

大规模信息系统构建技术导论

分布式MiniSQL系统模块实现与测试报告

|  |  |
| --- | --- |
| 姓名： | 李予谦 |
| 学院： | 计算机科学与技术学院 |
| 系： | 软件工程 |
| 专业： | 软件工程 |
| 学号： | 3200104657 |

2023 年 5 月 14 日

目录

[一．系统模块简介 3](#_Toc135062349)

[二．Client模块实现说明 3](#_Toc135062350)

[2.1 模块组件设计 3](#_Toc135062351)

[2.2 主要数据结构 5](#_Toc135062352)

[2.2.1类图 5](#_Toc135062353)

[2.2.2 Client类 5](#_Toc135062354)

[2.2.3 Buffer类 6](#_Toc135062355)

[2.3 流程图设计 8](#_Toc135062356)

[2.3.1 Client执行SQL流程图 8](#_Toc135062357)

[2.3.2 Buffer更新流程图 9](#_Toc135062358)

[四．测试结果 9](#_Toc135062359)

[4.1 Client基本功能测试 9](#_Toc135062360)

[4.2 Client执行SQL命令 12](#_Toc135062361)

[4.3 Client新建表 13](#_Toc135062362)

[4.4 Buffer自动刷新 14](#_Toc135062363)

[五．开发体会 17](#_Toc135062364)

[参考文献 17](#_Toc135062365)

# 一．系统模块简介

本人在分布式MiniSQL系统中负责研发Client模块，采用Python程序设计语言，在Windows10平台下编辑、编译与调试。与Zookeeper交流使用的库为kazoo。与Reigon交流的库为requests。

Client模块的具体功能如下：

（1）设置Zookeeper数据缓存，时刻与Zookeeper内元数据同步

（2）响应用户指令，发送SQL语句到缓存中对应的Region执行

（3）缓存线程安全、手动刷新缓存、缓存查看功能

（4）Zookeeper数据变更时，通过监视器回调函数自动同步更新

（5）新建表时根据主表数量选择Region建立主表达成负载均衡

# 二．Client模块实现说明

## 2.1 模块组件设计

本模块包括以下几个部分：

（1）Client类

Client类负责：

* 维护与Zookeeper的连接
* 响应用户输入的SQL指令
* 结合Buffer信息选择执行指令的Region

每次启动主程序时将会启动一个Client单例。该Client实例将会连接指定的Zookeeper服务，并开启缓存。该Client单例将会启动一个线程安全锁并传递给Buffer，保证在Client与Buffer中所有对缓存数据的操作都是线程安全的，从而确保SQL信息发送的准确性。

Client将会根据缓存信息和从用户SQL中提取的表信息来确定向哪个Region发送SQL语句。如果Client分析出用户的SQL语句是建表命令，则会从缓存信息中提取所有主表信息，并与Zookeeper沟通，获取当前所含主表最少的Region，并向该Region发送建表信息来达成负载均衡。

Client也采取了多种措施以预防出现未知的问题。例如良好的try-catch、请求发送的超时设置、关键操作时的缓存检查与刷新。

（2）Buffer类

Buffer类负责：

* 缓存Zookeeper中的Region List
* 缓存所有Region到其hosts+port的对应
* 缓存所有主表到其所在Region的对应
* 向Zookeeper注册监听器，动态监听Zookeeper的节点列表
* 向Zookeeper注册监听器，动态监听所有Region的节点内容

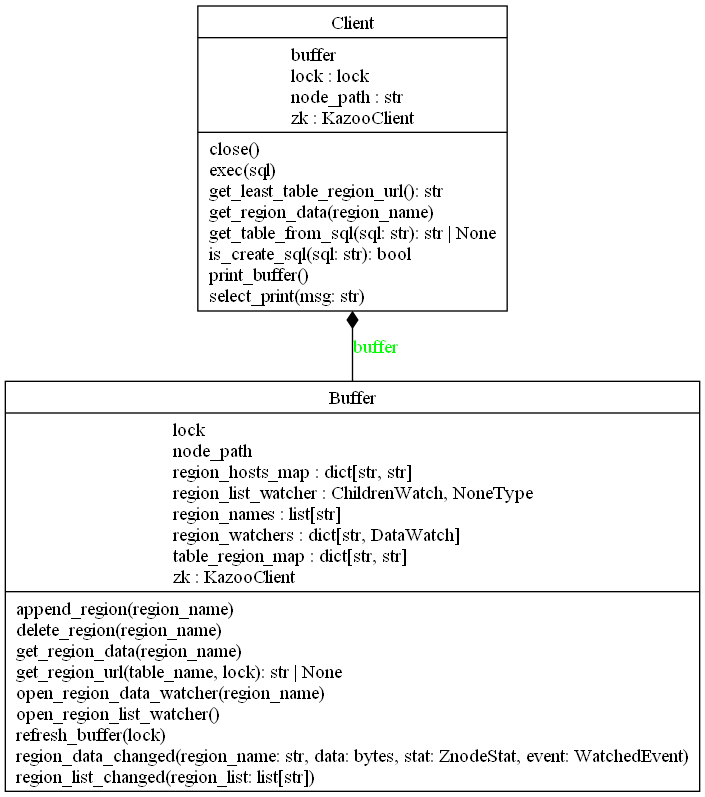
Buffer将会在Client启动时完成自动创建，并接受来自Client的线程安全锁。由于涉及到多线程编程，Buffer中所有对于缓存内容的操作均需要保证线程安全。

