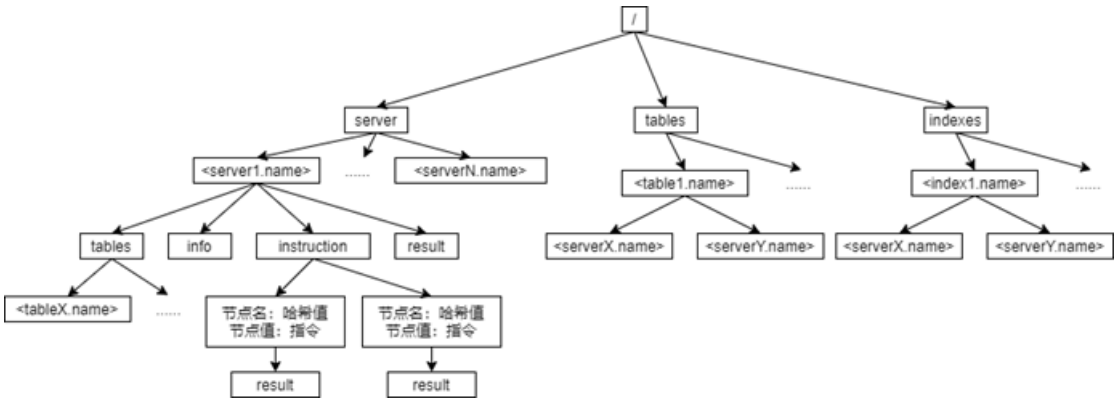
个人设计报告——Master，Region 与 zookeeper 部分

杨特 3190105709

1 Zookeeper 节点配置



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

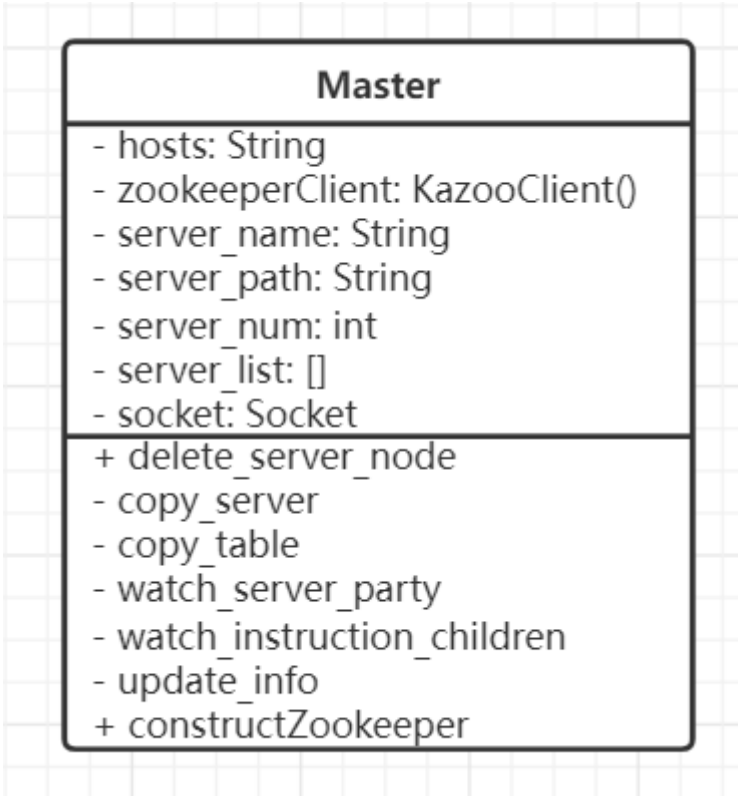
这两部分相应子节点均为对应表名称/索引名称，而每个对应的表与索引的子节点为存有该表/索引的对应 Server 名称。

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。

2 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。

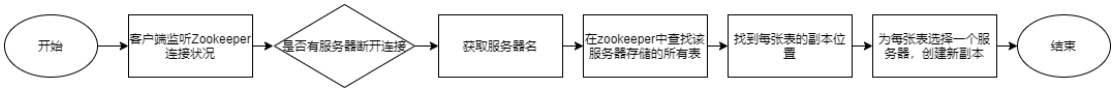
⑤ copy_table，执行复制表格、索引等操作。

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 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。

