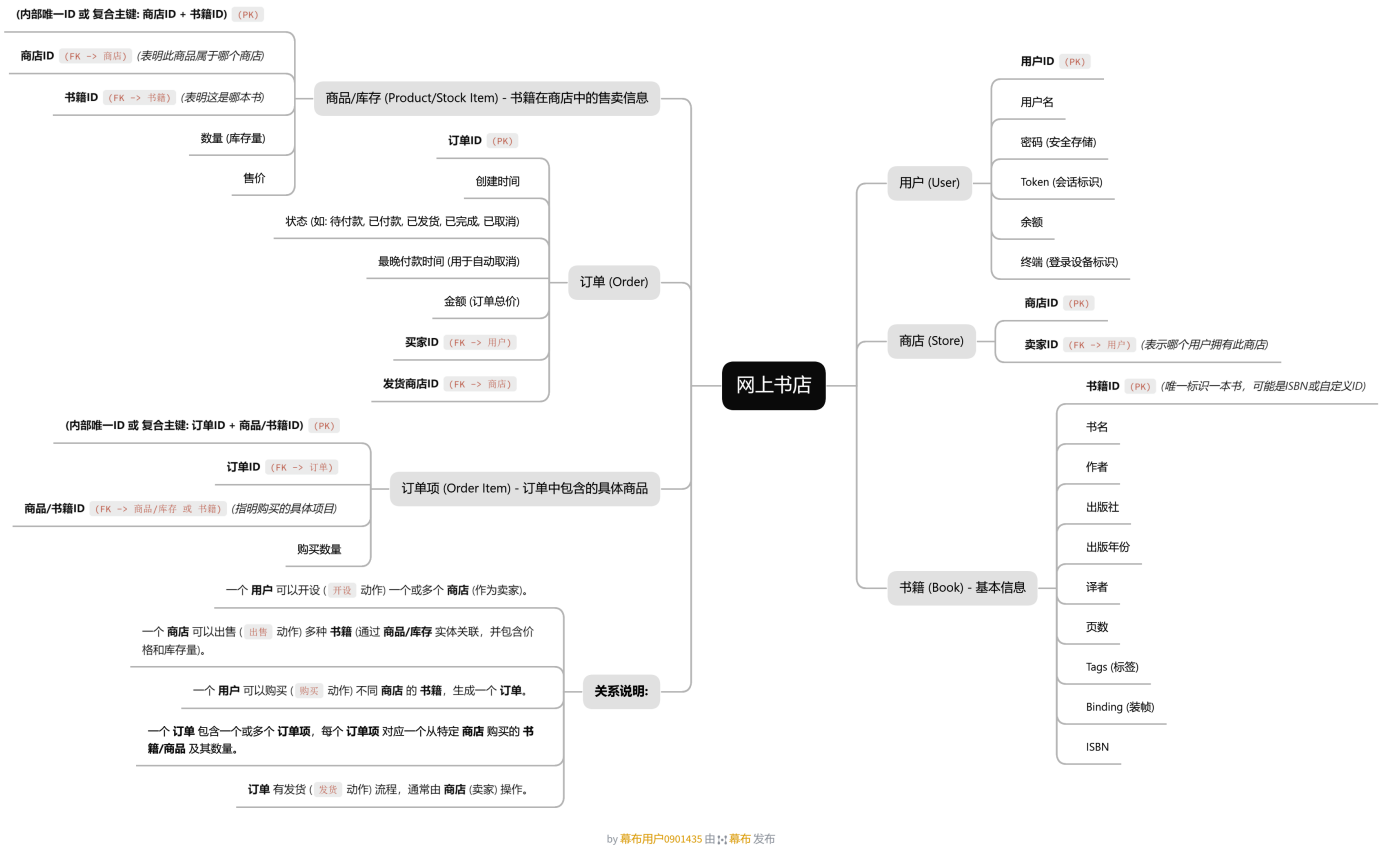
项目总结报告

成员：

吕佳鸿 肖岂源 柳絮源

概述

本项目采用MongoDB作为文档数据库，为在线书店系统提供高效、灵活的数据存储解决方案。MongoDB的文档模型非常适合存储结构复杂、关联灵活的商业数据，非常符合电子商务应用的需求。



数据库连接设计

系统通过 `database.py` 建立与MongoDB的连接：

```python
class MongoDB_client:
    def __init__(self):
        self.socket = pymongo.MongoClient(uri, server_api=pymongo.server_api.ServerApi('1'))
        self.check_and_delete_database('bookstore')
        self.database = self.socket['bookstore']
```

集合(Collection)设计

1. user集合

存储用户信息，包括买家和卖家。

Schema结构:

```
{
  "user_id": String,          // 用户唯一标识
  "password": String,         // 用户密码
  "balance": Number,          // 账户余额
  "token": String,            // JWT授权令牌
  "terminal": String,         // 登录设备标识
  "created_at": Number,       // 账户创建时间戳
  "last_login": Number,       // 最后登录时间戳(可选)
  "logout_time": Number       // 登出时间戳(可选)
}
```

索引:

```
{ "user_id": 1 }  // 升序索引，提高查询效率
```

2. user_store集合

存储商店与用户(卖家)的关联信息。

Schema结构:

```
{
  "store_id": String,        // 商店唯一标识
  "user_id": String,         // 店主用户ID
  "status": String,          // 商店状态(active/inactive)
  "created_at": Number       // 创建时间戳
}
```

索引:

```
{ "user_id": 1, "store_id": 1 }  // 复合索引
```

3. store集合

存储商店中的图书库存信息。

Schema结构:

```
{
  "book_id": String,         // 图书唯一标识
  "store_id": String,        // 所属商店ID
  "book_info": String,       // 图书详细信息(JSON字符串)
  "stock_level": Number,     // 库存数量
  "created_at": Number,      // 创建时间戳
  "updated_at": Number       // 更新时间戳
}
```

索引:

```
{ "book_id": 1, "store_id": 1 }  // 复合索引，提高查询和库存操作效率
```

4. new_order集合

存储新创建的订单信息(临时状态)。

Schema结构:

```
{
  "order_id": String,        // 订单唯一标识
  "user_id": String,         // 购买者用户ID
  "store_id": String         // 商店ID
}
```

5. new_order_detail集合

存储新订单的详细商品信息(临时状态)。

Schema结构:

```
{
  "order_id": String,        // 订单ID
  "book_id": String,         // 图书ID
  "count": Number,           // 购买数量
  "price": Number            // 商品总价
}
```

6. order_history集合

存储订单历史记录。

Schema结构:

```
{
  "order_id": String,        // 订单唯一标识
  "user_id": String,         // 购买者用户ID
  "store_id": String,        // 商店ID
  "status": String,          // 订单状态(pending/paid/shipped/received/cancelled)
  "created_at": Number,      // 创建时间戳
  "paid_at": Number,         // 支付时间戳(可选)
  "shipped_at": Number       // 发货时间戳(可选)
}
```

7. order_history_detail集合

存储订单历史的详细商品信息。

Schema结构:

```
{
  "order_id": String,        // 订单ID
  "book_id": String,         // 图书ID
  "count": Number,           // 购买数量
  "price": Number            // 商品总价
}
```

聚合查询设计

系统使用MongoDB的聚合查询功能获取复杂的数据关联，例如在 `get_order_history` 函数中：

```
pipeline = [
    {"$match": {"user_id": user_id}},
    {"$lookup": {
        "from": "order_history_detail",
        "localField": "order_id",
        "foreignField": "order_id",
        "as": "items"
    }}
]
```

