https://www.zhihu.com/question/19787937

作者：朱一聪  
链接：https://www.zhihu.com/question/19787937/answer/82340987  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

之前的回答本来就觉得一些细节处并不严谨，现在回看=/=。我觉得严谨是一个讨论技术的必要条件，觉得现在也有能力写的严谨，于是想把回答改的尽量严谨，最后发现不如重写，顺便补充了我想补充的内容，结果就是更长=.=。Paxos是个精巧，又强大的协议，仅从过程的复杂度来说，确实如作者本人一再说的那样是个“简单的协议”，但是可以从非常多的角度来理解它为何正确，而原本的流程也并不适合直接工程化，这也是大概为什么工程上它存在如此多的变体。希望这个回答的让人更快的感受paxos的魅力，建立一个初步印象的同时不给人以误导。最后依然推荐larmport自己写的和paxos相关的三篇论文：<< The Part-Time Parliament>>、<<Paxos made simple>>、<<Fast Paxos>>前面关于Paxos的论述。  
2016/12/28

上周和一个有真正paxos工程经验的人讨论一下paxos，paxos现在大多是应用于replication的一致性，用来实现一个 多节点一致的日志，和他 的讨论让我觉得要想真正的精确掌握paxos和它对应的强一致性领域，也许只有真正的在工程中实现过才行。这个回答只能当做是想要了解原理的入门吧，甚至可能有些微妙的地方还会产生误导。它介绍了paxos面向的问题，以及为何它的流程要这么设计，但还是希望对有兴趣阅读这个问题的有所帮助。  
2016 10/30

现在看开头这段话是在是觉得有点羞耻，遂改之。我了解paxos是从找工作开始，比较详细的了解则是在毕设，自己动手了写了个类似Zookeeper的系统，paxos本身并不复杂，在<<paxos made simple>> Lamport用两段话就描述清楚了它的流程，他老人家也说paxos其实是个简单的算法。但是是我在工程领域见过最为精妙的算法。我想论述Paxos为什么难以理解会比描述Paxos的流程长的多的多。我最初学习Paxos是从《从Paxos到Zookeeper:分布式一致性原理与实践》，现在看来并不是个很好选择，作者讲解的方式是直接翻译论文，论述ZAB和paxos关系的部分我也觉得不是很严谨。如果真心觉得Paxos的原文不知所云，也不知道能拿来干嘛，可以从阅读Raft的论文开始，如果真的有兴趣，强推Raft作者Diego Ongaro那篇300页的博士论文《CONSENSUS: BRIDGING THEORY AND PRACTICE》，不只是讲解了Raft协议，而且系统的论述Paxos类似的一致性协议，不仅从原理，也从工程角度出发，涉及方方面面，在写毕设中了就是写不动就翻翻的良作。我个人觉得阅读任何**号称浅显易懂的解说Paxos算法的描述（比如下文=/=），最多只是让你更好的入门，要更深的理解Paxos以及所有等同于它的一致性协议，ZAB,Raft,ViewStamp，直接阅读相关论文，理解证明，理解它们哪里不同，为何本质上相同，与人探讨，在工程上自己实现，或者阅读开源实现的源代码才是最好的方式。分布式一致性是个有趣的领域，而Paxos和类似的协议对这个问题的重要性不喻，在过去的十年，Paxos几乎等价于分布式一致性。**  
2016 6/20

之前的答案最大的不严谨之处在于两个事件“先后”这种时序关系的处理上。paxos是个分布式一致性协议，它的事件需要多个节点共同参与，一个事件完成是指多个节点上均完成了自身负责的单机子事件(就让我门把这样的事件称为"分布式事件")，这样的分布式事件可以看作是多个单机子事件的复合，但**是即不能从两个分布式事件的先后推导出某个节点上它们的单机子事件的先后，也不能根据某个节点上两个单机子事件的先后断言它们对应的分布式事件的先后**。举一个简单的例子，两个节点P1，P2;分布式事件a1设置每节点的本地变量v=1,分布式式事件a2设置每个节点本地变量v=2，如果我们称a1先于a2完成，那么对于节点P1而言，v=1发生在v=2之前还是之后都有可能;反之如果P1上v=1发生在v=2之前，a1和a2哪个县完成也都有可能。  
  
原来的回答有些地方论述 分布式事件a1在a2之后(先)时，默认了单节点上，a1会比a2先达成状态，或者反之。  
  
实际上为了论证paxos的正确性，并不需要借助于分布式事件的时序(起码无用太在意物理时序），对于paxos流程中的分布式事件，例如提案被通过，值被决定，让我们忘记它们之间物理时间上的先后关系吧。

下面就开始假装推导出paxos，作为一种理解协议的流程和协议如何保证正确性的方式。这样的推导的过程远比我想象的冗长；**相比之下，论文中Lamport自己推导出Paxos的过程简洁、巧妙、漂亮，但是更抽象。在末尾用自己的语言描述了这种方式，作为补充**。补充的版本基本思路来自<<Paxos made simple>>，和原文略有不同；总共不到1500字，却既说明了Paxos是如何得到的，又严谨的论证了Paxos的正确性。

