一文搞懂Raft算法 - xybaby

正文

raft是工程上使用较为广泛的强一致性、去中心化、高可用的分布式协议。在这里强调了是在工程上，因为在学术理论界，最耀眼的还是大名鼎鼎的Paxos。但Paxos是：少数真正理解的人觉得简单，尚未理解的人觉得很难，大多数人都是一知半解。本人也花了很多时间、看了很多材料也没有真正理解。直到看到raft的论文，两位研究者也提到，他们也花了很长的时间来理解Paxos，他们也觉得很难理解，于是研究出了raft算法。

raft是一个共识算法（consensus algorithm），所谓共识，就是多个节点对某个事情达成一致的看法，即使是在部分节点故障、网络延时、网络分割的情况下。这些年最为火热的加密货币（比特币、区块链）就需要共识算法，而在分布式系统中，共识算法更多用于提高系统的容错性，比如分布式存储中的复制集（replication），在带着问题学习分布式系统之中心化复制集一文中介绍了中心化复制集的相关知识。raft协议就是一种leader-based的共识算法，与之相应的是leaderless的共识算法。

本文基于论文In Search of an Understandable Consensus Algorithm对raft协议进行分析，当然，还是建议读者直接看论文。

本文地址：https://www.cnblogs.com/xybaby/p/10124083.html

Raft算法的头号目标就是容易理解（UnderStandable），这从论文的标题就可以看出来。当然，Raft增强了可理解性，在性能、可靠性、可用性方面是不输于Paxos的。

> Raft more understandable than Paxos and also provides a better foundation for building practical systems

为了达到易于理解的目标，raft做了很多努力，其中最主要是两件事情：

- 问题分解
- 状态简化

问题分解是将"复制集中节点一致性"这个复杂的问题划分为数个可以被独立解释、理解、解决的子问题。在raft，子问题包括，*leader election*， *log replication*，*safety*，*membership changes*。而状态简化更好理解，就是对算法做出一些限制，减少需要考虑的状态数，使得算法更加清晰，更少的不确定性（比如，保证新选举出来的leader会包含所有commited log entry）

> Raft implements consensus by first electing a distinguished leader, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

上面的引文对raft协议的工作原理进行了高度的概括：raft会先选举出leader，leader完全负责replicated log的管理。leader负责接受所有客户端更新请求，然后复制到follower节点，并在“安全”的时候执行这些请求。如果leader故障，followes会重新选举出新的leader。

这就涉及到raft最新的两个子问题： leader election和log replication

raft协议中，一个节点任一时刻处于以下三个状态之一：

- leader
- follower
- candidate

给出状态转移图能很直观的直到这三个状态的区别

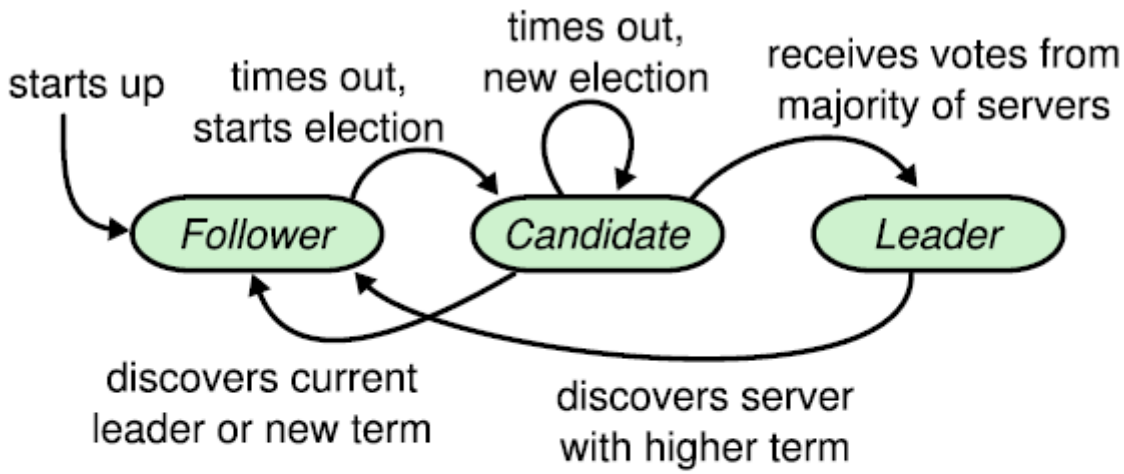


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

可以看出所有节点启动时都是follower状态；在一段时间内如果没有收到来自leader的心跳，从follower切换到candidate，发起选举；如果收到majority的造成票（含自己的一票）则切换到leader状态；如果发现其他节点比自己更新，则主动切换到follower。

总之，系统中最多只有一个leader，如果在一段时间里发现没有leader，则大家通过选举-投票选出leader。leader会不停的给follower发心跳消息，表明自己的存活状态。如果leader故障，那么follower会转换成candidate，重新选出leader。

term

从上面可以看出，哪个节点做leader是大家投票选举出来的，每个leader工作一段时间，然后选出新的leader继续负责。这根民主社会的选举很像，每一届新的履职期称之为一届任期，在raft协议中，也是这样的，对应的术语叫*term*。

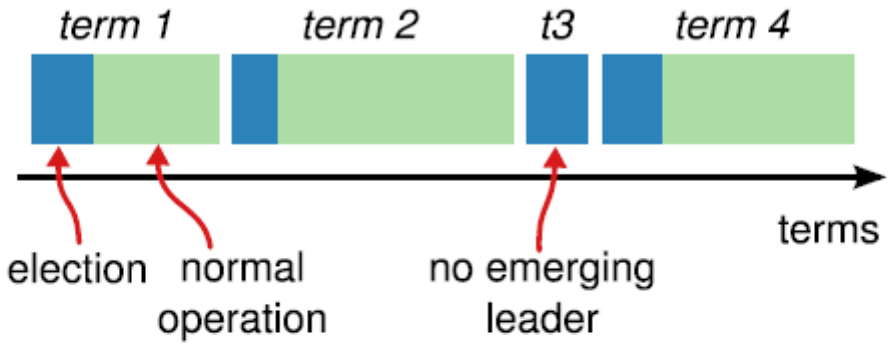


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

term（任期）以选举（election）开始，然后就是一段或长或短的稳定工作期（normal Operation）。从上图可以看到，任期是递增的，这就充当了逻辑时钟的作用；另外，term 3展示了一种情况，就是说没有选举出leader就结束了，然后会发起新的选举，后面会解释这种*split vote*的情况。

