浙江大學

数据库系统实验报告

作业名称：	图书管理系统
姓　　名：	龙永奇
学　　号：	3220105907
电子邮箱：	3220105907@zju.edu.cn
联系电话：	15393113093
指导老师：	孙建伶

2024 年 04 月 16 日

实验名称

一、 实验目的

1. 设计并实现一个精简的图书管理程序，要求具有图书入库、查询、借书、还书、借书证管理等功能。
2. 提供一个基于 MySQL(或 OpenGauss, SQL Server)的精简图书管理程序，该图书管理程序应具备较好的可扩展性、鲁棒性和安全性，并且在高并发场景下仍能正确运行。

二、 系统需求

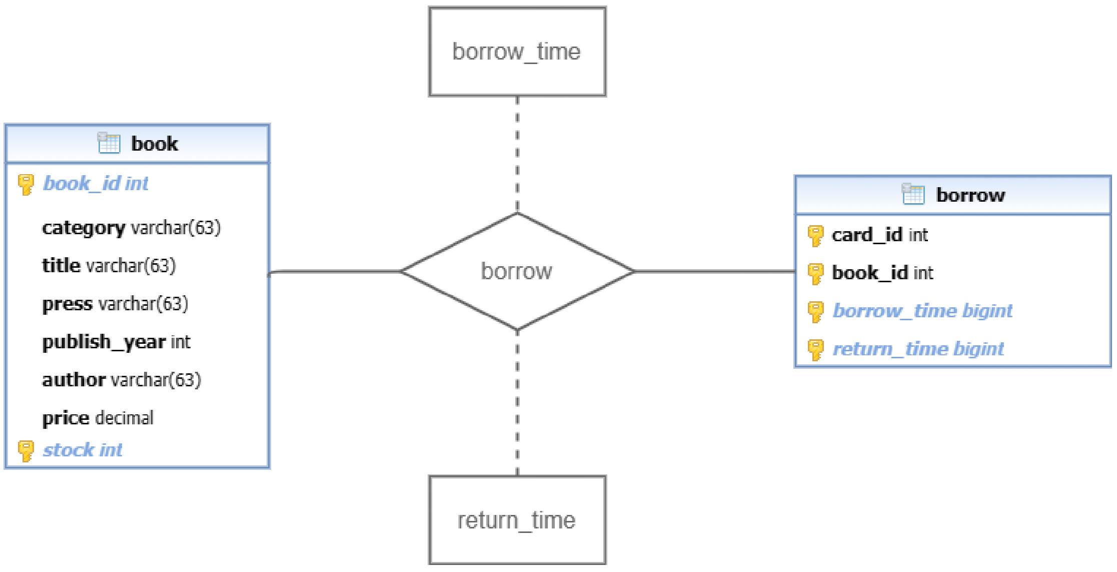
1. 操作系统：Windows 11

三、 实验环境

1. 数据库系统：MySQL 8.0.36
2. 开发环境：OpenJDK 17.0.10, Maven 3.9.6
3. 开发工具：MySQL Workbench 8.0 CE, IntelliJ IDEA 2024.1

四、 系统设计及实现

1. 绘制该图书管理系统的 E-R 图



2. 系统各函数的设计思路和实现

   (a) 图书入库模块 storeBook(Book book)

   ➢ 首先查询需要入库的图书是否在书库中，查询条件为：类别、书

名、出版社、年份、作者均相同则两本书相同

- 如果已存在，则返回 False
- 如果不存在，则执行插入语句，返回 True

```java
@Override
// funct1: 图书入库模块
public ApiResult storeBook(Book book) {
    Connection cont = connector.getConn();
    try {
        cont.setAutoCommit(false);  // 关闭自动提交事务
        // 计算类别, 书名, 出版社, 年份, 作者相同的书的个数
        String query = "SELECT COUNT(*) AS count FROM book
WHERE category = ? AND title = ? AND press = ? AND
publish_year = ? AND author = ?";
        PreparedStatement stmt =
cont.prepareStatement(query);
        stmt.setString(1, book.getCategory());
        stmt.setString(2, book.getTitle());
        stmt.setString(3, book.getPress());
        stmt.setInt(4, book.getPublishYear());
        stmt.setString(5, book.getAuthor());
        ResultSet rset = stmt.executeQuery();
        if(rset.next() && rset.getInt("count") >
0){         // 返回结果大于 0 说明有重复
            return new ApiResult(false, "Error: Book store
failed, " + book.getTitle() + " already exists!");
        }
        query = "INSERT INTO book(category, title, press,
publish_year, author, price, stock) values
(?, ?, ?, ?, ?, ?, ?);";
        stmt = cont.prepareStatement(query,
PreparedStatement.RETURN_GENERATED_KEYS);   // 返回生成的键
        stmt.setString(1, book.getCategory());
        stmt.setString(2, book.getTitle());
        stmt.setString(3, book.getPress());
        stmt.setInt(4, book.getPublishYear());
        stmt.setString(5, book.getAuthor());
        stmt.setDouble(6, book.getPrice());
        stmt.setInt(7, book.getStock());
        if (stmt.executeUpdate() == 0) {
// executeUpdate()方法用于执行 INSERT、UPDATE 或 DELETE 语句以及
SQL DDL 语句，返回一个整数，表示受影响的行数，如果返回值为 0，则表
示没有受影响
```

```
            return new ApiResult(false, "Error: Book store
failed!");
        }
        rset = stmt.getGeneratedKeys();
// getGeneratedKeys()方法用于获取由于执行此 Statement 对象而创
建的所有自动生成的键
        if (rset.next()) {
            book.setBookId(rset.getInt(1));
// getInt(1)取出第一列数据，将取出来的 bookid 赋值给 book
        }

        cont.commit(); // 提交事务
        return new ApiResult(true, "Book Store
successfull!");
    } catch (Exception expt) {
        rollback(cont); // 回滚事务
        return new ApiResult(false, "Error: book store
failed! " + expt.getMessage());
    }
}
```

