[1.1 为什么要学习zk？ 1](#_Toc75813036)

[1.1.1 分布式环境特点 2](#_Toc75813037)

[1.1.2 分布式环境存在的问题 2](#_Toc75813038)

[1.1.3 CAP理论 2](#_Toc75813039)

[1.1.4 BASE理论 3](#_Toc75813040)

[1.1.5 部署节点为奇数? 3](#_Toc75813041)

[1.2 zookeeper 安装 4](#_Toc75813042)

[1.2.1 zookeeper步骤 4](#_Toc75813043)

[1.2.2 zookeeper的api接口 5](#_Toc75813044)

[1.3 Zookeeper原理与架构 6](#_Toc75813045)

[1.3.1 配置文件及基本命令 6](#_Toc75813046)

[1.3.2 zookeeper特性 7](#_Toc75813047)

[1.3.3 Zookeeper的数据结构 7](#_Toc75813048)

[1.3.4 Znode类型 7](#_Toc75813049)

[1.3.5 zookeeper⻆⾊、节点状态 8](#_Toc75813050)

[1.3.6 Zk集群服务节点状态 9](#_Toc75813051)

[1.3.7 Zk的znode存储 9](#_Toc75813052)

[1.3.8 Zookeeper 的内部⽹络拓扑 9](#_Toc75813053)

[1.4 Paxos 算法 10](#_Toc75813054)

[1.4.1 分布式一致性算法paxos 10](#_Toc75813055)

[1.5 ZAB协议 12](#_Toc75813056)

[1.5.1 Zab协议的核心 13](#_Toc75813057)

[1.5.2 崩溃恢复 消息广播 13](#_Toc75813058)

[1.5.3 ZAB协议 16](#_Toc75813059)

[1.5.4 消息广播模式 17](#_Toc75813060)

[1.5.5 崩溃恢复 18](#_Toc75813061)

[1.5.6 数据同步 19](#_Toc75813062)

[1.5.7 ZAB协议原理 19](#_Toc75813063)

[1.5.8 Zookeeper设计目标 19](#_Toc75813064)

[1.6 分布式事务 20](#_Toc75813065)

[1.6.1 什么是事务 20](#_Toc75813066)

[1.6.2 本地事务 20](#_Toc75813067)

[1.6.3 分布式事务 20](#_Toc75813068)

[1.6.4 分布式事务产生的场景 21](#_Toc75813069)

[1.6.5 分布式事务基础理论 21](#_Toc75813070)

[1.6.6 分布式事务解决方案之2PC(两阶段提交) 23](#_Toc75813071)

## 为什么要学习zk？

重点掌握分布式环境的演进过程，从一个单节点开始，慢慢过渡到分布式，为什么单节点不行。

传统的单节点架构自然有问题，到了分布式架构中，问题肯定也不是，那么这些问题即使我们zk要解决的。

Zookeeper致力于提供一个高性能、高可用，且具备严格的顺序访问控制能力的分布式协调服务，是雅虎公司创建，是Hadoop和Hbase的重要组件。

哪些场景需要？

数据发布订阅，负载均衡，命名服务，Master选举，集群管理，配置管理，分布式队列，分布式锁

### 分布式环境特点

分布式系统由独⽴的服务器通过⽹络松散耦合组成的

1. 扩展性

分布式系统最大的特点是可扩展性，它能够适应需求变化而扩展。随着互联网企业的业务规模不断增大，业务变得越来越复杂，并发用户请求越来越多，要处理的数据也越来越多，这个时候企业级应用平台必须能够适应这些变化，支持高并发访问和海量数据处理。分布式系统有良好的可扩展性，可以通过增加服务器的数量来增强分布式系统整体的处理能力，以应对企业的也去增长带来的计算需求。

1. 分布式

分布式系统中的多台计算机都会在空间上随意分布，同时机器的分布情况也会随时变动

1. 对等性

分布式系统中的计算机没有主从之分，既没有控制整个系统的主机，也没有被控制的从机，组成分布式系统的所有计算机节点都是对等的。

1. 并发性

在一个计算机网络中，程序运行过程中的并发性操作是非常常见的行为。例如，同一个分布式系统中的多个节点，可能会并发地操作一些共享的资源，诸如数据库或分布式存储，如何准确并高效地协调分布式并发操作也成了分布式系统架构与设计中最大的挑战之一。

1. 无序性

分布式系统是由一系列在空间上随意分布的多个进程组成的，具有明显的分布性，这些进程之间通过交换信息来进行相互通信。进程之间的消息通信，会出现顺序不一致问题。

### 分布式环境存在的问题

1. 网络通信故障

从集中式向分布式演变的过程中，必然引入了网络因素，而由于网络本身的不可靠性，因此也引入了额外的问题。分布式系统需要在各个节点进行网络通信，因此每次网络通信都会导致最终分布式系统无法顺利完成一次网络通信。

1. 网络分区

当网络发生异常情况，导致分布式系统中部分节点之间的网络延时不断增加，最终导致组成分布式系统的所有节点中，只有部分节点之间能够进行正常通信，而另一些则不能，我们将整个现象称为网络分区，就是俗称的-脑裂-。当网络分区出现时，分布式系统会出现局部小集群，在极端情况下，这些局部小集群会独立完成原本需要整个分布式系统才能完成的功能，包括对数据的事务处理，这就对分布式一致性提出了非常大的挑战。

1. 三态

我们已经了解了在分布式环境下，网络可能会出现各种各样的问题，因此分布式系统的每一次请求与响应，存在特有的三态概念，即**成功、失败、超时**。**超时出现的原因**：1）由于网络原因，该请求并没有被成功的发送到接送方，而是在发送过程中就发生了消息丢失 2）该请求成功的被接收方接收后，并进行了处理，但是在将响应反馈给发送方的过程中，发生了消息丢失现象。

1. 节点故障

节点故障则是分布式环境下的另一个比较常见的问题，指的是组成分布式系统的服务器节点出现的宕机或僵死现象。

1. 分布式事务

因为现在的服务大都是部署在多台机器上，那么就将运行在不同的服务器上的，所以无法通过本地事务的方式来保证ACID

### CAP理论

CAP是Consistency、Availablity和Partition-tolerance的缩写，分别是指

一致性：每次操作都能保证返回的是最新的数据

可用性：任何一个没有发生故障的节点，都会在合理的时间内返回一个正常的结果

分区容忍性：当节点出现网络分区，照样可以提供服务

1. 一致性

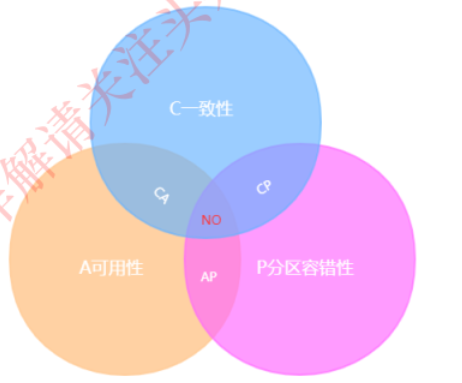
一致性是指数据在多个副本之间是否能够保持一致性的特性。假如现在的多个节点中的数据似乎保持一致的，当执行完某一个更新操作之后，应当要保证系统的数据然后处于一致性的状态，对于一个将数据副本分布在同的分布式节点上，如果对第一个节点的数据进行了更新的操作，并且更新成功之后，却没有让第二个节点得到相应的更新。当外部系统在去调用第二个节点时，获取到的依然是原始的数据，这就是分布式数据不一致的情况。在分布式系统中，如果能够做到针对一个数据项更新操作执行成功之后，所有的用户都可以读到最新的值，那么这样的系统就被认为是具有强一致性的。

