总体报告

OpenCourse 官方图标

1. 文档引言

1.1. 编写目的

本文档为总体报告（CR），基于完整项目需求和开发实际成果，进行全项目综合性总结与归纳展示，用于交付成果，提供开发流程凭证

1.2. 文档范围

OpenCourse 总体报告主要包含该软件系统整体功能的需求、涉及、实现方式、测试和项目总结

1.3. 读者对象

OpenCourse 项目开发成员、验收方、用户

1.4. 术语表

术语	定义/解释
OpenCourse	适用于在校大学生的信息共享平台，提供课程资源分享、交流互动等功能
Vue3	前端 JavaScript 框架，用于构建响应式用户界面

Vite	前端构建工具，提供快速的开发服务器和优化的打包功能
Element-Plus	基于 Vue3 的 UI 组件库，提供丰富的预定义组件
SpringBoot	Java 后端框架，简化 Spring 应用的初始搭建和开发
JWT (JSON Web Token)	用于身份验证的开放标准，通过令牌传递用户信息
MinIO	高性能的对象存储系统，用于存储和访问文件资源
Docker	容器化技术，用于打包应用及其依赖环境，实现跨平台部署
单例模式	设计模式，确保一个类仅有一个实例，并提供全局访问点
工厂方法模式	设计模式，定义一个创建对象的接口，由子类决定实例化的类
适配器模式	设计模式，使不兼容的接口能够协同工作
责任链模式	设计模式，将请求沿处理链传递，直到有处理者处理
JPA (Java Persistence API)	Java 持久化规范，用于简化数据库操作
Redis	高性能的键值存储数据库，常用于缓存
Pinia	Vue3 的状态管理库，用于集中管理应用状态
API Fox	API 管理工具，用于设计、测试和文档化 API
Maven	Java 项目管理工具，用于依赖管理和项目构建
Devcontainer	开发容器，提供一致的开发环境配置
GitHub	代码托管平台，支持版本控制和协作开发
Node.js	JavaScript 运行时环境，用于执行前端和后端代码
npm	Node.js 的包管理器，用于安装和管理依赖
MySQL	关系型数据库管理系统，用于结构化数据存储
Spring Security	安全框架，提供认证和授权功能
模板方法模式	设计模式，定义算法骨架，子类实现具体步骤
观察者模式	设计模式，实现对象间一对多依赖关系，状态变更时自动通知
享元模式	设计模式，通过共享对象减少内存占用

代理模式	设计模式，通过代理对象控制对目标对象的访问
集成测试	测试方法，验证多个组件是否能协调工作
边界测试	测试方法，检查系统对边界情况的处理能力

---

2. 需求分析

2.1. 项目背景

当前高校课程存在诸多信息屏障：

- 课程资源分散在不同的云盘 / 社交平台
- 跨年级学生之间缺乏有效知识传承渠道
- 课程评价体系不透明

基于此，本项目（ OpenCourse ）通过构建一个适用于在校大学生使用的信息共享平台，给予学生可以互相交流沟通、在线分享学习资源、交流学习经验的信息共享渠道，有助于打通信息屏障，促进学术交流，构建优秀的学术资源共享生态

本项目的目标用户主要为在校大学生群体（本科生、硕士生、博士生等）

2.2. 产品功能

在交付期附近，OpenCourse 开发团队完成了第一个功能完备可以上线的版本（v1.0.0），实现的基本产品功能如下：

- 用户系统
  - 基于用户邮箱 – 密码 – 验证码实现账户注册
  - 基于用户邮箱 – 密码进行账户登录
  - 用户个人信息主页
  - 用户注销
- 课程系统
  - 创建院系部门
  - 创建 / 删除课程

    - 所有院系 / 课程信息总览
- 资源系统
    - 创建 / 删除资源
    - 用户点赞 / 撤赞资源
    - 资源文件自由下载与访问
- 互动系统
    - 发表 / 更新 / 删除互动评论
    - 为课程评分
    - 用户点赞 / 撤赞互动评论

