**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, pymysql, datetime, json, logging, sqlite3, schedule, threading, os, PyJWT, random, base64等模块以保证实验的正常进行。

本次实验在**Github**的个人账号上建立了本项目的repository以存储项目代码，并采用多分支存储实现**版本控制**，确保代码的维护、回退和查看正常进行。

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；

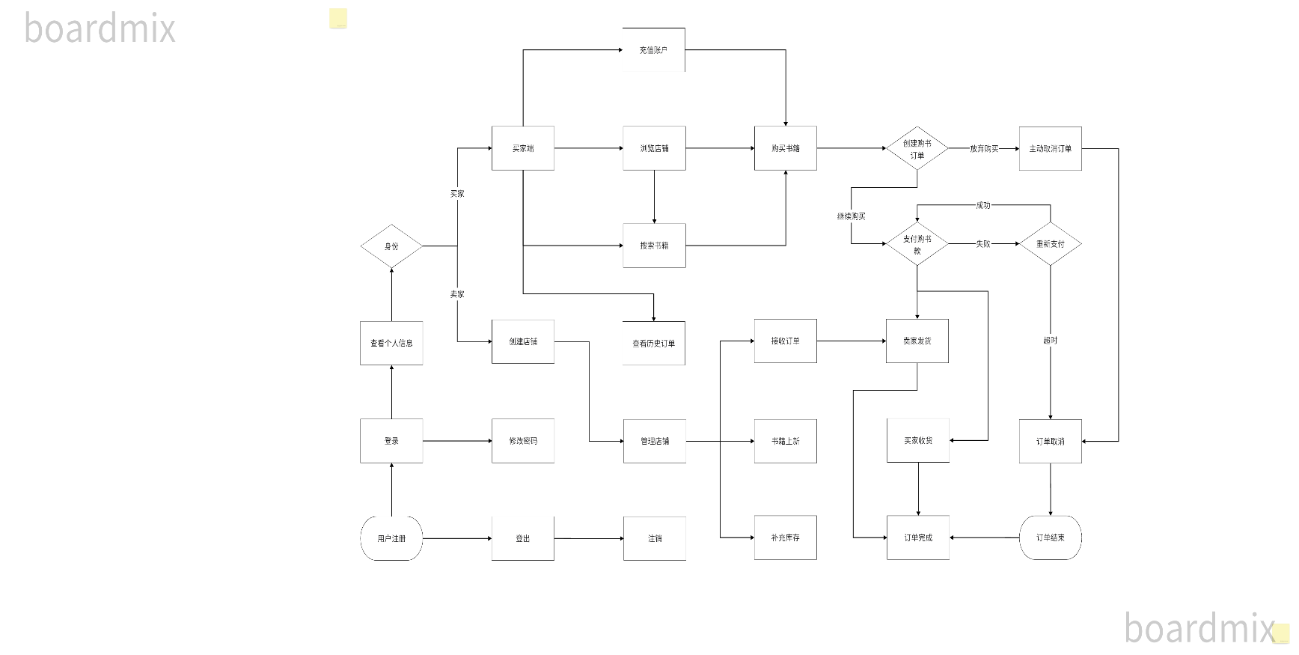
作为买家时，用户可以为自己的账户充值，并使用余额在任意商店购买图书；

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）。

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；

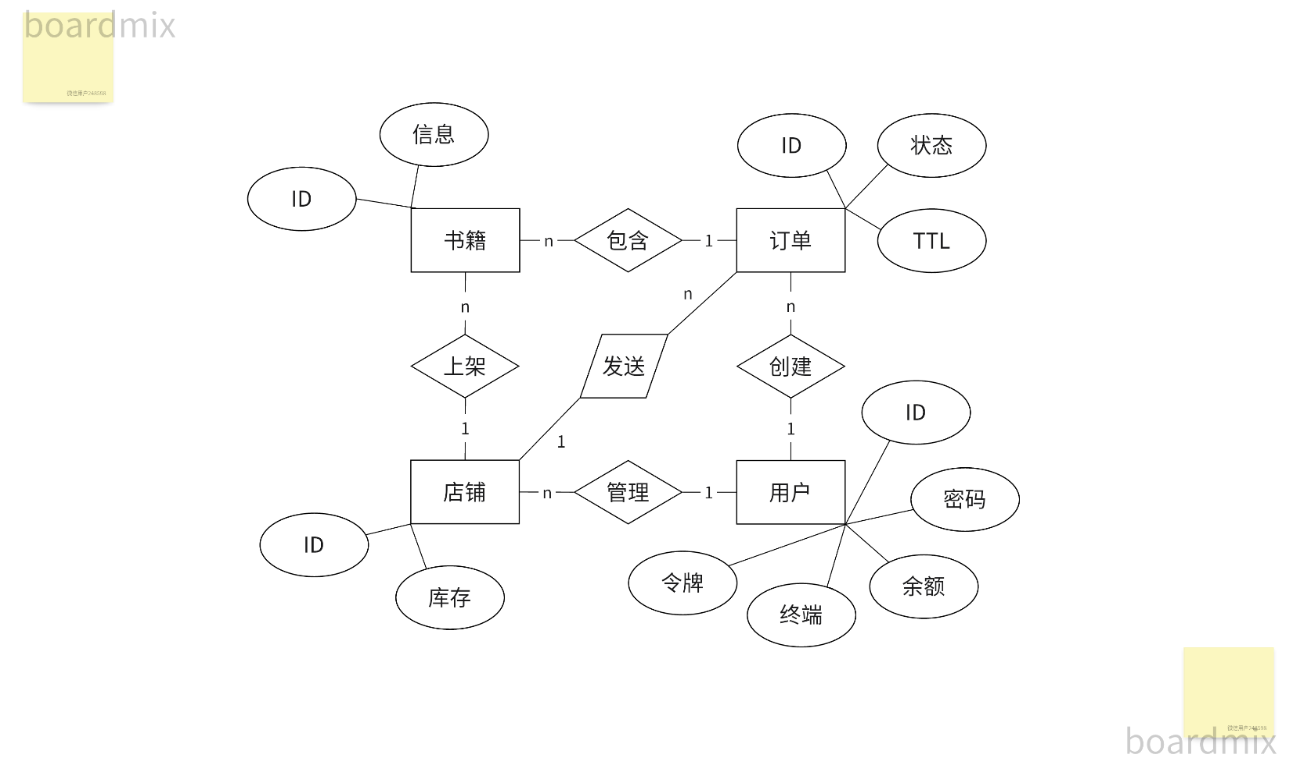
*书籍与店铺*：同理，一家店铺可以出售多本不同的书籍，同一的书籍也可以在不同的店铺出售；