首先我们简单介绍paxos所保证的一致性的具体含义；达成一致的条件(何时达成一致）；基于的一个基本的数学原理；以及它需要满足的假设。  
  
什么是一致性?实际上这个词在不同地方语义并不那么一致，Paxos面向的是一个理论的一致问题，这个问题的通俗描述是：  
  
有一个变量v，分布在N个进程中，每个进程都尝试修改自身v的值，它们的企图可能各不相同，例如进程A尝试另v=a,进程B尝试另v=b，但最终所有的进程会对v就某个值达成一致,即上述例子中如果v=a是v达成一致时的值，那么B上，最终v也会为a。**需要注意的是某个时刻达成一致并不等价于该时刻所有进程的本地的v值都相同**，有一个原因是进程可能挂掉，你不能要求挂掉的进程任何事；更像是最终所有存活的进程本地v的值都会相同。  
  
这个一致性需要满足三个要求：  
  
1.v达成一致时的值是由某个进程提出的。这是为了防止像这样的作弊方式：无论如何，最终都令每个进程的v为同一个预先设置好的值，例如都令v=2，那么这样的一致也太容易了，也没有任何实际意义。  
2.一旦v就某个值达成了一致，那么v不能对另一个值再次达成一致。这个要求称为安全性。  
3.一致总是能够达成，即v总会被决定为某个值。这是因为不想无休止的等待，这个要求也称为活性。

Paxos中变量v达成一致的条件： **N个进程中大多数（超过一半） 进程都认为v是同一个值，**例如c，那么我们称**v被决定为c**。这样即使少数进程挂掉了，也不会使得一致无法达成。

Paxos保证的一致性如下：**不存在这样的情形，某个时刻v被决定为c，而另一个时刻v又决定为另一个值d。**由这个定义我们也看到，当v的值被决定后，Paxos保证了它就像是个单机的不可变变量，不再更改。也因此，对于一个客户端可以多次改写值的可读写变量在不同的节点上的一致性问题，Paxos并不能直接解决，它需要和状态机复制结合。

Paxos基于的数学原理： 我们称大多数进程组成的集合为法定集合，**两个法定集合必然存在非空交集，即至少有一个公共进程，称为法定集合性质**。 例如A,B,C,D,F进程组成的全集，法定集合Q1包括进程A,B,C，Q2包括进程B,C,D，那么Q1和Q2的交集必然不在空，C就是Q1，Q2的公共进程。如果要说Paxos最根本的原理是什么，那么就是这个简单性质。同时，可以注意到，这个性质和达成一致的定义相呼应。

Paxos中进程之间是平等的，即不存在一个特殊的进程，这是由于**如果协议依赖于某个特殊的进程，那么这个进程挂掉势必会影响协议;而对于分布式环境，无法保证单个进程必然必活，能够容忍一定数量的进程挂掉，是分布式协议的必然要求。**这是推导过程所要遵循的一个原则，就称为平等性原则好了。

消息是进程间通信的唯一手段，对于分布式环境来说，这是显然的。

Paxos要求满足的前置假设只有一个：消息内容不会被篡改；更正式的说是无拜占庭将军问题。

假装的推导总是先从一些具体的场景开始，既然Paxos的假设仅仅只是消息不会被篡改,保证了这点任意场景下都能保证一致性，那么对于举例的场景它必然是能够保证一致性的；因此不妨先使得协议流程能在当前场景下能保证一致性，然后再举出另一个场景，当前的协议流程无法再该场景下满足一致性，接着再丰富协议流程，满足该场景，如此往复，最终得到完整的paxos协议，最后再不失一般性的论证协议对任意场景都能保证一致性。

进程的平等性假设会带来如下的问题，考虑如下的场景1：三个进程的场景P1,P2P3（n个进程的场景类似），P1尝试令v的值被决定为a,P2尝试令v被决定为b。假设它们都先改写了自身的v值，然后发送消息尝试改修P3的v值。显然如果P3收到两个消息后都满足了它们的企图，那么v就会两次被决定为不同的值，这破坏了之前的定义。因此P3必须得拒绝掉其中一个进程的请求，**如何拒绝也是我们最先考虑的问题。**一个最简单的拒绝策略是先来后到，P3只会接受收到的第一个消息，拒绝之后的消息，即只会改写v一次。按照这个策略，如果P1发送的消息首先到达P3，那么P3接受该消息令v=a,拒绝掉后到的来自P2的消息。但是这个策略会引入一个另外的问题；在场景1的基础上考虑这样的场景1’，P3也尝试决定v的值，P3尝试令v被决定为c，那么P1，P2，P3都尝试修改v的值，首先P1令v=a,P2令v=b,P3令v=c(相当于自己给自己发消息），按照之前的策略，每个进程只会改写v的值一次，那么将永远不会出现两个进程的v值相等的情况，即v永远无法被决定。更正式的说，这样的协议不满足活性，活性要求协议总能达成一致。由此我们也得到第一个结论:**进程必须能够多次改写v的值**。同时我们也要意识到：**当进程收到第一个消息时，进程是没有任何理由拒绝这个消息的请求的。**

拒绝策略总是需要有一个依据，之前我们的依据是消息到达的先后，只接受第一个到达的消息，但这导致不满足活性。现在我们需要另一个拒绝策略，也就是需要另一个依据，这个依据至少能够区分两个消息。**为此我们引入一个ID来描述这个消息，这样就可以根据ID的大小来作为拒绝或接受的依据；**选择ID更大的消息接受和选择ID更小的消息接受是两种完全对称的策略，不妨选择前者。这个策略不会导致明显的活性问题，ID更大的消息总是能被接受，一个节点可以多次更改v的值。例如在场景1'中，只要P1的消息ID比P3发送给自己的消息ID更大，P3就会接受P1的消息，令v=a，从而令v的值被决定为a。再来考虑最初的场景1，不妨假设P1的消息ID大于P2的消息ID，根据P3收到消息的先后可以分为两种情况：  
  