1. 可用性

可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内，返回结果。有限的时间内，对于用户的一个操作请求，系统必须能够在指定的时间内返回对应的结果。如果超过了这个时间，就认为系统是不可用的。返回结果是可用性的一个非常重要的指标，它要求系统在完成对用户请求的处理后，返回一个正常的响应结果。正常的响应结果包含成功或失败，而不是一个让用户迷惑的结果。

1. 分区容错性

分布式系统在遇到任何网络分区的故障的时候，仍然需要对外提供满足一致性和可用性的服务



一个分布式系统既然不能同时满足上述的三个需求时，因此在进行对cap定理的应用时，我们就需要抛弃一项。

①选择CA 放弃分区容错性，⽐较简单的⽅式就是把所有的数据都放在⼀个分布式节点 上。那不就⼜成为了单机应⽤了吗？ ②选择CP 放弃可⽤性，⼀旦出现⽹络故障，受到影响的服务需要再等待⼀定时间，因为 系统处于不可⽤的状态。

③选择AP 放弃⼀致性，这⾥所指的⼀致性是强⼀致性，但是确保最终⼀致性。是很多分 布式系统的选择。

⼩结：从cap的定理可以看出，分区容错性是⼀个最基本的要求，因为既然是 ⼀个分布式系统，必然要部署到两个或两个以上的节点上，否则，就不是分布式系统，因此我们只能在⼀致性和可⽤性寻求平衡

### BASE理论

base是Basically Available（基本可⽤）、 Soft state（软状态）和Eventually，consistent（最终⼀致性）三个短语的简写。 base是对cap中⼀致性和可⽤性的权衡的结果。是根据cao理论演变⽽来，核⼼思想是即使⽆法做到强⼀致性，但是每个应⽤根据⾃身的业务特点，采⽤适当的⽅式来使系统达到最终与执⾏。

①基本可⽤

基本可⽤指的是分布式系统出现了不可预知故障的时候，允许损失部分可⽤性。响应时间合理延⻓，功能上适当做服务降级。

②弱状态

弱状态指的是允许系统中的数据存在中间状态，并认为该中间状态不会影响系统的整体可⽤性，即允许在各个节点数据同步时存在延时。

③最终⼀致性

最终⼀致性强调的是系统中所有的数据副本，在经过⼀段时间的同步之后，最终能够达到⼀个⼀致的状态。因此，最终⼀致性的本质是需要系统保证数据最终能够达到⼀致。⽽不需要实时保证系统数据的⼀致性

### 部署节点为奇数?

|  |  |  |
| --- | --- | --- |
| 集群总节点数 | 最少可⽤节点数 | 可容忍失效节点数 |
| 3 | 2 | 1 |
| 4 | 3 | 1 |
| 5 | 3 | 2 |
| 6 | 4 | 2 |
| 2n-1 | n | n-1 |
| 2n | n+1 | n-1 |

总结原因如下：

1. 节约部署服务器资源

2. 容错

3. 防⽌⽹络分区导致脑裂问题

## zookeeper 安装

### zookeeper步骤

1. 前置条件—安装java环境

添加ppa仓库

这个是OpenJDK 8 ppa仓库。

sudo add-apt-repository ppa:openjdk-r/ppa

sudo apt-get update

sudo apt-get install openjdk-8-jdk

设置openjdk版本

sudo update-alternatives --config java

查看java 版本

root@alpha:/home/alpha/share/apache-zookeeper-3.5.5-bin/bin# java -version

openjdk version "1.8.0\_222"

OpenJDK Runtime Environment (build 1.8.0\_222-8u222-b10-1~14.04-b10)

OpenJDK 64-Bit Server VM (build 25.222-b10, mixed mode)

1. 安装maven

# 安装wget http://mirrors.hust.edu.cn/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz

# 解压

　tar -zxvf apache-maven-3.6.3-bin.tar.gz

　# 添加环境变量

　vim /etc/profile

export MAVEN\_HOME=/usr/local/maven-3.6.3

export PATH=PATH:JAVA\_HOME/bin:$MAVEN\_HOME/bin

# 重新加载环境变量

　source /etc/profile

　# 查看是否安装成功

　mvn -v

1. 重新编译zookeeper代码即可

cd /usr/local/zookeeper

cd /zookeeper-server

mvn package -Dmaven.test.skip=true

安装过程中会遇到很多问题--挨个解决吧

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

QuorumPeerMain进程的作⽤

QuorumPeerMain是zookeeper集群的启动⼊⼝类，是⽤来加载配置启动QuorumPeer线程的。

Quorum是法定⼈的意思， Peer是对等的意思，那么QuorumPeer中quorum代表的意思就是每个zookeeper集群启动的时候集群中zookeeper服务数量就已经确定了， zookeeper是基于paxos算法实现的，那是⼀个唯⼀的分布式集群⼀致性算法，在zookeeper中将这⼀算法演绎为集群分布式协调可持续服务。在每个zookeeper的配置⽂件中配置集群中的所有机器列表信息如下：

server.1=localhost:2287:3387

server.2=localhost:2288:3388

server.3=localhost:2289:3389

配置中每个server.X记录代表集群中的⼀个服务， QuorumPeerConfig会构建⼀个QuorumServer对象，其中的server.X中的X代表zookpeer的sid，每个zookeeper都会编辑⾃⼰的sid在dataDir⽬下的myid⽂件中， sid标记每个服务，在快速选举中起作⽤。如果将这个进程关闭的话就会导致服务不可⽤。

### zookeeper的api接口

提供C语言的接口和java的接口，C语言的接口详见：zookeeper-client/zookeeper-client-c目录

1. 在该目录 zookeeper-client/zookeeper-client-c. 执行 ant compile\_jute

2. change directory to zookeeper-client/zookeeper-client-c and do a "autoreconf -if" to bootstrap

3. do a "./configure [OPTIONS]" to generate the makefile.

4. do a "make" or "make install" to build the libraries and install them.

5.to generate doxygen documentation do a "make doxygen-doc"

【注】apache-zookeeper-3.5.5-bin.tar.gz这个bin版本是没有API接口的源码的，需要下载

<https://archive.apache.org/dist/zookeeper/stable/apache-zookeeper-3.5.5.tar.gz>。

这个也要安装，readme

我在编译过程中遇到了这个错误：

root@ntytcp:~/downloads/apache-zookeeper-3.5.5/zookeeper-client/zookeeper-client-c# autoreconf -if

configure.ac:37: warning: macro 'AM\_PATH\_CPPUNIT' not found in library

libtoolize: putting auxiliary files in '.'.

libtoolize: copying file './ltmain.sh'

libtoolize: Consider adding 'AC\_CONFIG\_MACRO\_DIRS([m4])' to configure.ac,

libtoolize: and rerunning libtoolize and aclocal.

libtoolize: Consider adding '-I m4' to ACLOCAL\_AMFLAGS in Makefile.am.

configure.ac:37: warning: macro 'AM\_PATH\_CPPUNIT' not found in library

configure.ac:37: error: possibly undefined macro: AM\_PATH\_CPPUNIT

If this token and others are legitimate, please use m4\_pattern\_allow.

