BOOKSTORE2

当代数据管理系统 第二次大作业 实验报告

10215501412 彭一珅

一、实验目标

本次实验的目标是实现一个提供网上购书功能的网站后端。网站支持书商在上面开商店，购买者可以通过网站购买。买家和卖家都可以注册自己的账号。一个卖家可以开一个或多个网上商店，买家可以为自己的账户充值，在任意商店购买图书。

功能上需要支持 下单->付款->发货->收货 流程。

比起第一次实验，本次实验不再使用mongodb这种文档型数据库，而是要求使用关系型数据库（PostgreSQL 或 MySQL 数据库）。

二、数据库设计

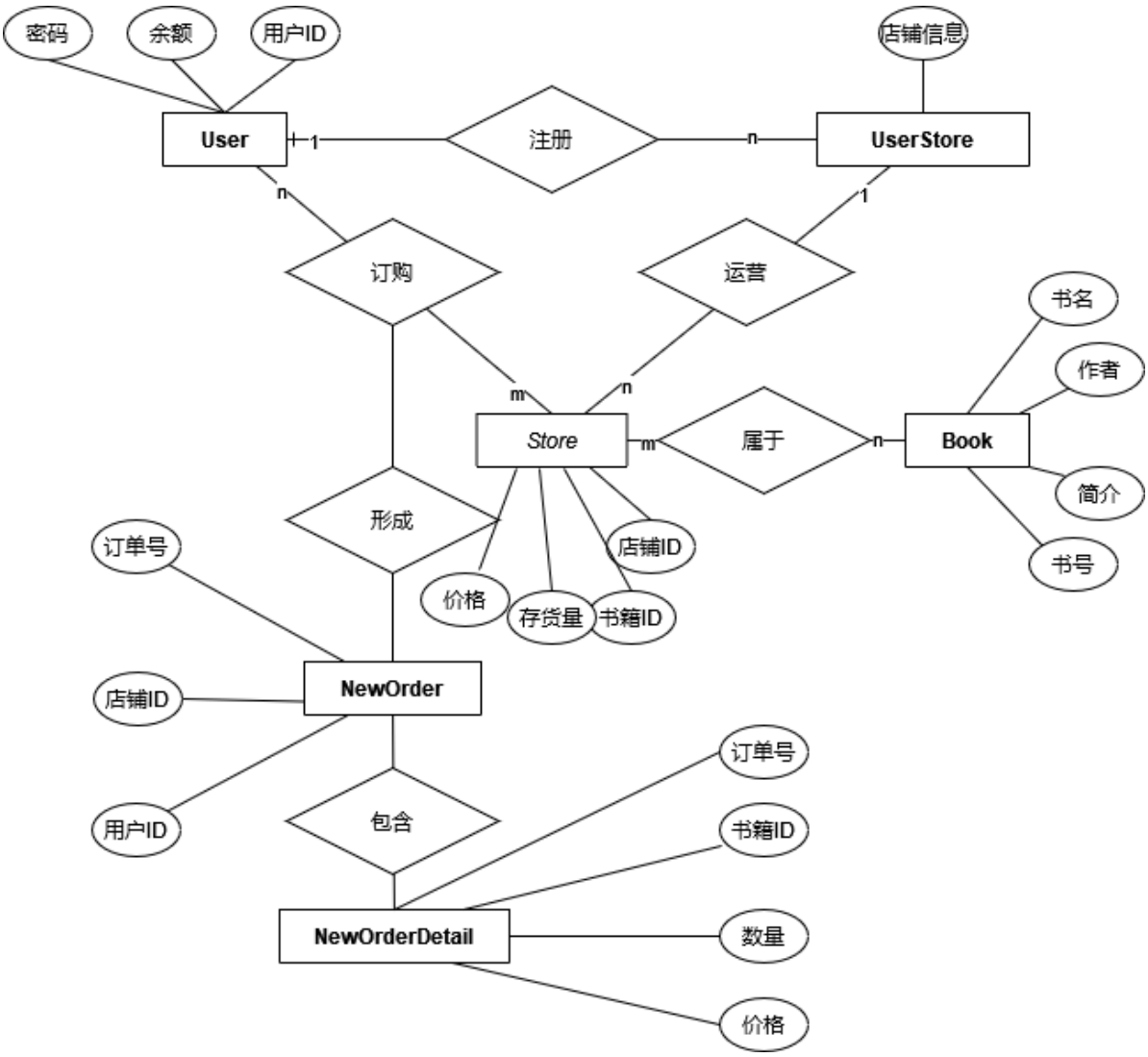
从文档型数据库到关系型数据库的改动，以及改动的理由。

文档数据库和关系数据库有以下区别：

- 文档数据库以json文档的方式存储，可以包含嵌套的子文档或数组，查询操作也更加灵活。关系数据库则以表格的方式进行存储，使用结构化查询语言。
- 文档数据库中同一表中可以存不同结构的数据。关系数据库每个表格有特定的列和行，同一表中的数据属性种类完全相同。
- 文档数据库没有固定的外键约束，关系数据库可以通过数据库约束来确保数据完整性。

具体到本项目中，对于文档型数据库，在用户开设商店的时候，可以在user表中包含一个store列表，其中包含了用户开设的所有商店id；在store表中，可以存一个列表为book，列表中保存所有相关的书本信息；在new_order表中，也可以存一个子文档为book，并包含书本的数目、价钱等信息。但是对于关系数据库，由于不便于嵌套文档，需要将用户开设的每个店铺分开记录；将商店内的每个种类的书本分多条信息记录等。

因此画出er图，设置了用户（User）、商铺（UserStore）、书籍（Book）、货架（Store）、订单（NewOrder）、订单明细（NewOrderDetail）等6个实体。其中设计货架是将用户、商铺和书籍有机的联系在一起。用户同时可以是店主和购买者，一个用户可以拥有多个商铺。



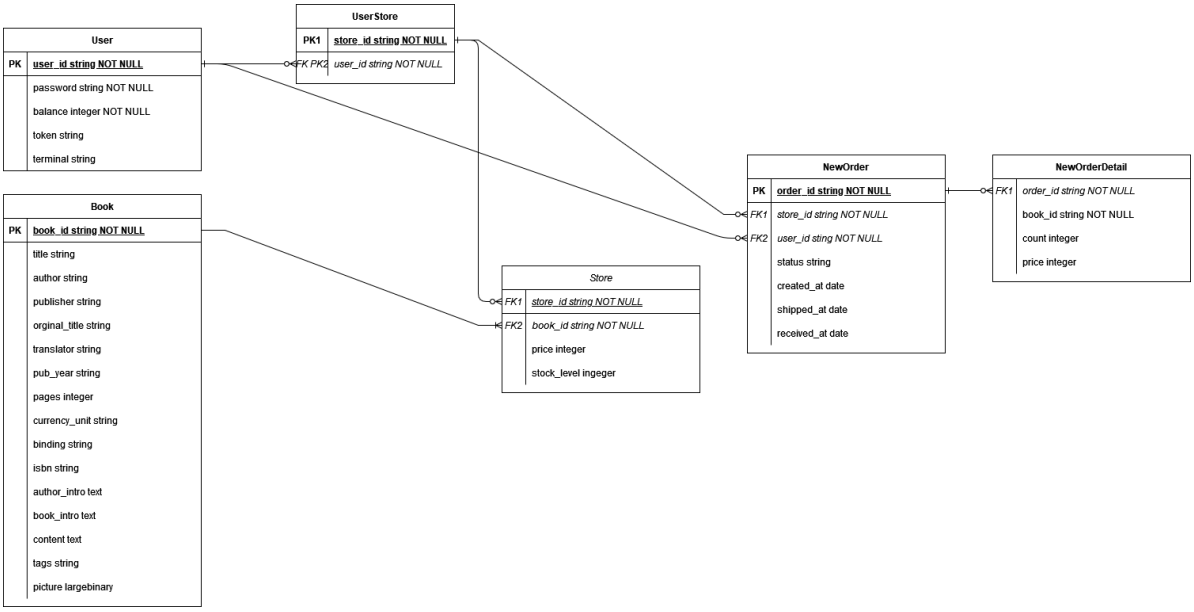
根据对用户实体梳理，设计了User表，其主键是user_id。记录了不可再分割的用户的基本信息、注册信息和充值信息；

