**华东师范大学数据科学与工程学院上机实践报告**

|  |  |  |
| --- | --- | --- |
| **课程名称**：当代数据管理系统 | **年级**：21级 | **上机实践名称**：网上书店 |
| **指导教师**：周烜 | **姓名**：姚修齐 |
| **上机实践编号**：No.1 | **学号：**10215501419 | **上机截止日期：**12.23 |
|  |  |  |

1. 实验目的
   1. 实现一个提供网上购书功能的网站后端。
   2. 网站支持书商在上面开商店，购买者可以通过网站购买。
   3. 买家和卖家都可以注册自己的账号。
   4. 一个卖家可以开一个或多个网上商店。
   5. 买家可以为自己的账户充值，在任意商店购买图书。
   6. 支持“下单→付款→发货→收货”的购买全流程。
2. 实验任务
   1. **实现对应接口的功能，并通过对应的功能测试，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。
   2. **为项目添加其它功能，包括：**
      1. 实现后续的流程，例如发货与收货；
      2. 图书搜索功能，例如关键字搜索、参数化的搜索方式、范围搜索、搜索结果分页、全文索引优化查找等；
      3. 订单状态，订单查询和取消定单，定单可由买家主动取消或超时仍未付款自动取消。
3. 实验环境

本次实验基于Python 3.7环境，且安装了Flask (2.0.0), Werkzeug (2.0.0), simplejson, uuid, logging, os, sqlite3, time, urljoin, random, base64, auth, PyJWT, requests, pymongo等模块以保证实验的正常进行。

为保证协作效率，我们在Github上建立了本项目的repository，并共同成为了Contributors. 在报告的撰写过程中，我们使用OneDrive进行协作。

本次实验中，我们使用Git Bash, Github Desktop和Visual Studio Code同步本地内容与线上仓库。代码测试过程中，我们使用Git Bash作为测试的终端。

1. 实验过程
   1. **文档数据库结构**
      1. **需求分析**

本次实验中，我们需要为一个线上书店设计后端架构。该网站应实现以下功能：

允许任何人于本网站注册账号，该账号可以同时以买家和卖家的身份活动；

作为卖家时，用户可以开设商店，个数不限；

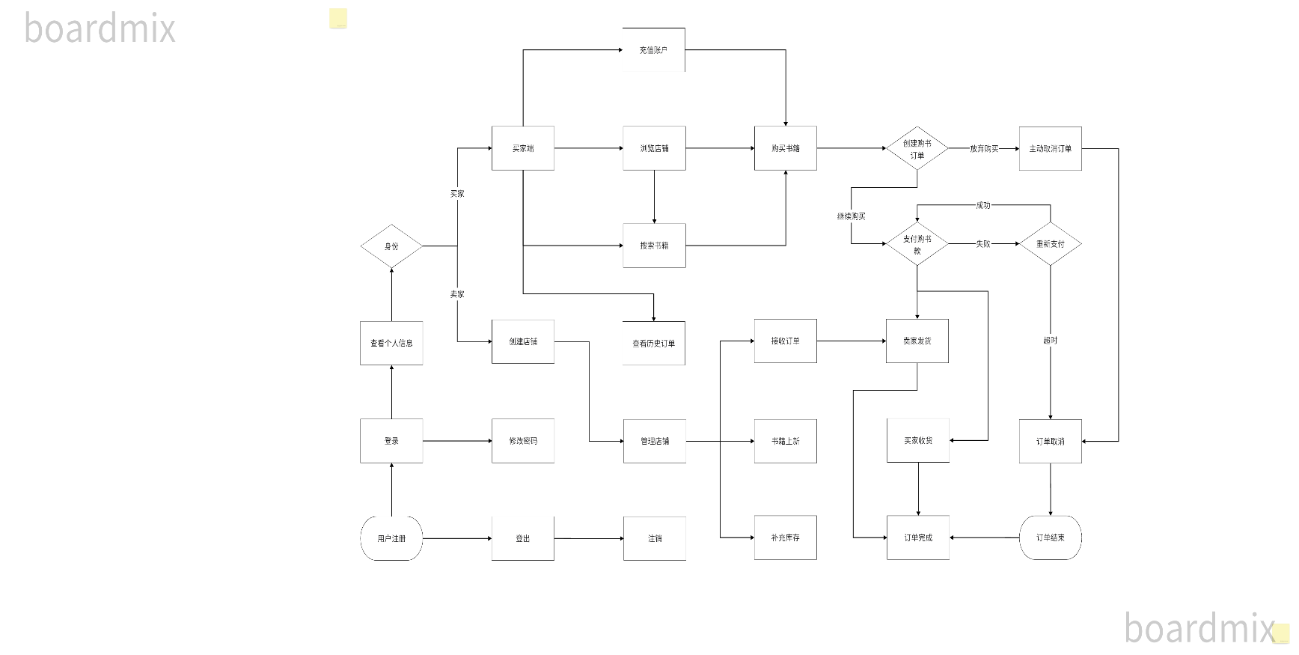
作为买家时，用户可以为自己的账户充值，并使用余额在任意商店购买图书；

网站功能应覆盖从下单到付款再到发货和收货的购买全流程。

*用户端*，我们允许用户注册账号并登录网站。登录后，用户可以浏览各家店铺。浏览结束后，用户可以登出账号。若用户不希望继续使用本平台产品，我们允许用户注销其账号。

*卖家端*，用户可以创建和管理自己的店铺，包括增加店铺中挂出的书籍，补充书籍库存等。在收到新订单后，卖家需要确认订单并发货。

*买家端*，用户可以通过搜索功能检索自己想要的书籍，并购买自己看中的书籍。用户也可以为自己的账户充值，所有购买行为都依赖于账户余额。一笔订单可以以“完成”或“取消”两种方式结束。买家确认收货后，订单视为完成。除买家自愿取消外，购书订单下单后一定时间内买家未支付购书款，订单也会取消。



**图1 数据库需求流程图**

* + 1. **概念结构设计**

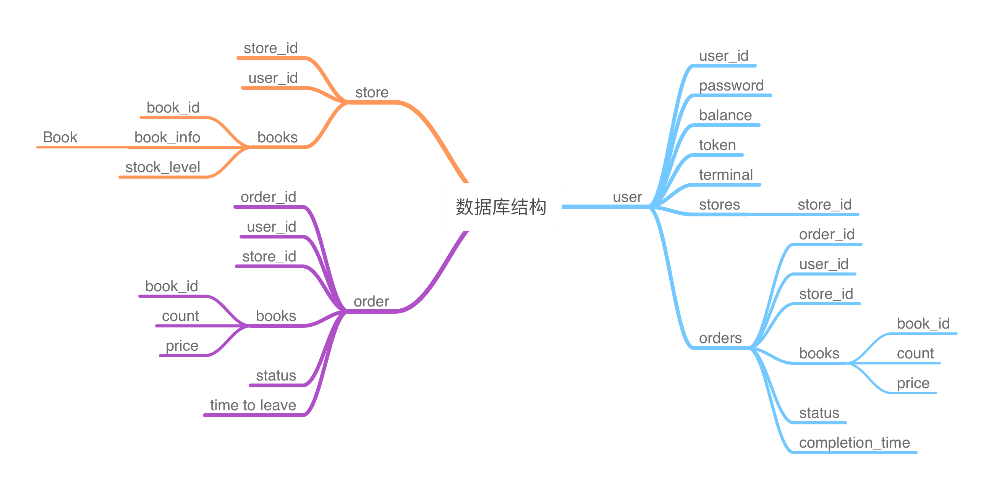
分析题设需求，我们计划对书籍（book），订单（order），店铺（store）和用户（user）四个对象进行设计。分别包含以下信息：

*书籍*：书籍ID、书籍题目、作者、出版社、原书题目、译者、出版年月、页数、价格、装帧方式、ISBN号、作者简介、书籍简介、样章试读、标签和照片；

*订单*：订单ID、买家ID、店铺ID、书籍ID、购买数量、价格、订单状态和截止日期；

*店铺*：卖家ID、店铺ID、书籍（包括书籍ID、书籍信息（见上述书籍信息）和库存）；

*用户*：用户ID、密码、余额、登录令牌、终端、店铺（见上述店铺信息）和订单（见上述订单信息）。



**图2 数据库结构导图**

上述信息中已经包含了一部分对象间的存储关系，我们再来梳理一下：

*书籍与订单*：一笔订单可以包含多本不同的书籍，同一书籍也可以在不同的订单中出现；

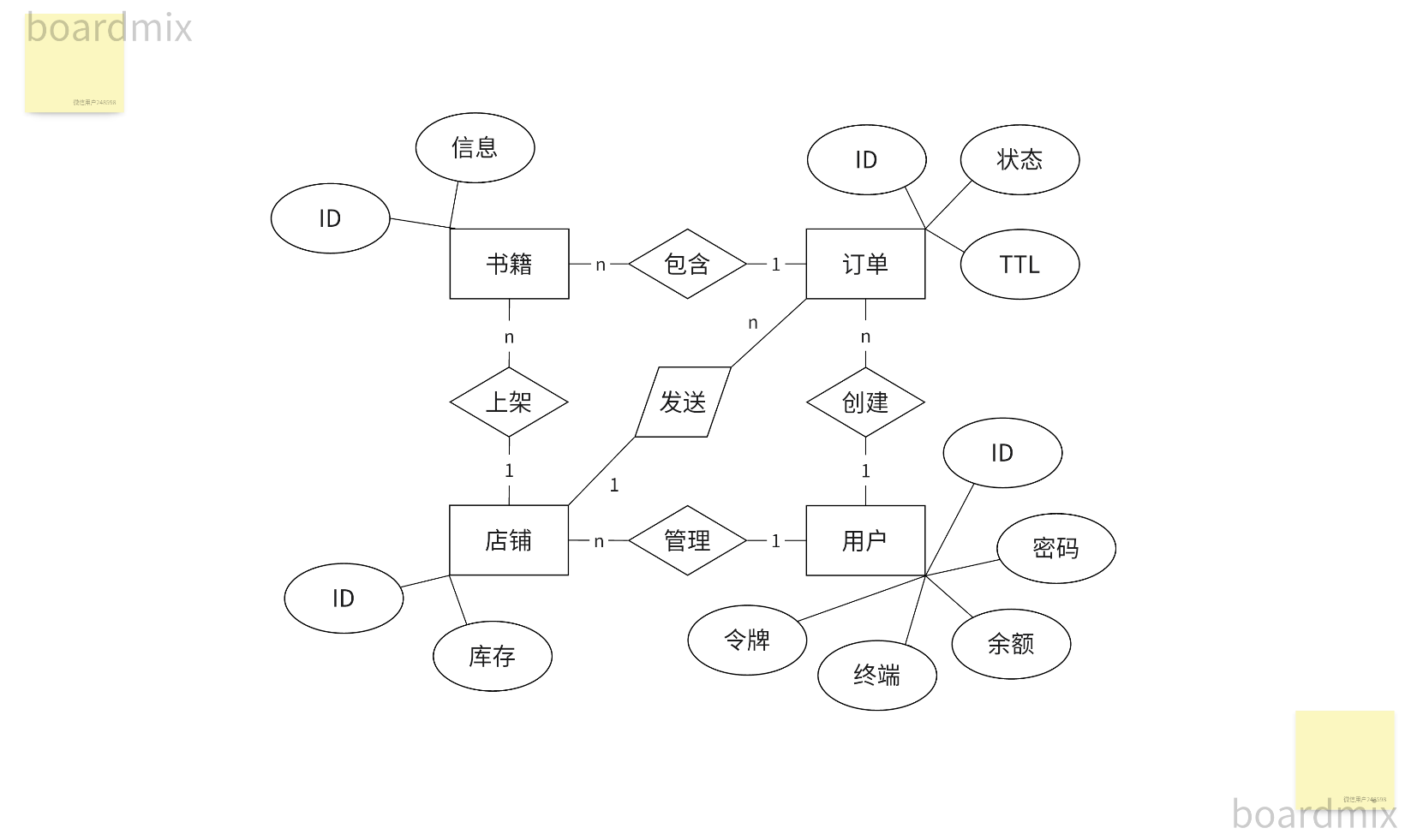
*书籍与店铺*：同理，一家店铺可以出售多本不同的书籍，同一的书籍也可以在不同的店铺出售；

