类与方法手册

本文档提供实验中可能用到的类与方法的详解。

存储引擎

StorageNode

StorageNode类储存一个文件的数据，并提供对该文件内容的读写功能。

重要的方法：

1.构造函数

```
/*******************************************************************************
    * @name     : StorageNode
    * @input    : filename:文件名(带有路径)
                  row_length:从元数据管理器中获取的一行数据的字节数
    * @output   : void
    * @brief    : 打开文件，得到对应的文件指针，初始化成员

*******************************************************************************/
    StorageNode(string filename, int row_length);
```

示例：打开`SQLFile文件夹`下的`system_db.dat`文件

```
string filename=string(SQLFILE)+string(SYS_DB);
StorageNode sn(filename, SYS_DB_TUPLE);
/*SQLFILE,SYS_DB,SYS_DB_TUPLE为宏定义，见define.h*/
```

2.rnd_next

```
/*******************************************************************************
    * @name     : rnd_next
    * @input    : void
    * @output   : 指向字节数组第二个元素的指针(第一个元素是标志位)
    * @brief    : 读一行数据并返回指向字节数组的指针。若读到了无效行或读取失败，返回NULL

*******************************************************************************/
    char* rnd_next();
```

示例：打开文件后读取一行数据

```
char* bytearray=sn.rnd_next();
```

3.write_row

```
/*****************************************************************************
    * @name     : write_row
    * @input    : 要写入文件的字节数组
    * @output   : 写入成功返回true，否则返回false
    * @brief    : 将row对应的字节数组写入文件

*****************************************************************************/
    bool write_row(char* row);
```

示例：将字节数组写入文件，并判断写入是否成功

```
    if (!sn.write_row(bytearray)) {
        cout << "write file error" << endl;
    }
```

4.update_row

```
/*****************************************************************************
    * @name     : update_row
    * @input    : 要更新的字节数组
    * @output   : 更新成功返回true，否则返回false
    * @brief    : 该函数用于update表操作，当查询处理层通过迭代器找到要修改的行时，
    *             就通过该函数先将读写指针移动到该行的起始处，然后用覆盖的方法更新
                  数据为row

*****************************************************************************/
    bool update_row(char* row);
```

示例：逐个读出元组，并更新

```
    while (!sn.isEOF()) {
            char* row = sn.rnd_next();
            if (row == NULL)    continue;
            /*更新该行数据*/
            ...
            sn.update_row(newrow);
    }
```

5.delete_row

```
/*****************************************************************************
    * @name     : delete_row
    * @input    : void
    * @output   : 删除成功返回true，否则返回false
    * @brief    : 该函数用于delete元组操作，当查询处理层通过迭代器找到要修改的行时，
    *             就通过该函数先将读写指针移动到该行的标志位处，然后用覆盖的方法将其
                  修改为true，再将指针移动回原位置

*****************************************************************************/
    bool delete_row();
```

使用方法类似于update_row

6.isEOF

```
/*****************************************************************************
    * @name     : isEOF
    * @input    : void
    * @output   : 若已到达文件末尾，返回true，否则返回false
    * @brief    : 判断当前读写指针是否在文件的末尾

*****************************************************************************/
    bool isEOF();
```

示例：逐个读出表中的元组

```
while(!sn.isEOF()){
    char* bytearray=sn.rnd_next();
}
```

StorageEngine

StorageEngine提供多个静态方法来完成对文件系统的操作。

1.createDBDir

```
/*****************************************************************************
    * @name     : createDBDir
    * @input    : 文件夹名(带有路径)
    * @output   : 创建成功返回true，否则返回flase
    * @brief    : 创建文件夹

*****************************************************************************/
    static bool createDBDir(string dirname);
```

示例：为了创建一个数据库db1，我们需要在SQLFile文件夹下新创建一个文件夹db1

```
string dirname=string(SQLFILE)+string("db1");
StorageEngine::createDBDir(dirname);
```

2.deleteDBDir

```
/*****************************************************************************
    * @name    : deleteDBDir
    * @input   : 文件夹名(带有路径)
    * @output  : 删除成功返回true，否则返回flase
    * @brief   : 删除指定的文件夹

*****************************************************************************/
    static bool deleteDBDir(string dirname);
```

