2023_ECNU_PJ2_报告

1. 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。

我将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，我将功能测试的代码放在/fe/test文件夹下。

Python

```
.
├─be
│   │  app.py
│   │  serve.py
│   │
│   ├─model
│   │   │  buyer.py
│   │   │  db_conn.py
│   │   │  error.py
│   │   │  seller.py
│   │   │  store.py
│   │   │  user.py
│   │
│   ├─view
│   │   │  auth.py
│   │   │  buyer.py
│   │   │  seller.py
│
├─fe
│   ├─access
│   │       auth.py
│   │       book.py
│   │       buyer.py
│   │       new_buyer.py
│   │       new_seller.py
│   │       seller.py
│   │
│   ├─data
│   │       data_transfer.py
│   │       gen_book_db.py
│   │       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
```

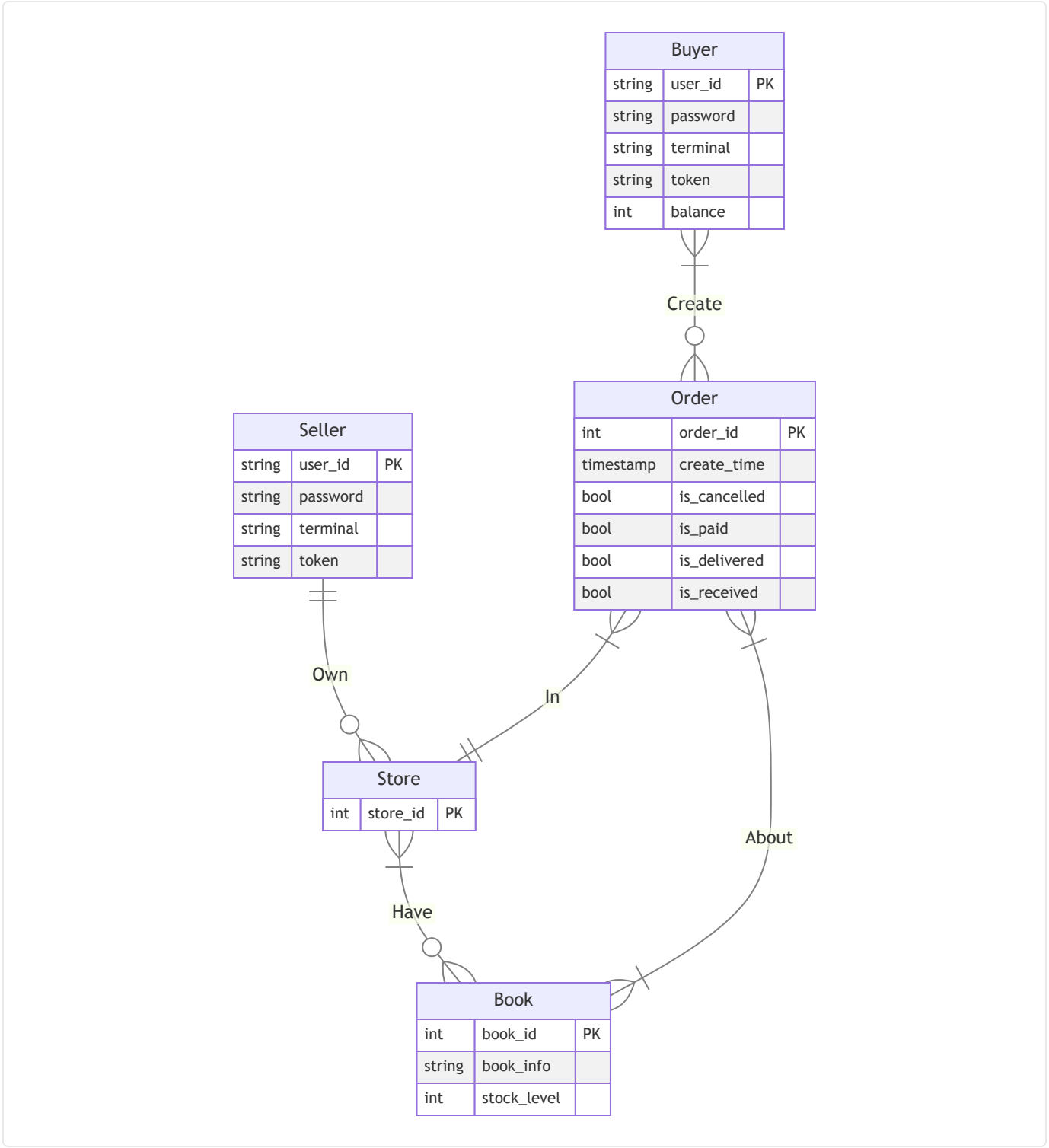
```
│          test_payment.py
│          test_receive_book.py
│          test_register.py
│          test_search_book.py
│          test_search_history_order.py
│
└─script
       test.sh
```

