一、SQL操作

1. create database name;
2. create table tableName(fieldName fieldType con);
   con包括not null, unique, primary key, foreign key, check, default;
3. insert into tableName fieldName values (fieldValue);
4. update tableName set fieldName = fieldValue where con;
5. delete from tableName where con;
6. truncate table tableName;
7. drop table if exists tableName;
8. tableA inner join tableB on con;
9. alter table tableName rename to anotherName;
10. select name1 [as otherName], fun(name2) from tableA, tableB where condition; //condition还包括 column in {};
11. select name from tableName into column;
12. insert into tableName table;
13. A union B;

二、接口设计

说明

所有private修饰的变量，访问时使用setXxx和getXxx方法；
如果要在外部访问cell内部数据，尽量在相应的类中封装好方法；
做好容错处理，有错误throw。

2.1 TSL设计

```
//数据库
cell struct Database {
 string name;  //库名
 List<CellId> tableList;  //所属的表
}
//表头
cell struct TableHead {
 string tableName;  //表名
 List<int> columnNameList;  //列名
 List<int> columnTypeList;  //列类型
 List<int> primaryIndex;  //主键索引
 List<Element> defaultValue;  //默认值
 List<CellId> rowList;  //行的cellid的集合
}

//每个元素
struct Element {
 string stringField;
 int intField;
 double doubleField;
 DateTime dateField;
 CellId idField;
}

//行
cell struct Row {
 List<Element> values;  //该行属性集合
}


//数据库
struct GetDatabaseMessage {  //查询数据库是否存在
```

```
 string name;  //数据库名字
}
struct GetDatabaseResponse {
 bool exists;  //是否存在
 List<CellId> tableList;  //所包含的table列表
}
protocol GetDatabase {
 Type: Syn;
 Request: GetDatabaseMessage;
 Response: GetDatabaseResponse;
}

//表
struct GetTableMessage {  //查询表是否存在
 string databaseName;
 string tableName;  //表名
 bool getAllValue;  //是否需要获取表的全部内容
}
struct GetTableResponse {  //返回表信息
 bool exists;  //是否存在
 List<int> columnNameList;  //列名
 List<int> columnTypeList;  //列类型
 List<CellId> rowList;  //行的cellid的集合
 List<Element> defaultValue;  //默认值
 List<int> primaryIndex;  //主键索引
 List<List<Element>> values;  //内容
}
protocol GetTable {
 Type: Syn;
 Request: GetTableMessage;
 Response: GetTableResponse;
}


//建表
struct CreateTableMessage {
 string databaseName;
 string tableName;
 List<int> columnNameList;  //列名
 List<int> columnTypeList;  //列类型
 List<Element> defaultValue;  //默认值
 List<int> primaryIndex;  //主键索引
}
protocol CreateTable {
 Type: Syn;
 Request: CreateTableMessage;
 Response: void;
}

//更新表
struct UpdateTableMessage {  //可以用于insert，update等与表内容有关的操作
 string databaseName;
 string tableName;
 List<CellId> rowList;  //行的cellid的集合
}
protocol UpdateTable {
 Type: Syn;
 Request: UpdateTableMessage;
 Response: void;
}

//表的更名
struct RenameTableMessage {
 string databaseName;
 string tableName;
 string newTableName;
}
```

```
struct RenameTableResponse {
 bool success;
}
protocol RenameTable {
 Type: Syn;
 Request: RenameTableMessage;
 Response: RenameTableResponse;
}

//行
struct GetRowMessage {
 CellId cellId;
}
struct GetRowResponse {
 List<Element> value;
}
protocol GetRow {
 Type: Syn;
 Request: GetRowMessage;
 Response: GetRowResponse;
}

server DatabaseServer {
 protocol: GetDatabase;
 protocol: GetTable;
 protocol: CreateTable;
 protocol: UpdateTable;
 protocol: RenameTable;
 protocol: GetRow;
}
```

2.2 基于database的操作

database封装为类，通过实例化创建数据库，并利用该对象进行建表，删表，并且该类对表的操作通过表名进行

```
class FieldType {  //各个类型的宏
 public static int STRING;
 public static int DOUBLE;
 public static int INTEGER;
 public static int LONG;
 public static int DATETIME;
 public static int CELLID;
}
//数据库类
class DataBase {
 private Database database;  //cell

 //建立数据库对象，但并非新建，该数据库必须已经存储在服务器上
 public DataBase(string databaseName);

 public Database createDatabase(string databaseName);

 /**
  建立一个新表
  @param tableName 字符串，表名
  @param fields 可变元组数组，第一个字段为FieldType中的宏，第二个为字段名，第三个为默认值，如果不需要默认值，可以设置为null
  @param primaryKeyList 主键列表
  @return 创建的Table对象
 */
 public Table createTable(string tableName, params Tuple<int, string, object>[] fields, string[] primaryKeyList=null);

 /**
  确认一个表是否存在
  @param tableName 表名
  @return 是否存在
 */
```

```
 public bool exists(string tableName);

 /**
  删除表，释放空间
  @param tableName 表名
 */
 public void drop(string tableName);

 /**
  对两个表做内连接
  @param tableA 可以单表，也可以是多表
  @param tableB
  @con 连接条件
  @return 新表
 */
 public static Table innerJoin(Table tableA, Table tableB, Condition con);

}
```

