

基于 Springboot+Vue 的电影论坛网站

梁宝丹(2112190536) 岑奕侃(2112190512) 王馨远(2112190510)

一、项目介绍 [王馨远]

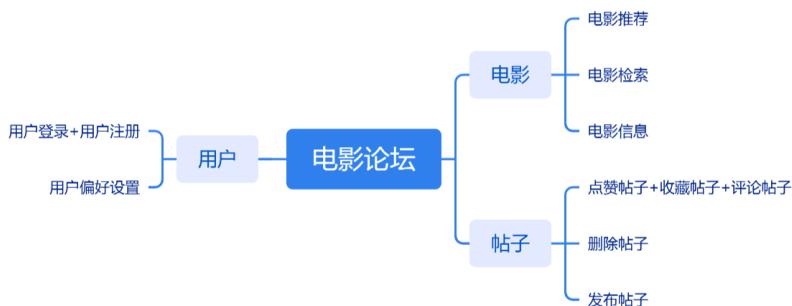
背景

创建一个电影论坛网站的项目，其核心价值在于提供一个平台，让电影爱好者、影评人、行业专家和普通观众能够聚集在一起，分享、讨论和探索电影世界的各个方面。以下是它能解决的问题和实现的功能：

- 信息聚合与分享：**电影论坛可以作为一个中心化的信息来源，整合来自全球的电影资讯、评论、预告片、幕后花絮等内容。这解决了用户在多个平台间寻找电影信息的不便，为他们提供了一个全面且易于访问的资源库。
- 社区互动与交流：**通过建立一个社区，项目鼓励用户之间的互动和讨论。这不仅包括对特定电影的评论和评分，还包括对电影制作技术、电影史、电影理论等更深层次话题的探讨。
- 个性化推荐：**利用用户行为分析和算法，电影论坛可以提供个性化的电影推荐服务。根据用户的偏好，系统能推荐可能感兴趣的电影，帮助用户发现新作品，同时也增加了网站的粘性。

综上所述，创建一个电影论坛网站旨在构建一个充满活力的在线社区，不仅满足用户获取和分享电影信息的需求，还促进了电影文化的繁荣和电影艺术的深入理解。

项目需求分析



用户功能：

用户登录和注册：用户可以通过输入用户名、密码等信息来创建账户或登录已有的账户。
用户偏好设置：用户可以根据自己的喜好设置喜欢的电影类型或其他个性化选项。

电影功能：

电影推荐：系统会根据用户的观看历史、评分记录等数据为用户推荐相关的电影。

电影检索：用户可以通过筛选来查找特定的电影，并查看其相关信息（如简介、演员表等）。

电影信息：展示每部电影的基本信息，包括海报、导演、主演、剧情简介等。

帖子功能：

发布帖子：允许用户发布关于某部电影的评论、感想或者讨论话题。

点赞/收藏/评论帖子：其他用户可以对某个帖子进行点赞表示赞同，收藏以便日后查看，或者发表自己的看法与原作者互动交流。

计划和分工

开发计划：

1. 需求分析与规划（3月1日-3月10日）

目标确定：明确网站的主要功能，如用户注册/登录、电影信息展示、评论系统、搜索功能等。

2. 系统设计（3月11日-3月20日）

界面设计：使用墨刀设计 UI 界面原型。

架构设计：

前端：选择 Vue 框架，考虑使用 Vuex 进行状态管理，Element UI 作为 UI 库。

后端：选择 Spring Boot，设计 RESTful API，考虑使用 MyBatis 作为 ORM 工具。

数据库设计：设计 ER 图，规划数据表结构，考虑使用 MySQL 或 PostgreSQL。

3. 技术选型与环境搭建（3月21日-3月31日）

开发环境：设置开发环境，安装必要的软件和工具。

技术栈确认：最终确认前后端技术栈。

基础代码框架：搭建 Spring Boot 和 Vue 的基础项目结构。

4. 开发实施（4月1日-5月1日）

后端开发：

实现用户认证模块。

开电影信息展示 API。

实现评论系统 API。

前端开发：

构建用户界面，实现响应式布局。

集成 API 调用，实现动态内容展示。

实现用户交互功能，如评论提交、搜索等。

5. 单元测试与集成测试（5月10日-5月20日）

单元测试：对各个模块进行单元测试，确保代码质量。

接口测试：使用 Postman 或 Apifox 测试 API 的正确性和稳定性。

集成测试：确保前后端能够顺利通信，功能完整。

6. 接口对接与优化（5月21日-5月31日）

数据一致性检查：确保前后端数据格式一致。

性能优化：根据测试结果优化代码和数据库查询。

安全加固：增强系统安全性，防止 SQL 注入、XSS 攻击等。

7. Docker 部署与上线准备（6月1日-6月10日）

Docker 镜像构建：将应用打包为 Docker 镜像。

容器化部署：在服务器上部署 Docker 容器，配置 Nginx 作为反向代理。

监控与日志：设置监控系统，记录应用运行日志。

分工：

梁宝丹：后端+测试+部署

岑奕侃：前端+后端

王馨远：前端+后端+数据库

二、界面原型设计 [梁宝丹，岑奕侃，王馨远]

电影功能：

首页：实现电影推荐功能，包括院线热映和用户偏好电影推荐。

搜索界面：实现搜索功能，包括输入关键字搜索、分类筛选 2 种方式。

电影信息界面：展示一部电影的基本信息，包括海报、导演、主演、剧情简介等。

帖子功能：

帖子界面：可以查看帖子的图文信息，可以对帖子进行点赞、收藏、评论。

上传帖子界面：可以编写一个帖子的标题、电影评分、正文、图片。

用户功能：

用户界面：展示用户头像、用户名称、用户偏好、用户帖子。

登录注册界面：用户的登录注册。

设置界面：用户偏好设置、头像名称上传修改、设置密码等功能。

具体跳转逻辑如下：





首页

点击上方导航栏的个人主页可跳转到个人页面，如果未登录，将显示登录页面

个人主页 (已登录)

查看个人信息，已发布的帖子，已收藏的帖子，浏览记录，设置用户感兴趣的电影类型

登录页面 (未登录)

登录账号，如未拥有账号，可跳转至注册页面

注册页面

注册账号



个人主页

点击关注电影类型后的“+”按钮，可添加关注电影类型，点击用户名右侧按钮可进入设置页面



设置

可以修改头像，昵称，密码等个人数据，也可清空浏览记录



添加关注电影的标签



电影详情

点击底部的“我要发贴”跳转到发帖上传页面



上传页面

在这里用户可以给电影评分，写帖子，支持输入文字和上传本地图片

主要界面的设计分析和可行性

1. 用户个人信息页面：

设计理念：

界面一致性，清晰的信息层次结构，用户对帖子的可操作性

优缺点：

优点：

一致的主题和颜色方案增强了界面设计的整体性。

清晰的信息层次结构使用户能够快速了解页面内容。

快捷跳转至用户帖子、收藏和浏览记录方便用户管理自己的活动。

缺点：

用户个人信息可填写的内容较少，不利于对用户信息的收集和分析。

实现难度：

在 Vue 中实现这样一个页面的难度适中。Vue 提供了丰富的组件库和数据绑定机制，可以帮助开发者快速构建出界面。但要实现的功能不少，工作量较大。

需要注意的问题包括：

如何处理用户数据的加载与渲染。

如何处理用户喜好标签的设置与更新。

如何实现页面内各部分的交互逻辑，例如点击链接跳转到相应的帖子、收藏或浏览记录页面。

2. 电影详情页面：

设计理念：

界面一致性，易于操作，互动性强

优缺点：

优点：

设计风格简约，用户体验好，容易上手。

提供用户评价和评论功能，鼓励用户参与讨论，增强社区氛围。

社区讨论和评分可以直观展示，有利于推广电影和吸引新用户。

缺点：

如果没有良好的后台算法支持，热门评论可能会被刷屏或者出现不相关的内容。

对于想要直接查找某个帖子的用户来说，缺乏搜索帖子的手段。

实现难度：

实现这样一个界面相对简单。

需要注意的问题包括：

如何处理大量帖子数据的加载与渲染速度，以及如何提高性能。

如何跳转页面时传递相应的电影帖子数据，实现相关传参功能。

3. 首页页面：

设计理念：

界面一致性，信息密度适中，易于导航

优缺点：

优点：

信息密度适中可以满足用户快速浏览的需求，无需频繁滚动屏幕。

彩色电影海报，吸引了用户的注意力。

易于导航的设计，使用户能够迅速找到感兴趣的内容。

缺点：

电影信息展示较少，无法根据用户关注的重点信息进行展示。

未提供电影详情或预告片链接，用户需进一步探索才能了解更多。

实现难度：

在 Vue 中实现这样一个界面相对简单。Vue 提供了丰富的组件库和数据绑定机制，可以帮助开发者快速构建出类似的界面。

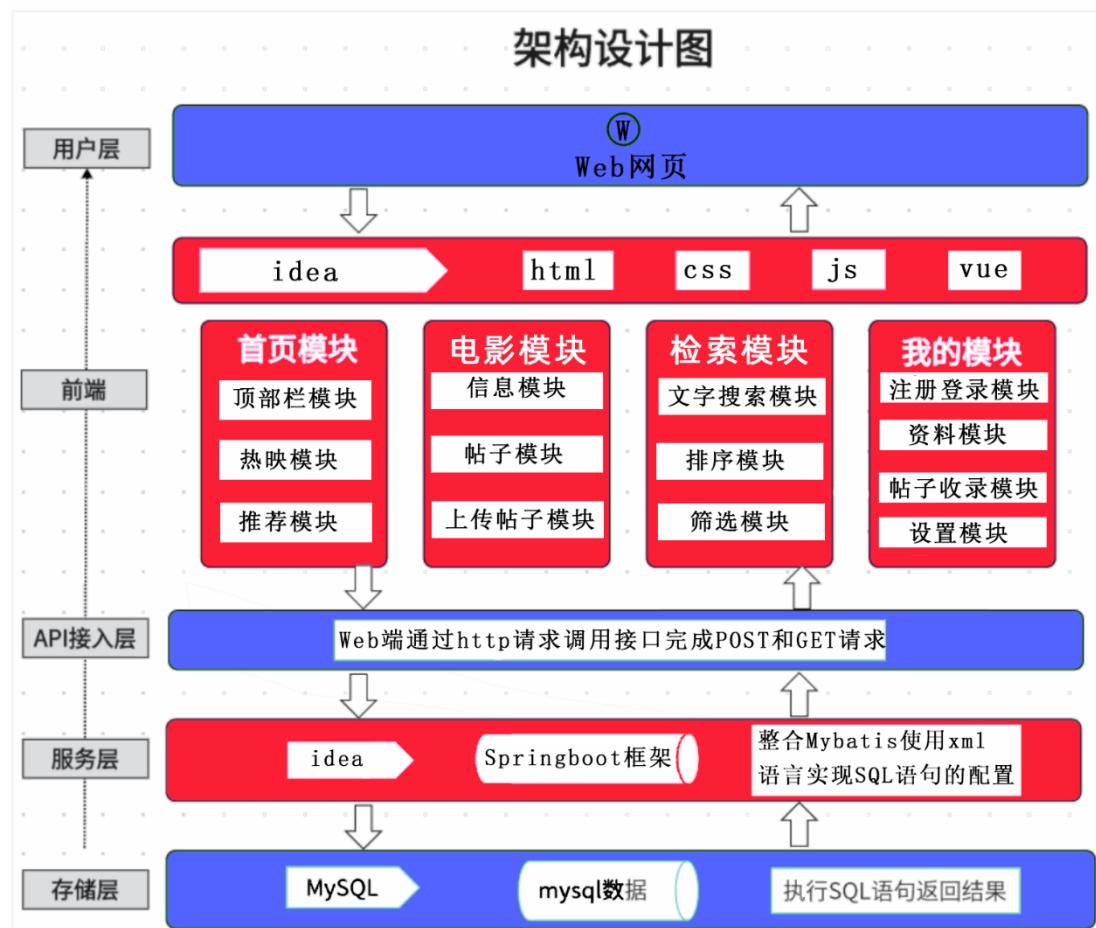
需要注意的问题包括：

如何处理热映电影展示的横向拉动。

如何根据用户喜好以获取实时的电影数据，并在首页展示。

如何实现导航栏的交互逻辑，包括点击事件和状态更新。

三、系统架构设计 [王馨远]



上图是我们的电影论坛 Web 开发的架构设计图，主要分为前端、API 接入层、服务层和存储层四个部分。

- 1) 前端：用户通过浏览器访问 Web 网页，页面由 HTML、CSS、JavaScript 和 Vue.js 组成。其中 HTML 负责内容展示，CSS 负责样式控制，JavaScript 提供交互功能，而 Vue.js 则是一种流行的前端框架，用于构建响应式用户界面。
- 2) API 接入层：Web 端通过 HTTP 请求调用接口完成 POST 和 GET 等请求。这个层次的作用

是将前端与后端进行解耦，使得前端可以独立开发和部署。

- 3) 服务层：使用 SpringBoot 框架整合 MyBatis，通过 XML 语言实现 SQL 语句配置。SpringBoot 简化了 Java Web 应用的搭建和配置过程，提供了自动配置的功能；MyBatis 是一个轻量级的持久化框架，它允许开发者直接编写 SQL 语句，提高了查询效率。
- 4) 存储层：使用 MySQL 作为数据存储。MySQL 是一款开源的关系型数据库管理系统，具有良好的性能和稳定性。

在设计过程中，我们采用了 MVC (Model-View-Controller) 模式，将业务逻辑、数据处理和用户界面分离，增强了代码的可维护性和复用性。同时，使用 RESTful API 规范设计接口，使得 API 更加清晰易懂，方便前后端协作。

对于前端模块的设计，首页模块包含了顶部栏、热映电影和推荐电影三个子模块；电影模块有信息、帖子两个子模块；检索模块包括文字搜索、排序和筛选三个子模块；我的模块则涵盖了注册登录、资料、帖子收藏和设置四个子模块。这些模块按照功能进行了划分，便于开发和维护。

在实现方案上，前端使用了 Vue.js 框架，结合 Element UI 库快速构建 UI 界面；后端使用 Spring Boot 框架，集成 MyBatis 作为 ORM 工具，使用 MySQL 作为数据存储。这样的组合能够提供一个高效、稳定的应用程序环境。

