2023_ECNU_PJ2_报告

10215501415 龙羿霏

一、目录结构

本次项目采用了MVC（Model-View-Controller）这种常见的软件架构模式。在MVC模式中，应用程序被分为三个主要组件：模型（Model）、视图（View）和控制器（Controller），这三个组件各自承担不同的责任，便于实现分层和松耦合的设计，以促进代码的可维护性和可扩展性。

这次实验中，将Model层代码放在 `/be/model` 文件夹下，将View层代码放在 `/be/view` 文件夹下，将Controller层代码放在 `/fe/access` 文件夹下。此外，功能测试的代码放在了 `/fe/test` 文件夹下。

```
.
├─be
|  |  app.py
|  |  serve.py
|  |  __init__.py
|  |
|  ├─model
|  |  |  buyer.py
|  |  |  db_conn.py
|  |  |  error.py
|  |  |  seller.py
|  |  |  store.py
|  |  |  user.py
|  |  |  __init__.py
|  |
|  └─view
|     |  auth.py
|     |  buyer.py
|     |  seller.py
|     |  __init__.py
|
├─fe
|  |  conf.py
|  |  conftest.py
|  |  __init__.py
|  |
|  ├─access
|  |     auth.py
|  |     book.py
|  |     buyer.py
|  |     new_buyer.py
|  |     new_seller.py
|  |     seller.py
|  |     __init__.py
|  |
|  ├─bench
|  |     run.py
|  |     session.py
|  |     workload.py
|  |     __init__.py
|  |
|  ├─data
|  |      book.db
|  |      book_lx.db
|  |      scraper.py
|  |
|  └─test
|        gen_book_data.py
|        test_add_book.py
|        test_add_funds.py
|        test_add_stock_level.py
|        test_bench.py
|        test_buyer_cancel_order.py
|        test_create_store.py
|        test_deliver_book.py
```

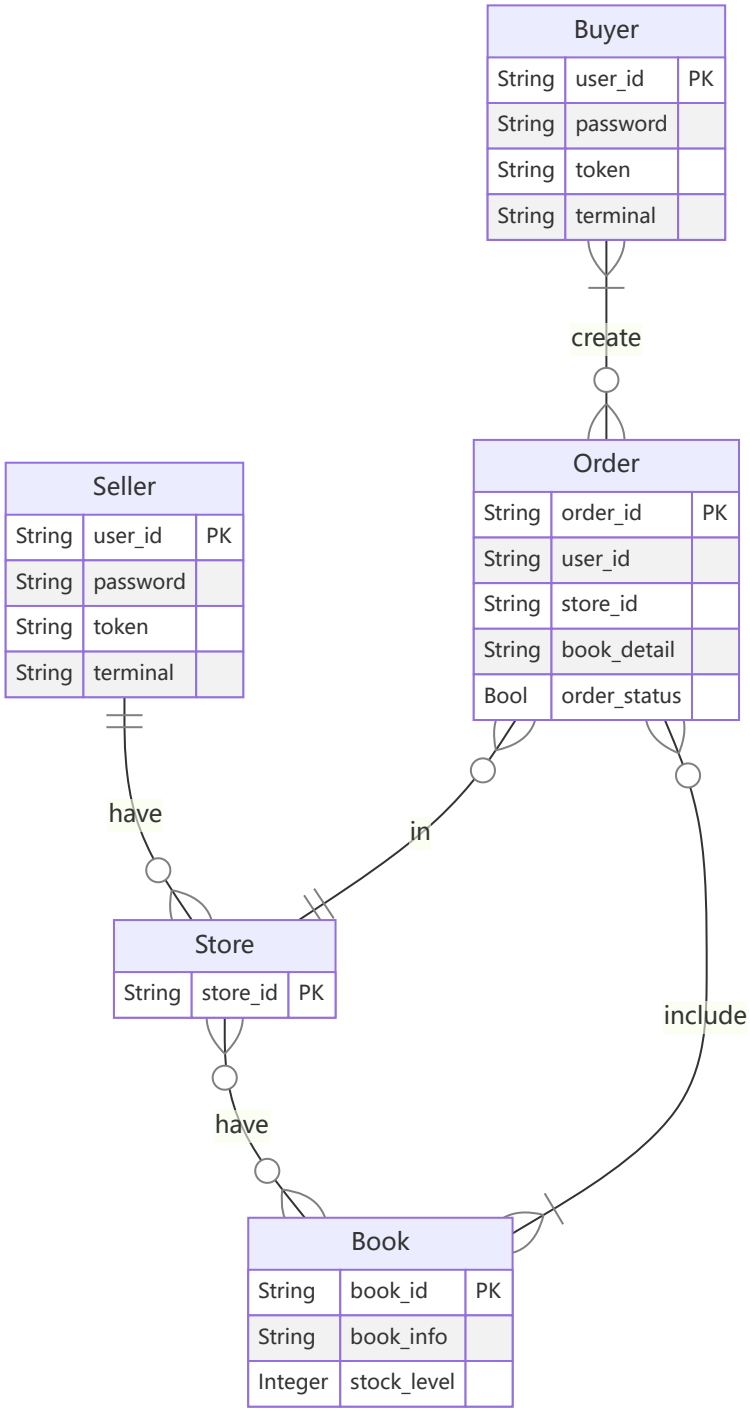
```
|        test_login.py
|        test_new_order.py
|        test_overtime_cancel_order.py
|        test_password.py
|        test_payment.py
|        test_receive_book.py
|        test_register.py
|        test_search_book.py
|        test_search_history_order.py
|
└─script
        test.sh
```

二、关系数据库设计

1. ER图

- 对于用户，他可以注册、注销、登录、登出、更改密码。
- 对于买家，他可以充值、创建订单、取消订单、付款、收货、搜索书籍、搜索历史订单。
- 对于卖家，他可以创建店铺、添加书籍信息、添加书籍库存、发货。

