浙江大学

大规模信息系统构建技术导论

分布式 MiniSQL 系统设计报告

2021 学年　春夏学期

组员信息（第一行请写组长信息）

学号	姓名
3190105709	杨特
3180105079	李昊

2022 年　　5 月　　25 日

目　录

1 引言

本次项目是开发一个分布式 Minisql 系统，在《数据库系统原理》课程上开发的 Minisql 的基础上，结合本课程所学分布式一致性系统的原理，来开发项目。通过设计 Master，RegionServer，结合 zookeeper 的配置以及 Client 的开发，实现一个在多主机上共享数据资源访问的分布式数据库系统。

1.1 系统目标

分布式 Minisql 是一个建立在 Minisql 核心功能之上，实现分布式存储与调用的数据库。本系统中，Zookeeer 用于实现服务器与客户端，服务器与服务器通信，总体设计中 Minisql 系统运行于每个服务器内部，由 Zookkeeper 将指令下发到各服务器，而服务器中的 Minisql 则接受指令并返回结果信息。具体功能如下：

(1) 数据库操作：支持 SQL 语句查询，包括数据库表的创建、表的删除、条目的插入、删除、检索以及索引的建立删除等基本功能：

(2) 集群管理

本项目采用 Replicated Mode 进行 Zookeeper 的配置与管理，在本地同时启动三台 Zookeeper 服务端，相互之间采用同一 IP 地址，不同端口的方式进行通信，并利用 zookeeper 内在功能进行 Leader 选举、消息同步等功能的实现。

(3) 副本管理

实现副本管理是因负载均衡以及容错容灾的需要，分布式 MiniSql 中需要实现同一张表分布在不同的主机中。其主要功能如下：

1. 实现容错容灾，同一张表在分布式 MiniSql 中至少在两台主机中存在一致的备份，保证某一台主机失效后依然保持服务的正常运行。

2. 实现负载均衡，对于表数据读可以直接定位到不同的主机上，较好的支持了高并发场景。

(4) 容错容灾

当主机数量众多时，发生错误或故障的概率较大。当单点故障发生时，系统的数据仍然需要保持完整和可用，因而需要对表和记录进行备份。故障发生后，需要检测故障服务器上丢失的表，然后根据记录将表重新备份到其他可用的服务

器上。

(5) 负载均衡

负载均衡是为了提高分布式系统的可用性，在数据表创建时通过节点负载相关信息（表数量、记录数量）选择相应节点，并在数据表读时则随机读取，来提高整体系统的并发度。

1．2 设计说明

本程序采用 Python 程序设计语言，在 Pycharm 平台下编辑、编译与调试。具体程序由 2 人组成的小组开发而成。小组成员的具体分工如表 1 所示：

表 1 各成员分工表

成员姓名	学号	分工
杨特	3190105709	Master, region, zookeeper 集群配置，interpreter, API
李昊	3180105079	Client，index_manager, buffer_manager, record_manager

2　总体设计

2．1　总体架构设计

本分布式 Minisql 主要架构如下：

(1) 该分布式 Minisql 的服务端主要由 Master，Region Server 以及 Zookeeper 集群组成，加上 Client 交互界面，形成一个完整的分布式 Minisql

