# Bookstore项目报告


**姓名：　张思雨　学号：　10214804409**


# 一、实验要求

> 本项目旨在实现一个提供网上购书功能的网站后端。该后端系统需支持书商开设网上商店，并允许购买者通过网站进行图书购买。系统包含买家和卖家的注册与登录功能，支持买家充值、下单、付款等操作，以及卖家创建店铺、添加书籍信息、增加库存等管理功能。此外，项目还要求实现订单的下单-付款-发货-收货流程，并添加搜索图书、订单状态查询、取消订单等额外功能。

## 具体功能要求

---

**用户权限接口**：实现注册、登录、登出、注销等用户权限管理功能。

**买家用户接口**：提供充值、下单、付款等买家相关功能。

**卖家用户接口**：实现创建店铺、添加书籍信息、增加库存等卖家管理功能。

**订单流程**：支持下单、付款、发货、收货等完整订单流程。

**搜索图书**：实现基于关键字、标签、目录、内容的图书搜索功能，支持全站或当前店铺搜索，并具备分页显示能力。

**订单状态查询与取消**：允许用户查询历史订单状态，并支持买家主动取消订单或系统自动取消超时未支付的订单。

# 二、项目运行

**Python版本**：Python 3.6或更高版本，确保兼容性和性能。使用 `pip`安装Python依赖包，包括Flask、SQLAlchemy、Elasticsearch客户端等。

**数据库**：在本项目中，我们使用 PostgreSQL 作为关系型数据库管理系统，并通过 SQLAlchemy 库来实现与数据库的交互操作。PostgreSQL 服务器运行在本地计算机上（在实际项目部署时，可根据需求配置为远程服务器）。我们通过 SQLAlchemy 的 `create_engine` 函数创建一个数据库引擎实例，连接到 PostgreSQL 数据库。

在 PostgreSQL 中，我们创建了名为 bookstore 的数据库（与文档中一致）。在该数据库中创建了以下表结构，其设计理念及详细结构见 “三、数据库设计” 中的相关内容

# 三、数据库设计

## 3.1 基本假设

1、用户会频繁下单、付款和取消。

2、买家下单后，未付款的订单会在指定时间（如24小时）后自动取消。

3、买家余额足够支付全额时才可以完成付款流程，不支持分期支付。

4、买家仅可取消下单未付款或付款未发货的订单，已发货或已收货的订单不可取消。

5、订单状态变更流程固定：下单 -> 付款 -> 发货 -> 收货，或下单-> 取消，或下单-> 付款->取消，不能跳过任意环节。

6、买家充值账户时，只允许使用特定货币（如人民币），且充值金额不能为负。

7、图书搜索支持标题、作者、标签和简介字段的匹配，但不包含详细内容（内容索引视需求）。

8、搜索结果分页，默认每页显示20条记录，结果过多时可以限制页数。

9、一次性购买多种书籍，按照店铺进行下单，即每一个订单仅包含在同一个店铺的书籍。

10、每种书籍只能在单个店铺中上架一次，不能重复上架在相同店铺。

11、用户会经常犯错但一般不会故意犯错

12、相对于卖家来说，买家的数量更多且权益应受到保护

13.、用户在搜索时对书籍并不了解

# 3.2 关系数据库设计

### 3.2.1 users table（用户表）

- **设计思路**：用于存储用户实体类及其属性，每一行对应一个用户，记录其基本信息。
- **表格结构**：
- | 字段名 | 数据类型 | 描述 |
  | --- | --- | --- |
  | user_id | text（字符串类型） | 主键，用于记录用户名，具有唯一性 |
  | password | text（字符串类型） | 存储用户密码的暗文，确保安全性，不存储明文密码 |
  | balance | integer（整数类型） | 记录用户账户内的金额，初始值为0 |
  | token | text（字符串类型） | 用于记录登录时用户名、时间和终端号生成的标记，在重要操作时用于验证消息来源 |
  | terminal | text（字符串类型） | 记录登录时终端号 |

### 3.2.2 user_store table（用户 - 商店关联表）

- **设计思路**：存储商店实体类以及开店联系类，每行代表一家店，建立用户与商店之间的关联关系。
- **表格结构**：
- | 字段名 | 数据类型 | 描述 |
  | --- | --- | --- |
  | user_id | text（字符串类型） | 外键，关联users表中的user_id，用于记录商店的店主的用户名 |
  | store_id | text（字符串类型） | 主键，用于记录商店名，具有唯一性 |

### 3.2.3 store table（商店库存表）

- **设计思路**：存储书店的库存信息，每行表示一本书在一家商店的库存情况。
- **表格结构**：

| 字段名 | 数据类型 | 描述 |
| --- | --- | --- |
| store_id | text（字符串类型） | 主键之一，外键，关联user_store表中的store_id，用于记录书籍所在的商店名 |
| book_id | text（字符串类型） | 主键之一，用于记录对应书籍的id |
| stock_level | integer（整数类型） | 记录书籍的库存数 |
| price | integer（整数类型） | 记录书籍的单价，不同商店中同一本书的单价可以不同 |

### 3.2.4 new_order table（新订单表）

- **设计思路**：存储订单相关信息，每条记录与书籍无关，记录订单整体信息。
- **表格结构**：

| 字段名 | 数据类型 | 描述 |
| --- | --- | --- |
| order_id | text（字符串类型） | 主键，记录订单的订单号 |
| user_id | text（字符串类型） | 外键，关联users表中的user_id，记录订单买家的用户名 |
| store_id | text（字符串类型） | 外键，关联store表中的store_id，记录订单的商家的商店名 |
| status | integer（整数类型） | 记录订单的状态，默认值为1，1表示已下单未付款，2表示已付款未发货，3代表已发货未收货，4表示已收货，0表示订单已取消 |
| total_price | integer（整数类型） | 记录订单的总价 |
| order_time | integer（整数类型） | 记录订单的下单时间，用时间戳的形式表示，用于自动取消等业务 |

### 3.2.5 new_order_detail_table（新订单详情表）

- **设计思路**：存储订单与书籍购买联系类的详细信息，每条记录表示某订单中购买某本书的信息。
- **表格结构**：

| 字段名 | 数据类型 | 描述 |
| --- | --- | --- |
| order_id | text（字符串类型） | 主键之一，外键，关联new_order表中的order_id，记录订单的订单号 |
| book_id | text（字符串类型） | 主键之一，用于记录订单中对应书籍的id，关联store表中的book_id |
| count | integer（整数类型） | 记录该订单中购买这本书的数量 |

### 3.2.6 invert_index table（倒排索引表）

- **设计思路**：倒排表，存储关键词与书籍的索引关系，用于加速书籍搜索。
- **表格结构**：

| 字段名 | 数据类型 | 描述 |
| --- | --- | --- |
| search_key | text（字符串类型） | 主键之一，记录倒排表的关键字，当用户输入倒排表中的内容时，可通过该字段查找相关书籍信息 |
| search_id | integer（整数类型） | 主键之一，序列类型，可根据关键字插入顺序生成自增的整数序列，用于分页查询 |
| book_id | text（字符串类型） | 外键，关联store表中的book_id，记录对应书籍的id，是倒排表的返回值之一 |
| book_title | text（字符串类型） | 记录对应书籍的标题，作为冗余属性返回便于买家查看 |
| book_author | text（字符串类型） | 记录对应书籍的作者，作为冗余属性返回便于买家查看 |

### 3.2.7 book collection（书籍表）

- **设计思路**：用于存储书籍的详细信息，每一个文档对象代表一本书的完整信息。
- **表格结构**：

- | 字段名 | 数据类型 | 描述 |
  | --- | --- | --- |
  | id | string（字符串类型） | 记录书的id，每本书的id具有唯一性 |
  | title | string（字符串类型） | 记录书的标题，理论上不可为空 |
  | author | string（大部分情况下）/ null（可空） | 记录书的作者 |
  | publisher | string（字符串类型） | 记录书的出版社 |
  | original_title | null（大部分情况下）/ string（非空时） | 记录外文书的原标题 |
  | translator | null（大部分情况下）/ string（非空时） | 记录外文书的译者 |
  | country | null（大部分情况下）/ string（非空时） | 记录外文书作者的国籍 |
  | pub_year | string（字符串类型） | 记录书的出版年月 |
  | pages | int32（32位整数类型） | 记录书的页码数 |
  | currency_unit | string（可为空字符串） | 记录书价格的货币单位，大部分为“元” |
  | isbn | string（字符串类型） | 记录书的ISBN号 |
  | author_intro | string（可为空字符串） | 记录书的作者简介 |
  | book_intro | string（可为空字符串） | 记录书的简介 |
  | content | string（可为空字符串） | 记录书的目录 |
  | tags | array（数组类型，每项为字符串） | 记录书的标签和关键字 |
  | pictures | array（数组类型，每项为一张图片） | 记录书的图片 |

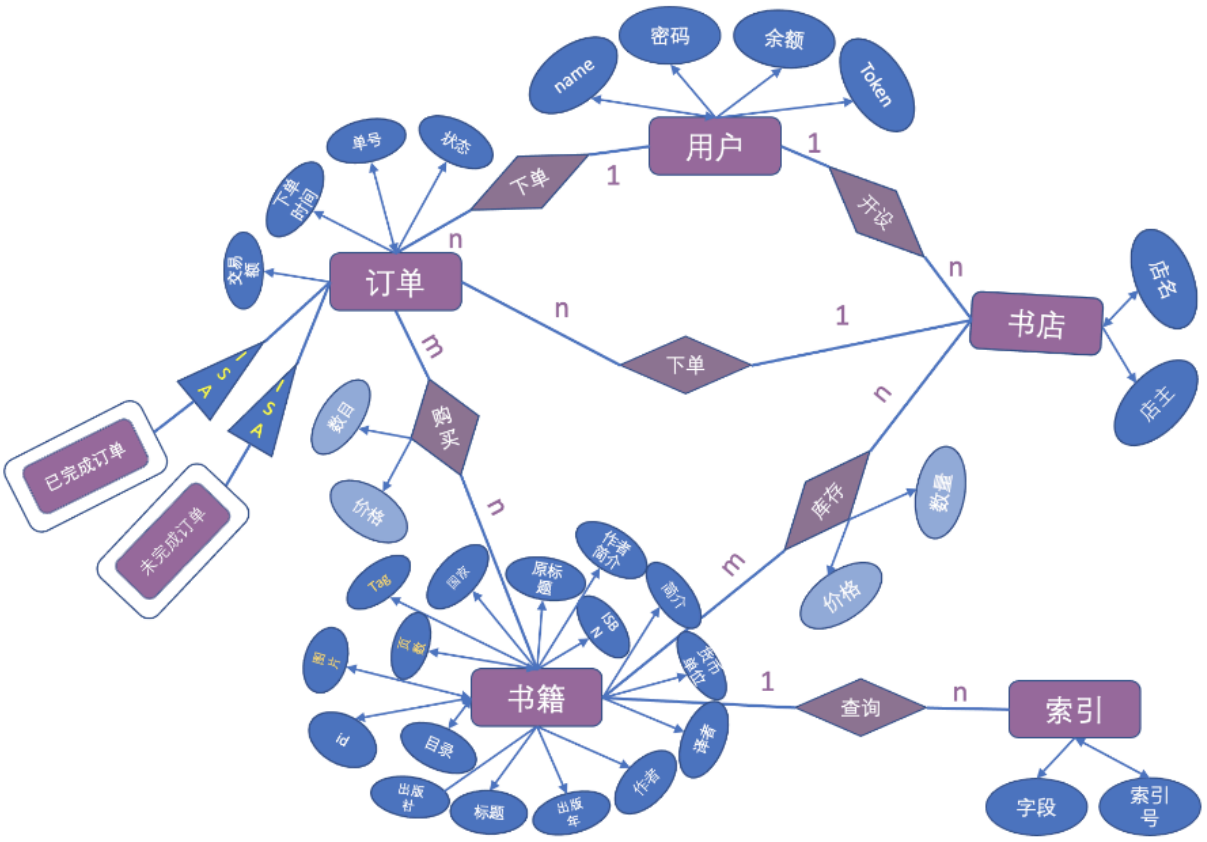
#### 3.2.8 history_order collection（历史订单表）