*订单与店铺*：一笔订单只能发送至一家店铺，但一家店铺可以拥有多笔订单；

*订单与用户*：同理，一笔订单只能由一个用户创建，但一个用户可以创建多笔订单；

*店铺与用户*：同理，一家店铺只能由一个用户创建，但一个用户可以创建多家店铺。

具体的概念结构如下所示：



**图3 数据库概念结构图**

* + 1. **逻辑结构设计**

本实验预先提供的代码已经为我们展示了目标功能如何通过SQL由关系数据库实现，我们的工作是使用MongoDB将整个数据库改为文档数据库形式。一个较为直接的思路是，先将提供的全部代码从SQL翻译为MongoDB，再以翻译后的代码为基础进行调整。在与其他组同学的交流中，我们发现有很多组的确选择了这样的方式。

然而我们认为，这样的修改方式是冗余且低效的。文档数据库与关系数据库有着根本的不同，将关系数据库转换为文档数据库最直接的方式是理清原数据库的设计逻辑后，在保持原有功能不变的前提下重新设计。 故有数据库逻辑结构如下：

User{  
 user\_id,  
 password,  
 balance,  
 token,  
 terminal,  
 stores[store\_id],  
 orders[

{ order\_id,  
 user\_id,  
 store\_id,  
 books[{book\_id, count, price}],  
 status,  
 completion\_time

}

],

}

Order{  
 order\_id,  
 user\_id,  
 store\_id,  
 books[{book\_id, count, price}],  
 status,  
 TTL,  
}

Store{  
 store\_id,  
 user\_id,  
 books[

{

book\_id,  
 book\_info[Book],  
 stock\_level

}

],  
 }

Book{  
 id,  
 title,  
 author,  
 publisher,  
 original\_title,  
 translator,  
 pub\_year,  
 pages,  
 price,  
 currency\_unit,  
 binding,  
 isbn,  
 author\_intro,  
 book\_intro,  
 content,  
 tags[],  
 picture[],  
}

* 1. **代码实现**
     1. **基础功能**

前60%的工作以修改demo中部分涉及SQlite数据库的操作为主，将其修改为对MongoDB数据库的操作即可。

**be/model/store.py**

该文件实现了初始化数据库的功能，连接数据库，并为每个文档集设置唯一索引，这里使用各文档的id作为唯一索引。在order文档集中同时设置一个TTL索引，后续详细讨论。

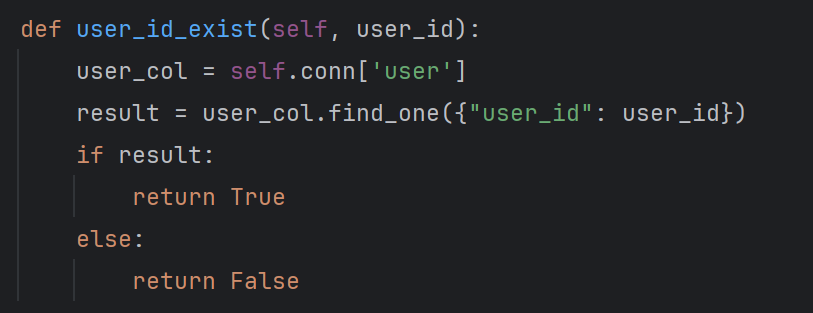


在get\_db\_conn()中连接mongodb数据库，返回数据库对象。



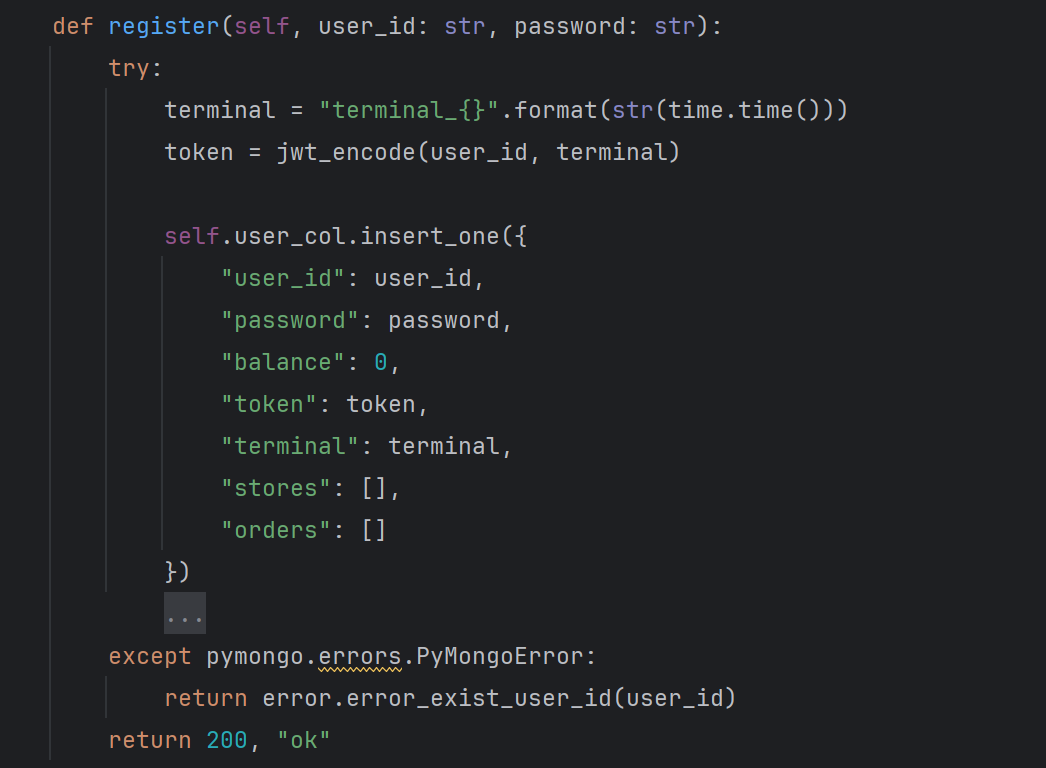
**be/model/db\_conn.py**

该文件通过调用store.py实现了连接数据库的功能，并有三个方法分别可以判断user、book、store的id是否存在。三种方法实现一样，这里以user\_id\_exist()为例，直接查找所给user\_id的用户，如果找不到就返回False。



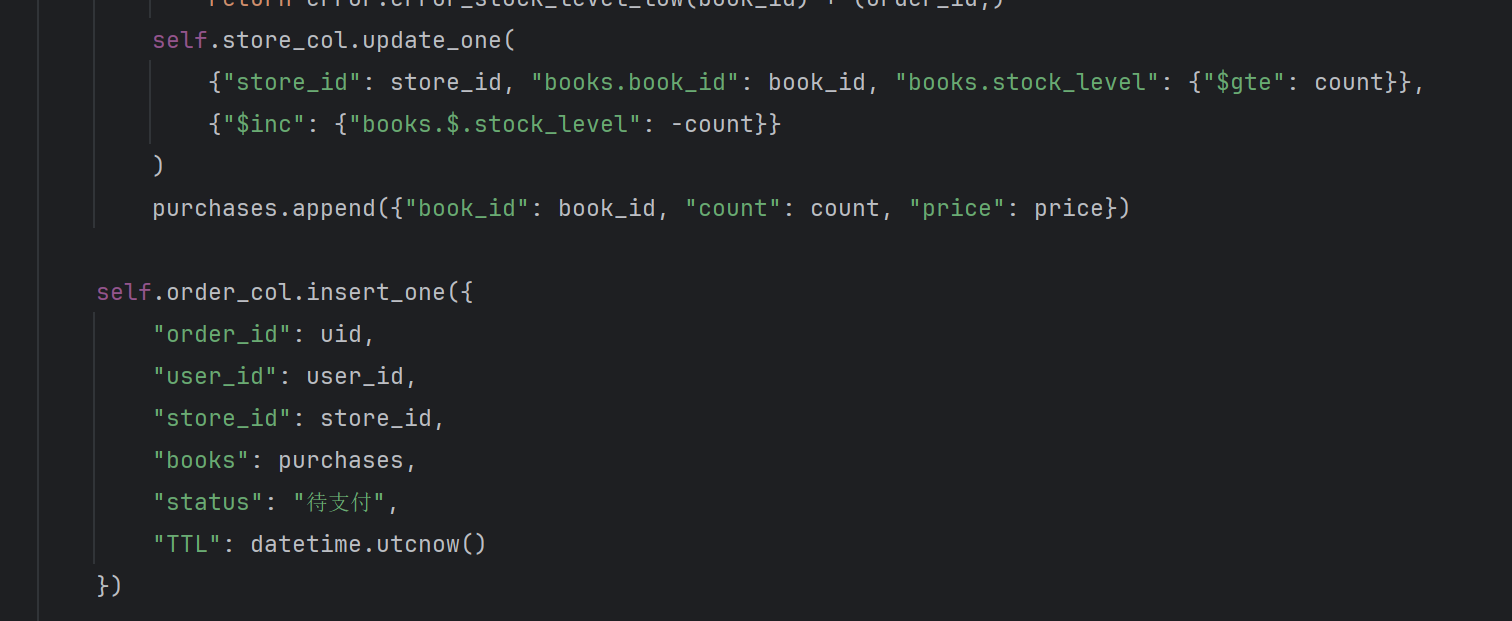
**be/model/user.py**

该文件实现了user的功能：注册、注销、登录、登出、修改密码。其中的代码修改部分也即将对SQlite的操作改为对Mongodb的操作，都是大同小异的，以register()为例，初始化用户的信息，将其插入进user文档集中。



