设计文档

通信模块

使用客户端/服务器架构，客户端提取用户的输入发送给服务器，由服务器进行处理，返回正确结果或者错误原因。rpc.thrift是一个接口定义文件，规定了C/S之间传输数据的格式。在本项目中，需要实现connect、disconnect和executeStatement三个服务的数据交换以及功能执行。

1.connect

1.1接口定义

客户端发送用户名和密码，服务器返回执行结果以及sessionId。

1.2实现

在服务器端通过sessionManager类为每一个客户端生成一个sessionId作为标示符。该sessionId由服务器时间戳+随机数组成。之后的disconnect和executeStatement服务都需要在连接的基础上进行。

2.disconnect

2.1接口定义

客户端发送sessionId，服务器返回执行结果。

2.2实现

服务器端通过sessionManager类把该sessionId从HashMap中移除。

3.executeStatement

3.1接口定义

客户端发送sessionId和要执行的语句statement，服务器返回执行结果、是否出错终止isAbort、是否有查询结果hasResult，如果是查询语句，还会有列名列表columnsList和数据列表rowList。

实现的语句包括：CREATE TABLE 、DROP TABLE 、SHOW TABLE 、SHOW TABLES、INSERT、DELETE 、UPDATE、SELECT、CREATE DATABASE、DROP DATABASE、USE DATABASE、 BEGIN TRANSACTION、COMMIT

实现

服务器接受指令，分解成一条条单一的SQL语句，交由SQLHandler进行处理。如果指令有多条，则只返回成功或者失败的原因。如果只有一条，且为select语句，则hasResult=true，且返回查询的列名以及数据。

在SQLHandler中，通过antlr工具实现词法分析、语法分析以及语义分析。词法分析和语法分析时添加自定义的错误重定向工具ErrorListener，输出错误位置。如果成功完成语法分析，将会生成一棵抽象语法树，交由自定义的SimpleSQLVisitor遍历抽象语法树进行语法分析。为了执行方便，在访问抽象语法树的时候就直接进行SQL语句的执行，而不是像hsqldb一样生成一个Statement类，再进行语句执行。

SimpleSQLVisitor继承自antlr自动生成的SQLBaseVisitor，重载了其中必要的节点访问函数，自顶向下对抽象语法树进行访问。

- CREATE TABLE ：提取出table_name、column_def和table_constraint，检查主键是否存在，最后把表名以及元信息交给Database类的create函数进行处理。
- DROP TABLE：提取出table_name交给Database类的drop函数进行处理。
- SHOW TABLE：提取出table_name，从Database类获取到对应的表，再把元信息返回给客户端。
- SHOW TABLES：获取当前数据库Database对象，再把其中所有表的表名返回给客户端。
- INSERT：提取出table_name、column_name和value_entry，检查column_name是否在table的元信息中，且column_name是否重复。最后把Column列表和值列表交给Table类的insert函数进行处理。
- DELETE ：提取出table_name和where子句的查询条件multiple_condition。如果没有查询条件，则删除所有数据，保留表；如果有，则把Logic传给Table类的delete函数进行处理。
- UPDATE：提取出table_name、column_name和新的值expression以及where子句的查询条件multiple_condition。因为没有实现表达式求值的功能，所有expression假定为数值或者字符串或者null。把where子句构建成Logic，再把column_name、value和Logic交给Table类的update函数进行处理。
- SELECT：提取出select子句、from子句、where子句。在select子句处理时，检查是否有distinct或all的标识，记录所有被查找的列名Colums。在from子句处理的时候，把查询的所有表都加入到ExtendedQueryTable中。在where子句处理的时候，往ExtendedQueryTable中添加查询条件Logic。最后把ExtendedQueryTable、Colums和distinct插入到QueryResult中，调用generateQueryRecord函数生成查询结果，再把结果返回给客户端。
- CREATE DATABASE、DROP DATABASE、USE DATABASE：提取出database_name交给Manager进行处理。
- BEGIN TRANSACTION、COMMIT：见事务模块

Reference

http://www.voidcn.com/article/p-yjkstlnk-bod.html

异常处理模块

实现方式为：将各种异常定义为一个单独的类，均继承RuntimeException，并且在其中设置返回的报错信息。

```
代码在/exception
```

存储模块

- **模块功能**
  - 支持五种列类型：INT, LONG, FLOAT, DOUBLE, STRING
  - Table <-- Row <-- Entry结构分层存储
  - 对表的增、删、查、改
  - 持久化