- **设计思路**：用于存储用户的历史订单信息，每一个文档对象代表一条完整的历史订单记录。
- **表格结构**：

- 

| 字段名 | 数据类型 | 描述 |
| --- | --- | --- |
| order_id | string（字符串类型） | 记录订单的订单号，与PostgreSQL数据库中的new_order表中的order_id对应 |
| user_id | string（字符串类型） | 记录订单所属用户的id，关联users表中的user_id |
| store_id | string（字符串类型） | 记录订单所属商店的id，关联store表中的store_id |
| status | integer（整数类型） | 记录订单的状态，取值只有0（已取消）和4（已收货）两种，与PostgreSQL数据库中的订单状态相关联 |
| total_price | integer（整数类型） | 记录订单的总价，与PostgreSQL数据库中的new_order表中的total_price对应 |
| order_time | integer（整数类型） | 记录订单的下单时间，与PostgreSQL数据库中的new_order表中的order_time对应 |
| books | array（数组类型） | 数组中每一项是一个子文档，子文档包含book_id（字符串类型，记录订单中书籍的id，关联store表中<br>的book_id）和count（整数类型，记录购买该书的数量），用于记录订单中的书籍信息，与PostgreSQL<br>数据库中的new_order_detail_table中的数据对应 |

# 3.3ER图




# 3.4索引创建

只用到了主键索引，没有使用其他索引。原因是在建表时已经考虑到了查询等因素，故不建立其他索引也能有较好的表现。

# 3.5事务处理

事务处理细节见具体功能函数

# 四、功能实现

# 4.1 用户权限功能

### 4.1.1用户注册

- **功能实现**:

1. 对用户输入的密码进行加密处理，使用 `bcrypt` 库的 `hashpw` 函数结合随机生成的盐值对密码进行哈希加密，得到加密后的密码。
2. 生成终端信息 `terminal`，格式为 "terminal_{当前时间戳}"，并使用 `jwt` 库（假设 `jwt_encode` 函数基于此库实现）根据用户ID和终端信息生成令牌 `token`。
3. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
4. 在事务块中执行数据库插入操作，尝试将用户信息（包括用户ID、加密后的密码、初始余额0、生成的令牌、终端信息）插入到 `users` 表中。如果插入过程中发生完整性错误（如用户ID已存在导致违反唯一约束），事务自动回滚，并返回 `error.error_exist_user_id(user_id)`。若事务执行成功（没有发生异常），则在事务块结束时自动提交事务。最后关闭会话以释放资源。

- **数据库操作**：

1. 执行一次插入操作，将用户信息插入到 `users` 表中。但由于添加了事务处理，实际涉及到与数据库的多次交互，包括开启事务、执行插入语句、根据结果决定提交或回滚事务等操作。在并发情况下，事务处理确保了多个线程同时操作数据库时的数据一致性。例如，当多个线程同时尝试插入相同的用户ID时，只有一个线程能够成功插入，其他线程会因为违反唯一约束而触发事务回滚，保证了用户ID的唯一性约束不被破坏。

- **功能接口**：

1. 依旧创建了一个名为 `bp_user` 的Flask蓝图，定义了一个路由 `/register`，接受 `POST` 方法。
2. 在函数 `user_register` 中，从请求的JSON数据中获取用户ID和密码，创建用户类的实例，调用其注册方法（即上述修改后的 `register` 方法），并根据方法返回的代码和消息以JSON格式返回。

- **测试用例**：

1. 使用 `pytest` 框架编写测试，首先导入了 `pytest`、自定义的 `new_user` 模块中的 `register_user` 函数以及 `uuid` 模块（假设 `new_user` 模块及 `register_user` 函数与当前 `register` 函数存在合理的测试调用关系）。
2. 在 `TestUserRegistration` 类中，`pre_run_initialization` 方法作为 `pytest.fixture` 在测试前初始化生成唯一的 `user_id` 和 `password`。
3. `test_user_id_not_exist` 方法用于测试正常注册的情况，先调用注册接口创建一个新用户，然后断言返回码为200，确保用户信息被成功插入，并且返

回中包含 JWT 令牌（依据实际代码中 register 函数返回值结构进行准确断言）。
4. test_user_id_exists 方法先注册新用户（断言返回码为200），然后再次用相同的 user_id 注册，断言返回码不等于200，以此测试创建已存在用户ID时的错误情况（依赖于 register 函数在用户ID已存在时返回正确的错误码）。通过这些测试用例，可以验证注册功能在正常和异常情况下的正确性，以及事务处理对数据完整性的保障作用。例如，在测试用户ID已存在的情况时，由于事务回滚机制，第二次插入相同用户ID时应触发错误并返回正确的错误码，保证数据库中不会出现重复的用户ID记录。

- user_id和错误的 password调用登录接口，断言返回码为401，并检查返回中包含错误信息 error_authorization_fail。

#### 4.1.2 用户登录

- **功能实现**：

1. 首先调用 check_password 函数验证用户身份，若密码正确，继续后续操作；若密码错误，直接返回相应错误码和消息（由 check_password 函数确定）。
2. 若密码验证通过，使用 jwt_encode 函数（假设基于 jwt 库实现）根据用户ID和传入的终端信息生成新的 JWT 令牌。
3. 创建数据库会话，通过 sessionmaker 绑定已有的数据库连接 self.conn 来创建会话对象 session。
4. 在事务块中执行更新操作，使用生成的令牌和终端信息更新 users 表中对应用户的记录。如果更新影响的行数为0（表示未找到对应的用户记录），则回滚事务并返回 error.error_authorization_fail()。若更新成功，提交事务。最后关闭会话以释放资源。

- **数据库操作**：

1. 执行一次查询操作（在 check_password 函数中，假设该函数内部会执行一次数据库查询来验证密码，此处计为一次数据库访问）用于验证用户密码。
2. 执行一次更新操作，更新用户的 JWT 令牌和终端信息。通过事务处理，确保了这两个操作的原子性，即在密码验证通过后，要么成功更新用户信息并提交事务，要么在任何一个环节出现问题（如未找到用户记录、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免了多个线程同时更新同一用户记录可能导致的数据冲突或不一致问题。

- **功能接口**：

1. 创建了一个名为 `bp_user` 的Flask蓝图，定义了一个路由 `/login`，接受 `POST` 方法。
2. 在函数 `user_login` 中，从请求的JSON数据中获取用户ID、密码和终端信息，创建用户类的实例，调用其登录方法（即上述修改后的 `login` 方法），并根据方法返回的代码、消息和令牌以JSON格式返回。

- **测试用例**：

1. 使用 `pytest` 框架编写测试，首先导入了 `pytest`、自定义的 `new_user` 模块中的 `login_user` 函数以及 `uuid` 模块（假设 `new_user` 模块及 `login_user` 函数与当前 `login` 函数存在合理的测试调用关系）。
2. 在 `TestUserLogin` 类中，`pre_run_initialization` 方法作为 `pytest.fixture` 在测试前初始化生成唯一的 `user_id` 和 `password`。
3. `test_login_success` 方法用于测试正常登录的情况，先调用注册接口创建一个新用户，然后使用正确的 `user_id`、`password` 和终端信息调用登录接口，并断言返回码为200，确保生成新的 `JWT` 令牌（依据实际代码中 `login` 函数返回值结构进行准确断言）。
4. `test_login_fail` 方法使用一个已注册的 `user_id` 和错误的 `password` 调用登录接口，断言返回码为401（假设 `error.error_authorization_fail()` 返回的错误码为401），并检查返回中包含错误信息 `error_authorization_fail`。通过这些测试用例，可以全面验证登录功能在不同情况下的正确性，包括正常登录时令牌的正确生成和更新，以及错误登录时的错误处理，同时确保事务处理机制在各种场景下能保证数据的完整性和一致性。

### 4.1.3 用户登出

- **功能实现**：#### * 调用 `check_token` 函数验证当前传入的 `JWT` 令牌是否有效，若令牌无效，直接返回相应错误码和消息（由 `check_token` 函数确定）。
  - 若令牌验证通过，生成新的终端信息 `terminal`，格式为 "terminal_{当前时间戳}"，并使用 `jwt_encode` 函数根据用户ID和新终端信息生成虚假令牌 `dummy_token`。
  - 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
  - 在事务块中执行更新操作，使用生成的虚假令牌和新终端信息更新 `users` 表中对应用户的记录。如果更新影响的行数为0（表示未找到对应的用户记录），则回滚事务并返回 `error.error_authorization_fail()`。若更新成功，提交事务。最后关闭会话以释放资源。
- **数据库操作**：

1. 执行一次查询操作（在 `check_token` 函数中，假设该函数内部会执行一次数据库查询来验证令牌，此处计为一次数据库访问）用于验证 `JWT` 令牌。
2. 执行一次更新操作，更新用户的令牌和终端信息。通过事务处理，确保了这两个操作的原子性，即在令牌验证通过后，要么成功更新用户信息并提交事务，要么在任何一个环节出现问题（如未找到用户记录、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，防止多个线程同时对同一用户的令牌进行更新而导致的数据冲突或不一致。

