华东师范大学数据科学与工程学院实验报告 - 作业2：书店

课程名称：当代数据管理系统

上机实践名称：书店

指导教师：周烜

姓名：李云帆 @Yunfan Li 、周子彦 @Ziyan Zhou 、周家纬 @JiaWei_Zhou

学号：10195501416、10181511117、10195501437

年级：2019

上机实践日期：2021/12/5 - 2021/12/19

Content

Content
1 关系数据库设计
1.1 数据库整体设计思路
1.1.1 用户
1.1.2 订单和订单历史
1.1.3 书店的设计
1.1.4 书籍的存放方式
1.2 数据库其它重要设计
1.2.1 字段约束
1.2.2 索引
1.2.3 冗余
1.3 ER图设计
1.4 数据库构建
2 架构与模块设计
2.1 设计原则
2.2 设计过程中的思考
2.2.1 数据访问层和业务逻辑层
2.2.2 数据库连接方法
2.2.3 事务处理
3 功能设计与业务逻辑

3.1 auth 用户权限接口
3.1.1 **注册用户**
3.1.2 **注销用户**
3.1.3 **用户更改密码**
3.1.4 **用户登出**
3.1.5 **实现token定期失效功能**
3.2 buyer 买家用户接口
3.2.1 **买家下单**
3.2.2 **买家付款**
3.2.3 **买家充值**
3.2.4 **买家取消订单**
3.2.5 **买家确认收货**
3.2.6 **买家通过关键词获取图书列表**
3.2.7 买家评价已完成订单
3.2.8 **买家查看店铺评价**
3.2.9 买家查看订单状态
3.2.10 **买家查看所有进行中的订单**
3.2.11 **买家查看所有交易完成订单**
3.2.12 **实现订单定期失效功能**
3.3 seller 卖家用户接口
3.3.1 创建商铺
3.3.2 商家添加书籍信息
3.3.3 商家添加书籍库存
3.3.4 卖家确认发货
4 测试
4.1 **正确性测试**
4.2 代码覆盖率
4.3 **并发场景下吞吐量测试**
4.4 **对原测试用例的疑问和改进**
4.4.1 book_id重复
4.4.2 负载生成过程中store_id与book_id不匹配
4.4.3 吞吐率测试和计算方式错误
5 开发过程与分工
5.1 开发过程
5.2 分工

1 关系数据库设计

1.1 数据库整体设计思路

1.1.1 用户

1. 根据测试用例，我们不拆分`user`表。每个人既可以成为买家也可以成为卖家。
2. 因为我们没有外接的支付方式，因此我们需要在`user`下额外加入其拥有的cash。
3. 由于没有给定用户名字，这里没有加入用户name字段，但我们在测试后发现实际上user_id就是user_name，在对主键的讨论中我们会谈到这个问题的解决方案。
4. 卖家拥有其自己的store，且一个人可以拥有多个store。
5. 卖家一般需要查看自己的流水，因此cash字段也可以成为卖家的流水情况。

1.1.2 订单和订单历史

1. 我们将正在交易的订单和订单历史分开存。理由：考虑一种情况：正在交易的订单的数量是极小的（相较于订单历史来说）如果一直放在此表中容易导致此表过大。
2. 由于有订单历史，因此我们可以针对已完成的订单额外添加评论和打分功能。
3. 通过表的分离，不仅能够提高事务的处理效率，还能够提升查询效率。
4. 通过加入时间戳后可以使用业务逻辑方便地进行订单自动撤销的操作。

1.1.3 书店的设计

1. 这里书店表只记录书店id和其对应的卖家id。
2. 由于没有给定书店名字，这里没有加入书店name字段。
3. 书店和书籍的关系为m:n。其中，因为有stock的要求，因此额外加入了count作为联系的属性。

1.1.4 书籍的存放方式

1. 目前所有的数据表都放在了mysql上，没有使用非关系型数据库。
    a. 首先，blob和text占的空间不是很大，没有必要分开存储。
    b. 其次，进行信息检索时使用关系型数据库会较为方便。
    c. 更好的实现事务功能。
2. 书分成了三张表：主表、tag表和pic表。因为tag和pic是多值属性。

1.2 数据库其它重要设计

1.2.1 字段约束

1. 主键
    a. 主键是表的唯一标识，且sql数据库会在主键上自动建立索引。
    b. 【对用户ID的讨论】根据文档可知，前后端交互使用的是"用户名"作为用户标识（尽管文档中的变量名为'user_id'，但其实际上是用户注册时所给出的"用户名"字符串）。通常来说，主键采用自增的整数型数据效率比字符串高很多，但此处为了避免进一步混淆"用户名"和id的概念，认为直接将"用户名"字符串作为主键比较合理。另一方面，由于交互过程中使用"用户名"作为标识，即使引入了id，也无法避免通过"用户名"检索对应id的步骤，这一步本身就需要较高的代价，可能反而使得效率降低。可参考 https://blog.csdn.net/slm2000/article/details/262547。
2. 外键
    a. 外键的设计可以很好的引导开发，避免引发数据不一致的错误。
    b. 但检查外键正确性本身就是一个耗费时间和资源的过程，它可能在高并发下影响性能。因此，如果在一些对正确性要求不是非常高的场景下，能够能用业务逻辑保证正确性的操作就不必通过外键进行约束。
3. 非空约束
    a. 虽然我们可以通过业务逻辑去解决取值为空的问题。但非空约束的加入能够带来一个最底层的保险。
    b. 非空约束还会带来对查询速度的提升。

1.2.2 索引

1. 由于我们的项目讲数据访问和业务逻辑分离了，因此我们可以很方便的观察到哪些取值是频繁访问的。具体的index建立如下：
    a. `book`的title，author
    b. `tag_for_book`的book_tag
    c. `store`的seller_id
    d. `book_for_store`的store_id，book_id
    e. `orderform`和`orderform_history`的buyer_id，store_id