See the Autoconf documentation.

autoreconf: /usr/bin/autoconf failed with exit status: 1

这个错误是因为没有安装cppunit工具。

PS 安装cppunit工具的方法：

1、下载 cppunit-1.12.1.tar.gz

https://sourceforge.net/projects/cppunit/files/cppunit/

2、解压 tar -xvzf cppunit-1.12.1.tar.gz

3、编译安装

./configure LDFLAGS='-ldl'

make

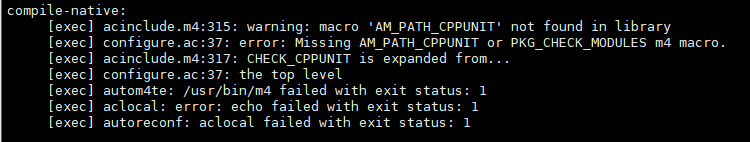
make check

make install

导入lib

运行时要先设置环境变量LD\_LIBRARY\_PATH到cppunit的安装目录，也就是/usr/local/lib，命令如下：

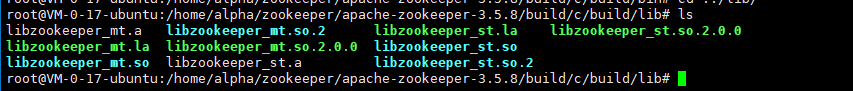
export LD\_LIBRARY\_PATH=/usr/local/lib:$LD\_LIBRARY\_PATH



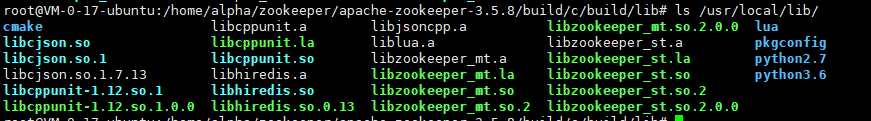
apt-get install libtool

 root@VM-0-17-ubuntu:/home/alpha/zookeeper/apache-zookeeper-3.5.8# apt-get install libcppunit-dev

安装成功后默认的情况下再/usr/local/lib目录下有以下标红的库，其中zookeeper\_mt是多线程库，而zookeeper\_st是单线程库。



将上面的库文件拷贝到



## Zookeeper原理与架构

### 配置文件及基本命令

zoo.cfg配置⽂件

|  |  |
| --- | --- |
| 默认配置 | 描述 |
| tickTime=2000 | zk中最⼩时间单位，默认ms |
| initLimit=10 | 表示follower节点与leader节点同步数据的时间 |
| syncLimit=5 | leader节点和follower节点⼼跳检测的最⼤时间 |
| dataDir=/zk/zookeeper-3 | 表示zk存取快照的⽬录 |
| clientPort=2181 | 客户端与服务端交互端⼝号 |

基本命令

|  |  |
| --- | --- |
| 基本命令 | 描述 |
| stat / | 查看节点状态 |
| get / | 获取节点信息 |
| create / | -s 顺序节点 -e 临时节点 acl权限 |
| set / | 更新节点 |
| delete / | 删除节点，不能删除⼦节点 |
| rmr / | 可以删除相关⼦节点 |
| ls / | 查看⽬录节点 |

节点stat

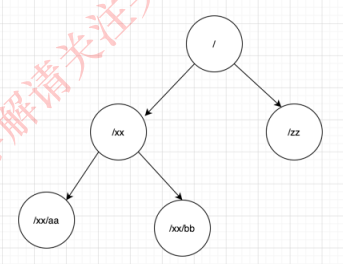
|  |  |
| --- | --- |
| 节点标示 | 描述 |
| cZxid | 创建节点事务标示 |
| ctime | 节点创建⽇期 |
| **mZxid** | 修该节点事务标示 |
| mtime | 修改⽇期 |
| pZxid | ⼦节点更新标示，包含增、删、改操作 |
| cversion | 直接⼦节点版本号(创建、删除) |
| dataVersion | 记录当前节点数据更新版本号 |
| aclVersion | 每次ACL发⽣改变，增加版本号 |
| ephemeralOwner | 增加临时节点，会将客户端sessionId记录在这 |
| dataLength | 节点存储数据的⻓度（B） |
| numChildren | 直接⼦节点的个数 |

### zookeeper特性

节点类型

|  |  |
| --- | --- |
| 特性 | 描述 |
| 顺序⼀致性 | 来⾃客户端的更新请求按照发送的顺序执⾏ |
| 原⼦性 | 更新成功或者失败，没有中间状态结果 |
| 统⼀视图 | ⽆论客户端连接到哪些服务器，都能看到相同的数据 |
| 可靠性 | 客户端连接到服务端后，传送⼀个**watch**，如果服务端发⽣改变，那么客户端能接受到最新的改变 |
| 及时性 | 确保系统客户视图在特定时间范围是最新的 |

### Zookeeper的数据结构



名称是由斜杠（/）分隔的⼀系列路径元素。 ZooKeeper命名空间中的每个节点都由路径进⾏唯⼀标识。

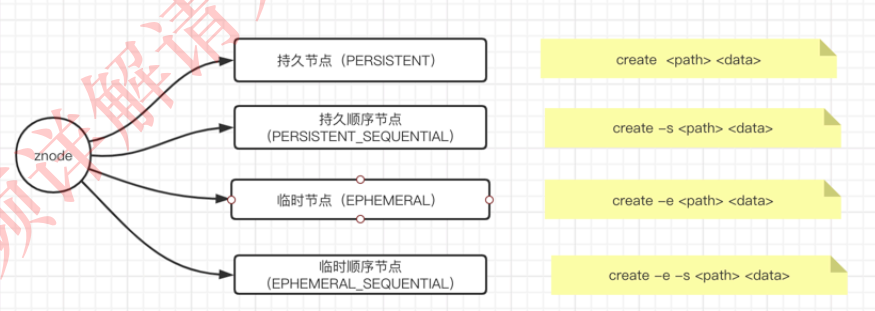
ZooKeeper的层次命名空间与标准⽂件系统不同， ZooKeeper命名空间中的每个节点都可以具有与其关联的数据以及⼦节点。就像拥有⼀个⽂件系统⼀样，该⽂件系统也允许⽂件成为⽬录。每个节点都可以存储数据，但是数据不能超过1M，建议不要⽤节点来存取数据，因为存取⼤数据的话， leader节点和follower或者observer节点同步数据，耗费性能

### Znode类型



|  |  |
| --- | --- |
| 节点类型 | 描述 |
| 持久节点 | 节点会被持久化 |
| 临时节点 | 客户端和服务端建立的一个session，存放在临时节点上，如果客户端断开连接，那么临时节点消失 |
| 顺序节点 | 顺序节点每次创建时，保持顺序性，从1开始 |

