raft论文理解

以Q&A的形式来描述raft是什么，怎么工作的。

刚开始一提到raft真的头大，想想一大堆“如果”，我该如何下手，但是我想到一种方法，就是把raft分成正常和异常两部分来处理，正常描述的就是假设系统不会出现异常的情况下raft是怎么跑的，异常就是raft会在哪里可能出现什么样的异常，这里我画个图简单归类下：

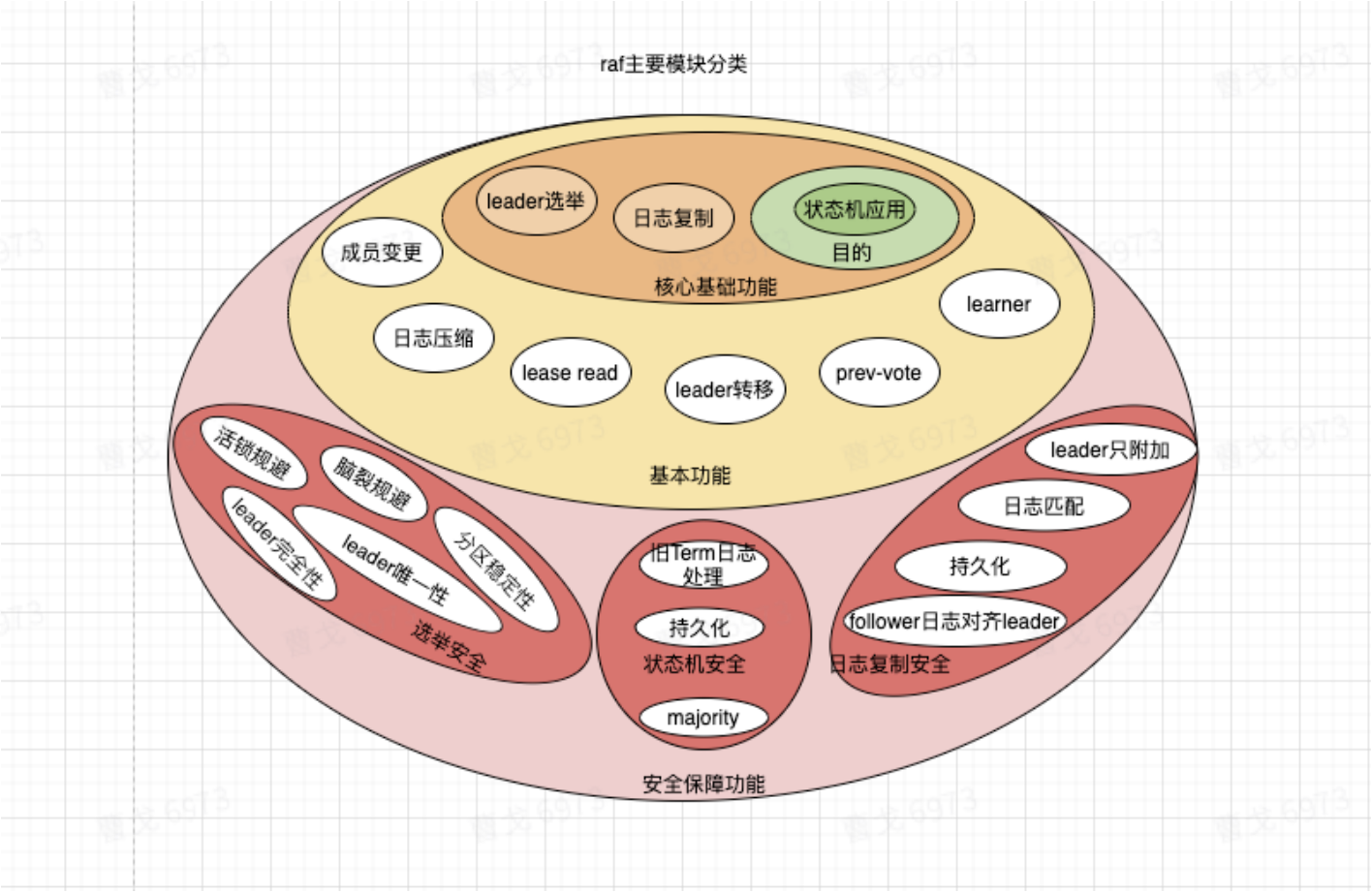


图-raft主要模块分类

下边我简单说明下各个模块的功能，更详细的功能在下边的Q&A都有涉及，除了一些安全性的模块暂时还未开展。

leader选举：

选举出本次Term的唯一leader作为对外服务的节点。

日志复制：

日志同步。

状态机应用：

日志状态应用。提供对外数据读取。

成员变更：

集群扩缩容。对raft成员组成进行调整。

日志压缩：

日志会随着日志的复制不断的增大，考虑读写性能及日志恢复的快速能力，需要对日志进行压缩,也可以理解为清理掉无用的日志。

lease read:

一种优化leader读取性能的措施，lease read主要给follower进行对外开放读能力的授权，在lease有效期间，follower可以提供读取能力，如果lease过期，可能是leader离线了，这时候从follower读取数据可能存在不一致。

leader转移(leader transfer):

leader节点可能要升级，transfer之后可以进行重启等操作。

prev-vote:

主要用在解决成员变更时候出现网络分区导致系统不稳定，具体的是新加入的节点先与集群中的其它节点通信，确认leader是否存活，如果存活，就不会转为candidate,不增加Term。当然这个没有彻底解决问题，主要还是要single-server changes,不过这能在一定程度提高效率。有leader的那个分区就是稳定的，无leader的那个分区它也选举不成功，如果结合single-change services，这样问题就解决了。

learner:

负责leader的备份，主要为了缓解在日志恢复的时候leader日志复制的压力。

活锁规避:

其实就是多个节点在选举超时时间内同时进行选举，这个原因也可能是分区稳定性中提到的因素导致的，怎么规避呢，通过随机等待的方式，这个在raft的paper里有实验证明，在150ms~300ms的随机等待时间有比较好的效果，能规避这个问题的同时还能减少等待时间。

脑裂规避:

能正常服务的节点只要不是偶数个就没有问题。

分区稳定性:

怎么产生的呢，存在这么个场景，假设有一个follower节点单独和leader产生了网络分区，这个时候它就会发起选举，这样就会扰乱正常的系统运行，而这个过程可以反复的执行，导致系统一直在进行选举，所以这个问题需要在工程实现上进行规避，可以通过增加一些额外的规则，例如follower先询问leader是否存在，然后通过majority判断是不是大部分节点认为leader是稳定的，如果是，就不发起选举。

leader 完全性:

leader必须具备所有已经提交的日志。

leader唯一性:

一次选举只能选出一个leader。

旧Term日志处理:

不能单独提交旧日志。

持久化:

在日志附加请求响应之前节点自身必须把日志持久化存储。

Majority:

就是大多数原则，无论系统中的节点有多少个，只要满足超过半数才认可的原则就一定会有至少一个共同节点。

leader只附加:

leader的历史日志就是只读的。只能append新的日志或者对历史日志进行压缩。

日志匹配:

日志条目中都有对应的Term和index，根据这两个进行日志匹配。

follower日志对齐leader:

为了简化日志复制的复杂性，raft简单高效的日志管理方式就是，无论follower日志和leader有什么样的差异，对齐leader的日志。

其实raft就是一种共识手段，而不是目的，对业务来说，唯一有收益的地方就是状态机应用，其它所有模块都是为保证状态机安全及正常应用提供保障的。

当然，raft依然在系统维度是一个单点系统，在工程上还是需要一个真正的raft集群，叫做multi-raft,主要解决的问题我个人理解有两个:

a. 高可用性

整个raft系统需要具备高可用性，不会因为个别raft组处于异常状态而让整个raft系统不可用。

b. 高性能

单个raft节点有单点性能问题，所以multi-raft就是为了解决这个问题。

具体的multi-raft本身需要解决下边三个问题:

c. 负载均衡:通过transfer leader的方式尽量保持每个节点的leader数量一致

d. 连接复用:合并心跳，租约请求

e. 中心节点:管理

a. Q&A

i. **Q:raft是什么，解决什么问题**

raft是一种multi-paxos算法，用来解决分布式系统中的共识问题的

ii. **Q:raft由哪些模块组成**

1. 领导选举:leader具有最高读写权限，日志复制实现一致性由它确保
2. 日志复制:leader从客户端接收日志并复制到其它节点并强制要求和自己保持一致，日志只能从自己复制出去
3. 状态机安全性:任何一个已经应用了一个确定的日志到自己状态机的机器，其它机器不能在同一个日志索引位置上和它有不同的指令(内容)

