寻找可理解的共识算法
扩展版本

Diego Ongaro
Stanford University

John Ousterhout
Stanford University

摘要

Raft 是一种用于管理复制日志的共识算法。 它产生一个等价于(multi-)Paxos的结果，和Paxos一样高效，但是它的结构与Paxos不同； 这使得 Raft 比 Paxos 更易于理解，也为构建实际系统提供了更好的基础。 为了增强可理解性，Raft 将共识算法的关键要素（如领导人选举、日志复制和安全性）分离出来，并强制执行更强的一致性以减少必须考虑的状态数量。 一项用户研究的结果表明，Raft 比 Paxos 更容易让学生学习。 Raft 还包括一种用于更改集群成员资格的新机制，该机制使用重叠多数来保证安全。

原论文地址：https://raft.github.io/raft.pdf，本文由 Redisant 提供翻译，由于译者水平有限，失误在所难免，如有任何意见或建议，请不吝指正。

1. 绪论

共识算法允许一组机器作为一个状态一致的组工作，可以在其某些成员出现故障时幸存下来。 正因为如此，它们在构建可靠的大型软件系统方面发挥着关键作用。 Paxos 在过去十年中主导了共识算法的讨论：大多数共识算法的实现都基于 Paxos 或受其影响，Paxos 已成为用于向学生教授共识算法的主要工具。

不幸的是，Paxos 很难理解，尽管进行了多次尝试使其更易于理解。 此外，其架构需要进行复杂的更改才能支持实际系统。 因此，系统构建者和学生都在与 Paxos 作斗争。

在我们与 Paxos 斗争之后，我们着手寻找一种新的共识算法，可以为系统构建和教育提供更好的基础。 我们的方法不同寻常，因为我们的主要目标是可理解性：我们能否为实际系统定义一个共识算法，并以比 Paxos 更容易学习的方式描述它？ 此外，我们希望该算法能够促进对系统构建者至关重要的直觉的发展。 重要的不仅是让算法起作用，而且让算法起作用的原因易于理解。

这项工作的成果是一种称为 Raft 的共识算法。 在设计 Raft 时，我们应用了特定的技术来提高可理解性，包括分解（Raft 将领导者选举、日志复制和安全性分开）和状态空间缩减（相对于 Paxos，Raft 降低了不确定性程度和减少了服务器之间可能不一致的方式）. 对两所大学的 43 名学生进行的用户研究表明，Raft 比 Paxos 更容易理解：在学习了这两种算法之后，其中 33 名学生能够更好地回答有关 Raft 的问题，而不是有关 Paxos 的问题。

Raft 在很多方面与现有的共识算法相似（最著名的是Oki 和Liskov 的Viewstamped Replication），但它有几个新颖的特性：

- 强领导者：Raft 使用比其他共识算法更强的领导人形式。 例如，日志条目仅从领导者流向其他服务器。 这简化了复制日志的管理，并使 Raft 更容易理解。
- 领导者选举：Raft 使用随机定时器来选举领导者。 这只为任何共识算法已经需要的心跳增加了少量机制，同时简单快速地解决了冲突。
- 成员变更：Raft 用于变更集群中服务器集合的机制使用了一种新的联合共识方法，其中两种不同配置的大多数在转换期间重叠。 这允许集群在配置更改期间继续正常运行。

我们认为 Raft 比 Paxos 和其他共识算法更胜一筹，无论是出于教育目的还是作为代码实现的基础。它比其他算法更简单，更

容易理解；它的描述足够完整，可以满足实际系统的需要；它有几个开源的实现，并被几个公司使用；它的安全属性已经被正式规定和证明；它的效率与其他算法相当。

本文的其余部分介绍了复制状态机问题（第 2 节），讨论了 Paxos 的优缺点（第 3 节），描述了我们实现可理解性的一般方法（第 4 节），介绍了 Raft 共识算法（第 5-8 节），评估 Raft（第 9 节），并讨论相关工作（第 10 节）。

2. 复制状态机

共识算法通常出现在复制状态机的上下文中。 在这种方法中，一组服务器上的状态机计算相同状态的相同副本，并且即使某些服务器关闭也可以继续运行。 复制状态机用于解决分布式系统中的各种容错问题。 例如，具有单个集群领导者的大型系统，如 GFS、HDFS 和 RAMCloud，通常使用单独的复制状态机来管理领导者选举和存储即使领导者崩溃也必须生存的配置信息。 复制状态机的示例包括 Chubby 和 ZooKeeper。

复制状态机通常使用复制日志实现，如 Figure 1 所示。每个服务器存储一个包含一系列命令的日志，其状态机按顺序执行这些命令。 每个日志包含相同顺序的相同命令，因此每个状态机处理相同的命令序列。由于状态机是确定性的，因此每个状态机都计算相同的状态和相同的输出序列。

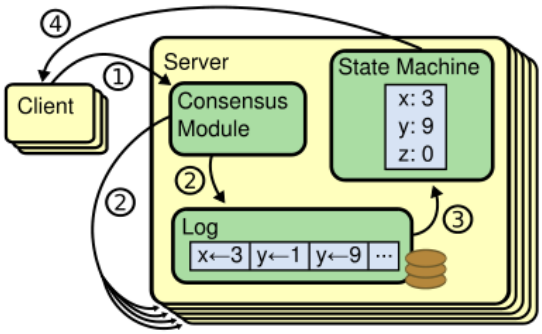

Figure 1: 复制状态机架构。 共识算法管理一个复制日志，其中包含来自客户端的状态机命令。 所有状态机处理来自日志的相同命令序列，因此它们产生相同的输出。

保持复制日志的一致性是共识算法的工作。服务器上的共识模块接收来自客户端的命令并将它们添加到其日志中。 它与其他服务器上的共识模块通信，以确保每个日志最终都以相同的顺序包含相同的请求，即使某些服务器出现故障也是如此。 一旦正确复制了命令，每个服务器的状态机就会按日志顺序处理它们，并将输出返回给客户端。 因此，这组服务器似乎形成了一个单一的、高度可靠的状态机。

实际系统的共识算法通常具有以下属性：

- 它们在所有非拜占庭条件下确保安全（永远不会返回错误的结果），包括网络延迟、分区和数据包丢失、重复和重新排序。
- 只要任何大多数服务器都可以运行并且可以相互通信以及与客户端通信，它们就可以完全发挥作用（可用）。 因此，典型的五台服务器集群可以容忍任意两台服务器发生故障。 假设服务器因停止而失败； 他们稍后可能会从稳定存储上的状态恢复并重新加入集群。
- 它们不依赖于时间来确保日志的一致性：错误的时钟和极端的消息延迟在最坏的情况下会导致可用性问题。
- 在通常情况下，一旦集群中的大多数响应了单轮远程过程调用，命令就可以完成； 少数慢速服务器不会影响整体系统性能。

3. Paxos 有什么问题？

在过去十年中，Leslie Lamport 的 Paxos 协议几乎成为共识的同义词：它是课程中最常教授的协议，大多数共识的实现都将其作为起点。 Paxos 首先定义了一个能够就单个决策达成一致的协议，例如单个复制的日志条目。 我们将这个子集称为单指令 Paxos。 然后 Paxos 将这个协议的多个实例结合起来，以促进一系列的决策，例如一个日志（multi-Paxos）。 Paxos 既保证了安全性又保证了活跃性，并且支持集群成员的变化。 它的正确性已经被证明，并且在正常情况下是有效的。

不幸的是，Paxos 有两个明显的缺点。 第一个缺点是 Paxos 异常难以理解。 众所周知，完整的解释是不透明的； 很少有人能成功地理解它，而且只有付出巨大的努力。 因此，已经有几次尝试用更简单的术语来解释 Paxos。 这些解释侧重于单指令子集，但它们仍然具有挑战性。 在 NSDI 2012 的与会者非正式调查中，我们发现很少有人对 Paxos 感到满意，即使是在经验丰富的研究人员中也是如此。 我们自己与 Paxos 作斗争； 直到阅读了几个简化的解释并设计了我们自己的替代协议后，我们才能够理解完整的协议，这个过程花了将近一年的时间。

