浙江大学

题　　目　总体设计报告

授课教师　鲁伟明

组　　长　

组　　员　

组　　员　

组　　员　

组　　员　

目录

1 引言

1.1 项目背景

本项目是一个分布式 MiniSQL 系统，属于《大规模信息系统构建技术导论》的课程项目。

本项目包含 Client，Master，Region Server 和 Zookeeper 共四个模块，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，还实现了前端可视化界面，能够对简单 SQL 语句进行处理和解析，实现了基本分布式数据库的功能，包括容错容灾、负载均衡、副本管理等。

本项目的 Master、Region Server、Zookeeper 使用 Java 语言开发，使用 Maven 作为项目管理工具，Client 和前端使用 Javascript、CSS、HTML 基于 Vue 框架进行开发。整体项目并使用 Github 进行版本管理和协作开发，由小组内的五名成员共同完成。

1.2 系统目标

本项目开发之初针对分布式数据库的特点设计了以下具体功能目标，均得到了较好的实现：

- **负载均衡**

为了避免单个数据点负载过重，我们设计了热点机制，将

频繁访问的数据分散在不同服务器上；同时，在 Master 服务器的统一协调下，对数据的访问会按照轮询规则分摊到不同的 Slave 服务器上。

- **容错容灾**

我们小组针对容错容灾设计了投票机制、多活动中心、冗余和备份、基于临时节点的心跳机制、故障监测与恢复共五项措施，有效保证系统在面对异常情况时的正常运行。

- **副本管理**

对于副本的管理，服务器之间我们使用主从复制来协调多个服务器之间的数据关系，同时在多个 Zookeeper 节点之间我们使用 Master 来统一进行副本一致性的管理。而对于数据库内的数据，我们使用了 CRC32 来完成数据完整性的校验。

- **数据分布和集群管理**

本项目系统中包含多个 Region，每个 Region 中都有一个 Master 和多个 Slave，由 Master 来统一进行集群管理和数据分布管理。

- **分布式查询**

本项目支持基本的 SQL 语句，并且基于 Master 的管理实现了分布式查询。

1.3 项目安装与使用

环境说明

- 系统：Windows 10
- ZooKeeper 3.8.0 + JRE 8
- NodeJS v18.16.0
- Oracle OpenJDK version 20

Zookeeper 安装与配置

1. 将 `apache-zookeeper-3.8.0-bin.tar.gz` 移动至合适的安装路径下，解压。
2. 安装目录下创建 `data` & `log`路径，分别用于存储数据和日志。
3. 复制 `/dependencies` 下的 `zoo.cfg` 至安装目录的 `/conf` 文件夹下
    - 请根据实际情况修改 `dataDir` 与 `dataLogDir` 配置项为实际绝对路径
    - 配置文件样例见 /dependencies/zoo.cfg
4. 将 JRE8 的安装路径设置为系统变量 `JAVA_HOME`

MySQL 初始化

1. 运行 regionServer 的机器需要在本地运行 `init.sql` 脚本以完成数据库初始化

2. 执行以下命令允许使用 `root` 账号进行远程连接：

```
USE mysql;
UPDATE user SET host='%' WHERE user='root';
flush privileges;
```

请关闭防火墙并开放 3306 & 9090 & 9091 以确保项目正常运行

前端项目初始化

1. 进入前端项目根目录 `/front` 执行 `npm install` 以安装必要的依赖
2. 请将 `vue.config.js` 中配置 `target` 为 Master 项目所在机器的实际 IP 地址

regionServer 项目数据源配置

请在 /master/src/main/resources/application.yml 中正确配置 MySQL 中的数据库名称、用户名与密码，以保证项目能正常连接并操作数据库。

运行项目

1. 打开 `/bin` 目录下的 `zkServer.cmd` 以运行 ZooKeeper 服务
2. 运行 Master 项目，自动初始化 Zookeeper 目录结构
3. 运行 RegionServer 项目

4. 运行 Front 项目，使用图形化界面操作数据库

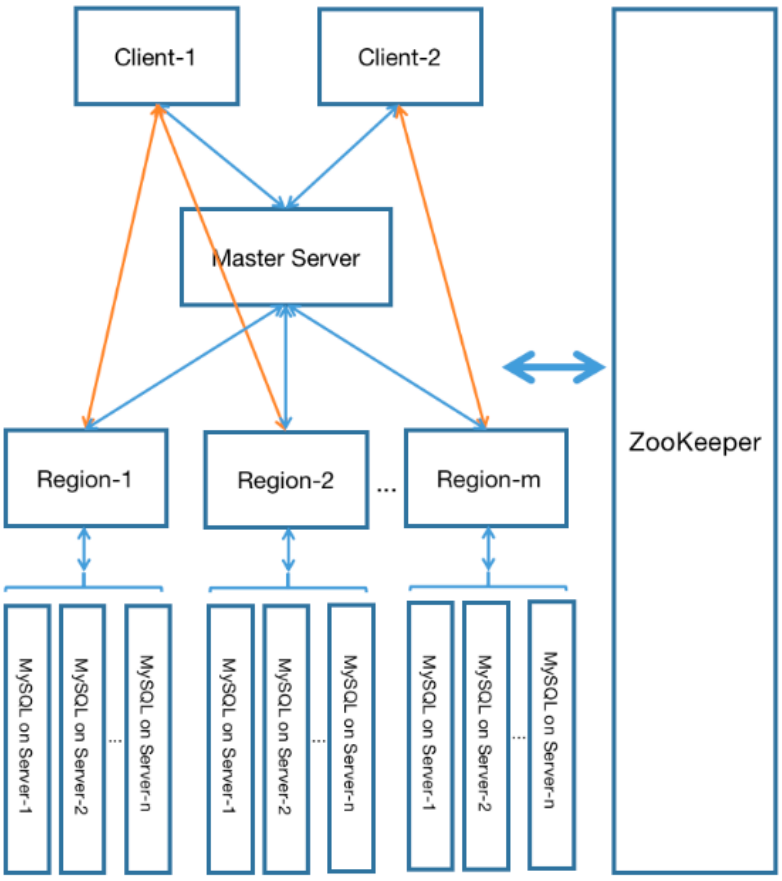
1.4 小组分工与协作

姓名	学号	任务
沈韵泓	3200104392	总体架构设计，Master 和 Region Server 的 Zookeeper 节点的设计与开发，实现基于临时节点的心跳机制、故障检测与恢复、集群管理
龙麟	3200102541	开发 Region Server 模块，实现冗余与备份、副本一致性、数据完整性校验和投票机制。
魏鼎坤	3200105652	开发 Master Server 模块，实现基于热点问题的负载均衡、分布式查询、多活动中心和故障检测与恢复。
裴宇航	3200104179	开发 Client 模块，完成前后端接口设计与实现，实现客户端缓存。
吴书研	3200105501	Web 前端页面开发

2 总体设计

2.1 总体架构

我们小组开发的分布式数据库系统包含了以下四个模块：Master Server（主服务器）、Region Server（区域服务器）、Zookeeper（Zookeeper 服务）和 Client（客户端），通过这四个四个模块的协同工作，实现了高可用性、容灾容错、副本管理、负载均衡、分布式查询等关键功能。Master Server 负责全局管理和调度，Region Server 存储和处理数据，Zookeeper 提供分布式协调和故障检测，Client 负责与分布式系统进行交互。



2.2 项目结构

我们的项目结构如下：

```
.
├── LICENSE
├── README.md
├── dependencies
│   ├── apache-zookeeper-3.8.0-bin.tar.gz
│   ├── jre-8u371-windows-x64.exe
│   └── zoo.cfg
├── front
│   ├── babel.config.js
│   ├── jsconfig.json
│   ├── package-lock.json
│   ├── package.json
│   ├── public
│   │   ├── favicon.ico
│   │   └── index.html
│   ├── src
│   │   ├── App.vue
│   │   ├── components
│   │   │   ├── CreateTable.vue
│   │   │   ├── DropTable.vue
│   │   │   ├── ElseChoice.vue
│   │   │   └── Select.vue
│   │   ├── main.js
│   │   ├── router
│   │   │   └── index.js
│   │   └── store
│   │       └── index.js
│   └── vue.config.js
├── init.sql
├── master
│   ├── pom.xml
│   └── src
│       └── main
│           ├── java
│           │   └── main
│           │       ├── MasterApplication.java
│           │       └── util
│           │           ├── RegionMeta.java
│           │           └── Zookeeper.java
│           └── resources
│               └── application.yml
```

```
└── regionServer
    ├── pom.xml
    ├── sql
    └── src
        └── main
            ├── java
            │   └── main
            │       ├── Dump.java
            │       ├── MasterApplication.java
            │       └── util
            │           ├── Checksum.java
            │           ├── ZkListener.java
            │           └── Zookeeper.java
            └── resources
                └── application.yml
```

2.3 各模块功能设计

Master Server 主服务器

主服务器是整个分布式数据库系统的核心组件，负责全局的管理和调度。其主要功能包括：

- 主从复制管理：负责管理和监控区域服务器的主从复制关系，确保数据的一致性和冗余。
- 元数据管理：维护数据库的元数据信息，包括表结构、索引信息等，用于分布式查询和数据管理。
- 负载均衡：根据系统的负载情况和区域服务器的状态，监测是否存在热点，如果存在热点再进行数据表的迁移，实现负载均衡；同时根据区域服务器的数据量，决定下一张数据表存在哪个区域服务器上。
- 故障监测与恢复：监测区域服务器的状态和健康情况，当发生

故障时，进行故障恢复和节点替换。

Region Server 区域服务器

区域服务器是存储和处理数据的节点，负责具体的数据操作和查询执行，每个区域服务器管理着自己的本地数据库。其主要功能包括：

- 数据存储和访问：负责接收来自客户端的读写请求（这些请求的管理是 Master Server 的工作），并将数据存储在本地的数据区域中。处理查询请求，执行查询计划，并返回结果。
- 副本管理：维护本地数据区域的副本状态，确保数据的冗余和容错性。与主服务器进行主从复制，保持副本之间的数据一致性。
- 基于临时节点的心跳机制：定期向主服务器发送心跳信号，报告自身的状态和可用性。
- 容错容灾与故障恢复：当区域服务器发生故障或离线时，参与故障检测和故障恢复过程，以保持数据的可用性和一致性。

Zookeeper

Zookeeper 是一个分布式协调服务，用于管理和协调分布式系统中的各个节点，主要负责以下任务

- 选举机制：协助主服务器的选举过程，确保从 Region 传出的数据是经过内部选举确认无误的。

- 节点注册和发现：区域服务器和客户端通过注册到Zookeeper，可以得到集群中其他节点的信息，以实现节点的发现和动态配置。
- 故障检测与恢复：监测节点的状态，当节点故障时，将自动删除其临时节点，然后进行故障恢复。

Client 客户端

客户端是与分布式数据库系统进行交互的应用程序或用户界面。客户端负责发送请求和接收响应，并与数据库进行通信。其主要功能包括：

- 数据访问接口：提供给应用程序的接口，用于读写数据、执行查询和事务管理等操作。
- 负载均衡与路由：根据负载情况和网络拓扑，将请求路由到合适的区域服务器，实现负载均衡和优化性能。
- 容错处理：当发生节点故障或网络中断时，客户端能够处理错误和自动切换到其他可用节点，保持数据的可用性。
- 客户端缓存：存储和记录分布式系统中数据表所属的服务器，提高用户访问分布式系统的效率。
- 安全认证与权限管理：负责用户认证和授权，确保只有经过授权的用户可以访问数据库，并保护数据的安全性。

