# 分布式一致性算法应用场景

分布式一致性问题，简单说就是一个或者多个进程提议了一个值后（这里一个值并不理解为一个数值，可以有很多的含义），使系统中所有进程对这个值达成一致意见。

这样的协定问题在分布式系统中很常见，比如：

如果在一个不出现故障的系统中，很容易解决分布式一致性问题。但是实际分布式系统一般是基于消息传递的异步分布式系统，进程可能会慢、被杀死或者重启，消息可能会延迟、丢失、重复、乱序等。

在一个可能发生上述异常的分布式系统中如何就某个值达成一致，形成一致的决议，保证不论发生以上任何异常，都不会破坏决议的一致性，这些正是一致性算法要解决的问题。

在分布式存储系统中经常使用多副本的方式容错，对于传统的主从同步方式，强一致性同步（主备副本都写成功返回）和异步同步（主副本写成功就返回再异步写备副本）。主从同步无法同时保证数据的一致性和可用性（CAP），而Paxos、Raft等分布式一致性算法则可在一致性和可用性之间取得很好的平衡，在保证一定的可用性的同时，能够对外提供强一致性，因此Paxos、Raft等分布式一致性算法被广泛的用于管理副本的一致性，提供高可用性。

# Paxos

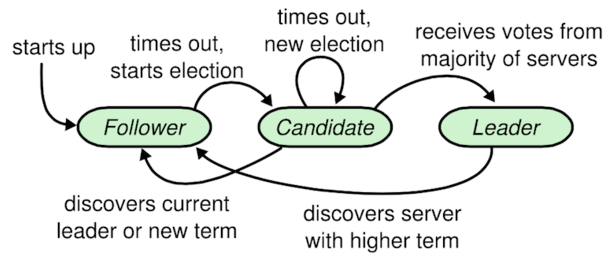
Paxos算法是基于**消息传递**且具有**高度容错**特性的一致性算法，是目前公认的解决**分布式一致性**问题最有效的算法之一。

在常见的分布式系统中，总会发生诸如机器宕机或网络异常（包括消息的延迟、丢失、重复、乱序，还有网络分区）等情况。Paxos算法需要解决的问题就是如何在一个可能发生上述异常的分布式系统中，**快速且正确地在集群内部对某个数据的值达成一致**，并且保证不论发生以上任何异常，都不会破坏整个系统的一致性。这里**某个数据的值**并不只是狭义上的某个数，它可以是一条日志，也可以是一条命令（command）。。。根据应用场景不同，某个数据的值有不同的含义。

# raft

在raft中，任何时候一个服务器可以扮演下面三个角色中的一个：

1. Leader：处理所有客户端交互，日志复制等，同一时刻系统中最多存在1个；
2. Follower：被动响应请求RPC，从不主动发起请求RPC；
3. Candidate：由Follower向Leader转换的中间状态。

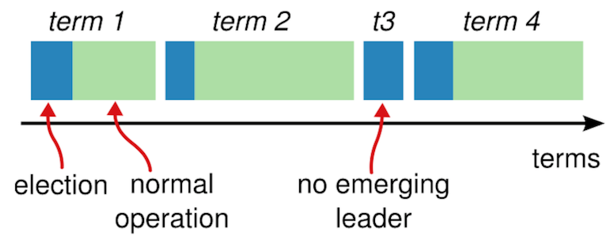


raft的工作模式是一个Leader和多个Follower模式，即我们通常说的领导者-追随者模式。这种模式下需要解决的**第一个问题**就是Leader的选举问题。**其次**是如何把日志从Leader复制到所有Follower上去。

## Terms

在分布式环境下时间同步是一个很大的问题，但是为了识别过期信息，时间信息又是必不可少的。raft中解决这个问题的办法是将时间切分为一个个的Term，可以认为是一种逻辑时间，Term就是一个数值，是全局增长的值。