通过以上3种类型可以组合成4种节点



Znode类型分为三类

持久节点：节点会被持久化存储

临时节点：客户端断开连接后，zookeeper会自动删除临时节点

顺序节点：每次创建舒徐节点时，zookeeper都会在路径后面自动添加上10位的数字，从1开始，最大到2^32-1

每个顺序节点都有一个单独的计数器，并且单调递增的，有zookeeper的leader实例维护

Znode实际上有四种形式，默认是持久化节点

Znode实际上有四种形式，默认是persistent。

PERSISTENT 持久节点： 通过 create 参数指定为持久节点

PERSISTENT\_SEQUENTIAL（持久顺序节点/s0000000001） ，通过 create -s参数指定为顺序节点

EPHEMERAL 临时节点，通过 create -e 参数指定为顺序节点

EPHEMERAL\_SEQUENTIAL（临时顺序节点/s0000000001） ，通过 create -s -e 参数指定为临时及顺序节点

### zookeeper⻆⾊、节点状态

**ZK**集群服务节点⻆色

ZK集群中的 server 分为三种⻆⾊： leader , follower , observer

那么⻆⾊是在哪配置定义的呢？⾸先我们需要打开zk集群的配置⽂件，⽬录为： zk解压⽬录/zookeeper/conf/zoo.cfg，我们搭建的节点信息如下：

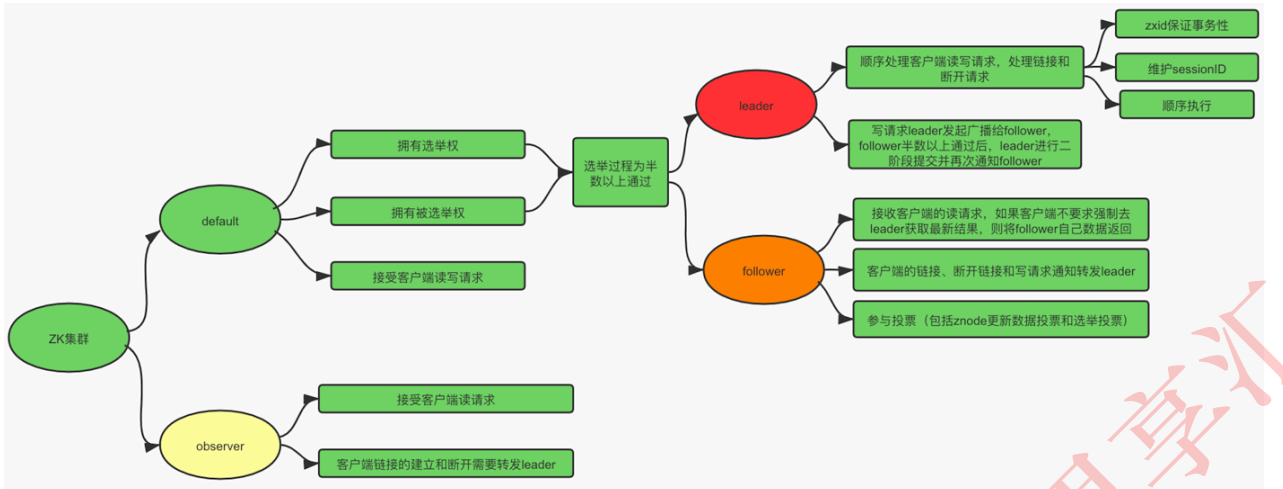
server.1=localhost:2287:3387

server.2=localhost:2288:3388

server.3=localhost:2289:3389

server.4=localhost:2290:3390:observer

下⾯我们⽤⼀张图来总结下ZK集群节点中 leader , follower , observer ⻆⾊的作⽤



1. leader是集群中最重要的角色。主要的工作任务有三项：
2. 事务请求的唯一处理者，保证集群事务的顺序性，它会将每个状态更新请求顺序管理，以便保证整个集群内部消息处理的fifo，遵循了顺序一致性
3. 集群内部各服务器的调度者，负责响应集群的所有对zookeeper数据状态变更的请求。Leader内部维护单调递增的zxid(zookeeper transaction id)，针对客户端的连接，断开连接，节点间的写操作都会分配一个全局唯一的zxid，同时这些操作是原子性的，并且是严格顺序性的，遵循zab原子广播一致性协议完成事务操作。如果客户端有写操作，都会被flower统一转发给leader节点处理，然后leader会下发天壤flower节点投票，如果投票过半数，则提交该事务并通知follower和observer.
4. Leader内部维护session，来自客户端的连接和断开连接，都会被统一follwer或observer转发给leader处理
5. Follwer节点具有选举权
6. 处理客户端非事务请求、转发事务请求给leader服务器
7. 参与事务请求Proposal的投票（需要半数以上服务器通过才能通知leader commit数据，leader发起的天，要求follow投票）
8. 参与leader选举的投票

3. observer没有选举权，observer是zookeeper3.3开始引入的一个全新的角色，从字面来理解，该角色充当了观察者的校色，观察服务器集群中最新状态变化并将这些状态变化同步到observer服务器上，observer工作原理与follwer角色基本一致，而和follwer角色唯一的不同在于observer不参与任何形式的投票，包括事务请求Proposal的投票和leader选举的投票。简单来说，observer服务器只提供非事务请求服务，通常在于不影响集群事务处理能力的前提下提升集群非事务处理的能力。相当于对zk集群完成了水平扩展。

### Zk集群服务节点状态

Looking，竞选状态

Following，随从状态，同步leader状态，参与投票决策提案

Observing，观察状态，同步leader状态，不参与投票决策提案

Leading，领导者状态，发起正常消息的提案

### Zk的znode存储

Zookeeper中的znode数据都是在内存中优先维护和提供服务，当事务被提交以及最终提交都会持久化到磁盘的日志文件中。

### Zookeeper 的内部⽹络拓扑

Zookeeper 在内部节点是怎么进⾏⽹络通信的呢？

我们需要打开zk集群的配置⽂件，⽬录为： zk解压⽬

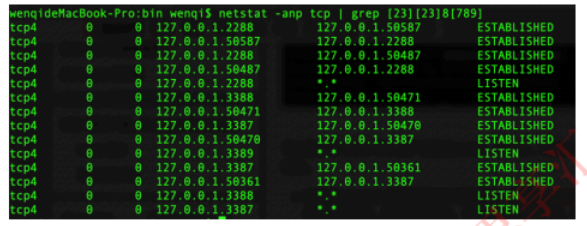
录/zookeeper/conf/zoo.cfg，我们搭建的节点信息如下：

server.1=localhost:2287:3387

server.2=localhost:2288:3388

server.3=localhost:2289:3389

我⽤mac模拟的集群搭建，在mac电脑上查看⽹络TCP命令为： netstat –anp tcp | grep [23][23]8[789]



