分布式数据库理论基础

之Raft一致性协议

目

[1. 介绍4](#_Toc23646)

[2. 复制状态机5](#_Toc6251)

[3. Paxos算法的不足8](#_Toc404)

[4. 易于理解的设计10](#_Toc1756)

[5. Raft一致性算法10](#_Toc10918)

[5.1 Raft基础16](#_Toc660)

[5.2 领导人选举18](#_Toc22107)

[5.3 日志复制20](#_Toc6147)

[5.4 安全性23](#_Toc27901)

[5.4.1 选举限制24](#_Toc28011)

[5.4.2 提交之前任期内的日志条目24](#_Toc18049)

[5.4.3 安全性论证26](#_Toc25301)

[5.5 跟随者和候选人崩溃28](#_Toc13062)

[5.6 时序和可用性28](#_Toc16154)

[6. 集群成员变化29](#_Toc8278)

[7. 日志压缩32](#_Toc14826)

[8. 客户端交互36](#_Toc3140)

**摘要**

Raft 是一种管理复制日志(replicated log)的一致性算法。它提供了和 Paxos 算法相同的功能和性能，但是它的算法结构和 Paxos 不同，使得 Raft 算法更加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 将一致性算法分解成了几个关键模块，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约束来减少需要考虑的状态的数量。从一个用户研究的结果可以证明，对于学生而言，Raft 算法比 Paxos 算法更加容易学习。Raft 算法还包括一个新的机制来允许集群成员的动态改变，它利用重叠的大多数来保证安全性。

#### 介绍

一致性算法允许一组机器像一个整体一样工作，即使其中的一些机器出现故障也能继续工作下去。正因为此，一致性算法在构建可信赖的大规模软件系统中扮演着重要的角色。在过去的十年中， Paxos算法统治着一致性算法这一领域：大多数一致性算法的实现都基于Paxos或者受其影响，并且 Paxos 也成为了教学领域里讲解一致性问题时的示例。

不幸的是，尽管在降低它的复杂性方面做了许多努力，Paxos 依旧很难理解。并且，Paxos 自身的算法结构需要进行复杂的修改才能够应用到实际的系统中。这导致了工程界和学术界都对Paxos算法感到十分头疼。

在被 Paxos 折磨很久之后，我们开始寻找一种新的一致性算法，它可以为构建实际的系统和教学提供更好的基础。我们的做法是不寻常的，我们的首要目标是让它**易于理解**（可理解性）：我们是否可以在实际系统中定义一个一致性算法，并且它能够比 Paxos 算法更容易学习。此外，我们希望这种算法能凭直觉就能明白，这对于一个系统构建者来说是十分必要的。对于一个算法来说，不仅让它工作起来很重要，知道它是如何工作的更重要。

我们工作的结果是一种新的一致性算法，这个算法叫做 Raft。在设计 Raft算法的过程中，我们使用了一些特别的技巧来提升它的**可理解性**，包括算法分解（Raft主要分为领导者选举，日志复制和安全性三个模块）和减少状态机的状态（state space reduction）（相对于 Paxos，Raft 减少了非确定性和服务器互相处于非一致性的方式）。一份针对两所大学 43 个学生的研究表明，Raft 明显比 Paxos 算法更加容易理解。这些学生同时学习了这两种算法之后，和Paxos比起来，其中的33个学生能够比较好地回答 Raft 算法的问题。

Raft 算法在许多方面和现有的一致性算法很相似（主要是 Oki 和 Liskov 的 Viewstamped Replication，但是它也有一些独特的特性：

* **强领导者（Strong Leader）**：和其他一致性算法相比，Raft 使用一种更强的领导能力形式。例如，日志条目只从领导者发送给其他的服务器。这种方式简化了对复制日志的管理，并且使得 Raft 算法更加易于理解。
* **领导选举（Leader Eelection）**：Raft算法使用一个随机计时器来选取领导者。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了一点变化，使得在解决冲突时更简单快捷。
* **成员变化（Membership Change）**：Raft 为了调整集群中成员关系使用了新的联合一致性（joint consensus）的方法。在这种方法下，处于调整过程中的两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。

我们相信，Raft 算法不论出于教学目的还是作为实践项目的基础都是要比 Paxos 或者其他一致性算法要优异。它比其他算法更简单、更容易理解；它的算法描述足以实现一个现实的系统；它拥有许多开源的实现并且被许多公司所使用；它的安全特性已经被证明；并且它的效率和其他算法相比也不相上下。

接下来，这篇论文会介绍下内容：**复制状态机问题**（第2节），讨论 Paxos 的优缺点（第3节），讨论我们为了可理解性而采取的方法（第4节），阐述 Raft 一致性算法（第5~8节）。

#### 复制状态机

一致性算法是在**复制状态机**的背景下提出来的。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，因此即使有一些服务器崩溃了，这组服务器也还能继续运行。复制状态机在分布式系统中被用于解决许多有关容错的问题。例如，大规模分布式系统中通常都有一个集群领导者，像GFS、HDFS和RAMCloud ，典型应用就是一个独立的复制状态机去管理领导者选举和存储配置信息，并且在领导者宕机的情况下也要存活下来。使用复制状态机的例子有 Chubby 和 ZooKeeper。

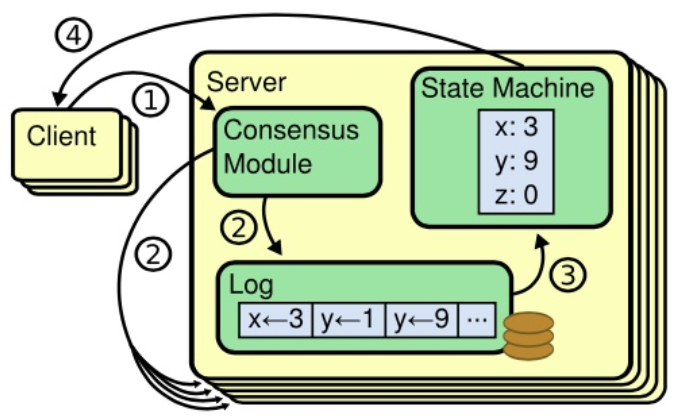


图1：复制状态机的结构.

上图是复制状态机的结构。一致性算法用来对客户端发出的状态命令所产生的日志，进行复制过程的管理。状态机从日志中处理相同顺序的相同命令，所以产生的结果也是相同的。

如图1所示，复制状态机通常都是基于复制日志来实现的。每一台服务器保存着一份日志，日志中包含一系列的命令，状态机会按日志的顺序执行这些命令。因为每一个日志都按照相同的顺序包含相同的命令，所以每一个服务器都执行相同的命令序列。因为每个状态机都是确定的，每一次执行操作都产生相同的状态和同样的序列，最后的执行结果是相同的。

如何保证复制日志的一致性（保证复制日志相同）就是一致性算法的工作了。在一台服务器上，一致性模块接收客户端发送来的命令，然后把它加入到自己的日志中。它和其他服务器上的一致性模块进行通信来确保每一个服务器上的日志最终都以相同的顺序包含相同的请求（命令），即使有一些服务器宕机了。一旦命令被正确的复制，每一个服务器的状态机都会按同样的顺序去处理它们，然后将结果返回给客户端。因此，服务器集群看起来形成一个高可靠的状态机。

应用于实际系统的一致性算法一般有以下特性：

* **确保安全性**（从来不会返回一个错误的结果），即使在所有的非拜占庭（Non-Byzantine）情况下，包括网络延迟、分区、丢包、冗余和乱序的情况下。
* **高可用性**，只要集群中有大多数机器都能运行并且可以相互通信，并且可以和客户端通信，那么这个集群就可用。因此，一个拥有 5 台机器的集群可以容忍其中的 2 台的失败（fail）。服务器停止工作了我们就认为它失败（fail）了。它们当有稳定的存储时可以从失败中恢复过来，重新加入到集群中。
* **不依赖时序保证一致性**，物理时钟错误或者极端情况下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，一条命令能够尽可能快地在集群中大多数节点对一轮远程调用作出响应时完成，一少部分慢的机器不会影响系统的整体性能。

