**In Search of an Understandable Consensus Algorithm**

**摘要**

Raft是管理副本日志的共识算法。它提供了和Paxos相同的功能，并且与Paxos一样高效，但是它的结构与Paxos有一些不同；这使得Raft更容易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将共识的关键分为几块，分别是leader选举，log复制和安全性，并且实现了更强壮的一致性程度以减少需要考虑的情况。研究结果证明对于学生来说Raft比Paxos更容易学习。Raft同时包括一个新的机制去修改成员集群，使用重叠多数（Overlapping Majorities）以保证安全性。

**1 Introduction**

共识算法允许机器集群像一个团结的能够容忍部分成员失败的集体一样工作。正因为如此，他们在构建可靠的大规模软件系统中扮演了关键角色。Paxos[15,16]在过去的十年间主导了共识算法的讨论：很多的共识算法实现都基于Paxos或者受其影响，而且Paxos变成了教导学生共识算法的主要方式。

不幸的是，尽管有很多使Paxos更容易被理解的尝试 ，它还是有一点难以理解。此外，它的架构需要复杂的修改以支持实际的系统。因为如此，系统构建者和学生都不喜欢Paxos。

在与Paxos抗争之后，我们着手去探寻新的可以提供更好的系统构建和教育基础的共识算法。我们的方法不是一般的方法，因为我们的主要目标是可理解性：我们可以对实际系统定义一个共识算法并且用明显比Paxos更简单学习去描述它吗？此外，我们希望算法可以促进直观的对于系统构建者的必要的发展。对于算法如何工作，为什么这么做都是十分重要的。

该项工作的结果是一个叫做Raft的共识算法。在设计Raft过程中我们应用了一些特殊的技术以改变可理解性，包括分解（Raft分隔了leader选举，log复制和安全性）和状态空间减少（相对Paxos，Raft减少了非确定性等级和servers可以和其他不一致的方法）一个针对两所大学43个学生的研究显示Raft相对Paxos明显更好理解：同时学习了两种算法后，其中的33个学生对Raft的问题回答的比Paxos更好。

Raft在很多地方和一些共识算法相似（特别是，Oki和Liskov的Viewstamped Replication[29,22]），但是有一些新特性：

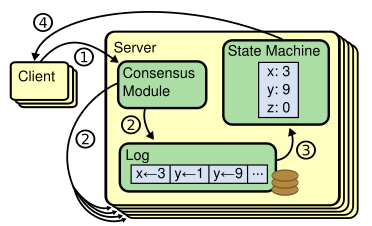
* Strong leader：Raft相对其他共识算法使用了更强的leadership构成。举例来说，log实体仅仅从leader到其他servers。这样简化了复制log的管理并且使得Raft更容易理解。
* Leader election：Raft使用随机定时器选举leader。该方式仅增加了一小部分其他算法早就有的心跳机制，可以使解决冲突更简单和更快。
* Membership changes：Raft对于更改server集群中集合的机制使用新共同共识（joint consensus）方式，这会使得在过渡时有两个不同配置的集合重叠。这样允许集群在配置修改时继续正常运作。

我们相信Raft比Paxos和其他共识算法优秀，不管是教育目的还是作为具体实现的基础。它比其他算法更简单，更易于理解；它的描述足够完整以满足实际系统的需求；它有很多的开源实现并且被很多公司使用；它的安全性属性被正式的指定和证明；它的效率也可以与其他算法相比。

论文的剩余部分介绍了复制状态机问题（第2节），讨论了Paxos的优劣势（第3节），描述了我们如何提高可理解性（第4节），现在的Raft共识算法（第5-8节），评价了Raft（第9节），并且讨论了相关工作（第10节）。

**2 Replicated state machines**

共识算法通常出现在复制状态机场景中。在这个方式中，server集合中的状态机计算相同的状态拷贝并且能在部分server宕机情况下继续执行。复制状态机是用来解决分布式系统中各式各样的容错问题的。举例来说，有一个单独的集群leader的大规模系统，就像GFS[8]，HDFS[38]，和RAMCloud[33]，一般使用一个隔离的复制状态机去管理leader选举和存储Leader崩溃后依然存在的配置信息。复制状态机的例子包括Chubby[2]和ZooKeeper[11]。

图1. 复制状态机架构。该共识算法管理一个包括了从client发出的状态机指令的日志副本。该状态机处理相同的从日志中获取的指令队列，所以他们产生相同的输出。

复制状态机一般使用复制日志实现，就如图1所示。每一个server存储一个包含了一系列相同顺序指令的日志，所以每一个状态机处理相同的指令序列。因此状态机是确定性的，每一次计算相同状态和相同输出序列。

保持日志副本一致是共识算法的职责。Server的共识模块收到client发送的指令并将它们写入日志中。它与其他server的共识模块通信以确认所有日志最终包含相同顺序和内容的请求，即使有些server失败了。一旦指令被正确复制了，每一个server的状态机根据日志的顺序处理它们，并且输出被回复给client。结果是，server似乎来自单独的高可靠的状态机。

实际系统的共识算法一般有以下属性：

* 在所有的无拜占庭情况下它们确保安全性（从不返回错误的结果），包括网络延迟，分隔，和包丢失，重复和重排序。
* 它们是完全实用（可用）的只要任意server的多数是工作的并且可以和其他server或client通信。
* 它们不依赖于定时去确认日志的一致性：不完善的时钟和极端的消息延迟，最坏造成可用性问题。
* 在常见场景下，一个指令可以在集群的多数响应一轮RPC调用后立即完成；小部分慢的server不会影响整体系统性能。

**3 What's wrong with Paxos?**

过去的十年间，Leslie Lamport的Paxos协议[15]成为了共识的代名词：它是课程上最广泛教导的，同时大部分共识算法实现使用它作为初始点。Paxos首次定义了协议的单个决定的达成协议的能力，比如一个单独的日志副本实体。我们将该子集称为single-decree Paxos。Paxos然后组合了协议多个实例以促进一系列决定比如一个日志（multi-Paxos）。Paxos同时确保了安全性和活跃性，而且它支持集群成员的改变。它的正确性也被证明了，在正常情况下效率也很不错。

