**2024-2025-2学期**

**C++项目报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 文件状态  [ ]草稿  [ ]正式发布  [ ]正在修改 | 文件标识 |  | |
| 当前版本 |  | |
| 小组成员 | （学号） | （姓名） |
| （学号） | （姓名） |
| （学号） | （姓名） |
|  |  |  |
| 完成日期 |  | |

电子科技大学信息与软件工程学院

版本历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本/状态 | 作者 | 参与者 | 起止日期 | 备注 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[第一章 引言 1](#_Toc147653016)

[1.1 项目背景 1](#_Toc147653017)

[1.2 术语与缩写解释 1](#_Toc147653018)

[第二章 系统需求 2](#_Toc147653019)

[第三章 系统设计 3](#_Toc147653020)

[第四章 系统实现 4](#_Toc147653021)

[第五章 系统测试 5](#_Toc147653022)

[第六章 结语 6](#_Toc147653023)

[参考文献 7](#_Toc147653024)

# 引言

（报告正文内容不少于2000字。报告写好后删掉红色文字。）

## 1.1 项目背景

本项目是《C++现代程序设计》课程的期末综合性大作业 。项目的核心目标是运用本学期所学的C++知识群，结合其他相关计算机科学基础，对标业界成熟的关系型数据库管理系统MySQL，自主设计并开发一个微型数据库管理系统（DBMS） 。

该项目要求我们以小组形式进行，通过明确的分工协作，共同完成从需求分析、系统设计、编码实现到最终测试的全过程 。开发过程中，我们被要求采用面向对象分析（OOA）与设计（OOD）的方法 ，并遵循C++20及以上的新标准进行编码 。这不仅是对我们C++编程能力的一次全面检验和实践，更是深化我们对数据库系统底层工作原理（如SQL解析、数据存储、索引机制等）理解的宝贵机会。通过完成此项目，我们旨在巩固理论知识，并锻炼解决复杂问题的工程实践能力。

（简要介绍项目背景。不少于100字）

## 1.2 术语与缩写解释

|  |  |
| --- | --- |
| 术语/缩写 | 解释 |
| DBMS | Database Management System (数据库管理系统)的缩写。 |
| DDL | Data Definition Language (数据定义语言)，SQL的一部分，用于定义数据结构，如 CREATE TABLE。 |
| DML | Data Manipulation Language (数据操纵语言)，SQL的一部分，用于管理数据，如 SELECT, INSERT。Object-Oriented Analysis / Object-Oriented Design (面向对象的分析与设计)。 |
| OOA/OOD | Object-Oriented Analysis / Object-Oriented Design (面向对象的分析与设计)。 |

（除以上示例的两节外，可根据实际情况自行添加。以下同此。）

# 第二章 系统需求

（说明系统的需求，需要画出系统的功能结构框图并做简要介绍。

提出系统要达到的预期目标。

必要是可以列出二级或更多级标题。

必须采用OOA方法。

以下同。）

本章将根据项目任务书的要求，采用面向对象分析（OOA）的方法，详细阐述该微型数据库管理系统的功能性与非功能性需求，并明确系统需要达成的预期目标 。

#### ****2.1 功能需求****

系统需要实现一个简化版的数据库管理系统，核心功能围绕数据定义、数据操作、数据存储和用户交互四个方面展开。

##### ****2.1.1 存储功能****

* 文件系统存储：系统需利用宿主操作系统（Linux）的文件系统来持久化数据。
* 文件与目录结构：数据表和索引（如果存在）应以独立文件的形式存储。 目录结构和文件格式可自行设计，例如，可以为每个数据库创建一个主目录，内部存放该库的表文件和索引文件 。

##### ****2.1.2 数据定义语言 (DDL)****

系统必须支持以下DDL语句：

* create database <dbname>：用于创建一个新的数据库。
* drop database <dbname>：用于删除一个已存在的数据库。
* use <dbname>：用于切换当前工作的数据库上下文。
* create table <table-name>(<column> <type> [primary], ...)：在当前数据库中创建新表。
  + 命名规则：表名和列名需为全英文小写，且不包含下划线 \_ 和特殊字符。
  + 数据类型：简化支持 int 和 string 两种类型。string为最长256个字符的定长UTF-8编码字符串。
  + 主键与索引：若某列被指定为 primary（主键），系统必须为该表建立索引。索引的数据结构可自行确定。
* drop table <table-name>：删除一个表，并同时删除其对应的索引文件（如果存在）。

##### ****2.1.3 数据操作语言 (DML)****

系统必须支持以下DML语句：

* select <column> from <table> [ where <cond> ]：查询数据。
  + 列选择：支持查询单个指定列或使用 \* 查询所有列。
  + 条件子句 where：可选，用于过滤数据。 条件表达式 <cond> 的格式为 <column> <op> <const-value>，其中操作符 <op> 支持 =、<、> 三种。
  + 索引利用：在执行查询时，如果表存在索引，应优先使用索引以提高效率。
* insert <table> values (<const-value>[, <const-value>...])：向表中插入一条新记录。字符串常量需要用双引号 "" 括起来。
* delete <table> [ where <cond> ]：根据 where 子句的条件删除表中的记录。
* update <table> set <column> = <const-value> [ where <cond> ]：根据 where 子句的条件更新记录。若无 where 子句，则更新整张表。字符串常量同样需要使用双引号。

