Bookstore 项目报告

项目分工

课程名称：当代数据管理系统	项目名称：bookstore	指导老师：周烜
姓名：徐以鑫	学号：10214602448	：60%基础功能中的用户权限接口和40%附加功能中的订单状态，订单查询和取消订单
姓名：朱季阳	学号：10225501428	60%基础功能中的买家用户接口和40%附加功能中的搜索图书功能
姓名：孙秋实	学号：10225501402	60%基础功能中的卖家用户接口和40%附加功能中的发货、收货

文档数据数据库设计

- **new_order**：

```
{
    "_id": "ObjectId",                  // MongoDB 自动生成的唯一标识符
    "order_id": "string",               // 订单ID (设置为唯一索引，以确保每个订单的唯一
性)
    "store_id": "string",               // 店铺ID (关联到商店)
    "user_id": "string",                // 用户ID (关联到下单用户)
    "payment_status": "string",         // 支付状态 (例如 "no_pay", "paid",
"shipped" 等)
    "payment_ddl": "int",               // 支付截止时间 (Unix时间戳)
    "total_price": "float"              // 订单总价 (可选字段，用于存储总金额)
}
```

- **new_order_detail**：

```
{
    "_id": "ObjectId",                  // MongoDB 自动生成的唯一标识符
    "order_id": "string",               // 订单ID (设置为索引)
    "book_id": "string",                // 书籍ID (设置为索引)
    "count": "int",                     // 书籍数量 (用户购买的数量)
    "price": "float"                    // 单本书籍价格
}
```

- **user**：

```
{
    "_id": "ObjectId",                  // MongoDB 自动生成的唯一标识符
    "user_id": "string",                // 用户ID (设置为唯一索引，以确保每个用户的唯一
性)
    "password": "string",               // 用户密码 (通常需要加密存储)
    "balance": "float",                 // 用户余额
    "token": "string",                  // 用户会话令牌 (用于身份验证)
    "terminal": "string"                // 用户设备信息 (如设备类型、操作系统等)
}
```

- **store:**

```
{
    "_id": "ObjectId",                  // MongoDB 自动生成的唯一标识符
    "store_id": "string",               // 店铺ID (设置为唯一索引，以确保每个店铺的唯一
性)
    "book_id": "string",                // 书籍ID (设置为索引)
    "book_title": "string",             // 书籍标题
    "book_tags": ["string"],            // 书籍标签 (数组形式，方便分类)
    "book_author": "string",            // 书籍作者
    "book_price": "float",              // 书籍价格
    "stock_level": "int"                // 库存量
}
```

- **user_store:**

```
{
    "_id": "ObjectId",                  // MongoDB 自动生成的唯一标识符
    "store_id": "string",               // 店铺ID (设置为索引)
    "user_id": "string"                 // 用户ID (设置为索引)
}
```

具体功能实现

1. be/model后端逻辑代码

1.1 buyer:

1. init

初始化买家类，继承了db_conn.DBConn类并调用其构造方法以建立数据库连接。

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

2. new_order

创建一个新的订单：检查用户ID和店铺ID是否存在。

为每本书检查库存是否足够，若不足则返回错误。

若库存充足，则扣减库存量，并创建订单详细信息，包括每本书的ID、数量和价格。

设置订单的付款截止时间为当前时间 +15秒。

返回成功的订单ID。

```
    def new_order(self, user_id: str, store_id: str, id_and_count: [(str,
int)]) -> (int, str, str):
        order_id = ""
        try:
            # 检查用户和店铺是否存在
            if not self.user_id_exist(user_id):
                return error.error_non_exist_user_id(user_id) +
(order_id,)
            if not self.store_id_exist(store_id):
                return error.error_non_exist_store_id(store_id) +
(order_id,)

            # 生成唯一订单ID
            uid = "{}_{}_{}".format(user_id, store_id, str(uuid.uuid1()))
            total_price = 0

            for book_id, count in id_and_count:
                # 查询库存和价格信息
                row = self.conn.store_col.find_one(
                    {"book_id": book_id, "store_id": store_id},
                    {"book_id": 1, "stock_level": 1, "book_price": 1}
                )

                if row is None:
                    return error.error_non_exist_book_id(book_id) +
(order_id,)

                stock_level = row["stock_level"]
                price = row["book_price"]

                # 检查库存
                if stock_level < count:
                    return error.error_stock_level_low(book_id) +
(order_id,)

                # 更新库存
                result = self.conn.store_col.update_one(
                    {"store_id": store_id, "book_id": book_id,
"stock_level": {"$gte": count}},
                    {"$inc": {"stock_level": -count}}
                )

                if result.modified_count == 0:
                    return error.error_stock_level_low(book_id) +
(order_id,)

                # 插入订单详情
                self.conn.new_order_detail_col.insert_one(
```

```
                    {"order_id": uid, "book_id": book_id, "count": count,
"price": price}
                )
                total_price += price * count

            # 设置订单支付截止时间
            current_time = int(time.time())
            payment_ddl = current_time + 15
            # 插入订单基础信息
            self.conn.new_order_col.insert_one({
                "order_id": uid, "store_id": store_id, "user_id": user_id,
                "payment_status": "no_pay", "payment_ddl": payment_ddl,
                "total_price": total_price
            })
            order_id = uid

        except BaseException as e:
            return 530, "{}".format(str(e)), ""

        return 200, "ok", order_id
```

3. payment

进行订单支付操作：检查订单状态，若已经支付则返回。

验证用户身份和密码。

计算订单总价并检查用户余额是否足够。

从买家余额中扣除订单金额，同时将金额加至卖家余额。

更新订单状态为"已支付"。

```
    def payment(self, user_id: str, password: str, order_id: str) -> (int,
str):
        try:
            # 获取订单信息
            order = self.conn.new_order_col.find_one({"order_id":
order_id})
            if order is None:
                return error.error_invalid_order_id(order_id)
            if order["payment_status"] != "no_pay":
                return error.error_invalid_payment_status(order_id)

            # 验证买家身份和密码
            user = self.conn.user_col.find_one({"user_id": user_id},
{"balance": 1, "password": 1})
            if user is None:
                return error.error_non_exist_user_id(user_id)
            if user["password"] != password:
                return error.error_authorization_fail()

            # 检查余额是否足够
            total_price = order["total_price"]
            if user["balance"] < total_price:
```

```
                return error.error_not_sufficient_funds(order_id)

            # 扣除买家余额并增加卖家余额
            self.conn.user_col.update_one({"user_id": user_id}, {"$inc":
{"balance": -total_price}})
            self.conn.user_col.update_one({"user_id": order["store_id"]},
{"$inc": {"balance": total_price}})

            # 更新订单状态为已支付
            self.conn.new_order_col.update_one({"order_id": order_id},
{"$set": {"payment_status": "paid"}})

        except pymongo.errors.PyMongoError as e:
            return 528, "{}".format(str(e))
        except BaseException as e:
            return 530, "{}".format(str(e))

        return 200, "ok"
```

4. add_funds

充值操作：验证用户身份和密码是否正确。

若正确，则在用户账户中增加充值金额。

```
    def add_funds(self, user_id, password, add_value) -> (int, str):
        try:
            # 验证用户身份和密码
            user = self.conn.user_col.find_one({"user_id": user_id},
{"password": 1})
            if user is None or user.get("password") != password:
                return error.error_authorization_fail()

            # 增加用户余额
            self.conn.user_col.update_one({"user_id": user_id}, {"$inc":
{"balance": add_value}})

        except BaseException as e:
            return 530, "{}".format(str(e))

        return 200, "ok"
```

5. book_search

搜索书籍：根据店铺ID、书籍ID、书名、标签和作者等条件搜索书籍。

返回符合条件的书籍（最多10个），若无结果则返回书籍不存在的错误。

```
    def book_search(self, store_id=None, book_id=None, book_title=None,
book_tags=None, book_author=None):
        try:
            query_conditions = {}
            if store_id:
                query_conditions["store_id"] = store_id
```

```
            if book_id:
                query_conditions["book_id"] = book_id
            if book_title:
                query_conditions["book_title"] = {"$regex": book_title,
"$options": "i"}
            if book_tags:
                query_conditions["book_tags"] = {"$all": book_tags}
            if book_author:
                query_conditions["book_author"] = book_author

            result = list(self.conn.store_col.find(query_conditions,
{}).limit(10))
            if not result:
                return error.error_non_exist_book_id(book_id)

        except BaseException as e:
            return 530, "{}".format(str(e))

        return 200, "ok"
```