我们假设 Paxos 的不透明性源于它选择单指令子集作为其基础。 Single-decree Paxos 密集而微妙：分为两个阶段，没有简单直观的解释，无法独立理解。 因此，很难形成关于 single-decree 协议为何有效的直觉。 multi-Paxos 的组合规则显著增加了额外的复杂性和微妙性。 我们认为，就多个决策（即日志而不是单个条目）达成共识的总体问题可以用其他更直接和明显的方式分解。

Paxos 的第二个问题是它没有为构建实际实现提供良好的基础。 一个原因是没有广泛认同的 multi-Paxos 算法。 Lamport 的描述主要是关于单指令 Paxos； 他勾勒出多 Paxos 的可能方法，但缺少许多细节。 已经有几次充实和优化 Paxos 的尝试，但这些尝试彼此不同，并且与 Lamport 的草图不同。 例如 Chubby 系统已经实现了类似 Paxos 的算法，但在大多数情况下它们的细节尚未公布。

此外，Paxos 架构对于构建实用系统来说是一个糟糕的架构； 这是单一指令分解的另一个结果。 例如，独立地选择一组日志条目然后将它们合并成一个顺序日志几乎没有什么好处； 这只会增加复杂性。 围绕日志设计系统更简单、更有效，其中新条目以受限顺序依次附加。 另一个问题是 Paxos 在其核心使用对称的点对点方法（尽管它最终建议使用弱领导形式作为性能优化）。 这在只会做出一个决定的简化世界中是有意义的，但很少有实际系统使用这种方法。 如果必须做出一系列决策，那么首先选举一个领导者，然后让领导者协调决策会更简单、更快捷。

因此，实际系统与 Paxos 几乎没有相似之处。 每个实现都是从 Paxos 开始的，发现实现它的困难，然后开发一个截然不同的架构。 这是耗时且容易出错的，理解 Paxos 的困难加剧了这个问题。 Paxos 的公式可能是证明其正确性定理的一个很好的公式，但实际实现与 Paxos 如此不同，以至于证明没有什么价值。 以下来自 Chubby 实现者的评论是典型的：

“Paxos 算法的描述与实际系统的需求之间存在显着差距。 …..最终系统将基于未经证实的协议。”

由于这些问题，我们得出结论，Paxos 没有为系统构建或教育提供良好的基础。 鉴于共识在大型软件系统中的重要性，我们决定看看是否可以设计一种比 Paxos 具有更好特性的替代共识算法。 Raft 就是这个实验的结果。

4. 为可理解性而设计

我们在设计 Raft 时有几个目标：它必须为系统构建提供一个完整且实用的基础，从而大大减少开发人员的设计工作量； 它必

须在所有条件下都是安全的，并且在典型的操作条件下可用； 并且它必须对常见操作有效。 但我们最重要的目标——也是最困难的挑战——是可理解性。 必须有可能让大量观众轻松理解该算法。 此外，必须有可能开发关于算法的直觉，以便系统构建者可以进行在现实世界实现中不可避免的扩展。

在 Raft 的设计中有很多点我们必须在替代方法中进行选择。 在这些情况下，我们根据可理解性评估备选方案：解释每个备选方案有多难（例如，它的状态空间有多复杂，它是否有微妙的含义？），以及读者完全理解该方法及其含义的难易程度？

我们认识到此类分析具有高度的主观性；尽管如此，我们还是使用了两种普遍适用的技术。 第一种技术是众所周知的问题分解方法：只要有可能，我们就将问题分成单独的部分，这些部分可以相对独立地解决、解释和理解。 例如，在 Raft 中，我们将领导选举、日志复制、安全和成员变更分开。

我们的第二种方法是通过减少要考虑的状态数量来简化状态空间，使系统更加连贯并尽可能消除不确定性。 具体来说，日志不允许有空洞，并且 Raft 限制了日志之间可能变得不一致的方式。 虽然在大多数情况下我们试图消除不确定性，但在某些情况下，不确定性实际上提高了可理解性。特别是，随机方法引入了不确定性，但它们倾向于通过以类似的方式处理所有可能的选择来减少状态空间（“选择任何一个；没关系”）。 我们使用随机化来简化 Raft 领导者选举算法。

5. Raft 共识算法

Raft 是一种用于管理第 2 节中描述的形式的复制日志的算法。Figure 3 以压缩形式总结了该算法以供参考，Figure 2 列出了该算法的关键属性； 这些图片中的元素将在本节的其余部分进行分段讨论。

Raft 通过首先选举一个领导者来实现共识，然后让领导者完全负责管理复制的日志。领导者接受来自客户端的日志条目，将它们复制到其他服务器上，并告诉服务器何时可以安全地将日志条目应用到它们的状态机。 拥有领导者可以简化复制日志的管理。 例如，领导者可以在不咨询其他服务器的情况下决定在日志中放置新条目的位置，并且数据以简单的方式从领导者流向其他服务器。 领导者可能会失败或与其他服务器断开连接，在这种情况下会选举出新的领导者。

考虑到领导方法，Raft将共识问题分解为三个相对独立的子问题，在后面的小节中讨论：

- 领导者选举（Leader election）：当现有领导者失败时，必须选择新的领导者（第 5.2 节）。
- 日志复制（Log replication）：领导者必须接受来自客户端的日志条目并在集群中复制它们，强制其他日志与其自己的日志一致（第 5.3 节）。
- 安全性（Safety）：Raft 的关键安全属性是 Figure 2 中的状态机安全属性：如果任何服务器已将特定日志条目应用到其状态机，则没有其他服务器可以对同一日志索引应用不同的命令。 5.4 节描述了 Raft 如何确保这个属性； 该解决方案涉及对第 5.2 节中描述的选举机制的额外限制。

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

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. §5.4.3

Figure 2: Raft 保证这些属性中的每一个在任何时候都是真实的。 节号表示讨论每个属性的位置。

在介绍了共识算法之后，本节讨论可用性问题和计时在系统中的作用。

5.1. Raft 基础

一个 Raft 集群包含多个服务器； 五是一个典型的数字，它允许系统容忍两次故障。 在任何给定时间，每个服务器都处于三种状态之一：领导者（leader）、追随者（follower）或候选者（candidate）。 在正常操作中，只有一个领导者，所有其他服务器都是追随者。 追随者是被动的：他们不自己发出请求，只是简单地响应领导者和候选者的请求。 领导者处理所有客户端请求（如果客户端请求跟随者，则跟随者将其重定向到领导者）。 第三种状态，候选者，用于选举新的领导者，如第 5.2 节所述。 Figure 4 显示了状态及其转换； 下面讨论这种转换。

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 3: Raft 共识算法的简要总结（不包括成员变更和日志压缩）。 左上框中的服务器行为被描述为一组独立且重复触发的规则。 §5.2 等节号表示讨论特定功能的位置。 正式规范更准确地描述了该算法。

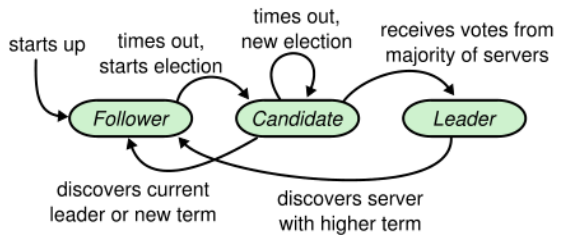

Figure 4: 服务器状态。 追随者只响应来自其他服务器的请求。 如果一个追随者没有收到任何通讯，它就成为候选者并发起选举。 从整个集群中获得大多数选票的候选者成为新的领导者。 领导者通常会一直运作到失败为止。

Raft将时间划分为任意长度的任期（term），如 Figure 5 所示。任期用连续的整数编号。每个任期都以选举开始，其中一名或多名候选者试图成为第 5.2 节所述的领导者。如果候选人赢得选举，那么它将在该任期的剩余时间内担任领导者。 在某些情况下，选举会导致分裂投票。 在这种情况下，任期将以没有领导者结束； 新任期（包括新选举）即将开始。 Raft 确保在给定任期内至多有一个领导者。