在初次启动的时候，Buffer会向Zookeeper请求所需要的数据，并针对整个Region List和各个Region注册监听器。当Zookeeper的Region下线或者上线或者节点数据发生改变时，会触发对应的监听器回调函数，对Buffer进行部分更新。

同时Buffer留出了手动强制刷新的借口，刷新时会清除之前留下的所有数据和监听器，重新进行Buffer的初始化。该功能仅用于发生未知错误的情况，正常情况下Buffer的所有数据将会自动与Zookeeper同步。

## 2.2 主要数据结构

### 2.2.1类图



### 2.2.2 Client类

|  |  |  |
| --- | --- | --- |
| 类成员 | 类型 | 说明 |
| buffer | Buffer | Client对应的Buffer |
| lock | threading.Lock | 线程安全锁 |
| node\_path | Str | Zookeeper中存储数据的节点路径 |
| zk | kazoo.client.KazooClinet | 与Zookeeper沟通的实例 |

|  |  |  |
| --- | --- | --- |
| 函数名 | 返回类型 | 功能说明 |
| close | None | 关闭客户端和Zookeeper的链接 |
| exec | None | 执行制定的SQL语句 |
| get\_least\_table\_region\_url | Str | 获取主表最少的Region的url |
| get\_region\_data | Str | 获取指定region节点的内容 |
| get\_table\_from\_sql | Str|None | 返回从SQL中解析出的第一张表 |
| is\_create\_sql | bool | 判断SQL是否为CREATE语句 |
| print\_buffer | None | 格式化打印目前的Buffer数据 |
| select\_print | None | 格式化打印SELECT语句返回的数据 |

