大型数据库应用开发心得

一.概述

在此次大型数据库应用开发中，我们需要通过Embedded SQL,cli,jdbc,sqlj这四种与数据库交互的方式，完成简单的应用程序开发。

我们使用db2数据库，并使用其自带创建的sample数据库作为交互对象。

开发出的应用程序可以与sample数据库进行交互，且包含有增删改查四种功能，并提供了一个主菜单来调用这些功能。

为了方便配置相关环境，我们在本机windows上安装了db2数据库，基于此进行开发。

具体环境配置如下：

- 数据库： IBM Db2 v11.5.8 Microsoft Windows(x64)
- 数据库实例：Db2
- 数据库端口：25000
- 用户：db2admin
- 开发IDE：Visual Studio 2019 , Intellij IDEA , DataGrip

接下来，我将从Embedded SQL开发、cli开发、jdbc开发、sqlj开发四个部分，介绍我的大型数据库应用程序开发。

二.Embedded SQL 开发

2.1 Static Embedded SQL开发

2.1.1 概述

Static Embedded SQL（静态嵌入式SQL）允许在应用程序代码中直接嵌入SQL语句，以实现对数据库的访问和操作。这种方法使用预编译器将应用程序代码和SQL语句组合在一起，生成可执行的代码。静态嵌入式SQL在编译时将SQL语句与应用程序代码静态绑定在一起，从而提供了一些优势和特点。

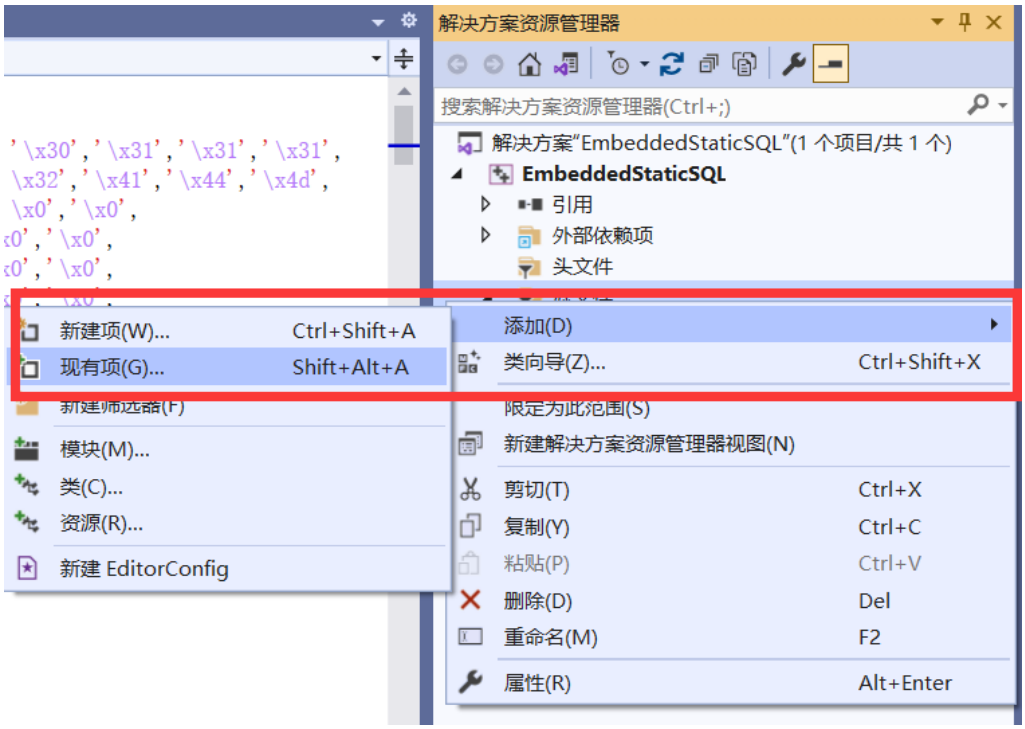
2.1.2 环境配置

环境配置的具体流程可参考《运行Embedded SQL,CLI,JDBC,SQLJ程序运行须知》，这里仅列出一些配置过程中遇到的重难点：

- 打开DB2命令窗口，进入到源代码文件StaticSQL.sqc所在目录:

  完成这一步的流程为：1.进入到源代码文件所在目录，在此打开终端 2.终端中输入db2start,db2cmd即可在此目录下打开DB2命令窗口。

- 在C工程项目中添加StaticSQL.c： 在创建的C工程中点击源文件 -> 添加 -> 现有项 将目录中的c文件添加进来。



- 修改sqc代码配置语句：

  需要修改sqc代码的相关环境配置语句：

```
    // Connect to the appropriate database
    EXEC SQL CONNECT TO SAMPLE USER db2admin USING ab6121166;

    // Assign values from host variables to SQL host variables
    strcpy(EmployeeNo, hostvars.EmployeeNo);
    strcpy(LastName, hostvars.LastName);
    Salary = hostvars.Salary;

    EXEC SQL INSERT INTO THEORY.EMPLOYEE (EMPNO, LASTNAME, SALARY) VALUES (:EmployeeNo, :LastName, :Salary);
    if (sqlca.sqlcode == SQL_RC_OK) {
        printf("Employee inserted successfully.\n");
    } else {
        printf("Failed to insert employee.\n");
        handleSqlcaError();
    }

    // Disconnect from the database
    EXEC SQL DISCONNECT CURRENT;
```

> 更改与数据库的连接语句，以及更改EMPLOYEE表的模式

2.1.3 开发流程

2.1.3.1 运行流程

静态sql嵌入需要我们去预编译sqc文件，并运行生成的c文件。

故我们在开发过程中的流程如下：

1.修改sqc文件 -> 2.使用db2 prep编译 -> 3.得到c文件，运行c文件

2.1.3.2 关键代码实现

与数据库连接：

```
EXEC SQL CONNECT TO SAMPLE USER db2admin USING ab6121166;
```

声明SQL查询变量

```
EXEC SQL BEGIN DECLARE SECTION;
        char     EmployeeNo[7];
        char     LastName[16];
        double   Salary;
EXEC SQL END DECLARE SECTION;
```

通过这种方式声明的变量可以直接运用于SQL语句中。

插入语句

```
 EXEC SQL INSERT INTO THEORY.EMPLOYEE (EMPNO, LASTNAME, SALARY) VALUES (:EmployeeNo,
:LastName, :Salary);
```

查询语句

```
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT EMPNO, LASTNAME, SALARY FROM THEORY.EMPLOYEE;

// Open the cursor
EXEC SQL OPEN C1;

// Fetch and display all records available
while (sqlca.sqlcode == SQL_RC_OK) {
    EXEC SQL FETCH C1 INTO :EmployeeNo, :LastName, :Salary;

    if (sqlca.sqlcode == SQL_RC_OK) {
        printf("%-8s %-16s %lf\n", EmployeeNo, LastName, Salary);
    }
}

// Close the cursor
EXEC SQL CLOSE C1;
```

我们需要声明一个游标CURSOR用于查询，并使用游标来输出查询结果

更新语句

```
EXEC SQL UPDATE THEORY.EMPLOYEE SET LASTNAME = :LastName, SALARY = :Salary WHERE
EMPNO = :EmployeeNo;
```

删除语句

```
EXEC SQL DELETE FROM THEORY.EMPLOYEE WHERE EMPNO = :EmployeeNo;
```

其他

sqc的其他部分代码直接使用C语言的语法进行开发即可。

即sqc文件就是C语言的语法加上额外的内嵌SQL语法

2.2 Dynamic Embedded SQL

2.2.1 概述

动态嵌入式SQL（Embedded Dynamic SQL）是一种在程序代码中使用SQL语句的方法，与静态嵌入式SQL类似，其也是将SQL语句写入待编译的代码中。

但与静态嵌入式SQL所不同的是，静态嵌入式SQL中，SQL语句是以字符串的形式嵌入到程序代码中，通常在编写程序时就确定了。而动态嵌入式SQL中，SQL语句可以在运行时根据程序逻辑和用户输入来构建。