基于上述逻辑绘制ER图，然后利用ER图构建关系模型（连线具体含义参照mermaid语法）。



2. 从ER图到关系型schema

由于书籍封面图片blob数据不适合使用表结构进行存储，因此将其存储在本地MongoDB文档数据库，其余核心数据使用**PostgreSQL数据库**进行存储。

主键及约束关系会在Model层中的 `store.py` 部分详细介绍。

根据ER图，为关系型数据库设计以下**7个数据表**：

- **user：用户**
  - `user_id`：用户id，设置为主键，不可为空，不可重复
  - `password`：用户密码，不可为空

    - `balance`：用户账户余额，初始值为0，不可为空
    - `token`：用户登录token
    - `terminal`：用户登录的终端号
- **user_store：商家及其店铺**
    - `store_id`：店铺id，不可为空
    - `user_id`：商家id，外键关联user表的 `user_id`，不可为空

  `store_id` 和 `user_id` 均为主键，表明商家与店铺的一对一关系。
- **store：店铺内书籍信息**
    - `store_id`：店铺id，不可为空
    - `book_id`：书籍id，不可为空
    - `book_info`：书籍详细信息（json字符串）
    - `stock_level`：书籍库存量

  `store_id` 与 `book_id` 均为主键，表明同一本书在同一家店中只可以被上架一次（一对一关系），但同一本书可以被多家店同时上架，同时一个商家也可以上架多种书。
- **new_order：新创建订单**
    - `order_id`：订单编号，设置为主键，不可重复，不可为空
    - `user_id`：下单买家id，外键关联user表的 `user_id`
    - `store_id`：下单店铺id
- **new_order_detail：新订单内容**
    - `order_id`：订单编号，设置为主键，不可为空
    - `book_id`：该笔订单包含的书籍id，设置为主键，不可为空
    - `count`：该种书下单数量
    - `price`：该种书单价

  `order_id` 和 `book_id` 均为主键，该表每行只记录一个订单中某一种书的下单信息，同一笔订单中可以同时下单多种书籍，因此需要 `order_id` 和 `book_id` 同时进行区分。
- **history_order：历史订单详细信息**
    - `order_id`：订单编号，设置为主键，不可为空
    - `store_id`：下单店铺id
    - `user_id`：下单买家id，设置为主键，不可为空
    - `book_id`：该笔订单包含的书籍id，每行数据只记录一种书籍
    - `book_count`：该种书下单数量
    - `price`：该种书单价
    - `is_cancelled`：该订单是否被取消，布尔变量
    - `is_paid`：该订单是否被支付，布尔变量
    - `is_delivered`：该订单是否发货，布尔变量
    - `is_received`：该订单是否被收货，布尔变量

  与new_order_detail相同， `order_id` 和 `book_id` 均为主键，该表每行只记录一个订单中某一种书的下单信息，同一笔订单中可以同时下单多种书籍。
- **book_detail：在售书籍详细信息**
    - `book_id`：书籍id，设置为主键，不可为空，不可重复
    - `tilte`：书籍标题
    - `author`：书籍作者
    - `book_intro`：书籍简介
    - `content`：书籍目录

- `tags`：书籍标签

3. 数据库改动

(1) 改动内容

这七个表中，大多数的转换是直接由文档结构中的字段对应到表中的属性，然后分别进行存储。这里从文档型数据库转化为关系型数据库做出的较大改动有：

- **模糊搜索**

  这里主要是在用户搜索书籍功能处，如果使用文档数据库，就需要先对所有文字信息进行分词，然后添加 `description` 属性，将该字符串进行存储，后续使用文本索引进行模糊搜索；使用关系数据库后直接进行存储，查询时使用 `ilike` 进行模糊搜索即可。

- **多表联合查询**

  这里也是在用户搜索书籍功能处，如果使用文档数据库，只能使用单表逐步查询，将上一步的筛选结果作为下一步的筛选条件传入进行筛选；而使用关系数据库后，只需要构建好所有查询条件，使用 `join` 方法将需要的表进行连接，即可在连接后的表进行一次查询得到结果。

- **ORM实现事务处理**

  ORM通过抽象，将事务管理的细节包装起来（比如处理失败回滚操作），提供了更高层次的接口，保证了数据的一致性和完整性，方便编写代码。

(2) 改动原因

- **提升模糊搜索功能**
  - 在文档数据库中，对于简介等文本信息，首先要进行分词，然后在数据库中添加一个冗余属性 `decription` 进行存储，而改用关系型数据库后可以抛弃掉这一冗余属性，减少了存储空间的消耗。
  - 分词操作有时会产生不正确的分词结果，因此即使文本中存在该内容，也可能因为不正确的分词而搜索失败；而使用 `ilike` 后，可以直接进行字符串匹配，提高了搜索精度。
- **多表联合查询简化流程**

  在文档数据库中，涉及到多个集合的信息时，只能单个集合查询，然后将查询结果作为下一次的查询条件传入，这大大增加了查询条件代码编写的复杂性；而使用关系数据库后，通过将多个数据表连接，可以在一次查询就得到想要的结果，简化代码编写流程。

- **ORM实现事务处理**

  关系数据库支持事务，保证了操作的ACID属性，这对于购书网站这种需要对账户余额和库存数量有精确要求的应用至关重要。而ORM提供了高级接口来自动实现事务处理，简化了代码编写流程的同时保证了数据的一致性、完整性。

- **提升查询效率**
  - 关系数据库使用SQL查询语言，并针对表之间的关联设计了优化算法和索引策略，使得复杂查询和表连接更为高效。
  - 从最终的测试运行时间可以看出这一点，上次全部使用MongoDB数据库总运行时间大概在8分钟左右，而这次将核心数据库改为PostgreSQL后，总运行时间降到了4分钟左右，运行效率有了明显的提升。
- **数据表达更直观清晰**
  - 关系数据库采用表格形式存储数据，这种标准化的数据模式使得数据更易于管理、理解和维护。这里用户、书籍、订单等信息均有比较一致的格式，因此使用关系数据库更容易维护。