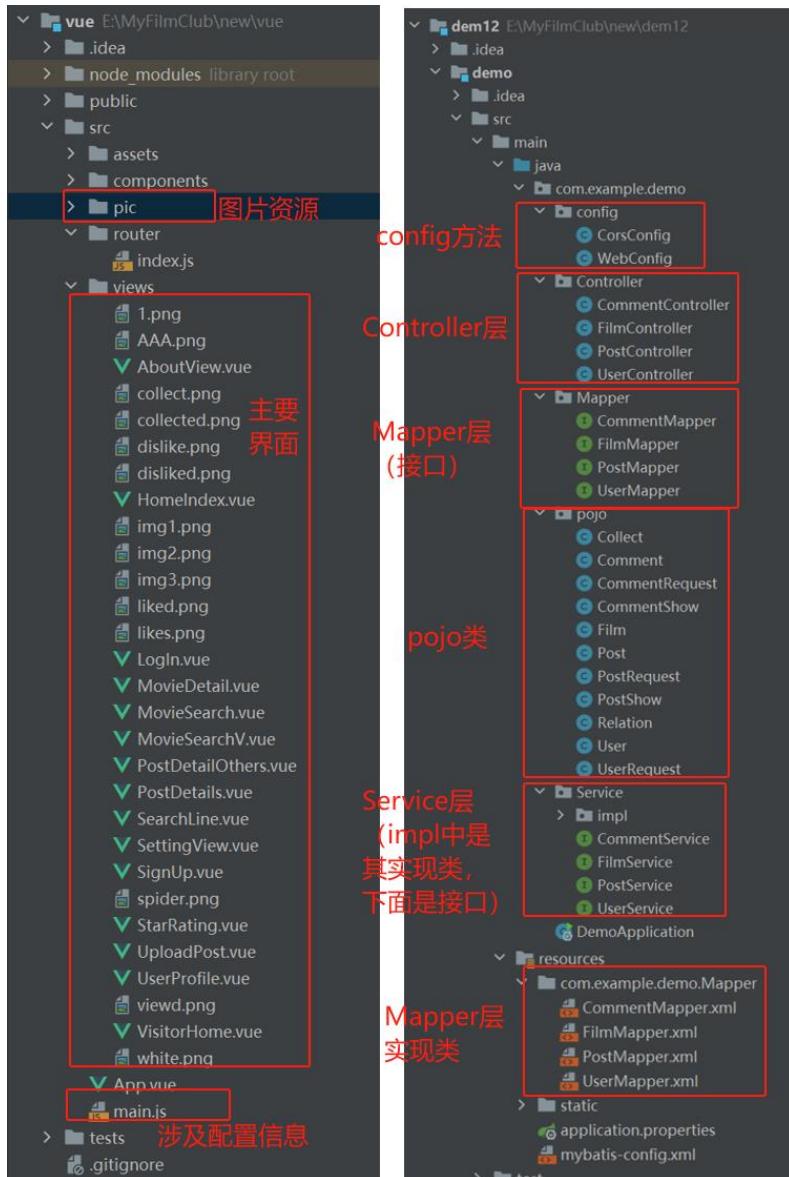
优点：

分层设计使系统具有更好的扩展性和灵活性；
使用成熟的框架和技术降低了开发难度和风险；
RESTful API 设计提高了前后端的协作效率。

缺点：

分层设计可能会增加系统的复杂度，需要更多的开发资源；
需要熟悉多种技术和框架才能进行开发；
对于小型项目来说，可能显得过于复杂。

具体模块如下：



四、API 设计 [岑奕侃]

这部分将所有 API 分为四大类，逐步进行介绍 API 的设计理念，使用、测试方法。

1. 用户账号操作模块

- ▼ **POST** 用户登录 (3)
 - ✗ 用户名或密码错误
 - ✓ 成功
 - ✗ 请求超时

- POST** 用户注册

- PUT** 修改用户信息

这三个关于用户账号信息的 API 接口，需要绝对的安全性保证，确保数据传输安全，采

用 HTTPS 协议，对密码进行加密处理。对于用户名只能唯一，不可重复。遵循 RESTful 原则，使用 JSON 格式返回响应，清晰定义错误码和消息。

1. 用户注册 (POST /user/signup)

设计理念：

使用 HTTP 的 POST 方法，因为这是一个创建资源的操作。

路径设计为 /user/signup，明确表示这是用于用户注册的功能。

请求体通常包含用户的邮箱、用户名、密码等信息，需要在服务器端进行验证和处理。

使用方法：

在 Apifox 中，创建一个新的接口，设置方法为 POST，路径为 /users/register。

定义请求体 (Body)，比如使用 JSON 格式，包含字段 username, password。

设定响应模型，预期服务器会返回注册状态和可能的错误信息。

测试方法：

使用 Apifox 的调试功能，填写注册所需的信息，发送请求。

验证服务器是否返回预期的响应代码和成功信息。

测试边缘情况，如已存在的用户名等。

2. 用户登录 (POST /users/login)

设计理念：

采用 POST 方法，路径为 /user/login，表明此操作用于用户认证。

请求体通常包含用户的认证信息，如用户名和密码。

使用方法：

在 Apifox 中创建接口，方法为 POST，路径为 /user/login。

定义请求体，username 和 password 字段。

响应中定义应包含认证令牌和其他可能的用户信息或状态码。

测试方法：

使用正确的凭据测试登录功能，确认能否正确收到令牌。

测试无效凭据的情况，确保服务器返回错误信息而非令牌。

测试令牌的有效性，可以在后续请求中使用令牌进行身份验证。

3. 用户修改密码 (PUT /user)

设计理念：

使用 PUT 方法，因为它代表更新现有资源。

请求体包括账号和密码，以验证用户身份并执行修改。

使用方法：

创建接口，方法为 PUT，路径为 /user。

请求体中定义 username 和 password。

设定响应模型，以反映修改操作的结果状态。

测试方法：

使用有效用户名尝试修改密码，验证操作成功。

验证修改后的密码在登录时是否生效。

2. 电影的浏览和筛选搜索

GET 展示主页电影

GET 查询电影

GET 根据标签搜索电影

GET 展示当前电影的帖子

1. 展示主页电影 (GET /filmall)

设计理念：

使用 GET 方法，因为这是一次获取资源的操作。

路径设计为/filmall，直接表达了请求的目的。

使用方法：

在 Apifox 中创建一个接口，方法设为 GET，路径设为上述定义。

设定响应模型，定义预期返回的电影列表结构，包括标题、海报、主演、导演等信息。

测试方法：

发送请求，检查返回的电影列表是否符合预期，包括数量、排序及信息完整性。

2. 查询电影 (GET /film/{fid})

设计理念：

通过 GET 方法和包含电影 ID 的路径参数来获取单一电影详情。

使用方法：

创建接口，路径设为/film/{fid}，并标记 fid 为路径参数。

定义响应模型，详细列出电影的所有属性，如导演、演员、评分等。

测试方法：

使用已知的电影 ID 测试，验证返回的电影详情是否准确无误。

测试不存在的 ID，确认 API 是否返回合理的错误信息，如 404 Not Found。

3. 根据标签搜索电影 (GET /film)

设计理念：

通过 GET 方法，查询电影，通过前端的展示函数来展示特定的电影。

使用方法：

设计接口路径为/film。

响应模型应包括所有匹配该标签的电影列表。

测试方法：

测试多种不同的标签，确保返回的电影列表都正确关联了该标签。

测试无结果情况，确认 API 正确处理空集返回。

4. 展示当前电影的帖子 (GET /film_post/{fid})

设计理念：

通过电影 ID 获取与之相关的讨论帖子，使用 GET 方法。

使用方法：

接口路径定义为/film_post/{fid}，其中 fid 为路径参数。

响应模型定义帖子列表，包括发帖人 uid，电影评分，帖子标题和内容。

测试方法：

对一部热门电影发送请求，验证帖子列表的返回是否完整、准确。

测试无帖子情况，确保 API 返回空列表而不是错误。

3. 帖子的相关操作

GET 展示帖子内容

POST 发布帖子

POST 添加浏览记录

POST 点赞

POST 收藏

GET 展示当前帖子的评论

POST 发布评论

1. 发布帖子 (POST /post)

设计理念：

使用 POST 方法，因为这是一个创建新资源的操作。

设计简洁的路径/post，直接反映创建帖子的行为。

使用方法：

在 Apifox 中创建接口，设置方法为 POST，路径为/post。

定义请求体，通常包括标题、内容、作者 uid。

响应模型应定义创建成功的帖子的基本信息，如帖子 pid 等。

测试方法：

发布一个测试帖子，验证帖子是否成功创建，并检查返回的帖子 pid、标题、内容。

测试边界条件，如过长的标题或内容是否被正确处理。

2. 添加浏览记录 (POST /addHistoryPost)

设计理念：

使用 POST 方法来记录一次浏览行为，通过路径参数指定帖子 ID。

使用方法：

接口路径设置为/addHistoryPost。

请求体需用户 uid 和帖子 pid。

测试方法:

对某一帖子发送浏览请求，可以通过查看后端数据库 prelationinfo 表查看是否有对应的关系。

测试同一用户重复浏览同一帖子的处理逻辑。

3. 点赞 (POST /like)

设计理念:

同样使用 POST 方法，路径中包含帖子 pid。

使用方法:

接口路径为 /like，请求体包含用户 uid，帖子 pid。

响应可以是简单的成功状态或更新的点赞数。

测试方法:

发送点赞请求，验证帖子的点赞数是否正确增加。

4. 收藏 (POST /collect)

设计理念:

通过 POST 方法，路径结合用户 uid 和帖子 pid，用户收藏特定帖子。

使用方法:

接口路径为 /collect。

通常无需复杂请求体，除非需要额外信息。

测试方法:

收藏一个帖子，然后检查用户收藏列表是否包含该帖子。

测试重复收藏的逻辑处理。

5. 展示当前帖子的评论 (GET /post_comments/{pid})

设计理念:

使用 GET 方法获取资源，路径清晰指出请求的是帖子的评论。

使用方法:

接口路径为 /post_comments/{pid}。

可以有查询参数来控制返回的评论数量或排序方式。

测试方法:

获取指定帖子的评论，验证返回的评论列表是否正确，包括排序和数量限制。

测试无评论的帖子，确认 API 正确处理。

6. 展示当前帖子的内容 (GET /post_detail/{pid})

设计理念:

通过 GET 方法直接根据帖子 pid 获取帖子内容。

使用方法:

接口路径为 /post_detail/{pid}。

无特殊请求体，直接根据 pid 获取详细内容。

测试方法:

获取单个帖子的全部内容，验证信息的完整性。

测试无效或不存在的帖子 pid，返回 404 错误。

4. 用户个性化模块

GET 展示用户标签

POST 修改用户标签

GET 展示浏览记录

GET 展示发布内容

GET 展示收藏内容

POST 清除浏览记录

1. 展示用户标签 (GET /tags/{uid})

设计理念:

使用 GET 方法获取资源，路径明确指出是获取指定用户的标签。

使用方法:

接口路径为/tags/{uid}，其中 {uid} 为路径参数，代表要查询的用户 id。

测试方法:

请求特定用户的标签，验证返回的数据结构和内容是否符合预期。

测试排列的有效性。

2. 修改用户标签 (POST /updateTags)

设计理念:

使用 POST 方法更新整个资源，适合替换用户的所有标签。

使用方法:

接口路径为/updateTags，请求体包含新的标签列表。

测试方法:

更新用户的标签并验证更改是否生效。

3. 展示浏览记录 (GET /findHistory/{uid})

设计理念:

使用 GET 方法获取指定用户的浏览历史记录。

使用方法:

接口路径为/findHistory/{uid} /

测试方法:

请求用户的浏览记录，验证返回的数据格式和内容。

测试无浏览记录的用户情况。

4. 展示发布内容 (GET /findPost/{uid})

设计理念：

GET 方法获取指定用户发布的所有内容。

使用方法：

接口路径为 /findPost/{uid}。

测试方法：

获取用户的发布内容，验证数据完整性。

5. 展示收藏内容 (GET /findCollectedPost/{uid})

设计理念：

使用 GET 方法获取用户收藏的所有内容。

使用方法：

接口路径为 /findCollectedPost/{uid}.

测试方法：

请求用户的收藏内容，检查返回数据的正确性。

测试无收藏内容的用户情况。

6. 清除浏览记录 (DELETE /deleteHistory/{uid})

设计理念：

使用 DELETE 方法清除指定用户的浏览历史。

使用方法：

接口路径为 /deleteHistory/{uid}，无需请求体。

测试方法：

发送清除请求后，验证用户的浏览记录是否被清空。

测试权限控制，确保只有用户本人或具有相应权限的账户能执行此操作。

五、数据库设计 [王馨远]

持久化数据

用户信息：包括用户名、密码、喜好标签等。

电影资料：包括电影名称、导演、演员、上映日期、评分、简介、海报图片链接等。

评论与评分：用户对电影帖子的评论、电影评分、点赞数等。

论坛帖子与回复：用户在论坛上发表的主题帖子和回复内容。

用户行为记录：如登录日志、浏览历史、搜索记录等，用于分析用户行为和优化推荐算法。

这些数据通常需要长期保存，因此适合使用关系型数据库进行存储，如 MySQL，因为其支持 ACID 特性，能够保证数据的一致性和事务完整性。此外，SQL 数据库的查询语言 SQL 易于理解和使用，适合复杂的查询需求，如统计分析和个性化推荐算法的实现。

缓存数据

热门电影列表: 根据用户浏览量和评分实时更新的电影列表。

用户推荐内容: 基于用户历史行为生成的个性化推荐电影和帖子。

用户最近浏览的电影: 用于快速加载用户个人主页上的浏览历史。

这些数据适合使用缓存技术，如 Redis，以减少对后端数据库的请求压力，提高响应速度。

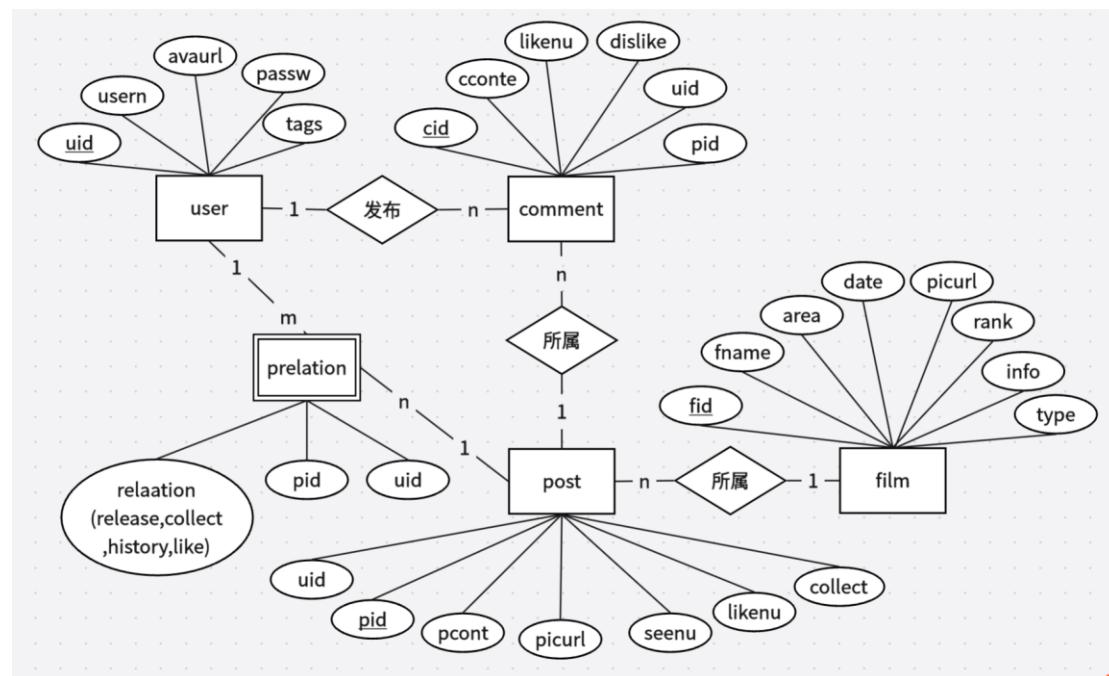
缓存数据库通常提供高速的读写性能，非常适合存储热点数据和频繁访问的内容，同时支持数据的过期策略，可以自动清理不再需要的缓存项。