6. delete_order

删除订单：若订单已支付，计算订单金额并将其返还给买家，同时从卖家账户中扣除金额。

删除订单记录及其详细信息。

若订单未支付，则直接删除，无需资金返还。

```
    def delete_order(self, user_id, order_id) -> (int, str):
        try:
            # 获取订单信息
            order = self.conn.new_order_col.find_one({"order_id":
order_id})
            if not order:
                return error.error_non_order_delete(user_id)

            # 根据订单支付状态进行处理
            if order["payment_status"] == "paid":
                total_price = order["total_price"]
                # 返还买家余额
                self.conn.user_col.update_one({"user_id": user_id},
{"$inc": {"balance": total_price}})
                # 扣减卖家余额
                self.conn.user_col.update_one({"user_id":
order["store_id"]}, {"$inc": {"balance": -total_price}})

            # 删除订单及详情
            self.conn.new_order_col.delete_one({"order_id": order_id})
            self.conn.new_order_detail_col.delete_many({"order_id":
order_id})

        except BaseException as e:
            return 530, "{}".format(str(e))
```

```
        return 200, "ok"
```

7. search_order

查找订单：删除超时未支付的订单。

查找用户作为买家的订单，若没有找到订单则返回空订单错误。

```
    def search_order(self, user_id) -> (int, str):
        try:
            # 删除超时未支付订单
            current_time = int(time.time())
            expired_orders = self.conn.new_order_col.find(
                {"user_id": user_id, "payment_ddl": {"$lt": current_time},
"payment_status": "no_pay"}
            )
            expired_order_ids = [order["order_id"] for order in
expired_orders]
            self.conn.new_order_col.delete_many({"order_id": {"$in":
expired_order_ids}})
            self.conn.new_order_detail_col.delete_many({"order_id":
{"$in": expired_order_ids}})

            # 返回用户所有订单
            orders = list(self.conn.new_order_col.find({"user_id":
user_id}))
            if not orders:
                return error.empty_order_search(user_id)

        except BaseException as e:
            return 530, "{}".format(str(e))

        return 200, "ok"
```

8. receive

确认收货：检查订单状态，若未支付或尚未发货则返回相应错误。

若符合条件，则更新订单状态为“已收货”。

```
 def receive(self, user_id: str, store_id: str, order_id: str) -> (int,
str):
        try:
            # 获取订单信息
            order = self.conn.new_order_col.find_one({"order_id":
order_id})
            if not order or order["user_id"] != user_id:
                return error.error_non_exist_user_id(user_id)
            if order["store_id"] != store_id:
                return error.error_non_exist_store_id(store_id)
            if order["payment_status"] == "no_pay":
                return 521, "订单未付款"
            if order["payment_status"] == "received":
                return 523, "订单已接收"
```

```
            # 更新订单状态为已接收
            self.conn.new_order_col.update_one({"order_id": order_id},
{"$set": {"payment_status": "received"}})

        except BaseException as e:
            return 532, "{}".format(str(e))

        return 200, "ok"
```

- **1.2 user：`User` 类为用户管理提供了完整的功能，包括用户的注册、登录、登出、密码管理等，使用 JWT 确保安全性，并通过 MongoDB 进行数据存储和管理。整体结构清晰，功能模块化，易于维护和扩展。**

  1. 初始化：

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

  2. 检查 Token

     ```
         def __check_token(self, user_id, db_token, token) -> bool:
         # 检查数据库中的 token 和提供的 token 是否一致，并验证时间戳
             try:
                 if db_token != token:
                     return False
                 jwt_text = jwt_decode(encoded_token=token,
     user_id=user_id)
                 ts = jwt_text["timestamp"]
                 if ts is not None:
                     now = time.time()
                     if self.token_lifetime > now - ts >= 0:
                         return True
             except jwt.exceptions.InvalidSignatureError as e:
                 logging.error(str(e))
                 return False
     ```

  3. 用户注册

     ```
         def register(self, user_id: str, password: str):
             # 检查用户是否已存在，不存在则注册用户
             try:
                 # 插入前进行重复冲突检查
                 existing_user = self.conn.user_col.find_one({"user_id":
     user_id})

                 if existing_user:
                     return 530, "User already exists"

                 terminal = "terminal_{}".format(str(time.time()))
     ```

```
            token = jwt_encode(user_id, terminal)

            # 确定无冲突后执行插入
            self.conn.user_col.insert_one({
                "user_id": user_id,
                "password": password,
                "balance": 0,
                "token": token,
                "terminal": terminal
            })
        except Exception as e:
            return 530, str(e)
        return 200, "ok"
```

4. 检查 Token

```
    def check_token(self, user_id: str, token: str) -> (int, str):
     # 根据用户ID检查 token 是否有效
        user_doc = self.conn.user_col.find_one({"user_id": user_id},
{"token": 1})

        if user_doc is None:
            return error.error_authorization_fail()

        db_token = user_doc.get("token")

        if not self.__check_token(user_id, db_token, token):
            return error.error_authorization_fail()

        return 200, "ok"
```

5. 检查密码

```
    def check_password(self, user_id: str, password: str) -> (int,
str):
    # 检查数据库中存储的密码与用户输入的密码是否匹配。
        user_doc = self.conn.user_col.find_one({"user_id": user_id})

        if user_doc is None:
            return error.error_authorization_fail()

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

        return 200, "ok"
```

6. 用户登录

```
    def login(self, user_id: str, password: str, terminal: str) ->
(int, str, str):
    # 用户登录，检查密码，生成并更新 token
```

```
        token = ""
        try:
            # 用户密码检查
            code, message = self.check_password(user_id, password)
            if code != 200:
                return code, message, ""

            token = jwt_encode(user_id, terminal)
            result = self.conn.user_col.update_one(
                {"user_id": user_id},
                {"$set": {"token": token, "terminal": terminal}}
            )
            if result.modified_count == 0:
                return 401, "Authorization failed", ""
        except Exception as e:
            return 500, str(e), ""
        return 200, "OK", token
```

7. 用户登出

```
    def logout(self, user_id: str, token: str) -> (int, str):
        # 用户登出，生成无效的 token 并更新数据库
        try:
            code, message = self.check_token(user_id, token)
            if code != 200:
                return code, message

            terminal = "terminal_{}".format(str(time.time()))
            dummy_token = jwt_encode(user_id, terminal)

            result = self.conn.user_col.find_one_and_update(
                {"user_id": user_id},
                {"$set": {"token": dummy_token, "terminal":
terminal}},
                return_document=ReturnDocument.AFTER
            )
            if result is None:
                return error.error_authorization_fail()

        except Exception as e:
            return 530, str(e)

        return 200, "ok"
```

8. 用户注销

```
    def unregister(self, user_id: str, password: str) -> (int, str):
    # 验证用户身份并从数据库中删除用户。
        try:
            code, message = self.check_password(user_id, password)
            if code != 200:
                return code, message
```

```
            result = self.conn.user_col.delete_one({"user_id":
user_id})

            if result.deleted_count == 1:
                return 200, "ok"
            else:
                return error.error_authorization_fail()

        except Exception as e:
            return 530, str(e)
```

9. 修改密码

```
    def change_password(
        self, user_id: str, old_password: str, new_password: str
    ) -> (int, str):
    # 验证旧密码并将密码更改为新密码，同时更新 token。
        try:
            code, message = self.check_password(user_id, old_password)
            if code != 200:
                return code, message

            terminal = "terminal_{}".format(str(time.time()))
            token = jwt_encode(user_id, terminal)

            result = self.conn.user_col.update_one(
                {"user_id": user_id},
                {"$set": {"password": new_password, "token": token,
"terminal": terminal}}
            )

            if result.modified_count == 0:
                return error.error_authorization_fail()

            return 200, "ok"
        except Exception as e:
            return 530, str(e)
```

