Paxos算法解决的问题：在分布式系统中，如何就某个值（决议）达成一致的问题。

通过Paxos可以实现多副本的一致性 分布式锁 名字管理 序列号分配等。

Paxos算法是基于消息传递且具有高度容错性的一致性算法。

在分布式系统中，会发生诸如机器宕机 网络异常（包括消息的延迟 丢失 重复 乱序 网络分区）等情况。Paxos算法解决的是正在一个可能发生上述异常的分布式系统中，快速并且正确的在集群内部对某个数据的值达成一致，并且保证无论发生以上任何异常，都不会破坏系统的一致性。

（这里某个数据的值不只是狭义上的某个数，也可以是一条日志 命令等，不同场景有不同的含义）比如在多个节点中选主，或者进行日志同步等。

Paxos是一个用于实现可容错的分布式系统的算法，主要用于保证分布式集群中多备份系统之间的操作和数据的一致性。这样集群中的每台机器对外相当于完全一致，从而相互之间互为备份，从而使系统能够容忍一定数量的机器出现问题。

主要的角色：

Proposer 提议者（接收用户请求，并向集群提出提案要求执行该request）

Acceptor 决策者（接收来自proposer的提案）

Learner 最终决策学习者 也是执行者（从acceptor中学习到已经批准过得提案）

一个进程可以同时充当多种角色。

重要的概念提案：最终要达成一致的value就在提案中。

注：暂且认为提案=value，在接下来的推导中，发现如果提案=value，会有问题，需要重新设计提案。

暂且任务proposer可以直接提出提案，在接下来的推导中，发现如果proposer直接提出提案会有提案，因此需要一个学习提案的过程。

对某个数据值达成一致指的是proposer acceptor learner都认为同一个value被选定。

Proposer：只要proposer发出的提案被acceptor接受（刚开始认为只需要一个acceptor接受即可，在推导过程中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。

Acceptor：只要acceptor接受了某个提案，acceptor就认为该提案里的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。

