# 背景

为了解决节点损坏的问题，业界通用的方案是采用Replication，在Master写数据的时候挂多个Slave，写完后保证将log同步到Slave，这样的流程下来才能认为写入是成功的。

通常情况下为了保证性能，我们都会使用异步复制（Async Replication），不过会产生一些问题。当Master挂掉了之后很有可能有些数据并没有同步到Slave，这时候如果把这个Slave提升到新的Master就会面临数据丢失的问题。

强同步（Sync Replication）解决了异步复制（Async Replication）可能潜在的问题。Master写入数据后必须保证Slave接收到log写入才算成功，当然这样会损耗一些性能，不过最严重在于Master挂掉后，Slave提升为Master，单节点的问题就又出现了。

为了保证系统中N台节点挂掉后，仍然能够对外提供服务，至少需要2N+1台机器，也就是传统的Quorum，这样最少就需要三台机器，但是数量的增大也会带来更多的问题。

首先Master挂掉后要需要判断该提升哪个Slave为新的Master。另外在多态集群下还容易遇到一个问题，就是在网络情况下以前的Master出现网络隔离，还在继续对外提供服务，而这时新的集群和Master已经形成了，新的数据被写入到新的Master，可是读取的却是旧的数据，这就是在分布式一致性领域所面临的线性一致性问题。

分布式一致性算法最著名的应该是Paxos，1990年提出，google的Chubby Lock服务就是使用的Paxos。之后的一些一致性算法基本都是在Paxos思路上的调整，例如ZooKeeper的ZAB。但Paxos算法一直被认为比较繁杂，很不好理解，大家对其调整优化，就是因为他的复杂。

2013年，斯坦福的两个人以易懂为目标，设计了一致性算法Raft，现在已经被广泛应用，比较有名的是etcd，Google的Kubernetes就使用了etcd作为他的服务发现框架。

严格来说Raft并不属于Paxos的一个变种。Raft协议并不是对Paxos的改进，也没有使用Paxos的基础协议（The Basic Protocol）。Raft协议在设计理念上和Paxos协议是完全相反的。正是由于这个完全不同的理念，使得Raft协议变得简单起来。

Paxos协议中有一个基本的假设前提：可能会同时有多个Leader存在。这里把Paxos协议执行的过程分为以下两个部分：

1. Leader选举
2. 数据广播

**Paxos对于Leader的选举没有限制，用户可以自己定义**。这是因为Paxos协议设计了一个巧妙的数据广播过程，即Paxos的基本通讯协议（The Basic Protocol）。它有很强的数据一致性保障，即使在多个Leader同时出现时也能够保证广播数据的一致性。

而**Raft协议走了完全相反的一个思路：保证不会同时有多个Leader存在（采用避免冲突的方式，Paxos则不会避免冲突，使用失败重试）**。因此Raft协议对Leader的选举做了详细的设计，从而保证不会有多个Leader同时存在。相反，数据广播的过程则变的简单易于理解了。

**分布式一致：**

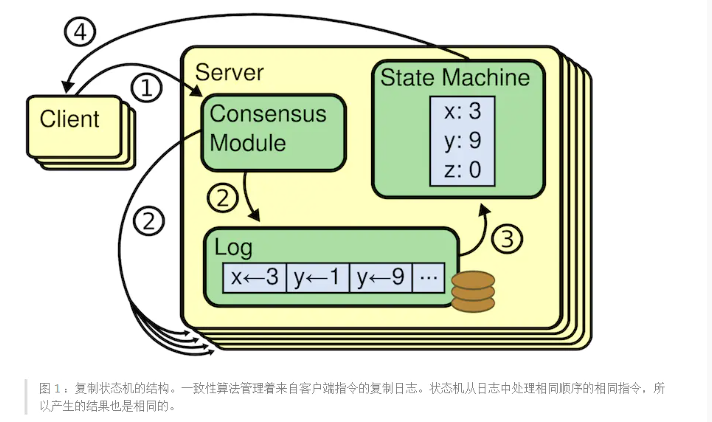
在单节点环境中，client向node发送一个值，很容易就达成一致了



但当我们有多个node时，我们应该如何做，才能实现一致性呢？

这就是分布式一致性问题，Raft就是用来解决此问题的。

# 复制状态机



大多数的一致性算法其实都是采用了复制状态机（Replicated State Machine）的模型。对于这个模型你可以认为数据是被放在状态机上，用户操作集群时首先需要写入log，然后通过一致性算法将log复制到其他机器，一旦确定其他机器都接收到log后就会认为该log被提交了，最后其他节点会依次将log存放到状态机。

## State

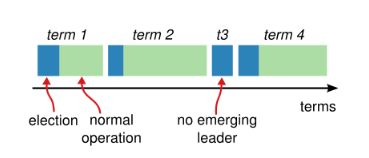
Raft有着三个状态，第一个状态就是Leader，即使Raft Group存在多个节点，Leader也只会存在一个，也只有Leader能负责所有数据的读写，这样就不会出现线性一致性的问题。

当一个集群选出了Leader后其他集群的节点都会变成Follow，这个Follow只负责一件事就是接收Leader的Replication logs。当一个Follow没有接收到数据或者发现集群中没有Leader的时候，就会进入Candidate状态，这个状态表示可以开始选举了。

## Term

Raft是不依赖系统的时间，它是将时间分成一个个Term，每个Term可以是任意长度，每次Term开始的时候都是由一次新的选举产生的，然后在这个Term内选出一个Leader，该Leader会一直服务到所在Leader结束。

结合Raft Term就可以进行Raft的选举。首先当系统启动后所有的节点都会进入Follow状态，Follow没有接收到任何请求的话，过一段时间就会进入Candidate状态，然后询问其他节点是否投票，如果其他节点反馈已经有新的Leader，这个Candidate就会变成Follow，而当Candidate接收到大多数的投票后就会变成Leader，之后会立即将一个log发送给其他节点开始续租Leader的时间租约。



每个Term都有一个唯一的数字编号。所有Term的数字编号是从小到大连续排列的。

**作废旧Leader**

Term编号在作废旧Leader的过程中至关重要，但却十分简单。过程如下：

1. 发送日志到所有Followers，Leader的Term编号随日志一起发送。
2. Followers收到日志后，检查Leader的Term编号。如果Leader的Term编号等于或者大于自己的当前Term（Current Term）编号，则存储日志到队列并且应答收到日志。否则发送失败消息给Leader，消息中包含自己的当前Term编号。
3. 当Leader收到任何Term编号比自己的Term编号大的消息时，则将自己变成Follower。收到的消息包括：Follower给自己的回复消息、新Leader的日志广播消息、Leader的选举消息。