数据库方案

主数据库: 选择 MySQL 作为主数据库，因为它在处理大量并发连接方面表现优异，高并发性能，这对于电影论坛这类用户活跃度高的网站尤为重要。MySQL 与多种编程语言和操作系统都有良好的集成，便于开发和部署。

而且其拥有庞大的用户群体和丰富的文档资源，遇到问题时更容易找到解决方案。

缓存数据库: 使用 Redis 作为缓存层，它可以存储热点数据和临时会话信息，提供快速的数据访问能力，并且支持数据持久化，防止服务重启时丢失缓存数据。



Commentinfo(评论表)

存放评论相关数据。

该表有以下字段:

cid: 这条评论的 id，主键，int 类型。

ccontent: 这条评论的内容，mediumtext 类型，适合存放比较长的文本。

likenum: 点赞数，int 类型。

dislikenum: 点踩数，int 类型。

uid: 发布这条评论的用户的 id，外键，int 类型。

pid: 发布在哪个帖子上，帖子的 id，外键，int 类型。

本机

film_forum

表

commentinfo

filminfo

postinfo

prelationinfo

userinfo

视图

函数

查询

备份

film6.6

information schema

对象

commentinfo... prelationinfo... filminfo @fil... postinfo @fil... userinfo @fil...

保存 添加字段 插入字段 删除字段 主键 上移 下移

字段 索引 外键 触发器 选项 注释 SQL 预览

名	类型	长度	小数点	不是 null	虚拟	键	注释
cid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	主键 评论内容
ccontent	mediumtext			<input type="checkbox"/>	<input type="checkbox"/>		
likenum	int	11		<input type="checkbox"/>	<input type="checkbox"/>		
dislikenum	int	11		<input type="checkbox"/>	<input type="checkbox"/>		
uid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		
pid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		

Filminfo(电影表)

存放电影相关数据。

该表有以下字段:

Fid:电影 id, 主键, int 类型。

Fname:电影名称, varchar 类型。

Rank:电影评分, 设为 1 到 5 的整数, 对应 5 种打星图片。

Area:制片地区, varchar 类型。

Date:上映日期, datetime 类型。

Picurl:电影海报 url, varchar 类型, 存放图片地址。

Language: 电影语种, varchar 类型。

Flength: 电影片长, int 类型。

Director: 电影导演, varchar 类型。

Scriptwriter: 电影编剧, varchar 类型。

Actor: 电影主演, varchar 类型。

Info:电影简介, mediumtext 类型。

Type:电影类型, varchar 类型。

本机

film_forum

表

commentinfo

filminfo

postinfo

prelationinfo

userinfo

视图

函数

查询

备份

film6.6

information_schema

mysql

performance_schema

serverside

sys

wulianwang

对象

commentinfo... prelationinfo... filminfo @fil... postinfo @fil... userinfo @fil...

保存 添加字段 插入字段 删除字段 主键 上移 下移

字段 索引 外键 触发器 选项 注释 SQL 预览

名	类型	长度	小数点	不是 null	虚拟	键	注释
fid	int	8		<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	8位数字
fname	varchar	255		<input checked="" type="checkbox"/>	<input type="checkbox"/>		
rank	int	1		<input type="checkbox"/>	<input type="checkbox"/>		0,1,2,3,4,5
area	varchar	255		<input checked="" type="checkbox"/>	<input type="checkbox"/>		制片地区
date	datetime			<input checked="" type="checkbox"/>	<input type="checkbox"/>		上映日期
picurl	varchar	255		<input checked="" type="checkbox"/>	<input type="checkbox"/>		电影海报
language	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>		语言
flength	int	11		<input type="checkbox"/>	<input type="checkbox"/>		片长
director	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>		导演
scriptwriter	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>		编剧
actor	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>		主演
info	mediumtext			<input checked="" type="checkbox"/>	<input type="checkbox"/>		电影简介
type	varchar	255		<input checked="" type="checkbox"/>	<input type="checkbox"/>		多个类型间用分隔符隔开

Postinfo(帖子表)

存放帖子相关数据。

该表有以下字段:

Pid:帖子 id, 主键, int 类型。

Uid:帖子作者 id, int 类型。

Pcontent:帖子内容, mediumtext 类型, 存放正文内容。

Picurl:帖子插图 url, text 类型, 存放用户上传的图片转码后的字节数据。

Ptitle:帖子标题, varchar 类型。

Seenum:浏览数, int 类型。

Likenum:点赞数, int 类型。

Collectnum:收藏数, int 类型。

Fid:对应的电影 id, int 类型。

Value:发帖人对电影的评分, int 类型。

The screenshot shows the MySQL Workbench interface with the 'film_forum' database selected. The '表' (Tables) section lists 'commentinfo', 'filminfo', 'postinfo', 'prelationinfo', and 'userinfo'. The 'prelationinfo' table is currently selected, shown in the main pane. The table structure is defined as follows:

名	类型	长度	小数点	不是 null	虚拟	键	注释
pid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		1 主键
uid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		
pcontent	mediumtext			<input checked="" type="checkbox"/>	<input type="checkbox"/>		帖子内容
picurl	text			<input type="checkbox"/>	<input type="checkbox"/>		图片url
ptitle	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>		帖子标题
seenum	int	10		<input type="checkbox"/>	<input type="checkbox"/>		默认为0
likenum	int	10		<input type="checkbox"/>	<input type="checkbox"/>		默认为0
collectnum	int	10		<input type="checkbox"/>	<input type="checkbox"/>		默认为0
fid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		
value	int	11		<input type="checkbox"/>	<input type="checkbox"/>		

Prelationinfo(帖子关系表)

Prelationinfo 是依赖于用户和帖子而存在的弱实体, 用于存放用户对帖子的相关操作记录。

该表有以下字段:

Pid:帖子 id, int 类型。

Relation:帖子与用户对应关系 (包括 r, c, h, l, 对应发布、收藏、浏览、点赞 4 种关系)

Uid:用户 id, int 类型。

The screenshot shows the MySQL Workbench interface with the 'film_forum' database selected. The '表' (Tables) section lists 'commentinfo', 'filminfo', 'postinfo', 'prelationinfo', and 'userinfo'. The 'userinfo' table is currently selected, shown in the main pane. The table structure is defined as follows:

名	类型	长度	小数点	不是 null	虚拟	键	注释
pid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		
relation	varchar	1		<input checked="" type="checkbox"/>	<input type="checkbox"/>		关系r,c,h,l (release,collect,history,like)
uid	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		

Userinfo(用户表)

存放用户相关数据。

该表有以下字段:

Uid:用户 id, 主键, int 类型。

Username:用户名称, varchar 类型。

Password:用户密码, varchar 类型, 6 位。

Avaurl:用户头像, varchar 类型。

Tags:用户喜好标签, varchar 类型。

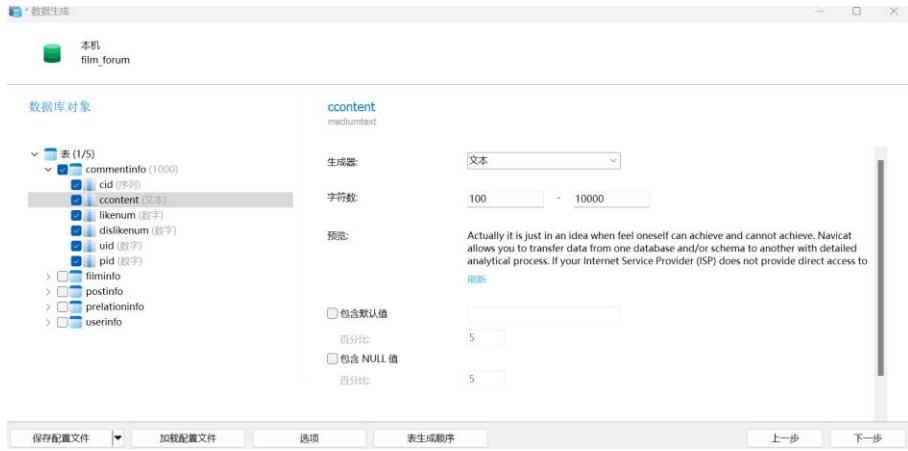
在数据库的建立与维护升级的过程中，我们使用了 Navicat，Navicat 是一套可创建多个连接的数据库管理工具，可以方便管理 MySQL 等数据库。



确保 MySQL 服务启动后，双击本机连接开启。

新建数据库 film_forum 后，可以新建各种表并添加字段，完成数据库结构设计。

新建好的数据库中没有数据，可以使用数据生成功能，按需求调整好相关参数进行批量数据生成。



若要导出数据库，可以进行如下操作即可导出数据及结构为 sql 文件。

导入数据库，则是运行 sql 文件，选择本地的 sql 文件运行，会覆盖原来的数据库。

六、电影论坛 Web 端的实现 [岑奕侃]

6.1 用户管理的实现

6.1.1 用户注册

```
<input type="username" class="form-control" v-model="user.username" placeholder="请输入用户名" required>
<input type="password" class="form-control" v-model="user.password" placeholder="请输入密码" required>
<input type="confirmPassword" class="form-control" v-model="user.confirmPassword" placeholder="请再次输入密码" required>
```

设计时考虑用户体验，确保表单简洁、直观，避免不必要的步骤或复杂要求。

对敏感数据如密码进行适当的处理，前端不保存明文密码。前后端都需要验证数据格式、长度、唯一性等，防止 SQL 注入、XSS 攻击等安全威胁。

创建了一个表单的 HTML 部分，它包含了三个输入框，分别用于用户输入用户名、密码和确认密码。这里使用了 Vue.js 的双向数据绑定特性（v-model 指令）来实时同步输入值到 Vue 实例的数据对象中。

```
async onRegisterSubmit() {
  if (!this.isPasswordMatch) {
    this.passwordMessage = '两次输入的密码不一致';
    return;
  }

  try {
    const response = await axios.post(url: this.$VUE_API_URL + '/user/signup', this.user);

    if (response.status === 200) {
      this.message = '注册成功!';
    } else {
      throw new Error('注册失败, 未知错误。');
    }
  } catch (error) {
    if (error.response && error.response.data.message === "username exist") {
      this.message = '用户名已存在!';
    } else {
      this.message = '注册时发生错误, 请重试。';
      console.error('注册失败:', error);
    }
  } finally {
    this.passwordMessage = ''; // 清除密码确认错误信息
  }
},
```

首先比较两次输入的密码是否一致，一致后向后端发送注册请求，后端访问数据库，如果发现用户名重复，则返回错误提示，如果没有重复，则显示成功注册。

界面图：



6.1.2 用户登录

```

<div class="rice-content-container" :style="{backgroundColor: '#EBC795'}">
  <input type="username" class="form-control" v-model="username" placeholder="请输入用户名" required>
  <input type="password" class="form-control" v-model="password" placeholder="请输入密码" required>

  <button type="submit">登录</button>
  <p>
    还没有账号?
    <a href="/signup" class="signup-link">点击注册</a>
  </p>

```

这部分与用户注册相似，用户输入正确的用户名和密码即可登录，点击注册高亮字即刻前往注册页面进行注册。

界面图：



6.1.3 用户修改信息

```

<input type="username" class="form-control" v-model="user.username" placeholder="请输入新用户名" required>
<input type="password" class="form-control" v-model="user.password" placeholder="请输入新密码" required>

```

这部分与用户注册相似，用户输入重新修改的用户名和密码即可完成信息更新。此功能是在

```

async updateUserInfo() {
  try {
    const response = await axios.put(url: this.$VUE_API_URL + '/user', this.user);

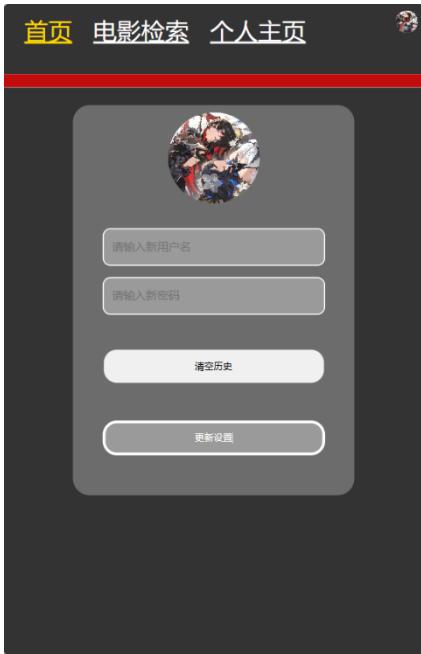
    if (response.status === 200) {
      this.$message.success(text: '更新成功'); // 假设使用element-ui的message组件
    } else {
      throw new Error('Unexpected response status');
    }
  } catch (error) {
    console.error('更新用户信息时出错:', error);
    this.$message.error('更新失败');
  }
},

```

设置界面部分。

调用后端接口，更新用户的信息，user 里存放的是用户的 uid，username 和 password。

界面图：



6.2 电影资源及筛选的实现

6.2.1 电影主页

```
async created() {
  try {
    // 两个接口分别用于获取院线热映和推荐电影数据
    const responseMovies1 = await axios.get(url: this.$VUE_API_URL + '/filmall');
    const responseMovies2 = await axios.get(url: this.$VUE_API_URL + '/filmall');

    // 将获取到的数据赋值给对应的data属性
    this.movies1 = responseMovies1.data.data;
    this.movies2 = responseMovies2.data.data;
  } catch (error) {
    console.error('获取电影数据时出错:', error);
  }
},
```

使用 Vue 组件的生命周期钩子 created() 完成异步数据获取。此段代码的核心目的是从后端接口异步获取两部分电影数据：院线热映和推荐电影，然后将这些数据分别赋值给组件内部的两个数据属性。