3.create

```
/*****************************************************************************
    * @name    : create
    * @input   : 文件名(带有路径)
    * @output  : 创建成功返回true，否则返回flase
    * @brief   : 创建物理文件

*****************************************************************************/
    static bool create(string filename);
```

示例：在db1数据库中创建表tb1时，需要在tb1文件夹下创建一个名为tb1.dat的文件

```
string filename=string(SQLFILE)+"db1/"+"tb1.dat";
sn.create(filename);
```

4.drop

```
/*****************************************************************************
    * @name    : drop
    * @input   : 文件名(带有路径)
    * @output  : 删除成功返回true，否则返回false
    * @brief   : 删除物理文件

*****************************************************************************/
    static bool drop(string filename);
```

元数据管理器

形式如"get..."或"set..."的类方法不详细讲解，读者只需要记住前者用作获取一个成员变量的值，后者用作设置一个成员变量的值。

dbNode

储存、管理一个数据库的库元数据。

1.构造函数

dbNode类有两种构造函数：

```
/*******************************************************************************
    * @name     : dbNode
    * @input    : 数据库名称(不带路径)
    * @output   : void
    * @brief    : 创建对象并初始化

*******************************************************************************/
    dbNode(string name);


/*******************************************************************************
    * @name     : dbNode
    * @input    : 从库元数据表中获取的字节数组
    * @output   : void
    * @brief    : 将从库元数据表中获取的字节数组转换为对应的库元数据对象

*******************************************************************************/
    dbNode(char* bytearray);
```

示例1：创建数据库db1对应的dbNode对象

```
dbNode* node=new dbNode("db1");
```

示例2：假设已从库元数据表中读取到了一个数据库的库元数据，但数据是存储在字节数组中的，那么可以

```
char* bytearray=sn.rnd_next();
dbNode* node=new dbNode(bytearray);
```

2.toByte

```
/*******************************************************************************
    * @name     : toByte
    * @input    : void
    * @output   : 该库元信息对应的字节数组
    * @brief    : 获取库元信息对应的字节数组

*******************************************************************************/
    char* toByte();
```

示例：假设根据用户的输入构造了一个dbNode对象储存库元数据，之后我们需要将库元数据写入物理文件。所以我们需要该方法来将库元数据转换为正确的字节数组。注意，返回的字节数组是使用动态分配的方法创建的，所以在将数组写入物理文件后，可能要释放数组的堆区内存。

```
/*node为指向一个dbNode对象的指针,sn是一个StorageNode对象*/
char* bytearray=node->toByte();
sn.write_row(bytearray);
delete[] bytearray;
```

tableNode

储存、管理一个表的表元数据

1.构造函数

与dbNode相似的，tableNode也有两种构造函数，具体用法也基本一致。

```
/*********************************************************************************
    * @name     : tableNode
    * @input    : 表名，表中列的数量，一行数据的长度(若列为字符串，则要记得在用户指定
                  的长度上再加1)
    * @output   :
    * @brief    : 构造函数

*********************************************************************************/
    tableNode(string tableName, int colNum, int rowLength);



/*********************************************************************************
    * @name     : tableNode
    * @input    : 从物理文件中读取到的一个字节数组，代表一个表的表元数据
    * @output   :
    * @brief    : 根据从文件中获取的字节数组构造一个表元数据对象

*********************************************************************************/
    tableNode(char* bytearray);
```

2.toByte

用法与dbNode的toByte方法一致。

```
/*********************************************************************************
    * @name     : toByte
    * @input    :
    * @output   :
    * @brief    : 将表元数据对象中的数据转换为字节数组

*********************************************************************************/
    char* toByte();
```

colNode

储存、管理一个列的列元数据

1.部分成员变量

```
    string colName;     /*列名*/
    string tableName;   /*所属的表的名称(可能多个表都有同名的列，所以需要区分)*/
    char colType;       /*列的数据类型(单字节存储),2--string,3--int*/
    int colLength;      /*列的长度(若为字符串类型,要比用户指定的长度多1个字节)*/
    int order;          /*该列在元组中的顺序*/
```

