Algo

2021年11月15日 8:18

==========

并行：双核就是2个并行

并发：许多个任务在 某个时间点 的人所见到的状态 都是执行中 (但是真正执行的是看CPU调度)

==========

限流算法：漏桶算法，令牌桶算法

漏桶算法：请求先进入到漏桶里，漏桶以一定的速度释放请求。当请求的速度大于释放的速度，桶会满，满了以后能直接拒绝(。。。或者其他，或许可以再起一个服务)。
能限制数据的传输速率

对于很多场景来说，除了要能限制数据的平均传输速率外，还要求允许某种程度的突发传输。此时漏桶算法就不再适用，令牌桶算法更合适。
。。但是 漏桶不合适，是因为 突发传输可能会 大于 桶的容量，导致部分被拒绝，但是 桶够大不就可以了？ 令牌桶也可以因为 突发传输过大 导致令牌发完啊。
。。还是说， 突发传输是指： 需要在一定的时间内处理完这些请求。 漏桶由于 释放请求是固定的，所以 无法大量并发。 令牌桶可以 短时大量并发(多少个令牌就能多少个并发)。

令牌桶算法：系统以一定的速度往令牌桶里放令牌，如果请求要被处理，则需要先从桶里获取一个令牌，当没有令牌时，拒绝服务(。。。或许也可以再起一个服务)

Guava有 RateLimiter 实现 令牌桶算法 来限流
```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire());
```
1、RateLimiter.create(5) 表示桶容量为5且每秒新增5个令牌，即每隔200毫秒新增一个令牌；
2、limiter.acquire()表示消费一个令牌，如果当前桶中有足够令牌则成功（返回值为0），如果桶中没有令牌则暂停一段时间，比如发令牌间隔是200毫秒，则等待200毫秒后再去消费令牌

（如上测试用例返回的为0.198239，差不多等待了200毫秒桶中才有令牌可用），这种实现将突发请求速率平均为了固定请求速率。

```
RateLimiter limiter = RateLimiter.create(5);
System.out.println(limiter.acquire(5));
```

漏桶算法，令牌桶算法 的场景不一样
令牌桶可以用来保护自己，主要用来对调用者频率进行限流，防止自己被击垮。所以如果自己有处理能力时，如果流量突发(实际消费能力大于配置的流量限制)，那么实际处理速率可以超过配置的限制。
漏桶算法，是用来保护它所调用的系统。主要场景是：当调用的第三方系统本身没有保护机制，或者有流量限制时，我们的调用速度不能超过对方的限制，由于我们无法更改第三方，所以只有在我们(调用方)控制。此时，即使流量突发，也必须舍弃，因为消费能力是第三方决定的。

服务治理-限流(令牌桶算法)
对于分布式服务的框架来说，除了远程调用，还要进行服务治理
当进行促销时，所有资源都用来完成重要的业务，如双11时，主要业务就是查询商品，购买支付，其他的如金币查询，积分查询等业务都是次要的，因为要对这些服务进行服务的降级，典型的服务降级算法是采用令牌桶算法。

在实施QOS策略时，可以将用户的数据限制在特定的带宽：
当网络设备衡量流量是否超过额定带宽时，需要查看令牌桶，一个令牌允许接口发送或接受1bit数据(有时是1byte数据)，当接口通过1bit数据后，要从桶中移除一个令牌。
当桶里没有令牌的时候，任何流量都被视为超过额定带宽，只有桶中有令牌时，数据才可以通过接口。
往桶里加令牌的速度，就决定了数据通过接口的速度。通过控制加令牌的速度来控制用户流量的带宽。

。。。QOS特性就是用来修理网络数据传输过程中的一些小瑕疵的特性。只要你把这个数据路径修理的足够光滑，在某种程度来说没有任何的阻碍了，那么数据跑起来就会相当的流畅，什么丢包啊，延迟啊，延迟抖动啊就都统统解决啦。速度和质量得到了双保障。

。。。这2个 应该 既可以限流 请求， 也可以限流 网络流量。
。。好像都是说 限流网络流量。

---

https://www.jianshu.com/p/c02899c30bbd

漏桶算法(Leaky Bucket)是 流量整型(Traffic Shaping) 和 速率限制(Rate Limiting) 经常使用的算法
令牌桶算法是 流量整型 和 速率限制 最常使用的算法

区别：漏桶算法能 限制数据的传输速率， 令牌桶算法能限制数据的平均传输速率的同时还允

许某种程度的突发传输。

几种算法：
Leaky Bucket
Fixed Window
Sliding Log
Sliding Window

Leaky Bucket
漏桶算法，预先设置一个请求数的上限，小于上限时 接受请求，大于上限时 直接拒绝，只有等系统处理完一部分请求，桶里有新坑位，才会接收新的请求。

优点：
能平滑请求数，使系统以一个均匀的速率处理请求
容易实现，可以用 队列/FIFO 来做
可以只用很小的内存做到为每个用户限流

缺点：
桶满以后，新的请求会被扔掉，系统忙着处理旧请求。。。。就是那个线程池的拒绝策略，可以把最老的未处理的请求扔掉。
无法保证请求能够在一个固定的时间内处理完

Fixed Window
固定窗口算法，设置一个时间段内(窗口) 接收的请求数，超过的请求会被丢弃
窗口通常选人们熟悉的时间段： 1分钟、1小时
窗口的起始时间通常是当前时间的floor，比如 以一分钟窗口为例，12:00:03的窗口就是 12:00:00 - 12:01:00

优点
和漏桶相比，能够让新来的请求也能被处理到。。。？估计是说周期结束后能保存新的请求，漏桶是永远卡住的，但是，永远卡住得是 处理永远卡住啊， 处理卡住，什么算法都不好用。

缺点：
在窗口的起始时间，最差情况下可能会带来 2倍的流量。。。不是瞬间就用完整个窗口期的流量吗，感觉瞬间是 无穷大的速率。。。可能说的是 前一个的尾巴和这一个的头 有大量流量，所以2倍？
很多消费者可能都在等待窗口被重置，造成惊群效应(。。。一个资源可用时，很多等待这个资源的线程被唤醒，开始抢，但是只有一个能抢到)。。。但是这个和惊群有什么关系，窗口满了不是直接丢弃了吗。。

Sliding Log
滑动日志算法，利用记录下来的用户的请求时间，请求数，当该用户的一个新请求进来时，比较这个用户在这个窗口内的请求数时候已经超过了限定，超过就拒绝。

优先
避免了固定窗口算法在窗口边界可能出现的2倍流量问题
由于是针对每个用户进行统计的，所以不会引发惊群效应

缺点
需要保存大量的请求日志
每个请求都要考虑用户之前的请求情况，在分布式系统中尤其难做到

Sliding Window
滑动窗口算法，结合了固定窗口算法的 低开小 和 滑动日志的解决边界问题

为每个窗口进行请求量的计数
结合上一个窗口的请求量和这一个窗口已经经过的时间来计算出上限，以此平滑请求尖峰

举例：限流的上限是每分钟10个请求，窗口大小为1分钟，上一个窗口总共处理了6个请求，现在假设这个 新窗口已经经过了20秒，那么到目前为止，允许的请求上限是 10 - 6 * （1-20/60） = 8

滑动窗口是最实用的算法：
很好的性能
避免了漏桶带来的饥饿问题
避免了固定算法的请求量突增问题

。。。怎么偏到这里了， 令牌桶算法呢。。。

分布式实现
同步策略
流量上限都是针对全站设置的，每个节点应该如何协调各自的流量？
解决方法通常是 使用一个统一的数据库来存放计数，比如redis或Cassandra。数据库中存放每个窗口和用户的计数值。这种方法的主要问题是需要多访问一次数据库，以及竞争问题。

竞争问题就是 多个线程同时指向i+=1是，如果没有同步这些操作的话，i的值可能有多种。
竞争问题可以通过 加锁来解决，但是在 限流的场景下，锁肯定会成为系统的瓶颈。
更好的方法是通过set-then-get，限流场景中用到的只是 计数+1，利用这一点 及 数据库实现的性能更好的原子操作可以达到我们的目的。。。。乐观锁？

使用集中式的数据库的 另一个问题是 每次请求都要 检索数据库 所带来的开销。
解决这个问题可以通过 在内存中 维护一个计数值，代价是稍微的放松 限流的精确度。
通过设置一个定时任务从数据库里拿计数值，周期内在内存中维护这个计数，周期结束把 计数同步到数据库 并拿取新的计数。
同步周期往往做出可配置的，小的周期能更精准的限流，大周期能减轻数据库IO压力。
。。。我还以为是 懒数据库查询，如果数据过期，就查，不过期就用这个值 来判断 是否限流。
。。。而且 怎么回写到数据库。 获取新的，然后 +用掉的， 再保存到数据库？cas有点麻烦。。不，可以减少很多次的cas(以前是+1就更新一次，现在是+N才更新一次，cas碰撞小了以后，次数直线下降)， 但是 周期结束为什么不直接丢弃，因为 滑动窗口需要上一个窗口的值。

。。。为什么不是 先申请 多个流量，然后自己开始处理，流量用完，再去尝试申请。。处理完把剩余的流量再写回去。或者不回写

---

https://www.cnblogs.com/duanxz/p/4123068.html

限流算法：
计数器
漏桶算法
令牌桶算法
滑动时间窗
三色速率标记法

滑动时间窗，三色速率标记法 是 漏桶和令牌桶 的变种。要么将 漏桶 容积换成时间单位，要么是按规则将请求标记颜色进行处理，底层还是 令牌 的意思

计数器
最简单，最容易实现
比如，我们规定，对于A接口，1分钟的访问次数不能超过100个，那么我们可以设置一个计数器counter，有效时间是1分钟(即每分钟会被重置为0)，每当一个请求过来时，counter就+1，如果counter大于100，就说明请求数过多。
有一个致命问题，就是临界问题： 0:59有100个请求，1:01有100个请求。

