实验的功能和要求

- 实现一个提供网上购书功能的网站后端。
- 网站支持书商在上面开商店，购买者可以通过网站购买。
- 买家和卖家都可以注册自己的账号。
- 一个卖家可以开一个或多个网上商店。
- 买家可以为自己的账户充值，在任意商店购买图书。
- 支持 下单->付款->发货->收货 流程。

1.实现对应接口的功能，见项目的 doc 文件夹下面的 .md 文件描述 （60%）

其中包括：

1)用户权限接口，如注册、登录、登出、注销

2)买家用户接口，如充值、下单、付款

3)卖家用户接口，如创建店铺、填加书籍信息及描述、增加库存

通过对应的功能测试，所有 test case 都 pass

2.为项目添加其它功能： （40%）

1)实现后续的流程： 发货 -> 收货

2)搜索图书： 用户可以通过关键字搜索，参数化的搜索方式；如搜索范围包括，题目，标签，目录，内容；全站搜索或是当前店铺搜索。如果显示结果较大，需要分页。(使用全文索引优化查找)

3)订单状态，订单查询和取消订单：用户可以查自己的历史订单，用户也可以取消订单。取消订单可由买家主动地取消，或者买家下单后，经过一段时间超时仍未付款，订单也会自动取消。

具体的一些要求：

1.`bookstore` 文件夹是该项目的 demo，采用 Flask 后端框架，实现了前60%功能以及对应的测试用例代码。要求大家创建本地 `MongoDB 数据库`，将 `bookstore/fe/data/book.db` 中的内容以合适的形式存入本地数据库(在 `bookstore/transfer_data.py` 文件内实现)，后续所有数据读写都在本地的 MongoDB 数据库中进行。书本的内容可自行构造一批，也可参从网 盘下载，下载地址为： （https://pan.baidu.com/s/1bjCOW8Z5N_ClcqU54Pdt8g）提取码：hj6q

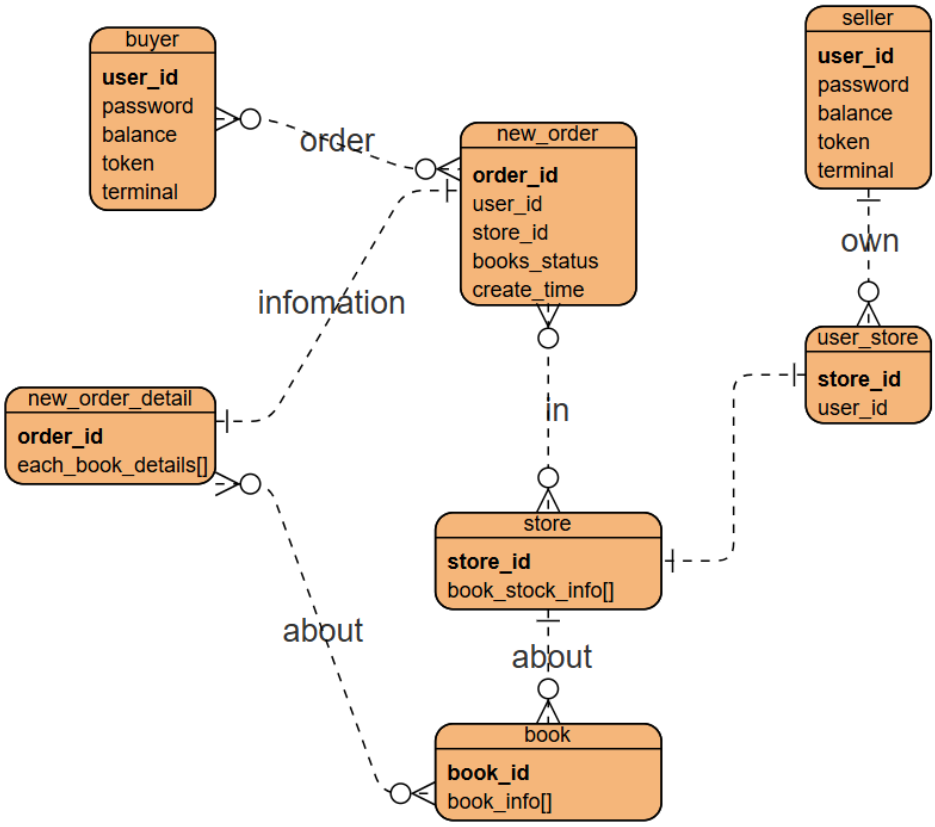
2.在完成前60%功能的基础上，继续实现后40%功能，要有接口、后端逻辑实现、数据库操作、代码测试。对所有 接口都要写 test case，通过测试并计算测试覆盖率（尽量提高测试覆盖率）。

3.尽量使用索引，对程序与数据库执行的性能有考量

4.尽量使用 git 等版本管理工具

5.不需要实现界面，只需通过代码测试体现功能与正确性

文档数据库的设计



其中buyer和user都是由同一个user集合得到的，字段值没有区别。下述是对各个集合的详细介绍。

文档集合描述：

用户集合 (user)

- `user_id`: 用户名
- `password`: 密码
- `balance`: 用户账户余额
- `token`: 加密标识符，由user_id和terminal与时间戳的加密连接字符串组成
- `terminal`: 登录终端信息
- 每个用户可能拥有多个商店，表示为子集合 `stores`

主键: `user_id`

商店店家个人信息集合(user_store)

- store_id:商店标识符
- user_id：用户名

主键: `store_id`

商店集合 (store)

- `store_id`: 商店标识符
- `book_stock_info`: 书本库存信息（数组结构）

- 每个书本库存信息包含 `book_id` 和 `stock_level`

主键： `store_id`

书本集合 (book)

- `book_id`: 书本标识符
- `book_info`: 书本详细信息，通常为 JSON 格式

主键： `book_id`

订单集合 (new_order)

- `order_id`: 订单标识符
- `user_id`: 关联用户 ID
- `store_id`: 关联商店 ID
- `books_status`: 订单状态
- `create_time`: 创建时间

主键： `order_id`

状态代码（books_status）：

- -1: 取消
- 0: 已发货，未收货
- 1: 已付款，待发货
- 2: 初始值，未付款
- 3: 已收货

订单详情集合 (new_order_detail)

- `order_id`: 订单标识符（与 Orders 集合的 `order_id` 一致）
- `each_book_details`:一个订单内购买的书本的信息(数组结构)
    - `book_id`: 书本标识符
    - `count`: 订购数量
    - `price`: 单价

主键： `order_id`

数据库结构设计理由：

1.尽可能**降低读放大。**

在原来的关系型数据库的store表内，其数据如下 `INSERT into store(store_id, book_id, book_info, stock_level)`，这里的`book_info`表示了每本书的详细信息。而在查询一个`store`内是否包含某一`book_id`时，会不可避免的读取`book_info`这一信息,因此我们将`book`的信息单独形成一个集合，后续通过`book_id`查找书本的具体信息。这一样在**降低读放大**的同时，也可以有效的降低b+树叶子结点的数量，降低高度，使得**查询更快。**也更好的实现了解耦合，可以方便的从外部查找某一书本的信息。