2.2.2 环境配置

与静态嵌入式类似。

我们也需要修改sqc文件中的对应配置语句：

```
// Connect To The Appropriate Database
EXEC SQL CONNECT TO SAMPLE USER administrator USING sc;

// Define A Dynamic UPDATE SQL Statement That Uses A
// Parameter Marker
strcpy(SQLStmt, "UPDATE ADMINISTRATOR EMPLOYEE SET JOB = ? ");
strcat(SQLStmt, "WHERE JOB = 'MANAGER'");
```

2.2.3 开发流程

类似的，动态SQL的开发流程也是： 修改sqc文件 -> 编译 -> 运行编译出的c文件

动态SQL可以动态的构建SQL语句：

```
strcpy(SQLStmt, "UPDATE THEORY.EMPLOYEE SET JOB = ? ");
strcat(SQLStmt, "WHERE JOB = 'MANAGER'");

// Populate The Host Variable That Will Be Used In
// Place Of The Parameter Marker
strcpy(JobType, "MANCHG");

// Prepare The SQL Statement
EXEC SQL PREPARE SQL_STMT FROM :SQLStmt;

// Execute The SQL Statement
EXEC SQL EXECUTE SQL_STMT USING :JobType;

// Commit The Transaction
EXEC SQL COMMIT;
```

至于具体的SQL增删改查的格式，乃至于与数据库连接等其他SQL语句，以及sqc文件中c语言部分的书写，动态SQL与静态SQL都是一致的，在此不再赘述。

三.cli开发

3.1 概述

CLI（Call Level Interface）是一种用于与数据库进行交互的编程接口。CLI提供了一组函数和结构体，用于连接数据库、执行SQL语句、处理结果集等操作。

CLI通过使用预编译的SQL语句和函数调用来与数据库进行通信。它允许开发人员在应用程序中嵌入SQL语句，并通过函数调用来执行这些语句，从而实现与数据库的交互。

我们可以直接在C文件中嵌入使用CLI的方式与数据库进行交互，而不需先手动编译再交互。

3.2 环境配置

参考《运行Embedded SQL,CLI,JDBC,SQLJ程序运行须知》即可

值得注意的点有：

- ODBC驱动程序填写数据源名称时，填写的是数据库实例名称，这里是DB2
- 同样需要修改代码中的环境配置，主要是与数据库的连接，以及表的模式

```
// Connect to the appropriate data source
if (conHandle != 0)
    retCode = SQLConnect(conHandle, (SQLCHAR*)"DB2", SQL_NTS, (SQLCHAR*)"db2admin", SQL_NTS, (SQLCHAR*)"ab6121166", SQL_NTS);
```

```
// Define a SELECT SQL Statement
SQLCHAR SQLStmt[255] = "SELECT EMPNO, LASTNAME FROM THEORY.EMPLOYEE";
```

3.3 开发流程

我们直接修改CLI.c代码，并运行即可。

所有代码都是在C语言环境下书写的。

关键代码如下：

初始化

```
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &envHandle);
```

使用SQLAllocHanlde来分配一个环境句柄

连接数据库

```
 // Connect to the appropriate data source
    if (conHandle != 0)
        retCode = SQLConnect(conHandle, (SQLCHAR*)"DB2", SQL_NTS,
(SQLCHAR*)"db2admin", SQL_NTS, (SQLCHAR*)"ab6121166", SQL_NTS);
```

插入语句

```
 // Allocate an SQL Statement Handle
SQLHANDLE stmtHandle;
SQLAllocHandle(SQL_HANDLE_STMT, conHandle, &stmtHandle);

// Define an INSERT SQL Statement
SQLCHAR SQLStmt[255];
sprintf((char*)SQLStmt, "INSERT INTO THEORY.EMPLOYEE (EMPNO, LASTNAME) VALUES ('%s',
'%s')", EmpNo, LastName);

// Execute the SQL Statement
SQLExecDirect(stmtHandle, SQLStmt, SQL_NTS);

printf("Employee inserted successfully.\n");

// Free the SQL Statement Handle
SQLFreeHandle(SQL_HANDLE_STMT, stmtHandle);
```