- **1.3 seller：为卖家提供了基本的书籍管理和订单处理功能，确保数据的完整性和一致性**

1. **add_book**:检查用户和店铺是否存在，以及书籍是否已存在。创建书籍数据字典。将书籍信息插入到 MongoDB 的 `store_col` 集合中。

```
    def add_book(self, user_id: str, store_id: str, book_id: str,
book_title: str, book_tags, book_author,
                 book_price: str,
                 stock_level: int):
        try:
            if not self.user_id_exist(user_id):
                return error.error_non_exist_user_id(user_id)
            if not self.store_id_exist(store_id):
                return error.error_non_exist_store_id(store_id)
```

```
            if self.book_id_exist(store_id, book_id):
                return error.error_exist_book_id(book_id)

            data = {
                "store_id": store_id,
                "book_id": book_id,
                "book_title": book_title,
                "book_tags": book_tags,
                "book_author": book_author,
                "book_price": book_price,
                "stock_level": stock_level
            }
            # 插入数据到MongoDB
            self.conn.store_col.insert_one(data)

        except BaseException as e:
            return 530, "{}".format(str(e))
        return 200, "ok"  # 成功添加书籍，返回状态码200和消息
```

2. **add_stock_level**：检查用户、店铺和书籍是否存在。使用 MongoDB 的 `$inc` 操作符更新库存数量。

```
    def add_stock_level(
            self, user_id: str, store_id: str, book_id: str,
add_stock_level: int
    ):
        try:

            if not self.user_id_exist(user_id):
                return error.error_non_exist_user_id(user_id)
            if not self.store_id_exist(store_id):
                return error.error_non_exist_store_id(store_id)
            if not self.book_id_exist(store_id, book_id):
                return error.error_non_exist_book_id(book_id)

            # self.conn.execute(
            #     "UPDATE store SET stock_level = stock_level + ? "
            #     "WHERE store_id = ? AND book_id = ?",
            #     (add_stock_level, store_id, book_id),
            # )

            filter_query = {"store_id": store_id, "book_id": book_id}
            update_query = {"$inc": {"stock_level": add_stock_level}}

            self.conn.store_col.update_one(filter_query, update_query)
        except BaseException as e:
            return 500, "{}".format(str(e))

        return 200, "ok"
```

3. **create_store**：检查用户是否存在，且店铺ID是否已存在。将店铺信息插入到 `user_store_col` 集合中。

```
    def create_store(self, user_id: str, store_id: str) -> (int, str):
        try:
            if not self.user_id_exist(user_id):
                return error.error_non_exist_user_id(user_id)
            if self.store_id_exist(store_id):
                return error.error_exist_store_id(store_id)

            data = {
                "store_id": store_id,
                "user_id": user_id
            }

            self.conn.user_store_col.insert_one(data)

        except BaseException as e:
            return 500, "{}".format(str(e))
        return 200, "ok"
```

4. **search_order**：遍历当前订单，删除超时未支付的订单。获取卖家名下所有店铺的ID。查找与这些店铺相关的订单ID。

```
    def search_order(self, user_id) -> (int, str):
        try:
            # 搜索前遍历订单删除超时订单
            current_time = int(time.time())
            payment_overtime_order_ids = [order['order_id'] for order
in

 self.conn.new_order_col.find({"payment_ddl": {"$lt": current_time},

 "payment_status": "no_pay"},

{"order_id": 1})]
            self.conn.new_order_col.delete_many({"order_id": {"$in":
payment_overtime_order_ids}})
            self.conn.new_order_detail_col.delete_many({"order_id":
{"$in": payment_overtime_order_ids}})

            # 将用户作为卖家进行搜索
            # 获取卖家名下店铺
            seller_store_ids = [store['store_id'] for store in

 self.conn.user_store_col.find({"user_id": user_id}, {"store_id": 1})]
            # 通过店铺store_id搜索订单order_id
            seller_order_ids = [order['order_id'] for order in

 self.conn.new_order_col.find({"store_id": {"$in": seller_store_ids}},

{"order_id": 1})]
            if not seller_order_ids:
                return error.empty_order_search(user_id)
            self.conn.new_order_col.find({"order_id": {"$in":
seller_order_ids}}, {})
```

```
        except BaseException as e:
            return 530, "{}".format(str(e))
        return 200, "ok"
```

5. **deliver**：检查订单是否存在，用户和店铺是否有效。检查订单的支付状态，如果未支付或已发货，返回相应错误。更新订单状态为“已发货”。

```
    def deliver(self, user_id: str, store_id: str, order_id: str) ->
(int, str):
        try:
            result = self.conn.new_order_col.find_one({"order_id":
order_id})
            if result is None:
                return error.error_non_exist_order(order_id)

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

            if not self.store_id_exist(store_id):
                return error.error_non_exist_store_id(store_id)

            result = self.conn.new_order_col.find_one({"order_id":
order_id})
            status = result['payment_status']

            if status == "no_pay":
                return 521, {"no_pay"}
            elif status == "shipped" or status == "received":
                return 522, {"shipped"}

            self.conn.new_order_col.update_one({"order_id": order_id},
{"$set": {"payment_status": 'shipped'}})  # 已发货

        except BaseException as e:
            return 531, "{}".format(str(e))
        return 200, "ok"
```

- ## 2. be/view：访问后端接口

  - ### 2.1buyer:

    1.新建订单接口 (/buyer/new_order)

    方法: POST

    功能: 创建新的订单。接收用户 ID、店铺 ID 和图书列表（包含每本书的 ID 和数量），调用后端逻辑生成订单，并返回订单 ID 和消息。

```
@bp_buyer.route("/new_order", methods=["POST"])
def new_order():
    user_id: str = request.json.get("user_id")
    store_id: str = request.json.get("store_id")
    books: [] = request.json.get("books")
```

```
    id_and_count = []
    for book in books:
        book_id = book.get("id")
        count = book.get("count")
        id_and_count.append((book_id, count))

    b = Buyer()
    code, message, order_id = b.new_order(user_id, store_id, id_and_count)
    return jsonify({"message": message, "order_id": order_id}), code
```

2.付款接口 (/buyer/payment)

方法: POST

功能: 处理订单支付。接收用户 ID、订单 ID 和密码，调用后端逻辑进行支付，并返回支付结果消息。

```
@bp_buyer.route("/payment", methods=["POST"])
def payment():
    user_id: str = request.json.get("user_id")
    order_id: str = request.json.get("order_id")
    password: str = request.json.get("password")
    b = Buyer()
    code, message = b.payment(user_id, password, order_id)
    return jsonify({"message": message}), code
```

3.添加资金接口 (/buyer/add_funds)

方法: POST

功能: 向用户账户添加资金。接收用户 ID、密码和添加的金额，调用后端逻辑处理资金添加，并返回操作结果消息。

```
@bp_buyer.route("/add_funds", methods=["POST"])
def add_funds():
    user_id = request.json.get("user_id")
    password = request.json.get("password")
    add_value = request.json.get("add_value")
    b = Buyer()
    code, message = b.add_funds(user_id, password, add_value)
    return jsonify({"message": message}), code
```

4.图书查询接口 (/buyer/search_book)

方法: POST

功能: 查询特定书籍。接收店铺 ID、书籍 ID、书名、标签和作者信息，调用后端逻辑进行图书搜索，并返回搜索结果消息。

```python
@bp_buyer.route("/search_book", methods=["POST"])
def book_search():
    store_id: str = request.json.get("store_id")
    book_id = request.json.get("book_id")
    book_title = request.json.get("book_title")
    book_tags = request.json.get("book_tags")
    book_author = request.json.get("book_author")
    b = Buyer()
    code, message = b.book_search(
        store_id, book_id, book_title, book_tags, book_author
    )
    return jsonify({"message": message}), code
```

5.删除订单接口 (/buyer/delete_order)

方法: POST

功能: 删除用户的订单。接收用户 ID 和订单 ID，调用后端逻辑删除订单，并返回操作结果消息。

```python
@bp_buyer.route("/delete_order", methods=["POST"])
def delete_order():
    # 请求需传入user_id
    user_id: str = request.json.get("user_id")
    order_id: str = request.json.get("order_id")
    b = Buyer()
    code, message = b.delete_order(user_id, order_id)
    return jsonify({"message": message}), code
```