此查询使用 `$lookup` 操作符实现了类似SQL JOIN的功能，将订单与订单详情关联起来。

数据完整性保障

尽管MongoDB是非关系型数据库，本系统依然通过应用程序逻辑实现了数据完整性保障：

1. **事务管理**：如在 `new_order` 和 `payment` 函数中，确保库存更新、订单创建和资金转移的原子性。
2. **库存管理**：通过 `$inc` 和 `$gte` 条件操作符确保库存不会出现负数。
3. **状态跟踪**：通过订单状态字段跟踪订单全生命周期。

扩展功能集合

根据项目代码，系统可能还实现了以下功能相关的集合(从代码中可以推断)：

用户收藏功能

可能存在用于存储用户收藏信息的集合，如：

```
// user_collection集合
{
  "user_id": String,         // 用户ID
  "book_id": String,         // 收藏的图书ID
  "collected_at": Number     // 收藏时间戳
}

// user_store_collection集合
{
  "user_id": String,         // 用户ID
  "store_id": String,        // 收藏的商店ID
  "collected_at": Number     // 收藏时间戳
}
```

性能优化设计

1. **索引策略**：对频繁查询的字段创建了索引，如 `user_id` 、 `book_id` 和 `store_id` 。
2. **文档结构**：采用合理的嵌套结构和引用，避免过度嵌套导致的性能问题。
3. **时间戳记录**：所有关键操作都记录时间戳，便于数据分析和业务追踪。

一. 后端核心逻辑（`/be/model`）- 我们系统的大脑

这部分代码是整个网上书店后台的核心，负责处理所有的业务逻辑和数据操作。

1. `/be/model/user.py`： 用户管理与认证模块

- **主要功能介绍**
  这个模块是管理用户账号的"大本营"，完成了用户从 **注册** 新账号开始的所有事情。注册时系统会确保每个用户名都是独一无二的，并给新用户一个初始为零的余额。为了安全，我们使用了 **JWT (JSON Web Token)** 技术来做用户身份认证。每次用户 **登录** 成功（当然要先对一对 **密码**），系统就会发一个有时效（我们设的是1小时）的 **"通行证"(Token)**，用户之后的请求需要带上这个通行证，这样可以让系统知道谁在操作。
  另外，这个模块也能处理用户 **登出**（让通行证失效）、安全地 **修改密码**（得先验证旧密码对不对），以及 **注销账号**（也得验证密码才行）。特别值得一提的是，我们在原有系统的基础上加入了 **书籍搜索** 功能。用户可以输入 **书名、书里的内容、标签** 等关键词来找书，还能指定是在某个 **店里找** 还是 **全网站找**，找书体验明显提升。
- **核心代码实现**

```python
# JWT Token 生成和验证
def generate_token(user_id: str, terminal: str) -> str:
    payload = {
        "user_id": user_id,
        "terminal": terminal,
        "timestamp": time.time()
    }
    encoded = jwt.encode(payload, key=user_id, algorithm="HS256")
    return encoded.encode("utf-8").decode("utf-8")

def verify_token(token_data: str, user_id: str) -> Dict:
    decoded = jwt.decode(token_data, key=user_id, algorithms="HS256")
    return decoded

# 用户注册
def register(self, user_id: str, password: str) -> Tuple[int, str]:
    try:
        db = self.db
        existing_account = db['user'].find_one({"user_id": user_id})

        if existing_account:
            return error.error_exist_user_id(user_id)

        device_id = f"terminal_{str(time.time())}"
        auth_token = generate_token(user_id, device_id)

        account_data = {
            "user_id": user_id,
            "password": password,
            "balance": 0,
            "token": auth_token,
            "terminal": device_id,
            "created_at": int(time.time())
        }

        db['user'].insert_one(account_data)
        return 200, "ok"

    except pymongo.errors.PyMongoError as e:
        return self._handle_error(528, str(e))
    except Exception as e:
        return self._handle_error(530, f"Unexpected error: {str(e)}")
```

```python
# 用户登录
def login(self, user_id: str, password: str, terminal: str) -> Tuple[int, str, str]:
    try:
        status_code, result_msg = self.check_password(user_id, password)
        if status_code != 200:
            return status_code, result_msg, ""

        auth_token = generate_token(user_id, terminal)

        update_data = {
            'token': auth_token,
            'terminal': terminal,
            'last_login': int(time.time())
        }

        update_result = self.db['user'].update_one(
            {'user_id': user_id},
            {'$set': update_data}
        )

        if not update_result.matched_count:
            return error.error_authorization_fail()

        return 200, "ok", auth_token
    except pymongo.errors.PyMongoError as e:
        return self._handle_error(528, str(e), "")
    except Exception as e:
        return self._handle_error(530, f"Unexpected error: {str(e)}", "")
```

- **设计优点**
  我们使用了 **JWT Token** 和 **密码验证** 的双重保险，使得我们的数据库设计安全性大大提升。
  另外Token还设有 **时间限制**，过期了就需要用户重新登录。对用户每次登录、登出的会话管理也做得比较详细，安全系数高。

2. `/be/model/buyer.py`：买家业务逻辑模块

- **主要功能介绍**
  这个模块管理了买家购买书籍的整个过程。其核心功能在于：
  下单（`new_order`）：买家选好书和数量，点击下单，系统会智能地 **检查库存够不够**，如果有存货就 **减少库存** 并生成订单记录存到数据库里。
  付钱（`payment`）：下单成功之后会进入付钱界面，这时候系统会检查订单状态是否正确、买家 **余额是否充足**、**密码** 输入是否正确，如果没有出错则支付成功，同时卖家的账户余额会增加。

充值（`add_funds`）：买家如果余额不足，可以通过充值功能给自己账户增加余额（需要输密码确认）。

查询历史订单（`get_order_history`）：卖家购买成功后自然会有看购买记录的需求，所以我们提供了查询历史订单的功能。

取消订单（`cancel_order`）：在订单尚未支付的时候，买家随时可以取消订单。考虑到一般人的支付习惯，我们设定如果下单超过一段时间后没有付钱，**订单会自动取消**，这样能避免资源浪费。

确认收货（`receive_order`）: 卖家发货后，买家收到货了可以点击确认收货。

收藏（`collect`）:为了方便买家下次惠顾，我们添加了 **收藏** 功能，可以 **收藏/取消收藏** 喜欢的 **书**（`collect_book`，`uncollect_book`）和 **店铺**（`collect_store`，`uncollect_store`），还能查看自己的 **收藏列表**（`get_collection`，`get_store_collection`）。

