- 当代数据管理系统项目 : Bookstore
  - 文档数据数据库设计
  - 具体功能实现
    - be/model后端逻辑代码
    - buyer
      - payment
      - add_funds
      - book_search
      - delete_order
      - search_order
      - receive
    - user
      - 初始化
      - 检查 Token
      - 用户注册
      - 检查密码
      - 用户登录
      - 用户登出
      - 用户注销
      - 修改密码
    - seller
      - add_book
      - add_stock_level
      - createStore
      - search_order
      - deliver
    - be/view: 访问后端接口
      - buyer
      - 新建订单接口(/buyer/new_order)
      - 付款接口(/buyer/payment)
      - 添加资金接口 (/buyer/add_funds)
      - 图书查询接口 (/buyer/search_book)
      - 删除订单接口(/buyer/delete_order)
      - 查询订单接口(/buyer/search_order)
      - 确认收货接口(/buyer/receive)
    - Auth
      - Login
      - logout
      - register
      - unregister
      - change_password
    - Seller
      - seller_create/store
      - seller_add_book
      - add_stock_level
      - search_order
      - deliver
    - fe/test:功能性测试
    - User:测试使用者各个接口
      - test_login
      - test.ok
      - test_error_user_id

- test_error_password
- test register
- test register ok
- test_unregister.ok
- test_unregister_error_authorization
- test register error exist user_id

test_password
- test.ok
- test_error_password
- test_error_user_id

Seller：测试卖家的各个接口
- test_create/store
- test.ok
- test_error_exists_store_id

test_add_book
- test.ok
- test_error_non existed_STORE_id
- test_error_exists_book_id
- test_error_non existed_user_id

test_add_stock_level
- test_error_user_id
- test_error/store_id
- test_error_book_id
- test.ok

test_search_order
- test.ok
- test_empty_order_search

test_deliver
- test_no_paid
- test_deliver

Buyer：测试买家的各个接口

test_add_funds
- test.ok
- test_error_user_id
- test_error_password

test-payment
- test.ok
- test_authorization_error
- test_not_suff_funds
- testrepeat_pay

test_new_order
- test_non Exist_book_id
- test_low_stock_level
- test.ok
- test_non Exist_user_id
- test_non existed_store_id

test_book_search
- test_book_id
- test_book_title
- test_booktags
- test_book_author
- test_book_in_store
- test_book_not_exit

1. 当代数据管理系统项目：Bookstore

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：张欣扬	学号：10235501413	负责用户权限接口（注册登录等）+ 订单状态查询取消 + 搜索图书功能 + 快速索引功能
姓名：贾馨雨	学号：10235501437	负责买家用户接口（充值下单等）+ 卖家用户接口（创建店铺等）+ 发货收货流程 + 覆盖率提升

我们基于Flask和MongoDB构建了一个完整的在线书店系统，完成了从SQLite到MongoDB的数据迁移，设计了合理的数据库结构，并通过建立复合索引优化了查询性能。系统实现了用户注册登录、图书搜索、下单购买、支付结算、发货收货等完整业务流程，订单系统支持状态跟踪和超时自动取消，搜索功能支持多条件查询。测试覆盖率达到94%，通过新增完整的错误处理测试，覆盖了重复支付、数据库操作失败、授权校验等异常场景，提升了系统的健壮性和可靠性。除了完成要求的所有功能之外，我们还通过索引优化提升了系统性能。

1.1 文档数据数据库设计

- **new_order:**

  订单主表用于存储订单的基本信息和状态跟踪。每个文档包含订单的唯一标识、关联的店铺和用户信息、支付状态以及订单金额等关键字段。其中order_id设置为唯一索引确保订单号的唯一性，payment_ddl采用Unix时间戳格式记录支付截止时间，payment_status通过状态值（如"no_pay"、"paid"、"shipped"等）来跟踪订单。

```
{
     "_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:**

  订单明细表与订单主表形成关联关系，详细记录每个订单中的商品信息。每个文档对应一个订单中的特定商品，包含书籍ID、购买数量和单价信息。order_id和book_id均设置为索引，便于快速查询订单商品明细和商品销售记录。

```
{
     "_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:**

  用户表管理平台用户账户信息，包含用户身份验证所需的凭证和账户财务数据。

  user_id作为唯一索引确保用户标识的唯一性，password字段采用加密存储，balance记录用户余额用于交易扣款，token和terminal字段支持用户会话管理和设备识别。

```
{
    "_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：**

  店铺商品表存储各店铺的商品库存和商品信息，每个文档代表一个店铺中的特定商品，包含完整的商品信息和库存数量。

  store_id和book_id设置为索引，支持高效的店铺商品查询和库存管理。booktags字段采用数组形式存储多个标签，便于商品分类和检索。

```
{
    "_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
    "booktags": ["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**

  用户店铺关系表建立用户与店铺之间的关联关系，用于权限管理和关系映射。通过store_id和user_id两个索引字段，高效支持"用户-店铺"双向查询。

```
{
    "_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）
}
```

1.2 具体功能实现

1.2.1 be/model后端逻辑代码

1.2.2 buyer

- **init**

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

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

- **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_idexist(user_id):
            return error(error_non existed_user_id用户的 + order_id,)
        if not self.store_idexist/store_id):
            return error(error_non existedstore_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 = selfconn 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 existed_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 = selfconn 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.connect_new_order/detail_col.insert_one(
            {"order_id": uid,"book_id": book_id,"count": count,"price": price}
        )
            # 计算总价格
            total_price += price * count

