

9. MyBatis 操作数据库(进阶)

本节目标

1. 学习MyBatis的动态SQL查询
2. 掌握MyBatis在项目中的应用, 可以使用Spring MVC完成一些基础的功能

1. 动态SQL

动态 SQL 是Mybatis的强大特性之一, 能够完成不同条件下不同的 sql 拼接
可以参考官方文档: [Mybatis动态sql](#)

1.1 <if>标签

在注册用户的时候, 可能会有这样一个问题, 如下图所示:

添加用户

* 真实姓名 请输入真实姓名 * 用户密码 请输入用户密码 必填字段 角色 请选择

比特校区 请选择归属校区

比特班级 请选择归属班级

* 手机号码 请输入手机号码 * 邮箱 请输入邮箱 生日 选择日期

用户性别 男 状态 ☒ 正常 ☐ 停用

备注 非必填字段

确定 取消

注册分为两种字段: 必填字段和非必填字段, 那如果在添加用户的时候有不确定的字段传入, 程序应该如何实现呢?

这个时候就需要使用动态标签 来判断了, 比如添加的时候性别 gender 为非必填字段, 具体实现如下:

接口定义:

```
1 Integer insertUserByCondition(UserInfo userInfo);
```

Mapper.xml实现:

```
1 <insert id="insertUserByCondition">
2     INSERT INTO userinfo (
3         username,
4         `password`,
5         age,
6         <if test="gender != null">
7             gender,
8         </if>
9         phone)
10    VALUES (
11        #{username},
12        #{age},
13        <if test="gender != null">
14            #{gender},
15        </if>
16        #{phone})
17 </insert>
```

或者使用注解方式(不推荐)

把上面SQL(包括标签), 使用 `<script></script>` 标签括起来就可以

```
1 @Insert("<script>" +
2         "INSERT INTO userinfo (username,`password`,age," +
3         "<if test='gender!=null'>gender,</if>" +
4         "phone)" +
5         "VALUES(#{username},#{age}," +
6         "<if test='gender!=null'>#{gender},</if>" +
7         " #{phone})"+
8         "</script>")
9 Integer insertUserByCondition(UserInfo userInfo);
```

注意 test 中的 gender，是传入对象中的属性，不是数据库字段

Q: 可不可以不进行判断, 直接把字段设置为null呢?

A: 不可以, 这种情况下, 如果gender字段有默认值, 就会设置为默认值

1.2 <trim>标签

之前的插入用户功能，只是有一个 gender 字段可能是选填项，如果有多个字段，一般考虑使用标签结合标签，对多个字段都采取动态生成的方式。

标签中有如下属性：

- prefix：表示整个语句块，以prefix的值作为前缀
- suffix：表示整个语句块，以suffix的值作为后缀
- prefixOverrides：表示整个语句块要去除掉的前缀
- suffixOverrides：表示整个语句块要去除掉的后缀

调整 Mapper.xml 的插入语句为：

```
1 <insert id="insertUserByCondition">
2     INSERT INTO userinfo
3     <trim prefix="(" suffix=")" suffixOverrides=",">
4         <if test="username != null">
5             username,
6         </if>
7         <if test="password != null">
8             `password`,
9         </if>
10        <if test="age != null">
11            age,
12        </if>
13        <if test="gender != null">
14            gender,
15        </if>
16        <if test="phone != null">
17            phone,
18        </if>
19    </trim>
20    VALUES
21    <trim prefix="(" suffix=")" suffixOverrides=",">
22        <if test="username != null">
23            #{username},
24        </if>
25        <if test="password != null">
26            #{password},
27        </if>
28        <if test="age != null">
29            #{age},
30        </if>
31        <if test="gender != null">
32            #{gender},
33        </if>
34        <if test="phone != null">
```

```

35         #{phone}
36     </if>
37 </trim>
38 </insert>

```

或者使用注解方式(不推荐)

```

1 @Insert("<script>" +
2     "INSERT INTO userinfo " +
3     "<trim prefix=(' suffix=') suffixOverrides=',>" +
4     "<if test='username!=null'>username,</if>" +
5     "<if test='password!=null'>password,</if>" +
6     "<if test='age!=null'>age,</if>" +
7     "<if test='gender!=null'>gender,</if>" +
8     "<if test='phone!=null'>phone,</if>" +
9     "</trim>" +
10    "VALUES " +
11    "<trim prefix=(' suffix=') suffixOverrides=',>" +
12    "<if test='username!=null'>#{username},</if>" +
13    "<if test='password!=null'>#{password},</if>" +
14    "<if test='age!=null'>#{age},</if>" +
15    "<if test='gender!=null'>#{gender},</if>" +
16    "<if test='phone!=null'>#{phone}</if>" +
17    "</trim>" +
18    "</script>")
19 Integer insertUserByCondition(UserInfo userInfo);

```

在以上 sql 动态解析时，会将第一个 部分做如下处理：

- 基于 `prefix` 配置，开始部分加上 `(`
- 基于 `suffix` 配置，结束部分加上 `)`
- 多个 组织的语句都以 `,` 结尾，在最后拼接好的字符串还会以 `,` 结尾，会基于 `suffixOverrides` 配置去掉最后一个 `,`
- 注意 `<if test="username !=null">` 中的 `username` 是传入对象的属性

1.3 <where>标签

看下面这个场景，系统会根据我们的筛选条件，动态组装where 条件



这种如何实现呢？

接下来我们看代码实现：

需求: 传入的用户对象，根据属性做where条件查询，用户对象中属性不为 null 的，都为查询条件. 如 username 为 "a"，则查询条件为 where username="a"

原有SQL

```
1 SELECT
2     *
3 FROM
4     userinfo
5 WHERE
6     age = 18
7     AND gender = 1
8     AND delete_flag =0
```

接口定义:

```
1 List<UserInfo> queryByCondition();
```

Mapper.xml实现

```
1 <select id="queryByCondition" resultType="com.example.demo.model.UserInfo">
2     select id, username, age, gender, phone, delete_flag, create_time,
3     update_time
4     from userinfo
5     <where>
6         <if test="age != null">
7             and age = #{age}
8         </if>
9         <if test="gender != null">
10            and gender = #{gender}
11        </if>
```

```

11         <if test="deleteFlag != null">
12             and delete_flag = #{deleteFlag}
13         </if>
14     </where>
15 </select>

```

`<where>` 只会在子元素有内容的情况下才插入where子句，而且会自动去除子句的开头的AND或OR

以上标签也可以使用 `<trim prefix="where" prefixOverrides="and">` 替换, 但是此种情况下, 当子元素都没有内容时, where关键字也会保留

或者使用注解方式

```

1 @Select("<script>select id, username, age, gender, phone, delete_flag,
   create_time, update_time" +
2         "    from userinfo" +
3         "    <where>" +
4         "        <if test='age != null'> and age = #{age} </if>" +
5         "        <if test='gender != null'> and gender = #{gender} </if>" +
6         "        <if test='deleteFlag != null'> and delete_flag = #
   {deleteFlag} </if>" +
7         "    </where>" +
8         "</script>")
9 List<UserInfo> queryByCondition(UserInfo userInfo);

```

1.4 <set>标签

需求: 根据传入的用户对象属性来更新用户数据, 可以使用标签来指定动态内容.

接口定义: 根据传入的用户 id 属性, 修改其他不为 null 的属性

```

1 Integer updateUserByCondition(UserInfo userInfo);

```

Mapper.xml

```

1 <update id="updateUserByCondition">
2     update userinfo
3     <set>
4         <if test="username != null">
5             username = #{username},
6         </if>

```

```

7      <if test="age != null">
8          age = #{age},
9      </if>
10     <if test="deleteFlag != null">
11         delete_flag = #{deleteFlag},
12     </if>
13 </set>
14     where id = #{id}
15 </update>

```

`<set>`：动态的在SQL语句中插入set关键字，并会删掉额外的逗号。(用于update语句中)
 以上标签也可以使用 `<trim prefix="set" suffixOverrides=",">` 替换。

或者使用注解方式

```

1 @Update("<script>" +
2     "update userinfo " +
3     "<set>" +
4     "<if test='username!=null'>username=#{username},</if>" +
5     "<if test='age!=null'>age=#{age},</if>" +
6     "<if test='deleteFlag!=null'>delete_flag=#{deleteFlag},</if>" +
7     "</set>" +
8     "where id=#{id}" +
9     "</script>")
10 Integer updateUserByCondition(UserInfo userInfo);

```

1.5 <foreach>标签

对集合进行遍历时可以使用该标签。标签有如下属性：

- collection：绑定方法参数中的集合，如 List，Set，Map或数组对象
- item：遍历时的每一个对象
- open：语句块开头的字符串
- close：语句块结束的字符串
- separator：每次遍历之间间隔的字符串

需求: 根据多个userid, 删除用户数据

接口方法:

```

1 void deleteByIds(List<Integer> ids);

```

ArticleMapper.xml 中新增删除 sql:

```
1 <delete id="deleteByIds">
2     delete from userinfo
3     where id in
4     <foreach collection="ids" item="id" separator="," open="(" close=")">
5         #{id}
6     </foreach>
7 </delete>
```

或者使用注解方式:

```
1 @Delete("<script>" +
2         "delete from userinfo where id in" +
3         "<foreach collection='ids' item='id' separator=',' open='('
4         close=')'>" +
5         "#{id}" +
6         "</foreach>" +
7         "</script>")
8 Integer deleteUser(Integer id);
```

1.6 <include>标签

问题分析:

- 在xml映射文件中配置的SQL，有时可能会存在很多重复的片段，此时就会存在很多冗余的代码


```

<select id="queryAllUser" resultMap="BaseMap">
    select id, username, age, gender, phone, delete_flag, create_time, update_time
    from userinfo
</select>
<select id="queryById" resultType="com.example.demo.model.UserInfo">
    select id, username, age, gender, phone, delete_flag, create_time, update_time
    from userinfo where id= #{id}
</select>
<select id="queryByCondition" resultType="com.example.demo.model.UserInfo">
    select id, username, age, gender, phone, delete_flag, create_time, update_time
    from userinfo
    <where>
        <if test="age != null">
            and age = #{age}
        </if>
        <if test="gender != null">
            and gender = #{gender}
        </if>
        <if test="deleteFlag != null">
            and delete_flag = #{deleteFlag}
        </if>
    </where>
</select>

```

我们可以对重复的代码片段进行抽取，将其通过 `<sql>` 标签封装到一个SQL片段，然后再通过 `<include>` 标签进行引用。

- `<sql>`：定义可重用的SQL片段
- `<include>`：通过属性`refid`，指定包含的SQL片段

```

1 <sql id="allColumn">
2     id, username, age, gender, phone, delete_flag, create_time, update_time
3 </sql>

```

通过 `<include>` 标签在原来抽取的地方进行引用。操作如下：

```

1 <select id="queryAllUser" resultMap="BaseMap">
2     select
3     <include refid="allColumn"></include>
4     from userinfo
5 </select>

```

```
6 <select id="queryById" resultType="com.example.demo.model.UserInfo">
7     select
8     <include refid="allColumn"></include>
9     from userinfo where id= #{id}
10 </select>
```

2. 案例练习

基于以上知识的学习, 我们就可以做一些简单的项目了

案例中配置文件, 统一使用yaml格式来进行讲解

2.1 表白墙

前面的案例中, 我们写了表白墙, 但是一旦服务器重启, 数据仍然会丢失.

