04 | 分布式选举：国不可一日无君

2019-09-30 聂鹏程 来自北京

《分布式技术原理与算法解析》

你好，我是聂鹏程。今天，我来继续带你打卡分布式核心技术。

相信你对集群的概念并不陌生。简单说，集群一般是由两个或两个以上的服务器组建而成，每个服务器都是一个节点。我们经常会听到数据库集群、管理集群等概念，也知道数据库集群提供了读写功能，管理集群提供了管理、故障恢复等功能。

接下来，你开始好奇了，对于一个集群来说，多个节点到底是怎么协同，怎么管理的呢。比如，数据库集群，如何保证写入的数据在每个节点上都一致呢？

也许你会说，这还不简单，选一个“领导”来负责调度和管理其他节点就可以了啊。

这个想法一点儿也没错。这个“领导”，在分布式中叫做主节点，而选“领导”的过程在分布式领域中叫作分布式选举。

然后，你可能还会问，怎么选主呢。那接下来，我们就一起去揭开这个谜底吧。

为什么要有分布式选举?

主节点，在一个分布式集群中负责对其他节点的协调和管理，也就是说，其他节点都必须听从主节点的安排。

主节点的存在，就可以保证其他节点的有序运行，以及数据库集群中的写入数据在每个节点上的一致性。这里的一致性是指，数据在每个集群节点中都是一样的，不存在不同的情况。

当然，如果主故障了，集群就会天下大乱，就好比一个国家的皇帝驾崩了，国家大乱一样。比如，数据库集群中主节点故障后，可能导致每个节点上的数据会不一致。

这，就应了那句话“国不可一日无君”，对应到分布式系统中就是“集群不可一刻无主”。总结来说，选举的作用就是选出一个主节点，由它来协调和管理其他节点，以保证集群有序运行和节点间数据的一致性。

分布式选举的算法

那么，如何在集群中选出一个合适的主呢？这是一个技术活儿，目前常见的选主方法有基于序号选举的算法（ 比如，Bully 算法）、多数派算法（比如，Raft 算法、ZAB 算法）等。接下来，就和我一起来看看这几种算法吧。

长者为大：Bully 算法

Bully 算法是一种霸道的集群选主算法，为什么说是霸道呢？因为它的选举原则是“长者”为大，即在所有活着的节点中，选取 ID 最大的节点作为主节点。

在 Bully 算法中，节点的角色有两种：普通节点和主节点。初始化时，所有节点都是平等的，都是普通节点，并且都有成为主的权利。但是，当选主成功后，有且仅有一个节点成为主节点，其他所有节点都是普通节点。当且仅当主节点故障或与其他节点失去联系后，才会重新选主。

