bookstore实验报告

姓名	学号
王一平	10185300235
杨爽	10174102137

bookstore实验报告

一、 项目介绍
二、 项目目录结构
三、 数据库设计
1、绘制ER图
2、表格介绍
1）users用户表
2）user_store用户_商店表
3）store商家表
4）new_order订单表、new_order_detail新订单表
6）not_pay_order未付款订单表
7）book图书表、search_title搜索题目表、search_author搜索作者表、search_tags搜索标签表、search_details搜索内容表：
四、 功能实现
用户端功能
1. 用户注册
2.用户登录
3. 退出登录
4. 用户注销
5. 修改密码
6. 查询历史
买家功能
1. 下单
2. 付款
3. 充值
4. 确认收货
5. 取消订单
6. 自动取消订单
卖家功能
1. 添加书籍信息
2. 增加库存
3. 创建商铺
4. 发货
搜索图书
1、按照题目搜索
2、按照作者搜索
3、按照标签搜索
4、按照内容搜索
五、代码测试
六、版本控制与分工
七、项目优缺点与总结

一、 项目介绍

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，购买者可能通过网站购买。

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。

支持下单->付款->发货->收货，流程。

1.实现对应接口的功能，见doc下面的.md文件描述 （60%分数）

其中包括：

1)用户权限接口，如注册、登录、登出、注销

2)买家用户接口，如充值、下单、付款

3)卖家用户接口，如创建店铺、填加书籍信息及描述、增加库存

通过对应的功能测试，所有test case都pass

测试下单及付款两个接口的性能（最好分离负载生成和后端），测出支持的每分钟交易数，延迟等

2.为项目添加其它功能： （40%分数）

1)实现后续的流程

发货 -> 收货

2)搜索图书

用户可以通过关键字搜索，参数化的搜索方式；
如搜索范围包括，题目，标签，目录，内容；全站搜索或是当前店铺搜索。
如果显示结果较大，需要分页
(使用全文索引优化查找)

3)订单状态，订单查询和取消定单

用户可以查自己的历史订单，用户也可以取消订单。

取消定单（可选项，加分 +5~10），买家主动地取消定单，如果买家下单经过一段时间超时后，如果买家未付款，定单也会自动取消。

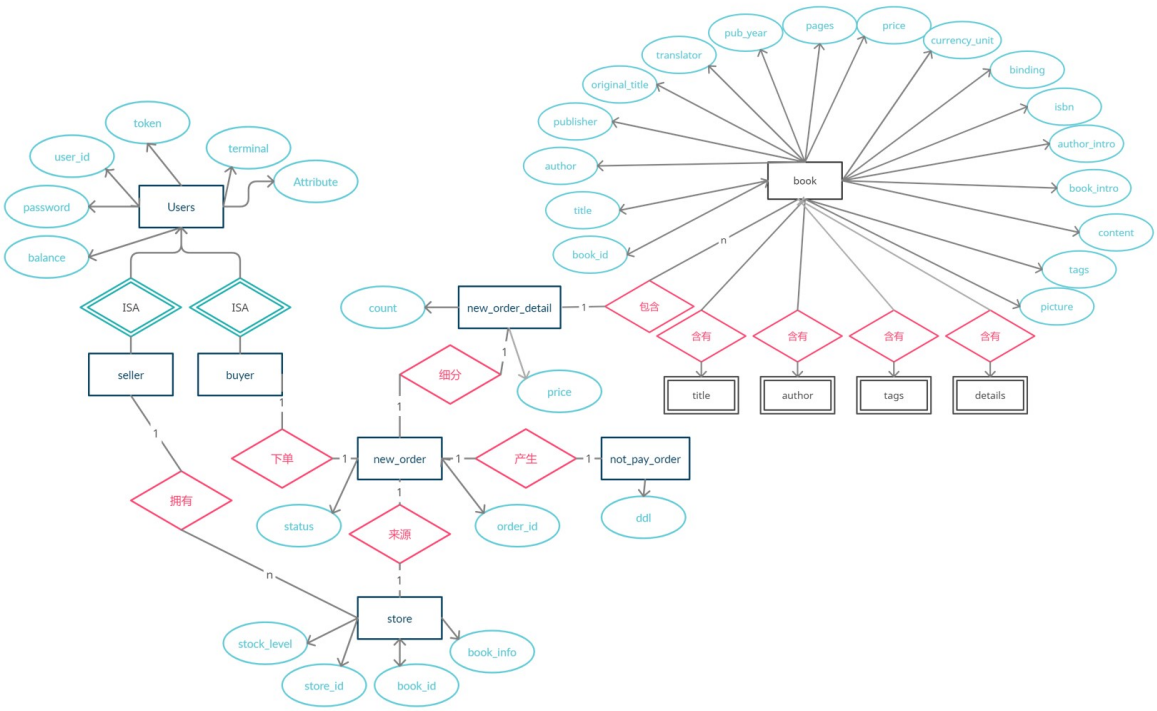
二、 项目目录结构

```
bookstore
  |-- be                              mock的后端
        |-- model             每个功能的函数实现
        |-- view                            路由
        |-- ....
```

```
  |-- doc                          JSON API规范说明
  |-- fe                           前端代码
        |-- access
        |-- bench                  效率测试
        |-- data
            |-- book.db            sqlite 数据库(book.db，较少量的测试数据)
            |-- book_lx.db         sqlite 数据库(book_lx.db， 较大量的测试数据，要从网盘下载)
            |-- scraper.py         从豆瓣爬取的图书信息数据
        |-- test                   功能性测试（不要修改这里的文件，可以提pull request或bug）
        |-- conf.py                测试参数，修改这个文件以适应自己的需要
        |-- conftest.py            pytest初始化配置，修改这个文件以适应自己的需要
        |-- ....
  |-- ....
```

三、 数据库设计

1、绘制ER图



2、表格介绍

本项目一共包含11张数据表格，下面将一一介绍它们的作用。

1）users用户表

users表主要用来存用户（包括买家和卖家）的基本信息，其中包括用户id、密码、余额、登录缓存令牌和终端标记，主键为用户id。

```
class User(base):
    __tablename__ = 'users'  # postgres中表格名不能为user
    user_id = Column('user_id', TEXT, primary_key=True)
    password = Column('password', TEXT, nullable=False)
    balance = Column('balance', Integer, nullable=False)  # 余额
    token = Column('token', TEXT)  # 登录缓存令牌
    terminal = Column('terminal ', TEXT)  # 标记终端
```

2）user_store用户_商店表

user_store表是一张关系表，用来存储用户（卖家）和商店的关系，一个卖家可以开多家书店，但一家书店只能有一个卖家，所以卖家和商店是一对多的关系。该表包括user_id和store_id，两者共同组成主键，且user_id是用户表的外键，store_id是商家表的外键。也可以通过将store_id的信息存放在user表中来取代该表的内容，但是毕竟大多数用户都不是卖家，将卖家分离出来会使得之后查询的效率变高。

```
class UserStore(base):
    __tablename__ = 'user_store'
    user_id = Column('user_id', TEXT, primary_key=True)
    store_id = Column('store_id', TEXT, primary_key=True)
```

3）store商家表

store表主要用来存储书店的库存信息，包括书籍信息和书籍的库存量。一个书店会售卖多本书，同样的书也可以在多个不同的书店售卖，这是一个多对多的关系，所以这里用store_id, book_id联合组成表的主键，并记录书籍信息和书籍库存量，满足功能实现需求。

```
class Store(base):
    __tablename__ = 'store'
    store_id = Column('store_id', TEXT, primary_key=True)
    book_id = Column('book_id', TEXT, primary_key=True)
    book_info = Column('book_info', TEXT)
    stock_level = Column('stock_level', Integer)
```