要想数据不丢失, 需要把数据存储到数据库中. 接下来咱们借助MyBatis来实现数据的操作

表白墙

输入后点击提交, 会将信息显示在表格中

谁:

对谁:

说什么:

提交

2.1.1 数据准备

```
1 DROP TABLE IF EXISTS message_info;
2 CREATE TABLE `message_info` (
3     `id` INT ( 11 ) NOT NULL AUTO_INCREMENT,
4     `from` VARCHAR ( 127 ) NOT NULL,
5     `to` VARCHAR ( 127 ) NOT NULL,
6     `message` VARCHAR ( 256 ) NOT NULL,
7     `delete_flag` TINYINT ( 4 ) DEFAULT 0 COMMENT '0-正常, 1-删除',
```

```
8      `create_time` DATETIME DEFAULT now(),
9      `update_time` DATETIME DEFAULT now() ON UPDATE now(),
10 PRIMARY KEY ( `id` )
11 ) ENGINE = INNODB DEFAULT CHARSET = utf8mb4;
```

ON UPDATE now(): 当数据发生更新操作时, 自动把该列的值设置为now(),

now() 可以替换成其他获取时间的标识符, 比如: CURRENT_TIMESTAMP(), LOCALTIME() 等

MySQL <5.6.5

1. 只有TIMESTAMP支持自动更新
2. 一个表只能有一列设置自动更新
3. 不允许同时存在两个列, 其中一列设置了DEFAULT CURRENT_TIMESTAMP, 另一个设置了ON UPDATE CURRENT_TIMESTAMP

MySQL >=5.6.5

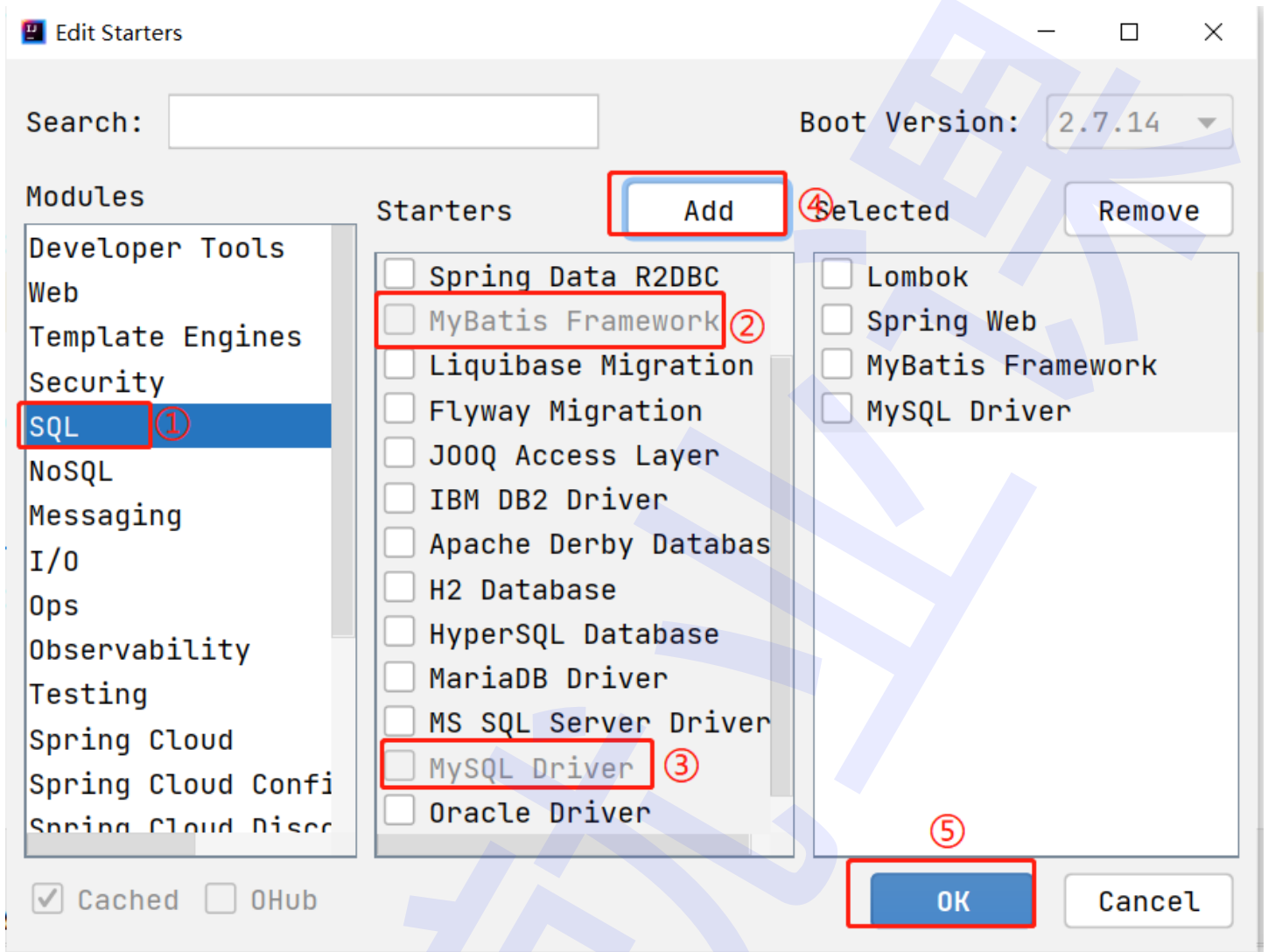
1. TIMESTAMP 和DATETIME都支持自动更新,且可以有多列

2.1.2 引入MyBatis 和 MySQL驱动依赖

修改pom文件

```
1 <dependency>
2   <groupId>org.mybatis.spring.boot</groupId>
3   <artifactId>mybatis-spring-boot-starter</artifactId>
4   <version>3.0.3</version>
5 </dependency>
6 <dependency>
7   <groupId>com.mysql</groupId>
8   <artifactId>mysql-connector-j</artifactId>
9   <scope>runtime</scope>
10 </dependency>
```

或者使用插件EditStarters来引入依赖



2.1.3 配置MySQL账号密码

```
1 spring:
2   datasource:
3     url: jdbc:mysql://127.0.0.1:3306/mybatis_test?
4       characterEncoding=utf8&useSSL=false
5     username: root
6     password: root
7     driver-class-name: com.mysql.cj.jdbc.Driver
8 mybatis:
9   configuration: # 配置打印 MyBatis日志
10    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
11    map-underscore-to-camel-case: true #配置驼峰自动转换
```

2.1.4 编写后端代码

Model

```
1 import lombok.Data;
```

```
2
3 @Data
4 public class MessageInfo {
5     private Integer id;
6     private String from;
7     private String to;
8     private String message;
9     private Integer deleteFlag;
10    private Date createTime;
11    private Date updateTime;
12 }
```

MessageInfoMapper

```
1 import com.example.demo.model.MessageInfo;
2 import org.apache.ibatis.annotations.Insert;
3 import org.apache.ibatis.annotations.Mapper;
4 import org.apache.ibatis.annotations.Select;
5
6 import java.util.List;
7
8 @Mapper
9 public interface MessageInfoMapper {
10     @Select("select `id`, `from`, `to`, `message` from message_info where delete_flag=0")
11     List<MessageInfo> queryAll();
12
13     @Insert("insert into message_info (`from`,`to`, `message`) values(#{from},#{to},#{message})")
14     Integer addMessage(MessageInfo messageInfo);
15 }
```

MessageInfoService

```
1 import com.example.demo.mapper.MessageInfoMapper;
2 import com.example.demo.model.MessageInfo;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 import java.util.List;
7
8 @Service
9 public class MessageInfoService {
```

```

10     @Autowired
11     private MessageInfoMapper messageInfoMapper;
12
13     public List<MessageInfo> queryAll() {
14         return messageInfoMapper.queryAll();
15     }
16
17     public Integer addMessage(MessageInfo messageInfo) {
18         return messageInfoMapper.addMessage(messageInfo);
19     }
20 }

```

MessageController

```

1  import com.example.demo.model.MessageInfo;
2  import com.example.demo.service.MessageInfoService;
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.util.StringUtils;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.RestController;
7
8  import java.util.List;
9
10 @RequestMapping("/message")
11 @RestController
12 public class MessageController {
13     @Autowired
14     private MessageInfoService messageInfoService;
15     /**
16      * 获取留言列表
17      * @return
18      */
19     @RequestMapping("/getList")
20     public List<MessageInfo> getList() {
21         return messageInfoService.queryAll();
22     }
23
24     /**
25      * 发表留言
26      * @param messageInfo
27      * @return
28      */
29     @RequestMapping("/publish")
30     public boolean publish(MessageInfo messageInfo) {
31         System.out.println(messageInfo);

```

```
32         if (StringUtils.hasLength(messageInfo.getFrom())
33             && StringUtils.hasLength(messageInfo.getTo())
34             && StringUtils.hasLength(messageInfo.getMessage())) {
35             messageInfoService.addMessage(messageInfo);
36             return true;
37         }
38
39         return false;
40     }
41 }
```

2.1.5 测试

部署程序, 验证服务器是否能正确响应: <http://127.0.0.1:8080/messagewall.html>

留言板

输入后点击提交, 会将信息显示下方空白处

谁:

对谁:

说什么:

提交

输入留言信息, 点击提交, 发现页面列表显示新的数据, 并且数据库中也添加了一条记录.

留言板

输入后点击提交, 会将信息显示下方空白处

谁:

对谁:

说什么:

提交

青蛙对乌鸦说:呱呱

开始事务	文本	筛选	排序	列	导入	导出	数据生成	创建图表
id	from	to	message	delete_flag	create_time	update_time		
1	青蛙	乌鸦	呱呱	0	2023-09-24 16:2	2023-09-24 16:2		

重启服务, 页面显示不变.

2.2 图书管理系统

前面图书管理系统, 咱们只完成了用户登录和图书列表, 并且数据是Mock的. 接下来我们把其他功能进行完善.

图书列表展示

添加图书

批量删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	0	书籍0	作者0	3	62	出版社0	可借阅	修改 删除
<input type="checkbox"/>	1	书籍1	作者1	8	15	出版社1	可借阅	修改 删除
<input type="checkbox"/>	2	书籍2	作者2	13	24	出版社2	可借阅	修改 删除
<input type="checkbox"/>	3	书籍3	作者3	18	90	出版社3	可借阅	修改 删除
<input type="checkbox"/>	4	书籍4	作者4	23	86	出版社4	可借阅	修改 删除

首页 上一页 1 2 3 4 5 下一页 最后一页

功能列表:

1. 用户登录
2. 图书列表
3. 图书的增删改查
4. 翻页功能

2.2.1 数据库表设计

数据库表是应用程序开发中的一个重要环节, 数据库表的设计往往会决定我们的应用需求是否能顺利实现, 甚至决定我们的实现方式. 如何设计表以及这些表有哪些字段, 这些表存在哪些关系 也是非常重要的.

数据库表设计是依据业务需求来设计的. 如何设计出优秀的数据库表, 与经验有很大关系.

数据库表**通常**分两种: 实体表和关系表.

分析我们的需求, 图书管理系统相对来说比较简单, 只有两个实体: 用户和图书, 并且用户和图书之间没有关联关系

表的具体字段设计, 也与需求有关.