成功时，从每个响应的 data.data 层级提取实际需要的电影数据，并将其分别赋值给组件的 movies1 和 movies2 数据属性。

通过 try...catch 语句捕获可能发生的任何异常，如网络错误、API 错误等。如果发生错误，会在控制台打印错误信息，并且可以根据需要在此处添加更多的错误处理逻辑。

```

<div v-for="movie in movies2" :key="movie.fid" class="movies2-item">
  <div class="movies2-info">
    <router-link :to="{ name: 'moviedetail', params: { id: movie.fid } }>
      
    </router-link>
    <div class="movies2-details">
      <router-link :to="{ name: 'moviedetail', params: { id: movie.fid } }" tag="div">
        <h3 class="movies2-title">{{ movie.fname }}</h3>
        <el-rate
          v-model="movie.rank"
          disabled
          show-score
          text-color="#ff9900"
          score-template="{{value}}"
          class="rate">
        </el-rate>
        <p class="movies2-description">导演: {{movie.director}}</p>
        <p class="movies2-description">编辑: {{ movie.scriptwriter }}</p>
        <p class="movies2-description">主演: {{ movie.actor }}</p>
        <p class="movies2-description">类型: {{ movie.type }}</p>
        <!-- 其他电影详情元素... -->
      </router-link>
    </div>
  </div>
</div>

```

通过一个 v-for 循环展示所有从数据库中获得的电影数据，key 为电影的 fid，其中使用了 elementUI 的星级组件，可以图文并茂地展示电影的评分。

界面图：



6.2.2 电影标签筛选

```

<div class="tag-selector">
  <div class="tag-list" ref="tagList">
    <!-- 全部类型特殊处理，可能默认选中 -->
    <div
      class="tag-item"
      :class="{ 'tag-selected': selectedTag2 === '全部类型' || !selectedTag2 }"
      @click="selectTag2( tag: '全部类型' )"
    >全部类型</div>

    <!-- 其他标签 -->
    <div
      v-for="(tag, index) in genreTags"
      :key="index"
      class="tag-item"
      :class="{ 'tag-selected': selectedTag2 === tag }"
      @click="selectTag2( tag )"
    >
      {{ tag }}
    </div>
  </div>
</div>
<div class="tag-selector">

```

首先默认选取为，全部类型和全部地区，设置布局格式，这里参照了 B 站的标签选取模式，多行标签且可滑动查看标签，多行标签取交集。

```

computed: {
  filteredMovies() {
    // 如果选中的是“全部类型”，则直接返回原始列表
    if (this.selectedTag2 === '全部类型' && this.selectedTag3 === '全部地区'){
      return this.movies1;
    }
    else if (this.selectedTag2 === '全部类型') {
      return this.movies1.filter(movie => movie.area === this.selectedTag3);
    }
    else if (this.selectedTag3 === '全部地区') {
      return this.movies1.filter(movie => movie.type === this.selectedTag2);
    }
    // 否则，根据类型筛选
    return this.movies1.filter(movie => movie.type === this.selectedTag2 && movie.area === this.selectedTag3);
  },
},

```

定义了一个名为 `filteredMovies` 的计算属性。好处是使得在处理数据过滤、排序等操作时能有效提高性能，避免不必要的计算。

筛选逻辑围绕着根据用户选择的筛选条件电影类型影地区来过滤原始电影列表，分为三种逻辑：

全选情况处理：

如果用户选择了“全部类型”且同时选择了“全部地区”，意味着没有应用任何筛选条件，因此直接返回原始电影列表。

单一条件筛选：

当类型是“全部类型”但地区有特定选择时，只根据地区 (`movie.area === this.selectedTag3`) 来过滤电影。

相反，如果地区是“全部地区”而类型有特定选择，则仅根据类型 (`movie.type === this.selectedTag2`) 来过滤电影。

复合条件筛选：

当用户既选择了特定类型又选择了特定地区时，执行最严格的筛选，即电影必须同时满足选定的类型和地区的条件 (`movie.type === this.selectedTag2 && movie.area ===`

this.selectedTag3)。

通过这种方式，filteredMovies 动态地提供了根据用户选择更新的、过滤后的电影列表，而无需修改原始数据，保证了视图的响应式更新。

界面图：



6.2.3 电影详细介绍

```
// 异步加载所有帖子
async loadFilm_posts() {
  try {
    // 发送GET请求到'/film_post'以获取所有帖子
    const response = await axios.get(url: this.$VUE_API_URL + '/film_post/' + this.$route.params.id);
    // 将返回的评论数据赋值给this.film_posts
    this.film_posts = response.data.data;
    // console.log(this.film_posts[2])
  } catch (error) {
    // 如果发生错误, 打印错误信息
    console.error('Error loading film_posts:', error);
  }
}

// 异步加载指定ID的电影详情
async loadMovie() {
  try {
    // 发送GET请求到'/film/{id}', 其中'{id}'是当前电影的ID
    const response = await axios.get(url: this.$VUE_API_URL + '/film/' + this.$route.params.id);
    // 将返回的电影数据赋值给this.movie
    this.movie = response.data.data;
    console.log(response.data);
  } catch (error) {
    // 如果发生错误, 打印错误信息
    console.error('Error loading movie:', error);
  }
},
```

通过两个异步方法，分别从后端异步加载单个电影的详细信息和与该电影相关的所有帖子。loadMovie() 方法根据当前路由参数中的电影 fid (this.\$route.params.id)，向服务器发出 GET 请求来获取指定电影的详细信息。成功获取数据后，将响应中的电影详情赋值给组件的 this.movie 属性。如果在请求过程中遇到任何错误，它将在控制台打印错误信息。

`loadFilm_posts()` 同样依据当前电影 `fid`, 此方法向另一个 API 端点发送 GET 请求来加载与该电影关联的所有帖子。获取到的帖子数据会被存储在组件的 `this.film_posts` 属性中。与上一个方法一样, 它也包含了错误处理逻辑, 用于捕捉并记录请求过程中的错误。

这两个方法通过使用 `async/await` 语法简化了基于 `Promise` 的异步操作, 提高了代码的可读性和维护性。它们共同作用于提升用户体验, 确保在用户查看某个电影页面时, 能够快速加载并展示该电影的详细信息及其相关的社区讨论内容。

关于这个页面, 还有两个点击事件的方法。

```
// 导航到上传帖子的页面
goToUploadPost() {
  console.log(this.movie.fid)
  this.$router.push({
    path: '/uploadpost/',
    query: { fid: this.movie.fid}, // 假设movie是当前组件data中的一个对象, 包含fid属性
  });
},
```

`goToUploadPost()` 是下方“我要发帖”按钮的点击事件, 点击后会跳转到发帖页面, 并把 `fid` 传过去。

还有一个是添加浏览记录并跳转到指定帖子的点击事件方法。

```
async gethistory(film_post) {
  try {
    // 构建对象
    const historyData = {
      uid: this.uid, // 用户ID
      pid: film_post.pid, // 帖子ID
    };
    console.log(historyData.uid);
    console.log(historyData.pid);
    // 发送POST请求到接口
    const response = await axios.post(url: this.$VUE_API_URL+'/addHistoryPost', historyData);

    if (response.status === 200) {
      await this.$router.push({name: 'postdetails', query: {id1: film_post.fid, id2: film_post.pid}});
    }
  } catch (error) {
    console.error('请求错误:', error);
    alert('发生错误, 请检查网络或稍后重试');
  }
},
```

<div class="post-grey-content-container" :style="{ backgroundColor: '#6C6C6C' }">
 <div @click="gethistory(film_post)">
 <div class="film_posts-info">

使用异步方法 `gethistory` 为用户添加历史浏览记录。

构建一个包含用户 ID (`uid`) 和帖子 ID (`pid`) 的请求体对象 `historyData`。

如果服务器响应状态码为 200, 表示添加历史记录成功, 方法将进一步导航用户至名为 ‘`postdetails`’ 的路由页面, 通过查询参数传递帖子所属电影 ID (`fid`) 和帖子 ID (`pid`), 以便展示帖子详情。

界面图：



6.3 电影帖子的实现

6.3.1 帖子详情

```
<div class="posts-info">
  
  <span class="posts-username">{{ posts.username }}</span>
</div>
<el-rate
  v-model="posts.value"
  disabled
  show-score
  text-color="#ff9900"
  score-template="{{value}}"
  class="posts-rate">
</el-rate>
<h1 class="posts-title">{{ posts.ptitle }}</h1>
<p class="posts-details">{{ posts.pcontent }}</p>

```

此处为帖子内容的布局

```
<div v-for="comment in post_comments" :key="comment.cid">
  <div class="comment-grey-content-container" style="backgroundColor: '#6C6C6C'>
    <div class="post_comments-info">
      
      <span class="post_comments-username">{{comment.username}}</span>
      <p class="post_comments-comment">{{ comment.ccontent }}</p>
      <div class="icon d-flex justify-content-between align-items-center">
```

此处为下方评论的布局

展示了帖子的所有内容，包括发帖人信息，标题，评分，内容（文字+图片）等。下方是帖子浏览量，点赞和收藏操作。最后是评论区。

界面图：



6.3.2 发布帖子

```
<input type="text" class="title-input" v-model="title" placeholder="标题">
<div class="separator-line"></div>

<input type="text" class="detail-input" v-model="content" placeholder="内容">
<input type="file" ref="fileInput" @change="onFileChange" style="display:none">

<div class="separator-line2"></div>
<i class="el-icon-picture" @click="uploadPic"></i>
```

给定两个输入框，一个输入帖子标题，一个输入帖子内容（文字加图片）。图片的上传是通过下方的按钮，上传本地图片。这里介绍一下几个实现函数。

```
convertToXML(reviewData) {
  let xmlString = '<review>';
  for (let key in reviewData) {
    xmlString += `<${key}>${reviewData[key]}</${key}>`;
  }
  xmlString += '</review>';
  return xmlString;
},
convertToXML(reviewData)
```

```
// 当文件选择发生变化时触发
onFileChange(e) {
  const file = e.target.files[0];
  if (!file) {
    this.post.postimageUrl = '';
    return;
  }
  // 然后用返回的URL更新this.imageUrl
  this.previewImage(file);
},
```

将 JavaScript 对象转换成 XML 字符串格式。遍历 reviewData 对象的所有属性，为每个属性生成一个 XML 标签，包括标签名和对应的值，最终封装在<review>标签内。

onFileChange(e)

当用户选择上传文件时被触发。此函数处理文件输入事件，获取选中的文件。如果没有选择文件，则清空预览图片地址；否则，调用 previewImage(file) 预览图片。

```
// 预览图片
previewImage(file) {
  const reader = new FileReader();
  reader.onload = (e) => {
    this.post.previewurl = e.target.result;
    console.log(this.post.previewurl);
  };
  reader.readAsDataURL(file);
},
uploadPic(){
  this.$refs.fileInput.click();
},
previewImage(file)
```

previewImage(file)
预览用户选择的图片。利用 FileReader API 读取文件内容，并将文件转换为 Base64 编码的 URL，用于在网页上预览。转换完成后，将预览 URL 赋值给 this.post.previewurl。

uploadPic()

模拟点击文件输入框，触发文件选择对话框的打开。通常用于用户界面中隐藏实际的文件输入元素，而通过按钮触发展示文件选择对话框。

```
try {
  // 转换数据为XML格式
  const reviewDataXML = this.convertToXML( reviewData: {
    value: this.value,
    ptitle: this.title,
    pcontent: this.content,
    uid: this.uid,
    fid: this.$route.query.fid,
    picurl : this.post.previewurl
  });
}
```

submitReview()

在这个函数中，我们将数据转化成后端需要的 XML 格式，并调用接口。

界面图：



6.3.3 帖子点赞、收藏、浏览

以收藏为例，点赞和浏览的逻辑都是一样的，只是调用的接口名称不一样，故不在此大篇幅赘述。

```
async toggleFavorite() {
  this.favorited = !this.favorited;
  if (this.favorited) {
    this.posts.collections++;
  } else {
    this.posts.collections--;
  }
  try {
    // 构建收藏对象
    const collectData = {
      uid: this.uid, // 用户ID
      pid: this.$route.query.id2, // 帖子ID
    };
    console.log(collectData.uid);
    console.log(collectData.pid);
    // 发送POST请求到收藏接口
    const response = await axios.post(url: this.$VUE_API_URL+'/_collect', collectData);

    if (response.status === 200) {
      console.log('收藏成功');
      alert('收藏成功');
    } else {
      console.error('收藏失败');
      alert('收藏失败，请重试');
    }
  } catch (error) {
    console.error('请求错误：', error);
    alert('发生错误，请检查网络或稍后重试');
  }
},

```

如果此帖子未被收藏，则点击后，收藏图标由原来的白色变为黑色，且收藏数增加，提示用户收藏成功。

6.4 用户个性化标签的实现

6.4.1 用户标签

```
<div class="nav-item-plus">
  <el-popover
    placement="bottom"
    trigger="click"
    width="300"
    height="auto"
  >
    <div class="posts-popover">
      <div v-for="(uniqueTag, index) in uniqueTags" :key="index">
        <el-button class="posts-button" type="primary" @click="addTags(uniqueTag)">
          {{uniqueTag}}
        </el-button>
      </div>
    </div>
    <el-icon slot="reference" class="el-icon-circle-plus-outline"></el-icon>
  </el-popover>
</div>
```

这处布局为添加关注电影类型的按钮，我使用了 emementUI 中的嵌套弹出框，使界面更加整洁，操作更加方便。点击内部嵌套的按钮，可以将选中的类型显示到“关注电影类型”部分，接下来讲一讲函数的实现。

```

async addTags(tag) {
  try {
    const addData = {
      uid: this.uid,
      newTags: tag
    };
    const response = await axios.post(url: this.$VUE_API_URL+'updateTags/', addData, config: {
      headers: { 'Content-Type': 'application/x-www-form-urlencoded' }
    });

    if (response.status === 200) {
      console.log("添加标签" + addData.newTags + "成功");

      // 确保新标签不在selectedtags中才添加，避免重复
      if (!this.selectedtags.includes(tag)) {
        this.selectedtags.push(tag);
      }
    }
  } catch (error) {
    console.error("添加标签时出错:", error);
  }
},

```

`addTags()`函数做了两件事，第一件事，把选中的标签传给后端数据库，第二件事，比较此标签是否不在已选择范围内。

```

async created() {
  try {
    const response = await axios.get(url: this.$VUE_API_URL+'/tags/'+this.uid);

    // 将获取到的数据赋值给对应的data属性
    this.dbTagsString = response.data.data;
    console.log(this.dbTagsString)
    // 将字符串转换为数组
    this.selectedtags = this.dbTagsString[0].split(separator: ',');

    console.log(this.selectedtags)
  } catch (error) {
    console.error('获取标签数据时出错:', error);
    // 可以在这里处理错误，例如显示错误消息给用户
  }
},

