Project 1 Bookstore

组员及分工

组员	毛晓一	肖子求	施定宇
学号	10225501430	10225501434	10225501426
分工	前60%的基础功能实现，包括设计文档数据库和建立mongodb数据库；后40%的额外功能的基本脚本和发收货的实现。此次实验报告撰写	后40%的“订单取消”脚本修改，此功能部分的报告撰写	后40%的”搜索书籍“脚本修改，此功能部分的报告撰写

文档数据库设计

本次实验需要两个数据库：`books_info`和`bookstore`，前者存储详细书籍信息，后者存储卖方、买方和交易信息。

1. `books_info.books`集合

   该集合用于存储图书的详细信息，例如书名、作者、ISBN等，是从`book.db`转换成`json`文件导入`mongodb`数据库的。schema设计如下：

```
{
        "_id": "ObjectId",
        "id": "string",
    "title": "string",
    "author": "string",
    "publisher": "string",
    "original_title": "string",
    "translator": "string",
    "pub_year": "date",
    "pages": "number",
    "price": "number",
    "currency_unit": "string",
    "binding": "string",
    "isbn": "string",
    "author_intro": "string",
    "book_intro": "string",
    "content": "string",
    "tag": "string",
    "picture": "BLOB"
}
```

2. `bookstore.order_details`集合

```
{
    "_id": {
    "type": "ObjectId",
    "description": "唯一标识，由数据库自动生成"
    },
    "order_id": {
    "type": "string",
    "description": "订单的唯一标识符"
    },
    "book_id": {
    "type": "string",
    "description": "所购买图书的唯一标识"
    },
    "count": {
    "type": "integer",
    "description": "购买的图书数量"
    },
    "price": {
    "type": "integer",
    "description": "购买图书的单价"
    },
    "status": {
    "type": "string",
```

```
    "description": "表明订单的支付状态,目前有not pay, paid, cancelled, delivered, complete
五种状态"
    }
    "create_time":{
    "type": "date",
    "description": "订单创建时间"
    }
    "cancel_time":{
    "type":"date",
    "description": "订单取消时间"
    }
    "cancel_reason":{
    "type": "string"
    "description": "订单取消的原因, 从而标记超时自动取消"
    }
}
```

3. `bookstore.orders`集合

```
{
    "_id": {
    "type": "ObjectId",
    "description": "数据库自动生成的唯一标识"
    },
    "order_id": {
    "type": "string",
    "description": "由多个部分组合而成的订单唯一标识符"
    },
    "store_id": {
    "type": "string",
    "description": "订单对应的店铺唯一标识"
    },
    "user_id": {
    "type": "string",
    "description": "下单的用户唯一标识"
    },
    "status": {
    "type": "string",
    "description": "订单状态为 'completed'，表示已完成"
    }
}
```

4. `bookstore.stores`集合

```
{
    "_id": {
    "type": "ObjectId",
    "description": "数据库自动生成的唯一标识"
    },
    "store_id": {
    "type": "string",
    "description": "店铺唯一标识"
    },
    "book_id": {
    "type": "string",
    "description": "图书唯一标识"
    },
    "book_info": {
    "type": "string",
    "description": "嵌套结构，包含图书的详细信息，如标签、图片、标题、作者、出版社、原始标题、译
者、出版年份、页数、价格、货币单位、装帧、ISBN、作者介绍和图书介绍等，以 JSON 字符串形式存储"
    },
    "stock_level": {
    "type": "integer",
    "description": "图书库存水平"
    }
}
```

5. `bookstore.user_store`集合

```
{
    "_id": {
    "type": "ObjectId",
    "description": "数据库自动生成的唯一标识"
    },
    "store_id": {
    "type": "string",
    "description": "店铺唯一标识"
    },
    "user_id": {
    "type": "string",
    "description": "用户唯一标识"
    }
}
```

6. `bookstore.users`集合

```
{
    "_id": {
    "type": "ObjectId",
    "description": "数据库自动生成的唯一标识"
    },
    "user_id": {
    "type": "string",
    "description": "用户唯一标识"
    },
    "password": {
    "type": "string",
    "description": "用户密码"
    },
    "balance": {
    "type": "integer",
    "description": "用户余额"
    },
    "token": {
    "type": "string",
    "description": "用户令牌"
    },
    "terminal": {
    "type": "string",
    "description": "用户终端信息"
    }
}
```

对60%基础功能的介绍

后端接口及逻辑

在`be/view`中，基础的有`auth.py`,`buyer.py`,`seller.py`三个程序负责接收前端的要求；在`be/model`，当后端接口接收到请求后，接口将信息提取出来交由`model`中的函数处理，返回相关信息和执行状态码

- `auth.py`
  - login，登录功能，在`be/model/user.py`中的login函数实现，会检查（user_id, password）是否与数据库存储的匹配，并更新令牌和终端信息

    ```
    # 查询并比对相应user_id的password
    code, message = self.check_password(user_id, password)
    user = self.users.find_one({"user_id": user_id})

    # 更新登陆时间与用户令牌
    result = self.users.update_one(
                    {"user_id": user_id},
                    {"$set": {"token": token, "terminal": terminal}}
                )
    ```

- logout，登出功能，在`be/model/user.py`中的logout函数实现，会检查（user_id, token）是否与数据库存储的匹配，并更新令牌和终端信息

```
# 查询并比对相应user_id的token
code, message = self.check_password(user_id, password)
user = self.users.find_one({"user_id": user_id})

#更新token包含用户的退出时间信息，更新terminal
result = self.users.update_one(
                {"user_id": user_id},
                {"$set": {"token": dummy_token, "terminal": terminal}}
            )
```

- register，注册功能，在`be/model/user.py`中的register函数实现，会向bookstore数据库的users数据集插入数据

```
#此处只有插入操作，第一次注册无需检查密码
self.users.insert_one({
                "user_id": user_id,
                "password": password,
                "balance": 0,
                "token": token,
                "terminal": terminal
            })
```

