Raft分布式 一致性算法原理(选举和同步)

一. 背景

在集群环境下，很容易出现单节点故障的问题，那么我们就需要进行集群部署，但是当集群部署的环境下，我们如何保证工作有序的调度与通信并且保证一致性呢，当客户端发送一连串指令，我们需要在集群环境下，所有服务机器最终要保证一致性，而且在出现一系列异常并且恢复之后的情况下，仍然要保证 **最终状态的一致性(状态机)**，于是就引出了分布式一致性协议。

在说到 **分布式一致性协议**，多数人第一个想到的就是大名鼎鼎的 **Paxos** 协议，因为我们最常用的 **Zookeeper** 采用的就是Paxos协议，只不过对Paxos进行了改造，而且Google的很多大型分布式系统都采用了Paxos算法来解决分布式一致性问题，但是由于Paxos对于初学者来说很不容易理解，Paxos的最大特点就是难，不仅难以理解，更难以实现，因此诞生了 **Raft** 分布式一致性协议，有意思的是，raft协议的两为创作者起初也是因为Paxos协议不容易理解和实现，于是Raft被迫营业了。

Raft协议 **(共识算法)** 是目前使用比较广泛的 **具有一致性，高可用且去中心化的分布式一致性协议**，比如现在比较流行的 **etcd分布式存储仓库** 组件就是使用raft协议实现分布式一致性的，在本文中主要介绍几个方面：

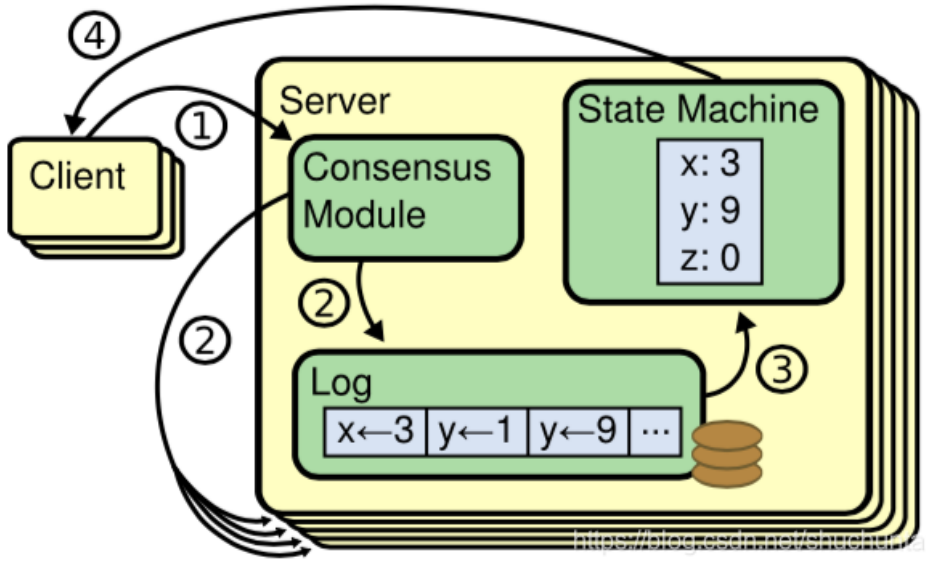
- 复制 状态机
- raft的基本概念
- raft主从选举原理
- raft数据同步原理

二. 复制状态机

何为复制状态机，首先状态机就是每个节点都是具有状态的，且状态机是确定性的，不同的状态对应不同的角色和需要在该状态下的一系列操作，这个状态可能是临时的，可能是长时间保持的，但是状态机都是为了实现容错和一致性所设计的，那么复制状态机就是在状态机的基础上增加同步复制的功能，在同步复制的时候，每个节点可以在缓存或者磁盘保存同步复制的位置和顺序，该模型一句话描述就是： **多个节点上，从相同的初始状态开始，执行相同的一串命令，产生相同的最终状态。** 所以状态机需要实现以下方法：

1. 在多个独立的服务器上放置状态机的副本。
2. 接收客户端请求，解释为输入到状态机。
3. 选择输入的顺序。
4. 在每台服务器上按所选顺序执行输入。
5. 通过状态机的输出响应客户端。

6. 监视状态或输出差异的副本



这就是复制状态机的逻辑图，利用 **state machine** 和 **log** 实现复制和一致性。

三. raft的基本概念

1. 三个状态

- **leader(领导者)**

  表示该节点是主节点，该节点主要处理与客户端的通讯和交互，还有与follower从节点的的数据同步与心跳机制的维护。

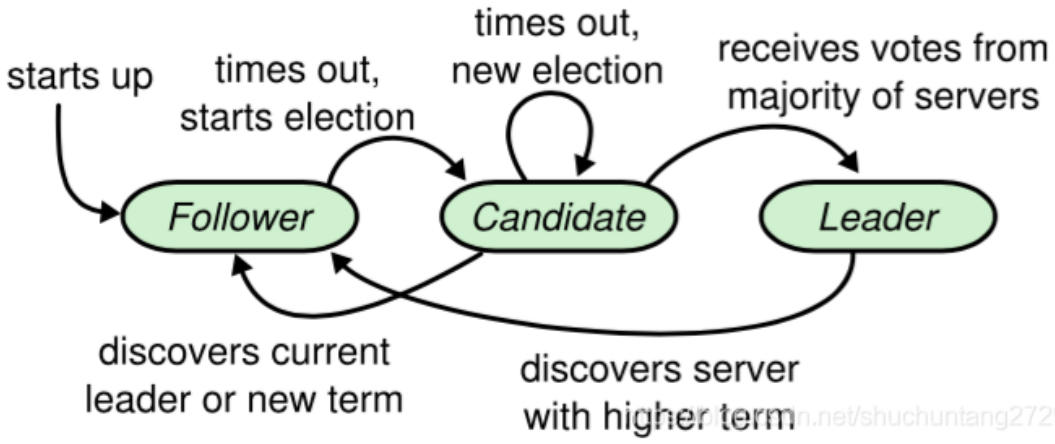
- **follower(追随者)**

  表示该节点是从节点，该节点主要负责转发客户端的请求到leader节点，然后保持与leader节点的数据同步。

- **candidate(候选者)**

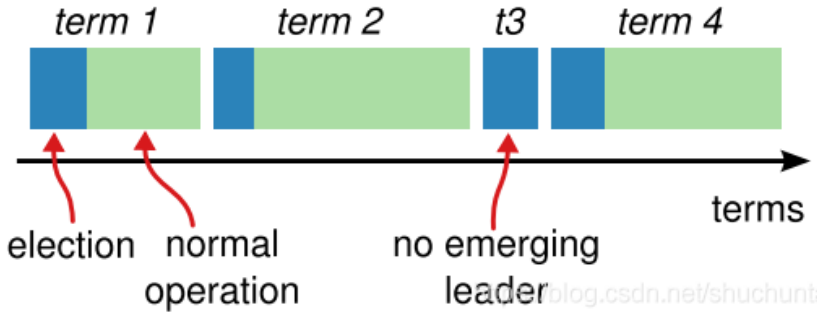
  表示该节点是选举主节点的候选人，该节点状态下表示现在是选举主从节点的时间点，作为候选人，既有可能成为leader，也有可能成为follower，具体的选举原理，下文会详细介绍。

