

大规模信息系统构建技术导论

分布式数据库系统设计报告

2023学年 春学期

组员信息

|  |  |
| --- | --- |
| 学号 | 姓名 |
| 321010 | 叶沐阳 |
| 321010 | 俞博 |
| 321010 | 郭一铭 |

2024年 5月 27日

目录

[大规模信息系统构建技术导论 1](#_Toc1745)

[分布式数据库系统设计报告 1](#_Toc26100)

[组员信息 1](#_Toc5543)

[1 引言 3](#_Toc27305)

[1.1 系统目标 3](#_Toc7423)

[1.2 设计说明和任务分工 4](#_Toc6359)

[2 总体设计 4](#_Toc14326)

[2.1 总体架构设计 4](#_Toc282)

[2.2 Client模块 5](#_Toc22232)

[2.2.1 架构设计 5](#_Toc5130)

[2.2.2 工作流程 6](#_Toc27486)

[2.3 Master模块 6](#_Toc6679)

[2.3.1 架构设计 6](#_Toc2946)

[2.3.2 工作流程 9](#_Toc11117)

[2.4 Region模块 11](#_Toc26710)

[2.4.1 架构设计 11](#_Toc12300)

[2.4.2 工作流程 12](#_Toc8265)

[2.5 util文件 12](#_Toc29034)

[3 分布式设计 13](#_Toc28846)

[3.1 通信架构和通信协议 13](#_Toc21084)

[3.1.1 通信架构 13](#_Toc10387)

[3.1.2 通信协议 13](#_Toc10012)

[3.2 缓存机制 13](#_Toc3872)

[3.3 数据分布 14](#_Toc13989)

[3.4 均衡负载 14](#_Toc16918)

[3.5 副本管理和容错容灾 14](#_Toc30647)

[3.6 集群管理 15](#_Toc29993)

[4 系统测试 15](#_Toc28486)

# 1 引言

本项目是《大规模信息系统构建技术导论》的课程项目，基于《数据库系统》课程的基本知识和本课程所学的大规模信息技术系统构建的相关知识进行开发，完成一个分布式大规模数据库系统。

本系统通过设计 Client、Master、Region模块，使用etcd作为集群管理工具，使用sqlite作为数据库，实现了一个在多台主机上共享数据资源访问的分布式数据库系统。

本系统使用go语言开发，使用GitHub进行版本管理和协作开发，由小组内的三名成员共同完成。

* 1. 系统目标

本次实验项目是一个分布式大规模数据库系统。

项目实现了以下具体功能：

1. 数据库查询：支持基本的 SQL 语句查询，包括数据库数据的插入、查询、修改、删除等数据库的基本功能。同时可以实现部分复杂的查询。
2. 数据分布：将完整数据表存储在服务器上，分割成若干个数据子集。一个Region管理若干table,Master记录Region与其管理的table的对应关系。
3. 集群管理：使用etcd作为集群管理工具。etcd是采用了 raft 分布式一致性算法的分布式键值对存储系统（key-value store）
4. 分布式查询：client维护两个缓存表以供查询对应table和index所在的region；同时Master也维护表，存储所有table和index与Region的关系，Region处理增删改查操作。
5. 副本管理：采用主从复制策略，两个region为一组，其中一个region作为主region，另一个为备份region。
6. 负载均衡：Master 可以对其他 Region进行管理。Master 内存中存储了分布式数据库所有表的信息（名称）及它们的位置，以及所有用户创建的索引信息（名称）；数据表与索引的数据存储于 Region 服务器中。用户有建表需求时，Master 进程将选择存储的表数量最少的 Region 主服务器作为新表的存储地。当某一Region出现繁忙时，Master通过设计好的负载均衡算法将某些 Region重新分配到其他的 Region上。同时各Region可以实现数据传输，等待负载均衡之后修改 table的定位信息。
7. 容错容灾：当某个Region 失效后，Master 可以将从region的信息升级为主region并创建新的备份。

## 1.2 设计说明和任务分工

本项目由小组3位成员合作编写完成，采用 go程序设计语言，分别使用 IDEA、vscode、goland作为主要开发工具，使用 etcd作为集群管理工具，支持在不同的操作系统与语言环境下跨平台使用。

各个成员的分工如下：

|  |  |  |
| --- | --- | --- |
| 学号 | 姓名 | 分工职责 |
| 321010 | 叶沐阳 |  |
| 321010 | 俞博 |  |
| 321010 | 郭一铭 |  |

