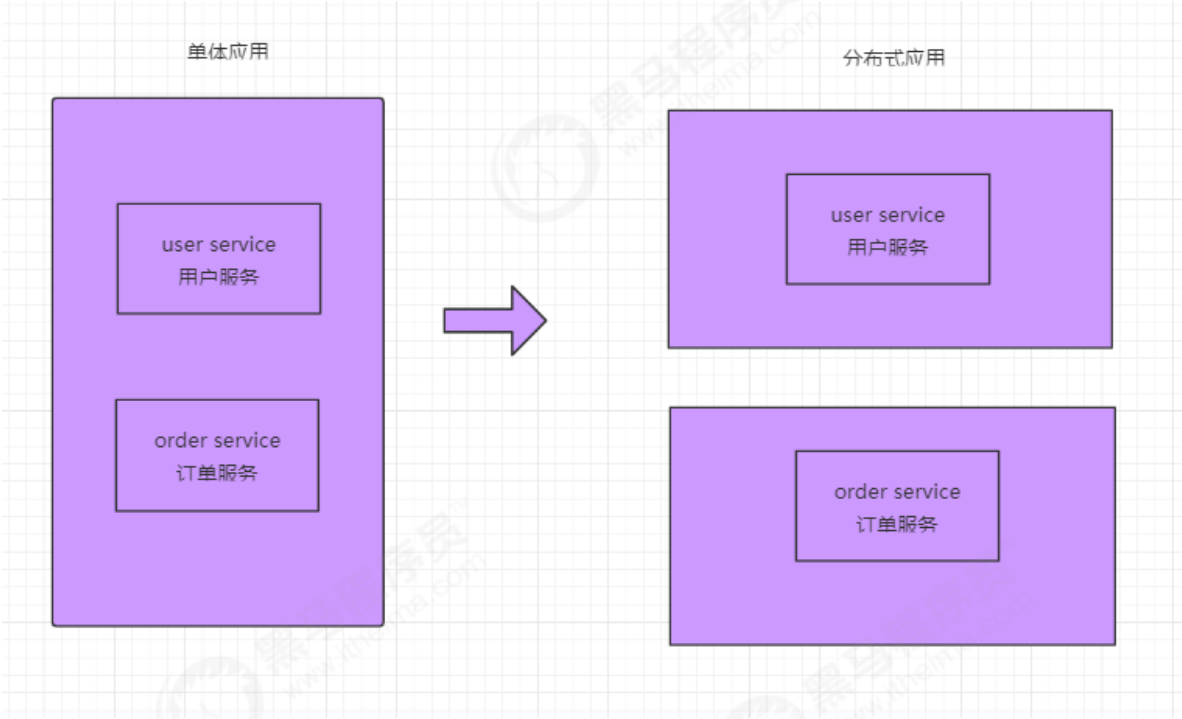
day01

1、分布式概述

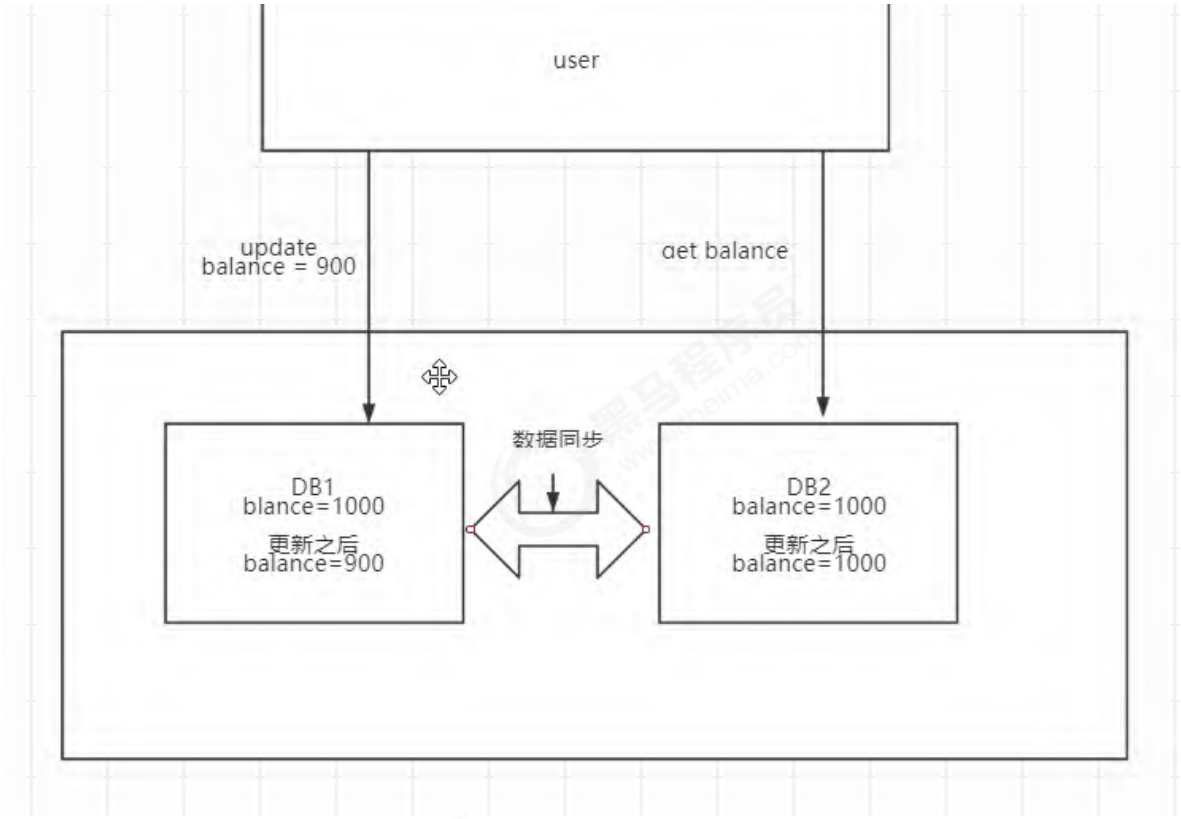
早期我们使用单体架构，即所有服务部署在一台服务器的一个进程中，随着互联网的发展，逐步演进为分布式架构，多个服务分别部署在不同机器的不同进程中。