2. 关系数据库设计

2.1. ER图

- 对于用户，他能够注册、注销、登录、登出、更改密码
- 对于买家，他还能够充值、搜索书籍、创建订单、取消订单、付款、收货、搜索历史订单
- 对于商家，他还能够创建店铺、添加书籍信息、添加书籍库存、发货

基于上述逻辑，先绘制ER图，再利用ER图指导进行关系模式设计。



2.2. 从ER图到关系模式设计

2.2.1. 关系表结构

根据ER图，共设计了7个关系表，具体表结构如下：

- user

```python
class User(Base):
    __tablename__ = 'user'
    // 用户名
    user_id = Column(String, primary_key=True, unique=True, nullable=False)
    // 用户密码
    password = Column(String, nullable=False)
    // 用户余额
    balance = Column(Integer, nullable=False)
    // 登录token
    token = Column(String)
    // 登录terminal
    terminal = Column(String)
```

- user_store

```python
class UserStore(Base):
    __tablename__ = 'user_store'
    // 商家名
    user_id = Column(String, ForeignKey('user.user_id'), primary_key=True, nullable=False)
    // 商店名
    store_id = Column(String, primary_key=True, nullable=False)
```

- store

```python
class Store(Base):
    __tablename__ = 'store'
    // 商店名
    store_id = Column(String, primary_key=True, nullable=False)
    // 书籍id
    book_id = Column(String, primary_key=True, nullable=False)
    // 书籍信息（标题、作者、简介、内容、标签）
    book_info = Column(String)
    // 书籍库存
    stock_level = Column(Integer)
```

- new_order

```python
class NewOrder(Base):
    __tablename__ = 'new_order'
    // 订单id
    order_id = Column(String, primary_key=True, unique=True, nullable=False)
    // 买家名
    user_id = Column(String, ForeignKey('user.user_id'))
    // 商店名
    store_id = Column(String)
```

- new_order_detail

```python
class NewOrderDetail(Base):
    __tablename__ = 'new_order_detail'
    // 订单id
    order_id = Column(String, primary_key=True, nullable=False)
    // 书籍id
    book_id = Column(String, primary_key=True, nullable=False)
    // 书籍数量
    count = Column(Integer)
    // 书籍单价
    price = Column(Integer)
```

- history_order

```python
class HistoryOrder(Base):
    __tablename__ = 'history_order'
    // 订单id
    order_id = Column(String, primary_key=True, unique=True, nullable=False)
    // 买家名
    user_id = Column(String, primary_key=True, nullable=False)
    // 商店名
    store_id = Column(String)
    // 订单创建时刻
    create_time = Column(TIMESTAMP)
    // 购买书籍的信息，包括书籍id、数量、单价
    book_info = Column(ARRAY(JSON))
    // 订单是否取消
    is_cancelled = Column(Boolean)
    // 订单是否支付
    is_paid = Column(Boolean)
    // 订单是否发货
    is_delivered = Column(Boolean)
    // 订单是否收货
    is_received = Column(Boolean)
```

- book_detail

```python
class BookDetail(Base):
    __tablename__ = 'book_detail'
    // 书籍id
    book_id = Column(String, primary_key=True, unique=True, nullable=False)
    // 书籍标题
    title = Column(String)
    // 书籍作者
    author = Column(String)
    // 书籍简介
    book_intro = Column(String)
    // 书籍内容
    content = Column(String)
    // 书籍标签
    tags = Column(String)
```

2.2.2. 约束

这一部分将在3.1节中详细介绍。

2.3. 数据存储

书籍的图片信息使用本地MongoDB数据库存储，其他数据均使用本地PostgreSQL数据库存储。

3. 功能介绍：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Controller层调用这些接口来相应前端的请求。

