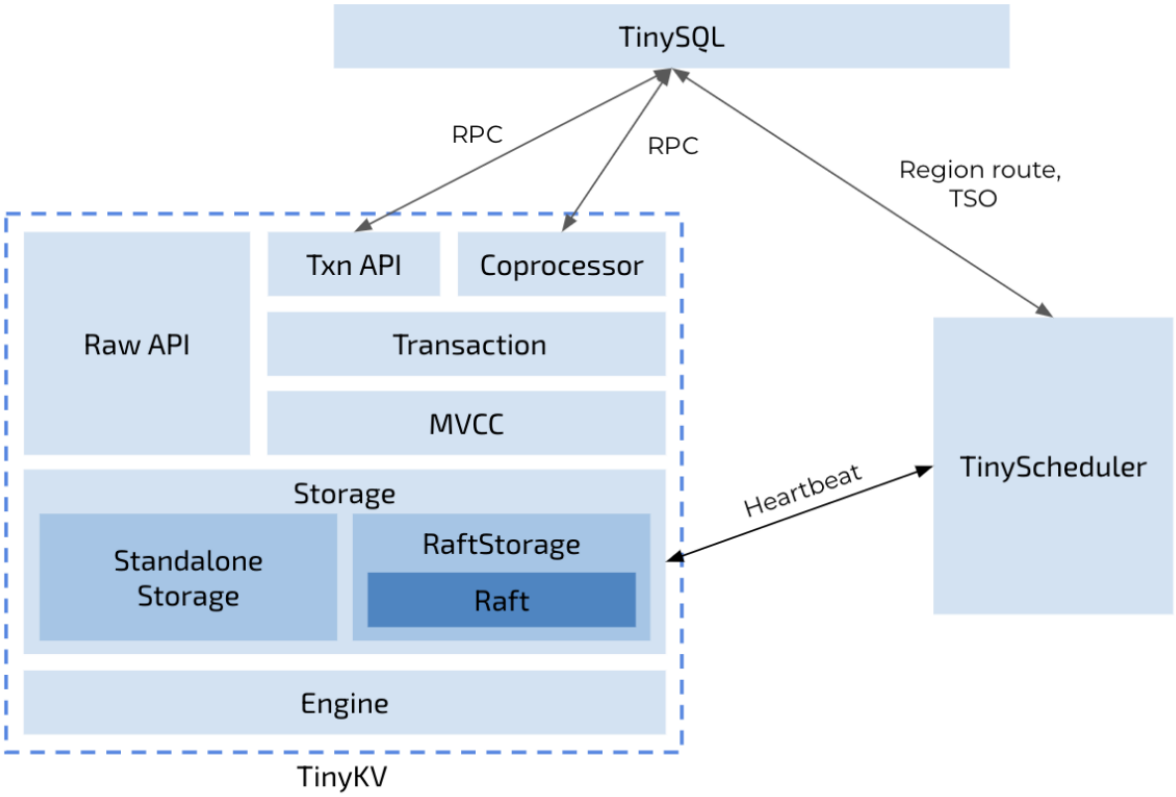
Raft 算法与代码框架介绍

华中科技大学 李响

TinyKV 的基本运行流程



上图官方给出的项目框架图，这幅图比较形象地体现了 TinyKV 的框架，~~但是对于第一次看到这幅图的人一般都是懵逼的，这其中就包括我，笑~~。一个完整的系统包括有强拓展能力、高可用性的分布式键值数据库 TinyKV，解析 Sql 语句并将语句转化成对应的读写请求的 TinySQL以及负责调度和管理的 TinyScheduler。我们这回需要实现的就是 TinyKV 部分。

TinyKV 项目的代码分为3层，**从上至下分别是 Server 层、Storage层以及最底层的 Engine层**。最上层的 Server 层负责接收读写请求，解析成对应的 raft 请求交给 Storage 层处理，并将 Storage 层返回的数据组织成对应的格式发送回 TinySQL。中间的 Storage 层将 raft 请求封装成日志进行同步到整个集群，并用合适的方式处理同步完成的请求，如将写请求写入底层的存储引擎 Engine 中，以及修改本身的配置等等。

为了帮助同学们更好地进行调试，在此梳理一下 TinyKV 的启动流程与运行流程。作为一个开源的具有教学性质的项目，本项目的启动流程分为两种：

- **常规真实的部署（你没看错 TinyKV 确实是一个可以使用的分布式数据库，~~不是教学模板式的花瓶，误~~）**，正常部署的函数入口是kv目录下的main文件；
- 模拟测试流程，本次项目中的所有测试走的都是这一套流程，入口是kv/test_raftstore/cluster.go 文件。

  此外，TinyKV 项目的测试主要分为以下两类：

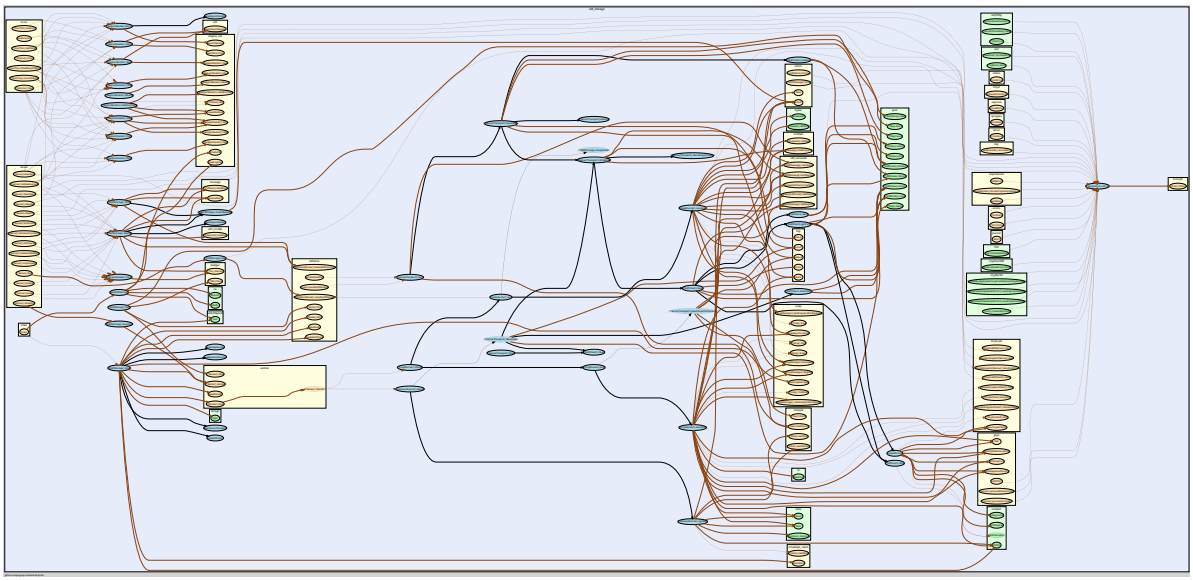
  - 功能性测试，本次 2A、3A 等模块使用，如果出现问题，调试和Debug非常方便与快速，可以使用平时调试的手段进行调试；
  - **模拟性测试**，2B、2C 等之后的几个模块采用，采用模拟集群的方式进行测试，通过观察输入与输出来判断正确性。这种测试由于其并发特点以及选举的随机性测试结果不可控，很难复现