```

这里是显示用户已选择的标签。

为了，界面美观和用户体验，已被选择的标签，不应该在弹出框中出现。那么，如何才能防止已选择的标签不会出现在弹出框中呢，于是有了下面的函数。

```

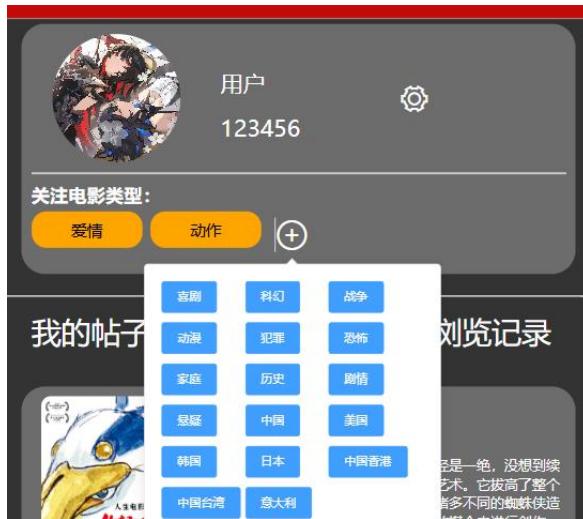
<div v-for="(uniqueTag, index) in uniqueTags" :key="index">
  <el-button class="posts-button" type="primary" @click="addTags(uniqueTag)">
    {{uniqueTag}}
  </el-button>
</div>

computed: {
  // 计算属性用于过滤掉selectedtags中存在的标签
  uniqueTags() {
    // 使用filter方法从tags中过滤掉重复的标签
    return this.tags.filter(tag => !this.selectedtags.includes(tag));
  }
},

```

`UniqueTags()` 把那些存在于展示处的标签，全部过滤掉，剩下来的标签就是未被选择的。这里为了响应的及时性，采用了，Vue 的响应式布局，可以实时显示已选择的标签而无需刷新。

界面图：



6.4.2 用户展示已发布帖子、已收藏帖子、已浏览帖子 这三者的实现逻辑是一样的。

```
async goToMyPosts() {
    // 在此处添加跳转到“我的帖子”页面的逻辑
    this.activeTab = 'posts';
    try {
        const response = await axios.get(url: this.$VUE_API_URL + '/findPost/' + this.uid); // API路径
        this.myPosts = response.data.data; // 假设返回的数据直接赋值给myPosts
    } catch (error) {
        console.error("Error loading posts:", error);
    }
},
```

调用不同的接口，`get` 到不同的数据，这关系到一张名为 `prelation` 的表，里面记录了用户和帖子的各种关系，比如发布(r)，喜欢(l)，浏览(h)，收藏(c)等。

界面图：

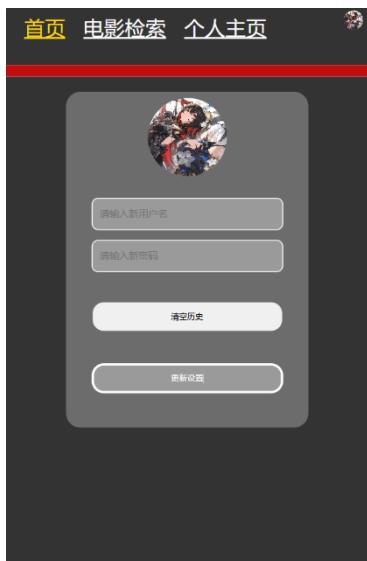


6.4.3 清除浏览记录

```
async deleteHistory(uid) {
  try {
    const response = await axios.get(url: this.$VUE_API_URL + '/deleteHistory/' + uid);
    console.log(uid)
    if (response.status === 200) {
      this.$message.success(text: '删除浏览历史成功'); // 假设使用element-ui的message组件
    } else {
      throw new Error('Unexpected response status');
    }
  } catch (error) {
    console.error('删除浏览历史时出错:', error);
    this.$message.error('删除浏览历史失败');
  }
},
```

`deleteHistory(uid)` 函数接受一个参数 `uid`, 表示用户唯一标识符, 用于确定要删除哪个用户的浏览历史记录。使用 GET 方法以保持 RESTful 原则。

界面图:



七、后端的实现 [梁宝丹]

(一) 用户模块

1. 用户登陆, 注册

用户登录: 因为设置用户名唯一, 根据用户名在数据库中查找密码, 判断用户输入密码与数据库中密码是否相同。首先在通过 Post 方法接受用户名和用户密码, 并通过 `userService` 的 `login` 方法调用 Mapper 层去访问与数据库中用户名和密码, 并做匹配, 如果匹配成功, 返回 code81, 失败返回 code26。

```

// 用户登录验证
@PostMapping(value="/user/login")
public ResponseEntity<Object> loginUser(@RequestParam String username,@RequestParam String password) {
    User user = userService.login(username, password);
    if (user != null) {
        // 登录成功，返回成功的 code
        return ResponseEntity.ok(Collections.singletonMap("code", 81));
    } else {
        // 用户名或密码错误，返回相应的 code
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(Collections.singletonMap("code", 26));
    }
}

// 用户注册（用户名不能重复）
@PostMapping(value="/user/signup")
public ResponseEntity<?> signUpUser(@RequestBody UserRequest userRequest) {
    boolean reulst = userService.registerUser(userRequest);
    if (reulst){
        return ResponseEntity.ok().body("{\"code\": 200, \"message\": \"Sign up successful\"}");
    }else{
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("{\"code\": 500, \"message\": \"username exist\"}");
    }
}

```

用户注册: 因为设置用户名唯一, 判断用户名是否在数据库已存在, 若不存在, 向数据库添加该用户的记录, 代表注册成功。通过 Post 方法接受一个 user 实体, 包含用户注册时填写的信息, 通过 userService 的 registerUser 方法调用 Mapper 层方法, 访问数据库中 UserInfo 表, 向表中添加该用户记录。

```

2 usages
public boolean registerUser(UserRequest user){
    int if_exist = userMapper.findUserByName(user.getUsername());
    if (if_exist>0){
        return false;
    }
    else {
        int num = userMapper.getMaxUid();
        User new_user = new User( uid: num + 1, user.getUsername(), user.getPassword(), aaurl: null, tags: null);
        int result = userMapper.insertUser(new_user);
        if (result > 0) {
            return true;
        } else {
            return false;
        }
    }
}

<!-- 用户登陆 -->
<select id="findByUsernameAndPassword" resultType="com.example.demo.pojo.User">
    SELECT * FROM userinfo WHERE username = #{username} AND password = #{password}
</select>

<!-- 用户注册 -->
<insert id="insertUser" parameterType="com.example.demo.pojo.User">
    INSERT INTO userinfo (uid,username, password) VALUES (#{uid},#{username}, #{password})
</insert>

```

2. 用户个人信息修改

修改用户信息: 如果用户修改用户名, 需要判断用户名是否已存在。根据唯一的用户 ID 查询数据库中用户信息, 并对应修改。通过 Put 方法接受用户修改的信息, 并通过 Mapper 层在数据库的 userinfo 表, 依次判断用户名, 密码, 头像, 喜爱标签是否有更改信息, 如果有修改表中记录信息。在修改用户名时需要在

Service 层先做对应判断用户想要修改后的名字是否与已有用户重名，如果重名需要返回修改失败。

```
// 修改用户信息
@PutMapping(PathVariable="/user")
public ResponseEntity<?> UpdateUserInfo(@RequestBody User updateUser) {
    boolean result = userService.updateUser(updateUser);
    if (result){
        return ResponseEntity.ok().body("{\"code\": 200, \"message\": \"Update successful\"}");
    }else{
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("{\"code\": 500, \"message\": \"Update failed\"}");
    }
}

@Override
public boolean updateUser(User updateUser) {
    //如果修改用户名，判断是否重名
    String new_name = updateUser.getUsername();
    if (userMapper.findUserByName(new_name)>0){
        return false;
    }
    int result = userMapper.updateUserInfo(updateUser);
    if(result>0){
        return true;
    }else{
        return false;
    }
}
```

```
<!-- 更新用户信息 -->
<update id="updateUserInfo" parameterType="com.example.demo.pojo.User">
    UPDATE userinfo
    <set>
        <if test="username != null">username = #{username},</if>
        <if test="password != null">password = #{password},</if>
        <if test="avaurl != null">avaurl = #{avaurl},</if>
        <if test="tags != null">tags = #{tags},</if>
    </set>
    WHERE uid = #{uid}
</update>
```

3. 收藏帖子，查询发布过的帖子，查询收藏过的帖子

收藏帖子：添加用户和该帖子的关系为收藏，同时该帖子的收藏数+1。通过 Post 方法获取用户信息和帖子信息，通过 Mapper 层去访问数据库，首先查看该用户是否已经收藏过该帖子，如果已经修改过了，不做任何操作，直接返回；如果没有收藏过，首先在 prelationinf 表中添加用户和帖子的收藏关系，再修改 postinfo 中该帖子的收藏数+1。

```

//收藏帖子(添加关系+收藏数+1)
@PostMapping(value="/collect")
public ResponseEntity<?> collectPost(@RequestBody Collect collect) {
    Relation relation = new Relation(collect.getUid(),collect.getPid(), relation: "c");
    boolean reulst = userService.collectPost(relation);
    if (reulst){
        return ResponseEntity.ok().body("{\"code\": 200, \"message\": \"Collect successful\"}");
    }else{
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("{\"code\": 500, \"message\": \"Collect failed\"}");
    }
}

@Override
public boolean collectPost(Relation relation) {

    userMapper.addCollectNum(relation.getPid());
    int result = userMapper.addRelation(relation);
    if (result>0){
        int row = postMapper.collect(relation.getPid());
        if (row>0){
            return true;
        }
    }
    return false;
}

<!-- 添加关系 -->
<insert id="addRelation" parameterType="com.example.demo.pojo.Relation">
    INSERT INTO prelationinfo (pid,relation,uid) VALUES (#{pid},#{relation}, #{uid})
</insert>

<!-- 收藏数+1 -->
<update id="addCollectNum" parameterType="Integer">
    UPDATE postinfo
    <set>collectnum = collectnum + 1</set>
    WHERE pid = #{pid}
</update>

```

查询发布的帖子：直接从数据库中查询该用户 id 的帖子。首先通过 Get 方法获取用户唯一的 ID，通过 Mapper 层访问数据库的 postinfo 表，查询所有用户 ID 匹配的帖子。并将该帖子的作为帖子实体类返回。

```
//查询发布过的帖子
@GetMapping("findPost/{uid}")
public ResponseEntity<?> findPost(@PathVariable Integer uid){
    List<Post> posts = userService.findPost(uid);
    Map<String, Object> responseData = new HashMap<>();
    responseData.put("code", 0);
    responseData.put("data", posts);
    return ResponseEntity.ok(responseData);
}
```

```
<!-- 查询发布帖子 -->
<select id="findPost" parameterType="Integer">
    SELECT * FROM postinfo
    WHERE uid = #{uid}
</select>
```

查询收藏的帖子：直接查找数据库中与该用户关系为收藏关系的帖子。首先通过 Get 方法获取到用户 ID，通过 Mapper 层首先查询 Prelation 表与该用户关系为收藏的 post ID，再向 postinfo 表里查询帖子

```
//查询收藏过的帖子
@GetMapping("findCollectedPost/{uid}")
public ResponseEntity<?> findCollectedPost(@PathVariable Integer uid){
    List<Post> posts = userService.findCollectedPost(uid);
    Map<String, Object> responseData = new HashMap<>();
    responseData.put("code", 0);
    responseData.put("data", posts);
    return ResponseEntity.ok(responseData);
}
```

```
<!-- 查询收藏的帖子 -->
<select id="findCollectedPost" parameterType="Integer">
    SELECT * FROM postinfo
    WHERE postinfo.pid in (SELECT prelationinfo.pid FROM prelationinfo WHERE prelationinfo.uid =#{uid} and prelationinfo.relation = 'c')
</select>
```

4. 查询，清除浏览历史。

```

//查询浏览历史
@GetMapping(PathVariable Integer uid)
public ResponseEntity<?> findHistory(@PathVariable Integer uid){
    List<Post> posts = userService.findHistory(uid);
    Map<String, Object> responseData = new HashMap<>();
    responseData.put("code", 0);
    responseData.put("data", posts);
    return ResponseEntity.ok(responseData);
}

//删除浏览历史
@GetMapping(PathVariable Integer uid)
public ResponseEntity<?> deleteHistory(@PathVariable Integer uid){
    boolean result = userService.deleteHistory(uid);
    if (result){
        return ResponseEntity.ok().body("{\"code\": 200, \"message\": \"Delete history successful\"}");
    }else{
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("{\"code\": 500, \"message\": \"Delete history failed\"}");
    }
}

```

5. 查询浏览历史：直接在数据库中查找与用户关系为浏览历史的帖子。首先通过 Get 方法获取到用户的 uid，通过 Mapper 层访问数据库的关系 prelation 表，查找该表中与该 uid 关系为浏览历史的所有记录。

```

<!-- 查询浏览历史 -->
<select id="findHistoryPost" parameterType="Integer">
    SELECT * FROM postinfo
    WHERE postinfo.pid IN (SELECT prelationinfo.pid FROM prelationinfo WHERE prelationinfo.uid =#{uid} AND (prelationinfo.relation = 'h'))
</select>

```

6. 删除浏览历史：如果用户浏览历史不为空，则删除数据库中与用户关系为浏览历史的记录。首先通过 Get 方法获取到用户的 uid，通过 Mapper 层访问数据库的关系 prelation 表，删除该表中与该 uid 关系为浏览历史的所有记录。

```

<!-- 清空浏览历史 -->
<delete id="deleteHistory" parameterType="Integer">
    DELETE FROM prelationinfo WHERE uid = #{uid} AND (relation = 'h')
</delete>

```

(二) 电影模块

1. 获取电影具体信息

查询电影信息：直接根据电影唯一的 ID，在数据库中查找该电影的全部信息。首先通过 Get 方法获取电影的 fid，通过 Mapper 层访问数据库的 filminfo 表，查询表中符合该 fid 的所有记录，并返回。

```

//查询电影 fid (integer)
@GetMapping("/{fid}")
public ResponseEntity<Object> getFilmByFid(@PathVariable Integer fid) {
    System.out.println(fid);
    Film film = filmService.getFilmByFid(fid);
    if (film != null) {
        // 构建成功时返回的 JSON 数据
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data", film);
        return ResponseEntity.ok(responseData);
    } else {
        // 返回 400 Response, 空的 JSON 对象
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Collections.emptyMap());
    }
}

```

```

<!-- 根据fid查询电影 -->
<select id="getFilmByFid" resultType="com.example.demo.pojo.Film">
    SELECT * FROM filminfo WHERE fid = #{fid}
</select>

```

2. 获取电影列表

查询电影列表：直接查找数据库中所有电影的具体信息。通过 Mapper 层访问数据库的 filminfo 表，将所有电影记录返回。

```

@GetMapping("/filmall")
public ResponseEntity<Object> getFilm() {
    List<Film> film = filmService.getFilm();
    if (film != null) {
        // 构建成功时返回的 JSON 数据
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data", film);
        return ResponseEntity.ok(responseData);
    } else {
        // 返回 400 Response, 空的 JSON 对象
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Collections.emptyMap());
    }
}

