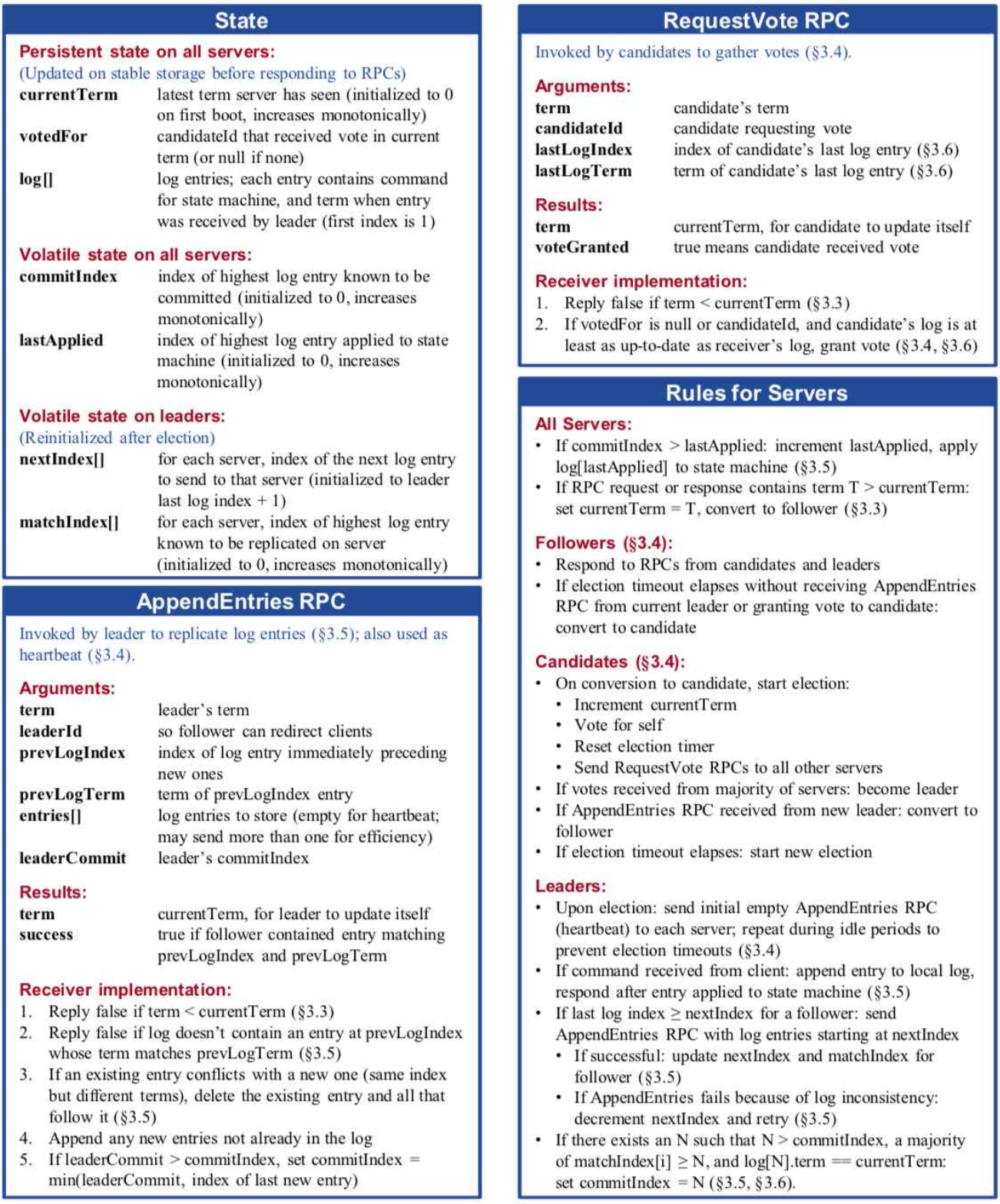
Raft 核心结构体及 RPC 实现

> 👉 要了解 Raft，就需要了解它的算法规则（方法实现）、
>
> 要实现 Raft，就需要了解它结构体定义及算法规则（方法实现）

Raft 博士论文的翻译



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 (§3.5); also used as heartbeat (§3.4).