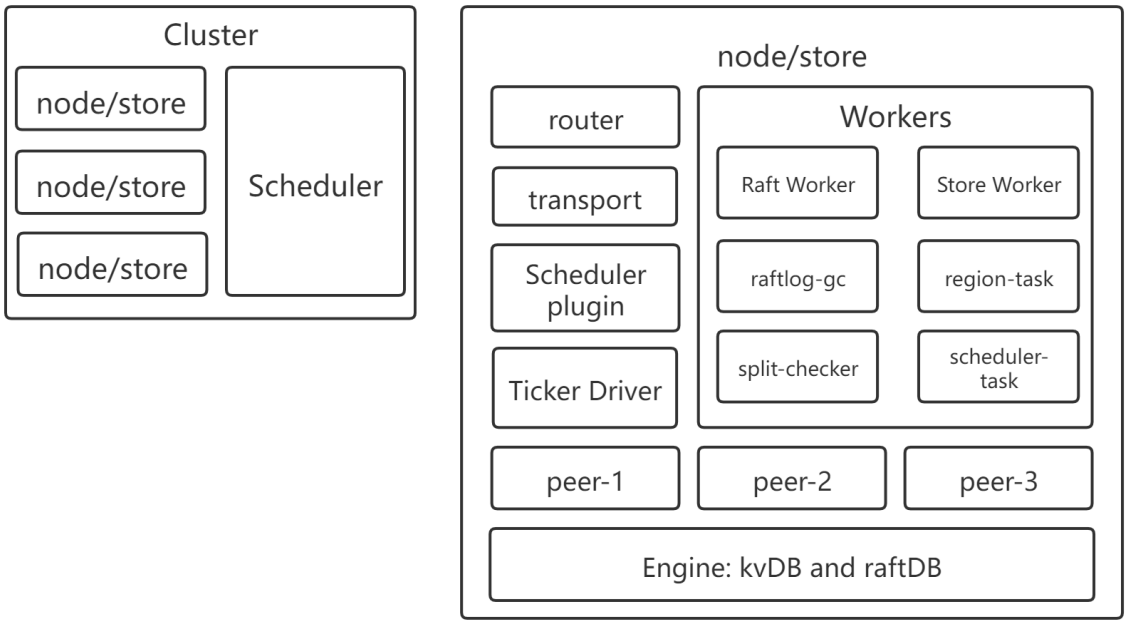
问题，主要的调试方式是通过日志进行调试。

模拟测试的具体的启动流程比较复杂这里推荐一个插件 `go-callvis`，安装和使用流程参见这篇文章：

https://www.cnblogs.com/realjimmy/p/13775842.html

下面是效果图，我会提供原图，~~还是蛮好玩的，~~误。





集群中分为两个部分，一个部分是**node组成的结点组**，一个**调度器**。结点的结构如右边所示，其组件比较多，大致分为以下三类：

- 最底层是 **BadgerDB** 分别存储键值对以及日志；
- 中间层的 peer 是 region 的一个封装，包含 raft 等一系列模块以及相关的处理逻辑，这部分将在下周的分享会中详细讲解；
- 最顶层包括 **6个workers** 以及一系列**其他组件**。这些 workers 以及组件的名称与主要作用如下：
  - `Ticker Driver` 系统时钟驱动器，驱动 peer 中时间触发的事件。
  - `router` 将其他结点发送的 msg 转发到 raft worker，然后再由 raft worker 分发到对应的 peer；此外，router 还将 store work 需要处理的信息转发到 store work。
  - `transport` 信息向外发送的接口，由于模拟测试并不发送实际的 RPC 请求，而是直接连接到 router 的一个map中，这样可以快速传递消息。

- `Raft Worker` 该线程内嵌包含 select 的循环，该循环的主要功能就是转发 router 通过 `raftCh` 发送的 msg，一旦收到消息就将其转发到对应的 Peer进行处理。该循环会一直运行，直到收到 `CloseCh` 信号。
- `Store Worker` 运行逻辑与 `Raft Worker` 类似，其功能是处理与 `Store` 相关的消息，并将 raft 模块的信息转发至 `Raft Worker` 。
- `raftlog-gc worker` 处理日志的压缩删除工作。
- `region-task worker` 负责快照 snapshot 生成、发送以及接收、应用。
- `split-checker worker` 发送区域分裂的指令，生成合适的 `split key`，并向中央管理器申请合适的 `RegionID`。
- `scheduler-task worker` 用于定期向调度器发送心跳，事实上 split 申请 `RegionID` 也是使用的这个途径。

以上是这些部件的简单作用，这些 worker 其作用不是本次重点，有兴趣可以自己去对应的文件中阅读代码，只需要知道 peer 使用 task 向这些 worker 发送任务，驱动其开始工作。详细之后的分享也许会有涉及。

raft 算法介绍

推荐一篇博文，讲解的非常详细 https://www.codedump.info/post/20180921-raft/ 有兴趣可以去阅读一下。

Raft算法是一个强一致性协议，由leader节点来处理一致性问题。leader节点接收来自客户端的请求日志数据，然后同步到集群中其它节点进行复制，当日志已经同步到超过半数以上节点的时候，leader节点再通知集群中其它节点哪些日志已经被复制成功，可以提交到raft状态机中执行。

Raft算法将要解决的一致性问题分为了以下几个子问题：

- leader选举：集群中必须存在一个leader节点。
- 日志复制：leader节点接收来自客户端的请求然后将这些请求序列化成日志数据再同步到集群中其它节点。
- 安全性：如果某个节点已经将一条提交过的数据输入raft状态机执行了，那么其它节点不可能再将相同索引的另一条日志数据输入到raft状态机中执行。