- **核心代码实现**

```python
# 创建新订单
def new_order(self, user_id: str, store_id: str, id_and_count: List[Tuple[str, int]]) -> Tuple[
    order_id = ""
    try:
        if not self.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id) + (order_id,)
        if not self.store_id_exist(store_id):
            return error.error_non_exist_store_id(store_id) + (order_id,)

        transaction_id = "{}_{}_{}".format(user_id, store_id, str(uuid.uuid1()))
        order_items = []

        for book_id, quantity in id_and_count:
            inventory_item = self.db["store"].find_one({"store_id": store_id, "book_id": book_i
            if not inventory_item:
                return error.error_non_exist_book_id(book_id) + (order_id,)

            available_stock = inventory_item["stock_level"]
            if available_stock < quantity:
                return error.error_stock_level_low(book_id) + (order_id,)

            query_conditions = {
                "book_id": book_id,
                "store_id": store_id,
                "stock_level": {"$gte": quantity}
            }
            stock_update = {"$inc": {"stock_level": -quantity}}

            update_result = self.db["store"].update_one(query_conditions, stock_update)
            if update_result.modified_count == 0:
                return error.error_stock_level_low(book_id) + (order_id,)

            book_info_json = json.loads(inventory_item["book_info"])
            item_price = book_info_json.get("price") * quantity

            order_item = {
                "order_id": transaction_id,
                "book_id": book_id,
                "count": quantity,
                "price": item_price
            }
            order_items.append(order_item)
```

```
        if order_items:
            self.db["new_order_detail"].insert_many(order_items)

        order_record = {
            "order_id": transaction_id,
            "user_id": user_id,
            "store_id": store_id
        }
        self.db["new_order"].insert_one(order_record)

        order_id = transaction_id
        self.timer = threading.Timer(10.0, self.cancel_order, args=[user_id, order_id])
        self.timer.start()

        history_record = order_record.copy()
        history_record["status"] = "pending"
        history_record["created_at"] = int(time.time())

        self.db["order_history"].insert_one(history_record)
        self.db["order_history_detail"].insert_many(order_items)

    except pymongo.errors.PyMongoError as e:
        logging.error("DB Error: {}".format(str(e)))
        return 528, str(e), ""
    except Exception as e:
        logging.info("System Error: {}".format(str(e)))
        return 530, str(e), ""

    return 200, "ok", order_id

# 支付订单
def payment(self, user_id: str, password: str, order_id: str) -> Tuple[int, str]:
    try:
        db = self.db
        order_record = db["new_order"].find_one({"order_id": order_id})
        if not order_record:
            return error.error_invalid_order_id(order_id)

        if order_record["user_id"] != user_id:
            return error.error_authorization_fail()

        buyer_record = db["user"].find_one({"user_id": user_id})
        if not buyer_record:
```

```
        return error.error_non_exist_user_id(user_id)

    if password != buyer_record["password"]:
        return error.error_authorization_fail()

    order_status = db["order_history"].find_one({"order_id": order_id})["status"]
    if order_status != "pending":
        error.error_invalid_order_status(order_id)

    if self.timer:
        self.timer.cancel()

    order_details = db["new_order_detail"].find({"order_id": order_id})
    total_amount = sum(item["price"] for item in order_details)

    if buyer_record["balance"] < total_amount:
        return error.error_not_sufficient_funds(order_id)

    update_result = db["user"].update_one(
        {"user_id": user_id, "balance": {"$gte": total_amount}},
        {"$inc": {"balance": -total_amount}}
    )

    if update_result.modified_count == 0:
        return error.error_not_sufficient_funds(order_id)

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

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

    seller_update = db["user"].update_one(
        {"user_id": seller_id},
        {"$inc": {"balance": total_amount}}
    )

    if seller_update.modified_count == 0:
        return error.error_non_exist_user_id(seller_id)

    db["new_order"].delete_one({"order_id": order_id})
    db["new_order_detail"].delete_many({"order_id": order_id})
```

```
            history_update = {
                "$set": {
                    "status": "paid",
                    "paid_at": int(time.time())
                }
            }
            db["order_history"].update_one({"order_id": order_id}, history_update)

        except pymongo.errors.PyMongoError as e:
            return 528, str(e)
        except Exception as e:
            return 530, str(e)

        return 200, "ok"
```

- **设计优点**
  这部分设计我们特别注意了 **数据的准确性和一致性**。譬如下单、付款等操作，我们确保了相关步骤（比如减库存和生成订单）状态同步（同时成功或者同时失败），不会出现数据不一致的情况。
  订单超时自动取消 的功能利用了 `threading.Timer` 实现，巧妙而富有现实意义。
  在安全性上，对于付钱、充值等敏感操作，都要求 **验证用户身份和密码**，保证了交易安全。

3. `/be/model/seller.py`：卖家业务逻辑模块

- **主要功能介绍**
  这个模块用于管理卖家的业务，方便卖家管理自己的店铺，其核心功能在于：
  新开店铺（`create_store`）：系统会保证店名（ID）不跟别人重复。
  上架新书（`add_book`）： 店铺开始营业之后，就能往店里上架新书了，卖家填上书的各种信息（书名、作者、简介、价格等等，我们用JSON格式传输）和初始 **库存**。
  增加库存（`add_stock_level`）：如果书籍畅销，库存没有剩余了，可以用 增加库存功能随时补充。
  发货（`ship_order`）：买家付钱后，卖家就需要发货，系统会判定这笔订单发货成功，订单状态会变成"已发货"。
  查看自己店铺的所有订单（`view_orders`）：同卖家查看自己的历史订单一样，卖家也能随时查看自己店铺的所有订单，方便管理。
- **核心代码实现**

```python
# 创建新店铺
def create_store(self, user_id: str, store_id: str) -> Tuple[int, str]:
    try:
        if not self.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        if self.store_id_exist(store_id):
            return error.error_exist_store_id(store_id)

        store_data = {
            "store_id": store_id,
            "user_id": user_id,
            "status": "active",
            "created_at": int(time.time())
        }

        self.db['user_store'].insert_one(store_data)
        return 200, "ok"

    except pymongo.errors.PyMongoError as e:
        return self._handle_error(528, str(e))
    except Exception as e:
        return self._handle_error(530, f"Unexpected error: {str(e)}")

# 上架新书
def add_book(self, user_id: str, store_id: str, book_id: str, book_json_str: str, stock_level: i
    try:
        if not self.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        if not self.store_id_exist(store_id):
            return error.error_non_exist_store_id(store_id)
        if self.book_id_exist(store_id, book_id):
            return error.error_exist_book_id(book_id)

        book_data = {
            "book_id": book_id,
            "store_id": store_id,
            "book_info": book_json_str,
            "stock_level": stock_level,
            "created_at": int(time.time()),
            "updated_at": int(time.time())
        }

        self.db['store'].insert_one(book_data)
```

```
        return 200, "ok"

    except pymongo.errors.PyMongoError as e:
        return self._handle_error(528, str(e))
    except Exception as e:
        return self._handle_error(530, f"Unexpected error: {str(e)}")

# 增加库存
def add_stock_level(self, user_id: str, store_id: str, book_id: str, add_stock_level: int) -> Tu
    try:
        if not self.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        if not self.store_id_exist(store_id):
            return error.error_non_exist_store_id(store_id)
        if not self.book_id_exist(store_id, book_id):
            return error.error_non_exist_book_id(book_id)

        result = self.db['store'].update_one(
            {'store_id': store_id, 'book_id': book_id},
            {
                '$inc': {'stock_level': add_stock_level},
                '$set': {'updated_at': int(time.time())}
            }
        )

        if result.matched_count == 0:
            return error.error_non_exist_book_id(book_id)

        return 200, "ok"

    except pymongo.errors.PyMongoError as e:
        return self._handle_error(528, str(e))
    except Exception as e:
        return self._handle_error(530, f"Unexpected error: {str(e)}")

# 发货
def ship_order(self, user_id: str, store_id: str, order_id: str) -> Tuple[int, str]:
    try:
        if not self.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        if not self.store_id_exist(store_id):
            return error.error_non_exist_store_id(store_id)
```

```
            order = self.db['order_history'].find_one({'order_id': order_id})
            if not order:
                return error.error_invalid_order_id(order_id)
            if order['status'] != 'paid':
                return error.error_invalid_order_status(order_id)

            result = self.db['order_history'].update_one(
                {'order_id': order_id},
                {
                    '$set': {
                        'status': 'shipped',
                        'shipped_at': int(time.time())
                    }
                }
            )

            if result.matched_count == 0:
                return error.error_invalid_order_id(order_id)

            return 200, "ok"

        except pymongo.errors.PyMongoError as e:
            return self._handle_error(528, str(e))
        except Exception as e:
            return self._handle_error(530, f"Unexpected error: {str(e)}")
```

