1、分布式系统基本概念

1、CAP 理论基础

分布式系统的最大难点，就是各个节点的状态如何同步。CAP 定理是这方面的基本定理，也是理解分布式系统的起点。

1998 年，加州大学的计算机科学家 Eric Brewer 提出，分布式系统有三个指标：

- Consistency
- Availability
- Partition tolerance

它们的第一个字母分别是 C、A、P。Eric Brewer 说，这三个指标不可能同时做到。这个结论就叫做 CAP 定理。

它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

- 一致性（Consistency） ：等同于所有节点访问同一份最新的数据副本，或者说同一数据在不同节点上的副本在同一逻辑时钟应当是相同的内容。
- 可用性（Availability）：每次请求都能获取到非错的响应，以及尽量保证低延迟，但是不保证获取的数据为最新数据。
- 分区容错性（Partition tolerance）：以实际效果而言，分区相当于对**通信的时限要求**。要求任意节点故障时，系统仍然可以对外服务。

2、数据一致性（C 侧）

一些分布式系统通过复制数据来提高系统的可靠性和容错性，并且将数据的不同的副本存放在不同的机器，由于维护数据副本的一致性代价高，因此许多系统**采用弱一致性来提高性能**，一些不同的一致性模型也相继被提出。

- **强一致性**： 要求无论更新操作实在哪一个副本执行，之后所有的读操作都要能获得最新的数据。
- **弱一致性**：用户读到某一操作对系统特定数据的更新需要一段时间，我们称这段时间为“不一致性窗口”。
- **最终一致性**：是弱一致性的一种特例，保证用户**最终（即窗口尽量长）**能够读取到某操作对系统特定数据的更新。

一致性解决方案

1. 分布式事务：两段提交
2. 分布式锁
3. 消息队列、消息持久化、重试、幂等操作
4. Raft / Paxos 等一致性算法

3、服务可用性（A 侧）

可用性，意思是只要收到用户的请求，服务器就必须给出回应。

高可用解决方案

- **负载均衡**：尽力将网络流量平均分发到多个服务器上，以提高系统整体的响应速度和可用性。
- **降级**：当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行。
- **熔断**：对于目标服务的请求和调用大量超时或失败，这时应该熔断该服务的所有调用，并且对于后续调用应直接返回，从而快速释放资源。确保在目标服务不可用的这段时间内，所有对它的调用都是立即返回的、不会阻塞的，等到目标服务好转后进行接口恢复。
- **流量控制**：流量控制可以有效的防止由于网络中瞬间的大量数据对网络带来的冲击，保证用户网络高效而稳定的运行，类似于 TCP 拥塞控制方法。
- **异地多活**：在不同地区维护不同子系统，并保证子系统的可用性

熔断是减少由于下游服务故障对自己的影响；而降级则是在整个系统的角度上，考虑业务整体流量，保护核心业务稳定。

4、分区容错性（P 侧）

大多数分布式系统都分布在多个子网络。每个子网络就叫做一个区（partition）。分区容错的意思是，区间通信可能失败。比如，一台服务器放在中国，另一台服务器放在美国，这就是两个区，它们之间可能无法通信。

般来说，分区容错无法避免，因此可以认为 CAP 的 P 总是成立。CAP 定理告诉我们，剩下的 C 和 A 无法同时做到。

2、系统一致性

1、基本要求

规范的说，理想的分布式系统一致性应该满足：

1. 可终止性（Termination）：一致的结果在有限时间内能完成；
2. 共识性（Consensus）：不同节点最终完成决策的结果应该相同；
3. 合法性（Validity）：决策的结果必须是其它进程提出的提案。

第一点很容易理解，这是计算机系统可以被使用的前提。需要注意，在现实生活中这点并不是总能得到保障的，例如取款机有时候会是 服务中断 状态，电话有时候是 无法连通 的。

第二点看似容易，但是隐藏了一些潜在信息。算法考虑的是任意的情形，凡事一旦推广到任意情形，就往往有一些惊人的结果。

例如现在就剩一张票了，中关村和西单的电影院也分别刚确认过这张票的存在，然后两个电影院同时来了一个顾客要买票，从各自观察看来，自己的顾客都是第一个到的.....