这三种状态的转换图：



2. Term和vote概念

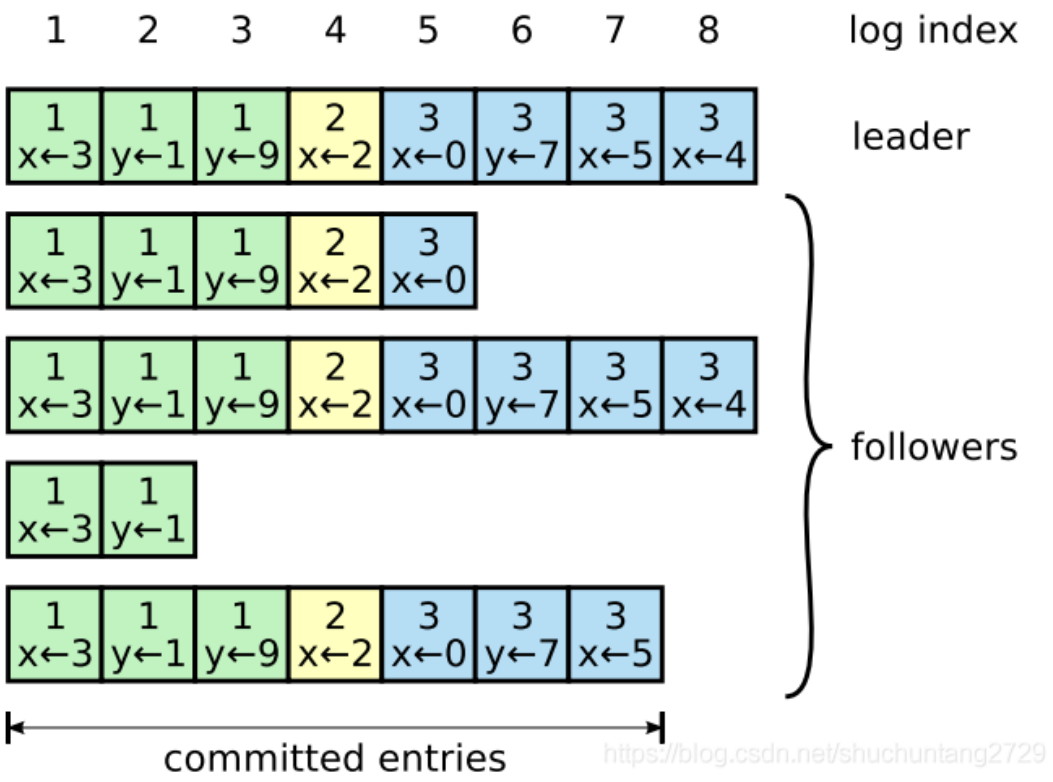
term属性表示投票的轮数，也可以理解是任期数，term用连续的数字进行表示。当然了，该属性一般会持久化到磁盘中，在所有节点刚开始的时候或者在重新选举的时候会进入candidate状态，进行选举投票，每一轮投票之后term都会加1，从下图可以看出来，不管这一轮选举是否成功选出leader，只要一轮结束，term都会加1。



而vote属性就是代表各个节点投票的时候，该节点选择投的那个节点，每个节点在收到投票请求的时候，就是根据request vote进行统计和判断是否自己成为leader的依据和条件。

3. log与index概念

log日志每个节点都持有一份，并且是用来保证一致性和容错性的，首先来看下图。从这张图可以看出来，每个日志文件包含： **index索引号，term任期号，以及term任期号下的数据信息，这么做的好处是为了节点启动后容错检查和主从同步的一致性** ，下文会详细介绍。



4. RPC协议类型

1. **AppendEntries RPC**
   leader使用AppendEntries RPC同步日志数据给其它follower, (心跳是没有日志记录的AppendEntries RPC)。
2. **RequestVote RPC**
   Candidate状态的时候所有节点通过发送RequestVote RPC来发起选举。
3. **InstallSnapshot RPC**
   正常情况下leader是使用AppendEntries RPC同步日志数据，但是当一个follower落后leader太多时，raft会使用InstallSnapshot RPC 来使其快速补充数据。

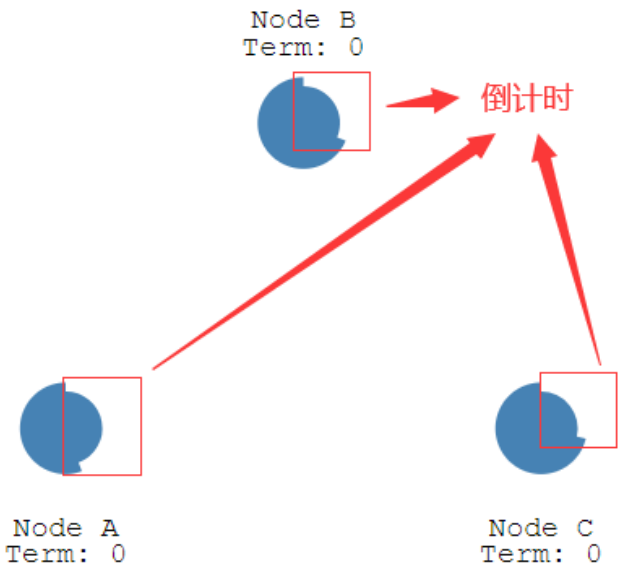
5. 小结

在这一节主要介绍了raft协议的三种状态，term任期概念，以及log日志和index索引的概念，在介绍完这些基础概念之后，可能大家还是一头雾水，别着急，介绍这些基本概念是为了在介绍下面的主从选举和数据同步做铺垫，下面开始进入核心内容。

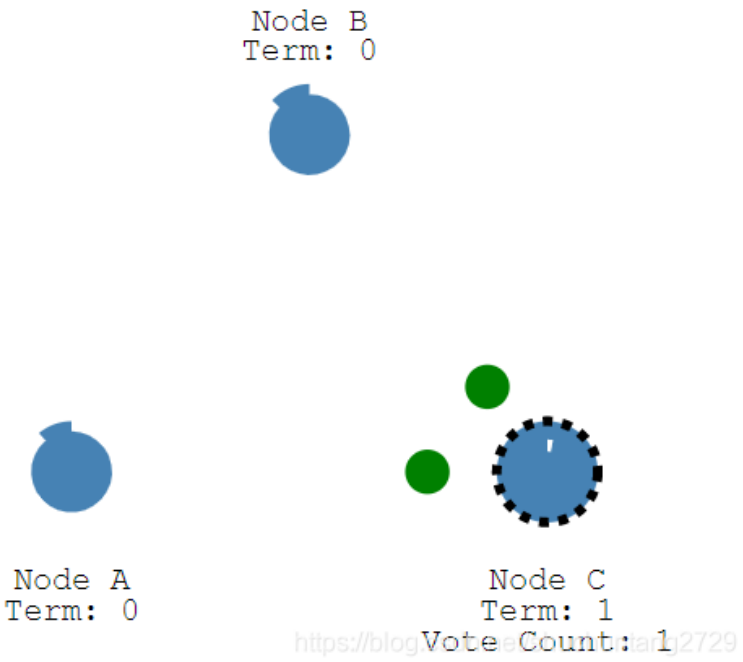
四. 主从选举原理

首先，我们需要知道的是，raft和 zookeeper 一样都采用 **过半选举** 的机制，所以我们推荐raft的实现同ookeeper一样，也是奇数个节点部署。现在，我们假设开始部署三个服务节点：