- **设计优点**
  这个模块的设计优点在于 **数据库操作高效**，并且 **逻辑严谨**。
  在增加库存功能中，我们用了MongoDB的一个特殊操作（`$inc`），能保证快速又准确地更新数字，能够支持多人同时操作。
  安全性和一致性同样是我们始终注重的问题，在做每一步重要操作前，比如开店、加书、发货，我们都会提前 **检查条件**（例如用户信息是否匹配、店铺是否营业、库存是否有存货、订单是否已付钱），确保操作合理，不会数据紊乱。

二. 后端接口层（`/be/view`）- 系统对外的窗口

这部分代码作为后端的 API 接口，是后端的"窗口"，将后端的核心功能和逻辑，包装到了外部（例如网页或者App）可以模块化调用的标准服务。

1. `/be/view/seller.py`： 卖家接口视图

- **主要功能介绍**
  这个文件是卖家界面的API接口：
  `/create_store` 接口是用来 **新店开业** 的，卖家发送他的用户ID和设定的店铺ID即可。
  `/add_book` 是用来 **上架新书** 的，卖家需要提供用户ID、店铺ID、书的详细信息（JSON格式）和库存。
  `/add_stock_level` 用来 **增加库存**。
  `/ship_order` 是卖家用来 **标记订单已发货**的，但系统会先检查这单是否真的已经付款。这些接口都设计成接收 POST 请求和 JSON 数据，返回结果也是统一的 JSON 格式。
- **核心代码实现**

```python
class Seller:
    def __init__(self, url_prefix: str, token: str):
        self.url_prefix = url_prefix
        self.token = token
        self.headers = {"token": token}

    def create_store(self, user_id: str, store_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "store_id": store_id
        }
        response = requests.post(f"{self.url_prefix}/create_store", json=json_data, headers=self.
        return response.status_code, response.json().get("message", "")

    def add_book(self, user_id: str, store_id: str, book_id: str, book_json_str: str, stock_leve
        json_data = {
            "user_id": user_id,
            "store_id": store_id,
            "book_id": book_id,
            "book_json_str": book_json_str,
            "stock_level": stock_level
        }
        response = requests.post(f"{self.url_prefix}/add_book", json=json_data, headers=self.hea
        return response.status_code, response.json().get("message", "")

    def add_stock_level(self, user_id: str, store_id: str, book_id: str, add_stock_level: int) -
        json_data = {
            "user_id": user_id,
            "store_id": store_id,
            "book_id": book_id,
            "add_stock_level": add_stock_level
        }
        response = requests.post(f"{self.url_prefix}/add_stock_level", json=json_data, headers=s
        return response.status_code, response.json().get("message", "")

    def ship_order(self, user_id: str, store_id: str, order_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "store_id": store_id,
            "order_id": order_id
        }
        response = requests.post(f"{self.url_prefix}/ship_order", json=json_data, headers=self.h
        return response.status_code, response.json().get("message", "")
```

```
def view_orders(self, user_id: str, store_id: str) -> Tuple[int, List[Dict]]:
    params = {
        "user_id": user_id,
        "store_id": store_id
    }
    response = requests.get(f"{self.url_prefix}/view_orders", params=params, headers=self.he
    if response.status_code == 200:
        return response.status_code, response.json().get("orders", [])
    return response.status_code, []
```

- **设计优点**
  整个模块的设计符合使用习惯， **风格统一**，使用方便。我们统一使用 POST 请求，传 JSON，返回 JSON，大大降低了和前端对接的难度。
  另外 **出错处理** 也合理完善，如果参数确实或者服务器内部出错，会返回明确的提示和错误代码（比如400, 500），方便开发者查明问题。**请求的参数会提前检查**，保证传输的数据准确无误。

2. `/be/view/buyer.py`： 买家接口视图

- **主要功能介绍**
  这里是买家能用到的所有API接口：
  `/new_order` 让买家能 **下订单**，告诉我们要买哪个店的哪些书（书ID和数量），成功了会返回一个订单号。
  `/payment` 就是 **付钱** 的接口，需要订单号和买家密码。
  `/add_funds` 是 **充值** 接口。
  `/get_order_history` 用来 **看买过的订单**。
  `/cancel_order` 可以 **取消还没付款的订单**。
  `/receive_order` 是 **确认收货**的接口。
  `collect` 是一整套 **管理收藏** 的接口：收藏/取消收藏书 (`/collect_book` , `/uncollect_book`)，看收藏的书 (`/get_collection`)；收藏/取消收藏店 (`/collect_store` , `/uncollect_store`)，看收藏的店 (`/get_store_collection`)。
- **核心代码实现**

```python
class Buyer:
    def __init__(self, url_prefix: str, token: str):
        self.url_prefix = url_prefix
        self.token = token
        self.headers = {"token": token}

    def new_order(self, user_id: str, store_id: str, books: List[Tuple[str, int]]) -> Tuple[int,
        json_data = {
            "user_id": user_id,
            "store_id": store_id,
            "books": books
        }
        response = requests.post(f"{self.url_prefix}/new_order", json=json_data, headers=self.he
        if response.status_code == 200:
            return response.status_code, response.json().get("order_id", "")
        return response.status_code, ""

    def payment(self, user_id: str, password: str, order_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "password": password,
            "order_id": order_id
        }
        response = requests.post(f"{self.url_prefix}/payment", json=json_data, headers=self.hea
        return response.status_code, response.json().get("message", "")

    def add_funds(self, user_id: str, password: str, add_value: int) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "password": password,
            "add_value": add_value
        }
        response = requests.post(f"{self.url_prefix}/add_funds", json=json_data, headers=self.he
        return response.status_code, response.json().get("message", "")

    def get_order_history(self, user_id: str) -> Tuple[int, List[Dict]]:
        params = {"user_id": user_id}
        response = requests.get(f"{self.url_prefix}/get_order_history", params=params, headers=s
        if response.status_code == 200:
            return response.status_code, response.json().get("orders", [])
        return response.status_code, []

    def cancel_order(self, user_id: str, order_id: str) -> Tuple[int, str]:
```

```
        json_data = {
            "user_id": user_id,
            "order_id": order_id
        }
        response = requests.post(f"{self.url_prefix}/cancel_order", json=json_data, headers=self
        return response.status_code, response.json().get("message", "")

    def receive_order(self, user_id: str, order_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "order_id": order_id
        }
        response = requests.post(f"{self.url_prefix}/receive_order", json=json_data, headers=sel
        return response.status_code, response.json().get("message", "")

    def collect_book(self, user_id: str, book_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "book_id": book_id
        }
        response = requests.post(f"{self.url_prefix}/collect_book", json=json_data, headers=self
        return response.status_code, response.json().get("message", "")

    def uncollect_book(self, user_id: str, book_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "book_id": book_id
        }
        response = requests.post(f"{self.url_prefix}/uncollect_book", json=json_data, headers=se
        return response.status_code, response.json().get("message", "")

    def get_collection(self, user_id: str) -> Tuple[int, List[Dict]]:
        params = {"user_id": user_id}
        response = requests.get(f"{self.url_prefix}/get_collection", params=params, headers=self
        if response.status_code == 200:
            return response.status_code, response.json().get("books", [])
        return response.status_code, []

    def collect_store(self, user_id: str, store_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "store_id": store_id
        }
```

```
        response = requests.post(f"{self.url_prefix}/collect_store", json=json_data, headers=se
        return response.status_code, response.json().get("message", "")

    def uncollect_store(self, user_id: str, store_id: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "store_id": store_id
        }
        response = requests.post(f"{self.url_prefix}/uncollect_store", json=json_data, headers=
        return response.status_code, response.json().get("message", "")

    def get_store_collection(self, user_id: str) -> Tuple[int, List[Dict]]:
        params = {"user_id": user_id}
        response = requests.get(f"{self.url_prefix}/get_store_collection", params=params, heade
        if response.status_code == 200:
            return response.status_code, response.json().get("stores", [])
        return response.status_code, []
```