不幸的是，Paxos有两个显著缺点。第一个缺点是Paxos异常难以理解。它的完整解释[15]是公认的不透明；少部分人花费大量功夫成功理解了它。结果是，这里有一些尝试用简单术语来解释Paxos的文章[16,20,21]。这些解释着眼于single-decree子集，这样也是极有挑战的。在NSDI 2012的参加者的非正式调查中，我们发现即使其中有老练的研究员，只有少部分人适应Paxos。我们与Paxos抗争；我们在直到读了一些简化的解释并设计我们自己的替代协议前都不能去理解完整的协议，这个进程花了大约一年。

我们推测Paxos的不透明来源于它选择single-decree子集作为基础。Single-decree Paxos是繁杂与细微的：它分为两个不能简单直观解释和独立理解的阶段。正因为如此，发展关于协议工作原理的思考是很难的。Multi-Paxos的组合规则添加了大量额外的复杂性和细节。我们相信多个决定（比如，一个日志代替一个单独实体）达到共识的问题可以用其他更直接明显方法分解。

第二个问题是没有给构建实用的实现提供一个良好的基础。一个理由是multi-Paxos没有被广泛接受。Lamport的描述更多是关于single-decree Paxos的；他只简单描述了可能实现multi-Paxos的方法，但是缺少了大量细节。有大量的尝试去充实优化Paxos，比如[26]，[39]和[13]，但是这些各自不同也和Lamport的描述不同。像Chubby这样的系统实现了类似Paxos的算法，但是很多场景下他们的处理细节都未公开。

此外，Paxos的架构对于构建实际系统而言太差了；这是拆分single-decree导致的另一个后果。举例来说，选择一个独立的日志实体集合然后将它们合并成一个有序的日志，没有任何益处；这仅仅增加了复杂度。设计一个围绕日志的系统更简单也更有效，新实体以受限的顺序有序的附加。另一个问题是Paxos使用了对称点对点方法作为核心（虽然它最后建议弱结构的领导来作为性能优化）。这在一个简化的只做一个决定的世界是有意义的，但是很少有实际系统会用这个方式。如果一系列决定必须确认，在首次选举leader时会更简单更快，然后有了leader协调决定。

结果是，实际系统使用类似Paxos的算法。每一个实现都始于Paxos，探索实现过程中的难点，然后开发一个明显不同的架构。这是非常耗时且容易出错的，而且理解Paxos的难度也会增加。Paxos的公式对于提供它的正确性理论是很好的，但是实际实现是不同于Paxos以至于证明的价值很少。下面Chubby实现者的评论是典型：

Paxos算法和实际系统需求有很明显的间隙，最终的系统将会基于未被证明的协议[4]。

因为这些原因，我们总结得出Paxos对于构建实际系统和教育都不适合作为基础。基于大规模软件系统中一致性的重要性，我们决定看看我们是否能够设计一个比Paxos有更好属性的替代共识算法。Raft正是实验的结果。

**4 Designing for understandability**

我们在设计Raft时有一些目标：它必须给系统构建提供一个完整的实用的基础，所以它明显减少了开发者的设计工作量；它必须在各种条件下是安全的，在一般操作环境下是可用的；对于常见操作必须是高效的。但是我们最重要的目标——也是最难的挑战——可理解性。它必须能够让大量的听众对算法感到舒适。此外，它必须可以发展关于算法的思考，所以系统构建者可以使用在真实实现中必须的扩展。

在Raft的设计中有很多点是需要我们在替代方案中选择的。这样的环境下我们基于可理解性评估了替代方案：解释每一个替代方案有多难（举例来说，它的状态空间多复杂，是否有细微的含义？），对于读者来说要完全理解它的方法和含义会多简单？

我们承认在这样的分析中有很高的主观性；尽管如此，我们使用两种普遍适用的技术。第一种技术是知名的问题分解方法：我们尽可能的把问题分解成可以解决的，解释的，相对独立理解的。举例来说，在Raft中我们分离了leader选举，log复制，安全性和成员变更。

我们的第二个方法是通过减少需要考虑的状态，使得系统更连贯和尽可能消除非确定性来简化状态空间。特别是日志不允许有缺口，并且Raft限制了日志会变得和其他不一致的方式。虽然在大多数情况下我们尝试消除非确定性，然而有些情况下非确定性可以提升可理解性。尤其是，随机方法引入非确定性，但是他们倾向于通过在相似风格（“任意选择；没关系的”）时处理所有的可能选项来减少状态空间。我们使用随机去简化Raft的leader选举算法。