2. 索引不是万能的，因为对数据的修改会导致对索引的修改。因此我们并不倾向立即使用索引来增加业务处理的效率。
3. 在对中文文章进行索引时，似乎建立全文索引是个挺好的想法，速度比全比 like + % 快非常多，但这也是需要做权衡的。主要是权衡读者全文搜索的使用率和建立全文索引对整体项目性能的影响。如果大部分读者只看不买，那么建立索引是相当划算的。但如果仅有少量读者通过此接口检索数据，那么这种建立索引就显得没有必要了。由于吞吐量测试也会生成对book表的负载，因此我们这里保守起见在测试时去除了这个全文索引。

1.2.3 冗余

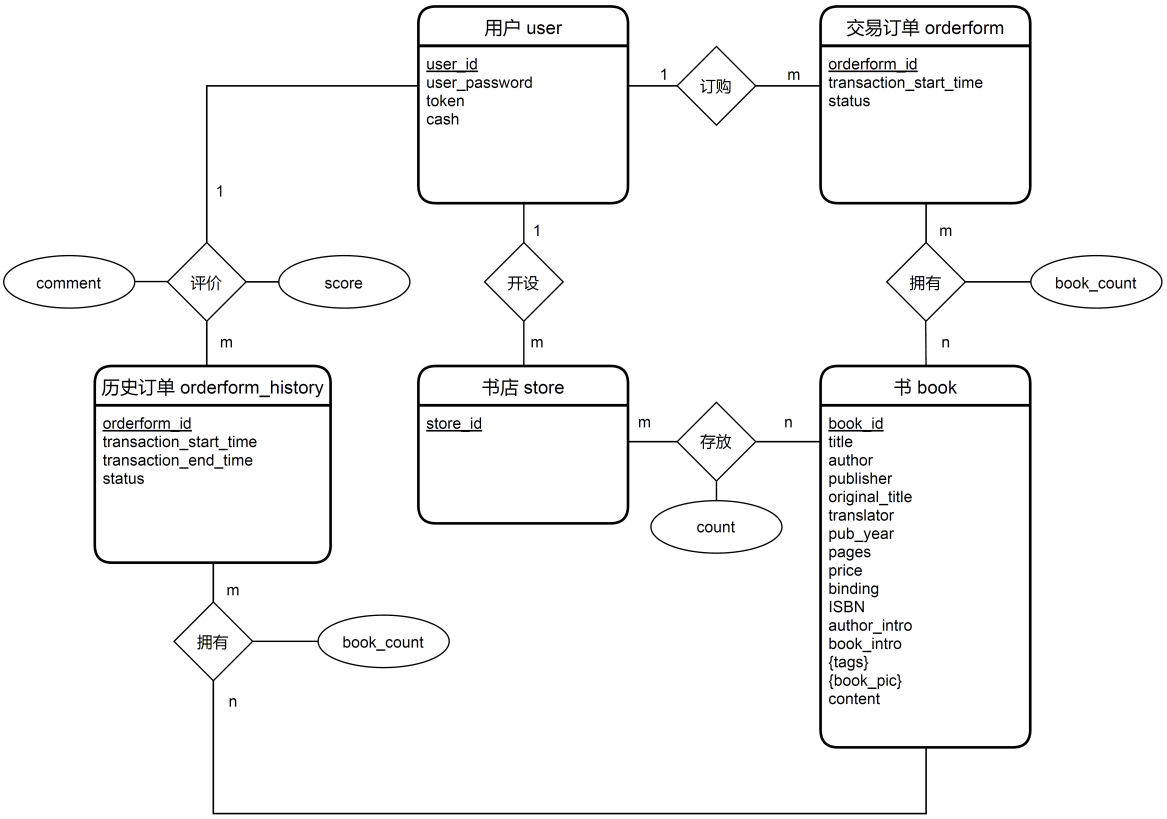
由于对订单orderform的总金额查询是经常需要被用到的属性，并且单次查询代价很大，需要跨多表进行查询、汇总、和计算，因此我们在订单总金额上设计了冗余。

（如果不作为冗余，可以通过book_for_orderform中的book_for_orderform_id、book_count以及书的price算出来）

对于其功能实现进行了细致的规划，保证了冗余不会造成正确性问题。

同理，我们在orderform_history的总金额一项也设置了冗余。

1.3 ER图设计



1.4 数据库构建

【若无特别说明，默认设置第一行属性为PRIMARY KEY】

`user`

记录user的单值属性

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
user_id	VARCHAR(255)	Y	N	Y
user_password	VARCHAR(255)	N	N	Y
cash	INT	N	N	Y
token	VARCHAR(255)	N	N	Y

`orderform`

记录还在执行中（已下单（未付款）/已付款（未发货）/已发货（未收货）状态）的订单

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
orderform_id	BIGINT	Y	N	Y
buyer_id	VARCHAR(255)	N	user(user_id)	Y
store_id	VARCHAR(255)	N	store(store_id)	Y
purchased_price	INT	N	N	Y
transaction_start_time	TIMESTAMP	N	N	Y
transaction_end_time	TIMESTAMP	N	N	Y
status	INT	N	N	Y

status状态码　1：待付款，2：待发货，3：待收货

`book_for_orderform`

记录还在执行中的订单的书籍

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
orderform_id	BIGINT	N	orderform(orderform_id)	Y
book_for_orderform_id	VARCHAR(255)	N	book(book_id)	Y
book_count	INT	N	N	Y

`orderform_history`

记录已完成（待评价/已评价状态）的历史订单

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
orderform_id	BIGINT	Y	N	Y
buyer_id	VARCHAR(255)	N	user(user_id)	Y
store_id	VARCHAR(255)	N	store(store_id)	Y
purchased_price	INT	N	N	Y
transaction_start_time	TIMESTAMP	N	N	Y
transaction_end_time	TIMESTAMP	N	N	Y
status	INT	N	N	Y

status状态码　4：待评价，5：已评价

`book_for_orderform_history`

记录已完成的订单的书籍

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
orderform_id	BIGINT	N	orderform(orderform_id)	Y
book_for_orderform_id	VARCHAR(255)	N	book(book_id)	Y
book_count	INT	N	N	Y

