

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 |
| 3210104095 | 叶沐阳 |
| 3210105512 | 俞博 |
| 3210104116 | 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](#_Toc30033)

[分布式数据库系统设计报告 1](#_Toc11916)

[组员信息 1](#_Toc16467)

[1 引言 3](#_Toc6921)

[1.1 系统目标 4](#_Toc16762)

[1.2 设计说明和任务分工 6](#_Toc13919)

[2 总体设计 6](#_Toc15155)

[2.1 总体架构设计 6](#_Toc28060)

[2.2 Client模块 8](#_Toc4455)

[2.2.1 架构设计 8](#_Toc15335)

[2.2.2 工作流程 10](#_Toc26100)

[2.3 Master模块 10](#_Toc27030)

[2.3.1 架构设计 10](#_Toc7684)

[2.3.2 工作流程 20](#_Toc13486)

[2.4 Region模块 21](#_Toc16128)

[2.4.1 架构设计 21](#_Toc13959)

[2.4.2 工作流程 24](#_Toc18479)

[2.5 util文件 24](#_Toc14189)

[3 分布式设计 27](#_Toc5701)

[3.1 通信架构和通信协议 27](#_Toc26740)

[3.1.1 通信架构 27](#_Toc2157)

[3.1.2 通信协议 28](#_Toc31972)

[3.2 副本管理 28](#_Toc6796)

[3.3 数据分布 31](#_Toc10200)

[3.4 均衡负载 33](#_Toc1755)

[3.5 容错容灾 36](#_Toc26634)

[3.6 集群管理 44](#_Toc20742)

[3.7 复杂sql语句的查询 45](#_Toc28520)

[4 系统测试 50](#_Toc22886)

[4.1 大规模测试 50](#_Toc11048)

[4.2 集群管理 55](#_Toc22082)

[4.3 分布式查询 56](#_Toc20925)

[4.4 副本管理 57](#_Toc11961)

[4.5 容错容灾 58](#_Toc18923)

[4.6 负载均衡 63](#_Toc16526)

[4.7 复杂查询 65](#_Toc10221)

# 1 引言

本文档所包含的项目是《大规模信息系统构建技术导论》的课程项目，项目目标是创建一个分布式大规模数据库系统。本系统的开发基于《数据库系统》课程的基础知识与相关项目经验，加之与在本课程所教授的大规模信息技术系统构建的相关知识所结合。

我们的系统设计包括了Client、Master和Region模块，这些模块共同工作，实现了在多台主机上共享数据资源的访问。本系统集成了所有的要求：**数据分布、集群管理、分布式查询、副本管理、容错容灾、负载均衡、复杂查询实现**(Join, Cross Join)，并在容错容灾与副本管理上进行了优化设计。为了实现集群管理，我们选择了etcd，它是一个高度可靠的开源分布式键值存储系统，可以用于配置管理、服务发现和其他需要共享信息的场景。关于基础的数据库我们选择了sqlite作为我们，它是一个轻量级的、自包含的、高可靠性的嵌入式SQL数据库引擎。

在开发过程中，我们选择了Go语言，这是一种静态类型、编译型的编程语言，其语法与C语言类似，但同时具有垃圾回收功能，并且支持并发编程，这使得它能够有效地利用多核处理器的计算能力。相比Java，Go具备了开发速度快、轻量级，更快的运行速度与编译速度等多项优点。我们使用GitHub进行版本管理和协作开发，这是一个面向开源及私有软件项目的托管平台，它为我们提供了方便的代码版本控制和协作开发工具。

这个项目由我们小组的三名成员在合理分工与高效沟通的情况下共同完成。通过这个项目，我们不仅得以将所学的理论知识应用到实践中，还有机会提升我们的团队协作能力和项目管理能力。这对我们未来的学术研究或职业生涯都将是非常有益的。此外，我们也对大规模信息系统的构建有了更深入的理解，这将对我们未来多式多样的领域与复杂的大规模的应用场景下的研究和工作产生积极影响。相信本课程设计为我们在AI时代的来临下处理海量数据的能力做出了一个不错的开始。

* 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。同时可以支持复杂的查询，如join, cross join等。
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。对用户而言，整个分布式系统应当是“隐藏”的，即用户仅能感受到与一台数据库交互，而非与一整个分布式系统进行交互。
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）。Etcd将自动检测集群的节点变化，在节点断联与新增节点的情况下由提供的hook函数来进行相应的操作，保证所有的数据完整性。
4. **分布式查询**：Master维护缓存表以供查询对应table和index所在的region，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。本项目规定实际投入运行的region数量必须是偶数个，保证所有投入运行的主节点都有一个备份节点来实现容错容灾。如果节点数量是奇数个，剩余的一个服务器转为Available(待用节点)，等待多余的节点进入系统形成新的主从对。
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。具体的负载均衡体现在两个方面:1. 用户有建表需求时，Master 进程将选择存储的表数量最少且不处于繁忙状态的 Region 主服务器作为新表的存储地。2. 当某一Region出现繁忙时，Master通过BestFit的负载均衡算法将繁忙Region的一半数据表逐一重新分配到其他的 Region上。同时各Region之间可以实现数据传输，等待负载均衡之后修改 table的定位信息。此负载均衡设计为本系统提供了总体的压力平衡，将局部的高峰操作平摊于整体。
7. **容错容灾**：主要处理各个服务器意外下线情况。通过主从服务器结构，来处理主服务器、副本服务器和Master服务器意外断联等情况，保证用户数据不丢失。

## 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** |
| **3210104095** | 叶沐阳 | 总体框架搭建，集群管理、副本管理、分布式查询，部分容错容灾 |
| **3210105512** | 俞博 | 部分分布式查询，部分副本管理，容错容灾，负载均衡 |
| **3210104116** | 郭一铭 | 分布式查询，数据库查询以及复杂查询 |

# 2 总体设计

## 2.1 总体架构设计

项目由三个主要模块组成：Client，Master和Region。这些模块各自承担不同的职责，共同构建了一个完整的分布式数据库系统。

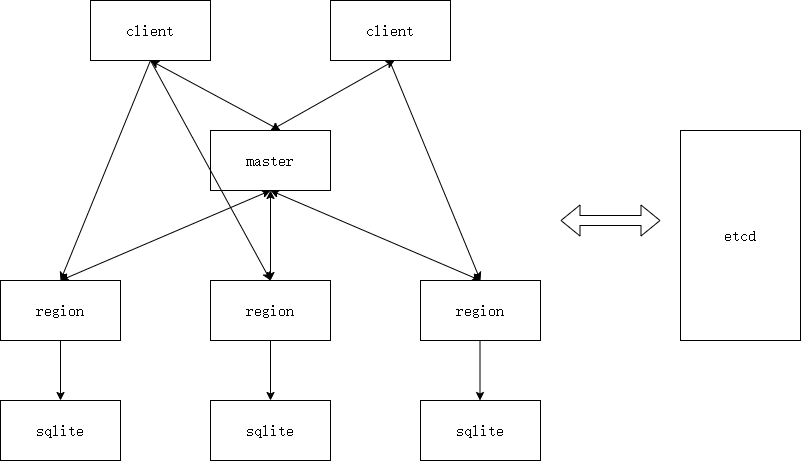
**Client模块**：这是用户与数据库系统交互的接口。它负责接收用户的请求，将其转化为数据库系统可以理解的命令，并将结果返回给用户。此外，Client模块也可能负责一些基本的错误处理和数据验证。

**Master模块**：Master模块是整个系统的核心，它管理着Region模块。Master模块负责实现系统的集群管理、负载均衡、副本管理、容错容灾、数据分布和分布式查询等功能。这些功能确保了系统的稳定性和可靠性，同时也提高了系统的性能。

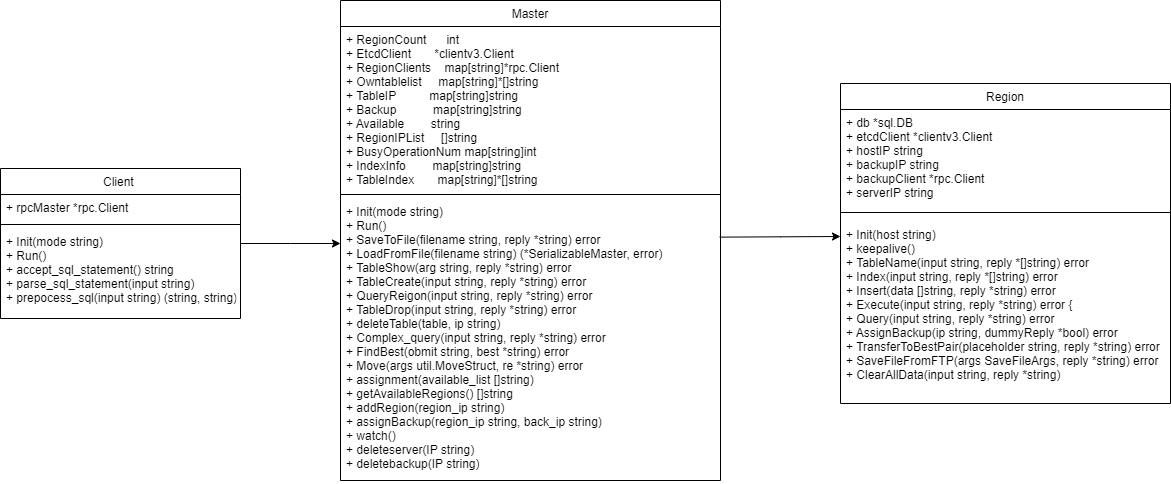
**Region模块**：Region模块负责数据库引擎的运行，并且服从Master的调度安排。它处理实际的数据存储和查询操作，并且可以根据Master模块的指示进行扩展或缩小。

这三个模块在一定的通信框架下进行通信，同时通过etcd集群进行协调。etcd是一个开源的分布式键值存储系统，可以用于配置共享和服务发现。在你的系统中，etcd可能被用来存储系统的配置信息，以及跟踪各个模块的状态。

