# 分布式选主

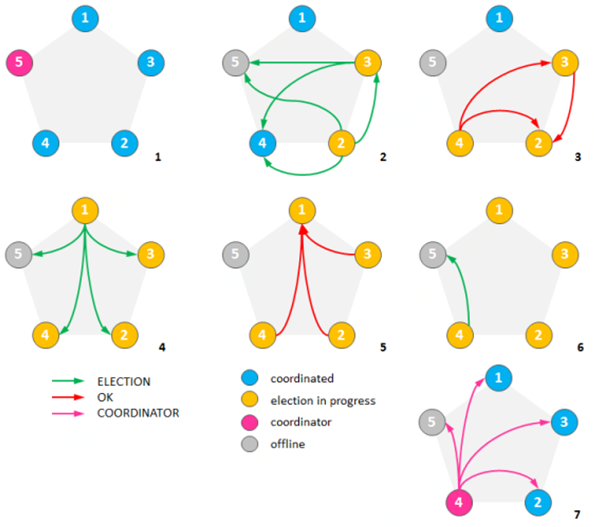
分布式选举算法面临最核心的问题是如何处理网络隔离问题，要保证在网络隔离的场景下，算法有且只有一个主的产生。

系统中有很多应用场景要类似主从架构，主服务器(Master)对外提供服务，从服务器(Salve)热备份，不提供服务但随时活着，如果Master出现宕机或者网络问题，Slave即可接替Master对外服务，并由Slave提升为Master(新主)。典型的多节点共存，但只能同时存在一个主，并且所有节点的状态能统一维护。

## **Bully算法**

MongoDB 用了这个算法来决定副本集中主要的那一个。Bully 算法的主要思想是集群的每个成员都可以声明它是协调者并通知其他节点。别的节点可以选择接受这个声明或是拒绝并进入协调者竞争。被其他所有节点接受的节点才能成为协调者。节点按照一些属性来判断谁应该胜出。这个属性可以是一个静态ID，也可以是更新的度量像最近一次事务ID（最新的节点会胜出）。