(2) 服务器的所有操作通过在 Zookeeper 节点中设置监听器的方式来进行

(3) 客户端通过与 Zookeeper 交互来间接获取 Minisql 服务

系统的总体架构如下图所示：

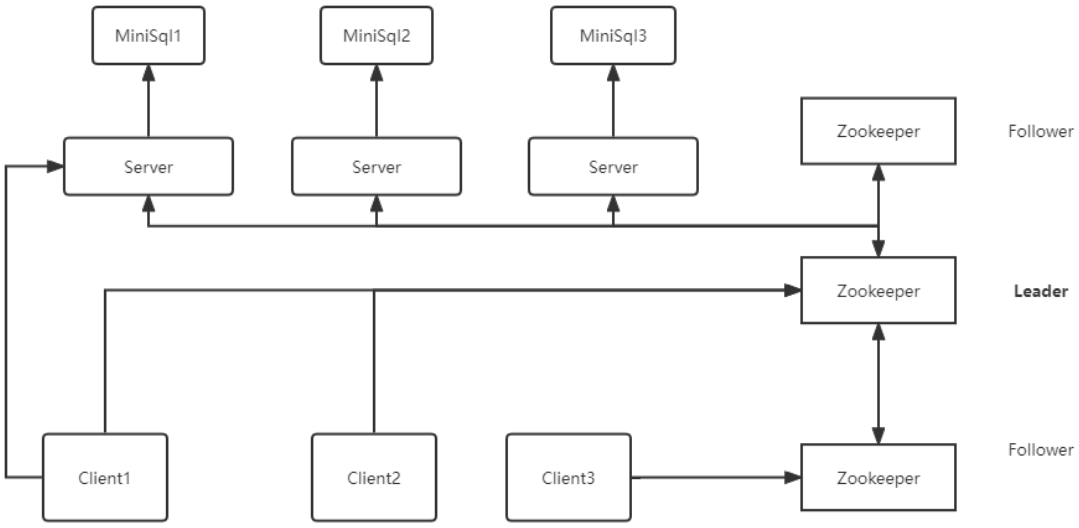


图 1　总体架构架构图

2．2　流程图设计

程序总体流程如图 2 所示：

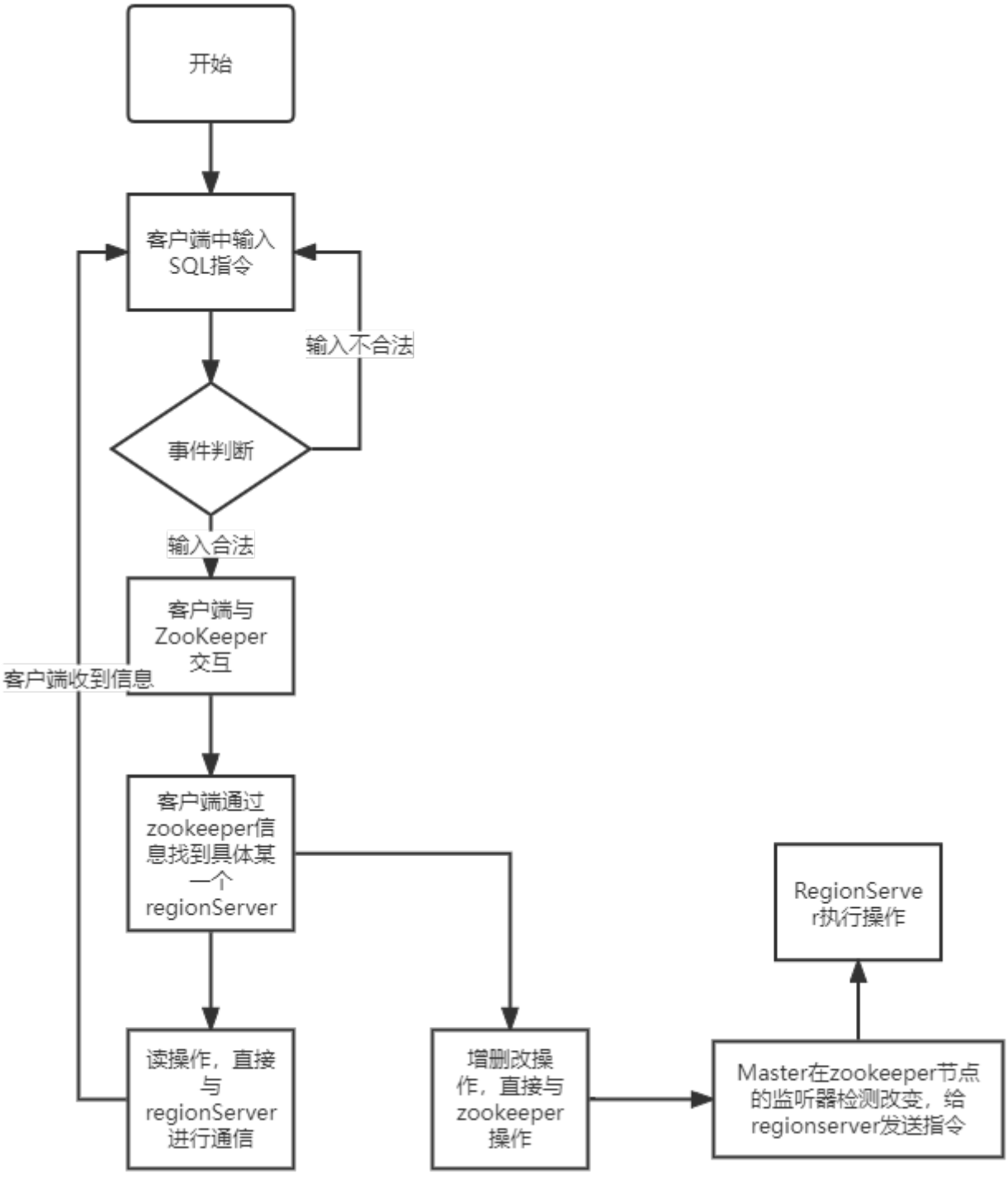
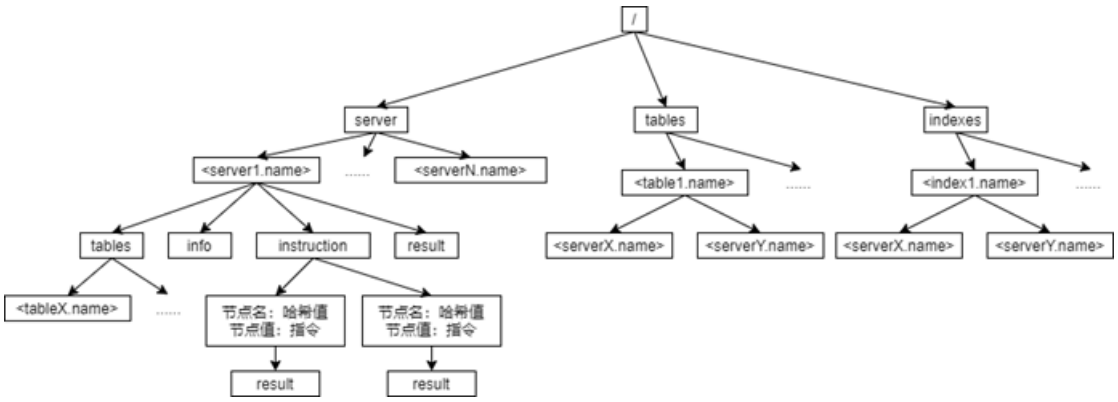


图 2 总体流程图

2. 3 Zookeeper 节点配置图



zookeeper 根节点下共有三层，分别为 Server、tables、indexes。

本架构分布式实现的核心逻辑便是利用 Zookeeper 提供的节点 WATCH 的功能，能够实现某一节点数据发生变动后全局共享。

因此采用节点架构如图所示，对于 table 部分采用持久化节点，对于每一张新创建的表都注册一个 table 节点，并由负载均衡相关模块分配到相应的主机上；对于每一个 Server，创建临时节点，同时存储相关表的信息、节点主机信息等。

Server

Server 节点下对应的是各台 Minisql 的数据，每一台 Server 创建时都会注册对应的集群节点，并且初始化 tables、info、instruction 等节点数据。

其中，table 管理了该台主机下的各表的名称，info 节点记录了记录数量 recordNum 以及表数量 tableNum，instruction 节点下表示处理的指令，每个子结点涵盖了节点哈希与指令，并且下方附带 Result 信息。