## Log Replication

一个新的Leader被选出来之后就会开始工作了，它会不停的去接收客户端发送过来的请求，这些请求都会通过log落地，而这些log一定要是单调递增，以保证数据的一致性。

之后log会被复制到其他的节点，绝大多数节点都接收到这个log后， Leader就认为该log是committed的。

## Membership Change

对于分布式集群来说添加节点其实是分成困难的操作，最常见的做法是先更改配置中心，然后将新的配置同步到旧的节点。不过这样在同步配置的时候，就需要停止外部服务。而Raft采用了一种动态的成员节点变更，它会将新的节点到当作Raft log通过Leader传递给其他节点，这样其他节点就知道了这个新的节点的信息。不过这个过程中有可能会在某一阶段出现2个Leader的情况，为了避免这种情况就要每次只变更一个节点，而不进行多节点变更。

Raft也提供了一种多节点变更的算法，它是一种两阶段提交，Leader在第一阶段会同时将新旧集群的配置同时当成一个Raft log发送给其他旧集群节点，当这些节点接收到数据后就会和新的集群节点进入join状态，所有的操作都要进过新旧集群的大多数节点同意才会执行，然后在新的join状态内重新提交新的配置信息，在配置被committed后新的节点会上线，旧的节点会下线。

# 原理

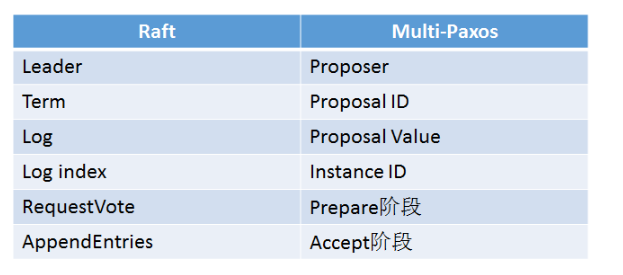
## Paxos VS Raft

Paxos和Raft最大的区别在于Paxos支持乱序同步，Raft只支持顺序同步。

Raft相当于是Paxos协议的一种简化，好处是实现简单，容易理解，坏处是损失了日志并行同步的性能。国内和开源界流行Raft，不过对于AWS、微软、Google等大公司，关键系统还是使用Paxos的。

### 相同

**Raft与Multi-Paxos中相似的概念：**



Raft的Leader即Multi-Paxos的Proposer。

Raft的Term与Multi-Paxos的Proposal ID本质上是同一个东西。

Raft的Log Entry即Multi-Paxos的Proposal。

Raft的Log Index即Multi-Paxos的Instance ID。

Raft的Leader选举跟Multi-Paxos的Prepare阶段本质上是相同的。

Raft的日志复制即Multi-Paxos的Accept阶段。

### 不同

**Raft与Multi-Paxos的不同：**



Raft假设系统在任意时刻最多只有一个Leader，提议只能由Leader发出（强Leader），否则会影响正确性；而Multi-Paxos虽然也选举Leader，但只是为了提高效率，并不限制提议只能由Leader发出（弱Leader）。

强Leader在工程中一般使用Leader Lease和Leader Stickiness来保证：

Leader Lease：上一任Leader的Lease过期后，随机等待一段时间再发起Leader选举，保证新旧Leader的Lease不重叠。

Leader Stickiness：Leader Lease未过期的Follower拒绝新的Leader选举请求。

Raft限制具有最新已提交的日志的节点才有资格成为Leader，Multi-Paxos无此限制。

Raft在确认一条日志之前会检查日志连续性，若检查到日志不连续会拒绝此日志，保证日志连续性，Multi-Paxos不做此检查，允许日志中有空洞。

Raft在AppendEntries中携带Leader的commit index，一旦日志形成多数派，Leader更新本地的commit index即完成提交，下一条AppendEntries会携带新的commit index通知其它节点；Multi-Paxos没有日志连接性假设，需要额外的commit消息通知其它节点。

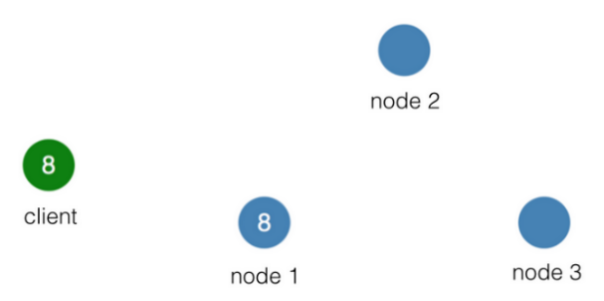
## 子问题

Paxos难以理解，raft应运而生，可以理解为简单的Paxos，它划分为三个子问题：

1. Leader Election
2. Log Replication
3. Safety

## 思路

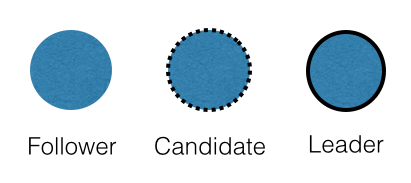
每个node都会处于以下3个状态之一：



（1）Follower 跟随者

（2）Candidate 候选人

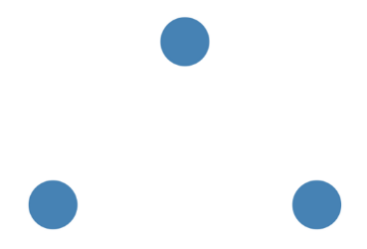
（3）Leader 领导人



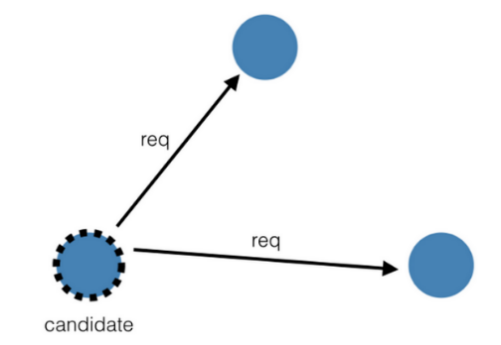
所有node开始时都是follower：

当follower没有收到leader的心跳时，他就会申请成为candidate，然后向其他node发送请求，说“我要成为Leader，请给我投票”

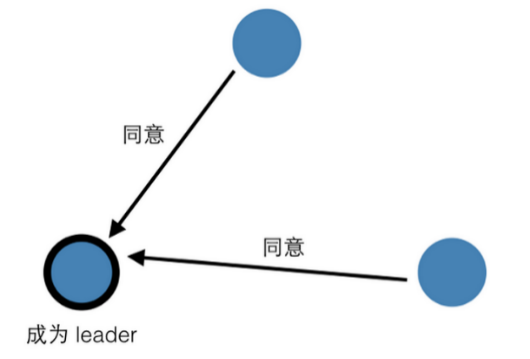
当candidate收到大多数node的同意后，就变为了Leader，以后对于系统的修改操作，都必须经过Leader

