进度

- 完成project1 project2AA 2AB
- 通过1 2AA
- 2AB测试部分未通过

project 1

1A 实现 storage engine

即实现storage interface

```
type Storage interface {
    // Other stuffs
    Write(ctx *kvrpcpb.Context, batch []Modify) error
    Reader(ctx *kvrpcpb.Context) (StorageReader, error)
}
```

`NewStandAloneStorage()`:初始化

- dbPath：根据conf获取
- kvPath := filepath.Join(dbPath, "kv")
- kvDB :通过 engine_util.CreateDB 创建
- engines : engine_util.NewEngines生成引擎

`Reader()`: 返回StorageReader接口，通过StorageReader读
运用 engine_util 提供的接口，实现StorageReader

- 成员：txn *badger.Txn
- GetCF()获取CF的key对应的value，调用engine_util.GetCFFromTxn
- IterCF获取CF迭代器，调用engine_util.NewCFIterator
- Close()关闭Reader，调用r.txn.Discard()

`Write()`:直接执行写操作 对batch中的每一个modify

- storage.Put:调用engine_util.PutCF
- storage.Delete：调用engine_util.DeleteCF

1B

实现`RawGet`、`RawScan`、`RawPut`、`RawDelete`

- `RawGet()`: 获取对应 CF_Key 的 Value
    1. 调用1A中的Reader
    2. reader.GetCF()，获取CF的key对应的value
    3. 返回RawGetResponse，包含value和NotFound
- `RawScan()`：在指定 CF 中，从 StartKey 开始扫描键值对，直到Limit

    1. 调用1A中的Reader
    2. 获取迭代器，reader.IterCF(req.Cf)
    3. 从StartKey开始遍历，加入Kvs
    4. 返回RawScanResponse，包含所有搜索到的Kvs
- `RawPut()`：写入键值对
    1. 使用Storage.Modify,类型为put,从而指定CF、key、value
    2. 调用server.storage.Write写入
    3. 返回空RawPutResponse
- `RawDelete()`：删除键值对
    1. 使用Storage.Modify,类型为delete,从而指定CF、key
    2. 调用server.storage.Write写入
    3. 返回空RawDeleteResponse

问题与总结

1. defer reader.Close()
   defer+函数名，延迟调用，return前执行
2. 运行测试不通过
   许多方法需要对err进行错误处理
3. 通过参数、返回类型、调用路径，判断函数的作用
4. 文档注释中包含很多提示

project 2

2A raft算法实现

3个板块：

- 2aa: Leader election 领导人选举
- 2ab: Log replication 日志复制
- 2ac: Rawnode interface Rawnode接口

其中 2aa 的 Leader election 和 2ab 的 Log replication 是对 Raft 基础算法的实现。2ac 的 Rawnode 对 Raft 封装，构成一个能收发消息的完整节点。

预备知识：raft算法

特点

- 强领导人：日志条目只从领导人发送给其他的服务器
- 领导选举：使用一个随机计时器来选举领导人
- 成员关系调整：调整中的两种不同的配置集群中大多数机器会有重叠，这就使得集群在成员变换的时候依然可以继续工作

3个角色

1. 领导者（Leader）：由选举产生。负责接收处理客户端请求，和同步日志条目给其他节点。每隔一段时间向跟随者发送心跳来维持领导状态。

2. 候选者（Candidate）：由跟随者选举超时转变而来。发送选举信息和统计选票，如果获得超过半数的同意选票，则转换为领导者；否则转换为跟随者。

3. 跟随者（Follower）：各节点初始为 Follower 。被动接收领导者的日志条目信息、响应候选者投票。如果选举超时则转换为候选者。

日志信息

1. 指令
2. 任期号
3. 日志索引

投票

1. 条件：候选人的term比跟随者大
2. 条件：跟随者不比候选人新（任期号大，日志长）-paper5.4.1
3. 先到先得
4. 跟随者投票时设置任期与候选人一致
5. 投票分裂：轮空，继续下一轮任期

