# Raft

## 引言

Raft是一种用来管理日志复制的协议，它和Paxos协议的功能是一样的，并且在性能上和Paxos相差无几。相信了解过Paxos协议的人都会觉得其晦涩难懂，工程实现和协议相差甚远。而Raft算法的核心设计理念就是简单，并且更易于建立实际的系统。Raft算法为了更便于理解，将问题拆分为Leader选举、日志复制、安全性这几个部分。还提供了动态改变集群成员的机制。

## 复制状态机

一致性协议都是在复制状态机的背景下提出的。基于这样一个理论：一组初始状态相同的状态机，如果按照相同顺序执行一批命令，那么这一组状态机的最终结果一定相同。通俗来讲，可以认为分布式系统中的每个节点内部有一个独立的状态机，每个节点通过执行自己日志中的命令，来达到系统中各个节点的一致性(如图1)。使用复制状态机的例子有Chubby和ZooKeeper等。Raft算法就是解决分布式系统中日志一致性问题的。

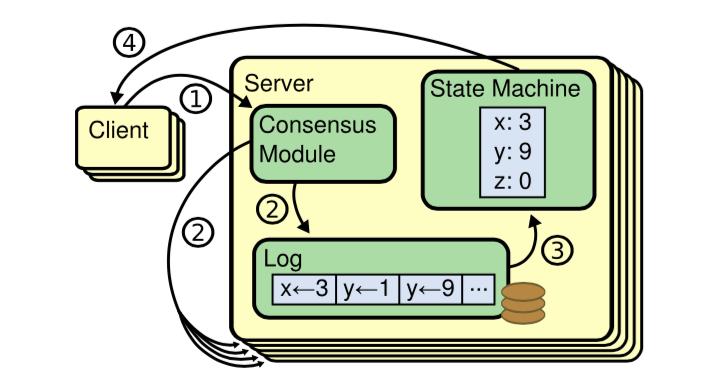


图1

## Paxos算法的不足

Paxos算法一直被认为分布式一致性算法中的鼻祖，无论是Raft算法还是其他的分布式协议，或多或少都借鉴了Paxos算法。

但是Paxos有两个缺点：

1. Paxos算法的晦涩难懂是一直以来被诟病的问题。
2. 难以在实际环境中实现，其中一个原因是对于多决策，大家还没有一个一致同意的算法。Lamport的描述大部分都是有关于单决策；他仅仅描述了实现多决策的可能的方法，缺少许多细节。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。例如，Chubby 实现的是一个类似 Paxos 的算法，但是在许多情况下的细节没有公开。

还有一点是Paxos使用点对点的实现作为它的核心(尽管提出了一种弱Leader的形式来优化性能)，这种方法通常比较适用于那些只需要一个决策被制定的情况下。如果分布式系统需要做很多决策，那么通过选择一个Leader，由Leader来协调是更简单高效的方法。

## Raft一致性算法

Raft是用来管理日志复制一致性的算法，通过选出一个Leader，由Leader完全负责管理复制日志的责任。Leader接收客户端的命令，并将命令转换为日志条目后复制到其他服务器上，Leader还需要告诉Follower什么时候将日志条目应用到状态机是安全的。在系统中Leader拥有对日志的绝对管理权，能够决定将新的日志条目放到哪，并不需要和其它服务器协商，这样能够简化复制日志的流程，当Leader宕机或者与其它服务器无法连接时，需要选取下一个Leader。接下来我们就来了解一下Raft算法。

### 角色

在Raft算法中定义了三种角色，分别为Leader、Follower和Candidate，在任意时间每个服务器一定会处于三种角色中的一个。正常情况下，只有一台服务器是Leader，剩下的服务器都是Follower。