Raft算法基础

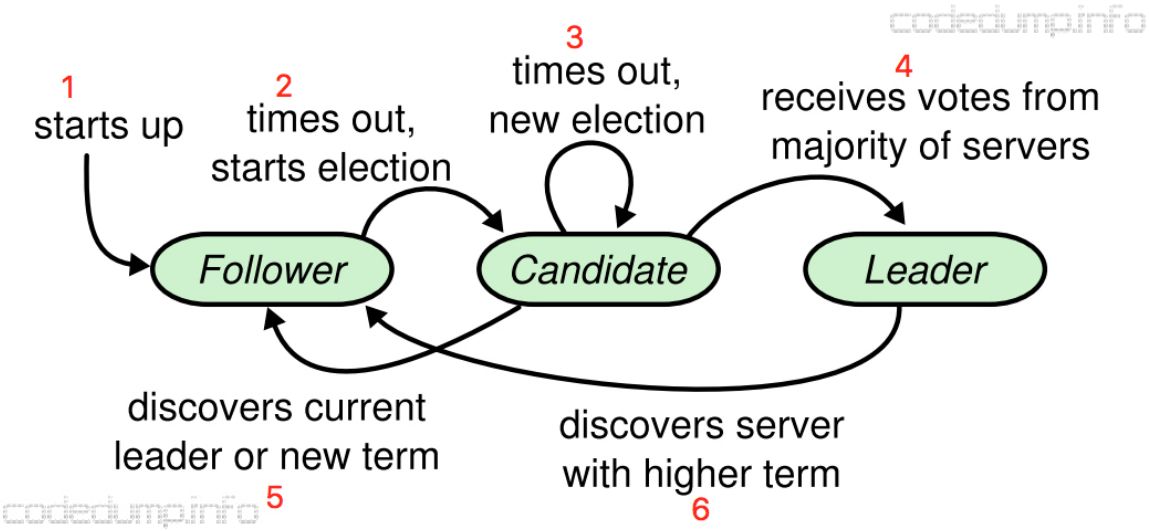
在Raft算法中，一个集群里面的所有节点有以下三种状态：

- Leader：领导者，一个集群里只能存在一个Leader。
- Follower：跟随者，被动接收并同步 Leader 发送的日志。0
- Candidate：参选人，一个节点切换到这个状态时，将开始进行一次新的选举。

每一次开始一次新的选举时，称为一个“任期”。每个任期都有一个对应的整数与之关联，称为“任期号”，任期号用单词“Term”表示，这个值是一个严格递增的整数值。

节点的状态切换状态机如下图所示。

推荐一个演示网页 https://raft.github.io/



leader选举

现在来讲解leader选举的流程。

raft算法是使用心跳机制来触发leader选举的。

在节点刚开始启动时，初始状态是follower状态。一个follower状态的节点，只要一直收到来自leader或者candidate的正确RPC消息的话，将一直保持在follower状态。leader节点通过周期性的发送心跳请求来维持着leader节点状态。每个follower同时还有一个选举超时（election timeout）定时器，如果在这个定时器超时之前都没有收到来自leader的心跳请求，那么follower将认为当前集群中没有leader了，将发起一次新的选举。

发起选举时，follower将递增它的任期号然后切换到candidate状态。然后通过向集群中其它节点发送RequestVote RPC请求来发起一次新的选举。一个节点将保持在该任期内的candidate状态下，直到以下情况之一发生。

- 该candidate节点赢得选举，即收到超过半数以上集群中其它节点的投票。
- 另一个节点成为了leader。
- 选举超时到来时没有任何一个节点成为leader。

下面来逐个分析以上几种情况。

第一种情况，如果收到了集群中半数以上节点的投票，那么此时candidate节点将成为新的leader。当一个candidate节点赢得选举成为leader后，它将发送心跳消息给其他节点来宣告它的权威性以阻止其它节点再发起新的选举。

第二种情况，当candidate节点等待其他节点时，如果收到了来自其它节点的心跳请求，**同时做个请求中带上的任期号不比candidate节点的小，那么说明集群中已经存在leader了**，此时candidate节点将切换到follower状态；但是，如果该RPC请求的任期号比candidate节点的小，那么将拒绝该RPC请求继续保持在candidate状态。

第三种情况，当选举超时到来时，如果集群中还没有一个leader存在，那么candidate节点将继续递增任期号再次发起一次新的选举。这种情况理论上可以一直无限发生下去。

为了减少第三种情况发生的概率，**使用随机过程将选举超时时间进行一定的随机化，以防止重复性的选举失败。**

日志复制

日志复制的流程大体如下：

1. 每个客户端的请求都会被重定向发送给leader，这些请求最后都会被输入到raft算法状态机中去执行。

2. leader在收到这些请求之后，会首先在自己的日志中添加一条新的日志条目。
3. 在本地添加完日志之后，leader将向集群中其他节点发送 AppendEntries 请求同步这个日志条目，当这个日志条目被成功复制之后，leader节点将会将这条日志输入到raft状态机中，然后应答客户端。

一条日志一旦被 **leader 同步到集群中超过半数的节点**，由于选举过程中对于日志的判断就可以保证这条日志一定能被成功同步，因此 Leader 就会提交该日志。如果一条日志已被提交，那么在这条日志之前的所有日志条目也是被提交的，包括之前其他任期内的leader提交的日志。

需要注意的是，Raft算法保持着以下两个属性：

- 如果两个日志条目有相同的索引号和任期号，那么这两条日志存储的是同一个指令。
- 如果在两个不同的日志数据中，包含有相同索引和任期号的日志条目，那么在这两个不同的日志中，位于这条日志之前的日志数据是相同的。

这两条性质保证了同步过程中，从后向前只需要检查到一条日志是相同的就能确定前面都是相同的。

安全性

前面章节已经将leader选举以及日志同步的机制介绍了，这一小节讲解安全性相关的内容。

选举限制

raft算法中，并不是所有节点都能成为leader。一个节点要成为leader，需要得到集群中半数以上节点的投票，而一个节点会投票给一个节点，其中一个充分条件是：**这个进行选举的节点的日志拥有与本节点相同或者更新的日志。**之所以要求这个条件，是为了保证每个当选的节点都有当前最新的数据。

