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：

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:

```
<insert id="insertUserByCondition">
    INSERT INTO userinfo (
    username,
    `password`,
    age,
    <if test="gender != null">
        gender,
    </if>
    phone)
    VALUES (
    #{username},
    #{age},
    <if test="gender != null">
        #{gender},
    </if>
    #{phone})
</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

```
@Insert("<script>" +
        "INSERT INTO userinfo (username,`password`,age," +
        "<if test='gender!=null'>gender,</if>" +
        "phone)" +
        "VALUES(#{username},#{age}," +
        "<if test='gender!=null'>#{gender},</if>" +
        "#{phone})"+
        "</script>")
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：

```
<insert id="insertUserByCondition">
    INSERT INTO userinfo
    <trim prefix="(" suffix=")" suffixOverrides=",">
        <if test="username !=null">
            username,
        </if>
        <if test="password !=null">
            `password`,
        </if>
        <if test="age != null">
            age,
        </if>
        <if test="gender != null">
            gender,
        </if>
        <if test="phone != null">
            phone,
        </if>
    </trim>
    VALUES
    <trim prefix="(" suffix=")" suffixOverrides=",">
        <if test="username !=null">
            #{username},
        </if>
        <if test="password !=null">
            #{password},
        </if>
        <if test="age != null">
            #{age},
        </if>
        <if test="gender != null">
            #{gender},
        </if>
        <if test="phone != null">
```

```
            #{phone}
        </if>
    </trim>
</insert>
```

或者使用注解方式(不推荐)

```
@Insert("<script>" +
        "INSERT INTO userinfo " +
        "<trim prefix='(' suffix=')' suffixOverrides=','>" +
        "<if test='username!=null'>username,</if>" +
        "<if test='password!=null'>password,</if>" +
        "<if test='age!=null'>age,</if>" +
        "<if test='gender!=null'>gender,</if>" +
        "<if test='phone!=null'>phone,</if>" +
        "</trim>" +
        "VALUES " +
        "<trim prefix='(' suffix=')' suffixOverrides=','>" +
        "<if test='username!=null'>#{username},</if>" +
        "<if test='password!=null'>#{password},</if>" +
        "<if test='age!=null'>#{age},</if>" +
        "<if test='gender!=null'>#{gender},</if>" +
        "<if test='phone!=null'>#{phone}</if>" +
        "</trim>"+
        "</script>")
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

```
SELECT
        *
FROM
        userinfo
WHERE
        age = 18
        AND gender = 1
        AND delete_flag =0
```

接口定义:

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

Mapper.xml实现

```
<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>
```

> `<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

```
@Select("<script>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>" +
        "</script>")
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

```
Integer updateUserByCondition(UserInfo userInfo);
```

Mapper.xml

```
<update id="updateUserByCondition">
    update userinfo
    <set>
        <if test="username != null">
            username = #{username},
        </if>
```

```
        <if test="age != null">
            age = #{age},
        </if>
        <if test="deleteFlag != null">
            delete_flag = #{deleteFlag},
        </if>
    </set>
    where id = #{id}
</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

```
@Update("<script>" +
        "update userinfo " +
        "<set>" +
        "<if test='username!=null'>username=#{username},</if>" +
        "<if test='age!=null'>age=#{age},</if>" +
        "<if test='deleteFlag!=null'>delete_flag=#{deleteFlag},</if>" +
        "</set>" +
        "where id=#{id}" +
        "</script>")
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:

```
void deleteByIds(List<Integer> ids);
```

ArticleMapper.xml 中新增删除 sql：

```xml
<delete id="deleteByIds">
    delete from userinfo
    where id in
    <foreach collection="ids" item="id" separator="," open="(" close=")">
        #{id}
    </foreach>
</delete>
```

