**数据库系统project报告**

2022-2023学年第2学期（CST21118）

|  |  |
| --- | --- |
| 数据库系统project任务书 | |
| 名称 | 数据库SQL执行器设计与模拟实现 |
| 类型 | □验证性 □设计性 综合性 |
| 内容 | 针对关系型数据库SQL语句执行，进行需求分析，并设计一个简单的数据库系统SQL执行器模块，根据设计模拟实现SQL基本操作等，主要实现：数据库创建、表格创建、数据添加、删除、更新等操作。  模拟实现采用：python或者Java实现具体功能，设计中若有数据的存储可以使用文本文件或者excel文件，SQL语句执行采用函数实现。 |
| 要求 | （1）设计方案要合理；  （2）能基于该SQL执行器模块完成SQL的模拟实现；  （3）设计方案有一定的效率分析。 |
| 任务时间 | 2023年4月17日至2023年5月20日 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 小组成员 | | | | | | |
| 20214578 | | 20215353 | | 20215149 | | 20214983 |
| 张梓健 | | 严玉光 | | 任俊璇 | | 刘宇桥 |
| 项目评分表 | | | | | | |
| 序号 | 评分项 | | 分值 | | 得分 | |
| 1 | 需求分析 | | 3分 | |  | |
| 2 | 综合设计与实现 | | 4分 | |  | |
| 3 | 团队协作 | | 3分 | |  | |
| 项目总得分： | | | | | | |

课程项目评分标准（总分10分）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 评分项目 | 完成情况 | 得分 |
| 1 | 需求分析 | 分析合理 | 3分 |
| 分析较合理 | 2分 |
| 分析不合理 | 1分 |
| 分析完全错误 | 0分 |
| 2 | 综合设计与实现 | 设计完整，设计合理，工具使用熟练 | 4分 |
| 设计较完整，设计合理，工具使用较熟练 | 3分 |
| 设计较完整，设计较合理，工具使用较熟练 | 2分 |
| 设计较完整，设计不合理，工具使用不熟练 | 1分 |
| 抄袭、被抄袭 | 0分 |
| 3 | 团队协作 | 有团队，分工合理，密切协作 | 3分 |
| 有团队，分工合理，有一定协作 | 2分 |
| 有团队，分工不合理，无协作 | 1分 |
| 无团队，无协作 | 0分 |

**I、小组分工：**

刘宇桥：进行需求分析，上网搜集资料，提出初步实现方案。

张梓健&任俊璇：进一步明确实现方案的细节，然后一起进行编程实现，探讨如何解决遇到的问题，以及系统功能的测试。

严玉光：分析和总结系统的效率和功能，编写实验报告。

**II、项目设计：**

一、需求分析

1.SQL语句解析：实现一个SQL解析器，能够接收标准SQL语句并将其解析为内部数据结构，例如解析出表名、列名、操作类型、条件等。

2.SQL语句执行：根据解析器解析后的结果，实现一个SQL执行引擎，能够执行基本的数据库操作，例如创建数据库、创建表格、插入数据、删除数据和更新数据。

3.数据存储：设计一种机制来存储和检索数据。在这个简单的数据库系统中，我们选择将数据存储在XML文件中。每个表格对应一个XML文件，每一行表示一条记录，每个列对应XML文件中的元素或属性。

4.数据库操作支持：实现支持常见的数据库操作，如创建数据库、删除数据库、创建表格、删除表格、插入数据、删除数据和更新数据等。

5.错误处理：实现错误处理机制，能够捕获和处理各种错误，例如语法错误、表不存在、列不存在等，并提供相应的错误提示信息。

二、实现方案

基于上述需求分析，可以提出以下实现方案

1.SQL语句解析器：设计一个SQL解析器，使用正则表达式或其他解析技术（apache calcite），将输入的SQL语句解析成内部的数据结构。解析器应能够提取出SQL语句的关键信息，例如操作类型（CREATE、INSERT、DELETE、UPDATE）、表名、列名、条件等。

2.SQL执行引擎：根据解析器提供的信息，设计一个SQL执行引擎，根据操作类型执行相应的数据库操作。例如，对于CREATE操作，引擎将创建一个新的XML文件；对于INSERT操作，引擎将在XML文件中插入一条新的记录；对于DELETE操作，引擎将从XML文件中删除符合条件的记录；对于UPDATE操作，引擎将更新符合条件的记录。

3.数据存储：使用XML作为数据存储的形式。每个表格对应一个XML文件，表格的每一行对应XML文件中的一个元素，每个列对应元素的属性或子元素。可以使用DOM解析器（如dom4j）来读取和写入XML文件。

4.数据库操作接口：设计一个简单的数据库操作接口，包括创建数据库、删除数据库、创建表格、删除表格、插入数据、删除数据和更新数据等操作。接口应该提供足够的灵活性，使得用户能够根据具体需求进行操作。