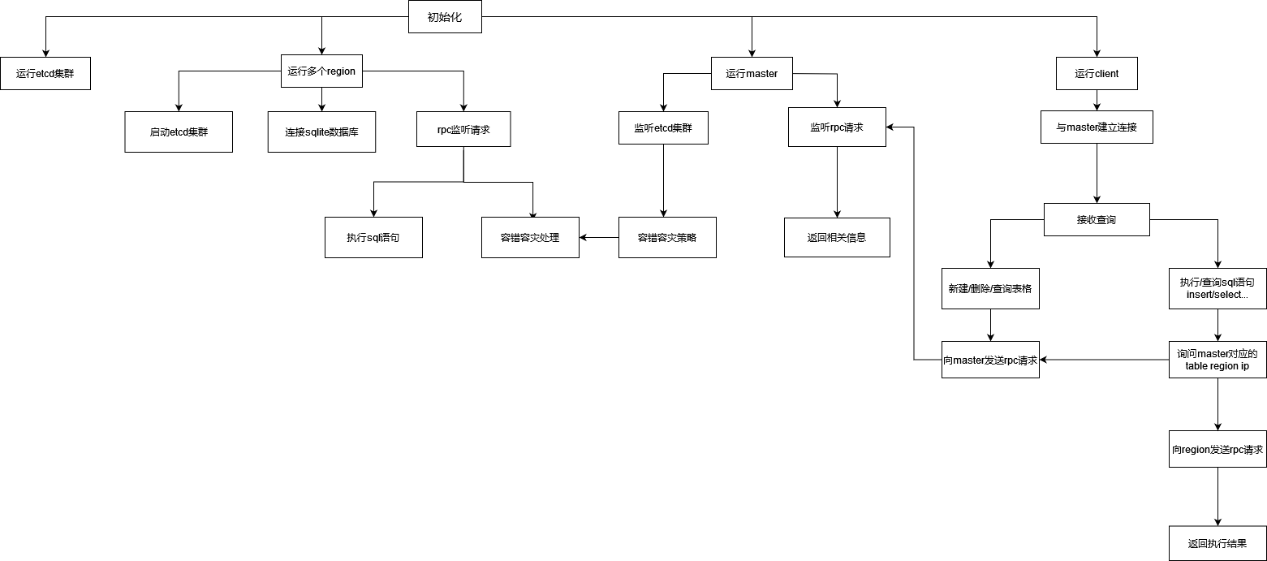
系统的总体架构示意图如下：



总体的类图如下：



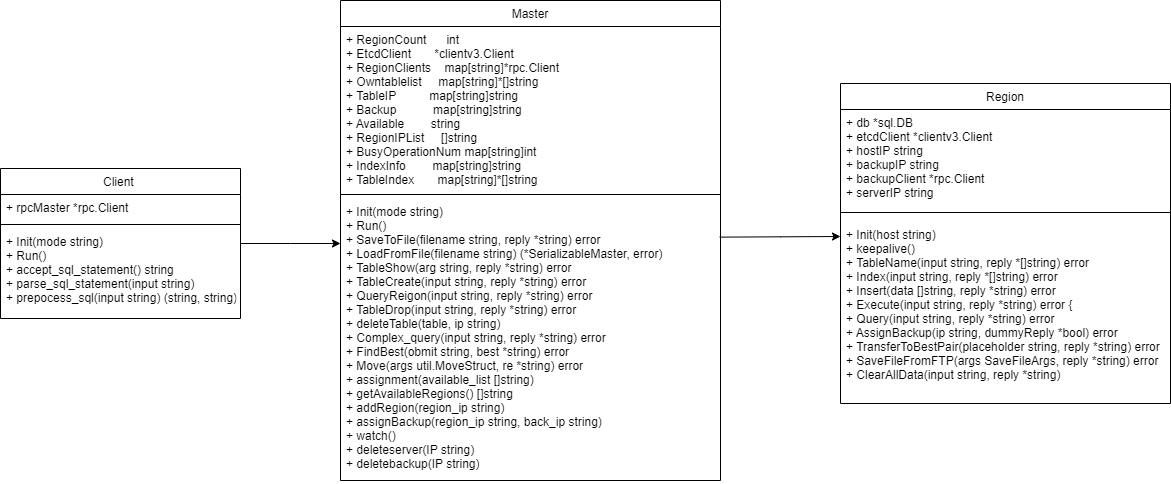
总体工作逻辑如下：



## 2.2 Client模块

### 2.2.1 架构设计

Client的类图如下：



Client模块在整个系统中扮演着至关重要的角色，既负责对外与用户进行直接交互，又对内与Region和Master模块进行连接和协调。其主要功能包括：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。

* 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

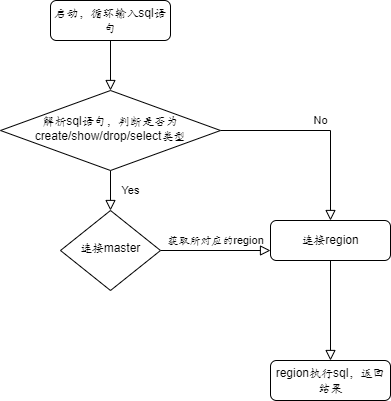
返回：无

* 调用接口：

如果语句为 create、show、drop、select，Client 模块将直接调用 master 接口，由 master 再调用相应的 region 接口来处理请求。这些操作通常涉及数据库的结构性变更或查询，需要 master 进行全局协调和管理。

对于其他类型的操作，例如 insert、update、delete 等，Client 模块首先会调用 master 接口，以确定目标表所在的 region。Master 接口会返回表所在的具体 region 信息。随后，Client 模块将调用对应的 region 接口来执行具体的操作。这种方式确保了操作的高效性和准确性，同时避免了不必要的全局锁定和协调。

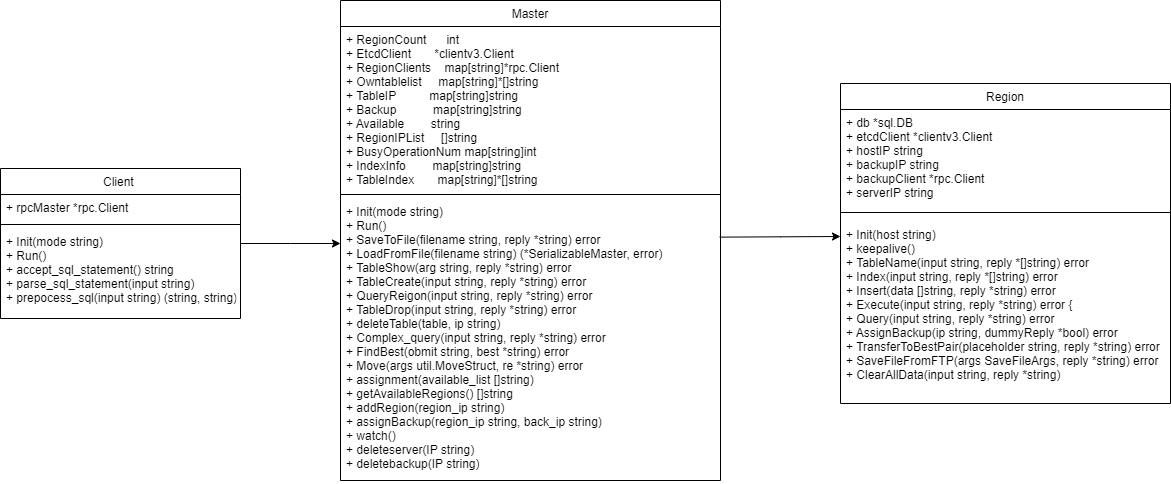
### 2.2.2 工作流程



## 2.3 Master模块

### 2.3.1 架构设计

Master的设计如下图所示:



1. master的主要变量

**type Master struct**中定义了如下变量和缓存表：在 Master 结构体中，定义了如下变量和缓存表，用于管理和协调分布式数据库系统中的各个 region 服务器：

**RegionCount (int)：**

记录了当前 region 主服务器的数量。这个变量用于了解系统中有多少个 region 服务器在运行，以便进行负载均衡和任务分配。

**EtcdClient (\*clientv3.Client)：**

这是 master 进程与 master 服务器上的 etcd 进程通信的媒介。etcd 是一个分布式键值存储，用于存储配置信息、服务发现和协调分布式系统中的各个组件。

**RegionClients (map[string]\*rpc.Client)：**

这是 master 进程与所有在线 region 服务器上 region 进程通信的媒介。通过这个映射表，master 可以方便地与各个 region 服务器进行远程过程调用（RPC），以执行各种管理和调度任务。

**Owntablelist (map[string]\*[]string) 和 TableIP (map[string]string)：**

Owntablelist 记录了每个 table 与其所在 region 的 IP 地址的对应关系。

TableIP 记录了每个 table 的 IP 地址。通过这两个映射表，master 可以快速查找某个 table 所在的 region 服务器，以便进行数据操作和查询。

**IndexInfo (map[string]\*[]string) 和 TableIndex (map[string]string)**：

IndexInfo 记录了每个索引（index）与其所在 table 的对应关系。

TableIndex 记录了每个 table 的索引信息。通过这两个映射表，master 可以管理和查询各个 table 的索引结构，优化查询性能。

**BusyOperationNum (map[string]int)：**

记录了每个 region 的忙碌状态。这个映射表用于跟踪各个 region 当前正在处理的操作数量，以便进行负载均衡和任务调度，避免某个 region 过载。

**RegionIPList ([]string)：**

记录了所有连接过的 region 的 IP 地址。这个列表用于跟踪系统中所有曾经连接过的 region 服务器，方便进行管理和监控。

**Backup (map[string]string)：**

记录了每个 region 和其备份 region 的对应关系。这个映射表用于实现数据的高可用性和容错机制，当某个 region 发生故障时，可以快速切换到其备份 region。

**Available (string)：**

记录当前活跃的 region 的 IP 地址。这个变量用于标识当前系统中哪个 region 是可用的，以便进行数据操作和查询。

通过这些变量和缓存表，master 能够有效地管理和协调分布式数据库系统中的各个 region 服务器，确保系统的高效运行和数据的一致性。

1. master的主要函数
2. Master.go中函数：

**func (master \*Master) Init(mode string)：**

该函数用于初始化 master 系统，包括读取数据库文件、初始化变量和缓存表以及加载数据。通过这个函数，master 系统能够在启动时准备好所有必要的数据和配置。

**func (master \*Master) Run()：**

该函数负责初始化 etcd 集群并开启 RPC 服务进程。通过这个函数，master 系统能够与 etcd 进行通信，并提供远程过程调用服务，以便其他模块进行交互。

**func (master \*Master) GetTableIP(table string, reply \*string) error：**

该函数用于查询 table 和 region 的对应关系。通过调用这个函数，可以获取指定 table 所在的 region 的 IP 地址。

**func (master \*Master) InitTableIP()：**

该函数用于同步本地数据库文件中的 table 信息。通过这个函数，master 可以确保本地缓存中的 table 信息与数据库中的数据一致。

**func (master \*Master) InitIndex(table string)：**

该函数用于同步本地数据库文件中的 index 信息。通过这个函数，master 可以确保本地缓存中的 index 信息与数据库中的数据一致。

1. Func.go中函数：

**func (m \*Master) TableShow(arg string, reply \*string) error：**

该函数用于直接查看 owntablelist 中的所有 region 的 table，并以表格形式返回所有 table 以及其所属 region 的 IP 地址。通过调用这个函数，用户可以方便地获取系统中所有 table 的分布情况，便于管理和查询。

**func (master \*Master) TableCreate(input string, reply \*string) error：**

该函数用于在数据库中创建新的 table。通过调用这个函数，用户可以向数据库中添加新的 table，并将其分配到合适的 region 中。函数会更新 owntablelist 和 TableIP 等缓存表，确保系统中的数据一致性。

**func (master \*Master) TableDrop(input string, reply \*string) error：**

该函数用于在数据库中删除指定的 table。通过调用这个函数，用户可以从数据库中移除不再需要的 table，并更新相关的缓存表和映射关系，确保系统中的数据一致性。

**func (master \*Master) deleteTable(table, ip string)：**

该函数用于删除缓存表中的 table 信息。通过调用这个函数，可以从本地缓存中移除指定的 table 信息，确保缓存表中的数据与数据库中的实际情况一致。

**func (master \*Master) check\_and\_reset\_Regions() error：**

该函数用于检查和重置 region。通过调用这个函数，系统可以定期检查各个 region 的状态，确保它们正常运行。如果发现异常，函数会进行必要的重置操作，以保证系统的稳定性和可靠性。

**func extractTable(s string) string：**

该函数用于从输入字符串中提取 table 名称。通过调用这个函数，可以从复杂的输入字符串中解析出 table 的名称，便于后续的处理和操作。

**func (master \*Master) IndexShow(arg string, reply \*string) error：**

该函数用于查询所有 table 的 index，并以表格形式返回所有 index 以及其所属 table。通过调用这个函数，用户可以方便地获取系统中所有 index 的分布情况，便于管理和查询。

**func (master \*Master) IndexCreate(input string, reply \*string) error：**

该函数用于在数据库中创建新的 index。通过调用这个函数，用户可以向数据库中添加新的 index，并将其关联到合适的 table 中。函数会更新 IndexInfo 和 TableIndex 等缓存表，确保系统中的数据一致性。

**func (master \*Master) IndexDrop(input string, reply \*string) error：**

该函数用于在数据库中删除指定的 index。通过调用这个函数，用户可以从数据库中移除不再需要的 index，并更新相关的缓存表和映射关系，确保系统中的数据一致性。

**func (master \*Master) deleteIndex(index string, table string)：**

该函数用于删除缓存表中的 index 相关信息。通过调用这个函数，可以从本地缓存中移除指定的 index 信息，确保缓存表中的数据与数据库中的实际情况一致。

**func (master \*Master) Move(args util.MoveStruct, re \*string) error：**

该函数用于将 table 移动到指定的 region。通过调用这个函数，可以将 table 从一个 region 移动到另一个 region，以实现负载均衡或其他管理需求。函数会更新相关的缓存表和映射关系，确保系统中的数据一致性。

**func (master \*Master) Complex\_query\_master(input string, reply \*string) error：**

该函数用于处理复杂查询。通过调用这个函数，用户可以对数据库进行复杂的查询操作，并获取查询结果。函数会根据查询条件进行优化和处理，确保查询的高效性和准确性。

**func (master \*Master) Copy(table string, ip string) error：**

该函数用于将 table 复制到本地缓存数据库。通过调用这个函数，可以将指定的 table 数据复制到本地缓存中，以提高查询和操作的效率。函数会更新相关的缓存表和映射关系，确保系统中的数据一致性。

通过这些函数，Master 结构体能够有效地管理和操作分布式数据库系统中的 table 和 index，确保系统的高效运行和数据的一致性。

1. db.go中函数：

这些函数与 region 模块的同名函数功能相似，故不在此赘述。

### 2.3.2 工作流程

在该项目中，Master 层的功能由四个子文件实现，分别是 master.go、etcd.go、func.go 和 db.go。每个文件都有其特定的职责和功能，共同构成了 Master 层的核心功能：

