随着大型网站的各种高并发访问、海量数据处理等场景越来越多，如何实现网站的高可用、易伸缩、可扩展、安全等目标就显得越来越重要。为了解决这样一系列问题，大型网站的架构也在不断发展。提高大型网站的高可用架构，不得不提的就是分布式。在[分布式一致性](http://www.hollischuang.com/archives/663)一文中主要介绍了分布式系统中存在的一致性问题。本文将简单介绍如何有效的解决分布式的一致性问题,其中包括什么是**分布式事务**，**二阶段提交**和**三阶段提交**。

**分布式一致性回顾**

在分布式系统中，为了保证数据的高可用，通常，我们会将数据保留多个副本(replica)，这些副本会放置在不同的物理的机器上。为了对用户提供正确的增\删\改\差等语义，我们需要保证这些放置在不同物理机器上的副本是一致的。

为了解决这种分布式一致性问题，前人在性能和数据一致性的反反复复权衡过程中总结了许多典型的协议和算法。其中比较著名的有**二阶提交协议**（Two Phase Commitment Protocol）、**三阶提交协议**（Three Phase Commitment Protocol）和**Paxos算法**。

**分布式事务**

分布式事务是指会涉及到操作多个数据库的事务。其实就是将对同一库事务的概念扩大到了对多个库的事务。目的是为了保证分布式系统中的数据一致性。分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）

在分布式系统中，各个节点之间在物理上相互独立，通过网络进行沟通和协调。由于存在事务机制，可以保证每个独立节点上的数据操作可以满足ACID。但是，相互独立的节点之间无法准确的知道其他节点中的事务执行情况。所以从理论上讲，两台机器理论上无法达到一致的状态。如果想让分布式部署的多台机器中的数据保持一致性，那么就要保证在所有节点的数据写操作，要不全部都执行，要么全部的都不执行。但是，一台机器在执行本地事务的时候无法知道其他机器中的本地事务的执行结果。所以他也就不知道本次事务到底应该commit还是 roolback。所以，常规的解决办法就是引入一个“协调者”的组件来统一调度所有分布式节点的执行。

**XA规范**

X/Open 组织（即现在的 Open Group ）定义了分布式事务处理模型。 X/Open DTP 模型（ 1994 ）包括应用程序（ AP ）、事务管理器（ TM ）、资源管理器（ RM ）、通信资源管理器（ CRM ）四部分。一般，常见的事务管理器（ TM ）是交易中间件，常见的资源管理器（ RM ）是数据库，常见的通信资源管理器（ CRM ）是消息中间件。 通常把一个数据库内部的事务处理，如对多个表的操作，作为本地事务看待。数据库的事务处理对象是本地事务，而分布式事务处理的对象是全局事务。所谓全局事务，是指分布式事务处理环境中，多个数据库可能需要共同完成一个工作，这个工作即是一个全局事务，例如，一个事务中可能更新几个不同的数据库。对数据库的操作发生在系统的各处但必须全部被提交或回滚。此时一个数据库对自己内部所做操作的提交不仅依赖本身操作是否成功，还要依赖与全局事务相关的其它数据库的操作是否成功，如果任一数据库的任一操作失败，则参与此事务的所有数据库所做的所有操作都必须回滚。一般情况下，某一数据库无法知道其它数据库在做什么，因此，在一个 DTP 环境中，交易中间件是必需的，由它通知和协调相关数据库的提交或回滚。而一个数据库只将其自己所做的操作（可恢复）影射到全局事务中。

XA 就是 X/Open DTP 定义的交易中间件与数据库之间的接口规范（即接口函数），交易中间件用它来通知数据库事务的开始、结束以及提交、回滚等。 XA 接口函数由数据库厂商提供。

**二阶提交协议**和**三阶提交协议**就是根据这一思想衍生出来的。可以说二阶段提交其实就是实现**XA分布式事务**的关键(确切地说：两阶段提交主要保证了分布式事务的原子性：即所有结点要么全做要么全不做)

**2PC**

二阶段提交(Two-phaseCommit)是指，在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。通常，二阶段提交也被称为是一种协议(Protocol))。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要引入一个作为协调者的组件来统一掌控所有节点(称作参与者)的操作结果并最终指示这些节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。因此，**二阶段提交的算法思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。**

