分布式系列文章——Paxos算法原理与推导

Paxos算法在分布式领域具有非常重要的地位。但是Paxos算法有两个比较明显的缺点：1.难以理解 2.工程实现更难。

网上有很多讲解Paxos算法的文章，但是质量参差不齐。看了很多关于Paxos的资料后发现，学习Paxos最好的资料是论文《Paxos Made Simple》，其次是中、英文版维基百科对Paxos的介绍。本文试图带大家一步步揭开Paxos神秘的面纱。

Paxos是什么

> Paxos算法是基于**消息传递**且具有**高度容错特性**的**一致性算法**，是目前公认的解决**分布式一致性**问题**最有效**的算法之一。

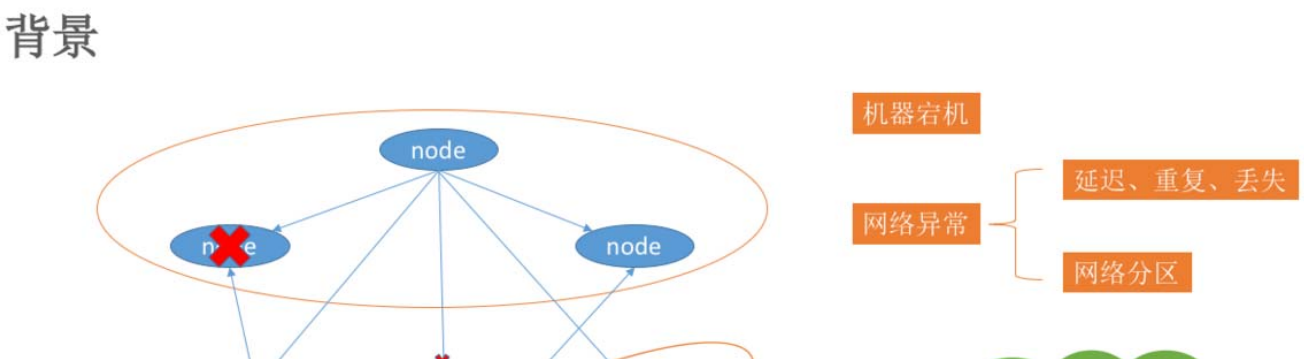
Google Chubby的作者Mike Burrows说过这个世界上**只有一种**一致性算法，那就是Paxos，其它的算法都是**残次品。**

虽然Mike Burrows说得有点夸张，但是至少说明了Paxos算法的地位。然而，Paxos算法也因为晦涩难懂而臭名昭著。本文的目的就是带领大家深入浅出理解Paxos算法，不仅理解它的执行流程，还要理解算法的推导过程，作者是怎么一步步想到最终的方案的。只有理解了推导过程，才能深刻掌握该算法的精髓。而且理解推导过程对于我们的思维也是非常有帮助的，可能会给我们带来一些解决问题的思路，对我们有所启发。

问题产生的背景

在常见的分布式系统中，总会发生诸如**机器宕机**或**网络异常**（包括消息的延迟、丢失、重复、乱序，还有网络分区）等情况。Paxos算法需要解决的问题就是如何在一个可能发生上述异常的分布式系统中，快速且正确地在集群内部对**某个数据的值**达成**一致**，并且保证不论发生以上任何异常，都不会破坏整个系统的一致性。

注：这里**某个数据的值**并不只是狭义上的某个数，它可以是一条日志，也可以是一条命令（command）。。。根据应用场景不同，**某个数据的值**有不同的含义。