漏桶算法
请求先到漏桶里，漏桶以一定的速度释放，当流入速度过大 会发生溢出，就拒绝请求。
流入：以 任意 速率往桶里增加请求
流出：以 固定 速度从桶里获得请求

令牌桶算法
系统以恒定的时间间隔往桶里加入令牌，如果桶已经满了，令牌就溢出。请求来的时候，需要从令牌桶里拿到一个令牌，如果没有令牌就阻塞或拒绝服务。

令牌桶可以方便改变速率。

流入： 以 固定 速度往桶里增加令牌
流出： 以 任意 速率从桶里拿令牌

令牌桶放在服务端，用来保护服务端(自己)，用来对 调用者进行限制，防止自己被压垮
漏桶算法，放在调用方，用来保护它所调用的系统。主要场景是，当调用的第三方系统本身没有保护机制，或者有流量限制的时候，我们的调用速度不能超过它的限制。

滑动时间窗
又称为Rolling Window
例子：一个时间窗口是1分钟，将时间窗口进行划分，比如划成6格，每格代表10秒，每过10秒，时间窗口就滑动一格。每个格子都有独立的计数器。

格子越多，滚动越平滑，限流越精确。

三色速率标记法

。。这个要2个令牌桶， 挺烦的。而且也没有 详细的解释。。还分色盲模式，色感模式。。还分单速率，双速率。。。 通信的比较多。。 RFC2698-双速率三色标记。

==========

读扩散

写扩散

==========

消息的 推/拉/推拉结合

==========

读写分离

==========

ID的 全局递增/用户级别递增/会话级别递增

==========

会话ID

==========

惊群效应

多进程(多线程) 同时阻塞 等待 一个事件的时候(休眠状态)， 这个事件发生，它会唤醒 所有的进程(或线程)，但是最终 只有一个 进程/线程 获得 控制权， 其他 进程/线程 重新进入休眠状态。

Linux 2.6 版本之后，通过引入一个标记位 WQ_FLAG_EXCLUSIVE，解决掉了 accept 惊群效应。

我们知道 epoll 对惊群效应的修复，是建立在共享在同一个 epoll 结构上的。epoll_create 在 fork 之后执行，每个进程有单独的 epoll 红黑树，等待队列，ready 事件列表。因此，惊群效应再次出现了。有时候唤醒所有进程，有时候唤醒部分进程，可能是因为事件已经被某些进程处理掉了，因此不用在通知另外还未通知到的进程了。

==========

分布式ID

---

https://juejin.cn/post/7148602372699193352

Twitter的 Snowflake算法
UUID

数据库自增

这里谈 MongoDB。

MongoDB 一开始的设计就是用来作为分布式DB。
插入数据时，默认使用 _id 作为主键。 这个 _id 就是 MongoDB 中 开源的 分布式系统ID 算法 ObjectId() 生成的

```
new ObjectId("632c6d93d65f74baeb22a2c9")
```

网上很多文章，都有这样一段描述

```
// 过时的规则，现在已经不用 机器标识 + 进程号
// 一种猜测，现在大多应用容器化，在容器内有独立的进程空间，它的进程号永远可能都为 1，还有创建几台虚拟机，其中的 hostname 可能也都为 localhost
4 字节的时间戳 + 3 个字节机器标识码 + 2 个字节进程号 + 3 个字节自增数
```

查看源码后，发现中间的 3字节机器标识码+2字节进程号 已经被替换为 5个字节的 进程唯一标识

```
// 当前 ObjectId 实现规则
4 字节的时间戳（单位：秒） + 5 个字节的进程唯一标识 + 3 个字节自增数
```

1. 由于前4个字节是 秒数，所以总体上是递增的，所以有时 可以使用 _id 来进行 按创建时间排序
2. 中间5个字节，是 进程唯一标识，在进程启动后，只需要生成一次。
3. 后3个字节为自增数，最大可以表达16777215 个值，所以 1秒内 不超过这个值 就是唯一的。。。这个值不是每秒从0开始，而是一开始一个初始值，后续每次都是+1，然后mod。

Twitter的 Snowflake算法

==========

==========

http://thesecretlivesofdata.com/raft/
动图Raft

。。只有这一个。。

Raft算法

https://github.com/maemual/raft-zh_cn/blob/master/raft-zh_cn.md
。。Raft论文的翻译

寻找一种易于理解的一致性算法（扩展版）

Raft 是一种 为了 管理复制日志 的 一致性算法。 提供了 和 Paxos 算法 相同的 功能 和 性能，但是 算法结构 不同，是的 Raft 更容易理解 并且 更容易 构建 实际的系统。
为了提升可理解性， Raft 将算法 分为 几个关键模块，例如，领导人选举，日志复制 和 安全性。 同时 它通过实施 一个更强的 一致性 来 减少 需要 考虑的 状态的数量。

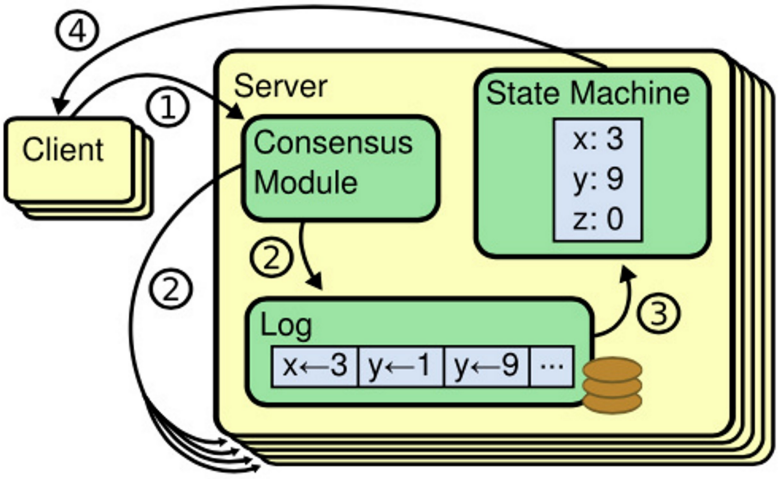
Raft 算法 在许多方面 和 现有的 一致性算法 都很相似，但是 它也有一些独特的 特性：

强领导人：和其他一致性算法相比， Raft 使用一种 更强的 领导能力形式。比如，日志条目 只从 领导人 发送给 其他的服务器。这种方式 简化了 对复制日志 的管理 并且 是的 Raft 算法 更加易于理解。

领导选举：Raft算法 使用一个 随机 计时器 来选举领导人，这种 方式 只是在 任何一致性算法 都必须实现的 心跳机制上 增加一点机制， 在解决冲突时 会更加简单快捷。

成员关系调整： Raft 使用一种 共同一致 的方式 来处理 集群成员变化 的问题， 在这种方法下，处于调整过程中的 2种 不同的配置 集群中 大多数机器 会有 重叠， 这就使得 集群在 成员变换的时候 依然可以继续工作。

复制状态机
一致性算法 是从 复制状态机 的背景下提出的。在这种方法中， 一组服务器上的 状态机 产生 相同状态的 副本，并且 在一些 机器 宕机的情况下 也能继续运行。复制状态机 在分布式系统中 被用于 解决 很多容错的 问题。例如，大规模的 系统中 通常都有 一个 集群领导人，想 GFS，HDFS，RAMCloud，典型应用就是 一个 独立的 复制状态机 去管理 领导选举 和 存储配置 信息 并且 在 领导人 宕机的情况下 也要 存活下来，比如 Chubby 和 ZooKeeper。



图：复制状态机的结构。一致性算法 管理着 来自客户端指令 的复制日志。状态机 从日志中处理 相同顺序的相同指令，所以产生的结果也是相同的。

复制状态机通常 都是基于 复制日志实现的。每个服务器 存储一个 包含 一系列指令的 日志，并且按照日志的顺序 进行执行。每个日志都 按照 相同的顺序 包含相同的指令，所以 每个服务器 都执行 相同的指令序列。因为每个状态机 都是确定的，每次执行操作 都产生 相同的 状态和 同样的序列。

一致性算法 的任务是 保证 复制日志的一致性。
服务器上的 一致性模块 接收客户端 发送的指令，然后添加到 自己的日志中。它和其他服务器上的 一致性模块 进行通信来 保证 每个服务器上的 日志 最终 都以相同的顺序 包含相同的请求，即使有些服务器 发生故障。
一旦指令被正确复制，每一个服务器的状态机 按照日志顺序 处理它们，然后输出结果 到 客户端。因此，服务器集群看起来 形成了 一个 高可用的 状态机。

实际系统中 使用的 一致性算法通常包含 以下特性：

安全性保证(绝对不会返回一个错误的结果)：在 非拜占庭错误 情况下，包括 网络延迟，分区，丢包，重复，乱序 等错误 都可以 保证 正确

可用性：集群中 只要有 大多数 的机器 可以运行 并且 能够互相通信，和客户端通信，就可以保证可用。因此，一个典型的 包含 5个节点的 集群 可以容忍 2 个节点的 失败。服务器被停止 就认为 是失败。 它们稍后 可能会从 可靠存储的 状态中 恢复 并 重新加入集群。

不依赖时序来保证一致性：物理时钟的错误 或 极端的消息延迟 只有在 最坏的情况下 才会导致 可用性问题。

通常情况下，一条指令 可以尽可能快的 在集群中 大多数节点响应 一轮远程 过程调用时完成。 小部分比较慢的节点 不会影响 系统的整体性能。

Paxos算法的问题
过去10年中，Leslie Lamport 的 Paxos 算法 几乎已经成了 一致性的代名词：Paxos 是课程教学中 最经常使用的算法，同时也是 大多数一致性算法 实现的 起点。
Paxos 首先定义了 一个 能够达成 单一决策一致 的协议。比如，单条的复制日志项。我们把这一子集叫做 单决策 Paxos。然后通过 组合多个 Paxos 协议的实例 来促进一系列决策 的达成。
Paxos 保证 安全性和活性，同时也支持 集群成员关系的变更。
Paxos 的正确性已经被证明，通常情况下 也很高效