1. 每个Term最多只存在1个leader；
2. 某些Term由于选举失败不存在leader；
3. 每个Server本地维护当前的Term值。



## Heartbeats and Timeout

所有的服务器都以follower的角色启动，并且启动选举定时器；

follower期望收到leader或者candidate的RPC消息，一旦收到消息，立即重置自己的选举定时器；

如果follower的选举定时器超时，会认为leader已经crash，开始新的选举。

## Leader election

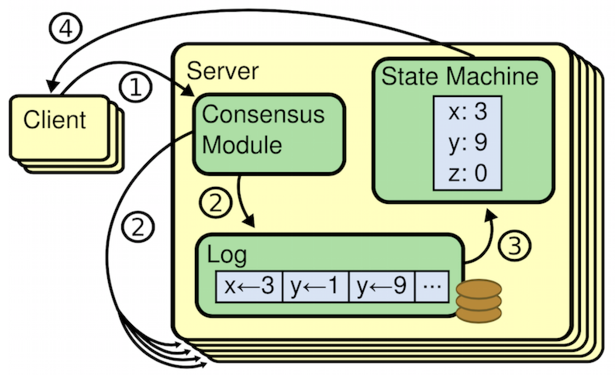
初始状态时，所有的server都是follower状态，随机睡眠一段时间，最先唤醒的server进入candidate状态，进入candidate之后会向其他所有的server发起request\_vote请求。如果follower还没有给其他candidate投票，那么follower会把仅有的一票投出，并保持follower的状态并重置选举计时器。当candidate收到过半（包括自己）的投票后就成为leader。leader向所有follower发送心跳包。follower收到心跳包后保持自己follower的状态并重置选举计时器。

自增当前的Term值，从follower转为candidate，设置为自己投票，并且发起为自己投票的请求，不断的重置直到：

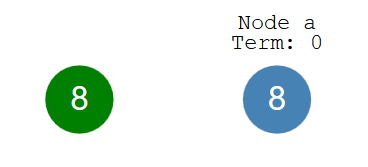
1. 获得超过半数服务器的投票（包括自己），称为leader，开始广播heartbeat；
2. 收到合法leader的heartbeat，变成follower；
3. 选举超时，在当前Term没有出现leader，自增当前的Term值，重新选择。

candidate既没有选举成功，也没有收到leader的heartbeat，比如两个server都是candidate，由于选举计时器同时结束，所以**两个candidate**同时发出为自己投片的请求。因为每个candidate都为自己投票了，接下来是不会给对方投票的，所以最后两个candidate都只收到一票。选举计时器的时间是随机值（150ms-300ms之间），下一次两个candidate可能就不会同时发起投票了，就能够解决这个问题。

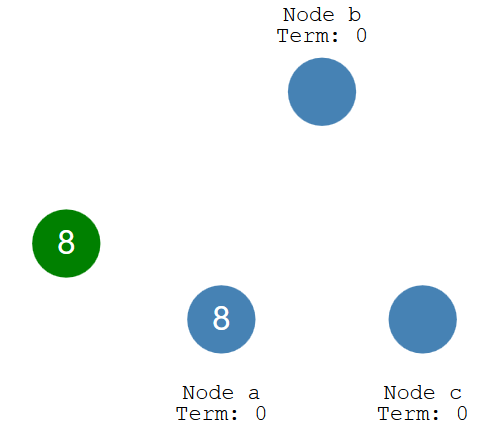
## Log replication



Raft是一种为了管理复制日志的一致性算法。上面的图就是raft的的工作过程，它保证拷贝到所有server上的每一条日志是一致的。



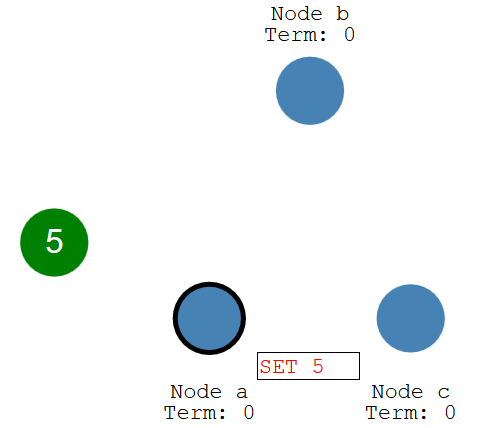
只有一个server的时候，client只要把值发送给server就可以保证一致。