用户表有用户名和密码即可(复杂的业务可能还涉及昵称, 年龄等资料)

图书表有哪些字段, 也是参考需求页面(通常不是一个页面决定的, 而是要对整个系统进行全面分析观察后定的)

创建数据库 book_test

```
1  -- 创建数据库
2  DROP DATABASE IF EXISTS book_test;
3
4  CREATE DATABASE book_test DEFAULT CHARACTER SET utf8mb4;
5
6  -- 用户表
7  DROP TABLE IF EXISTS user_info;
8  CREATE TABLE user_info (
9      `id` INT NOT NULL AUTO_INCREMENT,
10     `user_name` VARCHAR ( 128 ) NOT NULL,
11     `password` VARCHAR ( 128 ) NOT NULL,
12     `delete_flag` TINYINT ( 4 ) NULL DEFAULT 0,
13     `create_time` DATETIME DEFAULT now(),
14     `update_time` DATETIME DEFAULT now() ON UPDATE now(),
```

```

15     PRIMARY KEY ( `id` ),
16 UNIQUE INDEX `user_name_UNIQUE` ( `user_name` ASC )) ENGINE = INNODB DEFAULT
    CHARACTER
17 SET = utf8mb4 COMMENT = '用户表';
18
19 -- 图书表
20 DROP TABLE IF EXISTS book_info;
21 CREATE TABLE `book_info` (
22     `id` INT ( 11 ) NOT NULL AUTO_INCREMENT,
23     `book_name` VARCHAR ( 127 ) NOT NULL,
24     `author` VARCHAR ( 127 ) NOT NULL,
25     `count` INT ( 11 ) NOT NULL,
26     `price` DECIMAL (7,2 ) NOT NULL,
27     `publish` VARCHAR ( 256 ) NOT NULL,
28     `status` TINYINT ( 4 ) DEFAULT 1 COMMENT '0-无效, 1-正常, 2-不允许借阅',
29     `create_time` DATETIME DEFAULT now(),
30     `update_time` DATETIME DEFAULT now() ON UPDATE now(),
31 PRIMARY KEY ( `id` )
32 ) ENGINE = INNODB DEFAULT CHARSET = utf8mb4;
33
34 -- 初始化数据
35 INSERT INTO user_info ( user_name, PASSWORD ) VALUES ( "admin", "admin" );
36 INSERT INTO user_info ( user_name, PASSWORD ) VALUES ( "zhangsan", "123456" );
37
38 -- 初始化图书数据
39 INSERT INTO `book_info` (book_name,author,count, price, publish) VALUES ('活
    着', '余华', 29, 22.00, '北京文艺出版社');
40 INSERT INTO `book_info` (book_name,author,count, price, publish) VALUES ('平凡的世界', '路遥', 5, 98.56, '北京十月文艺出版社');
41 INSERT INTO `book_info` (book_name,author,count, price, publish) VALUES ('三体', '刘慈欣', 9, 102.67, '重庆出版社');
42 INSERT INTO `book_info` (book_name,author,count, price, publish) VALUES ('金字塔原理', '麦肯锡', 16, 178.00, '民主与建设出版社');

```

2.2.2 引入MyBatis和MySQL 驱动依赖

修改pom文件

```

1 <dependency>
2     <groupId>org.mybatis.spring.boot</groupId>
3     <artifactId>mybatis-spring-boot-starter</artifactId>
4     <version>3.0.3</version>
5 </dependency>
6 <dependency>
7     <groupId>com.mysql</groupId>

```

```
8     <artifactId>mysql-connector-j</artifactId>
9     <scope>runtime</scope>
10 </dependency>
```

2.2.3 配置数据库&日志

```
1 # 数据库连接配置
2 spring:
3     datasource:
4         url: jdbc:mysql://127.0.0.1:3306/book_test?
           characterEncoding=utf8&useSSL=false
5         username: root
6         password: root
7         driver-class-name: com.mysql.cj.jdbc.Driver
8 mybatis:
9     configuration:
10         map-underscore-to-camel-case: true #配置驼峰自动转换
11         log-impl: org.apache.ibatis.logging.stdout.StdOutImpl #打印sql语句
12 # 设置日志文件的文件名
13 logging:
14     file:
15         name: spring-book.log
16
```

2.2.4 Model创建

```
1 import lombok.Data;
2
3 import java.util.Date;
4
5 @Data
6 public class UserInfo {
7     private Integer id;
8     private String userName;
9     private String password;
10    private Integer deleteFlag;
11    private Date createTime;
12    private Date updateTime;
13 }
```

```
1 @Data
```

```

2 public class BookInfo {
3     //图书ID
4     private Integer id;
5     //书名
6     private String bookName;
7     //作者
8     private String author;
9     //数量
10    private Integer count;
11    //定价
12    private BigDecimal price;
13    //出版社
14    private String publish;
15    //状态 0-无效 1-允许借阅 2-不允许借阅
16    private Integer status;
17
18    private String statusCN;
19    //创建时间
20    private Date createTime;
21    //更新时间
22    private Date updateTime;
23 }

```

2.2.5 用户登录

约定前后端交互接口

```

1 [请求]
2 /user/login
3 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
4
5
6 [参数]
7 name=zhangsan&password=123456
8
9 [响应]
10 true //账号密码验证正确, 否则返回false

```

浏览器给服务器发送 `/user/login` 这样的 HTTP 请求, 服务器给浏览器返回了一个 Boolean 类型的数据. 返回 true, 表示 账号密码验证正确

实现服务器代码

控制层:

从数据库中, 根据名称查询用户, 如果可以查到, 并且密码一致, 就认为登录成功

创建UserController

```
1 import com.example.demo.model.UserInfo;
2 import com.example.demo.service.UserService;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.util.StringUtils;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 import jakarta.servlet.http.HttpSession;
9
10 @RequestMapping("/user")
11 @RestController
12 public class UserController {
13     @Autowired
14     private UserService userService;
15
16     @RequestMapping("/login")
17     public boolean login(String name, String password, HttpSession session){
18         //账号或密码为空
19         if (!StringUtils.hasLength(name) || !StringUtils.hasLength(password)){
20             return false;
21         }
22         UserInfo userInfo = userService.queryUserByName(name);
23         if (userInfo == null){
24             return false;
25         }
26         //账号密码正确
27         if(userInfo!=null && password.equals(userInfo.getPassword())){
28             //存储在Session中
29             userInfo.setPassword("");
30             session.setAttribute("session_user_key",userInfo);
31             return true;
32         }
33         //账号密码错误
34         return false;
35     }
36 }
```

业务层:

创建UserService

```

1 import com.example.demo.mapper.UserInfoMapper;
2 import com.example.demo.model.UserInfo;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class UserService {
8     @Autowired
9     private UserInfoMapper userInfoMapper;
10
11     public UserInfo queryUserByName(String name) {
12         return userInfoMapper.queryUserByName(name);
13     }
14 }

```

数据层:

创建UserInfoMapper

```

1 import com.example.demo.model.UserInfo;
2 import org.apache.ibatis.annotations.Mapper;
3 import org.apache.ibatis.annotations.Select;
4
5 @Mapper
6 public interface UserInfoMapper {
7     @Select("select id, user_name, `password`, delete_flag, create_time,
8         update_time " +
9         "from user_info where delete_flag=0 and user_name=#{name}")
10     UserInfo queryUserByName(String name);

```

访问数据库, 使用MyBatis来实现, 所以把之前dao路径下的文件可以删掉, 用mapper目录来代替, 创建UserInfoMapper

当然, 继续使用dao目录也可以, 此处为建议

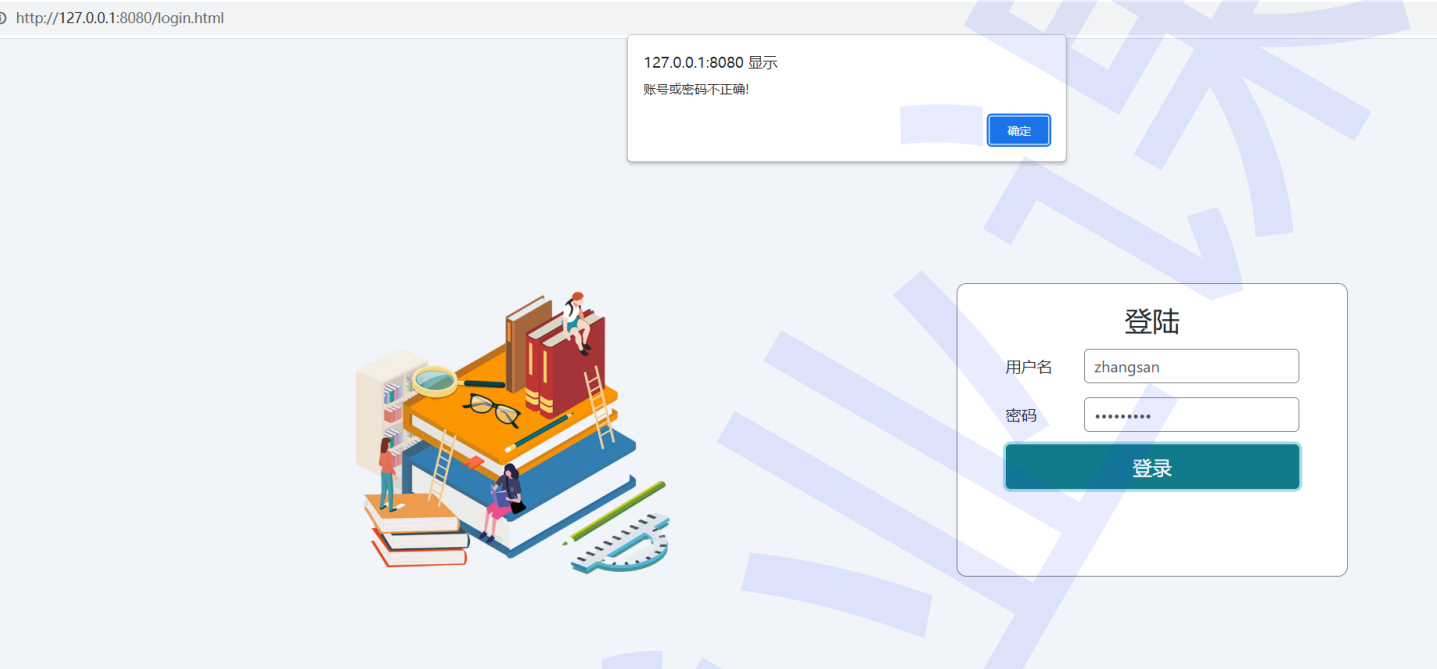
dao和mapper通常都被认为是数据库层

测试

部署程序, 验证服务器是否能正确返回数据 (使用 URL <http://127.0.0.1:8080/user/login?name=admin&password=admin> 即可).