不幸的是，Paxos 有2个 明显的缺点。第一个缺点是 Paxos 特别难以理解。完整的解释 是出了名的 不透明。
我们假设 Paxos 的不透明性 来自 它选择 单决策问题 作为 它的基础。单决策 Paxos 是 晦涩微妙的， 它被划分成了 2种 没有简单 直观解释 和 无法独立理解的情景。导致 很难建立直观的 感受 为什么单决策Paxos 能够工作。 构成多决策Paxos 增加了 错综复杂的规则。
我们相信， 在多决策上 达成一致性的 问题 (一份日志 而不是 单一的日志记录) 能够被分解成 其他的方式 并且 更加直接 和明显。

Paxos 的第二个问题就是 它 没有提供一个 足够好用的 用来构建一个 现实系统的基础。 一个原因是 还没有一种 被广泛认可的 多决策问题 的算法。 Lamport 的描述 基本上 都是关于单决策 Paxos 的， 他简要描述了 实施 多决策Paxos 的方法，但是缺乏很多细节。 当然也有很多具体化 Paxos 的尝试，但是他们都 互不一样，和Paxos 的概述也不同。例如 Chubby 这样的系统 实现了一个类似于 Paxos 的算法，但是大多数细节并没有被公开。

而且 Paxos 算法的结构 也 不是 易于构建实践的系统；单决策分解 会产生其他的结果。例如，独立地选择一组日志条目 然后合并成一个 序列化的 日志 并没有带来 太多的好处，仅仅增加了不少复杂性。围绕着 日志来 设计一个 系统是更加简单 高效的；新日志条目以 严格限制的顺序 添加到 日志中去。另一个问题是，Paxos 使用了 一种 对等 的点对点的方式 作为它的核心（尽管它 最终提议了一种 弱领导人的 方法 来优化性能）。在只有一个 决策会被制定 的简化世界中 是很有意义的， 但是很少有 现实的 系统 使用这种方式。 如果有一系列的决策需要 被制定，首先选择一个领导人，然后让他 去协调所有的 决议，会更加简单快速。

因此，实际的系统中很少有 和 Paxos 相似的实践。每一种 实现 都是从 Paxos 开始研究，然后发现 很多 实现上的难题，再然后 开发一种 和 Paxos 明显不一样的 结构。
Paxos算法 在理论上 被证明是 正确可行的，但是 现实的系统 和 Paxos 差别是如此的大，以至于 这些证明 没有什么太大的价值。

为了可理解性的设计
设计Raft 有几个初衷：它必须提供一个 完整的 实际的 系统实现基础，这样才能大大减少 开发者的工作，它必须在任何情况下 都是安全的 并且 在大多数情况下都是可用的；并且它的大部分操作 必须是高效的。 但是我们最重要 也是最大的挑战 是可理解性。 它必须保证 对于普通人 是可以容易理解的。 另外，它必须能够让人形成直观的认识，这样系统的 构建者 才能够 在现实中进行 必然的 扩展。

在设计Raft算法的时候，有很多的点 需要我们在各种备选方案中进行选择。这种情况下，我们基于 可理解性 来评估备选方案： 解释各个备选方案 有多大的难度（例如，Raft 的状态空间有多复杂，是否有微妙的暗示），对于一个读者而言，完全理解这个方案 和 暗示 是否容易。

我们意识到 这种可理解性分析 具有高度的 主观性；尽管如此，我们使用了 2种通常适用的技术 来解决这个问题。第一个技术 是 众所周知的 问题分解：我们尽可能地 将问题 分解为几个 相对独立的， 可被解决的，可解释的，可理解的 子问题。例如 Raft算法被 分为 领导人选举，日志复制，安全性，成员变更 几个部分。

使用的第二个方法是 通过减少 状态 的数量 来简化 需要考虑的 状态空间，使得系统 更加连贯 并且 在可能的时候 消除不确定性。 特别的，所有的日志是 不允许有 空洞的，并且 Raft限制了 日志之间 变成 不一致状态的 可能。尽管在 大多数情况下 我们都 试图 消除不确定性，但是 也有一些情况下 不确定性 可以提示 可理解性。 尤其是，随机化方法 增加了 不确定性，但是 它们有利于 减少 状态空间 数量，通过处理 所有可能选择时 使用 相似的方法。我们使用 随机化 来简化 Raft 中 领导人选举算法。

Raft一致性算法
Raft 是一种 用来管理 复制日志的 算法。
Raft 通过选举一个 杰出的领导人，然后给予 他 全部的 管理 复制日志的 责任来实现 一致性。领导人 从客户端 接收 日志条目(log entries)， 把日志条目 复制到 其他服务器上，并且告诉 其他服务器 什么时候 可以安全地将日志条目 应用到 它们的 状态机中。拥有一个 领

导人 大大简化了 对复制日志的管理。 例如，领导人可以决定 新的 日志条目需要放在 日志中的什么位置 而不需要 和其他服务器商议，并且 数据都是从 领导人 流向 其他服务器。一个 领导人可能发生故障，或者 和 其他服务器 失去连接，这种情况下，新的领导人会被选举出来。

通过领导人的方式，Raft将 一致性问题分为 3个相对独立的 子问题：

领导选举：当现存的 领导人发生故障时，一个新的领导人 需要被选举出来

日志复制：领导人 必须从 客户端接收日志条目 然后 复制到 集群中的其他节点，并强制要求 其他节点的 日志 和 自己保持一致

安全性：在Raft 中安全性的关键是 状态机安全：如果 有任何的服务器节点 已经应用了一个 确定的 日志条目到它的 状态机中，那么 其他服务器节点 不能在 同一个 日志索引位置 应用一个 不同的指令。

状态

所有服务器上的 持久性状态 (在响应 RPC 请求之前，已经更新到了 稳定的存储设备)

参数	解释
currentTerm	服务器已知的最新任期 (在服务器首次启动时初始化为0，单调递增)
votedFor	当前任期内收到的选票的 candidateId，如果没有投给任何候选人，则空
log[]	日志条目，每个条目包含了 用于状态机的命令，以及领导人收到该条目时的任期(初始索引为1)

所有服务器上的 易失性状态

参数	解释
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)
lastApplied	已经被应用到 状态机 的最高的 日志条目的 索引(初始0，单条递增)

领导人(服务器)上的 易失性状态(选举后 已经重新初始化)

参数	解释
nextIndex[]	对于每台服务器，发送到 该服务器的下一个日志条目的索引 (初始值为 领导人的最后 日志条目的 索引 + 1)
matchIndex[]	对于每台服务器，已知的 已经复制到 该服务器的 最高日志条目的索引(初始0，单调递增)

追加条目(AppendEntries) RPC

由领导人调用，用于日志条目的复制，同时也被当做心跳使用。

参数	解释
term	领导人的任期
leaderId	领导人ID 因此跟随者可以对 客户端进行重定向 (译者注：跟随者根据领导人ID 把客户单的 请求重定向到 领导人，比如 有时 客户端把请求发给了 跟随者 而不是领导人)

prevLogIndex	紧邻新日志条目之前的那个日志条目的 索引
prevLogTerm	紧邻新日志条目之前的 那个日志条目的 任期
entries[]	需要被保存的 日志条目(被当做心跳使用时， 日志条目内容为空；为了提高效率 可能一次性发送多个)
leaderCommit	领导人的已知 已提交的 最高的 日志条目的 索引

返回值	解释
term	当前任期，对于领导人而言，它会更新自己的任期
success	如果跟随者 所含有的条目 和 prevLogIndex 以及 prevLogTerm 匹配，则为 true

接受者的实现

1. 返回false，如果领导人的 任期 小于 接收者的当前任期 (译者注：这里的接收者 是指跟随者 或 候选人)
2. 返回false，如果接收者日志中 没有包含这样一个条目 即该条目的任期 在 prevLogIndex 上能和 prevLogTerm 匹配上 (译者注：在接受者日志中 如果能找到一个 和 prevLogIndex 以及 prevLogTerm 一样 的 索引 和 任期的 日志条目 则继续执行下面的步骤，否则返回假)
3. 如果一个已经存在的 条目 和 新条目(译者注：即刚刚收到的日志条目) 发生冲突 (因为索引相同，任期不同)， 那么就删除 这个已经存在的 条目 以及它之后的 所有条目。
4. 追加日志中 尚未 存在的 任何新条目
5. 如果领导人的 已知 已提交的 最高日志条目的 索引大于 接受者的 已知已提交 最高日志条目的 索引 (leaderCommit > commitIndex)， 则把 接受者的 已知 已提交的 最高的日志条目的 索引 commitIndex 重置为 领导人的 已知 已经提交的 最高的 日志条目的 索引 leaderCommit 或者是 上一个 新条目的 索引 取 2者的最小值。

请求投票 (RequestVote) RPC
由候选人 负责调用 用来征集选票

参数	解释
term	候选人的任期号
candidateId	请求选票的候选人的ID
lastLogIndex	候选人的 最后日志条目的 索引值
lastLogTerm	候选人 最后日志条目的 任期号

返回值	解释
term	当前任期号，以便于候选人 去更新自己的 任期号
voteGranted	候选人 赢得此张 选票时 为真

接收者实现：

1. 如果 term < currentTerm 返回 false
2. 如果 votedFor 为空 或者 为 candidateId，并且 候选人的 日志 至少和 自己一样新，则投票给他。

所有服务器 需遵循的规则

所有服务器：

如果 commitIndex > lastApplied，则 lastApplied 递增，并将 log[lastApplied] 应用到 状态机中

如果接收到的 RPC 请求 或响应中，任期号 T > currentTerm，则令 currentTerm = T，并切换为 跟随者状态。

跟随者：

响应来自候选人 和 领导人的请求

如果在 超过选举超时 时间的 情况之前 没有收到 当前领导人（即 该领导人 的任期 需要 与这个跟随者的 当前任期相同）的 心跳/附加日志，或者是 给某个 候选人投了票，就自己变成 候选人。

