

# Lab1实验报告

## 实验内容&目标

用自己熟悉的高级语言编写一个数据库初始化程序，将多种来源的外部数据导入MySQL（或Oracle, SqlServer）来掌握高级语言操作数据库的方法

## 实验环境

使用Java语言实现，涉及 JDBC API及 MyBatis 框架

JDK版本：openjdk-22

运行main方法

## 实验实现

### JDBC实现

#### DBHelper.java

输入本地数据库地址与用户密码完成通过jdbc实现的数据库驱动连接

```
package org.example;
import java.sql.Connection;
import java.sql.DriverManager;
public class DBHelper {
    private static final String URL = "jdbc:mysql:my database address";
    private static final String USER = "root";
    private static final String PASSWORD = "mypassword";

    public static Connection getConnection(){
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            return DriverManager.getConnection(URL, USER, PASSWORD);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

#### fileReader.java

`readCsv` 利用io类读取将csv数据文件转换为字符串形式，便于后续的批量数据插入指令的实现

由于room和student两个表的表结构不同且主键设置不同，直接使用了sql建表语句存储在 `.sql` 文件中

`readSQLFromFile` 直接读取 `.sql` 文件，读取sql命令

```
package org.example;
import java.io.BufferedReader;
import java.io.FileReader;
```

```

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
public class FileReader {
    public static List<String[]> readCsv(String filepath){
        List<String[]> records = new ArrayList<>();
        try(BufferedReader br = new BufferedReader(new FileReader(filepath))){
            String line;
            while ((line = br.readLine()) != null) {
                String[] values = line.split(",");
                for (int i = 0; i < values.length; i++) {
                    // 去除每个值周围的双引号
                    values[i] = values[i].trim().replace("\"", "");
                }
                records.add(values);
            }
        }catch (Exception e){
            e.printStackTrace();
        }
        return records;
    }

    public static String readSQLFromFile(String filePath) {
        StringBuilder sql = new StringBuilder();
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                sql.append(line).append(System.lineSeparator()); // 将每一行添加到
                // StringBuilder中
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return sql.toString(); // 将StringBuilder转换为String并返回
    }
}

```

### createTable.sql

根据字段说明，以及csv文件的格式，根据数据完整性约束原则，对缺少数据的 `room.papername` 属性，将其默认值设为 `unknown`

第一次student初始导入时我没有增加主键，运行以下命令后发现存在重复记录，重新查看数据后，确定 `registno` 作为主键

```

SELECT registno, COUNT(*)
FROM lab1.student
GROUP BY registno
HAVING COUNT(*) > 1;

```

```

CREATE TABLE room (
    kdno VARCHAR(50),
    kcno VARCHAR(50),
    ccno VARCHAR(50),
    kdname VARCHAR(255),
    exptime DATETIME,

```

```

        papername VARCHAR(100) DEFAULT 'unknown',
        PRIMARY KEY (kdno, kcno, ccno)
    );
CREATE TABLE student (
    registno VARCHAR(100) PRIMARY KEY,
    name VARCHAR(255),
    kdno VARCHAR(50),
    kcno VARCHAR(50),
    ccno VARCHAR(50),
    seat INT,
    FOREIGN KEY (kdno,kcno,ccno) REFERENCES
    room(kdno,kcno,ccno)
);

```

## DatabaseOperation.java

`executeSQLFile` 方法与 `csvReader` 中的 `readSQLFromFile` 相结合，将文件内容分割成单独的SQL命令，并逐条执行这些命令。

`generateInsertSQL(String tableName, String[] columnNames)` 利用 `StringJoiner` 根据表名和列名数组生成一个 `INSERT IGNORE INTO` SQL语句，其中使用占位符 `?` 代表后续将要插入的值，返回完整的insert sql语句

`insertData(Connection conn, String tableName, String insertSql, List<String[]> dataRows)` 根据生成的 `INSERT` SQL语句创建一个 `PreparedStatement`，遍历CSV文件中的每条数据行，对每个字段值进行处理：如果字段不为空，则插入该值；如果字段为空，则调用 `setNull` 让数据库使用默认值。使用预处理语句批量插入数据到数据库。

`importCsv(String csvFilePath)` 根据CSV文件的路径，确定要导入数据的表名(与csv文件名相同)，使用 `fileReader.readCsv` 方法读取CSV文件内容，将其分解成记录列表，每条记录是一个字符串数组。其中第一行被视为列名，剩余的行被视为数据行。将剩余数据通过 `generateInsertSQL` 和 `insertData` 方法将数据导入指定的数据表中。