- **功能接口**：
  1. 创建了一个名为 `bp_user` 的Flask蓝图，定义了一个路由 `/logout`，接受 `POST` 方法。
  2. 在函数 `user_logout` 中，从请求的JSON数据中获取用户ID和令牌，创建用户类的实例，调用其登出方法（即上述修改后的 `logout` 方法），并根据方法返回的代码和消息以JSON格式返回。
- **测试用例**：
  1. 使用 `pytest` 框架编写测试，首先导入了 `pytest`、自定义的 `new_user` 模块中的 `logout_user` 函数以及 `uuid` 模块（假设 `new_user` 模块及 `logout_user` 函数与当前 `logout` 函数存在合理的测试调用关系）。
  2. 在 `TestUserLogout` 类中，`pre_run_initialization` 方法作为 `pytest.fixture` 在测试前初始化生成唯一的 `user_id` 和 `JWT` 令牌。
  3. `test_logout_success` 方法用于测试有效令牌登出的情况，调用登录接口生成一个有效的 `JWT` 令牌后，使用该令牌调用登出接口，并断言返回码为200，确保令牌被成功替换为虚假令牌（依据实际代码中 `logout` 函数返回值结构进行准确断言）。
  4. `test_logout_fail` 方法则模拟使用一个无效的令牌调用登出接口，断言返回码为401（假设 `error.error_authorization_fail()` 返回的错误码为401），并检查返回中包含错误信息 `error_authorization_fail`。通过这些测试用例，可以全面验证登出功能在不同情况下的正确性，包括正常登出时令牌的正确更新，以及使用无效令牌登出时的错误处理，同时确保事务处理机制在各种场景下能保证数据的完整性和一致性。

#### 4.1.4 用户注销

- **功能实现**：
  1. 调用 `check_password` 函数验证用户密码，若密码不正确，直接返回相应错误码和消息（由 `check_password` 函数确定）。
  2. 若密码验证通过，创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。

3. 在事务块中执行删除操作，根据用户ID从 `users` 表中删除对应的用户记录。如果删除操作影响的行数为1（表示成功删除用户记录），则提交事务；若影响行数不为1（通常为0，表示未找到对应的用户记录），则回滚事务并返回 `error.error_authorization_fail()`。最后关闭会话以释放资源。

- **数据库操作**：
  1. 执行一次查询操作（在 `check_password` 函数中，假设该函数内部会执行一次数据库查询来验证密码，此处计为一次数据库访问）用于验证用户密码。
  2. 执行一次删除操作，从 `users` 表中删除用户信息。通过事务处理，确保了这两个操作的原子性，即在密码验证通过后，要么成功删除用户信息并提交事务，要么在任何一个环节出现问题（如未找到用户记录、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时尝试删除同一用户记录可能导致的数据不一致或错误删除其他用户记录的问题。
- **功能接口**：
  1. 创建了一个名为 `bp_user` 的Flask蓝图，定义了一个路由 `/unregister`，接受 `POST` 方法。
  2. 在函数 `user_unregister` 中，从请求的JSON数据中获取用户ID和密码，创建用户类的实例，调用其注销方法（即上述修改后的 `unregister` 方法），并根据方法返回的代码和消息以JSON格式返回。
- **测试用例**：
  1. 使用 `pytest` 框架编写测试，首先导入了 `pytest`、自定义的 `new_user` 模块中的 `unregister_user` 函数以及 `uuid` 模块（假设 `new_user` 模块及 `unregister_user` 函数与当前 `unregister` 函数存在合理的测试调用关系）。
  2. 在 `TestUserUnregister` 类中，`pre_run_initialization` 方法作为 `pytest.fixture` 在测试前初始化生成唯一的 `user_id` 和 `password`。
  3. `test_unregister_success` 方法用于测试密码正确的注销情况，先通过注册接口创建用户，然后调用注销接口，断言返回码为200，确保用户信息被成功删除（依据实际代码中 `unregister` 函数返回值结构进行准确断言）。
  4. `test_unregister_fail` 方法则模拟使用一个错误的密码调用注销接口，断言返回码为401（假设 `error.error_authorization_fail()` 返回的错误码为401），并检查返回中包含错误信息 `error_authorization_fail`。通过这些测试用例，可以全面验证注销功能在不同情况下的正确性，包括正常注销时用户信息的正确删除，以及密码错误时的错误处理，同时确保事务处理机制在各种场景下能保证数据的完整性和一致性。

#### 4.1.5 更改密码

- **功能实现**：

1. 调用 `check_password` 函数验证旧密码，若旧密码不正确，直接返回相应错误码和消息（由 `check_password` 函数确定）。
2. 若旧密码验证通过，生成新的终端信息 `terminal`，格式为 "terminal_{当前时间戳}"，并使用 `jwt_encode` 函数根据用户ID和新终端信息生成新的 `JWT` 令牌。
3. 使用 `encrypt` 函数（假设实现了密码加密逻辑）对新密码进行加密处理。
4. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
5. 在事务块中执行更新操作，使用加密后的新密码、新令牌和新终端信息更新 `users` 表中对应用户的记录。如果更新影响的行数为0（表示未找到对应的用户记录），则回滚事务并返回 `error.error_authorization_fail()`。若更新成功，提交事务。最后关闭会话以释放资源。

- **数据库操作**：
  1. 执行一次查询操作（在 `check_password` 函数中，假设该函数内部会执行一次数据库查询来验证密码，此处计为一次数据库访问）用于验证旧密码。
  2. 执行一次更新操作，更新用户的密码、令牌和终端信息。通过事务处理，确保了这两个操作的原子性，即在旧密码验证通过后，要么成功更新用户信息并提交事务，要么在任何一个环节出现问题（如未找到用户记录、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，防止多个线程同时更新同一用户的密码等信息而导致的数据冲突或不一致问题。
- **功能接口**：
  1. 创建了一个名为 `bp_user` 的Flask蓝图，定义了一个路由 `/change_password`，接受 `POST` 方法。
  2. 在函数 `user_change_password` 中，从请求的JSON数据中获取用户ID、旧密码和新密码，创建用户类的实例，调用其更改密码方法（即上述修改后的 `change_password` 方法），并根据方法返回的代码和消息以JSON格式返回。
- **测试用例**：
  1. 使用 `pytest` 框架编写测试，首先导入了 `pytest`、自定义的 `new_user` 模块中的 `change_password` 函数以及 `uuid` 模块（假设 `new_user` 模块及 `change_password` 函数与当前函数存在合理的测试调用关系）。
  2. 在 `TestChangePassword` 类中，`pre_run_initialization` 方法作为 `pytest.fixture` 在测试前初始化生成唯一的 `user_id`、`old_password` 和 `new_password`。
  3. `test_change_password_success` 方法用于测试旧密码正确的情况，先通过注册接口创建用户，然后调用更改密码接口，断言返回码为200，确保密码更新成功（依据实际代码中 `change_password` 函数返回值结构进行准确断言）。

4. `test_change_password_fail` 方法则模拟使用错误的旧密码调用更改密码接口，断言返回码为401（假设 `error.error_authorization_fail()` 返回的错误码为401），并检查返回中包含错误信息 `error_authorization_fail`。通过这些测试用例，可以全面验证更改密码功能在不同情况下的正确性，包括旧密码正确时的密码更新和新令牌生成，以及旧密码错误时的错误处理，同时确保事务处理机制在各种场景下能保证数据的完整性和一致性。

# 4.2 卖家功能

### 4.2.1 创建店铺

- **功能实现**：
  1. 调用 `user_id_exist` 函数检查用户ID是否存在，若不存在，则返回 `error.error_non_exist_user_id(user_id)`。
  2. 调用 `store_id_exist` 函数确认商店ID是否已存在，若已存在，则返回 `error.error_exist_store_id(store_id)`。
  3. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
  4. 在事务块中执行插入操作，将新的商店信息（商店ID和用户ID）插入到 `user_store` 表中。如果插入过程中发生错误，事务自动回滚，并返回相应的错误码和消息（如数据库操作错误 `528` 或其他未知错误 `530`）。
  5. 若插入成功，继续在事务块中执行更新操作（假设存在一个更新用户商店列表的函数 `update_user_store_list`），对 `users` 表中对应的用户记录更新其商店列表，添加新创建的商店ID。若更新过程中发生错误，事务回滚，并返回相应错误。若更新成功，提交事务。最后关闭会话以释放资源。
- **数据库操作**：
  1. 执行一次查询操作（在 `user_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查用户ID是否存在，此处计为一次数据库访问）用于检查用户ID。
  2. 执行一次查询操作（在 `store_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查商店ID是否存在，此处计为一次数据库访问）用于检查商店ID。
  3. 执行一次插入操作，向 `user_store` 表中插入新商店信息。
  4. 执行一次更新操作（在 `update_user_store_list` 函数中，假设该函数执行对 `users` 表的更新操作），更新用户的商店列表。通过事务处理，确保了这一系列操作的原子性，即在用户ID和商店ID检查通过后，要么所有操作（插入

商店信息和更新用户商店列表）成功完成并提交事务，要么在任何一个环节出现问题（如用户ID或商店ID不存在、数据库连接问题、插入或更新失败等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时创建相同商店ID或对同一用户的商店列表进行不一致的更新操作。

- **功能接口**：
  1. 创建了一个名为 `bp_seller` 的Flask蓝图，定义了一个路由 `/create_store`，接受 `POST` 方法。
  2. 定义函数 `seller_create_store`，从请求的JSON数据中获取 `user_id` 和 `store_id`，创建 `seller` 类的实例，调用其 `create_store` 方法（即上述修改后的方法），并根据方法返回的代码和消息，以JSON格式返回消息和对应的HTTP状态码。
- **测试用例**：
  1. 使用 `pytest` 框架编写测试，首先导入了 `pytest`、自定义的 `new_seller` 模块中的 `register_new_seller` 函数以及 `uuid` 模块（假设 `new_seller` 模块及相关函数与当前 `create_store` 函数存在合理的测试调用关系）。
  2. 在 `TestCreateStore` 类中，`pre_run_initialization` 方法作为 `pytest.fixture` 在测试前初始化生成唯一的 `user_id`、`store_id` 和 `password`。
  3. `test_ok` 方法用于测试正常创建商店的情况，先注册一个新卖家，然后调用其创建商店的方法，并断言返回码为200，确保商店创建成功（依据实际代码中 `create_store` 函数返回值结构进行准确断言）。
  4. `test_error_exist_store_id` 方法先注册新卖家并成功创建商店（断言返回码为200），然后再次用相同的 `store_id` 创建商店，断言返回码不等于200，以此来测试创建已存在商店ID时的错误情况（依赖于 `create_store` 函数在商店ID已存在时返回正确的错误码）。通过这些测试用例，可以全面验证创建店铺功能在不同情况下的正确性，包括正常创建时的信息插入和用户列表更新，以及创建已存在商店ID时的错误处理，同时确保事务处理机制在各种场景下能保证数据的完整性和一致性。

### 4.2.2 上架图书

- **功能实现**：

  1. 调用 `user_id_exist` 函数检查用户ID是否存在，若不存在，返回 `error.error_non_exist_user_id(user_id)`。
  2. 调用 `store_id_exist` 函数检查商店ID是否存在，若不存在，返回 `error.error_non_exist_store_id(store_id)`。