**概念解释：**  
 一般来说，设计一个计算机系统，小到一块芯片，大到一个分布式网络，都需要考虑一定的容错性(fault tolerance)。但根据错误不同的性质，可以分为两大类：

**拜占庭错误(Byzantine fault)**。这种错误，在不同的观察者看来，会有前后不一致的表现。

**非拜占庭错误(non-Byzantine fault)**。从字面意思看，是指那些不属于前一类错误的其它错误。

这两类错误的含义并没有字面上那么好理解。

**先说说拜占庭错误**。在「拜占庭将军问题」中，叛徒的恶意行为固然是属于这一类错误的。在不同的将军看来，叛徒可能发送完全不一致的作战提议。而在计算机系统中，出现故障的节点或部件也可能表现出前后不一致的行为，虽然这并非恶意，但也属于这一类错误。比如信道不稳定，导致节点发送给其它节点的消息发生了随机错误，或者说，消息损坏了(corrupted)。再比如，在数据库系统中，commit之后的数据明明已经同步给磁盘了（通过操作系统的fsync），但由于突然断电等原因，最终数据还是没有真正落盘成功，甚至出现数据错乱。

**再看一下非拜占庭错误**。Lamport在他关于Paxos的一篇论文中也使用了non-Byzantine这个词（见《Paxos Made Simple》）。但是这个词的命名的确让人有点不好理解。在分布式系统中，如果节点宕机了，或者网络不通了，都会导致某些节点不能工作。其它节点其实没法区分这两种情况，在它看来，只是发现某个节点暂时联系不上了（即接收消息超时了）。至于是因为那个节点本身出问题了，还是网络不通了，或者是消息出现了严重的延迟，是无法区分的。而且，过一会之后，节点可能会重新恢复（或是自己恢复了，或经过了人工干预）。换句话说，对于出现这种错误的节点，我们只是收不到它的消息了，而不会收到来自它的错误消息。相反，只要收到了来自它的消息，那么消息本身是「忠实」的。

可见，拜占庭错误是更强的一类错误。在「拜占庭将军问题」中，叛徒发送前后不一致的作战建议，属于拜占庭错误；而不发送任何消息，属于非拜占庭错误。所以，解决「拜占庭将军问题」的算法，既能处理拜占庭错误，又能处理非拜占庭错误。这听起来稍微有些奇怪，不过这只是命名带来的问题。

总之，「拜占庭将军问题」的解法应该是最强的一类分布式一致性算法，它理论上能够处理任何错误。而Paxos只能处理非拜占庭错误。通常把能够处理拜占庭错误的这种容错性称为「Byzantine fault tolerance」，简称为BFT。

#### Paxos算法的不足

在过去的10年里，Leslie Lamport 的 Paxos 算法几乎已经成为了一致性算法的代名词，它是授课中最常见的算法，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，例如：基于一个单一复制日志条目（single replicated log entry）达成一致协议。我们把这个子集叫做**单决策Paxos**（single-decree Paxos）。然后通过组合多个Paxos协议的实例来完成一系列决策的达成，例如：基于一份日志（即，多个复制日志条目的集合）的multi-Paxos。Paxos 确保安全性和活跃性（liveness），并且它支持集群成员关系的变更。**它的正确性已经被证明**，在通常情况下也很高效。

不幸的是，Paxos 有两个致命的缺点。第一个是 Paxos 太难以理解。它的完整解释晦涩难懂；很少有人能完全理解，只有少数人成功的读懂了它。并且大家做了许多努力来用一些简单的术语来描述它。尽管这些解释都关注于单一决策子集问题，但仍具有挑战性。在 NSDI 2012 会议上的一次非正式调查显示，我们发现大家对 Paxos 都感到不满意，其中甚至包括一些有经验的研究员。我们自己也尝试去理解Paxos，但是一直也没能理解它，直到我们在读过几篇简化它的文章并且设计了我们自己的算法之后才完全理解了 Paxos，而整个过程花费了将近一年的时间。

我们假定 Paxos 的晦涩来源于它将单决策子集作为它的基础。**单决策（Single-decree）Paxos** 是晦涩且微妙的：它被划分为两个没有简单直观解释并且难以独立理解的情景。正因为如此，它不能很直观的让我们知道为什么单一决策协议能够工作。为**多决策 Paxos** 设计的规则又添加了很多错综复杂的规则。我们相信多决策问题能够分解为其它更直观的方式。

Paxos 的第二个缺点是它难以在实际环境中实现（它没有提供一个足够好的用来构建一个现实系统的基础）。其中一个原因是，对于多决策 Paxos （multi-Paxos） ，还没有一种被大家广泛认同的多决策问题的算法。Lamport 的描述大部分都是关于单决策 Paxos （single-decree Paxos）的；他仅仅简要描述了实现多决策Paxos的可能方法，缺少很多细节。有许多实现 Paxos 和优化 Paxos 的尝试，但是他们都和 Lamport 的描述有些出入，和Paxos的概述也不同。例如，Chubby 实现的是一个类似 Paxos 的算法，但是大部分的细节没有公开。

此外，Paxos 算法的结构也是不容易在一个实际系统中进行实现的，这是单决策问题分解带来的又一个问题。例如，从许多日志条目中选出一组日志条目，然后把它们合并成一个序列化的日志，这样做并没有带来什么太多的好处，它仅仅增加了复杂性。围绕着日志来设计一个系统是更简单、更高效的：新日志条目按照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实现方式作为它的核心（尽管它最终提出了一种弱领导者的形式来优化性能）。这种方法在只有一个决策被制定的情况下才显得有效，但是很少有现实中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 算法相似的实践。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，再然后开发了一种和 Paxos 明显不一样的结构。 这是既费时又容易出错的，并且理解 Paxos 的难度使得这个问题更加糟糕。Paxos 算法在理论上被证明是正确可行的，但是在实现上的价值就远远不足了。来自 Chubby 的实现的一条评论就能够说明：

Paxos 算法的描述与实际实现之间存在巨大的鸿沟…最终的系统往往建立在一个没有被证明的算法之上。

正因为存在这些问题，我们认为 Paxos 不仅对于系统的构建者来说不友好，同时也不利于教学。鉴于一致性算法对于大规模软件系统的重要性，我们决定试着来设计一种另外的比 Paxos 更好的一致性算法。Raft 就是这样的一个算法。

#### 易于理解的设计

设计 Raft 的目标有如下几个：

* 它必须提供一个完整的、实际的基础来进行系统构建，为的是减少开发者的工作；
* 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；
* 它的大部分操作必须是高效的；

最重要的目标是：易于理解，它必须使得大多数人能够很容易的理解；另外，它必须能让开发者有一个直观的认识，这样才能使系统构建者们在现实中去对它进行扩展。

在设计 Raft算法的过程中，有很多的点需要我们在各种备选方案中进行选择。当面临这种情况时，我们评估备选方案基于可理解性原则：每种方法的可理解性是如何的（例如，它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、暗示和含义是否容易？

我们意识到对这种可理解性的分析具有高度的主观性；尽管如此，我们使用了两种通常适用的技术来解决这个问题。第一技术是众所周知的**问题分解**：我们尽可能将问题分解成为若干个独立的、可被解决的、可解释的和可被理解的小问题。例如，在 Raft 中，我们把问题分解成为了领导选取（leader election）、日志复制（log replication）、安全（safety）和成员变化（membership changes）等几个部分（模块）。

我们采用的第二个方法是通过**减少需要考虑的状态的数量将状态空间简化**，这能够使得整个系统更加一致并且尽可能消除不确定性。特别地，日志之间不允许出现空洞，并且 Raft 限制了日志之间变成不一致的可能性。尽管在大多数情况下，我们都在试图消除不确定性，但是在有些情况下，不确定性使得算法更易理解。尤其是，随机化方法使得不确定性增加，但是它减少了状态空间。我们使用随机化来简化了 Raft 中的领导选取算法。

#### Raft一致性算法

Raft是一种用来管理复制日志的算法。Raft 通过首先选出一个领导人，然后给予他完全管理复制日志（replicated log）的责任来实现一致性。领导人接收来自客户端的日志条目，把日志条目复制到其他的服务器上，并且领导人还要告诉其它服务器“什么时候将日志条目应用到它们自己的状态机是安全的”。拥有一个领导人大大简化了复制日志的管理。例如，领导人能够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**数据流被简化成从领导人流向其他服务器**。一个领导人可以宕机，可以和其他服务器失去连接，这时一个新的领导人会被选举出来。

通过领导人方式，Raft 将一致性问题分解成了三个相对独立的子问题，这些问题会在接下来的子章节中进行讨论：

* **领导人选取**（Leader election）： 在一个领导人宕机之后必须要选取一个新的领导人（5.2节）。
* **日志复制**（Log replication）： 领导人必须从客户端接收日志然后复制到集群中的其他服务器，并且强制要求其他服务器节点的日志保持和自己相同。
* **安全性**（Safety）：在Raft中安全性的关键是在下面的图3中展示的状态机安全原则（State Machine Safety）：如果集群中有任何服务器节点已经将日志文件中给定索引位置的日志条目应用到它自己的状态机中，那么其他所有服务器不能在该索引位置应用不同的日志条目。5.4节阐述了 Raft 是如何保证这条原则的，解决方案涉及到一个对于选举机制额外的限制，这一部分会在 5.2节中说明。