2.构造函数

```
/*********************************************************************************
    * @name     : colNode
    * @input    :
    * @output   :
    * @brief    : 无参构造函数

*********************************************************************************/
    colNode();


/*********************************************************************************
    * @name     : colNode
    * @input    : 列名，列所属表的表名，列的数据类型，列的长度，列在元组中的顺序
    * @output   :
    * @brief    : 普通构造函数

*********************************************************************************/
    colNode(string colName,string tableName,char colType,int colLength,int
order);


/*********************************************************************************
    * @name     : colNode
    * @input    : 从物理文件中读出的字节数组
    * @output   :
    * @brief    : 根据从文件中读取的字节数组构造列元数据对象

*********************************************************************************/
    colNode(char* bytearray);
```

3.toByte

写法与用法和dbNode的同名方法完全一致。

4.copy

由于colNode类被colvalue类继承，而后续查询处理层的开发中可能会使用一个已有的colNode类，将其元数据copy到colvalue::colNode中。

```
/*********************************************************************************
    * @name     : copy
    * @input    : 指向一个包含列元数据的colNode对象的指针
    * @output   :
    * @brief    : 将另一个给定的colNode对象的数据成员的值copy到此对象中

*********************************************************************************/
    void copy(colNode* node);
```

示例：将从元数据管理器中获取的列元数据copy到colvalue对象中。

```
/*node指向一个colNode对象，cv指向一个colvalue对象*/
cv->copy(node);
```

MetaDataManager

元数据管理器类，提供元数据的储存、管理、查询服务。

1.部分成员变量

```
    /* 储存所有库元数据,使用map容器，key为数据库名称，value为指向库元数据对象的指针 */
    map<string,dbNode*> sys_db;
    /* 储存所有表元数据，使用map容器，key为pair<数据库名称,用户表名称>
       value为指向表元数据对象的指针 */
    map<pair<string, string>, tableNode*> sys_table;
```

注：储存列元数据的容器在tableNode对象中，读者感兴趣可查看tableNode.h文件。

2.构造函数

```
/*********************************************************************************
    * @name     : MetaDataManager
    * @input    :
    * @output   :
    * @brief    : 构造函数，使用元数据表构造各个容器

*********************************************************************************/
    MetaDataManager();
```

该函数会打开各个元数据表的物理文件，读出所有的元数据并分别构造类中储存元数据的容器。

3.insertDB

为查询处理层的createDB方法提供服务。

```
/*********************************************************************************
    * @name     : insertDB
    * @input    : 指向新的库元数据对象的指针
    * @output   : 写入文件成功返回true，否则返回false
    * @brief    : 将一个数据库的库元数据加入库元数据表和对应的map容器中

*********************************************************************************/
    bool insertDB(dbNode* node);
```

示例：创建一个名为db1的数据库，则先要构造对应的库元数据对象，然后修改元数据。

```
dbNode* node=new dbNode("db1",0);
mdm->insertDB(node);    /*mdm为查询处理层的静态成员变量，是一个元数据管理器的实例*/
```

4.deleteDB

为查询处理层的dropDB方法提供服务。注意，该函数只删除容器中的库元数据和库元数据表中的库元数据，不负责数据库中其余表的删除。

```
/*********************************************************************************
    * @name     : deleteDB
    * @input    : 要被删除的数据库的名称(不带路径)
    * @output   : 删除成功返回true，若数据库不存在或删除失败返回false
    * @brief    : 将指定名称的数据库的记录从库元数据表和map容器中删除

*********************************************************************************/
    bool deleteDB(string dbname);
```

示例：删除名为db1的数据库的库元数据

```
mdm->deleteDB("db1");
```

5.selectDB

查找库元数据。

```
/*********************************************************************************
    * @name     : selectDB
    * @input    : 要查询的数据库名称(不带路径)
    * @output   : 查询值组成的动态数组
    * @brief    : 在库元数据表中查询名称为dbname的数据库，并获得该数据库的信息。
    *             若dbname为空，则返回所有数据库的信息

*********************************************************************************/
    vector<dbNode*> selectDB(string dbname);
```

示例：

```
/*查找数据库db1的库元数据*/
vector<dbNode*> vec=mdm->selectDB("db1");
```

6.createDBTable

在指定数据库的文件夹下创建一个名为`system_table.dat`的文件，作为表元数据表。

```
/*********************************************************************************
    * @name     : createDBTable
    * @input    : 数据库名称(不带路径)
    * @output   : 创建成功返回true，否则返回false
    * @brief    : 在指定的数据库目录下创建表元数据表

*********************************************************************************/
    bool createDBTable(string dbname);
```