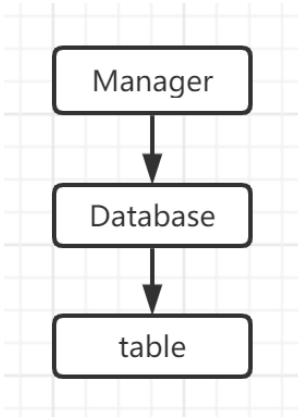
- 页式存储

- **代码结构**
  - 三级存储单元Table，Row，Entry在schema包下
  - 页式存储、持久化在Cache包下
- **设计实现**
  - Entry
    - 有compareTo、equals、toString、hashCode四个成员函数，实现Entry的比较和信息获取
  - Row
    - 有getEntries、getEntry、getDataList，用于获取单行内的Entry数据，或获取String形式的Entry
  - Table
    - match_column_entry(columns, entries)：将输入的Column与Entry数组一一对应。此外如果类型不匹配则尝试转换entry的数据类型，由于这一特性使得在insert表项时可以抵抗一部分错误输入，比如对类型STRING的COLUMN插入1，会转换为'1'再插入
    - insert(columns, entries)：接收输入数据类名和数据列表，插入一行
    - delete(primaryKey)：根据主键删除一行
      - delete(logic)：接收逻辑删除逻辑指定的行
      - deleteAll()：删除所有，只保留元数据
    - update(primaryKey, columns, entries)：根据主键更新一行
      - update(columnName, expression, logic)：根据逻辑将Expression包含的数据更新到指定column
    - query(primaryEntry)：根据主键查找一行
    - persist()：持久化存储
  - Cache
    - 设计思路：所有row存储在page中，B+树保存主键到pageID的索引，HashMap保存当前内存中的所有主键到pageID的索引，所有已有Page，不论是否在缓存中，都在B+树中有记录；
    - 替换算法设计：
      - 选择实现LRU算法，主要有三种实现方式
        - 基于HashMap+TimeStamp，遍历查找时间戳最小的替换
          - 优点：粒度小，自定义空间大
          - 缺点：操作量大，效率低；线程不安全
        - 基于HashMap+LinkedList，找到LinkedList最末尾的元素替换，手动维护顺序
          - 优点：操作少，效率高
          - 缺点：操作页面时需要手动更新LinkedList顺序，逻辑比较复杂；线程不安全
        - 基于LinkedHashMap，自动维护顺序、自动删除页面
          - 优点：策略2的javaSE实现，操作少，维护方便且可靠
          - 缺点：自定义空间小；难以获取过程信息；需要在原有基础上新增逻辑；线程不安全
      - 最终采用策略三实现
    - 存储操作：
      - searchPage(entry)：根据数据得到页号
      - getRow(entry, primaryKey)：根据数据和主键获得一行
      - insertRow(entries, primaryKey)：插入一行
      - deleteRow(entry, primaryKey)：根据entry定位页，primaryKey换页，再根据entry删除一行

        - updateRow(primaryEntry, primaryKey, targetKeys, targetEntries)：根据primaryEntry和primaryKey进行查页/换页，然后更新行内容
        - deleteAll(primaryKey)：删除全部，根据primaryKey换页
    - 页操作：
        - addPage(primaryKey)：根据primaryKey添加一页
        - expelPage(primaryKey)：根据primaryKey将页面移除
        - exchangePage(pageID, primaryKey)：根据primaryKey将旧页面移除，然后根据pageID添加新页
    - 持久化操作：
        - serialize(rows, filename)：根据filename存储rows
        - deserialize(file)：将file的数据读到内存
        - persist()：所有页序列化
        - recoevr(file, primaryKey)：恢复

元数据管理模块

- **元数据结构**



- **Manager**

管理数据库，最顶层，负责数据库的创建、删除、切换，遍历数据库并调用相关方法。
持久化的数据库元数据保存在/data/meta_manager.data里。

- **Database**

管理表，负责表的创建、删除，遍历表并调用 相关方法。
持久化的表元数据保存在/data/meta_database_tablename.data里。

- **Table**

管理每一行数据，负责数据的增删改查持久化等。
持久化使用cache管理。

实现（/schema）

- **Manager类**

```
private HashMap<String, Database> databases;//所有数据库
private static ReentrantReadWriteLock lock；
private Database current_database;//操作的数据库

public void createDatabaseIfNotExists(String dbname);//根据表名建立数据库，并初始设置当前数据库
public void deleteDatabase(String dbname);//删除数据库
```

```
public void switchDatabase(String dbname);//根据current_database的指向来确定当前数据库
public void persist();//数据库信息持久化
private void recover();//从持久化数据恢复
```

