simple-db

以下介绍了本数据库框架的基本结构及开发接口。在代码源文件中还有进一步的说明。

data_t (data_t.h/cpp)

封装数据库支持的基本数据类型，并存储数据。data_t 本身为抽象类，不能创建对象，全部具体功能由派生出的子类实现。

目前支持三种 MySQL 基本数据类型：int、double、char，通过相应的派生类 dataInt、dataDouble、dataString 实现。以下，当不区分具体是哪个派生类时，用 dataX 表示一个派生类，用 X 表示对应的 C++ 数据类型。

```
private: X dataX::value
```

存储一个 MySQL 数据。在重载比较运算符中被使用，详细说明见后。注意该成员不在基类 data_t 中。

```
public: dataX::dataX()
```

根据参数构造相应类型的 data_t 对象。

参数

三种派生类型的构造函数均只接受 1 个参数：

- 对于 dataInt 及 dataDouble 类型，可以使用相应类型的 MySQL 字面值（std::string 类型）或 C++ 值（int 或 double 类型）构造；
- 对于 dataString 类型，只支持通过 MySQL 字符串字面值构造（std::string 类型，其中包含的 MySQL 特殊字符会被自动转义）。**不支持直接以 C++ std::string 中的内容直接构造 dataString**。

```
public: static data_t* data_t::fromLiteral(std::string)
```

根据 MySQL 字面值（通过参数指定）获取指向相应的 data_t 派生类对象的指针。

备注：若要增加新的数据类型支持，可仿照 dataInt 等类创建 data_t 的新派生类，并扩充 data_t::fromLiteral() 函数。

```
public: virtual dataX* dataX::copy()
```

根据当前对象存储的内容，复制一个新的对象，并返回新对象的指针。

备注：该函数为 `data_t* data_t::copy() = 0` 的重写覆盖。参考协变返回类型。

```
public: virtual std::string dataX::get() const;
```

将对象存储的内容转换为字符串，并返回该字符串。

```
public: virtual bool dataX::operator < (const data_t&)
```

比较两个 `data_t` 对象存储内容的大小。

备注

1. 同类型的值都可比较大小；不同类型的值仅当有意义时才可比较大小，否则会抛出异常。
2. 为方便起见，我们提供了大于号 (>) 及等号 (==) 运算符的默认重载（位于 `data_t::operator >` 或 `data_t::operator ==` 中，会通过小于运算符的结果进行大于、等于的比较）。
3. 由于 C++ 语法的一些限制，很难使得二元比较运算符的两边同时多态。为了便于扩展，我们在 data_t.cpp 文件中，提供了 `compareHelper` 宏。该宏判断待比较的 `data_t` 参数的类型，并在类型正确时返回相应的比较结果；接收两个参数，第一个参数表示要判断的 `data_t` 派生类型，第二个参数表示对 `value` 成员变量的转换规则（便于大小写不敏感的字符串比较等场合；不需要转换规则时可省略该参数）。若出现了没有意义的比较，将抛出异常。具体请参见 data_t.cpp 中三种派生类型的比较运算符写法。

---

cond_t (cond_t.h/cpp)

函数（`std::function<bool(const Entry&)>`）对象，定义了判断一个 `Entry`（见后）是否符合一定条件的规则，应用于 `WHERE` 子句中。

`cond_t` 对象定义了与 (and)、或 (or)、非 (not) 逻辑运算，可将多个简单的 `cond_t` 函数复合成更复杂的 `cond_t` 函数。同时也定义了赋值与 `&=`、赋值或 `|=` 操作符。

```
cond_t constCond(bool);
```

根据布尔值 `true` 或 `false`（由参数指定）返回一个总是返回该布尔值的函数。

注释

`cond_t` 对象可通过相应参数和返回值类型的 Lambda 表达式赋值，也可直接通过 `std::function` 内置重载的 `()` 运算符运行保存的函数并求值。详细用法见后。

---

set_t (set_t.h)

函数（`std::function<void(Entry&>)`）对象，定义了修改一条 `Entry`（见后）的方法，应用于 `SET` 子句中。也可通过 Lambda 表达式赋值、通过 `()` 运算符执行。详细用法见后。

`Entry` (entry.h/cpp)

```
class Entry: public std::vector<data_t*>
```

定义了数据表中的一行，大部分用法同 `vector` 。只能移动构造、移动赋值，为避免误操作，禁止复制构造、复制赋值。复制请使用 `Entry::copy()` 。

```
Entry Entry::copy();
```

将当前行复制，返回复制的新行。

`Table` (table.h/cpp)

数据表，存储表中每列的类型、属性，以及每行的数据内容。

```
public: Table::Table(const tokens& attrClause)
```

根据参数内容构造具有相应结构的空表。`attrClause` 形如 `split("stu_id INT NOT NULL, stu_name CHAR, PRIMARY KEY(stu_id)")` 。

```
public: attrs Table::attrList()
```