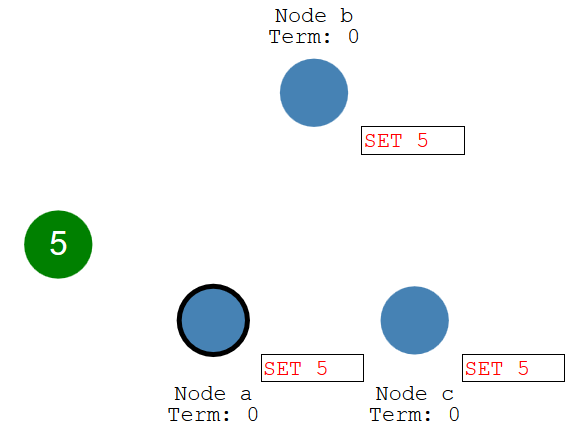
当有多个server的时候，如何保证client的一个值在多个服务器上也是一致的呢？

通过raft保证一致的过程：

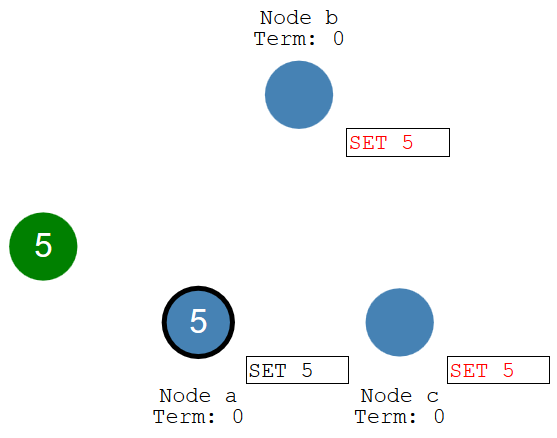
1. client发送command给leader，leader会在日志中记录下所收到的command，此时日志是uncommitted，所以不会修改真正的值。



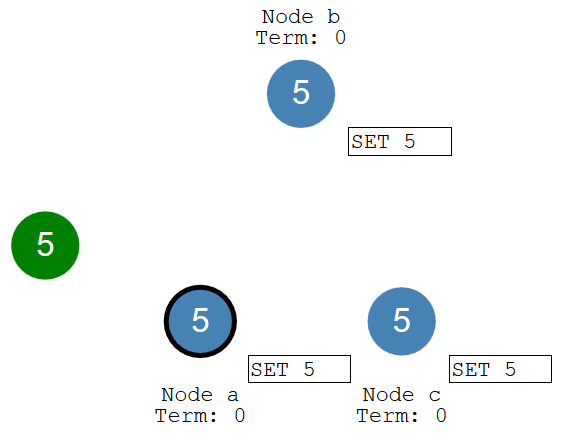
1. 接下来leader把日志备份到follower节点上



1. 所有的follower收到备份后告诉leader，leader会commit这次的操作并通知client。



1. leader commit注册操作后在通知follower也commit。



leader选举出来后，就可以开始处理客户端请求。leader收到客户端请求后，将请求内容作为一条log日志添加到自己的log记录中，并向其它server发送append\_entries(添加日志)请求。其它server收到append\_entries请求后，判断该append请求**满足接收条件**，如果满足条件就将其添加到本地的log中，并给Leader发送添加成功的response。Leader在收到大多数server添加成功的response后，就将该条log正式提交。提交后的log日志就意味着已经被raft系统接受，并能应用到状态机中了。

leader具有绝对的日志复制权力，其它server上存在日志不全或者与Leader日志不一致的情况时，一切都以Leader上的日志为主，最终所有server上的日志都会复制成与Leader一致的状态。