3.1. `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。

该类首先通过 `__init__` 函数建立与本地PostgreSQL数据库的连接，并默认使用 `postgres` 数据库和 `public` 模式；接下来在 `init_tables` 中建立后续代码中会使用到的**关系表**及其**约束**：

- `user`：存放已经注册的用户信息
  - 将 `user_id` 列作为主键，并设置唯一约束和非空约束，防止重复的用户数据插入，同时加快后续通过 `user_id` 查找用户的速度。
- `user_store`：存放用户（商家）id及其拥有的店铺id信息
  - 将 `user_id` 列和 `store_id` 列作为复合主键，并均设置非空约束，每个商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，复合主键可以加快查询速度），允许用户与店铺存在一对多或多对一的关系。在 `user_id` 列上设置外键约束，商家必须是已经注册的用户，
- `store`：存放店铺中在售的书籍id及其库存量
  - 将 `store_id` 列和 `book_id` 列作为复合主键，并均设置非空约束，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，复合主键相比于单一主键更有速度上的优势。
- `new_order`：存放用户（买家）新创建订单信息
  - 将 `order_id` 列作为主键，并设置唯一约束和非空约束，保证同一笔订单只插入一次，同时加快根据订单号对用户及店家信息的查找。在 `user_id` 列上设置外键约束，买家必须是已经注册的用户。
- `new_order_detail`：存放某笔订单中某本书的详细订单信息
  - 将 `order_id` 列和 `book_id` 列作为复合主键，并均设置非空约束，加快根据订单号和书籍id查找的速度。另外，由于单笔订单中可能存在下单多种书籍，因此不在 `order_id` 列上设置唯一约束，允许在同一个订单情况下重复插入不同书籍信息。
- `history_order`：存放所有订单的信息与状态

    - 将 `user_id` 列和 `order_id` 列作为复合主键，并均设置非空约束，在 `order_id` 列上设置唯一约束，这样既可以控制用户与订单一一对应的信息不重复插入，也可以加快后续历史订单组合条件查询的速度。
- `book_detail`：存放所有店铺在售书籍的详细信息
    - 将 `book_id` 列作为主键，并设置唯一约束和非空约束，防止多家店铺同时售卖同一种书籍造成的信息重复插入，同时加快用户搜索书籍速度。

接下来通过实例化一个 `Store` 类，返回其数据库接口以供其他函数调用。

3.2. `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`，在 `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`，在 `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`，在 `user_store` 表中寻找对应记录，如果成功，则该家店铺存在。

3.3. `user.py`

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

- `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），系统为其自动生成terminal和token，并默认账户余额为0。接下来尝试将该条数据存入 `user` 表，如果用户名已经存在，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，在 `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` 函数在 `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，在 `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，在 `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**
  在 `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**
  在 `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。

3.4. `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，在 `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` 在 `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` 表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` 表中对应 `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` 表中取出该 `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` 表中取出该 `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，并调用 `cancel_order` 函数取消该订单。（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` 表中所有订单：
  - 若订单状态为已支付 / 已取消，则跳过该条订单。
  - 否则获取下单时刻并与当前时刻计算得到订单存在时间，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)`：**具体的取消订单操作**
  - 根据 `order_id` 在 `history_order` 表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`，然后在 `store` 表中对应店铺将对应书籍库存进行还原。
  - 将 `new_order` 、`new_order_detail` 表中对应 `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`，则在 `history_order` 表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`，则在 `history_order` 表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**
  参数化搜索主要分为**三个阶段**：
  - 首先根据用户传入的 `store_id, title, author, intro, content, tags` 信息（均允许为空）构建查询条件，如果 `store_id` 为空，则为全站搜索，否则为指定店铺内搜索。
  - 接下来将 `book_detail` 表和 `store` 表连接，应用查询条件进行查询，对于 `title, author, intro, content, tags` 信息均使用 `ilike` 模糊搜索方法。
  - 最后仅选取查询结果中包含的 `book_detail` 表中信息，按照分页信息返回。

3.5. `seller.py`

该文件主要定义了**商家**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` 表中新加入一条数据，主要是对 `book_json_str` 中书籍的详细信息进行存储（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)`：**商家创建店铺**
  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)`：**商家发货**
  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。