---

3. 总体设计

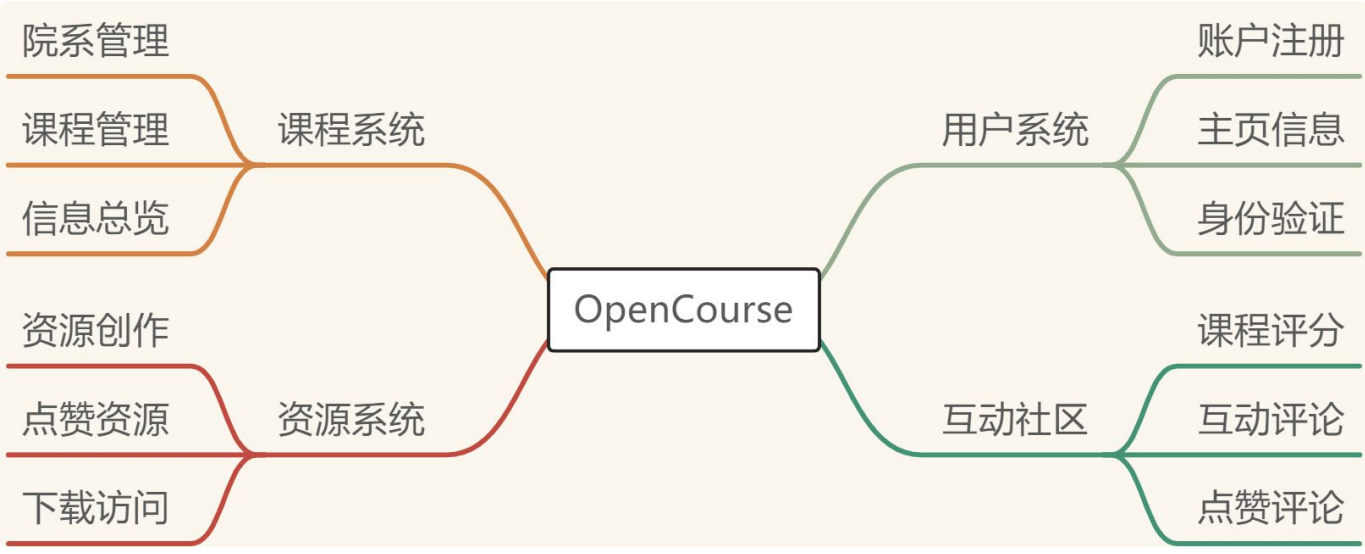
3.1. 系统总体设计

3.1.1. 总体架构设计

本项目采用经典五级分层架构，采用前后端分离模式进行开发，由前端负责视图层、后端负责其余四层进行功能实现

3.1.2. 功能划分

当前版本（v1.0.0）实现的所有功能可以划分为



3.1.3. 技术栈

类别	技术方案
前端框架	Vue3 + Vite
UI 组件库	Element-Plus
JS 运行环境	Node.js + npm 18.19.1
后端框架	Java 21 + SpringBoot 3.4.5
安全框架	Spring Security + JWT
构建工具	Maven 3.6.3
数据库	MySQL 8.0
高性能缓存	Redis 7.2-alpine
文件对象存储系统	MinIO RELEASE.2025-05-24T17-08-30Z
版本控制系统	Github + Git
API 管理工具	API Fox
容器技术	Docker + Devcontainer