- **设计优点**
  同卖家接口一样，买家接口也保持了 **返回格式的统一**，让前端处理起来方便快捷。
  错误处理周全，常常遇见的问题大都考虑在内。
  另外，我们把复杂的买家逻辑放到了 `model` 层的 `Buyer` 类里，这样这里的 **接口代码就比较干净**，主要负责接收请求和返回结果，**维护起来更容易**。

3. `/be/view/auth.py`：认证与公共接口视图

- **主要功能介绍**
  这个文件管着用户身份认证和一些任何身份都能使用的接口：
  `/login` 处理 **登录**，对了就发个 Token。
  `/logout` 用来 **登出**，让 Token 失效。
  `/register` 可以 **注册** 新用户。
  `unregister` 用于**注销账号**。
  `/password` 是 **改密码**。
  `/search_book` ，**搜书** 接口，可以根据 **书名、内容、标签、店铺** 等条件来搜，灵活便捷。
- **核心代码实现**

```python
class Auth:
    def __init__(self, url_prefix: str):
        self.url_prefix = url_prefix

    def login(self, user_id: str, password: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "password": password
        }
        response = requests.post(f"{self.url_prefix}/login", json=json_data)
        if response.status_code == 200:
            return response.status_code, response.json().get("token", "")
        return response.status_code, ""

    def logout(self, user_id: str, token: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id
        }
        headers = {"token": token}
        response = requests.post(f"{self.url_prefix}/logout", json=json_data, headers=headers)
        return response.status_code, response.json().get("message", "")

    def register(self, user_id: str, password: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "password": password
        }
        response = requests.post(f"{self.url_prefix}/register", json=json_data)
        return response.status_code, response.json().get("message", "")

    def unregister(self, user_id: str, password: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "password": password
        }
        response = requests.post(f"{self.url_prefix}/unregister", json=json_data)
        return response.status_code, response.json().get("message", "")

    def password(self, user_id: str, old_password: str, new_password: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "oldPassword": old_password,
            "newPassword": new_password
```

```
        }
        response = requests.post(f"{self.url_prefix}/password", json=json_data)
        return response.status_code, response.json().get("message", "")

    def search_book(self, query: str, store_id: str = None, page: int = 1, per_page: int = 10) -
        params = {
            "query": query,
            "page": page,
            "per_page": per_page
        }
        if store_id:
            params["store_id"] = store_id

        response = requests.get(f"{self.url_prefix}/search_book", params=params)
        if response.status_code == 200:
            return response.status_code, response.json().get("books", [])
        return response.status_code, []
```

- **设计优点**
  这部分的接口也设计 **规范**。
  返回格式统一，**错误处理和日志记录** 全部考虑在内，方便开发者调试。
  使用方便直接，我们采用 POST 请求，只需要传输对应的参数即可。
  参数检查严格，保证传输的数据是有效的。

三. 前端访问与测试层（`/fe/access`）- 模拟用户操作

这部分代码主要是用来 **模拟前端或者测试** 怎么去调用咱们前面写的后端接口的，帮助我们验证功能对不对。

1. `/fe/access/auth.py`：认证接口访问客户端

- **主要功能介绍**
  我们实现了 `Auth` 类，用来处理后端的认证接口。用这个类我们可以很方便地模拟 **用户登录（`login`）**、**注册（`register`）**、**改密码（`password`）**、**登出（`logout`）** 和 **注销（`unregister`）**。
  另外系统还包含了 **搜书（`search_book`）** 的功能，可以指定不同的条件去搜索。类里面的方法本质上能够帮我们发送 HTTP 请求给后端，并告诉我们后端返回的内容（例如状态码，登录成功会返回 Token）。
- **核心代码实现**

```python
class Auth:
    def __init__(self, url_prefix: str):
        self.url_prefix = url_prefix

    def login(self, user_id: str, password: str) -> Tuple[int, str]:
        json_data = {
            "user_id": user_id,
            "password": password
        }
        response = requests.post(f"{self.url_prefix}/login", json=json_data)
        if response.status_code == 200:
            return response.status_code, response.json().get("token", "")
        return response.status_code, ""

    def register(self, user_id: str, password: str) -> int:
        json_data = {
            "user_id": user_id,
            "password": password
        }
        response = requests.post(f"{self.url_prefix}/register", json=json_data)
        return response.status_code

    def password(self, user_id: str, old_password: str, new_password: str) -> int:
        json_data = {
            "user_id": user_id,
            "oldPassword": old_password,
            "newPassword": new_password
        }
        response = requests.post(f"{self.url_prefix}/password", json=json_data)
        return response.status_code

    def logout(self, user_id: str, token: str) -> int:
        json_data = {
            "user_id": user_id
        }
        headers = {"token": token}
        response = requests.post(f"{self.url_prefix}/logout", json=json_data, headers=headers)
        return response.status_code

    def unregister(self, user_id: str, password: str) -> int:
        json_data = {
            "user_id": user_id,
            "password": password
```

```
        }
        response = requests.post(f"{self.url_prefix}/unregister", json=json_data)
        return response.status_code

    def search_book(self, query: str, store_id: str = None, page: int = 1, per_page: int = 10) -
        params = {
            "query": query,
            "page": page,
            "per_page": per_page
        }
        if store_id:
            params["store_id"] = store_id

        response = requests.get(f"{self.url_prefix}/search_book", params=params)
        if response.status_code == 200:
            return response.status_code, response.json().get("books", [])
        return response.status_code, []
```

- **设计优点**
  这个 `Auth` 类 **包含了所有认证相关的操作**，使用简单，代码也显得整洁。它统一使用 `requests` 库发 POST 请求，地址自动拼接，**修改方便**。
  每个操作结束会显示结果（HTTP状态码），这样调用者就知道下一步的操作了。搜书功能的设计也**很灵活**。