关注状态机其实，分布式系统中的状态机都是相似的，只是具体的实现方式可能有所区别，如图:

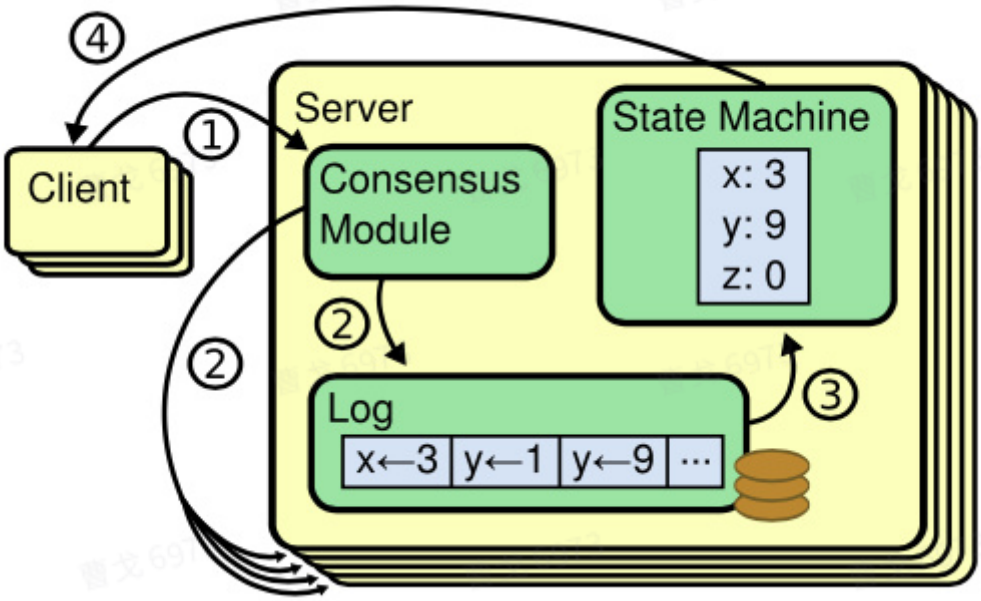


图-raft 复制状态机

4. 从功能上划分其实还有很多个模块，例如leader transfer，日志恢复等等

iii. 关于日志状态上有一些细节性的内容

1. 所有服务器上的持久性状态

	A	B
1	**参数**	**解释**
2	currentTerm	服务器已知最新的任期（在服务器首次启动的时候初始化为0，单调递增）
3	votedFor	当前任期内收到选票的候选者id 如果没有投给任何候选者 则为空
4	log[]	日志条目;每个条目包含了用于状态机的命令，以及领导者接收到该条目时的任期（第一个索引为1

2. 所有服务器上的易失性状态

	A	B
1	**参数**	**解释**
2	commitIndex	已知已提交的最高的日志条目的索引（初始值为0，单调递增）
3	lastApplied	已经被应用到状态机的最高的日志条目的索引（初始值为0，单调递增）

3. leader上的易失性状态

	A	B
1	**参数**	**解释**
2	nextIndex[]	对于每一台服务器，发送到该服务器的下一个日志条目的索引（初始值为领导最后的日志条目的索引+1）
3	matchIndex[]	对于每一台服务器，已知的已经复制到该服务器的最高日志条目的索引（初始为0，单调递增）

iv. **Q:详细的核心工作流程是怎样的(leader选举->复制状态机的工作，暂不详细讨论其它十几个功能模块例如leader transfer，日志压缩等)**

在描述工作流程之前，我们先了解下raft的一些基本概念，首先我们了解下raft的服务器状态有哪些，如下图：

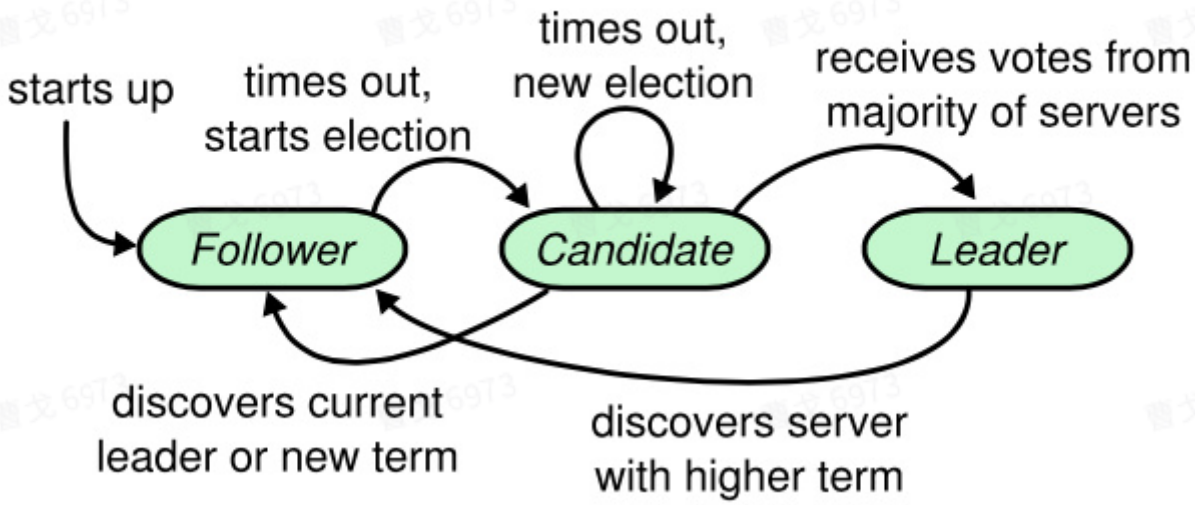
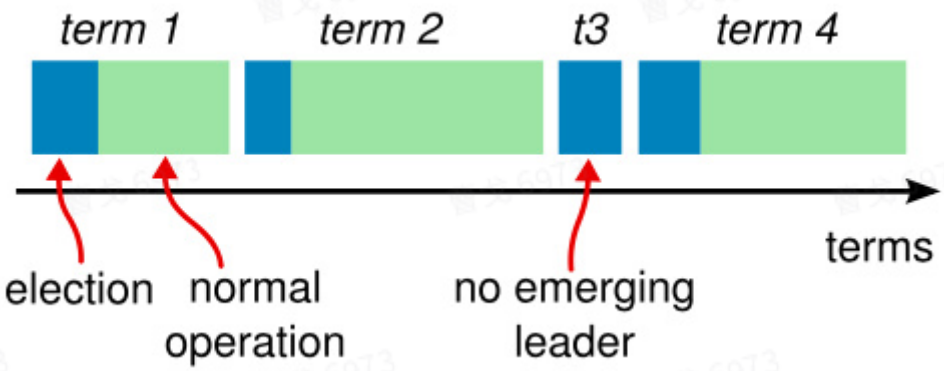


图-raft 服务器角色状态转换图

raft集群中服务器的节点无外乎上图中的三种角色，各自角色的职责在上边已经提过，这里描述了各个角色之间的关系。

其次是关于时间的概念，如下图：



raft将时间按照term声明周期划分，term指的就是一个leader稳定服务的时间范围，如图示例的，有四个term，每个term都有选举的阶段，但是不是每个term都会选举成功的，譬如脑裂造成的选举失败，如图中的term3，选举完成后就进入一个相对稳定的服务时间段，就是图中的浅绿色描述的时间段，该时间段内只有一个leader且leader不切换。所有读写都走leader。leader负责完成日志的复制及状态机的应用。至于安全性我们后边会马上讲到。下边我们看下最基础的核心模块，election，日志复制和状态机应用。

我们分为三个部分描述：

1. leader选举
    a. 触发

    其实内容都在上图的角色状态转换图中有体现，下边我们围绕该图来展开leader选举的核心流程。raft使用心跳机制触发选举的。当服务器启动之后默认进入follower状态，如果在心跳超时之前没有收到来自leader或者候选者发来的选举投票请求就会进入candidate状态并发起选举RPC请求给其它服务器。

    i. 自增自己的的term
    ii. 切换自己的状态为candidate
    iii. 发起选举RPC

    b. 处理RPC结果

    i. **当选**。收到>=majority个数的投票，成功当选当前任期的leader，立即向其它服务器节点发送心跳证明自己已经是当前term的leader了，以便尽快恢复系统的稳定。这个过成需要用到选举安全性的保障，即投票的节点只能对相同term投最多一次票，并且承诺不再给没有当前term大的任期投票。这里其实和paxos区别不大，主要区别还是在额外限制上，其实在raft的选举规则中，基本规则是按照先来先得，但是为了强化raft的一致性，减少状态数量，其实要当选leader还有一些额外的约束，这个是不同于paxos的地方，具体有哪些呢，分别如下:

