图书管理系统

Project 5 实验报告
数据库系统 (2024 年 春夏学期)
浙江大学

指导教师：苗晓晔

完成日期：2024 年 5 月 10 日

Made
With LaTeX

目录

第一章　简介

1.1　实验目标

完成基本实验及 Bouns 前后端。

1.2　实验环境

- 服务器：
  - 类型：阿里云轻量应用服务器
  - 配置：2 核 4GB
  - 操作系统：Debian 11
  - 数据库：MySQL Community Server 8.4.0
  - 前端：Node.js 20.13.0
  - 前端页面地址：
  - 后端：OpenJDK 17.0.11
  - 后端 API 地址：
- 本地开发环境：
  - 操作系统：macOS 14.5
  - IDE：IntelliJ IDEA 2023.3.4

第二章　系统设计及实现

2.1　图书管理系统 E-R 图

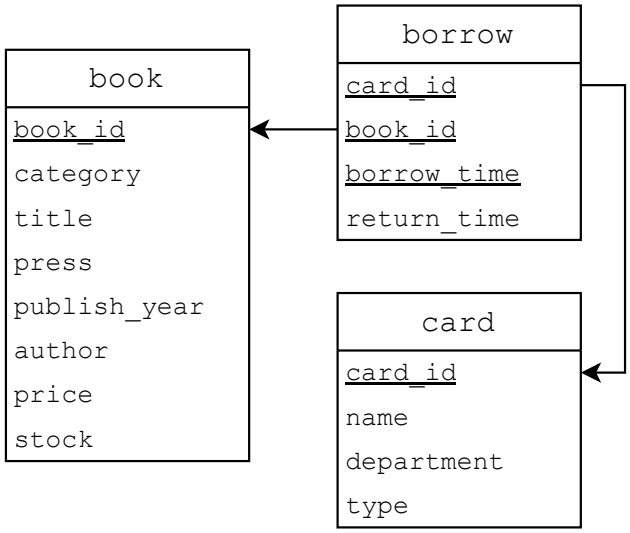


图 2.1: 图书管理系统 E-R 图

2.2　正确性部分：主要函数设计思路

正确性部分主要实现 `LibraryManagementSystemImpl.java` 中的函数，具体要求已在接口定义文件中给出。实现每个函数的步骤基本相同：

- 阅读接口要求，构造 SQL 语句
- 使用 JDBC 创建相应的 `PreparedStatement`
- 从函数调用参数中获取参数，设置 `PreparedStatement` 的参数
- 执行 SQL 语句
- 根据 SQL 语句的执行结果，返回相应的 `ApiResult`

作为示例，代码清单 2.1 是图书存量增加函数的代码：

```
@override
public ApiResult incBookStock(int bookId, int deltaStock) {
    Connection conn = connector.getConn();
    try {
        // make sure stock is always non-negative
```

```
        PreparedStatement stmt = conn.prepareStatement("SELECT stock FROM book WHERE book_id = ?");
        stmt.setInt(1, bookId);
        ResultSet rs = stmt.executeQuery();
        if (!rs.next()) {
            return new ApiResult(false, "Book not found");
        }
        int stock = rs.getInt("stock");
        if (stock + deltaStock < 0) {
            return new ApiResult(false, "Stock will be negative");
        }
        // update stock
        stmt = conn.prepareStatement("UPDATE book SET stock = stock + ? WHERE book_id = ?");
        stmt.setInt(1, deltaStock);
        stmt.setInt(2, bookId);
        stmt.executeUpdate();
        commit(conn);
    } catch (Exception e) {
        rollback(conn);
        LOGGER.log(Level.SEVERE, e.getMessage());
        e.printStackTrace();
        return new ApiResult(false, e.getMessage());
    }
    return new ApiResult(true, null);
}
```

Listing 2.1: 增加图书存量函数

对于并发请求部分，我选择的方法是将隔离级别设置为 TRANSACTION_SERIALIZABLE，这是最高级别的隔离。这意味着每个事务都完全与其他事务隔离，就好像它们是串行执行的一样。

```
public ApiResult borrowBook(Borrow borrow) {
    conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
}
```

Listing 2.2: 并发请求处理

所实现的程序通过了所有测试，结果如图 2.2 所示：

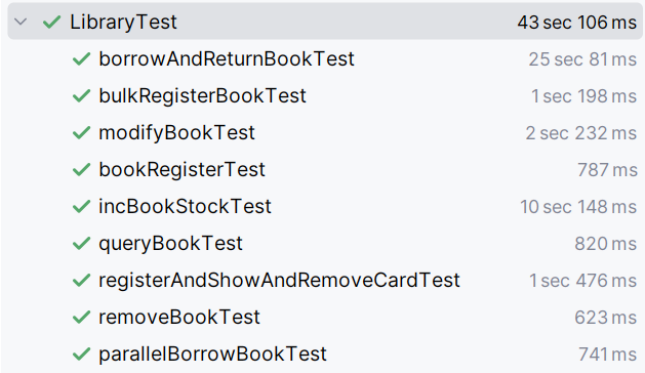
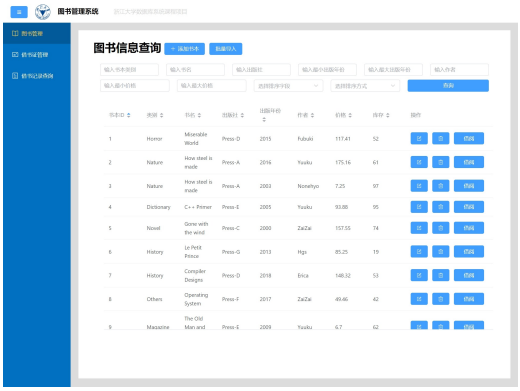