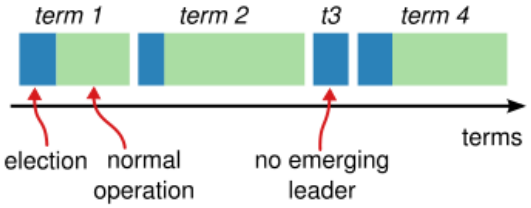

Figure 5: 时间分为任期，每个任期从选举开始。 选举成功后，一个领导者管理集群直到任期结束。 有些选举失败，在这种情况下，任期结束时不会选择领导者。 可以在不同服务器上的不同时间观察到任期之间的转换。

不同的服务器可能会在不同的时间观察任期之间的转换，并且在某些情况下，服务器可能不会观察到选举甚至整个任期。 任期在 Raft 中充当逻辑时钟，它们允许服务器检测过时的信息，例如陈旧的领导者。每个服务器存储一个当前任期号，它随时间单调增加。 每当服务器通信时，都会交换当前任期； 如果一个服务器的当前任期小于另一个，则它将其当前任期更新为较大的值。 如果候选者或领导者发现其任期已过期，则会立即恢复为追随者状态。 如果服务器收到带有陈旧任期号的请求，它会拒绝该请求。

Raft 服务器使用远程过程调用 (RPC) 进行通信，基本的共识算法只需要两种类型的 RPC。 RequestVote RPC 由候选者在选举期间发起（第 5.2 节），AppendEntries RPC 由领导者发起以复制日志条目并提供一种心跳形式（第 5.3 节）。 第 7 节添加了第三个 RPC，用于在服务器之间传输快照。如果服务器没有及时收到响应，它们会重试 RPC，并且它们会并行发出 RPC 以获得最佳性能。

5.2. 领导者选举

Raft 使用心跳机制来触发领导者选举。 当服务器启动时，它们以跟随者的身份开始。只要服务器从领导者或候选者那里收到有效的 RPC，它就会保持跟随者状态。 领导者定期向所有跟随者发送心跳（不携带日志条目的 AppendEntries RPC），以维护自己的权威。 如果跟随者在称为选举超时（election timeout）的一段时间内没有收到任何通信，那么它会假定没有可行的领导者并开始选举以选择新的领导者。

为了开始选举，追随者增加其当前任期并转换到候选者状态。 然后它为自己投票并向集群中的每个其他服务器并行发出 RequestVote RPC。 候选者一直处于这种状态，直到发生以下三种情况之一：（a）它赢得了选举，（b）其他服务器确立了自己的领导地位，或者（c）一段时间内没有赢家。这些结果将在下面的段落中分别讨论。

如果一个候选者在同一任期内获得了整个集群中大多数服务器的投票，那么它就赢得了选举。每台服务器在给定的任期内最多为一名候选者投票，以先到先得为原则（注：第5.4节对投票增加了一个额外的限制）。少数服从多数的原则保证了最多只有一名候选者能够在某一任期内赢得选举（Figure 2 中的选举安全（Election Safety））。一旦

一个候选者在选举中获胜，它就成为领导者。然后，它向所有其他服务器发送心跳信息，以建立其权威并防止新的选举。

在等待投票时，候选者可能会收到来自另一台声称是领导者的服务器的 AppendEntries RPC。 如果领导者的任期（包含在其 RPC 中）至少与候选者的当前任期一样大，则候选者承认领导者是合法的并返回到追随者状态。 如果 RPC 中的任期小于候选者的当前任期，则候选者拒绝 RPC 并继续处于候选者状态。

第三种可能的结果是候选者既不赢也不输：如果许多追随者同时成为候选者，则可以平分选票，从而没有候选者获得多数票。发生这种情况时，每个候选者都会超时并通过增加其任期并启动另一轮 RequestVote RPC 来开始新的选举。 然而，如果不采取额外措施，分裂投票可能会无限期地重复。

Raft 使用随机化的选举超时（election timeout）来确保分裂投票很少见并且可以快速解决。 为了首先防止分裂投票，选举超时是从固定间隔（例如，150-300 毫秒）中随机选择的。 这分散了服务器，因此在大多数情况下只有一个服务器会超时； 它赢得了选举，并在其他服务器超时之前发出心跳。 相同的机制用于处理分裂投票。每个候选者在选举开始时重新启动其随机选举超时，并在开始下一次选举之前等待该超时结束； 这减少了在新选举中再次出现分裂投票的可能性。 9.3 节展示了这种方法可以快速选出领导者。

选举是可理解性如何指导我们在设计方案之间做出选择的一个例子。 最初我们计划使用排名系统：为每个候选者分配一个唯一的排名，用于在竞争候选者之间进行选择。 如果一个候选者发现了另一个排名更高的候选者，它会回到追随者状态，以便排名更高的候选者更容易赢得下一次选举。我们发现这种方法在可用性方面产生了微妙的问题（如果排名较高的服务器出现故障，排名较低的服务器可能需要超时并再次成为候选者，但如果时间过早，它可能会重置选举领导者的进程). 我们对算法进行了多次调整，但每次调整后都会出现新的极端情况。 最终我们得出结论，随机重试方法更加明显和易于理解。

5.3. 日志复制

一旦领导者被选出，它就开始为客户请求提供服务。 每个客户端请求都包含要由复制状态机执行的命令。 领导者将该命令作为新条目附加到其日志中，然后向每个其他服务器并行发出 AppendEntries RPC 以复制该条目。 当条目已被安全复制（如下所述）后，领导者将条目应用于其状态机并将该执行的结果返回给客户端。 如果跟随者崩溃或运行缓慢，或者网络数据包丢失，领导者会无限期地重试 AppendEntries RPC（即使在它已经响应客户端之后）直到所有跟随者最终存储所有日志条目。

日志的组织方式如 Figure 6 所示。每个日志条目都存储一个状态机命令以及领导者收到该条目时的任期号。 日志条目中的任期号用于检测日志之间的不一致，并确保 Figure 2 中的某些属性。每个日志条目还有一个整数索引，用于标识其在日志中的位置。

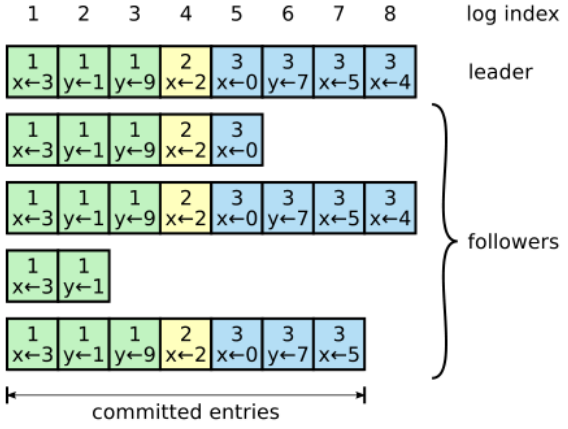

Figure 6: 日志是由条目组成的，这些条目按顺序编号。每个条目都包含创建它的任期（每个框中的数字）和状态机的命令。如果一个条目已被安全复制，那么该条目就被认为是已提交的。

领导者决定何时将日志条目应用到状态机是安全的；可以被安全地应用到状态机的条目称为已提交的。Raft 保证已提交的条目是持久的，并且最终会被所有可用的状态机执行。一旦创建条目的领导者已将其复制到大多数服务器（例如，Figure 6 中的条目 7），那么该日志就被称为已提交的（此时将该日志条目应用到状态机是安全的）。这也会提交领导者日志中所有先前的条目，包括前任领导者创建的条目。5.4 节讨论了在领导者变更后应用此规则时的一些微妙之处，并且还表明此提交定义是安全的。领导者跟踪它知道的已提交条目的最高索引，并将该索引包含在未来的 AppendEntries RPC（包括心跳）中，以便其他服务器最终找到。一旦跟随者得知日志条目已提交，它会将条目应用于其本地状态机（按日志顺序）。

我们设计了 Raft 日志机制来保持不同服务器上的日志之间的高度一致性。这不仅简化了系统的行为并使其更具可预测性，而且还是确保安全的重要组成部分。Raft 维护了以下属性，它们共同构成了 Figure 2 中的日志匹配（Log Matching ）属性：