新建句柄，写SQL语句，执行查询，释放句柄

删除语句

```
// Define a DELETE SQL Statement
SQLCHAR SQLStmt[255];
sprintf((char*)SQLStmt, "DELETE FROM THEORY.EMPLOYEE WHERE EMPNO = '%s'", EmpNo);
```

流程类似，更改具体SQL语句即可

更新语句

```
// Define an UPDATE SQL Statement
SQLCHAR SQLStmt[255];
sprintf((char*)SQLStmt, "UPDATE THEORY.EMPLOYEE SET LASTNAME = '%s' WHERE EMPNO =
'%s'", LastName, EmpNo);
```

流程类似，更改具体SQL语句即可

查询语句

```
 // Allocate an SQL Statement Handle
SQLHANDLE stmtHandle;
SQLAllocHandle(SQL_HANDLE_STMT, conHandle, &stmtHandle);

// Define a SELECT SQL Statement
SQLCHAR SQLStmt[255] = "SELECT EMPNO, LASTNAME FROM THEORY.EMPLOYEE";

// Execute the SQL Statement
SQLExecDirect(stmtHandle, SQLStmt, SQL_NTS);

printf("Search results:\n");
printf("EmpNo\tLast Name\n");

// Bind the columns in the result data set returned to application variables
SQLCHAR EmpNo[10];
SQLCHAR LastName[25];
SQLBindCol(stmtHandle, 1, SQL_C_CHAR, (SQLPOINTER)EmpNo, sizeof(EmpNo), NULL);
SQLBindCol(stmtHandle, 2, SQL_C_CHAR, (SQLPOINTER)LastName, sizeof(LastName), NULL);

// Fetch and display the result data
while (SQLFetch(stmtHandle) == SQL_SUCCESS) {
    printf("%-8s %s\n", EmpNo, LastName);
}

// Free the SQL Statement Handle
SQLFreeHandle(SQL_HANDLE_STMT, stmtHandle);
```

在以上流程的基础上，使用SQLBindCol语句来收集SQL语句执行的结果，并加以输出。

四.jdbc开发

4.1 概述

JDBC（Java Database Connectivity）是Java语言中与数据库交互的一种标准接口。它提供了一组类和接口，用于连接数据库、执行SQL语句、处理结果集等操作。

JDBC通过使用Java编程语言和数据库特定的驱动程序来实现与数据库的交互。开发人员可以使用JDBC编写Java应用程序，通过驱动程序连接到数据库并执行SQL操作。

4.2 环境配置

- 配置jdbc
- - 更改环境变量

```
class StmtDb {
  1 usage
  private static final String DB_URL = "jdbc:db2://127.0.0.1:25000/sample";
  1 usage
  private static final String USERNAME = "db2admin";
  1 usage
  private static final String PASSWORD = "ab6121166";
```

  - 手动编译

```
javac ConnDb.java
javac StmtDb.java
```

- 修改代码,更改模式
-

```
String insertSql = "INSERT INTO THEORY.EMPLOYEE (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL) " +
        "VALUES (?, ?, ?, ?, ?)";
```

在连接数据库时，我们还遇到了权限不足的问题：

我们目前有两个用户：

1.操作系统母用户 这里是THEORY

其权限最高

2.我们创建db2时弄的db2admin用户

但db2admin用户没有权限去访问sample数据库，因此这里我们直接赋予db2admin用户dbadm权 限：

grant dbadm on database to user db2admin

4.3 开发流程

4.3.1 运行流程

1.javac编译ConnDb.java

2.java ConnDb 运行类文件

3.更改StmtDb.java

4.javac编译StmtDb.java

5.java StmtDb 运行

4.3.2 关键代码

连接数据库

```
Class.forName("com.ibm.db2.jcc.DB2Driver");
Connection con = DriverManager.getConnection(DB_URL, USERNAME, PASSWORD);
```