超时设置自动选举：

1. 选举超时（某个节点自己产生随机时间）：选举超时后，追随者将成为候选人，并开始新的选举任期...如果接收节点还没有在这一任期中投票，那么它将投票给候选人...
2. 心跳超时：领导者开始向其追随者发送 Append Entries 信息。这些信息的发送间隔由心跳超时时间指定，然后，关注者会回复每条 Append Entries 信息。这一任期将持续到追随者停止接受心跳并成为候选人为止。

提交：

1. leader节点复制到跟随节点
2. leader等待大多数节点写入条目，提交该条目，并通知追随者
3. 跟随者根据追加条目中的leader_commit得知leader提交进度，在此之前的follower日志都可以提交
4. 不会计算之前任期的日志副本数目进行提交

日志复制：通过使用与心跳相同的 "附加条目 "信息来实现

1. 用户发送请求给领导日志，然后在下一次心跳时将更改发送给追随者。
2. 一旦大多数追随者承认该条目，该条目即生效.....并向客户端发送响应。

客户端如何知道谁是leader？随机发送请求给一个节点

1. leader
2. follower知道leader
3. 宕机，重新发送

一致性检查：检查上一条日志是否一致

2AA 领导者选举

总体思路

- 核心是Msg的收发与处理逻辑
- tick() step()推动Msg的传递

```
//raft数据结构
type Raft struct {
    id uint64

    Term uint64
    Vote uint64
    RaftLog *RaftLog            //存储日志，包括Entries，commit和apply等

    Prs map[uint64]*Progress    //集群节点的信息，包括id，Next，Match

    State StateType             //角色信息，0-Follower 1-Candidate 2-Leader
    votes map[uint64]bool       //作为候选者时接收到的选票信息统计

    msgs []pb.Message           //待发消息

    Lead uint64                     //节点当前的 Leader

    heartbeatTimeout int            ///心跳间隔
    electionTimeout int             //选举超时时间
    heartbeatElapsed int            //目前经过的心跳时间
    electionElapsed int             //目前经过的选举时间

    //add
    randomElectionTimeout int       //随机选举超时时间，每个节点不同

    // (Used in 3A leader transfer)
    leadTransferee uint64
    // (Used in 3A conf change)
    PendingConfIndex uint64
}
```

tick()逻辑时钟

系统间隔一段时间自动调用 Raft 节点中的 `tick()` 函数。每调用一次，逻辑时间增加1。分角色做不同处理：

Follower/Candidate：

1. electionElapsed增加1。
2. 若选举超时，则发起选举。给自己发送本地消息 `pb.MessageType_MsgHup`，触发选举。

Leader：

1. heartbeatElapsed增加1。
2. Leader心跳超时，处理：更新心跳并bcast心跳给所有追随者。给自己发送 `pb.MessageType_MsgBeat`。

step()消息处理器

根据不同角色与Msg类型，对消息进行处理

```
func (r *Raft) Step(m pb.Message) error {
    var err error = nil
    switch r.State {
    case StateFollower:
        switch m.MsgType {
        case pb.MessageType_MsgHup:
            r.becomeCandidate()
            r.broadcastVoteRequest()
        case pb.MessageType_MsgPropose:
            err = ErrProposalDropped
        ...
        }
    case StateCandidate:
        switch m.MsgType {
        case pb.MessageType_MsgHup:
        ...
        }
    case StateLeader:
        switch m.MsgType {
        case pb.MessageType_MsgBeat:
        ...
        }
    }
    return err
}
```

3个角色

`becomeFollower()`

- 触发条件：从候选人退出,给别人投票
- 状态更新为Follower
- 设置任期和领导者

`becomeCandidate()`

- 触发条件：选举超时
- 设置角色
- 开启新任期
- 给自己投票

`becomeLeader()`

- 设置状态State，Lead，Vote，重置超时
- 维护每个节点的日志状态，初始化Next为lastLogIndex+1
- 发送noop entry并广播
- 更新commit index