对于一致性算法，安全性要求如下：

只有提出的value才能被选定

只有一个value被选定

该value只有被选定后，才能被其他节点知道

Paxos算法目标：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

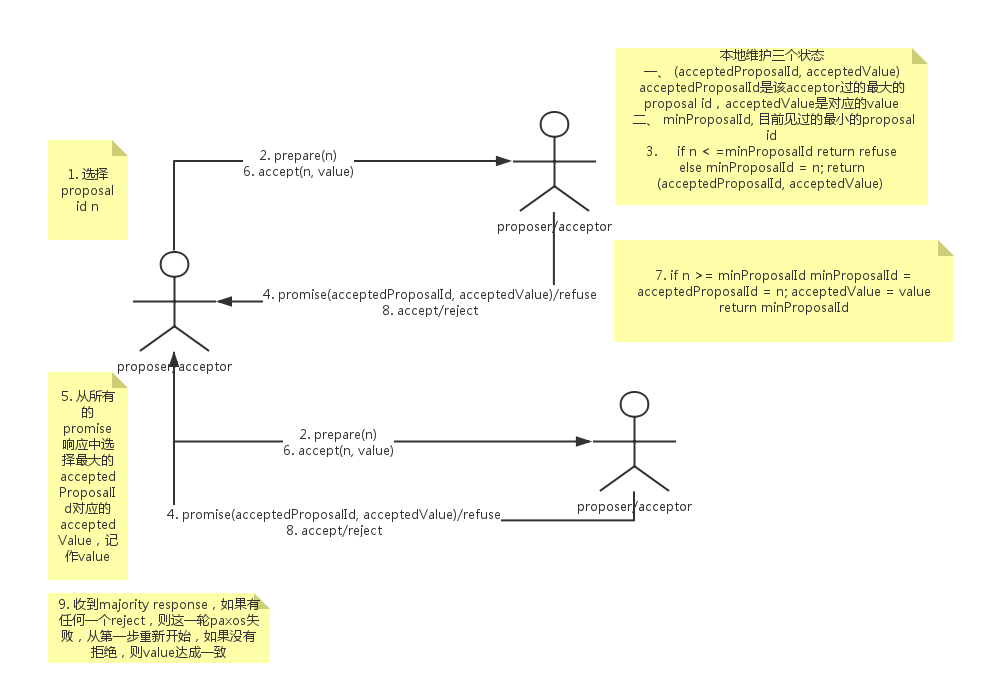
推导过程：

见paxos made simple文档或者

<http://www.cnblogs.com/linbingdong/p/6253479.html>

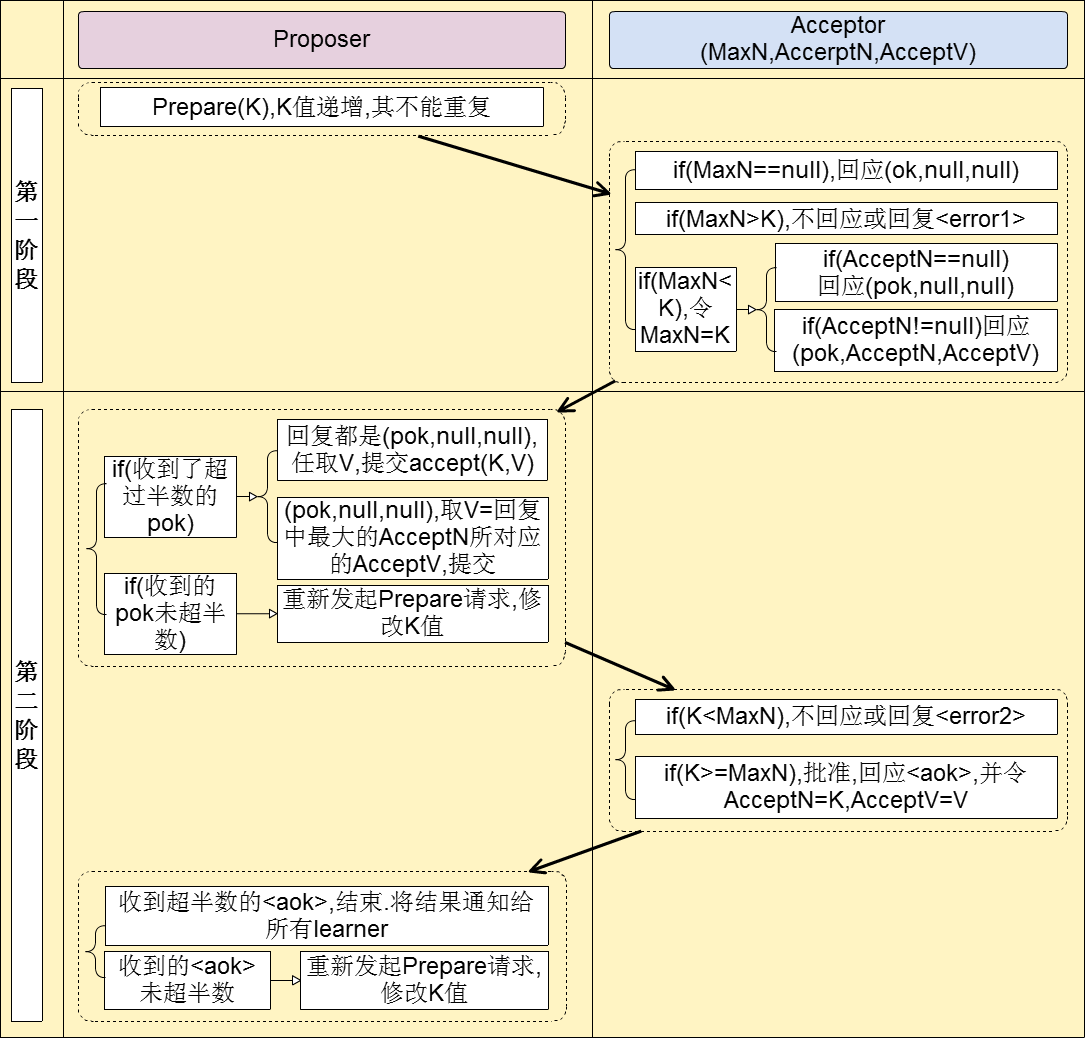
<http://blog.csdn.net/shenlan211314/article/details/7294312>

paxos算法总结：<http://www.udpwork.com/item/13394.html>



Value在被多数accept之前，每个acceptor都可以接受多个不同的值，但是一旦这个value被多数accept（即value达成一致），那么这个value不会再改变（选定）。因为在prepare阶段会将该value给找出来，随后accept阶段会使用这个value，后续所有的提案都会使用这个value。

