华东师范大学数据科学与工程学院实验报告

课程名称：当代数据管理系统　　**年级：**2021 级　　**上机实践成绩：**

指导教师：周烜　　**姓名：**李睿恩　　**学号：**10215501434

上机实践名称：Bookstore

上机实践编号：2

一、 实验目的

基于 MySQL 数据库，实现一个提供网上购书功能的网站后端。

网站支持书商在其中开商店，购买者可以通过网站购买书籍。

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”的流程。

二、 实验任务

- 实现对应接口的功能。这主要包括：

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、添加书籍信息及描述、增加库存。

- 为项目添加其它功能。这主要包括：

1. 实现后续的流程。如“发货→收货”。
2. 搜索图书功能的开发。

用户可以通过输入关键字，进行参数化的搜索。

搜索范围包括题目、标签、目录、内容。可能是全站搜索，也可能是当前店铺搜索。

如果显示结果较大，需要进行分页。

可以使用全文索引优化查找。

3. 记录订单状态，允许订单查询和取消订单。

用户可以查自己的历史订单，用户也可以取消订单。

取消订单可以由买家主动地取消订单，或者买家下单后，经过一段时间超时仍未付款，订单也会自动取消。

三、 实验环境

本次实验基于 Python 3.7 的环境，且安装了 Flask (2.0.0), Werkzeug (2.0.0), simplejson, lxml, codecov, coverage, pre-commit, pytest, PyJWT, requests, pymysql 模块以保证实验的正常进

行。

在进行数据库设计与代码编写时，我们使用了 **Github** 作为云上存储文件的平台。我们在 Github 中采取了**版本控制**，在代码编写过程中采用了不同的分支存储不同版本的代码文件，令代码更新与各版本代码保护更加完善。

在本次实验中，我们使用了 **Git Bash, Github Desktop** 作为将本地内容与 Github 线上仓库进行连接的方式。在进行代码测试的过程中，我们使用了 **Git Bash** 进行测试。

四、 实验过程

4.1 数据库设计

在实现数据库功能之前，我们需要解决数据库的设计问题，即了解该数据库存储的数据内容并研究数据存储的方式。

4.1.1 需求分析

我们需要设计一个线上书店。

任何到访此网站的人都可以注册自己的账号，无论其身份是买家还是卖家。

网站支持卖家在网站上开设属于自己的商店，且可以开设一个或多个网上商店。

买家则可以通过网站购买书籍，并且可以为自己的账户充值，在任意商店购买图书。

我们设计的线上书店可以支持“下单”、“付款”、“发货”与“收货”的流程。

具体地说，任何用户都可以注册自己的账号，他们既可以作为买家浏览书店，同时也可以作为卖家在这个书店中创建属于自己的店铺。如果用户暂时不希望浏览该书店，可以登出自己的账号，如果用户希望销毁自己的数据，可以注销自己的账号。

用户登录网站后，可以浏览各个店铺。所有用户都可以购买店铺中存在的书，且可以查看自己的订单，并可以为自己未付款的订单付款。如果一个订单一直都在未付款状态，或者买家自己希望取消订单，则订单会被取消。在该书店中，用户的付款依赖于用户账户中的余额，故用户可以实时查询自己的账户余额并为自己的账号充值。在购买的书籍到货后，用户可以确认收货来完成订单。如果用户想要搜索自己想要的书籍，他们可以通过搜索功能来寻找哪些店铺提供这些书本，搜索方式包括书名、作者、内容、全站搜索与店铺搜索等。

用户在网站中也可以担任卖家的角色。用户可以创建店铺，并管理自己的店铺，为自己的店铺添置新的图书，同时也可以为自己的图书补货。在接收到买家的订单后，作为卖家的用户需要及时处理订单，为买家准备需要的书并发货。

图 1 是我们设计的业务流程图。

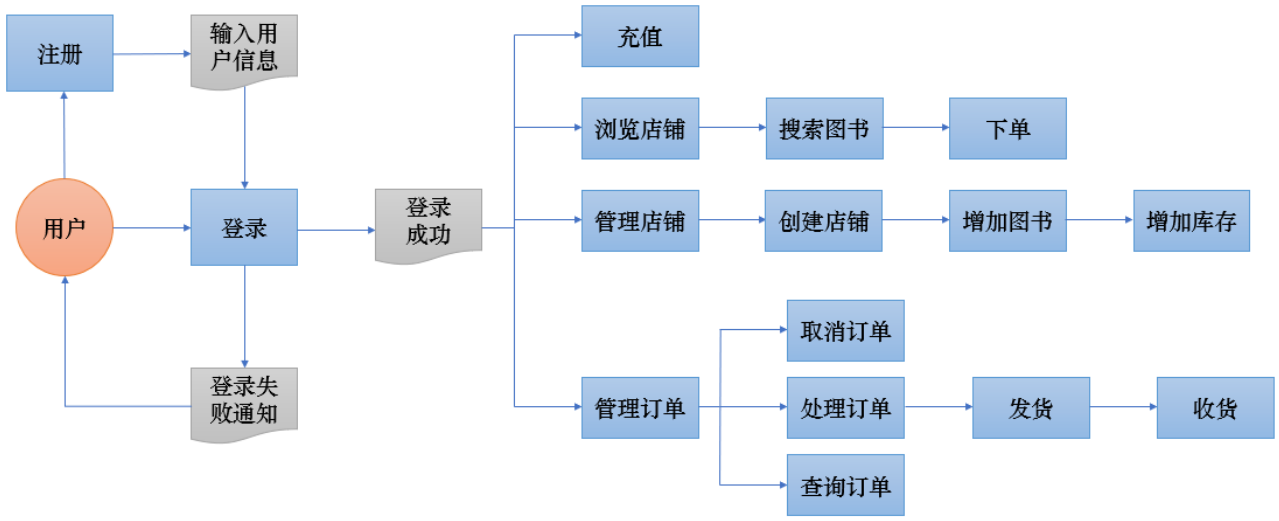


图 1 线上书店的业务流程图

4.1.2 概念结构设计

经过需求分析，我们可以发现在我们的数据库中，“**书本**”、“**店铺**”、“**用户**”、“**订单**”是我们主要设计的对象。

他们的实体内容包括以下内容：

- **用户**：用户 ID、密码、余额、Token 令牌、登录终端、拥有的店铺、拥有的订单 ID.
- **店铺**：店铺 ID、拥有者用户 ID，存储的书本。此处书本的数据包括书本 ID、标题、作者、出版社、书本介绍、内容、库存数量、价格与标签。
- **订单**：订单 ID，买家用户 ID，店铺 ID，订单状态，购买的时间，书本。此处书本的数据包括书本 ID、购买的书本数量与书本价格。
- **书本**：书本 ID、标题、作者、出版社、原标题、译者、出版年份、价格、货币单位、封皮、ISBN、作者简介、书本简介、内容、标签与图片。

在了解了数据存储的内容后，我们需要考量这些数据之间的存储关系。

这些实体之间存在以下联系：

- **用户与店铺的关系**：一个用户可以创建多个店铺，并且可以管理这些店铺，一个店铺只可以拥有一个拥有者。
- **用户与订单的关系**：一个用户可以在店铺中下单而产生订单，一个用户可以有多个订单，但是一个订单只有由一个用户创建。
- **店铺和订单的关系**：一个订单是在一个店铺中产生的，一个店铺中可以拥有由不同用户产生的不同的订单。
- **店铺和书的关系**：一个店铺可以有多种书，并且一本书可以在多个店铺中存在。
- **订单和书的关系**：一个订单可以有多种书，并且一本书可以被多个用户购买从而出现在多个店铺中。

