# Raft

raft是一种通过日志复制实现一致性算法，提供了和paxos相同的功能和性能，raft将一致性问题分解为三个相对独立的子问题

1. Leader选举
2. 日志复制
3. 安全性

# Leader选举

在raft中，任何时候一个服务器可以扮演以下角色：leader、follower、candidate，raft中使用任期的概念term，一个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。

Raft有两个timeout设置控制选举，第一个是election timeout，election timeout是follower等待成为candidate的时间在150ms到300ms之间，第二个是heartbeat timeout，AppendEntries RPC或心跳包文在heartbeat timeout发送。我们要求heartbeat timeout要比election timeout小很多。

**leader选举，在其中timing是非常关键的地方，只要系统满足如下timing必要条件，raft就能选举并维护一个稳定的leader：**

**broadcastTime << electionTimeout << MTBF**

* **broadcastTime：并行发送RPCS到集群中每一个节点的发送以及收到响应的平均时间**
* **election Timeout：leader选举中的timeout**
* **MTBF：对应一个节点来说，MTBF是两次失败之间时间间隔的平均值**

**broadcastTime小于等于election timeout，这样leader才能稳定往所有follower节点发送心跳消息，同时也使得split vote成为不可能。**

Raft选举是由定时器触发，每个节点选举定时器时间都不一样，默认是0-1000ms之间，开始状态都为Follower，当某个节点定时器触发选举后term递增，状态由Follower转为candidate，并向其他节点发起投票请求：

* 收到半数以上节点投票，从candidate转为leader，定期向其他节点发送heartBeat保持leader正常运转。
* 在此期间如果收到其他节点发送的AppendEntries请求，如果该节点的term大，则当前节点转为follower，否则保持candidate拒绝请求
* Election timeout发生，term递增，重新发起选举，每个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candidate收到投票数不过半问题，这时候每个candidate将term递增，重启定时器并重新发起选举，由于每个节点中的定时器时间都是随机的，所以就不会多次存在有多个candidate同时发起投票的问题

# 日志复制

Leader会把客户端请求作为一个log entry append到本机日志中，然后给其他的server发送AppendEntries RPC请求，当leader确定一个log entry被安全复制了（大多数副本已经将该命令写入日志中），就存储这条log entry到状态机中（committed），然后返回结果给客户端，下一次heartbeat带有通知其他节点提交命令的消息，Follower节点收到committed命令后将命令提交到状态机。如果某个Follower宕机了或者运行很慢，或者网络丢包，则会一直给这个Follower发送AppendEntries RPC直到日志一致。

1. Client向leader提交命令（set 5），leader收到命令（set 5）后，将命令追加到本地日志中，此时，命令处于uncommitted状态
2. Leader将命令（set 5）并发复制给所有follower，如果有节点失败或者比较慢，leader会一直重复发送，直至所有节点都保存命令道日志中。但当leader确定命令被多数节点写入到日志中，leader在本地提交（set 5）命令，并将结果返回给client节点。
3. Leader节点提交命令后，下一次heartBeat就带有通知其他节点提交命令的消息。Leader节点会记录已经提交的最大日志maxIndex，之后后续heartBeat和日志复制请求都会带上这个值，这样其他节点就知道哪些命令已经提交了（如果maxIndex等于本地最大的日志，则将未提交的日志全部提交，如果maxIndex大于本地最大日志，回复拒绝消息，leader将nextIndex--，直到找到第一个不同的日志）

当一个新的leader产生，它的日志和其他Follower的日志可能不一样，这个时候，就需要一个机制来保证日志的一致性。新leader产生后，就以leader上的log为准，其他follower要么少了数据，要么多了数据，要么既少了又多了数据。要使得Follower日志进入和自己一致的状态，leader必须找到最后两者达成一致的地方，leader针对每个follower维护一个nextIndex，表示leader给各个follower发送下一条log 的index，当一个节点刚成为leader，他将初始化所有nextIndex为自己最后一条日志的index+1，给follower发送消息，带着（termId， nextIndex - 1）term是（nextIndex - 1）位置的log的term ID，follower接收到消息后，会从自己的log中找是否存在这样的log entry，如果不存在，就回复拒绝消息，然后leader将nextIndex – 1，重复上述过程，直至消息被接收。