联动前端一起测试:

输入错误的用户名和密码, 页面弹窗警告



输入正确的用户名和密码, 页面正常跳转

图书列表展示

添加图书

批量删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	0	书籍0	作者0	3	21	出版社0	可借阅	修改 删除
<input type="checkbox"/>	1	书籍1	作者1	8	67	出版社1	可借阅	修改 删除
<input type="checkbox"/>	2	书籍2	作者2	13	44	出版社2	可借阅	修改 删除
<input type="checkbox"/>	3	书籍3	作者3	18	25	出版社3	可借阅	修改 删除
<input type="checkbox"/>	4	书籍4	作者4	23	57	出版社4	可借阅	修改 删除

首页

上一页

1

2

3

4

5

下一页

最后一页

2.2.6 添加图书

约定前后端交互接口

```
1 [请求]
2 /book/addBook
3 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
4
```

```
5 [参数]
6 bookName=图书1&author=作者1&count=23&price=36&publish=出版社1&status=1
7
8 [响应]
9 "" //失败信息, 成功时返回空字符串
```

我们约定, 浏览器给服务器发送一个 `/book/addBook` 这样的 HTTP 请求, form 表单的形式来提交数据

服务器返回处理结果, 返回""表示添加图书成功, 否则, 返回失败信息.

实现服务器代码

控制层:

在BookController补充代码

先进行参数校验, 校验通过了进行图书添加

实际开发中, 后端开发人员不关注前端是否进行了参数校验, 一律进行校验

原因是: 后端接口可能会被黑客攻击, 不通过前端来访问, 如果后端不进行校验, 会产生脏数据.

(咱们学习阶段, 暂不涉及安全领域模块的开发, 防攻击一般是企业统一来做)

```
1 @RequestMapping("/addBook")
2 public String addBook(BookInfo bookInfo) {
3     log.info("添加图书:{}", bookInfo);
4     if (!StringUtils.hasLength(bookInfo.getBookName())
5         || !StringUtils.hasLength(bookInfo.getAuthor())
6         || bookInfo.getCount() == null
7         || bookInfo.getPrice() == null
8         || !StringUtils.hasLength(bookInfo.getPublish())
9         || bookInfo.getStatus() == null
10    ) {
11         return "输入参数不合法, 请检查入参!";
12    }
13    try {
14        bookService.addBook(bookInfo);
15        return "";
16    } catch (Exception e) {
17        log.error("添加图书失败", e);
18        return e.getMessage();
19    }
20 }
```


业务层:

在BookService中补充代码

```
1 public void addBook(BookInfo bookInfo) {  
2     bookInfoMapper.insertBook(bookInfo);  
3 }
```

数据层:

创建BookInfoMapper文件

```
1 @Mapper  
2 public interface BookInfoMapper {  
3     @Insert("insert into book_info  
4         (book_name,author,count,price,publish,status) " +  
5         "values (#{bookName},#{author},#{count},#{price},#{publish},#  
6         {status})")  
7     Integer insertBook(BookInfo bookInfo);  
8 }
```

创建新的BookInfoMapper文件

实现客户端代码

提供的前面页面中,js已经提前留了空位

```
1 <button type="button" class="btn btn-info btn-lg" onclick="add()">确定</button>
```

点击确定按钮, 会执行 `add()` 方法

补全add()的方法

提交整个表单的数据: `$("#addBook").serialize()`

提交的内容格式: `bookName=图书1&author=作者1&count=23&price=36&publish=出版社1&status=1`

被form标签包括的所有输入表单(input,select)内容都会被提交

```
1 function add() {
```

```
2     $.ajax({
3         type: "post",
4         url: "/book/addBook",
5         data: $("#addBook").serialize(),
6         success: function (result) {
7             if (result == "") {
8                 location.href = "book_list.html"
9             } else {
10                console.log(result);
11                alert("添加失败:" + result);
12            }
13        },
14        error: function (error) {
15            console.log(error);
16        }
17    });
18 }
```

测试

添加图书前, 数据库内容:

```
1 select * from book_info;
```

信息	摘要	结果 1	剖析	状态					
id	book_name	author	count	price	publish	status	create_time	update_time	
1	活着	余华	29	22.00	北京文艺出	1	2023-09-04 11:20:05	2023-09-04 11:2	
2	平凡的世界	路遥	5	98.56	北京十月文	1	2023-09-04 11:20:05	2023-09-04 11:2	
3	三体	刘慈欣	9	102.67	重庆出版社	1	2023-09-04 11:20:05	2023-09-04 11:2	
4	金字塔原理	麦肯锡	16	178.00	民主与建设	1	2023-09-04 11:20:05	2023-09-04 11:2	

点击添加图书按钮, 跳转到添加图书的页面, 填写图书信息

添加图书

图书名称:

图书作者:

图书库存:

图书定价:

出版社:

图书状态:

可借阅

确定

返回

添加图书

图书名称:

图书作者:

图书库存:

图书定价:

出版社:

图书状态:

可借阅

确定

返回

填写图书信息

点击确定按钮, 页面跳转到图书列表页

图书列表还未实现, 页面上看不出来效果.

查看数据库数据:

```
1 select * from book_info;
```

数据插入成功

信息	摘要	结果 1	剖析	状态					
id	book_name	author	count	price	publish	status	create_time	update_time	
1	活着	余华	29	22.00	北京文艺出	1	2023-09-04 11:20:05	2023-09-04 11:2	
2	平凡的世界	路遥	5	98.56	北京十月文	1	2023-09-04 11:20:05	2023-09-04 11:2	
3	三体	刘慈欣	9	102.67	重庆出版社	1	2023-09-04 11:20:05	2023-09-04 11:2	
4	金字塔原理	麦肯锡	16	178.00	民主与建设	1	2023-09-04 11:20:05	2023-09-04 11:2	
5	图书1	作者1	23	65.00	出版社1	1	2023-09-04 11:39:16	2023-09-04 11:3	

测试输入不合法的场景, 比如什么信息都不填, 直接点击确定

127.0.0.1:8080 显示

添加失败:输入参数不合法, 请检查入参!

确定

添加图书

图书名称:

请输入图书名称

图书作者

请输入图书作者

图书库存

请输入图书库存

图书定价:

请输入价格

出版社

请输入图书出版社

图书状态

可借阅

确定

返回

页面也得到正确响应

2.2.7 图书列表

可以看到, 添加图书之后, 跳转到图书列表页面, 并没有显示刚才添加的图书信息, 接下来我们来实现图书列表

需求分析

我们之前做的表白墙查询功能, 是将数据库中所有的数据查询出来并展示到页面上, 试想如果数据库中的数据有很多(假设有十几万条)的时候, 将数据全部展示出来肯定不现实, 那如何解决这个问题呢?

使用分页解决这个问题。每次只展示一页的数据, 比如: 一页展示10条数据, 如果还想看其他的数据, 可以通过点击页码进行查询

图书列表展示

书名: 作者:

搜索图书 添加图书 批量删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
----	------	----	----	----	----	-----	----	----

首页

上一页

1

2

3

4

5

下一页

最后一页

分页时, 数据是如何展示的呢

第1页: 显示1-10 条的数据

第2页: 显示11-20 条的数据

第3页: 显示21-30 条的数据

以此类推...

要想实现这个功能, 从数据库中进行分页查询, 我们要使用 `LIMIT` 关键字, 格式为: `limit 开始索引 每页显示的条数(开始索引从0开始)`

我们先伪造更多的数据:

```
1 INSERT INTO `book_info` ( book_name, author, count, price, publish )
2 VALUES
3 ( '图书2', '作者2', 29, 22.00, '出版社2' ),( '图书3', '作者2', 29, 22.00, '出版社3' ),
4 ( '图书4', '作者2', 29, 22.00, '出版社1' ),( '图书5', '作者2', 29, 22.00, '出版社1' ),
5 ( '图书6', '作者2', 29, 22.00, '出版社1' ),( '图书7', '作者2', 29, 22.00, '出版社1' ),
6 ( '图书8', '作者2', 29, 22.00, '出版社1' ),( '图书9', '作者2', 29, 22.00, '出版社1' ),
7 ( '图书10', '作者2', 29, 22.00, '出版社1' ),( '图书11', '作者2', 29, 22.00, '出版社1' ),
8 ( '图书12', '作者2', 29, 22.00, '出版社1' ),( '图书13', '作者2', 29, 22.00, '出版社1' ),
9 ( '图书14', '作者2', 29, 22.00, '出版社1' ),( '图书15', '作者2', 29, 22.00, '出版社1' ),
10 ( '图书16', '作者2', 29, 22.00, '出版社1' ),( '图书17', '作者2', 29, 22.00, '出版社1' ),
11 ( '图书18', '作者2', 29, 22.00, '出版社1' ),( '图书19', '作者2', 29, 22.00, '出版社1' ),
```

```
12 ( '图书20', '作者2', 29, 22.00, '出版社1'), ( '图书21', '作者2', 29, 22.00, '出版社1');
```

查询第1页的SQL语句

```
1 SELECT * FROM book_info LIMIT 0,10
```

查询第2页的SQL语句

```
1 SELECT * FROM book_info LIMIT 10,10
```

查询第3页的SQL语句

```
1 SELECT * FROM book_info LIMIT 20,10
```

观察以上SQL语句，发现：开始索引一直在改变，每页显示条数是固定的

开始索引的计算公式：开始索引 = (当前页码 - 1) * 每页显示条数

我们继续基于前端页面，继续分析，得出以下结论：

1. 前端在发起查询请求时，需要向服务端传递的参数

- currentPage 当前页码 //默认值为1
- pageSize 每页显示条数 //默认值为10

为了项目更好的扩展性，通常不设置固定值，而是以参数的形式来进行传递

扩展性：软件系统具备面对未来需求变化而进行扩展的能力

比如当前需求一页显示10条，后期需求改为一页显示20条，后端代码不需要任何修改

2. 后端响应时，需要响应给前端的数据

- records 所查询到的数据列表(存储到List 集合中)
- total 总记录数 (用于告诉前端显示多少页，显示页数为： $(total + pageSize - 1) / pageSize$)

显示页数totalPage 计算公式为： $total \% pagesize == 0 ? total / pagesize : (total / pagesize) + 1 ;$

pagesize - 1 是 total / pageSize 的最大的余数，所以 $(total + \text{pagesize} - 1) / \text{pagesize}$ 就得到总页数

翻页请求和响应部分, 我们通常封装在两个对象中

翻页请求对象

```
1 @Data
2 public class PageRequest {
3     private int currentPage = 1; // 当前页
4     private int pageSize = 10; // 每页中的记录数
5
6 }
```

我们需要根据currentPage 和pageSize ,计算出来开始索引

PageRequest修改为:

```
1 @Data
2 public class PageRequest {
3     private int currentPage = 1; // 当前页
4     private int pageSize = 10; // 每页中的记录数
5     private int offset;
6
7     public int getOffset() {
8         return (currentPage-1) * pageSize;
9     }
10 }
```

翻页列表结果类:

```
1 import lombok.Data;
2
3 import java.util.List;
4
5 @Data
6 public class PageResult<T> {
7     private int total; // 所有记录数
8     private List<T> records; // 当前页数据
9
10     public PageResult(Integer total, List<T> records) {
11         this.total = total;
```

```
12         this.records = records;
13     }
14 }
```

返回结果中, 使用泛型来定义记录的类型

基于以上分析, 我们来约定前后端交互接口

约定前后端交互接口

```
1  [请求]
2  /book/getListByPage?currentPage=1&pageSize=10
3  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
4
5  [参数]
6
7
8  [响应]
9  Content-Type: application/json
10
11 {
12     "total": 25,
13     "records": [{
14         "id": 25,
15         "bookName": "图书21",
16         "author": "作者2",
17         "count": 29,
18         "price": 22.00,
19         "publish": "出版社1",
20         "status": 1,
21         "statusCN": "可借阅"
22     }, {
23         .....
24     } ]
25 }
```