所谓的两个阶段是指：第一阶段：**准备阶段(投票阶段)**和第二阶段：**提交阶段（执行阶段）**。

**准备阶段**

事务协调者(事务管理器)给每个参与者(资源管理器)发送Prepare消息，每个参与者要么直接返回失败(如权限验证失败)，要么在本地执行事务，写本地的redo和undo日志，但不提交，到达一种“万事俱备，只欠东风”的状态。

可以进一步将准备阶段分为以下三个步骤：

1）协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应。

2）参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。（注意：若成功这里其实每个参与者已经执行了事务操作）

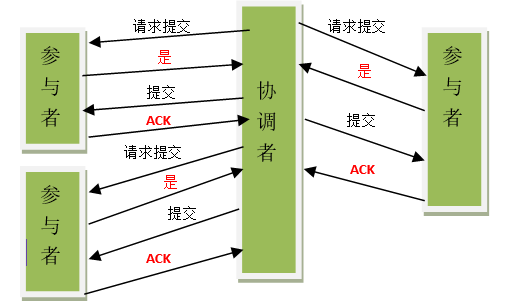
3）各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个”同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个”中止”消息。

**提交阶段**

如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源)

接下来分两种情况分别讨论提交阶段的过程。

当协调者节点从所有参与者节点获得的相应消息都为”同意”时:

[](http://www.hollischuang.com/wp-content/uploads/2015/12/success.png)

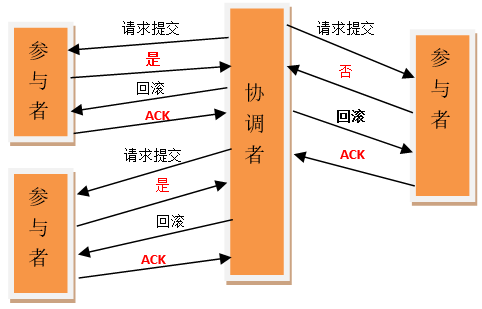
1）协调者节点向所有参与者节点发出”正式提交(commit)”的请求。

2）参与者节点正式完成操作，并释放在整个事务期间内占用的资源。

3）参与者节点向协调者节点发送”完成”消息。

4）协调者节点受到所有参与者节点反馈的”完成”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为”中止”，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：

[](http://www.hollischuang.com/wp-content/uploads/2015/12/fail.png)

1）协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求。

2）参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。

3）参与者节点向协调者节点发送”回滚完成”消息。

4）协调者节点受到所有参与者节点反馈的”回滚完成”消息后，取消事务。

　　不管最后结果如何，第二阶段都会结束当前事务。

二阶段提交看起来确实能够提供原子性的操作，但是不幸的事，二阶段提交还是有几个**缺点**的：

1、**同步阻塞问题**。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

2、**单点故障**。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

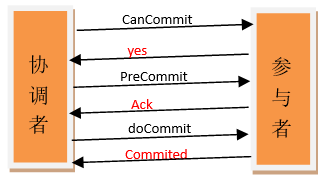
3、**数据不一致**。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

4、二阶段无法解决的问题：协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

由于二阶段提交存在着诸如同步阻塞、单点问题、脑裂等缺陷，所以，研究者们在二阶段提交的基础上做了改进，提出了三阶段提交。

**3PC**

三阶段提交（Three-phase commit），也叫三阶段提交协议（Three-phase commit protocol），是二阶段提交（2PC）的改进版本。

[](http://www.hollischuang.com/wp-content/uploads/2015/12/3.png)

与两阶段提交不同的是，三阶段提交有两个改动点。

1、引入超时机制。同时在协调者和参与者中都引入超时机制。

2、在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC把2PC的准备阶段再次一分为二，这样三阶段提交就有CanCommit、PreCommit、DoCommit三个阶段。

**CanCommit阶段**

3PC的CanCommit阶段其实和2PC的准备阶段很像。协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