或者使用注解方式:

```
@Delete("<script>" +
            "delete from userinfo where id in" +
            "<foreach collection='ids' item='id' separator=',' open='('
close=')'>" +
            "#{id}" +
            "</foreach>" +
        "</script>")
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>
```

```xml
<select id="queryById" resultType="com.example.demo.model.UserInfo">
    select
    <include refid="allColumn"></include>
    from userinfo where id= #{id}
</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yml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, 需要把数据存储在数据库中. 接下来咱们借助MyBatis来实现数据的操作

2.1.1 数据准备

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

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

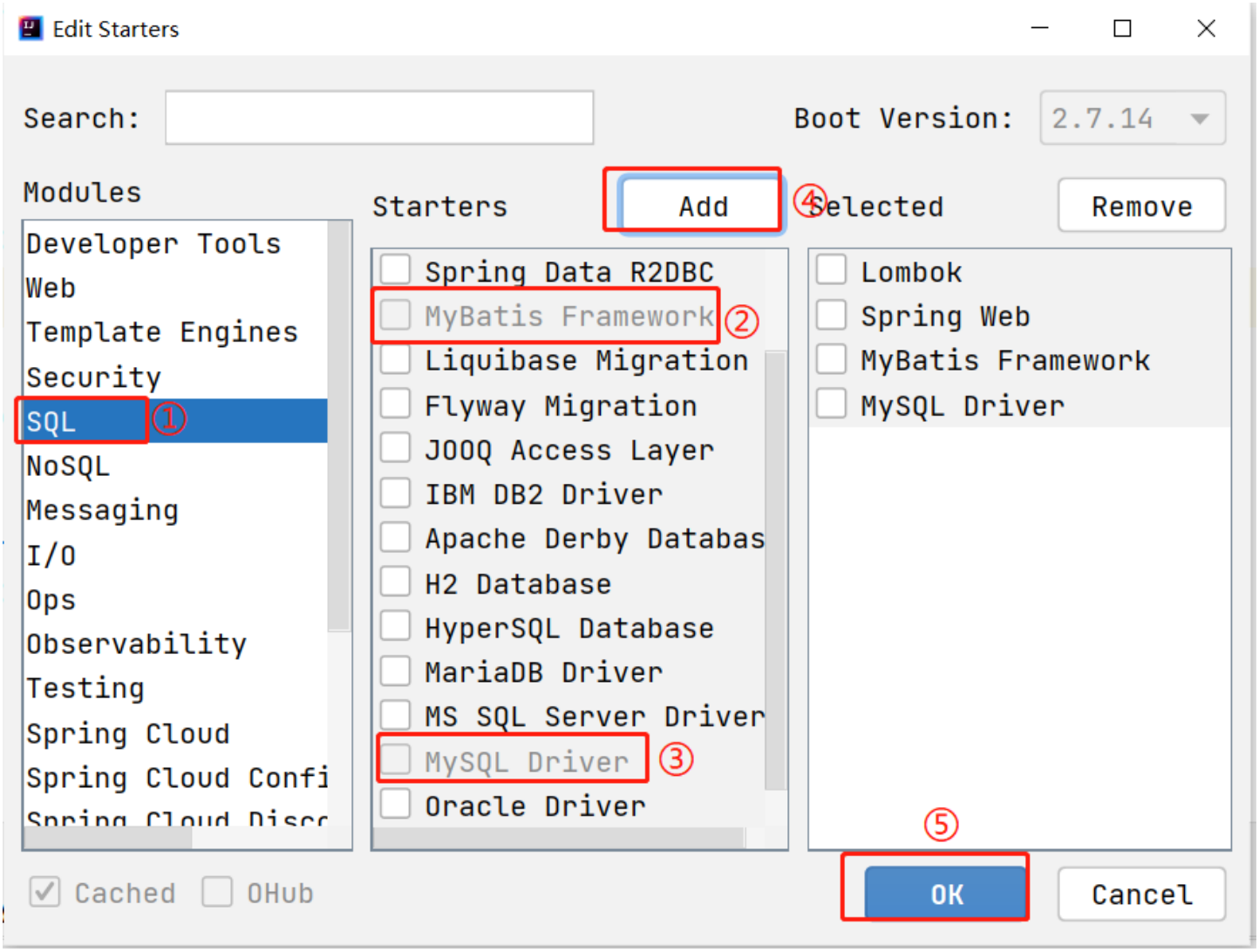
> ON UPDATE now(): 当数据发生更新操作时, 自动把该列的值设置为now(),
>
> now() 可以替换成其他获取时间的标识符, 比如: CURRENT_TIMESTAMP(), LOCALTIME()等
>
> MySQL <5.6.5
>
> 1. 只有TIMESTAMP支持自动更新
> 2. 一个表只能有一列设置自动更新
> 3. 不允许同时存在两个列, 其中一列设置了DEFAULT CURRENT_TIMESTAMP, 另一个设置了ON UPDATE CURRENT_TIMESTAMP
>
> MySQL >=5.6.5
>
> 1. TIMESTAMP 和DATETIME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

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

或者使用插件EditStarters来引入依赖



2.1.3 配置MySQL账号密码

```yaml
spring:
  datasource:
    url: jdbc:mysql://127.0.0.1:3306/mybatis_test?characterEncoding=utf8&useSSL=false
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
mybatis:
  configuration: # 配置打印 MyBatis日志
    map-underscore-to-camel-case: true #配置驼峰自动转换
```

2.1.4 编写后端代码

Model

```
import lombok.Data;

```

```
@Data
public class MessageInfo {
    private Integer id;
    private String from;
    private String to;
    private String message;
    private Integer deleteFlag;
    private Date createTime;
    private Date updateTime;
}
```

MessageInfoMapper

```
import com.example.demo.model.MessageInfo;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

import java.util.List;

@Mapper
public interface MessageInfoMapper {
    @Select("select `id`, `from`, `to`, `message` from message_info where delete_flag=0")
    List<MessageInfo> queryAll();

    @Insert("insert into message_info (`from`,`to`, `message`) values(#{from},#{to},#{message})")
    Integer addMessage(MessageInfo messageInfo);
}
```

MessageInfoService

```
import com.example.demo.mapper.MessageInfoMapper;
import com.example.demo.model.MessageInfo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class MessageInfoService {
    @Autowired
```

```
    private MessageInfoMapper messageInfoMapper;

    public List<MessageInfo> queryAll() {
        return messageInfoMapper.queryAll();
    }

    public Integer addMessage(MessageInfo messageInfo) {
        return messageInfoMapper.addMessage(messageInfo);
    }
}
```

MessageController

```
import com.example.demo.model.MessageInfo;
import com.example.demo.service.MessageInfoService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RequestMapping("/message")
@RestController
public class MessageController {
    @Autowired
    private MessageInfoService messageInfoService;
    /**
     * 获取留言列表
     * @return
     */
    @RequestMapping("/getList")
    public List<MessageInfo> getList() {
        return messageInfoService.queryAll();
    }

    /**
     * 发表留言
     * @param messageInfo
     * @return
     */
    @RequestMapping("/publish")
    public boolean publish(MessageInfo messageInfo) {
        System.out.println(messageInfo);
        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

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.