5.错误处理：实现错误处理机制，例如通过异常处理来捕获和处理语法错误、表不存在、列不存在等错误。对于错误的情况，向用户提供相应的错误提示信息，帮助其理解问题所在。

综上所述，以上方案提供了一个将数据存储在XML文件中的简单带有SQL解析功能以及sql执行器模块的数据库系统的实现方案。该方案可以满足基本的数据库操作需求，并通过XML作为存储介质，提供了一种简单而灵活的数据存储方式。

三、编程实现

附件的压缩包为IntelliJ IDEA项目的文件夹，解压到idea项目文件夹后可以在idea中打开运行，完整的项目文件我也上传到了GitHub上<https://github.com/atri45/CQU_Database_Project2>

我们使用java语言在IntelliJ IDEA 2022.2.3中编程实现了一个简单的数据库系统，该系统使用calcite解析sql语句，调用函数执行sql语句，表数据以xml文件形式存储，下面列出实现的各个类：

Database.java：数据类，存储数据库的一些基本信息，因为本次项目更注重于表的操作，所以与数据库相关的功能没有进一步的实现（例如展示数据库的元数据，创建，登录，删除数据库等）

SQLStatement.java：sql信息类，存储sql语句解析后的各种信息，例如操作符，表名，字段名等

SQLParser.java：sql解析类，用于解析SQL语句并将其转换为SQLStatement对象

SQLExecuter.java：sql执行类，用于执行SQL语句，对数据库进行操作

Main.java：主程序，提供用户与数据库的交互功能

DatabaseTest.java：测试类，使用junit对数据库系统执行各种sql语句的功能进行了测试验证

java源代码如下

1. Database.java

package org;

public class Database {

public static String databaseName;

public static String databasePath;

//public static String databaseInfoPath;

public Database(String databaseName, String databasePath) {

Database.databaseName = databaseName;

Database.databasePath = databasePath;

}

/\*

public List<String> getTableList() throws DocumentException {

List<String> tableList = new ArrayList<>();

Document document = loadDocument(databaseInfoPath);

Element rootElement = document.getRootElement();

List<Element> tables = rootElement.elements("table");

for (Element table : tables) {

tableList.add(table.attributeValue("name"));

}

return tableList;

}

private void updateDatabaseInfo(String tableName, boolean add) throws DocumentException, IOException {

Document document;

File file = new File(databaseInfoPath);

if (file.exists()) {

document = loadDocument(databaseInfoPath);

} else {

document = DocumentHelper.createDocument();

Element rootElement = document.addElement("database");

saveDocument(document, databaseInfoPath);

}

Element rootElement = document.getRootElement();

List<Element> tables = rootElement.elements("table");

boolean tableExists = false;

for (Element table : tables) {

if (table.attributeValue("name").equals(tableName)) {

tableExists = true;

if (!add) {

table.detach();

}

break;

}

}

if (!tableExists && add) {

Element tableElement = rootElement.addElement("table");

tableElement.addAttribute("name", tableName);

tableElement.addElement("columns");

tableElement.addElement("rows").setText("0");

}

saveDocument(document, databaseInfoPath);

}

private void updateTableInfo(String tableName, int count) throws DocumentException, IOException {

Document document = loadDocument(databaseInfoPath);

Element rootElement = document.getRootElement();

List<Element> tables = rootElement.elements("table");

for (Element table : tables) {

if (table.attributeValue("name").equals(tableName)) {

Element rowsElement = table.element("rows");

int rowCount = Integer.parseInt(rowsElement.getText()) + count;

rowsElement.setText(String.valueOf(rowCount));

break;

}

}

saveDocument(document, databaseInfoPath);

}\*/

}

1. SQLExecuter.java

package org;

import org.dom4j.\*;

import org.dom4j.io.OutputFormat;

import org.dom4j.io.SAXReader;

import org.dom4j.io.XMLWriter;

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class SQLExecuter {

public static Database db;

public SQLExecuter(Database database) {

this.db = database;

}

public static void execute(SQLStatement statement) throws Exception {

String tableName = statement.getTableName();

switch (statement.getOperation()) {

case CREATE:

createTable(tableName);

break;

case DROP:

dropTable(tableName);

break;

case INSERT:

List<String> columns = statement.getColumns();

List<String> values = statement.getValues();

insertData(tableName, columns, values);

break;

case DELETE:

String condition = statement.getCondition();

deleteData(tableName, condition);

break;

case UPDATE:

columns = statement.getColumns();

values = statement.getValues();

condition = statement.getCondition();

updateData(tableName, columns, values, condition);

break;

case SELECT:

condition = statement.getCondition();

selectData(tableName, condition);

break;

default:

throw new IllegalArgumentException("Invalid operation");

}

}

public static void createTable(String tableName) throws Exception {

// 创建文件夹

File directory = new File(db.databasePath);

if (!directory.exists()) {

directory.mkdir();

}

// 检查同名表是否存在

File tableXml = new File(getTablePath(tableName));

if (tableXml.exists()) {

throw new Exception("表已存在！");