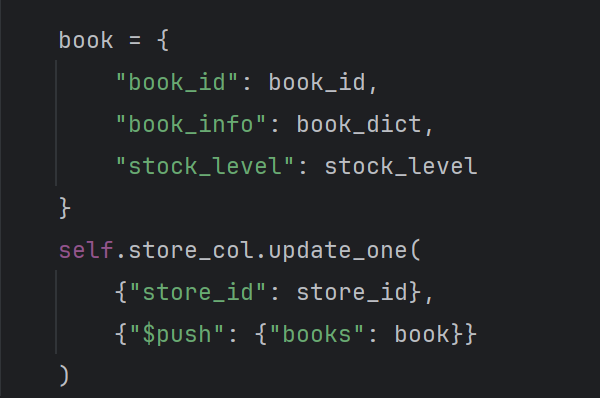
**be/model/buyer.py**

该文件实现了buyer的功能：下单、付款、充值。以下单为例，订单信息确认无误后将商店的库存更新，并将该订单插入order文档集中。



**be/model/seller.py**

该文件实现了seller的功能：创建商铺、添加书籍、添加库存。以add\_book()为例，先将book信息作为字典构造好，再将其插入store文档集的books列表中。



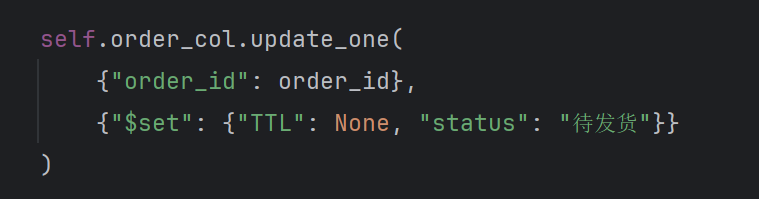
* + 1. **拓展功能**

后40%部分，对于每个功能，我们将从后端实现、接口、代码测试三方面介绍。具体实现的接口可以参考 ./doc/后续.md。

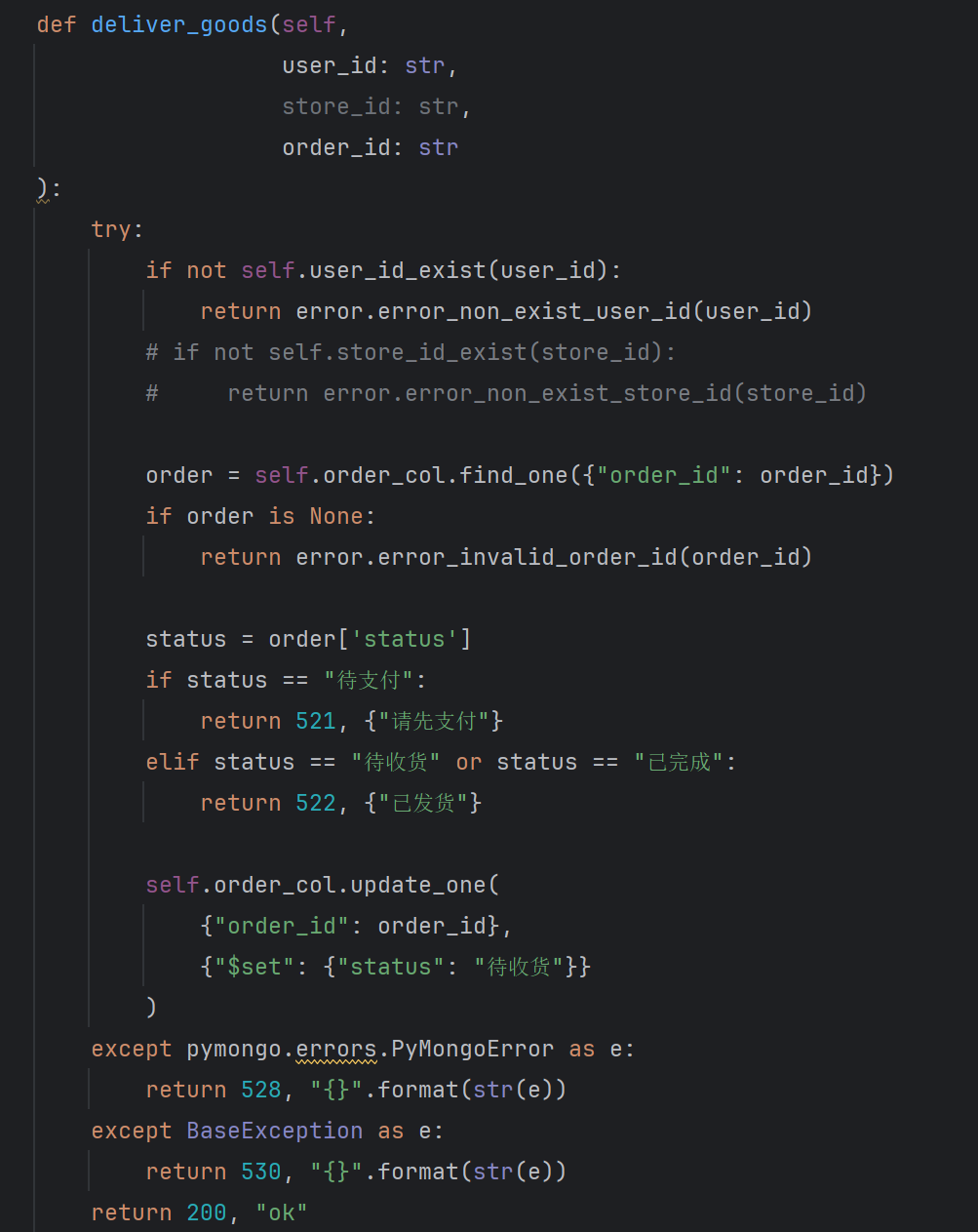
1. **卖家发货**

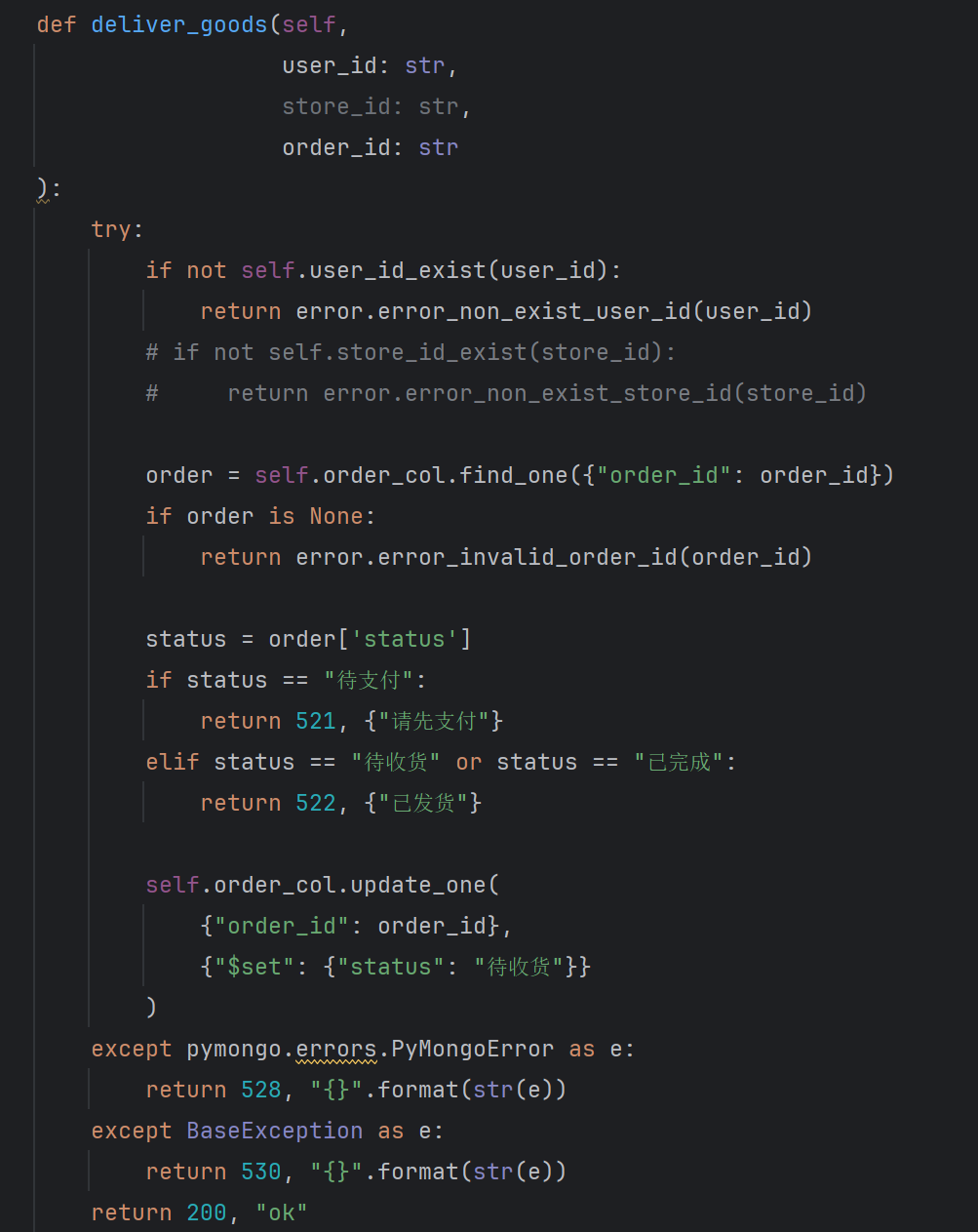
**后端实现**

显然，只有在用户支付之后才能执行发货操作，所以我们首先在payment()函数中将已支付的订单的状态status修改为“待发货”，并把TTL项删除。



对于已经支付的订单，我们将order文档中的状态status直接改为“待收货”即可；但是对于未支付的订单（status为“待支付”）或者已经发货的订单（status为“待收货”、“已完成”），程序将返回错误信息。