```

```

<!-- 返回所有电影 -->
<select id="findAllFilms" resultType="com.example.demo.pojo.Film">
    select * from filminfo
</select>

```

3. 根据标签查询电影

查询符合标签类型的电影：直接查找数据库中同时符合所有类型的电影。首先通过 Post 方法获取查询电影的标签，通过 Service 层将获得的标签进行一些处理，Mapper 层访问数据库的 filminfo 表，查询表中满足所有标签类型的电影记录，并返回。

```

//根据标签查询电影
@PostMapping(value="/film")
public ResponseEntity<Object> getFilmByTags(@RequestBody List<String> movieTypes) {
    List<Film> films = filmService.getFilmByTags(movieTypes);
    if (films != null) {
        log.info("success");
        // 构建成功时返回的 JSON 数据
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data", films);
        return ResponseEntity.ok(responseData);
    } else {
        // 返回 400 Response, 空的 JSON 对象
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Collections.emptyMap());
    }
}

```

```

<!-- 根据Tags查找电影 -->
<select id="getFilmsByTags" parameterType="java.util.List" resultType="com.example.demo.pojo.Film">
    SELECT * FROM filminfo
    <where>
        <if test="movieTypes != null and movieTypes.size() > 0">
            AND (
                <foreach collection="movieTypes" item="movieType" separator="AND" open="" close="" index="">
                    type LIKE CONCAT('%', #{movieType}, '%')
                </foreach>
            )
        </if>
    </where>
</select>

```

(三) 帖子模块

1. 查询某电影的所有帖子

查询电影相关帖子：直接根据唯一的电影 ID，在数据库中查找相关的帖子。首先通过 Get 方法获取电影的 fid，通过 Mapper 层访问数据库中的帖子表 postinfo 表，查找符合该 fid 的记录，并返回。

```

//根据电影fid查询所有帖子 pid (integer)
@GetMapping(value="/film_post/{fid}")
public ResponseEntity<Object> getPostsByFid(@PathVariable Integer fid) {
    System.out.println(fid);
    List<PostShow> film_posts = postService.getPostsByFid(fid);
    if (film_posts != null) {
        // 构建成功时返回的 JSON 数据
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data", film_posts);
        return ResponseEntity.ok(responseData);
    } else {
        // 返回 400 Response, 空的 JSON 对象
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Collections.emptyMap());
    }
}

```

```
<!-- 根据fid查询所有电影帖子 -->
<select id="getPostsByFid" resultType="com.example.demo.pojo.PostShow">
    SELECT p.*, u.username, u.avaurl
    FROM postinfo p
        JOIN userinfo u ON p.uid = u.uid
    WHERE p.fid = #{fid}
</select>
```

2. 查询某个帖子

查询帖子：直接根据唯一的帖子 ID，直接在数据库中查询记录。首先通过 Get 方法获取帖子的 pid，通过 Mapper 层访问数据库中的帖子表 postinfo 表，查找符合该 pid 的记录，并返回。

```
//查询帖子 pid (integer)
@GetMapping("/{post_detail}/{pid}")
public ResponseEntity<Object> getPostByPid(@PathVariable Integer pid) {
    System.out.println(pid);
    PostShow post = postService.getPostByPid(pid);
    if (post != null) {
        // 构建成功时返回的 JSON 数据
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data", post);
        return ResponseEntity.ok(responseData);
    } else {
        // 返回 400 Response, 空的 JSON 对象
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Collections.emptyMap());
    }
}
```

```
<!-- 查找帖子 -->
<select id="findPostById" parameterType="Integer">
    select * from postinfo
    where pid = #{pid}
</select>
```

3. 发布帖子

发布帖子：根据用户发布的帖子信息，生成唯一的帖子 ID，直接在数据库中添加一条记录。首先通过 Post 方法获取用户的帖子信息，并将该信息生成一个 Post 实体类，通过 Mapper 层访问数据库的帖子 postinfo 表，插入该记录。

```

//发布帖子
@PostMapping(path = "/uploadpost", consumes = MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<Object> createPostFromXml(@RequestBody PostRequest postRequest) {
    Post post = postService.createPost(postRequest);
    if(post!=null){
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data",post);
        return ResponseEntity.ok(responseData);
    }
    else{
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 1);
        return ResponseEntity.ok(responseData);
    }
}

<!-- 发布帖子 -->
<insert id="insertPost" parameterType="com.example.demo.pojo.Post">
    INSERT INTO postinfo (pid, uid, pcontent, picurl, seenum, likenum, collectnum,value,ptitle,fid)
    VALUES (#pid, #uid, #{pcontent}, #{picurl}, #{seenum}, #{likenum}, #{collectnum}, #{value}, #{ptitle}, #{fid})
</insert>

```

4. 更新浏览历史

添加浏览记录: 判断之前是否已经浏览过该帖子，即向数据库中查询是否存在该用户和该帖子的浏览历史的关系记录，如果已经存在，不做任何处理；如果不存在，向数据库添加一条关系记录。

```

//添加浏览历史
@PostMapping(path = "/addHistoryPost")
public ResponseEntity<?> findPost(@RequestBody Relation relation_history){
    Relation relation = new Relation(relation_history.getId(),relation_history.getPid(), relation: "h");
    boolean result = postService.addHistoryPost(relation);
    if (result){
        return ResponseEntity.ok().body("{\"code\": 200, \"message\": \"add history successful\"}");
    }else{
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("{\"code\": 500, \"message\": \"add history failed\"}");
    }
}

//添加浏览历史
2 usages
@Override
public boolean addHistoryPost(Relation relation_history) {
    //如果之前已经浏览过了就不添加了
    int count = postMapper.findRelation(relation_history);
    if (count>0){
        return true;
    }
    int result = postMapper.addHistoryPost(relation_history);
    if (result>0){
        return true;
    }else{
        return false;
    }
}

```

```

<!-- 添加浏览历史 -->
<insert id="addHistoryPost" parameterType="com.example.demo.pojo.Relation">
    INSERT INTO prelationinfo (pid, uid, relation)
    VALUES (#{pid}, #{uid}, #{relation})
</insert>

```

5. 帖子点赞

点赞帖子：判断之前是否已经点过赞，即查找数据库中是否已经已经存在该用户和该帖子的点赞关系，如果已经存在，不做任何处理；如果不存在，该帖子的点赞数+1，并添加用户和帖子的点赞关系。

```

//帖子点赞
@PostMapping(path = "/postlike")
public ResponseEntity<Object> addPostlike(@RequestParam Integer uid,@RequestParam Integer pid) {
    Relation relation = new Relation(uid,pid, relation: "l");
    boolean result = postService.addPostlike(relation);
    if(result){
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("message", "点赞成功");
        return ResponseEntity.ok(responseData);
    }
    else{
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 1);
        responseData.put("message", "点赞失败");
        return ResponseEntity.ok(responseData);
    }
}

//帖子点赞
2 usages
@Override
public boolean addPostlike(Relation relation) {
    int count = postMapper.findRelation(relation);
    if (count>0){
        //该用户已经对该帖子点过赞了
        return true;
    }
    //该用户还没对该帖子点过赞
    int result_1 = postMapper.addLikePost(relation);
    int result = postMapper.like(relation.getPid());
    if (result_1>0 && result>0){
        return true;
    }
    return false;
}

```

```

<!-- 点赞 -->
<update id="like" parameterType="Integer">
    UPDATE postinfo
    set likenum = likenum+1
    WHERE pid = #{pid}
</update>

```

```

<!-- 添加喜欢关系 -->
<insert id="addLikePost" parameterType="com.example.demo.pojo.Relation">
    insert into prelationinfo (pid, relation, uid) VALUES (#{pid},#{relation},#{uid})
</insert>

```

6. 删除帖子

删除帖子：根据唯一的帖子 ID，在数据库中删除该帖子的记录，包括该帖子，该帖子和其他用户的全部关系记录，该帖子的所有评论记录。首先通过 Delete 方法接受帖子的 pid，通过 Mapper 层访问数据库的 postinfo 表删除该 pid 的记录，同时还要删除 prelation 表中的和该 pid 的相关的记录，评论表 commentinfo 中和该 pid 相关的所有评论记录。

```

//删除帖子
@DeleteMapping("/post/{pid}")
public ResponseEntity<Object> deleteComment(@PathVariable("pid") Integer pid) {
    boolean result = postService.deletePostByPid(pid);
    if (result){
        return ResponseEntity.status(HttpStatus.OK).body(Map.of( k1: "code", v1: 31));
    }
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Map.of( k1: "code", v1: 0));
}

//删除帖子
2 usages
@Override
public boolean deletePostByPid(Integer pid) {
    int result3 = postMapper.deletePostById3(pid);
    int result1 = postMapper.deletePostById2(pid);
    int result2 = postMapper.deletePostById1(pid);
    if (result1>0 && result2>0 && result3>0){
        return true;
    }
    return false;
}

```

```

<!-- 删除postinfo中帖子 -->
<delete id="deletePostById1" parameterType="Integer">
    DELETE FROM postinfo
    WHERE pid = #{pid}
</delete>
<!-- 删除prelationinfo中帖子 -->
<delete id="deletePostById2" parameterType="Integer">
    DELETE FROM prelationinfo
    WHERE pid = #{pid}
</delete>
<!-- 删除commentinfo中帖子 -->
<delete id="deletePostById3" parameterType="Integer">
    DELETE FROM commentinfo
    WHERE pid = #{pid}
</delete>

```

(四) 评论模块

1. 查询某帖子的所有评论

查询具体帖子的全部评论：根据帖子唯一 ID，在数据库中查找该帖子的全部评论。首先通过 Get 方法获取帖子的 pid，通过 Mapper 层访问数据库中评论表 commentinfo 中符合该 pid 的记录，并返回 comment 实体。

```

//根据帖子pid查询所有评论 cid (integer)
@GetMapping(@v"/post_comments/{pid}")
public ResponseEntity<Object> getCommentsByPid(@PathVariable Integer pid) {
    System.out.println(pid);
    List<CommentShow> post_comments = commentService.getCommentsByPid(pid);
    if (post_comments != null) {
        // 构建成成功时返回的 JSON 数据
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data", post_comments);
        return ResponseEntity.ok(responseData);
    } else {
        // 返回 400 Response, 空的 JSON 对象
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Collections.emptyMap());
    }
}

```

```

<!-- 根据pid查询所有帖子评论 -->
<select id="getCommentsByPid" resultType="com.example.demo.pojo.CommentShow">
    SELECT c.*, u.username, u.avaurl
    FROM commentinfo c
        JOIN userinfo u ON c.uid = u.uid
    WHERE c.pid = #{pid};
</select>

```

2. 删除评论

删除某评论：直接根据唯一的评论 ID，删除数据库中这一记录。首先通过 Delete 方法获取评论 cid，通过 Mapper 层去访问数据库里评论表 commentinfo 中符合该 cid 的所有记录。

```
//删除评论
@DeleteMapping(PathVariable="/comment/{cid}")
public ResponseEntity<Object> deleteComment(@PathVariable("cid") Integer cid) {
    boolean result = commentService.deleteCommentByCid(cid);
    if (result){
        return ResponseEntity.status(HttpStatus.OK).body(Map.of( k1: "code", v1: 31));
    }
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Map.of( k1: "code", v1: 0));
}
```

```
<!-- 删除评论 -->
<delete id="deleteCommentById" parameterType="Integer">
    DELETE FROM commentinfo WHERE cid = #{cid}
</delete>
```

3. 评论点赞，点踩，取消点踩

点赞评论：根据唯一帖子 ID，在数据库中对应 ID 的帖子点赞数+1。首先通过 Post 方法获取评论的 cid，通过 Mapper 层访问数据库中评论表 commentinfo 表查找符合该 cid 的记录，对其 like 点赞数项+1。

```
//评论点赞
@PostMapping(path = "/commentlike/{cid}")
public ResponseEntity<Object> addCommentlike(@PathVariable Integer cid) {
    boolean result = commentService.addCommentlike(cid);
    if(result){
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        return ResponseEntity.ok(responseData);
    }
    else{
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 1);
        return ResponseEntity.ok(responseData);
    }
}
```

```
<!-- 点赞 -->
<update id="like" parameterType="Integer">
    UPDATE commentinfo
    set likenum = likenum+1
    WHERE cid = #{cid}
</update>
```

点踩评论：根据唯一帖子 ID，在数据库中对应 ID 的帖子点踩数+1。首先通过 Post 方法获取评论的 cid，通过 Mapper 层访问数据库中评论表 commentinfo 表查找符合该 cid 的记录，对其 dislike 点赞数项+1。

```
//评论点踩
@PostMapping(path = "/dislike/{cid}")
public ResponseEntity<Object> addCommentdislike(@PathVariable Integer cid) {
    boolean result = commentService.addCommentdislike(cid);
    if(result){
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        return ResponseEntity.ok(responseData);
    }
    else{
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 1);
        return ResponseEntity.ok(responseData);
    }
}

<!-- 点踩 -->
<update id="dislike" parameterType="Integer">
    UPDATE commentinfo
    set dislikenum = dislikenum+1
    WHERE cid = #{cid}
</update>
```

取消点踩：根据唯一帖子 ID，在数据库中对应 ID 的帖子点踩数-1。首先通过 Post 方法获取评论的 cid，通过 Mapper 层访问数据库中评论表 commentinfo 表查找符合该 cid 的记录，对其 like 点赞数项-1。

```
//取消评论点踩
@PostMapping(path = "/canceldislike/{cid}")
public ResponseEntity<Object> cancelCommentdislike(@PathVariable Integer cid) {
    boolean result = commentService.cancelCommentdislike(cid);
    if(result){
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        return ResponseEntity.ok(responseData);
    }
    else{
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 1);
        return ResponseEntity.ok(responseData);
    }
}
```

```

<!-- 取消点踩 -->
<update id="canceldislike" parameterType="Integer">
    update commentinfo set dislikenum = dislikenum -1 where cid = #{cid}
</update>

```

4. 发布评论

发布评论：根据用户的帖子信息，生成唯一的帖子 ID，并向数据库中添加一条帖子记录。首先通过 Post 方法获取评论信息，并生成一个 comment 评论实体类，通过 Mapper 层访问数据库中评论表 commentinfo 表，添加该条记录。

```

//发布评论
@PostMapping(path = @"/comment", consumes = MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<Object> createPostFromXml(@RequestBody CommentRequest commentRequest) {
    Integer new_cid = commentService.count()+1;
    Comment comment = new Comment(new_cid,commentRequest.getContent(),commentRequest.getLikenum(),commentRequest.getDislikenum(),
        ,commentRequest.getUid(),commentRequest.getPid());
    boolean result = commentService.createComment(comment);
    if(result){
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 0);
        responseData.put("data",comment);
        return ResponseEntity.ok(responseData);
    }
    else{
        Map<String, Object> responseData = new HashMap<>();
        responseData.put("code", 1);
        return ResponseEntity.ok(responseData);
    }
}

<!-- 发布评论 -->
<insert id="createComment" parameterType="com.example.demo.pojo.Comment">
    INSERT INTO commentinfo (cid,ccontent,likenum,dislikenum,uid,pid)
    VALUES (#{cid}, #{ccontent}, #{likenum}, #{dislikenum}, #{uid}, #{pid})
</insert>