1. P3先收到P1的消息，记做场景1-2。由于P1的消息是P3收到的第一个消息，P3接受该请求，令v=a；同时为了能对之后收到的消息作出是否接受的判断，P3需要记录该消息的ID作为判断的依据。之后P3又收到P2的消息，该消息的ID小于P3记录的ID(即P1的消息ID)，因此P3拒绝该消息，这样我们的目的就达到。  
  
2. P3先收到P2的消息，记作场景1-3。同样P3接受该消息，令v=b,记录该消息的ID。之后P3收到P1的消息，由于P1的消息ID大于P3记录的ID，因此P3无法拒绝该消息，之前的问题依旧存在。

尽管对于场景1-3，目前的策略依旧无法保证一致性，但是起码我们缩小协议不适用的范围。先总结下我们目前的策略，并定义一些称谓以方便后面的论述。**我们称呼进程P发送的尝试修改另一个进程中变量v的值的消息称之为提案，记作Proposal；提案的ID记作proposal\_id；提案中会附带一个值，如果一个进程接受一个提案，则修改自身的v值为该提案中的值。如果一个提案被大多数进程所接受，那么称提案被通过，此时显然v被决定为提案中的值。进程P记录的接受的提案ID记做a\_proposal\_id。**

之前我们尚未清晰定义a\_proposal\_id，实际上我们也就并未清晰的定义我们的拒绝策略，当P收到一个提案Proposal-i时，可能已经收到过多个提案，Proposal-i.proposal\_id该和其中哪个提案的proposal\_id比较，我们并未定义。我们定义为其中的最大者，这样实际上进程P只需维护一个a\_proposal\_id即可，当收到一个Proposal时，更新a\_proposal\_id = Max(Proposal.proposal\_id，a\_proposal\_id）。同时在之前的描述中我们应当注意到实际上一个进程存在**两个功能**：  
  
1. 进程主动尝试令v的值被决定为某个值，向进程集合广播提案。  
  
2. 进程被动收到来自其它进程的提案，判断是否要接受它。  
  
因此可以把一个**进程分为两个角色，称负责功能1的角色是提议者，记作Proposer，负责功能2的角色是接受者，记作Acceptor。**由于两者完全没有耦合，所以**并不一定需要在同个进程**，但是为了方面描述，我们假定一个进程同时担任两种角色，而实际的工程实现也往往如此。

接着我们尝试解决场景1-3,这看起来很难。P3作为接受者，收到P2的提案之前未收到任何消息，只能接受该提案，而由于P1的提案proposal\_id大于P2的提案，我们的拒绝策略也无法让P3拒绝P2。我们先不急着推导具体可行的策略，先考虑下解决1-3场景可能的角度，有如下三种角度可以入手：  
  
1. P3能够拒绝掉P2的提案。  
  
2. P3能够拒绝掉P1的提案。  
  
3. 限制P1提出的提案中的值，如果P1的提案中的值与P2的提案一致，那么接受P1也不会破坏一致性。  
  
接着我们分析三个角度的可行性：  
  
角度1需要P3有做出拒绝的依据，由于消息是进程间通信唯一手段，这要求P3在收到P2的提案之前必须先收到过其它消息。对于场景1-3，只有P1，P2是主动发送消息的进程，P2当然不可能额外还发送一个消息试图令P3拒绝自己随后的提案。那么唯一的可能是P1在正式发送提案前，还发送了一个消息给P3，这个消息先于P2的提案到达，给了P3拒绝P2提案的理由。如果沿用目前的拒绝策略，那么P1只需先发送随后提案的proposal\_id给P3，P3更新a\_proposal\_id 为 该消息附带的proposal\_id，这样a\_proposal\_id将大于P2的提案的proposal\_id，而导致P2的提案被拒绝，似乎这是一个可行的角度。  
  
对于角度2，我们目前的策略无法做到这一点，因此除了proposal\_id外，我们还得给提案附加上额外的信息作为另外的拒绝策略的依据。提案由进程提出，也许我们可以附加进程的信息，但是就算P3得知P1的提案由P1提出，P3又凭什么歧视P1，这违反进程的平等性假设。似乎这个角度不是一个好角度。  
  
最后我们分析一下角度3，角度3提供了与1,2截然不同的思路，它不再是考虑如何拒绝，而把注意力放在**如何对提案的值做出恰当的限制上**。对于场景1-3而言，从这个角度，由于P3无法拒绝P1和P2的提案中的任何一个，因此**P1的提案的值就必须与P2的提案一致**；这也意味着了P1在正式提出提案前，需要有途径能获悉P2的提案的值。如我们上面一直强调的，消息是唯一的通信手段，P1必须收到来自其它节点消息才有可能得知P2提出的提案的值。P1可以被动的等待收到消息，也可以主动的去询问其它节点等待回复。后者显然是更好的策略，没有收到想要的消息就一直等待未免也太消极了，这种等待也可能一直持续下去从而导致活性问题。

经过上面的分析，我们暂时排除了从角度2入手（实际上后面也不会再考虑，因为从1,3入手已经足以解决问题）。下面将沿着角度1,3进行更深入的分析，我们先尝试从角度1出发，毕竟考虑如何拒绝已经有了经验。先来总结下我们在分析角度1时引入的额外的流程:  
  