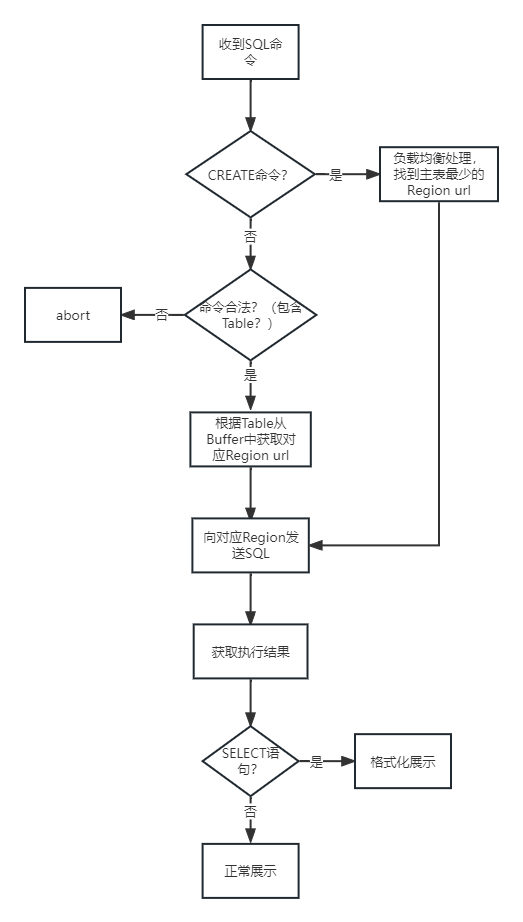
### 2.2.3 Buffer类

|  |  |  |
| --- | --- | --- |
| 类成员 | 类型 | 说明 |
| lock | threading.Lock | 线程安全锁 |
| node\_path | Str | Zookeeper中存储数据的节点路径 |
| zk | KazooClinet | 与Zookeeper沟通的实例 |
| region\_host\_map | dict[str, str] | 储存 region\_name -> hosts+port 的映射 |
| region\_list\_watcher | kazoo.recipe.watchers.ChildrenWatch|NoneType | Region List的children\_list的监听器 |
| region\_names | list[Str] | 储存所有的Region节点名称 |
| region\_watchers | dict[Str, kazoo.recipe.watchers.DataWatch] | 储存 region\_name -> region\_watcher 的映射 |
| table\_region\_map | dict[str,str] | 储存 table\_name -> region\_name 的映射 |
| zk | kazoo.client.KazooClinet | 与Zookeeper沟通的实例 |

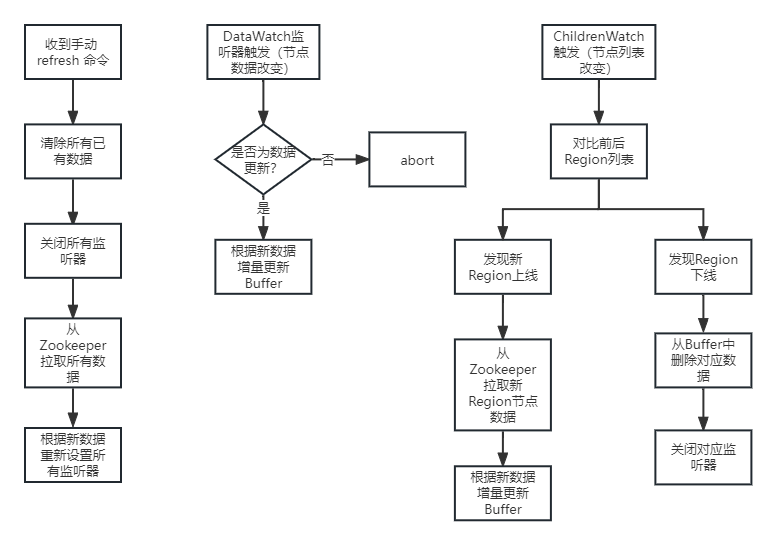
|  |  |  |
| --- | --- | --- |
| 函数名 | 返回类型 | 功能说明 |
| append\_region | None | 在整个缓存中新增一个Region |
| delete\_region | None | 在整个缓存中删除有关该Region的所有信息 |
| get\_region\_data | Str | 获取指定Region节点的内容 |
| get\_region\_url | Str | 获取指定Region的url |
| open\_region\_data\_watcher | None | 开启指定Region的数据监听器 |
| open\_region\_list\_watcher | None | 开启Region List监听器 |
| refresh\_buffer | None | 重置整个buffer |
| region\_data\_changed | None | region\_data\_watcher的回调函数,只处理CHANGE事件,节点被删除等事件由region\_list\_watcher来处理 |
| region\_list\_changed | None | region\_list\_watcher的回调函数,查看是否有节点被删除或新增 |