根据对商铺实体的梳理，设计了UserStore表，其主键为store_id。记录了简单的商铺信息，可以扩充登记商铺的工商登记、地址等详细信息；

根据对书籍实体的梳理，设计了Book表，其主键为book_id。记录了书籍的详细详细包括书名、作者、摘要、标准书号，等等；

根据对货架实体的梳理，设计了Store表，其主键为组合主键：store_id+book_id，记录当前上架的书籍的信息；

根据对订单和订单明细实体的梳理，设计了NewOrder表和NeworderDetail表，分别记录订单的基本信息和订单中若干货品（书籍）的信息。



以购书做为典型事务进行简要描述：

首先启动事务；然后查询Store表中是否有待购书籍并且存量大于0，成立则继续，不成立则提示无货；接着生成订单和订单明细，并提示购买者付款；如果付款成功，则对存货数量（Store表的stock_level字段）进行修改；如果购买者是用充值的钱支付，则对余额（user表的banlance字段）进行修改；最后结束事务。

1、 为User表的增加一个组合索引。因为经常需要查询“余额是否大于订单金额”，所以增加一个user_id+balance的组合索引。

2、 为Store表增加一个组合索引。因为经常需要查询“某本书是否还有库存“，所以增加一个book_id+stock_level的组合索引。

3、 为NewOrder表增加一个组合索引。因为经常需要查询“当前订单状态“，所以增加一个order_id+status的组合索引。

三、代码实现

对60%基础功能和40%附加功能的接口、后端逻辑、数据库操作、测试用例进行介绍。

本次实验相比第一次实验，由于更换了数据库，相关操作也需要进行调整。

首先在连接数据库的时候，通过sqlalchemy库来进行。连接postgres数据库的bookstore database，密码为123456. 在store.py文件中定义所有的表结构，可以直接通过metadata.create_all来初始化所有表。

```
class BookStore:

    def __init__(self):
        self.engine =
create_engine('postgresql://postgres:123456@localhost:5432/bookstore')
        self.DbSession = sessionmaker(bind=self.engine)
        self.session = self.DbSession()
        self.init_tables()

    def init_tables(self):
        Base.metadata.create_all(self.engine)

    def get_db_conn(self):
        return self.session
```

获取连接的方式也有所不同，调用get_db_conn函数可以获取到session。此外，DBConn类还自带了几个方法，这些方法可以检验各类的id（主键）在表内是否存在，以供后面的model使用。

```
class DBConn:
    def __init__(self):
        self.conn = store.get_db_conn()

    def user_id_exist(self, user_id):
        return self.conn.query(UserModel).filter(UserModel.user_id ==
user_id).first() is not None

    def book_id_exist(self, store_id, book_id):
        # 在这个店里已经存在这本书
        return (
                self.conn.query(StoreModel)
                .filter(StoreModel.store_id == store_id, StoreModel.book_id
== book_id)
                .first()
                is not None
        )

    def book_id_exist_all(self, book_id):
        return (
                self.conn.query(BookModel)
                .filter(BookModel.id == book_id)
                .first()
                is not None
        )

    def store_id_exist(self, store_id):
        return (
                self.conn.query(UserStore)
                .filter(UserStore.store_id == store_id)
                .first()
                is not None
        )
```

接口实现及数据库操作

用户权限接口

注册：先在user表内查询表中是否已经存在用户输入的user_id，如果没有，就在user表中添加一条数据，根据当前时间生成terminal和token，刚刚注册的用户初始金额为0.

```
def register(self, user_id: str, password: str):
    try:
        if self.user_id_exist(user_id):
            return error.error_exist_user_id(user_id)
        terminal = "terminal_{}".format(str(time.time()))
        token = jwt_encode(user_id, terminal)
        new_user = UserModel(
            user_id=user_id,
            password=password,
            balance=0,
            token=token,
            terminal=terminal,
        )

        self.conn.add(new_user)
        self.conn.commit()
    except IntegrityError as e:
        return 528, "{}".format(str(e))

    return 200, "ok"
```

登录：在user表内查询所输入user_id对应的password，如果用户名不存在或密码不正确，返回权限错误，如果存在的话，在user表内查询所输入user_id对应的token和terminal，将这两个表示对应用户权限的值返回，此后用户请求在headers使用这两个值就可以标明处于在线状态。

```
def login(self, user_id: str, password: str, terminal: str) -> (int, str,
str):
    token = ""
    try:
        code, message = self.check_password(user_id, password)
        if code != 200:
            return code, message, ""

        token = jwt_encode(user_id, terminal)
        user = self.conn.query(UserModel).filter_by(user_id=user_id).first()
        user.token = token
        user.terminal = terminal

        self.conn.commit()
    except IntegrityError as e:
        self.conn.rollback()
        return 528, "{}".format(str(e)), ""
    except BaseException as e:
        return 530, "{}".format(str(e)), ""
    return 200, "ok", token
```

登出：先查询user表，检查user_id和token是否存在且对应，然后根据当前时间戳生成新的terminal和token。

```
def logout(self, user_id: str, token: str):
    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)

        user = self.conn.query(UserModel).filter_by(user_id=user_id).first()
        user.token = dummy_token
        user.terminal = terminal

        self.conn.commit()
    except IntegrityError as e:
        self.conn.rollback()
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

注销：查询user表，检查输入的用户名和密码是否存在且匹配，如果是就从user表中删除user_id对应的记录。

```
def unregister(self, user_id: str, password: str) -> (int, str):
    try:
        code, message = self.check_password(user_id, password)
        if code != 200:
            return code, message

        user = self.conn.query(UserModel).filter_by(user_id=user_id).first()
        self.conn.delete(user)
        self.conn.commit()
    except IntegrityError as e:
        self.conn.rollback()
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

买家用户接口

下单：先查询user表，检查用户是否存在，再查询user_store表，检查商店是否存在，然后遍历用户给出的book_id和count列表，查询store表，检查每本书在商店里是否存在、书本数目是否充足，如果符合要求就扣除书店中的库存，然后创建订单详细信息，其中每本书对应new_order_detail表中的一条记录，将所有书本的价格累加，最后在new_order表中创建一条记录，将status设置为unpaid，created_time为当前时间，表示订单创建成功。

```
def new_order(
        self, user_id: str, store_id: str, id_and_count: [(str, int)]
```

```
) -> (int, str, str):
    order_id = ""
    try:
        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,)
        uid = "{}_{}_{}".format(user_id, store_id, str(uuid.uuid1()))

        total_price = 0
        new_order_detail_list = []
        for book_id, count in id_and_count:
            store_data = self.conn.query(StoreModel).filter_by(
                store_id=store_id,
                book_id=book_id
            ).first()

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

            stock_level = store_data.stock_level
            price = store_data.price

            if stock_level < count:
                return error.error_stock_level_low(book_id) + (order_id,)

            total_price += price * count

            # 扣除库存
            store_data.stock_level -= count

            # 创建订单详细信息
            new_order_detail = NewOrderDetail(
                order_id=uid,
                book_id=book_id,
                count=count,
                price=price
            )
            new_order_detail_list.append(new_order_detail)

        # 更新订单状态为 "unpaid"
        new_order = NewOrder(
            user_id=user_id,
            store_id=store_id,
            order_id=uid,
            status="unpaid",
            created_at=datetime.now().isoformat(),
            shipped_at=None,
            received_at=None
        )

        self.conn.add_all(new_order_detail_list)
        self.conn.add(new_order)
        self.conn.commit()