2.3 基于表的操作

初步考虑，把Table的大部分操作设置为静态操作，

```
//Table对象可能是一个表，也可能是多个表的笛卡儿积，也可能是join的结果
class Table {
 private int tableType;  //表的类型，CLOUD或者LOCAL
 private static int CLOUD;  //CLOUD表示，该表的数据完全存储在服务器上（即使是笛卡尔积或者join），没有进行运算，产生服务器上没有的数据
 private static int LOCAL;  //LOCAL表示，该表的数据完全存储在内存中，通常是由于进行了sum等计算之后，产生了服务器中没有的数据，而这样做
 private string databaseName;  //所属的数据库名，不同数据库的表不能操作

 private List<List<long>> cellIds;  //该项则具体标明了每个表所使用的行，需要按照顺序（包括join之后 各个表的行需要相对应），对应于CLOUD类
 private List<List<object>> values;  //对应于LOCAL类型表
 private List<string> tableNames;  //表名的集合
 private List<List<int>> indexes;  //index用来标识每个表所引用的列索引，如果为null，表明是直接的笛卡尔积，该项可以避免因列更名带来的混乱
 private List<List<string>> columns;  //各个表的列名

 //此构造方法可用于笛卡儿积，例如select tableA, tableB，也可以根据表名直接建立单表对象
 public Table(string databaseName, params string[] tableNames);

 /**
  删除，对应delete语句
  @param table 必须为单表（所有单表必须做检查）
  @param con 条件，默认为空
 */
 public static void delete(Table table, Condition con=null);

 /**
  更新表
  @param table 可以是单表，也可以是多表
  @param fieldNames 要更新的字段，不同表要使用tableName.fieldName的方法
  @param values 要设置的值
  @param con 条件
 */
 public static void update(Table table, string[] fieldNames, object[] values, Condition con=null);


 /**
  用一个表更新另一个表
 */
 public static void update(Table table, string[] fieldNames, Table anotherTable, Condition con=null);


 /**
  插入一行数据，如果fieldNames为空，表明插入所有列
  @param Table 必须单表
```

```
 @param fieldNames 要插入的列名数组，默认为空，即插入所有列
 @param values 插入的值
*/
public static void insert(Table table, string[] fieldNames=null, object[] values);

/**
 一次可以插入多行的insert
*/
public static void insert(Table table, string[] fieldNames=null, object[][] values);

/**
 将一个表的内容插入另一个表，列的类型与数量需要相对应（需要检查）
 @param anotherTable 另一个表，可以不是单表
*/
public static void insert(Table table, string[] fieldNames=null, Table anotherTable);

/**
 对表执行truncate操作
 @param table 要执行的表
*/
public static void truncate(Table table);

/**
 对表进行更名
 @param table 要更名的表
 @param newName 新名字
*/
public static void rename(Table table, string newName);

/**
 对并相容的两个表，做并操作
 @param tableA
 @param tableB
 @param return 合并后的表
public static Table union(Table tableA, Table tableB);

/**
 select操作
 @param fields 要选择的字段
 @param table 要使用的表
 @param con 条件
 @param vars 要into的变量指针数组
 @return 表对象
*/
public static Table select(params Tuple<Condition, int, string>[] fields, Table[] table, Condition con=null, object*[] vars=null)

}
```

2.4 Where条件的表达

由于where条件由表达式以及表达式的与或非构成，并且对每一行都要得到不同的结果，实际上是个非常复杂的模块，因此单独用语法树实现这个模块。

```
//Condition类可以表达非复合的表达式，也可以表示复合的
class Condition {
 private object content;  //当该节点是叶子节点时，所存储的内容，至少提供get方法
 private int contentType;  //content的类型，是下面几种宏的一种
 public static int CONSTANT;  //常量
 public static int FIELD;  //字段

 private Condition leftChild;  //左孩子
 private Condition rightChild;  //右孩子
 private Condition parent;  //父节点，如果实现时没有必要，可以去掉这个属性
 private string op;  //运算符（如果这个节点存在子节点的话），可以是"&","|","!","+","-","*","/","in"
```

```
 public Condition(object content, int contentType);

 /**
  重载&运算符，构建一颗新的语法树
  @param leftCon 左子树
  @param rightCon 右子树
  @return 父节点
 */
 public static Condition operator&(Condition leftCon, Condition rightCon);

 /**
  重载|运算符
 */
 public static Condition operator|(Condition leftCon, Condition rightCon);

 /**
  重载!运算符，仅一个操作数
 */
 public static Condition operator!(Condition con);

 public static Condition operator+(Condition leftCon, Condition rightCon);

 public static Condition operator-(Condition leftCon, Condition rightCon);

 public static Condition operator*(Condition leftCon, Condition rightCon);

 public static Condition operator/(Condition leftCon, Condition rightCon);

 /**
  由于不能重载in，因此写成静态函数
 */
 public static Condition inTable(Condition leftCon, Condition rightCon);

 /**
  根据表的内容，得到该语法树的结果
  @param table 表
  @return 表达式的运算结果(数组)
 */
 public object[] getResult(Table table);
}
```