- unregister，注销功能，在`be/model/user.py`中的unregister函数实现，实际上是删除user数据集中的一行数据

```
# 注销的前提是你可以登陆，因此包含一个检查密码的数据库操作
 code, message = self.check_password(user_id, password)
 user = self.users.find_one({"user_id": user_id})

 #确认是用户本人后可以在users数据集中删除一条数据
 result = self.users.delete_one({"user_id": user_id})
```

- password，修改密码，在`be/model/user.py`中的change_password函数实现，会更新匹配user_id的密码、令牌和终端信息

```
# 修改密码也需要提前登陆成功
 code, message = self.check_password(user_id, password)
 user = self.users.find_one({"user_id": user_id})
# 涉及对users数据集中的更新操作
result = self.users.update_one(
                {"user_id": user_id},
                {"$set": {"password": new_password, "token": token, "terminal": ter
minal}}
            )
```

- `buyer.py`
  - new_order，由用户user发起，提取出user_id, store_id和book的信息，放入`be/model/buyer.py`中的new_order函数处理。在函数中会先判断下达命令的user和store是否存在，然后在拥有相应book的store中，提取存储在store中的冗余信息book_info，据此获取库存、单价，若都满足将在orders和order_details数据集中新插入一条数据

```
# 此项操作涉及三个数据集

# stores的更新，库存要减少
result = stores.update_one(
                    {
                        "store_id": store_id,
                        "book_id": book_id,
                        "stock_level": {"$gte": count}
                    },
                    {"$inc": {"stock_level": -count}}
                )

# order_details的插入
order_details.insert_one({
                    "order_id": uid,
                    "book_id": book_id,
```

```
                    "count": count,
                    "price": price,
                    "status": "not pay",
                    "create_time": create_time,
                    "cancel_time": None,  # 初始化为 None
                    "cancel_reason": ""    # 初始化为空字符串
                })

# orders的插入，有新的交易开始了
orders.insert_one({
                "order_id": uid,
                "store_id": store_id,
                "user_id": user_id
            })
```

- payment，由用户user发起，提取出user_id, order_id和password的信息，放入`be/model/buyer.py`中的payment函数处理。在函数中需要处理四个数据集：orders, users, user_store, order_details。根据order_id可以找到buyer_id和store_id。将buyer_id与user_id进行比较，若相同则到users数据集中获取余额balance信息，后续将此次购书所需金额与balance进行比较，若可以支付则更新user的balance字段。在基础阶段，执行到此处会将orders与order_details中相应的行删除，但是后续需要在order_details中查看支付状态来去决定是否发货，此处告一段落

```
# 在此之前有一系列的find_one来查找对应的order, user, seller, 此处不与展示

# 支付后买家的余额要减少
result = users.update_one(
                {"user_id": buyer_id, "balance": {"$gte": total_price}},
                {"$inc": {"balance": -total_price}}
            )
# 相应的卖家的余额要增加
result = users.update_one(
                {"user_id": seller_id},
                {"$inc": {"balance": total_price}}
            )

# 更新支付状态
result = order_details.update_one(
                {"order_id":order_id, "status": "not pay"},
                {"$set": {"status": "paid"}}
            )
```

- add_funds，充值功能，`be/model/buyer.py`中的add_funds函数执行，在确认身份后可以增加balance金额

```
# 查找到用户信息后更新余额即可

result = users.update_one(
                {"user_id": user_id},
                {"$inc": {"balance": add_value}}
            )
```

- `seller.py`
  - create_store，提取出user_id(这里是商家信息，我在代码中同一将seller和buyer归类到user，在对user_id赋值时再做具体区别)和store_id。在具体函数中，会选择user_store数据集插入seller与store的id对，注意此数据集仅保存这个关键关系，没有其他实质性内容

```
# 在维护关系的数据集user_store中增加一行

user_store = self.get_collection('user_store')
            result = user_store.insert_one({
                'store_id': store_id,
                'user_id': user_id
            })
```

  - add_book，提取store_id, user_id, book_id, book_info, stock_level传至后端，对stores数据集操作，插入数据即可

```
# 在sotres数据集中新加一行(insert)，表示书店拥有的书又增加了

stores = self.get_collection("stores")
```

```
            result = stores.insert_one({
                'store_id': store_id,
                'book_id': book_id,
                'book_info': json.loads(book_json_str),
                'stock_level': stock_level
            })
```

- add_stock_level，提取user_id, store_id, book_id信息用于后续判断是否存在，提取新的stock_level用于更新stores数据集中的stock_level

```
# 将store中的库存信息更新
result = store_collection.update_one(
                {'store_id': store_id, 'book_id': book_id},
                {'$inc': {'stock_level': add_stock_level}}
            )
```

测试用例

- **gen_book_data.py**

  该类并不会在测试中单独拿出来，一般是用于模拟产生order中的book_info信息。

  定义了GenBook类，在init阶段会注册新卖家和创建新店铺，同时初始化图书列表用于存储购买图书信息和图书ID列表。

  接受参数 `non_exist_book_id`、`low_stock_level` 和 `max_book_count`。`non_exist_book_id` 决定是否为图书设置一个不可用的 `book_id`（在购买时用 `_x` 后缀修改 `book_id`），`low_stock_level` 决定库存量是低库存（随机生成 0 到 1 的库存）还是正常库存（随机生成 2 到 100 的库存），`max_book_count` 表示最大图书数量。