4）new_order订单表、new_order_detail新订单表

需要用到订单信息的功能有：
下单，付款，确认收货，发货，取消订单
其中，下单，付款，取消订单是需要知道订单内书籍的详细信息(书籍信息，价格，数量)，而收货，发货是不需要的。
如果用一张表记录所有的信息，表会很大，不容易维护，并且影响功能的执行效率，所以将订单情况和详细情况分成两张表
这两个表用来记录订单的信息。需要包括：
订单号(order_id)，买家信息(buyer_id)，商店信息（store_id)，购买的书单(book_id),购买数量(count)，购买价格(price).
在new_order表中记录order_id，user_id，store_id，status(订单状态)其中order_id是主键
在new_order_detail表存放order_id，book_id，count，price其中order_id和book_id是联合主键

```
class NewOrder(base):
    __tablename__ = 'new_order'
    order_id = Column('order_id', TEXT, primary_key=True)
    user_id = Column('user_id', TEXT)
    store_id = Column('store_id', TEXT)
    status = Column('status', Integer)
```

```
class NewOrderDetail(base):
    __tablename__ = 'new_order_detail'
    order_id = Column('order_id ', TEXT, primary_key=True)
    book_id = Column('book_id', TEXT, primary_key=True)
    count = Column('count ', Integer)
    price = Column('price ', Integer)
```

6）not_pay_order未付款订单表

存放订单号和过期时间，方便后续自动取消订单操作

```
class NotPayOrder(base):
    __tablename__ = 'not_pay_order'
    order_id = Column('order_id', TEXT, primary_key=True)
    ddl = Column('ddl', DateTime, nullable=False)
```

7）book图书表、search_title搜索题目表、search_author搜索作者表、search_tags搜索标签表、search_details搜索内容表：

这5个表格都是为了便于搜索图书所创建，虽然他们看似都是冗余的数据，没有这几张表格也可以完成搜索的查询，但是这样的话查询效率就会大大下降。图书表book是用来存放书籍的具体信息的，其余4张表都是基于倒排索引创建的表格，可以提高查询的效率。具体的介绍会在后面搜索图书部分给出。

```python
class Book(base):
    __tablename__ = 'book'
    book_id = Column('book_id', TEXT, primary_key=True)
    title = Column('title', TEXT)
    author = Column('author', TEXT)
    publisher = Column('publisher', TEXT)
    original_title = Column('original_title', TEXT)
    translator = Column('translator', TEXT)
    pub_year = Column('pub_year', TEXT)
    pages = Column('pages', Integer)
    price = Column('price', Integer)
    currency_unit = Column('currency_unit', TEXT)
    binding = Column('binding', TEXT)
    isbn = Column('isbn', TEXT)
    author_intro = Column('author_intro', TEXT)
    book_intro = Column('book_intro', TEXT)
    content = Column('content', TEXT)
    tags = Column('tags', TEXT)
    picture = Column('picture', LargeBinary)  # 二进制图片类型
```

```python
# 搜索标题表
class SearchTitle(base):
    __tablename__ = 'search_title'
    title = Column("title", TEXT, nullable=False, index=True)
    book_id = Column("book_id", TEXT, ForeignKey('book.book_id'), nullable=False)
    __table_args__ = (
        PrimaryKeyConstraint('title', 'book_id'),
        {},
    )


# 搜索标签表
class SearchTags(base):
    __tablename__ = 'search_tags'
    tags = Column("tags", TEXT, nullable=False, index=True)
    book_id = Column("book_id", TEXT, ForeignKey('book.book_id'), nullable=False)
    __table_args__ = (
        PrimaryKeyConstraint('tags', 'book_id'),
        {},
    )
```

```python
# 搜索作者表
class SearchAuthor(base):
    __tablename__ = 'search_author'
    author = Column("author", TEXT, nullable=False, index=True)
    book_id = Column("book_id", TEXT, ForeignKey('book.book_id'), nullable=False)
    __table_args__ = (
        PrimaryKeyConstraint('author', 'book_id'),
        {},
    )


# 搜索内容表
class SearchDetails(base):
    __tablename__ = 'search_details'
    details = Column("details", TEXT, nullable=False, index=True)
    book_id = Column("book_id", TEXT, ForeignKey('book.book_id'), nullable=False)
    __table_args__ = (
        PrimaryKeyConstraint('details', 'book_id'),
        {},
    )
```

四、 功能实现

用户端功能

1. 用户注册

代码实现

```python
# 用户注册
def register(user_id: str, password: str):
    try:
        terminal = "terminal_{}".format(str(time.time()))
        token = jwt_encode(user_id, terminal)
        query1 = db.session.query(st.User).filter(st.User.user_id==user_id).one_or_none()
        if query1 is not None:
            return error.error_exist_user_id(user_id)
        obj=st.User(user_id=user_id, password=password, balance=0, token=token, terminal=terminal)
        db.session.add(obj)
        print(user_id)
        db.session.commit()
    except exc.SQLAlchemyError as e:
        print(e)
        return error.error_exist_user_id(user_id, e)
    return 200, "ok"
```

功能说明：

1. 注册时，从url中获取要注册的用户的用户名和密码，在这里不允许存在用户名相同的用户。
2. 访问users关系表，若表中已存在此用户，返回512，注册不成功；若表中不存在此用户，初始化信息，并插入表中，返回200。

2.用户登录

代码实现

```python
# 用户登录
def login(user_id: str, password: str, terminal: str) -> (int, str, str):
    token = ""
    try:
        code, message = check_password(user_id, password)
        if code != 200:
            return code, message, ""
        token = jwt_encode(user_id, terminal)
        query1 = db.session.query(st.User).filter(st.User.user_id == user_id).update(
            {st.User.token: token, st.User.terminal: terminal}
        )
        if query1 == 0:
            return error.error_authorization_fail() + ("",)
        db.session.commit()
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e)), ""
    except BaseException as e:
        return 530, "{}".format(str(e)), ""
    return 200, "ok", token
```

功能说明：

> 通过url传入要登录的用户名和密码，并对其的正确性进行判断。

3. 退出登录

代码实现

```python
# 退出登录
def logout(user_id: str, token: str) -> (int, str):  # 此处原本是bool
    try:
        code, message = check_token(user_id, token)
        if code != 200:
            return code, message
        terminal = "terminal_{}".format(str(time.time()))
        dummy_token = jwt_encode(user_id, terminal)

        query1 = db.session.query(st.User).filter(st.User.user_id == user_id).update(
            {st.User.token: dummy_token, st.User.terminal: terminal}
        )
        if query1 == 0:
            return error.error_authorization_fail()
        db.session.commit()
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

> 通过url传入要登出的用户名和密码，并对其的正确性进行判断。

4. 用户注销

代码实现

```python
# 用户注销
def unregister(user_id: str, password: str) -> (int, str):
    try:
        code, message = check_password(user_id, password)
        if code != 200:
            return code, message

        query1 = db.session.query(st.User).filter(st.User.user_id == user_id).delete()
        if query1 == 1:
            db.session.commit()
        else:
            return error.error_authorization_fail()
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

通过url传入要注销的用户名和密码，并对其的正确性进行判断。
若判断正确，从users表中删除此用户相关信息，返回正确码200
若存在异常，返回异常码

5. 修改密码

代码实现

```python
# 更改密码
def change_password(user_id: str, old_password: str, new_password: str) -> (int, str):  # 此处原
    try:
        code, message = check_password(user_id, old_password)
        if code != 200:
            return code, message
        terminal = "terminal_{}".format(str(time.time()))
        token = jwt_encode(user_id, terminal)
        query1 = db.session.query(st.User).filter(st.User.user_id == user_id).update(
            {st.User.password: new_password, st.User.token: token, st.User.terminal: terminal}
        )
        if query1 == 0:
            return error.error_authorization_fail()
        db.session.commit()
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明:

通过url传入要修改的用户名，旧密码以及新密码，斌谷底用户名和旧密码的正确性进行判断。
若判断正确，对该条记录执行update操作，修改密码，并返回正确码200
若出现异常，返回异常码

6. 查询历史

代码实现

```
def history(user_id: str):
    try:
        if not db.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        query1 = db.session.query(st.NewOrder).join(st.NewOrderDetail,
                                                    st.NewOrder.order_id == st.NewOrderDetail.order_id
                                                    ).filter(st.NewOrder.user_id == user_id)
        row = query1.all()
        print(row)
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