##### ****2.1.4 交互界面****

* 界面模式：需要模仿MySQL的命令行交互式界面。
* 操作反馈：每条命令执行后，都必须向用户提供清晰、明确的反馈信息。
* 退出机制：提供 exit 命令，用于正常退出程序。

#### ****2.2 非功能性需求****

* 开发平台：项目必须在Linux操作系统下进行开发和编译。
* 编程语言：代码主体必须是C++，且C++代码量需占压倒性比例。
* C++标准：代码需至少符合C++20标准，鼓励使用C++23。
* 外部库限制：允许使用STL库（包括多线程库），但禁止使用任何其他第三方库，如网络库。应用定位为单机程序。
* 构建工具：必须使用 make 工具来构建应用程序，并提供名为 Makefile 的默认依赖文件。
* 错误处理：需要自行设计所有可能出现的错误（如表不存在、语法错误等）及其处理机制。
* 代码托管：强烈建议使用Git进行版本控制。

#### ****2.3 系统功能结构框图****

* **微型数据库管理系统 (MiniDBMS)**
  + **用户接口层 (User Interface)** (**client**)
    - 负责接收用户输入的SQL命令字符串。
    - 显示命令执行结果或错误信息。
  + **SQL解析层 (SQL Parser)** **(client)**
    - 对输入的命令字符串进行词法分析和语法分析。
    - 将合法的命令解析成内部数据结构，传递给执行器。
  + **命令执行层 (Command Executor)** **(server)**
    - **DDL执行器**：处理 create, drop, use 等数据库和表结构定义命令。
    - **DML执行器**：处理 select, insert, delete, update 等数据操作命令。
  + **存储引擎层 (Storage Engine)** **(server)**
    - **文件管理器**：负责数据库、表、索引等文件的创建、删除和读写。
    - **数据管理器**：负责记录的增、删、改、查等具体操作。
    - **索引管理器**：负责索引的创建、维护和使用，以加速数据检索。

#### ****2.4 预期目标****

* 成功实现任务书中要求的全部5条DDL和4条DML语句的功能。
* 提供一个稳定、流畅的命令行交互界面，并能对用户操作给出正确反馈。
* 建立一套合理的文件存储机制，能正确地持久化数据库、表和索引信息。
* 确保数据操作的准确性，查询结果符合预期。
* 设计全面的测试用例，覆盖所有核心功能，并通过测试。
* 系统具备基本的错误处理能力，对于非法操作或输入能给出提示，不会轻易崩溃。

# 第三章 系统设计

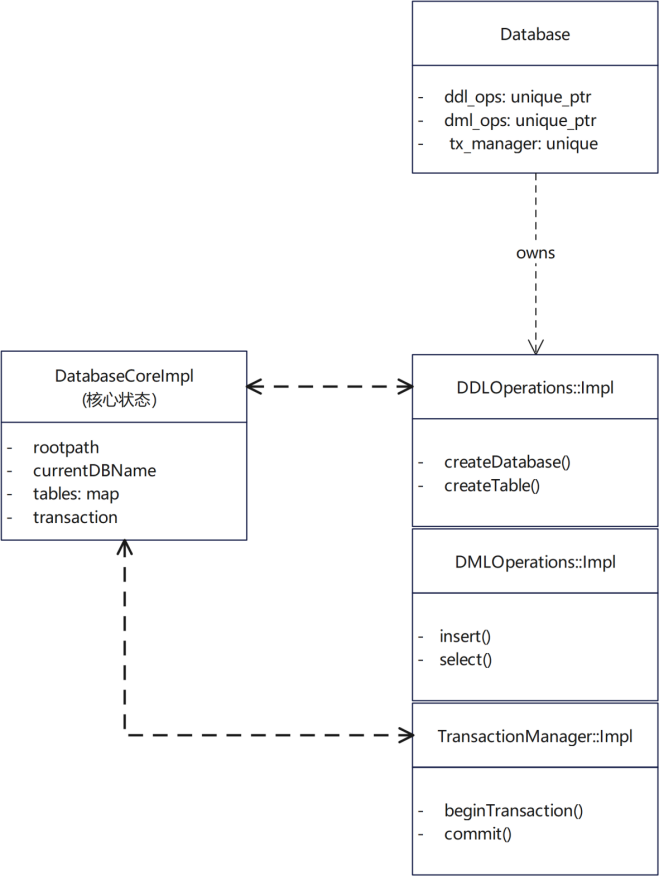
（说明你是如何对系统进行设计的，包括系统结构设计、模块设计、类图、界面设计和功能设计等。必须采用OOD方法。）