2.根据数据库特性改变结构：

由于之前的数据库是关系型数据库，这就导致无法在一个字段内存储数组信息，例如`INSERT INTO new_order_detail(order_id, book_id, count, price)`,需要重复的`order_id`，来记录本次订单内的某一书本的详细信息,而本实验是文档数据库，可以存储数组。此外还有一个好处是，原先的设计模式因为重复的`order_id`，其索引是`order_id, book_id`的组合。而现在索引只需要在`order_id`上设置索引即可。并且，按照实际情况单次`order`内的`book`种类不会过多。因此若要其对某一`order_id, book_id`的查找所增加的线性开销其影响是有限的。

3.增加可扩展性

```
(user_store)
- store_id:商店标识符
- user_id: 用户名
**商店集合 (store)**
- `store_id`: 商店标识符
- `book_stock_info`: 书本库存信息（数组结构）
  - 每个书本库存信息包含 `book_id` 和 `stock_level`
```

这里我们保留了`user_store`表。之所以不将`user_id`放入`store`集合，是因这会方便后续对于某一商店的用户信息进行相关调研，例如经济情况(`balance`).并且因为一个用户可能拥有多个`store`，若要根据`user_id`来对其所拥有的所有商户进行分析，这种设计方式可以仅仅通过少量的冗余信息，加快查找速度。

4.索引设计

采用索引对于“频繁查找，不常更改”的数据项之查找功能来说起到了十分好的性能增益作用。我们在每个文档集合的ID字段上设置了索引(即主键). 为了允许对文本字段进行全文搜索，`book`文档集上设置了多个索引，分别是在字段 title,tags,book_intro,content上（这些字段的内容都是文本，所以类型都设置为了text）。

基本功能实现

用户功能

用户功能的后端逻辑实现在`/be/model/user.py`中，它提供了`User`类：

```
class User(db_conn.DBConn):
    token_lifetime: int = 3600  # 3600秒
    def __init__(self):
        db_conn.DBConn.__init__(self)
```

`User`类继承了`DBConn`类以进行数据库操作，下面我们介绍其中的基础功能方法。

注册用户

```
    def register(self, user_id: str, password: str) -> (int, str):
```

`register`函数用于用户注册，参数`user_id`为用户的id，`password`为用户的密码。函数返回一个元组，分别为状态码和消息，200 是我们预期的正确返回码。

首先检查用户是否已经存在：

```
        res = self.db.user.find_one({"user_id": user_id})
        if res is not None:
            return error.error_exist_user_id(user_id)
```

如果用户存在，返回错误码；否则，创建用户并生成 token 及终端信息：

```
        token = jwt_encode(user_id, terminal)
        self.db.user.insert_one({
            "user_id": user_id,
            "password": password,
            "balance": 0,
            "token": token,
            "terminal": terminal,
        })
```

登录用户

```
    def login(self, user_id: str, password: str, terminal: str) -> (int, str,
str):
```

`login` 函数用于用户登录，参数 `user_id` 为用户id，`password` 为用户密码，`terminal` 为登录终端标识。函数返回状态码、消息和 token。

首先通过 `check_password` 检查用户密码是否正确：

```
        code, message = self.check_password(user_id, password)
        if code != 200:
            return code, message, ""
```

如果密码正确，生成新的 token 并更新数据库：

```
        token = jwt_encode(user_id, terminal)
        self.db.user.update_one({"user_id": user_id}, {"$set": {"token": token,
"terminal": terminal}})
```

用户登出

```
 def logout(self, user_id: str, token: str) -> (int, str):
```

`logout` 函数用于用户登出，参数 `user_id` 为用户id，`token` 为登录时生成的令牌。函数返回状态码和消息。

首先验证 token 的有效性：

```
    code, message = self.check_token(user_id, token)
        if code != 200:
            return code, message
```

如果 token 有效，生成并更新一个临时 token：

```
python复制代码        dummy_token = jwt_encode(user_id, terminal)
        self.db.user.update_one({"user_id": user_id}, {"$set": {"token":
dummy_token, "terminal": terminal}})
```

注销用户

```
    def unregister(self, user_id: str, password: str) -> (int, str):
```

`unregister` 函数用于用户注销，参数 `user_id` 为用户id，`password` 为用户密码。函数返回状态码和消息。

通过 `check_password` 验证密码正确性后，删除用户记录：

```
code, message = self.check_password(user_id, password)
        if code != 200:
            return code, message

        self.db.user.delete_one({"user_id": user_id})
```

买家功能

买家功能的后端逻辑实现在 `/be/model/buyer.py` 中，它提供了Buyer类：

```
class Buyer(db_conn.DBConn):
    def __init__(self):
        super().__init__()
```

Buyer类继承了DBConn类来进行数据库操作，下面我们介绍其中的基础功能方法

下订单

```
    def new_order(self, user_id: str, store_id: str, id_and_count: [(str, int)])
-> (int, str, str):
```

`new_order` 函数用于买家下订单，参数 `user_id` 为买家的id，`store_id` 为商店的id，`id_and_count` 为一个列表，其中每个元素为一个元组，包含书本的id和数量。函数返回一个元组，分别为状态码，消息和订单id。和先前一样，200是我们预期的正确返回码。

在函数里，我们首先需要检查用户id和商店id是否合法

```
            user = self.db.user.find_one({"user_id": user_id})
            if user is None:
                return error.error_non_exist_user_id(user_id) + (order_id,)

            if self.store_id_exist(store_id) is False:
                return error.error_non_exist_store_id(store_id) + (order_id,)
```

如果不合法则返回相应的错误码。

紧接着，我们通过`id_and_count`在相应的商店里找到每一本书，如果没有找到返回书不存在的错误：

```
            for book_id, count in id_and_count:
                result = self.db.store.find_one(
                    {"store_id": store_id,"book_stock_info.book_id": book_id},
                    {"book_stock_info.$": 1}
                    )
                if result is None:
                    return error.error_non_exist_book_id(book_id) + (order_id,)
```

然后我们还要检查商家的库存值是否能满足用户的下单数量

```
                stock_level = result["book_stock_info"][0]["stock_level"]
                price = self.get_book_price(book_id)

                if stock_level < count:
                    return error.error_stock_level_low(book_id) + (order_id,)
```

如果上述条件都满足，则更新商家的库存：

```
                condition = {
                    "store_id": store_id,
                    "book_stock_info.book_id": book_id,
                    "book_stock_info.stock_level": {'$gte': count}
                }
                self.db.store.update_one(
                    condition,
                    {"$inc": {"book_stock_info.$.stock_level": -1}}
                )
```

并且我们需要维护订单详情表，现在一个列表里面，把每本书的信息加入进去

```
                each_book_in_order_details.append({
                    "book_id": book_id,
                    "count": count,
                    "price": price
                })
```

当所有书都处理完后，可以正式插入`new_order_details`表里，它的一条记录包含了订单id和每本书的信息的数组

```
            new_order_detail = {
                "order_id": uid,
                "each_book_details": each_book_in_order_details
            }
            self.db.new_order_detail.insert_one(new_order_detail)
```

最后我们将记录插入到`new_order`表里

```
            new_order = {
                "order_id": uid,
                "user_id": user_id,
                "store_id": store_id,
                "books_status": 2,
                "create_time": datetime.now(),
            }

            self.db.new_order.insert_one(new_order)
```