怎么能达成结果的共识呢？记住我们的唯一秘诀：**核心在于需要把两件事情进行排序，而且这个顺序还得是合理的、大家都认可的**。

第三点看似绕口，但是其实比较容易理解，即达成的结果必须是节点执行操作的结果。仍以卖票为例，如果两个影院各自卖出去一千张，那么达成的结果就是还剩八千张，决不能认为票售光了。

2、强一致性

线性一致性

线性一致性或称 **原子一致性** 或 **严格一致性** 指的是程序在执行的历史中在存在可线性化点 P 的执行模型，这意味着一个操作将在程序的调用和返回之间的某个点 P 起作用。

这里"起作用"的意思是被系统中并发运行的所有其他线程所感知。要求如下：

1. **写后读** 这里写和读是两个操作，如果写操作在完成之后，读才开始，读要能读到最新的数据，而且保证以后也能读操作也都能读到这个最新的数据。
2. **所有操作的时序与真实物理时间一致**，要求即使不相关的两个操作，如果执行有先后顺序，线性一致性要求最终执行的结果也需要满足这个先后顺序。比如，操作序列(写 A，读 A，写 B，读 B)，那么不仅，读 A，读 B 能读到最新 A 值和 B 值；而且要保证，如果读 B 读到最新值时，读 A 一定也能读到最新值，也就是需要保证执行时序与真实时序相同。

3. 如果两个操作是并发的（比如读 A 没有结束时，写 B 开始了），那么这个并发时序不确定，但从最终执行的结果来看，要确保所有线程(进程，节点)看到的执行序列是一致的。

顺序一致性

相比线性一致性，主要区别在于，**对于物理上有先后顺序的操作，不保证这个时序**。

具体而言，对于单个线程，操作的顺序仍然要保留，对于多个线程(进程，节点)，执行的事件的先后顺序与物理时钟顺序不保证。但是要求，从执行结果来看，所有线程(进程，节点)看到的执行序列是一样的。

因果一致性

因果一致性，被认为是比顺序一致性更弱的一致性，在因果一致性中，只对**有因果关系的事件**有顺序要求。

3、带约束的一致性

绝对理想的 **强一致性（Strong Consistency）** 代价很大。除非不发生任何故障，所有节点之间的通信无需任何时间，这个时候其实就等价于一台机器了。实际上，越强的一致性要求往往意味着越弱的性能、越低的可用性。

强一致的系统往往比较难实现，很多时候，人们发现实际需求并没有那么强，可以适当放宽一致性要求，降低系统实现的难度。

例如在一定约束下实现所谓 **最终一致性（Eventual Consistency）**，即总会存在一个时刻（而不是立刻），系统达到一致的状态，这对于大部分的 Web 系统来说已经足够了。这一类弱化的一致性，被笼统称为 **弱一致性（Weak Consistency）**。

最终一致性

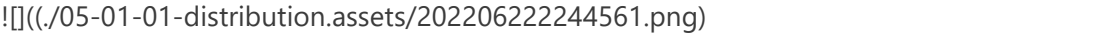
最终一致性也被称为 **乐观复制(optimistic replication)**，用户只能读到某次更新后的值，但系统保证数据将最终达到完全一致的状态，只是所需时间不能保障。这个达成一致所需要的时间，我们称为 **窗口时间**。

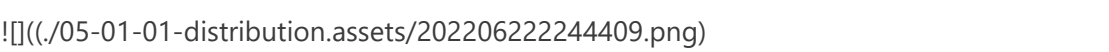
我们常见的 **异步复制的主从架构实现的是最终一致性** 。它的一个典型常见是用户读取异步从库时，可能读取到较旧的信息，因为该从库尚未完全与主库同步。注意，同步复制的主从架构会出现任一节点宕机导致的单点问题。

4、一致性（Consistency）与共识（Consensus）的关系

我们常说的 **一致性（Consistency）** 在分布式系统中指的是 副本（Replication） 问题中对于同一个数据的多个副本，其对外表现的数据一致性，如 线性一致性 、因果一致性、最终一致性等，都是用来描述副本问题中的一致性的。

而 **共识（Consensus）** 则不同，共识问题中所有的节点要最终达成共识，由于最终目标是所有节点都要达成一致，所以根本 **不存在一致性强弱** 之分。





