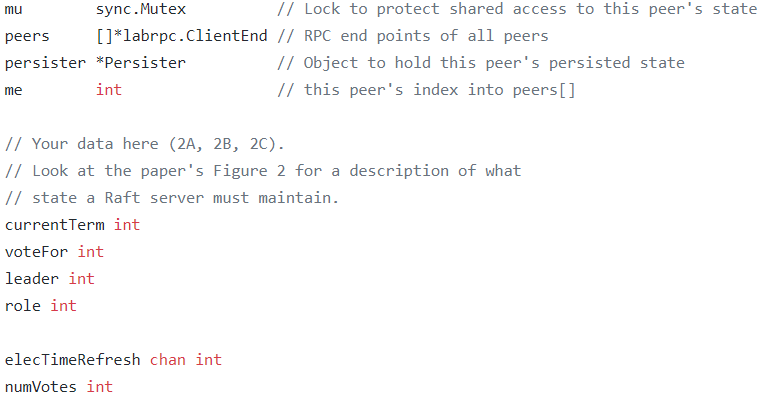
# KV存储

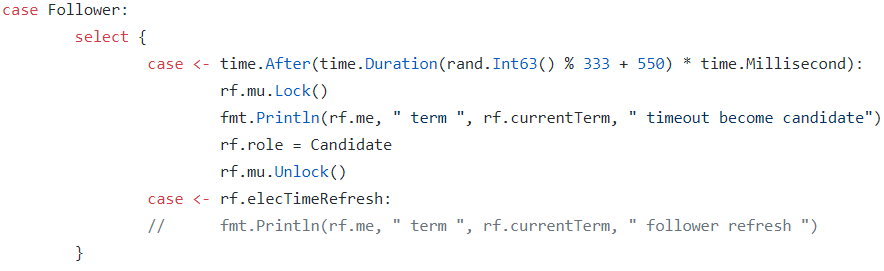
## raft

### 选主

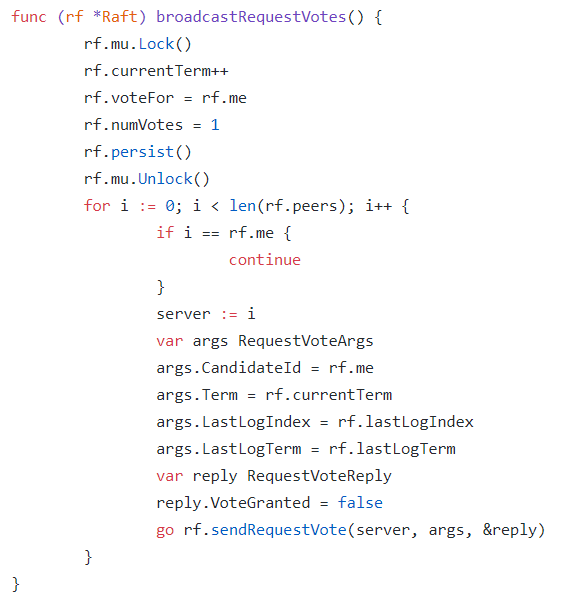
对于本阶段，每个节点需要维护其他节点的域名peers，currentTerm voteFor numVotes leader role elecTimeRefresh（是一个channel，用于收到心跳后刷新计时器状态）



