# DisEX01 Raft Leader Election

刘凡维 MF1933058

## 1. 实验环境

Ubuntu 18.4.1 下使用 Go 语言完成

```
lfw@lfw-pc:~/桌面$ go version
go version go1.10.4 linux/amd64
lfw@lfw-pc:~/桌面$
```

## 2. 问题分析

首先下载 NJU-DisSys-2017-master 文件，在 src/raft 目录下的 raft.go 文件中通过补全代码来实现 Raft Leader Election 功能。主要任务分为以下三个方面：

- 补全 RequestVoteArgs 和 RequestVoteReply 的数据结构
- 修改 Make()函数，实现当出现超时的情况时，通过发送 RequestVote RPC 的方式来开始一次 Election
- 定义 AppendEntries 的数据结构，并实现 AppendEntries RPC handler

## 3. Raft 结构与算法设计

代码的最终实现主要参考了 In Search of Understandable Consensus Algorithm 的 Section 5 的部分

**定义 follower、candidate、leader 三种状态**

```
47  const follower = 1
48  const candidate =2
49  const leader = 3
```

**补全 Raft 结构体**

```
56  type Raft struct {
57      mu        sync.Mutex
58      peers     []*labrpc.ClientEnd
59      persister *Persister
60      me        int // index into peers[]
61
62      // Your data here.
63      // Look at the paper's Figure 2 for a description of what
64      // state a Raft server must maintain.
65      currentTerm int   // latest term server has seen 当前任期
66      votedFor    int   // candidatedId that received vote in current term 投票给某个candidateId
67      status      int   // follower or candidate or leader _server当前的状态
68      voteCount   int   // 当前candidate获得的选票数目
69      log         []Log // log entries;each entry contains command 日志集合
70                        // for state machine,and term when entry was received by leader
71      heartbeatCh chan bool
72      voteCh      chan bool
73      leaderCh    chan bool
74  }
```

**获取节点当前状态（判断是否为 leader）**

```go
func (rf *Raft) GetState() (int, bool) {
    //获取节点当前状态
    var term int
    var isleader bool
    // Your code here.
    term = rf.currentTerm
    if rf.status == leader{
        isleader = true
    } else {
        isleader = false
    }
    return term, isleader
}
```

**随机生成 timeout，150ms-300ms**

```go
//设置timeout 时间为150ms-300ms
func genElectionTimeout() time.Duration {
    return time.Duration(rand.Intn(150)+150) * time.Millisecond
}
```

**RequestVote 结构体与 AppendEntries 结构体设计**

```go
//发出心跳
type AppendEntriesArgs struct {
    Term     int    //laeder's term
    LeaderId int    //so follower can redirect clients
}
//心跳回应
type AppendEntriesReply struct {
    Term    int     //currentTerm,for leader to update itself
    Success bool    //true if follower contained entry matching
                    //preLogIndex and preLogTerm
}
//要求投票
type RequestVoteArgs struct {
    // Your data here.
    Term         int    //candidate's term
    CandidateId  int    //candidate requesting vote
    LastLogIndex int    //index of candidate's last log entry
    LastLogTerm  int    //term of candidate's last log entry
}
//投票回应
type RequestVoteReply struct {
    // Your data here.
    Term        int     //currentTerm,for candidate to update itself
    VoteGranted bool    //true means candidate to update itself
    ServerId    int     //follower's id
}
```

**Make()函数设计**

一个 server 存在三种可能的状态，分别为 leader、candidate 和 follower。

*leader*：向所有节点发送 heartbeat，并且在每一个新周期重新发送 heartbeat 以维持自己的 leader 身份。对于所有收到 heartbeat 的 server 存在两种情况，若 leader 的 term 不小于当前 server 的 term 则该 server 则为 follower，若 leader 的 term 小于当前 server 的 term 则 server 成为 candidate 状态，并开始发起一个 Election。

*candidate*：每一个 server 从 follower 状态转换为 candidate 时都发起一次 Election，并令 term 加 1。

选举时首先投自己一票，同时像其他节点发送 RequestVote RPC 请求，如果收到的选票数超过 server 数的一般，则成为 leader。如果选举超时，则重新开始新一轮选举。如果收到了 leader 发来的 heartbeat，则判断 leader 的 term 与 currentTerm，如果 leader 的 term 不小于自己发起选举时的 term 则该 server 从 candidate 转换为 follower 状态，反之则不承认该 leader，继续选举。