那么，我们⾸先来分析下这个TCP的连接情况，因为mac和在Linux下命令不同，所以先总结下在Linux系统下查看TCP的命令

netstat -nalp|grep 1883 #查看1883端⼝的连接情况,观察TCP状态图

netstat -nalp|grep 1883|wc -l #查看1883端⼝的客户端连接数

TCP三次握⼿的过程如下：

主动连接端发送⼀个SYN包给被动连接端；

被动连接端收到SYN包后，发送⼀个带ACK和SYN标志的包给主动连接端；

主动连接端发送⼀个带ACK标志的包给被动连接端，握⼿动作完成。

主动关闭端发送⼀个FIN包给被动关闭端；

TCP四次挥⼿的过程如下：

被动关闭端收到FIN包后，发送⼀个ACK包给主动关闭端；

被动关闭端发送了ACK包后，再发送⼀个FIN包给主动关闭端；

主动关闭端收到FIN包后，发送⼀个ACK包，当被动关闭端收到ACK包后，四次挥

⼿动作完成，连接断开。

netstat中的各种状态：

CLOSED 初始（⽆连接）状态。

LISTEN 侦听状态，等待远程机器的连接请求。

SYN\_SEND 在TCP三次握⼿期间，主动连接端发送了SYN包后，进⼊SYN\_SEND状态，等待对⽅的ACK包。

SYN\_RECV 在TCP三次握⼿期间，主动连接端收到SYN包后，进⼊SYN\_RECV状态。

ESTABLISHED 完成TCP三次握⼿后，主动连接端进⼊ESTABLISHED状态。此时， TCP连接已经建⽴，可以进⾏通信。

FIN\_WAIT\_1 在TCP四次挥⼿时，主动关闭端发送FIN包后，进⼊FIN\_WAIT\_1状态。

FIN\_WAIT\_2 在TCP四次挥⼿时，主动关闭端收到ACK包后，进⼊FIN\_WAIT\_2状态。

TIME\_WAIT 在TCP四次挥⼿时，主动关闭端发送了ACK包之后，进⼊TIME\_WAIT状态，等待最多MSL时间，让被动关闭端收到ACK包。

CLOSING 在TCP四次挥⼿期间，主动关闭端发送了FIN包后，没有收到对应的ACK包，却收到对⽅的FIN包，此时，进⼊CLOSING状态。

CLOSE\_WAIT 在TCP四次挥⼿期间，被动关闭端收到FIN包后，进⼊CLOSE\_WAIT状态

LAST\_ACK 在TCP四次挥⼿时，被动关闭端发送FIN包后，进⼊LAST\_ACK状态，等待对⽅的ACK包。

状态分类：

主动连接端可能的状态有： CLOSED、 SYN\_SEND、 ESTABLISHED。

主动关闭端可能的状态有： FIN\_WAIT\_1、 FIN\_WAIT\_2、 TIME\_WAIT。

被动连接端可能的状态有： LISTEN、 SYN\_RECV、 ESTABLISHED。

被动关闭端可能的状态有： CLOSE\_WAIT、 LAST\_ACK、 CLOSED。

回归正题，我们观察到现在的链接状态有

1） LISTEN：侦听状态，等待远程机器的连接请求；

2） ESTABLISHED ：完成TCP三次握⼿后，主动连接端进⼊ESTABLISHED状态。此时， TCP连接已经建⽴，可以进⾏通信。因为我们本地模拟3台进⾏搭建的zk集群，所以展示的ip都是本地的，上图中第4列为作为服务端的，等待其他客户端链接的ip和端⼝号，

我们先来分析配置⽂件zo o.cfg中的第2列端⼝号，

server.1=localhost:2287:3387

server.2=localhost:2288:3388

server.3=localhost:2289:3389

## Paxos 算法

### 分布式一致性算法paxos

它是一个基于消息传递的一致性算法，近几年被广泛应用于分布式计算中，被认为到目前为止唯一的分布式一致性算法，其它的算法都是paxox的改进或简化。有一个问题要提一下，paxos有一个前提没有拜占庭将军问题。就是说paxos只有在一个可信的环境中才能成立，整个环境是不会被入侵所破坏的。

现在我们举一种现实场景的例子，现在居住小区基本都会有居委会，居委会由很多的楼长组成，这些楼长的总数是一定的，不能更改。小区的管理制度都是需要一个提议，每个提议都有一个唯一的提议编号，整个编号只能是顺序增加的，不能倒退。每个提议都需要超过半数的楼长统一了才可以生效。楼长将会按照提议编号一个一个顺序的表决是否通过该提议，为了保证每次开会的进度，错过表决的不能再次表决，错过的提议如果想要表决，那么将不会被大家理睬，并通知提议人，你的这个提议编号已经错过了，我们正在提议的内容是啥啥，每个代表都在自己的笔记本上记录着提议的编号，并不断的提议编号有延迟的情况，不能保证所有的楼长笔记本上的提议编号都是相同的。但是会有一个目标：保证所有的楼长对于提议能达成一致的看法。

比如说，假设现在开始开会，所有楼长开始记事本上面记录的编号都是0。有一个楼长发了一个提议：将物业费调整为3元/平。他首先看了下记事本，嗯当前提议的编号是0，那么我的提议编号是1，于是他给所有的楼长发消息：1号提议，将物业费调整为3元/平。其他楼长收到消息后查了下记事本，当前提议编号是0，这个提议可以接受，于是他记录下这个提议并恢复：我接受你的1号提议，同时他在记事本上记录：当前提议编号是1.发起的提议的楼长收到了超过半数的回复，立即给所有人发通知：1号提议生效！收到通知的楼长会修改自己记事本上的提议编号，将1号提议由记录改成正式提议，当有居民问物业费时，他会查看通过提议列表，并告知对方：物业费为3元/平

现在看冲突的解决：假设总共有三个楼长，R1，R2同时发起了一个提议：1号提议，物业费调整为多少合适呢。R1想设为3元/平，R2想设为4元/平。结果R3先收到了R1的提议，于是他做了和前面同样的操作。紧接着他又收到了R2的提议，结果他一查记事本，这个提议的编号小于等于我的当前编号1，于是他拒绝了这个提议:对不起，这个提议先前提过了。于是R2的提议被拒绝，R1正式发布了提议：1号提议生效。R2向R1或者R3打听并更新了1号通过提议的内容，然后他可以选择继续发起自己的提议，但是编号增加为2的提议。

还有就是居委会一般都会选出一个会长，类似于zk集群中的leader，其实leader的概念也应该属于paxos的范畴的。如果楼长投票平等，在某种情况下会由于提议的冲突而产生一个活锁：每个楼长都坚持自己的提议

解决方案：在所有的楼长中在选择出一个会长，只有会长有权发出提议，楼长的提议必须发给会长，由会长来提出。

以上是举了⼀个现实场景的例⼦，我们看下和ZK集群中的概念是怎么对应的？

zk集群：全体居⺠

提议： znode数据改变

正式提议：事务提交的znode数据

提议编号： zxid 事务id

leader：会⻓（myid选出）

follower：楼⻓

除楼⻓外居⺠： observer

模拟：

1. 查询操作

居民A到某个楼长那里询问某条正式提议的情况，楼长毫不犹豫的拿出记事本，查阅生效的提议并告诉他结果，同时声明：我的数据不一定是最新的。你想要最新的数据，没问题，等着，等我找会长sync下在告诉你。

