Raft

写在前面

这是一系列实验中的第一个实验，在这些实验中，你将构建一个容错的键/值存储系统。在这个实验中，你将实现 Raft，这是一种复制状态机协议。在下一个实验中，你将在 Raft 之上构建一个键/值服务。然后，你将通过多个复制状态机对服务进行“分片”以提高性能。

一个复制的服务通过将其状态（即数据）的完整副本存储在多个副本服务器上实现容错。复制允许服务即使有部分服务器发生故障（崩溃或网络中断）也能继续运行。挑战在于故障可能导致副本持有不同的数据副本。

Raft 将客户端请求组织成一个序列，称为日志，并确保所有副本服务器看到相同的日志。每个副本按照日志顺序执行客户端请求，并将其应用于服务状态的本地副本。由于所有活动的副本看到相同的日志内容，它们都会以相同的顺序执行相同的请求，从而保持相同的服务状态。如果一个服务器失败但后来恢复，Raft 会负责将其日志更新到最新状态。只要至少有一半以上的服务器存活并能够互相通信，Raft 将继续运行。如果没有这样的多数，Raft 将不会有任何进展，但一旦多数服务器能够再次通信，它将从上次中断的地方继续运行。

在本实验中，你需要将 Raft 实现为一个 Go 对象类型，并带有相关方法，旨在作为更大服务中的一个模块使用。一组 Raft 实例通过 RPC 相互通信以维护复制的日志。你的 Raft 接口将支持一系列编号的命令，也称为日志条目。这些条目带有索引编号。带有特定索引的日志条目最终会被提交。在这一点上，你的 Raft 应该将日志条目发送给更大的服务以执行。

您应该遵循扩展版 Raft 论文中的设计，特别注意图 2。您将实现论文中的大部分内容，包括保存持久状态并在节点失败后重启时读取它。您将不会实现集群成员变更（第 6 节）。

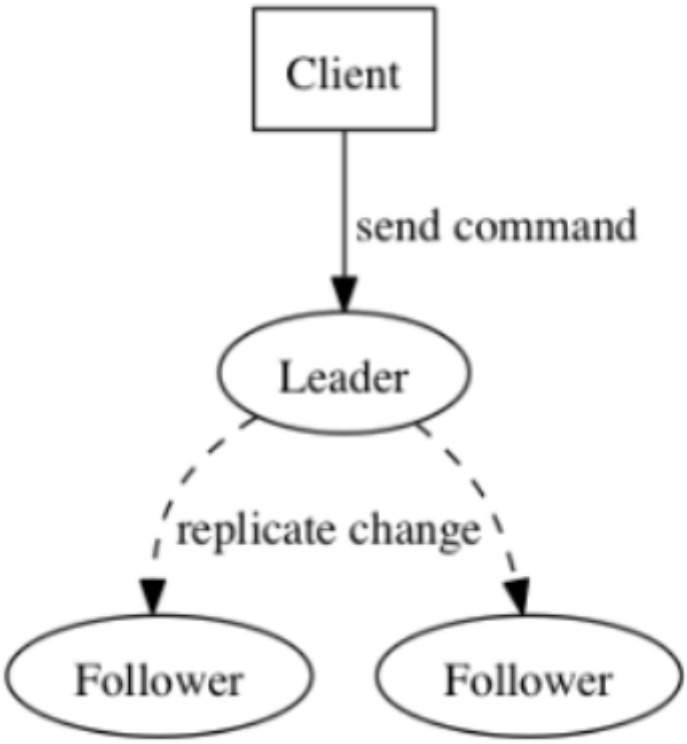
一、Raft算法概述

Raft算法是从多副本状态机的角度提出，用于管理多副本状态机的日志复制。Raft实现了和Paxos相同的功能，它将一致性分解为多个子问题：Leader选举（Leader election）、日志同步（Log replication）、安全性（Safety）、日志压缩（Log compaction）、成员变更（Membership change）等。同时，Raft算法使用了更强的假设来减少了需要考虑的状态，使之变的易于理解和实现。

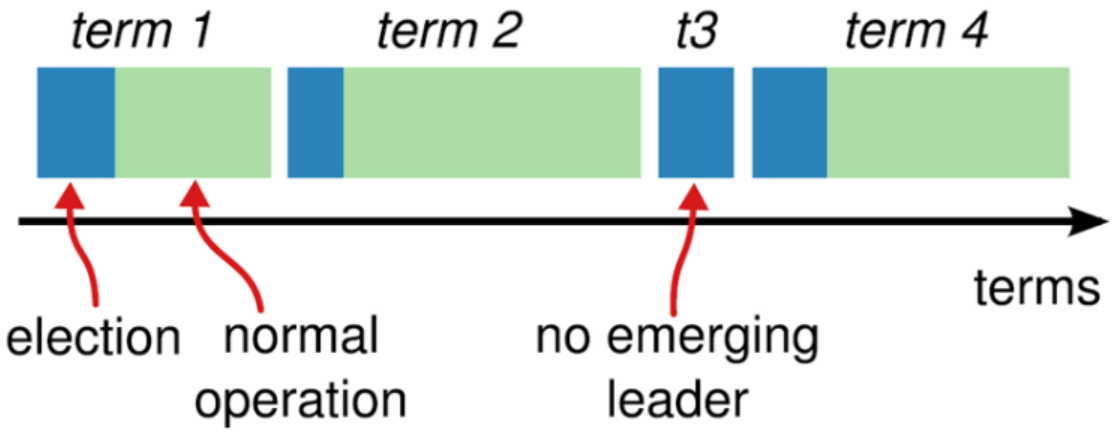
- **广播时间(broadcastTime) << 选举超时时间(electionTimeout) << 平均故障时间(MTBF)**

Raft将系统中的角色分为领导者（Leader）、跟从者（Follower）和候选人（Candidate）：

- **Leader**：接受客户端请求，并向Follower同步请求日志，当日志同步到大多数节点上后告诉Follower提交日志。
- **Follower**：接受并持久化Leader同步的日志，在Leader告之日志可以提交之后，提交日志。
- **Candidate**：Leader选举过程中的临时角色。



Follower只响应其他服务器的请求。如果Follower超时没有收到Leader的消息，它会成为一个Candidate并且开始一次Leader选举。收到大多数服务器投票的Candidate会成为新的Leader。Leader在宕机之前会一直保持Leader的状态。