通过url传入要查询历史订单的用户名
对new_order表和new_order_detail通过order_id相等进行连接，查找用户的历史记录，返回给用户并返回正确码200
若出现异常，返回异常码

测试方法

考虑两种情况，正确输入用户名和错误输入用户名

买家功能

1. 下单

代码实现

```
# 买家下单
def new_order(user_id: str, store_id: str, id_and_count: [(str, int)]) -> (int, str, str):
    order_id = ""
    try:
        if not db.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id) + (order_id,)
        if not db.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()))

        for book_id, count in id_and_count:  # 对于订单中的每一类书
            query1 = db.session.query(st.Store).filter(st.Store.store_id == store_id,
                                                       st.Store.book_id == book_id)
            row = query1.one_or_none()
            if row is None:
                return error.error_non_exist_book_id(book_id) + (order_id,)
            stock_level = row.stock_level
            book_info = row.book_info
            book_info_json = json.loads(book_info)
            price = book_info_json.get("price")
            if stock_level < count:
                return error.error_stock_level_low(book_id) + (order_id,)
```

```
            query2 = db.session.query(st.Store).filter(
                st.Store.store_id == store_id, st.Store.book_id == book_id,
                st.Store.stock_level >= count).update(
                {st.Store.stock_level: st.Store.stock_level - count}, synchronize_session="evaluate"
            )
            if query2 == 0:
                return error.error_stock_level_low(book_id) + (order_id,)
            # 新增一条订单详细信息
            db.session.add(st.NewOrderDetail(order_id=uid, book_id=book_id, count=count, price=price))
        # 新增一条订单信息
        db.session.add(st.NewOrder(order_id=uid, store_id=store_id, user_id=user_id, status=0))
        # 新增一条未付款订单消息
        time = datetime.datetime.utcnow() + datetime.timedelta(seconds=10)
        db.session.add(st.NotPayOrder(order_id=uid, ddl=time))
        db.session.commit()
        order_id = uid
    except exc.SQLAlchemyError as e:
        logging.info("528, {}".format(str(e)))
        return 528, "{}".format(str(e)), ""
    except BaseException as e:
        logging.info("530, {}".format(str(e)))
        return 530, "{}".format(str(e)), ""
    return 200, "ok", order_id
```

功能说明

1. 从url中获取此订单信息(购买者:user_id，要购买的店铺id:store_id，要购买的书籍的名称和数量[(book1,count1),(book2,count2)...]
2. 对user_id, store_id,书店中是否有足够的书籍售卖进行检验，若都符合进行后续操作；若失败返回响应的错误码
3. 减少书店中的相关书籍的库存量，获取当前时间新增一条订单信息，详细订单信息，和未付款订单信息(将相关信息插入new_order, new_order_detail, not_pay_order表中)，无异常返回正确码200
4. 若存在异常，返回响应异常码

2. 付款

代码实现

```
# 买家付款
def payment(user_id: str, password: str, order_id: str) -> (int, str):
    # conn = self.conn
    try:
        query = db.session.query(st.NotPayOrder).filter(st.NotPayOrder.order_id == order_id)
        row = query.one_or_none()
        if row is None:
            return error.error_invalid_order_id(order_id)
        # 找到订单信息
        query1 = db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id)
        row = query1.one_or_none()
        if row is None:
            return error.error_invalid_order_id(order_id)
        order_id = row.order_id
        buyer_id = row.user_id
        store_id = row.store_id
        if buyer_id != user_id:
            return error.error_authorization_fail()
        # 查看买家用户余额是否足够
        query2 = db.session.query(st.User).filter(st.User.user_id == buyer_id)
        row = query2.one_or_none()
        if row is None:
            return error.error_non_exist_user_id(buyer_id)
        balance = row.balance
        if password != row.password:
```

```
            return error.error_authorization_fail()
        # 获取卖家id
        query3 = db.session.query(st.UserStore).filter(st.UserStore.store_id == store_id)
        row = query3.one_or_none()
        if row is None:
            return error.error_non_exist_store_id(store_id)
        seller_id = row.user_id
        if not db.user_id_exist(seller_id):
            return error.error_non_exist_user_id(seller_id)
        # 获取订单详细信息
        query4 = db.session.query(st.NewOrderDetail).filter(st.NewOrderDetail.order_id == order_id).all()
        total_price = 0
        for row in query4:  # 查看每一本书的情况，计算订单总金额
            count = row.count
            price = row.price
            total_price = total_price + price * count
        if balance < total_price:
            return error.error_not_sufficient_funds(order_id)
        # 买家账户扣钱
        query5 = db.session.query(st.User).filter(st.User.user_id == buyer_id, st.User.balance >= total_pr
            {st.User.balance: st.User.balance - total_price}, synchronize_session="evaluate")
        if query5 == 0:
            return error.error_not_sufficient_funds(order_id)
        # 卖家账户加钱
```

```
        query6 = db.session.query(st.User).filter(st.User.user_id == seller_id).update(
            {st.User.balance: st.User.balance + total_price}, synchronize_session="evaluate")
        if query6 == 0:
            return error.error_non_exist_user_id(buyer_id)
        #  付款完成，更改订单状态
        query7 = db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id).update(
            {st.NewOrder.status: 1}
        )
        if query7 == 0:
            return error.error_invalid_order_id(order_id)
        # 删除未支付表中订单信息
        db.session.query(st.NotPayOrder).filter(st.NotPayOrder.order_id == order_id).delete()
        db.session.commit()
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

1. 从url中获取要付款的订单信息(购买者的用户名:user_id，购买者的密码:password，要付款的订单号:order_id)
2. 对相关信息进行检验(是否存在此用户，用户名密码是否匹配，是否存在此订单号，购买者账户中的余额是否足够)，若信息全部检验通过，继续后续操作，否则返回相应的错误码
3. 通过订单号访问new_order_detail表，获取计算订单详细信息(卖家id：seller_id，订单总金额：total_price)
4. 从买家账户扣钱，卖家账户加钱，将订单信息从not_pay_order表中删去
5. 付款完成，将订单状态(status)从0置为1，方便后续操作，并返回正确码200
6. 若捕获到异常，返回异常码

3. 充值

代码实现

```python
# 买家充值
def add_funds(user_id, password, add_value) -> (int, str):
    try:
        query1 = db.session.query(st.User).filter(st.User.user_id == user_id)
        row = query1.one_or_none()
        if row is None:
            return error.error_authorization_fail()
        if row.password != password:
            return error.error_authorization_fail()
        query2 = db.session.query(st.User).filter(st.User.user_id == user_id).update(
            {st.User.balance: st.User.balance + add_value}, synchronize_session="evaluate")
        if query2 == 0:
            return error.error_non_exist_user_id(user_id)
        db.session.commit()

    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

1. 从url中获取要充值的用户用户名，密码和要充值的金额(uesr_id, password, add_value)
2. 对用户和密码的相关信息进行检验，若正确进行后续操作，若失败返回相应的错误码
3. 更改user_id在users表中的balance信息，将要充值的金额加进去(balance=balance+add_vale)，操作成功返回正确码200
4. 若捕获到异常返回异常码

4. 确认收货

代码实现

```python
def shipping(user_id: str, order_id: str):
    try:
        if not db.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        # 订单是否存在
        query1 = db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id)
        row = query1.one_or_none()
        if row is None:
            return error.error_invalid_order_id(order_id)
        # 查看订单是否发货
        if row.status != 2:
            return error.error_not_receive(order_id)
        # 订单完成，更改订单状态
        db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id).update(
            {st.NewOrder.status: 3}
        )
        db.session.commit()

    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

1. 从url中获取相关信息(购买者id:user_id，订单号:order_id)
2. 对相关信息进行检验(购买者id，订单id是否存在,订单是否已取消，订单是否发货)，若user_id，order_id都没有问题且订单没有取消，且订单已经发货执行后续操作
3.将订单状态改为3，若操作成功，返回成功码200，若失败，捕获异常，并返回异常码

5. 取消订单

代码实现