候选人

在转变为 候选人 后立即开始 选举过程

- 自增当前的 任期号 (currentTerm)
- 给自己投票
- 重置选举 超时 计时器
- 发送 请求投票的 RPC 给其他所有 服务器

如果接收到 大多数服务器的 选票，那么就变成领导人

如果接收到 来自 新的 领导人的 附加日志 (AppendEntries) RPC，则转变为 跟随者

如果选举过程超时，则再次发起一轮选举

领导人

一旦成为领导人，发送 空的附加日志 (AppendEntries) RPC (心跳) 给其他所有的服务器，在一定的 空余时间后 不停地 重复发送，以防止 跟随者 超时。

如果接收到 来自 客户端的请求，附加条目 到 本地日志中，在条目 被 应用到 状态机后响应客户端。

如果对于 一个跟随者，最后日志条目的 索引值 大于 等于 nextIndex (lastLogIndex >= nextIndex)， 则发送从 nextIndex 开始的所有日志 条目：

- 如果成功：更新相应跟随者的 nextIndex 和 matchIndex
- 如果因为日志 不一致 而失败，则 nextIndex 递减并重试

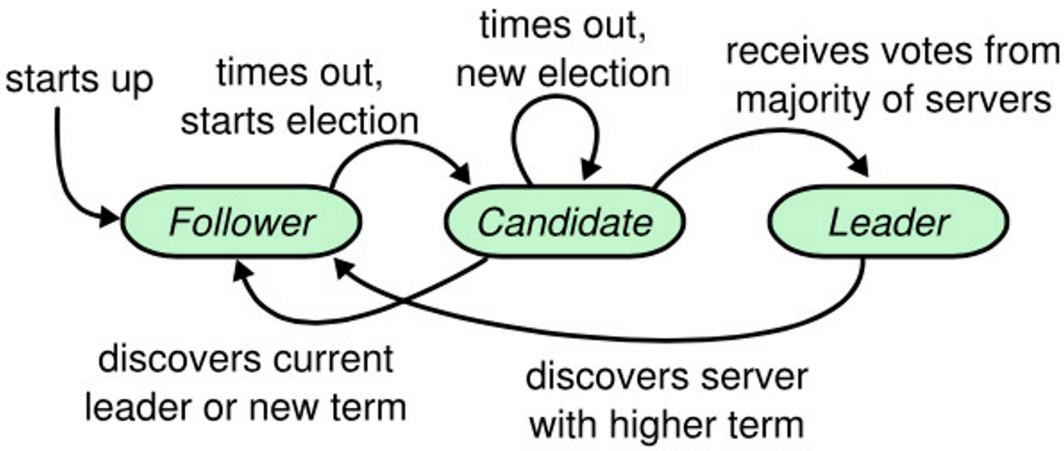
假设存在 N 满足 N > commitIndex，使得大多数的 matchIndex[i] >= N 以及 log[N].term == currentTerm 成立，则令 commitIndex = N

特性	解释
选举安全特性	对于一个给定的 任期号，最多只有一个 领导人会被选举出来
领导人只附加原则	领导人绝对不会删除 或者 覆盖 自己的日志，只会增加
日志匹配原则	如果2个日志在某一相同索引位置日志条目的任期号相同，那么 我们就认为这2个日志 从头到该索引 位置之间的 内容完全一致

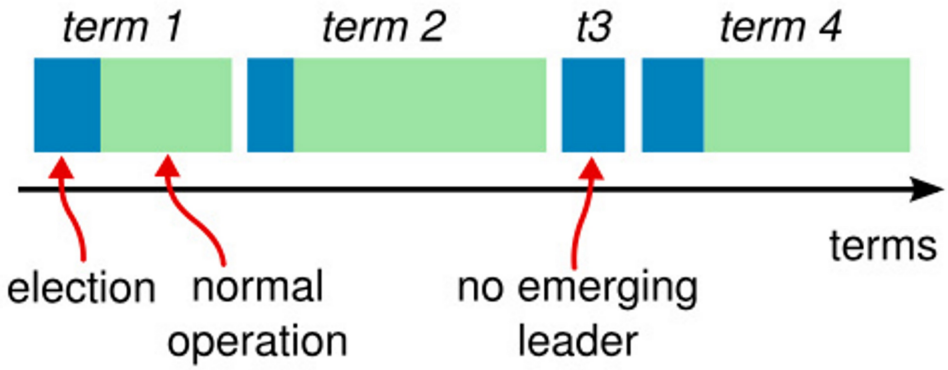
领导人完全特性	如果某个日志条目 在某个任期号中已经被提交，那么这个条目必然出现在更大任期号的 所有领导人中
状态机安全特性	如果某一服务器已将给定索引位置的 日志条目应用到 其状态机中，则其他任何服务器在该 索引位置不会应用不同的日志条目

Raft基础
一个 Raft 集群包含 若干个服务器节点。
5个服务器节点 是一个 典型的例子，这允许 整个系统 容忍 2个节点 失效。
任何时刻，每个服务器都处于 这3个 状态之一： 领导人，跟随者，候选人。
通常情况下， 系统中 只有一个 领导人 并且 其他的 节点 全部都是 跟随者。
跟随者是 被动的： 他们不发送任何 请求，只是简单地 响应 来自领导人 或候选人的 请求。
领导人处理所有的 客户端请求（如果 一个客户端 和 跟随者联系， 那么 跟随者 会把请求重定向 给 领导人）
第三种状态，候选人，是在 选举新领导人 时使用的。

下面展示了 这些状态 和 它们之间的 转换关系



上图是 服务器状态。跟随者值响应来自 其他服务器的请求。如果跟随者收不到消息，那么它就会变成 候选人 并发起一次 选举。 获得 集群中 大多数选票的 候选人 将成为 领导人。在一个任期内，领导人一直都是 领导人，直到 自己宕机。



上图中， 时间被划分为 一个个的任期，每个 任期始于一次选举。 在选举成功后，领导人会

管理 整个集群直到 任期结束。有时候 选举会失败，那么这个任期 就会没有 领导人而结束。任期之间的切换 可以在不同的时间 不同的服务器上 观察到。

。。上面说 直到自己宕机， 这个任期 是永久的吗？ 应该是的。 除非leader 主动不发送心跳给 follower
。。不，下面说到了， 领导人 任期号过期。

Raft 把时间 分隔成 任意长度的 任期。任期用连续的 整数标记。每一段任期 从 一次选举开始，一个或多个 候选人 尝试 称为 领导人。如果一个 候选人 赢得了 选举，那么 它就在 接下来的 任期内 充当领导人的职责。
某些情况下， 一次选举过程 会造成 选票的瓜分。在这种情况下，这一任期 会以 没有领导人结束， 一个新的任期（和一次新的选举）会很快 重新开始。 Raft 保证 在一个 给定任期内，最多只有一个 领导人。

不同的服务器节点 可能多次观察到 任期之间的切换， 但在某些情况下，一个节点 也可能 观查不到 任何 一次 选举 或者 整个任期 全程。
任期在 Raft 算法中 充当 逻辑时钟 的作用，任期使得 服务器可以 检测一些 过期的信息：比如 过期的领导人。每个节点存储一个 当前任期号，这个 编号 在 整个 时期内 单调递增。每当服务器 之间 通信的时候 都会交换 当前 任期号，如果 一个服务器的 当前任期号 比其他人小，那么 它会 更新自己的 任期号 到 较大的任期号值。如果一个 候选人 或者 领导人 发现 自己的任期号过期了，那么 它会立即 恢复成 follower。如果 一个节点 收到 一个 包含 过期 任期号的 请求，那么它会 直接拒绝这个请求。

Raft 算法中 服务器节点之间 通信使用 远程过程调用(RPC)，并且 基本的一致性算法 只需要2种 类型的 RPC。 请求投票(RequestVote) RPC 由 候选人 在选举期间发起， 然后 附加条目(AppendEntries) RPC 由领导人 发起，用来复制 日志 和提供一种心跳机制。
后续章节中 为了在 服务器间 传输快照 增加了 第三种 RPC。
当服务器 没有及时地 收到 RPC 的响应时，会进行重试，并且 它们能够 并行地 发起 RPC 来获得 最佳性能。

领导人选举

Raft 使用一种心跳机制 来触发领导人选举。当服务器程序启动时，它们都是 follower。
一个服务器节点 继续保持着 跟随者状态 只要它 从 领导人 或 候选人 处 接收到 有效的RPC。
领导人 周期性的向所有 跟随者 发送心跳包（即不包含 日志项内容的 附加条目(AppendEntries) RPC）来维持自己的权威。
如果一个 跟随者 在一段时间里 没有收到任何消息，也就是 选举超时，那么他就会认为系统中没有可用的 领导人，并且 发起选举 以选出 新的 领导人。

要开始一次 选举过程，跟随者先要增加 自己的 当前任期号 并转换到 候选人状态。然后 它会 并行地 向集群中的 其他服务器节点 发送 请求投票的 RPC 来给 自己投票。 候选人会继续保持着当前状态 直到 以下三件事情之一 发生：