在展示了一致性算法之后，本章会讨论有关可用性（availability）的一些问题和系统中候选人角色的问题。

下面是Raft算法总览：

**状态：**

|  |  |
| --- | --- |
| **状态** | **所有服务器上需持久化存储的** |
| currentTerm | 服务器最后一次知道的任期号（初始化为 0，持续递增） |
| votedFor | 在当前获得选票的候选人的id |
| log[] | 日志条目集；每一个日志条目（log entry）包含一个用户状态机执行的命令，和收到命令时的任期号 |

|  |  |
| --- | --- |
| **状态** | **所有服务器上经常变化的** |
| commitIndex | 已知的、最大的已经被提交的日志条目的索引值 |
| lastApplied | 最后被应用到状态机的日志条目索引值（初始化为 0，持续递增） |

|  |  |
| --- | --- |
| **状态** | **在领导人服务器上经常改变的（选举后重新初始化）** |
| nextIndex[] | 对于每一个服务器，需要发送给他的下一个日志条目的索引值（初始化为领导人最后索引值加一） |
| matchIndex[] | 对于每一个服务器，已经复制给他的日志的最高索引值 |

**日志条目：**

|  |  |
| --- | --- |
| **Field** | **描述** |
| term | 领导人接收客户端发来的日志条目所在的任期号 |
| index | 在日志中日志条目的位置 |
| command | 需要提交到状态机执行的指令 |

**附加日志RPC(appendLogRPC)：**

由领导人负责调用来复制日志的指令，也会用作heartbeat。

|  |  |
| --- | --- |
| **参数** | **解释** |
| term | 领导人的任期号 |
| leaderId | 领导人的 id，以便于跟随者重定向请求 |
| prevLogIndex | 新的日志条目紧随之前的索引值 |
| prevLogTerm | prevLogIndex 条目的任期号 |
| entries[] | 准备存储的日志条目（表示心跳时为空；一次性发送多个是为了提高效率） |
| leaderCommit | 领导人已经提交的日志的索引值 |

|  |  |
| --- | --- |
| **返回值** | **解释** |
| term | 当前的任期号，用于领导人去更新自己 |
| success | 跟随者包含了匹配上 prevLogIndex 和 prevLogTerm 的日志时为真 |

接收者实现：

1. 如果 term < currentTerm 就返回 false （5.1 节）
2. 如果日志(接收者的日志)在prevLogIndex 位置处的日志条目的任期号和 prevLogTerm 不匹配，则返回 false （5.3 节）
3. 如果已经存在的日志条目和新的产生冲突（索引值相同但是任期号不同），删除这一条和之后所有的 （5.3 节）
4. 附加(append)任何在已有的日志中不存在的条目
5. 如果 leaderCommit > commitIndex，令 commitIndex 等于 leaderCommit和新日志条目索引值中较小的一个

**请求投票RPC(requestVoteRPC)：**

由候选人负责调用用来征集选票（5.2节）

|  |  |
| --- | --- |
| **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 | 候选人赢得了此张选票时为真 |

接收者实现：

1. 如果 term < currentTerm 就返回 false （5.2 节）
2. 如果 votedFor为空或者就是 candidateId，并且候选人的日志至少和自己一样新，那么就投票给他（5.2 节，5.4 节）

**所有服务器需遵守的规则：**

所有服务器：

* 如果commitIndex > lastApplied，那么就 lastApplied 加一，并把log[lastApplied]应用到状态机中（5.3 节）
* 如果接收到的 RPC 请求或响应中，任期号Term > currentTerm，那么就令 currentTerm 等于 Term，并切换状态为跟随者（5.1 节）

跟随者（5.2 节）：

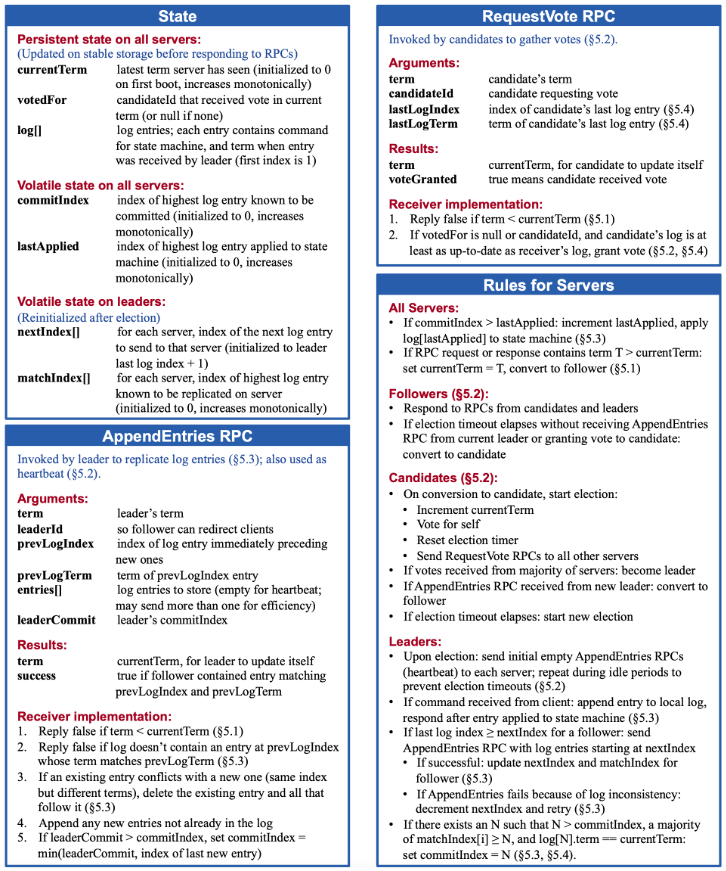
* 响应来自候选人和领导者的请求
* 如果在超过选举超时时间的情况之前都没有收到领导人的心跳，或者是候选人请求投票的，就自己变成候选人

候选人（5.2 节）：

* 在转变成候选人后就立即开始选举过程
* 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 RPC，转变成跟随者
* 如果选举过程超时，再次发起一轮选举

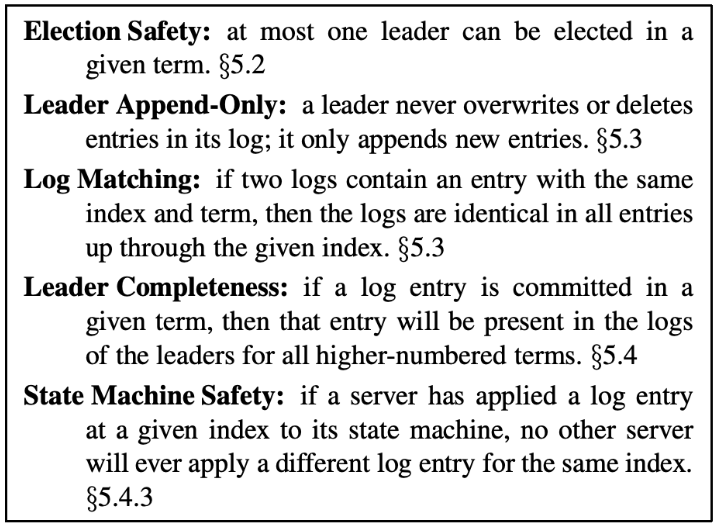
领导人：

* 一旦成为领导人：发送空的附加日志 RPC（心跳）给其他所有的服务器；在一定的空余时间之后不停的重复发送，以阻止跟随者超时（5.2 节）
* 如果接收到来自客户端的请求：附加日志条目到本地日志中，在日志条目被应用到状态机后响应客户端（5.3 节）
* 如果对于一个跟随者，最后日志条目的索引值大于等于 nextIndex，那么：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，减少 nextIndex 重试
* 如果存在一个满足N > commitIndex的 N，并且大多数的matchIndex[i] ≥ N成立，并且log[N].term == currentTerm成立，那么令 commitIndex 等于这个 N （5.3 和 5.4 节）



**图 2：一个关于 Raft 一致性算法的浓缩总结（不包括成员变换和日志压缩）。**

|  |  |
| --- | --- |
| **特性** | **解释** |
| 选举安全特性 | 对于一个给定的任期号，最多只会有一个领导人被选举出来（5.2 节） |
| 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（5.3 节） |
| 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，那么我们就认为这个日志从头到这个索引位置之间全部完全相同（5.3 节） |
| 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，那么这个日志条目必然出现在更大任期号的所有领导人中（5.4 节） |
| 状态机安全特性 | 如果一个领导人已经在给定的索引值位置的日志条目应用到状态机中，那么其他任何的服务器在这个索引位置不会提交一个不同的日志（5.4 节） |