# 2 总体设计

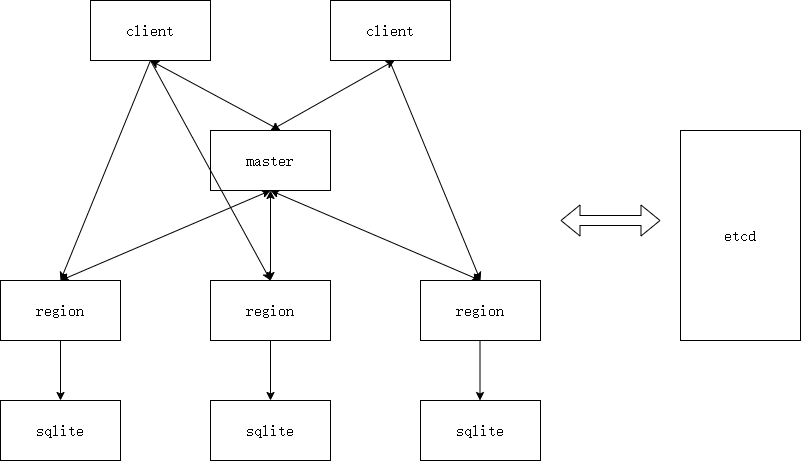
## 2.1 总体架构设计

系统将整个项目划分成三个模块，分别Client，Master 和 Region。

Client模块是用户与数据库交互的接口；Master模块管理Region模块，实现系统的集群管理、负载均衡、副本管理、容错容灾、数据分布、分布式查询等功能；Region负责数据库引擎的运行并且服从master的调度安排。

三者都可以在一定的通信框架下进行通信，同时通过etcd集群。

系统总体架构示意图如图1所示：



**图1 总体架构图**

## 2.2 Client模块

### 2.2.1 架构设计

client向上负责与用户进行直接交互，向下与region和master进行连接。其主要的功能包括：预处理输入的sql语句，根据解析调用master或者region中接口进行执行语句，维护缓存表来存储对应table所在的region，提供缓存服务。

* 预处理输入：

对用户输入的sql语句进行解析，获取需要操作的table和index。

函数签名为func (client \*Client) parse\_sql\_statement(input string)

输入：sql语句

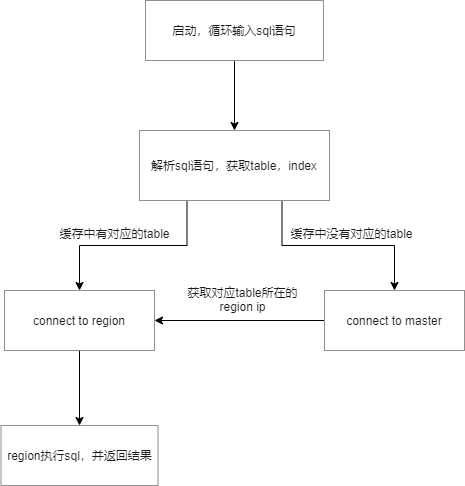
* 调用接口：

根据parse\_sql\_statement维护的缓存表中是否有缓存选择连接master或者region。没有缓存的情况下，如果语句为create、show、drop调用master接口，由master调用region接口；其他情况调用master得到table所在的region，然后调用对应的region接口。

* 缓存机制：

函数签名为func (client \*Client) updateCache(table string) string。为了加快大规模频繁操作时的速度，维护两个map，分别为table-IP表和IP-rpcRegion表，前一个表存储某一个表在哪个IP地址的机器上，后一个表表示IP地址及对应的rpcClient，用于直接和region进行信息的交互。前者可能为多对一，后者必为一对一。若某些IP地址对应的机器挂掉，表中的旧IP不会马上清除，而是通过将来可能有的二次查询而检测到并删除。

### 2.2.2 工作流程



**图2 Client架构图**

## 2.3 Master模块

### 2.3.1 架构设计

Master的设计如图3所示:

**图3 Master架构图**

1. master的主要变量

type Master struct中定义了如下变量和缓存表：

* RegionCount int：记录了region主服务器的数量；
* EtcdClient \*clientv3.Client：为 master 进程与 master 进程所在的服务器（即 master 服务器）上的 etcd 进程通信的媒介；
* RegionClients map[string]\*rpc.Client：为 master 进程与所有在线的 region 服务器上 region 进程通信的媒介。
* Owntablelist map[string]\*[]string和TableIP map[string]string：记录了table与其所在region的ip 的对应关系；
* IndexInfo map[string]\*[]string和TableIndex map[string]string：记录了index与其所在table 的对应关系；
* BusyOperationNum map[string]int ：记录了region的忙碌状态；
* RegionIPList []string：记录了所有连接过的region的 ip；
* Backup map[string]string：记录了region和其备份region的对应关系；
* Available string：记录当前活跃的region的 ip。