选举过程详解

上面已经说过，如果follower在*election timeout*内没有收到来自leader的心跳，（也许此时还没有选出leader，大家都在等；也许leader挂了；也许只是leader与该follower之间网络故障），则会主动发起选举。步骤如下：

- 增加节点本地的 *current term* ，切换到candidate状态
- 投自己一票
- 并行给其他节点发送 *RequestVote RPCs*
- 等待其他节点的回复

在这个过程中，根据来自其他节点的消息，可能出现三种结果

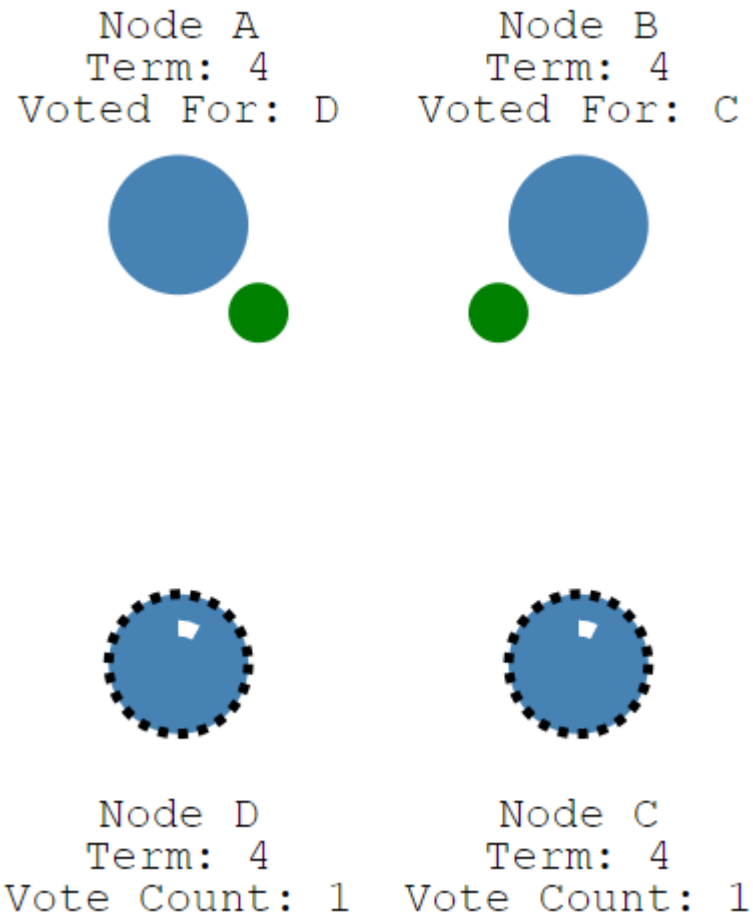
1. 收到majority的投票（含自己的一票），则赢得选举，成为leader
2. 被告知别人已当选，那么自行切换到follower
3. 一段时间内没有收到majority投票，则保持candidate状态，重新发出选举

第一种情况，赢得了选举之后，新的leader会立刻给所有节点发消息，广而告之，避免其余节点触发新的选举。在这里，先回到投票者的视角，投票者如何决定是否给一个选举请求投票呢，有以下约束：

- 在任一任期内，单个节点最多只能投一票
- 候选人知道的信息不能比自己的少（这一部分，后面介绍log replication和safety的时候会详细介绍）
- first-come-first-served 先来先得

第二种情况，比如有三个节点A B C。A B同时发起选举，而A的选举消息先到达C，C给A投了一票，当B的消息到达C时，已经不能满足上面提到的第一个约束，即C不会给B投票，而A和B显然都不会给对方投票。A胜出之后，会给B,C发心跳消息，节点B发现节点A的term不低于自己的term，知道有已经有Leader了，于是转换成follower。

第三种情况，没有任何节点获得majority投票，比如下图这种情况：



总共有四个节点，Node C、Node D同时成为了candidate，进入了term 4，但Node A投了NodeD一票，NodeB投了Node C一票，这就出现了平票 split vote的情况。这个时候大家都在等啊等，直到超时后重新发起选举。如果出现平票的情况，那么就延长了系统不可用的时间（没有leader是不能处理客户端写请求的），因此raft引入了randomized election timeouts来尽量避免平票情况。同时，leader-based 共识算法中，节点的数目都是奇数个，尽量保证majority的出现。

当有了leader，系统应该进入对外工作期了。客户端的一切请求来发送到leader，leader来调度这些并发请求的顺序，并且保证leader与followers状态的一致性。raft中的做法是，将这些请求以及执行顺序告知followers。leader和followers以相同的顺序来执行这些请求，保证状态一致。

Replicated state machines

共识算法的实现一般是基于复制状态机（Replicated state machines），何为复制状态机：

> If two identical, **deterministic** processes begin in the same state and get the same inputs in the same order, they will produce the same output and end in the same state.

简单来说：**相同的初识状态 + 相同的输入 = 相同的结束状态**。引文中有一个很重要的词`deterministic`，就是说不同节点要以相同且确定性的函数来处理输入，而不要引入一下不确定的值，比如本地时间等。如何保证所有节点`get the same inputs in the same order`，使用replicated log是一个很不错的注意，log具有持久化、保序的特点，是大多数分布式系统的基石。

因此，可以这么说，在raft中，leader将客户端请求（command）封装到一个个log entry，将这些log entries复制（replicate）到所有follower节点，然后大家按相同顺序应用（apply）log entry中的command，则状态肯定是一致的。