2. `/fe/access/buyer.py`：买家接口访问客户端

- **主要功能介绍**
  `Buyer` 类是模拟买家操作的客户端。这个类一经创建就会**自动登录**（`Auth` 类介入），获取用户ID和Token 存起来。然后你就可以用 `Buyer` 对象来实现买家部分的相关功能。这些操作发请求时都会自动捎带登录获取的 Token，后端就能通过 token 获取用户信息。
- **核心代码实现**

```python
class Buyer:
    def __init__(self, url_prefix: str, user_id: str, password: str):
        self.url_prefix = url_prefix
        self.user_id = user_id
        self.password = password
        self.token = ""
        self.auth = Auth(url_prefix)
        self.login()

    def login(self) -> int:
        code, token = self.auth.login(self.user_id, self.password)
        if code == 200:
            self.token = token
        return code

    def new_order(self, store_id: str, books: List[Tuple[str, int]]) -> Tuple[int, str]:
        json_data = {
            "user_id": self.user_id,
            "store_id": store_id,
            "books": books
        }
        headers = {"token": self.token}
        response = requests.post(f"{self.url_prefix}/new_order", json=json_data, headers=headers
        if response.status_code == 200:
            return response.status_code, response.json().get("order_id", "")
        return response.status_code, ""

    def payment(self, order_id: str) -> int:
        json_data = {
            "user_id": self.user_id,
            "password": self.password,
            "order_id": order_id
        }
        headers = {"token": self.token}
        response = requests.post(f"{self.url_prefix}/payment", json=json_data, headers=headers)
        return response.status_code

    def add_funds(self, add_value: int) -> int:
        json_data = {
            "user_id": self.user_id,
            "password": self.password,
            "add_value": add_value
        }
```

```
    headers = {"token": self.token}
    response = requests.post(f"{self.url_prefix}/add_funds", json=json_data, headers=headers
    return response.status_code

def get_order_history(self) -> Tuple[int, List[Dict]]:
    headers = {"token": self.token}
    response = requests.get(f"{self.url_prefix}/get_order_history", headers=headers)
    if response.status_code == 200:
        return response.status_code, response.json().get("orders", [])
    return response.status_code, []

def cancel_order(self, order_id: str) -> int:
    json_data = {
        "user_id": self.user_id,
        "order_id": order_id
    }
    headers = {"token": self.token}
    response = requests.post(f"{self.url_prefix}/cancel_order", json=json_data, headers=head
    return response.status_code

def receive_order(self, order_id: str) -> int:
    json_data = {
        "user_id": self.user_id,
        "order_id": order_id
    }
    headers = {"token": self.token}
    response = requests.post(f"{self.url_prefix}/receive_order", json=json_data, headers=hea
    return response.status_code

def collect_book(self, book_id: str) -> int:
    json_data = {
        "user_id": self.user_id,
        "book_id": book_id
    }
    headers = {"token": self.token}
    response = requests.post(f"{self.url_prefix}/collect_book", json=json_data, headers=head
    return response.status_code

def uncollect_book(self, book_id: str) -> int:
    json_data = {
        "user_id": self.user_id,
        "book_id": book_id
    }
```

```
        headers = {"token": self.token}
        response = requests.post(f"{self.url_prefix}/uncollect_book", json=json_data, headers=he
        return response.status_code

    def get_collection(self) -> Tuple[int, List[Dict]]:
        headers = {"token": self.token}
        response = requests.get(f"{self.url_prefix}/get_collection", headers=headers)
        if response.status_code == 200:
            return response.status_code, response.json().get("books", [])
        return response.status_code, []

    def collect_store(self, store_id: str) -> int:
        json_data = {
            "user_id": self.user_id,
            "store_id": store_id
        }
        headers = {"token": self.token}
        response = requests.post(f"{self.url_prefix}/collect_store", json=json_data, headers=hea
        return response.status_code

    def uncollect_store(self, store_id: str) -> int:
        json_data = {
            "user_id": self.user_id,
            "store_id": store_id
        }
        headers = {"token": self.token}
        response = requests.post(f"{self.url_prefix}/uncollect_store", json=json_data, headers=h
        return response.status_code

    def get_store_collection(self) -> Tuple[int, List[Dict]]:
        headers = {"token": self.token}
        response = requests.get(f"{self.url_prefix}/get_store_collection", headers=headers)
        if response.status_code == 200:
            return response.status_code, response.json().get("stores", [])
        return response.status_code, []
```

- **设计优点**
  这个类 **封装良好**，把买家操作都集中到一起。
  它能够 **自动处理了登录和 Token**，在使用的时候不用担心这些认证的细节了。整体代码风格和 `Auth` 类相似，**统一规范**。返回的状态码能够让用户 **了解操作是否成功**。在这样的设计框架下想要集成新的买家功能也非常方便。

3. `/fe/access/book.py`： 图书数据访问与处理

- **主要功能介绍**
  这里有两个类：
  `Book` 类是书籍的 **名片**，专门用来 **存放一本书的详细信息**，例如书名、作者、出版社、价格、ISBN，还有简介、标签，甚至 **书的封面图片**（我们将其转化为了 Base64 编码的文本）。
  `BookDB` 类则是用于与 **MongoDB 数据库交互** 的，用于查询书籍。这个类能显示数据库里 **书记总量**（`get_book_count`），并且能 **按页获取书的信息**（`get_book_info`），例如"给我第11到20本书的信息"，系统会把查到的信息包装成若干 `Book` 对象返还给用户。
- **核心代码实现**

```python
class Book:
    def __init__(self, book_info: Dict):
        self.id = book_info.get("id", "")
        self.title = book_info.get("title", "")
        self.author = book_info.get("author", "")
        self.publisher = book_info.get("publisher", "")
        self.original_title = book_info.get("original_title", "")
        self.translator = book_info.get("translator", "")
        self.pub_year = book_info.get("pub_year", "")
        self.pages = book_info.get("pages", 0)
        self.price = book_info.get("price", 0)
        self.currency_unit = book_info.get("currency_unit", "")
        self.binding = book_info.get("binding", "")
        self.isbn = book_info.get("isbn", "")
        self.author_intro = book_info.get("author_intro", "")
        self.book_intro = book_info.get("book_intro", "")
        self.content = book_info.get("content", "")
        self.tags = book_info.get("tags", [])
        self.picture = book_info.get("picture", "")

class BookDB:
    def __init__(self, db_url: str):
        self.client = pymongo.MongoClient(db_url)
        self.db = self.client["bookstore"]
        self.collection = self.db["books"]
        # 创建索引以提升查询性能
        self.collection.create_index("id")

    def get_book_count(self) -> int:
        return self.collection.count_documents({})

    def get_book_info(self, start: int, size: int) -> List[Book]:
        books = []
        cursor = self.collection.find().skip(start).limit(size)
        for book_info in cursor:
            books.append(Book(book_info))
        return books

    def get_book_by_id(self, book_id: str) -> Optional[Book]:
        book_info = self.collection.find_one({"id": book_id})
        if book_info:
            return Book(book_info)
        return None
```

```
    def search_books(self, query: str, store_id: str = None, page: int = 1, per_page: int = 10)
        search_filter = {
            "$or": [
                {"title": {"$regex": query, "$options": "i"}},
                {"author": {"$regex": query, "$options": "i"}},
                {"book_intro": {"$regex": query, "$options": "i"}},
                {"content": {"$regex": query, "$options": "i"}},
                {"tags": {"$regex": query, "$options": "i"}}
            ]
        }

        if store_id:
            search_filter["store_id"] = store_id

        total = self.collection.count_documents(search_filter)
        books = []

        cursor = self.collection.find(search_filter).skip((page - 1) * per_page).limit(per_page)
        for book_info in cursor:
            books.append(Book(book_info))

        return total, books
```

- **设计优点**
  分工明确： `Book` 类管理数据的特征， `BookDB` 类管理如何从数据库里获取数据。
  `BookDB` 用了 pymongo 和 MongoDB 交互，在 **分页查询**（`skip` 和 `limit`）处理逻辑优秀，在书籍总量较高的情况下，也能够高效地一页一页显示。
  另外，把 **封面图片转成 Base64** 放在 `Book` 对象里，传送给前端或者显示都非常便捷。我们还以书的 ID 添加了数据库 **索引**，这能 **大大提升查找速度**。