*follower*:响应来自 leader 的 heartbeat 以及来自 candidate 的 RequestVote，在发出响应后，更新自己的响应周期，若在出现响应周期的 timeout 时既未收到 leader 发来的 heartbeat，也未收到 candidate 发来的投票请求，则转换为 candidate 状态。

```
379  func Make(peers []*labrpc.ClientEnd, me int,
380      persister *Persister, applyCh chan ApplyMsg) *Raft {
381      rf := &Raft{}
382      rf.peers = peers
383      rf.persister = persister
384      rf.me = me
385
386      // Your initialization code here.
387      rf.status = follower
388      rf.votedFor = -1
389      rf.currentTerm = 0
390      rf.heartbeatCh = make(chan bool) //建立心跳通道
391      rf.voteCh = make(chan bool)      //建立投票通道
392      rf.leaderCh = make(chan bool)    //建立通知成为leader通道
393
394      // initialize from state persisted before a crash
395
396
397      go rf.startElection()
398
399      return rf
400  }
```

**startElection 函数实现（三种状态的转换）**

```
356  func (rf *Raft) startElection() {
357      for {
358          switch rf.status {
359          case follower:
360              select {
361              case <-rf.heartbeatCh:
362              case <-rf.voteCh:
363              //没收到心跳和投票通知，超时，变成候选人
364              case <-time.After(genElectionTimeout()):
365                  rf.status = candidate
366              }
367          case candidate:
368              //发起投票
369              rf.getLeader()
370          case leader:
371              //行使leader权利
372              go rf.broadcastAppendEntries()
373              time.Sleep(time.Duration(50) * time.Millisecond)
374          }
375
376      }
377  }
```

**getLeader 函数（对 Election 过程 candidate 行为的实现）**

```go
func (rf *Raft) getLeader() {
    rf.mu.Lock()
    //开始下一任选举并给自己投上一票
    rf.currentTerm++
    rf.votedFor = rf.me
    rf.voteCount = 1
    rf.mu.Unlock()
    go rf.broadcastRequestVote()
    select {
    //超时 什么也不做
    case <-time.After(genElectionTimeout()):
    //收到心跳 造反失败 变成follower
    case <-rf.heartbeatCh:
        rf.status = follower
    //选举成功 成为leader
    case <-rf.leaderCh:
        rf.status = leader
        //rf.print()
    }
}
```

**AppendEntries RPC handler 的实现**

AppendEntries 函数首先判断 leader 的 term 是否过期，若没有过期，则更新 term，各个 server 角色不变。若 leader 的 term 已经过期，则该 server 从 leader 状态转换为 follower 状态。

```go
func (rf *Raft) AppendEntries(args AppendEntriesArgs, reply *AppendEntriesReply) {
    rf.mu.Lock()
    defer rf.mu.Unlock()

    if args.Term < rf.currentTerm {
        // 当前 server 的 term 大于 leader 的 term ,
        // 更新 reply 中的 term, 让 leader 知道自己的 term 过期了
        reply.Term = rf.currentTerm
        reply.Success = false
        return
    } else if args.Term > rf.currentTerm {
        //leader不变 造反失败
        // 更新当前 server 的 term 并且将状态改变为 follower, 重置选举计时器
        rf.currentTerm = args.Term
        rf.status = follower
        rf.votedFor = -1
        reply.Term = args.Term
        reply.Success = true
    } else {
        reply.Success = true
    }
    go func() {
        rf.heartbeatCh <- true
    }()
}
```

sendAppendEntries 函数实现回应 heartbeat 的过程，如果一个 leader 发现 CurrentTerm 大于自己的 term，则 leader 状态会转换为 follower 状态。

```go
func (rf *Raft) sendAppendEntries(server int, args AppendEntriesArgs, reply *AppendEntriesReply) bool {
    ok := rf.peers[server].Call("Raft.AppendEntries", args, reply)
    rf.mu.Lock()
    defer rf.mu.Unlock()
    if ok {
        // 只有 leader 才进行转化
        if rf.status != leader || args.Term != rf.currentTerm {
            return ok
        }
        // 如果 reply 结果中的 Term 大于 Leader 节点的当前 Term，则 Leader 节点转换为 Follwer 状态
        if reply.Term > rf.currentTerm {
            rf.currentTerm = reply.Term
            rf.status = follower
            rf.votedFor = -1
            rf.persist()
            return ok
        }
    }
    return ok
}
```

broadcastAppendEntries 函数的作用为 leader 通过该函数实现对其余 server 发送 heartbeat 的功能

```go
func (rf *Raft) broadcastAppendEntries() {
    rf.mu.Lock()
    args := AppendEntriesArgs{rf.currentTerm, rf.me}
    defer rf.mu.Unlock()
    for i := 0; i < len(rf.peers); i++ {
        if rf.me == i || rf.status != leader {
            continue
        }
        go func(i int, args AppendEntriesArgs) {
            var reply AppendEntriesReply
            rf.sendAppendEntries(i, args, &reply)
        }(i, args)
    }
}
```

