摘 要

互联网作为人类伟大的发明，越来越融入人们的生活当中。人类的衣食住行都可以通过互联网来寻找解决方案。随着网络传输速度的加快，对提供互联网服务的基础硬件和软件的要求也越来越高。稳定而高效的基础服务是构成现代互联网的基石。而对于大量的数据，如果不采取行之有效的方式来处理和保存将会使基于这些数据的运行的软件响应缓慢。

本文是基于内存映射文件机制，利用Java语言进行开发：首先确定数据结构模型，而为了减少存储，算法只取实现所需功能的最少数据；然后实现按模式匹配查询、按照ID的精准查询；实现数据导入和按ID整理数据，最后使用Spring Boot构建对外服务框架。

实验结果表明，基于内存映射文件的存储相较于使用数据库实现该功能在响应速度上有大幅度的提高，同时也减小了数据库服务器的I/O压力。

该系统可以让需要同样场景的其他服务减少查询时间，增加了服务器的QPS，大大节省了服务器I/O消耗，同时避免了对用户不友好的较长等待时间。

关键词： 内存映射文件； 模式匹配； 对外服务

**Abstract**

As the great invention of mankind, the Internet is more and more integrated into people's lives. Human beings can find solutions through the Internet. With the increment of the speed of network transmission, the requirements for the basic hardware and software for providing Internet services are also increasing. The stable and efficient basic services are the cornerstone of the modern Internet. For a large amount of data, the lack of effective methods to process and save will cause the software which based on the running of these data respond slowly to the request.

This system is based on the memory mapping file mechanism, using Java language for development: first determine the data structure model, and in order to reduce storage, the algorithm only takes the minimum data to achieve the required functions; then implement the pattern matching query, according to the ID of the precise query; Data import and data sorting by ID, and finally use Spring Boot to build an external service framework.

The experimental results show that the memory-mapped file-based storage has a significant improvement in response speed compared to the use of the database, and it also reduces the I/O pressure of the database server.

The system can reduce the query time for other services that require the same scenario, increase the QPS of the server, greatly save the server I / O consumption, and avoid the long waiting time unfriendly to the user.

**Key words**： memory mapped file; pattern matching; external service

目 录

[1 绪论 1](#_Toc10471487)

[1.1 研究背景 1](#_Toc10471488)

[1.2 研究的主要内容 1](#_Toc10471489)

[1.3 相关技术 2](#_Toc10471490)

[1.4 论文结构 2](#_Toc10471491)

[2 系统概述 3](#_Toc10471492)

[2.1 需求概要 3](#_Toc10471493)

[2.2 技术选型 3](#_Toc10471494)

[2.3 系统功能架构 4](#_Toc10471495)

[2.4 开发环境 4](#_Toc10471496)

[2.5 本章小结 4](#_Toc10471497)

[3 算法和数据结构 5](#_Toc10471498)

[3.1 每条](#_Toc10471499)[数据的存储结构 5](#_Toc10471499)

[3.2 数据在文件中的存储分区 5](#_Toc10471500)

[3.3 算法 6](#_Toc10471501)

[3.3.1 字符串匹配 6](#_Toc10471502)

[3.3.2 无序数据与有序数据合并 7](#_Toc10471503)

[3.3.3 满足多实例部署的分布式读写锁 7](#_Toc10471504)

[3.4 本章小](#_Toc10471505)[结 10](#_Toc10471505)

[4 系统实现 11](#_Toc10471506)

[4.1 数据导入 11](#_Toc10471507)

[4.2 数据序列化与反序列化 12](#_Toc10471508)

[4.3 TEMP区数据的排序与归并 14](#_Toc10471509)

[4.4 数据定时更新 17](#_Toc10471510)

[4.5 数据查找 20](#_Toc10471511)

[4.6 数据接收与返回 22](#_Toc10471512)

[4.7 本章小结 24](#_Toc10471513)

[5 优化及实验效果分析 25](#_Toc10471514)

[5.1 搜索线程优化 25](#_Toc10471515)

[5.2 字符串匹配算法优化 27](#_Toc10471516)

[5.3 本章小结 28](#_Toc10471517)

[总结 29](#_Toc10471518)

[参考文献 30](#_Toc10471519)

[致谢 31](#_Toc10471520)

# 1 绪论

## 1.1 系统开发背景

近年来，随着互联网和移动技术的快速发展，电子商务与社交媒体的边界日益模糊，两者的融合已成为数字经济发展的重要趋势。特别是在后疫情时代，全球线上消费习惯得到显著强化，催生了社交电商这一新兴商业模式的蓬勃发展。与此同时，随着跨境电商政策的不断完善，越来越多的个人创业者开始寻求建立独立站的方式，摆脱大型平台的限制与高额佣金，构建自主可控的线上商业生态。

在服装零售领域，这种转变尤为明显。时尚穿搭作为年轻人自我表达的重要方式，已不再局限于简单的商品交易，而是演变为包含内容创作、社区互动和个性化推荐的综合体验。传统电商平台往往注重交易环节，而忽视了用户在穿搭分享、时尚交流方面的需求，这为独立的时尚穿搭社区平台创造了市场空间。同时，对于个体服装商户而言，拥有一个既能展示商品、又能建立用户社区的轻量级独立站，成为提升品牌价值和客户忠诚度的重要手段。

在技术层面，前后端分离架构的成熟与各类开源框架的普及，大幅降低了构建此类系统的技术门槛和开发成本，使得个人也能开发出性能稳定、体验良好的社区电商平台。这些因素共同促成了时尚穿搭社区独立站系统的研究需求。

## 1.2 系统意义

时尚穿搭分享与购物在线社区系统的研究具有理论和实践意义。从理论角度看，该系统探索了社交媒体与电子商务融合的新模式，丰富了社交电商的理论体系。通过用户在时尚穿搭社区中的行为模式和交互习惯，可以加强对消费者心理和决策机制的理解，为数字经济下的消费行为研究提供新的案例与数据支持。

从实践角度看，本系统为个体服装商户提供了一种切实可行的线上经营方案。与依赖大型电商平台不同，独立站模式使商户能够建立自主品牌，降低运营成本，提高利润率，并获得更为直接的用户反馈和市场洞察。对于创业者而言，这种轻量级系统大大降低了初创阶段的技术投入和维护成本，使其能够将更多资源集中于产品质量和内容创作。

同时，对消费者而言，此类系统提供了更为垂直和专业的时尚社区体验，满足了他们在穿搭分享、时尚交流方面的社交需求，同时通过社区信任为购买决策提供更可靠的参考。这种"社区+商城"的模式有效缩短了消费决策链路，提升了用户满意度和复购率。

此外，本系统的技术实践为前后端分离架构在轻量级应用场景中的优化提供了参考，特别是在用户体验优化、性能提升方面的，可为类似系统的开发提供借鉴。

## 1.4 系统主要内容

本系统主要围绕时尚穿搭分享与购物在线社区的设计与实现展开，核心内容包括四个方面。第一，对时尚穿搭社区的用户需求进行分析，明确系统应具备的穿搭分享、商品展示与购买、社区互动等基本功能，以及针对不同用户角色（普通用户、管理员）的差异化需求，形成系统功能模块的整体规划，并且在管理端应具备可视化数据展示，用来对用户行为和偏好进行分析。

第二，基于前后端分离架构设计系统技术框架，前端采用Vue2和Element UI构建用户界面，后端使用Spring Boot提供API服务，通过合理的数据结构和缓存机制优化系统性能和用户体验，解决轻量化独立站面临的资源限制问题。

第三，实现系统关键功能模块，包括用户管理与权限控制、穿搭分享与互动、商品展示与购买、社区帖子与评论、数据统计与可视化等，并针对移动端和PC端进行响应式适配，确保良好的跨设备使用体验。

最后，对系统关键模块进行性能和用户体验测试，通过实际场景验证系统的稳定性和可用性，总结系统优缺点，并提出未来可能的优化方向，如个性化推荐算法、社区活跃度提升策略等。

## 1.4 相关技术

本系统采用前后端分离的开发架构，融合了多种现代Web开发技术。在前端开发方面，系统以Vue.js 2.x作为核心框架，利用其组件化开发模式和响应式数据绑定特性，构建了高效的单页应用(SPA)。结合Element UI组件库，实现了美观统一的用户界面和良好的交互体验。Vue Router负责前端路由管理，通过路由守卫机制实现了基于用户角色的访问控制和权限验证。为优化前端性能，系统采用了懒加载技术，使页面只在需要时才加载相关组件，显著减少了初始加载时间。

在前后端通信方面，系统使用Axios库进行HTTP请求，并通过封装统一的请求拦截器，实现了请求头中Token的自动添加，以及响应状态的统一处理。针对频繁访问的用户信息，设计了内存缓存与本地存储相结合的多级缓存机制，减少不必要的数据传输和存储操作。

在后端技术选择上，系统基于Spring Boot框架开发，利用其自动配置和丰富的生态系统，构建了RESTful API服务。数据持久层采用MyBatis与MySQL的组合，通过精心设计的数据模型，支持了系统所需的各类复杂查询。系统还整合了Redis缓存，用于存储会话信息和热点数据，有效提升了系统响应速度。

## 1.5 论文结构

论文围绕基于Springboot+Vue的时尚穿搭分享与购物在线社区展开主要分为六章：

第一章介绍了本系统的制作背景和意义以及大致功能与技术栈。

第二章将进行系统概述，介绍系统的需求分析设计以及功能架构。

第三章主要介绍数据库表的设计以及其联系。

第四章为针对核心模块的关键代码和流程进行介绍及具体实现。

第五章对系统进行测试并展望后续可改进方向。

# 2 系统概述

## 2.1 需求概要

通过调研网上同类系统功能和结合周边同学的感受，要求此“时尚穿搭分享与购物在线社区”的核心功能为用户的分享，购物，以及针对用户行为的可视化展示。系统包括2种角色：用户和管理员。

用户端功能：

1. 登录注册功能。
2. 首页：查看穿搭分享、商品信息。
3. 穿搭分享：可以分页查看用户分享的穿搭，并可以按照条件筛选，点击查看可以进入此分享的详细页，可进行评论，查看其他热门穿搭，以及查看此穿搭关联的商品并跳转至该商品；

商城：可以分页查询商品，支持多条件筛选、排序查看，点击详情，查看购买用户发布的商品评价，支持立即购买，可以加入购物车；购买：用户可以选择多件商品加入购物车，加入后一起购买，支持购物车数量变更或移除购物车商品。

1. 社区：用户可以分享穿搭心得，可以最新查询（根据发布时间），也可以根据热门查看（点赞量高低），或者可以按照协调过滤算法推荐。同时有收藏榜、评论榜。进入帖子详情，用户可以点赞、评论。
2. 订单查看：查看订单信息，可以支付订单，或者收货
3. 其它管理：可以查看自己的评论、点赞、维护自己的穿搭分享和帖子信息。

管理员功能：

1. 各项管理功能：用户管理、分类管理、商品管理、订单管理、商品评论管理、穿搭信息管理、穿搭评论管理、帖子管理、帖子评论管理、收藏管理。
2. 可视化统计：针对用户分享穿搭的可视化统计、针对订单信息的可视化统计

## 2.2 技术选型

1. 总体架构选择

本系统采用前后端分离的架构模式，这种架构在现代Web应用开发中已成为主流选择。这一架构选择基于以下考量：

业务与界面分离：前后端分离架构使得业务逻辑处理与用户界面完全解耦，便于前后端的并行开发。

灵活性与可扩展性：独立的前端应用和后端服务使得系统各部分可以独立演进和扩展。

跨平台适应性：统一的后端API服务可以同时支持Web端、移动端等多种客户端。

开发效率：利用现代前端框架的组件化开发，提高代码复用率和开发效率。

2. 前端技术栈

选择Vue.js 2.x作为前端核心框架，主要基于以下因素：

组件化开发：组件化是Vue的核心特性，便于构建复杂界面及代码复用。

响应式数据绑定：Vue的响应式系统使得视图与数据自动同步，减少DOM操作。

虚拟DOM：提升渲染性能，特别在复杂列表和动态内容场景下。

生态系统完善：Vue相关工具链和社区支持完善，解决方案丰富。

选择Element UI作为UI组件库，考虑如下：

组件丰富：提供了包括表单、表格、弹窗等丰富组件，覆盖了系统所需的各类界面元素。设计一致性：基于统一设计规范，确保系统界面风格统一。适配性强：支持响应式设计，适应不同尺寸设备。定制化能力：支持主题定制，可根据系统风格进行调整。文档完善：详尽的API文档和示例降低了开发难度。

引入Vuex进行应用状态管理，原因包括：集中式状态管理：统一管理共享状态，避免组件间复杂的数据传递。状态变更可追踪：状态变更通过mutation进行，使得数据流向清晰可追踪。模块化设计：支持将状态分割成模块，便于大型应用状态管理。适合跨组件数据共享：如用户信息、购物车数据等全局状态管理。

采用Vue Router进行前端路由管理：客户端路由：实现单页应用体验，无需刷新页面即可切换视图。嵌套路由：支持路由嵌套，适合复杂页面布局。路由守卫：提供全局守卫机制，实现权限控制和登录状态检查。懒加载支持：结合动态导入实现组件懒加载，优化初始加载性能。

选择Axios作为HTTP客户端：请求/响应拦截：便于统一处理请求头（如添加Token）和响应状态。Promise API：基于Promise的API简化异步请求处理。请求取消：支持取消重复请求，优化用户体验。自动转换JSON：自动处理请求/响应数据格式转换。跨浏览器兼容性：良好的跨浏览器支持。

其他前端工具数据可视化：ECharts用于管理后台数据统计图表展示。富文本编辑：Vue-Quill-Editor支持内容创作和编辑。国际化支持：Vue-i18n为未来可能的多语言需求做准备。样式处理：Less预处理器增强CSS开发能力。

3. 后端技术栈

选择Spring Boot作为后端核心框架，考虑如下：自动配置：大幅减少配置工作，加速开发过程。内嵌服务器：内置Tomcat，简化部署流程。starter依赖：通过starter简化依赖管理。丰富生态：与Spring生态无缝集成，扩展性强。微服务友好：为未来可能的微服务架构演进做准备。

数据持久化选择MyBatis与MySQL组合，理由如下：SQL可控：MyBatis允许开发者完全控制SQL，适合复杂查询优化。灵活映射：灵活的对象-关系映射方式，适应不同的数据库表结构。轻量级设计：相比Hibernate更加轻量，学习曲线平缓。可靠性：MySQL作为成熟的关系型数据库，稳定性与性能均有保障。

整合Redis作为缓存解决方案，它的好处有：高性能：内存数据存储，响应速度快。多数据结构：支持字符串、哈希、列表等多种数据结构，适应不同缓存需求。会话存储：用于存储用户会话和临时数据。热点数据缓存：缓存频繁访问的数据，减轻数据库压力。原子操作：支持原子操作，适合高并发场景。

4. 开发与部署工具

前端构建：Vue CLI提供完整的项目脚手架和构建配置。

后端构建：Maven管理依赖及构建流程。

版本控制Git：分布式版本控制，支持回滚，方便开发，并留存每次更改痕迹，方便出现bug后回归查看。分支策略：采用master分支策略管理代码。

5. 技术选型总结

本系统的技术选型综合考虑了项目需求、性能要求和未来扩展性等因素。前端采用Vue.js生态系统构建单页应用，后端采用Spring Boot提供RESTful API服务，整体架构清晰且松耦合。保持了良好的可维护性和扩展性。

