Zookeeper2 一致性协议

> 为了解决分布式一致性问题，最著名的就是二阶段提交协议、三阶段提交协议和Paxos算法了。

2PC

`2PC`（`Two-Phase Commit`），即二阶段提交。二阶段提交协议也被认为是一种一致性协议，用来保证分布式系统数据的一致性。目前，绝大多数的关系型数据库都是采用二阶段提交协议来完成分布式事务处理的，因此二阶段提交协议被广泛地应用在许多分布式系统中。

协议说明

阶段一：提交事务请求

> 被称为"投票过程"，即各参与者投票表明是否要继续执行接下去的事务提交操作。

- 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，并开始等待参与者的响应

- 执行事务

  > 各参与者节点执行事务操作，并将Undo和Redo信息记入事务日记中

- 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。

阶段二：执行事务提交

> 协调者会根据各参与者的反馈情况来决定最终是否可以进行事务提交操作，正常情况下，包含以下两种可能

- 执行事务提交

  > 加入协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务提交。

  - 发送提交请求

    > 协调者向所有参与者节点发出Commit请求。

  - 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。

  - 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消息

- 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

- 中断事务

> 加入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，协调者尚无法接收到所有参与者的反馈响应，那么就会中断事务。

- 发送回滚请求

> 协调者向所参与者节点发出Rollback请求

- 事务回滚

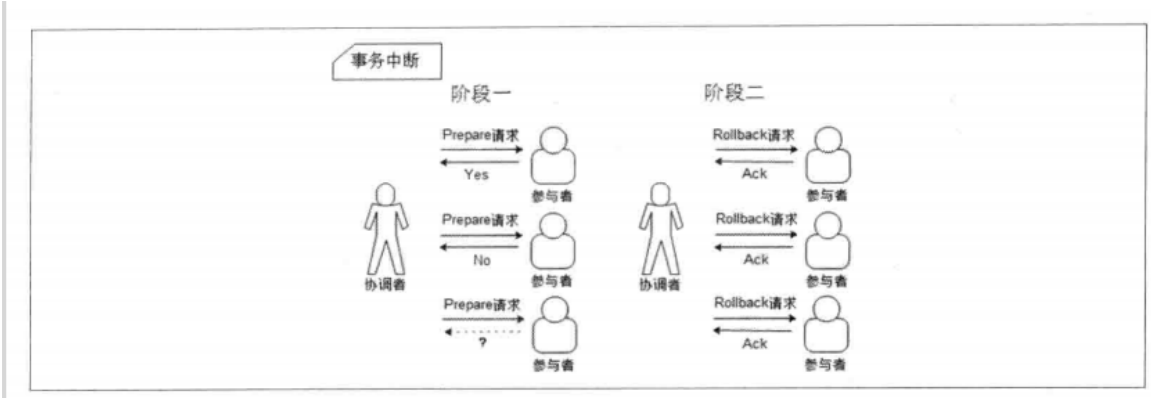
> 参与者接收到Rollback请求后，会利用其在阶段一中记录的Undo信息来执行事务回滚操作，并在完成回滚之后释放在整个事务执行期间占用的资源

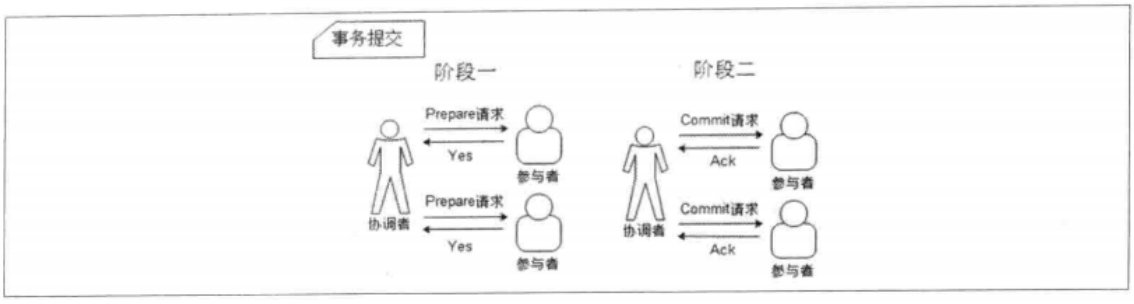
- 反馈事务回滚结果

> 参与者在完成事务回滚之后，向协调者发送Ack信息

- 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：

1. 原理简单
2. 实现方便

缺点:

1. 同步阻塞
2. 单点问题
3. 脑裂
4. 太过保守

- 同步阻塞

最大的问题就是同步阻塞，会极大限制分布式系统的性能，在二阶段提交的执行过程，所有参与该事务操作的逻辑都处于阻塞状态，将无法进行其他任何操作。

- 单点问题

协调者的角色在整个二阶段提交协议中起到了非常大的作用，一旦协调者出现了问题，那么整个二阶段提交流程就没办法允许。更严重的是，如果协调者是在阶段二中出现了问题的话，那么其他参与者将会一直处于锁定事务资源的状态，无法完成其他操作。

- 数据不一致

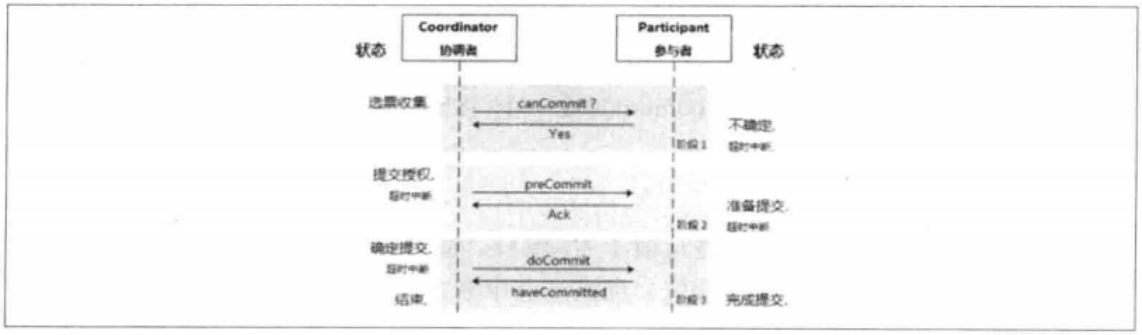
当协调者向所有参与者发送了`Commit`请求后，发生了局部网络异常或者是协调者在尚未发送完`Commit`请求之前自身发生了崩溃，导致只有部分参与者接收到了`Commit`请求，就会出现数据不一致情况。

- 太过保守

在进行事务提交询问的过程中，参与者出现故障而导致协调者无法获取到所有参与者的响应信息。这个时候协调者要自己来判断是否中断事务。

3PC

> 3PC，是Three-Phase Commit 的缩写，即三阶段提交，是2PC的改进版，其将二阶段提交协议的“提交事务请求”过程一分为二，变成了 CanCommit、PreCommit、和do Commit三个阶段组成的事务处理过程。



阶段一： CanCommit

- 事务询问

协调者向所有参与者发送一个包含事务内容的canCommit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

- 各参与者向协调者反馈事务查询的响应

参与者在接收到来自协调者的canCommit请求后，正常情况下，如果其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响应

阶段二： PreCommit

> 协调者会根据各参与者的反馈情况来决定是否可以进行事务的PreCommit操作，正常情况下，包含两种可能

- 执行事务预提交

加入协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务提交。

  - 发送预提交请求

协调者向所有参与者发送出preCommit请求，并进入Prepared阶段

  - 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

  - 各参与者向协调者反馈事务执行的响应

如果参与者成功执行了事务擦欧总，那么就会反馈给协调者Ack响应，同时等待最终的指令：提交或中止

- 中断事务

接入任何一个参与者协调者反馈No响应，或者等到超时后，协调者无法接收到所有参与者的反馈响应，那么就会中断事务。

  - 发送中断请求

协调者向参与者节点发出abort请求

  - 中断事务

无论是收到来自协调者的abort请求，或者是等待协调者请求过程中出现超时，参与者都会中断事务

阶段三：doCommit

- 执行提交
  - 发送提交请求

协调者向所有参与者节点发出doCommit请求。

- 事务提交

  参与者接收到doCommit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。

- 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消息

- 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

- 中断事务

  加入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，协调者尚无法接收到所有参与者的反馈响应，那么就会中断事务。