6.查询订单接口 (/buyer/search_order)

方法: POST

功能: 查询用户的订单。接收用户 ID，调用后端逻辑查找订单，并返回查询结果消息。

```python
@bp_buyer.route("/search_order", methods=["POST"])
def search_order():
    # 请求需传入user_id
    user_id: str = request.json.get("user_id")
    b = Buyer()
    code, message = b.search_order(user_id)
    return jsonify({"message": message}), code
```

7.确认收货接口 (/buyer/receive)

方法: POST

功能: 确认收到的订单。接收用户 ID、店铺 ID 和订单 ID，调用后端逻辑确认收货，并返回操作结果消息

```python
@bp_buyer.route("/receive", methods=["POST"])
def receive():
    user_id: str = request.json.get("user_id")
    order_id: str = request.json.get("order_id")
    store_id: str = request.json.get("store_id")
    b = Buyer()
    code, message = b.receive(user_id, store_id, order_id)
    return jsonify({"message": message}), code
```

- ## 2.2 Auth:

1. `login`

- **功能**: 用户登录。
- 实现:
  - 从请求的 JSON 数据中获取 `user_id`、`password` 和 `terminal`。
  - 创建 `User` 实例，并调用 `login` 方法进行登录验证。
  - 返回登录结果，包括消息和令牌。

```python
@bp_auth.route("/login", methods=["POST"])
def login():
    user_id = request.json.get("user_id", "")
    password = request.json.get("password", "")
    terminal = request.json.get("terminal", "")
    u = user.User()
    code, message, token = u.login(
        user_id=user_id, password=password, terminal=terminal
    )
    return jsonify({"message": message, "token": token}), code
```

2. `logout`

- **功能**: 用户登出。
- 实现:
  - 从请求中获取 `user_id` 和请求头中的 `token`。
  - 创建 `User` 实例，并调用 `logout` 方法执行登出操作。
  - 返回登出结果的消息。

```python
@bp_auth.route("/logout", methods=["POST"])
def logout():
    user_id: str = request.json.get("user_id")
    token: str = request.headers.get("token")
    u = user.User()
    code, message = u.logout(user_id=user_id, token=token)
    return jsonify({"message": message}), code
```

3. `register`

- **功能**: 用户注册。

- 实现:
  - 从请求的 JSON 数据中获取 `user_id` 和 `password`。
  - 创建 `User` 实例，并调用 `register` 方法进行注册。
  - 返回注册结果的消息。

```
@bp_auth.route("/register", methods=["POST"])
def register():
    user_id = request.json.get("user_id", "")
    password = request.json.get("password", "")
    u = user.User()
    code, message = u.register(user_id=user_id, password=password)
    return jsonify({"message": message}), code
```

4. `unregister`

- **功能**: 用户注销。
- 实现:
  - 从请求的 JSON 数据中获取 `user_id` 和 `password`。
  - 创建 `User` 实例，并调用 `unregister` 方法执行注销操作。
  - 返回注销结果的消息。

```
@bp_auth.route("/unregister", methods=["POST"])
def unregister():
    user_id = request.json.get("user_id", "")
    password = request.json.get("password", "")
    u = user.User()
    code, message = u.unregister(user_id=user_id, password=password)
    return jsonify({"message": message}), code
```

5. `change_password`

- **功能**: 修改用户密码。
- 实现:
  - 从请求的 JSON 数据中获取 `user_id`、`oldPassword` 和 `newPassword`。
  - 创建 `User` 实例，并调用 `change_password` 方法进行密码修改。
  - 返回修改结果的消息。

```
@bp_auth.route("/password", methods=["POST"])
def change_password():
    user_id = request.json.get("user_id", "")
    old_password = request.json.get("oldPassword", "")
    new_password = request.json.get("newPassword", "")
    u = user.User()
    code, message = u.change_password(
        user_id=user_id, old_password=old_password,
new_password=new_password
    )
    return jsonify({"message": message}), code
```

- ## 2.3 Seller

1. `seller_create_store`

- **功能**: 创建店铺。
- 实现:
  - 从请求的 JSON 数据中获取 `user_id` 和 `store_id`。
  - 创建 `Seller` 实例，并调用 `create_store` 方法创建店铺。
  - 返回创建结果的消息和状态码。

```
@bp_seller.route("/create_store", methods=["POST"])
def seller_create_store():
    user_id: str = request.json.get("user_id")
    store_id: str = request.json.get("store_id")
    s = seller.Seller()
    code, message = s.create_store(user_id, store_id)
    return jsonify({"message": message}), code
```

2. `seller_add_book`

- **功能**: 添加书籍到店铺。
- 实现:
  - 从请求中获取 `user_id`、`store_id` 和 `book_info`（书籍信息），以及 `stock_level`（库存数量）。
  - 创建 `Seller` 实例，并调用 `add_book` 方法将书籍添加到店铺。
  - 返回添加结果的消息和状态码。

```
@bp_seller.route("/add_book", methods=["POST"])
def seller_add_book():
    user_id: str = request.json.get("user_id")
    store_id: str = request.json.get("store_id")
    book_info = request.json.get("book_info")
    stock_level: int = request.json.get("stock_level", 0)

    s = seller.Seller()
    code, message = s.add_book(
```

```
        user_id, store_id, book_info.get("id"), book_info.get("title"),
book_info.get("tags"), book_info.get("author"),
        book_info.get("price"), stock_level
    )

    return jsonify({"message": message}), code
```

3. `add_stock_level`

- **功能**: 增加书籍的库存数量。
- 实现:
  - 从请求的 JSON 数据中获取 `user_id`、`store_id`、`book_id` 和要增加的库存数量 `add_stock_level`。
  - 创建 `Seller` 实例，并调用 `add_stock_level` 方法更新库存。
  - 返回更新结果的消息和状态码。

```
@bp_seller.route("/add_stock_level", methods=["POST"])
def add_stock_level():
    user_id: str = request.json.get("user_id")
    store_id: str = request.json.get("store_id")
    book_id: str = request.json.get("book_id")
    add_num = request.json.get("add_stock_level", 0)

    s = seller.Seller()
    code, message = s.add_stock_level(user_id, store_id, book_id, add_num)

    return jsonify({"message": message}), code
```

4. `search_order`

- **功能**: 搜索店铺订单。
- 实现:
  - 从请求中获取 `user_id`。
  - 创建 `Seller` 实例，并调用 `search_order` 方法获取订单信息。
  - 返回搜索结果的消息和状态码。

```
@bp_seller.route("/search_order", methods=["POST"])
def search_order():
    # 请求需传入user_id
    user_id: str = request.json.get("user_id")
    s = seller.Seller()
    code, message = s.search_order(user_id)
    return jsonify({"message": message}), code
```

5. `deliver`

- **功能**: 发货处理。
- 实现:

- 从请求的 JSON 数据中获取 `user_id`、`order_id` 和 `store_id`。
- 创建 `Seller` 实例，并调用 `deliver` 方法更新订单状态为已发货。
- 返回发货处理结果的消息和状态码。

```
@bp_seller.route("/deliver", methods=["POST"])
def deliver():
    user_id: str = request.json.get("user_id")
    order_id: str = request.json.get("order_id")
    store_id: str = request.json.get("store_id")
    s = seller.Seller()
    code, message = s.deliver(user_id, store_id, order_id)
    return jsonify({"message": message}), code
```

3. fe/test:功能性测试

3.1 User:测试使用者各个接口

test_login:

1. `test_ok`

- **功能**: 测试正常登录和登出流程。
- 实现:
  - 调用 `login` 方法，使用正确的 `user_id` 和 `password`，验证返回的状态码应为 200。
  - 测试登出功能:
    - 尝试使用错误的 `user_id` 和有效的 `token`，期望返回 401（未授权）。
    - 尝试使用有效的 `user_id` 和错误的 `token`，期望返回 401。
    - 使用正确的 `user_id` 和 `token` 登出，期望返回 200。

```
    def test_ok(self):
        code, token = self.auth.login(self.user_id, self.password,
self.terminal)
        assert code == 200

        code = self.auth.logout(self.user_id + "_x", token)
        assert code == 401

        code = self.auth.logout(self.user_id, token + "_x")
        assert code == 401

        code = self.auth.logout(self.user_id, token)
        assert code == 200
```