(b) 更新图书库存模块 incBookStock(int bookId, int deltaStock)

- 首先根据 bookId 查询图书当前库存
- 如果当前库存在更新后为负，则返回 False，图书库存不足
- 如果当前库存在更新后非负，则返回 True，库存更新成功

```
@Override
// funct2: 图书增加库存模块。为图书库中的某一本书增加库存,其中库
存增量 deltaStock 可正可负，若为负数，则需要保证最终库存是一个非负
数。
public ApiResult incBookStock(int bookId, int deltaStock) {
    Connection cont = connector.getConn();
    try {
        cont.setAutoCommit(false);

        String query = "SELECT stock FROM book WHERE book_id
= ? ";
        PreparedStatement stmt =
cont.prepareStatement(query);
        stmt.setInt(1, bookId);
        ResultSet rset = stmt.executeQuery();
        if (rset.next()) {
            if(rset.getInt(1) + deltaStock < 0) {
                return new ApiResult(false, "Modify stock
failed: Book is not enough!");
            }
            query = "UPDATE book SET stock = ? WHERE book_id
= ?";
            stmt = cont.prepareStatement(query);
            stmt.setInt(1, rset.getInt(1) + deltaStock);
            stmt.setInt(2, bookId);
            stmt.executeUpdate();
        } else {
            return new ApiResult(false, "Error: Modify stock
failed!");
        }
        cont.commit();
        return new ApiResult(true, "Modify stock
successfully!");
    } catch(Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Modify stock
failed! " + expt.getMessage());
    }
}
```

(c) 图书批量入库模块 storeBook(List<Book> books)

- 首先建立一个图书迭代器，对图书进行循环插入
- 使用 addBatch()以及 executeBatch()进行批量操作
- 插入结束后，通过循环查询每本书的书号，更新新插入图书的书号

```
@Override
// funct3: 图书批量入库模块。批量入库图书，如果有一本书入库失败，
那么就需要回滚整个事务(即所有的书都不能被入库)。
public ApiResult storeBook(List<Book> books) {
    Connection cont = connector.getConn();
    try {
        cont.setAutoCommit(false);

        String query = "INSERT INTO book(category, title,
press, publish_year, author, price, stock) values
(?, ?, ?, ?, ?, ?, ?);";
        PreparedStatement stmt =
cont.prepareStatement(query,
PreparedStatement.RETURN_GENERATED_KEYS);
        Iterator<Book> iterator = books.iterator();
        while(iterator.hasNext()) {
            Book book = iterator.next();
            stmt.setString(1, book.getCategory());
            stmt.setString(2, book.getTitle());
            stmt.setString(3, book.getPress());
            stmt.setInt(4, book.getPublishYear());
            stmt.setString(5, book.getAuthor());
            stmt.setDouble(6, book.getPrice());
            stmt.setInt(7, book.getStock());
            stmt.addBatch();
        }
        stmt.executeBatch();
        ResultSet rset = stmt.getGeneratedKeys();

        iterator = books.iterator();
        while(iterator.hasNext()){
            Book book = iterator.next();
            if(rset.next()) {   // 设置每本书的 book_id
                book.setBookId(rset.getInt(1));
            }
        }

        cont.commit();
        return new ApiResult(true, "Books store
successfully!");
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Books store
failed! " + expt.getMessage());
    }
}
```

(d) 图书删除模块 removeBook(int bookId)

- ➢ 首先在 borrow 表格中查找是否有人正在借阅此书，通过 return_time = 0 判断，如果有则直接返回 False，提示图书正在借阅中
- ➢ 使用 delete 语句对图书进行删除，通过 executeUpdate()函数的返回值判断是否删除成功，如果返回值为 0，说明没有行被影响，此时返回 False，否则删除成功

```
@Override
// funct4: 图书删除模块。从图书库中删除一本书。如果还有人尚未归还
这本书，那么删除操作将失败。
```

```
public ApiResult removeBook(int bookId) {
    Connection cont = connector.getConn();
    try {
        cont.setAutoCommit(false);

        String query = "SELECT * FROM borrow WHERE book_id
= ? AND return_time = 0;";  // 判断是否有人借了这本书
        PreparedStatement stmt =
cont.prepareStatement(query);
        stmt.setInt(1,bookId);
        ResultSet rset = stmt.executeQuery();
        if(rset.next()){
            return new ApiResult(false, "Error: Delete book
failed, the book " + bookId + " has been borrowed!");
        }

        query = "DELETE FROM book WHERE book_id = ?;";  // 删
除这本书
        stmt = cont.prepareStatement(query);
        stmt.setInt(1, bookId);
        if(stmt.executeUpdate() == 0) {
            return new ApiResult(false, "Error: Delete book
failed, the book " + bookId + " is not in the library!");
        }

        cont.commit();
        return new ApiResult(true, "Delete book
successfully!");
    } catch(Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Delete book
failed! " + expt.getMessage());
    }
}
```

(e) 图书信息更新模块 modifyBookInfo(Book book)