注意这里我们需要两个特殊的字段来实现我们的后续功能。首先是书本状态2，表示已下单未付款；其次是订单的创建时间，用于后续的超时取消订单功能。

支付

```
    def payment(self, user_id: str, password: str, order_id: str) -> (int, str):
```

`payment`函数用于买家支付订单，参数`user_id`为买家的id，`password`为密码，`order_id`为订单的id。函数返回状态码和消息。

首先我们检查`order_id`是否存在

```
            order_info = self.db.new_order.find_one({"order_id": order_id})
            if order_info is None:
                return error.error_invalid_order_id(order_id)
```

根据我们上一个函数中对`new_order`表的操作，我们可以从其中提取出用户的信息，如果出现了不一致也需要返回错误：

```
            order_id = order_info["order_id"]
            buyer_id = order_info["user_id"]
            store_id = order_info["store_id"]

            if buyer_id != user_id:
                return error.error_authorization_fail()
```

然后照常检查用户id和密码是否合法正确，我们还需要提取用户的余额信息：

```
            usr_info = self.db.user.find_one({"user_id": buyer_id})
            if usr_info is None:
                return error.error_non_exist_user_id(buyer_id)
            balance = usr_info["balance"]
            if password != usr_info["password"]:
                return error.error_authorization_fail()
```

然后检查商店id的合法性，并从商店表中提取出卖家的信息（即id，因为我们后续要操作卖家的余额），并检查卖家id的合法性

```
            store_info = self.db.user_store.find_one({"store_id": store_id})
            if  store_info is None:
                return error.error_non_exist_store_id(store_id)

            seller_id = store_info["user_id"]

            if not self.user_id_exist(seller_id):
                return error.error_non_exist_user_id(seller_id)
```

接着我们要计算订单的总价，这个信息需要从订单详情表里面找，遍历每本书，计算每本书的价格乘以数量的总和

```
            new_order_details_info = self.db.new_order_detail.find({"order_id":
order_id})
            total_price = 0
            for order_detail in new_order_details_info:
                for book in order_detail["each_book_details"]:
                    # 每本书的价格 * 数量
                    total_price += book["price"] * book["count"]
```

注意这边我们需要处理如果用户余额不足以支付总价，需要返回错误

```
            if balance < total_price:
                return error.error_not_sufficient_funds(order_id)
```

接着我们更新买家和卖家的余额，为了代码稳健性，如果发现更新失败时也返回错误：

```
            result = self.db.user.update_many(
                {"user_id": buyer_id, "balance": {"$gte": total_price}},
                {"$inc": {"balance": -total_price}}
            )
            if result.modified_count == 0:
                return error.error_not_sufficient_funds(order_id)

            result = self.db.user.update_many(
                {"user_id": seller_id},
                {"$inc": {"balance": total_price}}
            )
            if result.modified_count == 0:
                return error.error_not_sufficient_funds(order_id)
```

由于我们需要实现收获发货，查询历史订单等，所以不再需要在支付后删除订单和订单详情。但我们需要做的是更新订单的状态，即把订单状态由2（未支付）改为1（已支付，未发货）

```
            result = self.db.new_order.update_one(
                {"order_id": order_id, "books_status": 2},
                {"$set": {"books_status": 1}}
            )
            if result.matched_count == 0:
                return error.error_invalid_order_id(order_id)
```

添加余额

```
    def add_funds(self, user_id, password, add_value) -> (int, str):
```

`add_funds`函数用于买家添加余额，参数`user_id`为买家的id，`password`为密码，`add_value`为要添加的余额。函数返回状态码和消息。

首先找到用户，检查用户id和密码是否合法正确

```
            user_info = self.db.user.find_one({"user_id": user_id})

            if user_info is None:
                return error.error_authorization_fail()

            if user_info.get("password") != password:
                return error.error_authorization_fail()
```

然后更新用户的`balance`字段即可

```
            res = self.db.user.update_one({"user_id": user_id}, {"$inc":
{"balance": add_value}})
            if res.matched_count == 0:
                return error.error_non_exist_user_id(user_id)
```

卖家功能

卖家功能的后端逻辑实现在`bookstore/be/model/seller.py`中，它提供了`Seller`类。

```
class Seller(db_conn.DBConn):
    def __init__(self):
        db_conn.DBConn.__init__(self)
```

其继承了`DBConn`类来进行数据库操作，下面是其基础功能方法。

添加图书

```
def add_book(
    self,
    user_id: str,        # 用户 id
    store_id: str,       # 店铺 id
    book_id: str,        # 书本 id
    book_json_str: str, # 书本详细信息（JSON 字符串）
    stock_level: int,    # 书本的库存水平
) -> (int, str):         # 返回一个整数状态码和信息
```

函数的具体实现步骤如下：

1. 判断用户是否存在，若不存在返回错误；
2. 判断店铺是否存在，若不存在返回错误；
3. 判断书本是否存在，若不存在返回错误；
4. 将相应的信息插入到数据库中；
5. 若在插入过程中发生异常，返回错误码 528/530 和错误信息；

6. 若运行正常，返回 200 "ok"。

添加库存

```
def add_stock_level(
    self,
    user_id: str,         # 用户 id
    store_id: str,        # 店铺 id
    book_id: str,         # 书本 id
    add_stock_level: int # 要添加的库存数
) -> (int, str):          # 返回一个整数状态码和信息
```

函数的具体实现步骤如下：

1. 判断用户是否存在，若不存在返回错误；
2. 判断店铺是否存在，若不存在返回错误；
3. 判断书本是否存在，若不存在返回错误；
4. 将相应的信息插入到数据库中，并检查；
5. 若在插入或检查过程中发生异常，返回错误码 528/530 和错误信息；
6. 若运行正常，返回 200 "ok"。

添加店铺

```
def create_store(
    self,
    user_id: str, # 用户 id
    store_id: str # 店铺 id
) -> (int, str):  # 返回一个整数状态码和信息
```

函数的具体实现步骤如下：

1. 判断用户是否存在，若不存在返回错误；
2. 判断店铺是否存在，若不存在返回错误；
3. 将相应的信息插入到 `store` 和 `user_store` 数据库中；
4. 若在插入过程中发生异常，返回错误码 528/530 和错误信息；
5. 若运行正常，返回 `200 "ok"`。

拓展功能的实现

发货与收货

发货代码路径：`be/model/seller.py`

```
def send_books(self, user_id: str, order_id: str):
```

- `send_books` 函数接受两个参数：`user_id`：尝试将图书标记为已发货的用户 ID;`order_id`：要更新的订单 ID。

```
res = self.db.new_order.find_one({"order_id": order_id})
if res is None:
    return error.error_invalid_order_id(order_id)
```

根据 `order_id` 查找订单。如果订单不存在，则返回“无效订单 ID”错误。

```
store_id = res["store_id"]
books_status = res["books_status"]

result = self.db.user_store.find_one({"store_id": store_id})
seller_id = result["user_id"]

if seller_id != user_id:
    return error.error_authorization_fail()
```