3、高并发系统的设计

1、系统拆分

将一个系统拆分为多个子系统，用 RPC 来搞。然后每个系统连一个数据库，这样本来就一个库，现在多个数据库，不也可以扛高并发么。

2、缓存

大部分的高并发场景，都是读多写少，那你完全可以在数据库和缓存里都写一份，然后读的时候大量走缓存不就得了。毕竟 Redis 轻轻松松单机几万的并发。

所以你可以考虑考虑你的项目里，那些承载主要请求的读场景，怎么用缓存来抗高并发。

3、消息队列

可能你还是会出现高并发写的场景，比如说一个业务操作里要频繁搞数据库几十次，增删改增删改。那高并发绝对搞挂你的系统，你要是用 Redis 来承载写那肯定不行，人家是缓存，数据随时就被 LRU 了，数据格式还无比简单，没有事务支持。

所以该用 MySQL 还得用 MySQL 啊。那你咋办？用 MQ 吧，大量的写请求灌入 MQ 里，后边系统消费后慢慢写，控制在 MySQL 承载范围之内。

所以你得考虑考虑你的项目里，那些承载复杂写业务逻辑的场景里，如何用 MQ 来异步写，提升并发性。

4、分库分表

分库分表，可能到了最后数据库层面还是免不了抗高并发的要求。

好吧，那么就将一个数据库拆分为多个库，多个库来扛更高的并发；然后将一个表拆分为多个表，每个表的数据量保持少一点，提高 SQL 跑的性能。

5、读写分离

读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都集中在一个库上。

可以考虑搞个主从架构，主库写入，从库读取，搞一个读写分离。甚至说，读流量太多的时候，还可以加更多的从库。

分布式缓存、锁以及事务

1、分布式缓存

1、应用场景

1. **页面缓存**：用来缓存 Web 页面的内容片段,包括 HTML、CSS 和图片等;
2. **应用对象缓存**：缓存系统作为 ORM 框架的二级缓存对外提供服务,目的是减轻数据库的负载压力,加速应用访问;解决分布式 Web 部署的 session 同步问题，状态缓存.缓存包括 Session 会话状态及应用横向扩展时的状态数据等,这类数据一般是难以恢复的,对可用性要求较高,多应用于高可用集群。
3. **并行处理**：通常涉及大量中间计算结果需要共享;
4. **云计算领域提供分布式缓存服务**

2、缓存雪崩

缓存雪崩我们可以简单的理解为：由于原有缓存失效、新缓存未到之间(**例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期**)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

3、缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。**这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空**（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

4、缓存预热

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

5、缓存更新

除了缓存服务器自带的缓存失效策略之外，我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

1. 定时去清理过期的缓存；
2. 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的 key 是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

6、缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是 **保证核心服务可用，即使是有损的**。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

1. **一般**：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
2. **警告**：有些服务在一段时间内成功率有波动（如在 95~100%之间），可以自动降级或人工降级，并发送告警；
3. **错误**：比如可用率低于 90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；
4. **严重错误**：比如因为特殊原因数据错误了，此时需要紧急人工降级。

2、分布式锁

1、Redis 的 RedLock 锁

为了解决 Redis 单点的问题。 Redis 的作者提出了 RedLock 的解决方案。方案非常的巧妙和简洁。 RedLock 的核心思想就是，**同时使用多个 Redis Master 来冗余，且这些节点都是完全的独立的，也不需要对这些节点之间的数据进行同步**。

假设我们有 N 个 Redis 节点，N 应该是一个大于 2 的奇数。RedLock 的实现步骤:

1. 取得当前时间
2. 使用单节点获取锁的方式，依次获取 N 个节点的 Redis 锁。
3. 如果获取到的锁的数量大于 N/2+1N/2+1 个，且获取的时间小于锁的有效时间(lock validity time)就认为获取到了一个有效的锁，锁自动释放时间就是最初的锁释放时间减去之前获取锁所消耗的时间。

4. 如果获取锁的数量小于 N/2+1N/2+1 ，或者在锁的有效时间(lock validity time)内没有获取到足够的锁，就认为获取锁失败，这个时候需要向所有节点发送释放锁的消息。