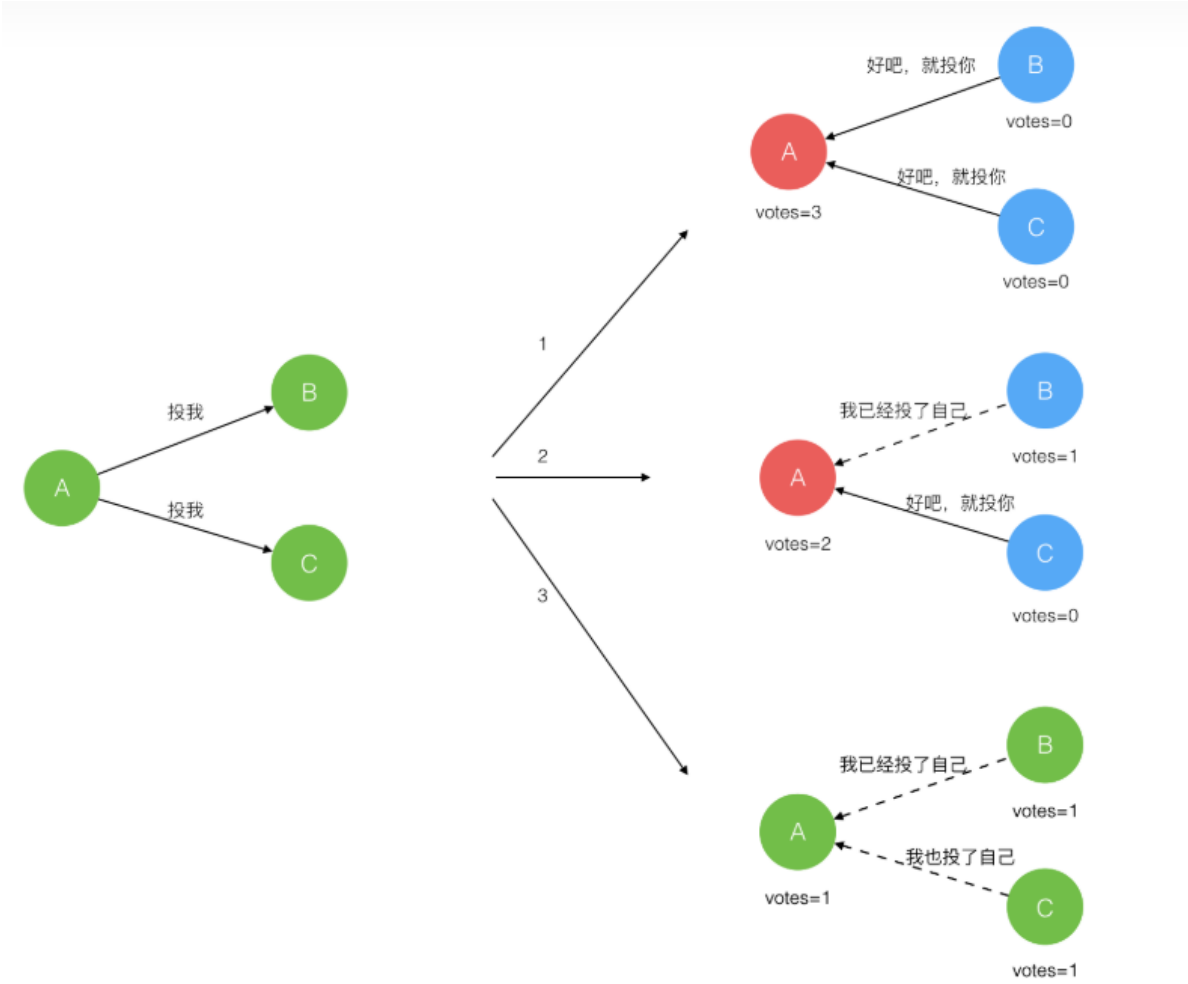
Raft算法将时间分为一个个的任期（term），每一个term的开始都是Leader选举。在成功选举Leader之后，Leader会在整个term内管理整个集群。如果Leader选举失败，该term就会因为没有Leader而结束。

二、Leader选举

Raft 使用心跳（heartbeat）触发Leader选举。当服务器启动时，初始化为Follower。Leader向所有Followers周期性发送heartbeat。如果Follower在选举超时时间内没有收到Leader的heartbeat，就会等待一段随机的时间后发起一次Leader选举。

Follower将其当前term加一然后转换为Candidate。它首先给自己投票并且给集群中的其他服务器发送RequestVote RPC （RPC细节参见八、Raft算法总结）。结果有以下三种情况：

- 赢得了多数的选票，成功选举为Leader；
- 收到了Leader的消息，表示有其它服务器已经抢先当选了Leader；
- 没有服务器赢得多数的选票，Leader选举失败，等待选举时间超时后发起下一次选举。

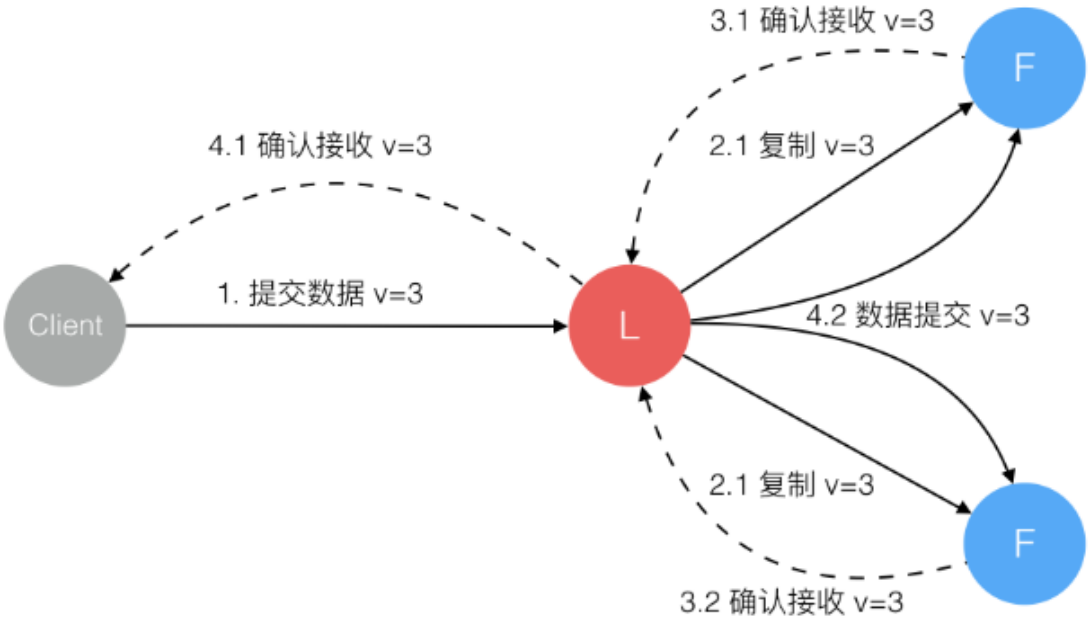


选举出Leader后，Leader通过定期向所有Followers发送心跳信息维持其统治。若Follower一段时间未收到Leader的心跳则认为Leader可能已经挂了，再次发起Leader选举过程。

Raft保证选举出的Leader上一定具有最新的已提交的日志，这一点将在四、安全性中说明。

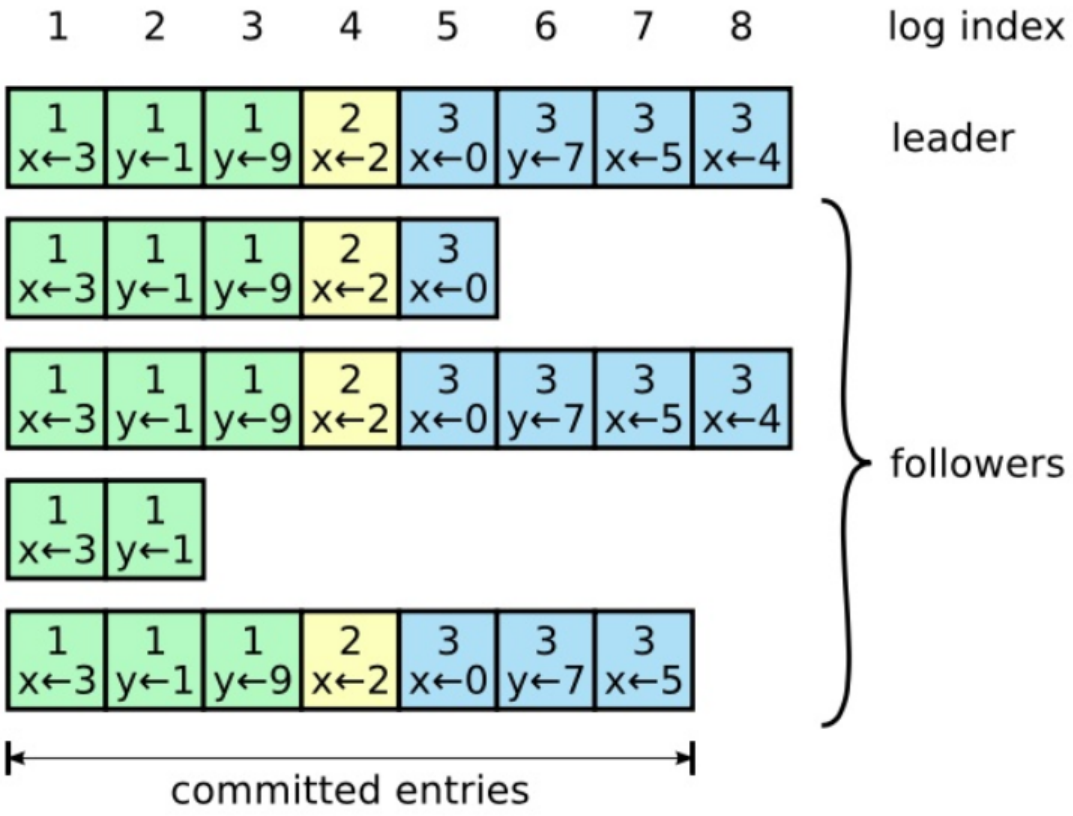
三、日志同步

Leader选出后，就开始接收客户端的请求。Leader把请求作为日志条目（Log entries）加入到它的日志中，然后并行的向其他服务器发起 AppendEntries RPC （RPC细节参见八、Raft算法总结）复制日志条目。当这条日志被复制到大多数服务器上，Leader将这条日志应用到它的状态机并向客户端返回执行结果。