**5 The Raft consensus algorithm**

Raft是一个像第2节描述的结构那样管理日志副本的算法。图2以浓缩的结构作为参考总结了算法，图3列出了算法的关键属性；这些图的元素在该节的剩余部分分段讨论。

|  |
| --- |
| State |
| 所有server上的持久化状态（在回复RPCs前更新稳定存储）：  currentTerm：server看到的最新term（首次引导被初始化为0，单调递增）  voteFor：当前term收到的投票的候选人ID（如果没有就为null）  log[]：日志实体；每个实体包含状态机指令，当实体被leader接收时的期（初始索引1）  所有server上的挥发性状态：  commitIndex：已知的提交的日志实体最高索引值（初始为0，单调递增）  lastApplied：状态机应用的日志实体最高索引值（初始为0，单调递增）  Leaders的挥发性状态（选举后重新初始化）：  nextIndex[]：对于每一个server的下一个会发送的日志实体的索引（初始为leader最后的日志索引+1）  matchIndex[]：对于每一个server的复制的已知日志实体的最高索引值（初始为0，单调递增） |
| RequestVote RPC |
| 由候选者调用以收集选票(5.2节)  参数：  term：候选者的term  candidateId：候选者要求投票  lastLogIndex：候选者的最新日志实体索引（5.4节）  lastLogTerm：候选者的最新日志实体term（5.4节）  结果：  term：当前term，给候选者更新自身的term  voteGranted：如果候选者收到则返回true  接收者实现：  1.如果term<currentTerm则回复false（5.1节）  2.如果votedFor是空的或者candidatedId，且候选者的日志最少是和接收者的日志是最新的，则投票（5.2，5.4节） |
| AppendEntries RPC |
| 由leader调用去复制日志实体（5.3节）；同时用作心跳（5.2节）  参数：  term：leader的term  leaderId：跟随者因此可以重定向clients  prevLogIndex：上一个日志实体的索引值  prevLogTerm：prevLogIndex对应实体的term  entries[]：要存储的日志实体（心跳时是空的；为了效率可能发送多个）  leaderCommit：leader的commitIndex  结果：  term：currentTerm，给leader更新自身  success：如果跟随者有匹配prevLogIndex和prevLogTerm的实体则返回true  接收者实现：  1.如果term<currentTerm返回false（5.1节）  2.如果日志在prevLogIndex上没有实体的term为prevLogTerm，返回false（5.3节）  3.如果已有的实体和新的冲突（相同索引但是不同term），删除已有的实体和它之后的实体  4.添加任意不在日志中的实体  5.如果leaderCommit>commitIndex，设置commitIndex=min(leaderCommit, 最新实体的索引) |
| Rules for Servers |
| 所有servers：   * 如果commitIndex>lastApplied；增加lastApplied，应用log[lastApplied]给状态机（5.3节） * 如果RPC请求或回复包含term T>currentTerm；设置currentTerm为T，转换为跟随者（5.1节）   跟随者（5.2节）：   * 回复候选者和leaders的RPC调用 * 如果没有接收到来自当前leader的AppendEntries RPC或给候选者的投票，则选举超时：转换为候选者   候选者（5.2节）：   * 转换为候选者，开始选举：   + 增加currentTerm   + 投给自己   + 重置选举计时器   + 发送RequestVote RPC给所有其他servers * 如果从大多数server中收到投票：变成leader * 如果从一个新leader处收到AppendEntries RPC：转换为跟随者 * 如果选举超时：开始新选举   Leaders：   * 当选：发送初始空AppendEntries RPC（心跳）给所有server；在空闲期间重复发送以避免选举超时（5.2节） * 如果从client处收到指令：添加实体到本地日志，在实体应用到状态及后回应（5.3节） * 如果对于一个跟随者最新的日志index≥nextIndex：发送从nextIndex开始的日志实体的AppendEntries RPC   + 如果成功：更新跟随者的nextIndex和matchIndex   + 如果由于日志不一致失败：降低nextIndex并重试 * 如果存在一个N使得N>commitIndex，多数matchIndex[i]≥N，且log[N].term==currentTerm：设置commitIndex=N（5.3，5.4节） |

图2：一个Raft共识算法的简短总结（排除成员修改和日志压缩）。Server的行为被描述为独立重复触发的规则集合。小节编号表示何处会说明具体特性。正式的规范[31]描述算法更确切。

|  |
| --- |
| Election Safety：最多只有一个Leader会被选举出来（5.2节）  Leader Append-Only：Leader绝不在它的日志中复写或删除实体；只会添加新实体（5.3节）  Log Matching：如果两个日志包含一个相同索引和term的实体，则这两份日志中所有给定索引的实体都是相同的。（5.3节）  Leader Completeness：如果一个日志实体在给定的term中被提交，则对于所有更高序号的term，该实体会存在于leader的日志中。（5.4节）  State Machine Safety：如果一个server给它自己的状态机应用了一个给定索引的日志实体，没有其他server会在相同索引应用不同的日志实体。（5.4.3节） |

图3：Raft一直保证这些属性。小节编号表示何处会说明具体特性。

Raft实现共识的方式是首先选举一个突出的leader，然后给这个leader完整的管理日志副本的职责。Leader从clients处收到日志实体，将它们复制到其他server上，然后告诉server什么时候将日志实体应用到他们的状态机是安全的。Leader简化了日志副本的管理。举例来说，leader可以在没有询问其他server的情况下决定将新实体放在log的什么位置，而且数据流根据简单的风格从leader到其他server。Leader可以失败或者与其他server断连，这时会选举一个新leader。

鉴于leader的方法，Raft将共识问题分解为3个相对独立的子问题，将会在接下去的字节中讨论：

* Leader Selection：当已有的leader失败时必须选一个新leader（5.2节）。
* Log replication：Leader必须接受client发送的日志实体并在急集群中复制它们，要求其他日志以它的为准（5.3节）。
* Safety：Raft关键的安全性属性是图3中的状态机安全属性：如果所有server给它自己的状态机应用特定的日志实体，则没有其他server会在相同日志索引上应用不同的指令。5.4节描述了Raft怎么确保该属性的；解决方案设计一个5.2节描述的选举机制的额外限制。

提出该共识算法后，该节讨论可用性问题和系统中时机的作用。

**5.1 Raft Basics**

一个Raft集群包含了一些servers；一般是5个，允许系统中有2个失败的。每一个server在任意时刻都在3个状态之一：leader, follower或者candidate。在一般情况下只有1个leader，其他的server都是follower。Follower是被动的：他们不发出请求，仅仅回复leader和candidate的请求。Leader处理所有client的请求（如果client联系follower，follower会将请求转发给leader）。第三个状态，candidate，在5.2节描述中是用来选举一个新leader的。图4展示了状态和转换关系；转换将在下面讨论。

Raft将时间根据任意长度分为term，就像图5所示的那样。Term以连续的整数编号。每一个term由一个选举开始，即5.2节中描述的一个或多个candidate尝试变成leader时。如果一个candidate赢得了选举，则它会在term接下去的时间作为leader。在一些场景中，一个选举会导致分拆投票。在该情况下，该term会以没有leader而结束，一个新term（和新选举）会在短时间内开始。Raft保证在指定term内至多一个leader。

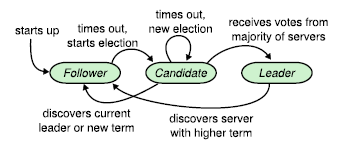


图4. Server状态。Followers只回复其他server的请求。如果一个follower不接收交流，它就变成candidate并且开始选举。从集群的多数处收到投票的candidate变成新的leader。Leader任期一般持续到它失败。

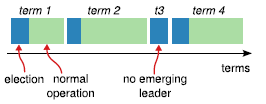


图5. 时间被分为不同term，每一个term由选举开始。在成功选举后，独立leader直到term结束一直管理集群。有些选举失败了，这时term由于没有选择一个leader而结束。可以在不同server的不同时间看到term的转换。