需要注意的是，每个阶段都是收到majority的响应后即开始处理。并且由于机器会宕机，acceptor需要对acceptedProposalID, acceptedValue和minProposal进行持久化。



解释：paxos算法主要分为两个阶段，第一阶段是prepare阶段，发送的是prepare request。proposer会发送提案编号K（递增）给acceptor，acceptor收到提案编号之后，会进行逻辑判断。如果acceptor是第一次收到提案编号，则会记录该提案编号，并向proposer回复ok，如果acceptor不是第一次收到提案编号，则会将编号N和之前记录下来的最大的提案编号maxN进行比较，如果k<maxN，则会忽略该prepare request，如果k>=maxN，则会更新maxN=k，如果此时acceptor之前没有批准过任何提案，则直接向proposer回复ok，如果acceptor之前有批准过提案，则在回复ok的时候，会将之前批准过提案的编号和值也传给proposer。

在第二阶段，是提议阶段，发送的是accept request。如果proposer收到了过半数的回复，此时如果回复中没有值value（对应于acceptor没有批准过提案），则proposer会发送编号k和值value（自定义）给acceptor（不一定是第一阶段对应的acceptor），如果回复中有value，则会选取回复中最大编号对应的value作为值发送出去。如果proposer没有收到过半的回复，则会重新发起prepare请求，并递增编号k的值。此时acceptor收到了accept 请求后，会进行判断，如果k<maxN，则不回应（说明在这期间，有其他的proposer提出的提案编号更大），否则则批准该提案，记录提案编号和值，并回复ok。Proposer收到过半的回复后，会将结果通知给learner（选定的提案），如果没有收到过半的回复，则会重新发起prepare请求，递增编号k的值。

如何产生全局唯一且递增的proposer number？

可以采用时间戳的高位+ip地址的低位组成

可以让所以proposer都从不相交的数据集合中选择，如系统中有5个proposer，则可以为每个proposer分配一个标识j（0-4），则每个proposer每次提出的编号可以为5\*i+j，i为提出提案的次数。

注意：paxos在第一阶段发送的prepare请求时要求发送给大多数的acceptor。Paxos算法适应于无拜占庭问题的场景，且宕机的节点不能过半，否则不能工作。

每个proposer并不是要自己的提案一定能够通过，而是希望尽快达成一致，所以在收到acceptor回复之后，会修改自己的提案中的value。

活性问题

两个proposer不断尝试提出更高版本的提案时发生，并且任何一个都不会被acceptor批准。

例子：p1在第一阶段提出版本号为n1的提议，p2在第一阶段提出版本号为n2的提议，且n2>n1，那么在p1的第二阶段，版本号为n1的提议会被忽略，那么p1会尝试提出版本号为n3的提议，且n3>n1，这就会导致p2在第二阶段的失败，然后p2又提出更高版本的提议，一直循环下去。。。这就是动态死锁的问题。

为了保证程序的正常运行，系统需要能够选出某个提议者作为唯一的提议者来发出提议。如果该提议者能够和一个大多数集进行有效的通信，并且它提出的版本号大于其他版本号的提议，那么显然，这个提议会被批准。如果提议者在发起提议的过程发现系统中存在比自己提议更高的版本，那么它将立即停止该提议过程，并再次发送更高版本的提议，在一次或者多次尝试后，该提议者将最终能够成功发起提议。