查找与订单关联的 `store_id`，并检查 `user_id` 是否与商店所有者 `seller_id` 匹配。如果不匹配，则返回授权失败错误。

```
if books_status == 0:
    return error.error_book_has_sent(order_id)

if books_status == 2:
    return error.error_not_paid_book(order_id)

if books_status == 3:
    return error.error_book_has_received(order_id)
```

根据 `books_status` 值，检查订单的当前状态：

- 若为 `0`，表示书籍已发货，返回相应错误消息。
- 若为 `2`，表示书籍未付款，返回相应错误消息。
- 若为 `3`，表示书籍已收货，返回相应错误消息。

```
res = self.db.new_order.update_one({"order_id": order_id}, {"$set":
{"books_status": 0}})
assert res.modified_count > 0  # 仅用于调试，多进程时可能为0
return 200, "ok"
```

1. 若订单状态允许发货，则更新 `books_status` 为 0表示书籍已发货。若 `modified_count` 不大于 `0`，在调试模式下将触发断言（多进程情况下可能等于零）。
2. 若操作成功，返回状态码 `200` 和消息 `"ok"`

收货代码路径：`be/model/buyer.py`

```
    """
    - `user_id`：用户的ID。
    - `order_id`：订单ID。
    return: status_code,msg
    """
    def receive_book(self,user_id: str, order_id: str) -> (int, str):
        try :
            #从 new_order集合中查找与 order_id 匹配的订单信息。如果订单不存在，返回状态码
和无效订单ID错误消息。
```

```
            res = self.db.new_order.find_one({"order_id": order_id})
            if res == None:
                return error.error_invalid_order_id(order_id)
            buyer_id = res["user_id"]
            paid_status = res["books_status"]
            #从查询结果中获取 buyer_id）和 books_status（订单状态）。如果 `buyer_id` 与传入的 `user_id` 不一致，返回授权失败的状态码和错误消息。
            if buyer_id != user_id:
                return error.error_authorization_fail()

            if paid_status == 1:
                return error.error_books_not_sent(order_id)#已经付钱，但是没有发货
            if paid_status == 2:
                return error.error_books_receive_without_payment(order_id)#没有付款就receive
            if paid_status == 3:
                return error.error_books_repeat_receive(order_id)
            self.db.new_order.update_one({"order_id": order_id}, {"$set": {"books_status": 3}})#订单已确认收货
        except BaseException as e:
            return 528, "{}".format(str(e))
        return 200, "ok"
```

图书查询

查询的代码在 `bookstore/be/model/book_searcher.py` 中

查询给定标题的图书

```
def search_title_in_store(
    self,
    title: str,         # 要搜索的标题
    store_id: str,      # 店铺的 id
    page_num: int,      # 页数（分页查询）
    page_size: int      # 页大小（分页查询）
): -> (int, str, list) # 返回状态码、消息和查询的列表
```

该函数实现在店铺里查找给定标题的图书，操作如下

1. 设置搜索条件，查询，并按要求整理；
2. 按店铺进行筛选；
3. 若搜索结果为空，返回错误，否则返回 `200 "ok"`。

若不指定店铺，直接将 `store_id` 置为空串。

```
def search_title(self, title: str, page_num: int, page_size: int):
    return self.search_title_in_store(title, "", page_num, page_size)
```

查询给定标签的图书

```
def search_tag_in_store(
    self,
    tag: str,           # 要搜索的标签
    store_id: str,      # 店铺的 id
    page_num: int,      # 页数（分页查询）
    page_size: int      # 页大小（分页查询）
): -> (int, str, list) # 返回状态码、消息和查询的列表
```

该函数实现在店铺里查找给定标题的图书，操作如下

1. 设置搜索条件，查询，并按要求整理；
2. 按店铺进行筛选；
3. 若搜索结果为空，返回错误，否则返回 `200 "ok"`。

若不指定店铺，直接将 `store_id` 置为空串。

```
def search_tag(self, tag: str, page_num: int, page_size: int):
    return self.search_tag_in_store(tag, "", page_num, page_size)
```

查询给定内容的图书

```
def search_content_in_store(
    self,
    content: str,       # 要搜索的内容
    store_id: str,      # 店铺的 id
    page_num: int,      # 页数（分页查询）
    page_size: int      # 页大小（分页查询）
): -> (int, str, list) # 返回状态码、消息和查询的列表
```

该函数实现在店铺里查找给定标题的图书，操作如下

1. 设置搜索条件，查询，并按要求整理；
2. 按店铺进行筛选；
3. 若搜索结果为空，返回错误，否则返回 `200 "ok"`。

若不指定店铺，直接将 `store_id` 置为空串。

```
def search_content(self, content: str, page_num: int, page_size: int):
    return self.search_content_in_store(content, "", page_num, page_size)
```

查询给定作者的图书

```
def search_author_in_store(
    self,
    author: str,        # 要搜索的作者
    store_id: str,      # 店铺的 id
    page_num: int,      # 页数（分页查询）
    page_size: int      # 页大小（分页查询）
): -> (int, str, list) # 返回状态码、消息和查询的列表
```

该函数实现在店铺里查找给定标题的图书，操作如下

1. 设置搜索条件，查询，并按要求整理；
2. 按店铺进行筛选；
3. 若搜索结果为空，返回错误，否则返回 `200 "ok"`。

若不指定店铺，直接将 `store_id` 置为空串。

```
def search_author(self, author: str, page_num: int, page_size: int):
    return self.search_content_in_store(author, "", page_num, page_size)
```

历史订单查询

```
    def search_order(self, user_id: str, password: str) -> (int, str, [(str,
str, str, int, int, int)]):
```

`search_order`函数用于买家查询历史订单，参数`user_id`为买家的id，`password`为密码。函数返回状态码，消息和一个列表，其中每个元素包含订单id，商店id，书本id，数量，价格和订单状态。

还是和之前一样，首先检查用户id和密码是否合法正确

```
            user_info = self.db.user.find_one({"user_id": user_id})
            if user_info is None:
                return error.error_authorization_fail() + ([])
            if user_info.get("password") != password:
                return error.error_authorization_fail() + ([])
```

然后我们找到用户的所有订单

```
            res = self.db.new_order.find({"user_id": user_id})
```

这里有一个特殊情况，就是用户没有任何订单，此时我们正常返回一个空列表并提示

```
            if res is None:
                return 200, "no orders", []
```

接着我们对于每个订单，提取出订单的id，商店id和状态，并用订单id在订单详情表里查找

```
            for row in res:
                order_id = row["order_id"]
                store_id = row["store_id"]
                status = row["books_status"]
                order_de = self.db.new_order_detail.find({"order_id": order_id})
```

然后我们对于每个订单详情，提取出书本id，数量和价格，并加入到返回的列表里

```
                for order_detail in order_de:
                    for book in order_detail["each_book_details"]:
                        book_id = book["book_id"]
                        count = book["count"]
                        price = book["price"]
                        order_list.append((order_id, store_id, book_id, count,
price, status))
```

订单取消

用户手动取消

```
    def cancel_order(self, user_id: str, password: str, order_id: str) -> (int,
str):
```

`cancel_order`函数用于买家取消订单，参数`user_id`为买家的id，`password`为密码，`order_id`为订单的id。函数返回状态码和消息。