以下的函数用法与上述函数基本一致：

```
/*********************************************************************************
    * @name     : insertDBTable
    * @input    : 数据库名称(不带路径)，表元数据节点的指针
    * @output   : 文件写入成功返回true，否则返回false
    * @brief    : 将一个表的表元数据加入元数据管理器的sys_table容器，并将对应的数据写入
    *             表元数据表的物理文件中

*********************************************************************************/
    bool insertDBTable(string dbname, tableNode* node);


/*********************************************************************************
    * @name     : deleteDBTable
    * @input    : 数据库名称(不带路径)，表名称(不带路径)
    * @output   : 删除成功返回true，若删除失败或不存在该表返回false
    * @brief    : 将指定的表的数据从表元数据表中删除，同时对元数据管理器中的sys_table
    *             容器做修改

*********************************************************************************/
    bool deleteDBTable(string dbname,string tablename);


/*********************************************************************************
    * @name     : selectDBTable
    * @input    : 数据库名称(不带路径),表名称(不带路径)
    * @output   : 储存查询结果的动态数组
    * @brief    : 查询给定的表的信息。若tablename为空，返回该数据库中所有表的信息

*********************************************************************************/
    vector<tableNode*> selectDBTable(string dbname,string tablename);


/*********************************************************************************
    * @name     : createTableCol
    * @input    : 数据库名称(不带路径)
    * @output   :
    * @brief    : 在指定数据库的目录下创建列元数据表

*********************************************************************************/
    bool createTableCol(string daname);
```

```
/*****************************************************************************
    * @name     : insertTableCol
    * @input    : 数据库名称(不带路径)，列所属表的名称(不带路径)，指向列元数据对象的指针
    * @output   :
    * @brief    : 将列元数据写入文件，同时更新元数据管理器中对应的容器

*****************************************************************************/
    bool insertTableCol(string dbname,string tablename,colNode* colnode);


/*****************************************************************************
    * @name     : deleteTableCol
    * @input    : 数据库名称(不带路径)，列所属表的名称(不带路径)
    * @output   : 删除成功返回true,若删除失败或不存在这样的数据行返回false
    * @brief    : 删除列元数据表中所有属于所给表的列的列元数据,元数据管理器中容器的修改
    *             放在tableNode的析构函数中做

*****************************************************************************/
    bool deleteTableCol(string dbname,string tablename);


/*****************************************************************************
    * @name     : selectTableCol
    * @input    : 数据库名称(不带路径),列所属表的名称(不带路径),列名
    * @output   :
    * @brief    : 查询所给列名对应的列元数据。若列名为空，返回表中所有列的数据

*****************************************************************************/
    vector<colNode*> selectTableCol(string dbname,string tablename,string
colname);
```

查询处理层与SQL引擎

colinf

该类用于create table sql，负责在SQL引擎中储存新创建的表的一些列元数据。

1.成员变量

```
    std::string colname;    /*列名*/
    int coltype;            /*列的数据类型,2--string,3--digits*/
    int collength;          /*列的长度(以逻辑模式来看)*/
```

示例：当创建一个表时，读者需要在SQL引擎中获取列的列元数据并设置该对象的相应成员变量。如果有一列为字符串类型，那么需要在yacc代码中这样编写：

```
coltype :  CHAR '(' NUMBER ')'  {
          $$=new colinf();
          $$->setColType(2);
          $$->setColLength($3->getDigitsValue());
         }
;
/*同理，若列为INT类型，则需要设置collength成员为4*/

col : colname coltype {
    $$=$2;
    $$->setColName($1->getStringValue());
}
;
```

2.构造函数

```
/*****************************************************************************
    * @name     : colinf
    * @input    :
    * @output   :
    * @brief    : 无参构造函数

*****************************************************************************/
    colinf();
```

colsinf

colsinf类封装了一个std::vector容器，储存多个colsinf对象。用于create table sql，储存表中所有列的列元数据。

1.成员变量:

```
std::vector<colinf*> vec;
```

colvalue

colvalue类储存了一个列的列元数据和实际的值，它公有继承自colNode类。该类为查询处理层的insertTable ， deleteTable , updateTable , selectTable函数以及SQL引擎提供服务。