        1. leader完全特性

        也是下文提到的，一个leader要当选，必须要有所有之前leader已经提交的日志。这个在具体的投票RPC中投票者是根据candidate的lastLogTerm和lastLogIndex与自身的日志记录对比的，如果lastLogTerm比自己本地记录的最后一条提交日志的Term大，则认为该candidate可投，如果比自己的小则直接拒绝，认为该candidate比较旧了，如果相等，就进一步比较自己最后提交的日志index是否比lastLogIndex小，如果比lastLogIndex小，则认为该candidate包含的日志比自己的新，统一投票，否则比自己小的话也是拒绝投票的。

    ii. **被别的节点当选**。收到更高term的附加日志RPC请求，当前节点会切换为follower，并重置选举超时计数器。
    iii. **脑裂**。

这个过程中，follower,candidate离线了都会被处理，follower离线了，如果还有>=majority个节点在线，本次选举不会受到影响，否则本次失败，但是不会产生错误的结果，candidate离线了，所有的follower会选举超时，然后会有一个新的节点重新当选为candidate。该过程可重复，直到选举成功。

2. 日志复制

一旦leader被选举出来，系统就处于稳定状态了，开始对外提供服务。具体的服务方式是，leader对客户端的请求会生成一个raft系统的日志，日志中包含客户端的operation信息例如用户的数据，同时也包含leader的term,logIndex,leader会发起日志附加RPC请求给follower，这个请求是并发的。当>=majority个数的follower确认日志已经持久化到自己的log记录中之后leader可认为符合状态机安全，leader紧接着会应用这条日志条目到它的状态机中然后执行的结果返回给客户端。

在日志附加RPC中如果失败，leader会不断的重试请求，这个过程和对客户端的返回可以是并发的，客户端能否收到回复不要求所有的follower都回复成功，满足majorityge个数即可，leader发送重试直到所有的follower最终响应成功。

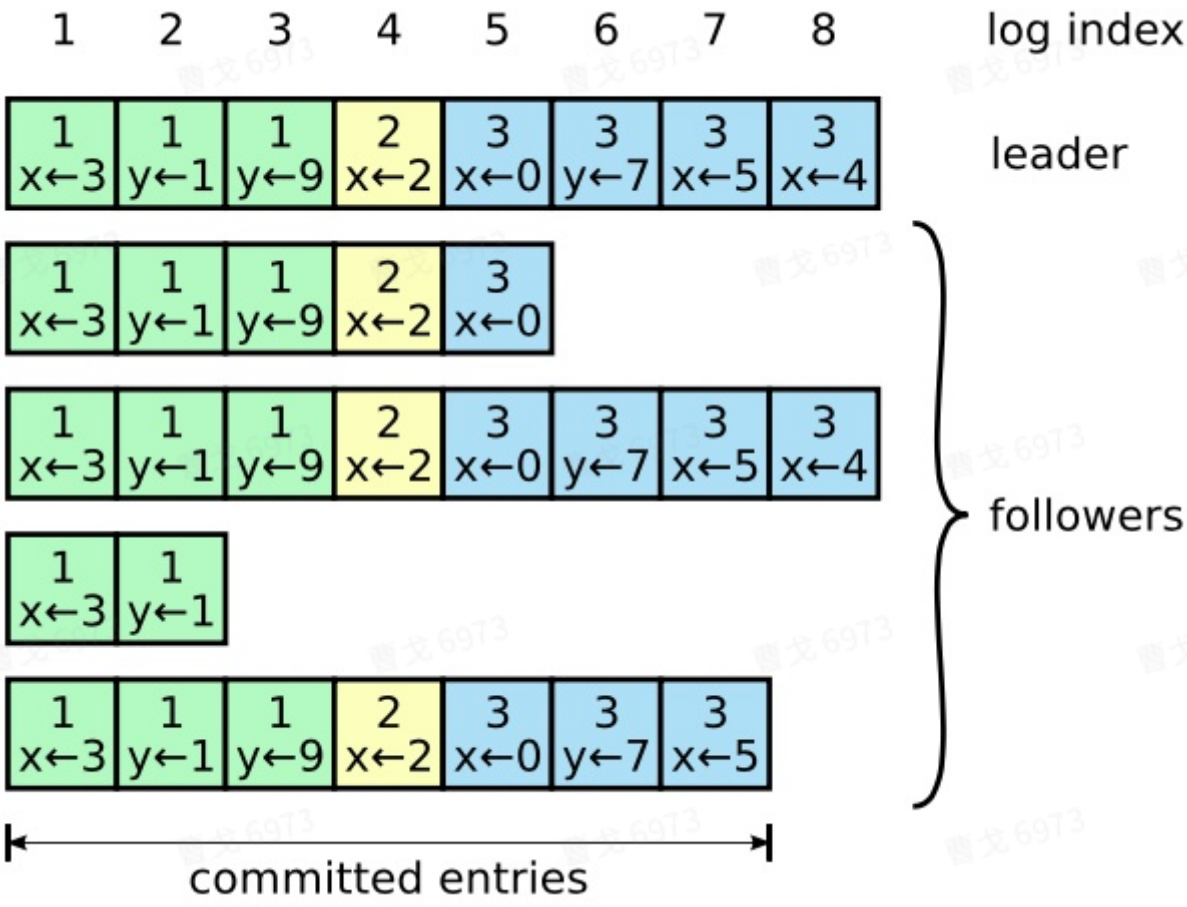


图-日志组成

日志中包含用户指令，leader Term，以及日志index,用户指令就是用户的operation，和raft本身无关，Term和Index是来给日志增加可比性的依据。通过Term和Index来确定日志是否匹配，能够接收这个append 日志的请求。当leader决定当前日志条目符合状态机安全的规则的话就认为这个日志条

目是已提交的。raft系统要保证这些已提交的日志条目已经被持久化存储了。leader在满足状态机安全之后就会应用自己的状态机，但是follower这时还没有应用自己的状态机，需要等到下次附加日志请求过来时候根据请求参数leaderCommit知晓leader当前的提交位置，然后follower会对leader已经提交且自己有该条日志的情况进行状态机应用。具体的细节见下边状态机应用。

这里我们再理解下日志机制的两个重要特性:不同日志中的两个日志条目拥有相同的索引和任期号，那么:

a. 存储了相同的指令

这个是基于leader日志只追加，不覆盖和删除的特性保证的

b. 之前所有的日志条目也全部相同

这个是在附加日志请求的时候有两个参数prevLogTerm，prevLogIndex,用来确保在当前日志被接受之前必须满足紧邻着的之前的日志也必须匹配，那这个特性用归纳法就比较好理解了，日志为空的时候，添加一条日志，这时候从开始到当前位置的日志在不同的节点上的日志是一致的，再添加一条，满足之前的日志相同，当前日志也相同，同理第n条日志，前n-1条日志一定相同，当前日志也相同。

在正常情况下，这种日志复制的一致性检查不会失败，但是在leader异常，follower异常的情况下，例如系统崩溃等会出现不一致的状态，而且是各种的不一致，这个就是raft中比较复杂晦涩的地方之一，也是raft的强大之处，它能保证无论是正常系统还是异常系统，只要满足>=majority的服务器能正常工作的情况下依然保持系统的可用性和强一致性。下边是各种情况，大概有六种:

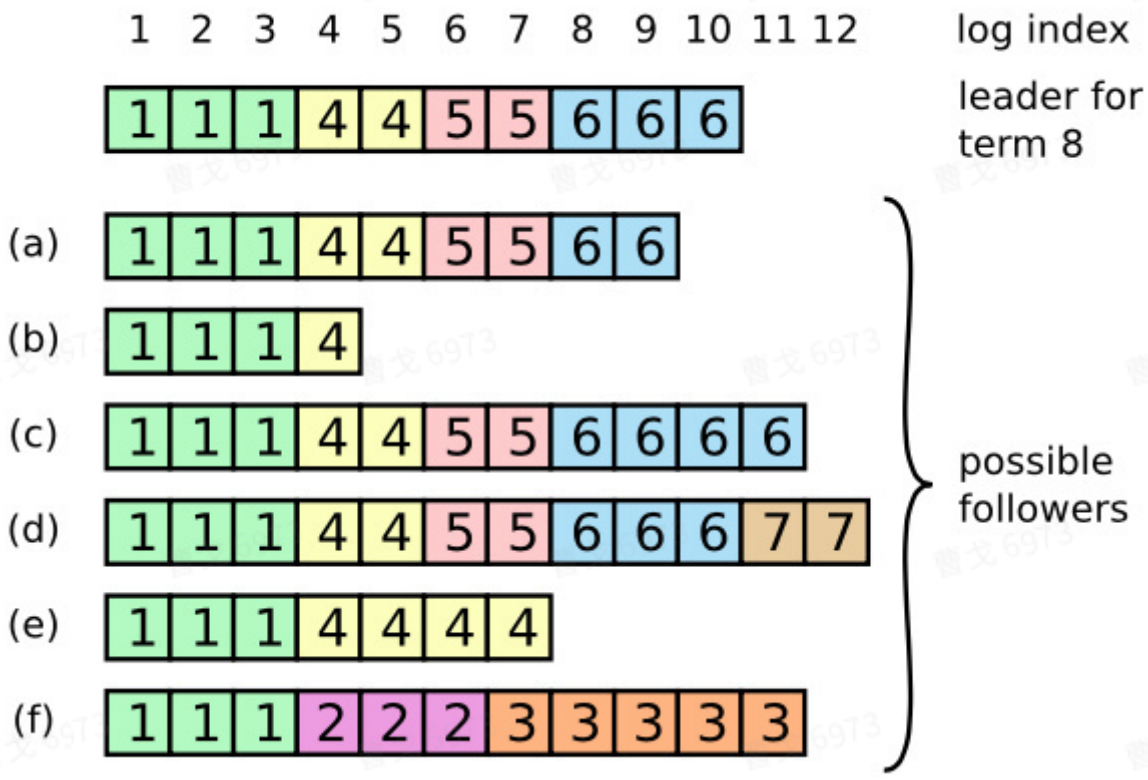


图-leader当选之后follower的可能情况

i. (a-b),缺leader的日志

ii. (c-d),有一些未提交的日志即leader没有的。

iii. (e-f),两种情况都存在

缺日志的比较好理解，可能是当时该节点不在线。多出的日志可能是当时该节点还没来得及提交就离线了，可能是当时的leader离线了，也可能是它自己离线了，然后上线之后leader已经变了。

raft的策略比较简单但非常有效，follower严格对齐leader，leader的日志就是权威日志，这样就极大的简化了这种状态不一致的处理逻辑，而且也使得在工程更利于实现更高效的raft系统。进一步具体的实现对应的就是附加日志RPC的一致性检查逻辑了，主要是通过nextIndex来和follower进行协调，follower会不断的回溯自己的日志直到和leader的日志匹配或者已经到自己日志的开头，这个过程每一次回溯就是一次RPC，leader和follower的协调是通过附加RPC的失败来确认的，leader发现RPC中的返回值success是false的话就会尝试一步步把nextIndex减少，直到RPC返回success=true,然后才会从netIndex的位置直接复制到当前已经提交的日志给该follower，同时，follower会把leader附加过来的日志之后的日志给删除掉，这样日志就一致了。例如a-f中的多余日志就会被覆盖或者删除。

follower离线的相对更简单一点，它在上线时收到相同的附加RPC请求，而且这个日志附加操作是幂等的，所以很容易通过让leader重试的方式完成follower的同步。

上述方式没有给leader增加额外的复杂性，这个是相比paxos减少复杂状态的地方之一。也是raft比较好理解和容易(相对paxos)工程化的地方。

3. 状态机应用

状态机应用这里主要说的是leader的状态机应用，leader应用状态机的时候会将之前的所有未提交的日志都会被提交，包括其它leader创建的条目，这种情况其实是符合状态机安全的，但是不能独立的去提交旧的Term的日志，具体我们分析下场景，如下图:

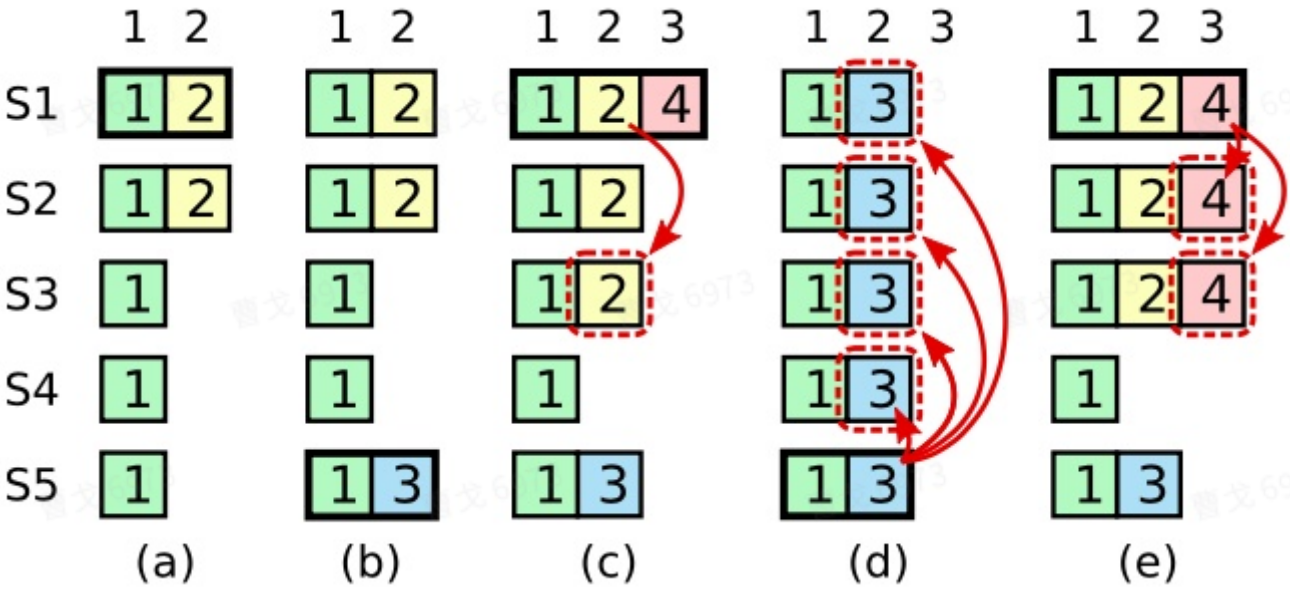
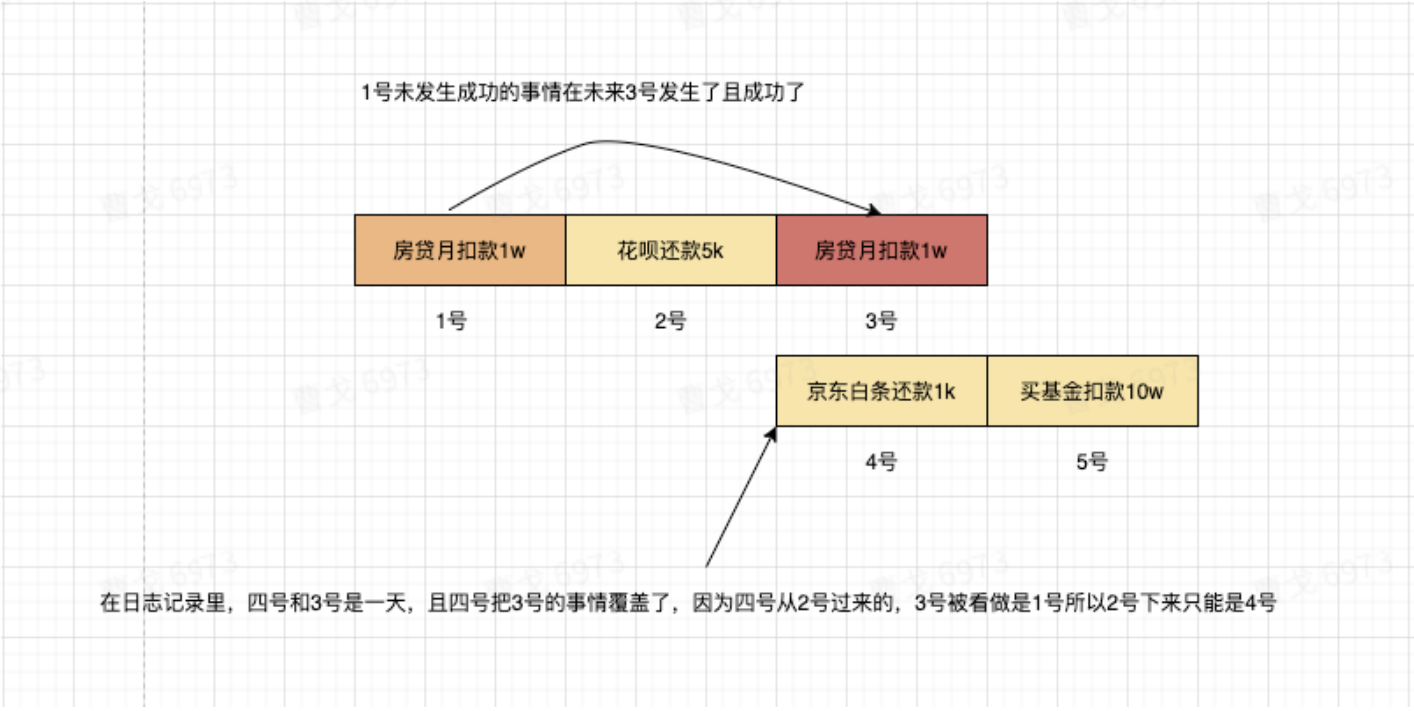