}

// 创建document

Document document = DocumentHelper.createDocument();

document.addElement(tableName);

// 将document转化为xml文件

try (FileWriter out = new FileWriter(getTablePath(tableName))) {

OutputFormat format = OutputFormat.createPrettyPrint();

format.setEncoding("UTF-8");

format.setIndentSize(4);

XMLWriter writer = new XMLWriter(out, format);

writer.write(document);

} catch (IOException e) {

System.out.println(e.getMessage());

}

}

public static void dropTable(String tableName) throws Exception {

String tablePath = getTablePath(tableName);

File file = new File(tablePath);

if (file.exists()) {

file.delete();

}

}

public static void insertData(String tableName, List<String> columns, List<String> values) throws IOException, DocumentException {

Document document = loadDocument(getTablePath(tableName));

Element rootElement = document.getRootElement();

Element rowElement = rootElement.addElement("row");

rowElement.addAttribute("rowID", generateRowId(rootElement));

for (int i = 0; i < columns.size(); i++) {

rowElement.addElement(columns.get(i)).setText(values.get(i));

}

saveDocument(document, getTablePath(tableName));

}

public static void deleteData(String tableName, String condition) throws IOException, DocumentException {

Document document = loadDocument(getTablePath(tableName));

Element rootElement = document.getRootElement();

String xpathExpression = "//row" + (condition != null ? "[" + condition + "]" : "");

List<Node> nodes = rootElement.selectNodes(xpathExpression);

int count = nodes.size();

for (Node node : nodes) {

node.detach();

}

saveDocument(document, getTablePath(tableName));

}

public static void updateData(String tableName, List<String> columns, List<String> values, String condition) throws IOException, DocumentException {

Document document = loadDocument(getTablePath(tableName));

Element rootElement = document.getRootElement();

String xpathExpression = "//row" + (condition != null ? "[" + condition + "]" : "");

List<Node> nodes = rootElement.selectNodes(xpathExpression);

int count = 0;

for (Node node : nodes) {

Element element = (Element) node;

for (int i = 0; i < columns.size(); i++) {

Element columnElement = element.element(columns.get(i));

if (columnElement != null) {

columnElement.setText(values.get(i));

count++;

}

}

}

saveDocument(document, getTablePath(tableName));

}

public static void selectData(String tableName, String condition) throws DocumentException {

Document document = loadDocument(getTablePath(tableName));

String xpathExpression = "//row" + (condition != null ? "[" + condition + "]" : "");

List<Node> nodes = document.selectNodes(xpathExpression);

for (Node node : nodes) {

Element recordElement = (Element) node;

Map<String, String> rowData = new HashMap<>();

for (Element fieldElement : recordElement.elements()) {

if (fieldElement != null) {

rowData.put(fieldElement.getName(), fieldElement.getText());

}

}

System.out.println(rowData);

}

}

private static String generateRowId(Element rootElement) {

int rowCount = rootElement.elements("row").size();

return String.valueOf(rowCount);

}

public static String getTablePath(String tableName) {

return db.databasePath + "/" + tableName + ".xml";

}

private static Document loadDocument(String filePath) throws DocumentException {

SAXReader reader = new SAXReader();

return reader.read(new File(filePath));

}

private static void saveDocument(Document document, String filePath) throws IOException {

OutputFormat format = OutputFormat.createPrettyPrint();

XMLWriter writer = new XMLWriter(new FileWriter(filePath), format);

writer.write(document);

writer.close();

}

}

1. SQLParser.java

package org;

import org.apache.calcite.sql.\*;

import org.apache.calcite.sql.ddl.SqlCreateTable;

import org.apache.calcite.sql.ddl.SqlDropTable;

import org.apache.calcite.sql.parser.SqlParseException;

import org.apache.calcite.sql.parser.SqlParser;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class SQLParser {

public static SQLStatement parse(String sql) {

SqlParser sqlParser = SqlParser.create(sql);

try {

SqlNode sqlNode = sqlParser.parseStmt();

return buildSQLStatement(sqlNode);

} catch (SqlParseException e) {

throw new IllegalArgumentException("Invalid SQL statement: " + sql, e);

}

}

private static SQLStatement buildSQLStatement(SqlNode sqlNode) {

SqlKind sqlKind = sqlNode.getKind();

switch (sqlKind) {

case CREATE\_TABLE:

return buildCreateTableStatement((SqlCreateTable) sqlNode);

case DROP\_TABLE:

return buildDropTableStatement((SqlDropTable) sqlNode);

case INSERT:

return buildInsertStatement((SqlInsert) sqlNode);

case DELETE:

return buildDeleteStatement((SqlDelete) sqlNode);

case UPDATE:

return buildUpdateStatement((SqlUpdate) sqlNode);

case SELECT:

return buildSelectStatement((SqlSelect) sqlNode);

default:

throw new IllegalArgumentException("Unsupported SQL statement: " + sqlNode);

}

}

private static SQLStatement buildCreateTableStatement(SqlCreateTable createTable) {

String tableName = createTable.name.toString();

// Extract column names and types from the CREATE TABLE statement

// ...