`book`

记录book的单值属性

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
book_id	VARCHAR(255)	Y	N	Y
title	VARCHAR(255)	N	N	N
author	VARCHAR(255)	N	N	N
publisher	VARCHAR(255)	N	N	N
original_title	VARCHAR(255)	N	N	N
translator	VARCHAR(255)	N	N	N
pub_year	YEAR	N	N	N
pages	INT	N	N	N
price	INT	N	N	N
biding	VARCHAR(255)	N	N	N
ISBN	VARCHAR(20)	N	N	N
author_intro	TEXT	N	N	N
book_intro	TEXT	N	N	N
content	TEXT	N	N	N

`store`

记录store的单值属性

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
store_id	VARCHAR(255)	Y	N	Y
seller_id	VARCHAR(255)	N	user(user_id)	Y

`book_for_store`

记录store和book的拥有关系

设置联合主键或由系统生成默认主键

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
store_id	VARCHAR(255)	N	store(store_id)	Y
book_id	VARCHAR(255)	N	book(book_id)	
book_count	INT	N	N	Y

`tag_for_book`

记录book的多值属性tag

设置联合主键或由系统生成默认主键

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
book_id	VARCHAR(255)	N	book(book_id)	Y
book_tag	VARCHAR(255)	N	N	Y

`pic_for_book`

记录book的多值属性pic

设置联合主键或由系统生成默认主键

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
book_id	VARCHAR(255)	N	book(book_id)	Y
book_pic	LONGBLOB	N	N	Y

`comment_for_store`

记录用户对历史记录的评价

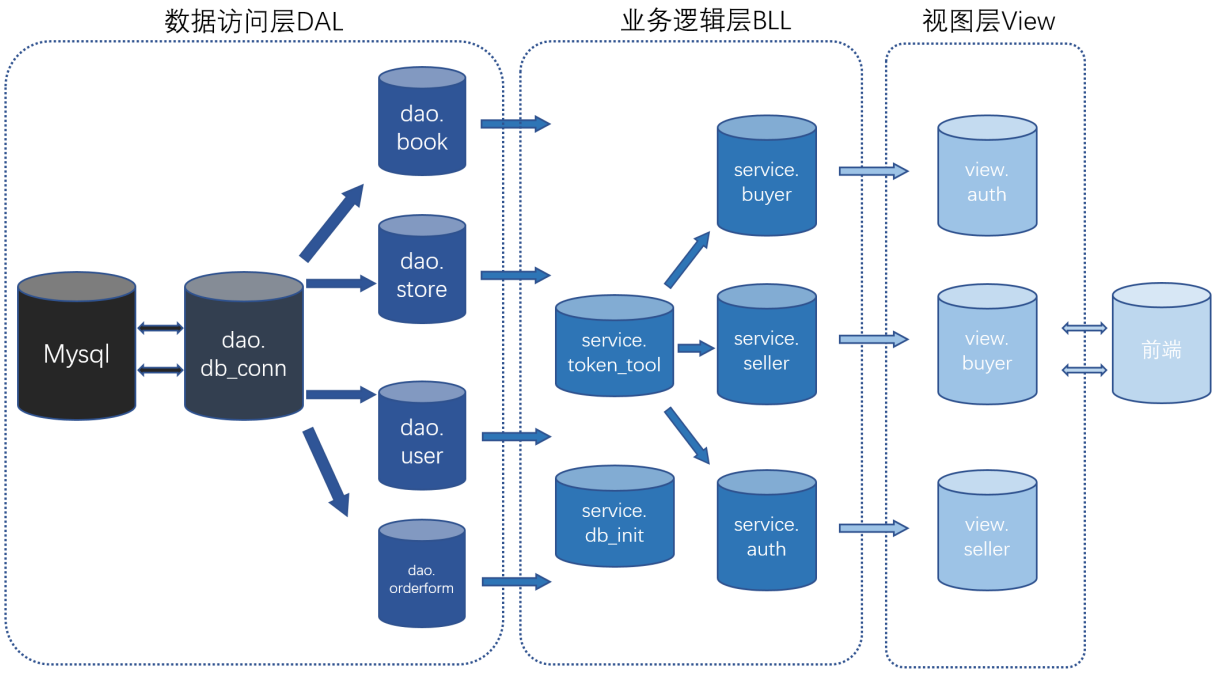
设置联合主键或由系统生成默认主键

NAME	TYPE	UNIQUE	FORENIGN KEY	NOT NULL
store_id	VARCHAR(255)	N	store(store_id)	Y
user_id	VARCHAR(255)	N	user(user_id)	Y
order_id	BIGINT	N	orderform_history(orderform_id)	Y
score	INT	N	N	Y
comment	TEXT	N	N	N

2 架构与模块设计

项目整体上遵循“三层架构”，将后端分为三层：数据访问层DAL、业务逻辑层BLL和视图层View。

- 数据访问层DAL负责对直接对数据库进行CRUD操作。
- 业务逻辑层BLL负责处理具体的业务逻辑。
- 视图层View负责前后端之间的交互，解析请求参数，并返回前端需要的数据和状态码。



2.1 设计原则

1. 严格分离数据访问层和业务逻辑层，减少模块耦合度。DAL层只负责数据存储和读取，业务逻辑只在BLL层体现。这要求DAL层模块中接口设计尽可能简单。
2. 尽可能避免接口跨层调用，每一个模块应只通过它上一层模块的接口实现，即模块的实现应当是面向上一层接口的。
3. 尽可能设计通用的模块接口，提高接口复用率，降低实现和维护成本。

2.2 设计过程中的思考

在设计架构与模块的过程中，我们对以下问题进行了重点考量，权衡各种设计的利弊，并选择尽可能合理的设计。

2.2.1 数据访问层和业务逻辑层

> 是否分离数据访问层和业务逻辑层？

在给出的后端参考代码中并没有严格分离数据访问层与业务逻辑层（业务逻辑层中包含数据库连接和sql语句，直接对数据库进行访问），我们认为这并不是很合理，原因如下：

该项目主要使用MySQL作为数据库系统，但考虑到同时存在了一部分blob数据（如书籍content和picture），该部分数据可能需要使用非结构化数据库存储来达到更好的效率，即涉及的数据库可能不止一种。为了提高开发效率，我们决定先仅使用MySQL数据库，后续再考虑增加其他数据库。

