浙江大学

数据库系统实验报告

作业名称：数据库程序设计

学　　号：3220103501

电子邮箱：1436572990@qq.com

联系电话：13567793981

指导老师：孙建伶

2024 年 4 月 24 日

1、实验目的

- 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、实验环境

操作系统：Windows11 22H2。

开发环境：VSCode + JDK 17.0.1 + Maven 3.8.4 + MySQL 8.0.27。

3、实验流程

3.1 接口部分

3.1.1 环境配置

从 https://git.zju.edu.cn/zjucsdb/librarymanagementsystem 下载代码，安装 Maven 和 JDK，配置环境变量。然后我们使用 VSCode + Java 扩展进行本次实验的进行和开发。

代码文件夹中 `src/main/resources` 目录下存放了数据库连接的相关配置以及 SQL 脚本的，在开始开发代码前，我们将 `src/main/resources/application_template.yaml` 文件拷贝一份，并重命名为 `application.yaml`，然后按需修改该文件内的配置参数。最终我的 yaml 文件如下：

```
host: "localhost"
port: "3306"
user: "sanaka"
password: 隐藏了
db: "library"
type: "mysql"
```

然后进入 MySQL，按照以下代码创建数据库：

```
create table `book` (
    `book_id` int not null auto_increment,
    `category` varchar(63) not null,
    `title` varchar(63) not null,
    `press` varchar(63) not null,
    `publish_year` int not null,
    `author` varchar(63) not null,
    `price` decimal(7, 2) not null default 0.00,
    `stock` int not null default 0,
    primary key (`book_id`),
    unique (`category`, `press`, `author`, `title`, `publish_year`)
);

create table `card` (
    `card_id` int not null auto_increment,
    `name` varchar(63) not null,
    `department` varchar(63) not null,
    `type` char(1) not null,
    primary key (`card_id`),
    unique (`department`, `type`, `name`),
    check ( `type` in ('T', 'S') )
);

create table `borrow` (
  `card_id` int not null,
  `book_id` int not null,
  `borrow_time` bigint not null,
  `return_time` bigint not null default 0,
  primary key (`card_id`, `book_id`, `borrow_time`),
  foreign key (`card_id`) references `card`(`card_id`) on delete cascade on update cascade
  foreign key (`book_id`) references `book`(`book_id`) on delete cascade on update cascade
);
```

环境配置就此完成。

3.1.2 代码实现

3.1.2.1 图书入库模块

首先我们实现图书入库模块，即 `storeBook` 方法。该方法的功能是向图书库中注册(添加)一本

新书，并返回新书的书号。如果该书已经存在于图书库中，那么入库操作将失败。其中图书相同的判定条件为：书的<类别, 书名, 出版社, 年份, 作者>均相同。插入完成后，需要根据数据库生成的 `book_id` 值去更新 `book` 对象里的 `book_id`。

由于还要判断是否有相同的书本，我们在这次事务中进行两次 SQL 调用。第一次查询使用 `checkStmt`，用于检查是否有相同的书本；第二次插入使用 `insertStmt`，用于插入新的书本。

> 事实上后来我发现，由于数据库定义了 `unique` 约束，我们可以直接插入数据，如果有重复的数据，数据库会自动返回错误。在后面的批量入库中为了提高效率我就没有写这段代码。

我们选择使用 JDBC 中的 `PreparedStatement` 来执行 SQL 语句，这样不仅方便代码编写，还能增加可读性，最重要的是可以避免 SQL 注入攻击。

对于并发问题，我们选择使用 `conn.setAutoCommit(false)` 和 `conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)` 来设置事务的隔离级别，这样可以避免并发问题。我在所有的修改操作中都使用了这两个方法，**也以此通过了最后一个测试点**。

根据课堂上学到的 JDBC 框架，我们可以编写如下代码：

```java
@Override
public ApiResult storeBook(Book book) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement checkStmt = conn.prepareStatement(
            "SELECT COUNT(*) FROM book WHERE category = ? AND title = ? AND press = ? AND
        );
        checkStmt.setString(1, book.getCategory());
        checkStmt.setString(2, book.getTitle());
        checkStmt.setString(3, book.getPress());
        checkStmt.setInt(4, book.getPublishYear());
        checkStmt.setString(5, book.getAuthor());
        ResultSet rs = checkStmt.executeQuery();
        if (rs.next() && rs.getInt(1) > 0) {
            return new ApiResult(false, "Book already exists.");
        }

        PreparedStatement insertStmt = conn.prepareStatement(
            "INSERT INTO book (category, title, press, publish_year, author, price, stock
            Statement.RETURN_GENERATED_KEYS
        );
        insertStmt.setString(1, book.getCategory());
        insertStmt.setString(2, book.getTitle());
        insertStmt.setString(3, book.getPress());
        insertStmt.setInt(4, book.getPublishYear());
        insertStmt.setString(5, book.getAuthor());
        insertStmt.setDouble(6, book.getPrice());
        insertStmt.setInt(7, book.getStock());
        insertStmt.executeUpdate();
        rs = insertStmt.getGeneratedKeys();
        if (rs.next()) {
            book.setBookId(rs.getInt(1));
        }
        commit(conn);
        return new ApiResult(true, "Book stored successfully.");
    } catch (Exception e) {
        rollback(conn);
```

```
            return new ApiResult(false, "Error storing book: " + e.getMessage());
        }
    }
```

由于 MySQL 中书本编号 `book_id` 是自增列，所以我们在插入数据时不需要指定 `book_id` 的值，而是通过 `getGeneratedKeys` 方法获取插入的 `book_id` 值。