// Create and return a CREATE operation SQLStatement object

return new SQLStatement(SQLStatement.Operation.CREATE, tableName, null, null, null);

}

private static SQLStatement buildDropTableStatement(SqlDropTable dropTable) {

String tableName = dropTable.name.toString();

// Create and return a DROP operation SQLStatement object

return new SQLStatement(SQLStatement.Operation.DROP, tableName, null, null, null);

}

private static SQLStatement buildInsertStatement(SqlInsert insert) {

String tableName = insert.getTargetTable().toString();

// Extract column names and values from the INSERT statement

List<String> columns = new ArrayList<>();

List<String> values;

SqlNodeList columnList = insert.getTargetColumnList();

SqlNode valueList = insert.getSource();

for (int i = 0; i < columnList.size(); i++) {

SqlIdentifier column = (SqlIdentifier) columnList.get(i);

String columnName = column.getSimple();

columns.add(columnName);

}

String valueString = valueList.toString().replace("'", "");

values = Arrays.asList(valueString.substring("[VALUES ROW(".length() - 1, valueString.length() - 1).split(","));

// Create and return an INSERT operation SQLStatement object

return new SQLStatement(SQLStatement.Operation.INSERT, tableName, columns, values, null);

}

private static SQLStatement buildDeleteStatement(SqlDelete delete) {

String tableName = delete.getTargetTable().toString();

String condition = parseCondition(delete.getCondition());

// Create and return a DELETE operation SQLStatement object

return new SQLStatement(SQLStatement.Operation.DELETE, tableName, null, null, condition);

}

private static SQLStatement buildUpdateStatement(SqlUpdate update) {

String tableName = update.getTargetTable().toString();

// Extract column names, values, and condition from the UPDATE statement

List<String> columns = new ArrayList<>();

List<String> values = new ArrayList<>();

String condition = null;

SqlNodeList updateList = update.getTargetColumnList();

SqlNodeList valueList = update.getSourceExpressionList();

SqlNode conditionNode = update.getCondition();

for (int i = 0; i < updateList.size(); i++) {

SqlIdentifier column = (SqlIdentifier) updateList.get(i);

String columnName = column.getSimple();

columns.add(columnName);

SqlNode valueNode = valueList.get(i);

String value = valueNode.toString();

values.add(value);

}

if (conditionNode != null) {

condition = parseCondition(conditionNode);

}

// Create and return an UPDATE operation SQLStatement object

return new SQLStatement(SQLStatement.Operation.UPDATE, tableName, columns, values, condition);

}

private static SQLStatement buildSelectStatement(SqlSelect select) {

String tableName = select.getFrom().toString();

String condition = null;

SqlNode whereNode = select.getWhere();

if (whereNode != null) {

condition = parseCondition(whereNode);

}

return new SQLStatement(SQLStatement.Operation.SELECT, tableName, null, null, condition);

}

public static String parseCondition(SqlNode conditionNode) {

String condition = conditionNode.toString();

if (condition.matches(".\*\\d$")) {

condition = condition.replace("`", "");

}

else {

condition = condition.replace("`", "'");

condition = condition.replaceFirst("'", "").replaceFirst("'", "");

}

return condition;

}

}

1. SQLStatement.java

package org;

import java.util.List;

public class SQLStatement {

private final Operation operation;

private final String tableName;

private final List<String> columns;

private final List<String> values;

private final String condition;

public SQLStatement(Operation operation, String tableName, List<String> columns, List<String> values, String condition) {

this.operation = operation;

this.tableName = tableName;

this.columns = columns;

this.values = values;

this.condition = condition;

}

public Operation getOperation() {

return operation;

}

public String getTableName() {

return tableName;

}

public List<String> getColumns() {

return columns;

}

public List<String> getValues() {

return values;

}

public String getCondition() {

return condition;

}

public enum Operation {

CREATE, DROP, INSERT, DELETE, UPDATE, SELECT

}

public void show() {

System.out.println("operation: " + operation);

System.out.println("tableName: " + tableName);

System.out.println("columns: " + columns);

System.out.println("values: " + values);

System.out.println("condition: " + condition);

}

}

1. Main.java

package org;

import java.io.BufferedReader;

import java.io.File;

import java.io.InputStreamReader;

public class Main {

public Main() {

}

public static void main(String[] args) throws Exception {

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

System.out.println("请输入您的数据库名：");

String databaseName = reader.readLine();

File directory = new File("");

Database db = new Database(databaseName, directory.getCanonicalPath() + "/target/classes/org/database");

SQLExecuter sqlExecuter = new SQLExecuter(db);

System.out.println("请输入您要执行的sql语句：");

String sql;

while((sql = reader.readLine()) != null) {

SQLStatement createStatement1 = SQLParser.parse(sql);

SQLExecuter.execute(createStatement1);

System.out.println("请输入您要执行的sql语句：");