4. 功能介绍：View层接口

View层主要定义了网站需要用到的各种路由。

4.1 `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` 信息，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` 信息，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` 信息，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` 信息，调用后端逻辑，在 `user` 表中修改用户密码值。

4.2 `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` 信息，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` 信息，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` 信息，调用后端Buyer接口，修改 `user` 表中用户的账户余额。
- `receive_book()`：`/receive_book`，**买家收货**
  解析用户传入的 `user_id` 和 `order_id` 信息，调用后端Buyer接口，将 `history_order` 表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` 表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` 信息，调用后端Buyer接口，将 `history_order` 表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`

信息，同时解析请求参数中的分页参数 `page` 和 `per_page` 信息（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` 信息，同时解析请求参数中的分页参数 `page` 和 `per_page` 信息（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。

4.3 `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` 信息，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` 信息，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` 信息，调用后端Seller接口，更新 `store` 表中对应书籍库存数量。
- `deliver_book()`：`/deliver_book`，**卖家发货**
  解析用户传入的 `user_id`、`store_id` 和 `book_id` 信息，调用后端Seller接口，更新 `history_order` 表中的订单状态为已发货。

5. 功能介绍：Controller层接口

Controller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。

5.1 `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。

5.2 `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。

5.3 `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。

5.4 `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。

5.5 `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。

5.6 `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。

6. 功能测试

6.1. 60%基础功能

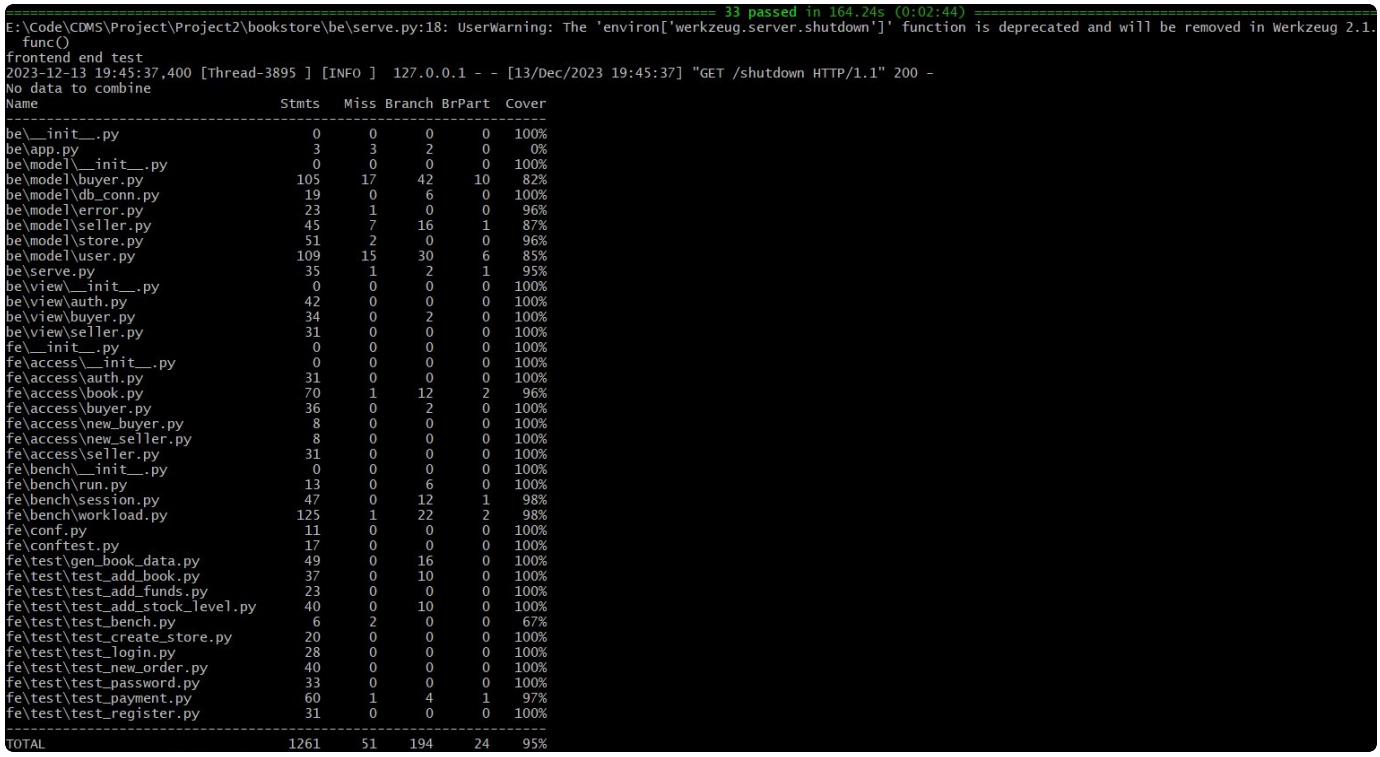
6.1.1. 测试用例

- `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
  - 若用户名不存在，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.db 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