3 核心模块设计与实现

3.1 Master Server 设计与实现

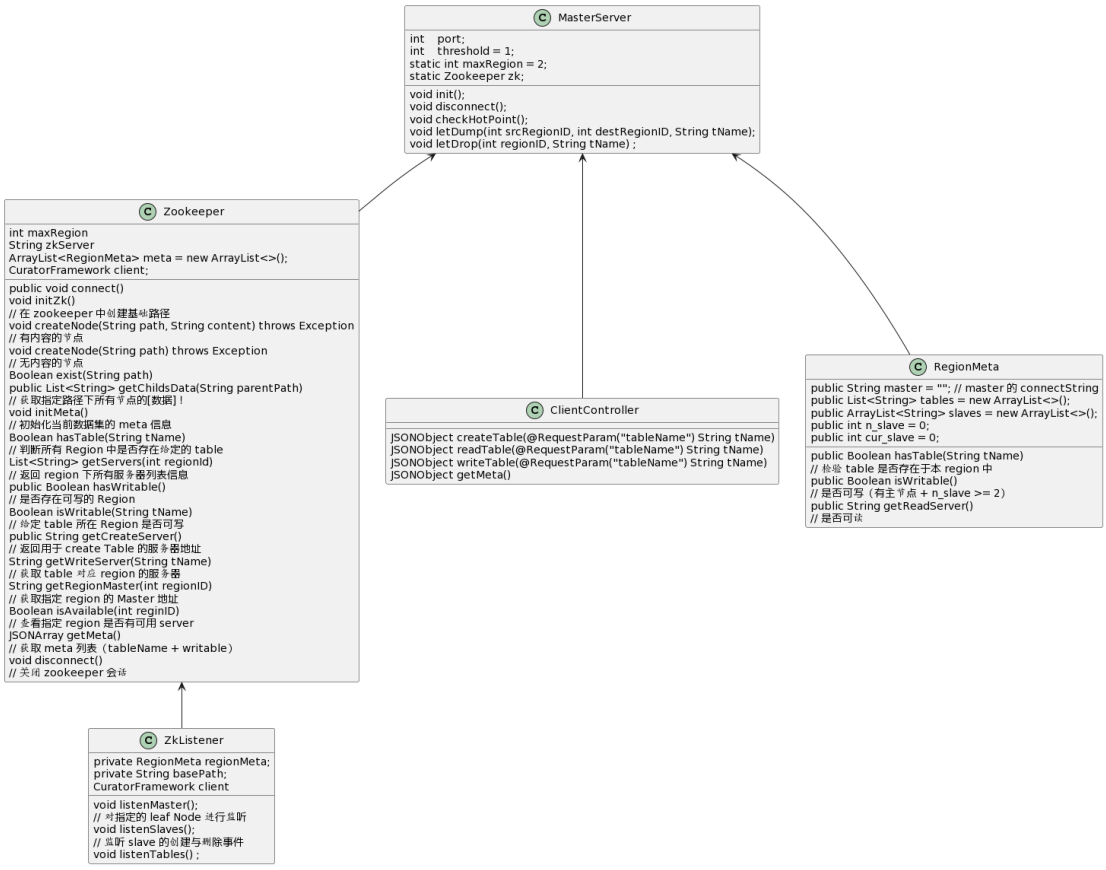
Master Server 功能

Master Server 主要需要完成以下功能：

- Master Server 需要与 Client 建立通信并处理来自 Client 端的请求。它作为系统的入口点，接受 Client 发起的各种操作请求，例如文件的读取、写入、删除等，并按照操作类型与访问数据对请求进行路由，最终返回实际响应操作的 Region Server 地址。这个过程需要确保通信的安全性和可靠性，以及提供适当的错误处理和反馈机制。
- Master Server 通过 Zookeeper 对 Region Server 的状态进行即使监听，从而实现节点间的协调管理。通过监听 Zookeeper 中特定节点的状态变化，Master Server 能够及时发现节点的故障或变动并进行响应。通过这种集中管理的方式，Master Server 可以实现副本管理、容错容灾等重要功能。它可以监控和管理文件的副本数量，确保数据的冗余备份，从而提高系统的可靠性和容错性。同时，Master Server 还负责调度和负载均衡，确保文件系统的高性能和高效利用。
- 在 Master Server 的管理之下，分布式文件系统可以实现更高级别的功能和特性。Master Server 能够定时统计访问热点，

并在多个 Region 之间进行数据表迁移，并将用户的 write/read 操作分别路由至各 Region 的 Master/Slave 节点，以实现负载均衡。它还可以支持文件系统的扩展性，允许动态地添加或删除 Region Server 节点，以适应不同规模和负载的变化。此外，Master Server 还承担着元数据的管理和维护任务，包括文件和目录的元数据存储、更新和检索。

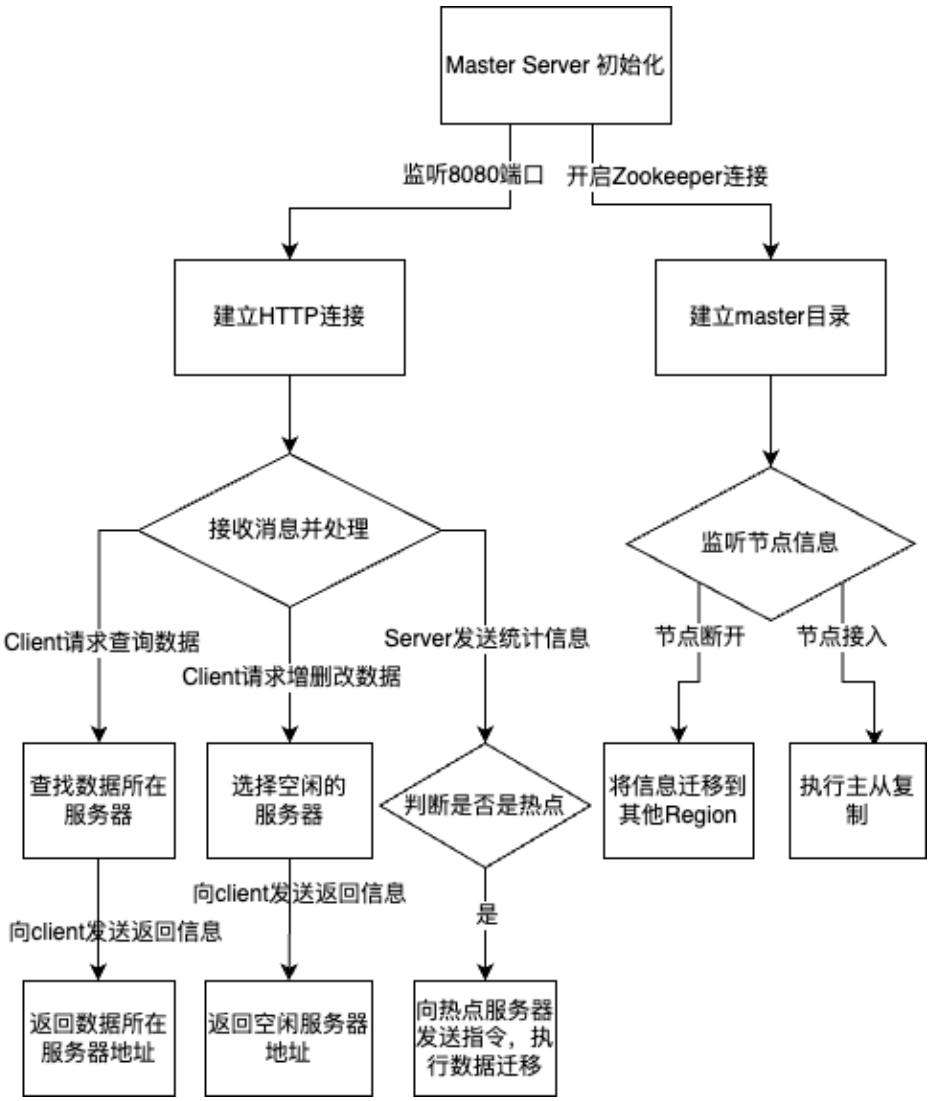
Master Server 类图



Master Server 工作流程

我们小组基于 Java 的 SpringBoot 框架与 Curator 客户端实

现了 Master Server。通过 Curator 提供的 REST ful 风格 API，与 ZooKeeper 进行交互来管理分布式文件系统。并对部分常用功能进行二次封装，并对 NodeCacheListener 与 TreeChacheListener 进行继承拓展，以适应不同节点各异的回调需要。



Master Server 实现

初始化阶段：

- 在应用程序启动时，通过@PostConstruct 注解的 init()方法进

行初始化操作。

- init()方法获取本机的 IP 地址，并打印出 Master 服务器和 ZooKeeper 服务器的信息。
- 创建一个ZooKeeper实例，并与ZooKeeper服务器建立连接，为特定节点注册监听。
- 通过 initZk() 方法初始化 ZooKeeper 中的节点结构。

请求处理阶段：

- Master Server使用@RequestMapping注解匹配不同的请求处理方法。
- createTable()方法处理创建表的请求，它首先检查是否存在同名的表，并检查是否有可用的 Region（可写的）。如果满足条件，返回响应状态码 200 和可用的服务器地址。
- readTable()方法处理读取表的请求，它检查请求的表是否存在，如果存在，返回响应状态码 200 和可用的服务器地址。
- writeTable()方法处理写入表的请求，它检查请求的表是否存在，并检查该表所在的 Region 是否可写。如果满足条件，返回响应状态码 200 和可用的服务器地址。
- getMeta()方法返回所有表的元信息，包括表名和可写性状态。

应用关闭阶段：

- 在应用程序关闭之前，通过@PreDestroy 注解的 disconnect() 方法断开与 ZooKeeper 服务器的连接，使得创建的临时节点能够被正确销毁。

3.2 Region Server 设计与实现

Region Server 功能

Region Server 作为分布式文件系统的组成部分之一，主要功能如下：

- 通过 ZooKeeper 对分区下的服务器集群进行管理，进一步划分为 1 个主服务器（Master）和 n 个从服务器（Slave），分别负责不同的工作。同时为了保证该划分方式的稳定性，Slaves 需要实时监听 Master 节点，一旦 Master 意外下线，所有 Slaves 将竞选 Master 以尝试弥补其空缺。
- 与 Master Server 建立通信，接受来自 Master Server 的客户端请求，并根据请求类型交由特定的主服务器或从服务器进行进一步处理。
- 通过监听 ZooKeeper 中的特定节点并处理来自 Master 的请求实现与同一 Region 中其余 Server 的数据同步与跨 Region 数据迁移。
- 负责实际执行用户提供的 SQL 操作。具体而言，为确保数据被优先更新到主服务器，由主服务器，即 Master 进行创建或删除表操作，并将表信息变化同步到 ZooKeeper 中，同时执行增删改操作，并通过与其他 Region Server 的数据同步实现数据的一致性和可靠性；而对于并不会对数据产生实际变更的查询（SELECT）操作，为减轻 Master 的负载，则会按照一

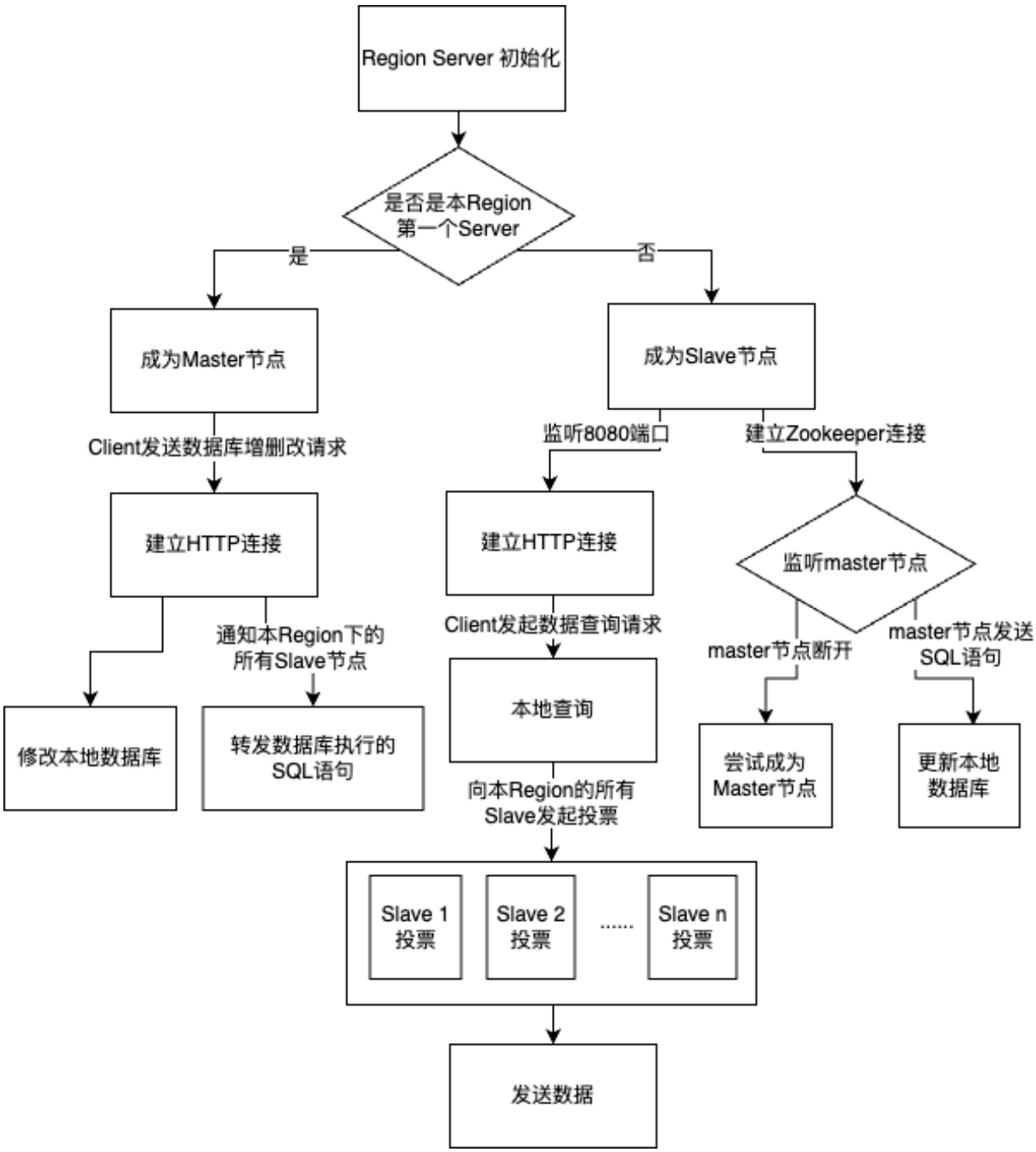
定的轮询规则转发给特定 Slave 执行，同时会根据投票结果，以少数服从多数的原则判断返回结果是否可靠有效，从而进一步保证 Slave 与 Master 中存储数据的一致性。