注意这里我们利用了 Java 的一个特性。我们在函数里改变了 `book` 对象的 `book_id` 值，这样在函数调用结束后，`book` 对象的 `book_id` 值也会被改变。

3.1.2.2 图书增加库存模块

接下来我们实现图书增加库存模块，即 `incBookStock` 方法。该方法的功能是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。

我们只需要一个 SQL 语句即可完成这个操作，即 `UPDATE book SET stock = stock + ? WHERE book_id = ? AND stock + ? >= 0`。这段代码的意义是：如果库存增量 `deltaStock` 为负数，那么只要库存最终不为负数，就可以更新库存。

同样我们选择相同的框架，代码如下：

```java
@Override
public ApiResult incBookStock(int bookId, int deltaStock) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement stmt = conn.prepareStatement(
            "UPDATE book SET stock = stock + ? WHERE book_id = ? AND stock + ? >= 0"
        );
        stmt.setInt(1, deltaStock);
        stmt.setInt(2, bookId);
        stmt.setInt(3, deltaStock);
        int affectedRows = stmt.executeUpdate();
        if (affectedRows == 0) {
            return new ApiResult(false, "Stock update failed, check book id or insufficien
        }
        commit(conn);
        return new ApiResult(true, "Stock updated successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error updating stock: " + e.getMessage());
    }
}
```

我们在 `ApiResult` 中除了返回 `ok` 字段外，还返回了一个 `message` 字段，用于返回错误信息。在后续测试的调试过程中，我们可以通过对测试代码进行修改，加入 `System.out.println` 语句，来输出错误信息。比如，如果库存不够，我们会返回 `Stock update failed, check book id or insufficient stock.`。

3.1.2.3 图书批量入库模块

接下来我们实现图书批量入库模块，即 `storeBook(List<Book> books)` 方法。该方法的功能是批量入库图书，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）。

我们可以使用 `addBatch` 方法来批量插入数据，这样可以减少 SQL 语句的调用次数，提高效率。在这个方法中，我们需要对每一本书都进行一次判断，如果有一本书插入失败，那么就需要回滚整个事务。

使用 `executeBatch` 方法执行批量插入后，使用 `getGeneratedKeys` 方法来获取插入的 `book_id` 值。注意这里我们需要使用一个 `index` 变量来记录当前插入的书本的索引，以便在插入成功后更新 `book` 对象的 `book_id` 值。代码如下：

```
@Override
public ApiResult storeBook(List<Book> books) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement insertStmt = conn.prepareStatement(
            "INSERT INTO book (category, title, press, publish_year, author, price, stock
            Statement.RETURN_GENERATED_KEYS
        );
        for (Book book : books) {
            insertStmt.setString(1, book.getCategory());
            insertStmt.setString(2, book.getTitle());
            insertStmt.setString(3, book.getPress());
            insertStmt.setInt(4, book.getPublishYear());
            insertStmt.setString(5, book.getAuthor());
            insertStmt.setDouble(6, book.getPrice());
            insertStmt.setInt(7, book.getStock());
            insertStmt.addBatch();
        }

        insertStmt.executeBatch();
        ResultSet rs = insertStmt.getGeneratedKeys();
        int index = 0;
        while (rs.next()) {
            books.get(index).setBookId(rs.getInt(1));
            index++;
        }
        commit(conn);
        return new ApiResult(true, "Books batch stored successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error storing books batch: " + e.getMessage());
    }
}
```

3.1.2.4 图书删除模块

接下来我们实现图书删除模块，即 `removeBook` 方法。该方法的功能是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。

我们需要先查询是否有人尚未归还这本书，然后再删除这本书。我们使用 `checkStmt` 来查询是否有人尚未归还这本书，如果有人尚未归还这本书，那么就返回错误信息。然后我们使用 `deleteStmt` 来删除这本书。

代码如下：

```
@Override
public ApiResult removeBook(int bookId) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement checkStmt = conn.prepareStatement(
            "SELECT COUNT(*) FROM borrow WHERE book_id = ? AND return_time = 0"
        );
        checkStmt.setInt(1, bookId);
        ResultSet rs = checkStmt.executeQuery();
        if (rs.next() && rs.getInt(1) > 0) {
            return new ApiResult(false, "Cannot remove book, it has not been returned by a
        }

        PreparedStatement deleteStmt = conn.prepareStatement(
            "DELETE FROM book WHERE book_id = ?"
        );
        deleteStmt.setInt(1, bookId);
        int affectedRows = deleteStmt.executeUpdate();
        if (affectedRows == 0) {
            return new ApiResult(false, "No book found with given ID or delete failed.");
        }
        commit(conn);
        return new ApiResult(true, "Book removed successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error removing book: " + e.getMessage());
    }
}
```

3.1.2.5 图书修改模块

接下来我们实现图书修改模块，即 `modifyBookInfo` 方法。该方法的功能是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。

我们可以使用 `updateStmt` 来执行 SQL 语句，即 `UPDATE book SET category = ?, title = ?, press = ?, publish_year = ?, author = ?, price =` 。由于框架和操作和之前基本一致，这里不再赘述。