2. `test_error_user_id`

- **功能**: 测试使用错误的用户 ID 登录。
- 实现:

- 尝试使用不存在的用户 ID 调用 `login`，验证返回的状态码应为 401（未授权）。

```
    def test_error_user_id(self):
        code, token = self.auth.login(self.user_id + "_x",
self.password, self.terminal)
        assert code == 401
```

3. `test_error_password`

- **功能**: 测试使用错误的密码登录。
- 实现:
  - 尝试使用正确的用户 ID 和错误的密码调用 `login`，验证返回的状态码应为 401（未授权）。

```
def test_error_password(self):
        code, token = self.auth.login(self.user_id, self.password +
"_x", self.terminal)
        assert code == 401
```

- **test_register**

1. `test_register_ok`

- **功能**: 测试用户注册的成功场景。
- 实现:
  - 调用 `register` 方法，使用生成的 `user_id` 和 `password`，验证返回的状态码应为 200（表示注册成功）。

```
    def test_register_ok(self):
        code = self.auth.register(self.user_id, self.password)
        assert code == 200
```

2. `test_unregister_ok`

- **功能**: 测试用户注销的成功场景。
- 实现:
  - 先注册用户，确保返回状态码为 200。
  - 随后调用 `unregister` 方法注销用户，验证返回的状态码应为 200（表示注销成功）。

```
 def test_unregister_ok(self):
        code = self.auth.register(self.user_id, self.password)
        assert code == 200

        code = self.auth.unregister(self.user_id, self.password)
        assert code == 200
```

3. `test_unregister_error_authorization`

- **功能**: 测试注销时使用错误的用户 ID 或密码的场景。
- 实现:
  - 首先注册用户，确保返回状态码为 200。
  - 尝试使用错误的用户 ID 调用 `unregister`，验证返回状态码应不等于 200（表示注销失败）。
  - 尝试使用错误的密码调用 `unregister`，同样验证返回状态码应不等于 200。

```
    def test_unregister_error_authorization(self):
        code = self.auth.register(self.user_id, self.password)
        assert code == 200

        code = self.auth.unregister(self.user_id + "_x",
self.password)
        assert code != 200

        code = self.auth.unregister(self.user_id, self.password +
"_x")
        assert code != 200
```

4. `test_register_error_exist_user_id`

- **功能**: 测试注册时使用已存在的用户 ID 的场景。
- 实现:
  - 首先注册用户，确保返回状态码为 200。
  - 再次尝试注册相同的 `user_id`，验证返回的状态码应不等于 200（表示注册失败）。

```
    def test_register_error_exist_user_id(self):
        code = self.auth.register(self.user_id, self.password)
        assert code == 200

        code = self.auth.register(self.user_id, self.password)
        assert code != 200
```

- **test_password**

1. `test_ok`

- **功能**: 测试修改密码的成功场景。
- 实现:
  - 调用 `password` 方法，使用 `user_id`、旧密码和新密码，验证返回的状态码应为 200（表示密码修改成功）。
  - 尝试使用旧密码登录，验证返回的状态码应不等于 200（表示登录失败）。
  - 使用新密码登录，验证返回的状态码应为 200（表示登录成功）。
  - 使用新密码注销用户，验证返回的状态码应为 200（表示注销成功）。

```
    def test_ok(self):
```

```
        code = self.auth.password(self.user_id, self.old_password,
self.new_password)
        assert code == 200

        code, new_token = self.auth.login(
            self.user_id, self.old_password, self.terminal
        )
        assert code != 200

        code, new_token = self.auth.login(
            self.user_id, self.new_password, self.terminal
        )
        assert code == 200

        code = self.auth.logout(self.user_id, new_token)
        assert code == 200
```

2. `test_error_password`

- **功能**: 测试使用错误的旧密码修改密码的场景。
- 实现:
  - 调用 `password` 方法，使用错误的旧密码和新密码，验证返回的状态码应不等于 200（表示密码修改失败）。
  - 尝试使用新密码登录，验证返回的状态码应不等于 200（表示登录失败）。

```
    def test_error_password(self):
        code = self.auth.password(
            self.user_id, self.old_password + "_x", self.new_password
        )
        assert code != 200

        code, new_token = self.auth.login(
            self.user_id, self.new_password, self.terminal
        )
        assert code != 200
```

3. `test_error_user_id`

- **功能**: 测试使用错误的用户 ID 修改密码的场景。
- 实现:
  - 调用 `password` 方法，使用错误的用户 ID 和旧密码、新密码，验证返回的状态码应不等于 200（表示密码修改失败）。
  - 尝试使用新密码登录，验证返回的状态码应不等于 200（表示登录失败）。

```
    def test_error_user_id(self):
        code = self.auth.password(
            self.user_id + "_x", self.old_password, self.new_password
        )
        assert code != 200

        code, new_token = self.auth.login(
            self.user_id, self.new_password, self.terminal
        )
        assert code != 200
```

3.2 Seller：测试卖家的各个接口

test_create_store

1. `test_ok`

- **功能**: 测试成功创建商店的场景。
- 实现:
  - 调用 `register_new_seller` 函数，注册新卖家并保存返回的卖家对象。
  - 使用 `create_store` 方法创建新商店，传入 `store_id`。
  - 验证返回的状态码应为 200（表示商店创建成功）。

```
    def test_ok(self):
        self.seller = register_new_seller(self.user_id, self.password)
        code = self.seller.create_store(self.store_id)
        assert code == 200
```

2. `test_error_exist_store_id`

- **功能**: 测试创建已存在的商店 ID 的场景。
- 实现:
  - 首先调用 `register_new_seller` 函数注册新卖家。
  - 第一次调用 `create_store` 方法创建商店，验证返回的状态码应为 200（表示商店创建成功）。
  - 再次调用 `create_store` 方法使用相同的 `store_id`，验证返回的状态码应不等于 200（表示商店 ID 已存在，创建失败）。

```
    def test_error_exist_store_id(self):
        self.seller = register_new_seller(self.user_id, self.password)
        code = self.seller.create_store(self.store_id)
        assert code == 200

        code = self.seller.create_store(self.store_id)
        assert code != 200
```

- **test_add_book**

1. `test_ok`

- **功能**: 测试成功添加书籍的场景。
- 实现:
  - 遍历 `self.books` 中的每本书，调用 `add_book` 方法将其添加到商店。
  - 验证返回的状态码应为 200（表示书籍添加成功）。

```
    def test_ok(self):
        for b in self.books:
            code = self.seller.add_book(self.store_id, 0, b)
            assert code == 200
```

2. `test_error_non_exist_store_id`

- **功能**: 测试使用不存在的商店 ID 添加书籍的场景。
- 实现:
  - 遍历 `self.books` 中的每本书，调用 `add_book` 方法，传入不存在的商店 ID（通过在现有 ID 后加 “x”）。
  - 验证返回的状态码应不等于 200（表示商店 ID 不存在，添加失败）。

```
    def test_error_non_exist_store_id(self):
        for b in self.books:
            # non exist store id
            code = self.seller.add_book(self.store_id + "x", 0, b)
            assert code != 200
```

3. `test_error_exist_book_id`

- **功能**: 测试添加已存在书籍 ID 的场景。
- 实现:
  - 首先将每本书添加到商店，验证返回的状态码应为 200。
  - 再次尝试添加相同的书籍，验证返回的状态码应不等于 200（表示书籍 ID 已存在，添加失败）。

```
    def test_error_exist_book_id(self):
        for b in self.books:
            code = self.seller.add_book(self.store_id, 0, b)
            assert code == 200
        for b in self.books:
            # exist book id
            code = self.seller.add_book(self.store_id, 0, b)
            assert code != 200
```