- 通过与 ZooKeeper 相连接，实现文件的分布式存储与请求的分布式处理。

Region Server 类图

Region Server 工作流程

我们小组基于 Java 实现的 Region Server 也是一个基于 Spring Boot 的应用程序，使用 Curator 客户端实现对 ZooKeeper 的操作。此外实现了数据校验功能，并使用 mysqldump 实现远程数据同步。



Region Server 实现

初始化阶段：

- 在应用程序启动时，通过@PostConstruct 注解的 init()方法进行初始化操作。

- init()方法从用户输入获取 ZooKeeper 服务器的 IP 地址，并获取本机的 IP 地址和端口号。
- 创建一个 ZooKeeper 实例，将 DataSource 和服务器地址传递给 ZooKeeper 对象，并与 ZooKeeper 服务器建立连接。
- 尝试通过创建 /region[n]/master 节点以成为 Region 中的主副本：成功，则更新 /region[n]/tables 下存放的 table 信息；反之，注册对 /region[n]/master 节点删除事件的监听，并在原本的主副本下线后尝试竞选。

请求处理阶段：

- Region Server 使用@RequestMapping 注解来匹配不同的请求处理方法。
- createTable()方法处理创建表的请求，它执行用户提供的 SQL 语句来创建新表，并将表信息同步到 ZooKeeper 中。
- dropTable()方法处理删除表的请求，它执行用户提供的 SQL 语句来删除表，并将表信息从 ZooKeeper 中移除。
- select()方法处理查询操作的请求，它执行用户提供的 SQL 语句并返回查询结果。
- execute()方法处理其他类型的 SQL 语句执行请求，它执行用户提供的 SQL 语句。

数据同步阶段：

- sync()方法用于向 slave 节点同步 SQL 操作。如果当前节点是主节点（Master Server），它将获取所有从节点（相同

Region 下其他 Region Server）的地址，并将 SQL 操作转发给它们。

- sync()方法构建一个 POST 请求，将 SQL 操作作为参数发送到从节点的/execute 路径。通过使用 RestTemplate 进行 HTTP 请求发送和接收响应。

应用关闭阶段：

- 在应用程序关闭之前，通过@PreDestroy 注解的 disconnect()方法断开与 ZooKeeper 服务器的连接，使临时节点能够正常删除，从而即时向 Master 更新服务器活跃状态。

3.3 Client 设计与实现

Client 功能

客户端是与分布式数据库系统进行交互的用户接口，主要实现了以下功能：

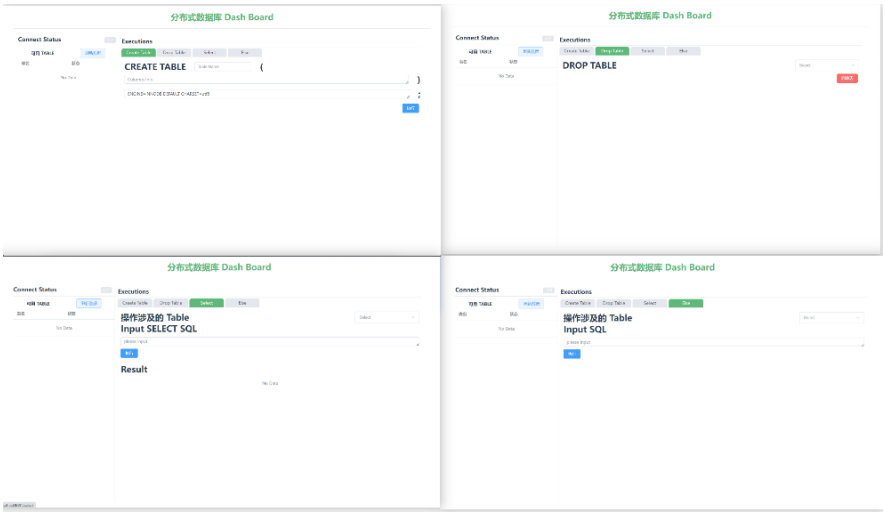
- Client 能接受用户的输入，并对输入进行简单解析以及适当处理，进而简化用户的交互过程。用户可以用 HTML 表单的形式执行数据表的创建和删除，也可以选择某一数据表并输入 SQL 语句来执行查询操作以及其他操作。
- Client 需要与 Master Server 建立通信，并向 Master Server 发送各种数据库操作请求，根据 Master Server 的响应来确定执行实际操作的 Region Server 节点。在此之后，Client 需要与

Region Server 节点建立连接，并请求其执行数据库操作。这个过程需要确保通信的安全性和可靠性，并提供适当的错误处理和反馈机制。

- Client 实现了客户端缓存功能，以提高系统的查询效率和 I/O 性能。Client 维护数据表与对应 Region Server 的 IP 地址的映射关系的缓存。在每次执行操作时，先在客户端缓存中检查是否包含该表对应的 Region Server 的 IP 地址。如果能直接获取到 Region Server 的地址，则不再与 Master Server 建立连接，而是直接与 Region Server 通信；否则，再向 Master 进行询问，并且把询问获知的数据表与 Region Server 地址的映射关系加入本地缓存。

Client 设计

Client 共包含四个界面，分别对应数据库的四类操作：创建数据表的 CreateTable 界面，删除数据表的 DropTable 界面，执行查询操作的 Select 界面，执行其他操作的 ElseChoice 界面。



Client 工作流程与实现

我们小组实现的 Client 是基于 Vue 框架和 Element 组件库实现的前端应用，通过 Axios 实现前端同 Master Server 和 Region Server 的交互。Client 使用 HTML 提供的 localStorage 实现客户端缓存功能。

处理用户输入阶段：

- 在创建表操作中，Client 会获取用户输入的表名以及表字段定义；在删除表操作中，Client 只获取用户选择的表；在查询操作和其他操作中，Client 会获取用户选择的表以及输入的 SQL 语句。
- 如果用户输入为空，则会通过$message 给出警告信息。
- 创建表和删除表的操作中，用户不会输入完整的 SQL 语句，因此 Client 会根据用户输入生成符合语法要求的 SQL 语句。

访问客户端缓存阶段：

- Client 根据表名在 localStorage 中查找是否存在对应的(tableName, address)映射关系。
- 如果成功在缓存中查找到对应的 address，则通过 Axios 发送 POST 请求给该地址对应的 Region Server，以 tableName 和 SQL 语句作为参数。如果访问成功，则可直接在 Region Server 上执行数据库操作；如果访问失败，则可能发生缓存失效、Region Server 故障等情况，Client 通过 localStorage.removeItem()清除该条缓存记录，并再去访问

Master Server。

- 需要说明的是，CreateTable 界面内的创建表操作不会访问客户端缓存（因为新创建的表不可能有缓存），而其他操作都会先访问客户端缓存。

访问 Master Server 阶段：

- 如果没有在缓存中找到对应的 address，或缓存地址访问失败，则 Client 通过 Axios 与 Master Server 建立连接，以 tableName 作为参数，通过 GET 方法询问维护该表的 Region Server 的地址。
- 在获知 Region Server 的地址后，Client 以 tableName 和 SQL 语句作为参数，通过 Axios 发送 POST 请求，执行实际的数据库操作。Client 为用户呈现数据库操作的结果或报错信息。

更新缓存阶段：

- 如果在 Region Server 上成功执行了数据库操作，则在 Client 本地缓存中通过 localStorage.setItem() 新增 (tableName, address)的映射关系。

3.4 Zookeeper 设计与实现

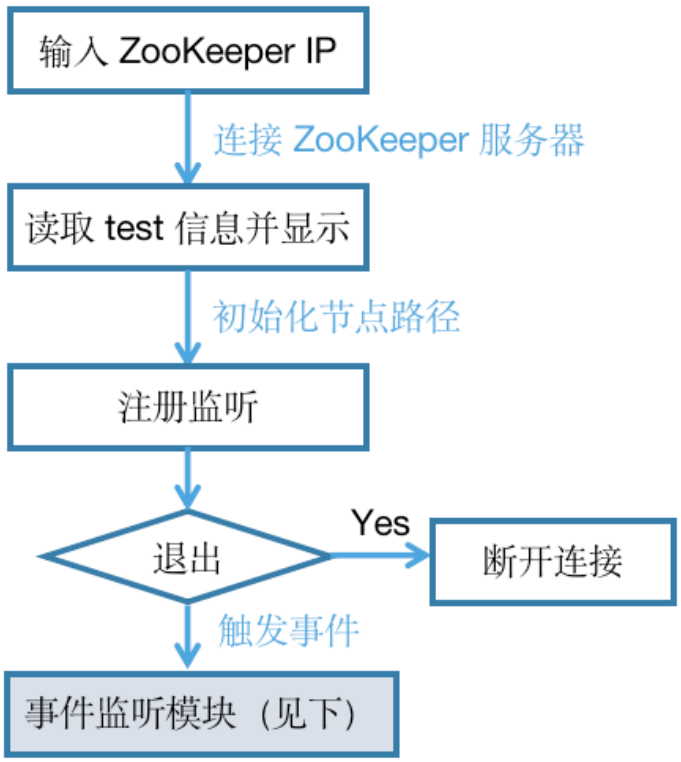
Zookeeper 架构设计

```
├──── test "hello" #  测试连接用数据
├──── region1
|         ├──── (tmp) master "masterIP:port"
|         ├──── tables
```

```
|       |       └──── (tmp) [table name] "CRC Code"
|       └──── slaves
|               └──── (tmp) [slave ID] "slaveIP:port"
└──── region2
        ├──── (tmp) master "masterIP:port"
        ├──── tables
        |       └──── (tmp) [table name] "CRC Code"
        └──── slaves
                └──── (tmp) [slave ID] "slaveIP:port"
```

Master Server 的 Zookeeper 功能与实现

Master Server 的 Zookeeper 流程图如下



- **初始化连接：**

在 connect()方法中，通过创建 CuratorFramework 实例连接到 ZooKeeper 服务器。这里使用了 Curator Framework 库，它是 Apache Curator 提供的一个高级 ZooKeeper 客户端，简化了与

ZooKeeper 的交互。通过指定 ZooKeeper 服务器的 IP 地址和重试策略，创建了 CuratorFramework 客户端。

- **初始化 ZooKeeper 节点：**

在 initZk()方法中，通过使用 CuratorFramework 客户端创建了一些基础路径节点。这些节点用于存储分布式系统中的数据和元数据。具体而言，我们在每个 region 下创建了/tables 和/slaves 两个持久化子节点，其中/tables 的各子节点用于保存 table 的元信息，/slaves 的各子节点用于保存备份服务器的元信息。

- **监听节点变化：**

在 initMeta()方法中，初始化了元数据信息。为每个区域创建了一个 RegionMeta 对象，并创建了相应的 ZkListener 实例。ZkListener 使用 CuratorFramework 客户端监听节点的变化，包括/master、/tables 和/slaves 节点。通过注册相应的监听器，可以实时获取节点的变化情况。

- **监听器：**

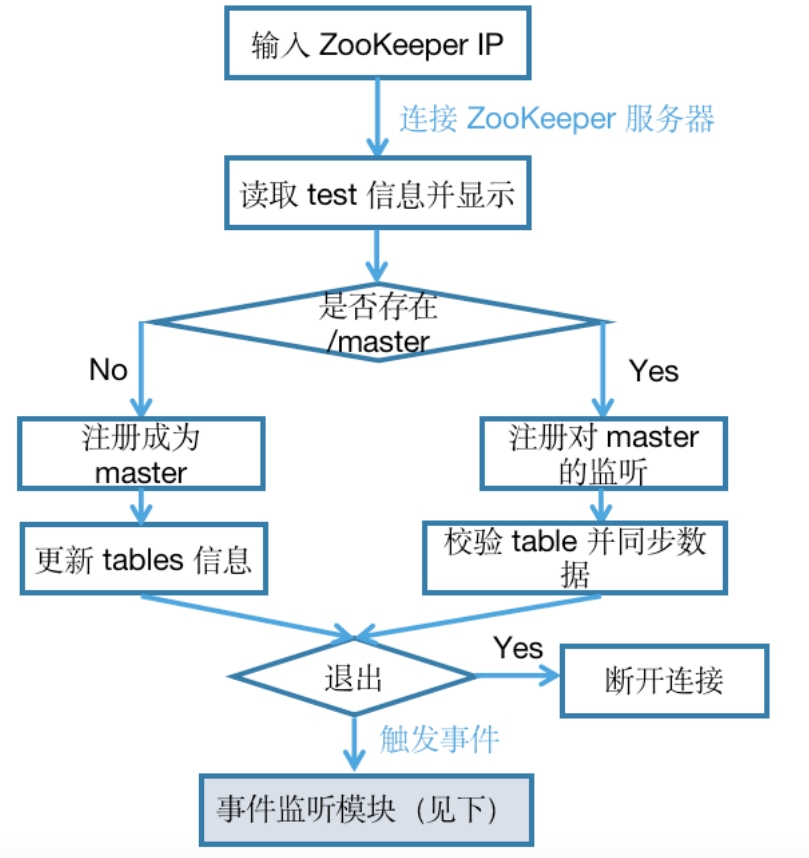
在 ZkListener 内部类中实现了三个监听器：MasterListener、SlaveListener 和 TablesListener。这些监听器通过 CuratorFramework 客户端注册到相应的节点上，以便监听节点的变化。例如，MasterListener 监听/master 节点的变化，当该节点被创建或删除时，会触发相应的处理逻辑。SlaveListener 和 TablesListener 分别监听/slaves 和/tables 节点的变化，用于更新 table-region 映射关系与可用节点列表信息。

