**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：

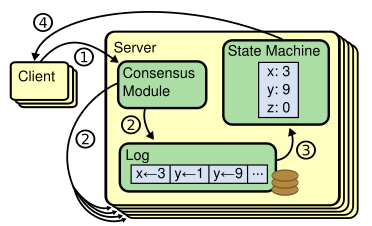
* Strong leader：Raft相对其他共识算法使用了更强的leadership构成。举例来说，log实体仅仅从leader到其他servers。这样简化了复制log的管理并且使得Raft更容易理解。
* Leader election：Raft使用随机定时器选举leader。该方式仅增加了一小部分其他算法早就有的心跳机制，可以使解决冲突更简单和更快。
* Membership changes：Raft对于更改server集群中集合的机制使用新共同共识（joint consensus）方式，这会使得在过渡时有两个不同配置的集合重叠。这样允许集群在配置修改时继续正常运作。

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。

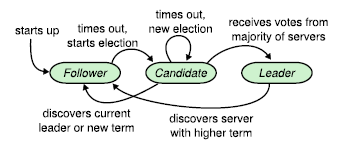


图4. Server状态。Followers只回复其他server的请求。如果一个follower不接收交流，它就变成candidate并且开始选举。从集群的多数处收到投票的candidate变成新的leader。Leader任期一般持续到它失败。

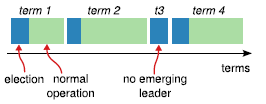


图5. 时间被分为不同term，每一个term由选举开始。在成功选举后，独立leader直到term结束一直管理集群。有些选举失败了，这时term由于没有选择一个leader而结束。可以在不同server的不同时间看到term的转换。

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。

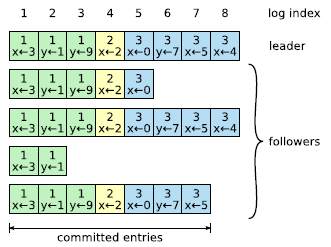


图6. 日志由序号排列的实体组成。所有实体包含了它被创造的term（每一个块中的数字）和一个状态机指令。一个实体如果能够安全的应用到状态机则被认为committed。

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。

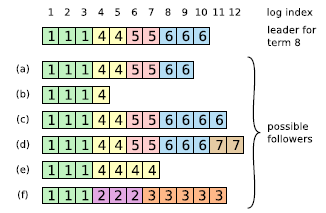


图7. 当顶上的leader上台，这些任意的场景（a-f）可能发生在follower的日志中。每个框表示一个日志实体；框中的数字是term。Follower可能丢失实体（a-b），可能有额外的未提交的实体（c-d），或者都有（e-f）。比如如果该server是term2的leader，场景（f）可能发生，在日志中添加了额外的实体，然后在提交它们前崩溃；它快速重启，变成term3的leader，然后又添加了一些实体；在term2或3的实体被提交前，server又崩溃并且在接下去的多个term中保留这些实体。

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**