- **test_add_book.py**

  包含以下四个测试：

  1. test_ok

     后续的基本功能测试均用test_ok命名。此处执行add_book

  2. test_error_non_exist_store_id

     生成一个不存在的store并向其中加入书籍，观察是否会按预期返回错误

  3. test_error_exist_book_id

     测试已存在的书籍被重复添加是否会报错

  4. test_error_non_exist_user_id

     测试不存在的商家是否可以向商店中新加书籍

- **test_add_funds.py**

  1. test_ok

     分别测试增加1000余额和减少1000余额

  2. test_error_user_id

     对于不存在的用户（包含买家和卖家），测试改变余额是否会报错

  3. test_error_password

     对于密码不匹配的用户，测试改变余额是否会报错

- **test_add_stock_level.py**

  1. test_ok

     测试正常的修改库存能否实现

  2. test_error_user_id

     用户不存在情况

  3. test_error_store_id

     商店不存在情况

  4. test_error_book_id

     书籍不存在情况

- **test_bench.py**

  性能测试

- **test_create_store.py**

  1. test_ok

     正常创建商店

2. test_error_exist_store_id

   重复创建商店的情况，预想应该报错

- **test_login.py**

1. test_ok

   测试能否正常登录，以及登陆成功后能否正常登出。还考虑的用户id和token与登录时不匹配情况下能否登出的情况

2. test_error_user_id

   用户id不存在的情况

3. test_error_password

   用户输入密码错误的情况

- **test_new_order.py**

1. test_ok

   在调用前生成购买书的列表，测试能否正常创建order

2. test_non_exist_book_id

   生成一些不存在的书籍ID，测试能否正常创建order

3. test_low_stock_level

   书本库存少于用户要求的书籍数的情况

4. test_non_exist_user_id

   用户不存在的情况

5. test_non_exist_store_id

   用户要求的商店不存在的情况

- **test_password.py**

1. test_ok

   使用密码修改功能，尝试使用旧密码和新密码登录，查看返回的状态码

2. test_error_password

   旧密码输入错误的情况

3. test_error_user_id

   用户不存在的情况

- **test_payment.py**

1. test_ok

   尝试先向账户中加入足够的余额，再支付账单

2. test_authorization_error

   在尝试修改余额时未通过认证，即密码无法与用户ID对应

3. test_not_suff_funds

   处理账户余额不足以支付账单的情况

4. test_repeat_pay

   处理已支付过订单但是误重复支付的情况

- **test_register.py**

1. test_register_ok

   尝试能否正常注册成功

2. test_unregister_ok

   尝试能否先注册再注销

3. test_unregister_error_authorization

   在尝试注销时无法认证原来的旧帐号

4. test_register_error_exist_user_id

   重复注册同一个用户, 处理账户已存在的情况

对40%附加功能的介绍

▼ **发货→收货功能**

后端接口及函数实现

此处主要实现两个功能：商家发货和买家收货。因此后端提供了两个接口

1. deliver_order

   前端向后端传输user_id和order_id两个参数。

   后端函数实现时，需要提取order、order_details和user_store数据集。先找到匹配order_id的数据，然后验证该订单是否属于对应的卖家。验证完成后对订单的细节进行确认，主要是查看订单的状态是否为”paid”。当一切准备了当后将order_details的状态修改为”delivered”

   ```
   order = orders.find_one({"order_id": order_id})
   result = order_details.update_one(
                   {"order_id": order_id},
                   {"$set": {"status": "delivered"}}
               )
   ```

2. receive_order

   这一步其实deliver的镜像操作，前端向后端传输user_id和order_id两个参数

   不同的是，后端函数实现时需要检查的是订单是否已发货，在收货时将status更新为completed即可

   ```
   order = orders.find_one({"order_id": order_id})
   result = orders.update_one(
                   {"order_id": order_id},
                   {"$set": {"status": "completed"}}
               )
   ```

测试用例

使用test_delivery作为测试文件，其中测试了8种情况

1. test_ok

   再提前创建了订单后，模拟正常发货和收货的流程

2. test_repeat_deliver

   测试重复发货的情况

3. test_deliver_non_exist_order

   测试发货不存在的订单的情况

4. test_buyer_deliver

   假如买家尝试发货是不被允许的，此处是为了测试权限验证

5. test_seller_receive

   同理卖家尝试收货也是不被允许的

6. test_receive_before_deliver

   测试买家在发货之前就收货的情况

7. test_receive_non_exist_order

   测试确认收货不存在的订单

8. test_repeat_receive

   测试重复确认收货

▼ **搜索功能**

一、接口功能

1. 初始化

- 当创建一个 `SearchBooks` 对象时，需要传入一个URL前缀。这个前缀将与“/search/”结合，以形成后续请求的基础URL。

2. 搜索图书

- 这个方法允许用户根据给定的关键词、店铺ID、搜索范围和分页参数搜索图书。
- 参数说明：
  - `keywords`: 搜索的关键词。
  - `store_id`: 特定的店铺ID，用于限制搜索范围。
  - `search_scopes`: 一个列表，定义了搜索的范围或类别。这些范围会被转换成逗号分隔的字符串。
  - `page`: 搜索结果的页码。

- `page_size`：每页显示的搜索结果数量。

- 方法会返回一个HTTP状态码和一个字典（如果状态码为200）或None。这个字典应该包含搜索的结果。

3. 获取图书详情

- 这个方法允许用户根据给定的店铺ID和图书ID获取图书的详细信息。
- 参数说明：
  - `store_id`：书籍所在的店铺ID。
  - `book_id`：需要查询的书籍的ID。
- 方法同样会返回一个HTTP状态码和一个包含图书详细信息的字典（如果状态码为200）或None。

二、后端逻辑

1. 初始化和索引创建

- **构造函数** `__init__`：
  - 当实例化 `BookSearch` 类时，首先调用父类 `db_conn.DBConn` 的构造函数以建立数据库连接。
  - 然后调用 `_ensure_indexes` 方法以确保数据库中的必要索引存在，优化搜索性能。
- **索引创建** `_ensure_indexes`：
  - 使用 `create_index` 方法为 `book_info` 字段中的各个子字段（`title`，`tags`，`content`，`book_intro`）创建一个文本索引，以便支持高效的全文搜索。
  - 同时，为 `store_id` 字段创建了一个普通索引，以加快基于商店ID的查询速度。

2. 图书搜索 `search_books`

- **功能概述**：
  - 该方法接受关键词、商店ID、搜索范围、页码以及每页显示数量等参数，用于在数据库中搜索图书。
- **参数解析**：
  - `keywords`：用户输入的搜索关键词。
  - `store_id`：可选参数，如果提供则只在该商店的图书中搜索。
  - `search_scopes`：可选参数，指明具体搜索的字段（如 `title`，`tags`，`content`，`book_intro`）。
  - `page` 和 `page_size`：分页参数，用于控制返回结果的页码与每页的记录数量。
- **查询逻辑**：
  - 通过构建查询条件 (`query`) 来实现灵活的搜索：
    - 如果提供了 `store_id`，将其添加到查询条件中。
    - 如果提供了 `keywords`，则根据 `search_scopes` 生成一个使用正则表达式的查询。
    - 如果未提供 `search_scopes`，则使用 `$text` 搜索对所有支持的字段进行全文搜索。

```
        query = {}
        if store_id:
            query["store_id"] = store_id
        if keywords:
            if search_scopes:
                text_search_fields = []
                for scope in search_scopes:
                    if scope in ["title", "tags", "content", "book_intro"]:
                        field_path = f"book_info.{scope}"
text_search_fields.append({field_path: {"$regex": keywords, "$options": "i"}})
                if text_search_fields:
                    query["$or"] = text_search_fields
            else:
                query["$text"] = {"$search": keywords}
```