进程P在发送提案前，先广播一轮消息，消息附带着接下来要发送的提案的proposal\_id。由于该消息和接下来发送的提案相关，且在提案被提出之前发送，不妨称这个消息为**预提案，记作PreProposal，预提案中附带着接下来的提案的proposal\_id**。当作为接受者的进程Pj收到预提案后，更新Pj. a\_proposal\_id。还记得我们之前的拒绝策略中a\_proposal\_id的更新策略嘛：a\_proposal\_id = max(proposal\_id,a\_proposal\_id)，a\_proposal\_id是递增的。因此如果预提案的proposal\_id小于Pj.a\_proposal\_id，Pj完全可以忽略这个预提案，因为这代表了该预提案对应的提案的proposal\_id小于Pj.a\_proposal\_id，必然会被拒绝。我们沿用之前拒绝策略中a\_proposal\_id的更新策略。这样当收到预提案或者提案后，a\_proposal\_id的值均更新为 max(a\_proposal\_id,proposal\_id)。  
  
接着我们来看看引入了预提案后，能否真正解决场景1-3。根据P1和P2的预提案到达P3的先后也存在两种场景：  
  
1.场景1-3-1：P1的预提案先到达，P3更新a\_proposal\_id 为该提案的proposal\_id，这导致随后到达的P2的提案的proposal\_id小于a\_proposal\_id，被拒绝。满足一致性  
  
2.场景1-3-2：P2的提案先到达，P3接受P2的提案，此时和原始的场景1-3存在同样的问题。归根结底，预提案阶段**能否使得P3拒绝该拒绝的，也依赖消息到达的顺序**，和提案阶段的拒绝策略存在相同的问题，但至少又缩小了不能保证安全性的场景范围。  
  
幸好我们还有角度3可以着手考虑，所以仍有希望完全解决场景1-3。在深入角度3之前，先总结下协议目前为止的流程，现在协议的流程已经分为了两个阶段：**预提案阶段和提案阶段**，两种消息：***预提案*** 和***提*案**，两种角色：***接受者*** 和 ***提议者***，流程如下：

***阶段一*** **提议者Proposer：向接受者Acceptor广播预提案，附带接下来提案Proposal的proposal\_id** **接受者Acceptor：收到预提案后更新a\_proposal\_id = max(proposal\_id,a\_proposal\_id)**

***阶段二* 提议者Proposer：向接受者Acceptor广播提案，和之前的预提案共享同一个proposal\_id** **接受者Acceptor：如果收到的提案的proposal\_id>= a.proposal\_id，那么接受这个提案，更新a\_proposal\_id = max(proposal\_id,a\_proposal\_id)**

为了更形象，之前的讨论是基于三个进程的场景,实际上对于N进程的场景也是一样的。N个进程时，与之前场景1对应的场景是：

N个进程，存在两个进程Pi,Pj，Pi尝试另v被决定为a,Pj尝试另v被决定为b，Pi提出的预提案记作PreProposal-i，提案记作Proposal-i；Pj的预提案PreProposal-j，提案Proposal-j。

之前拒绝策略的讨论都是基于一个关键的进程P3，只要P3最终能拒绝Proposal-i和Proposal-j中的一个，两个提案就不会都被通过，那么一致性就不会被破坏。Pi的提案被通过代表了存在一个法定集合Q-i，Q-i中的进程都接受了Proposal-i，Pj同理，存在一个Q-j，Q-j中的进程都接受了Proposal-j。**由于法定集合的性质，两个多数集Q-i和Q-j中必然存在一个公共进程Pk。Pk即相当于场景1中的P3，只要Pk能够拒绝Proposal-i和Proposal-j中的一个，协议依旧是安全的。**为了不失一般性，下**面我们都以N个进程的场景作为讨论的基础，**称为场景2，由于**场景1和场景2可以一一对应**，所以接下来顺着限制提案的值的角度，**我们直接针对场景2-3-2，之前的场景和场景1一样，我们的拒绝策略已经足以应付。**v的值被决定代表有一个提案，它被法定数目的集合接受，我们称这为提案被通过。

首先我们看下场景2-3-2，Pi对应场景1-3-2中的P1，Pj对应P2，Pk对应P3。Pj的提案Proposal-j最终会被法定集合Q-j接受，即v的值被决定为b，且Proposal-i.proposal-id > Proposal-j.proposal\_id。我们需要限制Pi的提案的值，不能让Pi自由的给Proposal-i中的v赋值。在2-3-2中，**由于拒绝策略失效，所以只能令Proposal-i.v = Proposal-j.v=b**。要做到这一点，正如前面的分析所言，Pi需要先主动询问进程集合，来得知Proposal-j.v =b这一事实。显然Pi是没有先验信息来得知Proposal-j由哪个进程提出，也不知道Q-i和Q-j的公共节点Pk是谁，因此Pi只能广播它的查询。由于我们需要允许少数进程失败，Pi可能只能得到大多数进程的回复，而这之中可能不包括Pj。我们称这些回复Pi的查询的进程的集合为Q-i-2,为了描述更简单，无妨假设Q-i-2=Q-i。尽管Pi的查询可能得不到Pj的回复，好在作为将会被通过的提案，Proposal-j将会被Q-j内所有进程接受，因此如果进程作为接受者在接受提案时，顺便记录该提案，那么Q-j内所有进程都将得知Proposal-j.v=b。**由于Pk属于Q-i和Q-j的交集，所以Pk即收到了Pi的查询，又接受了提案Proposal-j。**之前我们已经引入了预提案阶段，显然我们可以为预提案附带上查询的意图，即Pk作为接受者收到Pi的预提案后，会回复它记录的接受过的提案。有一个问题是Pk此时是否已经记录了Proposal-j呢?很巧的是**在场景2-3-2中，Pj的提案Proposal-j是先于Pi的预提案PreProposal-i先到达**，所以Pk已经记录了Proposal-j.v = b，Pj收到的来自Pk的回复中包含了提案Proposal-j，而2-3-2之外的场景，拒绝策略已经足以应付。这里依旧还有两个问题，先讨论第一个：  
  
