p32~p53 SSMP案例

制作分析

效果演示

流程解析

Java 复制代码

```
1. 案例实现方案分析
    实体类开发————使用Lombok快速制作实体类
    Dao开发————整合MyBatisPlus，制作数据层测试类
    Service开发————基于MyBatisPlus进行增量开发，制作业务层测试类
    Controller开发————基于Restful开发，使用PostMan测试接口功能
    Controller开发————前后端开发协议制作
    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
    项目异常处理
    按条件查询————页面功能调整、Controller修正功能、Service修正功能
2. SSMP案例制作流程解析
    先开发基础CRUD功能，做一层测一层
    调通页面，确认异步提交成功后，制作所有功能
    添加分页功能与查询功能

```

模块创建

- SpringBoot Initializr创建模块
- 勾选web、mysql dirver，手动添加mybatis-plus和druid的依赖坐标（需要添加version，因为SpringBoot官方没有收录他们的version坐标）

实体类开发

- 使用Lombok进行实体类的快速开发，Lombok提供了一组注解来简化POJO实体类的开发
- 在pom.xml文件中添加对应的依赖坐标，version坐标由SpringBoot提供

XML 复制代码

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

- 常用注解@Data，该注解可以为当前实体类在编译期设置对应的get/set方法，toString方法，hashCode方法，equals方法等。
- **注意：@Data不会给当前实体类添加constructor方法，需要自己添加@NoArgsConstructor或者@AllArgsConstructor到实体类上方。**

Java 复制代码

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Book {
    private Integer id;
    private String type;
    private String name;
    private String description;
}
```

#

数据层标准开发（基础CRUD）

- 导入MyBatis-plus和Druid对应的starter依赖坐标

XML 复制代码

```xml
<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>
```

- 配置Druid数据源和MyBatis-Plus对应的基础配置（表单中id的自增策略使用数据库自增而不是MyBatis-Plus的雪花算法的自增策略）

YAML 复制代码

```yaml
# 设置Druid相关配置
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/mybatis?serverTimezone=UTC
      username: root
      password: 865330

# 设置MyBatisPlus相关配置
mybatis-plus:
  global-config:
    db-config:
      table-prefix: tbl_
      # 使用数据库的id自增而不是MP自带的雪花算法自增
      id-type: auto
  # 开启MP日志模式
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

- Dao类（Mapper类）继承MP中提供的BaseMapper接口并指定泛型

Java | 复制代码

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

- 制作测试类

Java | 复制代码

```java
@SpringBootTest
public class BookDaoTestCase {

    @Autowired
    private BookDao bookDao;

    @Test
    void testGetById(){
        Book book = bookDao.selectById(1);
        // 开启MP日志之后，就不用打印出来看了，直接看日志
        System.out.println(book);
    }

    @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(14);
        book.setType("测试数据abc");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookDao.updateById(book);
    }

    @Test
    void testDelete(){
        bookDao.deleteById(15);
    }

    @Test
    void testGetAll(){
        // 开启MP日志之后，就不用打印出来看了，直接看日志
        bookDao.selectList(null);
    }

    @Test
    void testGetPage(){
        // 得到第1页的5条数据
```

```
46          IPage page = new Page(2,5);
47          bookDao.selectPage(page, null);
48          System.out.println(page.getPages());
49          System.out.println(page.getTotal());
50          System.out.println(page.getCurrent());
51          System.out.println(page.getSize());
52          System.out.println(page.getRecords());
53      }
54
55      @Test
56      void testGetBy(){
57          QueryWrapper<Book> qw = new QueryWrapper<>();
58          // select * from tbl_book where name like %Spring%
59          qw.like("name","Spring");
60          bookDao.selectList(qw);
61      }
62
63      @Test
64      void testGetBy2(){
65          String name = "Spring";
66          LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<>();
67          // select * from tbl_book where name like %Spring%
68          lqw.like(name!=null, Book::getName, "Spring");
69          bookDao.selectList(lqw);
70      }
71  }
```

开启MP运行日志

```yaml
# 设置MyBatisPlus相关配置
mybatis-plus:
global-config:
db-config:
table-prefix: tbl_
# 使用数据库的id自增而不是MP自带的雪花算法自增
id-type: auto
# 开启MP日志模式
configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

分页

- 分页需要设定分页对象IPage