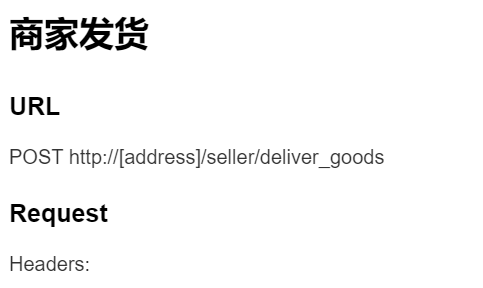




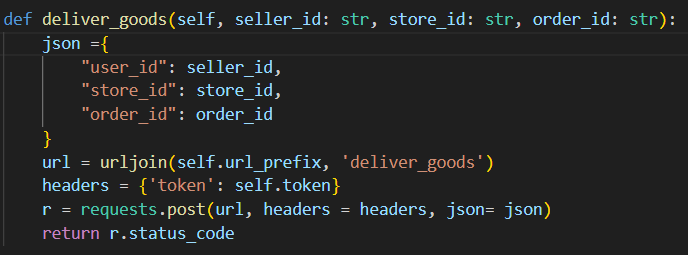
事实上，除非订单状态是“待发货”，前端网页不会显示发货按钮。

**前端接口**

在实现接口之前，我们首先要确定传递的数据有哪些。对此，我们借鉴doc文件夹里其他的markdown文件的写作模式，编写了一个markdown文件用于说明后40%功能的实现方式。



在这里，我们根据功能划分不同的函数。例如卖家函数对应的deliver\_goods需要卖家ID（seller\_id）、店铺ID（store\_id）和订单ID（order\_id）。同时，我们还需要在函数头部匹配登录时产生的会话标志token。

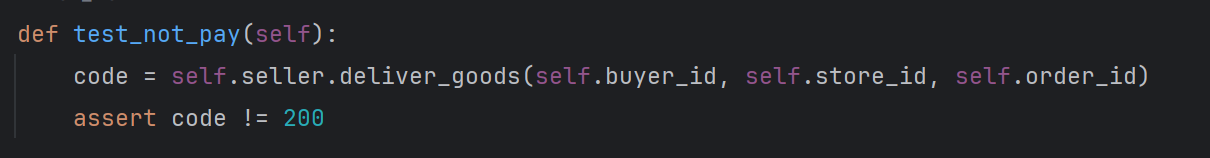


**测试代码**

测试之前将测试用户名不存在、密码错误、订单错误等信息有误的情况（后续测试相同，不再赘述）。首先模拟用户成功下单，而后对于该订单，分三种情况测试：

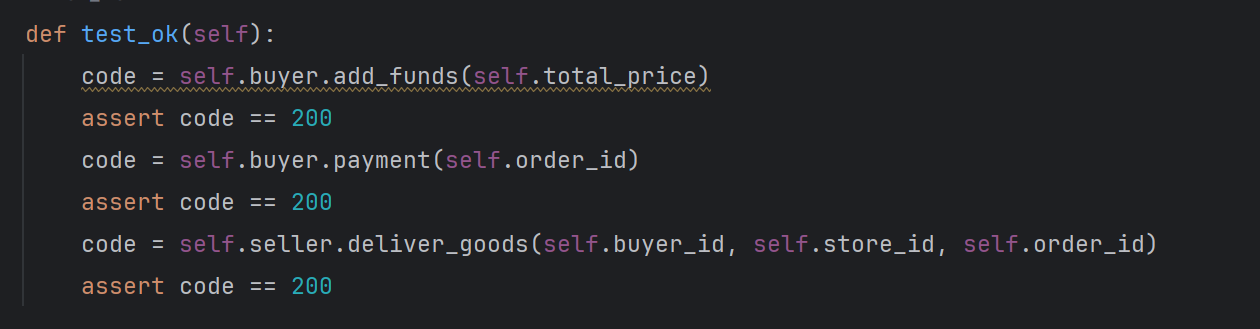
·**未支付**

对订单不支付而直接调用发货接口，理应报错。



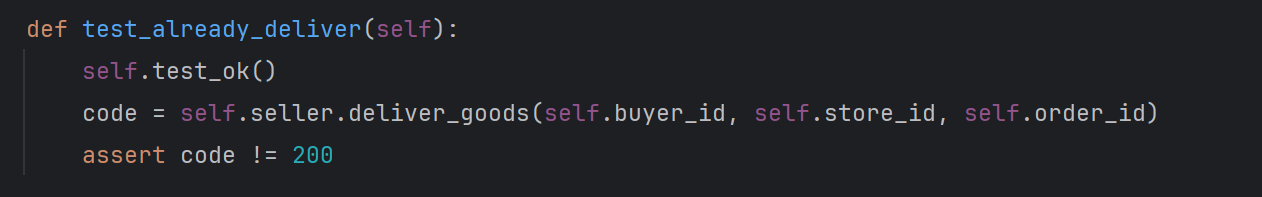
**·正常情况**

支付上述订单，支付成功后调用发货接口，理应通过。



**·已经发货**

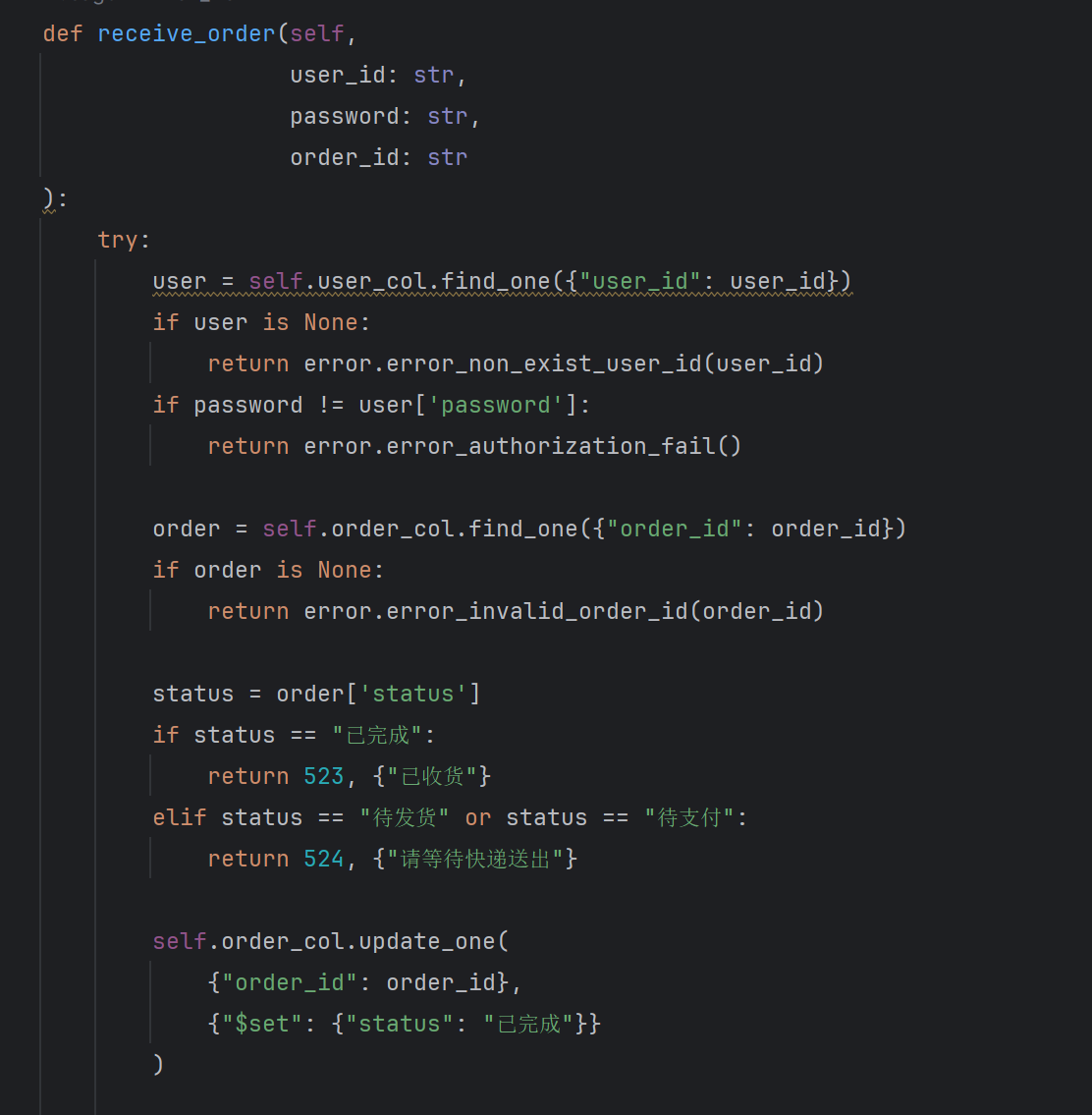
成功发货后再调用发货接口，理应报错。



1. **买家收货**

**后端实现**

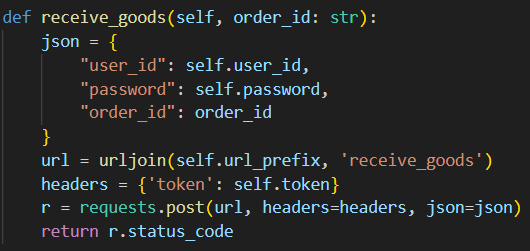
同发货相似，只有在卖家发货后卖家方可收货。所以，若订单状态为“待支付”、“待发货”或“已完成”时调用本函数都将报错。对于已经发货的订单，我们将order文档中的状态status直接改为“已完成”即可。



事实上，如果状态是除“待收货”以外的其他状态，前端网页不会显示收货按钮。

**接口**

买家收货的接口写在access/buyer.py中，同样的这里的receiver\_goods函数需要传递买家id：user\_id，密码：password和订单id：order\_id三个部分，同样需要匹配token标识符。