代码如下：

```
@Override
public ApiResult modifyBookInfo(Book book) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement updateStmt = conn.prepareStatement(
            "UPDATE book SET category = ?, title = ?, press = ?, publish_year = ?, author
        );
        updateStmt.setString(1, book.getCategory());
        updateStmt.setString(2, book.getTitle());
        updateStmt.setString(3, book.getPress());
        updateStmt.setInt(4, book.getPublishYear());
        updateStmt.setString(5, book.getAuthor());
        updateStmt.setDouble(6, book.getPrice());
        updateStmt.setInt(7, book.getBookId());
        int affectedRows = updateStmt.executeUpdate();
        if (affectedRows == 0) {
            return new ApiResult(false, "No book found with given ID or update failed.");
        }
        commit(conn);
        return new ApiResult(true, "Book information updated successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error updating book information: " + e.getMessage())
    }
}
```

3.1.2.6 图书查询模块

接下来我们实现图书查询模块，即 `queryBook` 方法。该方法的功能是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。

这是本次实验中最复杂的操作。我们需要根据不同的查询条件拼接 SQL 语句，然后执行查询操作。查询条件在 `BookQueryConditions` 类中定义，我们需要利用这个类中的参数来拼接 SQL

语句。

事实上这个复杂的多条件操作还是可以使用强大的 `prepareStatement` 来完成。我们可以用 `if` 语句遍历查询条件，然后根据不同的条件进行字符串拼接。最后我们需要根据排序条件进行排序。

注意这里有一个细节，就是 `ORDER BY` 不可以使用 `?` 占位符，所以我们需要在拼接 SQL 语句时，手动将 `ORDER BY` 的部分拼接到 SQL 语句中。

> 事实上，类中的 enum 保证了我们的排序条件是合法的，所以我们不需要担心 SQL 注入攻击。

代码如下：

```java
@Override
public ApiResult queryBook(BookQueryConditions conditions) {
    Connection conn = connector.getConn();
    try {
        String sql = "SELECT * FROM book WHERE ";
        List<Object> params = new ArrayList<Object>();
        int flag = 0;
        if (conditions.getCategory() != null) {
            sql += "category = ? AND ";
            params.add(conditions.getCategory());
            flag = 1;
        }
        if (conditions.getTitle() != null) {
            sql += "title LIKE ? AND ";
            params.add("%" + conditions.getTitle() + "%");
            flag = 1;
        }
        if (conditions.getPress() != null) {
            sql += "press LIKE ? AND ";
            params.add("%" + conditions.getPress() + "%");
            flag = 1;
        }
        if (conditions.getMinPublishYear() != null) {
            sql += "publish_year >= ? AND ";
            params.add(conditions.getMinPublishYear());
            flag = 1;
        }
        if (conditions.getMaxPublishYear() != null) {
            sql += "publish_year <= ? AND ";
            params.add(conditions.getMaxPublishYear());
            flag = 1;
        }
        if (conditions.getAuthor() != null) {
            sql += "author LIKE ? AND ";
            params.add("%" + conditions.getAuthor() + "%");
            flag = 1;
        }
        if (conditions.getMinPrice() != null) {
            sql += "price >= ? AND ";
```

```
            params.add(conditions.getMinPrice());
            flag = 1;
        }
        if (conditions.getMaxPrice() != null) {
            sql += "price <= ? AND ";
            params.add(conditions.getMaxPrice());
            flag = 1;
        }

        String sortBy = conditions.getSortBy().getValue();
        String sortOrder = conditions.getSortOrder().getValue();

        if (flag == 1) {
            sql = sql.substring(0, sql.length() - 4);
        } else {
            sql = sql.substring(0, sql.length() - 6);
        }

        sql += " ORDER BY " + sortBy + " " + sortOrder + ", book_id ASC";

        PreparedStatement stmt = conn.prepareStatement(sql);
        for (int i = 0; i < params.size(); i++) {
            stmt.setObject(i + 1, params.get(i));
        }
        ResultSet rs = stmt.executeQuery();
        List<Book> results = new ArrayList<>();
        while (rs.next()) {
            Book book = new Book();
            book.setBookId(rs.getInt("book_id"));
            book.setCategory(rs.getString("category"));
            book.setTitle(rs.getString("title"));
            book.setPress(rs.getString("press"));
            book.setPublishYear(rs.getInt("publish_year"));
            book.setAuthor(rs.getString("author"));
            book.setPrice(rs.getDouble("price"));
            book.setStock(rs.getInt("stock"));
            results.add(book);
        }
        BookQueryResults final_result = new BookQueryResults(results);
```

```
            return new ApiResult(true, final_result);
        } catch (SQLException e) {
            return new ApiResult(false, "Error querying books: " + e.getMessage());
        }
    }
```

3.1.2.7 借书模块

接下来我们实现借书模块，即 `borrowBook` 方法。该方法的功能是根据给定的书号、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。

我们需要先查询是否有人尚未归还这本书，然后再插入这条借书记录。我们使用 `checkStmt` 来查询是否有人尚未归还这本书，如果有人尚未归还这本书，那么就返回错误信息。然后我们使用 `insertStmt` 来插入这条借书记录。

代码如下：

```java
@Override
public ApiResult borrowBook(Borrow borrow) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement checkStmt = conn.prepareStatement(
            "SELECT COUNT(*) FROM borrow WHERE card_id = ? AND book_id = ? AND return_time
        );
        checkStmt.setInt(1, borrow.getCardId());
        checkStmt.setInt(2, borrow.getBookId());
        ResultSet rs = checkStmt.executeQuery();
        if (rs.next() && rs.getInt(1) > 0) {
            return new ApiResult(false, "Cannot borrow book, it has not been returned by
        }

        PreparedStatement insertStmt = conn.prepareStatement(
            "INSERT INTO borrow (card_id, book_id, borrow_time, return_time) VALUES (?, ?
        );
        insertStmt.setInt(1, borrow.getCardId());
        insertStmt.setInt(2, borrow.getBookId());
        insertStmt.setLong(3, borrow.getBorrowTime());
        insertStmt.setLong(4, borrow.getReturnTime());
        insertStmt.executeUpdate();
        commit(conn);
        return new ApiResult(true, "Book borrowed successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error borrowing book: " + e.getMessage());
    }
}
```