3.2. 模块分层设计

3.2.1. 视图层

视图层负责将服务端数据构建为响应式 Web 页面，提供用户友好的交互界面，将用户请求发送到服务端，并接收服务端响应渲染成对应 Web 页面显示给用户

视图层基于所有的产品需求和功能可以主要划分为以下部分：

- Web 界面级组件
- API 接口配置
- 路由配置
- Pinia 状态管理
- 工具调用库

视图层基于 Vue3+Vite 构建响应式 Web 界面，由 Pinia 负责状态管理，使用组件来自于 Element-Plus 组件库

3.2.2. 控制层

控制层负责接收前端发送的请求，进行请求过滤与身份验证、构造与验证请求参数、调用服务层接口，并返回服务层响应、封装成 HTTP 响应，返回给视图层

视图层总体实现如下：

- 用户认证控制器，用于处理用户登录、注册、个人主页等功能
- 院系控制器，处理院系相关请求
- 课程控制器，处理课程相关请求
- 互动控制器，处理互动评论相关请求
- 资源控制器，处理资源相关请求
- 用户控制器，处理用户个人信息相关请求
- JWT 认证工具，用于生成、校验 JWT 令牌
- 无认证路由过滤器，用于处理无需认证即可登录的路由配置
- 全局异常处理器，用于处理控制器所有异常

3.2.3. 服务层

服务层为系统所有服务的核心承载层，负责所有业务逻辑实现

服务层总体实现如下：

- 课程服务类，负责处理课程信息获取、添加删除等功能
- 院系服务类，负责处理院系部门的创建、获取等功能
- 互动服务类，负责处理互动评论的发表、点赞、评分等功能
- 资源服务类，负责处理资源的发表、点赞、下载访问等功能
    - 文件存储子服务，负责调用 MinIO 接口实现文件分布式存储与访问
    - 垃圾清理子服务，负责处理服务端中已经失效的资源文件的清理功能
- 用户服务类，负责处理用户相关的登录、注册、信息获取等功能
    - 邮件子服务，负责调用邮件发送接口实现注册时邮件验证码功能
    - 验证码子服务，负责生成与管理验证码
    - 身份信息子服务，负责从前端 HTTP 请求中加载用户信息