```

package org.example;
import java.io.BufferedReader;
import java.io.FileReader;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.sql.*;
import java.util.List;
import java.util.StringJoiner;

public class DatabaseOperations {
    public void executeSQLFile(String sqlFilePath) {
        String sqlCommands = readSQLFromFile(sqlFilePath);

        // 分割SQL命令
        String[] commands = sqlCommands.split(";\\s*\\n");

        try (Connection conn = DBHelper.getConnection();
            Statement stmt = conn.createStatement()) {

            // 逐条执行SQL命令
            for (String command : commands) {
                if (!command.trim().isEmpty()) {
                    stmt.executeUpdate(command);
                }
            }
        }
    }
}

```

```

    }

    System.out.println("SQL file executed successfully.");
} catch (SQLException e) {
    e.printStackTrace();
}
}

public void importCsv(String csvFilePath) {
    Path path = Paths.get(csvFilePath);
    String tableName = path.getFileName().toString().replaceFirst("[.]"
[^.]+$", "");
    System.out.println("开始从CSV文件导入数据: " + csvFilePath + " 到表: " +
tableName);

    List<String[]> records = fileReader.readCsv(csvFilePath);
    if (records.isEmpty()) {
        System.out.println("CSV文件没有数据, 导入过程终止。");
        return;
    }

    String[] columnNames = records.get(0); // The first row contains column
names
    List<String[]> dataRows = records.subList(1, records.size()); // Exclude
the first row
    System.out.println("CSV文件读取完成, 总计数据行数 (不含表头): " +
dataRows.size());

    try (Connection conn = DBHelper.getConnection()) {
        String insertSql = generateInsertSQL(tableName, columnNames);
        insertData(conn, tableName, insertSql, dataRows);
        System.out.println("数据成功导入到表: " + tableName);
    } catch (Exception e) {
        System.err.println("导入CSV数据到数据库时发生错误: " + e.getMessage());
        e.printStackTrace();
    }
}

private String generateInsertSQL(String tableName, String[] columnNames) {
    StringJoiner columns = new StringJoiner(", ", "(", ")");
    StringJoiner placeholders = new StringJoiner(", ", "(", ")");
    for (String columnName : columnNames) {
        columns.add(columnName);
        placeholders.add("?");
    }
    return "INSERT IGNORE INTO " + tableName + " " + columns + " VALUES " +
placeholders;
}

private void insertData(Connection conn, String tableName, String insertSql,
List<String[]> dataRows) throws SQLException {
    try (PreparedStatement pstmt = conn.prepareStatement(insertSql)) {
        int columnsCount = pstmt.getParameterMetaData().getParameterCount();
        for (String[] rowData : dataRows) {
            for (int i = 0; i < columnsCount; i++) {
                if (i < rowData.length && !rowData[i].isEmpty()) {
                    pstmt.setString(i + 1, rowData[i].trim());
                } else {

```

```

        pstmt.setNull(i + 1, Types.VARCHAR); // 允许数据库使用默认值
    }
}
pstmt.addBatch();
}
pstmt.executeBatch(); // 执行批处理
} catch (SQLException e) {
    e.printStackTrace();
    throw e; // 抛出异常供调用者处理
}
}
}
}

```

## Main.java

将方法整合实现

```

package org.example;

import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {
    public static void main(String[] args) {
        DatabaseOperations operations= new DatabaseOperations();

        String csvFilePath1 = "D:\\大学\\大二下\\数据库\\lab\\lab1\\lab1数据\\room.csv";
        String csvFilePath2 = "D:\\大学\\大二下\\数据库\\lab\\lab1\\lab1数据\\student.csv";

        String createSQLFilePath =
            "D:\\softwareEngineering\\dataBaseSystem\\lab1\\src\\main\\java\\org\\example\\createTable.sql";

        operations.executeSQLFile(createSQLFilePath);

        operations.importCsv(csvFilePath1);
        operations.importCsv(csvFilePath2);