本章采用面向对象设计(OOD)方法，聚焦服务层核心架构，同时兼顾网络层和客户层的设计规划。系统整体采用分层架构和Pimpl模式，确保模块化、可扩展性和实现隐藏。

### 3.1 服务层架构设计​​

服务层采用​**​核心-模块​**​架构，通过Database类作为统一入口，各功能模块通过裸指针共享核心状态。

#### ​​3.1.1 类图设计​​



**关键设计决策​**​：

1. ​**​Pimpl惯用法​**​：
   * 各模块(DDL/DML/TX)均采用Impl内部类实现核心逻辑
   * 公共接口类仅保留unique\_ptr<Impl>成员
   * 实现修改不影响头文件，减少编译依赖
2. ​**​核心状态共享​**​：
   * DatabaseCoreImpl封装所有共享状态（当前数据库、表数据、事务状态）
   * 模块通过裸指针访问核心状态（DatabaseCoreImpl\*）
   * 生命周期由Database类统一管理
3. ​**​文件存储策略​**​：

|  |
| --- |
| db\_root/  └── school\_db/  ├── students.meta # 表结构 (CSV格式)  ├── students.dat # 表数据 (CSV格式)  └── students.idx # 主键索引 (二叉搜索树序列化) |

### 3.2 核心模块设计​​

#### ​​3.2.1 DDL操作模块​​

|  |
| --- |
| class DDLOperations::Impl {  public:  bool createDatabase(const string& name);  bool useDatabase(const string& name);  bool createTable(const string& name,  const vector<ColumnDefinition>& cols);  private:  DatabaseCoreImpl\* core; // 核心状态访问  }; |

* ​**​原子操作保障​**​：数据库/表创建采用"创建目录→写入文件→内存加载"三步原子操作
* ​**​错误恢复​**​：文件操作异常时自动回滚（如删除半成品文件）

#### ​​3.2.2 DML操作模块​​

|  |
| --- |
| class DMLOperations::Impl {  public:  int insert(const string& table,  const map<string, string>& values);  unique\_ptr<QueryResult> select(...);  private:  // 条件评估引擎  bool evaluateCondition(const Row& row,  const string& condition);  }; |

* ​**​条件评估器​**​：支持多条件组合（AND/OR）和类型安全比较
* ​**​查询优化​**​：主键查询自动使用索引（.idx文件反序列化为内存BST）

#### ​​3.2.3 事务管理模块​​

|  |
| --- |
| class TransactionManager::Impl {  public:  void beginTransaction();  void commit();  void rollback();  private:  void writeLogEntry(LogType type,  const string& data);  }; |

* **日志格式​**​：

|  |
| --- |
| [操作类型];[表名];[旧值];[新值]  UPDATE;students;101,Alice,18;101,Alice,19 |

* ​**​崩溃恢复：**启动时检查未提交日志，自动执行redo

### 3.3 网络层设计

尽管当前实现聚焦服务层，网络层已定义清晰接口：

|  |
| --- |
| class NetworkServer {  public:  void start(int port); // 启动服务  void handleRequest(Socket client); // 请求处理  private:  Database& db; // 绑定服务层实例  };  // 应用层协议设计  struct DbProtocol {  uint32\_t length; // 数据长度  uint8\_t commandType; // DDL=0x01, DML=0x02, TX=0x03  char sql[]; // SQL命令文本  }; |

* ​**​线程模型​**​：每个客户端连接分配独立线程
* ​**​数据序列化​**​：查询结果采用TLV（Type-Length-Value）编码

### 3.4 客户层设计

|  |
| --- |
| #ifndef CLI\_APP\_HPP  #define CLI\_APP\_HPP  #include "Parser.hpp"  #include <string>  #include "../network/socket\_client.hpp"  #include "../network/query.hpp"  class CliApp {  private:  std::string current\_database;  std::string username;  std::string password;  bool logged\_in = false;  std::string server\_ip = "127.0.0.1";  short server\_port = 4399;  // 网络相关  NET::SocketClient client;  NET::NetworkQueryExecutor query\_executor;  std::string session\_token;  void execute(const std::string& line);  bool login(const std::string& username, const std::string& password);  void logout();  // DDL Handlers  void handle\_create\_database(const CreateDatabaseCommand& cmd);  void handle\_drop\_database(const DropDatabaseCommand& cmd);  void handle\_use\_database(const UseDatabaseCommand& cmd);  void handle\_create\_table(const CreateTableCommand& cmd);  void handle\_drop\_table(const DropTableCommand& cmd);  // DML Handlers  void handle\_insert(const InsertCommand& cmd);  void handle\_select(const SelectCommand& cmd);  void handle\_update(const UpdateCommand& cmd);  void handle\_delete(const DeleteCommand& cmd);  // 辅助方法  bool executeQuery(const NET::QueryRequest& request);  void handleQueryResponse(const NET::QueryResponse& response);  public:  CliApp() : query\_executor(client) {}  void run();  };  #endif // CLI\_APP\_HPP |