- 如果不同日志中的两个条目具有相同的索引和任期，则它们存储相同的命令。
- 如果不同日志中的两个条目具有相同的索引和任期，则在该条目之前的所有条目都是相同的。

第一个属性源于这样一个事实，即领导者在给定任期内最多创建一个具有给定日志索引的条目，并且日志条目永远不会改变它们在日志中的位置。第二个属性由 AppendEntries 执行的简单一致性检查保证。当发送 AppendEntries RPC 时，领导者在其日志中包含紧接在新条目之前的条目的索引和任期。如果跟随者在其日志中没有找到具有相同索引和任期的条目，那么它会拒绝新条目。一致性检查作为一个归纳步骤：日志的初始空状态满足日志匹配属性，并且只要追加日志，一致性检查就会保留日志匹配属性。因此，每当 AppendEntries 成功返回时，领导者就知道了跟随者的日志与自己的日志相同。

在正常运行期间，领导者和跟随者的日志保持一致，因此 AppendEntries 一致性检查永远不会失败。但是，领导者崩溃可能会使日志不一致（旧领导者可能没有完全复制其日志中的所有条目）。这些不一致可能会导致一系列领导者和追随者崩溃。Figure 7 说明了跟随者的日志可能与新领导者的日志不同的情况。跟随者可能缺少领导者中存在的条目，它可能具有领导者中不存在的额外条目，或两者兼而有之。日志中缺失和多余的条目可能跨越多个任期。

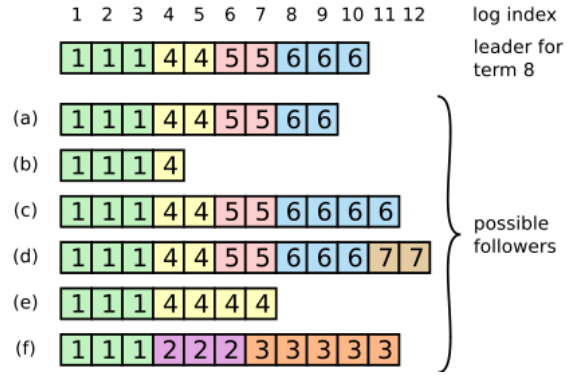

Figure 7: 当领导者上台时，追随者日志中可能会出现任何场景 (a–f)。每个方框代表一个日志条目；框中的数字是它的任期。追随者可能缺少条目 (a–b)，可能有额外的未提交条目 (c–d)，或两者都有 (e–f)。例如，如果该服务器是任期 2 的领导者，则可能会发生场景 (f)，向其日志添加多个条目，然后在提交其中任何一个之前崩溃；它很快重新启动，成为第 3 任期的领导者，并在其日志中添加了更多条目；在任期 2 或任期 3 中的任何条目被提交之前，服务器再次崩溃并保持停机几个任期。

在 Raft 中，领导者通过强制追随者的日志复制自己的日志来处理不一致。这意味着跟随者日志中的冲突条目将被领导者日志中的条目覆盖。第 5.4 节将表明，在再加上一个限制时，这是安全的。

为了使跟随者的日志与其自己的一致，领导者必须找到在跟随者的日志中和自己一致的最新日志条目，然后在跟随者日志中

删除该点之后的所有条目，并将该点之后的所有领导者日志条目发送给跟随者。 所有这些操作都在通过 AppendEntries RPC 执行的一致性检查时发生。 领导者为每个跟随者维护一个 nextIndex，这是领导者将发送给该跟随者的下一个日志条目的索引。 当一个领导者第一次掌权时，它会将所有 nextIndex 值初始化为其日志中最后一个索引之后的索引（Figure 7 中的 11）。 如果跟随者的日志与领导者的日志不一致，则在下一个 AppendEntries RPC 中，一致性检查将失败。 拒绝后，领导者递减 nextIndex 并重试 AppendEntries RPC。 最终 nextIndex 将达到领导者和跟随者日志匹配的点。 当发生这种情况时，AppendEntries RPC 将成功，它会删除跟随者日志中的所有冲突条目并追加领导者日志中的条目（如果有的话）。 一旦 AppendEntries RPC 成功，跟随者的日志与领导者的一致，并且在剩余的任期内保持这种状态。

如果需要，可以优化协议以减少被拒绝的 AppendEntries RPC 的数量。 例如，当拒绝 AppendEntries 请求时，跟随者可以包括冲突条目的任期和它为该任期存储的第一个索引。 有了这些信息，领导者可以减少 nextIndex 以绕过该任期中的所有冲突条目； 每个有冲突条目的任期都需要一个 AppendEntries RPC，而不是每个条目一个 RPC。 在实践中，我们怀疑这种优化是否必要，因为故障很少发生，而且不太可能有很多不一致的条目。

使用这种机制，领导者上台时不需要采取任何特殊措施来恢复日志一致性。 它刚刚开始正常运行，日志自动收敛以响应 AppendEntries RPC 一致性检查的失败。 领导者永远不会覆盖或删除自己日志中的条目（Figure 2 中的领导者仅附加（Leader Append-Only）属性）。

这种日志复制机制展示了第 2 节中描述的理想的共识属性：只要大多数服务器正常运行，Raft 就可以接受、复制和应用新的日志条目； 在正常情况下，可以通过单轮 RPC 将新条目复制到集群的大多数； 单个慢速跟随者不会影响性能。

5.4. 安全

前面的部分描述了 Raft 如何选举领导者和复制日志条目。 然而，到目前为止所描述的机制还不足以确保每个状态机以相同的顺序执行完全相同的命令。 例如，当领导者提交多个日志条目时，追随者可能不可用，然后它可以被选为领导者并用新的条目覆盖这些条目； 因此，不同的状态机可能会执行不同的命令序列。

本节通过添加对哪些服务器可以被选为领导者的限制来完成 Raft 算法。 该限制可确保任何给定任期的领导者都包含之前任期已提交的所有条目（Figure 2 中的领导者完整性（Leader Completeness）属性）。 考虑到选举限制，然后我们使提交规则更加精确。 最后，我们展示了领导者完整性的证明草图，并展示了它如何保证复制状态机的正确行为。

5.4.1. 选举限制

在任何基于领导者的共识算法中，领导者最终必须存储所有已提交的日志条目。 在一些共识算法中，例如 Viewstamped Replication，即使它最初不包含所有已提交的条目，也可以选出一个领导者。 这些算法包含额外的机制来识别丢失的条目并将它们传输给新的领导者，无论是在选举过程中还是之后不久。 不幸的是，这会导致相当多的额外机制和复杂性。

Raft 使用了一种更简单的方法，它保证从选举的那一刻起，每个新领导者都会出现以前任期的所有已提交条目，而无需将这些条目传输给领导者。 这意味着日志条目仅沿一个方向流动，从领导者到追随者，而领导者永远不会覆盖其日志中的现有条目。没有包含所有已提交日志条目的候选者成为不了领导者 。

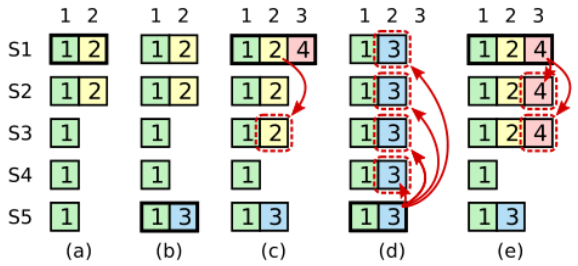

Figure 8: 一个时间序列显示了为什么领导者不能使用较早任期的日志条目来确定提交。在（a）中，S1是领导者，并部分复制了索引2的日志条目。在(b)中，S1崩溃了；S5凭借S3、S4和它自己的投票当选为第三任期的领导者，并接受了日志索引2的不同条目。在（c）中，S5崩溃了；S1重新启动，被选为领导者，并继续复制。在这一点上，任期2的日志条目已经在大多数服务器上复制，但它实际上并不能被认为是已提交的（即该日志条目不能被安全地应用到状态机）。因为如果S1像(d)那样崩溃，S5可以被选为领导者（有S2、S3和S4的投票），并用它自己的任期3的条目覆盖该条目。然而，如果S1在崩溃前在大多数服务器上复制了其当前任期的条目，如(e)，之后该条目被提交了（S5不能赢得选举）。在这一点上，日志中所有前面的条目也被提交。

