同步失败，自动刷新");
    }
  });
},
```

## 修改操作

```
//修改
handleEdit() {
  axios.put("/books",this.formData).then((res)=>{
    //如果操作成功，关闭弹层并刷新页面
    if(res.data.flag){
      this.dialogFormVisible4Edit = false;
      this.$message.success("修改成功");
    }else {
      this.$message.error("修改失败，请重试");
    }
  }).finally(()=>{
    this.getAll();
  });
},
```

## 总结

1. 加载要修改数据通过传递当前行数据对应的id值到后台查询数据（同删除与查询全部）
2. 利用前端双向数据绑定将查询到的数据进行回显（同查询全部）
3. 请求方式使用PUT调用后台对应操作（同新增传递数据）
4. 修改操作结束后动态刷新页面加载数据（同新增）
5. 根据操作结果不同，显示对应的提示信息（同新增）



## 10.业务消息一致性处理

目前的功能制作基本上达成了正常使用情况，什么叫正常使用呢？也就是这个程序不出BUG，如果我们搞一个BUG出来，你会发现程序马上崩溃掉。比如后台手工抛出一个异常，看看前端接收到的数据什么样子

```
{
  "timestamp": "2021-09-15T03:27:31.038+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/books"
}
```

面对这种情况，前端的同学又不会了，这又是什么格式？怎么和之前的格式不一样？

```
{
  "flag": true,
  "data": {
    "id": 1,
    "type": "计算机理论",
    "name": "Spring实战 第5版",
    "description": "Spring入门经典教程"
  }
}
```

看来不仅要正确的操作数据格式做处理，还要对错误的操作数据格式做同样的格式处理

首先在当前的数据结果中添加消息字段，用来兼容后台出现的操作消息

```
@Data
public class R {
    private Boolean flag;
    private Object data;
    private String msg;    //用于封装消息
}
```

后台代码也要根据情况做处理，当前是模拟的错误

```

@PostMapping
public R save(@RequestBody Book book) throws IOException {
    Boolean flag = bookService.insert(book);
    return new R(flag , flag ? "添加成功^^" : "添加失败--!");
}

```

然后在表现层做统一的异常处理，使用SpringMVC提供的异常处理器做统一的异常处理

```

@RestControllerAdvice
public class ProjectExceptionHandler {
    @ExceptionHandler(Exception.class)
    public R doOtherException(Exception ex){
        //记录日志
        //发送消息给运维
        //发送邮件给开发人员,ex对象发送给开发人员
        ex.printStackTrace();
        return new R(false,null,"系统错误，请稍后再试！");
    }
}

```

页面上得到数据后，先判定是否有后台传递过来的消息，标志就是当前操作是否成功，如果返回操作结果false，就读取后台传递的消息

```

//添加
handleAdd () {
    //发送ajax请求
    axios.post("/books",this.formData).then((res)=>{
        //如果操作成功，关闭弹层，显示数据
        if(res.data.flag){
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        }else {
            this.$message.error(res.data.msg); //消息来自于
            后台传递过来，而非固定内容
        }
    }).finally(()=>{
        this.getAll();
    });
},

```

总结

1. 使用注解@RestControllerAdvice定义SpringMVC异常处理器用来处理异常的
2. 异常处理器必须被扫描加载，否则无法生效
3. 表现层返回结果的模型类中添加消息属性用来传递消息到页面

## 11.页面功能开发

### F-5.分页功能

分页功能的制作用于替换前面的查询全部，其中要使用到elementUI提供的分页组件

```
<!--分页组件-->
<div class="pagination-container">
  <el-pagination
    class="pagiantion"
    @current-change="handleCurrentChange"
    :current-page="pagination.currentPage"
    :page-size="pagination.pageSize"
    layout="total, prev, pager, next, jumper"
    :total="pagination.total">
  </el-pagination>
</div>
```

为了配合分页组件，封装分页对应的数据模型

```
data:{
  pagination: {
    //分页相关模型数据
    currentPage: 1, //当前页码
    pageSize:10,    //每页显示的记录数
    total:0,        //总记录数
  }
},
```

修改查询全部功能为分页查询，通过路径变量传递页码信息参数

```

getAll() {

    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize).then((res) => {
        });
    },
}

```

后台提供对应的分页功能

```

@GetMapping("/{currentPage}/{pageSize}")
public R getAll(@PathVariable Integer currentPage,@PathVariable Integer pageSize){
    IPage<Book> pageBook = bookService.getPage(currentPage, pageSize);
    return new R(null != pageBook ,pageBook);
}

```

页面根据分页操作结果读取对应数据，并进行数据模型绑定

```

getAll() {

    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize).then((res) => {
        this.pagination.total = res.data.data.total;
        this.pagination.currentPage = res.data.data.current;
        this.pagination.pageSize = res.data.data.size;
        this.dataList = res.data.data.records;
    });
    },
}

```

对切换页码操作设置调用当前分页操作

```

//切换页码
handleCurrentChange(currentPage) {
    this.pagination.currentPage = currentPage;
    this.getAll();
},
}

```

总结

1. 使用el分页组件
2. 定义分页组件绑定的数据模型
3. 异步调用获取分页数据
4. 分页数据页面回显

## F-6.删除功能维护

由于使用了分页功能，当最后一页只有一条数据时，删除操作就会出现BUG，最后一页无数据但是独立展示，对分页查询功能进行后台功能维护，如果当前页码值大于最大页码值，重新执行查询。其实这个问题解决方案很多，这里给出比较简单的一种处理方案