  - 发送回滚请求

    协调者向所参与者节点发出abort请求

  - 事务回滚

    参与者接收到abort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在整个事务执行期间占用的资源

  - 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

  - 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。

需要注意的是，一旦进入了阶段三，可能会存在以下两种故障：

- 协调者出现问题
- 协调者和参与者之间的网络出现故障

优缺点

优点：

```
1. 降低了参与者阻塞范围
2. 出现单点故障后继续达成一致
```

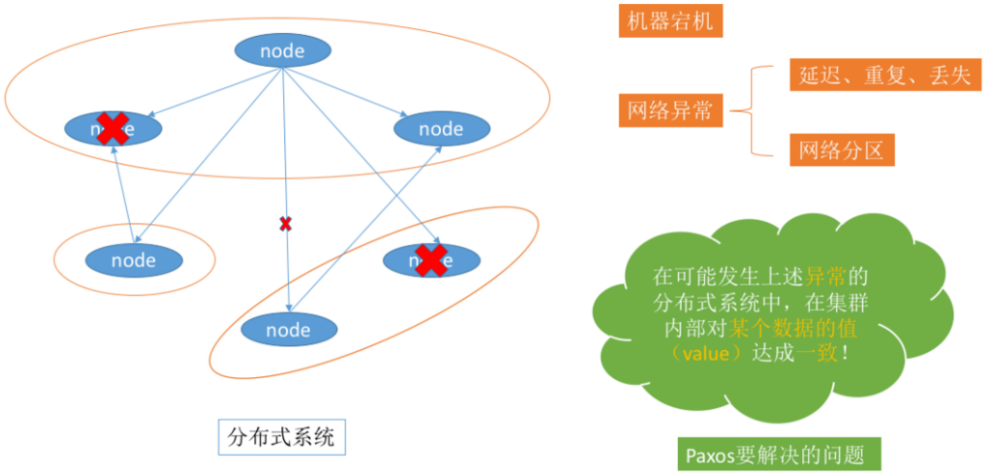
缺点：

```
1. 参与者接收到preCommit消息的时候，如果网络出现分区，此时协调者所在的节点和参与者都无法进行正常的网络通信，会出现数据的不一致性。
```

Paxos算法

> Paxos算法是基于，是目前公认的解决分布式一致性问题最有效的算法之一，其解决的问题就是在分布式系统中如何就某个值（决议）达成一致。

Paxos算法就是解决这种分布式场景中的一致性问题。对于一般的开发人员来说，**只需要知道paxos是一个分布式选举算法即可。**多个节点之间存在两种通讯模型：共享内存（Shared memory）、消息传递（Messages passing），Paxos是基于消息传递的通讯模型的。



前提假设

信道是安全的（信道可靠），发出的信号不会被篡改，因为Paxos算法是基于消息传递的。

算法概念

- 角色

在Paxos算法中，有三种角色：
1. Proposer
2. Acceptor
3. Learners

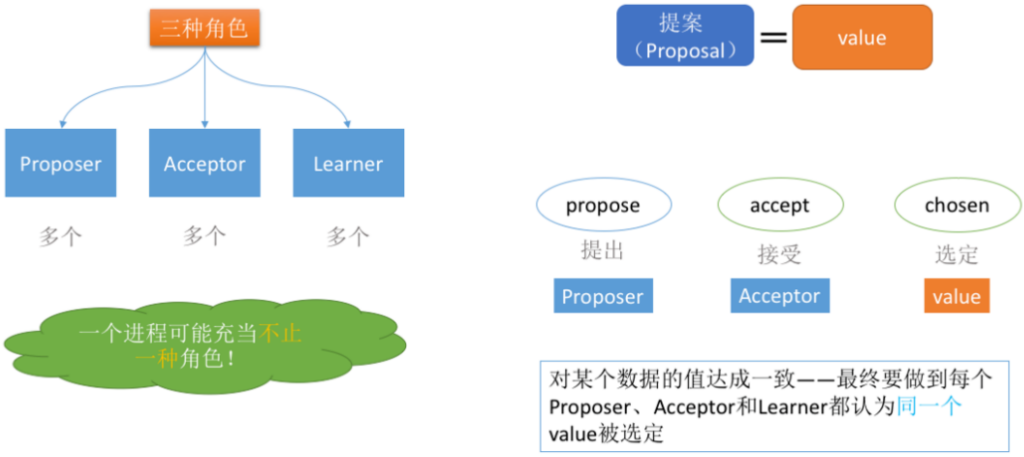
在具体的实现中，一个进程可能 同时充当多种角色 。比如一个进程可能 既是Proposer又是Acceptor又是Learner 。Proposer负责提出提案，Acceptor负责对提案作出裁决（accept与否），learner负责学习提案结果。

- 提案

还有一个很重要的概念叫提案（Proposal）。最终要达成一致的value就在提案里。只要Proposer发的提案被Acceptor接受（半数以上的Acceptor同意才行），Proposer就认为该提案里的value被选定了。Acceptor告诉Learner哪个value被选定，Learner就认为那个value被选定。只要Acceptor接受了某个提案，Acceptor就任务该提案里的value被选定了。

- 其他

为了避免单点故障，会有一个Acceptor集合，Proposer向Acceptor集合发送提案，Acceptor集合中的每个成员都有可能同意该提案且每个Acceptor只能批准一个提案，只有当一半以上的成员同意了一个提案，就认为该提案被选定了。