2、zookeeper概述

zookeeper是一个开源的分布式协调服务，提供分布式数据一致性解决方案，分布式应用程序可以实现数据发布订阅、负载均衡、命名服务、集群管理分布式锁、分布式队列等功能。

zookeeper提供了分布式数据一致性解决方案，那什么是分布式数据一致性？首先我们谈谈什么叫一致性？



如图在上图中有用户user在DB1中修改balance=900,如果user下一次read请求到DB2数据，此时DB1的数据还没及时更新到DB2中，就会造成整个数据库集群数据不一致。

数据一致性分为强一致性和最终一致性，强一致性指的如果数据不一致，就不对外提供数据服务，保证用户读取的数据始终是一致的。数据强一致性只需要通过锁机制即可解决，在案例中通过在DB2没有从DB1同步数据之前上锁，不对外提供读操作。只有当同步完成以后才对外提供服务。而最终一致性要求数据最终同步即可，没有实时性要求。

3、CAP原则

CAP在分布式系统中主要指的是一致性（(Consistency）、可用性（Availability）和分区容错性（Partition tolerance）

- 一致性

一致性指的是强一致性

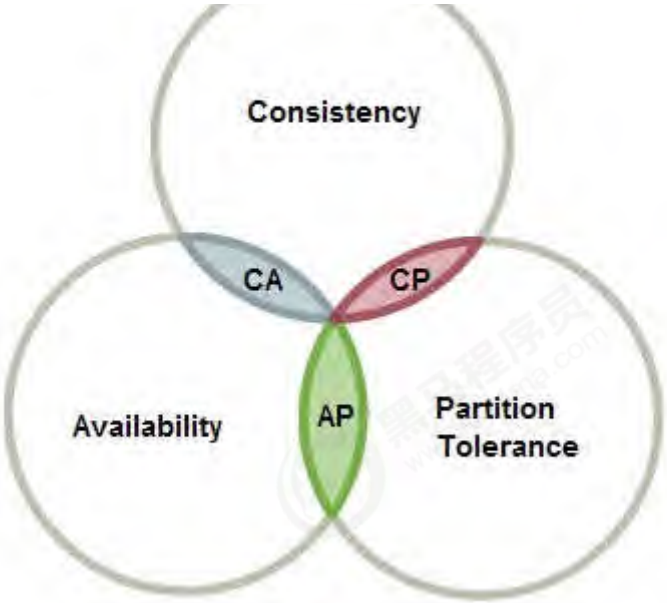
- 可用性

系统提供的服务一直处于可用状态，用户的操作请求在指定的响应时间内响应请求，超出时间范围，认为系统不可用

- 分区容错性

分布式系统在遇到任何网络分区故障的时候，仍需要能够保证对外提供一致性和可用性服务，除非是整个网络都发生故障。

在一个分布式系统中不可能同时满足一致性、可用性、分区容错性，最多满足两个，对于分布式互联网应用而言，必须保证P，所以要么满足AP模型、要么满足CP模型