某些Followers可能没有成功的复制日志，Leader会无限的重试 AppendEntries RPC直到所有的Followers最终存储了所有的日志条目。

日志由有序编号（log index）的日志条目组成。每个日志条目包含它被创建时的任期号（term），和用于状态机执行的命令。如果一个日志条目被复制到大多数服务器上，就被认为可以提交（commit）了。



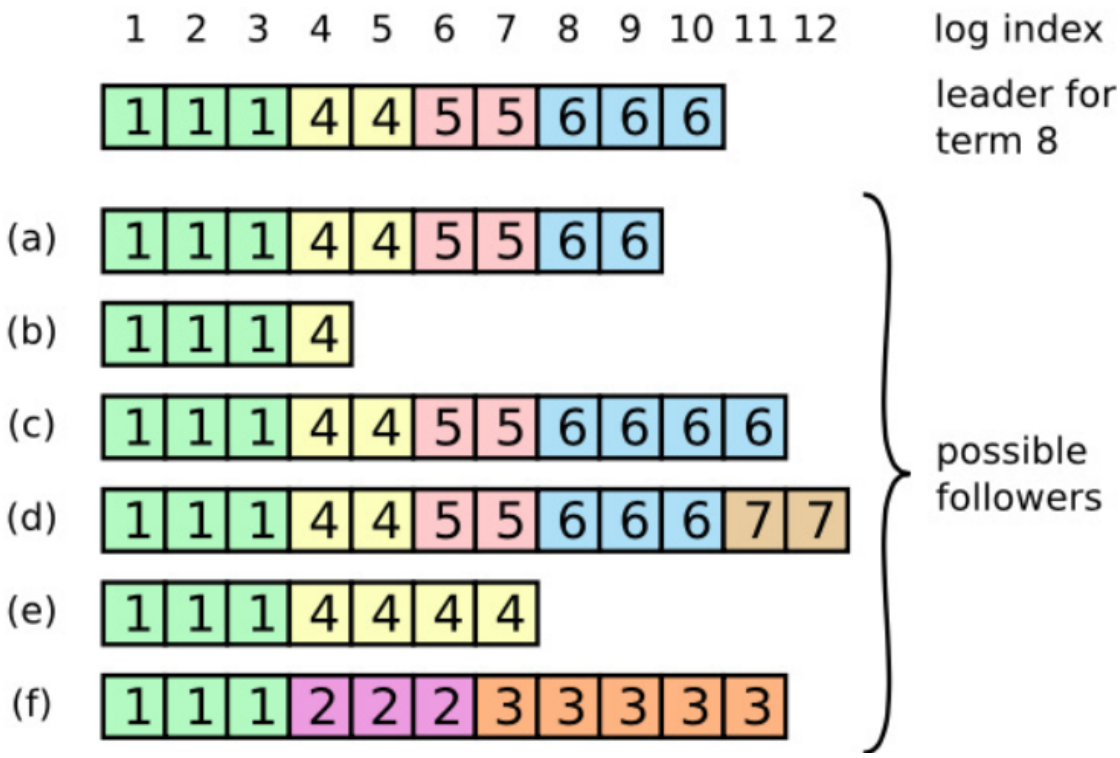
Raft日志同步保证如下两点：

- 如果不同日志中的两个条目有着相同的索引和任期号，则它们所存储的命令是相同的。
- 如果不同日志中的两个条目有着相同的索引和任期号，则它们之前的所有条目都是完全一样的。

第一条特性源于Leader在一个term内在给定的一个log index最多创建一条日志条目，同时该条目在日志中的位置也从来不会改变。

第二条特性源于 AppendEntries 的一个简单的一致性检查。当发送一个 AppendEntries RPC 时，Leader会把新日志条目紧接着之前的条目的log index和term都包含在里面。如果Follower没有在它的日志中找到log index和term都相同的日志，它就会拒绝新的日志条目。

一般情况下，Leader和Followers的日志保持一致，因此 AppendEntries 一致性检查通常不会失败。然而，Leader崩溃可能会导致日志不一致：旧的Leader可能没有完全复制完日志中的所有条目。



上图阐述了一些Followers可能和新的Leader日志不同的情况。一个Follower可能会丢失掉Leader上的一些条目，也有可能包含一些Leader没有的条目，也有可能两者都会发生。丢失的或者多出来的条目可能会持续多个任期。

Leader通过强制Followers复制它的日志来处理日志的不一致，Followers上的不一致的日志会被Leader的日志覆盖。

Leader为了使Followers的日志同自己的一致，Leader需要找到Followers同它的日志一致的地方，然后覆盖Followers在该位置之后的条目。

Leader会从后往前试，每次AppendEntries失败后尝试前一个日志条目，直到成功找到每个Follower的日志一致位点，然后向后逐条覆盖Followers在该位置之后的条目。

四、 安全性

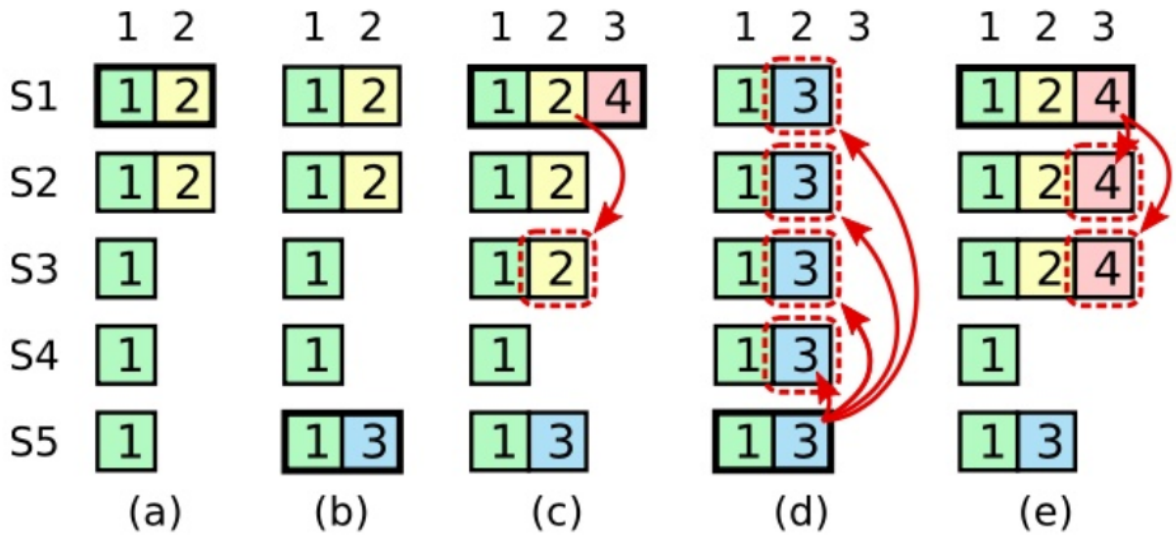
Raft增加了如下两条限制以保证安全性：

- 拥有最新的已提交的log entry的Follower才有资格成为Leader。

这个保证是在RequestVote RPC中做的，Candidate在发送RequestVote RPC时，要带上自己的最后一条日志的term和log index，其他节点收到消息时，如果发现自己的日志比请求中携带的更新，则拒绝投票。日志比较的原则是，如果本地的最后一条log entry的term更大，则term大的更新，如果term一样大，则log index更大的更新。