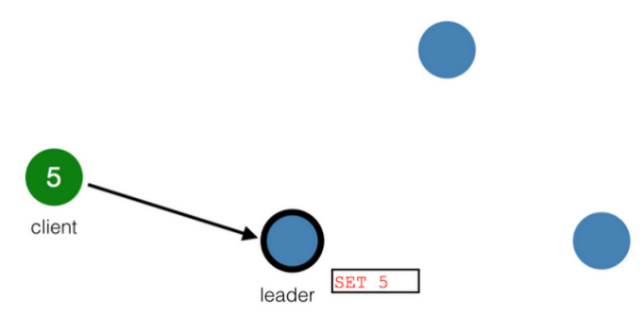


例如client要发送消息，会先发给leader，leader会把这个操作记录到自己的日志

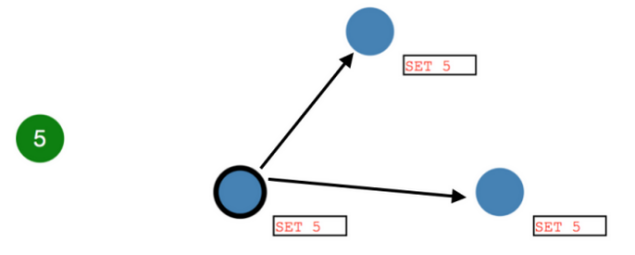


注意，是记录到日志，并没有实际修改node中的值

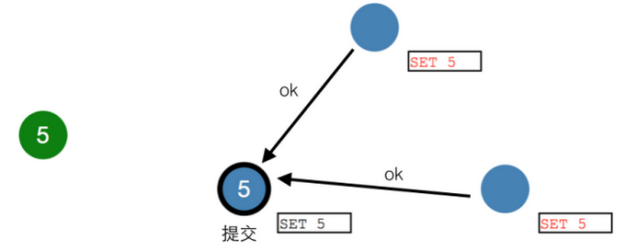




leader把这条操作记录发送给各个follower，follower收到后，也保存到自己的日志中

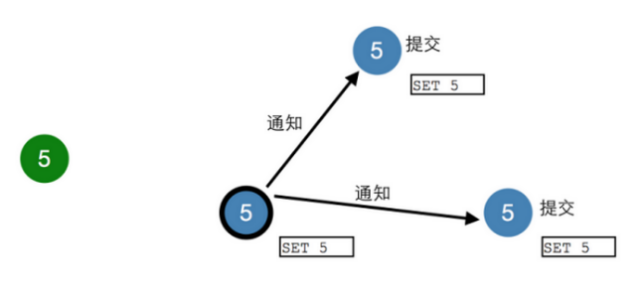


follower收到操作记录后，向leader发送消息，说自己安排好了



leader收到大多数的回馈后，就把这条记录进行提交，真正修改了node中的值

leader执行提交以后，就通知各个follower，“我已经提交了，你们可以更新了”



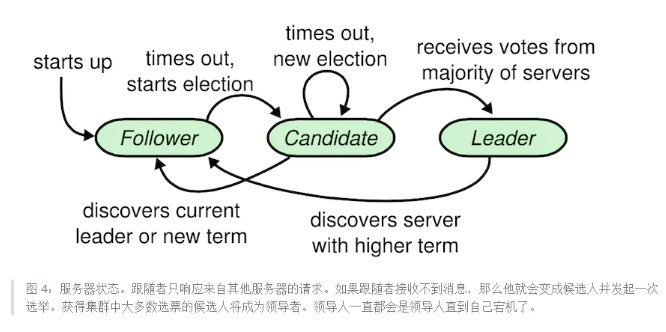
现在，系统就达成了一致的状态

这个过程叫做 Log Replication 日志复制，是Raft的核心之一，还有选举leader过程也是核心。

## 基本规则

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。

raft中的server有三种状态，除了已经提到的Leader和Follower状态外，还有Candidate状态，即竞选者状态。下面是这三种状态的转化过程。



### Leader选举过程

raft初始状态时所有server都处于Follower状态，并且随机睡眠一段时间，这个时间在0~1000ms之间。最先醒来的server A进入Candidate状态，Candidate状态的server A有权利发起投票，向其它所有server发出requst\_vote请求，请求其它server给它投票成为Leader。当其它server收到request\_vote请求后，将自己仅有的一票投给server A，同时继续保持Follower状态并重置选举计时器。当server A收到大多数（超过一半以上）server的投票后，就进入Leader状态，成为系统中仅有的Leader。raft系统中只有Leader才有权利接收并处理client请求，并向其它server发出添加日志请求来提交日志。

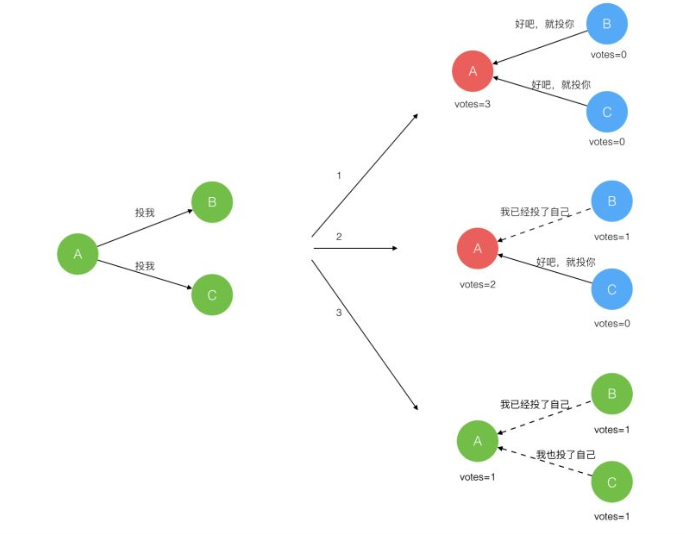
Raft 使用心跳（heartbeat）触发Leader选举。当服务器启动时，初始化为Follower。Leader向所有Followers周期性发送heartbeat。如果Follower在选举超时时间内没有收到Leader的heartbeat，就会等待一段随机的时间后发起一次Leader选举。

Follower将其当前term加一然后转换为Candidate。它首先给自己投票并且给集群中的其他服务器发送 RequestVote RPC。结果有以下三种情况：