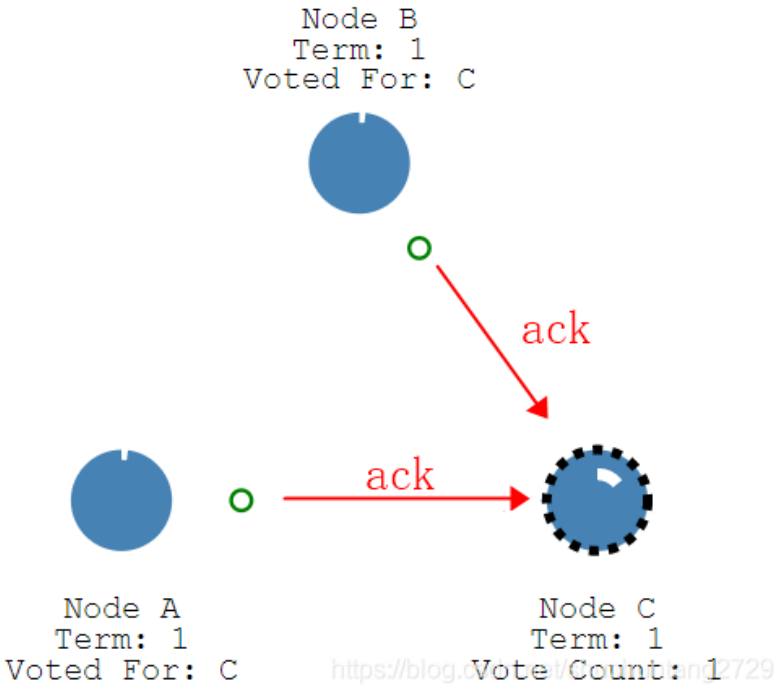
1. 在这三个节点进行启动的时候， **首先进入candidate(候选者)的状态** ，然后进行投票选举，这个时候，所有的节点都会在本地创建一个 **term属性，并且初始化值为0** ，表示还没有开始选举投票，然后每个节点都会设置一个 **随机数的倒计时(150~300ms)** ,当这个倒计时结束的时候，第一个倒计时结束的节点会率先发起投票，并且投票给自己，即携带（vote = 自己）的投票请求给其他两个节点。



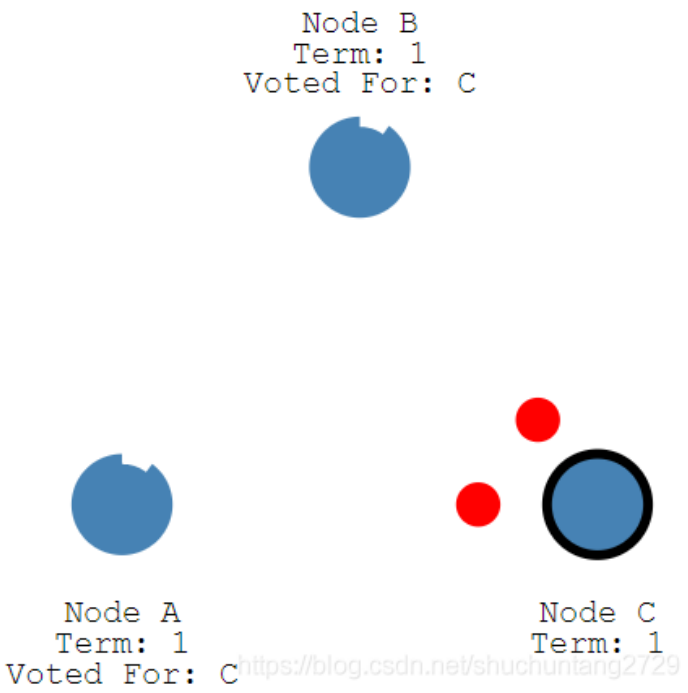
2. 假设节点C的倒计时先结束，它率先发起了投票，则节点C将自己的term从0置为1，并且vote票投给自己，然后将这些参数包装发送给节点A和节点B，这个时候节点A和B还没有结束倒计时：



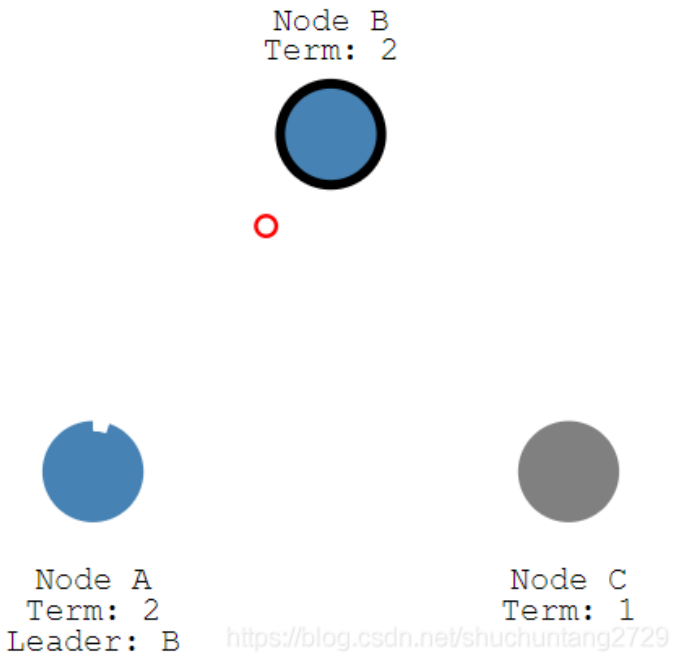
3. 节点A和B在收到C的投票请求之后，首先会比较收到的term和自己的term作比较，如果发现接收到的term数值比自己的大，则会放弃自己的投票，把自己的票(vote)投给第一个接收到的投票请求，即节点A和B在收到C的请求之后，比较发现自己的term为0，而接收到的term为1，则会把自己的vote置为节点C，然后响应请求：



4. 当节点C收到了节点A和节点B的响应(ack)和投票信息(vote)，则会计算票数，由于过半选举的机制，一个节点需要成为leader节点，必须得到 **(n/2 +1)** 的票数，这里n=3，即得到2票就可以当选leader节点，而C节点收到了节点A和B的两票，再加上自己的一票，总共 **3票** ，则满足条件，当选leader，当选leader之后，向节点A和B发送心跳包，并且告诉它们自己 **成为了leader节点** ，那么节点A和B就会 **从candidate(候选者)状态转为follower(追随者)** 状态，并开始正常工作流程了：

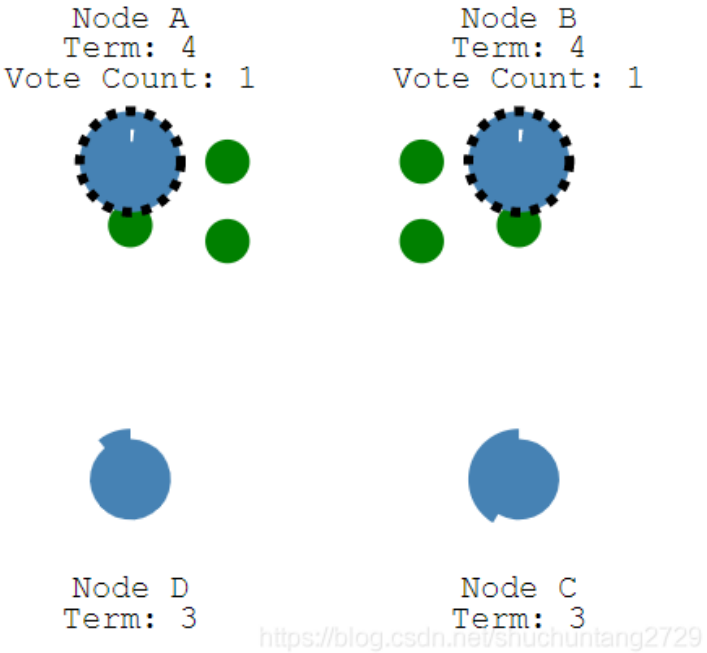


5. 正常情况下，选举就已经结束了，那么如果遇到异常情况呢，现在假设leader节点发生了故障宕机了，这里就要强调一个点，就是每个follower节点在和leader节点维持心跳的时候，都会设置一个 **超时时间(timeout)** ，当follower节点在等待leader节点的心跳请求的时候如果超过了这个超时时间，就会认为leader出现了异常，进而会 **重新进入candidate状态** ，然后重新选举leader：

