电影售票管理系统

1. 需求分析

1.1 意义和用途

设计电影售票系统具有多重意义和用途，它可以为电影院、用户以及电影产业链的不同参与者提供各种好处。以下是设计电影售票系统的主要意义和用途：

1. 提高用户体验：电影售票系统可以为用户提供便捷的购票渠道，包括浏览电影热点，在线购票等功能，提高了用户购票的便利性和体验。

2. 便捷自动化管理：系统可以自动处理票务信息、座位管理、电影排片、票价调整等任务，降低了电影院管理的人工工作量，提高了效率。还提供了用户管理功能，可让管理人员对用户账号进行封禁和编辑处理。

3. 提高票务销售效率：售票系统可以自动处理大量的票务销售和交易，加速了电影院售票流程，减少了排队等待时间。

4. 数据分析和决策支持：系统可以收集和分析用户购票行为、电影票房等数据，为电影院提供决策支持，包括电影排片、票价策略等方面的优化。

5. 安全和准确性：售票系统可以提供安全的支付和数据传输机制，减少了票务信息丢失和错误的风险。

6. 促进市场竞争：电影售票系统可以让不同的电影院之间进行竞争，吸引更多观众，提高电影产业的竞争力。

1.2 任务和功能

1. 用户注册和登录：用户可以注册账户、登录系统，以便购票和查看订单。

2. 电影信息管理：管理员可以添加、编辑和删除电影信息，包括电影名称、导演、演员、上映日期、时长等。

3. 电影及人员管理：管理员可以安排电影的排片时间和场次， 以及注册进入售票系统的人员管理

4. 购票功能：用户可以选择电影、场次、影厅，进行购票并支付。

5. 订单管理：用户可以查看当前订单、退票等操作。

6. 支付功能：即订单提交给后台，存入数据库。

7. 安全性：确保用户信息和支付信息的安全，防止数据泄露和网络攻击。

8. 用户体验：提供友好的用户界面，确保用户能够轻松快速地完成购票流程。

1.3 要求和目标：

1. 提高售票效率，减少排队等待时间。

2. 提供电影信息的完整性和准确性。

3. 提供用户友好的界面，易于操作。

4. 后台提供良好的数据库支持。

* 1. 功能模块图及功能简介

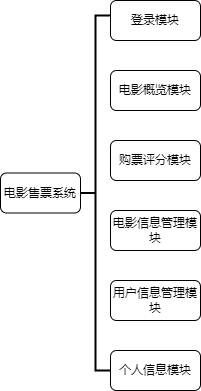


图1.4 功能模块图

1. 登录模块：

用户可以输入用户名和密码登录系统。系统验证用户的登录信息，并在验证成功后允许用户访问系统的其他功能模块。

登录模块同时包括用户注册功能，以确保新用户也能正常使用该系统

2. 电影概览模块：

用户可以浏览当前正在热映的电影列表。用户也可以查看每部电影的基本信息，如电影名称、导演、演员阵容、上映时间等。

3. 购票评分模块：

用户可以选择电影并购买电影票。在购票界面选择放映时间和想去的影院及影厅，完成电影票的购买。

用户也可以对浏览界面的电影进行评分。

4. 电影信息管理模块：

管理员可以添加、编辑和删除电影信息。包括更新电影的上映时间、票价和电影链接等信息。

5. 用户信息管理模块：

管理员可以查看和管理系统中的用户信息，如昵称，注册的邮箱，也可以禁用，删除和增添用户账户。

普通用户不能编辑自己的个人信息。

6. 个人信息模块：

用户可以查看自己的个人信息，也可以在主页上传头像更新在个人资料中。

这些功能模块共同构成了一个电影售票系统，为用户提供了浏览电影、购票、评分和管理个人信息的便利。管理员则能够管理电影信息和用户信息，确保系统的正常运行。

1. 概念结构设计

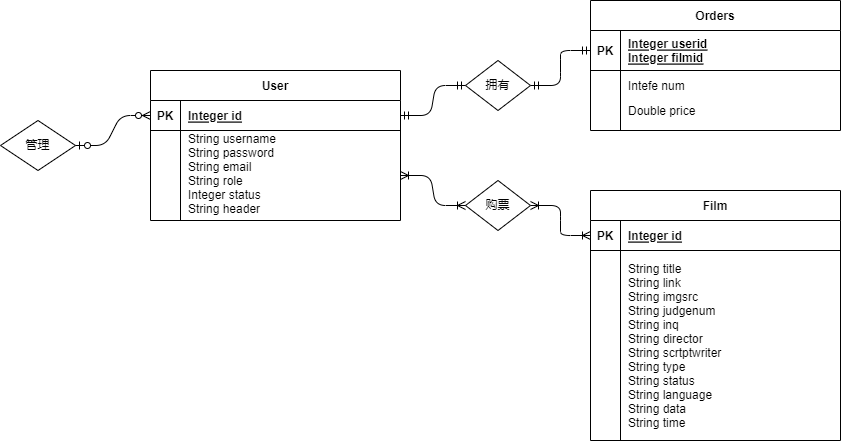


图2 工程E-R图

1. 逻辑结构设计

**表1 用户表**

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名 | 含义 | 类型 | 说明 |
| id | 用户id | Integer | 主键，每次自增1 |
| username | 用户名称 | String |  |
| password | 用户密码 | String |  |
| email | 用户邮箱 | String |  |
| role | 用户权限 | String | 分为超级管理员和普通用户 |
| header | 用户头像 | String | 用户头像路径 |
| status | 账号状态 | Integer | 0 封禁，1 可用 |

**表2 订单表**

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名 | 含义 | 类型 | 说明 |
| userid | 用户id | Integer | 主键，外键，来源于用户表 |
| filmid | 电影id | Integer | 主键，外键，来源于电影表 |
| num | 影票数量 | Integer |  |
| price | 单价 | Double | 影票的单价 |

**表3 电影表**

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名 | 含义 | 类型 | 说明 |
| id | 电影id | Integer | 主键，每次自增1 |
| title | 电影标题 | String |  |
| link | 电影链接 | String | 链接到豆瓣该电影界面 |
| imgsrc | 电影海报 | String | 电影封面图片链接 |
| rating | 电影评分 | String |  |
| judgenum | 评价人数 | String |  |
| inq | 影片概述 | String |  |
| director | 导演 | String |  |
| scriptweiter | 编剧名称 | String |  |
| type | 电影类型 | String |  |
| status | 电影地区 | String |  |
| language | 语言 | String |  |
| date | 上映日期 | String |  |
| time | 上映时间 | String |  |

