Distributed Key Value Store设计文档

目标

一个分布式的KV服务，client端可以有一个KV存储的操作界面，server端提供存储服务。通过replication来保证在有部分机器宕机的情况下，系统仍然可用。

实现的feature：

1️⃣ slave server **动态的primary-backup选举**，支持给一个group添加新的backup，在master fail之后重新选举

2️⃣ master server / slave server**并发控制**：通过读写锁的设置，master 的Join, Leave,Query, slave的Put, Get, Del, Sync,TransShard这些IPC都支持并发调用，内部进行并发控制。

3️⃣ **可扩展性**：可以动态的添加新的group，或者移除旧的group，每一个group只要有任何一个slave server存活，那么数据就不会丢失。采用了virtualNode的设计，添加新的group时，可以最小化数据迁移。

4️⃣ **一致性保证**：当client的请求成功返回时，可以保证primary slave已经处理完请求，所有的backup slave都已经得知了请求，并会最终将请求处理完毕。

5️⃣ Master 的primary-backup**没有实现**，实现逻辑和slave的primary-backup类似，预留了设计空间，但时间原因没有完成。

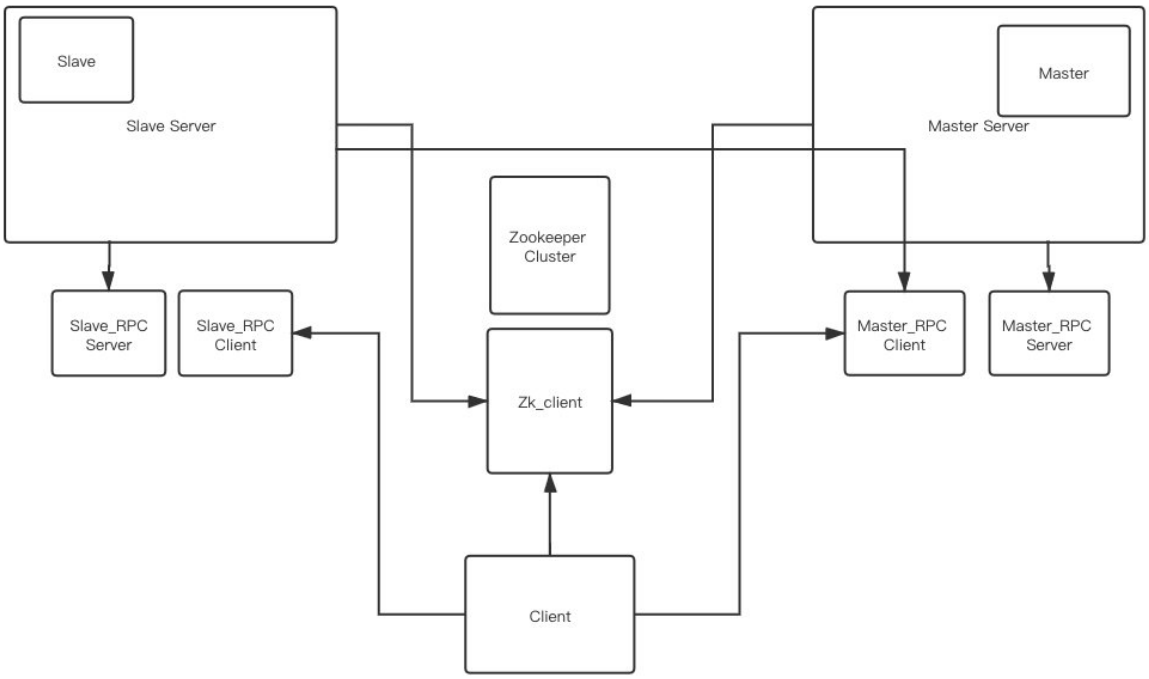
System Environment

- **语言**： **go** [version go1.14.2 darwin/amd64]
  - RPC框架: gRPC
  - zookeeper client: github.com/samuel/go-zookeeper/zk
- **系统环境**
  - macOS Catalina 10.15.5 , Darwin，Darwin Kernel Version 19.5.0, root:xnu-6153.121.2~2/RELEASE_X86_64 x86_64
  - **docker Engine**[version 19.03.8] **compose** [version 1.25.5]
  - **zookeeper**： 3.6.1