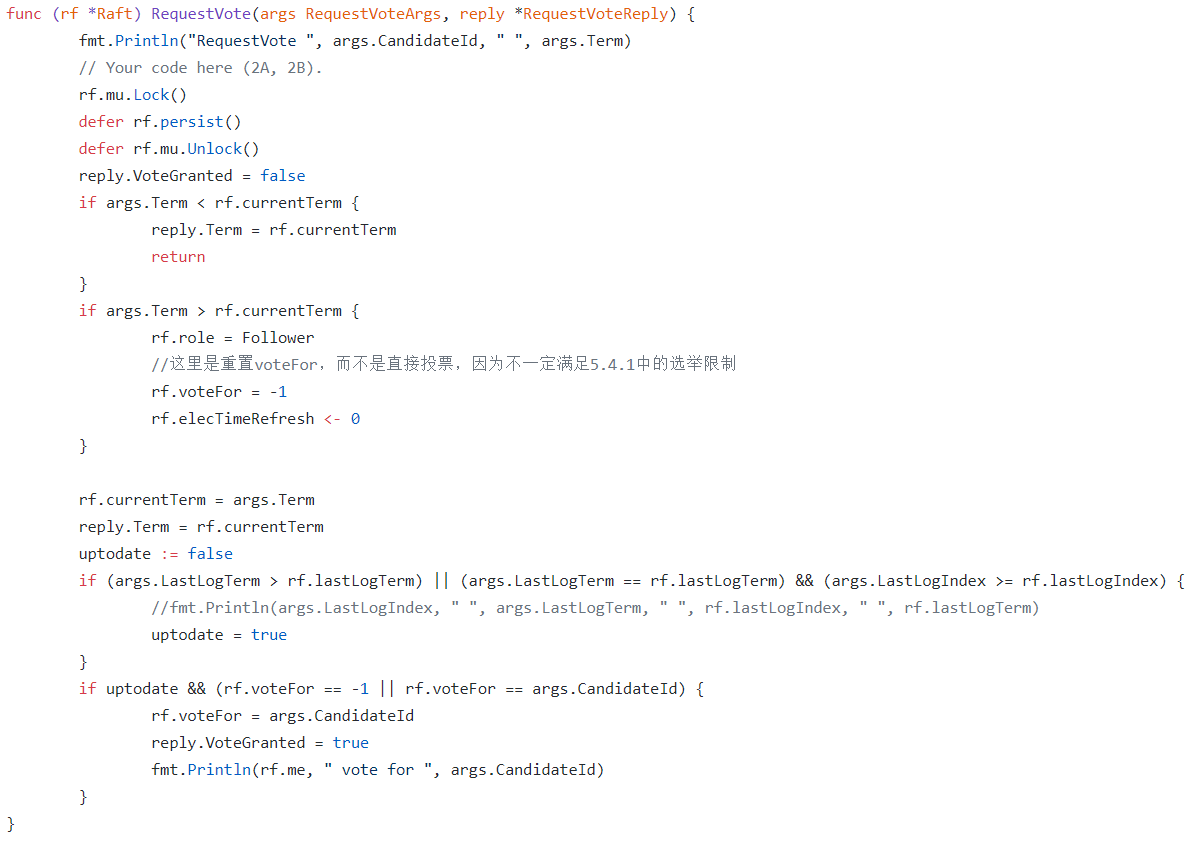
每台主机都会运行一个electionDaemon，主机初始role为follow，超时后转变为candidate，或收到心跳后刷新计时器状态。这部分实现如下：



主机状态变为candidate后，会先把自己的currentTerm++，voteFor自己，numVote++，然后广播sendRequestVote请求,请求中应该包括自己的term和自己的标识candidateId。（注意这里的每个请求都应该是go func()，因为对端可能宕机，这样本机也跟着阻塞） 这部分对应实现如下：



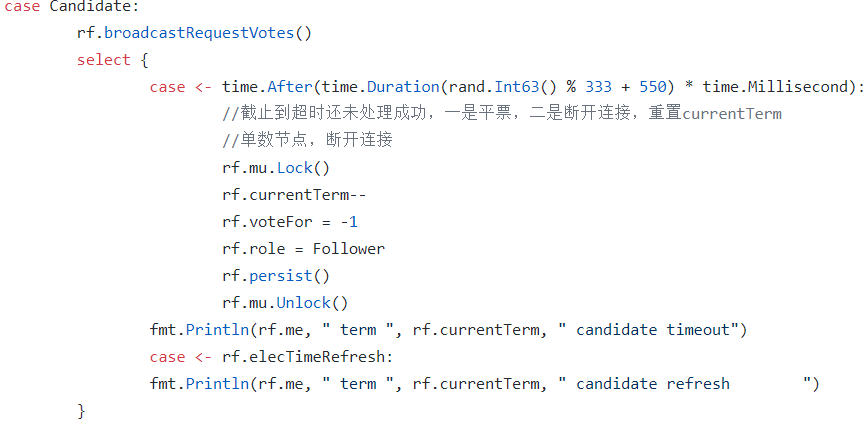
现在从对端的视角说说requestVote的处理，如果传过来的terrm小于自身currentTerm,则自己不会发生状态改变，此时把自身currentTerm及是否投票voteGrant=false返回给RPC的发起者sendRequestVote; 否则若currentTerm小于传过来的term，则修改自身currentTerm为Term，自身role转为follow，并刷新自身计时器（elecTimeRefresh），回送已经投票voteGrant=true