1. 开发工具

4.1开发环境

操作系统：Windows 10

开发工具：IntelliJ IDEA 2021.3.3

数据库：MySQL 8.0

前端框架：Vue.js 3

后端框架：Spring Boot 2.5.4

版本控制：Git

构建工具：Maven

Web 服务器：Tomcat

4.2编程语言

前端：JavaScript、HTML、CSS

后端：Java

4.3开发工具和库的版本

IntelliJ IDEA：Version 2021.3.3

MySQL：Version 8.0

Vue.js：Version 3

Spring Boot：Version 2.5.4

Git：Version 2.32.0

Maven：Version 3.8.4

Tomcat：Version 9.0.53

1. 具体实现

5.1 运行截图



图5.1 登录界面



图5.2 注册界面

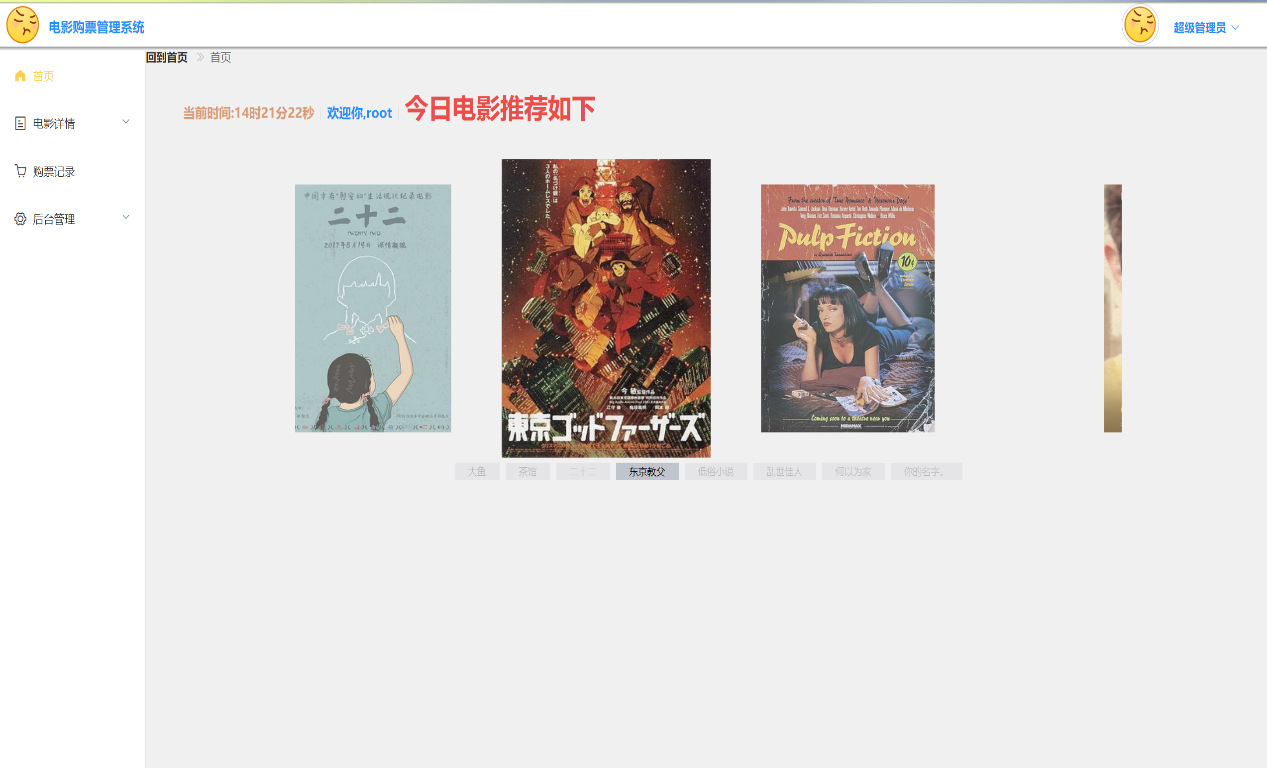


图5.3 首页

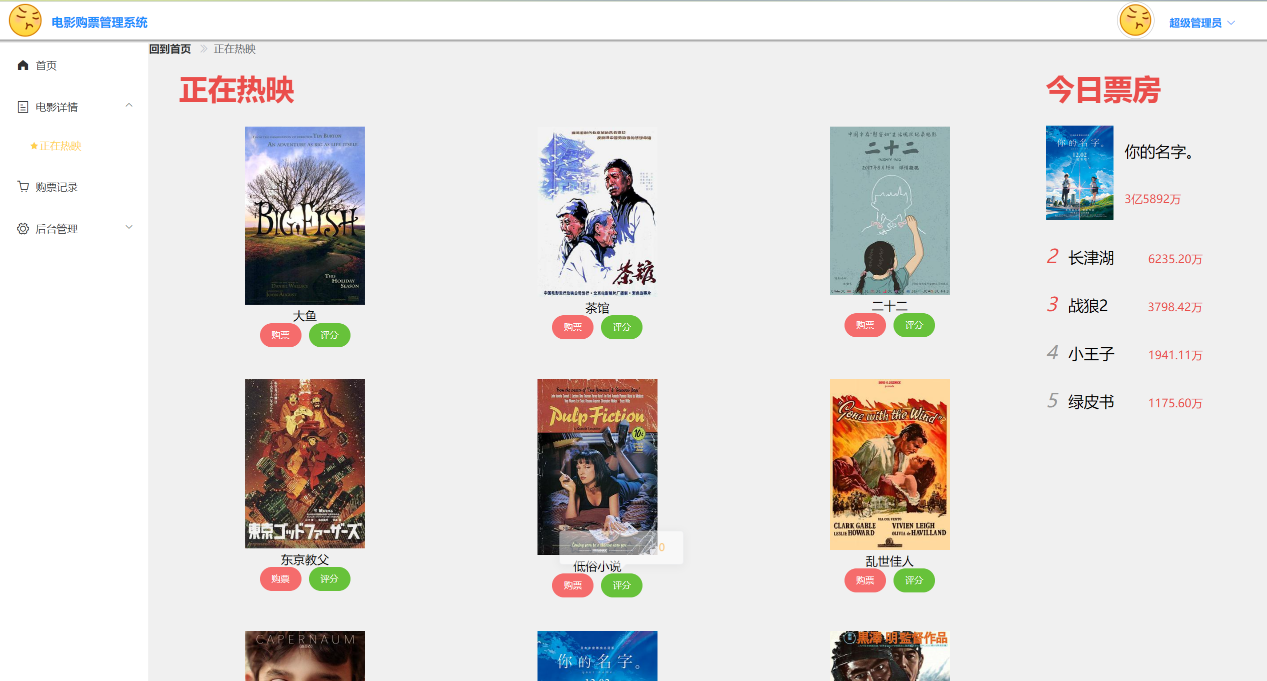


图5.4 电影概览

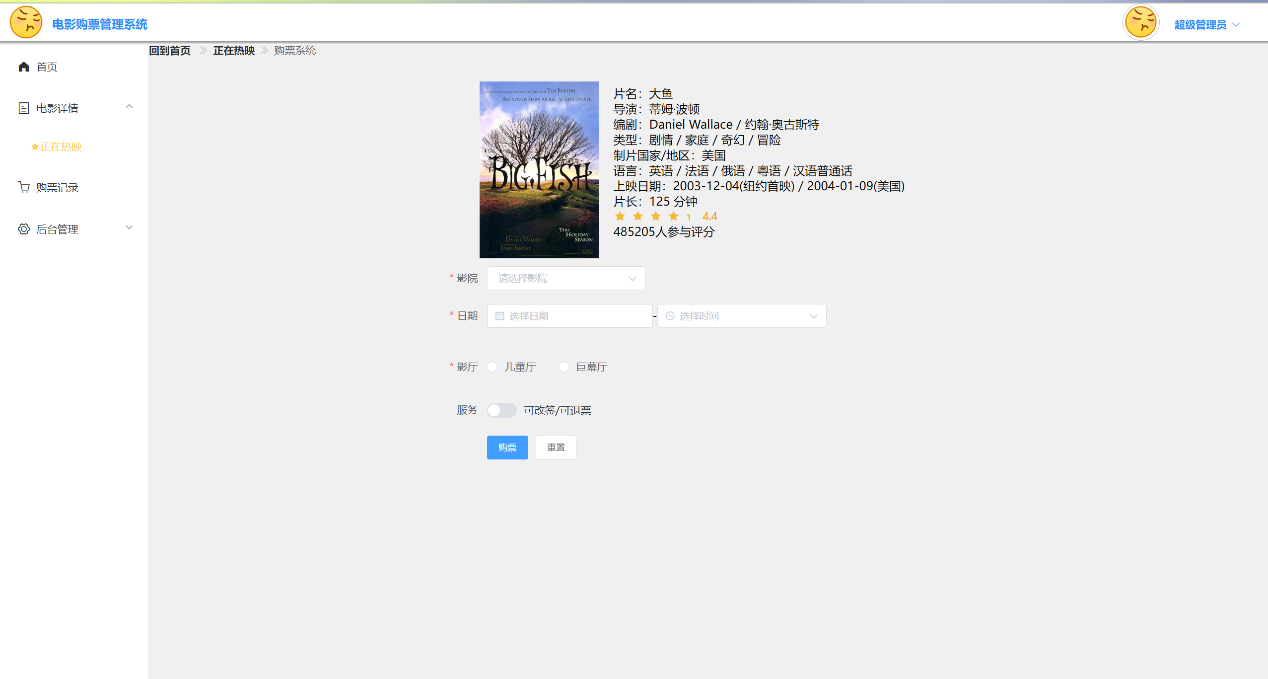


图5.5 电影购票

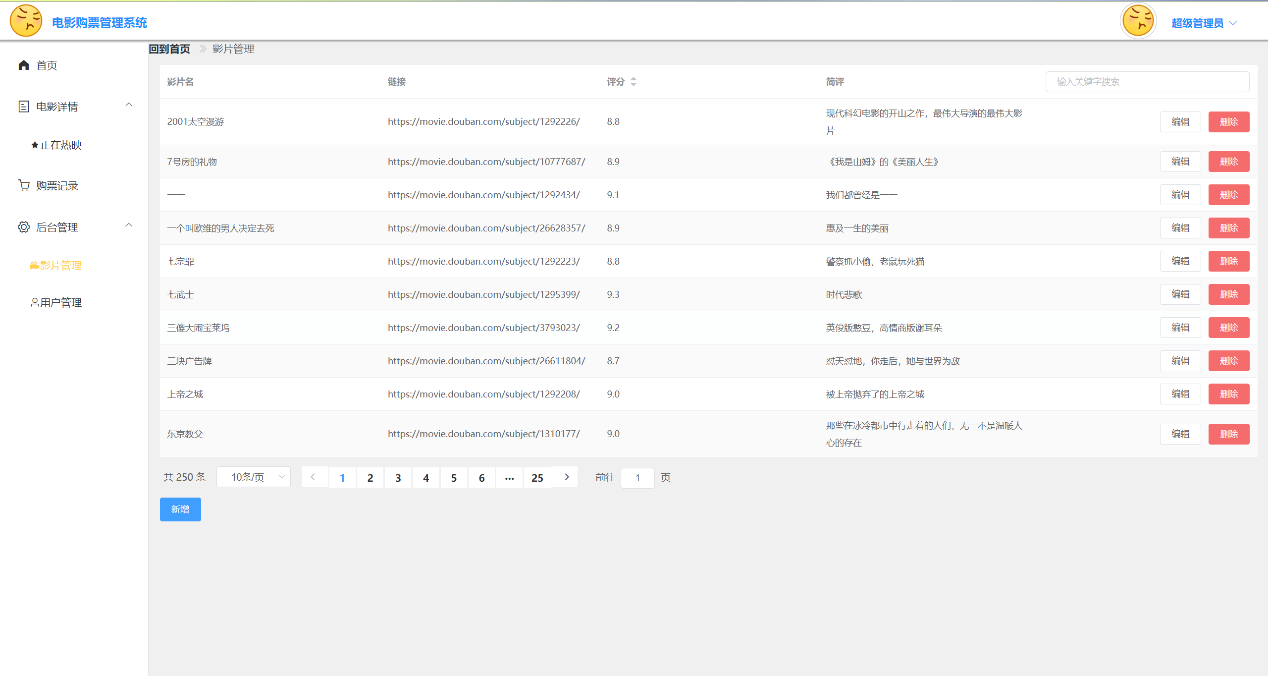