- 关系数据库在编写业务逻辑代码时能够比文档数据库更加直观地处理数据，因为关系数据库的表结构可以直接映射到业务模型的实体和关系，每个表代表一个实体，表中的列对应实体的属性，表之间的关联对应业务中的关系，有利于处理业务逻辑。

三、功能介绍

本次实验对于PostgreSQL数据库的操作均使用了ORM（sqlalchemy）。

1. Model层接口

Model层接口提供对数据库的原子操作，后续由View层和Access层调用这些接口来相应前端的请求。

(1) `store.py`

该文件主要用来**初始化书店网站的后端及其数据库**，主要是 `Store` **类**的建立与初始化。

- `__init__`：建立与本地PostgreSQL数据库的连接，这里使用 `be` 数据库；

接下来在 `init_collections` 中建立后续代码中会使用到的数据表和集合，其中关系数据库的主键默认会建立一个索引，多个主键会默认建立一个复合索引。

- `user`：存放已经注册的用户信息
  - 在 `user_id` 上建立主键，同时设置非空和不重复约束，防止重复的用户数据插入，同时索引可以加快后续通过 `user_id` 查找用户的速度。
- `user_store`：存放用户（卖家）id及其拥有的店铺id信息
  - 为 `user_id` 和 `store_id` 同时建立主键，同时设置非空和不重复约束，每个卖家与自己的店铺是绑定在一起的关系（所以查询时经常是两个信息组合出现，复合索引可以加快查询速度），允许用户与店铺存在一对多或多对一的关系。
- `store`：存放店铺中在售的书籍id及其库存量
  - 为 `store_id` 和 `book_id` 同时设置主键，同时设置非空和不重复约束，因为添加书籍或修改书籍库存时通常将这两个信息同时输入，复合索引更有速度上的优势。同一本书在同一家店中只可以被上架一次（一对一关系），但同一本书可以被多家店同时上架，同时一个商家也可以上架多种书。
- `new_order`：存放用户（买家）新创建订单信息
  - 在 `order_id` 上设置主键，同时设置非空和不重复约束，保证同一笔订单只插入一次，同时索引可以加快根据订单号对用户及店家信息的查找。
- `new_order_detail`：存放某笔订单中某本书的详细订单信息
  - 将 `order_id` 和 `book_id` 同时设置为主键，并且设置非空和不重复约束，每行只记录一个订单中某一种书的下单信息，同一笔订单中可以同时下单多种书籍，因此需要 `order_id` 和 `book_id` 同时进行区分。
- `history_order`：控制订单后续状态
  - 将 `order_id` 和 `book_id` 同时设置为主键，并且设置非空和不重复约束，该表每行只记录一个订单中某一种书的下单信息，同一笔订单中可以同时下单多种书籍。
- `book_detail`：所有店铺在售书籍的详细信息
  - 将 `book_id` 作为主键，并设置非空和不可重复约束，只要有一家店铺上架过该书，就不需要重复加入信息。

接下来通过实例化一个 `Store` 类，返回数据库接口，方便后续文件使用。

(2) `db_conn.py`

该文件主要初始化数据库连接，然后定义了一些到数据库中**验证存在性**的基本操作。

- `user_id_exist(user_id)`：**用户是否存在**

  根据传入 `user_id`，到 `user` 表中找到 `user_id` 对应记录，如果成功，则用户存在。

- `book_id_exist(store_id, book_id)`：**书籍是否在售**

  根据传入 `store_id` 和 `user_id`，到 `store` 表中寻找对应记录，如果成功，则该家店铺存在该在售书籍。

- `store_id_exist(store_id)`：**店铺是否存在**

  根据传入 `store_id`，到 `user_store` 表中寻找对应记录，如果成功，则该家店铺存在。

(3) `user.py`

该文件主要在 `User` 类中定义了一些**基本的用户操作**，后续操作中买家和卖家都会使用到。

- `register(user_id, password)`：**用户注册**

  用户传入注册所需基本信息（用户名、密码），然后判断到 `user` 表中查找该用户名是否已经存在：

  - 若不存在，系统为其自动生成terminal值和token，并默认账户余额为0，存入表中。
  - 若用户名已经存在，则用户不能成功注册，需要更改用户名重新尝试。