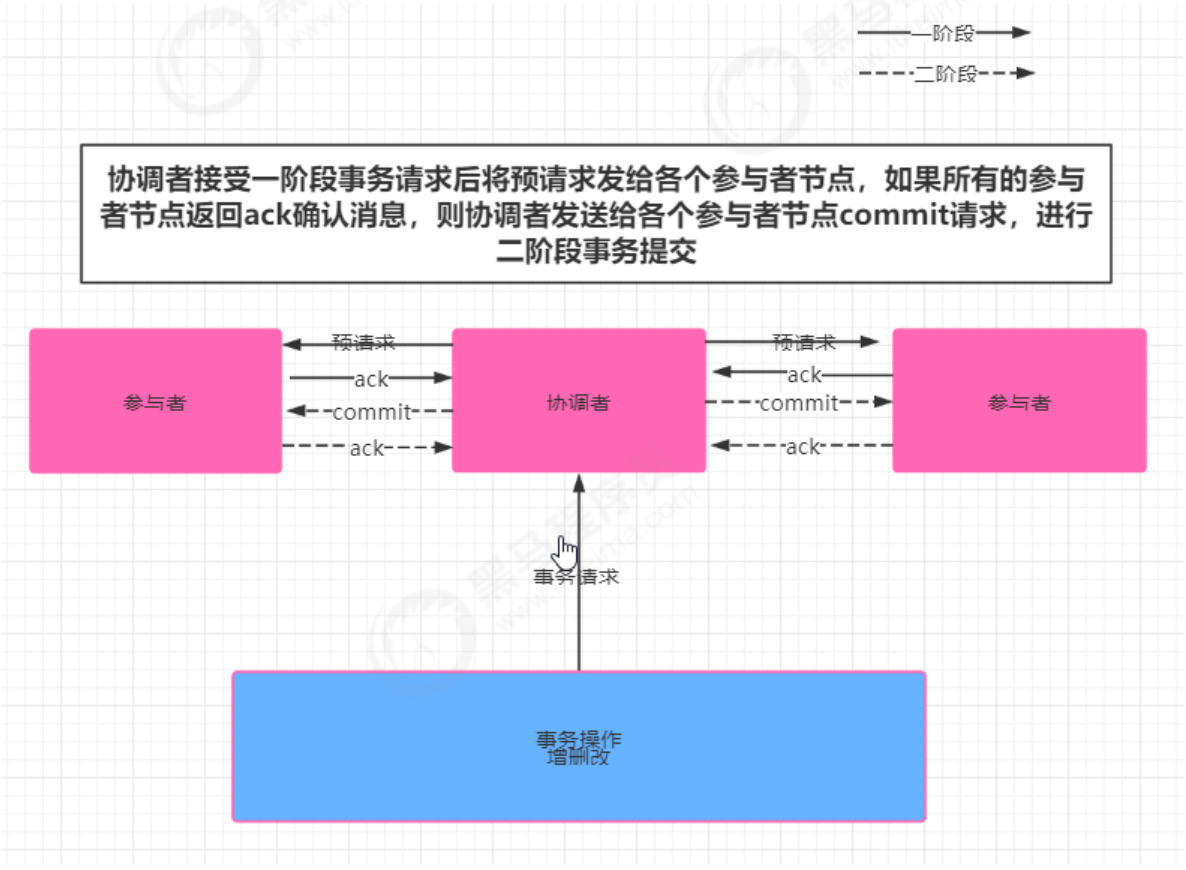


4、一致性协议

事务需要跨多个分布式节点时，为了保证事务的ACID特性，需要选举出一个协调者来协调分布式各个节点的调度，基于这个思想衍生了多种一致性协议:

1）2PC 二阶段提交

顾名思义，二阶段提交叫事务的提交过程分为两个阶段：



- 阶段一 提交事务请求

1、协调者向所有的参与者节点发送事务内容，询问是否可以执行事务操作，并等待其他参与者节点的反馈

3、各参与者节点反馈给协调者，事务是否可以执行

- 阶段二 事务提交

根据一阶段各个参与者节点反馈的ack,如果所有参与者节点反馈ack，则执行事务提交，否则中断事务

事务提交：

1、协调者向各个参与者节点发送commit请求

2、参与者节点接受到commit请求后，执行事务的提交操作

3、各参与者节点完成事务提交后，向协调者返送提交commit成功确认消息

4、协调者接受各个参与者节点的ack后，完成事务commit

中断事务：

1、发送回滚请求

2、各个参与者节点回滚事务

3、反馈给协调者事务回滚结果

4、协调者接受各参与者节点ack后回滚事务

二阶段提交存在的问题：

- 同步阻塞

二阶段提交过程中，所有参与事务操作的节点处于同步阻塞状态，无法进行其他的操作

- 单点问题

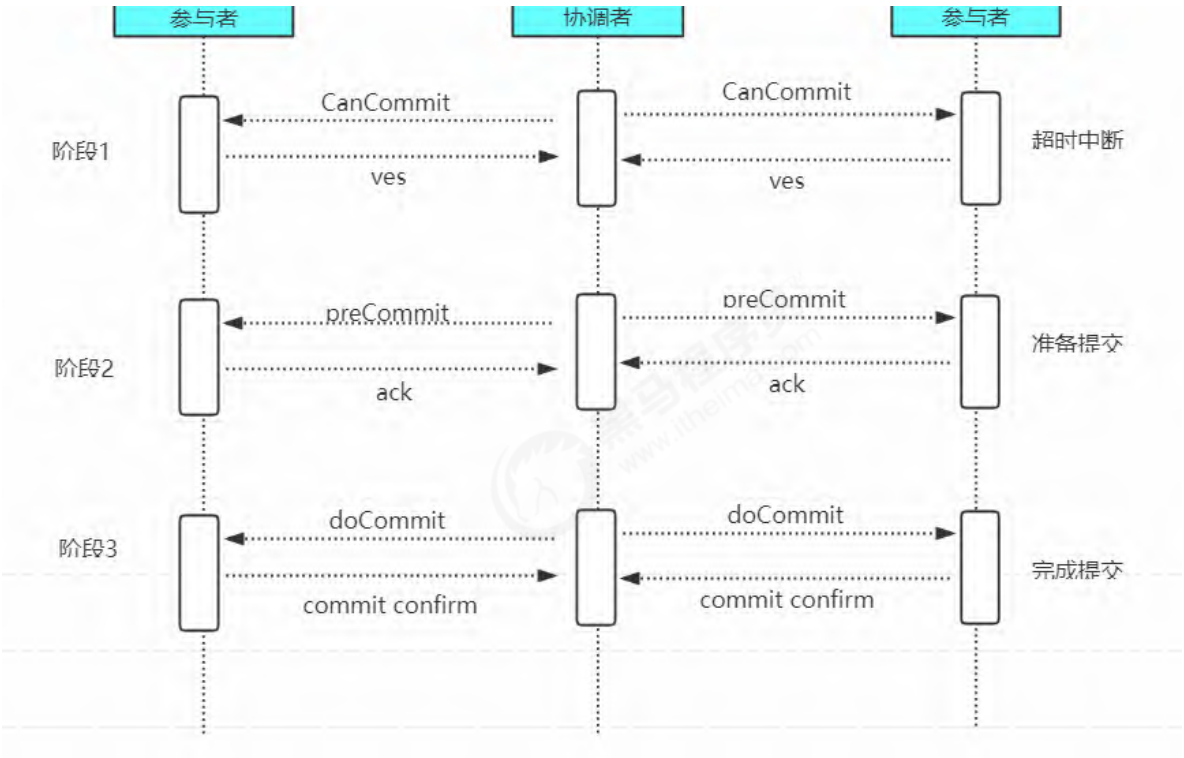
一旦协调者出现单点故障，无法保证事务的一致性操作

- 脑裂导致数据不一致

如果分布式节点出现网络分区，某些参与者未收到commit提交命令。则出现部分参与者完成数据提交。未收到commit的命令的参与者则无法进行事务提交，整个分布式系统便出现了数据不一致性现象。