判断合法性，找到订单对应的商店id和书籍状态，以及订单详情

```
            user_info = self.db.user.find_one({"user_id": user_id})
            if user_info is None:
                return error.error_authorization_fail()
            if user_info.get("password") != password:
                return error.error_authorization_fail()

            # find order
            res = self.db.new_order.find_one({"order_id": order_id})
            if res == None:
                return error.error_invalid_order_id(order_id)

            # check order status
            status = res["books_status"]
            store_id = res["store_id"]
            order_list = self.db.new_order_detail.find({"order_id": order_id})
```

接下来我们需要分情况考虑，即考虑订单的状态。最简单的是订单未付款，此时我们只需要返还书籍，即更新库存信息

```
            if status == 2:
                # 返还书籍
                for order_detail in order_list:
                    for book in order_detail["each_book_details"]:
                        book_id = book["book_id"]
                        count = book["count"]
                        self.db.store.update_one({"store_id": store_id,
"book_stock_info.book_id": book_id}, {"$inc": {"book_stock_info.$.stock_level":
count}})
                self.db.new_order.delete_one({"order_id": order_id})
                self.db.new_order_detail.delete_many({"order_id": order_id})
```

对于每一本书找到数量，并通过之前提取的商店信息更新商店库存。之后为了简单起见，我们直接在两个订单表里删除订单信息。

另外一种情况，已付款未发货要复杂一些，因为这时候除了返还书籍之外，我们还需要退款。因此这里我们需要像之前的支付函数一样，只不过这里我们要检查的是卖家的余额够不够书本的总价

```
            elif status == 1:
                # 检查商家余额是否足够退款
                store_info = self.db.user_store.find_one({"store_id": store_id})
                seller_id = store_info["user_id"]
```

```
                seller_info = self.db.user.find_one({"user_id": seller_id})
                total_price = 0
                for order_detail in order_list:
                    for book in order_detail["each_book_details"]:
                        price = book["price"]
                        count = book["count"]
                        total_price += count * price

                if seller_info["balance"] < total_price:
                    return error.error_not_sufficient_funds(order_id)
```

如果没有问题，就和之前一样返还书籍，当然还需要更新买家和卖家的余额字段来执行退款，并删除订单

```
                # 返还书籍
                for order_detail in order_list:
                    for book in order_detail["each_book_details"]:
                        book_id = book["book_id"]
                        count = book["count"]
                        self.db.store.update_one({"store_id": store_id,
"book_stock_info.book_id": book_id}, {"$inc": {"book_stock_info.$.stock_level":
count}})

                # 返还金额
                self.db.user.update_one({"user_id": seller_id}, {"$inc":
{"balance": -total_price}})
                self.db.user.update_one({"user_id": user_id}, {"$inc":
{"balance": total_price}})
                self.db.new_order.delete_one({"order_id": order_id})
                self.db.new_order_detail.delete_many({"order_id": order_id})
```

当然还有两种错误情况需要我们处理，即在我们的简单实现里面，卖家发货后是不可以取消订单的

```
            elif status == 0:
                return error.error_book_has_sent(order_id)
            elif status == 3:
                return error.error_book_has_received(order_id)
```

未支付超时自动取消

这个功能相对来说比较复杂，因为它需要一个后台任务来监控超市的订单。为了实现这个功能，我们在`Buyer`类的初始化中添加了启动这个任务的函数

```
    def __init__(self):
        super().__init__()
        self.cleanup_thread = None
        self.is_running = False
        self.start_cleanup_thread()
```

在这里我们利用了`threading`库，为这个线程配置启动和结束函数，在启动时我们调用我们定义的清理过期订单的函数，这里`daemon=True`就表示这个线程是后台守护线程

```
    def start_cleanup_thread(self):
        #启动后台清理线程
        self.is_running = True
        self.cleanup_thread =
threading.Thread(target=self._cleanup_expired_orders, daemon=True)
        self.cleanup_thread.start()
        logging.info("Started order cleanup thread")

    def stop_cleanup_thread(self):
        #停止后台清理线程
        self.is_running = False
        if self.cleanup_thread:
            self.cleanup_thread.join()
            logging.info("Stopped order cleanup thread")
```

为了确保资源回收，我们在`Buyer`类的析构函数中停止这个线程

```
    def __del__(self):
        #确保线程在对象销毁时正确关闭
        self.stop_cleanup_thread()
```

确保这些前置工作准备完毕后，我们实现清理过期订单的函数

```
    def _cleanup_expired_orders(self):
```

它的整体逻辑应该是，持续运行，但隔一段时间检查一次。即：

```
        #清理过期订单的后台任务
        while self.is_running:

            # 代码逻辑实现

            # 每10秒检查一次
            time.sleep(10)
```

这里为了测试方便，我们设置了每10s检查一次，当然在实际生产环境中我们需要调整。

接着我们实现代码逻辑。同样为了测试方便，我们这里的过期时间只有20s，现实中自然是不可能的

```
                # 计算20秒前的时间点
                expire_time = datetime.now() - timedelta(seconds=20)
```

接着我们找出早于超时时间创建，并且未支付（即状态为2）的订单，找到他们的`order_id`，并在两个订单表里删除：

```
                # 先找到过期订单的order_id，再在new_order和new_order_detail中删除
                expired_orders = self.db.new_order.find({"create_time": {"$lt":
expire_time}, "books_status": 2})
                for order in expired_orders:
                    order_id = order["order_id"]
                    self.db.new_order.delete_many({"order_id": order_id})
                    self.db.new_order_detail.delete_many({"order_id": order_id})
```

到这里我们实现了自动取消超时订单的功能。不过这个功能的实现其实由很多可以探讨的地方，我们在这里的实现可能是不够高效的，难以满足实际生产环境的。在未来，我们可以考虑利用数据库的TTL（Time-To-Live）索引，或者是采用更高级的后台任务API和调度工具，例如`apischeduler`等。这个问题需要考虑高并发场景下的性能问题，以及如何保证任务的可靠性和一致性等，有待我们未来进一步探索。

书籍推荐

由于本次实验中我们实现了用户历史订单查询和图书搜索两个功能，因此一个自然的想法是能否把这两个功能结合起来做一些事情，我们据此设计了一个简易的“猜你想看”：

```
    def recommend_books(self, user_id: str, password: str):
```

`recommend_books`函数用于为买家推荐书籍，参数`user_id`为买家的id，`password`为密码。函数返回状态码，消息和推荐的书的列表。
按惯例检查用户id和密码，不再赘述：

```
            # find user
            user_info = self.db.user.find_one({"user_id": user_id})
            if user_info is None:
                return error.error_authorization_fail() + ([])
            if user_info.get("password") != password:
                return error.error_authorization_fail() + ([])
```

接着调用我们之前实现的历史订单查询函数，找到用户的所有订单。注意函数返回的前面两个元素是返回码和消息，最后才是订单列表。

```
            # find order
            order_list =self.search_order(user_id, password)[2]
```

由于我们的简易推荐基于用户的历史订单，如果用户没有历史订单的话，我们也就没法推荐了

```
            # 没有先前订单无法推荐
            if not order_list:
                return 200, "no orders", []
```

接下来我们要使用我们编写的图书搜索功能，创建一个实例

```
            # 初始化一个booksearcher
            book_searcher = BookSearcher()
```