- 直接使用 Update 语句根据给出的 book 进行更新
- 更新属性包括：category，title，press，publish_year，author，price

```
@Override
// funct5: 图书修改模块。修改已入库图书的基本信息，该接口不能修改
图书的书号和存量。
public ApiResult modifyBookInfo(Book book) {
    Connection cont = connector.getConn();
    try {
        cont.setAutoCommit(false);

        String query = "UPDATE book SET category = ?, title 
= ?, press = ?, publish_year = ?, author = ?, price = ? 
WHERE book_id = ?";
        PreparedStatement stmt = 
cont.prepareStatement(query);
        stmt.setString(1, book.getCategory());
        stmt.setString(2, book.getTitle());
        stmt.setString(3, book.getPress());
        stmt.setInt(4, book.getPublishYear());
        stmt.setString(5, book.getAuthor());
        stmt.setDouble(6, book.getPrice());
        stmt.setInt(7, book.getBookId());

        if(stmt.executeUpdate() == 0) {
            return new ApiResult(false, "Error: Modify book 
information failed, the book is not in the library!");
        }
        cont.commit();
        return new ApiResult(true, "Modify book information 
successfully!");
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Modify book 
information failed! " + expt.getMessage());
    }
}
```

(f) 图书查询模块 queryBook(BookQueryConditions conditions)

- 由于字符串结构是模糊查询，因此查询语句中使用了 sql 语言的 like 以及%String%的形式
- 对于价格和出版年份是范围查询，如果查询条件中没有对应条件，则置为 MIN_VALUE 以及 MAX_VALUE
- 注意到函数要求的是按照指定的方式进行排序，因此在查询语句的末尾添加 `conditions.getSortBy() + " " + conditions.getSortOrder() + " " + ",book_id ASC;"`
- 随后对每种查询条件进行判断，如果 conditions 给出了对应的属性，则填入条件，否则使用通配符%进行占位

➢ 最后循环建立一个结果链表，输出结果

```
@Override
// funct6: 图书查询模块。根据提供的查询条件查询符合条件的图书，并
按照指定排序方式排序
// 查询条件包括：类别点查(精确查询)，书名点查(模糊查询)，出版社点
查(模糊查询)，年份范围查，作者点查(模糊查询)，价格范围差。
// 如果两条记录排序条件的值相等，则按 book_id 升序排序。
public ApiResult queryBook(BookQueryConditions conditions) {
    Connection cont = connector.getConn();
    List<Book> books_list = new ArrayList<>();
    try {
        cont.setAutoCommit(false);

        String query =  "SELECT * FROM book WHERE " +
                       "category like ? AND title like ? AND
press like ? AND publish_year >= ? AND publish_year <= ? AND
author like ? AND price >= ? AND price <= ? " +
                       "ORDER BY " + conditions.getSortBy()
+ " " + conditions.getSortOrder() + " " + ",book_id ASC;";
        PreparedStatement stmt =
cont.prepareStatement(query);
        if(conditions.getCategory() != null)
            stmt.setString(1, "%" + conditions.getCategory()
+ "%");
        else stmt.setString(1, "%");
        if(conditions.getTitle() != null)
            stmt.setString(2, "%" + conditions.getTitle() +
"%");
        else stmt.setString(2, "%");
        if(conditions.getPress() != null)
            stmt.setString(3, "%" + conditions.getPress() +
"%");
        else stmt.setString(3, "%");
        if(conditions.getMinPublishYear() != null)
            stmt.setLong(4, conditions.getMinPublishYear());
        else stmt.setLong(4, Long.MIN_VALUE);
        if(conditions.getMaxPublishYear() != null)
            stmt.setLong(5, conditions.getMaxPublishYear());
        else stmt.setLong(5, Long.MAX_VALUE);
        if(conditions.getAuthor() != null)
            stmt.setString(6, "%" + conditions.getAuthor() +
"%");
        else stmt.setString(6, "%");
        if(conditions.getMinPrice() != null)
```

```
            stmt.setDouble(7, conditions.getMinPrice());
        else stmt.setDouble(7, Double.MIN_VALUE);
        if(conditions.getMaxPrice() != null)
            stmt.setDouble(8, conditions.getMaxPrice());
        else stmt.setDouble(8, Double.MAX_VALUE);
        ResultSet rset = stmt.executeQuery();

        while(rset.next()) {
            Book book = new Book();
            book.setBookId(rset.getInt(1));
            book.setCategory(rset.getString(2));
            book.setTitle(rset.getString(3));
            book.setPress(rset.getString(4));
            book.setPublishYear(rset.getInt(5));
            book.setAuthor(rset.getString(6));
            book.setPrice(rset.getDouble(7));
            book.setStock(rset.getInt(8));
            books_list.add(book);
        }
        cont.commit();
        BookQueryResults Results = new
BookQueryResults(books_list);
        return new ApiResult(true, Results);
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Query book
failed! " + expt.getMessage());
    }
}
```

(g) 借书模块 borrowBook(Borrow borrow)