*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。

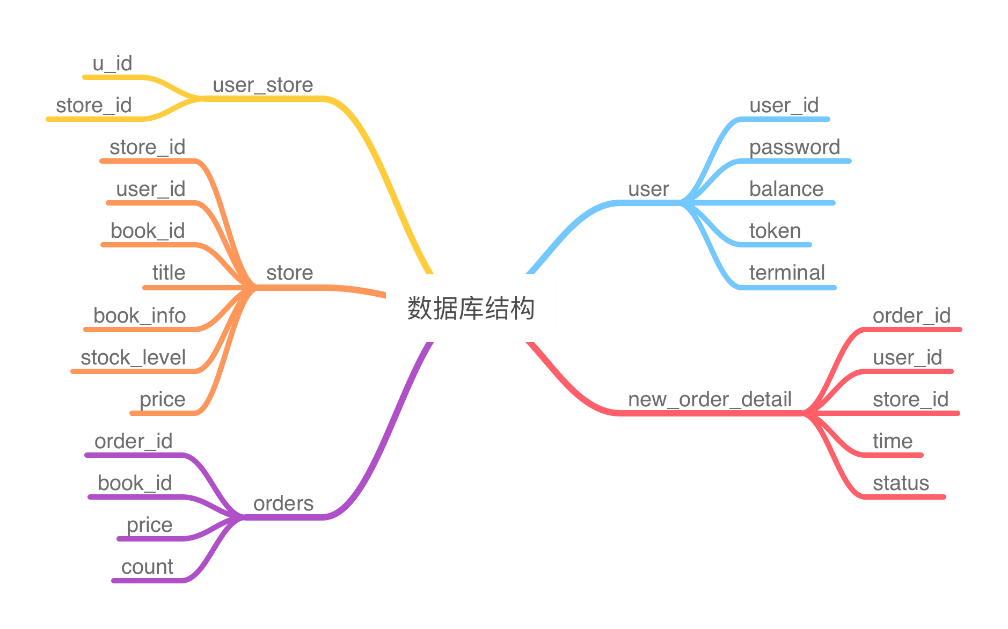
具体的概念结构如下所示：



**图2 数据库概念结构ER图**

* + 1. **逻辑结构设计**

上一次的项目中，我们使用MongoDB实现了该线上书店的功能。MongoDB作为一种文档数据库，其嵌套结构能够显著提升一部分查询的效率。其结构如下：



**图3 文档数据库结构导图**

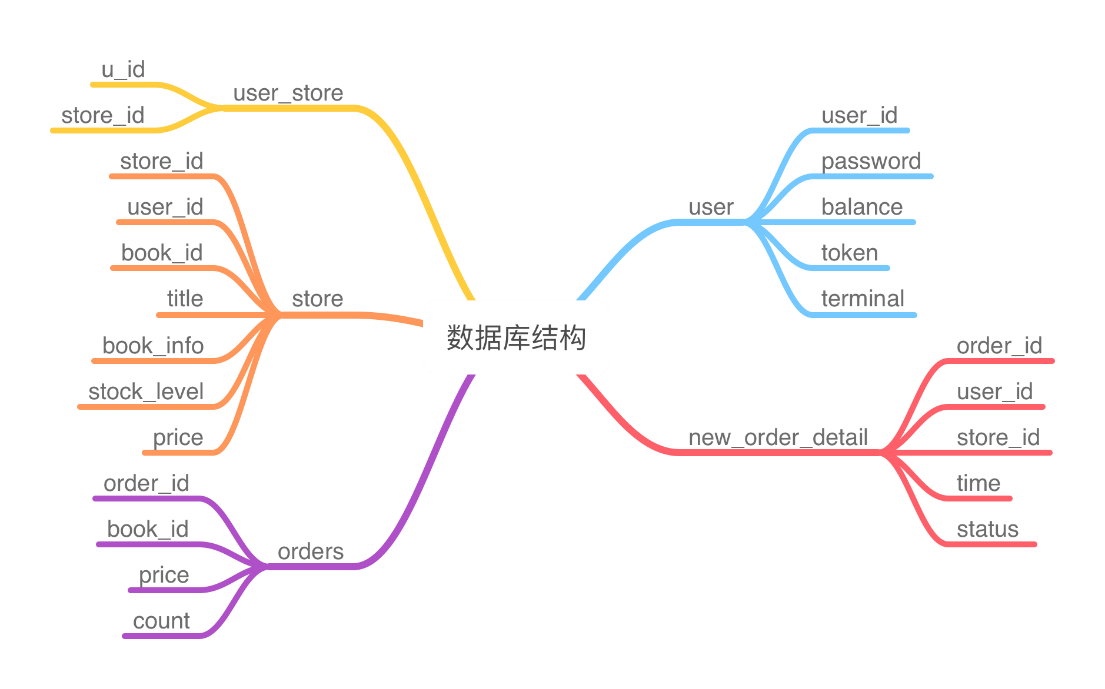
文档数据库通常以文档形式存储数据，这些文档可以是 JSON 或 BSON 格式。然而，这种非结构化的存储方式导致了数据的不同部分可能有不同的结构，难以对数据进行强制的结构化验证，可能导致数据一致性和完整性的问题。随着应用程序的复杂性增加，可能需要执行复杂的查询，而文档数据库对于这类查询的支持可能相对较弱。复杂的关联和聚合操作可能会变得冗长且难以维护。如果数据之间存在复杂的关系，例如多对多关系，文档数据库的表达能力可能较弱，而关系数据库在处理关系型数据方面更为适用。

相对应地，关系数据库使用表格结构，每个表格有预定义的列和数据类型，通过模式（Schema）对数据的结构进行强制性的定义，确保了数据的一致性和完整性。关系数据库具有强大的查询语言（SQL），支持复杂的查询操作，包括联结、子查询和聚合等。此外，关系数据库提供事务处理机制，确保了对数据库的多步骤操作是原子的、一致的、隔离的、持久的（ACID 特性）。关系数据库模型适用于处理多表之间的关系，能够有效地表示多对多、一对多和一对一等复杂的关系型数据模型。关系数据库是业界最为成熟和广泛应用的数据库模型之一，拥有丰富的工具、技术和社区支持，便于开发者进行使用、维护和优化。

