

数据科学与工程学院实验报告

课程名称：当代数据管理系统	指导教师：周烜	项目名称：Bookstore
组长：段书文	年级：2022级	学号：10224507009
组员：郭力维	年级：2022级	学号：10225501439
组员：蔡祺枫	年级：2022级	学号：10225501411

总览 (Overview)

实验要求

- 创建本地 MongoDB 数据库，将 `bookstore/fe/data/book.db` 中的内容以合适的形式存入本地数据库，后续所有数据读写都在本地的 MongoDB 数据库中进行。
- 完成前 60% 功能的基础上，继续实现后 40% 功能，要有接口、后端逻辑实现、数据库操作、代码测试。对所有接口都要写 `test case`，通过测试并计算测试覆盖率（尽量提高测试覆盖率）。
- 尽量使用索引，对程序与数据库执行的性能有考量。
- 尽量使用 `git` 等版本管理工具。
- 不需要实现界面，只需通过代码测试体现功能与正确性。

本组已完成

- 实现一个网上购书系统的网站后端，支持买家和卖家交互功能
- 实现一个测试前端，用于运行项目测试（功能测试、覆盖率测试、压力测试）代码
- 熟悉 Flask 框架、MongoDB 数据库的使用，掌握接口设计、后端逻辑开发、数据读写与测试
- 完成项目中的基础功能（前 60%）和附加功能（后 40%），确保所有测试用例通过
- 开发拓展功能（收藏功能），编写对应测试代码并通过
- 使用索引提升接口性能（详见 [数据库设计→索引设计](#)）
- 深度使用 `git` 工具进行协同开发
- 优化代码结构，提升代码覆盖率

功能实现

- 用户权限接口（基础）：**
 - 用户注册、用户注销、用户登录、用户登出
- 买家用户接口（基础）：**
 - 充值、下单、支付
- 卖家用户接口（基础）：**
 - 创建店铺、填写图书信息及描述、增加库存

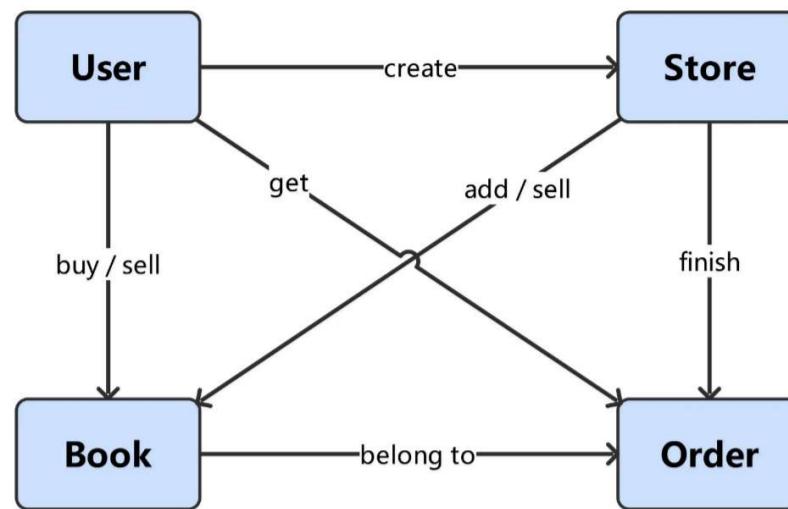
| 基础功能接口严格遵照实验指南 [doc](#) 文件夹中要求实现

- 发货/收货接口（附加）：**
 - 支持卖家发货
 - 支持买家收货
- 图书搜索接口（附加）：**
 - 支持基本搜索功能
 - 支持搜索参数：题目、标签、目录、内容
 - 支持全站搜索
 - 支持店铺内搜索
 - 分页显示：**若结果较多时，支持分页显示结果
 - 性能优化：**建立全文索引提高查询效率
- 订单管理接口（附加）：**
 - 支持查看历史订单
 - 支持取消订单
 - 用户可在付款前主动取消订单
 - 超时未支付订单自动取消（为了测试方便，默认订单超时时间为 10s）
- 收藏栏接口（拓展）：**
 - 支持收藏书籍
 - 支持收藏店铺

数据库设计 (Database)

数据库文档集合设计 (Schema)

基于 `bookstore` 所需要实现的功能，我们绘制如下抽象示意图。其中我们将需要实现的功能拆解为 `User`、`Store`、`Order`、`Book` 四个方面，彼此之间相互依赖。



基于上述示意图，为实现相应功能，我们为 `bookstore` 构建以下文档集合：

文档集合	属性名称
<code>user</code>	<code>user_id</code> , <code>password</code> , <code>balance</code> , <code>token</code> , <code>terminal</code> , <code>collection</code> , <code>store_collection</code>
<code>store</code>	<code>store_id</code> , <code>book_id</code> , <code>stock_level</code> , <code>book_info</code>
<code>order_history</code>	<code>order_id</code> , <code>user_id</code> , <code>store_id</code> , <code>status</code>
<code>order_history_detail</code>	<code>order_id</code> , <code>book_id</code> , <code>count</code> , <code>price</code>
<code>new_order</code>	<code>order_id</code> , <code>user_id</code> , <code>store_id</code>
<code>new_order_detail</code>	<code>order_id</code> , <code>book_id</code> , <code>count</code> , <code>price</code>
<code>user_store</code>	<code>user_id</code> , <code>store_id</code>
<code>books</code>	<code>id</code> , <code>title</code> , <code>author</code> , <code>publisher</code> , <code>original_title</code> , <code>translator</code> , <code>pub_year</code> , <code>pages</code> , <code>price</code> , <code>binding</code> , <code>isbn</code> , <code>author_intro</code> , <code>book_intro</code> , <code>content</code> , <code>tags</code> , <code>pictures</code>

各个文档集合作用如下：

- `user`：存放用户的 ID、密码、状态信息以及收藏信息
- `store`：存放各个店铺中的书籍信息和每本书的存量
- `order_history`：记录历史订单的订单以及订单状态信息
- `order_history_detail`：存放历史订单的细节（书籍编号，购买数量，价格）
- `new_order`：记录未付款订单，付款完成或取消会从库中移除记录，等待付款时间为 10s，超时会自动取消订单；本质是一个队列
- `new_order_detail`：记录未付款订单详情，付款完成或取消会从库中移除记录；本质是一个队列
- `user_store`：存放卖家开店信息
- `books`：存放书籍信息，具体信息如下表所示。注意，该文档集仅用于在功能测试时，生成用于插入 `store` 文档集的 `book_info` 内容；生产环境并不需要该文档集。