**测试代码**

同先前一样，首先模拟用户成功下单，而后对于该订单，分三种情况测试：

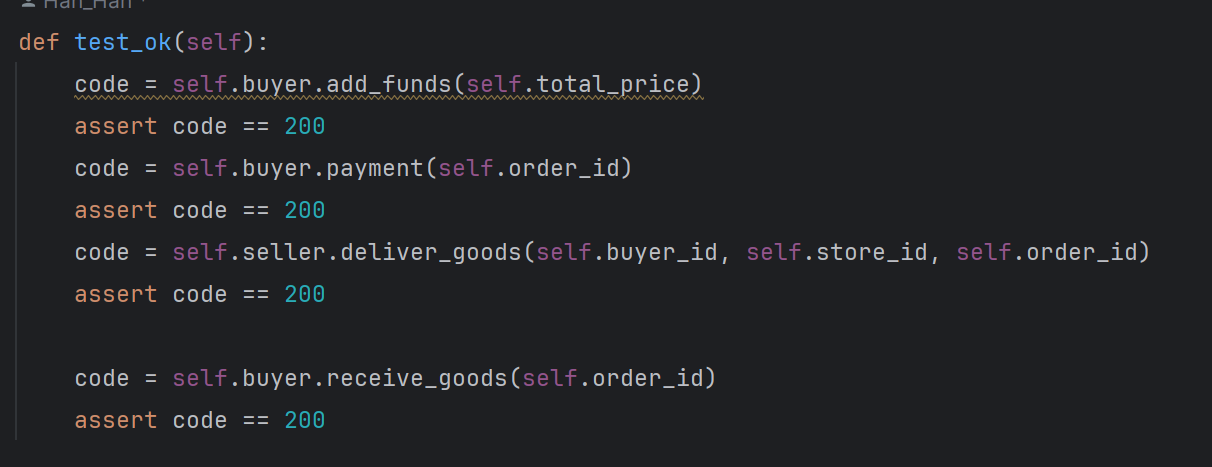
**·未发货**

对未支付或者未发货的订单调用收货接口，理应报错。



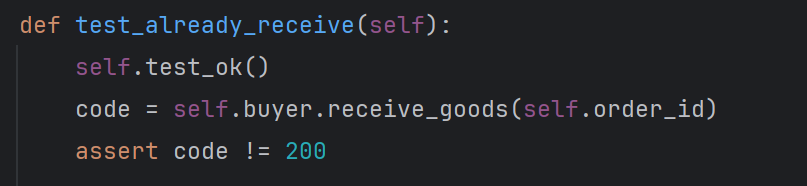
**·正常情况**

支付上述订单，卖家发货后调用收货接口，理应通过。



**·已经收货**

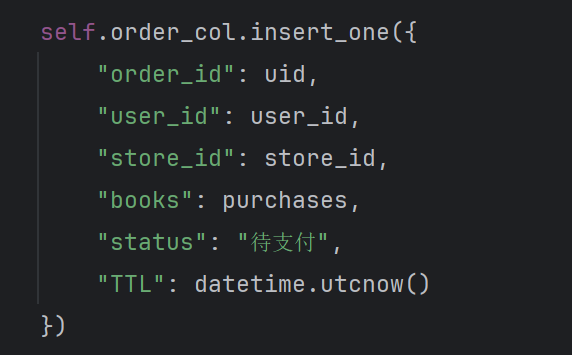
成功收货后再调用收货接口，理应报错。

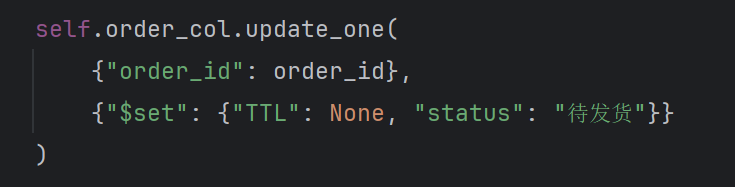


1. **取消订单**

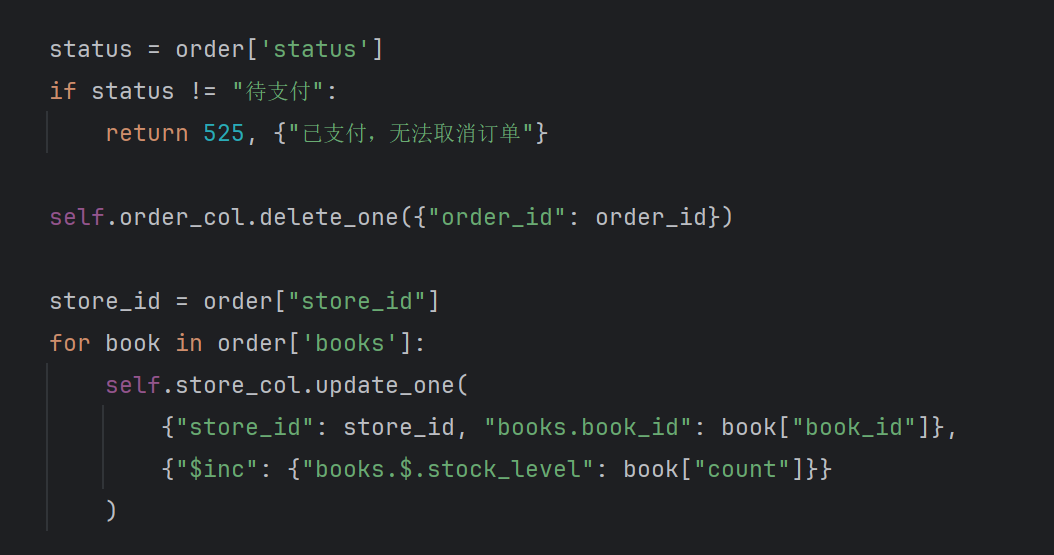
**后端实现**

自动取消订单，即买家下单后一定时间内未支付，系统删除该订单。该功能实现较为简单，只需在order文档集中添加一个TTL索引，设置过期时间为30秒，并在下单时将TTL设置为当前时间。此时若下单30秒后买家仍未支付，系统将删除订单。若30秒内付款成功则将TTL删除。



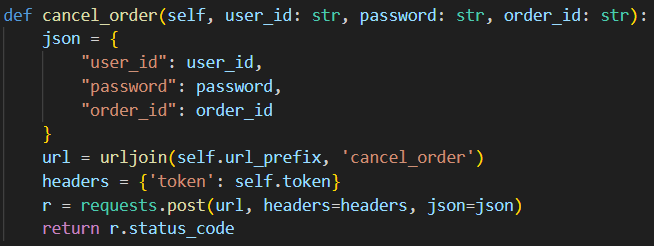


主动取消订单，该操作只能在买家尚未付款时执行，即status为“待支付”。取消时需将订单从order文档集中删除，并将包含于该订单中的书籍库存还原。



**接口**

根据markdown文件，此处接口与买家收货的receive\_goods函数的接口几乎一致，只是url的名字不同。故而我们将url的后缀修改为cancel\_order即可。

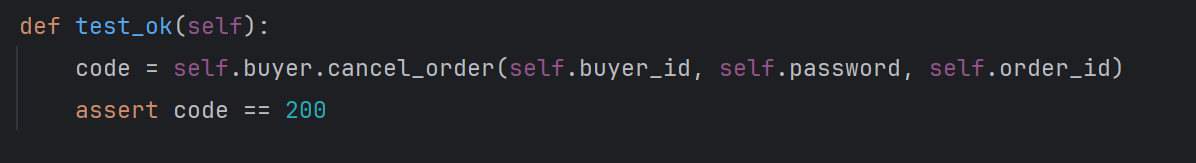


**测试代码**

实现逻辑同上，首先模拟用户成功下单，而后对于该订单，分三种情况测试：

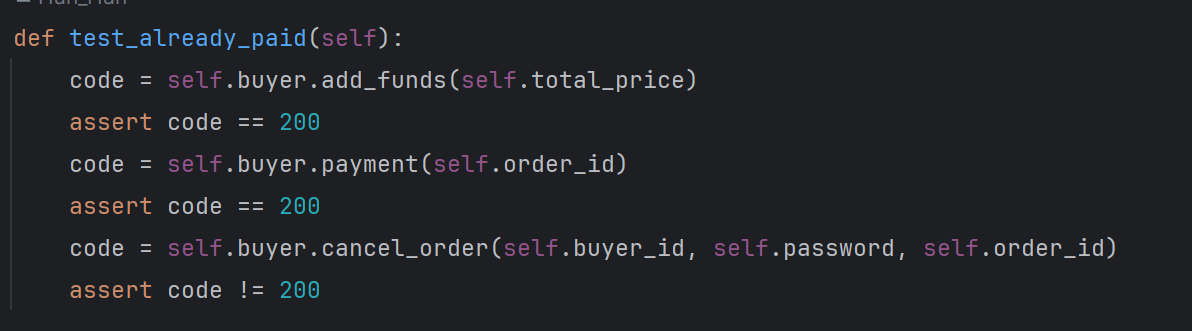
**·正常情况**

下单但未支付，取消订单理应成功。



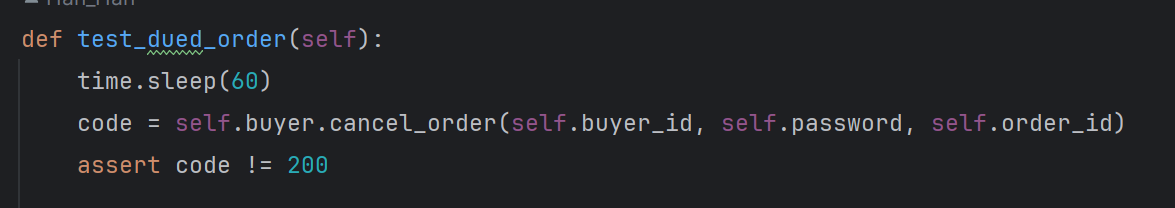
**·已经支付**

对于已经支付的订单调用取消接口理应报错。



**·订单过期**

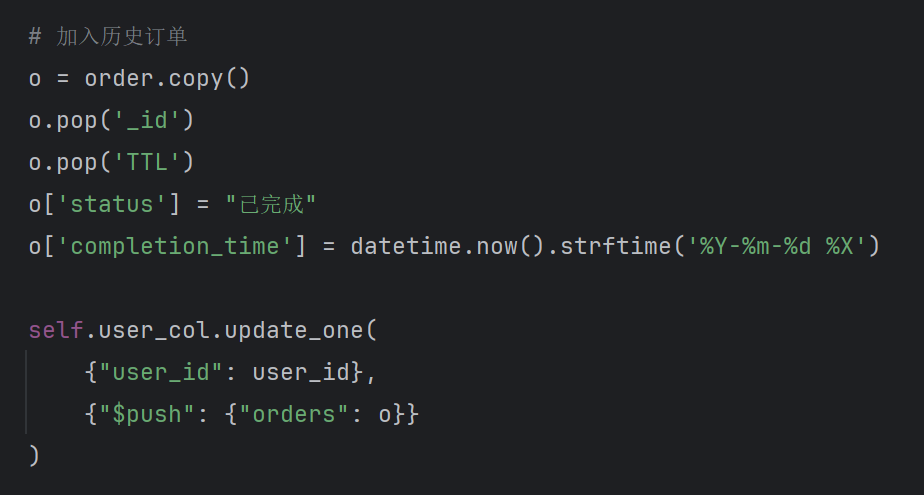
下单但超时未支付已自动取消的订单调用取消接口，理应报错。

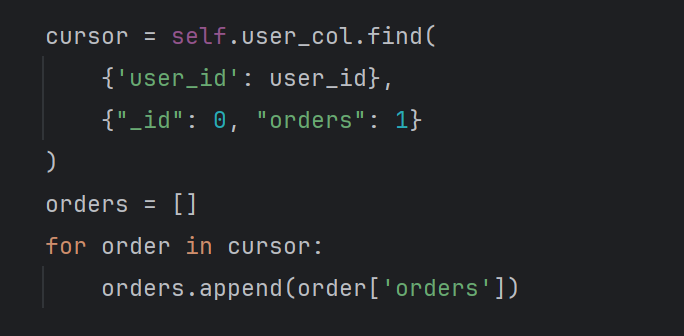


1. **查询历史订单**

**后端实现**

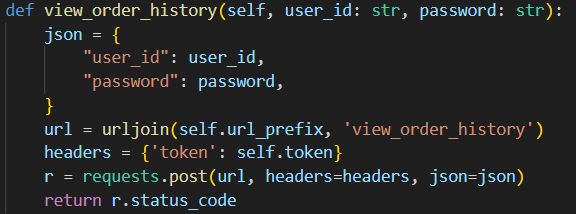
在user文档中，orders用来记录该用户所有历史订单。买家每次收货成功，即订单完成后，该订单将加入买家的历史记录中。需要查询历史订单时，直接查询orders项中内容即可。