不同的Server可能在不同时间点看到term转换，在一些场景下一个server可能无法察觉到选举甚至整个term。在Raft中term表现为一个逻辑时钟[14]，它们允许server检测例如过期leader之类的过时信息。每一个server都保存了current term编号，每次都会单调递增。当前term在每次server沟通的时候都会交换；如果一个server的当前term比其他的小，则它会将它的当前值更新会更大的值。如果candidate或leader发现他的term过期了，它会立即还原为follower状态。如果server收到一个过期term的请求，它会拒绝该请求。

Raft的server使用RPC调用沟通，基本的共识算法只要求两类RPC。RequestVote RPC由candidate在选举时（5.2节）初始化，AppendEntries RPC由leader初始化以复制日志实体和提供心跳功能（5.3节）。第7节添加了第三个RPC用来在server间传递快照。Server如果没有在有限时间内收到回复，会重发RPC，并且会通过并发方式发送RPC来提高性能。

**5.2 Leader election**

Raft使用心跳机制来触发leader选举。当server启动时，他们首先是follower。一个server只要收到来自leader或candidate的有效RPC就会保留在follower状态。Leader发送周期性的心跳（不带任何日志实体的AppendEntries RPC）给所有的follower以保证保持它的地位。如果一个follower在有限期内未收到任何通信叫做election timeout（选举超时），接着它会假设没有leader然后开始新选举去选择新leader。

为了开始选举，follower增加他的current term并且转换为candidate状态。然后它投票给它自己并且并发发出RequestVote RPC给集群中所有的server。一个candidate直到接下去三种事情之一发生之后才会转换状态：（a）它赢得了选举（b）另一个server成为leader或（c）没有人在有限期内赢得选举。这些结果会在下面分开讨论。

如果candidate从整个集群的多数server处收到了同一个term的投票则它赢得选举。在指定term内，每个server至多会投给一个candidate，基于先到先得（first-come-first-served）原则（注：5.4节添加了一个额外限制）。这个主要的规则确保了在特定term至多只有一个candidate可以赢选举（图3的Election Safety属性）。一旦一个candidate赢了选举，它变成leader。接着它会发送心跳信息给其他所有server去建立它的地位和防止新选举。

当在等待投票时，candidate可能从其他要求成为leader的server处收到AppendEntries RPC。如果leader的term（在它的RPC内）起码是和candidate的current term一样，则该candidate承认leader是合法的并且返回follower状态。如果RPC中的term小于candidate的current term，则它拒绝RPC并且持续为candidate状态。

第三个可能的结果是candidate选举没有赢也没有输：如果很多follower同时变成candidate，投票可能会被分裂而导致没有candidate获得多数投票。当这种情况发生时，所有candidate都会超时并且通过添加term值并且初始化另一轮RequestVote RPC来开始新选举。然而，如果没有额外的措施分割投票可能导致无限重复。

Raft使用随机选举超时来保证分割投票是少见的并且可以被快速解决。为了首先避免分割投票，选举超时从一个固定间隔（比如150-300ms）内随机选取。这扩展了server，所以大多数情况下只有一个server会超时；它赢得了选举并且在其他server超时前发送心跳包。所有的candidate在选举开始前重启它的随机选举超时，并且在下一次选举开始前等待超时时间过去；这降低了在新选举中另一个分割投票的可能性。9.3节展示了该方法快速选举一个leader。

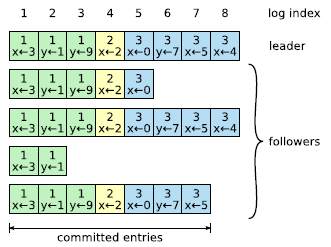


图6. 日志由序号排列的实体组成。所有实体包含了它被创造的term（每一个块中的数字）和一个状态机指令。一个实体如果能够安全的应用到状态机则被认为committed。

选举是一个可理解性是如何在设计替代方案时引导我们的选择的例子。原来我们计划使用rank系统：所有candidate被分配一个唯一的rank，会在与其他candidate竞争时拿来用。如果candidate发现另一个candidate由更高的rank，它将会回退为follower状态然后更高rank的candidate可以更简单的赢得接下来的选举。我们发现这个方法给可用性带来了一点小问题（低rank的server可能需要超时并且再次变成candidate如果更高rank的server失败，但是如果这样太快，它会重设选举leader的进度）。我们给这个算法做了多次调整，但是每次调整都会有新的问题出现。最终我们认为随机重试方法更容易被观察与理解。

**5.3 Log replication**

一旦leader被选举出来，它会开始处理client请求。所有client请求都包含了会被复制状态机执行的指令。Leader将指令作为新实体添加到它的日志，然后并发提出AppendEntries RPC给其他server去复制该实体。当该实体被安全的复制（如下描述），leader给它的状态机应用该实体并返回给client执行结果。如果follower崩溃或者运行慢，或者网络出现丢包，leader会无限重试AppendEntries RPC（即使在它回复给client之后）直到所有follower最终存储了所有日志实体。

日志就像图6中所示的一样有组织的。当实体被leader收到时，每一个日志实体单独存储状态机指令和term序号。日志实体中的term序号被用来检测日志间的不一致性和确保图3中的一些属性。所有日志实体也拥有一个表明它在日志中的位置的整数索引。

Leader决定何时给状态机应用日志实体是安全的；这样一个实体被称为已提交。Raft保证已提交的实体是耐用的并且将最终被所有可用的状态机执行。一个日志实体一旦被leader创建并在大多数server上被复制就是已提交（比如，图6中的实体7）。这也同样提交leader日志中所有的前置实体，包括由前一个leader创建的实体。5.4节讨论了一些当在leader改编后应用该规则的一些细节，也展示了承诺的定义是安全的。Leader保持跟踪已提交的最高索引，它将该索引包括在以后的AppendEntries RPC（包括心跳）中以便其他server最终可以找出。一旦follower学习了已提交的日志实体，它会将其应用到它的本地状态机（按日志顺序）。

我们设计Raft日志机制去维护一个不同server的日志间较高的相干性。不只是为了简化系统的行为和使其可预测，也是因为它是确保安全性的重要组件。Raft维护了以下属性，共同构成了图3中的Log Matching Property：

* 如果两个实体在不同的日志中有相同的索引和term，则它们存储相同的指令。
* 如果两个实体在不同的日志中有相同的索引和term，则该日志的之前所有的实体都是相同的。