使用静态ID作为度量，ID值更大的节点会胜出：



1.最初集群有5个节点，节点5是一个公认的协调者（主）。

2.假设节点5挂了，并且节点2和节点3同时发现了这一情况。两个节点开始竞选并发送竞选消息给ID更大的节点。

3.节点4淘汰了节点2和3，节点3淘汰了节点2。

4.这时候节点1察觉了节点5失效并向所有ID更大的节点发送了竞选信息。

5.节点2、3和4都淘汰了节点1。

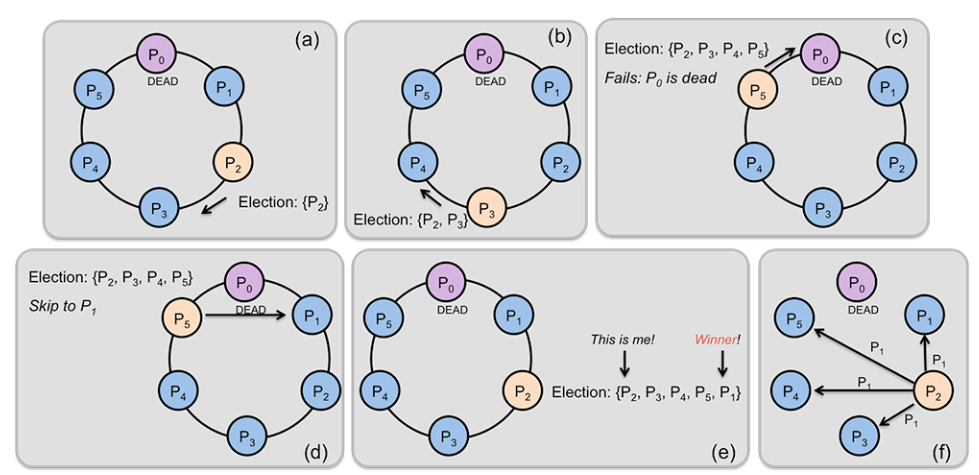
6.节点4发送竞选信息给节点5。

7.节点5没有响应，所以节点4宣布自己当选并向其他节点通告了这一消息。

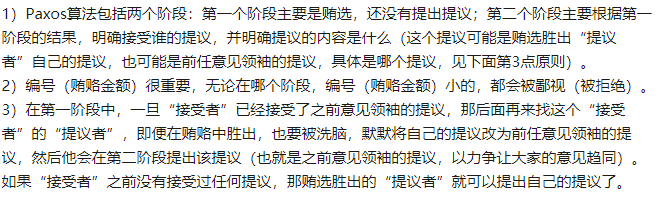
协调者竞选过程会统计参与的节点数目并确保集群中至少一半的节点参与了竞选。这确保了在网络隔离的情况下只有一部分节点能选出协调者（假设网络中网络会被分割成多块区域，之间互不联通，协调者竞选的结果必然会在节点数相对比较多的那个区域中选出协调者，当然前提是那个区域中的可用节点多于集群原有节点数的半数。如果集群被隔离成几个区块，而没有一个区块的节点数多于原有节点总数的一半，那就无法选举出协调者，当然这样的情况下也别指望集群能够继续提供服务了）。

## 环算法

1. 假设进程按照物理或者逻辑顺序进行排序，那么进程都知道它的后继者当任何一个进程注意到协调者不工作时，它构造一个带有自己的进程号的election消息，并将消息发送给后继者。如果后继者崩溃了，发送者沿着此环跳过它的后继者发送给下一个进程，或者再下一个进程。直到找到一个正在运行的进程。
2. 在每一步中，后继的发送者将自己的进程编号也加入到消息中，使自己也成为协调者候选人之一。
3. 最终消息返回到发起这次选举的进程，当发起者收到一条包含自己进程编号的消息时，识别出来。此时消息编程coordinator消息，并再一次绕环运行，向所有进程通知谁是协调者（例如选择成员列表中进程号最大的那个)。