* Leader：分布式系统中的管理者，负责处理客户端的读写请求，将写请求转换为日志条目复制到Follower上。
* Follower：Leader服务器的跟随者，转发写请求给Leader，通常为了性能考虑而允许Leader与Follower存在弱一致性的情况，会允许Follower处理读请求。接收Leader服务器发送过来的日志条目。当机器刚启动时默认为Follower，如果收到Leader的请求则与Leader建立连接，或经过超时时间后，转换为Candidate。
* Candidate：当系统中的节点经过超时时间后仍无法与Leader通信，则会转换为Candidate，发起Leader选举，根据投票结果成为Leader或者Follower。

图2展示了三种角色之间的转换。

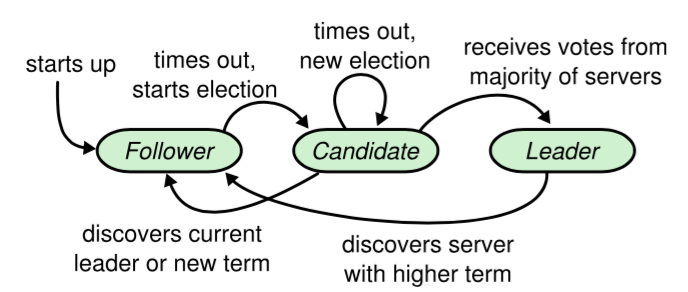


图2

简单描述一下图2：

1. 服务器启动后默认处于Follower角色，当发现Leader后与其保持跟随。
2. 当Follower经过超时时间后，转换为Candidate，开始选举，如果获得了大多数服务器的投票，则成为Leader；如果经过超时时间后仍没有获得大多数服务器的投票，则开始新一轮的选举；如果收到来自Leader的RPC请求或者收到了更高term的Candidate发来的投票请求，则转换为Follower与其保持跟随。
3. 当Leader发现更高term的服务器后，转换为Follower与其保持跟随。

### 任期term

Raft将时间分隔到任意长度的term中，每一个任期开始于选举，在选举成功后，Leader将负责管理集群直到任期结束(比如Leader宕机)，如果选举失败(比如投票被两个服务器平分)，则该任期就会因为没有Leader而结束，然后开始下一任期。

在某些情况下一台服务器可能看不到一次选举或者一个完整的任期。任期在Raft中充当逻辑时钟的角色，并且允许服务器检测过期信息，比如过时的Leader。每一台服务器都存储着当前任期的数字，这个数字会单调递增，当服务器之间进行通信时，会互相交换任期号，如果一个服务器的当前任期号比其它服务器的小，那么则更新为较大的任期号。如果一个Leader或者Candidate意识到自己的任期号过时了，那么就会转换为Follower。如果一台服务器接收到的请求中任期号是过时的，那么就会拒绝该请求。

### 状态

在Raft算法中，每个节点都有自己的一些状态，如下所示：

|  |  |
| --- | --- |
| 在所有服务器上稳定的状态(在响应RPCs请求前稳定存储的) | |
| currentTerm | 服务器已经知道的最新的任期（初始化为0在第一次启动时，单调增加） |
| votedFor | 在当前任期内收到投票的候选人 id（如果没有就为 null） |
| log[] | 日志条目集合，每个日志条目都包含状态机要执行的命令和产生该日志条目的任期 |

|  |  |
| --- | --- |
| 在所有服务器上不稳定的状态 | |
| commitIndex | 已知的被提交的最大日志条目索引值(从0开始，单调递增) |
| lastApplied | 被状态机执行的最大日志条目索引值(从0开始，单调递增) |

|  |  |
| --- | --- |
| Leader上不稳定的状态(在选举后重新初始化) | |
| nextIndex[] | 对于每一个服务器，记录需要发给它的下一个日志条目的索引（初始化为领导人最新日志条目的索引值+1） |
| matchIndex[] | 对于每一个服务器，记录已经复制到该服务器的日志的最高索引值（从0开始递增），通常在收到该服务器的ack后递增。 |

### 通信

Raft算法中各个服务器间通过RPC进行通信，基本的Raft算法只需要两种RPC，AppendEntries RPC是Leader触发的，负责复制日志和心跳消息；RequestVote RPC是由Candidate在选举期间触发的，用来进行选举。还有第三种RPC用来在服务器之间传输快照。

