数据库大作业第二阶段说明文档

类的封装与接口

基础库

- utils.h中包含一些基本工具
  - `string tolower(string)`，将传入字符串中的大写字母转成小写并返回
  - `string toupper(string)`，与`tolower`类似，不过是小写字母转大写
  - `string toregex(string)`，将字符串进行格式转换便于处理like用法
  - `void string trim(string&)`，去除字符串首尾空格
  - `void split(const string&, vector<string>&, const string&)`，分割字符串并去掉各子串的首尾空格
  - `void string_replace(string&, const string&, const string&)`，字符串替换
  - `char getlast_notblank(string&)`，得到字符串中最后一个不是空格的字符

数据库的类形式

- Value.h封装数据表中的数据单元
  - 将`ptr_v`定义为`shared_ptr<Value>`的缩写
  - `enum OprdType`数据类型的枚举类，`Null`表示`NULL`，`Zero`表示值为0的`INT`（同时也是`false`），`NZero`表示值不为0的`INT`（同时也是`true`），`Double`表示`DOUBLE`，`Char`表示`CHAR`
  - `class Value`，数据的基类，本身代表`Null`，衍生出其他数据类型
    - `Value()`，构造函数
    - `bool operator<(const Value&)`，判断是否小于
    - `bool operator==(const Value&)`，判断是否相等
    - `ptr_v cast_up(ptr_v)`，当自身为临近的细致数据类型时，将某种数据指针转化为自身类型的指针，如`Double`可将`Int`提升为`Double`，`Date`可将`Char`提升为`Date`，便于运算比较
    - `ostream& operator<<(ostream&, const Value&)`，输出数据内容
    - `bool isnull()`，返回是否为`Null`类型
    - `OprdType oprd_type()`，返回自身数据的类型
    - **静态成员函数（实现了一元、二元运算符和两个日期函数，将运算结果作为数据指针返回）**
    - `ptr_v v_not(ptr_v)`
    - ……
    - `ptr_v addtime(ptr_v, ptr_v)`
  - `class DoubleValue`
    - `DoubleValue(double)`，构造函数，下同
    - 继承并实现了基类，下同
    - `double get_value()`，得到值，下同
  - `class IntValue`
    - `IntValue(int)`
    - `shared_ptr<DoubleValue> to_double()`，类型转换，下同
    - `int get_value()`

    - `class CharValue`
        - `CharValue(char)`
        - `shared_ptr<DateValue> to_date()`
        - `shared_ptr<TimeValue> to_time()`
        - `string get_val()`
    - `class DateValue`
        - `DateValue(string)`，使用格式化的字符串构造，如"1000-01-01"
        - `DateValue(int)`，使用等价整数构造，整数的优势在于便于日期的加减
        - `string get_formated()`，得到格式化的字符串
        - `int get_val()`，得到等价整数
    - `class TimeValue`
        - `TimeValue(string)`，使用格式化的字符串构造，如“00:00:00”
        - `TimeValue(int)`，使用等价整数构造，整数的优势在于便于时间的加减
        - `string get_formated()`，得到格式化的字符串
        - `int get_val()`，得到等价整数
- Record.h
    - `class Record`，封装一行
        - `Record(const vector<shared_ptr<Value> >&)`，接受这行存储的信息的构造函数
        - `shared_ptr<Value> get_field`，通过列下标访问该域存储的信息
        - `void update(int id, shared_ptr<Value> value)`，修改第`id`列的值为`value`
        - `int size()`，返回行长度
        - `void addValue(shared_ptr<Value>)`，在行尾部插入数据
        - `bool operator==(const Record&)`，比较与另一行记录是否全同
    - `class RecordComparator`，是一个仿函数，用于在sort时作为`Record&`的比较器
        - `RecordComparator(int)`，传入列序号，以某一列作为比较的依据
        - `bool operator() (const Record&, const Record&)`，传入两个`Record&`，按照`<`返回它们之间的大小关系。
- Attribute.h
    - `class Attribute`，封装了表头的属性信息
        - `Attribute()`，默认构造函数，用于其它结构的初始化
        - `Attribute(const string&, const string&, int, bool)`，接受该属性的名称、类型、对应列下标、是否强制非空的构造函数
        - `get_name()`，`get_type()`，`get_id()`，`is_not_null()`分别返回该属性的名称、类型、对应列下标、是否强制非空
        - `shared_ptr<Value> create_field(const string&)`，接受一个值的字符串形式，返回`Value`存储的该值的指针
        - `bool operator<(const Attribute&)`，按名称比较
- SQL.h
    - `class SQL`，封装了对整个数据库的管理
        - `shared_ptr<DataBase> get_database(const string&)`，找到名字为传入的字符串的数据库(区分大小写)并以智能指针的形式返回，如果不存在相应的数据库则返回空指针
        - `shared_ptr<DataBase> get_current_database()`，以智能指针的形式返回当前被use命令指定的数据库
        - `void create_database(const string&)`，创建指定的名字的空数据库
        - `void drop_database(const string&)`，删除指定的名字的数据库
        - `void use(const string&)`，执行`use`指令