1、赢得了多数的选票，成功选举为Leader；

2、收到了Leader的消息，表示有其它服务器已经抢先当选了Leader；

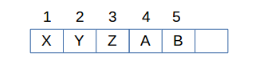
3、没有服务器赢得多数的选票，Leader选举失败，等待选举时间超时后发起下一次选举。



选举出Leader后，Leader通过定期向所有Followers发送心跳信息维持其统治。若Follower一段时间未收到Leader的心跳则认为Leader可能已经挂了，再次发起Leader选举过程。

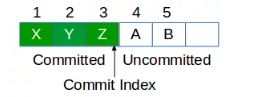
### 日志复制过程

Raft将用户数据称作日志（Log），存储在一个日志队列里。每个节点上都有一份。队列里的每个日志都一个序号，这个序号是连续递增的不能有缺。

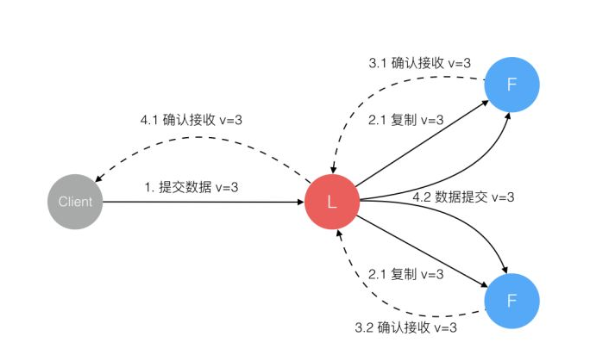


日志队列里有一个重要的位置叫做提交日志的位置（Commit Index）。将日志队列里的日志分为了两个部分：

1. 已提交日志：已经复制到超过半数节点的数据。这些日志是可以发送给应用程序去执行的日志。
2. 未提交日志：还未复制到超过半数节点的数据。

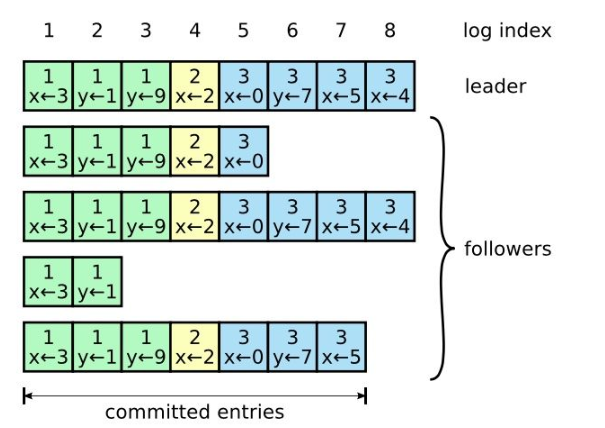


当Followers收到日志后，将日志按顺序存储到队列里。但这时Commit Index不会更新，因此这些日志是未提交的日志，不能发送给应用去执行。当Leader收到超过半数的Followers的应答后，会更新自己的Commit Index，并将Commit Index广播到Followers上。这时Followers更新Commit Index，未提交的日志就变成了已提交的日志，可以发送给应用程序去执行了。



某些Followers可能没有成功的复制日志，Leader会无限的重试 AppendEntries RPC直到所有的Followers最终存储了所有的日志条目。

日志由有序编号（log index）的日志条目组成。每个日志条目包含它被创建时的任期号（term），和用于状态机执行的命令。如果一个日志条目被复制到大多数服务器上，就被认为可以提交（commit）了。



Raft日志

**Raft日志同步保证如下两点：**

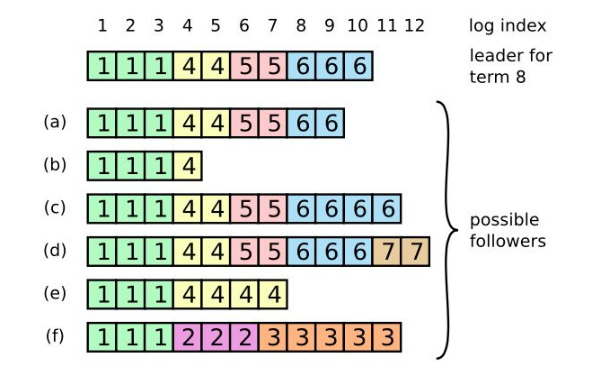
1、如果不同日志中的两个条目有着相同的索引和任期号，则它们所存储的命令是相同的。

2、如果不同日志中的两个条目有着相同的索引和任期号，则它们之前的所有条目都是完全一样的。

第一条特性源于Leader在一个term内在给定的一个log index最多创建一条日志条目，同时该条目在日志中的位置也从来不会改变。

第二条特性源于AppendEntries的一个简单的一致性检查。当发送一个AppendEntriesRPC时，Leader会把新日志条目紧接着之前的条目的log index和term都包含在里面。如果Follower没有在它的日志中找到log index和term都相同的日志，它就会拒绝新的日志条目。

一般情况下，Leader和Followers的日志保持一致，因此AppendEntries一致性检查通常不会失败。然而，Leader崩溃可能会导致日志不一致：旧的Leader可能没有完全复制完日志中的所有条目。



Leader和Followers上日志不一致

上图阐述了一些Followers可能和新的Leader日志不同的情况。一个Follower可能会丢失掉Leader上的一些条目，也有可能包含一些Leader没有的条目，也有可能两者都会发生。丢失的或者多出来的条目可能会持续多个任期。

Leader通过强制Followers复制它的日志来处理日志的不一致，Followers上的不一致的日志会被Leader的日志覆盖。

Leader为了使Followers的日志同自己的一致，Leader需要找到Followers同它的日志一致的地方，然后覆盖Followers在该位置之后的条目。

Leader会从后往前试，每次AppendEntries失败后尝试前一个日志条目，直到成功找到每个Follower的日志一致位点，然后向后逐条覆盖Followers在该位置之后的条目。

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（接收条件在后面安全保证问题3给出），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。

以上就是raft允许的基本规则，如果不出现任何异常情况，那么只要上面两个过程就能使raft运行起来了。但是现实的系统不可能这么一帆风顺，总是有很多异常情况需要考虑。raft的复杂性就来源于对这些异常情况的考虑。