a. 它自己赢得了这次选举
b. 其他服务器称为领导人
c. 一段时间后 没有 任何一个 获胜的人

当一个 候选人 从整个集群 的大多数服务器节点 获得了 针对 同一个任期号 的选票，那么它

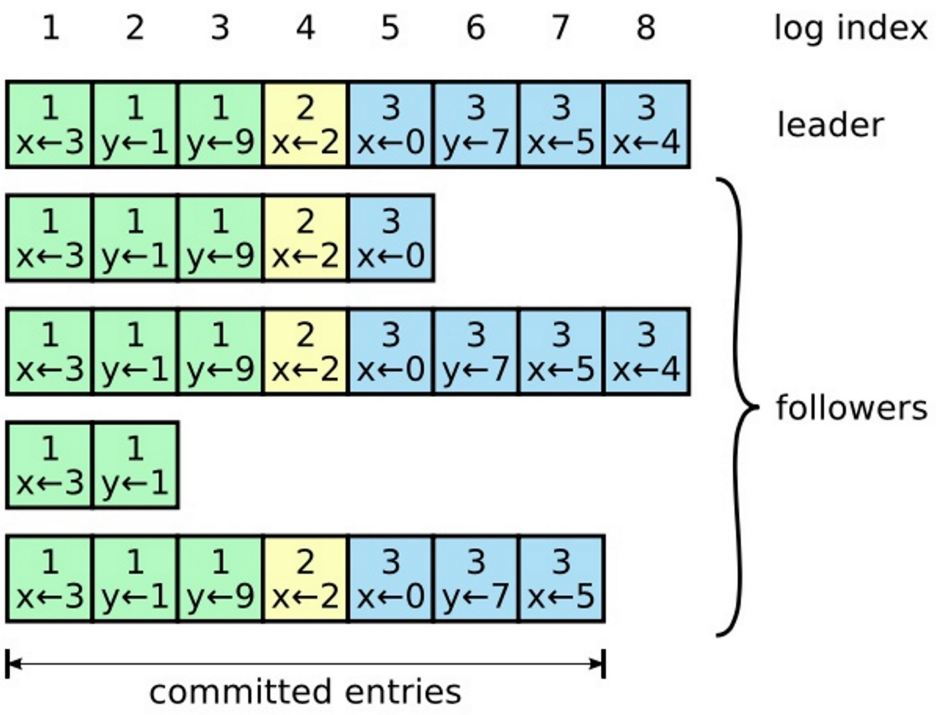
就赢得了 这次选举 并成为 领导人。每一个服务器 最多会 对一个任期号 投出一张 选票，按照 先到先投 的原则(还有一点额外的限制)。
要求大多数选票的规则 确保了 最多只会有一个 候选人赢得此次选举。 一旦候选人赢得了选举，它就立刻成为 领导人。然后 它会向 其他服务器发送 心跳信息 来建立自己的 权威 并阻止 发起新的选举。

在等待投票的时候，候选人 可能会从 其他的服务器接收到 声明 它是 领导人的 附加条目(AppendEntries) RPC。 如果这个领导人的 任期号 (包含于此次RPC中) 不小于 候选人 当前的 任期号，那么 候选人 会承认 领导人 合法 并 回到 follower 状态。如果 此次RPC 中的任期号 比自己小，那么 候选人 会拒绝 这次 RPC 并继续保持 候选人状态。

第三种可能的结果 是 没有人赢得选举，如果有多个 follower 同时成为 候选人，那么 选票可能被瓜分 以至于 没有候选人可以赢得 大多数人的 支持。 这种情况发生时，每个 候选人都会超时，然后通过 增加 当前任期号 来开始 一轮新的选举。 然而，没有其他机制的话，选票可能会被无限 重复瓜分。

Raft 算法 使用 随机选举 超时 时间的 方法 来确保 很少 会发生 选票瓜分的 情况，即使发生 也能很快解决。
为了阻止 选票 起初 就被瓜分，选举超时时间 是从一个 固定的区间(例如 150-300 ms) 随机选择。 这样可以把 服务器 都分散开 以至于 大多数情况下 只有一个服务器 会选举超时；然后它赢得 选举 并在 其他服务器超时之前 发送心跳。
同样的机制被用在 选票瓜分的情况下。每一个候选人 在开始一次 选举的时候 会重置 一个随机的 选举超时时间，然后在 超时时间内 等待投票的结果；这样减少了 在新的选举中 发生投票瓜分的可能性。

日志复制
一旦一个领导人被选举出来，它就开始 为客户端提供服务。客户端的每个请求 都包含 一条被 复制状态机 执行的指令。领导人 把这条指令 作为一条新的 日志条目 附加 到 日志中去，然后 并发地 发起 附加条目 RPC 给其他的服务器，让它们复制这条 日志条目。 当这条日志条目 被安全地复制(下面会介绍)，领导人 会应用这条日志条目 到它的状态机 中 然后把执行的结果 返回给 客户端。如果 follower 崩溃 或者 运行缓慢，或者 网络丢包，领导人 会 不断地 重复尝试 附加日志条目RPC (尽管已经回复了 客户端) 直到所有 follower 都最终存储了 所有的 日志条目。



图：日志由有序序号 标记的条目组成。每个条目都包含 创建时的 任期号(图中框中的数字)，和一个 状态机需要 执行的指令。 一个条目 当可以安全地 被应用到 状态机中去的时候，就认为是可以提交了。

日志已 上图的方式组织。每个日志条目存储了 一条 状态机指令 和 从领导人 收到这条指令时的 任期号。 日志中的任期号 用来检查 是否出现不一致的情况，同时也用来保证 某些性质。每一条 日志条目 同时 也都有 一个整数索引值 来表明 它在日志中的位置。

领导人来决定 什么时候 把日志条目 应用到 状态机中 是 安全的；这种日志条目被 称为已提交。Raft算法保证 所有 已提交 的 日志条目都是 持久化的 并且 最终会被 所有可用的 状态机 执行。 在领导人 将创建的 日志条目复制到 大多数的服务器上的时候，日志条目 就会被提交。 同时，领导人的日志中 之前的 所有日志条目 也都会被 提交，包括由 其他领导人创建的条目。
领导人 跟踪了 最大的 将会被 提交的 日志项 的索引，并且 索引值 会被包含在 未来的所有附加日志 RPC (包括心跳包)，这样其他的服务器才能最终知道 领导人的提交位置。 一旦 follower 知道 一条日志条目 已经被提交，那么 它也会将这个 日志条目 应用到 本地的状态机中(按照日志的顺序)。

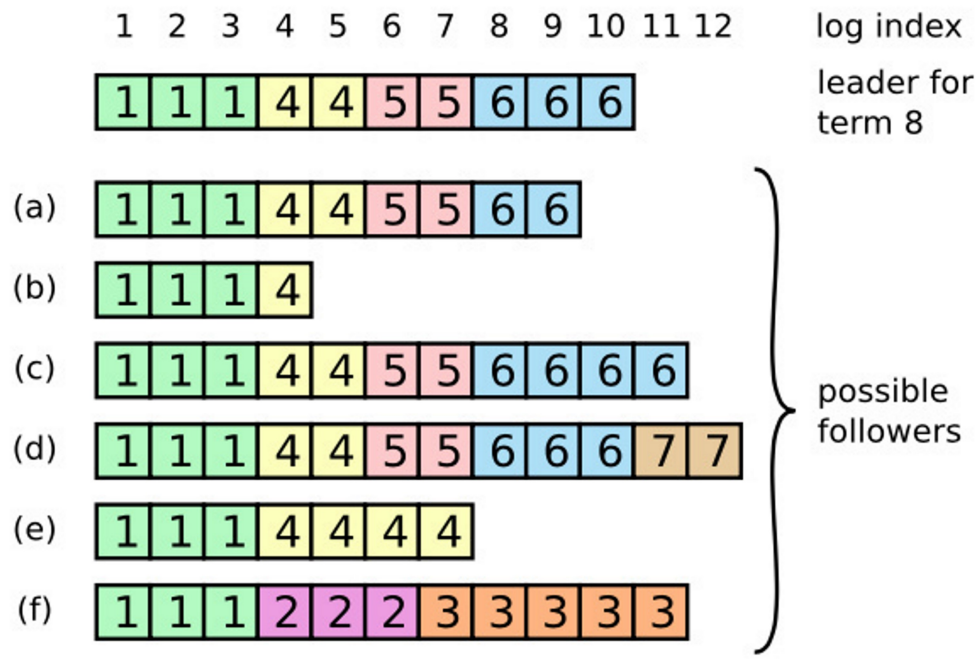
我们设计了 Raft 的日志机制来维护不同服务器日志之间的 高层次的一致性。这么做 不仅简化了 系统的行为 也使 其更具有可预测性，同时 它也是 安全性保证的 一个重要组件。
Raft 维护着 以下的特性，这些特性 共同组成了 日志匹配特性：

- 如果在不同的日志中的 2个条目拥有相同的 索引和 任期号，那么它们存储了 相同的指令。
- 如果在不同的日志中的 2个条目 拥有 相同的 索引 和任期号，那么 它们之前的 所有 日志条目 也全部相同。

第一个特性来自 这样的 一个事实，领导人 最多在 一个任期里 在 指定的 一个日志索引位置 创建 一条日志条目，同时 日志条目 在日志中的 位置 也从来 不会改变。
第二个特性 由 附加日志RPC 的一个简单的 一致性检查 所保证。在发送附加日志 RPC 的时候，领导人 会把 新的 日志条目 目前紧挨着的 条目的 索引位置 和任期号 包含在日志内。

如果跟随者 在它的日志中 找不到 包含 相同索引位置 和 任期号的 条目，那么它就会 拒绝接受新的 日志条目。一致性检查 就像一个归纳步骤：一开始空的 日志状态 肯定是满足日志匹配特性的，然后一致性检查 在日志扩展的时候 保护了 日志匹配特性。因此，每当 附加日志RPC 返回成功时，领导人 就知道 跟随者 的日志 一定是和 自己相同的了。

在正常的操作中，领导人 和 跟随者 的 日志保持一致性，所以 附加日志 RPC 的一致性检查从来不会失败。 然而，领导人 崩溃的情况下 会使得 日志 处于 不一致的状态（老的领导人可能还没有 完全复制 所有的 日志条目）。 这种不一致问题会在 领导人 和 跟随者 的一系列崩溃下 加剧。
下图展示了 follower 的日志 可能和 新的 领导人呢 不同。 跟随者 可能会丢失一些 在新的领导人中存在的 日志条目，它也可能拥有一些 领导人 没有的日志条目。丢失 或 多出 日志条目 可能会持续 多个 任期。



图：当一个 领导人成功当选时，跟个随着可能是 任何情况(a-f)。每个格子表示一个 日志条目；里面的数字表示 任期号。跟随者 可能会缺少一些日志条目(a-b)，可能会有一些 未被提交的 日志条目(c-d)，或者 2种情况都存在(e-f)。
例如，场景f可能会这样发生，某服务器在 任期 2的 时候 是领导人，已附加一些 日志条目到自己的日志中，但是 commit 之前就 崩溃了；很快这个机器被重启了，在任期3 重新被选举为领导人，并且 由增加了一些 日志条目 到 自己的日志中， 在任期2 和 任期3 的 日志 被提交前，这个服务器又宕机了，并且在 接下来的 几个任期内 一直处于 宕机状态。