        - `void show_databases()`，打印当前所有数据库的名字
- Database.h
    - `class DataBase`，封装了数据库
        - `DataBase(const string&)`，构造函数，接受字符串型形式给出的数据库名字
        - `string get_name()`，返回数据库的名字
        - `shared_ptr<Table> get_table(const string&)`，以智能指针的形式返回数据库中指定的表
        - `void create_table(const string&, const vector<Attribute>&, const string&)`，创建相应的表格
        - `void drop_table(const string&)`，删除指定名字的表
        - `void show_tables()`，列出现有的表
- Table.h
    - `class ValuePtrComparator`，是一个仿函数，用于在 map 和 set 中比较 `shared_ptr<Value>`
        - `bool operator() (shared_ptr<Value>, shared_ptr<Value>)`，传入两个 `shared_ptr<Value>`，按照`<`返回它们之间的大小关系。
    - `class Table`，类封装了数据库中的表
        - `string name`，存储表的名字
        - `vector<Attribute> id2attr`，按照建立表时的顺序、也就是表的列下标存储所有属性
        - `map<string, int> name2id`，建立一个属性的名字到属性的列下标的`map`
        - `int primary_id`，primary key 的列下标
        - `multimap<shared_ptr<Value>, Record, ValuePtrComparator> records`，一个 primary key 的数据到记录的 multimap，这样记录的顺序是按照 primary key 排序的
        - `Table(const string&, const vector<Attribute>&, const string&)`，传入表的名字、表的所有属性和 primary key 的名称
        - `string get_name()`，返回表的名字
        - `void show_columns()`，用于执行`SHOW COLUMNS FROM`指令，打印列信息
        - `void insert_into(const vector<string>&, const vector<string>&)`，用于执行`INSERT INTO`指令，传入所有要插入的属性的字符串和每个属性的值的字符串，插入一条新的记录
        - `void delete_from(const WhereClauses&)`，用于执行`DELETE FROM`指令，传入一个`WhereClauses`，在表中删去所有满足条件的记录
        - `void update(const string&, const string&, const WhereClauses&)`，用于执行`UPDATE`指令，传入需要更新的属性和值的字符串以及一个`WhereClauses`，更新所有满足条件的记录
        - `shared_ptr<vTable> get_vTable()`，将整张表导出为虚拟表

指令接收与处理

- Reader.h
    - `class Reader`，封装了指令的读入和处理
        - `shared_ptr<Instruction> read()`,读入下一条指令并返回指向相应指令的指针，如果读取失败返回空指针
- Parser.h
    - `class Parser`，封装了LL(1)语法分析器
        - `void reset(const string&)`，以字符串的形式传入指令，格式化字符串以重新构造分析器