然后我们遍历用户的所有订单，找到每个订单的书本id，在数据库中找到这本书的信息，我们关系的是标题，作者和标签的列表（用换行符分割为列表）。（注意根据我们之前的搜索订单的返回值，列表的第三个元素是书本id）

```
            # 在order_list中提取book_id，根据book_id，从self.db.book中搜索相应的书的标题，作者和标签
            book_list = []
            for order in order_list:
                book_id = order[2]
                book_info = self.db.book.find_one({"id":book_id})
                book_list.append((book_id, book_info["title"], book_info["author"], book_info["tags"].split("\n")))
```

一般来说，我们不会想给用户推荐他已经看过的书，但对这个数据库来说，精确实现这一点可能比较麻烦，所以简单来说我们维护一个用户买过的书的标题的集合，不推荐相同标题的书

```
            # 维护一个title的集合，因为不能推荐用户买过的书
            title_set = set()
            author = {}
            tags = []
```

接着我们遍历用户买过的书，统计有哪些作者，各买了多少本，并合并所有标签的列表

```
            for book in book_list:
                title_set.add(book[1])

                # 找到用户读过最多的作者
                if book[2] not in author:
                    author[book[2]] = 1
                else:
                    author[book[2]] += 1

                # 合并所有tags列表，找到用户读过最多的标签
                tags += book[3]
```

找到用户看过最多的作者，类似的找到看过最多的标签

```
            max_author = max(author, key=author.get)

            # 找到最多的标签
            tag = {}
            for t in tags:
                if t not in tag:
                    tag[t] = 1
                else:
                    tag[t] += 1

            max_tag = max(tag, key=tag.get)
```

在未来我们可以进一步拓展数量，不过在这里简单起见我们只是推荐看过最多的作者和最多的标签。然后用book_searcher中我们实现的方法检索

```
            # 根据这些信息，调用book_searcher的search_author, search_tag方法，分别搜索
            author_books = book_searcher.search_author(max_author, 1, 1)
            tag_books = book_searcher.search_tag(max_tag, 1, 1)
```

对于每一个搜索结果我们都需要首先判断是否非空，如果有结果的话，检查里面每一本书的标题，如果不在用户买过的里面就加入推荐书籍列表

```
            # 合并两个搜索结果，去掉用户买过的书
            recommend_books = []
            # 先判断搜出来的是不是空的
            if author_books[2]:
                for book in author_books[2]:
                    if book["title"] not in title_set:
                        recommend_books.append(book)

            if tag_books[2]:
                for book in tag_books[2]:
                    if book["title"] not in title_set:
                        recommend_books.append(book)
```

保险起见，我们最后再给推荐列表做一个去重：

```
            # recommend_books转化为集合防止重复
            recommend_books = list(set(recommend_books))
```

有一种很特殊的情况，虽然不怎么可能发生，就是两个搜索的结果都为空，最后的推荐列表也为空，不过这个情况并不影响我们程序的正确性。

到这里我们就完成了简易的推荐功能。在实际生产环境中，我们需要更复杂的推荐算法，例如协同过滤，基于内容的推荐，深度学习等。这些算法需要更多的数据，更多的计算资源，更多的调优，有待我们未来进一步探索。不过在这里我们成功用我们先前实现的功能组合成了一个新的实用模块，也是一个有趣的尝试。

新增接口

前端接口fe/access：

这里主要是根据新增的路由添加对应的参数处理函数，将参数以post方式发送至对应的路由。这里值得一提的是，在debug过程中，若没有涉及比较复杂的数据准备，我们可以直接用postman工具来模拟访问某一路由。

后端接口在 be/view/ ：

这里的修改与前端同理，主要是接受前端传递来的参数，然后根据其路由选择对应的函数进行处理。值得注意的是，我们并没有给自动取消添加接口，因为这是一个在后台自动运行的任务，在正常的逻辑中不会涉及前端访问。

新增的测试样例

`test_receive_and_delivery.py`

收获和发货测试代码路径：`bookstore\fe\test\test_receive_and_delivery.py`。测试说明如下

- `test_send_ok`:检查是否正常发货
- `test_invalid_order_id_send`：检查订单ID不存在的情况下尝试发货时，是否会检查出错误。
- `test_authorization_error_send`：检查使用不匹配的卖家ID尝试发货是，是否会检查出错误。
- `test_books_repeat_send`：检查重复发货后，能否检测出发货失败的情况。
- `test_receive_ok`：检测买家付款后，卖家发货，买家接收是否成功。
- `test_invalid_order_id_receive`：检测在订单ID不存在的情况下尝试接收时，是否返回错误状态。
- `test_authorization_error_receive`：检查使用不匹配的买家ID尝试接收时，是否会返回错误状态。
- `test_books_not_send_receive`：检查订单未发货情况下尝试接收时，是否返回错误状态。
- `test_books_repeat_receive`：检查买家在首次接收成功后再次尝试接收，是否会返回错误状态。

`TestSearchOrder`

这个测试用例用于测试历史订单查询功能，包含

- `test_search_order_no_order` 检查用户没有订单时，是否按我们规定正常返回
- `test_search_order_authorization_error` 检查用户密码错误
- `test_search_order_non_exist_user_id` 检查用户id不存在
- `test_search_order_ok` 检查正常情况

`TestCancel`

这个测试用例用于测试订单取消功能，包含

- `test_cancel_order_and_repeated_cancel` 这是一个综合的测试，我们同时连续取消两次相同的一个订单，检查是否第一次正常第二次错误
- `test_cancel_order_authorization_error` 检查用户密码错误
- `test_cancel_order_non_exist_order_id` 检查订单id不存在
- `test_cancel_order_non_exist_user_id` 检查用户id不存在
- `test_cancel_order_after_pay` 检查已支付订单后立刻取消是否能够正常取消
- `test_cancel_order_after_send` 检查已发货订单是否能够取消，应当不可以
- `test_cancel_order_after_receive` 检查已收货订单是否能够取消，应当不可以

`TestAutoCancel`

这个测试用例用于测试自动取消功能，包含

- `test_auto_cancel` 检查自动取消功能是否正常。它的场景是，在创建新订单后等待30s再去主动取消订单，这时候应该返回错误，因为已经被自动取消了
- `test_normal_cancel` 这次我们只等待5s就主动取消，应该正常取消，因为还没有超时
- `test_cancel_after_pay` 这里是下单后支付，然后等待30s再主动取消，应当正确，因为不会自动取消已经支付的订单

`TestRecommendBooks`

这个测试用例用于测试推荐书籍功能，包含

- `test_recommend_no_order` 检查用户没有订单时，是否按我们规定正常返回
- `test_recommend_authorization_error` 检查用户密码错误
- `test_recommend_non_exist_user_id` 检查用户id不存在
- `test_recommend_ok` 检查正常情况

`TestSearch`

这个测试用例用于测试查找功能。初始化后可调用以下方法进行测试：

- `test_all_field_search` 测试全局搜索功能是否正常
- `test_pagination` 测试分页功能是否正常(调用 `buyer.search` 查询时指定页的大小；)
- `test_search_title` 测试查找标题功能是否正常.
- `test_test_search_title_in_store` 测试按店铺查找标题功能是否正常.
- `test_search_tag` 测试查找标签功能是否正常.
- `test_search_tag_in_store` 测试按店铺查找标签功能是否正常.
- `test_search_author` 测试查找作者功能是否正常.
- `test_search_author_in_store` 测试按店铺查找作者功能是否正常.
- `test_search_context` 测试查找目录功能是否正常.
- `test_search_context_in_store` 测试按店铺查找目录功能是否正常.