图 2 为我们设计的线上书店的 E-R 图。

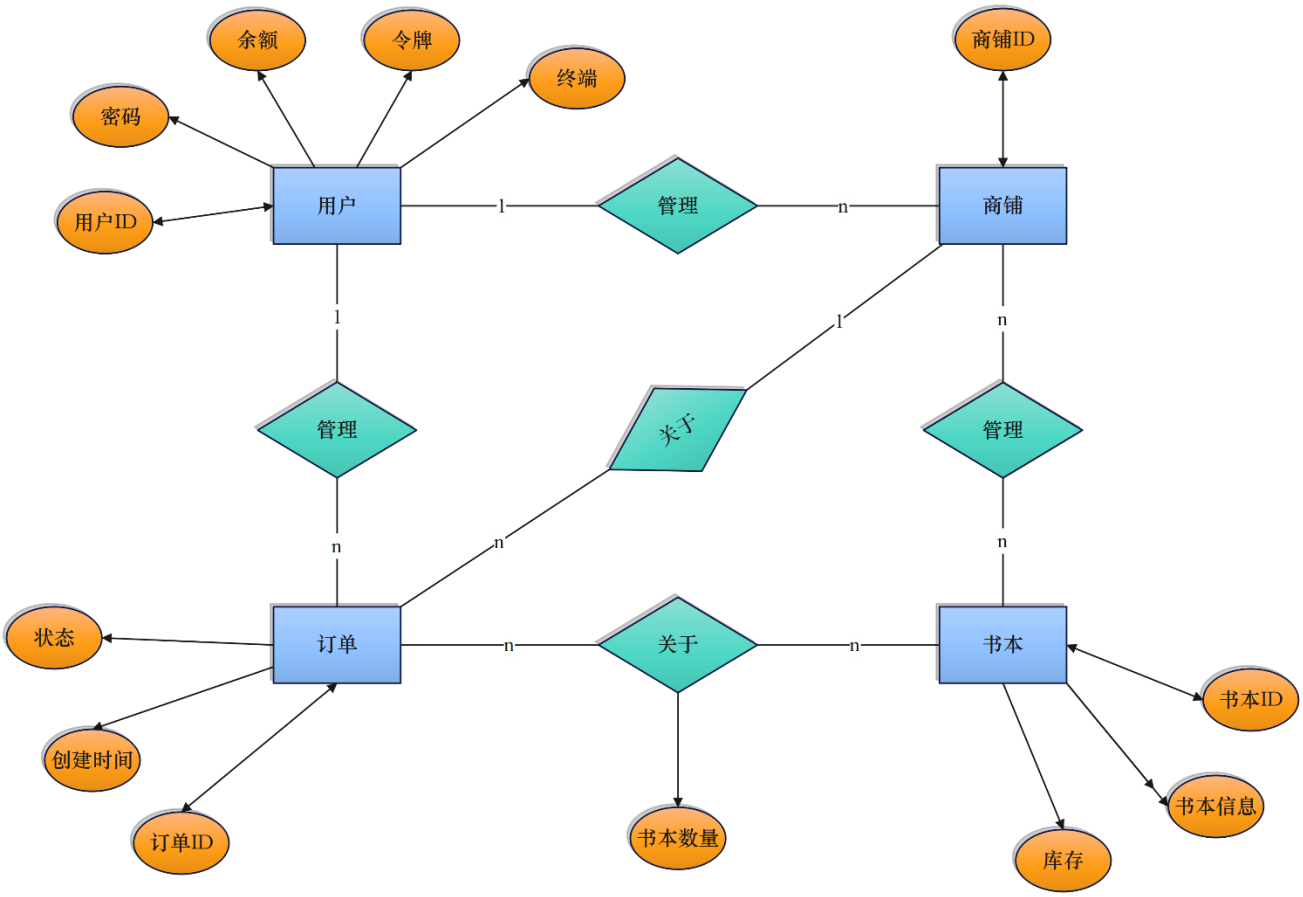


图 2 线上书店的 E-R 图

4.1.3 逻辑结构设计

在上一次的作业中，我们设计了“MongoDB 逻辑结构”，MongoDB 逻辑结构允许了嵌套的存在，具有较高的灵活度，使得个别查询的效率得到的提升。其逻辑结构如图 3 所示。

```
User {u_id, password, balance, token, terminal, stores[store_id], orders[order_id]}

Store {store_id, u_id,
        books[{
                book_id, title, author, publisher, book_intro, content, stock_level, price, tags
              }]}

Order {order_id, u_id, store_id,
        books[{
            book_id, count, price
        }],
        status, time}
```

图 3 MongoDB 逻辑结构

但是，MongoDB 逻辑结构也有不小的缺点。尽管在本次试验中并没有被提及，但在实际工业界使用中，由于数据库可能会出现故障，因此事务处理被认为是一个非常重要的机制，它可以处理数据库可能面对的故障以及故障所导致的数据不一致问题。但是，文档数据库

（如 MongoDB）缺乏有效的事务处理机制，因此它在实际使用时还需要另外考虑方法以规避此问题。另外，文档数据库往往涉及到内容的嵌套，虽然它可能可以提升数据访问的效率，但这也导致了用户如果想要对文档数据库进行有效地访问，需要对文档集的结构描述有一定的理解，这种复杂性的提升实际上对数据库整体的维护并不友好。

因此，关系数据库是另一个经常被使用到的数据库结构。关系数据库不但可以提供有效的事务处理，同时还因为数据普遍以表格的结构化形式存在，并不涉及复杂嵌套，因此易于理解，便于编写业务逻辑代码。由于数据都是存储在简单的表格结构中的，因此复杂查询的构造具有可行性。另外，关系数据库普遍有查询优化器，这使得关系数据库可能可以有更高效的查询效率。

然而，项目原先提供的 SQLite 逻辑结构过于简单，它可能无法让我们设计出更完备的功能，可能以较低的效率进行检索相关的工作。因此，我们在实现 MySQL 语法转换的基础上，**对 SQLite 逻辑结构进行了调整**，并且在创建表格时添加了**索引**，使得后续可以进行更完备的功能开发，并减少了与检索相关的操作时间。我们修改后的逻辑结构被称为“**MySQL 逻辑结构**”。本次实验全程基于此逻辑结构实现。

在这里，我们参考了我们设计的 ER 图来重新设计结构描述，并基于 ER 图进行了一定的调整。MySQL 逻辑结构见图 4.

```
user {u_id, password, balance, token, terminal}

user_store {u_id, store_id}

store {store_id, book_id, title, price, tags, author, book.intro, stock_level}

orders {order_id, book_id, count, price}

new_order_detail {order_id, user_id, store_id, time, status}
```

图 4 MySQL 逻辑结构

4.2 实验过程

4.2.1 理解前后端交互

在对代码进行修改前，我们需要先理解代码的执行顺序与效果。

当用户使用线上书店时，用户首先会先与前端进行交互。前端的代码均被存储在 fe 文件夹中。当用户与前端产生交互后，前端会向后端传输一定的数据，并让后端实现数据的分析与处理。后端的代码均被存储在 be 文件夹中。

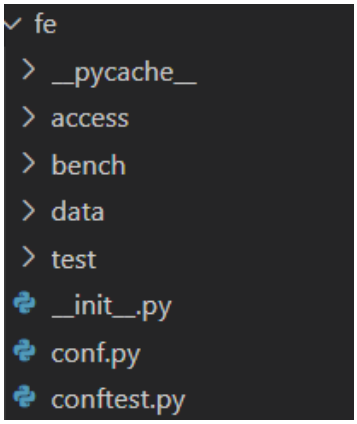


图 5 前端代码文件夹

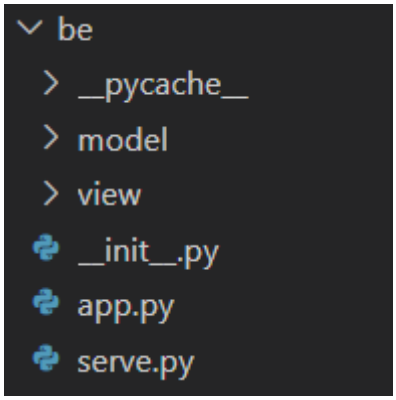


图 6 后端代码文件夹

在进行测试时，后端的主函数 app.py 会先被执行。

```
from be import serve

if __name__ == "__main__":
    serve.be_run()
```

它会执行后端的 serve.py 文件中的 be_run 函数。

```
def be_run():
    this_path = os.path.dirname(__file__)
    parent_path = os.path.dirname(this_path)
    log_file = os.path.join(parent_path, "app.log")
    init_database()
```

这段代码调用了后端中 model 文件夹中的 store.py 文件，从而导致函数 get_db_conn()被运行。通过执行这样的代码，我们便成功连接上了本地计算机的 MySQL 数据库。

```
class Store:
    database: str

    def __init__(self, db_path):
        self.database = os.path.join(db_path, "be.db")
        self.init_tables()

    def get_db_conn(self):
        return pymysql.connect(
            host="127.0.0.1",
            port=3306,
            user="root",
            password="root",
            database="DBMS"
        )
```

```
def init_database(db_path):
    global database_instance
    database_instance = Store(db_path)


def get_db_conn():
    global database_instance
    return database_instance.get_db_conn()
```

连接成功后，数据库就处于等待请求的状态。在这期间，如果我们对本地(127.0.0.1)的5000 端口发送一个特定的 POST 请求，那么我们的代码将解析该请求，并执行相应的操作。图 7 为利用 POSTMAN 发送 POST 请求并获得响应的示例。