1. **master.go文件。**

该文件存储了所有节点和表格的信息。它定义了缓存表，并进行了系统的初始化，同步了数据库中的信息。通过这个文件，Master 层能够管理和维护系统中的所有节点和表格信息，确保数据的一致性和完整性。

1. **etcd.go文件。**

该文件负责建立与 etcd 的连接，实现数据集群管理，对节点的状态改变进行监听，处理新增节点、节点出错等情况。基于此，Master 层实现了容错容灾等功能，确保系统的高可用性和稳定性。

1. **func.go文件。**

该文件建立了与 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。此外，该文件处理了增加/删除 table/index 请求时对 Master 缓存信息的更新，确保缓存表中的数据与实际情况一致。

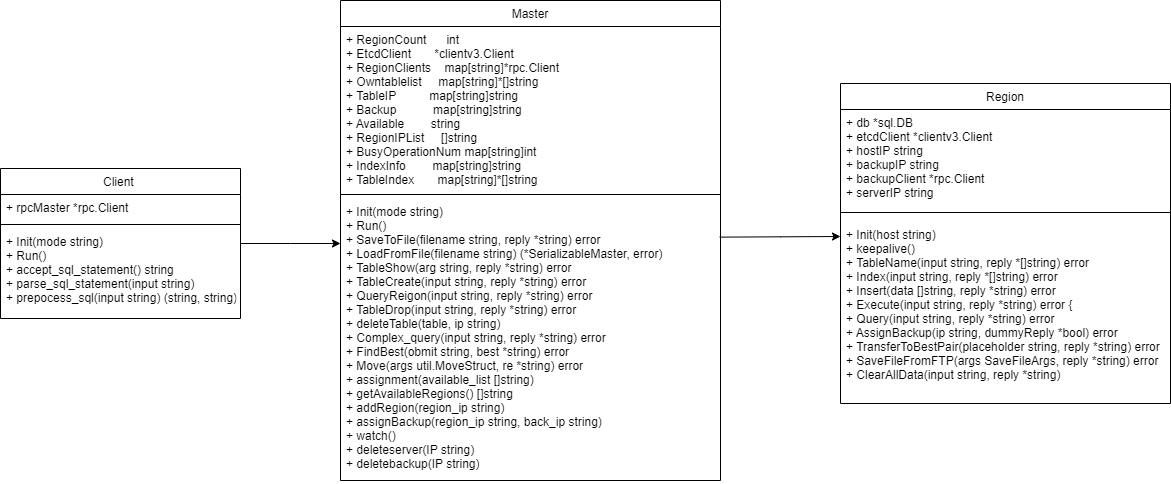
1. **db.go 文件。**

该文件存放了 Master 本地缓存数据库（用于复杂查询）所需要的函数。通过这些函数，Master 层能够高效地处理复杂查询请求，提高查询性能和响应速度。

## 2.4 Region模块

### 2.4.1 架构设计

Region的类图如下：



相关的主要函数如下：

1. Region.go 中的函数：

**func (region \*Region) Init()：**

该函数用于实现 region 的初始化，包括连接数据库文件和注册 RPC 服务。通过调用这个函数，region 服务器能够准备好所有必要的资源和服务，以便处理来自 Master 和 Client 的请求。

**func (region \*Region) keepalive()：**

该函数用于在 etcd 中保持活跃。通过调用这个函数，region 服务器可以定期向 etcd 发送心跳信号，确保自己在集群中的活跃状态，从而避免被误认为已下线。

**func (region \*Region) foundhostIP() string：**

该函数用于获取本机的 IP 地址。通过调用这个函数，region 服务器可以获取自己的 IP 地址，以便进行网络通信和服务注册。

1. Func.go中函数：

**func (region \*Region) TableName(input string, reply \*[]string) error：**

该函数用于返回当前 region 管理的 table 的名字。通过调用这个函数，可以获取该 region 服务器上所有 table 的名称列表，便于管理和查询。

**func (region \*Region) Index(input string, reply \*[]string) error：**

该函数用于返回当前 region 管理的 index 的名字。通过调用这个函数，可以获取该 region 服务器上所有 index 的名称列表，便于管理和查询。

**func (region \*Region) Execute(input string, reply \*string) error：**

该函数用于处理非查询类的 SQL 语句。通过调用这个函数，可以执行插入、更新、删除等操作，并返回执行结果。

**func (region \*Region) Query(input string, reply \*string) error：**

该函数用于处理查询类的 SQL 语句。通过调用这个函数，可以执行查询操作，并返回查询结果。

**func (region \*Region) AssignBackup(ip string, dummyReply \*bool) error：**

该函数用于给主 region 分配副本 region。通过调用这个函数，可以指定一个副本 region，以实现数据的高可用性和容错机制。

**func (region \*Region) Insert(data []string, reply \*string) error：**

该函数用于向 table 中批量插入 SQL 语句。通过调用这个函数，可以将多条数据批量插入到指定的 table 中，提高数据插入的效率。

**func (region \*Region) Get(input string, reply \*[]string) error：**

该函数用于 SQL 查询后以字符串数组形式返回获取的所有数据。每个字符串代表一个返回值。通过调用这个函数，可以获取查询结果，并以字符串数组的形式返回，便于后续处理。

通过这些函数，Region 结构体能够有效地管理和操作分布式数据库系统中的数据，确保系统的高效运行和数据的一致性。

### 2.4.2 工作流程

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

在util模块中，有几个关键函数用于实现模块的核心功能。这些函数涵盖了元素查找、RPC调用超时处理、字符串切片操作以及目录清理等操作。以下是每个函数的详细描述及其实现逻辑：

**FindElement(pSlice \*[]string, str string) int**

**功能描述：**

该函数在给定的字符串切片中查找指定的字符串，并返回其索引。如果找不到该字符串，则返回-1。

**实现逻辑：**

遍历字符串切片，比较每个元素与目标字符串。如果找到匹配的字符串，返回其索引。如果遍历结束仍未找到目标字符串，则返回-1。

该功能常用于在需要查找特定元素的位置时调用，比如在管理集群节点列表时。

**TimeoutRPC(call \*rpc.Call, ms int) (\*rpc.Call, error)**

**功能描述：**

该函数检查RPC调用是否在指定时间内完成。如果调用超时，返回错误信息；否则返回RPC调用结果。

**实现逻辑：**

使用定时器监控RPC调用的执行时间。在指定的毫秒数内，如果RPC调用完成，则返回结果。如果超过指定时间，仍未完成则返回超时错误。该功能用于确保系统的响应速度，避免长时间等待某个远程过程调用。

**DeleteFromSlice(pSlice \*[]string, str string) bool**

**功能描述：**

该函数在字符串切片中删除指定的字符串。如果成功删除，返回true；否则返回false。

**实现逻辑：**

遍历字符串切片，找到与目标字符串匹配的元素。删除匹配的元素，并调整切片长度。返回删除操作的结果状态。

该功能常用于从集群列表或其他字符串集合中移除特定元素。

**AddToSlice(ptr \*[]string, newString string)**

**功能描述：**

该函数在字符串切片的末尾添加一个新的字符串。

**实现逻辑：**

将新的字符串附加到字符串切片的末尾。更新切片长度。确保新的字符串成功添加到切片中。

该功能用于动态扩展字符串集合，例如添加新的节点或配置项。

**AddToSliceIndex(ptr \*[]string, newString string, index int)**

**功能描述：**

该函数在字符串切片的指定位置插入一个新的字符串。

**实现逻辑：**

确定插入位置是否合法（在切片范围内）。在指定位置插入新的字符串，并调整切片长度。确保插入操作不会破坏现有数据。

该功能用于在特定位置插入新的元素，例如在有序列表中插入新项。

**CleanDir(localDir string)**

**功能描述：**

该函数删除指定目录中的所有文件。

**实现逻辑：**

打开指定目录，读取目录中的所有文件。逐一删除目录中的文件，确保没有残留。返回清理操作的结果状态。

该功能用于清理临时文件或重置目录状态，确保系统的文件环境干净。

# 3 分布式设计

## 3.1 通信架构和通信协议

### 3.1.1 通信架构

在系统架构中，远程过程调用（RPC）扮演着连接各个组件的关键角色。在这个分布式系统中，Client与Master之间的连接是多对一的，这意味着多个Client节点可以同时连接到一个Master节点。Master节点负责协调和管理来自多个Client的请求，并将任务分配给合适的Region节点。而Region节点与Master之间也是多对一的连接，多个Region节点连接到同一个Master节点，Master通过RPC与这些Region节点通信，管理数据分片、分配任务以及监控Region的状态。与此同时，Client与Region的连接是不固定的，这是因为Client的连接数量会随着SQL语句的执行而动态变化。正常情况下，随着系统的运行，Client与Region之间的连接应该会逐步变成一对多的关系，这样可以更有效地处理大量的数据库操作请求。

建立rpcClient非常简单，只需要指定目标服务器的IP地址和端口号即可通过DialHttp函数获取到rpcClient。这个过程简化了客户端与服务器之间的通信，开发人员无需过多关注网络通信细节，只需专注于实现业务逻辑。一旦获取到rpcClient，就可以使用它来调用Region或Master节点上的函数，从而实现分布式系统中各个节点之间的协作与通信。这种简洁高效的RPC通信机制为系统的构建和扩展提供了便利，使得系统更加灵活和可靠。

### 3.1.2 通信协议

由2.5可以看到，在系统设计中，采用了非阻塞的RPC调用函数TimeoutRPC，这使得系统能够更好地处理可能出现的网络延迟或异常情况。在master/func.go和region/func.go中的大部分函数都遵循了一种参数传递和返回值处理的策略，即通过传入两个指针参数来实现函数的封装和结果的返回。

这种设计模式的优势在于，通过传入指针参数，函数可以直接修改指针指向的内存地址上的值，而无需额外的返回值。这样做可以减少函数调用时的内存开销，提高程序的运行效率。同时，将返回值放在指针参数中，也使得函数的调用更加灵活，可以直接在函数外部修改返回值所对应的内存地址，从而达到实时更新返回值的目的。

在大部分函数中采用这种参数传递和返回值处理的策略，使得系统的代码更加简洁清晰，同时也增强了系统的可维护性和扩展性。这种设计模式使得系统的各个组件之间能够更好地协作和通信，从而实现了系统功能的高效执行和良好的性能表现。

### *3.1.2.1* *master/func.go中的函数*

master/func.go中的函数都是由client.go进行调用，自身再调用Region中的函数或者使用自己的缓存表（得到返回值）而返回的，其涉及的操作包含：

|  |  |  |
| --- | --- | --- |
| 函数签名 | 作用简介 | 返回值含义 |
| func (master \*Master) TableCreate(input string, reply \*string) error | 根据传入的input语句，在一个region上创建table | 新建table所在机器IP |
| func (m \*Master) TableShow(arg string, reply \*string) error | 直接使用缓存表查询当前所有的table和table所在的region\_ip | 表格形式返回所有table以及其所属region\_ip |
| func (master \*Master) TableDrop(input string, reply \*string) error | 根据传入的input语句，调用相关region，删掉某一table | 返回执行结果 |
| func (m \*Master) TableShow(arg string, reply \*string) error | 通过缓存表，返回所有已建的table | 缓存表中所有的table |
| func (master \*Master) IndexCreate(input string, reply \*string) error | 根据传入的index参数，在某一table上创建索引 | 新建index所在机器IP |
| func (master \*Master) IndexDrop(input string, reply \*string) error | 根据传入的index参数，丢弃某一table上的索引 | 无含义，不使用 |
| func (master \*Master) IndexShow(arg string, reply \*string) error | 通过缓存表，返回所有已建的index和对应的table | 已建的index和对应的table |
| func (master \*Master) Complex\_query(input string, reply \*string) error | 对给定的input语句实现复杂查询，将需要的表格都移入master本地缓存数据库中再执行命令进行查询 | 复杂查询的执行结果 |
| func (master \*Master) FindBest(obmit string, best \*string) error { | 根据缓存表中的信息，计算并返回当前负载最轻的节点 | 负载最轻的节点IP |
| func (master \*Master) Move(args util.MoveStruct, re \*string) error | 读取一个指定的表，将该表内容移动到指定的目标region节点中，主要用于负载均衡 | 无含义，不使用 |
| func (master \*Master) MoveHalf(args util.MoveStruct, re \*string) error | 从指定的源节点中移动一半的表格到当前负载最轻的目标节点中 | 无含义，不使用 |
| func (master \*Master) LoadBalance(placeholder string, re \*string) error | 负载均衡函数，遍历缓存寻找到操作数最多的节点，调用Move和MoveHalf等函数将一半表格转移到当前负载最轻的节点 | 无含义，不使用 |
| func (master \*Master) NowInfo(input string, reply \*string) error | 返回当前master缓存表中的关键信息，主要用于调试和观察节点使用情况 | master缓存表中TableIp，Backup等关键变量信息 |
| func (master \*Master) SaveToFile(filename string, reply \*string) error | 把当前master缓存表中的信息序列化并保存到指定文件中 | 是否保存成功 |