Raft使用投票过程来防止候选者赢得选举，除非其日志包含所有已提交的条目。候选者必须与集群中的大多数跟随者联系才能当选，这意味着每个已提交的日志条目必须至少存在于其中一个服务器中。如果候选者的日志至少和该多数跟随者中的任何其他日志一样是最新的（这里的“最新“在下面有精确的定义），那么它将包含所有已提交的条目。RequestVote RPC实现了这一限制：RPC包括关于候选人日志的信息，如果投票人自己的日志比候选人的日志更新时，则拒绝投票。

Raft 通过比较日志中最后条目的索引和任期来确定两个日志中哪一个是最新的。 如果日志的最后条目具有不同的任期，则具有较晚任期的日志是最新的。 如果日志以相同的任期结尾，则具有更大索引的那个条目是最新的。

5.4.2. 提交以前任期的日志条目

如第 5.3 节所述，领导者知道一旦该条目存储在大多数服务器上，其当前任期的条目就会被认为是已提交的。 如果领导者在此时崩溃，未来的领导者将尝试完成复制条目。 但是，领导者不能立即断定上一个任期的条目一旦存储在大多数服务器上就已提交。 Figure 8 说明了一种情况，其中旧日志条目存储在大多数服务器上，但仍可能被未来的领导者覆盖。

为了消除 Figure 8 中的问题，Raft 限制只能通过判断大多数的方式提交当前任期的日志条目，进而对之前的日志条目间接提交（也就是说，对之前任期的日志条目不是通过通过判断大多数的方式来提交，而是通过提交当前任期的日志条目来间接提交）； 一旦以这种方式提交了当前任期的条目，则由于日志匹配（Log Matching）属性，所有先前的条目都将间接提交。 在某些情况下，领导者可以安全地断定一个较旧的日志条目已提交（例如，如果该条目存储在每个服务器上），但 Raft 为简单起见采用了更保守的方法。

Raft 在提交规则中引入了这种额外的复杂性，因为当领导者从以前的任期复制条目时，日志条目会保留其原始任期号。 在其他共识算法中，如果新领导者重新复制先前“任期”中的条目，则它必须使用新的“任期号”进行复制。 Raft 的方法使得对日志条目的推理变得更容易，因为它们随着时间的推移和跨日志保持相同的任期编号。此外，与其他算法相比，Raft 中的新领导人发送的前任期日志条目更少（其他算法必须发送冗余日志条目以重新编号，然后才能提交）。

5.4.3. 安全论证

给定完整的Raft 算法，我们现在可以更准确地论证领导者完整性（Leader Completeness）属性成立（该论证基于安全证明；参见第 9.2 节）。 我们假设领导者完整性（Leader

Completeness）不成立，然后我们证明一个矛盾。 假设任期 T 的领导者 ($leader_T$) 从其任期提交了一个日志条目 a，但该日志条目未被未来某个任期的领导者存储。 考虑领导者（$leader_U$）不存储该日志条目的最小任期 U（U > T）。

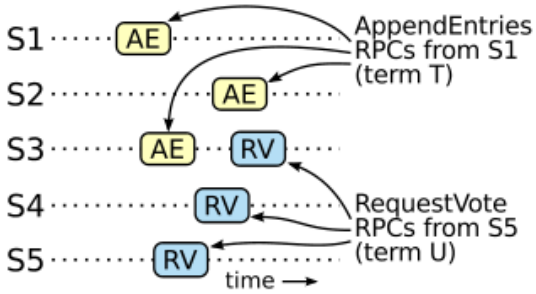

Figure 9: 如果 S1（任期 T 的领导者）从其任期提交了一个新的日志条目，并且 S5 被选为后来的任期 U 的领导者，那么必须至少有一个服务器（S3）接受了日志条目并且也投票给了 S5。

1. 提交的条目 a 在选举时一定不在 $leader_U$ 的日志中（领导者永远不会删除或覆盖条目）。
2. $leader_T$ 在大多数跟随者上复制了该条目 a，而 $leader_U$ 收到了来自大多数跟随者的投票。 因此，至少有一个服务器（“投票者”）既接受了来自 $leader_T$ 的条目 a，又投票给了 $leader_U$，如 Figure 9 所示。投票者是达成矛盾的关键。
3. 投票者必须在投票给 $leader_U$ 之前接受来自 $leader_T$ 的提交条目 a； 否则它会拒绝来自 $leader_T$ 的 AppendEntries RPC 请求（它的当前任期会高于 T）。
4. 投票者在投票给 $leader_U$ 时仍然存储该条目 a，因为领导者永远不会删除条目，而追随者只有在与领导者发生冲突时才会删除条目。
5. 投票者将其投票给了 $leader_U$，因此 $leader_U$ 的日志必须至少与投票者的日志一样最新。 这导致了下面两个矛盾之一。
6. 首先，如果投票者和 $leader_U$ 共享相同的最后一个日志任期，那么 $leader_U$ 的日志一定至少和投票者的一样新，所以它的日志包含了投票者日志中的每个条目。这是一个矛盾，因为投票者包含已提交的条目 a，而 $leader_U$ 被假定为不包含 a。
7. 否则，$leader_U$ 的最后一个日志条目一定比投票者的大。 此外，它比 T 大，因为投票者的最后一个日志任期至少是 T（它包含来自任期 T 的已提交条目）。 创建 $leader_U$ 的最后一个日志条目的较早领导者必须在其日志中包含已提交的条目（根据假设）。 那么根据日志匹配属性（Log Matching），$leader_U$的日志也必须包含已提交的条目 a，这是矛盾的。
8. 这就完成了矛盾。 因此，所有大于 T 的任期的领导者必须包含任期 T 中在任期 T 中提交的所有条目。
9. 日志匹配属性（Log Matching）保证未来的领导者也将包含间接提交的条目，例如 Figure 8 （d） 中的索引 2。

鉴于证领导者完整性（Leader Completeness Property）成立，我们可以证明 Figure 2 中的状态机安全属性，该属性表明如果服务器已将给定索引处的日志条目应用于其状态机，则相同的索引处不会有另一个不同的日志条目应用于状态机。当服务器将日志条目应用于其状态机时，其日志必须与通过该条目向上的领导者日志相同，并且必须提交该条目。 现在考虑任何服务器应用给定日志索引的最低期限；日志完整性属性保证所有更高任期的领导者将存储相同的日志条目，因此在以后的任期应用索引的服务器将应用相同的值。 因此，状态机安全属性成立。

最后，Raft 要求服务器按日志索引顺序应用条目。 结合状态机安全属性，这意味着所有服务器将以相同的顺序将完全相同的一组日志条目应用到它们的状态机。

5.5. 追随者和候选者崩溃

到目前为止，我们一直关注领导者的失败。跟随者和候选者的崩溃比领导者的崩溃更容易处理，并且它们的处理方式相同。 如果一个追随者或候选人崩溃，那么未来发送给它的 RequestVote 和 AppendEntries RPCs

将失败。 Raft 通过无限重试来处理这些失败； 如果崩溃的服务器重新启动，则 RPC 将成功完成。 如果服务器在完成 RPC 之后但在响应之前崩溃，那么它将在重新启动后再次收到相同的 RPC。 Raft RPCs 是幂等的，所以这不会造成伤害。 例如，如果跟随者收到一个 AppendEntries 请求，其中包含已存在于其日志中的日志条目，它会忽略新请求中的这些条目。

5.6. 时间和可用性

我们对 Raft 的要求之一是安全性不能依赖于时间：系统不能仅仅因为某些事件发生得比预期快或慢就产生错误的结果。 然而，可用性（系统及时响应客户的能力）不可避免地取决于时间。 没有稳定的领导者，Raft 就无法进步。