6.1.2. 测试结果

经过测试，33个测试用例全部通过，测试覆盖率为95%，说明取得了一个良好的测试结果。



6.2. 40%附加功能

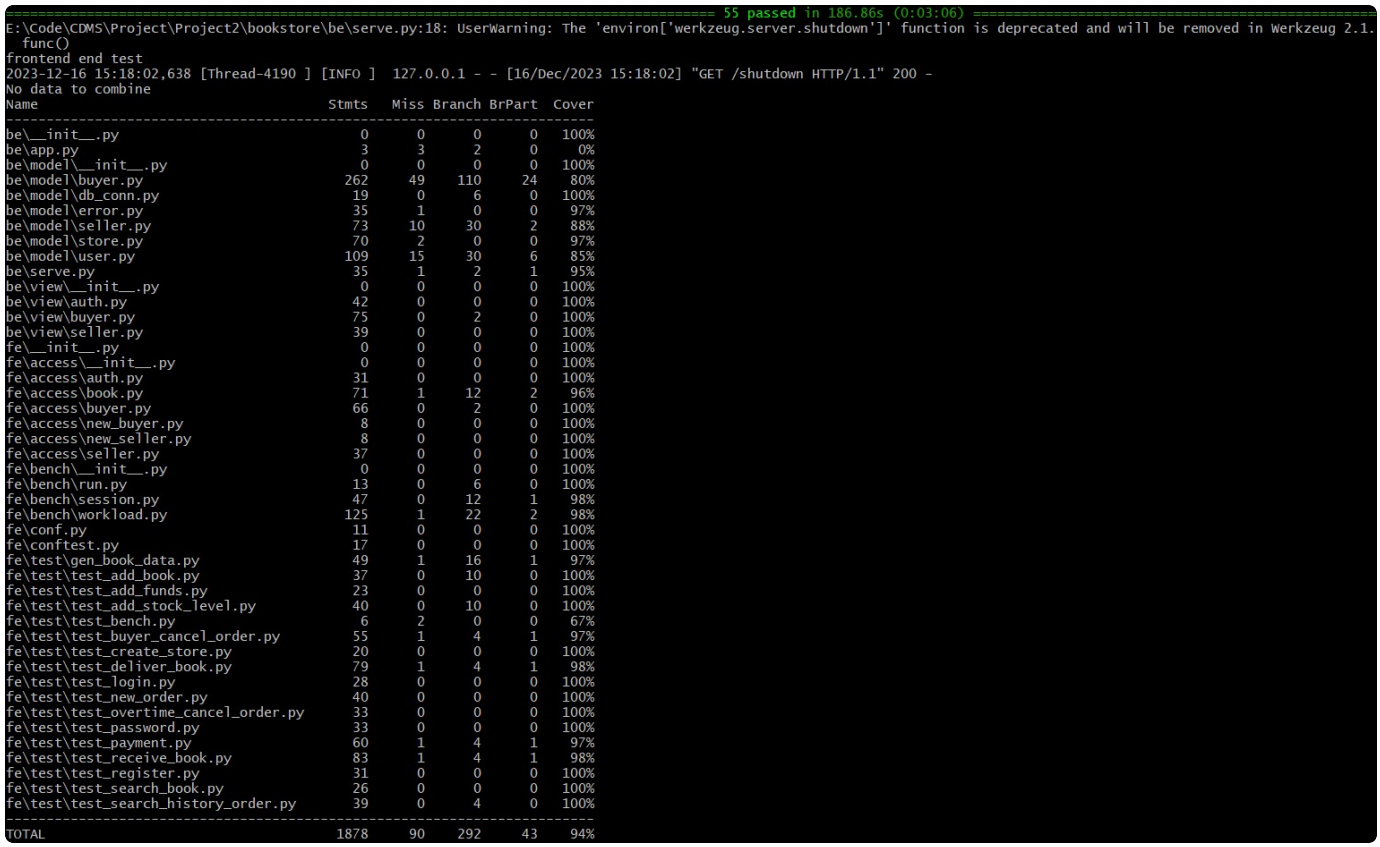
6.2.1. 测试用例

- `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（可能因为超时自动取消），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

6.2.2. 测试结果

经过测试，55个测试用例全部通过，测试覆盖率为94%（在实现附加功能时，添加了较多 `try ... except ...` 语句用于错误捕获，所以部分文件的覆盖率可能有所下降），说明取得了一个良好的测试结果。



```
========================================================================== 55 passed in 186.86s (0:03:06) ==========================================================================
E:\Code\CDMS\Project\Project2\bookstore\be\serve.py:18: UserWarning: The 'environ['werkzeug.server.shutdown']' function is deprecated and will be removed in Werkzeug 2.1.
  func()
frontend end test
2023-12-16 15:18:02,638 [Thread-4190 ] [INFO ]  127.0.0.1 - - [16/Dec/2023 15:18:02] "GET /shutdown HTTP/1.1" 200 -
No data to combine
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                               262     49    110     24    80%
be\model\db_conn.py                              19      0      6      0   100%
be\model\error.py                                35      1      0      0    97%
be\model\seller.py                               73     10     30      2    88%
be\model\store.py                                70      2      0      0    97%
be\model\user.py                                109     15     30      6    85%
be\serve.py                                      35      1      2      1    95%
be\view\__init__.py                               0      0      0      0   100%
be\view\auth.py                                  42      0      0      0   100%
be\view\buyer.py                                 75      0      2      0   100%