图5.6 电影管理界面

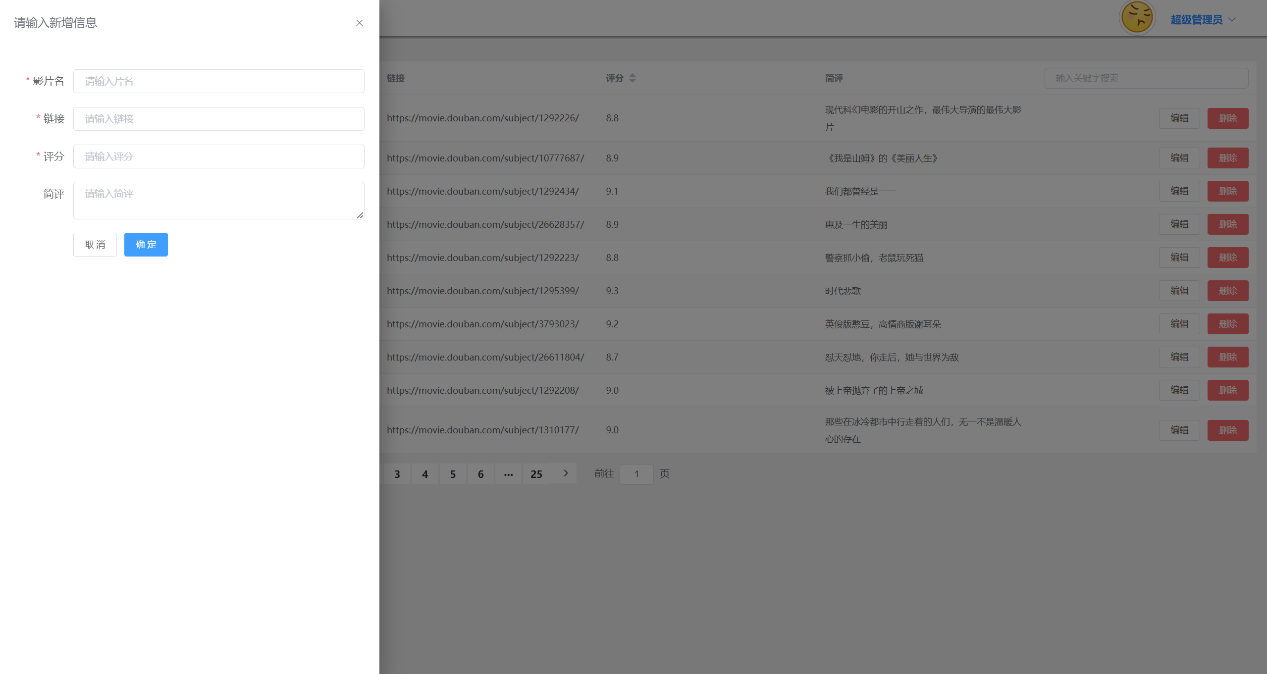


图5.7 电影新增界面

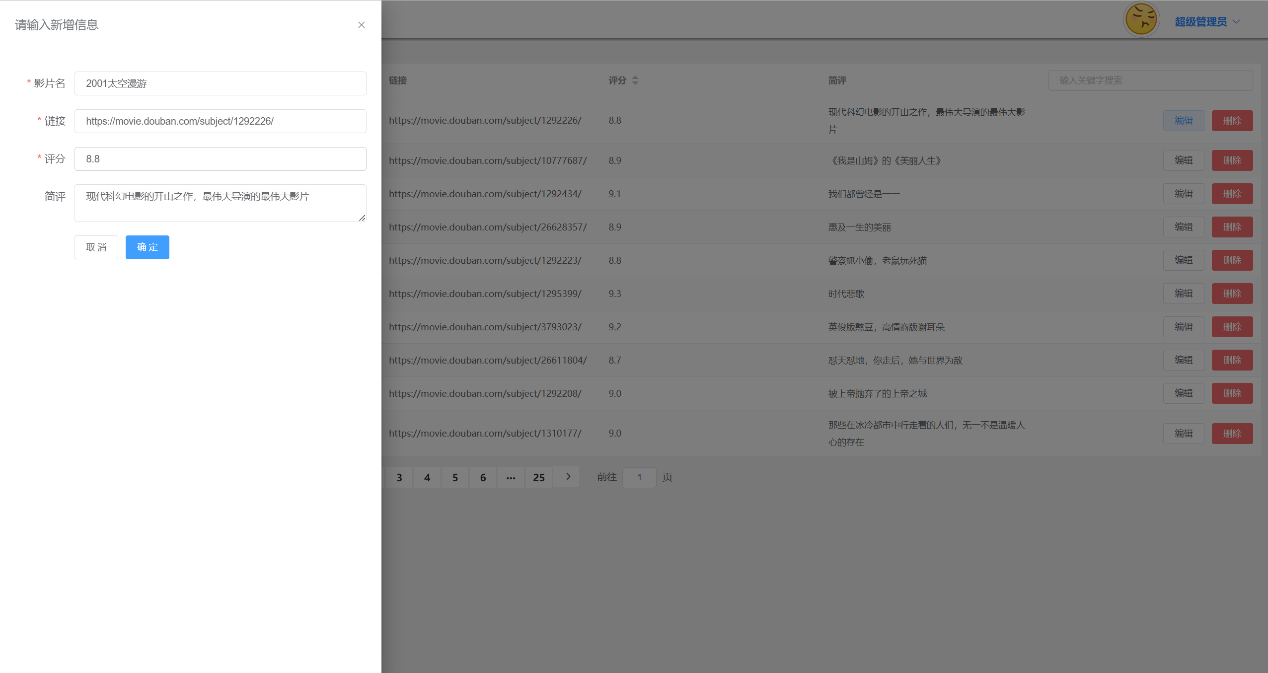


图5.8 电影编辑界面

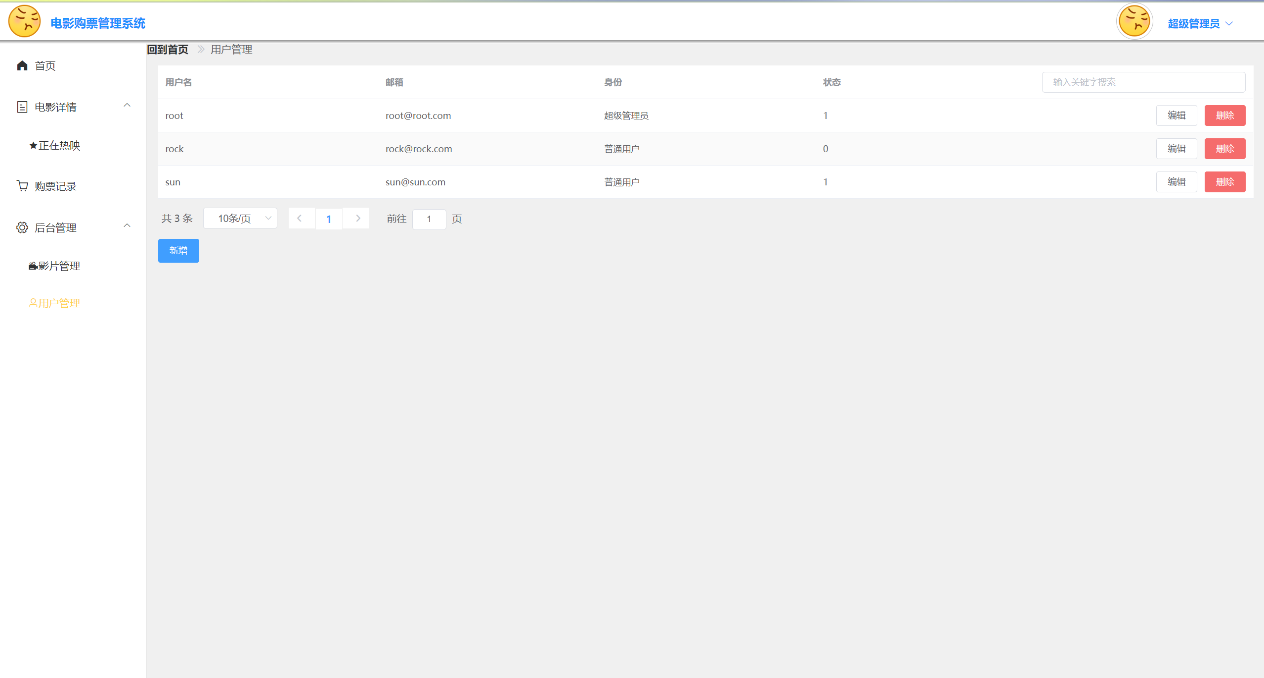


图5.9 用户管理界面

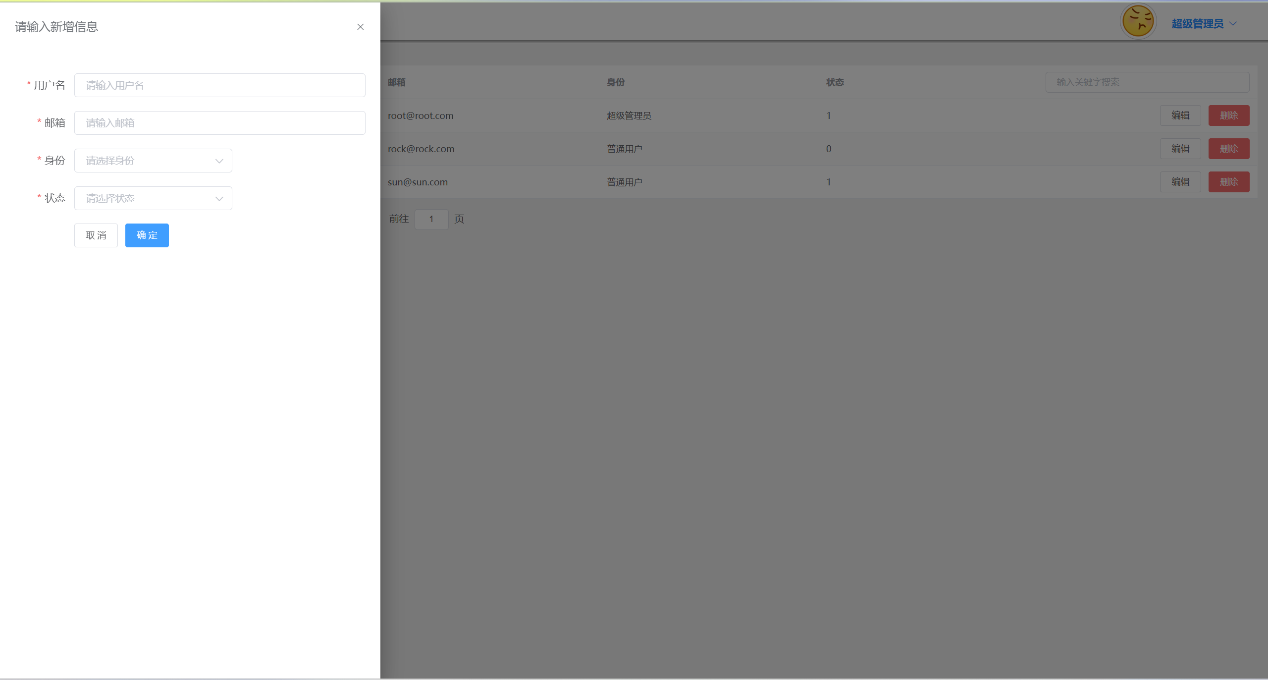


图5.10 新增用户界面

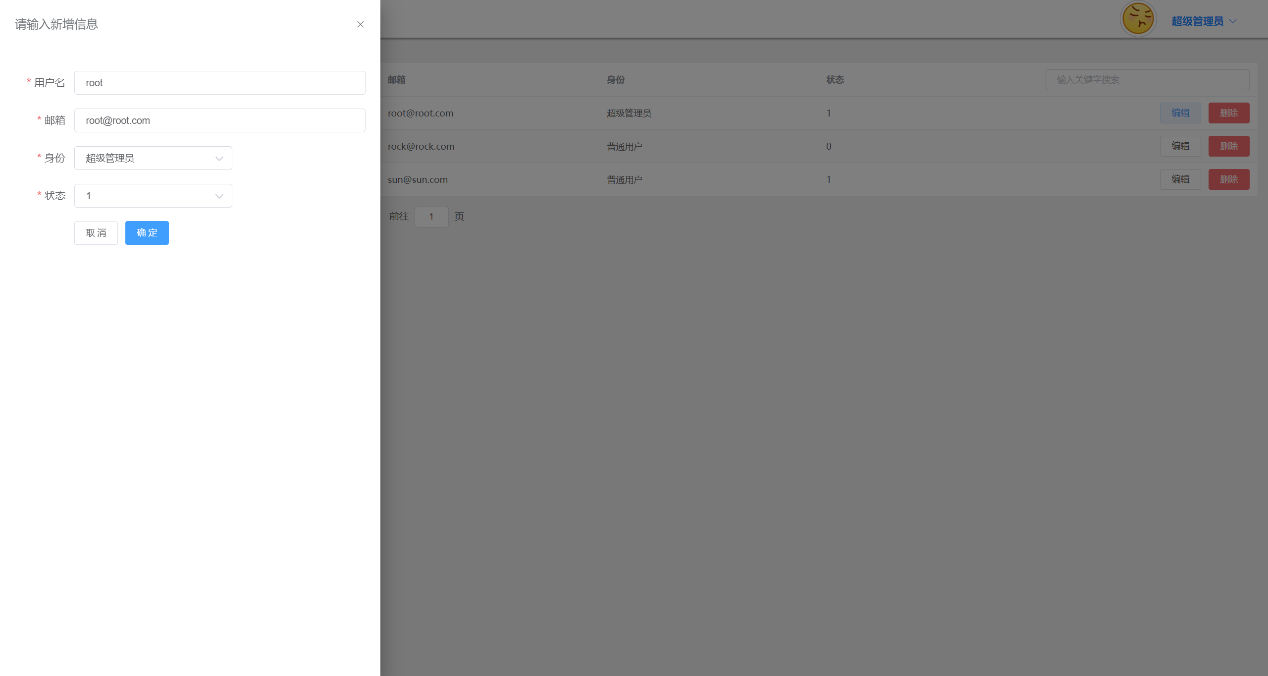


图5.11 用户编辑界面

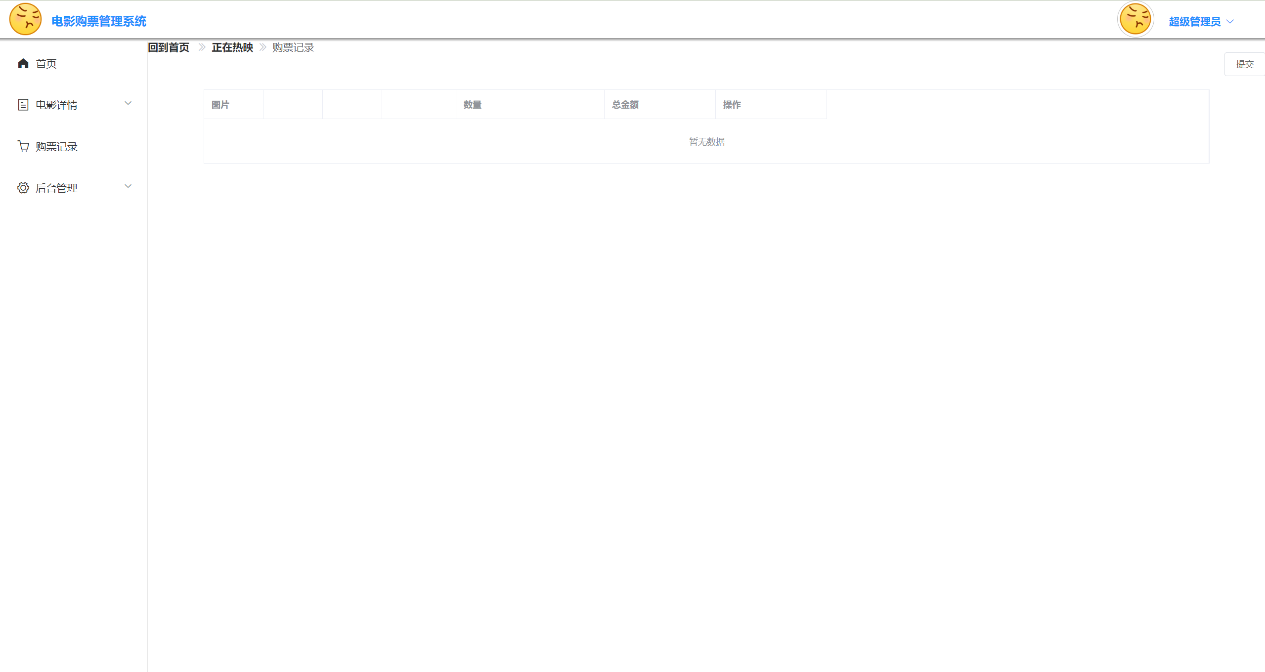


图5.12 购票记录

5.2 后端代码

5.2.1 实体类

根据需求需要三张表存储数据，因此创建实体类分为三个，分别是User，Order和Film。



图5.2.1.1 电影实体类

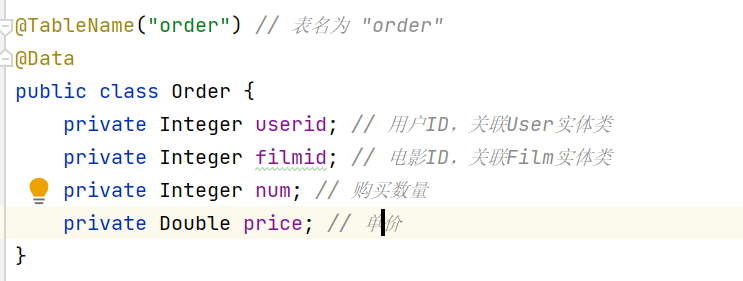


图5.2.1.2 订单实体类



图5.2.1.3用户实体类

5.2.2 Mapper接口