- `login(user_id, password, terminal)`：**用户登录**

  将 `user_id` 和密码传入 `check_password` 函数，到 `user` 表中验证二者是否匹配：

  - 如果不匹配或者该用户不存在，登录失败。
  - 如果匹配，登录成功，自动为用户生成登录token并存入数据库。

- `logout(user_id, token)`：**用户登出**

  用户传入 `user_id` 和登录时产生的token，调用 `check_token` 函数到 `user` 表中验证二者是否匹配：

  - 如果不匹配或者该用户不存在，登出失败。
  - 如果匹配，登出成功，更新对应token。

- `unregister(user_id, password)`：**用户注销**

  将 `user_id` 和密码传入 `check_password` 函数，到 `user` 表中验证二者是否匹配：

  - 验证失败则直接返回。
  - 验证成功，则在 `user` 表中找到 `user_id` 对应记录进行删除。

- `change_password(user_id, old_password, new_password)`：**修改密码**

  将 `user_id` 和密码传入 `check_password` 函数，到 `user` 表中验证二者是否匹配：

  - 验证失败则直接返回。
  - 验证成功，则在 `user` 表中找到 `user_id` 对应记录，将其密码进行更新。

- `check_password(user_id, password)`：**检查密码**

  到 `user` 表中找到 `user_id` 对应记录：

  - 如果没找到或密码不相同，验证失败。
  - 密码相同，验证成功。

- `check_token(user_id, token)`：**检查token**

  到 `user` 表中找到 `user_id` 对应记录：

  - 如果没找到 / token不相同 / token生成时间超过3600秒，验证失败。

- 不是以上情况，验证成功。

(4) `buyer.py`

该文件主要定义了**买家**的各种操作。

- `new_order(user_id, store_id, id_and_count)`：**买家下单**

  首先验证买家和店铺是否存在，若均存在：

  - 使用 `uid` 生成该订单的 `order_id`。
  - 从 `id_and_count` 中下单书籍的 `book_id` 和数量，到 `store` 表中验证是否有足够库存，若有，则更新商店库存，然后分别向 `new_order` 、`new_order_detail` 、`history_order` 表中插入一条新记录。
- `payment(user_id, password, order_id)`：**买家付款**

  先使用 `order_id` 到 `new_order` 表中验证订单是否存在，若存在：

  - 验证用户、密码是否均存在且对应。
  - 若通过验证，在 `user` 集合中验证用户余额是否足够支付，足够就更新买家账户余额。
  - 支付成功后，删除 `new_order` 、`new_order_detail` 中对应 `order_id` 的记录，同时在 `history_order` 表中设置该订单状态为已支付。
- `add_funds(user_id, password, add_value)`：**用户充值**

  验证该用户与密码是否存在且匹配，若验证成功，则在 `user` 表中更新该用户账户余额。
- `receive_book(user_id, order_id)`：**买家收货**

  首先验证该用户是否存在，若存在，就从 `history_order` 取出该 `order_id` 对应记录，如果订单存在且没有被取消 / 已经收货，就将该条记录状态更新为已收货。
- `buyer_cancel_order(user_id, order_id)`：**买家主动取消订单**

  首先验证该用户是否存在，若存在，就从 `history_order` 取出该 `order_id` 对应记录，如果订单存在且没有被取消 / 已经付款，就将该条记录状态更新为已取消。

  （这里预设买家支付后不能再主动取消订单）
- `overtime_cancel_order()`：**超时未付款自动取消订单**

  默认设置未支付订单最长保留时间为15分钟，并且默认为系统行为。

  检查 `history_order` 中所有订单：

  - 若订单状态为已支付 / 已取消，则跳过该条订单。
  - 否则对 `order_id` 进行解码，获取下单时间并与现在时间进行比较，若已超出最长保留时间，则调用 `cancel_order` 函数取消该订单。
- `cancel_order(history_order_col, order, order_id)`：**取消订单信息更新**
  - 根据 `order_id` 到 `history_order` 中将订单状态修改为已取消。
  - 根据 `order_id` 找到对应 `store_id`、`book_id` 和 下单数量 `count`，然后到 `store` 表中对应店铺将对应书籍库存进行还原。
  - 将 `new_order` 、`new_order_detail` 中对应 `order_id` 的记录删除。
- `search_history_order(user_id, order_id, page, per_page)`：**搜索历史订单**

  首先验证用户是否存在，若存在：

  - 如果用户传入 `order_id`，则从 `history_order` 中找到对应订单信息，按照分页信息返回。
  - 如果用户没有传入 `order_id`，则从 `history_order` 中找到该用户对应所有订单信息，按照分页信息返回。

- `search_book(store_id, title, author, intro, content, tags, page, per_page)`：**用户参数化搜索在售书籍**
  - 首先根据用户是否传入 `store_id`，判断是否为店铺内搜索 / 全站搜索。
  - 接下来将 `book_detail` 表和 `store` 表进行连接，根据用户传入的 `author`、`tags`、`title`、`content` 和 `intro` 信息进行查询条件的组合，然后进行 `ilike` 模糊查询。
  - 最后将 `book_detail` 表中信息分页返回。

(5) `seller.py`

该文件主要定义了**卖家**的各种操作。

- `add_book(user_id, store_id, book_id, book_json_str, stock_level)`：**向店铺添加在售书籍信息**

  首先判断用户和店铺是否存在，如果存在，判断这本书是否已经在当前店铺添加过，如果不存在，更新两部分内容：
  - 向 `store` 表中新加入一条数据，包括用户传入的所有参数信息。
  - 向 `book_detail` 集合中新加入一条数据，主要是对 `book_json_str` 中书籍的详细信息分条目进行存储（方便买家搜索书籍）。
- `add_stock_level(user_id, store_id, book_id, add_stock_level)`：**添加书籍库存**

  判断用户、店铺和待修改书籍是否都存在，如果都存在，就更新 `store` 表中对应的书籍库存数。
- `create_store(user_id, store_id)`：**卖家创建店铺**

  首先判断用户是否存在，若存在，继续判断该店铺是否已经存在，若店铺不存在，则向 `user_store` 表中添加一条数据，绑定用户与店铺的关系。
- `deliver_book(user_id, store_id, order_id)`：**卖家发货**

  首先判断用户、店铺和订单是否均存在，若存在，则判断订单是否已经被支付且未被取消，若均满足条件，则将 `history_order` 表中该订单对应状态修改为已发货。

2. View层接口

View层主要定义了网站需要用到的各种路由。

(1) `auth.py`

这个文件下所定义的请求url前缀均为 `/auth/...`，发送请求方法均为POST，都是关于用户的操作。

- `login()`：`/login`，**用户登录**

  解析用户传入的 `user_id`，`password` 和 `terminal` 信息，调用后端逻辑，生成此次登录token并存入 `user` 表中并返回到前端。
- `logout()`：`/logout`，**用户登出**

  解析用户传入的 `user_id` 和 `token`，调用后端逻辑，修改用户在 `user` 表中的token值。
- `register()`：`/register`，**用户注册**