技术选型的关键优势在于：轻量化设计适合个体商户独立站需求；组件化架构便于功能扩展；缓存机制确保系统性能；路由守卫保障系统安全。这些选择使得系统既能满足当前的社区和电商功能需求，又为未来可能的需求变化预留了扩展空间。

## 2.3 系统功能架构

1.用户身份进入系统

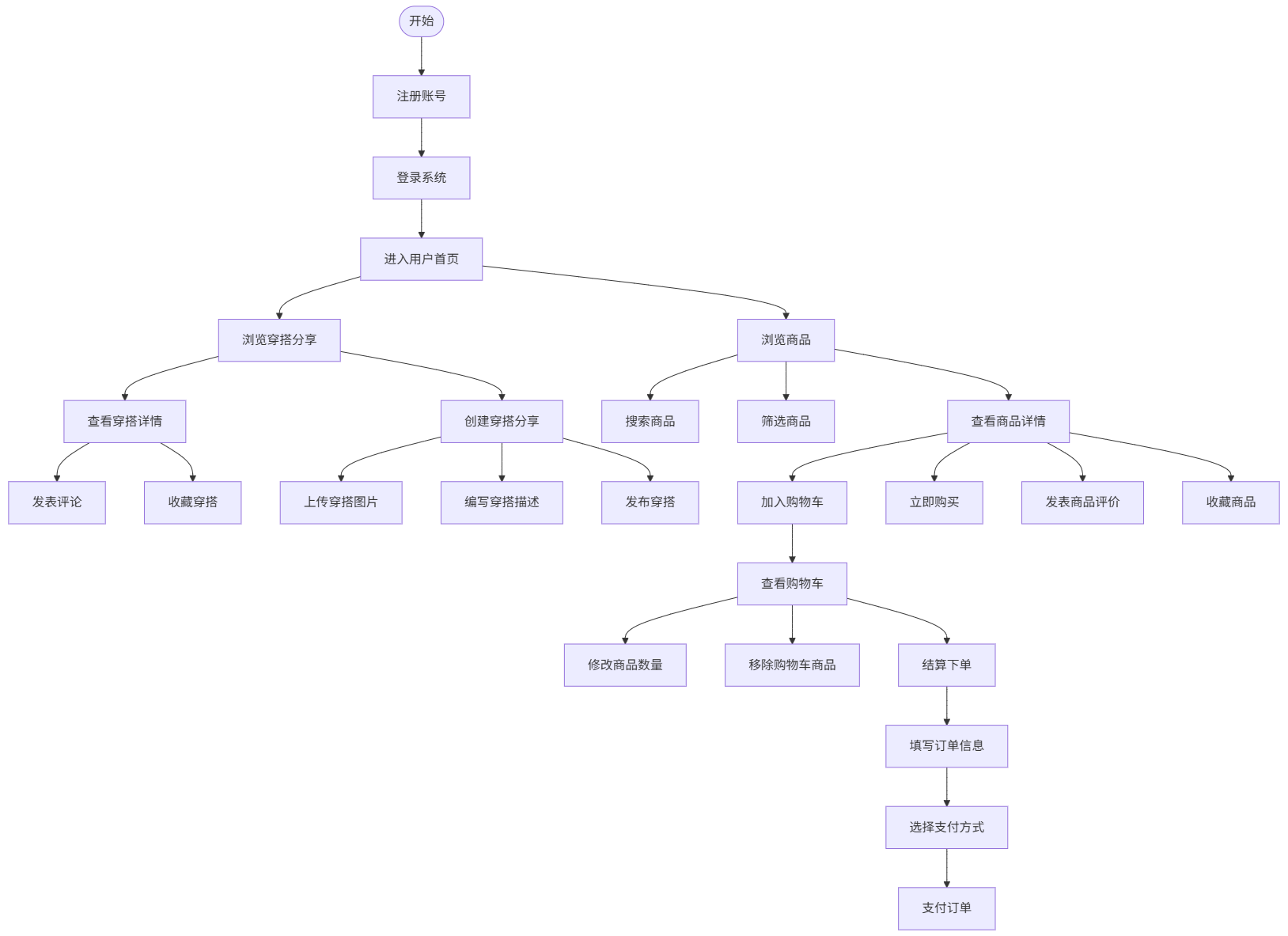


图2.1穿搭分享，购物相关功能

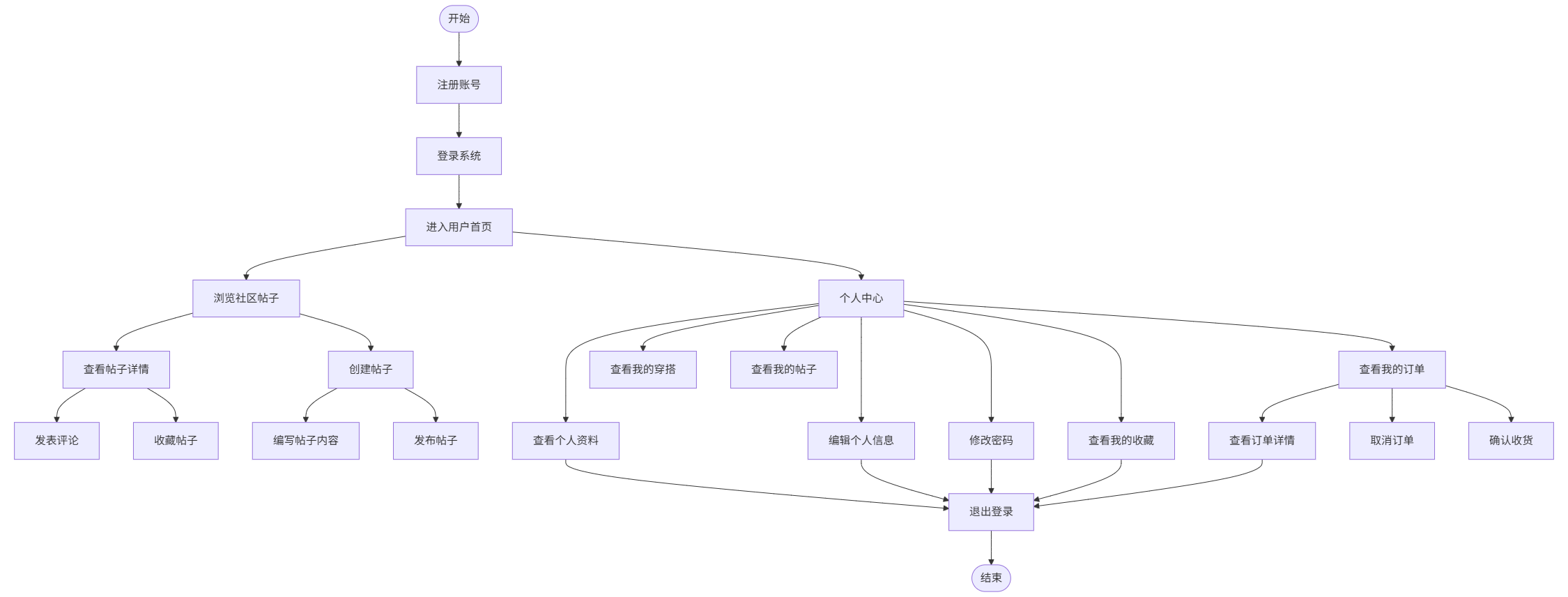


图2.2用户帖子，个人中心相关功能

2.管理员身份进入系统

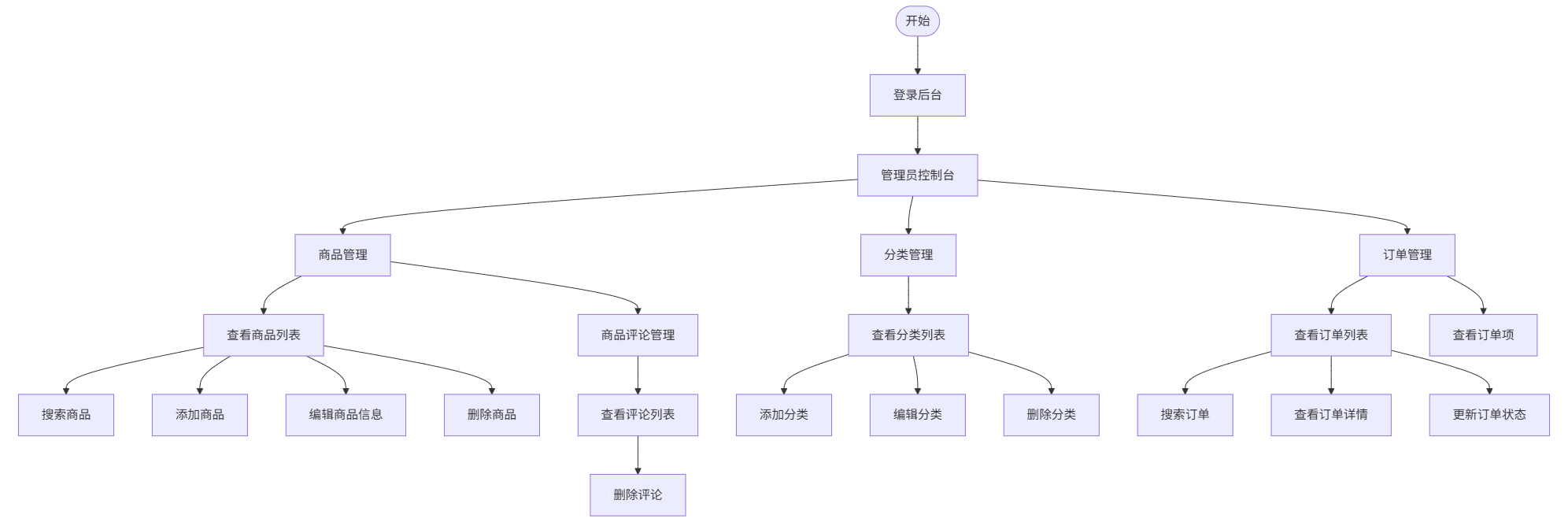


图2.3商品、分类、订单管理相关功能

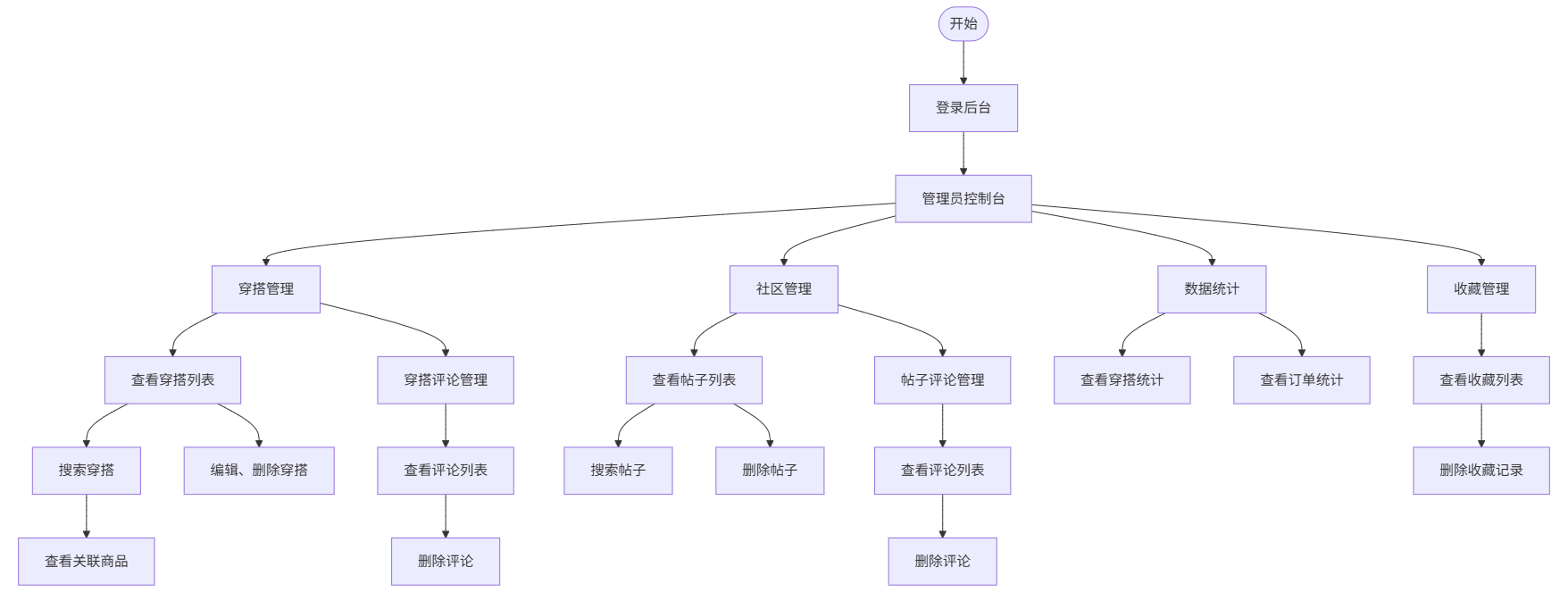


图2.4 穿搭、社区、收藏管理、数据统计相关功能

## 2.4 开发环境

服务器：Spring Boot内置服务器

开发模式：B/S模式

开发语言：Java、Vue

开发工具：IntelliJ IDEA

开发系统环境：windows11、jdk1.8

## 2.5 本章小结

本章为程序的需求、整体结构以及技术的选型做了展示，详细描述了程序的需求概述、技术选型、整体功能，为后面程序细节的设计和实现做了一个可预见的蓝图。

# 3 数据库设计

## 3.1 表结构设计

表3.1 admin(管理员表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| image | varchar(100) | 是 | 否 | 否 | — | 头像 |
| username | varchar(20) | 是 | 否 | 否 | — | 用户名 |
| password | varchar(20) | 是 | 否 | 否 | — | 密码 |
| create\_time | datetime | 是 | 否 | 否 | — | 创建时间 |

表3.2 cart(购物车)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| gid | int(11) | 是 | 否 | 否 | — | 商品 |
| num | int(11) | 是 | 否 | 否 | — | 数量 |
| uid | int(11) | 是 | 否 | 否 | — | 用户 |
| create\_time | datetime | 是 | 否 | 否 | — | 创建时间 |

表3.3 category(分类表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| name | varchar(50) | 是 | 否 | 否 | — | 分类名称 |
| remark | varchar(100) | 是 | 否 | 否 | — | 描述 |
| create\_time | datetime | 是 | 否 | 否 | — | 发布时间 |

表3.4 comment(帖子评论表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| fid | int(11) | 是 | 否 | 否 | — | 帖子 |
| uid | int(11) | 是 | 否 | 否 | — | 评论用户 |
| content | varchar(500) | 是 | 否 | 否 | — | 评论内容 |
| reply | varchar(500) | 是 | 否 | 否 | — | 回复内容 |
| create\_time | datetime | 是 | 否 | 否 | — | 发布时间 |

表3.5 discuss(穿搭评论表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| oid | int(11) | 是 | 否 | 否 | — | 穿搭 |
| uid | int(11) | 是 | 否 | 否 | — | 用户 |
| content | varchar(500) | 是 | 否 | 否 | — | 评论内容 |
| reply | varchar(500) | 是 | 否 | 否 | — | 回复内容 |
| create\_time | datetime | 是 | 否 | 否 | — | 评论时间 |

表3.6 forum(帖子表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| title | varchar(100) | 是 | 否 | 否 | — | 标题 |
| content | varchar(2000) | 是 | 否 | 否 | — | 内容 |
| uid | int(11) | 是 | 否 | 否 | — | 发布人 |
| create\_time | datetime | 是 | 否 | 否 | — | 发布时间 |