设置自动commit

```
con.setAutoCommit(true);
```

插入语句

```
 String insertSql = "INSERT INTO THEORY.EMPLOYEE (EMPNO, FIRSTNME, MIDINIT,
LASTNAME, EDLEVEL) " + "VALUES (?, ?, ?, ?, ?)";

try (PreparedStatement pstmt = con.prepareStatement(insertSql)) {
  pstmt.setString(1, empNo);
  pstmt.setString(2, firstName);
  pstmt.setString(3, midInit);
  pstmt.setString(4, lastName);
  pstmt.setInt(5, edLevel);
  pstmt.executeUpdate();
  System.out.println("Record inserted successfully.");
}
```

查询语句

```
String selectByIdSql = "SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL " +
            "FROM THEORY.EMPLOYEE WHERE EMPNO = ?";

try (PreparedStatement pstmt = con.prepareStatement(selectByIdSql)) {
```

```
  pstmt.setString(1, empNo);
  try (ResultSet rs = pstmt.executeQuery()) {
    if (rs.next()) {
      System.out.println("EMPNO: " + rs.getString(1));
      System.out.println("FIRSTNME: " + rs.getString(2));
      System.out.println("MIDINIT: " + rs.getString(3));
      System.out.println("LASTNAME: " + rs.getString(4));
      System.out.println("EDLEVEL: " + rs.getInt(5));
    } else {
      System.out.println("Employee with ID " + empNo + " does not exist.");
    }
  }
}
```

更新语句

```
 String updateSql = "UPDATE THEORY.EMPLOYEE SET FIRSTNME = ?, MIDINIT = ?, LASTNAME
= ?, EDLEVEL = ? WHERE EMPNO = ?";

try (PreparedStatement pstmt = con.prepareStatement(updateSql)) {
  pstmt.setString(1, firstName);
  pstmt.setString(2, midInit);
  pstmt.setString(3, lastName);
  pstmt.setInt(4, edLevel);
  pstmt.setString(5, empNo);
  int rowsAffected = pstmt.executeUpdate();
  System.out.println(rowsAffected + " record(s) updated successfully.");
}
```

删除语句

```
 String deleteSql = "DELETE FROM THEORY.EMPLOYEE WHERE EMPNO = ?";

try (PreparedStatement pstmt = con.prepareStatement(deleteSql)) {
  pstmt.setString(1, empNo);
  int rowsAffected = pstmt.executeUpdate();
  System.out.println(rowsAffected + " record(s) deleted successfully.");
}
```

判断用户是否存在

```
String selectByIdSql = "SELECT EMPNO FROM THEORY.EMPLOYEE WHERE EMPNO = ?";

try (PreparedStatement pstmt = con.prepareStatement(selectByIdSql)) {
  pstmt.setString(1, empNo);
  try (ResultSet rs = pstmt.executeQuery()) {
    return rs.next();
  }
}
```

我们编写了一个函数来用于判断用户是否存在，这个函数会在多处被用到

五.sqlj开发

5.1 概述

SQLJ是一种与数据库交互的编程语言扩展，它将SQL语句嵌入到Java代码中，提供了一种方便、类型安全的方式来执行SQL操作。

SQLJ通过在Java代码中使用特定的注释和语法，将SQL语句与Java代码集成在一起。这种集成使得开发人员可以在Java程序中直接使用SQL查询、更新和存储过程等数据库操作，而无需显式编写SQL语句和处理结果集。

我们首先需要定义并编写SQLJ代码，然后编译SQLJ代码，形成java代码，最后运行java程序。

5.2 环境配置

sqlj的环境配置相对来说比较简单

我们调整环境变量后，直接修改Iter.sqlj,运行sqlj Iter.sqlj来编译文件，最后输入java Iter来运行类文件即可。

5.3 开发流程

5.3.1 运行流程

1.修改Iter.sqlj

2.编译sqlj文件

3.运行编译出来的java文件

5.3.2 关键代码

连接数据库

```
Class.forName("com.ibm.db2.jcc.DB2Driver");
Ctx connCtx = new Ctx(DB_URL, USERNAME, PASSWORD, true);
```

插入语句

```
 try {
        #sql [connCtx] {
        INSERT INTO THEORY.EMPLOYEE (EMPNO, FIRSTNME, MIDINIT, LASTNAME, EDLEVEL)
        VALUES (:empNo, :firstName, :midInit, :lastName, :edLevel );
            };
}catch (SQLException e) {
                handleSQLException(e);
                return;
             }
```