Bully 算法在选举过程中，需要用到以下 3 种消息：

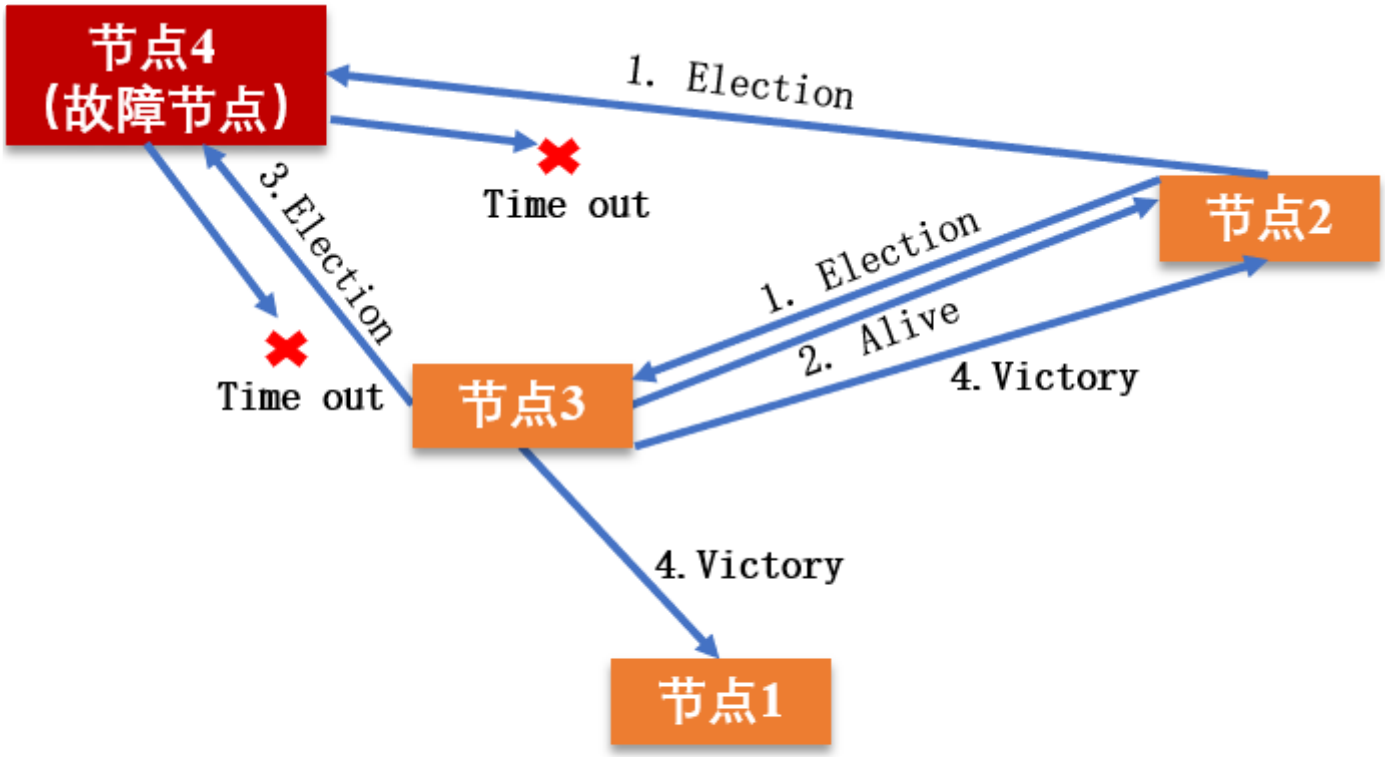
Election 消息，用于发起选举；

Alive 消息，对 Election 消息的应答；

Victory 消息，竞选成功的主节点向其他节点发送的宣誓主权的消息。

Bully 算法选举的原则是“长者为大”，意味着它的**假设条件是，集群中每个节点均知道其他节点的 ID。**在此前提下，其具体的选举过程是：

1. 集群中每个节点判断自己的 ID 是否为当前活着的节点中 ID 最大的，如果是，则直接向其他节点发送 Victory 消息，宣誓自己的主权；
2. 如果自己不是当前活着的节点中 ID 最大的，则向比自己 ID 大的所有节点发送 Election 消息，并等待其他节点的回复；
3. 若在给定的时间范围内，本节点没有收到其他节点回复的 Alive 消息，则认为自己成为主节点，并向其他节点发送 Victory 消息，宣誓自己成为主节点；若接收到来自比自己 ID 大的节点的 Alive 消息，则等待其他节点发送 Victory 消息；
4. 若本节点收到比自己 ID 小的节点发送的 Election 消息，则回复一个 Alive 消息，告知其他节点，我比你大，重新选举。



注：节点i的ID值即为i

目前已经有很多开源软件采用了 Bully 算法进行选主，比如 MongoDB 的副本集故障转移功能。MongoDB 的分布式选举中，采用节点的最后操作时间戳来表示 ID，时间戳最新的节点其 ID 最大，也就是说时间戳最新的、活着的节点是主节点。

小结一下。Bully 算法的选择特别霸道和简单，谁活着且谁的 ID 最大谁就是主节点，其他节点必须无条件服从。这种算法的优点是，选举速度快、算法复杂度低、简单易实现。

但这种算法的缺点在于，需要每个节点有全局的节点信息，因此额外信息存储较多；其次，任意一个比当前主节点 ID 大的新节点或节点故障后恢复加入集群的时候，都可能会触发重新选举，成为新的主节点，如果该节点频繁退出、加入集群，就会导致频繁切主。

民主投票：Raft 算法

Raft 算法是典型的多数派投票选举算法，其选举机制与我们日常生活中的民主投票机制类似，核心思想是“少数服从多数”。也就是说，Raft 算法中，获得投票最多的节点成为主。

