数据库说明文档

- 成员函数
  - DataBaseManager::DataBaseManager
  - DataBaseManager::~ DataBaseManager
  - DataBaseManager::Query
  - DataBaseManager::CreateBase
  - DataBaseManager::DropBase
  - DataBaseManager::UseBase
  - DataBaseManager::ShowBase

DataBase.h

- class DataBase
  - 成员变量
  - 成员函数
    - DataBase::DataBase
    - DataBase::~ DataBase
    - DataBase::CreateTable
    - DataBase::DropTable
    - DataBase::ShowTableCol
    - DataBase::ShowTableAll
    - DataBase::InsertData
    - DataBase::DeleteData
    - DataBase::UpdateData
    - DataBase::SelectData

DataTable.h

- class DataTable
  - 成员变量
  - 成员类
  - 成员函数
    - DataTable::DataTable
    - DataTable::~ DataTable
    - DataTable::Insert
    - DataTable::Remove
    - DataTable::Update
    - DataTable::Select
    - DataTable::GetDataWhere
    - DataTable::GetTypeof
    - DataTable::GetPrimaryKey
    - DataTable::GetAttributeTable
    - DataTable::PrintAttributeTable
    - DataTable::CheckPrimaryKey
    - DataTable::CheckNotNullKey
    - DataTable::CheckAttributeName
    - DataTable::PopStack
    - DataTable::CalcExpr
    - DataTable::CheckingSingleClause
    - DataTable::TransValue
    - DataTable::SortData

ParamSpliter.h

- class ParamSpliter
  - 成员变量
  - 成员函数
    - ParamSpliter::Split
    - ParamSpliter::split_use
    - ParamSpliter::split_show

ParamSpliter::split_drop
ParamSpliter::split_delete
ParamSpliter::split_create
ParamSpliter::split_select
ParamSpliter::split_update
ParamSpliter::split_insert
ParamSpliter::split_create_table
ParamSpliter::split_where
errorstream.h
class DataBaseErrorEvent
成员变量
成员函数
DataBaseErrorEvent::DataBaseErrorEvent
DataBaseErrorEvent::getType
DataBaseErrorEvent::getErrorInfo
DataBaseErrorEvent::what()
expression.h
常量
枚举类 OPERATORS
Exprs::oprTYPE
函数
is_logic_oprt
upperized
str_algorithm.h
函数
stralgo::EraseSpace
stralgo::CompressSpace
stralgo::ReplaceMark
stralgo::str2int
stralgo::str2double

数据库说明文档

支持的功能

作为一个基本的数据库，目前她支持的基本操作：

- `CREATE DATABASE DBname`
  - 创建一个名为DBname的数据库。
- `DROP DATABASE DBname`
  - 删除一个名为 `DBname` 的数据库。
- `USE DBname`
  - 将名为 `DBname` 的数据库作为当前使用的数据库。
- `SHOW DATABASES`
  - 输出现有的数据库以及其包含的所有表名。

- `CREATE TABLE tableName(attrName1 Type1, attrName2 Type2, … , attrNameN TypeN NOT NULL, PRIMARY KEY(attrName1))`
  - 创建一个这样子的表。
- `DROP TABLE tableName`
  - 删除名为`tableName`的表。
- `SHOW TABLES`
  - 列出当前数据库的所有表的名字。
- `SHOW columns from tableName`
  - 输出名为`tableName`的表中的所有属性（列）。
- `INSERT INTO [tableName(attrName1, attrName2,…, attrNameN)] VALUES(attrValue1, attrValue2,…, attrValueN)`
  - 插入一个这样的数据。
- `DELETE FROM tableName [WHERE whereClauses]`
  - 从名为`tableName`的表中删除数据，其中`whereClauses`如果不填则我就只好默认你删除所有数据了。
- `UPDATE tableName SET attrName1 = value1 [, attr2Name2 = value2, ...] [WHERE whereClauses]`
  - 仅支持将某些属性**设置**成某些**定值**。~~实现这个函数的人太菜了，不会写表达式求值~~
  - 如果没有where语句则默认`UPDATE`所有数据。
- `SELECT [AttrName1, AttrName2, ...] FROM tableName [WHERE whereClauses]`
  - 如果`[AttrList]`处填`*`则选择所有属性。~~不好意思我们没有写有限状态自动机来搞正则表达式~~
  - 如果没有where语句则默认`SELECT`所有数据。
- 我们并没有实现更多的其它扩展功能，但提供了参数分割器以及简单的~~没什么卵用的~~错误处理机制。