1. master的主要函数

Master.go中函数：

* func (master \*Master) Init(mode string) ：master系统的初始化，包括对数据库文件的读取，对变量和缓存表的初始化和数据载入；
* func (master \*Master) Run() ：初始化etcd集群并开启rpc服务进程；
* func (master \*Master) GetTableIP(table string, reply \*string) error：用于查询table和region的对应关系；
* func (master \*Master) InitTableIP() ：同步本地数据库文件中的table信息；
* func (master \*Master) InitIndex(table string) ：同步本地数据库文件中的index信息；

Func.go中函数：

* func (m \*Master) TableShow(arg string, reply \*string) error：直接查看owntablelsit查询所有region的table，以表格形式返回所有table以及其所属regionip；
* func (master \*Master) TableCreate(input string, reply \*string) error ：在数据库中创建table；
* func (master \*Master) TableDrop(input string, reply \*string) error：在数据库中删除table；
* func (master \*Master) deleteTable(table, ip string) ：删除缓存表中的table信息；
* func (master \*Master) check\_and\_reset\_Regions() error ：检查和重置region；
* func extractTable(s string) string ：提取table名称；
* func (master \*Master) IndexShow(arg string, reply \*string) error：查询所有table的index，以表格形式返回所有index以及其所属table；
* func (master \*Master) IndexCreate(input string, reply \*string) error ：在数据库中创建index；
* func (master \*Master) IndexDrop(input string, reply \*string) error：在数据库中删除index；
* func (master \*Master) deleteIndex(index string, table string) ：删除缓存表中的index相关信息；

Etcd.go中函数：

* func (master \*Master) assignment(available\_list []string)：记录了etcd的相关代码，监视在util中的region\_ip的etcd变化；
* func (master \*Master) getAvailableRegions()[]string：获取当前活跃的region；
* func (master \*Master) addRegion (region\_ip string) ：用于初始化和后续加入region的连接；
* func (master \*Master) watch() ：进行对 etcd 键值对存储系统的各种更新（键值对的增添与删除）的监听，并进行相应的操作：
* func (master \*Master) deleteserver(IP string) ：如果有活跃的region，启动副本region;否则把副本region中的内容都转存到某个region对中；
* func (master \*Master) deletebackup(IP string) ：如果有活跃的region，将其设为副本region，否则把副本region中内容都转存到其他region对中，将副本region转换为活跃region。

### 2.3.2 工作流程

该项目中 Master 层的功能由三个子文件实现：

1. etcd.go文件。

建立与 etcd的连接，实现数据集群管理，对节点的状态改变进行监听， 基于此可以实现容错容灾等功能。

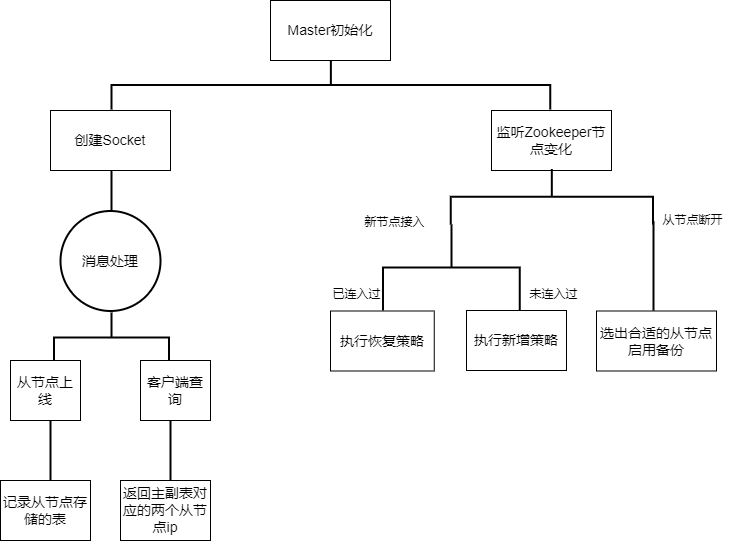
1. func.go文件。

建立与 Client 和Region的通信，通过 Client 的请求查找相关 Region 信息，并将结果返回至 Client。 Region 在增加删除表时，需要通过 rpc连接给 Master 发消息。同时，Master 在检测到有region挂了之后，需要在负载均衡后选出合适的两个region，给两个region发消息以通知它完成容错容灾。