## **2.3 流程图设计**

### 2.3.1 Client执行SQL流程图



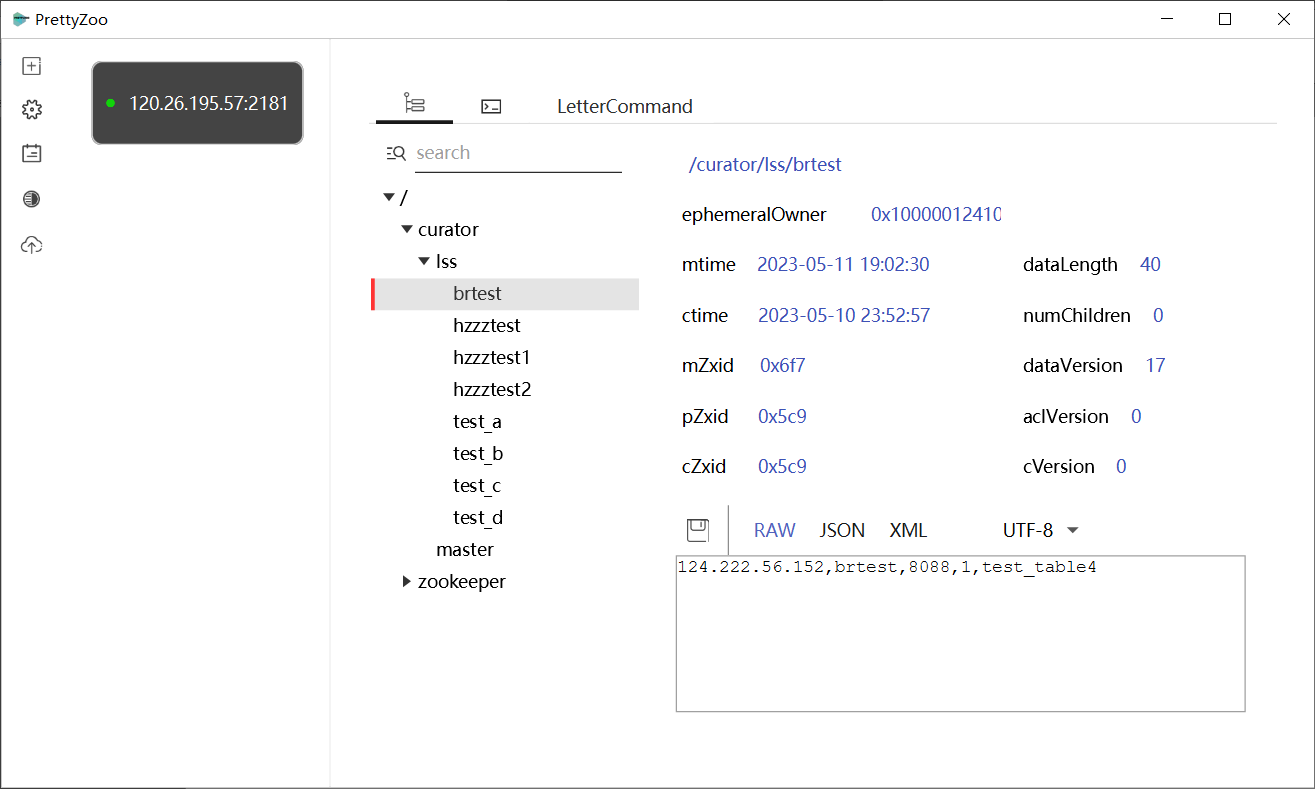
### 2.3.2 Buffer更新流程图



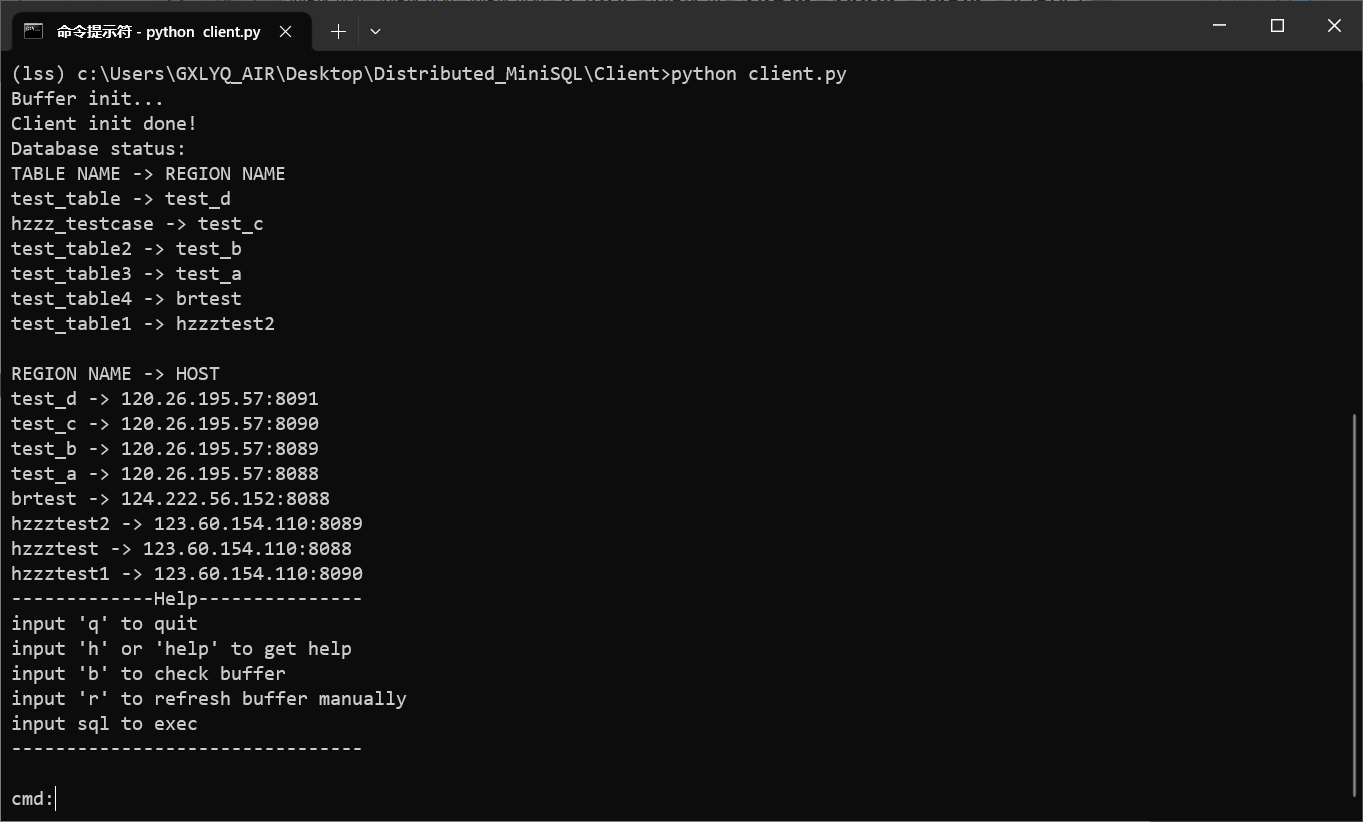
# 四．测试结果

## 4.1 Client基本功能测试

首先确认Zookeeper状态

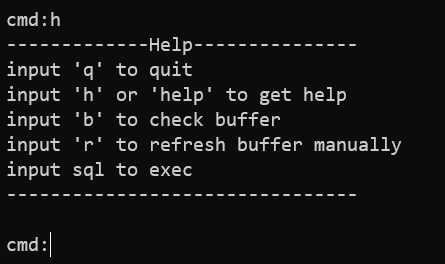


开启Client，自动拉取信息构建Buffer如图



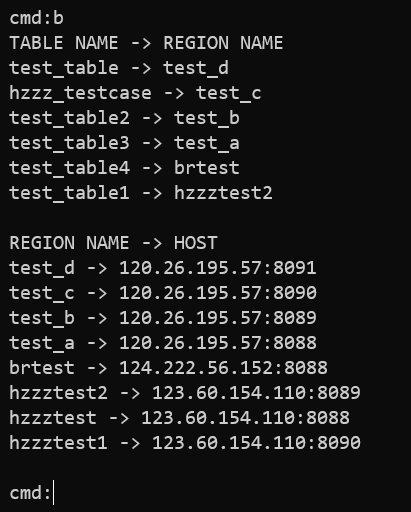
可见Buffer构建成功，Client成功启动。

测试’查看帮助’功能：



成功输出帮助信息。