}

}

}

1. Databasetest.java

import org.Database;

import org.SQLExecuter;

import org.SQLParser;

import org.SQLStatement;

import org.junit.Test;

import java.io.File;

import java.io.IOException;

import static org.junit.Assert.assertEquals;

public class DatabaseTest {

public DatabaseTest() throws IOException {}

public String getDatabasePath() throws IOException {

File directory = new File("");

return directory.getCanonicalPath()+"/target/classes/org/database";

}

Database db = new Database("CQU-DB", getDatabasePath());

SQLExecuter sqlExecuter = new SQLExecuter(db);

@Test

public void testGetDataBasePath() throws IOException {

String databasePath = getDatabasePath();

assertEquals(databasePath, "/home/zijian/IdeaProjects/database\_project2/target/classes/org/database");

}

@Test

public void testCreateTable() throws Exception {

if (new File(sqlExecuter.getTablePath("TEST")).exists()) {

sqlExecuter.dropTable("TEST2");

}

sqlExecuter.createTable("TEST2");

}

@Test

public void testDropTable() throws Exception {

sqlExecuter.dropTable("TEST2");

}

@Test

public void testInsertData() throws Exception {

if (!new File(sqlExecuter.getTablePath("TEST")).exists()) {

sqlExecuter.createTable("TEST");

}

String insertSQL1 = "INSERT INTO test (id, name, age) VALUES (1, 'Mike', 19)";

SQLStatement insertStatement1 = SQLParser.parse(insertSQL1);

sqlExecuter.execute(insertStatement1);

String insertSQL2 = "INSERT INTO test (id, name, age) VALUES (2, 'Adam', 33)";

SQLStatement insertStatement2 = SQLParser.parse(insertSQL2);

sqlExecuter.execute(insertStatement2);

String insertSQL3 = "INSERT INTO test (id, name, age) VALUES (3, 'Hertz', 20)";

SQLStatement insertStatement3 = SQLParser.parse(insertSQL3);

sqlExecuter.execute(insertStatement3);

}

@Test

public void testUpdateData() throws Exception {

if (!new File(sqlExecuter.getTablePath("TEST")).exists()) {

sqlExecuter.createTable("TEST");

}

String insertSQL = "INSERT INTO test (id, name, age) VALUES (1, 'Mike', 19)";

SQLStatement insertStatement = SQLParser.parse(insertSQL);

sqlExecuter.execute(insertStatement);

String updateSQL = "UPDATE test SET name = 'Hertz' WHERE name = \"Mike\"";

SQLStatement updateStatement = SQLParser.parse(updateSQL);

sqlExecuter.execute(updateStatement);

}

@Test

public void testSelectData() throws Exception {

if (!new File(sqlExecuter.getTablePath("TEST")).exists()) {

sqlExecuter.createTable("TEST");

}

String selectSQL = "SELECT \* FROM test";

SQLStatement selectStatement = SQLParser.parse(selectSQL);

sqlExecuter.execute(selectStatement);

}

@Test

public void testDeleteData() throws Exception {

if (!new File(sqlExecuter.getTablePath("TEST")).exists()) {

sqlExecuter.createTable("TEST");

}

String deleteSQL1 = "DELETE FROM test WHERE id = 1";

SQLStatement deleteStatement1 = SQLParser.parse(deleteSQL1);

sqlExecuter.execute(deleteStatement1);

String deleteSQL2 = "DELETE FROM test WHERE name = \"Adam\"";

SQLStatement deleteStatement2 = SQLParser.parse(deleteSQL2);

sqlExecuter.execute(deleteStatement2);

String deleteSQL3 = "DELETE FROM test WHERE age = 20";

SQLStatement deleteStatement3 = SQLParser.parse(deleteSQL3);

sqlExecuter.execute(deleteStatement3);

}

}

四、测试

首先使用junit对数据库系统执行各种sql语句的功能进行测试：

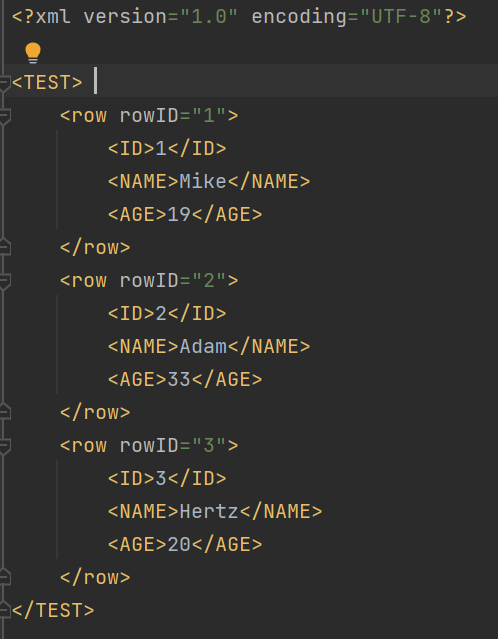
1. 如下方代码所示，运行代码后在指定路径创建了表格test和test2.