本次实验中，原先提供的SQLite是一种嵌入式数据库，适用于单用户或轻量级应用，但在处理高并发和大规模数据时性能相对较差。由于其单一文件存储结构，SQLite 在需要处理大量并发连接或进行水平扩展时表现不佳。虽然支持基本的数据库操作，但相对而言，SQLite的功能相对较少，缺少一些高级特性，如存储过程和触发器。

而MySQL是一种独立的服务器型数据库，适用于高并发和大规模数据处理，具有更优越的性能，还能提供更好的可扩展性，支持复杂的并发操作和水平扩展。它还拥有丰富的数据库特性，适合处理复杂的业务逻辑。并且，MySQL支持复杂的事务处理，提供完备的事务控制和隔离级别以及更大的存储容量支持，适用于需要存储大量数据的应用。

基于以上需求，我们在转换语法的同时**对原有的数据库逻辑结构进行了调整**，增加了**索引**以进一步提升检索效率，并为未来的开发提供方便。新的MySQL逻辑结构如下：



**图4 关系数据库结构导图**

* 1. **代码实现**
     1. **理解代码**

正式开始工作前，我们应对本项目所包含的文件有一个初步的了解。

1. fe：前端。其中test负责测试代码正确性，bench负责测试数据库吞吐量，access负责前后端交互。
2. be：后端。其中view负责解析前端发送的请求，model负责与MySQL数据库进行交互。本次实验主要对model文件夹中的buyer，db\_conn，error，seller，store和user文件作出修改。
   * 1. **基础功能**

实现基础功能，首先我们需要将原有的SQLite语法转换为MySQL语法。具体而言，包括：

1. 修改占位符，从“?”修改为“%s”；
2. 增加游标提取操作，从“self.conn”修改为“self.cu=onn.cursor()”
3. 重新创建表格，按照**新的数据库逻辑结构**增加**键**与**索引；**

以上三步我们以store.py中创建表格的代码为例进行对比：

cursor = self.conn.execute(

                    "SELECT book\_id, stock\_level, book\_info FROM store "

                    "WHERE store\_id = ? AND book\_id = ?;",

                    (store\_id, book\_id),

                )

conn = self.get\_db\_conn()

            cursor = conn.cursor()

            cursor.execute("USE DBProj2;")

            cursor.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));"

            )

1. 修改报错，从“sqlite.Error”修改为“pymysql.Error”;

以store.py中报错的代码为例进行对比：

        except sqlite.Error as e:

            logging.error(e)

            conn.rollback()

        except pymysql.Error as e:

            logging.error(e)

            conn.rollback()

经历了这些修改之后，我们再逐步修改每一个文件。

**be/model/store.py**

这个文件实现了初始化数据库的功能，并连接数据库。根据前文所述，我们重新对数据库的逻辑结构进行了设计。我们在这里为每个表格设置了主键、外键和索引。

在MySQL中，主键确保表中的条目在某个属性上不重复，作为区分不同条目的标志。有些表格中，我们为特定属性添加了主键以确保互异性。外键在MySQL中用于确定数据一致性，建立表之间的联系。通过添加外键约束，我们可以避免存在没有实际意义的孤立数据。因此，在一些表格中，我们引入了外键约束，以维护数据关联性。索引的使用有助于提升数据检索效率，使得检索不再依赖于简单的遍历数据库。因此，我们在一些表格的特定属性上添加了索引，以加速数据检索过程。

在`user`表的创建中，我们为`user\_id`添加了主键，确保`user`表中不允许存在重复的`user\_id`。同时，为了提高检索效率，我们为`user\_id`添加了索引，优化了对`user`表的检索操作。

在创建`user\_store`表时，我们为`store\_id`创建了主键，杜绝了`user\_store`表中存在重复的`store\_id`条目。此外，为了维护数据一致性，我们为`user\_store`表中的`user\_id`添加了外键约束，确保每个`user\_id`必须对应于`user`表中的某个`user\_id`的值。为了提升检索效率，我们还为`store\_id`添加了索引，优化了对`user\_store`表的检索。

在`store`表的创建中，我们为`(store\_id, book\_id)`创建了主键，允许店铺ID和书本ID分别重复，但不允许它们同时重复。同时，为了保持数据关联性，我们为`store\_id`属性添加了外键约束，要求每个`store\_id`必须对应于`user\_store`表中的某个`store\_id`值。为了提高检索效率，我们为`(store\_id, book\_id)`创建了复合索引，并为`title`、`tags`、`author`和`book\_intro`分别添加了全文本索引，以在检索ID和买家进行图书检索时提升效率。