图-新leader对旧leader的日志无法提交的问题

图中日志框框中的数字代表的是Term。我们把(a)->(b)->(c)-(d)看做是一个时序图，时间从前到后。如果leader在提交日志之前崩溃了，未来后续的leader会继续尝试复制这条日志记录，当然前提是该leader本地是有这个未提交的日志的，但是，leader不能确定一个之前Term里的日志条目被保存到大多数服务器上之后就一定已经提交了，所以这里是状态机不安全的。为什么，因为一条已经被存储到>=majority节点上的日志条目有可能依然会被未来的leader覆盖掉，如上图。在(a)中，S1是leader,复制了索引位置为2的日志条目。在(b)中，S1离线，然后S5在Term=3中通过S3,S4和自己的投票成为新的leader，然后从客户端收到了一条心的不一样的日志条目放在了索引为2的地方。(c)中，S5离线，S1重新成为新的leader，然后复制旧日志即Term=2的日志。(d)中，S1离线，S5成为leader,然后S5复制Term=3的日志，并完尝试更新S1老领导留下的痕迹。(e)中，不幸的是S5还没提交日志就离线了，而且S1当选leader,且S5的日志在S1没有复制成功，这个时候S1继续进行日志复制，Term=4的日志进行复制，并成功进行提交，这个时候S1的日志是1，2，4，S2是1，2，4，S3是1，2，4,S4是1，但是上一任leaderS5已经提交成功日志，S5,S4,S3是1,3,只是S3被现任S1覆盖了，因为Term=4>Term=3,这样就出现问题了，S5已经提交的日志被覆盖了，这个就是不一致，怎么解决的，一个简单但高效的方法是，不单独提交旧的日志，而是和新的日志一并提交，为什么这样做会有效果呢，因为上述出现的根本原因在S3上，S3因为Term=3<Term=4被强制覆盖掉了，但是如果是和当前的Term一并提交的话，S1就不会再(e)中当前leader，为什么，因为按照leader必须含有所有已经提交日志的原则，S1不可能当选，因为在投票的时候S1的最新提交的日志的Term没有（d）中S5的Term大。换个角度理解就是,你不能违背当前Term去对日志做不是当前Term的事情，因为Term是一个逻辑时钟，提交旧日志，旧日志的Term又不能变，相当于现在的事情在未来发生了，大概率是要出问题的。可以理解为，借助当前leader的Term避免提交更小的Term，从而避免违背日志的逻辑时钟--Term。下边我用一个图来再次解释一下:



v. **Q:raft如何确保state machine replication的强一致性的**

1. 日志只从leader复制出去，无歧义

2. leader在追加日志条目的RPC调用入参中有两个相关参数，分别是prevLogIndex和prevLogTerm,分别表示新追加的日志的前一个日志的index和它的term,接收者会利用这两个参数做对比，如果和自己本地的log条目匹配，则同意本次日志条目的追加，并RPC返回success,否则返回false.不予以追加。
3. 接收者严格对齐leader的日志，有冲突的地方，以leader的为准，覆盖自己的日志。
4. leader的日志只追加，不覆盖和删除

vi. **Q:raft的哪些特性使得既能保证共识又比paxos更容易理解或简洁的地方**

下边是我理解的raft安全性机制，同时也是比paxos更简洁的地方

1. **选举安全特性**

   一个任期号内，最多只会有一个leader被选举出来。paxos在理论上存在活锁。
2. **leader只附加原则**

   leader不能删除，覆盖自己的日志，只能追加。paxos不同，它允许日志有空洞，增加了日志同步的复杂性。
3. **日志匹配原则**

   如果两个日志在相同的索引位置的日志条目的任期号相同，则认为这个日志从开始到这个索引位置之间都是一致的。paxos没有限制。--raft通过增强自身的一致性约束降低状态的复杂性
4. **leader完全特性**

   如果某个日志条目在某个任期号中已经被提交了，那该条目必须出现在更大任期号的所有leader中。paxos没限制。

vii. **Q:raft和paxos有哪些共通点**

1. **状态机安全特性**

   如果leader已经将给定的索引位置的日志条目应用到状态机了，那其它任何节点在这个索引位置不会出现不同的日志。这一点paxos和raft相同，

viii. **Q:有哪些具体的规则**

1. 所有服务器
   a. 如果commitIndex > lastApplied,则更新lastApplied+=1，并log[lastApplied]应用到状态机；这个可能的原因是机器重启，lastApplied是易失性状态，commitIndex如果是非leader的话，是通过附加日志RPC中的入参leaderCommit进行更新的。
   b. 如果接收到的RPC请求或者响应中，任期号T>currentTerm,就切换Wiefollower状态。
2. Follower
   a. 响应candidate，leader的请求
   b. 在选举超时之前没有收到当前leader的心跳，附加日志，或者candidate的投票请求，就切换为candidate状态。
3. Candidate

a. 自增当前的任期号
b. 投自己一票
c. 重置选举超时器
d. 发送投票请求RPC给其它服务器
e. 如果收到大多数选票，切换状态为leader
f. 收到新leader的附加日志RPC，转变成follower
g. 如果选举超时，重新发起选举RPC

4. leader
   a. 发送附加日志为空内容的RPC作为心跳给其它服务器，如果持续空闲超过一定时间之后再次发送心跳日志给其它服务器，并保持这个心跳发送时机的探测。
   b. 如果收到客户端的请求，附加条目到本地日志中，在条目被应用到状态机之后再响应客户端
   c. 如果对于一个follower，最后的日志条目的索引值大于等于nextIndex,那么，leader就发送从nextIndex开始，之后的所有日志条目给这个follower
      i. 成功：更新该follower对应的nextIndex和matchIndex
      ii. 如果失败，且因为日志不匹配导致的，则减少nextIndex再重试，知道满足为止，满足之后从新从nextIndex开始发送之后所有的日志。

      出现这个的可能原因是follower有一段时间没有参与到现在的leader的附加日志中了，如果失败，则可能的原因是该follower含义当前leader不曾有的Log但是放心，这些Log最终并没有被应用到状态机，否则当前leader页当选不了。
   d. 如果存在一个满足N>commitIndex的N，且超过半数的matchIndex[i]>=N,且log[N].Term = currentTerm,则更新commitIndex = N;出现这个gap的原因可能是该机器重启了，因为commitIndex是个易失性状态。

ix. **Q:在日志追加中，接收者需要做哪些事情**
   1. prevLogIndex检查
   2. 删除与leader冲突的日志及之后的所有日志
   3. 追加日志中不存在的日志
   4. 如果leaderCommit(追加日志RPC入参)大于接收者的commitIndex(易失性)，则commitIndex = min(leaderCommit, 上一个新条目的索引)

x. **Q:raft在日志同步阶段是2pc模式还是多数模式**

采用的是多数模式，但是对于附加日志失败的RPC会不断的重试，直到成功。

xi. **Q:multi-raft怎么设计**

xii. **Q:raft/分布式系统有什么关键问题及对应策略**

1. 选举冲突

   如果在选举的时候多个服务器节点同时发起选举请求，这样会增加选举过程的冲突，而且不利于整体系统的稳定性，因为非leader和follower状态的节点无法提供服务，所以一种简单但非常有效的方案，随机time wait,这样能降低冲突，而且容易理解，实现也简单，