第一个属性从事实来看leader在给定的日志索引和term内至多创建一个实体，日志实体不会改变在日志中的位置。第二个属性由通过AppendEntries的简单一致性检测来保证。当发送一个AppendEntries RPC时，leader先在它的日志中包括了新实体的索引和term。如果follower没有在它的日志中找到相同索引和term的实体，则它会拒绝新实体。一致性检查作为诱导步骤：初始日志的空状态满足Log Matching Property，每当日志延长一致性检查都保留Log Matching Property。结果是，每当AppendEntries成功返回，leader通过新实体就知道follower的日志和它的是一致的。

通过正常操作，leader和follower的日志保持一致，所以AppendEntries一致性检查从不失败。然而，leader崩溃可以使得日志不一致（老leader可能没有完全复制它的所有实体）。这个不一致可能在一系列leader和follower的崩溃中更复杂。图7说明follower的日志可能和新leader的不同的方式。一个follower可能丢失当前leader的实体，可能有额外的不是当前leader的实体，或两者都有。日志中丢失或外来的实体可能跨越多个term。

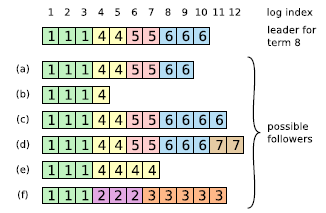


图7. 当顶上的leader上台，这些任意的场景（a-f）可能发生在follower的日志中。每个框表示一个日志实体；框中的数字是term。Follower可能丢失实体（a-b），可能有额外的未提交的实体（c-d），或者都有（e-f）。比如如果该server是term2的leader，场景（f）可能发生，在日志中添加了额外的实体，然后在提交它们前崩溃；它快速重启，变成term3的leader，然后又添加了一些实体；在term2或3的实体被提交前，server又崩溃并且在接下去的多个term中保留这些实体。

在Raft中，leader通过强制follower的日志复制它的日志来处理不一致。这意味着follower日志中冲突的实体会被来自leader的日志复写。5.4节会展示在再加上一个限制后它是安全的。

为了使follower的日志和自己的日志，leader必须找到两个日志中相同的最新的日志实体，删除follower的日志中这个点之后的所有实体，然后给follower发leader在该点之后的所有日志实体。所有这些操作都发生在AppendEntriesr RPC的一致性检查的回复中。Leader对所有follower都维护了nextIndex，即下一个将会发送给follower的日志实体的索引。当leader首次上台，它初始化所有的nextIndex值为它日志中最后一个之后的索引（图7中的11）。如果follower的日志是和leader的不一致，AppendEntries的一致性检查将会在下一次AppendEntries RPC中失败。在拒绝之后，leader降低nextIndex然后重试AppendEntries RPC。最终nextIndex将会到达leader和follower日志相匹配的点。当该情况发生时，AppendEntries将会成功，也就消除了所有follower日志中的的冲突实体并添加leader的日志中的实体（如果有）。一旦AppendEntries成功，follower的日志也就leader的一致，并会一直在当前term的剩余时间内保留该方式。

如果需要，协议可以通过减少拒绝的AppendEntries RPC数量来优化。比如，当拒绝一个AppendEntries请求，follower可以包括冲突实体的term和它存储的该term的第一个实体的索引。有了这个信息，leader可以省略所有该term的冲突实体来减少nextIndex；一个AppendEntries RPC将会被所有有冲突实体的term要求，比每个实体一个RPC好。在实践中，我们怀疑这个优化是否必要，因为失败不常发生并不会有很多冲突实体。

有了这个机制，leader不需要在它上台后执行任何特殊行动去恢复日志一致性。只需要正常开始，日志自动汇集于AppendEntries一致性检查失败的回应。Leader不需要在它的日志中复写或删除实体（图3的Leader Append-Only Property）。

日志复制机制展示了第2节中描述的理想的共识属性：Raft可以接收，复制，并应用新日志实体只要多数的server启动；在正常情况下一个新实体可以通过一轮给集群中的多数的RPC来被复制；一个单独的运行慢的follower不会影响性能。

**5.4 Safety**

上一节描述了Raft如何选举leader和复制日志实体。然而，到目前为止描述的机制都还不能完全足够的确保每一个状态机以相同顺序执行相同的指令。举例来说，一个follower可能在leader提交一些日志实体时是不可用状态，然后可能被选为新leader并用新实体复写了这些旧的；结果是，不同的状态机可能执行了不同的指令序列。

本节通过添加限制什么server可以被选为leader来完善Raft算法。该限制确保了任意term的leader包含之前所有term所有已提交的实体（图3的Leader Completeness Property）。有了该选举限制，我们可以完善提交的规则使其更精确。最终，我们证明Leader Completeness Property概念原型并且展示它是如何引导复制状态机做正确行为的。

**5.4.1 Election restriction**

在所有基于Leader的共识算法中，leader必须最终存储所有已提交的日志实体。在一些共识算法中，比如Viewstamped Replication[22]，一个leader即使原来没有包含所有的已提交的实体也可以被选举。这些算法包含了额外的机制去鉴定丢失的实体和将他们发送给新leader，可以在选举期间或者之后。不幸的是，这导致了大量的额外机制和复杂度。Raft使用了一个简单的方法保证所有先前term的已提交的实体从选举的时刻已经在新leader处了，不需要转发这些实体给leader。这意味着日志实体仅仅流入一个方向，从leader到follower，而且leader绝不重写它们日志中存在的实体。

Raft使用投票过程去避免candidate没有包含所有已提交的日志实体就赢得选举。一个candidate为了选举必须和集群中的多数联系，也就意味着每一个已提交的实体必须存在于这些server的至少一个中。如果candidate的日志至少和其他多数的日志一样是最新的（“最新”在之前有明确定义），然后它将持有所有已提交的实体。RequestVote RPC实现了该限制：RPC包含了candidate的日志信息，如果投票者自己的日志比candidate的更新则否认candidate的投票。

Raft通过比较日志中最新实体索引和term来决定哪两个日志是最新的。如果日志最新的实体有不同的term，则有最新term的日志是最新的。如果日志以相同term结束，则更长的日志是最新的。