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 (§3.3)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5)
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 (§3.4).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§3.6)
lastLogTerm	term of candidate's last log entry (§3.6)

Results:

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

Receiver implementation:

1. Reply false if term < currentTerm (§3.3)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§3.4, §3.6)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§3.5)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§3.3)

Followers (§3.4):

- 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 (§3.4):

- 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 RPC (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§3.4)
- If command received from client: append entry to local log, respond after entry applied to state machine (§3.5)
- 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 (§3.5)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§3.5)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§3.5, §3.6).

结构体

Raft 结点

```go
// A Go object implementing a single Raft peer.
type Raft struct {
    mu sync.Mutex // Lock to protect shared access to this peer's state
    // 所有对等端的RPC终结点，是一个数组。
```

```
    peers []*labrpc.ClientEnd // RPC end points of all peers
    // 用于持久化此对等方状态的对象
    persister *Persister // Object to hold this peer's persisted state
    // 表示该对等端在peers数组中的索引
    me int // this peer's index into peers[]
    // 如果该对等端已停止，则设置为1。
    dead int32 // set by Kill()

    // 2A
    // 当前最新的日志索引，从1开始
    logIndex int // current newest log index, start from 1
    // 当前的任期号，初始为0
    currentTerm int
    // 当前的状态（领导者、跟随者、候选人）
    currentState int
    // 当前的领导者ID
    currentLeader int
    // every time term increased, votedFor should be set none
    // 在当前任期内获得选票的候选人ID，如果没有则为-1。
    votedFor int

    // 用于保护随机数生成器的互斥锁
    randLock *sync.Mutex

    // 用于发送心跳信号的定时器
    heartBeatTimer *time.Timer
    // 用于检查是否需要发起新选举的定时器。
    electionTimeoutTimer *time.Timer

    // 2B
    // 用于将已提交的日志条目发送到状态机的通道。
    applyCh chan ApplyMsg

    // 包含所有日志条目的数组
    log []Entry
    // 已提交的最高日志条目的索引
    commitIndex int
    // 已应用于状态机的最高日志条目的索引
    lastApplied int

    // use for leader, reset after each election
    // 对于每个服务器，需要发送给它的下一个日志条目的索引。
    nextIndex []int
    // 对于每个服务器，已知已复制到该服务器的最高日志条目的索引。
    matchIndex []int

    // locked by mu
```

```
52      //  一个布尔数组，表示每个对等端的复制器是否处于活动状态
53      replicatorStatus []bool
54      // 用于唤醒等待已提交日志条目的状态机的条件变量
55      applyCond *sync.Cond
56  }
```

日志条目

```
1  // 日志条目
2  type Entry struct {
3      // 该条目所属的任期号
4      Term int
5      // 该条目在日志中的索引位置
6      Index int
7      // 该条目所存储的具体数据，可以是任何类型的数据，由具体实现决定
8      Data interface{}
9  }
```