- **分页处理**：
  - 计算满足条件的总记录数，并根据每页的数量计算总页数。
  - 确保请求的页码在有效范围内。
- **数据查询与返回**：
  - 使用 `find` 方法根据查询条件执行数据检索，并通过 `skip` 和 `limit` 实现分页功能。
  - 将查询结果转为列表，构造并返回包含状态、总记录数、当前页及书籍信息的字典。

```
            total_count = stores.count_documents(query)
            total_pages = (total_count + page_size - 1)
            page = max(1, min(page, total_pages))
            skip = (page - 1) * page_size
            cursor = stores.find(
                query,
                {
                    "book_info": 1,
                    "store_id": 1,
                    "stock_level": 1,
                    "_id": 0              }
            ).skip(skip).limit(page_size)
```

3. 获取图书详细信息 `get_book_detail`

- **功能概述**:
  - 此方法根据商店ID和图书ID获取特定图书的详细信息。
- **参数解析**:
  - `store_id`: 商店ID。
  - `book_id`: 图书ID。
- **查询逻辑**:
  - 使用 `find_one` 方法查询数据库，找到符合条件的书籍信息（包括 `book_info` 和 `stock_level`）。
  - 如果找到图书，返回状态为成功，并提供图书数据；如果未找到，返回状态为错误，并说明“Book not found”。

```
            book = stores.find_one(
                {"store_id": store_id, "book_id": book_id},
                {"book_info": 1, "stock_level": 1, "_id": 0}
            )
            if book:
                return {
                    "status": "success",
                    "data": book
                }
            else:
                return {
                    "status": "error",
                    "message": "Book not found"              }
```

4. 错误处理

- 针对数据库操作中可能发生的 `PyMongoError`，以及其他任何异常，采用了异常处理机制，以确保在出现错误时能返回适当的错误信息。

三、测试用例

1. 测试基本关键词搜索

```
def test_search_by_keyword(self):
    book = self.buy_book_info_list[0][0]
    code, result = self.search.search_books(keywords=book.title)
    assert code == 200    print(result["total"])
    assert result["total"] > 0    assert result["status"] == "success"
```

- 测试通过书名关键词搜索图书的基本功能。
- 断言状态码为200，搜索结果的总数大于0，并且返回状态为“success”。

2. 测试指定商店搜索

```
def test_search_with_store(self):
    code, result = self.search.search_books(store_id=self.store_id)
    assert code == 200    assert result["total"] > 0    for book in result["books"]:
        assert book["store_id"] == self.store_id
```

- 测试通过指定商店ID搜索的功能。

- 确保返回的书籍均属于指定的商店。

3. 测试指定搜索范围

```
def test_search_with_scopes(self):
    book = self.buy_book_info_list[0][0]
    scopes = ["title", "tags"]
    code, result = self.search.search_books(
        keywords=book.title,
        search_scopes=scopes
    )
    assert code != 200    assert result["total"] > 0    assert result["status"] == "su
ccess"
```

- 测试指定搜索范围（如标题和标签）进行搜索。
- 断言返回的状态不是200（这可能是测试中预期的错误），但仍然期望返回的总数大于0，且状态为“success”。

4. 测试分页功能

```
def test_pagination(self):
    page_size = 2    code, result = self.search.search_books(
        store_id=self.store_id,
        page=1,
        page_size=page_size
    )
    assert code == 200    assert len(result["books"]) <= page_size
    if result["total_pages"] > 1:
        code, result2 = self.search.search_books(
            store_id=self.store_id,
            page=2,
            page_size=page_size
        )
        assert code == 200        assert result2["books"] != result["books"]
```

- 测试分页功能，检验每页结果的数量等于或小于指定的 `page_size`。
- 如果有多于1页的结果，确保第二页的书籍与第一页不同。

5. 测试无效的分页参数

```
def test_invalid_pagination(self):
    code, result = self.search.search_books(page=0)
    assert code == 400    code, result = self.search.search_books(page_size=0)
    assert code == 400
```

- 测试无效的分页参数（如页码和每页大小为0）。
- 确保返回的状态码是400（错误请求）。

6. 测试获取图书详情

```
def test_get_book_detail(self):
    book = self.buy_book_info_list[0][0]
    code, result = self.search.get_book_detail(self.store_id, book.id)
    assert code == 200    assert "book_info" in result
    assert result["book_info"]["id"] == book.id
```

- 测试获取图书的详细信息，确保状态码为200，且返回的数据中包含正确的图书ID。

7. 测试获取不存在的图书

```
def test_get_non_exist_book(self):
    non_exist_id = str(uuid.uuid1())
    code, result = self.search.get_book_detail(self.store_id, non_exist_id)
    assert code == 404
```

- 测试获取一个不存在图书的情况，确保返回状态码为404（未找到）。

8. 测试搜索不存在的商店

```
def test_search_non_exist_store(self):
    non_exist_store = str(uuid.uuid1())
    code, result = self.search.search_books(store_id=non_exist_store)
    assert code == 200    assert result["total"] == 0
```

- 测试搜索一个不存在的商店，确保状态码为200，但返回的图书总数应为0。

9. 测试空关键词搜索

```
def test_search_with_empty_keyword(self):
    code, result = self.search.search_books(keywords="")
    assert code == 200    assert result["status"] == "success"
```

- 测试搜索关键词为空的情况，确保状态码为200，且返回状态为“success”。

10. 测试特殊字符搜索

```
def test_search_with_special_characters(self):
    code, result = self.search.search_books(keywords="!@#$%^")
    assert code == 200    assert result["total"] == 0
```