采用 Raft 算法选举，集群节点的角色有 3 种：

Leader，即主节点，同一时刻只有一个 Leader，负责协调和管理其他节点；

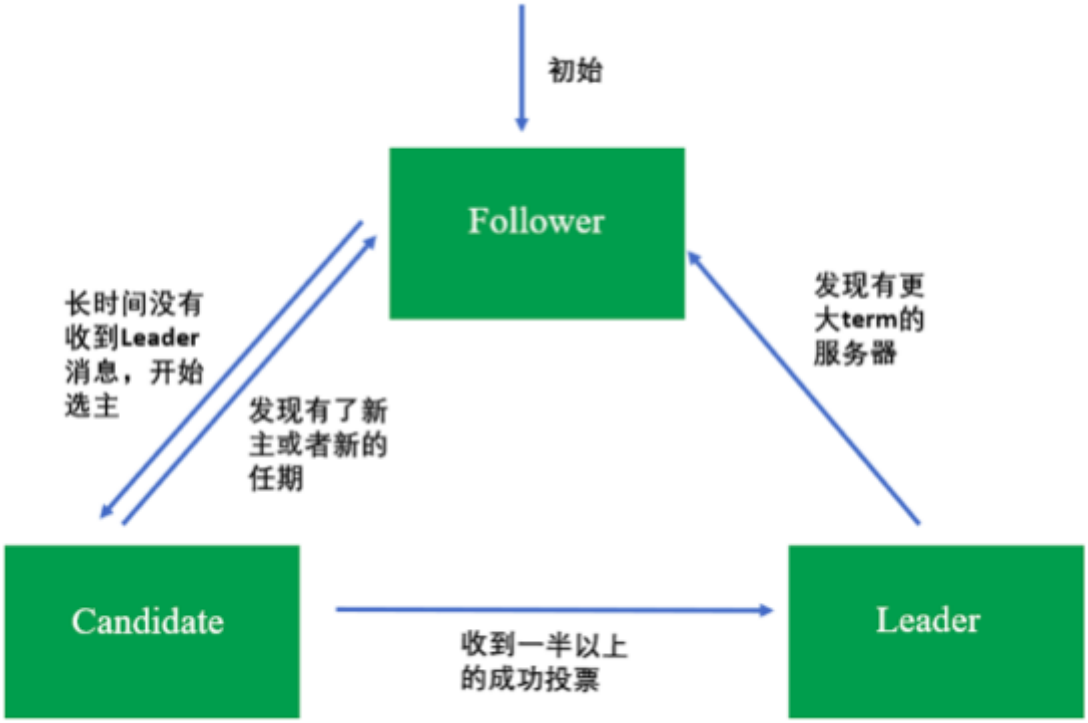
Candidate，即候选者，每一个节点都可以成为 Candidate，节点在该角色下才可以被选为新的 Leader；

Follower，Leader 的跟随者，不可以发起选举。

Raft 选举的流程，可以分为以下几步：

1. 初始化时，所有节点均为 Follower 状态。
2. 开始选主时，所有节点的状态由 Follower 转化为 Candidate，并向其他节点发送选举请求。
3. 其他节点根据接收到的选举请求的先后顺序，回复是否同意成为主。这里需要注意的是，在每一轮选举中，一个节点只能投出一张票。
4. 若发起选举请求的节点获得超过一半的投票，则成为主节点，其状态转化为 Leader，其他节点的状态则由 Candidate 降为 Follower。Leader 节点与 Follower 节点之间会定期发送心跳包，以检测主节点是否活着。
5. 当 Leader 节点的任期到了，即发现其他服务器开始下一轮选主周期时，Leader 节点的状态由 Leader 降级为 Follower，进入新一轮选主。

节点的状态迁移如下所示（图中的 term 指的是选举周期）：



请注意，**每一轮选举，每个节点只能投一次票。**这种选举就类似人大代表选举，正常情况下每个人大代表都有一定的任期，任期到后会触发重新选举，且投票者只能将自己手里唯一的票投给其中一个候选者。对应到 Raft 算法中，选主是周期进行的，包括选主和任值两个时间段，选主阶段对应投票阶段，任值阶段对应节点成为主之后的任期。但也有例外的时候，如果主节点故障，会立马发起选举，重新选出一个主节点。

Google 开源的 Kubernetes，擅长容器管理与调度，为了保证可靠性，通常会部署 3 个节点用于数据备份。这 3 个节点中，有一个会被选为主，其他节点作为备。Kubernetes 的选主采用的是开源的 etcd 组件。而，etcd 的集群管理器 etcds，是一个高可用、强一致性的服务发现存储仓库，就是采用了 Raft 算法来实现选主和一致性的。

小结一下。Raft 算法具有选举速度快、算法复杂度低、易于实现的优点；缺点是，它要求系统内每个节点都可以相互通信，且需要获得过半的投票数才能选主成功，因此通信量大。该算法选举稳定性比 Bully 算法好，这是因为当有新节点加入或节点故障恢复后，会触发选主，但不一定会真正切主，除非新节点或故障后恢复的节点获得投票数过半，才会导致切主。

具有优先级的民主投票：ZAB 算法

ZAB（ZooKeeper Atomic Broadcast）选举算法是为 ZooKeeper 实现分布式协调功能而设计的。相较于 Raft 算法的投票机制，ZAB 算法增加了通过节点 ID 和数据 ID 作为参考进行选主，节点 ID 和数据 ID 越大，表示数据越新，优先成为主。相比较于 Raft 算法，ZAB 算法尽可能保证数据的最新性。所以，ZAB 算法可以说是对 Raft 算法的改进。