```

```
        order_id = uid

    except IntegrityError as e:
        return str(e)
    except Exception as e:
        traceback.print_exc()
        return 530, "{}".format(str(e)), ""

    return 200, "ok", order_id
```

充值：查询user表，查看用户是否存在，password是否匹配，如果存在且匹配就修改这条记录，在balance属性上增加用户输入的add_value。

```
def add_funds(self, user_id, password, add_value) -> (int, str):
    try:
        user_data =
self.conn.query(UserModel).filter_by(user_id=user_id).first()

        if user_data is None:
            return error.error_authorization_fail()

        if user_data.password != password:
            return error.error_authorization_fail()

        # 增加用户余额
        user_data.balance += add_value

        self.conn.commit()

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

    return 200, "ok"
```

付款：查询new_order表，确认订单号存在，然后确认订单对应的user_id是否与用户输入的user_id相对应，并检查订单状态是否正确，如果是paid、cancelled、shipped、received就返回订单状态错误。检查user表中输入的user_id和passwd是否存在且匹配，并获取用户当前的balance。如果balance小于订单的总价，就返回错误信息，没有足够的金额，如果足够付款，就从balance中扣除总价，并将订单状态更新为paid。

```
def payment(self, user_id: str, password: str, order_id: str) -> (int, str):
    try:
        order_data =
self.conn.query(NewOrder).filter_by(order_id=order_id).first()

        if order_data is None:
            return error.error_invalid_order_id(order_id)

        if order_data.user_id != user_id:
            return error.error_authorization_fail()

        if order_data.status != "unpaid":
            return error.error_status_fail(order_id)

```

```
        user_data = self.conn.query(UserModel).filter_by(user_id=user_id).first()
        if user_data is None:
            return error.error_non_exist_user_id(user_id)

        if password != user_data.password:
            return error.error_authorization_fail()

        balance = user_data.balance

        # 获取订单详细信息
        order_detail_data = self.conn.query(NewOrderDetail).filter_by(order_id=order_id).all()
        total_price = sum([order_detail.price * order_detail.count for order_detail in order_detail_data])

        if balance < total_price:
            return error.error_not_sufficient_funds(order_id)

        # 扣款，更新用户余额
        new_balance = balance - total_price
        user_data.balance = new_balance

        # 更新订单状态为 "paid"
        order_data.status = "paid"

        self.conn.commit()

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

    return 200, "ok"
```

卖家用户接口

创建店铺：先查询user表，检查输入的卖家id是否存在，然后查询user_store表，检查输入的store_id是否已经被创建过，如果都没有，就可以在user_store表中创建一条新的数据，包含store_id（唯一）和user_id（不唯一，因为多个用户可以创建多个商店）。

```
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)

        # 使用SQLAlchemy插入数据
        new_user_store = UserStore(
            store_id=store_id,
            user_id=user_id
        )

        self.conn.add(new_user_store)
        self.conn.commit()

```

```
    except IntegrityError as e:
        return 528, "{}".format(str(e))
    except Exception as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

添加书籍信息及描述：检查user_id、store_id的存在之后，查询user_store表并匹配输入的store_id，检查book_id在这个商店中是否存在，如果不存在，则查询book表，如果book表中存在对应的id，则表示这本书存在于别的商店里，则不需要再次插入到book表中，只需要在store表中插入一条记录即可，这条记录包含store_id，book_id，price和stock_level。如果这本书也不存在与book表中，则在book表中要插入详细的记录，记录的各项属性由用户输入的book_json决定。

```
def add_book(
        self,
        user_id: str,
        store_id: str,
        book_json_str: str,
        stock_level: int,
):
    book_info=json.loads(book_json_str)
    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_info['id']):
            return error.error_exist_book_id(book_info['id'])
        if self.book_id_exist_all(book_info['id']):
            new_store = StoreModel(
                store_id=store_id,
                book_id=book_info['id'],
                price=book_info["price"],
                stock_level=stock_level
            )

            self.conn.add(new_store)
            self.conn.commit()
        else:
            # 使用SQLAlchemy插入数据
            new_book = BookModel(
                id=book_info['id'],
                title=book_info["title"],
                author=book_info["author"],
                publisher=book_info["publisher"],
                original_title=book_info["original_title"],
                translator=book_info["translator"],
                pub_year=book_info["pub_year"],
                pages=book_info["pages"],
                price=book_info["price"],
                currency_unit=book_info["currency_unit"],
                binding=book_info["binding"],
                isbn=book_info["isbn"],
                author_intro=book_info["author_intro"],
                book_intro=book_info["book_intro"],
```

```
                content=book_info["content"],
                tags=book_info["tags"],
                # picture=book_info["picture"]  # 假设这里的图片存储在数据库中的字段为 LargeBinary
            )
            # self.sqlite_conn.execute('INSERT INTO book_info (picture) VALUES (?)', (book_info["picture"],))
            # self.sqlite_conn.commit()

            new_store = StoreModel(
                store_id=store_id,
                book_id=book_info['id'],
                price=book_info["price"],
                stock_level=stock_level
            )

            self.conn.add_all([new_book, new_store])
            self.conn.commit()

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

        return 530, "{}".format(str(e))
    return 200, "ok"
```

增加库存：查询user表和user_store表，检查用户和商店是否存在，然后查询store表，检查book_id是否存在于特定商店中。如果存在，那么对于store表中的对应记录，将stock_level添加指定的个数即可。

```
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)

        # 使用SQLAlchemy更新数据
        store = self.conn.query(StoreModel).filter_by(store_id=store_id, book_id=book_id).first()
        store.stock_level += add_stock_level
        self.conn.commit()

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

发货收货流程

卖家发货：先检查对应的store_id是否存在，如果存在，就查询new_order表，找到order_id和store_id对应的记录，如果订单不存在则报错，订单状态为shipped，就返回200并声明订单已经发货，如果订单状态不为paid，那么就返回状态错误，此外，就将订单状态设为shipped，并记录发货时间即可。

```
def ship_order(self, store_id: str, order_id: str) -> (int, str):
    try:
        if not self.store_id_exist(store_id):
            return error.error_exist_store_id(store_id)

        # 使用SQLAlchemy更新数据
        order = self.conn.query(NewOrder).filter_by(order_id=order_id,
store_id=store_id).first()

        if order is None:
            return error.error_invalid_order_id(order_id)

        if order.status == "shipped":
            return 200, "Order is already shipped."

        if order.status != "paid":
            return error.error_status_fail(order_id)

        order.status = "shipped"
        order.shipped_at = datetime.now()
        self.conn.commit()
    except Exception as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

买家收货：查询new_order表，查看order_id是否存在，确认user_id是否匹配，检查订单状态，如果状态为received，就返回200，并提示当前订单已经收货，如果订单状态不是shipped（已发货状态），就返回状态错误。如果状态为shipped，就将状态更新为received，并记录收货时间。

```
def receive_order(self, user_id: str, order_id: str) -> (int, str):
    try:
        order_data =
self.conn.query(NewOrder).filter_by(order_id=order_id).first()

        if order_data is None:
            return error.error_invalid_order_id(order_id)

        if order_data.user_id != user_id:
            return error.error_authorization_fail()

        if order_data.status == "received":
            return 200, "Order is already received"

        if order_data.status != "shipped":
            return error.error_status_fail(order_id)