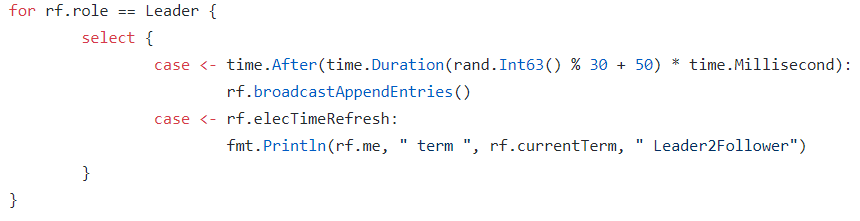
①对于实际发送RequestVote RPC的sendRequestVote,它会检测RPC返回的值，如果返回的term大于currentTerm,则修改自身currentTerm，转变状态并刷新计时器。若返回请求表情已经投票，则numVote++，若此时numVote多于一半，则转为leader状态，②并刷新自己处于candidate状态下的计时器（因为有可能选举超时）

分别如下



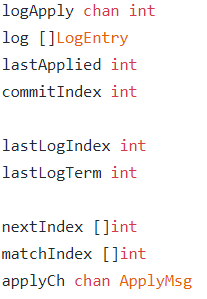


成为leader后会立即广播空的appendEntries(不含log)作为心跳包（之后每隔一段时间广播一次），同样会在sendAppendEntries检测回应，若其他主机term大于本身currentTerm,则更改自身role，currentTerm（leader变为folow后会立即写elecTimeRefresh,立刻转变状态）



### 日志同步

节点需要新增如下信息。LogApply用于通知后台日志线程有新的日志已提交，可以apply了；log存放了日志条目；lastApplied指最后应用的日志条目，在其小于commitIndex时，会不断应用；commitIndex指最新提交日志条目的index；lastLogIndex/lastLogTerm对应log中最后一条日志的index和term；nextIndex记录了对应每一个follow，leader节点要发给他们的下一日志条目的index(初始化为leader的lastLogIndex+1)；matchIndex记录了对应每个folow，它们自己的commitIndex（初始化为0）。这部分内容对应结构及初始化如下：