对于释放锁的实现就很简单了，向所有的 Redis 节点发起释放的操作，无论之前是否获取锁成功。

2、基于 ZooKeeper 的分布式锁

方案

基于 ZK 的特性，很容易得出使用 ZK 实现分布式锁的落地方案：

1. 使用 ZK 的临时节点和有序节点，每个线程获取锁就是在 ZK 创建一个临时有序的节点，比如在 `/lock/` 目录下。
2. 创建节点成功后，获取 `/lock` 目录下的所有临时节点，再判断当前线程创建的节点是否是所有的节点的序号最小的节点。
3. 如果当前线程创建的节点是所有节点序号最小的节点，则认为获取锁成功。
4. 如果当前线程创建的节点不是所有节点序号最小的节点，则对节点序号的 **前一个节点** 添加一个事件监听。

缺陷

1. **羊群效应**：当一个节点变化时，会触发大量的 `watches` 事件，导致集群响应变慢。每个节点尽量少的 `watches`，这里就只注册 **前一个节点** 的监听
2. ZK 集群的读写吞吐量不高
3. 网络抖动可能导致 Session 离线，锁被释放

3、分布式事务

1、2PC/XA 方案

所谓的 XA 方案，即：两阶段提交，有一个事务管理器的概念，负责协调多个数据库（资源管理器）的事务，事务管理器先问问各个数据库你准备好了吗？

如果每个数据库都回复 ok，那么就正式提交事务，在各个数据库上执行操作；如果任何其中一个数据库回答不 ok，那么就回滚事务。

这种分布式事务方案，比较适合单块应用里，跨多个库的分布式事务，而且因为严重依赖于数据库层面来搞定复杂的事务，效率很低，绝对不适合高并发的场景。

一般来说某个系统内部如果出现跨多个库的这么一个操作，是不合规的。如果你要操作别人的服务的库，你必须是通过调用别的服务的接口来实现，绝对不允许交叉访问别人的数据库。

2、TCC 强一致性方案

TCC 的全称是：Try、Confirm、Cancel。

- **Try 阶段**：这个阶段说的是对各个服务的资源做检测以及对资源进行 **锁定或者预留**。
- **Confirm 阶段**：这个阶段说的是在各个服务中执行实际的操作。
- **Cancel 阶段**：如果任何一个服务的业务方法执行出错，那么这里就需要 **进行补偿**，就是执行已经执行成功的业务逻辑的回滚操作。（把那些执行成功的回滚）

这种方案说实话几乎很少人使用，但是也有使用的场景。因为这个**事务回滚实际上是严重依赖于你自己写逻辑来实现回滚和补偿**，会造成巨大的补偿代码量。

3、可靠消息最终一致性方案

基于 MQ 来实现事务。比如阿里的 RocketMQ 就支持消息事务。大概的意思就是：

1. A 系统先发送一个 prepared 消息到 MQ，如果这个 prepared 消息发送失败那么就直接取消操作别执行了；
2. 如果这个消息发送成功过了，那么接着执行本地事务，如果成功就告诉 MQ 发送确认消息，如果失败就告诉 MQ 回滚消息；
3. 如果发送了确认消息，那么此时 B 系统会接收到确认消息，然后执行本地的事务；
4. mq 会自动定时轮询所有 prepared 消息回调你的接口，问你，这个消息是不是本地事务处理失败了，所有没发送确认的消息，是继续重试还是回滚？一般来说这里你就可以查下数据库看之前本地事务是否执行，如果回滚了，那么这里也回滚吧。这个就是避免可能本地事务执行成功了，而确认消息却发送失败了。
5. 这个方案里，要是系统 B 的事务失败了咋办？重试咯，自动不断重试直到成功，如果实在是不行，要么就是针对重要的资金类业务进行回滚，比如 B 系统本地回滚后，想办法通知系统 A 也回滚；或者是发送报警由人工来手工回滚和补偿。

4、最大努力通知方案

1. 系统 A 本地事务执行完之后，发送个消息到 MQ；
2. 这里会有个专门消费 MQ 的最大努力通知服务，这个服务会消费 MQ 然后写入数据库中记录下来，或者是放入个内存队列也可以，接着调用系统 B 的接口；
3. 要是系统 B 执行成功就 ok 了；要是系统 B 执行失败了，那么最大努力通知服务就定时尝试重新调用系统 B，反复 N 次，最后还是不行就放弃。