每当 Client 将指令传输到 zookeeperinstruction 节点当中，Master 监听器监视到 instruction 节点发生变化，Server 便读取 instruction 中对应的指令并执行。对于创建/删除数据表的请求，Server 需要在对应的 tables 节点下注册/删除对应节点信息，同时修改 info 中对应的 tableNum 表数量，并将最终结果写入 instruction 下的 result 节点。

对于插入/删除语句的操作，Server 需要修改 info 中 recordNum 记录数量，操作后再将对应的结果写入 instruction 下。

对于普通的 select 操作则只需要查询并将结果写入对应 result 即可。

tables/indexes

这两部分相应子节点均为对应表名称/索引名称，而每个对应的表与索引的子节点为存有该表/索引的对应 Server 名称。

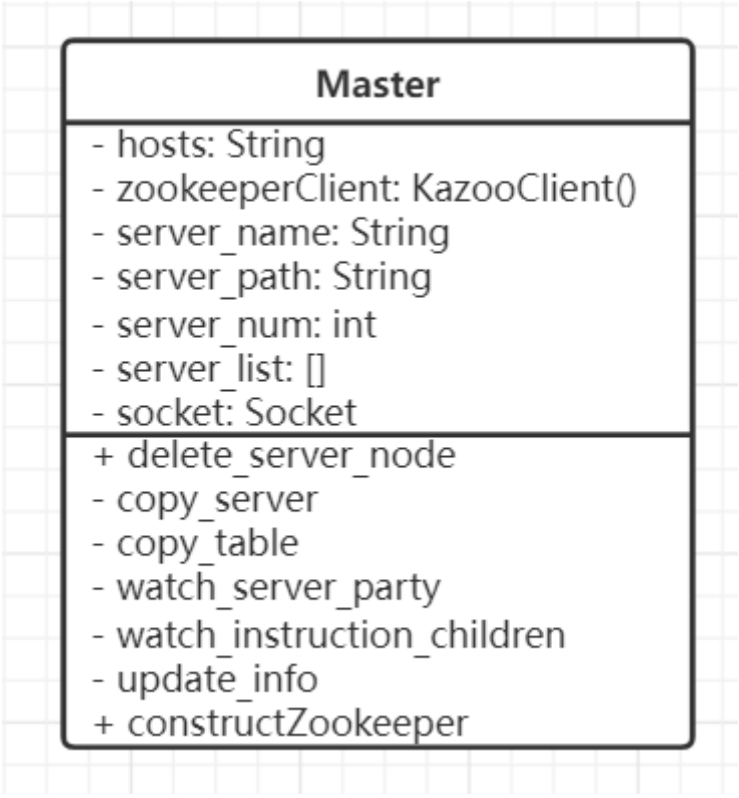
对于表与索引的操作，Client 需要通过 Zookeeper 获取相应的主机名并且向对应 server 发送相关指令，因此这样的 Zookeeper 节点结构能够保证客户端不用遍历主机信息便能够快速获取对应需要操作的 Server 节点

3　详细设计

3.1　Master 设计

（一）Master 类的设计

Master 类实现 Master 相关功能，通过 Socket 与 RegionServer 建立通信，并且通过 zookeeperClient 与 zookeeper 进行交互，以维护 zookeeper 节点信息，从而对 regionServer 进行管理。



以下是对 Master 的详细阐释：

（1）成员变量

① hosts 用来存储 zookeeper 服务端信息，以与 zookeeperServer 建立连接。

② zookeeperClient 为 zookeeper 客户端，通过 zkClient 与 zookeeper 直接交互。

③ server_name 存储 regionServer 的名字

④ server_path 存储 Regionserver 在 zookeeper 中的路径

⑤ server_list 存储已经与 Master 建立连接的 Client

⑥ socket 用来与 RegionServe 建立通讯

（2）方法

① watch_server_party，监视 zookeeper 中 Party 节点的变化，以判断是否有 regionServer 下线。若下线，进行容错容灾的部分。

② watch_instruction_children，监视 zookeeper 每一个服务器 instruction 节点的变化，如果变化，则通过 socket 与 regionServer 进行通信，使 regionServer 执行相关操作。

③ ConstructZooKeeper，在初始化时建立 zookeeper 目录结构。

④ copy_server，服务器断线后，确定将其内容复制到另外哪台服务器。

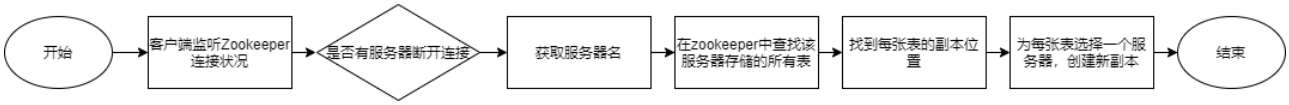
⑤ copy_table，执行复制表格、索引等操作。

⑥ delete_server_node，删除该 server 在 zookeeper 目录中的节点信息。、

⑦ update_info，处理各种指令导致的信息更新。

（二）容错容灾流程

集群的每个服务器持续监听 zookeeper 的连接情况，如果有服务器断开连接，通过对比服务器列表，获取掉线的服务器名，再从 zookeeper 的/severs 目录下找到掉线服务器丢失的表，从/table 下找到这些表的副本位置，最后为每张表选择服务器创建新的副本，并删除掉线服务器在 zookeeper 结构中相关的结点。



（三）故障监听机制

这部分通过 kazoo 的 party 机制实现，具体做法是，当服务器上线运行时，会在/party 下创建一个结点，

来标记自己，当服务器发生故障，终止时，这个结点会被删除。这样，服务器的可用与否就与/party 下

结点的存在一一对应，监听这些结点即可检测到故障的发生，所有服务器都会进

行监听。

```
# 开始心跳
Master.zk.start()
# party部分，用于检测服务器断线情况
Master.zk.ensure_path('/party')
Master.zk.ChildrenWatch('/party', Master.watch_server_party)
party = Master.zk.Party('/party', Master.server_name)
party.join()
```