ApplyMsg

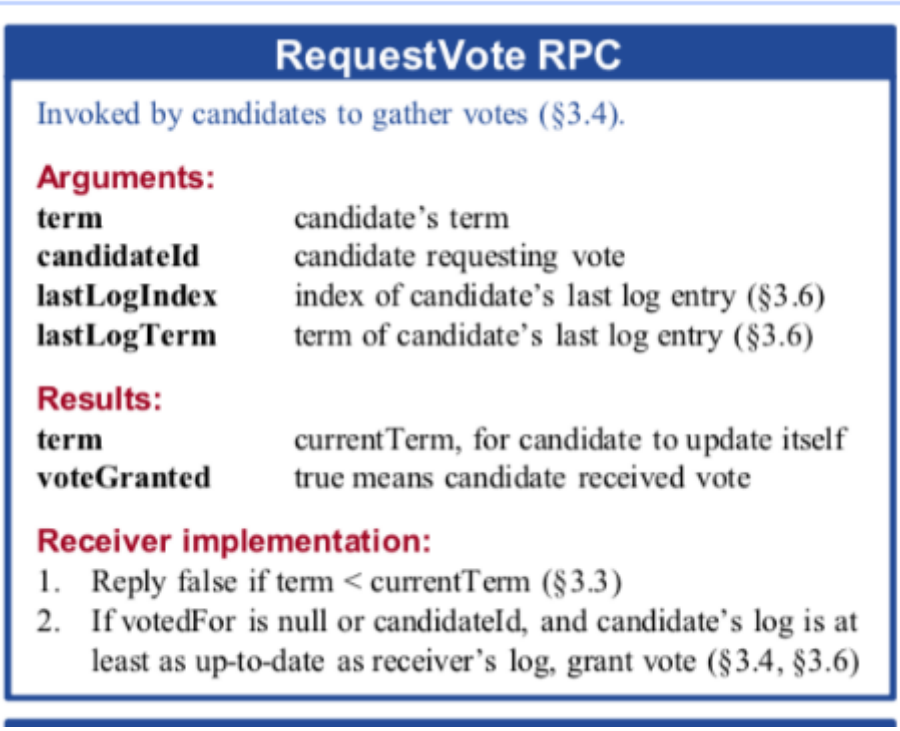
```
1  // ApplyMsg 提交命令时形成的
2  type ApplyMsg struct {
3      // 表示命令是否有效
4      CommandValid bool
5      // 具体的命令
6      Command interface{}
7      // 命令 Index
8      CommandIndex int
9      // used in kv raft
10     CommandTerm int
11
12     // For 2D:
13     SnapshotValid bool
14     Snapshot      []byte
15     SnapshotTerm  int
16     SnapshotIndex int
17 }
```

PersistData

```go
// 包含了 Raft 状态的持久化数据
/**
它包含了当前任期（CurrentTerm）、
该服务器在当前任期内投票给了哪个候选人（VotedFor）、
当前最新的日志索引（LogIndex）以及日志条目（Log）等信息。
这些信息可以通过 Persister 对象进行持久化存储，
以便在服务器重启时恢复 Raft 状态。
**/
type PersistData struct {
    CurrentTerm int
    VotedFor    int

    LogIndex int
    Log      []Entry
}
```

RPC 方法

RequestVote RPC



参数结构体

```go
type RequestVoteArgs struct {
    // 2A
    Term        int
    CandidateId int
```

```
5
6       // 2B
7       // 候选人最后一条日志条目的索引值
8       LastLogIndex int
9       // 候选人最后一条日志条目的任期号
10      LastLogTerm int
11  }
```

响应结构体

```
1  type RequestVoteReply struct {
2      // Your data here (2A).
3      Term int
4      // 指示候选人是否获得了投票
5      VoteGranted bool
6  }
```

