Report of STJU DB Bookstore Project 2

0. 成员

注：此作业为个人作业。

姓名	学号	分工
林超凡	520021911042	将第一次作业的成品修改为 ORM，重写了部分测试最后完成了第二次作业。

1. 实验内容与项目结构介绍

实验内容

本次数据库第二次大作业，我主要使用 Python 的 SQL ORM 框架 **SQLAchemy** 对项目后端进行了一次重构。SQL Server 使用 **PostgreSQL**。具体来讲，相较于第一次大作业，主要做了以下工作：

- 重构了大部分后端代码，使用 ORM 的逻辑进行编程。（`be/model/`）
- 重写了部分测试代码来适配新的后端架构。
- 修复了第一次版本的一些 Bug。
- 重新在新的正确性测试与性能测试上测了新后端的 Performance。

实验环境为 WSL2 Ubuntu 20.04。

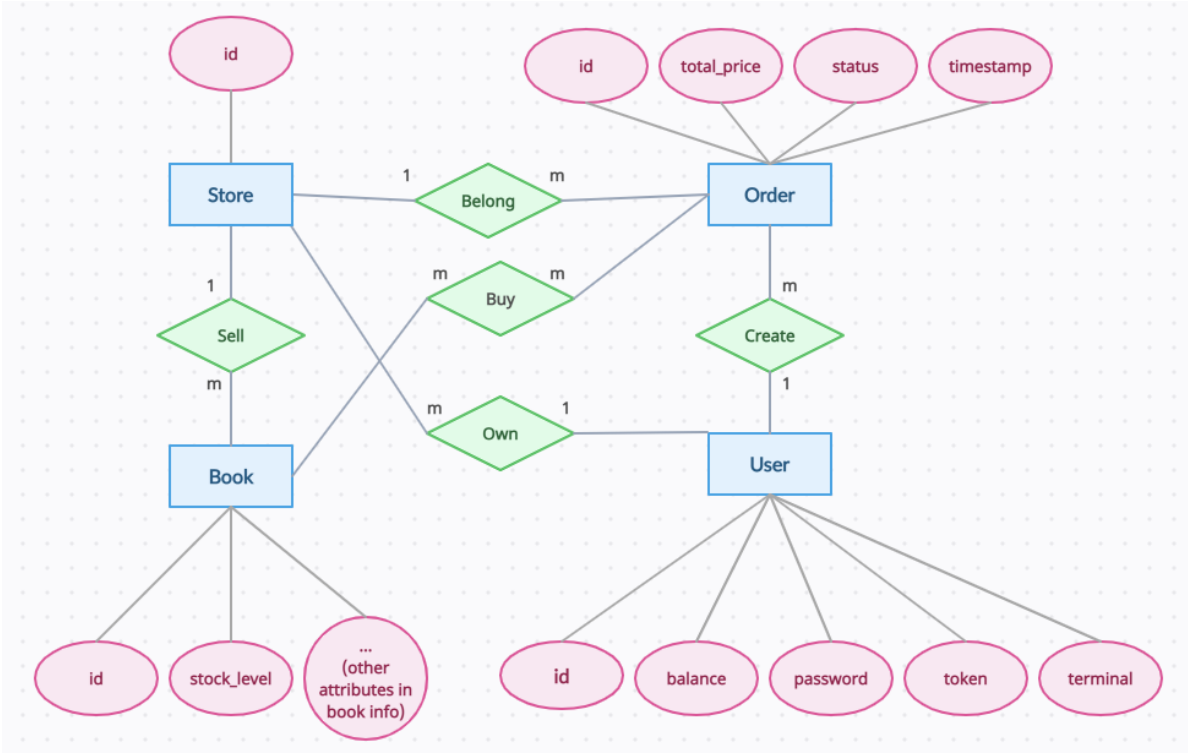
项目结构

```
be/model/
- base.py           # 定义ORM对象与SQL连接处理
- error.py          # 错误码定义
- buyer.py          # buyer 相关API实现
- seller.py         # seller 相关API实现
- search.py         # search 相关API实现
- user.py           # user(auth) 相关API实现
- utils.py          # 一些工具函数实现
```

2. 关系数据库设计

作为一个书店管理系统，主要的 Entity 包括用户、书本、书店与订单这几种。

ER 图



结构设计

根据 ER 图，我们先对每个 Entity 设计一张表：`User`，`Book`，`Order`，`Store`；

对于 Relation：

- 所有 1 对多的关系均在 "多" 的那一侧存储相应 id。例如对于 Store 与 Book 的 1 对多关系，在 Book 处存储一个 store_id。
- 对于 Order 与 Book 间的多对多关系，选择另开一张关系表 `OrderDetail` 进行存储。

此外注意，对于 book info，我们将这个字典的所有键值（title，author，等等）展开并放到最上层，并分别选择合适的数据类型。

ORM 与表结构

基于结构设计，ORM 的类型定义见：`bookstore/be/mode/base.py`.