测试’查看Buffer’功能：



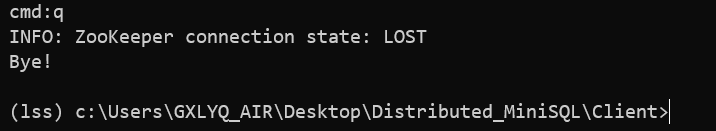
Buffer成功格式化打印。

测试手动强制刷新Buffer功能：



刷新成功，由于Zookeeper并未发生变化，Buffer数据并未改变。在随后的测验中，理论上不应该使用手动刷新功能，因为在正常情况下，Buffer将会由监听器触发自动更新。

测试’退出’功能：

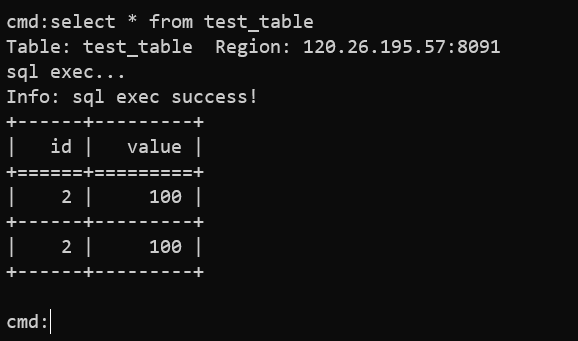


程序正确断开与Zookeeper的链接并退出。

## 4.2 Client执行SQL命令

输入以下SQL命令

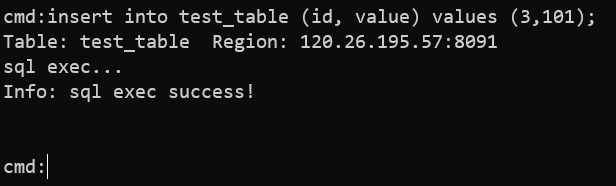
select \* from test\_table;