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.

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(),
```

```
        PRIMARY KEY ( `id` ),
UNIQUE INDEX `user_name_UNIQUE` ( `user_name` ASC )) ENGINE = INNODB DEFAULT CHARACTER
SET = utf8mb4 COMMENT = '用户表';

-- 图书表
DROP TABLE IF EXISTS book_info;
CREATE TABLE `book_info` (
        `id` INT ( 11 ) NOT NULL AUTO_INCREMENT,
        `book_name` VARCHAR ( 127 ) NOT NULL,
        `author` VARCHAR ( 127 ) NOT NULL,
        `count` INT ( 11 ) NOT NULL,
        `price` DECIMAL (7,2 ) NOT NULL,
        `publish` VARCHAR ( 256 ) NOT NULL,
        `status` TINYINT ( 4 ) DEFAULT 1 COMMENT '0-无效, 1-正常, 2-不允许借阅',
        `create_time` DATETIME DEFAULT now(),
        `update_time` DATETIME DEFAULT now() ON UPDATE now(),
PRIMARY KEY ( `id` )
) ENGINE = INNODB DEFAULT CHARSET = utf8mb4;

-- 初始化数据
INSERT INTO user_info ( user_name, PASSWORD ) VALUES ( "admin", "admin" );
INSERT INTO user_info ( user_name, PASSWORD ) VALUES ( "zhangsan", "123456" );

-- 初始化图书数据
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');
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');
INSERT INTO `book_info` (book_name,author,count, price, publish) VALUES ('三体', '刘慈欣', 9, 102.67, '重庆出版社');
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

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>com.mysql</groupId>
```

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

2.2.3 配置数据库&日志

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

```

2.2.4 Model创建

```
import lombok.Data;

import java.util.Date;

@Data
public class UserInfo {
    private Integer id;
    private String userName;
    private String password;
    private Integer deleteFlag;
    private Date createTime;
    private Date updateTime;
}
```

```
@Data
```

```java
public class BookInfo {
    //图书ID
    private Integer id;
    //书名
    private String bookName;
    //作者
    private String author;
    //数量
    private Integer count;
    //定价
    private BigDecimal price;
    //出版社
    private String publish;
    //状态 0-无效 1-允许借阅   2-不允许借阅
    private Integer status;

    private String statusCN;
    //创建时间
    private Date createTime;
    //更新时间
    private Date updateTime;
}
```

2.2.5 用户登录

约定前后端交互接口

```
[请求]
/user/login
Content-Type: application/x-www-form-urlencoded; charset=UTF-8


[参数]
name=zhangsan&password=123456

[响应]
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

```java
import com.example.demo.model.UserInfo;
import com.example.demo.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpSession;

@RequestMapping("/user")
@RestController
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/login")
    public boolean login(String name, String password, HttpSession session){
        //账号或密码为空
        if (!StringUtils.hasLength(name) || !StringUtils.hasLength(password)){
            return false;
        }
        UserInfo userInfo = userService.queryUserByName(name);
        if (userInfo == null){
            return false;
        }
        //账号密码正确
        if(userInfo!=null && password.equals(userInfo.getPassword())){
            //存储在Session中
            userInfo.setPassword("");
            session.setAttribute("session_user_key",userInfo);
            return true;
        }
        //账号密码错误
        return false;
    }
}
```

业务层:

创建UserService

```java
import com.example.demo.mapper.UserInfoMapper;
import com.example.demo.model.UserInfo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private UserInfoMapper userInfoMapper;

    public UserInfo queryUserByName(String name) {
        return userInfoMapper.queryUserByName(name);
    }
}
```

数据层:

创建UserInfoMapper

```java
import com.example.demo.model.UserInfo;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