2) 3PC 三阶段提交

3PC是2PC的改进版，实质是将2PC中提交事务请求拆分为两步，形成了CanCommit、PreCommit、doCommit三个阶段的事务一致性协议



- **阶段一 : CanCommit**

  1、事务询问

  2、各参与者节点向协调者反馈事务询问的响应

- **阶段二 : PreCommit**

  根据阶段一的反馈结果分为两种情况

  1、执行事务预提交

  1）发送预提交请求

  协调者向所有参与者节点发送preCommit请求，进入prepared阶段

  2）事务预提交

  各参与者节点接受到preCommit请求后，执行事务操作

  3）各参与者节点向协调者反馈事务执行

  2、中断事务

  任意一个参与者节点反馈给协调者响应No时，或者在等待超时后，协调者还未收到参与者的反馈，就中断事务，中断事务分为两步：

  1）协调者向各个参与者节点发送abort请求

  2）参与者收到abort请求，或者等待超时时间后，中断事务

- **阶段三 : doCommit**

  1、执行提交

  1）发送提交请求

  协调者向所有参与者节点发送doCommit请求

  2）事务提交

  各参与者节点接受到doCommit请求后，执行事务提交操作

  3）反馈事务提交结果

4）事务完成

协调者接受各个参与者反馈的ack后，完成事务

2、中断事务

1）参与者接受到abort请求后，执行事务回滚

2）参与者完成事务回滚以后，向协调者发送ack

3）协调者接受回滚ack后，回滚事务

3PC相较于2PC而言，解决了协调者挂点后参与者无限阻塞和单点问题，但是仍然无法解决网络分区问题

3) Paxos算法

Paxos算法是Leslie Lamport 1990年提出的一种一致性算法，该算法是一种提高分布式系统容错性的一致性算法，解决了3PC中网络分区的问题，paxos算法可以在节点失效、网络分区、网络延迟等各种异常情况下保证所有节点都处于同一状态，同时paxos算法引入了“过半”理念，即少数服从多数原则。

paxos有三个版本：

- Basic Paxos
- Multi Paxos
- Fast Paxos

在paxos算法中，有四种种角色，分别具有三种不同的行为，但多数情况，一个进程可能同时充当多种角色。

- client：系统外部角色，请求发起者，不参与决策
- proposer：提案提议者
- acceptor：提案的表决者，即是否accept该提案，只有超过半数以上的acceptor接受了提案，该提案才被认为被“选定”
- learners：提案的学习者，当提案被选定后，其同步执行提案，不参与决策

Paxos算法分为两个阶段：prepare阶段、accept阶段

- prepare阶段

  <1> proposer提出一个提案，编号为N,发送给所有的acceptor。

  <2> 每个表决者都保存自己的accept的最大提案编号maxN，当表决者收到prepare(N)请求时，会比较N与maxN的值，若N小于maxN,则提案已过时，拒绝prepare(N)请求。若N大于等于maxN，则接受提案，并将该表决者曾经接受过的编号最大的提案Proposal(myid,maxN,value)反馈给提议者：其中myid表示表决者acceptor的标识id，maxN表示接受过的最大提案编号maxN，value表示提案内容。若当前表决者未曾accept任何提议，会将proposal(myid,null,null)反馈给提议者。

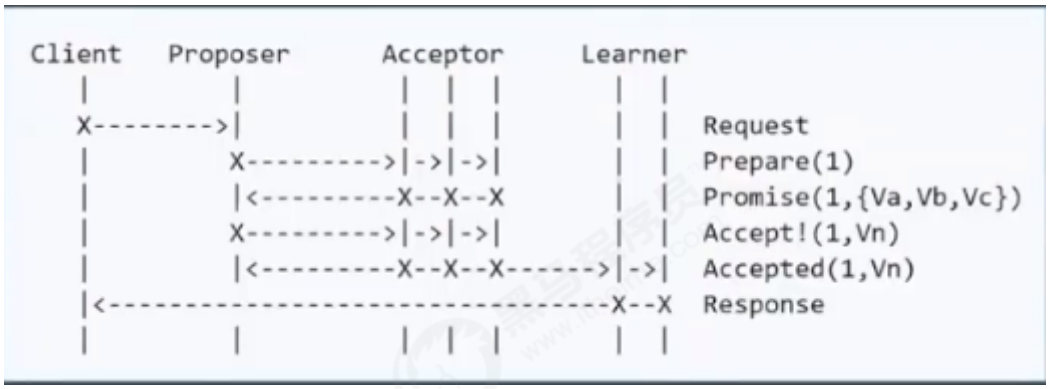
- accept阶段

  <1> 提议者proposal发出prepare(N),若收到超过半数表决者acceptor的反馈，proposal将真正的提案内容proposal(N,value)发送给所有表决者。

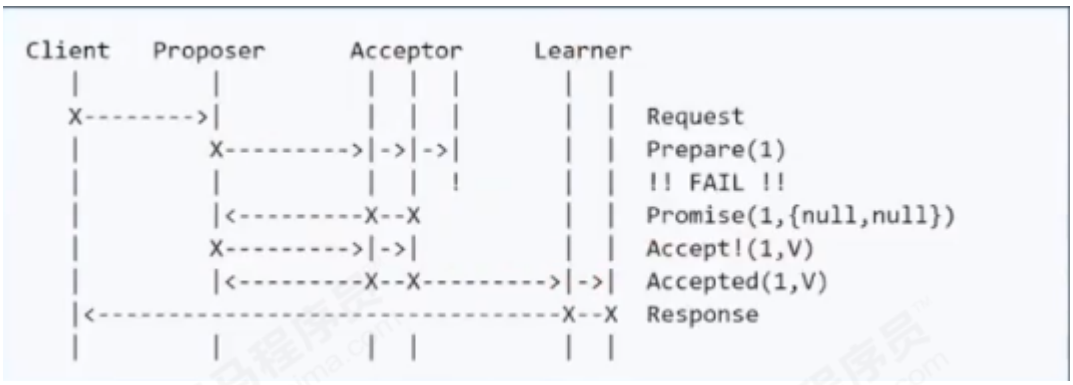
  <2> 表决者acceptor接受提议者发送的proposal(N,value)提案后，会将自己曾经accept过的最大提案编号maxN和反馈过的prepare的最大编号，若N大于这两个编号，则当前表决者accept该提案，并反馈给提议者。否则拒绝该提议。