- 历史记录服务类，负责记录应用运行过程中所有用户行为，辅助其它服务类协调完成业务

3.2.4. 映射层

映射层向服务层提供持久化数据的访问接口，通过调用映射层数据接口实现数据库访问，由映射层封装数据库访问操作

映射层基于每个模型层实体分别构建对应的数据访问仓库，基于 JPA 实现

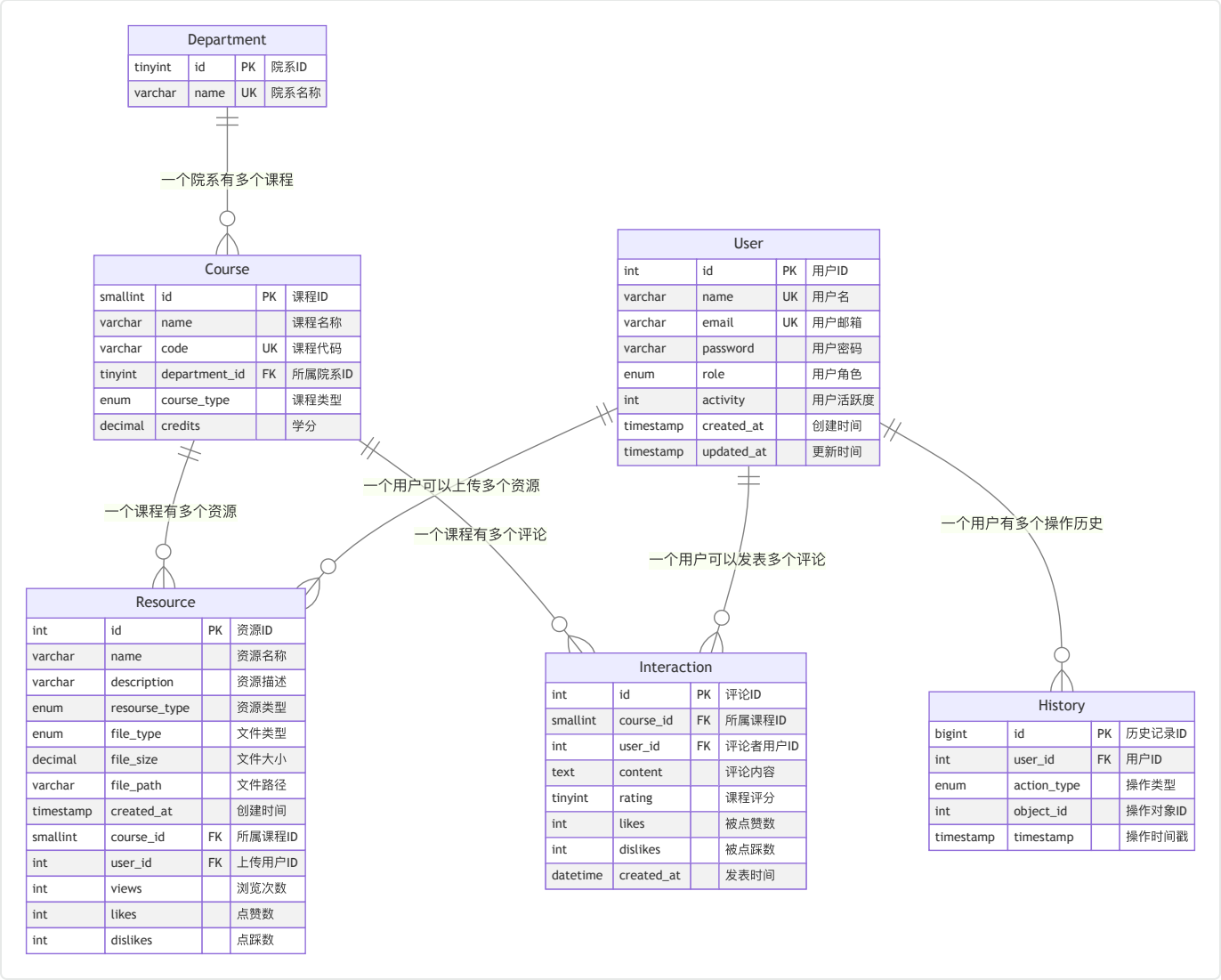
3.2.5. 模型层

模型层定义了所有的业务实体，是整个系统中最基本的服务对象和数据类型

模型层的实体包括：

- 用户实体，代表一个用户
- 院系实体，代表一个院系部门
- 课程实体，代表一个课程
- 资源实体，代表一个资源
    - 资源帖子元数据，如发布用户、发布时间、点赞数量、访问数量
    - 资源文件元数据，如资源文件存储路径、资源文件大小
- 互动实体，代表一个互动评论
- 历史记录实体，代表一个历史记录项

所有以上实体都继承自抽象基类，包含公有属性 ID 作为标识符



除去实体，还有枚举类型定义：

- 用户身份权限，标识用户是游客、普通用户、管理员
- 课程类别，表示课程是否是必修课、专业课
- 资源类型，表示资源是课件、笔记、历年卷等
- 动作类型，表示历史记录项中的用户操作动作类型

---

4. 设计模式

4.1. 概述

在软件工程中，设计模式（ Design Pattern ）是针对软件设计中常见问题的可复用解决方案，它并不是可以直接转换成代码的完整实现，而是一种最佳实践模板，帮助开发者设计更灵活、可维护

和可扩展的软件系统

设计模式类似于建筑领域的蓝图，提供了一套经过验证的结构，使开发者能够避免重复造轮子，而是利用已有的经验来优化代码设计，增强代码的复用性和可维护性，提升系统的扩展性，让开发者高效沟通

4.2. 应用

4.2.1. 创建型模式

创建型模式（Creational Patterns）主要关注**对象创建机制**（如 `Singleton` 确保唯一实例，`Factory` 解耦实例化过程）

4.2.1.1. 单例模式

单例模式（Singleton）确保一个类只有一个实例，并提供全局访问点来获取该实例，其核心思想是**控制实例数量，防止外部通过 `new` 创建多个对象**，并提供静态方法（如 `getInstance()`）以访问唯一实例

在本项目中，后端采用 SpringBoot 框架，由 SpringBoot 框架负责每个 Bean 的创建和管理，实现单例模式，确保每个类只有一个实例

单例模式：Minio 客户端 Java

```java
/**
 * MinIO client bean.
 *
 * @return MinIO client
 */
@Bean
public MinioClient minioClient() {
    return MinioClient.builder()
        .endpoint(minioConfigProperties.getEndpoint())
        .credentials(
            minioConfigProperties.getAccessKey(),
            minioConfigProperties.getSecretKey())
        .build();
}
```