#### AppendEntries RPC

附加日志的RPC请求与响应，Leader用来复制日志和发送心跳。

请求：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | Leader的任期 |
| leaderId | Leader的 id，为了其他服务器能重定向到客户端 |
| prevLogIndex | 新的日志条目的前一条日志条目的索引 |
| prevLogTerm | prevLogIndex的日志条目的任期 |
| entries[] | 当前发送的日志条目集合，为了效率可能会包含多条日志，心跳请求时为空 |
| leaderCommit | Leader已经提交了的日志条目的索引 |

响应：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | currentTerm，用于Leader更新自己的任期 |
| success | 如果Follower服务器包含能够匹配上 prevLogIndex 和 prevLogTerm 的日志时为true |

接收者需要实现：

1. 如果term< currentTerm，返回false。
2. 如果在prevLogIndex处的日志的任期号与prevLogTerm不匹配时，返回 false。
3. 如果一条已经存在的日志与新的冲突（index 相同但是任期号 term 不同），则删除已经存在的日志和它之后所有的日志。
4. 添加任何在log[]中不存在的日志条目。
5. 如果leaderCommit>commitIndex，那么将commitIndex更新为leaderCommit和“最新日志条目的索引”中较小的一个，即commitIndex=MIN(leaderCommit, 最新一条日志条目的索引)。

这里解释一下第5条，因为如果leaderCommit>commitIndex，说明Follower提交的日志条目落后于Leader，需要更新Follower的commitIndex，之所以在leaderCommit和“最新一条日志的索引”选择较小的一个是因为Follower可能接收到的日志条目落后于leaderCommit，这时对于Follower来说leaderCommit索引处的日志条目是不存在的，自然无法提交，所以只能将commitIndex更新为自己最新的日志条目的索引。反之则将commitIndex更新为leaderCommit。

#### RequestVote RPC

上面了解了复制日志的RPC请求与响应，接下来再看一下Leader选举投票的RPC请求与响应。

请求：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | candidate的任期 |
| candidateId | candidate的Id |
| lastLogIndex | 最后一条日志的索引 |
| lastLogTerm | 最后一条日志的任期 |

响应：

|  |  |
| --- | --- |
| 参数 | 描述 |
| term | currentTerm，用于candidate更新自己的任期 |
| voteGranted | true表示candidate获得投票 |

接收者需要实现：

1. 如果term<currentTerm，返回false
2. 如果votedFor为null或者与candidateId相同，并且candidate的日志至少跟接收者一样新，那么candidate获得投票，返回true。

### 服务器需要遵守的规则

Raft算法定义了分布式系统中服务器需要遵守的规则，不同角色的服务器需要遵守不同的规则，才能确保分布式系统的一致性。

#### 所有服务器的规则

* 如果commitIndex>lastApplied，那么自增lastApplied，然后将log[lastApplied]应用到状态机。
* 如果RPC请求或者响应中的term T>currentTerm，那么设置currentTerm=T，然后将自己转换为Follower。

#### Followers

* 响应来自Leader和Candidate的RPC请求。
* 如果超过“选举超时时间”还没有收到来自Leader的AppendEntries RPC请求或者没有给candidate投票，则转换为candidate。

#### Candidates

* 转换为candidate后，开始选举

1. 自增currentTerm
2. 投票给自己
3. 重置选举超时时间
4. 发送RequestVote RPC请求给其它所有服务器

* 如果获得超过半数的投票，那么自己就转换为Leader。
* 如果收到了来自新Leader的AppendEntries RPC请求，那么将自己转换为Follower。
* 如果选举时间超过了“选举超时时间”，则开始新一轮的选举。

#### Leader

* 一旦成为Leader就发送空的AppendEntries RPC请求(心跳)给其它所有服务器，并且需要在空闲时间重复发送，以防止其它服务器选举超时。
* 如果收到了来自客户端的写请求，则将写请求转换为日志条目写入到日志，在该日志条目应用到状态机之后响应客户端。
* 如果Leader的最后一条日志的索引大于Follower的nextIndex，则将从nextIndex位置开始之后所有的日志条目通过AppendEntries RPC请求发送给该Follower。