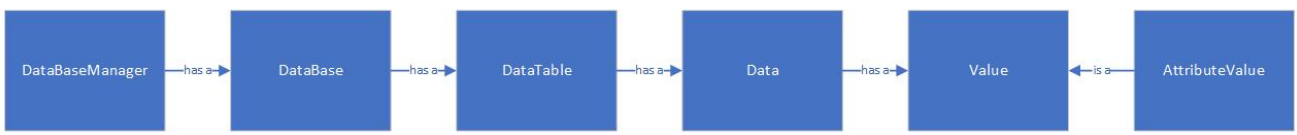
架构

数据库应用程序按照以下的逻辑关系搭建而成：

控制台(DataBaseManager)管理多个数据库(DataBase)，每个数据库(DataBase)内有多个表格(DataTable)，每个表格记录了多条数据(Data)，每个数据(Data)有多种属性组成，属性有属性名和属性值(Value)，不同类型的属性值通过AttributeValue实现。

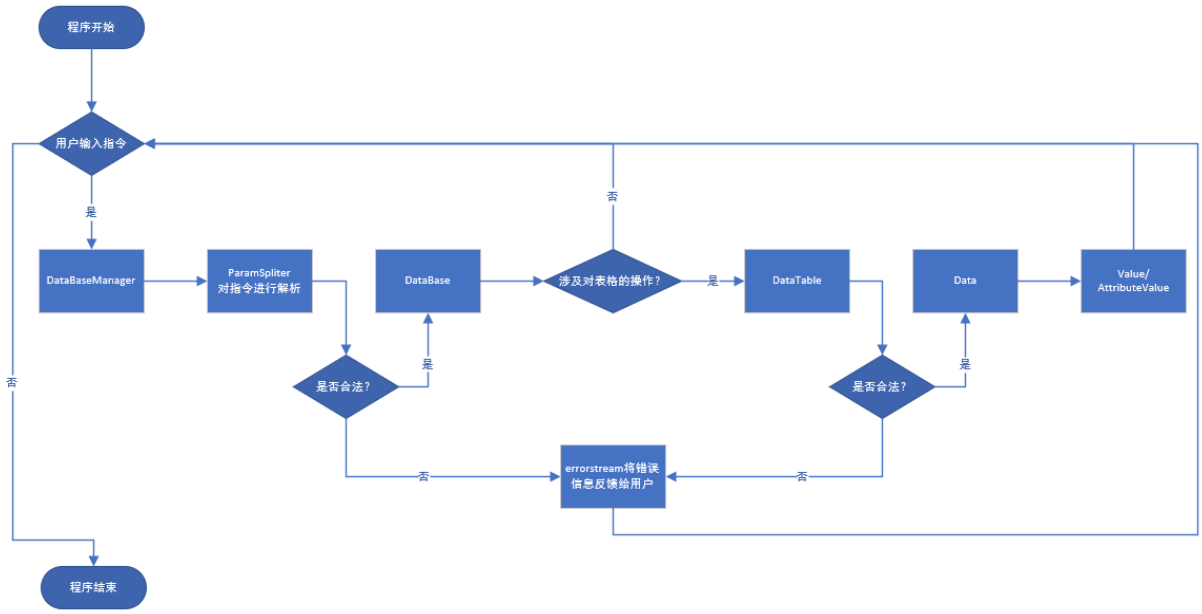
此外，通过ParamSpliter对用户输入的指令进行处理；通过errorstream输出错误信息，提示用户是否出现指令格式错误、数据类型错误等情况。

Value和ParamSpliter通过模板参数的形式作用于类中，因此开发者可以实现其他的数据类型或指令分割规则并运用在程序中。同时，各个类中的主要功能函数都是virtual类型，开发者也可以通过继承的方式对这一部分的函数进行重写。



类名	主要组成部分/作用
DataBaseManager	map {key=DataBaseName, value=DataBase*}
DataBase	map {key=DataTableName, value=DataTable*}
DataTable	list<Data*>
Data	vector<Value*>
Value	属性的值
AttributeValue	继承Value，支持多种数据类型。
ParamSpliter	对用户输入的指令进行处理。
errorstream	继承std::exception，用于输出错误信息。

运行流程



下面举例说明：

- 假设用户输入的指令为`INSERT INTO oop_info(stu_id, stu_name) VALUES (2018011243, "a")`：

  DataBaseManager内调用ParamSpliter中的函数对该指令进行分割，识别出该指令是在表格中插入数据，表格名为oop_info，以及插入的数据的信息；然后调用当前DataBase中表格内插入数据的函数，进入DataTable部分，DataTable再调用插入数据的函数，构造一个Data对象。