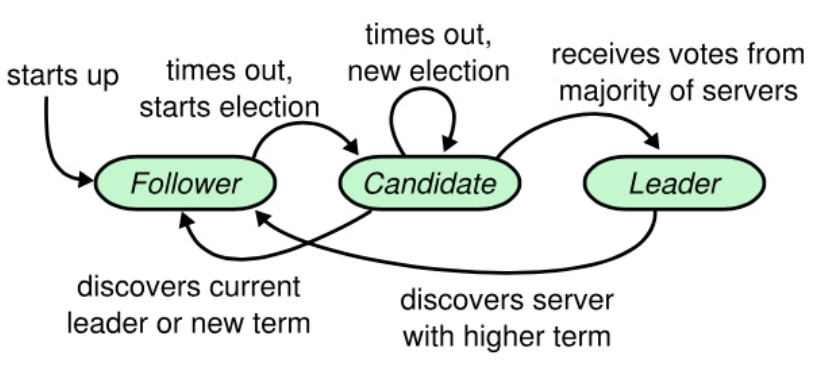


**图 3：Raft 在任何时候都保证以上的各个特性**

##### Raft基础

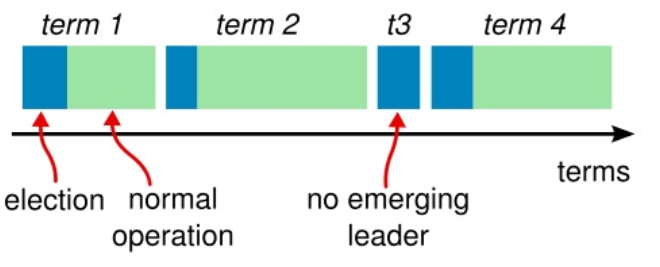
一个 Raft 集群包括若干个服务器节点；对于一个典型的 5 服务器集群，该集群能够容忍 2 台机器故障，而仍然能保持整个系统正常运行。在任何时刻，每一个服务器节点一定会处于以下三种状态中的一个：“领导人”、“跟随者”、“候选人”。在通常情况下，系统中只有一个领导人，其余的服务器是跟随者。跟随者都是被动的：他们不会发送任何请求，只是简单的响应来自领导人和候选人发出的请求。“领导人”处理所有的客户端请求（如果一个客户端向某个跟随者发送请求，那么这个跟随者会将请求重定向给领导人）。

第三种状态“候选人”是用来选取新领导人的，这种状态会在 “5.2节”进行描述。下图阐述了这些状态以及它们之间的转换；它们的转换会在下边进行讨论。



**图 4**：**服务器状态转换**

“跟随者”只响应来自其他服务器(领导者或者候选者)的请求。如果跟随者没有收到任何消息(请求)，那么他就会变成“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者。在一个任期内，领导人一直都会是领导人，直到自己宕机了。



**图5：任期（term）**

时间被分为一个个的任期（term），每一个任期的开始都是一次领导人选举。在选举成功后，领导人会管理整个集群，直到任期结束。如果选举失败(no emerging leader)，那么这个任期就会因为没有领带人而结束。任期之间的这种切换会在不同的时间、不同的服务器上观察到。

如图5所示，Raft 算法将时间划分成为任意不同长度的任期（term）。任期用连续的整数进行表示。每一段任期都是从一次选举（election）开始，就像下一节（5.2）所描述的那样，一个或多个候选人会尝试成为领导者。如果一个候选人赢得选举，它就会在该任期的剩余时间内担任领导人。在某些情况下，选票会被瓜分，有可能导致没有选出领导人。在这种情况下，这一任期就会因为没有领导人而结束；马上开始一个新的任期，并且立刻开始选举。Raft 算法保证在给定的一个任期内最多只有一个领导者。

不同的服务器节点可能会在任期内观察到多次不同的状态转换，在某些情况下，一台服务器可能看不到任何一次选举或者一个完整的任期。任期在 Raft算法中充当逻辑时钟的角色，这允许服务器节点检测一些过期的信息，比如陈旧的领导者。每一台服务器都存储着一个当前任期号（currentTerm，是一个整数），这个编号会单调的增加。当服务器之间进行通信时，会互相交换当前任期号；如果一台服务器的当前任期号比其它服务器的小，那么它会更新为较大的任期号。如果一个候选人或者领导人发现自己的任期号过时了，那么它会立刻转换为追随者状态。如果一台服务器收到的请求的任期号是过时的，那么它会拒绝此次请求。

Raft算法中服务器节点之间是通过远程过程调用（RPC）来进行通信的，基本的Raft一致性算法仅需要 2 种 RPC**。RequestVote RPC** 是由候选人在选举期间发起的（5.2节）；**AppendEntries RPC** 是领导人发起的，用来复制日志条目和提供一种心跳（heartbeat）机制（5.3节）。第7节加入了第三种 RPC 来在各个服务器之间传输快照（snapshot）。如果服务器没有及时收到 RPC 的响应时，会进行重试，并且它们能够并行的发起RPCs来获得最好的性能。

##### 领导人选举

Raft 使用一种心跳机制（heartbeat）来触发领导人的选取。当服务器程序启动时，它们都会初始化为跟随者。一台服务器只要它们能够接收到来自“领导人”或者“候选人”的有效 RPC就一直保持追随者的状态。领导人会向所有跟随者周期性发送心跳（heartbeat，不带有任何日志条目的 AppendEntries RPC）来保持它们的领导人地位。如果一个“跟随者”在一段时间里没有收到心跳信息，这叫做“选举超时”（election timeout），那么他就会假定集群中没有可用的领导人，其状态会变为候选人并且发起一次选举来选出一个新的领导人。

为了开始选举，跟随者会自增它的当前任期号并且转换到“候选人”状态。然后，它会给自己投票并且并行地向集群中的其他服务器节点发送RequestVote RPC以请求投票。候选人会一直处于该状态，直到下列三种情形之一发生：

* 它赢得了选举（赢得选举）；
* 其他服务器赢得了选举（输掉选举）；
* 一段时间后没有任何一台服务器赢得选举（即没输也没赢）

这些情形会在下面的内容中分别讨论：

**第一种情形**，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，那么他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，按照先到先服务原则（first-come-first-served）（注意：在5.4节 针对投票添加了一个额外的限制）。“大多数原则”使得在一个任期内最多有一个候选人能赢得选举（即上面的图3中提到的选举安全原则）。一旦有一个候选人赢得了选举，它就会成为领导人。然后它会向其他服务器发送心跳消息来维持自己的领导地位（建立自己的权威并且阻止新的领导人的产生）。

**第二种情形**，当一个候选人等待别人的选票时，它有可能会收到来自其他服务器发来的声明其为领导人的 AppendEntries RPC。如果这个领导人的任期（包含在它发出的 RPC请求中）比当前候选人的当前任期要大，那么候选人认为该领导人合法，并且立刻把自己的状态转换为“跟随者”。如果在这个 RPC 中的任期小于候选人的当前任期，那么候选人会拒绝这次RPC请求， 继续保持候选人状态。

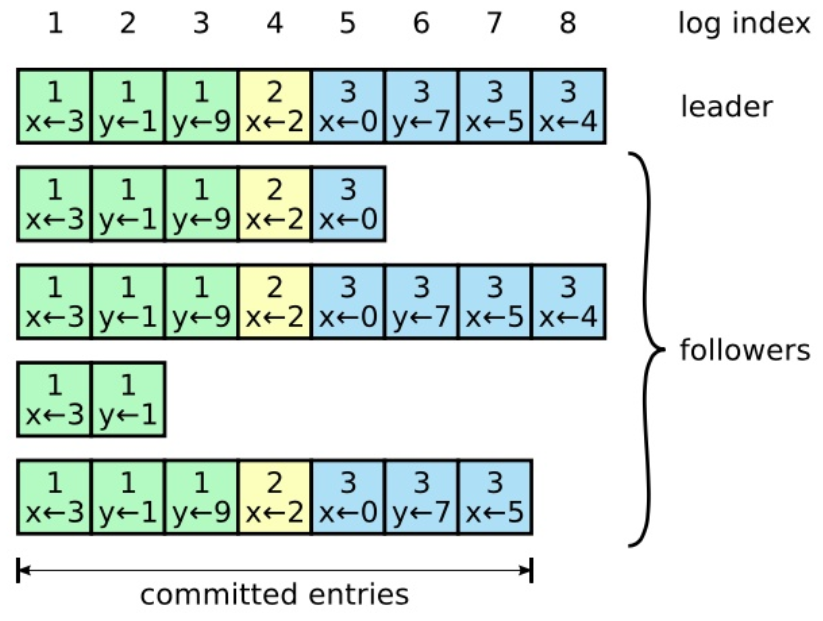
**第三种情形**是一个候选人既没有赢得选举也没有输掉选举：如果有多个跟随者同时成为候选人，那么选票会被瓜分，以至于没有候选人能获得大多数选票。当这种情形发生时，每一个候选人都会超时，然后通过自增任期号来发起一轮新的选举。然而，如果没有其它机制来分配选票的话，这种情形可能会无限的重复下去。