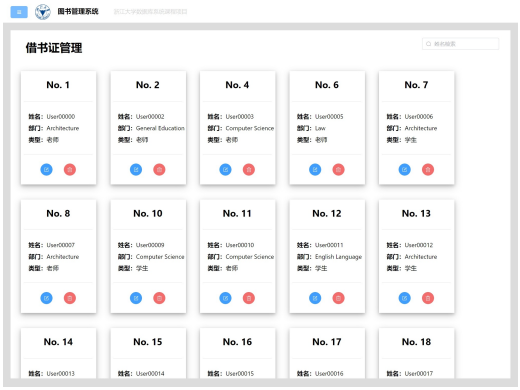
图 2.2: 通过测试截图

2.3 功能性部分：前端设计思路

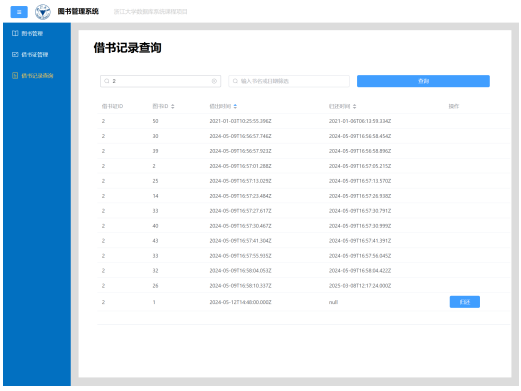
前端采用 Vue + Element-UI 库实现，按照实验指导中的思路结合个人审美，最终设计出的前端界面如下三图所示：



(a) 书籍页面



(b) 借书卡页面



(c) 借书记录页面

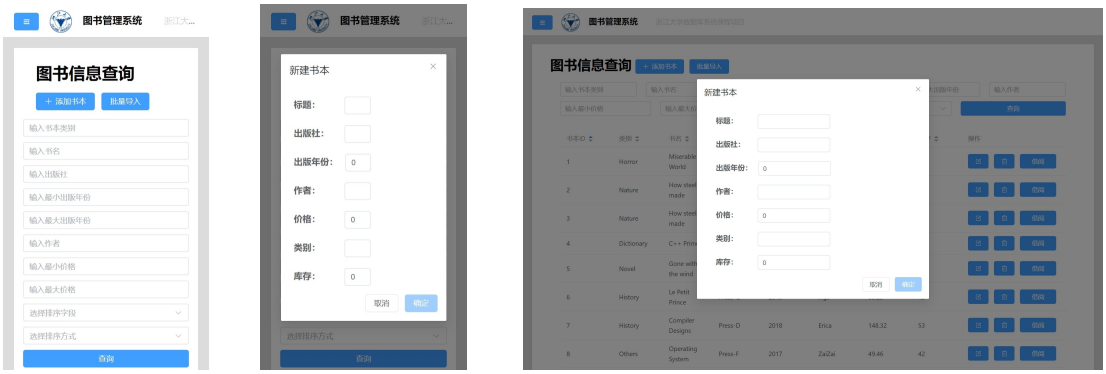
图 2.3: 前端界面设计

我进一步考虑了页面对移动端的兼容性，对整体页面进行了一些改进：

- 为侧边栏添加了展开/折叠按钮。
- 修改 Header 的溢出表现，改进前“浙江大学数据库系统课程项目”等文字会溢出页首下方，影响侧边栏，改进后溢出部分由 ... 省略号替代。
- 使用 `flex` 对页面上的元素灵活布局。
- 使用 `min-width, max-width` 灵活调整对话框的大小，使得桌面端和移动端都能有良好的体验。

最终效果如图 2.4a 所示。具体实现如代码清单 2.3 和 2.4 所示。

```
<!-- 折叠按钮 -->
<el-header>
  <el-button
    type="primary"
    :icon="isCollapse ? 'Expand' : 'Fold'"
    style="color: white; margin-right: 20px; vertical-align: middle"
    @click="toggleSidebar"
  ></el-button>
</el-header>
```



(a) 移动端界面　(b) 移动端对话框（固定）　(c) 桌面端对话框（占比）

图 2.4: 多端适配

```vue
<!-- 折叠属性 -->
<el-container style="width: 100%">
  <el-aside
    :collapse="isCollapse"
    v-bind:style="{ width: isCollapse ? '0%' : '180px' }"
  >
</el-container>
```

Listing 2.3: App.vue

```vue
<!-- flxe 布局 -->
<div
  style="
    display: flex;
    flex-wrap: wrap;
    gap: 10px 20px;
    align-items: center;
    justify-content: space-between;
    width: 90%;
    margin: 0 auto;
    padding-top: 2vh;
  "
>
<el-input
  v-model="this.toQueryCategory"
  style="flex: 1; min-width: 150px"
  size="big"
  placeholder="输入书本类别"
></el-input>
<!-- 比例和定宽结合 -->
<el-dialog
  v-model="newBookVisible"
  title="新建书本"
  style="min-width: 300px; max-width: 40%;"
  align-center
>
```