对于该选择，可以使用随机算法或者根据真实时间。让proposer在重新提议之前等待一小段的随机时间来规避。

Paxos算法没有静态死锁，因为请求者可以覆盖接受者的状态，算法又保证了多数派不会被覆盖。

对于任何一个分布式性一致算法，都可能存在无法终止的可能性。

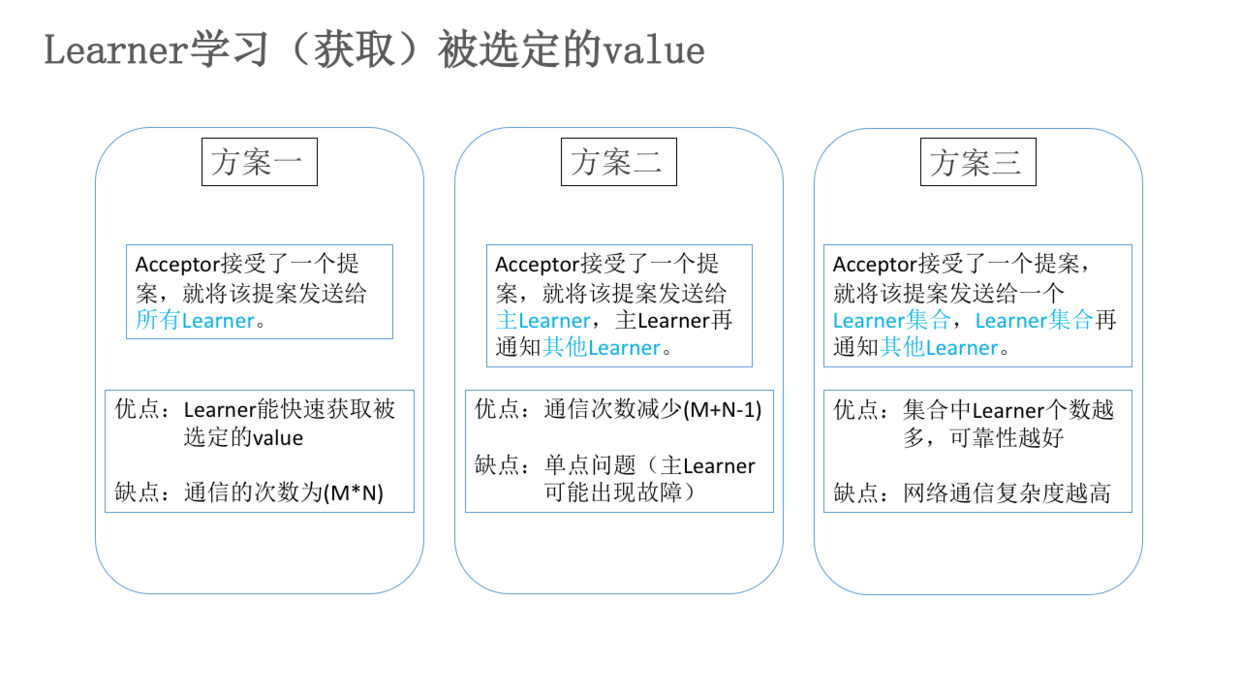
活性：最终能够达成一致

简单描述为：proposer首先从大多数acceptor那里学习提案的最新内容，然后根据学习到的编号最大的提案内容组成新的提案提交，如果提案获得大多数acceptor的投票通过则意味着提案被通过。

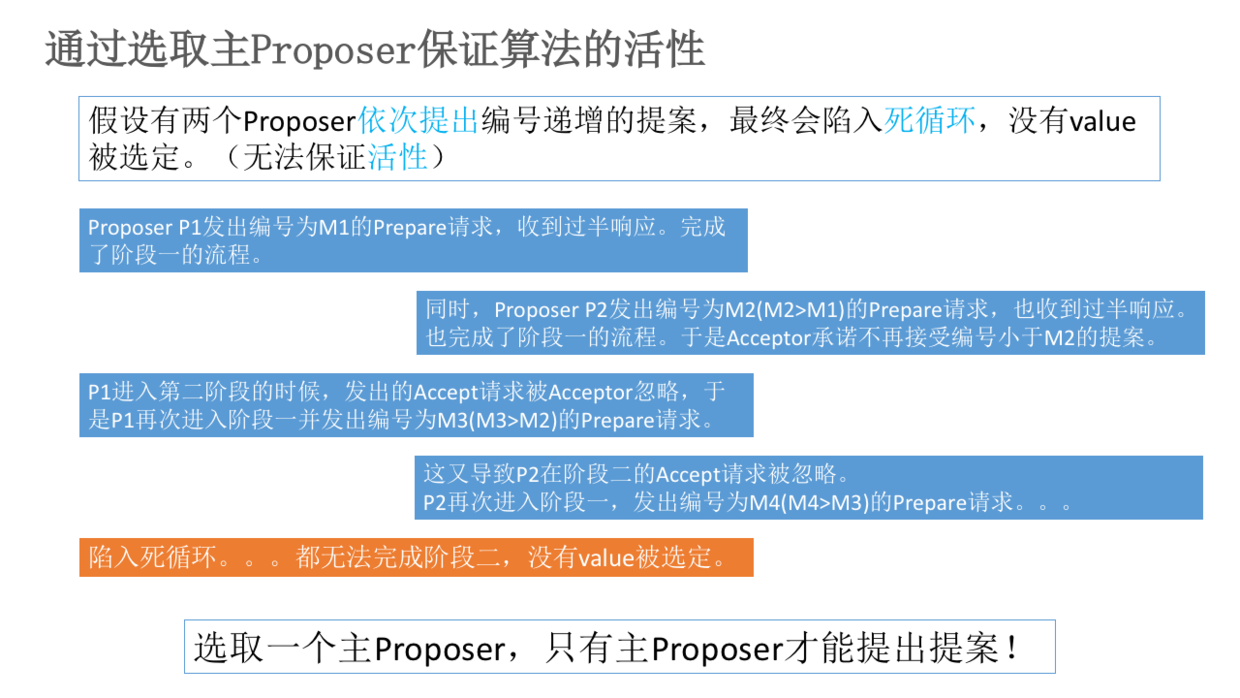
由于学习提案和通过提案的acceptor集合超过半数，所以一定能学到最新通过的提案值，两次提案通过的acceptor集合中也一定存在一个公共的acceptor，在满足约束条件b时，这个公共的acceptor保证了数据的一致性。