        # 设置订单支付截止时间
        current_time = int(time.time())
        payment_ddl = current_time + 15

        try:
            # 插入订单基础信息
            self.connect.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
```

1.2.2.1 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 Exception as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))

    return 200, "ok"
```

1.2.2.2 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 = selfconn.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
        selfconn.user_col.update_one({'user_id': user_id}, {"$inc": {"balance":
add_value}]
except BaseException as e:
    return 530, {"}.format(str(e))
return 200, "ok"
```

1.2.2.3 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"] = {"$in": [book_tags]}
        if book_author:
            query_conditions["book_author"] = {"$regex": book_author, "$options": "i"}
        # books = self.conn.book_col.find(query_conditions)
        # return 200, list(books)

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

    return 200, "ok"
```

1.2.2.4 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 = selfconn.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
            selfconn.user_col.update_one({'user_id': user_id},{"$inc": {"balance":
total_price}})
            # 扣减卖家余额
            selfconn.user_col.update_one({'user_id':order["store_id"]}, {"$inc":
{"balance": -total_price}})
            # 删除订单及详情
            selfconn.new_order_col.delete_one({'order_id': order_id})
            selfconn.new_order_detail_col.delete_many({'order_id':order_id})
```

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

1.2.2.5 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 = selfconn.new_order_col.find(
            {"user_id": user_id, "payment_ddl": {"$lt": current_time},"payment_status":
"no_pay"}
        )
        expired_order_ids = [order["order_id"] for order in expired Orders]
        selfconn.new_order_col.delete_many({"order_id": {"$in": expired_order_ids}})
        selfconn.new_order_detail_col.delete-many({"order_id": {"$in":
expired_order_ids}})
        # 返回用户所有订单
        orders = list(selfconn.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"
```

1.2.2.6 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 = selfconn.new_order_col.find_one(['order_id':order_id])
        if not order or order["user_id"] != user_id:
            return error(error_non existed_user_id(user_id))
        if order["store_id"] != store_id:
            return error(error_non existed/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.connect.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"
```

1.2.3 user

User 类负责处理用户身份验证与账户管理，为系统提供安全的访问控制。该类通过 JWT 令牌机制管理用户会话，并封装了完整的用户生命周期操作。其初始化过程建立了必要的数据库连接。核心的令牌验证方法 `__check_token` 负责解密传入的 JWT 令牌，并校验其签名有效性与时间戳，确保会话安全。

在用户账户管理方面，`register` 方法执行新用户注册，在插入新用户文档前会先行检查用户 ID 是否已存在，以此保证唯一性。`login` 方法在验证用户密码正确后，会生成一个新的 JWT 令牌并更新至数据库，完成登录流程。与之对应的 `logout` 方法则通过将用户令牌替换为一个新生成的无效令牌，使其原会话失效。系统同时提供了关键的安全维护功能。`check_password` 用于核验密码是否正确`change_password` 方法要求用户提供旧密码以通过身份验证，随后才允许更新为新密码，并同时刷新用户令牌。而 `unregister` 方法在执行最终的用户文档删除前，同样会进行严格的密码确认，确保账户注销操作的安全性。

1.2.3.1 初始化

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

1.2.3.2 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(encrypted_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 JWTexceptions.InvalidSignatureError as e:
        logging.error(str(e))
    return False
```

```
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(errorAuthorization_fail())
    return 200, "ok"
```

1.2.3.3 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 = selfconn.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, term)
    # 确定无冲突后执行插入
    self.connect.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"
```

1.2.3.4 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"
```

1.2.3.5 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(ucer_id, terminal)
        result = self.connect_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
```

1.2.3.6 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.connect.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"
```

1.2.3.7 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)
```

1.2.3.8 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)
```

1.2.4 seller

Seller 类为卖家提供了店铺运营所需的核心功能，包括商品管理、库存维护和订单处理。

在商品管理方面，`add_book` 方法负责上架新书籍。该方法会验证用户权限与店铺有效性，并确保书籍编号在店铺内的唯一性。验证通过后，将包含书籍标题、作者、价格、标签及库存量等完整信息的文档插入数据库。`add_stock_level` 方法则专注于库存补充，通过 MongoDB 的 `$inc` 操作符对指定书籍的库存量进行原子性增加。

店铺体系的管理由 `create_store` 方法实现。它在创建新店铺前会校验用户身份的合法性及店铺 ID 的未被占用性，随后在 `user_store` 集合中建立用户与店铺的归属关系。

订单处理功能包含查询与发货两个关键操作。`search_order` 方法首先会清理系统中所有超时未支付的订单，随后基于卖家所拥有的店铺 ID 列表，检索出与该卖家相关的所有订单。`deliver` 方法负责执行发货流程，它会严格校验订单状态，确保订单处于“已支付”状态且未被发货后，才将订单状态更新为“已发货”，从而推动订单进入下一个环节。

1.2.4.1 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)
        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
```

1.2.4.2 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"
```

1.2.4.3 createStore

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userstore_col集合中。

```
    def create_store(self, user_id: str, store_id: str) -> (int, str):
        try:
            if not self.user_idexist(user_id):
                return error(error_nonEXIST_user_id(user_id))
            if self.store_idexist(storage_id):
                return error(errorEXIST_store_id):
            data = {
                "store_id": store_id,
                "user_id": user_id
            }
            self.connect_user/store_col.insert_one(data)

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

1.2.4.4 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"
```