在Paxos算法中，有三种角色：

- **Proposer**
- **Acceptor**
- **Learners**

在具体的实现中，一个进程可能**同时充当多种角色**。比如一个进程可能**既是Proposer又是Acceptor又是Learner**。

还有一个很重要的概念叫**提案（Proposal）**。最终要达成一致的value就在提案里。

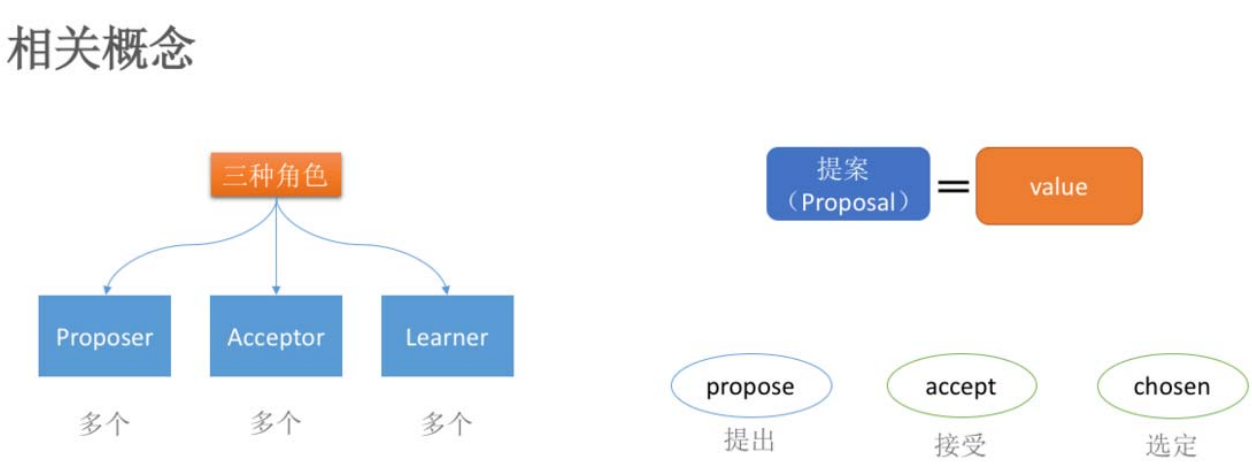
注：

- **暂且**认为『**提案=value**』，即提案只包含value。在我们接下来的推导过程中会发现如果提案只包含value，会有问题，于是我们再对提案**重新设计**。
- **暂且**认为『**Proposer可以直接提出提案**』。在我们接下来的推导过程中会发现如果Proposer直接提出提案会有问题，需要增加一个学习提案的过程。

Proposer可以提出（propose）提案；Acceptor可以接受（accept）提案；如果某个提案被选定（chosen），那么该提案里的value就被选定了。

回到刚刚说的『对某个数据的值达成一致』，指的是Proposer、Acceptor、Learner都认为同一个value被选定（chosen）。那么，Proposer、Acceptor、Learner分别在什么情况下才能认为某个value被选定呢？

- Proposer：只要Proposer发的提案被Acceptor接受（刚开始先认为只需要一个Acceptor接受即可，在推导过程中会发现需要半数以上的Acceptor同意才行），Proposer就认为该提案里的value被选定了。
- Acceptor：只要Acceptor接受了某个提案，Acceptor就任务该提案里的value被选定了。
- Learner：Acceptor告诉Learner哪个value被选定，Learner就认为那个value被选定。



- 只有被提出的value才能被选定。
- 只有一个value被选定，并且
- 如果某个进程认为某个value被选定了，那么这个value必须是真的被选定的那个。

我们不去精确地定义其**活性（liveness）**要求。我们的目标是保证**最终有一个提出的value被选定**。当一个value被选定后，进程最终也能学习到这个value。

> Paxos的目标：保证最终有一个value会被选定，当value被选定后，进程最终也能获取到被选定的value。

假设不同角色之间可以通过发送消息来进行通信，那么：

- 每个角色以任意的速度执行，可能因出错而停止，也可能会重启。一个value被选定后，所有的角色可能失败然后重启，除非那些失败后重启的角色能记录某些信息，否则等他们重启后无法确定被选定的值。
- 消息在传递过程中可能出现任意时长的延迟，可能会重复，也可能丢失。但是消息不会被损坏，即消息内容不会被篡改（拜占庭将军问题）。