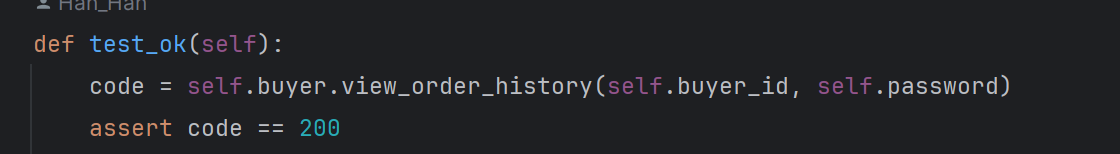
**接口**

买家查询订单历史只需要匹配买家id（user\_id）和密码（password）即可，同样也需要登录才能操作。



**测试代码**

此功能当用户名密码正确时调用接口应正确。



1. **搜索图书**

**后端实现**

搜索图书需要传入一些搜索信息：store\_id、title、tags、content，这些信息都可以为空，为空意味着不考虑该项信息，如  表示在“wyh的小店”中搜索所有标题中含“计算机”的书，其他条件不限。

搜索时先将数据库按照book展开，并在其中使用正则表达式搜索符合条件的项目，最后输出其所在店铺、标题等信息。

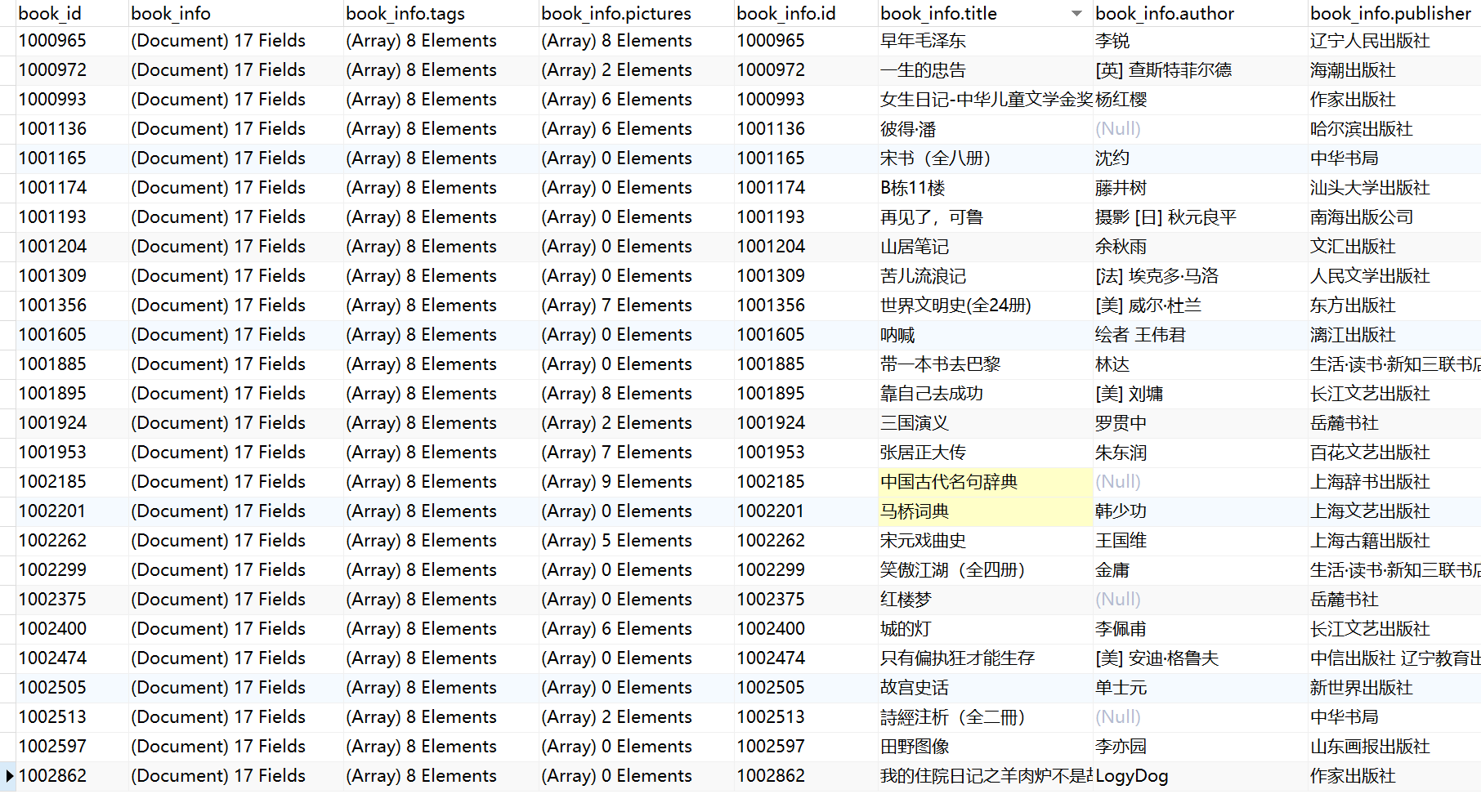


**测试代码**

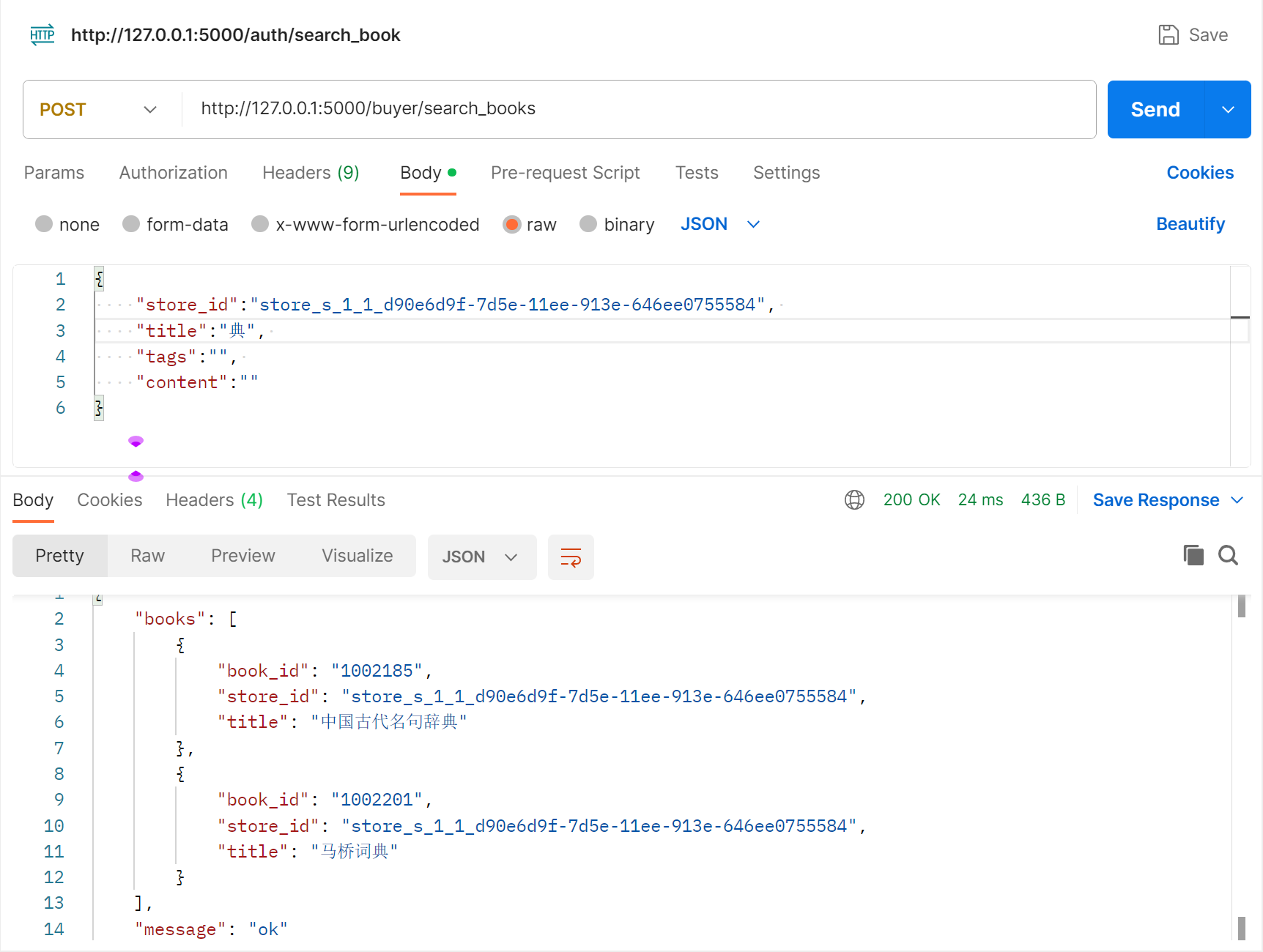
搜索书籍不需要登录，即使是没有注册的用户也可以搜索，并且一旦搜索总是能成功（除非服务器崩溃）。所以对于该功能，设计测试代码没有应用意义，此处使用postman展示代码的正确性。