- 假设用户输入的指令为`SHOW colunms from oop_info`：

  DataBaseManager内调用ParamSpliter中的函数对该指令进行分割，发现不存在colunms关键字，于是通过errorstream向用户反馈`Error: The form of your command is wrong. Please check your input.`

功能拓展示例

例1. 拓展UPDATE语句的功能

假设需要支持`UPDATE table1 SET attribute1 = attribute + value1`这样的求值式SET。 首先需要更换参数分割的方式，改变split_update函数。

```
class newParamSpliter: public ParamSpliter
{
public:
  int split_update(std::stringstream &ss, std::vector<std::string> &param) override;
};

int newParamSpliter::split_update(std::stringstream &ss, std::vector<std::string>
&param)
{
  // Your new implementation
  return DATA_UPDATE;
}
```

实现完新的split_update后，需要在DataBase中实现新的`UpdateData`。因SET的特性，一定是将某个属性设为某个定值，所以理论上无需在DataTable实现新的`update`。

```
template<class Value, class DataTable, class ParamSpliter>
class newDataBase: public DataBase
{
public:
  newDataBase(std::string name): DataBase<Value,DataTable,ParamSpliter>(name) {}
  void UpdateData(const std::vector<std::string> &param)
  {
    // Your new implementation
    // ......
    // Calculate which attribute to update, find the selected data list.
    for (int i = 0; i < update_attributes.size(); i ++) // only a sample code
    {
      mTable[ param[0] ]->Update(update_attributes[i], selected_data_list);
    }
  }
};
```

最后只需在`DataBaseManager`的参数中使用这些新的类即可：

```
DataBaseManager<Value, newDataBase<Value, DataTable<Value>, newParamSpliter>,
newParamSpliter>* master = new DataBaseManager<Value, newDataBase<Value,
DataTable<Value>, newParamSpliter>, newParamSpliter>();
```

这样在master处理命令时即会调用新的update方式。

例2. 增加支持的数据类型

假设需要增加新的存储的数据类型，以`size_t`为例（假设其枚举类型为`SIZE_T`）。**(新的数据类型应支持<与==符号)** 首先需要对Value类中的`getCopy(),__compare(),operator<<,transValue`作修改（以transValue为例）：

```
class newValue: public Value
{
public:
  newValue* transValue(std::string StrVal, int dataType)
  {
    newValue* pt = NULL;
    switch (dataType)
    {
      case SIZE_T :
      {
        size_t val = 0;
        if (str2size_t(StrVal, val))
          pt = new AttributeValue<size_t>(val);
        break;
      }
      default: {
        pt = Value::transValue(StrVal, dataType);
        break;
      }
    }
    return pt;
  }
};
```

接着要增加`AttributeValue`中支持的数据类型。

```
template<class T>
class newAttr: public AttributeValue<T>, public: newValue
{
public:
  newAttr(T attrValue):Value(), AttributeValue<T>(attrValue) {
      if(typeid(size_t) == typeid(T)){
        _typeName = SIZE_T;
      }
    }
};
```

同时还要新增从“数据类型字符串名到该数据类型的枚举类”的映射，即修改`attrTypeMap`,`attrTypeInvMap`,`attrTypeWidth`这三个map中的值。修改的值的格式可以参考`Value.cpp`中对默认数据类型的初始化。比如：

```
attrTypeMap["SIZE_T"] = SIZE_T;
attrTypeInvMap[SIZE_T] = "SIZE_T";
attrTypeWidth[SIZE_T] = 11;
```

最后在`DataBaseManager`的参数中使用新的类即可。

```
DataBaseManager<newValue, DataBase<newValue, DataTable<Value>, ParamSpliter>,
ParamSpliter>* master = new DataBaseManager<newValue, DataBase<newValue,
DataTable<Value>, ParamSpliter>, ParamSpliter>();
```

运行环境

为了保证本程序正常运行，请确保你的电脑上安装了操作系统。

文件说明

以下内容较长，**建议在使用的时候直接看目录找对应的函数调用**，*不建议直接阅读完以下所有函数*。

Data.h

class Data

是数据表中每行数据的抽象，包含一个 {属性名：属性值(指针)} 的映射

成员变量

- `std::map<std::string, Value*> varMap`：{属性名：属性值(指针)} 的映射，储存该条数据中所有的属性对应的值

成员函数

Data::Data

```
Data();
```

默认构造函数

Data::~ Data

```
~Data();
```

默认析构函数

Data::getValue

```
virtual Value* getValue(std::string attrName) const;
```

用于取得一个属性的值(指针)