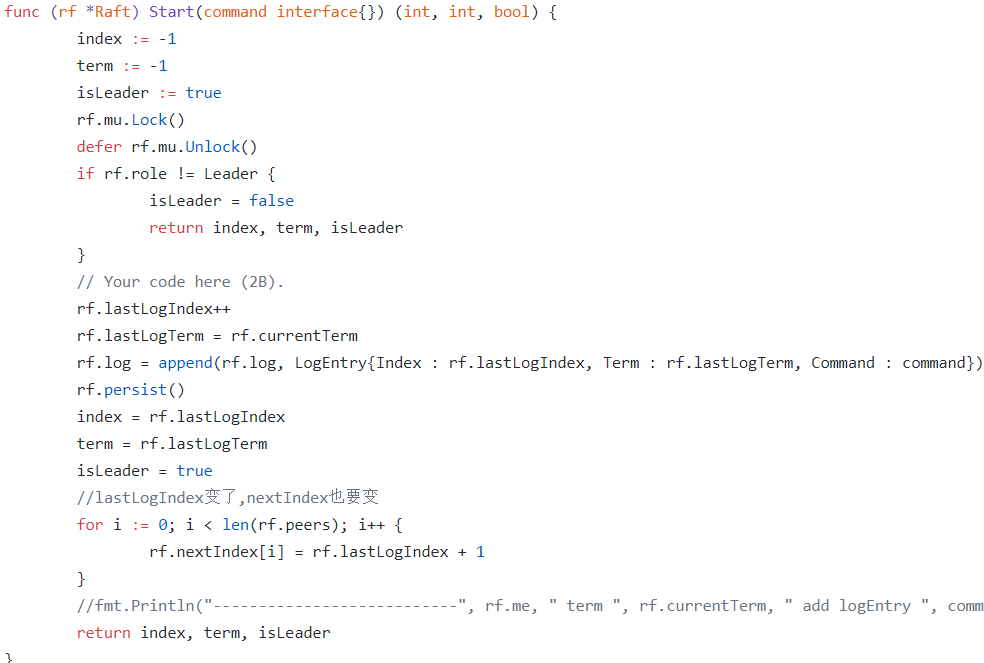




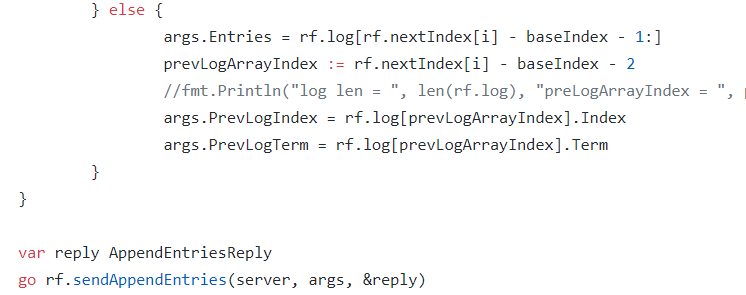
每个节点除了elecDaemon外，还会运行一个logDaemon，用于apply新的已提交的日志。应用就是发回给前端调用raft的数据库一个Msg，并指明可以被数据库执行的日志条目。



主节点leader的log如何更新，通过start将新来的日志条目追加到leader的log尾部，并同时更新leader的lastLogIndex,lastLogTerm(就是currentTerm)，还要更新与之相关的nextIndex



leader在广播心跳包appendEntries，对每个主follow，会发送对应follow的nextIndex及之后的日志条目，同时会附加对应nextIndex的前一日志条目的term和index



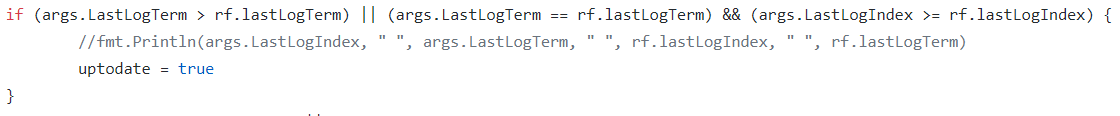
Leader会保证在sendAppendEntries中一定把日志条目发送给follow，否则会一直重试（每次发送心跳都会尝试）。发送成功则表明follow已经有了对应leader上nextIndex及之后的所有日志条目，此时会修改leader上面的nextIndex和matchIndex；发送失败过程可以优化，把nextIndex--改成 nextIndex= follow回送的NextIndex。 这部分实现如下：



在follow中对于leader发来的appendEntries，会首先检测周期要求。在此之后，follow会通过leader发来的preLogIndex和自己的lastLogIndex检测是否发生日志缺失，若缺失，则回送失败和实际要求的nextIndex；如果没有缺失，则通过preLogIndex和log条目比较来确定要把leader发来的日志条目放在log中的位置。也有可能比较到了follow的log[0].index发现follow还是落后，即preLogIndex还是大于此值，则回送失败和实际要求的nextIndex（commitIndex+1），这种情况发生在follow过于落后的情况，可能需要leader发送snapshot