- **数据查询和操作：**

在 Zookeeper 类中提供了一些方法来查询和操作分布式系统的数据和元数据。具体而言，hasTable()方法用于判断所有区域中是否存在指定的表；isWritable()方法用于判断指定表所在的区域是否可写；getCreateServer()方法返回可用于创建表的服务器地址等等。

Region Server 的 ZooKeeper 功能与实现

Region Server 的 Zookeeper 流程图如下



- **连接 ZooKeeper 服务器：**

在 connect() 方法中，Region Server 通过创建

CuratorFramework 实例连接到 ZooKeeper 服务器。这个连接允许 Region Server 与 ZooKeeper 进行通信，执行各种操作。

- **初始化：**

init()方法用于 Region Server 的初始化。它创建一个 ZkListener 实例，并进行一些初始化操作。其中，主要是检查是否存在/master 节点来确定 Region Server 的角色（主节点或从节点）。

- **Master 选举：**

与 Master 选举相关的两个函数是 beMaster()和 beSlave()，他们将在 Server 上线的时候根据实际情况被调用：

beMaster()：当 Region Server 启动时，它首先尝试成为主节点（Master）。它在/master 节点下创建一个临时节点，并将自身的地址信息写入该节点。如果创建成功，表示成为主节点。在从从节点转变为主节点后，它关闭对/master 节点的监听，并删除/slaves 节点下与自身对应的节点。然后，它将本地数据库中的表信息写入/tables 节点下的各子节点，表示它负责这些表的管理。

beSlave()：如果 Region Server 无法成为主节点，则成为从节点（Slave）。它在/slaves 节点下创建一个临时节点，并将自身的地址信息写入该节点。然后，它注册对/master 节点的监听。作为从节点，它从/tables 节点中获取所有表的信息，并使用 JDBC 连接远程数据库，将远程数据库中的表数据复制到本地数据库以

实现初始数据同步。

- **添加和移除表信息：**

addTable()方法允许 Region Server 将新的表信息写入到 ZooKeeper 的/tables 节点下。它创建一个临时节点来表示表的存在。

removeTable()方法允许 Region Server 从 ZooKeeper 的/tables 节点中移除表信息，通过删除相应的临时节点来表示表已被从该 Region 中移除。

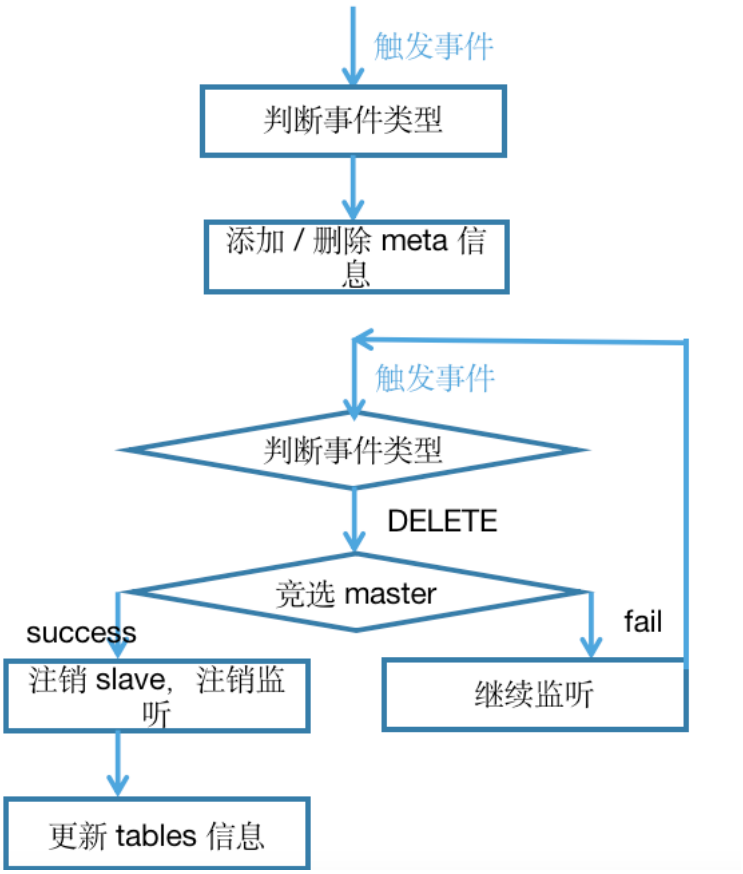
- **获取从节点信息：**

getSlaves()方法允许 Region Server 从 ZooKeeper 的/slaves 节点中获取所有从节点的地址信息。它返回一个列表，包含指定 Region 中所有从节点的地址。

- **监听器实现：**

ZkListener 是一个内部类，用于注册和处理 ZooKeeper 节点的监听器。listenMaster()方法注册对/master 节点的监听。当/master 节点被删除时，会触发相应的处理逻辑，如尝试成为主节点。stopListenMaster()方法用于停止对/master 节点的监听。

ZooKeeper 的监听功能设计与实现



- **支持 master 模块 meta 信息的自动维护**

  通过对事件类型的判断，动态对本地的 meta 信息进行修改，即时维护可用服务器、数据表信息，以及 table-region 映射关系，实现对用户请求的正确路由

- **支持 regionServer 自动进行 master 选举**

  上线时：自动检测 master 是否存在

  原 master 故障时：自动尝试竞选主节点

4 核心功能设计与实现

4.1 负载均衡

在分布式数据库中，负载均衡将工作和数据均匀分布到多个数据库节点上，避免单个数据点负载过重。在本项目中，我们小组从以下角度进行思考，完成了负载均衡设计

热点问题

我们小组参考了 Google File System 中对热点问题的设计，我们让 Region Server 定期向 Master Server 发送信息，信息内容是这段时间内 Region Server 收到的访问量。当一个 Region Server 收到大量访问的时候，我们可以认为这个 Region Server 上的数据库中保存了一些特别容易被访问的 Table，从而导致了大量的访问。

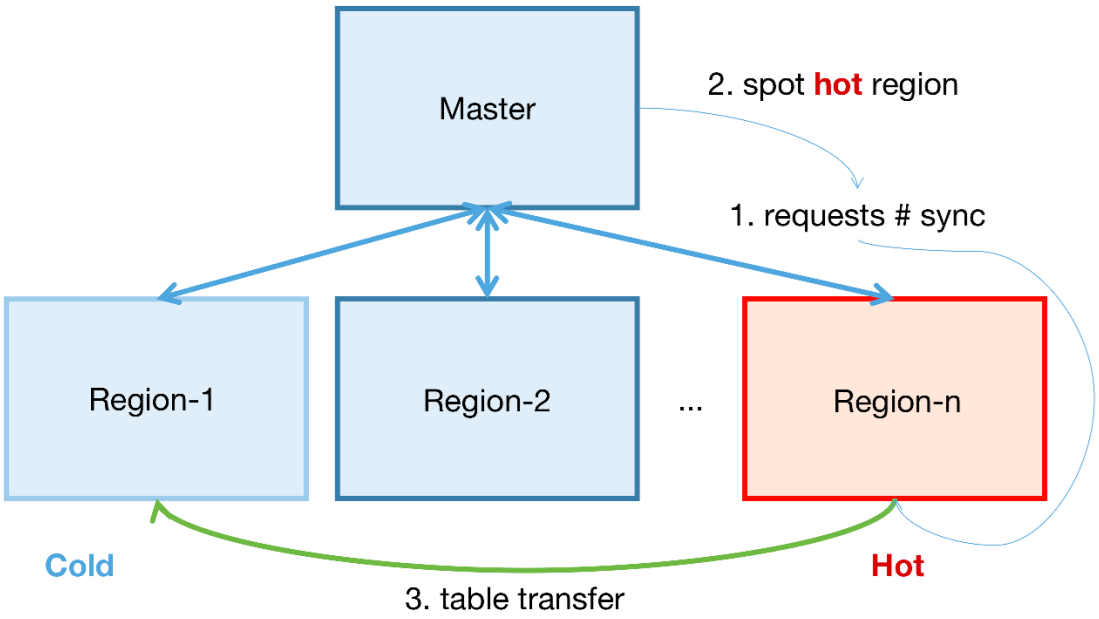
因此解决方法也非常显而易见，我们将这些容易被访问的 Table 转移到其他数据节点上即可。具体而言，我们设计了如下机制：

1. 每个 Region Server 进行本节点的访问计数，当 Master 进行热点统计时返回期间访问数
   - 目前定期返回的时间间隔设置为 2000ms
   - 统计结果返回之后会清零
2. 在 Master 节点中预先设置判定热点的阈值与定时任务的

Interval。在 Interval 内接受了超过阈值请求的 Region 将作为热点候选等待迁移。

3. 每间隔一个 Interval，Master 向所有存活的 Region Server 发起计数请求，并按以 Region 为最小单位进行聚合统计，找出具有 Max/Min 访问总量的 Region。
4. 按照以下标准判断是否进行热点迁移：
   - 具有 >= 2 个可写的 Region
   - MaxVisited Region 的总访问量 > 阈值
   - MaxVisited Region > 2 * MinVisited Region

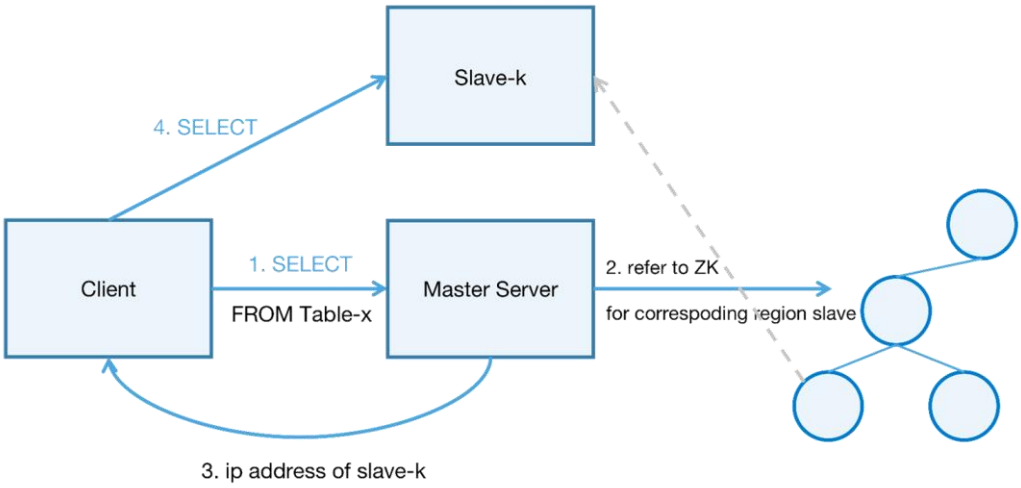
   若满足以上条件，则在 MaxVisited Region 与 MinVisited Region 间以 Table 为单位进行数据迁移。
5. 数据迁移
   - 将 MaxVisited Region 中的半数数据表迁移至 MinVisited Region：
   - Master 获取待迁移 Table 列表（MaxVisited Region 中的前半数 Table）
   - Master 要求 MinVisited 中的所有 region Server 从 MaxVisited Region 的主副本中同步所有待迁移表
   - Master 要求 MaxVisited 中的所有 region Server 删除已完成迁移的表。



轮询查询

在每一个 Region 中，都存在一个 Master 和多个 Slave。本身 Slave 的存在是为了保护数据，在 Slave 上保存这个 Region 的数据，可以保证在 Master 下线之后，还能从其他 Slave 上读取到数据。但既然 Slave 保存了数据，我们便想到了可以让 Slave 分担一些 Master 的访问，从而保证 Master 的负载不会过大。