下图形象展示了这种log-based replicated state machine

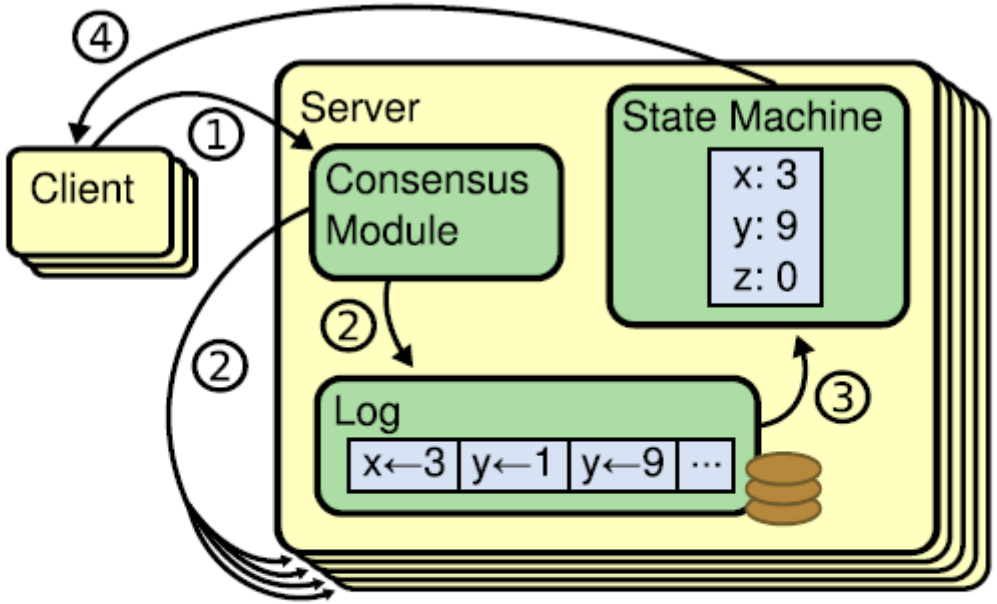


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

请求完整流程

当系统（leader）收到一个来自客户端的写请求，到返回给客户端，整个过程从leader的视角来看会经历以下步骤：

- leader append log entry
- leader issue AppendEntries RPC in parallel
- leader wait for majority response
- leader apply entry to state machine
- leader reply to client

- leader notify follower apply log

可以看到日志的提交过程有点类似两阶段提交(2PC)，不过与2PC的区别在于，leader只需要大多数（majority）节点的回复即可，这样只要超过一半节点处于工作状态则系统就是可用的。

那么日志在每个节点上是什么样子的呢

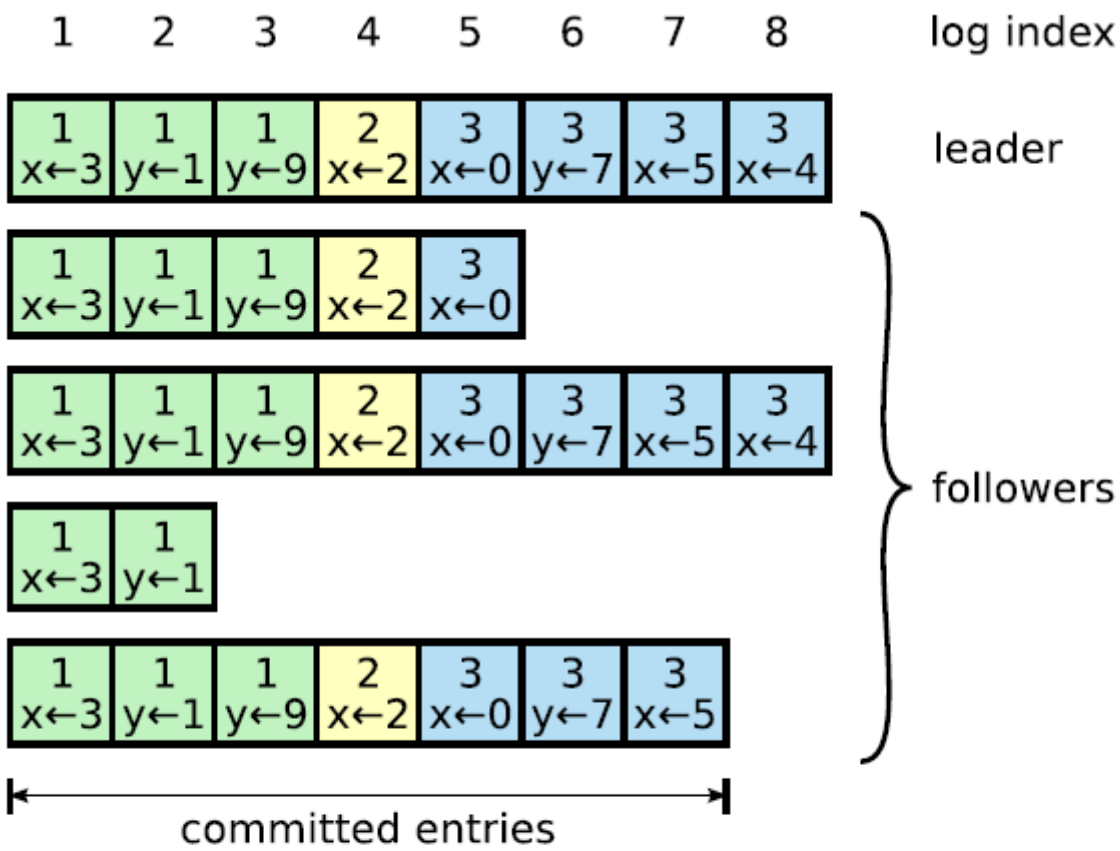


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

不难看到，logs由顺序编号的log entry组成，每个log entry除了包含command，还包含产生该log entry时的leader term。从上图可以看到，五个节点的日志并不完全一致，raft算法为了保证高可用，并不是强一致性，而是最终一致性，leader会不断尝试给follower发log entries，直到所有节点的log entries都相同。

在上面的流程中，leader只需要日志被复制到大多数节点即可向客户端返回，一旦向客户端返回成功消息，那么系统就必须保证log（其实是log所包含的command）在任何异常的情况下都不会发生回滚。这里有两个词：commit（committed），apply(applied)，前者是指日志被复制到了大多数节点后日志的状态；而后者则是节点将日志应用到状态机，真正影响到节点状态。

> The leader decides when it is safe to apply a log entry to the state machines; such an entry is called committed. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines. A log entry is committed once the leader that created the entry has replicated it on a majority of the servers

在上面提到只要日志被复制到majority节点，就能保证不会被回滚，即使在各种异常情况下，这根leader election提到的选举约束有关。在这一部分，主要讨论raft协议在各种各样的异常情况下如何工作的。

衡量一个分布式算法，有许多属性，如

- safety：nothing bad happens,
- liveness： something good eventually happens.