3. 调用 `book_id_exist` 函数检查书籍ID在指定商店中是否已存在，若已存在，返回 `error.error_exist_book_id(book_id)`。
4. 在MongoDB的 `book` 集合中查找书籍（通过书籍ID），判断待上架图书是否已存在于数据库中。
5. 若书籍已存在，解析 `book_json_str` 为 `book_info_json`，获取价格信息并从字典中移除价格键（因为之后插入 `store` 表时单独处理价格），提取相关信息（如处理作者国籍、提取关键字、构建倒排索引等操作），然后将书籍信息插入到 `store` 表（表示该书籍在指定商店中的库存信息）。
6. 若书籍不存在，进行一系列数据处理操作：
    - 处理作者国籍信息，将其从作者字段中分离并添加到 `book_info_json` 中。
    - 提取书籍简介、作者简介、目录中的关键字，并添加到 `book_info_json` 的标签字段中。
    - 构建倒排索引，将书籍的标题、作者、标签等信息分词后组成后缀，分离前缀作为关键字加入倒排索引表（`invert_index`）。
    - 将处理后的书籍信息插入到MongoDB的 `book` 集合中，并获取插入后的文档ID（`mongo_id`）。
    - 将书籍的库存信息（包括商店ID、书籍ID、库存水平和价格）插入到 `store` 表中。
7. 创建数据库会话（用于关系型数据库操作），通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。在事务块中执行上述插入 `store` 表的操作，如果插入过程中发生错误（如SQLAlchemy相关错误），事务自动回滚，并返回相应的错误码和消息（如 `528` 表示SQLAlchemy错误）。若插入成功，提交事务。最后关闭会话以释放资源。同时，对于MongoDB操作，如果发生错误（如PyMongo相关错误），也返回相应的错误码和消息（如 `529` 表示PyMongo错误）。

- **数据库操作**：

1. 执行一次查询操作（在 `user_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查用户ID是否存在，此处计为一次数据库访问）用于检查用户ID。
2. 执行一次查询操作（在 `store_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查商店ID是否存在，此处计为一次数据库访问）用于检查商店ID。
3. 执行一次查询操作（在 `book_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查书籍ID在指定商店中的存在性，此处计为一次数据库访问）用于检查书籍ID。

4. 执行一次查询操作（在MongoDB的 `book` 集合中查找书籍，此处计为一次数据库访问）用于判断书籍是否已存在。
5. 执行至少一次插入操作（在 `store` 表中插入书籍库存信息，以及可能在MongoDB的 `book` 集合中插入书籍信息，计为至少一次插入操作，因为插入 `book` 集合的操作可能因为书籍已存在而不执行）。此外，在构建倒排索引时，可能会执行多次插入操作到 `invert_index` 表（根据书籍信息分词后的前缀数量而定）。通过事务处理，确保了在关系型数据库操作中的原子性，即在所有检查通过后，对于 `store` 表的插入操作要么全部成功提交事务，要么在出现问题（如数据约束违反、数据库连接问题等）时回滚事务，保证数据的一致性。同时，对于MongoDB操作，虽然没有显式的事务处理（假设MongoDB未配置事务支持），但通过合理的错误处理机制，确保在MongoDB操作出现错误时也能正确返回错误信息，避免数据处于不一致状态。在并发情况下，避免多个线程同时对同一书籍或商店进行不一致的操作（如同时插入相同书籍ID到同一商店等情况）。

- **功能接口**：

1. 定义一个Flask路由处理函数，该函数对应的路由是 `/add_book`。
2. 定义函数 `seller_add_book`，首先从请求的JSON数据中获取 `user_id`、`store_id`、`book_info`（这是一个字典）和 `stock_level`（若未提供则默认为0）。然后创建了 `seller.Seller` 类的一个实例。接着调用该实例的 `add_book` 方法，传入用户ID、商店ID、书籍信息中的 `id`（转换为字符串）、将书籍信息字典转换为字符串后的内容以及库存水平作为参数，并获取返回的状态码 `code` 和消息 `message`。最后，将消息以JSON格式返回，并使用对应的状态码。

- **测试用例**：使用 `pytest` 框架编写的测试类，用于测试添加书籍相关功能。

1. `TestAddBook` 类的结构和初始化：`pre_run_initialization` 方法（`pytest.fixture`）为一个自动使用的测试夹具。测试开始，它首先生成唯一的 `seller_id`、`store_id` 和 `password`（`password` 与 `seller_id` 相同），然后使用 `register_new_seller` 函数注册一个新卖家，并将返回的卖家对象存储在 `self.seller` 中，接着调用卖家对象的 `create_store` 方法创建商店，并断言返回码为200，表示商店创建成功。从 `book.BookDB` 中获取两本图书的信息，并存储在 `self.books` 中。
2. `test_ok` 方法：对于从 `self.books` 中获取的每一本书，调用卖家对象的 `add_book` 方法将书添加到商店（`self.store_id`）中，库存水平设为0，并断言返回码为200，即测试在正常情况下添加书籍是否成功。

3. `test_error_non_exist_store_id` 方法：对于每一本书，尝试使用不存在的商店ID（在正确的 `self.store_id` 基础上添加了"x"）调用 `add_book` 方法，并断言返回码不等于200，用于测试使用不存在的商店ID添加书籍时的错误情况。
4. `test_error_exist_book_id` 方法：首先，对于每一本书，使用正确的商店ID和库存水平调用 `add_book` 方法，并断言返回码为200，确保第一次添加书籍成功。然后，再次对每一本书使用相同的商店ID和库存水平调用 `add_book` 方法，并断言返回码不等于200，用于测试添加已存在书籍ID时的错误情况。
5. `test_error_non_exist_user_id` 方法：对于每一本书，修改卖家ID为不存在的ID（在正确的 `seller_id` 基础上添加了"_x"），然后调用 `add_book` 方法，并断言返回码不等于200，用于测试使用不存在的用户ID添加书籍时的错误情况。通过这些测试用例，可以全面验证上架图书功能在不同情况下的正确性，包括正常上架、使用不存在的用户ID、商店ID或添加已存在书籍ID时的错误处理，同时确保事务处理机制在关系型数据库操作中能保证数据的完整性和一致性，以及MongoDB操作的错误处理机制能正确反馈问题。

#### 4.2.3 添加库存

- **功能实现**：

1. 调用 `user_id_exist` 函数检查用户ID是否存在，若不存在，返回 `error.error_non_exist_user_id(user_id)`。
2. 调用 `store_id_exist` 函数检查商店ID是否存在，若不存在，返回 `error.error_non_exist_store_id(store_id)`。
3. 调用 `book_id_exist` 函数检查书籍ID在指定商店中是否存在，若不存在，返回 `error.error_non_exist_book_id(book_id)`。
4. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
5. 在事务块中执行更新操作，根据商店ID和书籍ID找到对应的库存记录，将库存数量增加指定的数量（`add_stock_level`）。如果更新过程中发生错误（如SQLAlchemy相关错误），事务自动回滚，并返回相应的错误码和消息（如 `528` 表示SQLAlchemy错误）。若更新成功，提交事务。最后关闭会话以释放资源。

- **数据库操作**：

1. 执行一次查询操作（在 `user_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查用户ID是否存在，此处计为一次数据库访问）用于检查用户ID。

2. 执行一次查询操作（在 `store_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查商店ID是否存在，此处计为一次数据库访问）用于检查商店ID。
3. 执行一次查询操作（在 `book_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查书籍ID在指定商店中的存在性，此处计为一次数据库访问）用于检查书籍ID。
4. 执行一次更新操作，更新指定商店中指定书籍的库存数量。通过事务处理，确保了在库存更新操作中的原子性，即在所有检查通过后，库存更新操作要么成功提交事务，要么在出现问题（如数据约束违反、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时对同一库存记录进行不一致的更新操作（如同时增加库存导致数据错误等情况）。

- **功能接口**：

1. 定义一个Flask路由函数，它对应的路由是 `/add_stock_level` 且仅接受 `POST` 请求。
2. 定义函数 `add_stock_level`，首先从请求的JSON数据里获取用户ID、商店ID、书籍ID和要增加的库存数量（若未提供则默认为0）。接着创建一个 `seller.Seller` 类的实例，调用该实例的 `add_stock_level` 方法，并传入获取到的参数，得到返回的状态码和消息。最后，将消息以JSON格式返回，并附带对应的状态码。

- **测试用例**：使用 `pytest` 框架编写的测试类，用于测试添加书籍库存的相关功能。

1. `TestAddStockLevel` 类的结构和初始化：`pre_run_initialization` 方法首先生成唯一的 `user_id`、`store_id` 和 `password`（`password` 与 `user_id` 相同），然后通过 `register_new_seller` 函数注册一个新卖家，并将返回的卖家对象存储为 `self.seller`。接着调用卖家对象的 `create_store` 方法创建商店，并断言返回码为200，表示商店创建成功。从 `book.BookDB` 中获取5本图书的信息并存储在 `self.books` 中。对于获取到的每一本书，调用卖家对象的 `add_book` 方法将书添加到商店（`self.store_id`）中，库存水平设为0，并断言返回码为200，确保书籍添加成功。最后通过 `yield` 暂停，等待测试方法执行。
2. `test_error_user_id` 方法：对于每一本书，获取其 `id`，然后尝试使用错误的 `user_id`（在正确的 `user_id` 基础上添加了"_x"）调用卖家对象的 `add_stock_level` 方法来增加库存，并断言返回码不等于200，用于测试使用错误用户ID增加库存时的错误情况。
3. `test_error_store_id` 方法：对于每一本书，获取其 `id`，然后尝试使用错误的 `store_id`（在正确的 `store_id` 基础上添加了"_x"）调用卖家对象的

`add_stock_level` 方法来增加库存，并断言返回码不等于200，用于测试使用错误商店ID增加库存时的错误情况。

4. `test_error_book_id` 方法：对于每一本书，获取其 `id`，然后尝试使用错误的 `book_id`（在正确的 `book_id` 基础上添加了"_x"）调用卖家对象的 `add_stock_level` 方法来增加库存，并断言返回码不等于200，用于测试使用错误书籍ID增加库存时的错误情况。
5. `test_ok` 方法：对于每一本书，获取其 `id`，然后使用正确的 `user_id`、`store_id` 和 `book_id` 调用卖家对象的 `add_stock_level` 方法来增加库存，并断言返回码为200，用于测试在正常情况下增加库存是否成功。通过这些测试用例，可以全面验证添加库存功能在不同情况下的正确性，包括正常增加库存、使用错误的用户ID、商店ID或书籍ID时的错误处理，同时确保事务处理机制在库存更新操作中能保证数据的完整性和一致性。

#### 4.2.4 卖家发货

- **功能实现**：

1. 调用 `store_id_exist` 函数检查商店ID是否存在，若不存在，返回 `error.error_non_exist_store_id(store_id)`。
2. 调用 `order_id_exist` 函数检查订单ID是否存在，若不存在，返回 `error.error_invalid_order_id(order_id)`。
3. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
4. 在事务块中执行查询操作，根据订单ID从 `new_order` 表中获取订单状态。如果查询结果为空（表示未找到订单），则回滚事务并返回相应错误。
5. 验证获取到的订单状态是否为已付款状态（假设状态值为2表示已付款），若不是，则回滚事务并返回 `error.error_invalid_order_status(order_id)`。
6. 若订单状态验证通过，在事务块中执行更新操作，将订单状态更新为已发货状态（假设状态值为3表示已发货）。如果更新过程中发生错误（如SQLAlchemy相关错误），事务自动回滚，并返回相应的错误码和消息（如 `528` 表示SQLAlchemy错误）。若更新成功，提交事务。最后关闭会话以释放资源。