- Leader只能推进commit index来提交当前term的已经复制到大多数服务器上的日志，旧term日志的提交要等到提交当前term的日志来间接提交（log index 小于 commit index的日志被间接提交）。

之所以要这样，是因为可能会出现已提交的日志又被覆盖的情况：



已提交的日志被覆盖

阶段 A：初始状态

1. **Leader 为 S1**，term 为 2。
2. S1 在其日志中写入 `(2, 2)`，这是该 Leader 在 term 2 中写入的日志。
3. 该日志成功同步到了 Follower（S2）。这意味着 `(2, 2)` 已经被 S2 接收并保存。

阶段 B：S1 下线，新的选举

1. S1 离线后，发生了一次新的选举，选举出 **S5** 作为新的 Leader。
2. 此时系统的 **term** 被更新为 3，并且 S5 写入了 `(3, 2)`，这是新 Leader（S5）在 term 3 中写入的日志。请注意，虽然日志索引为 2，但这条日志是在新的 term 下写入的，所以它被视为一个新的日志。
3. 由于 S5 还没有将新日志同步到 Followers，它又发生了下线，这导致了另一次选举。

阶段 C：S1 重新上线并成为新的 Leader

1. S1 重新上线并成功当选为 Leader，系统的 **term** 更新为 4。
2. S1 会将日志同步给 Followers，并将 `(2, 2)` 同步到了 S3，因为该日志已经在 S1 和 S2 中存在，所以 S3 会接收该日志。
3. 由于日志 `(2, 2)` 已经被大多数节点（S1, S2, S3）所保存，Raft 协议允许这个日志被提交。注意，日志 `(2, 2)` 在 term 2 中写入，但是随着 S1 成为新的 Leader（term 4），这个日志的提交条件得到满足，因而 `(2, 2)` 被提交。

阶段 D：S1 离线，再次选举

1. 当 S1 下线时，系统再次触发选举。此时 S5 可以被选举为新的 Leader，**term** 更新为 5。
2. 在选举时，S5 具有比 S1 更高的 term（term = 5），并且它的日志也比 S1 的日志更新：日志 `(3, 2)` 比 S1 的日志 `(2, 2)` 更新。
3. 由于 Raft 协议要求 Leader 必须包含大多数节点的日志，S5 会截断其不一致的日志并同步更新。
    - 这意味着 S2 和 S3 中已提交的日志 `(2, 2)` 被 **截断**。
    - 即便 S2 和 S3 已经确认了 `(2, 2)`，由于 S5 需要保证其日志一致性，并且它拥有更高 term 的日志（即 `(3, 2)`），它会丢弃这些旧的日志（来自 term 2 的 `(2, 2)`）并替换为自己的日志。

最终结论

1. 即便 `(2, 2)` 已经在 S1, S2, 和 S3 中被提交，但由于 Raft 协议要求 Leader 的日志必须是更新的，且新的 Leader（S5）会截断不一致的日志，因此 **日志 `(2, 2)` 无法最终提交。**
2. 只有在当前 Leader（term = 4）生成的日志（如 `(4, 4)`）被大多数节点确认并提交后，才能保证最终的日志一致性。
3. 如果 S1 生成了 `(4, 4)`，并且这个日志被大多数节点确认，那么在此基础上的所有日志（包括之前的日志）都可以被提交。即便 S1 之后下线，新的 Leader 也必须包含该日志（`(4, 4)`），否则它无法成为新的 Leader。

五、日志压缩

在实际的系统中，不能让日志无限增长，否则系统重启时需要花很长的时间进行回放，从而影响可用性。Raft采用对整个系统进行snapshot来解决，snapshot之前的日志都可以丢弃。

每个副本独立的对自己的系统状态进行snapshot，并且只能对已经提交的日志记录进行snapshot。

Snapshot中包含以下内容：

- 日志元数据。最后一条已提交的 log entry的 log index和term。这两个值在snapshot之后的第一条log entry的AppendEntries RPC的完整性检查的时候会被用上。
- 系统当前状态。

当Leader要发给某个日志落后太多的Follower的log entry被丢弃，Leader会将snapshot发给Follower。或者当新加进一台机器时，也会发送snapshot给它。发送snapshot使用InstalledSnapshot RPC（RPC细节参见八、Raft算法总结）。

做snapshot既不要做的太频繁，否则消耗磁盘带宽， 也不要做的太不频繁，否则一旦节点重启需要回放大量日志，影响可用性。推荐当日志达到某个固定的大小做一次snapshot。

做一次snapshot可能耗时过长，会影响正常日志同步。可以通过使用copy-on-write技术避免snapshot过程影响正常日志同步

六、成员变更

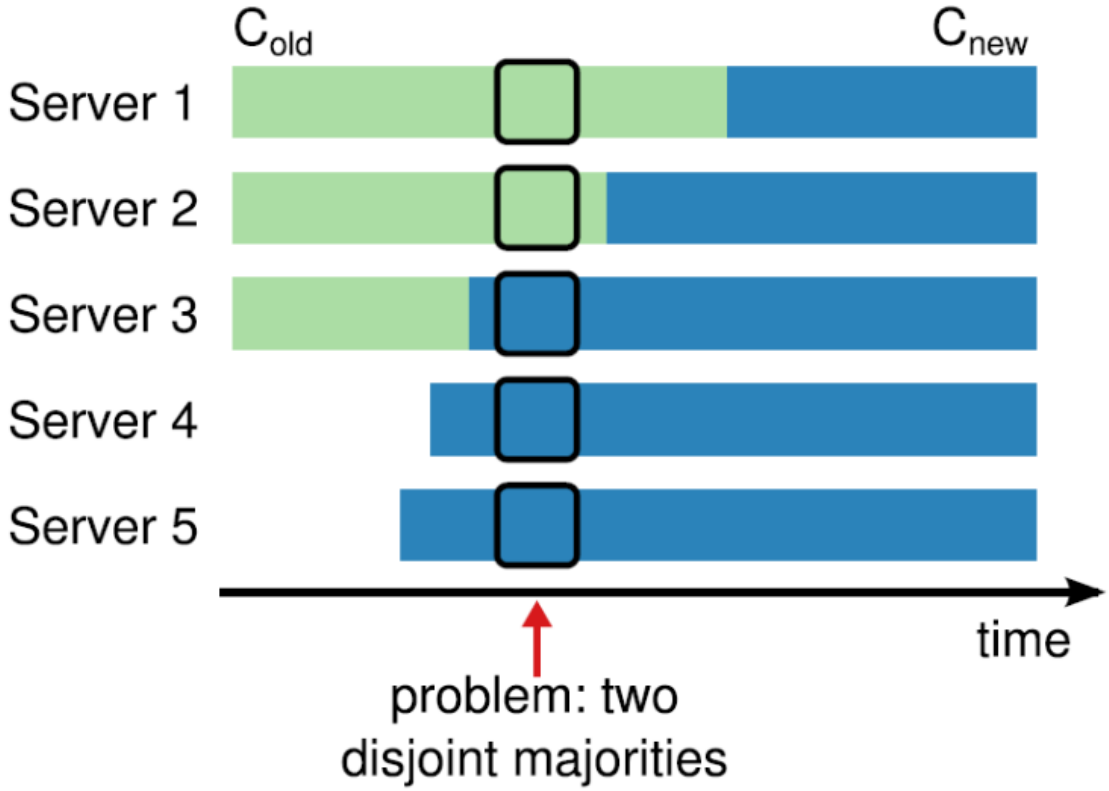
成员变更是在集群运行过程中副本发生变化，如增加/减少副本数、节点替换等。

成员变更也是一个分布式一致性问题，既所有服务器对新成员达成一致。但是成员变更又有其特殊性，因为在成员变更的一致性达成的过程中，参与投票的进程会发生变化。

如果将成员变更当成一般的一致性问题，直接向Leader发送成员变更请求，Leader复制成员变更日志，达成多数派之后提交，各服务器提交成员变更日志后从旧成员配置（Cold）切换到新成员配置（Cnew）。