```java
@Test
    void testGetPage(){
        // 得到第1页的5条数据
        IPage page = new Page(1,5);
        bookDao.selectPage(page, null);
        System.out.println(page.getPages());
        System.out.println(page.getTotal());
        System.out.println(page.getCurrent());
        System.out.println(page.getSize());
        System.out.println(page.getRecords());
    }
```

- bookDao.selectPage(page, null)返回的是IPage对象，该对象中封装了分页操作中的所有数据：
  - 数据
  - 当前页码值
  - 每页数据总量
  - 最大页码值
  - 数据总量
- 分页操作是在MyBatisPlus的常规操作基础上增强得到，**内部是动态的拼写SQL语句（在这**

里，分页操作，也就是给SQL语句尾部动态添加LIMIT关键词），因此需要增强对应的功能，使用MyBatisPlus拦截器实现

Java 复制代码

```java
@Configuration
public class MPConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        // 1.定义拦截器
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        // 2.添加具体的拦截器
        interceptor.addInnerInterceptor(new PaginationInnerInterceptor());
        return interceptor;
    }
}
```

- 测试

Java 复制代码

```java
@Test
void testGetPage() {
    IPage page = new Page(1, 5);
    bookDao.selectPage(page, null);
    System.out.println(page.getCurrent());
    System.out.println(page.getSize());
    System.out.println(page.getPages());
    System.out.println(page.getTotal());
    System.out.println(page.getRecords());
}
```

数据层标准开发（条件查询）

- 使用QueryWrapper对象封装查询条件，推荐使用LambdaQueryWrapper对象，所有查询操作封装成方法调用

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


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

- 条件查询支持动态拼接查询条件

```java
@Test
void testGetBy2() {
    String name = "1";
    LambdaQueryWrapper<Book> lambdaQueryWrapper = new
LambdaQueryWrapper<>();
    //if (name != null) lambdaQueryWrapper.like(Book::getName,name);
    lambdaQueryWrapper.like(Strings.isNotEmpty(name), Book::getName,
name);
    bookDao.selectList(lambdaQueryWrapper);
}

```

小结:

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

业务层标准开发（CRUD）

- 业务层（service）接口定义与数据层（Dao）接口定义有较大区别，注意区分不要混用
  - example

Java 复制代码

```
selectByUserNameAndPassword(String username,String password); //数据层接口
login(String username,String password); //Service层接口
```

- 定义接口

Java 复制代码

```
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);
}

```

- 接口实现类

Java | 复制代码

```java
// 定义成业务层的bean
@Service
public class BookServiceImpl2 implements BookService {

    @Autowired
    private BookDao bookDao;

    @Override
    public Boolean save(Book book) {
        // bookDao.insert(book)返回的是操作的行数，如果行数大于0说明操作成功
        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 current, int pageSize) {
        IPage page = new Page(current, pageSize);
        bookDao.selectPage(page, null);
        // selectPage()之后page对象会被修改，最后还是返回page对象
        return page;
    }
}
```

- 测试类

Java | 复制代码

```java
@SpringBootTest
public class BookServiceTestCase {

    @Autowired
    private BookService 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(14);
        book.setType("测试数据abc");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.update(book);
    }

    @Test
    void testDelete(){
        System.out.println("delete true or false:
"+bookService.delete(15));;
    }

    @Test
    void testGetAll(){
        System.out.println("====testGetAll starts=====");
        System.out.println(bookService.getAll());;
        System.out.println("====testGetAll ends=====");
    }

    @Test
    void testGetPage(){
        // 得到第2页的5条数据
```

```
45            IPage<Book> page = bookService.getPage(2, 5);
46            System.out.println(page.getPages());
47            System.out.println(page.getTotal());
48            System.out.println(page.getCurrent());
49            System.out.println(page.getSize());
50            System.out.println(page.getRecords());
51        }
52    }
```

小结:

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（基于MyBatisPlus构建）

- 快速开发方案
  - 使用MyBatisPlus提供有业务层通用接口（IServivce）与业务层通用实现（ServiceImpl<M,T>）
  - 在通用类基础上做功能重载或功能追加
  - 注意重载时不要覆盖原始操作，避免原始提供的功能丢失
- 定义接口，继承IService

Java 复制代码