- **数据库操作**：

1. 执行一次查询操作（在 `store_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查商店ID是否存在，此处计为一次数据库访问）用于检查商店ID。

2. 执行一次查询操作（在 `order_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查订单ID是否存在，此处计为一次数据库访问）用于检查订单ID。
3. 执行一次查询操作，获取订单的当前状态。
4. 执行一次更新操作，更新订单状态为已发货。通过事务处理，确保了卖家发货操作的原子性，即在所有检查通过后，订单状态更新操作要么成功提交事务，要么在出现问题（如商店或订单不存在、订单状态错误、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时对同一订单进行不一致的操作（如重复发货或对未付款订单发货等情况）。

- **功能接口**:

1. 定义一个Flask蓝图中的路由处理函数。该函数对应的路由是 `/send_books`，且仅接受 `POST` 请求。
2. 在函数内部，首先从请求的JSON数据中获取商店ID和订单ID。接着创建了一个 `seller.Seller` 类的实例，调用该实例的 `send_books` 方法，并传入获取到的商店ID和订单ID，得到返回的状态码和消息。最后，将消息以JSON格式返回，并附带对应的状态码。

- **测试用例**:

1. 定义 `TestSendBooks`类测试卖家发货功能，使用了 `pytest`框架。
2. `pre_run_initialization`方法:（`pytest.fixture`）生成唯一的 `seller_id`、`store_id`、`buyer_id`和 `password`（`password`与 `seller_id`相同）。使用 `register_new_buyer`函数注册一个新买家，并将买家对象存储为 `self.buyer`。创建一个 `GenBook`对象（`self.gen_book`），并通过它获取卖家对象（`self.seller`）。初始化一个临时订单对象为 `None`（`self.temp_order`）
3. `test_status_error`方法: 调用 `self.gen_book.gen`方法生成书籍购买列表，确保生成过程没有问题（`assert ok`）。用 `non_exist_book_id=False`和 `low_stock_level=False`表示生成的书籍ID是存在的且库存足够。 买家使用生成的书籍购买列表创建一个新订单（`self.buyer.new_order`），获取订单ID。卖家尝试发送书籍（`self.seller.send_books`），然后断言返回码不等于200，以测试未付款状态下卖家发货的错误情况。
4. `test_send_books_ok`方法: 同样调用 `self.gen_book.gen`方法生成书籍购买列表并确保生成成功。买家创建新订单，获取订单ID。 买家添加大量资金（`self.buyer.add_funds`），然后对订单进行付款（`self.buyer.payment`）。之后卖家发送书籍，然后断言返回码等于200，用来测试正常付款后卖家发货成功的情况。

5. `test_non_exist_book_id`方法：调用 `self.gen_book.gen`方法，这次 `non_exist_book_id=True`，表示生成的书籍购买列表中存在不存在的书籍ID。买家尝试使用这个包含不存在书籍ID的列表创建订单，断言返回码不等于200，以测试买家使用不存在书籍ID创建订单的错误情况。
6. `test_non_exist_order_id`方法：调用 `self.gen_book.gen`方法生成正常的书籍购买列表并创建订单，获取订单ID。之后，卖家尝试使用错误的订单ID（在正确订单ID基础上加了"_x"）发送书籍，然后断言返回码不等于200，以测试卖家使用不存在的订单ID发货的错误情况。
7. `test_non_exist_store_id`方法：调用 `self.gen_book.gen`方法生成正常的书籍购买列表并创建订单，获取订单ID。卖家尝试使用错误的商店ID（在正确商店ID基础上加了"_x"）发送书籍，然后断言返回码不等于200，以测试卖家使用不存在的商店ID发货的错误情况。

# 4.3 买家功能

### 4.3.1 充值

- **后端逻辑**
  1. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
  2. 在事务块中执行查询操作，根据用户ID在 `users` 表中查询用户信息，获取用户的密码。如果查询结果为空（表示未找到用户），则回滚事务并返回 `error.error_authorization_fail()`。
  3. 比对查询到的密码与传入的密码（使用 `encrypt` 函数加密后）是否一致，若不一致，回滚事务并返回 `error.error_authorization_fail()`。
  4. 若密码验证通过，在事务块中执行更新操作，将用户的余额增加指定的金额（`add_value`）。如果更新影响的行数为0（表示未找到对应的用户记录），则回滚事务并返回 `error.error_non_exist_user_id(user_id)`。若更新成功，提交事务。最后关闭会话以释放资源。
- **数据库操作**
  1. 执行一次查询操作，根据用户ID获取用户的密码信息，用于验证用户身份。
  2. 执行一次更新操作，增加用户的余额。通过事务处理，确保了充值操作的原子性，即在密码验证通过后，要么成功更新用户余额并提交事务，要么在任何一个环节出现问题（如用户不存在、密码错误、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时对同一用户的余额进行不一致的更新操作（如重复充值或充值金额错误等情况）。
- **功能接口**

1. 定义了一个路由 `/add_funds`，用于买家账户充值。
2. 获取请求数据：使用 `request.json.get()` 获取前端发送的JSON请求体中的 `user_id`、`password` 和 `add_value`（充值金额）。
3. 调用 `Buyer` 类的 `add_funds` 方法：创建 `Buyer` 类的实例 `b`，并调用其 `add_funds` 方法，将 `user_id`、`password` 和 `add_value` 传入，以执行具体的充值逻辑。
4. 返回结果：将 `add_funds` 方法返回的状态码 `code` 和消息 `message` 转换为JSON格式并返回给前端。

- **测试用例**
  1. 测试成功充值的情况：使用正确的用户ID、密码和充值金额调用充值接口，断言返回码为200，表示充值成功。
  2. 测试用户ID无效的情况：使用不存在的用户ID调用充值接口，断言返回码不等于200，验证是否正确处理用户ID不存在的情况。
  3. 测试用户密码错误的情况：使用正确的用户ID但错误的密码调用充值接口，断言返回码不等于200，验证是否正确处理密码错误的情况。通过这些测试用例，可以全面验证充值功能在不同情况下的正确性，包括正常充值、用户ID无效和密码错误时的错误处理，同时确保事务处理机制在充值操作中能保证数据的完整性和一致性。

### 4.3.2 下单

- **后端逻辑**
  1. 调用 `user_id_exist` 函数检查用户ID是否存在，若不存在，返回 `error.error_non_exist_user_id(user_id)` 以及空的订单ID。
  2. 调用 `store_id_exist` 函数检查商店ID是否存在，若不存在，返回 `error.error_non_exist_store_id(store_id)` 以及空的订单ID。
  3. 生成唯一的订单ID（`uid`），格式为 "user_id_store_id_uuid"。
  4. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
  5. 遍历购买书籍列表（`id_and_count`），对于每一本书：
     - 在事务块中执行更新操作，根据商店ID和书籍ID检查库存并更新（减少购买数量），同时使用 `RETURNING` 关键字获取该书的价格。如果更新影响的行数为0（表示库存不足或书籍不存在），则回滚事务并返回 `error.error_stock_level_low(book_id)` 以及空的订单ID。
     - 若库存更新成功，获取该书价格，将订单详情（订单ID、书籍ID、购买数量）插入到 `new_order_detail` 表中。
     - 计算订单总价（累计每本书的价格乘以购买数量）。

6. 在事务块中，将订单信息（订单ID、商店ID、用户ID、总价、下单时间）插入到 `new_order` 表中。如果插入过程中发生错误（如SQLAlchemy相关错误），事务自动回滚，并返回相应的错误码、错误消息以及空的订单ID。若插入成功，提交事务，设置订单ID为生成的 `uid`，并将订单ID添加到未付款订单数组（`add_unpaid_order(order_id)`）。最后关闭会话以释放资源。

- **数据库操作**
  1. 执行一次查询操作（在 `user_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查用户ID是否存在，此处计为一次数据库访问）用于检查用户ID。
  2. 执行一次查询操作（在 `store_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查商店ID是否存在，此处计为一次数据库访问）用于检查商店ID。
  3. 对于购买书籍列表中的每一本书，执行一次更新操作检查并更新库存（计为多次更新操作，次数等于购买书籍的数量），以及一次查询操作获取该书价格（通过 `RETURNING` 关键字实现）。
  4. 执行一次插入操作，将订单详情插入到 `new_order_detail` 表中。
  5. 执行一次插入操作，将订单信息插入到 `new_order` 表中。通过事务处理，确保了下单操作的原子性，即在所有检查通过后，库存更新、订单详情插入、订单信息插入等操作要么全部成功提交事务，要么在出现问题（如用户或商店不存在、库存不足、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时对同一订单或库存进行不一致的操作（如重复下单、超卖等情况）。
- **功能接口**
  1. 定义了一个下单路由 `/new_order`，用于创建新订单。
  2. 获取请求数据：从前端的JSON请求体中提取 `user_id`（用户ID）、`store_id`（商店ID）和 `books`（包含书籍ID和数量的列表）。
  3. 整理书籍信息：遍历 `books` 列表，将每本书的 `id` 和 `count` 提取出来，并组合成一个包含 `(book_id, count)` 的列表 `id_and_count`。
  4. 调用 `Buyer` 类的 `new_order` 方法：创建 `Buyer` 类的实例 `b`，并调用 `new_order` 方法，将 `user_id`、`store_id` 和 `id_and_count` 传入，执行下单逻辑。`new_order` 方法返回状态码 `code`、消息 `message` 和订单ID `order_id`。
  5. 返回结果：将 `message` 和 `order_id` 包装成JSON格式返回给前端，并带上状态码 `code`。
- **测试用例**
  1. 测试成功下单的情况：使用有效的用户ID、商店ID和足够库存的书籍列表调用下单接口，断言返回码为200，并检查返回的订单ID不为空，验证订单是否成功创建。

2. 测试无效用户ID的情况：使用不存在的用户ID调用下单接口，断言返回码不等于200，验证是否正确处理用户ID不存在的情况。
3. 测试无效书籍ID的情况：使用有效的用户ID、商店ID和包含无效书籍ID（不存在于商店中）的书籍列表调用下单接口，断言返回码不等于200，验证是否正确处理书籍ID无效的情况。
4. 测试无效商店ID的情况：使用有效的用户ID和不存在的商店ID调用下单接口，断言返回码不等于200，验证是否正确处理商店ID不存在的情况。
5. 测试书籍库存不足的情况：使用有效的用户ID、商店ID和超过库存数量的书籍列表调用下单接口，断言返回码不等于200，验证是否正确处理库存不足的情况。通过这些测试用例，可以全面验证下单功能在不同情况下的正确性，包括正常下单、用户ID、商店ID或书籍ID无效以及库存不足时的错误处理，同时确保事务处理机制在下单操作中能保证数据的完整性和一致性。

