

**程序设计实践**

**详 细 设 计 报 告**



**题目: BUPT HUB论坛**

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | **学号** | **学院** | **班级** |
| **刘兆宏** | **2021213579** | **人工智能学院** | **2021219113** |
| **沈尉林** | **2021213595** | **人工智能学院** | **2021219113** |
| **李小芊** | **2021213633** | **人工智能学院** | **2021219113** |

**2023年5月**

目录

**[1、系统概述 3](#_Toc16636)**

[1.1、系统简介 3](#_Toc3068)

1.2、术语表 3

1.3、系统运行环境 3

[1.4、开发环境 3](#_Toc9248)

**[2、数据结构说明 4](#_Toc758)**

[2.1、常量 4](#_Toc30882)

[2.2、变量 5](#_Toc30882)

[2.3、数据结构 5](#_Toc30882)

**[3、模块设计 5](#_Toc2375)**

[3.1、软件结构 5](#_Toc18888)

[3.2、功能设计说明 5](#_Toc18888)

[3.3、模块1 6](#_Toc18888)

[3.4、模块2](#_Toc18888) 8

[3.6、模块3 1](#_Toc18888)1

[3.7、模块5 1](#_Toc18888)3

[3.8、模块6 1](#_Toc18888)9

[3.9、模块7 2](#_Toc18888)3

**[4、接口设计 27](#_Toc29113)**

[4.1、 用户接口 27](#_Toc22543)

[4.2、 外部接口 2](#_Toc21053)7

[4.3、 内部接口 2](#_Toc31255)7

**[5、数据库设计 2](#_Toc9612)9**

**[6、系统出错处理 3](#_Toc18318)1**

[6.1 出错信息 31](#_Toc14297)

[6. 补救措施 3](#_Toc14297)2

**[7、其他设计 3](#_Toc3090)2**

[7.1 系统安全设计 3](#_Toc14297)2

[7.2 性能设计 3](#_Toc7610)3

# 1、系统概述

**1.1、系统简介**

BuptHub是一个小巧轻量、集发布和讨论于一体的论坛，旨在加强学生交流互助、促进知识共享和学术讨论。本论坛具有以下功能：

1、论坛用户可以注册和登录，并能够发布帖子、查看系统消息

2、论坛用户可以对论坛上的帖子进行评论、点赞和收藏

3、管理端可以对用户发送系统消息和禁用账户

4、管理端可以对论坛上的帖子和评论进行审核、排序和删除

平台的建立背景是满足学生社区的交流需求，在传统社交媒体之外开辟出具有特定环境的学生平台，使同学们拥有可以讨论课程内容、分享学习资源、解决问题的专属空间，为同学们提供一个安全、友好的学术交流环境；同时，基于北邮人论坛的使用体验，BuptHub将让用户体验更加简约流畅。

**1.2、术语表**

|  |  |  |
| --- | --- | --- |
| **序号** | **术语或缩略语** | **说明性定义** |
| 1 | Vue.js | 一款用于构建用户界面的 JavaScript 框架 |
| 2 | Vue Route | Vue.js 的官方路由 |
| 3 | Naïve UI , Element Plus | 基于 Vue 3，面向设计师和开发者的组件库 |
| 4 | Less | Leaner Style Sheets是一门向后兼容的 CSS 扩展语言 |
| 5 | Axios | 一个基于 promise 的网络请求库 |
| 6 | FastAPI | 一个使用 Python 3.6+ 用于构建 API 的 web 框架 |
| 7 | SQLAlchemy | 一个python语言实现的针对关系型数据库的ORM库 |
| 8 | ORM | 对象关系映射（Object Relational Mapping，简称ORM）是解决面向对象与关系数据库互不匹配的技术模式 |

**1.3、系统运行环境**

硬件平台：Legion Y7000P 2019、Dell G15 5511、MacBook Air(M1)

操作系统：Windows、Mac OS、Linux

前端框架：Vue.js + Vue Router + Naïve UI + Element Plus + Less + Axios

后端框架：FastAPI+SQLAlchemy

数据库：Sqlite

编程平台：VSCode

网络协议：TCP/IP

**1.4、开发环境**

工程工具：

集成开发环境：VSCode、PyCharm

版本控制工具：Git

自动化构建工具：Vite

调试工具：Volar、Unicorn、Chrome浏览器

截止到项目发布时的最新版

开发语言：

Vue：3.2.47

Python：3.9.12

# 2、数据结构说明

2.1、常量

- SECRET\_KEY：

常量名称：SECRET\_KEY

所在目录：backend/app/core/security.py

常量取值：生成的32位随机密钥

功能说明：用于生成安全令牌（token）的密钥。

- ALGORITHM：

常量名称：ALGORITHM

所在目录：backend/app/core/security.py

常量取值：HS256

功能说明：指定生成 JWT token 使用的算法，使用 HMAC-SHA256 算法进行签名。

- ACCESS\_TOKEN\_EXPIRE\_MINUTES：

常量名称：ACCESS\_TOKEN\_EXPIRE\_MINUTES

所在目录：backend/app/core/security.py

常量取值：60 \* 24 \* 7

功能说明：指定 JWT token 的过期时间，单位为分钟。在本程序中设置为7天有效期。

- UPLOAD\_FOLDER：

常量名称：UPLOAD\_FOLDER

所在目录：backend/app/routers/users.py

常量取值：文件路径 "../frontend/demo1/public/static/avatar/"

功能说明：指定用户头像文件的上传路径，即将用户上传的头像文件存储在该路径下。

- SQLALCHEMY\_DATABASE\_URL：

常量名称：SQLALCHEMY\_DATABASE\_URL

所在目录：backend/app/sql\_app/database.py

常量取值："sqlite:///./sql\_app.db"