领导者选举是 Raft 中最关键的方面。 只要系统满足以下时序要求，Raft 就可以选举并维持稳定的领导者：

$$broadcastTime \ll electionTimeout \ll MTBF$$

在这个不等式中，broadcastTime 是服务器向集群中的每个服务器并行发送 RPC 并接收它们的响应所花费的平均时间； electionTimeout 是 5.2 节中描述的选举超时； MTBF 是单个服务器的平均故障间隔时间。 broadcastTime 应该比 electionTimeout 小一个数量级，以便领导者可以可靠地发送阻止追随者开始选举所需的心跳消息； 考虑到用于选举超时的随机方法，这种不平等也使得分裂投票不太可能发生。 electionTimeout 应该比 MTBF 小几个数量级，这样系统才能稳步前进。 当领导者崩溃时，系统将在大致选举超时时间内不可用； 我们希望这只占总时间的一小部分。

broadcastTime 和 MTBF 是底层系统的属性，而 electionTimeout 是我们必须选择的。 Raft 的 RPC 通常需要接收者将信息持久化到稳定的存储中，因此 electionTimeout 可能在 0.5ms 到 20ms 之间，具体取决于存储技术。 因此，选举超时可能介于 10 毫秒和 500 毫秒之间。 典型的服务器 MTBF 为几个月甚至更长，很容易满足时序要求。

6. 集群成员变化

到目前为止，我们假设集群配置（参与共识算法的服务器集）是固定的。 在实践中，有时需要更改配置，例如在服务器发生故障时更换服务器或更改复制程度。 尽管这可以通过使整个集群脱机、更新配置文件然后重新启动集群来完成，但这会使集群在转换期间不可用。 此外，如果有任何手动步骤，则存在操作员失误的风险。 为了避免这些问题，我们决定自动化配置更改并将其纳入 Raft 共识算法。

为了使配置更改机制安全，在过渡期间同一任期不能有两个领导者被选出。 不幸的是，服务器直接从旧配置切换到新配置的任何方法都是不安全的。 不可能一次自动切换所有服务器，因此集群可能会在过渡期间分裂成两个独立的部分（参见 Figure 10 ）。

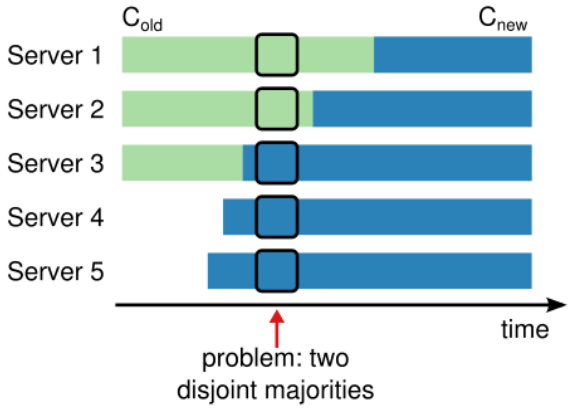

Figure 10: 直接从一种配置切换到另一种配置是不安全的，因为不同的服务器会在不同的时间切换。 在此示例中，集群从三台服务器增长到五台。 不幸的是，有一个时间点可以为同一任期选出两个不同的领导者，一个拥有大多数旧配置（C_{old}），另一个拥有大多数新配置（C_{new}）。

为了确保安全，配置更改必须使用两阶段的方法。 有多种方法可以实现这两个阶段。 例如，某些系统使用第一阶段禁用旧配置，因此它无法处理客户端请求； 然后

第二阶段启用新配置。 在 Raft 中，集群首先切换到我们称为联合共识的过渡配置；一旦达成共同共识，系统就会过渡到新的配置。 联合共识结合了新旧配置：

- 日志条目被复制到两种配置中的所有服务器。
- 来自任一配置的任何服务器都可以充当领导者。
- 达成共识（用于选举和提交日志条目）需要来自旧配置和新配置的不同多数。

联合共识允许各个服务器在不同时间在配置之间进行转换，而不会影响安全性。 此外，联合共识允许集群在整个配置更改期间继续为客户端请求提供服务。

集群配置使用复制日志中的特殊条目进行存储和通信；Figure 11 说明了配置更改过程。 当领导者收到将配置从 C_{old} 更改为 C_{new} 的请求时，它将联合共识的配置（图中的 Cold,new）存储为日志条目，并使用前面描述的机制复制该条目。 一旦给定的服务器将新的配置条目添加到其日志中，它将使用该配置来进行所有未来的决策（服务器始终使用其日志中的最新配置，无论该条目是否已提交）。 这意味着领导者将使用 $C_{old,new}$ 的规则来确定何时提交 $C_{old,new}$ 的日志条目。 如果领导者崩溃，则可以在 C_{old} 或 $C_{old,new}$ 下选择新的领导者，这取决于获胜的候选人是否收到了 $C_{old,new}$。 无论如何，C_{new}在此期间不能单方面做出决定。

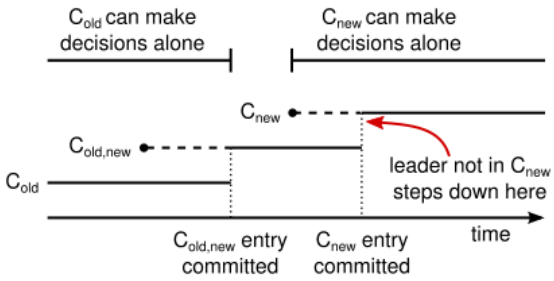

Figure 11: 配置更改的时间线。 虚线表示已创建但未提交的配置条目，实线表示最新提交的配置条目。 领导者首先在其日志中创建 $C_{old,new}$ 配置条目并将其提交给 $C_{old,new}$（大多数为 C_{old} 和大多数为 C_{new}）。 然后它创建 C_{new} 条目并将其提交给大多数 C_{new}。 没有时间点可以让 C_{old} 和 C_{new} 都独立做出决定。

联合共识允许各个服务器在不同时间在配置之间进行转换，而不会影响安全性。 此外，联合共识允许集群在整个配置更改期间继续为客户端请求提供服务。

集群配置使用复制日志中的特殊条目进行存储和通信； Figure 11 说明了配置更改过程。 当领导者收到将配置从 C_{old} 更改为 C_{new} 的请求时，它将联合共识的配置（图中的 $C_{old,new}$）存储为日志条目，并使用前面描述的机制复制该条目。 一旦给定的服务器将新的配置条目添加到其日志中，它将使用该配置来进行所有未来的决策（服务器始终使用其日志中的最新配置，无论该条目是否已提交）。 这意味着领导者将使用 $C_{old,new}$ 的规则来确定何时提交 $C_{old,new}$ 的日志条目。 如果领导者崩溃，则可以在 C_{old} 或 $C_{old,new}$ 下选择新的领导者，这取决于获胜的候选人是否收到了 $C_{old,new}$。 无论如何，C_{new}在此期间不能单方面做出决定。

一旦 $C_{old,new}$ 被提交，C_{old} 和 C_{new} 都不能在未经对方批准的情况下做出决定，并且 Leader Completeness Property 确保只有具有 $C_{old,new}$ 日志条目的服务器才能被选为领导者。 现在领导者可以安全地创建一个描述 C_{new} 的日志条目并将其复制到集群中。 同样，此配置将在每台服务器上看到后立即生效。 当新的配置按照 C_{new} 的规则提交后，旧的

配置就无关紧要了，不在新配置中的服务器可以被关闭。 如 Figure 11 所示，没有时间 C_{old} 和 C_{new} 都可以做出单边决策； 这保证了安全。

重新配置还有另外三个问题需要解决。 第一个问题是新服务器最初可能不存储任何日志条目。 如果以这种状态将它们添加到集群中，它们可能需要很长时间才能赶上进度，在此期间可能无法提交新的日志条目。 为了避免可用性差距，Raft 在配置更改之前引入了一个额外的阶段，在该阶段中，新服务器作为非投票成员加入集群（领导者将日志条目复制给他们，但他们不考虑多数）。 一旦新服务器赶上了集群的其余部分，重新配置就可以如上所述进行。