因为各个服务器提交成员变更日志的时刻可能不同，造成各个服务器从旧成员配置（Cold）切换到新成员配置（Cnew）的时刻不同。

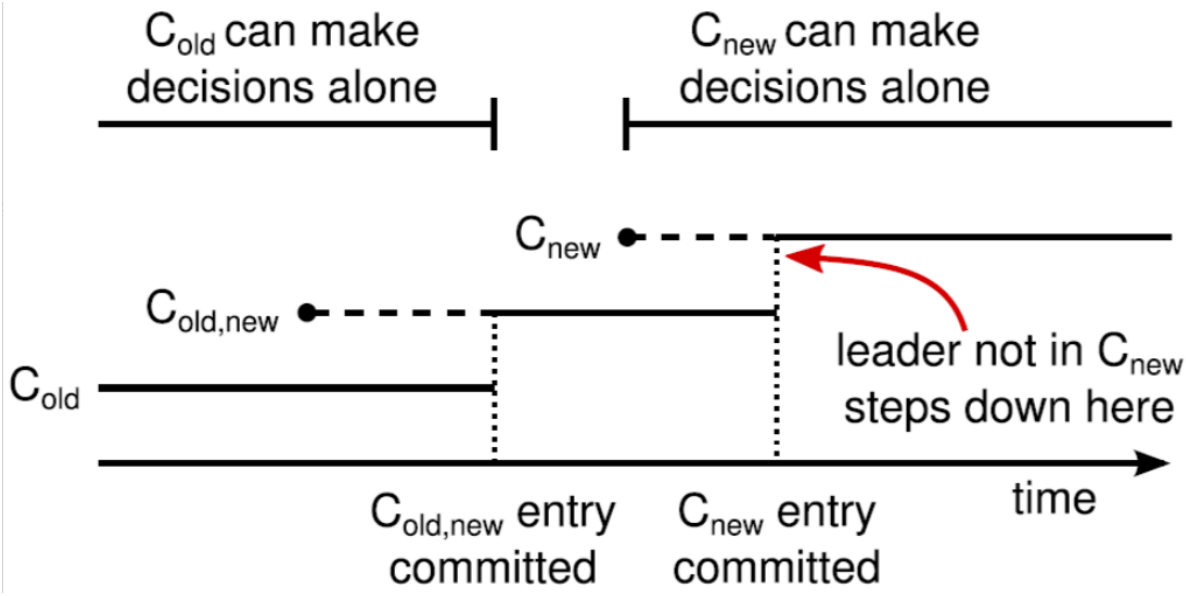
成员变更不能影响服务的可用性，但是成员变更过程的某一时刻，可能出现在Cold和Cnew中同时存在两个不相交的多数派，进而可能选出两个Leader，形成不同的决议，破坏安全性。



成员变更的某一时刻Cold和Cnew中同时存在两个不相交的多数派

由于成员变更的这一特殊性，成员变更不能当成一般的一致性问题去解决。

为了解决这一问题，Raft提出了两阶段的成员变更方法。集群先从旧成员配置Cold切换到一个过渡成员配置，称为共同一致（joint consensus），共同一致是旧成员配置Cold和新成员配置Cnew的组合Cold U Cnew，一旦共同一致Cold U Cnew被提交，系统再切换到新成员配置Cnew。



Raft两阶段成员变更过程如下：

1. Leader收到成员变更请求从Cold切成Cold,new；
2. Leader在本地生成一个新的log entry，其内容是Cold∪Cnew，代表当前时刻新旧成员配置共存，写入本地日志，同时将该log entry复制至Cold∪Cnew中的所有副本。在此之后新的日志同步需要保证得到Cold和Cnew两个多数派的确认；
3. Follower收到Cold∪Cnew的log entry后更新本地日志，并且此时就以该配置作为自己的成员配置；

4. 如果Cold和Cnew中的两个多数派确认了Cold U Cnew这条日志，Leader就提交这条log entry并切换到Cnew；
5. 接下来Leader生成一条新的log entry，其内容是新成员配置Cnew，同样将该log entry写入本地日志，同时复制到Follower上；
6. Follower收到新成员配置Cnew后，将其写入日志，并且从此刻起，就以该配置作为自己的成员配置，并且如果发现自己不在Cnew这个成员配置中会自动退出；
7. Leader收到Cnew的多数派确认后，表示成员变更成功，后续的日志只要得到Cnew多数派确认即可。Leader给客户端回复成员变更执行成功。

异常分析：

- 如果Leader的Cold U Cnew尚未推送到Follower，Leader就挂了，此后选出的新Leader并不包含这条日志，此时新Leader依然使用Cold作为自己的成员配置。
- 如果Leader的Cold U Cnew推送到大部分的Follower后就挂了，此后选出的新Leader可能是Cold也可能是Cnew中的某个Follower。
- 如果Leader在推送Cnew配置的过程中挂了，那么同样，新选出来的Leader可能是Cold也可能是Cnew中的某一个，此后客户端继续执行一次改变配置的命令即可。
- 如果大多数的Follower确认了Cnew这个消息后，那么接下来即使Leader挂了，新选出来的Leader肯定位于Cnew中。

两阶段成员变更比较通用且容易理解，但是实现比较复杂，同时两阶段的变更协议也会在一定程度上影响变更过程中的服务可用性，因此我们期望增强成员变更的限制，以简化操作流程。

两阶段成员变更，之所以分为两个阶段，是因为对Cold与Cnew的关系没有做任何假设，为了避免Cold和Cnew各自形成不相交的多数派选出两个Leader，才引入了两阶段方案。

如果增强成员变更的限制，假设Cold与Cnew任意的多数派交集不为空，这两个成员配置就无法各自形成多数派，那么成员变更方案就可能简化为一阶段。

那么如何限制Cold与Cnew，使之任意的多数派交集不为空呢？方法就是每次成员变更只允许增加或删除一个成员。

可从数学上严格证明，只要每次只允许增加或删除一个成员，Cold与Cnew不可能形成两个不相交的多数派。

一阶段成员变更：

- 成员变更限制每次只能增加或删除一个成员（如果要变更多个成员，连续变更多次）。
- 成员变更由Leader发起，Cnew得到多数派确认后，返回客户端成员变更成功。
- 一次成员变更成功前不允许开始下一次成员变更，因此新任Leader在开始提供服务前要将自己本地保存的最新成员配置重新投票形成多数派确认。
- Leader只要开始同步新成员配置，即可开始使用新的成员配置进行日志同步。