### *3.1.2.2* *region/func.go中的函数*

该部分的函数由master和client进行调用，起到备份策略和处理select, insert, delete操作并获得返回值的作用。

|  |  |  |
| --- | --- | --- |
| 函数签名 | 作用简介 | 返回值含义 |
| func (region \*Region) Execute(input string, reply \*string) error | client调用，操作本机sqlite3数据库获得SQL操作的结果，如果是主服务器同时通知副本服务器执行语句 | SQL操作得到的结果 |
| func (region \*Region) AssignBackup(ip string, dummyReply \*bool) error | 给当前region分配备份，调用SaveFileFromFTP通知备用服务器从ftp服务器处下载主服务器数据库文件 | 未使用，无意义 |
| func (region \*Region) SaveFileFromFTP(args SaveFileArgs, reply \*string) error | 备份服务器Region 从ftp服务器上下载主服务数据库文件到本地 | 是否下载成功 |
| func (region \*Region) TransferToBestPair(bestIp string, reply \*string) error | 将当前region中的内容全部转存到给定的bestIp中，通常用于容错容灾处理 | 返回接受内容的服务器IP |
| func (region \*Region) Query(input string, reply \*string) error | 向连接sqlite数据库输入sql语句查询数据，通常是select和show类型的sql语句，返回结果 | SQL操作查询到的结果 |
| func (region \*Region) ClearAllData(input string, reply \*string) error | 删除当前节点所有的数据库文件，清空数据库，通常用于容错容灾处理 | 未使用，无意义 |

## 3.2 副本管理

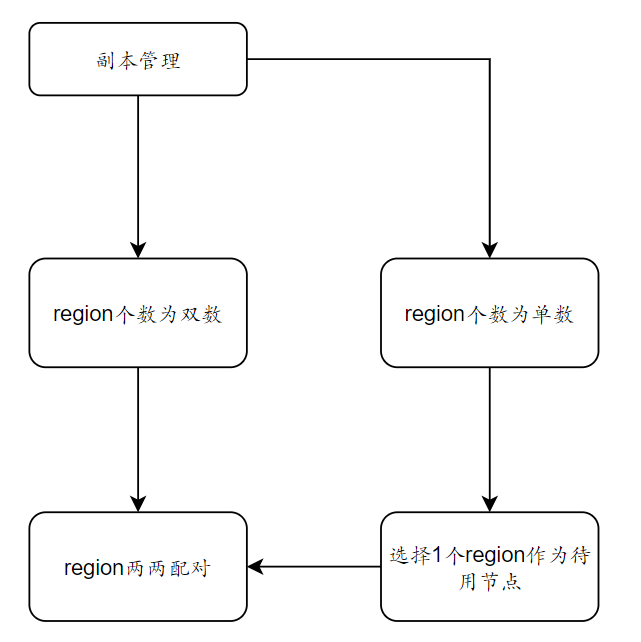
在这个系统中，副本管理功能的设计采用了主从服务器的架构。在这种架构下，每个主服务器都有一个或多个副本服务器，副本服务器对主服务器进行实时备份，确保数据的安全性和可靠性。主服务器上的所有操作都会同时在副本服务器上执行，以保持数据的一致性。

系统要求节点个数至少为2个。如果节点个数为单数，则多余的节点将被作为待用节点，等待后续分配。这些待用节点可以作为备份服务器的备份，以应对主服务器或其他副本服务器的故障。这样做可以提高系统的容错能力，保障数据的持久性和可靠性。

如果节点个数为双数，则系统将节点两两配对，形成主从服务器对。每个主服务器都有一个对应的副本服务器，负责对主服务器的数据进行实时备份。这种配对的方式可以使得系统在发生故障时能够快速切换到备用服务器，保证系统的高可用性和稳定性。

采用主从服务器的架构，配合合适的节点配对策略，可以有效提高系统的可靠性和容错能力，保障系统数据的安全性和持久性。

该功能的流程图如下：



该功能的关键函数如下：

// 给server region分配backup，由server给backup下载data.db  
func (region \*Region) AssignBackup(ip string, dummyReply \*bool) error {  
 fmt.Printf("Region.AssignBackup called,backup ip: %v", ip)  
 client, err := rpc.DialHTTP("tcp", "localhost:"+ip)  
 if err != nil {  
  fmt.Printf(ip, " rpc.DialHTTP err: %v")  
 } else {  
  //重新写server和backup的IP  
  region.serverIP = ""  
  region.backupClient = client  
  region.backupIP = ip  
  // 通知backup下载data.db,直接覆盖了本地data.db  
  //在本地测试中，IP都改成127.0.0.1  
  //util.TransferFile(region.hostIP+util.FILE\_PORT, ip+util.FILE\_PORT, "./data/"+region.hostIP+".db")  
  util.TransferFile("127.0.0.1", "127.0.0.1"+util.FILE\_PORT, "./data/"+region.hostIP+".db")  
  
  backupClient, err := rpc.DialHTTP("tcp", "localhost:"+region.backupIP)  
  if err != nil {  
   fmt.Println("Error:", err)  
  }  
  
  var res []string  
  args := SaveFileArgs{  
   ServerIP:     region.hostIP,  
   FileName:     region.hostIP + ".db",  
   SaveFileName: "",  
  }  
  // 通知backup从ftp获取data.db  
  backupClient.Go("Region.SaveFileFromFTP", args, &res, nil)  
  
 }  
 return err  
}

## 3.3 数据分布

在系统设计中，为了最小化系统的复杂度，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的服务器上。这种设计简化了系统的架构，减少了数据管理和维护的复杂度，提高了系统的可靠性和性能。

Master进程的内存中存储了分布式数据库所有表的信息，包括表的名称以及它们与Region服务器的关系。此外，Master还存储了所有索引的信息，包括索引的名称以及它们与相应数据表的关系。这些信息的存储在Master进程的内存中，可以快速访问和更新，用于管理数据库的元数据信息。

而数据表的实际数据和索引数据则存储在Region服务器中。每个Region服务器负责存储和管理一部分数据表和索引的数据，根据Master进程提供的信息，Region服务器知道自己负责哪些数据表和索引的数据。这种数据存储方式使得数据在物理上集中存储，便于管理和维护，并且提高了数据的访问效率和一致性。

将完整的数据表存储在服务器中，结合Master进程存储的元数据信息，以及Region服务器存储的实际数据和索引数据，构成了分布式数据库系统的基本架构。这种设计简化了系统的复杂度，提高了系统的可靠性和性能，是一种有效的数据管理方案。

Master中的存储信息如下：

type Master struct {  
 RegionCount      int  
 EtcdClient       \*clientv3.Client  
 db               \*sql.DB  
 RegionClients    map[string]\*rpc.Client  
 Owntablelist     map[string]\*[]string // ip -> tables  
 TableIP          map[string]string    // table -> ip  
 Backup           map[string]string    // region server ip -> Backup server ip  
 Available        string               // available regions  
 RegionIPList     []string  
 BusyOperationNum map[string]int       // operations for each region in 1 minute, > BUSY\_THRESHOLD deemed as busy  
 IndexInfo        map[string]string    // index->table  
 TableIndex       map[string]\*[]string // table->indexs  
 TableCnt         map[string]int       // table->count 存储本地数据库中table应该copy了多少次  
}

## 3.4 均衡负载

在该项目中，我们通过Master节点的调度实现了负载均衡的设计。在这个设计中，负载大小是综合考虑了表格数和数据条数的综合计算的结果。

在创建一张表时，系统总是会选择负载最少的节点进行创建。这意味着新创建的表格会被分配到当前负载最轻的节点上，以确保系统的负载均衡和资源利用率最优。同样地，在节点下线后，孤立的节点内容也会被转移到负载最少的节点上，以保证系统数据的完整性和可用性。

另外，系统还实现了负载监控机制。当某个节点的负载超过了预先设定的阈值，系统会自动触发负载均衡策略，将该节点的部分表格转移到当前负载最轻的节点上。这样做可以避免单个节点负载过重导致系统性能下降，同时保证整个系统的稳定运行。

通过以上的设计和实现，系统能够有效地进行负载均衡，保证了各个节点的负载相对均衡，提高了系统的整体性能和可靠性。这种负载均衡的设计方案使得系统能够更好地应对不同负载情况下的数据管理需求，为用户提供更加稳定、高效的服务。

负载均衡相关的代码段如下：

1. 创建表时选择负载最少的节点：

 //寻找table数最少的节点  
  min, best := math.MaxInt, ""  
  for ip, pTables := range master.Owntablelist {  
   if len(\*pTables) < min && master.BusyOperationNum[ip] < util.BUSY\_THRESHOLD {  
    min, best = len(\*pTables), ip  
   }  
  }

1. 负载监控：

func (master \*Master) MoveHalf(args util.MoveStruct, re \*string) error {  
 fmt.Println("-----------------------------------------")  
 source := args.Source  
 backupSource := master.Backup[source]  
  
 tables := \*master.Owntablelist[source]  
 if len(tables) <= 2 {  
  // 如果只有两个表，直接返回，不需要挪到其他region上  
  return nil  
 }  
 fmt.Println("Move half of " + source)  
  
 //迁移  
 var bestIp string  
 for i := 0; i < len(tables)/2; i++ {  
  master.FindBest(source, &bestIp)  
  args := util.MoveStruct{  
   Table:  tables[i],  
   Region: bestIp,  
   Source: source,  
  }  
  var tmp string  
  fmt.Println("MoveHalf from" + source + " TO " + bestIp)  
  master.Move(args, &tmp)  
  args = util.MoveStruct{  
   Table:  tables[i],  
   Region: master.Backup[bestIp],  
   Source: backupSource,  
  }  
  fmt.Println("MoveHalf from" + backupSource + " TO " + master.Backup[bestIp])  
  master.Move(args, &tmp)  
  master.TableIP[tables[i]] = bestIp  
  \*master.Owntablelist[bestIp] = append(\*master.Owntablelist[bestIp], tables[i])  
  util.DeleteFromSlice(master.Owntablelist[source], tables[i])  
 }  
  
 fmt.Println("-----------------------------------------")  
 return nil  
}

## 容错容灾

在系统中，容错容灾主要处理服务器意外下线的情况，结合了副本管理中设计的主从服务器架构。具体处理方式如下：

1. 某主服务器下线：

如果系统中有待用的Region节点，系统会选择其中一个作为新的主服务器，将其启动并作为新的主服务器。

如果没有待用的Region节点，则系统会将副本Region中的内容转存到某个Region对中，然后将副本Region变成待用Region，等待后续的分配和使用。

1. 某副本服务器下线：

如果系统中有待用的Region节点，系统会将该下线的副本服务器设为副本Region，以保持数据的备份和冗余。

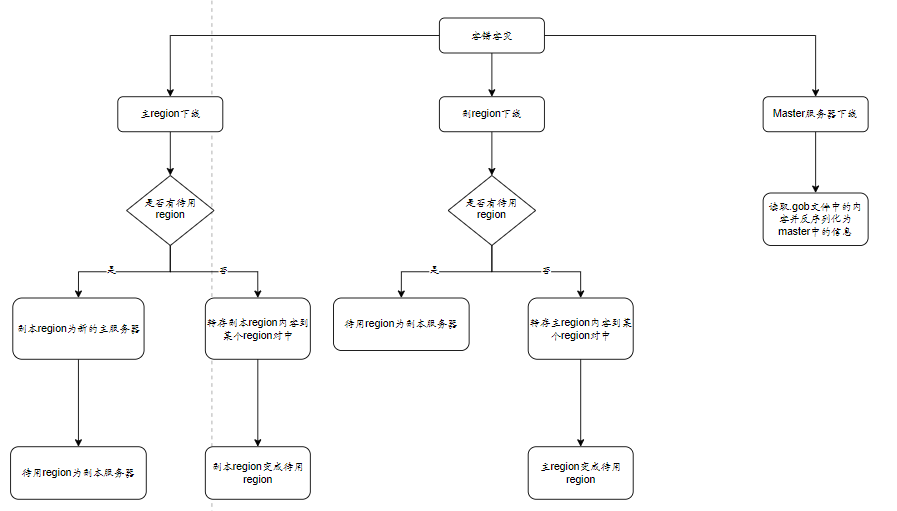
如果没有待用的Region节点，则系统会将主服务器Region中的内容转存到其他Region对中，然后将主Region转换为待用Region，确保数据的安全和可靠性。