1.2.4.5 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 = selfconn.new_order_col.find_one({'order_id': order_id})
        if result is None:
            return error(error_non existed_order(order_id)
        if not self.user_idexist(user_id):
            return error(error_non existed_user_id(user_id))
        if not self.store_idexist/store_id):
            return error(error_non existedstore_id/store_id)
        result = selfconn.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"
```

1.2.5 be/view: 访问后端接口

1.2.5.1 buyer

1.2.5.2 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
```

1.2.5.3 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
```

1.2.5.4 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。

```python
@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
```

1.2.5.5 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_labels = request.json.get("book_labels")
    book_author = request.json.get("book_author")
    b = Buyer()
    code, message = b.book_search(
        store_id, book_id, book_title, book_labels, book_author)
    return jsonify({'message': message}), code
```

1.2.5.6 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
```

1.2.5.7 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
```

1.2.5.8 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
```

1.2.6 Auth

1.2.6.1 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.logout(
        user_id=user_id, password=password, terminal=terminal)
    return jsonify({"message": message, "token": token}), code
```

1.2.6.2 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。

```
@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
```

1.2.6.3 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
```

1.2.6.4 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
```

1.2.6.5 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
```

1.2.7 Seller

1.2.7.1 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 = request.json.get("user_id")
    store_id = request.json.get("store_id")
    s = seller.Seller()
    code, message = s.create_store(user_id, store_id)
    return jsonify({"message": message}), code
```

1.2.7.2 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。

```
@bpSellerroute("/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 = sellerSeller()
    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)

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

1.2.7.3 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。

```
@bpSeller.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 = sellerSeller()
    code, message = s.add_stock_level(user_id, store_id, book_id, add_num)

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

1.2.7.4 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。

```
@bpSellerroute("/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
```

1.2.7.5 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。
  - 返回发处理结果的消息和状态码。

```
@bpSeller.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 = sellerSeller()
    code, message = s.deliver(user_id, store_id, order_id)  # 修复了sdeliver的拼写错误（缺少点.）
    return jsonify({"message": message}), code
```

1.2.8 fe/test:功能性测试

1.2.9 User:测试使用者各个接口

1.2.9.1 test_login

1.2.9.2 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, selfterminal)
    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
```

1.2.9.3 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, selfterminal)
    assert code == 401test_error_password
```

1.2.9.4 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.root)
    assert code == 401
```

1.2.9.5 test register

1.2.9.6 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
```

1.2.9.7 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
```

1.2.9.8 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
```

1.2.9.9 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_errorEXIST_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
```

1.2.10 test_password

1.2.10.1 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)  #
修复连字符和方法调用
    assert code != 200

    code, new_token = self.auth.login(self.user_id, self.new_password, self.terminal)  #
修复波浪线和连字符
    assert code == 200

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

1.2.10.2 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
```

1.2.10.3 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
```

1.2.11 Seller：测试卖家的各个接口

1.2.11.1 test_create/store

1.2.11.2 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):
    selfseller = register_newSeller(self.user_id, self.password)
    code = selfseller.create_store(self.store_id)
    assert code == 200
```

1.2.11.3 test_error_exists_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_errorEXIST_STORE_id(self):
    selfseller = register_newSeller(self.user_id, self.password)
    code = selfSeller.create_store(self.store_id)
    assert code == 200
    code = selfSeller.create_store(self.store_id)
    assert code != 200
```

1.2.12 test_add_book

1.2.12.1 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 = selfseller.add_book(self/store_id, 0, b)
        assert code == 200
```

1.2.12.2 test_error_non existed_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:
        # 非存在的商店ID（在原store_id后加"x"）
        code = self.seller.add_book(self.store_id + "x", 0, b)
        assert code != 200
```

1.2.12.3 test_error_exists_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 = selfseller.add_book(self.store_id, 0, b)
        assert code == 200
    for b in self.books:
        # exist book id
        code = selfseller.add_book(self.store_id, 0, b)
        assert code != 200
```

1.2.12.4 test_error_non existed_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:
        # 非存在的用户ID（在原user_id后加"_x"）
        # 假设原用户ID变量为self.seller_seller_id，这里构造错误ID
        invalid_user_id = self.seller_seller_id + "_x"
        # 注意：此处需确认add_book方法是否需要传入user_id参数，以下为示例
        code = self.seller.add_book(invalid_user_id, self.store_id, 0, b)
        assert code != 200
```

1.2.13 test_add_stock_level

1.2.13.1 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 = selfseller.add_stock_level(
            self.user_id + "_x", self.store_id, book_id, 10
        )
    assert code != 200
```

1.2.13.2 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，传入不存在的商品 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_errorstore_id(self):
    for b in self.books:
        book_id = b.id
        code = selfseller.add_stock_level(
            self.user_id, self/store_id + "_x", book_id, 10
        )
    assert code != 200
```

1.2.13.3 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 = selfseller.add_stock_level(
            self.user_id, self/store_id, book_id + "_x", 10)
        assert code != 200
```

1.2.13.4 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 = selfSeller.add_stock_level(self.user_id, self/store_id, book_id, 10)
        assert code == 200
```

1.2.14 test_search_order

1.2.14.1 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.buyers.new_order(self/store_id, self.buy_book_id_list)
    assert code == 200
    code = self.buyers.search_order()
    assert code == 200
    code = selfseller.search_order()
    assert code == 200
```