Raft算法使用随机的选举超时时间来确保第三种情形很少发生，就算发生也能很快地解决。为了阻止在一开始选票就被瓜分，“选举超时时间”是在一个固定区间 (例如，150~300ms) 随机选择。这种机制使得在大多数情况下只有一个服务器会率先超时，它会在其它服务器超时之前赢得选举并且向其它服务器发送心跳信息。同样的机制被用于选票一开始被瓜分的情况下。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，在超时进行下一次选举之前一直等待。这能够减小在新的选举中一开始选票就被瓜分的可能性。

##### 日志复制

一旦选出了领导人，他就开始为客户端提供服务，接收客户端的请求(命令)。每一个客户端请求都包含一条需要被复制状态机（replicated state machine）执行的命令。领导人把这条命令作为新的日志条目加入到它的日志中，然后并行地向其他服务器发起 AppendEntries RPC请求 ，以便向其它服务器复制这个日志条目。当这个条目被安全的复制之后（下面的部分会详细阐述），领导人会将这个条目应用到它自己的状态机中，然后把执行的结果返回给客户端。

如果跟随者崩溃了或者运行缓慢，再或者是网络丢包了，领导人会不断地重试 AppendEntries RPC（甚至在它已经回复了客户端之后），直到所有的跟随者最终存储了所有的日志条目。



**图6：**日志由有序的、带有序号标记的日志条目组成

我们看到，在上图中日志由有序编号的日志条目组成。每个日志条目包含它被创建时的任期号（每个方块中顶部的数字），和一个被状态机执行的命令。如果一个“日志条目”能够被状态机安全执行，就被认为可以提交了。

日志就像图6所示那样组织的。每个日志条目存储着一条被状态机执行的命令和当这条日志条目被领导人接收时的任期号。日志条目中的任期号用来检查是否出现不一致的情况，同时也用来保证图3中的某些特性。每个日志条目也包含一个整数索引来表示它在日志中的位置。

领导人决定什么时候将日志条目应用到状态机是安全的；这种日志条目被称为**已提交的**（commited）。Raft算法保证所有已提交（commited）的日志条目是持久化的，并且最终会被所有可用的状态机执行。一旦被领导人创建的日志条目已经复制到了大多数的服务器上，这个日志条目就是已提交的（例如，图6中的日志条目7）。领导人日志中之前的日志条目（即已提交日志条目之前的）都是已提交的（commited），包括由其他领导人创建的日志条目。5.4节将会讨论当领导人改变之后应用这条规则的细节，并且也讨论了这种提交的定义是安全的。领导人跟踪记录它所知道的已提交日志条目的最大索引值，并且这个索引值会包含在之后的 AppendEntries RPC 中（即包括在心跳 heartbeat 中），这样其他服务器才能知道领导人的提交位置。一旦一个跟随者知道了一个日志条目已经被提交，那么它也会将这个日志条目应用到他自己本地的状态机中(按照日志的顺序)。

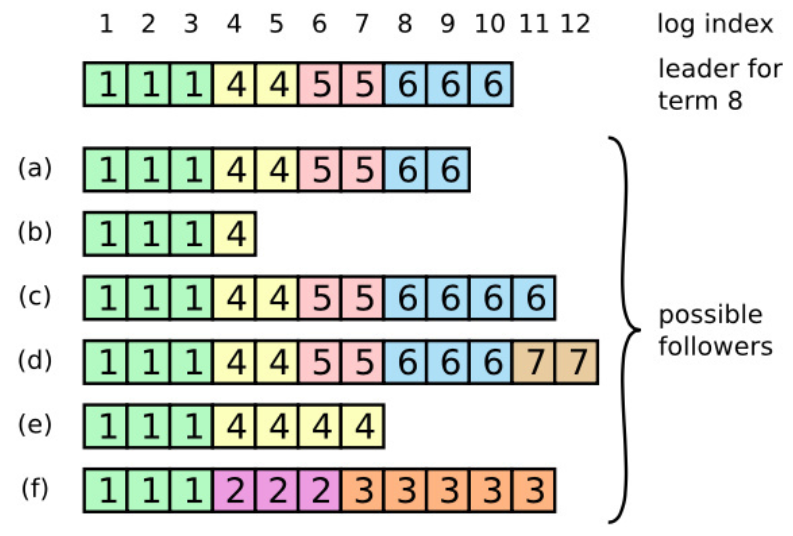
我们设计了 Raft的日志机制来保证不同服务器之间日志的一致性。这样做不仅简化了系统的行为，也使得它更可预测，同时它也是安全性保证不可或缺的一部分。Raft 保证以下特性，这些同时也构成了图3中所描述的“志匹配原则（Log Matching Property）”：

* 如果在不同日志中的两个日志条目有着相同的索引和任期号，那么它们所存储的命令是相同的。
* 如果在不同日志中的两个日志条目有着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。

第一条特性源于领导人在一个任期里在给定的一个日志索引位置最多创建一条日志条目，同时该日志条目在日志中的位置也从来不会改变。

第二条特性源于AppendEntries RPC的一个简单的一致性检查所保证。当发送一个 AppendEntries RPC 时，领导人会把新日志条目紧接着之前的日志条目的索引位置和任期号都包含在里面（即prevLogIndex和prevLogTerm）。如果跟随者没有在它的日志中找到相同索引和任期号的日志条目，那么它就会拒绝接收新的日志条目。这个一致性检查就像一个归纳步骤：一开始空的日志的状态一定是满足日志匹配原则的，然后一致性检查保证了当日志添加时的日志匹配原则。因此，每当AppendEntries RPC返回成功的时候，领导人就知道跟随者们的日志和它自己的是一致的了。

在正常的操作中，领导人和追随者们的日志保持一致，因此 AppendEntries RPC 一致性检查不会失败。然而，领导人崩溃的情况会导致日志不一致（旧的领导人可能没有完全复制完日志中的所有条目）。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。下面的图7展示了一些跟随者可能和新的领导人日志不同的情况。一个跟随者可能会丢失一些在新的领导人中有的日志条目，他也有可能拥有一些领导人没有的日志条目，或者两者都会发生。丢失的或者多出来的日志条目可能会持续多个任期。



**图7：**领导者同跟随着日志不一致的几种情况

上图中，当最上边的领导人掌权之后，跟随者的日志可能有以下情况（a~f）。一个格子表示一个日志条目；格子中的数字是任期号。一个跟随者可能会丢失一些已提交的条目（例如：a、b）；可能多出来一些未提交的条目（c, d）；或者两种情况都有（e, f）。

例如，场景 f 可能会在这样的情况下发生：如果一台服务器在任期2时是领导人，并且它往日志中添加了一些条目，然后在将它们提交之前就宕机了，之后它很快重启了，成为了任期3的领导人，又往它的日志中添加了一些条目，然后在任期2和任期3中的条目提交之前它又宕机了并且几个任期内都一直处于宕机状态。

在 Raft 算法中，领导人处理日志不一致是通过强制跟随者们直接复制领导人的日志来解决的。这就意味着，在跟随者中的冲突日志会被领导者的日志覆盖。5.4节会说明当添加了一个额外的限制之后这样的操作是安全的。

为了使得跟随者的日志同自己的一致，领导人需要找到跟随者同自己日志一致的地方，然后删除跟随者在该位置之后的条目，随后领导者将自己在该位置之后的条目发送给追随者。这些操作都在 AppendEntries RPC 进行一致性检查时完成。领导者针对每一个跟随者维护了一个**nextIndex**，它表示领导人将要发送给该跟随者的下一条日志条目的索引。

当一个领导人开始掌权时，它会将nextIndex初始化为自己的最后一条日志条目的索引+1（上图7中的 11）。如果一个跟随者的日志和领导人的不一致，那么在下一次AppendEntries RPC时的一致性检查会返回失败。在失败之后，领导人会将nextIndex递减然后重试 AppendEntries RPC。最终，nextIndex会在某个位置使得领导人和追随者的日志达成一致。这时，AppendEntries RPC 会返回成功，这时就会把跟随者中冲突的日志条目全部删除，并且添加上所缺少的领导人的日志条目。一旦 AppendEntries RPC返回成功，跟随者和领导人的日志就一致了，这样的状态会保持到该任期结束。