Listing 2.4: Book.vue

2.4 功能性部分：后端设计思路

后端采用 RESTful API 设计，参考 Web API design best practices - Azure Architecture Center 进行设计。资源及其允许的请求如表 2.1 所示：

表 2.1: RESTful API 设计

资源	GET	POST	PUT	DELETE
/books	查询	入库		
/books/{id}			修改	出库
/cards	查询	注册		
/cards/{id}			修改	注销
/cards/{id}/borrows	查询	借书		还书

约定使用的状态码含义如表 2.2所示：

表 2.2: RESTful 状态码约定

状态码	200	204	404	405	500
含义	动作完成	用于 OPTION 预检	资源不存在	不允许的操作	SQL 服务器错误

接下来是 Java 部分的实现。对于每一个请求，使用 `uri.getPath.split("/")` 分割路径，结合 `Switch` 语句对请求类型进行分类路由，如代码清单 2.5 所示：

```
public void handle(HttpExchange exchange) throws IOException {
    String requestMethod = exchange.getRequestMethod();
    URI uri = exchange.getRequestURI();
    String path = uri.getPath();
    log.log(Level.INFO, "[Web] Request: " + path + " " + requestMethod);
    String[] pathParts = path.split("/");
    switch (pathParts.length) {
        case 2: // /books
        //...
        case 3: // /books/{bookId}
            try {
                Integer.parseInt(pathParts[2]);
            } catch (NumberFormatException e) {
                exchange.sendResponseHeaders(404, -1);
                return;
            }
            switch (requestMethod) {
                case "PUT":
                    booksIdPut(exchange, pathParts);
                    break;
                case "DELETE":
                    booksIdDelete(exchange, pathParts);
                    break;
                case "OPTIONS":// preflight response
                    exchange.getResponseHeaders().set("Access-Control-Allow-Methods", "
    PUT, DELETE");
```

```
                    exchange.sendResponseHeaders(204, -1);
                    break;
                default:
                    exchange.sendResponseHeaders(405, -1);
                    break;
            }
        default:
            exchange.sendResponseHeaders(404, -1);
            break;
    }
}
```

Listing 2.5: 路由

请求的解析涉及消息体或 URL 参数，使用 Java 的库都能轻松解决，如代码清单 2.6 所示：

```
// URL 参数
private static Map<String, String> parseQueryParams(String query) {
    Map<String, String> queryParams = new HashMap<>();
    String[] params = query.split("&")
    for (String param : params) {
        String[] keyValue = param.split("=");
        String key = keyValue[0];
        String value = keyValue.length > 1 ? keyValue[1] : null;
        queryParams.put(key, value);
    return queryParams;
}
private void booksGet(HttpExchange exchange) throws IOException {
    String s = exchange.getRequestURI().getQuery();
    Map<String, String> queryParams = parseQueryParams(s)
    String category = getQueryParam(queryParams, "category");
    //...
}
// JSON 消息体
public Card(JSONObject json) {
    this.cardId = json.getInteger("cardId") == null ? 0 : json.getInteger("cardId");
    this.name = json.getString("name");
    this.department = json.getString("department");
    this.type = CardType.values(json.getString("type"));
}
private void cardsIdPut(HttpExchange exchange, String[] pathParts) throws IOException {
    String requestBody = new BufferedReader(new InputStreamReader(exchange.getRequestBody
    ())).lines().collect(Collectors.joining("\n"));
    Card card = new Card(JSONObject.parseObject(requestBody));
    ApiResult ar = library.modifyCardInfo(card)
    makeResponse(exchange, ar);
}
```

Listing 2.6: 请求解析

HTTP 消息体中携带的信息可以是 JSON 或者错误信息，如代码清单 2.7 展示了一个包装函数 `makeResponse()` 用于根据 MySQL 服务器的返回生成简单的 HTTP 响应。

```
// 简单响应
```

```
private static void makeResponse(HttpExchange exchange, ApiResult ar) throws IOException
    {
    exchange.sendResponseHeaders(ar.ok ? 200 : 500, 0);
    if (!ar.ok) {
        exchange.getResponseHeaders().set("Content-Type", "text/plain");
        OutputStream outputStream = exchange.getResponseBody();
        outputStream.write(ar.message.getBytes());
        outputStream.close();
    }
}
// JSON 消息体
private void cardsGet(HttpExchange exchange) throws IOException {
    ApiResult ar = library.showCards()
    CardList cardList = (CardList) ar.payload;
    JSONArray jsonArray = new JSONArray();
    List<Card> lc = cardList.getCards();
    for (Card c : lc) {
        jsonArray.add(JSONObject.parseObject(c.toJson()));
    exchange.getResponseHeaders().set("Content-Type", "application/json");
    exchange.sendResponseHeaders(200, 0);
    OutputStream outputStream = exchange.getResponseBody();
    outputStream.write(jsonArray.toJSONString().getBytes());
    outputStream.close();
}
```