# 安全性

* 选举安全性：对于一个给定的任期号，最多只会有一个leader被选举出来
* Leader只附加原则：leader绝对不会删除或者覆盖自己的日志，只会增加
* 日志匹配原则：如果两个日志在相同的索引位置的日志条目、任期号相同，那么我们就认为这个日志从头到这个索引位置之间完全相同
* Leader完全特性：如果某个日志在某个任期号中已经被提交，那么这个条目必然出现在更大任期号的所有leader中
* 状态机安全特性：如果一个leader已经在给定的索引位置的日志条目应用到状态机中，那么其他任何服务器在这个索引位置不会提交一个不同的日志

# raft选举触发条件

* raft初次启动，不存在leader，发起选举
* leader宕机
* follower在election timeout时间内没有收到leader的heartBeat

# raft协议疑问

1. 当一个follower经过一个heartbeat没有收到包，就会认为leader宕机，就会开始新一轮的选举，如果集群个数很多，难免会有机器延迟很高，这样会造成频繁选主。

Raft group是一个容错单位，通常这个group会很小，一般是3、5，7就比较大了，要伸缩的话，一般是使用更多的raft group

1. 若集群中没有发生分区，而只是部分不连通，比如5个机器的集群，A、B之间互相无法连接，而与其他机器都可以联通，这种情况下，如果A为leader，B收不到heartBeat就会选主成为leader，这时A收不到B的回复，就会有两个leader存在？？

在raft协议中被投票是有前提的，只有在某个多数派中日志是最新的节点才可能被选出来，否则拥有更新日志（先比term，term相同比较log index）的节点是不会投给比自己日志还旧的节点，所以上述情形有两种可能：

在B发起投票之前，A在多数派中写操作成功，此时CDE中一定至少有两个日志比B新，之后B发起投票，由于节点不会投票给自己日志旧的节点，所以B不会成为leader。

如果B发起投票之前没有写请求，B携带新的term请求成为leader，而后成为新的leader，之后A通过CDE发现自己的term落后而转化为follower。

# Raft成员变更

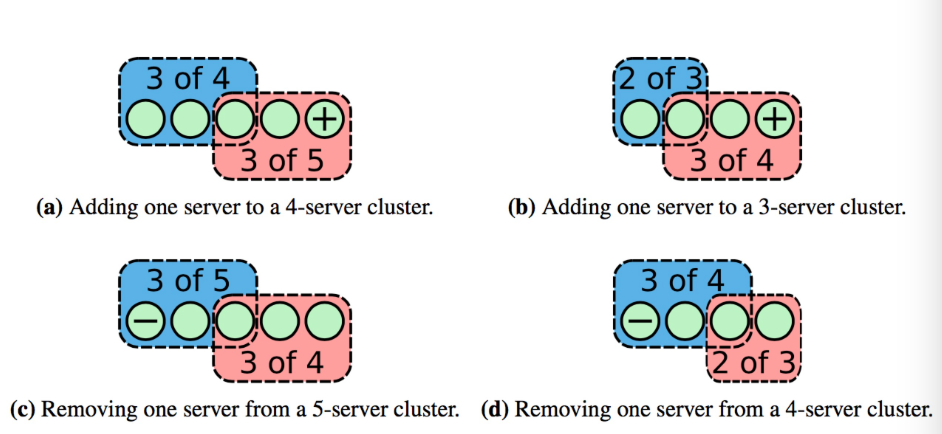
成员变更在一致性协议中稍微复杂一些，由于不同的成员不可能在同一时刻从旧成员组切换至新成员组，所以可能出现两个不相交的majority，从而导致同一个term出现两个leader，进而导致同一个index的日志不一致。

新成员在加入集群时已经知道旧成员配置，现在的问题是如何让旧成员知道新成员的存在。

当集群成员变更后面，即添加或者删除节点的时候，如何让其他节点知道成员变更了，最常用的做法，可能就是通过一个全局的协调器，譬如zookeeper和etcd这种，做一个两阶段提交的变更，但整个过程会导致服务暂时不可用。