### 安全性保证

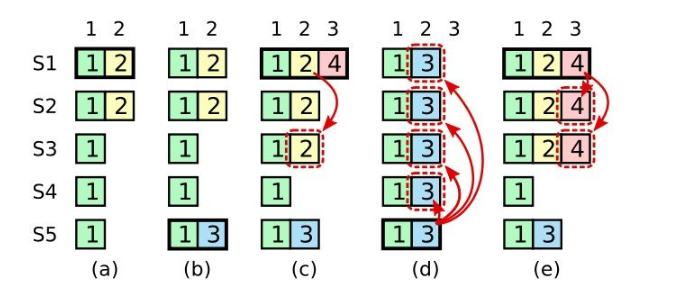
**Raft增加了如下两条限制以保证安全性：**

1、拥有最新的已提交的log entry的Follower才有资格成为Leader。

这个保证是在RequestVote RPC中做的，Candidate在发送RequestVote RPC时，要带上自己的最后一条日志的term和log index，其他节点收到消息时，如果发现自己的日志比请求中携带的更新，则拒绝投票。日志比较的原则是，如果本地的最后一条log entry的term更大，则term大的更新，如果term一样大，则log index更大的更新。

2、Leader只能推进commit index来提交当前term的已经复制到大多数服务器上的日志，旧term日志的提交要等到提交当前term的日志来间接提交（log index小于commit index的日志被间接提交）。

之所以要这样，是因为可能会出现已提交的日志又被覆盖的情况：



已提交的日志被覆盖

在阶段a，term为2，S1是Leader，且S1写入日志（term, index）为(2, 2)，并且日志被同步写入了S2；

在阶段b，S1离线，触发一次新的选主，此时S5被选为新的Leader，此时系统term为3，且写入了日志（term, index）为（3， 2）;

S5尚未将日志推送到Followers就离线了，进而触发了一次新的选主，而之前离线的S1经过重新上线后被选中变成Leader，此时系统term为4，此时S1会将自己的日志同步到Followers，按照上图就是将日志（2， 2）同步到了S3，而此时由于该日志已经被同步到了多数节点（S1, S2, S3），因此，此时日志（2，2）可以被提交了。；

在阶段d，S1又下线了，触发一次选主，而S5有可能被选为新的Leader（这是因为S5可以满足作为主的一切条件：1. term = 5 > 4，2. 最新的日志为（3，2），比大多数节点（如S2/S3/S4的日志都新），然后S5会将自己的日志更新到Followers，于是S2、S3中已经被提交的日志（2，2）被截断了。

增加上述限制后，即使日志（2，2）已经被大多数节点（S1、S2、S3）确认了，但是它不能被提交，因为它是来自之前term（2）的日志，直到S1在当前term（4）产生的日志（4， 4）被大多数Followers确认，S1方可提交日志（4，4）这条日志，当然，根据Raft定义，（4，4）之前的所有日志也会被提交。此时即使S1再下线，重新选主时S5不可能成为Leader，因为它没有包含大多数节点已经拥有的日志（4，4）。

### 日志压缩

在实际的系统中，不能让日志无限增长，否则系统重启时需要花很长的时间进行回放，从而影响可用性。Raft采用对整个系统进行snapshot来解决，snapshot之前的日志都可以丢弃。

每个副本独立的对自己的系统状态进行snapshot，并且只能对已经提交的日志记录进行snapshot。

Snapshot中包含以下内容：

1、日志元数据。最后一条已提交的 log entry的 log index和term。这两个值在snapshot之后的第一条log entry的AppendEntries RPC的完整性检查的时候会被用上。

2、系统当前状态。

当Leader要发给某个日志落后太多的Follower的log entry被丢弃，Leader会将snapshot发给Follower。或者当新加进一台机器时，也会发送snapshot给它。发送snapshot使用InstalledSnapshot RPC（RPC细节参见八、Raft算法总结）。

做snapshot既不要做的太频繁，否则消耗磁盘带宽， 也不要做的太不频繁，否则一旦节点重启需要回放大量日志，影响可用性。推荐当日志达到某个固定的大小做一次snapshot。

做一次snapshot可能耗时过长，会影响正常日志同步。可以通过使用copy-on-write技术避免snapshot过程影响正常日志同步。

### 成员变更

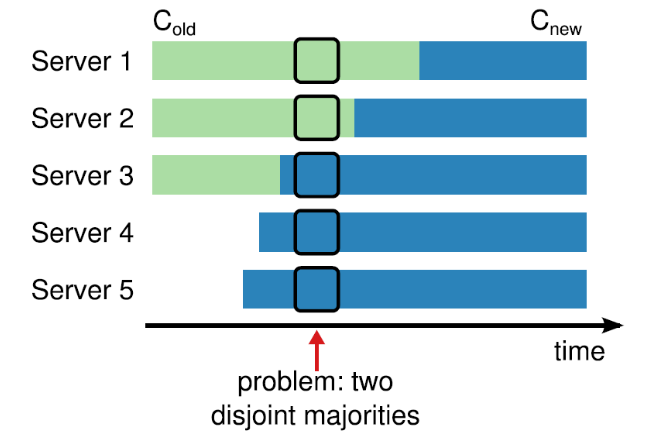
成员变更是在集群运行过程中副本发生变化，如增加/减少副本数、节点替换等。

成员变更也是一个分布式一致性问题，即所有服务器对新成员达成一致。但是成员变更又有其特殊性，因为在成员变更的一致性达成的过程中，参与投票的进程会发生变化。

如果将成员变更当成一般的一致性问题，直接向Leader发送成员变更请求，Leader复制成员变更日志，达成多数派之后提交，各服务器提交成员变更日志后从旧成员配置（Cold）切换到新成员配置（Cnew）。

因为各个服务器提交成员变更日志的时刻可能不同，造成各个服务器从旧成员配置（Cold）切换到新成员配置（Cnew）的时刻不同。

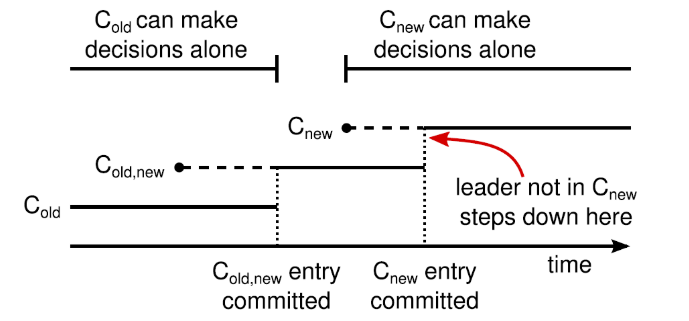
成员变更不能影响服务的可用性，但是成员变更过程的某一时刻，可能出现在Cold和Cnew中同时存在两个不相交的多数派，进而可能选出两个Leader，形成不同的决议，破坏安全性。



成员变更的某一时刻Cold和Cnew中同时存在两个不相交的多数派

由于成员变更的这一特殊性，成员变更不能当成一般的一致性问题去解决。

为了解决这一问题，Raft提出了两阶段的成员变更方法。集群先从旧成员配置Cold切换到一个**过渡成员配置**，称为**共同一致（joint consensus）**，共同一致是旧成员配置Cold和新成员配置Cnew的组合Cold U Cnew，**一旦共同一致Cold U Cnew被提交，系统再切换到新成员配置Cnew**。