变量名	类型	描述	是否可为空
<code>id</code>	string	书籍ID	N
<code>title</code>	string	书籍题目	N
<code>author</code>	string	作者	Y
<code>publisher</code>	string	出版社	Y
<code>original_title</code>	string	原书题目	Y
<code>translator</code>	string	译者	Y
<code>pub_year</code>	string	出版年月	Y
<code>pages</code>	int	页数	Y
<code>price</code>	int	价格(以分为单位)	N
<code>binding</code>	string	装帧，精状/平装	Y
<code>isbn</code>	string	ISBN号	Y
<code>author_intro</code>	string	作者简介	Y
<code>book_intro</code>	string	书籍简介	Y
<code>content</code>	string	样章试读	Y
<code>tags</code>	array	标签	Y
<code>pictures</code>	array	照片	Y

索引设计 (Index)

对于数据库的查找速度来说，索引是十分重要的。[lab2](#) 中我们已经通过实验证明了索引的添加可以有效提高数据库查找的效率。同时索引的引入也可以让逻辑更加清晰简单。在深入分析了实验设计的情况后，我们设计了如下索引策略，以提高数据访问的高效性：

- 针对 `store` 文档集，我们在 `store_id` 和 `book_id` 字段上构建了升序索引，并严格保证了其唯一性。由于 `store_id` 使用次数较多，在订单下达等方面都使用频繁，所以在这里添加索引可以极大提高效率；`store_id` 同理。

- 对于 `user` 文档集，我们在 `user_id` 字段上设置了升序索引，并确保其唯一性。`user_id` 是我们功能的基础，在此添加索引可以有效提高效率。
- 对于 `user_store` 文档集，我们添加了索引 `user_id` 和 `store_id`，这可以有效对于用户创建的商店的查询效率，尤其是判断某用户是否创建了商店的效率。

```
self.database["user"].create_index([("user_id", pymongo.ASCENDING)])
self.database["user_store"].create_index([('user_id', pymongo.ASCENDING), ('store_id', pymongo.ASCENDING)])
self.database["store"].create_index([('book_id', pymongo.ASCENDING), ('store_id', pymongo.ASCENDING)])
```

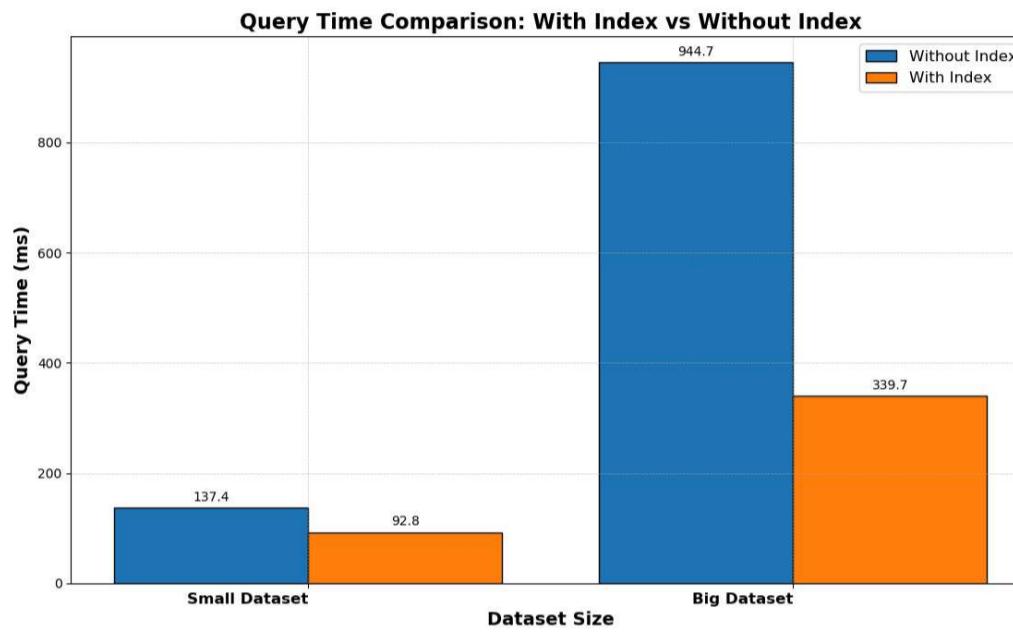
在 `be.model.database` 中初始化数据库时创建索引

- 对于 `books` 文档集，我们在 `id` 字段上添加了索引。这个文档集管理测试用书本信息—因此，该文档集具有大量读需求。使用索引显然能够增加测试表现。

```
try:
    self.db['books'].create_index([('id', pymongo.ASCENDING)])
except:
    pass
```

在 `fe.access.book` 中对 `books` 文档集创建索引

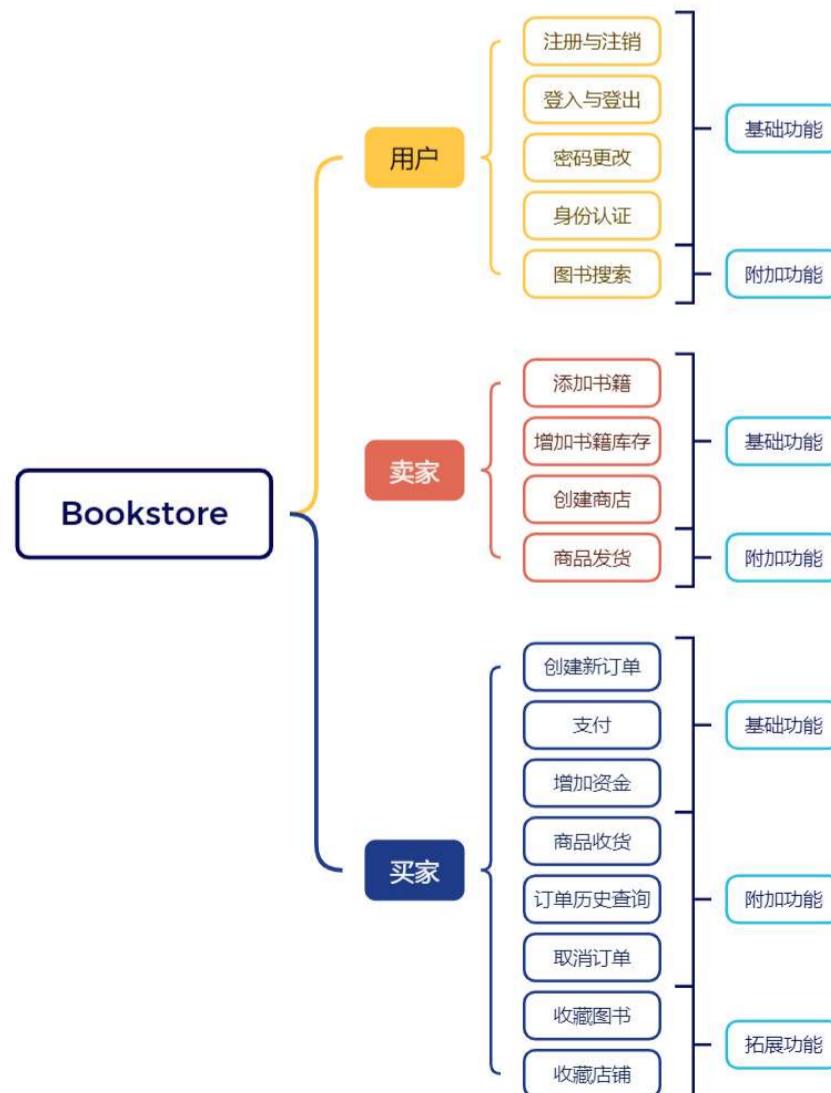
通过上述索引策略，我们不仅显著提升了数据访问速度，还确保了数据管理的高效与可靠，为系统的整体性能优化奠定了坚实的基础。结果最后的实验发现，在添加索引之后，数据库的查询时间有了不错的减少，这都证明了索引的重要作用。我们测量的实验数据如下：