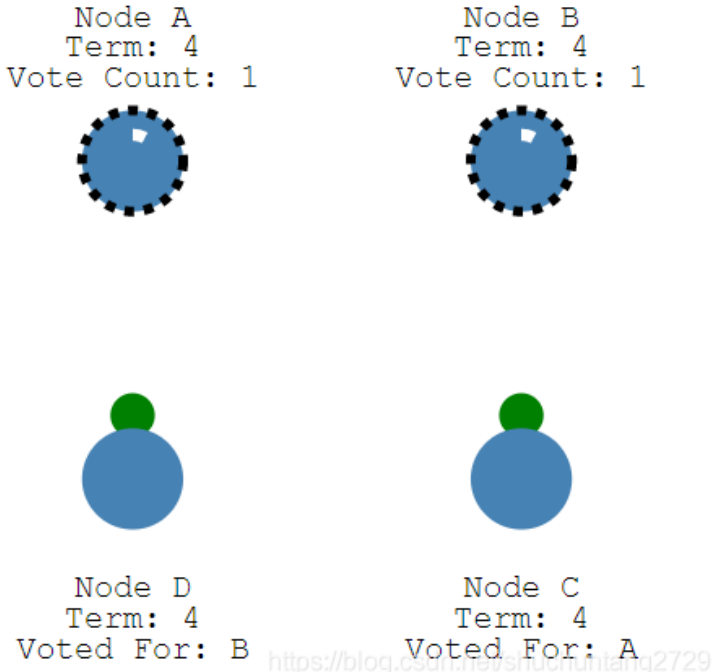


6. 从上图可以看出来，现在假设节点B经过上面的流程然后成为了新的leader节点，这个时候节点A和节点B的term(任期数)从1增加到了2，但是由于节点C的宕机，节点C的term还是原来的1，这么做的好处就是当节点C重新启动的时候，发现自己的term已经落后了，那么就不能成为leader，就等待新的leader将新的数据同步给自己，不会出现数据丢失的情况。

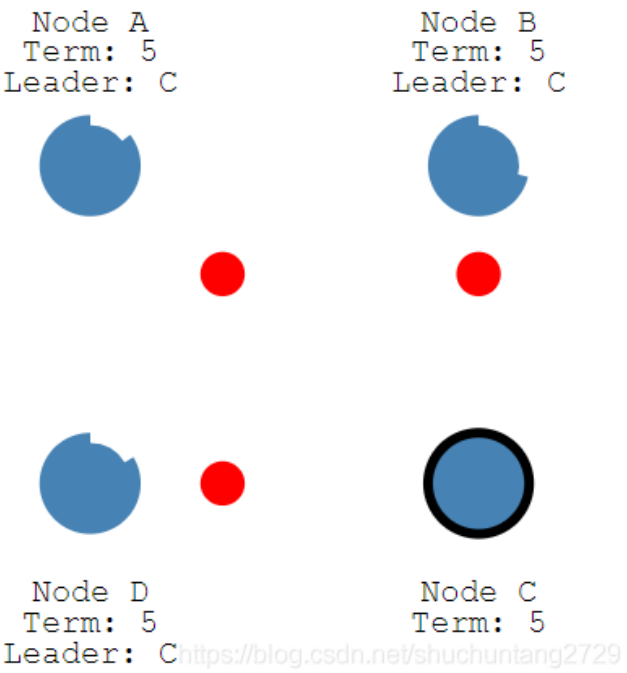
7. 可能有人就有疑问了，会不会出现多个leader的情况呢，现在假设增加一个节点，即现在有 **4个node** 节点，然后由于新节点的加入，所有节点进入candidate状态并开始 **重新选举leader** 流程，从下图可以看到，节点A和节点B的倒计时同时结束，然后term加1，并发起投票：



8. 由于A和B同时发起投票，而节点C首先收到的是节点A的请求，则C把票投给了A，而节点D首先收到的是节点B的请求，则D把票投给了B，这样就会出现节点A和节点B收到的投票数都不过半，那么就要 **开启下一轮请求**：