算法要求

安全性（safaty）要求如下：

- 只有被提出的value才能被选定。
- 只有一个value被选定，并且
- 如果某个进程认为某个value被选定了，那么这个value必须是真的被选定的那个。

Paxos的目标：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。

算法过程

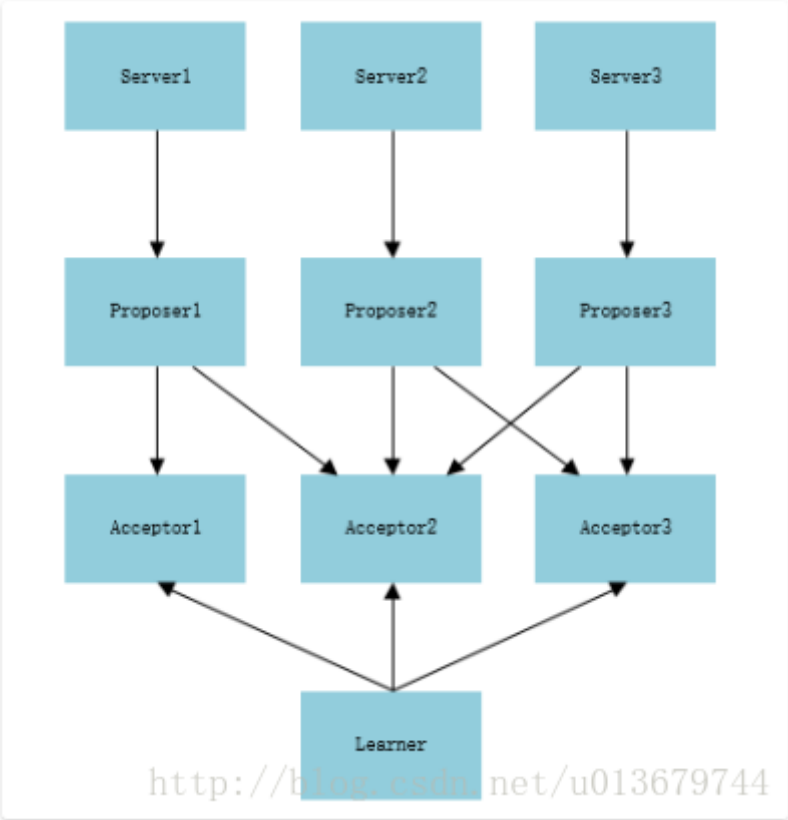
Paxos算法类似于两阶段提提交，其算法执行过程分为两个阶段。

- **阶段一（prepare阶段）：**
  - Proposer选择一个提案编号N，然后向**半数以上**的Acceptor发送编号为N的Prepare请求。
  - 如果一个Acceptor收到一个编号为N的Prepare请求，如果小于它已经响应过的请求，则拒绝，不回应或回复error。若N大于该Acceptor已经响应过的所有Prepare请求的编号（maxN），那么它就会将它已经接受过（已经经过第二阶段accept的提案）的编号最大的提案（如果有的话，如果还没有的accept提案的话返回{pok，null，null}）作为响应反馈给Proposer，**同时该Acceptor承诺\不再接受任何编号小于N的提案**
- **阶段二（accept阶段）**
  - 如果一个Proposer收到半数以上Acceptor对其发出的编号为N的Prepare请求的响应，那么它就会发送一个针对[N,V]提案的Accept请求给半数以上的Acceptor。注意：**V就是收到的响应中编号最大的提案的value（某个acceptor响应的它已经通过的{acceptN，acceptV}，如果响应中不包含任何提案，那么V就由Proposer自己决定。**
  - 如果Acceptor收到一个针对编号为N的提案的Accept请求，只要该Acceptor没有对编号大于N的Prepare请求做出过响应，它就接受该提案。如果N小于Acceptor以及响应的prepare请求，则拒绝，不回应或回复error（当proposer没有收到过半的回应，那么他会重新进入第一阶段，递增提案号，重新提出prepare请求）。

自己对accept的理解：

1. 每个Acceptor每时每刻只能接收一个提案
2. 所谓的解决一致性就像对所提的提案进行一个先后顺序
3. 每个Acceptor接收的提案只会越来越大

案例



第一个prepare阶段

第一步：

每个`server`向`proposer`发送消息，表示自己要当leader，假设proposer收到消息的时间不一样，顺序是：`proposer2` -> `proposer1` -> `proposer3`，消息编号依次为1、2、3。因为需要发送给超过半数的成员，这里选择两个。

第二步：

假设`proposer1`发送的消息`2`先到达`acceptor1`和`acceptor2`，它们都没有接收过请求，所以接收该请求并返回`【pok，null，null】`给`proposer1`，同时`acceptor1`和`acceptor2`承诺不再接受编号小于2的请求。