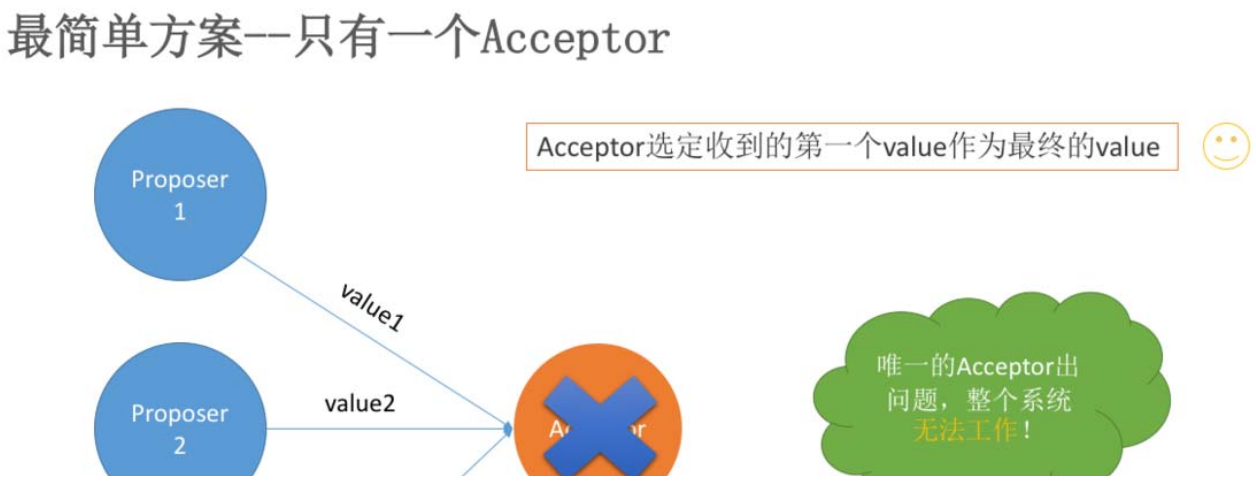
推导过程

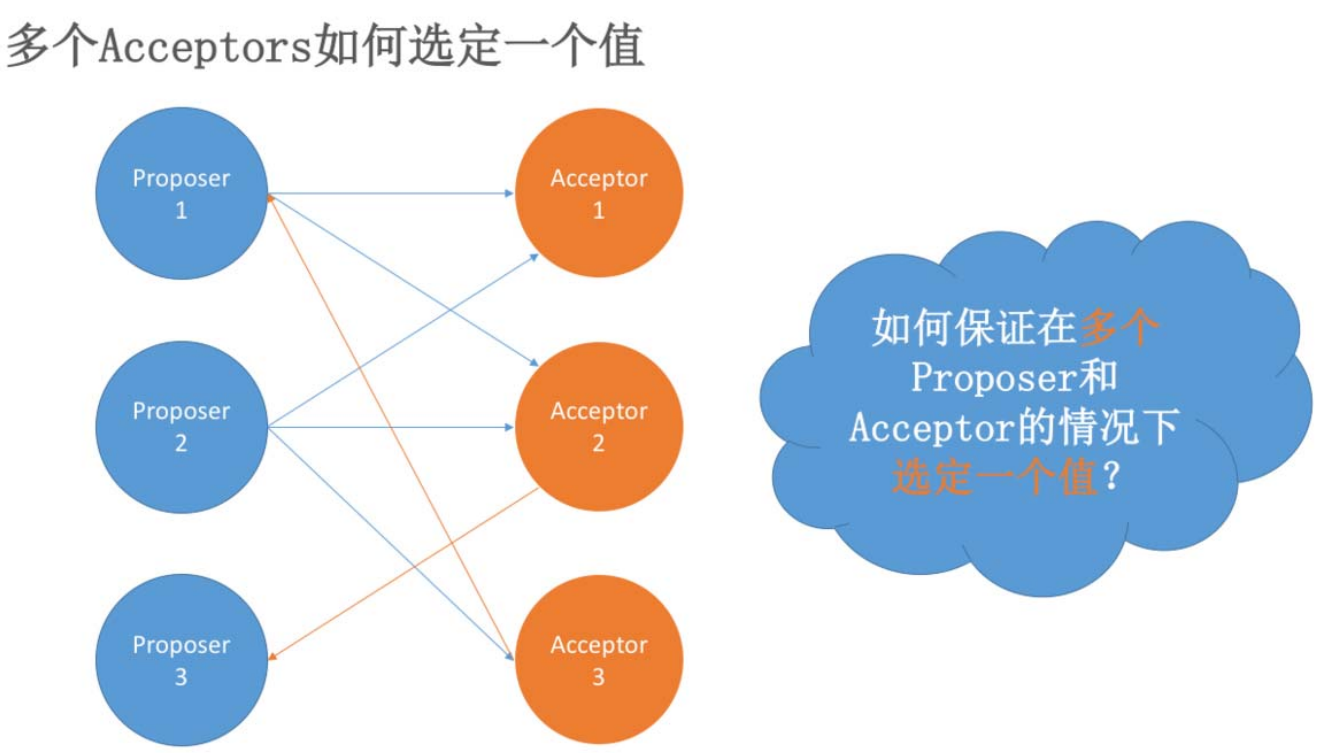
最简单的方案——只有一个Acceptor

假设只有一个Acceptor（可以有多个Proposer），只要Acceptor接受它收到的第一个提案，则该提案被选定，该提案里的value就是被选定的value。这样就保证只有一个value会被选定。

但是，如果这个唯一的Acceptor宕机了，那么整个系统就**无法工作**了！

因此，必须要有**多个Acceptor**！





下面开始寻找解决方案。

如果我们希望即使只有一个Proposer提出了一个value，该value也最终被选定。

那么，就得到下面的约束：

> P1：一个Acceptor必须接受它收到的第一个提案。

但是，这又会引出另一个问题：如果每个Proposer分别提出不同的value，发给不同的Acceptor。根据P1，Acceptor分别接受自己收到的value，就导致不同的value被选定。出现了不一致。如下图：

刚刚是因为『一个提案只要被一个Acceptor接受，则该提案的value就被选定了』才导致了出现上面不一致的问题。因此，我们需要加一个规定：

> 规定：一个提案被选定需要被**半数以上**的Acceptor接受

这个规定又暗示了：『一个Acceptor必须能够接受不止一个提案！』不然可能导致最终没有value被选定。比如上图的情况。v1、v2、v3都没有被选定，因为它们都只被一个Acceptor的接受。