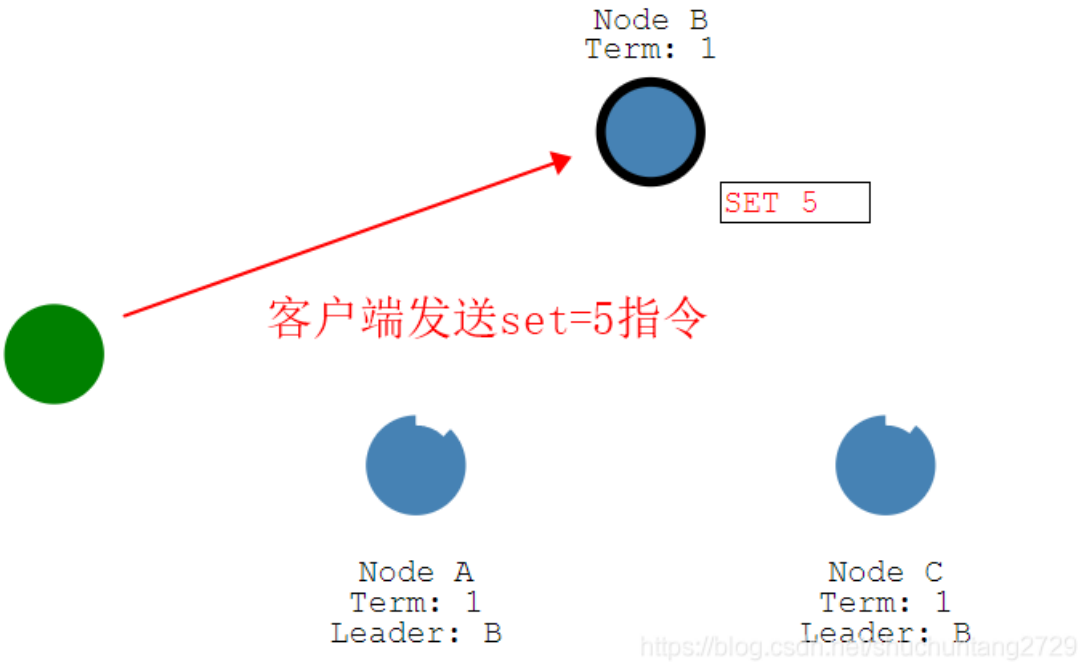


9. 这里所有节点的term都增加到了5，说明又走了一遍投票流程，而且新的一轮选举出了节点C为leader：



五. 数据同步原理

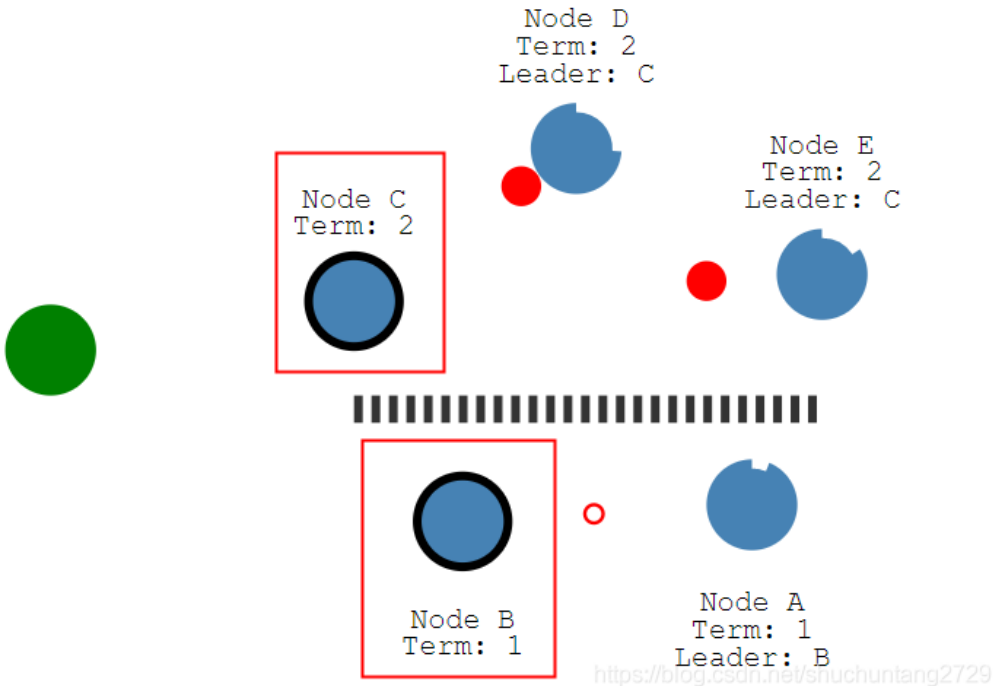
前面说到了复制状态机的其中一个作用就是保证容错和一致性，当一个主节点正常工作的同时，还要保证与从节点的数据同步，这里的同步不仅仅是简单的同步，还要保证某个节点宕机重启后还能拿到所有的数据达到一致性，那么下面开始介绍Log Replication同步的过程和步骤：



1. 看上图，现在有三个节点A,B,C，其中节点B是leader节点，A和C是follower节点，客户端向leader节点B发送数据(set 5)，节点B在收到指令的时候开始开始做数据存储和数据同步，这个时候并不是直接存储，而是 **采用2PC两阶段提交** ,先去请求所有的folloer节点，具体流程是：
    1. leader节点B收到客户端数据请求，对所有follower发起2PC请求，
    2. follower节点收到leader的请求，确认自己可以执行数据存储，响应ack给leader，
    3. leader收到ack响应，会进行判断，如果有 **过半的节点** 返回了可以执行的ack，则 **leader进行数据存储,并记录日志** ，
    4. leader **返回给客户端成功的响应** ，
    5. 然后leader再次向所有folower发送确认执行的请求，
    6. follower **进行数据存储,并记录日志** ，
    7. 需要注意的是： **从节点也是接受客户端请求的，只不过不处理请求，会转发给leader节点处理。**

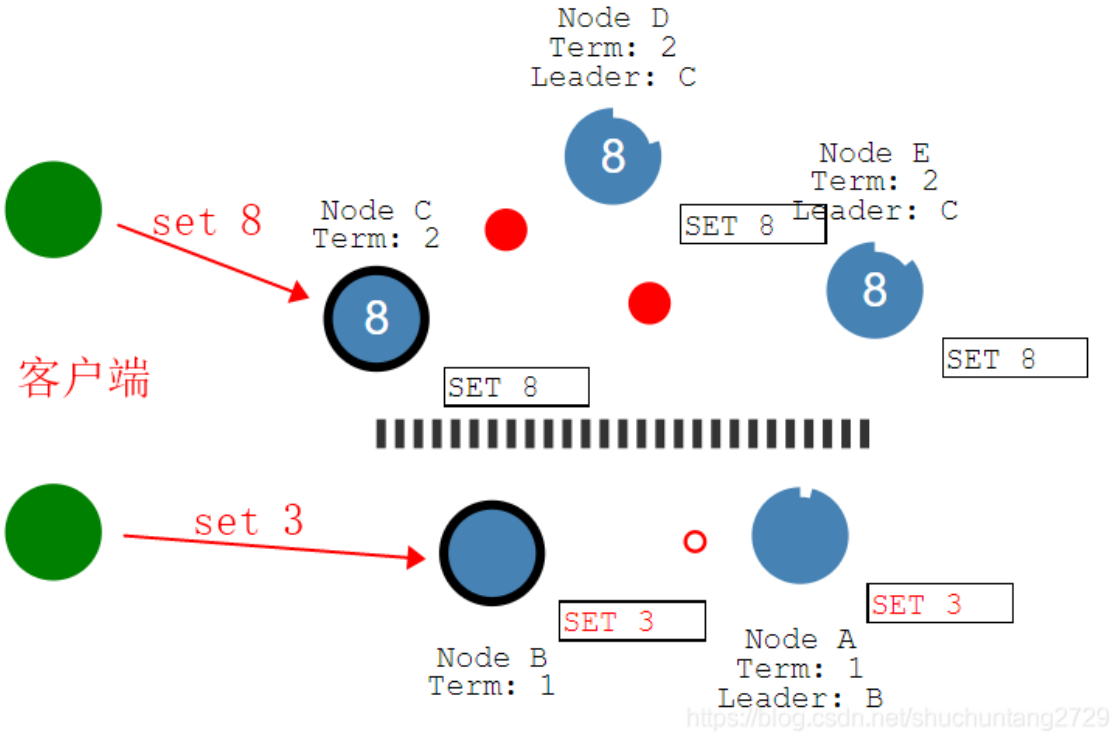
> 如果大家对2PC两阶段提交协议还不是很清楚的，可以看这一片文章，传送门：分布式事务