获取结果成功

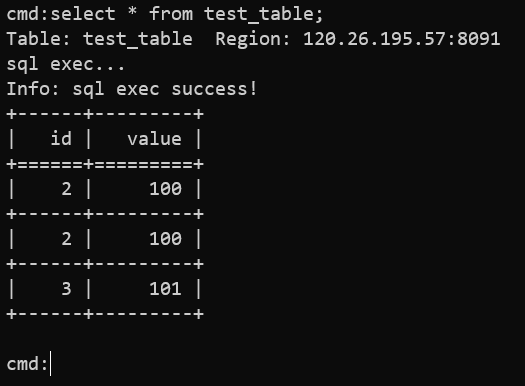
输入以下SQL命令

insert into test\_table (id, value) values (3,101);



提示SQL执行成功。再输入以下SQL命令：

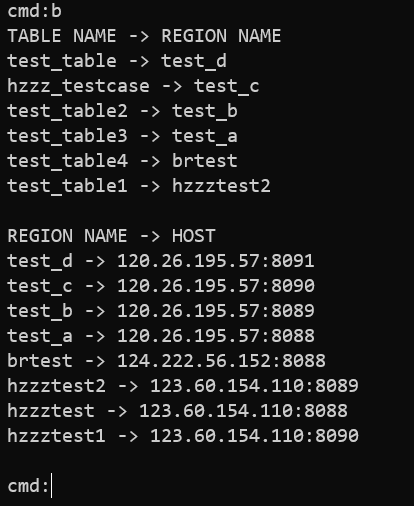
select \* from test\_table;



可见新数据被成功插入。

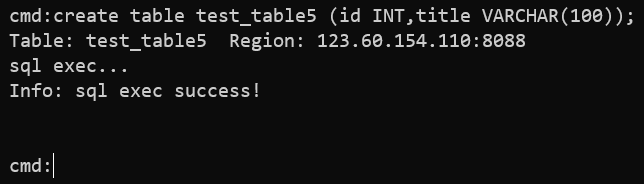
## 4.3 Client新建表

我们首先确认当前的库表和Region情况

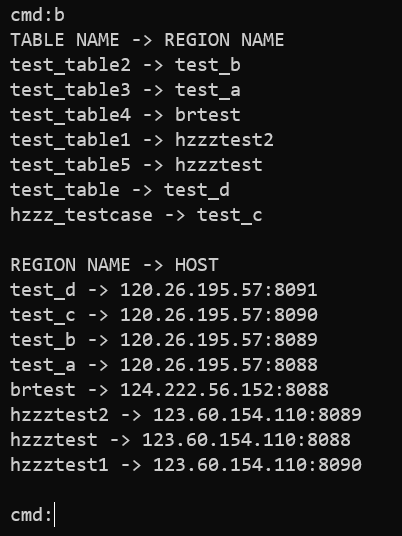


我们执行建表命令：

create table test\_table5 (id INT,title VARCHAR(100));