### ​​3.5 关键设计模式应用​​

**工厂方法模式​**​：

|  |
| --- |
| unique\_ptr<QueryResult> createResultSet(ResultType type) {  switch(type) {  case IN\_MEMORY:  return make\_unique<InMemoryQueryResult>();  case STREAMING:  return make\_unique<StreamingResult>(); // 未来扩展  }  } |

**观察者模式​**​（审计日志）：

|  |
| --- |
| class AuditLog : public IOperationObserver {  void onTableCreated(const string& table) override {  log << "CREATED TABLE " << table;  }  };  // 在Database中注册观察者  db.addObserver(make\_unique<AuditLog>()); |

**策略模式​**​（索引引擎）：

|  |
| --- |
| class IndexStrategy {  public:  virtual void createIndex(...) = 0;  virtual bool queryByKey(...) = 0;  };  class BSTStrategy : public IndexStrategy {...}; // 当前实现  class HashStrategy : public IndexStrategy {...}; // 未来扩展 |

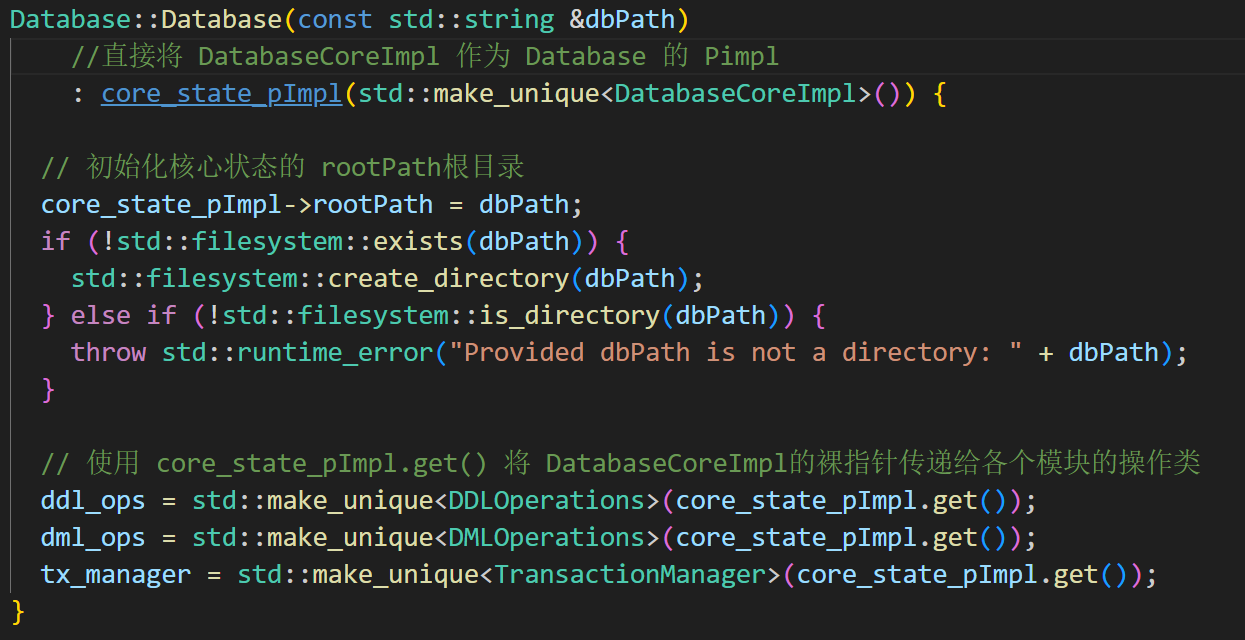
# 第四章 系统实现

（包括主要界面的截图、主要功能的流程图和/或的核心代码。每段代码原则上不超过1页。代码主题必须是C++，但允许用其他语言作为辅助。）

### 4.1 服务层核心实现​​

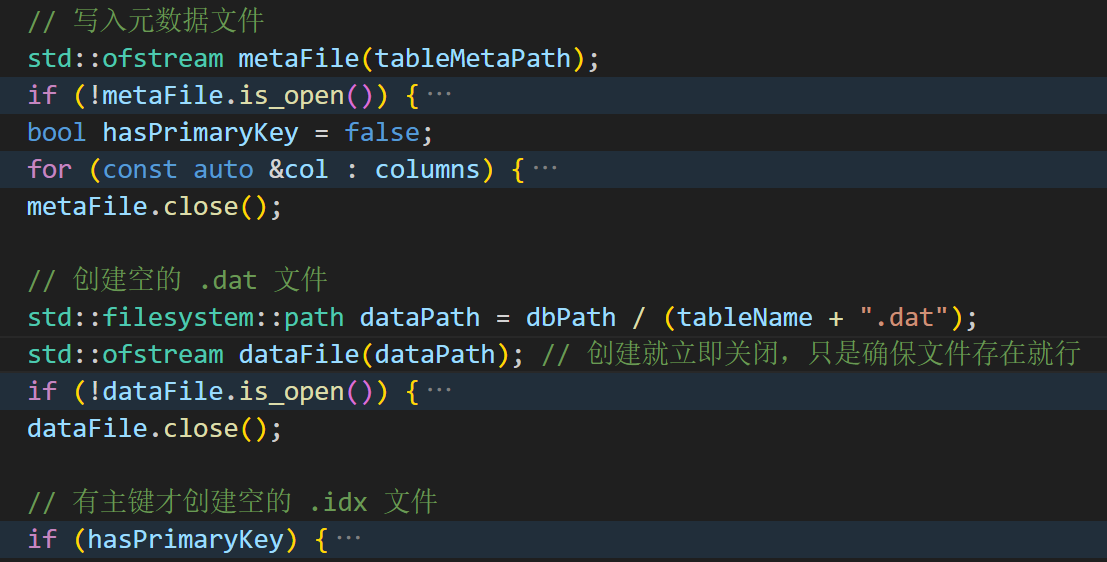
服务层是数据库管理系统的核心，采用Pimpl模式实现接口与实现分离，确保高内聚低耦合。主要包含以下模块：