如果需要的话，算法还可以通过减少AppendEntries RPC失败的次数来进行优化。例如，当 AppendEntries RPC请求被拒绝的时候，跟随者可以记录下冲突日志条目的任期号和自己存储的那个任期的最早的索引。借助这些信息，领导人能够直接递减nextIndex跨过那个任期内所有的冲突条目；这样的话，一个冲突的任期需要一次 AppendEntries RPC，而不是每一个冲突条目需要一次 AppendEntries RPC。在实践中，我们怀疑这种优化是否是必要的，因为AppendEntries 一致性检查很少失败并且也不太可能出现大量的日志条目不一致的情况。

通过这种机制，一个领导人在掌权时不需要采取另外特殊的方式来恢复日志的一致性。它只需要使用一些常规的操作，通过响应 AppendEntries 一致性检查的失败能使得日志自动的趋于一致。一个领导人从来不会覆盖或者删除自己的日志（领导人只增加原则）。

日志复制机制展示了所希望的一致性特性：Raft 能够接受，复制并且应用新的日志条目，只要大多数服务器是正常的。在通常情况下，一条新的日志条目可以在一次RPC中被复制给集群中的大多数服务器；并且一个速度很慢的跟随者并不会影响整体的性能。

##### 安全性

在前面的内容中我们讨论了Raft一致性算法是如何进行领导选举和日志复制的。然而，到目前为止描述的机制还不能充分保证每一个状态机会按照相同的顺序执行同样的指令。例如，当领导人提交了若干日志条目的同时某个跟随者可能宕机了，之后这个跟随者可能会被选为了领导人，然后用新的日志条目覆盖掉了旧的那些，最后，不同的状态机可能执行不同的命令序列。

这一节通过在领导人选举部分加入了一个限制来完善了 Raft 算法。这个限制能够保证对于指定的任期，任何的领导人都拥有之前任期提交的全部日志条目（领导人完全原则）。有了这一限制，日志提交的规则就更清晰了。最后，我们提出了对于领导人完全原则的简单证明并且展示了它是如何修正复制状态机的行为的。

###### 选举限制

在所有的以领导人为基础的一致性算法中，领导人最终必须存储所有已经提交的日志条目（不多、不少、没有空洞）。在一些一致性算法中，例如：Viewstamped Replication，即使一开始没有包含全部已提交的条目也可以被选为领导人。这些算法都有一些另外的机制来保证找到丢失的条目并将它们传输给新的领导人，这个过程要么在选举过程中完成，要么在选举之后立即开始。不幸的是，这种方式大大增加了复杂性。

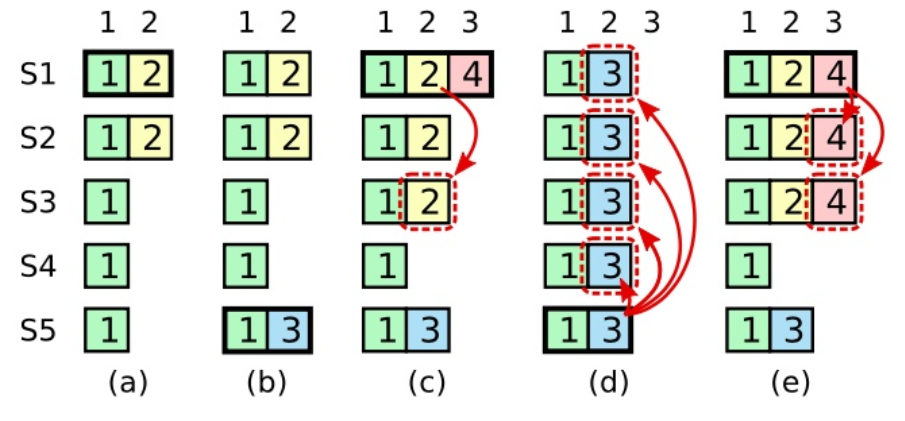
Raft 使用了一种更加简单的方法，它可以保证所有之前的任期号中已经提交的日志条目在选举的时候都会出现在新的领导人的日志中，不需要传送这些日志条目给领导人。这就意味着日志条目只有一个流向：从领导人流向追随者。领导人永远不会覆盖已经存在的日志条目。

Raft 使用投票的方式来阻止没有包含全部日志条目的服务器赢得选举。一个候选人为了赢得选举必须要和集群中的大多数节点进行通信，这就意味着每一条已经提交的日志条目最少在其中一台服务器上出现。如果候选人的日志至少和大多数服务器上的日志一样新（up-to-date，这个概念会在下边有详细介绍），那么它一定包含有全部的已经提交的日志条目。RequestVote RPC 实现了这样的限制：这个 RPC（远程过程调用）中包括候选人的日志信息（lastLogIndex和lastLogTerm），如果“投票人”自己的日志比候选人的日志要新，那么它会拒绝候选人的投票请求。

Raft 通过比较日志中最后一个条目的索引（lastLogIndex）和任期号（lastLogTerm）来决定两个日志哪一个更新。如果两个日志的任期号不同，任期号大的更新；如果任期号相同，更长的日志更新。

###### 提交之前任期内的日志条目

正如 3.5.5.3节中描述的那样，只要一个日志条目被存在了在多数的服务器上，领导人就知道当前任期内的这个日志条目可以被提交的。如果领导人在提交之前就崩溃了，之后的领导人会继续尝试复制这条日志记录。然而，领导人并不能断定，一条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可以断定它已经被提交了（可以提交，但不能判断是否已经提交）。图8说明了一种情况，一条存储在了大多数服务器上的日志条目仍然被新上任的领导人覆盖了。



**图8：为什么领导人无法决定对老任期号的日志条目进行提交**

上图：如图的时间序列说明了为什么领导人不能通过之前任期的日志条目判断它的提交状态：

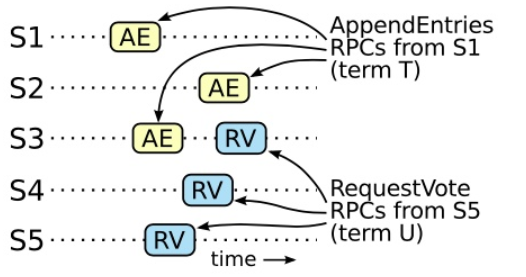
1. 中的 S1 是领导人，部分的复制了索引2上的日志条目。
2. 中 S1 崩溃了，然后S5通过 S3，S4 和自己的选票赢得了选举，然后从客户端接收了一条不一样的日志条目放在了索引2处。
3. 中 S5 崩溃了，S1 重启了，通过 S2，S3 和自己的选票赢得了选举，并且继续索引2处的复制，这时任期2的日志条目已经在大部分服务器上完成了复制，但是还并没有提交。
4. 如果在（d）时刻 S1 崩溃了，S5 会通过 S2，S3，S4 的选票成为领导人，然后用它自己在任期3的日志条目覆盖掉其他服务器在索引2出的日志条目。反之，如果在崩溃之前，S1 把自己主导的新任期里产生的日志条目复制到了大多数机器上，就像在（e）中那样，那么在后面任期里这些新的日志条目就会被提交（S5就不会赢得选举）。在这时，之前的日志条目就会正常被提交。

为了消除图8中描述的问题，Raft 永远不会通过计算副本数目的方式来提交一个之前任期内的日志条目。只有领导人当前任期中的日志条目才能通过计算副本数目来进行提交。一旦当前任期的日志条目以这种方式被提交，那么由于日志匹配原则（Log Matching Property），之前的日志条目也都会被间接的提交。在某些情况下，领导人可以安全的知道一个老的日志条目是否已经被提交（例如，通过观察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。

因为当领导人复制之前任期里的日志时，Raft为这些日志条目保留了它们原始的任期号，所以这使得 Raft 在提交规则中增加了额外的复杂性。在其他的一致性算法中，如果一个新的领导人要从之前的任期中复制日志条目，它必须要使用当前的新任期号。Raft 使用的方法更加容易辨别出日志，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号。另外，和其它的一致性算法相比，Raft 算法中的新领导人会发送更少的之前任期的日志条目（其他算法必须要发送冗余的日志条目并且在它们被提交之前来重新排序）。