基础功能 (Basic API)

按照不同功能，我们将需要实现的层次分为三个方面：`user`、`seller`、`buyer`，通过对三者的设计，可以构建出平台的基本功能。

需要强调的是，这里和数据库交互所采用的均为MongoDB的API进行调用，因此在功能部分，我们不仅需要考虑实现的逻辑还需要考虑API的改写。思维导图如下：



通过导图我们可以清晰看到Bookstore的整体框架。下面是代码的具体实现方式。

用户

用户注册与注销 (`register`、`unregister`)

`register` 函数:

- 参数:
 - `user_id` : 用户 ID。
 - `password` : 用户密码。
- 返回值:
 - 状态码, 200 表示成功, 528 表示数据库错误。
 - 操作结果, “ok”或错误信息。
- 流程:
 1. 检查数据库中是否已有相同 `user_id` 的用户。
 2. 生成唯一的终端标识符以区别, 格式为 “`terminal_<时间戳>`”。
 3. 使用 `jwt_encode` 函数生成 JWT token, 用于身份验证和信息交换。
 4. 将用户信息 (包括 ID、密码、余额、token、终端) 插入数据库。
 5. 如遇数据库错误, 返回状态码 528 和错误信息。

`unregister` 函数:

- 参数:
 - `user_id` : 要注销的用户 ID。
 - `password` : 用户密码。
- 返回值:
 - 状态码, 200 表示成功, 530 表示异常。
 - 操作结果。
- 流程:
 1. 检查密码是否正确。
 2. 如果密码正确, 则删除用户的注册记录。
 3. 如删除失败或异常, 则返回错误码和信息。

用户登录与登出 (`login`、`logout`)

`login` 函数:

- 参数:
 - `user_id` : 用户 ID。
 - `password` : 用户密码。
 - `terminal` : 终端标识。
- 返回值:
 - 状态码, 200 表示成功。
 - 结果信息。
 - 生成的 token。
- 流程:
 1. 检查密码是否正确。
 2. 如果正确, 生成新的 JWT token, 并更新数据库中的用户记录。
 3. 如果操作失败, 返回错误信息。

`logout` 函数:

- 参数:
 - `user_id` : 用户 ID。
 - `token` : 当前 token。
- 流程:
 1. 检查 token 是否有效。
 2. 如果有效, 则生成一个占位的 dummy token 并更新数据库。
 3. 如果更新失败, 返回错误信息。

用户密码更改 (`change_password`)

`change_password` 函数:

- 参数:
 - `user_id` : 用户 ID。
 - `old_password` : 旧密码。

- `new_password` : 新密码。

- 流程:

1. 检查旧密码是否正确。
2. 如果正确，则更新数据库中的密码和 `token`。
3. 如果更新失败，返回错误信息。

用户认证模块 (`jwt_encode`、`check_token`)

`jwt_encode` 函数:

- 目的: 生成JWT `token`, 用于用户身份验证和信息交换

- 参数:

- `user_id` : 用户 ID。
- `terminal` : 终端标识。

- 流程:

1. 使用用户 `ID` 作为密钥，生成 JWT `token`，包含用户 `ID`、终端和时间戳。
2. 将生成的 `token` 返回。

`check_token` 函数:

- 目的: 验证用户身份。

- 参数:

- `user_id` : 用户 `ID`。
- `token` : 提供的 `token`。

- 流程:

1. 从数据库中获取用户 `token`。
2. 比较数据库中的 `token` 和提供的 `token` 是否匹配。
3. 验证 `token` 的时间戳是否在有效期内。

卖家

添加书籍 (`add_book`)

- 参数:

- `user_id` : 用户 `ID`。
- `store_id` : 商店 `ID`。
- `book_id` : 书籍 `ID`。
- `book_json_str` : 包含书籍信息的 JSON 字符串。
- `stock_level` : 书籍的初始库存数量。

- 返回值:

- 状态码, `200` 表示成功, `528` 表示 MongoDB 错误, `530` 表示未知异常。
- 操作结果，通常为成功消息或错误消息。

- 流程:

1. 调用 `self.user_id_exist(user_id)` 检查用户 `ID` 是否存在。
 - 如果用户不存在，返回 `error.error_non_exist_user_id(user_id)`。
2. 调用 `self.store_id_exist(store_id)` 检查商店 `ID` 是否存在。
 - 如果商店不存在，返回 `error.error_non_exist_store_id(store_id)`。
3. 调用 `self.book_id_exist(store_id, book_id)` 检查该商店是否已经有这本书。
 - 如果书籍已存在，返回 `error.error_exist_book_id(book_id)`。
4. 创建书籍文档 `book_doc`，包括商店 `ID`、书籍 `ID`、书籍信息和库存量。使用 `self.conn['store'].insert_one(book_doc)` 将书籍插入数据库。
5. 异常处理：
 - 如果发生 MongoDB 错误，捕获异常并返回状态码 `528` 和错误信息。
 - 如果发生其他未知异常，捕获并返回状态码 `530` 和错误信息。

添加书籍库存 (`add_stock_level`)

- 参数:

- `user_id` : 用户 `ID`。
- `store_id` : 商店 `ID`。
- `book_id` : 书籍 `ID`。
- `add_stock_level` : 要增加的库存数量。

- 返回值:

- 状态码，`200` 表示成功，`528` 表示 MongoDB 错误，`530` 表示未知异常。
- 操作结果或错误信息。
- 流程：
 1. 检查用户是否存在：如果用户 ID 不存在，返回 `error.error_non_exist_user_id(user_id)`。
 2. 检查商店是否存在：如果商店 ID 不存在，返回 `error.error_non_exist_store_id(store_id)`。
 3. 检查书籍是否存在于商店中：如果书籍 ID 不存在，返回 `error.error_non_exist_book_id(book_id)`。
 4. 使用 `update_one` 方法，根据商店 ID 和书籍 ID，将库存增加指定数量，我们使用MongoDB的API实现，实现代码如下：

```
self.conn['store'].update_one(
    {'store_id': store_id, 'book_id': book_id},
    {'$inc': {'stock_level': add_stock_level}},
)
```

5. 异常处理：

- 捕获 MongoDB 错误并返回状态码 `528` 和错误信息。
- 捕获未知异常并返回状态码 `530` 和错误信息。

创建商店 (`create_store`)