#### ​​4.1.1 数据库核心管理 (Database.cpp)​​



* ​**​功能​**​：管理数据库根目录，初始化DDL/DML/事务模块
* ​**​关键技术​**​：
  + 使用std::filesystem处理跨平台路径
  + Pimpl模式隐藏DatabaseCoreImpl实现细节
  + 模块间通过裸指针共享核心状态（无所有权传递）

#### ​​4.1.2 DDL操作实现 (DDLOperations.cpp)​​



* **关键能力​**​：
  + 原子性文件操作（创建表需同时生成.meta/.dat/.idx文件）
  + 主键冲突检测（禁止创建多个主键）
  + 切换数据库时自动加载所有表结构

#### ​​4.1.3 DML操作实现 (DMLOperations.cpp)​​

cpp

复制

int DMLOperations::Impl::update(...) {

for (Row &row : table->rows) {

*// 使用条件评估引擎*

if (DMLHelpers::evaluateCondition(row, \*table, whereClause)) {

*// 事务日志记录旧值*

if (core\_impl\_->isTransactionActive)

logOldRow(oldRowDataString);

*// 执行更新*

row[colIndex] = pair.second;

*// 事务日志记录新值*

logNewRow(newRowDataString);

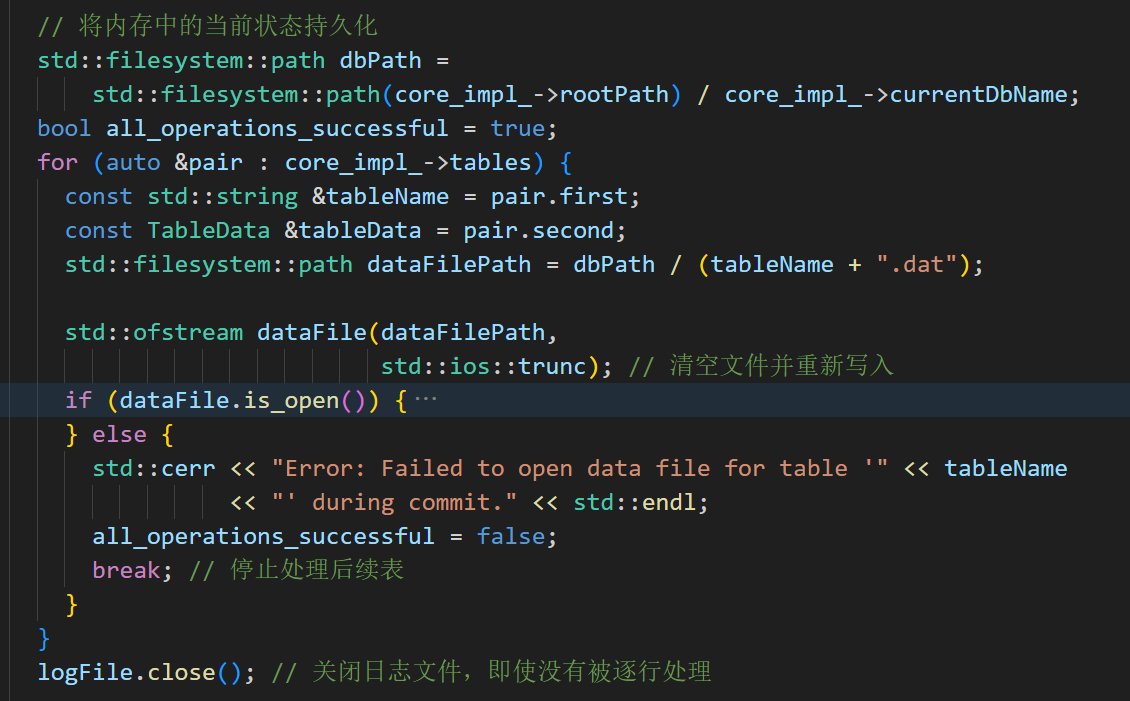
}

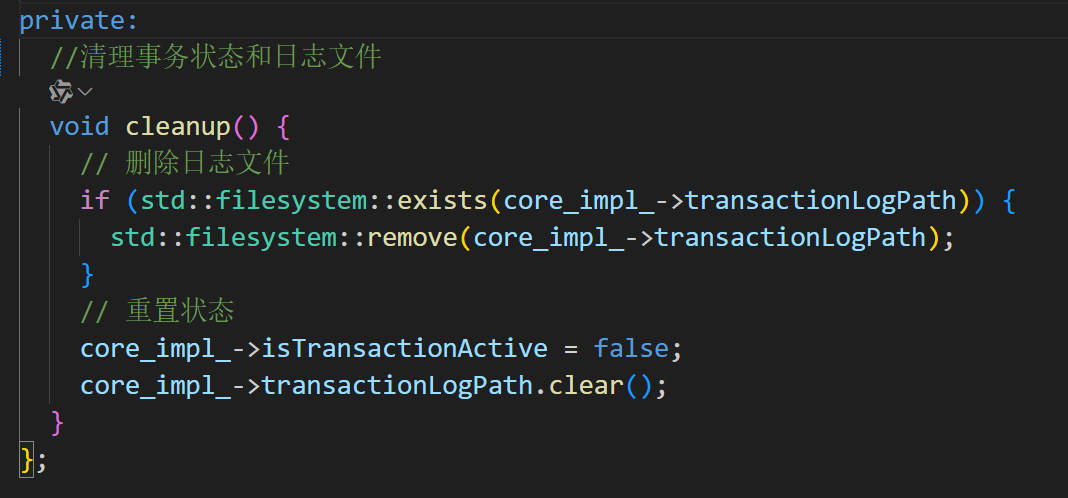
}

}

* ​**​核心技术​**​：
  + ​**​条件评估引擎​**​：支持age>25 AND name='Alice'复杂表达式
  + ​**​类型安全转换​**​：自动处理string↔int/double/bool转换
  + ​**​事务感知​**​：所有操作自动记录undo/redo日志

#### ​​4.1.4 事务管理 (TransactionManager.cpp)​​





* ​**​ACID保障​**​：
  + 原子性：日志记录所有修改
  + 持久性：commit时批量写入磁盘
  + 隔离性：单连接独占事务

### 4.2 网络层设计​

尽管当前实现聚焦服务层，网络层已规划以下设计：

*// 伪代码：网络通信协议*

struct NetworkPacket {

uint32\_t length; *// 数据长度*

uint8\_t type; *// 命令类型（DDL/DML/TX）*

char payload[]; *// SQL命令序列化数据*

};

*// 服务端监听循环*

void Server::run() {

while (auto client = socket.accept()) {

auto packet = receivePacket(client);

DatabaseCommand cmd = parsePacket(packet);

auto result = executeCommand(cmd); *// 调用服务层API*