4. `test_error_non_exist_user_id`

- **功能**: 测试使用不存在的用户 ID 添加书籍的场景。

- 实现:
  - 在遍历 `self.books` 时，将卖家的 `seller_id` 修改为一个不存在的 ID（通过在现有 ID 后加 “_x”）。
  - 调用 `add_book` 方法尝试添加书籍，验证返回的状态码应不等于 200（表示用户 ID 不存在，添加失败）。

```
    def test_error_non_exist_user_id(self):
        for b in self.books:
            # non exist user id
            self.seller.seller_id = self.seller.seller_id + "_x"
            code = self.seller.add_book(self.store_id, 0, b)
            assert code != 200
```

- ## test_add_stock_leve

1. `test_error_user_id`

- **功能**: 测试使用不存在的用户 ID 添加库存的场景。
- 实现:
  - 遍历 `self.books` 中的每本书，调用 `add_stock_level` 方法，传入不存在的用户 ID（通过在现有 ID 后加 “_x”）。
  - 验证返回的状态码应不等于 200（表示用户 ID 不存在，添加库存失败）。

```
    def test_error_user_id(self):
        for b in self.books:
            book_id = b.id
            code = self.seller.add_stock_level(
                self.user_id + "_x", self.store_id, book_id, 10
            )
            assert code != 200
```

2. `test_error_store_id`

- **功能**: 测试使用不存在的商店 ID 添加库存的场景。
- 实现:
  - 遍历 `self.books` 中的每本书，调用 `add_stock_level` 方法，传入不存在的商店 ID（通过在现有 ID 后加 “_x”）。
  - 验证返回的状态码应不等于 200（表示商店 ID 不存在，添加库存失败）。

```
    def test_error_store_id(self):
        for b in self.books:
            book_id = b.id
            code = self.seller.add_stock_level(
                self.user_id, self.store_id + "_x", book_id, 10
            )
            assert code != 200
```

3. `test_error_book_id`

- **功能**: 测试使用不存在的书籍 ID 添加库存的场景。
- 实现:
  - 遍历 `self.books` 中的每本书，调用 `add_stock_level` 方法，传入不存在的书籍 ID（通过在现有 ID 后加 "_x"）。
  - 验证返回的状态码应不等于 200（表示书籍 ID 不存在，添加库存失败）。

```
    def test_error_book_id(self):
        for b in self.books:
            book_id = b.id
            code = self.seller.add_stock_level(
                self.user_id, self.store_id, book_id + "_x", 10
            )
            assert code != 200
```

4. `test_ok`

- **功能**: 测试成功添加库存的场景。
- 实现:
  - 遍历 `self.books` 中的每本书，调用 `add_stock_level` 方法，将库存数量设置为 10。
  - 验证返回的状态码应为 200（表示库存添加成功）。

```
    def test_ok(self):
        for b in self.books:
            book_id = b.id
            code = self.seller.add_stock_level(self.user_id,
self.store_id, book_id, 10)
            assert code == 200
```

- ## test_search_order

1. `test_ok`

- **功能**: 测试成功创建订单并搜索订单的场景。
- 实现:
  - 买家调用 `new_order` 方法创建新订单，传入商店 ID 和书籍 ID 列表，验证返回的状态码应为 200（表示订单创建成功）。
  - 买家调用 `search_order` 方法搜索自己的订单，验证返回的状态码应为 200（表示搜索成功）。
  - 卖家也调用 `search_order` 方法搜索订单，验证返回的状态码应为 200（表示搜索成功）。

```
    def test_ok(self):
        code, self.order_id = self.buyer.new_order(self.store_id,
self.buy_book_id_list)
        assert code == 200

        code = self.buyer.search_order()
        assert code == 200

        code = self.seller.search_order()
        assert code == 200
```

2. `test_empty_order_search`

- **功能**: 测试在没有创建订单的情况下进行搜索。
- 实现:
  - 卖家调用 `search_order` 方法进行搜索，验证返回的状态码应为 525（表示没有订单）。
  - 买家同样调用 `search_order` 方法进行搜索，验证返回的状态码应为 525（表示没有订单）。

```
    def test_empty_order_search(self):
        # 不创建订单直接进行搜索
        code = self.seller.search_order()
        assert code == 525

        code = self.buyer.search_order()
        assert code == 525
```

- ## test_deliver

1. `test_no_paid`

- **功能**: 测试未支付情况下的发货功能。
- 实现:
  - 卖家调用 `deliver` 方法尝试发货，传入卖家 ID、商店 ID 和订单 ID，验证返回的状态码不为 200（表示发货失败）。

```
    def test_no_paid(self):
        code = self.seller.deliver(self.seller_id, self.store_id,
self.order_id)
        assert code != 200
```

2. `test_deliver`

- **功能**: 测试支付后正常发货的场景。
- 实现:
  - 买家调用 `add_funds` 方法充值，确保账户有足够的余额，验证返回状态码为 200（表示充值成功）。

- 买家调用 `payment` 方法进行订单支付，验证返回状态码为 200（表示支付成功）。
- 卖家再次调用 `deliver` 方法发货，验证返回状态码为 200（表示发货成功）。

```
    def test_deliver(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.seller.deliver(self.seller_id, self.store_id,
self.order_id)
        assert code == 200
```

3.3 Buyer：测试买家的各个接口

test_add_funds

1. `test_ok`

- **功能**: 测试正常充值功能。
- 实现:
  - 买家调用 `add_funds` 方法充值 1000，验证返回状态码为 200（表示充值成功）。
  - 买家再次调用 `add_funds` 方法充值 -1000，检查是否能成功处理负数充值，验证返回状态码仍为 200（表示处理成功，可能是账户余额未改变）。

```
    def test_ok(self):
        code = self.buyer.add_funds(1000)
        assert code == 200

        code = self.buyer.add_funds(-1000)
        assert code == 200
```

2. `test_error_user_id`

- **功能**: 测试无效用户 ID 情况下的充值功能。
- 实现:
  - 修改买家的 `user_id`，使其变为无效（添加后缀 "_x"）。
  - 调用 `add_funds` 方法进行充值，验证返回状态码不为 200（表示充值失败）。

```
    def test_error_user_id(self):
        self.buyer.user_id = self.buyer.user_id + "_x"
        code = self.buyer.add_funds(10)
        assert code != 200
```

3. `test_error_password`

- **功能**: 测试无效密码情况下的充值功能。
- 实现:

- 修改买家的 `password`，使其变为无效（添加后缀 "_x"）。
- 调用 `add_funds` 方法进行充值，验证返回状态码不为 200（表示充值失败）。

```
    def test_error_password(self):
        self.buyer.password = self.buyer.password + "_x"
        code = self.buyer.add_funds(10)
        assert code != 200
```

- **test_payment**

1. `test_ok`

- **功能**: 测试正常支付流程。
- 实现:
  - 买家为订单添加足够的资金，调用 `add_funds` 方法，验证返回状态码为 200（成功）。
  - 调用 `payment` 方法支付订单，验证返回状态码也为 200（成功）。

```
    def test_ok(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
```

2. `test_authorization_error`

- **功能**: 测试授权错误情况下的支付功能。
- 实现:
  - 买家添加资金并成功支付。
  - 修改买家的密码，使其无效，尝试再次支付，验证返回状态码不为 200（失败）。

```
    def test_authorization_error(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        self.buyer.password = self.buyer.password + "_x"
        code = self.buyer.payment(self.order_id)
        assert code != 200
```

3. `test_not_suff_funds`

- **功能**: 测试资金不足情况下的支付功能。
- 实现:
  - 买家添加少于订单总金额的资金，验证状态码为 200（成功）。
  - 尝试支付订单，验证返回状态码不为 200（失败）。

```
    def test_not_suff_funds(self):
        code = self.buyer.add_funds(self.total_price - 1)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code != 200
```

4. `test_repeat_pay`

- **功能**: 测试重复支付相同订单的情况。
- 实现:
  - 买家添加足够的资金并成功支付订单。
  - 尝试再次支付相同订单，验证返回状态码不为 200（失败）。

```
    def test_repeat_pay(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200

        code = self.buyer.payment(self.order_id)
        assert code != 200
```