## Safety

整个过程如何保证正确性，raft保证一下属性在任何时刻为真：

1. 在任意指定的Term内，只有一个leader；
2. leader从不“重写”或者“删除”本地log，仅“追加”本地log；
3. 如果两个节点上的日志项拥有相同的Index和Term，那么这两个节点[0, Index]范围内的Log完全一致；
4. 如果某个日志项在某个Term被commit，那么后续任意Term的Leader均拥有该日志项；
5. 一旦某个server将某个日志项应用于本地状态机，以后所有server对于该偏移都将应用相同日志项。

### 同时出现两个candidate发起投票

每个server只能投一票，所以不可能有两个candidate同时获得过半的投票。如果两个candidate恰好各获得一般的票，两个都无法成为leader。两个candidate会随机睡眠一段时间，同时当前的Term值加1。由于睡眠时间是随机的，下一个选举发送冲突的可能性不大。

### 重新选举leader

Leader正常运作时，会周期性地发出append\_entries请求。这个周期性的append\_entries除了可以更新其它Follower的log信息，另外一个重要功能就是起到心跳作用。Follower收到append\_entries后，就知道Leader还活着。如果Follower经过一个预定的时间(一般设为2000ms左右)都没有收到Leader的心跳，就认为Leader挂了，再次选举新的leader。

每个新的leader都对应一个唯一的Term值。在candidate发起投票时，会把自己当前的Term值加1通过request\_vote发送给其他的server，follower只会接受Term值比自己大的request\_vote请求并为之投票。这条规则保证了只有最新的Candidate才有可能成为Leader。

### follower收到append\_entries后的处理

Follower在收到一条append\_entries后，首先会检查这条append\_entries的来源信息是否与本地保存的leader信息符合，包括leaderId和任期号term。检查合法后就将日志保存到本地log中，并给Leader回复添加log成功，但是不会立即将其应用到本地状态机。Leader收到大部分Follower添加log成功的回复后，就正式将这条日志commit提交。Leader在随后发出的心跳append\_entires中会带上已经提交日志索引。Follower收到Leader发出的心跳append\_entries后，就可以确认刚才的log已经被commit(提交)了，这个时候Follower才会把日志应用到本地状态机。下表即是append\_entries请求的内容，其中leaderCommit即是Leader已经确认提交的最大日志索引。Follower在收到Leader发出的append\_entries后即可以通过leaderCommit字段决定哪些日志可以应用到状态机。

### 宕机很久的server重新上线会不会成为leader？

如果宕机很久的server成为leader，这个leader的log应该是过期的，按照所有的日志由leader复制给follower的规则，那么follower上最新的日志会被删除，这样肯定是不对的。实际上，虽然携带旧日志的server可以竞选leader，但是raft规定只有具有最新日志的server才可以成为leader。candidate的request\_vote中还包括日志的信息，这些信息标明了它的日志新旧程度，比如lastLogTerm最后日志条目的Term值。其他server收到request\_vote后会比较lastLogTerm的值和自己的Term值，如果比自己的小就不会给投票。

# 参考

Paxos算法原理与推导

<https://www.cnblogs.com/linbingdong/p/6253479.html>

分布式一致性最佳实践分享

<https://zhuanlan.zhihu.com/paxos>

理解这两点，也就理解了paxos协议的精髓

<https://zhuanlan.zhihu.com/p/29706905>

使用Paxos前的八大问题

<https://zhuanlan.zhihu.com/p/23811020>

The Raft Consensus Algorithm（raft文章以及实现版本）

<https://raft.github.io/>

raft动画展示

<http://thesecretlivesofdata.com/raft/>

raft

<https://www.jianshu.com/p/4711c4c32aab>

<http://blog.csdn.net/cszhouwei/article/details/38374603>

raft论文中文

<http://www.infoq.com/cn/articles/raft-paper>