位三个实体类创建相应的Mapper接口，只需继承BaseMapper，BaseMapper 是 MyBatis-Plus 框架提供的一个接口，用于简化 MyBatis 的数据库操作，主要用于进行常见的 CRUD（增删改查）操作。它的主要作用如下：

1. 继承方法： BaseMapper 接口提供了许多通用的数据库操作方法，包括插入、更新、删除、查询单个实体、查询列表、分页查询等。这些方法可以通过继承 BaseMapper 来使用，而不需要手动编写 SQL 语句。
2. 自动映射： MyBatis-Plus 会根据实体类的命名规则和数据库表的映射关系，自动执行 SQL 操作，无需手动编写 SQL。
3. 通用条件构造器： MyBatis-Plus 还提供了一个通用条件构造器（Wrapper），可以方便地构建条件查询，包括等于、模糊查询、排序、分页等功能。
4. 自动填充： 可以配置自动填充策略，使得在插入和更新操作时，某些字段的值可以自动填充，例如创建时间、更新时间等。
5. 逻辑删除： 支持逻辑删除，可以在删除数据时将数据标记为已删除而不是物理删除，以满足软删除的需求。
6. 多租户支持： 支持多租户模式，可以在多租户系统中使用，区分不同租户的数据。

使用 BaseMapper 可以大大简化数据库操作的代码，提高开发效率,也简化了MyBatis写sql语句或配置xml文件的方式实现增删改查，在简易的系统开发中于Wrapper结合后甚至可以不用再构建服务逻辑

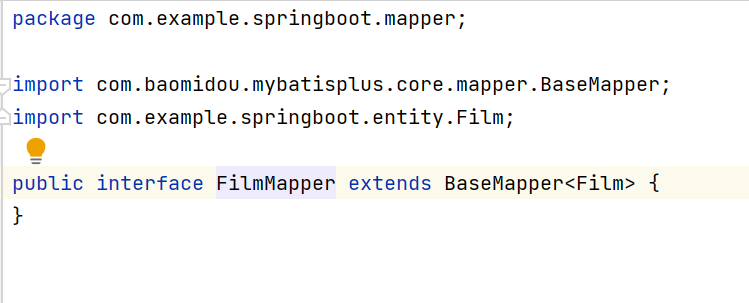