设计

1. 整体架构

   分为Master_Server, Slave_Server,Client, Admin四个部分，另外还有和Zookeeper集群交互的模块。



其中Slave Server是*多个group，每个group分为primary backup*，Master Server也可以有primary backup。但是暴露的接口相同。

2. Master (Shard分配逻辑)

   借鉴6.824的设计。Master提供的功能是对metadata的维护。生成从shardID -> groupID的配置，动态的将所有shard均匀分配到不同的slave服务器组。

   采用的是**VirtualNode虚拟节点**的设计，初始化时配置Master维护ShardNum个虚拟节点，通过crc32将key映射到0~ShardNum个虚拟节点上，然后Master维护从虚拟节点到实际节点的映射，并使得每次新增/移除实际节点时，移动的虚拟节点数量最少。

   提供Join, Leave, Query三个接口 (由RPC Server封装成RPC接口)。

```
type ShardMaster struct {
  latest        int        // 最新的版本号

  ip       string
  port     int
  hostname string
  shardNum int

  shardConfLock sync.RWMutex // 访问Configuration时需要拿取对应读写锁
  shardConfs    []*Configuration
}

/* Join 往配置中增加多个group, 每个group包含多个服务器名。拿写锁*/
func (m *ShardMaster) join(newGroups map[int][]string) error

/* Leave 除配置中多个group, 拿写锁*/
func (m *ShardMaster) leave(groupIDs []int) error

/* Query 查询对应序号的配置, -1为最新。拿读锁*/
```

```
func (m *ShardMaster) query(version int) (*Configuration, error)
```

核心的逻辑是在实际节点增加(Join)和减少(Leave)时, 最少地移动虚拟节点。

使用的算法简单的说就是先计算新的配置，然后让现有的配置向新的配置靠拢，多去少补，

```
以Join为例
假设共有ShardNum个虚拟节点，原来的配置是1,2,3三个group，每个group分配到7,7,6个虚拟节点
通过一次Join操作增加了4,5两个新的group。计算可得新的分配方案为[4,4,4,4,4]
先从原先的1,2,3三个group中拿取虚拟节点,让其管理的虚拟节点数量减少到4， 分别拿取[3,3,2]个
然后将这8个节点分配给新的4,5两个group，每个分到4个
往ShardConfs中增加新的配置
```

如果预期实际节点比较多，那么ShardNum可以配置的大一些，但过大也会影响性能。

3. Master_Server

   将master包装了一层，并提供RPC服务， Master的主备应该在这里实现(还没有做), 逻辑应该和slave的主备类似。

4. Slave（Put,Get,Del 存储逻辑）

   Slave对它所管理的虚拟节点提供KV读写服务。提供Put, Get,Del接口

```
type LocalStorage struct {
  cond    *sync.Cond
  lock    *sync.RWMutex

  storage map[string]string  /* KV 存储*/
  state   StorageState  /* UNREADY, READY, EXPIRED*/
}

type Slave struct {
  /*
    shards: 当前slave管理的虚拟节点列表*/
  shardsLock *sync.RWMutex
  shards    []int
  /*
    localstorages:  虚拟节点号 -> 本地KV存储 */
  storageLock   *sync.RWMutex
  localStorages map[int]*LocalStorage

}

/* 往第ShardID这个虚拟节点中，增加/修改 一个Key-Value映射*/
func (s *Slave) put(key string , value string , shardID int ) error

/* 在第ShardID这个虚拟节点中，查询一个Key*/
func (s *Slave) get(key string , shardID int)  (string,error)
```

```
/* 从第ShardID这个虚拟节点中，删除一个Key*/
func (s *Slave) del(key string, shardID int) error
```

- **并行的数据访问**
  - 使用读写锁, 包括访问shards数组，访问shardID->LocalStorage的映射, 访问LocalStorage都需要拿取对应的锁。

5. **Slave_Server**

   设计上，Slave应该只负责和KV存储相关的逻辑，而Slave_Server去负责包括更新配置，Primary-Backup 选举, 同步主备的任务

```
type ServerConf struct {
  Hostname string
  IP       string
  Port     int
  GroupID  int
}

type Server struct {
  zkClient *zk_client.Client
  path string /*记录当前服务器的zookeeper路径，用于删除节点*/

  localVersion int
  conf ServerConf

  primary        bool
  /*primary需要连接到所有的backup节点*/
  backupConfLock *sync.RWMutex
  backupServers []*RPCClient
  /*backup需要处理从primary来的同步请求*/
  syncReqs chan request

  /*本地存储*/
  slave *Slave
  /*rpc 服务*/
  rpcServer *RPCServer
}
```