提示SQL执行成功。我们查看目前的Buffer：



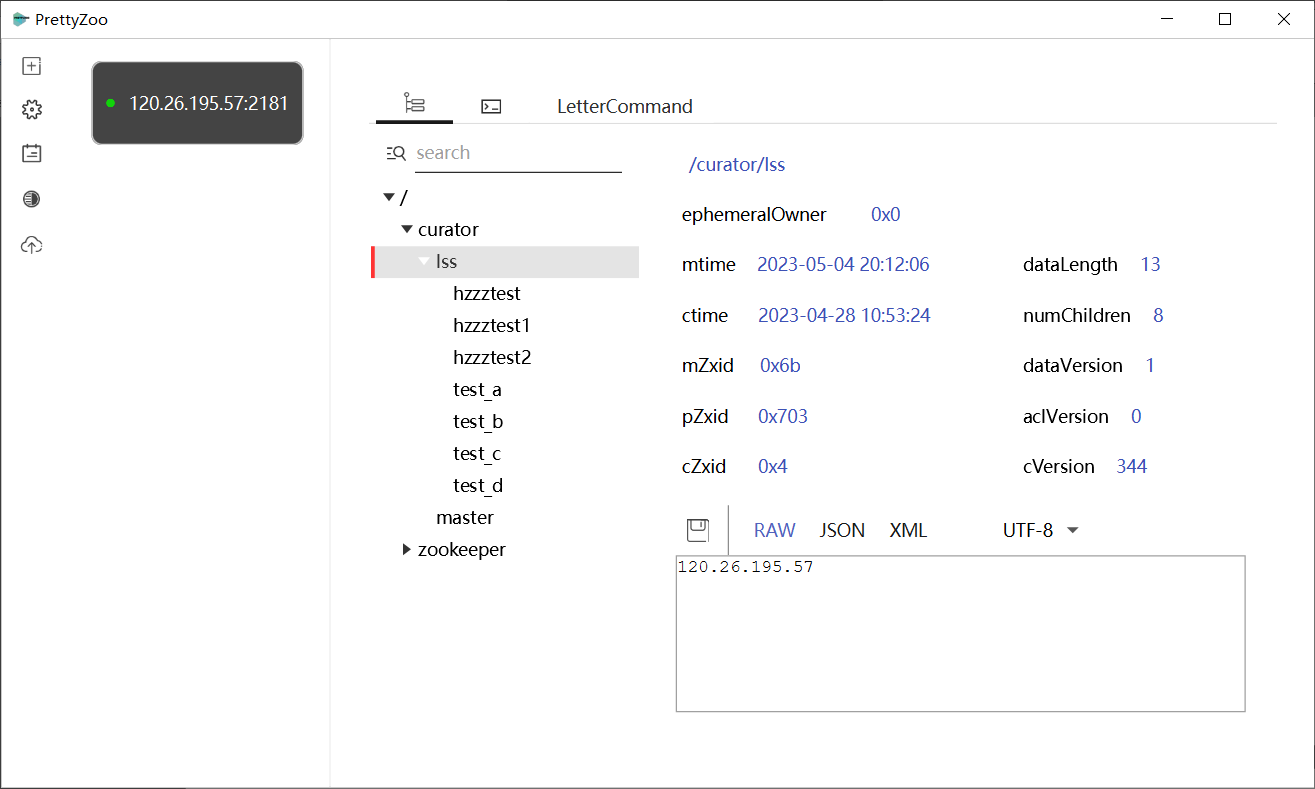
可见Client选取了目前主表数量最少的（空闲的）Region创建新的主表。

## 4.4 Buffer自动刷新

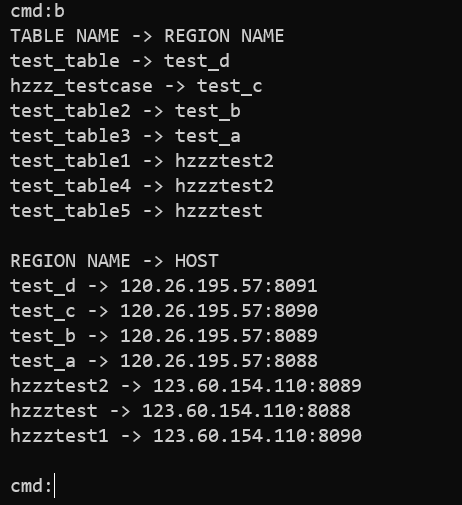
在服务器上关闭brtest Region



查看Zookeeper，发现brtest正确下线



查看Client的Buffer：

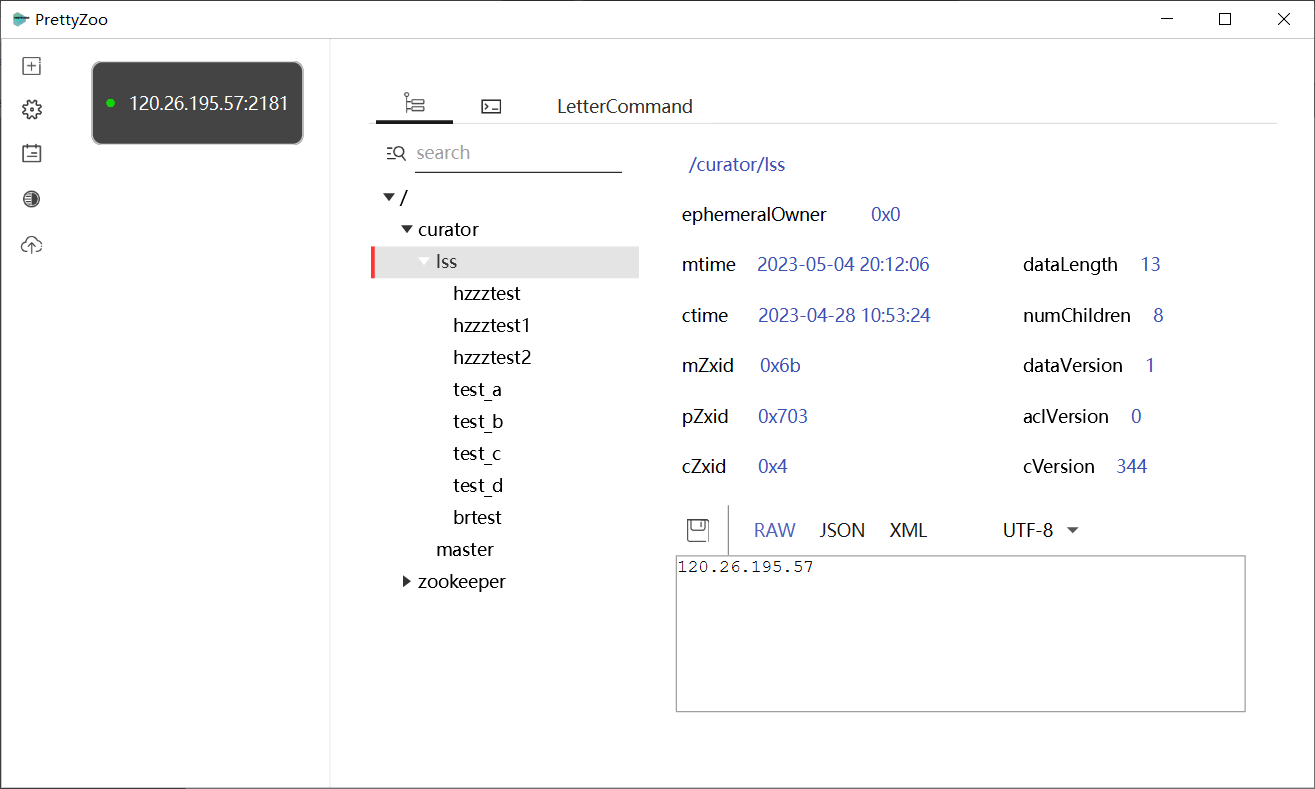


发现brtest已自动清除，Buffer自动刷新成功。

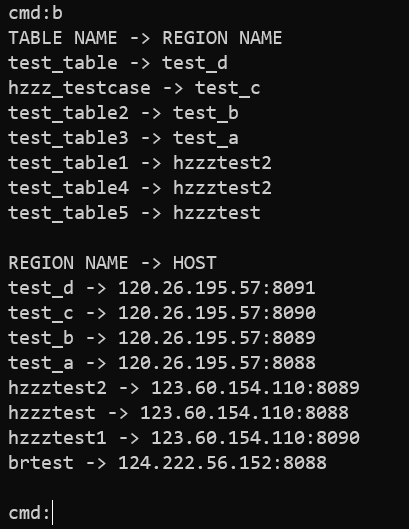
在服务器上重新启动brtest：



查看Zookeeper，发现brtest已经成功上线：



查看Client的Buffer：



发现brtest已经成功加入，Buffer已经自动刷新。

# 五．开发体会

这次分布式数据库的开发让我对于分布式的了解更加深入了，同时也是使用python来进行开发的一次很好的练习。本次项目核心使用了Kazoo库，但是这个库的资料也比较少，很多细节需要详细阅读英文文档才能知晓，因此也踩了不少的坑。整个项目也花了挺长的时间，绝大部分时间都在和一些琐碎但是又重要细节做斗争。印象最深的一个细节是refresh\_buffer函数需要手动停止所有已经打开的监听器，否则可能会出现一次Zookeeper更新引发多次回调函数调用。但是Kazoo库并没有留下对应的api，最后翻找了好久的资料+借助new bing强大的搜索能力才在stackoverflow上找到了非常别扭但是有效的解决方案。

由于多方同时推进的原因，Client的代码是一次性写完的，全部完成之前一次都没有运行过。本以为用了这么多从未使用的外部库，而且这些库还资料很少，试运行+debug会花很多时间，但是最后却很快地通过了测试，并没有预料中的出现debug艰难的情况，也算是一个惊喜了。

# 参考文献

[1] https://kazoo.readthedocs.io/en/latest/

[2]https://stackoverflow.com/questions/40153340/how-to-stop-datawatch-on-zookeeper-node-using-kazoo