1. 如果Follower返回成功，则更新该Follower的nextIndex和matchIndex。
2. 如果因为日志不一致而返回失败，则递减nextIndex后重试。

* 如果存在一个N, N>commitIndex，并且大多数match[i]≥N,并且log[N].term==currentTerm，那么更新commitIndex=N。这条规则简单来说就是当一条在当前Leader周期提出的日志被超过半数的Follower收到后，就可以提交这条日志了。后面会解释为什么是当前Leader周期提出的日志条目。

### Raft提供的保证

* **选举安全性**：同一任期最多有一个Leader。
* **Leader只能追加：**Leader绝不覆盖或者删除它自己的日志条目，只能追加日志条目。
* **日志匹配原则：**如果两条日志拥有相同的index和term，那么这两条日志完全相同，并且从日志开头到这两条日志的index处的所有日志条目也完全相同。
* **Leader完整性：**如果一条日志在给定的term被提交了，那么这条日志一定会出现在更高term的Leader的日志中。
* **状态机安全性：**如果一台服务器将某个索引位置的日志条目应用到了状态机，那么其他服务器不会在相同索引位置应用不同的日志条目到状态机。

在Raft算法深入分析一节中会对Raft提供的这些保证进行论证。

### Raft算法深入分析

我们已经了解了Raft算法的一些基本概念，接下来就来深入了解一下Raft算法。在上面我们提到了Raft算法为了方便理解，基础Raft算法主要分为Leader选举、日志复制和安全性三个部分，下面就会从这三个方面来深入了解Raft算法。

为了方便理解，我们下面将使用一个假设由五台服务器组成的分布式系统进行举例，这样一个系统最多允许两台服务器宕机(Leader选举的过半原则)。

#### Leader选举

Raft使用心跳机制来触发选举，当服务器启动时默认为Follower，一台服务器只要能收到来自Leader或者Candidate有效的RPC请求，就会一直保持Follower状态。Leader会周期性地发送心跳(不包含任何日志条目的AppendEntries RPC)给其它所有服务器，从而保持自己的Leader地位。如果Follower在超过“选举超时时间”后还没有收到任何有效的通信，那么它就认为没有可行的Leader，从而开始进行Leader选举。

开始选举后，Follower会自增自己的term，并且转换为Candidate，然后先自己给自己投一票，再并发的发送 RequestVote RPC请求给集群中其它所有的服务器，然后会一直保持Candidate状态直到发生以下三种情况之一：(a)赢得本次选举 (b)其它服务器成为了Leader (c)直到过了选举超时时间仍没有选举出Leader。下面将分开讨论这三种情况。

情况a，当一个Candidate在同一term获得集群中超过半数的投票，那么就竞选成功了。每一个服务器在给定的term都只能投一票，采用先到先得的原则，会将投票投给第一个收到的RequestVote RPC请求的发送者(后面会增加一条规则)。基于这样的规则就可以保证在给定的term最多只有一个Candidate可以赢得选举成为Leader，符合了**选举安全性**。一旦一个Candidate赢得了选举，那么它就成为了Leader。然后通过不断发送心跳消息从而保持自己的Leader地位并且阻止其它服务器发起新一轮选举。

情况b，在Candidate等待期间，可能会收到来自另一台声称自己是Leader角色的服务器发来的AppendEntries RPC请求。如果这个Leader的term(包含在AppendEntries RPC中)至少和当前Candidate一样大，那么当前Candidate就认可这个Leader是合法的，从而转换为Follower与这个Leader保持跟随状态。如果这个Leader的term小于当前Candidate的term，则拒绝这个Leader发来的RPC请求，保持Candidate状态继续等待。

情况c，Candidate即没有赢得选举也没有输掉选举，这种情况可能是多个Follower同时成为了Candidate，投票被平分给了多个Candidate，这种情况也就没有一个Candidate能够获得超过半数的投票，直到选举超时。当出现这种情况时，每个Candidate会再次对term自增后重新发起新一轮选举。然而，如果没有额外的措施的话，投票被平分的情况可能会无限循环。