实际上除了Pi和Pj外可能还会有多个进程发起预提案和提案，所以收到 PreProposal-i时Pk可能已经接受过多个提案，并非只有Proposal-j，那么Pk应该回复PreProposal-i其中哪个提案，或者都回复？Pk并不知道Proposal-j会被通过，它只知道自己接受了该提案。**都回复是个效率很低但是稳妥，可以保证Pk不会遗漏Proposal-j**，Pk已经回复了它所能知道的全部，我们也无法要求更多。需要注意到的是进程是平等的，所以Q-i中所有进程都和Pk一样回复了它接受过的所有提案。当Pi收到所有来自Q-i的回复时，随之而来的是第二个问题：  
  
Pi收到了多个Proposal作为一个Acceptor组成的法定集合Q-i对PreProposal-i的回复，记这些Proposal组成的集合记坐K-i，那么它应当**选择K-i中哪个一个提案的值作为它接下来的提案Proposal-i的v值**？记最终选择的这个提案为Proposal-m。

在场景2-3-2中，我们第一直觉是希望选择的Proposal-m 即是 Proposal-j，但是实际上，我们只要**保证Proposal-m .v = Proposal-j.v即可**。从另一个角度 ，K-i中很可能存在这样的**提案Proposal-f,Proposal-f.v!=Proposal-j.v**，我们要做的是**避免**选择到这类提案。我们可以根据一些依据瞎选碰碰运气，但是这并明智。我们不妨假设存在一个策略cf，cf满足需求，使得选择出提案Proposal-m满足Proposal-m.v= Proposal-j.v。然后让我们来分析一下此时Proposal-f有什么特征。

**Proposal-f能够被提出，代表存在一个多数集合Q-f，Q-f中每个进程都接受了PreProposal-f**，同时假设是进程P-f提出了PreProposal-f和Proposal-f。Q-f和Q-j必然存在一个公共节点，记做Ps，Ps即接受了PreProposal-f又接受了Proposal-j。Ps收到PreProposal-f和Proposal-j的顺序只有两种可能：  
  
1.Ps先收到PreProposal-f  
  
2.Ps先收到Proposal-j  
  
PreProposal-f.proposa-id和Proposal-j. proposal\_id的大小也只有两种可能，**不妨先假设PreProposal-f.proposal\_id > Proposal-j.proposal\_id**。

对于情形1，Ps先收到PreProposal-f，接受它，更新Ps.a\_proposal\_id = PreProposal-f.proposal\_id > Proposal-j.proposal\_id，同时之前的a\_proposal\_id的更新策略又使得Ps.a\_proposal\_id是递增的，于是导致收到Proposal-j时,Proposal-j.proposal\_id小于Ps.a\_proposal\_id,被拒绝，而这于Ps的定义矛盾。  
  
对于情形2，Ps将把提案Proposal-j回复给PreProposal-f。**由于我们假设了策略cl的存在，于是P-f在收到所有Q-f对PreProposal-f的回复后，将令Proposal-f.v=Proposal-j.v**，cl就是干这个的。因此由于Proposal-f.v!=Proposal-j.v矛盾。  
  
于是当假设PreProposal-f.proposal\_id > Proposal-j.proposal\_id 时，情形1,2我们都得出了矛盾，同时两者的proposal\_id又不相等(最初的假设），所以**必然PreProposal-f.proposal\_id < Proposal-j.proposal\_id，即Propsoal-f.proposal\_id < Proposal-j.proposal\_id**。  
  
于是我们得到的结论是：如果策略cl存在，提案Proposal-j最终会被通过，任意一个proposal\_id更大的预提案PreProposal-i，对于它得到的Q-i的回复K-i中的Proposal-f，只要Proposal-f.v!= Proposal-j.v，那么必然 Proposal-f.proposal\_id < Proposal-j.proposal\_id。  
  
既然K-i中所有v值不等于Proposal-j.v的提案，proposal\_id都比Proposal-j更小，**那代表所有proposal\_id比Proposal-j更大的提案，v值都等于Proposal-j.v**，**因此选择K-i中proprosal\_id最大的提案，就能保证Proposal-i.v = Proposal-j.v**。于是我们得到了策略cl的具体形式。  
  
我们得到了具体可行的策略cl是**建立在策略cl存在这一前提之上**，因此反过来，对于这个具体的选值策略cl，结合之前我们得到了协议流程，它是否能保证如下的性质**CP1**，依旧需要进一步的论证 ：  
**如果一个提案Proposal-j最终会被通过，那么对于任意的一个提案Proposal-i,如果Proposal-i.proposal\_id > Proposal-j.proposal\_id，那么Proposal-i.v = Proposal-j.v。**

我们先总结下目前得到的协议流程：

***阶段一 预提案阶段*** **提议者Proposer：向接受者Acceptor广播预提案，附带接下来提案Proposal的proposal\_id**

**接受者Acceptor：收到预提案后更新a\_proposal\_id = max(proposal\_id,a\_proposal\_id)，如果预提案的proposal\_id大于a\_proposal\_id，那么回复该预提案的提议者改接受者接受过的所有提案。**

