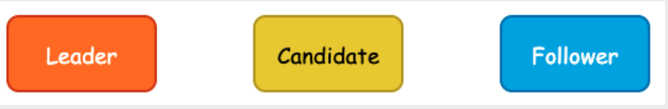
Raft

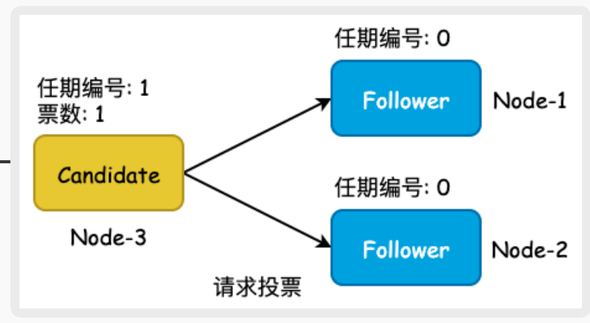
前言

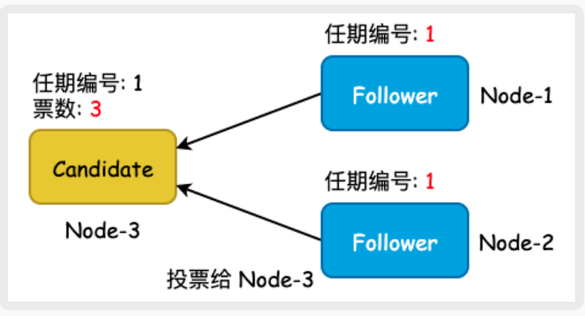
- Raft 是基于 Multi-Paxos 思想所实现的共识算法，它不应该被理解成一致性协议或者是算法
- Raft 在 Multi-Paxos 思想的基础之上做了很多的改进和限制，比如将两阶段提交改为了一阶段提交，集群节点角色包括领导者（Leader）、候选者（Candidate）和追随者（Follower），日志必须连续等特性。此外，Raft 是一个单领导者的复制模型，并要求集群中有且只有一个领导者
- 节点角色定义
  - 

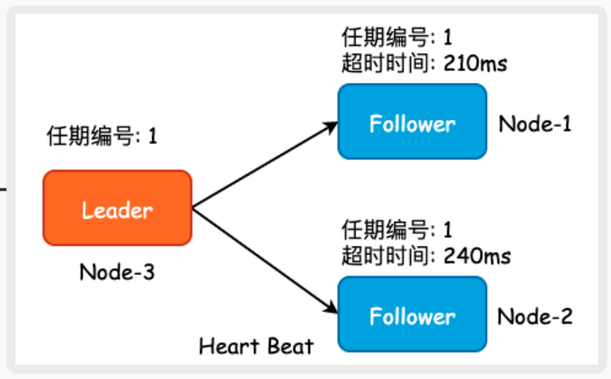
  - Raft 算法将节点角色定义为领导者（Leader）、候选者（Candidate）和追随者（Follower），并且和 Basic Paxos 算法不同的是，在同一个时刻一个节点仅能有一种角色，无法"身兼数职"。也就是说，在任何时刻，每一个服务器节点都只能是领导者、候选者和追随者中的一个
  - ① 追随者（Follower）：普罗大众的一员，主要的工作就是接收和处理领导者的消息，当和领导者的心跳包超时以后，就主动地站出来推荐自己担任下一届的领导者
  - ② 候选者（Candidate）：候选者向其它所有的节点发送投票 RPC 消息，如果得到了集群中大多数节点的投票的话，候选者将晋升为领导者
  - ③ 领导者（Leader）：处理集群中的数据写入和读取，以及将对应的日志项发送给其它的追随者，并使用心跳包告诉追随者领导者仍然存活
- 也就是说，假如集群中出现了候选者的话，要么是领导者节点客观下线，要么是候选者单方面的认为领导者下线，此时就是主观下线
  - 这和 Redis Sentinel 是不是很像？因为 Sentinel 就是使用的 Raft 共识算法来实现选主的

领导选举过程

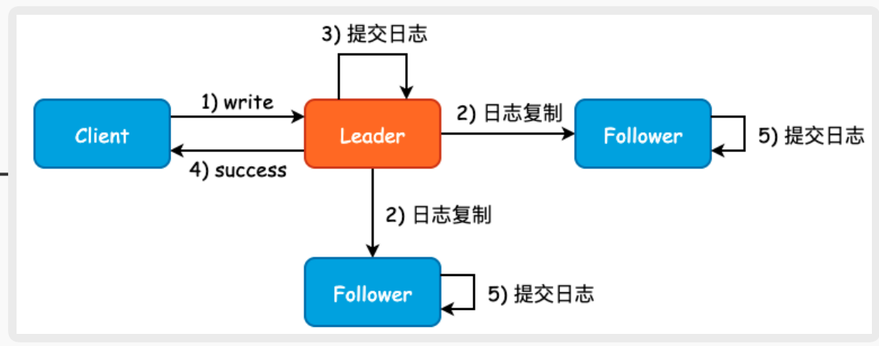
- 理解随机超时时间
  - 

  - 首先来看什么是超时时间。超时时间是指 Follower 收到 Leader 发送的心跳包的最大间隔时间，一旦 Follower 发现心跳包超时，就会立马将自身状态更新为 Candidate，发起投票
  - 随机超时时间是指每个 Follower 等待 Leader 心跳信息的超时时间间隔是随机的，在 Hashicorp/Raft 的实现中，这个随机超时时间在 [minTimeout, 2*minTimeout) 之间
  - Raft 算法巧妙地利用了随机超时时间，使得当 Leader 节点下线以后，在大多数情况下只会有一个节点率先发起投票，而不是所有节点同时发起投票，导致最终无法获得多数节点的投票导致选举失败的情况。选举时间持续的越久，系统不可用的时间也就越长，因此，随机超时时间是 Raft 算法中非常重要的一环
- ① 如上图所示，在集群初始化时，所有的节点均为 Follower，并且 Node-3 的随机超时时间更短，所以将率先触发超时事件，向 Node-1 和 Node-2 发起投票请求
  - 

