Raft协议简单实现

# 算法描述

## 基本概念

### 角色

raft协议中将Server分为以下三种角色：

1. Leader

负责Client交互和log复制，同一时刻系统中最多存在1个，

1. Follower

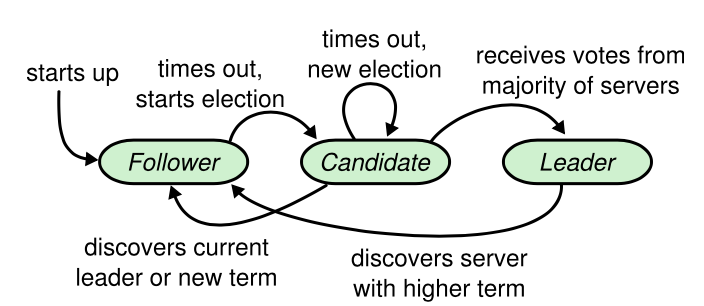
每个Server初始时的角色，被动响应来自Leader和Candidate的RPC请求，从不主动发起请求RPC；

如果长时间没收到Leader或者Candidate的RPC则转换成Candidate；

复制将Client的请求重定向到Leader。

1. Candidate

由Follower向Leader转换的中间状态，向其他Server拉选票，如果获得半数以上的支持就变成Leader。



### Server状态

1. Term

在Raft中使用了一个可以理解为周期（第几届、任期）的概念，用Term作为一个周期，每个Term都是一个连续递增的编号，每一轮选举都是一个Term周期，在一个Term中只能产生一个Leader；先简单描述下Term的变化流程： Raft开始时所有Follower的Term为0，其中一个Follower逻辑时钟到期后转换为Candidate，Term加1这时Term为1（任期），然后开始选举，这时候有几种情况会使Term发生改变：

　　1：如果当前Term为1的任期内没有选举出Leader或出现异常，则Term递增，开始新一任期选举

　　2：当这轮Term为1的周期选举出Leader后，过后Leader宕掉了，然后其他Follower转为Candidate，Term递增，开始新一任期选举

　　3：当Leader或Candidate发现自己的Term比别的Follower小时Leader或Candidate将转为Follower，Term递增

　　4：当Follower的Term比别的Term小时Follower也将更新Term保持与其他Follower一致；

可以说每次Term的递增都将发生新一轮的选举，Raft保证一个Term只有一个Leader，在Raft正常运转中所有的节点的Term都是一致的，如果节点不发生故障一个Term（任期）会一直保持下去，当某节点收到的请求中Term比当前Term小时则拒绝该请求；

1. voteFor

当前term中Server同意投票的CandidateId。为了减少竞选，冲突每个term中每个Follower只投票一次。

1. commitIndex

当Leader的日志的一条记录被大多数（超过半数）Follower追加到自己的日志中时称这条记录被提交了。commitIndex是日志中最后一条被提交的记录的索引。

1. lastApplied

当日志被提交过后才能被应用到本地状态机（写入数据库），lastApplied是日志中最后一条被应用记录的索引，lastApplied ≤ commitIndex

1. nextIndex[]

对每个Follower，nextIndex是Leader需要对其发送的下一条记录的索引，初始为Leader最后一条日志的索引加1，每次同步日志后维护

1. matchIndex[]

对每个Follower，matchIndex是Leader已知的，日志中已同步的最后一条记录的索引，初始为0，每次同步日志后维护

### RequestVote RPC

由Candidate发起投票时调用

1. 参数

term：Candidate的term

CandidateId：Candidate的id

lastLogIndex：Candidate最后一条日志记录的索引

lastLogTerm：Candidate最后一条日志记录的term

1. 返回值

term：接收者的currentTerm，用于Candidate更新自己的term

voteGranted：如果接收者同意则置为true

1. 接收者实现

如果term < currentTerm，返回false和term；

否则，如果voteFor不为空（排除当前的CandidateId），则返回false和term，因为每个Term内只能投票一次；

否则，如果Candidate的日志和接收者的日志至少一样新（比较方式为谁的lastLog的term越大谁越新，如果term相同，谁的lastLog的index越大谁越新），则同意投票，返回true和term

### AppendEntries RPC

由Leader发送用来同步日志；也可以用作heartbeat包

1. 参数

term：Leader的term

leaderId：Leader的id，用于Follower使Client重定向到Leader

prevLogIndex：新的日志条目之前的第一个索引值，也就是nextIndex-1；

prevLogTerm：日志中对应的prevLogIndex的term

entries[]：从nextIndex起到日志中的最后一条记录，如果数目为0则用作heartbeat包

leaderCommit：Leader的commitIndex

1. 返回值

term：接收者的currentTerm，用于Leader更新自己的term

success：如果日志保持一致返回true

1. 接收者实现
2. 如果 term < currentTerm 就返回 false
3. 如果日志在 prevLogIndex 位置处的日志条目的term和 prevLogTerm 不匹配，则返回 false ，并且返回自己日志的最后一条记录的索引（Leader用此来更新自己的matchIndex和nextIndex，下次同步从此处开始发送）
4. 如果已经已经存在的日志条目和新的产生冲突（相同index但是term不同），删除这一条和之后所有的
5. 添加任何在已有的日志中不存在的条目
6. 如果 leaderCommit > commitIndex，令 commitIndex 等于 leaderCommit 和 新日志条目索引值中较小的一个
7. 完成和Leader的日志同步后，同样返回自己日志的最后一条记录的索引让Leader来更新自己的matchIndex和nextIndex

