STJU DB Bookstore Project, Report of Group 4

0. 组员

组号：第四小组

姓名	学号	分工
林超凡（组长）	520021911042	前 60% 基础架构搭建，文档撰写，完善测试，代码管理
潘开森	520030910140	订单查询，订单发货、收货、超时过期机制
翟明舒	520021910671	图书查询，索引添加

1. 数据库设计

本项目后端数据库采用 MongoDB 文档数据库。在设计上分为四个文档：

Doc user

- _id (user_id), str
- password, str
- balance, int
- token, str
- terminal, str

Doc book

- _id (store_id, book_id), pair of str
- **book_info, class
- stock_level, int

Doc store

- _id (store_id), str
- owner (user_id), str

Doc order

- _id (order_id), str
- buyer (user_id), str
- store (store_id), str
- total_price, int
- books, list
    - book_id, str
    - count, int
    - price, int
- state, str

    - 可能出现的值: `unpaid`, `paid`, `delivered`, `canceled`, `finished`
- timestamp: float

说明

- **_id (user_id)** 的意思是，这个文档我直接将 _id 赋值为相应的 user_id，因为在作业中像 user_id 这种属性天然可以作为主键。
- 在 book 这一文档里，book_info 这里前面加上了 `**`。这个意思是，book_info 这个 class 在我们的文档数据库其实是被展开的，也就是说 book_info 中的成员将直接作为这个文档的第一层 key 存在。（借用了 Python 里 `**` 的意思）
- 关于 book 这个文档：在设计考虑上并没有将一个 store 出售的所有 books 以一个 list 存在 store 文档下，而是仍然采用类似关系数据库的设计，将 book 文档的 _id 定义为 `(store_id, book_id)`，这是因为考虑到在本作业中，书永远是从属于一个 store 的，这样设计更贴近需求，并且效率更高。
- 关于订单 order：在设计上运用了一些冗余，比如储存了这个订单的总价（total_price），每本书的价格（price）。总价在构造订单时即可比较方便地算出，这样在 payment 的时候只需要查此字段即可；并且考虑到应用程序查看订单的一般需求，将本属于书籍的信息 price 在订单这里存储，使得 query_order 时不需要再去书籍文档查询。

2. 功能介绍 / APIs

注：简介部分带 * 表示该功能为新添加的后 40% 部分。我们添加了**一个新模块与六个新 API** 来实现额外的功能。

User / Auth 模块

Route	Args	简介	后端逻辑 / 数据库部分实现
/login	user_id, password, terminal	用户登出	先将 password 与 db 中的比对，成功后 encode 新的 token，在数据库中更新 token 和 terminal。其中 token 是每次登录产生的标识符，terminal 是设备码。
/logout	user_id, token	用户登出	检查 token 是否匹配。若匹配，更新 token（使得之前的 token 无效）作为登出的实现。
/register	user_id, password	用户注册	若 user_id 不重复（不抛出 DuplicateKeyError），向 user 文档中添加新用户信息。其中 balance 置为 0，token 和 terminal 均实时生成。
/unregister	user_id, password	用户注销	若密码匹配，在 user 文档中删除该用户。
/password	user_id, old_password, new_password	修改密码	检查 old_password 是否匹配。若匹配，在数据库中将 password 设置为 new_password。