`roposer2`的消息`1`到达`acceptor2`和`acceptor3`，`acceptor3`没有接受过请求，所以返回`proposer2` `【pok，null，null】`，`acceptor3`并承诺不再接受编号小于`1`的消息。而`acceptor2`已经接受`proposer1`的请求并承诺不再接收编号小于`2`的请求，所以`acceptor2`拒绝`proposer2`的请求；

`proposer3`的消息到达`acceptor2`和`acceptor3`，它们都接受过提议，但编号`3`的消息大于`acceptor2`已接受的`2`和`acceptor3`已接受的`1`，所以他们都接受该提议，并返回`proposer3` `【pok，null，null】`；

第三步：

`proposer2`没有收到过半的回复，所以重新取得编号`4`，并发送给`acceptor2`和`acceptor3`，此时编号4大于它们已接受的提案编号3，所以接受该提案，并返回`proposer2` ``【pok，null，null】`。

第一个accept方案

第一步：

`Proposer3`收到半数以上（两个）的回复，并且返回的`value`为`null`，所以，`proposer3`提交了`【3，server3】`的提案。
`Proposer1`也收到过半回复，返回的`value`为`null`，所以`proposer1`提交了`【2，server1】`的提案。
`Proposer2`也收到过半回复，返回的`value`为`null`，所以`proposer2`提交了`【4，server2】`的提案。

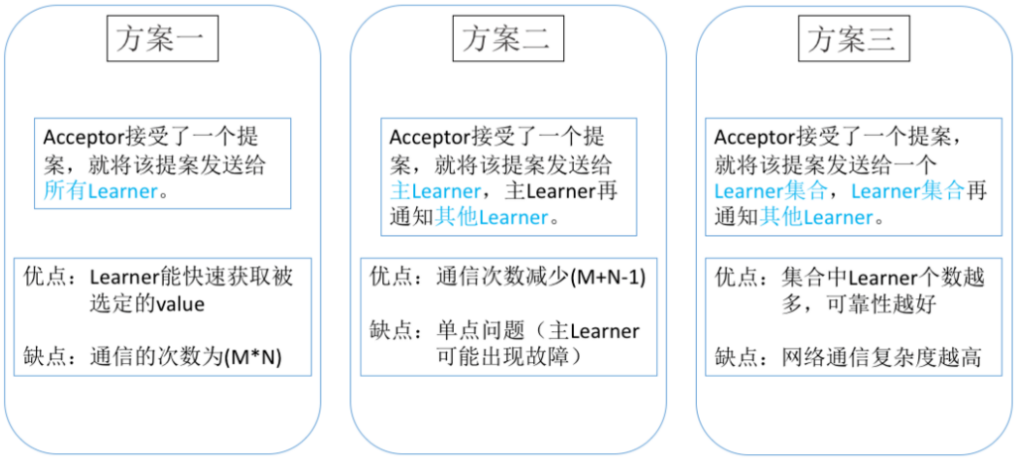
第二步：

`Acceptor1`和`acceptor2`接收到`proposer1`的提案`【2，server1】`，`acceptor1`通过该请求，`acceptor2`承诺不再接受编号小于4的提案，所以拒绝；
`Acceptor2`和`acceptor3`接收到`proposer2`的提案`【4，server2】`，都通过该提案,；
`Acceptor2`和`acceptor3`接收到`proposer3`的提案`【3，server3】`，它们都承诺不再接受编号小于4的提案，所以都拒绝。

最后

`proposer1`和`proposer3`会再次进入第一阶段，但这时候`Acceptor2`和`acceptor3`已经通过了提案`（AcceptN = 4，AcceptV=server2）`，并达成了多数，所以`proposer`会递增提案编号，并最终改变其值为`server2`。最后所有的`proposer`都肯定会达成一致，这就迅速的达成了一致。

提案的获取



通过选取主Proposer保证算法的活性

两个Proposer分别生成提案后陷入死循环？
出现原因：Proposer P1 提出了一个编号为M1的提案，并完成了上述阶段一的流程。但是这个时候Proposer P2提出了一个编号为M2的提案，同样也完成了阶段一的流程，于是Accept已经承诺步再批准小于M2的提案，因此，P1进入第二阶段时候，Accept请求被拒绝，然后又提出了M3提案，然后P2又被拒绝
为了避免出现选择一个主Proposer，并规定只有Z主Proposer才能提出议案。主要主的Proposer和过半的Accept能够正常通信，则主Proposer提出一个编号更高的提案，该提案终会被批准。