```
1   public interface IBookService extends IService<Book> {}
```

- 追加接口功能

```java
public interface IBookService extends IService<Book> {

    // 追加的操作与原始操作通过名称区分，功能类似
    Boolean delete(Integer id);

    Boolean insert(Book book);

    Boolean modify(Book book);

    Book get(Integer id);
}

```

- 实现类

```java
@Service
public class BookServiceImpl extends ServiceImpl<BookDao, Book>
implements IBookService {}
```

- 实现类追加功能

Java | 复制代码

```java
@Service
public class BookServiceImpl extends ServiceImpl<BookDao, Book>
implements IBookService {

    @Autowired
    private BookDao bookDao;

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

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

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

    public Book get(Integer id) {
        return bookDao.selectById(id);
    }
}

```

- 测试类

Java | 复制代码

```java
@SpringBootTest
public class IBookServiceTestCase {

    @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(14);
        book.setType("测试数据abc");
        book.setName("测试数据123");
        book.setDescription("测试数据123");
        bookService.updateById(book);
    }

    @Test
    void testDelete(){
        System.out.println("delete true or false:
"+bookService.removeById(15));;
    }

    @Test
    void testGetAll(){
        System.out.println("====testGetAll starts=====");
        System.out.println(bookService.list());;
        System.out.println("====testGetAll ends=====");
    }

    @Test
    void testGetPage(){
        // 得到第2页的5条数据
```

```
45            IPage<Book> page = new Page<>(2, 5);
46            bookService.page(page);
47            System.out.println(page.getPages());
48            System.out.println(page.getTotal());
49            System.out.println(page.getCurrent());
50            System.out.println(page.getSize());
51            System.out.println(page.getRecords());
52        }
53    }
```

小结：

1. 使用通用接口（ISerivce）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

表现层标准开发

- 基于RESTFul风格进行表现层开发
- 使用Postman测试表现层接口功能
- 表现类

Java | 复制代码

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

    @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 Integer currentPage, @PathVariable int pageSize) {
        return bookService.getPage(currentPage, pageSize);
    }

}
```

- 添加分页的业务层方法

IBookService.java Java 复制代码

```java
IPage<Book> getPage(int currentPage,int pageSize);
```

BookServiceImpl.java Java 复制代码

```java
@Override
public IPage<Book> getPage(int currentPage, int pageSize) {

    IPage page = new Page(currentPage, pageSize);
    bookDao.selectPage(page, null);

    return page;
}
```

小结:

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

表现层数据一致性处理（R对象）

- 增加response的状态属性（flag）

- 增删改

```
{
    "flag": true,
    "data": null
}
```
成功

```
{
    "flag": false,
    "data": null
}
```
失败

- 查单条

```
{
    "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"
    }
}
```

```
{
    "flag": true,
    "data": null
}
```
查询数据不存在

```
{
    "flag": false,
    "data": null
}
```
查询过程抛异常

- 查全部

```
{
    "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"
        },
        {
            "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"
        }
    ]
}
```


- 当数据为 null 可能出现的问题
  - 查询id不存在的数据，返回 null
  - 查询过程中抛出异常，catch 中返回 null
- 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**

Java | 复制代码

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

    public R() {
    }

    /**
     * 不返回数据的构造方法
     *
     * @param flag
     */
    public R(Boolean flag) {
        this.flag = flag;
    }

    /**
     * 返回数据的构造方法
     *
     * @param flag
     * @param data
     */
    public R(Boolean flag, Object data) {
        this.flag = flag;
        this.data = data;
    }
}
```

- 表现层接口统一返回值类型结果

Java | 复制代码

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

    @Autowired
    private IBookService bookService;

    @GetMapping
    public R getAll() {
        return new R(true, bookService.list());
    }

    @PostMapping
    public R save(@RequestBody Book book) {
        return new R(bookService.save(book));

    }

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

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

    @GetMapping("{id}")
    public R getById(@PathVariable Integer id) {
        return new R(true, bookService.getById(id));
    }

    @GetMapping("{currentPage}/{pageSize}")
    public R getPage(@PathVariable Integer currentPage, @PathVariable int pageSize) {
        return new R(true, bookService.getPage(currentPage, pageSize));
    }

}
```