4. `/fe/access/new_seller.py` 和 `/fe/access/new_buyer.py`：新用户（卖家/买家）注册与初始化

- **主要功能介绍**
  这两个脚本扮演着 **新用户快速入口** 的角色，分别用于 **注册新卖家** 和 **新买家**。它们的核心功能是提供一个便捷的函数（`register_new_seller` 和 `register_new_buyer`），接收用户期望的 `user_id` 和 `password`。函数内部首先会利用共享的 `Auth` 模块（连接到由 `conf.URL` 指定的后端认证服务）来完成用户的实际注册过程。一旦认证服务确认注册成功（通过检查返回的状态码是否为 200），脚本会紧接着 **创建并返回** 一个对应类型的用户实例（`Seller` 或 `Buyer`）。这个返回的对象不仅代表了刚刚注册成功的用户，而且已经包含了必要的认证信息（`user_id`，`password`，和服务 `URL`），

可以 **立即用于** 后续的操作，比如卖家上架商品或买家浏览下单。可以认为，这两个脚本封装了“注册”和“获取可用用户对象”这两个步骤，提供了一个 **一站式** 的新用户创建体验。

- **核心代码实现**

```
# /fe/access/new_seller.py
from fe import conf
from fe.access import seller, auth

def register_new_seller(user_id, password) -> seller.Seller:
    a = auth.Auth(conf.URL)
    code = a.register(user_id, password)
    assert code == 200 # 确保注册成功
    s = seller.Seller(conf.URL, user_id, password)
    return s

# /fe/access/new_buyer.py
from fe import conf
from fe.access import buyer, auth

def register_new_buyer(user_id, password) -> buyer.Buyer:
    a = auth.Auth(conf.URL)
    code = a.register(user_id, password)
    assert code == 200 # 确保注册成功
    s = buyer.Buyer(conf.URL, user_id, password)
    return s
```

- **设计优点**
  这两个脚本的设计 **极大简化了** 新用户的创建流程，将注册和后续用户对象的初始化 **打包成单一函数调用**。它们巧妙地 **隐藏了底层认证交互** 的细节，使得上层调用（尤其是在测试或初始化场景中）更为 **简洁直观**。同时，使用 `assert` 来校验注册结果，确保了只有在用户成功创建后才会返回有效的用户对象，这增强了使用的 **健壮性**，避免了后续操作中因用户不存在或未成功注册而引发的问题。

测试

除了基本的测试之外，我们还自行增加了一些测试来对新开发的功能进行测试，以提高代码覆盖率率。

测试取消订单（TestCancelOrder）

1. test_ok (测试正常取消订单)：

- 操作步骤：取消第二个订单（未支付的订单），期望返回状态码200。
- 验证逻辑：确保订单取消功能在正常条件下能够正确执行。

2. test_wrong_user_id (测试错误用户ID)：
   - 操作步骤：错误地修改买家的用户ID后尝试取消订单，期望返回状态码非200。
   - 验证逻辑：由于验证系统能否正确识别并阻止因用户ID错误而造成的非法操作。
3. test_non_exist_order_id (测试不存在的订单ID)：
   - 操作步骤：使用一个不存在的订单ID尝试取消订单，期望返回状态码非200。
   - 验证逻辑：确保系统能正确处理不存在的订单ID，避免非法取消。
4. test_repeat_cancel (测试重复取消订单)：
   - 操作步骤：先取消一个订单，然后尝试重复取消同一订单，期望第二次取消的返回状态码非200。
   - 验证逻辑：检查订单的状态管理是否能有效阻止对已取消订单的重复操作。
5. test_cancel_paid_order (测试取消已支付订单)：
   - 操作步骤：尝试取消已经支付的订单，期望返回状态码非200。
   - 验证逻辑：验证订单状态判断逻辑，确保已支付的订单无法取消。
6. test_cancel_long_time_order (测试取消超时的订单)：
   - 操作步骤：等待一定时间后尝试取消订单，模拟订单过期的取消尝试，期望返回状态码非200。
   - 验证逻辑：测试系统是否能处理订单时间的限制，确保长时间后无法取消订单。

测试订单历史查询（TestGetOrderHistory）

1. test_ok：
   - 验证正常情况下订单历史查询的功能。
   - 操作步骤：
     a. 调用get_order_history方法查询订单历史。
     b. 使用断言（assert）检查返回状态码是200，确保查询成功。
2. test_non_exist_user_id：
   - 验证错误用户ID的订单历史查询的反馈。
   - 操作步骤：
     a. 人为修改买家ID以制造"不存在的用户"情形。
     b. 查询该用户的订单历史。
     c. 使用断言（assert）确认返回状态码不是200，确保系统正确处理不存在的用户查询请求。

测试搜索功能（TestSearch）

1. test_search_global：

- 功能验证：在整个数据库中搜索书籍，不限于任何店铺。
- 检查点：确保输入书籍的标题、内容或标签都能正确返回状态码200，表示搜索成功。

2. test_search_global_not_exists：
   - 功能验证：尝试搜索数据库中不存在的书籍信息。
   - 检查点：对于每一种搜索（标题、内容、标签），都应返回529状态码，表示没有找到相关书籍。
3. test_search_in_store：
   - 功能验证：在指定的店铺内搜索书籍。
   - 检查点：分别使用标题、内容和标签进行搜索，都应返回状态码200，表明在指定店铺内成功找到了相关书籍。
4. test_search_not_exist_store_id：
   - 功能验证：使用不存在的店铺ID进行搜索尝试。
   - 检查点：任何形式的搜索都应返回513状态码，指出店铺不存在。
5. test_search_in_store_not_exist：
   - 功能验证：在指定的店铺内搜索不存在的书籍。
   - 检查点：对于不存在的书籍，在指定的店铺内进行搜索，应返回529状态码，表示书籍未找到。

测试订单发货和接收（TestShipReceive）

1. test_ship_ok：
   - 验证卖家可以成功发货，系统返回状态码200。
2. test_receive_ok：
   - 先确保卖家成功发货，然后验证买家接收订单后，系统应返回状态码200。
3. test_error_store_id：
   - 使用错误的store_id尝试发货，验证系统返回状态码应非200。
4. test_error_order_id：
   - 使用错误的order_id尝试发货，验证系统返回状态码应非200。
5. test_error_seller_id：
   - 修改卖家ID后进行发货，验证系统返回状态码应非200。
6. test_error_buyer_id：
   - 在买家接收订单之前修改买家ID，验证系统返回状态码应非200。
7. test_ship_not_pay：
   - 尝试发货未支付的订单，验证系统返回状态码应非200。
8. test_receive_not_ship：
   - 尝试接收未发货的订单，验证系统返回状态码应非200。
9. test_repeat_ship：
   - 验证重复发货的订单，系统第二次应返回状态码非200。

10. test_repeat_receive：
    - 买家接收同一订单两次，第二次接收应返回状态码非200。

测试书本和店铺收藏夹（TestCollection）

1. test_ok：
    - 环境准备：注册新的买家和卖家账户，创建商店。
    - 操作步骤：
        a. 收藏两本书，每次操作后验证返回状态码为200，确认收藏成功。
        b. 获取并验证用户收藏的图书列表，确保返回码为200，表示查询成功。
        c. 取消这两本书的收藏，每次操作后验证状态码为200，确认取消成功。
        d. 收藏一个商店，然后验证状态码为200，确认收藏成功。
        e. 获取并验证收藏的商店列表，确保状态码为200，表示查询成功。
        f. 取消收藏的商店，验证状态码为200，确保取消成功。