最开始讲的『**提案=value**』已经不能满足需求了，于是重新设计提案，给每个提案加上一个提案编号，表示提案被提出的顺序。令『**提案=提案编号+value**』。

虽然允许多个提案被选定，但必须保证所有被选定的提案都具有相同的value值。否则又会出现不一致。

于是有了下面的约束：

> P2：如果某个value为v的提案被选定了，那么每个编号更高的被选定提案的value必须也是v。

一个提案只有被Acceptor接受才可能被选定，因此我们可以把P2约束改写成对Acceptor接受的提案的约束P2a。

> P2a：如果某个value为v的提案被选定了，那么每个编号更高的被Acceptor接受的提案的value必须也是v。

只要满足了P2a，就能满足P2。

但是，考虑如下的情况：假设总的有5个Acceptor。Proposer2提出[M1,V1]的提案，Acceptor2~5（半数以上）均接受了该提案，于是对于Acceptor2~5和Proposer2来讲，它们都认为V1被选定。Acceptor1刚刚从宕机状态恢复过来（之前Acceptor1没有收到过任何提案），此时Proposer1向Acceptor1发送了[M2,V2]的提案（V2≠V1且M2>M1），对于Acceptor1来讲，这是它收到的第一个提案。根据P1（一个Acceptor必须接受它收到的第一个提案。），Acceptor1必须接受该提案！同时Acceptor1认为V2被选定。这就出现了两个问题：