在Raft 算法中，领导人是 通过 强制 跟随者 直接复制 自己的日志 来处理不一致问题的。这意味着 在跟随者中的 冲突的日志 会被 领导人的日志 覆盖。

要使得 follower 的日志 进入 和自己一直的状态，领导人 必须 找到 最后 2者 达成一致的地方，然后 删除 follower 从 那个点 之后的 所有 日志条目，并发送到 自己在 那个点 之后的日志给跟随者。 所有的这些操作都在 进行 附加日志RPC 的 一致性检查时 完成。
领导人为 每个 follower 维护一个 nextIndex，这表示 下一个 要发送给 follower 的日志条目的 索引笛子。
当一个领导人刚获得权利时，它初始化所有的 nextIndex 为 自己的最后一条日志的 index + 1。 如果一个 follower 的 日志 和 领导人不一致，那么在下一次 的附加日志 RPC 时的 一

致性检查就会失败。 在被 follower 拒绝之后，领导人 就会 减小 nextIndex 的值 进行重试。最终 nextIndex 会在某个位置 使得 leader 和 follower 的日志达成一致。

如果需要的话，算法可以通过 减少 被拒绝的 附加日志RPC 的次数 来优化。例如，当 附加日志 RPC 的请求被拒绝时，follower 可以 返回 冲突条目的 任期号 和 该任期号 对应的 最小索引地址。 借助这些信息，领导人 可以 减少 nextIndex 来 一次性跳过 该冲突任期的所有日志条目，这样 就变成了 每个任期 需要 一次 附加条目 RPC 而不是 每个条目一次。 在实践中，我们 十分怀疑 这种 优化 是否有必要， 因为失败是 很少发生 且 不太可能 一次性出现 很多的 不一致日志。

安全性

前面描述了 Raft 是如何 选举 和 复制日志的。
但是，到目前为止 描述的机制并不能 充分保障 每个 状态机 都以 相同的 顺序执行 相同的指令。例如，一个follower 可能会进入 不可用状态 同时 leader 已经提交了 若干个 日志条目， 然后 这个 follower 可能会被选举为 leader 并覆盖这些 日志条目， 因此，不同的状态机 可能会执行不同的 指令序列。
。。都 commit 了， 那状态机的 指令 还能撤销？
。不，下面有限制， 而且 如果 状态机执行， 那么就说明 大部分机器 上 都已经有这个日志了， 那么 没有日志的 机器 是无法成为 leader 的。

这一节通过 在 选举时 增加 一些 限制 来完善 Raft 算法。这一限制 充分保证了 任何的领导人 对于 给定的 任期号， 都拥有了 之前任期的 所有 被提交的 日志条目。
增加这一 选举时的限制，我们对于 提交时的规则 也更加清晰。 最终 我们将展示 对于 领导人完整特性 的 简要证明，并且 说明 该特性是 如何 引导 复制状态机 做出正确行为的。

选举限制

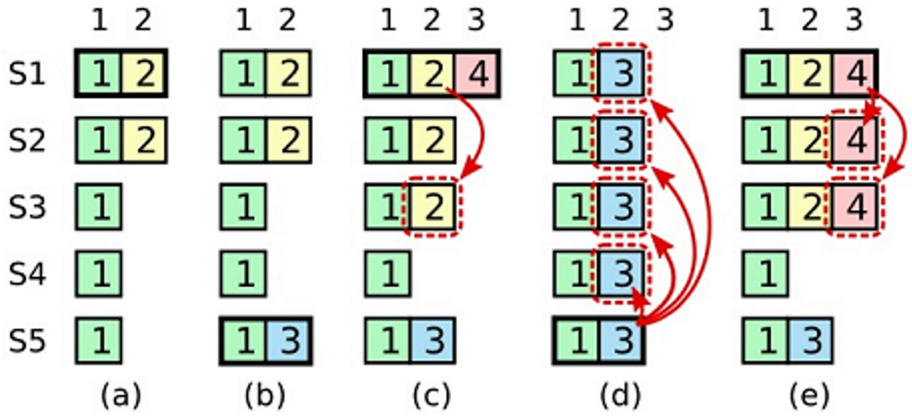
在任何 基于领导人的一致性算法中， 领导人 都必须存储 所有 已经提交的 日志条目。在某些一致性算法中，例如 Viewstamped Replication， 某个节点 即使 一开始 没有包含 所有已提交的日志条目， 它也能被选举为 leader。 这些算法 都包含了 一些 额外的 机制 来识别 丢失的日志条目 并把 它们传给 新的leader，在选举阶段 或 选举之后很快进行 。 但是，这种方法 导致 相当大的 额外机制 和复杂性。
Raft 使用一种更加简单的方法，它可以保证在选举的时候 新的领导人 拥有之前任期中的 已经提交的 日志条目，而不需要 传输这些日志条目 给领导人。这意味着 日志条目 的传输是单向的，只 从leader 传输到 follower，并且 leader 从不覆盖 自己本地日志中 已经存在的条目。

Raft 使用投票的方式 来阻止 一个 候选人赢得选举，除非 这个候选人 包含了 所有 已经提交的 日志条目。 候选人 为了 赢得 选举 必须联系 集群中的 大部分节点，这意味着每一个 已经提交的 日志条目 在这些服务器节点 中肯定存在于 至少一个节点上。 如果 候选人的 日志至少 和 大多数的服务器节点一样新，那么 它一定持有了 所有 已提交的 日志条目。
请求投票 RPC 实现了 这样的限制： RPC 中包含了 候选人的 日志信息，投票人会 拒绝那些日志没有 自己新的 投票请求。
。。只有小部分机器 有最新的 日志，它们反对， 大部分机器 日志 落后，投给了 候选人，这种怎么处理。
。。不，无所谓的， 只有小部分机器有， 那么 leader 不会 commit，不commit，就意味着没有 被 状态机执行。 所以 无所谓的。

Raft 通过比较 2份日志中 最后一条 日志条目的 索引值 和 任期号 定义 谁的日志比较新。如果任期号不同，那么 任期号大的 新。 如果 任期号相同，那么 日志长的那个 新。

提交之前任期内的日志条目
领导人 知道一条当前任期内的 日志记录 是 可以被提交的，只要它被存储到了 大多数的服务器上。如果一个领导人 在提交 日志条目之前 就崩溃了， 未来 后续的 leader 会继续尝试复制这条 日志自己了。然而，一个leader 不能断定 一个之前任期里的 日志条目被保存到 大多数服务器上的 时候 就一定已经提交了。

下图 展示了一种情况，一条已经被存储到 大多数节点上的 老日志条目，也依然有可能会被未来的领导人 覆盖掉。



图：上图的 时间序列 展示了 为什么 leader 无法决定 对老任期号的日志条目进行 提交。

a. S1 是leader，部分follower 复制了 索引位置2 的 日志条目。
b. S1 崩溃，然后 S5 在 任期3 中 通过 S3，S4，S5 的选票 成为 leader， 然后从 客户端收到了一条 不同的日志 存放到 索引2处。
c. S5 崩溃，S1 重启，选举成功，开始复制日志。这时，来自任期 2 的那条日志 已经被复制到 大多数机器上，但是 还没有 commit
d. S1 崩溃， S5 赢得选举 (通过S2,S3,S4 的选票)， 然后覆盖 它们在 索引2处的日志。
e. 反之，如果 崩溃之前，S1 把 自己主导的 新任期产生的日志复制到 大多数的机器上，那么在后面任期里 这些 新的日志条目 就会被 提交(因为 S5 不可能选举成功)

这样，在同一时刻就 同时保证了，之前的 所有 老的日志条目 就会被提交。

。。c-d，S5 怎么赢？。。因为 S1 只把 任期2 的复制出去了。 导致 S2,3,4 的最大任期是2， 而 S5的 最大任期是 3， 所以 S5可以赢。

为了消除 上图中的 情况， Raft 永远不会 通过 计算 副本数目的 方式 去提交 一个 之前任期内的 日志条目。
只有leader 当前任期里的日志条目 通过计算副本数据 可以被提交；一旦当前任期的日志条目以这种方式被提交，那么 由于日志匹配特性，之前的日志条目也都会 被间接的提交。
在某些情况下，leader 可以安全地知道 一个 老的日志条目是否已经被提交 (例如，该条目是否存储到 所有服务器上)，但是 Raft 为了简化问题 使用了 一种更加保守的方法。

当领导人复制 之前任期的日志时，Raft 会为所有的 日志 保留原始的 任期号，这在提交规则上 产生了 额外的复杂性。

在其他 一致性算法中，如果 一个新的 leader 要 重新复制 之前的 任期里的日志时，它必须使用 当前 新的任期号。
Raft 使用的方法 更容易 辨别出 日志，因为 它可以随着 时间 和日志的变化 对 日志 维护着 同一个任期 编号。
另外，和其他算法相比，Raft 中的 新leader 只需要发送 更少日志条目 (在其他算法中 必须在它们被提交之前 发送更多的 冗余日志条目 来为它们 重新编号)

。。。？感觉 不需要 消除 图里的情况啊， 而且 也没有说 怎么消除。 感觉 现在的逻辑 会自动消除的啊。

安全性论证

。。跳

通过领导人完全特性，我们就能证明状态机安全特性，即如果服务器已经在某个给定的索引值应用了日志条目到自己的状态机里，那么其他的服务器不会应用一个不一样的日志到同一个索引值上。在一个服务器应用一条日志条目到他自己的状态机中时，他的日志必须和领导人的日志，在该条目和之前的条目上相同，并且已经被提交。现在我们来考虑在任何一个服务器应用一个指定索引位置的日志的最小任期；日志完全特性保证拥有更高任期号的领导人会存储相同的日志条目，所以之后的任期里应用某个索引位置的日志条目也会是相同的值。因此，状态机安全特性是成立的。

最后，Raft 要求服务器按照日志中索引位置顺序应用日志条目。和状态机安全特性结合起来看，这就意味着所有的服务器会应用相同的日志序列集到自己的状态机中，并且是按照相同的顺序。