```
def cancel_order(buyer_id: str, order_id: str):
    try:
        if not db.user_id_exist(buyer_id):
            return error.error_non_exist_user_id(buyer_id)
        # 订单是否存在
        query1 = db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id)
        row = query1.one_or_none()
        if row is None:
            return error.error_invalid_order_id(order_id)
        status = row.status  # 订单状态
        store_id = row.store_id  # 卖家店铺
        query2 = db.session.query(st.UserStore).filter(st.UserStore.store_id == store_id).one_or_none()
        seller_id = query2.user_id  # 卖家用户名
        if status == 0:  # 用户下单(0)后取消
            # 查看是否在未付款订单中
            queryx = db.session.query(st.NotPayOrder).filter(st.NotPayOrder.order_id == order_id).one_o
            if queryx is None:
                return error.error_invalid_order_id(order_id)
            # 删除在未付款表中的此订单
            db.session.query(st.NotPayOrder).filter(st.NotPayOrder.order_id == order_id).delete()
            # 更改订单状态
            db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id).update(
                {st.NewOrder.status: -1}
            )
```

```
            # 更改卖家库存
            query3 = db.session.query(st.NewOrderDetail).filter(st.NewOrderDetail.order_id == order_id).
            for row1 in query3:  # 查看订单中每一类书的情况，更改卖家信息
                count = row1.count
                book_id = row1.book_id
                query4 = db.session.query(st.Store).filter(st.Store.store_id == store_id, st.Store.book_
                row2 = query4.one_or_none()
                if row2 is None:
                    return error.error_non_exist_book_id(book_id)

                # 增加书店中书的库存量
                query5 = db.session.query(st.Store).filter(
                    st.Store.store_id == store_id, st.Store.book_id == book_id).update(
                    {st.Store.stock_level: st.Store.stock_level + count}, synchronize_session="evaluate"
                )
                if query5 == 0:
                    return error.error_stock_level_low(book_id)
        elif status == 1 or status == 2:  # 用户付款，或卖家发货后取消订单
            # 更改订单状态
            db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id).update(
                {st.NewOrder.status: -1}
            )
```

```
            query6 = db.session.query(st.NewOrderDetail).filter(st.NewOrderDetail.order_id == order_id)
            total_price = 0
            for row4 in query6:  # 查看每一本书的情况，计算订单总金额
                count = row4.count
                price = row4.price
                total_price = total_price + price * count
            # 更改账户余额
            # 买家加钱
            query7 = db.session.query(st.User).filter(st.User.user_id == buyer_id).update(
                {st.User.balance: st.User.balance + total_price}, synchronize_session="evaluate")
            # 卖家账户扣钱
            query8 = db.session.query(st.User).filter(st.User.user_id == seller_id, st.User.balance>=tot
                {st.User.balance: st.User.balance - total_price}, synchronize_session="evaluate")
            if query8 == 0:
                return error.error_not_sufficient_funds(order_id)
        else:
            # 用户已付款或订单已经取消，无法取消订单
            return error.error_invalid_order_id(order_id)
        db.session.commit()
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明：

1. 从url中获取要取消的订单的相关信息(购买者的id:buyer_id，订单号:order_id)
2. 对信息进行检验(是否存在此用户，是否存在此订单)，若检验成功进行后续操作，若不成功返回响应的错误码
3. 通过order_id获取订单的详细信息(订单状态:status，卖家店铺id:store_id，卖家id:seller_id)
4. 对于不同状态的订单采取不同的操作
   1)用户下单(status=0)后取消
      更改订单状态为-1，更改卖家库存
   2)用户付款(status=1)卖家发货(status=2)后取消
      更改订单状态为-1，修改买家和卖家账户余额
   3)确认收货(status=3)已取消订单(status=-1)
      提示不可取消
5. 对于不同的情况，若操作成功，返回正确码200，若存在异常，返回异常码

6. 自动取消订单

代码实现

```python
def auto_cancel_order():
    # time = datetime.utcnow() + datetime.timedelta(minutes=5)
    create_timer()
    print('hello')
    now = datetime.datetime.utcnow()
    query = db.session.query(st.NotPayOrder).filter(st.NotPayOrder.ddl <= now).all()
    print(len(query))
    if len(query) > 0:
        for row in query:
            order_id = row.order_id
            print(order_id)
            q = db.session.query(st.NotPayOrder).filter(st.NotPayOrder == order_id).delete()
            # 更改这些订单的状态
            if q!=0:
                print("delete")
            query1 = db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id).update(
                {st.NewOrder.status: -1}
            )
        db.session.commit()
        #time.sleep(1)

def create_timer():
    t = threading.Timer(1,auto_cancel_order)
    t.start()
```

功能说明

开启一个线程，每隔一秒执行以下操作：
1. 获取当前时间now
2. 查看not_pay_order表中是否存在ddl<=now的记录，若存在就删除此条记录，并将此订单在new_order表中的状态改为-1

卖家功能

1. 添加书籍信息

代码实现

```python
# 添加书籍信息
def add_book(user_id: str, store_id: str, book_id: str, book_json_str: str, stock_level: int):
    try:
        if not db.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        if not db.store_id_exist(store_id):
            return error.error_non_exist_store_id(store_id)
        if db.book_id_exist(store_id, book_id):
            return error.error_exist_book_id(book_id)

        query = db.session.add(
            st.Store(store_id=store_id, book_id=book_id, book_info=book_json_str, stock_level=stock_level))
        db.session.commit()

    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

1. 从url中获取相关信息(卖家id: user_id, 店铺id:store_id, 书籍信息book_json_str, 书籍库存量:stock_level)
2. 对信息的正确性进行检验(user_id, store_id, book是否存在)，若信息检验正确进行后续操作，否则返回响应的错误代码
3. 在store表中插入此书籍信息，若操作成功返回正确码200，若失败，捕获异常并返回响应的异常码

2. 增加库存

代码实现

```python
# 增加库存
def add_stock_level(user_id: str, store_id: str, book_id: str, add_stock_level: int):
    try:
        if not db.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        if not db.store_id_exist(store_id):
            return error.error_non_exist_store_id(store_id)
        if not db.book_id_exist(store_id, book_id):
            return error.error_non_exist_book_id(book_id)

        db.session.query(st.Store).filter(st.Store.store_id == store_id, st.Store.book_id == book_id).updat
            {st.Store.stock_level: st.Store.stock_level + add_stock_level}, synchronize_session="evaluate"
        )
        db.session.commit()

    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

1. 从url中获取相关信息(要增加库存的卖家id:user_id, 要增加库存的店铺id:store_id, 书籍id:book_id, 要增加的数量: add_stock_level)
2. 对相关信息进行验证(卖家id, 店铺id, 书籍id是否存在)，若信息检验正确进行后续操作，若失败，返回相应的错误码
3. 更改store表中store_id店铺book_id书籍的库存量(stock_level = stock_level+add_value).若操作成功返回正确码200，若出现异常，捕获异常码并返回异常码

3. 创建商铺

代码实现

```python
# 创建商铺
def create_store(user_id: str, store_id: str) -> (int, str):
    try:
        if not db.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        if db.store_id_exist(store_id):
            return error.error_exist_store_id(store_id)

        db.session.add(st.UserStore(store_id=store_id, user_id=user_id))
        db.session.commit()

    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

1. 从url中获取相关信息(要创建店铺的用户id:user_id，要创建的店铺id:store_id)
2. 对相关信息进行检验(用户是否存在，要创建的店铺名是否存在)，若用户存在，且要创建的店铺不存在进行后续操作，否则返回相应的错误码
3. 在userstore表中新增(user_id，store_id)的数据，若操作成功返回正确码200，若失败，捕获异常并返回异常码

4. 发货

代码实现