系统复杂性不高，而且在raft的paper中也有实验性的证明该方法的有效性。不过，具体的time wait时间才是最关键的，一般建议是150ms~300ms,中进行随机该值是一个参考值。

2. 时间和可用性

   广播时间(broadcastTime) << 选举超时时间(electionTimeout) << 平故障时间(MTBF)

   broadcastTime指的是从一个服务器并发的发送RPC给集群中的其它服务器并受到响应的平均时间。electionTimeout就是选举时间超时限制。平均故障时间就是对于一台机器而言，两次故障之间的平均时间。容易理解，broadcastTime必须比选举超时时间小一个量级，这种领导人才能够发送稳定的heartbeat来阻止跟随者开始进入选举状态，同时选举RPC也才有成功的可能性，否则选举会无限重试，更容易形成活锁。平均故障时间就比选举超时时间更大了，可能是天，月，甚至年的级别。只有满足上述时间约束的关系，才能让raft系统稳定运行。leader离线后，在正是上线新leader之前，整个系统(单一raft)是不可用的。

   更具体的，raft的RPC延迟大概在0.5ms~20ms,整个和公司的网络架构及存储技术有关系，内网往往会非常快，基本都在<0.5ms,rpc加上rpc自身的损耗，基本上在10ms内能完成一次RPC请求。所以选举超时可以设定在10ms~500ms,如果担心网络抖动的问题，可以设定的更长一些，但是过长会增加失败探测的代价，如果真的是leader离线了，那么在选举超时时间+random wait +选举耗时=1s+200ms+20ms，差不多有1s多的时间会服务不可用。

   该问题也不是不能解决:

   a. 根据自身网络架构及服务的SLA保障，可制定一个符合自己实际情况的选举超时时间。
   b. 采用multi-raft,单一raft从系统维度看，还是一个单点系统，所以multi-raft提高整个系统的可用性。
   c. 建议选取150ms~300ms。

xiii. **Q:raft和paxos对比有哪些区别**

1. 和basic对比--子问题相对独立，更清晰易理解，易实现
   a. 模块分工明确，效率更高：先进行Leader选举，之后统一由该leader负责日志同步及客户端读写应答，在内部通信次数上相比paxos每次先prepare,再等待accepted少了一次。
   b. 状态机更简洁--减少状态数量
      i. leader的当选必须满足有之前所有已提交的日志
      ii. 不存在同时有多个提议被通过，也就不存在paxos中需要保持提议内容和比当前提议小的提议中议案编号最大的内容一致，paxos之所以这么做是因为它要保证它同意当前议案的同时不能违背它之前对其它已经被它投过票的提议，所以要保持相同议案号上的内容和之前议案编号最大的内容一致。(我在想，为啥不直接改成，相同议案号中最早的那个议案为准呢)

iii. 日志只能从leader复制

c. 随机化方法减少leader选举的冲突

2. 和multi-paxos对比

<暂留空白,后续补充>

xiv. **有哪些存储项目，包括企业解决方案的存储项目中，有哪些是基于raft算法**

1. etcd
2. bytestore

b. **模块组成及工作原理**

i. **状态机**

就是工作流程中的状态机应用，其实是对业务的一个承诺，状态机里的东西绝不会凭空消失。具体的描述的是日志应用状态不是系统本身。

ii. **选举**

细节见上述工作流程中的leader选举描述。

iii. **日志复制**

同上。

c. **集群成员变化**

集群的动态扩缩容总是需要满足的，只是在扩缩容的过程中需要注意raft的稳定性，为了避免新旧配置同时生效的问题，需要在添加新节点的时候一次只能生效一个。具体是什么样的问题及怎么高效的应对的我们来具体分析下。

i. 成员变更的问题

假设我们有一个由A,B,C节点组成的raft集群，现在我们需要增加副本数，增加2个副本，扩展为A,B,C,D,E 5个节点:

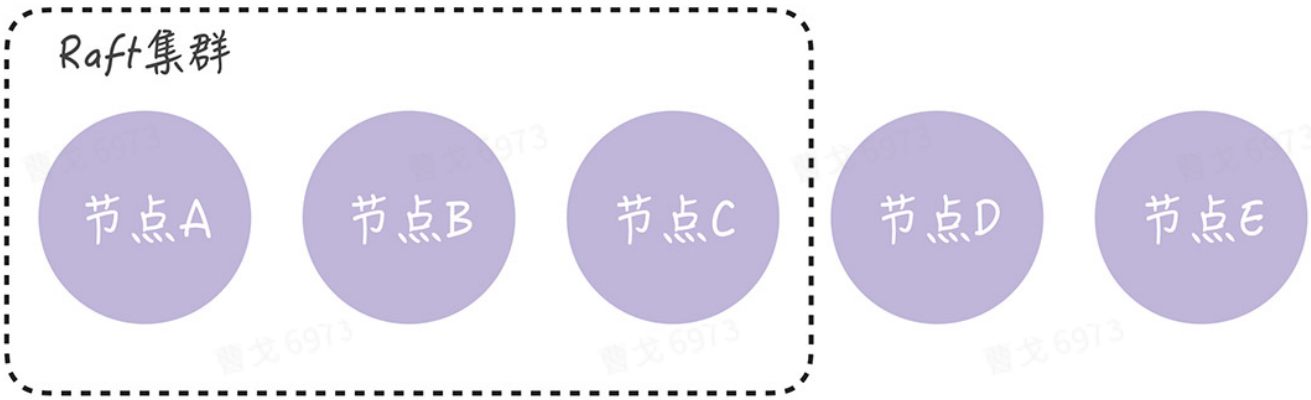


图-配置变更前raft系统节点示意图(图片来自极客时间)

如果在成员变更是，节点A,B和C发生了网络分区，节点A,B组成旧配置的majority,那么下图中的A依然能维持leader的身份对外提供服务，另一方面，节点C和D,E组成了新配置的大多数，它们可能会选举出新的leader，这个时候就出现了两个leader:

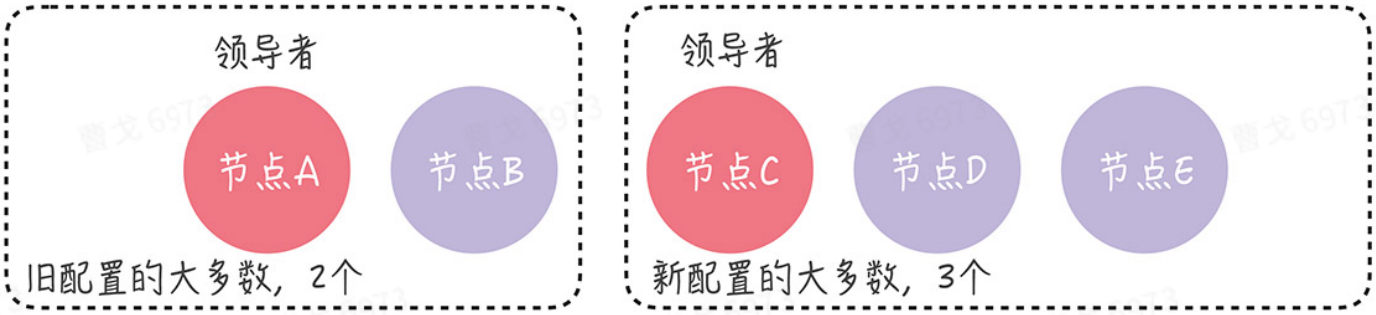
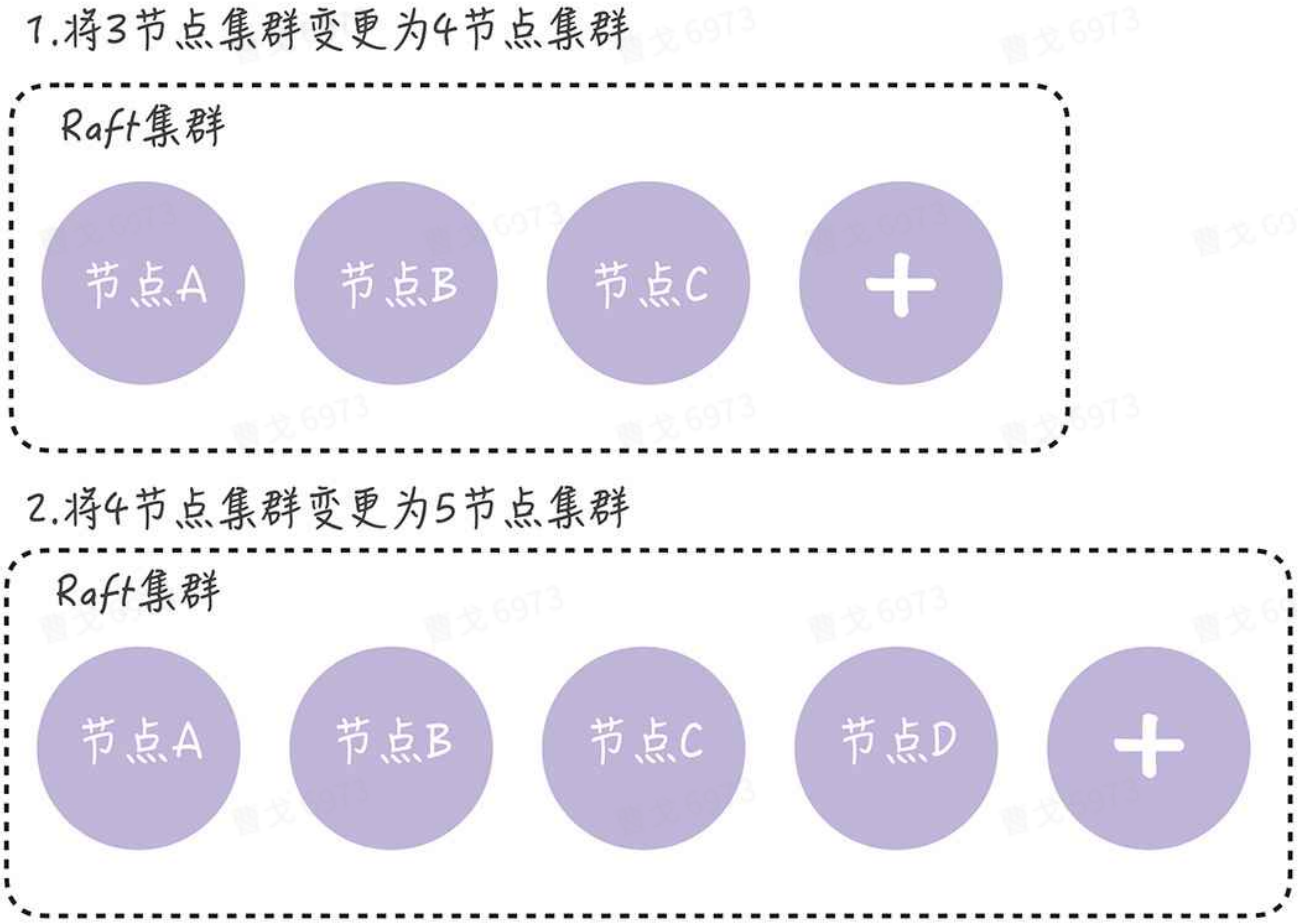


图-同时存在新旧配置的leader(图片来自极客时间)

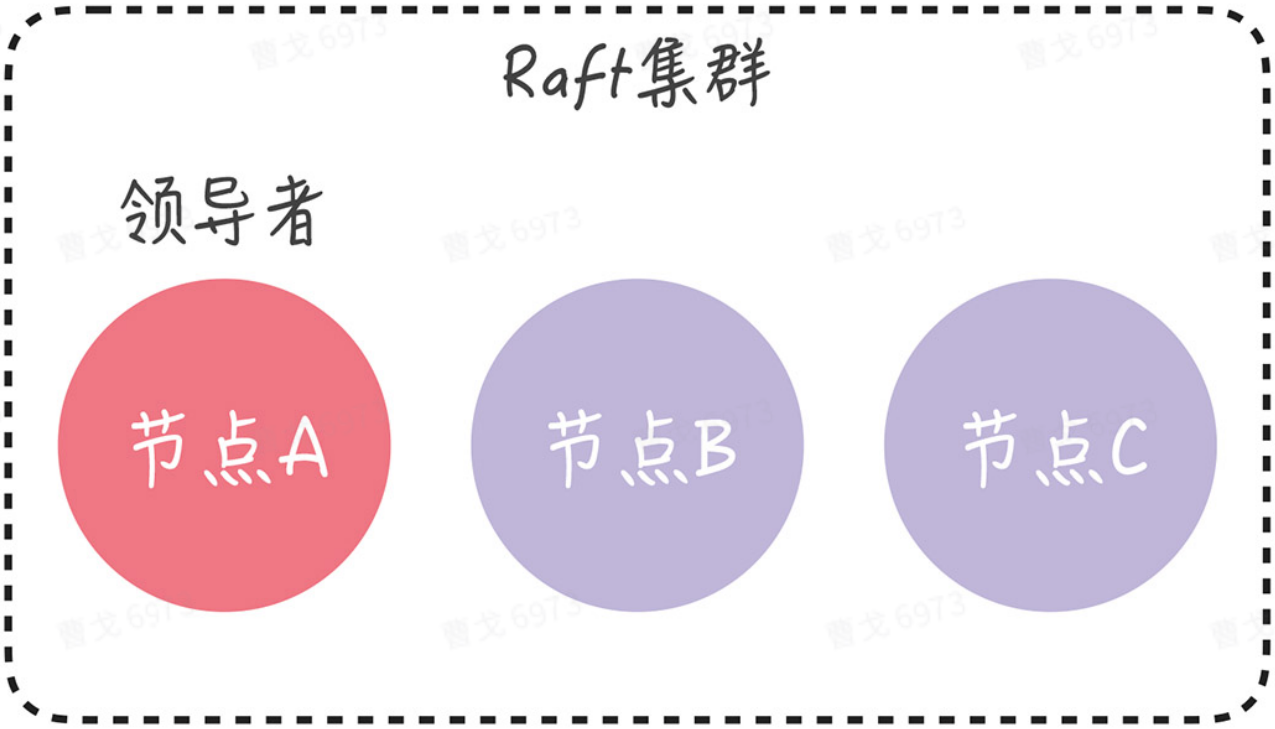
那么接下来看看我们怎么解决这个问题的。

ii. 如何通过单节点变更(single-server changes)解决问题

一次变更一个节点，比如3节点先变更为4节点再变更为5节点。



我们假设A是leader，具体步骤如下：



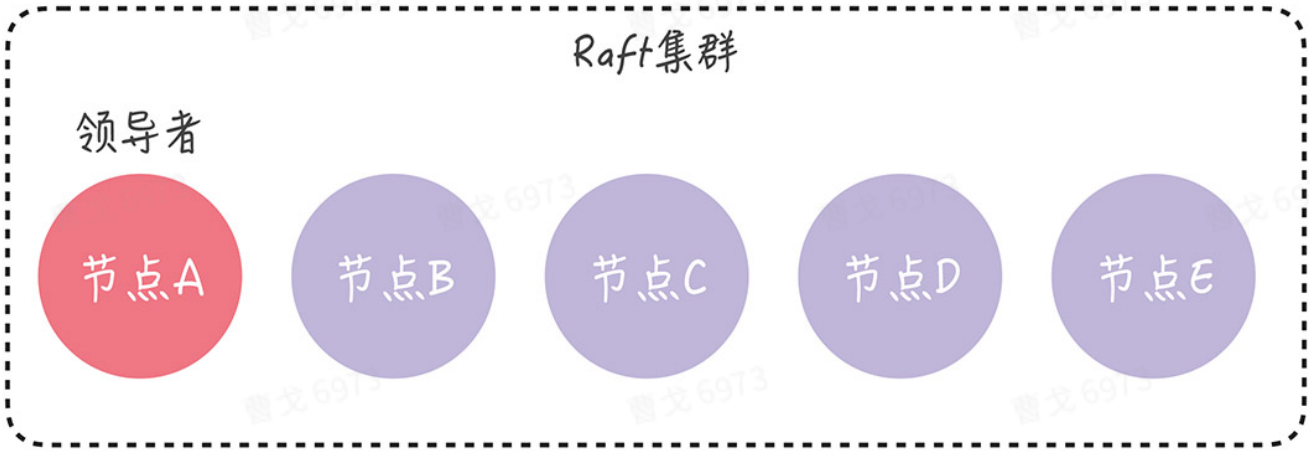
第一步，A同步数据给D

第二步，A将配置[A,B,C,D]当做一个日志条目进行日志复制，然后配置通过状态机应用得到更新:

继续，现在配置是[A,B,C,D]:

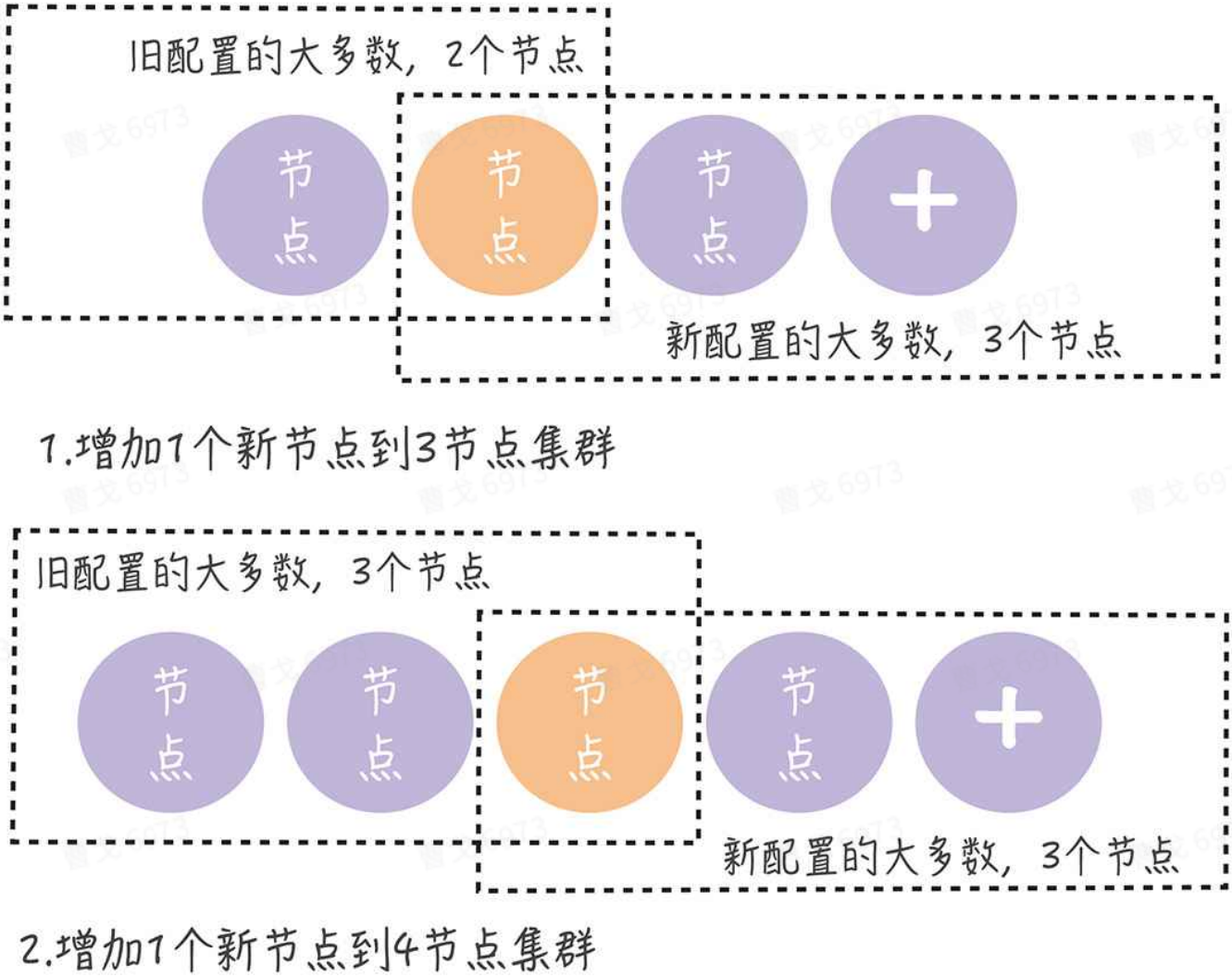
第一步，A同步数据给E

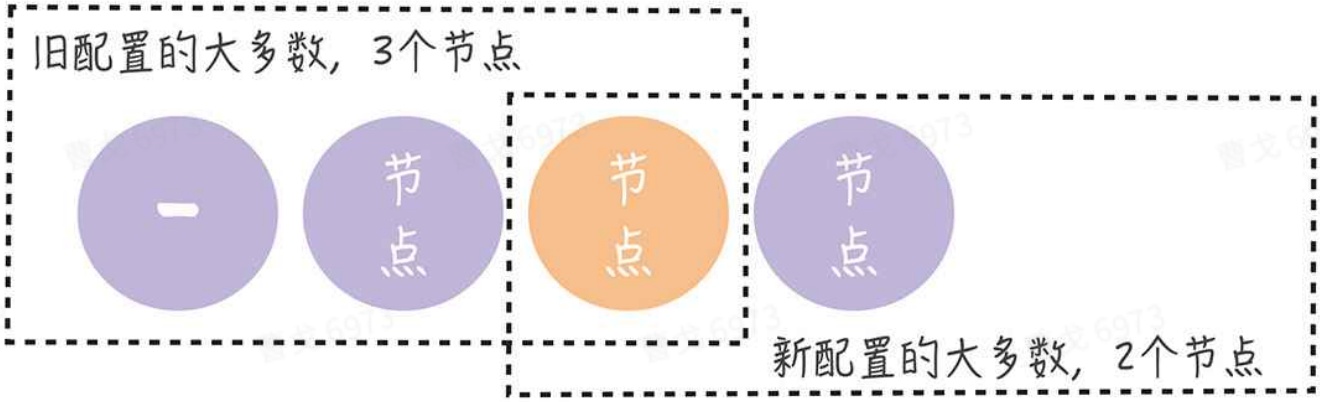
第二步，A将新配置[A,B,C,D,E]应用到日志状态机，新配置生效：



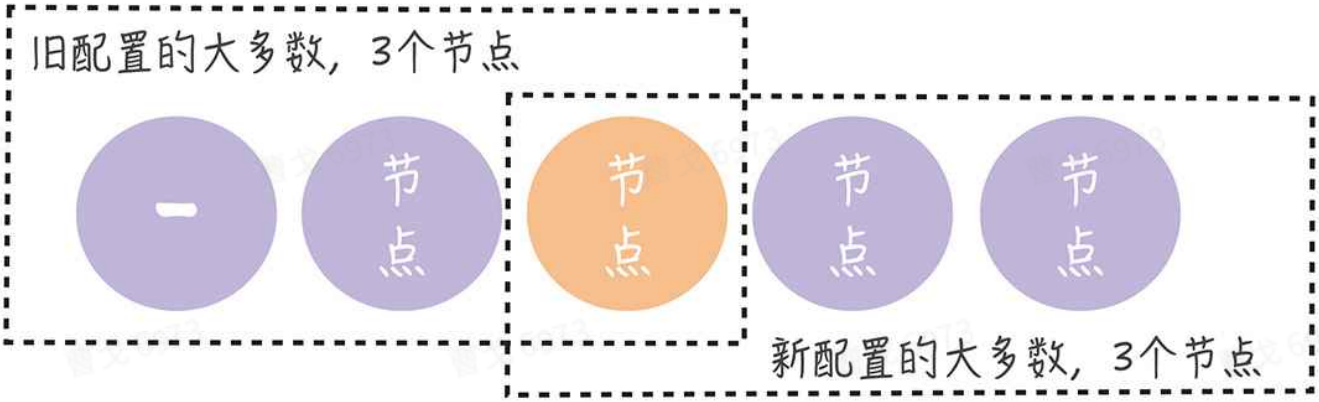
这样，完成了成员变更。

在正常情况下，不管旧的配置怎么组成的，就的配置和新的配置的“大多数”都会有一个重叠，具体如下：





3.从4节点集群中移除1个节点



4.从5节点集群中移除1个节点

不管集群是偶数节点，还是奇数节点，不管是增加节点，还是移除节点，新旧配置的“大多数”都会存在重叠（图中的橙色节点）。

需要注意的是，在分区错误、节点故障等情况下，如果我们并发执行单节点变更，那么就可能出现一次单节点变更尚未完成，新的单节点变更又在执行，导致集群出现 2 个领导者的情况。

d. 日志压缩

<暂留空白，下一轮学习实现>

e. 日志恢复

<暂留空白，下一轮学习实现>

f. leader转移

<暂留空白，下一轮学习实现>

g. 读写请求处理及优化

<暂留空白，待实践>