保证一致性依赖的事实是：两个多数集合中至少有一个公共成员，将全局信息存储在多数集合返回的统计结果上。

如果一个值曾经通过了，多数集合中至少有一个成员记得它。所以在提议新值之前，先通过prepare阶段从一个多数集合中收集到可能已经通过的值。然后在accept阶段发送可能通过的值，或者在没有值的情况下，提议一个新值。



如何保证活性？



选举出一个leader，这就保证了只有leader可以提出提案。

参考链接：

<http://blog.jobbole.com/106327/>

## paxos算法为什么需要两阶段?

第一个阶段主要是为了阻塞旧的（proposal id小的）提案，并且返回自己已经接受过的提案。

如果没有第一阶段，那么每个proposer都会提出自己的value，这样会导致最终value不一致的问题。（p1向s1 s2 s3提出提案，value=1，p2向s3 s4 s5提出提案，value=2 s3会修改自己的值，这样最后s1 s2值是1，而s3 s4 s5值是2，不一致）（paxos算法要求acceptor必须接受自己收到的第一个提案）

## PaxosLease算法

### Lease

Master选举过程：从众多node节点中选择一个node作为master，如果该master一直alive，则无需选举，如果master宕掉了，则在剩下的node中再次选出一个master。

选择正确性的保证：任何时刻最多只能有一个master

逻辑上，master更像一把锁，任何一个节点拿到这个锁，都可以成为master。所以master选举本质上是分布式锁的问题，但是如果一个node拿到锁之后，就死掉了，那么会导致锁无法释放，产生死锁问题。一种可行的方案是给锁加个时间（lease，租期），拿到锁的master只能在lease有效期内访问锁定的资源，在lease timeout后，其他node可以继续竞争锁，这避免了死锁问题。如果master拿到锁之后，一直alive，那么可以向其他node续租锁，避免频繁的选举活动。

### Master选举

Mater选举和value选举不同之处：

Master选举执行频率低（每个instance都会选出value，比较频繁和高密度）

无paxos持久化，重启后状态丢失

### PaxosLease

<http://blog.csdn.net/chen77716/article/details/6265394>

基于paxos算法和lease的master选举算法。角色与paxos算法一样，行为也相似，区别是：

不要求acceptor持久化数据

不存在leader

Paxoslease将paxos算法的accept阶段重新命名为propose，因此存在prepare propose两个阶段。

T：每个Master的Lease时间（s）

M：全局Lease时间，确保M>T

Ballot number: 每次发送的proposal编号

## MultiPaxos算法

Paxos算法是对一个值达成一致，而multi-paxos算法是多个paxos instance对多个值达成一致，这里最核心的是multi-paxos协议中有一个leader，leader是系统中唯一的proposer，在lease租期内所提出的提案具有相同的proposal id，可以跳过prepare阶段，议案只有accept过程，一个proposal id可以对应多个value，所以称为multi-paxos算法。