功能说明：指定 SQLite 数据库文件的路径，以及数据库的名称。

- emailh：

常量名称：emailh

所在目录：backend/app/mail.py

常量取值：smtp.qq.com

功能说明：指定邮箱服务器地址，即用于发送电子邮件的服务器的地址。

- emailu：

常量名称：emailu

所在目录：backend/app/mail.py

常量取值：邮箱用户名

功能说明：指定邮箱用户名，即用于发送电子邮件的邮箱的用户名。

- emailpsw：

常量名称：emailpsw

所在目录：backend/app/mail.py

常量取值：邮箱密码

功能说明：指定邮箱密码，即用于发送电子邮件的邮箱的密码。

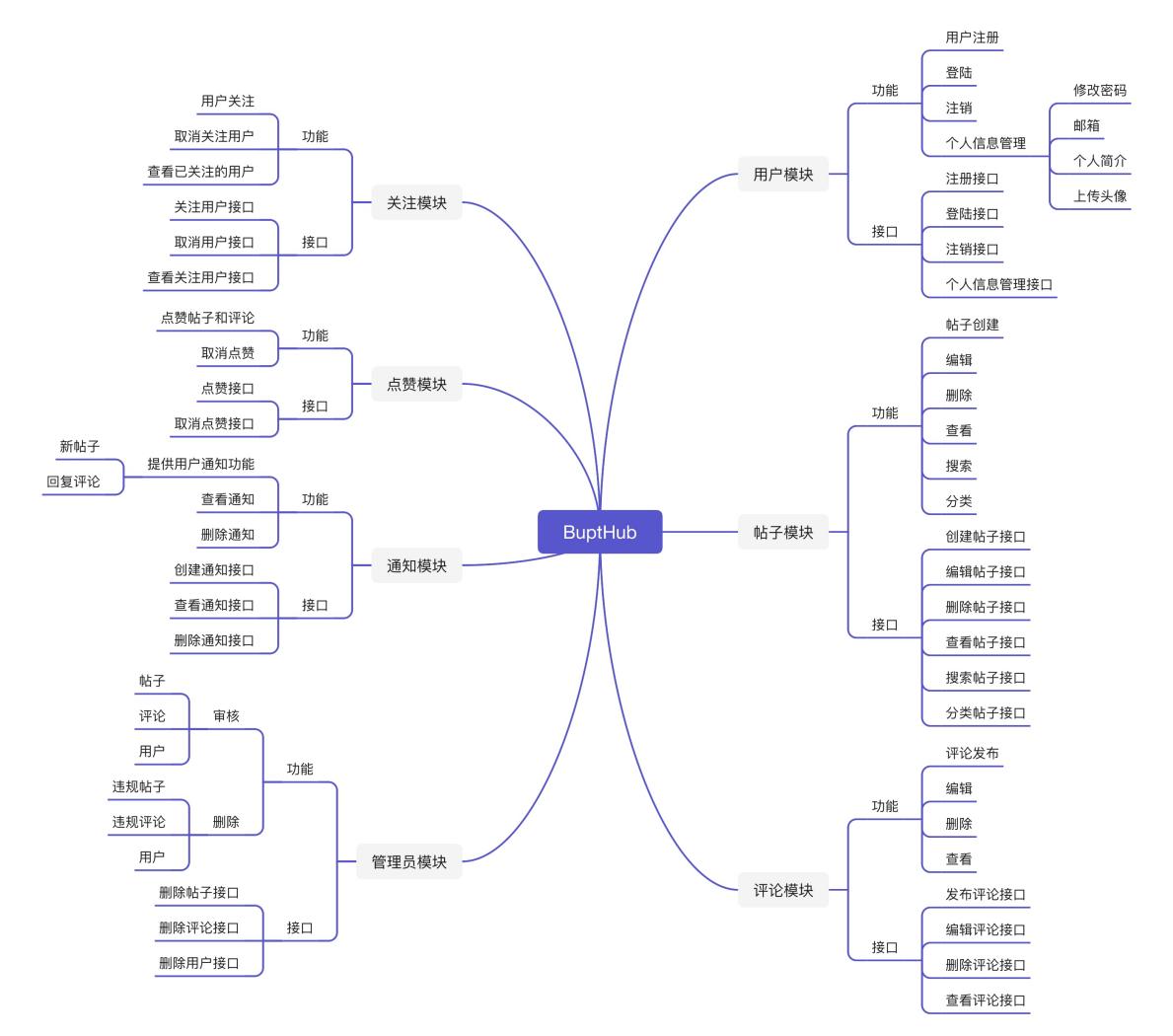
2.2、变量

2.3、数据结构

变量和数据结构的详细信息请参考5.数据库设计章节

# 3、模块设计

**3.1、软件结构**

****

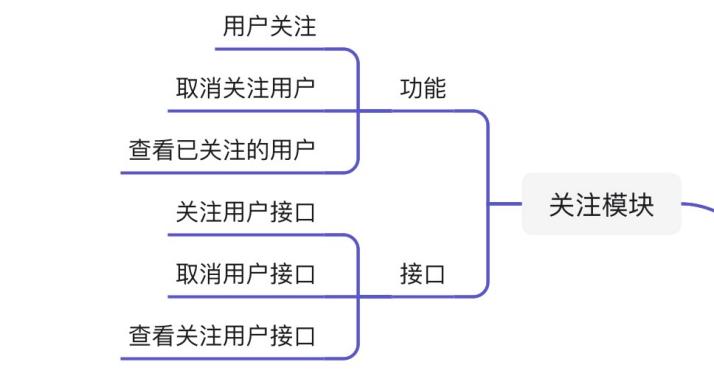
**3.2、功能设计说明**

本论坛项目的基本设计思想和理念是建立一个用户友好、功能丰富的论坛平台，使用户能够方便地进行帖子发布、交流和互动。以下是功能设计说明：

1. 模块化设计：系统采用模块化设计，将功能划分为不同的模块，每个模块负责特定的功能。这种模块化设计使系统结构清晰，便于维护和扩展。
2. 用户中心：用户是系统的核心，因此设计了用户模块来处理用户的注册、登录、注销以及个人信息管理。通过用户中心，用户可以轻松管理自己的账号和个人信息。
3. 帖子管理：帖子模块负责帖子的发布、编辑、删除、查看和搜索功能。用户可以发布自己的帖子，编辑已发布的帖子，并且可以按照关键词进行帖子搜索。这样可以促进用户之间的交流和分享。
4. 评论功能：评论模块允许用户对帖子进行评论，并提供编辑和删除评论的功能。通过评论功能，用户可以表达自己对帖子的观点和建议，增加用户之间的互动和讨论。
5. 用户关系管理：关注模块允许用户之间建立关注关系，用户可以关注其他用户，并查看自己已关注的用户。这样可以方便用户关注感兴趣的内容和用户。
6. 点赞功能：点赞模块允许用户对帖子和评论进行点赞，用户可以表达对内容的喜爱和支持。同时，用户也可以取消已点赞的帖子或评论。
7. 通知系统：通知模块提供系统向用户发送通知的功能，包括新帖子、回复评论等。用户可以方便地查看收到的通知，并及时了解到与自己相关的动态。
8. 管理员权限：管理员模块提供管理员对帖子、评论和用户的管理权限。管理员可以删除违规的帖子和评论，以及处理违规用户。

**3.3、模块1**

3.3.1、设计图



3.3.2、功能描述

关注模块允许用户关注其他用户，取消关注已关注的用户，并查看已关注的用户列表。

3.3.3、输入数据

1. 输入数据：

- 关注用户接口：当前用户ID和要关注的用户ID。

- 取消关注接口：当前用户ID和要取消关注的用户ID。

2. 有效性检验规则：验证用户ID的有效性，确保用户存在且有效。

3.3.4、输出数据

关注模块的输出数据包括执行操作后的成功消息或错误消息。

3.3.5、数据设计

用户关注关系：存储用户之间的关注关系，包括关注者ID和被关注者ID。

相关数据库表和数据存储设计会在5.数据库设计中详细说明

3.3.6、算法和流程

1. 关注用户接口：验证用户ID的有效性，检查关注关系是否已存在，如果不存在，则在用户关注关系表中创建新的关注关系。

2. 取消关注接口：验证用户ID的有效性，检查关注关系是否存在，如果存在，则在用户关注关系表中删除关注关系。

3. 查看关注用户接口：根据当前用户ID在用户关注关系表中检索并返回关注的用户列表。

3.3.7、函数说明

A. 函数名称：create\_follow

- 文件：follow.py

- 功能：关注指定用户

- 格式：

async def create\_follow(

*follow*: schemas.FollowCreate,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- follow: 关注关系的信息，包括被关注用户的ID。

- current\_user: 当前登录用户的信息。

- db: 数据库会话对象。

- 全局变量：无

- 局部变量：db\_follow (数据库中的关注关系)

- 返回值：成功创建关注关系后返回关注关系的信息。

- 算法说明：

1. 验证当前用户的身份和权限。
2. 创建关注关系并将其存储在数据库中。
3. 函数名称：delete\_follow

- 文件：follow.py

- 功能：取消关注指定用户

- 格式：

async def delete\_follow(

*follow*: schemas.FollowBase,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- follow: 关注关系的信息，包括被关注用户的ID。

- current\_user: 当前登录用户的信息。

- db: 数据库会话对象。

- 全局变量：db\_follow (数据库中的关注关系)

- 局部变量：无

- 返回值：成功取消关注后返回操作成功的消息。

- 算法说明：

1. 验证当前用户的身份和权限。
2. 根据关注者ID和被关注者ID查询关注关系。
3. 如果关注关系存在，则从数据库中删除该关注关系。
4. 函数名称：get\_followed

- 文件：follow.py

- 功能：获取当前用户关注的所有用户

- 格式：

async def get\_followed(

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- current\_user: 当前登录用户的信息。

- db: 数据库会话对象。

- 全局变量：无

- 局部变量：无

- 返回值：返回当前用户关注的所有用户列表。

- 算法说明：

1. 验证当前用户的身份和权限。
2. 根据当前用户的ID在数据库中获取关注的用户列表。
3. 函数名称：get\_follower

- 文件：follow.py

- 功能：获取关注当前用户的所有用户（粉丝）

- 格式：

async def get\_follower(

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- current\_user: 当前登录用户的信息。

- db: 数据库会话对象。

- 全局变量：无

- 局部变量：无

- 返回值：返回关注当前用户的所有用户（粉丝）列表。

- 算法说明：

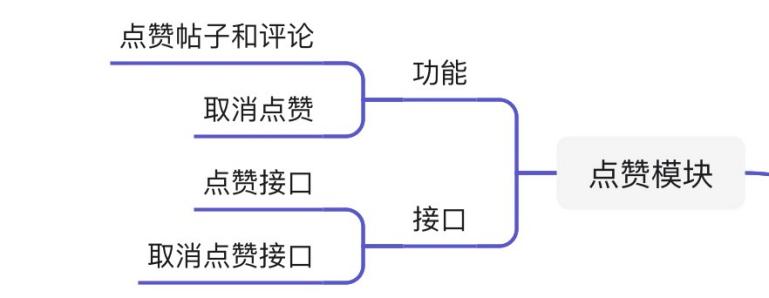
1. 验证当前用户的身份和权限。
2. 根据当前用户的ID在数据库中获取粉丝列表。