#### 4.3.3 付款

- **后端逻辑**
  1. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
  2. 在事务块中执行查询操作，根据订单ID从 `new_order` 表中获取订单信息。如果查询结果为空（表示未找到订单），则回滚事务并返回 `error.error_invalid_order_id(order_id)`。
  3. 验证订单中的买家ID与传入的用户ID是否一致，若不一致，回滚事务并返回 `error.error_authorization_fail()`。
  4. 验证订单状态是否为已下单未付款状态（假设状态值为1表示已下单未付款），若不是，回滚事务并返回 `error.error_invalid_order_status()`。
  5. 检查订单是否超时（调用 `check_order_time` 函数），若超时，提交事务（因为之前的操作未对数据进行修改，提交事务可释放锁等资源），然后删除未付款订单记录（`delete_unpaid_order(order_id)`），取消订单（`o.cancel_order(order_id)`），并返回 `error.error_invalid_order_id()`。
  6. 在事务块中执行查询操作，根据买家ID从 `users` 表中获取买家余额和密码。如果查询结果为空（表示未找到用户），则回滚事务并返回 `error.error_non_exist_user_id(buyer_id)`。
  7. 验证买家密码是否正确，若不正确，回滚事务并返回 `error.error_authorization_fail()`。
  8. 检查买家余额是否足够支付订单总价，若不足，回滚事务并返回 `error.error_not_sufficient_funds(order_id)`。

9. 若余额充足，在事务块中执行更新操作，扣除买家余额（减少订单总价的金额）。如果更新影响的行数为0（表示可能出现并发问题或数据不一致，如余额在检查后被其他操作修改），则回滚事务并返回 `error.error_unknown("update_user_error")`。
10. 在事务块中执行更新操作，将订单状态更新为已付款状态（假设状态值为2表示已付款）。如果更新过程中发生错误（如SQLAlchemy相关错误），事务自动回滚，并返回相应的错误码和消息（如 `528` 表示SQLAlchemy错误）。若更新成功，提交事务，从数组中删除未付款订单记录（`delete_unpaid_order(order_id)`）。最后关闭会话以释放资源。

- **数据库操作**
  1. 执行一次查询操作，根据订单ID获取订单信息。
  2. 执行一次查询操作，根据买家ID获取买家余额和密码。
  3. 执行一次更新操作，扣除买家余额。
  4. 执行一次更新操作，更新订单状态为已付款。通过事务处理，确保了付款操作的原子性，即在所有检查通过后，买家余额扣除、订单状态更新等操作要么全部成功提交事务，要么在出现问题（如订单或用户不存在、密码错误、余额不足、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时对同一订单或用户余额进行不一致的操作（如重复付款、超付、余额错误更新等情况）。
- **功能接口**
  1. 定义一个付款路由 `/payment`，用于处理订单的付款操作。
  2. 获取请求数据：从前端的JSON请求体中提取 `user_id`（用户ID）、`order_id`（订单ID）和 `password`（密码）。
  3. 调用 `Buyer` 类的 `payment` 方法：创建 `Buyer` 类的实例 `b`，并调用其 `payment` 方法，将 `user_id`、`password` 和 `order_id` 传入，以执行付款逻辑。`payment` 方法返回状态码 `code` 和消息 `message`。
  4. 返回结果：将 `message` 包装为JSON格式，并带上状态码 `code` 返回给前端。
- **测试用例**
  1. 测试成功付款的情况：使用有效的用户ID、正确的密码和有效的订单ID（处于可付款状态）调用付款接口，断言返回码为200，表示付款成功。
  2. 测试用户权限错误的情况：使用错误的用户ID或密码调用付款接口，断言返回码不等于200，验证是否正确处理用户权限错误的情况。
  3. 测试买家余额不足情况：使用余额不足的用户账户调用付款接口，断言返回码不等于200，验证是否正确处理余额不足的情况。
  4. 测试重复下单的情况：使用已付款的订单ID再次调用付款接口，断言返回码不等于200，验证是否正确处理重复付款的情况。通过这些测试用例，可以全面验证付款功能在不同情况下的正确性，包括正常付款、用户权限错误、余额不

足和重复付款时的错误处理，同时确保事务处理机制在付款操作中能保证数据的完整性和一致性。

#### 4.3.4 买家收货

- **功能实现**：

  1. 验证用户身份和订单的有效性。
  2. 从订单中获取信息，并通过buyer_id与user_id是否相等来验证订单的买家身份，验证订单的状态是否为"shipped"。
  3. 遍历书籍列表，循环访问 `store_book_collection`数据集合，更新对应书目的库存。
  4. 通过访问 `stores_collection`数据集合，获取卖家ID并检查卖家是否存在。
  5. 使用update_one函数结合 `$inc` 更新 `users_collection`数据集合的balance字段，增加商家收入。
  6. 更新 `orders_collection`数据集合订单状态为“received”，并记录收货时间。
  7. 将订单插入到历史订单集合 `history_orders_collection`，并再次访问 `orders_collection`数据集合以删除原订单。

- **数据库操作**：

  需要访问n+8 次数据库，验证用户身份、订单的有效性及获取订单信息等需访问2次，遍历订单的书籍列表及更新库存需访问n次数据集合，获取卖家ID并检查卖家是否存在需访问2次，增加商家收入需访问1次，更新订单状态需访问1次数据库，将订单插入到历史订单集合并删除原订单需访问2次数据集合。

- **功能接口**：

  定义一个Flask蓝图中的路由处理函数，该函数对应的路由是 `/receive_books`，且只接受 `POST`请求。在函数内部，首先从请求的JSON数据中获取用户ID、订单ID和密码。接着创建了一个 `Buyer`类的实例，调用该实例的 `receive_books`方法，并传入获取到的用户ID、密码和订单ID，得到返回的状态码和消息。最后，将消息以JSON格式返回，并附带对应的状态码。

- **测试用例**：

  测试收货的部分方法和测试卖家发货相同，如 `test_non_exist_store_id`，`test_non_exist_order_id`，`test_non_exist_book_id`，`test_status_error`，这几个方法与 `TestSendBooks`类的方法完全相同，这里便不再介绍了。

1. 定义 `TestReceiveBooks`类测试卖家发货功能，使用了 `pytest`框架。
2. `pre_run_initialization`方法:（`pytest.fixture`）生成唯一的 `seller_id`、`store_id`、`buyer_id`和 `password`（`password`与 `seller_id`相同）。使用 `register_new_buyer`函数注册一个新买家，并将买家对象存储为 `self.buyer`。创建一个 `GenBook`对象（`self.gen_book`），并通过它获取卖家对象（`self.seller`）。初始化一个临时订单对象为 `None`（`self.temp_order`）。
3. `test_receive_books_ok`方法：生成正常购买列表并创建订单，买家添加大量资金并对订单进行付款，卖家发送书籍，然后买家收货，断言返回码为 200，测试正常情况下买家收货成功的情况。
4. `test_receive_books_wstat`方法：生成正常购买列表并创建订单，买家直接尝试收货，断言返回码不等于 200，测试在未满足收货条件下买家收货的错误情况。
5. `test_receive_non_exist_buyer_id`方法：生成正常购买列表并创建订单，卖家发送书籍，然后买家使用错误的买家 ID（在正确 ID 基础上加 "_x"）尝试收货，断言返回码不等于 200，测试使用不存在的买家 ID 收货的错误情况。
6. `test_receive_non_exist_order_id`方法：生成正常购买列表并创建订单，卖家发送书籍，然后买家使用错误的订单 ID（在正确 ID 基础上加 "_x"）尝试收货，断言返回码不等于 200，测试使用不存在的订单 ID 收货的错误情况

#### 4.3.5 用户取消订单

- **后端逻辑**
    1. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
    2. 在事务块中执行查询操作，根据订单ID从 `new_order` 表中获取订单状态。如果查询结果为空（表示未找到订单），则回滚事务并返回相应错误。
    3. 验证订单状态是否为可取消状态（假设只有状态值为1表示可取消，即处于 `pending` 状态的订单才可取消），若不是可取消状态，回滚事务并返回 `error.error_invalid_order_status(order_id)`。
    4. 调用 `user_id_exist` 函数检查用户ID是否存在，若不存在，回滚事务并返回 `error.error_non_exist_user_id(buyer_id)`。
    5. 调用 `order_id_exist` 函数检查订单ID是否存在，若不存在，回滚事务并返回 `error.error_invalid_order_id(order_id)`。
    6. 在事务块中执行更新操作（此处原代码注释掉了相关更新语句，若要完整实现功能，应取消注释并正确执行），将订单状态更新为已取消状态（假设状态值为0表示已取消）。如果更新过程中发生错误（如SQLAlchemy相关错误），事务自动回滚，并返回相应的错误码和消息（如 `528` 表示SQLAlchemy错误）。

7. 从数组中删除未付款订单记录（`delete_unpaid_order(order_id)`），创建 `Order` 类的实例 `o`，调用其 `cancel_order` 方法执行其他与取消订单相关的逻辑（比如可能涉及到在其他相关数据表中的操作等，具体取决于 `cancel_order` 方法的实现）。若所有操作成功，提交事务。最后关闭会话以释放资源。

- **数据库操作**
  1. 执行一次查询操作（在 `user_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查用户ID是否存在，此处计为一次数据库访问）用于检查用户ID。
  2. 执行一次查询操作（在 `order_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查订单ID是否存在，此处计为一次数据库访问）用于检查订单ID。
  3. 执行一次查询操作，获取订单状态。
  4. 执行一次更新操作，更新订单状态为已取消（按功能逻辑需求执行）。通过事务处理，确保了用户取消订单操作的原子性，即在所有检查通过后，订单状态更新等操作要么成功提交事务，要么在出现问题（如用户或订单不存在、订单状态不符合取消条件、数据库连接问题等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时对同一订单进行不一致的取消操作（比如重复取消、对不可取消的订单进行取消等情况）。
- **功能接口**
  1. 定义了一个用户主动取消订单的路由 `/user_cancel_order`。
  2. 获取请求数据：从前端的JSON请求体中提取 `user_id`（用户ID）和 `order_id`（订单ID）。
  3. 调用 `Buyer` 类的 `user_cancel_order` 方法：创建 `Buyer` 类的实例 `b`，并调用其 `user_cancel_order` 方法，将 `user_id` 和 `order_id` 传入，以执行取消订单的逻辑。该方法返回状态码 `code` 和消息 `message`。
  4. 返回结果：将 `message` 包装为JSON格式，并带上状态码 `code` 返回给前端。
- **测试用例**
  1. 测试成功取消订单的情况：使用有效的用户ID和处于可取消状态的订单ID调用取消订单接口，断言返回码为200，表示订单取消成功。
  2. 测试使用无效的订单取消的情况：使用有效的用户ID但不可取消状态（如已付款、已发货等状态）的订单ID调用取消订单接口，断言返回码不等于200，验证是否正确处理无效订单取消的情况。
  3. 测试使用无效的用户取消的情况：使用不存在的用户ID和任意订单ID调用取消订单接口，断言返回码不等于200，验证是否正确处理无效用户取消订单的情况。通过这些测试用例，可以全面验证用户取消订单功能在不同情况下的正确性，包括正常取消、无效订单和无效用户时的错误处理，同时确保事务处理机制在取消订单操作中能保证数据的完整性和一致性。