Raft两阶段成员变更

**Raft两阶段成员变更过程如下：**

1. Leader收到成员变更请求从Cold切成Cnew；
2. Leader在本地生成一个新的log entry，其内容是Cold∪Cnew，代表当前时刻新旧成员配置共存，写入本地日志，同时将该log entry复制至Cold∪Cnew中的所有副本。在此之后新的日志同步需要保证得到Cold和Cnew两个多数派的确认；
3. Follower收到Cold∪Cnew的log entry后更新本地日志，并且此时就以该配置作为自己的成员配置；
4. 如果Cold和Cnew中的两个多数派确认了Cold U Cnew这条日志，Leader就提交这条log entry；
5. 接下来Leader生成一条新的log entry，其内容是新成员配置Cnew，同样将该log entry写入本地日志，同时复制到Follower上；
6. Follower收到新成员配置Cnew后，将其写入日志，并且从此刻起，就以该配置作为自己的成员配置，并且如果发现自己不在Cnew这个成员配置中会自动退出；
7. Leader收到Cnew的多数派确认后，表示成员变更成功，后续的日志只要得到Cnew多数派确认即可。Leader给客户端回复成员变更执行成功。

**异常分析：**

1. 如果Leader的Cold U Cnew尚未推送到Follower，Leader就挂了，此后选出的新Leader并不包含这条日志，此时新Leader依然使用Cold作为自己的成员配置。
2. 如果Leader的Cold U Cnew推送到大部分的Follower后就挂了，此后选出的新Leader可能是Cold也可能是Cnew中的某个Follower。
3. 如果Leader在推送Cnew配置的过程中挂了，那么同样，新选出来的Leader可能是Cold也可能是Cnew中的某一个，此后客户端继续执行一次改变配置的命令即可。
4. 如果大多数的Follower确认了Cnew这个消息后，那么接下来即使Leader挂了，新选出来的Leader肯定位于Cnew中。

两阶段成员变更比较通用且容易理解，但是实现比较复杂，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的服务可用性，因此我们期望增强成员变更的限制，以简化操作流程。

两阶段成员变更，之所以分为两个阶段，是因为对Cold与Cnew的关系没有做任何假设，为了避免Cold和Cnew各自形成不相交的多数派选出两个Leader，才引入了两阶段方案。

如果增强成员变更的限制，假设Cold与Cnew任意的多数派交集不为空，这两个成员配置就无法各自形成多数派，那么成员变更方案就可能简化为一阶段。

那么如何限制Cold与Cnew，使之任意的多数派交集不为空呢？方法就是每次成员变更只允许增加或删除一个成员。

可从数学上严格证明，只要每次只允许增加或删除一个成员，Cold与Cnew不可能形成两个不相交的多数派。

**一阶段成员变更：**

1、成员变更限制每次只能增加或删除一个成员（如果要变更多个成员，连续变更多次）。

2、成员变更由Leader发起，Cnew得到多数派确认后，返回客户端成员变更成功。

3、一次成员变更成功前不允许开始下一次成员变更，因此新任Leader在开始提供服务前要将自己本地保存的最新成员配置重新投票形成多数派确认。

Leader只要开始同步新成员配置，即可开始使用新的成员配置进行日志同步。

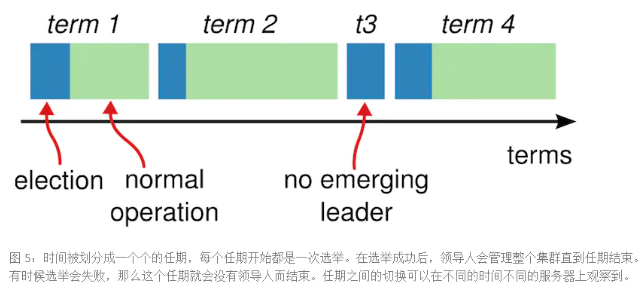
### QA

**1、Leader选举过程中，如果有两个serverA和B同时醒来并发出request\_vote请求怎么办？**

由于在一次选举过程中，一个server最多只能投一票，这就保证了serverA和B不可能同时得到大多数（一半以上）的投票。如果A或者B中其一幸运地得到了大多数投票，就能顺利地成为Leader，raft系统正常运行下去。但是A和B可能刚好都得到一半的投票，两者都成为不了Leader。这时A和B继续保持Candidate状态，并且随机睡眠一段时间，等待进入到下一个选举周期。由于所有server都是随机选择睡眠时间，所以连续出现多个server竞选的概率很低。

**2、Leader挂了后，如何选举出新的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。于是转入Candidate状态，开始发起投票竞选新的Leader。每个新的Leader产生后就是一个新的任期，每个任期都对应一个唯一的任期号term。这个term是单调递增的，用来唯一标识一个Leader的任期。投票开始时，Candidate将自己的term加1，并在request\_vote中带上term；Follower只会接受任期号term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。



**3、Follower在收到一条append\_entries添加日志请求后，是否立即保存并将其应用到状态机中去？如果不是立即应用，那么由什么来决定该条日志生效的时间？**

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。

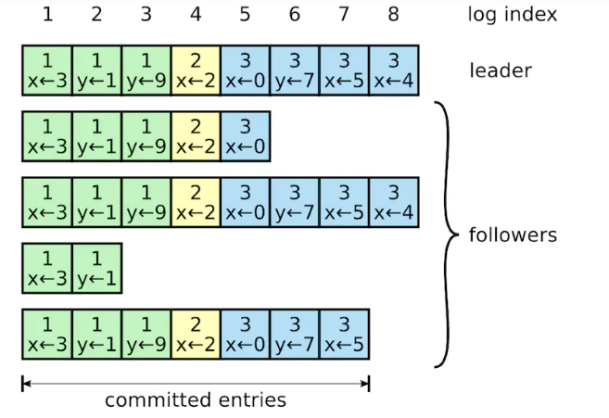


**4、假设有一个server A宕机了很长一段时间，它的日志已经落后很多。如果A重新上线，而且此时现有Leader挂掉，server A刚好竞选成为了Leader。按照日志都是由Leader复制给其它server的原则，server A会把其它Follower已经提交的日志给抹掉，而这违反了raft状态机安全特性，raft怎么解决这种异常情况？**