  解析用户传入的 `user_id` 和 `password`，调用后端逻辑，在 `user` 表中新加入一条用户记录。
- `unregister()`：`/unregister`，**用户注销**

  解析用户传入的 `user_id` 和 `password`，调用后端逻辑，在 `user` 表中删除对应用户记录。
- `change_password()`：`password`，**修改密码**

解析用户传入的 `user_id`，`oldPassword` 和 `newPassword`，调用后端逻辑，在 `user` 表中修改用户密码值。

(2) `buyer.py`

这个文件下所定义的请求url前缀均为 `/buyer/...`，发送请求方法均为POST，都是关于买家的操作。

- `new_order()`：`/new_order`，**买家下单**

  解析用户传入的 `user_id`，`store_id` 和 `books`，调用后端Buyer接口，向数据库中添加新订单对应信息。

- `payment()`：`/payment`，**买家付款**

  解析用户传入的 `user_id`，`order_id` 和 `password`，调用后端Buyer接口，更新数据库中的订单状态。

- `add_funds()`：`/add_funds`，**买家充值**

  解析用户传入的 `user_id`，`password` 和 `add_value`，调用后端Buyer接口，修改 `user` 集合中用户的账户余额。

接下来为**新加入**的接口：

- `receive_book()`：`/receive_book`，**买家收货**

  解析用户传入的 `user_id` 和 `order_id`，调用后端Buyer接口，将`history_order` 中的订单状态修改为已收货。

- `overtime_cancel_order()`：`/overtime_cancel_order`，**超时取消订单**

  调用后端Buyer接口中自动检查逻辑，将`history_order` 中的超时订单状态修改为取消。

- `cancel_order()`：`/buyer_cancel_order`，**买家取消订单**

  解析用户传入的 `user_id` 和 `order_id`，调用后端Buyer接口，将`history_order` 中的未付款订单状态改为已取消。

- `search_book()`：`/search_book`，**买家搜索书籍信息**

  解析用户传入的 `store_id`、`title`、`author`、`book_intro`、`content` 和 `tags`，同时解析请求参数中的分页参数 `page` 和 `per_page`（若未传入，则默认为1和3），将这些信息一起传入后端，返回分页后的书籍信息。

- `search_history_order()`：`/search_history_order`，**买家搜索历史订单**

  解析用户传入的 `user_id` 和 `order_id`，同时解析请求参数中的分页参数 `page` 和 `per_page`（若未传入，则默认为1和3），将这些信息一起传入后端，返回分页后的历史订单信息。

(3) `seller.py`

这个文件下所定义的请求url前缀均为 `/seller/...`，发送请求方法均为POST，都是关于卖家的操作。

- `seller_create_store()`：`/create_store`，**卖家创建店铺**

  解析用户传入的 `user_id` 和 `store_id`，调用后端Seller接口，向 `user_store` 表中增加一条店铺信息。

- `seller_add_book()`：`/add_book`，**卖家添加在售书籍**

  解析用户传入的 `user_id`、`store_id`、`book_info` 和 `stock_level`，调用后端Seller接口，向 `store` 和 `book_detail` 表中添加该条书籍信息。

- `add_stock_level()`：`/add_stock_level`，**卖家增加库存**

解析用户传入的 `user_id`、`store_id`、`book_id` 和 `add_stock_level`，调用后端Seller接口，更新 `store` 中对应书籍库存数量。

- `deliver_book()`：`/deliver_book`，**卖家发货（新加入接口）**

  解析用户传入的 `user_id`、`store_id` 和 `book_id`，调用后端Seller接口，更新 `history_order` 中的订单状态为已发货。

3. Access层接口

Access层的功能是使用POST方法发送HTTP请求到服务器，以及接收服务器的状态码等内容。

(1) `auth.py`

该文件是在 `Auth` 类中定义了关于用户认证的请求。

- `login`：**用户登录请求**

  将 `user_id`，`password` 和 `terminal` 放入请求中发送，请求用户登录。

- `register`：**用户注册请求**

  将 `user_id` 和 `password` 放入请求中发送，请求新用户注册。

- `password`：**用户修改密码请求**

  将 `user_id`，`oldPassword` 和 `newPassword` 放入请求中发送，请求修改密码。

- `logout`：**用户登出请求**

  将 `user_id` 和 `token` 放入请求中发送，请求用户登出。

- `unregister`：**用户注销请求**

  将 `user_id` 和 `password` 放入请求中发送，请求用户注销。

(2) `book.py`

该文件中 `Book` 类定义了书的详细信息的基本样式，然后在 `BookDB` 类中定义了从SQLite数据库中读取数据的路径以及读取信息的操作。

- `__init__(large: bool = False)`：**设置读取数据库**

  默认 `large = False`，即从 `book.db` 中读取书籍信息；若 `large = True`，则从更大的数据库 `book_lx.db` 中读取书籍信息用于后续操作。

- `get_book_count`：**获取数据库书籍数量**

  获取 `__init__` 中选择数据库中数据条数。

- `get_book_info`：**获取书籍信息**

  从已选择的数据库中按照指定的起始与终止位置，按行读取书籍信息，并将每行数据转换为一个 `Book` 对象，添加到 `books` 列表中并返回。

(3) `new_buyer.py`

该文件使用 `register_new_buyer` 函数，使用传入的用户名和密码参数，发送一个注册用户的请求，接着创建一个Buyer对象并返回，生成一个新的买家。