在`new\_order`表的创建中，我们为`order\_id`创建了主键，防止表中存在相同的`order\_id`。为了维护数据一致性，我们为`user\_id`和`store\_id`添加了外键，确保每个`user\_id`对应于`user`表中的某个`user\_id`，每个`store\_id`对应于`user\_store`表中的某个`store\_id`。为了提升检索效率，我们还为`order\_id`添加了索引，提高了对`new\_order`表的检索效率。

在`orders`表的创建中，我们为`(order\_id, book\_id)`创建了主键，禁止同时重复的条目。此外，为了维护数据关联性，我们为`order\_id`添加了外键约束，确保每个`order\_id`对应于`new\_order`表中的某个`order\_id`。为了提高检索效率，我们为`(order\_id, book\_id)`创建了复合索引，以提升对`orders`表的检索效率。

def \_\_init\_\_(self, db\_path):

        self.database = os.path.join(db\_path, "be.db")

        self.init\_tables()

    def init\_tables(self):

        try:

            conn = self.get\_db\_conn()

            cursor = conn.cursor()

            cursor.execute("USE DBProj2;")

            cursor.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));"

            )

            cursor.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))"

            )

            cursor.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),"

                "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))"

            )

            cursor.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))"

            )

            cursor.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()

            def update\_data():

                cursor.execute(

                    "SELECT \* from new\_order"

                    "WHERE status = 0"

                )

                row = cursor.fetchall()

                for each in row:

                    if (datetime.now() - each[3]).total\_seconds() > 20:

                        cursor.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 < 10:

                    schedule.run\_pending()

                    time.sleep(1)

            schedule\_thread = threading.Thread(target=run\_schedule)

            schedule\_thread.start()

        except pymysql.Error as e:

            logging.error(e)

            conn.rollback()

    def get\_db\_conn(self):

        return pymysql.connect(

            host="127.0.0.1",

            port=3306,

            user="root",

            password="021103",

            database="DBProj2"

        )

我们在创建表时采用`CREATE TABLE IF NOT EXISTS`语句。这确保了在数据库中表不存在时创建表，存在时不执行创建操作，避免重复创建同一表带来的混乱。在创建表时，我们为关键属性添加了主键和索引。例如，在创建user表时，我们为user\_id添加了主键，以防止存在重复的用户ID。同时，为user\_id添加索引可以提高在检索`user`表时的效率。

在get\_db\_conn()中连接MySQL数据库，返回数据库对象。

def get\_db\_conn():

    global database\_instance

    return database\_instance.get\_db\_conn()

**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store\_id\_exist()为例，查找特定store\_id的用户，若找不到则返回False。

    def store\_id\_exist(self, store\_id):

        self.cursor = self.conn.cursor()

        self.cursor.execute(

            "SELECT store\_id FROM user\_store"

            "WHERE store\_id = %s;",

            (store\_id,)

        )

        row = self.cursor.fetchone()

        if row is None:

            return False

        else:

            return True

**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。此处以register()为例，初始化用户的信息，将其插入进user表中。

    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"

**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表中。

def new\_order(

        self, user\_id: str, store\_id: str, id\_and\_count: [(str, int)]

    ) -> (int, str, str):

        order\_id = ""

        try:

            if not self.user\_id\_exist(user\_id):

                return error.error\_non\_exist\_user\_id(user\_id) + (order\_id,)

            if not self.store\_id\_exist(store\_id):

                return error.error\_non\_exist\_store\_id(store\_id) + (order\_id,)

            uid = "{}\_{}\_{}".format(user\_id, store\_id, str(uuid.uuid1()))

            self.cursor = self.conn.cursor()

            self.cursor.execute(

                "INSERT INTO new\_order(order\_id, user\_id, store\_id, time, status) "

                "VALUES(%s, %s, %s, %s, %s);",

                (uid, user\_id, store\_id, datetime.now(), 0)

            )

            for book\_id, count in id\_and\_count:

                self.cursor.execute(

                    "SELECT book\_id, stock\_level, price FROM store "

                    "WHERE store\_id = %s AND book\_id = %s;",

                    (store\_id, book\_id)