```

八、系统测试 [梁宝丹]

8.1 单元测试

(1) 测试框架：JUnit5 作为测试框架，结合 SpringBoot Test 模块，主要对分别对评论，电影，帖子，用户的核心接口的测试。

(2) 针对 Http 请求响应层（Controller）编写单元测试，验证业务逻辑的正确性。

对评论的部分单元测试：主要对删除评论，对评论点赞，对评论点踩等接口进行了单元测试，测试连接的数据库是本地的数据库。

```

5 usages
@Autowired
private MockMvc mockMvc;

@Test
void deleteComment() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.delete(urlTemplate: "/comment/{cid}", ...uriVariables: 1))
        .andExpect(status().isOk());
}

@Test
void addCommentlike() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate: "/commentlike/{cid}", ...uriVariables: 1))
        .andExpect(status().isOk());
}

@Test
void addCommentdislike() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate: "/dislike/{cid}", ...uriVariables: 1))
        .andExpect(status().isOk());
}

```

对电影的单元测试：主要对通过电影 id 去查找电影详细信息，通过电影名称查找电影详细信息，通过电影类型检索电影详细信息等接口进行了单元测试，测试连接的数据是本地的数据库。

```

@Test
void getFilmByFid() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get(urlTemplate: "/film/{fid}", ...uriVariables: 1))
        .andExpect(status().isOk());
}

@Test
void searchFilmByName() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate: "/film/ser/{fname}", ...uriVariables: "你想活出怎样的人生"))
        .andExpect(status().isOk());
}

@Test
void getFilmByTags() throws Exception {
    // 构造电影类型列表
    List<String> movieTypes = Arrays.asList("喜剧", "动漫");

    // 将电影类型列表转换为 JSON 格式
    String jsonContent = objectMapper.writeValueAsString(movieTypes);

    // 发起 POST 请求，查询电影
    mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate: "/film")
        .contentType(MediaType.APPLICATION_JSON)
        .content(jsonContent))
        .andExpect(status().isOk());
}

```

对帖子的部分单元测试：主要对通过对帖子点赞，取消对帖子的点赞，删除评论等接口进行了单元测试，测试连接的数据是本地的数据库。

```

    @Test
    void addPostlike() throws Exception {
        // 发起 POST 请求, 给文章点赞
        mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate: "/postlike")
            .param(name: "uid", ...values: "1")
            .param(name: "pid", ...values: "2")
        )
        .andExpect(status().isOk());
    }

    @Test
    void cancelPostlike() throws Exception {
        // 发起 POST 请求, 取消文章点赞
        mockMvc.perform(MockMvcRequestBuilders.delete(urlTemplate: "/cancelpostlike")
            .param(name: "uid", ...values: "1")
            .param(name: "pid", ...values: "3"))
        .andExpect(status().isOk());
    }

    @Test
    void deleteComment() throws Exception {
        // 发起 DELETE 请求, 删除评论
        mockMvc.perform(MockMvcRequestBuilders.delete(urlTemplate: "/post/{pid}", ...uriVariables: 2))
        .andExpect(status().isOk());
    }
}

```

对用户的部分单元测试：主要对用户的登陆，注册，更新用户信息等接口进行了单元测试，测试连接的数据库是本地的数据库。

```

    @Test
    void loginUser() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate: "/user/login")
            .param(name: "username", ...values: "Lily")
            .param(name: "password", ...values: "1234"))
        .andExpect(status().isOk());
    }

    @Test
    void signUpUser() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.post(urlTemplate: "/user/signup")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{\"username\": \"Alice\", \"password\": \"1234\"}"))
        .andExpect(status().isOk());
    }

    @Test
    void updateUserInfo() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.put(urlTemplate: "/user")
            .contentType(MediaType.APPLICATION_JSON)
            .content("{\"uid\": \"1\", \"username\": \"Anna\"}"))
        .andExpect(status().isOk());
    }
}

```

(3) 针对服务层（Service）编写单元测试，验证业务逻辑的正确性。

对评论的部分单元测试：主要对删除评论，对评论点赞，对评论点踩等接口进行了单元测试，主要测试能否正确处理获取到的数据库返回信息。测试连接的数据库是本地的数据库。

```
@Test
void deleteCommentByCid() {
    // 测试删除评论功能
    boolean result = commentService.deleteCommentByCid(1);
    assertTrue(result); // 断言删除成功
}

@Test
void addCommentlike() {
    // 测试增加评论点赞功能
    boolean result = commentService.addCommentlike( cid: 2);
    assertTrue(result); // 断言点赞是否成功
}

@Test
void addCommentdislike() {
    // 测试增加评论点踩功能
    boolean result = commentService.addCommentdislike( cid: 2);
    assertTrue(result); // 断言点踩是否成功
}
```

对电影的单元测试：主要对通过电影 id 去查找电影详细信息，通过电影名称查找电影详细信息，通过电影类型检索电影详细信息等接口进行了单元测试，主要测试能否正确处理获取到的数据库返回信息。测试连接的数据库是本地的数据库。

```
@Test
void getFilmByFid() {
    // 测试根据电影ID获取电影信息功能
    Film film = filmService.getFilmByFid(1);
    assertNotNull(film); // 断言电影对象不为空
    // 可以根据实际情况进行进一步的断言验证
}

@Test
void getFilmsByName() {
    // 测试根据电影名称获取电影列表功能
    String filmName = "你想活出怎样的人生";
    // 假设数据库中有符合条件的电影数据
    assertNotNull(filmService.getFilmsByName(filmName)); // 断言获取的电影列表不为空
}

@Test
void getFilmByTags() {
    // 测试根据标签获取电影列表功能
    String tag = "喜剧";
    List<String> tags = Arrays.asList(tag);
    // 假设数据库中有符合条件的电影数据
    assertNotNull(filmService.getFilmByTags(tags)); // 断言获取的电影列表不为空
}
```

对帖子的部分单元测试：主要通过对帖子点赞，取消对帖子的点赞，删除评论等接口进行了单元测试，主要测试能否正确处理获取到的数据库返回信息。测试连接的数据库是本地的数据库。

```
@Test
void addPostlike() {
    // 测试增加文章点赞功能
    Relation relation = new Relation( uid: 1, pid: 2, relation: "l");
    boolean result = postService.addPostlike(relation);
    assertTrue(result); // 断言点赞是否成功
}

@Test
void addHistoryPost() {
    // 测试添加历史文章功能
    Relation relation = new Relation( uid: 1, pid: 3, relation: "h");
    boolean result = postService.addHistoryPost(relation);
    assertTrue(result); // 断言添加历史文章是否成功
}

@Test
void deletePostByPid() {
    // 测试删除文章功能
    boolean result = postService.deletePostByPid(2);
    assertTrue(result); // 断言删除是否成功
}
```

对用户的部分单元测试：主要对用户的登陆，注册，更新用户信息等接口进行了单元测试，主要测试能否正确处理获取到的数据库返回信息。测试连接的数据库是本地的数据库。

```
@Test
void login() {
    // 测试用户登录功能
    String username = "test_user";
    String password = "test_password";
    User user = userService.login(username, password);
    assertNotNull(user); // 断言登录后返回的用户对象不为空
    // 可以根据实际情况进行进一步的断言验证
}

@Test
void registerUser() {
    // 测试用户注册功能
    UserRequest user = new UserRequest();
    user.setUsername("new_user");
    user.setPassword("542");
    // 假设还设置了其他属性...
    boolean result = userService.registerUser(user);
    assertTrue(result); // 断言注册是否成功
}

@Test
void updateUser() {
    // 测试更新用户信息功能
    User user = new User();
    user.setUid(1);
    user.setUsername("updated_user");
    user.setPassword("4432");
    // 假设还设置了其他属性...
    boolean result = userService.updateUser(user);
    assertTrue(result); // 断言更新用户信息是否成功
}
```

(4) 针对数据访问层 (DAO) 编写单元测试，使用内存数据库 (如 H2) 或 Mock 对象来模拟数据库操作。

首先需要配置 Mybatis，连接本地的数据库。并创建 SqlSessionFactory，执行 sql 语句。

对评论的部分单元测试：主要对删除评论，对评论点赞，对评论点踩等接口进行了单元测试，主要测试能否正确访问数据库中表，并执行相关增删改查的操作。

```
@BeforeAll
public static void setUp() throws IOException {
    // 读取 MyBatis 配置文件
    Reader reader = Resources.getResourceAsReader("mybatis-config.xml");
    // 创建 SqlSessionFactory
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
    reader.close();
}

@Test
void deleteCommentById() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        CommentMapper commentMapper = session.getMapper(CommentMapper.class);
        int result = commentMapper.deleteCommentById( cid: 1);
        //验证应该删除一条记录
        assertEquals( expected: 1, result);
    }
}

@Test
void canceldislike() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        CommentMapper commentMapper = session.getMapper(CommentMapper.class);
        int result = commentMapper.canceldislike( cid: 1);
        //验证应该有一条记录修改
        assertEquals( expected: 1, result);
    }
}
```

对电影的单元测试：主要对通过电影 id 去查找电影详细信息，通过电影名称查找电影详细信息，通过电影类型检索电影详细信息等接口进行了单元测试，主要测试能否正确访问数据库中表，并执行相关增删改查的操作。

```

@BeforeAll
public static void setUp() throws IOException {
    // 读取 MyBatis 配置文件
    Reader reader = Resources.getResourceAsReader("mybatis-config.xml");
    // 创建 SqlSessionFactory
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
    reader.close();
}

@Test
void testGetFilmByFid() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        FilmMapper filmMapper = session.getMapper(FilmMapper.class);
        Film film = filmMapper.getFilmByFid(1);
        assertNotNull(film);
        assertEquals(expected: 1, film.getId());
    }
}

@Test
void testFindFilmsByName() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        FilmMapper filmMapper = session.getMapper(FilmMapper.class);
        List<Film> films = filmMapper.findFilmsByName( fname: "你想活出怎样的人生" );
        assertNotNull(films);
        assertFalse(films.isEmpty());
        assertEquals(expected: "你想活出怎样的人生", films.get(0).getFname());
    }
}

```

对帖子的部分单元测试：主要对通过对帖子点赞，取消对帖子的点赞，删除评论等接口进行了单元测试，主要测试能否正确访问数据库中表，并执行相关增删改查的操作。

```

@BeforeAll
static void setUp() throws IOException {
    // 读取 MyBatis 配置文件
    Reader reader = Resources.getResourceAsReader("mybatis-config.xml");
    // 创建 SqlSessionFactory
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
    reader.close();
}

@Test
void testInsertPost() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        PostMapper postMapper = session.getMapper(PostMapper.class);
        Post post = new Post();
        post.setPid(postMapper.getArticlePageCount()+1);
        post.setUid(2);
        post.setPtitle("Test Post");
        post.setPcontent("test new a post");
        post.setSeenum(0);
        post.setCollectnum(0);
        post.setLikenum(0);
        // 设置其他属性...

        int result = postMapper.insertPost(post);

        assertNotNull(post.getPid()); // 确保插入后主键被赋值
        assertEquals(1, result); // 验证插入操作影响的行数
    }
}

```

对用户的部分单元测试：主要对用户的登陆，注册，更新用户信息等接口进行了单元测试，主要测试能否正确访问数据库中表，并执行相关增删改查的操作。

```
@BeforeAll
static void setUp() throws IOException {
    // 读取 MyBatis 配置文件
    Reader reader = Resources.getResourceAsReader("mybatis-config.xml");
    // 创建 SqlSessionFactory
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);
    reader.close();
}

@Test
void testFindByUsernameAndPassword() {
    try (SqlSession session = sqlSessionFactory.openSession()) {
        UserMapper userMapper = session.getMapper(UserMapper.class);
        User user = userMapper.findByUsernameAndPassword( username: "Lily", password: "1234" );

        assertNotNull(user); // 确保用户不为空
        // 根据实际情况验证用户对象的其他属性
    }
}
```

8.2 测试部署及结果

(1) Github 部署

1. 代码提交：将后端代码 Controller 层、Service 层、Dao 层的代码提交到 Github 仓库中。

2. 持续集成(CI)配置：在 Github 仓库中配置 Workflow, CI 工具将自动检测 Github 仓库的更新，并触发构建和测试流程。当 master 分支上有新的代码推送时，自动触发构建和测试流程。当对 master 分支创建新的拉取请求时，也会触发构建和测试流程；代码检出这个步骤会从 Git 仓库中检出指定分支（在这里是 master 分支）的代码到构建机器上。设置 JDK 18 这个步骤会在构建机器上安装 JDK 18 环境，这里指定使用 adopt 发行版的 JDK 18。依赖缓存，Maven 项目通常会有大量的依赖项，这些依赖项需要下载到本地仓库。为了加速构建过程，这个步骤会尝试从缓存中恢复这些依赖项，而不是每次都重新下载。使用 Maven 构建这个步骤使用 Maven 执行 clean 和 package 目标，清理之前的构建并打包项目。这会编译 Java 代码，运行任何编译时的插件（如代码生成），并生成项目所需的任何包（如 JAR 或 WAR 文件）；运行测试，这个步骤使用 Maven 执行 test 目标，运行项目中定义的任何单元测试。Maven 会查找项目源代码目录下的测试类（通常是 src/test/java），并使用 JUnit 运行这些测试。如果所有测试都成功通过，那么这个步骤就会成功；如果有任何测试失败，那么这个步骤就会失败，并且整个构建过程也会失败。

```

1   name: Java CI
2
3   on:
4     push:
5       branches: [ "master" ]
6     pull_request:
7       branches: [ "master" ]
8
9   jobs:
10    build:
11      runs-on: ubuntu-latest
12
13    steps:
14      - uses: actions/checkout@v2
15      - name: Set up JDK 18
16        uses: actions/setup-java@v2
17        with:
18          java-version: '18'
19          distribution: 'adopt'
20      - name: Dependencies Cache
21        uses: actions/cache@v2
22        with:
23          path: ~/.m2/repository
24          key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
25          restore-keys: |
26            ${{ runner.os }}-maven-
27      - name: Build with Maven
28        run: mvn clean package
29      - name: Run Tests
30        run: mvn test

```

(2) 结果

Dao 层测试:

单元测试: 针对 Dao 层的每个方法编写单元测试，验证其数据访问和数据库操作的正确性。

Service 层测试:

单元测试: 验证 Service 层方法的业务逻辑是否正确，包括数据的转换、计算、验证等。

模拟测试: 使用模拟对象（Mock）来模拟 Dao 层的行为，以便在 Service 层测试中隔离 Dao 层的影响。

Controller 层测试:

集成测试: 通过 Apifox 软件，测试接口能否实现 Controller 层与 Service 层之间的交互，确保 Controller 层能够正确地调用 Service 层的方法，并处理 HTTP 请求和响应。

九、系统部署 [梁宝丹]

(一) 环境准备

- 将项目 Docker 部署到本地，本地安装了 Docker 和 Docker Compose。

2. 项目代码包括后端 Spring Boot 项目、前端 Vue 项目以及 MySql 数据库脚本。

(二) Docker 镜像构建

1. SpringBoot 后端的 Dockerfile

首先后端容器 spring 配置依赖环境为 openjdk17，设置接口暴露为 8081，并将后端的打包 jar 包放入 docker 容器目录中。

```
C: > Users > 32796 > Desktop > film_forum > spirng > Dockerfile > ...
1 #依赖jdk8环境
2 FROM openjdk:17
3
4 #对外暴露8081
5 EXPOSE 8081
6 #复制server-1.0-SNAPSHOT到docker容器中并命名为app.jar
7 ADD demo-0.0.1-SNAPSHOT.jar app.jar
8 #执行命令
9 RUN bash -c 'touch /app.jar'
10 ENTRYPOINT ["java", "-jar", "/app.jar"]
11
```

2. Vue 前端的 Dockerfile

首先前端容器 vue 配置依赖环境为 nginx，设置接口暴露为 8080，并将前端的打包 dist 文件夹以及需要用到的图片 pic 文件夹放入 docker 容器目录中。

```
1 #依赖nginx环境
2 FROM nginx
3
4 ADD nginx.conf /etc/nginx/conf.d/
5 # 复制静态文件到nginx静态资源目录
6 COPY dist/ /usr/share/nginx/html/
7 COPY pic/ /usr/share/nginx/html/static
8
9 # 暴露端口
10 EXPOSE 8080
```

(三) Docker Compose 配置

通过 docker-compose.yml 文件文件将前端 spring, 后端 vue, 以及数据库 database 镜像构建容器，并配置后端的数据库连接具体信息，与 database 容器连接，通过 bridge 方式，将三个容器连接，实现前后端以及数据库的访问。

```

1  version: '3.8'
2  services:
3
4    database:
5      image: mysql:8.0
6      environment:
7        MYSQL_DATABASE: film_forum
8        MYSQL_ROOT_PASSWORD: 123456
9        MYSQL_ROOT_HOST: '%'
10     ports:
11       - "3308:3306"
12     command: --default-authentication-plugin=mysql_native_password
13     volumes:
14       - ./mysql_data:/var/lib/mysql
15     networks:
16       - my-network
17
18   app:
19     image: spring
20     container_name: spring
21     build:
22       context: .
23       dockerfile: Dockerfile
24     environment:
25       SPRING_DATASOURCE_URL: jdbc:mysql://database:3306/film_forum
26       SPRING_DATASOURCE_USERNAME: root
27       SPRING_DATASOURCE_PASSWORD: 123456
28       SPRING_DATASOURCE_DRIVER_CLASS_NAME: com.mysql.cj.jdbc.Driver
29     ports:
30       - "8081:8081"
31     depends_on:
32       - database
33     volumes:
34       - ./app_files:/app
35     networks:
36       - my-network
37
38   nginx:
39     image: vue
40     container_name: vue
41     networks:
42       - my-network
43     ports:
44       - "8080:80"
45     privileged: true
46
47   networks:
48     my-network:
49       driver: bridge
50

```

(四) 本地部署

生产镜像: 通过命令 docker build 先生成 spring 后端和 vue 前端的容器, 而数据库的容器在 docker 中直接 pull 一个。再通过 docker-compose up -d 将将三个容器连

接同一个网络，实现项目的本地部署。通过本地的 docker 部署可以实现环境一致性，Docker 容器确保应用程序在本地和生产环境中具有相同的运行环境，从而减少了“在我的机器上运行良好”这类问题。同时容器内包含了应用程序所需的所有依赖项和库，使得在不同操作系统或配置的机器上部署变得更加容易。快速部署： Docker 容器是轻量级的，可以快速启动和停止，从而缩短了部署时间。容器化应用程序可以打包成一个可移植的镜像，并轻松地在任何 Docker 环境中运行。简化开发流程： 开发人员可以在本地环境中使用与生产环境完全相同的容器来测试应用程序，从而提高了开发效率。容器化应用程序可以轻松地与其他开发人员共享，以便在团队中协作开发。通过 Docker Compose 等工具，开发者可以轻松地管理多个容器，创建复杂的、多服务的应用程序架构。版本控制： Docker 镜像可以被版本化并存储在镜像仓库中，从而允许开发者追踪和管理应用程序的变更历史，容器化应用程序可以轻松回滚到以前的版本，以应对潜在的问题或错误。容器化应用程序可以减少对底层操作系统的依赖，从而降低潜在的安全漏洞和攻击面。 Docker 容器提供了强大的隔离性，确保应用程序不会干扰或受到其他应用程序的影响。这种隔离性还允许开发者在同一台机器上同时运行多个版本的应用程序，而不会发生冲突。可移植性： Docker 容器可以在任何支持 Docker 的平台上运行，无论是本地机器、虚拟机还是云服务器。这种可移植性使得容器化应用程序可以轻松地从一个环境迁移到另一个环境。减少依赖冲突： 应用程序所需的依赖项被封装在容器中，这有助于减少不同版本或库之间的依赖冲突。

```
PS C:\Users\32796\Desktop\film_forum\spirng> docker build -f Dockerfile -t spring .
[+] Building 0.9s (8/8) FINISHED
PS C:\Users\32796\Desktop\film_forum\spirng> docker build -f Dockerfile -t vue .
[+] Building 0.0s (0/0) docker:default
2024/06/10 18:40:19 http2: server: error reading preface from client //./pipe/docker_engine: file has already been closed
[+] Building 0.5s (8/8) FINISHED
运行容器
PS C:\Users\32796\Desktop\film_forum\spirng> docker-compose up -d
```

十、功能展示 [岑奕侃]

10.1 在使用功能之前首先需要登录/注册



10.2 登录成功，可以开始浏览电影的内容了，用户可以看看首页推送的电影，也可以筛选特定类型的电影。



10.3 找到了感兴趣的电影，看看它的详细介绍吧，也有不少小伙伴对这部电影发表了自己的看法哦。

你可以看看别人的评价



你想活出怎样的人生

★★★★★ 4

导演: 吉野能 / 岩井俊二
主演: 山崎贤人 / 菅田将晖 / 桥本环奈 / 木村拓哉 / 行天音子 / 风吹淳 / 大竹忍 / 阿川佐和子 / 鹤村隼
类型: 动漫
制片国家/地区: 日本
语言: 日语
上映日期: 2024-04-12T00:00:02
片长: 124

电影简介:
少年救火人(山崎贤人 配音)的母亲葬身火海后, 他随父亲住在一起(木村拓哉 配音)与母亲(木村佳乃 配音)相依为命。深陷悲痛的真人阴阳师(岩井俊二 配音)难以融入新环境。一次意外, 他偶遇一只只会说话的琵琶(菅田将晖 配音)闯入更外的神秘领域。却不断进入了奇幻的“平行世界”, 开始了一场不可思议的冒险。



蜘蛛侠：平行宇宙

★★★★★ 5 我是标题党

本以为《蜘蛛侠：平行宇宙》已经是一绝, 没想到续作更是令人惊喜, 是天花板, 是艺术。它提高了整个超级英雄电影类别的艺术水准, 诸多不同的蜘蛛侠造型都是漫画艺术家们使用了不同的媒介来进行创作, 他们的绘画方式都带一点自己的独特性。有些人使用记号笔、有些人使用油漆刷、钢笔和墨水、铅笔等。续作将它们带入到三维空间中, 让观众感觉自己身临其境。人物在大银幕中移动时

本以为《蜘蛛侠：平行宇宙》已经是一绝, 没想到续作更是令人惊喜, 是天花板, 是艺术。它提高了整个超级英雄电影类别的艺术水准, 诸多不同的蜘蛛侠造型都是漫画艺术家们使用了不同的媒介来进行创作, 他们的绘画方式都带一点自己的独特性。有些人使用记号笔、有些人使用油漆刷、钢笔和墨水、铅笔等。续作将它们带入到三维空间中, 让观众感觉自己身临其境。人物在大银幕中移动时

首页 电影检索 个人主页

写影评: 你想活出怎样的人生

发布

点击星星评分

标题

内容

我可以发表评价

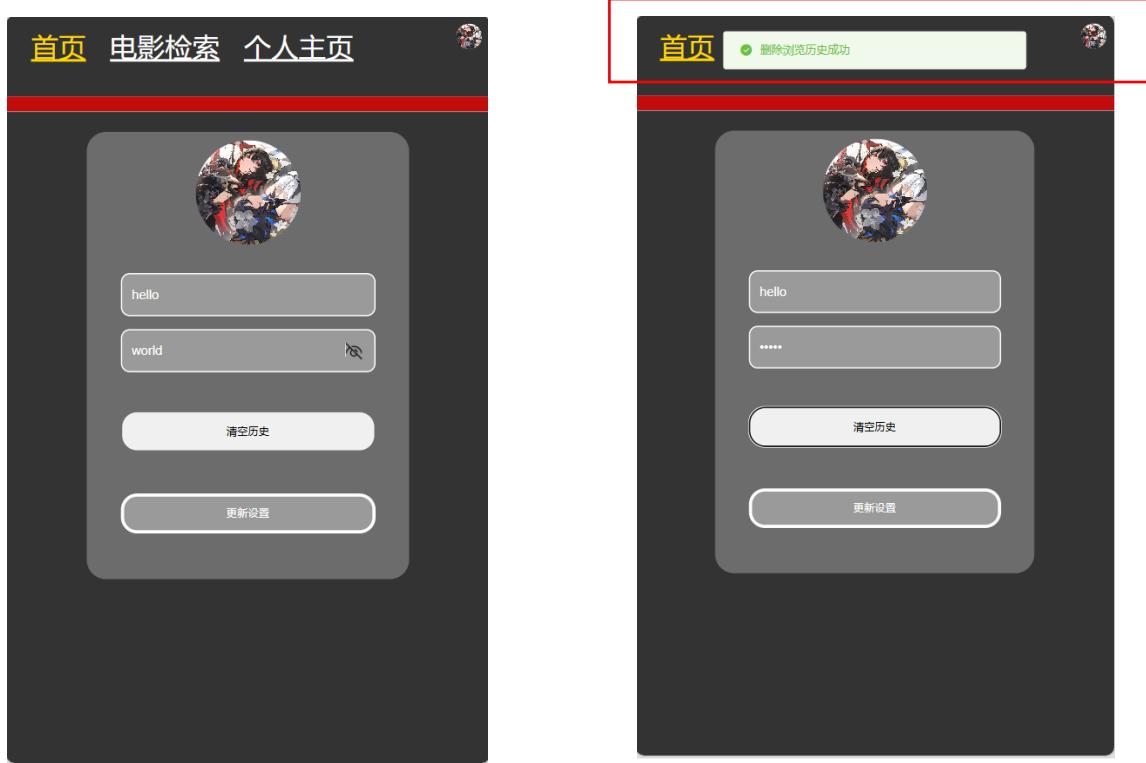
也可以和他人进行思想的碰撞

10.4 在个人界面，你能够知道你对这个平台的使用情况。你可以在个人界面设置一些标签，来反映自己的喜好，也能够查看自己的帖子，收藏的帖子和看过的帖子。



A grid-based interface showing user activity. It consists of several cards arranged in a grid. The columns are labeled "我的帖子" (My Posts), "我的收藏" (My Collections), and "浏览记录" (Browsing History). Each card contains a thumbnail image, a title, a rating (e.g., ★★★★★), and a brief description. For example, one card for "youdianla" shows a koala illustration and the text "youdianla ★★★★★ buhuoshuo". Another card for "123" shows a bird illustration and the text "123 ★★★★★ 本以为《幽默侠：平行宇宙》已经是一绝, 没想到续作更是令人称奇, 是天花板, 是艺术。它提升了整个差不多不同的幽默技巧的艺术性。" Other cards show similar content related to posts, collections, and browsing history.

10.4 最后，用户可以在设置界面修改用户名和密码，也能够清除浏览记录。



十一、清单 [王馨远]

这部分列出项目提交的清单，如：

master 分支

- docker 部署代码：docs 目录
- 前端代码（Vue 项目）：docs/code/ 目录
- 后端代码（demo 项目）：docs/code/ 目录
- 数据库文件（film_forum.sql）：docs/code 目录

main 分支

- 原型设计文件：docs/design 目录
- 软件架构文件：docs/软件架构 目录
- API 接口文档.pdf：docs 目录
- 原型设计文件：docs/design 目录

十二、总结 [王馨远]

本次电影论坛 Web 开发项目，是我们团队的一次重要实践，我们采用了 SpringBoot、Vue、MySQL 和 Docker 等技术栈，构建了一个功能完备、性能稳定的电影帖子用户平台。项目主要包括三大模块：用户管理、帖子发布与评论、电影信息展示。通过这个项目，我们不仅深入理解了前后端分离的架构模式，还掌握了微服务部署的相关技能，实现了从需求分析到系统设计，再到代码实现、测试与上线的全流程开发。

参与此次项目，我深刻体会到了团队协作的重要性。在项目初期，我们共同讨论需求，细化功能点，这让我学会了如何更有效地沟通和协调。在开发过程中，面对技术难题，我们相互支持，共同研究解决方案，这种团队精神让我受益匪浅。同时，通过实战，我对 SpringBoot、

Vue 等技术有了更深的理解和应用能力，特别是后端分层次调用封装方法以及前后端接口对接的调试经验增加了不少。在 Docker 容器化部署方面，我掌握了如何构建和运行 Docker 镜像，这对我未来的技术发展有着极大的帮助。

虽然项目已经成功上线，但我们的工作并未结束。未来，我们计划对项目进行持续优化，包括但不限于：

性能优化: 进一步提升系统的响应速度和并发处理能力，优化数据库查询效率，减少延迟。

用户体验: 参考竞品设计，调整界面布局，增加更多个性化设置选项，提高用户使用体验。

功能扩展: 优化推荐系统，基于用户的浏览历史等数据分析，改善个性化的电影推荐算法。

安全加固: 加强系统的安全性，引入更严格的用户认证机制，防止 SQL 注入等常见攻击。

持续集成/持续部署(CI/CD): 优化自动化测试和部署流程，提高开发效率和软件质量。

十二、参考文献 [王馨远]

系统所参考的文献或者代码，比如：

- element-ui: Element – 网站快速成型工具