3.3.8 全局数据结构与该模块的关系

关注模块访问数据库中的关注表，以获取和更新用户关注的相关信息。该模块与全局数据结构的关系主要是通过数据库访问和操作实现的。

**3.4、模块2**

3.4.1、设计图



3.4.2、功能描述

点赞模块用于实现用户对某个内容进行点赞的功能。

3.4.3、输入数据

- 用户ID：标识进行点赞操作的用户。

- 帖子ID或评论ID：标识用户对哪个评论或帖子进行点赞。

3.4.4、输出数据

- 操作成功或失败的消息。

3.4.5、数据设计

局部数据结构包括点赞信息：用于存储用户的点赞信息，包括点赞ID、用户ID、帖子ID或评论ID等。

相关数据库表和数据存储设计会在5.数据库设计中详细说明。

3.4.6、算法和流程

1. 用户发送点赞请求，提供用户ID和内容ID作为输入数据。

2. 系统验证用户身份和权限。

3. 系统检查点赞表中是否已存在该用户对该内容的点赞记录。

- 如果存在，表示用户已经点过赞，返回操作失败的消息。

- 如果不存在，执行下一步。

4. 系统在点赞表中创建一条新的点赞记录，记录用户ID和内容ID。

5. 返回操作成功的消息。

3.4.7、函数说明

A. 函数名称：create\_like

- 文件：like.py

- 功能：创建一个新的点赞记录

- 格式：

async def create\_like(

*like*: schemas.LikeCreate,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- like: schemas.LikeCreate 类型，包含点赞记录的信息

- current\_user: schemas.User 类型，当前用户的信息

- db: Session 类型，数据库会话对象

- 全局变量：无

- 局部变量：

- user\_id：当前用户的ID

- 返回值：返回新创建的点赞记录

- 算法说明：

1. 获取当前用户的ID
2. 将点赞记录插入数据库，并返回该记录
3. 使用约束：需要验证用户的身份
4. 函数名称：delete\_like

- 文件：like.py

- 功能：删除一个点赞记录

- 格式：

async def delete\_like(

*like*: schemas.LikeBase,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- like: schemas.LikeBase 类型，要删除的点赞记录信息

- current\_user: schemas.User 类型，当前用户的信息

- db: Session 类型，数据库会话对象

- 全局变量：无

- 局部变量：

- db\_like：数据库中查询到的点赞记录

- 返回值：返回被删除的点赞记录

- 算法说明：

1. 根据当前用户的ID和点赞记录的相关信息，在数据库中查询该点赞记录
2. 如果查询不到记录，返回 404 错误
3. 否则，将该点赞记录从数据库中删除，并返回该记录

- 使用约束：需要验证用户的身份

1. 函数名称：get\_post\_like\_count

- 文件：like.py

- 功能：获取文章的点赞数量

- 格式：

async def get\_post\_like\_count(

*post\_id*: int,

*db*: Session = Depends(get\_db),

):

- 参数：

- post\_id: int 类型，要获取点赞数量的文章ID

- db: Session 类型，数据库会话对象

- 全局变量：无

- 局部变量：无

- 返回值：返回文章的点赞数量

- 算法说明：

1. 根据文章ID，在数据库中查询到该文章的点赞记录
2. 返回点赞记录数量

- 使用约束：无

1. 函数名称：get\_comment\_like\_count

- 文件：like.py

- 功能：获取评论的点赞数量

- 格式：

async def get\_comment\_like\_count(

*comment\_id*: int,

*db*: Session = Depends(get\_db),

):

- 参数：

- comment\_id: int 类型，要获取点赞数量的评论ID

- db: Session 类型，数据库会话对象

- 全局变量：无

- 局部变量：无

- 返回值：返回评论的点赞数量

- 算法说明：

1. 根据评论ID，在数据库中查询到该评论的点赞记录
2. 返回点赞记录数量

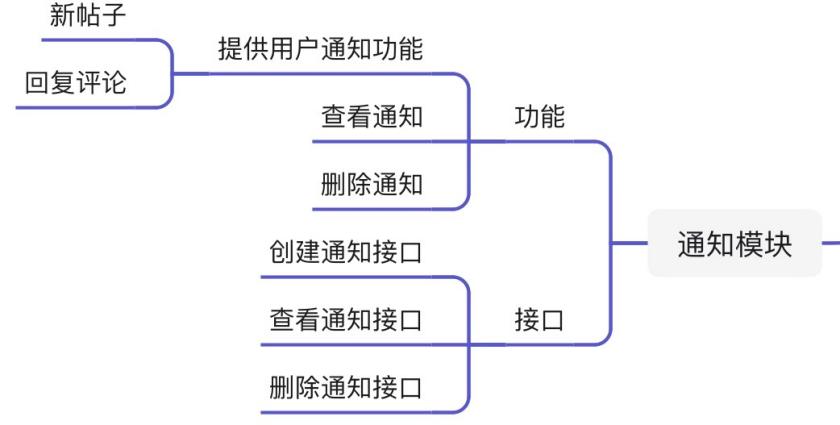
- 使用约束：无

3.4.8 全局数据结构与该模块的关系

点赞模块访问数据库中的点赞表，以获取和更新点赞的相关信息。该模块与全局数据结构的关系主要是通过数据库访问和操作实现的。

**3.5、模块3**

3.5.1、设计图



3.5.2、功能描述

通知模块负责处理系统中的通知功能，包括发送通知、接收通知和管理通知。

3.5.3、输入数据

1. 输入数据：用户发送通知的请求数据，包括通知内容、接收用户等。

2. 有效性检验规则：对用户输入的通知内容进行长度限制、格式验证等。

3.5.4、输出数据

- 发送通知的结果，如成功或失败的信息。

- 接收到的通知内容和相关信息。

3.5.5、数据设计

局部数据结构:

- 通知内容: 包含通知标题、内容、发送时间等。

- 用户信息: 包含用户ID、用户名等。

相关数据库表和数据存储设计会在5.数据库设计中详细说明。

3.5.6、算法和流程

1. 发送通知: 当用户发送通知的请求到达时，根据请求数据创建通知记录，并将通知存储到数据库中。

2. 接收通知: 当用户请求接收通知时，从数据库中检索相关的通知记录，并返回给用户。

3. 管理通知: 用户可以对通知进行管理，包括删除通知、标记通知状态等操作。

3.5.7、函数说明

A. 函数名称: get\_notification

- 文件: notification.py

- 功能：根据通知ID获取通知信息。

- 格式：

async def get\_notification(

*notification\_id*: schemas.NotificationDelete,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- notification\_id: 通知ID。

- current\_user: 当前用户的信息。

- db: 数据库会话对象。

- 全局变量：无。

- 局部变量：db\_notification用于存储从数据库中获取的通知信息。

- 返回值：返回获取的通知信息。

- 算法说明：根据提供的通知ID，从数据库中查询对应的通知信息。

- 使用约束：用户必须登录才能调用该函数。

1. 函数名称: get\_all\_notification

- 文件: notification.py

- 功能：获取当前用户的所有通知信息。

- 格式：

async def get\_all\_notification(

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- current\_user: 当前用户的信息。

- db: 数据库会话对象。

- 全局变量：无。

- 局部变量：无。

- 返回值：返回当前用户的所有通知信息。

- 算法说明：从数据库中查询当前用户的所有通知信息。

- 使用约束：用户必须登录才能调用该函数。

1. 函数名称: create\_notification

- 文件: notification.py

- 功能：创建通知信息。

- 格式：

async def create\_notification(

*notification*: schemas.NotificationCreate,

*db*: Session = Depends(get\_db),

):

- 参数：

- notification: 要创建的通知信息。

- db: 数据库会话对象。

- 全局变量：无。

- 局部变量：无。

- 返回值：返回创建的通知信息。

- 算法说明：将提供的通知信息存储到数据库中。

- 使用约束：无。

1. 函数名称: delete\_notification

- 文件: notification.py

- 功能：删除通知信息。

- 格式：

async def delete\_notification(

*notification*: schemas.NotificationDelete,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- notification: 要删除的通知信息。