参考文档

<http://www.jianshu.com/p/06a477a576bf>

## ZAB协议

### 设计目标

1 一致性

2 有序性：有序性是zab协议和paxos协议的核心区别。Zab协议的有序性主要表现在：

1. 全局有序：如果消息a在消息b之前投递，那么在任何一台服务器，消息a都会在消息b之前被投递
2. 因果有序：如果消息a在消息b之前发生（a导致b），并一起发送，则a始终在b之前被执行

Zookeeper致力于提供一个高性能 高可用 且具有严格的顺序访问控制能力的分布式协调系统。为了保证上述两个有序，zk使用了TCP协议和leader。

通过TCP协议保证了消息的全序特性（先发先到）。

通过leader解决了因果顺序问题：先到leader的先执行，但是这样的话，leader可能出现网络中断 崩溃退出与重启等异常情况，这就引入了leader选举算法。

ZAB：zookeeper atomic broadcast，zookeeper原子消息广播协议——zookeeper数据一致性核心算法

3 容错性：2f+1台服务器，只要大于等于f+1的服务器正常工作，就能完全正常工作。

### 协议内容

Zab协议分为两部分：

广播：zab协议中，所有的写请求由leader处理。正常的工作状态下，leader接收请求并通过广播协议来处理。

恢复：当服务初次启动时，或者leader节点挂了，系统会进入恢复模式，直至选出了有合法数量follower的新leader，然后新leader负责将整个系统同步到最新状态。

#### 广播

类似于一个简化的二阶段提交过程

1 leader收到消息后，将消息赋予一个全局唯一的64位自增id——zxid（zookeeper transaction id），通过zxid大小比较即可实现因果有序这个特性。

2 leader通过先进先出队列（通过tcp协议实现，以此实现了全局有序特性）将带有zxid的消息作为一个提案分发给所有follower。

3 当follower收到提案后，首先将提案写到磁盘中，写入成功后再向leader回复ack

4 当leader收到过半数量的ack后，leader就向所有follower发送commit消息，同时会在本地也执行该消息。

5 当follower收到消息commit后，就会执行该消息。

Zab和两阶段提交不同的是：zab不能终止事务，follower要么回复ack给leader，要么丢弃提案，不回复leader。而两阶段提交协议中，只要收到一个拒绝的消息，就会回滚事务。Zab协议只要求收到过半数量的ack就可以继续下一步骤，也正是因为过半这个特点，让zab协议可以解决两阶段和三阶段提交不能解决的脑裂或者假死等问题。同时，过半也提升了整体性能，因为不需要收到所有的follower的回复。

#### 恢复

广播不能处理leader挂掉的情况，zab协议通过恢复模式来处理这个问题。

要使leader挂了后，系统仍能正常工作，需要解决以下两个问题：

1 已经被处理的消息不能丢

2 被丢弃的消息不能再次出现

* 已经被处理的消息不能丢

当leader收到集群中过半数量的ack后，就会向各个follower广播commit消息，同时也会在本地执行commit并向连结的客户端返回成功。但是如果各个follower在收到commit前，leader就挂了，导致剩下的服务器没有执行这个消息。（此时leader自身已经执行了commit并向客户端返回成功）

Zab采取以下方式解决了这个问题：

1 在进行leader选举时，会选举出proposal最大值（即zxid最大）的节点作为新的leader：由于所有提案被commit之前必须有合法数量的follower ack，即必须有过半数量的服务器的事务日志上有该提案的proposal，因此只要有过半数量的节点正常工作，就必然有个节点保存了所有被commit消息的proposal状态。

2 新的leader将自己事务日志中proposal但未commit的消息进行处理。

3 新的leader与follower建立先进先出的队列，将自身有而follower没有的proposal发送给follower，再将这些proposal的commit命令发送给follower，以保证所有的follower都保存了所有的proposal、所有的follower都处理了所有的消息。

* 被丢弃的消息不能再次出现

当leader接收到消息请求生成proposal后，就挂了，其他的follower并没有收到此proposal，当通过恢复模式重新选了leader后，这条消息是被跳过的。此时之前挂了的leader重新启动并注册成了follower，他保留了跳过消息的proposal状态，与整个系统的状态是不一致的，需要将其删除。