        - `void resetExpr(const string &)`，以字符串的形式传入表达式，格式化字符串以重新构造分析器
        - `string lookahead()`，返回未匹配的第一个输入符号lookahead
        - `bool lookahead(const string&)`，判断传入的字符串是否等于lookahead
        - `bool ended()`，判断所有输入符号是否都已匹配完毕
        - `void skip(int cnt=1)`，跳过`cnt`个输入符号
        - `void match_token(const string&)`，用传入的字符串匹配lookahead，匹配失败则产生运行时错误
        - `string get_str()`，无条件匹配并返回lookahead，如果输入符号全部匹配完毕，返回"#"
- Instruction.h
    - `class Instruction`，抽象类，其派生类封装了SQL指令(每个派生类对应一条指令)
        - 以字符串形式传入指令，进行语法分析、解析命令参数的构造函数
        - `void exec(SQL&)`，虚函数，在传入的数据库上执行相应的指令
        - **其中只有select的实现稍显复杂，详见设计思路**

select语句及复杂表达式处理

- Operator.h
    - `namespace Operator`，运算符map和一些相关联的辅助方法
        - `map<string, int> priority`，运算符到优先级的映射
        - `map<string, function<shared_ptr<Value>(shared_ptr<Value>)>> optr_unit`，一元运算符到对应一元方法的映射
        - `map<string, function<shared_ptr<Value>(shared_ptr<Value>, shared_ptr<Value>)>> optr_binary`，二元运算符到对应二元方法的映射
        - `bool hasOptr(string)`，判断传入的字符串是否为支持的运算符
        - `bool isBinary(string)`，判断传入的字符串是否为二元运算符
- Func.h
    - `namespace RecordFunc`，单函数map和一些相关联的辅助方法
        - `map<string, function<shared_ptr<Value>(vector<shared_ptr<Value>>)>> name_expr`，单函数名称到其对应方法的映射
        - `map<string, int> param_num`，单函数名称到其参数个数的映射
        - `bool hasFunc(string)`，判断传入的字符串是否为单函数
    - `namespace GroupFunc`，聚合函数map和一些相关联的辅助方法
        - `map<string, function<shared_ptr<Value>(vector<shared_ptr<Value>>)>> name_expr`，聚合函数名称到其对应方法的映射
        - `bool hasFunc(string)`，判断传入的字符串是否为聚合函数
- SyntaxTree.h
    - `class Node`，抽象类，代表表达式处理过程中的结点，通过一系列虚函数多态可精准定位到具体结点类型。派生结构如下
        - `class OpndNode`，代表操作数，使用时传入求值作用的行`Record&`可得到具体值`shared_ptr<Value>`
            - `class ConstNode`，如6、"aa"等常量
            - `class AttrNode`，代表stu_id、stu_name等字段
        - `class OptrNode`，代表操作符，包括逻辑算术运算符、函数、括号
            - `class Optr`，逻辑算术运算符
            - `class FuncNode`，函数

        - `class BracketNode`，括号
- Expr.h
    - `class Expr`，表达式基类，通过传入的表达式字符串与字段名和id的映射进行解析重构，建立`Node`组成的逆波兰表达式堆栈，提前锁定运算结构。有两个派生类
        - `class RecordExpr`，行表达式类，行表达式指作用在行上，不含聚合函数
            - `RecordExpr(const string&, const map<string, int>&)`，调用基类进行构造
            - `shared_ptr<Value> get_value(const Record&)`，传入行求值
        - `class GroupExpr`，组表达式类，组表达式指作用在组上，函数为聚合函数
            - `GroupExpr(const string&, const map<string, int>&)`，调用基类进行构造
            - `shared_ptr<Value> get_value(shared_ptr<Group>&)`，传入组求值
- WhereClauses.h
    - `class WhereClauses`，where子句类，包装了`RecordExpr`(由于where一定作用在行上)，具备检查某一行是否满足条件的功能
        - `WhereClauses(const string&, const map<string, int>&)`，构造函数
        - `bool check(const Record &r)`，检查某行是否符合where的筛选条件
- vTable.h
    - `class RecordMultipleComparator`，是一个仿函数，用于在sort时作为`Record&`的多重比较器，可按照多个关键字依次排序
        - `RecordMultipleComparator(vector<int>&)`，按顺序传入列序号，以某几列作为比较的依据
        - `bool operator() (const Record&, const Record&)`，传入两个`Record&`，按照`<`返回它们之间的大小关系。
    - `class vTable`，处理select语句时为了方便建立的虚拟数据表，作为中转
        - `vTable(const vector<Attribute>&)`，用字段属性数字构建vTable
        - `void insert_into(Record&)`，插入一条记录
        - `Record& getRecord(int id)`，得到第id条记录
        - `int addColumn(string&)`，添加某RecordExpr确定的字段，返回对应id
        - `void addColumn(string&, vector<shared_ptr<Value>>&)`，添加字段和列
        - `vector<Attribute>& export_id2attr()`，得到id到attribute的映射
        - `vector<string> export_id2name()`，得到id到name的映射
        - `map<string, int>& export_name2id()`，得到name到id的映射
        - `void applywhere(const WhereClauses&)`，应用whereclauses去除不符合的记录
        - `void output(vector<string>&, map<string, string>&, ostream&, bool)`，输出表中内容（select的结果）
        - `void outputall(ostream& o, bool)`，输出表中全部内容（select *的结果）
        - `vector<shared_ptr<Group>> getGrouped(string&)`，得到根据某字段分组后的组数组
        - `void order(vector<string>&)`，按照字段名排序
        - `shared_ptr<Group> getWholeAsGroup()`，将整个表化为一个组返回
        - `deleteRepeated()`，去除重复记录
        - **静态成员函数**
        - `shared_ptr<vTable> getEmpty()`，得到一张空表，只有一个字段*下的一条记录Null
        - `shared_ptr<vTable> getCartesian(shared_ptr<vTable>, shared_ptr<vTable>, string, string)`，得到两张虚拟表的笛卡尔积，便于联合处理