1. 修改操作

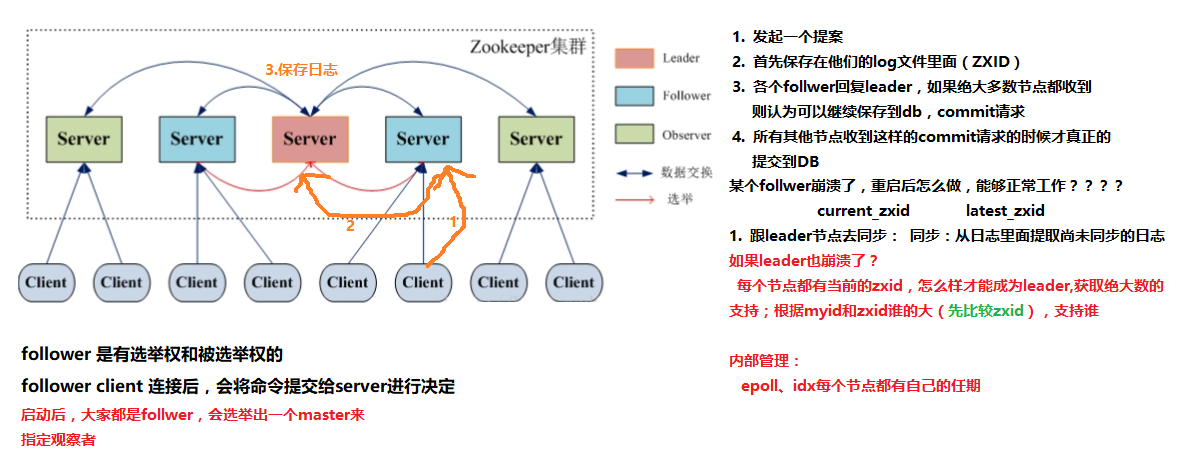
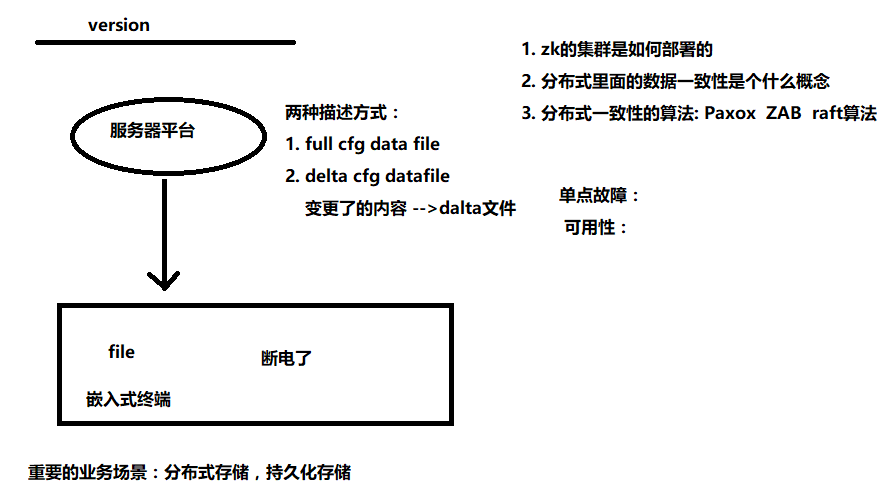
居⺠A(Client)到某个楼⻓(ZK Server)那⾥交物业费100元，楼⻓让他在办公室等着，⾃⼰将问题反映给了会⻓，会⻓询问所有楼⻓的意⻅，多数楼⻓表示物业费可以收，于是会⻓发表声明，所有物业费都可以交给我统⼀收取了哈，于是物业费从1000变为1100，并将收据给了楼⻓，转交给了交钱的居⺠A

1. 选举操作

会⻓由于要搬家，于是卸任了，楼⻓接⼆连三的发现联系不上会⻓，于是各⾃发表声明，推选新的会⻓，会⻓⼤选期间居委会暂停服务，临时不处理业务。

选举原理：

<https://blog.csdn.net/alpha_love/article/details/108175863>



## ZAB协议

Zab协议的全程是zookeeper atomic broadcast 。zookeeper是通过zab协议来保证分布式事物的最终一致性

Zab协议是为分布式协调服务zookeeper专门设计的一种支持崩溃恢复的原子广播协议，是zookeeper保证数据一致性的核心算法。Zab借鉴了paxox算法，但又不像paxos那样，是一种通用的分布式一致性算法。它是特别为zookeeper设计的支持崩溃恢复的原子广播协议。

在zookeeper中主要依赖zab协议来实现数据一致性，基于该协议，zk实现了一种主备模型的系统架构来保证集群中各个副本之间数据的一致性。这里的主备系统架构摩西那个，就是指只有一台服务端leader负责处理外部的写事务请求，当服务器数据的状态发生变更后，集群采用zab原子广播协议，以事务提案prosal的形式广播到所有的副本进程上。Zab协议能够保证一个全局的变更序列，即可以为每一个事务分配一个全局的递增编号zxid，然后leader客户端将数据同步到其他follwer节点。

Zookeeper客户端会随机的连接到zookeeper集群中的一个节点，如果是读请求，就直接从当前节点中读取数据；如果是写请求，那么节点就会向leader提交事务，leader接收到事务提交，会广播该事务，只要超过半数节点写入成功，该事务就会被提交。

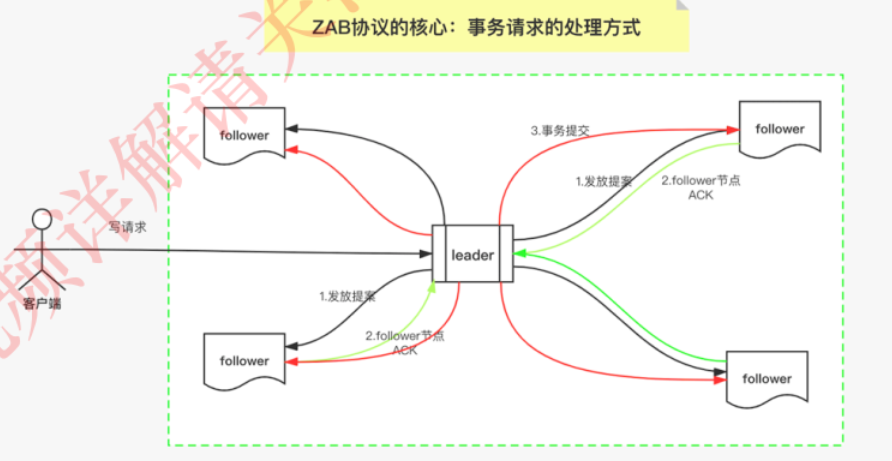
Zab协议原理：

Zab协议要求每个leader都要经历三个阶段：发现，同步，广播。

1. **发现：**要求zookeeper集群必须选举出一个leader进程，同时leader会维护一个follwer可用客户端列表。将来客户端可以和这些follwer节点进行通信。
2. **同步：**leader要负责将本身的数据于follwer完成同步，做到多副本存储，这样也似乎体现了cap中的高可用和分区容错ap模型。Follower将队列中未处理完的请求消费完成后，写入本地事务日志中。
3. **广播：**leader可以接受客户端新的事务prosal请求，将新的prosal请求广播给所有的follower。