Raft

1、Raft Overview

> Raft 算法讲解

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。

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

2、leader election

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

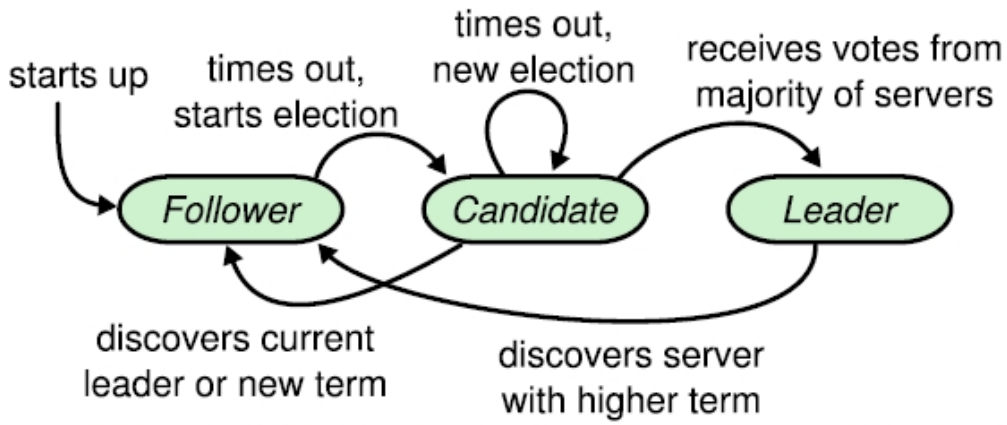


Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

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**。

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

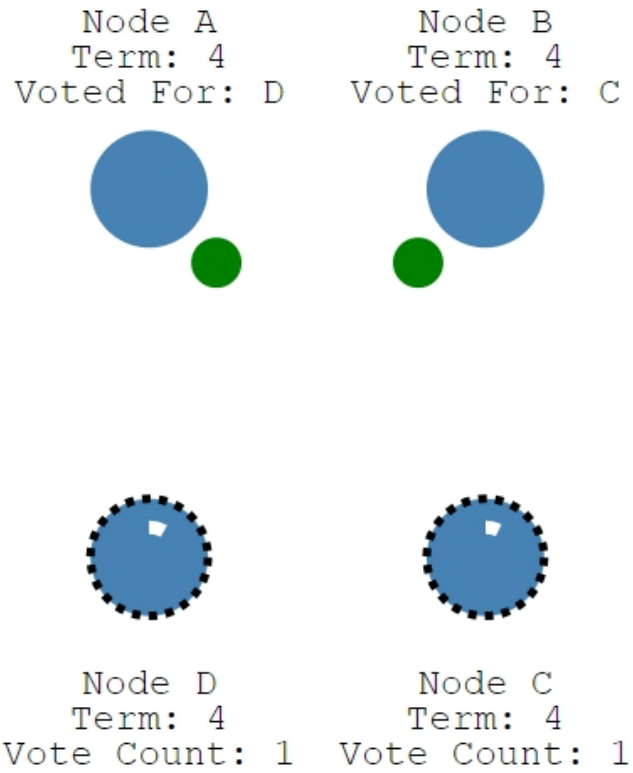
1. 收到 majority 的投票（含自己的一票），则赢得选举，成为 leader
2. 被告知别人已当选，那么自行切换到 follower
3. 一段时间内没有收到 majority 投票，则保持 candidate 状态，重新发出选举