这个限制保障了所有 Committed 的日志都是可靠的，不会被覆盖的。

集群成员变更

在上面描述Raft基本算法流程中，都假设集群中的节点是稳定不变的。但是在某些情况下，需要手动改变集群的配置。

安全性是变更集群成员时首先需要考虑到的问题，任何时候都不能出现集群中存在一个以上leader的情况。为了避免出现这种情况，**每次变更成员时不能一次添加或者修改超过一个节点**，集群不能直接切换到新的状态。

raft采用将修改集群配置的命令放在日志条目中来处理，这样做的好处是：

- 可以继续沿用原来的AppendEntries命令来同步日志数据，只要把修改集群的命令做为一种特殊的命令就可以了。
- 在这个过程中，可以继续处理客户端请求。

在成员变更过程中，论文中提到过使用新老配置共同决定的方式防止出现多个领导人，这一点在实现过程中主要是通过选举过程实现的。配置的修改主要分为加入节点和删除节点两种，在此系统中，不能一次添加或者删除超过一个节点。对于这两种方式处理是不同的，具体如下：

添加新节点到集群中

添加一个新的节点到集群时，需要考虑一种情况，即新节点可能落后当前集群日志很多的情况，在这种情况下集群出现故障的概率会大大提高。

因此添加进来的新节点首先将不加入到集群中，而是等待数据追上集群的进度后再加入集群。

这里有两种实现方案：

- 第一种，**也就是说未同步数据的节点是暂时不参加选举的，集群中的节点按照旧配置进行选举**，当节点其赶上进度后，将其提升成可以进行选举的结点，这种实现方式**类似与 etcd 中 learner 的角色。**

  这种实现方式中，最重要的是如何**通知集群中的节点（包括新加入的节点）新节点已经被提升**，这也是一种配置变更，不仅需要使用新的消息进行通知，还需要考虑清楚如果本消息丢失可能出现的问题。比如可能存在以下这种情况：

  > **新节点不知道自己已经被提升了，是合法的选举人，需要正常进行投票。在这种情况下，集群可能无法选出合法的 leader。**因此，虽然理论上 leader 不应该向 learner 发送选举投票请求，learner 也不应该处理投票请求。但是为了解决如上所述的情况，如果 learner 收到选举请求，那么需要进行正常投票。

  由于这种实现方式本身较为复杂，不仅需要增加 learner 这种角色，还可能需要新增消息进行处理。**本次不推荐使用这种方式，**但是这种方式事实上是更加稳定的。

- 第二种是，集群中的节点按照新的配置进行选举，**新加入的、未同步数据的节点不能发起选举，但是能正常投票，在进行日志判断时，需要认为任何别的节点都比本地新，其他对于先来后到等原则都不变，**直到日志完成同步可以正常加入集群。

  在这样的设计下，在使用快照（或者不使用）日志同步完成后，**自行提升其角色即可。**这种方式虽然需要增加一个状态进行一定的判断（**其实也可以不增加，因为对于新加入的节点就算发起了选举，由于其日志的落后性也是不可能获胜的**），但是不需要设计新的消息进行通信。

这里推荐使用第二种方式实现，这种方式实现起来比较简单，在实际实现过程中基本没有特殊的处理。只需要注意新加入的节点应该能够正常选举即可。

删除当前集群的leader节点

当需要下线当前集群的leader节点时，leader节点将发出一个变更节点配置的命令，只有在该命令被提交之后，原先的leader节点才下线，然后集群会自然有一个节点选举超时而进行新的一轮选举。

此外存在一种特殊的情况，如果某个节点在一次配置更新之后，被移出了新的集群，**但是这个节点又不知道这个情况**，那么这个节点可能会无限制地不停发起选举。

为了解决这个问题，**可以使用 `Prs` 映射对消息进行过滤，当节点被移除后，`Prs` 应该不存在对应的 `Progress` 量，直接丢弃此类消息即可。**

日志压缩

日志数据如果不进行压缩处理掉的话，会一直增长下去。为此Raft使用快照数据来进行日志压缩，具体的就不论述了，可以参见论文。

raft 代码框架介绍

可以参考一下 etcd 的代码实现，我们这次实现的系统和设计的结构都与之高度相似。这里推荐一篇代码解析的文章，非常详细。

https://www.codedump.info/post/20180922-etcd-raft/

与 raft 模块高度相关的主要是三个文件 raft.go/log.go/rawnode.go。

rawnode 封装了 raft 模块对外提供的一系列接口，通过这些接口可以驱动 raft 模块进行消息的处理与同步，也可以通过接口读取 raft 模块的一些信息。

这些接口包括与驱动 raft 相关的 Tick、Step，与日志同步相关的 Propose，以及与状态保存和持久化等相关的 Ready 与 Advance 等几个方法。

函数	作用
Tick	定期调用以驱动raft进行消息处理、选举操作。
Campaign	强制驱动节点竞选，进入候选人状态。
Propose	提交日志，可能会返回错误。
ProposeConfChange	提交配置变更。
Step	驱动 raft 处理接收到的消息。
Ready	这里是核心函数，将返回一些需要进行处理的数据，这些数据打包成Ready。这些数据需要进行持久化或者其他处理。
Advance	当上层将Ready数据处理完毕后，调用该方法进行收尾，并告知raft模块之前的ready数据已经处理完成。

- 上层应用通过定期调用 Tick 向 raft 模块传递时钟信号驱动 raft 模块中的定时器，继而驱动raft 模块的运行。
- 当 Peer 接收到 raft 的message 后，通过 Step 方法传递给 raft 模块进行处理。
- 当 Peer 接收到 读写请求 或者 涉及到配置更改 等需要进行同步的操作时，调用 Propose 以及 ProposeConfChange 将日志添加到 raft 模块，让其进行同步。
- 每隔一段时间，系统会调用 Ready 相关的方法取出需要进行持久化的数据进行处理，并在处理完后调用 Advance 修改 raft 的状态，并在合适的时候压缩其日志。

raft 封装了一些在日志同步、选举、成员转换过程中的一些状态量以及raft的日志。

log 则是对于 raft 日志的一个封装。

log 模块

log 是 raft 模块的一个核心结构，其中缓存了日志信息，由于 raft 模块使用快照的方式进行日志的压缩，实际上 raft 的日志分为两部分，前半部分是已经压缩成 snapshot 已经删除的日志，后半部分是日志条目组成的日志，日志的构成如下图所示：

```
//  snapshot/first.....applied....committed....stabled.....last
//  --------|------------------------------------------------|
//                            log entries
```

committed 就是代表已经提交的日志；**applied 意思是应用，该处表示已经将日志中的读写操作完成，已将键值对写入 BadgerDB 中**；**stabled 则是代表是否已经持久化**，在这个量之前的日志都已经存储进底层的 DB 中。