- **Slave更新配置**
  - Slave_server在启动之后以100ms(预设)的间隔不断的向Master获取最新配置

    如果最新配置比本地更新，对于
    - 不再由自己管理的Shard，(Primary)将其发送给最新的管理者，并在成功后将本地的存储删除，并通知backup把EXPIRED状态的存储删除；（Backup）不做发送，但会将本地存储状态设置为EXPIRED。

        - 由自己管理的新的Shard，为其分配空间，并将状态设置为UNREADY, 一直等到他人通过TransferShard送达才可以访问。

          转移过程的**Commit Point**是最新的管理节点收到并处理完TransferShard请求的时候。在TransferShard请求发送之前，Shard一直由原来的管理节点负责，访问新节点会被拒绝；在TransferShard请求处理之后，Shard由新的管理节点负责，访问旧节点会被拒绝。

        - 如果在sendShard期间发生了crash，也应该保证一致性。backup在重新成为primary时，应该检查自己处于EXPIRED状态的本地存储，如果存在，仍然要试图把他发送给最新的owner.(尚未实现)
    - **Primary-Backup如何同步**(高一致性)
        - 引入新的RPC call.

          ```
          func (s* Slave)Sync(req request) error {
            /* no need to acquire lock when sync because primary already has lock */
            s.syncReqs <- req
            return nil
          }
          ```

          在Primary收到client发来的RPC请求并处理完成时，不会立刻返回，而是先通过Sync这个RPC call将请求转发给所有的backup.并且收到所有RPC call成功返回的结果时，才将原本的RPC请求正确返回。

          分析：可以保证在client收到请求成功的消息时，就算primary挂了，backup也可以最终将数据同步。
    - **在Master改变shard配置的时候如何对应的移动本地的localstorage**
        - 引入新的RPC call

          ```
          func (s *Slave) TransferShard(shardID int,storage map[string]string) error
          ```

          允许通过调用节点A的TransferShard将自己的一个localstorage传递给节点A。一般来说节点A是primary，A还会同步给他的所有backup
    - **Primary选举**
        - 利用zookeeper的Create(FlagEphemeral选项)，所有的Slave server在启动的时候尝试去在目标路径下创建一个临时节点，Zookeeper可以保证只有一个创建者可以成功创建。

          利用这个性质，让所有slave server在/node/slave_primary/[groupNum]路径下尝试创建节点，创建成功的自动为Primary节点，并开始进行和primary相关的逻辑（包括监控backup变化，转发请求等等）。创建失败的需要在/node/slave_backup/[groupNum]目录下去使用EphemeralSequential选项来创建节点，这个选项中所有人都会被赋予一个独立的序列号，必然会创建成功。需要寻找某一个group的primary/backup节点时，去查询对应的路径下的最新信息即可。
        - **重新选举** 所有的backup会监视primary路径下的节点的存在状况，如果primary节点突

然消失（primary机器和zookeepr连接断开，临时节点自动删除）,所有的backup将自己当前的backup节点数据删除，并再次尝试去primary路径下创建节点，如果创建成功，就升级为primary。如果创建失败，则重新按照backup来存在。

- **新的机器加入group** 如果一个新的机器加入了某个group，此时，这台机器上没有任何数据，应该由当前的master监控backup的变化，并将所有的shards转移到新出现的backup上。并注意，如果在reelection期间发生的重新选举，新启动的server不应该加入选举，因为他没有数据。

6. **Client**

   用户和KV存储系统交互的程序。实现了一个简单的REPL交互，允许用户通过get [key], put [key] [value], del [key]进行交互。

   当请求结果异常时，会进行三次配置更新，如果都失败了，则会告知用户现在数据不可用，需要重试。

7. **Admin**

   管理员添加新的group或者移除旧的group的接口，需要对应的group内的slave servers启动之后再运行，会调用master的RPC call来改动配置。