be\view\seller.py                                39      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                                71      1     12      2    96%
fe\access\buyer.py                               66      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      1    98%
fe\bench\workload.py                            125      1     22      2    98%
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      1    97%
fe\test\test_add_book.py                         37      0     10      0   100%
fe\test\test_add_funds.py                        23      0      0      0   100%
fe\test\test_add_stock_level.py                  40      0     10      0   100%
fe\test\test_bench.py                             6      2      0      0    67%
fe\test\test_buyer_cancel_order.py               55      1      4      1    97%
fe\test\test_create_store.py                     20      0      0      0   100%
fe\test\test_deliver_book.py                     79      1      4      1    98%
fe\test\test_login.py                            28      0      0      0   100%
fe\test\test_new_order.py                        40      0      0      0   100%
fe\test\test_overtime_cancel_order.py            33      0      0      0   100%
fe\test\test_password.py                         33      0      0      0   100%
fe\test\test_payment.py                          60      1      4      1    97%
fe\test\test_receive_book.py                     83      1      4      1    98%
fe\test\test_register.py                         31      0      0      0   100%
fe\test\test_search_book.py                      26      0      0      0   100%
fe\test\test_search_history_order.py             39      0      4      0   100%
-----------------------------------------------------------------------------
TOTAL                                          1878     90    292     43    94%
```