- 首先查询借阅人是否已经借阅该本图书但未归还，判断条件为 return_time = 0，如果未归还则返回 False
- 这里需要使用 for update 加锁，防止表的数据被改变
- 随后判断是否存在此图书，此图书的余量是否足够，判断条件为 stock 属性是否小于 1，如果余量不足或不存在该本图书，则返回 False
- 最后执行 stock – 1 语句，并增加借书记录

```
@Override
```

```
// funct7: 借书模块。根据给定的书号、卡号和借书时间添加一条借书记录，然后更新库存。若用户此前已经借过这本书但尚未归还，那么借书操作将失败。
public ApiResult borrowBook(Borrow borrow) {
    Connection cont = connector.getConn();
    try{
        cont.setAutoCommit(false);

        // 书是否已经借出但没有还
        String query = "SELECT * FROM borrow WHERE card_id = ? AND book_id = ? AND return_time = 0;";
        PreparedStatement stmt = cont.prepareStatement(query);
        stmt.setInt(1, borrow.getCardId());
        stmt.setInt(2, borrow.getBookId());
        ResultSet rset = stmt.executeQuery();
        if(rset.next()){
            return new ApiResult(false, "Error: You have already borrowed this book!");
        }
        // 书是否存在，书的余量是否足够
        query = "SELECT stock FROM book WHERE book_id = ? for update;"; // for update 加锁，防止表数据被改变
        stmt = cont.prepareStatement(query);
        stmt.setInt(1, borrow.getBookId());
        rset = stmt.executeQuery();
        if(rset.next()) {
            if(rset.getInt("stock") < 1){
                return new ApiResult(false, "Error: There is not enough stock to borrow the book " + borrow.getBookId());
            }
        } else {
            return new ApiResult(false, "Error: There is no such book!");
        }
        // 库存-1
        query = "UPDATE book SET stock = stock - 1 WHERE book_id = ?;";
        stmt = cont.prepareStatement(query);
        stmt.setInt(1, borrow.getBookId());
        stmt.executeUpdate();
        // 增加借书记录
        query = "INSERT INTO borrow(card_id, book_id, borrow_time) VALUES(?, ?, ?);";
        stmt = cont.prepareStatement(query);
        stmt.setInt(1, borrow.getCardId());
        stmt.setInt(2, borrow.getBookId());
        stmt.setLong(3, borrow.getBorrowTime());
        stmt.executeUpdate();

        cont.commit();
        return new ApiResult(true, "Borrow book 
successfully!");
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Borrow book 
failed! " + expt.getMessage());
    }
}
```

(h) 还书模块 returnBook(Borrow borrow)

- ➢ 使用 Update 语句对 return_time 进行补充
- ➢ 检查 executeUpdate()函数的返回值，如果为 0，说明归还失败，返回 False，否则返回 True

```
@Override
// funct8: 还书模块。根据给定的书号、卡号和还书时间，查询对应的借
书记录，并补充归还时间，然后更新库存。
public ApiResult returnBook(Borrow borrow) {
    Connection cont = connector.getConn();
    try {
        cont.setAutoCommit(false);
        // 设置归还时间
        String query = "UPDATE borrow SET return_time = ? 
WHERE card_id = ? AND book_id = ? AND borrow_time = ?;";
        PreparedStatement stmt = 
cont.prepareStatement(query);
        stmt.setLong(1, borrow.getReturnTime());
        stmt.setInt(2, borrow.getCardId());
        stmt.setInt(3, borrow.getBookId());
        stmt.setLong(4, borrow.getBorrowTime());
        if(stmt.executeUpdate() == 0){
            return new ApiResult(false, "Error: There is no 
such borrow record!");
        }
        // 更新库存
        query = "UPDATE book SET stock = stock + 1 WHERE 
book_id = ?;";
```

```
        stmt = cont.prepareStatement(query);
        stmt.setInt(1, borrow.getBookId());
        stmt.executeUpdate();

        cont.commit();
        return new ApiResult(true, "Return book 
successfully!");
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Return book 
failed! " + expt.getMessage());
    }
}
```

(i) 借书记录查询模块 showBorrowHistory(int cardId)

- 使用 Select 语句查询给出 cardId 用户的借书记录
- 返回结果根据 borrow_time 降序、book_id 升序的顺序排列
- 通过遍历将借书记录存入 BorrowHistories.Item borrow 中

```
@Override
// funct9: 借书记录查询模块。查询某个用户的借书记录，按照借书时间
递减、书号递增的方式排序。
public ApiResult showBorrowHistory(int cardId) {
    Connection cont = connector.getConn();
    List<BorrowHistories.Item> borrow_list = new 
LinkedList<>();
    try {
        cont.setAutoCommit(false);

        String query = "SELECT * FROM borrow join book on 
borrow.book_id = book.book_id WHERE card_id = ? ORDER BY 
borrow_time DESC, book.book_id ASC;";
        PreparedStatement stmt = 
cont.prepareStatement(query);
        stmt.setInt(1, cardId);
        ResultSet rset = stmt.executeQuery();

        while(rset.next()) {
            BorrowHistories.Item borrow = new 
BorrowHistories.Item();
            // 借书信息
            borrow.setCardId(rset.getInt("card_id"));
            borrow.setBookId(rset.getInt("book_id"));
            borrow.setBorrowTime(rset.getLong("borrow_time")
);
            borrow.setReturnTime(rset.getLong("return_time")
);
            // 书籍信息
            borrow.setCategory(rset.getString("category"));
            borrow.setTitle(rset.getString("title"));
            borrow.setPress(rset.getString("press"));
            borrow.setPublishYear(rset.getInt("publish_year"
));
            borrow.setAuthor(rset.getString("author"));
            borrow.setPrice(rset.getDouble("price"));

            borrow_list.add(borrow);
        }
        BorrowHistories borrowHistories = new
BorrowHistories(borrow_list);
        commit(cont);
        return new ApiResult(true, borrowHistories);
    } catch(Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Error: Show borrow
history failed! " + expt.getMessage());
    }
}
```

(j) 借书证注册模块 registerCard(Card card)