1. Acceptor1认为V2被选定，Acceptor2~5和Proposer2认为V1被选定。出现了不一致。
2. V1被选定了，但是编号更高的被Acceptor1接受的提案[M2,V2]的value为V2，且V2≠V1。这就跟P2a（如果某个value为v的提案被选定了，那么每个编号更高的被Acceptor接受的提案的value必须也是v）矛盾了。

推导过程

P2a是对Acceptor接受的提案约束，但其实提案是Proposer提出来的，所有我们可以对Proposer提出的提案进行约束。得到P2b：

> P2b：如果某个value为v的提案被选定了，那么之后任何Proposer提出的编号更高的提案的value必须也是v。

由P2b可以推出P2a进而推出P2。

那么，如何确保在某个value为v的提案被选定后，Proposer提出的编号更高的提案的value都是v呢？

只要满足P2c即可：

> P2c：对于任意的N和V，如果提案[N, V]被提出，那么存在一个半数以上的Acceptor组成的集合S，满足以下两个条件中的任意一个：

- S中每个Acceptor都没有接受过编号小于N的提案。
- S中Acceptor接受过的最大编号的提案的value为V。

Proposer生成提案

为了满足P2b，这里有个比较重要的思想：Proposer生成提案之前，应该先去『**学习**』已经被选定或者可能被选定的value，然后以该value作为自己提出的提案的value。如果没有value被选定，Proposer才可以自己决定value的值。这样才能达成一致。这个学习的阶段是通过一个『**Prepare请求**』实现的。

于是我们得到了如下的**提案生成算法**：

1. Proposer选择一个**新的提案编号N**，然后向**某个Acceptor集合**（半数以上）发送请求，要求该集合中的每个Acceptor做出如下响应（response）。（a）向Proposer承诺保证**不再接受**任何编号**小于N的提案**。