在任何系统模型下，都需要满足safety属性，即在任何情况下，系统都不能出现不可逆的错误，也不能向客户端返回错误的内容。比如，raft保证被复制到大多数节点的日志不会被回滚，那么就是safety属性。而raft最终会让所有节点状态一致，这属于liveness属性。

raft协议会保证以下属性

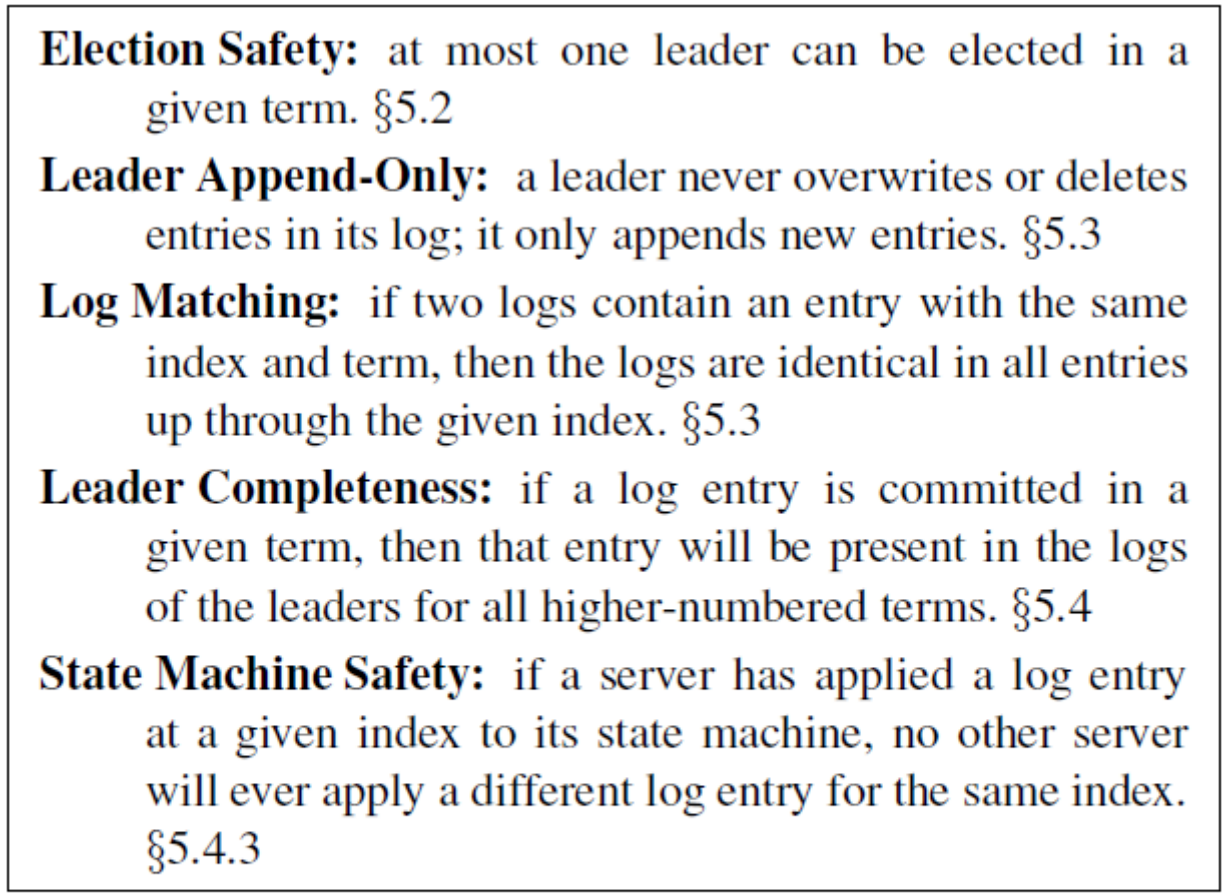


Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

Election safety

选举安全性，即任一任期内最多一个leader被选出。这一点非常重要，在一个复制集中任何时刻只能有一个leader。系统中同时有多余一个leader，被称之为脑裂（brain split），这是非常严重的问题，会导致数据的覆盖丢失。在raft中，两点保证了这个属性：

- 一个节点某一任期内最多只能投一票；

- 只有获得majority投票的节点才会成为leader。

因此，**某一任期内一定只有一个leader**。

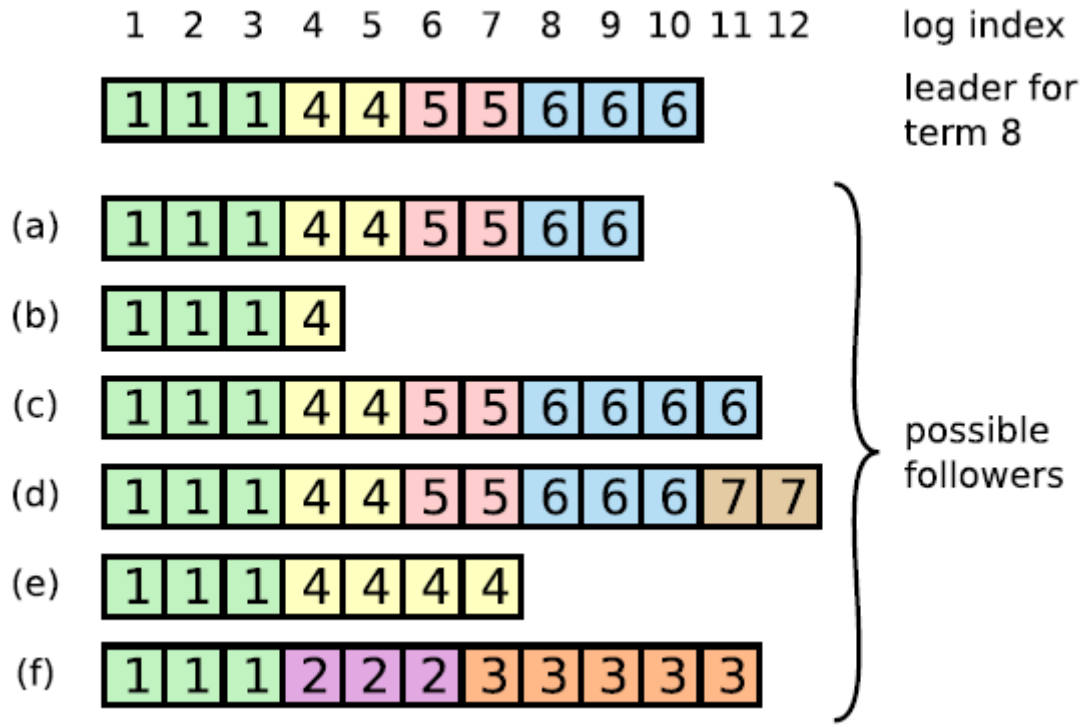
log matching

很有意思，log匹配特性， 就是说如果两个节点上的某个log entry的log index相同且term相同，那么在该index之前的所有log entry应该都是相同的。如何做到的？依赖于以下两点

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

首先，leader在某一term的任一位置只会创建一个log entry，且log entry是append-only。其次，consistency check。leader在AppendEntries中包含最新log entry之前的一个log 的term和index，如果follower在对应的term index找不到日志，那么就会告知leader不一致。