再来看 Raftlog 提供的接口，主要的作用是将底层的量提供给上层，以及对日志进行截断删除等一系列的操作。这一部分需要对着测试函数进行编程难度不大，具体的编写流程我就不详细说了，有些方法的编写需要阅读对应的 Storage 的接口，需要对异常进行处理。

需要注意的 **raftlog 的初始化过程**，也就是 NewLog 的实现，由于 raft 模块可能是重启的，其中的量不一定是初始量，**需要调用 storage 中的一些方法获取已经持久化的量进行初始化。**

这一部分涉及到部分 storage 的接口与结构，当初我在做的时候也是踩了很多雷，尤其是其中 applied 的赋值问题，这里的实现和 etcd 中是一样的，也就是此处对其进行非常简单的初始化，通常是 firstindex - 1。

但是在raft初始化时，需要使用 config 中的 Applied 变量进行修正。否则会造成比较严重的问题，该问题在 2A 通常看不出来，但是在之后的重启中会出现严重隐患。

```
type RaftLog struct {
    storage Storage
    applied uint64
    committed uint64
    stabled uint64
    entries []pb.Entry
    pendingSnapshot *pb.Snapshot
}

func newLog(storage Storage) *RaftLog
func (l *RaftLog) unstableEntries() []pb.Entry
func (l *RaftLog) nextEnts() (ents []pb.Entry)
func (l *RaftLog) LastIndex() uint64
func (l *RaftLog) FirstIndex() uint64
func (l *RaftLog) Term(i uint64) (uint64, error)
/* 调用 storage.FirstIndex() 即可知道当前截断的index */
func (l *RaftLog) maybeCompact()
```

raft 模块

log 之上就是 raft 模块，该模块主要负责选举、日志复制以及成员变更等功能，raft 组中不同的节点通过 Msg 进行通信，因此本节最重要的就是对于 Msg 的处理。我们先看 raft 消息的结构体，主要成员如下：

```
type Raft struct {
    id   uint64
    Term uint64
    Vote uint64

    RaftLog *RaftLog
    Prs map[uint64]*Progress
    State StateType
    votes      map[uint64]bool
    heartbeats map[uint64]bool
    msgs []pb.Message
    Lead uint64

    heartbeatTimeout int
    electionTimeout int
    heartbeatElapsed int
    electionElapsed int
    leadTransferee uint64
    PendingConfIndex uint64
}

type Message struct {
    MsgType              MessageType
    To                   uint64       // 接收节点ID
    From                 uint64       // 发送节点ID
    Term                 uint64       // 任期号
    LogTerm              uint64       // 日志任期号
    Index                uint64       // 日志索引号，在不同通信处理中有不同的含义
    Entries              []*Entry     // 日志
    Commit               uint64       // 已经提交的日志索引号
    Snapshot             *Snapshot    // 快照数据
    Reject               bool         // 是否拒绝
```

```
}

const (
    MessageType_MsgHup MessageType = 0
    MessageType_MsgBeat MessageType = 1
    MessageType_MsgPropose MessageType = 2
    MessageType_MsgAppend MessageType = 3
    MessageType_MsgAppendResponse MessageType = 4
    MessageType_MsgRequestVote MessageType = 5
    MessageType_MsgRequestVoteResponse MessageType = 6
    MessageType_MsgSnapshot MessageType = 7
    MessageType_MsgHeartbeat MessageType = 8
    MessageType_MsgHeartbeatResponse MessageType = 9
    MessageType_MsgTransferLeader MessageType = 11
    MessageType_MsgTimeoutNow MessageType = 12
)
```

这个结构体中的成员非常杂乱，每种消息需要使用到的成员也各不相同，为了方便编程与理解，我会列出每种消息需要使用的成员，同时描述一下几个比较重要的消息的处理流程。

MsgHup

触发选举流程，通知节点进行选举。该消息仅供本地使用，不用于结点之间的通信。通常的调用方式是直接使用 Step 传递一个 MsgHup 类型的信息给 raft 模块处理，不需要经过消息的发送流程，因此理论上只需要使用 MsgType 这一个类型即可。

如果将本地消息发送到其他节点，该节点应该直接将该信息 drop 丢弃。

成员	类型	作用
MessageType	MsgHup	不用于节点间通信，仅用于发送给本节点让本节点进行选举
from	uint64	本节点ID
to	uint64	消息接收者的节点ID

MsgBeat

触发 leader 发送心跳。该消息仅供 leader 进行本地使用，其作用仅仅是起到通知 leader 其心跳周期耗尽，需要向其他节点发送心跳保持连接。该处的使用这种方式而不是使用函数调用的方式，更多是为了统一处理的流程，简化接口的设计，减少阅读者的理解成本。

如果将本地消息发送到其他节点，或者是非 leader 节点接收到此消息，应该直接将该信息 drop 丢弃。

成员	类型	作用
MessageType	MsgBeat	仅供 leader 本地使用，作用是起到通知 leader 其心跳周期耗尽，需要向其他节点发送心跳保持连接。
from	uint64	本节点ID
to	uint64	消息接收者的节点ID

MsgPropose

上层将请求封装成日志 Entries 后，使用该消息通知 leader 将日志添加到本地日志序列中。该消息仅供 leader 进行本地使用。

需要注意的是，不同类型的节点处理本消息的方式存在不同。

本系统中，模拟测试中不存在将请求发送给非 leader 的情况（至少不会一直发送给非 leader 节点）。该消息是本地使用的，如果本地节点非 leader 节点，或者将其转发至其他节点，该消息应该直接被忽略，不做任何处理或者返回一个对应的 error。

在 etcd 中，如果是 follower 追随者接收到此类信息，如果 leader 存在，需要转发给对应的 leader。

对于 leader 而言，需要将 Entries 中的日志加到本地的日志序列。需要注意的是，如果集群中正在进行领导权转移 Transfer Leader时，是不允许进行日志的提交的；而且，每次只能处理最多一个 配置变更 Conf Change 请求，因此需要检查合法性再进行日志的附加操作，合法性检查如下：

- 检查 leadTransferee 是否为 None，该量表示领导权的转移对象，为 None 才能进行日志附加。
- 检查 Entries 中是否存在 `EntryType_EntryConfChange` 类型的日志条目，如果存在，则表明需要进行配置的变更。但是，该系统同时不能处理多个配置变更请求，所以当 PendingConfIndex 大于当前已经应用 **applied** 的日志编号时（**PendingConfIndex 保存的是最新一次配置变更日志的编号**），表示上一次的配置变更还未处理完成，需要将该日志删除或置为空。否则，则将 PendingConfIndex 置为当前日志的编号，用于之后的判断。