1.2.14.2 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 = selfseller.search_order()
    assert code == 525
    code = selfbuyer.search_order()
    assert code == 525
```

1.2.15 test_deliver

1.2.15.1 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 = selfsellerdeliver(self seller_id, self/store_id, self.order_id)
    assert code != 200
```

1.2.15.2 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 testdeliver(self):
    code = self.buyer.add_funds(self.total_price)
    assert code == 200
    code = self.buyer.payment(self.order_id)
    assert code == 200
    code = selfsellerdeliver(self seller_id, self/store_id, self.order_id)
    assert code == 200
```

1.2.16 Buyer：测试买家的各个接口

1.2.17 test_add_funds

1.2.17.1 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
```

1.2.17.2 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
```

1.2.17.3 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
```

1.2.18 test-payment

1.2.18.1 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
```

1.2.18.2 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
```

1.2.18.3 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
```

1.2.18.4 testrepeat_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 testrepeat_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
```

1.2.19 test_new_order

1.2.19.1 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(
        nonExist_book_id=True, low_stock_level=False
    )
    assert ok
    code, _ = self.buyers.new_order(self/store_id, buy_book_id_list)
    assert code != 200
```

1.2.19.2 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
```

1.2.19.3 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.buyers.new_order(self.store_id, buy_book_id_list)
    assert code == 200
```

1.2.19.4 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.shop_id, buy_book_id_list)
    assert code != 200
```

1.2.19.5 test_non existed_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(
        nonExist_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
```

1.2.20 test_book_search

1.2.20.1 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.searchstore_id, self.book_id, self.book_title,
self.book_labels, self.book_author)
        assert code == 200
```

1.2.20.2 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.searchstore_id, self.book_id, self.book_title,
self.book_labels, self.book_author)
        assert code == 200
```

1.2.20.3 test_booktags

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.searchstore_id, self.book_id, self.book_title,
self.book-tags, self.book_author)
        assert code == 200
```

1.2.20.4 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.searchstore_id, self.book_id, self.book_title,
self.book_labels, self.book_author)
        assert code == 200
```

1.2.20.5 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.searchstore_id = self.store_id
        self.book_id = bk.__dict__.get("id")
        code = self.buyer.book_search(self.searchstore_id, self.book_id, self.book_title,
self.book_labels, self.book_author)
        assert code == 200
```

1.2.20.6 test_book_not_exit

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.searchstore_id,
                        self.book_id, self.book_title, self.book_labels,
                        self.book_author)
        assert code != 200
```

1.2.21 test_search_order

1.2.21.1 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.buyers.new_order(self/store_id, self.buy_book_id_list)
    assert code == 200
    code = self.buyers.search_order()
    assert code == 200
    code = selfseller.search_order()
    assert code == 200
```

1.2.21.2 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 = selfseller.search_order()
    assert code == 525
    code = selfbuyer.search_order()
    assert code == 525
```

1.2.22 test_delete_order

1.2.22.1 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(selfseller_id, self/store_id)
    ok, buy_book_id_list = gen_book.gen(
        nonEXIST_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.buy.new_order(self/store_id, buy_book_id_list)
        assert code == 200
        # 删除buyer
        code = self.buy(delete_order(self.buy_id, self.order_id))
        assert code == 200
```

1.2.22.2 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
```

1.2.23 test_payment_overtime

1.2.23.1 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
```

1.2.23.2 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
```

1.2.24 test_receive

1.2.24.1 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
```

1.2.24.2 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
```

1.2.24.3 test_received

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_received(self):
    code = self.buyer.add_funds(self.total_price)
    assert code == 200
    code = self.buyer.payment(self.order_id)
    assert code == 200
    code = self.sellerdeliver(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
```

1.3 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

1.4 新增：快速索引功能

1.4.1 整体架构

1.快速索引功能通过 **MongoDB 索引机制**实现，主要包括两个层面：

1. **索引创建层**：在数据库初始化时创建多种类型的索引
2. **索引使用层**：在搜索查询时利用索引加速查询

2.具体来说，快速索引功能通过以下方式实现：

1. **初始化阶段：** 在 `Store.init_collections()` 中创建多种索引
    - 单字段索引：加速精确查询
    - 文本索引：支持全文搜索
    - 多键索引：支持数组字段查询
2. **查询阶段：** 在 `Buyer.book_search_advanced()` 中使用索引

- `$text` 查询触发文本索引
- 组合条件利用多个索引协同工作
- 相关性分数支持智能排序

3. **性能提升：** 索引将查询时间从毫秒级优化到微秒级，特别是在大数据集上效果显著

1.4.2 索引创建（`be/model/store.py`）

新增方法： `init_collections()`

1.4.2.1 创建的索引类型

1.4.2.1.1 单字段索引（快速精确查询）

```
# 书名索引
self.db.store.create_index([("book_title", pymongo.ASCENDING)])

# 作者索引
self.db.store.create_index([("book_author", pymongo.ASCENDING)])

# 标签索引（支持数组字段的多键索引）
self.db.store.create_index([("book_tags", pymongo.ASCENDING)])
```