#### 4.3.6 搜索

- **功能实现**：

  1. 首先判断传入的页码参数 `page` 是否大于0。若大于0，则计算偏移量 `page_lower`，用于分页查询。
  2. 根据页码情况构建不同的SQL查询语句，在倒排索引表（`invert_index`）中根据搜索关键词（`search_key`）进行查询。如果 `page > 0`，则执行带有分页参数的查询，按照 `search_id` 升序排序并限制返回结果数量；若 `page = 0`，则执行全量查询，仅按照 `search_id` 升序排序。
  3. 使用 `cursor.fetchall()` 获取查询结果的所有行。
  4. 将查询结果行转换为包含书籍ID（`bid`）、标题（`title`）和作者（`author`）的字典格式，并添加到结果列表 `result` 中。
  5. 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。在事务块中执行上述查询操作，如果查询过程中发生错误（如SQLAlchemy相关错误），事务自动回滚，并返回相应的错误码、错误消息以及空的结果列表（如 `528` 表示SQLAlchemy错误）。若查询成功，提交事务。最后关闭会话以释放资源。

- **数据库操作**：

  1. 执行一次查询操作，在倒排索引表中根据搜索关键词查找相关书籍信息。通过事务处理，确保了搜索操作的原子性，即在查询过程中，要么成功获取数据并提交事务，要么在出现问题（如数据库连接问题、查询语法错误等）时回滚事务，保证数据的一致性。在并发情况下，避免多个线程同时执行查询操作可能导致的数据不一致或错误结果。

- **功能接口**：

  1. 创建了一个 `/search` 路由，接受 `GET` 方法。
  2. 在 `search_books` 函数中处理搜索请求，支持传入关键字和页码参数，调用 `search` 方法执行搜索逻辑，获取匹配的书籍信息列表，并根据页码分页展示。返回给前端的是包含状态码、消息和搜索结果的JSON数据，以便前端进行相应的处理和展示。

- **测试用例**：

  使用 `pytest` 框架编写 `search` 方法的测试用例。在 `TestSearch` 类中，创建了一个自动使用的 `pytest.fixture`，用于预先加载测试数据集合并确保全文索引已建立。测试分为两部分：

1. **test_search_success**：测试输入有效关键字时的情况。在测试前，先向 `store_book_collection` 插入几条包含关键字的测试数据，通过调用 `search` 方法获取查询结果，断言返回的 `status_code` 为 200，并验证查询结果中包含预期的书籍信息。
2. **test_search_no_results**：测试输入无效或不存在的关键字时的情况。调用 `search` 方法，并断言 `status_code` 为 200，同时检查返回的书籍列表为空，确保方法在未找到匹配数据时的返回正确。

#### 4.3.7 店铺内搜索

- **功能实现**： `search` 和 `searchmany` 方法扩展了店铺内搜索的功能，通过传入 `store_id` 参数进行区分。若传入 `store_id`，则仅在指定店铺中搜索；若不传，则进行全局搜索。
- **数据库操作**：
  - 在搜索和分页搜索中，添加 `store_id` 条件的查询重载操作，查询逻辑与普通搜索一致。
  - 该部分的性能依赖于 `search` 和 `searchmany` 方法的优化情况，查询性能与基本搜索一致。

#### 4.3.8 查询历史订单

- **后端逻辑**

1. **对于 `store_processing_order` 函数**：
   - 调用 `user_id_exist` 函数检查卖家ID是否存在，若不存在，返回 `error.error_non_exist_user_id(seller_id)`。
   - 创建数据库会话，通过 `sessionmaker` 绑定已有的数据库连接 `self.conn` 来创建会话对象 `session`。
   - 在事务块中执行查询操作，通过关联 `new_order` 表和 `user_store` 表，获取该卖家对应的正在处理中的订单相关信息（订单ID、商店ID、状态、总价、下单时间等）。
   - 如果查询结果不为空（有正在处理的订单），遍历查询结果，对于每个订单：
     - 构建订单字典，包含基本信息（订单ID、商店ID等）。
     - 再次在事务块中执行查询操作，根据订单ID从 `new_order_detail` 表中获取该订单包含的书籍信息（书籍ID和数量），并添加到订单字典中。
     - 将构建好的订单字典添加到结果列表 `result` 中。
   - 如果查询结果为空（没有正在处理的订单），设置 `result` 为相应提示信息。最后提交事务，关闭会话以释放资源。若在查询或其他操作过程中发生错误

（如SQLAlchemy相关错误），事务回滚，并返回相应的错误码、错误消息以及空的结果列表（如 `528` 表示SQLAlchemy错误）。

2. **对于 `store_history_order` 函数**：
    - 调用 `store_id_exist` 函数检查商店ID是否存在，若不存在，返回 `error.error_non_exist_store_id(store_id)`。
    - 在MongoDB的 `history_order` 集合中查找指定商店ID的历史订单信息（这里假设是MongoDB操作，具体根据实际数据库类型确定），将查找到的订单信息添加到结果列表 `result` 中。若在查找过程中发生MongoDB相关错误（如 `PyMongoError`），返回相应的错误码、错误消息以及空的结果列表。若出现其他未知错误（`BaseException`），同样返回对应错误码、错误消息和空列表。若查找成功，返回状态码 `200`、消息 `"ok"` 以及包含历史订单信息的结果列表 `result`。

- **数据库操作**

    1. **在 `store_processing_order` 函数中**：
        - 执行一次查询操作（在 `user_id_exist` 函数中，假设该函数内部会执行一次数据库查询来检查用户ID是否存在，此处计为一次数据库访问）用于检查卖家ID。
        - 执行一次查询操作，获取卖家对应的正在处理的订单基本信息。对于每个正在处理的订单，又执行一次查询操作获取其包含的书籍信息（执行次数取决于正在处理订单的数量）。通过事务处理，确保了在查询处理订单相关信息操作中的原子性，即在卖家ID验证通过后，订单基本信息和书籍信息的查询要么全部成功完成并提交事务，要么在出现问题（如数据库连接问题、查询语法错误等）时回滚事务，保证数据的一致性，避免因部分查询成功部分失败导致的数据不一致情况，特别是在并发场景下防止多个线程同时查询同一卖家订单时出现数据混乱。
    2. **在 `store_history_order` 函数中**：执行一次查询操作，在MongoDB的 `history_order` 集合中查找指定商店的历史订单信息（假设此处为一次数据库访问，具体取决于MongoDB的实际操作计数方式）。虽然MongoDB本身的事务特性与传统关系型数据库有所不同，但合理的错误处理机制能确保在出现问题时正确返回错误信息，保证整个查询历史订单功能的稳定性和数据准确性。

- **功能接口**

    1. 定义了一个查询历史订单的路由 `/get_orders`，用于获取用户的历史订单。
    2. 获取请求数据：从前端的JSON请求体中提取 `user_id`（用户ID）。

3. 调用 `Buyer` 类的 `get_orders` 方法：创建 `Buyer` 类的实例 `b`，并调用其 `get_orders` 方法，将 `user_id` 传入，以执行查询订单的逻辑。`get_orders` 方法返回状态码 `code` 和查询结果 `result`。
4. 返回结果：将 `result` 作为订单信息在JSON格式中返回。如果 `code` 为200，则返回订单列表和消息 `"ok"`；否则，返回空列表 `[]` 和 `result` 作为错误消息，同时带上状态码 `code`。

- **测试用例**

1. 测试成功查询订单历史的情况：使用有效的用户ID（卖家ID或与商店相关的有效ID，具体看调用的是哪个函数逻辑）调用查询历史订单接口，断言返回码为200，并检查返回的结果列表包含预期的历史订单信息，验证是否能正确查询到历史订单。
2. 测试当买家没有历史订单时的查询情况：使用有效的用户ID（但该用户确实没有历史订单）调用查询接口，断言返回码为200，且返回的结果列表符合没有订单时的预期格式（如为空列表或包含相应提示信息），验证对于无历史订单情况的处理是否正确。
3. 测试使用无效的用户ID或未注册用户的查询情况：使用不存在的用户ID调用查询接口，断言返回码不等于200，验证是否能正确处理无效用户ID的情况，避免出现错误的查询结果或系统异常。通过这些测试用例，可以全面验证查询历史订单功能在不同情况下的正确性，包括正常查询、无订单以及无效用户ID时的处理，同时确保事务处理机制（关系型数据库部分）和错误处理机制（包括MongoDB操作部分）能保证数据的完整性和整个功能的稳定性。

# 五、亮点展示

1、**密码加密处理**：在用户注册和密码更改时，采用安全的哈希算法, 对用户密码进行加密存储，避免明文密码存储，提升安全性。我们使用的是bcrypt库进行密码哈希,`bcrypt` 是一个广泛使用的密码哈希库，它基于哈希函数提供了安全的哈希算法，可以有效防止暴力破解。

代码片段如下:

```
# **使用 bcrypt 哈希密码**
hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

self.users_collection.insert_one({
    "user_id": user_id,
    # **存储哈希密码**
    "password": hashed_password.decode('utf-8'),
```

```
        "balance": 0,
        "token": token,
        "terminal": terminal
    })
```

使用`bcrypt.hashpw()` 函数将用户输入的密码进行哈希。`password.encode('utf-8')` 将密码字符串转换为字节串，而 `bcrypt.gensalt()` 生成一个随机的盐值（salt），以增强哈希的安全性。将哈希后的密码存储到 MongoDB 数据库中，而不是直接存储明文密码。
`hashed_password.decode('utf-8')` 将字节串转换回字符串以便存储。

2、**单元测试和集成测试**：使用pytest等框架编写测试用例，确保每个功能模块正常工作，提高代码的可维护性和稳定性。

3、**数据规范化**： 在设计数据库时，遵循第三范式（3NF）等规范化原则，避免不必要的数据冗余。将相关的数据拆分成多个表/集合，以减少重复数据的存储。每个商店和书籍的基本信息都独立存储，简化商店和书籍管理，便于扩展。将历史订单移至独立集合，降低 `orders_collection`的访问压力，提高系统整体性能。

4、**适度反规范化**: 在读取性能至关重要的场景下，考虑了适度反规范化，比如user数据集合中保存有stores列表，冗余存储一些常用信息，如store_id，方便得到某一用户创建的所有店铺，适度冗余能够提升性能。

4、**索引创建**：可以在 `user_id`、`store_id`、`book_id`等字段上创建索引，提供了数据的唯一性约束，防止重复数据的出现，同时提高了数据检索的效率。在涉及大量数据的应用中，适当的索引能够显著改善查询性能，减少响应时间。