Listing 2.7: 消息体

对应地，在 Axios 中，将根据消息状态码和消息体进行处理。代码清单 2.8 展示了 Axios 中通用的 HTTP 消息处理模式。值得一提的是，Axios 认为非 2XX 的状态码均表示错误，使用 catch 才能处理。

```
methods: {
  async ConfirmRemoveBook() {
    axios
      .delete(`/books/${this.toRemove}`)
      .then(() => {
        this.Success();
        this.removeBookVisible = false;
        this.QueryBooks();
      })
      .catch((error) => {
        this.ErrorHandling(error);
      });
  },
  Success() {
    ElMessage.success("操作成功");
  },
  ErrorHandling(error) {
    if (error.response) {
      ElMessage.error("操作失败: " + error.response.data);
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else if (error.request) {
```

```
      ElMessage.error("服务器无返回：" + error.request);
      console.log(error.request);
    } else {
      ElMessage.error("网页内部错误：" + error.message);
      console.log("Error", error.message);
    }
  },
}
```

Listing 2.8: Axios

2.5 部署

修改 pom.xml 的 <build> 部分，生成 .jar 文件。

```
<packaging>jar</packaging>
<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>false</filtering>
        </resource>
    </resources>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.4.1</version>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.2.4</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <minimizeJar>true</minimizeJar>
                        <filters>
                            <filter>
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                        <artifactSet>
```

```
                            <excludes>
                                <exclude>junit:junit</exclude>
                                <exclude>jmock:*</exclude>
                                <exclude>*:xml-apis</exclude>
                                <exclude>org.apache.maven:lib:tests</exclude>
                                <exclude>log4j:log4j:jar:</exclude>
                            </excludes>
                        </artifactSet>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Listing 2.9: pom.xml

当 resource 打包进入 .jar 后，不可再视为文件，只能使用流读取，因此 ConnectConfig() 也需要做相应修改：

```
public ConnectConfig() throws FileNotFoundException {
    InputStream stream = ConnectConfig.class.getClassLoader().getResourceAsStream("
    application.yaml");
    if (stream == null) {
        throw new FileNotFoundException();
    }
    InputStreamReader isr = new InputStreamReader(stream);
    Yaml yaml = new Yaml();
    Map<String, Object> objectMap = yaml.load(isr);
    //...
}
```

Listing 2.10: 更改为流读取

修改前端 package.json 使其监听目标为宿主机的请求。修改 main.js，将 Axios 请求定向到后端主机名 https://lab5-api.bowling233.top。

```
// package.json
"scripts": {
  "dev": "vite --host",
  "build": "vite build",
  "preview": "vite preview"
},
// main.js
axios.defaults.baseURL = 'https://lab5-api.bowling233.top';
```

Listing 2.11: 前端配置

为站点创建 Nginx 配置文件进行反向代理，以后端为例，配置文件如代码清单 2.12 所示。我选择将一部分 CORS 控制头（如 Access-Control-Allow-Origin）交给反代服务器，因为这些头和站点本身强相关，应当由反代服务器而不是后端进行处理。

```
server {
    error_log /var/log/nginx/lab5-api.log;
    server_name  lab5-api.bowling233.top;

```

```
    location / {
        add_header 'Access-Control-Allow-Origin' 'https://lab5.bowling233.top' always;
        add_header 'Access-Control-Allow-Credentials' 'true' always;
        add_header 'Access-Control-Allow-Headers' '*' always;
        add_header 'Access-Control-Max-Age' '86400' always;
        #add_header 'Vary' "Accept-Encoding, Origin";
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $proxy_host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/lab5-api.bowling233.top/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/lab5-api.bowling233.top/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}
server {
    if ($host = lab5-api.bowling233.top) {
        return 301 https://$host$request_uri;
    } # managed by Certbot

    listen       80;
    server_name  lab5-api.bowling233.top;
    return 404; # managed by Certbot
}
```

Listing 2.12: Nginx 反代

在服务器上运行前后端：

```
java -cp ./LibraryManagementSystem-FINAL.jar Main
npm run dev
```

Listing 2.13: pom.xml

后端主程序和数据库部分均使用 `java.util.logging` 进行日志记录，能够清晰地了解后端对响应的处理情况。

```
root@iZbp1ah8iba4f0no3ydfe0Z ~/db-lab5# java -cp ./LibraryManagementSystem-FINAL.jar Main
ZJUDB LibMgmt Backend Version Final.
Editor: bowling
INFO: [DB] Success to connect database. [Sun May 12 20:30:22 CST 2024]
May 12, 2024 8:30:22 PM LibraryManagementSystemImpl <init>
INFO: LibraryManagementSystemImpl created
INFO: [Web] Server started on port 8000. [Sun May 12 20:30:22 CST 2024]
INFO: [Web] Request: /books GET [Sun May 12 20:54:01 CST 2024]
INFO: [Web] /books: GET. [Sun May 12 20:54:01 CST 2024]
INFO: [DB] Query for book
BookQueryConditions {category='null', title='null', press='null', minPublishYear=null,
    maxPublishYear=null, author='null', minPrice=null, maxPrice=null, sortBy=BOOK_ID,
    sortOrder=ASC} [Sun May 12 20:54:01 CST 2024]
```