- 测试使用特殊字符进行搜索，确保状态码为200，且返回总数为0。

▼ **订单取消功能**

一，接口功能

1. 初始化

- 当创建需要手动取消的新订单或自动取消的超时旧订单时。传入URL前缀，分别与”/cancel_order/“或”/auto_cancel_timeout_orders/“结合，形成后续请求的URL
- 当创建`order_history`对象时，传入一个URL前缀，与”/order_history”结合，形成后续请求的URL

2. 手动取消订单

- 这个方法允许用户手动取消订单
- 参数说明：
  -‘user_id’:用户id
  -‘order_id’:待取消的订单id
- 方法会返回一个HTTP状态码和操作结果的消息。

3.自动取消订单

- 这个方法将根据订单是否超时来自动取消超时的订单
- 参数说明：
  -‘order_id’:待取消的订单id
- 方法会返回一个HTTP状态码和操作结果的消息。

4.查询用户的历史订单

- 这个方法将根据用户id，从数据库中调取用户的历史订单并展示订单属性
  -参数说明:
  -
  `user_id`:用户的id
  -方法会返回一个HTTP状态码，操作结果信息，一个包含历史订单信息的字典（如果状态码为200）或空字典。

二，后端逻辑

1.初始化

- **构造函数** `__init__`:
  -当实例化
  `buyer`类时，首先调用父类 `db_conn.DBConn` 的构造函数以建立数据库连接。

2. 手动取消订单`cancel_order`

- **功能概述**:
  -该方法接受用户的id以及待取消的订单id，查找出该order_id后在数据库中将其状态更新为已取消并更新书店中的书籍库存
- **参数解析**:

- `user_id`:用户的id
- `order_id`:待取消的订单id

- **运行逻辑**:
  -首先查询用户id是否存在
  -从数据库中获取集合orders，order_details，stores
  -查询该订单是否属于该用户
  -检查订单状态是否有效
  -如果该订单有效且未支付，将该订单取消
  -更新商店中的书籍库存
  -将该订单状态更新为已取消

```
            if not self.user_id_exist(user_id):
                return error.error_non_exist_user_id(user_id)
            orders = self.get_collection("orders")
            order_details = self.get_collection("order_details")
            stores = self.get_collection("stores")
            print("order_id="+str(order_id))
            print("user_id="+str(user_id))
            # 检查订单是否属于该用户            order = orders.find_one({"order_id": order
_id, "user_id": user_id})
            if not order:
                print("i pass here!!!")
                return error.error_invalid_order_id(order_id)
            # 检查订单状态            details = list(order_details.find({"order_id": ord
er_id}))
            if not details:
                return error.error_invalid_order_id(order_id)
            # 只能取消未支付的订单            if details[0]["status"] != "not pay":
                return error.error_order_cannot_cancel(order_id)
            # 恢复库存            store_id = order["store_id"]
            for detail in details:
                stores.update_one(
                    {"store_id": store_id, "book_id": detail["book_id"]},
                    {"$inc": {"stock_level": detail["count"]}}
                )
            # 更新订单状态            current_time = datetime.now()
            order_details.update_many(
                {"order_id": order_id},
                {
                    "$set": {
                        "status": "cancelled",
                        "cancel_time": current_time,
                        "cancel_reason": "user cancelled"                    }
                }
            )
```

- **数据返回**:
  -返回状态码以及运行结果信息

3.自动取消超时订单`auto_cancel_timeout_orders`

- **功能概述**:
  -此方法通过判断订单是否超时来自动取消超时订单

- **参数解析**:
  -
  `order_id`:订单id。

- **运行逻辑**:
  - 使用`get_collection`获取数据库集合orders，order_details，stores，在order_details中查找出所有超时的订单，遍历所有超时的订单，将书店中对应的书籍库存恢复并修改订单状态为已取消

```
            orders = self.get_collection("orders")
            order_details = self.get_collection("order_details")
            stores = self.get_collection("stores")
            # 设置超时时间为30分钟            timeout = datetime.now() - timedelta(minutes
=30)
            print("order_id=" + str(order_id))
            # 查找超时未支付的订单            timeout_details = list(order_details.find({
```

```
                "order_id": order_id,
                "status": "not pay",
                "create_time": {"$lt": timeout}
            }))
            # print("******")          # print(str(timeout_details))          # print
("******")          # 按订单ID分组处理          processed_orders = set()
            for detail in timeout_details:
                order_id = detail["order_id"]
                if order_id in processed_orders:
                    continue              processed_orders.add(order_id)
                order = orders.find_one({"order_id": order_id})
                if not order:
                    continue              # 恢复库存              store_id = order["st
ore_id"]
                order_details_list = list(order_details.find({"order_id": order_id}))
                for od in order_details_list:
                    stores.update_one(
                        {"store_id": store_id, "book_id": od["book_id"]},
                        {"$inc": {"stock_level": od["count"]}}
                    )
                # 更新订单状态              current_time = datetime.now()
                order_details.update_many(
                    {"order_id": order_id},
                    {
                        "$set": {
                            "status": "cancelled",
                            "cancel_time": current_time,
                            "cancel_reason": "timeout"                    }
                    }
                )
```

- 最后返回状态码以及运行结果信息

4.查询历史订单 `get_order_history`

- **功能概述**:
  -此方法根据用户id查询数据库中该用户的所有历史订单的详细信息
- **参数解析**:
  - `user_id`:用户id。
- **查询逻辑**:
  -先查询该用户id是否存在。
  -使用
  `get_collection`方法获取数据库集合orders，order_details。
  -使用
  `find`方法查询数据库中该用户的所有订单
  -遍历找到的所有订单，将每个订单的属性记录并添加到字典order_history中

```
            if not self.user_id_exist(user_id):
                return error.error_non_exist_user_id(user_id) + ([],)
            print("user_id="+str(user_id))
            orders = self.get_collection("orders")
            order_details = self.get_collection("order_details")
            user_orders = list(orders.find({"user_id": user_id}))
            order_history = []
            for order in user_orders:
                order_id = order["order_id"]
                details = list(order_details.find({"order_id": order_id}))
                order_info = {
                    "order_id": order_id,
                    "store_id": order["store_id"],
                    "create_time": details[0]["create_time"] if details else None,
                    "status": details[0]["status"] if details else None,
                    "books": [{
                        "book_id": detail["book_id"],
                        "count": detail["count"],
                        "price": detail["price"]
                    } for detail in details],
```

```
                    "total_price": sum(detail["count"] * detail["price"] for detail in
details)
                }
                order_history.append(order_info)
```

```
-最后返回状态码，运行结果信息以及字典order_history
```