### Zab协议的核心

Zab协议的核心：定义了事务请求的处理方式



1. 所有的失去请求必须由一个全局唯一的服务器来协调处理，这样的服务器被叫做leader服务器。其他剩余的服务器则是follow服务器。
2. Leader服务器负责将一个客户端事务请求，转换成一个事务proposal，并将该proposal分发给集群中的follower服务器，也就是向所有follower节点发送数据广播请求（或数据复制）
3. 分发之后leader服务器需要等待所有follower服务器进行了正确的反馈后（ack请求），在zab协议中，只要超过半数的follower的ack请求），那么leader就会再次向所有的follwer服务器发送commit消息，要求其将上一个事务proposal进行提交。

Zab协议实现的作用：

1. 使用一个单一的主进程来接收并处理客户端的事务请求，并采用了zab的原子广播协议，将服务器数据的状态变更以事务的状态变更以事务的形式广播到所有的副本进程上去。
2. 保证一个全局的变更序列被顺序引用，zookeeper是一个树形结构，很多操作都要先检查才能确定是否可以执行，比如P1的事务t1可能是创建节点/a，P2可能是创建节点/a/b，只有先创建了父节点/a，才能创建子节点/a/b，为了保证这一点，zab要保证同一个leader发起的事务要按顺序被apply，同时还要保证只有先前的leader的事务被apply，新选举出来的leader才能再次发起事务。
3. 当主节点leader出现异常的时候，整个zk集群依旧能正常工作。

### 崩溃恢复 消息广播

Zab协议包括两种基本的模式： 崩溃恢复和消息广播

1. 协议过程：

当整个集群启动过程中，当leader服务器出现网络中断、崩溃退出或重启异常时，zab协议就会崩溃恢复模式，选举产生新的leader。

当选举产生了新的leader，同时集群中有过半的机器与该leader服务器完成了状态之后，zab协议就会退出崩溃恢复模式，进入消息广播模式。

这时，如果有一台遵循zab协议的服务器加入集群，因为此时集群中已经存在一个leader服务器在广播消息，那么该新加入的服务器自动进入恢复模式：找到leader服务器，并且完成数据同步。同步完成后，作为新的follower一起参与到消息广播流程中。

1. 协议状态切换

当leader出现崩溃退出或者机器重启，亦或是集群中不存在超过半数的服务器与leader保存正常通信，zab就会再一次进入崩溃恢复，发起新一轮leader选举并实现数据同步。同步完成后又会进入消息广播模式，接收事务请求。

1. 保证消息有序

在整个消息广播中，leader会将每一个事务请求转换成对应的propsoal来进行广播，并且在广播事务propoasl之前，leader服务器会首先为整个事务分配一个全局单调递增的唯一ID，称之为事务ID，由于zab协议需要保证每一个消息的严格的顺序关系，因此必须将每一个Proposal按照其zxid的信后顺序进行排序和处理。

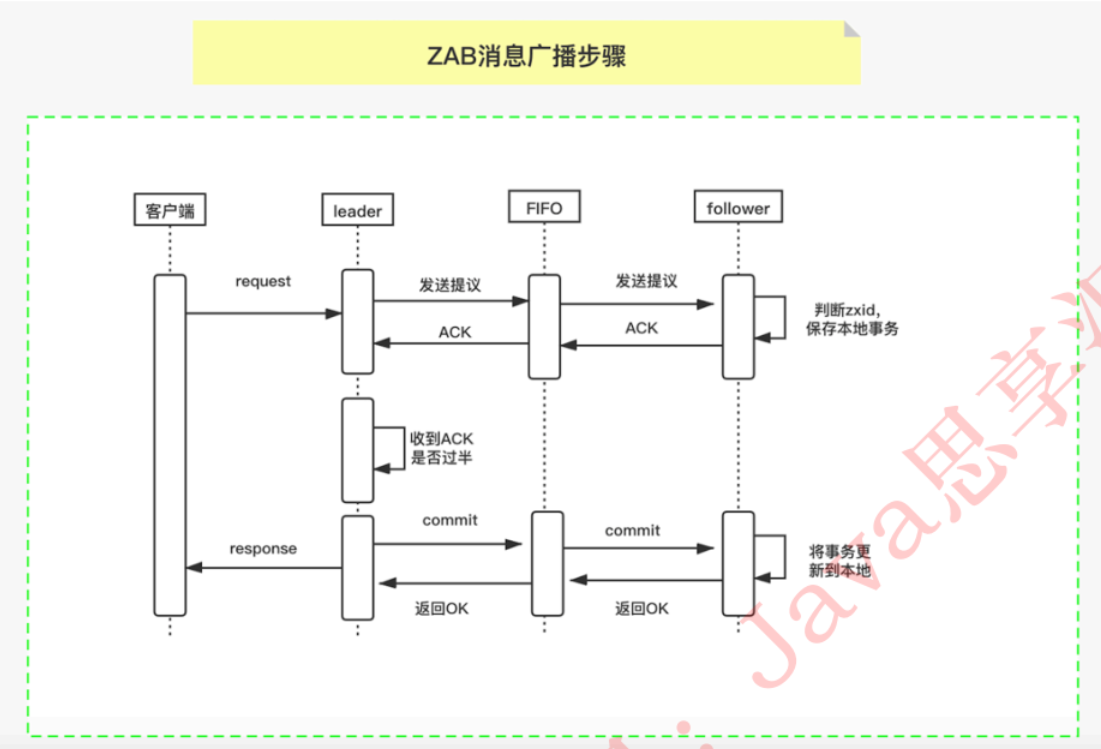
##### 消息广播：

1. 在zookeeper集群中，数据副本的传递策略就是采用消息广播模式

Zookeeper中数据副本的同步方式为二段提交相似，但是却又不同。二段提交要求协调者必须等到所有的参与者全部反馈ack确认消息后，在发送commit消息。要求所有的参与者要么全部成功，要么全部失败。二段提交会产生严重的阻塞问题。

1. Zab协议中leader等待follower的ack反馈消息是指 只要半数以上的follwer成功反馈即可，不需要收到全部follwer反馈。

**消息广播：**



1. 客户端发起一个写操作请求
2. Leader服务器将客户端的i请求转换为事务Proposal提案，同时为每个Proposal分配一个全局的ID，即zixd
3. Leader服务器为每一个follwer服务器分配一个单独的队列，然后将需要广播的Proposal依次存放到队列中，并且根据FIFO策略进行消息发送
4. Follower接收到Proposal后，会首先将其以事务日志的方式写入本地磁盘中，写入成功后向leader反馈一个ack响应消息
5. Leader接收到超过半数以上follwer的ack响应消息后，即认为消息发送成功，可以发送commit消息。
6. Leader向所有follwer广播commit消息，同时自身也会完成事务提交。Follwer接收到commit消息后，会将上一条事务提交，并将执行成功结果返回leader
7. 最终leader将处理结果返回给客户端。