小结：

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

前后端调用（axios发送异步请求）

- 前后端分离结构设计中页面归属前端服务器
- 单体工程中页面放置在resources目录下的static目录中（建议执行clean）
- 前端发送异步请求，调用后端接口

JavaScript 复制代码

```
//钩子函数，VUE对象初始化完成后自动执行
created() {
  //调用查询全部数据的操作
  this.getAll();
}
```

JavaScript 复制代码

```
//列表
getAll() {
    //发送异步请求
    axios.get("/books").then((res)=>{
        console.log(res.data);
    })
}
```

小结：

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

列表功能

- 列表页

```JavaScript
//列表
getAll() {
    //发送异步请求
    axios.get("/books").then((res) => {
        //console.log(res.data);
        this.dataList = res.data.data;
    })
}
```

小结：

1. 将查询数据返回到页面，利用前端v-bind数据双向绑定进行数据展示

添加功能

- 弹出添加窗口

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

- 清除数据

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

- 在弹出添加窗口时 清除数据

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

- 发送请求

```javascript
//添加
handleAdd() {
    axios.post("/books", this.formData).then((res) => {
        //判断当前操作是否成功
        if (res.data.flag) {
            //1.关闭弹层
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        } else {
            this.$message.error("添加失败");
        }
    }).finally(() => {
        //2.重新加载数据
        this.getAll();
    })
}
```

- 取消添加

```javascript
//取消
cancel() {
    //1.关闭弹层
    this.dialogFormVisible = false;
    //2.提示用户
    this.$message.info("当前操作取消");
}
```

小结:

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

删除功能

- 删除

JavaScript | 复制代码

```
// 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();
    });
}
```

- 加入确认删除对话框

```javascript
// 删除
handleDelete(row) {
    //1. 弹出提示框
    this.$confirm("些操作永久删除当前信息,是否继续?", "提示", {type: "info"}).then(() => {
        //2. 做删除业务
        axios.delete("/books/" + row.id).then((res) => {
            //判断当前操作是否成功
            if (res.data.flag) {
                this.$message.success("删除成功");
            } else {
                this.$message.error("删除失败");
            }
        }).finally(() => {
            //2.重新加载数据
            this.getAll();
        })
    }).catch(() => {
        //3. 取消删除
        this.$message.info("取消操作");
    });

}
```

小结:

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

修改功能（加载数据）

- 弹出修改窗口

JavaScript | 复制代码

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

- 删除消息维护

JavaScript | 复制代码

```javascript
// 删除
handleDelete(row) {
    //1. 弹出提示框
    this.$confirm("些操作永久删除当前信息,是否继续?", "提示", {type: "info"}).then(() => {
        //2. 做删除业务
        axios.delete("/books/" + row.id).then((res) => {
            //判断当前操作是否成功
            if (res.data.flag) {
                this.$message.success("删除成功");
            } else {
                this.$message.error("数据同步失败，自动刷新");
            }
        }).finally(() => {
            //2.重新加载数据
            this.getAll();
        });
    }).catch(() => {
        //3. 取消删除
        this.$message.info("取消操作");
    });

}
```

修改功能

- 修改

```javascript
//修改
handleEdit() {
    axios.put("/books", this.formData).then((res) => {
        //判断当前操作是否成功
        if (res.data.flag) {
            //1.关闭弹层
            this.dialogFormVisible4Edit = false;
            this.$message.success("修改成功");
        } else {
            this.$message.error("修改失败");
        }
    }).finally(() => {
        //2.重新加载数据
        this.getAll();
    });
}
```

- 取消添加和修改

```javascript
//取消
cancel() {
    //1.关闭弹层
    this.dialogFormVisible = false;
    this.dialogFormVisible4Edit = false;
    //2.提示用户
    this.$message.info("当前操作取消");
}
```

小结:

1. 请求方式使用PUT调用后台对应操作
2. 修改操作结束后动态刷新页面加载数据（同新增）
3. 根据操作结果不同，显示对应的提示信息（同新增）

异常消息处理

- 业务操作成功或失败返回数据格式

JSON 复制代码

```json
{
  "flag": true,
  "data": null
}

{
  "flag": false,
  "data": null
}
```