- 参数
  - attrName: 待查找属性名
- 返回值
  - 该属性名对应值(指针)，如果未找到则为NULL

Data::setValue

```
virtual Value* setValue(std::string attrName, Value* src);
```

用于设定一个属性的值(指针)

- 参数
  - attrName: 待设定属性名
  - src: 待设定属性值(指针)
- 返回值
  - 成功则返回设定后该值的指针，失败则返回NULL

Data::getTypename

```
virtual int getTypename(std::string attrName) const;
```

用于取得一个属性的值的数据类型

- 参数
  - attrName: 待查找属性名
- 返回值
  - 该属性的值的数据类型(枚举)

Value.h

class Value

是储存数据的基类，派生出class AttributeValue<T>储存具体数据

成员变量

- `_typeName`: 储存该Value储存的数据类型(枚举)
- `std::map<std::string, int> attrTypeMap`: 存储从数据类型的字符串名字到数据类型的映射
- `std::map<int, std::string> attrTypeInvMap`: 存储从数据类型到数据类型名字的映射
- `std::map<int, int> attrTypeWidth`: 我也不知道这个存的应该是什么，总之是从数据类型到“show columns时数据类型后面接的括号里的数字，如果不输出数字则为0”

成员函数

Value::Value

```
Value();
```

默认构造函数

```
Value(const Value& src);
```

拷贝构造函数

Value::~ Value

```
virtual ~Value();
```

默认析构函数

Value::setValue

```
virtual Value* setValue();
```

为子类提供设定数据的接口

Value::getCopy

```
virtual Value* getCopy();
```

用于取得一个当前Value的拷贝(的指针)

- 参数

  无
- 返回值
  - 当前Value对象的拷贝的指针

Value::getTypename

```
int getTypename() const;
```

为子类提供取得储存的数据类型的接口

- 返回值
  - 当前Value对象的_typeName

Value::operator==

```
bool operator==(const Value& b) const;
```

Value::operator!=

```
bool operator!=(const Value& b) const;
```

Value::operator>

```
bool operator>(const Value& b) const;
```

Value::operator<

```
bool operator<(const Value& b) const;
```

Value::operator>=

```
bool operator>=(const Value& b) const;
```

Value::operator<=

```
bool operator<=(const Value& b) const;
```

friend operator<<

```
friend std::ostream& operator<<(std::ostream& out, Value& b);
```

AttributeValue.h

class AttributeValue<T>

派生自class Value

是数据的储存单元，比如一个AttributeValue<int>可以储存一个int

T为数据类型，如int, double等

实际上可以支持任意的类型，只要该类型实现了`operator<`，`operator==`，扩展方式见扩展例子。

成员变量

- `_attrValue`: 储存数据

成员函数

AttributeValue::AttributeValue

```
AttributeValue(T attrValue):Value(), _attrValue(attrValue);
```

构造函数

- 参数
  - attrValue: 待储存的值

```
AttributeValue(const AttributeValue& src);
```

拷贝构造函数

AttributeValue::~ AttributeValue

```
~AttributeValue();
```

默认析构函数

AttributeValue::setValue

```
const AttributeValue<T>* setValue(T attrValue);
```

用于设定一个AttributeValue储存的值

- 参数
  - attrValue: 待储存的值
- 返回值
  - 设定成功则返回该AttributeValue<T>的指针，失败则返回NULL

AttributeValue::getValue

```
T getValue() const;
```

用于取得一个AttributeValue储存的值

- 参数

  无
- 返回值
  - 该AttributeValue储存的_attrValue

AttributeValue::operator==

`bool operator==(const AttributeValue<T>& b) const;`

AttributeValue::operator!=

`bool operator!=(const AttributeValue<T>& b) const;`

AttributeValue::operator>

`bool operator>(const AttributeValue<T>& b) const;`

AttributeValue::operator<

`bool operator<(const AttributeValue<T>& b) const;`

AttributeValue::operator>=

`bool operator>=(const AttributeValue<T>& b) const;`

AttributeValue::operator<=

`bool operator<=(const AttributeValue<T>& b) const;`

DataBaseManager.h

class DataBaseManager<class Value, class DataBase, class ParamSpliter>

成员变量

- `std::map<std::string, DataBase*> mBase`: {数据库名: 数据库指针}映射
- `DataBase* mWorkBase`: 当前数据库指针