- current\_user: 当前用户的信息。

- db: 数据库会话对象。

- 全局变量：无。

- 局部变量：db\_notification用于存储从数据库中获取的通知信息。

- 返回值：返回删除的通知信息。

- 算法说明：根据提供的通知ID，从数据库中删除对应的通知信息。

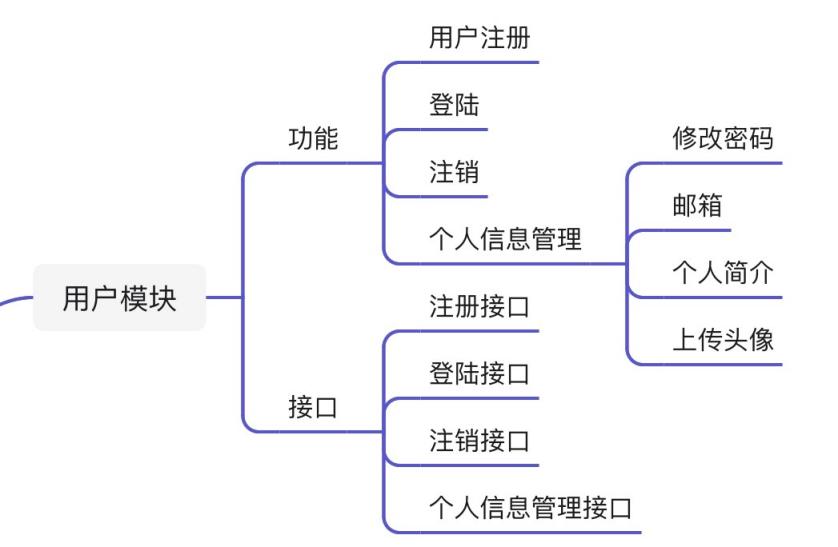
- 使用约束：用户必须登录才能调用该函数。

3.5.8 全局数据结构与该模块的关系

通知模块访问数据库中的通知表，以获取和更新点赞的相关信息。该模块与全局数据结构的关系主要是通过数据库访问和操作实现的。

**3.7、模块5**

3.7.1、设计图



3.7.2、功能描述

用户模块负责处理用户相关的功能，包括用户的注册、登录、注销和个人信息管理，以下是接口描述：

1. 注册接口：提供用户注册功能，用户可以通过注册接口创建新的账号。
2. 登录接口：提供用户登录功能，用户可以通过登录接口使用已注册的账号登录系统。
3. 注销接口：提供用户注销功能，用户可以通过注销接口退出当前登录的账号。
4. 个人信息管理接口：提供用户管理个人信息的功能，包括修改密码、邮箱、个人简介和上传头像等。

3.7.3、输入数据

1. 注册接口：需要用户提供注册所需的信息，包括用户名、密码、邮箱等。输入数据需要进行有效性检验，例如检查用户名是否已被使用、用户名是否符合要求、密码的复杂度要求、邮箱是否已被使用、邮箱是否存在等。

2. 登录接口：需要用户提供登录所需的信息，包括用户名和密码。输入数据需要进行有效性检验，检查用户名和密码是否匹配。

3. 注销接口：无需输入数据，直接执行注销操作。

4. 个人信息管理接口：根据用户需求，可能需要输入修改后的个人信息，例如新密码、新邮箱、新个人简介、新头像等。输入数据需要进行有效性检验，例如检查密码强度、邮箱格式等。

3.7.4、输出数据

用户模块在进行各项操作后，会输出相应的数据作为结果或反馈信息。具体的输出数据表现形式根据接口设计和系统需求而定，包括以下内容：

1. 注册接口的输出数据：

- 注册成功的提示信息

- 注册失败的错误信息

1. 登录接口的输出数据：

- 登录成功的身份验证信息，生成的 JWT token

- 登录失败的错误信息

1. 注销接口的输出数据：

- 注销成功的提示信息

- 注销失败的错误信息

1. 个人信息管理接口的输出数据：

- 修改个人信息成功的提示信息

- 修改个人信息失败的错误信息"

这些输出数据将作为接口调用的返回值，用于向用户提供操作结果或反馈信息。

3.7.5、数据设计

用户模块涉及的局部数据结构包括用户信息、注册信息、登录信息等。

- 用户信息：包括用户的唯一标识（ID）、用户名、密码、邮箱、个人简介、头像等。

- 注册信息：包括用户注册时提供的用户名、密码、邮箱等。

- 登录信息：包括用户登录时提供的用户名和密码。

相关数据库表和数据存储设计会在5.数据库设计中详细说明。

3.7.6、算法和流程

用户模块的算法和流程主要包括以下几个步骤：

1. 注册流程：

- 接收用户提供的注册信息。

- 验证注册信息的有效性。

- 如果验证通过，创建新的用户记录，并保存到数据库中。

- 返回注册成功的提示信息。

1. 登录流程：

- 接收用户提供的登录信息。

- 验证登录信息的有效性。

- 如果验证通过，生成登录凭证（如JWT token）作为身份验证标识。

- 返回登录成功的身份验证信息。

1. 注销流程：

- 销毁当前登录的用户凭证，使其失效。

- 返回注销成功的提示信息。

1. 个人信息管理流程：

- 根据用户需求，接收修改后的个人信息。

- 验证修改后的个人信息的有效性。

- 如果验证通过，更新用户记录中对应的信息。

- 返回修改成功的提示信息。

3.7.7、函数说明

A. 函数名称：login

- 文件：login.py

- 功能：处理用户登录请求并生成访问令牌

- 格式：

async def login(

*form\_data*: OAuth2PasswordRequestForm = Depends(), *db*: Session = Depends(get\_db)

):

- 参数：

- form\_data: OAuth2PasswordRequestForm 类型，请求表单数据，包括用户名和密码。

- db: Session 类型，数据库会话对象。

- 全局变量：

- ACCESS\_TOKEN\_EXPIRE\_MINUTES: int 类型，访问令牌的过期时间（分钟）。

- 局部变量：

- user: schemas.User 类型，根据用户名查找到的用户信息。

- email: schemas.User 类型，根据邮箱查找到的用户信息。

- access\_token\_expires: timedelta 类型，访问令牌的过期时间段。

- access\_token: str 类型，生成的访问令牌。

- 返回值：

- schemas.Token 类型，包含生成的访问令牌和令牌类型的数据结构。

- 算法说明：

1. 首先，根据用户名或邮箱从数据库中获取用户信息。
2. 检查用户是否存在，若不存在则抛出 HTTPException 错误，提示用户名或邮箱输入有误。
3. 验证密码是否正确，若不正确则抛出 HTTPException 错误，提示密码输入有误。
4. 若用户通过邮箱登录，将用户名修改为对应的邮箱用户名。
5. 根据 ACCESS\_TOKEN\_EXPIRE\_MINUTES 设置访问令牌的过期时间段。
6. 使用 create\_access\_token 函数生成访问令牌。
7. 返回包含访问令牌和令牌类型的数据结构。

注：使用约束

- 该函数依赖于数据库会话对象和验证密码的辅助函数（verify\_password、create\_access\_token）。

- 在调用该函数时需要确保正确的依赖注入，并提供有效的请求表单数据。

- 函数的返回值为生成的访问令牌和令牌类型的数据结构。

1. 函数名称：send\_verify\_code

- 文件：register.py

- 功能：发送验证码到用户的邮箱

- 格式：

async def send\_verify\_code(*Email*: schemas.Email, *db*: Session = Depends(get\_db)):

- 参数：

- Email：schemas.Email类型的参数，包含用户的电子邮件地址

- db：数据库会话对象，依赖注入方式获取

- 全局变量：无

- 局部变量：email，db\_user，origin\_code，db\_code

- 返回值：返回包含消息的字典

- 算法说明：

1. 根据提供的电子邮件地址，生成验证码并存储到数据库中。
2. 然后使用邮件发送函数mail.get\_code\_email发送电子邮件包含验证码到用户的邮箱。

- 使用约束：需要使用有效的数据库会话对象db。

1. 函数名称：register

- 文件：register.py

- 功能：处理用户注册请求

- 格式：

async def register(

*user*: schemas.UserCreate,

*Code*: schemas.Code,

*db*: Session = Depends(get\_db),

):

- 参数：

- user：schemas.UserCreate类型的参数，包含用户的注册信息

- Code：schemas.Code类型的参数，包含用户输入的验证码

- db：数据库会话对象，依赖注入方式获取

- 全局变量：无

- 局部变量：db\_code，db\_user

- 返回值：返回包含消息的字典

- 算法说明：验证用户提供的验证码是否有效，然后检查用户名是否已被注册。如果验证通过，将用户信息存储到数据库中，并删除相应的验证码记录。

- 使用约束：需要使用有效的数据库会话对象db。

1. 函数名称: send\_verify\_code

- 文件: register.py

- 功能：发送找回密码的验证码到用户的邮箱

- 格式：

async def send\_verify\_code(*Email*: schemas.Email, *db*: Session = Depends(get\_db)):