        - `shared_ptr<vTable> jointable(shared_ptr<vTable>, shared_ptr<vTable>, string, string, string&,int)`，join功能
        - `shared_ptr<vTable> unionvtable(shared_ptr<vTable>, shared_ptr<vTable>, vector<string>, vector<string>)`，union功能
- Group.h
    - `class Group`，表示组，其内部存储的其实是Record的vector
        - `int size()`，返回组内记录的条数
        - `Record get_record(int i)`，得到第i条记录
        - `void addRecord(Record&)`，添加记录
        - `void setValue(int)`，设置分组依据列的序号
        - `int getValue()`，得到分组依据列的序号
        - `shared_ptr<Value> caculatePoland(vector<shared_ptr<Node>>&)`，计算作用在自身上的逆波兰式的值
        - **静态成员函数**
        - `vector<shared_ptr<Group>> get_group(const vector<Record>&,int)`，将一群记录按某列分组

select语句解析的设计思路及代码执行流程

select语句解析思路

由于select语句附件（子句）较多，对应的处理流程具有较大差异，若对每一种情况、每一个函数，每一个运算符均底层类和统筹类联动特化地写会使代码十分庞杂，且不能具备语句使用的灵活性。要达到代码重用、可拓展性、完善性的兼备，亟需自顶向下，高屋建瓴地总结出一套处理的范式，找出不同流程的不变环节。

select大致会出现几种情况。（暂时不考虑union和join）

1.select XX。无表源，XX代表的表达式独立于字段。

2.select XX from XX where XX order by XX。这是不分组前提下的最完整情况。要注意select后的Expr若为行表达式则无妨，若为组表达式，则应将整个表看做一整个组，且默认不会同时出现组表达式和行表达式（除非对应字段在所有记录中值均相同）。同时注意表源可能有多个，此时所有的字段名前面均加上tablename.

3.select XX from XX group by XX order by XX where XX。有分组和没有分组的情况截然不同，这决定着order by、select后的表达式作用于组上。

本架构的处理思路大致为：由于中间操作有很浓厚的临时意味，且涉及到字段名的更改、字段的添加等，于是不能直接操作Table。先从表源处得到vTable（无表源则使用空表，多个表源则使用笛卡尔积合并），经过where筛选（whereclause此时一定作用在行上，是行表达式）。若无分组，将order by、select后的表达式块均看做虚拟表的字段（只要不重复），用每个行表达式作用在对应行上得到相应字段在相应行上的值，之后经过别名代换输出selected的字段即可；若有分组，where筛选后按照group by后的行表达式求每一行的值，生成新字段，之后按照关键字分组，此时group by后的字段是每个Group的唯一标识，同时select、order by后的均为组表达式，将其作用于每个组，则每个组表达式在每个组中的现实也是只有一个值，此后将这一群每个组的唯一值，以表达式为字段名，一组对应一条记录新生成一个虚拟表，进行后续的排序、输出等操作。