   （b）如果Acceptor已经接受过提案，那么就向Proposer响应**已经接受过**的编号小于N的**最大编号的提案**。

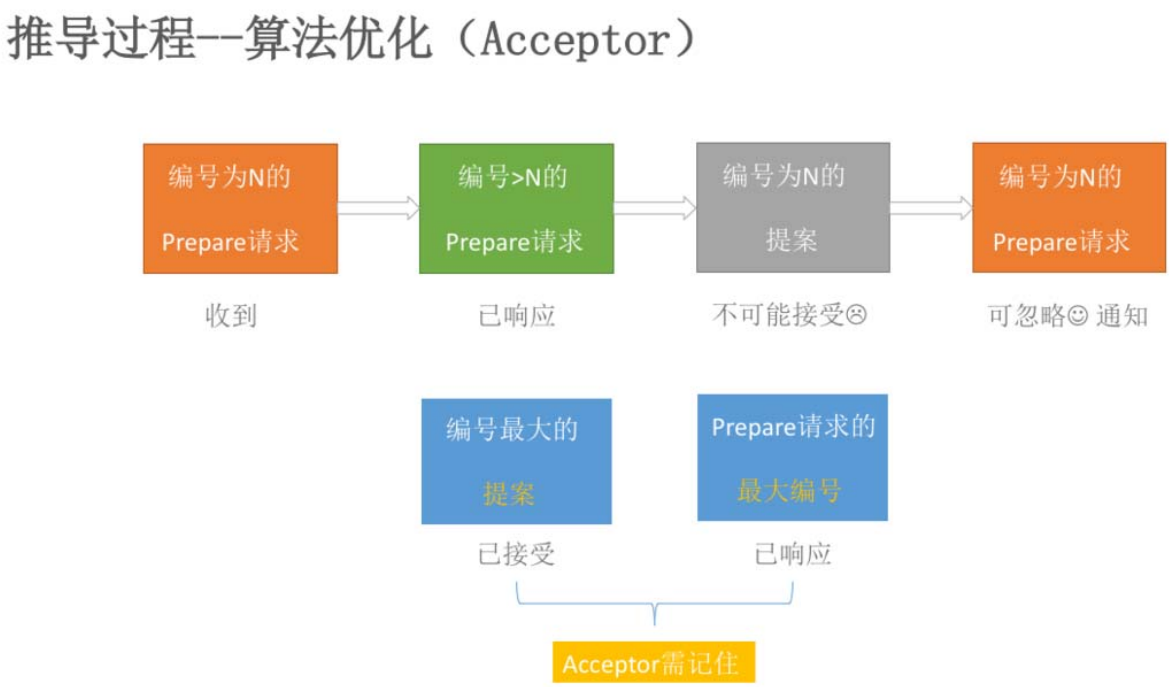
   我们将该请求称为**编号为N**的**Prepare请求**。

2. 如果Proposer收到了**半数以上**的Acceptor的**响应**，那么它就可以生成编号为N，Value为V的**提案[N,V]**。这里的V是所有的响应中**编号最大的提案的Value**。如果所有的响应中**都没有提案**，那 么此时V就可以由Proposer**自己选择**。
   生成提案后，Proposer将该**提案**发送给**半数以上**的Acceptor集合，并期望这些Acceptor能接受该提案。我们称该请求为**Accept请求**。（注意：此时接受Accept请求的Acceptor集合**不一定**是之前响应Prepare请求的Acceptor集合）

Acceptor接受提案

Acceptor**可以忽略任何请求**（包括Prepare请求和Accept请求）而不用担心破坏算法的**安全性**。因此，我们这里要讨论的是什么时候Acceptor可以响应一个请求。

