ShardCtrler相关结构体及方法

结构体

Clerk

```
type Clerk struct {
    servers []*labrpc.ClientEnd
    // Your data here.

    lastLeaderId int
    clientId     int64
    sequenceNum  int

    mu *sync.Mutex
}
```

Config

```
type Config struct {
    // 顺序号
    Num    int              // config number

    // 分片的位置信息
    // shards[3] = 2，表示分片序号为 3 的分片对应的负责集群是 Group2（gid = 2）
    Shards [NShards]int     // shard -> gid

    // 集群成员信息
    // Group[3] = ["ip1","ip2"]，表示 gid = 3 的集群 Group3 包含两台名称为 IP1，IP2
    Groups map[int][]string // gid -> servers[]
}
```

ControllerArgs

```
type ControllerArgs struct {
    ControllerType int
    ClientId int64
```

```
    SequenceNum int

    // join
    Servers map[int][]string // new GID -> servers mappings
    // leave
    GIDs []int
    // move
    Shard int
    GID   int
    // query
    Num int // desired config number
}
```

ControllerReply

```
type ControllerReply struct {
    // use for common
    Err         Err

    // use for query
    Config      Config
}
```

ShardCtrler

```
type ShardCtrler struct {
    mu      sync.Mutex
    me      int
    rf      *raft.Raft
    applyCh chan raft.ApplyMsg

    // Your data here.
    dead                int32 // set by Kill()
    lastClientOperation map[int64]ClientOperation
    notifyChans         map[int]chan ChanResult
    lastApplied         int

    configs []Config // indexed by config num
}
```

方法

Query

查询最新的Config信息

```go
func (ck *Clerk) Query(num int) Config {
    args := &ControllerArgs{}
    // 命令类型为 query
    args.ControllerType = ControllerTypeQuery
    // 需要的配置数
    args.Num = num

    // 发送并获取回复
    reply := ck.sendControllerCommand(args)
    logPrint("%v: Query config %v => %+v", ck.clientId, num, reply.Config)
    return reply.Config
}
```

Join

新加入的Group信息。

新加入的 `Group` 信息，要求在每一个 `group` 平衡分布 `shard` ，即任意两个 `group` 之间的 `shard` 数目相差不能为 `1` ，具体实现每一次找出含有 `shard` 数目最多的和最少的，最多的给最少的一个，循环直到满足条件为止。坑为： `GID = 0` 是无效配置，一开始所有分片分配给 `GID=0` ，需要优先分配； `map` 的迭代时无序的，不确定顺序的话，同一个命令在不同节点上计算出来的新配置不一致，按 `sort` 排序之后遍历即可。且 `map` 是引用对象，需要用深拷贝做复制（在server处理时有体现）

```go
func (ck *Clerk) Join(servers map[int][]string) {
    args := &ControllerArgs{}
    // 命令类型为 join
    args.ControllerType = ControllerTypeJoin
    args.Servers = servers

    // 发送 join 命令
    ck.sendControllerCommand(args)
}
```

Leave

哪些Group要离开。

移除 `Group` ，同样别忘记实现均衡，将移除的 `Group` 的 `shard` 每一次分配给数目最小的 `Group` 就行，如果全部删除，别忘记将 `shard` 置为无效的0。

```
func (ck *Clerk) Leave(gids []int) {
    args := &ControllerArgs{}
    args.ControllerType = ControllerTypeLeave
    args.GIDs = gids

    ck.sendControllerCommand(args)
}
```

Move

将Shard分配给GID的Group,无论它原来在哪

```
func (ck *Clerk) Move(shard int, gid int) {
    args := &ControllerArgs{}
    args.ControllerType = ControllerTypeMove
    args.Shard = shard
    args.GID = gid

    ck.sendControllerCommand(args)
}
```

发送四种命令

```
// 发送四种命令，query、join、leave、move
func (ck *Clerk) sendControllerCommand(args *ControllerArgs) *ControllerReply {
    ck.mu.Lock()
    defer ck.mu.Unlock()

    args.ClientId = ck.clientId
    args.SequenceNum = ck.sequenceNum

    for {
        reply := &ControllerReply{}
        // 通过 rcp 发送
```

```
        if ck.servers[ck.lastLeaderId].Call("ShardCtrler.HandleControllerCommand
            if reply.Err == OK {
                ck.sequenceNum++
                return reply
            } else if reply.Err == ErrNone || reply.Err == ErrTimeout {
                continue
            }
        }

        ck.lastLeaderId = (ck.lastLeaderId + 1) % len(ck.servers)
        time.Sleep(100 * time.Millisecond)
    }
}
```

处理 client 发来的四种命令

```
// 处理 client 发来的四种命令，query、join、move、leave
func (sc *ShardCtrler) executeControllerCommandWithoutLock(args ControllerArgs)
    switch args.ControllerType {
    case ControllerTypeMove:
        // use for tester
        lastConfig := sc.configs[len(sc.configs)-1]
        newConfig := Config{
            Num:    lastConfig.Num + 1,
            Shards: lastConfig.Shards,
            Groups: deepCopyMap(lastConfig.Groups),
        }
        newConfig.Shards[args.Shard] = args.GID
        sc.configs = append(sc.configs, newConfig)
    case ControllerTypeLeave:
        lastConfig := sc.configs[len(sc.configs)-1]
        newConfig := Config{
            Num:    lastConfig.Num + 1,
            Shards: lastConfig.Shards,
            Groups: deepCopyMap(lastConfig.Groups),
        }
        for _, gid := range args.GIDs {
            delete(newConfig.Groups, gid)
        }
        balanceShards(sc.me, &newConfig)
        sc.configs = append(sc.configs, newConfig)
    case ControllerTypeJoin:
        // 获取最新的 config
        lastConfig := sc.configs[len(sc.configs)-1]

```

```
30          newConfig := Config{
31              Num:    lastConfig.Num + 1,
32              Shards: lastConfig.Shards,
33              Groups: deepCopyMap(lastConfig.Groups),
34          }
35
36          // add groups
37          for gid, servers := range args.Servers {
38              // 将新加入的 server 记录到配置中
39              newConfig.Groups[gid] = servers
40          }
41
42          // balance shards
43          balanceShards(sc.me, &newConfig)
44
45          sc.configs = append(sc.configs, newConfig)
46      case ControllerTypeQuery:
47          // 如果该数字为 -1 或大于已知的最大配置数字，则 shardctrler 应回复最新配置。
48          if args.Num == -1 || args.Num >= len(sc.configs)-1 {
49              return sc.configs[len(sc.configs)-1]
50          } else {
51              return sc.configs[args.Num]
52          }
53      }
54
55      return Config{}
56  }
```