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。

3、log replication

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

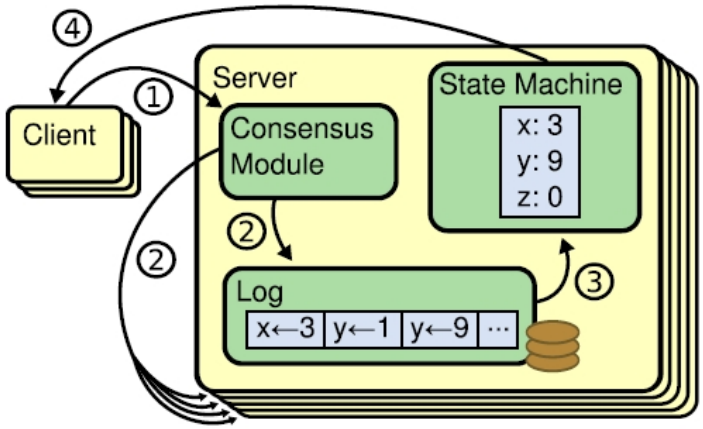


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

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

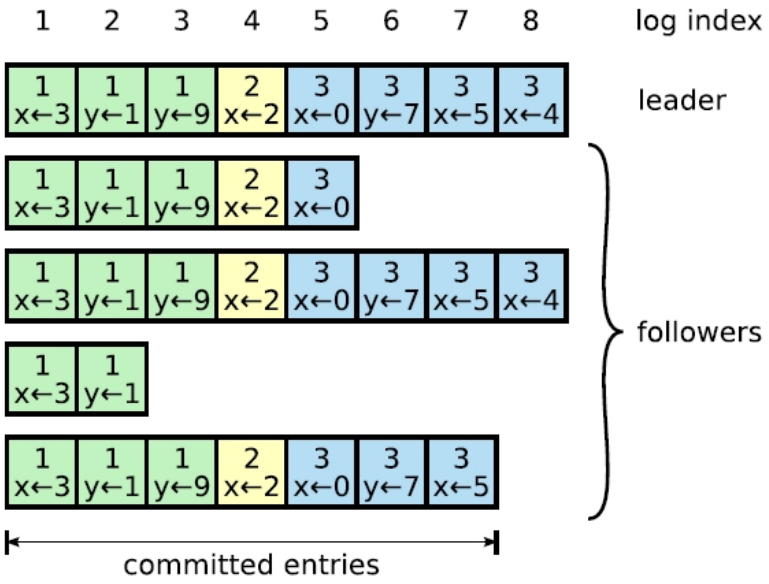


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

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

4、safety

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

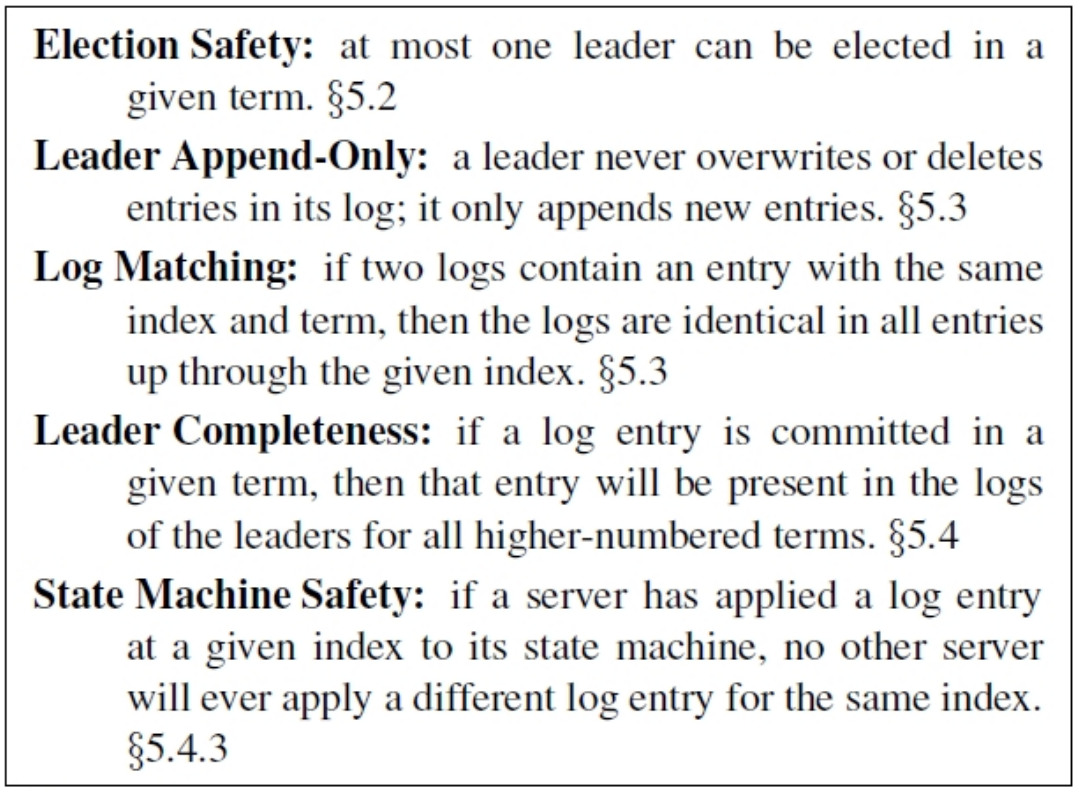


Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

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**。

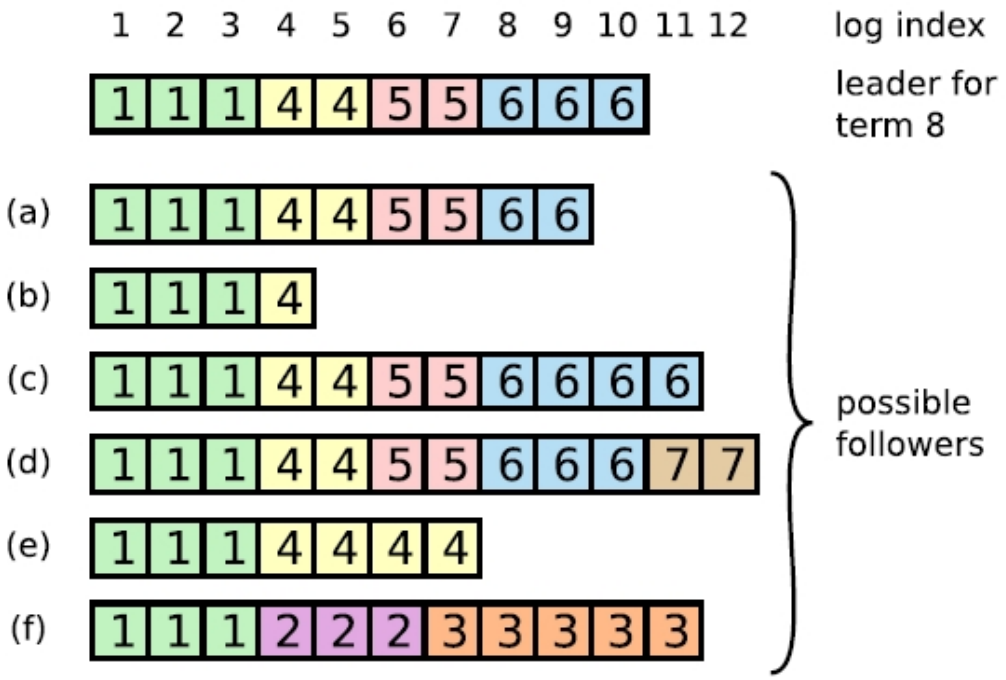
log matching

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 leader 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 的回复，如果返回值是 False，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.

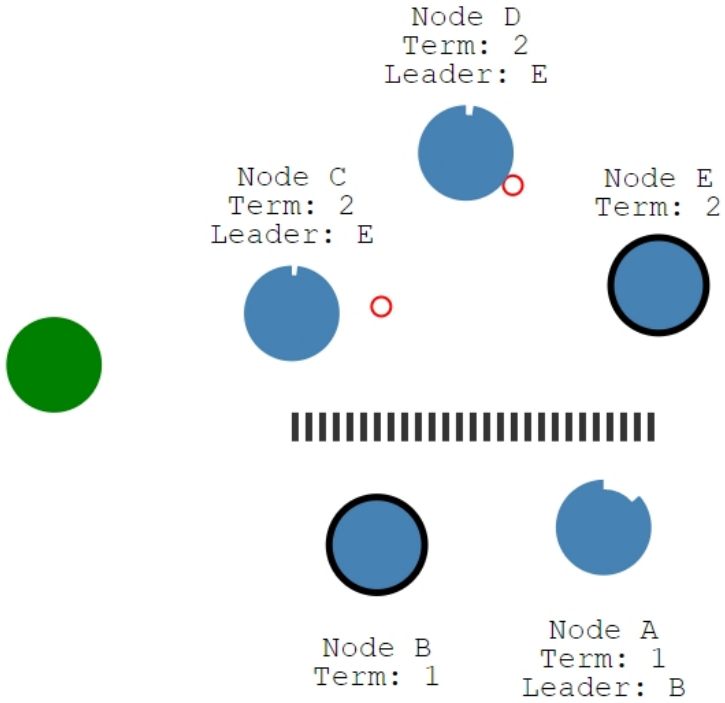
上面两点都提到了 majority：commit majority and vote majority，根据 Quorum，这两个 majority 一定是有重合的，因此被选举出的 leader 一定包含了最新的 committed 的日志。