在合法性检查完成后，即可进行后续的操作。将日志附加到本地，然后向集群中广播 MsgAppend 消息，向其他节点同步日志，具体的发送细节详见 MsgAppend 节讲解。

此外，这里有一个点需要特别注意，当集群中只有一个节点时，是不允许对自己广播 MsgAppend 消息的。在这种情况下，需要在此处 **显式修改 committed** 以通过 2A 的测试。

成员	类型	作用
MsgType	MsgPropose	上层将请求封装成日志 Entries 后，使用该消息通知 leader 将日志添加到本地日志序列中。该消息仅供 leader 进行本地使用。
From	uint64	本节点ID
To	uint64	消息接收者的节点ID
Entries	Entry	日志条目数组

MsgAppend

该消息是本节的重点消息，作用是向集群中的节点同步日志数据（snapshot也用于同步数据，但是其方式存在不同，详见 MsgSnapshot）。本节主要分为两个部分，分别是如何发送 MsgAppend 消息以及如何处理接收到的 MsgAppend 消息。

发送

首先，leader 会维护一个名为 `Prs` 的 map 型变量（节点ID：`Progress`），`Progress`中包含 `Match` 以及 `Next`，这两个量分别表示每个节点已同步日志的编号 以及 下一个需要同步的日志的开始编号。需要发送进行同步的日志从 `Next` 代表的日志开始。

需要注意的是，如果需要同步的节点是新加入的，或者与 leader 存在较长时间的网络隔离，可能存在日志已经压缩的情况。在这种情况下，需要同步的日志本身就已经被删除了，就需要使用 snapshot 快照数据进行同步。判断是否需要快照同步，需要检查索引编号为 `Next - 1` 的日志是否存在即可。

当不需要进行快照同步，就将对应的日志封装进 Entries 中发送即可，需要使用的成员如下：

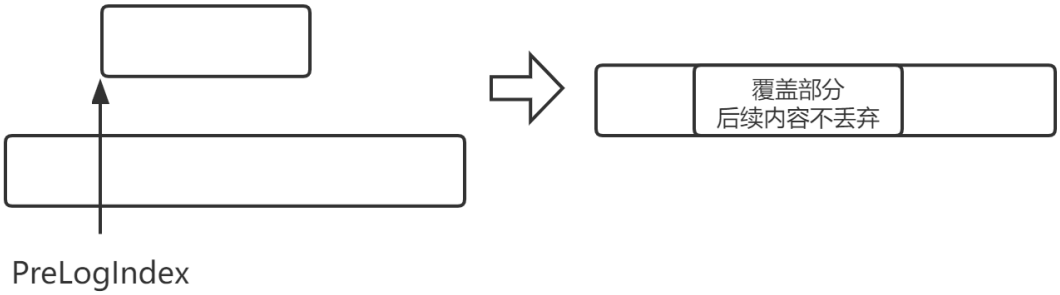
- `from`、`to` 分别表示 发送消息的节点ID 以及 接收消息的节点ID
- `Term` 和 `Commit` 也许可以不需要，我当初设计的时候是将 MsgAppend 也当做一种心跳进行操作，也需要同步这些状态，不实现这个状态的同步功能也许也是可行的。
- `logTerm` 和 `index` 此处表示的是之前已经同步的日志（索引编号为 `Next - 1` 的日志）最后的任期号和索引号，该部分的作用是在同步时进行日志的匹配（本系统中**索引号**以及**任期号**都相同的日志被视为相同的日志），如果找到了就将日志附加到对方的日志序列中（追加或覆盖）；如果无法找到，说明 leader 维护的变量 Next 有误，通过 Next 自减或者直接一次性越过该冲突任期的所有日志条目的方式重新发送 MsgAppend 同步日志。

成员	类型	作用
MsgType	MsgAppend	用于leader向集群中其他节点同步数据的消息
From	uint64	本节点ID
To	uint64	消息接收者的节点ID
Term	uint64	任期号
Commit	uint64	已提交日志编号
LogTerm	uint64	已同步的最新日志所处的任期号
Index	uint64	已同步的最新日志的索引号
Entries	Entry	日志条目数组

接收

当接收到 MsgAppend 类型的信息后，需要进行如下处理：

- 检查消息中的 `Term` 是否大于本地的 `Term`，如果大于本地则需要对任期进行修正，如果节点状态是 Candidate 或者 Leader 则表示有新的 leader 产生了，自身需要退回到 Follower 的状态。
- 检查消息中的 `Term` 是否小于本地的 `Term`，如果小于说明出现了异常，需要返回拒绝增加日志的回复。
- 检查 `logTerm` 与 `index` 所指向的日志是否存在，如果不存在表示 `leader` 维护的 `Next` 异常，返回对应的拒绝回复。反之，如果指向的日志存在，怎需要将日志附加到本地的日志列表。
- 这里有一个需要注意的点，这个点单纯是为了通过 2A 的测试，但我个人觉得没有必要。**当附加日志最后的 index 小于本地日志序列最后的 index，也就是说附加日志无法完全覆盖原有日志的时候，不能将原有日志全部丢弃，而是采取部分覆盖的方式进行处理。**



- 最后根据需求返回 MsgAppendResponse 类型的消息即可，该消息的处理见 MsgAppendResponse。

MsgAppendResponse

日志附加的返回消息，需要根据该消息调整节点维护的 `Prs` 变量。此外，该消息的处理是 leader 独有的，其他类型的节点收到此类消息可直接丢弃。

根据 `Reject` 的值可以将回复消息分成两大类，拒绝附加日志和接收附加日志两种。

当 `Reject` 为 True 时表示当前附加日志的请求被拒绝了，说明当前 leader 节点维护的 `Prs` 中的 `Next` 存在问题，需要进行修改，通常的方式直接自减然后重新发送 MsgAppend 消息进行日志的同步。但是，由于本次集群测试存在网络先隔离再合并的情况（2B），也就是说本次存在**较严重的日志落后问题**，如果使用自减的方式进行处理效率很低甚至会触发 Request Timeout 的情况，所以**需要使用 Index 加快这个过程，如果返回拒绝类型的 MsgAppendResponse，则需要将 Index 置为当前日志中最后一条日志的序号，这样可以让 leader 快速调整 Next 的数值。**