1. Master服务器下线：

每次操作都会序列化Master中的关键属性，并保存到.gob文件中。如果Master服务器意外下线，再次启动时，系统会读取.gob文件中的内容，并将其反序列化为Master中的信息，恢复系统状态和数据管理。

通过以上的处理方式，系统能够有效地应对服务器意外下线的情况，保障系统的可用性和数据的完整性。通过灵活地使用待用的Region节点和备份的副本Region，系统能够快速地进行故障恢复和数据恢复，确保系统的稳定运行和持续可用性。

具体流程图如下：



容错容灾关键代码段如下：

1. 主服务器下线  
   // 如果server挂了  
   func (master \*Master) deleteserver(IP string) {  
    //如果有available，启动backup为server;否则把backup中内容都转存到某个ip中。  
    if master.Available != "" {  
     //从master.RegionIPList中删除  
     util.DeleteFromSlice(&master.RegionIPList, IP)  
       
     //删除client  
     client, ok := master.RegionClients[IP]  
     if ok {  
      client.Close()  
      delete(master.RegionClients, IP)  
     }  
     //拨号添加backup  
     new\_server := master.Backup[IP]  
     new\_client, err := rpc.DialHTTP("tcp", "localhost:"+new\_server)  
     if err != nil {  
      fmt.Println("master error >>> region rpc "+new\_server+" dial error:", err)  
      return  
     }  
     master.RegionClients[new\_server] = new\_client //添加到server列表  
     master.RegionIPList = append(master.RegionIPList, new\_server)  
     //转移owntablelist  
     \_, ok = master.Owntablelist[IP]  
     if ok {  
      backupIP, ok := master.Backup[IP]  
      if ok {  
       master.transferOwnTables(IP, backupIP)  
      } else {  
       log.Printf("%v has no backup", IP)  
       master.removeOwnTables(IP)  
      }  
     }  
     delete(master.Backup, IP)  
     master.Backup[new\_server] = master.Available  
     master.Available = ""  
     //拨号通知server他的backup  
     master.assignBackup(new\_server, master.Backup[new\_server])  
     
     master.RegionIPList = append(master.RegionIPList, new\_server)  
     fmt.Println("server " + IP + " down, " + new\_server + "change to server with backup is " + master.Backup[new\_server])  
       
     //把backup存到client??  
     
     
    } else {  
     util.DeleteFromSlice(&master.RegionIPList, IP)  
     // 把server-backup中内容都转存到best pair中  
     // backup 变成available  
     backup\_ip := master.Backup[IP]  
     
     client := master.RegionClients[backup\_ip]  
     fmt.Println("IP: " + IP + "BACKUP: " + backup\_ip)  
     
     //保存table名  
     table\_name := \*master.Owntablelist[IP]  
     
     var accept\_ip string  
     master.FindBest(IP, &accept\_ip)  
     \_, err := util.TimeoutRPC(client.Go("Region.TransferToBestPair", accept\_ip, &accept\_ip, nil), util.TIMEOUT\_M)  
     if err != nil {  
      fmt.Println("server "+backup\_ip+" TransferToBestPair return err ", err)  
     }  
     fmt.Println(" server " + IP + "'s content transfer to " + accept\_ip)  
     
     //table都转存到table\_accept\_ip中  
     //遍历table\_name，类型是[]string  
     
     for \_, table := range table\_name {  
      master.TableIP[table] = accept\_ip  
     }  
     \*master.Owntablelist[accept\_ip] = append(\*master.Owntablelist[accept\_ip], table\_name...)  
     
     var res string  
     //删除backup中的所有信息  
     \_, err = util.TimeoutRPC(client.Go("Region.ClearAllData", "", &res, nil), util.TIMEOUT\_M)  
     if err != nil {  
      fmt.Println("Clear Region  "+backup\_ip+" return err ", err)  
     }  
     //从master中删除server和backup的信息  
     master.DeleteRegionInfo(backup\_ip, false)  
     master.DeleteRegionInfo(IP, true)  
     //backup清空变成avaiable  
     master.Available = backup\_ip  
     
    }  
   }

（2）副本服务器下线

// 如果backup挂了  
func (master \*Master) deletebackup(IP string) {  
 //查询backup中是哪个server的值是IP  
 server := ""  
 for k, v := range master.Backup {  
  if v == IP {  
   server = k  
   break  
  }  
 }  
 if server == "" {  
  fmt.Printf("backup for %v not found", IP)  
  return  
 }  
  
 if master.Available != "" {  
  //把avaiable设为backup  
  //查询backup中是哪个server的值是IP  
  
  new\_backup := master.Available  
  master.Available = ""  
  master.Backup[server] = new\_backup  
  //拨号通知server他的backup  
  master.assignBackup(server, new\_backup)  
  
  fmt.Println("server " + server + " 's backup " + IP + " change to " + master.Backup[server])  
  //从master.Backup中删除  
  delete(master.RegionClients, IP)  
  
 } else {  
  //把server中内容都转存到某个server pair中,server转为available  
  util.DeleteFromSlice(&master.RegionIPList, server)  
  //转存到accept\_ip中  
  client := master.RegionClients[server]  
  var accept\_ip string  
  master.FindBest(IP, &accept\_ip)  
  err := client.Call("Region.TransferToBestPair", accept\_ip, &accept\_ip)  
  if err != nil {  
   fmt.Println("server "+server+" TransferToBestPair return err ", err)  
  }  
  fmt.Println(" server " + server + "'s content transfer to " + accept\_ip)  
  
  //tableIP和owntablelist都转存到accept\_ip中  
  table\_name := \*master.Owntablelist[server]  
  for \_, table := range table\_name {  
   master.TableIP[table] = accept\_ip  
  }  
  \*master.Owntablelist[accept\_ip] = append(\*master.Owntablelist[accept\_ip], table\_name...)  
  
  var res string  
  //删除backup中的所有信息  
  \_, err = util.TimeoutRPC(client.Go("Region.ClearAllData", "", &res, nil), util.TIMEOUT\_M)  
  if err != nil {  
   fmt.Println("Clear Region  "+server+" return err ", err)  
  }  
  
  //从master中删除server和backup的信息  
  master.DeleteRegionInfo(IP, false)  
  master.DeleteRegionInfo(server, true)  
  master.Available = server  
  
 }  
  
}

## 3.6 集群管理

在本系统中，我们使用etcd作为集群管理工具。etcd是一个分布式键值对存储系统（key-value store），采用了raft分布式一致性算法，保证了数据的一致性和可靠性。通过利用etcd提供的Grant、Put、KeepAlive和Watch等API，我们实现了集群节点的管理功能。

当一个Region节点上线时，它会在etcd的键值对存储系统中创建一个具有生命周期的键值对。这个键值对以该Region节点所在服务器的IP地址为键，持续维持其生命。通过这个键值对，其他节点和组件可以获取到该Region节点的信息，从而实现对节点的管理和通信。

而当Master节点开启后，它会持续监测etcd键值对存储系统中的各种更新。这意味着Master节点会实时感知到集群中各个节点的变化，例如新节点的上线、节点的下线等。通过监测这些变化，Master节点可以动态地调整集群的状态和配置，确保集群的稳定运行和数据一致性。

通过etcd作为集群管理工具，我们实现了集群节点的动态管理和监控。这种基于键值对存储系统的集群管理机制具有高可用性、高一致性和高可靠性，为系统的稳定运行提供了坚实的基础。

## 3.7 复杂sql语句的查询

在本系统中，我们实现了对复杂SQL语句的查询支持。特别是当查询涉及到不止一个表格时，系统采取了一种高效的处理方式：将所有涉及的表格缓存到Master节点中，在Master节点上进行查询操作，最后输出结果并清空缓存。

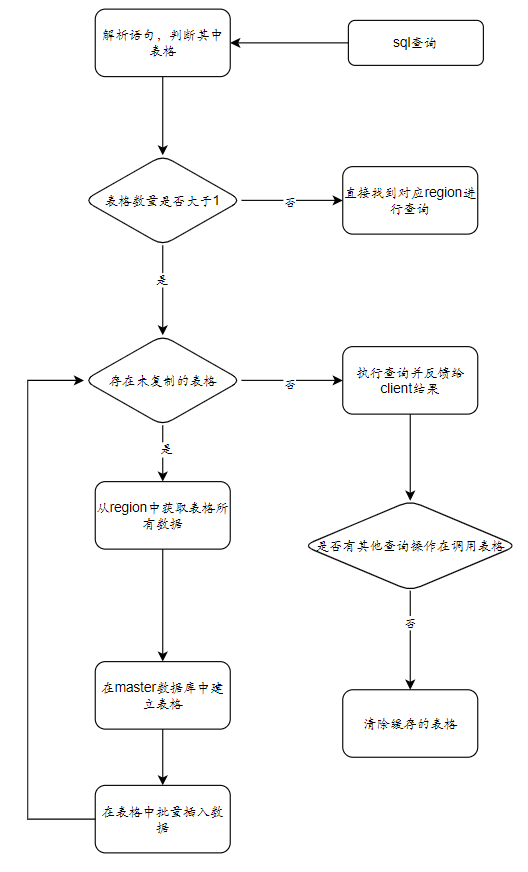
当系统收到一个复杂SQL查询请求时，它首先会解析该查询语句，确定其中涉及到的所有表格。然后，系统会将这些表格从各个Region节点中获取，并缓存到Master节点的内存中。

接下来，在Master节点上执行查询操作。由于所有涉及到的表格都已经缓存到了Master节点中，系统可以直接在Master节点上进行联合查询，处理复杂的SQL语句，以获取查询结果。

最后，系统将查询结果输出，并清空Master节点中的表格缓存，以释放资源并确保下一次查询的准确性和高效性。

通过这种方式，系统能够快速高效地处理复杂的SQL查询请求，避免了在多个Region节点之间进行数据交互和联合查询的开销，提高了查询的执行效率和系统的整体性能。同时，通过在Master节点上进行查询操作，也使得系统的架构更加简洁清晰，降低了系统的复杂度和维护成本。

本功能的流程图如下：



本功能的关键代码如下：

（1）处理sql语句、调用复制函数、查询并清理

func (master \*Master) Complex\_query\_master(input string, reply \*string) error {  
 fmt.Println("master Complex\_query\_master.called")  
 items := strings.Split(input, " ")  
 var tables []string  
 var size int  
 var ip []string  
 for i := 0; i < len(items); i++ {  
  name := items[i]  
  found := master.TableIP[name]  
  if found != "" { //存在该table  
   tables = append(tables, name)  
   ip = append(ip, found)  
   size++  
  }  
 }  
  
 //copy到master本地的数据库中  
 for i := 0; i < size; i++ {  
  found := master.TableCnt[tables[i]]  
  //fmt.Println(found)  
  if found == 0 {  
   master.Copy(tables[i], ip[i])  
  }  
  master.TableCnt[tables[i]] += 1  
  //fmt.Println(master.TableCnt[tables[i]])  
 }  
  
 //查询  
 master.Query(input, reply)  
  