3.1.2.8 还书模块

接下来我们实现还书模块，即 `returnBook` 方法。该方法的功能是根据给定的书号、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。

我们需要先查询是否有人尚未归还这本书，然后再更新这条借书记录。我们使用 `checkStmt` 来查询是否有人尚未归还这本书，如果有人尚未归还这本书，那么就返回错误信息。然后我们使用

`updateStmt` 来更新这条借书记录。

代码如下：

```
@Override
public ApiResult returnBook(Borrow borrow) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement checkStmt = conn.prepareStatement(
            "SELECT COUNT(*) FROM borrow WHERE card_id = ? AND book_id = ? AND return_time
        );
        checkStmt.setInt(1, borrow.getCardId());
        checkStmt.setInt(2, borrow.getBookId());
        ResultSet rs = checkStmt.executeQuery();
        if (rs.next() && rs.getInt(1) == 0) {
            return new ApiResult(false, "Cannot return book, it has not been borrowed by
        }

        PreparedStatement updateStmt = conn.prepareStatement(
            "UPDATE borrow SET return_time = ? WHERE card_id = ? AND book_id = ? AND retu
        );
        updateStmt.setLong(1, borrow.getReturnTime());
        updateStmt.setInt(2, borrow.getCardId());
        updateStmt.setInt(3, borrow.getBookId());
        int affectedRows = updateStmt.executeUpdate();
        if (affectedRows == 0) {
            return new ApiResult(false, "No borrow record found with given ID or return fa
        }
        commit(conn);
        return new ApiResult(true, "Book returned successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error returning book: " + e.getMessage());
    }
}
```

3.1.2.9 借书记录查询模块

接下来我们实现借书记录查询模块，即 `showBorrowHistory` 方法。该方法的功能是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。

我们只需要执行一条 SQL 语句即可完成这个操作，即 `SELECT * FROM borrow WHERE card_id = ? ORDER BY borrow_time DESC, book_id ASC`。

代码如下：

```
@Override
public ApiResult showBorrowHistory(int cardId) {
    Connection conn = connector.getConn();
    try {
        PreparedStatement stmt = conn.prepareStatement(
            "SELECT * FROM borrow WHERE card_id = ? ORDER BY borrow_time DESC, book_id ASC
        );
        stmt.setInt(1, cardId);
        ResultSet rs = stmt.executeQuery();
        List<Borrow> results = new ArrayList<>();
        while (rs.next()) {
            Borrow borrow = new Borrow();
            borrow.setCardId(rs.getInt("card_id"));
            borrow.setBookId(rs.getInt("book_id"));
            borrow.setBorrowTime(rs.getLong("borrow_time"));
            borrow.setReturnTime(rs.getLong("return_time"));
            results.add(borrow);
        }
        BorrowQueryResults final_result = new BorrowQueryResults(results);
        return new ApiResult(true, final_result);
    } catch (SQLException e) {
        return new ApiResult(false, "Error querying borrow history: " + e.getMessage());
    }
}
```

3.1.2.10 借书证注册模块

接下来我们实现借书证注册模块，即 `registerCard` 方法。该方法的功能是注册一个借书证，若借书证已经存在，则该操作将失败。借书证存在的判定条件为：姓名、单位、身份均相同。

我们只需要执行一条 SQL 语句即可完成这个操作，即 `INSERT INTO card (name, department, type) VALUES (?, ?, ?)`。

> 我的原始实现中还是使用了两个 SQL 语句，后续有机会我会考虑优化它。

代码如下：

```
@Override
public ApiResult registerCard(Card card) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement checkStmt = conn.prepareStatement(
            "SELECT COUNT(*) FROM card WHERE name = ? AND department = ? AND type = ?"
        );
        checkStmt.setString(1, card.getName());
        checkStmt.setString(2, card.getDepartment());
        checkStmt.setString(3, card.getType());
        ResultSet rs = checkStmt.executeQuery();
        if (rs.next() && rs.getInt(1) > 0) {
            return new ApiResult(false, "Card already exists.");
        }

        PreparedStatement insertStmt = conn.prepareStatement(
            "INSERT INTO card (name, department, type) VALUES (?, ?, ?)"
        );
        insertStmt.setString(1, card.getName());
        insertStmt.setString(2, card.getDepartment());
        insertStmt.setString(3, card.getType());
        insertStmt.executeUpdate();
        commit(conn);
        return new ApiResult(true, "Card registered successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error registering card: " + e.getMessage());
    }
}
```

3.1.2.11 删除借书证模块

接下来我们实现删除借书证模块，即 `removeCard` 方法。该方法的功能是删除借书证，如果该借书证还有未归还的图书，那么删除操作将失败。

我们需要先查询是否有人尚未归还这本书，然后再删除这个借书证。我们使用 `checkStmt` 来查询是否有人尚未归还这本书，如果有人尚未归还这本书，那么就返回错误信息。然后我们使用 `deleteStmt` 来删除这个借书证。