函数实现

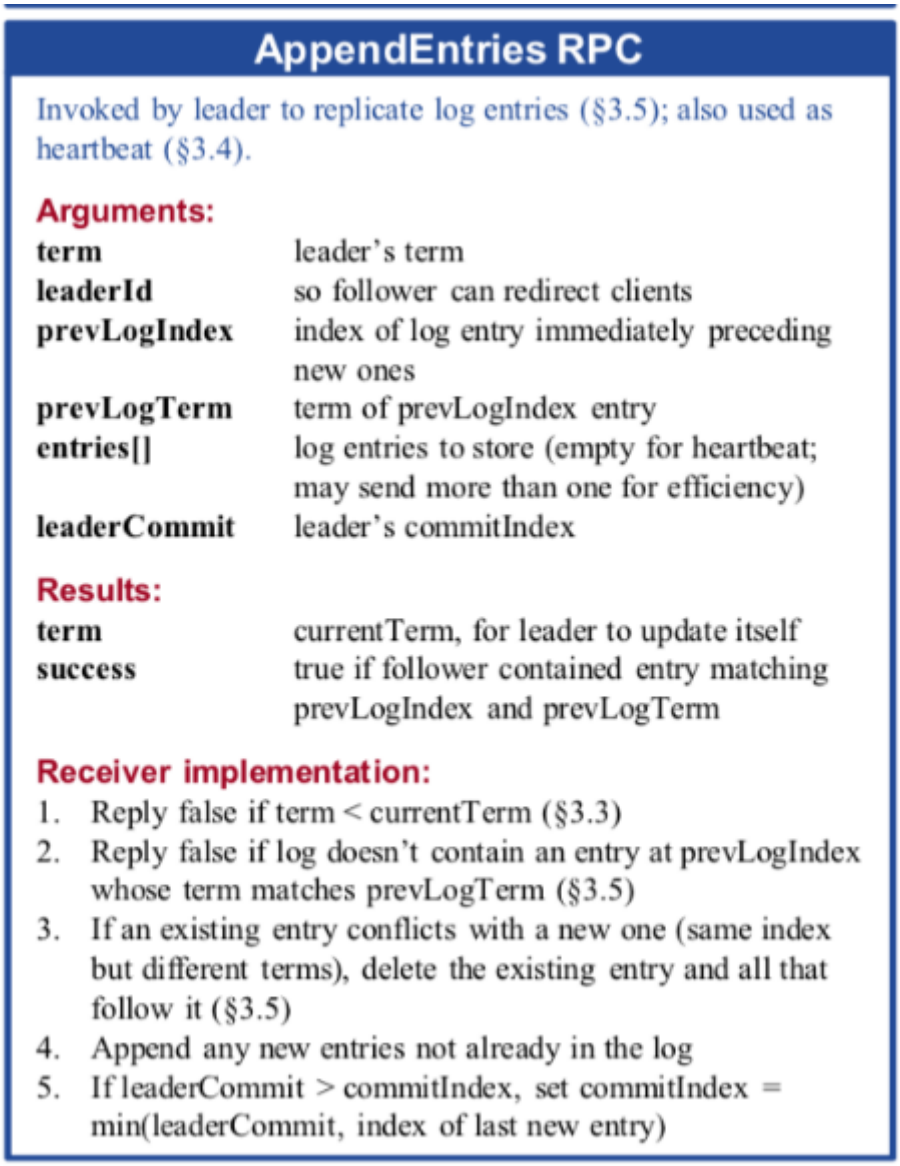
> **Receiver implementation:**
> 1. Reply false if term < currentTerm (§3.3)
> 2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§3.4, §3.6)

1.若term < currentTerm 返回false
2.若votedFor是null或candidateid，并且candidate的log至少和接受者的log保持更新，则赋予投票

```
1  // RequestVote
2  // example RequestVote RPC handler.
3  func (rf *Raft) RequestVote(args *RequestVoteArgs, reply *RequestVoteReply) {
4      // Your code here (2A, 2B).
5
6      // 加锁，避免并发问题
7      rf.mu.Lock()
8      defer rf.mu.Unlock()
9
10     // 规则 1 :
11     // 如果收到的投票请求的任期小于当前任期，则拒绝投票
12     if args.Term < rf.currentTerm {
```

```
13          reply.VoteGranted = false
14          reply.Term = rf.currentTerm
15          return
16      }
17
18      // 保存原来的状态，用于后面判断是否需要重置选举定时器
19      originalState := rf.currentState
20
21      // 如果收到的投票请求的任期大于当前任期，
22      // 则更新当前节点的任期，并转变成 FOLLOWER 状态
23      if args.Term > rf.currentTerm {
24          logPrint("%v: Receive larger term %v>%v in requestVote, convert to follo
25          rf.incTermWithoutLock(args.Term)
26          rf.changeStateWithoutLock(FOLLOWER)
27          // 暂时还不要 reset，只有投票给它才 reset
28          //rf.electionTimeoutTimer.Reset(getRandomElectionTimeout())
29          if originalState == LEADER {
30              // 用于修复可能无法启动 timer 的 bug
31              rf.electionTimeoutTimer.Reset(rf.getRandomElectionTimeout())
32          }
33      }
34
35      // 设置 reply 的任期为当前节点的任期
36      reply.Term = rf.currentTerm
37
38       // 规则 2 :
39      // 如果当前节点还没有投票给其它节点，并且收到投票请求的节点的日志比当前节点新，则投票给
40      if rf.votedFor == NONE && rf.isLargerHostLogWithoutLock(args.LastLogIndex, a
41          // 只有 follower 才会到这一步
42          logPrint("%v: Voted for candidate: %v, term: %v", rf.me, args.CandidateI
43          rf.votedFor = args.CandidateId
44          reply.VoteGranted = true
45          rf.persistWithoutLock(nil)
46          rf.electionTimeoutTimer.Reset(rf.getRandomElectionTimeout())
47      } else {
48          // 如果当前节点已经投票给其它节点或者收到的投票请求的日志不够新，则拒绝投票
49          logPrint("%v: Already voted for candidate: %v or log is not up-to-date,
50          reply.VoteGranted = false
51      }
52  }
```