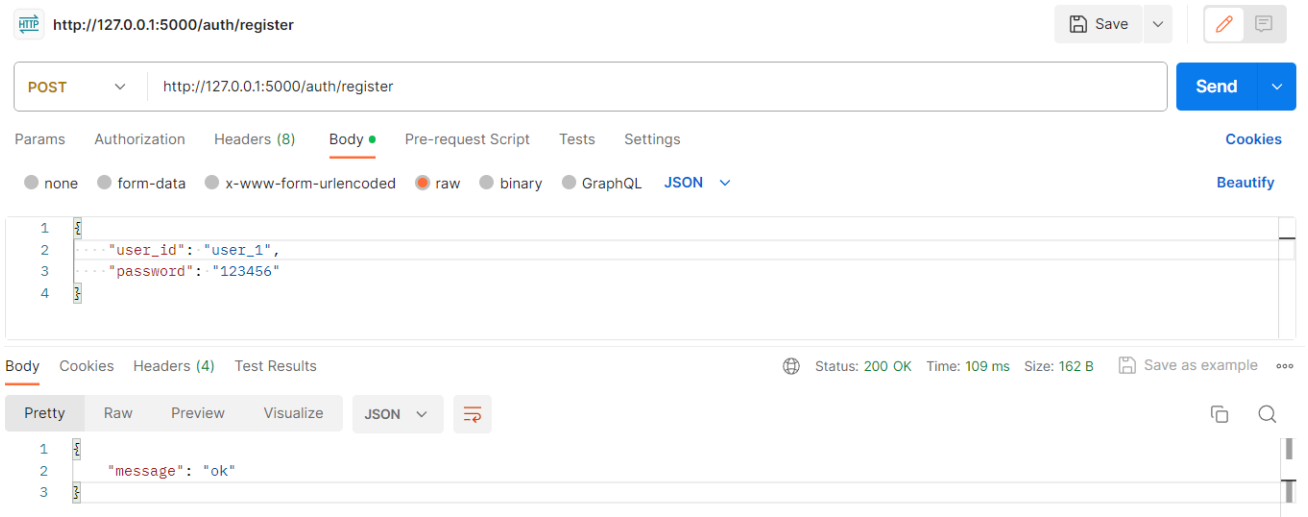


图 7 利用 POSTMAN 实现数据库操作的示例

为了确保数据库的代码编写是正确的，我们会使用一些在前端文件夹的测试代码。这些测试代码的整体运行逻辑的相似的，在这里我们以 test_login.py 为例，分析测试代码的执行顺序与效果。

在 test_login.py 中，我们有 TestLogin 类。在该类中，我们会执行初始化函数 pre_run_initialization，它会利用时间戳生成互不相同的 user_id, password 与 terminal，并调用前端 access 文件夹的 auth.py 文件中的 register 函数。

```
class TestLogin:
    @pytest.fixture(autouse=True)
    def pre_run_initialization(self):
        self.auth = auth.Auth(conf.URL)
        # register a user
        self.user_id = "test_login_{}".format(time.time())
        self.password = "password_" + self.user_id
```

```
        self.terminal = "terminal_" + self.user_id
        assert self.auth.register(self.user_id, self.password) == 200
        yield
```

register 函数会将我们传入的 user_id 与 password 生成为一个 json 格式的数据，并以此生成一个含有/auth/register 关键字的 URL，并以 POST 的方式发送给后端。该 URL 中含有我们生成的 json 格式的数据。

```
    def __init__(self, url_prefix):
        self.url_prefix = urljoin(url_prefix, "auth/")
    def register(self, user_id: str, password: str) -> int:
        json = {"user_id": user_id, "password": password}
        url = urljoin(self.url_prefix, "register")
        r = requests.post(url, json=json)
        return r.status_code
```

后端的 auth.py 在接收到一个含有/auth/register 的 POST 请求后，会开始解析我们发出的请求。它从我们给出的.json 格式的数据中读取 user_id 与 password，并进一步调用了 user.py 中的 User 类，执行 User 类中的 register 函数。

```
@bp_auth.route("/register", methods=["POST"])
def register():
    user_id = request.json.get("user_id", "")
    password = request.json.get("password", "")
    u = user.User()
    code, message = u.register(user_id=user_id, password=password)
    return jsonify({"message": message}), code
```

而只有到了这一步，我们的代码才开始真正执行 MySQL 的相关操作。在这里，我们会试图向数据库中插入一条新的用户数据，代表着我们注册了一位用户的信息。如果插入成功，那么它会返回 200 与“ok”，而如果出现了错误，则它会返回 error_exist_user_id，这是一个表示数据库中已经存在该用户 ID，不可以重复注册的错误代码。

```
    def register(self, user_id: str, password: str):
        try:
            terminal = "terminal_{}".format(str(time.time()))
            token = jwt_encode(user_id, terminal)
            self.cursor = self.conn.cursor()
            # self.cursor.execute("USE DBMS;")
            self.cursor.execute(
                "INSERT into user(user_id, password, balance, token,
terminal) "
                "VALUES (%s, %s, %s, %s, %s);",
                (user_id, password, 0, token, terminal)
```

```
            )
            self.conn.commit()

        except pymysql.Error:
            return error.error_exist_user_id(user_id)
        return 200, "ok"
```

error_exist_user_id 是一个被存放在后端 error.py 文件中的错误代码。在该文件中，我们设置了多个错误代码与返回错误代码和错误信息的函数。

```
error_code = {
    512: "exist user id {}",
}
def error_exist_user_id(user_id):
    return 512, error_code[512].format(user_id)
```

上述的 register 函数返回的数值会通过层层返回，最终返回到 test_login.py 文件中。在 test_login.py 中，assert 语句表示“断言”，如果 assert 描述的两个变量的关系不符合预期，那么代码就会报错，这也说明我们的代码编写存在错误或漏洞。如代码中的 assert self.auth.register(self.user_id, self.password) == 200 意味着，如果上述的 register 函数返回的数值是 200，则说明我们的 login 函数代码编写没有出现问题，而一旦其返回值不是 200，就说明我们的 login 函数代码编写出现了问题。

在注册完账号后，test_login.py 文件会再调用其下的 test_ok, test_error_user_id 与 test_error_password 的函数。这些函数分别用于测试“正确登录”、“登录时输错了用户 ID”、“登录时输错了密码”三种情况。比如在 test_ok 中，如果其在应该正常登录时没有返回 200，就说明测试没有通过，而如果其返回了 200，则说明我们登录代码编写正确。

```
   def test_ok(self):
        code, token = self.auth.login(self.user_id, self.password,
self.terminal)
        assert code == 200
```

在本次实验中，所有的测试文件均以类似的思路进行。因此，我们后续的编写也会基于这样的测试流程进行考量。

4.2.2 理解代码功能

在我们正式进行修改前，我们还需要了解前后端各.py 文件的功能。

1. fe 文件夹

该文件夹中的文件主要负责前端工作。

test 文件夹中的文件主要负责测试代码的正确性。其中的文件大多最终会向后端发送数据，等待后端返回一个数值，并检验后端返回的数值是否符合预期。这一点在 4.2.1 中已经有所体现。

bench 文件夹中的文件主要负责测试数据库的吞吐量。这里的代码与数据库本身并无过

大关系。

access 文件夹中的文件主要实现了前后端的交互。当 test 中的测试文件生成了一些数据后，测试文件就会执行 access 文件夹中的文件的代码，从而将数据交由后端处理。

2. be 文件夹

该文件夹中的文件主要负责后端工作。

view 文件夹中的文件主要是路由，他们会解析来自前端的 URL 发送的 POST 请求。

model 文件夹中的文件是真正地与 MySQL 数据库进行交互，因此他们会是之后修改的主要对象。

与 MySQL 有明显关联的函数以及他们实现的功能如下：

- store.init_tables: 连接数据库并创建表格，对数据库进行初始化。
- db_conn.__init__: 连接数据库。
- db_conn.user_id_exist: 检测用户 ID 是否存在。
- db_conn.book_id_exist: 检测书本 ID 是否在某个店铺中存在。
- db_conn.store_id_exist: 检测商店 ID 是否存在。
- user.register: 用户注册账号。
- user.check_token: 检查用户的令牌。
- user.check_password: 检查用户的密码。
- user.login: 用户登录。
- user.logout: 用户登出。
- user.unregister: 用户注销。
- user.change_password: 用户更改密码。
- buyer.new_order: 买家下订单购买书本。
- buyer.payment: 买家为一笔订单进行支付操作。
- buyer.add_funds: 买家为账户充值。
- seller.add_book: 卖家为店铺增加新书。
- seller.add_stock_level: 卖家为店铺中某本书增加库存。
- seller.create_store: 卖家创建店铺。

这些函数会成为我们关注的重点。

4.2.3 V1.0 – V1.2——语法转换

如果我们不能基于 MySQL 实现最为基础的功能，那么我们更加无法实现更进阶的功能。因此，我们在初期的工作是**使用 MySQL 逻辑结构更新关系数据库的结构描述，将 SQLite 语法转化为 MySQL 语法**。

前端代码由于与数据库层面实际无关，因此大部分前端代码没有被修改。

在后端中，位于 model 文件夹中的.py 文件基本都需要重写。在这里，我们将阐释重写的原则，并为每一个原则进行举例。

1. 在基于 SQLite 逻辑结构所提供的代码中，所有的“?”均需要被转化为“%s”。SQLite 中的“?”与 MySQL 的“%s”基本上有着相同的功能，他们都是占位符。在这里，我们只需要将代