跟随者和候选人崩溃
到目前为止，我们都只关注了 领导人崩溃的情况。
跟随者 和 候选人 崩溃后的 处理方式 比领导人 要简单的多，并且 它们的处理方式相同。
如果跟随者 或 候选人 崩溃了， 那么后续发送 给 它们的 RPC 都会失败。 Raft 中 处理这种失败 就是 简单的 无限重试； 如果 崩溃的机器重启了， 那么 这些 RPC 就会完整的成功。如果一个服务器在 完成了一个 RPC，但是 还没有发出 响应的 时候崩溃了，那么在它 重启之后 会收到 相同的请求。 Raft 的 RPC 都是幂等的，所以重试不会造成任何问题。 例如，一个follower 如果收到 附加日志请求 但是他已经包含了 这一日志，那么它就会 直接忽略这个新的请求。

时间和可用性
Raft 的要求之一 就是 安全性不能依赖时间：整个系统不能因为 某些事件运行的 比 预期 快 或 慢 就产生 错误的结果。 但是 可用性 不可避免地 依赖于 时间。 例如， 如果消息交换比 服务器孤战 间隔时间长，候选人 将没有足够长的 时间 来赢得选举， 没有一个 稳定的 leader， Raft 将无法工作。

领导人选举 是 Raft 中 对 时间 要求最为关键的方面。 Raft 可以选举 并维持 一个 稳定的 leader，只要 系统满足下面的时间要求：
广播时间 << 选举超时时间 << 平均故障间隔时间

在这个不等式中，
广播时间 是指 从一个 服务器并行的 发送 RPC 到集群中的 其他服务器 并接收响应的 平均时间；
选举超时时间 就是 之前介绍的 选举的 超时时间限制
平均故障间隔时间 就是 对于一台服务器而言， 2次故障之间的 平均时间。

广播时间 必须必 选举超时时间 小一个量级，这样 leader 才能发送 稳定的 心跳 来阻止 follower 进入 选举状态
通过随机化 选举超时时间的 方法，这个不等式 也使得 选票瓜分的情况 变得不可能。
选举超时时间 应该 比 平均故障间隔时间 小 几个数量级， 这样 整个系统才能稳定运行。
当leader 崩溃后，整个系统 会 在 大约 相当于 选举超时时间的 时间中 不可用。

广播时间 和 平均故障间隔时间 是由系统决定的
选举超时时间 是我们决定的。 Raft 的 RPC 需要 接收方 将 信息持久化 到 存储(磁盘)中，所以 广播时间 大约是 0.5ms 到 20ms，取决于 存储的技术。 因此 选举超时时间 可能需要在 10ms 到 500ms 之间。 大多数 服务器的 平均故障间隔 是 几个月 或更久，很容易满足时间的要求。

集群成员变化

到目前为止，我们都假设集群的配置(加入到 一致性算法的服务器集合) 是固定不变的。但是在实践中，偶尔是会改变 集群的配置的，例如替换那些宕机的机器 或者 改变 复制级别。
尽管可以通过 暂停整个集群，更新所有配置，然后重启 整个集群的方式 来实现，但是 在更改的时候 集群 会不可用。 另外，如果存在手工操作步骤，那么就会有操作失误的风险。为了避免这样的问题，我们决定 自动化配置改变 并且 将纳入到 Raft 一致性算法中来。

为了让配置修改机制能够安全，那么在转换过程中 不能够 存在任何时间点 使得 2个领导人在同一个任期中 同时被选举成功。 不幸的是，任何服务器直接从 旧的配置 直接转换到 新的配置 的方案 都是 不安全的。 一次性 原子地 转换 所有服务器是不可能的，所以 在 转换期间 整个集群 存在 划分成 2个 独立的 大多数群体的 可能性。

图：。。略。。 就是 从3台服务器 变成5台， 一个候选人 获得 2台老的服务器的选票，根据旧配置(3台服务器) 赢得了选举， 另一个候选人 获得了 1台老的 + 2台新的 的选票，根据新配置(5台服务器)，赢得了 选举

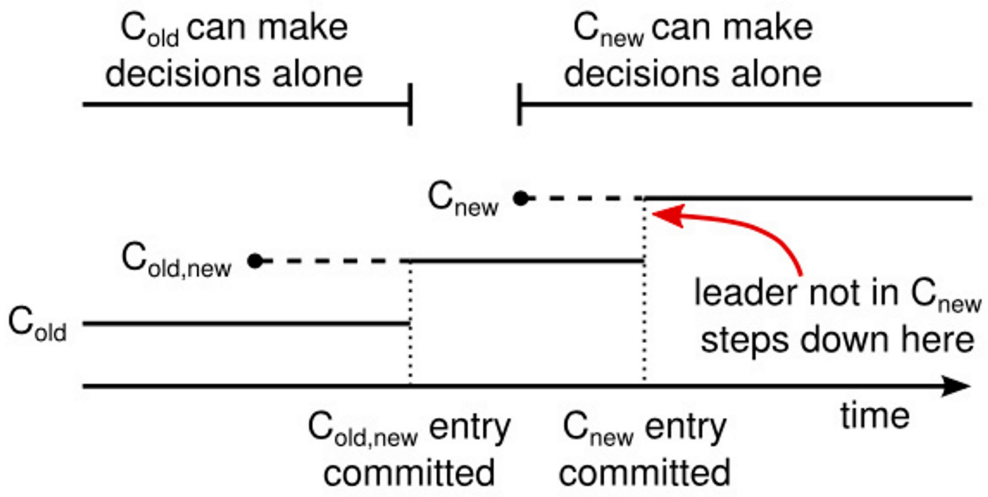
为了保证安全性，配置更改必须使用 两阶段方法。目前有很多 两阶段的实现。
例如，有些系统 在第一阶段 停掉 旧的配置 所以集群就不能处理客户端请求。 然后在 第二阶段 启用 新的配置。
在 Raft中， 集群先 切换到 一个 过度的配置，我们称之为 共同一致 (joint consensus)；一旦 共同一致 已经被提交，那么 系统就切换到 新的配置上。 共同一致 是老配置 和 新配置的 结合：

- 日志条目 被复制给 集群中的 新 老 配置的 所有服务器。
- 新 旧 配置 的服务器 都可以成为 leader
- 达成一致 (针对 选举 和提交) 需要分别在 2种配置上 获得 大多数的支持。

共同一致 允许 独立的服务器在 不影响 安全性的前提下，在不同的 时间 进行配置 转换过程。此外，共同一致 可以让集群 在配置 转换的过程中 依然响应 客户端的 请求。

集群配置在 复制日志中 以特殊的日志 条目 来存储 和 通信。
下图 展示了 配置转化 的过程。 当一个领导人 接收到 一个 改变配置 从 C-old 到 C-new 的请求， 它会为了 共同一致存储配置， 以 前面描述的 日志条目 和 副本的形式。 一旦一个服务器将 新配置日志条目 增加到 它的日志中， 它就会使用这个配置 来 做出 未来 所有的 决定 (服务器 总是 使用 最新的 配置，无论它是否 已经被 提交)。 这意味着 领导人 要使用 C-old, new 的规则 来决定 日志条目 C-old，new 什么时候 需要被提交。 如果领导人崩溃了，被选出来的 新领导人 可能是 使用 C-old 也可能是 C-new， 这取决于 赢得选举的候选人 是否已经 接收到 C-old，new 配置。 在任何情况下，C-new 配置 在这一时期 都不会单方面 做出决定。

一旦 C-old，new 被提交，那么无论是 C-old 还是 C-new， 如果不经过 另一个配置的 允许都不能 单独做出决定，并且 领导人完全特性 保证了 只有拥有 C-old，new 日志条目的 服务器 才有可能被选为 领导人。
这个时候，领导人 创建 一条 关于 C-new 配置的 日志条目 并复制给 集群 就是安全的了。再者，每个服务器在 见到 新的配置的 时候 就会 立即生效。 当新的配置 在C-new 的规则下被提交， 旧的配置 就变得 无关紧要，同时 不使用 新的 配置的 服务器 就可以 被关闭了。C-old 和 C-new 没有任何 机会 同时做出 单方面的决定，这保证了 安全性。



图： 一个配置切换的时间线。虚线表示已经被创建 但是还没有被提交的 配置日志条目，实线表示 最后被提交 的配置日志条目。 leader 首先创建 C-old,new 的配置条目 到自己的日志中，并提交到 C-old,new 中 (C-old 的大多数 和 C-new 的大多数)。 然后它创建 C-new 条目 并提交到 C-new 中的大多数。 这样就不存在 C-new 和 C-old 可以同时 做出决定的 时间点。

关于重新配置 还有3个问题 需要提出。 第一个问题是，新的服务器 可能初始化 没有存储任何的 日志条目。当这些服务器 以这种状态 加入到 集群中， 那么 它们需要 一段时间 来更新追赶，这时还不能提交 新的日志条目。为了避免这种 可用性的 间隔时间，Raft 在配置更新 之前 使用了 一种 额外的 阶段，在这个 阶段，新的服务器 以没有 投票权的 身份 加入到 集群中来 (领导人 复制日志给它们，但是不考虑 它们 是大多数)。 一旦新的服务器追赶上了 集群中的 其他机器，重新配置 可以像 上面描述的 一样处理。

第二个问题是，集群的领导人 可能不是 新配置的一员。在这种情况下， 领导人 就会在 提交 C-new 日志后 退位 (回到 跟随者状态)。 这意味着 有这样的一段时间，领导人管理者 集群，但是不包括它自己， 它复制日志 但是 不把自己 当做大多数。 当 C-new 被提交时，会发生 领导人 过度，因为 这时 是最早的 新配置 可以独立工作的 时间点 (将总是 能够在