* 解决方案1：

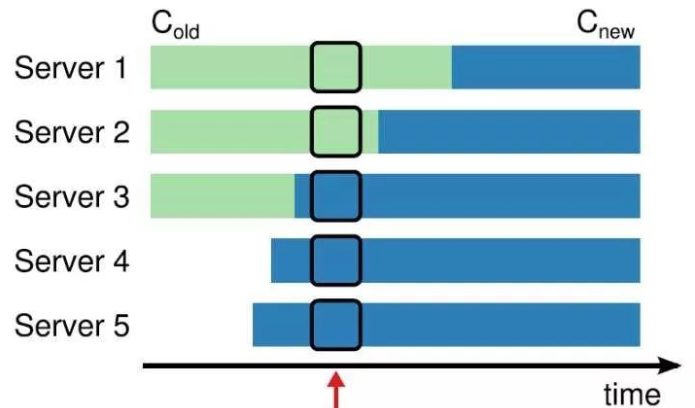
一次只增加或减少一个成员，这样能够保证任何时刻，新旧成员配置都不可能出现两个不相交的majority，所以可以直接从旧成员组切换到新成员组。



这样即使发生leader选举，也不会出现两个leader的状况，因为两个多数派必然相交。

* 解决方案2：

Joint consensus算法，旧配置1、2、3，新配置1、2、3、4、5，则某一时刻存在节点1和2仍使用旧配置，节点3、4、5使用新配置，



那么存在旧配置多数派1和2，新配置多数派3、4、5，这时候可能产生两个leader，leader1和leader5。

C-old为当前配置，C-new为目标配置，当leader收到一个成员变更请求时，会创建一个c-old-new（C-old&C-new）的配置（joint consensus），所有节点在接收到配置时就采用新的配置，不需要等到commit，joint consensus把新旧配置联系起来，代表当前时刻新旧成员配置共存，写入本地日志，在此之后新的日志同步需要保证得到C-old和C-new两个多数派的确认。

Follower收到C-old-new的log entry之后更新本地日志，并立即以该配置作为自己的成员配置，不需要等到commit之后才是用。

如果C-old和C-new两个多数派确认了C-old-new这条日志，leader就提交这条log entry，接下来leader生成一条新的log entry，其内容是新成员配置C-new，Follower接收到C-old日志后立即作为自己的成员配置，并且发现如果自己不再C-new这个成员配置中就会自动退出（节点删除）。

需要注意的是，两阶段成员变更（C-old-new和C-new）比较通用且容易理解，但实现比较复杂，同时两阶段的变更协议也会在一定程度上影响变更过程中服务的可能性，因此我们通常用的一阶段变更方案，也就是限制C-old和C-new任意多数派交集不为空，即最终只会产生一个leader，方法就是每个限制加入或是删除的节点只能有一个。

## joint consensus

当leader收到成员变更请求，会创建一个C-old-new配置，称为joint consensus，所有节点收到配置后立即采用新配置，而无需等待commit，当leader节点处于C-old-new状态时，所有日志必须收到c-old和C-new的多数派同意才能提交或者选举。

* 当C-old-new被majority确认后，leader宕机，新的leader当选必然有C-old-new，则会继续提交C-new
* 当C-old-new没有被majority确认，leader宕机，此时拥有C-old-new配置的节点或者拥有C-old的节点都有可能成为leader，若拥有C-old配置的节点成为leader，则客户端重新发送新配置文件，则继续上述过程，若拥有C-old-new配置节点成为leader，则会提交C-new日志

# 日志压缩

Raft日志会随着处理客户端请求数量增多而不断增大，在实际的系统中，由于日志越来越多而占用更多的存储空间以及宕机重启后回放的时间越长，日志不可能无限增长。

因此，需要定期清理日志，raft采用最简单的快照方法，对系统当前做快照时，会把当前状态持久化到存储中，然后快照点的日志项都可以删除。

Raft算法中每个节点单独做快照，即把当前状态机的状态写入到存储中（状态机中的状态都是已提交的log entry回放出来的），除了状态机的状态之外，raft快照中还需要一些元数据信息，包括：

* 快照中包含最后一个日志条目的索引值和任期号，记录这些信息的目的是为了使得AppendEntries RPC的一致性检查能通过，因为，在复制紧跟快照后的日志条目时，AppendEntries RPC带上需要复制的日志条目前一个日志条目（索引值和任期号term），即快照的最后一个日志条目的（索引值和任期号）
* 为了支持集群成员变更，快照保存的元数据还会存储集群最新的配置信息

当服务器完成快照后，可以删除快照最后一个日志条目及其之前的所有日志条目，以及之前的快照。

有一个特殊的情况，需要leader向follower发送整个快照的情况，例如：当一个follower日志处于leader最近一次快照之前，恰好leader删除了follower需要的日志，这是leader无法通过发送日志条目来同步了，只能通过发送完整快照。