Buyer 模块

Route	Args	简介	后端逻辑 / 数据库部分实现
/new_order	user_id, store_id, books	新建订单	若 order_id 不重复（不抛出 DuplicateKeyError），向 order 文档中添加新订单信息。其中每本书的 price 需要向书籍文档查询，同时在**创建订单时即扣除库存量**（stock_level），更新书籍数据库。然后在创建订单时将 total_price 计算出来，存入最后的订单中。订单刚创建时为 **unpaid 状态。**
/payment	user_id, password, order_id	订单付款	订单只有**在 unpaid 状态才能被付款。**订单付款时，首先进行密码检查，然后查询对应订单的 total_price 字段，将 buyer 的 balance 扣除掉该字段的值。注意此时暂时先不给 seller 增加对应金钱，而是**将订单先置为 paid 状态**，等待后续流程走完。当然，也不会删除订单。 注意，在付款前我们会查看订单的 timestamp 来确定订单是否 expired。如果订单过期，则本次 payment 无效，订单自动取消。
/add_funds	user_id, password, add_value	用户充值	查询到对应的用户数据段，若密码匹配，给其 balance 字段添加对应值。
/mark_order_received	user_id, password, order_id	订单签收*	订单只有**在 delivered 状态才能被签收。**订单签收时，首先进行密码检查，若情况无误，**即可将订单置为 finished 状态。**注意我们这里也不删除该订单，因为用户可能有查询历史已完成订单的需求。签收成功后，给 seller 增加对应金钱。
/cancel_order	user_id, password, order_id	取消订单*	订单**在非 canceled 与 finished 状态都能被取消。**订单取消时，首先进行密码检查，若情况无误，则**将订单置为 canceled 状态**，退还所有书籍的库存量（stock_level），然后退还用户的金额（给买家的 balance 字段加上 total_price）。
/query_all_orders	user_id, password	查询自己全部历史订单*	需要密码检查。使用 user_id 作为 key 来对订单数据库进行查找。会返回所有状态下的自己的历史订单。

Route	Args	简介	后端逻辑 / 数据库部分实现
/query_one_order	user_id, password, order_id	查询自己某一历史订单*	需要密码检查。使用 order_id 作为 key 来对订单数据库进行查找。返回特定历史订单。

Seller 模块

Route	Args	简介	后端逻辑 / 数据库部分实现
/create_store	user_id, store_id	创建商店	若 store_id 不重复（不抛出 DuplicateKeyError），向 store 文档中添加一条新数据。
/add_book	user_id, store_id, book_info, stock_level	添加书本	向指定商店添加书本。注意这里的 book_info 在存储到数据库中会被**展开**，也就是说 book_info 中的 title, author 等在文档中直接作为第一层的 key 值。
/add_stock_level	user_id, store_id, book_id, add_stock_level	添加书本库存	用户给商店内的指定书本添加库存。以 (store_id, book_id) 为 key，update 对应书本的 stock_level。
/mark_order_shipped	store_id, order_id	订单发货*	订单只有**在 paid 状态才能发货。**会检查订单购买时的商店是否与 store_id 匹配（同一家店）。若情况无误，会将订单**置为 delivered 状态。**

Search 模块

由于查书功能在我们组内讨论后，一致认为不需要与账户系统耦合（也就是说，游客也可以查书），因此它需要单独被放在一个模块中。我们称其为 Search 模块。

Route	Args	简介	后端逻辑 / 数据库部分实现
/query_book	restriction	查询书本	参数化的查询书本方式。参数 restriction 是一个 dict，表示查询时的约束。关于此 dict 的详细规范见下面。

query_book restriction 规范

关于查询书本可以使用的约束，基本上 book info 中的所有键值都可以。并且考虑到实际的查询需求，我们给 title 这一键值增加了模糊查询的功能。

```
The keys of this dict can be:
    id
    store_id
    title
    author
    publisher
    original_title
    translator
    pub_year
    pages
    price
    currency_unit
    binding
    isbn
    author_intro
    book_intro
    content
```

以上内容对应 book_info 中的相应内容。这里我们使用参数化的查询 API，也就是说，如果你指定如下的一次查询：

```
query_book(title="美丽心灵", author="xxx")
```

它会返回所有标题为 `美丽心灵` 且作者为 `xxx` 的书籍。

默认情况下这种查询行为就是全站查询。为了实现店铺内查询，需要在查询时指定 `store_id`：

```
query_book(store_id="store1", title="美丽心灵", author="xxx")
```

以及最后，关于 title 的模糊匹配功能，我们预留了一个特殊的 dict key：`title_keyword`。当你指定

```
query_book(title_keyword="美丽", author="xxx")
```

