##### 投票策略（& election restriction）：

candidate先将自身term+1，然后发出RequestVote RPC，此RPC中会携带term、日志中最后一条entry的term、日志中最后一条entry的index；

follower通过以下两步判断完成投票：

follower首先判断自身currentTerm是否小于RPC中term，如果currentTerm > term，return false；

通过currentTerm判断后，检查RPC中最后一条entry的term和index与自身保存的最后一条entry的term及index的大小，RPC中term更大，则通过，两者term相同，则检查index，RPC中index更大，亦通过

也就是说 所谓“更新”的日志的评判标准是：该日志的term和index同时决定

##### not commiting entries from previous terms

新leader会保存旧leader时的entry，但不会提交这些已达多数的entry，当有新的entry到达时，并达到提交条件时，会自动提交起那么那些未提交的entry（根据一致性检查）；而提出这一标准的原因是：如果不这样做，会有已经被提交的日志被删除的情况出现；而此要求下，那些可能被删除的日志不会被过早的提交，也就是说raft满足了它自己提出的Leader Completeness：已被提交的日志不会被删除，但对那些已在大多数服务器达成一致但未提交的日志，认为是可以放弃的。

新leader启动后如何判定后，旧的entry是否已提交？

根据状态机是否是持久性的，如果状态机是持久性的，那么为防止重复的应用日志条目，服务器的lastApplied状态会被持久化，所以重启后，可以找到对应已经应用到状态机的索引（即说明新leader中也保存了该值）；而如果状态机是不持久的，那么服务器重启后会通过snapshot恢复，而snapshot中会持久化lastApplied，所以重启的服务器仍然拥有了最后应用的entry索引

raft中存在的可能隐患：

假定现有三台服务器A、B、C，分别对应term和log为 (3,3) (2,2) (3,3)，log均已被提交，因为(3,3)已达到半数以上；但B节点未和A、C节点保持同步（可能由于leader和B节点之间存在暂时的网络故障或网络拥堵），如果此时三节点同时掉电宕机，并且原有的leader节点C磁盘故障，更换了磁盘后，原有持久化数据全部清空，即状态变为(0,0)，如果此时B节点首先发起选主，那么C节点会投赞同票，B节点将变为leader，等B节点将所有数据刷新至AC节点，将导致A节点的(3,3)数据被删除；从结果来看，(3,3)log属于已被提交的日志，但是仍然被删除，这与raft论文中的Leader Completeness 相悖，而究其原因：数据被清空的C节点参与了投票，而C节点由于数据被清空，它对于收到的所有RequestVote均会赞同，这就导致新的leader不一定是数据最新的。

为规避此类现象，应该对状态为0的节点做出投票限制，如果所有节点状态均为0，那么正常发起投票，因为这种情况是raft集群的最初启动场景；而如果只有少数节点状态为0，那么此0节点应该被限制参与；如果半数以上的节点状态为0，似乎也可以正常参与选举投票。

为实现这种策略，每个节点应该具备全局的状态信息，即log[]