考虑到这种后续开发与拓展的可能性，如果不对数据访问层和业务逻辑层进行分离，那么在增加数据库时就需要将业务逻辑层和数据访问层一起做修改。而这种高耦合的设计可能会导致在修改数据访问的过程中，原本正确的业务逻辑被修改而无法正确实现，非常大地影响了项目的维护成本与可扩展性。

另外，即使不需要增加或更换数据库，一个低耦合的设计本身也更有利于接口设计、项目分工、代码复用，进而提高项目的可扩展性和开发效率。

因此，我们认为严格分离数据访问层和业务逻辑层是有必要的，即DAL层只负责数据存储和读取，业务逻辑只在BLL层体现。

2.2.2 数据库连接方法

> 如何选择数据库连接方法？

为了使项目具有处理并发请求的能力，我们设计了以下三种数据库连接的方法：

1. 所有并发请求共享一个连接，通过加锁的方式避免并发场景下的竞争条件。
2. 每个并发请求单独发起一个连接，并在请求结束后关闭连接。
3. 使用连接池管理连接，并发请求从连接池中获取连接，在请求结束后不关闭连接，而是将连接放回连接池。

第一种方法没有利用多线程处理数据库请求，需要频繁加锁和释放锁，效率较低。第二种方法过于频繁地建立和关闭连接，代价较大。

第三种方法既能利用数据库多线程处理请求的能力，又避免了过于频繁的建立和关闭连接，效率较高，也是我们选择的方法。实现方面，使用了DBUtils和pymysql建立MySQL数据库连接池。

2.2.3 事务处理

> 如何实现事务处理？

由于项目分离了数据访问层和业务逻辑层，数据访问层的每个数据访问对象对应一张表的CRUD操作，分别通过不同的连接完成。这虽然降低了耦合度，提高开发效率，但对业务逻辑层的事务处理实现造成了一定的困难。事务处理要求事务的所有操作处在同一个连接中，当service层同时操作两张或更多张表时，这些CRUD操作将在不同的连接中完成，导致这些操作无法成为一个事务，即这种架构设计使得项目无法将service层的操作作为事务来保证原子性。

为了解决该问题，可以对DAO再进行分层，每个底层DAO(BDAO)仍然只负责一张表，顶层DAO(TDAO)使用底层DAO暴露的方法，保证所有对表的操作在一个连接里实现，实现事务处理的能力。但该方法的缺点是需要实现业务逻辑中每个事务的TDAO，工作量相较于原方法明显大。

以上分析了事务处理能力的实现方法，但考虑到工作量和开发效率的问题，目前并没有实现逻辑层的事务处理。后续可以根据实际业务场景，根据实际的并发量大小和因并发导致的错误情况决定是否对DAO进一步分层并实现事务处理。

3 功能设计与业务逻辑

我们实现了以下**基本功能和附加功能**，其中，附加功能包括不限于：

- 实现发货 -> 收货功能
- 搜索图书：给定搜索范围，进行全局搜索
- 订单状态，订单查询和取消定单
    - 订单状态：查询符合特定订单状态的订单信息
    - 实现了买家主动地取消定单，和订单超时自动取消
- **店铺评价**：
    - 买家在订单结束后给店铺评价：评论和打分

- 用户给定用户店铺ID，返回用户给店铺的评价

在接口实现中，我们基于原有实验要求文档进行了合理的扩展，我们还是尽量按照实验要求完成，但由于有些实验要求中存在指定 Status Code 不够合理的问题（例如对于401的错误范围定义不明确）。

在未来的工作中，我们希望能更确切地指定 Status Code 所对应的错误信息，通过预先 config 设置，并在返回错误消息时直接调用。

3.1 auth 用户权限接口

3.1.1 注册用户

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
注册用户	POST http://$address$/auth/register	user_id, password	200:注册成功 508:注册失败，用户名重复 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 User().get_user_by_id(user_id) 返回值验证用户名未重复
3. 通过 User().add_user(user_id, password) 在User表中添加新用户信息

3.1.2 注销用户

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
注册用户	POST http://$address$/auth/register	user_id, password	200:注册成功 508:注册失败，用户名重复 518:表单缺值	message

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
注销用户	POST http://$address$/auth/unregister	user_id, password	200:注销成功 401:注销失败，用户名不存在或密码不正确 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 check_user_valid() 验证用户存在、用户密码正确
3. 通过 token_tool.encode(user_id, terminal) 生成 token
4. 通过 User().update_user_token(user_id, token) 在User表中变更用户 token 信息

3.1.3 用户更改密码

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
用户更改密码	POST http://$address$/auth/login	user_id, oldPassword, newPassword	200:更改密码成功 401:更改密码失败，用户名不存在或原先的密码不正确 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 check_user_valid() 验证用户存在、用户原先的密码正确
3. 通过 User().update_user_password(user_id, newPassword) 在User表中变更用户密码

3.1.4 用户登出

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
用户登出	POST http://$address$/auth/logout	user_id, token(Header中传入)	200:登录成功 401:登录失败，用户名不存在或token不正确 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 auth.get_user(user_id) 验证用户存在
3. 通过 token_tool.check_token_validation(token, user_id) 验证token是user_id的token，过时清除token
4. 通过 User().update_user_token(user_id, user_token="") 在User表中变更用户 token 信息

3.1.5 实现token定期失效功能

这里我们使用了 `token_tool.check_token_validation(token, user_id)` 这个函数，它会对 user_id 的 token 进行检查，如果 token 解码所得的 timestamp 跟现在的 timestamp 时间大于一个小时，那么就会通过 User().update_user_token(user_id, user_token="") 在User表中变更用户 token 信息。

3.2 buyer 买家用户接口

3.2.1 买家下单

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家下单	POST http://$address$/buyer/new_order	user_id, store_id, books列表包含id和count, token(Header中传入)	200:下单成功 509:买家用户ID不存在 401:token不正确 503:商铺ID不存在 510:购买的图书不存在 511:商品库存不足 518:表单缺值	order_id