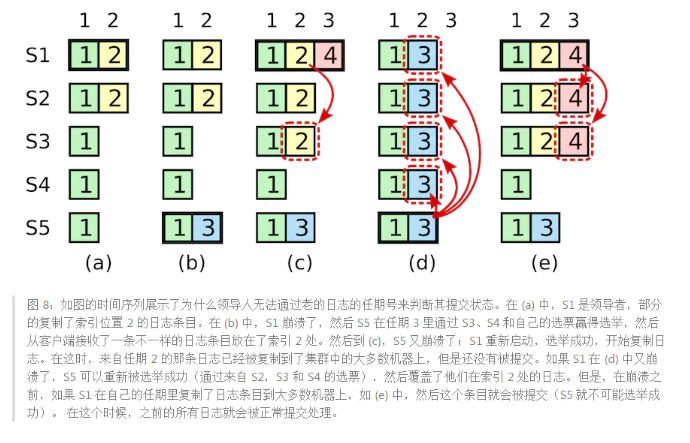
所谓的状态机安全特性即是“如果一个领导人已经在给定的索引值位置的日志条目应用到状态机中，那么其他任何的服务器在这个索引位置不会提交一个不同的日志”。如果server在竞选Leader的过程中不加任何限制的话，携带旧日志的server也有可能竞选成为Leader，就必然存在覆盖之前Leader已经提交的日志可能性，从而违反状态机安全特性。raft的解决办法很简单，就是只有具有最新日志的server的才有资格去竞选当上Leader，具体是怎么做到的呢？首先任何server都还是有资格去发起request\_vote请求去拉取投票的，request\_vote中会带上server的日志信息，这些信息标明了server日志的新旧程度，如下表所示。



其它server收到request\_vote后，判断如果lastLogTerm比自己的term大，那么就可以给它投票；lastLogTerm比自己的term小，就不给它投票。如果相等的话就比较lastLogIndex，lastLogIndex大的话日志就比较新，就给它投票。下图是raft日志格式，每条日志中不仅保存了日志内容，还保存了发送这条日志的Leader的任期号term。为什么要在日志里保存任期号term，由于任期号是全局单调递增且唯一的，所以根据任期号可以判断一条日志的新旧程度，为选举出具有最新日志的Leader提供依据。



**5、存在如下图一种异常情况，server S5在时序(d)中覆盖了server S1在时序(c)中提交的index为2的日志，方框中的数字是日志的term。这违反了状态机的安全特性--“如果一个领导人已经在给定的索引值位置的日志条目应用到状态机中，那么其他任何的服务器在这个索引位置不会提交一个不同的日志”，raft要如何解决这个问题？**

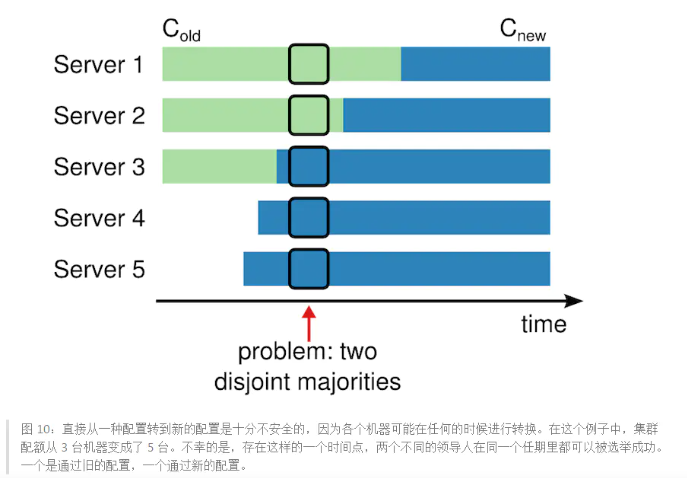


出现这个问题的根本原因是S1在时序(c) 的任期4内提交了一个之前任期2的log，这样S1提交的日志中最大的term仅仅是2，那么一些日志比较旧的server，比如S5(它最日志的term为 3)，就有机会成为leader，并覆盖S1提交的日志。解决办法就是S1在时序(c)的任期term4提交term2的旧日志时，旧日志必须附带在当前term 4的日志下一起提交。这样就把S1日志的最大term提高到了4，让那些日志比较旧的S5没有机会竞选成为Leader，也就不会用旧的日志覆盖已经提交的日志了。

简单点说，Leader如果要提交之前term的旧日志，那么必须要提交一条当前term的日志。提交一条当前term的日志相当于为那些旧的日志加了一把安全锁，让那些日志比较旧的server失去得到Leader的机会，从而不会修改那些之前term的旧日志。

怎么具体实现旧日志必须附带在当前term的日志下一起提交呢？在问题3中有给出append\_entries请求中的字段，其中有两个字段preLogIndex和preLogTerm的作用没有提到，这两个字段就是为了保证Leader和Followers的历史日志完全一致而存在的。当Leader在提交一条新日志的时候，会带上新日志前一条日志的index和term，即preLogIndex和preLogTerm。Follower收到append\_entries后，会检查preLogIndex和preLogTerm是否和自己当前最新那条日志的index和term对得上，如果对不上就会给Leader返回自己当前日志的index和term。Leader收到后就将Follower返回的index对应的日志以及对应的preLogIndex和preLogTerm发送给Follower。这个过程一直重复，直到Leader和Follower找到了第一个index和term能对得上的日志，然后Leader从这条日志开始拷贝给Follower。回答段首的问题，Leader在提交一条最新的日志时，Follow会检验之前的日志是否与Leader保持了一致，如果不一致会一直同步到与Leader一致后才添加最新的日志，这个机制就保证了Leader在提交最新日志时，也提交了之前旧的日志。

**6、向raft系统中添加新机器时，由于配置信息不可能在各个系统上同时达到同步状态，总会有某些server先得到新机器的信息，有些server后得到新机器的信息。比如下图raft系统中新增加了server4和server5这两台机器。只有server3率先感知到了这两台机器的添加。这个时候如果进行选举，就有可能出现两个Leader选举成功。因为server3认为有3台server给它投了票，它就是Leader，而server1认为只要有2台server给它投票就是Leader了。raft怎么解决这个问题呢？**



产生这个问题的根本原因是，raft系统中有一部分机器使用了旧的配置，如server1和server2，有一部分使用新的配置，如server3。解决这个问题的方法是添加一个中间配置(Cold, Cnew)，这个中间配置的内容是旧的配置表Cold和新的配置Cnew。还是拿上图中的例子来说明，这个时候server3收到添加机器的消息后，不是直接使用新的配置Cnew，而是使用(Cold, Cnew)来做决策。比如说server3在竞选Leader的时候，不仅需要得到Cold中的大部分投票，还要得到Cnew中的大部分投票才能成为Leader。这样就保证了server1和server2在使用Cold配置的情况下，还是只可能产生一个Leader。当所有server都获得了添加机器的消息后，再统一切换到Cnew。raft实现中，将Cold，(Cold,Cnew)以及Cnew都当成一条普通的日志。配置更改信息发送Leader后，由Leader先添加一条 (Cold, Cnew)日志，并同步给其它Follower。当这条日志(Cold, Cnew)提交后，再添加一条Cnew日志同步给其它Follower，通过Cnew日志将所有Follower的配置切换到最新。