**1.事务询问** 协调者向参与者发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

**2.响应反馈** 参与者接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No

**PreCommit阶段**

协调者根据参与者的反应情况来决定是否可以记性事务的PreCommit操作。根据响应情况，有以下两种可能。

**假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务的预执行。**

**1.发送预提交请求** 协调者向参与者发送PreCommit请求，并进入Prepared阶段。

**2.事务预提交** 参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

**3.响应反馈** 如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

**假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。**

**1.发送中断请求** 协调者向所有参与者发送abort请求。

**2.中断事务** 参与者收到来自协调者的abort请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

**doCommit阶段**

该阶段进行真正的事务提交，也可以分为以下两种情况。

**执行提交**

**1.发送提交请求** 协调接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。

**2.事务提交** 参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

**3.响应反馈** 事务提交完之后，向协调者发送Ack响应。

**4.完成事务** 协调者接收到所有参与者的ack响应之后，完成事务。

**中断事务** 协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

**1.发送中断请求** 协调者向所有参与者发送abort请求

**2.事务回滚** 参与者接收到abort请求之后，利用其在阶段二记录的undo信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

**3.反馈结果** 参与者完成事务回滚之后，向协调者发送ACK消息

**4.中断事务** 协调者接收到参与者反馈的ACK消息之后，执行事务的中断。

在doCommit阶段，如果参与者无法及时接收到来自协调者的doCommit或者rebort请求时，会在等待超时之后，会继续进行事务的提交。（其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了PreCommit请求，那么协调者产生PreCommit请求的前提条件是他在第二阶段开始之前，收到所有参与者的CanCommit响应都是Yes。（一旦参与者收到了PreCommit，意味他知道大家其实都同意修改了）所以，一句话概括就是，当进入第三阶段时，由于网络超时等原因，虽然参与者没有收到commit或者abort响应，但是他有理由相信：成功提交的几率很大。 ）

**2PC与3PC的区别**

相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

了解了2PC和3PC之后，我们可以发现，无论是二阶段提交还是三阶段提交都无法彻底解决分布式的一致性问题。Google Chubby的作者Mike Burrows说过， there is only one consensus protocol, and that’s Paxos” – all other approaches are just broken versions of Paxos. 意即**世上只有一种一致性算法，那就是Paxos**，所有其他一致性算法都是Paxos算法的不完整版。后面的文章会介绍这个公认为难于理解但是行之有效的Paxos算法。

Google Chubby的作者Mike Burrows说过， there is only one consensus protocol, and that’s Paxos” – all other approaches are just broken versions of Paxos. 意即**世上只有一种一致性算法，那就是Paxos，所有其他一致性算法都是Paxos算法的不完整版。**

Paxos算法是莱斯利·兰伯特（Leslie Lamport，就是 LaTeX 中的”La”，此人现在在微软研究院）于1990年提出的一种基于消息传递的一致性算法。为描述 Paxos 算法，Lamport 讲述了这样一个故事：

在古希腊有一个岛屿叫做Paxos，这个岛屿通过议会的形式修订法律。执法者（legislators，后面称为牧师priest）在议会大厅（chamber）中表决通过法律，并通过服务员传递纸条的方式交流信息，每个执法者会将通过的法律记录在自己的账目（ledger）上。问题在于执法者和服务员都不可靠，他们随时会因为各种事情离开议会大厅、服务员也有可能重复传递消息（或者直接彻底离开），并随时可能有新的执法者（或者是刚暂时离开的）回到议会大厅进行法律表决，因此，议会协议要求保证上述情况下可以能够正确的修订法律并且不会产生冲突。

**什么是paxos算法**

Paxos 算法是分布式一致性算法用来解决一个分布式系统如何就某个值(决议)达成一致的问题。

人们在理解paxos算法是会遇到一些困境，那么接下来，我们带着以下几个问题来学习paxos算法：

1、paxos到底在解决什么问题？

2、paxos到底如何在分布式存储系统中应用？

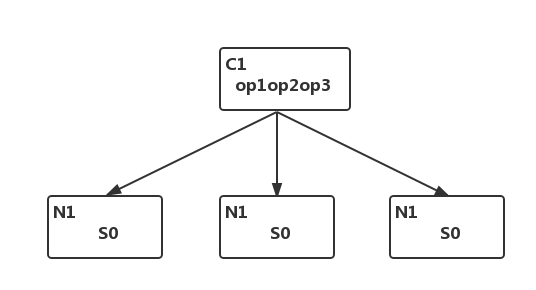
3、paxos的核心思想是什么？

**paxos解决了什么问题**

在[关于分布式一致性的探究](http://www.hollischuang.com/archives/681)中我们提到过，分布式的一致性问题其实主要是指分布式系统中的数据一致性问题。所以，为了保证分布式系统的一致性，就要保证分布式系统中的数据是一致的。