learner，主动同步提议者的提案。

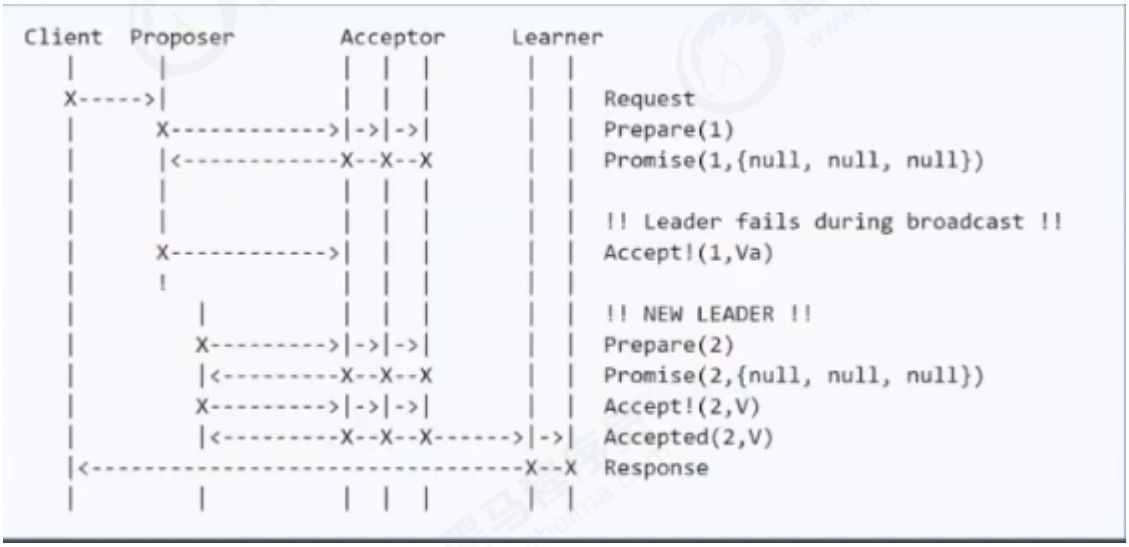
正常流程



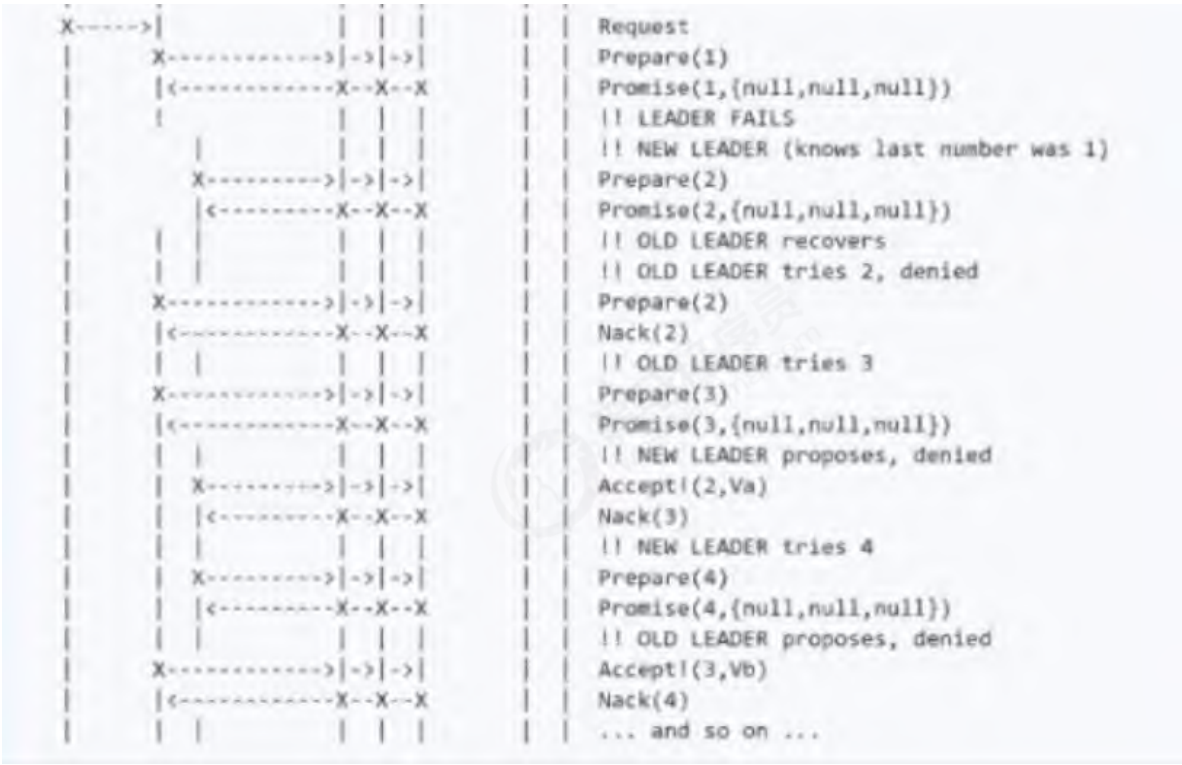
单点故障，部分节点失败



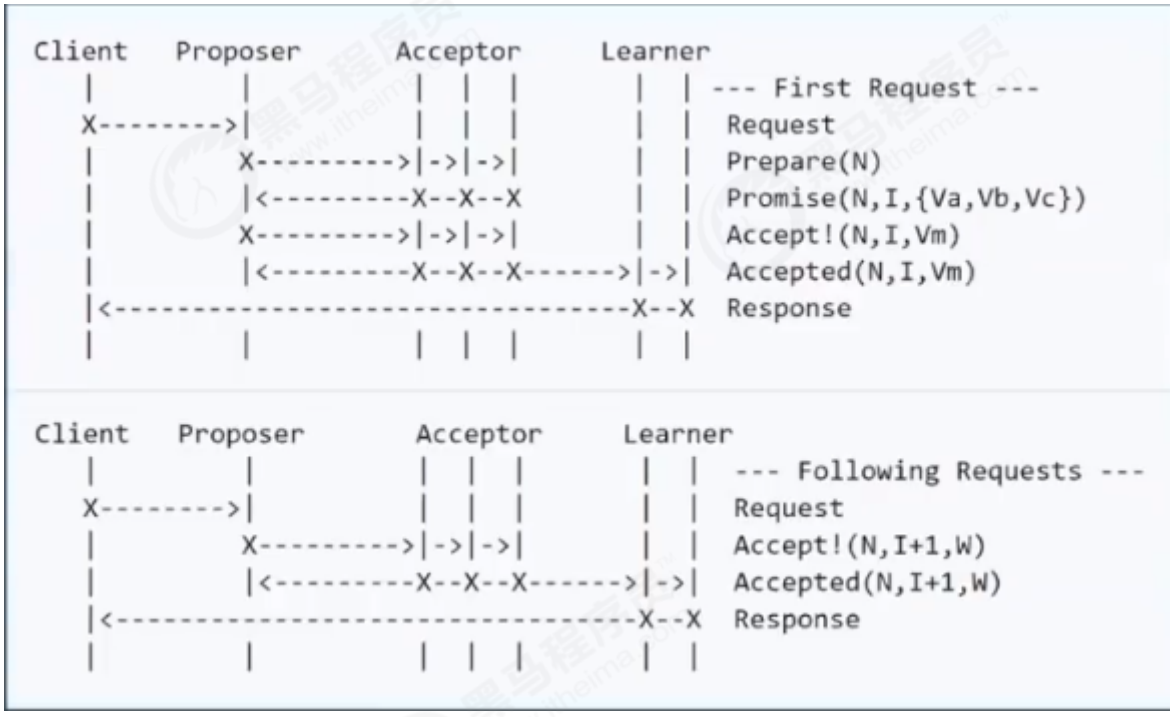
proposer失败



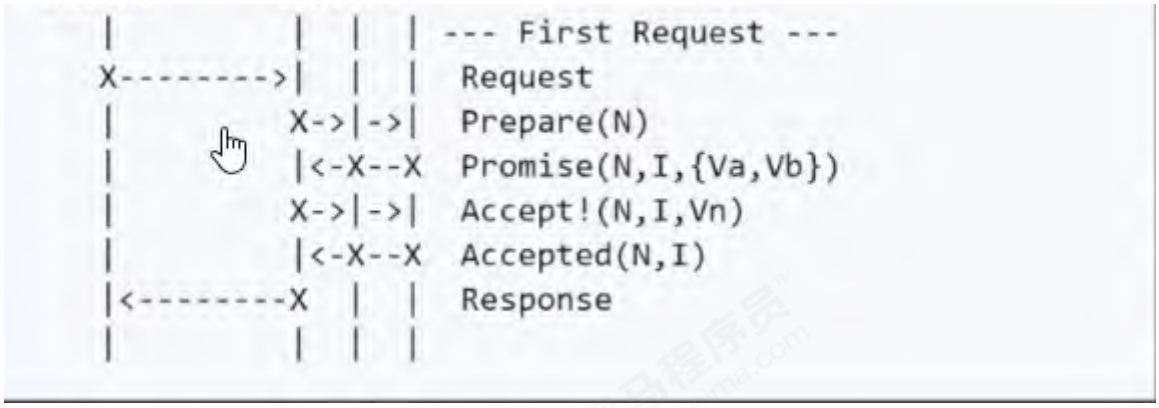
Basic Paxos算法存在活锁问题（liveness）或dueling，而且较难实现



Multi Paxos: 唯一的proposor，即leader



简化角色



4) ZAB协议(Fast Paxos)

由于paxos算法实现起来较难，存在活锁和全序问题（无法保证两次最终提交的顺序），所以zookeeper并没有使用paxos作为一致性协议，而是使用了ZAB协议。

ZAB（`zookeeper atomic broadcast`）:是一种支持崩溃恢复的原子广播协议，基于Fast paxos实现

ZooKeeper使用单一主进程Leader用于处理客户端所有事务请求，，即写请求。当服务器数据发生变更好，集群采用ZAB原子广播协议，以事务提交proposal的形式广播到所有的副本进程，每一个事务分配一个全局的递增的事务编号xid。

若客户端提交的请求为读请求时，则接受请求的节点直接根据自己保存的数据响应。若是写请求，且当前节点不是leader，那么该节点就会将请求转发给leader，leader会以提案的方式广播此写请求，如果超过半数的节点同意写请求，则该写请求就会提交。leader会通知所有的订阅者同步数据。

`zookeeper的三种角色`：

为了避免zk的单点问题，zk采用集群方式保证zk高可用

- leader

  leader负责处理集群的写请求，并发起投票，只有超过半数的节点同意后才会提交该写请求

- follower

  处理读请求，响应结果。转发写请求到leader，在选举leader过程中参与投票

- observer

  observer可以理解为没有投票权的follower，主要职责是协助follower处理读请求。那么当整个zk集群读请求负载很高时，为什么不增加follower节点呢？原因是增加follower节点会让leader在提出写请求提案时，需要半数以上的follower投票节点同意，这样会增加leader和follower的通信通信压力，降低写操作效率。

`zookeeper两种模式`

- 恢复模式

  当服务启动或领导崩溃后，zk进入恢复状态，选举leader，leader选出后，将完成leader和其他机器的数据同步，当大多数server完成和leader的同步后，恢复模式结束

- 广播模式

  一旦Leader已经和多数的Follower进行了状态同步后，进入广播模式。进入广播模式后，如果有新加入的服务器，会自动从leader中同步数据。leader在接收客户端请求后，会生成事务提案广播给其他机器，有超过半数以上的follower同意该提议后，再提交事务。