**5.4.2 Committing entries from previous terms**

如5.3节描述的，leader知道一个当前term的实体一旦被大多数server存储则它就是已提交的。如果leader在提交实体前崩溃，未来的leader将会尝试结束复制实体。然而，leader不能立即认为之前term的实体一旦在多数server上存储就是已提交的。图8说明了一个旧日志实体在多数server上存储的场景，但是它仍然可以被后面的leader复写。

为了消除图8那样的问题，Raft绝不通过计算副本数量来提交之前term的日志实体。只有leader当前term的日志实体通过计算副本数量来提交；一旦一个当前term的实体以该方式提交，则由于Log Matching Property，所有先前的实体间接来说是已提交的。这有一些leader可以安全的认为一个旧日志实体是已提交（举例，如果该实体在所有server上存储）的场景，但是Raft为了简单采取了更保守的方法。

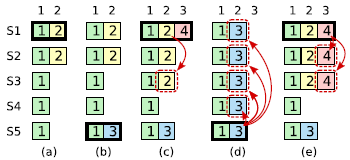


图8. 一个展示了为什么leader不能使用之前term的实体来决定已提交的时间序列。在（a）S1是leader和部分在索引2复制日志实体。在（b）S1崩溃；S5是term3的leader，收到了S3，S4和它自己的投票，并且收到了在索引2的不同日志实体。在（c）S5崩溃；S1重启，是新的leader，继续复制。在这个点，term2的日志实体被复制在多数server上，但是未提交。如果S1在（d）崩溃，S5可以被选举为leader（S2，S3，S4投票）并且用它自己的term3的日志实体来复写。然而如果S1在崩溃前在它当前term的多数server上复制一个实体，就像（e），接着该实体被提交（S5不能赢得选举）。此时所有日志中之前的实体都是已提交的。

Raft因为在leader复制之前term的实体时日志实体保留它们原始term序号而在提交规则上招致额外的复杂性。在其他共识算法中，如果新leader复制之前term的实体，它必须以当前term标号。Raft的方法使其更容易去推出日志实体，因此它们即使时间推移和跨日志也保持相同的term序号。另外，Raft的新leader发送更少的之前term的日志实体（其他算法必须发送冗余的日志实体去在它们可提交前重新计算它们）。

**5.4.3 Safety argument**

鉴于完整的Raft算法，我们可以更准确的讨论持有Leader Completeness Property（该争论基于安全证明；见9.2节）。我们假设没有Leader Completeness Property，然后我们证明矛盾。假设term T的leader（leaderT）提交一个当前term的日志实体，但是该日志实体没有通过之后term的leader存储。考虑最小term U>T其中leader（leaderU）没有存该实体。

1. 当leaderU选举时，该已提交的实体必须不在它的日志中（leader绝不会删除或重写实体）。
2. LeaderT在集群的多数上复制该实体，leaderU收到集群多数的投票。因此，至少一个server（“the voter”）同时收到了leaderT的实体并投给了leaderU，如图9所示。该投票者时研究该矛盾的关键。
3. 投票者必须在它投给leaderU前收到leaderT发的已提交的实体；否则它将会拒绝leaderT的AppendEntries RPC（它的当前term比T高）。
4. 投票者在它投给leaderU时仍然存储实体，因此所有介入的leader都包含实体（通过假设），leaders从不删除实体，follower只会在与leader冲突时删除实体。
5. 投票者投给leaderU，所以leaderU的日志必须和投票者的保持更新。这导致了两个矛盾之一。
6. 首先，如果投票者和leaderU共享相同的最新日志term，然后leaderU的日志必须至少等于投票者的，所以它的日志包含所有投票者日志中的实体。这就是一个矛盾，因为投票者包含已提交实体而leaderU被假设为没有。
7. 除此之外，leaderU的最新日志term必须比投票者的大。此外，它要比T大，因为投票者的最新日志term至少是T（它包含了term T的已提交日志）。更早的创造了leaderU的最新日志实体的leader必须在它日志中包含了该已提交实体（通过假设）。接着，通过Log Matching Property，leaderU的日志必须同样包含该已提交实体，这又是一个矛盾。
8. 这完成了矛盾。因此所有term大于T的leader必须包含所有T中提交的实体。
9. Log Matching Property保证未来的leader将同样间接包含所以已提交的实体，如图8（d）中的索引2。

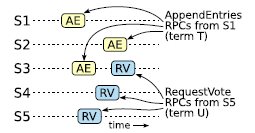


图9. 如果S1（leaderT）提交一个新日志实体，S5是之后的term U的leader，则必须由至少一个server（S3）收到该日志实体并投票给S5。

鉴于Leader Completeness Property，我们可以证明图3的状态机安全性属性，即如果一个server在它状态机上应用了一个给定索引的日志实体，则其他server不会在相同索引应用不同的日志实体。在server给它状态机应用日志实体的时候，它的日志通过实体和肯定被提交的实体和leader的日志保持一致。现在考虑任意server应用一个给定日志索引的最低term；Log Completeness Property保证所有更高term的leader将会存储相同的日志实体，所以server在后续term使用索引将会应用相同的值。因此，状态机安全属性也就保证了。

最后，Raft要求server按照索引顺序应用实体。和状态机安全性属性组合，这意味着所有server将会给它们的状态机以相同顺序应用相同日志实体集合。

**5.5 Follower and candidate crashes**

我们一直在关注leader失败。Follower和candidate崩溃比leader崩溃更容易处理，并且可以用同样的方法处理。如果follower或者candidate崩溃，则之后对其的RequestVote和AppendEntries RPC调用将会失败。Raft通过无限重试来处理这种失败；如果崩溃的机器重启，则RPC将会完全成功。如果server在完整一个RPC之后，回复前崩溃，则它将在它重启后收到同样的RPC请求。Raft的RPC是幂等的，所以这不会有什么问题。举个例子，如果一个follower收到一个AppendEntries请求包含了早就在其日志中的日志实体，它会忽略新请求中的日志实体。

**5.6 Timing and availability**

我们对Raft的要求之一是安全性必须不依赖于时机：系统必须不因为一些事件发生的比预期更快或慢而产生任何错误结果。然而，可用性（系统及时回复client的能力）必须依赖于时机。举个例子，如果消息交换由于server崩溃而消耗超过正常时间，candidates将不会有足够的时间赢得选举；没有稳定的leader，Raft不能有所进展。