码中的“?”直接替换为“%s”即可。如 user.py 中的 check_password 函数，这里只需要作占位符的替换即可。

```
    def check_password(self, user_id: str, password: str) -> (int, str):
        self.cursor = self.conn.cursor()
        self.cursor.execute(
            "SELECT password from user where user_id = %s;", (user_id,)
        )
        row = self.cursor.fetchone()
        if row is None:
            return error.error_authorization_fail()

        if password != row[0]:
            return error.error_authorization_fail()

        return 200, "ok"
```

2. 所有的 sqlite.Error 均被转化为 pymysql.Error。在 MySQL 中，与 SQLite 中的 sqlite.Error 相对应的即 pymysql.Error，在这里，我们只需要将代码中所有的 sqlite.Error 改为 pymysql.Error 即可。如 user.py 的 register 函数，它的 sqlite.Error 就需要被改为 pymysql.Error，通过这样的修改，如果我们的代码存在 MySQL 层面的错误，我们可以很快定位到错误。

```
    def register(self, user_id: str, password: str):
        try:
            terminal = "terminal_{}".format(str(time.time()))
            token = jwt_encode(user_id, terminal)
            self.cursor = self.conn.cursor()
            self.cursor.execute(
                "INSERT into user(user_id, password, balance, token,
terminal) "
                "VALUES (%s, %s, %s, %s, %s);",
                (user_id, password, 0, token, terminal)
            )
            self.conn.commit()

        except pymysql.Error:
            return error.error_exist_user_id(user_id)
        return 200, "ok"
```

3. 所有的对 MySQL 的操作，都必须基于游标（cursor）进行。在 SQLite 的代码中，我们可以直接使用 self.conn 来执行一句数据库的操作。但在 MySQL 中，我们必须使用 self.conn.cursor()来进行数据库的操作。因此，我们需要在所有数据库操作前添加一行提取游

标的代码。如 user.py 中的 login 函数，它执行数据库操作之前，必须先使用 self.conn.cursor() 语句提取游标。

```
    def login(self, user_id: str, password: str, terminal: str) -> (int, str,
str):
            self.cursor = self.conn.cursor()
            self.cursor.execute(
                "UPDATE user set token = %s , terminal = %s where user_id
= %s;",(token, terminal, user_id)
            )
            self.conn.commit()
```

4. store.py 需要大量重写，因为它涉及了连接数据库并创建表格的数据库初始化的问题。尽管目前我们尚未添加新的功能，但如果我们在需要添加新功能的时候再重新设计表格的结构描述，这会是一个不小的工作量，因此我们在 V1.0 – V1.2 版本中就采用了新的结构描述。在这里，我们在创建表格时大量采用了**键**与**索引**，对表格之间的关系作出明确限制，并且使用索引增加了检索相关操作的效率。**我们会在 4.2.5 中具体阐释这样设计的原因**。

```
def init_tables(self):
        try:
            conn = self.get_db_conn()
            cur = conn.cursor()
            cur.execute("USE DBMS;")
            cur.execute(
                "CREATE TABLE IF NOT EXISTS user ("
                "user_id VARCHAR(300) PRIMARY KEY, password VARCHAR(300) NOT
NULL, "
                "balance INTEGER NOT NULL, token VARCHAR(500), terminal
VARCHAR(500), "
                "INDEX index_user (user_id));"
            )

            cur.execute(
                "CREATE TABLE IF NOT EXISTS user_store ("
                "user_id VARCHAR(300), store_id VARCHAR(300) PRIMARY KEY,"
                "FOREIGN KEY (user_id) REFERENCES user(user_id),"
                "INDEX index_store (store_id))"
            )

            cur.execute(
                "CREATE TABLE IF NOT EXISTS store ("
                "store_id VARCHAR(300), book_id VARCHAR(300), title
VARCHAR(100), price INTEGER, "
                "tags VARCHAR(100), author VARCHAR(100),"
```

```
                "book_intro VARCHAR(2000),stock_level INTEGER,"
                "PRIMARY KEY (store_id, book_id),"
                "FOREIGN KEY (store_id) REFERENCES user_store(store_id),"
                # 复合索引
                "INDEX index_store_book (store_id, book_id),"
                "FULLTEXT INDEX index_title(title),"
                "FULLTEXT INDEX index_tags(tags),"
                "FULLTEXT INDEX index_author(author),"
                "FULLTEXT INDEX index_book_intro(book_intro))"
            )

            cur.execute(
                "CREATE TABLE IF NOT EXISTS new_order ("
                "order_id VARCHAR(300) PRIMARY KEY , user_id VARCHAR(300),
store_id VARCHAR(300), "
                "time TIMESTAMP, status INTEGER,"
                "FOREIGN KEY (user_id) REFERENCES user(user_id), "
                "FOREIGN KEY (store_id) REFERENCES user_store(store_id),"
                "INDEX index_order (order_id))"
            )

            cur.execute(
                "CREATE TABLE IF NOT EXISTS orders ("
                "order_id VARCHAR(300), book_id VARCHAR(300), count INTEGER,
price INTEGER,"
                "FOREIGN KEY (order_id) REFERENCES new_order(order_id),"
                "PRIMARY KEY (order_id, book_id), "
                "INDEX index_order_book (order_id, book_id))"
            )

            conn.commit()
```

基于上述原则，be 文件夹中的 model 文件夹中的 buyer.py, db_conn.py, seller.py, store.py 与 user.py 均被修改为 MySQL 可接受的语法，并且表格的结构描述也获得了更新。

修改过代码后，我们启动 Git Bash 运行测试文件。经过测试，所有测试文件均以 PASSED 通过了检验。

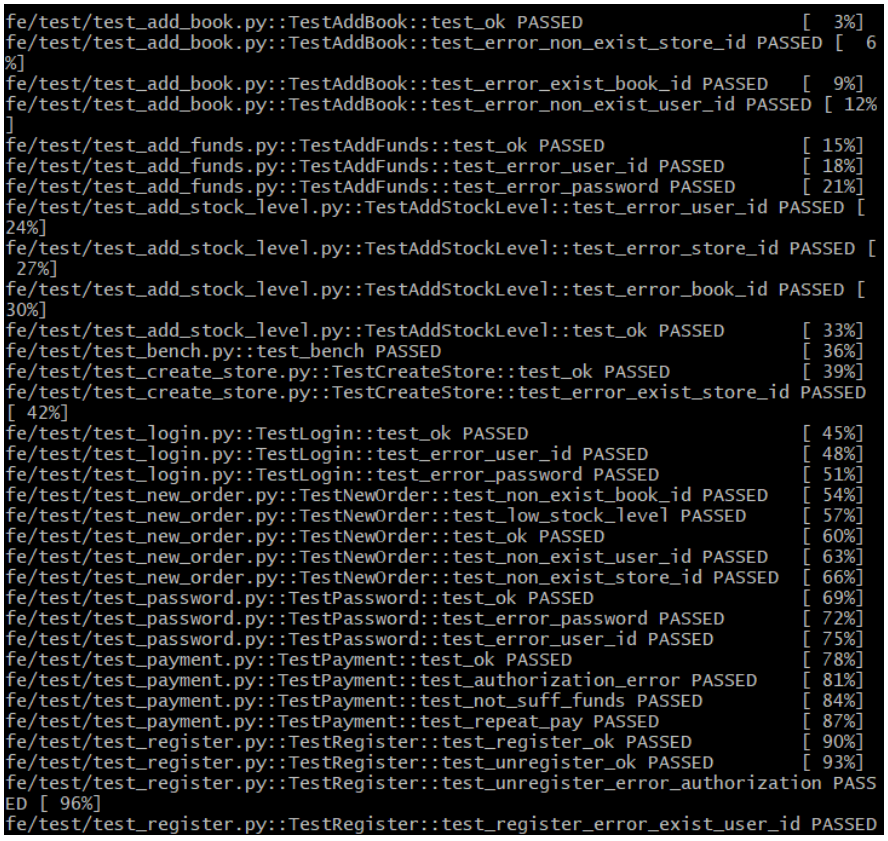


图 8 初步修改后的测试结果

其中，V1.0 是没有任何修改的原始代码文件，V1.1 是经过了大量修改，但少量代码仍存在漏洞的版本，V1.2 是成功实现了基础功能的版本。通过使用**版本控制**，我们在出现严重故障或希望查看前置版本时有了非常有效的手段。

4.2.5 V2.0——功能完备

尽管上述改写后的语法可以通过最基础的测试，但其功能单薄，并且也未能体现出关系数据库的优势。

在第一次实验中，我们了解到使用 MongoDB 逻辑结构这样的文档数据库的逻辑结构，它的数据存储基于键值对，数据没有耦合性，容易扩展，如果善用其独特的“文档嵌入”特性，可能可以增加增删改查的效率。本次实验的 MySQL 逻辑结构则是一种典型的关系数据库的逻辑结构。它以表格的形式组织数据，优点是容易理解，使用方便，格式一旦确定则易于维护，普遍存在优化器，最重要的是我们可以通过 SQL 实现复杂的查询，上述的优点可以让关系数据库结构可能具有更高的访问速度，并便于使用者编写业务逻辑代码。因此，在本次实验中，我们会将数据库的逻辑结构从 MongoDB 逻辑结构与原始代码提供的简单 SQLite 逻辑结构进一步修改为功能完备的 MySQL 逻辑结构。