@Mapper
public interface UserInfoMapper {
    @Select("select id, user_name,`password`, delete_flag, create_time, update_time " +
            "from user_info where delete_flag=0 and user_name=#{name}")
    UserInfo queryUserByName(String name);
}
```

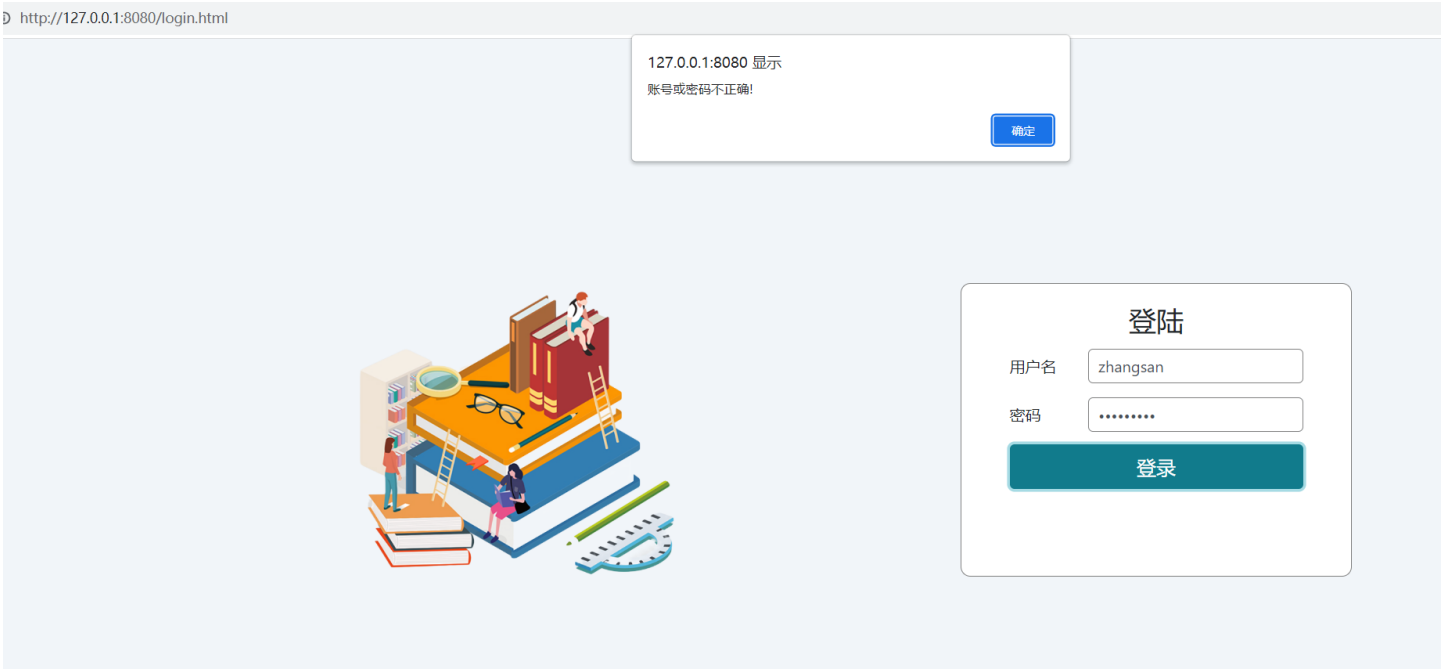
> 访问数据库, 使用MyBatis来实现, 所以把之前dao路径下的文件可以删掉, 用mapper目录来代替, 创建UserInfoMapper
>
> 当然, 继续使用dao目录也可以,此处为建议
>
> dao和mapper通常都被认为是数据库层

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

2.2.6 添加图书

约定前后端交互接口

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

```

```
[参数]
bookName=图书1&author=作者1&count=23&price=36&publish=出版社1&status=1

[响应]
""    //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)

```java
@RequestMapping("/addBook")
public String addBook(BookInfo bookInfo) {
    log.info("添加图书:{}", bookInfo);
    if (!StringUtils.hasLength(bookInfo.getBookName())
            || !StringUtils.hasLength(bookInfo.getAuthor())
            || bookInfo.getCount()==null
            || bookInfo.getPrice()==null
            || !StringUtils.hasLength(bookInfo.getPublish())
            || bookInfo.getStatus() ==null
    ) {
        return "输入参数不合法，请检查入参!";
    }
    try {
        bookService.addBook(bookInfo);
        return "";
    } catch (Exception e) {
        log.error("添加图书失败", e);
        return e.getMessage();
    }
}
```

业务层:

在BookService中补充代码

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

数据层:

创建BookInfoMapper文件

```
@Mapper
public interface BookInfoMapper {
    @Insert("insert into book_info
(book_name,author,count,price,publish,status) " +
            "values (#{bookName},#{author},#{count},#{price},#{publish},#
{status})")
    Integer insertBook(BookInfo bookInfo);
}
```

> 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

```
<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

```
function add() {
```

```
    $.ajax({
        type: "post",
        url: "/book/addBook",
        data: $("#addBook").serialize(),
        success: function (result) {
            if (result == "") {
                location.href = "book_list.html"
            } else {
                console.log(result);
                alert("添加失败:" + result);
            }
        },
        error: function (error) {
            console.log(error);
        }
    });
}
```

测试

添加图书前, 数据库内容:

```
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

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

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

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:

```
INSERT INTO `book_info` ( book_name, author, count, price, publish )
VALUES
( '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' ),
( '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' ),
( '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' ),
( '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' ),
( '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'),
( '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'),
( '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'),
( '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'),
( '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 + pagesize -1) / 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

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

}
```

我们需要根据currentPage 和pageSize ,计算出来开始索引

PageRequest修改为:

```
@Data
public class PageRequest {
    private int currentPage = 1; // 当前页
    private int pageSize = 10; // 每页中的记录数
    private int offset;

    public int getOffset() {
        return (currentPage-1) * pageSize;
    }
}
```

翻页列表结果类:

```
import lombok.Data;

import java.util.List;

@Data
public class PageResult<T> {
    private int total;//所有记录数
    private List<T> records; // 当前页数据

    public PageResult(Integer total, List<T> records) {
        this.total = total;
```

```
        this.records = records;
    }
}
```

> 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

```
[请求]
/book/getListByPage?currentPage=1&pageSize=10
Content-Type: application/x-www-form-urlencoded; charset=UTF-8

[参数]


[响应]
Content-Type: application/json

{
    "total": 25,
    "records": [{
            "id": 25,
            "bookName": "图书21",
            "author": "作者2",
            "count": 29,
            "price": 22.00,
            "publish": "出版社1",
            "status": 1,
            "statusCN": "可借阅"
    }, {
            ......
    } ]
}
```

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

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

业务层:

BookService

```java
public PageResult<BookInfo> getBookListByPage(PageRequest pageRequest) {
    Integer count = bookInfoMapper.count();
    List<BookInfo> books = bookInfoMapper.queryBookListByPage(pageRequest);
    for (BookInfo book:books){
        if (book.getStatus()==0){
            book.setStatusCN("无效");
        }else if (book.getStatus()==1){
            book.setStatusCN("可借阅");
        }else {
            book.setStatusCN("不可借阅");
        }
    }
    return new PageResult<>(count,books);
}
```

> 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enmus目录, 创建BookStatus类:

```java
public enum BookStatus {
    DELETED(0,"无效"),
    NORMAL(1,"可借阅"),
    FORBIDDEN(2,"不可借阅");

    private Integer code;
    private String name;

```

```
    BookStatus(int code, String name) {
        this.code = code;
        this.name = name;
    }

    public static BookStatus getNameByCode(Integer code){
        switch (code){
            case 0: return DELETED;
            case 1: return NORMAL;
            case 2: return FORBIDDEN;
        }
        return null;
    }

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

> 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:

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

数据层:

翻页查询SQL

```
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

```
@Select("select count(1) from book_info where status<>0")
Integer count();

@Select("select * from book_info  where status !=0 order by id desc limit #{offset}, #{pageSize}")
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

```
//获取图书列表
getBookList();
function getBookList() {
    $.ajax({
        type: "get",
        url: "/book/getListByPage",
        success: function (result) {
            if (result != null) {
                var finalHtml = "";
                for (var book of result) {
                    finalHtml += '<tr>';
                    finalHtml += '<td><input type="checkbox" name="selectBook" value="'+book.id+'" id="selectBook" class="book-select"></td>';
                    finalHtml += '<td>'+book.id+'</td>';
                    finalHtml += '<td>'+book.bookName+'</td>';
                    finalHtml += '<td>'+book.author+'</td>';
                    finalHtml += '<td>'+book.count+'</td>';
                    finalHtml += '<td>'+book.price+'</td>';
                    finalHtml += '<td>'+book.publish+'</td>';
                    finalHtml += '<td>'+book.statusCN+'</td>';
                    finalHtml += '<td><div class="op">';
                    finalHtml += '<a href="book_update.html?bookId='+book.id+'">修改</a>';
                    finalHtml += '<a href="javascript:void(0)" onclick="deleteBook('+book.id+')">删除</a>';
                    finalHtml += '</div></td>';
                    finalHtml += "</tr>";
                }

                $("tbody").html(finalHtml);
            }
        }
    });
}
```

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

```
//获取图书列表
function getBookList() {
    $.ajax({
        type: "get",
        url: "/book/getListByPage" + location.search,
        //...
    });
}
```

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>

保持一致

//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);
    }
});
```

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

```
@Data
public class PageResult<T> {
    private int total;//所有记录数
    private List<T> records; // 当前页数据
    private PageRequest pageRequest;

    public PageResult(Integer total, PageRequest pageRequest, List<T> records)
{
        this.total = total;
        this.pageRequest = pageRequest;
        this.records = records;
    }
}
```

BookService.java

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

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

```
function getBookList(currentPage) {
    $.ajax({
        type: "get",
        url: "/book/getListByPage?currentPage="+currentPage,
        success: function (result) {
            console.log(result);
            if (result != null) {
                var finalHtml = "";
                for (var book of result.records) {
                    finalHtml += '<tr>';
                    finalHtml += '<td><input type="checkbox" name="selectBook" value="' + book.id + '" id="selectBook" class="book-select"></td>';
                    finalHtml += '<td>' + book.id + '</td>';
                    finalHtml += '<td>' + book.bookName + '</td>';
                    finalHtml += '<td>' + book.author + '</td>';
                    finalHtml += '<td>' + book.count + '</td>';
                    finalHtml += '<td>' + book.price + '</td>';
                    finalHtml += '<td>' + book.publish + '</td>';
                    finalHtml += '<td>' + book.statusCN + '</td>';
                    finalHtml += '<td><div class="op">';
                    finalHtml += '<a href="book_update.html?bookId=' + book.id + '">修改</a>';
                    finalHtml += '<a href="javascript:void(0)" onclick="deleteBook(' + book.id + ')">删除</a>';
                    finalHtml += '</div></td>';
                    finalHtml += "</tr>";
                }

                $("tbody").html(finalHtml);
                //分页信息
                $("#pageContainer").jqPaginator({
                    totalCounts: result.total, //总记录数
                    pageSize: 10,    //每页的个数
                    visiblePages: 5, //可视页数
                    currentPage: result.pageRequest.currentPage,  //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);

                    }
                });

            }
        }
    });
}
```

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:

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

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
☐	25	图书21	作者2	29	22	出版社1	可借阅	修改 删除
☐	24	图书20	作者2	29	22	出版社1	可借阅	修改 删除
☐	23	图书19	作者2	29	22	出版社1	可借阅	修改 删除
☐	22	图书18	作者2	29	22	出版社1	可借阅	修改 删除
☐	21	图书17	作者2	29	22	出版社1	可借阅	修改 删除
☐	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除
☐	19	图书15	作者2	29	22	出版社1	可借阅	修改 删除
☐	18	图书14	作者2	29	22	出版社1	可借阅	修改 删除
☐	17	图书13	作者2	29	22	出版社1	可借阅	修改 删除
☐	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
☐	5	图书1	作者1	23	65	出版社1	可借阅	修改 删除
☐	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
☐	3	三体	刘慈欣	9	102.67	重庆出版社	可借阅	修改 删除
☐	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
☐	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