3A 领导人选举

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

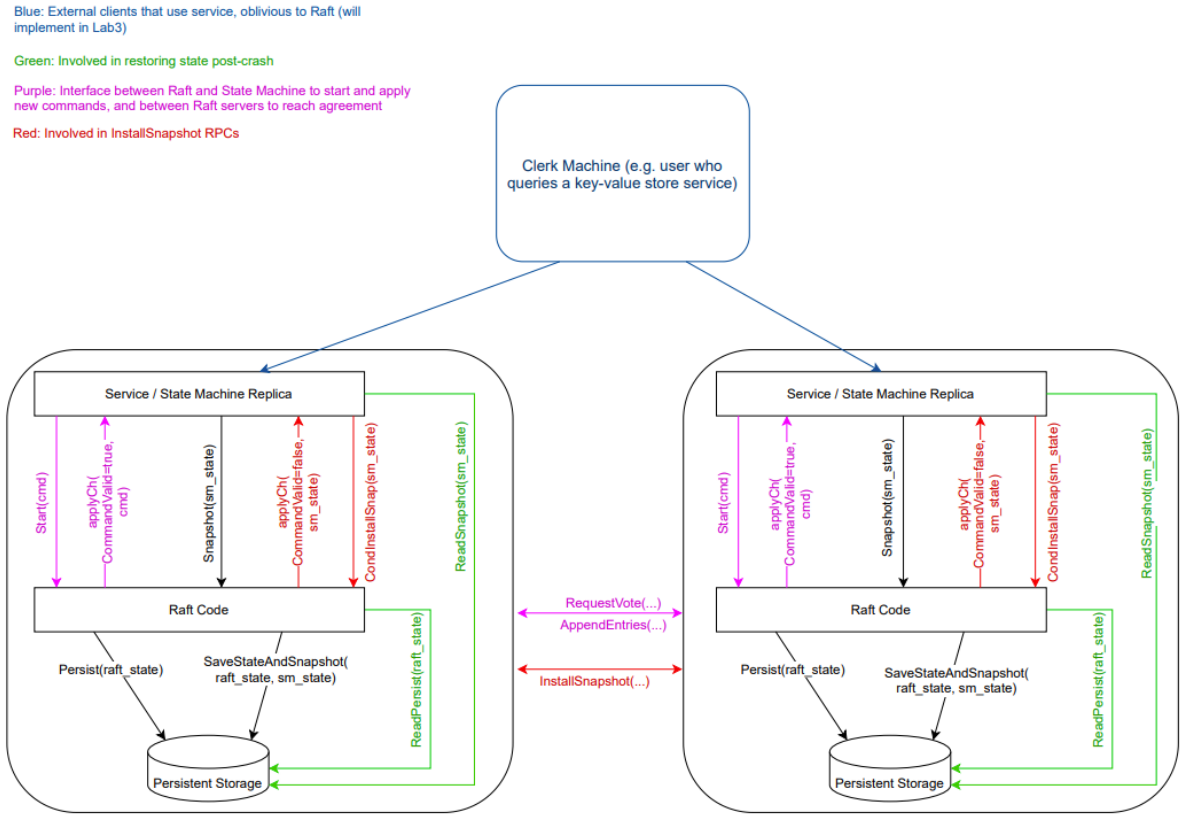
Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Raft交互图



代码部分

Raft 结构体

```
// A Go object implementing a single Raft peer.
type Raft struct {
    mu        sync.RWMutex        // Lock to protect shared access to this
peer's state
    peers     []*labrpc.ClientEnd // RPC end points of all peers
    persister *Persister          // Object to hold this peer's persisted state
    me        int                 // this peer's index into peers[]
    dead      int32               // set by Kill()

    // Your data here (3A, 3B, 3C).
    // Look at the paper's Figure 2 for a description of what
    // state a Raft server must maintain.
    currentTerm int        //当前任期号
    votedFor    int        //candidateId在当前任期内收到的选票（如果没有，则为null）
    logs        []LogEntry //日志条目;每个条目包含状态机的命令，以及leader接收条目的时间
（第一个索引为1）。

    // Volatile state on all servers
    commitIndex int //已知提交的最高日志项的索引（初始化为0，单调递增）
    lastApplied int //应用于状态机的最高日志项的索引（初始化为0，单调递增）

    // Volatile state on leaders(选举后重新初始化)
    nextIndex  []int //对于每个服务器，发送到该服务器的下一个日志条目的索引（初始化为leader
最后一个日志索引+ 1）
    matchIndex []int //对于每个服务器，已知在服务器上复制的最高日志条目的索引（初始化为0，单
调递增）

    //
```

```
    state          NodeState     // current state of the server
    electionTimer  *time.Timer   //选举超时计时器
    heartbeatTimer *time.Timer   //心跳超时计时器
    applych        chan ApplyMsg //向服务发送应用消息的通道
    applyCond      *sync.Cond    //应用程序的条件变量
    replicatorCond []*sync.Cond  //复制程序程序的条件变量
}
```

测试函数回调用GetState来判断每个RaftServer的状态

```
/ return currentTerm and whether this server
// believes it is the leader.
func (rf *Raft) GetState() (int, bool) {
    rf.mu.RLock()
    defer rf.mu.RUnlock()

    return rf.currentTerm, rf.state == Leader
}
```

raftServer的三种状态

```
//6.5840/src/raft/util.go
type NodeState uint8

const (
    Follower NodeState = iota
    Candidate
    Leader
)

type LogEntry struct { //日志结构体
    Command interface{}
    Term    int
    Index   int
}
```

Make函数

```
// the service or tester wants to create a Raft server. the ports
// of all the Raft servers (including this one) are in peers[]. this
// server's port is peers[me]. all the servers' peers[] arrays
// have the same order. persister is a place for this server to
// save its persistent state, and also initially holds the most
// recent saved state, if any. applyCh is a channel on which the
// tester or service expects Raft to send ApplyMsg messages.
// Make() must return quickly, so it should start goroutines
// for any long-running work.
func Make(peers []*labrpc.ClientEnd, me int,
    persister *Persister, applyCh chan ApplyMsg) *Raft {
    rf := &Raft{
        mu:        sync.RWMutex{},
        peers:     peers,
        persister: persister,
        me:        me,
        dead:      0,
```

```
        currentTerm: 0,
        votedFor:    -1,
        logs:        make([]LogEntry, 1), //索引0处的虚拟条目

        commitIndex: 0,
        lastApplied: 0,

        nextIndex:  make([]int, len(peers)),
        matchIndex: make([]int, len(peers)),

        state: Follower,
        //心跳超时时间<<小于选举超时(follower需要成为candidate所需要等待的时间)<<平均故障
时间
        electionTimer:  time.NewTimer(RandomElectionTimeout()),
        heartbeatTimer: time.NewTimer(StableHeartbeatTimeout()),

        applych: applyCh,
        // replicatorCond: make([]*sync.Cond, len(peers)),
    }

    SugarLogger.Infof("%v raft Server Initialization", me)
    // Your initialization code here (3A, 3B, 3C).
    // rf.applyCond = sync.NewCond(&rf.mu)

    // initialize from state persisted before a crash
    rf.readPersist(persister.ReadRaftState())

    // initialize nextIndex and matchIndex, and start replicator goroutine
    // for peer := range peers {
    //  rf.matchIndex[peer], rf.nextIndex[peer] = 0, rf.getLastLog().Index+1
    //  if peer != rf.me {
    //      rf.replicatorCond[peer] = sync.NewCond(&sync.Mutex{})

    //      go rf.replicator(peer)
    //  }
    // }
    // start ticker goroutine to start elections
    go rf.ticker()

    // start apply goroutine to apply log entries to state machine
    // go rf.applier()

    return rf
}
```