作用：

- 加速按书名、作者、标签的精确查询
- `book_tags` 使用 **多键索引**，支持数组字段的高效查询

1.4.2.1.2 复合文本索引（全文搜索）

```
# 文本索引：跨字段关键词搜索
self.db.store.create_index([
    ("book_title", pymongo.TEXT),
    ("book_author", pymongo.TEXT),
    ("book_tags", pymongo.TEXT),
], name="store_text_idx")
```

作用：

- 在 `book_title`、`book_author`、`book_tags` 三个字段上创建**复合文本索引**
- 支持跨字段的关键词全文搜索
- 自动计算相关性分数

1.4.2.1.3 唯一索引（数据完整性）

```
# 复合唯一索引：确保同一店铺中图书ID唯一
self.db.store.create_index([("store_id", pymongo.ASCENDING), ("book_id",
pymongo.ASCENDING)], unique=True)
```

1.4.2.2 异常处理

```
try:
    self.db.store.create_index([...], name="store_text_idx")
except Exception:
    # 忽略索引已存在或不支持的情况
    pass
```

原因： 防止重复创建索引导致错误

1.4.3 索引使用（`be/model/buyer.py`）

1.4.3.1 基础搜索（`book_search`）

使用场景： 精确字段匹配

实现方式：

- 使用单字段索引加速查询
- 通过 `$regex` 进行模糊匹配（大小写不敏感）

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

索引利用：

- `book_title` 索引：加速书名正则匹配
- `book_tags` 多键索引：加速标签数组查询

1.4.3.2 高级搜索（`book_search_advanced`）

使用场景： 关键词全文搜索 + 多条件过滤 + 分页排序

1.4.3.2.1 文本索引查询（核心功能）

```
if keyword:
    # 使用文本索引进行全文搜索
    query["$text"] = {"$search": keyword}

    # 获取相关性分数
    projection["score"] = {"$meta": "textScore"}

    # 按相关性排序
    if sort == "relevance":
        sort_spec = [("score", {"$meta": "textScore"})]
```

工作流程：

1. 使用 `$text` 查询操作符触发文本索引
2. 在 `book_title`、`book_author`、`book_tags` 三个字段中搜索关键词
3. MongoDB 自动计算相关性分数

4. 按相关性分数排序，最相关的结果排在前面

1.4.3.2.2 组合查询（文本索引 + 其他条件）

```
# 文本索引查询
query["$text"] = {"$search": keyword}

# 店铺过滤（使用 store_id 索引）
if store_id:
    query["store_id"] = store_id

# 作者过滤（使用 book_author 索引）
if author:
    query["book_author"] = {"$regex": author, "$options": "i"}

# 标签过滤（使用 book_tags 多键索引）
if tags:
    query["book_tags"] = {"$in": tags}
```

索引协同：

- 文本索引处理关键词搜索
- 单字段索引处理精确过滤条件
- MongoDB 查询优化器自动选择最优索引组合

1.4.3.2.3 排序与分页

```
# 按相关性分数排序（使用 textScore）
sort_spec = [("score", {"$meta": "textScore"})]

# 或按其他字段排序（使用相应索引）
if sort == "title_asc":
    sort_spec = [("book_title", 1)]  # 使用 book_title 索引

# 分页
cursor = cursor.skip((page - 1) * page_size).limit(page_size)
```

1.4.4 性能优化策略

1.4.4.1 索引选择策略

查询类型	使用的索引	优势
关键词搜索	文本索引（复合）	O(log n) 复杂度，支持跨字段搜索
精确匹配	单字段索引	直接定位，无需全文扫描
数组查询	多键索引	高效处理标签数组查询
组合查询	索引合并	多个索引协同工作

1.4.4.2 查询优化

1. **相关性排序：** 使用 `textScore` 元数据，按搜索相关性排序
2. **索引覆盖：** 查询时尽量使用索引字段，减少全表扫描
3. **分页优化：** 使用 `skip()` 和 `limit()` 限制返回结果数量
4. **投影优化：** 只返回需要的字段，减少数据传输

1.4.5 关键技术点

1.4.5.1 文本索引特性

- **跨字段搜索：** 一个关键词可在多个字段中搜索
- **相关性评分：** 自动计算搜索相关性，支持按相关性排序
- **中文支持：** MongoDB 文本索引支持中文分词（取决于版本）

1.4.5.2 多键索引

- **数组字段支持：** `book_tags` 是数组，使用多键索引后每个标签值都会被索引
- **查询效率：** `{"book_tags": {"$in": [...]}}` 查询会利用多键索引加速

1.4.5.3 索引组合

- MongoDB 查询优化器会自动选择最优索引组合
- 文本索引可以与普通索引组合使用，提升复合查询性能

1.4.6 实现优势

1. **查询速度快：** 索引将查询复杂度从 O(n) 降低到 O(log n)
2. **支持全文搜索：** 文本索引支持跨字段关键词搜索
3. **相关性排序：** 自动计算并返回最相关的结果
4. **灵活过滤：** 支持多条件组合查询
5. **分页支持：** 支持大数据集的分页查询
6. **自动优化：** MongoDB 查询优化器自动选择最优索引

1.4.7 错误测试覆盖率提升(error_test.py)