AppendEntries RPC



AppendEntries RPC

Invoked by leader to replicate log entries (§3.5); also used as heartbeat (§3.4).

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 (§3.3)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

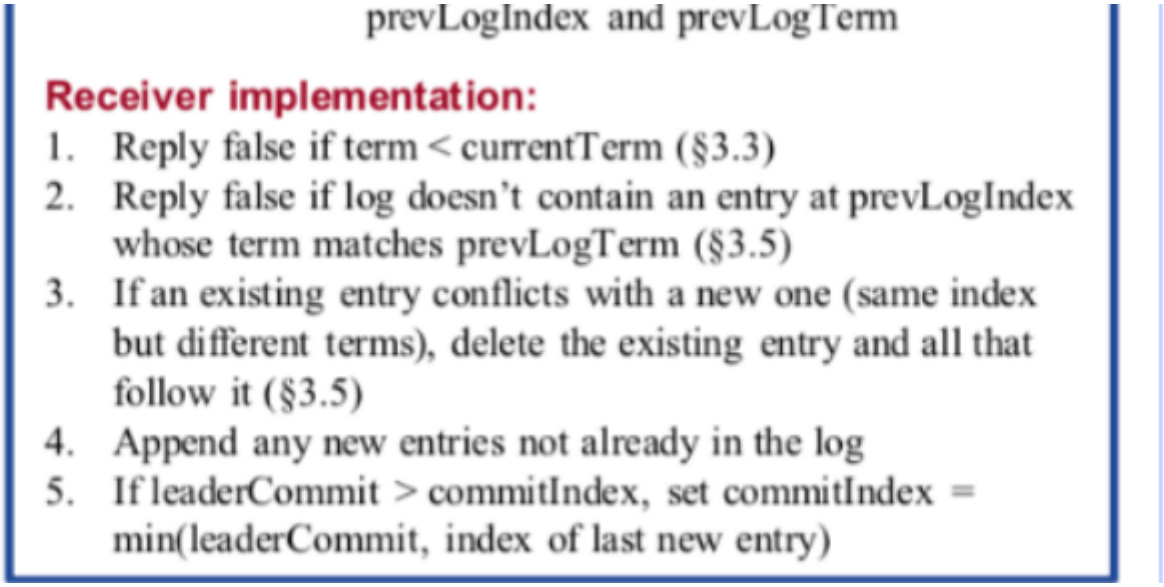
参数结构体

```go
type AppendEntriesArgs struct {
    // 领导者当前的任期号。
    Term int
    // 领导者的 Id，用于跟踪领导者当前是谁
    LeaderId int

    // 2b
    // 当前待追加日志条目的前一条日志条目的索引
    PrevLogIndex int
    // 当前待追加日志条目的前一条日志条目的任期号
    PrevLogTerm int
    // 待追加的日志条目列表
    Entries []Entry
    // 领导者已经提交的日志条目索引，该值一定大于等于 PrevLogIndex
    LeaderCommit int
}
```

响应结构体

```go
type AppendEntriesReply struct {
    // 表示当前Follower的任期号，Leader用来更新自己的任期号；
    Term int
    // 表示当前的AppendEntries RPC请求是否成功，如果成功则为true，否则为false；
    Success bool

    // not official
    // 表示冲突的日志条目的索引，用于优化日志的复制。
    ConflictIndex int
}
```