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 User().get_user_by_id(user_id) 返回值验证用户名是否存在
3. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token
4. 通过 Store().check_book_store_exists(store_id, book_id) 验证图书是否存在于商铺
5. 通过 Store().get_stock_level(store_id, book_id) 验证商品库存是否充足
6. 通过 Orderform().add_book_for_orderform(order_id, books) 修改book_for_orderform表，将订单中的图书添加到book_for_orderform表中
7. 通过 Store().add_stock_level(store_id, book_id, -book_count) 修改book_for_store表，修改book_count属性，减少对应书在商铺中的库存
8. 通过 Book().get_book_price(book_id) * book_count 计算订单总价
9. 通过 Orderform().update_orderform_price(order_id, purchased_price) 更新订单总价

3.2.2 买家付款

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家付款	POST http://$address$/buyer/payment	user_id, order_id, password	200:付款成功 512:账户余额不足 513:订单ID不存在，或订单和买家关系不正确 401:用户名不存在或密码不正确 514:订单已被支付 522:订单超时 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 auth.check_user_valid(buyer_id, password) 验证用户存在、用户密码正确
3. 通过 Orderform().check_orderform_valid(buyer_id, order_id) 判断订单ID是否正确，订单和买家关系是否正确
4. 通过 Orderform().get_orderform_status(order_id) 检查订单状态
5. 通过记录的 transaction_start_time.timestamp() 判断**订单是否超时**
6. （如果超时）从 order_id 查找到对于的 store_id 和 books 信息，通过 Store().add_stock_level(store_id, book_id, book_count) 修改book_for_store表，修改book_count属性，重新增加对应书在商铺中的库存
7. （如果超时）通过 Orderform().delete_invalid_orderform(order_id) 删除订单
8. 分别获取账户余额和订单总金额，判断账户余额是否充足
9. 通过 Orderform().update_orderform_status(order_id, 2) 更新订单状态
10. 通过 User().add_user_cash(buyer_id, -order_price) 扣除订单总金额

3.2.3 买家充值

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家充值	POST http://$address$/buyer/add_funds	user_id, password, add_value	200:充值成功 401:用户名不存在或密码不正确 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 auth.check_user_valid(buyer_id, password) 验证用户存在、用户密码正确
3. 通过 User().add_user_cash(buyer_id, -order_price) 增加金额

3.2.4 买家取消订单

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家取消订单	POST http://$address$/buyer/cancel	user_id, order_id, token(Header中传入)	200:取消订单成功 509:买家用户ID不存在 401:token不正确 513:订单ID不存在，或订单和买家关系不正确 514:订单状态不正确 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 User().get_user_by_id(user_id) 返回值验证用户名是否存在
3. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token
4. 通过 Orderform().check_orderform_valid(buyer_id, order_id) 判断订单ID是否正确，订单和买家关系是否正确
5. 通过 Orderform().get_orderform_status(order_id) 检查订单状态
6. 从 order_id 查找到对于的 store_id 和 books 信息，通过 Store().add_stock_level(store_id, book_id, book_count) 修改book_for_store表，修改book_count属性，重新增加对应书在商铺中的库存
7. 通过 Orderform().delete_invalid_orderform(order_id) 删除订单

3.2.5 买家确认收货

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家确认收货	POST http://$address$/buyer/delivery_confirmed	buyer_id, order_id	513:订单ID不存在，或订单和买家关系不正确 524:订单状态不正确 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 Orderform().check_orderform_valid(buyer_id, order_id) 判断订单ID是否正确，订单和买家关系是否正确
3. 通过 Orderform().get_orderform_status(order_id) 检查订单状态
4. 通过 Orderform().orderfrom_into_history(order_id, now) 将订单和订单包含的书籍都移入 orderform_history 和 book_for_orderform_history 表中，记录当前时间，从 orderform 和 book_for_orderform 中删除信息

3.2.6 买家通过关键词获取图书列表

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家通过关键词获取图书列表	POST http://$address$/buyer/search_book	keywords, params(全局搜索设置范围)	200:查询成功 520:搜索输入为空 521:范围参数非法，需要至少指定一个有效的查询范围 518:表单缺值	message, book_list

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 判断 keywords 是否合法
3. 判断 params 是否合法

4. 通过 Book().search_book_by_keywords(keywords, is_title, is_content, is_author, is_tag) 进行全局搜索

3.2.7 买家评价已完成订单

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家评价已完成订单	POST http://$address$/buyer/comment	user_id, order_id, score, comment, token(Header中传入)	200:评价成功 401:用户名不存在或密码不正确 503:商铺ID不存在 513:订单ID不存在，或订单和买家关系不正确 524:订单状态不正确 525:评价分数不合法 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 User().get_user_by_id(user_id) 返回值验证用户名是否存在
3. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token
4. 通过 Orderform().get_orderform_status(order_id) 检查订单状态
5. 判断 score 是否合法
6. 通过 Orderform().update_orderform_history_status(order_id, 5) 修改订单状态为已评价
7. 通过 Store().add_comment_for_store(store_id, buyer_id, order_id, score, comment) 增加评价

3.2.8 买家查看店铺评价

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家查看店铺评价	POST http://$address$/buyer/search_comment	store_id	200:查询成功 503:商铺ID不存在 518:表单缺值	message, comment_list

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 Store().check_store_exists(store_id) 判断商家是否存在
3. 通过 Store().get_comment(store_id) 查找全部评价

3.2.9 买家查看订单状态

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家查看订单状态	POST http://$address$/buyer/check_order_status	user_id, order_id, token(Header中传入)	200:查询成功 513:订单ID不存在，或订单和买家关系不正确 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 User().get_user_by_id(user_id) 返回值验证用户名是否存在
3. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token
4. 通过 Orderform().get_orderform_status(order_id) 和 Orderform().check_orderform_history_valid(buyer_id, order_id) 获得订单状态