Zab通过zxid来解决这个问题。一个zxid是64位组成，高32位是纪元（epoch）编号，每经过一次leader选举产生一个新的leader，新的leader会将epoch号+1。低32位是消息计数器，每接收到一条消息这个值+1，新的leader选举后这个值置为0.这样设计的好处是，当旧的leader挂了后重启，它不会被选举为leader，因此此时它的zxid肯定小于当前新的leader。当旧的leader作为follower接入新的leader后，新的leader会让它将所有的拥有旧的epoch号的未被commit的proposal清除。

当选举产生了新的leader，并且集群中有过半的机器与该leader服务器完成了状态同步后，zab协议就会退出崩溃模式，进入消息广播模式。

当有新的服务器加入到集群中，如果此时集群中已经存在一个leader在进行消息广播，那么新加入的服务器会自动进入数据恢复模式，找到leader服务器，与其进行数据同步，然后一起参与到消息广播流程中去。

也就是说，一共三个步骤：崩溃恢复（选leader） 数据同步（leader和follower） 消息广播（leader发数据给follower）

Zxid即事务id，是全局单调递增的，分配给每个proposal。Zab协议需要保证每个消息严格有序的因果关系，因此必须将每个proposal按照其zxid的先后顺序

选leader时，首先选zxid最大的server作为leader，如果zxid相同，则再按照序号选择最大的那个。（zk中的每个server都有一个编号id）

Zab leader选举算法要求：

旧的leader宕机后，选举新的leader中，旧的leader重启后不可能再次成为这次选举的新leader。

旧的leader宕机后，在剩下的follower中选取新的leader标准：一定是zxid最大的那个follower成为leader。

事务id（zxid）是64位数字，其中低32位是一个单调递增，高32位代表一个leader从生到死的epoch编号。

新的leader选举出来，从事务proposal分析出旧的leader的epoch编号，并递增1，作为新的事务id的高32位，低32位从0重新计数。

新的leader通过事务id和所有的follower机器上的事务id进行对比，确保数据同步。保证数据在所有的follower上与之同步。旧的leader上新提出的事务被抛弃。当数据达到同步后，才将follower加入到可用的follower服务器列表中。然后开始消息广播。

## Zab和paxos算法

Paxos算法存在一个问题：在一个提案在提议过程中，其他的提案会被否决，否决意味着更多的网络io和更多的冲突。而且paxos算法中存在活锁问题，每个server都可以发起提案，然后被否决后再次提出。

Zab对paxos算法做了改进。由leader进行发起提案。

## Zab如何做到多个副本一致性？

在zab中只有leader可用处理事务请求，将数据变更的操作以事务proposal的形式广播给集群中所有节点，然后等待反馈，如果过半节点通过，再次发送commit请求完成事务变更。

## Zab如何保证数据变更的全局顺序?

通过zxid来保证，zxid是一个64位的数字，前32位是一个epoch编号，每次领导选举之后就会加1，后32位是一个自增的序列，每个事务请求自动加1，并且leader会为所有的follwer维护一个发送队列，将发送的proposal按照先后顺序存入队列中依次发送。

## Zab如何支持崩溃恢复？

集群中所有的服务器都以长连接的形式保持通信，如果集群中超过一半以上的follwer无法连接到leader，就会自动进入领导选举，进而选举出新的leader。

## Zab的工作模式

广播模式（处理事务请求） 恢复模式（leader选举）

## Zab领导选举过程

首先服务器将自身状态转换为looking，并向集群中所有的服务器发起投票（myid,zxid），接收其他服务器发送的投票，并进行处理，按照zxid最大的作为leader，如果相同按照myid最大的作为leader，更新投票信息，再次向集群发送投票  
统计投票结果，超过半数以上的投票即为leader  
如果leader是自己，就把自己的状态变更为leading，否则变为follwing

## Zab事务请求流程

只有leader可以处理事务请求，follower接收到事务请求后要转发给leader，leader以proposal形式发送给所有的follower，等待响应。Follower收到请求后首先记录事务日志，然后返回响应。如果超过半数以上的follower都返回正确的响应，再次向所有的follower发送commit请求，follower开始变更内存数据库。最后响应客户端。