- ## test_new_order

1. `test_non_exist_book_id`

- **功能**: 测试使用不存在的书籍 ID 创建订单。
- 实现:
  - 调用 `gen` 方法生成包含不存在书籍 ID 的书籍列表。
  - 尝试使用这些书籍 ID 创建订单，验证返回状态码不为 200（失败）。

```
    def test_non_exist_book_id(self):
        ok, buy_book_id_list = self.gen_book.gen(
            non_exist_book_id=True, low_stock_level=False
        )
        assert ok
        code, _ = self.buyer.new_order(self.store_id,
buy_book_id_list)
        assert code != 200
```

2. `test_low_stock_level`

- **功能**: 测试使用库存不足的书籍创建订单。
- 实现:
  - 调用 `gen` 方法生成包含库存不足书籍的书籍列表。
  - 尝试创建订单，验证返回状态码不为 200（失败）。

```
    def test_low_stock_level(self):
        ok, buy_book_id_list = self.gen_book.gen(
            non_exist_book_id=False, low_stock_level=True
        )
        assert ok
        code, _ = self.buyer.new_order(self.store_id,
buy_book_id_list)
        assert code != 200
```

3. `test_ok`

- **功能**: 测试正常情况下创建订单的功能。
- 实现:
  - 调用 `gen` 方法生成有效书籍 ID 的书籍列表。
  - 尝试创建订单，验证返回状态码为 200（成功）。

```
    def test_ok(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
```

4. `test_non_exist_user_id`

- **功能**: 测试使用不存在的用户 ID 创建订单。
- 实现:
  - 调用 `gen` 方法生成有效书籍 ID 的书籍列表。
  - 修改买家的用户 ID，使其无效，尝试创建订单，验证返回状态码不为 200（失败）。

```
    def test_non_exist_user_id(self):
        ok, buy_book_id_list = self.gen_book.gen(
            non_exist_book_id=False, low_stock_level=False
        )
        assert ok
        self.buyer.user_id = self.buyer.user_id + "_x"
        code, _ = self.buyer.new_order(self.store_id,
buy_book_id_list)
        assert code != 200
```

5. `test_non_exist_store_id`

- **功能**: 测试使用不存在的店铺 ID 创建订单。
- 实现:
  - 调用 `gen` 方法生成有效书籍 ID 的书籍列表。

- 尝试使用不存在的店铺 ID 创建订单，验证返回状态码不为 200（失败）。

```
    def test_non_exist_store_id(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 + "_x",
buy_book_id_list)
        assert code != 200
```

- **test_book_search**

1. `test_book_id`

- **功能**: 测试通过书籍 ID 搜索书籍。
- 实现:
  - 遍历已添加的书籍，获取书籍 ID。
  - 调用买家的 `book_search` 方法以书籍 ID 进行搜索，验证返回状态码为 200（成功）。

```
    def test_book_id(self):
        for bk in self.books:
            self.book_id = bk.__dict__.get("id")
            code = self.buyer.book_search(self.search_store_id,
self.book_id, self.book_title, self.book_tags,
                                          self.book_author)
            assert code == 200
```

2. `test_book_title`

- **功能**: 测试通过书籍标题搜索书籍。
- 实现:
  - 遍历已添加的书籍，获取书籍标题。
  - 调用买家的 `book_search` 方法以书籍标题进行搜索，验证返回状态码为 200（成功）。

```
    def test_book_title(self):
        for bk in self.books:
            self.book_title = bk.__dict__.get("title")
            code = self.buyer.book_search(self.search_store_id,
self.book_id, self.book_title, self.book_tags,
                                          self.book_author)
            assert code == 200
```

3. `test_book_tags`

- **功能**: 测试通过书籍标签搜索书籍。
- 实现:
  - 遍历已添加的书籍，获取书籍标签。

- 调用买家的 `book_search` 方法以书籍标签进行搜索，验证返回状态码为 200（成功）。

```
    def test_book_tags(self):
        for bk in self.books:
            self.book_tags = bk.__dict__.get("tags")
            code = self.buyer.book_search(self.search_store_id,
self.book_id, self.book_title, self.book_tags,
                                          self.book_author)
            assert code == 200
```

4. `test_book_author`

- **功能**: 测试通过书籍作者搜索书籍。
- 实现:
  - 遍历已添加的书籍，获取书籍作者。
  - 调用买家的 `book_search` 方法以书籍作者进行搜索，验证返回状态码为 200（成功）。

```
    def test_book_author(self):
        for bk in self.books:
            self.book_author = bk.__dict__.get("author")
            code = self.buyer.book_search(self.search_store_id,
self.book_id, self.book_title, self.book_tags,
                                          self.book_author)
            assert code == 200
```

5. `test_book_in_store`

- **功能**: 测试在店铺中搜索书籍。
- 实现:
  - 设置搜索的店铺 ID 为当前卖家的店铺 ID，获取书籍 ID。
  - 调用买家的 `book_search` 方法以书籍 ID 进行搜索，验证返回状态码为 200（成功）。

```
    def test_book_in_store(self):
        for bk in self.books:
            self.search_store_id = self.store_id
            self.book_id = bk.__dict__.get("id")
            code = self.buyer.book_search(self.search_store_id,
self.book_id, self.book_title, self.book_tags,
                                          self.book_author)
            assert code == 200
```

6. `test_book_not_exist`

- **功能**: 测试搜索不存在的书籍。
- 实现:
  - 遍历已添加的书籍，构造一个不存在的书籍 ID（在原有 ID 后添加 "000"）。

- 调用买家的 `book_search` 方法以不存在的书籍 ID 进行搜索，验证返回状态码不为 200（失败）。

```
    def test_book_not_exist(self):
        for bk in self.books:
            self.book_id = bk.__dict__.get("id") + "000"
            code = self.buyer.book_search(self.search_store_id,
self.book_id, self.book_title, self.book_tags,
                                          self.book_author)
            assert code != 200
```

- **test_search_order**

1. `test_ok`

- **功能**：测试订单的成功创建和搜索功能。
- 实现：
  - 使用买家创建一个新订单，并断言返回代码为 200（成功）。
  - 分别使用买家和卖家搜索订单，断言两个返回结果也都为 200。

```
    def test_ok(self):
        code, self.order_id = self.buyer.new_order(self.store_id,
self.buy_book_id_list)
        assert code == 200

        code = self.buyer.search_order()
        assert code == 200

        code = self.seller.search_order()
        assert code == 200
```

2. `test_empty_order_search`

- **功能**：测试在没有订单的情况下搜索功能。
- 实现：
  - 在不创建任何订单的情况下，直接调用卖家和买家的搜索订单方法，并断言返回的错误代码为 525，表示未找到订单。

```
    def test_empty_order_search(self):
        # 不创建订单直接进行搜索
        code = self.seller.search_order()
        assert code == 525

        code = self.buyer.search_order()
        assert code == 525
```

- ## test_delete_order

1. `test_delete_ok`

- **功能**：测试成功删除订单的流程。
- 实现：
  - 创建卖家的店铺并生成书籍，确保书籍信息有效。
  - 买家创建一个新订单，并断言返回代码为 200（成功）。
  - 调用买家的 `delete_order` 方法删除订单，断言返回代码为 200，表示删除成功。

```
    def test_delete_ok(self):
        # 创建卖家店铺并读入书籍
        gen_book = GenBook(self.seller_id, self.store_id)
        ok, buy_book_id_list = gen_book.gen(
            non_exist_book_id=False, low_stock_level=False,
max_book_count=5
        )
        self.buy_book_info_list = gen_book.buy_book_info_list
        assert ok

        code, self.order_id = self.buyer.new_order(self.store_id,
buy_book_id_list)
        assert code == 200

        # 删除buyer
        code = self.buyer.delete_order(self.buyer_id, self.order_id)
        assert code == 200
```

2. `test_non_order_delete`

- **功能**：测试尝试删除不存在的订单时的错误处理。
- 实现：
  - 调用买家的 `delete_order` 方法，使用一个无效的订单 ID 删除订单，并断言返回的错误代码为 520，表示没有找到订单。

```
    def test_non_order_delete(self):
        # 删除buyer
        code = self.buyer.delete_order(self.buyer_id,
"test_delete_order_buyer_id")
        assert code == 520  # error code of non_order_delete
```