3.2.10 买家查看所有进行中的订单

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家查看所有进行中的订单	POST http://$address$/buyer/check_order_history_ongoing	user_id, token(Header中传入)	200:查询成功 518:表单缺值	message, history（包含订单号 订单状态 订单总价 订单开始时间和书籍列表）

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 User().get_user_by_id(user_id) 返回值验证用户名是否存在
3. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token， 过时清除token
4. 通过 Orderform().check_order_history_ongoing(buyer_id) 看买家进行中的所有订单

3.2.11 买家查看所有交易完成订单

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
买家查看所有交易完成订单	POST http://$address$/buyer/check_order_history	user_id, token(Header中传入)	200:查询成功 518:表单缺值	message, history_order（包含订单号 订单状态 订单总价 订单开始时间和书籍列表）

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 User().get_user_by_id(user_id) 返回值验证用户名是否存在

3. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token
4. 通过 Orderform().check_order_history(buyer_id) 看买家完成交易的所有订单

3.2.12 实现订单定期失效功能

我们设定每过10分钟定时对Orderform表进行全部扫描，对于还未付款、但已经超时的订单进行了处理。由于已经完成的订单数据已经转移到 Orderform_history 表，因此我们任务每过10分钟进行超时数据的整理和清除是一个比较合适的频率。

另外，我们在买家付款功能之前，通过每个通过记录的 transaction_start_time.timestamp() 判断了订单是否超时。这样做兼顾了正确性和性能。

3.3 seller 卖家用户接口

3.3.1 创建商铺

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
创建商铺	POST http://[address]/seller/create_store	user_id, store_id, token(Header中传入)	200:注册成功 501:商铺已经存在 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 Store().get_store_by_id(store_id) 检查商铺是否不存在
3. 通过 User().get_user_by_id(user_id) 返回值验证用户名是否存在
4. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token
5. 通过 Store().add_store(store_id, seller_id) 创建商铺

3.3.2 商家添加书籍信息

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
商家添加书籍信息	POST http://[address]/seller/add_book	user_id, book_info, stock_level, token(Header中传入)	200:添加图书信息成功 401:token不正确 503:商铺ID不存在 504;图书ID已存在 506:seller_id与store_id关系不存在 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token
3. 通过 Store().get_store_by_id(store_id) 检查商铺是否存在
4. 通过 Book().check_book_exists(book_id) 检查书籍是否不存在
5. 通过 Store().check_store_for_seller(seller_id, store_id) 检查商家跟商铺的关系是否正确
6. 通过 Book().add_book(book_info, store_id) 增加书籍信息
7. 通过 Store().add_book_for_store(store_id, book_id, stock_level) 为商家添加书籍

3.3.3 商家添加书籍库存

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
商家添加书籍库存	POST http://[address]/seller/add_stock_level	user_id, store_id, book_id, add_stock_level, token(Header中传入)	200:登录成功 503:商铺ID不存在 519:图书ID不存在 518:表单缺值	message

业务逻辑：

1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 token_tool.check_token_validation(token, buyer_id 验证token是user_id的token，过时清除token

3. 通过 Store().get_store_by_id(store_id) 检查商铺是否存在
4. 通过 Store().check_store_for_seller(seller_id, store_id) 检查商家跟商铺的关系是否正确
5. 通过 Store().add_stock_level(store_id, book_id, add_stock_level) 为商家添加书籍库存

3.3.4 卖家确认发货

要求：

实现功能	Request Method + URL	Request Header / Body	Response Status Code	Response Body
卖家确认发货	POST http://[address]/seller/delivery	user_id, store_id, order_id, token(Header中传入)	200:发货成功 502:卖家ID不存在 506:seller_id与store_id关系不存在 401:用户名不存在或密码不正确 523:order_id与store_id关系不存在 524:订单状态不对 518:表单缺值	message

业务逻辑：

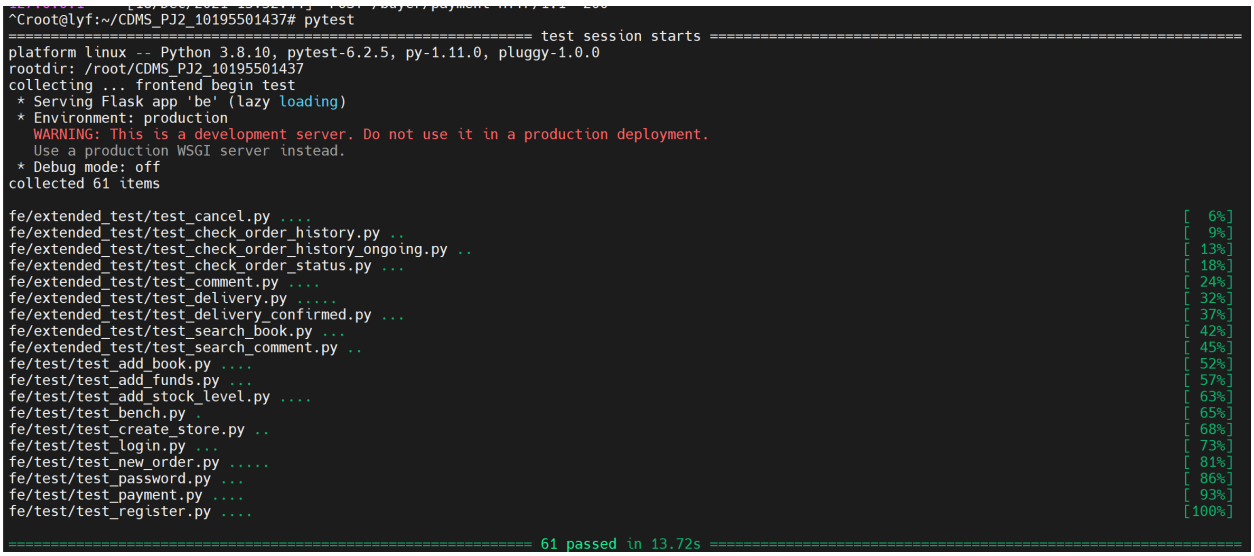
1. 验证 Request Body 提交表单的合法性和完整性
2. 通过 Store().get_store_by_id(store_id) 检查商铺是否存在
3. 通过 Store().check_store_for_seller(seller_id, store_id) 检查商家跟商铺的关系是否正确
4. 通过 Orderform().check_order_exists(order_id) 检查订单是否存在
5. 通过 Orderform().check_order_for_store(store_id, order_id) 检查商家跟订单的关系是否正确
6. 通过 Orderform().get_orderform_status(order_id) 检查订单状态
7. 通过 Orderform().update_orderform_status(order_id, 3) 修改订单状态