在没有异常的情况下，log matching是很容易满足的，但如果出现了node crash，情况就会变得负责。比如下图



注意：上图的a-f不是6个follower，而是某个follower可能存在的六个状态

leader、follower都可能crash，那么follower维护的日志与leader相比可能出现以下情况

- 比leader日志少，如上图中的ab
- 比leader日志多，如上图中的cd
- 某些位置比leader多，某些日志比leader少，如ef（多少是针对某一任期而言）

当出现了leader与follower不一致的情况，leader强制follower复制自己的log

> To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point.

leader会维护一个nextIndex[]数组，记录了leader可以发送每一个follower的log index，初始化为eader最后一个log index加1， 前面也提到，leader选举成功之后会立即给所有follower发送AppendEntries RPC（不包含任何log entry，

也充当心跳消息）,那么流程总结为：

> s1 leader 初始化nextIndex[x]为 leader最后一个log index + 1
> s2 AppendEntries里prevLogTerm prevLogIndex来自 logs[nextIndex[x] - 1]
> s3 如果follower判断prevLogIndex位置的log term不等于prevLogTerm，那么返回 false，否则返回True
> s4 leader收到follower的恢复，如果返回值是True，则nextIndex[x] -= 1, 跳转到s2. 否则
> s5 同步nextIndex[x]后的所有log entries

leader completeness vs elcetion restriction

leader完整性：如果一个log entry在某个任期被提交（committed），那么这条日志一定会出现在所有更高term的leader的日志里面。这个跟leader election、log replication都有关。

- 一个日志被复制到majority节点才算committed
- 一个节点得到majority的投票才能成为leader，而节点A给节点B投票的其中一个前提是，B的日志不能比A的日志旧。下面的引文指处了如何判断日志的新旧

> voter denies its vote if its own log is more up-to-date than that of the candidate.

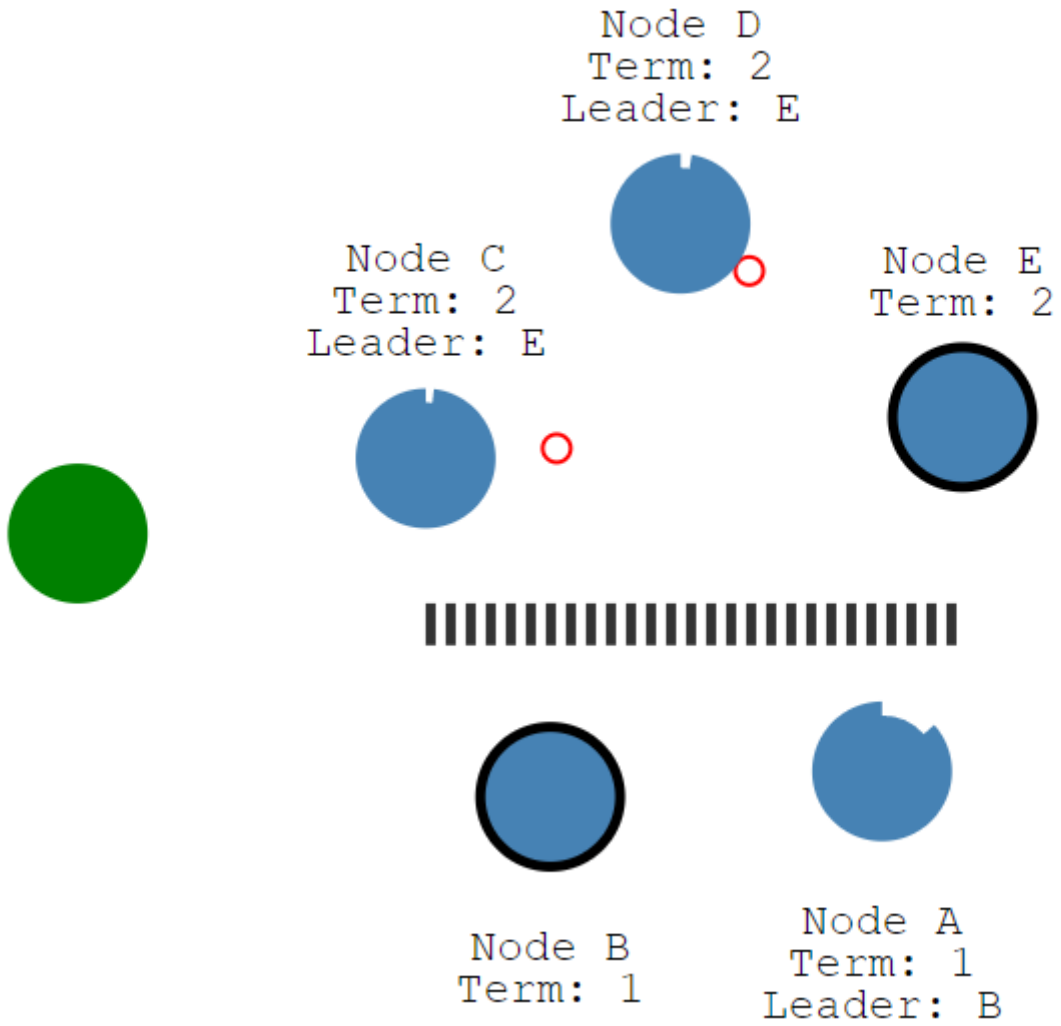
> If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

上面两点都提到了majority：commit majority and vote majority，根据Quorum，这两个majority一定是有重合的，因此被选举出的leader一定包含了最新的committed的日志。

raft与其他协议（Viewstamped Replication、mongodb）不同，raft始终保证leade包含最新的已提交的日志，因此leader不会从follower catchup日志，这也大大简化了系统的复杂度。

stale leader

raft保证Election safety，即一个任期内最多只有一个leader，但在网络分割（network partition）的情况下，**可能会出现两个leader，但两个leader所处的任期是不同的**。如下图所示



系统有5个节点ABCDE组成，在term1，Node B是leader，但Node A、B和Node C、D、E之间出现了网络分割，因此Node C、D、E无法收到来自leader（Node B）的消息，在election time之后，Node C、D、E会分期选举，由于满足majority条件，Node E成为了term 2的leader。因此，在系统中貌似出现了两个leader：term 1的Node B， term 2的Node E, Node B的term更旧，但由于无法与Majority节点通信，NodeB仍然会认为自己是leader。

在这样的情况下，我们来考虑读写。

首先，如果客户端将请求发送到了NodeB，NodeB无法将log entry 复制到majority节点，因此不会告诉客户端写入成功，这就不会有问题。

对于读请求，stale leader可能返回stale data，比如在read-after-write的一致性要求下，客户端写入到了term2任期的leader Node E，但读请求发送到了Node B。如果要保证不返回stale data，leader需要check自己是否过时了，办法就是与大多数节点通信一次，这个可能会出现效率问题。另一种方式是使用lease，但这就会依赖物理时钟。