图 3 描述了原始的 SQLite 逻辑结构。在该逻辑结构中，我们有 user, user_store, store, new_order, new_order_detail 的 5 个文档集。

- user 文档集存储了“用户 ID”、“密码”、“余额”、“令牌”、“终端”。
- user_store 文档集存储了“用户 ID”、“店铺 ID”。

- store 文档集存储了“店铺 ID”、“书本 ID”、“书本信息”、“书本库存”。
- new_order 文档集存储了“订单 ID”、“店铺 ID”、“用户 ID”。
- new_order_detail 文档集存储了“订单 ID”、“书本 ID”、“购买书本数量”、“支付价格”。

我们注意到，这样的设计虽然可以实现基础功能，但如果想要拓展其他功能，那么这样的结构描述实际上会令数据的访问效率较低。为此，我们需要对该逻辑结构进行修改与优化。

图 4 为我们新设计的 MySQL 逻辑结构。在该逻辑结构中，我们有 user, user_store, store, orders, new_order 的 5 个表格，其中 store, orders 和 new_order 经历了重新设计。

- store 表格存储了“店铺 ID”、“书本 ID”、“书本标题”、“书本价格”、“书本标签”、“书本作者”、“书本介绍”、“库存”。
- orders 表格存储了“订单 ID”、“书本 ID”、“书本数量”、“书本价格”。
- new_order 表格存储了“订单 ID”、“用户 ID”、“店铺 ID”、“订单状态”、“时间”。

store 表格被我们设计为一个能更具体地体现店铺与书本关系的表格。其中，title, tags, author, book.intro 被设计为“用户检索图书”的检索方法，因此将这些属性从原始的 book.db 中提取出来并存储在 store 表格中是可以有效增加检索效率的。另外，我们还添加了价格属性，这使得用户创建订单时检索书本价格也更加具有效率。

orders 表格承载了 SQLite 逻辑结构中的 new_order 表格的作用。

new_order 表格详细地描述了一个订单的发起者、相关店铺、创建时间与目前订单状态。我们增加了时间的属性，这是为了在后续额外添加有关订单自动取消的功能。我们也增加了订单状态的属性，它的数值决定了一个订单处于“-1: 已取消”“0: 未付款”、“1: 已付款”、“2: 已发货”、“3: 已收货”。订单的属性将影响发货、收获、取消订单等对订单的操作。

在这里，我们之所以没有将 orders 与 new_order 合并在一张表里以描述订单信息，是因为我们考虑到，一个订单可能同时涉及同一店铺不同名字的图书，如果我们将所有信息杂合在一张表中，这会导致这张表有非常丰富的冗余信息。如果一个订单涉及了同一店铺不同名字的图书，这就会导致同样的 order_id, store_id 出现多次。因此，我们将一些属性分成两张表，使得存储的冗余信息减少了。

详细了解了我们对表格结构描述的优化后，我们就可以基于此进行项目功能的添加。

1. 发货功能

作为卖家，当一笔订单已经处于付款完成状态，则卖家可以择时发货。为此，我们编写了 deliver 函数，它的主体思路是使用 MySQL 中的 SELECT 语句查找订单，确认该订单是否可以发货，然后对订单状态进行更新。当我们判断一笔订单存在，且一笔订单如果已经处于付款完成的状态，则该函数成功执行。

```
    # 发货操作
    def deliver_order(self, order_id: str) -> (int, str):
        try:
            self.cursor = self.conn.cursor()
```

```
            self.cursor.execute("SELECT status FROM new_order where order_id
= %s and status < 3;",
                                (order_id, ))
            row = self.cursor.fetchone()

            if row is None:
                return error.error_invalid_order_id(order_id)

            status = row[0]

            if status == -1:
                return error.error_invalid_order_id(order_id)
            elif status == 0:
                return error.error_order_not_paid(order_id)
            elif status == 2:
                return error.error_order_delivered(order_id)

            self.cursor.execute("UPDATE new_order set status = %s where
order_id = %s;",
                                (2, order_id))
            self.conn.commit()
```

在编写该段函数时，我们在 error.py 中添加了两个错误代码与函数，分别为 520 – order not paid 和 521 – order was delivered，他们分别表示了“订单未付款”与“订单已发货”的两种错误。

```
    520: "order {} not paid",
    521: "order {} was delivered",
def error_order_not_paid(order_id):
    return 520, error_code[520].format(order_id)

def error_order_delivered(order_id):
    return 521, error_code[521].format(order_id)
```

2. 具有分页效果的查询功能

所有线上书店都会允许用户进行查询，从而让用户更快地找到自己希望购买的图书。基于此，我们设计了 search_books 函数，它的主体思路是接受用户的选择与用户给出的关键词，基于 MySQL 的 SELECT 语句查询数据库中的内容。它允许用户选择查询方式（题目、标签、书本简介、作者），输入关键词，以及选择是在全站搜索还是在某一个特定店铺搜索。

在这里，我们还利用 LIMIT 与 OFFSET 语句实现了分页效果。尽管我们只是在对后端进行优化，但如果该项目真正被投入使用，且最终返回给前端的内容使得用户只能在体量非常

大的一页里浏览所有符合条件的内容，这不但不够美观，用户的体验感也会不佳。因此，我们设计了每一页只能显示 20 条数据的分页效果。当用户在进行搜索功能时，前端接收到的 URL 中会有一个数字代表页数，如，收到的页数为 2，每页只能显示 20 条数据，则第二页需要从第 21 条数据开始，展示到第 40 条数据。

```
# 搜索图书
    def search_book(self, keywords, method: str = 'title', store_id: str =
None, page_number: int = 1) -> (int, str):
        PAGE_SIZE = 20
        available_method = ['title', 'tags', 'author', 'book_intro']
        if method not in available_method:
            return error.error_invalid_search_method(method)

        if store_id is not None and not self.store_id_exist(store_id):
            return error.error_non_exist_store_id

        offset = (page_number - 1) * PAGE_SIZE

        # like 是模糊查找
        search = "SELECT book_id FROM store where %s like %s LIMIT %s
OFFSET %s;"
        if store_id is not None:
            search = "SELECT book_id FROM store where %s like %s AND store_id
= %s LIMIT %s OFFSET %s;"

        try:
            self.cursor = self.conn.cursor()
            if store_id is not None:
                self.cursor.execute(search, (method, keywords, store_id,
PAGE_SIZE, offset))
            else:
                self.cursor.execute(search, (method, keywords, PAGE_SIZE,
offset))
```

可以看到在代码中，存在对于 store 是否为空的判断。如果 store 为空，则默认为全站搜索，反之，则为店铺内搜索。如果为店铺内搜索，则需要对 store_id 进行限制。我们还使用了 LIKE 语句实现了部分关键词的模糊搜索。

在编写该段函数时，我们在 error.py 中添加了一个错误代码与函数，为 524 – invalid search method，这表示用户给出的查询范围是不合法的。根据实际需求，用户一般会使用标题，标签，书本简介，作者来搜索图书，所以我们此处只提供这些搜索方式。其余搜索方式均无法通过检验。

```
    524: "invalid search method {}",
```

```
def error_invalid_search_method(method):
    return 524, error_code[524].format(method)
```

3. 取消订单

如果买家不想要再购买一本已经下订单的图书，则该买家可以自行取消订单。基于此，我们设计了 cancel_order 函数，它的主体思路是利用 MySQL 的 SELECT 语句查找符合要求的订单，再利用 UPDATE 语句将店铺中已经减少的书本库存、卖家的账户余额、买家的账户余额增加回下订单之前的状态，并设置订单的状态为-1。

```
    # 取消订单
    def cancel_order(self, user_id, password, order_id) -> (int, str):
        try:
            self.cursor = self.conn.cursor()
            self.cursor.execute("SELECT password FROM user WHERE user_id
= %s;", (user_id, ))
            row = self.cursor.fetchone()

            if row is None:
                return error.error_non_exist_user_id(user_id)

            if not password == row[0]:
                return error.error_authorization_fail()

            self.cursor.execute("SELECT user_id, status, store_id FROM
new_order WHERE order_id = %s;", (order_id, ))
            row = self.cursor.fetchone()
            if row is None:
                return error.error_invalid_order_id(order_id)

            status = row[1]
            store_id = row[2]

            if status == -1:
                return error.error_invalid_order_id(order_id)
            elif status == 2:
                return error.error_order_delivered(order_id)
            elif status == 3:
                return error.error_order_was_received(order_id)

            self.cursor.execute("SELECT book_id, count, price FROM orders
WHERE order_id = %s;", (order_id, ))
            book_info = []
            row = self.cursor.fetchall()
```

```
            for each in row:
                temp = {"book_id": each[0], "count": each[1], "price":
each[2]}
                book_info.append(temp)

            self.cursor.execute("SELECT user_id FROM user_store WHERE
store_id = %s;", (store_id, ))
            row = self.cursor.fetchone()
            seller_id = row[0]

            self.cursor.execute("UPDATE new_order SET status = %s WHERE
order_id = %s;", (-1, order_id))

            total_price = 0
            for each in book_info:
                self.cursor.execute("UPDATE store SET stock_level =
stock_level + %s WHERE store_id = %s AND book_id = %s", (each["count"],
store_id, each["book_id"]))
                total_price = total_price + each["count"] * each["price"]

            self.cursor.execute("UPDATE user SET balance = balance + %s WHERE
user_id = %s;", (total_price, user_id))
            self.cursor.execute("UPDATE user SET balance = balance + %s WHERE
user_id = %s;", (-total_price, seller_id))

            self.conn.commit()

order_col.delete_one({"order_id":order_id})
```

