文 2PC到3PC到Paxos到Raft到ISR (/a/1190000004474543)

xixicat (https://segmentfault.com/u/xixicat) 6 天前发布

> 分布式理论系列

- 从ACID到CAP到BASE (https://segmentfault.com/a/1190000004468442)
- 2PC到3PC到Paxos到Raft到ISR (https://segmentfault.com/a/1190000004474543)
- 复制、分片和路由 (https://segmentfault.com/a/1190000004485355)
- 副本更新策略 (https://segmentfault.com/a/1190000004480546)
- 负载均衡算法及手段 (https://segmentfault.com/a/1190000004492447)

序

本文主要讲述2PC及3PC，以及Paxos以及Raft协议。

两类一致性(操作原子性与副本一致性)

- 2PC协议用于保证属于多个数据分片上的操作的原子性。这些数据分片可能分布在不同的服务器上，2PC协议保证多台服务器上的操作要么全部成功，要么全部失败。
- Paxos协议用于保证同一个数据分片的多个副本之间的数据一致性。当这些副本分布到不同的数据中心时，这个需求尤其强烈。

一、2PC（阻塞、数据不一致问题、单点问题）

`Two-Phase Commit`，两阶段提交

1、阶段一：提交事务请求（投票阶段）

（1）事务询问

协调者向所有的参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者的响应

（2）执行事务

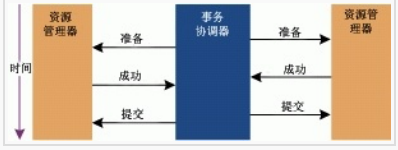
各参与者节点执行事务操作，并将`Undo`和`Redo`信息计入事务日志中

（3）各参与者向协调者反馈事务询问的响应

如果参与者成功执行了事务操作，那么就反馈给协调者`Yes`响应，表示事务可以执行；如果参与者没有成功执行事务，那么就反馈给协调者`No`响应，表示事务不可以执行。