它会返回所有书名包含 `美丽` 且作者为 `xxx` 的书籍。

3. 测试介绍

正确性测试

全部的正确性测试都位于 `fe/test` 之下。为了提高测试覆盖率，我们在原有的一些测试上也新加了一些测试函数，同时对新的 API 进行了相对完整的测试。

每个测试点的意义基本可以通过它的函数名来推断出。例如 `test_ok` 表示该功能模块一个合理的运行过程，通常返回值是 200；而 `test_error_non_exist_user_id` 表示该功能模块在 `user_id` 不存在情况下的处理，通常返回值是一个错误码。

对于新功能，我们增加了以下测试：

测试文件名	测试内容
test_cancel_order.py	测试取消订单的相关内容
test_order_state.py	测试订单在整个购买流程的状态变化
test_query_order.py	测试查询订单
test_query_book.py	测试查询书本
test_ship_and_receive_order.py	测试收发订单

同时对于之前的测试点，我们也着重 improve 了一些测试：

测试文件名	修改内容
test_payment.py	之前的测试点并没有测试金额是否能正确在账户之间流通，我们添加了相关测试；此外，我们还添加了一个用户付款多个订单的测试。
test_order_state.py	除了正常的 bench 测试以外，我们还添加了 query_order 的 bench 测试和 query_books 的 bench 测试。

性能测试（Bench）

性能测试主要包含原来的多线程 bench 以及我们添加的关于查询订单、查询书本的 bench。**如果想快速跑完所有正确性测试，请将此部分测试参数调小或者删掉！**

- 多线程 bench performance

  conf 如下：

  ```
  Book_Num_Per_Store = 2000
  Store_Num_Per_User = 2
  Seller_Num = 2
  Buyer_Num = 10
  Session = 1
  Request_Per_Session = 1000
  Default_Stock_Level = 1000000
  Default_User_Funds = 10000000
  Data_Batch_Size = 100
  Use_Large_DB = False
  ```

  运行表现是 116s。

- 查询订单 bench performance

查询次数	有无索引	时间（s）
500	无	59.60
500	有	57.27

- 查询书本 bench performance

查询次数	有无索引	时间（s）
1000	无	280.20
1000	有	264.50

由此也可以窥见索引的加速效果。

测试表现

我们的代码测试覆盖率能达到 92% 以上。对于有些部分（比如 MongoDB 错误和 Python 错误），测试可能覆盖不到而且我们认为也没必要覆盖。以下是一份报告。

```
Name                                    Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------------------
be/__init__.py                              0      0      0      0   100%
be/app.py                                   3      3      2      0     0%
be/model/__init__.py                        0      0      0      0   100%
be/model/buyer.py                         168     24     78      4    85%
be/model/error.py                          33      2      0      0    94%
be/model/mongo_manager.py                  54      1      0      0    98%
be/model/search.py                         26      5     10      1    78%
be/model/seller.py                         75     24     32      2    68%
be/model/user.py                          144     46     40      4    65%
be/model/utils.py                           6      0      2      0   100%
be/serve.py                                38      1      2      1    95%
be/view/__init__.py                         0      0      0      0   100%
be/view/auth.py                            42      0      0      0   100%
be/view/buyer.py                           65      0      2      0   100%
be/view/search.py                          10      0      0      0   100%
be/view/seller.py                          38      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                         62      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/search.py                        10      0      0      0   100%
fe/access/seller.py                        38      0      0      0   100%
fe/bench/__init__.py                        0      0      0      0   100%
fe/bench/query_book_bench.py               18      0      2      0   100%
fe/bench/query_order_bench.py              36      0      6      0   100%
fe/bench/run.py                            37      3     14      3    88%
fe/bench/session.py                        49      0     12      1    98%
```

```
fe/bench/workload.py                      147      1     22      2    98%
fe/conf.py                                 13      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                  26      0      0      0   100%
fe/test/test_add_stock_level.py            39      0     10      0   100%
fe/test/test_bench.py                      16      6      0      0    62%
fe/test/test_cancel_order.py               92      2      8      2    96%
fe/test/test_create_store.py               25      0      0      0   100%
fe/test/test_login.py                      28      0      0      0   100%
fe/test/test_new_order.py                  46      0      2      0   100%
fe/test/test_order_state.py                60      1      4      1    97%
fe/test/test_password.py                   33      0      0      0   100%
fe/test/test_payment.py                   110      2      8      2    97%
fe/test/test_query_book.py                 50      0     14      1    98%
fe/test/test_query_order.py                77      1      8      1    98%
fe/test/test_register.py                   31      0      0      0   100%
fe/test/test_ship_and_receive_order.py     95      1      4      1    98%
-------------------------------------------------------------------------
TOTAL                                    2058    124    322     28    92%
```