在编写该段函数时，我们在 error.py 中添加了一个错误代码与函数，为 523 – order was received，这表示该订单的状态为已发货，该订单不可以被取消。

```
    523: "order {} was received",
def error_order_was_received(order_id):
    return 523, error_code[523].format(order_id)
```

订单的取消实际上也可能是因为用户虽然下了订单，但却在规定时间没有付款。该功能的实现并没有被存放在该函数中，而是存放在了同在 store.py 下初始化函数中。

buyer.py 中的 new_order 函数为每一笔订单都添加了一个时间戳。在 store.py 中，我们在试图创建表格后，额外创建了一个线程，该线程会利用 schedule 模块的功能，每秒检查一次所有状态为 0 的订单中是否有订单持续状态为 0 超过 20 秒。这使得一笔订单如果在 20 秒之内都没有付款，该订单会自动被取消。

在这里，我们人为地为这个线程设置了 180 秒的运行时间。我们认为，一个真正被投入工业生产的数据库如果使用这样的线程来进行订单自动取消的功能实现，那么该线程的运行时间应当是无穷大，即只要主函数一直保持运行，该线程就应当一直保持运行。而我们目前的目的是通过测试文件的检测，因此我们设置了平均每次运行测试文件需要的时间，即 180 秒，作为线程的存活时间。在线程开始运行的 180 秒内，检测未付款订单存在时间是否超过 20 秒的功能会每秒被触发一次，一旦运行时间超过 180 秒，线程就会停止运行。

```
            def update_data():
                cur.execute("SELECT * from new_order WHERE status = 0")
                row = cur.fetchall()
                for each in row:
                    if (datetime.now() - each[3]).total_seconds() > 20:
                        cur.execute("UPDATE new_order SET status = -1 WHERE
order_id = %s;", (each[0], ))
                conn.commit()

            schedule.every(1).second.do(update_data)

            def run_schedule():
                while time.time() - start_time < 180:
                    schedule.run_pending()
                    time.sleep(1)

            schedule_thread = threading.Thread(target=run_schedule)
            schedule_thread.start()
```

4. 查询订单

订单一旦完成，就会被留存在历史记录中。基于此，我们编写了 search_order 函数以实现查询订单记录的功能，它的主体思路是利用 MySQL 的 SELECT 语句查找特定用户的历史订单记录。在这里，我们希望查询到的是一个订单的所有信息，所以我们采用了 LEFT JOIN 的语句，让描述订单的两个表格 new_order 与 orders 可以被连接在一起，这样做的目的是为了一次性获得该订单的所有信息。

```
        try:
            self.cursor = self.conn.cursor()
            self.cursor.execute("SELECT password FROM user where user_id
= %s;", (user_id, ))
            row = self.cursor.fetchone()

            if row is None:
                return error.error_non_exist_user_id(user_id)

            if not password == row[0]:
```

```
                return error.error_authorization_fail()

            self.cursor.execute("SELECT * FROM new_order LEFT JOIN orders ON
new_order.order_id = orders.order_id "
                                "WHERE new_order.user_id = %s;", (user_id, ))
```

5. 收货功能

当买家收到书籍后，买家理应可以确认收货。基于此，我们编写了 receive 函数以允许买家确认收货，其主体思路是使用 MySQL 中的 SELECT 语句确保用户存在，以及订单处于 2:已发货的状态，再利用 UPDATE 语句更新订单的状态以实现确认收货的状态。

```
    def receive_order(self, user_id, password, order_id) -> (int, str):
        try:
            self.cursor = self.conn.cursor()
            self.cursor.execute(
                "SELECT password from user where user_id=%s;", (user_id,)
            )
            row = self.cursor.fetchone()
            if row is None:
                return error.error_authorization_fail()

            if row[0] != password:
                return error.error_authorization_fail()

            self.cursor.execute(
                "SELECT user_id, status from new_order where order_id = %s;",
(order_id, )
            )
            row = self.cursor.fetchone()
            if row is None:
                return error.error_invalid_order_id(order_id)

            status = row[1]

            if status == -1:
                return error.error_invalid_order_id(order_id)
            elif status == 0:
                return error.error_order_not_paid(order_id)
            elif status == 1:
                return error.error_order_not_delivered(order_id)

            self.cursor.execute("UPDATE new_order SET status = %s where
order_id = %s;",
```

```
                                (3, order_id))
            self.conn.commit()
        except pymysql.Error as e:
            return 528, "{}".format(str(e))
        except BaseException as e:
            return 530, "{}".format(str(e))

        return 200, "ok"
```

在编写该段函数时，我们在 error.py 中添加了一个错误代码与函数，为 522 – order not delivered，这表示该订单的状态不在 2-已发货的状态，不可以确认收货。

```
    522: "order {} not delivered",
def error_order_not_delivered(order_id):
    return 522, error_code[522].format(order_id)
```

6. 添加键与索引

在设计数据库时，我们还需要考量两个重要的因素。

其一是互异性。在同一个数据库中，有的属性是不被允许重复出现的，如用户 ID、店铺 ID、订单 ID，这些属性只有保持互异性才能为不同的条目之间提供有效的区分方式。

其二是检索效率。按照传统的文档数据库的结构，所以需要依赖于查询的操作，如查询、删除特定条目、更新特定条目等，都需要通过遍历所有条目才能准确实现，因此在设计数据库时，最好可以考虑到提升其检索效率。

考虑到上述两个因素，我们在 store.py 中为每个表格都设置了**主键**、**外键**和**索引**。

在 MySQL 中，**主键**（Primary Key）的作用主要在于**确保表中的条目在某一个属性上不会存在相同的取值，从而让该属性作为区分不同条目的标志**。在有的表中，我们为个别属性添加了主键，以区分该表中的不同条目。

在 MySQL 中，**外键**（Foreign Key）的作用主要在于**确定数据一致性，以及让两张表存在一定的联系**。通过添加外键约束，表中将不会存在孤零零的，没有任何实际意义的数据。在有的表中，我们添加了外键的约束。

在数据库中，**索引**（Index）的使用可以**提升数据的检索效率**，让数据的检索不再是简单的遍历数据库。我们在有的表中的个别属性中添加了索引。

在 store.py 中，我们会执行创建表的操作。我们在创建表时均采用了 CREATE TABLE IF NOT EXISTS 语句，该语句会在数据库中不存在该表的时候创建表，在数据库中存在该表的时候不执行创建操作，这使得使用者无论是在什么时候执行这段代码都可以保证正确性。在创建表时，我们会在表中添加键与索引。

在创建 user 表时，我们为 user_id 添加了主键，使得 user 表中不可以存在 user_id 重复的条目。我们还为 user_id 添加了索引，这可以使得在检索 user 表时获得更高的检索效率。

```
            cur.execute(
                "CREATE TABLE IF NOT EXISTS user ("
```

```
                "user_id VARCHAR(300) PRIMARY KEY, password VARCHAR(300) NOT
NULL, "
                "balance INTEGER NOT NULL, token VARCHAR(500), terminal
VARCHAR(500), "
                "INDEX index_user (user_id));"
            )
```

在创建 user_store 表时，我们为 store_id 创建了主键，这使得 user_store 表中不存在 store_id 重复的条目。另外，我们为 user_store 表中的 user_id 添加了外键约束，使得 user_store 里的每个 user_id，必须对应于 user 表中的某个 user_id 的值。最后，我们为 store_id 添加了索引，这使得检索 user_store 的效率提升了。

```
            cur.execute(
                "CREATE TABLE IF NOT EXISTS user_store ("
                "user_id VARCHAR(300), store_id VARCHAR(300) PRIMARY KEY,"
                "FOREIGN KEY (user_id) REFERENCES user(user_id),"
                "INDEX index_store (store_id))"
            )
```