测试结果

```
collected 71 items

bookstore/fe/test/test_add_book.py::TestAddBook::test_ok PASSED                                    [  1%]
bookstore/fe/test/test_add_book.py::TestAddBook::test_error_non_exist_store_id PASSED              [  2%]
bookstore/fe/test/test_add_book.py::TestAddBook::test_error_exist_book_id PASSED                   [  4%]
bookstore/fe/test/test_add_book.py::TestAddBook::test_error_non_exist_user_id PASSED               [  5%]
bookstore/fe/test/test_add_funds.py::TestAddFunds::test_ok PASSED                                  [  7%]
bookstore/fe/test/test_add_funds.py::TestAddFunds::test_error_user_id PASSED                       [  8%]
bookstore/fe/test/test_add_funds.py::TestAddFunds::test_error_password PASSED                      [  9%]
bookstore/fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_user_id PASSED            [ 11%]
bookstore/fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_store_id PASSED           [ 12%]
bookstore/fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_book_id PASSED            [ 14%]
bookstore/fe/test/test_add_stock_level.py::TestAddStockLevel::test_ok PASSED                       [ 15%]
bookstore/fe/test/test_auto_cancel.py::TestAutoCancel::test_auto_cancel PASSED                     [ 16%]
bookstore/fe/test/test_auto_cancel.py::TestAutoCancel::test_normal_cancel PASSED                   [ 18%]
bookstore/fe/test/test_auto_cancel.py::TestAutoCancel::test_cancel_after_pay PASSED                [ 19%]
bookstore/fe/test/test_bench.py::test_bench PASSED                                                 [ 21%]
bookstore/fe/test/test_cancel.py::TestCancel::test_cancel_order_and_repeated_cancel PASSED         [ 22%]
bookstore/fe/test/test_cancel.py::TestCancel::test_cancel_order_authorization_error PASSED         [ 23%]
bookstore/fe/test/test_cancel.py::TestCancel::test_cancel_order_non_exist_order_id PASSED          [ 25%]
bookstore/fe/test/test_cancel.py::TestCancel::test_cancel_order_non_exist_user_id PASSED           [ 26%]
bookstore/fe/test/test_cancel.py::TestCancel::test_cancel_order_after_pay PASSED                   [ 28%]
bookstore/fe/test/test_cancel.py::TestCancel::test_cancel_order_after_send PASSED                  [ 29%]
bookstore/fe/test/test_cancel.py::TestCancel::test_cancel_order_after_receive PASSED               [ 30%]
bookstore/fe/test/test_create_store.py::TestCreateStore::test_ok PASSED                            [ 32%]
bookstore/fe/test/test_create_store.py::TestCreateStore::test_error_exist_store_id PASSED          [ 33%]
bookstore/fe/test/test_login.py::TestLogin::test_ok PASSED                                         [ 35%]
bookstore/fe/test/test_login.py::TestLogin::test_error_user_id PASSED                              [ 36%]
bookstore/fe/test/test_login.py::TestLogin::test_error_password PASSED                             [ 38%]
bookstore/fe/test/test_new_order.py::TestNewOrder::test_non_exist_book_id PASSED                   [ 39%]
bookstore/fe/test/test_new_order.py::TestNewOrder::test_low_stock_level PASSED                     [ 40%]
bookstore/fe/test/test_new_order.py::TestNewOrder::test_ok PASSED                                  [ 42%]
bookstore/fe/test/test_new_order.py::TestNewOrder::test_non_exist_user_id PASSED                   [ 43%]
bookstore/fe/test/test_new_order.py::TestNewOrder::test_non_exist_store_id PASSED                  [ 45%]
bookstore/fe/test/test_password.py::TestPassword::test_ok PASSED                                   [ 46%]
bookstore/fe/test/test_password.py::TestPassword::test_error_password PASSED                       [ 47%]
bookstore/fe/test/test_password.py::TestPassword::test_error_user_id PASSED                        [ 49%]
bookstore/fe/test/test_payment.py::TestPayment::test_ok PASSED                                     [ 50%]
```

```
bookstore/fe/test/test_payment.py::TestPayment::test_ok PASSED                                                    [ 50%]
bookstore/fe/test/test_payment.py::TestPayment::test_authorization_error PASSED                                   [ 52%]
bookstore/fe/test/test_payment.py::TestPayment::test_not_suff_funds PASSED                                        [ 53%]
bookstore/fe/test/test_payment.py::TestPayment::test_repeat_pay PASSED                                            [ 54%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_send_ok PASSED                                  [ 56%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_authorization_error_send PASSED                 [ 57%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_invalid_order_id_send PASSED                    [ 59%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_books_repeat_send PASSED                        [ 60%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_books_not_paid_receive PASSED                   [ 61%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_receive_ok PASSED                               [ 63%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_authorization_error_receive PASSED              [ 64%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_invalid_order_id_receive PASSED                 [ 66%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_books_not_send_receive PASSED                   [ 67%]
bookstore/fe/test/test_receive_and_delivery.py::TestReceive::test_books_repeat_receive PASSED                     [ 69%]
bookstore/fe/test/test_recommend.py::TestRecommendBooks::test_recommend_authorization_error PASSED                [ 70%]
bookstore/fe/test/test_recommend.py::TestRecommendBooks::test_recommend_non_exist_user_id PASSED                  [ 71%]
bookstore/fe/test/test_recommend.py::TestRecommendBooks::test_recommend_no_order PASSED                           [ 73%]
bookstore/fe/test/test_recommend.py::TestRecommendBooks::test_recommend_ok PASSED                                 [ 74%]
bookstore/fe/test/test_register.py::TestRegister::test_register_ok PASSED                                         [ 76%]
bookstore/fe/test/test_register.py::TestRegister::test_unregister_ok PASSED                                       [ 77%]
bookstore/fe/test/test_register.py::TestRegister::test_unregister_error_authorization PASSED                      [ 78%]
bookstore/fe/test/test_register.py::TestRegister::test_register_error_exist_user_id PASSED                        [ 80%]
bookstore/fe/test/test_search_book.py::TestSearch::test_all_field_search PASSED                                   [ 81%]
bookstore/fe/test/test_search_book.py::TestSearch::test_pagination PASSED                                         [ 83%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_title PASSED                                       [ 84%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_title_in_store PASSED                              [ 85%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_tag PASSED                                         [ 87%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_tag_in_store PASSED                                [ 88%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_author PASSED                                      [ 90%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_author_in_store PASSED                             [ 91%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_content PASSED                                     [ 92%]
bookstore/fe/test/test_search_book.py::TestSearch::test_search_content_in_store PASSED                            [ 94%]
bookstore/fe/test/test_search_order.py::TestSearchOrder::test_search_order_no_order PASSED                        [ 95%]
bookstore/fe/test/test_search_order.py::TestSearchOrder::test_search_order_authorization_error PASSED             [ 97%]
bookstore/fe/test/test_search_order.py::TestSearchOrder::test_search_order_non_exist_user_id PASSED               [ 98%]
bookstore/fe/test/test_search_order.py::TestSearchOrder::test_search_order_ok PASSED                              [100%]

============================================== 71 passed in 706.75s (0:11:46) ==============================================
```