## 4. 实验演示

首先使用 export GOPATH=$PWD 命令设置环境变量，之后在./src/raft/目录下执行测试命令测试代码

实验运行截图：

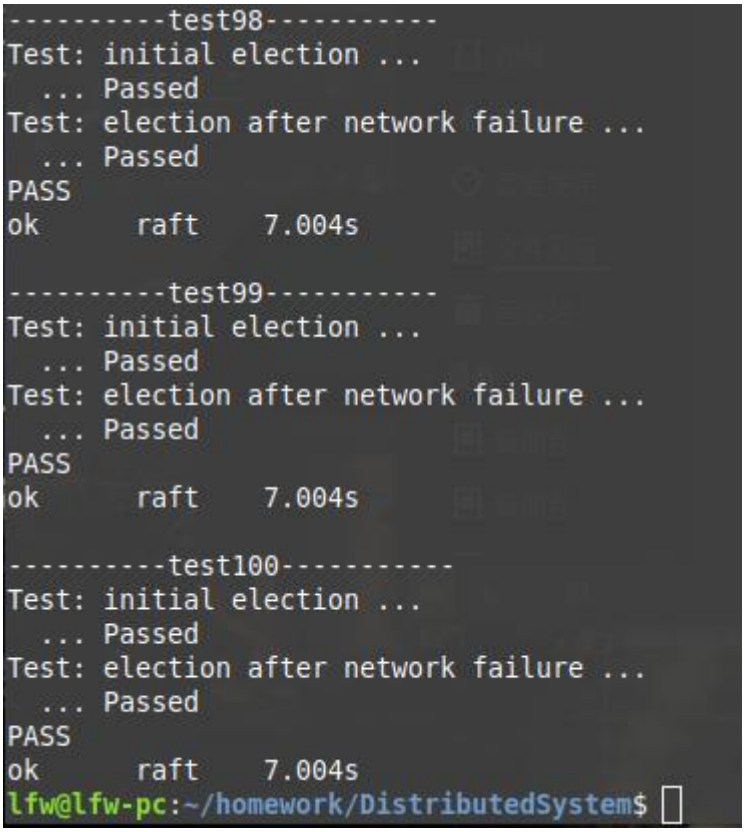
```
lfw@lfw-pc:~/homework/DistributedSystem/NJU-DisSys-2017-master$ export GOPATH=$PWD
lfw@lfw-pc:~/homework/DistributedSystem/NJU-DisSys-2017-master$ cd src/raft/
lfw@lfw-pc:~/homework/DistributedSystem/NJU-DisSys-2017-master/src/raft$ go test -run Election
Test: initial election ...
  ... Passed
Test: election after network failure ...
  ... Passed
PASS
ok      raft    7.004s
```

更进一步，通过编写脚本的方式来进行 100 次测试，测试脚本 test 代码如下：

```bash
#!/bin/bash
#进行100次测试
cd NJU-DisSys-2017-master
export GOPATH=$PWD
cd src/raft/
for (( i=1;i <= 100; i++ ))
do
    echo -e "\n----------test$i----------";
    go test -run Election;
done
```

在脚本目录下通过./test 命令进行了 100 次测试。结果显示 100 次测试均成功，且耗时范围为 7.003s-7.005s。

实验部分结果如图：




## 5. 总结

本次实验中，首先学习了 Go 语言相关知识，对于 Go 语言的并发性有了一定了解。通过观看动画以及阅读相关论文

学习了 Raft 一致性算法，最终通过论文的 Section 5 部分完成了 Raft Leader Election 的实验，并利用脚本进行了 100 次测试并全部通过。

实验过程中，由于对于 leader、candidate、follower 三种状态间转换条件的考虑不全，导致了有时候实验会失败，尤其应注意若出现 leader 收到 heartbeatReply 的 CurrentTerm 大于自己 term 的情况时，说明产生了新 leader，该 server 应从 leader 状态转换为 follower。而从 follower 状态转换为 candidate 状态时，只有产生 timeout 一种可能。

编写脚本时创建的脚本文件 test，umask 的值决定了文件的默认权限设置。对于新创建的文件只有属组和属主才有读/写权限。因此在第一次执行脚本文件时需要通过 chmod u+x test 命令赋予 test 文件属主执行文件的权限，否则会执行文件失败。