- 首先执行 Insert 语句，将给出的 card 插入 Card 表格中
- 根据 executeUpdate()函数返回的结果判断是否插入成功，如果返回 0，则插入失败，否则插入成功

```
@Override
// funct10: 借书证注册模块。注册一个借书证，若借书证已经存在，则
该操作将失败。当且仅当<姓名, 单位, 身份>均相同时，才认为两张借书证
相同。
public ApiResult registerCard(Card card) {
    Connection cont = connector.getConn();
    try {
        cont.setAutoCommit(false);

        String query = "INSERT INTO card(name, department,
type) VALUES(?, ?, ?);";
```

```
        PreparedStatement stmt =
cont.prepareStatement(query,
PreparedStatement.RETURN_GENERATED_KEYS);
        stmt.setString(1, card.getName());
        stmt.setString(2, card.getDepartment());
        stmt.setString(3, card.getType().getStr());

        if(stmt.executeUpdate() == 0) {
            return new ApiResult(false, "Register card
failed!");
        }
        ResultSet rs = stmt.getGeneratedKeys();
        if(rs.next()){
            card.setCardId(rs.getInt(1));
        }

        cont.commit();
        return new ApiResult(true, "Register card
successfully! Your card id is " + card.getCardId());
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Register card failed! "
+ expt.getMessage());
    }
}
```

(k) 借书证删除模块 removeCard(int cardId)

- 首先使用 Select 语句查询当前借书证是否有未归还的图书，查询条件为 return_time = 0，如果有则直接返回 False
- 使用 Delete 语句对借书证进行删除
- 使用 executeUpdate()函数判断是否删除成功

```
@Override
// funct11: 删除借书证模块。如果该借书证还有未归还的图书，那么删
除操作将失败。
public ApiResult removeCard(int cardId) {
    Connection cont = connector.getConn();
    try{
        cont.setAutoCommit(false);

        String query = "SELECT * FROM borrow WHERE card_id
= ? AND return_time = 0;";    // 一定是未归还的图书
        PreparedStatement stmt =
cont.prepareStatement(query);
```

```
        stmt.setInt(1, cardId);
        ResultSet rset = stmt.executeQuery();
        if(rset.next()){
            return new ApiResult(false, "Remove card failed!
You have books that haven't been returned.");
        }

        query = "DELETE FROM card WHERE card_id = ?;";
        stmt = cont.prepareStatement(query);
        stmt.setInt(1, cardId);
        if(stmt.executeUpdate() == 0){
            return new ApiResult(false, "Remove card
failed!");
        }
        cont.commit();
        return new ApiResult(true, "Remove card
successfully!");
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Remove card failed! " +
expt.getMessage());
    }
}
```

(l) 借书证查询模块 showCards()

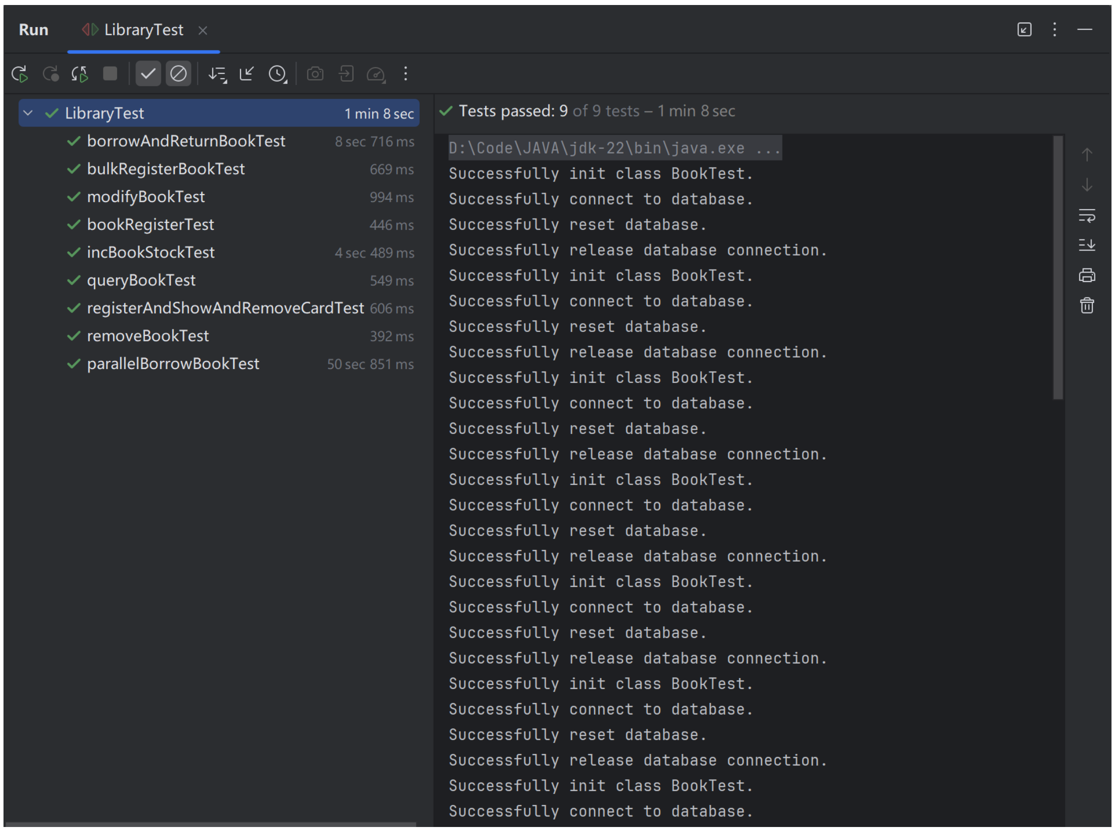
➢ 此函数返回的是所有借书证，直接使用 Select 语句对整个 Card 表进行选择，按照 CardId 的顺序列出，将结果存在链表中

```
@Override
// funct12: 借书证查询模块。列出所有的借书证。
public ApiResult showCards() {
    Connection cont = connector.getConn();
    List<Card> card_list = new LinkedList<>();
    try{
        cont.setAutoCommit(false);

        String query = "SELECT * FROM card ORDER BY card_id
ASC;"  // 按顺序列出
        PreparedStatement stmt =
cont.prepareStatement(query);
        ResultSet rset = stmt.executeQuery();
        while(rset.next()) {
            Card card = new Card();
            card.setCardId(rset.getInt("card_id"));
            card.setName(rset.getString("name"));
            card.setDepartment(rset.getString("department"))
;
            card.setType(Card.CardType.values(rset.getString
("type")));
            card_list.add(card);
        }
        cont.commit();
        return new ApiResult(true, new CardList(card_list));
    } catch (Exception expt) {
        rollback(cont);
        return new ApiResult(false, "Show cards failed! " +
expt.getMessage());
    }
}
```