        # 更新订单状态为 "received" 并记录收货时间
        order_data.status = "received"
```

```
        order_data.received_at = datetime.now().isoformat()

        self.conn.commit()

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

    return 200, "ok"
```

图书搜索功能

店内搜索：首先将`store_alias`和`book_alias`创建为`StoreModel`和`BookModel`的别名，以便在后续的查询中使用，然后构建一个基本的查询`query`，关联了`StoreModel`和`BookModel`表，过滤条件是`store_id`等于给定的`store_id`，根据提供的可选搜索条件，逐一添加过滤条件到查询中。如果有传递的条件（如 title、author、publisher 等），则将相应的 ilike 过滤条件添加到查询中。如果传递了`tags`列表，使用循环为每个标签添加过滤条件，确保图书的标签包含列表中的任何标签。计算查询的分页结果，通过`query.all()`获取所有结果，然后使用切片操作（`[skip: skip + limit]`）获取指定页数和每页数量的子集。遍历分页结果，将每本图书的相关信息构建成一个字典，并添加到结果列表中。返回查询结果，如果没有异常发生，返回状态码 200 和查询结果列表。如果发生异常，打印异常信息，并返回状态码 530 和异常描述字符串。

```
def search_in_store(self, store_id, title, author, publisher, isbn, content, tags, book_intro, page, per_page):
    try:
        store_alias = aliased(StoreModel)
        book_alias = aliased(BookModel)

        query = (
            self.conn.query(book_alias)
            .join(store_alias, store_alias.book_id == book_alias.id)
            .filter(store_alias.store_id == store_id)
        )
        if title:
            query = query.filter(book_alias.title.ilike(f'%{title}%'))
        if author:
            query = query.filter(book_alias.author.ilike(f'%{author}%'))
        if publisher:
            query = query.filter(book_alias.publisher.ilike(f'%{publisher}%'))
        if isbn:
            query = query.filter(book_alias.isbn.ilike(f'%{isbn}%'))
        if content:
            query = query.filter(book_alias.content.ilike(f'%{content}%'))
        if book_intro:
            query = query.filter(book_alias.book_intro.ilike(f'%{book_intro}%'))

        if tags:
            for tag in tags:
                query = query.filter(book_alias.tags.ilike(f'%{tag}%'))

        result = []
        skip = (page - 1) * per_page
        limit = per_page
```

```
        paged_result=query.all()[skip: skip + limit]
        for book in paged_result:
            result.append({
                'id': book.id,
                'title': book.title,
                'author': book.author,
                'publisher': book.publisher,
                'original_title': book.original_title,
                'translator': book.translator,
                'pub_year': book.pub_year,
                'pages': book.pages,
                'price': book.price,
                'currency_unit': book.currency_unit,
                'binding': book.binding,
                'isbn': book.isbn,
                'author_intro': book.author_intro,
                'book_intro': book.book_intro,
                'content': book.content,
                'tags': book.tags,
                # 'picture': book.picture
            })
    except Exception as e:
        traceback.print_exc()
        return 530, str(e)

    return 200, result
```

路由函数使用了装饰器 `@bp_book.route("/search_in_store", methods=["POST"])`，指定了路径为 "/search_in_store"，请求方法为 POST。从请求的 JSON 数据中提取了相关的参数，包括 `store_id`、`title`、`author`、`publisher`、`isbn`、`content`、`tags`、`book_intro`、`page` 和 `per_page`。创建了一个 `book.Book` 类的实例 `u`，该类可能用于处理与图书相关的业务逻辑。调用了 `u.search_in_store` 方法，传递了从请求中提取的各个参数。这个方法可能在后台进行数据库查询和处理。返回一个 JSON 格式的响应，包含查询结果 `data` 和状态码 `code`。响应中使用了 Flask 的 `jsonify` 函数。

```
@bp_book.route("/search_in_store", methods=["POST"])
def search_in_store():
    store_id = request.json.get("store_id", "")
    title = request.json.get("title", "")
    author = request.json.get("author", "")
    publisher = request.json.get("publisher", "")
    isbn = request.json.get("isbn", "")
    content = request.json.get("content", "")
    tags = request.json.get("tags", "")
    book_intro = request.json.get("book_intro", "")
    page = int(request.json.get("page", ""))
    per_page = int(request.json.get("per_page", ""))

    u = book.Book()
    code, data = u.search_in_store(
        store_id, title, author, publisher, isbn, content, tags,
book_intro,page,per_page
    )
    return jsonify({"data": data}), code
```

全网搜索：`search_all` 函数接收一系列搜索条件（如标题、作者、出版商等），以及分页参数（页数和每页数量）。通过 SQLAlchemy 构建了一个基本的查询 `query`，该查询目标是 `BookModel` 表。根据传递的搜索条件，逐一添加 ilike 过滤条件到查询中，以实现模糊搜索。如果条件存在，则在相应的列上添加 ilike 过滤条件。如果传递了 `tags` 列表，使用循环为每个标签添加 ilike 过滤条件，确保图书的标签包含列表中的任何标签。计算查询的分页结果，通过 `query.all()` 获取所有结果，然后使用切片操作（`[skip: skip + limit]`）获取指定页数和每页数量的子集。如果查询结果的总数小于等于分页的结束位置（`skip + limit`），则获取所有结果；否则，获取指定范围内的子集。将分页结果中的每本图书的相关信息构建成一个字典，并添加到结果列表中。返回查询结果，如果没有异常发生，返回状态码 200 和查询结果列表。如果发生异常，打印异常信息，并返回状态码 530 和异常描述字符串。

```
def search_all(self, title, author, publisher, isbn, content, tags,
book_intro,page,per_page):
    try:

        query = (
            self.conn.query(BookModel)
        )
        if title:
            query = query.filter(BookModel.title.ilike(f'%{title}%'))
        if author:
            query = query.filter(BookModel.author.ilike(f'%{author}%'))
        if publisher:
            query = query.filter(BookModel.publisher.ilike(f'%
{publisher}%'))
        if isbn:
            query = query.filter(BookModel.isbn.ilike(f'%{isbn}%'))
        if content:
            query = query.filter(BookModel.content.ilike(f'%{content}%'))
        if book_intro:
            query = query.filter(BookModel.book_intro.ilike(f'%
{book_intro}%'))

        if tags:
            for tag in tags:
                query = query.filter(BookModel.tags.ilike(f'%{tag}%'))
        result = []
        skip = (page - 1) * per_page
        limit = per_page
        if skip + limit< len(query.all()):
            paged_result = query.all()[skip: skip + limit]
        else:
            paged_result= query.all()
        print(len(query.all()))
        for book in paged_result:
            result.append({
                'id': book.id,
                'title': book.title,
                'author': book.author,
                'publisher': book.publisher,
                'original_title': book.original_title,
                'translator': book.translator,
                'pub_year': book.pub_year,
                'pages': book.pages,
                'price': book.price,
```

```
                'currency_unit': book.currency_unit,
                'binding': book.binding,
                'isbn': book.isbn,
                'author_intro': book.author_intro,
                'book_intro': book.book_intro,
                'content': book.content,
                'tags': book.tags,
                # 'picture': book.picture
            })
    except Exception as e:
        traceback.print_exc()
        return 530, str(e)

    return 200, result
```

全网搜索的接口与店内搜索差不多。

```
@bp_book.route("/search_all", methods=["POST"])
def search_all():
    title = request.json.get("title", "")
    author = request.json.get("author", "")
    publisher = request.json.get("publisher", "")
    isbn = request.json.get("isbn", "")
    content = request.json.get("content", "")
    tags = request.json.get("tags", "")
    book_intro = request.json.get("book_intro", "")
    page = int(request.json.get("page", ""))
    per_page = int(request.json.get("per_page", ""))

    u = book.Book()
    code, data = u.search_all(
        title, author, publisher, isbn, content, tags,
book_intro,page,per_page
    )
    return jsonify({"data": data}), code
```

订单查询和取消

买家订单查询：查询new_order表，并匹配有着输入的user_id的记录，将这些订单的信息作为字典返回。