1.成员变量

strval和digitsval同时只有一个有效。

```
    /*字符串的值*/
    string strval;
    /*整数的值*/
    int digitsval;
    /*该列在所属的用户表的元组中的起始位置*/
    int index;
    /*记录元组中该列的值是否为NULL。若为NULL,则isNULL==true*/
    bool isNULL;
```

2.构造函数

未显式定义构造函数，使用默认构造函数即可。

colsvalue

colsvalue类封装了一个std::vector容器，储存多个表中多个列的colvalue对象，并提供多个处理列值的重要方法。该类为查询处理层的insertTable ， deleteTable , updateTable , selectTable函数以及SQL引擎提供服务。

1.成员变量

```
    /*储存多个列的列值和列元数据*/
    std::vector<colvalue*> vec;

    /*只用于select table,若为true，说明是select* from tbale */
    bool isALL;
```

2.构造函数

构造函数有两个重载

```
/*****************************************************************************
    * @name     : colsvalue
    * @input    :
    * @output   :
    * @brief    : 无参构造函数

*****************************************************************************/
    colsvalue();


/*****************************************************************************
    * @name     : colsvalue
    * @input    : 储存多个列的列元数据的vector容器
    * @output   :
    * @brief    : 通过列元数据tablecol创建对应的colvalue(主要用在创建dataarray)

*****************************************************************************/
    colsvalue(vector<colNode*> tablecol);
```

示例：使用条件查询时，我们需要一个colsvalue对象储存表中所有列的实际值，对象的vec成员即为dataarray数组。假设我们select sql的对象只有表tb1，那么我们需要这样编写：

```
vector<colNode*> v=mdm->selectTableCol(usedatabase,"tb1","");
colsvalue cols(v);
vector<colvalue*>& dataarray=cols.vec;
```

3.addCols

上述构造函数只能根据一个表所有列的列元数据来创建colvalue，而在多表连接查询中，需要根据多个表的列元数据创建colvalue对象并加入vec成员，这时就需要使用addCols方法。

```
/*****************************************************************************
    * @name     : addCols
    * @input    : 储存多个列的列元数据的vector容器
    * @output   :
    * @brief    : 根据列元数据增加对应的colvalue(用于多表连接和创建dataarray)

*****************************************************************************/
    void addCols(vector<colNode*> tablecol);
```

示例：假设要对两个表tb1,tb2进行多表连接，那么需要这样编写：

```
colsvalue cols;
vector<colvalue*>& dataarray=cols.vec;
cols.addCols(mdm->selectTableCol(usedatabase,"tb1",""));
cols.addCols(mdm->selectTableCol(usedatabase,"tb2",""));
```

4.setIndex

```
/*****************************************************************************
    * @name     : setIndex
    * @input    :
    * @output   :
    * @brief    : 根据元数据管理器设置cols中各个列数据的inedx成员,只能处理一个表的列

*****************************************************************************/
    void setIndex(MetaDataManager* mdm,string dbname,string tablename);
```

注意：该方法每次只能处理属于一个表的所有列。

示例：在addCols示例的基础上，设置tb1，tb2所有列对应的colvalue对象的index成员

```
cols.setIndex(mdm,usedatabase,"tb1");
cols.setIndex(mdm,usedatabase,"tb2");
```

5.setCols

从物理文件中读出一行数据后，需要将其转换为每个列的值。

```
/**************************************************************************
    * @name     : setCols
    * @input    : row:指向字节数组首元素的指针
    * @output   :
    * @brief    : 将从表中读取的字节数组转换为每一列的值,每次只能处理一个表

**************************************************************************/
    void setCols(char* row,MetaDataManager* mdm,string dbname,string tablename);
```

示例：考虑一个稍为复杂的问题。假设还是使用多表连接进行查询，读出的字节数组便是两个表笛卡尔积的一个元组，所以分两步来转换：

```
/*假设tb1的元组长度为20个字节,tb2的元组长度为10个字节,row为笛卡尔积的一个元组*/
cols.setCols(row,mdm,usedatabase,"tb1");
cols.setCols(row+20,mdm,usedatabase,"tb2");      /*将指针向后移动20个字节*/
```