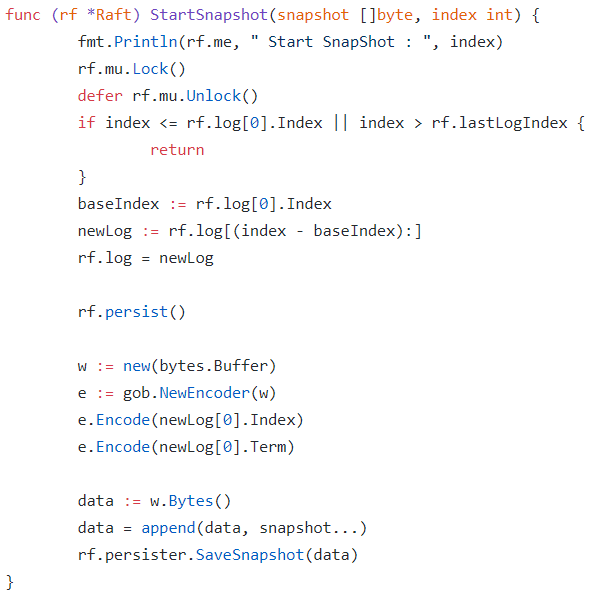


Raft对可以被选为leader的有个要求，即它的日志至少比大多数节点的日志新，即在follow的requestVote中满足如下要求才会给该candidate投票



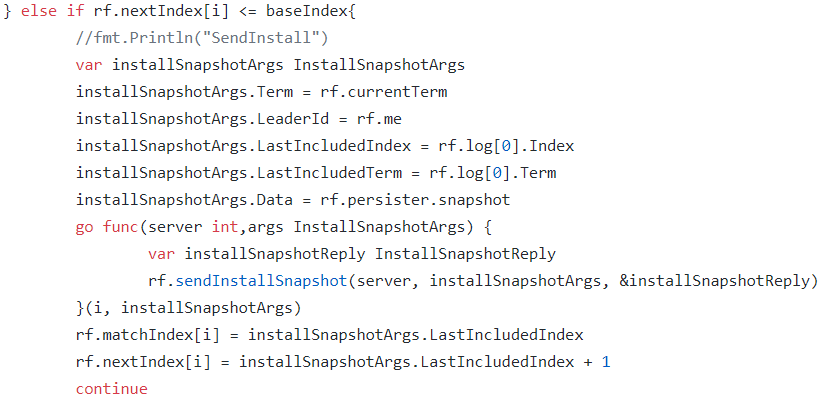
### 日志压缩

前端数据库调用startSnapshot进行日志压缩，参数包括数据库数据snapshot和index，raft的log中只需要保留index及之后的日志条目，然后需要把currentTerm voteFor log和snapshot信息持久化。这部分实现如下：

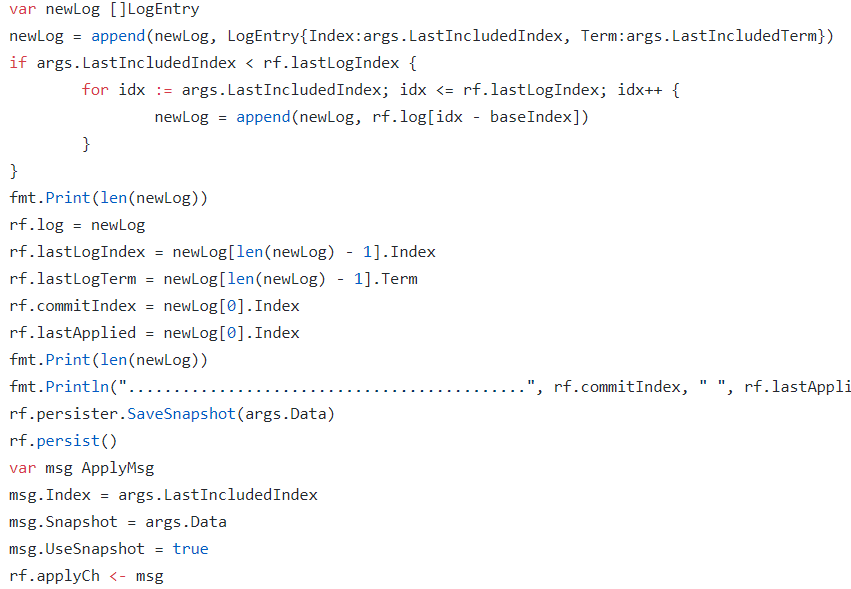


由于有了日志压缩，leader在向follow发送日志时可能会出现nextIndex < log[0].Index这种情况，此时需要由leader向follow发送快照。于是又有了sendInstallSnapshot和InstallSnapshot。