最终设计出的 SQL 表结构如下所示：

```
 Schema |    Name     | Type  |  Owner
--------+-------------+-------+----------
 public | Book        | table | postgres
 public | Order       | table | postgres
 public | OrderDetail | table | postgres
 public | Store       | table | postgres
 public | User        | table | postgres
```

Table User

```
                      Table "public.User"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 id       | character varying(256) |           | not null |
 password | character varying(256) |           | not null |
 balance  | integer                |           | not null |
 token    | character varying(512) |           |          |
 terminal | character varying(512) |           | not null |
Indexes:
    "User_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE ""Order"" CONSTRAINT "Order_buyer_fkey" FOREIGN KEY (buyer) REFERENCES
"User"(id) ON DELETE SET NULL
    TABLE ""Store"" CONSTRAINT "Store_owner_fkey" FOREIGN KEY (owner) REFERENCES
"User"(id)
```

Table Book

```
                         Table "public.Book"
     Column     |          Type          | Collation | Nullable | Default
----------------+------------------------+-----------+----------+---------
 id             | character varying(256) |           | not null |
 store_id       | character varying(256) |           | not null |
 stock_level    | integer                |           | not null |
 title          | text                   |           | not null |
 author         | text                   |           | not null |
 publisher      | text                   |           | not null |
 original_title | text                   |           | not null |
 translator     | text                   |           | not null |
 pub_year       | text                   |           | not null |
 pages          | integer                |           | not null |
 price          | integer                |           | not null |
 binding        | text                   |           | not null |
 isbn           | text                   |           | not null |
 currency_unit  | text                   |           | not null |
 tags           | text                   |           | not null |
 pictures       | text                   |           | not null |
 author_intro   | text                   |           | not null |
 book_intro     | text                   |           | not null |
 content        | text                   |           | not null |
 Indexes:
    "Book_pkey" PRIMARY KEY, btree (id, store_id)
    "ix_Book_author" btree (author)
    "ix_Book_binding" btree (binding)
    "ix_Book_currency_unit" btree (currency_unit)
    "ix_Book_isbn" btree (isbn)
    "ix_Book_original_title" btree (original_title)
    "ix_Book_pages" btree (pages)
    "ix_Book_price" btree (price)
    "ix_Book_pub_year" btree (pub_year)
    "ix_Book_publisher" btree (publisher)
    "ix_Book_title" btree (title)
    "ix_Book_translator" btree (translator)
Foreign-key constraints:
    "Book_store_id_fkey" FOREIGN KEY (store_id) REFERENCES "Store"(id)
```

Table Order

```
                        Table "public.Order"
   Column    |          Type          | Collation | Nullable | Default
-------------+------------------------+-----------+----------+---------
 id          | character varying(256) |           | not null |
 buyer       | character varying(256) |           |          |
 store_id    | character varying(256) |           | not null |
 total_price | integer                |           | not null |
 status      | status                 |           | not null |
 timestamp   | double precision       |           | not null |
Indexes:
    "Order_pkey" PRIMARY KEY, btree (id)
    "ix_Order_store_id" btree (store_id)
Foreign-key constraints:
    "Order_buyer_fkey" FOREIGN KEY (buyer) REFERENCES "User"(id) ON DELETE SET
NULL
    "Order_store_id_fkey" FOREIGN KEY (store_id) REFERENCES "Store"(id)
Referenced by:
    TABLE ""OrderDetail"" CONSTRAINT "OrderDetail_order_id_fkey" FOREIGN KEY
(order_id) REFERENCES "Order"(id) ON DELETE CASCADE
```

Table Store

```
                       Table "public.Store"
 Column |          Type          | Collation | Nullable | Default
--------+------------------------+-----------+----------+---------
 id     | character varying(256) |           | not null |
 owner  | character varying(256) |           | not null |
Indexes:
    "Store_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "Store_owner_fkey" FOREIGN KEY (owner) REFERENCES "User"(id)
Referenced by:
    TABLE ""Book"" CONSTRAINT "Book_store_id_fkey" FOREIGN KEY (store_id)
REFERENCES "Store"(id)
    TABLE ""Order"" CONSTRAINT "Order_store_id_fkey" FOREIGN KEY (store_id)
REFERENCES "Store"(id)
```

Table OrderDetail

```
                    Table "public.OrderDetail"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 order_id | character varying(256) |           | not null |
 book_id  | character varying(256) |           | not null |
 count    | integer                |           | not null |
 price    | integer                |           | not null |
Indexes:
    "OrderDetail_pkey" PRIMARY KEY, btree (order_id, book_id)
Foreign-key constraints:
    "OrderDetail_order_id_fkey" FOREIGN KEY (order_id) REFERENCES "Order"(id) ON
DELETE CASCADE
```