针对投票被平分的情况，Raft使用随机的“选举超时时间”，确保出现投票被平分的情况是罕见的，并且能被快速解决。首先，为了阻止投票被平分，选举超时时间是从固定的时间间隔中随机选取的(比如150-300ms)，这种机制就使得在大多数情况下只有一台服务器会率先超时，然后这台服务器在其它服务器到达选举超时时间之前赢得了选举成为Leader后发送心跳消息给其它服务器。这个机制也同样适用于出现投票被平分的情况，每个Candidate在开始选举的时候都会重置一个随机的选举超时时间，并且在开始下一轮选举之前一直等待直到选举时间超时；这样就减少了在新一轮选举中出现选票平分的可能。

#### 日志复制

一旦Leader被选举出来，就可以接收客户端的请求了。每个客户端请求都包含一个可被复制状态机执行的命令，Leader将客户端请求中的命令作为新的日志条目添加到自己的日志中文件中，然后并发发送AppendEntries RPC请求给集群中的其它机器，要求其它服务器复制这个日志条目。当这个日志条目被安全的复制后(后面会解释何为被安全的复制)，Leader就可以将该条日志中的命令应用到状态机中了，然后将执行结果返回给客户端。如果Follower宕机或者运行缓慢，或者因为网络问题丢包，Leader将不断重发AppendEntries RPC请求，直到Follower的日志与Leader保持一致。

日志的结构如图3所示，每个日志条目都包含了以下内容：

1. 需要被状态机执行的命令
2. 该日志条目被创建时的term，被用来检测日志间的一致性，以及保证**日志匹配原则**
3. 一个整数索引，以表示其在日志文件中的位置

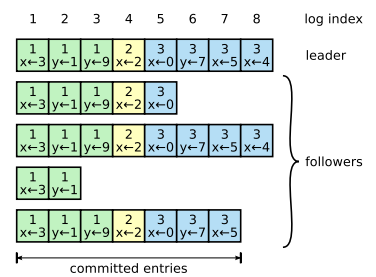


图3

Leader需要确定什么时候将日志条目应用到状态机是安全的，我们称这样的日志条目是committed(已提交的)。Raft保证所有committed的日志条目满足持久性，并且最终会被所有可用的状态机执行。基于这样的要求，当一个**由该Leader创建的日志条目**被复制到了集群中过半的机器上时就可以被提交了(例如图3中index为7的日志条目)，同时Leader日志中在它之前的日志条目也都可以提交了，在后面的安全性中会解释为什么要强调是由该Leader创建的日志条目。同时Leader还维护了已知的已提交的最大日志条目的索引，并且会包含在AppendEntries RPC请求中，以便Follower能够发现被提交的日志条目，当服务器发现一个日志条目是已提交的，那么就会将其应用到自己的状态机(按照日志中的顺序)。

通过这样的日志机制可以保证不同服务器上的日志一致性，Raft保证了以下两个特性，这两个特性共同构成了**日志匹配原则**：

* 如果在不同日志中的两个日志条目具有相同的index和term，那么这两个日志条目存储的命令一定相同。
* 如果在不同日志中的两个日志条目具有相同的index和term，那么它们在该日志条目前的所有日志条目也都一致。

第一条特性源于每个Leader在给定的term和index只能创建一个日志条目，并且日志条目的位置不会改变。

第二个特性源于AppendEntries RPC的一致性检查，当发送一个AppendEntries RPC请求时，Leader会把新增加的日志条目的前一个日志条目的index和term也包含在里面，如果Follower在自己的日志中无法找到相同index和term的日志条目，那么就会拒绝这个新的日志条目。一开始空的日志肯定满足日志匹配原则，一致性检查可以保证日志添加时满足日志匹配原则，当AppendEntries RPC返回true时，Leader就知道Follower的日志和自己是一致的了。