 //清理  
 for i := 0; i < size; i++ {  
  master.TableCnt[tables[i]] -= 1  
  if master.TableCnt[tables[i]] == 0 {  
   input = "drop table " + tables[i]  
   master.db.Exec(input)  
  }  
 }  
  
 return nil  
}

（2）从region中获取表格并缓存到master本地服务器中

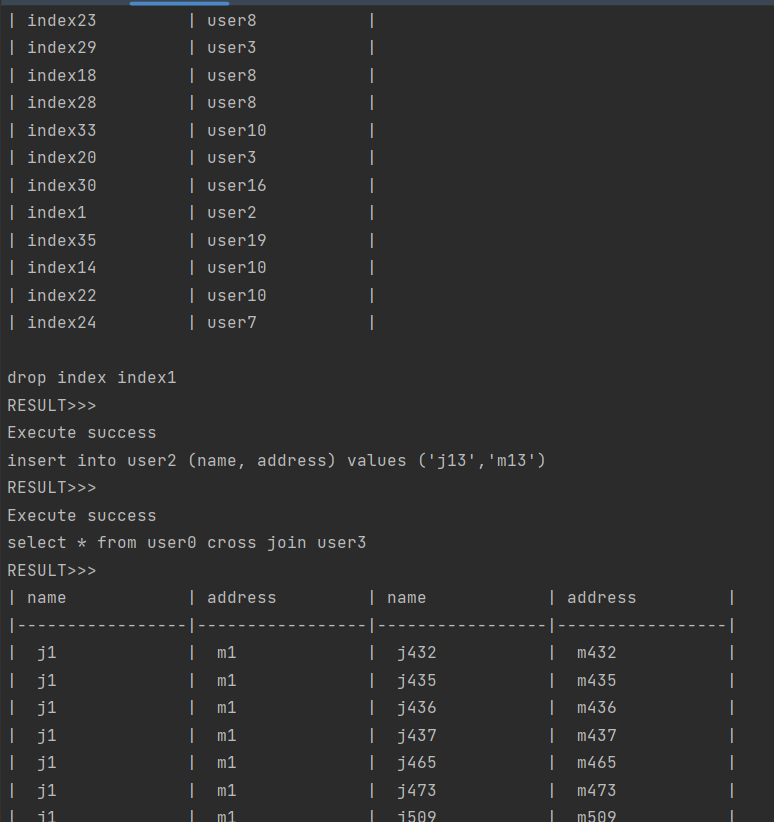
// 将table复制到本地数据库中  
func (master \*Master) Copy(table string, ip string) error {  
 fmt.Println("master copy.called")  
 oldip := master.TableIP[table]  
 rpcOldRegion := master.RegionClients[oldip]  
  
 //从旧region中获取数据  
 input := "select \* from " + table  
 var res []string  
 call, err := util.TimeoutRPC(rpcOldRegion.Go("Region.Get", input, &res, nil), util.TIMEOUT\_M)  
 if err != nil {  
  fmt.Println("SYSTEM HINT>>> timeout, master down!")  
 }  
 if call.Error != nil {  
  fmt.Println("RESULT>>> failed ", call.Error)  
 }  
 //fmt.Println(res)  
  
 //获取create表的sql  
 input = "select sql from sqlite\_master where tbl\_name = "  
 input = input + "'" + table + "'"  
 var res2 []string  
 call2, err2 := util.TimeoutRPC(rpcOldRegion.Go("Region.Get", input, &res2, nil), util.TIMEOUT\_M)  
 if err2 != nil {  
  fmt.Println("SYSTEM HINT>>> timeout, master down!")  
 }  
 if call2.Error != nil {  
  fmt.Println("RESULT>>> failed ", call2.Error)  
 }  
  
 //在master本地的数据库中建表  
 for \_, line := range res2 {  
  //fmt.Println("line:", line)  
  master.db.Exec(line)  
 }  
  
 var inputt []string  
 inputt = append(inputt, table)  
 inputt = append(inputt, res...)  
 master.Insert(inputt)  
 fmt.Println("copy success")  
 return nil  
}

# 4 系统测试

# 4.1 大规模测试

测试100000条代码成功。

部分截图如下：



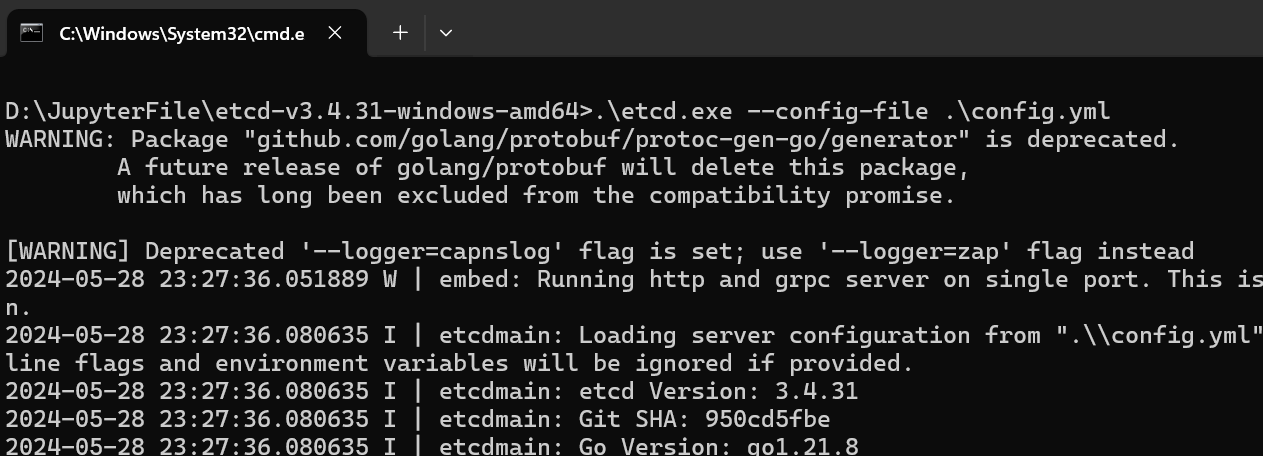
# 

测试用例构造代码如下：

*#include<bits/stdc++.h>*  
using namespace std;  
int size;  
int cnt;  
int index;  
void s1(){  
 string s="create table user";  
 s=s+to\_string(size)+"(name TEXT,address TEXT);";  
 size++;  
 cout<<s<<endl;  
}  
void s2(){  
 string s="show tables;";  
 cout<<s<<endl;  
}  
void s3(){  
 string s="drop table user";  
 size--;  
 s=s+to\_string(size)+";";  
 cout<<s<<endl;  
}  
void s4(){  
 string s="insert into user";  
 int x=rand()%size;  
 s=s+to\_string(x)+" (name, address) VALUES ('J";  
 cnt++;  
 s=s+to\_string(cnt);  
 s=s+"','M"+to\_string(cnt)+"');";  
 cout<<s<<endl;  
}  
void s5(){  
 string s="create index index";  
 index++;  
 s=s+to\_string(index)+" on user";  
 int x=rand()%size;  
 s=s+to\_string(x)+"(name);";  
 cout<<s<<endl;  
}  
void s6(){  
 string s="show indexes;";  
 cout<<s<<endl;  
}  
void s7(){  
 string s="drop index index";  
 s=s+to\_string(index)+";";  
 index--;  
 cout<<s<<endl;  
}  
void s8(){  
 string s="select name from user";  
 int x=rand()%size;  
 s=s+to\_string(x)+";";  
 cout<<s<<endl;  
}  
  
void s9(){  
 string s="select \* from user";  
 int x=rand()%size;  
 s=s+to\_string(x)+" cross join user";  
 x=rand()%size;  
 s=s+to\_string(x)+";";  
 cout<<s<<endl;  
}  
int main(){  
 freopen("testbig.txt","w",stdout);  
 srand(time(NULL));  
 int T=100000;   
 size=1;  
 cout<<"create table user0(name TEXT,address TEXT);"<<endl;  
 for(int i=0;i<T;i++){  
  if(size==0) {  
   size=1;  
   cout<<"create table user0(name TEXT,address TEXT);"<<endl;  
  }  
  int k=rand()%9;  
  switch (k){  
   case 0:  
    s1();  
    break;  
   case 1:  
    s2();  
    break;  
   case 2:  
    s3();  
    break;  
   case 3:  
    s4();  
    break;  
   case 4:  
    s5();  
    break;  
   case 5:  
    s6();  
    break;  
   case 6:  
    if(index>0)s7();  
    break;  
   case 7:  
    s8();  
    break;  
   case 8:  
    if(size>2)s9();  
    break;  
   }  
 }  
 return 0;  
}

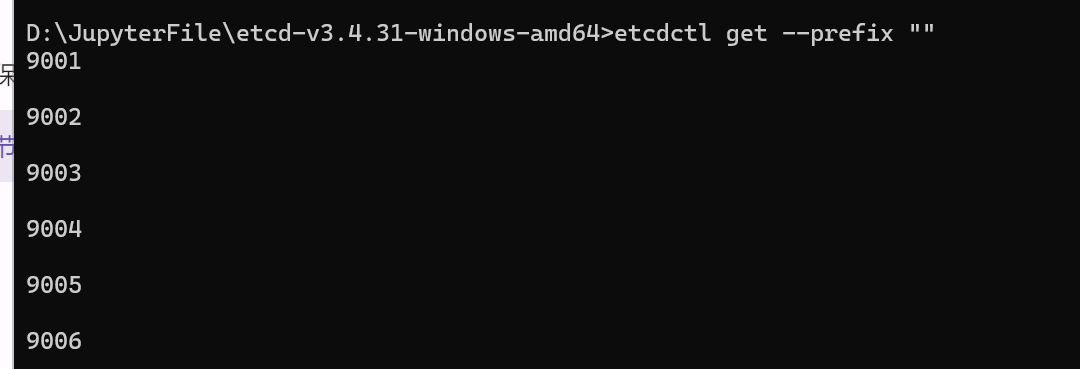
# 4.2 集群管理

通过etcd实现分布式集群管理。在代码执行前打开etcd终端：



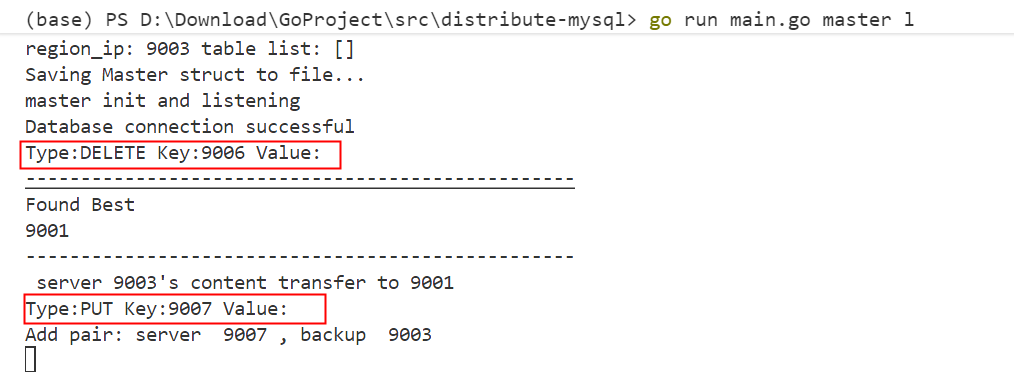
每启动一个region节点，都会创建一个租约，然后周期性地向 etcd 发送心跳信号来证明节点存在。

启动6个节点后：可以看到etcd中存在6个key：



当某个节点意外关闭后，可以看到master中监视到etcd的键值对变化，启动对应的函数。

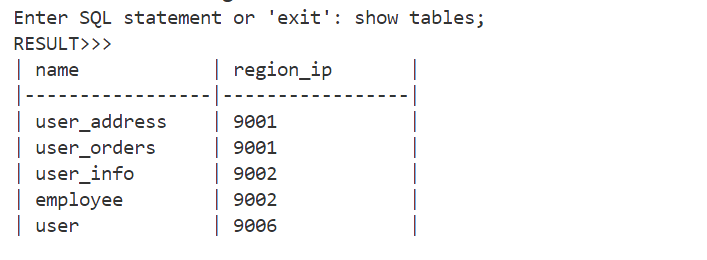
下图中，监视到9006节点关闭，由于当前没有待用节点，将该主从服务器对（9003，9006）中的内容转到目前内容最少的9001节点中，随后9003号节点转化为待用节点；随后监视到新增9007号节点，9007和9003号节点组成新的主从服务器对。



# 4.3 分布式查询

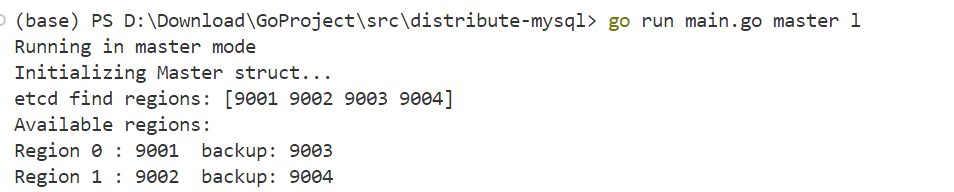
master中的TableIP变量和Owntablelist变量记录了各个表格存在的节点；；Backup变量存储了每个主服务器对应的副本服务器。具体增删改查操作在节点服务器中进行。