函数实现


prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§3.3)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

> 1.若term小于currentterm返回false
>
> 2.若log不包含prevlogindex中的一个entry，其中这个logindex的term是符合prevlogterm的，返回false（一致性检查）
>
> 3.若一个已有的entry和一个新的发生矛盾（同一个index但是不同terms），删除已有的entry和所有follow它的
>
> 4.增加任何新的不在log中的entries
>
> 5.若leaderCommit > commitIndex, 设置commitIndex = min(leaderCommit, 最新entry的index)

```go
// 用于在Raft集群中的leader节点向follower节点发送附加日志条目的请求。
func (rf *Raft) AppendEntries(args *AppendEntriesArgs, reply *AppendEntriesReply
    rf.mu.Lock()
```

```
4       defer rf.mu.Unlock()
5
6       // 规则 1 :
7       // 如果请求的term小于本地节点的currentTerm,
8       // 说明Leader节点已经落后了，Follower节点返回false，并返回自己的currentTerm
9       if args.Term < rf.currentTerm {
10          reply.Success = false
11          reply.Term = rf.currentTerm
12          return
13      }
14
15      // 如果请求的term大于本地节点的currentTerm,
16      // 说明本地节点已经过期，需要更新currentTerm
17      if args.Term > rf.currentTerm {
18          rf.incTermWithoutLock(args.Term)
19      }
20
21      // 重置electionTimeoutTimer，因为接收到了Leader节点的AppendEntries请求
22      // 此时 term 一定相等了
23      rf.electionTimeoutTimer.Reset(rf.getRandomElectionTimeout())
24
25      // 由于接收到Leader节点的请求，Follower节点需要变为FOLLOWER状态
26      // 有可能自己是和 leader 一样 term 的candidate, 收到来自同 term 的 leader，强制转
27      rf.changeStateWithoutLock(FOLLOWER)
28
29      // 更新currentLeader
30      rf.currentLeader = args.LeaderId
31      reply.Term = rf.currentTerm
32
33      // 获取本地节点的第一条日志的索引号
34      /**
35       func (rf *Raft) getBeforeLogIndexWithoutLock() int {
36          return rf.log[0].Index
37       }
38      **/
39      beforeLogIndex := rf.getBeforeLogIndexWithoutLock()
40
41      // 如果请求的 PrevLogIndex 小于本地节点的 beforeLogIndex,
42      // 说明 Leader节点已经落后了，Follower节点返回false，并返回冲突的索引号
43      if args.PrevLogIndex < beforeLogIndex {
44          reply.Success = false
45          reply.ConflictIndex = NONE
46          return
47      }
48
49      // 如果请求的 PrevLogIndex大于本地节点的 logIndex 或者请求的 PrevLogTerm 与本地节
50      // 说明请求的日志和本地节点不一致
```

```
51      if args.PrevLogIndex > rf.logIndex || rf.log[args.PrevLogIndex-beforeLogInde
52          // 日志不一致，返回false，并加速日志回溯
53          logPrint("Follower %v, Received log is not consistent with me, reply fal
54          reply.Success = false
55
56          // 如果请求的 PrevLogIndex大于本地节点的 logIndex，
57          // 说明 Leader节点的日志比 Follower节点多，Follower节点返回自己的 logIndex作
58          // not heartbeat, do something to accelerated log backtracking
59          if args.PrevLogIndex > rf.logIndex {
60              reply.ConflictIndex = rf.logIndex
61              logPrint("Follower %v: Own logIndex %v < args.PrevLogIndex %v, set c
62          } else {
63              // 一致性检查?
64              // 如果请求的 PrevLogIndex 小于等于本地节点的 logIndex，说明有冲突，需要找
65              conflictTerm := rf.log[args.PrevLogIndex-beforeLogIndex].Term
66              for i := args.PrevLogIndex; i > beforeLogIndex; i-- {
67                  if rf.log[i-beforeLogIndex].Term != conflictTerm {
68                      reply.ConflictIndex = i + 1
69                      break
70                  }
71              }
72              if reply.ConflictIndex == 0 {
73                  reply.ConflictIndex = beforeLogIndex + 1
74              }
75              logPrint("Follower %v: Own logIndex %v >= args.PrevLogIndex %v, set
76          }
77      } else {
78          reply.Success = true
79          // 如果leader传来的日志不为空
80          if len(args.Entries) > 0 {
81              logPrint("Follower %v: Receive log and merge it, term: %v, argsPrevI
82              // 循环处理leader传来的每个日志条目
83              for i, v := range args.Entries {
84                  // 根据当前日志条目的索引计算其在rf.log中的索引
85                  index := args.PrevLogIndex + 1 + i
86                  // 判断当前日志条目是否在rf.log中已存在
87                  if len(rf.log)-1 >= index-beforeLogIndex {
88                      // 如果已存在，则覆盖原有日志条目
89                      // overwrite existed log
90                      rf.log[index-beforeLogIndex] = v
91                  } else {
92                      // 如果不存在，则将当前日志条目添加到 rf.log 末尾
93                      // append new log
94                      rf.log = append(rf.log, v)
95                  }
96
97                  // 更新rf.logIndex为最后一个日志条目的索引
```

```
98                  rf.logIndex = index
99              }
100             // 持久化rf.log，注意此处不能加锁，否则会导致死锁
101             rf.persistWithoutLock(nil)
102         } else {
103             // 如果leader传来的日志为空，则说明是心跳消息
104             logPrint("Follower %v: Receive heartbeat from %v", rf.me, args.Leade
105         }
106
107         // 规则 5:
108         // 更新commitIndex
109         // update commitIndex
110         // 如果leader传来的commitIndex大于follower的commitIndex
111         if args.LeaderCommit > rf.commitIndex {
112             // 此处有坑，不能用 len(rf.log) 来确定 commitIndex，因为这不能代表你接收的
113             // 计算follower应该更新的commitIndex
114             rf.commitIndex = min(args.LeaderCommit, args.PrevLogIndex+len(args.E
115             // 唤醒applyCond条件变量的等待者
116             rf.applyCond.Signal()
117         }
118     }
119 }
```