                )

                row = self.cursor.fetchone()

                if row is None:

                    return error.error\_non\_exist\_book\_id(book\_id) + (order\_id,)

                stock\_level = row[1]

                price = row[2]

                if stock\_level < count:

                    return error.error\_stock\_level\_low(book\_id) + (order\_id,)

                self.cursor.execute(

                    "UPDATE store set stock\_level = stock\_level - %s "

                    "WHERE store\_id = %s and book\_id = %s and stock\_level >= %s; ",

                    (count, store\_id, book\_id, count),

                )

                if self.cursor.rowcount == 0:

                    return error.error\_stock\_level\_low(book\_id) + (order\_id,)

                self.cursor.execute(

                    "INSERT INTO orders(order\_id, book\_id, count, price) "

                    "VALUES(%s, %s, %s, %s);",

                    (uid, book\_id, count, price)

                )

            self.conn.commit()

            order\_id = uid

        except pymysql.Error as e:

            logging.info("528, {}".format(str(e)))

            return 528, "{}".format(str(e)), ""

        except BaseException as e:

            logging.info("530, {}".format(str(e)))

            return 530, "{}".format(str(e)), ""

        return 200, "ok", order\_id

**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信息组织好，再将其插入store表中。

def add\_book(

        self,

        user\_id: str,

        store\_id: str,

        book\_id: str,

        book\_json\_str: str,

        stock\_level: int,

    ):

        try:

            if not self.user\_id\_exist(user\_id):

                return error.error\_non\_exist\_user\_id(user\_id)

            if not self.store\_id\_exist(store\_id):

                return error.error\_non\_exist\_store\_id(store\_id)

            if self.book\_id\_exist(store\_id, book\_id):

                return error.error\_exist\_book\_id(book\_id)

            book\_info\_json = json.loads(book\_json\_str)

            title = book\_info\_json.get("title")

            tags = book\_info\_json.get("tags")

            if tags is not None:

                tags = ",".join(tags)

            author = book\_info\_json.get("author")

            book\_intro = book\_info\_json.get("book\_intro")

            price = book\_info\_json.get("price")

            self.cursor = self.conn.cursor()

            self.cursor.execute(

                "INSERT into store(store\_id, book\_id, title, price, tags, author, book\_intro, stock\_level)"

                "VALUES (%s, %s, %s, %s, %s, %s, %s, %s);",

                (store\_id, book\_id, title, price, tags, author, book\_intro, stock\_level)

            )

            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"

* + 1. **拓展功能**

后40%部分由于代码水平和时间原因没能很好地编写测试接口。然而，我们仍然在这里将已写好的代码给出。

1. **卖家发货**

编写函数使用SELECT语句查找订单，判断是否可以发货，更新订单。若订单存在且已付款，则函数执行。

    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()

        except pymysql.Error as e:

            return 528, "{}".format(str(e))

        except BaseException as e:

            return 530, "{}".format(str(e))

        return 200, "ok"

1. **买家收货**

与发货类似，编写函数使用SELECT语句查找用户，判断订单状态并使用UPDATE语句更新订单状态。

    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"

本操作中，我们在error.py中添加错误代码522来返回订单状态非已发货状态的情况。

    522: "order {} undelivered",

1. **取消订单**

设计函数使用SELECT语句查找待取消订单，之后使用UPDATE语句回滚书本库存、卖家余额和买家余额，最后修改订单状态。

    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()

        except pymysql.Error as e:

            return 528, "{}".format(str(e))

        except BaseException as e:

            return 530, "{}".format(str(e))

        return 200, "ok"

本操作中，我们在error.py中添加错误代码523来返回订单已发货无法取消的情况。

def error\_order\_was\_received(order\_id):

    return 523, error\_code[523].format(order\_id)

1. **查询历史订单**

使用SELECT语句查找特定用户的历史订单，使用LEFT JOIN连接new\_order与orders以获取完整信息。

    def search\_order(self, user\_id: str, password: str) -> (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 \* FROM new\_order LEFT JOIN orders ON new\_order.order\_id = orders.order\_id "

                "WHERE new\_order.user\_id = %s;",

                (user\_id, )

            )

        except pymysql.Error as e:

            return 528, "{}".format(str(e))

        except BaseException as e:

            return 530, "{}".format(str(e))

        return 200, "ok"