(4) `buyer.py`

该文件在 `Buyer` 类中注册并登录一个买家对象，然后定义了买家的后续操作请求。

- `new_order`：**买家下单请求**

  通过将 `store_id` 和对应的下单书籍与数量数组 `book_id_and_count` 放入请求中发送，创建一个新订单。

- `payment`：**买家付款请求**

  通过将 `order_id` 和新建买家对象的用户名、密码放入请求中发送，买家完成该笔订单支付。

- `add_funds`：**买家充值请求**

  通过将 `add_value` 和新建买家对象的用户名、密码放入请求中发送，买家对自己的账户进行充值。

- `receive_book`：**买家收货请求**

  通过将 `order_id` 和新建买家对象的用户名、密码放入请求中发送，买家完成该笔订单收货。

- `buyer_cancel_order`：**买家取消订单请求**

  通过将 `order_id` 和新建买家对象的用户名、密码放入请求中发送，买家主动取消未支付的订单。

- `overtime_cancel_order`：**超时订单取消请求**

  发送该请求，系统自动取消当前所有超时未付款订单。

- `search_history_order`：**买家搜索历史订单请求**

  通过将 `order_id` 和新建买家对象的用户名、密码，以及分页参数放入请求中发送，买家搜索指定历史订单，按分页返回结果。

- `search_book`：**买家搜索书籍信息请求**

  通过将 `store_id`、`title`、`author`、`intro`、`content`、`tags` 书籍信息以及 `page`、`per_page` 的分页信息放入请求发送，买家按条件搜索图书，按分页返回结果。

(5) `new_seller.py`

该文件使用 `register_new_seller` 函数，使用传入的用户名和密码参数，发送一个注册用户的请求，接着创建一个Seller对象并返回，生成一个新的卖家。

(6) `seller.py`

该文件在 `Seller` 类中注册并登录一个卖家对象，然后定义了卖家的后续操作请求。

- `create_store`：**卖家创建店铺请求**

  通过将 `store_id` 和新建卖家对象的用户名、密码放入请求中发送，卖家完成该店铺创建。

- `add_book`：**卖家添加在售书籍请求**

  通过将 `store_id` 、`book_info` 、`stock_level` 和新建卖家的用户名放入请求中发送，卖家完成在售书籍的添加。

- `add_stock_level`：**卖家增加库存请求**

  通过将 `store_id` 、`book_info` 、`add_stock_level` 和新建卖家的用户名放入请求中发送，卖家完成该书库存的增加。

- `deliver_book`：**卖家发货请求**

通过将 `store_id` 和新建卖家对象的用户名、密码放入请求中发送，卖家完成该订单发货。

四、功能测试

1. 60%基础功能

(1) 测试用例

- `test_register`
  - 测试用户注册与注销
  - 若用户名不存在，则注销报错
  - 若用户名已存在，则注册报错
- `test_login`
  - 测试用户登录
  - 若用户名不存在，则登录报错
  - 若密码不正确，则登录报错
- `test_password`
  - 测试用户更改密码
  - 若原密码不正确，则更改密码报错
  - 若用户不存在，则更改密码报错
- `test_create_store`
  - 测试商家创建店铺
  - 若商店名已存在，则创建店铺报错
- `test_add_book`
  - 测试商家添加书籍信息
  - 若商店名不存在，则添加书籍信息报错
  - 若书籍id已存在，则添加书籍信息报错
  - 若商家名不存在，则添加书籍信息报错
- `test_add_stock_level`
  - 测试商家添加书籍库存
  - 若商店名不存在，则添加书籍库存报错
  - 若书籍id已存在，则添加书籍库存报错
  - 若商家名不存在，则添加书籍库存报错
- `test_add_funds`
  - 测试买家充值
  - 若用户名不存在，则充值报错
  - 若密码不正确，则充值报错
- `test_new_order`
  - 测试买家创建订单
  - 若用户名不存在，则创建订单报错
  - 若商店名不存在，则创建订单报错
  - 若书籍id不存在，则创建订单报错
  - 若书籍库存不足，则创建订单报错
- `test_payment`
  - 测试买家付款
  - 若密码不正确，则付款报错

    - 若余额不足，则付款报错
    - 若重复付款，则付款报错
- `test_bench`
    - 测试后端吞吐量
    - 首先把 book_lx.db（这次实验选用较大数据库进行测试）中的内容通过调用插入书本的后端插入到PostgreSQL数据库中，然后通过大量线程同时调用下订单和付款的后端接口，来测试读写性能

(2) 测试结果

经过测试，33个测试用例全部通过，测试覆盖率为94%，总运行时间为4分26秒，说明运行效率合理，取得了一个良好的测试结果。

```
============================================= test session starts =============================================
platform win32 -- Python 3.7.16, pytest-7.4.3, pluggy-1.2.0 -- D:\Anaconda\envs\db\python.exe
cachedir: .pytest_cache
rootdir: E:\database\project2\bookstore
collecting ... frontend begin test
collected 33 items

fe/test/test_add_book.py::TestAddBook::test_ok PASSED                                                [  3%]
fe/test/test_add_book.py::TestAddBook::test_error_non_exist_store_id PASSED                          [  6%]
fe/test/test_add_book.py::TestAddBook::test_error_exist_book_id PASSED                               [  9%]
fe/test/test_add_book.py::TestAddBook::test_error_non_exist_user_id PASSED                           [ 12%]
fe/test/test_add_funds.py::TestAddFunds::test_ok PASSED                                              [ 15%]
fe/test/test_add_funds.py::TestAddFunds::test_error_user_id PASSED                                   [ 18%]
fe/test/test_add_funds.py::TestAddFunds::test_error_password PASSED                                  [ 21%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_user_id PASSED                        [ 24%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_store_id PASSED                       [ 27%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_book_id PASSED                        [ 30%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_ok PASSED                                   [ 33%]
fe/test/test_bench.py::test_bench PASSED                                                             [ 36%]
fe/test/test_create_store.py::TestCreateStore::test_ok PASSED                                        [ 39%]
fe/test/test_create_store.py::TestCreateStore::test_error_exist_store_id PASSED                      [ 42%]
fe/test/test_login.py::TestLogin::test_ok PASSED                                                     [ 45%]
fe/test/test_login.py::TestLogin::test_error_user_id PASSED                                          [ 48%]
fe/test/test_login.py::TestLogin::test_error_password PASSED                                         [ 51%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_book_id PASSED                               [ 54%]
fe/test/test_new_order.py::TestNewOrder::test_low_stock_level PASSED                                 [ 57%]
fe/test/test_new_order.py::TestNewOrder::test_ok PASSED                                              [ 60%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_user_id PASSED                               [ 63%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_store_id PASSED                              [ 66%]
fe/test/test_password.py::TestPassword::test_ok PASSED                                               [ 69%]
fe/test/test_password.py::TestPassword::test_error_password PASSED                                   [ 72%]
fe/test/test_password.py::TestPassword::test_error_user_id PASSED                                    [ 75%]
fe/test/test_payment.py::TestPayment::test_ok PASSED                                                 [ 78%]
fe/test/test_payment.py::TestPayment::test_authorization_error PASSED                                [ 81%]
fe/test/test_payment.py::TestPayment::test_not_suff_funds PASSED                                     [ 84%]
fe/test/test_payment.py::TestPayment::test_repeat_pay PASSED                                         [ 87%]
fe/test/test_register.py::TestRegister::test_register_ok PASSED                                      [ 90%]
fe/test/test_register.py::TestRegister::test_unregister_ok PASSED                                    [ 93%]
fe/test/test_register.py::TestRegister::test_unregister_error_authorization PASSED                   [ 96%]
fe/test/test_register.py::TestRegister::test_register_error_exist_user_id PASSED                     [100%]

========================================== 33 passed in 266.34s (0:04:26) ==========================================
```