```
def get_buyer_orders(self, user_id: str) -> (int, str, list):
    try:
        orders = self.conn.query(NewOrder).filter_by(user_id=user_id).all()
        buyer_orders = []
        # print(orders)
        for order in orders:
            buyer_orders.append(
                {'store_id': order.store_id,
                 'order_id': order.order_id,
                 'status': order.status}
            )
        return 200, "ok", buyer_orders
    except Exception as e:
```

```
        return 530, "{}".format(str(e)), []
```

接口返回一个 JSON 格式的响应，其中包含获取订单操作的消息 `message`、订单列表 `orders` 和状态码 `code`。

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

卖家订单查询：由于一个用户可以开设多个店铺，因此首先查询user_store，找到用户输入的user_id所开设的所有店铺的store_id，然后遍历这些store_id，在new_order中查询每个store_id，然后将得到的订单信息放入列表中，最后返回这些订单信息。

```
def get_seller_orders(self, user_id: str) -> (int, str, list):
    try:
        # 使用SQLAlchemy查询数据
        seller_stores =
self.conn.query(UserStore).filter_by(user_id=user_id).all()

        # print(seller_stores)

        seller_orders = []

        for store in seller_stores:
            orders =
self.conn.query(NewOrder).filter_by(store_id=store.store_id).all()
            order_dict=[]
            for order in orders:
                order_dict.append({
                    'store_id': order.store_id,
                    'order_id': order.order_id,
                    'status': order.status,
                })
            seller_orders.extend(order_dict)

        return 200, "ok", seller_orders
    except Exception as e:
        return 530, "{}".format(str(e)), []
```

接口将返回的字典列表翻译为json

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

用户取消订单：首先检查order_id和user_id是否存在且匹配，然后检查订单状态，只有还没发货的订单可以取消，因此如果订单为shipped和received，就返回状态错误。如果订单已经取消了就返回200并提示订单已经被取消了。如果订单状态是paid，就需要将付款退回给用户，这时需要查询new_order_detail表，根据order_id找到订单所需的总钱数，然后查询用户id，更新用户余额，并将订单状态更新为cancelled。

```
def cancel_order(self, user_id: str, order_id: str) -> (int, str):
    try:
        order_data = self.conn.query(NewOrder).filter_by(order_id=order_id).first()

        if order_data is None:
            return error.error_invalid_order_id(order_id)

        if order_data.user_id != user_id:
            return error.error_authorization_fail()

        if order_data.status == "shipped" or order_data.status == "received":
            return error.error_status_fail(order_id)

        if order_data.status == "cancelled":
            return 200, "Order is already cancelled."

        if order_data.status == "paid":
            # 获取订单详细信息
            order_detail_data = self.conn.query(NewOrderDetail).filter_by(order_id=order_id).all()
            total_price = sum([order_detail.price * order_detail.count for order_detail in order_detail_data])

            # 更新用户余额，将付款退还给用户
            user_data = self.conn.query(UserModel).filter_by(user_id=user_id).first()
            if user_data is None:
                return error.error_non_exist_user_id(user_id)

            # 计算退款金额
            refund_amount = total_price
            current_balance = user_data.balance
            new_balance = current_balance + refund_amount

            # 更新用户余额
            user_data.balance = new_balance

        # 取消订单，更新状态为 "cancelled"
        order_data.status = "cancelled"

        self.conn.commit()

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

    return 200, "ok"
```

自动取消订单：`OrderAutoCancel` 类在初始化时调用了 `db_conn.DBConn` 的构造函数，建立了数据库连接，并创建了一个定时器 `cancel_timer`。定时器 `cancel_timer` 被设置为每隔 60 秒执行一次 `cancel_unpaid_orders` 方法，即自动取消超过一分钟未支付的订单。在类的构造函数中，启动了定时器，并输出 "First start" 提示信息。`cancel_unpaid_orders` 方法被定时器执行，首先获取当前时间，并计算出一个时间间隔（1分钟前的时间）。通过查询数据库，找到所有状态为 'unpaid' 且创建时间在指定时间间隔之前的订单。这些订单被视为超时未支付的订单。遍历未支付订单列表，将每个订单的状态更新为 'cancelled'，并提交事务。如果在执行过程中出现异常，打印错误信息，但不中断程序执行。在方法执行结束后，重新设置定时器 `cancel_timer`，以便下一次定时执行，并输出 "Second start" 提示信息。

为了实现订单自动取消，`cancel_unpaid_orders` 方法获取当前时间，并查询 `new_order` 表，找到所有订单创建时间距离现在超过1分钟的订单，然后将状态设置为cancelled。方法结束前，还会重新建立一个线程，在60秒后再调用这个方法。这里的60秒可以改到更小的值，从而更频繁地轮询 `new_order` 表，更及时找出超时的订单并取消。该功能的实现也可以通过给每个订单都加一个定时器，但本项目为了效率的要求，60秒才执行一次轮询，可以保证一个超时的订单在超时2分钟之内一定会被取消。

```
class OrderAutoCancel(db_conn.DBConn):
    def __init__(self):
        db_conn.DBConn.__init__(self)
        self.cancel_timer = threading.Timer(60, self.cancel_unpaid_orders)
 # Timer executes every minute
        print('First start')
        self.cancel_timer.start()

    def cancel_unpaid_orders(self):
        try:
            current_time = datetime.now()
            time_interval = current_time - timedelta(minutes=1)

            unpaid_orders = (
                self.conn.query(NewOrderModel)
                .filter(NewOrderModel.status == 'unpaid',
NewOrderModel.created_at < time_interval)
                .all()
            )

            for order in unpaid_orders:
                order.status = 'cancelled'
                self.conn.commit()

        except Exception as e:
            print(f"Error canceling unpaid orders: {str(e)}")

        # Restart the timer
        self.cancel_timer = threading.Timer(60, self.cancel_unpaid_orders)
        print('Second start')
        self.cancel_timer.start()
```

在后端服务器运行的时候，创建 `start_order_auto_cancel` 函数并调用即可。

```python
def start_order_auto_cancel():
    scheduler = BackgroundScheduler()
    scheduler.add_job(OrderAutoCancel().cancel_unpaid_orders, 'interval', minutes=1)  # 每隔15分钟触发一次
    scheduler.start()
```

测试用例

测试用例相比第一次大作业，修改较少，

1.图书搜索功能

由于增加了测试用例，现有access中的文件不足以覆盖测试所需的前后端通信功能，所以首先要对其中的文件进行修改。

修改fe\access\book.py：

1. BookDB类中，测试用的书籍数据从默认的sqlite数据库读取

```python
class BookDB:
    def __init__(self, large: bool = False):
        parent_path = os.path.dirname(os.path.dirname(__file__))
        self.db_s = os.path.join(parent_path, "data/book.db")
        self.db_l = os.path.join(parent_path, "data/book_lx.db")
        if large:
            self.book_db = self.db_l
        else:
            self.book_db = self.db_s

    def get_book_count(self):
        conn = sqlite.connect(self.book_db)
        cursor = conn.execute("SELECT count(id) FROM book")
        row = cursor.fetchone()
        return row[0]

    def get_book_info(self, start, size) -> [Book]:
        books = []
        conn = sqlite.connect(self.book_db)
        cursor = conn.execute(
            "SELECT id, title, author, "
            "publisher, original_title, "
            "translator, pub_year, pages, "
            "price, currency_unit, binding, "
            "isbn, author_intro, book_intro, "
            "content, tags FROM book ORDER BY id "
            "LIMIT ? OFFSET ?",
            (size, start),
        )
        for row in cursor:
            book = Book()
            book.id = row[0]
            book.title = row[1]
            book.author = row[2]
            book.publisher = row[3]
            book.original_title = row[4]
```