代码如下：

```java
@Override
public ApiResult removeCard(int cardId) {
    Connection conn = connector.getConn();
    try {
        conn.setAutoCommit(false);
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        PreparedStatement checkStmt = conn.prepareStatement(
            "SELECT COUNT(*) FROM borrow WHERE card_id = ? AND return_time = 0"
        );
        checkStmt.setInt(1, cardId);
        ResultSet rs = checkStmt.executeQuery();
        if (rs.next() && rs.getInt(1) > 0) {
            return new ApiResult(false, "Cannot remove card, it has not returned all books
        }

        PreparedStatement deleteStmt = conn.prepareStatement(
            "DELETE FROM card WHERE card_id = ?"
        );
        deleteStmt.setInt(1, cardId);
        int affectedRows = deleteStmt.executeUpdate();
        if (affectedRows == 0) {
            return new ApiResult(false, "No card found with given ID or delete failed.");
        }
        commit(conn);
        return new ApiResult(true, "Card removed successfully.");
    } catch (Exception e) {
        rollback(conn);
        return new ApiResult(false, "Error removing card: " + e.getMessage());
    }
}
```

3.1.2.12 借书证查询模块

最后我们实现借书证查询模块，即 `showCards` 方法。该方法的功能是列出所有的借书证。

我们只需要执行一条 SQL 语句即可完成这个操作，即 `SELECT * FROM card`。

代码如下：

```java
@Override
public ApiResult showCards() {
    Connection conn = connector.getConn();
    try {
        PreparedStatement stmt = conn.prepareStatement(
            "SELECT * FROM card"
        );
        ResultSet rs = stmt.executeQuery();
        List<Card> results = new ArrayList<>();
        while (rs.next()) {
            Card card = new Card();
            card.setCardId(rs.getInt("card_id"));
            card.setName(rs.getString("name"));
            card.setDepartment(rs.getString("department"));
            card.setType(rs.getString("type"));
            results.add(card);
        }
        CardQueryResults final_result = new CardQueryResults(results);
        return new ApiResult(true, final_result);
    } catch (SQLException e) {
        return new ApiResult(false, "Error querying cards: " + e.getMessage());
    }
}
```

至此，我们已经完成了所有的接口实现。接下来我们将进行测试。

3.1.3 测试

在代码根目录使用命令 `mvn clean compile` 进行编译，然后使用。

测试结果如下：

```
Successfully release database connection.
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 11.85 s -- in LibraryTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  18.323 s
[INFO] Finished at: 2024-04-25T23:26:36+08:00
[INFO] ------------------------------------------------------------------------
```

可以发现，我们的所有接口通过了所有的测试。

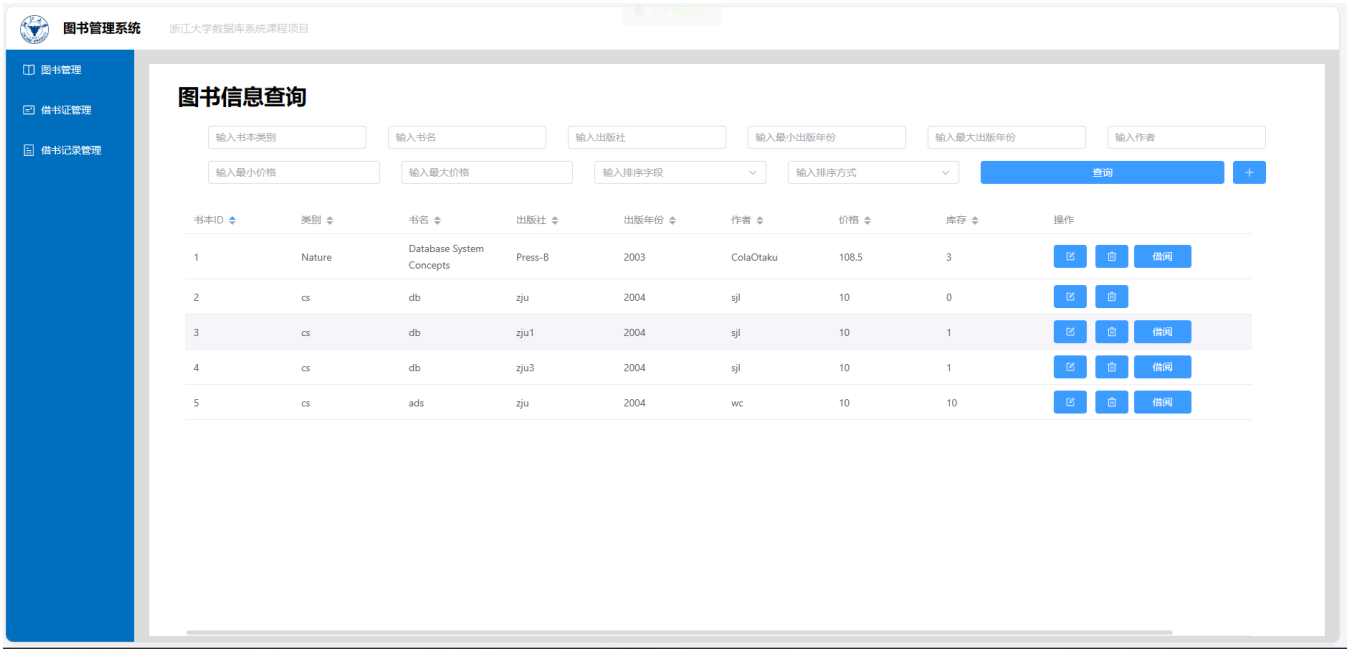
3.2 前后端部分

前端部分我们使用 Vue.js 来实现。我们需要在 `Book.vue`，`Borrow.vue`，`Card.vue` 三个文件中进行前端代码的编写，也需要在后端中的 `Main.java` 中添加对应的接口。由于这不是我们的重点，而且我已经通过验收，所以我们不会详细讲解前端代码的编写。如果有需要，可以查看源代码。