第二个问题是集群领导者可能不是新配置的一部分。 在这种情况下，领导者一旦提交了 C_{new} 日志条目，就会下台（返回到跟随者状态）。 这意味着当领导者正在管理一个不包括自己的集群时，会有一段时间（当它正在提交 C_{new} 时）； 它复制日志条目但不计入多数。 领导者转换发生在 C_{new} 被提交时，因为这是新配置可以独立运行的第一点（总是可以从 C_{new} 中选择一个领导者）。 在此之前，可能只有来自 C_{old} 的服务器才能被选为领导者。

第三个问题是移除的服务器（那些不在 C_{new} 中的）会破坏集群。 这些服务器不会收到心跳，因此它们会超时并开始新的选举。 然后他们将发送带有新任期号的 RequestVote RPC，这将导致当前领导者恢复为追随者状态。 最终会选举出新的领导者，但是被移除的服务器会再次超时，重复这个过程，导致可用性很差。

为防止出现此问题，服务器在认为当前领导者存在时会忽略 RequestVote RPC。 具体来说，如果服务器在从当前领导者那里听到的最短选举超时内收到 RequestVote RPC，则它不会更新其任期或授予其投票权。 这不会影响正常的选举，其中每个服务器在开始选举之前至少等待最小选举超时。 然而，它有助于避免被移除的服务器造成的中断：如果一个领导者能够获得其集群的心跳，那么它就不会被更大的任期数所取代。

7. 日志压缩

Raft 的日志在正常运行期间会增长以包含更多的客户端请求，但在实际系统中，它不能无限制地增长。 随着日志变长，它占用更多空间并需要更多时间来重播。 如果没有某种机制来丢弃日志中积累的过时信息，这最终会导致可用性问题。

快照是最简单的压缩方法。 在快照中，整个当前系统状态被写入稳定存储上的快照，然后丢弃到该点的整个日志。 Chubby 和 ZooKeeper 中使用了快照，本节的其余部分描述了 Raft 中的快照。

增量压缩方法，例如日志清理和日志结构合并树（log-structured merge tree，LSM tree），也是可能的。 它们一次对一小部分数据进行操作，因此它们会随着时间的推移更均匀地分配压缩负载。 他们首先选择一个已经积累了许多已删除和覆盖对象的数据区域，然后他们更紧凑地重写该区域中的活动对象并释放该区域。 与快照相比，这需要显着的额外机制和复杂性，快照通过始终对整个数据集进行操作来简化问题。 虽然日志清理需要修改 Raft，但状态机可以使用与快照相同的接口来实现 LSM 树。

图 12 显示了 Raft 中快照的基本思想。 每个服务器独立拍摄快照，仅覆盖其日志中已提交的条目。 大部分工作包括状态机将其当前状态写入快照。 Raft 还在快照中包含少量元数据：最后包含的索引是快照替换的日志中最后一个条目的索引（状态机应用的最后一个条目），最后包含的任期是这个条目的任期。 保留这些以支持对快照后的第一个日志条目的 AppendEntries 一致性检查，因为该条目需要先前的日志索引和任期。 为了启用集群成员更改（第 6 节），快照还包括日志中截至最后包含索引的最新配置。 一旦服务器完成写入快照，

它可能会删除所有日志条目，直到最后一个包含的索引，以及任何先前的快照。

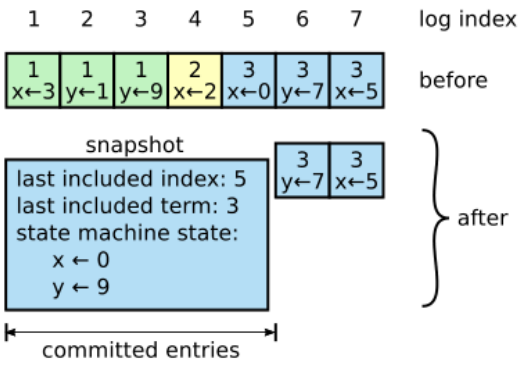

Figure 12: 服务器用新快照替换其日志中已提交的条目（索引 1 到 5），该快照仅存储当前状态（本例中为变量 x 和 y）。 快照最后包含的索引和任期用于将快照定位在条目 6 之前的日志中。

虽然服务器通常独立拍摄快照，但领导者必须偶尔将快照发送给落后的跟随者。 当领导者已经丢弃了它需要发送给跟随者的下一个日志条目时，就会发生这种情况。 幸运的是，这种情况在正常操作中不太可能发生：跟上领导者的跟随者已经有了这个条目。 然而，一个特别慢的跟随者或一个加入集群的新服务器（第 6 节）不会。 使这样的跟随者保持最新状态的方法是领导者通过网络向其发送快照。

领导者使用一个名为 InstallSnapshot 的新 RPC 将快照发送给落后太多的追随者； 请参见 Figure 13。当跟随者收到带有此 RPC 的快照时，它必须决定如何处理其现有日志条目。 通常快照会包含跟随者日志中没有的新信息。 在这种情况下，跟随者丢弃它的整个日志； 它全部被快照取代，并且可能有与快照冲突的未提交条目。 相反，如果跟随者收到描述其日志前缀的快照（由于重传或错误），则快照覆盖的日志条目将被删除，但快照之后的条目仍然有效且必须保留。

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.

Arguments:

term	leader's term
leaderId	so follower can redirect clients
lastIncludedIndex	the snapshot replaces all entries up through and including this index
lastIncludedTerm	term of lastIncludedIndex
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot chunk, starting at offset
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. Reply immediately if term < currentTerm
2. Create new snapshot file if first chunk (offset is 0)
3. Write data into snapshot file at given offset
4. Reply and wait for more data chunks if done is false
5. Save snapshot file, discard any existing or partial snapshot with a smaller index
6. If existing log entry has same index and term as snapshot's last included entry, retain log entries following it and reply
7. Discard the entire log
8. Reset state machine using snapshot contents (and load snapshot's cluster configuration)

Figure 13: InstallSnapshot RPC 的摘要。 快照被分成块进行传输； 这为跟随者提供了每个块的生命迹象，因此它可以重置其选举计时器。

这种快照方法背离了 Raft 的强领导者原则，因为跟随者可以在领导者不知情的情况下拍摄快照。 然而，我们认为这种偏离是合理的。 虽然拥有领导者有助于避免在达成共识时做出冲突的决策，但在快照时已经达成共识，因此没有决策冲突。 数据仍然只从领导者流向追随者，只是追随者现在可以重组他们的数据。

我们考虑了一种基于领导者的替代方法，其中只有领导者会创建快照，然后它将此快照发送给它的每个追随者。 然而，这有两个缺点。 首先，将快照发送给每个跟随者会浪费网络带宽并减慢快照过程。 每个跟随者都已经拥有生成自己的快照所需的信息，服务器从其本地状态生成快照通常比通过网络发送和接收快照便宜得多。 其次，领导者的实施会更复杂。 例如，领导者需要在向追随者复制新日志条目的同时向追随者发送快照，以免阻塞新的客户端请求。

还有两个影响快照性能的问题。 首先，服务器必须决定何时进行快照。 如果服务器快照过于频繁，则会浪费磁盘带宽和能量；如果快照频率太低，则可能会耗尽其存储容量，并且会增加重新启动期间重播日志所需的时间。 一种简单的策略是在日志达到固定大小（以字节为单位）时拍摄快照。如果此大小设置为明显大于快照的预期大小，则快照的磁盘带宽开销将很小。

第二个性能问题是写入快照会花费大量时间，我们不希望这会延迟正常操作。 解决方案是使用写时复制技术，这样可以在不影响正在写入的快照的情况下接受新的更新。 例如，使用功能数据结构构建的状态机自然支持这一点。 或者，操作系统的写时复制支持（例如，Linux 上的 fork）可用于创建整个状态机的内存快照（我们的实现使用这种方法）。

8. 客户端交互

本节描述客户端如何与 Raft 交互，包括客户端如何找到集群领导者以及 Raft 如何支持线性化语义。 这些问题适用于所有基于共识的系统，而 Raft 的解决方案与其他系统类似。

Raft 的客户将他们所有的请求发送给领导者。 当客户端首次启动时，它会连接到随机选择的服务器。 如果客户端的第一选择不是领导者，该服务器将拒绝客户端的请求并提供有关它收到的最近领导者的信息（AppendEntries 请求包括领导者的网络地址）。 如果领导者崩溃，客户端请求将超时；客户端然后使用随机选择的服务器重试。