2. Raft甚至可以在面对网络分区时保持一致，大家在看到这种数据同步的问题的时候，都会考虑脑裂或者网络分区的问题，比如redis，但是raft是可以解决这个问题的：

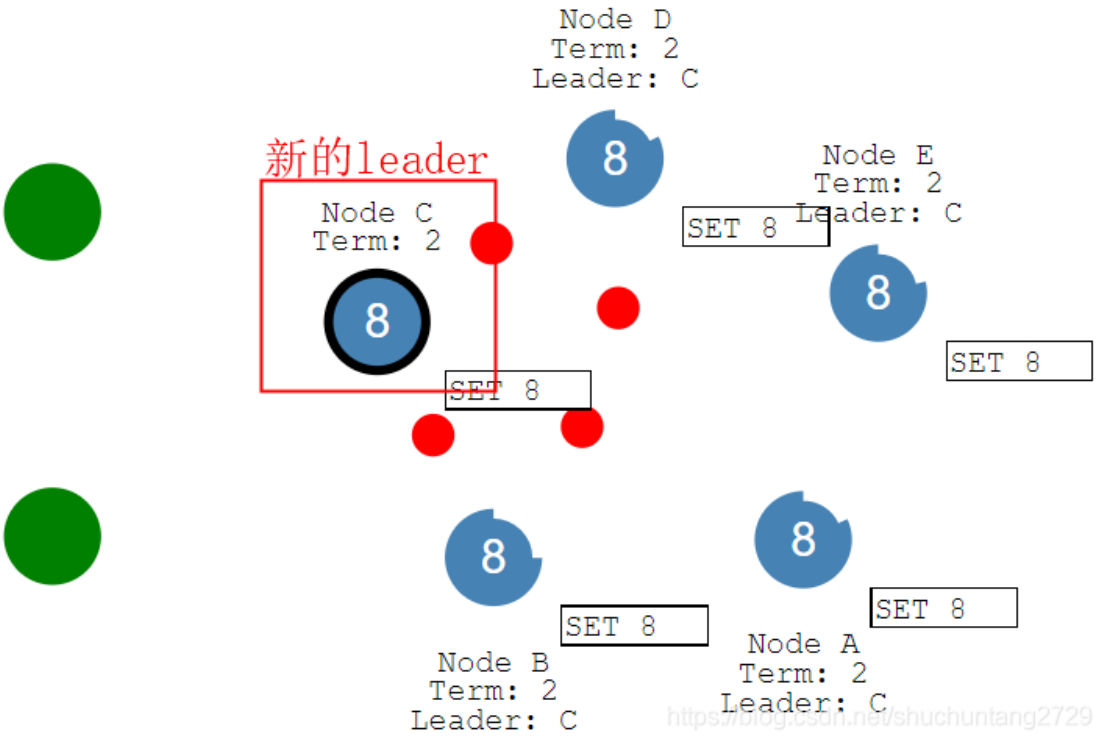


这里可以看到，5个节点A,B,C,D,E，一开始的时候节点B是leader节点，由于网络原因，产生了分区，其中A和B在一个区，而C,D,E在一个区，由于C,D,E所在的区没有leader节点，所以他们重新进行了选举，所以上面三个节点的term任期都是2，而A和B一直是正常工作的，所以他们的term还是原来的1，现在的情况就是，分区1：A和B保持心跳连接，B是leader；分区2：C和D和E保持心跳连接，C是leader。

3. 然后客户端与服务节点正常通信，首先客户端向B发送set 5的指令，由于节点B和节点A **无法满足所有follower节点过半成功** 的机制，所以无法进行数据存储和同步，也无法正确的响应客户端，而节点C,D,E在收到客户端set 8的请求后，是可以正常进行数据存储和同步的，并给客户端一个正确的响应：

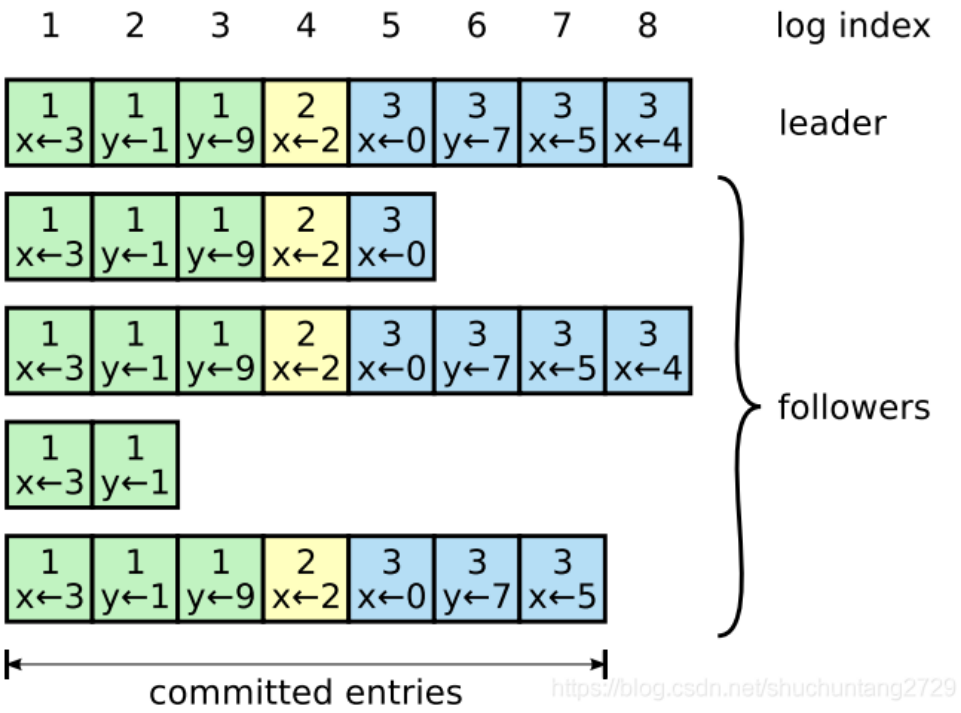


4. 当网络正常了之后，这五个节点收到了其余节点的心跳，而由于节点A和B的term已经落后了其他三个节点，则他俩无法成为leader，所以两个 **节点A和B都将回滚其未提交的条目，并与新的leader的日志相匹配，并且更新term为最新的2**：



5. 上面简单介绍一下数据同步的思想和步骤，下面再详细介绍数据同步是如何利用log进行容错和一致性的，我们知道，我们在考虑服务集群的情况下，总是认为服务是不稳定的，因此可能其他节点当宕机然后重启之后，会出现数据版本落后的情况，那在这种情况下，leader是如何保证让这个follower拿到所有落后的数据，然后达到一致性的呢，具体步骤如下：

    1. leader节点会为所有的follower节点都维护一个 **next index变量，next index = log index + 1** ,这个变量表示当前最新日志位置的下一个位置，
    2. leader在向follower节点发送心跳的时候，会带上 **(term,next index)** 信息，
    3. 当存在落后的follower节点收到心跳之后，会那自己的(term,next index)与收到的(term,log index + 1)比较，
    4. 当follower发现自己不匹配，会响应给leader拒绝的消息，
    5. 然后leader收到了拒绝的消息，会重新发一条消息，这次携带的是 **(term,next index - 1)** 的消息，
    6. 然后重复上面的步骤，总的来说就是，当follower拒绝消息，leader则发送上一次的log位置给follower，依次递减，直到follower返回正确的响应，表示接受数据的同步， **则向后补满所有的log日志** ，达到数据一致性的结果。

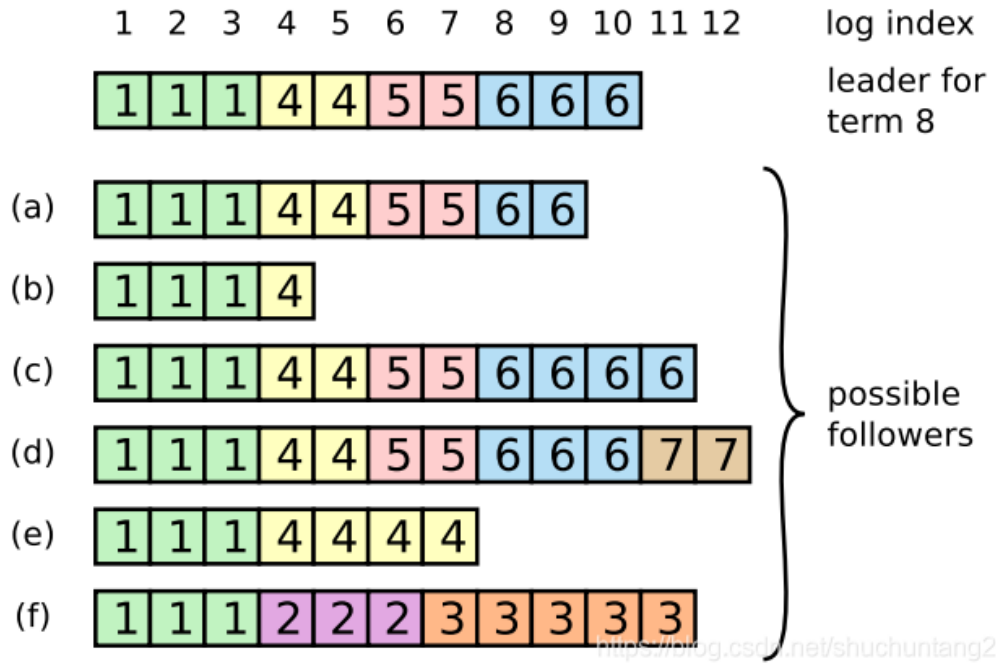


根据上图的例子来看，现在的leader最新的日志文件保存的信息是：log index = 8；next index = 9，term = 3；set x = 4，然后第一个follower节点现在是落后了三个log index，当这个follower启动后，leader与它同步数据的步骤如下：