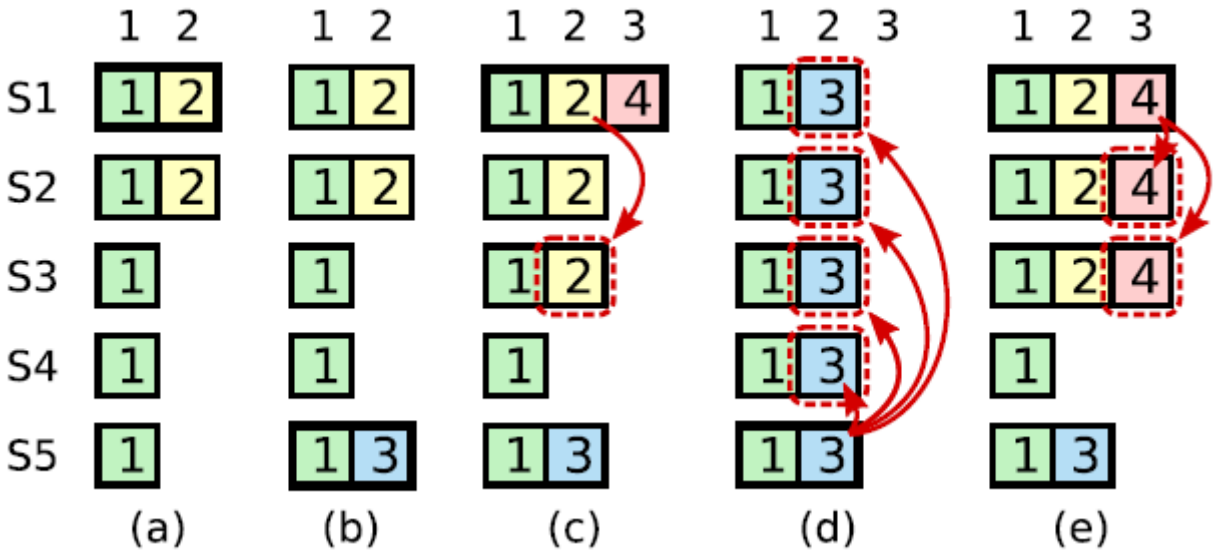
从raft的论文中可以看到，leader转换成follower的条件是收到来自更高term的消息，如果网络分割一直持续，那么stale leader就会一直存在。而在raft的一些实现或者raft-like协议中，leader如果收不到majority节点的消息，那么可以自己step down，自行转换到follower状态。

State Machine Safety

前面在介绍safety的时候有一条属性没有详细介绍，那就是State Machine Safety：

> State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

如果节点将某一位置的log entry应用到了状态机，那么其他节点在同一位置不能应用不同的日志。简单点来说，所有节点在同一位置（index in log entries）应该应用同样的日志。但是似乎有某些情况会违背这个原则：



上图是一个较为复杂的情况。在时刻(a), s1是leader，在term2提交的日志只赋值到了s1 s2两个节点就crash了。在时刻（b), s5成为了term 3的leader，日志只赋值到了s5，然后crash。然后在(c)时刻，s1又成为了term 4的leader，开始赋值日志，于是把term2的日志复制到了s3，此刻，可以看出term2对应的日志已经被复制到了majority，因此是committed，可以被状态机应用。不幸的是，接下来（d）时刻，s1又crash了，s5重新当选，然后将term3的日志复制到所有节点，这就出现了一种奇怪的现象：被复制到大多数节点（或者说可能已经应用）的日志被回滚。

究其根本，是因为term4时的leader s1在（C）时刻提交了之前term2任期的日志。为了杜绝这种情况的发生：

> **Raft never commits log entries from previous terms by counting replicas.**
> Only log entries from the leader’s current term are committed by counting replicas; once an entry from the current term has been committed in this way, then all prior entries are committed indirectly because of the Log Matching Property.

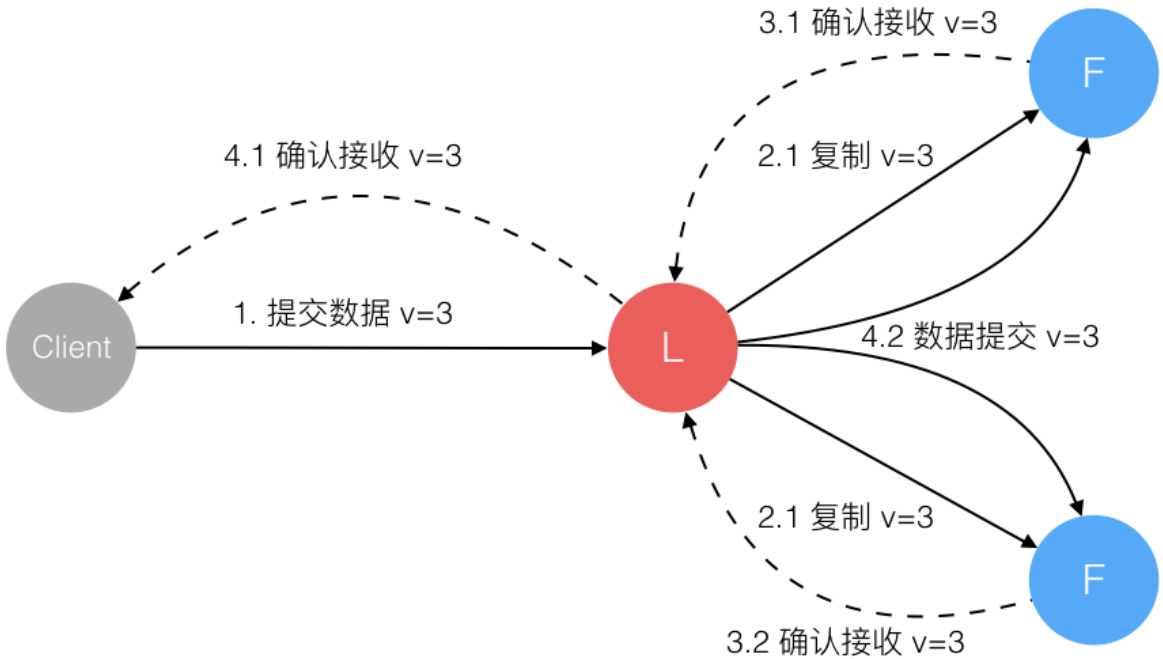
也就是说，某个leader选举成功之后，不会直接提交前任leader时期的日志，而是通过提交当前任期的日志的时候“顺手”把之前的日志也提交了，具体怎么实现了，在log matching部分有详细介绍。那么问题来了，如果leader被选举后没有收到客户端的请求呢，论文中有提到，在任期开始的时候发立即尝试复制、提交一条空的log

> Raft handles this by having each leader commit a blank no-op entry into the log at the start of its term.