首先如图选用先前创建的商店，搜索标题中含“典”的书籍，该商店中只有两本。

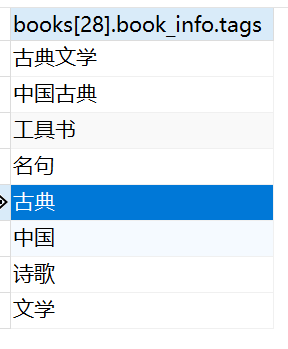


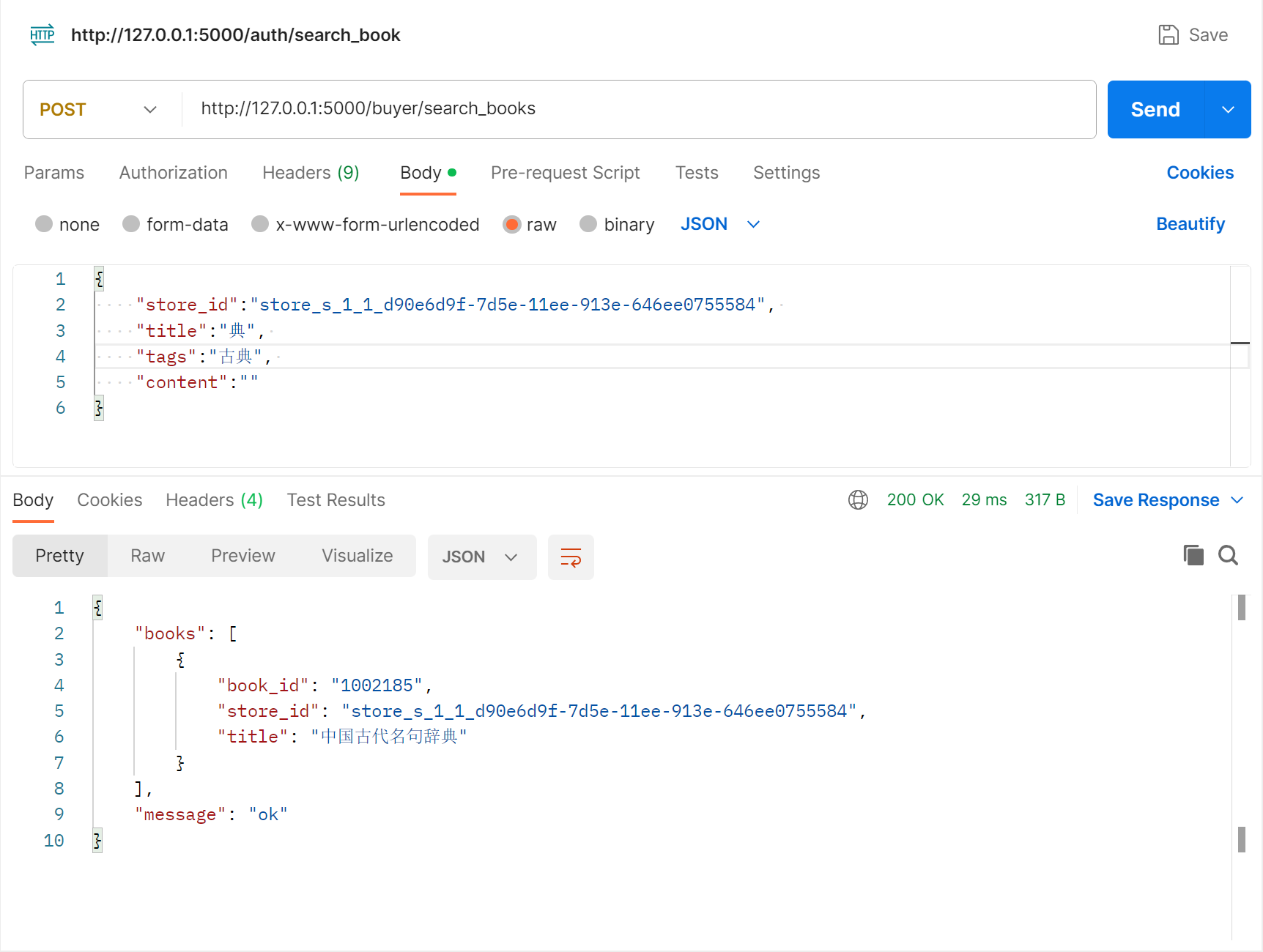


使用postman发送如下请求，由返回结果可见与数据库中一致。



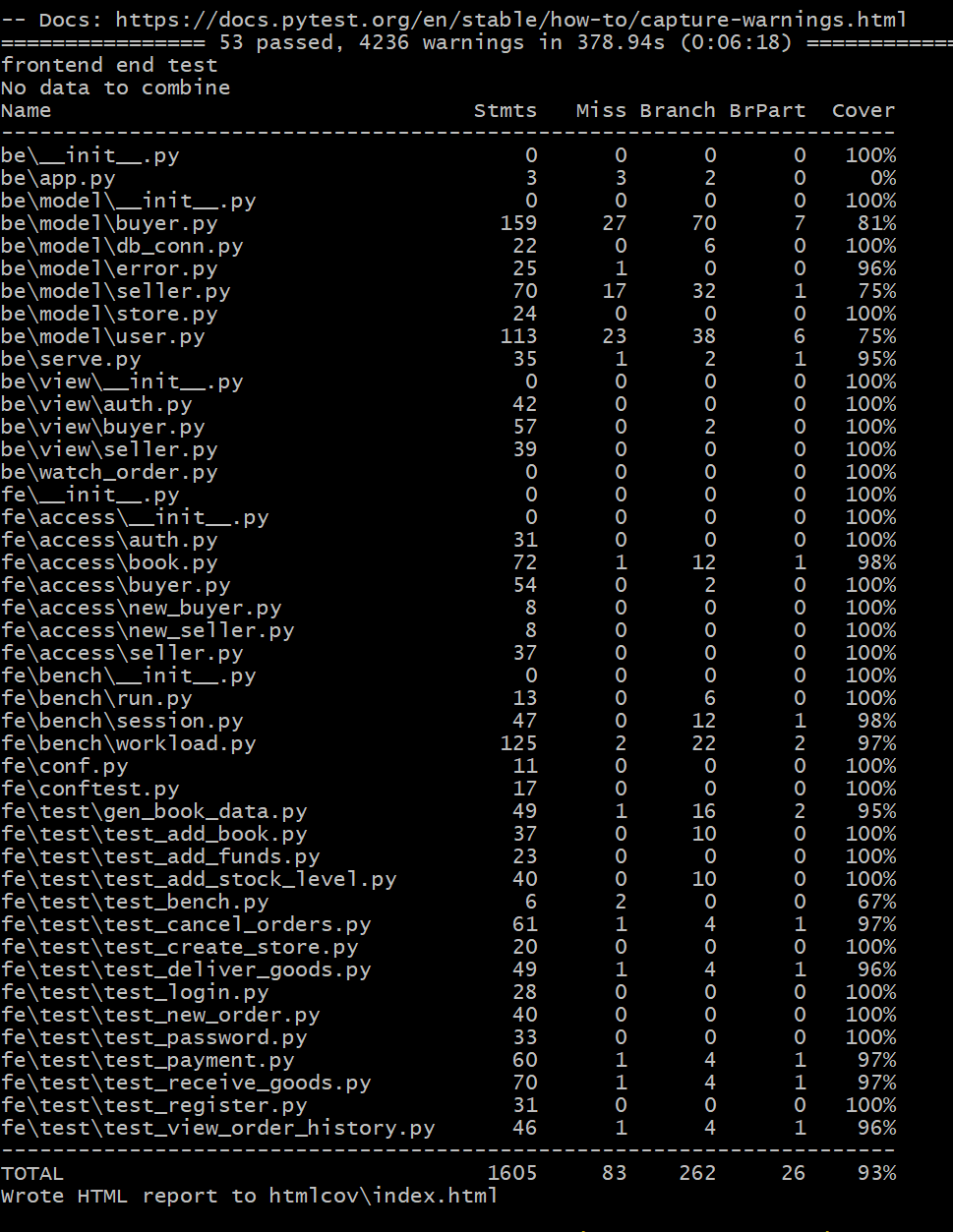
进一步地，前者标签中含“古典”而后者不含，故将标签加入搜索条件，结果也符合预期。

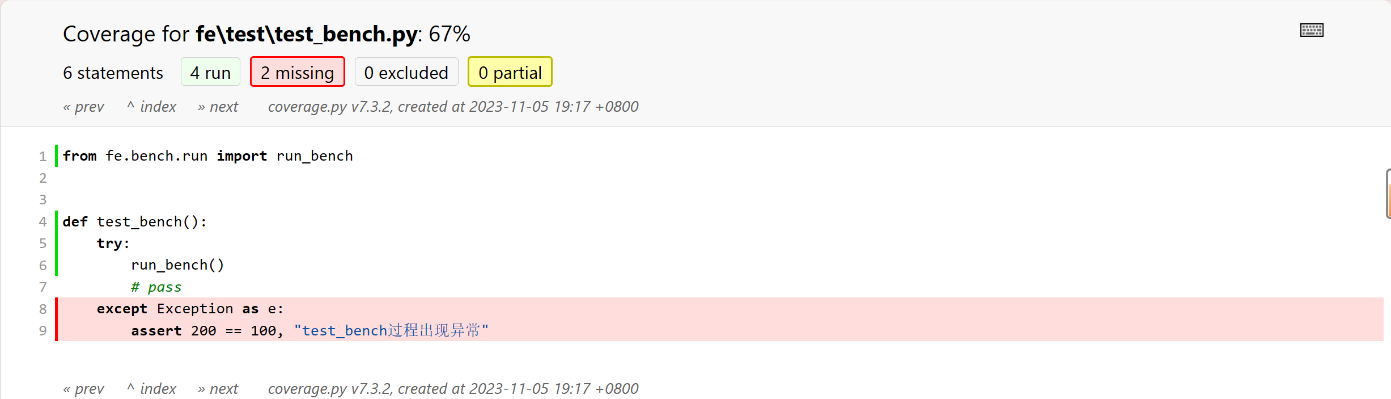




* + 1. **pytest测试结果**

接下来展示完整的代码测试结果。由于search\_books()功能并未编写pytest测试代码，考虑到覆盖率问题，进行pytest测试时先将其注释掉。测试结果如下图所示，所有测试案例均pass，覆盖率达到93%。由于代码中设计try-except结构，所以部分文件覆盖率低（例如test\_bench的覆盖率仅有67%）是正常情况，不影响实际测试覆盖率。