- 后台代码BUG导致数据格式不统一性

JSON 复制代码

```json
{
    "timestamp": "2021-11-07T12:44:29.343+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/books"
}
```

- 对异常进行统一处理，出现异常后，返回指定信息

Java 复制代码

```java
@RestControllerAdvice
public class ProjectExceptionAdvice {

    //拦截所有的异常信息
    @ExceptionHandler(Exception.class)
    public R doException(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！");
    }
}
```

- 修改表现层返回结果的模型类，封装出现异常后对应的信息

  flag：false

  Data: null

  消息(msg): 要显示信息

Java 复制代码

```java
@Data
public class R {
    private Boolean flag;
    private Object data;
    private String msg;

    public R() {
    }

    public R(Boolean flag) {
        this.flag = flag;
    }

    public R(Boolean flag, Object data) {
        this.flag = flag;
        this.data = data;
    }

    public R(Boolean flag, String msg) {
        this.flag = flag;
        this.msg = msg;
    }

    public R(String msg) {
        this.flag = false;
        this.msg = msg;
    }
}
```

- 页面消息处理，没有传递消息加载默认消息，传递消息后加载指定消息

JavaScript 复制代码

```javascript
//添加
handleAdd() {
    axios.post("/books", this.formData).then((res) => {
        //判断当前操作是否成功
        if (res.data.flag) {
            //1.关闭弹层
            this.dialogFormVisible = false;
            this.$message.success("添加成功");
        } else {
            this.$message.error(res.data.msg);
        }
    }).finally(() => {
        //2.重新加载数据
        this.getAll();
    })
}
```

- 在表现层Controller中进行消息统一处理

Java 复制代码

```java
@PostMapping
public R save(@RequestBody Book book) throws IOException {
    //if (book.getName().equals("123")) throw new IOException();
    boolean flag = bookService.save(book);
    return new R(flag, flag ? "添加成功^_^" : "添加失败-_-!");
}
```

- 页面消息处理

JavaScript | 复制代码

```javascript
//添加
handleAdd() {
    axios.post("/books", this.formData).then((res) => {
        //判断当前操作是否成功
        if (res.data.flag) {
            //1.关闭弹层
            this.dialogFormVisible = false;
            this.$message.success(res.data.msg);
        } else {
            this.$message.error(res.data.msg);
        }
    }).finally(() => {
        //2.重新加载数据
        this.getAll();
    })
}
```

小结:

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

分页

- 页面使用 el 分页组件添加分页功能

HTML | 复制代码

```html
<!--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>
```

- 定义分页组件需要使用的数据并将数据绑定到分页组件

Vue | 复制代码

```vue
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
    }
}
```

- 替换查询全部功能为分页功能

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

- 分页查询

  使用路径参数传递分页数据或封装对象传递数据

```javascript
@GetMapping("{currentPage}/{pageSize}")
public R getPage(@PathVariable Integer currentPage, @PathVariable int pageSize) {
    return new R(true, bookService.getPage(currentPage, pageSize));
}
```

- 加载分页数据

```javascript
//分页查询
getAll() {
   //发送异步请求
   axios.get("/books/" + this.pagination.currentPage + "/" + this.pagination.pageSize).then((res) => {
       //console.log(res.data);
       this.pagination.currentPage = res.data.data.current;
       this.pagination.pageSize = res.data.data.size;
       this.pagination.total = res.data.data.total;

       this.dataList = res.data.data.records;
   })
}
```

- 分页页码值切换

JavaScript 复制代码

```javascript
//切换页码
handleCurrentChange(currentPage) {
    //修改页码值为当前选中的页码值
    this.pagination.currentPage = currentPage;
    //执行查询
    this.getAll();
}
```

小结:

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

分页功能维护（删除BUG）

- 对查询结果进行校验，如果当前页码值大于最大页码值，使用最大页码值作为当前页码值重新查询

Java 复制代码

```java
@GetMapping("{currentPage}/{pageSize}")
public R getPage(@PathVariable Integer 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);
}
```

小结:

1. 基于业务需求维护删除功能

条件查询

- 查询条件数据封装
  单独封装
  与分页操作混合封装

JSON 复制代码

```json
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
  type: "",
  name: "",
  description: ""
}
```

- 使用v-model实现页面数据模型绑定

HTML 复制代码

```html
<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>
```

- 组织数据成为get请求发送的数据

JavaScript 复制代码

```javascript
//分页查询
getAll() {
    console.log(this.pagination.type);

    //  /books/1/10?type=???&name=???&decription=?? ;
    //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);

    //发送异步请求
    axios.get("/books/" + this.pagination.currentPage + "/" + this.pagination.pageSize + param).then((res) => {
        //console.log(res.data);
        this.pagination.currentPage = res.data.data.current;
        this.pagination.pageSize = res.data.data.size;
        this.pagination.total = res.data.data.total;

        this.dataList = res.data.data.records;
    })
}
```

- 条件参数组织可以通过条件判定书写的更简洁
- Controller接收参数

Java 复制代码

```java
@GetMapping("/{current}/{pageSize}")
// 传递的参数名为name,type,description，参数名与实体类Book中的属性名一直，SpringMVC自动为实体类注入属性
public R getPage(@PathVariable int current, @PathVariable int pageSize, Book book){
    IPage<Book> page = bookService.getPage(current, pageSize, book);
    // 如果当前页码值大于总页码值，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 (current > page.getPages()){
        page = bookService.getPage((int) page.getPages(), pageSize, book);
    }
    return new R(true, page);
}
```

- 业务层接口功能开发

```java
/**
    * 分页的条件查询
    *
    * @param currentPage
    * @param pageSize
    * @param book
    * @return
*/
IPage<Book> getPage(Integer currentPage, int pageSize, Book book);
```

- 业务层接口实现类功能开发

```java
@Override
public IPage<Book> getPage(int current, int pageSize, Book book) {
    LambdaQueryWrapper<Book> lqw = new LambdaQueryWrapper<>();
    // like的参数：第一个是条件，第二个是对应的属性，第三个是对应的值
    lqw.like(Strings.isNotEmpty(book.getType()), Book::getType, book.getType());
    lqw.like(Strings.isNotEmpty(book.getName()), Book::getName, book.getName());
    lqw.like(Strings.isNotEmpty(book.getDescription()), Book::getDescription, book.getDescription());

    IPage page = new Page(current, pageSize);
    bookDao.selectPage(page, lqw);
    // selectPage()之后page对象会被修改，最后还是返回page对象
    return page;
}
```

- Controller调用业务层分页条件查询接口

Java 复制代码

```java
@GetMapping("/{current}/{pageSize}")
// 传递的参数名为name,type,description，参数名与实体类Book中的属性名一直，SpringMVC自动为实体类注入属性
public R getPage(@PathVariable int current, @PathVariable int pageSize, Book book){
    IPage<Book> page = bookService.getPage(current, pageSize, book);
    // 如果当前页码值大于总页码值，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 (current > page.getPages()){
        page = bookService.getPage((int) page.getPages(), pageSize, book);
    }
    return new R(true, page);
}
```

- 页面回显数据

JavaScript 复制代码

```javascript
//分页查询
getAll() {
    console.log(this.pagination.type);

    //  /books/1/10?type=???&name=???&decription=?? ;
    //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);

    //发送异步请求
    axios.get("/books/" + this.pagination.currentPage + "/" + this.pagination.pageSize + param).then((res) => {
        //console.log(res.data);
        this.pagination.currentPage = res.data.data.current;
        this.pagination.pageSize = res.data.data.size;
        this.pagination.total = res.data.data.total;

        this.dataList = res.data.data.records;
    })
}
```

小结:

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

流程总结

1. pom.xml
    a. 配置起步依赖
2. application.yml
    a. 设置数据源、端口、框架技术相关配置等
3. dao
    a. 继承BaseMapper、设置@Mapper
4. dao测试类
5. service
    a. 调用数据层接口或MyBatis-Plus提供的接口快速开发
6. service测试类
7. controller
    a. 基于Restful开发，使用Postman测试跑通功能
8. 页面
    a. 放置在resources目录下的static目录中