在正常情况下，Leader和Follower的日志保持一致，所以AppendEntries RPC的一致性检查不会失败，但是当Leader宕机的时候就可能带来日志不一致(Leader可能还没将它的全部日志复制到所有Follower)。如图4所示，Follower的日志和新的Leader的日志不一致，Follower可能缺少日志，可能比Leader日志多，或者两者都有。

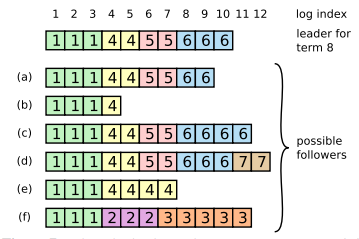


图4.一个小方框代表一个日志条目，小方框中的数字代表创建该日志条目的term

(a)和(b)展示了Follower日志缺少的情况，(c)和(d)展示了Follower日志中存在一些未提交的日志条目，(e)和(f)展示了两种情况都有的情况。

在Raft算法中，Leader通过强制Follower复制它的日志来处理日志间的不一致，这意味着Follower上冲突的日志将被Leader覆盖，在安全性一节中会再添加一条限制以保证这样做是安全的。

Leader通过AppendEntries RPC一致性检查找到Follower与自己相同的最新的日志条目，Follower删除该日志条目后的所有日志，然后Leader将自己该位置后的所有日志发送给Follower。具体的，Leader为每个Follower都维护了一个nextIndex，表示将要发送给Follower的下一个日志条目的索引，当Leader刚被选举出来时，Leader会初始化所有的nextIndex为自己的最大日志条目的索引+1(如图4中Leader的nextIndex被初始化为11)。如果Follower与Leader的日志不一致，就会在下一次AppendEntries RPC请求的一致性检查就会失败，然后Leader递减该Follower的nextIndex后再次重试，直到找到Leader和Follower日志能够匹配的位置，这时一致性检查就会成功，Follower会删除该位置后的所有日志，然后Leader把自己日志中该位置后的所有日志发送给Follower(如果有的话)，一旦日志同步成功，该Follower和Leader的日志就保持一致了，并且在当前term一致保持这样。

在作者的论文中也提到了可以进行优化从而减少AppendEntries RPC一致性检查失败的次数，但是作者考虑这不是必要的，因为在实际情况中这种情况很少发生。

可以看出Raft的这种日志复制机制是比较简单的，不需要特别的操作就可以恢复日志的一致性，仅通过正常的AppendEntries RPC请求的一致性检查就可以恢复日志一致性。并且Leader通常只需要一轮AppendEntries RPC请求就可以将日志条目复制到集群中过半的服务器上，一台较慢的机器不会影响整体性能。

#### 安全性

通过前面的内容，我们了解了Leader选举和日志复制，但是到目前为止前面提到的机制与约束并不能确保集群中的所有状态机都以同样的顺序执行相同的命令，比如，某台Follower宕机了，在此期间Leader提交了一些日志条目后宕机了，此时该Follower重启后成为了新的Leader，并用新的日志条目覆盖了之前Leader提交的日志条目，这样就可能造成不同的状态机执行了不同的命令。

本节我们就通过附加一条Leader选举的约束来完善Raft算法，这个约束确保新选举出的Leader必须包含所有已经提交的日志条目(**Leader完整性**)。

Raft使用投票的方式来阻止没有包含所有已提交日志条目的Candidate成为Leader。具体的，一个Candidate至少需要与集群中超过半数的服务器通信才能竞选成功，而一个日志条目至少需要被集群中超过半数的服务器复制才能被提交，所以这两个超过半数的集合的交集一定不为空，如果Candidate的日志至少与集群中过半服务器上的日志一样新，那么就可以保证该Candidate一定包含所有已经提交的日志条目。所以Candidate发送的RequestVote RPC请求中包含了自己日志信息，并且收到RequestVote RPC请求的服务器需要判断该请求中的日志信息至少跟自己一样新才会投票给它，否则则拒绝该请求。