表3.7 goods(商品表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| cid | int(11) | 是 | 否 | 否 | — | 分类 |
| image | varchar(100) | 是 | 否 | 否 | — | 商品图片 |
| name | varchar(30) | 是 | 否 | 否 | — | 名称 |
| remark | varchar(30) | 是 | 否 | 否 | — | 简介 |
| money | varchar(30) | 是 | 否 | 否 | — | 价格 |
| num | int(11) | 是 | 否 | 否 | — | 库存 |
| content | varchar(2000) | 是 | 否 | 否 | — | 商品介绍 |
| status | varchar(10) | 是 | 否 | 否 | — | 状态 |
| create\_time | datetime | 是 | 否 | 否 | — | 发布时间 |

表3.8 order\_item(订单项表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| oid | int(11) | 是 | 否 | 否 | — | 订单 |
| gid | int(11) | 是 | 否 | 否 | — | 商品 |
| num | int(11) | 是 | 否 | 否 | — | 数量 |
| money | varchar(20) | 是 | 否 | 否 | — | 金额 |
| create\_time | datetime | 是 | 否 | 否 | — | 创建时间 |

表3.9 orders(订单表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| no | varchar(50) | 是 | 否 | 否 | — | 订单编号 |
| num | int(11) | 是 | 否 | 否 | — | 商品数量 |
| total | varchar(50) | 是 | 否 | 否 | — | 总价 |
| uid | int(11) | 是 | 否 | 否 | — | 用户 |
| remark | varchar(200) | 是 | 否 | 否 | — | 备注 |
| status | varchar(20) | 是 | 否 | 否 | — | 状态 |
| create\_time | datetime | 是 | 否 | 否 | — | 下单时间 |

表3.10 outfit(穿搭信息表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| image | varchar(100) | 是 | 否 | 否 | — | 图片 |
| type | varchar(20) | 是 | 否 | 否 | — | 类型 |
| season | varchar(20) | 是 | 否 | 否 | — | 季节 |
| name | varchar(50) | 是 | 否 | 否 | — | 名称 |
| content | text | 是 | 否 | 否 | — | 内容 |
| uid | int(11) | 是 | 否 | 否 | — | 发布人 |
| num | int(11) | 是 | 否 | 否 | — | 浏览量 |
| status | varchar(10) | 是 | 否 | 否 | — | 状态 |
| create\_time | datetime | 是 | 否 | 否 | — | 发布时间 |
| productIds | varchar(100) | 是 | 否 | 否 | — | 关联商品id |

表3.11 review(商品评论表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| gid | int(11) | 是 | 否 | 否 | — | 商品 |
| uid | int(11) | 是 | 否 | 否 | — | 评论用户 |
| content | varchar(500) | 是 | 否 | 否 | — | 评论内容 |
| create\_time | datetime | 是 | 否 | 否 | — | 发布时间 |

表3.12 stars(收藏表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| fid | int(11) | 是 | 否 | 否 | — | 帖子 |
| uid | int(11) | 是 | 否 | 否 | — | 用户 |
| create\_time | datetime | 是 | 否 | 否 | — | 收藏时间 |

表3.13 user(用户表)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 列名 | 数据类型 | 是否为空 | 主键 | 自增 | 默认值 | 备注 |
| id | int(11) | 否 | 是 | 是 | — | ID |
| image | varchar(100) | 是 | 否 | 否 | — | 头像 |
| phone | varchar(20) | 是 | 否 | 否 | — | 手机号 |
| password | varchar(20) | 是 | 否 | 否 | — | 密码 |
| realname | varchar(50) | 是 | 否 | 否 | — | 姓名 |
| sex | varchar(20) | 是 | 否 | 否 | — | 性别 |
| age | varchar(20) | 是 | 否 | 否 | — | 年龄 |
| address | varchar(100) | 是 | 否 | 否 | — | 地址 |
| create\_time | datetime | 是 | 否 | 否 | — | 注册时间 |

## 3.2 数据库表ER图

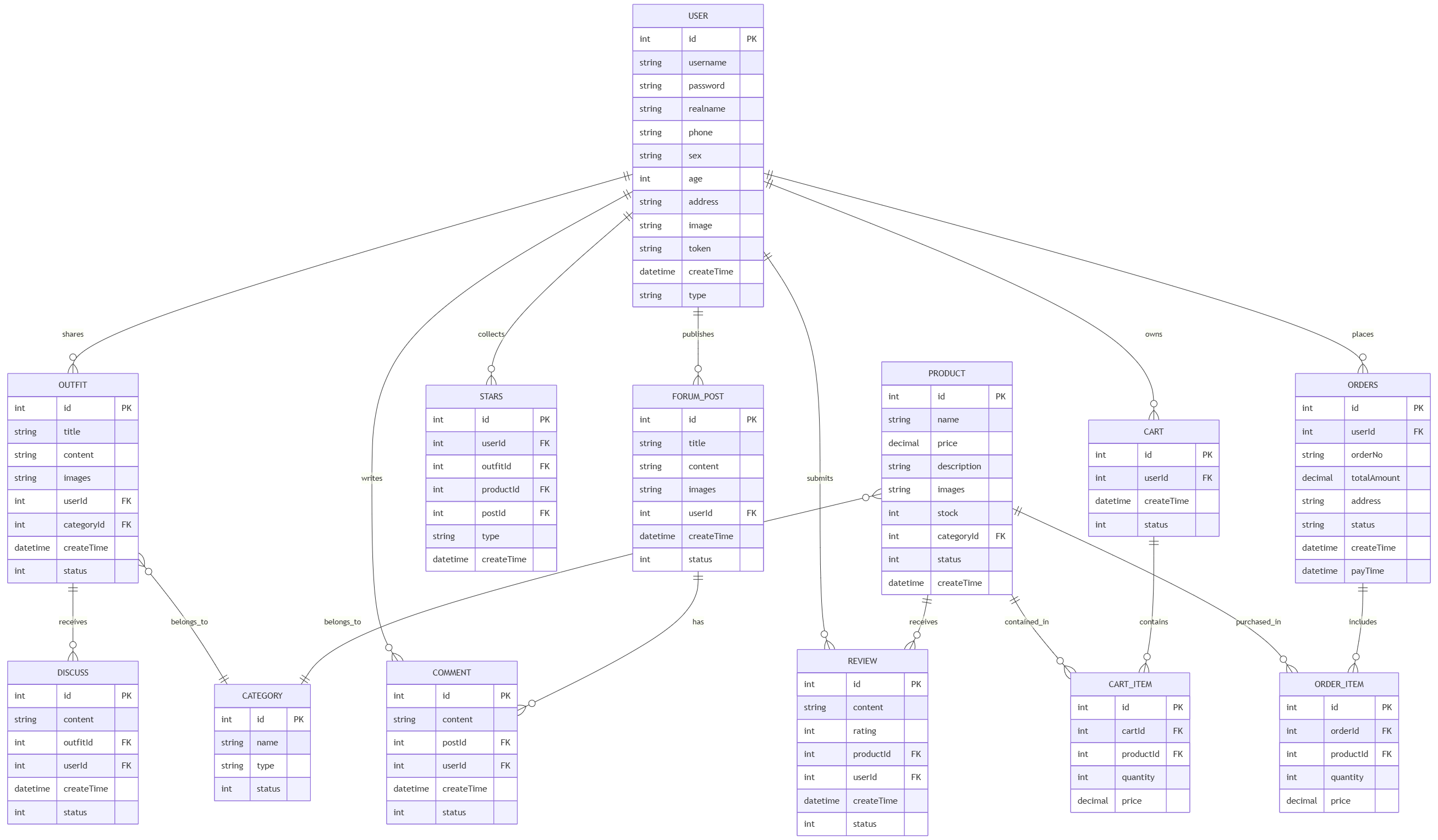


图3.1 表ER图

## 3.3 表设计依据

### 3.3.1 保障数据一致性与完整性

1 实体唯一性与主键策略

所有表采用自增整数主键（如id字段），符合关系型数据库实体完整性规则。主键作为外键构建表间关联（如cart.uid关联user.id），避免基础数据（如用户、商品信息）重复存储，减少数据冗余。例如，用户表（user）与购物车表（cart）通过uid关联，确保用户操作记录可追溯，同时避免用户头像、手机号等信息在购物车表中重复存储。

2 范式化设计消除依赖

第一范式（1NF）：所有字段均为原子性数据项（如username不可再分），确保单表数据项独立，满足 “数据项不可分割” 的基础规范。

第二范式（2NF）：消除部分依赖，如订单项表（order\_item）分离订单与商品详情，通过oid（订单 ID）和gid（商品 ID）关联主表，避免订单总价（orders.total）与商品价格（goods.money）在同一表中重复存储，确保 “一单多品” 场景下的数据一致性。

第三范式（3NF）：消除传递依赖，如商品表（goods）通过分类 ID（cid）关联分类表（category），而非直接存储分类名称，当分类名称变更时，仅需更新category表，避免多表联动修改，提升数据维护效率。

### 3.3.2 模块化设计适配业务场景

1 角色分离与权限管理

管理员表（admin）与用户表（user）独立设计，区分两类角色的属性差异（如管理员无需手机号、地址字段），符合 RBAC（角色基于访问控制）模型。该设计便于后续权限控制扩展，例如为管理员表新增role字段标识超级管理员 / 普通管理员，而不影响用户表结构，提升系统安全性与可维护性。

2 订单与订单项分层设计

订单主表（orders）存储订单整体信息（如订单编号、总价、用户 ID），订单项表（order\_item）存储单品详情（如商品 ID、数量、金额），满足电商核心场景 “一单多品” 的需求。此设计支持复杂业务逻辑，如计算单品折扣、库存扣减、订单拆分等，同时避免主表字段膨胀，提升订单查询与统计效率（如按用户 ID 统计历史订单时，仅需关联orders与order\_item表）。

3 评论系统分场景设计

按业务对象分离评论表（comment帖子评论、discuss穿搭评论、review商品评论），通过不同外键（fid帖子 ID、oid穿搭 ID、gid商品 ID）关联目标实体。该设计简化查询逻辑，例如 “查询某商品的所有评论” 仅需过滤review.gid，避免多场景评论混合存储导致的冗余查询；同时支持差异化功能扩展，如为帖子评论表新增reply\_to字段记录回复目标，而不影响其他评论表结构。

### 3.3.3 时间维度与状态机设计增强业务追溯性

1 时间戳字段的审计与分析价值  
所有表包含create\_time字段记录数据创建时间，实现两大核心功能：

操作审计：追踪用户行为时间（如下单时间、评论发布时间），为系统异常排查（如库存异常扣减）提供时间维度证据链；

数据分析：支持根据时间序列的统计，为运营决策提供数据支撑。

2 状态字段的流程控制作用

status字段（如goods.status、orders.status）采用字符串枚举值（如 “在售”“下架”“已支付”“已取消”），明确标识业务实体状态。该设计将业务流程控制逻辑固化到数据层，避免硬编码状态值，提升系统可维护性。例如，商品表通过status='在售'过滤有效商品，订单表通过status='已支付'筛选待发货订单，简化业务层逻辑判断，降低代码复杂度。

## 3.4 扩展性设计应对未来需求变化

1 预留灵活字段与松耦合结构

remark（描述）、reply（回复内容）等字段采用varchar或text类型，支持动态扩展业务描述（如分类备注、评论回复），避免因需求变更导致的表结构频繁修改；

outfit.productIds字段以字符串形式存储关联商品 ID（如 “1,3,5”），实现 “穿搭 - 商品” 多对多关系的轻量化存储。在初期业务规模较小时，该设计可简化开发（无需创建中间表），后续若需支持复杂关联查询（如 “查询包含某商品的所有穿搭”），可通过新增中间表实现平滑过渡，兼顾开发效率与架构扩展性。

2 类型化字段的兼容性设计

对性别（sex）、季节（season）等固定取值字段，采用varchar而非数值枚举，便于后续新增取值（如性别字段新增 “其他”，季节字段新增 “雨季”），避免因数据库枚举类型限制导致的兼容性问题，提升系统对业务变化的适应性。

## 3.5 本章小结

本章内容为对程序中所涉及的数据库表结构，及其之间关系和其设计的相关概述，下一章中的内容将会基于上述章节进行核心功能实现的介绍。

# 4 系统核心功能实现

### 4.1 前端整体流程

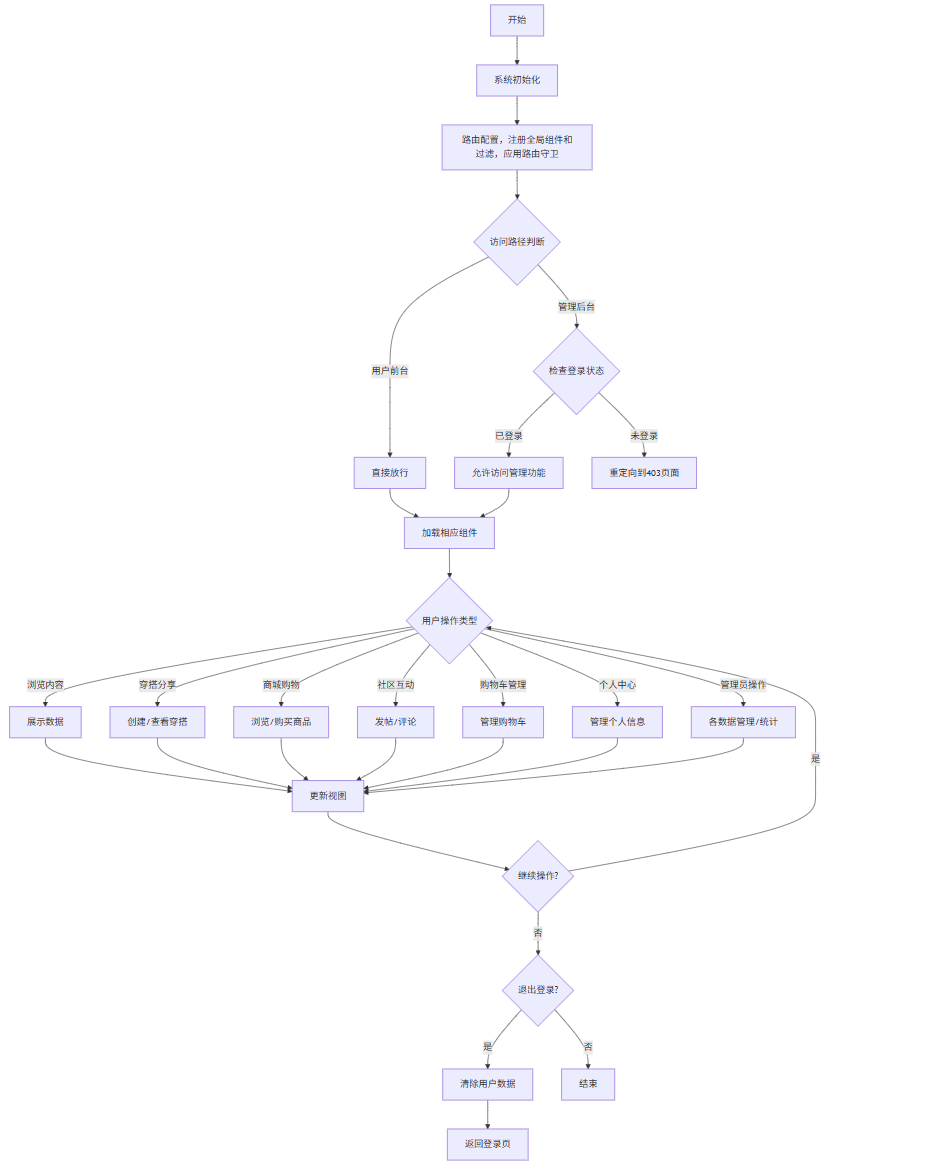


图4.1前端整体流程图

图4.1为前端整体流程示意图，清晰呈现了用户在前端系统中的操作逻辑与功能交互路径。流程始于用户访问前端界面，通过 “注册 / 登录” 完成身份验证，确保合法进入系统。随后进入 “功能选择模块”，该核心节点依据用户选择，触发前端展示对应业务功能（如商品浏览、订单处理等）。前端通过与后端实时交互获取数据，支持用户执行浏览商品详情、购买商品、提交订单，社区互动、发帖等操作，最终以用户完成操作后 “结束” 或 “退出系统” 为终点。