1. master.go文件。

定义了缓存表并进行了系统的初始化，同步了数据库中的信息。

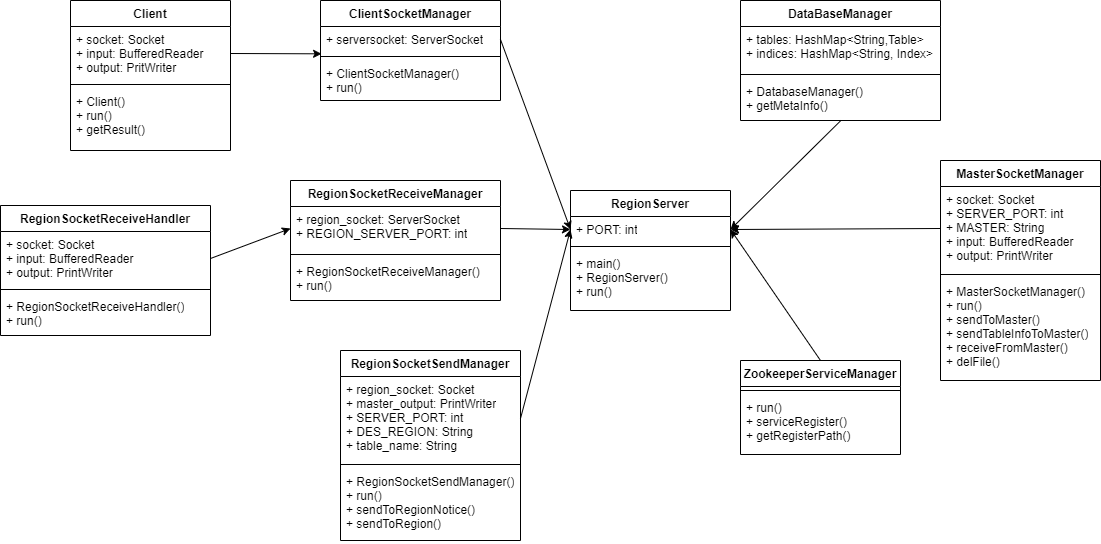
整体工作流程图如下：



**图4 Master工作流程图**

## 2.4 Region模块

### 2.4.1 架构设计



**图5 Region架构图**

相关的主要函数如下：

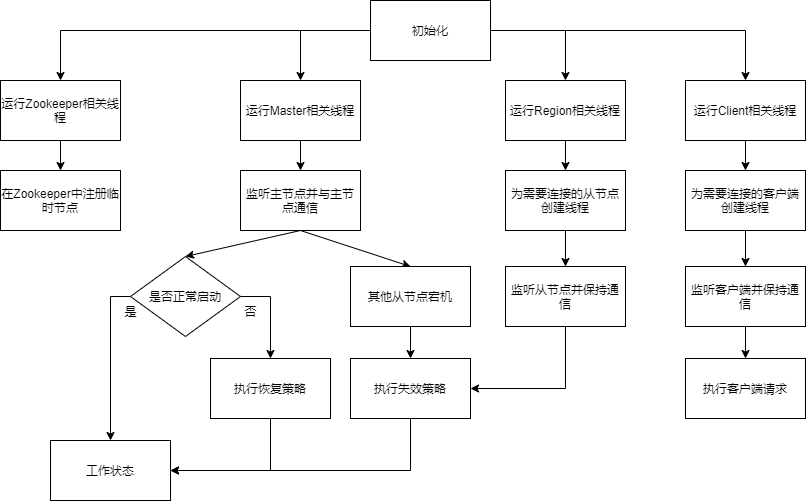
Region.go中函数：

* func (region \*Region) Init()：实现region的初始化，包括连接数据库文件和注册rpc服务；
* func (region \*Region) keepalive()：在etcd中保持活跃；
* func (region \*Region)foundhostIP()string：获取本机IP地址。

Func.go中函数：

* func (region \*Region) TableName(input string, reply \*[]string) error ：返回当前region管理的table的名字；
* func (region \*Region) Index(input string, reply \*[]string) error：返回当前region管理的index的名字；
* func (region \*Region) Execute(input string, reply \*string) error：处理非查询类的sql语句；
* func (region \*Region) Query(input string, reply \*string) error ：处理查询类的sql语句；
* func (region \*Region) AssignBackup(ip string, dummyReply \*bool) error ：给主region分配副本region。