因此，在上图中，不会出现（C）时刻的情况，即term4任期的leader s1不会复制term2的日志到s3。而是如同(e)描述的情况，通过复制-提交 term4的日志顺便提交term2的日志。如果term4的日志提交成功，那么term2的日志也一定提交成功，此时即使s1crash，s5也不会重新当选。

leader crash

follower的crash处理方式相对简单，leader只要不停的给follower发消息即可。当leader crash的时候，事情就会变得复杂。在这篇文章中，作者就给出了一个更新请求的流程图。



我们可以分析leader在任意时刻crash的情况，有助于理解raft算法的容错性。

raft将共识问题分解成两个相对独立的问题，leader election，log replication。流程是先选举出leader，然后leader负责复制、提交log（log中包含command）

为了在任何异常情况下系统不出错，即满足safety属性，对leader election，log replication两个子问题有诸多约束

leader election约束：

- 同一任期内最多只能投一票，先来先得
- 选举人必须比自己知道的更多（比较term，log index）

log replication约束：

- 一个log被复制到大多数节点，就是committed，保证不会回滚
- leader一定包含最新的committed log，因此leader只会追加日志，不会删除覆盖日志
- 不同节点，某个位置上日志相同，那么这个位置之前的所有日志一定是相同的
- Raft never commits log entries from previous terms by counting replicas.

本文是在看完raft论文后自己的总结，不一定全面。个人觉得，如果只是相对raft协议有一个简单了解，看这个动画演示就足够了，如果想深入了解，还是要看论文，论文中Figure 2对raft算法进行了概括。最后，还是找一个实现了raft算法的系统来看看更好。

正文

在一文搞懂raft算法一文中，从raft论文出发，详细介绍了raft的工作流程以及对特殊情况的处理。但算法、协议这种偏抽象的东西，仅仅看论文还是比较难以掌握的，需要看看在工业界的具体实现。本文关注MongoDB是如何在复制集中使用raft协议的，对raft协议做了哪些扩展。

阅读本文，需要对MongoDB复制集replication有一定认识，特别是replicat set protocol version。

在带着问题学习分布式系统之中心化复制集一文中，介绍了中心化副本控制协议。在raft（mongodb pv1）中，也是通过先选举出leader(primary)，然后通过leader(primary)管理整个复制集。

在3.2以及之后的版本中，mongodb默认使用protocol version 1。从官方的一些资料、视频可以看到，这个是一个raft-like的协议。本文主要从leader-election和log replication这两个角度来对比mongodb rs pv1与raft，并试图分析差异的原因。

需要注意的是，本文所有对MongoDB复制集的分析都是基于MongoDb3.4

本文地址：https://www.cnblogs.com/xybaby/p/10165564.html

首先对raft协议中leader election做几点总结:

1. 同一任期内最多只能投一票，先来先得
2. 选举人必须比自己知道的更多（比较term，log index）
3. 为了understandability，raft中节点之间没有ranking，公平参与投票

选举、投票资格

为了简化协议，使得raft更容易理解，raft中所有节点都能发起选举、参与投票。但在MongoDB中，有更为丰富的选举控制策略，我们从Replica Set Configuration就能看出来，replica set中的节点可以配置以下属性

```
members: [
  {
    _id: <int>,
    host: <string>,
    arbiterOnly: <boolean>,
    buildIndexes: <boolean>,
    hidden: <boolean>,
    priority: <number>,
    tags: <document>,
    slaveDelay: <int>,
    votes: <number>
  },
  ...
],
```

- arbiterOnly: Arbiter上没有用户数据，只能投票，不能发起选举，其作用在于用尽量少的资源使得复制集中节点数目为奇数。
- hidden：虽然有数据，但对客户端不可见，可以用来做备份等其他用途。hidden的priority一定是0，因此不可以发起选举，但是可以投票
- priority：`A number that indicates the relative eligibility of a member to become a primary.` priority为0时是不能发起选举的。
- votes：是否可以参与投票，mongodb复制集中最多可以有50个节点，但最多只有7个可以投票，其作用在于降低复杂度。

priority

这里再单独强调一下priority，mongodb中

> Changing the balance of priority in a replica set will **trigger one or more elections**. If a lower priority secondary is elected over a higher priority secondary, replica set members will continue to call elections until the highest priority available member becomes primary.

通过rs.reconfig()修改节点的优先级的时候，会触发重新选举。整个复制集会不断发起选举，直到最高优先级的节点成为primary。当然，在选举-投票的过程中，还是必须满足候选者数据足够新的约束。

priority很有用，比如在multi datacenter deploy的情况下，我们可能根据用户的分布情况来确定primary在哪个datacenter。

heartbeat

raft中，只有leader给follower发心跳信息（心跳是没有log-entry的Append Entries rpc），然后follower回复心跳消息。

> Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates.

在mongodb中，节点两两之间有心跳

> Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible.

primary handover

在raft中，只有当leader收到来自term更高的节点的消息时，才会切换到follower状态。如果出现网络分割（network partition），那么这个过期的leader还会一直认为自己是leader

> If a candidate or leader discovers
> that its term is out of date, it immediately reverts to follower
> state.

在mongodb中，primary在election timeout时间还没有收到来自majority 节点的消息时，会主动切换成secondary。这样可以避免过期的Primary（stale primary）继续对外提供服务，尤其是MongoDB允许writeConcern:1.