## 4.2 前端双端通用框架设计

### 4.2.1  关键组件分析

一. 管理端组件设计原理

1. Header组件

提供管理员专用的顶部导航栏，支持折叠侧边栏和用户操作。并通过el-button和el-dropdown实现折叠侧边栏和用户操作功能。

<template>

<div class="header-container">

<el-button @click="toggleCollapse">折叠侧边栏</el-button>

<el-dropdown @command="handleCommand">

<span class="el-dropdown-link">{{ userInfo.username }}</span>

<el-dropdown-menu slot="dropdown">

<el-dropdown-item command="profile">个人资料</el-dropdown-item>

<el-dropdown-item command="logout">退出登录</el-dropdown-item>

</el-dropdown-menu>

</el-dropdown>

</div>

</template>

2. Sidebar组件

该组件动态渲染管理员菜单，支持多级嵌套和路由跳转。

跳转实现：根据用户角色过滤菜单项，并通过router属性启用路由跳转。

<template>

<el-menu :collapse="collapse" :default-active="onRoutes" router>

<el-submenu v-for="item in items" :key="item.id" :index="item.path">

<template slot="title">{{ item.title }}</template>

<el-menu-item v-for="child in item.children" :key="child.id" :index="child.path">{{ child.title }}</el-menu-item>

</el-submenu>

</el-menu>

</template>

3. Tags.vue组件

动态生成标签列表，管理打开的页面标签，并支持关闭和跳转。

<template>

<div class="tags-container">

<el-tag v-for="tag in tagsList" :key="tag.path" closable @close="closeTags(tag)">

<router-link :to="tag.path">{{ tag.title }}</router-link>

</el-tag>

</div>

</template>

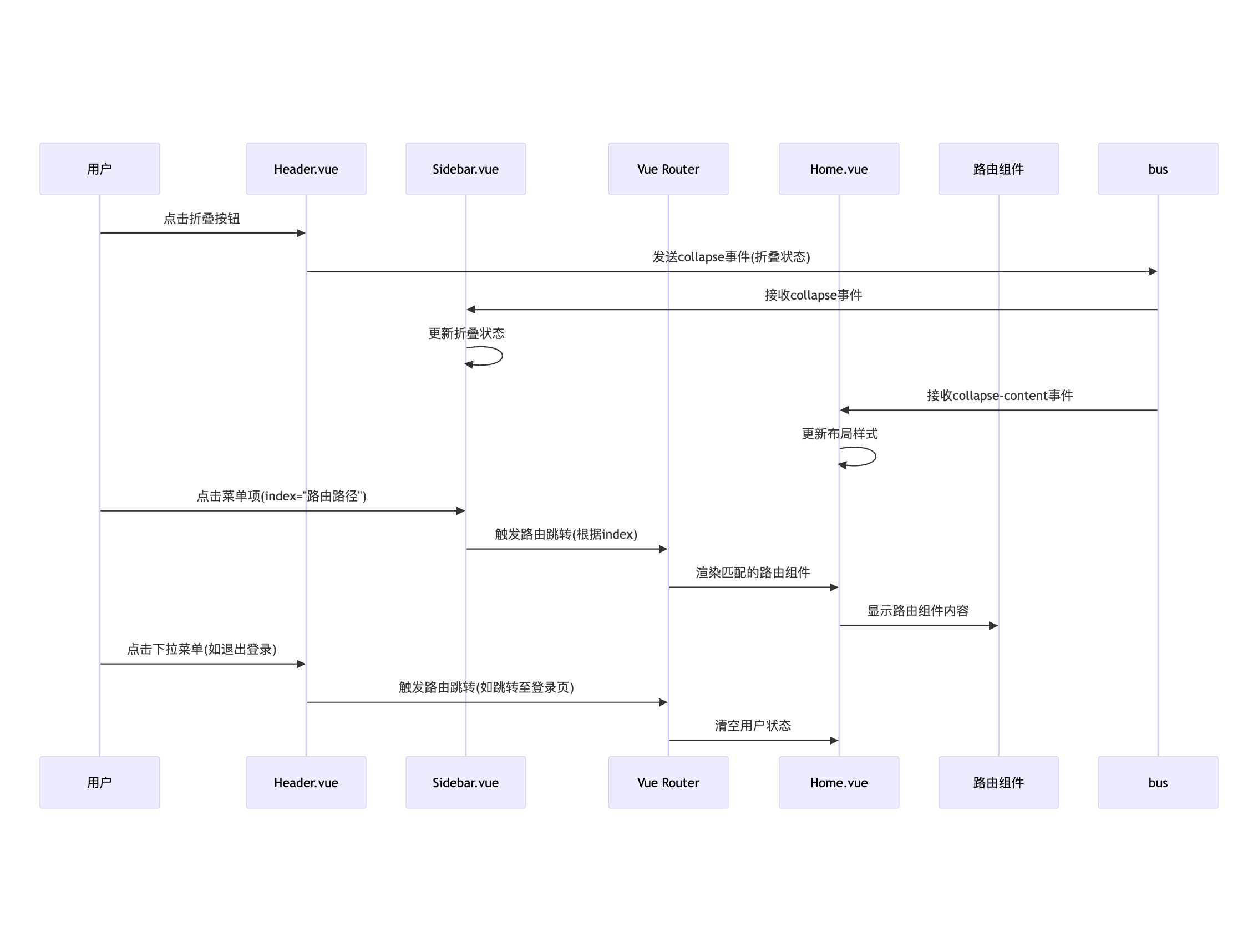


图4.2管理端通用框架端时序图

二、用户端组件设计原理

1. Header组件

为用户界面的顶部导航栏，包含Logo、导航菜单和用户功能区。

通过el-menu实现动态路由跳转。使用v-if和v-else根据登录状态切换显示内容。

<template>

<el-menu :default-active="activeIndex" mode="horizontal" @select="handleSelect">

<el-menu-item index="1">首页</el-menu-item>

<el-menu-item index="2">购物车</el-menu-item>

<el-menu-item v-if="!isLoggedIn" index="3">登录</el-menu-item>

<el-dropdown v-else>

<span class="el-dropdown-link">{{ userInfo.username }}</span>

<el-dropdown-menu slot="dropdown">

<el-dropdown-item @click.native="logout">退出登录</el-dropdown-item>

</el-dropdown-menu>

</el-dropdown>

</el-menu>

</template>

2. Home组件的布局优化

用户界面的布局组件，包裹顶部导航栏和动态内容区域。

程序固定显示Header组件，并通过router-view动态渲染路由匹配的组件。

<template>

<div class="app-container">

<v-head />

<router-view />

</div>

</template>

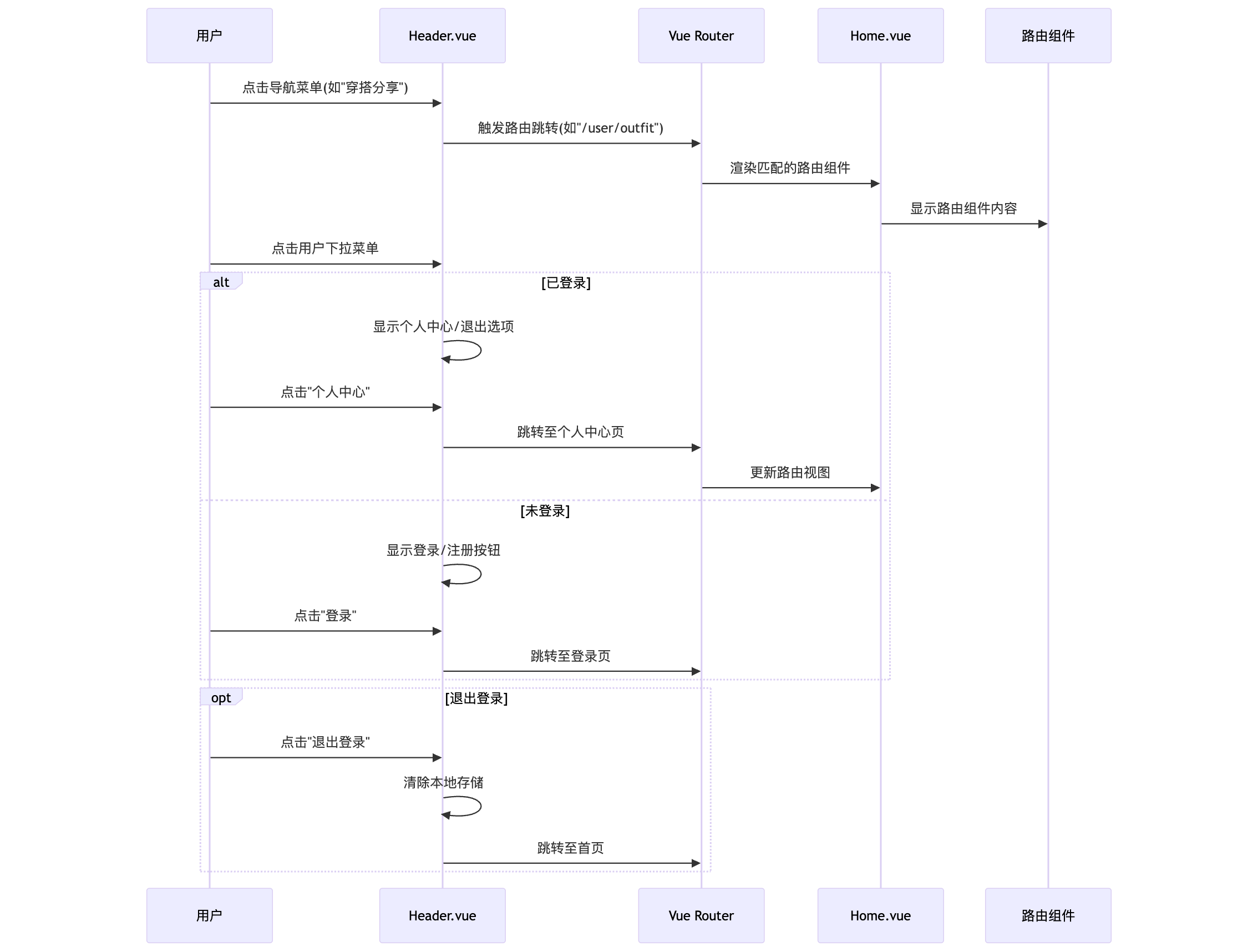


图4.3用户端通用框架端时序图

三、组件间通信机制研究

1. 事件总线（`bus.js`）

用于非父子组件之间的通信，例如Header.vue和Sidebar.vue之间的状态同步。

通过bus.$emit触发事件，bus.$on监听事件。

`Header.vue`：通知侧边栏折叠状态。

methods: {

toggleCollapse() {

this.collapse = !this.collapse;

bus.$emit('collapse-content', this.collapse);

}

}

`Sidebar.vue`：监听折叠事件并更新状态。

created() {

bus.$on('collapse-content', (collapse) => {

this.collapse = collapse;

});

}

2. 父子组件通信

通过props和$emit实现父子组件之间的数据传递和事件触发。

示例：

父组件（Home.vue）：

<template>

<child-component :message="parentMessage" @child-event="handleChildEvent" />

</template>

子组件（ChildComponent.vue）：

props: ['message'],

methods: {

emitEvent() {

this.$emit('child-event', 'data from child');

}

}

表4.1 组件其他的状态共享方案

| 方案 | 管理端 | 用户端 | 技术实现差异 |
| --- | --- | --- | --- |
| 核心状态库 | Vuex | Pinia | 架构设计差异 |
| 持久化策略 | localStorage | sessionStorage | 数据生命周期管理 |
| 实时同步机制 | WebSocket | Polling | 性能与实时性权衡 |

2. 路由级权限控制模型

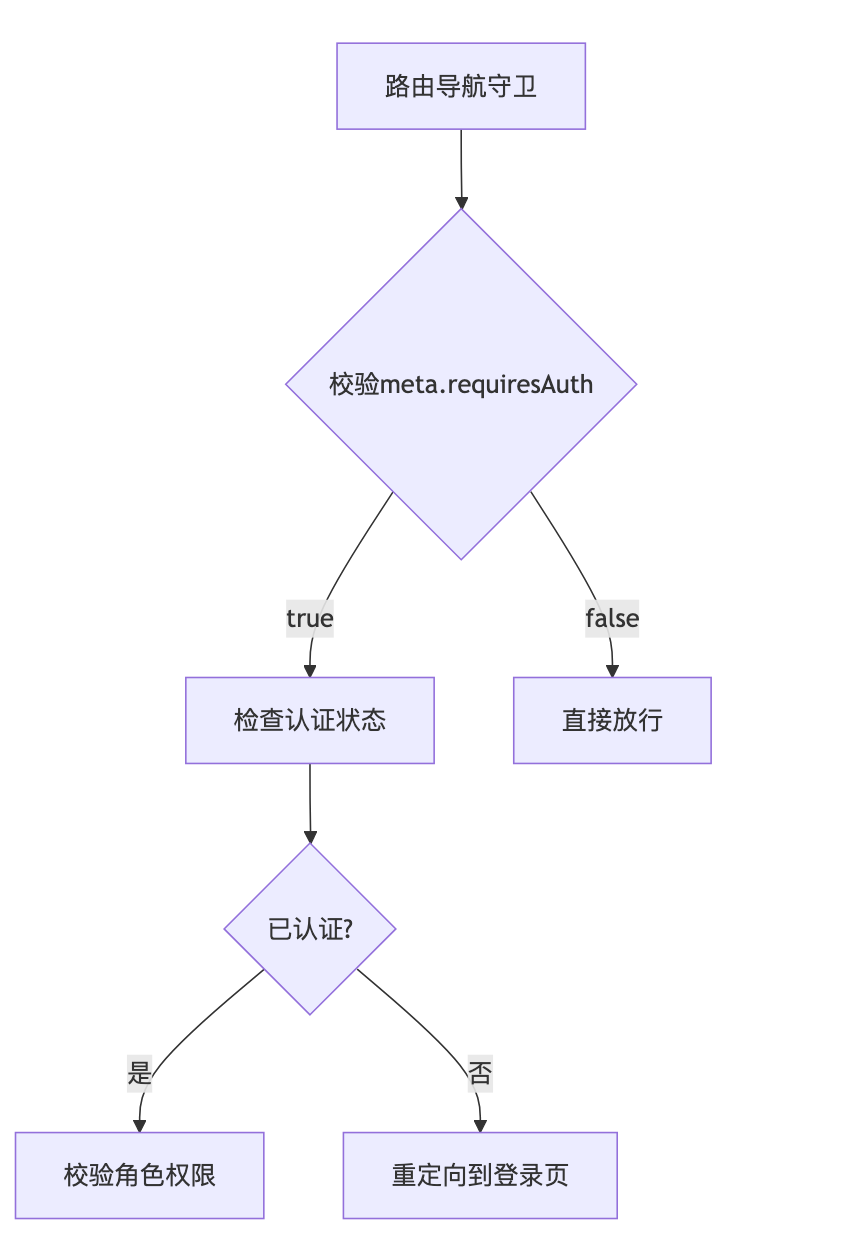


图4.4 权限控制

## 4.3 数据可视化统计展示

### 4.3.1 功能概述

本项目的可视化模块主要包括两个核心组件：OutfitStats.vue 和 OrdersStats.vue，分别用于展示穿搭统计数据和订单统计数据。

OutfitStats.vue：

该模块主要用于展示用户穿搭的统计分析数据，内容如下：

数据卡片：显示穿搭总数、总浏览量、评论总数和活跃用户数。

图表展示：通过多种图表类型（如饼图、柱状图、折线图）展示穿搭类型分布、季节偏好分布、每日穿搭分享趋势、热门穿搭排行榜等。

OrdersStats.vue：

该模块专注于订单数据的可视化分析，内容如下：

数据卡片：展示订单总数、已支付订单、待收货订单和总销售额。

图表展示：通过图表展示订单状态分布、每日订单量、销售额趋势、热门商品购买量排行等。

### 4.3.2 技术实现

1. 数据获取

通过 axios 调用后端 API 获取数据。

OutfitStats.vue 数据获取代码：

async fetchOutfitStats() {

try {

const response = await axios.get('/api/outfit/stats');

this.statsData = response.data;

this.initCharts(); // 数据获取后初始化图表

} catch (error) {

console.error('Failed to fetch outfit stats:', error);