@Test  
public void testCreateTable() throws Exception {  
 String createSQL1 = "CREATE TABLE TEST";  
 SQLStatement createStatement1 = SQLParser.*parse*(createSQL1);  
 SQLExecuter var10000 = this.sqlExecuter;  
 SQLExecuter.*execute*(createStatement1);  
 String createSQL2 = "CREATE TABLE TEST2";  
 SQLStatement createStatement2 = SQLParser.*parse*(createSQL2);  
 var10000 = this.sqlExecuter;  
 SQLExecuter.*execute*(createStatement2);  
}



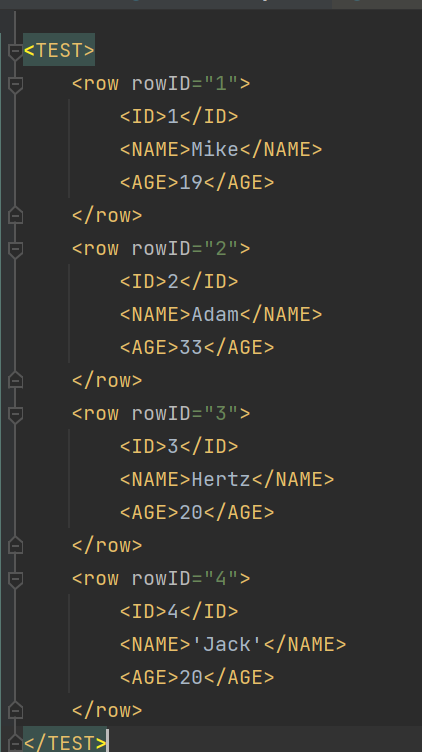
2. 如下方代码和图片所示，运行代码后在test表中新建了三行数据

@Test  
public void testInsertData() throws Exception {  
 SQLExecuter var10002 = this.sqlExecuter;  
 SQLExecuter var10000;  
 if (!(new File(SQLExecuter.*getTablePath*("TEST"))).exists()) {  
 var10000 = this.sqlExecuter;  
 SQLExecuter.*createTable*("TEST");  
 }  
  
 String insertSQL1 = "INSERT INTO test (id, name, age) VALUES (1, 'Mike', 19)";  
 SQLStatement insertStatement1 = SQLParser.*parse*(insertSQL1);  
 var10000 = this.sqlExecuter;  
 SQLExecuter.*execute*(insertStatement1);  
 String insertSQL2 = "INSERT INTO test (id, name, age) VALUES (2, 'Adam', 33)";  
 SQLStatement insertStatement2 = SQLParser.*parse*(insertSQL2);  
 var10000 = this.sqlExecuter;  
 SQLExecuter.*execute*(insertStatement2);  
 String insertSQL3 = "INSERT INTO test (id, name, age) VALUES (3, 'Hertz', 20)";  
 SQLStatement insertStatement3 = SQLParser.*parse*(insertSQL3);  
 var10000 = this.sqlExecuter;  
 SQLExecuter.*execute*(insertStatement3);  
}



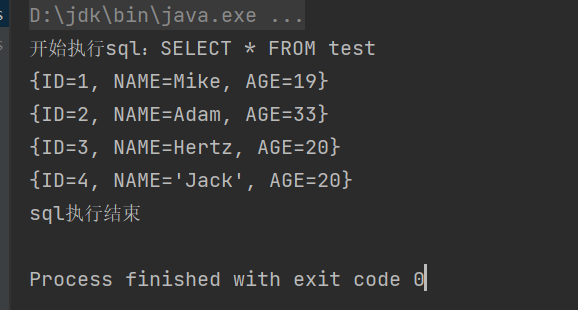
3. 如下方代码和图片所示，运行代码后在test表中插入一条新数据后又进行了修改

@Test  
public void testUpdateData() throws Exception {  
 if (!new File(sqlExecuter.*getTablePath*("TEST")).exists()) {  
 sqlExecuter.*createTable*("TEST");  
 }  
 String insertSQL = "INSERT INTO test (id, name, age) VALUES (4, 'Tom', 20)";  
 SQLStatement insertStatement = SQLParser.*parse*(insertSQL);  
 sqlExecuter.*execute*(insertStatement);  
 String updateSQL = "UPDATE test SET name = 'Jack' WHERE name = \"Tom\"";  
 SQLStatement updateStatement = SQLParser.*parse*(updateSQL);  
 sqlExecuter.*execute*(updateStatement);  
}



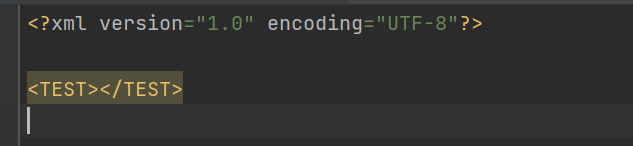
4.运行下方代码后，控制台输出了表格信息

@Test  
public void testSelectData() throws Exception {  
 if (!new File(sqlExecuter.*getTablePath*("TEST")).exists()) {  
 sqlExecuter.*createTable*("TEST");  
 }  
 String selectSQL = "SELECT \* FROM test";  
 SQLStatement selectStatement = SQLParser.*parse*(selectSQL);  
 sqlExecuter.*execute*(selectStatement);  
}