而union和join的处理在虚拟表的预设下变得不费吹灰之力，union即为将多条select语句的结果合并为一个虚拟表再排序，join则可以在笛卡尔积的基础上操作。

代码执行流程

以一个测例作例子分析代码执行流程：

```
CREATE DATABASE OOP;
USE OOP;
CREATE TABLE oop_info(stu_id INT NOT NULL, PRIMARY KEY(stu_id), stu_name CHAR, grade int);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (2018011243, "aaaaa", -1);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (2018011043, "abb", 2);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (2018010243, "ab", 3);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (201801024, "acccaaa", -3);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (2018010243, "a", 3);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (2018001344, "bbbbbb", -2);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (2018001394, "bbb", 1);
INSERT INTO oop_info(stu_id, stu_name, grade) VALUES (2018011445, "c", 4);
SELECT grade*grade,1 - COUNt(*), Avg(grade) as avggrade, min(stu_id) from oop_info GROUP BY grade*grade ORDER BY COUNT(*),-min(stu_id) where char_length(stu_name)+grade = 4;
DROP DATABASE OOP;
```

- 表源为oop_info，得到vTable:

```
stu_id  stu_name    grade
2018011243  aaaaa   -1
2018011043  abb 2
2018010243  ab  3
201801024   acccaaa -3
2018010243  a   3
2018001344  bbbbbb  -2
2018001394  bbb 1
2018011445  c   4
```

- 使用"char_length(stu_name)+grade = 4"和字段名到id映射的map构建WhereClauses对象，使用vTable->applywhere()筛选掉不符合的记录，此时vTable变为

```
stu_id  stu_name    grade
2018011243  aaaaa   -1
201801024   acccaaa -3
2018010243  a   3
2018001344  bbbbbb  -2
2018001394  bbb 1
```

- 有group，先添加group by后面的字段grade*grade

```
stu_id  stu_name    grade   grade*grade
2018011243  aaaaa   -1  1
201801024   acccaaa -3  9
2018010243  a   3   9
2018001344  bbbbbb  -2  4
2018001394  bbb 1   1
```

根据grade*grade分组

grade*grade=1

```
2018011243  aaaaa   -1  1
2018001394  bbb 1   1
```

grade*grade=4

```
2018001344  bbbbbb  -2  4
```

grade*grade=9

```
201801024   acccaaa -3  9
2018010243  a   3   9
```

- select和order by后的表达式有grade*grade、1 - COUNt(*)、Avg(grade)、min(stu_id)、-min(stu_id)、COUNT(*)

  对每个组单独计算，结果如下

  grade*grade=1、1 - COUNt(*)=-1、Avg(grade)=0.0000、min(stu_id)=2018001394、-min(stu_id)=-2018001394、COUNT(*)=2

```
2018011243  aaaaa   -1  1
2018001394  bbb 1   1
```

  grade*grade=4、1 - COUNt(*)=0、Avg(grade)=-2.0000、min(stu_id)=2018001344、-min(stu_id)=-2018001344、COUNT(*)=1

```
2018001344  bbbbbb  -2  4
```

  grade*grade=9、1 - COUNt(*)=-1、Avg(grade)=0.0000、min(stu_id)=201801024、-min(stu_id)=-201801024、COUNT(*)=2

```
201801024   acccaaa -3  9
2018010243  a   3   9
```

- 构建新的vTable如下

```
grade*grade 1 - COUNt(*)    Avg(grade)  min(stu_id) -min(stu_id)
COUNT(*)
1   -1  0.0000  2018001394  -2018001394 2
4   0   -2.0000 2018001344  -2018001344 1
9   -1  0.0000  201801024   -201801024  2
```

- 排序

```
grade*grade 1 - COUNt(*)    Avg(grade)  min(stu_id) -min(stu_id)
COUNT(*)
4   0   -2.0000 2018001344  -2018001344 1
1   -1  0.0000  2018001394  -2018001394 2
9   -1  0.0000  201801024   -201801024  2
```

- 输出，字段名替换为别名

```
grade*grade     1 - COUNt(*)    avggrade    min(stu_id)
4       0       -2.0000 2018001344
1       -1      0.0000  2018001394
9       -1      0.0000  201801024
```