我们约定, 浏览器给服务器发送一个 `/book/getListByPage` 这样的 HTTP 请求, 通过 `currentPage` 参数告诉服务器, 当前请求为第几页的数据, 后端根据请求参数, 返回对应页的数据

第一页可以不传参数, `currentPage` 默认值为 1

实现服务器代码

控制层:

完善 BookController


```

1 @RequestMapping("/getListByPage")
2 public PageResult<BookInfo> getListByPage(PageRequest pageRequest) {
3     log.info("获取图书列表, pageRequest:{})", pageRequest);
4     PageResult<BookInfo> pageResult =
        bookService.getBookListByPage(pageRequest);
5     return pageResult;
6 }

```

业务层:

BookService

```

1 public PageResult<BookInfo> getBookListByPage(PageRequest pageRequest) {
2     Integer count = bookInfoMapper.count();
3     List<BookInfo> books = bookInfoMapper.queryBookListByPage(pageRequest);
4     for (BookInfo book:books){
5         if (book.getStatus()==0){
6             book.setStatusCN("无效");
7         }else if (book.getStatus()==1){
8             book.setStatusCN("可借阅");
9         }else {
10             book.setStatusCN("不可借阅");
11         }
12     }
13     return new PageResult<>(count,books);
14 }

```

1. 翻页信息需要返回数据的总数和列表信息, 需要查两次SQL
2. 图书状态: 图书状态和数据库存储的status有一定的对应关系

如果后续状态码有变动, 我们需要修改项目中所有涉及的代码, 这种情况, 通常采用枚举类来处理映射关系

创建enums目录, 创建BookStatus类:

```

1 public enum BookStatus {
2     DELETED(0,"无效"),
3     NORMAL(1,"可借阅"),
4     FORBIDDEN(2,"不可借阅");
5
6     private Integer code;
7     private String name;
8 }

```

```

9      BookStatus(int code, String name) {
10          this.code = code;
11          this.name = name;
12      }
13
14      public static BookStatus getNameByCode(Integer code){
15          switch (code){
16              case 0: return DELETED;
17              case 1: return NORMAL;
18              case 2: return FORBIDDEN;
19          }
20          return null;
21      }
22
23      public Integer getCode() {
24          return code;
25      }
26
27      public void setCode(Integer code) {
28          this.code = code;
29      }
30
31      public String getName() {
32          return name;
33      }
34
35      public void setName(String name) {
36          this.name = name;
37      }
38  }

```

getNameByCode: 通过code来获取对应的枚举, 以获取该枚举对应的中文名称

后续如果有状态变更, 只需要修改该枚举类即可

此时, BookService的代码, 可以修改如下:

```

1  public PageResult<BookInfo> getBookListByPage(PageRequest pageRequest) {
2      Integer count = bookInfoMapper.count();
3      List<BookInfo> books = bookInfoMapper.queryBookListByPage(pageRequest);
4      for (BookInfo book:books){
5          book.setStatusCN(BookStatus.getNameByCode(book.getStatus()).getName());
6      }
7      return new PageResult<>(count,books);
8  }

```

数据层:

翻页查询SQL

```
1 select * from book_info where status !=0 order by id desc limit #{offset}, #{pageSize}
```

其中offset 在PageRequest类中已经给赋值

BookInfoMapper

图书列表按id降序排列

```
1 @Select("select count(1) from book_info where status<>0")
2 Integer count();
3
4 @Select("select * from book_info where status !=0 order by id desc limit #{offset}, #{pageSize}")
5 List<BookInfo> queryBookListByPage(PageRequest pageRequest);
```

启动服务, 访问后端程序:

<http://127.0.0.1:8080/book/getListByPage> 返回1-10条记录 (按id降序)

<http://127.0.0.1:8080/book/getListByPage?currentPage=2> 返回11-20条记录

实现客户端代码

我们定义:

访问第一页图书的前端url为: http://127.0.0.1:8080/book_list.html?currentPage=1

访问第二页列表的url为: http://127.0.0.1:8080/book_list.html?currentPage=2

浏览器访问 book_list.html 页面时, 就去请求后端, 把后端返回数据显示在页面上

调用后端请求: `/book/getListByPage?currentPage=1`

修改之前的js, 后端请求方法从 `/book/getList` 改为 `/book/getListByPage?currentPage=1`

修改后的js为

```

1 //获取图书列表
2 getBookList();
3 function getBookList() {
4     $.ajax({
5         type: "get",
6         url: "/book/getListByPage",
7         success: function (result) {
8             if (result != null) {
9                 var finalHtml = "";
10                for (var book of result) {
11                    finalHtml += '<tr>';
12                    finalHtml += '<td><input type="checkbox" name="selectBook"
13value="'+book.id+'" id="selectBook" class="book-select"></td>';
14                    finalHtml += '<td>'+book.id+'</td>';
15                    finalHtml += '<td>'+book.bookName+'</td>';
16                    finalHtml += '<td>'+book.author+'</td>';
17                    finalHtml += '<td>'+book.count+'</td>';
18                    finalHtml += '<td>'+book.price+'</td>';
19                    finalHtml += '<td>'+book.publish+'</td>';
20                    finalHtml += '<td>'+book.statusCN+'</td>';
21                    finalHtml += '<td><div class="op">';
22                    finalHtml += '<a href="book_update.html?
23bookId='+book.id+'">修改</a>';
24                    finalHtml += '<a href="javascript:void(0)"
25onclick="deleteBook('+book.id+')">删除</a>';
26                    finalHtml += '</div></td>';
27                    finalHtml += "</tr>";
28                }
29            }
30        });
31    }

```

此时, url还未设置 `currentPage` 参数

我们直接使用 `location.search` 从url中获取参数信息即可

`location.search`: 获取url的查询字符串 (包含问号)

如:

url: `http://127.0.0.1:8080/book_list.html?currentPage=1`

`location.search`: `?currentPage=1`

所以,把上述url改为: `"/book/getListByPage" + location.search`

```
1 //获取图书列表
2 function getBookList() {
3     $.ajax({
4         type: "get",
5         url: "/book/getListByPage" + location.search,
6         //...
7     });
8 }
```

接下来处理分页信息

分页插件

本案例中,分页代码采用了一个分页组件

分页组件文档介绍: [jqPaginator分页组件](#)

使用时,只需要按照 [使用说明]部分的文档,把代码复制粘贴进来就可以了(提供的前端代码中,已经包含该部分内容)简单介绍下使用

```
<div class="demo">
|   <ul id="pageContainer" class="pagination justify-content-center"></ul>
</div>

//翻页信息
$( "#pageContainer" ).jqPaginator({
    totalCounts: 100, //总记录数
    pageSize: 10,     //每页的个数
    visiblePages: 5,  //可视页数
    currentPage: 1,   //当前页码
    first: '<li class="page-item"><a class="page-link">首页</a></li>',
    prev: '<li class="page-item"><a class="page-link" href="javascript:void(0);">上一页</a></li>',
    next: '<li class="page-item"><a class="page-link" href="javascript:void(0);">下一页</a></li>',
    last: '<li class="page-item"><a class="page-link" href="javascript:void(0);">最后一页</a></li>',
    page: '<li class="page-item"><a class="page-link" href="javascript:void(0);">{{page}}</a></li>',
    //页面初始化和页码点击时都会执行
    onPageChange: function (page, type) {
        console.log("第" + page + "页, 类型:" + type);
    }
});
```

保持一致

`onPageChange`:回调函数,当换页时触发(包括初始化第一页的时候),会传入两个参数:

- 1、"目标页"的页码, `Number`类型
- 2、触发类型,可能的值: `"init"` (初始化), `"change"` (点击分页)

我们在图书列表信息加载之后, 需要分页信息, 同步加载

分页组件需要提供一些信息: totalCounts: 总记录数, pageSize: 每页的个数, visiblePages: 可视页数
currentPage: 当前页码

这些信息中, pageSize 和 visiblePages 前端直接设置即可. totalCounts 后端已经提供, currentPage 也可以从参数中取到, 但太复杂了, 咱们直接由后端返回即可.

修改后端代码

1. 为避免后续还需要其他请求处的信息, 我们直接在PageResult 添加 PageRequest 属性
2. 处理返回结果, 填充 PageRequest

PageResult.java

```
1 @Data
2 public class PageResult<T> {
3     private int total; //所有记录数
4     private List<T> records; // 当前页数据
5     private PageRequest pageRequest;
6
7     public PageResult(Integer total, PageRequest pageRequest, List<T> records)
8     {
9         this.total = total;
10        this.pageRequest = pageRequest;
11        this.records = records;
12    }
13 }
```

BookService.java

```
1 public PageResult<BookInfo> getBookListByPage(PageRequest pageRequest) {
2     Integer count = bookInfoMapper.count();
3     List<BookInfo> books = bookInfoMapper.queryBookListByPage(pageRequest);
4     for (BookInfo book:books){
5         book.setStatusCN(BookStatus.getNameByCode(book.getStatus()).getName());
6     }
7     return new PageResult<>(count,pageRequest, books);
8 }
```

后端数据返回后, 我们加载页面信息, 把分页代码挪到getBookList方法中

```

1 function getBookList(currentPage) {
2     $.ajax({
3         type: "get",
4         url: "/book/getListByPage?currentPage="+currentPage,
5         success: function (result) {
6             console.log(result);
7             if (result != null) {
8                 var finalHtml = "";
9                 for (var book of result.records) {
10                     finalHtml += '<tr>';
11                     finalHtml += '<td><input type="checkbox" name="selectBook"
value="" + book.id + ' id="selectBook" class="book-select"></td>';
12                     finalHtml += '<td>' + book.id + '</td>';
13                     finalHtml += '<td>' + book.bookName + '</td>';
14                     finalHtml += '<td>' + book.author + '</td>';
15                     finalHtml += '<td>' + book.count + '</td>';
16                     finalHtml += '<td>' + book.price + '</td>';
17                     finalHtml += '<td>' + book.publish + '</td>';
18                     finalHtml += '<td>' + book.statusCN + '</td>';
19                     finalHtml += '<td><div class="op">';
20                     finalHtml += '<a href="book_update.html?bookId=' + book.id
+ '>修改</a>';
21                     finalHtml += '<a href="javascript:void(0)"
onclick="deleteBook(' + book.id + ')">删除</a>';
22                     finalHtml += '</div></td>';
23                     finalHtml += "</tr>";
24                 }
25
26                 $("tbody").html(finalHtml);
27                 //分页信息
28                 $("#pageContainer").jqPaginator({
29                     totalCounts: result.total, //总记录数
30                     pageSize: 10, //每页的个数
31                     visiblePages: 5, //可视页数
32                     currentPage: result.pageRequest.currentPage, //当前页码
33                     first: '<li class="page-item"><a class="page-link">首页</a>
</li>',
34                     prev: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">上一页</a></li>',
35                     next: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">下一页</a></li>',
36                     last: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">最后一页</a></li>',
37                     page: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">{{page}}</a></li>',
38                     //页面初始化和页码点击时都会执行
39                     onPageChange: function (page, type) {

```

```
40         console.log("第" + page + "页, 类型:" + type);
41
42     }
43     });
44
45 }
46 }
47 });
48 }
```

完善页面点击代码:

当点击页码时: 跳转到页面: `book_list.html?currentPage=?`

修改上述代码代码:

```
1 onPageChange: function (page, type) {
2     if (type !== 'init') {
3         location.href = "book_list.html?currentPage=" + page;
4     }
5 }
```