2、阶段二：执行事务提交（执行阶段）

（1）执行事务提交



如果所有参与者的反馈都是`Yes`响应，那么

- A、发送提交请求

  协调者向所有参与者节点发出`Commit`请求

- B、事务提交

  参与者接收到`Commit`请求后，会正式执行事务提交操作，并在完成提交之后释放在整个事务执行期间占用的事务资源

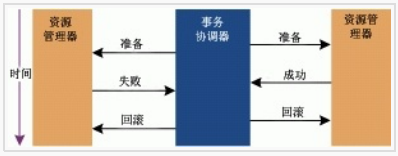
- C、反馈事务提交结果

  参与者在完成事务提交之后，向协调者发送ACK信息

- D、完成事务

  协调者接收到所有参与者反馈的ACK消息后，完成事务

（2）中断事务



任何一个参与者反馈了No响应，或者在等待超时之后，协调者尚无法接收到所有参与者的反馈响应，那么就会中断事务。

- A、发送回滚请求

  协调者向所有参与者节点发出Rollback请求

- B、事务回滚

  参与者接收到rollback请求后，会利用其在阶段一中记录的Undo信息来执行事务回滚操作，并在完成回滚之后释放整个事务执行期间占用的资源

- C、反馈事务回滚结果

  参与者在完成事务回滚之后，向协调者发送ACK信息

- D、中断事务

  协调者接收到所有参与者反馈的ACK信息后，完成事务中断

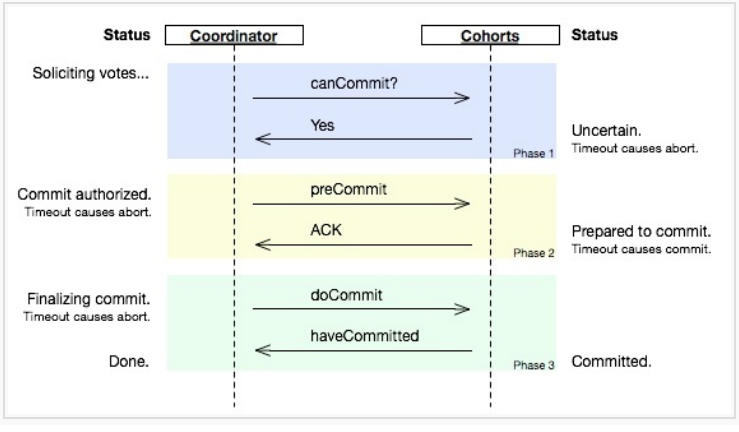
优缺点

优点：原理简单、实现方便
缺点：同步阻塞、单点问题、数据不一致、太过保守

- （1）同步阻塞
  同步阻塞会极大地限制分布式系统的性能。在二阶段提交的执行过程中，所有参与该事务操作的逻辑都处于阻塞状态，各个参与者在等待其他参与者响应的过程中，将无法进行其他任何操作。
- （2）单点问题
  一旦协调者出现问题，那么整个二阶段提交流程将无法运转，更为严重的是，如果是在阶段二中出现问题，那么其他参与者将会一直处于锁定事务资源的状态中，无法继续完成事务操作。
- （3）数据不一致
  在阶段二，当协调者向所有参与者发送commit请求之后，发生了局部网络异常或协调者在尚未发完commit请求之前自身发生了崩溃，导致最终只有部分参与者接收到了commit请求，于是这部分参与者执行事务提交，而没收到commit请求的参与者则无法进行事务提交，于是整个分布式系统出现了数据不一致性现象。
- （4）太过保守
  如果参与者在与协调者通信期间出现故障，协调者只能靠超时机制来判断是否需要中断事务，这个策略比较保守，需要更为完善的容错机制，任意一个节点的失败都会导致整个事务的失败。

二、3PC（解决2PC的阻塞，但还是可能造成数据不一致）

Three-Phase Commit，三阶段提交，分为CanCommit、PreCommit、do Commit三个阶段。



为了避免在通知所有参与者提交事务时，其中一个参与者crash不一致时，就出现了三阶段提交的方式。三阶段提交在两阶段提交的基础上增加了一个preCommit的过程，当所有参与者收到preCommit后，并不执行动作，直到收到commit或超过一定时间后才完成操作。

1、阶段一CanCommit

- （1）事务询问
  协调者向各参与者发送CanCommit的请求，询问是否可以执行事务提交操作，并开始等待各参与者的响应
- （2）参与者向协调者反馈询问的响应
  参与者收到CanCommit请求后，正常情况下，如果自身认为可以顺利执行事务，那么会反馈Yes响应，并进入预备状态，否则反馈No。

2、阶段二PreCommit

（1）执行事务预提交

```
如果协调者接收到各参与者反馈都是Yes，那么执行事务预提交
```

- A、发送预提交请求
  协调者向各参与者发送preCommit请求，并进入prepared阶段
- B、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信息记录到事务日记中
- C、各参与者向协调者反馈事务执行的响应
  如果各参与者都成功执行了事务操作，那么反馈给协调者Ack响应，同时等待最终指令，提交commit或者终止abort

（2）中断事务

如果任何一个参与者向协调者反馈了No响应，或者在等待超时后，协调者无法接收到所有参与者的反馈，那么就会中断事务。

- A、发送中断请求

  ```
  协调者向所有参与者发送abort请求
  ```

- B、中断事务

  ```
  无论是收到来自协调者的abort请求，还是等待超时，参与者都中断事务
  ```

3、阶段三doCommit

（1）执行提交

- A、发送提交请求
  假设协调者正常工作，接收到了所有参与者的ack响应，那么它将从预提交阶段进入提交状态，并向所有参与者发送doCommit请求
- B、事务提交
  参与者收到doCommit请求后，正式提交事务，并在完成事务提交后释放占用的资源
- C、反馈事务提交结果
  参与者完成事务提交后，向协调者发送ACK信息
- D、完成事务
  协调者接收到所有参与者ack信息，完成事务

（2）中断事务

假设协调者正常工作，并且有任一参与者反馈No，或者在等待超时后无法接收所有参与者的反馈，都会中断事务

- A、发送中断请求
  协调者向所有参与者节点发送abort请求
- B、事务回滚
  参与者接收到abort请求后，利用undo日志执行事务回滚，并在完成事务回滚后释放占用的资源
- C、反馈事务回滚结果
  参与者在完成事务回滚之后，向协调者发送ack信息
- D、中断事务
  协调者接收到所有参与者反馈的ack信息后，中断事务。

阶段三可能出现的问题：
协调者出现问题、协调者与参与者之间网络出现故障。不论出现哪种情况，最终都会导致参与者无法及时接收到来自协调者的doCommit或是abort请求，针对这种情况，参与者都会在等待超时后，继续进行事务提交（timeout后中断事务）。

优点：降低参与者阻塞范围，并能够在出现单点故障后继续达成一致
缺点：引入preCommit阶段，在这个阶段如果出现网络分区，协调者无法与参与者正常通信，参与者依然会进行事务提交，造成数据不一致。

三、Paxos（解决单点问题）

基于消息传递且具有高度容错性的一致性算法。Paxos算法要解决的问题就是如何在可能发生几起宕机或网络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。