使用 ZAB 算法选举时，集群中每个节点拥有 3 种角色：

Leader，主节点；

Follower，跟随者节点；

Observer，观察者，无投票权。

选举过程中，集群中的节点拥有 4 个状态：

Looking 状态，即选举状态。当节点处于该状态时，它会认为当前集群中没有 Leader，因此自己进入选举状态。

Leading 状态，即领导者状态，表示已经选出主，且当前节点为 Leader。

Following 状态，即跟随者状态，集群中已经选出主后，其他非主节点状态更新为 Following，表示对 Leader 的追随。

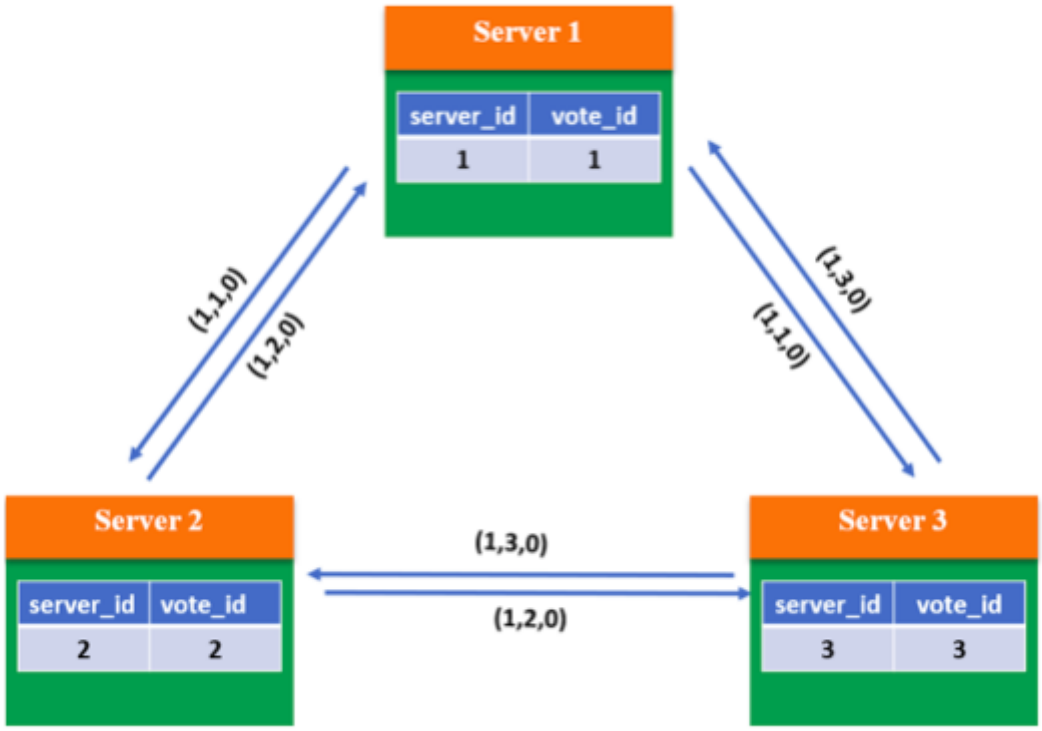
Observing 状态，即观察者状态，表示当前节点为 Observer，持观望态度，没有投票权和选举权。

投票过程中，每个节点都有一个唯一的三元组 (server_id, server_zxID, epoch)，其中 server_id 表示本节点的唯一 ID；server_zxID 表示本节点存放的数据 ID，数据 ID 越大表示数据越新，选举权重越大；epoch 表示当前选取轮数，一般用逻辑时钟表示。

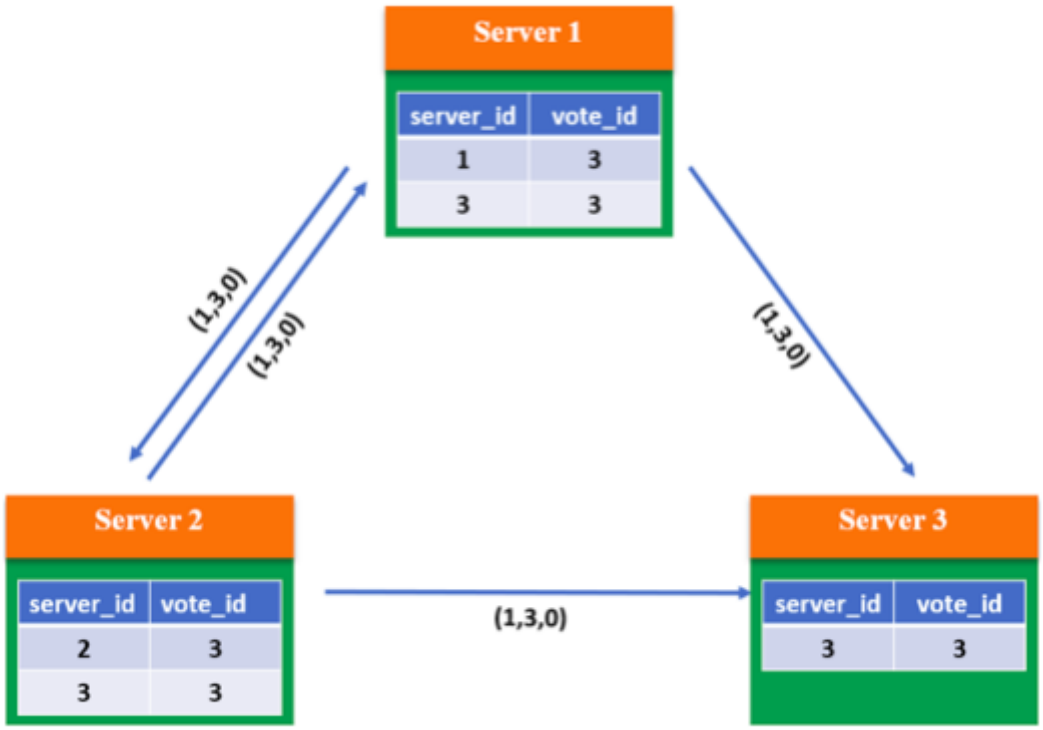
ZAB 选举算法的核心是“少数服从多数，ID 大的节点优先成为主”，因此选举过程中通过 (vote_id, vote_zxID) 来表明投票给哪个节点，其中 vote_id 表示被投票节点的 ID，vote_zxID 表示被投票节点的服务器 zxID。**ZAB 算法选主的原则是：server_zxID 最大者成为 Leader；若 server_zxID 相同，则 server_id 最大者成为 Leader。**