- 参数：
 - `user_id`：用户 ID。
 - `store_id`：新创建的商店 ID。
- 返回值：
 - 状态码，`200` 表示成功，`528` 表示 MongoDB 错误，`530` 表示未知异常。
 - 操作结果或错误信息。
- 流程：
 1. 检查用户是否存在：如果用户 ID 不存在，返回 `error.error_non_exist_user_id(user_id)`。
 2. 检查商店 ID 是否已存在：如果商店 ID 已存在，返回 `error.error_exist_store_id(store_id)`。
 3. 插入商店信息到数据库：创建商店文档 `user_store_doc`，包含商店 ID 和用户 ID，并插入 `user_store` 集合，实现代码如下：

```
self.conn['user_store'].insert_one(user_store_doc)
```

4. 异常处理：

- 捕获 MongoDB 错误并返回状态码 `528` 和错误信息。
- 捕获未知异常并返回状态码 `530` 和错误信息。

买家

创建新订单 (`new_order`)

- 目的：从用户和商店订购书本。
- 参数：
 - `user_id`：用户 ID。
 - `store_id`：商店 ID。
 - `id_and_count`：书本的 ID 和数量。
- 返回值：
 - 状态码。`200` 表示成功，其他值表示错误。
 - 操作结果，成功或错误消息。
 - 订单 ID。成功创建订单时返回订单 ID，否则为空字符串。
- 流程：
 1. 初始化订单 ID：定义一个空字符串 `order_id`，用于后续存储生成的订单 ID。
 2. 验证用户和商店的存在：使用函数 `self.user_id_exist` 和 `self.store_id_exist` 来验证用户 ID 和商店 ID 的存在。如果这些 ID 不存在，函数将返回相应的错误消息和空的 `order_id`。
 3. 生成唯一订单 ID：使用 `uuid.uuid1()` 生成基于时间的唯一标识符，并与用户 ID 及商店 ID 结合形成订单唯一 ID。
 4. 遍历书本与数量：遍历 `id_and_count` 数组中的每个元组，对每本书进行处理：
 - 查询商店中是否有对应的书本，以及库存状态。
 - 如果书本不存在或库存不足，返回相应的错误消息和空 `order_id`。
 - 如果书本存在且库存足够，通过更新操作减少库存，并计算该书本的总价格。
 - 将每本书的信息添加到订单详情列表 `order_details`。
 5. 插入订单详情和订单信息：如果订单详情列表不为空，则向 `new_order_detail` 集合插入详情，并在 `new_order` 集合中插入订单基本信息。
 6. 设置自动取消订单定时器：使用 `threading.Timer` 设置一个300秒后执行的定时器，自动取消订单。

- 插入历史订单：将订单状态设置为"pending"，插入 `order_history` 和 `order_history_detail` 集合，以便跟踪订单的历史状态。
- 异常处理：捕获并处理 `pymongo.errors.PyMongoError` 和其他异常，返回错误代码和消息。

支付 (payment)

- 目的：用户付款支付他们的订单。
- 参数：
 - `user_id`：用户ID。
 - `password`：用户密码。
 - `order_id`：订单ID。
- 返回值
 - 状态码。200 表示成功，其他值表示有错误。
 - 操作的结果，成功或错误。
- 流程：
 - 查询订单：在 `new_order` 集合中查找给定ID的订单内容。
 - 检验用户匹配及权限：验证找到的订单是否属于执行支付操作的用户。
 - 验证用户存在及密码正确：在 `user` 集合中查找用户信息，并验证密码。
 - 检查订单是否超时：从 `order_history` 集合中查找订单状态，确认未过期可以进行支付。
 - 检查余额：计算订单总金额并与用户余额比较。
 - 更新买家和卖家余额：如果余额充足，从买家账户扣款，相应增加卖家账户余额。
 - 更新订单状态：将订单从新订单删除，更新历史订单状态为已支付。
 - 异常处理：处理 `pymongo.errors.PyMongoError` 和其他异常，记录错误并返回。

添加资金 (add_funds)

- 参数：
 - `user_id`：用户ID。
 - `password`：用户密码。
 - `add_value`：要添加到用户余额的金额。
- 返回值：
 - 状态码，200 表示成功，528 表示PyMongo错误，530 表示其他异常。
 - 操作结果，“ok”或错误信息。
- 流程：
 - 函数通过用户ID在数据库中寻找用户，并验证密码，如果用户不存在或密码不正确，则返回授权失败错误。
 - 如果用户验证成功，使用 `update_one` 方法增加用户的余额。
 - 检查数据库更新操作是否成功执行，如果没有找到对应用户或无法更新，则返回不存在用户ID的错误。
 - 如果所有步骤顺利完成，则返回状态码200和消息“ok”。
 - 捕获并处理PyMongo错误和其他异常，分别返回相应的错误代码和异常信息。

附加功能 (Advanced API)

发货与收货

商家发货 (ship_order)

- 参数：
 - `user_id`：用户 ID。
 - `store_id`：商店 ID。
 - `order_id`：要发货的订单 ID。
- 返回值：
 - 状态码，200 表示成功，528 表示 MongoDB 错误，400 表示业务逻辑错误。
 - 操作结果或错误信息。
- 流程：
 - 检查用户是否存在：如果用户 ID 不存在，返回 `error.error_non_exist_user_id(user_id)`。
 - 检查商店是否存在：如果商店 ID 不存在，返回 `error.error_non_exist_store_id(store_id)`。
 - 检查订单是否存在且已付款：从 `order_history` 集合中查找订单，实现代码如下：

```
order = self.conn['order_history'].find_one({'order_id': order_id})
```

- 如果订单不存在，返回状态码 400 和错误信息 "Invalid order ID"。

- 如果订单状态不是 "paid"，返回状态码 400 和错误信息 "Order is not paid"。
4. 更新订单状态为已发货：

使用 `update_one` 方法将订单状态更新为 "shipped"：

```
self.conn['order_history'].update_one(  
    {'order_id': order_id},  
    {'$set': {'status': 'shipped'}},  
)
```

5. 异常处理：

- 捕获 MongoDB 错误并返回状态码 528 和错误信息。

买家收货 (`receive_order`)

• 参数：

- `user_id`：字符串，用户的 ID。
- `order_id`：字符串，订单的 ID。

• 返回值：

- 状态码，200 表示成功，528 表示 PyMongo 错误，530 表示其他异常。
 - 操作结果的消息。
- 流程：
- 查询指定订单，检查是否存在，如果不存在，返回错误消息。
 - 检查订单是否属于当前用户，以及订单的状态是否为已发货，如果不是则返回相应错误。
 - 更新订单状态为已收货，如果更新失败，返回错误消息。
 - 捕获并处理 PyMongo 错误和其他异常，返回相应的错误代码和异常信息。

图书检索

搜索图书 (`search_book` 函数)