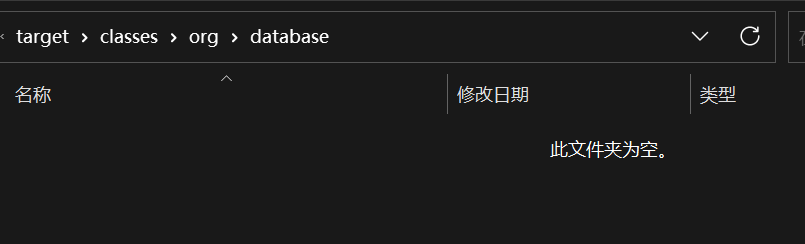
5.运行以下代码后，表格中对应数据被成功删除

@Test  
 public void testDeleteData() throws Exception {  
 if (!new File(sqlExecuter.*getTablePath*("TEST")).exists()) {  
 sqlExecuter.*createTable*("TEST");  
 }  
 String deleteSQL1 = "DELETE FROM test WHERE id = 1";  
 SQLStatement deleteStatement1 = SQLParser.*parse*(deleteSQL1);  
 sqlExecuter.*execute*(deleteStatement1);  
 String deleteSQL2 = "DELETE FROM test WHERE name = \"Adam\"";  
 SQLStatement deleteStatement2 = SQLParser.*parse*(deleteSQL2);  
 sqlExecuter.*execute*(deleteStatement2);  
 String deleteSQL3 = "DELETE FROM test WHERE age = 20";  
 SQLStatement deleteStatement3 = SQLParser.*parse*(deleteSQL3);  
 sqlExecuter.*execute*(deleteStatement3);  
 }  
}



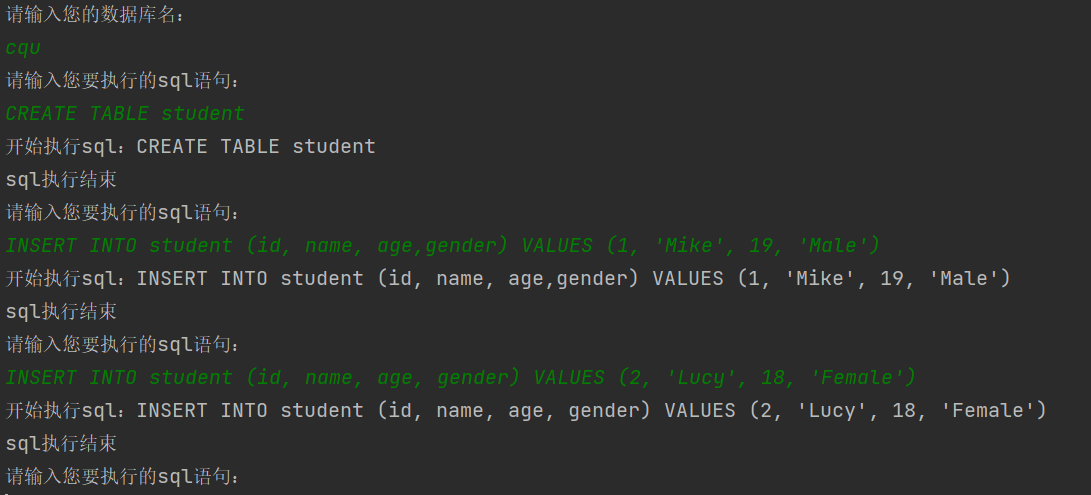
6.运行以下代码对应的两个表格被删除

@Test  
public void testDropTable() throws Exception {  
 String dropSQL1 = "DROP TABLE TEST";  
 SQLStatement dropStatement1 = SQLParser.*parse*(dropSQL1);  
 SQLExecuter var10000 = this.sqlExecuter;  
 SQLExecuter.*execute*(dropStatement1);  
 String dropSQL2 = "DROP TABLE TEST2";  
 SQLStatement dropStatement2 = SQLParser.*parse*(dropSQL2);  
 var10000 = this.sqlExecuter;  
 SQLExecuter.*execute*(dropStatement2);  
}