## Zab watch机制

Zookeeper中所有读操作（getData getChildren exists）都可以设置watch。Watch事件是一次性触发的特性。

Watch事件是异步发送到client。Zookeeper保证客户端发送过去的更新顺序是有序的。

Watch是轻量级的，watchevent是最小的通信单元，结构上只包含通知状态 事件类型和节点路径。Zookeeper服务端只会通知客户端发生了什么，并不告诉具体的内容。比如当节点内容发生变更，服务端只会告诉客户端数据节点的内容发生变化，而对于原始数据和变化之后的数据是无法在这个事件中直接获取到，需要客户端主动重新去获取数据。

<https://www.ibm.com/developerworks/cn/opensource/os-cn-apache-zookeeper-watcher/>

Watch的回调执行是顺序执行的，并且客户端在没有收到关注数据的变化事件通知之前是不会看到最新的数据。

客户端在zookeeper服务器上注册watcher时，会将watcher对象存储在客户端的watchManager中。当zk服务器触发了watcher事件后，会向客户端发送通知，客户端从watchManager中取出对应的watcher对象来执行回调逻辑。watchManager中用map集合存放watcher对象。

WatchManager类

private final HashMap<String, HashSet<Watcher>> watchTable =

new HashMap<String, HashSet<Watcher>>();

public synchronized void addWatch(String path, Watcher watcher) {

HashSet<Watcher> list = watchTable.get(path);

if (list == null) {

// don't waste memory if there are few watches on a node

// rehash when the 4th entry is added, doubling size thereafter

// seems like a good compromise

list = new HashSet<Watcher>(4);

watchTable.put(path, list);

}

list.add(watcher);

HashSet<String> paths = watch2Paths.get(watcher);

if (paths == null) {

// cnxns typically have many watches, so use default cap here

paths = new HashSet<String>();

watch2Paths.put(watcher, paths);

}

paths.add(path);

}

Watcher的理念是启动一个客户端接收从zk服务端发送过来的消息并且同步的处理这些消息。Watcher接口中有个回调方法process，还包含keeperState和EventType两个枚举类，代表通知状态和事件类型。

当zk向客户端发送了一个watcher事件后，客户端就会调用相应的process方法回调，从而对事件进行处理。Process方法包含watcherevent类型参数，watcherevent包含了每个事件的三个基本属性：通知状态（keeperstate） 事件类型（eventype）和节点路径（path），zk使用watchedevent对象来封装了服务端事件并传递给watcher，从而方便回调方法process对服务端事件进行处理。

WatchedEvent 和 WatcherEvent 都表示的是同一个事物，都是对一个服务端事件的封装。不同的是，WatchedEvent 是一个逻辑事件，用于服务端和客户端程序执行过程中所需的逻辑对象，而 WatcherEvent 因为实现了序列化接口，因此可以用于网络传输。

服务端在线程 WatchedEvent 事件之后，会调用 getWrapper 方法将自己包装成一个可序列化的 WatcherEvent 事件，如清单 7 所示，以便通过网络传输到客户端。客户端在接收到服务端的这个事件对象后，首先会将 WatcherEvent 事件还原成一个 WatchedEvent 事件，并传递给 process 方法处理，回调方法 process 根据入参就能够解析出完整的服务端事件了。

服务端只存储事件的信息,客户端存储事件的信息和Watcher的执行逻辑.ZooKeeper客户端是线程安全的每一个应用只需要实例化一个ZooKeeper客户端即可，同一个ZooKeeper客户端实例可以在不同的线程中使用。ZooKeeper客户端会将这个Watcher对应Path路径存储在ZKWatchManager中,同时通知ZooKeeper服务器记录该Client对应的Session中的Path下注册的事件类型。当ZooKeeper服务器发生了指定的事件后,ZooKeeper服务器将通知ZooKeeper客户端哪个节点下发生事件类型，ZooKeeper客户端再从ZKWatchManager中找到相应Path，取出相应watcher引用执行其回调函数process。

<http://kavy.iteye.com/blog/2102154>

## zookeeper会话管理

zk使用会话管理器sessiontraaacccker来管理sesion。Sessiontracker使用分桶策略来管理session。客户端创建或者刷新session的时候会按照超时时间来把session放到不同的桶中。后台有一个专门线程负责来在固定的时间点定时检查session，如果session在超时时间范围内已经刷新，就会被重新放到新的桶里，如果没有刷新，就会被清理掉。

## Zk的事务存储

Zk事务日志

Zk dump ，定期dump，可以配置日志数量超过过少次之后就可以把内存中的数据dump到快照文件中去。