输入show tables指令，返回各个表和存在的节点位置。可见各个表存在不同的服务器上。



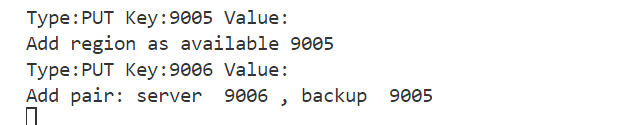
# 4.4 副本管理

本项目在初始化时能根据可用节点来自动分配主从服务器对，任何在主服务器中的操作都会同时在副本服务器中进行。下图初始化时检测到9001，9002，9003，9004四个节点，配对成（9001，9003），（9002，9004）两对主从服务器，对于使用者来说，只存在9001和9002两个节点。

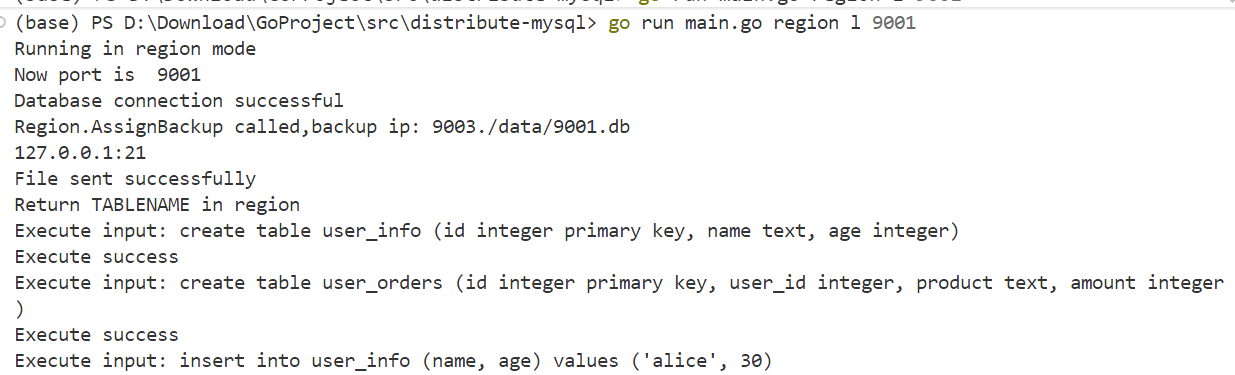


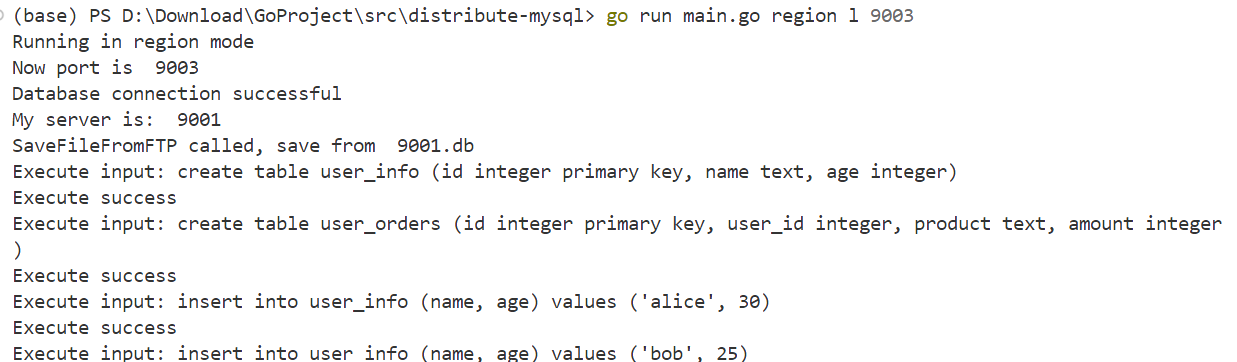
每当增加一个新的节点，如果存在待用节点，新节点将和待用节点配对形成主从服务器对，否则新节点变成待用节点。

对于上图中的4个节点，继续上线9005节点，由于目前时单数，9005节点变成待用节点。上线9006节点，和9005节点配对形成主从服务器对。



测试主服务器9001和副本服务器9003，副本服务器开始时复制了主服务的data.db文件，两服务器执行的命令完全相同：

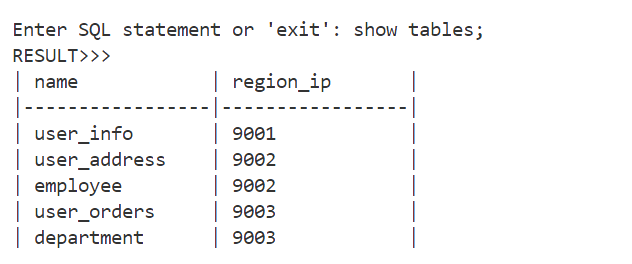




# 4.5 容错容灾

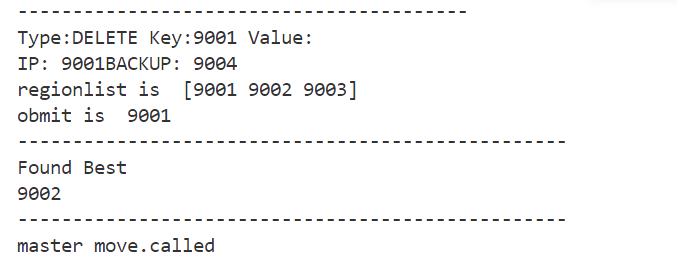
4.5.1节点数双数

单数时，没有待用节点。当前主从节点对（9001,9004），（9002,9005），（9003,9006）。存储表格情况：



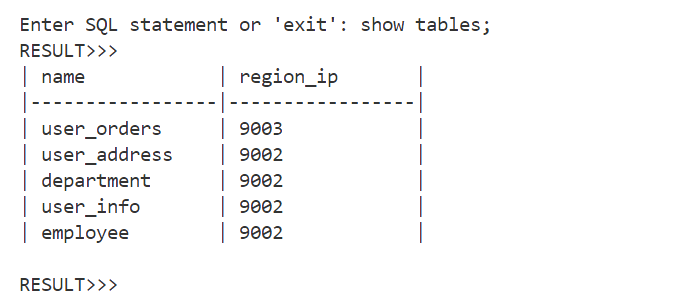
**下线主服务器9001：**

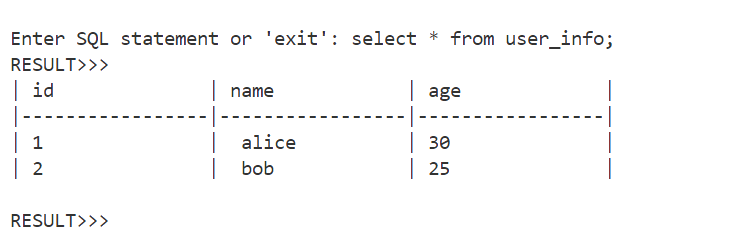
将所有内容从副本节点9004转存到当前负载最少的9002节点，将9004节点转为待用节点。





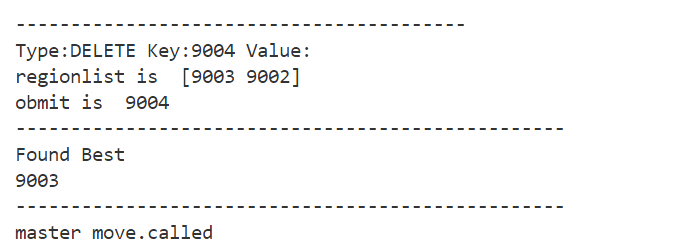
查看table情况，发现原来在9001号节点中的user\_info全部转存到9002号节点中，select数据查询正常：

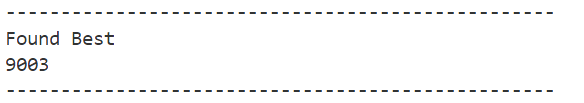




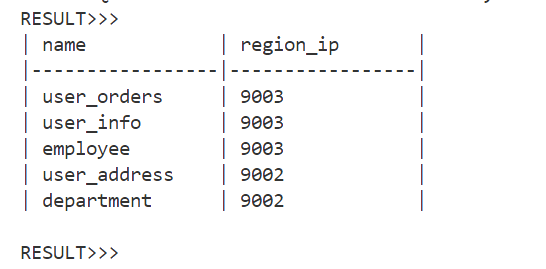
**下线副服务器9004：**

将所有内容从主节点9001转存到当前负载最少的9003节点，将9004节点转为待用节点。



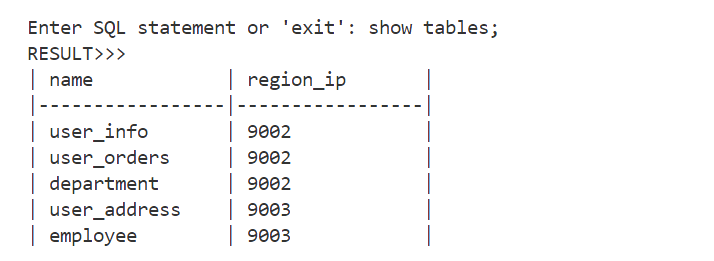
 

查看table情况，发现原来在9001号节点中的user\_info全部转存到9003号节点中。



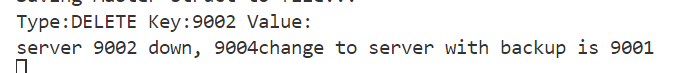
4.5.2节点数单数

单数时，存在待用节点。当前主从节点对（9002,9004），（9003,9005），待用节点9001。存储表格情况：

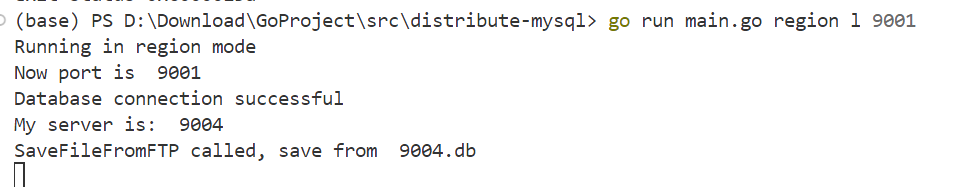


**下线主服务器9002：**

9001的备用服务器9004转为主服务器，待用节点9001做9002的备用服务器。

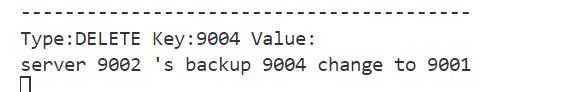
****

查看服务器9001的输出，发现9001节点通过ftp服务器获得了9004的data.db数据，和9004节点保持了同步。

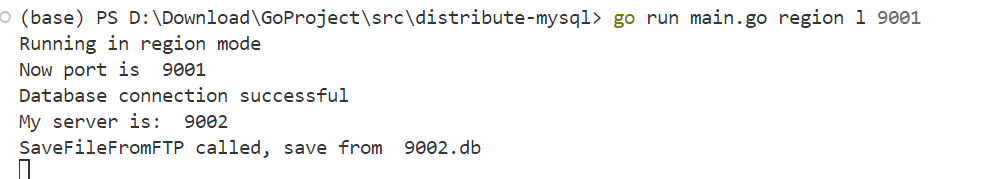
****

**下线副服务器9004：**

9004是9002的副本服务器，因此将待用节点9001分配给9002做副服务器。



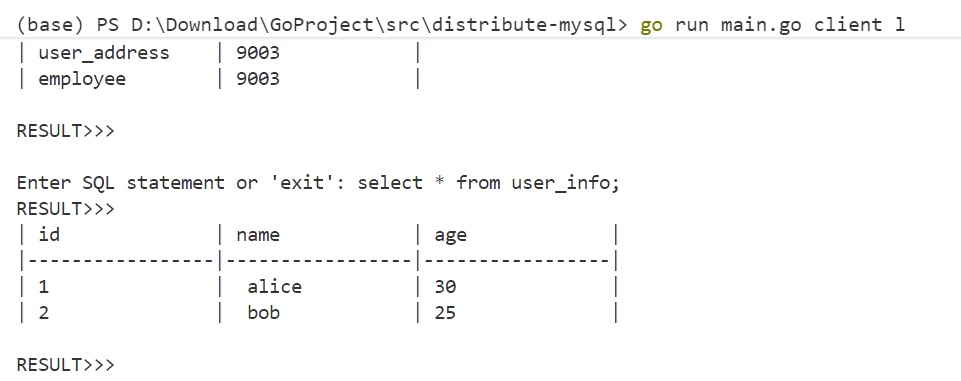
查看节点9001的输出，发现9001节点通过ftp服务器获得了9002的data.db数据，和9002节点保持了同步。