上面说到了需要在投票时对比哪个日志更新，那么怎么样的日志才算是更新的呢？Candidate会在RequestVote RPC请求中包含自己最后一条日志的index和term，接收到请求的服务器会用自己日志中最后一条日志与之对比，先根据term判断，term大的日志就更新；如果term相同，则根据index判断，index大的日志就更新。

在日志复制一节中讲到日志条目提交时特意强调了“与Leader相同term的日志条目才可以通过统计日志条目是否被集群中过半服务器复制来判断是否可以提交”。对于之前term创建的日志条目，Leader无法通过统计是否被集群中过半服务器复制来判断该日志条目是否可以提交，因为存在如图5的情况。

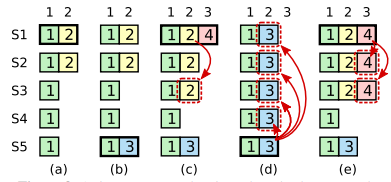


图5.

如图5的时间序列说明了之前term的日志条目为什么Leader无法通过统计是否被集群中过半服务器复制来判断该日志条目是否可以提交。(a)、(b)、(c)、(d)、(e)代表时间序列，在(a)中S1作为Leader复制了索引2上的日志条目到部分服务器。在(b)中S1宕机了，S5成为了新的Leader（通过S3、S4和自己的投票），并且在索引2上接收了另一条日志条目。在(c)中S5宕机了，S1重启后竞选成为Leader(通过S2、S3和自己的投票)，此时将自己索引2处的日志继续复制，此时来自term2的日志条目已经被复制到了大多数服务器上，但是它并没有被提交。假设在(d)中S1又宕机了，S5重启后竞选成为Leader(通过S2、S3、S4和自己的投票)，并且使用自己term3的日志覆盖了其它服务器上的日志。然而，如果S1在宕机之前复制了一个在它自己term创建的日志条目到集群中过半的服务器上，就像(e)中那样，那么这个日志条目就是被提交的(S5无法竞选成功)，同时这个日志条目之前的所有日志(term2的日志)也都是被提交的。

所以，为了避免图5中的问题，Raft不会根据统计复制数来提交之前term的日志条目，只有当前term的日志条目才可以根据统计复制数来提交。一旦一个日志条目被提交了，那么根据日志匹配原则，这个日志条目之前的所有日志也就被间接地提交了。当然，在一些情况下Leader是可以安全的判断之前term的日志条目可以被提交，比如说该日志条目已经被复制到了所有服务器上，但是这样会带来复杂性，所以Raft为了简单性，采用了一种保守的方式。

#### 安全性论证

##### Leader完整性论证

我们已经得到了完整的Raft算法，现在需要对**Leader完整性原则**进行论证。首先我们假设Leader完整性原则是不成立的，然后推导出矛盾。假设任期 T的Leader(LeaderT)提交了一个日志条目，但是这个日志条目并没有存储在之后任期的Leader中，假设大于任期T的最小的任期U的Leader(LeaderU)没有存储这个日志条目。

1. 在LeaderU选举的时候，其一定不包含这个日志条目，因为Leader从来不会删除和覆盖自己的日志。
2. 因为日志提交规则，LeaderT一定把该日志条目复制到了过半的服务器上；因为选举规则，LeaderU一定收到了过半服务器的投票；那么就一定至少有一个复制了该日志的服务器投票给了LeaderU，如图6所示，这个投票者是推导出矛盾的关键。
3. 投票者收到来自LeaderT的日志条目一定是在任期U之前，否则它将会拒绝复制该日志条目(因为当前的任期U比T大)。
4. 根据投票规则，LeaderU的日志至少要跟投票者的日志一样新，投票者才会投票给LeaderU，这就出现了矛盾。
5. 首先，如果LeaderU的最后一条日志的term与投票者相同，那么LeaderU的日志肯定至少与投票者一样长，所以LeaderU的日志就包含了投票者的所有日志条目，这就矛盾了，因为投票者包含了被提交的日志条目。
6. 另外，如果LeaderU的最后一条日志的任期比投票者的最后一条日志的任期大，根据投票规则，LeaderU最后一条日志的任期肯定大于T，因为投票者最后一条日志的任期为T(投票者包含了任期T提交的日志条目)。创建LeaderU中最后一条日志的上一任Leader其必须包含了已提交的日志条目(基于假设)。那么，根据日志匹配原则，LeaderU一定包含已经提交的日志条目，这就矛盾了。
7. 这样就完成了矛盾推导，所以所有任期大于T的Leader一定包含了在任期T提交的日志条目。
8. 根据日志匹配规则，之后任期的Leader也同样包含被间接提交的日志条目。