InstallSnapshot RPC

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIndex	the snapshot replaces all entries up through and including this index
lastTerm	term of lastIndex
lastConfig	latest cluster configuration as of lastIndex (include only with first chunk)
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. If lastIndex is larger than latest snapshot's, save snapshot file and Raft state (lastIndex, lastTerm, lastConfig). Discard any existing or partial snapshot.
6. If existing log entry has same index and term as lastIndex and lastTerm, discard log up through lastIndex (but retain any following entries) and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load lastConfig as cluster configuration)

参数结构体

```go
// 用于封装发送给follower的安装快照请求
// 相当于将日志变成快照（为日志瘦身）
type InstallSnapshotArgs struct {
    // leader的任期号
    Term     int
    LeaderId int
    // 该快照中包含的最后一条日志的索引值
    LastIncludedIndex int
    // 该快照中包含的最后一条日志的任期号
    LastIncludeTerm int
    // 快照数据，是经过Raft层面处理后的状态数据
    Data []byte
}
```

响应结构体

```go
type InstallSnapshotReply struct {
    Term int
}
```

函数实现

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. If lastIndex is larger than latest snapshot's, save snapshot file and Raft state (lastIndex, lastTerm, lastConfig). Discard any existing or partial snapshot.
6. If existing log entry has same index and term as lastIndex and lastTerm, discard log up through lastIndex (but retain any following entries) and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load lastConfig as cluster configuration)