5.错误处理

- 针对数据库操作中可能发生的 `PyMongoError`，以及其他任何异常BaseException，采用了异常处理机制，以确保在出现错误时能返回适当的错误信息。

三，测试用例

1.测试历史订单查询

```
def test_order_history(self):
    ok, buy_book_id_list = self.gen_book.gen(
        non_exist_book_id=False, low_stock_level=False    )
    assert ok
    code, _ = self.buyer.new_order(self.store_id, buy_book_id_list)
    assert code == 200    code, orders = self.buyer.order_history()
    assert code == 200    assert isinstance(orders, list)
```

- 测试通过用户id查询历史订单的功能
- 断言状态码为200， 并返回包含历史订单的字典

2.测试手动取消订单

```
def test_cancel_order(self):
    ok, buy_book_id_list = self.gen_book.gen(
        non_exist_book_id=False, low_stock_level=False    )
    assert ok
    code, order_id = self.buyer.new_order(self.store_id, buy_book_id_list)
    assert code == 200    print("order_id_in_order.py",order_id)
    code = self.buyer.cancel_order(order_id)
    assert code == 200
```

- 测试通过用户id和订单id手动取消订单的功能
- 断言状态码为200

3.测试自动取消订单

```
def test_auto_cancel_timeout_orders(self):
    # 生成一本书并创建一个未支付的订单    ok, buy_book_id_list = self.gen_book.gen(non_exis
t_book_id=False, low_stock_level=False)
    assert ok
    code, order_id = self.buyer.old_order(self.store_id, buy_book_id_list)
    assert code == 200    # 模拟等待超时并调用自动取消函数    code, message = self.buyer.au
to_cancel_timeout_orders(order_id)  # 假设这里调用的是 Buyer 类的函数    assert code == 20
0    assert message == "ok"
```

- 测试通过查询订单id自动删除超时订单的功能
- 断言码状态为200

4.测试取消不存在的订单

```
def test_cancel_non_exist_order(self):
    order_id = "non_exist_order_id"    code = self.buyer.cancel_order(order_id)
    assert code != 200
```

- 测试取消不存在的订单
- 断言码状态不为200

5.测试取消不存在的用户的订单

```
def test_cancel_order_for_non_exist_user(self):
    self.buyer.user_id = self.buyer.user_id + "_x"    order_id = "test_order_id_{}".fo
rmat(str(uuid.uuid1()))
    code = self.buyer.cancel_order(order_id)
    assert code != 200
```

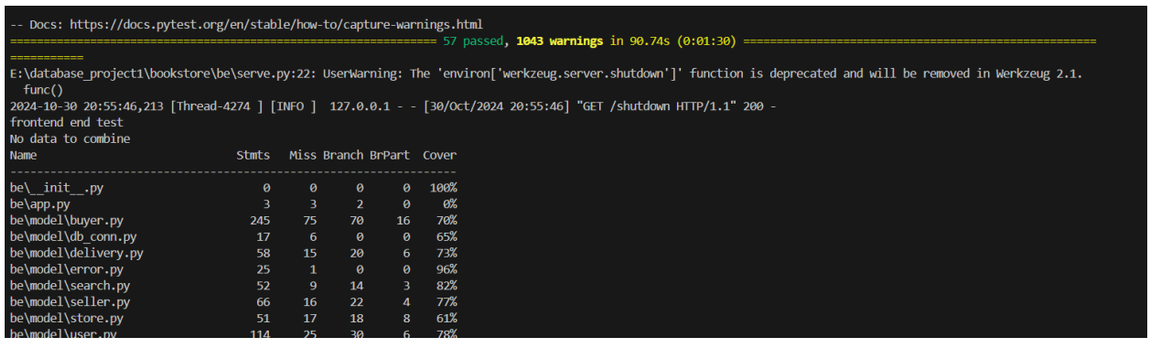
- 测试取消不存在的用户的订单
- 断言码状态不为200

6.测试查询不存在的用户的历史订单

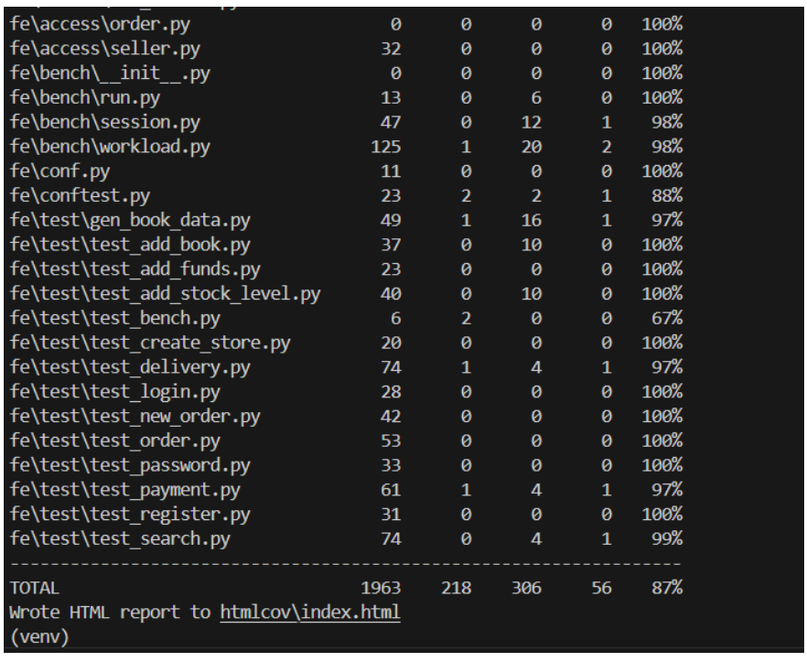
```
def test_order_history_for_non_exist_user(self):
    self.buyer.user_id = self.buyer.user_id + "_x"    code, orders = self.buyer.order_
history()
    assert code != 200    assert orders == []
```