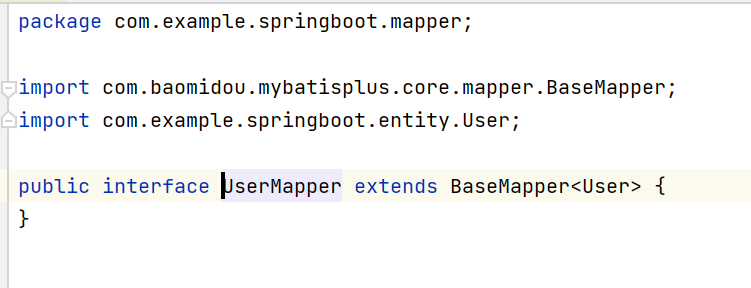
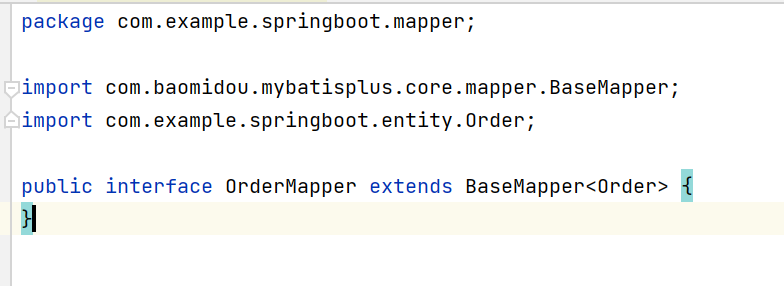


图5.2.2 Mapper接口代码

5.2.3 控制器类

控制器是后端处理请求的重要部分，作用如下

1. 路由请求： 控制器类根据不同的请求URL（路由）来确定如何处理请求。每个控制器类通常会处理一个或多个相关的路由。路由可以映射到不同的控制器方法。
2. 请求处理： 控制器方法负责处理特定的HTTP请求。例如，GET请求可能用于检索资源，而POST请求用于创建新资源。控制器方法执行相关的操作，如查询数据库、处理数据、调用服务等。
3. 参数解析： 控制器方法可以从HTTP请求中提取参数，这些参数可以是URL路径参数、查询字符串参数、请求正文中的数据等。这些参数用于控制器方法的操作。
4. 业务逻辑： 控制器方法通常包含与业务逻辑相关的代码。它们可能调用服务层、数据访问层或其他组件来执行业务操作，如验证用户、处理订单等。
5. 返回响应： 控制器方法生成HTTP响应并将其返回给客户端。响应可以是HTML页面、JSON数据、XML数据等，具体取决于客户端请求的类型。
6. 异常处理： 控制器可以捕获并处理异常，然后返回适当的错误响应给客户端。这有助于应对应用程序中可能出现的问题。