具体而言，我们设计了对 Table 的轮询查询。当一条 Select 指令到达 Master 时，我们不会从 Master Server 的本地数据库读取数据（因为 Master 还要处理数据增删、DDL 语句和对 Zookeeper 节点的控制，因此本身就有大量的负载），而是将对这个 Table 的访问根据轮询的规则分摊到不同的 Slave 上。具体而言，Master 会根据轮询规则将负责本次查询的 Slave 的 IP 地址发送给 Client，然后 Client 会向 Slave 发起查询请求，从而实现轮询查询。



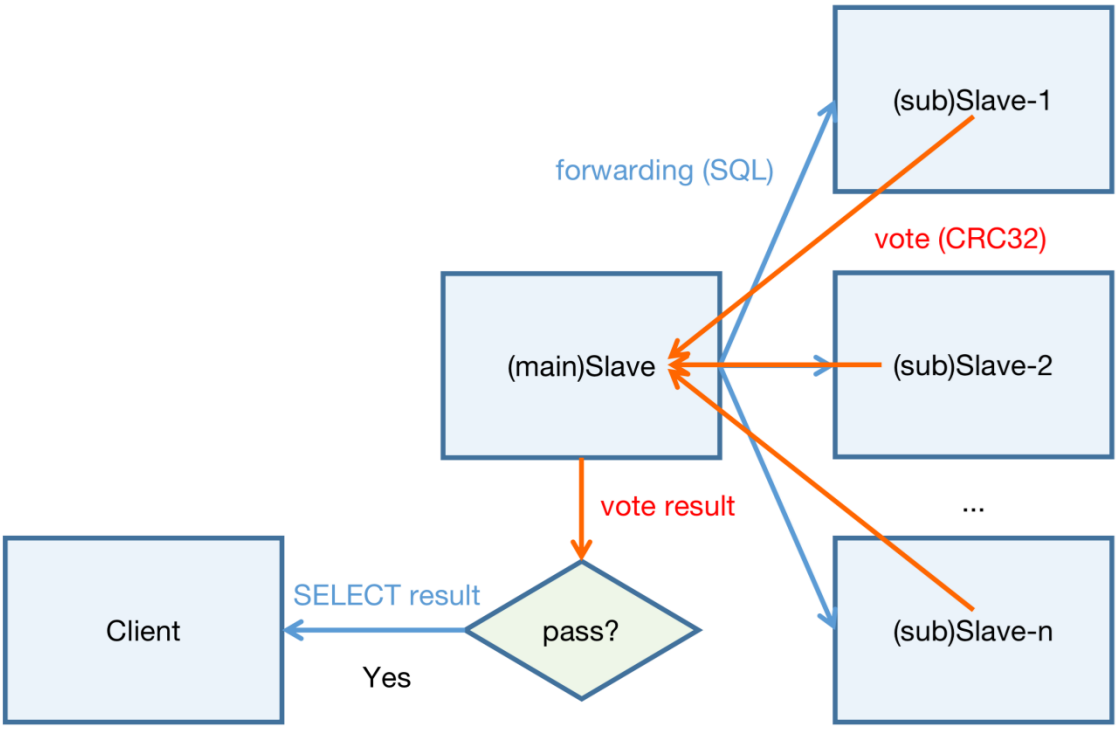
4.2 容错容灾

容错和容灾是分布式系统中确保系统可靠性和可用性的重要保障。它们旨在处理节点故障、网络中断和其他意外情况，以确保系统在面对异常情况时仍能正常运行。在我们的项目中，我们从以下方面来实现容错容灾：

投票机制

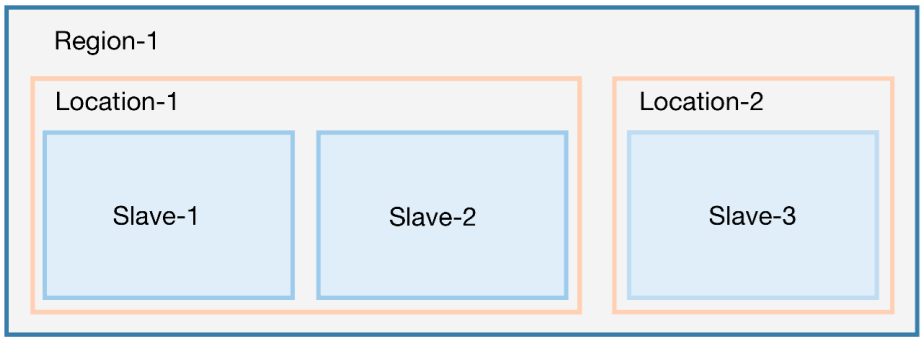
由于 Slave 对于数据更新的优先级较低，为了对其存储的数据进行版本校验，当一个 Slave（下称主 Slave）接收到 SELECT 指令时，它同时也会将该 SQL 指令通过/vote 路由转发给其他所有 Slaves，其他 Slaves 接收到请求后会在本地执行相应的查询操作，并返回查询结果的校验和。主 Slave 在接收到所有投票结果后，会依次与自身本地查询结果的校验和进行比较，倘若与其相同的票数少于总数的一半，则认为主 Slave 查询结果出错，需要重新

向 Master 请求更新数据；反之则认为结果可信，可返回给客户端。



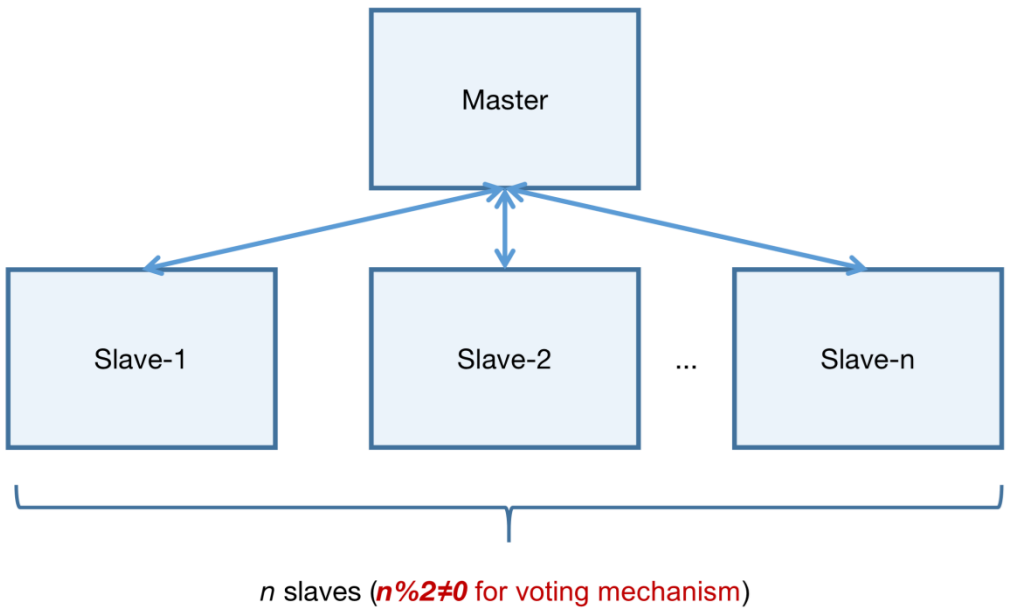
多活动中心

在实际生活中，有时会因为不可抗力因素，导致某些地理位置的所有服务器全部出错。针对此，我们借鉴了 Google File System 的多活动中心的做法，在运行项目的时候，同一个 Region 要至少保证有一个 Slave 服务器和其他几个服务器的 IP 地址不同，确保位于不同的活动中心。



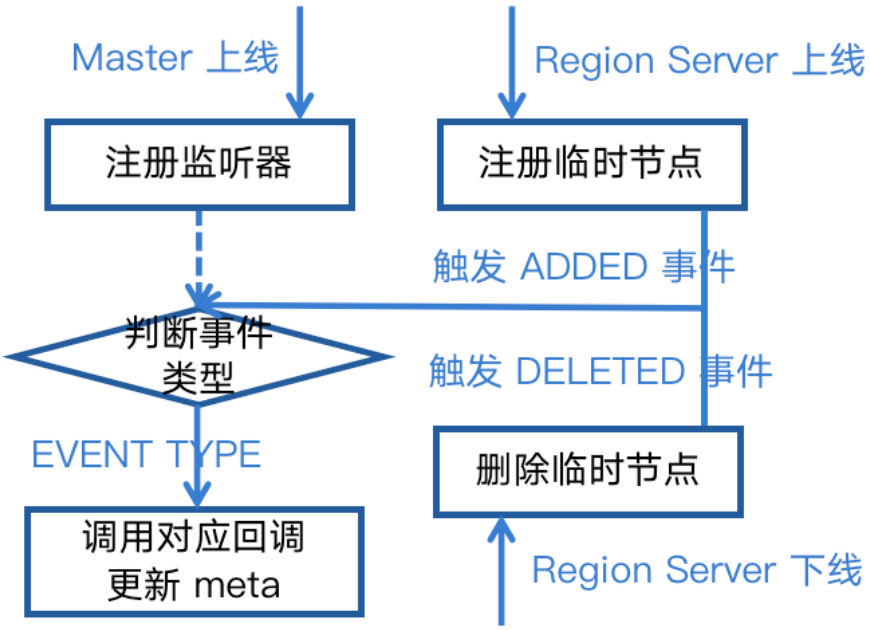
冗余与备份

冗余与备份是通过数据备份和复制来提供容错和容灾。我们小组为每一个 Region 都设计了一个 Master 和多个 Slave，每个 Slave 都会保存一份 Master 的数据，并且 Master 每次更新数据都会通知所有 Slave，从而实现数据的冗余与备份。



基于临时节点的心跳机制

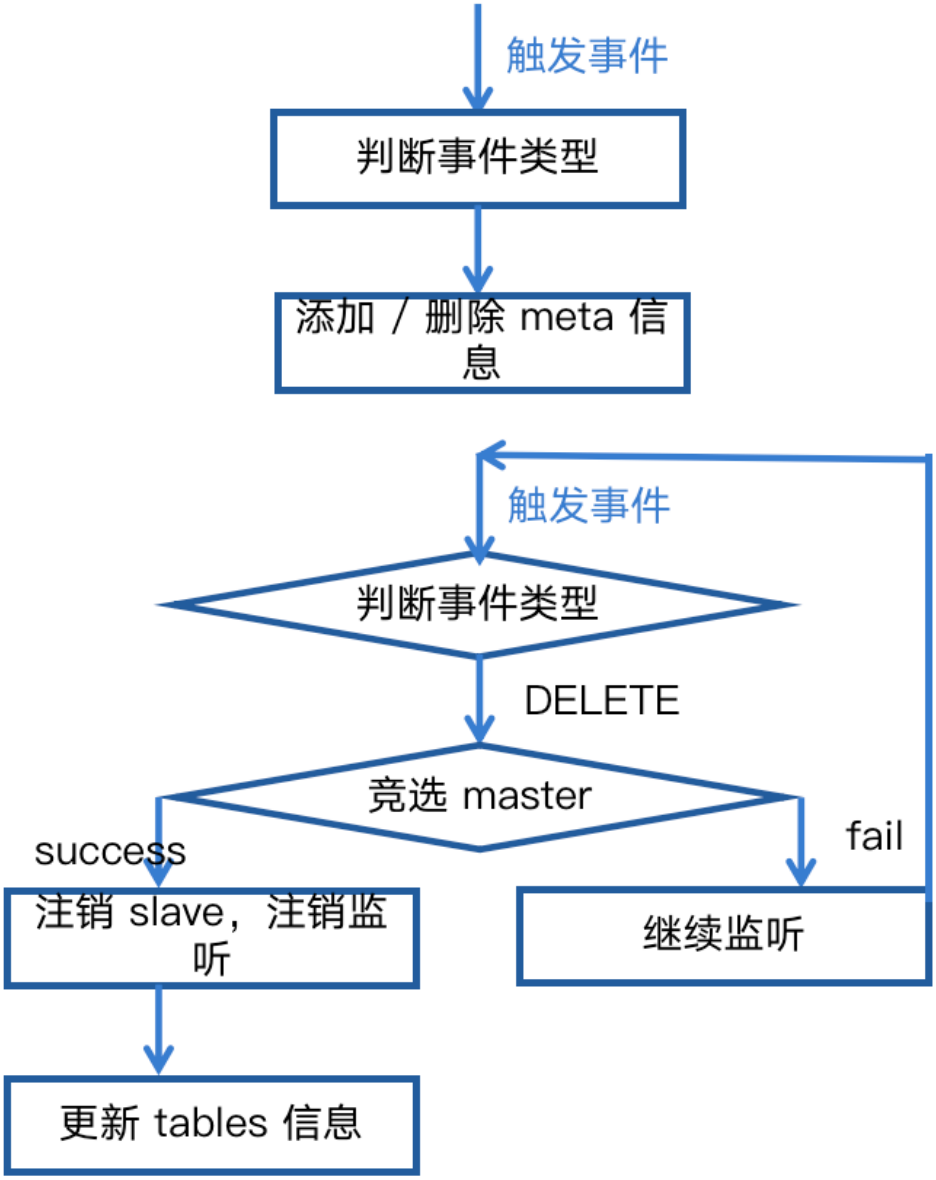
在我们小组参考了 Google File System 和 DHFS 的心跳机制，针对 Zookeeper 的特点设计了基于临时节点的心跳机制。心跳机制是一种在分布式系统中用于检测和监控节点状态的通信机制。它通过定期发送心跳消息来表示节点的存活状态，并及时检测和响应节点故障。在我们的分布式数据库中利用了 ZooKeeper 中临时节点会在会话结束时删除的特性，监测所有 Zookeeper 的活动状态。Master 可以通过监听临时节点的变化情况来完成对各 Region Server 可用状态的监控。