将 kazoo 中的 children 集合与 sever_list 集合比较，若 sever_list 集合元素更多，则说明有服务器掉线。

（四）故障发生后的恢复工作

当故障发生后，需要：

1. 在/table 下查找掉线服务器中包含的表。
2. 寻找合适的服务器，将丢失的表复制到那个服务器上。
3. 将掉线的服务器的相关结点删除。
4. 更新服务器列表。

从 zookeeper 维持的 children 集合，获取现存的服务器列表

```
if Master.server_num < len(children):
    Master.server_num = len(children)
    Master.server_list.clear()
    for ch in children:
        data, stat = Master.zk.get('/party/{}'.format(ch))
        Master.server_list.append(data.decode('utf-8'))
```

将现存服务器列表与 sever_list 对比，找出掉线的服务器，拷贝丢失的表，然后更新 sever_list

```
for server in Master.server_list:
    if server not in cur_server_list:
        Master.copy_server(server)
        Master.delete_server_node(server)
Master.server_list = cur_server_list[:]
```

从/table 下找到相关表的记录，将表随机备份到可用的服务器上

```
# 确定复制到哪台服务器
@staticmethod
def copy_server(offline_server_name):
    table_list = Master.zk.get_children('/tables')
    remained_server_list = Master.server_list[:]
    remained_server_list.remove(Master.server_name)
    remained_server_list.remove(offline_server_name)
    for table in table_list:
        children = Master.zk.get_children('/tables/{}'.format(table))
        if Master.server_name in children and offline_server_name in children:
            random_index = math.floor(random.random() * len(remained_server_list))
            Master.copy_table(table, remained_server_list[random_index])
```

删除掉线服务器的结点，/severs 和/table 下的记录都要删除

```
@staticmethod
def delete_server_node(offline_server_name):
    Master.zk.delete('/servers/{}'.format(offline_server_name), recursive=True)
    tables = Master.zk.get_children('{}/tables'.format(Master.server_path))
    for table in tables:
        if offline_server_name in Master.zk.get_children('/tables/{}'.format(table)):
            Master.zk.delete('/tables/{}/{}'.format(table, offline_server_name))
```

3．2　Region 设计

Region 类实现 Region 相关功能，通过 Socket 与 Master 建立通信，并且实时响应来自 Client 的读指令。

（1）成员变量

① hosts 用来存储 zookeeper 服务端信息，以与 zookeeperServer 建立连接。

② server_name，本 regionServer 的 name。

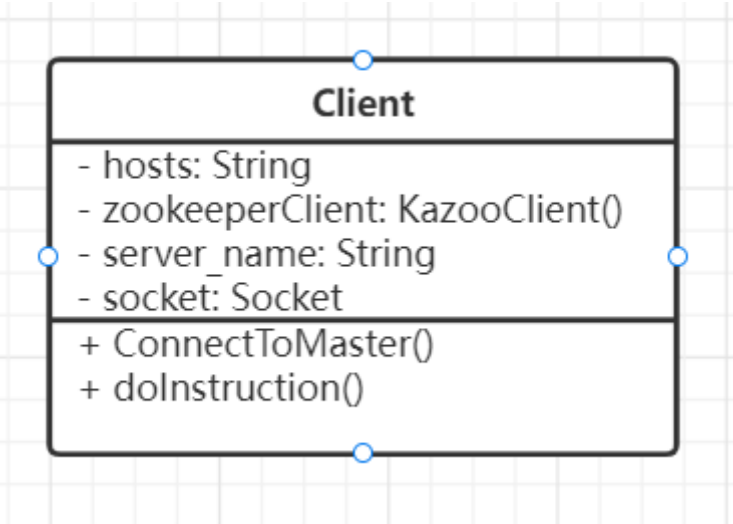
③ socket，用来与 master 建立通信

④ zookeeperClient，与 zookeeper 交互。

（2）方法

① ConnectToMaster：与 Master 通过 socket 建立连接。

② doInstruction： 回应 Client 与 Master 的增删改查相关指令。



3．3　Minisql 设计

（一）Intepreter

1.负责与用户的交互部分，允许用户在命令行中输入SQL语句，MiniSQL会执行语句并返回执行的结果。

2.对输入的语句进行分析，粗略的解析语句并调用对应的API函数进行执行，对返回的结果在命令行进行标准化的打印。

3.捕捉其他模块抛出的异常，在命令行中显示错误信息。

（二）API（Façade）

1.信息提取

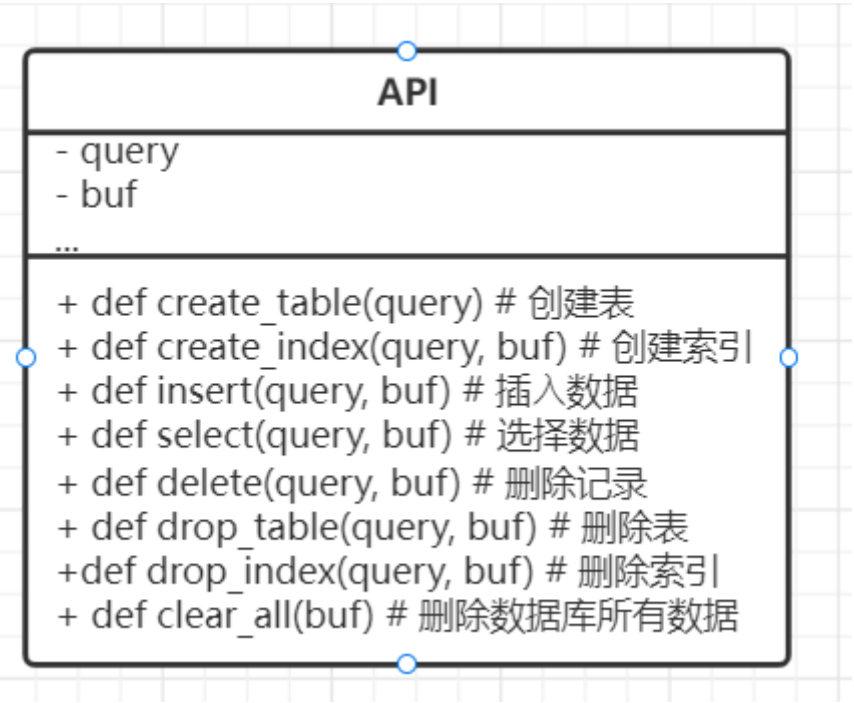
API模块会对SQL语句进行详细的解析，把语句包含的信息进行提取，传入Record或者Catalog模块中。