成员函数

DataBaseManager::DataBaseManager

```
DataBaseManager();
```

默认构造函数

DataBaseManager::~ DataBaseManager

```
~DataBaseManager();
```

默认析构函数

DataBaseManager::Query

```
void Query(const std::string &command);
```

处理SQL命令

- 参数
  - const std::string &command: SQL命令
- 返回值

  无

DataBaseManager::CreateBase

```
void CreateBase(const std::string &DBname);
```

创建数据库

- 参数
  - DBname: 数据库名
- 返回值

  无

DataBaseManager::DropBase

```
void DropBase(const std::string &DBname);
```

删除数据库

- 参数
  - DBname: 数据库名
- 返回值

无

DataBaseManager::UseBase

```
void UseBase(const std::string &DBname);
```

切换当前数据库

- 参数
  - DBname: 数据库名
- 返回值

  无

DataBaseManager::ShowBase

```
void ShowBase();
```

展示所有数据库与数据表

- 参数

  无
- 返回值

  无

DataBase.h

class DataBase<class Value, class DataTable, class ParamSplitter>

数据库的类。

成员变量

- `mTable`: 数据库中存储的表格。
- `__name`: 数据库的名称。

成员函数

DataBase::DataBase

```
DataBase(cosnt std::string &DBName);
```

DataBase的构造函数。

- 参数
  - DBName: 数据库的名称。

DataBase::~ DataBase

```
~DataBase();
```

DataBase的析构函数。

DataBase::CreateTable

```
void CreateTable(const std::string &command);
```

在该数据库中创建一个表格。

- 参数
  - tableName: 表格的名称。
- 返回值

  无

DataBase::DropTable

```
void DropTable(const std::string &tableName);
```

删除指定名称的表格。

- 参数
  - tableName: 被删除表格的名称。
- 返回值

  无

DataBase::ShowTableCol

```
void ShowTableCol(const std::string &tableName);
```

显示指定名称表格中的属性名称。

- 参数
  - tableName: 表格的名称。
- 返回值

  无

DataBase::ShowTableAll

```
void ShowTableAll(bool PrintBaseName = true);
```

显示该数据库下所有表格的名称。

- 参数
  - PrintBaseName: 是否要显示数据库的名称。
- 返回值

  无

DataBase::InsertData

```
void InsertData(const std::vector<std::string> &param);
```

向该数据库中指定名称的表格中插入数据(Data)。

- 参数
  - param
    - param[0]: 表格的名称。
    - (param[i], param[i+param.size()/2]): 属性名及对应的属性值（用户指令中的string类型）
- 返回值

  无

DataBase::DeleteData

```
void DeleteData(const std::vector<std::string> &param);
```

从该数据库中指定名称的表格中删除数据(Data)。

- 参数
  - param
    - param[0]: 表格的名称。
    - param[1]: （如果有的话）用于定向被删除数据(Data)的where语句。
- 返回值

  无

DataBase::UpdateData

```
void UpdateData(const std::vector<std::string> &param);
```

在该数据库中指定名称的表格中更新数据(Data)。

- 参数
  - param
    - param[0]: 表格的名称。
    - param[2*i+1,2*i+2]: 被更新的属性的属性名以及更新的属性值。
    - param[last]: 用于定向被更新数据(Data)的where语句。
- 返回值

  无

DataBase::SelectData

```
void SelectData(const std::vector<std::string> &param);
```

从该数据库中指定名称的表格中获取指定数据(Data)的指定属性的属性值。

~~数据库层面的Select写得比较丑不好意思hhhh~~

- 参数
  - param

    - param[0]: 表格的名称。
    - param[i]: 制定的属性。
    - param[last]: 用于定向被选中的数据(Data)的where语句。
- 返回值

  无

DataTable.h

class DataTable<class Value>

用于存储表格的对象。

开发者可以使用别的Value类型作为表格中数据(Data)的属性值类型，例如支持更多的数据类型。

成员变量

- `mData_`: 表内存储的数据单元。例如，表格oop_info中各个学生的信息。
- `table_name_`: 表格名称。
- `not_null_key_`: 记录哪些属性为非空。
- `attribute_table_`: 记录表格所有的属性的属性名和属性类型，通过pair绑定在一起。
- `sequential_attribute_table_`: 按照表格创建顺序记录表格的所有属性名和属性类型。

成员类

- class DataCompare: 用于根据某一属性名对表格中数据进行大小比较。

成员函数

DataTable::DataTable

```
DataTable(const std::string& table_name, std::vector< std::pair<std::string, int> >&
attribute_table,const std::string& primary_key, const std::vector<std::string>&
not_null_key);
```

DataTable的构造函数。

- 参数
    - table_name: 表格名称。
    - attribute_table: 表格的属性名及对应的属性类型。

DataTable::~ DataTable

```
~DataTable();
```

DataTable的析构函数。

DataTable::Insert

```
void Insert(const std::vector< ATTRIBUTE >& attributes);
```