```
[请求]
/book/queryBookById?bookId=25

[参数]
无

[响应]
{
        "id": 25,
        "bookName": "图书21",
        "author": "作者2",
        "count": 999,
        "price": 222.00,
        "publish": "出版社1",
        "status": 2,
        "statusCN": null,
        "createTime": "2023-09-04T04:01:27.000+00:00",
        "updateTime": "2023-09-05T03:37:03.000+00:00"
}
```

> 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

```
[请求]
/book/updateBook
Content-Type: application/x-www-form-urlencoded; charset=UTF-8

[参数]
id=1&bookName=图书1&author=作者1&count=23&price=36&publish=出版社1&status=1

[响应]
""   //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:

```
@RequestMapping("/queryBookById")
public BookInfo queryBookById(Integer bookId){
    if (bookId==null || bookId<=0){
        return new BookInfo();
    }
    BookInfo bookInfo = bookService.queryBookById(bookId);
    return bookInfo;
}

@RequestMapping("/updateBook")
public String updateBook(BookInfo bookInfo) {
    log.info("修改图书:{}", bookInfo);
    try {
        bookService.updateBook(bookInfo);
        return "";
    } catch (Exception e) {
        log.error("修改图书失败", e);
        return e.getMessage();
    }
}
```

业务层:

BookService:

```
public BookInfo queryBookById(Integer bookId) {
    return bookInfoMapper.queryBookById(bookId);
}

public void updateBook(BookInfo bookInfo) {
    bookInfoMapper.updateBook(bookInfo);
}
```

数据层:

根据图书ID,查询图书信息

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

```
mybatis:
  mapper-locations: classpath:mapper/**Mapper.xml
```

最终整体的yml配置文件为:

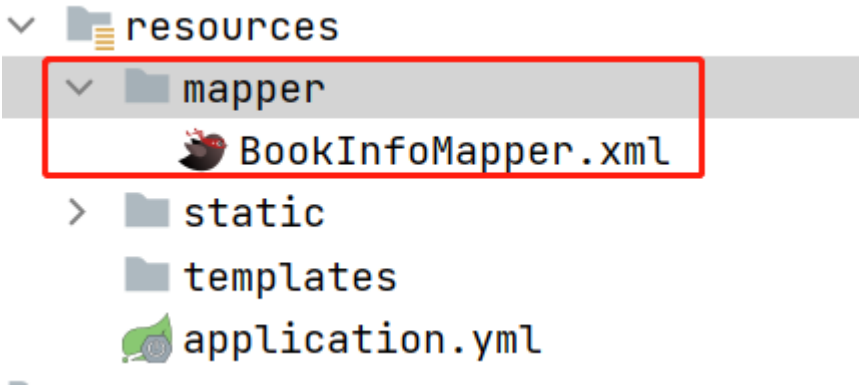
```
# 数据库连接配置
spring:
  datasource:
    url: jdbc:mysql://127.0.0.1:3306/mybatis_test?characterEncoding=utf8&useSSL=false
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
mybatis:
  configuration:
    map-underscore-to-camel-case: true #配置驼峰自动转换
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl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

定义Mapper接口: BookInfoMapper

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

xml实现:

创建BookInfoMapper.xml文件




```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.example.demo.mapper.BookInfoMapper">
    <update id="updateBook">
        update book_info
        <set>
            <if test='bookName!=null'>
                book_name = #{bookName},
            </if>
            <if test='author!=null'>
                author = #{author},
            </if>
            <if test='price!=null'>
                price = #{price},
            </if>
            <if test='count!=null'>
                count = #{count},
            </if>
            <if test='publish!=null'>
                publish = #{publish},
            </if>
            <if test='status!=null'>
                status = #{status},
            </if>
        </set>
        where id = #{id}
    </update>
</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)

```
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, 需要先从后端拿到当前图书的信息, 显示在页面上

```
$.ajax({
    type:"get",
    url: "/book/queryBookById"+location.search,
    success:function(book){
        if(book!=null){
            $("#bookName").val(book.bookName);
            $("#bookAuthor").val(book.author);
            $("#bookStock").val(book.count);
            $("#bookPrice").val(book.price);
            $("#bookPublisher").val(book.publish);
            $("#bookStatus").val(book.status);

        }
    }
});
```

补全修改图书的方法:

```
function update() {
    $.ajax({
        type: "post",
        url: "/book/updateBook",
        data: $("#updateBook").serialize(),
        success: function (result) {
            if (result == "") {
                location.href = "book_list.html"
            } else {
                console.log(result);
                alert("修改失败:" + result);
            }
        },
        error: function (error) {
            console.log(error);
```

```
        }
    });
}
```

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

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

> 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

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

此时前端js完整代码:

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

```
            $("#bookId").val(book.id);
            $("#bookName").val(book.bookName);
            $("#bookAuthor").val(book.author);
            $("#bookStock").val(book.count);
            $("#bookPrice").val(book.price);
            $("#bookPublisher").val(book.publish);
            $("#bookStatus").val(book.status);

        }
    }
});
function update() {
    $.ajax({
        type: "post",
        url: "/book/updateBook",
        data: $("#updateBook").serialize(),
        success: function (result) {
            if (result=="") {
                location.href = "book_list.html"
            } else {
                console.log(result);
                alert("修改失败:"+result);
            }
        },
        error: function (error) {
            console.log(error);
        }
    });
}
```

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
☐	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

随机修改数据, 点击确定按钮, 观察数据是否被修改

☐	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

☐	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

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

> 物理删除