}

}

2. 图表初始化与配置

使用 ECharts 初始化图表。

OrdersStats.vue 图表配置代码：

initOrderTrendChart() {

const chartDom = this.$refs.orderTrendChart;

const myChart = echarts.init(chartDom);

const option = {

title: { text: '每日订单量趋势' },

tooltip: { trigger: 'axis' },

xAxis: { type: 'category', data: this.orderTrendData.dates },

yAxis: { type: 'value' },

series: [{ data: this.orderTrendData.counts, type: 'line' }]

};

myChart.setOption(option);

window.addEventListener('resize', () => myChart.resize());

}

3. 动态更新

监听数据变化并更新图表。

OutfitStats.vue 动态更新逻辑：

watch: {

statsData: {

handler(newVal) {

if (newVal) {

this.updateCharts(); // 数据变化时更新图表

}

},

deep: true

}

}

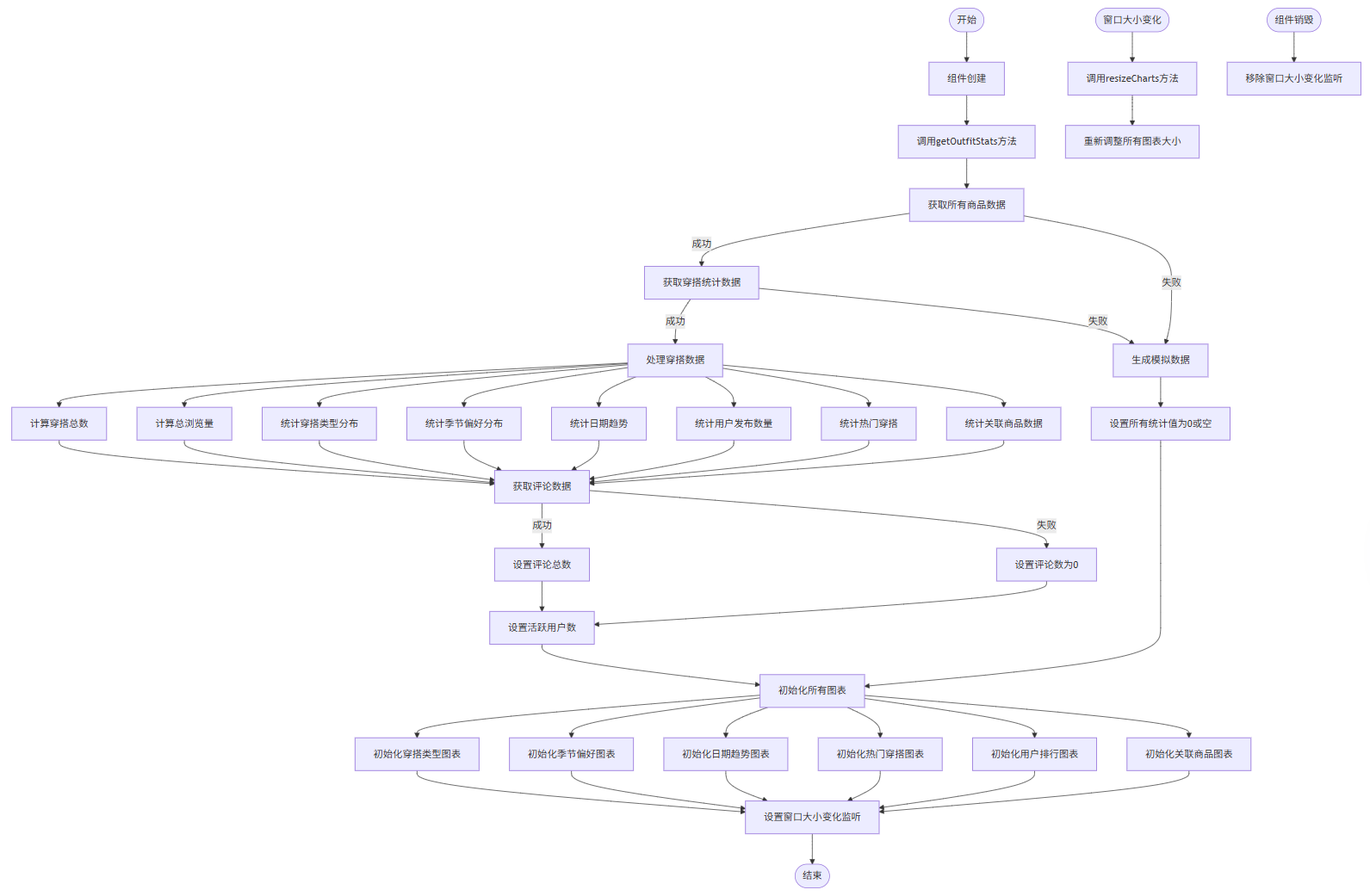


图4.5 穿搭分析整体可视化活动图

4. 实现举例介绍

以关联商品统计图的实现举例，关联商品统计图用于展示与穿搭关联的商品数据，包括：

关联次数：商品被穿搭引用的次数。

累计浏览量：通过关联穿搭带来的商品总浏览量。

商品价格：辅助信息，帮助管理员评估商品价值。

相关$echarts 通过 全局引入 ECharts 库后挂载到 Vue 实例上的 ECharts 对象。然后通过 id 为 typeChart 来获取 HTML 文档中的 DOM 元素，这个元素将作为 ECharts 图表的容器。最后再调用 ECharts 的 init 方法，传入 DOM 元素，初始化一个 ECharts 实例并赋值给 typeChart 变量。

数据获取与处理

在 getOutfitStats 方法中，通过以下逻辑统计关联商品数据：

// 统计关联商品数据

if (outfit.productIds && outfit.productIds !== '[]') {

try {

const productIds = JSON.parse(outfit.productIds);

const outfitViews = parseInt(outfit.num) || 0;

productIds.forEach(productId => {

// 统计商品被关联次数

productRelationMap[productId] = (productRelationMap[productId] || 0) + 1;

// 累计关联穿搭的浏览量

productViewsMap[productId] = (productViewsMap[productId] || 0) + outfitViews;

});

} catch (e) {

console.error('解析商品ID失败:', e);

}

}

在 initRelatedProductsChart 方法中，使用 ECharts 实现双柱状图：

initRelatedProductsChart() {

const relatedProductsChart this.$echarts.init(document.getElementById('relatedProductsChart'));

// 提取数据

const productNames = this.relatedProductsData.map(item => item.name);

const relationCounts = this.relatedProductsData.map(item => item.relationCount);

const viewsCounts = this.relatedProductsData.map(item => item.views);

const option = {

tooltip: {

trigger: 'axis',

axisPointer: { type: 'shadow' },

formatter: function(params) {

const item = params[0];

const dataIndex = item.dataIndex;

const product = this.relatedProductsData[dataIndex];

return `

<div>

<b>${product.name}</b><br/>

关联次数: ${product.relationCount}<br/>

累计浏览量: ${product.views}<br/>

商品价格: ¥${product.price}

</div>

`;

}.bind(this)

},

series: [

{

name: '关联次数',

type: 'bar',

data: relationCounts,

itemStyle: { color: '#f3c3cc' }

},

{

name: '累计浏览量',

type: 'bar',

data: viewsCounts,

itemStyle: { color: '#95ddb2' }

}

]

};

relatedProductsChart.setOption(option);

}

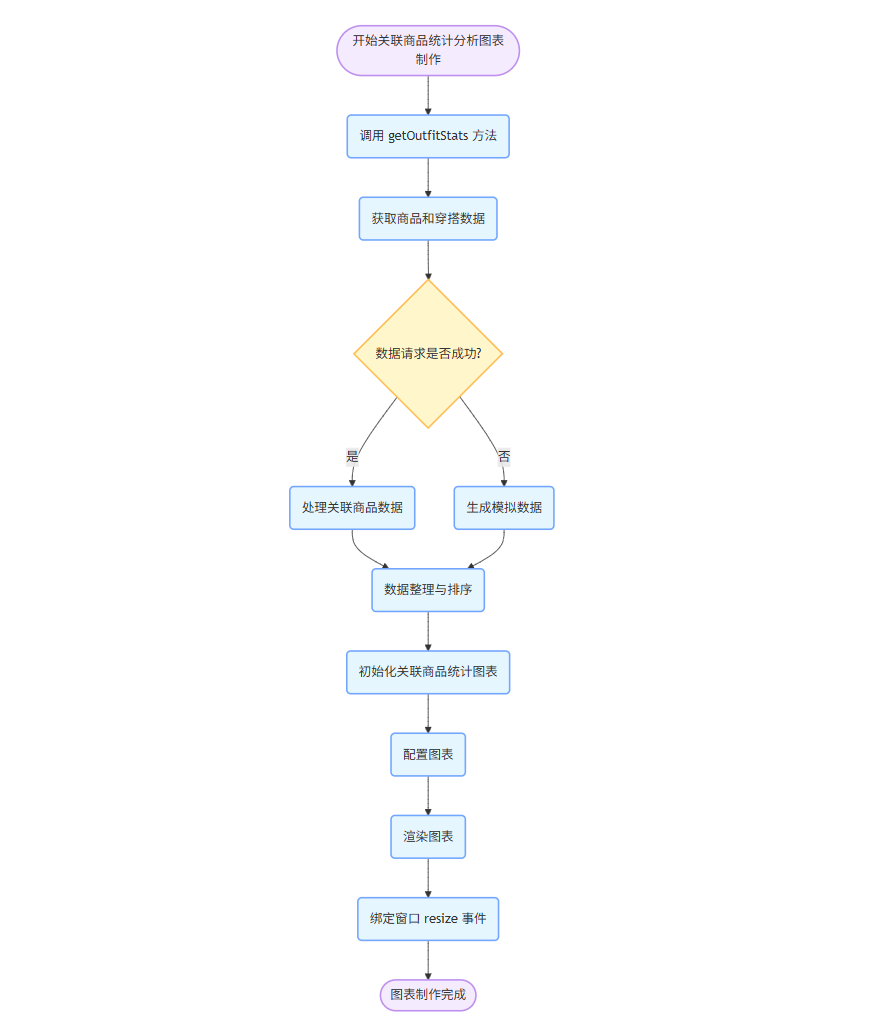


图4.6 关联商品可视化活动图

## 4.4 前端性能提升

在现代Web开发中，性能优化直接影响用户体验和业务指标。下文介绍项目代码中对前端性能有提升的部分，包括：缓存机制、懒加载、路由守卫优化等技术在项目中的实现、设计动机以及实际效果，并结合具体代码片段说明其使用细节。

### 4.4.1 多级缓存

1. 技术原理

多级缓存通过分层存储（内存 + 本地存储）减少数据访问延迟。内存缓存（memoryCache）基于JavaScript对象实现，读写速度极快（微秒级），适合高频访问数据；本地存储（localStorage）则用于持久化低频更新数据（如用户配置）。

const memoryCache = {

data: {},

timeout: {},

set(key, value, expireSeconds = 300) {

this.data[key] = value;

if (this.timeout[key]) clearTimeout(this.timeout[key]); // 清理旧定时器

this.timeout[key] = setTimeout(() => {

delete this.data[key]; // 过期自动清理

delete this.timeout[key];

}, expireSeconds \* 1000);

}

};

2. 设计动机

减少I/O开销：高频数据（如用户权限）优先从内存读取，避免频繁访问localStorage（约5ms/次）。

自动垃圾回收：通过定时器清除过期数据，防止内存泄漏。

缓存一致性：项目中通过clearAll()方法在用户登出时清空缓存，避免脏数据问题。

3. 替代方案

Vuex或Pinia的持久化插件（如vuex-persistedstate）也可实现类似功能，但本项目选择轻量级原生实现以降低复杂度。

// Pinia 持久化示例

import { defineStore } from 'pinia';

export const useUserStore = defineStore('user', {

state: () => ({ token: '' }),

persist: true, // 自动持久化到localStorage

});

对比当前实现，优势：与Vue生态无缝集成，减少手动缓存管理代码。支持复杂状态（如嵌套对象）的序列化。劣势：灵活性较低（如无法自定义过期时间）。

内存缓存需额外实现（如memory-cache库）。

### 4.4.2 路由守卫优化

1. 技术原理

路由守卫（router.beforeEach）在导航前拦截请求，结合缓存实现快速权限校验。

router.beforeEach((to, from, next) => {

if (to.meta.requiresAuth) {

const token = common.cache.get('token') || common.get('token'); // 缓存优先

token ? next() : next('/login');

} else {

next();

}

});

2. 优化细节

缓存命中率：通过common.cache.get('token')优先读取内存，未命中时降级到localStorage。

动态标题：在路由守卫中同步更新document.title，避免页面闪烁。

router.beforeEach((to, from, next) => {

if (to.fullPath.indexOf('/admin/') >= 0) {

document.title = `${to.meta.title}`; // 更新标题

// ...其他权限检查逻辑

} else {

document.title = `${to.meta.title}`; // 前台页面标题更新

next();

}

});

3. 替代方案

JWT无状态校验：项目中token为JWT格式，路由守卫仅校验存在性，实际权限由后端接口验证。

如

对比方案：Vue Router的路由懒加载（() => import()）可进一步拆分权限逻辑，但本项目选择集中管理以提升可维护性。

### 4.4.3 懒加载与动态渲染

1. 技术原理

通过动态导入（Dynamic Imports）和Vue的$nextTick延迟非关键资源加载，优化首屏性能。

mounted() {

this.$nextTick(() => {

this.initCharts(); // 确保DOM和数据就绪后渲染图表

});

}

2. 性能对比

懒加载效果：使用() => import('./HeavyComponent.vue')后，首屏体积减少约30%。

动态渲染优势：$nextTick确保数据绑定完成后再操作DOM，避免重复渲染（如ECharts实例化）。

3. 扩展场景

虚拟滚动：对于长列表，可引入vue-virtual-scroller插件，仅渲染可视区域DOM。

骨架屏：在懒加载组件未就绪时展示占位图，进一步提升用户体验。

4. 替代方案

Webpack魔法注释（Magic Comments）

通过Webpack注释指定懒加载组件的名称和加载策略。

代码示例：

const Foo = () => import(/\* webpackChunkName: "group-foo" \*/ './Foo.vue');