- 测试查询不存在的用户的历史订单
- 断言码状态不为200，并返回一个空字典

结果展示



```
-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
============================================================ 57 passed, 1043 warnings in 90.74s (0:01:30) ============================================
==========
E:\database_project1\bookstore\be\serve.py:22: UserWarning: The 'environ['werkzeug.server.shutdown']' function is deprecated and will be removed in Werkzeug 2.1.
  func()
2024-10-30 20:55:46,213 [Thread-4274 ] [INFO ]  127.0.0.1 - - [30/Oct/2024 20:55:46] "GET /shutdown HTTP/1.1" 200 -
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                     245     75     70     16    70%
be\model\db_conn.py                    17      6      0      0    65%
be\model\delivery.py                   58     15     20      6    73%
be\model\error.py                      25      1      0      0    96%
be\model\search.py                     52      9     14      3    82%
be\model\seller.py                     66     16     22      4    77%
be\model\store.py                      51     17     18      8    61%
be\model\user.py                      114     25     30      6    78%
be\serve.py                            42      1      2      1    95%
be\view\auth.py                        42      0      0      0   100%
be\view\buyer.py                       80     10     10      1    83%
be\view\delivery.py                    19      0      0      0   100%
be\view\order.py                        0      0      0      0   100%
be\view\search.py                      39      7      8      1    83%
be\view\seller.py                      31      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                      67      4     10      1    91%
fe\access\book_search.py               18      0      4      0   100%
fe\access\buyer.py                     93     21      8      1    76%
fe\access\delivery.py                  27      0      0      0   100%
fe\access\new_buyer.py                  8      0      0      0   100%
fe\access\new_seller.py                13      0      0      0   100%
fe\access\order.py                      0      0      0      0   100%
fe\access\seller.py                    32      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%
```



索引介绍

在be/model/store.py的init_collections函数中，我们创建了一些简单的索引。
首先，它检查每个集合是否存在，如果不存在则创建该集合，并为`user_id`、`store_id`、`order_id`和组合字段（如`user_id`和`store_id`）设置唯一索引，以确保数据的唯一性和完整性。如果在创建过程中发生任何异常，会记录错误信息并抛出该异常。例如，对于 "users" 集合中的 "user_id" 字段设置唯一索引，可以防止重复用户的创建。同样，对于 "stores" 集合中的 "store_id" 和 "book_id" 组合字段设置唯一索引，可以确保每个商店中的每本书都是唯一的。这种索引策略不仅优化了数据库性能，还为应用程序的数据完整性提供了额外的保障。

```
def init_collections(self):
        """
        初始化所需的collections(相当于关系型数据库中的表)
        """
        try:
            # 用户集合
            if "users" not in self.db.list_collection_names():
                self.db.create_collection("users")
                self.db.users.create_index("user_id", unique=True)

            # 用户商店关系集合
            if "user_store" not in self.db.list_collection_names():
                self.db.create_collection("user_store")
                self.db.user_store.create_index([("user_id", 1), ("store_id", 1)], unique
=True)

            # 商店集合
            if "stores" not in self.db.list_collection_names():
                self.db.create_collection("stores")
                self.db.stores.create_index([("store_id", 1), ("book_id", 1)], unique=Tru
e)

            # 订单集合
            if "orders" not in self.db.list_collection_names():
                self.db.create_collection("orders")
                self.db.orders.create_index("order_id", unique=True)

            # 订单详情集合
            if "order_details" not in self.db.list_collection_names():
                self.db.create_collection("order_details")
                self.db.order_details.create_index([("order_id", 1), ("book_id", 1)], uni
que=True)

        except Exception as e:
            logging.error(f"初始化MongoDB集合时出错: {str(e)}")
            raise e
```

版本管理工具介绍

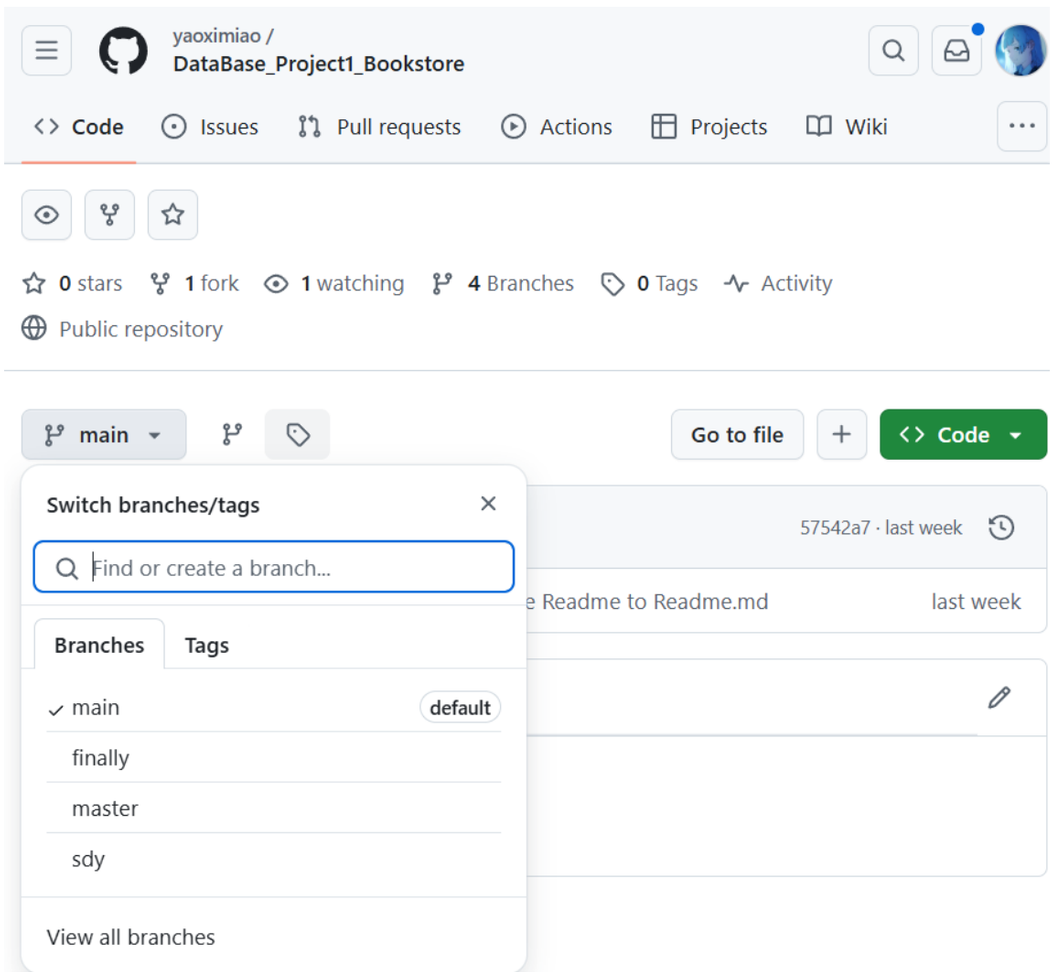
我们采用了Git作为版本控制系统，充分利用了其分支管理和协作功能。通过创建不同的分支，我们能够并行开发各个功能模块，同时保持代码的整洁性和可追踪性。在开发过程中，我们定期进行代码审查和合并，确保了团队成员之间的有效沟通和代码质量的提升。