2.检查语法错误

检查SQL语句是否符合标准语法，如果存在错误则抛出MiniSQLSyntaxError 异常，异常由Interpreter模块负责处理。一般的语法错误不会中断程序，而会在命令行界面上显示出来，提醒用户语句存在语法错误。

3. 清空数据库

为了方便程序进行调试、测试，额外增加了一个clear伪SQL语句，执行clear会清楚数据库内所有数据。



（三）Catalog Manager

Catalog模块负责管理表的schema数据，用一个二进制文件保存起来，这些信息包括：

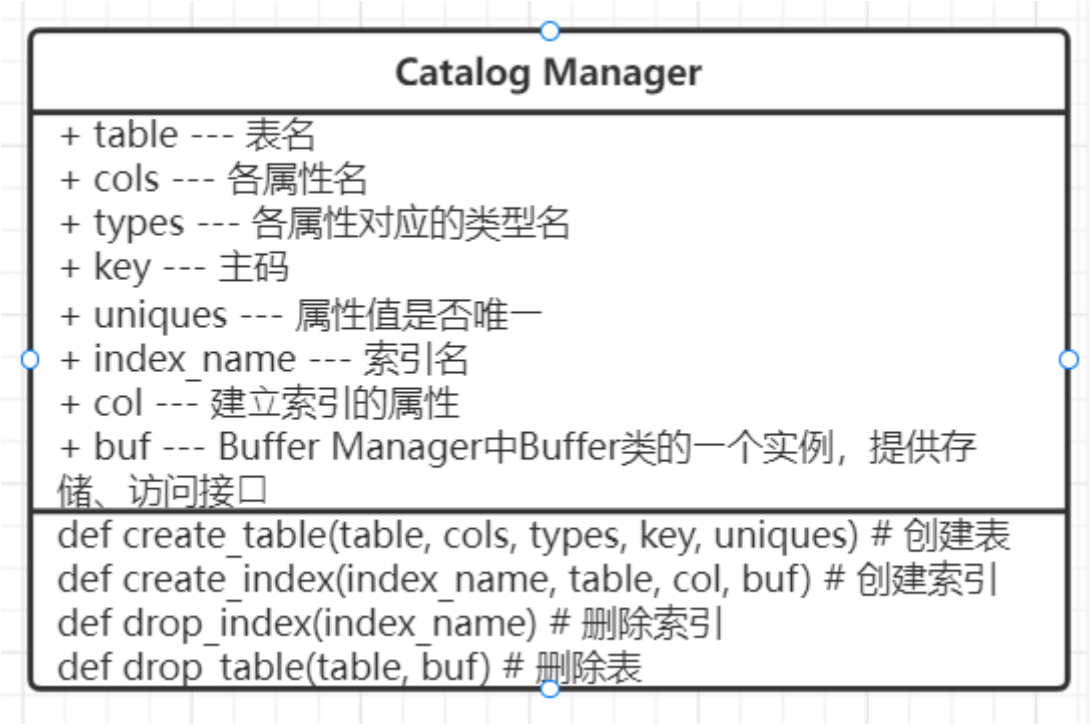
1.表的名称，各属性名称

2.每一个属性对应的类型（int、float、char(n) ），以及是否为unique

3.主码（primary key）

4.索引信息，包括索引的名字、对应哪个属性等等

表的建立、删除以及索引的建立、删除都会调用Catalog模块的接口。



（四）Record Manager 模块

Record Manager 负责管理记录表中数据的数据文件。主要功能为实现数据文件的创建与删除（由表的定义与删除引起）、记录的插入、删除与查找操作，并对外提供相应的接口。其中记录的查找操作要求能够支持不带条件的查找和带一个条件的查找（包括等值查找、不等值查找和区间查找）。

数据文件由一个或多个数据块组成，块大小应与缓冲区块大小相同。一个块中包含一条至多条记录，为简单起见，只要求支持定长记录的存储，且不要求支持记录的跨块存储。

（五）Index Manager

Index Manager 负责 B+树索引的实现，实现 B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

（六）Buffer Manager

Buffer Manager 负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文 件

2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换

3. 记录缓冲区中各页的状态，如是否被修改过等

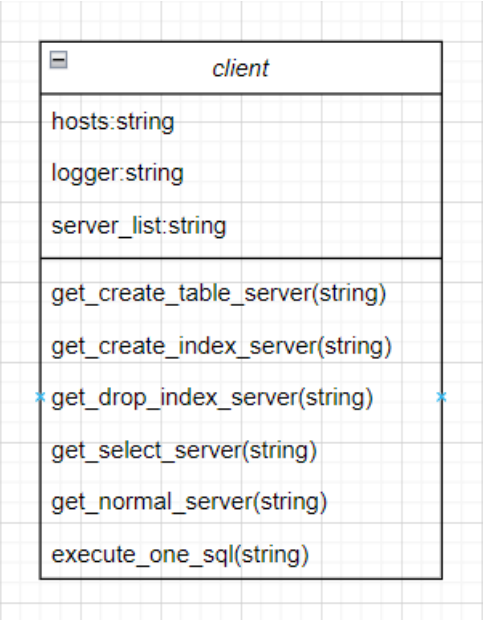
4. 提供缓冲区页的 pin 功能，及锁定缓冲区的页，不允许替换出去

为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数倍，一般可定为 4KB 或 8KB。

3．4 client 设计

客户端主要用于接收用户命令行或者文件输入的 sql 语句，并根据副本和负载均衡的相关配置，得出该 sql 语句对应的 MiniSQL 服务器，然后将该指令同步到 zookeeper 的对应服务器结点上。