接下来，我以 3 个 Server 的集群为例，此处每个 Server 代表一个节点，与你介绍 ZAB 选主的过程。

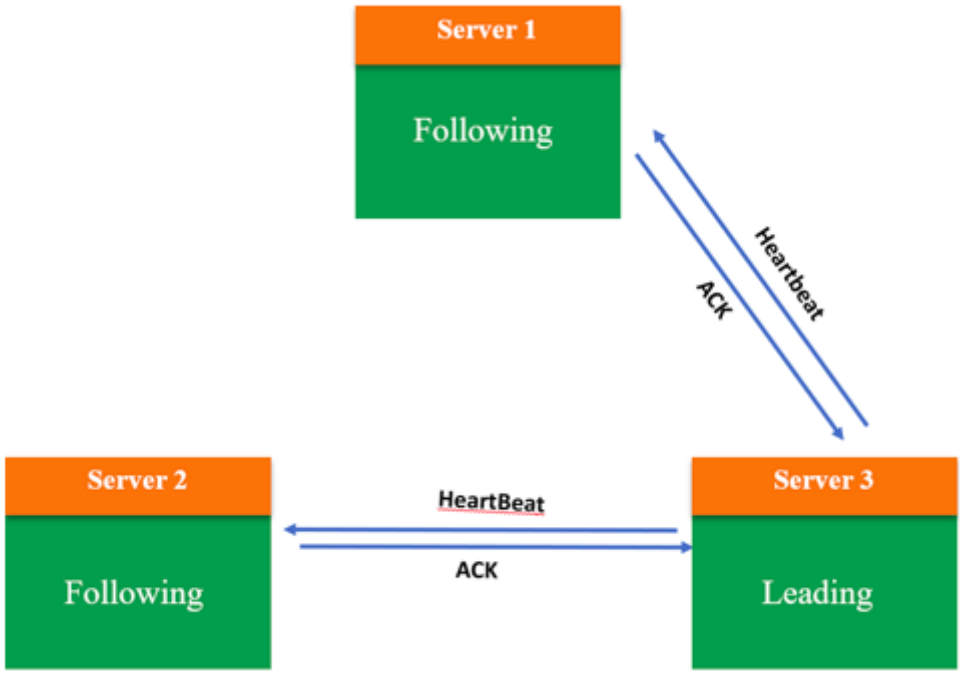
第一步：当系统刚启动时，3 个服务器当前投票均为第一轮投票，即 epoch=1，且 zxID 均为 0。此时每个服务器都推选自己，并将选票信息 <epoch, vote_id, vote_zxID> 广播出去。



第二步：根据判断规则，由于 3 个 Server 的 epoch、zxID 都相同，因此比较 server_id，较大者即为推选对象，因此 Server 1 和 Server 2 将 vote_id 改为 3，更新自己的投票箱并重新广播自己的投票。



第三步：此时系统内所有服务器都推选了 Server 3，因此 Server 3 当选 Leader，处于 Leading 状态，向其他服务器发送心跳包并维护连接；Server1 和 Server2 处于 Following 状态。



小结一下。ZAB 算法性能高，对系统无特殊要求，采用广播方式发送信息，若节点中有 n 个节点，每个节点同时广播，则集群中信息量为 n*(n-1) 个消息，容易出现广播风暴；且除了投

票，还增加了对比节点 ID 和数据 ID，这就意味着还需要知道所有节点的 ID 和数据 ID，所以选举时间相对较长。但该算法选举稳定性比较好，当有新节点加入或节点故障恢复后，会触发选主，但不一定会真正切主，除非新节点或故障后恢复的节点数据 ID 和节点 ID 最大，且获得投票数过半，才会导致切主。

三种选举算法的对比分析

好了，我已经带你理解了分布式选举的 3 种经典算法，即 Bully 算法、Raft 算法和 ZAB 算法。那么接下来，我就从消息传递内容、选举机制和选举过程的维度，对这 3 种算法进行一个对比分析，以帮助你理解记忆。