```
INFO: [Web] Request: /cards/0/borrows OPTIONS [Sun May 12 20:54:06 CST 2024]
INFO: [Web] Request: /cards/0/borrows POST [Sun May 12 20:54:06 CST 2024]
INFO: [Web] /cards/{id}/borrows: POST. [Sun May 12 20:54:06 CST 2024]
May 12, 2024 8:54:06 PM LibraryManagementSystemImpl borrowBook
SEVERE: Cannot add or update a child row: a foreign key constraint fails (`library`.`
    borrow`, CONSTRAINT `borrow_ibfk_1` FOREIGN KEY (`card_id`) REFERENCES `card` (`
    card_id`) ON DELETE CASCADE ON UPDATE CASCADE)
java.sql.SQLIntegrityConstraintViolationException: Cannot add or update a child row: a
    foreign key constraint fails (`library`.`borrow`, CONSTRAINT
`borrow_ibfk_1` FOREIGN KEY (`card_id`) REFERENCES `card` (`card_id`) ON DELETE CASCADE
    ON UPDATE CASCADE)
        at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:117)
...
```

Listing 2.14: 日志

第三章　问题与解决方法

3.1　本地 MySQL 并行测试通过，但远程 MySQL 服务器并行测试无法通过

这段代码显然不满足一致性的要求，借书和库存的变化没有作为一个整体。在本地测试时网络延迟低，两个事务能够快速被处理。经过多次测试，使用的 16 个线程都侥幸没有触发错误。但使用远程 MySQL 服务器时，其中就有 3-4 个线程开始出现错误。从这里我们可以看到测试环境和生产环境的不同，这是我们进行测试时需要保持关注的。

决方法是将所有操作合并到一起，在最后进行 commit()。这样在事务隔离为串行级别时能够处理并发请求。在考虑性能和具体应用的场景下，应当使用锁。

```
public ApiResult borrowBook(Borrow borrow) {
    Connection conn = connector.getConn();
    // set serializable isolation level
    try {
        conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        // if user has borrowed the book, return error
        PreparedStatement stmt = conn.prepareStatement("SELECT * FROM borrow WHERE
    card_id = ? AND book_id = ? AND return_time = 0");
        stmt.setInt(1, borrow.getCardId());
        stmt.setInt(2, borrow.getBookId());
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
            return new ApiResult(false, "Book already borrowed");
        }
    } catch (Exception e) {
        rollback(conn);
        LOGGER.log(Level.SEVERE, e.getMessage());
        e.printStackTrace();
        return new ApiResult(false, e.getMessage());
    }
    try{
        // if book is out of stock, return error
        PreparedStatement stmt = conn.prepareStatement("SELECT stock FROM book WHERE
    book_id = ?");
        stmt.setInt(1, borrow.getBookId());
        ResultSet rs = stmt.executeQuery();
        if (!rs.next()) {
            return new ApiResult(false, "Book not found");
        }
        int stock = rs.getInt("stock");
```

```
29          if (stock == 0) {
30              return new ApiResult(false, "Book out of stock");
31          }
32      } catch (Exception e) {
33          rollback(conn);
34          LOGGER.log(Level.SEVERE, e.getMessage());
35          e.printStackTrace();
36          return new ApiResult(false, e.getMessage());
37      }
38      try{