###### 安全性论证

在给定了完整的 Raft 算法之后，我们现在可以更加精确的讨论领导人完整性特性。我们假设领导人完全性特性是不存在的，然后我们推出矛盾来。假设任期 T 的领导人（领导人 T）在任期内提交了一条日志条目，但是这条日志条目没有被存储到未来某个任期的领导人的日志中。设大于 T 的最小任期 U 的领导人 U 没有这条日志条目。



**图 9**：如果 S1 （任期 T 的领导者）提交了一条新的日志在它的任期里，然后 S5 在之后的任期 U 里被选举为领导人，然后至少会有一个机器，如 S3，既拥有来自 S1 的日志，也给 S5 投票了。

1. 在领导人 U 选举的时候一定没有那条被提交的日志条目（领导人从不会删除或者覆盖任何条目）。
2. 领导人 T 复制这条日志条目给集群中的大多数节点，同时，领导人U 从集群中的大多数节点赢得了选票。因此，至少有一个节点（投票者、选民）同时接受了来自领导人T 的日志条目，并且给领导人U 投票了，如图 9。这个投票者是产生这个矛盾的关键。
3. 这个投票者必须在给领导人 U 投票之前先接受了从领导人 T 发来的已经被提交的日志条目；否则他就会拒绝来自领导人 T 的附加日志请求（因为此时他的任期号会比 T 大）。
4. 投票者在给领导人 U 投票时依然保存有这条日志条目，因为任何中间的领导人都包含该日志条目（根据上述的假设），领导人从不会删除条目，并且跟随者只有在和领导人冲突的时候才会删除条目。
5. 投票者把自己选票投给领导人 U 时，领导人 U 的日志必须和投票者自己一样新。这就导致了两者矛盾之一。
6. 首先，如果投票者和领导人 U 的最后一条日志的任期号相同，那么领导人 U 的日志至少和投票者一样长，所以领导人 U 的日志一定包含所有投票者的日志。这是另一处矛盾，因为投票者包含了那条已经被提交的日志条目，但是在上述的假设里，领导人 U 是不包含的。
7. 除此之外，领导人 U 的最后一条日志的任期号就必须比投票人大了。此外，他也比 T 大，因为投票人的最后一条日志的任期号至少和 T 一样大（他包含了来自任期 T 的已提交的日志）。创建了领导人 U 最后一条日志的之前领导人一定已经包含了那条被提交的日志（根据上述假设，领导人 U 是第一个不包含该日志条目的领导人）。所以，根据日志匹配特性，领导人 U 一定也包含那条被提交的日志，这里产生矛盾。
8. 这里完成了矛盾。因此，所有比 T 大的领导人一定包含了所有来自 T 的已经被提交的日志。
9. 日志匹配原则保证了未来的领导人也同时会包含被间接提交的条目，例如图 8 (d) 中的索引 2。

通过领导人完全特性，我们就能证明图 3 中的状态机安全特性，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，并且他们的处理方式是相同的。如果跟随者或者候选人崩溃了，那么后续发送给他们的 RPCs 都会失败。Raft 中处理这种失败就是简单的通过无限的重试；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，但是还没有响应的时候崩溃了，那么在他重新启动之后就会再次收到同样的请求。Raft 的 RPCs 都是幂等的，所以这样重试不会造成任何问题。例如一个跟随者如果收到附加日志请求但是他已经包含了这一日志，那么他就会直接忽略这个新的请求。

##### 时序和可用性

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一点或者慢一点就产生了错误的结果。但是，可用性（系统可以及时的响应客户端）不可避免的要依赖于时间。例如，如果消息交换比服务器故障间隔时间长，候选人将没有足够长的时间来赢得选举；没有一个稳定的领导人，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领导人,只要系统满足下面的时间要求：

广播时间（broadcastTime） << 选举超时时间（electionTimeout） << 平均故障间隔时间（MTBF）

在这个不等式中，广播时间指的是从一个服务器并行的发送 RPCs 给集群中的其他服务器并接收响应的平均时间；选举超时时间就是在 3.5.5.2 节中介绍的选举的超时时间限制；然后平均故障间隔时间就是对于一台服务器而言，两次故障之间的平均时间。广播时间必须比选举超时时间小一个量级，这样领导人才能够发送稳定的心跳消息来阻止跟随者开始进入选举状态；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。选举超时时间应该要比平均故障间隔时间小上几个数量级，这样整个系统才能稳定的运行。当领导人崩溃后，整个系统会大约相当于选举超时的时间里不可用；我们希望这种情况在整个系统的运行中很少出现。

广播时间和平均故障间隔时间是由系统决定的，但是选举超时时间是我们自己选择的。Raft 的 RPCs 需要接收方将信息持久化的保存到稳定存储中去，所以广播时间大约是 0.5 毫秒到 20 毫秒，取决于存储的技术。因此，选举超时时间可能需要在 10 毫秒到 500 毫秒之间。大多数的服务器的平均故障间隔时间都在几个月甚至更长，很容易满足时间的需求。

#### 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其纳入到 Raft 一致性算法中来。

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。一次性自动的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两个独立的大多数群体的可能性（见图 10）。

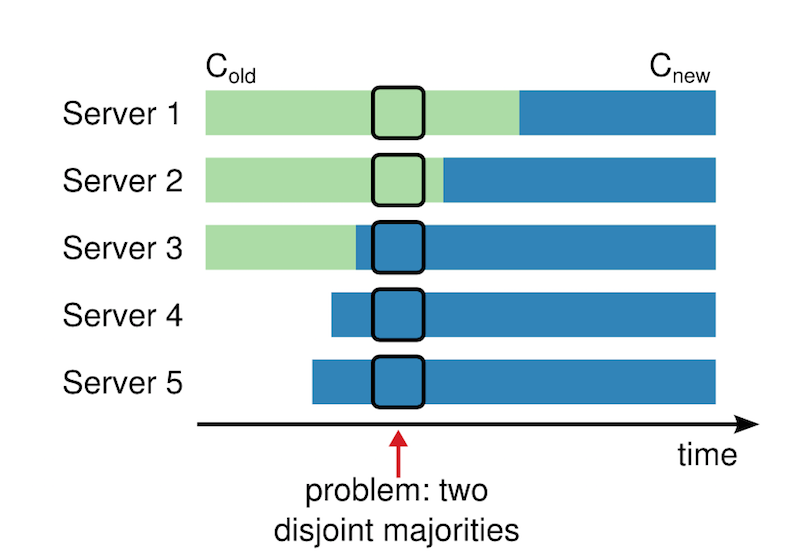


图 10：直接从一种配置转到新的配置是十分不安全的，因为各个机器可能在任何的时候进行转换。在这个例子中，集群配额从 3 台机器变成了 5 台。不幸的是，存在这样的一个时间点，两个不同的领导人在同一个任期里都可以被选举成功。一个是通过旧的配置，一个通过新的配置。

为了保证安全性，配置更改必须使用两阶段方法。目前有很多种两阶段的实现。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；然后在第二阶段在启用新的配置。在 Raft 中，集群先切换到一个过渡的配置，我们称之为共同一致；一旦共同一致已经被提交了，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新、老配置的所有服务器。
* 新、旧配置的服务器都可以成为领导人。
* 达成一致（针对选举和提交）需要分别在两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人依然响应服务器请求。

集群配置在复制日志中以特殊的日志条目来存储和通信；图 11 展示了配置转换的过程。当一个领导人接收到一个改变配置从 C-old 到 C-new 的请求，他会为了共同一致存储配置（图中的 C-old,new），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的配置日志条目增加到它的日志中，他就会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，无论他是否已经被提交）。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 配置也可能是 C-old,new 配置，这取决于赢得选举的候选人是否已经接收到了 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的做出决定。

一旦 C-old,new 被提交，那么无论是 C-old 还是 C-new，在没有经过他人批准的情况下都不可能做出决定，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举为领导人。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。如图 11，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。