以下是UserController的代码：

1. @RestController
2. @RequestMapping("/user")
3. **public** **class** UserController {
4. @Resource
5. //依赖注入：@Resource 注解可以将依赖的资源注入到类中，使得类能够使用这些资源，而不需要手动创建或获取。
6. //    注入名称匹配：@Resource 注解可以根据名称匹配来注入资源，默认情况下会根据变量名或方法名来匹配。如果没有明确指定 name 属性，它会按照以下顺序匹配：
7. //    根据变量名（字段名）匹配。
8. //    根据方法名匹配（去掉 get、set 等前缀，将剩余部分作为名称匹配）。
9. //    支持类型匹配：@Resource 注解也可以根据类型匹配来注入资源，如果名称匹配失败，它会尝试根据类型匹配。
10. //    按照名称注入：通过 name 属性指定要注入的资源名称，这样可以明确指定要注入的资源，不依赖于默认的名称匹配。
11. UserMapper userMapper; // 注入 UserMapper 对象，用于数据库操作
12. //     分页查询用户列表，pageNum  当前页码，默认为 1，pageSize 每页显示数量，默认为 10 ，search   搜索关键字，默认为空字符串，@return 分页查询结果
13. @GetMapping
14. **public** Result<?> findPage(@RequestParam(defaultValue = "1") Integer pageNum,
15. @RequestParam(defaultValue = "10") Integer pageSize,
16. @RequestParam(defaultValue = "") String search) {
17. LambdaQueryWrapper<User> wrapper = Wrappers.<User>lambdaQuery();
18. **if** (StringUtils.isNotBlank(search)) {
19. wrapper.like(User::getUsername, search);
20. }
21. Page<User> userPage = userMapper.selectPage(**new** Page<User>(pageNum, pageSize), wrapper);
22. **return** Result.success(userPage);
23. }
24. //用户登录接口，user 包含用户名和密码的 User 对象，return登录结果，成功返回用户信息，失败返回错误信息
25. @PostMapping("/login")
26. **public** Result<?> login(@RequestBody User user) {
27. User res = userMapper.selectOne(Wrappers.<User>lambdaQuery().eq(User::getUsername, user.getUsername()).eq(User::getPassword, user.getPassword()));
28. **if** (res == **null**) {
29. **return** Result.error("-1", "用户名或密码错误");
30. }
31. **return** Result.success(res);
32. }
33. //根据用户名查询用户信息
34. @GetMapping("/selectUsername")
35. **public** User selectUsername(@RequestParam("username") String username) {
36. QueryWrapper wrapper = **new** QueryWrapper();
37. wrapper.eq("username", username);
38. User user = userMapper.selectOne(wrapper);
39. **return** user;
40. }
41. // 插入新用户，@return 插入结果，成功返回成功信息，失败返回错误信息
42. @PostMapping
43. **public** Result insert(@RequestBody User user) {
44. userMapper.insert(user);
45. **return** Result.success();
46. }
47. //更新用户信息，user 包含用户信息的 User 对象，return 更新结果，成功返回成功信息，失败返回错误信息
48. @PutMapping
49. **public** Result update(@RequestBody User user) {
50. userMapper.updateById(user);
51. **return** Result.success();
52. }
53. //根据用户ID删除用户，@param id 用户ID，return 删除结果，成功返回成功信息，失败返回错误信息
54. @DeleteMapping("/{id}")
55. **public** Result<?> update(@PathVariable Long id) {
56. userMapper.deleteById(id);
57. **return** Result.success();
58. }
59. }

@RestController: 这个注解告诉Spring框架，这是一个RESTful风格的控制器，用于处理HTTP请求，并将响应直接返回给客户端，而不是渲染HTML视图。这使得它适用于构建Web API。

@RequestMapping("/user"): 这个注解定义了基本的请求映射路径，即以 "/user" 开头的所有HTTP请求都会由这个控制器处理。"/user/login"将映射到登录方法。

@Resource: 这个注解用于依赖注入，它将 UserMapper 对象注入到 UserController 中，以便在控制器中可以使用该对象执行数据库操作。

@GetMapping: 这个注解定义了一个HTTP GET请求的处理方法。具体来说，findPage 方法用于分页查询用户列表。它接受三个请求参数：pageNum（当前页码）、pageSize（每页显示数量）和search（搜索关键字）。

@PostMapping("/login"): 这个注解定义了一个HTTP POST请求的处理方法，即用户登录方法。用户可以通过提交用户名和密码的 User 对象进行登录。如果登录成功，将返回用户信息；如果失败，将返回错误信息。

@GetMapping("/selectUsername"): 这个注解定义了一个HTTP GET请求的处理方法，用于根据用户名查询用户信息。它接受一个请求参数 username，通过这个参数来查询用户信息。

@PostMapping: 这个注解定义了一个HTTP POST请求的处理方法，用于插入新用户。它接受一个 User 对象作为请求体，将这个用户信息插入到数据库中。

@PutMapping: 这个注解定义了一个HTTP PUT请求的处理方法，用于更新用户信息。它接受一个 User 对象作为请求体，将更新后的用户信息保存到数据库中。

@DeleteMapping("/{id}"): 这个注解定义了一个HTTP DELETE请求的处理方法，用于根据用户ID删除用户。它接受一个路径参数 id，根据这个参数来删除相应的用户。

findPage 方法：这个方法的主要作用是分页查询用户列表。它接收三个请求参数，pageNum 表示当前页码，pageSize 表示每页显示的用户数量，search 用于进行用户名称的模糊搜索。这个方法适用于在用户管理界面中展示用户列表，并且支持分页和搜索功能。

login 方法：这个方法用于处理用户登录请求。它接受一个包含用户名和密码的 **User** 对象作为请求体。在用户提交登录信息后，系统会验证用户名和密码是否匹配数据库中的用户信息，如果匹配成功，将返回用户信息，表示登录成功；如果匹配失败，将返回错误信息，通常是用户名或密码错误。这个方法用于用户身份验证。

selectUsername 方法：这个方法的作用是根据用户名查询用户信息。它接受一个 username 参数，通过该参数查询数据库中对应用户名的用户信息。这个方法用于检查用户名是否已经被注册，或者用于获取特定用户名的用户信息。

insert 方法：这个方法用于插入新用户信息到数据库中。它接受一个包含用户信息的 User 对象作为请求体，然后将该用户信息保存到数据库中。用于用户注册功能。

update 方法：这个方法的作用是更新用户信息。它接受一个包含用户信息的 User 对象作为请求体，然后根据用户ID在数据库中更新相应的用户信息。这个方法用于用户个人信息的修改功能。

update 方法（删除用户）：这个方法用于根据用户ID删除用户。它接受一个路径参数 id，根据该参数在数据库中删除对应ID的用户信息。用于管理员删除用户账号的操作。

然后是FilmComtroller的代码

1. @RestController
2. @RequestMapping("/film")
3. **public** **class** FilmController {
5. @Resource
6. FilmMapper filmMapper;
8. @GetMapping
9. **public** Result<?> findPage(@RequestParam(defaultValue = "1") Integer pageNum,
10. @RequestParam(defaultValue = "10") Integer pageSize,
11. @RequestParam(defaultValue = "") String search){
12. LambdaQueryWrapper<Film> wrapper = Wrappers.<Film>lambdaQuery();
13. **if**(StringUtils.isNotBlank(search)){
14. wrapper.like(Film::getTitle, search);
15. }
16. Page<Film> filmPage = filmMapper.selectPage(**new** Page<Film>(pageNum, pageSize), wrapper);
17. **return** Result.success(filmPage);
18. }
20. @GetMapping("/selectTitle")
21. **public** Film selectFilm(@RequestParam("title") String title){
22. QueryWrapper wrapper = **new** QueryWrapper();
23. wrapper.eq("title", title);
24. Film film = filmMapper.selectOne(wrapper);
25. **return** film;
26. }
27. @PostMapping
28. **public** Result insert(@RequestBody Film film){
29. filmMapper.insert(film);
30. **return** Result.success();
31. }
32. @PutMapping
33. **public** Result update(@RequestBody Film film){
34. filmMapper.updateById(film);
35. **return** Result.success();
36. }
38. @DeleteMapping("/{id}")
39. **public** Result<?> update(@PathVariable Long id){
40. filmMapper.deleteById(id);
41. **return** Result.success();
42. }
43. }

findPage 方法：这个方法用于分页查询电影信息。它接收三个请求参数：pageNum 表示当前页码，pageSize 表示每页显示的电影数量，search 用于进行电影标题的模糊搜索。方法内部使用 LambdaQueryWrapper 创建查询条件，如果有搜索关键字，它将进行电影标题的模糊匹配。接着，通过 filmMapper.selectPage 方法分页查询电影信息，并将结果封装成 Result 对象返回。