4.2.1.2. 工厂方法模式

工厂方法模式（Factory Method Pattern）负责定义一个创建对象的接口，由子类决定实例化哪个类，即让类的实例化推迟到子类，其核心思想是**将对象创建于使用分离以实现解耦**，通过子类实现具体对象的多态创建，同时使用工厂方法模式可以在新增产品类时无需修改客户端代码而实现良好扩展性

在本项目中，实现文件存储系统时采用工厂方法模式，由 `FileStorageService` 定义文件存储服务接口，由 `MinIOFileStorageService` 实现，分离对象创建

由于目前文件存储仅采用 MinIO 而没有其它系统，因此并没有体现多态创建子类，所以本设计模式不显著

工厂方法模式：文件存储系统 Java

```java
public interface FileStorageService {

    /**
     * Store a file.
     *
     * @param file     The file to be stored.
     * @param fileType The type of the file.
     * @param courseId The ID of the course associated with the file.
     * @return The stored file information in {@link ResourceFile} if stored successfully, null otherwise.
     */
    ResourceFile storeFile(MultipartFile file, ResourceFile.FileType fileType, Short courseId);

    /**
     * Get a file.
     *
     * @param file The file to be retrieved.
     * @return The file content as an {@link InputStream} or null if error.
     */
    InputStream getFile(ResourceFile file);

    /**
     * Delete a file.
     *
     * @param filePath The file path.
     * @return True if the file is deleted successfully, false otherwise.
     */
    boolean deleteFile(String filePath);

    /**
     * Calculate the file size.
     *
     * @param fileSize File size in bytes.
     * @return File size in MB.
     */
    BigDecimal calculateFileSizeMB(long fileSize);
}

@Service
public class MinioFileStorageService implements FileStorageService {...}
```

4.2.1.3. 抽象工厂模式

抽象工厂模式（Abstract Factory Pattern）提供接口用于创建相关或依赖对象的家族，而无需指定具体类，其核心思想是**构建一个产品家族以创建一组相互关联和依赖的对象**（如 UI 组件：按钮+文本框+复选框），客户端代码只通过抽象工厂的抽象接口和抽象产品交互

在本项目中，前端采用 Element-Plus 组件库，由前端直接调用组件库实现抽象工厂模式

4.2.1.4. 建造者模式

建造者模式（Builder Pattern）负责分步骤构造复杂对象，允许相同的构建过程创建不同的表示形式，其核心思想是**将对象的构造过程与对象本身分离**，通过链式调用逐步设置属性，使用者可以创建不同风格的对象而无需修改构造代码

在本项目中，由后端从 HTTP 请求中分离出用户认证身份信息时采用建造者模式，由 `SecurityUtils` 负责抽离用户认证信息，其它模块只需要调用接口即可

建造者模式：用户身份信息提取 Java

```java
@Component
public class SecurityUtils {

    /**
     * Get the current authenticated user from the security context.
     *
     * @return The current authenticated user.
     * @throws RuntimeException If the user is not authenticated or cannot be retrieved.
     */
    public static User getCurrentUser() throws RuntimeException {
        Authentication auth = SecurityContextHolder.getContext().getAuthentication();
        if (auth == null || !auth.isAuthenticated()) {
            throw new RuntimeException("用户未认证");
        }
        Object userObj = auth.getPrincipal();
        if (userObj instanceof User) {
            return (User) userObj;
        }
        throw new RuntimeException("无法获取当前用户信息");
    }
}
```

4.2.1.5. 原型模式

原型模式（Prototype Pattern）通过复制现有对象（原型）来创建新对象，而不是通过 `new` 实例化，其核心思想是**通过克隆代替新建以避免重复初始化操作**，提高性能，由运行时决定克隆对象类型