3. 程序运行结果场景以及截图说明（即实验指导文档中的系统功能验证）

本次实验未完成前端部分，以下是测试结果：



```
✓ LibraryTest                              1 min 8 sec
    ✓ borrowAndReturnBookTest              8 sec 716 ms
    ✓ bulkRegisterBookTest                 669 ms
    ✓ modifyBookTest                       994 ms
    ✓ bookRegisterTest                     446 ms
    ✓ incBookStockTest                     4 sec 489 ms
    ✓ queryBookTest                        549 ms
    ✓ registerAndShowAndRemoveCardTest     606 ms
    ✓ removeBookTest                       392 ms
    ✓ parallelBorrowBookTest               50 sec 851 ms
```

```
✓ Tests passed: 9 of 9 tests – 1 min 8 sec
D:\Code\JAVA\jdk-22\bin\java.exe ...
Successfully init class BookTest.
Successfully connect to database.
Successfully reset database.
Successfully release database connection.
Successfully init class BookTest.
Successfully connect to database.
Successfully reset database.
Successfully release database connection.
Successfully init class BookTest.
Successfully connect to database.
Successfully reset database.
Successfully release database connection.
Successfully init class BookTest.
Successfully connect to database.
Successfully reset database.
Successfully release database connection.
Successfully init class BookTest.
Successfully connect to database.
Successfully reset database.
Successfully release database connection.
Successfully init class BookTest.
Successfully connect to database.
Successfully reset database.
Successfully release database connection.
Successfully init class BookTest.
Successfully connect to database.
```

所有测试点均已通过

4. SQL 注入攻击

SQL 注入指的是：在进行数据交互中，当前端的数据传入后端进行处理时，由于没有做严格的判断，导致其传入的“数据”在拼接到 SQL 语句

中之后，由于其特殊性，被当作 SQL 语句的一部分被执行，从而导致数据库受损（被脱库、被删除、甚至整个服务器权限沦陷）

其主要原因应归结于程序没有细致过滤用户输入的数据，导致数据库受损。

例如，在大学中，使用 Select 语句查询一个 Instructor

```
"select * from instructor where name = ’" + name + "’"
```

如果我们输入：

```
X’ or ’Y’ = ’Y
```

则实际的 SQL 语句则变为：

```
select * from instructor where name = ’X’ or ’Y’ = ’Y’
```

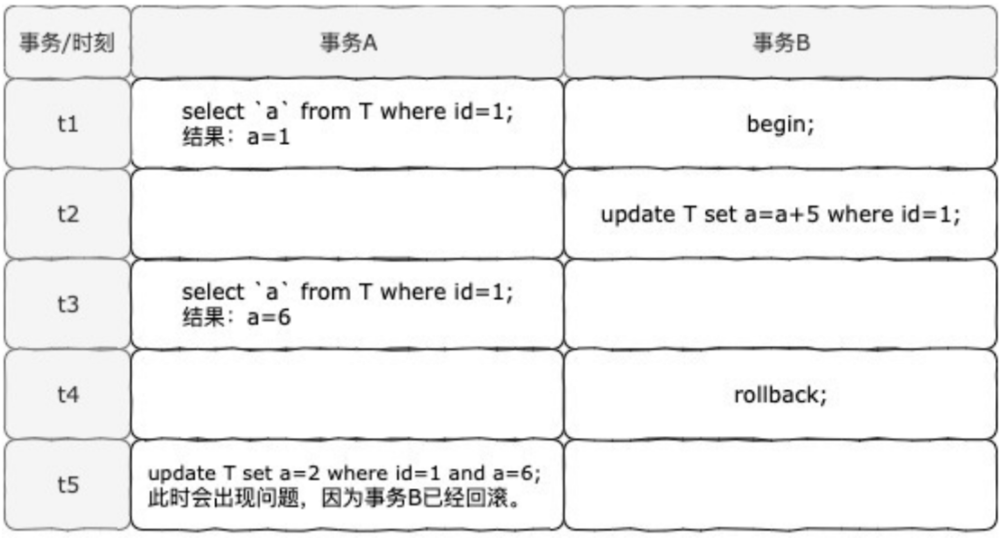
由于`’X’ or ’Y’ = ’Y’` 恒为真，因此此次查询会返回所有 instructor