在创建 store 表时，我们为(store_id, book_id)创建了主键，即店铺 ID 可以重复，书本 ID 可以重复，但是不允许店铺 ID 和书本 ID 同时重复的条目出现。另外，我们为 store_id 属性添加了外键约束，要求 store 表里的每个 store_id，必须对应于 user_store 表中的某个 store_id 值。最后，我们为(store_id, book_id)的组合添加了复合索引，并分别为 title, tags, author 和 book_intro 添加了全文本索引，这不但使得在检索 ID 时可以提高效率，也可以在买家进行图书检索时可以提升文本检索效率。

在上次作业中，我们没有为书本内容添加索引。在本次作业中，我们为 book_intro 中添加了索引，使得用户在检索书本的内容信息时可以有效地提升检索速度。

```
            cur.execute(
                "CREATE TABLE IF NOT EXISTS store ("
                "store_id VARCHAR(300), book_id VARCHAR(300), title
VARCHAR(100), price INTEGER, "
                "tags VARCHAR(100), author VARCHAR(100),"
                "book_intro VARCHAR(2000),stock_level INTEGER,"
                "PRIMARY KEY (store_id, book_id),"
                "FOREIGN KEY (store_id) REFERENCES user_store(store_id),"
                # 复合索引
                "INDEX index_store_book (store_id, book_id),"
                "FULLTEXT INDEX index_title(title),"
                "FULLTEXT INDEX index_tags(tags),"
                "FULLTEXT INDEX index_author(author),"
                "FULLTEXT INDEX index_book_intro(book_intro))"
            )
```

在创建 new_order 表时，我们为 order_id 创建了主键，使得该表中不可以存在相同的 order_id。并且，我们为 user_id 和 store_id 添加了外键，规定该表中的每个 user_id，必须对应于 user 表中的某个 user_id，以及该表中的每个 store_id 必须对应于 user_store 表中的某个 store_id。我们还为 order_id 添加了索引，使得该表的检索效率得到了提升。

```
            cur.execute(
                "CREATE TABLE IF NOT EXISTS new_order ("
                "order_id VARCHAR(300) PRIMARY KEY , user_id VARCHAR(300),
store_id VARCHAR(300), "
                "time TIMESTAMP, status INTEGER,"
                "FOREIGN KEY (user_id) REFERENCES user(user_id), "
                "FOREIGN KEY (store_id) REFERENCES user_store(store_id),"
                "INDEX index_order (order_id))"
            )
```

在创建 orders 表时，我们为(order_id, book_id)创建了主键，不允许二者同时重复的条目出现。另外，我们还为 order_id 添加了外键约束，规定该表里的每个 order_id 都要对应于 new_order 表中的某个 order_id。我们还为(order_id, book_id)添加了复合索引，这使得我们在检索该表时可以获得效率的提升。

```
            cur.execute(
                "CREATE TABLE IF NOT EXISTS orders ("
                "order_id VARCHAR(300), book_id VARCHAR(300), count INTEGER,
price INTEGER,"
                "FOREIGN KEY (order_id) REFERENCES new_order(order_id),"
                "PRIMARY KEY (order_id, book_id), "
                "INDEX index_order_book (order_id, book_id))"
            )
```

在完成了上述功能完善后，我们也仿照项目原先提供的测试代码，自行在 fe/test 文件夹中添加了测试文件，在 be/view 文件夹中添加了后端路由代码。

为了测试卖家的发货功能是否正常，我们在前端设计了 test_deliver.py 文件，并在后端添加了 deliver 的路由。

```
    def test_ok(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code == 200

    def test_invalid_order(self):
```

```
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        self.order_id = self.order_id + '_x'
        code = self.seller.deliver_order(self.order_id)
        assert code != 200

    def test_canceled_order(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.buyer.cancel_order(self.order_id)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code != 200

    def test_unpaid_order(self):
        code = self.buyer.add_funds(self.total_price - 1)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code != 200

    def test_delivered_order(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code != 200
```

正确的发货：买家充值，付款，发货。

错误的发货：用户 ID 错误、用户取消订单、订单未付款、订单已发货，在这些情况发生后仍然尝试发货。

```
@bp_seller.route("/deliver_order", methods=["POST"])
def deliver_order():
    order_id: str = request.json.get("order_id")
```

```
    s = seller.Seller()
    code, message = s.deliver_order(order_id)

    return jsonify({"message": message}), code
```

为了测试买家的查询图书功能是否正常，我们在前端设计了 test_search_book.py 文件，并在后端添加了 search_books 的路由。

```
    def test_ok(self):
        ok, method, info = self.gen_book.gen_book_info()
        assert ok

        code = self.buyer.search_book("title", "test")
        assert code == 200

    def test_ok_store(self):
        ok, method, info = self.gen_book.gen_book_info()
        assert ok

        code = self.buyer.search_book("title", "test", self.store_id)
        assert code == 200

    def test_not_exist_store(self):
        ok, method, info = self.gen_book.gen_book_info()
        assert ok

        self.store_id = self.store_id + "_x"
        code = self.buyer.search_book("title", "test", self.store_id)
        assert code != 200
```

正确的搜索：方法正确，内容正确，店铺 id 正确或店铺 id 不存在。

错误的搜索：店铺不存在。

```
@bp_buyer.route("/search_book", methods=["POST"])
def search_book():
    store_id = request.json.get("store_id")
    method = request.json.get("method")
    keywords = request.json.get("keywords")
    page_number = request.json.get("page_number")
    b = Buyer()
    code, message = b.search_book(keywords, method, store_id, page_number)
    return jsonify({"message": message}), code
```

为了测试买家的取消订单功能是否正常，我们在前端设计了 test_cancel_order.py 文件，并在后端添加了 cancel_order 的路由。

```
    def test_ok(self):
        code = self.buyer.cancel_order(self.order_id)
        assert code == 200

    def test_authorization_error(self):
        self.buyer.password = self.buyer.password + "_x"
        code = self.buyer.payment(self.order_id)
        assert code != 200

    def test_order_not_exist_error(self):
        self.order_id = self.order_id + "_x"
        code = self.buyer.payment(self.order_id)
        assert code != 200

    def test_has_delivered(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code == 200
        code = self.buyer.cancel_order(self.order_id)
        assert code != 200
```

错误的情况包括：密码错误，订单不存在，订单已经被发出。

```
@bp_buyer.route("/cancel_order", methods=["POST"])
def cancel_order():
    user_id = request.json.get("user_id")
    password = request.json.get("password")
    order_id = request.json.get("order_id")
    b = Buyer()
    code, message = b.cancel_order(user_id, password, order_id)
    return jsonify({"message": message}), code
```

为了测试买家的检索订单功能是否正常，我们在前端设计了 test_search_order.py 文件，并在后端添加了 search_order 的路由。

```
    def test_ok(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.buyer.search_order(self.buyer_id, self.password)
        assert code == 200

    def test_authorization_error(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        self.buyer.password = self.buyer.password + "_x"
        code = self.buyer.search_order(self.buyer_id, self.buyer.password)
        assert code != 200

    def test_not_exist_user_error(self):
        self.buyer_id = self.buyer_id + "_x"
        code = self.buyer.search_order(self.buyer_id, self.buyer.password)
        assert code != 200
```

错误的情况包括：密码错误，用户不存在。

```
@bp_buyer.route("/search_order", methods=["POST"])
def search_order():
    user_id = request.json.get("user_id")
    password = request.json.get("password")
    b = Buyer()
    code, message = b.search_order(user_id, password)
    return jsonify({"message": message}), code
```

为了测试买家的确认收货功能是否正常，我们在前端设计了 test_receive.py 文件，并在后端添加了 receive 的路由。

```
    def test_ok(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code == 200
        code = self.buyer.receive_order(self.order_id)
        assert code == 200
```

```
    def test_authorization_error(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        self.buyer.password = self.buyer.password + "_x"
        code = self.buyer.receive_order(self.order_id)
        assert code != 200

    def test_not_exist_user_error(self):
        self.buyer_id = self.buyer_id + "_x"
        code = self.buyer.receive_order(self.order_id)
        assert code != 200

    def test_order_not_exist_error(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.seller.deliver_order(self.order_id)
        assert code == 200
        self.order_id = self.order_id + "_x"
        code = self.buyer.receive_order(self.order_id)
        assert code != 200

    def test_not_reached(self):
        code = self.buyer.add_funds(self.total_price)
        assert code == 200
        code = self.buyer.payment(self.order_id)
        assert code == 200
        code = self.buyer.receive_order(self.order_id)
        assert code != 200

    def test_not_paid(self):
        code = self.buyer.receive_order(self.order_id)
        assert code != 200

    def test_canceled_order(self):
        code = self.buyer.cancel_order(self.order_id)
        assert code == 200
        code = self.buyer.receive_order(self.order_id)
        assert code != 200
```

错误的情况：密码不正确，用户不存在，订单不存在，订单未发货，订单未付款，订单

已取消。

```python
@bp_buyer.route("/receive_order", methods=["POST"])
def receive_order():
    user_id = request.json.get("user_id")
    password = request.json.get("password")
    order_id = request.json.get("order_id")
    b = Buyer()
    code, message = b.receive_order(user_id, password, order_id)
    return jsonify({"message": message}), code
```