在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。

**所以，paxos算法主要解决的问题就是如何保证分布式系统中各个节点都能执行一个相同的操作序列。**

[](http://www.hollischuang.com/wp-content/uploads/2015/12/Package-Diagram2.png)

上图中，C1是一个客户端，N1、N2、N3是分布式部署的三个服务器，初始状态下N1、N2、N3三个服务器中某个数据的状态都是S0。当客户端要向服务器请求处理操作序列：op1op2op3时（op表示operation）（这里把客户端的写操作简化成向所有服务器发送相同的请操作序列，实际上可能通过Master/Slave模式处理）。如果想保证在处理完客户端的请求之后，N1、N2、N3三个服务器中的数据状态都能从S0变成S1并且一致的话（或者没有执行成功，还是S0状态），就要保证N1、N2、N3在接收并处理操作序列op1op2op3时，严格按照规定的顺序**正确**执行opi，要么全部执行成功，要不就全部都不执行。

所以，针对上面的场景，paxos解决的问题就是如何依次确定不可变操作opi的取值，也就是确定第i个操作什么，在确定了opi的内容之后，就可以让各个副本执行opi操作。

**Paxos算法详解**

Paxos是一个十分巧妙的一致性算法，但是他也十分难以理解，就连他的作者Lamport都被迫对他做过多种讲解。**我认为对paxos算法讲解的最清楚的就是**[**维基百科**](https://zh.wikipedia.org/wiki/Paxos%E7%AE%97%E6%B3%95)**了。但是要看懂维基百科中的介绍需要很强的数学思维（paxos毕竟是一个算法），而且有很多关于定理的推论、证明等过程。那么本篇文章主要站在程序的角度，通俗的，循序渐进的讲解到底什么是paxos算法。**

我们先把前面的场景简化，把我们现在要解决的问题简化为**如何确定一个不可变变量的取值**（每一个不可变变量可以标识一个操作序列中的某个操作，当确保每个操作都正确之后，就可以按照顺序执行这些操作来保证数据能够准确无误的从一个状态转变成另外一个状态了）。

接下来，请跟我一步一步的学习paxos算法。

要学习paxos算法，我们就要从他要解决的问题出发，假如没有paxos算法，当我们面对**如何确定一个不可变变量的取值**这样一个吻问题的时候，我们应该如何解决呢？

这里暂不介绍paxos中的角色的概念，读者可以自行从[维基百科](https://zh.wikipedia.org/wiki/Paxos%E7%AE%97%E6%B3%95)中了解。不了解的话也可以直接往下看，看着看着就了解了。

**问题抽象**

我们把确定一个不可变变量的取值问题定义成：

设计一个系统，来存储名称为var的变量。

var的取值可以是任意二进制数

系统内部由多个Accepter组成,负责管理和存储var变量。

系统对外提供api,用来设置var变量的值  
propose(var,V) => <ok,f> or <error>

将var的值设置为V，系统会返回ok和系统中已经确定的取值f，或者返回error。

外部有多个Proposer机器任意请求系统，调用系统API(propose(var,V) => <ok,f> or <error>)来设置var变量的值。

如果系统成功的将var设置成了V，那么返回的f应该就是V的值。否则，系统返回的f就是其他的Proposer设置的值。

>

系统需要保证var的取值满足一致性

如果var没有被设置过，那么他的初始值为null

一旦var的值被设置成功，则不可被更改，并且可以一直都能获取到这个值

系统需要满足容错特性

可以容忍任意proposer出现故障  
可以容忍少数acceptor故障（半数以下）

暂时忽略网络分化问题和acceptor故障导致var丢失的问题。

到这里，问题已经抽象完成了，读者可以再仔细看看上面的系统描述。如果这样设置一个系统，是不是就可以保证变量var的不可变性了呢？

这里还是再简单讲解一下，上面的系统确实可以保证变量var的不可变性。

因为var的初始值为null，当有proposer请求接口propose（var，v）设置var的值的时候，系统会将var设置为v，并返回f（f==v）。

var变量被初始化以后，再有proposer请求propose(var,v)设置var的值的时候，系统会直接返回系统中已有的var的值f，而放弃proposer提供的v。