解决方法：先写 prepare（预编译语句），进行语义分析、语义检查、优化，防止 SQL 注入

5. 并发访问

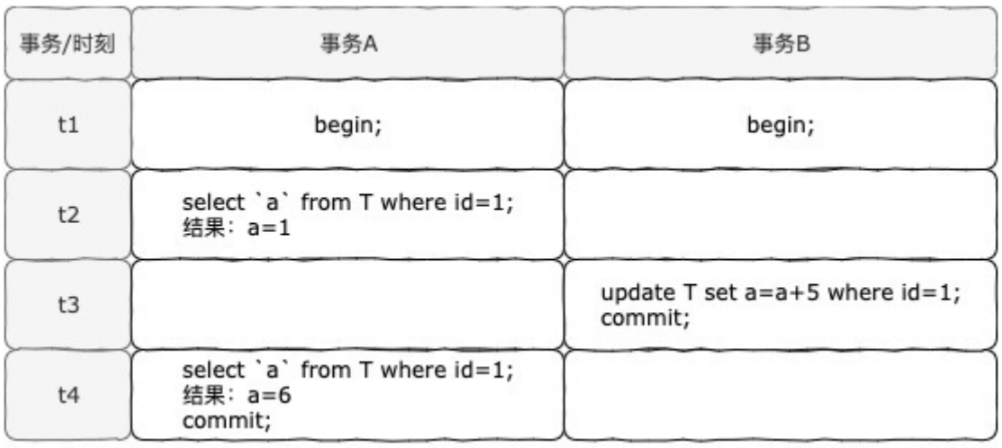
经过学习了解，多事务并发的场景下会出现以下问题：

(a) 脏读：事务 A 执行过程中的一次查询得到了事务 B 修改但是未能成功提交的数据，B 回滚，导致 A 读取到错误数据



事务/时刻	事务A	事务B
t1	select \`a\` from T where id=1; 结果：a=1	begin;
t2		update T set a=a+5 where id=1;
t3	select \`a\` from T where id=1; 结果：a=6	
t4		rollback;
t5	update T set a=2 where id=1 and a=6; 此时会出现问题，因为事务B已经回滚。	

(b) 不可重复读：事务 A 两次查询的间隔中，事务 B 修改了事务 A 查询的数据，导致数据重复读取的结果不一样

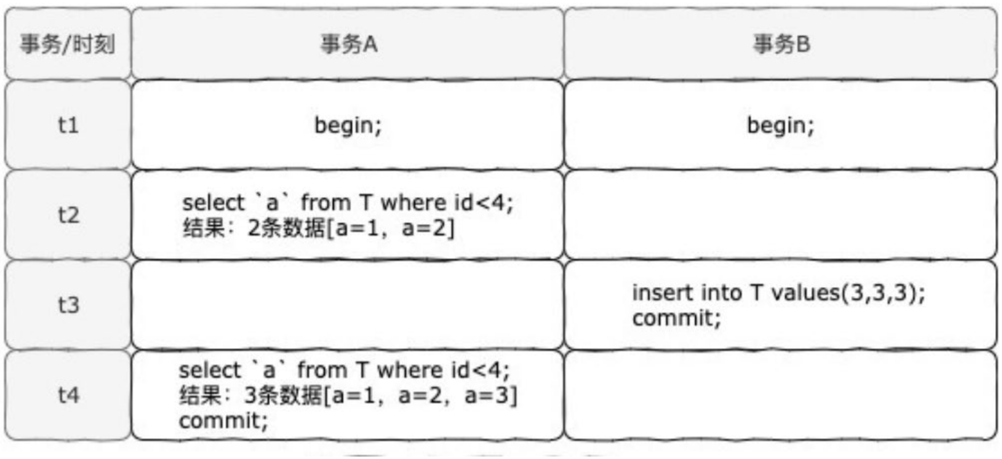


事务/时刻	事务A	事务B
t1	begin;	begin;
t2	select \`a\` from T where id=1; 结果：a=1	

t3		update T set a=a+5 where id=1;
		commit;

t4	select \`a\` from T where id=1;
	结果：a=6
	commit;	

（c）幻读：和不可重复读类似，事务 A 两次查询的间隔中，事务 B 插入了符合事务 A 查询条件的记录并成功提交，导致事务 A 读取的数据不一样



事务/时刻	事务A	事务B
t1	begin;	begin;
t2	select \`a\` from T where id<4; 结果：2条数据[a=1，a=2]	

t3		insert into T values(3,3,3);
		commit;

t4	select \`a\` from T where id<4;
	结果：3条数据[a=1，a=2，a=3]
	commit;	

此外还包括四种标准隔离级别：

（a）读未提交 Read Uncommited

无实际应用

（b）读已提交 Read Commited

绝大多数数据库的隔离级别，Oracle、SQL Server，但不包括 MySQL 的 InnoDB

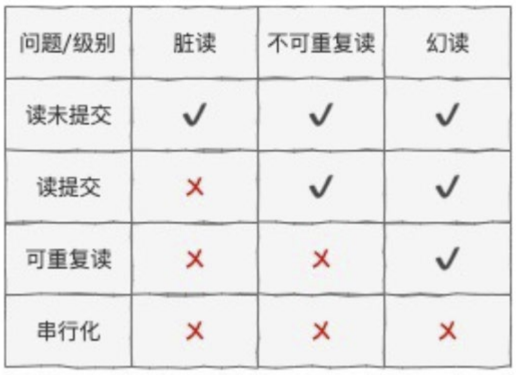
（c）可重复读 Repeatable Read

即 MySQL 的 InnoDB 的默认级别

（d）可串行化 Serializable

最高级别事务隔离，会导致并发性能降低，实际应用较少

每个级别可以应对的多事务并发场景下出现的问题如下：



问题/级别	脏读	不可重复读	幻读
读未提交	✓	✓	✓
读提交	×	✓	✓
可重复读	×	×	✓
串行化	×	×	×

其中，在 MySQL 的 RR 事务隔离级别中，使用 MVCC 机制避免幻读，其具体原理为：通过在每行记录后保存储存该行创建时间和删除时间的两个列，储存数据为版本号，开启一个新事物后会自动递增。