测试

访问url: http://127.0.0.1:8080/book_list.html

图书列表展示

添加图书

批量删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	25	图书21	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	24	图书20	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	23	图书19	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	22	图书18	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	21	图书17	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	19	图书15	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	18	图书14	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	17	图书13	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	16	图书12	作者2	29	22	出版社1	可借阅	修改 删除

首页

上一页

1

2

3

下一页

最后一页

点击页码, 页面信息得到正确的处理

图书列表展示

添加图书

批量删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	5	图书1	作者1	23	65	出版社1	可借阅	修改 删除
<input type="checkbox"/>	4	金字塔原理	麦肯锡	16	178	民主与建设出版社	可借阅	修改 删除
<input type="checkbox"/>	3	三体	刘慈欣	9	102.67	重庆出版社	可借阅	修改 删除
<input type="checkbox"/>	2	平凡的世界	路遥	5	98.56	北京十月文艺出版社	可借阅	修改 删除
<input type="checkbox"/>	1	活着	余华	29	22	北京文艺出版社	可借阅	修改 删除

首页

上一页

1

2

3

下一页

最后一页

2.2.8 修改图书

约定前后端交互接口

进入修改页面, 需要显示当前图书的信息

```
1 [请求]
2 /book/queryBookById?bookId=25
3
4 [参数]
5 无
6
7 [响应]
8 {
9     "id": 25,
10    "bookName": "图书21",
11    "author": "作者2",
12    "count": 999,
13    "price": 222.00,
14    "publish": "出版社1",
15    "status": 2,
16    "statusCN": null,
17    "createTime": "2023-09-04T04:01:27.000+00:00",
18    "updateTime": "2023-09-05T03:37:03.000+00:00"
19 }
```

根据图书ID, 获取当前图书的信息

点击修改按钮, 修改图书信息

```
1 [请求]
2 /book/updateBook
3 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
4
5 [参数]
6 id=1&bookName=图书1&author=作者1&count=23&price=36&publish=出版社1&status=1
7
8 [响应]
9 "" //失败信息, 成功时返回空字符串
```

我们约定, 浏览器给服务器发送一个 `/book/updateBook` 这样的 HTTP 请求, form 表单的形式来提交数据

服务器返回处理结果, 返回 "" 表示添加图书成功, 否则, 返回失败信息.

实现服务器代码

BookController:

```
1 @RequestMapping("/queryBookById")
2 public BookInfo queryBookById(Integer bookId){
3     if (bookId==null || bookId<=0){
4         return new BookInfo();
5     }
6     BookInfo bookInfo = bookService.queryBookById(bookId);
7     return bookInfo;
8 }
9
10 @RequestMapping("/updateBook")
11 public String updateBook(BookInfo bookInfo) {
12     log.info("修改图书:{}", bookInfo);
13     try {
14         bookService.updateBook(bookInfo);
15         return "";
16     } catch (Exception e) {
17         log.error("修改图书失败", e);
18         return e.getMessage();
19     }
20 }
```

业务层:

BookService:

```
1 public BookInfo queryBookById(Integer bookId) {
2     return bookInfoMapper.queryBookById(bookId);
3 }
4
5 public void updateBook(BookInfo bookInfo) {
6     bookInfoMapper.updateBook(bookInfo);
7 }
```

数据层:

根据图书ID,查询图书信息

```
1 @Select("select id, book_name, author, count, price, publish, `status`,
2         create_time, update_time " +
3         "from book_info where id=#{bookId} and status<>0")
3 BookInfo queryBookById(Integer bookId);
```

更新逻辑相对较为复杂, 传递了哪些值, 咱们更新哪些值, 需要使用动态SQL
对于初学者而言, 注解的方式拼接动态SQL不太友好, 咱们采用xml的方式来实现

配置xml路径

```
1 mybatis:
2   mapper-locations: classpath:mapper/**/*.xml
```

最终整体的yml配置文件为:

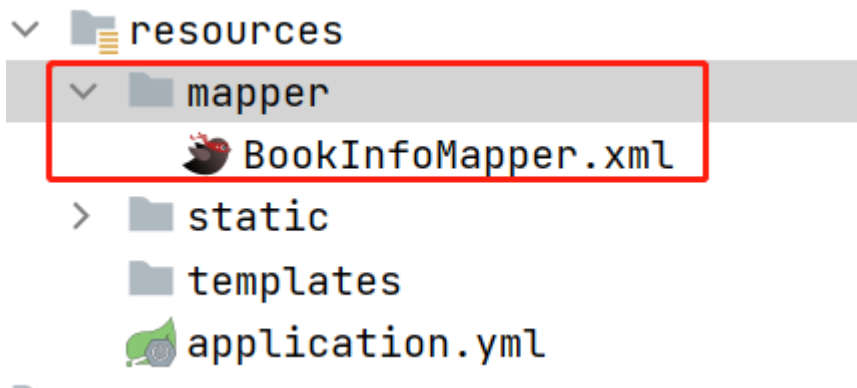
```
1 # 数据库连接配置
2 spring:
3   datasource:
4     url: jdbc:mysql://127.0.0.1:3306/mybatis_test?
       characterEncoding=utf8&useSSL=false
5     username: root
6     password: root
7     driver-class-name: com.mysql.cj.jdbc.Driver
8 mybatis:
9   configuration:
10    map-underscore-to-camel-case: true #配置驼峰自动转换
11    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl #打印sql语句
12    mapper-locations: classpath:mapper/**/*.xml
```

定义Mapper接口: BookInfoMapper

```
1 Integer updateBook(BookInfo bookInfo);
```

xml实现:

创建BookInfoMapper.xml文件



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3 <mapper namespace="com.example.demo.mapper.BookInfoMapper">
4     <update id="updateBook">
5         update book_info
6         <set>
7             <if test='bookName!=null'>
8                 book_name = #{bookName},
9             </if>
10            <if test='author!=null'>
11                author = #{author},
12            </if>
13            <if test='price!=null'>
14                price = #{price},
15            </if>
16            <if test='count!=null'>
17                count = #{count},
18            </if>
19            <if test='publish!=null'>
20                publish = #{publish},
21            </if>
22            <if test='status!=null'>
23                status = #{status},
24            </if>
25        </set>
26        where id = #{id}
27    </update>
28 </mapper>
```

实现客户端代码

我们观察, 在列表页时, 我们已经补充了[修改] 的链接

http://127.0.0.1:8080/book_update.html?bookId=25 (25为对应的图书ID)

```
1 finalHtml += '<a href="book_update.html?bookId=' + book.id + '">修改</a>';
```

点击[修改] 链接时, 就会自动跳转到 http://127.0.0.1:8080/book_update.html?bookId=25 (25为对应的图书ID)

进入[修改图书]页面时, 需先从后端拿到当前图书的信息, 显示在页面上

```
1 $.ajax({
2     type: "get",
3     url: "/book/queryBookById"+location.search,
4     success: function (book) {
5         if (book != null) {
6             $("#bookName").val(book.bookName);
7             $("#bookAuthor").val(book.author);
8             $("#bookStock").val(book.count);
9             $("#bookPrice").val(book.price);
10            $("#bookPublisher").val(book.publish);
11            $("#bookStatus").val(book.status);
12        }
13    }
14 });
15 });
```

补全修改图书的方法:

```
1 function update() {
2     $.ajax({
3         type: "post",
4         url: "/book/updateBook",
5         data: $("#updateBook").serialize(),
6         success: function (result) {
7             if (result == "") {
8                 location.href = "book_list.html"
9             } else {
10                console.log(result);
11                alert("修改失败:" + result);
12            }
13        },
14        error: function (error) {
15            console.log(error);
16        }
17    });
18 }
```

```
16     }  
17     });  
18 }
```

我们修改图书信息, 是根据图书ID来修改的, 所以需要前端传递的参数中, 包含图书ID.

有两种方式:

1. 获取url中参数的值(比较复杂, 需要拆分url)
2. 在form表单中, 再增加一个隐藏输入框, 存储图书ID, 随 `$("#updateBook").serialize()` 一起提交到后端

我们采用第二种方式

在form表单中, 添加隐藏输入框

```
1 <form id="updateBook">  
2   <input type="hidden" class="form-control" id="bookId" name="id">  
3   <div class="form-group">  
4     <label for="bookName">图书名称:</label>  
5     <input type="text" class="form-control" id="bookName" name="bookName">  
6   </div>  
7   <!-- ..... -->
```

hidden 类型的 `<input>` 元素

隐藏表单, 用户不可见、不可改的数据, 在用户提交表单时, 这些数据会一并发送出

使用场景: 正被请求或编辑的内容的 ID. 这些隐藏的 input 元素在渲染完成的页面中完全不可见, 而且没有方法可以使它重新变为可见.

页面加载时, 给该hidden框赋值

```
1 $("#bookId").val(book.id);
```

此时前端js完整代码:

```
1 $.ajax({  
2   type: "get",  
3   url: "/book/queryBookById"+location.search,  
4   success: function(book) {  
5     if (book != null) {
```

```

6         $("#bookId").val(book.id);
7         $("#bookName").val(book.bookName);
8         $("#bookAuthor").val(book.author);
9         $("#bookStock").val(book.count);
10        $("#bookPrice").val(book.price);
11        $("#bookPublisher").val(book.publish);
12        $("#bookStatus").val(book.status);
13
14    }
15 }
16 });
17 function update() {
18     $.ajax({
19         type: "post",
20         url: "/book/updateBook",
21         data: $("#updateBook").serialize(),
22         success: function (result) {
23             if (result=="") {
24                 location.href = "book_list.html"
25             } else {
26                 console.log(result);
27                 alert("修改失败:"+result);
28             }
29         },
30         error: function (error) {
31             console.log(error);
32         }
33     });
34 }

```

程序运行, 测试

点击[修改]链接

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	25	图书21	作者2	29	22	出版社1	可借阅	修改 删除

跳转到图书修改页面, 页面加载出该图书的信息

修改图书

图书名称:

图书21

图书作者

作者2

图书库存

29

图书定价:

22

出版社

出版社1

图书状态

可借阅

确定

返回

随机修改数据, 点击确定按钮, 观察数据是否被修改

<input type="checkbox"/>	25	图书21	作者2	29	22	出版社1	可借阅	修改 删除
数据修改前后								
<input type="checkbox"/>	25	图书21	作者2	999	222	出版社1	不可借阅	修改 删除

2.2.9 删除图书

约定前后端交互接口

删除分为 逻辑删除 和 物理删除

逻辑删除

逻辑删除也称为软删除、假删除、Soft Delete，即不真正删除数据，而在某行数据上增加类型 is_deleted 的删除标识，一般使用 UPDATE 语句

物理删除

物理删除也称为硬删除，从数据库表中删除某一行或某一集合数据，一般使用DELETE语句

删除图书的两种实现方式

逻辑删除

```
1 update book_info set status=0 where id = 1
```

物理删除

```
1 delete from book_info where id=25
```

数据是公司的重要财产, 通常情况下, 我们采用逻辑删除的方式, 当然也可以采用[物理删除+归档]的方式

物理删除并归档

1. 创建一个与原表差不多结构, 记录删除时间, 实现INSERT .. SELECT即可 [SQL INSERT INTO SELECT 语句 | 菜鸟教程](#)
2. 插入和删除操作, 放在同一个事务中执行

物理删除+归档的方式实现有些复杂, 咱们采用逻辑删除的方式

逻辑删除的话, 依然是更新逻辑, 我们可以直接使用修改图书的接口

/book/updateBook