- **Database类**

```
private String name;//数据库名
private HashMap<String, Table> tables;//当前数据库下的所有表
public void persist();//持久化，表信息持久化
public void create()；//建表
public void drop(String tbname);//删表
private void recover();
```

- **Table类**

```
private String databaseName;//数据库名
public String tableName;//表名
public ArrayList<Column> columns;//列名
private Cache cache;//存储管理
private int primaryIndex;// 主键索引
private void recover();
// 数据的增删改查
public void insert(ArrayList<Column> columns, ArrayList<Entry> entries);
public void delete(Entry primaryEntry);
public void update(Entry primaryEntry, ArrayList<Column> columns, ArrayList<Entry> entries);
public Row query(Entry primaryEnter);
public void persist();
```

查询模块

查询逻辑解析：

- Logic=Logic op Logic | Condition
- Condition=Expression op Expression
- 没有支持表达式，因此Expression是包括type和value的数据

数据类型：

- Logic：AND，OR
- Condition：=，!=，<，>，<=，>=
- Expression：NUMBER，STRING，COLUMN，NULL

代码结构

- Expression类
  - 存储值和类型
  - 获取值和类型的成员函数
- Condition类
  - 成员变量包含两个Expression和一个比较类型

    - getResult方法返回比较结果
- Logic类
    - 成员变量包括两个Logic，一个连接类型，一个Condition和一个bool值；bool值用于判断Logic形如Logic op Logic还是Condition
    - getResult方法根据成员bool值递归判断逻辑结果
- MetaInfo类
    - 成员变量有一个表名和一个Column链表
- QueryRow类
    - 查询中用到的行类，继承自Row
    - 成员变量是一个Table链表tables和一个entries链表，entries存放所有table的表项；查询结果是将tables所有列按顺序连接后查询到的结果
    - getCompareType方法将比较式中对应列的类型转化为对应的EXPRESSION类型
    - getExpressionFromColumn方法根据列明从表中获取Expression，支持databse.table和table两种名称格式
- QueryTable（抽象类）
    - 成员变量包含一个QueryRow链表queue，一个用于select的Logic，和一个bool值；bool值用于判断该queryTable是否是第一次获取查询到的行
    - hasNext()：判断是否全部查询完成，如果不是第一次且queue为空则查询完成
    - next()：获取下一个查询到的QueryRow，如果queue不为空则返回queue的第一个，并且查询下一个列（ 保证queue不空）；如果不是第一次且queue为空则返回null
    - nextQueryRow()：根据select逻辑查询下一个结果，抽象函数
- ExtendedQueryTable类
    - 继承QueryTable，整合单表和多表查询的逻辑
    - 成员变量除了QueryTable的成员变量，还包含一个Table链表，一个Column链表，一个Iterator<Row>链表，一个用于join的Logic，和一个用于join的Row链表
    - Join的逻辑如下：
        - Iterator<Row>链表保存与Tables链表一一对应的迭代器，每次寻找新Join时，就将最低位的迭代器后移，如果已经到达末尾就回到开头，然后将前一位迭代器后移，前一位迭代器再判断；全部移动完成后就生成一个新Join结果，最高位移动一轮则全部Join完成
    - buildQueryRow()：生成一个Join结果行QueryRow
    - nextQueryRow()：调用buildQueryRow()，根据用于join的Logic判断是否符合条件，符合条件即在queue中加入一条
- QueryResult类
    - 成员变量包括一个QueryTable，一个MetaInfo链表，一个记录选择的列的下标的int链表，一个保存选择的列名的String链表，一个表示是否distinct的bool值isDistinct，一个用于辅助distinct判断的String链表stringResults，一个查询结果Row链表，一个表示结果是否错误的bool值wrong，一个String变量sql，一个String变量message
    - 如果是错误的查询，只会构造wrong和message两个变量
    - 如果是正确的查询，传入查询用表queryTable，选择的列名数组，是否distinct
    - getIndexFromColumn(full_name)：根据列名根据MetaInfo链表找到对应下标
    - generateQueryRecord()：生成所有查询结果。根据queryTable获取选择的行，根据选择的列名数组抽取选择的列构造Row，根据成员变量isDistinct和stringResults判断是否符合distinct条件，符合就加入结果链表。

事务与恢复模块

并发支持

调用thrift提供的函数实现客户端并发。

```
server = new TThreadPoolServer(new
TThreadPoolServer.Args(transport).processor(processor));
```