sendResult(client, result);

}

}

* ​**​协议特点​**​：
  + 定长包头（5字节） + 变长负载
  + 支持异步响应（查询结果分块传输）
  + 加密预留字段（未来扩展TLS）

### ​​4.3 客户层设计​​

命令行界面交互流程：

*> CREATE TABLE Users (id INT PRIMARY, name STRING)*

[SYSTEM] Table 'Users' created

*> INSERT Users VALUES (1, "Alice")*

[SYSTEM] 1 row affected

*> SELECT \* FROM Users WHERE id=1*

+----+-------+

| id | name |

+----+-------+

| 1 | Alice |

+----+-------+

* ​**​关键技术​**​：
  + 表格化输出（自动对齐列宽）
  + 仿mysql语句规范，便于上手

# 第五章 系统测试

（给出测试用例和测试结果，包括性能、有无错误等等。要对标预期结果）

### 5.1 测试环境​

| **组件** | **版本** |
| --- | --- |
| OS | Ubuntu 24.04 |
| Compiler | GCC 14 |
| C++ Standard | C++20 |

### 5.2 服务层核心测试​​

测试用例1：测试 DDL (数据定义语言) 操作**​**​

|  |
| --- |
| std::cout << "\n--- Testing DDL Operations ---" << std::endl;      bool res = ddl\_ops.createDatabase(dbName1);      printTestResult("Create Database '" + dbName1 + "'", res);      res = ddl\_ops.useDatabase("non\_exist\_db");      printTestResult("Use non-existent database", !res);      res = ddl\_ops.useDatabase(dbName1);      printTestResult("Use Database '" + dbName1 + "'", res);      std::vector<ColumnDefinition> studentColumns = {          {"id", DataType::INT, true}, // 主键          {"name", DataType::STRING},          {"age", DataType::INT},          {"grade", DataType::STRING},          {"is\_active", DataType::BOOL}};      res = ddl\_ops.createTable(tableName1, studentColumns);      printTestResult("Create table '" + tableName1 + "'", res); |

结果:

测试用例2：测试 DML (数据操作语言) 操作​

|  |
| --- |
| // 插入数据      std::cout << "\n--- 插入数据到 '" << tableName1 << "' ---" << std::endl;      std::map<std::string, std::string> student1 = {{"id", "101"},                                                     {"name", "Alice"},                                                     {"age", "18"},                                                     {"grade", "A"},                                                     {"is\_active", "true"}};      if (dml\_ops.insert(tableName1, student1)) {        std::cout << "插入 Alice 成功。" << std::endl;        printTestResult("Insert Alice", true);      } else {        std::cerr << "错误：插入 Alice 失败。" << std::endl;        printTestResult("Insert Alice", false);      }      std::map<std::string, std::string> student2 = {{"id", "102"},                                                     {"name", "Bob"},                                                     {"age", "19"},                                                     {"grade", "B"},                                                     {"is\_active", "false"}};      if (dml\_ops.insert(tableName1, student2)) {        std::cout << "插入 Bob 成功。" << std::endl;        printTestResult("Insert Bob", true);      } else {        std::cerr << "错误：插入 Bob 失败。" << std::endl;        printTestResult("Insert Bob", false);      }      std::map<std::string, std::string> student3 = {{"id", "103"},                                                     {"name", "Charlie"},                                                     {"age", "18"},                                                     {"grade", "A"},                                                     {"is\_active", "true"}};      if (dml\_ops.insert(tableName1, student3)) {        std::cout << "插入 Charlie 成功。" << std::endl;        printTestResult("Insert Charlie", true);      } else {        std::cerr << "错误：插入 Charlie 失败。" << std::endl;        printTestResult("Insert Charlie", false);      }      // 尝试插入重复主键      std::map<std::string, std::string> student\_dup = {          {"id", "101"}, {"name", "Eve"}, {"age", "20"}, {"grade", "C"}};      bool dup\_insert\_res = dml\_ops.insert(tableName1, student\_dup) == 0;      printTestResult("Insert duplicate primary key (id=101)", dup\_insert\_res);      // 查询所有数据      std::cout << "\n--- 查询所有学生 ---" << std::endl;      auto allStudents = dml\_ops.select(tableName1);      printQueryResult(allStudents);      printTestResult("Select all students",                      allStudents && allStudents->getRowCount() == 3);      // 带条件查询      std::cout << "\n--- 查询 age = 18 且 grade = 'A' 的学生 ---" << std::endl;      auto filteredStudents =          dml\_ops.select(tableName1, "age = 18 AND grade = 'A'");      printQueryResult(filteredStudents);      printTestResult("Select students with age=18 AND grade=A",                      filteredStudents && filteredStudents->getRowCount() == 2);      std::cout << "\n--- 查询 is\_active = true 的学生 ---" << std::endl;      auto activeStudents = dml\_ops.select(tableName1, "is\_active = 'true'");      printQueryResult(activeStudents);      printTestResult("Select active students",                      activeStudents && activeStudents->getRowCount() == 2);      // 带排序查询      std::cout << "\n--- 查询所有学生并按 age 排序 ---" << std::endl;      auto sortedByAge = dml\_ops.select(tableName1, "", "age");      printQueryResult(sortedByAge);      // 验证排序：手动检查输出或更复杂的断言      printTestResult("Select all students ordered by age",                      sortedByAge != nullptr);      std::cout << "\n--- 查询所有学生并按 name 排序 ---" << std::endl;      auto sortedByName = dml\_ops.select(tableName1, "", "name");      printQueryResult(sortedByName);      printTestResult("Select all students ordered by name",                      sortedByName != nullptr);      // 更新数据      std::cout << "\n--- 更新 Bob 的 age 为 20 ---" << std::endl;      std::map<std::string, std::string> updates = {{"age", "20"}};      int updatedRows = dml\_ops.update(tableName1, updates, "name = 'Bob'");      printTestResult("Update Bob's age to 20", updatedRows == 1);      std::cout << "\n--- 再次查询所有学生以验证更新 ---" << std::endl;      allStudents = dml\_ops.select(tableName1);      printQueryResult(allStudents);      // 删除数据      std::cout << "\n--- 删除 age < 20 的学生 ---" << std::endl;      int removedRows = dml\_ops.remove(tableName1, "age < 20");      printTestResult("Remove students with age < 20",                      removedRows == 2); // Alice and Charlie should be deleted      std::cout << "\n--- 再次查询所有学生以验证删除 ---" << std::endl;      allStudents = dml\_ops.select(tableName1);      printQueryResult(allStudents);      printTestResult("Select all students after removal",                      allStudents && allStudents->getRowCount() == 1); |