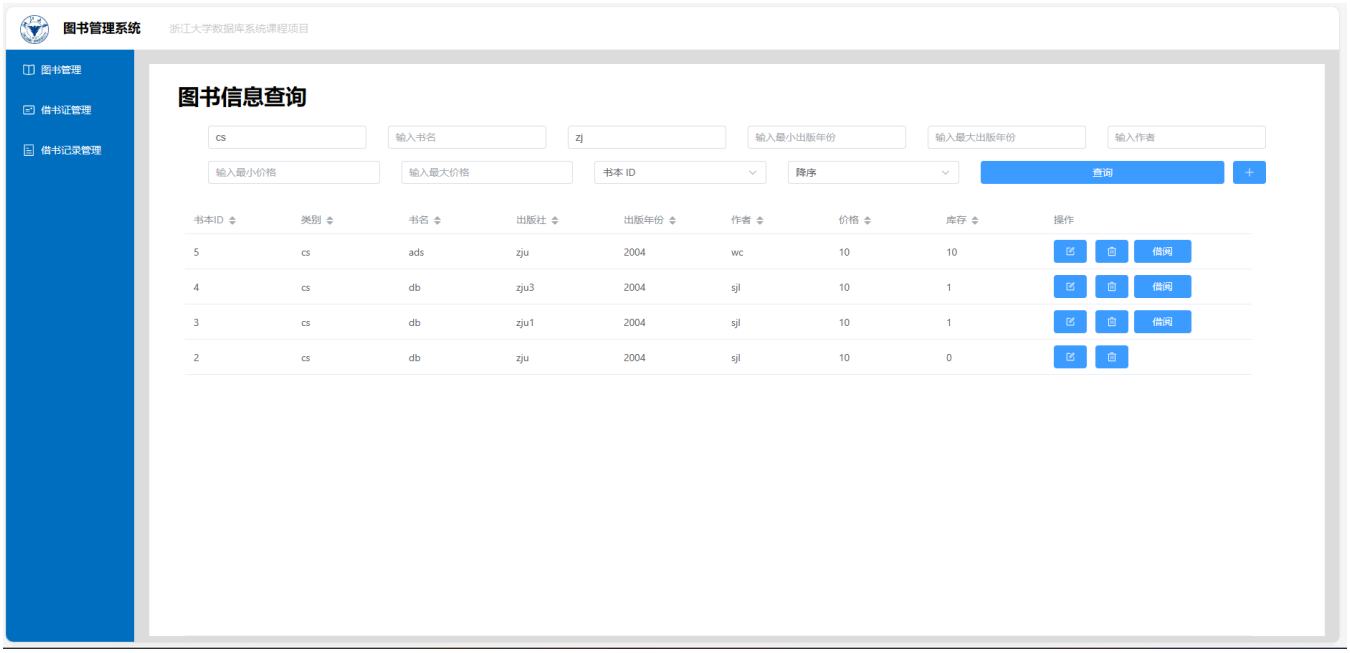
这里我会简单介绍一下我自认为实现的不错的功能：

- 如果有重复的书，那么不能插入。其他的不合法操作也是相同的。

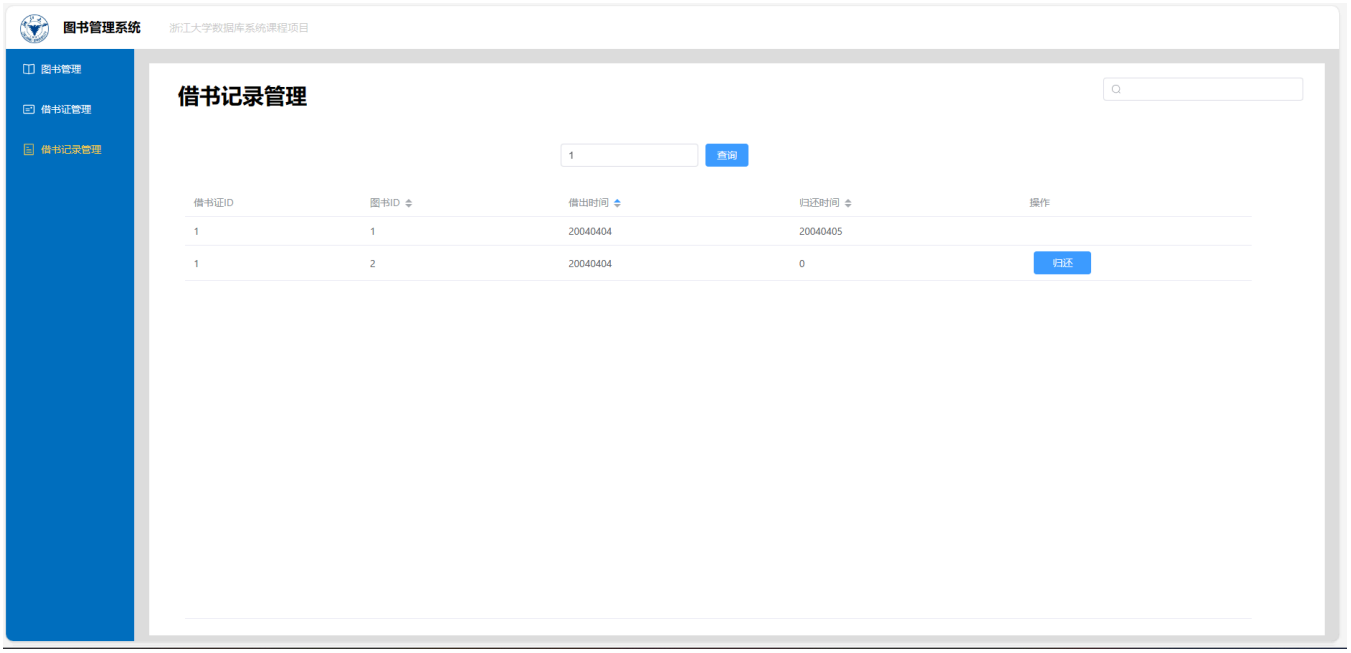
- 如果库存修改后为 0，那么隐藏“借阅”的按钮。

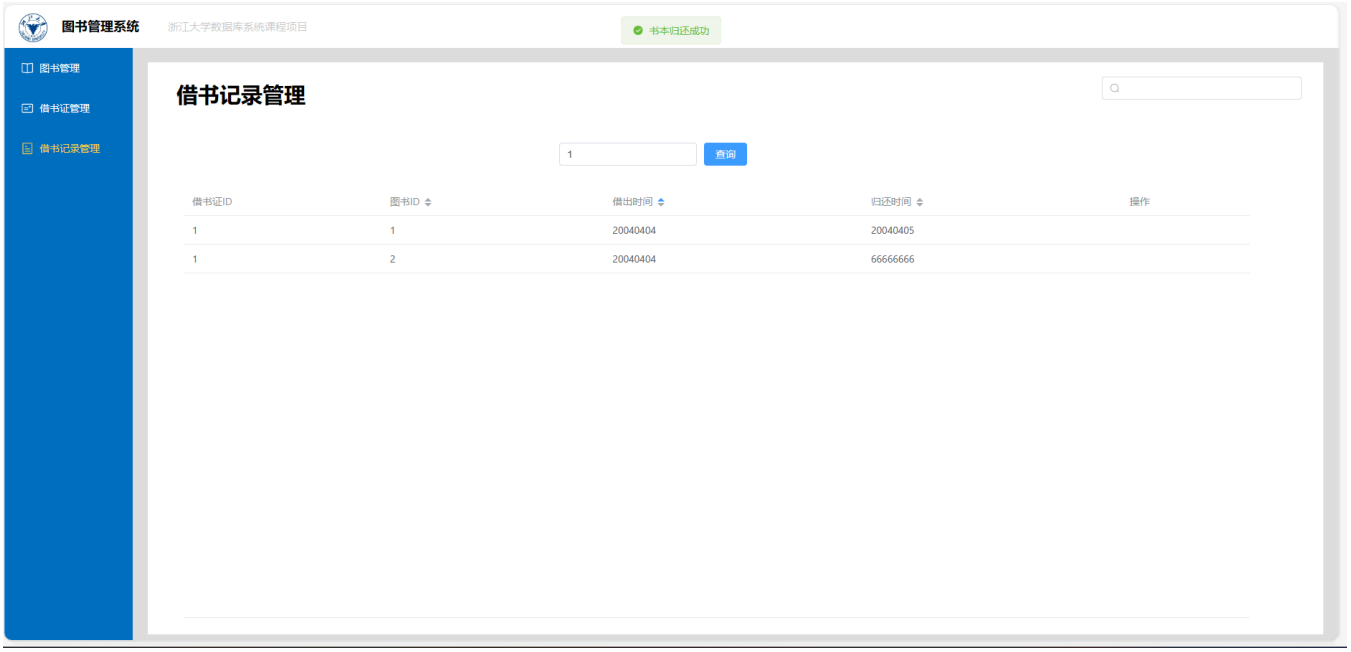


- 能支持多条件的查询。



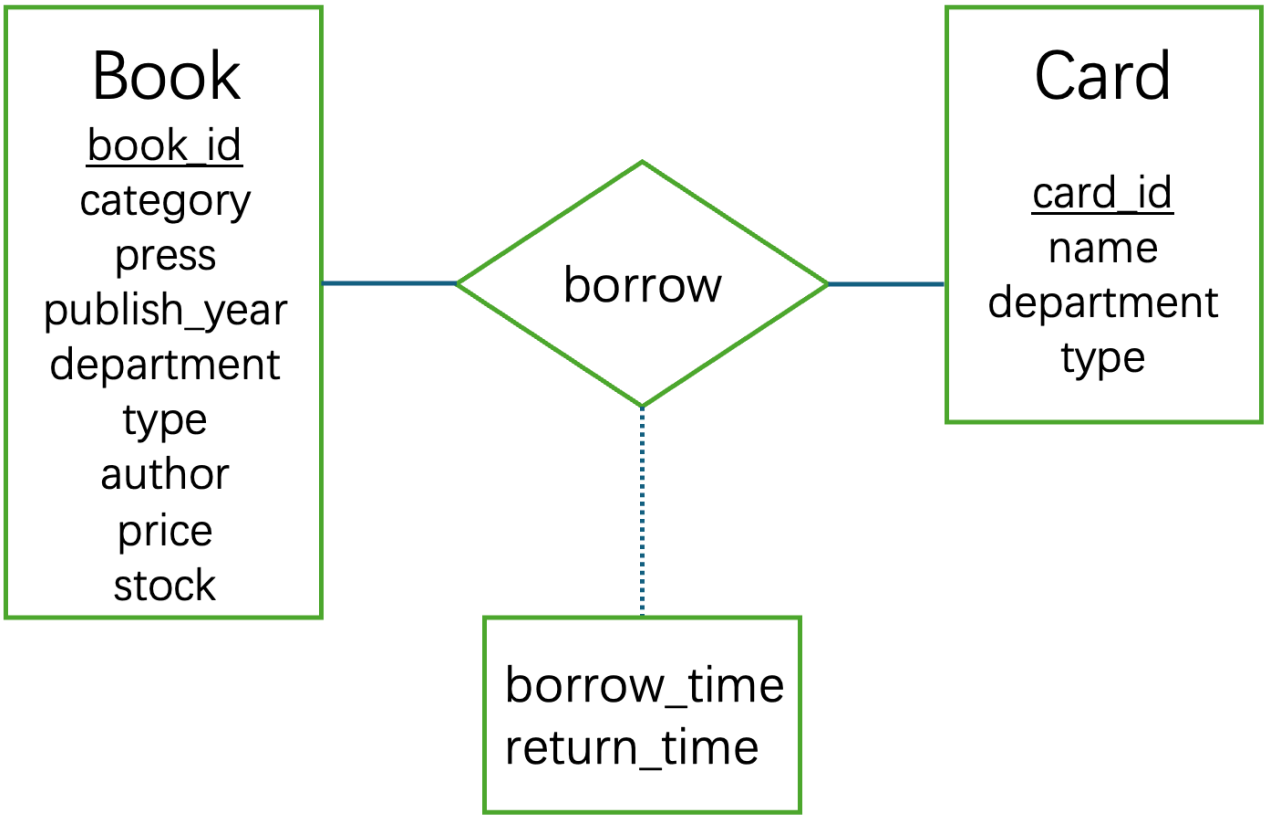
- 如果书本已经归还，隐藏“归还”按钮。





4、思考题

- 绘制该图书管理系统的 **E-R** 图。



- **描述 SQL 注入攻击的原理（并简要举例）。在图书管理系统中，哪些模块可能会遭受 SQL 注入攻击？如何解决？**

SQL 注入攻击是一种常见的网络攻击方式，攻击者通过在输入框中输入恶意 SQL 语句，从而获取数据库中的数据或者进行破坏。SQL 注入攻击的原理是攻击者在输入框中输入恶意 SQL 语句，然后通过程序的漏洞，将这个 SQL 语句传递给数据库，从而对数据库进行一些恶意的操作。

举个例子：假设我们有一个图书管理系统，用户可以通过输入书名来查询图书。如果用户输入的内容没有经过过滤，那么攻击者可以输入 `UPDATE book SET stock = 0 WHERE 1 = 1` 这样的 SQL 语句，从而将所有的图书库存都设置为 0。