Name	Stmts	Miss	Branch	BrPart	Cover
be__init__.py	0	0	0	0	100%
be\app.py	3	3	2	0	0%
be\model__init__.py	0	0	0	0	100%
be\model\buyer.py	103	17	42	10	81%
be\model\db_conn.py	23	0	6	0	100%
be\model\error.py	23	1	0	0	96%
be\model\seller.py	43	7	16	1	86%
be\model\store.py	26	2	0	0	92%
be\model\user.py	109	15	30	6	85%
be\serve.py	34	1	2	1	94%
be\view__init__.py	0	0	0	0	100%
be\view\auth.py	37	0	0	0	100%
be\view\buyer.py	31	0	2	0	100%
be\view\seller.py	28	0	0	0	100%
fe__init__.py	0	0	0	0	100%
fe\access__init__.py	0	0	0	0	100%
fe\access\auth.py	31	0	0	0	100%
fe\access\book.py	69	1	12	2	96%
fe\access\buyer.py	36	0	2	0	100%
fe\access\new_buyer.py	8	0	0	0	100%
fe\access\new_seller.py	8	0	0	0	100%
fe\access\seller.py	31	0	0	0	100%
fe\bench__init__.py	0	0	0	0	100%
fe\bench\run.py	13	0	6	0	100%
fe\bench\session.py	47	0	12	2	97%
fe\bench\workload.py	125	2	22	2	97%
fe\conf.py	11	0	0	0	100%
fe\conftest.py	17	0	0	0	100%
fe\test\gen_book_data.py	49	0	16	1	98%
fe\test\test_add_book.py	36	0	10	0	100%
fe\test\test_add_funds.py	22	0	0	0	100%
fe\test\test_add_stock_level.py	39	0	10	0	100%
fe\test\test_bench.py	6	2	0	0	67%
fe\test\test_create_store.py	19	0	0	0	100%
fe\test\test_login.py	27	0	0	0	100%
fe\test\test_new_order.py	39	0	0	0	100%
fe\test\test_password.py	32	0	0	0	100%
fe\test\test_payment.py	59	1	4	1	97%
fe\test\test_register.py	30	0	0	0	100%
TOTAL	1214	52	194	26	94%