在这个函数里面初始化log

```
//6.5840/src/raft/test_test.go
func TestInitialElection3A(t *testing.T) {
    servers := 3
    InitLogger()
    cfg := make_config(t, servers, false, false)
    defer cfg.cleanup()

    // 初始化log
```

```
    cfg.begin("Test (3A): initial election")

    // is a leader elected?
    cfg.checkOneLeader()

    // sleep a bit to avoid racing with followers learning of the
    // election, then check that all peers agree on the term.
    time.Sleep(50 * time.Millisecond)
    term1 := cfg.checkTerms()
    if term1 < 1 {
        t.Fatalf("term is %v, but should be at least 1", term1)
    }

    // does the leader+term stay the same if there is no network failure?
    time.Sleep(2 * RaftElectionTimeout)
    term2 := cfg.checkTerms()
    if term1 != term2 {
        fmt.Printf("warning: term changed even though there were no failures")
    }

    // there should still be a leader.
    cfg.checkOneLeader()

    cfg.end()
}
```

ticker函数

3A测试脚本是先通过构建多个raft实例，然后开启ticker函数进行集群领导人选举和心跳传输

```
// ticker 选举超时与心跳超时逻辑
func (rf *Raft) ticker() {
    for !rf.killed() {
        select {
        case <-rf.electionTimer.C:
            //选举超时逻辑
            rf.mu.Lock()
            rf.ChangeState(Candidate)
            rf.currentTerm += 1
            // rf.persist()
            // start election
            rf.StartElection()
            rf.electionTimer.Reset(RandomElectionTimeout()) //在分裂投票的情况下，重
置选举计时器
            rf.mu.Unlock()
        case <-rf.heartbeatTimer.C:
            //心跳超时逻辑
            rf.mu.Lock()
            if rf.state == Leader {
                //重新发送心跳
                rf.BroadcastHeartbeat()
                rf.heartbeatTimer.Reset(StableHeartbeatTimeout())
            }
            rf.mu.Unlock()
        }
```

```
        // Your code here (3A)
        // Check if a leader election should be started.

        // pause for a random amount of time between 50 and 350
        // milliseconds.
        // ms := 50 + (rand.Int63() % 300)
        // time.Sleep(time.Duration(ms) * time.Millisecond)
    }
}
```

这里介绍ChangeState函数，是进行raft身份转变的，在转变的时候涉及定时器的充值与暂停操作

```
// ChangeState 更改状态并重置超时器
func (rf *Raft) ChangeState(NewState NodeState) {
    if rf.state == NewState {
        return
    }
    DPrintf("{Node %v} changes state from %v to %v", rf.me, rf.state, NewState)
    rf.state = NewState
    switch rf.state {
    case Follower:
        // Follower 节点定期检查是否需要发起选举，这个定时器用于触发选举逻辑
        rf.electionTimer.Reset(RandomElectionTimeout())
        //Follower 状态下，节点会被 Leader 定期发送心跳信号，不需要自己发送心跳
        rf.heartbeatTimer.Stop()
    case Candidate:
    case Leader:
        //节点会定期发送心跳信号给 Follower，以保持其领导地位
        rf.heartbeatTimer.Reset(StableHeartbeatTimeout())
        //因为作为 Leader，节点无需再发起选举
        rf.electionTimer.Stop()
    }
}
```

定时器设置

心跳传输的超时时间 一定要远远小于领导人选举超时时间，不然就会出现领导人心跳还没传就已经开始新的一轮选举了。这里领导人选举设置未1000ms-1999ms之间的随机值，而心跳超时定时器则是固定的。

```
// 超时设置
const ElectionTimeout = 1000
const HeartbeatTimer = 125

// ============================================定时器设置
// 超时设置
const ElectionTimeout = 1000
const HeartbeatTimer = 125

type LockedRand struct {
    mu   sync.Mutex
    rand *rand.Rand
}

func (r *LockedRand) Intn(n int) int {
    r.mu.Lock()
```

```
    defer r.mu.Unlock()
    return r.rand.Intn(n)
}

// 初始化全局随机器
var GlobalRand = &LockedRand{
    rand: rand.New(rand.NewSource(time.Now().UnixNano())), // 通过当前的 Unix 时间戳（单位是纳秒）作为种子来初始化
}

// RandomElectionTimeout 设置随机选举超时时间
func RandomElectionTimeout() time.Duration {
    return time.Duration(ElectionTimeout+GlobalRand.Intn(ElectionTimeout)) * time.Microsecond //1000-1999之间
}

// StableHeartbeatTimeout 设置心跳超时时间
func StableHeartbeatTimeout() time.Duration {
    return time.Duration(HeartbeatTimer) * time.Millisecond //0-124
}

//                                                  定时器设置
==========================================
```

选举

选举主要逻辑如下：

1. 构建请求参数
2. 向每个peer（不包括自己，因为自己直接为自己投票）发送请求投票RPC--（这里启用go协程。因为，在raft集群中，即使成为候选人由于网络等各种问题，在某个时候他的状态可能已经被改变。所以在RPC之后先锁住，然后检查状态，合理则处理投票）
3. 查看投票是否满足半数要求
    - 如果满足，则转为Leader
    - 如果没有投票给候选人，说明选人要么日志落后 要么周期落后，周期落后直接将为Follower
4. 结束

```
// StartElection raft开始选举
func (rf *Raft) StartElection() {
    //这里应该不加锁
    rf.votedFor = rf.me //为自己投票
    args := rf.genRequestVoteArgs()
    grantedVotes := 1
    for peer := range rf.peers {
        if peer == rf.me {
            continue
        }
        go func(peer int) {
            reply := new(RequestVoteReply) // 返回一个指向类型 T 的指针
            if rf.sendRequestVote(peer, args, reply) {
                rf.mu.Lock()
                defer rf.mu.Unlock()
                if args.Term == rf.currentTerm && rf.state == Candidate { //这里再次检查是避免rpc在调用过程中超时导致，当前Term和state发生了变化
                    if reply.VoteGranted { //该peer 投票给了这个候选人
```

```
                        grantedVotes += 1
                        if grantedVotes > len(rf.peers)/2 {
                            rf.ChangeState(Leader)
                            rf.BroadcastHeartbeat()
                        }
                    } else if reply.Term > rf.currentTerm { //没有投票给候选人，说明选人要么日志落后 要么周期落后
                        rf.ChangeState(Follower)
                        rf.currentTerm, rf.votedFor = reply.Term, -1
                    }
                }
            }
        }(peer)
    }
}
```

在开始选举之前，我们应该构建好用于RPC的参数和响应，RPC的参数保存在rpc.go文件中

```
// example RequestVote RPC arguments structure.
// field names must start with capital letters!
type RequestVoteArgs struct {
    // Your data here (3A, 3B).
    Term         int
    CandidateId  int
    LastLogIndex int
    LastLogTerm  int
}

// example RequestVote RPC reply structure.
// field names must start with capital letters!
type RequestVoteReply struct {
    // Your data here (3A).
    Term        int
    VoteGranted bool
}
```

在构建好结构体之后，构建请求参数函数

```
func (rf *Raft) genRequestVoteArgs() *RequestVoteArgs {
    args := &RequestVoteArgs{
        Term:         rf.currentTerm,
        CandidateId:  rf.me,
        LastLogIndex: rf.getLastlog().Index,
        LastLogTerm:  rf.getLastlog().Term,
    }
    return args
}
```

当然RPC过程调用是官方给出的

```
// example code to send a RequestVote RPC to a server.
// server is the index of the target server in rf.peers[].
// expects RPC arguments in args.
// fills in *reply with RPC reply, so caller should
// pass &reply.
```

```
// the types of the args and reply passed to Call() must be
// the same as the types of the arguments declared in the
// handler function (including whether they are pointers).
//
// The labrpc package simulates a lossy network, in which servers
// may be unreachable, and in which requests and replies may be lost.
// Call() sends a request and waits for a reply. If a reply arrives
// within a timeout interval, Call() returns true; otherwise
// Call() returns false. Thus Call() may not return for a while.
// A false return can be caused by a dead server, a live server that
// can't be reached, a lost request, or a lost reply.
//
// Call() is guaranteed to return (perhaps after a delay) *except* if the
// handler function on the server side does not return.  Thus there
// is no need to implement your own timeouts around Call().
//
// look at the comments in ../labrpc/labrpc.go for more details.
//
// if you're having trouble getting RPC to work, check that you've
// capitalized all field names in structs passed over RPC, and
// that the caller passes the address of the reply struct with &, not
// the struct itself.
func (rf *Raft) sendRequestVote(server int, args *RequestVoteArgs, reply
*RequestVoteReply) bool {
    ok := rf.peers[server].Call("Raft.RequestVote", args, reply)
    return ok
}
```