@GetMapping("/selectTitle") 方法：这个方法用于根据电影标题查询电影信息。它接收一个 title 参数，通过该参数查询数据库中与标题匹配的电影信息。方法内部使用 QueryWrapper 创建查询条件，然后通过 filmMapper.selectOne 方法查询电影信息，并将结果返回。

@PostMapping 方法：这个方法用于插入新的电影信息到数据库中。它接收一个包含电影信息的 Film 对象作为请求体，然后将该电影信息保存到数据库中。

@PutMapping 方法：这个方法用于更新电影信息。它接收一个包含电影信息的 Film 对象作为请求体，然后根据电影的 ID在数据库中更新相应的电影信息。

@DeleteMapping("/{id}") 方法：这个方法用于根据电影ID从数据库中删除电影信息。

再是OrderController的代码

1. @RestController
2. @RequestMapping("/cart")
3. **public** **class** OrderController {
5. @Resource
6. OrderMapper orderMapper; // 注入 orderMapper 对象，用于数据库操作
7. @GetMapping("/list/{userid}")
8. **public** List<Order> selectUsername(@RequestParam("userid") String userid) {
9. QueryWrapper wrapper = **new** QueryWrapper();
10. wrapper.eq("userid", userid);
11. List<Order> Order = orderMapper.selectList(wrapper);
12. **return** Order;
13. }
15. // 插入新订单
16. @PostMapping("/insert")
17. **public** Result<?> insertOrder(@RequestBody Order order) {
18. **if** (order != **null**) {
19. orderMapper.insert(order);
20. **return** Result.success("订单插入成功");
21. }
22. **return** Result.error("-1", "订单插入失败");
23. }
24. }

selectUsername方法的作用是根据用户ID查询订单列表。

@GetMapping("/list/{userid}")注解：\*\* 这个方法处理 HTTP GET 请求，并指定了请求路径为 "/list/{userid}"，其中 `{userid} 是路径变量，表示要查询的用户的ID。

@RequestParam("userid")` 注解：\*\* 用于接收名为 "userid" 的请求参数，这个参数是从请求路径中获取的。使用 `QueryWrapper` 来创建一个查询条件，然后通过 `orderMapper.selectList(wrapper)` 查询满足条件的订单列表，最后将结果返回。

insertOrder方法的作用是插入新的订单， @RequestBody 注解用于接收请求体中的 JSON 数据并将其转换为 `Order` 对象。

首先检查传入的订单对象是否为 null，如果不为 null，则使用 `orderMapper.insert(order) 将订单插入到数据库中，然后返回成功的响应消息。如果订单对象为 null，则返回插入失败的错误消息。

1. @RestController
2. @RequestMapping("/files")
3. **public** **class** FileController {
5. @Value("${server.port}")
6. **private** String port;
8. **private** **static** **final** String ip = "http://localhost";
10. /\*\*
11. \* 上传文件的接口
12. \* @param file 要上传的文件
13. \* @return 包含上传后文件访问路径的 Result 对象
14. \* @throws IOException 文件上传过程中的IO异常
15. \*/
16. @PostMapping("/upload")
17. **public** Result<?> upload(MultipartFile file) **throws** IOException {
18. // 获取原始文件名
19. String originalFilename = file.getOriginalFilename();
20. // 生成唯一标识符，用于文件名
21. String flag = IdUtil.fastSimpleUUID();
22. // 拼接文件在服务器端的保存路径
23. String rootFilePath = System.getProperty("user.dir") + "/src/main/resources/files/" + flag + "\_" + originalFilename;
24. // 将文件字节写入服务器
25. FileUtil.writeBytes(file.getBytes(), rootFilePath);
26. // 返回上传成功的结果，包括文件的访问路径
27. **return** Result.success(ip + ":" + port + "/files/" + flag);
28. }
30. /\*\*
31. \* 根据文件标识符获取文件
32. \* @param flag 文件标识符
33. \* @param response HTTP响应对象，用于输出文件内容
34. \*/
35. @GetMapping("/{flag}")
36. **public** **void** getFiles(@PathVariable String flag, HttpServletResponse response) {
37. OutputStream os;  // 新建一个输出流对象
38. String basePath = System.getProperty("user.dir") + "/src/main/resources/files/";  // 定义文件上传的根路径
39. List<String> fileNames = FileUtil.listFileNames(basePath);  // 获取所有的文件名称
40. // 找到与参数一致的文件名
41. String fileName = fileNames.stream().filter(name -> name.contains(flag)).findAny().orElse("");
42. **try** {
43. **if** (StrUtil.isNotEmpty(fileName)) {
44. // 设置响应头，指定文件名和内容类型
45. response.addHeader("Content-Disposition", "attachment;filename=" + URLEncoder.encode(fileName, "UTF-8"));
46. response.setContentType("application/octet-stream");
47. **byte**[] bytes = FileUtil.readBytes(basePath + fileName);  // 通过文件的路径读取文件字节流
48. os = response.getOutputStream();   // 通过输出流返回文件
49. os.write(bytes);
50. os.flush();
51. os.close();
52. }
53. } **catch** (Exception e) {
54. System.out.println("文件下载失败");
55. }
56. }
57. }

最后这是一个处理头像文件上传的控制它包含两个主要方法：upload和getFiles。

upload方法：该方法用于处理文件上传请求，客户端可以通过该接口上传头像照片。先获取上传的文件名 `originalFilename`。生成一个唯一的标识符flag，用于构建文件在服务器端的保存路径。再构建文件在服务器端的保存路径rootFilePath。使用 FileUtil.writeBytes将上传的文件内容写入服务器指定路径。返回一个包含上传后文件访问路径的Result对象，以便客户端可以通过该路径访问上传的头像照片。

getFiles方法：该方法用于根据文件标识符获取已上传的头像照片。Flag是文件的标识符，用于确定要获取哪个文件。HttpServletResponse response：HTTP 响应对象，用于输出文件内容。

先定义一个输出流对象 os。获取文件上传的根路径basePath。获取所有已上传文件的文件名列表fileNames。通过流操作筛选出与传入的flag匹配的文件名fileName。如果找到匹配的文件名，设置响应头，包括文件名和内容类型。通过文件路径读取文件字节流，并通过输出流将文件内容返回给客户端。

5.2.4 分页处理类和返回结果类

1. @Configuration
2. @MapperScan("com.example.springboot.mapper")
3. **public** **class** MybatisPlusConfig {
5. // 最新版
6. @Bean
7. **public** MybatisPlusInterceptor mybatisPlusInterceptor() {
8. MybatisPlusInterceptor interceptor = **new** MybatisPlusInterceptor();
9. interceptor.addInnerInterceptor(**new** PaginationInnerInterceptor(DbType.MYSQL));
10. **return** interceptor;
11. }
13. }

这是一个Spring Boot中的配置类，主要配置了 MyBatis Plus 的分页插件。@Configuration注解表示这是一个配置类，用于配置 Spring 应用上下文中的 bean。而@MapperScan("com.example.springboot.mapper")用于扫描指定包下的 MyBatis Mapper 接口，并将它们注册为 Spring 的 bean，以便 MyBatis Plus 可以自动管理这些 Mapper。@Bean注解用于定义 Spring Bean，这里定义了一个名为mybatisPlusInterceptor的 Bean。public MybatisPlusInterceptor mybatisPlusInterceptor()方法创建并配置了 MyBatis Plus 的分页拦截器MybatisPlusInterceptor,通过interceptor在拦截器中添加了分页插件 PaginationInnerInterceptor，并指定数据库类型为 MySQL。这个插件用于处理分页查询，它会拦截 MyBatis Plus 的查询请求，在查询 SQL 中自动添加分页查询的语句，以实现分页功能。这个类使得在使用 MyBatis Plus 进行数据库查询时可以方便地进行分页查询操作。