备注

- 大部份表都使用相应的 `id` 作为唯一主键，除了 `Book` 使用联合主键，这是因为在需求上每个店卖的书可能是不同的，书无法脱离店铺存在。
- 设计上所有 1 对多的关系转化为的 id 都加上了外键（Foreign Key）约束，除了 `OrderDetail` 的 `book_id` 属性（这是因为 Book 使用的是 `book_id` + `store_id` 联合主键，单个 `book_id` 无法满足外键的唯一约束条件）。
- 外键有两个地方设置了 ondelete action。考虑到用户可以注销（unregister），而 `Order` 外键引用了 `user_id`，因此需要考虑用户注销时对应的订单如何处理。
  - 根据实际需求我们认为用户注销后也应该保存它的订单，因此选择将订单的 `buyer` 属性设置为 `SET NULL`（即用户注销时将对应订单的购买者置为 NULL）。
  - 同时，`Order` 和 `OrderDetail` 我们采用 `CASCADE` 关系，即订单删除后对应的详单也全部删除。这是因为 `OrderDetail` 只不过是订单与书本多对多关系的储存方式，删除是合理的。
- 键值的索引部分将在第五部分详细讲解。

3. 功能介绍 / APIs

API 部分相对第一次大作业的版本改动不大。主要参考第一次文档。

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。

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

4. 测试介绍

测试部分相对第一次大作业的版本改动不大（只有一些为了迁移到 PostgreSQL 而修改的逻辑）。

在第一次大作业的正确性测试与 Bench 上重新测试了性能表现。

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）

重新在基于 SQL ORM 的新后端上进行了性能测试。

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：

  ```
  time_new_order=75.18s, time_payment=56.56s
  ```

- 查询订单 bench performance

查询次数	有无索引	时间（s）
500	无	62.60
500	有	58.91

- 查询书本 bench performance

查询次数	有无索引	时间（s）
1000	无	81.24
1000	有	76.01

由此也可以窥见索引的加速效果。

此外，与第一次大作业的效率相比，使用 ORM 重构过的第二次大作业后端效率有明显的提升。

测试表现

重新在基于 SQL ORM 的新后端上进行了正确性测试，导出相应的 coverage report。

我们的代码测试覆盖率能达到 90% 以上。对于有些部分（比如 SQL 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/base.py                             70      3      0      0    96%
be/model/buyer.py                           237     48     80      4    80%
be/model/error.py                            33      2      0      0    94%
be/model/search.py                           32      8     10      0    76%
be/model/seller.py                          106     34     34      2    67%
be/model/user.py                            174     59     38      3    64%
be/model/utils.py                            36      4      8      0    91%
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                              39      4     16      4    85%
```

```
fe/bench/session.py                          49      0     12      3    95%
fe/bench/workload.py                        146      3     22      4    96%
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                   58      0     18      1    99%
fe/test/test_query_order.py                  69      1      4      1    97%
fe/test/test_register.py                     31      0      0      0   100%
fe/test/test_ship_and_receive_order.py       95      1      4      1    98%
-------------------------------------------------------------------------
TOTAL                                      2241    183    332     31    90%
```

5. 索引与事务设计

索引

我们需要对频繁查询且不易修改的字段加上索引。根据分析，大部分查询都是通过相应表的 `id` 进行，而由于 `id` 本身是表的 Primary Key，因此自身带有索引，无需特别指明。

除此以外，还有两处需要我们进行索引考量：

- `query_all_orders`。此接口需要以 `buyer` 为 key 对 order 文档进行查询，注意到订单的 `buyer` 字段不会被修改，因此我们也对其添加索引。
- `query_books`。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。不过在实践上有些 columns 字段太大，无法添加索引（PostgreSQL 会报错），因此选择部分字段进行添加索引。

最终索引的添加效果为：

- 所有 Primary Keys。
- 订单部分的索引。

  ```
  "ix_Order_store_id" btree (store_id)
  ```

- 书本部分的索引。

```
"ix_Book_author" btree (author)
"ix_Book_binding" btree (binding)
"ix_Book_currency_unit" btree (currency_unit)
"ix_Book_isbn" btree (isbn)
"ix_Book_original_title" btree (original_title)
"ix_Book_pages" btree (pages)
"ix_Book_price" btree (price)
"ix_Book_pub_year" btree (pub_year)
"ix_Book_publisher" btree (publisher)
"ix_Book_title" btree (title)
"ix_Book_translator" btree (translator)
```

在第四部分的测试结果中可以看到索引有着显著的加速效果。

事务

在后端逻辑编写中，我们使用 SQLAchemy 提供的 Session 抽象给我们的逻辑加上了事务处理部分。具体来讲，事务处理在实现中包括：

- 对于本次的书店任务，每个 API（注册，新订单，查书）均在一个事务（session）中完成。当所有数据操作完成后，统一在结尾 commit。
- 对于一些特殊的小任务（例如 utils 中查询某些 id 是否存在），它们被封装在一个单独的事务中进行。
- 异常处理上，如若中间发生任何异常，立即 rollback。当然如果本次事务并不包含数据修改，只需要 close 当前 session 即可。

6. 版本管理

本项目采用 git 作为版本管理工具，使用 Github 作为云代码托管服务。

由于本次为单人作业，故并不采用分支进行开发，而是直接在 main 分支 commit。

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