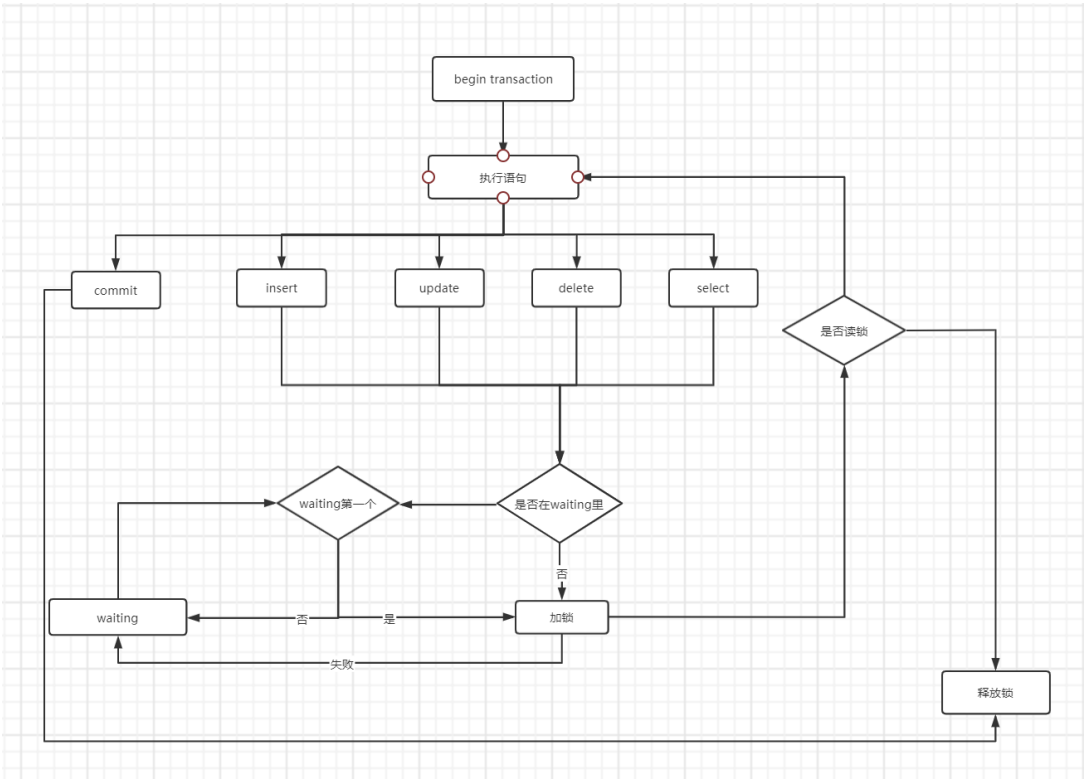
事务

使用二级锁协议，实现read committed隔离级别。
默认情况下，每条语句为一个事务，自动在语句前后插入begin transaction和commit。
实现方式：
利用排他锁（x_lock）和共享锁（s_lock），主要数据结构如下：

```
//Manager.java
public HashMap<Long, ArrayList<String>> s_lock_list;//session和相关读锁的表
public HashMap<Long, ArrayList<String>> x_lock_list;//session和相关读写的表
public ArrayList<Long> waiting_session;//目前处于等待的事务
public ArrayList<Long> transaction_session; //所有处于事务的session

// Table.java
private Long x_lock_session; // 当前表的写锁session
private ArrayList<Long> s_lock_sessions;//当前表相关的读锁
```

实现的流程如下：



主要过程为：begin transaction后，先尝试加锁，如果失败，则加入等待队列，等待一段时间后再次尝试。如果成功，则记录相关的表，并释放读锁。commit时，释放写锁，并清空相关记录。

```
//主要函数：
SimpleSQLVisitor.java
public QueryResult visitCommit_stmt();
public QueryResult visitBegin_transaction_stmt();
```

具体的加锁过程：
在Table内，若未被上锁，直接上锁并记录session；如果当前表已经被session上锁，且已经是所需要的锁，或者等级高于请求的锁(x_lock > s_lock)，则直接返回；其它情况返回标记加锁失败。

```
//主要函数：
//Table.java
public int put_x_lock(Long sessionId)；
public int put_s_lock(Long sessionId)；
```

WAL机制

主要实现方式为：每当执行select、update、delete、insert、begin transaction、commit，就将语句存入log文件。当文件大小超过一定值，然后将数据持久化，清空log文件。
恢复时，先根据meta里的数据库、表信息，以及已经持久化的数据进行恢复，具体逻辑在元数据模块。然后再读取.log文件，并将其中的语句重新执行一遍，对于不闭合的<begin transaction>，直接舍弃这部分语句。最后将log文件重新写回。

```
//主要函数：
//Manager.java
public void read_log(String database_name)；
public void write_log(String statement)；
```