故障监测与恢复

在分布式数据库运行过程中，难免会遇到故障。对此我们设计了故障监测与恢复流程。首先，当一个 region 中某台服务器出现

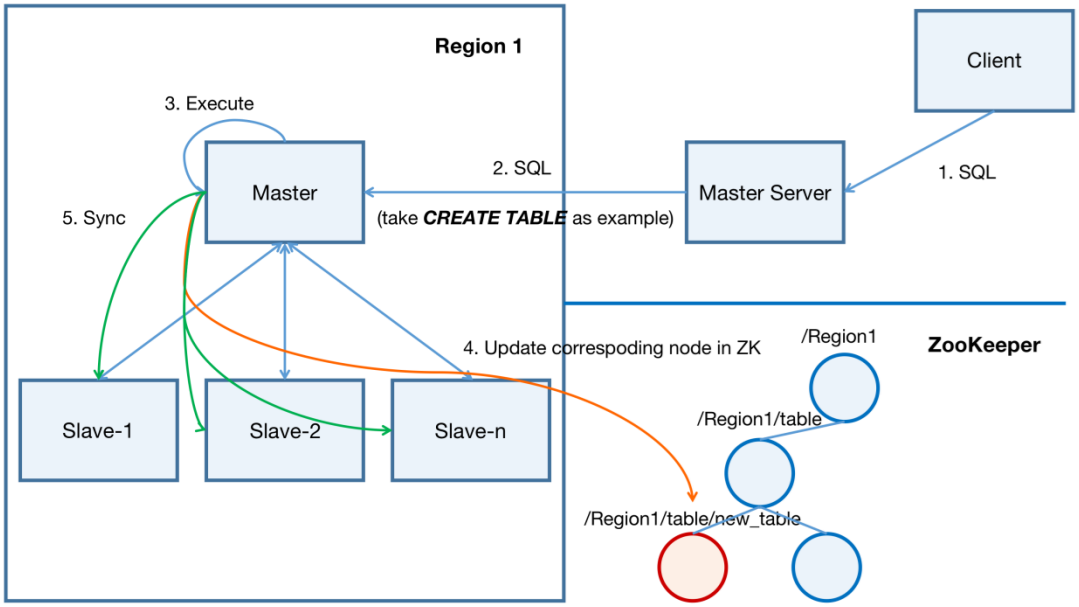
故障时，Zookeeper 会自动将其临时节点删除，其他节点监听到了这个故障后，会针对性做出反应，如果出故障的节点是 Master，那么其他节点中会自动产生一个 Master 来继续完成本 Region 的工作。



4.3 副本管理

主从复制

4.2 中我们通过多活动中心实现了容错容灾。而要具体实现多活动中心，我们还需要协调多个中心之间的关系。我们参考了 Redis，使用了主从复制来协调多个中心，具体而言，我们将 Master 作为主要中心，完成数据的增删改等操作，而其他中心（称之为 Slave）从 Master 复制数据。



副本一致性

副本一致性是指分布式系统中的副本数据在多个节点之间保持一致的状态。在我们的分布式数据库中，一个 Region 中的所有服务器都保存同样的数据副本，因此我们需要确保这些副本之间的数据保持同步，即当一个副本发生更新时，其他副本也会相应地进行更新，以保持数据的一致性。对此，我们在所有 Slave

上线的时候，都适用 Mysqldump 命令从 Master 服务器复制一份完整的数据库备份到本地，然后再恢复服务；之后，Master 服务器对数据库的所有更改都将同步通知到所有 Slave。经过上述两个步骤，我们可以保证副本的数据一致性和同步。

数据完整性校验

为了检验表信息是否正确，在本项目中我们引入了 Cyclic Redundancy Check 32（CRC32）机制对数据库表的信息进行编码。具体实现方法是对数据库表中的每一条记录（Record）的每一项属性（Attribute）分别求其 CRC32 值，并进行累加，最终得到一个针对全表的校验和。如此计算的目的是为了保证在表的内容相同的情况下，校验和不会因行/列调换顺序而发生改变。之前介绍的投票机制就利用了 CRC32 校验。除此以外，我们将所有 table 的 CRC32 校验码保存在了 Zookeeper 节点中，在 Slave 服务器上线并从 Master 服务器备份完整的数据库之后，会对数据库内容和 Zookeeper 节点中 CRC32 校验码进行对比，确保数据完整性。

4.4 数据分布与集群管理

在本系统的设计中，每个表都对应一个 Region。每个从节点管理若干个 Region，即管理若干张表。Master 节点维护从节点和 Region 之间的映射关系，记录每个从节点分别管理哪些表。

当客户端需要访问某张表时，需要首先向 Master 询问哪个从节点管理着它所需要的表，进而再访问对应的从节点。

实际运行中存在着从节点故障的情况，此时就要求 Master 执行容错容灾策略，并即时更新从节点与 Region 的映射关系。

本系统使用 ZooKeeper 进行集群管理，它包含如下几个作用：

- ZooKeeper 维护整个集群以及内部每个节点的状态信息，用于集群的配置管理
- ZooKeeper 提供命名服务，每个从节点接入系统时，会在 ZooKeeper 的 db 目录下进行注册，并获取到自己的唯一编号，具体的编号按照注册顺序依次递增。
- ZooKeeper 监控各个节点的健康状态，可以有效监测到节点增加、失效、恢复等事件，及时做出相应处理。

4.5 分布式查询

在本系统中，我们使用了客户端缓存来提高查询效率和 I/O 性能。客户端缓存通过 HashMap 来实现，定期更新和清除缓存内

容。客户端在把 SQL 语句发送给服务器之前，会先对输入的 SQL 语句进行简单的解析，提取出要处理的表名或索引名。客户端会先在本地缓存中进行查询，如果找到了对应的 Region 的地址，就直接和该 Region 建立连接，而不需要再向 Master 询问。如果没有找到，客户端就先和 Master 建立连接，向其询问并获取对应的 Region 地址，然后再连接该 Region。

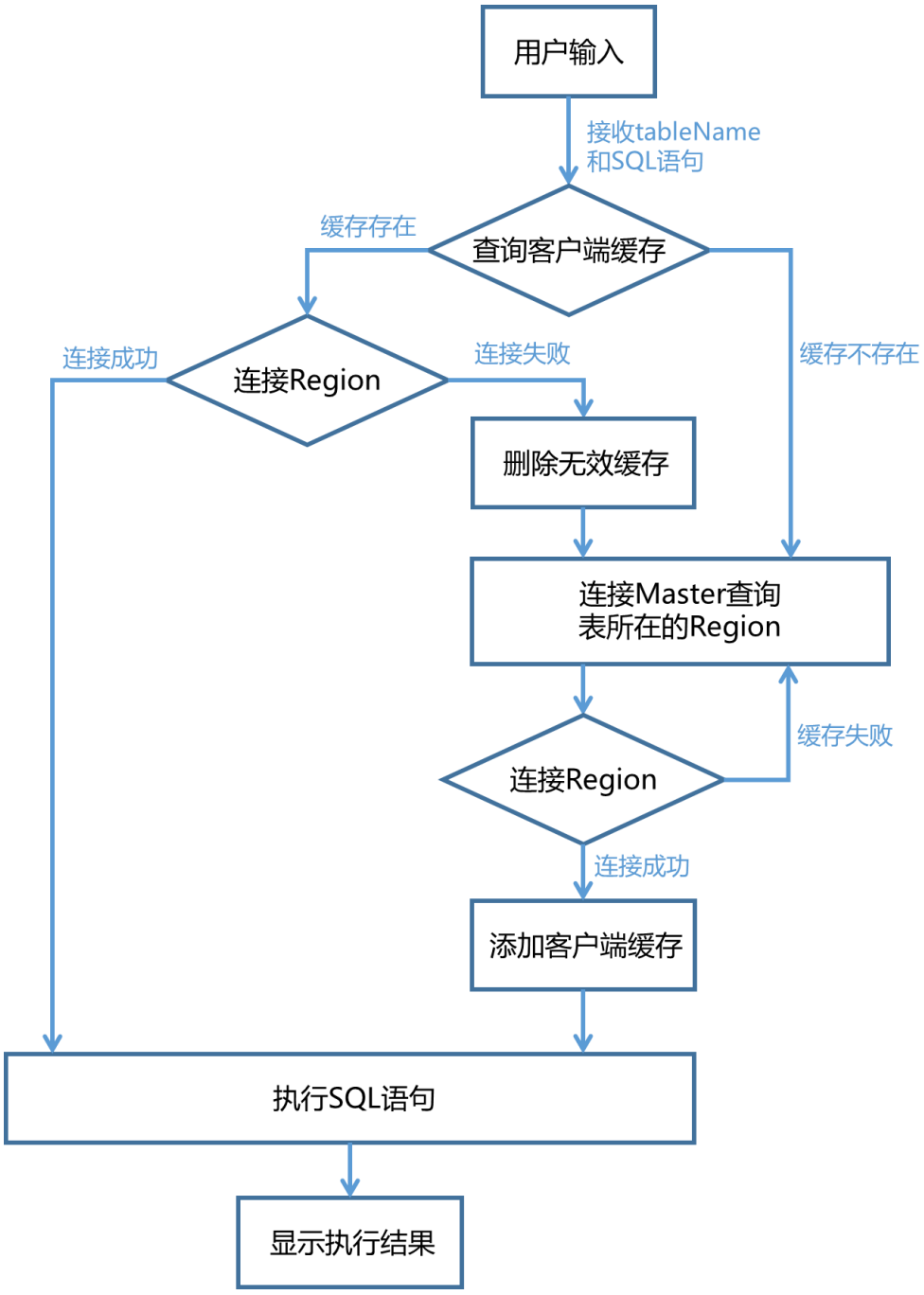
在客户端缓存的基础上，我们实现了分布式查询——客户端在执行 SQL 语句时，不再完全依赖 Master 提供的元信息，而是使用一套分布式缓存机制将缓存分布到各个客户端上。这种机制使得客户端能拥有一定的独立性和自主性：客户端可以直接与 Region 建立连接并进行通信，而不完全依赖于 Master。

这种基于客户端缓存的分布式查询有三大优点：

- 拥有更高的查询效率
- 减少客户端对 Master 的依赖
- 一定程度上提高了容错容灾能力

4.6 前后端接口设计

客户端缓存设计



Master 接口

获取 Meta 信息

返回所有 Region 上的 Table Name 及其状态（是否可写）			
方法：	GET	路径：	/meta
返回样例	{ "data":[{"name":"passwords","writable":true}, {"name":"users","writable":true}, {"name":"t2-1","writable":false}] }		

Create Table

创建 Table			
方法：	GET	路径：	/create
Query 参数：	tableName		

返回样例	# 正常返回
	{
	"status": 200,
	"addr": "127.0.0.1:8080"
	}
	# 异常返回
	{
	"status": 204,		

	"msg": "报错信息" }

Select

获取指定表格信息			
方法：	GET	路径：	/read
Query 参数：	tableName		

返回样例	# 正常返回
	{
	"status": 200,
	"addr": "127.0.0.1:8080"
	}
	
	# 异常返回
	{
	"status": 204,
	"msg": "表 XXX 不存在"
	}		

Write

Write 操作包括：建立索引、INSERT、UPDATE、DELETE、DROP			
方法：	GET	路径：	/write
Query 参数：	tableName		

返回样例	# 正常返回
	{
	"status": 200,
	"addr": "127.0.0.1:8080"
	}
	
	# 异常返回
	{
	"status": 204,
	"msg": "表 XXX 不存在"
	}		

RegionServer 接口

Create Table

需要往 zookeeper 里插入数据，所以单列			
方法：	POST	路径：	/new
Query 参数：	tableName、完整 SQL 语句		
返回样例	# 正常返回		

	{
	"status": 200
	}
	
	# 异常返回
	{
	"status": 204,
	"msg": "SQL 执行失败"
	}

DROP Table

需要往 zookeeper 里插入数据，所以单列			
方法：	POST	路径：	/drop
Query 参数：	tableName、完整 SQL 语句		

返回样例	# 正常返回
	{
	"status": 200
	}
	
	# 异常返回		

	{
	"status": 204,
	"msg": "SQL 执行失败"
	}

Select 接口

需要返回查询结果，单列			
方法：	POST	路径：	/select
Query 参数：	完整 SQL 语句		

返回样例	# 正常返回
	{
	// 列名信息
	"meta": [
	"username",
	"id",
	"password"
	],
	// 查询结果 - 每一条记录都是一个 record
	"data": [
	{
	"username": "root",		