在本项目中，暂时没有体现

4.2.2. 结构型模式

结构型模式（Structural Patterns）主要关注**类和对象的组合**（如 `Adapter` 兼容接口，`Decorator` 动态扩展功能）

4.2.2.1. 适配器模式

适配器模式（Adaptor Pattern）允许不兼容接口的类能够协同工作，通过将一个类的接口转换成客户端期望的另一个接口实现不同接口适配，其核心思想是**充当两个不兼容接口间的接口转换**，通过复用现有类而无需修改原有代码即可整合新功能，客户端可以只与目标接口交互而不直接依赖被适配者

在本项目中，用户实体类 `User` 通过继承自所有实体公共基类 `Model` 并同时实现 `UserDetails` 接口以实现本项目与 SpringBoot 框架的适配

适配器模式：用户实体 Java

```java
@Entity
@Table(name = "`User`")
public class User extends Model<Integer> implements UserDetails {

    ...

    // Model methods implementation.

    @Override
    public Integer getId() {
        return id;
    }

    // UserDetails methods implementation.

    @Override
    public String getUsername() {
        return email;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.singletonList(new SimpleGrantedAuthority("ROLE_" + role.name()));
    }
```

```
45    }
```

4.2.2.2. 桥接模式

桥接模式（Bridge Pattern）将抽象部分与实现部分分离，使它们可以独立变化，通过组合关系代替继承关系，其核心思想是**解耦抽象与实现，组合优于继承和运行时绑定**

在本项目中，由 JPARepository 负责桥接项目与数据库连接，使用时只需要在配置文件中选择需要连接的数据库类型和 URL 即可完成桥接

4.2.2.3. 组合模式

组合模式（Composite Pattern）允许你将对象组合成树形结构来表示部分–整体的层次关系，使得客户端可以统一处理单个对象和组合对象，其核心思想是**通过树形结构来表示层次关系**，对单个对象和组合对象使用统一接口

在本项目中，所有的服务类都需要组合使用多个映射层的数据仓库以实现所有的服务功能

组合模式：课程服务类 Java

```java
@Service
public class CourseManager {

    private final CourseRepo courseRepo;
    private final DepartmentRepo departmentRepo;
    private final HistoryManager historyManager;

    ...
}
```

4.2.2.4. 装饰器模式

装饰器模式（Decorator Pattern）允许动态地向一个对象添加额外职责同时又不改变其结构，其核心思想是**通过创建包装对象的方式提供比继承更灵活的功能扩展**

在本项目中，由历史记录动作对象获取子服务类实现装饰器模式，向历史记录服务类中动态添加了这个服务以获取历史记录的动作操作对象，可以用于未来服务扩展

装饰器模式：历史记录动作对象获取子服务类 Java

```java
@Service
public class HistoryObjectService { ... }
```

4.2.2.5. 外观模式

外观模式（Facade Pattern）提供了一个统一的简化接口来访问复杂子系统中的多个接口，其核心思想是**为复杂系统提供单一的高层接口**，解耦客户端与子系统的直接交互，将多个步骤的操作封装为简单调用

在本项目中，所有的映射层数据访问对象都继承自 `JpaRepository` 以实现外观模式下的数据库访问，开发者只需要实现需要的 CRUD 方法函数签名即可由其自动实现而无需手写 SQL 语句

外观模式：数据访问层 Java

```java
@Repository
public interface HistoryRepo extends JpaRepository<History, Long> { ... }
```

4.2.2.6. 享元模式

享元模式（Flyweight Pattern） 通过共享对象高效支持大量细粒度对象的复用，减少内存占用，其核心思想是**将对象的内部状态和外部状态分离**，维护共享对象的缓存池以利用共享对象来减少大量相似对象的内存占用，提高对象的复用性

在本项目中，所有数据库连接操作均使用数据库连接池以实现连接对象复用，体现了享元模式

4.2.2.7. 代理模式

代理模式（Proxy Pattern）为其它对象提供一种代理以控制对这个对象的访问，代理对象在客户端和目标对象之间起到中介作用，其核心思想是**通过代理控制对目标对象的访问**