6.toByte

toByte函数重写(override)了基类的方法，并且有两个重载：

```
/**************************************************************************
    * @name     : toByte
    * @input    :
    * @output   :
    * @brief    : 获取这些列对应的元组的字节数组(需要先把列元数据补充完整)。用于insertTable

**************************************************************************/
    char* toByte();


/**************************************************************************
    * @name     : toByte
    * @input    :
    * @output   :
    * @brief    : row为旧的行,使用vec中的更新值更新row。用于updateTable

**************************************************************************/
    char* toByte(char* row, MetaDataManager* mdm, string dbname, string
tablename);
```

对于无参的形式，一般用来获取一个元组对应的字节数组，使用时colsvalue对象中必须只能有一个表的列。若对象中只有部分列的colvalue对象，则对于不在colsvalue对象中的列，将NULL值填入元组。

对于有参的形式，一般用于更新元组对应的字节数组。

示例1：对于关系模式 tb1(col1 int , col2 char(10))，若使用这条sql语句：

```
insert into tb1(col2) values('zhg');
```

那么colsvalue对象中只有col2的colvalue对象，此时调用toByte()，得到的字节数组对应于元组(NULL,'zhg')

示例2：若使用这条sql语句：

```
update tb1 set col1=10;
```

那么需要这样编写：

```
...获取cols...
while(!sn.isEOF()){
    char* row = sn.rnd_next();
    cols.setCols(row,mdm,usedatabase,"tb1");
    将cols中col1对应的colvalue的列值设置为10;
    row=cols.toByte(row,mdm,usedatabase,"tb1");
    sn.update_row(row);
}
```

conditiontype

conditiontype类用于储存条件表达式左/右部分的数据类型和值。在表达式的生成树中，每个节点的左右子女节点可能是列名、数字字面量、字符串字面量，也可能是另一个表达式。

成员变量：

```
    /*左/右部分的数据类型,1--[表名.]列名,2--STRING,3--INT,4--条件表达式*/
    int type;
    colvalue* col;      /*[表名.]列名,从SQL引擎返回后还要负责绑定另一个数组*/
    string str;         /*字符串的值*/
    int digits;         /*数字的值*/
    condition* cond;    /*后代指针*/
```

在SQL引擎中，你编写的语义动作要正确的设置conditiontype对象的成员变量。示例：

如果表达式为 col1=10

那么，表达式左部对应的conditiontype对象中，type==1,col为指向一个colvalue对象的指针，且该对象的colname成员为"col1"；表达式右部对应的conditiontype对象中，type==3，digits==10。

condition

condition类表示一个表达式 `left operator right`，可以用作一棵表达式的生成树中的一个节点，并提供计算表达式值的方法。

1.部分成员变量

```
    /*左，右部的数据类型和值*/
    conditiontype left;
    conditiontype right;

    /*true表示该表达式运算符为逻辑运算符，否则为算术运算符*/
    bool iscond;

    /*比较运算符,1:<   2:>  3:<=  4:>=  5:=  6:!=*/
    int comp_op;

    /*逻辑运算符 1--AND  2--OR  3--NOT  */
```

```
    int comp_cond;

    /*该表达式的逻辑值*/
    bool result;
```

示例：在SQL引擎中，需要编写语义动作来正确的设置condition对象的成员。示例：

在conditiontype类的示例的基础上，我们得到了两个conditiontype对象，命名为leftchild和rightchild。然后创建并设置condition对象：

```
left=leftchild;
right=rightchild;
iscond=false;
comp_op=5;
```

2.构造函数

函数定义：

```
condition::condition() {
    left.cond = NULL;
    right.cond = NULL;
    result = false;
    isunknown = false;
}
```

3.addTableName

该函数具有两个重载：

```

/*****************************************************************************
    * @name     : addTableName
    * @input    :
    * @output   :
    * @brief    : 在delete和update中可能从SQL引擎获取的colvalue对象中没有tablename成员
    *             所以手动加上

*****************************************************************************/
    void addTableName(string tablename);


/*****************************************************************************
    * @name     : addTableName
    * @input    : tables储存多表连接中的所有表的名称
    * @output   :
    * @brief    : 用于多表连接,根据元数据管理器,遍历整棵树,补充condition节点中的
                  tablename数据

*****************************************************************************/
    void addTableName(MetaDataManager* mdm,
                      string dbname,
                      vector<colvalue*> tables);
```