raft 与其他协议（Viewstamped Replication、mongodb）不同，raft 始终保证 leader 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。

5、corner case

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。

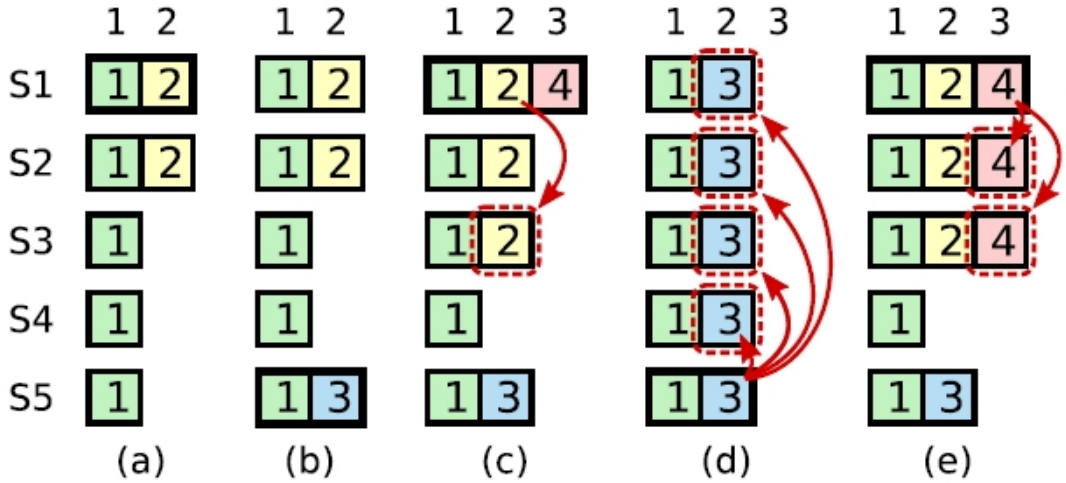
State Machine Safety

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。

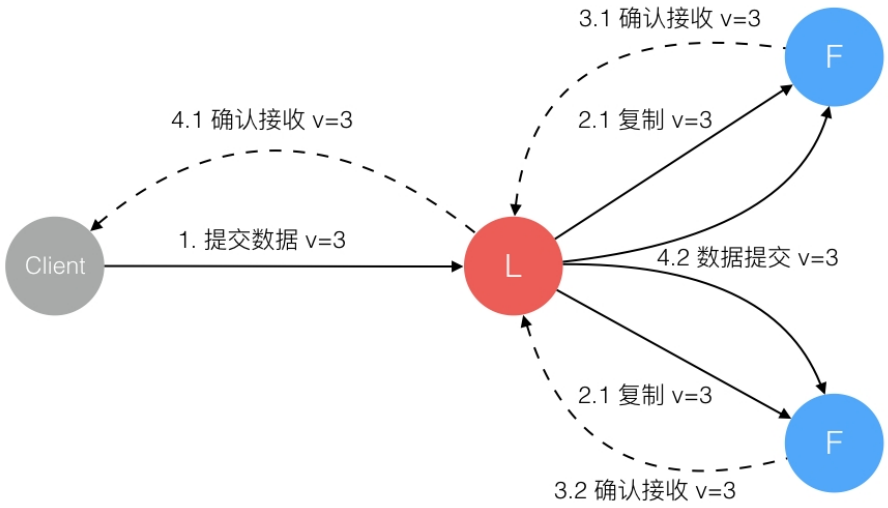
那么问题来了，如果 leader 被选举后没有收到客户端的请求呢，论文中有提到，在任期开始的时候发立即尝试复制、提交一条空的 log

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。

6、总结

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。