C-new 配置下 选出 新的领导人)。 在此之前，可能只能从 C-old 中选出领导人。

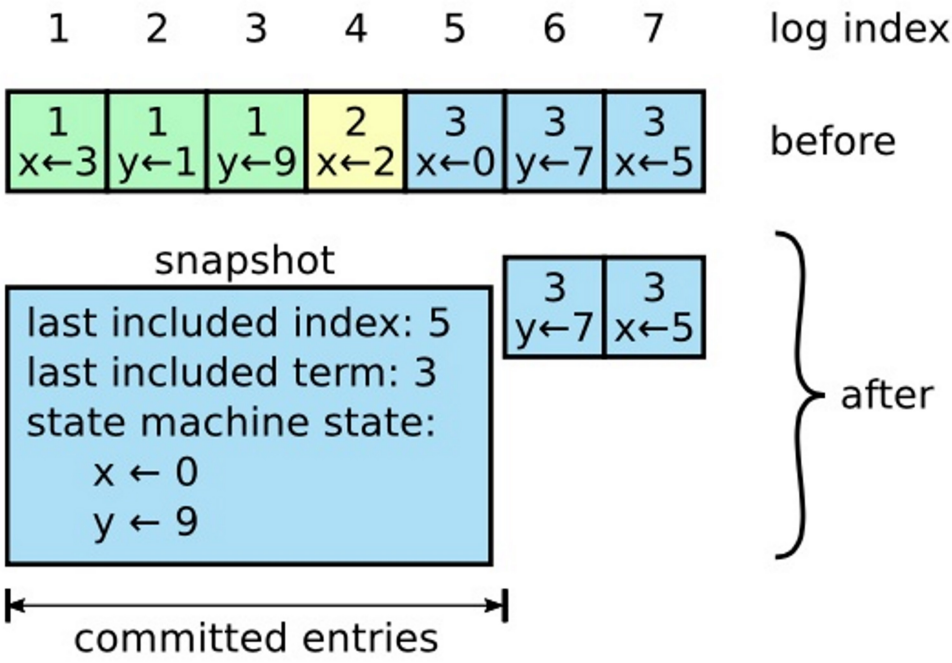
第三个问题是，移除不在 C-new 中的 服务器可能会 扰乱集群。 这些服务器 将不会再 接收到 心跳，所以 当 选举超时， 它们就会 进行 新的选举过程。它们会发送 包含 新的任期号的 请求投票RPC， 这样会导致 当前领导人 回退成 跟随者状态。新的领导人 最终 会被选出来， 但是 被移除的 服务器将会 再次超时，然后这个过程 再次重复，导致 整体可用性 降低。

为了避免这个问题，当服务器 确认当前 领导人存在时，服务器会忽略 请求投票 RPC。
确切地说，当服务器 在 当前 最小选举超时 时间内 收到一个 请求投票RPC，他不会更新当前的 任期号 或 投出选票。这不会 影响正常的 选举，每个服务器在 开始一次 选举之前，至少等待 一个 最小 选举超时 时间。 然而，这有利于 避免被移除的 服务器扰乱：如果leader能够发送 心跳给集群，那么它就不会 被 更大的 任期号 废黜。

日志压缩

快照是 最简单的 压缩方法。在快照系统中，整个系统的状态 都以 快照 的形式 写入到 稳定的 持久化存储中，然后 快照的时间点 之前的 日志 全部丢弃。 快照技术 被使用在 Chubby 和 Zookeeper 中， 接下来的章节 会介绍 Raft 中的 快照技术。

增量压缩的方法，例如 日志清理 或 日志结构合并树， 都是可行的。
这些 方法每次只对 一小部分 数据 进行操作，这样 就分散了 压缩的 负载压力。 首先，他们先选择一个 已经积累了 大量 已被删除 或 覆盖 的对象的 区域， 然后 重写 那个区域 还活跃的 对象，之后 释放 那个区域。 和简单操作 整个数据集合 的快照相比， 需要 增加 复杂的 机制 来实现。 状态机 可以实现 LSM tree 使用 和 快照 相同的接口， 但是 日志清除方法 就需要 修改 Raft。



图，一个服务器用新的快照 替换了 1 到 5 的条目， 快照值存储了 当前的状态。快照中 包含了 最后的 索引位置 和 任期号。

上图展示了 Raft 中 快照的 基础思想。每个服务器独立地 创建快照，只包括 已经被提交的日志。
主要的工作包括 将 状态机 的状态 写入到 快照中。 Raft 也包含一些 少量 的元数据 到快照中： 最后被包含索引(last included index) 是指 被快照取代的 最后的 条目 在日志中的 索引值 (状态机最后应用的日志)， 最后被包含的任期 (last included term) 指的是 该条目 的任期号。 保留这些数据 是为了 支持 快照后 紧接着的 第一个条目的 附加日志请求时 的 一致性检查， 因为 这个条目 需要 前一个日志条目的 索引值 和 任期号。 为了支持集群成员更新， 快照中 也将 最后的一次配置 作为 最后一个 条目存下来。 一旦服务器完成一次快照，它就可以 删除 最后 索引位置 之前的 所有日志 和快照了。

尽管通常服务器 都是独立地 创建快照，但是 领导人 必须 偶尔的 发送快照 给一些 落后的 follower。 这通常发生在 当 领导人 已经 丢弃了 下一条 需要 发送给 follower 的 日志条目的时候。 幸运的是，这种情况不是 常规操作： 一个 与 leader 保持同步的 跟随者 通常都有 这个 条目。 然而 一个运行非常慢 的follower 或 新加入集群的 服务器 将不会有 这个条目。 这时 让这个 follower 更新到 最新的状态的 方式 就是 通过 网络 把快照 发送给它们。

安装快照RPC
由 leader 调用 以将快照 的分块 发送给 follower， 领导人 总是 按顺序 发送分块

参数	解释
term	leader 的任期号
leaderId	leader 的id， 以便 follower 重定向 请求
lastIncludedIndex	快照中包含的 最后日志条目的 索引值
lastIncludedTerm	快照中包含的 最后日志条目的 任期号
offset	分块 在 快照中的 字节偏移量
data[]	从偏移量开始的 快照分块的 原始字节
done	如果这是 最后一个分块 则为 true

结果	解释
term	当前任期号，以便 leader 更新自己

接受者实现
1. 如果 term < currentTerm 就立即回复
2. 如果是 第一个分块 (offset 为 0) 就创建 一个新的 快照
3. 在指定 偏移量写入 数据
4. 如果 done 是false， 则继续等待 更多的数据
5. 保存快照文件，丢弃具有较小索引的 任何 现有 或 部分快照。
6. 如果现存的 日志条目 与 快照中 最后包含的日志 条目具有相同的 索引值 和 任期号，则保留 其后的 日志条目 并进行回复。
7. 丢弃整个日志
8. 使用快照重置状态机 (并加载快照的 集群配置)

领导人 使用 叫做 安装快照 的 新RPC 来发送 快照给 太落后的 follower。当跟随者 通过这种 RPC 接收到 快照时，它必须 自己决定 对于 已经存在的 日志 该如何处理。 通常 快照

会包含 没有 在 接收者 日志中存在的 信息。在这种情况下， follower 丢弃整个日志， 它全部被 快照取代，并且 可能包含 与 快照冲突的 未 提交条目。 如果接收到的 快照是 自己的日志 的 前面部分 (由于 网络重传 或错误)，那么 快照 包含的 条目 将会被 全部删除，但是快照 后面的条目 依然有效，必须保留。

。。？ 快照是自己日志的前面部分，那么完全不需要 理睬 啊。 后面的条目是 什么？ 一个快照 应该是 原子的吧。

这种快照的方式 背离了 Raft 的 强leader 原则，因为 follower 可以在 不知道 leader 的情况下 创建快照。 但是 我们认为 这种 背离是值得的。
领导人的存在，是为了解决在达成一致性的时候的冲突，但是在创建快照的时候，一致性已经达成，这时不存在冲突了，所以没有领导人也是可以的。数据依然是从领导人传给跟随者，只是跟随者可以重新组织他们的数据了。

还有两个问题影响了快照的性能。首先，服务器必须决定什么时候应该创建快照。一个简单的策略就是当日志大小达到一个固定大小的时候就创建一次快照。如果这个阈值设置的显著大于期望的快照的大小，那么快照对磁盘压力的影响就会很小了。
第二个影响性能的问题就是写入快照需要花费显著的一段时间，并且我们还不希望影响到正常操作。解决方案是通过写时复制的技术，这样新的更新就可以被接收而不影响到快照。

客户端交互
Raft 中的客户端发送所有请求给领导人。当客户端启动的时候，他会随机挑选一个服务器进行通信。如果客户端第一次挑选的服务器不是领导人，那么那个服务器会拒绝客户端的请求并且提供他最近接收到的领导人的信息（附加条目请求包含了领导人的网络地址）。如果领导人已经崩溃了，那么客户端的请求就会超时；客户端之后会再次重试随机挑选服务器的过程。

只读的操作可以直接处理而不需要记录日志。但是，在不增加任何限制的情况下，这么做可能会冒着返回脏数据的风险，因为响应客户端请求的领导人可能在他不知道的时候已经被新的领导人取代了。线性化的读操作必须不能返回脏数据，Raft 需要使用两个额外的措施在不使用日志的情况下保证这一点。首先，领导人必须有关于被提交日志的最新信息。领导人完全特性保证了领导人一定拥有所有已经被提交的日志条目，但是在他任期开始的时候，他可能不知道哪些是已经被提交的。为了知道这些信息，他需要在他的任期里提交一条日志条目。Raft 中通过领导人在任期开始的时候提交一个空白的没有任何操作的日志条目到日志中去来实现。第二，领导人在处理只读的请求之前必须检查自己是否已经被废黜了（他自己的信息已经变脏了如果一个更新的领导人被选举出来）。Raft 中通过让领导人在响应只读请求之前，先和集群中的大多数节点交换一次心跳信息来处理这个问题。可选的，领导人可以依赖心跳机制来实现一种租约的机制，但是这种方法依赖时间来保证安全性（假设时间误差是有界的）。

。。？只读的话， follower 不能确保 自己的数据 是最新的，所以 没有办法 返回，还是得让 leader 来返回数据。 但是感觉 读也好费劲。
。。还有，似乎 leader 可以永久的。。？ 只要 配置不变，网络正常，服务器不宕机。

算法实现和评估
可理解性

正确性

性能

====================================

Paxos

====================================

====================================

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========