	"id": "1",
	"password": "123456"
	},
	{
	"username": "admin",
	"id": "2",
	"password": "123"
	},
	],
	"status": 200
	}
	
	# 异常返回
	{
	"status": 204,
	"msg": "SQL 执行失败"
	}

其他 SQL 语句

只返回执行 成功/失败			
方法：	POST	路径：	/execute
Query 参数：	tableName、完整 SQL 语句		

返回样例	# 正常返回
	{
	"status": 200
	}
	
	# 异常返回
	{
	"status": 204,
	"msg": "SQL 执行失败"
	}

投票

只返回执行 成功/失败			
方法：	POST	路径：	/vote
Query 参数：	完整 SQL 语句		

返回样例	# 正常返回
	{
	"status": 200,
	“data”: CRC 校验码
	}
	
	# 异常返回		

	{
	"status": 204,
	"msg": "SQL 执行失败"
	}

从远端同步数据

只返回执行 成功/失败			
方法：	POST	路径：	/dump
Query 参数：	目标 IP 地址、tableName		

返回样例	# 正常返回
	{
	"status": 200
	}		

访问计数

只返回执行 成功/失败			
方法：	GET	路径：	/count
Query 参数：	（无）		
返回样例	自上一次技术请求后本服务器的总访问量		

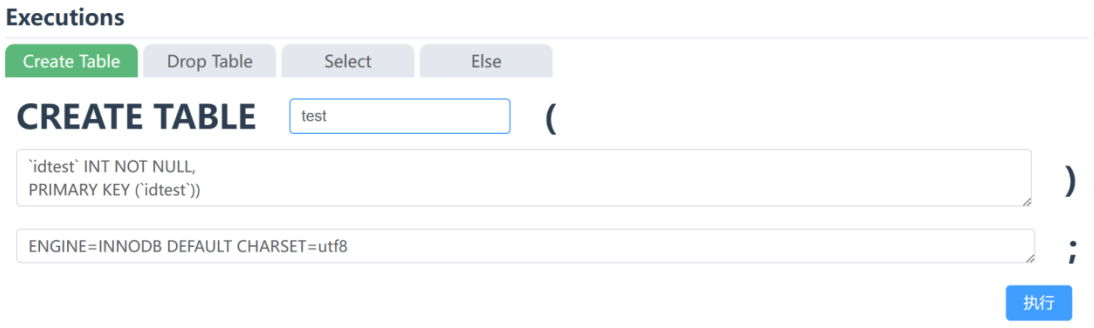
5 系统测试

5.1 Client 可用性测试

Meta 信息获取

页面挂载后，自动向 Master 请求 Meta 并渲染

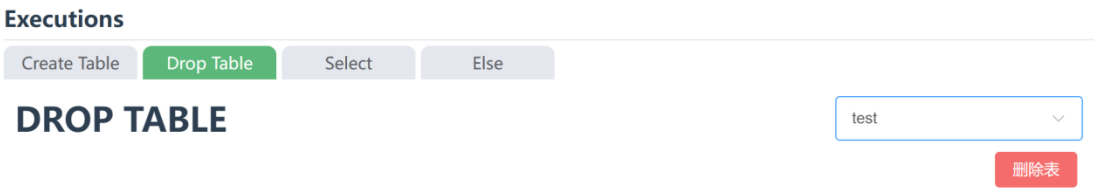
Create Table 操作与 Meta 自动更新



发送 create table（test 表）请求

创建成功，可用列表中添加 test 项

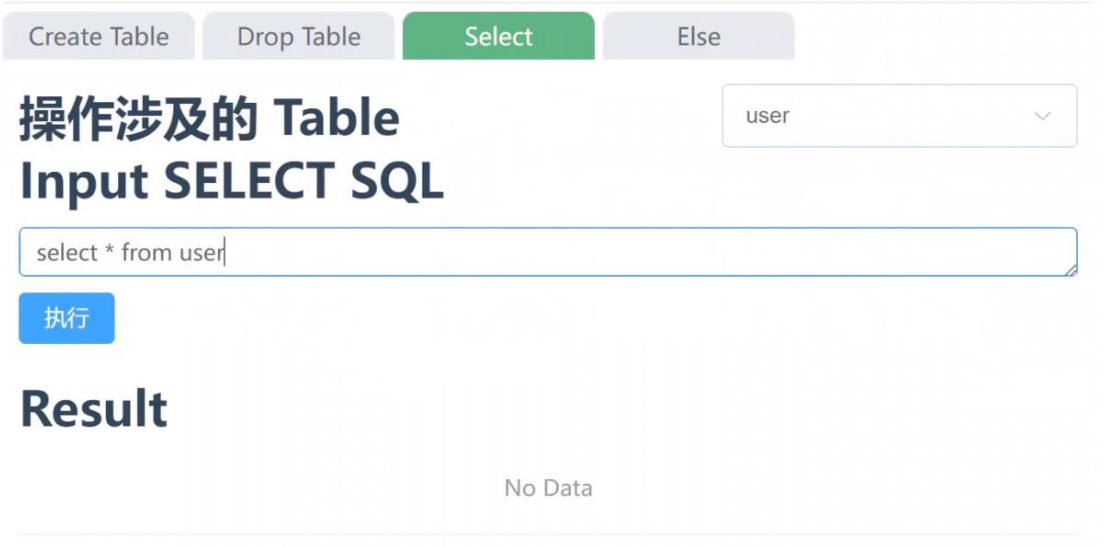
Drop Table 操作



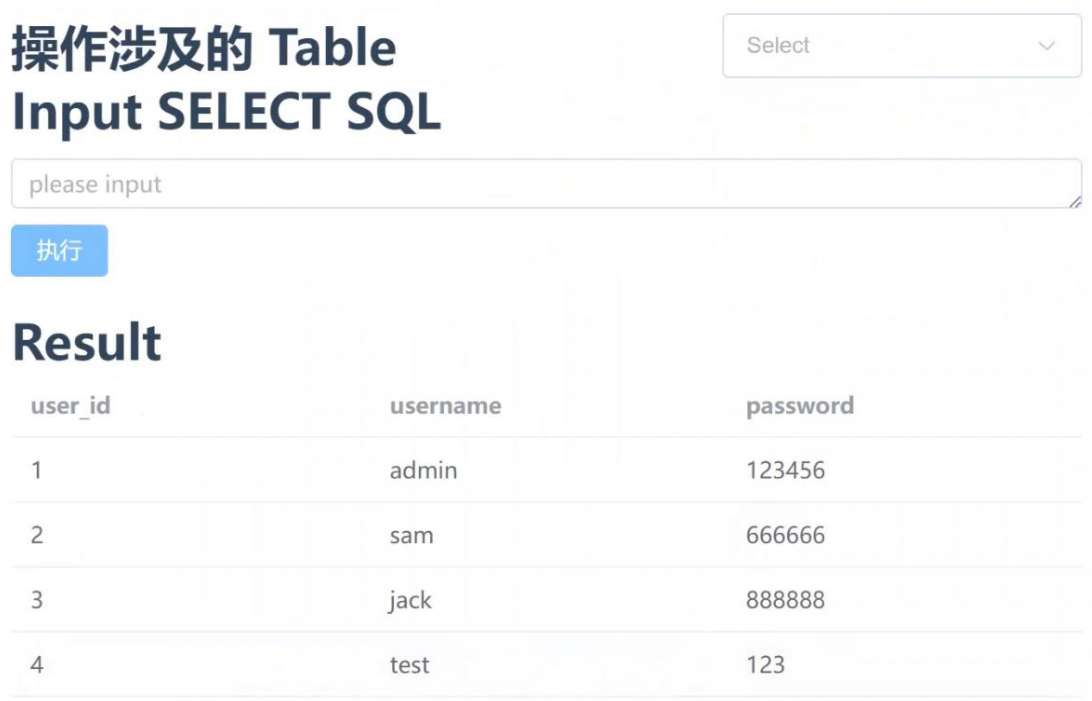
发起删除 test 表请求

删除成功，可用列表移除 test 项

Select 操作

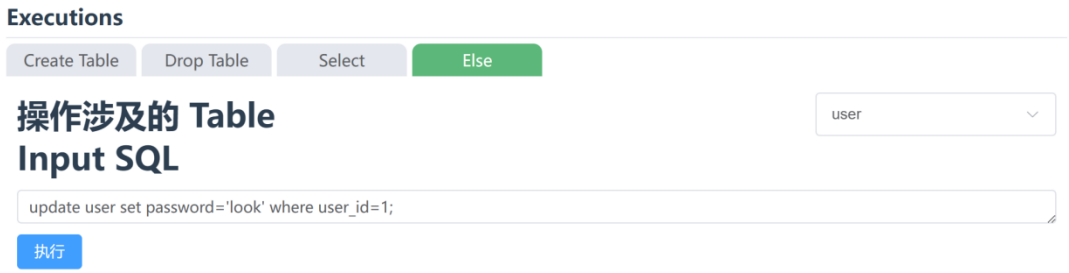


查询表 user 中的全部内容

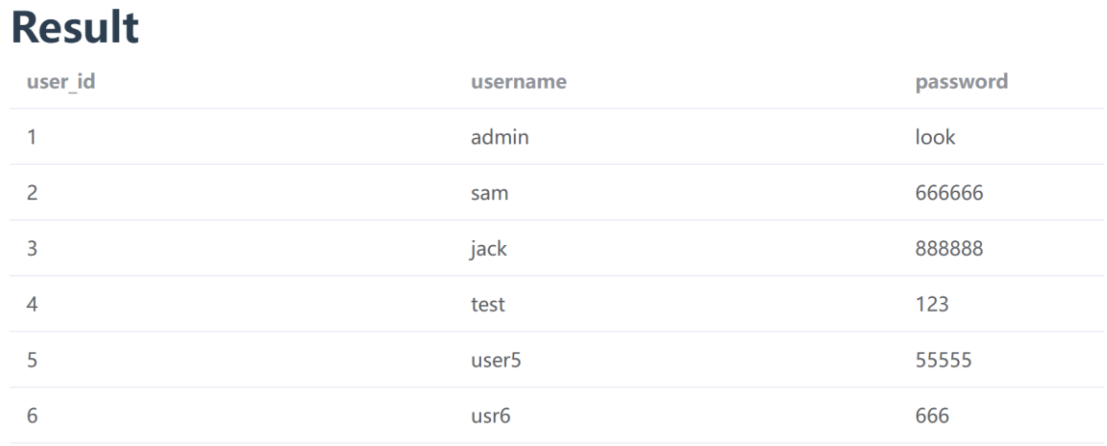


呈现查询 user 表得到的结果

Update 操作

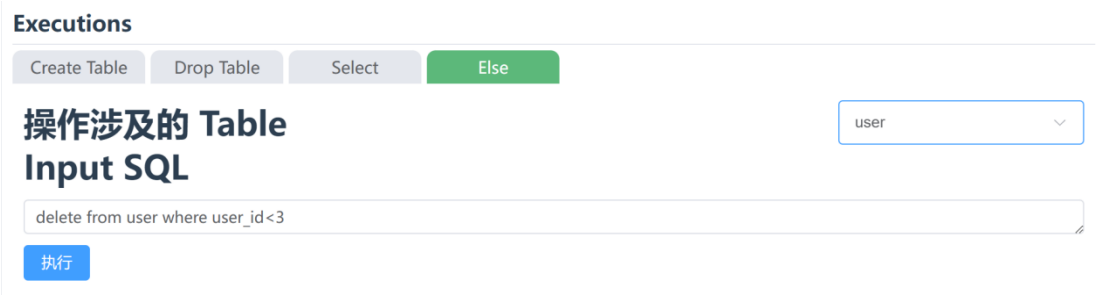


将 user_id=1 的用户密码更新为 look

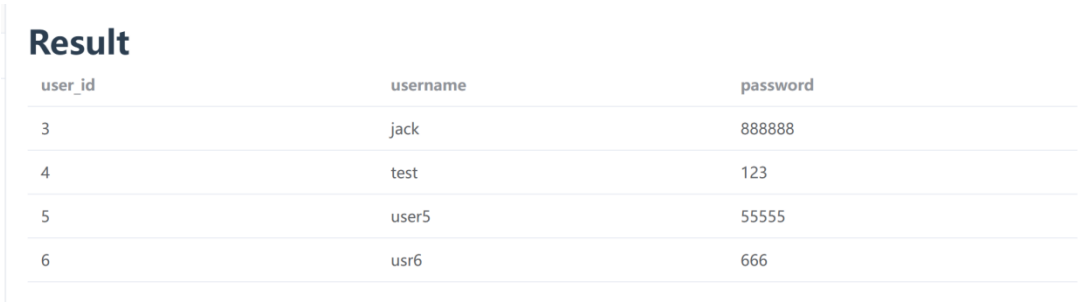


user_id=1 的用户密码被正确更新为 look

Delete 操作

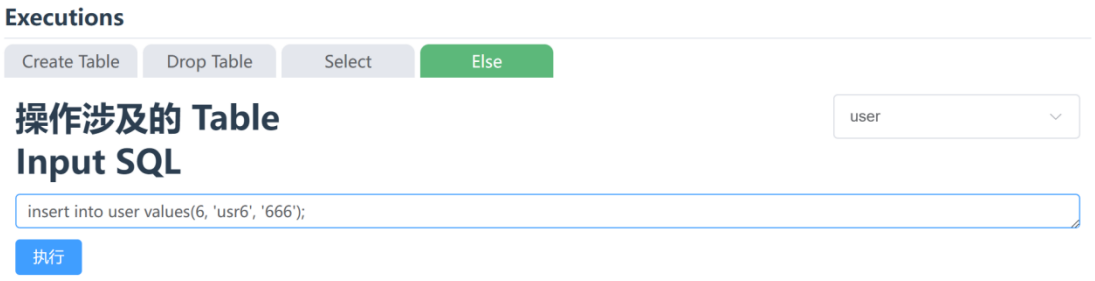


删除 user_id 小于 3 的记录

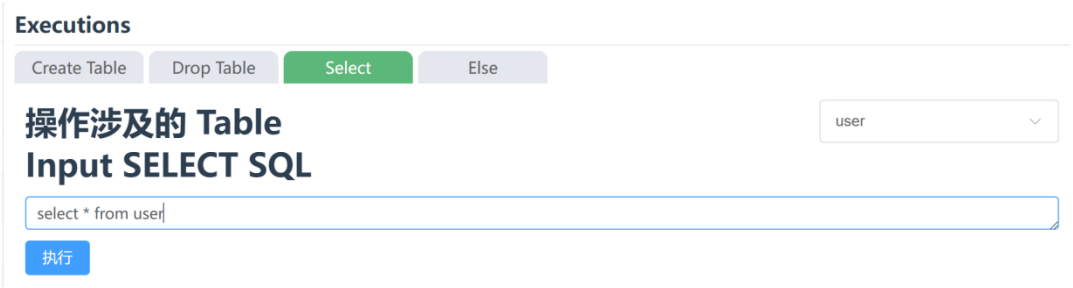


仅剩 user_id = [3,6] 的记录

Insert 操作



向 user 表插入 user_id=6 的新纪录



从 user 表中读取所有记录

Result

user_id	username	password
1	admin	123456
2	sam	666666
3	jack	888888
4	test	123
5	user5	55555
6	usr6	666

已成功添加 user_id=6 的记录

5.2 Master 可用性测试

ZooKeeper 连接与路径初始化

```
Master & Zookeeper are @10.181.215.240
Master listening Port: 9090
Trying to connect Zk Sever @10.181.215.240:2181
2023-05-18T04:17:00.769+08:00  INFO 12948 --- [           main] o.a.c.f.imps.CuratorFrameworkImpl
2023-05-18T04:17:00.775+08:00  INFO 12948 --- [           main] org.apache.zookeeper.ZooKeeper
2023-05-18T04:17:00.775+08:00  INFO 12948 --- [           main] org.apache.zookeeper.ZooKeeper
```

成功获取本机局域网 IP

```
initZk
2023-05-18T04:17:00.813+08:00  INFO 12948 --- [1.215.240:2181)] org.apache.zookeeper.ClientCnxn
2023-05-18T04:17:00.818+08:00  INFO 12948 --- [ain-EventThread] o.a.c.f.state.ConnectionStateManager
2023-05-18T04:17:00.827+08:00  INFO 12948 --- [ain-EventThread] o.a.c.framework.imps.EnsembleTracker
2023-05-18T04:17:00.827+08:00  INFO 12948 --- [ain-EventThread] o.a.c.framework.imps.EnsembleTracker
data @ /test = hello!
if you can see the return value, then you're successfully connected.
```

成功连接 ZooKeeper 、完成路径初始化并读取 /test 节点值

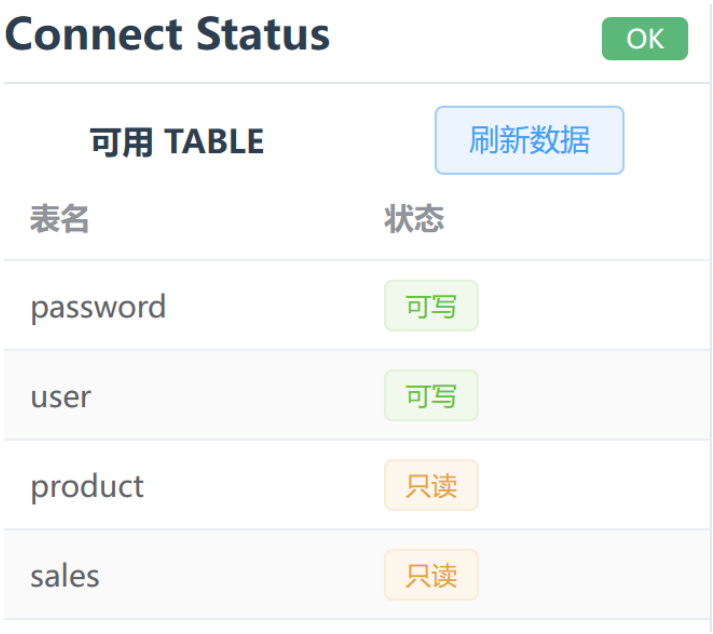
节点信息监听与 Meta 信息更新

```
/region1's MASTER is @10.181.215.240:9091
new TABLE for /region1 :password
new TABLE for /region2 :product
new TABLE for /region1 :user