1. leader发送(3,9)的心跳给第一个follower，
2. follower收到(3,9)的请求之后发现自己的log信息为(3,5 + 1)，并不相等，则返回拒绝的响应，
3. leader收到第一个follower的响应之后，然后向再次发送(3,8)的请求，
4. follower收到(3,8)的请求之后发现自己的log信息为(3,5 + 1)，并不相等，则返回拒绝的响应，
5. leader收到第一个follower的响应之后，然后向再次发送(3,7)的请求，
6. follower收到(3,7)的请求之后发现自己的log信息为(3,5 + 1)，并不相等，则返回拒绝的响应，
7. leader收到第一个follower的响应之后，然后向再次发送(3,6)的请求，
8. follower收到(3,6)的请求之后发现自己的log信息为(3,5 + 1)，相等，则返回正确的响应，表示接受创建(3,6)位置数据同步的请求，
9. 以此类推，follower将日志补满。

6. 到这里大家可能就清楚很多了，但是还是存在一个疑问，就是当客户端不仅仅是落后数据，而是其他的情况怎么解决，就比如说下图中，follower即有可能出现(a),(b)这种数据落后的情况，也可能出现例如©数据多出来的情况，还有可能出现(d),(e),(f)这三种数据异常的情况，raft是如

何解决的呢，很简单，Raft 始终会认为领导者的日志总是正确的，即 **所有的follower节点必须强制保持和leader节点一致数据** ，也就是说，不管出现下面的(a),(b),©,(d),(e),(f)中的哪一种情况，在正常工作的情况下，所有的日志数据都必须强制最终与leader节点一样，这也是可以理解的，因为作为leader并不知道你的异常数据是什么原因导致的，但是我既然是leader，那所有的follower必须与我的log保持一致，不然肯定会出现日志记录会无限堆积并出现混乱的状态等各种各样的问题。



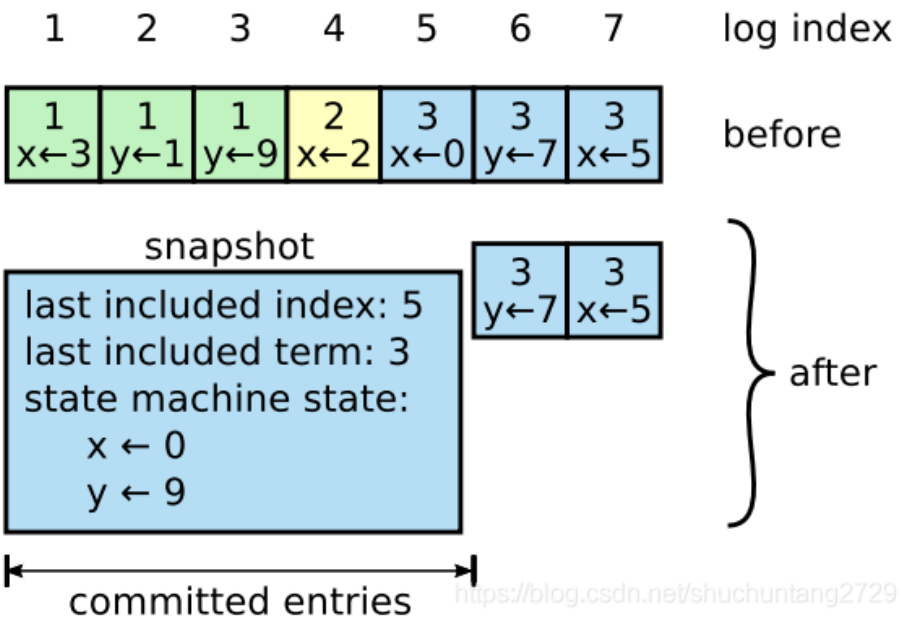
7. 可能又有人提出疑问了，假如一个日志文件落后了很多版本的节点当选了leader，然后又强制所有的follower与自己保证一直，那这数据不全都丢失了吗，对于这个问题，raft肯定也想到了，所以当进行选举投票的时候，所有的节点都要携带自己的term和log index信息， **对于那些日志数据落后的节点，直接拒绝投票，也就是说，只有拥有最新的日志信息的节点，才有可能当选leader，比较规则是：比较term，term大的拥有选举权，如果term相等，则比较log index，大的拥有选举权，如果都相等，则公平竞争。**

六. 日志快照

随着raft所有的节点不断运作，不管是leader还是follower都会产生大量的log数据，就会占用各种资源，而且数据量一旦过大，也会影响正常的效率，所以raft采用 **日志快照snapshot** 思想，进行日志归档和清理。其中每一个节点都会维护自己的日志快照，默认规则是当日志量达到一个阈值，就会对之前的一组数据进行快照生成，例如下图，生成的snapshot包含：

- 当前快照包含的最后一条日志信息的 **term：3**
- 当前快照包含的最后一条日志信息的 **log index：5**

- 当前快照包含的所有的数据并集(因为存在对同一数据的多次操作，而这里只保存最终结果，删除过期且无用的数据记录，所以 **x:0,y:9**)。



在多数情况下，每个节点只维护自己的快照数据，leader并不会进行干预，但是当某一个follower落后了太多数据信息的时候，leader会主动干预，发送InstallSnapshot RPC使follower快速补充数据。

七. 节点扩缩容

> 当raft集群的服务节点正常工作的时候，我们突然想对该集群进行扩容操作，比如原来集群是三个节点，现在想扩容到五个节点，这 里就需要考虑采用下面的三种实现模式了。