        operations.executeSQLFile(removeDuplicateSqlPath);
    }
}

```

## MyBatis实现

MyBatis 实现中，额外定义了 Java 对象模型（Room 和 Student 类），Mapper 接口（RoomMapper 和 StudentMapper），以及相应的 MyBatis Mapper XML 配置文件，来描述如何将这些 Java 对象持久化到数据库中。

1. **Java 对象模型**：定义了 Room 和 Student 两个 Java 类，它们分别对应数据库中的 room 和 student 表。这些类包含与表列对应的属性和相应的 getter/setter 方法。这样的对象模型允许在 Java 代码中方便地操作数据库记录。
2. **Mapper 接口**：

- 定义了与数据库操作相关的接口方法。RoomMapper 接口中的 insertRooms 方法和 StudentMapper 接口中的 insertStudents 方法
- Mapper接口通过MyBatis的Mapper XML文件与具体的SQL语句关联。在这些接口中定义的方法名称与Mapper XML文件中的 id 属性值相匹配。

### 3. Mapper XML文件:

- 包含了实现Mapper接口方法的SQL语句。例如, insertRoom 和 insertStudent 方法对应的 INSERT 语句。
- 使用了MyBatis的 <foreach> 标签来实现批量插入操作, 这标签遍历提供的对象列表, 并为每个对象生成一个 INSERT 语句。

### 4. 导入CSV数据流程:

- CSV文件被解析为 Room 或 Student 对象的列表。
- 通过MyBatis的 SqlSession 获取Mapper接口的实例, 并调用相应的方法 (如 insertRooms 或 insertStudents ), 将数据批量插入到数据库中。

## MyBatis vs JDBC

1. **代码声明的实现**: 原始的实现直接使用JDBC PreparedStatement 来执行SQL语句, 而在MyBatis实现中, 这些操作被抽象化为Mapper接口和XML配置文件中的声明, 减少了SQL代码在Java类中的直接编写, 提高了代码的可维护性和可读性。
2. **批量插入的实现**: 通过MyBatis的 <foreach> 标签实现了批量插入, 使得对于大量数据的插入操作更为简洁和高效。
3. **类型安全和解耦**: 使用MyBatis, 数据模型 (如 Room 和 Student ) 与数据库操作更好地解耦, 同时提供了类型安全的操作, 避免了直接使用JDBC时可能出现的类型匹配错误。
4. **减少SQL注入风险**: 通过使用MyBatis的参数绑定, 减少了SQL注入的风险
5. **简化数据转换逻辑**: 在从CSV文件导入数据时, MyBatis允许直接将Java对象列表持久化到数据库中, 简化了数据转换和插入的逻辑。

使用MyBatis框架可以使得数据库操作更加模块化、类型安全, 同时也使得代码更容易维护和扩展。

## 数据库导入结果

47 rows							Tx: Auto	DDL
WHERE								
	kdno	kcno	ccno	kdname	exptime	papername		
1	1101	1	1	复旦大学	2004-06-10 08:00:00	<null>		
2	1101	1	10	复旦大学	2004-06-10 11:10:00	<null>		
3	1101	1	11	复旦大学	2004-06-10 12:00:00	<null>		
4	1101	1	12	复旦大学	2004-06-10 12:20:00	<null>		
5	1101	1	13	复旦大学	2004-06-10 12:40:00	<null>		
6	1101	1	14	复旦大学	2004-06-10 13:00:00	<null>		
7	1101	1	15	复旦大学	2004-06-10 13:20:00	<null>		
8	1101	1	16	复旦大学	2004-06-10 13:40:00	<null>		
9	1101	1	17	复旦大学	2004-06-10 14:00:00	<null>		
10	1101	1	18	复旦大学	2004-06-10 14:20:00	<null>		
11	1101	1	19	复旦大学	2004-06-10 14:40:00	<null>		
12	1101	1	2	复旦大学	2004-06-10 08:20:00	<null>		
13	1101	1	20	复旦大学	2004-06-10 15:20:00	<null>		
14	1101	1	21	复旦大学	2004-06-10 15:40:00	<null>		
15	1101	1	22	复旦大学	2004-06-10 16:00:00	<null>		
16	1101	1	23	复旦大学	2004-06-10 16:20:00	<null>		
17	1101	1	3	复旦大学	2004-06-10 08:40:00	<null>		
18	1101	1	4	复旦大学	2004-06-10 09:00:00	<null>		
19	1101	1	5	复旦大学	2004-06-10 09:20:00	<null>		
20	1101	1	6	复旦大学	2004-06-10 09:50:00	<null>		