39          // borrow book
40          PreparedStatement stmt = conn.prepareStatement("INSERT INTO borrow (card_id,
    book_id, borrow_time) VALUES (?, ?, ?)");
41          stmt.setInt(1, borrow.getCardId());
42          stmt.setInt(2, borrow.getBookId());
43          stmt.setLong(3, borrow.getBorrowTime());
44          stmt.executeUpdate();
45          commit(conn);
46      } catch (Exception e) {
47          rollback(conn);
48          LOGGER.log(Level.SEVERE, e.getMessage());
49          e.printStackTrace();
50          return new ApiResult(false, e.getMessage());
51      }
52      try{
53          // decrease stock
54          PreparedStatement stmt = conn.prepareStatement("UPDATE book SET stock = stock - 1
     WHERE book_id = ?");
55          stmt.setInt(1, borrow.getBookId());
56          stmt.executeUpdate();
57          commit(conn);
58      } catch (Exception e) {
59          rollback(conn);
60          LOGGER.log(Level.SEVERE, e.getMessage());
61          e.printStackTrace();
62          return new ApiResult(false, e.getMessage());
63      }
64      return new ApiResult(true, null);
65  }
```

Listing 3.1: Q1

3.2 构建和运行 jar 文件

因为计划部署到服务器上，所以需要构建一个 `.jar` 文件。当然，能直接构建 Docker 更好，但查到的资料不是很多。

由于是第一次构建 Java 项目，对 Maven 也不熟，浪费了不少时间在上面，尝试了很多插件和依赖处理办法等。遇到过的错误有：

- 各种编译和依赖问题：`Invalid signature file digest for Manifest main attributes...`。

- Maven test 出现问题 `The forked VM terminated without properly saying googbye. VM crash ...`，只能跳过。
- `.jar` 文件中只有 `Main`，没有其他依赖。
- 在前文提及的 `resource` 放置在 `.jar` 文件中如何读取的问题。
- `java -jar` 找不到 `MANIFEST` 或找不到其中的类 `Error: Could not find or load main class Main while ex`

最后的解决办法是使用 `maven-shade-plugin` 插件对依赖进行处理、删除某些签名的 `jar` 文件的签名，具体的配置已在前文展示。运行时，使用 `java -cp` 指定 ClassPath。

希望实验手册能够对 Maven 的构建过程特别是 `.jar` 文件的生成给出一些必要的提示。

3.3 前端无法收到 2XX 以外消息的消息体

在约定中，500 表示 SQL 服务器产生了错误信息，错误信息应当以文本的形式作为消息体传递给前端并展示。但是测试时，Axios 的 `error.response` 是空的，表现为 Network Error，这非常奇怪。

使用浏览器开发者工具，控制台显示 CORS 被拒，浏览器也确实没有接受消息体。可是我已经在 Nginx 中配置好了 CORS，怎么回事呢?

很可惜配置了 HTTPS 所以 WireShark 无法抓包。因为不是很信任浏览器说的话，我又使用 Postman 构造请求，结果响应头中确实没有 CORS。从而断定是 Nginx 的问题。

查阅资料得知：Unfortunately `add_header` won't work with status codes other than 200, 204, 301, 302 or 304. 在 Nginx 为响应的选项添加 `always` 即可解决改问题。

3.4 Axios 是如何看待状态码的？

起初我试图在 `.then()` 中使用 `response.status` 进行判断，为 500 错误码设置弹窗。但是，怎么都弹不出来。打印到控制台，发现根本没进 `.then()`。

Axios 如何 Handle 非 2XX 状态码这个问题在 GitHub 上有不少讨论,有人使用 `config` 修改 `validateStatus`，因为 Axios 基于它判定异常处理，不过被认为是 Hack 而不建议泛用。所以我也选择了使用异常处理。[1]

不过这也让我思考在 RESTful API 中状态码的意义。它究竟应该代表对对象操作的执行情况，还是用于表示网络通信情况呢? 如果我收到了 500 的回复，是网络通信、基础设施故障的问题还是对对象的操作没有成功?

此外，使用 Axios 发送 DELETE 请求时，携带请求体的格式也与其他请求不一样，需要注意。[2]

3.5 在服务器上运行一段时间后，后端响应全部失败

部署在服务器上，一段时间后访问发现后端返回值为空。登录服务器查询日志:

```
Caused by: com.mysql.cj.exceptions.CJCommunicationsException: The last packet
    successfully received from the server was 38,363,785 milliseconds ago. The last
    packet sent successfully to the server was 38,363,787 milliseconds ago. is longer
    than the server configured value of 'wait_timeout'. You should consider either
    expiring and/or testing connection validity before use in your application,
    increasing the server configured values for client timeouts, or using the Connector/J
     connection property 'autoReconnect=true' to avoid this problem.
```

Listing 3.2: 日志

在设计时，我出于性能（和服务器资源）的考虑，选择让程序启动时就与 MySQL 服务器建立连接，之后持续复用连接。我看到身边有同学选择在每次收到请求时建立连接，感觉这种做法的适用性比较局限。然而 MySQL 是有超时限制的，因此需要修改 `DatabaseType.java` 中的 `url` 方法，添加 `?autoReconnect=true` 即可。

3.6 时间戳问题

查看源码可以发现，单元测试中在数据库存储的时间是 UNIX 时间戳，将其直接展示到前端是非常不友好的。考虑在后端 `toJson` 函数将其转换为 ISO 8601 表示如 `yyyy-mm-dd hh:mm:ss`。让 Copilot 写了一通，看起来不错。用户在前端进行输入时，转换为 UNIX 时间戳发回后端查询，也挺好。

结果在还书测试时失败了：Book Not Borrowed。这条借书记录明明就在表上，怎么会查不到呢？

所幸调试前端时在控制台打印了转换的时间戳，发现后三位始终为 0。原来毫秒级别的 UNIX 时间戳转换为 ISO 8601 应当完整地表示为如 `2024-05-12T14:48:06.331Z` 的格式，这一信息从后端发出时就已丢失，当然无法找到对应的借书记录。Copilot 再背一锅。

最后选择在前后端之间传递时间戳，由前端统一处理 UNIX 时间戳与 ISO 8601 的转换。

3.7 注入攻击

做 Lab 的同学应该都能意识到 SQL 注入的危害，并使用参数化查询进行规避。但是，对于课外的 JSON 可能就会忽略。我就犯了这样的错误，被同学发现了：

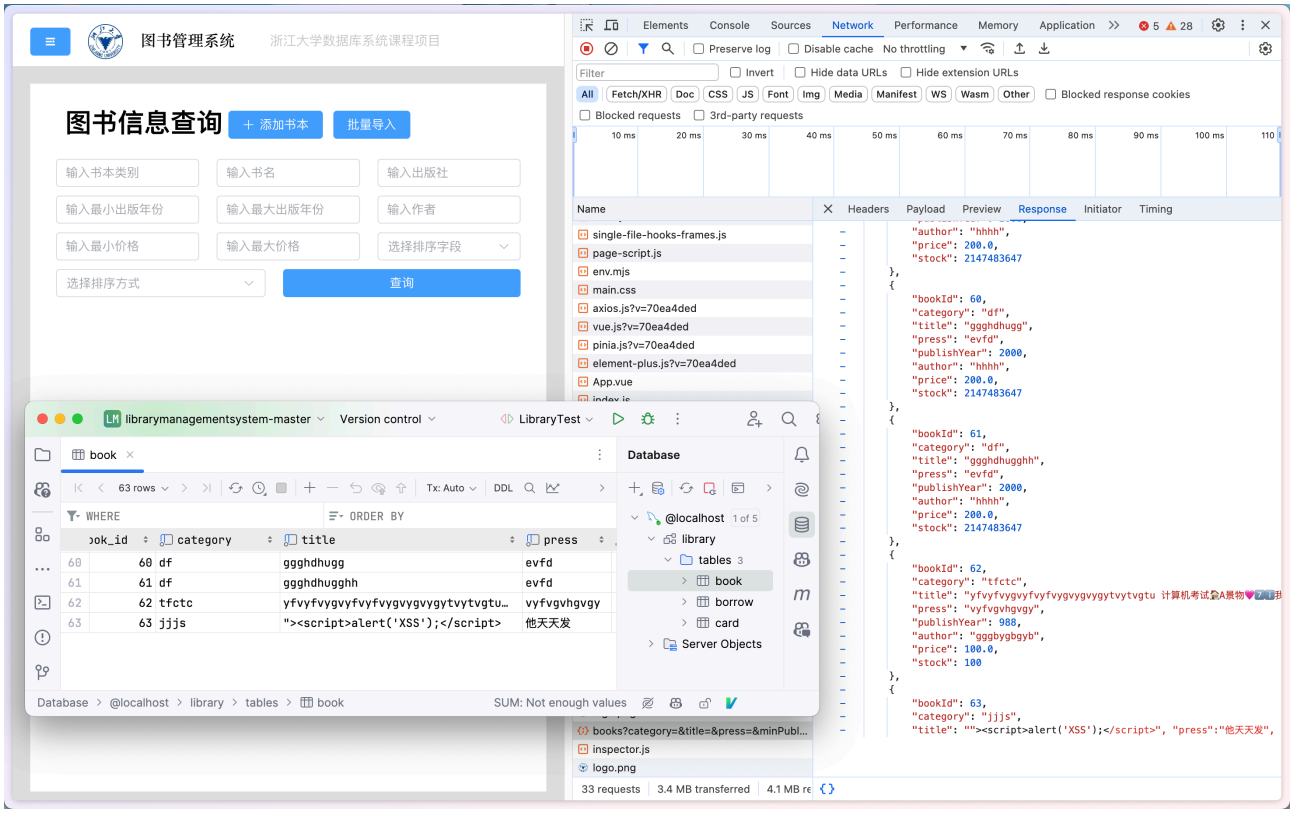
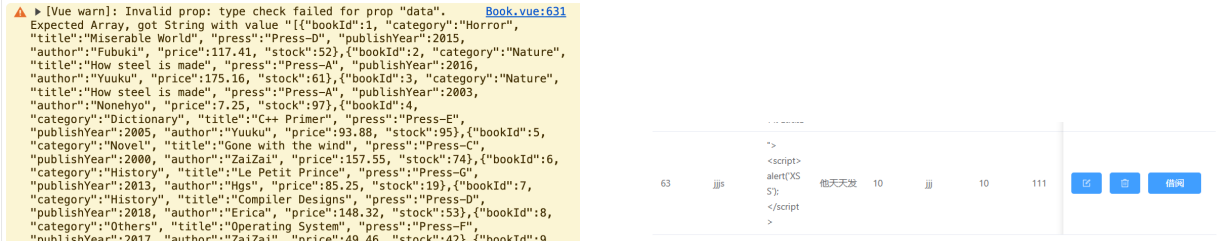


图 3.1: JSON “注入攻击”

同学发起了书名为 `"><script>alert('XSS');</script>` 的提交，成功进入数据库。但是我从数据库返回的结果构建 JSON 使用的是字符串拼接，从上图可以看到消息体中的 JSON 完全乱套了，Vue 在控制

台中提示数据错误如 3.2a 所示，图书页面的表格无法显示。将构建 JSON 的方法改为 `JSONObject.put()` 即可解决问题，正常显示。



(a) Vue 数据错误警告　　(b) 正确显示注入数据

图 3.2: 警告和解决成功图示

3.8 502 Bad Gateway

我们并没有约定这个错误码的含义，所以是网络通信的问题。

这个问题一般是在 Java 后端在请求处理时抛出异常，没有回复，Nginx 等待超时后自动中断并回复 502。查看 Nginx 日志可以了解问题细节，并给予针对性的修复。