返回表中各列名称，返回值形如 `{"attrName1","attrName2"}` 。

```
public: int Table::insert(const tokens& attrNames, const tokens& attrValues)
```

向表中插入一行数据。返回插入的行数（即1）。

`attrNames` 形如 `split("attrName1, attrName2")`（也即 `{"attrName1", ",", "attrName2"}` ）。

`attrValues` 形如 `split("3.14, 'w'")`。

注意：目前该函数不会判断插入的数据是否违反了约束（`NOT NULL`, `PRIMARY KEY`）。

```
public: int Table::remove([const tokens& whereClause])
```

删除表中数据。参数指定删除条件。若省略该参数，删除表中全部数据，但保留表的结构。返回删除的行数。

`whereClause` 形如 `split("id=10492 or name='q'")`，下同。

```
public: int Table::update(const tokens& setClause [, const tokens& whereClause])
```

setClause 形如 split("stu_name='b'")

修改表中数据。第一个参数指定修改方法，第二个指定修改条件。若省略第二个参数，则对所有记录进行修改。返回修改的行数。

```
public: int Table::select(const attrs& attrName [, const tokens& whereClause])
```

attrName 形如 {"name","id"}。

查询表中数据并输出。第一个参数指定要输出的列，第二个指定查询条件。若省略第二个参数，输出表中全部数据。返回被输出的行数。

```
public: void Table::show()
```

输出表的结构，格式与 SHOW COLUMNS FROM table 相同。

```
public: void Table::sort(std::string attrName)
```

对表中数据进行排序。参数指定根据哪一列进行排序。若省略该参数，则按主键排序。若省略该参数且表无主键，则什么都不做。

Database (database.h/cpp)

定义了数据库对象。包含数据库名称及其各表。

```
private: std::string Database::dbName
```

保存数据库的名称。

```
public: std::map<std::string, Table*> Database::table
```

保存表名到指向表的指针的映射。

```
public: Database::Database(std::string dbName)
```

构造空数据库。参数指定表的名称。

```
public: void Database::drop(std::string tableName)
```

删除一张表。参数指定要删的表的名称。

```
public: void Database::create(std::string tableName, const tokens& traits)
```

创建新表。第一个参数指定表的名称。第二个参数指定表的各列属性。

```
public: void show()
```

输出数据库中包含的各表。

```
public: void show(std::string tableName)
```

输出指定表的结构，格式与 `SHOW COLUMNS FROM table` 相同。

```
public: Table* Database::operator [] (std::string)
```

根据名称返回相应表的指针。实质上直接访问了 `Database::table` 映射，是 `Database::table.operator[] (std::string)` 的简化写法。

Tools (tools.h/cpp)

提供其它模块中用到的辅助工具。

```
typedef std::vector<std::string> tokens
```

一列若干个字符串。用于 MySQL 语法解析，此时每个元素都是一个 MySQL 语法中的符号（当前包含数字、标识符、关键字、运算符、字符串、分隔符六种），并且将整个 `tokens` 中的全部符号按顺序连在一起即得一条完整的 MySQL 语句或语句中用到的完整的子句。

```
typedef std::vector<std::string> attrs
```

一列若干个字符串。本质上与 `tokens` 是同一类型，但在本数据库中含义不同。`attrs` 用于保存若干个列的名称的集合，中间不需要任何分隔符（如果完全按照 `tokens` 的语义，需要在各列之间加逗号分开）。

```
std::vector<std::string> split(std::string)
```

解析 MySQL 语句，将其切分为 MySQL 中最小的语法单位（token，单词/符号）并返回。返回的数组中，每个元素都一定是数字、标识符（数据库名、表名、列名）、关键字（例如 `SELECT`、`WHERE` 等）、运算符（`AND`、`>` 等）、字符串、分隔符（例如括号、逗号、分号等）之一。

注意：本数据库当前不支持非布尔运算。负数被认为是一个整体，而不是被拆分为负号和绝对值两个 token；MySQL 语法中，两个紧挨一起的字符串会被连接，认为是一个整体，但本数据库程序会将它们认为是两个 token。

```
std::string stringToLower(std::string)
```

将参数中的大写字母全部转换为小写字母。

Client (main.cpp)

主程序，实现了完整的关系型数据库管理系统（RDBMS）。

本文件 main.cpp 同时也是数据库第一阶段的测试代码。

```
void drop(std::string dbName)
```

根据指定的名称删除一个数据库。

```
void create(std::string dbName)
```

创建指定名称的数据库。

```
void use(std::string dbName)
```

选中指定名称的数据库。

```
void show()
```

输出所有数据库的名称。

```
std::map<std::string, Database*> dbList
```

保存数据库名到指向数据库的指针的映射。

```
Database* selected
```

指向当前通过 USE 语句选中的数据库。

```
int main()
```

主程序入口。通过以上各个文件中提供的功能，解析 MySQL 命令并执行对应操作。