	Bully算法	Raft算法	ZAB算法
选举消息回复类型	alive消息	同意或不同意选举的消息	投票信息 <epoch, vote_id, vote_zxID>
Leader选举机制	偏向于让ID更大的节点作为Leader	收到过半数的投票，则当选为Leader	倾向于让数据最新或者ID值最大的节点作为Leader
选举过程	只要节点发现Leader无响应时，或者ID较大的节点恢复故障时，就会发起选举	每个角色为Candidate的节点可参与竞选Leader，且每一个Follower只有一次投票权，即同意或者不同意Candidate的选举	每个节点都可以处于Looking状态参与竞选，都可以多次重新投票，根据epoch、zxID、server_id来选择最佳的节点作为Leader
选举所需时间	短	较短	较长
性能	Bully < Raft < ZAB		

知识扩展：为什么“多数派”选主算法通常采用奇数节点，而不是偶数节点呢?

多数派选主算法的核心是少数服从多数，获得投票多的节点胜出。想象一下，如果现在采用偶数节点集群，当两个节点均获得一半投票时，到底应该选谁为主呢?

答案是，在这种情况下，无法选出主，必须重新投票选举。但即使重新投票选举，两个节点拥有相同投票数的概率也会很大。因此，多数派选主算法通常采用奇数节点。

这，也是大家通常看到 ZooKeeper、 etcd、Kubernetes 等开源软件选主均采用奇数节点的一个关键原因。

总结

今天，我首先与你讲述了什么是分布式选举，以及为什么需要分布式选举。然后，我和你介绍了实现分布式选举的 3 种方法，即：Bully 算法、Raft 算法，以及 ZooKeeper 中的 ZAB 算法，并通过实例与你展示了各类方法的选举流程。

我将今天的主要内容总结为了如下所示的思维导图，来帮助你加深理解与记忆。

思考题

1. 分布式选举和一致性的关系是什么？
2. 你是否见到过一个集群中存在双主的场景呢？

我是聂鹏程，感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎你把这篇文章分享给更多的朋友一起阅读。我们下期再会！

精选留言 (77)

1024
2019-09-30

两主的情况出现在集群因为网络原因，被划分了两部分局部可通信的区域。下面的链接详细讲解了Raft算法，及双主出现后集群是如何恢复的。
https://www.infoq.cn/article/coreos-analyse-etcd/
还有一个Raft算法动画链接
http://thesecretlivesofdata.com/raft/#election

共 15 条评论 >　　86

每天晒白牙
2019-09-30

今天这篇文章赚到了
1.分布式选举算法是为了保证数据一致性的
在集群中存在多个节点提供服务，如果每个节点都可以写数据，这样容易造成数据的不一致，所以需要选举一个leader，往leader节点中写数据，然后同步到follower节点中。这样就能更好的保证一致性
但因为同步数据存在延迟，所以follower节点的数据不是每时每刻都和leader节点数据保持一致的
有的框架是由leader提供读写操作，这样能保证写和读都是最新的数据，我没记错的话kafka就属于这种，读写都发生在主副本上。
而像mysql集群是在主节点写入数据，从节点提供读功能，即主从架构

总之，我觉得，一个好的分布式选举算法能更好的保证数据的一致性

2.老师说的集群中存在双主是说选举算法出了问题，出现了两个主，还是说双主是正常情况，两个主互提供写服务，然后再互相同步数据的情况呢?

> 作者回复: 从你对分布式选举的总结可以看出，你很善于思考和总结。关于双主的情况，一般是因为网络故障，比如网络分区等导致的。出现双主的期间，如果双主均提供服务，可能会导致集群中数据不一致。所以，需要根据业务对数据不一致的容忍度来决定是否允许双主情况下提供服务。

共 13 条评论 >　　35

cp★钊
2019-09-30

想问下老师，选举的性能，评判的标准是什么？为什么zab的性能最好，是指哪方面的性能？

共 4 条评论 >

29

游弋云端
2019-09-30

1、分布式选举和一致性的关系是什么？
个人理解选举本身其实就是一致性的一次协商，要求全局认可同一个节点做主节点。选举的目的是为了简化一致性协商的流程，让选出的master来协调各成员针对某项决议达成一致；
2、你是否见到过一个集群中存在双主的场景？
双主是可能发生的，例如原主网络与外部中断，集群发生脑裂，则老的集群主还存在，分裂的剩余节点由于与老主失联，大家重新选了新主出来。此时就会产生双主。规避双主的影响，需要通过租约机制，让老主发现在租约到期后与大多数节点失联主动降备；新主的选举也要等待超过这个租约时间后去选举新主，避免业务同一时刻看到双主。但是由于各个服务器资源、负载、调度等问题，时间并不是一个精确的可靠保障，例如定时器失真，还是可能导致同一时刻出现双主，所以每个地方的租约时间配置是个技术点。另外新主产生，生成新的epoch（+1），这样可以避免大家去处理老消息，从而进一步规避双主的问题。