当 `Reject` 为 False 时，表示节点接收了本次的日志附加，这个时候 Index 中保存的是最后一条日志的序号，使用这个量维护 `Prs` 变量，并更新本地的 Committed 的量，**需要注意的 raft 不会提交一个之前任期的日志**，也就是说如果**当前计算出来的 Committed 指向的日志的任期 Term 不等于当前任期，则表示该日志同步未完成，不能更新 Committed。**

此外，还需要额外关注一个点，涉及到 3A 的实现。如果已经附加了最新的日志，当现在正在进行领导权的转移，且该消息代表的节点是准备迁移过去的新leader节点 `m.From == r.leadTransferee`，则表示可以进行领导权转移的操作了，具体操作详见 MsgTransferLeader 中的描述。

成员	类型	作用
MsgType	MsgAppendResponse	集群中其他节点针对leader的 MsgAppend 消息的应答消息
From	uint64	本节点ID
To	uint64	消息接收者的节点ID
Index	uint64	日志索引ID，用于节点向leader汇报自己的日志数据ID
Reject	bool	是否拒绝同步日志的请求

MsgRequestVote

该消息用于选举过程中请求别的节点把选票投给自己，使用到的量如下表所示，将对应的量填充并向集群广播即可。

成员	类型	作用
MsgType	MsgRequestVote	节点投票选举
From	uint64	本节点ID
To	uint64	消息接收者的节点ID
Term	uint64	任期ID
LogTerm	uint64	日志所处的任期ID
Index	uint64	日志索引ID

当节点接收到此消息后，需要对此进行处理，总体的处理原则是向比自己具有更新信息的节点投票，具体的处理流程如下：

- 判断消息的任期 Term 是否大于本地的 Term，如果小于说明不是合法的选举，直接发送拒绝投票请求。
- 注意，论文中还有一条合法性的判断，那就是检查是不是有合法的 leader 与本节点保持联系，如果有的话就拒绝选举（强制领导权转移除外）。

  该判断主要是为了防止有的**节点因为被移除出集群但本身不知道**的情况下，**频繁向节点发送无效的选举请求。**

  但是由于本系统是简化的系统，本次就不进行此判断了，此外，此判断可以在 Step 中使用 `Prs` 进行过滤，如果发送方不在 `Prs` 映射下就直接将消息丢弃也可以在某种意义上防止此类问题。
- 如果任期检查无问题，则改变自身的任期与状态。**如果消息任期大于本地**，说明对方的选举比本地更晚，如果本地是 Candidate 或者 Leader 状态，需要回到 Follower 状态，并**修改本地的任期。**
- 判断是否给对方进行投票，需要满足两个条件，首先是对方的日志数据是最新的 `m.LogTerm > lastTerm ||(m.LogTerm == lastTerm && m.Index >= lastIndex)`；其次，符合以下三种条件中的一种：
  - 当前没有给任何节点投票 `r.Vote == None`
  - 消息的任期大于本地 `m.Term > r.Term`
  - 之前已经投票给 `m.From` 代表的节点

  如果满足以上两个条件则将选票投给对方，否则拒绝向对方投票。

MsgRequestVoteResponse

如果对方拒绝了本次选举请求，`Reject` 会被置为 `True`，反之则会置为 `False`。

当收到选举投票的应答后，需要计算有多少节点投了同意选票，如果有**半数以上同意了则切换到 Leader 状态**，**半数以上节点拒绝了则切换到 Follower 状态。**

成员	类型	作用
MsgType	MsgRequestVoteResponse	投票应答消息
From	uint64	本节点ID
To	uint64	消息接收者的节点ID
Reject	bool	是否拒绝本次选举请求

MsgSnapshot

如果在日志附加过程中发现需要同步的日志已经被压缩了，就需要发送此请求进行日志压缩，获得日志的方式是调用 storage 的Snapshot 方法`r.RaftLog.storage.Snapshot()`。

然后按照下表对于请求进行填充即可。

成员	类型	作用
MsgType	MsgSnapshot	用于leader使用快照同步数据
From	uint64	本节点ID
To	uint64	消息接收者的节点ID
Term	uint64	任期ID
Snapshot	Snapshot	快照数据

当一个结点接收到此信息后，需要进行处理，使用消息和 snapshot 中的信息修改本地的信息，如任期以及日志的各种属性。然后，修改本地的日志，如果本地日志包含 snapshot 中的日志，说明这些日志需要被压缩，直接截断删除即可。

特别需要注意的是，如果日志长度为空，下次进行日志附加的时候可能会出现错误，需要向本地增加一条空日志。

如果本节点是新增加的节点，可能是未初始化的，需要使用 `ConfState` 对 `Prs` 进行初始化。**如果长期的网络隔离，也可能导致这种情况，比如在隔离期间进行了一系列的配置变更，如果本身已经被移除出集群了，也需要对本身进行处理。**

最后，将 snapshot 保存在 `pendingSnapshot` 中，用于上层调用 Ready 过程写入数据。

MsgHeartbeat & MsgHeartbeatResponse

这个消息的处理比较简单，类似于 MsgAppend 的简化版，仅需要修改本地的状态而不需要附加日志。

需要修改的量包括 Lead、Term 以及本地的 Committed量，同时修改本地的选举周期，采取自增的方式即可。

此外可能针对测试需要进行一些与论文中不太相同的调整，~~这点就靠大家自己尝试了，笑~~。

MsgTransferLeader

成员	类型	作用
MsgType	MsgTransferLeader	领导权转移
From	uint64	领导权转移对象的ID
To	uint64	Leader节点的ID

该消息某种程度是本地消息，不能向外传递，该消息仅供 Leader 进行处理，From 代表的不是发送者而是领导权转移的对象。

当接收到该消息后，首先需要给 leadTransferee 赋值，将 From 赋值给 leadTransferee，表示当前需要进行领导权的转移。

检查转移对象是否就是本身，如果是本身，则忽略此消息，将 leadTransferee 更改回 None。

检查对方的 Match Index 是否等于自身最后一条日志的编号，如果是表明对方拥有最新的日志，可以进行领导权的转移，发送 MsgTimeoutNow 类型的消息强制对方超时重新选举。

如果不相等则需要进行日志的同步后再进行领导权的转移。

MsgTimeoutNow

该消息处理极其简单，直接发起选举即可，和 MsgHup 的处理类似。

以上就是各种消息的处理流程，具体的我会写成文档供大家参考，分享结束后我会发到群里。

最后一个模块是 rawnode 模块，该模块的主要功能是对 raft 进行封装，向下传递消息与日志，向上提供 raft 信息的读取接口，并提供最重要的 Ready、Advance 以及 HasReady 三个用于消息日志应用与 raft 状态转换的接口。