成果

项目链接：https://github.com/manchestersskyisred/ECNU-Bookstoredemo

```
frontend end test
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/buyer.py	267	81	90	30	68%
be/model/database.py	21	0	2	0	100%
be/model/db_conn.py	23	0	6	0	100%
be/model/error.py	25	1	0	0	96%
be/model/seller.py	93	36	36	3	64%
be/model/user.py	173	47	46	5	75%
be/serve.py	35	1	2	1	95%
be/view/auth.py	109	29	24	11	70%
be/view/buyer.py	171	38	30	14	74%
be/view/seller.py	75	17	18	9	72%
fe/__init__.py	0	0	0	0	100%
fe/access/__init__.py	0	0	0	0	100%
fe/access/auth.py	36	0	0	0	100%
fe/access/book.py	67	2	10	1	96%
fe/access/buyer.py	88	0	0	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	20	2	98%
fe/conf.py	11	0	0	0	100%
fe/conftest.py	21	0	0	0	100%
fe/test/gen_book_data.py	49	0	16	0	100%
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_cancel_order.py	58	0	0	0	100%
fe/test/test_collection.py	34	0	4	0	100%
fe/test/test_create_store.py	20	0	0	0	100%
fe/test/test_get_order_history.py	27	0	0	0	100%
fe/test/test_login.py	28	0	0	0	100%
fe/test/test_new_order.py	40	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_register.py	31	0	0	0	100%
fe/test/test_search.py	82	0	8	0	100%
fe/test/test_ship_receive.py	95	0	0	0	100%
TOTAL	2119	259	356	78	86%

```
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_store_id PASSED                [ 15%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_book_id PASSED                 [ 17%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_ok PASSED                            [ 19%]
fe/test/test_bench.py::test_bench PASSED                                                      [ 21%]
fe/test/test_cancel_order.py::TestCancelOrder::test_ok PASSED                                 [ 22%]
fe/test/test_cancel_order.py::TestCancelOrder::test_wrong_user_id PASSED                      [ 24%]
fe/test/test_cancel_order.py::TestCancelOrder::test_non_exist_order_id PASSED                 [ 26%]
fe/test/test_cancel_order.py::TestCancelOrder::test_repeat_cancel PASSED                      [ 28%]
fe/test/test_cancel_order.py::TestCancelOrder::test_cancel_paid_order PASSED                  [ 29%]
fe/test/test_cancel_order.py::TestCancelOrder::test_cancel_long_time_order PASSED             [ 31%]
fe/test/test_collection.py::TestCollection::test_ok PASSED                                    [ 33%]
fe/test/test_create_store.py::TestCreateStore::test_ok PASSED                                 [ 35%]
fe/test/test_create_store.py::TestCreateStore::test_error_exist_store_id PASSED               [ 36%]
fe/test/test_get_order_history.py::TestGetOrderHistory::test_ok PASSED                        [ 38%]
fe/test/test_get_order_history.py::TestGetOrderHistory::test_non_exist_user_id PASSED         [ 40%]
fe/test/test_login.py::TestLogin::test_ok PASSED                                              [ 42%]
fe/test/test_login.py::TestLogin::test_error_user_id PASSED                                   [ 43%]
fe/test/test_login.py::TestLogin::test_error_password PASSED                                  [ 45%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_book_id PASSED                        [ 47%]
fe/test/test_new_order.py::TestNewOrder::test_low_stock_level PASSED                          [ 49%]
fe/test/test_new_order.py::TestNewOrder::test_ok PASSED                                       [ 50%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_user_id PASSED                        [ 52%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_store_id PASSED                       [ 54%]
fe/test/test_password.py::TestPassword::test_ok PASSED                                        [ 56%]
fe/test/test_password.py::TestPassword::test_error_password PASSED                            [ 57%]
fe/test/test_password.py::TestPassword::test_error_user_id PASSED                             [ 59%]
fe/test/test_payment.py::TestPayment::test_ok PASSED                                          [ 61%]
fe/test/test_payment.py::TestPayment::test_authorization_error PASSED                         [ 63%]
fe/test/test_payment.py::TestPayment::test_not_suff_funds PASSED                              [ 64%]
fe/test/test_payment.py::TestPayment::test_repeat_pay PASSED                                  [ 66%]
fe/test/test_register.py::TestRegister::test_register_ok PASSED                               [ 68%]
fe/test/test_register.py::TestRegister::test_unregister_ok PASSED                             [ 70%]
fe/test/test_register.py::TestRegister::test_unregister_error_authorization PASSED            [ 71%]
fe/test/test_register.py::TestRegister::test_register_error_exist_user_id PASSED              [ 73%]
fe/test/test_search.py::TestSearch::test_search_global PASSED                                 [ 75%]
fe/test/test_search.py::TestSearch::test_search_global_not_exists PASSED                      [ 77%]
fe/test/test_search.py::TestSearch::test_search_in_store PASSED                               [ 78%]
fe/test/test_search.py::TestSearch::test_search_not_exist_store_id PASSED                     [ 80%]
fe/test/test_search.py::TestSearch::test_search_in_store_not_exist PASSED                     [ 82%]
fe/test/test_ship_receive.py::TestShipReceive::test_ship_ok PASSED                            [ 84%]
fe/test/test_ship_receive.py::TestShipReceive::test_receive_ok PASSED                         [ 85%]
fe/test/test_ship_receive.py::TestShipReceive::test_error_store_id PASSED                     [ 87%]
fe/test/test_ship_receive.py::TestShipReceive::test_error_order_id PASSED                     [ 89%]
fe/test/test_ship_receive.py::TestShipReceive::test_error_seller_id PASSED                    [ 91%]
fe/test/test_ship_receive.py::TestShipReceive::test_error_buyer_id PASSED                     [ 92%]
fe/test/test_ship_receive.py::TestShipReceive::test_ship_not_pay PASSED                       [ 94%]
fe/test/test_ship_receive.py::TestShipReceive::test_receive_not_ship PASSED                   [ 96%]
fe/test/test_ship_receive.py::TestShipReceive::test_repeat_ship PASSED                        [ 98%]
fe/test/test_ship_receive.py::TestShipReceive::test_repeat_receive PASSED                     [100%]

============================================ 57 passed in 57.19s ============================================
```

大部分没覆盖到的代码都是一些不会被执行的或者一些异常抛出的代码，为了代码功能的完整和结构的严谨，我们保留了这部分代码

分工&协作

本项目由团队成员紧密协作完成，充分利用 Git 进行版本控制与协同开发。各位成员在负责核心模块开发的同时，积极参与了代码审查、功能测试、BUG修复及报告撰写等工作，具体分工如下：

- **柳絮源 (33.33%):** 负责 **用户管理与认证 (`user.py`)** 模块的增强，主导开发并测试了核心附加功能——**书籍搜索 (`search_book`)**，涵盖后端逻辑、API 设计及前端接口模拟，参与代码仓库的主要维护工作
- **吕佳鸿 (33.33%):** 负责 **卖家核心业务逻辑 (`seller.py`)** 的实现（店铺创建、图书上/下架、库存管理），并开发了订单生命周期中的关键环节，如 **订单自动取消、发货与收货** 功能，参与代码仓库的主要维护工作
- **肖岂源 (33.33%):** 负责 **买家核心业务逻辑 (`buyer.py`)** 的实现（下单、支付、历史订单、充值），并独立开发完成了 **收藏夹（书籍/店铺）** 功能及其相关接口与测试，参与代码仓库的主要维护工作

所有成员均深度参与了项目各阶段，确保了功能的完整性和代码质量。Git 提交历史（见截图）反映了团队的协作过程。

git使用情况：

```
(base) kerwinlv@kerwindeMacBook-Pro-2 ECNU-Bookstoredemo % git log --oneline | cat
bd31aa8 update be
15110a0 data
3bf5a05 update fe
6d6a7fa error solve
4fc1343 update fe
a9c8f2e data
f5f1169 update book
022949a update fe/auth
d250256 first update
f4c92f6 Initial commit
```