向表格中插入一行数据(Data)。

- 参数
  - attributes: 用于初始化被插入数据的属性名及其对应的属性类型。
- 返回值

  无

DataTable::Remove

```
void Remove(std::vector<Data*> &data_list);
```

从表格中删除指定的数据(Data)。

- 参数
  - data_list: 指定要被删除的数据(Data)。
- 返回值

  无

DataTable::Update

```
void Update(const ATTRIBUTE &attribute, std::vector<Data*> &data_list);
```

将表格中指定数据(Data)的某一个属性更新为给定的值。

- 参数
  - attribute: 指定被更新的属性名及用于更新的值。
  - data_list: 指定更新的数据(Data)。
- 返回值

  无

DataTable::Select

```
void Select(const std::string &attribute_name, const std::vector<Data*> &data_list,
std::vector<Value*> &attribute_value);
```

获得指定数据(Data)某一个属性的值。

- 参数
  - attribute_name: 要获取的属性的名称。
  - data_list: 指定的数据(Data)。
  - attribute_value: 用于返回的属性的值。
- 返回值

  无

DataTable::GetDataWhere

```
void GetDataWhere(const std::string &clause, std::vector<Data*> &data_list);
```

将where语句解析为指定的数据(Data)的指针。

例如，where语句为"runoob_id=2018011343"，则将其解析为属性runoob_id的值为2018011343的数据的指针。

- 参数
  - clause: where语句。
  - data_list: 用于返回的指定的数据。
- 返回值

  无

DataTable::GetTypeof

```
int GetTypeof(const std::string &attrName);
```

返回表格中指定属性名对应的属性类型。

- 参数
  - attrName: 制定的属性名。
- 返回值
  - 属性名对应的属性类型（枚举类）。

DataTable::GetPrimaryKey

```
Value* GetPrimaryKey(const Data* data);
```

返回指定数据的主键值。

- 参数
  - data: 指定的数据。
- 返回值
  - 指定数据的主键值(Value*)。

DataTable::GetAttributeTable

```
std::vector< std::pair<std::string, int> > GetAttributeTable();
```

获取表格的属性名和对应的属性类型。

- 参数

  无
- 返回值
  - 表格的属性名和对应的属性类型，通过pair绑定在一起。

DataTable::PrintAttributeTable

```
virtual void PrintAttributeTable();
```

打印表格的属性信息。

- 参数

  无

- 返回值

  无

DataTable::CheckPrimaryKey

```
bool CheckPrimaryKey(const std::vector<ATTRIBUTE>& attributes);
```

检查待插入的数据(Data)的主键是否与现有数据主键重复。

- 参数
  - attributes: 待插入的数据的属性名与对应的属性值。
- 返回值
  - 若不重复，则返回true。

```
bool CheckPrimaryKey(const ATTRIBUTE& attribute);
```

检查将某一个数据(Data)的一个属性修改后是否与现有数据主键重复。

- 参数
  - attribute: 待修改的数据的属性名与对应的属性值。
- 返回值
  - 若不重复，则返回true。

DataTable::CheckNotNullKey

```
virtual bool CheckNotNullKey(const std::vector< Attribute<Value> >& attributes);
```

检查待插入的数据(Data)是否存在非空属性未初始化的情况。

- 参数
  - attributes: 带插入数据的属性名与对应的属性值。
- 返回值
  - 若全部初始化，则返回true。

DataTable::CheckAttributeName

```
virtual bool CheckAttributeName(const std::vector< Attribute<Value> >& attributes);
```

检查待插入的数据(Data)的属性名是否在该表格中存在。

- 参数
  - attributes: 待插入数据(Data)的属性名与对应的属性类型。
- 返回值

    - 若全部存在，则返回true。

```
virtual bool CheckAttributeName(const std::string& attribute_name);
```

检查输入的属性名是否在该表格中存在。

- 参数
    - attribute_name: 输入的属性名。
- 返回值
    - 若存在，则返回true。

DataTable::PopStack

```
virtual void PopStack(std::stack<bool> &val, std::stack<int> &opr);
```

通过变量栈和运算符栈进行计算。

- 参数
    - val: 变量栈。
    - opr: 运算符栈。
- 返回值

  无

DataTable::CalcExpr

```
virtual bool CalcExpr(const Data<Value>* it, const std::string &clause);
```

计算某一个属性值是否满足该表达式。

- 参数
    - it: 属性值。
    - clause: 表达式。

DataTable::CheckingSingleClause

```
virtual bool CheckSingleClause(const Data<Value>* it, const std::vector<std::string>
&_param);
```