这里的测试总时间会略高于实际的运行时间，主要是因为我们在订单超时取消部分，所等待的时间较多。以及部分新建的测试样例会涉及比较多和比较复杂的数据准备。

覆盖率如下：

```
---------- coverage: platform win32, python 3.7.0-final-0 -----------
Name                                      Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------------
bookstore\be\__init__.py                      0      0      0      0   100%
bookstore\be\app.py                           9      9      2      0     0%
bookstore\be\model\book_searcher.py          49      2     16      0    97%
bookstore\be\model\buyer.py                 278     44    128     20    83%
bookstore\be\model\db_conn.py                24      4      2      1    81%
bookstore\be\model\error.py                  37      3      0      0    92%
bookstore\be\model\seller.py                 67     15     26      3    81%
bookstore\be\model\store.py                  32      3      0      0    91%
bookstore\be\model\user.py                  113     25     32      6    79%
bookstore\be\serve.py                        42      2      2      1    93%
bookstore\be\view\auth.py                    42      0      0      0   100%
bookstore\be\view\buyer.py                   71      0      2      0   100%
bookstore\be\view\search.py                  86      4     32     12    86%
bookstore\be\view\seller.py                  38      0      0      0   100%
bookstore\fe\__init__.py                      0      0      0      0   100%
bookstore\fe\access\__init__.py               0      0      0      0   100%
bookstore\fe\access\auth.py                  31      0      0      0   100%
bookstore\fe\access\book.py                  59      0      6      0   100%
bookstore\fe\access\buyer.py                 67      1      4      1    97%
bookstore\fe\access\new_buyer.py              8      0      0      0   100%
bookstore\fe\access\new_seller.py             8      0      0      0   100%
bookstore\fe\access\search.py                53      0      0      0   100%
bookstore\fe\access\seller.py                37      0      0      0   100%
bookstore\fe\bench\__init__.py                0      0      0      0   100%
bookstore\fe\bench\run.py                    13      0      6      0   100%
bookstore\fe\bench\session.py                47      0     12      1    98%
bookstore\fe\bench\workload.py              125      1     20      2    98%
bookstore\fe\conf.py                         11      0      0      0   100%
bookstore\fe\conftest.py                     19      0      0      0   100%
bookstore\fe\test\gen_book_data.py           49      1     16      1    97%
-------------------------------------------------------------------------
TOTAL                                      1415    114    306     48    90%
```

在部分文件发现覆盖率在80%左右，这是因为我们在这里添加了部分debug专用的函数，便于程序员在运行时进行相关快捷的调试。

而总体的覆盖率达到了90%，这是一个比较不错的数值了。

遇到的问题

因为本实验的文档数据mongodb在访问一个集合内不存在的属性时，会返回None值,而不是像`postgresql`一样会报错。这就导致在进行一些创建操作时，需要及时对一些属性进行初始化，例如对数组元素，给一个空数组，利于后续`append`操作。否则会造成程序正常运行，但测试样例无法通过的情况。

github协作图

代码仓库地址：https://github.com/LL1122LL/bookstore/commits/master/。

```
PS E:\juypter\DataSql\bookstore3\CDMS.Xuan_ZHOU.2024Fall.DaSE\project1> git log --graph --decorate --pretty=oneline --abbrev-commit
 --all
* 1d8f97a (HEAD -> master, origin/master) 更新报告，修正错误
* c412bbb 增加了卖家后端基本操作和查找图书的功能，新增查找图书的测试样例。
* 24faa2e 新增基于历史订单查询和图书搜索的简易推荐功能，及相应测试以及报告
* bd025ea 补充了两个和收获发货联动的取消订单测试
* f4c701b small rectify
* 7c7fdda finish all!pass all test!
* 0929343 完成search代码，能独立通过测试，但不能通过所有测试
* 354ea6d pass search test
* b8d2de9 增添delivery test
* 314e343 根据新的detail表调整了相应功能
* a267c1c add index and adjust structure of new_order_detail
* 0259835 添加超时自动取消
* b09b598 更正取消订单的测试
* d820546 订单取消和查询相关测试
* 576545e 买家接口补全
* 7f90b98 历史订单查询和订单取消初稿
| * 284ccf8 (lzj_branch) some bug
| * cca8a09 add index
| *   7751827 (origin/lzj_branch) 修改gitnore文件
| |\
| |/
|/|
* | a53baef merge
| * dd49e5e pass receive and delivery test
|/
* 7fbf715 修改文档内的设计结构，成功完成60%的test
* 4ece119 修改report文档
* dd50a11 增加收货与发货的功能，定义status状态，在report.md中
* 38288b4 (origin/buyer, buyer) passed payment test
* a2caed5 passed new order tests
* 17020db simply fixes the picture problem
* 9c4c1fb 完成用户接口
* c1776f7 改变jwt_encode版本不兼容的问题,user.py文件
* 0f618c0 (origin/seller) 修改server.run中的数据库初始化
* a6d4a52 60%work finished temporaly
* 23c0e10 (origin/main) Initial commit
```

相关commits:

Commits on Oct 23, 2024

merge ll committed last week	a53baef	
修改文档内的设计结构，成功完成60%的test ll committed last week	7fbf715	

Commits on Oct 21, 2024

修改report文档 ll committed last week	4ece119	
增加收货与发货的功能，定义status状态，在report.md中 ll committed last week	dd50a11	

Commits on Oct 17, 2024

passed payment test Avalonere authored 2 weeks ago	Verified	38288b4
passed new order tests Avalonere authored 2 weeks ago	Verified	a2caed5
simply fixes the picture problem Avalonere authored 2 weeks ago	Verified	17020db

Commits on Oct 16, 2024

完成用户接口 ll committed 2 weeks ago	9c4c1fb	
改变jwt_encode版本不兼容的问题,user.py文件 ll committed 2 weeks ago	c1776f7	

Commits on Oct 13, 2024

修改server.run中的数据库初始化 ll committed 3 weeks ago	0f618c0	

Commits on Oct 12, 2024

60%work finished temporaly ll committed 3 weeks ago	a6d4a52	

其中小组成员如下

ll:林子骥

Avalonere：施长林

GMMari02024:沈超

小组分工

林子骥：数据库业务逻辑设计，实现基本功能中的用户功能与卖家功能；对于扩展功能，实现大发货收货功能与大部分图书查找功能；编写相应前端代码与测试样例；参与报告的编写。

施长林：实现基本功能中的买家功能；对于扩展功能，实现历史订单查询与订单取消功能，在图书查找中增添书本推荐功能；编写相应的前端代码与测试样例；参与报告的编写。

沈超：对于基本功能中的卖家功能查漏补缺；对于扩展功能，对图书查找功能查漏补缺；编写相应测试样例；参与报告撰写。