Leader选举是Raft受到时机影响最严重的一方面。Raft将可能选举和维护一个稳定的leader只要系统满足该时机要求：broadcastTime<<electionTimeout<<MTBF（Mean Time Between Failure）。在这个不等式中，broadcastTime是一个server并发发送RPC给集群中每一个server并且收到它们的回复的平均消耗时间；electionTimeout是5.2节中描述的选举超时时间；MTBF是单个server失败之间的间隔平均时间。这个广播时间应该比选举超时小一个数量级，这样leader可以可靠的发送心跳信息避免follower开始选举；鉴于随机数方法用来做选举超时，该公式同样使得投票分裂情况不太可能出现。选举超时应该比MTBF小一个数量级，这样系统可以有稳定的进展。当leader崩溃时，系统将会在大约选举时间范围内不可用；我们希望这只是总体时间中的一小部分。

广播时间和MTBF底层系统的属性，选举超时时间是我们必须选择的。Raft的RPC一般要求接收者将信息持久化存储，所以广播时间可能在0.5ms到20ms范围内，依赖于存储技术。结果是，选举超时时间大约是10ms到500ms之间。一般服务器的MTBF是几个月或更久，轻松满足时机要求。

**6. Cluster membership changes**

到现在为止我们假设了集群配置（参与共识的server集合）是固定的。在实际中，偶尔修改配置是必要的，比如在server失败时为了替换server或者修改复制的等级。虽然这可以通过下线整个集群来实现，但是这会导致集群在更改期间不可用。而且，如果其中有手动的步骤，会存在操作失误的风险。为了避免这些问题，我们决定自动修改配置并且将其集成到Raft算法中。

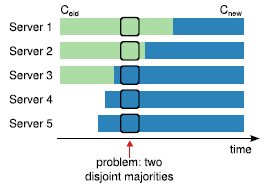


图10. 从一个配置直接切换至另一个配置是不安全的，因为不同的server将会在不同时间切换。在这个例子中，集群从3个server涨到5个。不幸的是，时间线上有个点可以让两个不同的leader在同一个term被选举出来，一个是老配置（Cold）的多数投的票，另一个是新配置（Cnew）的多数投的票。

为了使配置更改机制安全，必须在转换时不可能存在相同term存在两个leader的时间点。不幸的是，任何直接切换新旧配置的方法都是不安全的。不可能一次性自动切换所有server的配置，所以集群存在切换时分裂成两个独立的集合的潜在危险（见图10）。

为了确保安全性，配置更改必须使用两阶段方法。有很多方法可以实现两阶段。比如，一些系统（如[22]）使用第一阶段去关闭老配置这样就不会处理client请求；然后第二阶段开启新配置。在Raft中，集群首先切换至一个中间配置，我们称之为（共同共识）joint consensus；一旦共同共识被提交，系统接着转换为新配置。共同共识同时组合了新老的配置：

* 日志实体在两个配置中都会被拷贝到所有server。
* 两个配置的任意server都可能作为leader。
* 协议（选举和实体提交）要求从新旧配置中分离多数。

共同共识允许独立的server在不同时间转换不同配置而不损害安全。此外，共同共识允许集群在切换配置时持续处理客户端请求。

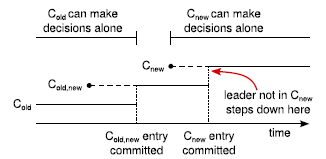


图11. 配置切换的时间线。虚线表示配置实体被创建但未提交，实线表示最新被提交的配置实体。Leader首先创建Cold,new配置实体并提交给Cold,new（Cold的多数和Cnew的多数）。接着创建Cnew实体然后提交给Cnew。这样旧没有Cold和Cnew同时可以独立做决定的时间了。

集群配置使用复本日志中的特殊实体来存储和通信；图11描绘了配置更改过程。当leader收到切换Cold为Cnew配置的请求，它存储共同共识的配置（图中Cold,new）作为一个日志实体并且使用之前描述的机制复制该实体。一旦给定的server在其日志中添加了该新配置，它会用该配置来做之后的决策（server总是使用日志中最新的配置，而不管哪个实体是已提交的）。这意味着leader将使用Cold,new的规则来决定何时Cold,new被提交。如果leader崩溃，新leader可能是Cold或Cold,new的配置，取决于获胜的candidate是否收到了Cold,new。在任何情况下，Cnew不可能在此期间做单方面的决定。

一旦Cold,new提交，Cold或Cnew都不能做决定，Leader Completeness Property确保只有Cold,new实体的server可以被选为leader。现在对于leader创建日志实体去描述Cnew并且将其复制到集群中是安全的。再一次，配置将会在见到该实体的server上起效。当新配置在Cnew的规则下被提交，旧配置也就无效了，没有使用新配置的server也可以被关闭。就像图11所示，没有Cold和Cnew可以单方面做决定的时间；这保证了安全性。

这有三个问题要解决重新配置。第一个问题是新server可能没有存储任何日志实体。如果他们此时被添加到集群中，需要一段时间去跟上，此时可能不能提交新实体。为了避免可用性问题，Raft在配置改变前采用了额外的阶段，即新加入集群的server作为不能投票的成员（leader将日志实体复制给它们，但是它们不认为是多数中的一部分）。一旦新server跟上了集群中其他那些，重新配置可以按上述的继续。

第二个问题是集群leader可能不是新配置的一部分。此时，leader在其提交Cnew实体后回退（返回follower状态）。这意味着有一段时间（当期正在提交Cnew）是leader管理集群但是不包括它自己；它复制日志实体但是不将其自己算入多数。Leader转换发生在Cnew被提交时，因为这是新配置独立可以运转的第一点（它将一直可以选择Cnew的Leader）。在该点之前，可能是只有Cold的server可以被选举为leader的情况。

第三个问题是移除的server（不在Cnew）可以破坏集群。这些server不会收到心跳，所以它们将会超时并开始新选举。他们将会使用新term的序号接着发送RequestVote RPC，可能导致当前的leader回到follower状态。新leader将最终被选举出，但是移除的server将会再次超时并且重复进程，导致低可用性。

为了避免这个问题，server在其相信当前leader存在时无视RequestVote RPC。特别是，如果server在收取当前leader心跳的最小选举超时时收到RequestVote RPC，它不会更新它的term或者投票。这不会影响正常选举，所有server在开始新选举时最少等待选举超时时间。然而，这帮忙避免了已移除server的干扰：如果leader可以收到它集群的心跳，则它不会因为更大term序号而废弃。