```
            book.translator = row[5]
            book.pub_year = row[6]
            book.pages = row[7]
            book.price = row[8]

            book.currency_unit = row[9]
            book.binding = row[10]
            book.isbn = row[11]
            book.author_intro = row[12]
            book.book_intro = row[13]
            book.content = row[14]
            tags = row[15]
            picture = row[16]

            for tag in tags.split("\n"):
                if tag.strip() != "":
                    book.tags.append(tag)
            books.append(book)

        return books
```

2. 增加search_in_store与search_all函数，分别用来与后端的search_in_store与search_all接口通信，即构造json，向指定地址发送post请求，并返回查询到的数据与状态码。

```
def search_in_store(store_id,title,author,publisher,isbn,content,tags,book_intro,page=1,per_page=10):
    json ={
            'store_id':store_id,
            'title': title,
            'author': author,
            'publisher': publisher,
            'isbn': isbn,
            'content': content,
            'tags': tags,
            'book_intro': book_intro,
            'page': page,
            "per_page": per_page
        }
    url = urljoin(urljoin(conf.URL, "book/"), "search_in_store")
    r = requests.post(url, json=json)
    return r.status_code,r.json()

def search_all(title,author,publisher,isbn,content,tags,book_intro,page=1,per_page=10):
    json ={
            'title': title,
            'author': author,
            'publisher': publisher,
            'isbn': isbn,
            'content': content,
            'tags': tags,
            'book_intro': book_intro,
            'page': page,
```

```
            "per_page": per_page
        }
    url = urljoin(urljoin(conf.URL, "book/"), "search_all")
    r = requests.post(url, json=json)
    return r.status_code,r.json()
```

测试参数化局部搜索功能的正确性

该测试用例在`test_search_books_in_store.py`中实现

首先需要初始化正式运行前的操作：

1）指定新的卖家对应的用户名与密码、店铺名称，创建新的卖家与店铺，并通过`assert`确保店铺创建成功

2）创建`BookDB`实例，从对应的数据库中，随机取一定数量的图书存入该商店

3）随机选择`self.books`中的一本书，对'title', 'author', 'publisher', 'isbn', 'content', 'tags', 'book_intro'几个属性，分别以20%的概率，从中这本书的对应条目中抽取连续的两个字符存入`self.json`。

4）用`yield`等待上述操作完成后再进行正式的的测试。

这种随机化搜索条目的方法可以使得正确的搜索至少可以搜索到一本图书，方便确认搜索的正确性。

```
class TestSearchBooksInStore:
    @pytest.fixture(autouse=True)
    def pre_run_initialization(self, str_len=2):

        # 创建卖家与商店
        self.seller_id = "test_search_in_store_books_seller_id_{}".format(str(uuid.uuid1()))
        self.store_id = "test_search_in_store_books_store_id_{}".format(str(uuid.uuid1()))

        self.password = self.seller_id
        self.seller = register_new_seller(self.seller_id, self.password)

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

        # 向商店中添加图书
        book_db = book.BookDB(conf.Use_Large_DB)
        self.books = book_db.get_book_info(0, random.randint(1, 20))
        for b in self.books:
            code = self.seller.add_book(self.store_id, 0, b)
            assert code == 200

        # 构造搜索参数
        self.json = {
            "store_id": self.store_id,
            "title": "",
            "author": "",
            "publisher": "",
            "isbn": "",
            "content": "",
```

```
            "tags": "",
            "book_intro": ""
        }
        selected_book = random.choice(self.books)

        for i in ['title', 'author', 'publisher', 'isbn', 'content', 'tags',
'book_intro']:
            text_length = len(getattr(selected_book, i))
            if random.random() > 0.8 and text_length >= str_len:
                start_index = random.randint(0, text_length - 2)
                self.json[i] = getattr(selected_book, i)
[start_index:start_index + 2]

        yield
```

接下来，开始编写测试用例test_ok，来检验搜索功能的正确性。

1）利用access\book中的`search_in_store`函数发送搜索请求，获取搜索结果的id列表

2）利用`check_ok`函数在前端计算正确的结果id列表

2）计算两者长度是否一致，值是否一一对应，否则就通过assert抛出错误。

```
    def test_ok(self):
        json_list = list(self.json.values())
        code, res = book.search_in_store(json_list[0], json_list[1],
json_list[2], json_list[3], json_list[4],
                                         json_list[5], json_list[6],
json_list[7], 1, 10000000)
        assert code == 200

        res = [i['id'] for i in res['data']]
        print('搜索结果', res)

        right_answer = check_ok()
        print('真实结果', right_answer)
        assert len(right_answer) == len(res)
        # check_ok()
        for i in res:
            if i not in right_answer:
                assert False  # 搜索结果不正确

```

其中`check_ok`的实现如下，简单的来说就是，先获得有效搜索参数`processed_json`，再遍历`self.books`中是否有每个属性都含有对应字符串的书本，将其id加入`res`，其中list类型数据（tags）分开处理，遍历列表的每一个元素，检查是否存在于书本中的tags。

```
        def check_ok():
            processed_json = {}
            for key, value in self.json.items():
                if len(value) != 0 and key != 'store_id':
                    processed_json[key] = value
            print('pro', processed_json)
            if len(processed_json.keys()) == 0:
                return [book.id for book in self.books]
```

```
            res = []
            for d in self.books:
                flag = 0
                for key, substring in processed_json.items():
                    attr_value = getattr(d, key)
                    if attr_value is not None:
                        if isinstance(attr_value, str):
                            if attr_value.find(substring) == -1:
                                flag = 1
                        elif isinstance(attr_value, list):
                            for sub in substring:
                                if sub not in attr_value:
                                    flag = 1
                                    break
                        else:
                            flag = 1
                    else:
                        flag = 1

                if flag == 0:
                    res.append(d.id)

            return res
```

测试参数化全局搜索功能的正确性

该测试用例在`test_search_books_all.py`中实现

首先需要初始化正式运行前的操作。操作与局部搜索相似，与局部搜索不同的是，由于全局搜索时直接在book表中搜索全网存在的图书，因此不再需要创建卖家与店铺。

```
class TestSearchBooksAll:
    @pytest.fixture(autouse=True)
    def pre_run_initialization(self, str_len=2):

        # 测试的时候要用已有的数据，已有的数据存在book里，不应该添加conf.Use_Large_DB
        book_db = book.BookDB()
        self.books = book_db.get_book_info(0, book_db.get_book_count())
        self.json = {
            "title": "",
            "author": "",
            "publisher": "",
            "isbn": "",
            "content": "",
            "tags": "",
            "book_intro": ""
        }
        selected_book = random.choice(self.books)
        for i in ['title', 'author', 'publisher', 'isbn', 'content', 'tags',
'book_intro']:
            # if getattr(selected_book, i) is not None:
            text_length = len(getattr(selected_book, i))
            if random.random() > 0.8 and text_length >= str_len:
                start_index = random.randint(0, text_length - 2)
```

```
                self.json[i] = getattr(selected_book, i)
[start_index:start_index + 2]
        yield
```

测试用例`test_ok`用来检验搜索功能的正确性，其实现与局部搜索基本一致：