完成上述修改后，我们再次于 Git Bash 中运行了测试代码。最终我们得到的结果如图 9 与图 10 所示，整体的覆盖率达到了 92%。

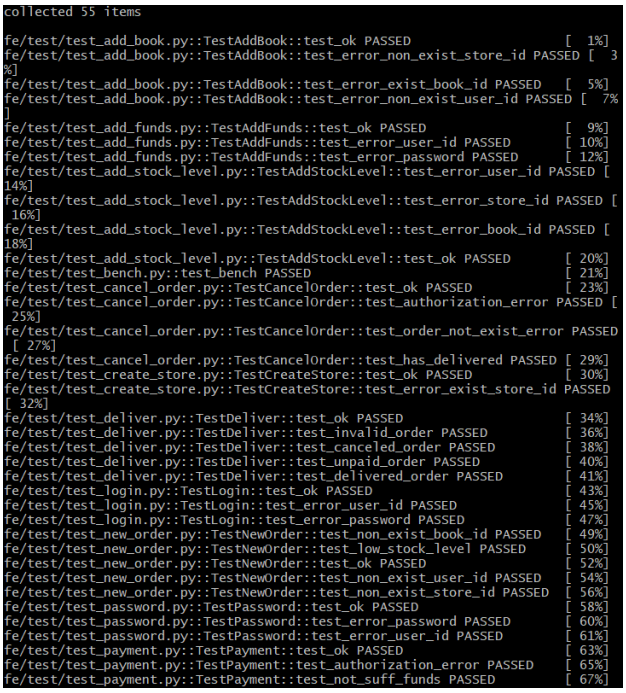
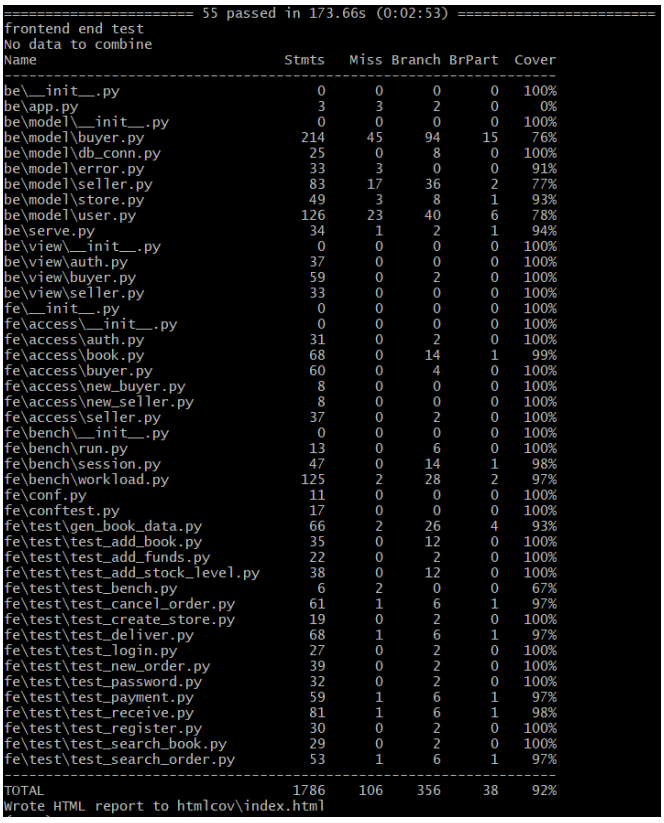


图 9 优化后的测试结果



```
====================== 55 passed in 173.66s (0:02:53) ======================
frontend end test
No data to combine
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                      214     45     94     15    76%
be\model\db_conn.py                     25      0      8      0   100%
be\model\error.py                       33      3      0      0    91%
be\model\seller.py                      83     17     36      2    77%
be\model\store.py                       49      3      8      1    93%
be\model\user.py                       126     23     40      6    78%
be\serve.py                             34      1      2      1    94%
be\view\__init__.py                      0      0      0      0   100%
be\view\auth.py                         37      0      0      0   100%
be\view\buyer.py                        59      0      2      0   100%
be\view\seller.py                       33      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      2      0   100%
fe\access\book.py                       68      0     14      1    99%
fe\access\buyer.py                      60      0      4      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      2      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     14      1    98%
fe\bench\workload.py                   125      2     28      2    97%
fe\conf.py                              11      0      0      0   100%
fe\conftest.py                          17      0      0      0   100%
fe\test\gen_book_data.py                66      2     26      4    93%
fe\test\test_add_book.py                35      0     12      0   100%
fe\test\test_add_funds.py               22      0      2      0   100%
fe\test\test_add_stock_level.py         38      0     12      0   100%
fe\test\test_bench.py                    6      2      0      0    67%
fe\test\test_cancel_order.py            61      1      6      1    97%
fe\test\test_create_store.py            19      0      2      0   100%
fe\test\test_deliver.py                 68      1      6      1    97%
fe\test\test_login.py                   27      0      2      0   100%
fe\test\test_new_order.py               39      0      2      0   100%
fe\test\test_password.py                32      0      2      0   100%
fe\test\test_payment.py                 59      1      6      1    97%
fe\test\test_receive.py                 81      1      6      1    98%
fe\test\test_register.py                30      0      2      0   100%
fe\test\test_search_book.py             29      0      2      0   100%
fe\test\test_search_order.py            53      1      6      1    97%
------------------------------------------------------------------------
TOTAL                                 1786    106    356     38    92%
Wrote HTML report to htmlcov\index.html
```

图 10 优化后的测试的覆盖率

五、 亮点总结

5.1 实现额外功能，改写了后端

在本次作业中，我们修改了后端，实现了作业要求的其他 40%的功能，包括卖家发货、买家收货、买家根据给出的关键词基于不同方法进行图书查询、订单的信息包含订单状态、买家订单查询、买家取消订单、超时自动取消订单。

5.2 编写了新的测试接口与后端路由

实现了的额外功能需要有测试代码进行检验。基于此，我们为大部分额外添加的功能添加了测试代码与对应的后端路由，并获得了 92%的代码覆盖率。详情可见 fe/test 与 be/view。

5.3 基于 Github 实现了代码备份与版本控制

我们知道一个实际上的开发工作往往会基于 Github 平台运作。在这里，我们采用了 Github 平台作为线上仓库，使得我们的代码得到了线上的备份，提升了代码的安全性。另外，我们为每一次较大的更新都上传了新的分支，这使得我们可以实现较好的版本控制。

在我们的线上仓库中，我们创建了五个分支。其中，V1.0 是最原始版本的代码，V1.1 是

经过了大量从 SQLite 至 MySQL 语法转换的，但仍然存在部分漏洞的版本，V1.2 是实现了 60%基础功能的版本，V2.0 是实现了除超时自动取消订单的其他额外功能的代码，V2.1 实现了我们计划的全部 40%额外功能。

详情可见 URL: https://github.com/HonokaKousaka/DataBaseAssignment2/tree/V2.1

5.4 绘制 ER 图并基于 ER 图导出关系模式、键与索引

ER 图正是在关系数据库中使用的，用于观察不同对象之间关系的关系图。我们绘制了 ER 图，并基于该 ER 图改造了原始代码提供的表格的结构描述，并在新创建的表格中添加了主键、外键与索引，使得该关系数据库的结构设计较为完备。

5.5 发现了代码的一个错误，并提交了一个 pull request

在代码的编写过程中，我们发现了在 fe/access/book.py 中，Book 类中没有描述货币单位的属性 currency_unit。该单位在 book.db 中存在，并且在之后的 get_book_info 中均有使用，因此该属性理应是需要补全的。

在发现了该漏洞后，我们提交了一个 pull request，并被合并到了源代码中。

六、 总结归纳

设计数据库需要通过多个步骤，不可跳跃。

通过需求分析，我们可以明白数据库可能会存储的数据。

通过概念分析，我们可以得到数据的实体内容与数据之间的关系

通过上述的考量，我们可以得到数据库的最终的逻辑结构。只有进行过这些分析，我们才可以正式进行数据库的实现。

每一步操作后都需要确认数据库中所有数据的一致性，避免出现错误。

文档数据库采用了“文档”的数据结构存储数据，并将数据展示给了用户。关系数据库则采用“关系的模型”，有类似于“面向对象”的思路。关系数据库采用了表格的形式，每一列可以看作一个属性。

通过给表格添加主键，可以让表格里的某个属性不允许重复出现。通过给表格添加外键，可以明确两个表格之间的关系。通过给表格添加索引，可以让表格的检索速度得到提升。

七、 参考文献

[1] 华东师范大学数据科学与工程学院课程《当代数据管理系统》电子版教材
作者：周烜、周欢、钱卫宁、周傲英
URL: https://github.com/xuanzhouhub/onlineDBMSbook