  注意在ZAB的事务的二阶段提交中，移除了事务中断的逻辑，follower要么ack，要么放弃，leader无需等待所有的follower的ack。

zxid是64位长度的Long类型，其中高32位表示纪元epoch，低32位表示事务标识xid。即zxid由两部分构成：epoch和xid

每个leader都会具有不同的epoch值，表示一个纪元，每一个新的选举开启时都会生成一个新的epoch，新的leader产生，会更新所有的zkServer的zxid的epoch，xid是一个依次递增的事务编号。

leader选举算法：

启动过程

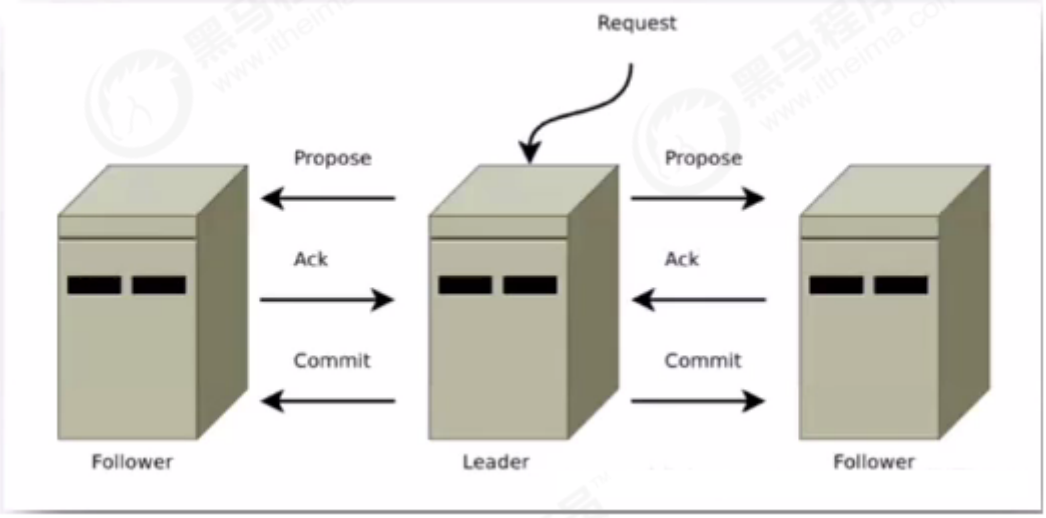
- 每一个server发出一个投票给集群中其他节点
- 收到各个服务器的投票后，判断该投票有效性，比如是否是本轮投票，是否是 looking状态
- 处理投票，pk别人的投票和自己的投票 比较规则xid>myid “取大原则”
- 统计是否超过半数的接受相同的选票
- 确认leader，改变服务器状态
- 添加新server，leader已经选举出来，只能以follower身份加入集群中

崩溃恢复过程

- leader挂掉后，集群中其他follower会将状态从FOLLOWING变为LOOKING,重新进入leader选举
- 同上启动过程

消息广播算法：

一旦进入广播模式，集群中非leader节点接受到事务请求，首先会将事务请求转发给服务器，leader服务器为其生成对应的事务提案proposal,并发送给集群中其他节点，如果过半则事务提交；



- leader接受到消息后，消息通过全局唯一的64位自增事务id，zxid标识
- leader发送给follower的提案是有序的，leader会创建一个FIFO队列，将提案顺序写入队列中发送给follower
- follower接受到提案后，会比较提案zxid和本地事务日志最大的zxid，若提案zxid比本地事务id大，将提案记录到本地日志中，反馈ack给leader，否则拒绝
- leader接收到过半ack后，leader向所有的follower发送commit，通知每个follower执行本地事务

5、zookeeper环境搭建

zookeeper安装以linux环境为例，windows省略

1）单机环境

1. 安装jdk

2. zookeeper压缩包zookeeper-3.4.6.tar.gz上传到linux系统

Alt+P 进入SFTP ，输入put d:\zookeeper-3.4.6.tar.gz 上传，d:\zookeeper-3.4.6.tar.gz是本地存放zookeeper的路径

或者rz上传

3. 解压缩压缩包

```
tar -zxvf zookeeper-3.4.6.tar.gz
```

4. 进入 zookeeper-3.4.13 目录，创建 data 文件夹

```
cd conf
mv zoo_sample.cfg zoo.cfg
```

5. 修改zoo.cfg中的data属性

```
dataDir=/root/zookeeper-3.4.6/data
```

6. zookeeper服务启动

进入bin目录，启动服务输入命令

```
 ./zkServer.sh start
```

输出以下内容表示启动成功

```
JMX enabled by default
Using config: /root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

关闭服务输入命令

```
./zkServer.sh stop
```

输出以下提示信息

```
JMX enabled by default
Using config: /root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
```

查看状态

```
./zkServer.sh status
```

如果启动状态，提示

```
JMX enabled by default
Using config: /root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: standalone
```

如果未启动状态，提示:

```
JMX enabled by default
Using config: /root/zookeeper-3.4.6/bin/../conf/zoo.cfg
Error contacting service. It is probably not running.
```

真实的集群是需要部署在不同的服务器上的，但是在我们测试时启动多个虚拟机的内存消耗太大，所以我们通常会搭建**伪集群**，也就是把所有的服务都搭建在一台虚拟机上，用端口进行区分。

准备工作

（1）安装JDK 【此步骤省略】。

（2）Zookeeper压缩包上传到服务器

（3）将Zookeeper解压到 /usr/local/zkcluster，创建data目录 ，将 conf下zoo_sample.cfg 复制三份文件并改名为 zoo1.cfg、zoo2.cfg、zoo3.cfg

（4）在解压后的Zookeeper目录下创建data目录，并分别创建三个子目录data1、data2、data3

/usr/local/zookeeper-cluster/zookeeper-1

/usr/local/zookeeper-cluster/zookeeper-2

/usr/local/zookeeper-cluster/zookeeper-3

```
[root@localhost ~]# mkdir /usr/local/zkcluster/data
[root@localhost ~]# cd data
[root@localhost ~]# mkdir data1
[root@localhost ~]# mkdir data2
[root@localhost ~]# mkdir data3
```

（5） 配置每一个Zookeeper 的dataDir（zoo.cfg） clientPort 分别为2181 2182 2183

修改 /usr/local/zookeeper-cluster/zookeeper-1/conf/zoo.cfg