在本项目中，我们采用一个代理对象 `FileInfo` 以代理访问资源文件本身

代理模式：资源文件信息 Java

```java
public class FileInfo {

    private InputStream inputStream;
    private String fileName;

}
```

4.2.3. 行为型模式

行为型模式（Behavioral Patterns）主要关注**对象间通信**（如 `Observer` 实现事件通知， `Strategy` 封装算法族）

4.2.3.1. 责任链模式

责任链模式（Chain of Responsibility Pattern）允许你将请求沿着处理链传递，直到有一个处理者能够处理，每个处理这决定自己处理请求或将其传递给链中的下一个处理者，其核心思想是**解耦发送者与接收者使得发送者无需知道具体由谁处理**

在本项目中，所有 HTTP 请求都是首先进入路由过滤器进行过滤，然后进入 JWT 验证工具进行令牌验证，验证通过后进入各个控制器，这个过程中采用责任链模式，只有对应的控制器接受了才进入对应服务调用

4.2.3.2. 命令模式

命令模式（Command Pattern）将请求封装为独立对象，使操作者可以参数化客户端使用的不同请求，其核心思想是**解耦调用者与执行者使调用者无需知道具体执行细节**，同时把请求操作封装为包含具体信息的独立对象

在本项目中，用户活跃度分配采用命令模式，所有活跃度分配指标写在配置文件中以实现参数化客户端

命令模式：用户活跃度分配参数 YAML

```yaml
app:
  # Verification code settings.
  verification-code:
    expiration: 300000
  # User activity settings.
  activity:
    resource:
      add: 1
      delete: -1
      like: 1
      unlike: -1
      view: 1
    interaction:
      add: 1
      delete: 1
      like: 1
      unlike: -1
```

4.2.3.3. 解释器模式

解释器模式（Interpreter Pattern）定义了一个语言的文法表示，并提供一个解释器来处理这个语句中的句子，主要用于解析特定语法规则的文本输入，其核心思想是**语法解析将特定语言表示为语**

法树，通过组合多个表达式对象完成复杂解析

在本项目中，暂未体现解释器模式

4.2.3.4. 迭代器模式

迭代器模式（Iterator Pattern）用于顺序访问一个聚合对象中的各个元素而不暴露其内部的表示，为遍历不同的数据结构提供了一种统一的接口，使得客户端代码可以独立于聚合对象的内部结构进行遍历操作，其核心思想是**提供一种方式来访问聚合对象中的元素同时隐藏内部的实现细节**

在本项目中，所有列表类型的 Java 对象均体现迭代器模式

4.2.3.5. 中介者模式

中介者模式（Mediator Pattern）通过封装一组对象间的交互来降低对象间的直接耦合，使得这些对象不必显式相互引用，其核心思想是**集中控制交互的对象使得各对象只需与中介者通信**，复杂交互逻辑统一由中介者管理

在本项目中，资源文件的数据由资源实体实现中介访问

中介模式：资源文件数据 Java

```java
@Entity
@Table(name = "`Resource`")
public class Resource extends Model<Integer> {

    /**
     * Resource file class to represent a concrete file object.
     */
    @Embeddable
    public static class ResourceFile { ... }

    @Embedded
    private ResourceFile resourceFile;

}
```

4.2.3.6. 备忘录模式

备忘录模式（Memento Pattern）用于在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态允许对象在状态变化后恢复到某个特定的“快照”状态，其核心思想是**将对象的状态封装在备忘录对象中，确保状态的保存和恢复操作不会影响对象的封装性**

在本项目中，用户的点赞和撤赞是基于用户最近一次的操作的，因此需要由历史记录实体来负责追溯用户历史行为，以实现点赞或撤赞的功能

备忘录模式：历史记录追溯 Java

```java
@Service
public class ResourceManager {

    /**
     * Get the like status of a resource by a user.
     *
     * @param user The user.
     * @param resource The resource.
     * @return True if the user has liked the resource, false otherwise.
     */
    public boolean getLikeStatus(User user, Resource resource) {
        return historyManager.getLikeStatus(user, resource);
    }
}
```