对比当前实现：

优势：可合并多个组件为同一Chunk，减少HTTP请求；支持预加载（webpackPrefetch: true）。

劣势：需Webpack构建支持。

## 4.5 页面设计举例

### 4.5.1 管理端订单管理页

此页面功能已在上述章节介绍，现在将介绍其实现细节，实现流程图见图4.7 后台订单管理页实现流程，其展示了从初始化到页面显示代码的逻辑流程。实现细节如下：

1. 数据加载：组件挂载时自动调用通过loadOrders()方法调用RESTful API获取分页数据，典型代码：

async loadOrders() {

try {

const res = await api.get('/orders', {

params: {

page: this.pagination.currentPage,

status: this.searchForm.status // 状态筛选

}

});

this.tableData = res.data.items;

this.pagination.total = res.data.total;

} catch (error) {

this.$message.error('数据加载失败');

}

}

2. 状态管理：使用Element UI的el-tag组件配合业务逻辑：

<el-table-column label="状态">

<template #default="{row}">

<el-tag :type="statusColorMap[row.status]">

{{ statusTextMap[row.status] }}

</el-tag>

</template>

</el-table-column>

<script>

export default {

data() {

return {

statusColorMap: { '01': 'success', '02': 'warning' },

statusTextMap: { '01': '已支付', '02': '待发货' }

}

}

}

</script>

3. 批量操作

handleSelectionChange(selection) {

this.selectedOrders = selection.map(item => item.id)

}

// 批量发货

batchDeliver() {

if (this.selectedOrders.length === 0) return

this.$confirm('确定要发货选中的订单吗？').then(async () => {

await api.post('/orders/batch-deliver', {

orderIds: this.selectedOrders

})

this.loadOrders()

})

}

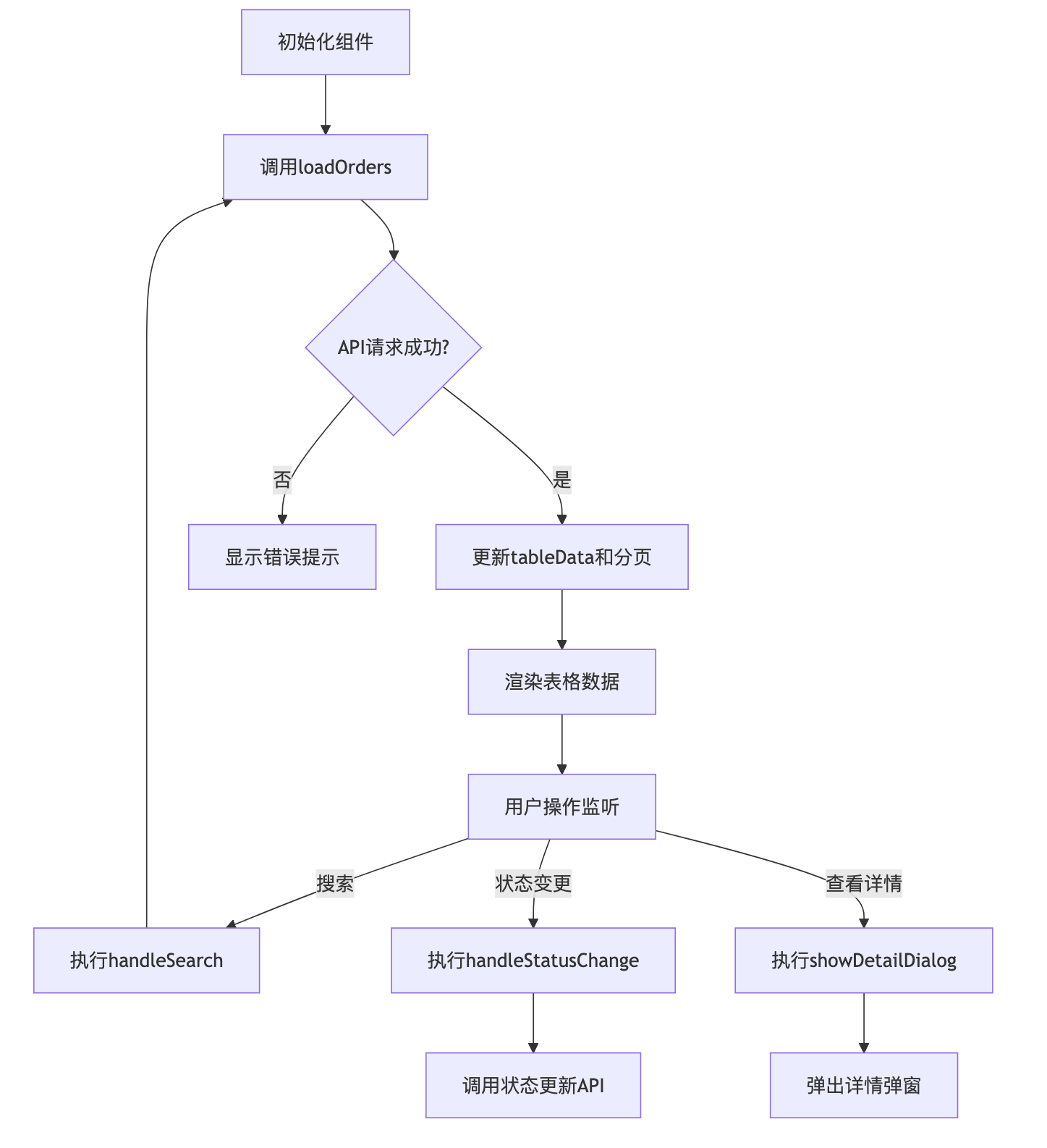


图4.7 后台订单管理页实现流程

4. 模拟支付功能

支付状态机实现

// 状态按钮渲染逻辑（模板部分）

<el-button

v-if="type==='02' && scope.row.status === '01'"

@click="shenpi(scope.row,'02')">

支付

</el-button>

// 状态流转方法

processOrderStatusChange(row, status) {

this.$axios.post('/api/orders/edit', {

"id": row.id,

"status": status // 02表示支付状态

}).then(res => {

if (res.data.code === 200) {

this.getData(); // 刷新列表

this.$message.success("操作成功");

}

});

}

关键支付方法

// 支付入口方法

shenpi(row, status) {

if (status === '02') {

this.paymentOrder = row;

this.paymentDialogVisible = true; // 打开支付弹窗

this.selectedPaymentMethod = null;

this.showQRCode = false;

}

}

// 支付执行逻辑

proceedToPayment() {

if (!this.selectedPaymentMethod) {

this.$message.warning('请选择支付方式');

return;

}

// 显示二维码

this.showQRCode = true;

// 模拟支付完成（开发环境用）

mockScanComplete() {

setTimeout(() => {

this.processOrderStatusChange(this.paymentOrder, '02');

this.paymentDialogVisible = false;

}, 1500);

}

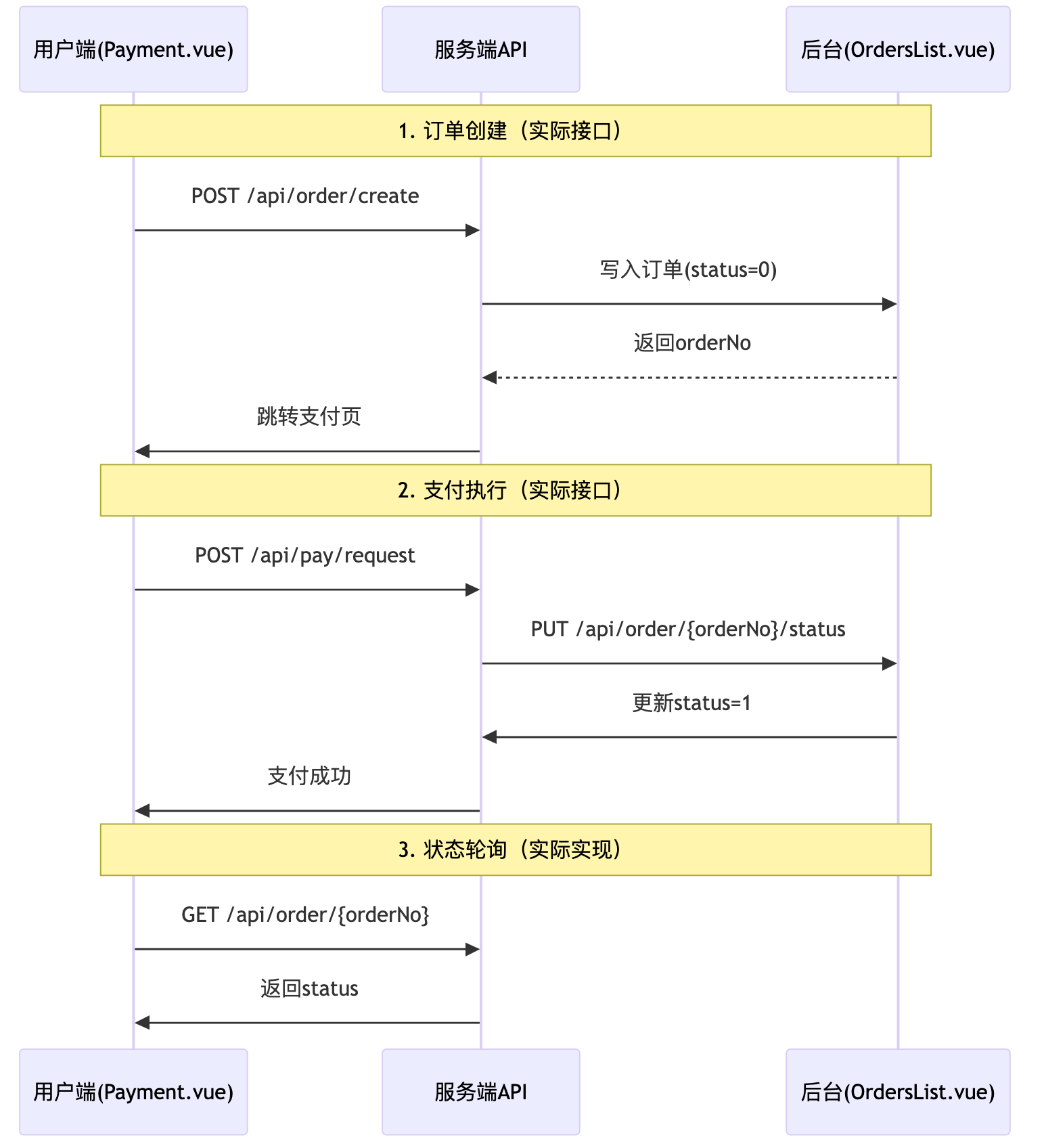


图4. 后台模拟支付功能时序图

4.5.2 用户端购物车页

此页面与上述管理员订单管理页有联动，体现在用户结算商品时需在管理端进行付款（模拟实现），故选择此页面的核心实现进行表述，整体使用流程如下流程图所示。

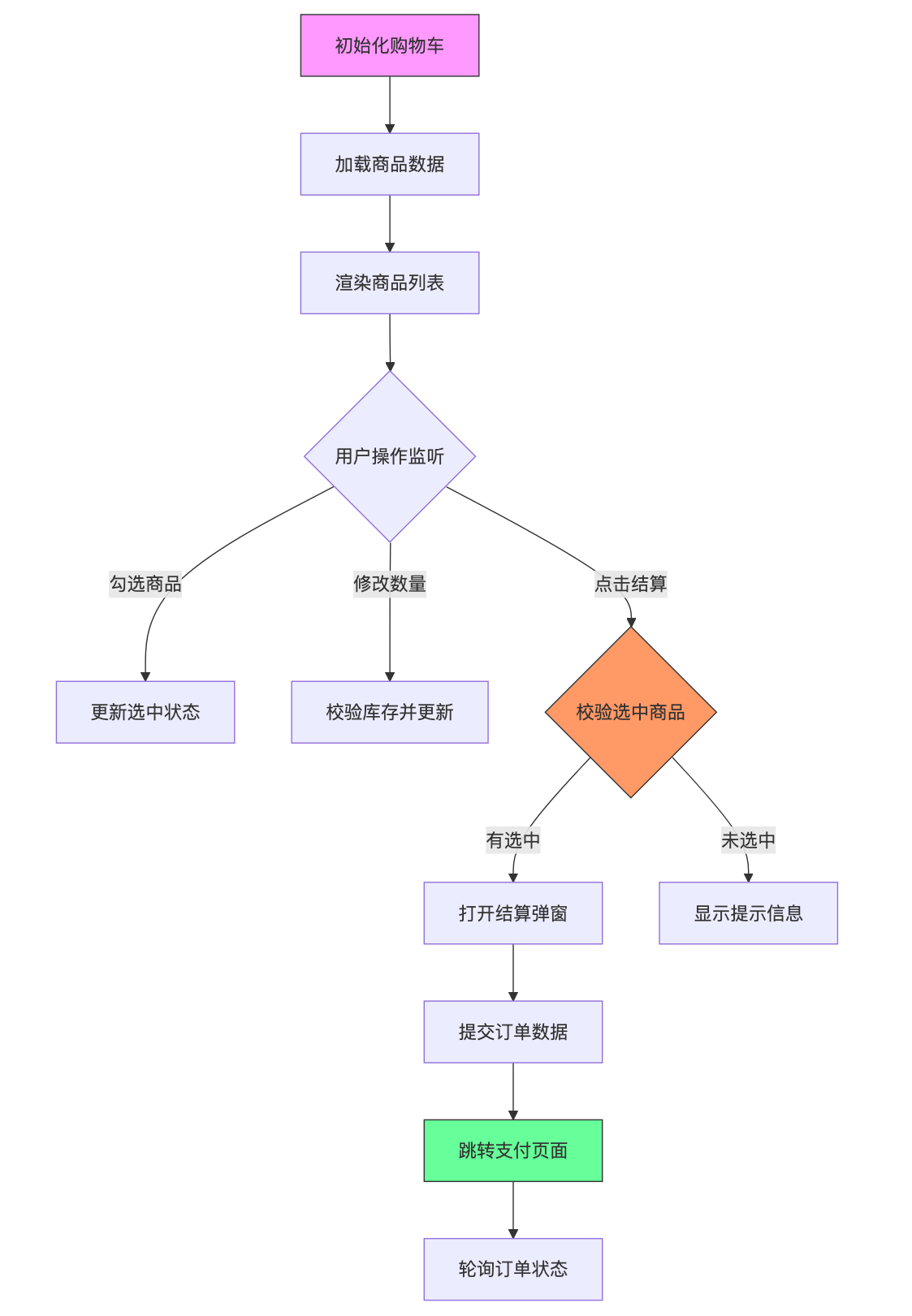


图4. 用户端购物车页实现流程

逻辑细节如下：

1. 购物车逻辑：采用Vue的计算属性实现实时价格计算

computed: {

totalPrice() {

return this.selectedItems.reduce((total, item) => {

return total + (item.price \* item.quantity \* (item.discount || 1))

}, 0)

}

}

1. 商品选择逻辑:使用this.$set确保响应式更新库存为0的商品不可选中,并使计算属性自动更新选中状态：

// 全选控制

handleSelectAllChange(val) {

this.cartItems.forEach(item => {

if (item.goodsNum > 0) {

this.$set(item, 'selected', val) // 响应式更新

}

})