我们对 Raft 的目标是实现可线性化的语义（每个操作似乎在其调用和响应之间的某个时间点立即执行一次）。 然而，正如目前为止所描述的，Raft 可以多次执行一个命令：例如，如果领导者在提交日志条目之后但在响应客户端之前崩溃了，客户端将使用新的领导者重试该命令，导致它被执行第二次。 解决方案是让客户为每个命令分配唯一的序列号。 然后，状态机跟踪为每个客户端处理的最新序列号，以及相关的响应。 如果它接收到一个序列号已经被执行过的命令，它会立即响应而不会重新执行该请求。

可以在不向日志中写入任何内容的情况下处理只读操作。 然而，如果没有额外的措施，这将有返回陈旧数据的风险，因为响应请求的领导者可能已经被一个它不知道的新领导者取代。 可线性化读取不得返回陈旧数据，并且 Raft 需要两个额外的预防措施来保证这一点而不使用日志。

首先，领导者必须拥有关于哪些条目已提交的最新信息。 Leader Completeness Property 保证领导者拥有所有已提交的条目，但在其任期开始时，它可能不知道那些是哪些。要找出答案，它需要从它的任期提交一个条目。 Raft 通过让每个领导者在其任期开始时向日志中提交一个空白的无操作条目来处理这个问题。

其次，领导者必须在处理只读请求之前检查自己是否已被废除（如果选举了更新的领导者，其信息可能会过时）。 Raft 通过让领导者在响应只读请求之前与大多数集群交换心跳消息来处理这个问题。 或者，领导者可以依靠心跳机制来提供一种租约形式，但这将依赖于安全时序（它假设有界时钟偏差）。

9. 实现与评估

我们已经将 Raft 实现为复制状态机的一部分，该状态机存储 RAMCloud 的配置信息并协助 RAMCloud 协调器的故障转移。Raft 实现包含大约 2000 行 C++ 代码，不包括测试、注释或空行。 源代码可免费获得。 基于本文的草稿，还有大约 25 个处于不同开发阶段的 Raft 的独立第三方开源实现。 此外，各种公司正在部署基于 Raft 的系统。

略

10. 相关工作

有许多与共识算法相关的出版物，其中许多属于以下类别之一：

- Lamport 对 Paxos 的原始描述，并试图更清楚地解释它。
- Paxos 的精化，填补了缺失的细节并修改了算法，为实现提供了更好的基础。
- 实现共识算法的系统，例如 Chubby、ZooKeeper 和 Spanner。Chubby 和 Spanner 的算法尚未详细公布，尽管两者都声称基于 Paxos。 ZooKeeper 的算法已经发布了更详细的内容，但它与 Paxos 有很大不同。
- 可应用于 Paxos 的性能优化。
- Oki 和 Liskov 的 Viewstamped Replication (VR)，一种与 Paxos 大约同时开发的共识替代方法。 最初的描述与分布式事务协议交织在一起，但核心共识协议在最近的更新中已被分离。 VR 使用与 Raft 有许多相似之处的基于领导者的方法。

Raft 和 Paxos 最大的区别在于 Raft 的强大领导力：Raft 将领导选举作为共识协议的重要组成部分，并将尽可能多的功能集中在领导者身上。 这种方法导致更简单的算法，更容易理解。 例如，在 Paxos 中，领导者选举与基本共识协议正交：它仅作为性能优化，而不是达成共识所必需的。 然而，这导致了额外的机制：Paxos 包括用于基本共识的两阶段协议和用于领导人选举的单独机制。 相比之下，Raft 将领导者选举直接纳入共识算法，并将其作为共识两个阶段中的第一个阶段。 这导致比 Paxos 更少的机制。

略

11. 结论

算法的设计通常以正确性、效率和/或简洁性为主要目标。 尽管这些都是有价值的目标，但我们认为可理解性同样重要。 在开发人员将算法转化为实际实现之前，其他目标都无法实现，这将不可避免地偏离并扩展已发布的形式。 除非开发人员对算法有深刻的理解并能对其产生直觉，否则他们将很难在实现中保留其理想的特性。

略

12. 致谢

略

13. 参考文献

略

14. 习题

这些习题可以检测你对 Raft 的理解程度，认真思考每道题目，并在必要时重读论文。本节的内容并不属于原论文，Redisant 提供这些习题的目的是为了巩固读者对 Raft 的理解。Redisant 为 etcd 提供了一款可视化管理工具，你可以在需要时使用它。

1. 日志条目在什么条件下被认为是已提交的？

   可以被安全地应用到状态机的条目称为已提交的。 Raft 保证已提交的条目是持久的，并且最终会被所有可用的状态机执行。 一旦创建条目的领导者已将其复制到大多数服务器，那么该日志就被称为已提交的（此时将该日志条目应用到状态机是安全的）。

2. 已提交的日志最终都会被应用到状态机吗？

   当条目已被提交后，领导者将条目应用于其状态机并将该执行的结果返回给客户端。 如果跟随者崩溃或运行缓慢，或者网络数据包丢失，领导者会无限期地重试 AppendEntries RPC（即使在它已经响应客户端之后）直到所有跟随者最终存储所有日志条目。并且，每台服务器在启动时会执行以下检测，如果已提交的日志条目的索引位置（commitIndex）大于已应用于状态机的条目的索引位置（lastApplied），则该位置的日志条目将应用到状态机。

3. 安全选举（Election Safety）属性怎么得到保证？

   如果一个候选者在同一任期内获得了整个集群中大多数服务器的投票，那么它就赢得了选举。每台服务器在给定的任期内最多为一名候选者投票，以先到先得为原则（注：第5.4节对投票增加了一个额外的限制）。少数服从多数的原则保证了最多只有一名候选者能够在某一任期内赢得选举。

4. 日志匹配（Log Matching）属性怎么得到保证？

   Raft 保证了以下条件成立，从而保证了日志匹配：
   1. 如果不同日志中的两个条目具有相同的索引和任期，则它们存储相同的命令。
   2. 如果不同日志中的两个条目具有相同的索引和任期，则在该条目之前的所有条目都是相同的。

   第一个条件成立源于这样一个事实，即领导者在给定任期内最多创建一个具有给定日志索引的条目，并且日志条目永远不会改变它们在日志中的位置。 第二个条件由 AppendEntries RPC 执行的简单一致性检查保证。 当发送 AppendEntries RPC 时，领导者在其日志中包含紧接在新条目之前的条目的索引和任期。 如果跟随者在其日志中没有找到具有相同索引和任期的条目，那么它会拒绝新条目。 一致性检查作为一个归纳步骤：日志的初始空状态满足日志匹配属性，并且只要追加日志，一致性检查就会保留日志匹配属性。 因此，每当 AppendEntries RPC 成功返回时，领导者就知道了跟随者的日志与自己的日志相同。

5. 请详细描述 AppendEntries RPC 执行一致性检查的过程。

   为了使跟随者的日志与其自己（领导者）的一致，领导者必须找到在跟随者的日志中和自己一致的最新日志条目，然后在跟随者日志中删除该点之后的所有条目，并将该点之后的所有领导者日志条目发送给跟随者。 所有这些操作都在通过 AppendEntries RPC 执行的一致性检查时发生。 领导者为每个跟随者维护一个 nextIndex，这是领导

者将发送给该跟随者的下一个日志条目的索引。当一个领导者第一次掌权时，它会将所有nextIndex 值初始化为其日志中最后一个索引之后的索引（Figure 7 中的 11）。如果跟随者的日志与领导者的日志不一致，则在下一个 AppendEntries RPC 中，一致性检查将失败。 拒绝后，领导者递减 nextIndex 并重试 AppendEntries RPC。 最终 nextIndex 将达到领导者和跟随者日志匹配的点。 当发生这种情况时，AppendEntries RPC 将成功，它会删除跟随者日志中的所有冲突条目并追加领导者日志中的条目（如果有的话）。 一旦 AppendEntries RPC 成功，跟随者的日志与领导者的一致，并且在剩余的任期内保持这种状态。