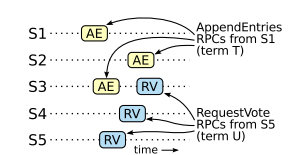


图6.如果S1(任期T的Leader)提交了一个日志条目，那么必然将该日志条目复制到了过半的服务器上；如果S5选举成为任期U的Leader，那么就肯定至少有一个复制了该日志条目的服务器投票给了S5，即图中的S3。

##### 状态机安全性论证

根据Leader完整性，我们就可以证明状态机安全性，也就是如果一个服务器将日志中某个索引位置的日志中的命令应用到了状态机，那么其他服务器在相同索引位置不可能应用其他日志条目到状态机。在一台服务器应用一条日志条目到它自己的状态机时，它的日志必须和Leader一致，并且该日志条目是已提交的。现在我们假设在所有被应用到状态机的日志中term最小的日志条目为A，根据Leader完整性，所有在该term之后的Leader一定包含这个日志条目A，所以之后的term中应用某个索引位置的日志条目也会是相同的，这样就保证了状态机安全性。

#### Follower或者Candidate宕机

到目前为止我们只讨论了Leader宕机的问题。Follower和Candidate宕机处理起来要比Leader简单的多，并且它们都采用相同的方式处理。如果Follower或Candidate宕机，那么AppendEntries RPC请求和 RequestVote RPC请求都会失败，Leader会不断重试，如果宕机的服务器恢复了正常，那么RPC请求就会成功。如果服务器是在完成RPC请求后但是没有来得及响应Leader时宕机了，那么该服务器重启后就会再次收到相同的RPC请求，因为Raft的RPC请求时幂等的，所以很容易解决这个问题。比如服务器已经收到了一个AppendEntries RPC请求，当再次收到相同的AppendEntries RPC请求时会忽略。

#### 时序和可用性

我们的需求之一就是安全性不能依赖时序，不能因为消息比预期的早到或者晚点了一些就产生错误。然而可用性(系统可以及时响应客户端)必然会依赖时序。例如，在服务器宕机时，消息通讯的时间比超时时间还要长，Candidate选举不能等待太久，这就导致可能在选举超时时间内选举出Leader，从而Raft可能就无法运行。

Leader选举是Raft中对时序要求最为关键的地方，只要满足下面的时序要求，Raft就能选举出Leader并且保持稳定：

broadcastTime ≪ electionTimeout ≪ MTBF

broadcastTime是服务器并发发送RPC请求到集群中所有服务器并且收到它们的响应的平均时间；electionTimeout就是之前说的选举超时时间；MTBF是单台服务器发生故障的间隔时间。broadcastTime应该比electionTimeout小一个数量级，这样Leader就可以通过心跳消息阻止Follower进行选举。electionTimeout也应该比MTBF小一个数量级，这样可以保持系统的稳定性，当Leader宕机时，系统会在electionTimeout时间内不可用，并且通过将electionTimeout随机化，减少了选票平分的可能，我们希望这只占所有时间中的很小一部分。

broadcastTime和MTBF取决于服务器，只有electionTimeout是我们必须选择的。Raft的RPC请求需要接收方将信息持久化存储下来，根据存储技术不同，请求一般在0.5ms~20ms内完成。因此electionTimeout一般在10ms~500ms内。MTBF通常为几个月或者更多。所以很容易就可以满足时序要求。