2. 40%附加功能

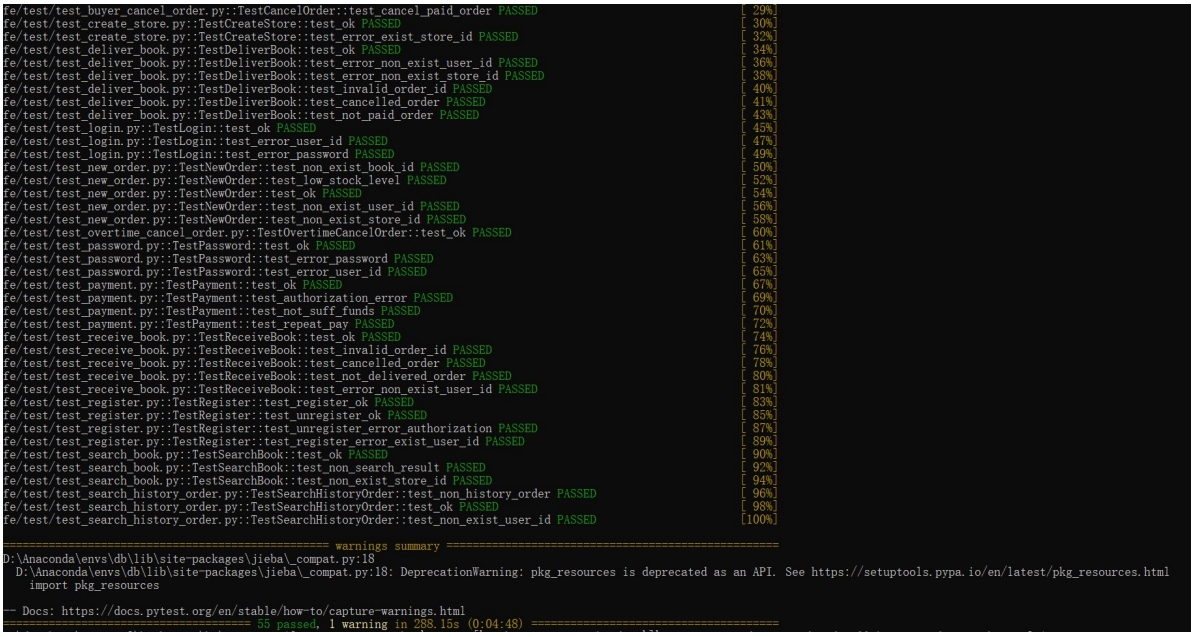
(1) 测试用例

- `test_buyer_cancel_order`
  - 测试买家取消订单
  - 若订单id不存在，则取消订单报错
  - 若重复取消订单，则取消订单报错
  - 若订单已取消（比如因为超时自动取消），则取消订单报错
- `test_overtime_cancel_order`
  - 测试超时自动取消订单
- `test_deliver_book`
  - 测试商家发货
  - 若用户名不存在，则发货报错
  - 若商店名不存在，则发货报错
  - 若订单id不存在，则发货报错
  - 若订单已取消，则发货报错
  - 若订单未支付（还未取消），则发货报错
- `test_receive_book`
  - 测试买家收货
  - 若用户名不存在，则收货报错
  - 若订单id不存在，则收货报错

    - 若订单已取消，则收货报错
    - 若订单未发货，则收货报错
- `test_search_book`
    - 测试买家搜索书籍
    - 若没有搜索结果，则搜索书籍报错
    - 若想在当前店铺搜索，而商店名不存在，则搜索书籍报错
- `test_search_history_order`
    - 测试买家搜索历史订单
    - 若没有搜索结果，则搜索历史订单报错
    - 若用户名不存在，则搜索历史订单报错

(2) 测试结果

经过测试，55个测试用例全部通过（1个warning是`jieba`包自带的一些问题），测试覆盖率为90%（在`buyer.py`中新添加代码有较多异常的捕获，所以覆盖率有所下降），总运行时间为4分48秒，运行效率合理，说明取得了一个良好的测试结果。



Name	Stmts	Miss	Branch	BrPart	Cover
be__init__.py	0	0	0	0	100%
be\app.py	3	3	2	0	0%
be\model__init__.py	0	0	0	0	100%
be\model\buyer.py	329	99	162	34	66%
be\model\db_conn.py	23	0	6	0	100%
be\model\error.py	35	2	0	0	94%
be\model\seller.py	80	9	30	1	91%
be\model\store.py	37	2	0	0	95%
be\model\user.py	112	17	32	6	84%
be\serve.py	34	1	2	1	94%
be\view__init__.py	0	0	0	0	100%
be\view\auth.py	37	0	0	0	100%
be\view\buyer.py	67	0	2	0	100%
be\view\seller.py	35	0	0	0	100%
fe__init__.py	0	0	0	0	100%
fe\access__init__.py	0	0	0	0	100%
fe\access\auth.py	31	0	0	0	100%
fe\access\book.py	69	1	12	2	96%
fe\access\buyer.py	67	0	2	0	100%
fe\access\new_buyer.py	8	0	0	0	100%
fe\access\new_seller.py	8	0	0	0	100%
fe\access\seller.py	37	0	0	0	100%
fe\bench__init__.py	0	0	0	0	100%
fe\bench\run.py	13	0	6	0	100%
fe\bench\session.py	47	0	12	2	97%
fe\bench\workload.py	125	2	22	2	97%
fe\conf.py	11	0	0	0	100%
fe\conftest.py	17	0	0	0	100%
fe\test\gen_book_data.py	49	1	16	2	95%
fe\test\test_add_book.py	36	0	10	0	100%
fe\test\test_add_funds.py	22	0	0	0	100%
fe\test\test_add_stock_level.py	39	0	10	0	100%
fe\test\test_bench.py	6	2	0	0	67%
fe\test\test_buyer_cancel_order.py	54	1	4	1	97%
fe\test\test_create_store.py	19	0	0	0	100%
fe\test\test_deliver_book.py	78	1	4	1	98%
fe\test\test_login.py	27	0	0	0	100%
fe\test\test_new_order.py	39	0	0	0	100%
fe\test\test_overtime_cancel_order.py	39	1	4	1	95%
fe\test\test_password.py	32	0	0	0	100%
fe\test\test_payment.py	59	1	4	1	97%
fe\test\test_receive_book.py	82	1	4	1	98%
fe\test\test_register.py	30	0	0	0	100%
fe\test\test_search_book.py	29	0	0	0	100%
fe\test\test_search_history_order.py	40	0	4	0	100%
TOTAL	1905	144	350	55	90%

3. 测试驱动开发

在进行书籍和历史订单搜索的测试中，我一直遇到在注册买家步骤时的注册错误，后来发现是因为数据库判断插入用户名重复，所以抛出错误。因此我在注册函数中新增加条件分支，首先查找数据库中是否已经存在该用户名，如果已经存在就不继续进行后续的数据库插入操作。这样，通过测试可以弥补编写代码时的逻辑漏洞。

五、版本管理

本次实验使用了git作为版本管理工具，完成一个版本的代码后就将其上传到github上，同时在每个版本进行后续修改后也上传到github上，这样方便回溯到项目的任何历史版本。

本次实验主要分为两个版本：前60%基础功能版本和完整功能版本，分别存储在 `bookstore_60` 和 `bookstore` 两个文件夹下。

同时，最初的两个版本均是直接使用 `psycopg2` 直接对数据库进行操作，后来改为使用 `sqlalchemy` 进行ORM相关操作，这也可以在github中的commit记录里找到。

仓库链接：https://github.com/YifeiLong/CDMS/tree/main/Project2

六、总结

这次实验我主要有以下收获：

- 加强了对ORM的理解与运用，对于sqlalchemy常见的出错情况有了一定的了解，针对前后端项目的debug能力有一定提升。
- 对于关系型数据库表设计有了更好的理解。