```
@GetMapping("{currentPage}/{pageSize}")
public R getPage(@PathVariable int currentPage,@PathVariable int
pageSize){
    IPage<Book> page = bookService.getPage(currentPage, pageSize);
    //如果当前页码值大于了总页码值，那么重新执行查询操作，使用最大页码值作为当前页
    码值
    if( currentPage > page.getPages()){
        page = bookService.getPage((int)page.getPages(), pageSize);
    }
    return new R(true, page);
}
```

## F-7.条件查询功能

最后一个功能来做条件查询，其实条件查询可以理解为分页查询的时候除了携带分页数据再多带几个数据的查询。这些多带的就是查询条件。比较一下不带条件的分页查询与带条件的分页查询差别之处，这个功能就好做了

- 页面封装的数据：带不带条件影响的仅仅是一次性传递到后台的数据总量，由传递2个分页相关的数据转换成2个分页数据加若干个条件
- 后台查询功能：查询时由不带条件，转换成带条件，反正不带条件的时候查询条件对象使用的是null，现在换成具体条件，差别不大
- 查询结果：不管带不带条件，出来的数据只是有数量上的差别，其他都差别，这个可以忽略

经过上述分析，看来需要在页面发送请求的格式方面做一定的修改，后台的调用数据层操作时发送修改，其他没有区别

页面发送请求时，两个分页数据仍然使用路径变量，其他条件采用动态拼装url参数的形式传递

页面封装查询条件字段

```
pagination: {  
  //分页相关模型数据  
  currentPage: 1,      //当前页码  
  pageSize: 10,        //每页显示的记录数  
  total: 0,            //总记录数  
  name: "",  
  type: "",  
  description: ""  
},
```

页面添加查询条件字段对应的数据模型绑定名称

```
<div class="filter-container">  
  <el-input placeholder="图书类别" v-  
model="pagination.type" class="filter-item"/>  
  <el-input placeholder="图书名称" v-  
model="pagination.name" class="filter-item"/>  
  <el-input placeholder="图书描述" v-  
model="pagination.description" class="filter-item"/>  
  <el-button @click="getAll()" class="dalfBut">查询</el-  
button>  
  <el-button type="primary" class="butT"  
@click="handleCreate()">新建</el-button>  
</div>
```

将查询条件组织成url参数，添加到请求url地址中，这里可以借助其他类库快速开发，当前使用手工形式拼接，降低学习要求

```

getAll() {
    //1. 获取查询条件,拼接查询条件
    param = "?name="+this.pagination.name;
    param += "&type="+this.pagination.type;
    param += "&description="+this.pagination.description;
    console.log("-----"+ param);

    axios.get("/books/"+this.pagination.currentPage+"/"+this.pagination.pageSize+param).then((res) => {
        this.dataList = res.data.data.records;
    });
},

```

后台代码中定义实体类封装查询条件

```

@GetMapping("/{currentPage}/{pageSize}")
public R getAll(@PathVariable int currentPage,@PathVariable int pageSize,Book book) {
    System.out.println("参数====>"+book);
    IPage<Book> pageBook =
    bookService.getPage(currentPage,pageSize);
    return new R(null != pageBook ,pageBook);
}

```

对应业务层接口与实现类进行修正

```

public interface IBookService extends IService<Book> {
    IPage<Book> getPage(Integer currentPage,Integer
    pageSize,Book queryBook);
}

```

```

@Service
public class BookServiceImpl2 extends
ServiceImpl<BookDao, Book> implements IBookService {
    public IPage<Book> getPage(Integer currentPage, Integer
pageSize, Book queryBook){
        IPage page = new Page(currentPage, pageSize);
        LambdaQueryWrapper<Book> lqw = new
LambdaQueryWrapper<Book>();

        lqw.like(Strings.isNotEmpty(queryBook.getName()), Book::getN
ame, queryBook.getName());

        lqw.like(Strings.isNotEmpty(queryBook.getType()), Book::getT
ype, queryBook.getType());

        lqw.like(Strings.isNotEmpty(queryBook.getDescription()), Boo
k::getDescription, queryBook.getDescription());
        return bookDao.selectPage(page, lqw);
    }
}

```

页面回显数据

```

getAll() {
    //1. 获取查询条件, 拼接查询条件
    param = "?name="+this.pagination.name;
    param += "&type="+this.pagination.type;
    param += "&description="+this.pagination.description;
    console.log("-----"+ param);

    axios.get("/books/"+this.pagination.currentPage+"/"+this.pa
gination.pageSize+param).then((res) => {
        this.pagination.total = res.data.data.total;
        this.pagination.currentPage = res.data.data.current;
        this.pagination.pageSize = res.data.data.size;
        this.dataList = res.data.data.records;
    });
},

```

总结

1. 定义查询条件数据模型（当前封装到分页数据模型中）



## 2. 异步调用分页功能并通过请求参数传递数据到后台

### 基础篇完结

基础篇到这里就全部结束了，在基础篇中带着大家学习了如果创建一个SpringBoot工程，然后学习了SpringBoot的基础配置语法格式，接下来对常见的市面上的实用技术做了整合，最后通过一个小的案例对前面学习的内容做了一个综合应用。整体来说就是一个最基本的入门，关于SpringBoot的实际开发其实接触的还是很少的，我们到实用篇和原理篇中继续吧，各位小伙伴，加油学习，再见。