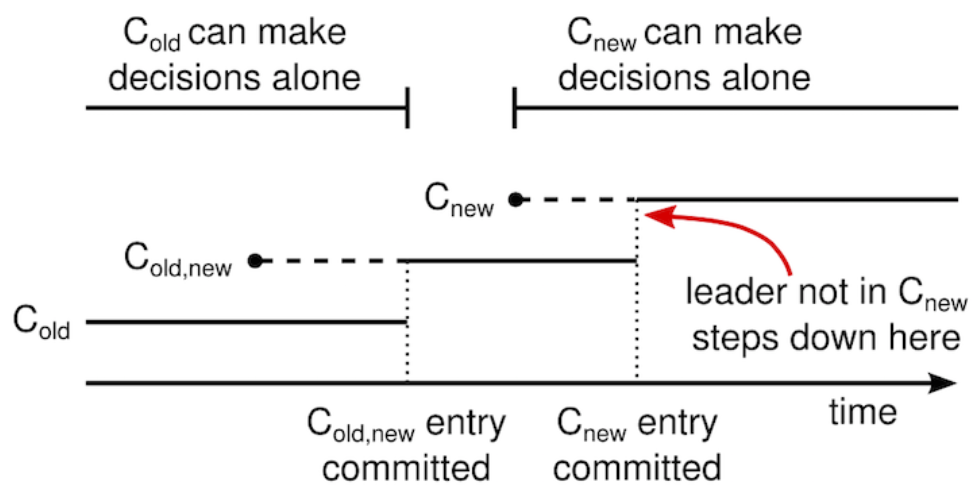


图 11：一个配置切换的时间线。虚线表示已经被创建但是还没有被提交的条目，实线表示最后被提交的日志条目。领导人首先创建了 C-old,new 的配置条目在自己的日志中，并提交到 C-old,new 中（C-old 的大多数和 C-new 的大多数）。然后他创建 C-new 条目并提交到 C-new 中的大多数。这样就不存在 C-new 和 C-old 可以同时做出决定的时间点。

在关于重新配置还有三个问题需要提出。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，那么他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 在配置更新的时候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身份加入到集群中来（领导人复制日志给他们，但是不考虑他们是大多数）。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，领导人就会在提交了 C-new 日志之后退位（回到跟随者状态）。这意味着有这样的一段时间，领导人管理着集群，但是不包括他自己；他复制日志但是不把他自己算作是大多数之一。当 C-new 被提交时，会发生领导人过渡，因为这时是最早新的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，他们就会进行新的选举过程。他们会发送拥有新的任期号的请求投票 RPCs，这样会导致当前的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，然后这个过程会再次重复，导致整体可用性大幅降低。

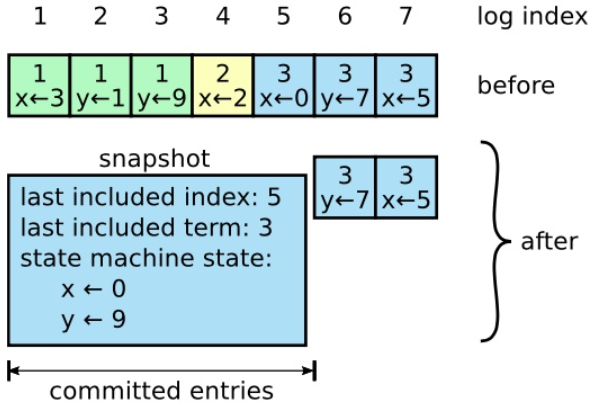
为了避免这个问题，当服务器确认当前领导人存在时，服务器会忽略请求投票 RPCs。特别的，当服务器在当前最小选举超时时间内收到一个请求投票 RPC，他不会更新当前的任期号或者投出选票。这不会影响正常的选举，每个服务器在开始一次选举之前，至少等待一个最小选举超时时间。然而，这有利于避免被移除的服务器扰乱：如果领导人能够发送心跳给集群，那么他就不会被更大的任期号废黜。

#### 日志压缩

Raft 的日志在正常操作中不断的增长，但是在实际的系统中，日志不能无限制的增长。随着日志不断增长，他会占用越来越多的空间，花费越来越多的时间来重置。如果没有一定的机制去清除日志里积累的陈旧的信息，那么会带来可用性问题。

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，然后到那个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，例如日志清理或者日志结构合并树，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的大量已经被删除或者被覆盖对象的区域，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了。



**图 12：**一个服务器用新的快照替换了从 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。

图 12 展示了 Raft 中快照的基础思想。每个服务器独立的创建快照，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：最后被包含索引指的是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日志条目的索引值和任期号。为了支持集群成员更新（第 6 节），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，他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的创建快照，但是领导人必须偶尔的发送快照给一些落后的跟随者。这通常发生在当领导人已经丢弃了下一条需要发送给跟随者的日志条目的时候。幸运的是这种情况不是常规操作：一个与领导人保持同步的跟随者通常都会有这个条目。然而一个运行非常缓慢的跟随者或者新加入集群的服务器（第 6 节）将不会有这个条目。这时让这个跟随者更新到最新的状态的方式就是通过网络把快照发送给他们。

**安装快照 RPC：**

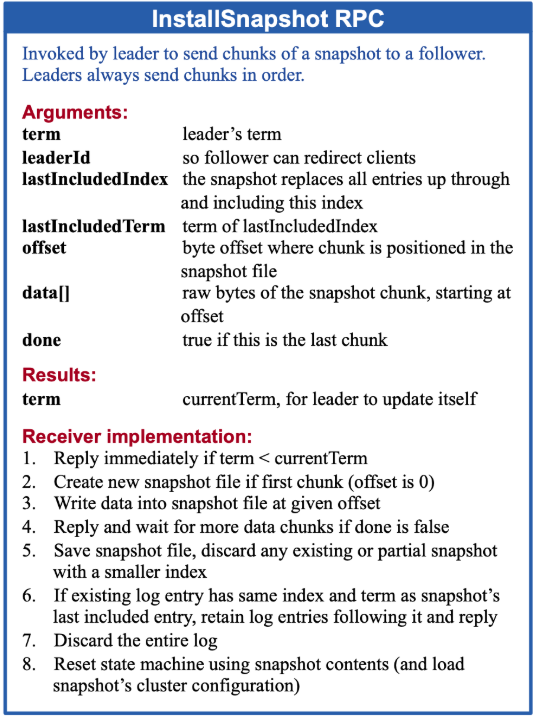
由领导人调用以将快照的分块发送给跟随者。领导者总是按顺序发送分块。

|  |  |
| --- | --- |
| **特性** | **解释** |
| term | 领导人的任期号 |
| leaderId | 领导人的 Id，以便于跟随者重定向请求 |
| 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[] | 原始数据 |
| done | 如果这是最后一个分块则为 true |

|  |  |
| --- | --- |
| **结果** | **解释** |
| term | 当前任期号（currentTerm），便于领导人更新自己 |

**接收者实现：**

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）



**图 13：**一个关于安装快照的简要概述。为了便于传输，快照都是被分成分块的；每个分块都给了跟随者生命的迹象，所以跟随者可以重置选举超时计时器。

在这种情况下领导人使用一种叫做安装快照的新的 RPC 来发送快照给太落后的跟随者；见图 13。当跟随者通过这种 RPC 接收到快照时，他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，跟随者丢弃其整个日志；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日志的前面部分（由于网络重传或者错误），那么被快照包含的条目将会被全部删除，但是快照后面的条目仍然有效，必须保留。

这种快照的方式背离了 Raft 的强领导人原则，因为跟随者可以在不知道领导人情况下创建快照。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。

我们考虑过一种替代的基于领导人的快照方案，即只有领导人创建快照，然后发送给所有的跟随者。但是这样做有两个缺点。第一，发送快照会浪费网络带宽并且延缓了快照处理的时间。每个跟随者都已经拥有了所有产生快照需要的信息，而且很显然，自己从本地的状态中创建快照比通过网络接收别人发来的要经济。第二，领导人的实现会更加复杂。例如，领导人需要发送快照的同时并行的将新的日志条目发送给跟随者，这样才不会阻塞新的客户端请求。

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。如果快照创建的过于频繁，那么就会浪费大量的磁盘带宽和其他资源；如果创建快照频率太低，他就要承受耗尽存储容量的风险，同时也增加了从日志重建的时间。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。例如，具有函数式数据结构的状态机天然支持这样的功能。另外，操作系统的写时复制技术的支持（如 Linux 上的 fork）可以被用来创建完整的状态机的内存快照（我们的实现就是这样的）。

#### 客户端交互

这一节将介绍客户端是如何和 Raft 进行交互的，包括客户端如何发现领导人和 Raft 是如何支持线性化语义的。这些问题对于所有基于一致性的系统都存在，并且 Raft 的解决方案和其他的也差不多。

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。

我们 Raft 的目标是要实现线性化语义（每一次操作立即执行，只执行一次，在他调用和收到回复之间）。但是，如上述，Raft 是可以执行同一条命令多次的：例如，如果领导人在提交了这条日志之后，但是在响应客户端之前崩溃了，那么客户端会和新的领导人重试这条指令，导致这条命令就被再次执行了。解决方案就是客户端对于每一条指令都赋予一个唯一的序列号。然后，状态机跟踪每条指令最新的序列号和相应的响应。如果接收到一条指令，它的序列号已经被执行了，那么就立即返回结果，而不重新执行指令。

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，因为领导人响应客户端请求时可能已经被新的领导人作废了，但是他还不知道。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，他可能不知道那些是已经被提交的。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）。