发送快照需要由leader告知follow自己的 之前压缩时的位置，也就是log[0].Index log[0].Term，还要有压缩时的数据库数据snapshot，还要包括raft节点状态currenTerm和LeaderId。sendSnapshot中



Follow在收到来自leader发来的快照后，可以根据快照立即构建自己的日志和数据库。构造数据库就是给前端数据库发送快照中的数据库数据。



### 持久化

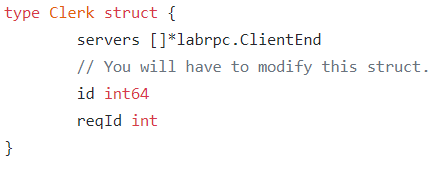
要持久化的内容包括currentTerm voteFor log，还包括在发生日志压缩时（StartSnapshot）的 压缩日志位置index和term和数据库数据snapshot 。在raft节点的上述数据发生变化时，便会把上述数据持久化。

Raft节点初始化时会先读取自己持久化的数据，以恢复自己的节点状态和log

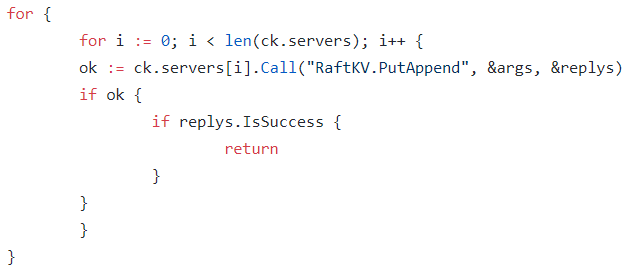
## KV存储

### 无分片的KV server

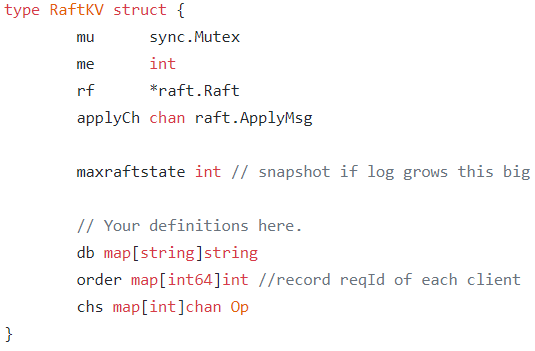
最基本的kv server，多主机之间通过raft保证只有一个leader且其他follow会逐渐和leader一致。客户端发给leader的命令不会立即写数据库，需要在对应该命令的log commit后才会执行，怎么commit？即该日志条目已复制到大多数节点上。每个客户端需要 自身ID和执行命令的reqId（保证过期的命令再到达leader时不会执行），client结构如下：



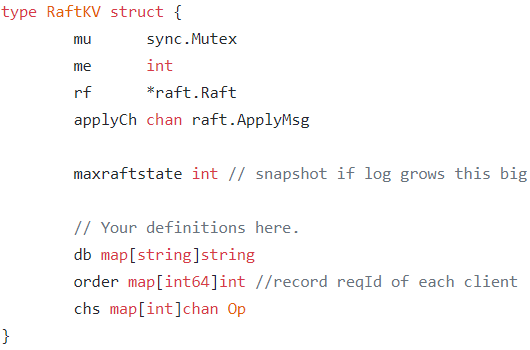
Client通过RPC向各个节点发送命令，直到返回成功。因为client并不知道哪个是leader