1. 如果term < currentTerm立刻回复
2. 如果是第一个分块（offset 为 0）则创建新的快照
3. 在指定的偏移量写入数据
4. 如果 done为 false，则回复并继续等待之后的数据
5. 保存快照文件，丢弃所有存在的或者部分有着更小索引号的快照
   假如说Follower已经有快照了，并且快照最后索引为1000，而新的快照的索引为2000，则将前面的快照丢弃。
6. 如果现存的日志拥有相同的最后任期号和索引值，则后面的数据继续保留并且回复
   意思说接收节点如果有相应的日志了，则后面的日志保留，此消息可以直接回复。
   打个比方，如果Follower B的索引已经到2002，此索引对应的term为102，其中2000索引的term为101，如果这时收到一个安装快照的消息，最后1条的term为101，最后1条的索引为2000，通过对比发现此日志已经存在节点上，并且Term也对的上，因此2001之后的日志保留。
7. 丢弃全部日志
   上面条件满足后，将快照保存到本地，本地所有日志全部丢弃。
   当然前提是前面的条件都不满足，具体不细述。
8. 能够使用快照来恢复状态机（并且装载快照中的集群配置）
   恢复状态机就不用说了，直接拿快照恢复状态机的数据，举例来说KV系统，发送的快照如果只有a=1, b=2这样的状态，即把所有数据清空，只保留上面2条数据。

   并且装载快照中的集群配置，意思是说快照还包含集群配置信息，主是要为了支持集群成员更新；

   所以快照必须以下信息：
   最后一条日志的Index；
   最后一条日志的Term；
   生成快照时的集群配置信息；
   状态机数据；

https://www.cnblogs.com/szprg/p/13873864.html

```
// 用于 Leader 节点向 Follower 节点发送 InstallSnapshot RPC 请求，
// 安装快照并更新 Follower 节点的状态。
func (rf *Raft) InstallSnapshot(args *InstallSnapshotArgs, reply *InstallSnapsho
    // 首先获取 Raft 实例的互斥锁，以防止多个线程同时修改 Raft 实例的状态
    rf.mu.Lock()
    defer rf.mu.Unlock()
```

```
7
8       // 将 reply.Term 设置为当前 Raft 实例的 currentTerm，以便返回给 Leader。
9       reply.Term = rf.currentTerm
10
11      // 规则 1 ：
12      // 收到的 args.Term 比当前 Raft 实例的 currentTerm 小
13      if args.Term < rf.currentTerm {
14          return
15      }
16
17      //  args.Term 比当前 Raft 实例的 currentTerm 大
18      if args.Term > rf.currentTerm {
19          // 更新 currentTerm 并将 Raft 实例的状态更改为 FOLLOWER
20          // 此时，该 Raft 实例将重置选举超时计时器以避免成为 Leader。
21          // 如果 Raft 实例原来的状态是 Candidate，则此时会放弃 Candidate 身份。
22          logPrint("%v: Receive larger term %v>%v in installSnapshot", rf.me, args
23          rf.incTermWithoutLock(args.Term)
24      }
25      // 不管怎么样，直接 change state 到 follower
26      rf.changeStateWithoutLock(FOLLOWER)
27      rf.electionTimeoutTimer.Reset(rf.getRandomElectionTimeout())
28
29      //  Leader 发送的快照数据比 Raft 实例当前的 commitIndex 小或相等
30      if args.LastIncludedIndex <= rf.commitIndex {
31          // 认为该数据已过时，不需要处理，直接返回。
32          // outdated data
33          logPrint("Follower %v: Receive outdated installSnapshot, %v<=%v", rf.me,
34          return
35      }
36
37      // 如果快照数据有效，则创建一个 ApplyMsg 结构体，
38      // 其中 SnapshotValid 被设置为 true，以指示该消息是基于快照数据的。
39      applyMsg := ApplyMsg{
40          SnapshotValid: true,
41          Snapshot:      args.Data,
42          SnapshotTerm:  args.LastIncludeTerm,
43          SnapshotIndex: args.LastIncludedIndex,
44      }
45
46      // 将 ApplyMsg 发送到 applyCh 通道中，以通知应用程序更新其状态。
47      // 注意，这是在一个新的 goroutine 中进行的，以防止阻塞 Raft 实例的其他操作。
48      go func() {
49          logPrint("Follower %v: Apply snapshot term: %v, index: %v", rf.me, apply
50          rf.applyCh <- applyMsg
51      }()
52  }
```