4. 索引设计

索引可以大大加速查询速度，但是在相应字段修改时，索引反而需要额外的维护成本。

本次作业中大部分查询需求都是通过 id（user_id, order_id, store_id, book_id）来查找的。由于我们将其作为了 mongodb 中的 **_id**，它本身就带有索引。

除了这部分外，可以总结出本次作业需求主要有以下两部分其它查询需求：

- API query_all_orders。此接口需要以 user_id 为 key 对 order 文档进行查询。我们注意到订单的 user_id 在 app 运作时是不会被修改的，因此我们直接对其创建索引。加速效果可以对比上述性能测试中 query order bench 中的数据。
- API query_books。此接口需要以 book_info 中出现的所有 key 值作为键值对 book 文档进行查询。
    - 由于 book 文档基本不会发生改变，因此我们可以尽量对我们可能查询到的 key 都加上索引。注意在实践中，有一些 columns 的 value 太大，无法建立 hash index。这类 key 值包括 `author_intro`，`book_intro`，`content`。
    - 此外，注意到我们对 title 有模糊查询的需求，所以我们对 title 这个 key 值可以建立全文索引（`create_index(type="text")`）。但经过查看发现，数据中的书名大多是中文，而 mongodb 对于中文文本的分词支持不是很好，因此实际实现中我们并没有采用

      ```
      {"$text": {"$search": kwd}}
      ```

      而是直接采用

      ```
      {"$title": {"$regex": kwd}}
      ```

5. 代码润色部分

后端结构设计

将原来的基于 sqlite 的 demo 转换到 mongodb 上实现后，整理 & Refactor 了一下代码结构。

```
- mongo_manager.py
- user.py
- buyer.py
- seller.py
- search.py
```

`mongo_manager.py` 负责 MongoDB 数据库的连接、索引建立、初始化，以及一些全局使用的 API 等。采用 `global` 关键字保证了单例模式的实现，同时向外尽量暴露最小的接口，使得代码结构清晰。

其它文件即为各个模块对应的 API 实现。以 user 模块为例，该文件中包含一个对该模块 API 封装起来的类：

```
class UserAPI:
    """Backend APIs related to user manipulation."""

    token_lifetime: int = 3600  # 3600 second

    @staticmethod
    def register(user_id: str, password: str) -> (int, str):
        ...

    @staticmethod
    def login(user_id: str, password: str, terminal: str) -> (int, str, str):
        ...
```

接口方面基本为 `staticmethod`，返回类型与原来保持一致。

错误码

由于实现了额外功能，我们也增加了一些新的错误码与错误信息。现有的错误码与对应错误信息规范如下：

Error Code	Error Message
401	authorization fail.
511	non exist user id
512	exist user id
513	non exist store id
514	exist store id
515	non exist book id
516	exist book id
517	stock level low, book id
518	<留空备用>
519	not sufficient funds, order id
520	non exist order id
521	exist order id
522	the order state is error
523	the user is not match
524	the store is not match
525	invalid behaviour in query book API

6. 版本管理

本项目采用 git 作为版本管理工具。小组成员间协作采用 Github。

项目地址：https://github.com/SiriusNEO/SJTU-CS3952-Database-System

三人合作开发采用如下模式：

- 成员林超凡：直接在 main 分支下开发，负责基础架构搭建以及 Code Review、代码合并管理。
- 成员潘开森：在分支 pks 下开发，负责 40% 第二部分的开发，开发完毕后被 merge 进主分支。
- 成员翟明舒：在分支 zms 下开发，负责 40% 第三部分的开发，开发完毕后被 merge 进主分支。