1. **public** **class** Result<T> {
2. // HTTP 状态码
3. **private** String code;
4. // 响应消息
5. **private** String msg;
6. // 传输的数据
7. **private** T data;
8. **public** Result() {
9. }
10. // 带参构造函数，用于初始化响应数据
11. **public** Result(T data) {
12. **this**.data = data;
13. }
14. // 获取 HTTP 状态码
15. **public** String getCode() {
16. **return** code;
17. }
18. // 设置 HTTP 状态码
19. **public** **void** setCode(String code) {
20. **this**.code = code;
21. }
22. // 获取响应消息
23. **public** String getMsg() {
24. **return** msg;
25. }
26. // 设置响应消息
27. **public** **void** setMsg(String msg) {
28. **this**.msg = msg;
29. }
30. // 获取响应数据
31. **public** T getData() {
32. **return** data;
33. }
34. // 设置响应数据
35. **public** **void** setData(T data) {
36. **this**.data = data;
37. }
38. // 静态方法，用于创建成功的响应对象（无数据）
39. **public** **static** Result success() {
40. Result result = **new** Result<>();
41. result.setCode("0");   // 成功状态码通常为 "0"
42. result.setMsg("成功"); // 设置默认成功消息
43. **return** result;
44. }
45. // 静态方法，用于创建成功的响应对象（带数据）
46. **public** **static** <T> Result<T> success(T data) {
47. Result<T> result = **new** Result<>(data);
48. result.setCode("0");   // 成功状态码通常为 "0"
49. result.setMsg("成功"); // 设置默认成功消息
50. **return** result;
51. }
52. // 静态方法，用于创建失败的响应对象
53. **public** **static** Result error(String code, String msg) {
54. Result result = **new** Result();
55. result.setCode(code); // 设置自定义状态码
56. result.setMsg(msg);   // 设置自定义错误消息
57. **return** result;
58. }
59. }

以上是 结果返回类，用于结果返回统一化，有数据传数据，无数据默认构造，以0作为成功码，含有返回的信息，用于controller中处理请求的返回值。

5.2.5 过滤器

1. @Component
2. @WebFilter(urlPatterns = "/\*", filterName = "myFilter")
3. **public** **class** MyFilter **implements** Filter {
5. // 使用SLF4J记录日志
6. **private** Logger logger = LoggerFactory.getLogger(MyFilter.**class**);
8. @Override
9. **public** **void** init(FilterConfig filterConfig) **throws** ServletException {
10. // 初始化方法
11. }
13. @Override
14. **public** **void** doFilter(ServletRequest sRequest, ServletResponse sResponse, FilterChain chain)
15. **throws** IOException, ServletException {
17. // 将ServletResponse强制类型转换为HttpServletResponse以处理HTTP响应头
18. HttpServletResponse response = (HttpServletResponse) sResponse;
20. // 设置CORS相关响应头，解决跨域问题
21. response.setHeader("Access-Control-Allow-Origin", "\*"); // 允许任何来源访问
22. response.setHeader("Access-Control-Allow-Credentials", "true"); // 允许发送凭据（如Cookie）
23. response.setHeader("Access-Control-Allow-Methods", "POST, PUT, GET, OPTIONS, DELETE"); // 允许的HTTP方法
24. response.setHeader("Access-Control-Max-Age", "3600"); // 预检请求（OPTIONS请求）的缓存时间
25. response.setHeader("Access-Control-Allow-Headers",
26. "Origin, X-Requested-With, Content-Type, Accept, client\_id, uuid, Authorization"); // 允许的请求头字段
27. response.setHeader("Cache-Control", "no-cache, no-store, must-revalidate"); // 不缓存响应
28. response.setHeader("Pragma", "no-cache"); // 兼容HTTP 1.0
29. response.setHeader("Expires", "0"); // 兼容HTTP 1.0
31. // 继续处理请求，将请求和响应传递给下一个过滤器或Servlet
32. chain.doFilter(sRequest, sResponse);
33. }
35. @Override
36. **public** **void** destroy() {
37. // 销毁方法
38. }
39. }

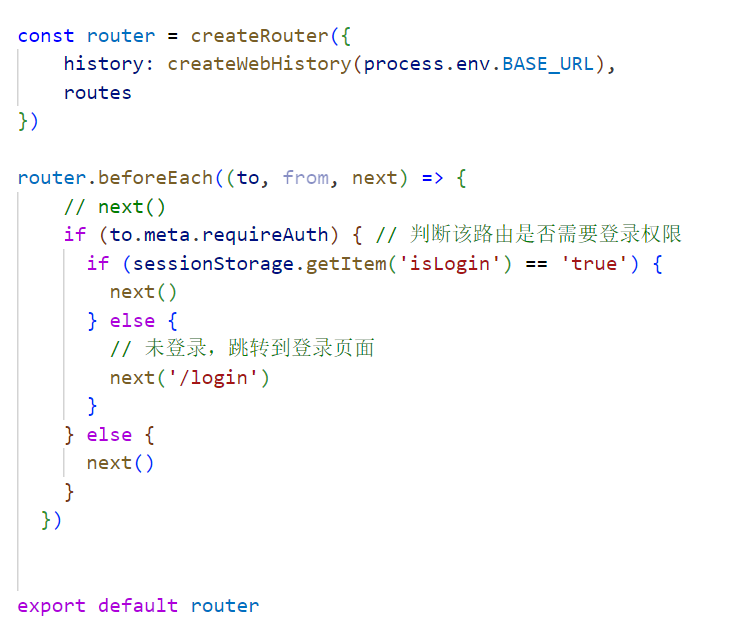
过滤器代码，拦截部分请求。

5.3 前端

前端主要是界面的布局，在根组件中放置个个子组件，以及向后端传数据，从后端请求数据和不同组件之间传数据

5.3.1 路由配置

要在Vue的各个组件下跳转并自由传输数据，路由是最好的选择，以下是前端代码对路由的配置



对组件的路由配置其路径和名称，以及路由守卫检测现在是否处于登录状态

1. 总结

心得总结：

1. 技术栈选择与整合： 选择了 Spring Boot 和 Vue3 作为后端和前端框架，这两者的整合相对容易，但需要熟悉两者的基本用法和配置。使用 MyBatis-Plus 简化了数据库操作，提高了开发效率。
2. 前后端分离： 采用了前后端分离的架构，前后端通过 RESTful API 进行数据交互，提高了系统的灵活性和可维护性。
3. 数据库设计： 数据库设计需要仔细考虑，包括表的关系、字段的类型和索引的使用。对数据库进行合理的规划可以提高系统性能。
4. 异常处理： 在后端开发中，要充分考虑异常处理，合理定义异常类和统一异常处理，提高系统的容错性。
5. 前端交互体验： 前端界面应具备良好的用户交互体验，包括友好的界面、用户提示和表单验证等功能。

遇到的 Bug 及解决方法：

* 跨域问题： 前后端分离时，可能遇到跨域问题。解决方法是在后端配置跨域允许访问的域名或使用代理服务器。
* 数据传输格式问题： 前后端数据传输格式需要保持一致，可以使用 JSON 数据格式，确保数据正确解析。
* 前端路由问题： 前端路由配置时，需要注意路由的顺序，避免出现匹配问题。

不足和待完善的地方：

1. 安全性加强： 可以进一步加强系统的安全性，包括用户认证、授权和数据加密等方面的处理。
2. 性能优化： 随着系统的使用，可能会面临性能瓶颈，需要进行性能优化，包括数据库查询优化和缓存机制的引入。
3. 前端页面优化： 前端页面可以进一步优化，包括响应式设计、页面加载速度和用户体验的提升。
4. 功能扩展： 系统还可以扩展更多功能，如电影推荐、在线支付等，以增加系统的吸引力和实用性。
5. 测试和文档： 缺乏充分的测试和文档可能导致系统的可维护性下降，需要加强测试覆盖率和编写详细的文档。

综上所述，这个电影售票系统项目，已经完成了基本的功能，但仍需要不断改进和完善，以适应不断变化的需求和提高系统的质量。系统的安全性、性能、用户体验和功能扩展都是可改进的方面。