19

kylexy_0817
2019-10-23

老师，本节为何不提及一下Paxos算法？

共 2 条评论 >

13

清风
2019-10-04

一个问题:如果初始情况下，按照约定，给与奇数节点数，但是选举是这时一个节点挂了？岂不是一定是偶数节点数？只是为了初始选举方便？不考虑故障情况？

9

AllenGFLiu
2019-10-19

老师，在Raft算法中，每个节点只有一票可以投，要么同意要么拒绝，可是节点是基于什么条件作出的判断呢？Bully算法中我看老师又说到是论资排辈的。

共 3 条评论 >

8

钱

2020-02-13

阅过留痕

赞，老师的专栏有两个创新点，一是有一个扩展点，另一个是专栏的总结，特别是以脑图的形式展现。

之前也学习过分布式选举算法，不知道是老师有裁剪还是怎么回事，感觉比这复杂难懂，老师讲解的基本能听懂。

OK，记一下自己的理解。

1：分布式选主算法，为选主而生，那为啥非要有一个主哪？人人平等不好嘛？分布式系统和人类社会非常的像，如果没有主，有些事情特别是有冲突的需要协作完成的，有谁来负责呢？针对数据库，好多机器都存数据，为了提高性能和可用性，如果都能接受写请求，各个库中的数据不一致了，又该怎么办呢？这就是要有主的原因了！

2：选主的算法，老师介绍了三种经典的，已经总结的很好了，我就不总结啦！我来个比喻，方便记忆。

bully算法——类似选武林盟主，谁武功最高，谁来当

raft算法——类似选总统，谁票数最高，谁来当

zab算法——类似选优秀班干部，是班干部且票多才行

感觉只有能确定一台电脑为主就行，不管什么方式，比如：一组两台跑定时任务的集群，刚开始无主，谁先启动谁就是主，当启动一台机器后先看有没有主，有主自己就是从，否则就告诉其他机器自己是主。

作者回复: 赞👍，总结得很到位，加油

7

王喜春

2019-11-21

1. https://github.com/sluk3r/Bully-Algorithm
2. https://github.com/sluk3r/sofa-jraft
3. http://thesecretlivesofdata.com/raft/#election 动画效果。
4. 自己搞一个新的算法的微创新?

6

辉

2020-04-28

Raft哪有说的这么简单。其他候选者发起选举，在leader在的情况下，可以被认为无效

作者回复: 本文主要是针对raft算法进行选主的原理进行介绍，确实raft算法并没有那么简单，比如，在真正实现过程中，心跳检测及超时的限制，以及你说的leader存在的情况下，其他候选者发起选举。对于你说的问题，又可以进一步深挖，什么情况下导致leader存在时，其他候选者发起选举呢？心跳超时还是其他原因？在实践中，具体问题需要具体分析。

5

张理查

2019-12-25

分布式技术原理与算法#Day7

前几天我们看到了分布式集群中都会有一个协调者存在，比如那个厕所叫号员就是这么个角色。那究竟是谁能来做这个协调者或者管理者呢？选出这个负责协调和管理的主节点的过程就是分布式选举。同样我们前面也提到过多次的单点问题，很多时候就是领导者"驾崩"导致的，如何在领导者驾崩后立即再次选出下一任领导者，也是分布式选举中需要十分关注的一个点。

选主的常见原则是基于序号以及基于投票。类似资历优先（岁数大）还是能力优先（票数高）。

基于资历（序号）的常见算法就是Bully（恶霸）算法：就是选取所有节点中，ID最大的节点作为主节点。

既然需要判断自己是不是最大，就需要自己来存储其他所有节点的ID。选举开始后：

1. 宣誓主权：判断自己是不是最大的，如果是就通知各位自己是主了（向所有人发送Victory消息）
2. 认怂：如果自己不是最大的，需要向比自己大的认怂，承认各位大佬江湖地位，并等待回复（向大佬们发送Election）
3. 大佬收到小弟发来的Election消息，就会给小弟发送Alive，表示自己活着。（向小弟发送Alive信息）/
4. 小弟如果没有收到大佬发来的Alive信息，说明比自己大的都死绝了，我可以登基了（向所有人发送Victory消息）

可见其选举速度快、复杂度低、简单易实现。

但所有节点的信息存了所有其他节点的ID，存储的冗余度较高。而且有一个风险，就是ID最大的频繁进出集群，就会频繁切换主节点。还有一个缺点没提到就是ID最大的并不一定是能力最强的。

基于能力，民主一点的可以是Raft算法：核心思想是"少数服从多数"。

Raft算法规定了包括Leader主节点、Candidate 候选节点以及Follower跟随节点：

1. 开始选举时，大家都会变成候选节点，向其他节点发送选举请求
2. 每个节点根据先后顺序回复是否同意
3. 若同意数超过一半，则成为主节点，其他节点降为跟随节点，并与主节点互发心跳包
4. 主节点失效活着任期结束再来一轮选举