客户端的负载均衡实现如下所示：



以下是方法的详细说明：

① get_create_table_server 方法。当用户输入create table 命令时，客户端会读取 zookeeper 中各个服务器 info 结点下的统计信息，选取存储数据较少的两个服务器，在对应 instruction 结点下生成指令结点。Minisql 程序处理完后，会将结果写入指令结点下的 result 结点，并且会在服务器下的 tables 结

点和根目录下的 tables 结点创建该 table 结点，还会在 info 下更新相关统计信息。

② get_create_index_server 方法。create index 指令会在根目录下的 tables 结点，查找储存对应表的服务器，然后将指令同时写入这些服务器的对应结点中。处理完成后同样会写入result 结点，并在 info 结点下更新相关统计信息。

③ get_drop_index_server 方法。由于此指令只提供了 index 的 name，需要从根目录下的 indexes 查找对应的服务器。

④ get_select_server 方法。select 指令会从储存对应表的服务器中随机选取一台写入指令，实现简单的负载平衡。

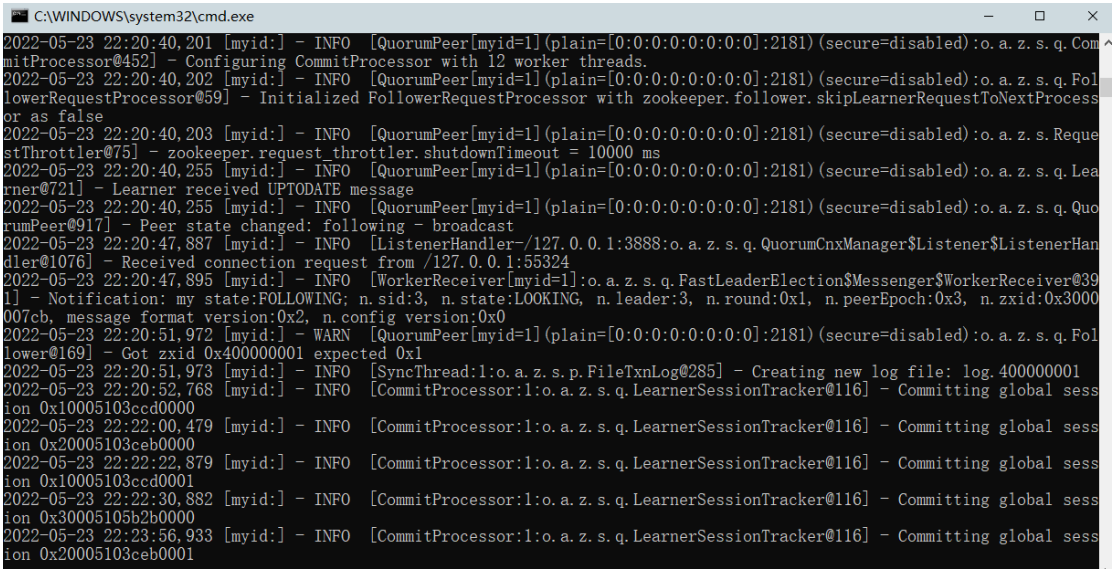
⑤ get_normal_server 方法。找到表所对应的服务器。

⑥ execute_one_sql 方法，读取并处理指令。

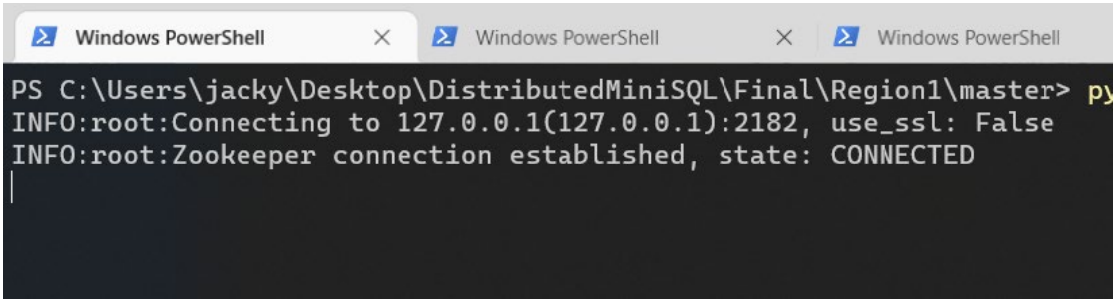
4　测试与运行

我们通过使用 zookeeper 的 ReplicaMode，在本地三个端口同时运行三台 zookeeper 服务端作为 zookeeper 集群，并在另外三个端口运行三台 regionServer 以及一台 Master，进行整体模拟测试。