7. 性能测试

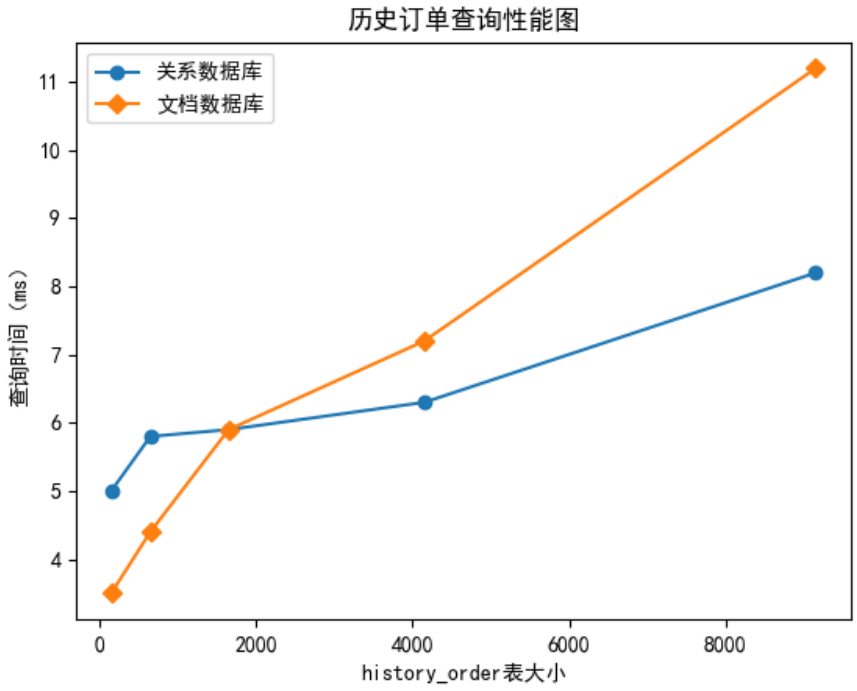
7.1. 历史订单查询性能

7.1.1. 测试用例

`test_search_history_order_performance`

- 先生成一个原始买家，并创建一个属于他的订单
- 再设置重复轮数，每次循环中，生成新的买家、商家、商店，并创建一个属于新买家的订单。循环结束后查询原始买家的历史订单，计算查询时间，并获得此时 `history_order` 表的大小
- 重复轮数取值为[100, 500, 1000, 2500, 5000]

7.1.2. 测试结果



观察上图可得，当数据库存储的数据量较大时，改用关系数据库有助于提升程序运行和事务处理的性能。

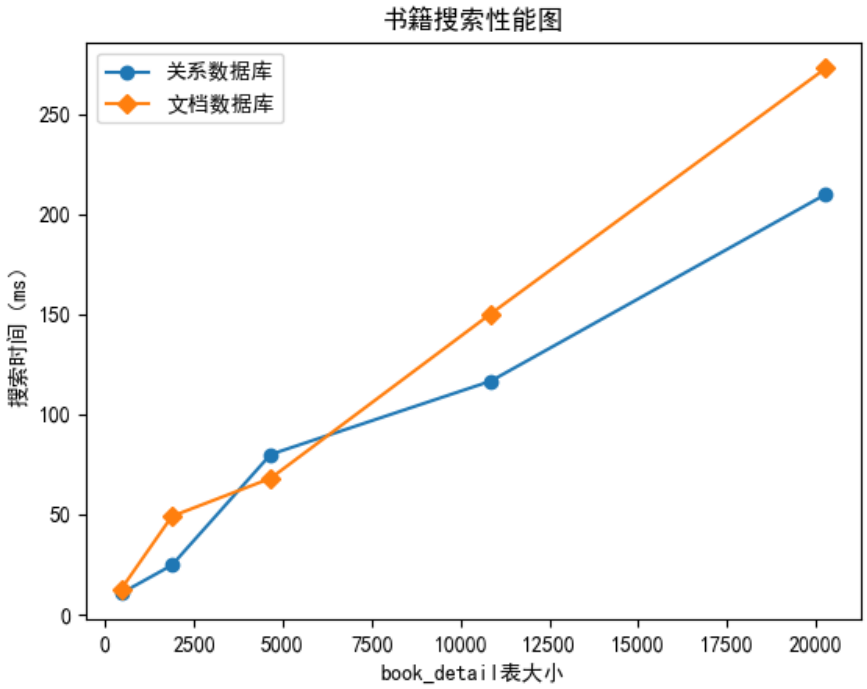
7.2. 书籍搜索性能

7.2.1. 测试用例

`test_search_book_performance`

- 先生成一个买家
- 再设置重复轮数，每次循环中，生成新的商家、商店，并且商家会向商店添加书籍，同时会向全站书籍名录添加书籍。循环结束后买家全站搜索书籍，计算搜索时间，并获得此时 `book_detail` 表的大小
- 重复轮数取值为[100, 500, 1000, 2500, 5000]