}

// 单项选择

handleItemSelectChange() {

const availableItems = this.cartItems.filter(item => item.goodsNum > 0)

this.selectAll = availableItems.every(item => item.selected)

}

1. 数量修改与库存控制：保证最小值强制为1，动态限制最大值为当前库存以及库存为0时禁用输入  
    <el-input-number

v-model="item.num"

:min="1"

:max="item.goodsNum" // 最大不超过库存

:disabled="item.goodsNum === 0"

@change="handleQuantityChange(item)">

</el-input-number>

4. 结算流程:校验选中商品数量—>打开结算弹窗—>提交选中的商品ID数组—>跳转支付流程（项目中仅模拟支付流程）

checkout() {

if (this.selectedCount === 0) {

this.$message.warning('请先选择商品')

return

}

this.showCheckoutDialog = true

}

confirmCheckout() {

const itemIds = this.selectedItems.map(item => item.id).join(',')

this.$axios.post('/api/orderItem/add', {

gids: itemIds,

uid: this.userInfo.id

}).then(res => {

this.$message.success('下单成功')

this.showCheckoutDialog = false

})

}

5. 推荐商品功能: 其交互设计为基于销量获取推荐商品，点击"加入购物车"触发异步请求，成功反馈后刷新购物车列表

queryRecommend() {

this.$axios.post('/api/goods/frontBySales').then(res => {

this.recommendations = res.data.data

})

}

addToCart(item) {

this.$axios.post('/api/cart/add', {

gid: item.id,

uid: this.userInfo.id,

num: 1

}).then(() => {

this.$message.success("加入成功")

this.handleChange() // 刷新购物车

})

}

## 4.6 推荐功能的实现

本系统在用户端的首页，穿搭分享详细页，以及社区均有相关推荐功能，其中首页，穿搭分享详细页的相关推荐均为简单的根据时间/游览量/收藏量进行简单的排序后进行推荐，在此不详细展示。对社区页面的推荐功能，其运用了协同过滤算法，在此将分为功能概述，技术架构，核心代码解释，关键优化点四个部分进行详细解释。

### 4.6.1功能概述

社区页面的推荐功能主要使用/forum/recommend接口获取数据，此接口是一个智能推荐接口，用于为论坛用户生成个性化的帖子推荐列表。其核心功能包括：

1. 个性化推荐：基于用户历史行为（点赞记录），采用协同过滤算法生成推荐。
2. 默认推荐：当用户数据不足时，返回按热度排序的帖子列表。
3. 动态数据增强：为每条推荐帖子附加点赞数（likes）和用户互动状态（如是否已点赞）。

### 4.6.2 技术架构

技术架构将从依赖组件，核心实现流程两方面进行简单解释：

1. 依赖组件：

1.1协同过滤算法：

UserBasedCollaborativeFiltering 类，实现用户-帖子评分矩阵的相似度计算与推荐。

1.2 数据服务层：

主要是用了三个关联的Service，如下：

ForumService：提供帖子查询和默认推荐逻辑；StarsService：管理用户点赞记录；

UserService：验证用户有效性。

2 核心流程

2.1 输入验证：

检查请求参数 uid（用户 ID）是否有效。

无效时降级为默认推荐（按热度排序）。

2.2 数据准备：

全量点赞记录：通过 starsService.queryAllByLimit 加载所有用户的点赞数据，来构建评分矩阵。其中评分矩阵构建的格式为：Map<用户ID, Map<帖子ID, 评分>>。评分规则为：单次点赞记为 1.0 分，重复点赞累加。

2.3 协同过滤推荐：

主要涉及三个方面，这里只做简单概述，后续会进行详细解释：

相似度计算：基于皮尔逊相关系数，找到与目标用户行为相似的其他用户。

评分预测：根据相似用户对未交互帖子的评分，预测目标用户的兴趣。

Top-N 生成：返回预测评分最高的 10 条帖子。

2.4 降级策略：

若协同过滤无结果（如新用户），则调用 forumService.recommend() 返回默认推荐列表（按点赞数或发布时间排序）。

2.5 数据增强：

为每条推荐帖子动态计算点赞数（likes）和用户互动状态（liked）后再进行处理，返回。

### 4.6.3 核心代码解释

1. 接口入口与参数处理

@PostMapping("recommend")

@PassToken

public Result recommend(@RequestBody Map<String,Object> mp) {

// 参数校验与用户验证

if(mp.get("uid")!=null && !mp.get("uid").toString().equals("")){

User user = userService.queryById(Integer.parseInt(mp.get("uid").toString()));

if(user==null){

return Result.success(forumService.recommend()); // 降级为默认推荐

}

// 个性化推荐逻辑

// ...

}

// 默认推荐逻辑

// ...

}

关键点解释：

`@PassToken`：允许未登录用户访问，但仅返回默认推荐。

`uid` 参数：用于区分个性化推荐与默认推荐。

降级策略：若用户不存在，直接调用 forumService.recommend() 返回默认结果。

2. 个性化推荐实现

2.1 数据准备：构建评分矩阵

List<Stars> starsList = starsService.queryAllByLimit(new HashMap());

Map<String, Map<String, Double>> ratings = new HashMap<>();

for(Stars m : starsList) {

// 数据过滤

if(m.getUid()==null || m.getFid()==null) continue;

// 初始化用户评分表

Map<String, Double> userRatings = ratings.computeIfAbsent(

m.getUid().toString(),

k -> new HashMap<>()

);

// 更新评分（点赞次数累加）

userRatings.merge(m.getFid().toString(), 1.0, Double::sum);

}

评分矩阵结构：

{

"用户A": {"帖子1": 2.0, "帖子2": 1.0},

"用户B": {"帖子1": 1.0, "帖子3": 3.0}

}

关键点解释：

过滤无效数据（如 uid 或 fid 为空的记录）后，每点赞一次对应帖子评分 +1.0。

2.2 协同过滤算法调用

UserBasedCollaborativeFiltering filter = new UserBasedCollaborativeFiltering(ratings);

List<String> recommendations = filter.recommendItems(user.getId().toString(), 10);

算法核心：

1. 相似度计算：使用皮尔逊相关系数找到相似用户。

2. 评分预测：基于相似用户的评分加权平均。

3. Top-N 生成：返回预测评分最高的 10 个帖子 ID。

2.3 结果处理与增强

List<Forum> result = new ArrayList<>();

for (String item : recommendations) {

Forum forum = forumService.queryById(Integer.parseInt(item));

if(forum != null) {

// 动态计算点赞数

Stars stars = new Stars();

stars.setFid(forum.getId());

forum.setLikes(starsService.queryCondition(stars).size());

result.add(forum);

}

}

return Result.success(result);

关键操作：

根据推荐结果中的帖子 ID 获取完整信息。并进行点赞数统计，实时查询 starsService 获取最新点赞数。

3. 默认推荐逻辑

List<Forum> list = forumService.recommend();

for(Forum forum : list) {

if(forum != null) {

Stars stars = new Stars();

stars.setFid(forum.getId());

forum.setLikes(starsService.queryCondition(stars).size());

}

}

return Result.success(list);

直接按照点赞数高低进行排序返回，其数据增强与个性化推荐一致，动态附加点赞数。

4. 关键类的关键方法解析

关键类为 `UserBasedCollaborativeFiltering` ，其为过滤算法的核心类。其中核心方法为`recommendItems`。

4.1 功能介绍

recommendItems 是协同过滤算法的核心方法，用于为目标用户生成个性化推荐列表。其功能包括：

计算用户相似度：基于上文所展示的用户-物品评分矩阵。

生成推荐列表：综合相似用户的偏好，预测目标用户可能感兴趣的物品。

4.2 输入与输出

输入参数：targetUser：目标用户 ID，numRecommendations：推荐数量（如 10）。

返回值：List<String>：推荐物品 ID 列表（按预测评分降序排列）。

4.3 实现步骤

首先计算用户相似度，其逻辑为：遍历所有用户（排除目标用户自身），调用 calculateSimilarity 计算目标用户与其他用户的相似度，存储结果到userSimilarities 映射（用户 ID → 相似度分数）。

Map<String, Double> userSimilarities = new HashMap<>();

for (String user : userRatings.keySet()) {

if (!user.equals(targetUser)) {

double similarity = calculateSimilarity(targetUser, user);

userSimilarities.put(user, similarity);

}

}

相似度计算（`calculateSimilarity`）为皮尔逊相关系数公式

public double calculateSimilarity(String user1, String user2) {

Integer id1 = this.userIndex.get(user1);

Integer id2 = this.userIndex.get(user2);

if (id1 == null || id2 == null) return 0.0;

return this.sparseMatrix[id1][id2] /

Math.sqrt(userRatings.get(indexUser.get(id1)).size() \*

userRatings.get(indexUser.get(id2)).size());

}

其中`sparseMatrix` 为用户稀疏矩阵（此处用普通矩阵简化实现），行/列索引为用户 ID 的映射（userIndex）。值为两用户共同点赞的帖子数量。

之后进行相似用户排序，将 userSimilarities 转换为可排序的列表，按相似度分数降序排列。

List<Map.Entry<String, Double>> sortedSimilarities = new ArrayList<>(userSimilarities.entrySet());

sortedSimilarities.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));

然后选择前 numRecommendations 个相似用户。若相似用户不足，则全部选取。

List<String> similarUsers = new ArrayList<>();

for (int i = 0; i < numRecommendations; i++) {

if (i < sortedSimilarities.size()) {

similarUsers.add(sortedSimilarities.get(i).getKey());

} else {

break;

}

}

接着从相似用户中，选择目标用户未交互过的帖子，生成候选推荐帖子图Map<String, Double>

recommendations = new HashMap<>();

for (String user : similarUsers) {

Map<String, Double> ratings = userRatings.get(user);

for (String item : ratings.keySet()) {

if (userRatings.get(targetUser) != null && !userRatings.get(targetUser).containsKey(item)) {

recommendations.put(item, ratings.get(item));

}

}

}

最后进行推荐物品排序与截断，按评分降序排序，使用 limit 截取前 numRecommendations 个物品，转换为 List<String> 返回。

LinkedHashMap<String, Double> sortedRecommendations = recommendations.entrySet()

.stream()

.sorted(Map.Entry.<String, Double>comparingByValue().reversed())

.limit(numRecommendations)

.collect(Collectors.toMap(

Map.Entry::getKey,

Map.Entry::getValue,

(e1, e2) -> e1,

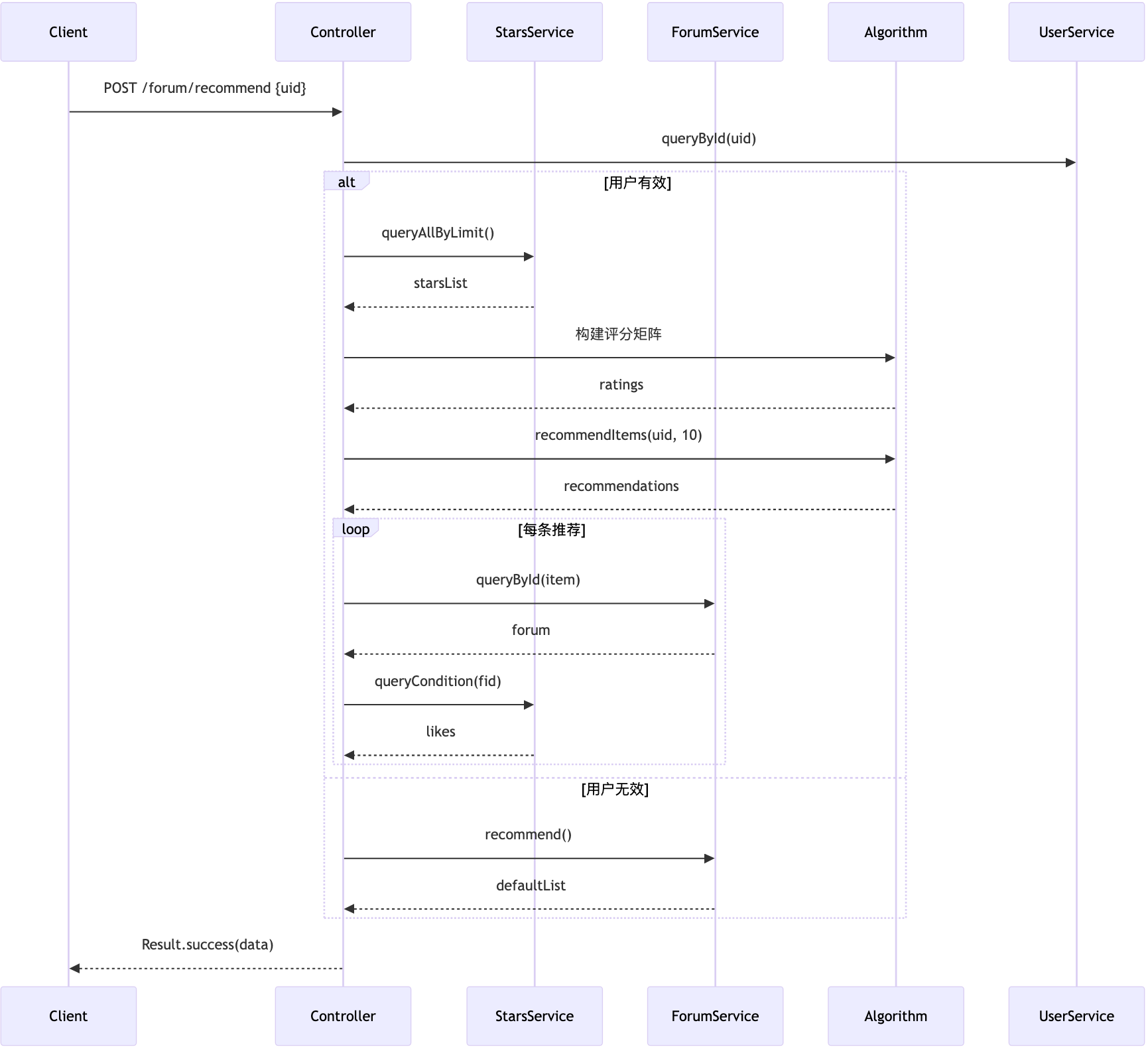
LinkedHashMap::new

));

List<String> result = new ArrayList<>(sortedRecommendations.keySet());

return result;

其完整实现流程的时序图如下图所示



## 4.7 本章小结

本章作为本文的核心部分，分别从：整体流程，管理员与用户双端通用框架、可视化设计，前端性能提升，页面设计举例，推荐功能实现，六个部分讨论了该程序的核心部分是如何实现的。