结果：

测试用例3：测试 TransactionManager (事务管理) 操作

|  |
| --- |
| // 创建新的数据库和表用于事务测试      ddl\_ops.createDatabase(dbName2);      ddl\_ops.useDatabase(dbName2);      std::vector<ColumnDefinition> employeeColumns = {          {"id", DataType::INT, true},          {"name", DataType::STRING},          {"salary", DataType::DOUBLE}};      ddl\_ops.createTable(tableName2, employeeColumns);      printTestResult("Create Database '" + dbName2 + "' and Table '" +                          tableName2 + "' for transactions",                      true); // Assuming success      // 测试正常提交流程      std::cout << "\nTesting successful commit..." << std::endl;      tx\_manager.beginTransaction();      std::map<std::string, std::string> emp1 = {          {"id", "1"}, {"name", "Frank"}, {"salary", "50000.0"}};      dml\_ops.insert(tableName2, emp1);      std::map<std::string, std::string> emp2 = {          {"id", "2"}, {"name", "Grace"}, {"salary", "60000.5"}};      dml\_ops.insert(tableName2, emp2);      std::cout << "事务内插入 Frank 和 Grace。" << std::endl;      tx\_manager.commit();      std::cout << "事务已提交。" << std::endl;      auto employeesAfterCommit = dml\_ops.select(tableName2);      printQueryResult(employeesAfterCommit);      printTestResult("Employees after commit (expected 2 rows)",                      employeesAfterCommit &&                          employeesAfterCommit->getRowCount() == 2);      // 测试回滚流程      std::cout << "\nTesting rollback..." << std::endl;      tx\_manager.beginTransaction();      std::map<std::string, std::string> emp3 = {          {"id", "3"}, {"name", "Heidi"}, {"salary", "75000.0"}};      dml\_ops.insert(tableName2, emp3);      std::cout << "事务内插入 Heidi。" << std::endl;      tx\_manager.rollback();      std::cout << "事务已回滚。Heidi 应该不在表中。" << std::endl;      auto employeesAfterRollback = dml\_ops.select(tableName2);      printQueryResult(employeesAfterRollback);      printTestResult(          "Employees after rollback (expected 2 rows, Heidi not present)",          employeesAfterRollback && employeesAfterRollback->getRowCount() == 2); |

结果：

### 5.3 服务层核心测试

# 第六章 结语

（对项目做出总结。项目的不足、未来展望等等。每位成员的心得体会。）

# 参考文献

（列出参考文献，不少于5篇。格式参照标准格式）

**附1：评分表**

**C++项目评分表**

1. 学生自评

请小组组长填写此表。

|  |  |  |  |
| --- | --- | --- | --- |
| 成员 | 学号 | 承担任务量 | 学生自评等级 |
| Xxx | Xxx | 34% | A |
| Xxx | Xxx | 33% | A- |
| Xxx | Xxx | 33% | B+ |

注：

1). 承担任务量用百分比。标红内容为示例

2). 等级为A+/A/A-/B+/B/B-/C+/C/C-/D+/D/F，其中A+最好，F最差；只有F为不合格等级。

3）自评等级将作为教师评价的参考。

1. 教师评价

**下表由教师填写**

|  |  |  |
| --- | --- | --- |
| 项目完成度 | 报告质量 | 代码质量 |
|  |  |  |

注：最终成绩重，项目完成度占70%，报告质量占15%，代码质量占15%。

教师签字：



**附2：报告撰写格式要求**

1. **标题**
   1. 字体：中文宋体，西文Times New Roman
   2. 字号：1级标题四号，2级和3级标题小四号，粗体
   3. 行间距：固定值，20磅
   4. 缩进：按tab键
2. **正文**
   1. 字体：中文宋体，西文Times New Roman
   2. 字号：小四号
   3. 行间距：固定值，20磅
   4. 缩进：按tab键
   5. 对齐：两端对齐

Tips：选中所有文字，先选择字体为宋体，再选择Times New Roman。

1. **图表**
   1. 字体：中文宋体，西文Times New Roman
   2. 字号：五号
   3. 行间距：单倍行距，上下6磅
   4. 对齐：居中
   5. 位置：表头在表格上方；图题在图下方。另，表格根据窗口宽度自动调整。
   6. 编号：格式类为：章号-章内序号。例如：表1-2 xxx，图2-3 yyy。
   7. 引用：表头和图题必须在正文中有引用，例如：如图2-3所示。
2. **代码**
   1. 不用截图，粘贴文本
   2. 字体：Consolas
   3. 字号：小五号
   4. 行间距：固定值，12磅
   5. C:\Users\白忠建\AppData\Roaming\Tencent\Users\253766161\QQ\WinTemp\RichOle\Q5S%UB3AO[40U`D(}2JL7[S.png加框。选中要加框的代码，选择“开始菜单”，在工具图标栏中部可以看到这个图标：

点击它，在下拉列表中选择外侧框线。

例：

void SocketError( int nErrorCode )

{

CString errmsg;

switch ( nErrorCode )

{

case WSANOTINITIALISED:

errmsg = "WSANOTINIT";

break;

Tip：粘贴代码前，在粘贴位置按Del，直到光标移动到文档最左侧，然后再粘贴，这样可以保持代码原来的缩进格式。此后再调整字体等。

**注意：不按格式写的报告评分至少要降一个等级，甚至评为不合格！**