Msg消息收发与处理

数据结构： MsgType 和 To 是必须的，其他字段含义由MsgType决定。

```
type Message struct {
    MsgType              MessageType
    To                   uint64
    From                 uint64
    Term                 uint64
    LogTerm              uint64
    Index                uint64
    Entries              []*Entry
    Commit               uint64
    Snapshot             *Snapshot    //2C
    Reject               bool
    ...
}
```

消息类型: 本地消息3种: `MessageType_MsgHup`、`MessageType_MsgBeat`、`MessageType_MsgPropose`，非本地消息三对： `RequestVote`、`AppendEntries`、`HeartBeat` 以及它们对应的 Response

```
type MessageType int32
const (
    //local Message
    MessageType_MsgHup MessageType = 0
    MessageType_MsgBeat MessageType = 1
    MessageType_MsgPropose MessageType = 2
    MessageType_MsgTransferLeader MessageType = 11
    MessageType_MsgTimeoutNow MessageType = 12

    //None local Message
    MessageType_MsgAppend MessageType = 3
    MessageType_MsgAppendResponse MessageType = 4
    MessageType_MsgRequestVote MessageType = 5
    MessageType_MsgRequestVoteResponse MessageType = 6
    MessageType_MsgHeartbeat MessageType = 8
    MessageType_MsgHeartbeatResponse MessageType = 9

    //SnapShot Message
    MessageType_MsgSnapshot MessageType = 7
)
```

Vote

- sendVoteRequest(): 由于投票条件2（跟随者不比候选人新），需要传递lastLogTerm，lastLogIndex
- broadcastVoteRequest()：领导向跟随者广播sendVoteRequest()
- handleVoteRequest()：满足任期，新，先到先得条件即可投票，发response

- sendRequestVoteResponse()：是否拒绝
- handleVoteResponse()：统计票数，成为leader

Append

- sendAppend()：
    1. 日志复制：leader发送新日志信息，从nextIndex开始
    2. 一致性检查：发送prevLogTerm，prevLogIndex,
    3. 日志提交：通知follower当前leader提交进度，Msg包含Commit索引
- handleAppendEntries()
    1. 任期检查
    2. 一致性检查
    3. 处理冲突日志，复制日志
    4. 修改提交进度
- sendAppendResponse()：reject
- handleAppendResponse()
    1. 判断是否复制成功，更新 match 和 next
    2. 更新commit信息

HeartBeat

- sendHeartbeat():发送commit索引
- handleHeartbeat(): 任期检查，一致性检查，日志提交(设置commit)
- sendHeartbeatResponse():reject
- handleHeartbeatResponse()：任期检查(设置term，state)，是否拒绝（设置next,match）

2AB 日志复制

RaftLog 结构

RaftLog 存储 Raft 中的日志信息，并提供一些日志处理的功能。

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

    // (Used in 2C)
    pendingSnapshot *pb.Snapshot

}
```

RaftLog 的结构如下：

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

```
//                              log entries
```

`entries` 存储未压缩的日志，`applied`、`committed` 和 `stabled` 分别记录了已经应用、已提交、已稳定存储的日志的最小Index。

Entry 的 Index 均是连续的，即 len(entries) == lastlogIndex - FirstIndex + 1

需实现方法：allEntries,unstableEntries,nextEnts,EntriesFromIndex,FirstIndex,LastIndex...根据index在起始，结束区间中的位置，返回对应的切片即可

测试问题

- 测试要求becomeCandidate和broadcastVoteRequest分离，(TestCandidateStartNewElection2AA)
- 只有一个节点时，直接成为leader (TestLeaderElectionInOneRoundRPC2AA)
- 消息处理时必须最先进行任期检查与更新
- leader在发送空条目之后，再修改本身的match和next