拜占庭问题：消息不完整或者被篡改。Paxos在维持领导者选举或者变量修改一致性上，采取一种类似议会投票的过半同意机制，比如设定一个领导者，需要将此看做一个议案，征求过半同意，每个节点通过一个议案会有编号记录，再次收到此领导者的不同人选，发现已经有编号记录便驳回，最后以多数通过的结果为准。

我们举个简单的例子，来阐述一下Paxos的基本思想：假设我们有5台计算机A、B、C、D、E，每台计算机保存着公司CEO的信息，现在CEO任期到了，需要进行新一界选举了。

A计算机发起一个选举议案，提议CEO为“张三”，如果没有其他候选人议案，也没有网络问题，只要其中半数以上计算机收到并通过议案，那么最终“张三”当选CEO。由于是分布式环境，并发请求、机器故障、网络故障等问题是常态，如果A和E同时提交选举议案，A提名“张三”，E提名“李四”，那么肯定会涉及多计算机的一致性问题了：假设A、B、C先收到A的议案，D、E先收到E的议案，那么A继续提交给D时，D告诉它已经先收到E的议案了，因此驳回了A的请求。同样E继续提交给A、B、C时也碰到相同的问题。

我们可以通过“在每台计算机同时接受议案提交时设置一个编号，编号先的通过，编号后的驳回”的方式来实现。议案提交上去后，发现A、B、C投票“张三”为CEO，D、E投票“李四”为CEO，少数服从多数，因此最后结果为“张三”当选CEO。

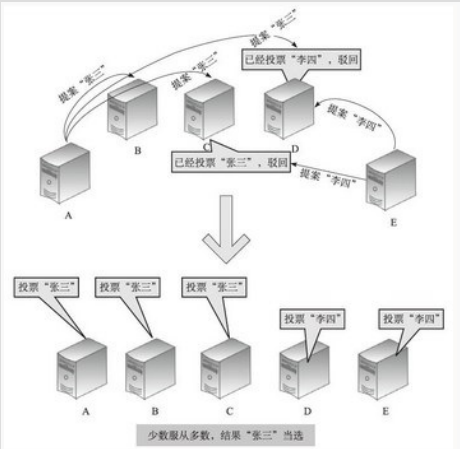


图 3-2 Paxos 算法示意图

如果是C计算机发生了网络问题或者故障，双方投票相同，那么选举无法完成。

如果C计算机发生了网络问题或者故障，A、B、D投票“张三”，E投票“李四”，那么结果为“张三”当选，而C对于这些情况一无所知，但是当C计算机恢复正常时，他会发起一个“询问谁是CEO”的议案获取最新信息。简言之，Paxos对每个节点的并发修改采取编号记录的方式保持一致性，对多个节点的并发修改采取少数服从多数的方式保持一致性。Paxos有点类似分布式二阶段提交方式，但是又不同，二阶段提交不能是多数节点同意，必须是全部同意。为了遵守过半节点同意的约束，Paxos算法往往要求节点总数为奇数。

Paxos 算法解决的问题是在一个可能发生上述异常的分布式系统中如何就某个值达成一致，保证不论发生以上任何异常，都不会破坏决议的一致性。一个典型的场景是，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。一个通用的一致性算法可以应用在许多场景中，是分布式计算中的重要问题。从20世纪80年代起对于一致性算法的研究就没有停止过。

简单说来，Paxos的目的是让整个集群的结点对某个值的变更达成一致。Paxos算法基本上来说是个民主选举的算法——大多数的决定会成个整个集群的统一决定。任何一个点都可以提出要修改某个数据的提案，是否通过这个提案取决于这个集群中是否有超过半数的结点同意（所以Paxos算法需要集群中的结点是单数）。

这个算法有两个阶段（假设这个有三个结点：A，B，C）：

第一阶段：Prepare阶段

A把申请修改的请求Prepare Request发给所有的结点A，B，C。注意，Paxos算法会有一个Sequence Number（你可以认为是一个提案号，这个数不断递增，而且是唯一的，也就是说A和B不可能有相同的提案号），这个提案号会和修改请求一同发出，任何结点在“Prepare阶段”时都会拒绝其值小于当前提案号的请求。所以，结点A在向所有结点申请修改请求的时候，需要带一个提案号，越新的提案，这个提案号就越是是最大的。

如果接收结点收到的提案号n大于其它结点发过来的提案号，这个结点会回应Yes（本结点上最新的被批准提案号），并保证不接收其它<n的提案。这样一来，结点上在Prepare阶段里总是会对最新的提案做承诺。

优化：在上述 prepare 过程中，如果任何一个结点发现存在一个更高编号的提案，则需要通知 提案人，提醒其中断这次提案。

第二阶段：Accept阶段

如果提案者A收到了超过半数的结点返回的Yes，然后他就会向所有的结点发布Accept Request（同样，需要带上提案号n），如果没有超过半数的话，那就返回失败。