• 参数：

- `title`：图书标题。
- `content`：图书内容。
- `tag`：图书标签。
- `store_id`：商店 ID。

• 流程：

- 根据提供的参数构建查询条件。
- 如果指定了 `store_id`，则查询该商店的图书 ID，并将其用于过滤。
- 在 `books` 集合中执行查询，返回符合条件的图书列表。
- 如果查询失败或没有结果，返回错误信息。

查询、取消订单

获取订单历史 (`get_order_history` 函数)

• 参数：

- `user_id`：用户 ID。

• 返回值：

- 状态码，200 表示成功，528 表示 PyMongo 错误，530 表示其他异常。
- 操作结果的消息。
- 包含用户所有订单的字典列表。

• 流程：

- 查询用户的订单历史，使用“aggregate”进行数据关联查询。
- 如果用户没有订单或用户 ID 不存在，则返回相应的错误消息和空订单列表。
- 构造每个订单及其详细信息的列表。
- 返回成功的状态码和订单列表。
- 捕获并处理 PyMongo 错误和其他异常，返回相应的错误代码和异常信息。

取消订单 (`cancel_order` 函数)

• 参数：

- `user_id`：用户 ID。
- `order_id`：订单 ID。

- **返回值:**
 - 状态码，200表示成功，528表示PyMongo错误，530表示其他异常。
 - 操作结果。
- **流程:**
 1. 查询具体的订单，如果找不到指定的订单ID，返回无效订单ID的错误。
 2. 检查订单用户ID是否与传入的用户ID匹配，如果不匹配，返回授权失败错误。
 3. 删除订单，如果没有成功删除，返回错误消息。
 4. 删除订单详情，使用“`delete_many`”确保关联数据的完整删除。
 5. 更新订单状态为“`cancelled`”，如果没有成功更新，返回错误消息。
 6. 捕获并处理PyMongo错误和其他异常，返回相应的错误代码和异常信息。

拓展功能 (Extended API)

在实验要求之外，我们还对项目进行了拓展实现。为了让我们的网上商城更加丰富真实，我们添加了书本和商家的收藏功能，这就让项目更加真实完善。

收藏书本

收藏图书（`collect_book` 函数）

- **参数:**
 - `user_id`：用户ID。
 - `book_id`：书籍ID。
- **返回值:**
 - 状态码。200 表示操作成功，528 代表数据库错误，530 代表其他异常。
 - 操作结果，成功消息或错误消息。
- **流程:**
 1. **检查用户是否存在:** 使用 `user_id` 在数据库中查询用户。如果用户不存在，返回特定的错误消息 `"non exist user id"` 和状态码。
 2. **检查书籍是否已在收藏中:** 如果给定的 `book_id` 已包含在用户的收藏列表中，返回状态码 200 和消息 `"book already in collection"`。
 3. **将书籍添加到收藏中:** 如果书籍不在收藏中，则使用 `$addToSet` 操作将书籍ID添加到用户的 `collection` 字段。这可以避免重复添加相同的书籍。
 4. **异常处理:** 捕获任何由数据库操作引发的异常，并根据异常类型返回相应的状态码和错误消息。

取消收藏图书（`uncollect_book` 函数）

- **参数:**
 - `user_id`：用户ID。
 - `book_id`：书籍ID。
- **返回值:**
 - 状态码，200 表示操作成功，528 代表数据库错误，530 代表其他异常。
 - 操作结果。
- **流程:**
 1. **检查用户是否存在:** 使用 `user_id` 在数据库中查询用户。如果用户不存在，返回特定的错误消息和状态码。
 2. **从收藏中移除书籍:** 使用 `$pull` 操作从用户的 `collection` 字段中移除指定的 `book_id`。
 3. **异常处理:** 捕获并处理由数据库操作引发的任何异常，返回相应的状态码和错误消息。

获取用户收藏的图书（`get_collection` 函数）

- **参数:**
 - `user_id`：用户ID。
- **返回值:**
 - 状态码，200 表示操作成功，528 代表数据库错误，530 代表其他异常。
 - 操作结果，为 `collection` 列表或错误消息。
- **流程:**
 1. **检查用户是否存在:** 使用 `user_id` 在数据库中查询用户。如果用户不存在，返回特定的错误消息和状态码。
 2. **获取用户的收藏列表:** 如果用户存在，则获取其 `collection` 字段的内容，返回给用户。
 3. **异常处理:** 捕获并处理由数据库操作引发的任何异常，返回相应的状态码和错误消息。

收藏店铺

收藏店铺（`collect_store` 函数）

- **参数:**
 - `user_id`：用户ID。

- `store_id`: 店铺ID。
- **返回值:**
 - 状态码, 200 表示操作成功, 528 代表数据库错误, 530 代表其他异常。
 - 操作结果, 成功消息或错误消息。
- **流程:**
 - **检查用户是否存在:** 使用 `user_id` 在数据库中查询用户。如果用户不存在, 返回特定的错误消息和状态码。
 - **检查店铺是否已在收藏中:** 如果 `store_id` 已包含在用户的 `store_collection` 字段中, 返回状态码 200 和消息 "store already in collection"。
 - **将店铺添加到收藏中:** 如果店铺不在收藏中, 则使用 `$addToSet` 操作将店铺ID添加到 `store_collection` 字段。
 - **异常处理:** 捕获并处理由数据库操作引发的任何异常, 返回相应的状态码和错误消息。

取消收藏店铺 (`uncollect_store` 函数)

- **参数:**
 - `user_id`: 用户ID。
 - `store_id`: 店铺ID。
- **返回值:**
 - 状态码, 200 表示操作成功, 528 代表数据库错误, 530 代表其他异常。
 - 操作结果。
- **流程:**
 - **检查用户是否存在:** 使用 `user_id` 在数据库中查询用户。如果用户不存在, 返回特定的错误消息和状态码。
 - **从收藏中移除店铺:** 使用 `$pull` 操作从用户的 `store_collection` 字段中移除指定的 `store_id`。
 - **异常处理:** 捕获并处理由数据库操作引发的任何异常, 返回相应的状态码和错误消息。

获取用户收藏的店铺 (`get_store_collection` 函数)

- **参数:**
 - `user_id`: 用户ID。
- **返回值:**
 - 状态码。200 表示操作成功, 528 代表数据库错误, 530 代表其他异常。
 - 操作的结果, 为 `store_collection` 列表或错误消息。
- **流程:**
 - **检查用户是否存在:** 使用 `user_id` 在数据库中查询用户。如果用户不存在, 返回特定的错误消息和状态码。
 - **获取用户的店铺收藏列表:** 如果用户存在, 则获取其 `store_collection` 字段的内容, 返回给用户。
 - **异常处理:** 捕获并处理由数据

接口

前后端接口交互逻辑

前端测试代码 `fe/test` 中调用 `fe/access` 中相应测试功能的接口, `fe/access` 产生相应的请求, 然后后端 `be/view` 中的接口会识别和接收这些请求, 并发送给 `be/model` 中的相应接口函数, 这些函数会对数据库进行操作并返回结果。