- 参数：

- Email：schemas.Email类型的参数，包含用户的电子邮件地址

- db：数据库会话对象，依赖注入方式获取

- 全局变量：无

- 局部变量：email，db\_user，origin\_code，db\_code

- 返回值：返回包含消息的字典

- 算法说明：根据提供的电子邮件地址，生成验证码并存储到数据库中。然后使用邮件发送函数mail.get\_code\_email发送电子邮件包含验证码到用户的邮箱。

- 使用约束：需要使用有效的数据库会话对象db。

1. 函数名称: find\_password

- 文件: register.py

- 功能：处理找回密码请求

- 格式：

async def find\_password(

*Code*: schemas.Code,

*password*: schemas.password,

*db*: Session = Depends(get\_db),

):

- 参数：

- Code：schemas.Code类型的参数，包含用户输入的验证码

- password：schemas.password类型的参数，包含用户要设置的新密码

- db：数据库会话对象，依赖注入方式获取

- 全局变量：无

- 局部变量：code，db\_code，email，db\_user

- 返回值：返回包含消息的字典

- 算法说明：验证用户提供的验证码是否有效，然后根据邮箱地址找到用户，并更新其密码。最后，删除相应的验证码记录。

- 使用约束：需要使用有效的数据库会话对象db。

1. 函数名称: get\_me

- 文件: user.py

- 功能：获取当前用户的信息

- 格式：

async def get\_me(*user*: schemas.User = Depends(security.get\_current\_user)):

- 参数：

- user：schemas.User类型的参数，表示当前用户

- 全局变量：无

- 局部变量：db\_user

- 返回值：返回当前用户的信息

- 算法说明：直接返回当前用户的信息

1. 函数名称: get\_user

- 文件: user.py

- 功能：根据用户ID获取部分用户信息（以JSON格式返回）

- 格式：

async def get\_user(

*user\_id*: int,

*db*: Session = Depends(get\_db),

):

- 参数：

- user\_id：int类型的参数，表示要获取信息的用户ID

- db：数据库会话对象，依赖注入方式获取

- 全局变量：无

- 局部变量：db\_user

- 返回值：返回包含部分用户信息的字典

- 算法说明：根据提供的用户ID从数据库中获取用户信息，并以字典形式返回。

1. 函数名称: update\_me

- 文件: user.py

- 功能：更新当前用户的信息

- 格式：

async def update\_me(

*user\_update*: schemas.UserUpdate,

*user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- user\_update：schemas.UserUpdate类型的参数，包含要更新的用户信息

- user：schemas.User类型的参数，表示当前用户

- db：数据库会话对象，依赖注入方式获取

- 全局变量：无

- 局部变量：db\_user

- 返回值：返回更新后的用户信息

- 算法说明：根据当前用户的ID和提供的用户更新信息，更新数据库中对应用户的信息，并返回更新后的用户信息。

1. 函数名称: upload\_avatar

- 文件: user.py

- 功能：上传用户头像

- 格式：

async def upload\_avatar(

*avatar*: UploadFile = File(...),

*user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- avatar：UploadFile类型的参数，表示要上传的头像文件

- user：schemas.User类型的参数，表示当前用户

- db：数据库会话对象，依赖注入方式获取

- 全局变量：UPLOAD\_FOLDER

- 局部变量：file\_path，db\_user

- 返回值：返回更新后的用户信息

- 算法说明：对上传的头像文件进行大小和类型的限制，然后将文件保存到指定路径。接着更新数据库中当前用户的头像

1. 函数名称: reset\_password

- 文件: user.py

- 功能：重置用户密码

- 格式：

async def reset\_password(

*origin\_passowrd*: schemas.password,

*new\_password*: schemas.password,

*user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- origin\_passowrd：schemas.password类型的参数，表示原始密码

- new\_password：schemas.password类型的参数，表示新密码

- user：schemas.User类型的参数，表示当前用户

- db：数据库会话对象，依赖注入方式获取

- 全局变量：无

- 局部变量：hashed\_password，db\_user

- 返回值：返回更新后的用户信息

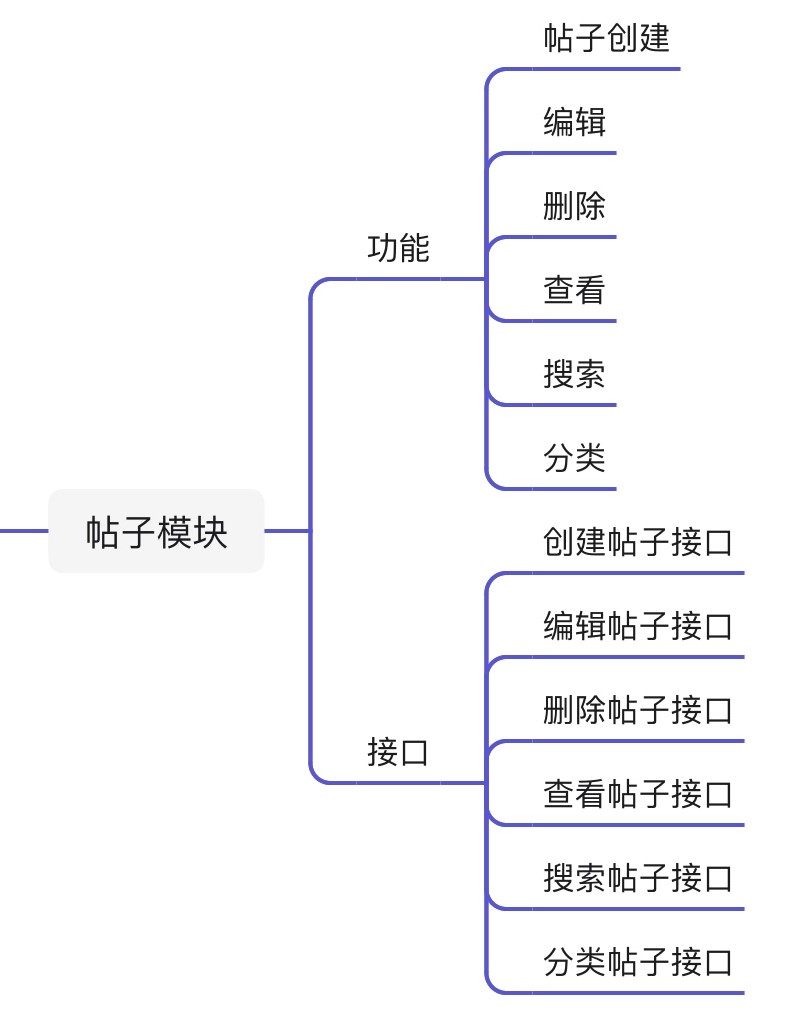
- 算法说明：验证原始密码是否正确，如果正确则对新密码进行哈希处理。然后更新数据库中当前用户的密码，并返回更新后的用户信息。

3.7.8 全局数据结构与该模块的关系

用户模块访问数据库中的用户表和验证码表，以获取和更新用户的相关信息。该模块与全局数据结构的关系主要是通过数据库访问和操作实现的。

**3.8、模块6**

3.8.1、设计图



3.8.2、功能描述

1. 创建帖子：用户可以创建新的帖子，并提供标题、内容等信息。
2. 更新帖子：用户可以更新已发布的帖子的标题、内容等信息。
3. 删除帖子：用户可以删除自己发布的帖子。
4. 查看帖子：用户可以查看已发布的帖子。
5. 搜索帖子：用户可以根据关键字搜索帖子。
6. 分类帖子：帖子可以按照分类进行分类展示。