有的raft实现采用了一种更加简单粗暴的方法来解决成员变化的问题。这个办法就是每次只更新一台机器的配置变化，收到配置变化的机器立马采用新的配置。这样的做法为什么能确保安全性呢？下面举例说明。比如说系统中原来有5台机器A,B,C,D,E，现在新加了一台机器F，A,B,C三台机器没有感知到F的加入，只有D,E两台机器感知到了F的加入。现在就有了两个旧机器集合X｛A, B, C, D, E}和新机器集合Y｛F｝。假设A和D同时进入Candidate状态去竞选Leader，其中D要想竞选成功，必须得有一半以上机器投票，即6/2+1=4台机器，就算Y集合中的F机器给D投了票，还得至少在集合X中得到3票；而A要想竞选成功，也必须得到5/2+1 = 3张票，由于A只知道集合X的存在，所以也必须从集合X中获得至少3票。而A和D不可能同时从集合X同时获得3票，所以A和D不可能同时竞选成为Leader，从而保证了安全性。可以使用更加形式化的数学公式来证明一次添加一台机器配置不会导致产生两个Leader，证明过程就暂时省略了。

# 优化

## Pre-Vote

在Follow处于网络抖动无法接受到Leader的消息的时候，它就会变成Candidate并且Term加一，但是其他集群其实还是在正常工作的，这样整个集群的就混乱了。

Pre-Vote机制会在Follow没有接收到Leader的消息并且开始投票之前进入Pre-Candidate状态，在想其他节点发送投票请求，并获得同意后才会进入Candidate状态。

## Pipeline

正常的Raft流程中，客户端事先给Leader写入数据，Leader处理后会追加到log中，追加成功后Replication到其他节点中，当Leader发现log被整个集群大多数节点接收后就会进行Apply。

这样的一个过程其实是非常低效的，所以就需要引入Pipeline，它可以将多个请求进行并发处理，有效提高效率。

## Batch

通常情况下接收到的客户端请求都是依次过来的，而当有一批请求过来的时候，就可以通过Batch将这些请求打包成一个Raft log发送出去。

# Multi-Raft

当目前的Raft Group无法支撑所有的数据的时候，就需要引入Multi-Raft处理数据，第一种做法是将数据切分成多个Raft Group分担到不同的机器上。

为了应对更复杂的情况就需要使用动态分裂的方法，假设最开始的只有3台机器以及一个Region，这个Region就布置在一台机器上，所有的数据都写入这台机器，当数据量达到某个预值后Region就产生分裂，得到的新的Region就可以移植到新的机器上。

# 文档

如果对Raft算法有兴趣，强烈建议看一下他的动态演示

地址 http://thesecretlivesofdata.com/raft/

非常易懂，上面介绍的日志复制过程就是整理自这个演示，里面还有很多其他内容，看过后就会对Raft有了整体认识。

还有Raft的详细说明文档，中文的，很好的资料，地址：

https://github.com/maemual/raft-zh\_cn/blob/master/raft-zh\_cn.md

# 源码

## raft\_new

### log\_new

#### log\_alloc

##### log\_clear

#### raft\_set\_snapshot\_metadata

## ----

## raft\_recv\_requestvote\_response

### raft\_get\_nvotes\_for\_me

#### raft\_node\_has\_vote\_for\_me

### raft\_votes\_is\_majority

## raft\_recv\_requestsvote

### \_\_should\_grant\_vote

#### raft\_already\_voted

## ----

## raft\_recv\_entry

## raft\_msg\_entry\_response\_commited

## raft\_get\_first\_entry\_idx

## raft\_get\_last\_applied\_entry

## ----

## raft\_set\_election\_timeout

## raft\_set\_request\_timeout

## raft\_get\_election\_timeout

## raft\_get\_request\_timeout

## raft\_get\_timeout\_elapsed

## ----

## raft\_recv\_appendentries

### raft\_is\_candidate

### raft\_become\_follower

### raft\_delete\_entry\_from\_idx

#### log\_delete

##### raft\_node\_set\_voting

##### raft\_node\_set\_active

### raft\_append\_entry

#### raft\_entry\_is\_voting\_cfg\_change

#### log\_append\_entry

##### \_\_ensurecapacity

##### raft\_offer\_log

###### raft\_add\_non\_voting\_node

raft\_add\_node

raft\_node\_new

## raft\_recv\_appendentries\_response

### raft\_node\_get\_match\_idx

### raft\_voting\_change\_is\_progress

### raft\_node\_is\_voting\_commited

### raft\_node\_has\_suffient\_logs

## ----

## raft\_begin\_snapshot

## raft\_begin\_load\_snapshot

### log\_load\_from\_snapshot

## raft\_end\_snapshot

### raft\_get\_num\_snapshottable\_logs

#### raft\_get\_log\_count

#### log\_get\_base

### raft\_poll\_entry

#### log\_poll

### raft\_end\_load\_snapshot

### raft\_get\_snapshot\_last\_idx

### raft\_get\_snapshot\_last\_term

## ----

## raft\_periodic

### raft\_get\_num\_voting\_nodes

#### raft\_node\_is\_active

#### raft\_node\_is\_voting

### raft\_node\_is\_viting

### raft\_get\_my\_node

#### raft\_get\_nodeid

#### raft\_node\_get\_id

### raft\_is\_leader

### raft\_become\_leader

#### raft\_set\_state

#### raft\_node\_set\_next\_idx

#### raft\_get\_current\_idx

#### raft\_node\_set\_match\_idx

#### raft\_send\_appendentries

### raft\_send\_appendentries\_all

### raft\_snapshot\_is\_in\_progress

### raft\_get\_num\_voting\_nodes

### raft\_get\_commit\_idx

### raft\_apply\_all

## ----

## raft\_free

## raft\_clear

## raft\_set\_callbacks

## ----

## raft\_get\_num\_nodes

## raft\_get\_voted\_for

## raft\_get\_node\_from\_idx

## raft\_get\_current\_leader

## raft\_get\_current\_leader\_node

### log\_get\_at\_idx

## raft\_set\_last\_applied\_idx

## raft\_is\_follower

## raft\_is\_connected

## ----

## raft\_node\_get\_udata

## raft\_node\_set\_udata

## raft\_node\_has\_vote\_for\_me