21	1101	1	7	复旦大学	2004-06-10 10:10:00	<null>		
22	1101	1	8	复旦大学	2004-06-10 10:30:00	<null>		
23	1101	1	9	复旦大学	2004-06-10 10:50:00	<null>		
24	1101	2	1	复旦大学	2004-06-10 08:00:00	<null>		
25	1101	2	10	复旦大学	2004-06-10 11:10:00	<null>		
26	1101	2	11	复旦大学	2004-06-10 12:00:00	<null>		
27	1101	2	12	复旦大学	2004-06-10 12:20:00	<null>		
28	1101	2	13	复旦大学	2004-06-10 12:40:00	<null>		
29	1101	2	14	复旦大学	2004-06-10 13:00:00	<null>		
30	1101	2	15	复旦大学	2004-06-10 13:20:00	<null>		

room表格成功导入数据

1,772 rows							Tx: Auto		DDL	
WHERE							ORDER BY			
	registno	name	kdno	kcno	ccno	seat				
1	0016125	赵 懿	1101	2	19	0				
2	0018108	杜 林	1101	1	7	0				
3	0119067	杨凡	1101	2	19	0				
4	0150036	李昂	1101	1	2	0				
5	0212076	王一伟	1101	2	23	0				
6	0212077	赵童建	1101	1	14	0				
7	0212078	金 兰	1101	2	13	0				
8	0212079	姚微娜	1101	2	13	0				
9	0212080	曹满泽	1101	2	13	0				
10	0212085	毛蓓俊	1101	2	13	0				
11	0212086	王慧隽	1101	1	20	0				
12	0212089	朱依丽	1101	2	13	0				
13	0212090	刘 肇	1101	1	20	0				
14	0212091	朱加桢	1101	1	21	0				
15	0212110	韩 婷	1101	1	20	0				
16	0212111	陆丹红	1101	1	20	0				
17	0212113	施 能	1101	1	20	0				
18	0212114	孙 思	1101	1	20	0				
19	0212116	吴佳	1101	1	20	0				
20	0212138	赵巍巍	1101	1	20	0				
21	0212143	严霁帆	1101	1	20	0				
22	0212145	张婉	1101	2	13	0				
23	0213152	王颖彦	1101	1	14	0				
24	0216001	罗仕通	1101	2	3	0				
25	0218030	郑亦惟	1101	1	9	0				
26	0218054	郑 杨	1101	1	9	0				
27	0218055	周映	1101	1	9	0				
28	0229020	薛一品	1101	1	14	0				
29	0234018	努尔曼·司马义	1101	1	4	0				

student表格成功导入数据

## 思考&实验中问题

- 同样的创表定义，使用Mybatis框架导入时出现了违反外键约束的情况，而JDBC实现时并没有出现这样的情况，使用count(\*)计数后也发现少了3条记录，不知道是什么原因
- 如果外部数据（原始数据表）数据不完整（例如某个不应该为空的字段缺失数据）或不一致（例如本应有外键关系的数据并没有保持引用完整性），有哪些方法可以处理？
  - 数据清洗：对于缺失的数据，可以使用默认值填充，或基于其他数据计算出一个合理的值
  - 数据验证：在处理数据之前，对数据进行验证，确保它们满足特定的规则或约束。对不符合规则的数据项，根据情况修正或排除。
  - 使用临时表或中间层：将原始数据首先导入到临时表中，进行清洗和转换操作，不符合要求的数据可以被修正或删除，最后再将数据移动到最终的表中
- 处理原始数据的原则
  1. 主键约束（PRIMARY KEY）
    - 确保表中每行数据的唯一性。
    - 一个表只能有一个主键，主键可以包括一个或多个列（复合主键）。
  2. 外键约束（FOREIGN KEY）
    - 用于建立两个表之间的关系，确保引用的完整性。
    - 外键指向另一个表的主键，确保了数据之间的一致性。
  3. 唯一约束（UNIQUE）
    - 保证一列或列组合中的所有值都是唯一的，但允许有 NULL 值（取决于数据库的实现）。
  4. 检查约束（CHECK）



- 用于确保列中的值符合特定条件。
- 5. 非空约束 (NOT NULL)
  - 确保列中的值不能为 `NULL`。
- 6. 默认值约束 (DEFAULT)
  - 为列定义默认值。如果插入行时未提供值，则使用默认值。
  - 有助于保证数据的完整性，特别是对于非空列。

### 约束原则

- **数据完整性**：通过使用约束，可以确保数据的准确性和可靠性，防止错误数据的输入。
- **数据一致性**：约束帮助维护数据之间的逻辑关系，确保数据库内部数据的一致性。
- **数据安全性**：约束可以预防意外或恶意的数据修改，提高数据的安全性。
- **性能优化**：某些类型的约束（如主键和唯一约束）可以帮助数据库优化查询和数据操作的性能。