### 2.4.2 工作流程



**图6 Region工作流程图**

本项目中Region模块主要实现了以下功能：

1. 加入etcd的集群管理
2. 处理Client端和master端发送的sql语句对数据库进行增删改查

## 2.5 util文件

该模块主要包含的函数如下：

* func FindElement(pSlice \*[]string, str string) int ：查找元素；
* func TimeoutRPC(call \*rpc.Call, ms int) (\*rpc.Call, error)：反馈rpc调用是否超时；
* func DeleteFromSlice(pSlice \*[]string, str string) bool ：删除字符串组中字符串；
* func AddToSlice(ptr \*[]string, newString string) & func AddToSliceIndex(ptr \*[]string, newString string) ：在字符串组中加入字符串；
* func CleanDir(localDir string) ：删除指定目录中的所有文件。

# 3 分布式设计

## 3.1 通信架构和通信协议

### 3.1.1 通信架构

远程过程调用（Remote Procedure Call，RPC）是一个计算机通信协议，该协议允许运行于一台计算机的程序调用另一台计算机的子程序。

在架构中，client与Master进行多对一的连接，Region与Master进行多对一的连接，而client与Region的连接不定（正常操作下随着SQL语句的执行应该会逐步变成一对多）。

建立rpcClient时，只需要指定IP地址和端口号即可通过DialHttp函数获取到rpcClient，随后便可使用rpcClient对region/master的函数进行调用。

### 3.1.2 通信协议

由2.5可以看到，系统封装了非阻塞的RPC调用函数TimeoutRPC，master/func.go和region/func.go中的部分函数采用了基于传入两个指针参数，一个作为封装好的函数固有参数，另一个作为返回值的策略。

## 3.2 缓存机制

为了加快大规模频繁操作时的速度，我们设置了客户端缓存机制，并且会并且会定期更新和清除缓存的内容，客户端在向服务器发送 SQL 语句之前先进行一步预处理，提取出要处理的表名或者索引名，然后在缓存表中先进行查询，如果查到了对应的Region 地址就直接和 Region 进行连接，如果没有查到就和 Master 先连接并获取Region 的地址之后再和 Region 连接。

## 3.3 数据分布

出于最小化系统复杂度的考虑，我们选择将完整的数据表存储在服务器中，而不是将数据表的所有记录分散存储在不同服务器上。

master 进程的内存中存储了分布式数据库所有表的信息（名称）及它们与region的关系，以及所有索引信息（名称）及它们与table的关系；数据表与索引的数据存储于 region 服务器中。

Master中的存储情况如下图：

## 3.4 均衡负载

在该项目中我们通过 master 的调度实现了负载均衡的设计。

在创建一张表时，总是会选择负载最少的两个region进行创建，挂掉的region的备份也会被转移到当前表最少的region上。

每次找到当前负载最轻的region作为新的region。然后master给原来的region发送消息，告诉他被新选出来的region的ip以及表的名字，使得两个region之间能够传输信息，达到均衡负载。

负载均衡相关的代码段如下图：

## 3.5 副本管理和容错容灾

本系统的副本管理功能是通过两个Region节点备份实现的。每当master检测到客户端发送的命令，对数据库的数据进行了修改，就把修改后的数据同时在两个Region上进行更新。

副本管理代码段如下图：

如果某个Region 挂了，master监测到后寻找表少的Region，向该Region 发送备份指令（包括挂了的Region 所存储的所有表）。Region 将表和索引从挂了的Region 的副本Region 上读取下来，读取完成后给master发消息，master收到后修改缓存表。

如果Region 重新连上，执行恢复策略，master向Region 发送恢复指令，Region 收到后将表全部删除，删除完成后给master发消息，master收到后将其状态变更为活跃的Region ，恢复正常使用。

容错容灾代码段如下图：

## 3.6 集群管理

本系统使用 etcd 作为集群管理工具。etcd 是采用了 raft 分布式一致性算法的分布式键值对存储系统（key-value store），我们利用 etcd 的 Grant、Put、KeepAlive、Watch 这几个 API 完成了集群节点管理的功能。

当 region 节点上线时，其将在 etcd 键值对存储系统中创建一个具有生命周期的，以该进程所在服务器 IP 地址为键的键值对，并持续维持其生命；master 节点开启后，其将持续监测键值对存储系统中的各种更新。

代码段如下图：

## 3.7 复杂sql语句的查询

# 4 系统测试

使用 docker 对系统进行测试。

在测试中，创建了一个 docker network，用于模拟实际环境中的网络，并创建了5个连接到上述创建的 docker network 的容器，用于模拟5个服务器的集群。

随机挑选一个容器开启服务端进程。

（测试过程）

数据分布：

简单数据操作

集群管理：

Etcd

master监视Watch

分布式查询：

Master中保存index和table的内容。

副本管理：

初始化分配region

添加新的region之后管理region

容错容灾：

某个server挂了

某个backup挂了

4种情况

负载均衡：

插入table优选选择少的

复杂查询实现：

Join