**系统难点**

要设计以上系统存在以下难点：

1、管理多个proposer并发执行

2、容忍var变量的不可变性

3、容忍任意Proposer的故障

4、容忍半数以下的acceptor的故障

**解决方案一**

先考虑整个系统由单个acceptor组成。通过类似[互斥锁](https://zh.wikipedia.org/wiki/%E4%BA%92%E6%96%A5%E9%94%81)的方式来管理并发的proposer的请求。

proposer向acceptor申请acceptor的互斥访问权，当取得互斥访问权之后才能调用api给var变量赋值。  
accepter向proposer发放互斥访问权，谁取得了互斥访问权，acceptor就接收谁的请求。  
这样通过互斥访问的机制，proposer就要按照获取互斥访问权的顺序来请求系统。  
一旦acceptor接收到一个proposer请求，并成功给var变量赋值之后，就不再允许其他的proposer设置var变量的值。每当再有proposer来请求设置var变量的值的时候，acceptor就会将var里面现有的值返回给他。

**基于互斥访问权的acceptor的实现**

acceptor会保存变量var的值和一个互斥锁Lock。

提供接口prepare()

加互斥锁，给予var的互斥访问权，并返回当前var的取值

提供接口release()

用于释放互斥访问权

提供接口accept(var, v)

如果已经加锁，并且当前var没有值，则将var的值设置成v，并释放锁。

**proposer采用两阶段来实现**

Step1、通过调用prepare接口来获取互斥性访问权和当前var的取值

如果无法获取到互斥性访问权，则返回，并不能进入到下一个阶段，因为其他proposer获取到了互斥性访问权。

Step2、根据当前var的取值f选择执行

1、如果f的取值为null，说明没有被设置过值，则调用接口accept(var ,v)来将var的取值设置成v，并释放掉互斥性访问权。  
2、如果f的取值不为null，说明var已经被其他proposer设置过值，则调用release接口释放掉互斥性访问权。

总结：方案一通过互斥访问的方式来保证所有的proposer能够串行的访问acceptor，这样其实并没有解决多个proposer并发执行的问题。只是想办法绕开了并发执行。虽然可以在一定程度上保证var变量的取值是确定的。但是一旦获取到互斥访问权的proposer在执行过程中出现故障，那么就会导致所有其他proposer无法再获取到互斥访问权，就会发生死锁。。所以，方案一不仅效率低、而且还会产生死锁问题，不能容忍任意Proposer出现故障。  
在之前提到的四个系统难点中，方案一可以解决难点1和难点2，但是无法解决难点3和难点4。

**解决方案二**

通过引入[抢占式访问权](https://zh.wikipedia.org/wiki/%E6%8A%A2%E5%8D%A0%E5%BC%8F%E5%A4%9A%E4%BB%BB%E5%8A%A1%E5%A4%84%E7%90%86)来取代互斥访问权。acceptor有权让任意proposer的访问权失效，然后将访问权发放给其他的proposer。

在方案二中，proposer向acceptor发出的每次请求都要带一个编号（epoch），且编号间要存在全序关系。一旦acceptor接收到proposer的请求中包含一个更大的epoch的时候，马上让旧的epoch失效，不再接受他们提交的取值。然后给新的epoch发放访问权，让他可以设置var变量的值。

为了保证var变量取值的不变性，不同epoch的proposer之前遵守后者认同前者的原则：

在确保旧的epoch已经失效后，并且旧的epoch没有设置var变量的值，新的epoch会提交自己的值。  
当旧的epoch已经设置过var变量的取值，那么新的epoch应该认同旧的epoch设置过的值，并不在提交新的值。

**基于抢占式访问权的acceptor的实现**