查询语句

```
NameIter nIter;
try {
    #sql [connCtx] nIter = {
        SELECT EMPNO, LASTNAME FROM THEORY.EMPLOYEE;
    };

    while (nIter.next()) {
        System.out.println(nIter.LASTNAME() + ", ID #" + nIter.EMPNO());
          }
    nIter.close();
} catch (SQLException e) {
          handleSQLException(e);
          return;
      }
```

我们需要定义一个nIter来接受sql语句的结果，并使用nIter来输出结果：

```
#sql iterator NameIter(String EMPNO,  String LASTNAME);
```

更新语句

```
 try {
    #sql [connCtx] {
        UPDATE THEORY.EMPLOYEE
        SET LASTNAME = :newLastName
        WHERE EMPNO = :empNo;
    };
}catch (SQLException e) {
         handleSQLException(e);
         return;
     }
```

删除语句

```
 try {
    #sql [connCtx] {
        DELETE FROM THEORY.EMPLOYEE WHERE EMPNO = :empNo;
    };
}catch (SQLException e) {
         handleSQLException(e);
         return;
     }
```

判断用户是否存在

```
CountIter cIter;
    try {
        int count = 0;
        #sql [connCtx] cIter =  {
                    SELECT COUNT(*) AS COUNT FROM THEORY.EMPLOYEE WHERE EMPNO =
:empNo;
                };

        while (cIter.next()) {
                count = cIter.COUNT();
                          }

        return count > 0;
    }catch (SQLException e) {
             handleSQLException(e);
             return false;
         }
```

同理，我们书写一个函数来判断用户是否存在，并使用CIter来接受SQL查询的结果。

六.实验心得

在进行大型数据库应用开发的实验中，我们使用了四种与数据库交互的方式，包括Embedded SQL、CLI、JDBC和SQLJ。通过这些方式，我们成功地开发了一个能够与DB2数据库中的sample数据库进行交互的应用程序，并实现了增删改查的功能，并提供了一个主菜单来方便地调用这些功能。

这次实验让我深入了解了不同的数据库交互方式以及它们的特点和用法。以下是我在实验中的一些心得体会：

1. **Embedded SQL**: 通过在应用程序中嵌入SQL语句，使用预编译和绑定变量的方式与数据库进行交互。这种方式相对简单直接，能够在应用程序中直接使用SQL语句，但需要注意安全性和可维护性。
2. **CLI (Call Level Interface)**: CLI提供了一组函数调用来执行SQL语句和处理数据库操作。使用CLI需要手动编写调用和参数传递的代码，相对较低级，但更加灵活，可以更精确地控制数据库操作。
3. **JDBC (Java Database Connectivity)**: JDBC是Java语言与数据库交互的标准接口，提供了一套面向对象的API，使得Java应用程序可以通过JDBC驱动程序与数据库进行通信。JDBC提供了高级别的抽象，简化了数据库操作的编码，且具有较好的跨平台性。
4. **SQLJ (SQL in Java)**: SQLJ允许将SQL语句嵌入到Java代码中，并提供了类型安全和编译时检查的优势。SQLJ使得数据库操作更加直观和易于维护，但需要使用SQLJ编译器和运行时库进行支持。

在开发过程中，我注意到了不同方式的特点和适用场景。Embedded SQL适用于简单的SQL操作，CLI适用于对数据库操作有更精确控制需求的情况，JDBC适用于Java应用程序与数据库的通用交互，而SQLJ则提供了一种更加直观和类型安全的方式来处理SQL操作。

同时，在开发过程中，我们需要对这四种方式分别配置环境，在配置环境的过程中，我也对db2数据库等方面有了一个更深的了解。尽管配置环境中可能会遇到一些困难，但当困难解决后，我发现我对知识的了解也更深了。

总结起来，通过本次大型数据库应用开发的实验，我对于不同的数据库交互方式有了更深入的了解，并学会了如何使用这些方式与DB2数据库进行交互。这对于今后的数据库应用开发和数据管理工作都具有重要的指导意义。