4 测试

4.1 正确性测试

对于以上的所有项目功能，我们使用pytest框架编写了对应的测试用例。测试用例一共包含61组，基本包括了所有业务中可能出现的情况。除了正常情况的测试，也考虑了各种可能导致错误的边界情况，如id不存在、用户权限认证失败、重复操作、表单缺值等等。通过这些样例可以保证业务逻辑在绝大多数情况下的正确性。

对于两种与“超时”相关的业务逻辑（token超时和订单超时），由于不易编写测试用例，利用Postman等发包软件进行手动测试，验证其实现正确性。



4.2 代码覆盖率

利用coverage工具统计测试的代码覆盖率。为了达到较高的测试覆盖率，一方面需要测试用例覆盖尽可能多的情况，另一方面要求代码冗余尽可能少，模块和接口有较高的复用率。经过各方面的设计和优化，最终覆盖率达到94%。

```
No data to combine
Name                                                                    Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------------------------------------------
be/__init__.py                                                             20      1      2      1    91%
be/dao/book.py                                                             66      2     12      4    92%
be/dao/db_conn.py                                                          25      0      0      0   100%
be/dao/orderform.py                                                        97      0      4      0   100%
be/dao/store.py                                                            49      0      0      0   100%
be/dao/user.py                                                             34      0      0      0   100%
be/service/auth.py                                                         40      0     14      0   100%
be/service/buyer.py                                                       141     19     76     12    85%
be/service/db_init.py                                                      27      2      0      0    93%
be/service/seller.py                                                       55      6     34      6    87%
be/service/token_tool.py                                                   22      3      4      2    81%
be/view/auth.py                                                            66     10      2      0    85%
be/view/buyer.py                                                          140     22      2      0    85%
be/view/seller.py                                                          59      8      0      0    86%
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                                                          71      1     12      2    96%
fe/access/buyer.py                                                         81      0      2      0   100%
fe/access/new_buyer.py                                                      8      0      0      0   100%
fe/access/new_seller.py                                                     8      0      0      0   100%
fe/access/seller.py                                                        37      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      3    95%
fe/bench/workload.py                                                      122      3     20      3    96%
fe/conf.py                                                                 11      0      0      0   100%
fe/conftest.py                                                             23      0      0      0   100%
fe/extended_test/test_cancel.py                                            44      0      0      0   100%
fe/extended_test/test_check_order_history.py                               46      1      4      1    96%
fe/extended_test/test_check_order_history_ongoing.py                       42      1      4      1    96%
fe/extended_test/test_check_order_status.py                                61      1      4      1    97%
fe/extended_test/test_comment.py                                           54      1      4      1    97%
fe/extended_test/test_delivery.py                                          51      1      4      1    96%
fe/extended_test/test_delivery_confirmed.py                                47      1      4      1    96%
fe/extended_test/test_init_db.py                                            1      0      0      0   100%
fe/extended_test/test_search_book.py                                       49      0     12      3    95%
fe/extended_test/test_search_comment.py                                    47      1      4      1    96%
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_bench.py                                                       6      2      0      0    67%
fe/test/test_create_store.py                                               20      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                                                    60      1      4      1    97%
fe/test/test_register.py                                                   31      0      0      0   100%
-------------------------------------------------------------------------------------------------------
TOTAL                                                                    2099     87    282     44    94%
```

4.3 并发场景下吞吐量测试

以上测试的不足之处在于并没有考虑到并发请求，测试都是单线程进行的。因此还需要对并发场景下进行测试，并测量下单和付款两个接口的吞吐量。

由于原测试用例中对吞吐量的计算方式**明显有误**，且并没有采用多线程并发的方式发起请求（详细见下一部分）。为了得到正确的吞吐量，我们设计了一段测试用例，对并发业务场景进行模拟并测试吞吐率（fe/tps/testTps.py)

> **测试环境**
> 操作系统：Ubuntu20.04
> 处理器：Intel(R) Xeon(R) Platinum 8269CY
> CPU虚拟核数：4vCPU
> 内存：16 GiB

对测试结果绘制折线图如下(fe/tps/tps_latency_pic_example.png)：

左图记录了吞吐率与并发线程数的关系，右图记录了延迟与并发线程数的关系。由结果可见，吞吐率首先随着并发线程数增加而增加，当线程数逐渐增大时，负载逐渐饱和，吞吐率趋向于稳定（线程数>=30后吞吐率达到稳定），其波动主要是由于数据库本身的性能波动造成的。延迟则随着并发线程数增加不断增大。

稳定情况下，**下单接口平均吞吐量约为700，付款接口平均吞吐量约900**。蓝线和绿线分别代表了下单和付款接口的测试情况。由于下单接口涉及的数据库查询和修改操作略复杂于付款接口，其访问代价高于付款接口，所以多数情况下单接口的吞吐量较低，延迟较高。

总体结果符合预期。

4.4 对原测试用例的疑问和改进

4.4.1 book_id重复

在测试商家添加书籍接口（fe/test/test_add_book.py）时进行多次测试，每次测试的都是book.db中id最小的两种书。但如果这样测试，每次都会因book_id重复而返回错误，起不到测试效果。

于是我们对book_id的生成方式进行了适当的修改：在每次获取图书信息时给图书ID加上一个通过随机生成的后缀，保证每次测试使用的图书id不一样，例如修改fe/access/book.py

```
# book.id = row[0]
book.id = row[0] + '_' + str(str(uuid.uuid1()))
```