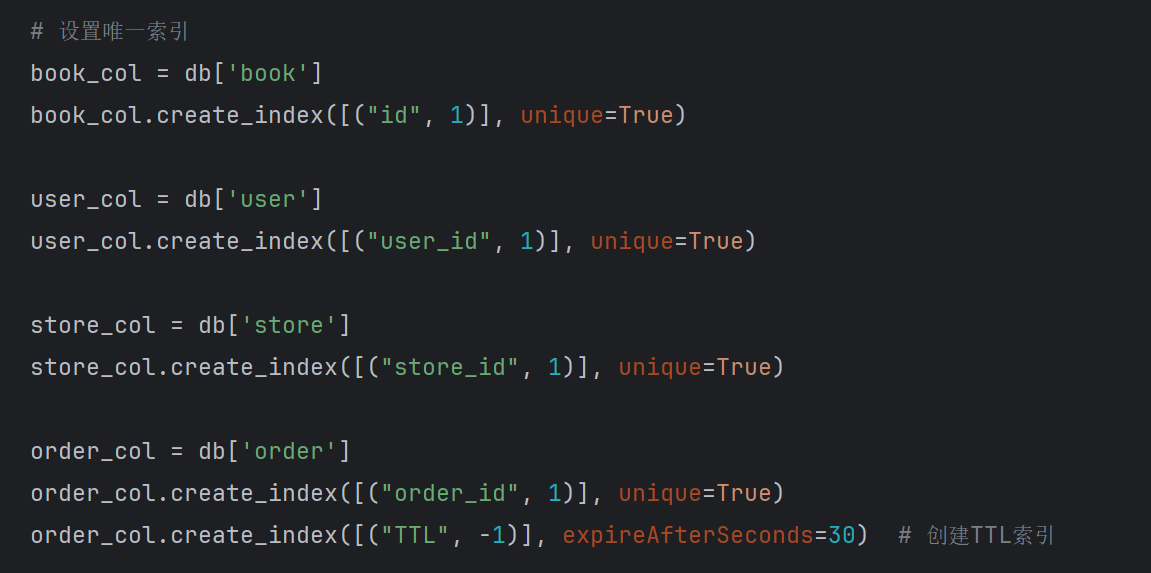




* 1. **设计亮点**

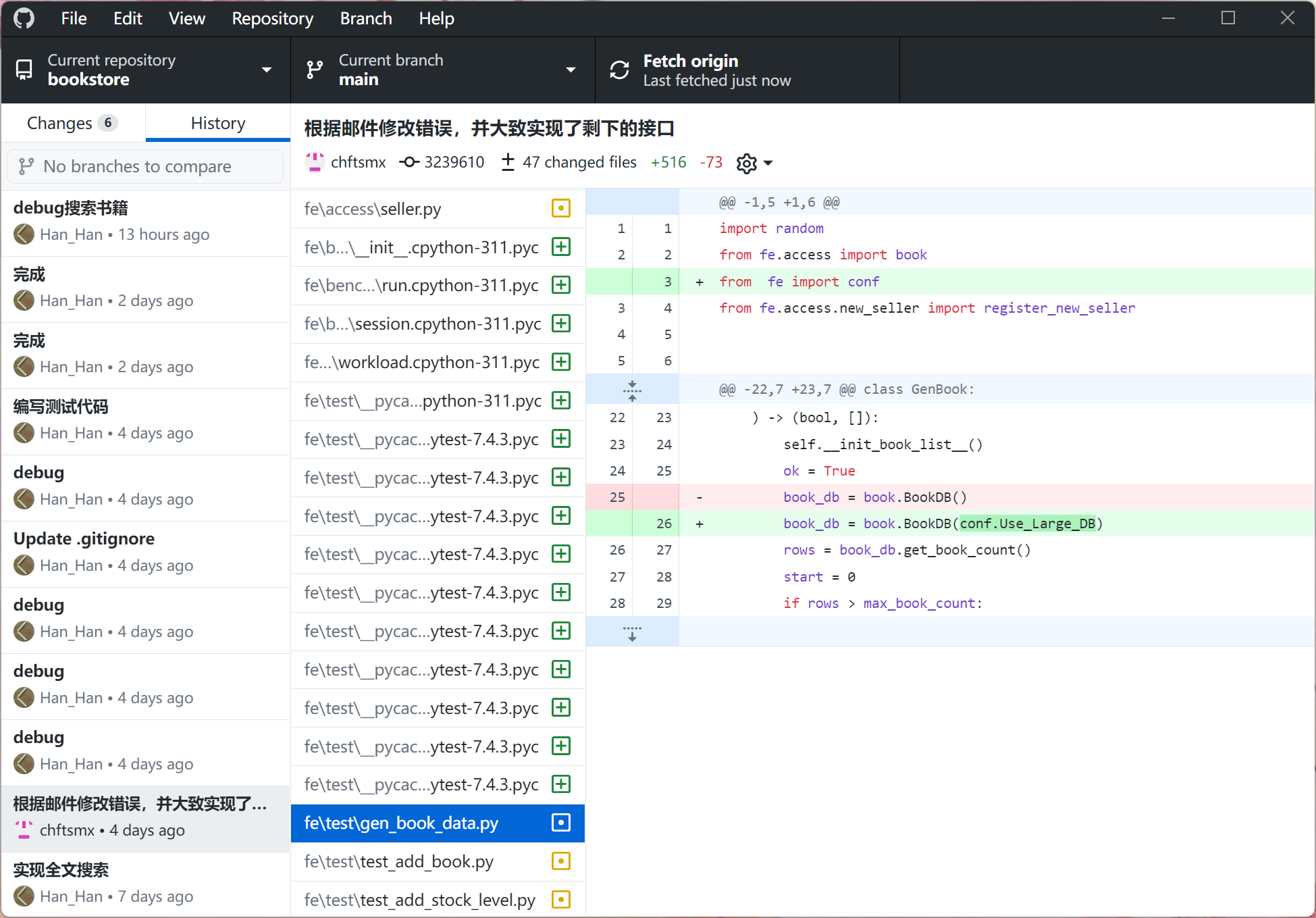
**·添加索引加快数据库执行**

考虑到寻找文档是所有数据库操作的执行前提，并且通过id寻找是唯一的寻找方式，我们为每个文档集都添加了id索引，以加快数据库执行速度，提升其性能。同时，这些id均为唯一索引，确保不会有同名的用户、商店和书籍等。



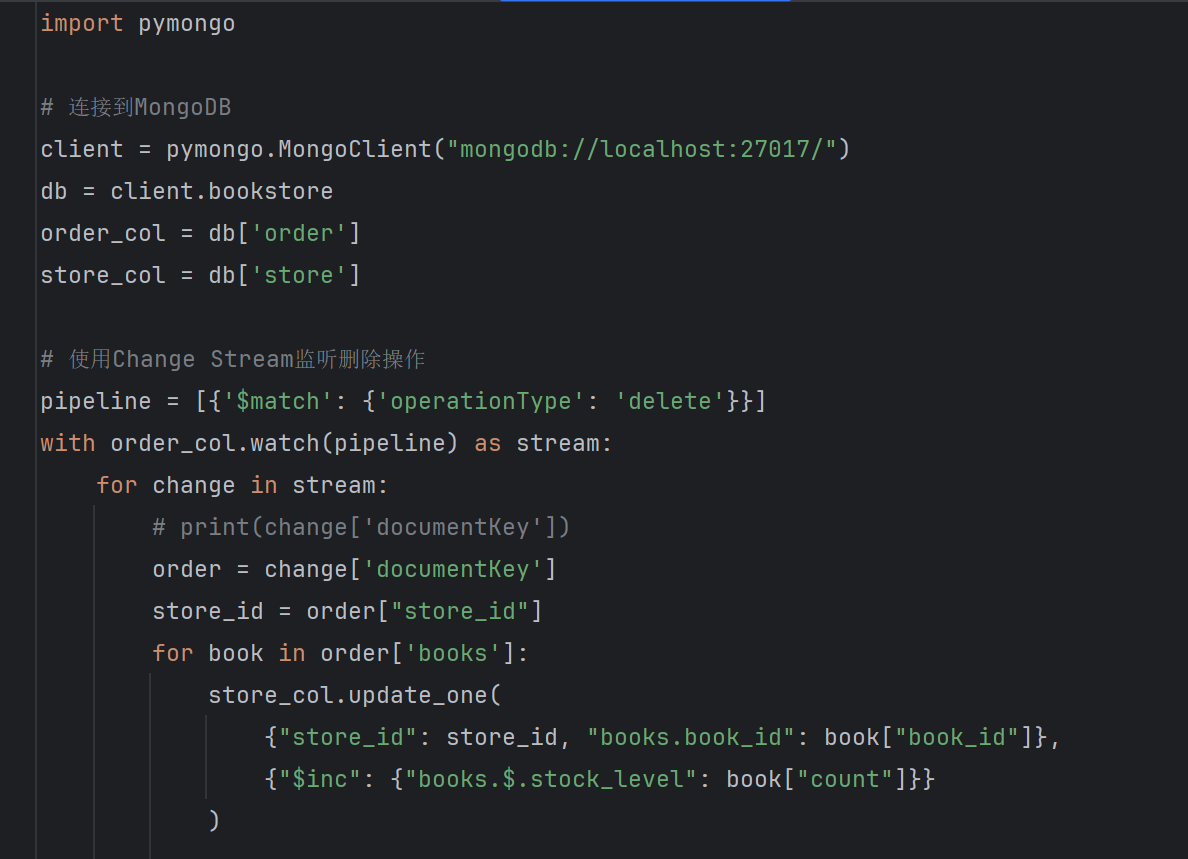
**·基于git共同创作管理项目**

本次项目由三人共同协作完成，项目已上传至[wyhwhye/bookstore: DBMSProject1 (github.com)](https://github.com/wyhwhye/bookstore)，所有修改提交都是基于git管理系统实现以提高项目开发效率、灵活回退版本，降低开发难度。

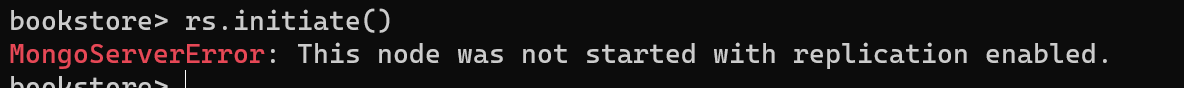


* 1. **不足与展望**

如前文所示，用户手动取消订单时，可以很容易获取到订单信息，并将订购的书的数量加回库存。但是在订单过期，即系统自动删除该订单时如何将书加回库存呢？此处我们尝试使用Change Stream监听数据库的变化，但是因其只能对复制集合监听，并没有成功。因此，我们尝试对order文档集复制一个副本，但模仿网上windows的相关操作方式并没有成功，此为本实验的一个遗憾之处。日后我们将考虑使用云linux系统和云数据库重新实现这一功能。







1. 任务分工

丁子昕：修改user.py、book.py并补齐后者；实现后40%任务的后端、测试代码，撰写实验报告

吴郁涵：修改store.py、db\_conn.py；实现后40%任务的后端、测试代码；撰写实验报告

姚修齐：修改buyer.py、seller.py；数据库设计可视化；撰写实验报告