## 重要步骤

### 选举

Raft的选举由定时器来触发，每个节点的选举定时器时间都是不一样的，开始时状态都为Follower某个节点定时器触发选举后Term递增，状态由Follower转为Candidate，向其他节点发起RequestVote RPC请求，这时候有三种可能的情况发生：

　　1：该RequestVote请求接收到n/2+1（过半数）个节点的投票，从Candidate转为Leader，向其他节点发送heartBeat以保持Leader的正常运转

　　2：在此期间如果收到其他节点发送过来的AppendEntries RPC请求，如该节点的Term大则当前节点转为Follower，否则保持Candidate拒绝该请求

　　3：Election timeout发生则Term递增，重新发起选举

　　在一个Term期间每个节点只能投票一次，所以当有多个Candidate存在时就会出现每个Candidate发起的选举都存在接收到的投票数都不过半的问题，这时每个Candidate都将Term递增、重启定时器并重新发起选举，由于每个节点中定时器的时间都是随机的，所以就不会多次存在有多个Candidate同时发起投票的问题。

有这么几种情况会发起选举，1：Raft初次启动，不存在Leader，发起选举；2：Leader宕机或Follower没有接收到Leader的heartBeat，发生election timeout从而发起选举;

### 日志复制

日志复制（Log Replication）主要作用是用于保证节点的一致性，这阶段所做的操作也是为了保证一致性与高可用性；当Leader选举出来后便开始负责客户端的请求，所有事务（更新操作）请求都必须先经过Leader处理，这些事务请求或说成命令也就是这里说的日志，我们都知道要保证节点的一致性就要保证每个节点都按顺序执行相同的操作序列，日志复制（Log Replication）就是为了保证执行相同的操作序列所做的工作；在Raft中当接收到客户端的日志（事务请求）后先把该日志追加到本地的Log中，然后通过heartbeat把该Entry同步给其他Follower，Follower接收到日志后记录日志然后向Leader发送ACK，当Leader收到大多数（n/2+1）Follower的ACK信息后将该日志设置为已提交并追加到本地磁盘中，响应客户端，并在下个heartbeat中Leader将通知所有的Follower将该日志存储在自己的本地磁盘中。

具体的实现流程在上述AppendEntries RPC的介绍中

# 主要实现

本次实现的的大致流程和实现都与上述算法保持一致，以下列出在实现细节上有不同的几点：

1. 在原来算法中，Client将更新请求发送到Leader后，Leader先加入自己的日志，然后同步给其他Server，当收到半数以上的Server的ACK信息后，更新自己的commitIndex，然后才能将该指令应用到本地状态机（写入数据库），这之后才能向Client端回复执行结果。每个服务器有专门的进程将日志中commitIndex之前的记录应用到自己的状态机，本实现仅有Leader写入数据库，Follower只更新了日志。
2. Client端需要实现重发请求的功能。在接收了Client的请求之后，Leader可能宕机，不能回复Client，因此Client需要重发请求。但是，此时上一条请求可能已经写入到大部分的Follower中，并且新的Leader当选之后会应用这条指令到状态机了，因此Client需要在请求中加入指令的id，Leader根据id判断，如果该指令已经应用过就直接回复Client，否则重新加入日志。
3. 未实现集群成员调整的功能。

# 测试结果记录

## 测试条件

Server数量：5

Client数量：5

每次写入数量：500

总测试次数：6

## 测试结果



说明：

1. 每次执行插入语句500条，共执行六次；
2. 时间单位为毫秒；
3. JDBC时间是仅在本机执行数据库操作的时间；
4. 单服务器时间是客户端仅和一个服务器通信，然后执行数据库操作的时间；
5. RAFT服务器时间是客户端和应用raft协议通信并进行日志复制和数据库操作的时间；
6. 第一次执行由于涉及初始化通信和JDBC，时间较长，因此统计了除去第一次执行时间的平均时间。

## 性能分析

本次测试没有测试服务器宕机情况下的性能，因为改变锁结构后，服务器宕机后加锁对象不能转移给新的Leader，此时需要客户端重发请求。

为了提高响应速度，测试中加快了Leader向其他服务器发送AppendEntriesRPC的频率（本次测试为20ms发送一次），可能会大量增加网络负担；但是如果按照论文建议的150ms发送一次AppendEntriesRPC，则写入500的时间增加到15s。

# 实际应用中的改进

1. 增加选举计数，当多次漏掉heartbeat包才认为Leader宕机，然后才能发起选举
2. 增加预选举功能，选举前先向Leader发送尝试请求，如果Leader响应则放弃选举

# 遗留问题

1. 实现异步RPC请求；
2. 服务器宕机处理