可见这种方式选举速度快、复杂度低、易于实现。
但选举时的通信量是n * (n -1)。Raft如果ID最大的频繁进出集群，虽然会触发选主，但不一定真正切主。除非得票数过半。
基于能力，还有一种算法是ZAB算法：核心思想是“少数服从多数+ID大的优先”
ZAB是ZK实现协调所设计的。偏向于数据更新的节点。ZAB添加了server_zxID 来表示本节点存放的数据ID，越大越新（能力越强）。选举过程与Raft算法类似，只不过Raft根据先后顺序来判断，而ZAB先比较数据ID，数据ID相同的再比较ServerID，也就是说ZAB喜欢“能力强的年轻人”。
但同样也有广播风暴的问题，且增加了一点选举的复杂度，但会使得主节点更加稳定，因为新节点增加或故障节点（数据不会太新）恢复，触发选举后切主的可能性要更小一些。
因此你看后两种投票少数服从多数的情况下，最好还必须候选者是单数，否则可能因为票数相同而需要多次重新投票。所以经常看到ZK、etcd、K8S都会建议奇数节点。

共 1 条评论 > 4

Mr.Brooks
2019-10-31

raft算法选举的时候每一轮只有一个选票，这个选票是如何确定投给哪一个节点呢?

共 2 条评论 > 3

Lugyedo
2019-10-01

奇数个节点的集群当一个节点故障时会变成偶数个节点吧，这个时候“多数派”算法怎么选主

共 4 条评论 > 3

Will
2019-09-30

问下老师，Bully 和 ZAB 都是根据 ID 的大小投票，那 Raft 算法选举时的投票依据是什么？是随机投票么，如果是随机投的话，奇数节点好像也并不能保证投票结果不会出现同票的情况啊?
希望老是解答一下

共 3 条评论 > 3

随心而至
2019-09-30

1.分布式选举和一致性，感觉是密不可分的。重新选举依靠一致性提供的数据，一致性又要依靠选举出来的主节点进行。这里我只了解过raft算法
https://www.cnblogs.com/xybaby/p/10124083.html

2.有个brain split（脑裂），比如说两个机房原来网络想通，可以正确选主，后来网络不通，每个机房都只知道自己的小山头，他们就容易各自占山为王。
http://www.ruanyifeng.com/blog/2018/07/cap.html
也可以搜下维基百科brain split。

在地铁上写的，有不对之处，请老师指出

共 1 条评论 >　　3

毛小驴的日常
2021-02-27

对于选主，也有一种分类角度
1. 投票（选举）类：raft
2. 竞争（证明）类：PoX
3. 随机（指定）类：PBFT，PoET，Bully
4. 联盟（代议）类：DPOS
5. 混合类：如 PoW+PoS等等

1

南国
2020-04-09

学到了很多知识呀!
1:与一致性的关系就是分布式选举可以在一些场景下决定一致性,比如raft中就是在客户端向leader写入信息.leader在收到大多数follower的回复后视为写入成功,这就是保证了最终一致性,如果改为收到全部节点的回复的话就是强一致性了.之所以说是一些场景是因为选举其实并不一定发生在选一个主节点去处理客户端请求,也可以是选举进行故障转移等.例子是redis哨兵或者redis集群的故障转移需要选一个leader进行操作.
2.双主的情况发生在网络分区中,这其实就是算法可用性的一个体现,可以容忍N/2个节点失效(奇数),因为在那个较少节点网络分区中的节点写的数据不会被视为成功,节点数不可能大于一半,自然也不会收到一半的回复了从而视为写入成功了.这也是raft的过程.当然这里的双主说的是选举出来的主.复制模型中我们还可以使用多主节点模型,这样主节点不就有很多啦(偷换概念小能手)

还有一点很有意思也想提一下,这里的ZAB算法让我想起了redis集群中在选举选出来一个leader执行故障转移的时还需要选择一个从节点,使用的就是简化版ZAB(不需要通信),因为心跳包的原因所有主节点中有宕机主节点从节点的信息,直接选一个数据最新ID最大的即可!

1

鸭先知
2020-03-29

核心是为了数据一致性，分布式选举为数据一致性提供了保障；网络分区会导致脑裂

作者回复: 是的，数据一致性是分布式领域中一个非常重要的问题。

1

Lane
2020-03-09

双主是脑裂吧

作者回复: 双主是网络分区导致的，是脑裂的一种情况。

1

Joe Black
2019-11-18

raft算法中每个节点都可以参与选举，也可以发起选举，当有多个节点发起时候，收到消息的节点如何决定投票给谁？或者说自己也是发起投票的节点，但是收到了其它节点的发起投票请求，那么自己是投还是不投呢？

作者回复: 投票的原则通常采用谁的请求先到，就投票给谁。

共 2 条评论 >

1