```python
def receiving(user_id: str, order_id: str)-> (int, str):
    try:
        if not db.user_id_exist(user_id):
            return error.error_non_exist_user_id(user_id)
        # 订单是否存在
        query1 = db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id)
        row = query1.one_or_none()
        if row is None:
            return error.error_invalid_order_id(order_id)
        # 查看订单是否付款
        if row.status != 1:
            return error.error_not_pay(order_id)
        # 更改订单状态：已发货
        db.session.query(st.NewOrder).filter(st.NewOrder.order_id == order_id).update(
            {st.NewOrder.status: 2}
        )
        db.session.commit()
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
        return 530, "{}".format(str(e))
    return 200, "ok"
```

功能说明

1. 从url中获取相关信息(购买者id:user_id，订单号:order_id)
2. 对相关信息进行检验(购买者id，订单id是否存在,订单是否已取消，订单是否付款)，若user_id，order_id都没有问题且订单没有取消，且订单已经付款执行后续操作
3.将订单状态改为2，若操作成功，返回成功码200，若失败，捕获异常，并返回异常码

搜索图书

用户可以通过关键字进行图书搜索，搜索范围包括题目、作者、标签、内容，采用全站式搜索方式，对所有店铺的图书进行搜索。如果显示结果较大（超过10条信息），则会分页显示，每页10条信息。

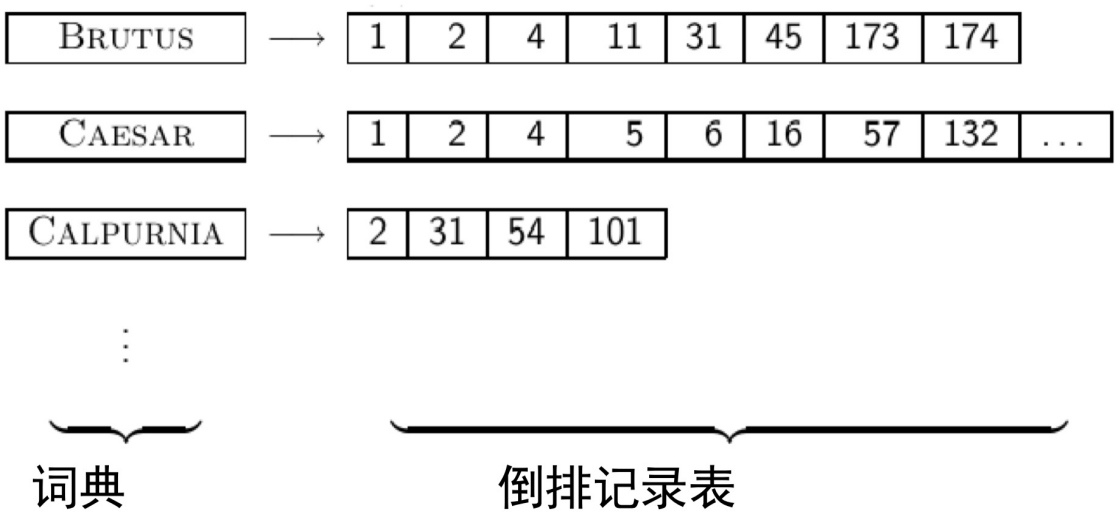
为了方便搜索，需要新建一个图书的表格，将book.db中的图书数据都存储到postgreSQL数据库中，其中包含了图书所有的详情信息。

由于存储图书具体数据的图书表格行列数目都非常多，搜索起来十分缓慢，为了加快搜索速度，采用倒排索引的方式，将查询单词与book_id进行一对一的映射，需要新增加题目、作者、标签、内容这4张表格，每个表格中都存放book_id和各自对应的题目或作者或标签或内容，这两个值合在一起成为表格的主键。并且，由于搜索都是基于题目、作者、标签、内容这4个值进行搜索的，故而在这4个值上设置索引，加快查询的进行。

1、按照题目搜索

在进行搜索之前，首先需要将基于标题的倒排索引存入到search_title表格中，考虑到很多人在搜索图书名称的时候都不会输入完整的图书名和特殊符号进行搜索，所以我先用正则表达式将图书名称中的一些特殊符号如"【"，"】"等进行去除，然后用结巴分词对题目进行精确模式的分词，然后将完整的书名、分词的结果和它们对应的book_id一并存到数据库中，词汇（含书名）和book_id共同组成表格的主键。每个book_id都只有唯一的书名和分词结果，但每个词汇不一定只有一个对应的book_id，所以倒排索引中词典是词汇的集合，倒排表是一系列book_id的链表。由于代码较多，不在此插入图片了，完整代码见inser_search_book.py文件。

倒排索引示例：



在search_title表格中插入数据之后，需要基于此表对用户输入的标题进行查找，如果没有找到该词汇，则直接返回没有找到；如果找到了该词汇，则将其对应的一个或多个book_id拿到store表格中进行搜索，查看是否有店铺含有此图书，若没有则直接返回没有找到，否则，将书籍的详情信息、商店id、存货返回给用户。

此时可能有很多书名都包含该词汇，以及很多书店都有该书籍的存货，返回的信息可能有很多条，无法在一个页面内放下所有信息，所以将按照顺序其进行分页操作，每个页面只放入10条信息，用户可以传入页码，后端将排好序的10条信息返回给用户。

值得注意的是，用户每次对页面进行换页，我的代码都必须对重复的进行上述的全局搜索，从全部的结果中按顺序选择用户需要的信息进行返回，而不能每次只查10条信息就返回给用户，其原因是数据库中每次查询结果的顺序都是不一致的，不能保证返回给用户的结果是与之前不重复的。所以当数据量比较大时，查询效率可能会低一些。

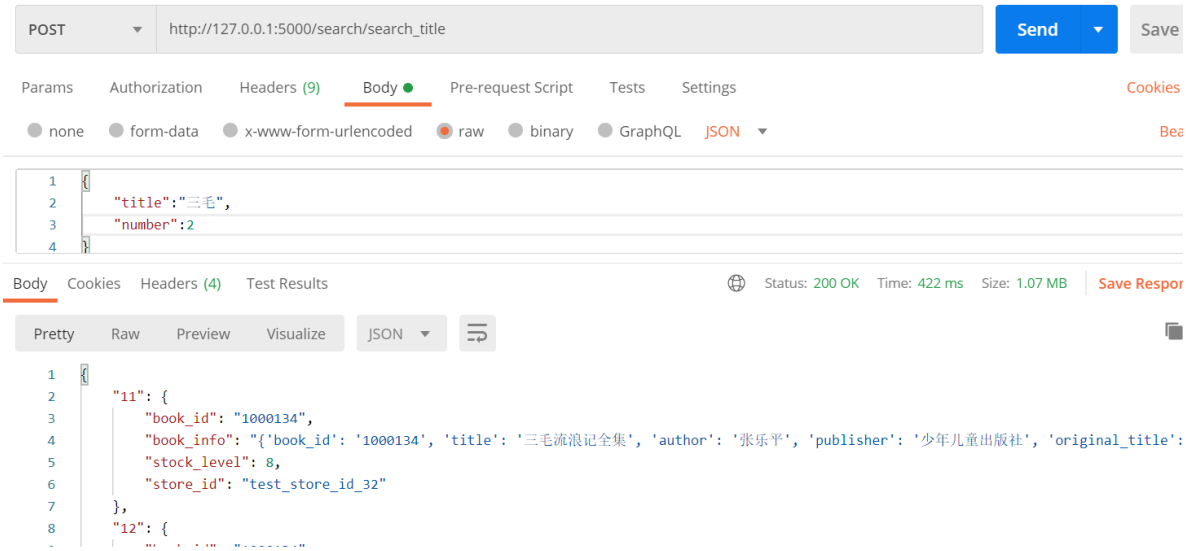
基于题目进行查询：