```
1 [请求]
2 /book/updateBook
3 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
4
5 [参数]
6 id=1&status=0
7
8 [响应]
9 "" //失败信息, 成功时返回空字符串
```

实现客户端代码

点击删除时, 调用delete()方法, 我们来完善delete方法

前端代码已经提供了

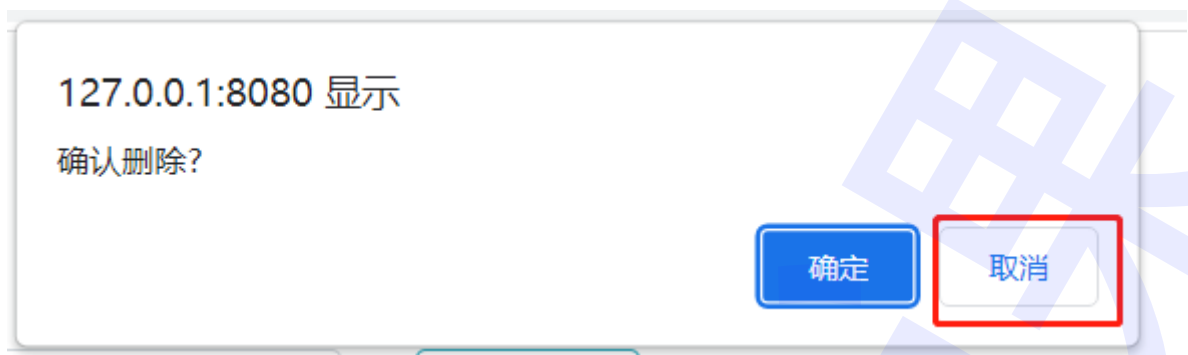
```
finalHtml += '<a href="javascript:void(0)" onclick="deleteBook(' + book.id + ')">删除</a>';
```

```
1 function deleteBook(id) {
2     var isDelete = confirm("确认删除?");
3     if (isDelete) {
4         //删除图书
5         $.ajax({
6             type: "post",
7             url: "/book/updateBook",
8             data: {
9                 id: id,
10                status: 0
11            },
12            success: function () {
13                //重新刷新页面
14                location.href = "book_list.html"
15            }
16        });
17    }
18 }
```

测试

点击[删除]

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	24	图书20	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	23	图书19	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	22	图书18	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	21	图书17	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除



点击[取消], 观察数据依然存在

点击[确定], 数据删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	23	图书19	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	22	图书18	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	21	图书17	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除

2.2.10 批量删除

批量删除, 其实就是批量修改数据

约定前后端交互接口

```
1 [请求]
2 /book/batchDeleteBook
3 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
4
5 [参数]
6
7
8 [响应]
9 "" //失败信息, 成功时返回空字符串
```

点击[批量删除]按钮时, 只需要把复选框选中的图书的ID,发送到后端即可

多个id, 我们使用List的形式来传递参数

实现服务器代码

控制层:

BookController

```

1 @RequestMapping("/batchDeleteBook")
2 public boolean batchDeleteBook(@RequestParam List<Integer> ids){
3     log.info("批量删除图书, ids:{",ids);
4     try{
5         bookService.batchDeleteBook(ids);
6     }catch (Exception e){
7         log.error("批量删除异常,e:",e);
8         return false;
9     }
10
11     return true;
12 }

```

业务层:

BookService

```

1 public void batchDeleteBook(List<Integer> ids) {
2     bookInfoMapper.batchDeleteBook(ids);
3 }

```

数据层:

批量删除需要用到动态SQL, 初学者建议使用动态SQL的部分, 都用xml实现

BookInfoMapper.java

接口定义

```

1 void batchDeleteBook(List<Integer> ids);

```

xml接口实现

```

1 <update id="batchDeleteBook">
2     update book_info set status=0
3     where id in
4     <foreach collection="ids" item="id" separator="," open="(" close=")">
5         #{id}
6     </foreach>
7 </update>

```

实现客户端代码

点击[批量删除]按钮时, 需要获取到所有选中的复选框的值

```
1 function batchDelete() {
2     var isDelete = confirm("确认批量删除?");
3     if (isDelete) {
4         //获取复选框的id
5         var ids = [];
6         $("input:checkbox[name='selectBook']:checked").each(function () {
7             ids.push($(this).val());
8         });
9         console.log(ids);
10        //批量删除
11        $.ajax({
12            type: "post",
13            url: "/book/batchDeleteBook?ids="+ids,
14            success: function (result) {
15                if (result) {
16                    alert("删除成功");
17                    //重新刷新页面
18                    location.href = "book_list.html"
19                }
20            }
21        });
22    }
23 }
```

测试

选择删除的图书, 点击[批量删除]

图书列表展示

书名:

作者:

搜索图书

添加图书

批量删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input checked="" type="checkbox"/>	23	图书19	作者2	29	22	出版社1	可借阅	修改 删除
<input checked="" type="checkbox"/>	22	图书18	作者2	29	22	出版社1	可借阅	修改 删除
<input checked="" type="checkbox"/>	21	图书17	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	19	图书15	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	18	图书14	作者2	29	22	出版社1	可借阅	修改 删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	19	图书15	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	18	图书14	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	17	图书13	作者2	29	22	出版社1	可借阅	修改 删除

2.2.11 强制登录

虽然我们做了用户登录,但是我们发现,用户不登录,依然可以操作图书. 这是有极大风险的. 所以我们需要进行强制登录. 如果用户未登录就访问图书列表或者添加图书等页面,强制跳转到登录页面.

实现思路分析

用户登录时,我们已经把登录用户的信息存储在了Session中. 那就可以通过Session中的信息来判断用户都是登录.

1. 如果Session中可以取到登录用户的信息,说明用户已经登录了,可以进行后续操作
2. 如果Session中取不到登录用户的信息,说明用户未登录,则跳转到登录页面.

以图书列表为例
现在图书列表接口返回的内容如下:

```

1 {
2     "total": 25,
3     "records": [{
4         "id": 25,
5         "bookName": "图书21",
6         "author": "作者2",
7         "count": 29,
8         "price": 22.00,
9         "publish": "出版社1",
10        "status": 1,
11        "statusCN": "可借阅"
12    }, {
13        .....
14    } ]
15 }

```

这个结果上, 前端没办法确认用户是否登录了. 并且后端返回数据为空时, 前端也无法确认是后端无数据, 还是后端出错了.

当前后端接口数据返回类:

```

1 @Data
2 public class PageResult<T> {
3     private int total; //所有记录数
4     private List<T> records; // 当前页数据
5     private PageRequest pageRequest;
6
7     public PageResult(Integer total, PageRequest pageRequest, List<T> records)
8     {
9         this.total = total;
10        this.pageRequest = pageRequest;
11        this.records = records;
12    }
13 }

```

我们需要再增加一个属性告知后端的状态以及后端出错的原因. 修改如下:

```

1 @Data
2 public class PageResult<T> {
3     private int status;
4     private String errorMessage;
5
6     private int total; //所有记录数

```



```

7     private List<T> records; // 当前页数据
8     private PageRequest pageRequest;
9
10    public PageResult(Integer total, PageRequest pageRequest, List<T> records)
11    {
12        this.total = total;
13        this.pageRequest = pageRequest;
14        this.records = records;
15    }

```

但是当前只是图书列表而已, 图书的增加, 修改, 删除接口都需要跟着修改, 添加两个字段. 这对我们的代码修改是巨大的.

我们不妨对所有后端返回的数据进行一个封装

```

1 @Data
2 public class Result<T> {
3     private int status;
4     private String errorMessage;
5     private T data;
6 }

```

data为之前接口返回的数据

status 为后端业务处理的状态码, 也可以使用枚举来表示

```

1 public enum ResultStatus {
2
3     SUCCESS(200),
4     UNLOGIN(-1),
5     FAIL(-2);
6
7     private Integer code;
8
9     ResultStatus(int code) {
10         this.code = code;
11     }
12
13
14     public Integer getCode() {
15         return code;
16     }
17 }

```

```
18     public void setCode(Integer code) {
19         this.code = code;
20     }
21
22 }
```

修改Result, 并添加一些常用方法

```
1  import com.example.demo.enums.ResultStatus;
2  import lombok.Data;
3
4  @Data
5  public class Result<T> {
6      private ResultStatus status;
7      private String errorMessage;
8      private T data;
9
10     /**
11      * 业务执行成功时返回的方法
12      *
13      * @param data
14      * @return
15      */
16     public static <T> Result success(T data) {
17         Result result = new Result();
18         result.setStatus(ResultStatus.SUCCESS);
19         result.setErrorMessage("");
20         result.setData(data);
21         return result;
22     }
23
24     /**
25      * 业务执行失败时返回的方法
26      *
27      * @param
28      * @return
29      */
30     public static Result fail(String msg) {
31         Result result = new Result();
32         result.setStatus(ResultStatus.FAIL);
33         result.setErrorMessage(msg);
34         result.setData("");
35         return result;
36     }
37 }
```

```

38  /**
39   * 业务执行失败时返回的方法
40   *
41   * @param
42   * @return
43   */
44  public static Result unlogin() {
45      Result result = new Result();
46      result.setStatus(ResultStatus.UNLOGIN);
47      result.setErrorMessage("用户未登录");
48      result.setData(null);
49      return result;
50  }
51  }

```

接下来修改服务器和客户端代码

实现服务器代码

修改图书列表接口, 进行登录校验

```

1  @RequestMapping("/getListByPage")
2  public Result getListByPage(PageRequest pageRequest, HttpSession session) {
3      log.info("获取图书列表, pageRequest:{})", pageRequest);
4      //判断用户是否登录
5      if (session.getAttribute("session_user_key")==null){
6          return Result.unlogin();
7      }
8      UserInfo userInfo = (UserInfo) session.getAttribute("session_user_key");
9      if (userInfo==null || userInfo.getId()<0 ||
10         "".equals(userInfo.getUserName())){
11          return Result.unlogin();
12      }
13      //用户登录, 返回图书列表
14      PageResult<BookInfo> pageResult =
15         bookService.getBookListByPage(pageRequest);
16      return Result.success(pageResult);
17  }

```

问题: 如果修改常量session的key, 就需要修改所有使用到这个key的地方, 出于高内聚低耦合的思想, 我们把常量集中在一个类里

创建类: Constants

```
1 public class Constants {
2     public static final String SESSION_USER_KEY = "session_user_key";
3 }
```

常量命名规则:

常量命名全部大写, 单词间用下划线隔开, 力求语义表达完整清楚, 不要在意名字长度.

正例: MAX_STOCK_COUNT / CACHE_EXPIRED_TIME

反例: COUNT / TIME

修改之前使用到 session_user_key

登录接口

```
1 @RequestMapping("/login")
2 public boolean login(String name, String password, HttpSession session){
3     //账号或密码为空
4     if (!StringUtils.hasLength(name) || !StringUtils.hasLength(password)){
5         return false;
6     }
7     UserInfo userInfo = userService.queryUserByName(name);
8     if (userInfo == null){
9         return false;
10    }
11    //账号密码正确
12    if(userInfo!=null && password.equals(userInfo.getPassword())){
13        //存储在Session中
14        userInfo.setPassword("");
15        session.setAttribute(Constants.SESSION_USER_KEY,userInfo);
16        return true;
17    }
18    //账号密码错误
19    return false;
20 }
```

图书列表接口:

```
1 @RequestMapping("/getListByPage")
2 public Result getListByPage(PageRequest pageRequest, HttpSession session) {
3     log.info("获取图书列表, pageRequest:{}, pageRequest);
4     //判断用户是否登录
5     if (session.getAttribute(Constants.SESSION_USER_KEY)==null){
6         return Result.unlogin();
7     }
8 }
```

```

8      UserInfo userInfo = (UserInfo)
      session.getAttribute(Constants.SESSION_USER_KEY);
9      if (userInfo==null || userInfo.getId()<0 ||
      "".equals(userInfo.getUserName())){
10          return Result.unlogin();
11      }
12      //用户登录，返回图书列表
13      PageResult<BookInfo> pageResult =
      bookService.getBookListByPage(pageRequest);
14      return Result.success(pageResult);
15  }

```

实现客户端代码

由于后端接口发生变化, 所以前端接口也需要进行调整

这也就是为什么前后端交互接口一旦定义好, 尽量不要发生变化.

所以后端接口返回的数据类型一般不定义为基本类型, 包装类型或者集合类等, 而是定义为自定义对象. 方便后续做扩展

修改图书列表的方法(下面代码为修改部分):

```

1  function getBookList() {
2      //...
3      success: function (result) {
4
5          //真实前端代码需要分的更细一点，此处不做判断
6          if (result == null || result.data == null) {
7              location.href = "login.html";
8              return;
9          }
10
11          //....
12          var data = result.data;
13          $("#pageContainer").jqPaginator({
14              totalCounts: data.total, //总记录数
15              pageSize: 10, //每页的个数
16              visiblePages: 5, //可视页数
17              currentPage: data.pageRequest.currentPage, //当前页码
18              //....
19          });
20      }
21  }

```

完整代码如下:

```

1  getBookList();
2  function getBookList() {
3      $.ajax({
4          type: "get",
5          url: "/book/getListByPage" + location.search,
6          success: function (result) {
7              console.log(result);
8              //真实前端代码需要分的更细一点, 此处不做判断
9              if (result == null || result.data == null) {
10                  location.href = "login.html";
11                  return;
12              }
13
14              var finalHtml = "";
15              var data = result.data;
16              for (var book of data.records) {
17                  finalHtml += '<tr>';
18                  finalHtml += '<td><input type="checkbox" name="selectBook"
value="" + book.id + ' id="selectBook" class="book-select"></td>';
19                  finalHtml += '<td>' + book.id + '</td>';
20                  finalHtml += '<td>' + book.bookName + '</td>';
21                  finalHtml += '<td>' + book.author + '</td>';
22                  finalHtml += '<td>' + book.count + '</td>';
23                  finalHtml += '<td>' + book.price + '</td>';
24                  finalHtml += '<td>' + book.publish + '</td>';
25                  finalHtml += '<td>' + book.statusCN + '</td>';
26                  finalHtml += '<td><div class="op">';
27                  finalHtml += '<a href="book_update.html?bookId=' + book.id +
'">修改</a>';
28                  finalHtml += '<a href="javascript:void(0)"
onclick="deleteBook(' + book.id + ')">删除</a>';
29                  finalHtml += '</div></td>';
30                  finalHtml += "</tr>";
31              }
32
33              $("tbody").html(finalHtml);
34
35              $("#pageContainer").jqPaginator({
36                  totalCounts: data.total, //总记录数
37                  pageSize: 10, //每页的个数
38                  visiblePages: 5, //可视页数
39                  currentPage: data.pageRequest.currentPage, //当前页码
40                  first: '<li class="page-item"><a class="page-link">首页</a>
</li>',
41                  prev: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">上一页</a></li>',

```

```

42         next: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">下一页</a></li>',
43         last: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">最后一页</a></li>',
44         page: '<li class="page-item"><a class="page-link"
href="javascript:void(0);">{{page}}</a></li>',
45         //页面初始化和页码点击时都会执行
46         onPageChange: function (page, type) {
47             if (type !== 'init') {
48                 location.href = "book_list.html?currentPage=" + page;
49             }
50         }
51     });
52
53     }
54 });
55 }

```

测试

1. 用户未登录情况,访问图书列表: http://127.0.0.1:8080/book_list.html

发现跳转到了登录页面



2. 登录用户, 图书列表正常返回

图书列表展示

添加图书

批量删除

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
<input type="checkbox"/>	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	19	图书15	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	18	图书14	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	17	图书13	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	16	图书12	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	15	图书11	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	14	图书10	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	13	图书9	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	12	图书8	作者2	29	22	出版社1	可借阅	修改 删除
<input type="checkbox"/>	11	图书7	作者2	29	22	出版社1	可借阅	修改 删除

首页

上一页

1

2

下一页

最后一页

2.2.12 思考

强制登录的模块, 我们只实现了一个图书列表, 上述还有图书修改, 图书删除等接口, 也需要一一实现. 如果应用程序功能更多的话, 这样写下来会非常浪费时间, 并且容易出错. 有没有更简单的处理办法呢? 接下来我们学习SpringBoot对于这种"统一问题"的处理办法.

3. MyBatis Generator

MyBatis Generator是一个为MyBatis框架设计的代码生成工具, 它可以根据数据库表结构自动生成相应的Java Model, Mapper接口以及SQL映射文件, 简化数据访问层的编码工作, 使得开发者可以更专注于业务逻辑的实现. 接下来我们看下, 如何使用MyBatis Generator 来生成代码.

3.1 引入插件

```
1 <plugin>
```



```

2      <groupId>org.mybatis.generator</groupId>
3      <artifactId>mybatis-generator-maven-plugin</artifactId>
4      <version>1.3.6</version>
5      <executions>
6          <execution>
7              <id>Generate MyBatis Artifacts</id>
8              <phase>deploy</phase>
9              <goals>
10                 <goal>generate</goal>
11             </goals>
12         </execution>
13     </executions>
14     <configuration>
15         <!--generator配置文件所在位置-->
16
17         <configurationFile>src/main/resources/mybatisGenerator/generatorConfig.xml</con
18         figurationFile>
19         <!-- 允许覆盖生成的文件，mapxml不会覆盖，采用追加的方式-->
20         <overwrite>true</overwrite>
21         <verbose>true</verbose>
22         <!--将当前pom的依赖项添加到生成器的类路径中-->
23         <includeCompileDependencies>true</includeCompileDependencies>
24     </configuration>
25     <dependencies>
26         <dependency>
27             <groupId>mysql</groupId>
28             <artifactId>mysql-connector-java</artifactId>
29             <version>8.0.33</version>
30         </dependency>
31     </dependencies>
32 </plugin>

```

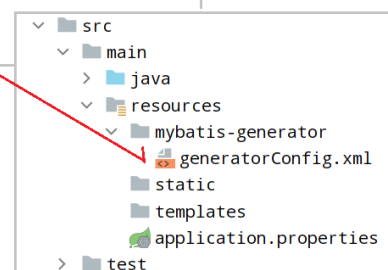
3.2 添加generatorConfig.xml并修改

文件路径和上述配置保持一致:

```

<configuration>
    <!--generator配置文件所在位置-->
    <configurationFile>src/main/resources/mybatis-generator/generatorConfig.xml</configurationFile>
    <overwrite>true</overwrite>
    <verbose>true</verbose>
</configuration>

```



完善文件内容

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE generatorConfiguration
3     PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
4     "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
5 <!-- 配置生成器 -->
6 <generatorConfiguration>
7     <!-- 一个数据库一个context -->
8     <context id="MysqlTables" targetRuntime="MyBatis3Simple">
9         <!--禁用自动生成的注释-->
10        <commentGenerator>
11            <property name="suppressDate" value="true"/>
12            <property name="suppressAllComments" value="true" />
13        </commentGenerator>
14        <!--数据库连接信息-->
15        <jdbcConnection driverClass="com.mysql.jdbc.Driver"
16
17            connectionURL="jdbc:mysql://127.0.0.1:3306/java_blog_spring?
18            serverTimezone=Asia/Shanghai&nullCatalogMeansCurrent=true"
19            userId="root"
20            password="root">
21        </jdbcConnection>
22        <!-- 生成实体类, 配置路径 -->
23        <javaModelGenerator targetPackage="com.example.demo.model"
24            targetProject="src/main/java" >
25            <property name="enableSubPackages" value="false"/>
26            <property name="trimStrings" value="true"/>
27        </javaModelGenerator>
28        <!-- 生成mapxml文件 -->
29        <sqlMapGenerator targetPackage="mapper"
30            targetProject="src/main/resources" >
31            <property name="enableSubPackages" value="false" />
32        </sqlMapGenerator>
33        <!-- 生成mapxml对应client, 也就是接口dao -->
34        <javaClientGenerator targetPackage="com.example.demo.mapper"
35            targetProject="src/main/java" type="XMLMAPPER" >
36            <property name="enableSubPackages" value="false" />
37        </javaClientGenerator>
38        <!-- table可以有多个, tableName表示要匹配的数据库表 -->
39        <table tableName="user" domainObjectName="UserInfo"
40            enableSelectByExample="true"
41            enableDeleteByExample="true" enableDeleteByPrimaryKey="true"
42            enableCountByExample="true"
43            enableUpdateByExample="true">
```

```

37      <!-- 类的属性是否用数据库中的真实字段名做为属性名，不指定这个属性会自动转
      换 _ 为驼峰命名规则 -->
38      <property name="useActualColumnNames" value="false" />
39      <!-- 数据库表主键 -->
40      <generatedKey column="id" sqlStatement="Mysql" identity="true" />
41  </table>
42
43  </context>
44 </generatorConfiguration>

```

3.3 生成文件

双击运行就可以了

- > Lifecycle
- ▼ Plugins
 - > clean (org.apache.maven.plugins:maven-clean-plugin:3.3)
 - > compiler (org.apache.maven.plugins:maven-compiler-plugin:3.3)
 - > deploy (org.apache.maven.plugins:maven-deploy-plugin:3.1.1)
 - > install (org.apache.maven.plugins:maven-install-plugin:3.0.1)
 - > jar (org.apache.maven.plugins:maven-jar-plugin:3.4.2)
 - ▼ mybatis-generator (org.mybatis.generator:mybatis-generator:1.3.5)
 - ⚙ mybatis-generator:generate
 - ⚙ mybatis-generator:help
 - > resources (org.apache.maven.plugins:maven-resources-plugin:3.1.0)
 - > site (org.apache.maven.plugins:maven-site-plugin:3.3)
 - > spring-boot (org.springframework.boot:spring-boot-maven-plugin:2.1.6)
 - > surefire (org.apache.maven.plugins:maven-surefire-plugin:2.22.1)

4. 总结

- 学习了MyBatis动态SQL的一些标签使用. `<if>`标签中, 使用的是Java对象的属性, 而非数据库字段.
- 动态SQL的实现, 注解和xml的实现方式相似, 区别是注解方式需要添加 `<script></script>`. 但是使用注解的方式时, Idea不会进行格式检测, 容易出错, 建议初学者用xml的方式
- MyBatis的学习, 需要更多的练习才能掌握. 代码一定要自己写, 切忌眼高手低.