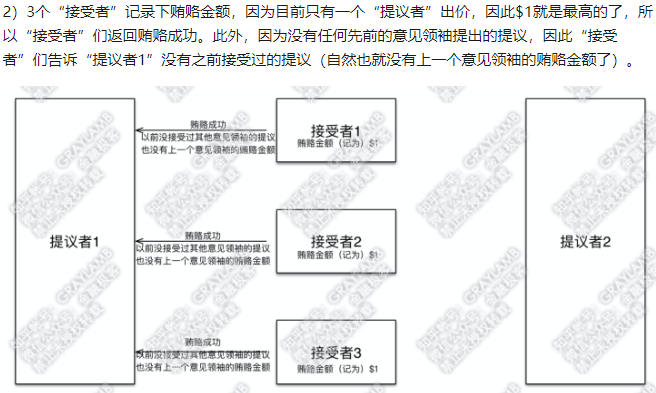
## **Paxos算法**

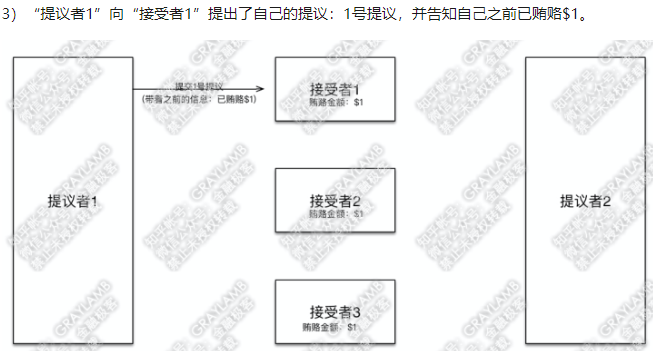


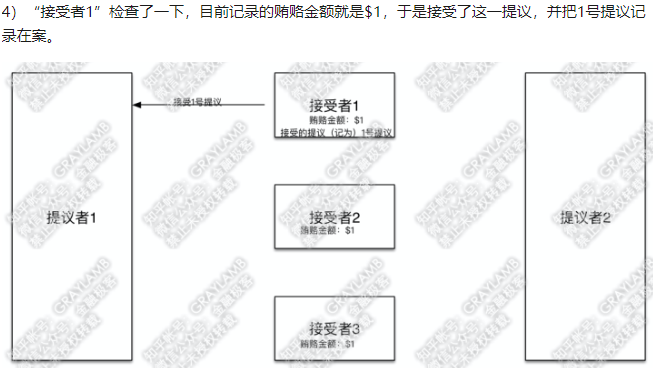
### 例子

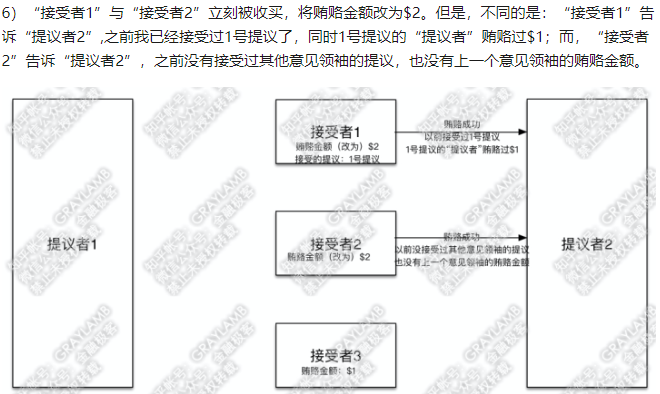
有两个“提议者”和三个“接受者”。

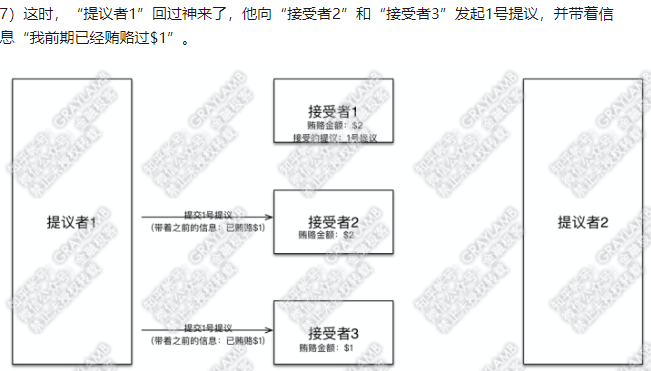


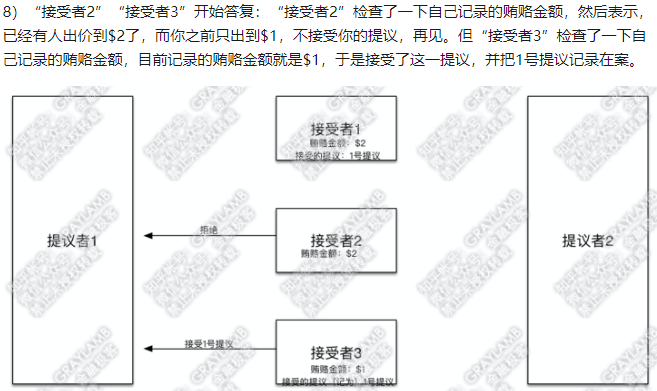




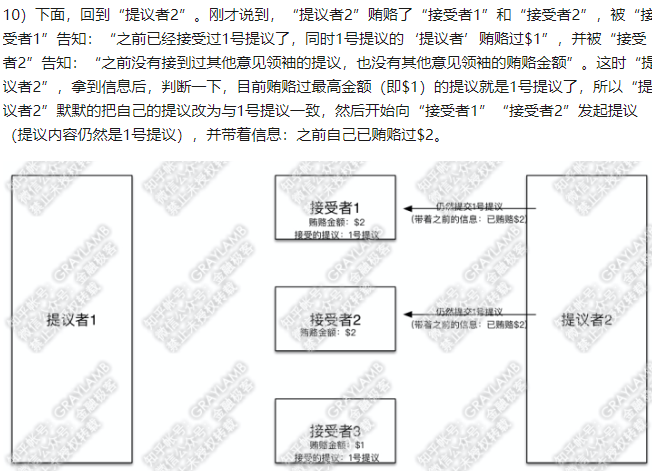














12）于是，“提议者2”也拿到了多数派的意见，最终通过的也是1号提议。

## Zab协议

Zab 协议借鉴了 Paxos 的思想，并进行了改进，以满足工程上的实际需求。

### 设计目标

 一致性

 有序性：有序性是 Zab 协议与 Paxos 协议的一个核心区别。Zab 的有序性主要表现在两个方面：

1. 全局有序：如果消息 a 在消息 b 之前被投递，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 导致了 b），并被一起发送，则 a 始终在 b 之前被执行。

 容错性：有 2f+1 台服务器，只要有大于等于 f+1 台的服务器正常工作，就能完全正常工作。

### 协议内容

广播（boardcast）：Zab 协议中，所有的写请求都由 leader 来处理。正常工作状态下，leader 接收请求并通过广播协议来处理。

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

#### 广播（boardcast）

广播的过程实际上是一个简化的二阶段提交过程：

1.Leader 接收到消息请求后，将消息赋予一个全局唯一的 64 位自增 id，叫做：zxid，通过 zxid 的大小比较即可实现因果有序这一特性。

2.Leader 通过先进先出队列（通过 TCP 协议来实现，以此实现了全局有序这一特性）将带有 zxid 的消息作为一个提案（proposal）分发给所有 follower。

3.当 follower 接收到 proposal，先将 proposal 写到硬盘，写硬盘成功后再向 leader 回一个 ACK。

4.当 leader 接收到合法数量的 ACKs 后，leader 就向所有 follower 发送 COMMIT 命令，同事会在本地执行该消息。

5.当 follower 收到消息的 COMMIT 命令时，就会执行该消息.