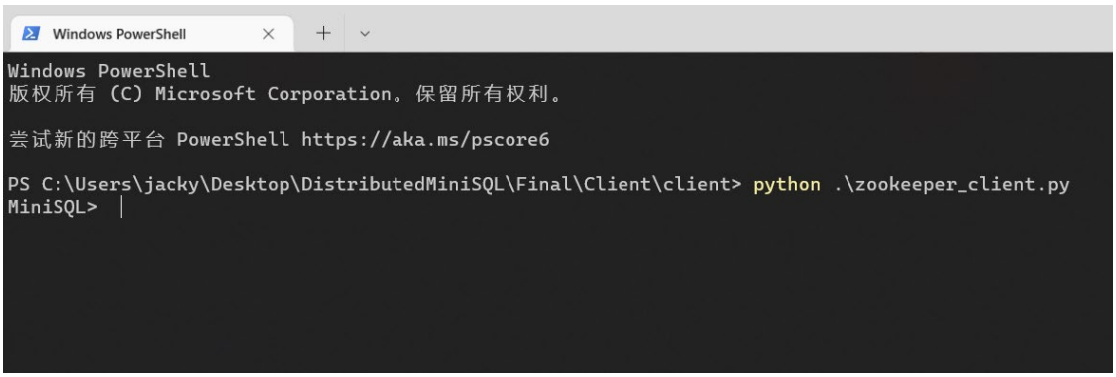
4．1　初始化



启动 Zookeeper 集群

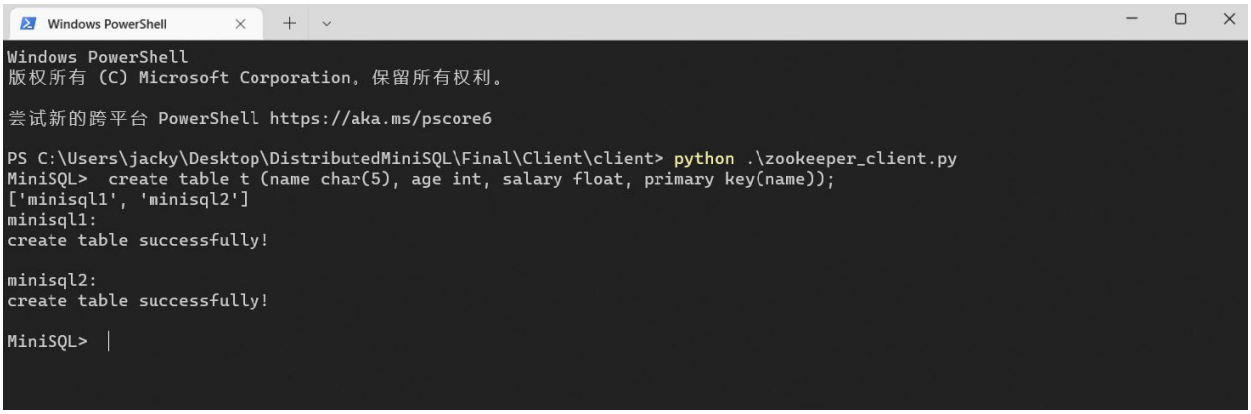


分别启动三台 RegionServer 和 Master。

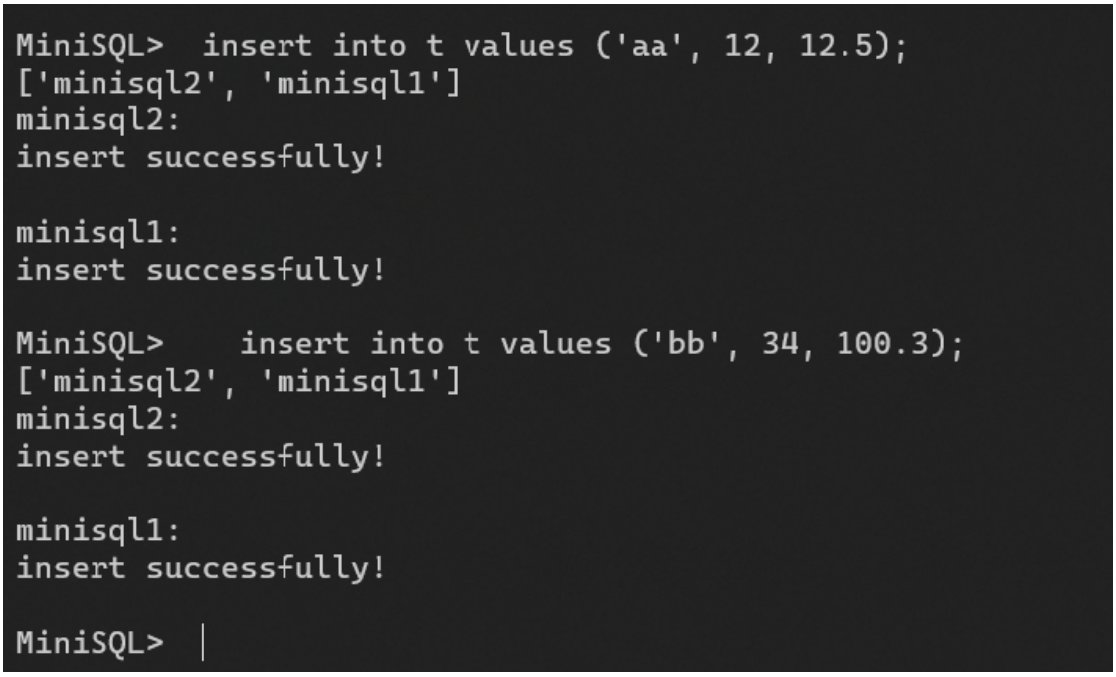


启动 Client 客户端。

4．2　创建表



4．3　插入记录



4．4　查询记录

1. 条件查询

```
MiniSQL>  select * from t where name = 'cc';
['minisql2']
minisql2:
***************
name | age | salary
***************
cc | 56 | 55.0

MiniSQL>
```

2. 全部查询

```
MiniSQL>  select * from t;
['minisql1']
minisql1:
***************
name | age | salary
***************
aa | 12 | 12.5
bb | 34 | 100.3
cc | 56 | 55.0
```

4．5　删除记录

```
MiniSQL>  delete from t where name = 'aa';
['minisql2', 'minisql1']
minisql2:
delete successfully!

minisql1:
delete successfully!

MiniSQL>
```