```
    def test_ok(self):
        def check_ok():
            processed_json = {}
            for key, value in self.json.items():
                if len(value) != 0:
                    processed_json[key] = value
            print('pro', processed_json)
            if len(processed_json.keys()) == 0:
                return [book.id for book in self.books]

            res = []
            for d in self.books:
                flag = 0
                for key, substring in processed_json.items():
                    attr_value = getattr(d, key)
                    if attr_value is not None:
                        if isinstance(attr_value, str):
                            if attr_value.find(substring) == -1:
                                flag = 1
                        elif isinstance(attr_value, list):
                            for sub in substring:
                                if sub not in attr_value:
                                    flag = 1
                                    break
                        else:
                            flag = 1
                    else:
                        flag = 1

                if flag == 0:
                    res.append(d.id)

            return res

        json_list = list(self.json.values())
        print('json_list',json_list)
        code, res = book.search_all(json_list[0], json_list[1],
json_list[2], json_list[3], json_list[4],
                                    json_list[5], json_list[6], 1,
100000000)
        assert code == 200
        res = [i['id'] for i in res['data']]
        print('搜索结果', len(res), res)
        right_answer = check_ok()
        print('真实结果', len(right_answer), right_answer)
        assert len(right_answer) == len(res)
        for i in res:
            if i not in right_answer:
                assert False  # 搜索结果不正确
```

2.订单状态

订单状态的检测是通过类`TestOrderStatus`来实现的。这个类用于测试订单的正常流程（下单->付款->发货->收货）和各种可能的异常情况。这个类里的每个测试方法包含多个断言，用于检查不同步骤的预期结果是否正确。

```
class TestOrderStatus:
    seller_id: str
    store_id: str
    buyer_id: str
    password: str
    buy_book_info_list: [Book]
    total_price: int
    order_id: str
    buyer: Buyer
```

`TestOrderStatus`方法基于在access目录下实现各个方法，包括`buyer.py`和`seller.py`。

buyer.py

`get_order_info`方法的功能是根据订单号查询到订单信息，这个功能对实现的接口`buyer_orders`发送post请求，以获取相应用户的所有id，然后再比对这些订单号与传入需要查询的订单号，最终得到相应订单号的信息。

```
def get_order_info(self, order_id):
    json = {
        "user_id": self.user_id,
        "order_id": order_id,
    }
    url = urljoin(self.url_prefix, "buyer_orders")
    headers = {"token": self.token}
    r = requests.post(url, headers=headers, json=json)
    assert r.status_code == 200
    orders_info = r.json()
    order_info = {}
    for o in orders_info['orders']:
        if o['order_id'] == order_id:
            order_info = o
    assert len(order_info.keys()) != 0
    return order_info
```

`receive_order`方法是对`receive_order`接口发送post请求，并获取状态码，是对收货流程的测试。

```
    def receive_order(self, order_id):
        json = {
            "user_id": self.user_id,
            "order_id": order_id,
        }
        url = urljoin(self.url_prefix, "receive_order")
        headers = {"token": self.token}
        r = requests.post(url, headers=headers, json=json)
        return r.status_code
```

`cancel_order`方法是对`cancel_order`接口发送post请求，用于测试取消订单的流程是否成功。

```
def cancel_order(self, order_id):
    json = {
        "user_id": self.user_id,
        "order_id": order_id,
    }
    url = urljoin(self.url_prefix, "cancel_order")
    headers = {"token": self.token}
    r = requests.post(url, headers=headers, json=json)
    return r.status_code
```

seller.py

`ship_order`是对seller的`ship_order`接口的检验，发送post请求查看发货流程是否成功。

```
def ship_order(store_id, order_id):
    json = {
        "store_id": store_id,
        "order_id": order_id
    }
    url = urljoin(urljoin(conf.URL, "seller/"), "ship_order")
    r = requests.post(url, json=json)
    return r.status_code
```

TestOrderStatus类的各方法

`pre_run_initialization`方法的目的是在数据库中提前插入好订单流程测试所需要的前提，它在每个测试方法之前都会运行。首先为卖家、店铺和买家生成唯一的ID，然后使用卖家创建书籍，使用买家创建并购买订单。

```
    @pytest.fixture(autouse=True)
    def pre_run_initialization(self):
        # do before test
        self.seller_id = "test_order_status_seller_id_{}".format(str(uuid.uuid1()))
        self.store_id = "test_order_status_store_id_{}".format(str(uuid.uuid1()))
        self.buyer_id = "test_order_status_buyer_id_{}".format(str(uuid.uuid1()))
        self.password = self.seller_id
        print(self.store_id,self.seller_id)
        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
        # print(self.buy_book_info_list)
        print(gen_book.buy_book_id_list)
        assert ok
        self.b = register_new_buyer(self.buyer_id, self.password)
        self.buyer = self.b
        code, self.order_id = self.b.new_order(self.store_id, buy_book_id_list)
```

```
        assert code == 200

        order_info=self.b.get_order_info(self.order_id)
        assert order_info['status'] == 'unpaid'

        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
        yield
```

以下方法的功能是测试正常的订单流程是否可以顺利完成。包括买家充值、付款、发货、收货，最后断言订单状态符合预期。

```
    # 正常订单流程
    def test_ok(self):
        # 买家充值
        code = self.buyer.add_funds(self.total_price)
        assert code == 200

        # 买家付钱
        code = self.buyer.payment(self.order_id)
        assert code == 200
        order_info=self.b.get_order_info(self.order_id)
        assert order_info['status'] == 'paid'

        # 卖家发货
        code = ship_order(self.store_id,self.order_id)
        assert code == 200
        order_info=self.b.get_order_info(self.order_id)
        assert order_info['status'] == 'shipped'

        # 买家收货
        code = self.b.receive_order(self.order_id)
        assert code == 200
        order_info=self.b.get_order_info(self.order_id)
        assert order_info['status'] == 'received'
```

买家可以实现自己主动取消订单。以下方法测试了买家取消订单流程，包括买家取消订单、卖家发货，以及买家收货，此时应该无法收发货成功。

```
    # 买家取消订单流程
    def test_cancel(self):
        # 买家取消订单
        code = self.b.cancel_order(self.order_id)
        assert code == 200
        order_info=self.b.get_order_info(self.order_id)
        assert order_info['status'] == 'cancelled'

        # 卖家发货
```

```
        code = ship_order(self.store_id,self.order_id)
        assert code != 200

        # 买家收货
        code = self.b.receive_order(self.order_id)
        assert code != 200
```

测试卖家在买家付款前发货，预期收发货操作都会失败。

```
    def test_ship_before_pay(self):
        # 卖家发货
        code = ship_order(self.store_id,self.order_id)
        assert code != 200

        # 买家收货
        code = self.b.receive_order(self.order_id)
        assert code != 200
```

测试买家在卖家发货前收货，预期收货操作会失败。

```
    def test_receive_before_ship(self):
        # 买家充值
        code = self.buyer.add_funds(self.total_price)
        assert code == 200

        # 买家付钱
        code = self.buyer.payment(self.order_id)
        assert code == 200
        order_info=self.b.get_order_info(self.order_id)
        assert order_info['status'] == 'paid'

        # 买家收货
        code = self.b.receive_order(self.order_id)
        assert code != 200
```

测试订单自动取消功能，需要等待一段时间（120秒），然后检查订单状态是否已自动设置为"cancelled"，以验证自动取消的功能。

```
    def test_auto_cancel(self):
        time.sleep(120)

        order_info=self.b.get_order_info(self.order_id)
        assert order_info['status'] == 'cancelled'
```

TestGetOrder类的各方法

这个 Fixture 主要用于在测试用例执行之前初始化一些共享的数据和状态，生成了三个唯一的卖家、商店和买家的标识符，并为每个卖家创建一个密码，然后注册这些买家。此外，还创建了一组 `GenBook` 的实例，这些实例用于生成书籍信息。