```
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`

```
[请求]
/book/updateBook
Content-Type: application/x-www-form-urlencoded; charset=UTF-8

[参数]
id=1&status=0

[响应]
""    //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

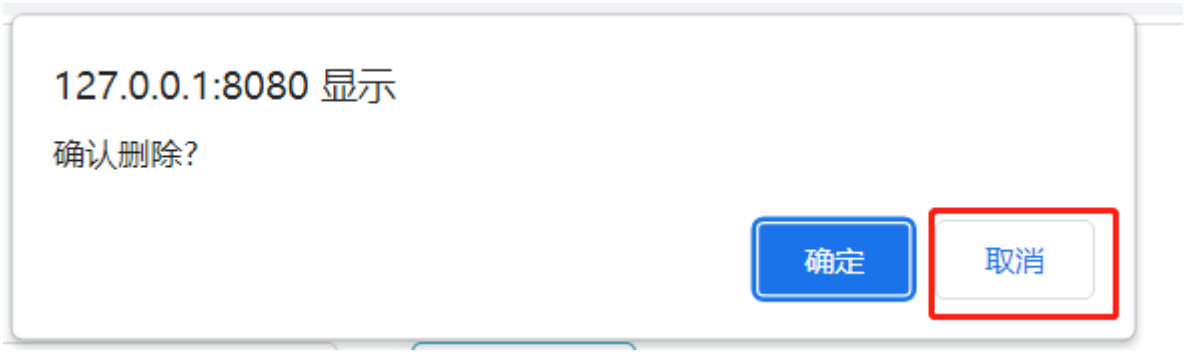
```
finalHtml += '<a href="javascript:void(0)" onclick="deleteBook(' + book.id + ')">删除</a>';
```

```
function deleteBook(id) {
    var isDelete = confirm("确认删除?");
    if (isDelete) {
        //删除图书
        $.ajax({
            type: "post",
            url: "/book/updateBook",
            data: {
                id: id,
                status: 0
            },
            success: function () {
                //重新刷新页面
                location.href = "book_list.html"
            }
        });
    }
}
```

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
☐	24	图书20	作者2	29	22	出版社1	可借阅	修改 删除
☐	23	图书19	作者2	29	22	出版社1	可借阅	修改 删除
☐	22	图书18	作者2	29	22	出版社1	可借阅	修改 删除
☐	21	图书17	作者2	29	22	出版社1	可借阅	修改 删除
☐	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
☐	23	图书19	作者2	29	22	出版社1	可借阅	修改 删除
☐	22	图书18	作者2	29	22	出版社1	可借阅	修改 删除
☐	21	图书17	作者2	29	22	出版社1	可借阅	修改 删除
☐	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

```
[请求]
/book/batchDeleteBook
Content-Type: application/x-www-form-urlencoded; charset=UTF-8

[参数]


[响应]
""    //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

```java
@RequestMapping("/batchDeleteBook")
public boolean batchDeleteBook(@RequestParam List<Integer> ids){
    log.info("批量删除图书, ids:{}",ids);
    try{
        bookService.batchDeleteBook(ids);
    }catch (Exception e){
        log.error("批量删除异常,e:",e);
        return false;
    }

    return true;
}
```

业务层:

BookService

```java
public void batchDeleteBook(List<Integer> ids) {
    bookInfoMapper.batchDeleteBook(ids);
}
```

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

```java
void batchDeleteBook(List<Integer> ids);
```

xml接口实现

```xml
<update id="batchDeleteBook">
    update book_info set status=0
    where id in
    <foreach collection="ids" item="id" separator="," open="(" close=")">
        #{id}
    </foreach>
</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

```
function batchDelete() {
    var isDelete = confirm("确认批量删除?");
    if (isDelete) {
        //获取复选框的id
        var ids = [];
        $("input:checkbox[name='selectBook']:checked").each(function () {
            ids.push($(this).val());
        });
        console.log(ids);
        //批量删除
        $.ajax({
            type: "post",
            url: "/book/batchDeleteBook?ids="+ids,
            success: function (result) {
                if (result) {
                    alert("删除成功");
                    //重新刷新页面
                    location.href = "book_list.html"
                }
            }
        });
    }
}
```

测试

选择删除的图书, 点击[批量删除]

选择	图书ID	书名	作者	数量	定价	出版社	状态	操作
☐	20	图书16	作者2	29	22	出版社1	可借阅	修改 删除
☐	19	图书15	作者2	29	22	出版社1	可借阅	修改 删除
☐	18	图书14	作者2	29	22	出版社1	可借阅	修改 删除
☐	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:

```
{
    "total": 25,
    "records": [{
            "id": 25,
            "bookName": "图书21",
            "author": "作者2",
            "count": 29,
            "price": 22.00,
            "publish": "出版社1",
            "status": 1,
            "statusCN": "可借阅"
    }, {
            ......
    } ]
}
```

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:

```
@Data
public class PageResult<T> {
    private int total;//所有记录数
    private List<T> records; // 当前页数据
    private PageRequest pageRequest;

    public PageResult(Integer total, PageRequest pageRequest, List<T> records)
{
        this.total = total;
        this.pageRequest = pageRequest;
        this.records = records;
    }
}
```

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:

```
@Data
public class PageResult<T> {
    private int status;
    private String errorMessage;

    private int total;//所有记录数
```

```
    private List<T> records; // 当前页数据
    private PageRequest pageRequest;

    public PageResult(Integer total, PageRequest pageRequest, List<T> records)
{
        this.total = total;
        this.pageRequest = pageRequest;
        this.records = records;
    }
}
```

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