计算某一个属性值下单个逻辑表达式的值。

- 参数
    - it: 属性值。
    - _param: 单个表达式。
        - _param[0]: 运算符左侧的字符串。
        - _param[1]: 运算符。
        - _param[2]: 运算符右侧的字符串。
- 返回值
    - 表达式的值。

DataTable::TransValue

```
virtual Value* TransValue(const Data<Value>* _attr, std::string val, int dataType);
```

将表达式中的字符串转换成Value类型。若该字符串为属性名，则将其转换为_attr中对应属性名的属性值；否则通过字符串构造一个Value类型。

- 参数
  - _attr: 一个数据。
  - val: 被转换的字符串。
  - dataType: 字符串被转换为的类型。
- 返回值
  - 字符串转换得到的Value。

DataTable::SortData

```
virtual void SortData();
```

对表格中数据进行排序。

- 参数

  无
- 返回值

  无

ParamSpliter.h

class ParamSpliter

用于解析指令

成员变量

`static std::map<std::string, int> cmdType`: 指令名到指令枚举的映射

成员函数

ParamSpliter::Split

```
virtual int Split(const std::string &Command, std::vector<std::string> &param);
```

将字符指令分割为参数字符数组，并返回Command的枚举

- 参数
  - Command: 待分割指令
  - param: 用于储存分割后参数的数组，具体内容参见各split子函数。
- 返回值
  - 若成功则返回对应的指令的枚举类型，否则返回 `FORM_ERROR`。

ParamSpliter::split_use

```
virtual int split_use(std::stringstream &ss, std::vector<std::string> &param);
```

分割Use指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组
    - param[0]: 需要use的数据库的名字
- 返回值
  - 若成功则返回 `BASE_USE`，否则返回 `FORM_ERROR`。

ParamSpliter::split_show

```
virtual int split_show(std::stringstream &ss, std::vector<std::string> &param);
```

分割show指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组（当且仅当需要SHOW COLUMNS时才非空）
    - param[0]: 当需要SHOW COLUMNS时，param[0]为需要SHOW的表名。
- 返回值
  - 如果是合法的 `SHOW TABLES` 命令，则返回 `TABLE_SHOW_ALL`
  - 如果是合法的 `SHOW COLUMNS` 命令，则返回 `TABLE_SHOW_COL`
  - 如果是合法的 `SHOW DATABASES` 命令，则返回 `BASE_SHOW`
  - 否则返回 `FORM_ERROR`

ParamSpliter::split_drop

```
virtual int split_drop(std::stringstream &ss, std::vector<std::string> &param);
```

分割drop指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组
    - param[0]：需要drop的表或数据库的名字
- 返回值
  - 如果是合法的 `DROP TABLE` 指令，则返回 `TABLE_DROP`
  - 如果是合法的 `DROP DATABASE` 指令，则返回 `BASE_DROP`
  - 否则返回 `FORM_ERROR`

ParamSpliter::split_delete

```
virtual int split_delete(std::stringstream &ss, std::vector<std::string> &param);
```

分割delete指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组
    - param[0]：指定的表的名字
    - param[1]：delete语句的where语句（如果指令中没有显示给出where，则param[1]为一空字符串）
- 返回值
  - 如果格式正确则返回`DATA_DELETE`，否则返回`FORM_ERROR`

ParamSpliter::split_create

```
virtual int split_create(std::stringstream &ss, std::vector<std::string> &param);
```

分割create指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组**（CREATE TABLE的参数在函数split_create_table中处理）**
    - param[0]：创建的 **数据库** 的名字
- 返回值
  - 如果是合法的`CREATE DATABASE`指令，则返回`BASE_CREATE`
  - 如果是`CREATE TABLE`指令，则返回`TABLE_CREATE` **（并未对指令进行分割）**
  - 否则返回`FORM_ERROR`

ParamSpliter::split_select

```
virtual int split_select(std::stringstream &ss, std::vector<std::string> &param);
```

分割select指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组
    - param[0]：指定的表的名字
    - param[1...]（除最后一个元素）：需要选择的属性名
    - param的最后一个元素：select语句的where语句（如果命令中没有显式地给出where，则这是一个空字符串）
- 返回值
  - 如果是合法的`SELECT`指令，则返回`DATA_SELECT`，否则返回`FORM_ERROR`

ParamSpliter::split_update

```
virtual int split_update(std::stringstream &ss, std::vector<std::string> &param);
```

分割update指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组
    - param[0]：指定的表的名字
    - (param[2i+1],param[2i+2])：两两作一个组合，前者是属性的名字，后者是对应属性要设置的值
- 返回值
  - 如果是合法的 `UPDATE` 指令，则返回 `DATA_UPDATE`，否则返回 `FORM_ERROR`