7.2.2. 测试结果



观察上图同样可得，当数据库存储的数据量较大时，改用关系数据库有助于提升程序运行和事务处理的性能。

8. 从文档数据库到关系数据库的改动

8.1. 改动内容

- 实现模糊搜索功能：

- 以实现参数化搜索在售书籍功能为例，如果使用文档数据库，在向 `book_detail` 集合中添加书籍信息时，需要对 `title, content, book_intro` 信息进行分词处理，并将分词结果整合另存为 `description` 属性，在搜索时利用 `description` 属性和搜索条件进行匹配，否则无法实现模糊搜索功能
- 改用关系数据库后，无需进行分词处理，也不需要多添加 `description` 字段，只需使用 `ilike` 模糊搜索方法即可实现模糊搜索功能

- 实现多表联合查询：
  - 以实现参数化搜索在售书籍功能为例，如果使用文档数据库，只能采用单表逐步筛选查询的方法，在进行当前店铺搜索时，最多需要分为三个阶段：先根据 `store_id` 从 `store` 集合中将该店铺中所有对应 `book_id` 取出，作为第一步的筛选条件；然后将 `author` 和 `tags` 信息作为精确查询条件，与上一步得到的 `book_id` 信息组合在一起传入 `book_detail` 集合中再次进行查询，返回第二步符合条件的 `book_id` 信息；最后将 `title, content, book_intro` 信息的分词结果与 `book_detail` 集合中 `description` 属性进行匹配，得到最终查询结果
  - 改用关系数据库后，可以采用多表联合查询的方法，仅需一步构建好查询条件，然后使用 `join` 方法对 `store` 表和 `book_detail` 表进行联合查询，即可得到查询结果
- 采用ORM技术实现事务处理：
  - ORM（Object-Relational Mapping）是一种用于在关系数据库和面向对象编程语言之间进行映射的技术，通过抽象事务的管理细节，提供了更高层次的接口，便于编写业务逻辑代码
  - 具体而言，与文档数据库相比，关系数据库支持原子性事务，即在事务中的所有操作要么全部成功，要么全部失败回滚，ORM工具通常提供方便的方法来管理和处理这些事务，确保数据的一致性；此外，关系数据库允许定义数据完整性约束，如主键、外键等，这有助于保证数据的完整性，ORM工具可以利用这些约束，提供更好的数据验证和保护机制

8.2. 改动理由

- 实现模糊搜索功能：
  - `title, content, book_intro` 字段均为文本信息，所以它们的分词结果的数据量很大，改动后减少了 `description` 这一个近似“冗余”的属性，可以有效减少数据库存储的数据量，有利于提升查询性能和网络传输效率
  - 分词操作的准确度难以完全保证，可能在复合词和新词汇等方面出现错误，改动后使用 `ilike` 模糊搜索，可以匹配在任何位置包含搜索条件的字符串，更加符合在日常场景下参数化搜索功能的要求，有利于提升搜索在售书籍的准确度
- 实现多表联合查询：
  - 单表逐步筛选查询中每一步都需要引入查询、过滤和保存筛选结果操作，所以需要编写多个查

询语句，可能导致代码复杂性增加，改动后可以将多个条件和关系集中到一个查询语句中，便于编写代码，也使查询更为简洁和可读

    - 单表逐步筛选查询中每一步都要将大量数据传输到服务器，增加了网络开销，可能会影响到查询性能，改动后多表联合查询可以通过在数据库服务器上执行连接操作，利用数据库引擎的优化和索引来提高查询性能
- 采用ORM技术实现事务处理：
    - 改动后为事务的ACID属性（原子性、一致性、隔离性、持久性）提供了更强的支持与保证，有助于提高开发效率、简化代码逻辑、减少错误和异常处理，并提高系统的可靠性和可维护性

9. 版本管理 & 项目总结

9.1. 版本管理

- 我使用git作为版本管理工具，方便回溯到项目的任何历史版本，使得项目进展更加容易和高效
- 仓库链接：https://github.com/KaihengQian/CDMS/tree/main/Project2

9.2. 项目总结

- 对关系数据库的关系模式设计思路有了更好的理解，广泛了解了PostgreSQL事务处理的相关内容
- 熟悉了PostgreSQL数据库的各种操作方式和建立约束的方法及作用，掌握了ORM技术
- 掌握了功能测试以及性能测试的方法，提升了前后端项目开发中的Debug能力