**优点：**

相比于完整的二阶段提交，Zab 协议最大的区别就是不能终止事务，follower 要么回 ACK 给 leader，要么抛弃 leader，在某一时刻，leader 的状态与 follower 的状态很可能不一致，因此它不能处理 leader 挂掉的情况，所以 Zab 协议引入了恢复模式来处理这一问题。从另一角度看，正因为 Zab 的广播过程不需要终止事务，也就是说不需要所有 follower 都返回 ACK 才能进行 COMMIT，而是只需要合法数量（2f+1 台服务器中的 f+1 台） 的follower，也提升了整体的性能。

#### 恢复（recovery）

为了使 leader 挂了后系统能正常工作，需要解决以下两个问题：

**1.已经被处理的消息不能丢**

当 leader 收到合法数量 follower 的 ACKs 后，就向各个 follower 广播 COMMIT 命令，同时也会在本地执行 COMMIT 并向连接的客户端返回「成功」。但是如果在各个 follower 在收到 COMMIT 命令前 leader 就挂了，导致剩下的服务器并没有执行都这条消息。为了实现已经被处理的消息不能丢这个目的，Zab 的恢复模式使用了以下的策略：

1.选举拥有 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，即必须有合法数量的服务器的事务日志上有该提案的 proposal，因此，只要有合法数量的节点正常工作，就必然有一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 的消息处理。

3.新的 leader 与 follower 建立先进先出的队列， 先将自身有而 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。

**2. 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 follower 并没有收到此 proposal，因此经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，他保留了被跳过消息的 proposal 状态，与整个系统的状态是不一致的，需要将其删除。

Zab 通过巧妙的设计 zxid 来实现这一目的。一个 zxid 是64位，高 32 是纪元（epoch）编号，每经过一次 leader 选举产生一个新的 leader，新 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 leader 选举后这个值重置为 0。这样设计的好处是旧的 leader 挂了后重启，它不会被选举为 leader，因为此时它的 zxid 肯定小于当前的新 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。