顾名思义，HasReady 判断是否存在数据需要进行处理，如果存在则返回 True，后续将调用 Ready 方法取出对应的数据，将取出的数据处理完毕后，使用 Advance 方法进行收尾。知道了用途后就知道如何编程了。

我们先了解一下 Ready 包含的成员，对需要处理的数据进行一个初步的了解，如下表所示， Ready 存在5个成员。

成员名称	类型
SoftState	SoftState
HardState	HardState
Entries	[]pb.Entry
Snapshot	pb.Snapshot
CommittedEntries	[]pb.Entry
Messages	[]pb.Message

`SoftState` 保存的是系统的易变状态，这个状态是随系统变化的，不需要进行持久化，后续事实上不需要使用，~~我也不知道为啥要有这个~~，按照提示进行赋值即可。

`HardState` 保存的是系统的硬状态，也是就是需要进行持久化的状态，包括节点当前Term、Vote、Commit。当系统崩溃重启后，这些状态需要恢复到崩溃前的情况。

`Entries` 未进行持久化，需要保存到持久化存储的日志。

`CommittedEntries` 已经提交但是尚未进行应用，也就是包含读写请求以及其他一些配置请求，还没处理的日志。

`Messages` 包含需要发送的消息。

HasReady 方法判断是否这些数据有更新，需要进行处理，Ready 方法则是取出需要更新的数据。在了解了这些量的含义就很好判断是否需要处理以及如何处理了。

`Advance` 需要进行收尾，修改 Log 的 stabled 以及 applied 指向的序号，同时调用 `maybeCompact` 方法进行可能的日志截断。

以下是我之前写的部分文档内容，仅供参考：

实现 leader transfer

为了实现领导者转移，引入两种新消息类型: `MsgTransferLeader` 以及 `MsgTimeoutNow`.

当需要进行领导权的转移时，首先将类型为 `MsgTransferLeader` 的信息传递给 leader 的 `raft.Raft.Step` 方法进行处理，该方法被 `raft/rawnode.go` 中的 `TransferLeader` 方法显式调用。

```
func (rn *RawNode) TransferLeader(transferee uint64) {
    _ = rn.Raft.Step(pb.Message{
        MsgType: pb.MessageType_MsgTransferLeader, From: transferee})
}
```

为确保传输的成功，当前的 leader 应该确认转移目标的资质，如是否保存了最新的日志信息（**需要检查转移对象是否存在**）等。如果转移目标不符合转移条件，就需要帮助转移目标，使得其最终符合转移资质。如果转移目标的日志不是最新的，那么将**发送** `MsgAppend` **message 更新日志信息，并停止接收新的日志。**

停止接收日志，使用 `leadTransferee` 进行控制，当每次结点成为 leader 时，将其置为0表示没有转移对象；当收到对应的转移信息后，将其置为转移对象的id，然后用此量控制是否接收 `MessageType_MsgPropose` 进行日志的处理。

在接收到 appendresponse 后判断是否可以进行转移，如不行则继续进行日志的更新；为防止某些命名在网络中丢失导致转移失败，通过心跳的方式定期检查是否可以转移，直到转移成功。

当转移对象符合转移条件后，leader 应该发送 `MsgTimeoutNow` message 到转移目标，当转移目标接收到此信号后，应该立即开始进行新的选举。

实现 conf change 配置变迁

本次实现的 Conf change 算法只能一个一个地添加或删除peer，需要注意的是conf change 起始于调用 leader 的 `raft.RawNode.ProposeConfChange` 方法，此方法会提交类型 `pb.Entry.EntryType` 为 `EntryConfChange` 的日志，日志的 `pb.Entry.Data` 存放 `pb.ConfChange` 的信息。

当类型为 `EntryConfChange` 的日志被提交后，需要通过 `RawNode.ApplyConfChange` 方法进行应用，该过程在 `peer_msg_handler.go` 中 `HandleRaftReady` 方法中，对于 `ready.CommittedEntries` 的 process 中。

这样之后，你就可以根据 `pb.ConfChange` 信息，通过 `raft.Raft.addNode` 以及 `raft.Raft.removeNode` 方法修改结点。

> Hints:
>
> - 可以将 `MsgTransferLeader` message 的 `Message.from` 设置为转移目标id
> - 将 `MsgHup` message 传递给 `Raft.Step` 方法可以开始选举，也可以直接调用选举方法进行选举
> - 调用 `pb.ConfChange.Marshal` 将 `pb.ConfChange` 转换成字节流，并将之存放进 `pb.Entry.Data` 中进行传递。
> - 注意修改 `PendingConfIndex`，在上一次 conf change 为完全生效时，下一次的 conf change 不得起效。该值将被设置为上一次 conf change 日志的序号，每次进行 conf change 日志 propose 时，只有当 applied index 小于等于 该日志序号才行，**否则先拒绝处理该请求，将msg重新放入msgs数组中。**
> - 移除节点后，**某些日志可能可以提交了，因此需要在移除节点方法中增加检验 commited 的部分**

测试问题与解决方案：

1. 测试函数 `TestLeaderTransferToUpToDateNodeFromFollower3A` 非常奇怪，他**将 `MsgTransferLeader` 信息发送给了非leader**，这种行为非常迷惑，为了通过测试，必须做出一些奇怪操作，如将此信息转发给leader结点，但是这样就**违背了 `MsgTransferLeader` message 是本地信息不源自网络的要求**，总之非常迷惑。

   ```
   nt.send(pb.Message{From: 2, To: 2, MsgType:
   pb.MessageType_MsgTransferLeader})
   ```

2. 测试函数 `TestTransferNonMember3A` 其作用是测试向已经从集群中移除的结点发送信息后，应该什么都不发生，但是这个测试函数比较诡异。`r.Prs` 中仅删除了 1，我认为既然结点从集群中删除，对于删除的结点来说，`r.Prs` 应该直接置为空即可。

```
r := newTestRaft(1, []uint64{2, 3, 4}, 5, 1, NewMemoryStorage())
r.Step(pb.Message{From: 2, To: 1, MsgType: pb.MessageType_MsgTimeoutNow})
r.Step(pb.Message{From: 2, To: 1, MsgType:
pb.MessageType_MsgRequestVoteResponse})
r.Step(pb.Message{From: 3, To: 1, MsgType:
pb.MessageType_MsgRequestVoteResponse})
```