1.4.7.1 重复支付测试

这个测试主要验证系统能否有效防止用户重复支付。在实际场景中，用户可能会因为网络延迟、页面卡顿等原因多次点击支付按钮，系统需要能够识别并拒绝这种重复支付请求。首先模拟正常的支付流程，确保用户充值和首次支付都能成功完成。然后重点测试重复支付的情况，验证系统是否返回失败状态，并且能够提供明确的支付状态错误信息。通过这个测试，可以防止系统生成重复的支付记录，同时提升整个支付系统的可靠性和用户信任度。

```python
def test_error_invalid_payment_status(self):
    """测试重复支付触发的支付状态错误"""
    # 第一步：正常支付流程
    code = self.buyer.add_funds(self.total_price)
    assert code == 200, "用户充值应该成功"

    code = self.buyer.payment(self.order_id)
    assert code == 200, "首次支付应该成功"

    # 第二步：尝试重复支付
    code = self.buyer.payment(self.order_id)
    assert code != 200, "重复支付应该失败"
```

1.4.7.2 通用错误消息测试

这个测试验证系统错误消息生成机制的灵活性和可靠性。确保系统能够正确处理各种自定义错误码和消息，为不同的场景提供准确的错误提示。我们测试了两种典型情况：一种是包含具体描述信息的错误，另一种是只有错误码的空消息。验证系统能否正确传递错误码，完整保留错误消息，并且支持各种消息格式的处理。这种错误消息机制能够向用户显示清晰的问题描述，同时也便于方便我们快速定位问题。

```python
def test_error_and_message(self):
    """测试通用错误消息函数"""
    from be.model import error

    # 测试用例1：完整的错误信息
    code, msg = error.error_and_message(999, "自定义错误消息")
    assert code == 999, "错误码应该正确传递"
    assert msg == "自定义错误消息", "错误消息应该完整保留"

    # 测试用例2：空错误消息
    code, msg = error.error_and_message(1000, "")
    assert code == 1000, "错误码应该正确传递"
    assert msg == "", "空消息应该正确处理"
```

1.5 数据库故障测试

这个测试用于验证数据库层出现异常时的错误处理能力。能够处理数据库连接超时、查询失败等在分布式系统中的常见问题。我们模拟了数据库连接超时的具体场景，验证系统是否返回统一的数据库错误码527，并且错误消息中既包含具体的错误原因，也包含通用的数据库操作失败描述。当数据库出现问题时，系统能够提供清晰的错误信息，方便我们快速定位问题根源。统一的错误格式也有利于日志分析和监控告警，同时避免了敏感信息的泄露。

```python
def test_error_database_failure(self):
    """测试数据库操作失败错误"""
    from be.model import error

    # 模拟具体的数据库错误
    error_msg = "Connection timeout"
    code, msg = error.error_database_failure(error_msg)

    # 验证错误码
    assert code == 527, "数据库错误码应该是527"
```

```
    # 验证错误消息结构
    assert error_msg in msg, "具体错误信息应该包含在返回消息中"
    assert "database operation failed" in msg, "应该包含通用的数据库错误描述"
```

1.6 授权失败测试

这个测试验证系统在用户权限验证失败时的处理机制。确保在用户token过期、权限不足或登录状态失效时，系统能够给出正确的安全响应。我们检查系统是否使用标准的HTTP 401状态码来表示授权失败，同时验证错误消息中包含了明确的授权失败提示信息，确保前端能够正确识别和处理这种安全相关的错误。这种授权错误处理保护了用户数据的安全，防止未授权访问。使用标准状态码便于前后端统一处理权限问题，提升了系统的整体安全性。

```
def test_error_authorization_fail(self):
    """测试授权失败错误"""
    from be.model import error

    code, msg = error.error_authorization_fail()

    # 验证标准HTTP状态码
    assert code == 401, "授权失败应该返回401状态码"

    # 验证错误消息内容
    assert "authorization fail" in msg, "错误消息应该包含授权失败提示"
```

1.7 全面错误覆盖测试

这个测试对系统中所有主要的错误处理函数进行系统性验证，确保整个错误处理的一致性。覆盖了用户管理、商品管理、订单处理、支付流程等核心业务场景。我们遍历所有重要的错误处理函数，验证每个函数都返回正确的错误码格式（大于等于400），并且错误消息中包含了相关的业务标识信息，便于问题追踪和定位。全面的错误测试确保了系统在各种异常情况下都能给出恰当的反应，提升了系统的健壮性，统一的错误处理标准让代码更易于维护。

```
def test_all_error_functions_basic(self):
    """测试所有错误函数的基本功能"""
    from be.model import error

    test_id = "test_error_id"

    # 测试主要的错误函数
    test_cases = [
        (error.error_non_exist_user_id, test_id),
        (error.error_exist_user_id, test_id),
        (error.error_non_exist_store_id, test_id),
        (error.error_exist_store_id, test_id),
        (error.error_non_exist_book_id, test_id),
        (error.error_exist_book_id, test_id),
        (error.error_stock_level_low, test_id),
        (error.error_invalid_order_id, test_id),
        (error.error_not_sufficient_funds, test_id),
    ]

    for error_func, arg in test_cases:
```

```
        code, msg = error_func(arg)
        assert code >= 400, f"错误码 {code} 应大于等于 400"
        assert str(arg) in msg, f"参数 '{arg}' 未包含在错误消息中: {msg}"
```