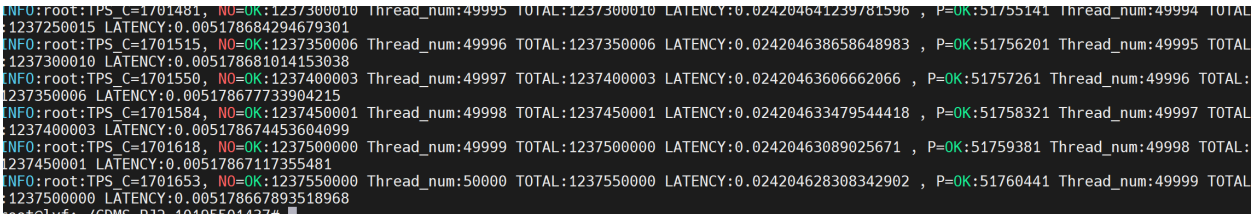
4.4.2 负载生成过程中store_id与book_id不匹配

在fe/bench/workload.py中生成新订单的方法是：在所有店铺中随机选择一个store_id，然后在所有书籍中随机选择一个book_id作为新的订单。但问题在于，这样选择的书籍不一定在选择的店铺中，这会导致生成的订单无效，不能生成有效的负载

于是我们对订单生成方式进行了适当的修改：选择book_id时，不在所有书籍中随机选择，而只在对应店铺中的书籍中随机选择

4.4.3 吞吐率测试和计算方式错误

起初使用原测试用例进行吞吐量测试时，发现线程数与吞吐量呈线性关系，增加线程数至5万后本地测试得到的吞吐量有**170万**，这显然是不合理的。



查看其测试和计算方式后发现吞吐率测试过程中存在以下较明显的错误：

1. `run_get()`函数中，new_order_i变量记录了累计的订单总数，并将该变量作为参数传入`update_stat()`函数更新吞吐量，但在`update_stat()`中却将该参数作为两次更新间隔

中的订单数量，这导致吞吐量被错误地增加了很多

2. 测试没有使用多线程方式发起请求，`update_stat()`函数中直接将订单数作为线程数输出，即使是以伪并发方式进行测试，我认为这也是不合理的
3. `run_get()`函数中，对更新状态的条件判断是：

```
if self.new_order_i % 100 or self.new_order_i == len(self.new_order_request)
```

即原本希望每100个订单对状态进行一次更新，但实际上是每100个订单只有1次不更新，正确写法应该是：

```
if self.new_order_i % 100 == 0 or self.new_order_i == len(self.new_order_request)
```

鉴于原测试存在较多不合理之处，为了测试并发吞吐率，我们在原测试负载生成的基础上自己编写了一段测试代码，完成了上一部分的测试内容。

5 开发过程与分工

5.1 开发过程

1. 合理使用设计分析工具

   我们首先对业务场景进行了分析，设计了ER图，从ER图导出了关系模式，并综合各方面考量，合理设计了索引、冗余和DDL约束。

2. **自顶向下**的软件开发方法

   在开发过程中遵循自顶向下，先设计再实现的原则，即先对各个模块的接口文档进行设计，明确各模块的接口以及模块间的通信方式，再在代码层面实现接口函数。这种开发方式使得模块的职责更加明确，有利于代码解耦合，详实的说明文档则有利于组内成员了解项目进度，以及如何使用其他人实现的接口。

3. git协作开发与版本控制

   我们小组**全程利用git**进行协作开发以及版本控制，分模块完善项目的设计、功能实现与测试。

   branch description:

Branch Name	负责人	Discription	完成情况
Master	@JiaWei_Zhou @Yunfan Li @Ziyan Zhou	总代码仓库	Yes ✅
Data_Manipulation	@JiaWei_Zhou	数据库关系模式与物理设计	Yes ✅
Framework	@Yunfan Li	架构、模块、接口设计	Yes ✅
Realization	@Ziyan Zhou	Auth接口实现	Yes ✅
Seller	@Yunfan Li	Seller接口实现	Yes ✅
Realization 2	@Ziyan Zhou	Buyer接口实现	Yes ✅
Realization 3	@Ziyan Zhou	发货→收货功能	Yes ✅
Auto_cancel	@JiaWei_Zhou	取消订单	Yes ✅
Pytest	@Yunfan Li	pytest修改	Yes ✅
Data_Search	@JiaWei_Zhou	搜索功能、添加常用索引	Yes ✅
Buyer_check_info	@JiaWei_Zhou	买家查找购买历史、订单查询	Yes ✅
Extensive_test	@Yunfan Li	编写附加功能测试用例	Yes ✅
Comment	@Yunfan Li	用户评论和评论查询功能	Yes ✅
Benchmark	@Yunfan Li	吞吐量测试功能修改	Yes ✅
Coverage	@Yunfan Li	完善代码、性能调优	Yes ✅

4. 完整的安装和部署方法

   在功能完成后，利用setuptools等工具，为项目配备了完整的安装、初始化和部署流，并将服务器和数据库在云上成功部署。

5.2 分工

- Git
  - 新建仓库 @JiaWei_Zhou
  - 上传原代码 @Ziyan Zhou
- 关系模式和ER图设计 @JiaWei_Zhou
  - ER图改进：@Ziyan Zhou

- 数据库设计 @JiaWei_Zhou @Yunfan Li
- 项目架构设计、数据库上云、模块设计、接口设计 @Yunfan Li
- 接口实现
  - 基础 auth 用户权限接口 @Ziyan Zhou
  - 基础 buyer 买家用户接口 @Ziyan Zhou @JiaWei_Zhou
  - 基础 seller 卖家用户接口 @Yunfan Li
  - 进阶 发货 -> 收货 功能 @Ziyan Zhou
  - 进阶 搜索图书 @JiaWei_Zhou
  - 进阶 订单状态，订单查询和取消定单 @JiaWei_Zhou
  - 扩展 实现店铺评价、查询评价功能 @Yunfan Li
- 测试
  - 通过基础功能对应的功能测试 fe/test @Yunfan Li @Ziyan Zhou
  - 附加功能测试接口 13 个 @Yunfan Li
  - Pytest 测试接口性能 @Yunfan Li