ParamSpliter::split_insert

```
virtual int split_insert(std::stringstream &ss, std::vector<std::string> &param);
```

分割insert指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组
    - param[0]：指定的表的名字
    - (param[i],param[i+param.size()/2])：两两作一个组合，前者是属性的名字，后者是对应属性的值
- 返回值
  - 如果是合法的 `INSERT` 指令，则返回 `DATA_INSERT`，否则返回 `FORM_ERROR`

ParamSpliter::split_create_table

```
virtual int split_create_table(const std::string &Command, std::vector<std::string>
&param, std::vector<std::string> &not_null, std::string &pri_key);
```

分割create table指令

- 参数
  - Commands: 待分割指令
  - param: 用于储存分割后参数的数组
    - param[0]：指定的表的名字
    - (param[2i+1],param[2i+2])：两两作一个组合，前者是属性的名字，后者是对应属性的值
  - not_null: 用于储存非空属性名的数组
  - pri_key: 用于储存主键属性名的数组
- 返回值
  - 如果是合法的 `CREATE TABLE` 指令，则返回 `TABLE_CREATE`，否则返回 `FORM_ERROR`

ParamSpliter::split_where

```
virtual void split_where(std::stringstream &ss, std::vector<std::string> &param);
```

分割where指令

- 参数
  - ss: 待分割指令的stringstream
  - param: 用于储存分割后参数的数组，每个param[i]有两种可能的类型：
    - 运算符；
    - 运算数；
    - 其中运算符与运算数按原指令的顺序排列。
- 返回值

  无

errorstream.h

class DataBaseErrorEvent

用于处理及输出信息的类。继承自`std::exception`。

成员变量

- `type_`: 表示错误事件（枚举类型）。

成员函数

DataBaseErrorEvent::DataBaseErrorEvent

```
DataBaseErrorEvent(): type_(ERROR_NONE);
```

DataBaseErrorEvent的构造函数。

```
DataBaseErrorEvent(int type): type_(type);
```

DataBaseErrorEvent的构造函数。

- 参数
  - type: 错误事件（枚举类型）。

DataBaseErrorEvent::getType

```
int getType() const;
```

返回type_。

- 参数

  无
- 返回值

- type_的值。

DataBaseErrorEvent::getErrorInfo

```
virtual const char* getErrorInfo() const;
```

将错误事件（枚举类型）转换为错误信息字符串。

- 参数

  无

- 返回值
  - 错误信息的字符串。

DataBaseErrorEvent::what()

```
virtual const char* what() const throw();
```

返回错误信息的字符串。

- 参数

  无

- 返回值
  - 错误信息的字符串。

expression.h

这个库原本是用来做表达式的。

由于简化了问题，因此所需要考虑的表达式只有逻辑运算。

常量

枚举类 OPERATORS

存储了逻辑运算符的枚举类。

Exprs::oprTYPE

存储了从逻辑运算符的字符串到其枚举类的映射。 注意在这里等值判断是用单等号，即“=”的。

函数

is_logic_oprt

```
bool is_logic_oprt(std::string opr)
```

判断字符串`opr`是否为逻辑运算符。

- 参数
  - str：待判断字符串。
- 返回值

- 如果是则逻辑运算符"AND","OR","NOT"，则返回true。

upperized

```
std::string upperized(std::string str);
```

将字符串str中的小写字母全部换成大写字母。

- 参数
  - str：待处理字符串。
- 返回值
  - 返回处理完后的字符串。

str_algorithm.h

这个库提供一些简单的字符串处理函数。

函数

stralgo::EraseSpace

```
void EraseSpace(std::string &str);
```

删除换行符与空格

- 参数
  - str: 待处理字符串
- 返回值

  无

stralgo::CompressSpace

```
void CompressSpace(const std::string &src, std::string &dst);
```

压缩空格数

- 参数
  - src: 原字符串
  - dst: 压缩后字符串
- 返回值

  无

stralgo::ReplaceMark

```
void ReplaceMark(const std::string &src, std::string &dst);
```

将'('')'',''';'字符转换为空格

- 参数
  - src: 原字符串
  - dst: 转换后字符串
- 返回值

  无

stralgo::str2int

```
bool str2int(const std::string &str, int &val);
```

字符串转整型

- 参数
  - str: 待转换字符串
  - val: 转换后值
- 返回值
  - 是否转换成功的布尔变量

```
int str2int(const std::string &str);
```

字符串转整型的重载版本

stralgo::str2double

类似`stralgo::str2int`。