- ② 若其它节点在任期编号为 1 的任期内没有进行过其它投票的话，那么就会将选票投给 Node-3，并增加自己的任期编号
  - 

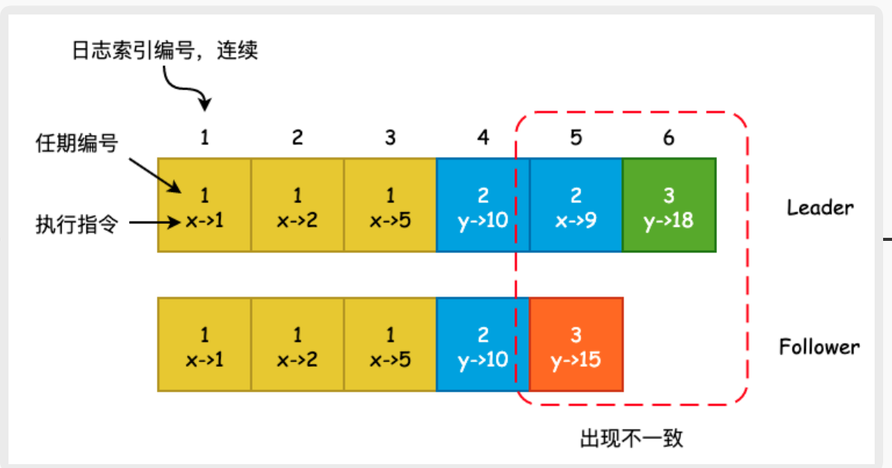
- ③ 当 Node-3 获得了大多数节点的选票以后，那么他就会晋升成为领导者，并周期性地向其它节点发送心跳信息
  - 

Raft 中的两阶段提交

- 首先我们需要明确的是，Raft 算法中日志的复制和应用（Apply）是一个两阶段提交的过程，只不过 Raft 巧妙地利用心跳包机制对其进行了优化
- 

- Client 向 Leader 提交一个写请求，Leader 根据请求写入的指令创建一个新的日志项，并追加至本地日志文件中（WAL）
- Leader 将该日志项通过 RPC 请求将其复制到其它 Follower 节点中，当多数 Follower 复制成功以后，Leader 就会将这条日志应用（Apply）到自己的状态机上，并返回成功给客户端，这是数据复制的第一阶段
- 而后，Leader 通过心跳包或者是另一个日志复制的 RPC 消息，将已经提交的最大日志项编号发送给 Follower。Follower 发现自己还没有将该条日志提交至状态机，那么就会将这条日志项应用到自身的状态机中，这是数据复制的第二阶段

日志复制

- 上述过程中是一个非常理想的日志复制状态，但在实际的日志复制中可能会发生节点宕机、网络分区，那么此时 Raft 又通过什么方式来保证日志的一致性呢？
- 

  - 当 Leader 和 Follower 的日志出现不一致时，Raft 通过以领导者为主，直接覆盖 Follower 和 Leader 不一样的日志项来实现日志的一致性
- 简而言之，日志复制 RPC 除了进行日志复制以外，还会进行额外的一致性检查，也就是检查 Follower 和当前 Leader 的日志项是否一致
- Leader 通过日志复制 RPC 进行一致性检查，并找到 Follower 节点上与自己日志项相同（日志索引编号相同、任期编号相同）日志项的最大索引值，然后复制并更新覆盖该索引值之后的日志项，可以认为是一种强制更新的方式

再议领导选举与日志复制

- 在前面的领导选举和日志复制节点中，我们描述了最简单情况，也就是 Raft 算法在正常情况下的运转流程，接下来我们就对其进行扩充，讨论异常情况下 Raft 算法是如何进行的，以及补充更多的细节
- 在领导选举的过程中，Candidate 除了提供任期编号以外，还需要提供未提交的最大日志项索引值（Max Uncommitted Log Entry Index），而不是已提交的最大日志项索引值
-

  - 假设现集群状态如左图所示，Leader 已经将 "set z = 3" 这一指令复制到了大多数节点，且 Leader 提交了这条日志项
  - 如果此时 Leader 宕机的话，那么 Follower 将不会收到提交 Log Entry Index 为 4 的心跳包，并且在心跳超时后 Follower 开始发起选举投票
  - 我们假设 Node-3 率先超时并发起投票，其最后一个日志索引项为 3，并将此信息发送给其它节点。然而其它节点的最后一个日志索引为 4，因此会拒绝给 Node-3 进行投票
  - 而后，Node-2 或 Node-4 开始发起投票，它们的 Log Entry Index 均大于等于其它的两个节点，因此会得到大多数节点的选票，成为新的 Leader。在成为新的 Leader 之后，会将未提交的日志提交（Apply），并利用心跳机制使得其它节点 Apply 此条日志
  - 这种提交方式其实就是 Raft 算法中的隐式 commit，保证数据的线性一致性
- Candidate 得到投票的前提: Candidate 节点的最后一个 LogEntry 的任期编号要比当前节点大，或者在任期编号相同的情况下，Log Entry 的 LSN（Log Serial Number）必须大于等于投票者。这里的 LSN 是指未提交日志的最大日志索引项