# 客户端交互

Raft中client发送所有请求到leader，当客户端启动时，他会随机挑选一个节点进行通信

* 如果选择的节点不是leader，该节点会把leader的地址告诉client，client会把请求发送给leader
* 如果此时没有leader，那么client会timeout，client会重试其他节点，直至找到leader

当leader在提交了日志，在响应client之前崩溃了，那么client会和新的leader重试这条指令，导致这条指令被再次执行，解决方法是用幂等性去重，client对每一条指令都赋予一个唯一的序列号，然后状态机跟踪每条指令最新的序列号和相应的响应，如果接收到一条指令，他的序列号已经被执行，立即返回结果不必重新执行指令。

## 只读请求

所有的请求都会转移到leader节点上，但如果leader直接返回查询数据可能会有返回过时数据的风险，因为leader响应客户端请求时可能已经被新的leader代替了，但它自己还不知道，raft使用两个额外措施解决这个问题。

我们需要考虑两个问题，一是leader返回的可能会是过时数据，二是leader地位被其他节点得到但它自己并不知道。

* 首先，leader必须知道哪些日志条目是被提交了的，因为在它刚成为leader时，leader完全性特性保证了它一定拥有所有已经被提交的日志，但它并不知道哪些日志是被提交了的，如果查询的数据是未被提交的数据，则有可能会在它变为follower后被其他leader覆盖，为了知道这些信息，leader需要在任期开始时提交一个空的日志来处理该位置，该日志被提交后，则在这条空日志之前的日志都会被提交，这样leader就知道了所有被提交的日志。
* 第二，leader在处理只读请求时必须检查自己已经失去了leader地位，raft通过让leader先和集群中的多数派交换一次心跳包文来处理问题。另一种可选方案是leader可以依赖心跳机制，心跳信息会重置follower节点的election timeout，那么在这个election timeout时间内leader可以认为不会有新的leader出现，在这段时间内leader会直接返回查询结果，不需要先与其他节点通信。

# 10.Prevote算法

在basic raft算法中，当一个follower节点x与其他节点网络隔离，会发生如下状况：节点x在election timeout没有收到心跳报文会发起选举，并转为candidate，每次发起选举时，会把term+1，由于网络隔离，他既不会被选为leader，也会收到leader的消息，所以会一直不断发起选举，term会不断变大，一段时间之后，这个节点term会非常大，在网络恢复之后，这个节点会把他的term传播到集群的其他节点，导致其他节点更新自己的term，变为follower，然后触发重新选主，但这个旧的节点x由于其日志不是最新的，不会成为leader。这个集群被这个网络隔离过的旧节点扰动。

在prevote算法中，candidate首先要确认自己能够赢得大多数节点的投票，这样才会把自己的term增加，然后发起真正的投票，其他节点同意发起选举的条件是：

* 没有收到有效leader的心跳报文，至少有一次选举超时
* Candidate的日志足够新（term更大，或者term相同index更大）

当网络分区节点重新加入时，会中断集群问题，在prevote算法中，网络分区节点由于无法获得大多数节点的认可，因此无法增加其term，然后当他重新加入集群时，他还是无法增加term，因为其他服务器将一直收到leader的心跳报文，一旦网络分区节点收到leader的心跳报文，他将返回到follower状态，term和leader一致。

# 11.成员变更系统可用性

证明集群成员变更期间，依旧可以正常响应客户端请求并完成集群成员变更操作（这里指的是一次删除或增加一个节点的成员变更）

1. 配置变更期间任何一个步骤都可以选择新的leader

每次只增加或删除一个节点，保证在任何情况下，新旧成员配置都不可能形成两个不相交的多数派，最终也只会有一个leader

# 12.leadership transfer

当需要移除的节点是leader时，leader需要管理不包含自己的集群，直到提交之后再step down，可以通过leadership transfer将leadership转移到其他节点，然后移除原先的leader，比如当leader所在节点负载较高，需要转移到低负载机器上。

其流程如下：

* Leader停止接收新的请求
* 通过log replication使得leader和transferee的log相同，确保transferee能够赢得选举
* Leader发送timeoutNow给transferee，它会马上发起选举，leader收到transferee的选举消息后宕机