Dis 实验报告

一、实验目标

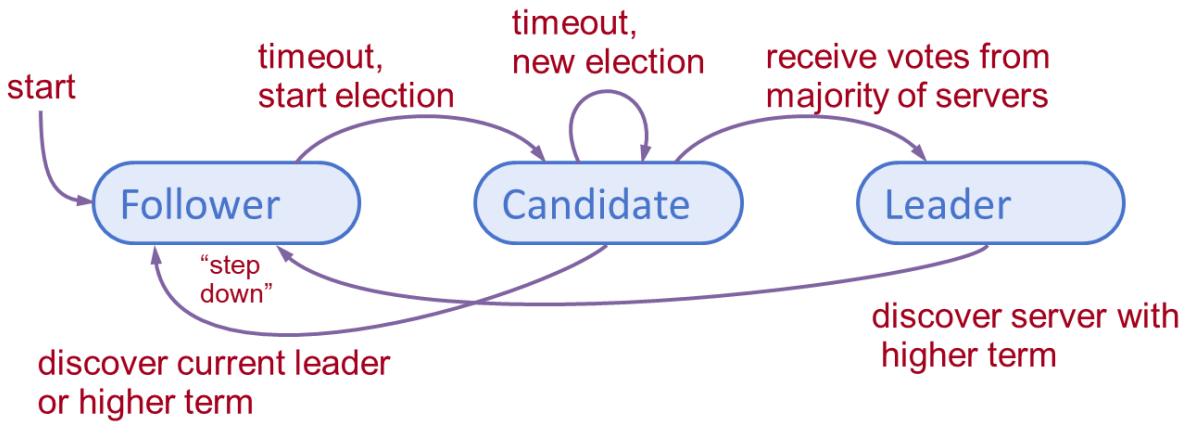
本次实验主要是用 go 语言实现简单的 raft 算法，熟悉分布式系统的一致性算法，算法的主要思想来源于文章 In search of an understandable consensus algorithm。

二、实现思路

1. 在任何时刻，每个服务器节点的状态都只有 3 种之中的一种，Leader, Follower, Candidate，在代码中用枚举关系表示。

```
const (
        Leader = iota
        Candidate
        Follower
)
```

它们之间的演化关系如下图所示：



根据这个关系，分别对以下步骤与代码实现展示如下。

2.发起领导选举

当先前的 Leader 宕机的时候，需要进行选举以得到一个新的 Leader 节点。因此对 Raft 结构的变量域设计如下：

```
type Raft struct {
    mu        sync.Mutex
    peers     []*labrpc.ClientEnd
    persister *Persister
    me        int // index into peers[]

    state  int // 节点状态
    term   int // 任期号
    votes  int // 得票数
    voteTo int // 投票对象
}
```

要开始一次选举过程，Followers 先要增加自己的当前任期号并且转换到候选人状态，然后它会并行的向集群中的其他服务器节点发送请求投票的 RPCs 来给自己投票。所以先要实现 RequestVoteArgs 结构体记录任期号和节点编号，实现 RequestVoteReply 结构体记录任期号和是否被投票，用来传递请求投票 RPC 的参数和反馈结果。代码实现：

```
type RequestVoteArgs struct {
   // Your data here.
   term   int //term of service
   nodeID int
}
type RequestVoteReply struct {
   // Your data here.
   term  int
   voted bool
}
```

3.投票过程

发送 RPC 请求后，若节点超时且节点不为领导人，它的状态转换成候选者，把票投给自己，然后 term 增加一，并构建请求投票 RPC 的参数，向其他节点发送投票 RPC 请求，当有反馈时处理 RPC 调用的结果。对于每一个节点，先判断当前节点日志是否比候选者的新。如果当前节点日志比候选者的新，那么就不进行投票。如果当前节点的 Term 比候选者节点的 Term 小，那么当前节点转换为 Follwer 状态，并且更新该节点的当前 Term 为候选者节点的 Term，如果能投票则投票给该候选者节点。然后判断当前节点的 Term 是否比候选者当前的 Term 大，如果大那么拒绝投票。

构建 RPC 请求的代码见 sendRequestVote 函数，负责广播请求的函数为 broadcastRequestVote，处理 RPC 请求的代码见 RequestVote 函数。

这样，就能确保在投票完毕后，term 最高且正常的节点成为 Leader，其他节点成为 Follower，当 Leader 宕机进行下一次投票时重复相同的过程。

4.一致性操作 Append Entries

和领导选举投票十分类似，利用与 RequestVoteArgs 和 RequestVoteReply 相似的 AppendEtryArgs 和 AppendEntryReply 结构体来表示附加日志 RPC 请求的参数和反馈结果。代码实现如下：

```
type AppendEntriesArgs struct {
   term     int
   leaderID int
}
type AppendEntriesReply struct {
   term     int
   recieved bool
}
```

Leader 节点需要发送 AppendEntry 给所有 Follwer 以维持和确保日志一致，对于非 Leader 节点，转换为 Candidate 状态并发出投票请求，而 Leader 节点则发出 AppendEntry 请求。在 Follwer 节点处理 AppendEntries 的 RPC 请求函数中，先判断请求中的 Term 是否比该节点的当前 Term 大，如果小则拒绝该请求。

否则节点转换为 Follwer 状态，并重置选举参数。如果 RPC 请求中的日志项为空，则说明该 RPC 请求为 Heartbeat，则提交未提交的日志。对于 Leader 节点，如果反馈结果中的 Term 大于 Leader 节点的当前 Term，则 Leader 节点转换为 Follwer 状态。如果 AppendEntries 的 RPC 请求成功，则更新相应的数据域，然后统计系统中是否大部分节点都追加了新的日志项，如果是则提交该日志项。如果 RPC 请求失败，则说明 Follwer 节点的日志不一致。

对于 Leader 节点构建并发送心跳广播的函数见 sendAppendEntries 和 broadcastAppendEntries，其他节点接收并处理 RPC 请求的函数见 AppendEntries。这样就能保证领导人必须从客户端接收日志然后复制到集群中的其他节点，并且强制要求其他节点的日志保持和自己相同。

三、实验结果

```
PS C:\Users\Fox> d:
PS D:\> cd .\github\
PS D:\github> cd .\NJU-DisSys-2017\
PS D:\github\NJU-DisSys-2017> cd .\src\raft\
PS D:\github\NJU-DisSys-2017\src\raft> go test -run Election
Test: initial election ...
  ... Passed
Test: election after network failure ...
  ... Passed
PASS
ok      _/D_/github/NJU-DisSys-2017/src/raft    8.752s
PS D:\github\NJU-DisSys-2017\src\raft>
```

四、总结

提前上手，多看文章，了解实现思路，不懂就问。