```
root@iZbp1ah8iba4f0no3ydfe0Z /v/w/librarymanagementsystem-frontend# tail -f /var/log/nginx/lab5-api.log                    (base)
2024/05/12 07:29:48 [error] 949#949: *281 upstream prematurely closed connection while reading response header from upstream, client: 183.157.163.15
7, server: lab5-api.bowling233.top, request: "GET /books HTTP/1.1", upstream: "http://127.0.0.1:8000/books", host: "lab5-api.bowling233.top"
2024/05/12 07:29:50 [error] 949#949: *290 upstream timed out (110: Connection timed out) while reading upstream, client: 183.157.163.157, server: la
b5-api.bowling233.top, request: "POST /books HTTP/1.1", upstream: "http://127.0.0.1:8000/books", host: "lab5-api.bowling233.top", referrer: "https:/
/lab5.bowling233.top/"
2024/05/12 07:32:37 [error] 949#949: *316 upstream timed out (110: Connection timed out) while reading upstream, client: 183.157.163.157, server: la
b5-api.bowling233.top, request: "POST /books HTTP/1.1", upstream: "http://127.0.0.1:8000/books", host: "lab5-api.bowling233.top", referrer: "https:/
/lab5.bowling233.top/"
2024/05/12 07:45:40 [error] 949#949: *325 upstream timed out (110: Connection timed out) while reading upstream, client: 183.157.163.157, server: la
b5-api.bowling233.top, request: "POST /books HTTP/1.1", upstream: "http://127.0.0.1:8000/books", host: "lab5-api.bowling233.top", referrer: "https:/
/lab5.bowling233.top/"
2024/05/12 16:47:00 [error] 958#958: *1879 upstream prematurely closed connection while reading response header from upstream, client: 183.157.163.1
31, server: lab5-api.bowling233.top, request: "POST /cards HTTP/1.1", upstream: "http://127.0.0.1:8000/cards", host: "lab5-api.bowling233.top", refe
rrer: "https://lab5.bowling233.top/"
2024/05/12 16:57:46 [error] 958#958: *1960 upstream prematurely closed connection while reading response header from upstream, client: 183.157.163.1
31, server: lab5-api.bowling233.top, request: "DELETE /cards/1/borrows HTTP/1.1", upstream: "http://127.0.0.1:8000/cards/1/borrows", host: "lab5-api
.bowling233.top", referrer: "https://lab5.bowling233.top/"
2024/05/12 16:58:44 [error] 958#958: *1960 upstream prematurely closed connection while reading response header from upstream, client: 183.157.163.1
31, server: lab5-api.bowling233.top, request: "DELETE /cards/1/borrows HTTP/1.1", upstream: "http://127.0.0.1:8000/cards/1/borrows", host: "lab5-api
.bowling233.top", referrer: "https://lab5.bowling233.top/"
2024/05/12 17:13:49 [error] 958#958: *2008 upstream prematurely closed connection while reading response header from upstream, client: 183.157.163.1
31, server: lab5-api.bowling233.top, request: "DELETE /cards/2/borrows HTTP/1.1", upstream: "http://127.0.0.1:8000/cards/2/borrows", host: "lab5-api
.bowling233.top", referrer: "https://lab5.bowling233.top/"
2024/05/12 17:28:54 [error] 954#954: *52 upstream prematurely closed connection while reading response header from upstream, client: 183.157.163.131
, server: lab5-api.bowling233.top, request: "DELETE /cards/2/borrows HTTP/1.1", upstream: "http://127.0.0.1:8000/cards/2/borrows", host: "lab5-api.b
owling233.top", referrer: "https://lab5.bowling233.top/"
2024/05/12 17:34:00 [error] 954#954: *158 upstream timed out (110: Connection timed out) while reading response header from upstream, client: 183.15
7.163.131, server: lab5-api.bowling233.top, request: "DELETE /cards/2/borrows HTTP/1.1", upstream: "http://127.0.0.1:8000/cards/2/borrows", host: "l
ab5-api.bowling233.top", referrer: "https://lab5.bowling233.top/"
```

图 3.3: 502 Bad Gateway 的相关日志

第四章　思考题

E-R 图、注入攻击、隔离级别均已在前文介绍和提及，这里不再赘述。作为超算队运维，我负责队内分布式集群管理，因此对思考题中针对电商系统中的高并发活动通常采取的措施更有兴趣。

查阅资料，可以了解到一般有以下措施：

- 限流控制：通过限制每秒钟的请求次数或者并发连接数，来控制系统的访问量，防止系统被过多的请求压垮。
- 队列处理：将请求放入队列中逐个处理，保证每个请求按顺序执行，避免并发访问导致的竞争条件。
- 缓存优化：合理利用缓存技术，减轻数据库的压力，提高系统的响应速度。
- 分布式架构：采用分布式架构，将负载分散到多个服务器上，提高系统的承载能力。
- 数据库优化：针对热点数据进行优化，减少数据库的访问压力，提高系统的并发处理能力。

这其中用到的数据库其实远不止 MySQL 这种关系型数据库，还有 Redis 这种 KV 作为缓存，MongoDB 这种文档型数据库做收集。我们不仅要掌握好课内的关系型数据库，还应当多关注工业界新型存储技术、数据处理技术的发展。

在分布式集群运维中，我们也面临着相似的问题：集群每台机器的日志-> 分析器-> 数据库，其中分析器和数据库都需要解决高并发问题。目前我们采用 Elastic Stack，在分析器前加消息队列集群，数据库使用 Elastic 分布式集群。Elastic 作为文档型数据库，和课内学习的 MySQL 等关系型数据库不同，在分布式集群上的扩展性非常好，能够轻松处理海量数据。

第五章　总结

非常感谢助教 gg 本学期对 Lab 进行的更改，新的 Lab 前后端非常有意思，我身边做了的同学几乎都会在朋友圈晒一晒自己的成果，可以看出大家都从 Lab 中学到了很多。这次 Lab 的开发过程也让我对网络的知识得到了进一步实践，具备了构建、调试基础网络应用程序的能力。

参考文献

[1] IJAZ U. How to make delete requests with axios?[J/OL]. Rapid API Guides. https://rapidapi.com/guides/delete-requests-axios.

[2] AXIOS. Support delete body · issue #897 · axios/axios[J/OL]. GitHub. https://github.com/axios/axios/issues/897.