在图书管理系统中，可能会遭受 SQL 注入攻击的模块有：图书查询模块、借书记录查询模块、借书证查询模块等。这些模块都涉及到用户输入的内容，如果没有经过过滤，就有可能遭受 SQL 注入攻击。解决 SQL 注入攻击的方法有很多，最常见的方法是使用 `PreparedStatement` 来执行 SQL 语句。`PreparedStatement` 可以预编译 SQL 语句，从而避免 SQL 注入攻击。**事实上我们的代码已经实现了这一点。**

- **在 InnoDB 的默认隔离级别 (RR, Repeated Read) 下，当出现并发访问时，如何保证借书结果的正确性?**

InnoDB 的默认隔离级别 (RR, Repeated Read) 是指：在同一个事务中，多次读取同一数据，如果这个数据在事务执行期间没有被其他事务修改，那么多次读取的结果应该是一样的。在这个隔离级别下，当出现并发访问时，可能会出现以下问题：

1. **脏读**：一个事务读取到了另一个事务未提交的数据。
2. **不可重复读**：一个事务多次读取同一数据，但是每次读取的结果不同。
3. **幻读**：一个事务读取到了另一个事务插入的数据。

为了保证借书结果的正确性，经过资料查阅，我使用了 `Serializable` 隔离级别。`Serializable` 隔离级别是最高的隔离级别，它可以避免脏读、不可重复读和幻读。在 `Serializable` 隔离级别下，事务会被逐个执行，从而避免了并发访问时的问题。