**7. Log compaction**

Raft的日志在正常操作记录更多的client请求时增长，但是在实际系统中不可能无限增长。因为日志越来越长，它消耗更多空间且需要更多时间去再现。这最终会导致可用性的问题如果没有任何机制去丢弃日志中堆积的过时信息。

快照是最简单的方法去实现。在使用快照时，整个当前系统状态被写入快照中并稳定存储，然后该点之前的整个日志都被消除。快照在Chubby和ZooKeeper中使用，本章接下去的章节描述Raft中的快照。

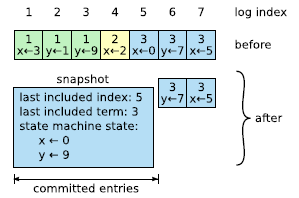


图12.一个server用一个新的快照替换它日志（索引1到5）中已提交的实体，该快照仅包含了当前状态（在该例中是x和y的值）。该快照最后包含的索引和term是为了定位日志中实体6前的快照。

压实的增量方法，比如日志清除[36]和日志结构化合并树[30,5]，也是可能的。这些操作每次对数据的一小部分操作，所以随着时间推移更均匀传播压实负载。它们首先选出一个积累了很多已删除和复写对象的数据区域，接着它们重写该区域中存活的对象使其更紧实并释放该区域。这相对快照要求更重要的额外机制和复杂度，快照每次都操作整个数据集合。相对日志清除需要修改Raft，状态机可以使用和快照相同的接口实现LSM树。

图12展示了Raft中快照的基本想法。每一个server独立进行快照，仅覆盖它日志中已提交的实体。大多数工作是由状态机将其当前状态写入快照组成。Raft在快照中也包括了一小部分元数据：最后包括的索引（last included index）是日志中快照替换的最后一个实体的索引（状态机最后应用的实体），最后包括的term（last included term）是该实体的term。这些被保留以支持对快照之后第一个日志实体的AppendEntries一致性检查，因此该实体需要之前的日志索引和term。为了支持集群成员更改（第6节），该快照同时也包括了日志中最新配置作为最后包括的索引。一旦一个server完成了写快照，它可能删除该索引之前所有日志实体，以及之前所有快照。

虽然server正常是独立进行快照，但leader必须偶尔在follower落后时给它发送快照。这发生在leader早就已经删除了要发送给follower的下一个日志实体时。幸运的是，这种场景在正常情况下时不会出现：一个保持更新leader的follower应该早就有这个实体了。然而，一个意外缓慢的follower或者新server加入集群（第6节）时不会有该实体。将这种follower更新至最新状态需要leader通过网络发送快照给它。

|  |
| --- |
| **InstallSnpshot RPC** |
| 由leader调用来发送快照的块给follower。Leader总是按照顺序发送块。  参数：  term： leader的term  leaderId： follower以此重定向clients  lastIncludedIndex： 快照会替换所有通过或包含此索引的实体  offset： 块在快照文件中的位置字节偏移  data[]： 原始快照文件块的字节，从偏移位开始  done： 如果是最后一块则为true  结果：  term： 当前term，给leader去更新它自己  接收者实现：   1. 如果term＜currentTerm，立即回复 2. 如果是第一个块（offset为0）则创建新快照文件 3. 在快照文件中按照给定位移写数据 4. 如果done为false则回复并等待更多的数据块 5. 保存快照文件，消除任何已存在或局部更小索引的快照文件 6. 如果存在和快照最后一块实体一样索引和term的日志实体，保留之后的日志实体并回复 7. 消除整个日志 8. 使用快照内容重设状态机（并加载快照的集群配置） |

图13. InstallSnapshot RPC的总结。快照分为多块来传输；这使follower成为每一块生命的标志，所以它可以重设它的选举计时器。

Leader使用叫做InstallSnapshot的新RPC来发送快照给差距太大的follower；见图13。当一个follower通过该RPC收到快照，他必须决定如何处理它已存在的日志实体。一般快照将包含不在接收者日志中的新信息。在这个场景中，该follower消除它的整个日志；它完全被快照所取代且可能由未提交和实体和快照重复了。如果follower收到描述了它日志前面的快照（由于重发或失误），则被快照所覆盖的日志实体被删除但是跟着快照的实体仍旧有效且必须被保留。

该快照方法背离了Raft的强大leader原则，因为follower可以不需要leader的知识就进行快照。然而，我们认为这样的离开是合理的。Leader有助于在达成共识时避免冲突的决定，共识早就在快照时达成了，所以不会有决定冲突。数据仍旧只从leader流向follower，只是现在follower可以自己改组它的数据。

我们考虑过一个替代基于leader的方法，只有leader可以创建快照，接着它将该快照发给它的follower。然而，这样有两个缺点。首先，发送快照给所有follower将会浪费网络带宽并减慢快照的过程。每个follower早就有可以产生它自己的快照所需的信息，并且一般而言比server用它当前状态产生快照并通过网络传输消耗更少。其次，leader的实现将会更复杂。比如leader将会需要在有新日志实体时并发发送快照给follower，一面组织新的客户端请求。

这里还有两个影响快照性能的问题。首先，server必须决定何时进行快照。如果server快照的太频繁，将会浪费硬盘空间和能源；如果太慢，有可能耗尽它的存储空间，并且在重启时增加了需要回放的数量。一个简单的方法是当日志达到一个固定的大小时进行快照。如果大小设置的比快照的期望大小大很多，则快照的磁盘空间消耗就很小。

第二个性能问题是写快照需要消耗大量时间，并且我们不希望这会延迟正常的操作。解决方法是使用写入时复制（copy-on-write）技术，这样新更新可以不需要影响写快照就被接受。比如，状态机与函数式数据结构组合天生支持这一点。或者操作系统的写入时复制支持（比如，Linux上的fork）可以被用来创建一个在内存中的整个日志的快照（我们的实现使用了这个方法）。

**8. Client interaction**

本节描述client如何与Raft交互，包括client如何查找集群leader和Raft如何支持线性化语义[10]。这些问题适用于所有基于共识的系统，而且Raft的解决方法和其他的相似。