将事务开始时刻的版本号作为该事务的版本号，和查询到的每行记录版本号比较。

InnoDB 对于不同操作的版本号选择：

(a) Select

1. 行的创建版本号小于等于事务版本号，保证了该行是事务开始前已经存在或已经经过事务自身插入或修改。

2. 行的删除版本号大于事务版本号或未定义，保证了当前事务读到该行的时候，该行仍存在（未被删除）

(b) Insert

插入的行保存当前系统的版本号

(c) Delete

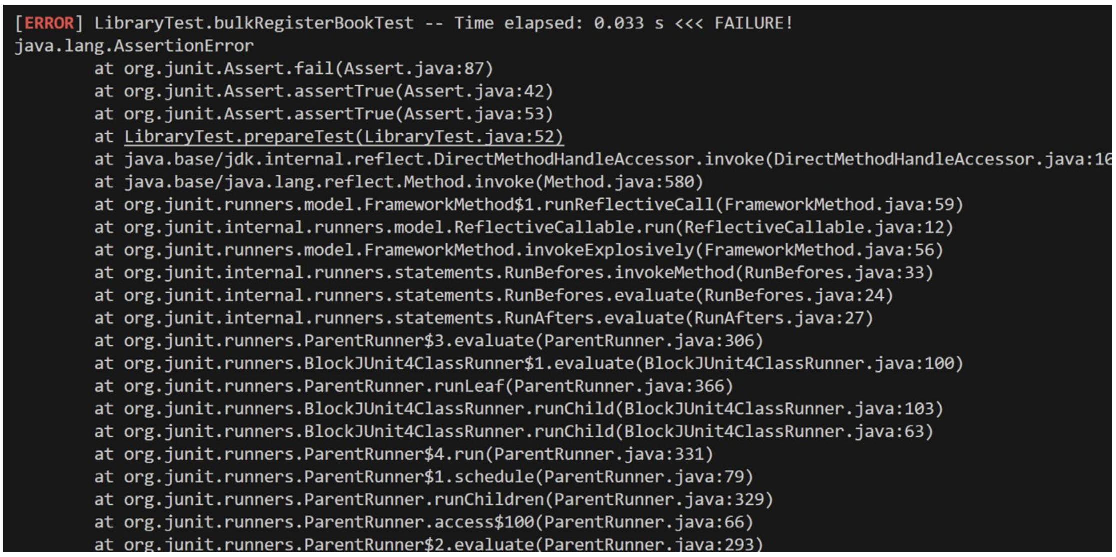
删除的行保存当前系统的版本号

(d) Update

新记录保存当前版本号，原行的删除版本号保存当前系统的版本号

五、 遇到的问题及解决方法

1. 由于本次实验是第一次使用 JAVA 语言完成一个项目，因此对 JAVA 的环境配置，语法结构以及语法特性均不熟悉，根据实验文档的流程，第一次配置 JAVA 环境使用的是较为熟悉的 VSCode，但是由于不熟悉项目结构，不会调试代码：



```
[ERROR] LibraryTest.bulkRegisterBookTest -- Time elapsed: 0.033 s <<< FAILURE!
java.lang.AssertionError
        at org.junit.Assert.fail(Assert.java:87)
        at org.junit.Assert.assertTrue(Assert.java:42)
        at org.junit.Assert.assertTrue(Assert.java:53)
        at LibraryTest.prepareTest(LibraryTest.java:52)
        at java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.java:10
        at java.base/java.lang.reflect.Method.invoke(Method.java:580)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:59)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:56)
        at org.junit.internal.runners.statements.RunBefores.invokeMethod(RunBefores.java:33)
        at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:24)
        at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:27)
        at org.junit.runners.ParentRunner$3.evaluate(ParentRunner.java:306)
        at org.junit.runners.BlockJUnit4ClassRunner$1.evaluate(BlockJUnit4ClassRunner.java:100)
        at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:366)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:103)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:63)
        at org.junit.runners.ParentRunner$4.run(ParentRunner.java:331)
        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:79)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:329)
        at org.junit.runners.ParentRunner.access$100(ParentRunner.java:66)
        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:293)
```

遇到类似这样的问题无能为力，VSCode 调试功能并不友好，后续换为了 Idea，会轻松的得到出问题的测试点位置以及原因，项目得以继续进行

2. 实验初期，在完成第一个函数后，直接运行了测试代码，但所有测试点均报错，后来了解到需要写完所有函数才能进行测试

六、 总结

本次实验通过图书管理系统中各种操作的实现，初步掌握了数据库应用开发的设计方法，同时熟悉了 JAVA 语言以及 JDBC 语法，对数据库的结构于特性有了更深入的认知。总体来说，本次实验难度适中，实验文档的指导也很完善，收获颇丰！