接下来就是投票处理函数：

投票规则如下：

1. 任期比较：如果候选人的任期 (args.Term) 小于当前服务器的任期 (rf.currentTerm)，则拒绝投票。
2. 已经投过票的候选人：如果当前服务器已经在同一任期内投过票，且投给了另一个候选人，那么拒绝投票给当前请求的候选人。
3. 投票限制：每个服务器在同一任期内只能投一次票。如果在当前任期内已经投票，不能再投票给其他候选人，除非当前的任期被更新（当选举成功或者新任期开始时）
4. 如果当前周期小于候选人周期，则说明当前周期较旧，需要转为Follow状态，是否投票还是要进一步看日志的新旧
5. 日志一致性:如果候选人的日志比当前服务器的日志更新（通过 isLogUpToDate 函数判断），则投票给该候选人。

```
// example RequestVote RPC handler.
func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
    // Your code here (3A, 3B).
    //根据 Raft 协议，服务器在投票时会依据以下规则判断是否可以投票给某个候选人

    rf.mu.Lock()
    defer rf.mu.Unlock()
    defer DPrintf("{Node %v}'s state is {state %v, term %v}} after processing
RequestVote,  RequestVoteArgs %v and RequestVoteReply %v ", rf.me, rf.state,
rf.currentTerm, args, reply)

    //如果候选人请求的任期比当前服务器的任期小，拒绝投票。
    //如果请求的任期和当前服务器的任期相同，但当前服务器已经投票给了其他候选人，则拒绝投票。
    // 如果rf.votedFor!=-1是不是每次变成follow都要给他的投票设置为-1
```

```
    //1.任期比较：如果候选人的任期（args.Term）小于当前服务器的任期（rf.currentTerm），则拒绝投票。
    //2.已经投过票的候选人：如果当前服务器已经在同一任期内投过票，且投给了另一个候选人，那么拒绝投票给当前请求的候选人。
    //3.投票限制：每个服务器在同一任期内只能投一次票。如果在当前任期内已经投票，不能再投票给其他候选人，除非当前的任期被更新（当选举成功或者新任期开始时）
    if rf.currentTerm > args.Term || (rf.currentTerm == args.Term && rf.votedFor != -1 && rf.votedFor != args.CandidateId) {
        reply.VoteGranted = false
        reply.Term = rf.currentTerm
        return
    }
    //日志更新的判断标准是：候选人的日志的任期大于当前服务器的日志，或者在同一任期下，候选人的日志索引更大或相等。
    // 如果当前周期小于候选人周期，则说明当前周期较旧，需要转为Follow状态，是否投票还是要进一步看日志的新旧
    if rf.currentTerm < args.Term {
        rf.ChangeState(Follower)
        //如果候选人的任期大于当前服务器的任期，服务器会更新自己的任期并投票给该候选人
        rf.currentTerm, rf.votedFor = args.Term, -1
    }

    //4.日志一致性:如果候选人的日志比当前服务器的日志更新（通过 isLogUpToDate 函数判断），则投票给该候选人。
    // 如果候选人的log不是最新的，拒绝投票
    if !rf.isLogUpToDate(args.LastLogIndex, args.LastLogTerm) {
        reply.VoteGranted = false
        reply.Term = rf.currentTerm
        return
    }

    rf.votedFor = args.CandidateId
    rf.electionTimer.Reset(RandomElectionTimeout())
    reply.Term, reply.VoteGranted = rf.currentTerm, true
    return
}
```

```
//6.5840/src/raft/util.go

// getLastlog 获取最后一条log日志
func (rf *Raft) getLastlog() LogEntry {
    return rf.logs[len(rf.logs)-1]

}

// getLastlog 获取第一条log日志
func (rf *Raft) getFirstlog() LogEntry {
    return rf.logs[0]
}
```

这里论文中有提到日志是否最新，判断日志新的规则：

1. 谁的最后一个日志的周期大，谁就新
2. 或者在周期相等的情况下，谁的log长谁就新

```go
// isLogUpToDate 判断传入的日志是否比当前的日志“新”
func (rf *Raft) isLogUpToDate(index int, term int) bool {
    //如果候选人请求的日志比当前日志“新”，且任期合理，服务器可以投票给该候选人。
    // 否则，服务器拒绝投票或继续等待。
    lastlog := rf.getLastlog()
    if term > lastlog.Term || (term == lastlog.Term && index >= lastlog.Index) {
        return true
    } else {
        return false
    }
}
```

心跳

心跳处理上和发送请求投票逻辑基本差不多

```go
// BroadcastHeartbeat Leader发送心跳
func (rf *Raft) BroadcastHeartbeat() {
    for peer := range rf.peers {
        if peer == rf.me {
            continue
        }
        SugarLogger.Infof("%v Server 发送心跳", rf.me)
        go func(peer int) { //向每个领导者发送心跳，所以要开go协整
            //发送心跳
            rf.mu.RLock()
            if rf.state != Leader { //检查状态，因为可能在发送心跳中途已经有新的领导人出现了，自己的状态已经被修改了
                rf.mu.RUnlock()
                return
            }
            //
            args := rf.genAppendEntriesArgs()
            rf.mu.RUnlock()
            reply := new(AppendEntriesReply)
            if rf.sendAppendEntries(peer, args, reply) {
                rf.mu.Lock()
                if rf.currentTerm == args.Term && rf.state == Leader { //检查经过RPC后，当前leader的状态有没有改变
                    if !reply.Success {
                        //说明集群中可能有新的Leader出现了或者更新的Leader
                        if reply.Term > rf.currentTerm {
                            rf.ChangeState(Follower)
                            rf.currentTerm, rf.votedFor = reply.Term, -1
                        }
                    }
                }
                rf.mu.Unlock()
            }
        }(peer)
    }
}
```

下面是心跳的RPC调用逻辑

```go
// AppendEntriesArgs 发送心跳参数
```

```go
type AppendEntriesArgs struct {
    Term         int
    LeaderId     int
    PrevLogIndex int
    PrevLogTerm  int
    LeaderCommit int
    Entries      []LogEntry
}

// AppendEntriesReply 心跳消息结构
type AppendEntriesReply struct {
    Term    int
    Success bool
}

// genAppendEntriesArgs  构造请求投票参数
func (rf *Raft) genAppendEntriesArgs() *AppendEntriesArgs {
    args := &AppendEntriesArgs{
        Term:     rf.currentTerm,
        LeaderId: rf.me,
    }
    return args
}

// sendAppendEntries 发送心跳RPC
func (rf *Raft) sendAppendEntries(peer int, args *AppendEntriesArgs, reply
*AppendEntriesReply) bool {
    ok := rf.peers[peer].Call("Raft.AppendEntries", args, reply)
    return ok
}

// AppendEntries 处理心跳日志
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply
*AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()
    defer DPrintf("{Node %v}'s state is {state %v, term %v}} after processing
AppendEntries,  AppendEntriesArgs %v and AppendEntriesReply %v ", rf.me,
rf.state, rf.currentTerm, args, reply)

    //如果当前周期大于 发送周期，说明发送周期已经落后。设置返回消息，立刻返回
    if args.Term < rf.currentTerm {
        reply.Term, reply.Success = rf.currentTerm, false
        return
    }
    //如果当前周期落后发送周期，说明当前Server未更新leader,首先设置该Raft的周期
    if args.Term > rf.currentTerm {
        rf.currentTerm, rf.votedFor = args.Term, -1
    }
    // 改为Follow
    rf.ChangeState(Follower)
    rf.electionTimer.Reset(RandomElectionTimeout())

    reply.Term, reply.Success = rf.currentTerm, true
}
```

结果

终端src/raft下执行，执行前记得删除test.log,不然可能会由于test.log写入过慢和其过大，影响测试

```
go test -run 3A
```

```
xy@xy:~/mit2024/6.5840/src/raft$ go test -run 3A
Test (3A): initial election ...
warning: term changed even though there were no failures  ... Passed --   3.0  3
8193 1661409    0
Test (3A): election after network failure ...
  ... Passed --   5.0  3 11311 1852934    0
Test (3A): multiple elections ...
  ... Passed --  12.9  7 218880 28098595    0
PASS
ok      6.5840/raft     20.972s
```