5、**事务处理**: 在涉及资金变动（如支付和添加余额）时，使用MongoDB的原子操作，确保数据一致性。

# 六、测试结果

```
fe/test/test_add_book.py::TestAddBook::test_ok PASSED                    [  1%]
fe/test/test_add_book.py::TestAddBook::test_error_non_exist_store_id PASSED [  3%]
fe/test/test_add_book.py::TestAddBook::test_error_exist_book_id PASSED   [  5%]
fe/test/test_add_book.py::TestAddBook::test_error_non_exist_user_id PASSED [  7%]
fe/test/test_add_funds.py::TestAddFunds::test_ok PASSED                  [  9%]
fe/test/test_add_funds.py::TestAddFunds::test_error_user_id PASSED       [ 11%]
fe/test/test_add_funds.py::TestAddFunds::test_error_password PASSED      [ 12%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_user_id PASSED [ 14%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_store_id PASSED [ 16%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_book_id PASSED [ 18%]
fe/test/test_add_stock_level.py::TestAddStockLevel::test_ok PASSED       [ 20%]
fe/test/test_bench.py::test_bench PASSED                                 [ 22%]
fe/test/test_change_status.py::TestSendBooks::test_status_error PASSED   [ 24%]
fe/test/test_change_status.py::TestSendBooks::test_send_books_ok PASSED  [ 25%]
fe/test/test_change_status.py::TestSendBooks::test_non_exist_book_id PASSED [ 27%]
fe/test/test_change_status.py::TestSendBooks::test_non_exist_order_id PASSED [ 29%]
fe/test/test_change_status.py::TestSendBooks::test_non_exist_store_id PASSED [ 31%]
fe/test/test_create_store.py::TestCreateStore::test_ok PASSED            [ 33%]
fe/test/test_create_store.py::TestCreateStore::test_error_exist_store_id PASSED [ 35%]
fe/test/test_login.py::TestLogin::test_ok PASSED                         [ 37%]
fe/test/test_login.py::TestLogin::test_error_user_id PASSED              [ 38%]
fe/test/test_login.py::TestLogin::test_error_password PASSED             [ 40%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_book_id PASSED   [ 42%]
fe/test/test_new_order.py::TestNewOrder::test_low_stock_level PASSED     [ 44%]
fe/test/test_new_order.py::TestNewOrder::test_ok PASSED                  [ 46%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_user_id PASSED   [ 48%]
fe/test/test_new_order.py::TestNewOrder::test_non_exist_store_id PASSED  [ 50%]
fe/test/test_password.py::TestPassword::test_ok PASSED                   [ 51%]
fe/test/test_password.py::TestPassword::test_error_password PASSED       [ 53%]
fe/test/test_password.py::TestPassword::test_error_user_id PASSED        [ 55%]
fe/test/test_payment.py::TestPayment::test_ok PASSED                     [ 57%]
fe/test/test_payment.py::TestPayment::test_authorization_error PASSED    [ 59%]
fe/test/test_payment.py::TestPayment::test_not_suff_funds PASSED         [ 61%]
fe/test/test_payment.py::TestPayment::test_repeat_pay PASSED             [ 62%]
fe/test/test_receive_books.py::TestSendBooks::test_status_error PASSED   [ 64%]
fe/test/test_receive_books.py::TestSendBooks::test_send_books_ok PASSED  [ 66%]
fe/test/test_receive_books.py::TestSendBooks::test_non_exist_book_id PASSED [ 68%]
fe/test/test_receive_books.py::TestSendBooks::test_non_exist_order_id PASSED [ 70%]
fe/test/test_receive_books.py::TestSendBooks::test_non_exist_store_id PASSED [ 72%]
fe/test/test_receive_books.py::TestSendBooks::test_receive_books_ok PASSED [ 74%]
fe/test/test_receive_books.py::TestSendBooks::test_receive_books_wstat PASSED [ 75%]
fe/test/test_receive_books.py::TestSendBooks::test_receive_non_exist_buyer_id PASSED [ 77%]
fe/test/test_receive_books.py::TestSendBooks::test_receive_non_exist_order_id PASSED [ 79%]
fe/test/test_register.py::TestRegister::test_register_ok PASSED          [ 81%]
fe/test/test_register.py::TestRegister::test_unregister_ok PASSED        [ 83%]
fe/test/test_register.py::TestRegister::test_unregister_error_authorization PASSED [ 85%]
fe/test/test_register.py::TestRegister::test_register_error_exist_user_id PASSED [ 87%]
fe/test/test_search.py::TestSearch::test_search_books PASSED             [ 88%]
fe/test/test_search.py::TestSearch::test_search_non_exist_keyword PASSED [ 90%]
fe/test/test_search.py::TestSearch::test_searchmany_books PASSED         [ 92%]
fe/test/test_search.py::TestSearch::test_searchmany_non_exist_keyword PASSED [ 94%]
fe/test/test_search_cancelled.py::TestOrder::test_get_order PASSED       [ 96%]
fe/test/test_search_cancelled.py::TestOrder::test_user_cancel_order PASSED [ 98%]
fe/test/test_search_cancelled.py::TestOrder::test_auto_cancel_order PASSED [100%]D:\2024.2\DB\Task\Project1\cdms.xuan_zhou.2024fall.dase\project1\bookstore\
iron['werkzeug.server.shutdown']' function is deprecated and will be removed in Werkzeug 2.1.
  func()
```

| Name | Stmts | Miss | Branch | BrPart | Cover |
|---|---|---|---|---|---|
| be__init__.py | 0 | 0 | 0 | 0 | 100% |
| be\app.py | 5 | 5 | 2 | 0 | 0% |
| be\create.py | 4 | 4 | 0 | 0 | 0% |
| be\model\buyer.py | 226 | 100 | 90 | 21 | 55% |
| be\model\db_conn.py | 20 | 0 | 0 | 0 | 100% |
| be\model\error.py | 27 | 2 | 0 | 0 | 93% |
| be\model\seller.py | 80 | 23 | 24 | 2 | 76% |
| be\model\store.py | 35 | 3 | 0 | 0 | 91% |
| be\model\user.py | 116 | 23 | 30 | 6 | 80% |
| be\serve.py | 36 | 1 | 2 | 1 | 95% |
| be\view\auth.py | 42 | 0 | 0 | 0 | 100% |
| be\view\buyer.py | 62 | 0 | 2 | 0 | 100% |
| be\view\seller.py | 36 | 0 | 0 | 0 | 100% |
| fe__init__.py | 0 | 0 | 0 | 0 | 100% |
| fe\access__init__.py | 0 | 0 | 0 | 0 | 100% |
| fe\access\auth.py | 31 | 0 | 0 | 0 | 100% |
| fe\access\book.py | 59 | 0 | 2 | 0 | 100% |
| fe\access\buyer.py | 91 | 12 | 10 | 4 | 84% |
| fe\access\new_buyer.py | 14 | 5 | 0 | 0 | 64% |
| 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 | 19 | 0 | 0 | 0 | 100% |
| fe\test\gen_book_data.py | 51 | 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_change_status.py | 85 | 0 | 0 | 0 | 100% |
| fe\test\test_create_store.py | 20 | 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 | 37 | 0 | 0 | 0 | 100% |
| fe\test\test_search_cancelled.py | 48 | 3 | 6 | 2 | 87% |
| TOTAL | 1683 | 185 | 246 | 40 | 87% |

```
Wrote HTML report to htmlcov\index.html
(bookstore)
```

# 七、总结与改进

## 7.1 总结

该项目成功构建了用户权限接口，涵盖注册、登录、登出和注销功能，为买家和卖家提供了进入系统的入口，确保了用户操作的合法性和安全性。例如，在注册过程中，对用户输入的信息进行验证和存储，在登录时通过验证用户凭证来授予访问权限。对于买家用户接口，实现了充值、下单和付款等功能。对于家用户接口，支持创建店铺、添加书籍信息及描述、增加库存等操作，卖家可以灵活地开设多个网上商店，并对书籍相关信息进行管理。该项目完整地支持了“下单 -> 付款 -> 发货 -> 收货”的交易流程。这确保了买卖双方的交易能够有序地进行，保障了双方的权益。 此外，该项目也实现了较为全面的图书搜索功能，用户可以通过关键字进行搜索，搜索范围包括题目、标签、目录和内容等，并且支持全站搜索或当前店铺搜索。当搜索结果较多时，采用分页展示，同时利用全文索引优化查找，提高了搜索效率和用户体验。 该项目也实现了较完整的订单管理功能，实现了订单状态查询、订单查询和取消订单功能，用户能够方便地查看自己的历史订单，并且在符合条件时取消订单。

在实现上述功能时，可能涉及到对数据库的频繁操作。对于用户信息、书籍信息、店铺信息、订单信息等都需要进行有效的存储和管理，而本项目设计数据库并创建索引实现了这一需求。数据库设计符合第三范式（3NF），体现了良好的规范化原则，确保了数据的一致性和可维护性。每个字段存储单一的、不可再分的值，避免了数据的重复和复杂结构。这使得数据更易于管理和查询。不同实体（用户、商店、书籍、订单等）被清晰地分开，减少了数据冗余。每个集合的设计专注于特定的业务逻辑，确保数据在逻辑上的一致性。当前设计支持未来功能的扩展。新用户角色、商品类型或其他功能可以在不干扰现有结构的情况下添加。

## 7.2 改进方向

1、**查询优化**：检查数据库查询语句，避免不必要的复杂查询和嵌套查询。对于一些需要获取大量数据的操作，可以考虑采用缓存机制，减少对数据库的直接查询。例如，对于热门书籍的信息，可以缓存到内存中，当用户查询时可以直接从缓存中获取，提高响应速度。

2、**服务器性能**： 如果网站的用户量增加，可以考虑采用负载均衡技术。将用户的请求均匀地分配到多个服务器上，避免单个服务器负载过高，提高系统的整体性能。 对于一些耗时的操作，如订单处理、库存更新等，可以采用异步处理方式。例如，当卖家发货后，不需要让用户等待库存更新等操作完成，可以将这些操作放到后台异步执行，提高用户界面的响应速度。

3、**用户信息安全**：在用户登录和注册过程中，确保用户的密码等敏感信息采用加密传输方式，防止信息在网络传输过程中被窃取。 对用户密码采用更安全的存储方式，如哈希加密并添加盐值，避免用户密码被破解。 对于买家的付款操作，加强与支付平台的安全对接。采用更高级别的安全协议，确保支付过程的安全可靠，防止支付信息泄露和支付

欺诈。在整个交易流程中，采用数据签名等技术保证数据的完整性。防止订单信息、商品信息等在传输和存储过程中被篡改。

4、 **用户评价与反馈**： 增加用户对卖家和书籍的评价与反馈功能。买家在完成交易后，可以对卖家的服务和书籍的质量进行评价，这不仅可以帮助其他买家做出更好的购买决策，也可以促使卖家提高服务质量。

5、 **推荐系统** ： 基于用户的购买历史和浏览行为，开发推荐系统。向用户推荐可能感兴趣的书籍，提高用户的购买率和用户粘性。 通过以上对项目的总结和改进方向的分析，可以进一步完善网上购书网站后端项目，使其在性能、安全和用户体验等方面都得到提升，更好地满足用户和商家的需求。

6、**数据验证**： 在数据输入阶段加入严格的验证机制，确保用户ID、价格、库存等字段的数据格式和范围有效性。这可以减少无效数据的存储，降低后续处理的复杂度。