对于每个kv server，包括如下数据结构：



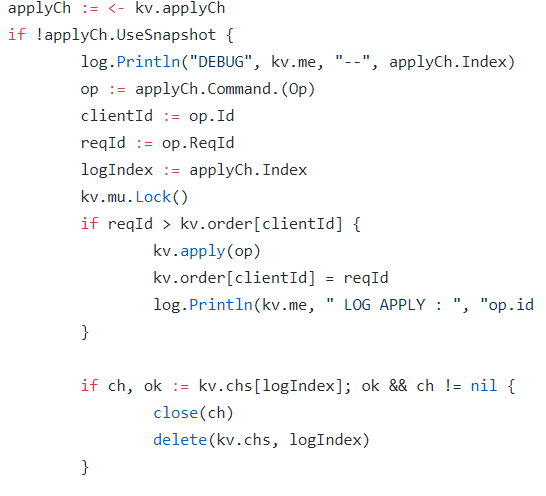
rf保存raft状态，db保存数据库数据，applyMsg用于前端数据库与后台raft交互，maxraftstate指会在持久化数据到一定大小时 进行日志压缩，order记录了每个client的reqId，具体实现如下：



对于客户发送来的RPC，需要先通过raft.Start写日志



每个kv server运行后，会在后台运行线程，检测applyMsg中是否有raft发过来的回送请求。可能是日志commit消息，这样如果reqId满足就写数据库；如果此时rf的持久化数据太大，就通过startSnapshot开始日志压缩；也可能是落后节点收到了快照，把快照中的数据库数据发送给前端数据库，这种情况就根据raft发来的数据回复db order等





### 无备份的分片KV

整体两个目的：

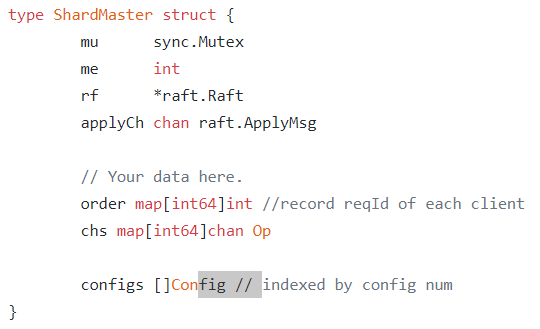
①负载均衡。每个服务器负责的分片应大概相等（相差不能超过1）

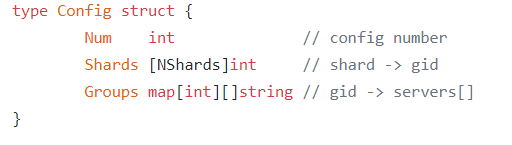
②集群中节点离开或加入时，发生移动的分片数应最少

ShardMaster需要保存分片配置，由于配置会变更，故配置数组中的没分配置都会有个编号，标记这是第几次配置。

配置信息 包含 分片到组号的映射，还要保存每个分组包含哪些机器。

一个分组对应一个raft集群





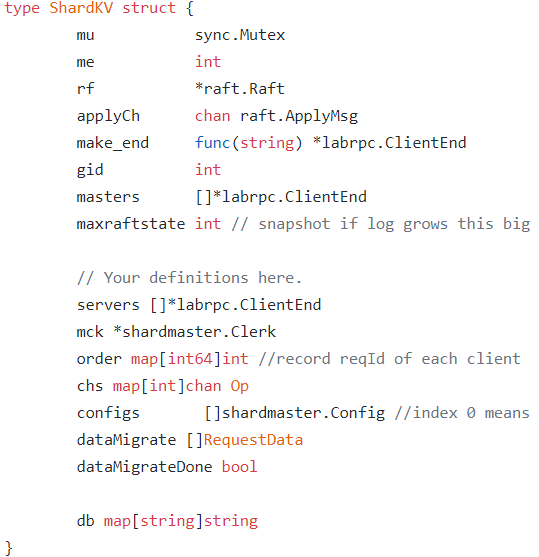
命令处理模式类似于上部分 无分片的KV server。先等日志commit再做具体处理。