new TABLE for /region2 :sales
new SLAVE for /region2 @10.181.215.240:9091, 1 SLAVES available now
2023-05-18T04:25:12.856+08:00  INFO 16536 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
```

启动后自动获取可用的 region Server 与 table 信息

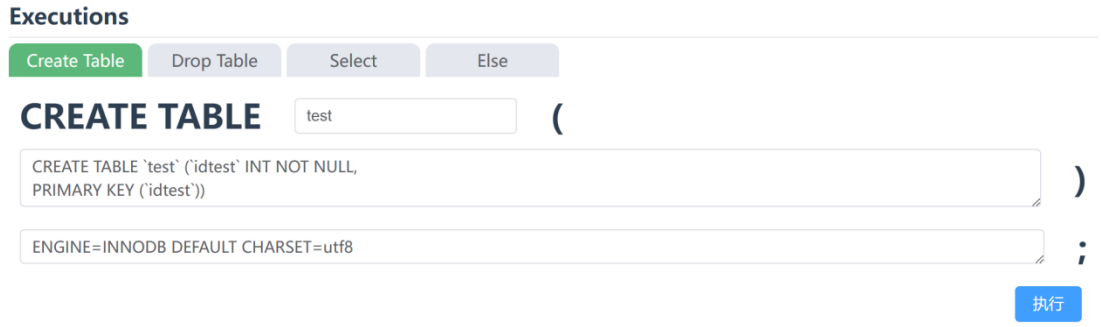
```
2023-05-18T04:29:54.679+08:00  INFO 10068 --- [           main] main.MasterApplication
region1 - 0 times
region2 - 0 times
new SLAVE for /region1 @10.181.215.240:9091, 1 SLAVES available now
new SLAVE for /region1 @10.181.215.240:9091, 2 SLAVES available now
Checking HOT POINT...
```

监听到 Slave 节点添加事件



Region 1 状态转变为可写

用户请求路由



尝试创建新表 test

Master 收到请求，并路由至 Region1-Master

同步更新 Meta 信息

热点发现与数据表跨 Region 迁移

```
Checking HOT POINT...
region1 - 9 times
region2 - 0 times
HOST POINT is region[1], 对以下 table 进行迁移
 password test
Checking HOT POINT...
region1 - 0 times
region2 - 0 times
```

Master 通过定时任务发现热点并迁移

5.3 Region Server 可用性测试

自定义路径 ZooKeeper 连接

```
Please input zkServer IP: 10.181.215.240
Current Server is @10.181.215.240:9091
Trying to connect Zk Sever @10.181.215.240:2181
2023-05-18T04:21:12.823+08:00  INFO 12716 --- [           main] o.a.c.f.imps.CuratorFrameworkImpl        : Starti
```

指定 Zookeeper Serve 地址并尝试连接

```
2023-05-18T04:21:12.865+08:00  INFO 12716 --- [ain-EventThread] o.a.c.framework.imps.EnsembleTracker      : New co
data @ /test = hello!
if you can see the return value, then you're successfully connected.
```

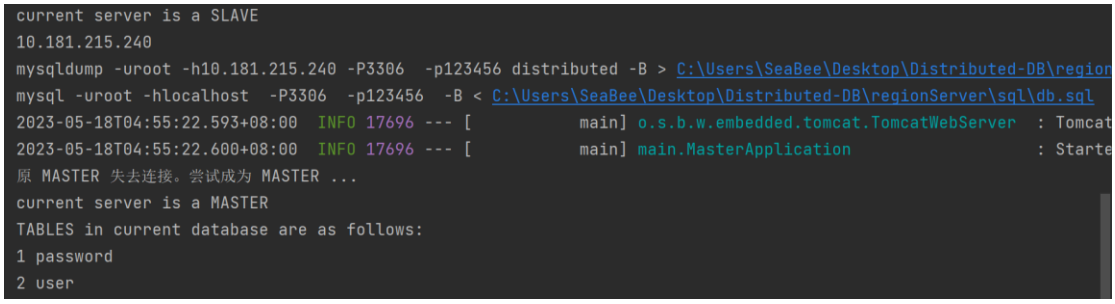
成功连接并读取测试数据 /test

beMaster() 测试

```
current server is a MASTER
TABLES in current database are as follows:
1 password
2 user
```

当选 Master，获取 table 列表信息

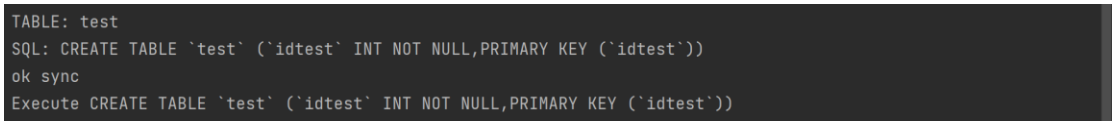
beSlave() 测试



Master 已存在，从远端同步数据库

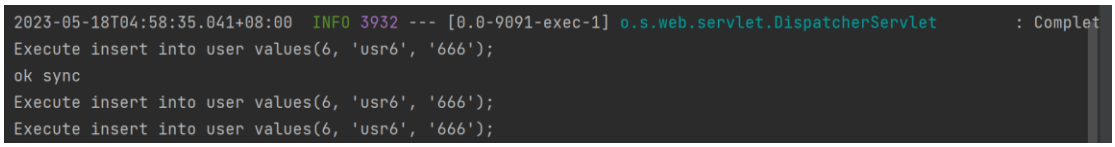
Master 已失效，竞选成功并修改 meta 数据

Create Table 与节点更新



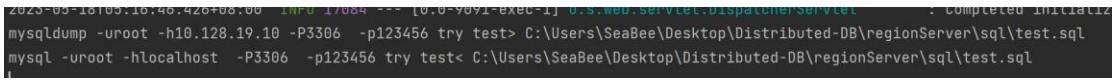
作为 Master 接受创建表请求，并同步至 slave

Execute 与信息同步



Master 接受 inset 请求，并同步至 slave

远程 dump 数据表



尝试从 10.128.19.10 dump 数据表 test 到本地

```
-- MySQL dump 10.13  Distrib 8.0.33, for Win64 (x86_64)
--
-- Host: 10.128.19.10    Database: distributed
-- ------------------------------------------------------
-- Server version    8.0.33

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!50503 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;
```

生成的 test.sql 部分内容

执行结果投票

```
2023-05-18T05:22:27.269+08:00  INFO 12308 --- [0.0-9091-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
连接数据库...
连接数据库...
连接数据库...
得票数：2/2
投票通过
```

执行 select 语句时，对多个 region server 返回结果的 CRC 校验码进行比对投票

6 总结

在开发大规模分布式数据库项目的过程中，我们经历了许多挑战和收获。首先，团队合作起到了关键的作用。每个成员都充分参与到了项目中，密切合作并共同解决问题。通过有效的沟通和协作，我们能够克服各种困难，确保项目按时完成。

同时，在项目进行过程中，我们主动地去学习了很多知识，我们阅读了 Google File System 的论文，我们了解了 DHFS 的独

特机制，我们学习了 CePH 对于冗余和备份的处理。我们还主动思考了数据完整性的问题。这些经验都是我们项目成功的关键。

通过这个大规模分布式数据库项目的开发，我们不仅学到了许多技术知识和经验，还培养了团队合作和解决问题的能力。我们为能够成功交付一个高性能、可扩展和可靠的分布式 MisniSQL 系统感到自豪，并期待在未来的项目中应用和进一步发展我们所学到的知识。