4.2.3.7. 观察者模式

观察者模式（Observer Pattern）定义对象间的一对多依赖关系，当一个对象状态改变时，所有依赖它的对象都自动收到通知并更新，其核心思想是**发布–订阅机制**，由发布者维护订阅者列表

在本项目中，当后端数据更新时，前端自行重新加载后端数据

4.2.3.8. 状态模式

状态模式（State Pattern）允许对象在其内部状态改变行为时改变行为，其核心思想是**把每个状态的行为封装到独立的类中**，由上下文对象将请求委托给当前状态的对象

在本项目中暂未体现状态模式

4.2.3.9. 策略模式

策略模式（Strategy Pattern）负责定义一系列算法，封装每个算法，使它们可以互相替换，其核心思想是**封装算法实现使之与使用解耦**，同时支持运行时的动态策略选择

在本项目中，暂未体现策略模式

4.2.3.10. 模板方法模式

模板方法模式（Template Method Pattern）在父类中定义算法骨架，将某些步骤延迟到子类实现，其核心思想是**由父类控制不可变流程而子类实现可变步骤**

在本项目中，所有实体均继承自公共父类 `Model<? extends Number>` 以实现模板方法模式

模板方法模式：实体父类 Java

```java
public abstract class Model<T extends Number> {

    /**
     * Get the ID of the model.
     *
     * @return The ID of the model.
     */
    public abstract T getId();
}
```

4.2.3.11. 访问者模式

访问者模式（Visitor Pattern）负责将算法与对象结构分离，在不修改元素类的前提下定义新操作，其核心思想是**通过双重分发实现动态绑定**，将数据结构与操作逻辑分离

在本项目中，暂未体现访问者模式

---

5. 测试

> 所有测试相关详细文档均位于项目仓库的 `docs/test/` 目录下

结合所有项目需求及目前实现的项目成果，本项目主要进行了以下测试：

测试项目	测试目的	测试内容	测试结果
实体层测试	测试业务实体是否正确定义，是否符合业务逻辑，是否符合外键约束	所有业务实体	正确定义

映射层测试	测试映射层数据仓库是否正确定义，所有数据访问方法是否能获取预期结果，数据库是否正确连接	所有数据仓库	所有方法可以正确且符合预期访问数据库并得到正确结果，部分方法冗余可以删除
服务层测试	测试服务逻辑是否符合预期，结果是否正确，异常处理是否周全，代码是否健壮	• 所有主服务类 • 文件存储子服务 • 邮件子服务类	所有服务业务逻辑基本正确，部分服务未考虑高并发场景和性能优化，只能符合基本使用功能，未来可以继续版本迭代
映射层测试	测试业务路由是否正确定义，请求与响应是否正确接收与返回，身份认证是否通过，路由过滤是否正确	所有控制器	所有控制器能够正确处理请求与响应，并正确调用服务层接口
配置测试	测试配置信息是否正确读取，配置信息是否正确加载到服务端	• 应用程序专用配置 • MinIO 配置 • JWT 配置 • 数据库配置 • 邮件配置	配置信息读取正确无误，程序可以正常加载配置信息
集成测试	验证多个组件是否协调工作，是否正确处理组件间连接异常	MinIO 服务、数据库访问、前后端协作	系统协作运行成功
边界测试	测试系统对边界情况是否正确处理	空值、特殊字符等特殊情况	基本正确

---

6. 总结

在验收期附近，项目的基本产品需求均已实现，基本功能完备，满足《软件工程》验收标准

本项目最初立意不仅限于《软件工程》专业课，而是希望能够推广到真正落地应用的场景，真正上线于高校的网络平台，真正惠及每一位在校大学生，然而当前版本下项目规模较小、健壮性不足，仅限于小规模演示，许多高阶需求并未实现，项目的性能和完备性不足，有很大的提升空间

未来项目继续维护时可以考虑以下方面的发展：

- 高并发访问控制
- 性能优化
- 复杂功能实现
- 生产部署