每次发生 分组增加/减少，移动分片位置 时需要reblance。Reblance会先计算平均每个节点需要保存都少个分片，遍历所有节点，如果有节点分片富裕，则把他们拿出来，再次遍历每个节点，把富裕的分片分配到缺少分片的节点上。

### 通过raft管理备份的分片KV

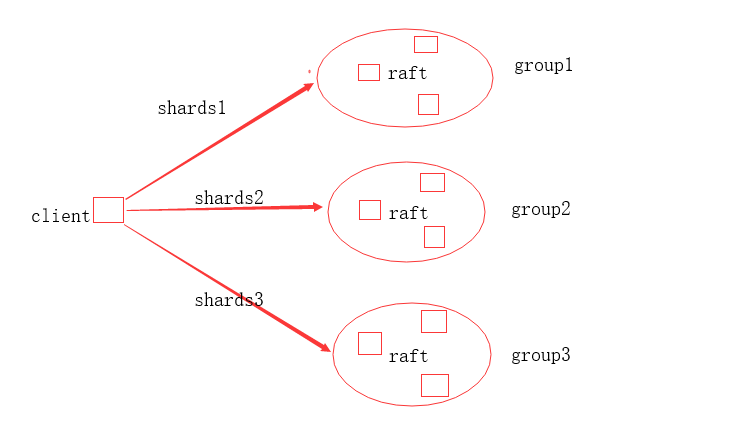
整体框架如上，先等日志commit再做具体处理，KV server会包含shardmaster(含raft)，gid，configures，db等信息。 逻辑层面一台主机分成3层，db层，处理客户的put/get请求；shardMaster层，处理分片位置，集群节点等信息，也可以与客户交互，处理流程见上一部分；最底层是raft，用于预写日志和日志同步。

kvServer的结构如下，它会包含这三层所有相关信息

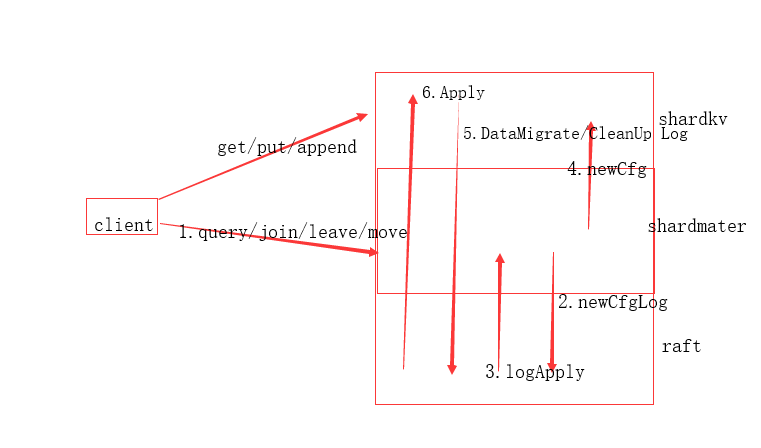


Kv server启动后会运行两个后台线程，一个用于常规的命令处理，这里的命令分为4种，put/get请求，数据库配置更新，数据迁移请求，垃圾回收；另一个用于每隔一段时间向shardMaster层请求最新配置。

整个框架的处理逻辑如下。KV分片管理，每个raft集群会管理一组分片。Client会持有所有分片分组信息，对于每个KV，它向对应集群发送get/put/append请求。



对于每个集群中的节点，分为shardkv层 ，ShardMaster层，raft层。Shardkv层负责管理db，shardmaster层负责管理分组分片信息cfg，raft用于预写日志和日志同步。 client会向所有节点发送 cfg变更信息（query/join/leave/move）,之后所有的处理流程如下：



对于shardkv层的DataMigrate，它会先请求数据（下图DataRequest），等请求的数据发过来（下图Data），便会预写DataMigrate日志，待日志apply之后便可以把数据应用到自己的数据库；本分片组也可能在配置变更后不再需要某些分片的信息，此时需要CleanUP来删除不要的分片信息。