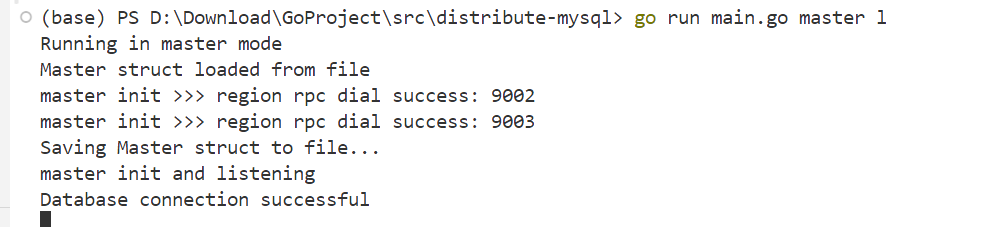
4.5.3 Master服务器下线

新建表格并插入部分数据，将Master服务器下线后再上线，重新查询，发现仍然能查询成功。

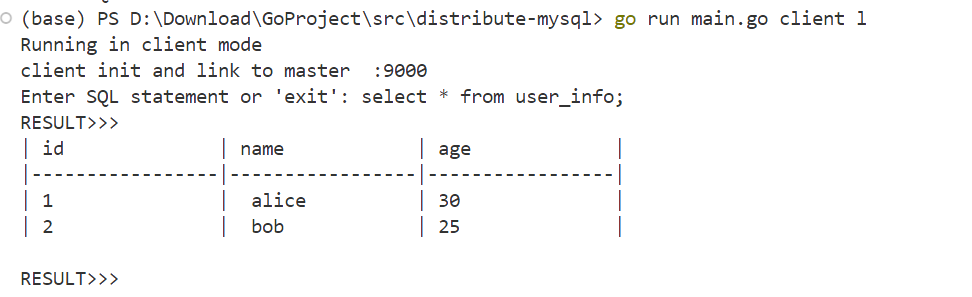
Master下线前：



Master重启后，查看Master输出，Master重新和server 9002和9003连接：



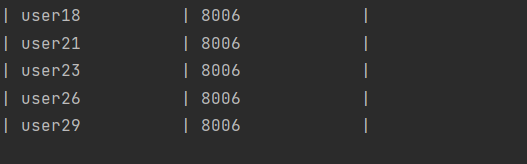
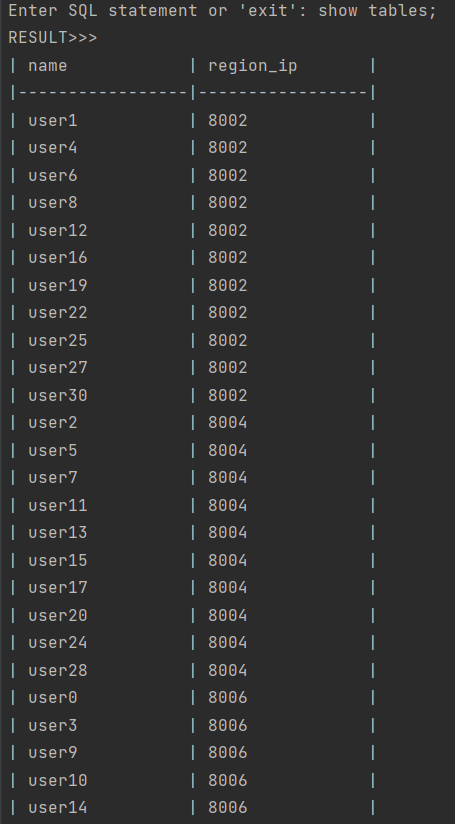
重新查询，发现仍然成功：



# 4.6 负载均衡

4.6.1 创建表的均衡

Create的时候选择最优的region（最优：当前拥有的table最少且操作数不超过所设阈值），进行大量的create后，show table如下：

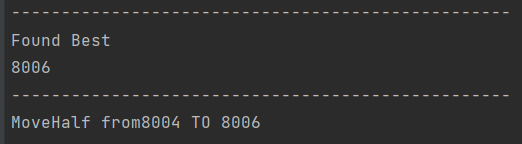


可见所有的表都均匀地分布在各个服务器上。从表的创建数量上来看，每台服务器的存储单元都得到了平衡的利用，存储负载较为均衡。

4.6.2 所有操作的均衡

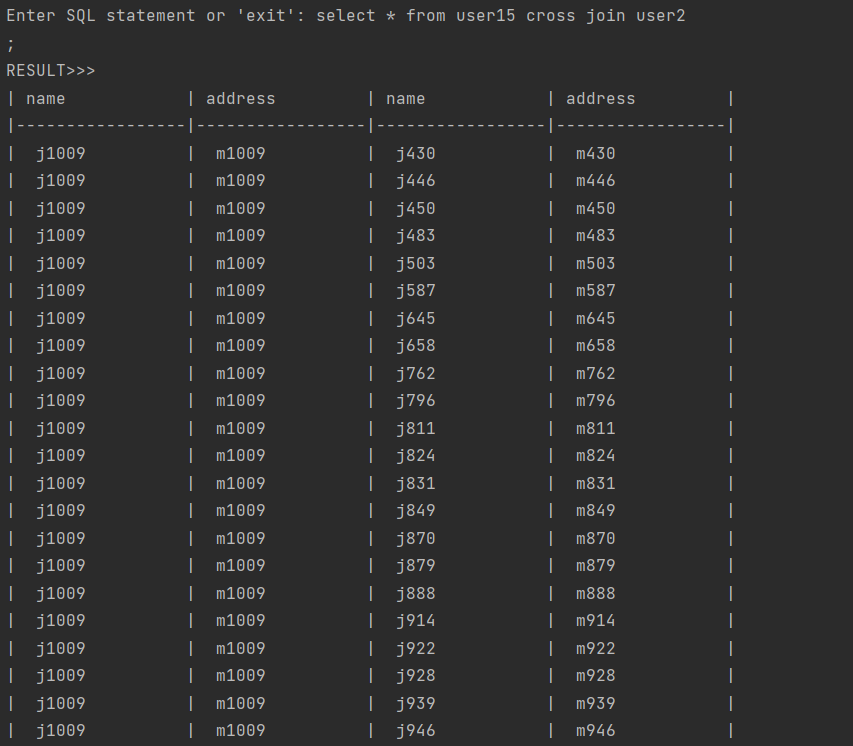
当region的操作数达到一定阈值，且当前region至少有两张表，会把一半的表挪到最优的region（每次挪表都重新找优的region）：通过这样的操作，可以让被认为BUSY的region（操作数超过某个阈值）的表转移到其他的region服务器里，相当于将多余的操作均衡到整个系统上，达到负载均衡。

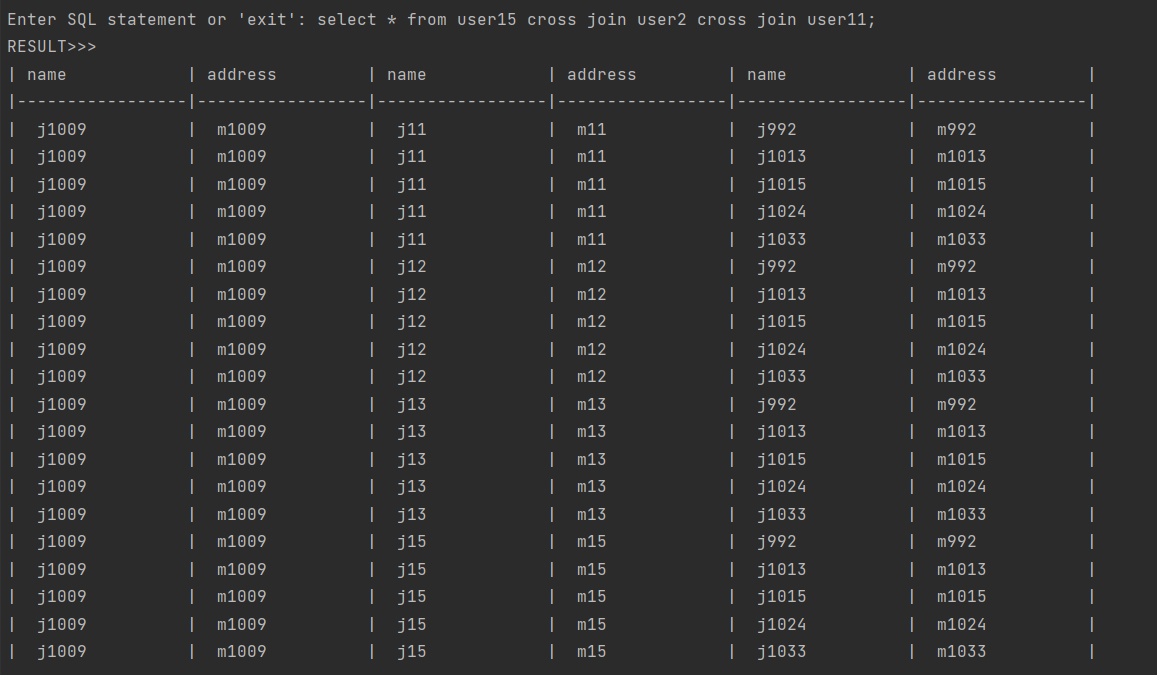
执行show tables;命令，8004服务器达到BUSY状态，因此自动将一半的表挪至8006.



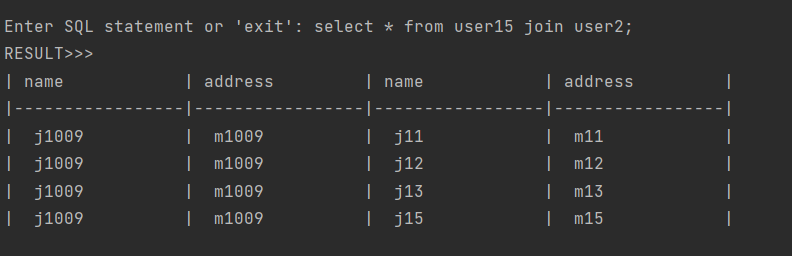
# 4.7 复杂查询

下图展示了cross join，能够输出结果。

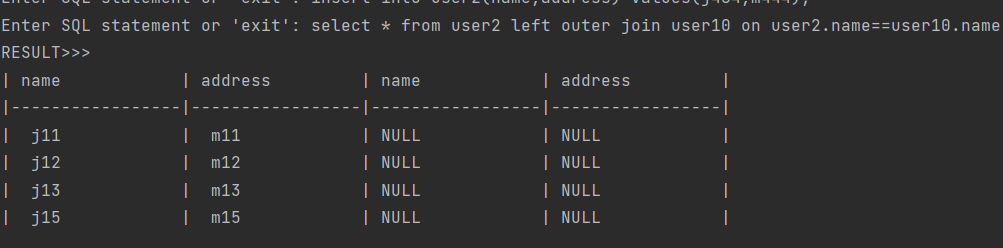


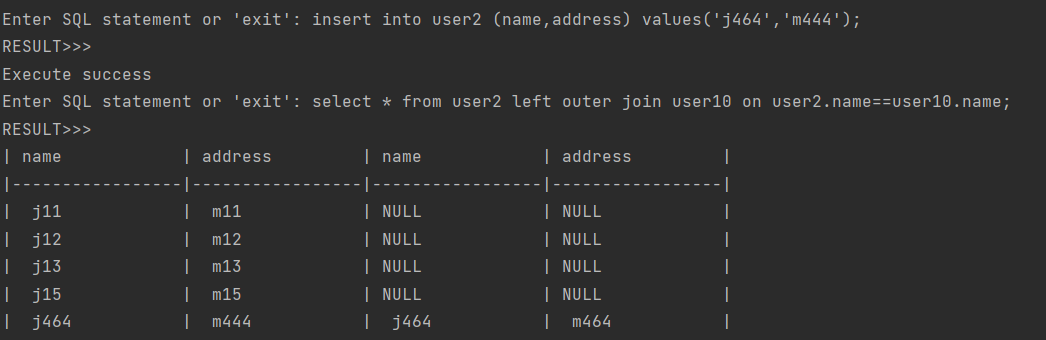


（inner）join：



Left out join：





支持对数据库的简单操作。

以下是来自不同region的table进行join查询的过程：