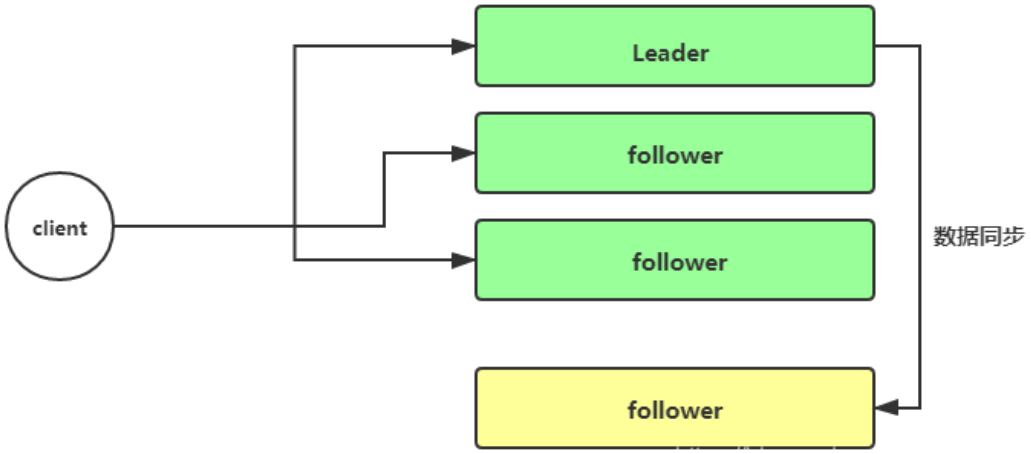
1. 停机扩缩容模式

停机扩缩容，顾名思义就是停止该服务，然后准备好新的节点之后采用新的配置文件启动，重新进行选举和数据同步，这个过程中，raft服务整体是对外不可用的，不可用时间的时常取决于服务什么时候重启好然后工作，这样肯定就 **破坏了服务的可用性** ，很不友好，但是采用这种模式， **不会出现脑裂或者数据不一致的问题** ，这个在下面的扩缩容模式会详细说明。

2. 单节点扩缩容模式

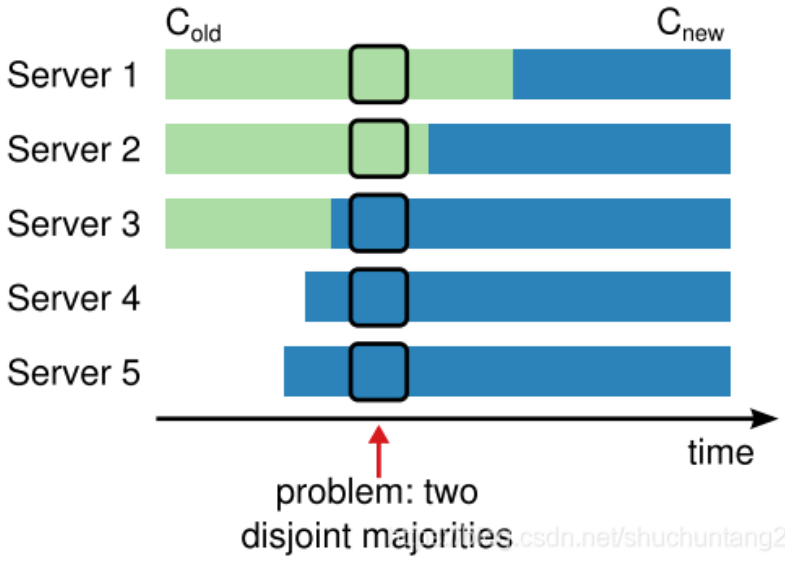
单节点扩缩容，由于停机扩缩容模式会导致服务短时间不可用，因此出现单节点扩缩容模式，该模式下原来的集群继续对外提供服务，然后先扩容一台，这个时候每一个节点依次修改配置文件，进行升级扩缩容，这里需要注意一点的就是，扩容时，当新加入的节点刚启动之后，需要从leader节点同步大量的日志数据，因此这个时候leader仍然处于短暂的服务不可用状态，这又是和停机扩缩容模式一样的问题，不过单节点扩缩容模式下raft解决了这个问题，即 **日志追赶** ，就是 **当新节点加入进**

来之后，暂时不作为服务节点对外提供服务，只有当该节点同步完数据之后，才真正的加入集群对外提供服务，这个过程中原来的集群中的服务仍然正常对外提供服务 。这种模式也是多数公司采用的模式，不仅解决了可用性的问题，也解决了动态扩容的问题。



3. 联合共识模式(joint consensus)

当我们不想停机扩缩容，也不想单节点扩缩容，而是想直接对多个服务节点同时进行扩缩容，这种方式是我们觉得最方便快捷的，也是最省力的，但是同样也会带来脑裂和数据不一致的问题：

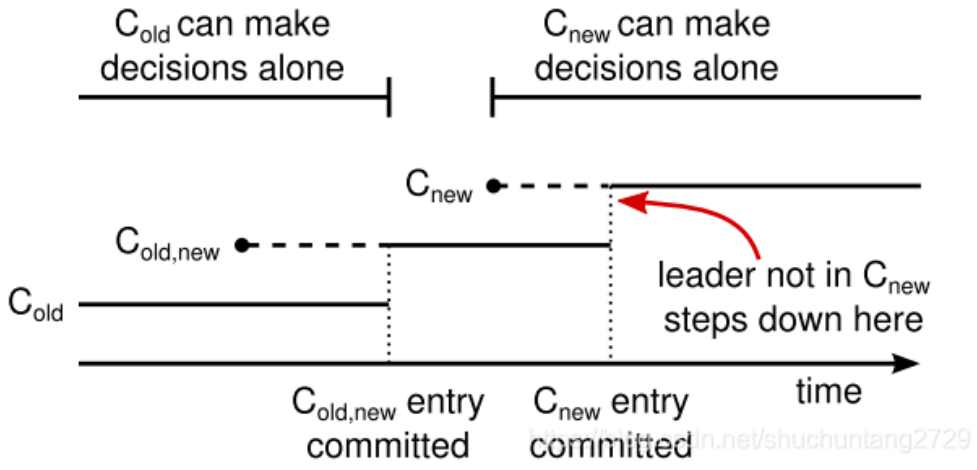


比如上图中，原来的集群有三个节点，这个时候我们动态的扩容到五个节点，当对节点3进行升级之后，这个时候由于各种原因重新触发了选举，那么由于节点1和节点2还是老的配置文件，那么他们会在节点1-3中重新选出一个leader，而节点3-5是新的配置文件，那么他们会在节点3-5中选出一个leader，这个时候就会出现两个leader，也就是常说的 **脑裂问题** ，这个时候，会有两个集群对外提

供服务，即使最终他们触发选举形成了新的并且完整的集群，那由于之前分别工作的问题，也 **会出现数据丢失和不一致的问题** 。

raft为了解决这个问题，设计了 **联合共识模式(joint consensus)** ，这个模式大致的思想是：

- 首先leader节点向所有得follower节点发送联合共识日志(C-old,C-new)；
- 所有的follower节点收到联合共识日志(C-old,C-new)之后，立马提交并使用，不会等待leader的类似2PC的commit指令；
- 这个联合共识日志(C-old,C-new) 是包含了旧配置和新配置的并集，假设在这个阶段有客户端的请求操作，leader会向所有的follower节点发送同步数据请求，在(C-old,C-new)阶段，不仅保证旧配置下的节点同步，也保证新配置下的节点同步都满足过半响应才会真正响应客户端；
- 当所有服务节点都升级扩缩容之后，leader会向所有follower节点发送(C-new)配置变更请求，告诉所有节点，可以切换到新的集群环境工作了；
- 然后所有follower节点收到请求后，仍然立马提交并使用，不会等待leader的类似2PC的commit指令；
- 接下来就是正常的工作流程了。



八. 总结

这一篇主要介绍了raft分布式一致性协议的思想和实现原理：

- 首先是raft的目标是为了解决在集群环境下，需要所有服务节点保持容错和一致性的问题；
- 其次是raft利用状态复制机的原理和思想实现了该目标；
- 再然后就是利用复制状态机的思想进行主从选举和数据同步，以及容错的实现。

对于哪些开源项目是利用raft协议实现的，大家可以去看一下etcd开源项目，只不过etcd是使用go语言实现的，推荐大家去看一下阿里开源的nacos项目，是用java实现的，可读性还是很好的(这里指的是nacos的CP模式)。