```
def search_title(title: str):
    try:
        search = []
        result = db.session.query(st.SearchTitle).filter(st.SearchTitle.title == title).all()
        if len(result) == 0:
            return error.error_no_title(title)
        else:
            for item in result:
                book_id = item.book_id
                result1 = db.session.query(st.Store).filter(st.Store.book_id == book_id).all()
                if len(result1) == 0:
                    continue
                else:
                    for item1 in result1:
                        store_id = item1.store_id
                        book_info = item1.book_info
                        stock_level = item1.stock_level
                        temp_list = [book_id, store_id, book_info, stock_level]
                        search.append(temp_list)
        if len(search) == 0:
            return error.error_no_title(title)
        return 200, str(search)
    except exc.SQLAlchemyError as e:
        return 528, "{}".format(str(e))
    except BaseException as e:
```

对查询结果进行分页：

```python
@bp_search.route("/search_title", methods=["POST"])
def search_title():
    title: str = request.json.get("title")
    number: int = request.json.get("number")
    code, message = search.search_title(title)
    if code != 200:
        return jsonify({"message": message}), code
    else:
        final_dict = {}
        result = eval(message)
        len_result = len(result)
        page_number = 10
        start = (number-1)*page_number
        if len_result <= start:
            return jsonify({"message": "this page have no information"}), code
        else:
            end = min(len_result, start+page_number)
            for index in range(start, end):
                temp_dict = {"book_id": result[index][0], "store_id": result[index][1],
                             "book_info": result[index][2], "stock_level": result[index][3]}
                final_dict[index+1] = temp_dict
            return jsonify(final_dict), code
```

搜索结果显示：

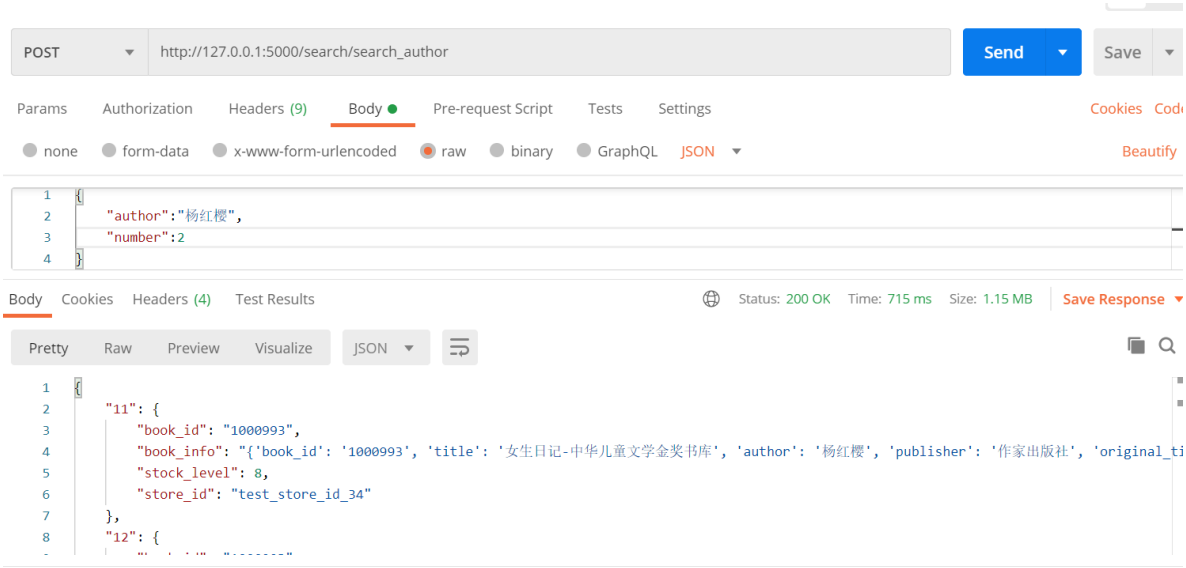


2、按照作者搜索

与基于题目进行搜索相似，首先要将作者的倒排索引存入到search_author表格中。基于生活经验，用户在搜索作者的时候一般会输入全名，所以对于作者的名字就暂且无需分词，但该数据库中作者的名字中也会包含形如“【美国】”这一类特殊的字符串，而用户又不会这样搜索，故而采用正则表达式将其去除，然后将作者的名字和对应的book_id存入到数据库中。倒排索引中词典是作者的集合，倒排表是一系列book_id的链表。

剩下的搜索和分页步骤与按照题目搜索类似，在此不详细阐述，也不列出图片了。

搜索结果展示：

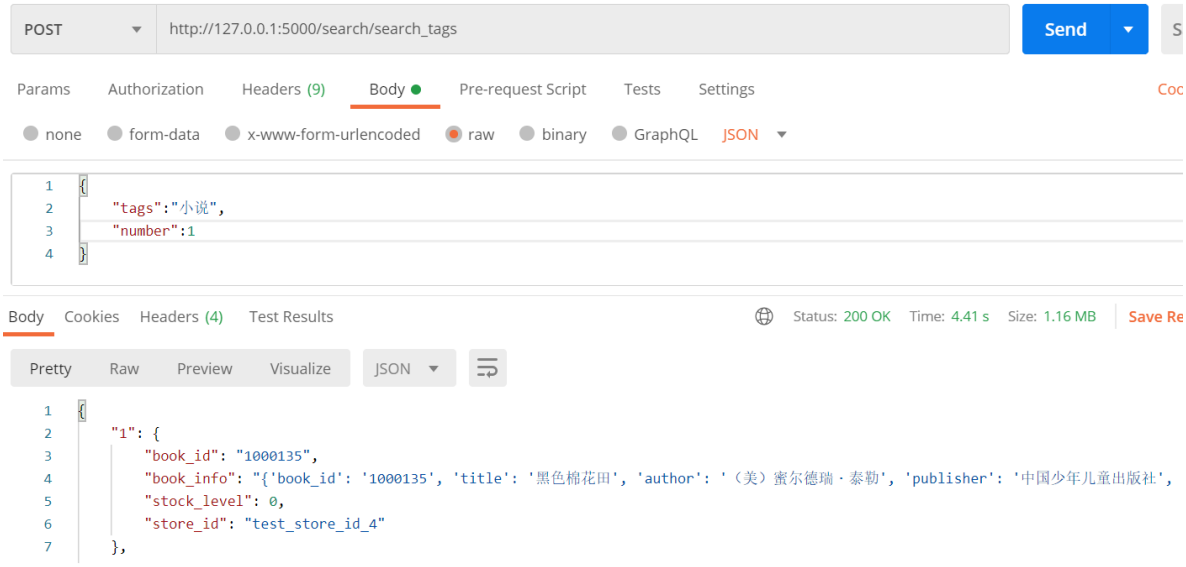


3、按照标签搜索

首先要将标签的倒排索引存入到search_tags表格中。由于标签本身就已经是很简洁的表达了，无需进行分词，但由于数据库中存储标签的格式的TEXT，所以需要将封存于字符串中的标签列表恢复，然后将标签和对应的book_id存入到search_tags表格中即可。

剩下的搜索和分页步骤与按照题目搜索类似，在此不详细阐述，也不列出图片了。

搜索结果展示：

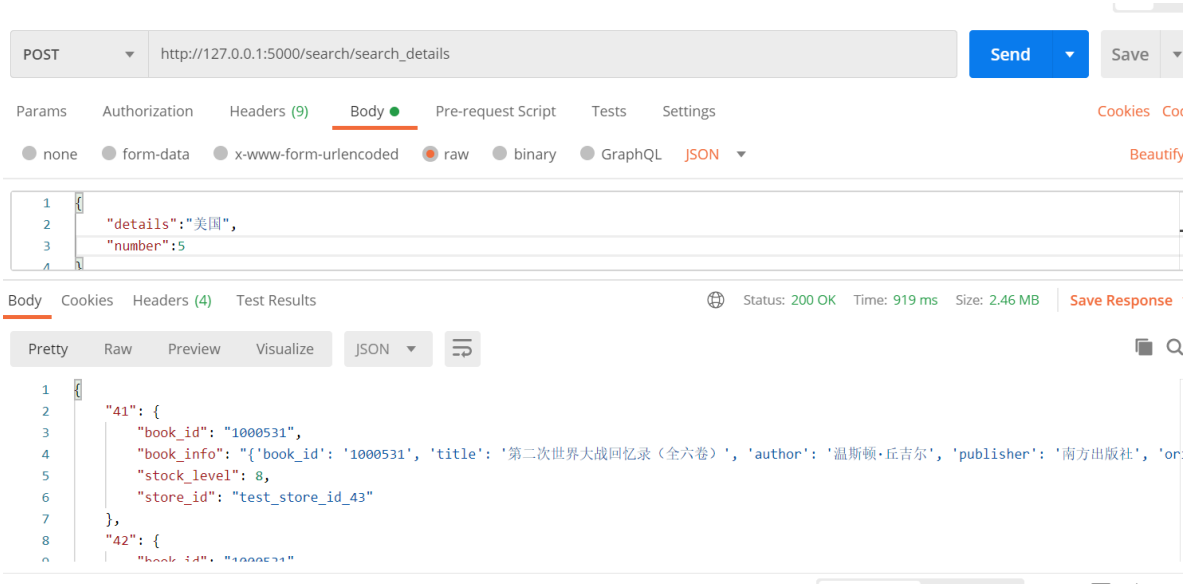


4、按照内容搜索

首先要将内容的倒排索引存入到search_details表格中。由于内容有非常多文字，无法基于所有内容进行搜索，需要先对其进行分词，然后取出有代表性的词汇放入search_details表格中。我使用了结巴分词提取内容中词频最高的5个词汇作为关键词，使用语句为：jieba.analyse.extract_tags(detail, topK=5)。然后将关键词和其对应的book_id存入到search_details表格中。

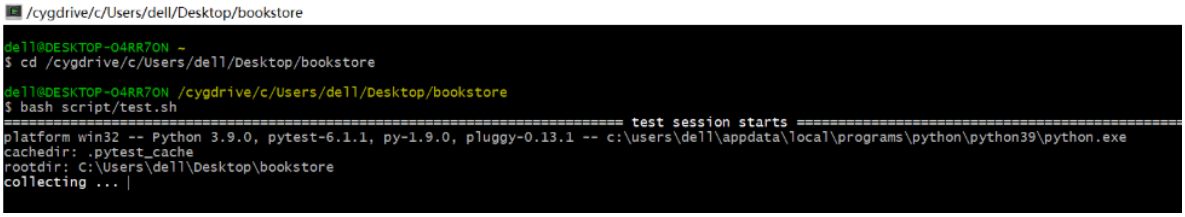
剩下的搜索和分页步骤与按照题目搜索类似，在此不详细阐述，也不列出图片了。

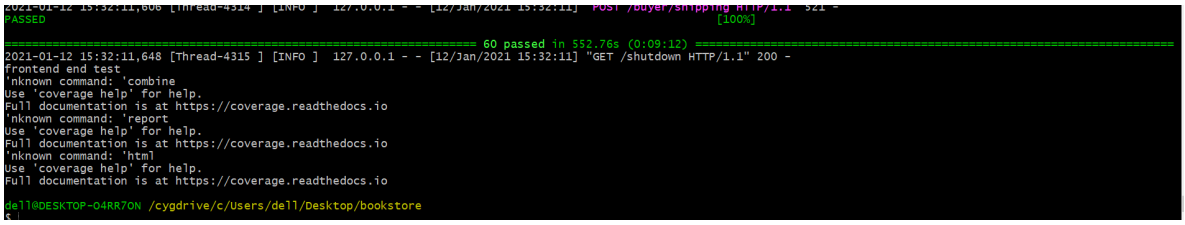
搜索结果展示：



五、代码测试

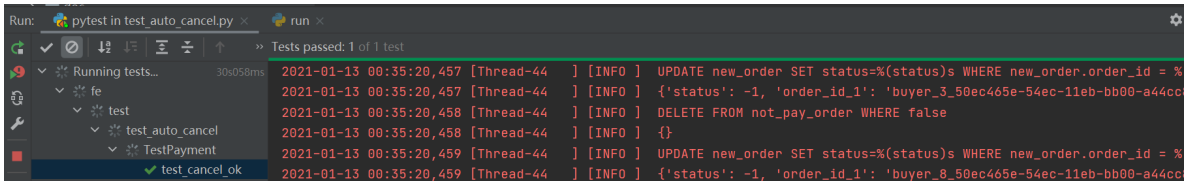
```
bash script/test.sh
```





测试结果为60pass，100%success

自动取消订单由于不能和前面的一起进行测试，所以拿出来单独进行测试，结果如下，通过。