3.8.3、输入数据

1. 输入数据：

- 帖子ID（用于获取帖子信息、更新帖子、删除帖子）。

- 帖子标题、内容、分类等信息（用于创建帖子、更新帖子）。

- 关键字（用于搜索帖子）。

2. 输入数据的有效性检验规则包括：

- 帖子ID必须是有效的整数值。

- 帖子标题不能为空且长度不能超过指定限制。

- 帖子内容不能为空且长度不能超过指定限制。

- 关键字不能为空。

3. 数据获取条件：

- 获取帖子信息需要提供帖子ID。

- 更新帖子和删除帖子需要提供帖子ID，并验证操作用户是否为帖子的发布者。

- 详细描述用户输入的数据(包括任何输入设备)以及这些数据的有效性检验规则

- 详细描述从物理模型中的哪些表获取数据以及获取这些数据的条件。

3.8.4、输出数据

帖子的详细信息，包括帖子ID、标题、内容、分类等。

3.8.5、数据设计

1. 帖子模块的数据结构包括：

- 帖子信息：包括标题、内容、分类等。

2. 相关数据库表：

- posts：存储帖子的数据表，包括帖子ID、标题、内容、分类等字段。

相关数据库表和数据存储设计会在5.数据库设计中详细说明

3.8.6、算法和流程

1. 创建帖子：

- 验证用户身份，确保用户已登录。

- 根据输入的帖子标题、内容、分类等信息创建新的帖子。

- 将帖子信息存储到数据库。

- 返回创建的帖子信息。

2. 更新帖子：

- 验证用户身份，确保用户已登录。

- 根据输入的帖子ID查询数据库，获取帖子的详细信息。

- 如果帖子不存在或当前用户不是帖子的发布者，则抛出HTTP异常，返回相应的错误信息。

- 根据输入的更新字段，更新帖子的标题、内容等信息。

- 将更新后的帖子信息存储到数据库。

- 返回更新后的帖子信息。

3. 删除帖子：

- 验证用户身份，确保用户已登录。

- 根据输入的帖子ID查询数据库，获取帖子的详细信息。

- 如果帖子不存在或当前用户不是帖子的发布者，则抛出HTTP异常，返回相应的错误信息。

- 删除帖子及相关的评论等数据。

- 返回删除成功的提示信息。

4. 查看帖子信息：

- 根据输入的帖子ID查询数据库，获取帖子的详细信息。

- 如果帖子不存在，则抛出HTTP异常，返回相应的错误信息。

- 如果帖子存在，则返回帖子的详细信息。

5. 搜索帖子：

- 根据输入的关键字查询数据库，获取符合条件的帖子列表。

- 返回符合条件的帖子列表。

6. 分类帖子：

- 查询数据库，按照分类获取帖子列表。

- 返回按照分类分类的帖子列表。

3.8.7、函数说明

A. 函数名称：get\_post

- 文件：post.py

- 功能：获取指定帖子的信息

- 格式：

async def get\_post(

*post\_id*: int,

*db*: Session = Depends(get\_db),

):

- 参数：

- post\_id: 帖子的ID

- db: 数据库会话对象

- 全局变量：无

- 局部变量：db\_post - 数据库中查询到的帖子对象

- 返回值：帖子的详细信息

- 算法说明：查询数据库中是否存在指定ID的帖子，如果存在则返回帖子的详细信息，否则抛出HTTP异常

- 使用约束：需要验证用户权限

1. 函数名称：create\_post

- 文件：post.py

- 功能：创建新的帖子

- 格式：

async def create\_post(

*post*: schemas.PostBase,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- post: 帖子的基本信息

- current\_user: 当前登录用户的信息

- db: 数据库会话对象

- 全局变量：无

- 局部变量：无

- 返回值：创建的帖子信息

- 算法说明：验证用户身份，根据输入的帖子信息创建新的帖子，并将帖子信息存储到数据库中

- 使用约束：需要验证用户身份

1. 函数名称：update\_post

- 文件：post.py

- 功能：更新帖子信息

- 格式：

async def update\_post(

*post\_update*: schemas.PostUpdate,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- post\_update: 帖子的更新信息

- current\_user: 当前登录用户的信息

- db: 数据库会话对象

- 全局变量：无

- 局部变量：db\_post - 数据库中查询到的帖子对象

- 返回值：更新后的帖子信息

- 算法说明：验证用户身份，查询数据库中指定ID的帖子，如果帖子存在且当前用户是帖子的发布者，则更新帖子的相关信息，并将更新后的帖子信息存储到数据库中

- 使用约束：需要验证用户身份

1. 函数名称：delete\_post

- 文件：post.py

- 功能：删除帖子

- 格式：

async def delete\_post(

*post\_id*: schemas.PostDelete,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- post\_id: 帖子的删除信息

- current\_user: 当前登录用户的信息

- db:数据库会话对象

- 全局变量：无

- 局部变量：db\_post - 数据库中查询到的帖子对象

- 返回值：删除的帖子信息

- 算法说明：验证用户身份，查询数据库中指定ID的帖子，如果帖子存在且当前用户是帖子的发布者，则从数据库中删除该帖子

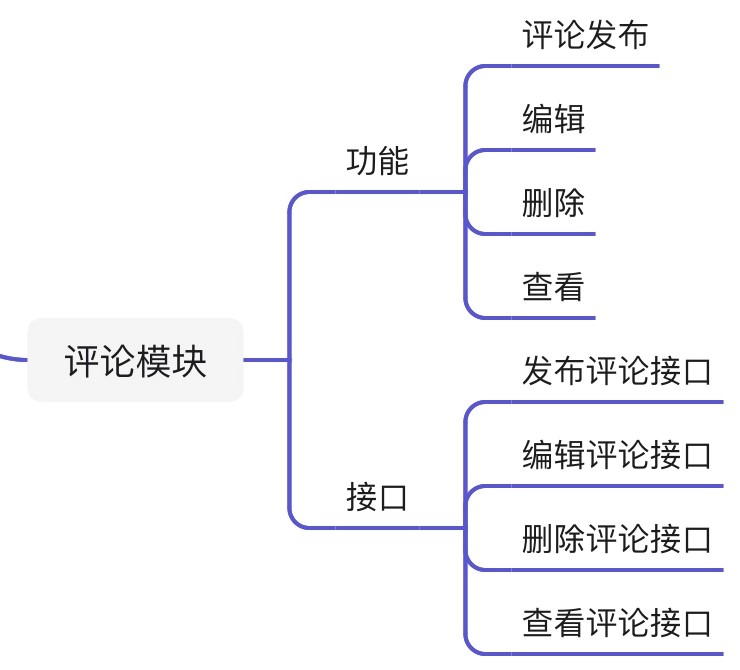
- 使用约束：需要验证用户身份

3.8.8 全局数据结构与该模块的关系

帖子模块访问数据库中的帖子表，以获取和更新帖子的相关信息。该模块与全局数据结构的关系主要是通过数据库访问和操作实现的。

**3.9、模块7**

3.9.1、设计图



3.9.2、功能描述

简要描述模块1的业务功能。

3.9.3、输入数据

1. 输入数据：用户输入的评论内容，包括评论文本、所评论的帖子或评论的ID等。

2. 数据有效性检验规则：评论内容不能为空，评论ID必须存在。

3.9.4、输出数据

评论信息：包括评论的ID、评论内容、评论所属帖子或评论的ID、评论者的用户ID等。

3.9.5、数据设计

评论信息：包括评论的ID、评论内容、评论所属帖子或评论的ID、评论者的用户ID等。

相关数据库表和数据存储设计会在5.数据库设计中详细说明。

3.9.6、算法和流程

1. 发布评论：

- 接收用户输入的评论内容和所评论的帖子或评论的ID。

- 验证评论内容的有效性。

- 创建评论对象并存储到数据库中。

2. 编辑评论：

- 接收用户输入的评论ID和新的评论内容。

- 验证评论ID的有效性。

- 更新评论对象的评论内容。

3. 删除评论：

- 接收用户输入的评论ID。

- 验证评论ID的有效性。

- 从数据库中删除评论对象。

4. 查看评论：

- 接收用户输入的评论ID。

- 验证评论ID的有效性。

- 根据评论ID从数据库中获取评论对象并返回给用户。

3.9.7、函数说明

A. 函数名称：get\_comment

- 文件：comment.py

- 功能：获取指定评论ID的评论信息

- 格式：

async def get\_comment(

*comment\_id*: schemas.CommentDelete,

*db*: Session = Depends(get\_db),

):

- 参数：

- comment\_id: schemas.CommentDelete，包含要获取的评论ID

- db: 数据库会话对象

- 全局变量：无

- 局部变量：db\_comment (models.Comment)，存储从数据库中获取到的评论对象

- 返回值：获取到的评论对象 db\_comment

- 算法说明：调用 crud.get\_comment(db, comment\_id=comment\_id.id) 获取指定评论ID的评论对象，如果评论对象不存在，则抛出 HTTPException 异常。

1. 函数名称：get\_post\_comment

- 文件：comment.py

- 功能：获取指定帖子ID的评论列表

- 格式：

async def get\_post\_comments(

*post\_id*: int,

*db*: Session = Depends(get\_db),

):

- 参数：

- post\_id: int，指定帖子ID

- db: 数据库会话对象

- 全局变量：无

- 局部变量：db\_post (models.Post)，存储从数据库中获取到的帖子对象

- 返回值：帖子的评论列表

- 算法说明：

1. 调用 crud.get\_post(db, post\_id=post\_id) 获取指定帖子ID的帖子对象。
2. 如果帖子对象不存在，则抛出 HTTPException 异常。
3. 否则调用 crud.get\_post\_comments(db=db, post\_id=post\_id) 获取帖子的评论列表。
4. 函数名称：create\_comment