当结点们收到了Accept Request后，如果对于接收的结点来说，n是最大的了，那么，它就会通过request（修改这个值），如果发现自己有一个更大的提案号，那么，结点就会拒绝request（拒绝修改）。

我们可以看以，这似乎就是一个“两段提交”的优化。其实，2PC/3PC都是分布式一致性算法的残次版本，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。

我们还可以看到：对于同一个值的在不同结点的修改提案就算是在接收方被乱序收到也是没有问题的。

四、Raft协议(解决paxos的实现难度)

Paxos 相比 Raft 比较复杂和难以理解。角色扮演和流程比 Raft 都要啰嗦。比如 Agreement 这个流程，在 Paxos 里边：Client 发起请求举荐 Proposer 成为 Leader，Proposer 然后向全局 Acceptors 寻求确认，Acceptors 全部同意 Proposer 后，Proposer 的 Leader 地位得已承认，Acceptors 还得再向Learners 进行全局广播来同步。而在 Raft 里边，只有 Follower/Candidate/Leader 三种角色，角色本身代表状态，角色之间进行状态转移是一件非常自由民主的事情。Raft虽然有角色之分但是是全民参与进行选举的模式；但是在Paxos里边，感

觉更像议员参政模式。

三个角色

follower、candidate、leader。
最开始大家都是follower，当follower监听不到leader，就可以自己成为candidate，发起投票

leader选举：timeout限制

选举的timeout

follower成为candidate的超时时间，每个follower都在150ms and 300ms之间随机，之后看谁先timeout，谁就先成为candidate，然后它会先投自己一票，再向其他节点发起投票邀请。
如果其他节点在这轮选举还没有投过票，那么就给candidate投票，然后重置自己的选举timeout。
如果得到大多数的投票就成为leader，之后定期开始向follower发送心跳。

如果两个follower同时成为candidate的话，如果最后得到的票数相同，则等待其他follower的选择timeout之后成为candidate，继续开始新一轮的选举。

log复制

leader把变动的log借助心跳同步给follower，过半回复之后才成功提交，之后再下一次心跳之后，follower也commit变动，在自己的node上生效。

分裂之后，另一个分区的follower接受不到leader的timeout，然后会有一个先timeout，成为candidate，最后成为leader。
于是两个分区就有了两个leader。

当客户端有变动时，其中的leader由于无法收到过半的提交，则保持未提交状态。有的leader的修改，可以得到过半的提交，则可以修改生效。

当分裂恢复之后，leader开始对比选举的term，发现有更高的term存在时，他们会撤销未提交的修改，然后以最新的为准。

五、ISR的机制(解决f容错的2f+1成本问题)

Kafka并没有使用Zab或Paxos协议的多数投票机制来保证主备数据的一致性，而是提出了ISR的机制（In-Sync Replicas）的机制来保证数据一致性。

ISR认为对于2f+1个副本来说，多数投票机制要求最多只能允许f个副本发生故障，如果要支持2个副本的容错，则需要至少维持5个副本，对于消息系统的场景来说，效率太低。

ISR的运行机制如下：将所有次级副本数据分到两个集合，其中一个被称为ISR集合，这个集合备份数据的特点是即时和主副本数据保持一致，而另外一个集合的备份数据允许其消息队列落后于主副本的数据。在做主备切换时，只允许从ISR集合中选择主副本，只有ISR集合内所有备份都写成功才能认为这次写入操作成功。在具体实现时，kafka利用zookeeper来保持每个ISR集合的信息，当ISR集合内成员变化时，相关构件也便于通知。通过这种方式，如果设定ISR集合大小为f+1，那么可以最多允许f个副本故障，而对于多数投票机制来说，则需要2f+1个副本才能达到相同的容错性。

参考

- 分布式系统的Raft算法 (http://www.jdon.com/artichect/raft.html)
- 英文动画演示Raft (http://thesecretlivesofdata.com/raft/)(推荐)
- paxos-by-example (https://angus.nyc/2012/paxos-by-example/)
- 为啥CoreOS没用Paxos，重新搞了Raft？ (http://chuansong.me/n/1019861)

6 天前发布 (/a/1190000004474543)

你可能感兴趣的文章

说说分布式事务(一) (https://segmentfault.com/a/1190000004474144) 74 浏览

kbengine开源分布式游戏服务端引擎 (https://segmentfault.com/a/1190000000663501) 2 收藏，971 浏览

理解 CAP 理论 - 分布式数据库相关理论 Part1 (https://segmentfault.com/a/1190000002802523) 6 收藏，980 浏览

关注专栏

分享扩散：

•••