注意：将生成树的节点与dataarray数组绑定时，若生成树中某节点的左/右部为列名，那么在查询处理层函数的开始部分**必须把所属表的名称加入左/右部的colvalue对象中。**

示例：以多表连接查询为例。在selectTable中，tables是输入的参数，储存被查询的所有表的名称，root为指向生成树根节点的指针。

```
root->addTableName(mdm,usedatabase,tables);
```

4.dealCondition

```
/*****************************************************************************
    * @name     : dealConditions
    * @input    :
    * @output   : pair的first是子树的result值,second是子树的isunknown值
    * @brief    : 计算以当前对象为根节点的生成树的逻辑值，设置自身的result成员并返回

*****************************************************************************/
    pair<bool,bool> dealConditions();
```

该函数用于计算表达式的逻辑值。其内部实现使用了递归的方法，并且支持三态逻辑，读者会使用即可。

示例：计算以root为根节点的生成树对应的表达式的值：

```
bool logicalvalue=(root->dealConditions()).first;
```

values

values类主要用于SQL引擎的词法分析部分，储存字符串、数字、变量的数据(即终结符的属性)。

1.成员变量

```
    /*终结符的类型,1-id(变量),2-string,3-digits(数字)*/
    int type;
    /*终结符的值*/
    string strval;
    int digitsval;
    /*值在内存中的长度*/
    int size;
```

2.构造函数

```
/*****************************************************************************
    * @name     : values
    * @input    : 终结符类型，储存终结符属性的字符串
    * @output   :
    * @brief    : 构造函数

*****************************************************************************/
    values(int type,char* array);
```

以lex文件中识别字符串的代码为例：

```
{string}    {
    int type=2;
    char* array=new char[strlen(yytext)-1];
    memcpy(array,yytext+1,strlen(yytext)-2);
    array[strlen(yytext)-2]='\0';
    yylval.value=new values(type,array);
    delete[] array;

    return STRING;
}
```

QueryProcessor

QueryProcessor直接向SQL引擎提供处理sql语句的函数。文档中只对部分私有函数进行讲解，其余部分读者可查看QueryProcessor.h文件。

1.部分成员变量

```
    static MetaDataManager* mdm;        /*元数据管理器对象*/
    static string usedatabase;          /*当前正在使用的数据库的名称*/
```

mdm成员在初始化系统的时候已创建，无需读者创建。

2.conditionLinkData

在实验指导.ppt中，我们已经讨论过问什么需要使用dataarray数组并将生成树与dataarray绑定。而conditionLinkData就是用于将数组与生成树进行绑定。

```
/*****************************************************************************
    * @name    :
    * @input   :
    * @output  :
    * @brief   : 将dataarray与表达式生成树的列名绑定起来

*****************************************************************************/
    void conditionLinkData(vector<colvalue*>& dataarray,condition* root);
```

示例：在delete sql中进行绑定数组和生成树操作：

```
conditionLinkData(dataarray,root);
```

3.tableJoin

```
/*********************************************************************************
    * @name     : tableJoin
    * @input    : 临时表中一行的长度
    * @output   :
    * @brief    : 为selectTable()提供服务

*********************************************************************************/
    void tableJoin(vector<colvalue*> tables, int temp_rowlength);
```

多表连接中，我们需要将计算得到的笛卡尔积储存在一个临时表中。该函数即可完成此功能。

示例：假设对tb1和tb2两个表进行连接运算，首先对这两个表进行笛卡尔积。设tb1一个元组长度为20字节，tb2一个元组长度为15字节，tables是selectTable函数的输入参数。

```
tableJoin(tables,20+15);
```

执行完毕后可发现当前数据库的文件夹下增加了一个名为`temp.dat`的临时表文件。

4.dropTempTable

```
/*********************************************************************************
    * @name     : dropTempTable
    * @input    :
    * @output   :
    * @brief    : 删除临时表的物理文件

*********************************************************************************/
    void dropTempTable();
```

多表查询结束后，需要将临时表删除。dropTempTable函数负责删除物理文件`temp.dat`。