- 文件：comment.py

- 功能：创建评论

- 格式：

async def create\_comment(

*comment*: schemas.CommentBase,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- comment: schemas.CommentBase，包含要创建的评论信息

- current\_user: schemas.User，当前登录的用户信息

- db: 数据库会话对象

- 全局变量：无

- 局部变量：无

- 返回值：创建的评论对象

- 算法说明：

- 调用 crud.create\_comment(db=db, comment=comment, user\_id=current\_user.id) 创建评论，并返回创建的评论对象。

1. 函数名称：delete\_comment

- 文件：comment.py

- 功能：删除评论

- 格式：

async def delete\_comment(

*comment\_id*: schemas.CommentDelete,

*current\_user*: schemas.User = Depends(security.get\_current\_user),

*db*: Session = Depends(get\_db),

):

- 参数：

- comment\_id: schemas.CommentDelete，包含要删除的评论ID

- current\_user: schemas.User，当前登录的用户信息

- db: 数据库会话对象

- 全局变量：无

- 局部变量：

- db\_comment (models.Comment)，存储从数据库中获取到的评论对象

- db\_post (models.Post)，存储从数据库中获取到的帖子对象

- db\_user (models.User)，存储从数据库中获取到的用户对象

- 返回值：无

- 算法说明：

1. 首先调用 crud.get\_comment(db, comment\_id=comment\_id.id) 获取指定评论ID的评论对象，如果评论对象不存在，则抛出 HTTPException 异常。
2. 然后调用 crud.get\_post(db, post\_id=db\_comment.post\_id) 获取评论所属的帖子对象，如果帖子对象不存在，则抛出 HTTPException 异常。
3. 接着调用 crud.get\_user(db, user\_id=db\_post.user\_id) 获取帖子的作者对象。
4. 最后判断当前用户是否有权限删除评论，若无权限则抛出 HTTPException 异常，否则调用 crud.delete\_comment(db=db, comment\_id=comment\_id.id) 删除评论。

3.9.8 全局数据结构与该模块的关系

帖子模块访问数据库中的评论表，以获取和更新评论的相关信息。该模块与全局数据结构的关系主要是通过数据库访问和操作实现的。

# 4、 接口设计

**4.1、 用户接口**

1. 用户注册接口：允许用户创建一个新的论坛账户。该接口应包括用户名、密码、电子邮件等必要的信息，并提供适当的验证和错误处理机制。

2. 用户登录接口：允许已注册用户使用其凭据登录到论坛。该接口应验证用户提供的用户名和密码，并返回一个身份验证令牌或会话标识符，以便用户进行后续请求。

3. 用户注销接口：使当前登录的用户退出登录状态。

4. 用户个人信息管理接口：提供用户个人信息的查询和更新功能。用户可以查看和编辑其个人资料、头像、签名等信息。

5. 创建帖子接口：创建新的帖子

6. 编辑帖子接口：编辑指定ID的帖子。

7. 删除帖子接口：删除指定ID的帖子。

8. 查看帖子接口：获取指定ID的帖子详情

9. 搜索帖子接口：根据关键字搜索帖子。

10. 发布评论接口：发布评论。

11. 编辑评论接口：编辑指定ID的评论。

12. 删除评论接口：删除指定ID的评论。

13. 查看评论接口：获取指定ID的评论详情。

14. 关注用户接口：用户关注其他用户的接口。

15. 取消关注接口：用户取消关注其他用户的接口。

16. 查看关注用户接口：查看关注用户接口

17. 点赞接口：给指定ID的帖子或评论点赞。

18. 取消点赞接口：取消对指定ID的帖子或评论的点赞

19. 查看通知接口：获取当前用户的所有通知

20. 删除通知接口：删除指定ID的通知。

**4.2、 外部接口**

前端和后端之间的接口采用RESTful API设计风格，使用HTTP协议进行通信。前端通过axios库向后端发送请求，后端返回JSON格式的响应。

平台与外界的交互接口：

1. Web应用程序接口：用户通过浏览器访问本系统，需要通过Web应用程序接口进行通信。Web应用程序接口使用HTTP协议，支持GET、POST、DELETE和PUT等请求方式。系统对外提供的接口包括用户模块接口、帖子模块接口、评论模块接口、关注模块接口、点赞模块接口、通知模块接口等。
2. 数据库接口：本系统需要与数据库进行交互，使用关系型数据库管理系统Sqlite。
3. 邮件发送接口：本系统需要向用户发送邮件，包括注册确认、密码重置等功能。系统使用QQ邮箱发送邮件，支持SMTP协议。