选举过程 - 预投票

raft中，在election timeout超时后，立即会发起选举，执行以下操作

- 增加节点本地的 current term ，切换到candidate状态
- 投自己一票
- 并行给其他节点发送 RequestVote RPCs
- 等待其他节点的回复
- 如果得到majority投票，成为leader

mongodb增加了一个预投票的过程（dry-run），即在不增加新的term的情况下先问问其他节点，是否可能给自己投票，得到大多数节点的肯定回复之后才会发起真正的选举过程。其作用在于尽可能减少不必要的主从切换，这部分后面还会提到。

复制集中，各个节点数据的一致性是必须要解决的问题。而对于客户端（应用）而言，复制集则需要承诺已提交的数据不能回滚。

同步or异步

在带着问题学习分布式系统之中心化复制集一文中，介绍了复制集中数据的两种复制方式，并分析了各自的优缺点。简而言之，同步方式可靠性更高，但可用性更差，网络延时更大；异步模式则恰好相反。

raft协议则是这两种方式的折中，当log复制到了大多数的节点就可以向客户端返回了。大多数节点既保证了数据的可靠性：数据不会被回滚；又保证了有较高的可用性：只有有超过一半节点存活整个系统就能正常工作。

MongoDB通过Write Concern选项将选择权交给了用户,用户可以根据实际情况来选择将数据复制到了多少节点再向客户端返回。writeconcern有三个参数

- w：写到多少节点即可向客户端返回
  - 1，默认值，即写primary即可返回，性能最高，延迟最低
  - majority，同raft，写到大多数节点才返回
  - tag set，写到指定的节点才返回，用于特殊场景

- j：是否写到journal（保证持久化）
- wtimeout：多长时间如果没有写到`w`个节点就向客户端返回错误

由于默认写到primary即可向客户端返回，那么不难想到，如果oplog尚未同步到secondary，primary挂掉，那么新选举出来的Primary可能没有最新的已经向客户端确认的数据，导致数据的回滚，后面会提到mongodb通过`catchup`来尽量避免回滚。

data flow

在带着问题学习分布式系统之中心化复制集中也给出了两种数据从primary到secondary的方式：主从模式，链式模式。其中，主从模式是priamry推送给所有的secondary，显然raft就是这种模式。

而在MongoDB中，可以通过参数settings.chainingAllowed控制使用主从模式，还是链式模式。默认值为True，即默认情况下，mongodb中secondary可以从其他secondary同步数据，这样secondary可以选择一个离自己最近（心跳延时最小的）节点来复制oplog，在MongoDB中，称oplog的同步源为SyncSource。

push or pull

raft中，leader并行将数据push到follower。而在MongoDB中，primary将数据写到local.oplog.rs，secondary定期从其SyncSource（参考上一节，不一定是从priamry拉数据，也可能是从其他secondary）读取oplog，并应用到本地。

深入浅出MongoDB复制一文中给出了一个oplog拉取的流程

MongoDB选择了pull的策略，显然会加大在writeConcern: majority时的延迟，但对于默认的链式复制，pull是更合适的，因为secondary更清楚自己的SyncSource。

append vs apply

在诸多共识算法中，都是将command封装到有序、持久化的log当中，raft和MongoD也是如此。

对于raft，leader先将log先append到本地的log entries，然后等到收到majority节点的回复后再apply log到状态机，如下入所示:

但是在mongodb中，即使客户端要求writeconcern：majority，primary也是先apply，将变更作用到状态机，再写oplog。之后，secondary再从其SyncSource的local.oplog.rs collection 拉取oplog，本地apply，然后写oplog。

MongoDB先Apply再写oplog，以及异步复制的机制，会导致即使数据无法写到大多数节点（可能primary与其他节点间网络故障），即使向客户端返回写入失败，写到primary的数据也不会回滚。

catchup

catchup既与write concern有关，也跟leader election有关。

mongodb中，有这么一个参数`settings.catchUpTimeoutMillis`, 其作用是

> Time limit in milliseconds for a **newly elected primary to sync (catch up) with the other replica set members** that may have more recent writes.
> The newly elected primary ends the catchup period early once it is fully caught up with other members of the set.
> During the catchup period, the newly elected primary is unavailable for writes from clients.

也就是说，在primary选举出来之后，会有一段时间，让primary尝试去其他节点读取到更新的写操作（more recent）。直到追加到最新的oplog，或者超时，primary才进入工作状态（接收客户端写请求）

究其原因，MongoDB允许用户自定义writeconcern，且默认只要求写到primary。因此选举的时候即使得到了大多数节点的投票，且primary的数据在这些大多数节点中是最新的，但原来的primary可能没有参与投票，那么就可能导致数据的回滚。catchup能够尽量避免回滚的出现，如果无法在`settings.catchUpTimeoutMillis`时间内完成catchup，也会将回滚的内容写入一个rollback文件。

MongoDB作为一个分布式数据库系统，既要支持OLTP，又要支持OLAP，既要满足水平伸缩，又要保证高可用、高可靠，还要支持分布式事务（Mongodb 4.x）。因此为了尽量满足不同场景下的业务需求，MongoDB提供了大量的选项，供用户选择，更加灵活。对于复制集这一块而言，选项包括但不限于：

- WriteConcern
- ReadConcern
- ReadPreference
- settings.chainingAllowed
- settings.catchUpTimeoutMillis

所以，作为MongoDB的用户，首先得清楚这些可选项的意义，然后根据自己的业务需求，合理配置。

从这些选项的默认值以及MongoDB的实现，个人觉得，在CAP这个问题上，MongoDB应该是更倾向于A（availability，可用性）的。

而诸如链式复制，Leader Priority这些特性，在分布式系统的部署层面来说都是很有用的，比如multi datacenter，很多分布式存储系统也支持同样的特性。Raft协议虽然说是为工业实现提供了很好的指导，但到具体的应用，还是得有诸多的调整和完善。

dry-run or pre-vote

MongoDB中的预投票是对raft协议很好的改进，之前，我在看Raft论文的时候也想到了一些corner case，在论文中并没有很清楚的阐述，但预投票能很好解决这些问题。

事实上，MongoDB中的预投票(dry-run)并不是独创的，在raft协议的超长版解释Consensus: Bridging Theory and Practice中，raft协议的作者就建议实现pre-vote来增加系统的鲁棒性。而在Four modifications for the Raft consensus algorithm（PS，该文的作者就是MongoDB的开发者）详细阐述了Pre-vote的原因以及实现方法。Pre-vote是为了防止一个隔离的follower不断发起选举 导致term值的激增，以及不必要的主从切换。

如上图所以，系统由s1 s2 s3三个节点组成，其中s1是leader，另外两个节点是follower。

pre-vote考虑的是这样一种情况，（s2）与（s1 s3）之间出现了网络分割（network partition），那么按照raft算法，s2会不断的尝试发起选举，意味着不断的增加term。那么当网络自愈之后，s2将消息发送到s1 s3. 按照raft论文figure2 `Rules for servers`

> If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower

因此s1 会切换到follower， s1 s3 term修改为57，但s2的log 大概率是过旧的（out of date），因此s2无法获得选举，s1 s3会在election timeout后发起选举，其中一个成为term 58的leader。

pre-vote 避免了term inflation，但更重要的是，避免了一次没有必要的重新选举: s1一定会切换到follower，然后s1或者s3再次发起选举，在这个过程中，由于没有leader，整个系统其实是不可用的（至少不可写）。