我们对Acceptor接受提案给出如下约束：



Paxos算法描述

经过上面的推导，我们总结下Paxos算法的流程。

Paxos算法分为**两个阶段**。具体如下：

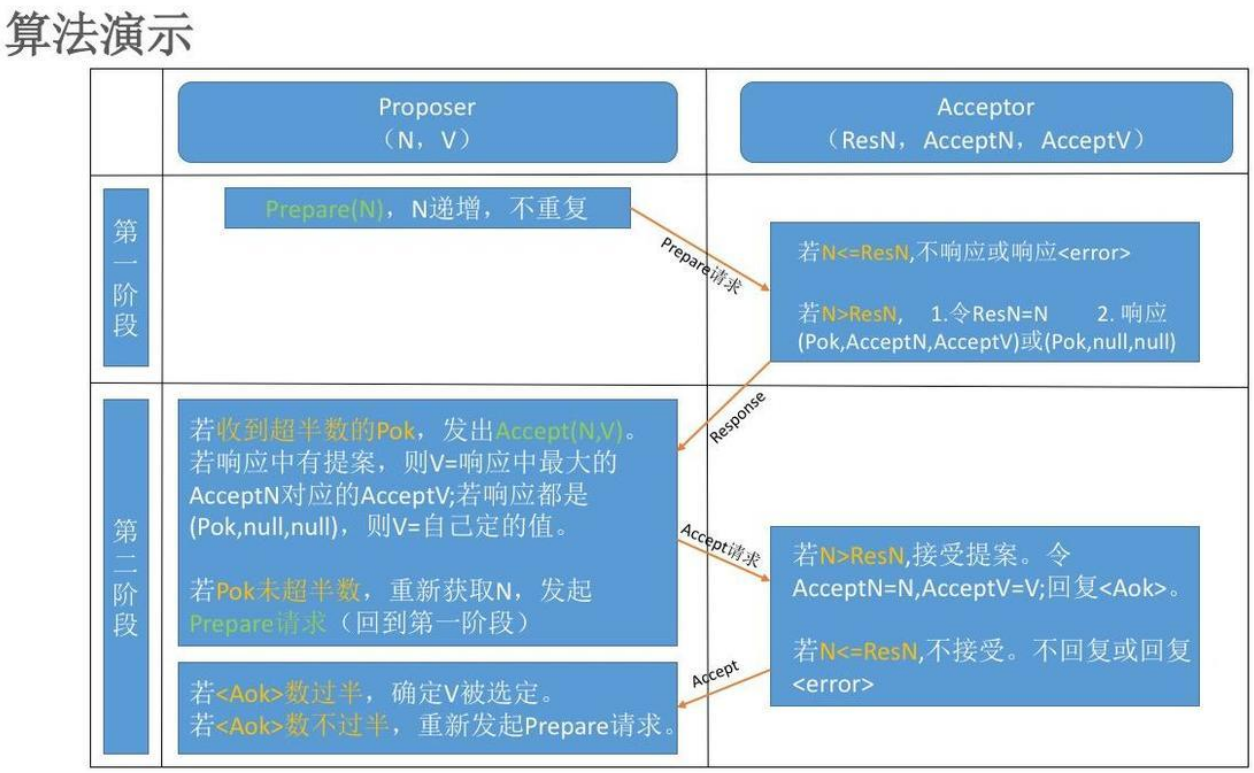
- **阶段一：**

  (a) Proposer选择一个**提案编号N**，然后向**半数以上**的Acceptor发送编号为N的**Prepare请求**。

  (b) 如果一个Acceptor收到一个编号为N的Prepare请求，且N**大于**该Acceptor已经**响应过的**所有**Prepare请求**的编号，那么它就会将它已经**接受过的编号最大的提案（如果有的话）**作为响应反馈给Proposer，同时该Acceptor承诺**不再接受**任何**编号小于N的提案**。

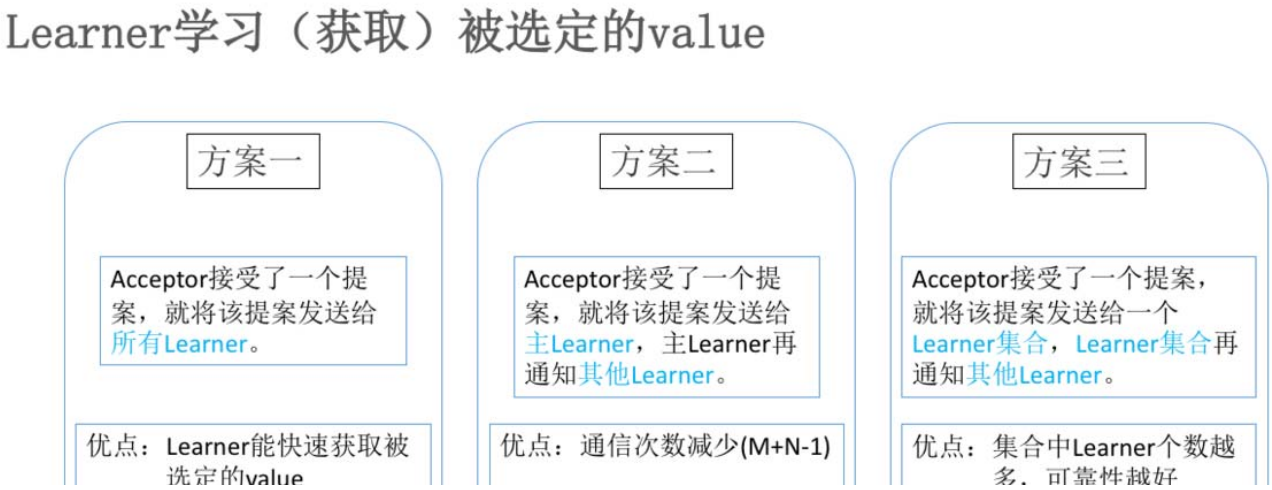
- **阶段二：**

  (a) 如果Proposer收到**半数以上**Acceptor对其发出的编号为N的Prepare请求的**响应**，那么它就会发送一个针对**[N,V]提案**的**Accept请求**给**半数以上**的Acceptor。注意：V就是收到的**响应**中**编号最大的提案的value**，如果响应中**不包含任何提案**，那么V就由Proposer**自己决定**。



Learner学习被选定的value

Learner学习（获取）被选定的value有如下三种方案：



通过选取**主Proposer**，就可以保证Paxos算法的活性。至此，我们得到一个**既能保证安全性，又能保证活性**的**分布式一致性算法——Paxos算法。**