**4.3、 内部接口**

4.3.1、 接口说明

1. 用户模块：

- 注册接口：用户模块调用用户模块的注册函数进行用户注册。

- 登录接口：用户模块调用用户模块的登录函数进行用户登录验证。

- 注销接口：用户模块调用用户模块的注销函数进行用户注销操作。

- 个人信息管理接口：用户模块调用用户模块的个人信息管理函数进行用户个人信息的修改和更新。

1. 帖子模块：

- 创建帖子接口：帖子模块调用帖子模块的创建帖子函数进行帖子的创建和保存。

- 编辑帖子接口：帖子模块调用帖子模块的编辑帖子函数进行帖子的修改和更新。

- 删除帖子接口：帖子模块调用帖子模块的删除帖子函数进行帖子的删除操作。

- 查看帖子接口：帖子模块调用帖子模块的查看帖子函数进行帖子的查询和返回结果。

- 搜索帖子接口：帖子模块调用帖子模块的搜索帖子函数进行帖子的搜索和返回结果。

- 分类帖子接口：帖子模块调用帖子模块的分类帖子函数进行帖子的分类和返回结果。

1. 评论模块：

- 发布评论接口：评论模块调用评论模块的发布评论函数进行评论的发布和保存。

- 编辑评论接口：评论模块调用评论模块的编辑评论函数进行评论的修改和更新。

- 删除评论接口：评论模块调用评论模块的删除评论函数进行评论的删除操作。

- 查看评论接口：评论模块调用评论模块的查看评论函数进行评论的查询和返回结果。

1. 关注模块：

- 关注用户接口：关注模块调用关注模块的关注用户函数进行用户关注操作。

- 取消关注接口：关注模块调用关注模块的取消关注函数进行取消关注操作。

- 查看关注用户接口：关注模块调用关注模块的查看关注用户函数进行已关注用户的查询和返回结果。

1. 点赞模块：

- 点赞接口：点赞模块调用点赞服务的点赞函数，实现对帖子或评论的点赞操作。

- 取消点赞接口：点赞模块调用点赞模块的取消点赞函数，实现对帖子或评论的取消点赞操作。

1. 通知模块：

- 创建通知接口：通知模块调用通知服务的创建通知函数，实现创建新的通知消息。

- 查看通知接口：通知模块调用通知服务的查看通知函数，实现查看用户的通知消息。

- 删除通知接口：通知模块调用通知服务的删除通知函数，实现删除用户的通知消息。

4.3.2、 调用方式

1. 用户模块：

- 注册接口调用方式：发送POST请求到/register，传递用户注册信息作为请求参数。

- 登录接口调用方式：发送POST请求到/login，传递用户登录信息作为请求参数。

- 注销接口调用方式：发送POST请求到/user，不需要传递参数。

- 个人信息管理接口调用方式：发送PUT请求到/user/update，传递要修改的用户信息作为请求参数。

1. 帖子模块：

- 创建帖子接口调用方式：发送POST请求到/post/create，传递帖子内容作为请求参数。

- 编辑帖子接口调用方式：发送PUT请求到/post/update，传递帖子ID和要修改的内容作为请求参数。

- 删除帖子接口调用方式：发送DELETE请求到/posts/delete，传递帖子ID作为请求参数。

- 查看帖子接口调用方式：发送GET请求到/posts/{post\_id}，传递帖子ID作为请求参数。

1. 评论模块：

- 发布评论接口调用方式：发送POST请求到/comment/create，传递评论内容和所属帖子ID作为请求参数。

- 删除评论接口调用方式：发送DELETE请求到/comment/delete，传递评论ID作为请求参数。

- 查看评论接口调用方式：发送GET请求到/comments/{comment\_id.id}，传递评论ID作为请求参数。

1. 关注模块：

- 关注用户接口调用方式：发送POST请求到/follow/create，传递被关注用户的ID作为请求参数。

- 取消关注接口调用方式：发送DELETE请求到/follow/delete，传递被取消关注用户的ID作为请求参数。

- 查看关注用户接口调用方式：发送GET请求到/follow/get\_followed，不需要传递参数。

1. 点赞模块：

- 点赞接口调用方式：发送POST请求到/like/create，传递帖子或评论的ID作为请求参数。

- 取消点赞接口调用方式：发送DELETE请求到/like/delete，传递帖子或评论的ID作为请求参数。

1. 通知模块：

- 创建通知接口调用方式：发送POST请求到/notification/create，传递通知内容和接收用户ID作为请求参数。

- 查看通知接口调用方式：发送GET请求到/all，不需要传递参数。

- 删除通知接口调用方式：发送DELETE请求到/notification/delete，传递通知ID作为请求参数。

# 5、数据库设计

1. 本系统中使用的数据库包括以下数据表：

- 验证码表（codes）：

标识符：codes

数据项：

id: 验证码ID，整数类型，主键，自增长

code: 验证码，字符串类型，唯一

email: 电子邮箱，字符串类型，唯一

- 用户表（users）：

标识符：users

数据项：

id: 用户ID，整数类型，主键，自增长

username: 用户名，字符串类型，唯一

email: 电子邮件，字符串类型，唯一

password\_hash: 密码哈希值，字符串类型

avatar\_url: 头像URL，字符串类型

introduction: 个人介绍，字符串类型

create\_time: 注册时间，日期时间类型

update\_time: 最后一次更新时间，日期时间类型

is\_admin: 是否为管理员，布尔类型

is\_active: 是否已被激活，布尔类型

- 帖子表（posts）：

标识符：posts

数据项：

id: 帖子ID，整数类型，主键，自增长

title: 帖子标题，字符串类型

content: 帖子内容，字符串类型

create\_time: 帖子创建时间，日期时间类型

update\_time: 最后一次更新时间，日期时间类型

is\_deleted: 是否已被删除，布尔类型

user\_id: 发帖用户ID，整数类型，外键，关联用户表的id字段

- 评论表（comments）：

标识符：comments

数据项：

id: 评论ID，整数类型，主键，自增长

content: 评论内容，字符串类型

create\_time: 评论创建时间，日期时间类型

update\_time: 最后一次更新时间，日期时间类型

is\_deleted: 是否已被删除，布尔类型

user\_id: 评论用户ID，整数类型，外键，关联用户表的id字段

post\_id: 所属帖子ID，整数类型，外键，关联帖子表的id字段

parent\_id: 父评论ID，整数类型，外键，可以为空，关联评论表的id字段

- 关注表（follows）：

标识符：follows

数据项：

id: 关注ID，整数类型，主键，自增长

follower\_id: 关注者ID，整数类型，外键，关联用户表的id字段

followed\_id: 被关注者ID，整数类型，外键，关联用户表的id字段

- 点赞表（likes）：

标识符：likes

数据项：

id: 点赞ID，整数类型，主键，自增长

user\_id: 点赞用户ID，整数类型，外键，关联用户表的id字段

post\_id: 被点赞帖子ID，整数类型，外键，关联帖子表的id字段

comment\_id: 被点赞评论ID，整数类型，外键，可以为空，关联评论表的id字段

- 通知表（notifications）：

标识符：notifications

数据项：

id: 通知ID，整数类型，主键，自增长

content: 通知内容，字符串类型

create\_time: 通知创建时间，日期时间类型

is\_read: 是否已读，布尔类型

user\_id: 接收通知的用户ID，整数类型，外键，关联用户表的id字段

2. 这些表之间的关系如下：

- 用户表（users）和帖子表（posts）是一对多的关系，即一个用户可以创建多个帖子，但一个帖子只能由一个用户创建。

- 帖子表（posts）和评论表（comments）也是一对多的关系，即一个帖子可以有多个评论，但一个评论只能属于一个帖子。

- 评论表（comments）和自身之间存在一种多级关系，即一个评论可以有多个子评论（即回复），也可以没有子评论。

- 用户表（users）和关注表（follows）之间是多对多的关系，即一个用户可以关注多个其他用户，也可以被多个用户关注。

- 用户表（users）和点赞表（likes）之间也是多对多的关系，即一个用户可以给多个帖子或评论点赞，一个帖子或评论也可以被多个用户点赞。

- 用户表（users）和通知表（notifications）是一对多的关系，即一个用户可以收到多条通知，但一条通知只能发送给一个用户。

# 6、系统出错处理

**6.1、 出错信息**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **可能的错误和故障** | **输出信息的形式** | **输出信息的含义** | **处理方式** |
| 1 | 资源未找到 | 404 Not Found | 请求的资源在服务器上未找到 | 检查请求的 URL 是否正确，确保所请求的资源存在。如果资源不存在，可以返回适当的错误响应或重定向到合适的页面。 |
| 2 | 请求错误 | 400 Bad Request | 客户端发送的请求存在语法错误或无效数据 | 解析请求数据并验证其格式和内容。如果请求无效，可以返回适当的错误响应，包含错误细节和可能的修复建议。 |
| 3 | 未经授权 | 401 Unauthorized | 请求需要进行身份验证，但未提供有效的身份凭证 | 请求需要进行身份验证，但未提供有效的身份凭证。  处理方式：要求用户提供有效的身份凭证，如用户名和密码。如果仍未经授权，返回适当的错误响应或重定向到登录页面。 |
| 4 | 禁止访问 | 403 Forbidden | 请求被服务器理解，但服务器拒绝执行该请求 | 检查请求的权限和访问控制设置。如果用户没有足够的权限，返回适当的错误响应，说明原因或提供其他可行的操作。 |
| 5 | 服务器内部错误 | 500 Internal Server Error | 服务器在处理请求时遇到了意外的错误 | 记录错误详细信息，包括错误类型等，并将其显示或记录在日志中。对于用户，可以返回一个友好的错误消息，表明系统出现了问题，并建议稍后重试或与管理员联系。 |
| 6 | 实体无法处理 | 422 Unprocessable Entity | 请求的数据存在语义错误或无效而验证失败时，FastAPI 会自动生成带有错误详细信息的 422 响应，以告知客户端请求存在问题。 | 根据需求自定义错误处理器，并选择其他错误码来表示不同类型的错误。可以使用 FastAPI 的异常处理机制和自定义异常处理器来捕获和处理验证错误，以及返回适当的错误码和错误信息。 |
| 7 | 数据库错误 | 数据库查询错误、连接超时、写入冲突等 | 与数据库交互时发生错误或异常 | 捕获数据库相关的异常并进行适当处理，如重新连接数据库、重试操作，或返回与数据库错误相关的错误信息。 |
| 8 | 网络故障 | 网络连接中断、DNS 解析失败等 | 与外部服务通信时出现问题 | 检查网络连接、服务可用性和配置。对于无法访问的外部服务，可以返回一个适当的错误响应，指导用户稍后重试或检查网络连接。 |

**6.2、 补救措施**

对于每种错误和故障，将根据实际情况采取适当的处理方式，包括返回适当的错误响应、记录日志、提供用户友好的错误消息、重试操作或与管理员联系等。确保错误和故障信息能够及时捕获、记录和通知，以便及时解决问题并提供良好的用户体验。

# 7、其他设计

**7.1 系统安全设计**

1. 用户认证与授权：使用身份验证机制来验证用户身份，使用OAuth 2.0 和 Bearer JWT来生成和验证访问令牌，并确保只有经过身份验证和授权的用户才能访问受保护的资源。

密码安全：采用适当的密码存储和加密方式，使用哈希函数bcrypt对用户密码进行加密，确保用户密码不以明文形式存储在数据库中。

1. 输入验证与过滤：在前后端都进行输入验证，过滤用户输入的数据，防止恶意输入、SQL注入等攻击。
2. 安全漏洞防护：定期更新框架和依赖库，及时修复安全漏洞。限制对敏感接口和资源的访问。

**7.2 性能设计**

1. 数据库性能优化：使用适当的索引和查询优化来提高数据库查询性能。合理设计数据库表结构，避免数据冗余和不必要的关联。
2. 前端性能优化：使用前端技术和工具来优化页面加载速度。
3. 后端性能优化：使用异步处理来提高请求处理能力。
4. 代码优化：优化代码逻辑，避免不必要的计算和数据库查询操作。使用合适的数据结构和算法来提高代码效率。
5. 异常处理与报警：及时捕获和处理异常，发送报警通知以便及时响应和解决问题。