```
coverage report
```

```
dell@DESKTOP-O4RR7ON /cygdrive/c/Users/dell/Desktop/bookstore
$ coverage report
Name                                    Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------------
be\__init__.py                              0      0      0      0   100%
be\app.py                                   8      8      2      0     0%
be\model\__init__.py                        0      0      0      0   100%
be\model\buyer.py                         209     56     88     14    70%
be\model\db_conn.py                        22      0      6      0   100%
be\model\error.py                          35      1      0      0    97%
be\model\search.py                         98     24     48      8    73%
be\model\seller.py                         62     17     30      1    72%
be\model\store.py                          82      0      0      0   100%
be\model\user.py                          119     26     40      6    75%
be\serve.py                                35      1      2      1    95%
be\view\__init__.py                         0      0      0      0   100%
be\view\auth.py                            38      0      0      0   100%
be\view\buyer.py                           48      0      2      0   100%
be\view\search.py                          81      4     24      4    92%
be\view\seller.py                          34      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                         54      0      2      0   100%
fe\access\new_buyer.py                      8      0      0      0   100%
fe\access\new_searcher.py                   8      0      0      0   100%
fe\access\new_seller.py                    11      0      0      0   100%
fe\access\search.py                        38      0      0      0   100%
fe\access\seller.py                        39      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                      126     24     22     13    74%
fe\conf.py                                 11      0      0      0   100%
fe\conftest.py                             17      0      0      0   100%
fe\test\gen_book_data.py                   48      0     16      0   100%
fe\test\test_add_book.py                   36      0     10      0   100%
fe\test\test_add_funds.py                  23      0      0      0   100%
fe\test\test_add_stock_level.py            39      0     10      0   100%
fe\test\test_auto_cancel.py                46      1      4      1    96%
fe\test\test_bench.py                       6      2      0      0    67%
fe\test\test_cancel_order.py               70      0      2      0   100%
fe\test\test_create_store.py               20      0      0      0   100%
fe\test\test_history.py                    58      0      8      1    98%
fe\test\test_login.py                      28      0      0      0   100%
```