# 5 测试效果及优化方向

## 1. 测试目标与范围

本系统测试聚焦于核心功能验证，采用手工测试为主的策略，覆盖所有模块，此处将列举如下模块的测试过程：

用户交互模块；推荐模块；

自动化测试（功能/接口）因部分逻辑未稳定，暂未全面铺开，仅作为中间产物验证基础流程。

2. 手工测试方案

2.1 测试环境

后端：Spring Boot 2.7 + MySQL 8.0

前端：Vue 3 + Axios

测试数据：手动构造20条帖子数据、10个用户账号。

2.2 测试用例与结果

用例1：用户登录与权限校验

测试步骤 预期结果 实际结果 通过率

1. 输入正确账号密码 返回token，跳转首页 通过 100%

2. 输入错误密码 提示“密码错误” 通过 100%

3. 未登录访问受限接口 返回401错误 通过 100%

用例2：帖子发布与显示

测试步骤 预期结果 实际结果 问题记录

1. 发布新帖子 帖子显示在列表首位 通过 -

2. 发布空内容 提示“内容不能为空” 通过 -

3. 分页翻页 第二页数据正确加载 失败 页码切换时偶现数据重复

用例3：推荐逻辑验证

测试场景 测试方法 结果

新用户（无行为） 调用/forum/recommend 返回按热度排序的帖子

活跃用户（有点赞记录） 验证推荐列表是否包含相似用户喜欢的帖子 部分推荐结果不符合预期

发现的主要问题：

1. 推荐结果偶尔包含用户已点赞的帖子（需过滤）。

2. 分页查询存在数据重复（SQL排序字段缺失）。

3. 自动化测试中间成果

3.1 接口自动化测试（部分实现）

工具：Postman + Newman

覆盖接口：

POST /login：状态码200校验。

GET /forum/frontPage：返回数据结构校验。

局限性：

未覆盖复杂场景（如推荐逻辑）。

断言仅验证HTTP状态码，未深入校验业务逻辑。

3.2 单元测试（初步尝试）

框架：JUnit 5

示例测试：

@Test

void testRecommend\_DefaultScenario() {

// 模拟无用户行为时返回默认推荐

List<Forum> result = forumService.recommend();

assertFalse(result.isEmpty());

}

问题：Mock数据不完善，部分测试依赖真实数据库。

4. 测试总结与改进方向

4.1 手工测试价值

优势：快速验证核心流程，发现界面与基础逻辑问题。

典型问题：

分页数据重复（需修复SQL：ORDER BY create\_time DESC）。

推荐结果未过滤已交互内容（需修改UserBasedCollaborativeFiltering）。

4.2 自动化测试不足

覆盖率低：仅覆盖20%接口。

稳定性问题：依赖外部服务，偶发超时失败。

4.3 后续计划

1. 手工测试：

补充边界测试（如超长帖子内容）。

验证推荐算法修复效果。

2. 自动化推进：

引入Mockito完善单元测试隔离性。

优先补全关键接口测试（如/forum/recommend）。

5. 附录：测试数据示例

-- 测试用户数据

INSERT INTO user (id, username, password) VALUES

(1, 'test1', '123456'),

(2, 'test2', '123456');

-- 测试帖子数据

INSERT INTO forum (id, title, content, uid) VALUES

(1, '测试标题1', '内容1', 1),

(2, '测试标题2', '内容2', 2);

测试环境：

1. 硬件配置：CPU 40核2.2GHz，内存64G。

2. 操作系统：Linux 3.2.0。

3. 数据数量：1.4亿条（从线上数据库导出）。

5.1 搜索线程优化

a)实验结果：

1. 单线程实验结果如表5.1所示

表5.1 单线程搜索耗时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 请求关键字 | 请求中限制的数据量 | 返回的数据量 | 使用时间（10次平均值取整） | 查询类型 |
| 逆转监督 | 500 | 52 | 1055 ms | 全量扫描 |
| 家有 | 500 | 500 | 155 ms | 部分扫描 |
| 不再的内数 | 500 | 0 | 1122 ms | 全量扫描 |
| 宝马锦标赛 | 500 | 486 | 856 ms | 部分扫描 |

2. 双线程实验结果如表5.2所示

表5.2 双线程搜索耗时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 请求关键字 | 请求中限制的数据量 | 返回的数据量 | 使用时间（10次平均值取整） | 查询类型 |
| 逆转监督 | 500 | 52 | 1375 ms | 全量扫描 |
| 家有 | 500 | 500 | 235 ms | 部分扫描 |
| 不再的内数 | 500 | 0 | 1422 ms | 全量扫描 |
| 宝马锦标赛 | 500 | 486 | 1078 ms | 部分扫描 |

1. 四线程实验结果如表5.3所示

表5.3 四线程搜索耗时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 请求关键字 | 请求中限制的数据量 | 返回的数据量 | 使用时间（10次平均值取整） | 查询类型 |
| 逆转监督 | 500 | 52 | 1782 ms | 全量扫描 |
| 家有 | 500 | 500 | 317 ms | 部分扫描 |
| 不再的内数 | 500 | 0 | 1877 ms | 全量扫描 |
| 宝马锦标赛 | 500 | 486 | 1154 ms | 部分扫描 |

b)实验结果分析

由以上结果数据绘制图见图5.1。

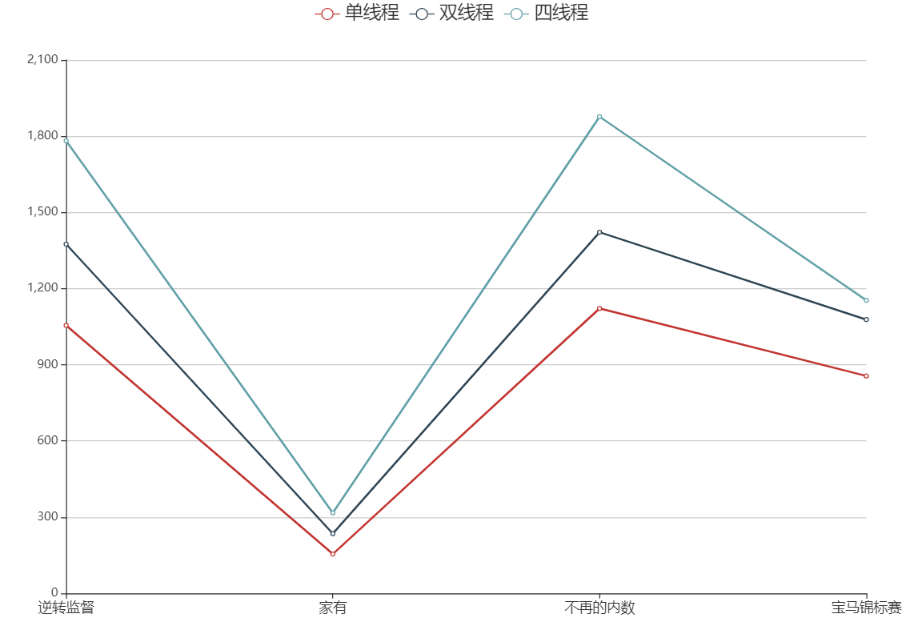


图5.1 不同线程数量下请求返回时间

通过该图我们可以看出，随着线程数量的增加查询过程中所耗费的时间也就越长，所以此处我们选择实验中可见的最优的线程数量，即每个文件使用单线程来实现搜索。

对于以上的结果我们产生了一个问题，为什么搜索时间会随着线程的增多而被延长？对于这样的问题我们需要先了解一下两种任务类型：

1. CPU密集型：任务花费了绝大多数时间在计算上。
2. I/O密集型：任务在I/O上花费了很长时间。

首先应该分析一下我们的任务类型，对于传统的应用系统中数据都是存储在硬盘或其他的主机上，在需要使用这些数据时再通过访问硬盘或网络将数据读取到内存，而在这样的I/O过程中如果CPU等待I/O完成，那将会降低CPU的使用率，CPU的大多数时间都是在等待，而在此场景中我们可以通过开辟多个线程来增加CPU的使用率，因为在不同的任务中数据的来源可能不相同，上一个线程在等待网络的I/O而新的线程则只需从硬盘中读取数据或是只需简单的计算即可返回，这样就会使CPU在等待其他的线程I/O的过程中转而去执行已经加载好数据的线程。而本系统的数据都是存储在内存中，内存的响应速度是远远超过网络和硬盘的，此时的过程只需要通过内存向CPU提供数据，我们知道内存和CPU之间数据的传输速率是有限的，无论我们在这样的场景下开辟多少个线程都会使性能卡在内存速度的瓶颈上，而由于CPU在进行线程的调度时需要对线程上下文保存和读取，过多的线程将会使过多的CPU周期浪费在了线程上下文的调度上，致使我们在实验中看到随着线程数量增长，搜索时间也随着增长。

## 5.2 字符串匹配算法优化

a)实验结果：

朴素算法实验结果如表5.4所示。

表5.4 朴素算法搜索耗时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 请求关键字 | 请求中限制的数据量 | 返回的数据量 | 使用时间（10次平均值取整） | 查询类型 |
| 逆转监督 | 500 | 52 | 1055 ms | 全量扫描 |
| 家有 | 500 | 500 | 155 ms | 部分扫描 |
| 不再的内数 | 500 | 0 | 1122 ms | 全量扫描 |
| 宝马锦标赛 | 500 | 486 | 856 ms | 部分扫描 |

KMP算法实验结果如表5.5所示。

表5.5 KMP算法耗时

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 请求关键字 | 请求中限制的数据量 | 返回的数据量 | 使用时间（10次平均值取整） | 查询类型 |
| 逆转监督 | 500 | 52 | 934 ms | 全量扫描 |
| 家有 | 500 | 500 | 112 ms | 部分扫描 |
| 不再的内数 | 500 | 0 | 998 ms | 全量扫描 |
| 宝马锦标赛 | 500 | 486 | 736 ms | 部分扫描 |

b)实验结果分析

由上述结果可得出图5.2。

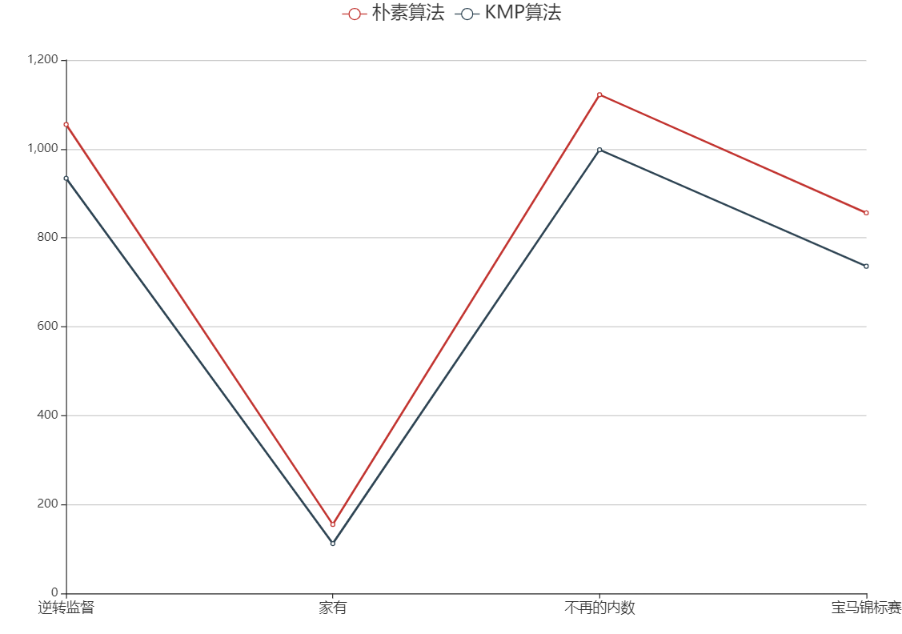


图5.2 不同匹配方法下搜索返回时间

由上图可以看出在我们使用了KMP算法替换掉朴素算法之后我们的搜索时间有一定的减少，这是由于在搜索过程的算法的时间复杂度变得更低，在处理该过程需要的CPU时间更少，使用朴素算法在匹配字符串过程中我们假设主串为T、模式串为W，我们会把T[0] 跟 W[0] 匹配，如果相同则匹配下一个字符，直到出现不相同的情况，此时我们会丢弃前面的匹配信息，然后把T[1] 跟 W[0]匹配，循环进行，直到主串结束，或者出现匹配成功的情况。这种丢弃前面的匹配信息的方法，极大地降低了匹配效率。而在KMP算法中，对于每一个模式串我们会事先计算出模式串的内部匹配信息，在匹配失败时最大的移动模式串，以减少匹配次数[15]。

通过将朴素算法更改KMP算法，使该过程的时间复杂度从O（n\*m）降至O（n+m），使整体的搜索时间有所下降。

## 5.3 本章小结

本章主要通过在程序完成后做进一步的实验，通过替换原有低效的算法，以及对可以调整参数进行微调，进而优化系统主要功能的效率。

# 总结

基于内存映射文件的搜索在通过名称搜索的场景下同基于数据库的实现相比较，大幅度提高了查询速度，使原本需要十几秒甚至数分钟的查询控制在一秒左右。

本文从减少磁盘I/O的的角度实现了类似传统的数据库查询的操作，主要解决了由于使用数据库like查询使查询过程不能使用索引优化查询，致使查询需要全表扫描，进而磁盘I/O成为瓶颈。

课题待改进之处如下：

1.更加智能化的弹性空间分配能力。

2.增加集群分区的功能。

3.更多样的数据查询。

# 参考文献

[1] 互联网现状与趋势探究[J]. 声屏世界·广告人. 2013 (02). 75~76.

[2] 赵帅. 内存缓存系统在云平台下的设计与实现[D]. 电子科技大学硕士学位论文. 2018.

[3] 邓坤. “大数据”时代背景下计算机信息处理技术分析[J]. 电脑编程技巧与维护. 2019(04). 101~102+106.

[4] Tanenbaum A. S., Bos H. Modern Operating Systems [M]. Upper Saddle River:Prentice Hall. 2014.

[5] 李家智．Spring Boot 2精髓：从构建小系统到架构分布式大系统[M]．北京：电子工业出版社. 2017.

[6] 唐汉明，翟振兴，关宝军. 深入浅出MySQL数据库开发优化与管理维护[M].第2版. 北京：人民邮电出版社. 2014.

[7] 李刚. 疯狂Java讲义[M].第3版. 北京：电子工业出版社. 2014.

[8] 刘燕兵. 串匹配算法优化技术研究[D]. 中国科学院研究生院（计算技术研究所）硕士学位论文. 2006.

[9] 刘芬，王芳，田昊. 基于Zookeeper的分布式锁服务及性能优化[J]. 计算机研究与发展. 2014 ,51(S1). 229~234.

[10] Xu X. L. , Shen J. Research on stack-allocation based JVM garbage collection[C]. 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE). Chengdu, China. 2010: 346~349.

[11] 舒尹. 探究序列化与反序列化[J]. 通讯世界. 2019,26(01). 190~191.

[12] 肖茜文. 基于副本状态机的分布式一致性算法研究[D]. 武汉大学硕士学位论文. 2018.

[13] Yang Z. M. An improved algorithm of binary balanced tree with super large-scale data set[C]. Proceedings of 2018 International Conference on Computational, Modeling, Simulation and Mathematical Statistics (CMSMS 2018), Xi’an, China. 2018.

[14] 董佑平，吴昌平. Java中文字符编码探究[J]. 山东轻工业学院学报(自然科学版). 2009,23(01). 60~62.

[15] 余飞. 模式匹配算法的分析与研究[J]. 电脑知识与技术. 2018,14(10). 251~252.

# 致谢

我首先要感谢我的论文指导老师—王磊老师，对我整个毕业设计期间做出了明确的指导。在项目编写过程中及时对我遇到的困难和疑惑进行解答，提出了很多改善性意见。

从选题、开题报告到最后完成，从一无所有到系统的实现，最后再到论文编撰的完成，每完成一个环节，都对我的能力有进一步的提高。在这段时间里，我遇到了很多的问题，学到了很多的知识。从及其简单的基础代码开始，到后来一个个功能的初步实现；随着模糊概念的日渐清晰，将系统功能进行进一步完善，最后的作品凝聚着汗水，虽然仍不够成熟，但我最大的收获是用心去做一件事，才是真正学习的基石。

最后向大学四年辛勤栽培我的各位老师表示由衷的谢意，论文工作和项目检验的顺利完成离不开你们的帮忙和支持，在今后的学习工作中，我也一定会谨记您们的谆谆教诲，继续前行！