5、遇到的问题及解决方法

在实验过程中，我遇到了一些问题，这里我简单介绍一下：

1. **没有学过 Java 语言**：我本来对 Java 一窍不通，所以在实验开始时，我花费了不少时间来研究 Java 的基本语法和特性。比如在修改 `book_id` 的值时，我一时半会儿不知道该如何进行，是否需要用指针的类似操作。后来我发现 Java 是值传递，所以只需要修改 `book` 对象的 `book_id` 值即可。
2. **事务锁未释放出现死锁**。在设计前端的过程中，我一开始的实现是没有处理完 `POST` 请求后释放锁，导致出现奇怪的现象：更新完以后前端的数据没有更新！但是我用 MySQL 观察后，发现数据确实成功修改了，这也让我不知所措地处理了一个小时。后来我意识到可能是 `conn` 没有释放的问题。于是在我在后端的每一个操作后都添加了 `conn.release()` 语句，成功解决了这个问题。

6、总结

本次实验中，我学习了如何使用 Java 语言和 MySQL 数据库来实现一个简单的图书管理系统。我本来对 Java 一窍不通，但也通过自己对源代码和框架的摸索，最终学会了 Java 的基本语法，也顺利地完成了实验。

对我而言，收获最大的是第一次写出了一个完整的 Vue 前端，这花费了我两个晚上 16 个小时的

时间，当我实现完所有的功能时已经是凌晨 4:30 分了。相信这会成为我毕生难忘的经历，也为我未来的开发之路、科研之路积攒了宝贵的经验。