```
fe\test\test_login.py                      28      0      0      0   100%
fe\test\test_new_order.py                  40      0      0      0   100%
fe\test\test_password.py                   33      0      0      0   100%
fe\test\test_payment.py                    61      1      4      1    97%
fe\test\test_receiving.py                  67      1      4      1    97%
fe\test\test_register.py                   31      0      0      0   100%
fe\test\test_search_author.py              19      0      0      0   100%
fe\test\test_search_details.py             19      0      0      0   100%
fe\test\test_search_tags.py                19      0      0      0   100%
fe\test\test_search_title.py               19      0      0      0   100%
fe\test\test_shipping.py                   60      1      4      1    97%
-------------------------------------------------------------------------
TOTAL                                    2086    168    358     55    89%
```

覆盖率为89%，是一个相对比较高的覆盖率

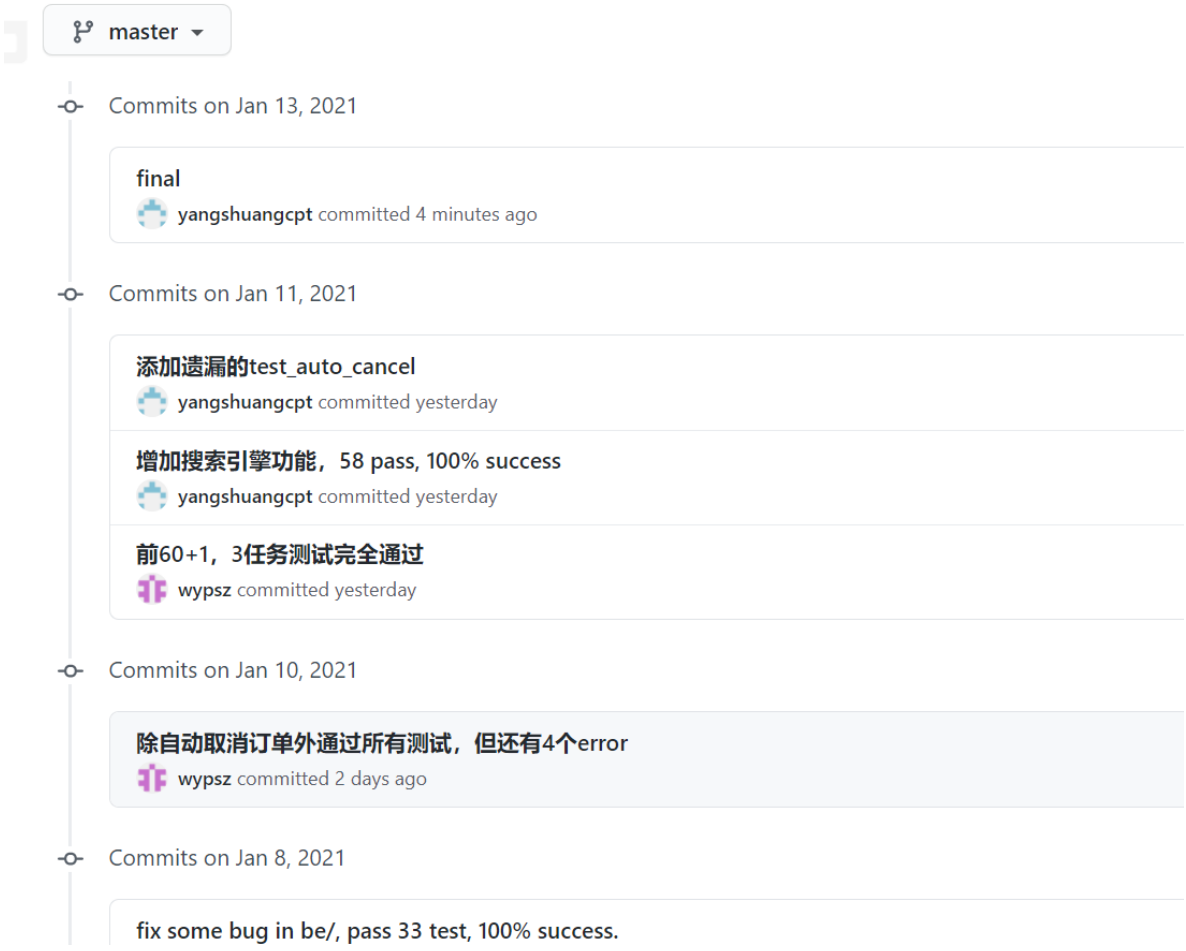
bench.run中测试订单吞吐量和延迟

```
LATENCY:0.01818191730044815
INFO:root:TPS_C=17801, NO=OK:483078 Thread_num:987 TOTAL:483078 LATENCY:0.03727982308945248 , P=OK:481838 Thread_num:986 TOTAL:482091
LATENCY:0.018182259671852878
INFO:root:TPS_C=17819, NO=OK:484066 Thread_num:988 TOTAL:484066 LATENCY:0.03728092413799351 , P=OK:482814 Thread_num:987 TOTAL:483078
LATENCY:0.018182596116244464
INFO:root:TPS_C=17837, NO=OK:485055 Thread_num:989 TOTAL:485055 LATENCY:0.037282063413932746 , P=OK:483791 Thread_num:988 TOTAL:484066
LATENCY:0.01818293494284334
INFO:root:TPS_C=17854, NO=OK:486045 Thread_num:990 TOTAL:486045 LATENCY:0.03728320775756406 , P=OK:484768 Thread_num:989 TOTAL:485055
LATENCY:0.018183267883700453
INFO:root:TPS_C=17872, NO=OK:487036 Thread_num:991 TOTAL:487036 LATENCY:0.037284336599570804 , P=OK:485745 Thread_num:990 TOTAL:486045
LATENCY:0.018183580575208438
INFO:root:TPS_C=17889, NO=OK:488028 Thread_num:992 TOTAL:488028 LATENCY:0.037285495222274846 , P=OK:486723 Thread_num:991 TOTAL:487036
LATENCY:0.018183895724844554
INFO:root:TPS_C=17907, NO=OK:489021 Thread_num:993 TOTAL:489021 LATENCY:0.03728660152432362 , P=OK:487701 Thread_num:992 TOTAL:488028
LATENCY:0.018184194872048996
INFO:root:TPS_C=17924, NO=OK:490015 Thread_num:994 TOTAL:490015 LATENCY:0.03728767420923647 , P=OK:488680 Thread_num:993 TOTAL:489021
LATENCY:0.018184486294311665
INFO:root:TPS_C=17942, NO=OK:491010 Thread_num:995 TOTAL:491010 LATENCY:0.03728871548778644 , P=OK:489660 Thread_num:994 TOTAL:490015
LATENCY:0.018184777409369566
INFO:root:TPS_C=17960, NO=OK:492006 Thread_num:996 TOTAL:492006 LATENCY:0.037289741801014194 , P=OK:490641 Thread_num:995 TOTAL:491010
LATENCY:0.01818505831423783
INFO:root:TPS_C=17977, NO=OK:493003 Thread_num:997 TOTAL:493003 LATENCY:0.0372907917761893 , P=OK:491623 Thread_num:996 TOTAL:492006
LATENCY:0.018185333897688715
INFO:root:TPS_C=17995, NO=OK:494001 Thread_num:998 TOTAL:494001 LATENCY:0.03729185918980081 , P=OK:492606 Thread_num:997 TOTAL:493003
LATENCY:0.018185616099081446
INFO:root:TPS_C=18012, NO=OK:495000 Thread_num:999 TOTAL:495000 LATENCY:0.03729291766436413 , P=OK:493589 Thread_num:998 TOTAL:494001
LATENCY:0.018185881610285424
INFO:root:TPS_C=18030, NO=OK:496000 Thread_num:1000 TOTAL:496000 LATENCY:0.03729401563925128 , P=OK:494573 Thread_num:999 TOTAL:495000
LATENCY:0.018186141633024l2
```

吞吐量为18030，延迟为0.0372

六、版本控制与分工

版本控制：基于github上传代码，部分截图如下



分工：

王一平：实现接口的基础功能（大部分）、完成收发货、订单查询与取消订单、自动取消订单

杨爽：完善并修改接口的基础功能，测试出基础功能的test，实现图书题目、作者、标签和内容的搜索功能

七、项目优缺点与总结

通过本次项目，我们更加深入的了解到了电商平台bookstore系统后端的实现方式，学习了如何利用多线程实现自动取消订单的功能，并且学会利用倒排索引和分词等方法来构建搜索引擎。而且由于本项目是小组作业，我们也学会了如何分工与协商，并学习了如何利用github完成版本控制。

与此同时我们在实践的过程中也发现了一些自己代码的不足之处，在实现搜索图书功能时，虽然完成了分页查询，但是用户每次翻页都会找出全部的结果，从中截取用户需要的部分返回，并不能实现找到所需的一页信息就迅速返回的情况，相比之下效率会偏低，但是能保证输出的结果都是按序的。其次，由于book_info在数据库中是按照TEXT的格式进行存储的，不是很好恢复为正常格式，所以在前端显示的时候，有些信息会出现一部分特殊字符，这一点也需要继续完善。以及在实现自动取消订单功能时，每隔一秒钟开启一个线程去表中查找有没有过期的订单，如果有就执行删除操作。这样的消耗是很大的，时间误差较大，并且对数据库有大量的读写操作，效率也很低。可以尝试使用redis数据库中的定时功能，开启一个线程监听，捕获过期订单信息。这样可以减少对数据库读写操作，时间误差也会小很多，执行效率也能有所提高。