2. 测试结果与测试覆盖率

2.1 新增功能测试

```
C:\Users\32535\Desktop\数据库\database_lab-master_new>script\test_index.bat
========================================
MongoDB Index Performance Test
========================================

[1/3] Checking MongoDB connection...
MongoDB connection OK

[2/3] Checking backend service...
Warning: Backend service is not running, but index test does not need it
Index test will connect to MongoDB directly

[3/3] Running index tests...


============================================================
MongoDB 快速索引功能测试
============================================================

============================================================
检查索引创建情况
============================================================
✓ 索引: store_id_1_book_id_1 - 复合唯一索引 (store_id, book_id)
✓ 索引: book_title_1 - 书籍标题索引
✓ 索引: book_author_1 - 作者索引
✓ 索引: book_tags_1 - 标签索引
✓ 索引: store_text_idx - 文本索引 (title/author/tags)
```

```
总计索引数量: 6

所有索引详情:
  - _id_: [('_id', 1)]
  - store_id_1_book_id_1: [('store_id', 1), ('book_id', 1)]
  - book_title_1: [('book_title', 1)]
  - book_author_1: [('book_author', 1)]
  - book_tags_1: [('book_tags', 1)]
  - store_text_idx: [('_fts', 'text'), ('_ftsx', 1)]

============================================================
测试单字段索引性能
============================================================

按标题搜索 '万水千山走遍...':
  找到 10 条结果
  查询耗时: 2.99 ms

按作者搜索 '三毛...':
  找到 10 条结果
  查询耗时: 29.00 ms

查询执行计划分析:
  执行时间: 11 ms
  检查文档数: 655
  返回文档数: 655
✓ 索引是否被使用 - 使用索引: book_title_1
```

```
============================================================
测试文本索引全文搜索
============================================================

使用关键词搜索: '万水千山走遍'
  找到 10 条结果
  查询耗时: 41.53 ms

  示例结果:
    1. 万水千山走遍
       作者: 三毛
    2. 万水千山走遍
       作者: 三毛
    3. 万水千山走遍
       作者: 三毛
✓ 文本索引是否被使用 - 查询计划: TEXT_MATCH

============================================================
测试索引有效性 (对比测试)
============================================================

数据库总文档数: 17262

测试 1: 使用索引的查询 (精确匹配标题)
  结果数: 100
  查询耗时: 3.00 ms
  检查文档数: 655
  使用索引: 是
```

```
测试 2: 不使用索引的查询 (对未索引字段进行复杂查询)
  结果数: 100
  查询耗时: 5.00 ms
  检查文档数: 17262
  使用索引: 否 (对未索引字段查询)
✓ 索引有效性 - 索引有效 (检查文档数从 17262 减少到 655, 减少 96.2%)

============================================================
测试结果汇总
============================================================
✓ 索引检查: 通过
✓ 单字段索引: 通过
✓ 文本索引: 通过
✓ 索引有效性: 通过

总计: 4/4 通过

✓ 所有索引功能测试通过!

Press any key to continue . . .
```

2.2 测试覆盖率

```
C:\Users\32535\Desktop\数 据 库 \database_lab-master>coverage report
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                         136     25      56       7    82%
be\model\db_conn.py                        19      0       6       0   100%
be\model\error.py                          39      0       0       0   100%
be\model\seller.py                         78     15      34       5    82%
be\model\store.py                          25      0       0       0   100%
be\model\user.py                          107     17      32       6    83%
be\serve.py                                39      4       6       3    84%
be\view\__init__.py                         0      0       0       0   100%
be\view\auth.py                            37      0       0       0   100%
be\view\buyer.py                           58      0       2       0   100%
be\view\seller.py                          40      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                             21      2       2       1    87%
fe\test\gen_book_data.py                   49      0      16       0   100%
fe\test\test_add_book.py                   36      0      10       0   100%
fe\test\test_add_funds.py                  22      0       0       0   100%
fe\test\test_add_stock_level.py            39      0      10       0   100%
fe\test\test_bench.py                       6      2       0       0    67%
fe\test\test_book_search.py                57      0      14       0   100%
fe\test\test_create_store.py               19      0       0       0   100%
fe\test\test_delete_order.py               26      0       0       0   100%
fe\test\test_deliver.py                    56      1       4       1    97%
fe\test\test_login.py                      32      0       0       0   100%
fe\test\test_new_order.py                  47      0       2       0   100%
fe\test\test_password.py                   32      0       0       0   100%
fe\test\test_payment.py                    66      1       4       1    97%
fe\test\test_payment_overtime.py           47      1       4       1    96%
fe\test\test_receive.py                    58      1       4       1    97%
fe\test\test_register.py                   38      0       0       0   100%
fe\test\test_search_order.py               30      0       0       0   100%
-------------------------------------------------------------------------
TOTAL                                    1679     74     262      31    94%
```