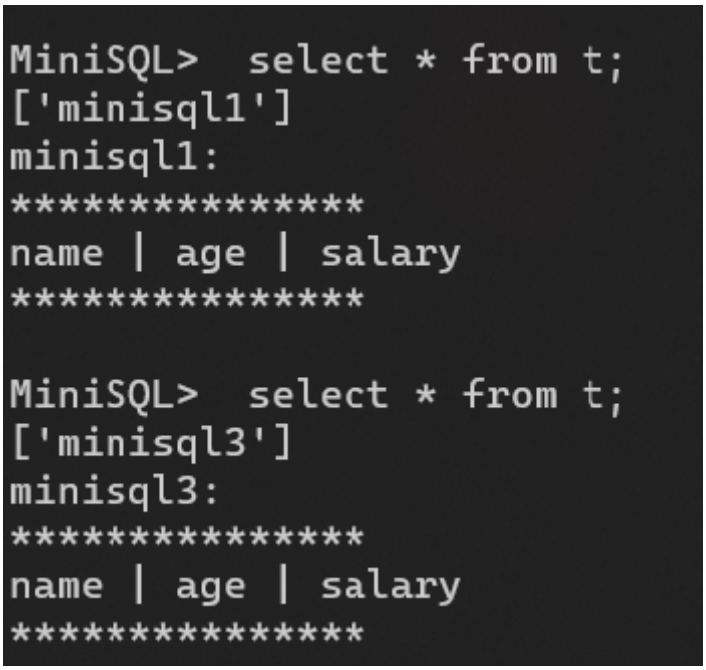
4．6　删除表

```
MiniSQL>  drop table t;
['minisql2', 'minisql1']
minisql2:
drop table successfully!

minisql1:
drop table successfully!

MiniSQL>  select * from t;
Table not exists.
[]
MiniSQL>
```

4．7　负载均衡

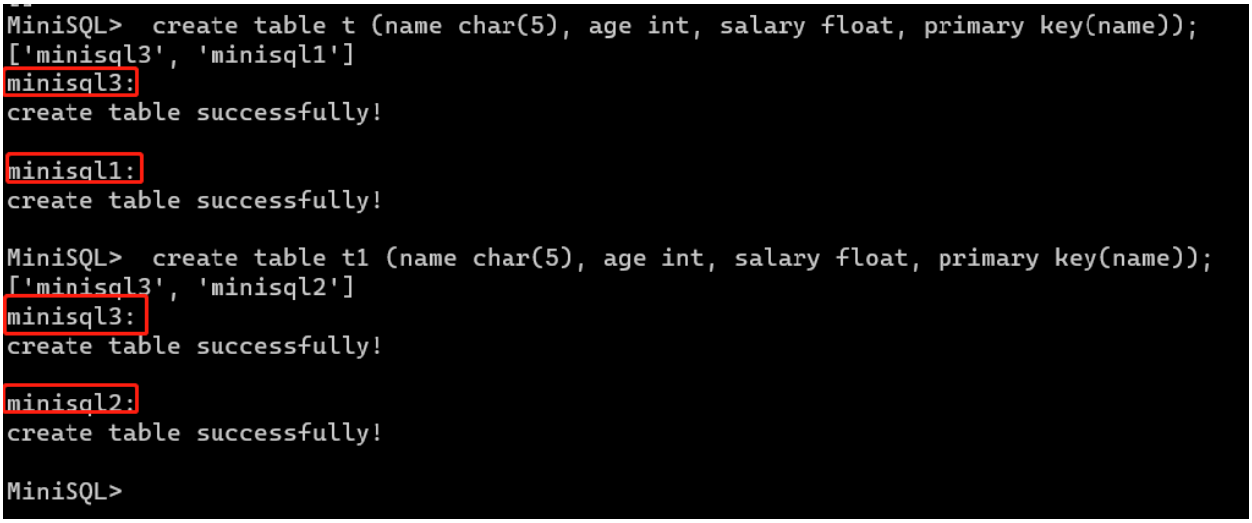
```
MiniSQL>  create table t (name char(5), age int, salary float, primary key(name));
['minisql3', 'minisql1']
minisql3:
create table successfully!

minisql1:
create table successfully!

MiniSQL>  create table t1 (name char(5), age int, salary float, primary key(name));
['minisql3', 'minisql2']
minisql3:
create table successfully!

minisql2:
create table successfully!

MiniSQL>
```

根据表的 recordNum 与 TableNum，动态负载均衡选择服务器创建。



读取数据时也根据相应条件负载均衡，每次从负载最小的服务器中读取。

4．8　副本管理与容错容灾



每次建表、建索引均选择两个服务器进行创建，防止服务器断线。

```
MiniSQL>  create table t1 (name char(5), age int, salary float, primary key(name));
['minisql3', 'minisql2']
minisql3:
create table successfully!

minisql2:
create table successfully!

MiniSQL>  select * from t;
['minisql1']
minisql1:
***************
name | age | salary
***************

MiniSQL>  select * from t;
['minisql3']
minisql3:
***************
name | age | salary
***************

MiniSQL>  select * from t1;
['minisql1']
minisql1:  已经将minisql2中的内容拷贝到minisql1中，实现了容错容灾
***************
name | age | salary
***************

MiniSQL>
```

手动断掉 minisql2，则发现系统自动将 minisql3 中的表格内容复制到 minisql1 中。

5．总结

分布式 miniSql 是在 minisql 的基础上实现分布式管理，包括数据复制，负载均衡，容错容灾等处理，难度偏大。

通过该课程设计，我们深入理解了分布式数据库系统的架构，巩固了数据库 b+树的索引结构以及 CAP 的理念。另外，通过对 zookeeper 的使用，我们也进一步了解了 zookeeper 在分布式架构中的重要地位以及核心原理。

在开发过程中，我们深刻体会到项目管理的重要性，在日后的开发中，我们要在 coding 前进行明确细化的设计，建立明晰的代码结构，做好版本控制，定期开会以跟踪组员进度。