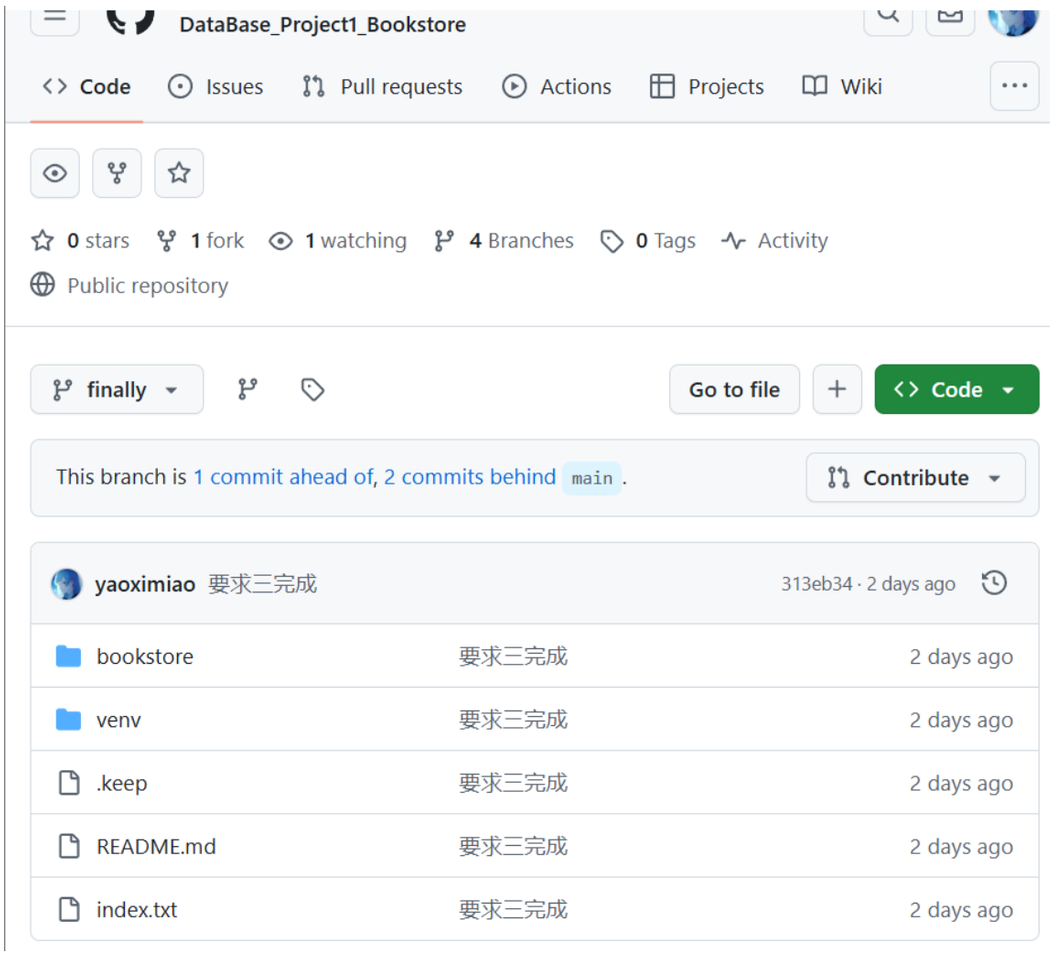
具体来说，我们的工作流程如下：

- 主分支（main）：保持稳定的生产就绪代码。
- 开发分支（develop）：作为功能开发的基础分支。
- 功能分支（feature branches）：为每个新功能或模块创建单独的分支，如"60%基础功能"、"40%要求1"等。
- 合并请求（Pull Requests）：在功能完成后，通过PR进行代码审查和讨论。
- 版本标签（Tags）：在重要里程碑使用标签标记版本。

这种方法使我们能够更好地管理复杂的开发过程，减少冲突，并确保代码质量。通过GitHub，我们还能够利用问题跟踪、项目看板等功能来增强团队协作和项目管理。

在项目的最后阶段，我们创建了"finally"分支作为最终稿，汇总了所有功能和改进。这个分支经过全面测试和审查后，最终合并到主分支，标志着项目的成功完成。




DataBase_Project1_Bookstore
Code
Issues
Pull requests
Actions
Projects
Wiki
0 stars
1 fork
1 watching
4 Branches
0 Tags
Activity
Public repository
finally
Go to file
Code
This branch is 1 commit ahead of, 2 commits behind main.
Contribute
yaoximiao 要求三完成
313eb34 · 2 days ago
bookstore 要求三完成 2 days ago
venv 要求三完成 2 days ago
.keep 要求三完成 2 days ago
README.md 要求三完成 2 days ago
index.txt 要求三完成 2 days ago