```
@pytest.fixture(autouse=True)
def pre_run_initialization(self):
    # 初始化一些测试所需的变量和数据
```

```
    # 这个 Fixture 被标记为自动使用（autouse=True），在每个测试用例运行之前都会被调用
    # 这个 Fixture 主要用于为测试用例准备一些共享的数据和状态
    self.seller_ids = []
    self.store_ids = []
    self.buyer_ids = []
    self.passwords = []
    self.buyers = []
    self.gen_books = []

    # 循环3次，为每个循环生成一组唯一的标识符，并初始化一些测试对象
    for i in range(3):

 self.seller_ids.append("test_new_order_seller_id_{}".format(str(uuid.uuid1(
))))

 self.store_ids.append("test_new_order_store_id_{}".format(str(uuid.uuid1())
))

 self.buyer_ids.append("test_new_order_buyer_id_{}".format(str(uuid.uuid1())
))
        self.passwords.append(self.seller_ids[i])
        self.buyers.append(register_new_buyer(self.buyer_ids[i],
self.passwords[i]))
        self.gen_books.append(GenBook(self.seller_ids[i],
self.store_ids[i]))
    yield  # yield 用于分隔 setup 部分和 teardown 部分
```

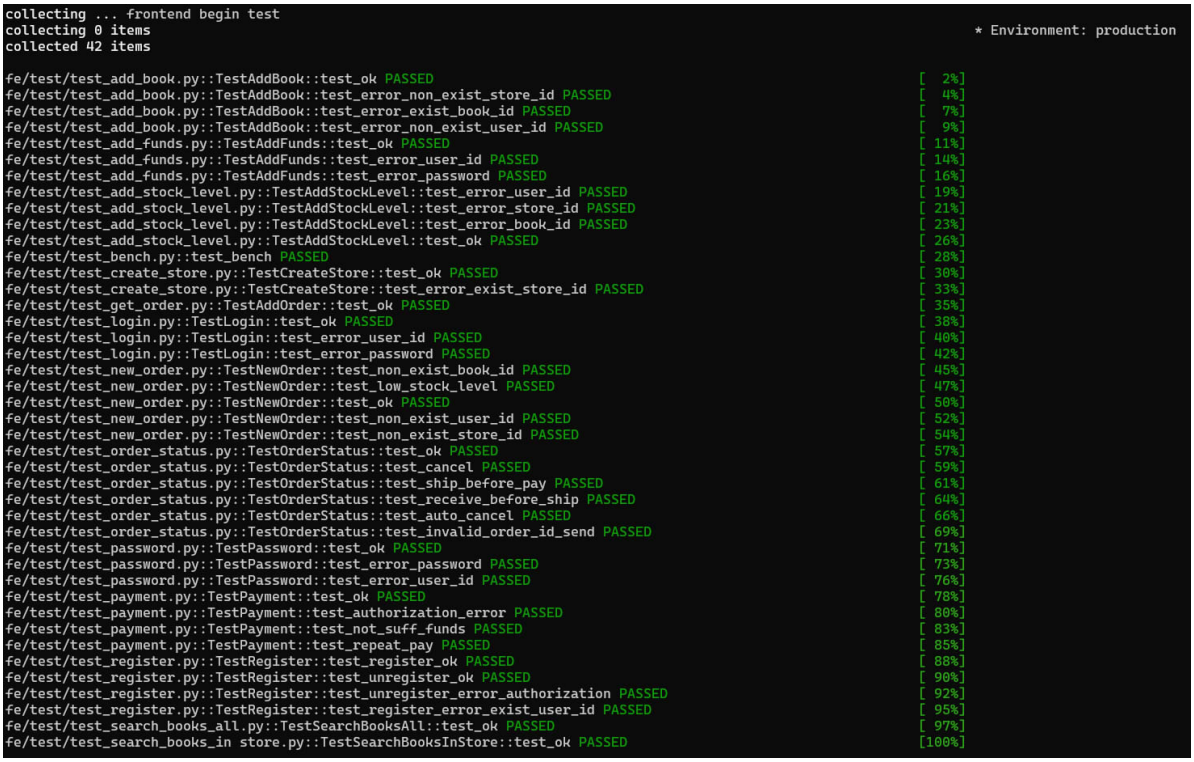
这个测试用例通过使用之前初始化的数据执行了一系列的操作。首先，使用 `GenBook` 的实例生成了购买书籍的信息，然后为每个买家执行了新订单的操作，并断言返回的状态码为200，获取每个卖家和买家的订单信息，并同样断言返回的状态码为200。

```
def test_ok(self):
    # 在前面的 Fixture 中初始化的数据的基础上，执行一系列测试操作
    buy_book_id_lists = []

    # 循环3次，为每个循环生成购买书籍的信息
    for i in range(3):
        ok, buy_book_id_list = self.gen_books[i].gen(
            non_exist_book_id=False, low_stock_level=False
        )
        assert ok
        buy_book_id_list = [(i[0], 1) for i in buy_book_id_list]
        buy_book_id_lists.append(buy_book_id_list)

    # 为每个买家执行新的订单操作，并断言返回的状态码为200
    for i in range(2):
        for j in range(3):
            code, _ = self.buyers[i].new_order(self.store_ids[j],
buy_book_id_lists[j])
            assert code == 200

    # 获取卖家的订单，并断言返回的状态码为200
    for i in range(3):
        code, result = get_seller_order(self.seller_ids[i])
        assert code == 200
```

```
24
25      # 获取买家的订单，并断言返回的状态码为200
26      for i in range(2):
27          code, result = self.buyers[i].get_seller_order()
28          assert code == 200
29
```

四、实验总结

经过以上后端逻辑、接口和测试用例的实现，再次执行测试命令，得到以下结果，可以看到所有的测试用例都通过。



从book_lx.db中抽取部分数据，运行覆盖率测试，得到如下结果：

```
(bookstore) PS D:\githubcode\2\bookstore> 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\book.py                             76     14     38     15    73%
be\model\buyer.py                           134     32     54     11    74%
be\model\db_conn.py                          14      0      0      0   100%
be\model\error.py                            25      1      0      0    96%
be\model\order_auto_cancel.py                25      1      4      1    93%
be\model\seller.py                           90     21     34      4    77%
be\model\store.py                            71      0      0      0   100%
be\model\user.py                            124     25     32      2    78%
be\serve.py                                  43      1      2      1    96%
be\view\__init__.py                           0      0      0      0   100%
be\view\auth.py                              42      0      0      0   100%
be\view\book.py                              34      0      0      0   100%
be\view\buyer.py                             56      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                            78      1      8      1    98%
fe\access\buyer.py                           66      0      6      1    99%
fe\access\new_buyer.py                        8      0      0      0   100%
fe\access\new_seller.py                       8      0      0      0   100%
fe\access\seller.py                          44      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                         7      2      0      0    71%
fe\test\test_create_store.py                 20      0      0      0   100%
fe\test\test_get_order.py                    53      0      6      0   100%
fe\test\test_login.py                        28      0      0      0   100%
fe\test\test_new_order.py                    40      0      0      0   100%
fe\test\test_order_status.py                 92      4      6      2    94%
fe\test\test_password.py                     33      0      0      0   100%
fe\test\test_payment.py                      60      1      4      1    97%
fe\test\test_register.py                     31      0      0      0   100%
fe\test\test_search_books_all.py             64      6     40      5    88%
fe\test\test_search_books_in store.py        71     12     40      5    77%
---------------------------------------------------------------------------
TOTAL                                      1878    125    354     52    91%
(bookstore) PS D:\githubcode\2\bookstore>
```

可以看到，由于测试用例覆盖了较为全面的流程，测试覆盖率较高。

在代码管理中使用了git工具，可以在仓库https://github.com/tuziTZ/2中查看。