8. **测试**

   由于master, slave的独立性较好，因此对master动态更新配置和slave的并行put/get/del操作进行了单元测试。其他的特性集成度比较高，测试比较难写，暂时使用手动测试。

Zookeeper Install & Configuration process

- **构建zookeeper的docker image**

```
$ chmod +x bin/docker-entrypoint.sh
$ docker build zk-cluster/zk_docker  -t myzk
# may use proxy --build-arg http_proxy=http:ip:port
```

- **启动zookeeper本地伪集群**

```
$ docker-compose -f zk-cluster/zk-cluster.yml up -d
```

配置文件如下

```
version: '3.1'

services:
  zoo1:
    image: myzk
    restart: always
    hostname: zoo1
    ports:
      - 2181:2181
    environment:
      ZOO_MY_ID: 1
```

```
      ZOO_SERVERS: server.1=0.0.0.0:2888:3888;2181 server.2=zoo2:2888:3888;2181
server.3=zoo3:2888:3888;2181

  zoo2:
    image: myzk
    restart: always
    hostname: zoo2
    ports:
      - 2182:2181
    environment:
      ZOO_MY_ID: 2
      ZOO_SERVERS: server.1=zoo1:2888:3888;2181 server.2=0.0.0.0:2888:3888;2181
server.3=zoo3:2888:3888;2181

  zoo3:
    image: myzk
    restart: always
    hostname: zoo3
    ports:
      - 2183:2181
    environment:
      ZOO_MY_ID: 3
      ZOO_SERVERS: server.1=zoo1:2888:3888;2181 server.2=zoo2:2888:3888;2181
server.3=0.0.0.0:2888:3888;2181
```

maxClientCnxns, initLimit,syncLimit都采用默认，如果未来有性能或其他需求，再进行配置。

启动后即可在localhost:2181, localhost:2182, localhost:2183分别访问到三个zookeeper节点，需要相应的配置configuration.json文件来告知程序。

- 扩展到多机

  如果需要在多台机器上配置zookeeper，可以通过docker swarm，然后对应修改zk-cluster.yml

  同时修改项目根目录下的configuration.json, 将真实的zookeeper 配置填入，
  master,slave,client,admin启动时都会自动读取配置

```
{
    "zookeepers":[
        {
            "ip": "IP1/Hostname1",
            "port":"port1"
        },
        {
            "ip": "IP2/Hostname2",
            "port":"port2"
        },
        {
            "ip": "IP3/Hostname3",
            "port":"port3"
```

```
        }
    ],
    "zkRoot": "/node",
    "zkTimeout": 10
}
```

- **使用zoo.cfg配置, 使用zkServer.sh来启动zookeeper**。这种配置方法单机上配置和多机上配置需要做较多修改，因为实际生产肯定部署在多机上，就不使用单机进行配置。对应的zoo.cfg文件也有实现，写在了项目根目录下 /zk-cluster/zoo1,/zk-cluster/zoo2,/zk-cluster/zoo3内。

How To Run

Master

先在一个机器上运行Master

```
$ go run go/master_server.go [ip] [port] [hostname]
# ip: master_server所在机器的ip， port：提供服务的端口， hostname：该master的名称（用于在zookeeper中登记）
# example: go run go/master_server.go 127.0.0.1 4100 master1
```

Slave

在多台机器上运行多个slave

```
$ go run go/slave_server.go [ip] [port] [hostname] [groupID]
# ip: slave_server所在机器的ip, port: 提供服务的端口, hostname：该slave的名称（用于在zookeeper中登记）
# groupID 该slave属于的group号.同组内会分primary-backup
```

Client

用于用户发送put/get/del请求

```
$ go run go/client.go
# 提供repl接口，自动连接配置文件中的zookeeper，并根据登记的信息，寻找master和slave，发送请求。
```

Admin

管理员，用于添加新的groupID/删除现有的groupID, 在对应group的Slave启动后运行admin，将新的group加入master的服务配置列表。

```
$ go run go/admin.go [command] [groupIDs ...]
# command = "join-group" / "leave-group"
# groupIDs 是添加和删除的group号数组(不限长)
```

一键运行脚本

会启动一个master, 7个slave 按照 [2,2,2,1]分成四组, 并join到master中,并启动一个client.

```
$ ./run.sh
```

详见README.md