以基本功能中的“用户注册”功能为例, 在 `fe/test` 相应功能测试代码中会调用 `fe/access/auth.py` 中的 `register()` 函数接口, 该函数会将用户设置的用户ID和密码塞进请求体并发起一个POST请求, 然后 `be/view/auth.py` 中的 `register()` 函数会根据路径 “/register” 识别并接收前端的注册请求, 再将请求体中的用户ID和密码解析出来并发给 `be/model/user.py` 中的 `register()` 函数, 由它来执行注册过程中的数据库操作。

其他相应功能的前后端接口交互逻辑也类似。

后端接口

be/model

实现后端功能逻辑的主体, 我们在原先的结构上进行了相应的调整和修改, 其中各个模块功能具体如下:

1. `buyer.py`
定义买家 (`Buyer`) 类, 处理买家相关的业务逻辑, 如下单、支付、查询订单历史、收藏图书和店铺等。
 - 继承自 `db_conn.py` 中的 `DBConn` 类, 使用数据库连接。
 - 使用 `error.py` 中定义的错误代码和消息处理错误。
 - 与 `database.py` 中的 `MongoDB_client` 类交互, 获取数据库连接。
 -
2. `database.py`
定义 `MongoDB_client` 类, 负责创建和维护MongoDB数据库连接。
 - 被 `db_conn.py` 中的 `DBConn` 类使用, 以获取数据库连接。
 - 负责初始化数据库中的文档集合, 并创建索引以优化查询性能。

3. `db_conn.py`

定义 `DBConn` 类，提供数据库连接和一些基础的数据库查询方法，如检查用户ID、书籍ID和店铺ID是否存在。

- 被 `buyer.py` 和 `seller.py` 中的类继承，以使用数据库连接和查询方法。

4. `error.py`

定义错误代码和相应的错误消息，用于在整个应用中统一错误处理。

- 被 `buyer.py` 和 `seller.py` 中的类使用，以返回标准化的错误响应。

5. `seller.py`

定义了卖家（Seller）类，处理卖家相关的业务逻辑，如添加书籍、增加库存、创建店铺和发货。

- 继承自 `db_conn.py` 中的 `DBConn` 类，使用数据库连接。
- 使用 `error.py` 中定义的错误代码和消息处理错误。
- 与 `database.py` 中的 `MongoDB_client` 类交互，获取数据库连接。

6. `user.py`

定义用户（User）类，处理用户相关的业务逻辑，如注册、登录、登出、密码更改和图书搜索。

- 继承自 `db_conn.py` 中的 `DBConn` 类，使用数据库连接。
- 使用 `error.py` 中定义的错误代码和消息处理错误。
- 使用JWT（JSON Web Tokens）进行用户认证。

具体后端功能接口实现参照之前基本功能和扩展功能的描述，此处不再赘述。

`be/view`

识别和接收前端 `fe/access` 产生的请求，根据不同的请求发送给 `be/model` 中的相应接口函数执行功能。

前端接口

`fe/test`

测试书店实现的功能，包含基本功能测试代码和补充的扩展功能测试代码。

具体前端测试接口实现参照后文中的测试部分内容。

`fe/access`

不同的功能测试产生相应请求，由后端 `be/view` 中的接口识别并接收。

`fe/bench`

用于效率测试。

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。

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：用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，模拟订单过期后的取消尝试，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：
 1. 调用 `get_order_history` 方法查询订单历史。
 2. 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时订单历史查询的反馈。
 - 操作步骤：
 1. 人为修改买家ID以制造“不存在的用户”情形。
 2. 查询该用户的订单历史。
 3. 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：
 1. 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。
 2. 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。
 3. 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。
 4. 收藏一个商店，然后验证状态码为200，确认收藏成功。
 5. 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。
 6. 取消收藏的商店，验证状态码为200，确保取消成功。

执行结果

导入数据

创建本地 MongoDB 数据库，执行 `load_data.py` 将 `bookstore/data/book.db` 中的内容存入本地数据库，使用 `mongosh` 检查是否导入成功：

```
test> use bookstore
switched to db bookstore
bookstore> show collections
books
new_order
new_order_detail
order_history
order_history_detail
store
user
user_store
```

bookstore中包含的各个文档集合的信息

接下来我们查询books文档集合，确认书籍信息是否成功导入到本地的MongoDB数据库中：

功能测试

基于给出的测试代码脚本，我们对代码的执行情况进行测试，测试结果如下：

```

fe/test/test_add_book.py::TestAddBook::test_ok PASSED
fe/test/test_add_book.py::TestAddBook::test_error_non_exist_store_id PASSED
fe/test/test_add_book.py::TestAddBook::test_error_exist_book_id PASSED
fe/test/test_add_book.py::TestAddBook::test_error_non_exist_user_id PASSED
fe/test/test_add_funds.py::TestAddFunds::test_ok PASSED
fe/test/test_add_funds.py::TestAddFunds::test_error_user_id PASSED
fe/test/test_add_funds.py::TestAddFunds::test_error_password PASSED
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_user_id PASSED
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_store_id PASSED
fe/test/test_add_stock_level.py::TestAddStockLevel::test_error_book_id PASSED
fe/test/test_add_stock_level.py::TestAddStockLevel::test_ok PASSED
fe/test/test_bench.py::test_bench PASSED
fe/test/test_cancel_order.py::TestCancelOrder::test_ok PASSED
fe/test/test_cancel_order.py::TestCancelOrder::test_wrong_user_id PASSED
fe/test/test_cancel_order.py::TestCancelOrder::test_non_exist_order_id PASSED
fe/test/test_cancel_order.py::TestCancelOrder::test_repeat_cancel PASSED
fe/test/test_cancel_order.py::TestCancelOrder::test_cancel_paid_order PASSED
fe/test/test_cancel_order.py::TestCancelOrder::test_cancel_long_time_order PASSED
fe/test/test_collection.py::TestCollection::test_ok PASSED
fe/test/test_create_store.py::TestCreateStore::test_ok PASSED
fe/test/test_get_order_history.py::TestGetOrderHistory::test_error_exist_store_id PASSED
fe/test/test_get_order_history.py::TestGetOrderHistory::test_ok PASSED
fe/test/test_login.py::TestLogin::test_ok PASSED
fe/test/test_login.py::TestLogin::test_error_password PASSED
fe/test/test_new_order.py::TestNewOrder::test_non_exist_book_id PASSED
fe/test/test_new_order.py::TestNewOrder::test_low_stock_level PASSED
fe/test/test_new_order.py::TestNewOrder::test_ok PASSED
fe/test/test_new_order.py::TestNewOrder::test_non_exist_user_id PASSED
fe/test/test_new_order.py::TestNewOrder::test_non_exist_store_id PASSED
fe/test/test_password.py::TestPassword::test_error_password PASSED
fe/test/test_password.py::TestPassword::test_error_user_id PASSED
fe/test/test_payment.py::TestPayment::test_ok PASSED
fe/test/test_payment.py::TestPayment::test_authorization_error PASSED
fe/test/test_payment.py::TestPayment::test_not_suff_funds PASSED
fe/test/test_payment.py::TestPayment::test_repeat_pay PASSED
fe/test/test_register.py::TestRegister::test_register_ok PASSED
fe/test/test_register.py::TestRegister::test_unregister_ok PASSED
fe/test/test_register.py::TestRegister::test_unregister_error_authorization PASSED
fe/test/test_register.py::TestRegister::test_unregister_error_exist_user_id PASSED
fe/test/test_search.py::TestSearch::test_search_global_results PASSED
fe/test/test_search.py::TestSearch::test_search_in_store PASSED
fe/test/test_search.py::TestSearch::test_search_not_exist_store_id PASSED
fe/test/test_search.py::TestSearch::test_search_in_store_not_exist PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_ship_ok PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_receive_ok PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_error_store_id PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_error_order_id PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_error_seller_id PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_error_buyer_id PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_ship_not_pay PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_receive_not_ship PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_repeat_ship PASSED
fe/test/test_ship_receive.py::TestShipReceive::test_repeat_receive PASSED