***阶段二 提案阶段*** **提议者Proposer：等待直到收到大多数接受者对预提案的回复，从所有回复的提案组合成的集合K中挑选proposal\_id最大的提案，以该提案的值作为本次提案的值。如果K是空集，那么可以给提案任意赋值。** **向接受者Acceptor广播提案，和之前的预提案共享同一个proposal\_id** **接受者Acceptor：如果收到的提案的proposal\_id>= a.proposal\_id，那么接受这个提案，更新a\_proposal\_id = max(proposal\_id,a\_proposal\_id)**

这些流程是为了解决举例的场景而不断丰富的，接着就让我们论证下协议流程是否总是可以确保CP1。

**首先假设Proposal-i.v != Proposal-j.v，如果得出矛盾即可证明CP1。**在尝试推出矛盾前，我们先做一些定义，以便后续的推导。  
  
记大多数接受者组成的法定集合为Q，**K是提议者在提案阶段收到的所有Q回复的提案组成的集合**，如果K不为空**，记K中proposal\_id最大的提案是MaxProposal(K)，本次提案的值即是MaxProposal(K).v**；如果K是空集，那么MaxProposal(K).v = null。特别的，对于提案Proposal-i,回复它预提案接受者的集合为Q-i,回复的提案组成的集合为K-i，**Proposal-i.v = MaxProposal(K-i)**，Proposal-i.v=null代表可以随意赋值。为了描述方便，我们令Proposal-i的proposal\_id为i，即Proposal-i代表了proposal\_id=i的提案，Proposal-j意味着Proposal-j.proposal\_id =j。  
  
论证过程如下：  
  
(1) Proposal-i.v!=Proposal-j.v，即MaxProposal(K-i) .v！= Proposal-j.v，即MaxProposal(K-i)!=Proposal-j  
  
(2) Proposal-j最终会被通过，代表最终会存在一个多数集合Q-j，Q-j中每个接受者都接受了Proposal-j。  
  
(3) 两个多数集必然存在公共成员，故Q-j和Q-i必然存在一个公共的进程Pk，Pk即收到了PreProposal-i又收到了Proposal-j，且都接受了它们；Pk收到消息的先后关系只存在如下两种可能：  
  
1.Pk先收到了PreProposal-i  
  
2.Pk先收到了Proposal-j  
  
(4) 情形1中Pk先收到了PreProposal-i，那么Pk收到Proposal-j时，Pk.a\_proposal >= PreProposal-i.proposal\_id ＞Proposal-j.proposal\_id，Pk会拒绝Proposal-j，与(3)矛盾，因此情况1不可能，**Pk必然是先收到Proposal-j**。  
  
(5) 情形2中Pk收到PreProposal-i时，已经接受了Proposal-j，因此Pk回复PreProposal-i的提案中包含了Proposal-j，因此K-i中必然包含了Proposal-j。  
  
(6) 由(1)已知MaxProposal(K-i) != Proposal-j，即**存在另一个提案Proposal-m** = MaxProposal(K-i),而Proposal-j属于K-i,因此**Proposal-m.proposal\_id > Proposal-j.proposal\_id**，且**Proposal-m.v != Proposal-j.v**。  
  
(7)由预提案阶段，接受者回复预提案的条件可知：Proposal-i.proposal\_id大于集合K-i中任意一个提案的Proposal-id，**故Proposal-i.proposal\_id>Proposal-m.proposal\_id。**  
  
(8) 目前我们已经论证如下一点：  
  
在Proposal-j最终会被通过的前提下，**如果存在一个提案Proposal-i.v!=Proposal-j.v，且Proposal-i.proposal\_id >Proposal-j.proposal\_id,**我们一个数学符号来带表示这个情况，记**CF(j,i)；那么 必然存在**一个提案**Proposal-m, Proposal-m！=Proposal-j.v,且Proposal-m.proposal\_id > Proposal-j.proposal\_id**，同样的我们可以记做***CF(j,m)***。并且Proposal-m.proposal\_id < Proposal-i.proposal\_id，**m < i**。  
  
即如果CF(i,j)成立，那么必然CF(m,j)成立，且i>m，即 **CF(i,j) —> CF(m,j)**。这个过程可以继续往下递归，但由于区间[j,i]范围是有限的，因此一定会递归到一个CF(j,e)，此时不可能存在一个提案，它的proposal\_id在区间(j,e)内，无法往下递归，这与(8)矛盾。这也就意味着CF(e,j)不成立，而如果CF(i,j)成立，那么CF(e,j)成立,因此CF(i,j)不成立，故假设不成立，即Proposal-i.v 必然等于Proposal-j.v，即证CP1。

通过归约的方式得出矛盾的方式依旧有些抽象，我们可以通过更好的定义假设来更容易得到的矛盾：  
  
我们加强对Proposal-i的约束;先假设存在一个提案的非空集合KD，KD中的任意一个提案Proposal-k,Proposal-k.v!=Proposal-j.v，且Proposal-k.proposal\_id > Proposal-j.v;再假设**Proposal-i是KD中proposal\_id最小的提案**；由于KD是非空集合，故Proposal-i必然存在。  
  
我们依旧可以从Proposal-i出发，(1)~(7)与上面相同，同理得到：存在一个提案Proposal-m, Proposal-m！=Proposal-v,且Proposal-m.proposal\_id > Proposal-j.proposal\_id，且Proposal-m.proposal\_id < Proposal-i.proposal\_id。  
  