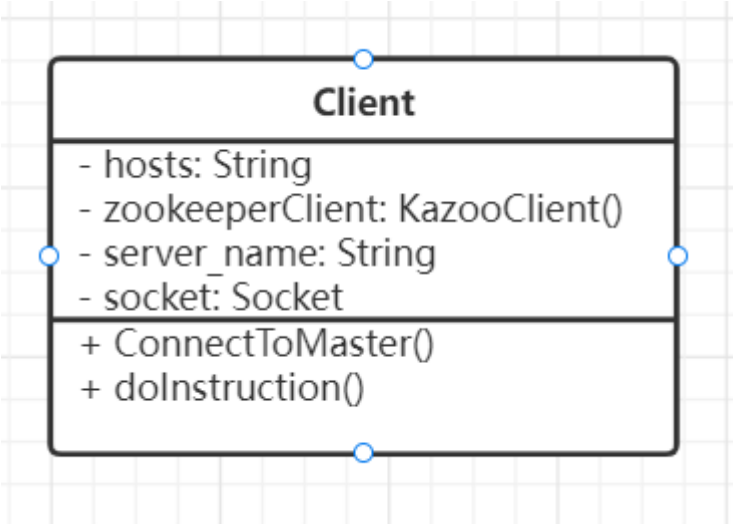
③ socket，用来与 master 建立通信

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。



4 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

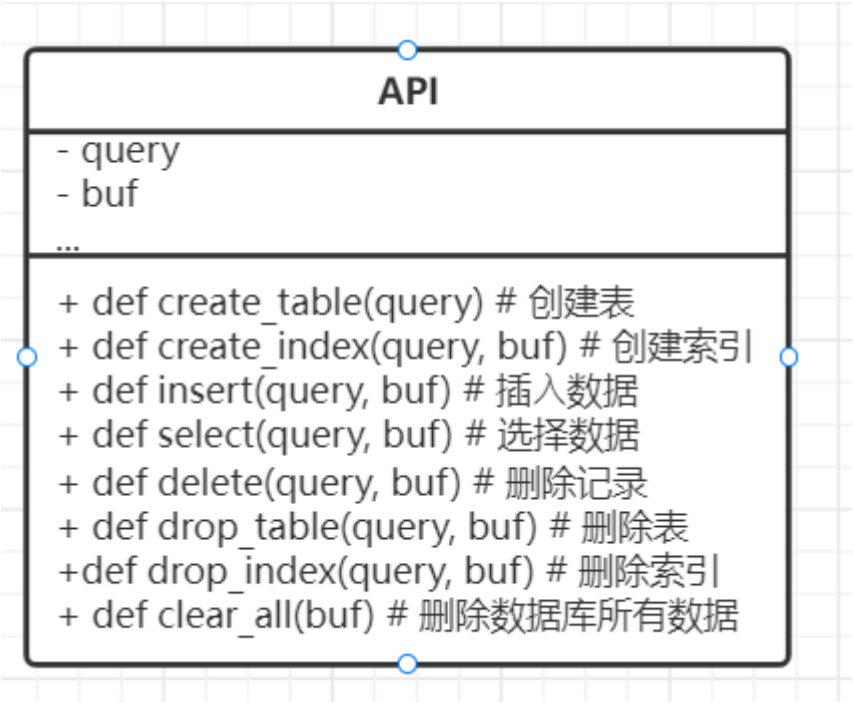
API模块会对SQL语句进行详细的解析，把语句包含的信息进行提取，传入Record或者Catalog模块中。

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：

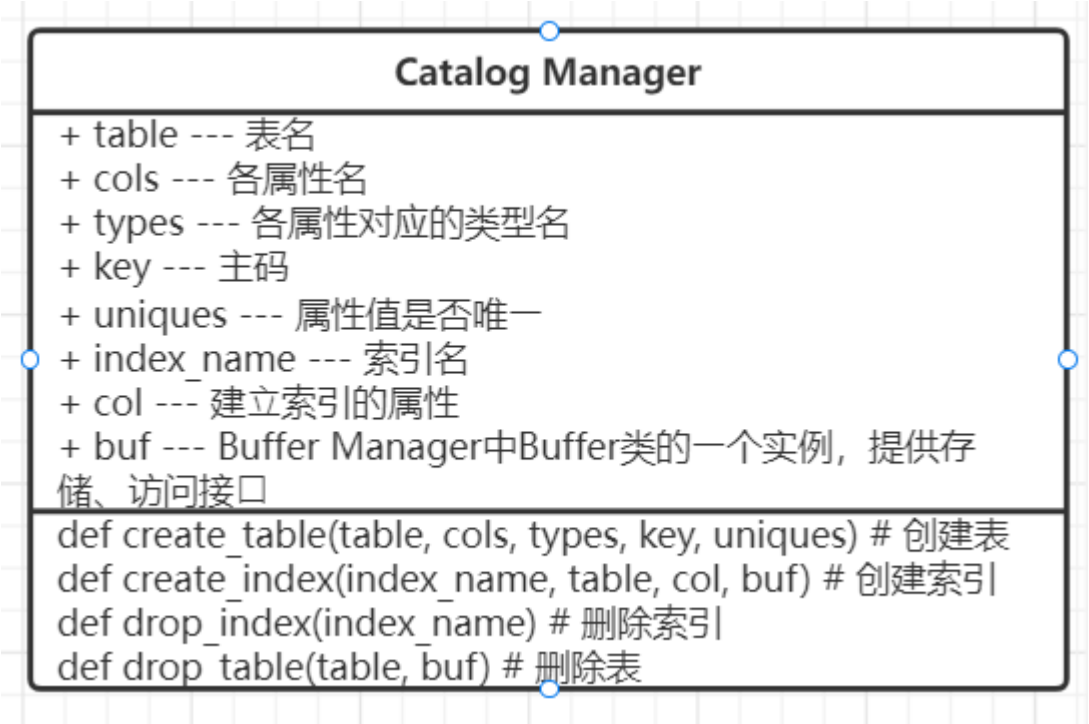
1.表的名称，各属性名称

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。



5 个人小结

我本次负责 zookeeper 集群的搭建，结构的建立以及 Master、region 的撰写。

在设计 zookeeper 时，我一开始并不理解 zookeeper 的原理，不清楚每一个节点对应的用途。后来，结合上课知识以及自学，最终通过建立该 zookeeper 架构，并且结合其监视器来实现分布式管理。同时，由于 minisql 使用 python 编写，我们 zookeeper 也找到了相应的 python Kazoo 包来进行客户端的 API 调用。

在 Master 开发中，我主要负责容错容灾以及 Region 的调度等工作。在容错容灾过程中，由于时间原因，仅实现了掉线监视以及数据备份，并没有实现断线重连的功能，也将在之后进一步加以改进。

本次开发过程中，由于 zookeeper 集群在本地运行，因而和组员无法采取并行开发的模式，导致整体效率较为低下，同时，我也深刻体会到项目管理的重要性，在日后的开发中，应在 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。