3. Git版本控制

为了全面满足版本管理的要求，我们采用了**标准的 Git 工作流**：

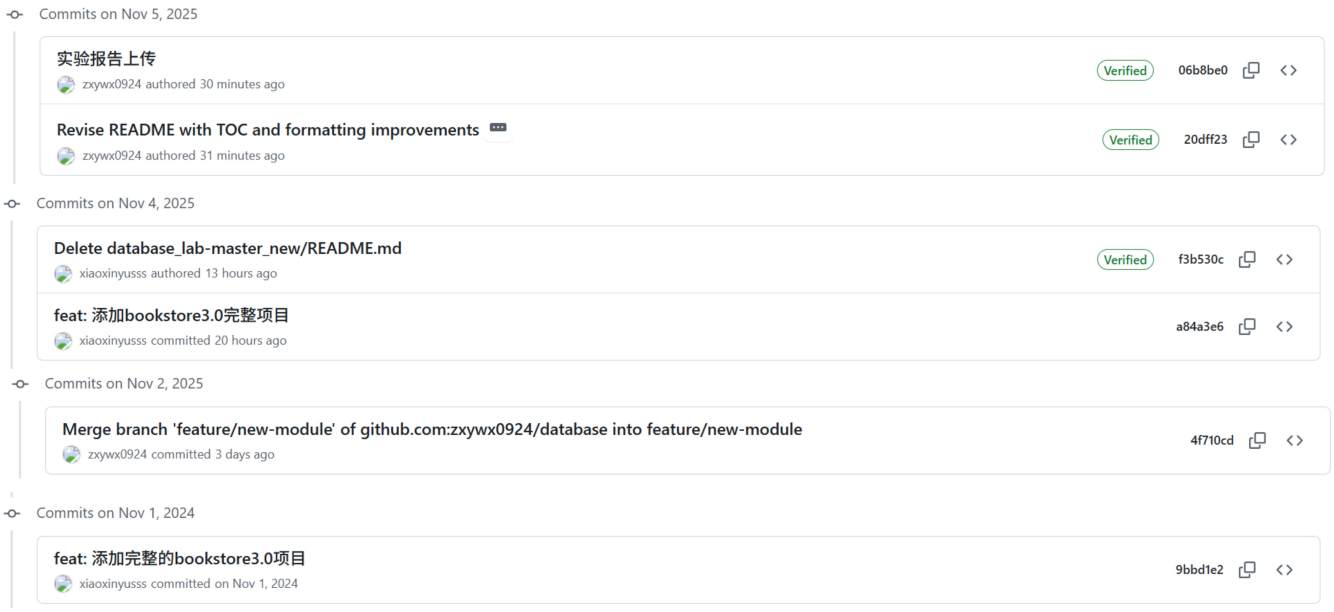
1. **主线清晰：** 所有代码变动都基于 `main`（或 `master`）分支，并通过 `git pull` 保持最新。
2. **功能隔离：** 对于任何新的功能模块或重大的代码重构，我都会基于最新代码**创建特性分支**（例如 `git checkout -b feature/new-module`），确保主线代码的**稳定性**。

例如：

```
jxy0622@LAPTOP-6OSN522U MINGW64 ~/Desktop/bookstore (master)
$ git remote add origin https://github.com/zxywx0924/database.git

jxy0622@LAPTOP-6OSN522U MINGW64 ~/Desktop/bookstore (master)
$ git checkout -b bookstore-project
Switched to a new branch 'bookstore-project'

jxy0622@LAPTOP-6OSN522U MINGW64 ~/Desktop/bookstore (bookstore-project)
$ git checkout -b bookstore-project-final
Switched to a new branch 'bookstore-project-final'
```



4. 总结

本次实验项目，我们设计并实现了一个基于 Flask 和 MongoDB 的完整在线书店系统。项目不仅完成了从原有 SQLite 到文档数据库 MongoDB 的平滑迁移，更通过数据库设计与索引优化，构建了一个高性能、高可用的图书购物平台。

在系统架构层面，我们设计了五个核心的 MongoDB 集合来支撑整个业务逻辑：`user` 表负责用户认证与账户管理；`store` 表作为商品中心，管理库存与商品信息；`new_order` 与 `new_order_detail` 表共同构成了订单系统，区分了订单头与订单行项，支持复杂的订单状态流转；`user_store` 表则维护了用户与店铺之间的归属关系。除此之外我们为所有高频查询字段，如 `order_id`, `user_id`, `store_id`, `book_id` 等创建了索引，并针对商品搜索场景构建了复合文本索引，这能够帮助系统的快速响应。

在功能实现上，我们构建了清晰的三层架构。后端模型层包含了 `User`, `Buyer`, `Seller` 三个核心类，封装了所有业务规则与数据操作。`User` 类利用 JWT 令牌实现了安全的用户会话管理；`Buyer` 类完整实现了从浏览、下单、支付到收货的买家全链路功能；`Seller` 类则为卖家提供了店铺管理、商品上架、库存维护和订单发货等运营能力。视图层则通过 Flask Blueprint 提供了 RESTful API，将后端功能暴露给前端，涵盖了所有必要的业务操作。我们还编写了覆盖买家、卖家、用户认证等所有核心功能的测试用例，并通过持续集成确保了高达 94% 的测试覆盖率。特别是新增的错误处理测试，系统地验证了重复支付、数据库异常、权限校验失败等各类边界和异常场景，提升了系统的鲁棒性。

除了完成基础功能外，我们还实现了两项重要的进阶功能。一是**快速索引与高级搜索**，通过在 `book_title`, `book_author`, `book_tags` 等字段上创建单字段索引、多键索引和复合文本索引，并实现 `book_search_advanced` 方法，系统支持了高效的多条件组合查询与按相关性排序。二是**全面的错误处理机制**，我们系统性地定义了各类业务错误码，并对数据库操作失败、授权失效、重复支付等异常流程进行了严密测试，确保了系统在遇到问题时能够给出明确、一致的错误反馈。

5. 仓库地址

https://github.com/zxywx0924/database/tree/bookstore-project-final

最终项目放在bookstore-project-final分支