```
@Data
public class Result<T> {
    private int status;
    private String errorMessage;
    private T data;
}
```

> 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

```
public enum ResultStatus {

    SUCCESS(200),
    UNLOGIN(-1),
    FAIL(-2);

    private Integer code;

    ResultStatus(int code) {
        this.code = code;
    }


    public Integer getCode() {
        return code;
    }

```

```
    public void setCode(Integer code) {
        this.code = code;
    }

}
```

修改Result, 并添加一些常用方法

```
import com.example.demo.enums.ResultStatus;
import lombok.Data;

@Data
public class Result {
    private ResultStatus status;
    private String errorMessage;
    private Object data;

    /**
     * 业务执行成功时返回的方法
     *
     * @param data
     * @return
     */
    public static Result success(Object data) {
        Result result = new Result();
        result.setStatus(ResultStatus.SUCCESS);
        result.setErrorMessage("");
        result.setData(data);
        return result;
    }

    /**
     * 业务执行失败时返回的方法
     *
     * @param
     * @return
     */
    public static Result fail(String msg) {
        Result result = new Result();
        result.setStatus(ResultStatus.FAIL);
        result.setErrorMessage(msg);
        result.setData("");
        return result;
    }

```

```
    /**
     * 业务执行失败时返回的方法
     *
     * @param
     * @return
     */
    public static Result unlogin() {
        Result result = new Result();
        result.setStatus(ResultStatus.UNLOGIN);
        result.setErrorMessage("用户未登录");
        result.setData(null);
        return result;
    }

}
```

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

```
@RequestMapping("/getListByPage")
public Result getListByPage(PageRequest pageRequest, HttpSession session) {
    log.info("获取图书列表, pageRequest:{}", pageRequest);
    //判断用户是否登录
    if (session.getAttribute("session_user_key")==null){
        return Result.unlogin();
    }
    UserInfo userInfo = (UserInfo) session.getAttribute("session_user_key");
    if (userInfo==null || userInfo.getId()<0 ||
"".equals(userInfo.getUserName())){
        return Result.unlogin();
    }
    //用户登录, 返回图书列表
    PageResult<BookInfo> pageResult =
bookService.getBookListByPage(pageRequest);
    return Result.success(pageResult);
}
```

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

```java
public class Constants {
    public static final String SESSION_USER_KEY = "session_user_key";
}
```

> 常量名命名规则:
>
> 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

```java
@RequestMapping("/login")
public boolean login(String name, String password, HttpSession session){
    //账号或密码为空
    if (!StringUtils.hasLength(name) || !StringUtils.hasLength(password)){
        return false;
    }
    UserInfo userInfo = userService.queryUserByName(name);
    if (userInfo == null){
        return false;
    }
    //账号密码正确
    if(userInfo!=null && password.equals(userInfo.getPassword())){
        //存储在Session中
        userInfo.setPassword("");
        session.setAttribute(Constants.SESSION_USER_KEY,userInfo);
        return true;
    }
    //账号密码错误
    return false;
}
```

图书列表接口:

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

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

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):

```
function getBookList() {
        //...
        success: function (result) {

            //真实前端代码需要分的更细一点，此处不做判断
            if (result == null || result.data == null) {
                location.href = "login.html";
                return;
            }

            //....
            var data = result.data;
            $("#pageContainer").jqPaginator({
                totalCounts: data.total, //总记录数
                pageSize: 10,    //每页的个数
                visiblePages: 5, //可视页数
                currentPage: data.pageRequest.currentPage,  //当前页码
                //....
            });
        }
}
```

完整代码如下:

```
getBookList();
function getBookList() {
    $.ajax({
        type: "get",
        url: "/book/getListByPage" + location.search,
        success: function (result) {
            console.log(result);
            //真实前端代码需要分的更细一点，此处不做判断
            if (result == null || result.data == null) {
                location.href = "login.html";
                return;
            }

            var finalHtml = "";
            var data = result.data;
            for (var book of data.records) {
                finalHtml += '<tr>';
                finalHtml += '<td><input type="checkbox" name="selectBook"
value="' + book.id + '" id="selectBook" class="book-select"></td>';
                finalHtml += '<td>' + book.id + '</td>';
                finalHtml += '<td>' + book.bookName + '</td>';
                finalHtml += '<td>' + book.author + '</td>';
                finalHtml += '<td>' + book.count + '</td>';
                finalHtml += '<td>' + book.price + '</td>';
                finalHtml += '<td>' + book.publish + '</td>';
                finalHtml += '<td>' + book.statusCN + '</td>';
                finalHtml += '<td><div class="op">';
                finalHtml += '<a href="book_update.html?bookId=' + book.id +
'">修改</a>';
                finalHtml += '<a href="javascript:void(0)"
onclick="deleteBook(' + book.id + ')">删除</a>';
                finalHtml += '</div></td>';
                finalHtml += "</tr>";
            }

            $("tbody").html(finalHtml);

            $("#pageContainer").jqPaginator({
                totalCounts: data.total, //总记录数
                pageSize: 10,    //每页的个数
                visiblePages: 5, //可视页数
                currentPage: data.pageRequest.currentPage,  //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) {
                    if (type != 'init') {
                        location.href = "book_list.html?currentPage=" + page;
                    }
                }
            });

        }
    });
}
```

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

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. 总结

1. 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.
2. 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
3. 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.