显然Proposal-m满足集合KD对提案的要求，故Proposal-m属于KD，又Proposal-m.proposal\_id<Proposal-i.proposal\_id，这和Proposal-i是KD中proposal\_id最小的提案的定义矛盾。因此不存在这样的非空集合KD，即不存在一个提案Proposal-k,Proposal-k.v!=Proposal-j.v且Proposal-k.proposal\_id>Proposal-j.proposal\_id，即如果一个提案Proposal-j最终会被通过，对于任意的一个提案Proposal-i,如果Proposal-i.proposal\_id > Proposal-j.proposal\_id，那么必定Proposal-i.v = Proposal-j.v，即CP1。

CP1约束了proposal\_id大于Proposal-j的提案的值，保证了如果一个提案Proposal-j最终会被通过，不会存在另一个proposal-id更大且值不同的提案被通过，因为这些提案的值都和Proposal-j相同。那么对于proposal\_id更小的提案呢？ 我们假设存在一个提案Proposal-o,Proposal-o.proposal\_id < Proposal-j.proposal\_id，且Proposal-o.v!=Proposal-j.v,Proposal-o最终会被通过，将CP1应用于Proposal-o,则可知Proposal-j不存在，这矛盾，故Proposal-o不存在。故由CP1我们可知：如果一个提案Proposal-j最终会被通过，那么不存在另一个提案，它最终会被通过，且它的值与Proposal-j不同。由此协议必然是安全的。

虽然我们得到了一个安全的一致性协议，基本上它就是Paxos，但是真正的Paxos要比我们假装推导出的协议更简单一点。  
  
回过头来看下我们的阶段1中接受者Acceptor的行为，它要回复所有的它接受过的提案，从实践的角度，不论是在本地保存所有它接受过的提案还是通过网络将它们传输给提议者，开销都太大且不可控。再看下阶段二中，提议者的选值策略，它只是选择了收到的多数集接受者回复的提案中proposal\_id最大的那一个，因此**接受者实际上只需要回复它接受过的proposal\_id最大的提案即可，因为其它提案根本不可能会被选值策略选中**。因此最终的协议如下，它就是Paxos:

***阶段一 预提案阶段*：** **提议者Proposer：向接受者Acceptor广播预提案，附带接下来提案Proposal的proposal\_id** **接受者Acceptor：收到预提案后更新a\_proposal\_id = max(proposal\_id,a\_proposal\_id)，如果预提案的proposal\_id>a\_proposal\_id，Acceptor回复记录的接受过的proposal\_id最大的提案。**

***阶段二 提案阶段*：** **提议者Proposer：等待直到收到大多数接受者对预提案的回复，从所有回复的提案组成的法定数目的提案集合K中挑选proposal\_id最大的提案，以该提案的值作为本次提案的值。如果K是空集，那么可以给提案任意赋值。然后把该提案广播给接受者们，提案和预提案共享同一个proposal\_id。** **接受者Acceptor：如果收到的提案的proposal\_id>= a.proposal\_id，那么接受这个提案，更新a\_proposal\_id = max(proposal\_id,a\_proposal\_id)，更新记录的提案。**

**补充部分**：  
  
上面的过程从具体的场景开始推导Paxos，虽然直观但是繁琐，如果从抽象的概念和分析入手，那么过程将会相当简洁和漂亮，这也是Lamport的原始论文中的方式。这种方式理解起来更困难的地方在于：  
  
1.没有任何具体的认知下，直接抽象的讨论容易让人摸不着头脑。  
  
2.大神总是在一些地方觉得显然而不加以展开论述，而普通读者如我的内心OS:显然你mei!  
  
但是原文引出Paxos算法的过程实在是简洁、漂亮；而经过上面的轮述，相信有了直观的印象后，再来看抽象的方式也不那么困难，所以补充下。

回顾下定理**CP1**：  
  
如果一个提案Proposal-j最终会被通过，那么对于任意的一个提案Proposal-i,如果Proposal-i.proposal\_id > Proposal-j.proposal\_id，那么必定Proposal-i.v = Proposal-j.v。

上面我们已经论证了只要协议能够保证CP1就能够保证一致性。但是CP1依旧过于宽泛，从CP1引出具体的协议流程依然是一头雾水，那么我们是否可以得到一个更加具体的定理CP2，保证CP2即可保证CP1，同时从CP2出发更容易引出协议的具体流程。为了描述方便，我们令Proposal-i的proposal\_id为i，即Proposal-i代表了proposal\_id=i的提案。

要导出CP2不妨先考虑下如何证明CP1，利用归纳法，只要如能证明如下性质成立，即可证明CP1：  
**如果proposal\_id在区间[j,i）内任意的提案，提案的值均为Proposal-j.v，那么必定Proposal-i.v=v;这个定理记做CP1\_2。**  
  
现在我们用高中时简单而效果神奇的归纳法，利用CP1\_2证明下CP1:

假设propsal\_id小于i的提案中最大的提案是Proposal-(i-1)。

1.如果对于[j,i-1)内的任意提案，值均为Proposal-j.v，那么由CP1\_2可知Proposal-i.v = Proposal-j.v。

2.由1可知如果对于[j,i-1)内的任意提案，值均为Proposal-j.v，[j,i)内的任意提案，值均为Proposal-j.v

3.假设Proposal-(j+1)是proposal-id大于j的最小提案，由CP1\_2可知Proposal-(j+1).v = Proposal-j.v

4.由3,2归纳可知[j, )内任意提案Proposal-i，Proposal-i.v = Proposal-j.v，即CP1

来看下CP1\_2，相比CP1，它结论不变，但是多了一个前置条件：**proposal\_id在区间[j,i）内任意的提案值均为Proposal-j.v；这是一个重大的进步。**CP1\_2相比CP1看起来容易保证 很多，但是它们却是等价的。考虑CP1\_2的三个前置条件：