```
clientPort=2181
dataDir=/usr/local/zookeeper-cluster/zookeeper-1/data
```

修改/usr/local/zookeeper-cluster/zookeeper-2/conf/zoo.cfg

```
clientPort=2182
dataDir=/usr/local/zookeeper-cluster/zookeeper-2/data
```

修改/usr/local/zookeeper-cluster/zookeeper-3/conf/zoo.cfg

```
clientPort=2183
dataDir=/usr/local/zookeeper-cluster/zookeeper-3/data
```

配置集群

（1）在每个zookeeper的 data 目录下创建一个 myid 文件，内容分别是1、2、3 。这个文件就是记录每个服务器的ID

（2）在每一个zookeeper 的 zoo.cfg配置客户端访问端口（clientPort）和集群服务器IP列表。

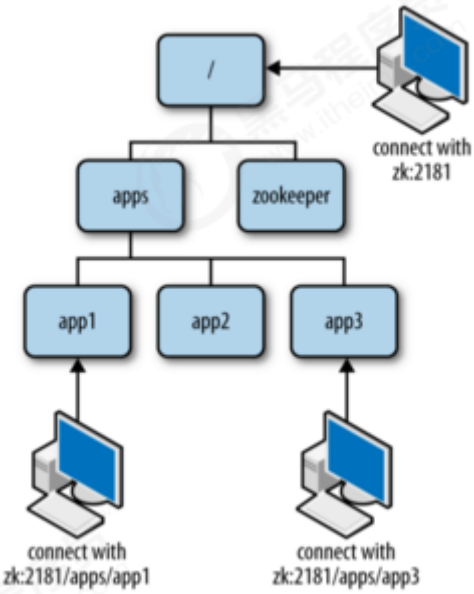
```
server.1=192.168.25.129:2881:3881
server.2=192.168.25.129:2882:3882
server.3=192.168.25.129:2883:3883
server.服务器ID=服务器IP地址：服务器之间通信端口：服务器之间投票选举端口
```

依次启动三个zk实例，其中有一个leader和两个follower

6、zookeeper基本使用

数据结构

ZooKeeper数据模型的结构与Unix文件系统很类似，整体上可以看作是一棵树，每个节点称做一个ZNode，每个ZNode都可以通过其路径唯一标识



Znode节点类型

- 持久化目录节点（PERSISTENT）

  客户端与zookeeper断开连接后，该节点依旧存在
- 持久化顺序编号目录节点（PERSISTENT_SEQUENTIAL）

  客户端与zookeeper断开连接后，该节点依旧存在，Zookeeper会给该节点按照顺序编号
- 临时目录节点（EPHEMERAL）

  客户端与zookeeper断开连接后，该节点被删除
- 临时顺序编号目录节点（EPHEMERAL_SEQUENTIAL）

  客户端与zookeeper断开连接后，该节点被删除，Zookeeper会给该节点按照顺序编号

命令行使用

通过zkClient进入zookeeper客户端命令行，输入help查看zookeeper客户端的指令

如上图列出了zookeeper所有的客户端命令行，下面主要讲解常见的几个命令行

1. 使用 ls 命令来查看当前znode中所包含的内容

```
ls path [watch]
```

2. 查看当前节点数据并能看到更新次数等数据

```
ls2 path [watch]
```

3. 创建节点 -s 含有序列 -e 临时

```
create
```

4. 获得节点的值

```
get path [watch]
```

5. 设置节点的值

```
set
```

6. 查看节点状态

```
stat
```

7. 删除节点

```
delete
```

```
rmr
```

api使用

maven依赖

```
        <dependency>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
            <version>3.4.13</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
```

ZooKeeper zk = new ZooKeeper(String connectString, int sessionTimeout, Watcher watcher)	创建zookeeper连接，connectString表示连接的zookeeper服务器的地址，sessionTimeOut指定会话的的超时时间，Watcher配置监听
String create(String path, byte[] data, List acl,CreateMode createMode)	创建一个给定的目录节点 path, 并给它设置数据，CreateMode 标识有四种形式的目录节点，分别是 PERSISTENT：持久化目录节点，这个目录节点存储的数据不会丢失；PERSISTENT_SEQUENTIAL：顺序自动编号的目录节点，这种目录节点会根据当前已近存在的节点数自动加 1，然后返回给客户端已经成功创建的目录节点名；EPHEMERAL：临时目录节点，一旦创建这个节点的客户端与服务器端口也就是 session 超时，这种节点会被自动删除；EPHEMERAL_SEQUENTIAL：临时自动编号节点
Stat exists(String path, boolean watch)	判断某个 path 是否存在，并设置是否监控这个目录节点，这里的 watcher 是在创建 ZooKeeper 实例时指定的 watcher，exists方法还有一个重载方法，可以指定特定的watcher

Stat exists(String path,Watcher watcher)	重载方法，这里给某个目录节点设置特定的 watcher，Watcher 在 ZooKeeper 是一个核心功能，Watcher 可以监控目录节点的数据变化以及子目录的变化，一旦这些状态发生变化，服务器就会通知所有设置在这个目录节点上的 Watcher，从而每个客户端都很快知道它所关注的目录节点的状态发生变化，而做出相应的反应
void delete(String path, int version)	删除 path 对应的目录节点，version 为 -1 可以匹配任何版本，也就删除了这个目录节点所有数据
ListgetChildren(String path, boolean watch)	获取指定 path 下的所有子目录节点，同样 getChildren方法也有一个重载方法可以设置特定的 watcher 监控子节点的状态
Stat setData(String path, byte[] data, int version)	给 path 设置数据，可以指定这个数据的版本号，如果 version 为 -1 怎可以匹配任何版本
byte[] getData(String path, boolean watch, Stat stat)	获取这个 path 对应的目录节点存储的数据，数据的版本等信息可以通过 stat 来指定，同时还可以设置是否监控这个目录节点数据的状态