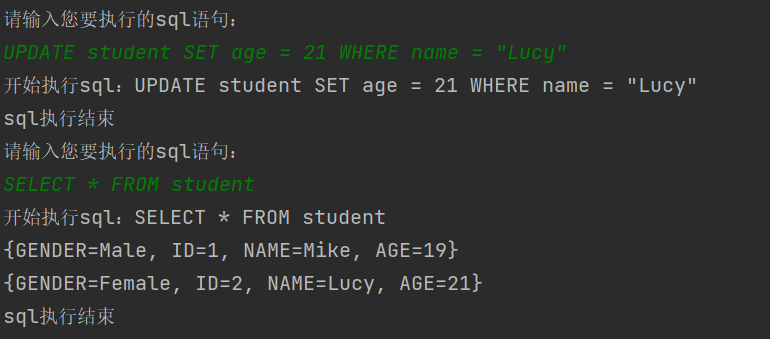


接着，测试用户直接通过控制台与数据库的交互功能

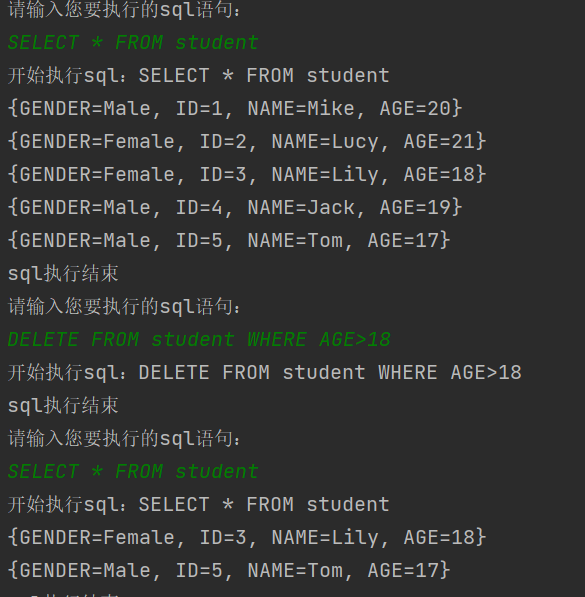
1.在控制台输入sql语句后，成功在对应路径创建了student表格，并增加了相应实例。



2. 在控制台输入相应语句后，成功对数据进行了更新，并进行查找。



3.通过前后输出对比，可以看到相应的实例被删除了



五、效率分析

对于上述实现方案的效率分析，我们主要从以下几个方面进行：

1. SQL语句解析器：对SQL语句进行解析的效率取决于选择的解析技术。对于一些复杂的SQL语句，可能需要花费较多的时间进行解析。如果频繁地执行这些复杂的SQL语句，那么解析器的性能可能会成为系统性能的瓶颈。使用成熟的开源库（如Apache Calcite）可以帮助提高解析效率。

2. SQL执行引擎：执行SQL语句的效率主要取决于具体的数据库操作和数据存储方式。对于实现小型数据库，xml文件可以作为一个选择，xml提供良好了的数据存储性能。

3. 数据存储：使用XML文件作为数据存储方式具有一定的灵活性，但也有一些限制，XML文件的解析和生成通常需要较大的处理时间，所以我们使用dom4j来进行xml文件的解析和生成，使用类似于正则表达式的xpath可以大大提高数据查询的效率。

4. 数据库操作接口：接口的执行效率会直接影响到整个数据库系统的性能，所有我们在设计接口时尽可能简洁易用，各个模块各个函数功能明确，解耦性强，以减少用户操作的复杂性和错误发生的可能性。

总体来说，我们的实现方案主要适用于小型系统或者教学环境，对于大规模数据和高并发场景不太适用。如果要实现的系统需要处理大量的数据或者需要支持高并发操作，需要考虑使用更为强大和高效的数据存储方式和sql解析执行方式。

**III、项目总结：**

根据上述方案，我们实现了一个基于XML存储的简单数据库SQL执行器。该系统具有SQL解析器和执行引擎，能够执行基本的数据库操作，包括创建和删除数据库、创建和删除表格、插入、删除和更新数据等。我们使用XML文件作为数据存储的形式，每个表格对应一个XML文件，表格的每一行对应XML文件中的一个元素，每个列对应元素的属性或子元素。

在系统设计中，我们采用了正则表达式（regex）和apache calcite等技术来实现SQL语句解析器，将用户输入的SQL语句解析成内部的数据结构。解析器提取了SQL语句的关键信息，例如操作类型、表名、列名和条件等，并将其传递给执行引擎进行处理。

在执行引擎中，我们根据操作类型执行相应的数据库操作，如创建和删除数据库、创建和删除表格、插入、删除和更新数据等。对于每个操作，我们都使用DOM解析器（如dom4j）来读取和写入XML文件。在实现过程中，我们还考虑了性能优化措施，如缓存机制和索引优化，以提高系统的执行效率和响应速度。

在数据库操作接口中，我们设计了一个简单的接口，包括创建数据库、删除数据库、创建表格、删除表格、插入数据、删除数据和更新数据等操作。接口提供了足够的灵活性，使得用户能够根据具体需求进行操作。

在错误处理方面，我们实现了错误处理机制，通过异常处理来捕获和处理语法错误、表不存在、列不存在等错误，并向用户提供相应的错误提示信息，以帮助其理解问题所在。

本数据库系统实现了基本的数据库操作需求，并提供了一种简单而灵活的数据存储方式。但是，在实际场景中，该系统还需要进一步完善，如增加更多的数据库操作功能、提高系统的安全性等。