1.i > j  
2.提案Proposal-j最终会被通过。因此由提案被通过的定义可知必然存在一个法定集合Q-j，Q-j中任意一个接受者最终都接受了Proposal-j  
3.proposal\_id在区间[j,i)内的提案的值均为Proposal-j.v  
  
对于任意的一个法定集合Q，**考虑Q最终（包括过去和未来的所有时空）会接受的所有proposal\_id小于i的提案组成的集合K**。根据法定集合性质，Q和Q-j必然存在一个公共的节点，即Q中必然存在一个节点，该节点最终会接受Proposal-j，因此**集合K包含Proposal-j。**

由K包含Proposal-j可知K中最大的提案proposal\_id >= j；由CP1\_2的前置条件3和K的定义可知如果K中存在proposal-id大于j的提案，那么该提案的值等于Proposal-j.v，因此K中proposal\_id最大的提案的值等于Proposal-j.v。  
  
综上所述由CP1\_2的前置条件可知：**对于任意的一个法定集合Q，Q最终会接受的proposal\_id小于i的提案组成的集合K，K中proposal\_id最大的提案的值必定为Proposal-j.v**。如果我们能证明该条件下，Proposal-i.v = Proposal-j.v，即可证明CP1\_2。将CP1\_2的前置条件替换为该条件，我们可以得到一个如下的性质**CP2**，保证CP2即可保证CP1\_2：  
**对于任意的一个法定集合Q，Q最终会接受的所有proposal\_id小于i的提案组成的集合K，如果K中proposal\_id最大的提案的值为Proposal-j.v;那么Proposal-i.v = Proposal-j.v。**

而引出满足CP2的流程就比较容易了，由于集合K中proposal\_id最大的提案的值等于Proposal-j.v，看起来只要令Proposal-i的值为K中proposal-id最大提案的值就可以保证CP2。由于Q是任意一个法定集合，因此获取K似乎在实现上也不难，提出Proposal-i的提议者只要向Q中所有接受者询问即可。

**然后： CP2 —> CP1\_2—> CP1 —>一致性**

但是实际上获取K没有那么简单，K包含的是Q所有最终接受的proposal-id小于i的的提案，不仅包含**已经接受过**的提案，还包括**未来会接受**的提案。获取**已经接受过的提案是容易的**，Q中的接受者只需记录它所有接受过的提案，当收到提出Proposal-i的提议者询问时，回复当中proposal\_id小于i的提案即可;**但是如何知晓未来？**我们可以换个思路，**既然无法知晓未来，那么我们约束未来**，收到询问后，**令Q中的接受者都承诺不再接受任何proposal\_id小于i的提案**，即接受者未来将不接受任何proposal\_id小于i的提案;既然未来已不存在，那么Proposal-i的提议者根据Q的回复获能得到**完整的K。**

于是协议的流程如下：  
  
对于提议者，在正式提案前，先向任意的法定集合Q发送一个消息，这个消息即是预提案，消息中要附带提案的proposal-id，作为接受者**承诺**和**回复**的依据。  
  
接受者收到预提案后，**承诺：**不再接受比预提案中附带的proposal-id更小的提案；并**回复:**已经接受的proposal-id比于提案的proposal-id更小的提案，如之前所论述的，回复的所有满足条件的提案可以优化为只回复一个比预提案proposal\_id更小的提案中proposal\_id最大的那个提案。

提议者收到所有Q中接受者回复的提案后，挑选其中proposal\_id最大的提案的值作为本次提案的值。

这样我们就得到了Paxos中最为关键的几步，阅读了之前冗长的假装推导，相信读者很容易就能补全它得到完整的Paxos。

相比于之前近万字的假装推导，这个推导过程才1500字左右，但是即说清了Paxos是如何得出的，又论证Paxos为何正确，简洁却更有力。所以最后还是建议真有兴趣的话去看下原文，在我看来它无疑是计算机领域那数不尽的论文中最值得阅读的那一类。末尾我所描述的版本思路来自<<Paxos made simple>>，基本一致但也并不完全相同；而<< The Part-Time Parliament>>则别有一番风味。

最后需要注意的是Paxos并不完全满足开头解决一致性问题需要满足的三个条件中的3。理论上，Paxos存在永远无法达成一致的可能，哪怕是在所有进程都存活的情况下。想象一下这样的场景，一个提案Proposal-j被提出时，恰好一个proposal-id更大的预提案Proposal-i被提出，导致Proposal-j无法被通过，而Proposal-i同样的 又因为一个proposal\_id更大的其它预提案被提出，导致无法被通过。这种情况理论上存在无限递归的可能，这个问题也称为活锁;FLP早就证明了就算是容忍一个进程的失败，异步环境下任何一致性算法都存在永不终止的可能。但是实际的工程中，很多手段可以来减小两个提案的冲突概率，使得v被决定的均摊开销是一个提案，多个提案还无法决定v值的情形是极小概率事件，且概率随着提案个数增加越来越小。另外的一点，通常认为Paxos可以容忍少数进程挂掉 ，但这只是为了保证它的活性，对于安全性，实际上Paxos永远满足1,2，哪怕进程都挂掉了，此时只是显然一致无法达成而已。