```

可以看到我们通过了测试，通过率达到了100%，顺利完成了各项任务。

我们修改了conftest.py文件，以确保在每次测试前正确地向MongoDB导入书籍信息

覆盖率测试

覆盖率测试的结果如下：

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	248	83	86	28	66%
be/model/database.py	21	0	2	0	100%
be/model/db_conn.py	23	0	6	0	100%
be/model/error.py	25	2	0	0	92%
be/model/seller.py	63	15	24	1	82%
be/model/user.py	140	31	42	5	86%
be/serve.py	35	1	2	1	95%
be/view/auth.py	51	0	0	0	100%
be/view/buyer.py	94	0	2	0	100%
be/view/seller.py	39	0	0	0	100%
fe/_init_.py	0	0	0	0	100%
fe/access/_init_.py	0	0	0	0	100%
fe/access/auth.py	36	0	0	0	100%
fe/access/book.py	67	2	10	1	96%
fe/access/buyer.py	90	0	2	0	100%
fe/access/new_buyer.py	8	0	0	0	100%
fe/access/new_seller.py	8	0	0	0	100%
fe/access/seller.py	37	0	0	0	100%
fe/bench/_init_.py	0	0	0	0	100%
fe/bench/run.py	13	0	6	0	100%
fe/bench/session.py	47	1	12	2	95%
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	1	16	1	97%
fe/test/test_add_book.py	37	0	10	0	100%
fe/test/test_add_funds.py	23	0	0	0	100%
fe/test/test_add_stock_level.py	40	0	10	0	100%
fe/test/test_bench.py	6	2	0	0	67%
fe/test/test_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	1868	143	268	42	91%

我们最后的覆盖率达到90%+。这说明我们的代码有效地覆盖绝大多数的情况。剩下一些没有被覆盖的部分，大多是一些不会被执行的代码以及一些异常抛出的情况，为保证代码结构的完整，我们保留了这些代码部分，理论上覆盖率还能更高。

协同开发 & 分工

深度利用Git

本次实验的实现过程中，我们充分利用了Git版本控制系统来促进团队协作。Git不仅帮助我们有效地管理代码版本，还使得多人同时开发变得更加高效和有序。通过创建分支、合并请求和代码审查，我们确保了代码质量并减少了冲突。这种协作方式极大地提高了我们的开发效率，同时也为我们提供了一个学习和分享的平台。

为了避免潜在的代码冲突和维护`main`分支的稳定性，我们采取了分支策略。我们在仓库中创建了`dev`的开发分支，与`main`分支并行存在。这个`dev`分支成为了我们日常开发和功能迭代的主要分支，为团队成员提供了一个安全的环境来进行代码实验和功能开发。最后在`dev`分支代码测试无误之后，我们将其合并到`main`分支上，得到了最终的代码版本。

在代码编写过程中，我们保持了良好的习惯，在每个功能实现后，通过`git`工具将代码变更`commit`到仓库中去。我们`github`仓库的`commit`记录如下所示：

The screenshot shows a list of commits from a GitHub repository. It is organized into three sections by date:

- Commits on Oct 27, 2024:**
 - Update README.md (Verified, 8def5c8)
 - Update README.md. Add store_collection (Verified, 880022d)
 - Implement store collection of user (Verified, 8b22661)
- Commits on Oct 29, 2024:**
 - Add files via upload (Verified, be30728)
 - Add test for feature: get order history (Verified, a6cf158)
 - Add test for feature: search (Verified, c4dfc3c)
 - Add test for feature: ship and receive (Verified, d8858e9)
 - add load_data.py (Verified, d9cff7f)
 - Update README.md (Verified, b7b152d)
- Commits on Oct 30, 2024:**
 - Fix issue#2 (Verified, 4b5305e)
 - Fix issue#5 (Verified, fe29eb8)
 - Update model.database (Verified, f5130ce)
 - Update .gitignore (Verified, 2bdcc1e7)
 - Fix test_cancel_order.py (Verified, 1dcc8b7)
 - Complete: Now test would use the data from MongoDB which is provided ... (Verified, cb8a080)
 - Fix test_cancel_order.py (Verified, 978542f)

同时，为了方便协作开发，我们还积极使用 `PR` 和 `Issue` 功能，来确保开发流程的合规性；并使用 `Tag` 进行版本控制与管理。具体也有一些截图：

The screenshot shows the GitHub Issues page with the following details:

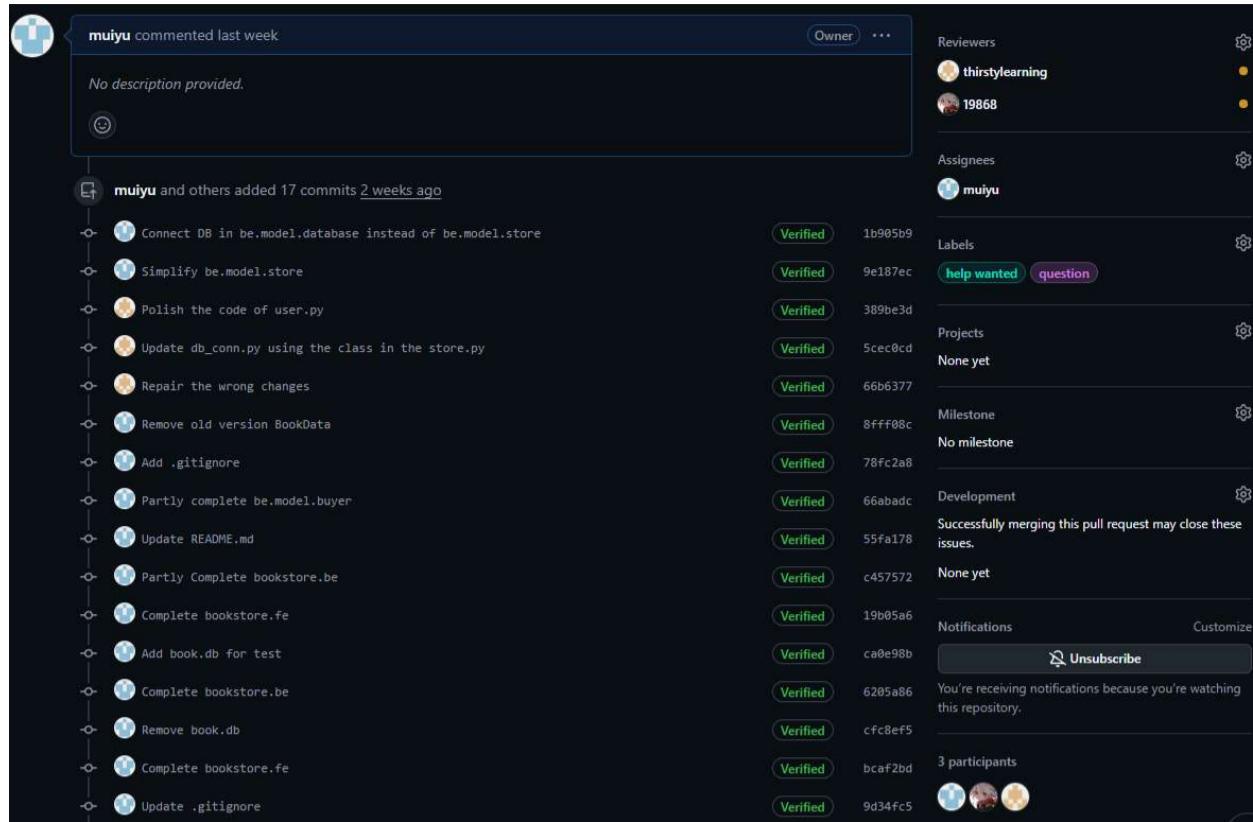
- Issues:** 2 closed issues are listed:
 - #5 by muiyu was closed 2 days ago: `test_get_order_history.py error` (bug)
 - #2 by 19868 was closed 2 days ago: `test_ship_receive.py`
- Filters:** The filter dropdown shows "is:issue is:closed".
- Search:** The search bar contains "is:issue is:closed".
- Labels:** 9 labels are present.
- Milestones:** 0 milestones are present.
- New issue:** A green "New issue" button is visible.

使用 Issue 合作的情况

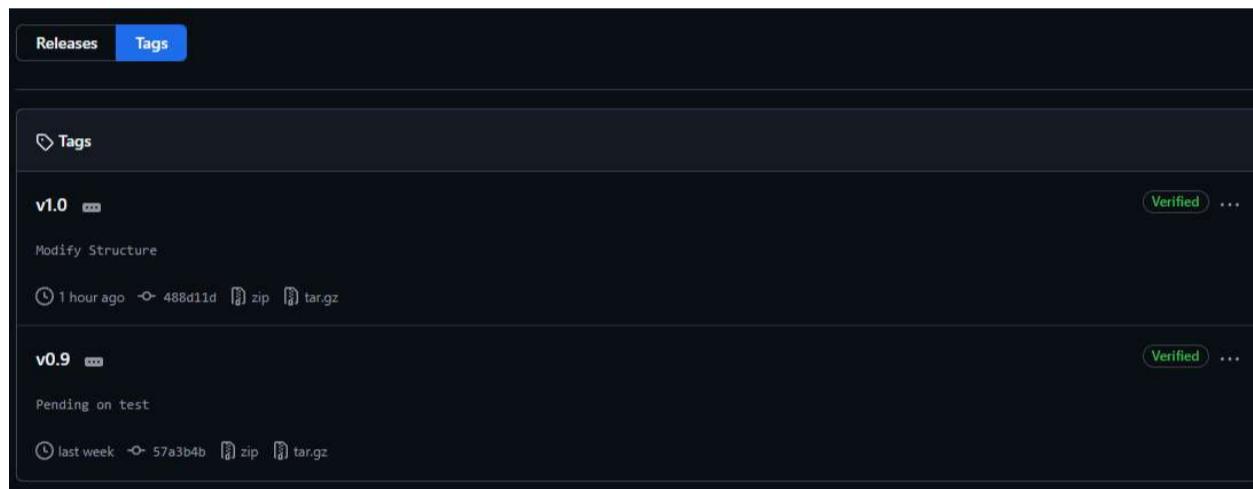
The screenshot shows the GitHub Pull Requests (PR) page with the following details:

- Pulls:** 2 closed pull requests are listed:
 - #6 by muiyu was merged now: `Final Version` (documentation, enhancement)
 - #1 by muiyu was merged last week: `Dev` (help wanted, question)
- Filters:** The filter dropdown shows "is:pr is:closed".
- Search:** The search bar contains "is:pr is:closed".
- Labels:** 9 labels are present.
- Milestones:** 0 milestones are present.
- New pull request:** A green "New pull request" button is visible.

使用 PR 进行合规开发的情况



PR 的详情和参与者



项目 Tag 情况

感兴趣的话。您也可以访问<https://github.com/muiyu/BookStore>获得更多信息。事实上，上面提供了甚至更加详细的数据库 Schema 解说！

分工

在项目的完成过程中，所有成员相互沟通、通力协作，充分利用git工具完成了本项目的所有工作，其中所有人的工作量分布如下：

- 段书文：33.33% 贡献
 - 附加功能开发
 - 图书搜索接口
 - 附加功能测试
 - BUG 修复
 - 报告撰写
- 郭力维：33.33% 贡献
 - 基础功能开发（全部基础接口）
 - 附加功能开发
 - 发货/收货接口
 - 订单操作接口
 - 代码仓库日常维护
 - 版本管理
 - 分支管理
 - PR 与 Issue
 - BUG 修复
- 蔡祺枫：33.33% 贡献
 - 拓展功能开发
 - 收藏接口
 - BUG 修复
 - 报告撰写

总结

在本次实验中，我们实现一个功能丰富网上购书系统。基于Python代码和MongoDB数据库，我们完成了系统的开发，并实现了买家和卖家的交互功能。在基本功能基础上，我们还实现了订单流程管理、图书搜索、收藏图书和卖家等扩展功能，实现了一个丰富的购书系统后端。通过优化数据库和接口性能，我们有效提高了代码测试覆盖率，确保了系统的稳定性和可靠性。结果证明，我们的系统通过了所有的测试，具有不错的性能。

在实验过程中，我们深入学习了后端开发的各个环节，包括MongoDB数据库设计、API实现、性能优化等。我们还学会了使用Git来进行版本控制和团队协作，这不仅我们提高了开发效率，也积累了宝贵的项目管理经验。这次实验让我们对数据库系统的后端开发有了更深入的理解，培养了我们的实践能力和团队协作精神，为今后的软件开发工作打下了坚实的基础。