1. **搜索图书**

使用SELECT语句查询数据库中内容，查询关键字和查询范围可选。使用LIMIT和OFFSET分页，每页25条。

    def search\_order(self, user\_id: str, password: str) -> (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 \* FROM new\_order LEFT JOIN orders ON new\_order.order\_id = orders.order\_id ",

                "WHERE new\_order.user\_id = %s;",

                (user\_id, )

            )

        except pymysql.Error as e:

            return 528, "{}".format(str(e))

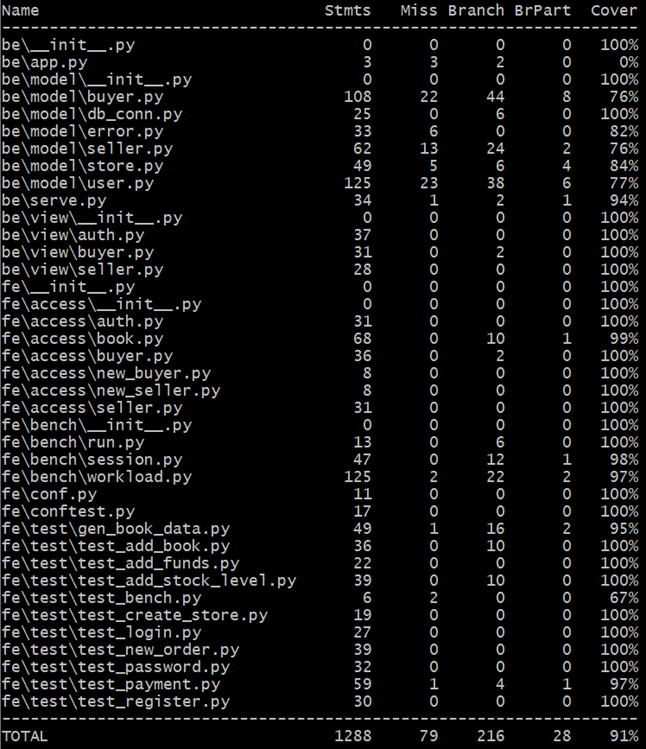
        except BaseException as e:

            return 530, "{}".format(str(e))

        return 200, "ok"

* + 1. **pytest测试结果**

接下来展示完整的代码测试结果。测试结果如下图所示，所有测试案例均通过，覆盖率达到91%。由于代码中存在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。



**图5 测试结果**

* 1. **设计亮点**

**·实现额外功能**

实现了包括发货、收货、查询书籍、查询订单、取消订单在内的额外功能。

**·基于Github进行版本管理**

本项目存在多个代码版本，为了实现更好的回滚和修改我们基于Github实现版本管理。具体请参见[AntaresXY/DBMSProj2: 数据库第二次大作业 (github.com)](https://github.com/AntaresXY/DBMSProj2)

**·绘制 ER 图并基于 ER 图导出关系模式、键与索引**

ER图是关系数据库中的一种工具，用于可视化不同对象之间的关系。我们通过绘制ER图，对原始代码中的表结构进行了改造，为每个表格添加了主键、外键和索引，以更全面地描述关系数据库的结构设计。这一优化使得数据库的结构更加完备，有助于更清晰地理解各个对象之间的关系。

1. 总结

在进行关系数据库大作业的设计过程中，我们首先进行了需求分析，深入理解了数据库可能存储的数据。通过概念分析，我们得到了数据的实体内容以及它们之间的关系，为后续设计奠定了基础。在这一过程中，我们注重数据一致性的确认，以避免潜在错误。

随后，我们通过逻辑设计步骤，将概念分析的结果转化为数据库的最终逻辑结构。这一过程是有序且不可跳跃的，确保数据库的设计合理且完备。在每一步操作后，我们都对数据库中的所有数据一致性进行确认，以确保设计的质量和可靠性。

在实现阶段，我们采用文档数据库和关系数据库两种不同的数据存储模型。文档数据库使用“文档”结构存储数据，将数据以用户友好的方式呈现。而关系数据库采用“关系模型”，以表格形式组织数据，类似于面向对象的思路。通过添加主键、外键和索引，我们增强了数据库的结构，使得数据管理更加精确和高效。

总的来说，在这个大作业中，我们通过系统性的分析和设计过程，建立了一个完备而一致的关系数据库。这不仅为实际应用提供了可靠的数据支持，也提高了数据库的可维护性和性能。