- ## test_payment_overtime

1. `test_delete_payment_overtime`

- **功能**：测试订单超时未付款的情况。
- 实现：
  - 买家创建一个新订单，并断言返回代码为 200（成功）。

- 买家搜索订单，确认订单存在。
- 等待 20 秒后，再次搜索订单，断言返回代码为 525，表示订单已超时且被删除。

```
    def test_delete_payment_overtime(self):
        code, self.order_id = self.buyer.new_order(self.store_id,
self.buy_book_id_list)
        assert code == 200

        # 搜索时执行删除超时订单
        code = self.buyer.search_order()
        assert code == 200

        # 等待20秒
        time.sleep(20)

        # 搜索时执行删除超时订单
        code = self.buyer.search_order()
        assert code == 525
```

2. `test_payment_completely`

- **功能**：测试完成付款后订单未删除的情况。
- 实现：
  - 买家创建一个新订单，并断言返回代码为 200。
  - 计算订单总价并为买家账户添加资金，确保操作成功。
  - 买家搜索订单，确认订单存在。
  - 买家完成付款，并断言付款成功。
  - 等待 20 秒后，再次搜索订单，确认已付款的订单仍然存在，返回代码为 200。

```
    def test_payment_completely(self):
        code, self.order_id = self.buyer.new_order(self.store_id,
self.buy_book_id_list)
        assert code == 200

        self.total_price = 0
        for item in self.buy_book_info_list:
            book: Book = item[0]
            num = item[1]
            if book.price is None:
                continue
            else:
                self.total_price = self.total_price + book.price * num

        code = self.buyer.add_funds(self.total_price)
        assert code == 200

        # 刚下单成功订单存在
        code = self.buyer.search_order()
```

```
        assert code == 200

        # 完成付款
        code = self.buyer.payment(self.order_id)
        assert code == 200

        # 等待20秒
        time.sleep(20)

        # 已付款订单未删除,仍存在
        code = self.buyer.search_order()
        assert code == 200
```

- **test_receive**

1. `test_no_paid`

- **功能**：测试在未付款情况下接收订单的行为。
- 实现：
  - 调用买家接收订单的方法，断言返回代码不等于 200，表示接收失败。

```
    def test_no_paid(self):
        code = self.buyer.receive(self.buyer_id, self.store_id,
self.order_id)
        assert code != 200
```

2. `test_paid_no_deliver`

- **功能**：测试在已付款但未发货的情况下接收订单的行为。
- 实现：
  - 买家为账户添加资金并完成付款，确保返回代码为 200。
  - 调用接收订单的方法，断言返回代码不等于 200，表示接收失败。

```
    def test_paid_no_deliver(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.buyer.receive(self.buyer_id, self.store_id,
self.order_id)
        assert code != 200
```

3. `test_receive`

- **功能**：测试在完成付款并发货后接收订单的成功情况。
- 实现：
  - 买家为账户添加资金并完成付款，确保返回代码为 200。
  - 卖家发货，并确保返回代码为 200。
  - 买家调用接收订单的方法，断言返回代码为 200，表示接收成功。

```
    def test_receive(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.seller.deliver(self.seller_id, self.store_id,
self.order_id)
        assert code == 200
        code = self.buyer.receive(self.buyer_id, self.store_id,
self.order_id)
        assert code == 200
```

- **test_bench效率测试**

  通过调用插入书本的后端插入书本内容到 MongoDB 的数据库中，然后通过大量线程同时调用下订 单和付款的后端接口，来测试数据库的吞吐量

测试结果与测试覆盖率

MINGW64:/e/DB_bookstore

```
INFO     werkzeug:_internal.py:225 127.0.0.1 - - [31/Oct/2024 00:46:45] "POST /b
uyer/new_order HTTP/1.1" 200 -
---------------------------- Captured stderr call -----------------------------
2024-10-31 00:46:45,496 [Thread-3946 ] [INFO ]  127.0.0.1 - - [31/Oct/2024 00:46
:45] "POST /buyer/add_funds HTTP/1.1" 200 -
2024-10-31 00:46:45,508 [Thread-3947 ] [INFO ]  127.0.0.1 - - [31/Oct/2024 00:46
:45] "POST /buyer/payment HTTP/1.1" 200 -
2024-10-31 00:46:45,520 [Thread-3948 ] [INFO ]  127.0.0.1 - - [31/Oct/2024 00:46
:45] "POST /buyer/receive HTTP/1.1" 200 -
------------------------------ Captured log call ------------------------------
INFO     werkzeug:_internal.py:225 127.0.0.1 - - [31/Oct/2024 00:46:45] "POST /b
uyer/add_funds HTTP/1.1" 200 -
INFO     werkzeug:_internal.py:225 127.0.0.1 - - [31/Oct/2024 00:46:45] "POST /b
uyer/payment HTTP/1.1" 200 -
INFO     werkzeug:_internal.py:225 127.0.0.1 - - [31/Oct/2024 00:46:45] "POST /b
uyer/receive HTTP/1.1" 200 -
=========================== short test summary info ===========================
FAILED fe/test/test_receive.py::Test_Receive::test_paid_no_deliver - assert 2...
=================== 1 failed, 49 passed in 97.93s (0:01:37) ===================
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\__init__.py                        0      0      0      0   100%
be\model\buyer.py                         133     25     50      7    83%
be\model\db_conn.py                        19      0      6      0   100%
be\model\error.py                          39      8      0      0    79%
be\model\seller.py                         78     15     28      5    81%
be\model\store.py                          25      0      0      0   100%
be\model\user.py                          107     17     32      6    83%
be\serve.py                                35      1      2      1    95%
be\view\__init__.py                         0      0      0      0   100%
be\view\auth.py                            42      0      0      0   100%
be\view\buyer.py                           65      0      2      0   100%
be\view\seller.py                          45      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                          70      1     12      2    96%
fe\access\buyer.py                         61      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                        43      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     22      2    98%
fe\conf.py                                 11      0      0      0   100%
fe\conftest.py                             17      0      0      0   100%
fe\test\gen_book_data.py                   49      0     16      0   100%
fe\test\test_add_book.py                   37      0     10      0   100%
fe\test\test_add_funds.py                  23      0      0      0   100%
fe\test\test_add_stock_level.py            40      0     10      0   100%
fe\test\test_bench.py                       6      2      0      0    67%
fe\test\test_book_search.py                58      0     14      0   100%
fe\test\test_create_store.py               20      0      0      0   100%
fe\test\test_delete_order.py               27      0      0      0   100%
fe\test\test_deliver.py                    50      1      4      1    96%
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_payment_overtime.py           48      1      4      1    96%
fe\test\test_receive.py                    59      1      4      1    97%
fe\test\test_register.py                   31      0      0      0   100%
fe\test\test_search_order.py               31      0      0      0   100%
-------------------------------------------------------------------------
TOTAL                                    1665     77    242     28    94%
Wrote HTML report to htmlcov\index.html

Bachopininoff@Bachmozaninoff MINGW64 /e/DB_bookstore (main)
$
```

Git版本控制

```
Bachopininoff@Bachmozaninoff MINGW64 /e/CDMS.Xuan_ZHOU.2024Fall.DaSE/project1/bo
okstore (master)
$ git remote add origin https://github.com/Sakura-Hydrangea/database_lab

Bachopininoff@Bachmozaninoff MINGW64 /e/CDMS.Xuan_ZHOU.2024Fall.DaSE/project1/bo
okstore (master)
$ git push -u origin master
Enumerating objects: 206, done.
Counting objects: 100% (206/206), done.
Delta compression using up to 20 threads
Compressing objects: 100% (199/199), done.
Writing objects: 100% (206/206), 7.04 MiB | 7.22 MiB/s, done.
Total 206 (delta 37), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (37/37), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/Sakura-Hydrangea/database_lab/pull/new/master
remote:
To https://github.com/Sakura-Hydrangea/database_lab
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

项目地址：[Sakura-Hydrangea/database_lab: lab for database course]

总结:

本实验实现了完整的书店数据库系统，具有较好的代码可读性和覆盖率，项目涵盖了从买家注册、订单创建、支付到订单接收的完整流程，体现了电商系统中常见的业务逻辑。通过允许用户自定义卖家和买家信息，增强了代码的灵活性，使得测试可以在多种情况下进行。