Zookeeper采用zab协议的核心，就是只要有一台服务器提交了proposal，就要确保所有的服务器最终都能正确提交proposal，这也是CAP/BASE实现最终一致性的体现。

Leader服务器与每一个服务器之间都维护了一个单独的FIFO队列消息队列进行收发消息，使用队列消息可以做到异步解耦。Leader和follwer之间只需要往队列中发消息即可。如果使用同步的方式会引起阻塞，性能要下降很多。

##### 崩溃恢复：

一旦leader服务器出现崩溃或者由于网络原因导致leader服务器失去了与过半follwer的联系，那么就会进入崩溃恢复模式。

在zab协议中，为了保证程序的正确运行，整个恢复过程结束后需要选举出一个新的leader服务器。因此zab协议需要一个高效且可靠的leader选举算法，从而保证能够快速选举出新的leader。

Leader选举算法不仅仅需要让leader自己知道自己已经被选举为leader，同时还需要让集群中的所有其他机器也能够快速感知到自选举产生的新leader服务器。

**崩溃服务主要包括两部分： leader选举和数据恢复**

1. 假设两种异常情况：

一个事务在leader上提交了，并且过半的follwer都响应ack了，但是leader在commit消息发出之前挂了。

1. 假设一个事务在leader提出后，leader挂了

对于以上要恢复的数据状态需要遵循两个原则：

* 1. 已被处理过的消息不能丢

当leader收到超过半数follwer的acks后，就向各个follwer广播commit消息，批准各个server执行该写操作事务。当各个server在接收到leader的commit消息后就会在本地执行该写操作，然后会向客户端响应写操作成功。

但是如果在非全部follwer收到commit消息之前leader就挂了，这将导致一种后果：部分server已经执行了该事务，而部分server尚未收到commit消息，所以并没有执行该事务，当新的leader被选举出，集群经过恢复模式后需要保证所有的server上都执行了哪些已经被部分server执行过的事务。

* 1. 被丢弃的消息不能再现

当在leader新事物已经通过，其已经将该事务更新到了本地，但所有follwer还都没有收到commit之前，leader宕机（比前面叙述的宕机更早），此时，所有follwer根本就不知道该Proosal的存在。当新的leader选举出来，整个集群进入正常服务状态后，之前挂了的leader主机重新启动并注册称为了follwer。若那个别人根本不知道的Proposal还保留在那个主机，那么其数据就会比其它主机多出了内容，导致整个系统状态的不一致。所以，该Proposal应该被丢弃掉，类似这样应该被丢弃的事务，是不能再次出现在集群中的，应该被清楚。

ZAB数据同步过程中，如何处理需要丢弃的Proposal?

在ZAB的事务编号中zxid设计中，zxid是一个64位的数字。

其中低32位可以看成一个简单的单增计数器，针对客户端每一个事务请求，leader在产生的新的Proposal事务时，都会对该计数器加1。而高32位则代表了leader周期的epoch编号。

Epoch编号可以理解为当前集群所处的年代，或者周期。每次leader变更之后都会在epoch的基础上加1，这样旧的leader崩溃恢复之后，其它follwer也不会在听它的了，因为foller只服从epoch最高的leader命令。

每当选举产生一个新的leader，就会从这个leader服务器上取出本地事务日志中最大编号Proposal的zxid，并从zxid中解析得到对应的epoch值，并将低32位数字归零，由0开始重新生成zxid。

Zab协议通过epoch编号来区分leader变化周期，能够有效避免不同的leader错误的使用了相同的zxid编号提出了不一样的Proposal异常情况。

基于以上策略当一个包含了上一个leader周期中尚未提交过的事务Proposal的服务器启动时，当这台机器加入集群中，以foller角色连上leader服务器后，leader服务器会根据自己服务器上最后提交的proposal来和follower服务器的proposal进行对比，对比的结果肯定是leader要求和follower进行一个回退操作，回退到一个确实已经被集群中过半机器commit的最新proposal。

##### 三类角色

为了避免zookeeper的单点问题，zk也是以集群的形式出现的，zk集群中的角色主要有以下三类：

Leader： 接收和处理客户端的请求；zk集群中事务请求的唯一处理者，并负责发起决议和投票，然后将通过事务请求在本地进行处理后，将处理结果同步给集群中的其它主机。

Follower：接收和处理客户端的读请求；将事务请求转给leader；同步leader中的数据；当leader挂了，参与leader的选举(具有选举权和被选举权)

Observer：就是没有选举权和被选举权，且没有投票权的Follower。做zk集群中的读压力比较到，则需要增加Observer，最好不要增加Follwer。因为增加Follwer将会增大投票与统计选票的压力，降低写操作效率，及leader选举的效率。

三个数据：

在ZAB中有三个很重要的数据：

Zxid： 是一个64位长度的Long类型。其中高32位表示epoch，低32位表示xid

Epoch：每个leader都会具有一个不同的epoch，用于区分不同的时期

Xid：事务id，是一个流水号，每次版本号更替，即leader更换，从0开始递增

每当选举产生一个新的leader，就会从整个leader服务器上取出本地事务日志最大编号Proposal的zxid，并从zxid中解析得到epoch编号，然后在对其加1，之后该编号就作为新的epoch值，并将低32位数字归零，由0开始重新生成zixd。

##### Zab四阶段

myid:这是 zk 集群中服务器的唯⼀标识，称为 myid。例如，有三个 zk 服

务器，那么编号分别 是 1,2,3。逻辑时钟:逻辑时钟， Logicalclock，是⼀个整型数，该概念在选举时称为logicalclock，⽽在选举结 束后称为 epoch。即 epoch 与 logicalclock 是同⼀个值，在不同情况下的不同名称。

1).选举阶段（Leader Election）

节点在⼀开始都处于选举节点，只要有⼀个节点得到超过半数节点的票数，它就可以当选准 Leader，只有到达第三个阶段（也就是同步阶段），这个准Leader 才会成为真正的 Leader。Zookeeper 规定所有有效的投票都必须在同⼀个 轮次 中，每个服务器在开始新⼀轮投票时，都会对⾃⼰维护的 logicalClock 进⾏⾃增操作。每个服务器在⼴播⾃⼰的选票前，会将⾃⼰的投票箱（recvset）清空。该投票箱记录了所收到的选票。

例如： Server\_2 投票给 Server\_3， Server\_3 投票给 Server\_1，则Server\_1的投票箱为(2,3)、 (3,1)、 (1,1)。（每个服务器都会默认给⾃⼰投票）前⼀个数字表示投票者，后⼀个数字表示被选举者。票箱中只会记录每⼀个投票者的最后⼀次投票记录，如果投票者更新⾃⼰的选票，则其他服务器收到该新选票后会在⾃⼰的票箱中更新该服务器的选票。 思考下：这⾥在实现中应该

怎么实现呢？ 等我们分析源码时就可以看到，⾮常的巧妙

这⼀阶段的⽬的就是为了选出⼀个准 Leader ，然后进⼊下⼀个阶段。

2). 发现阶段（Descovery）

在这个阶段， Followers 和上⼀轮选举出的准 Leader 进⾏通信，同步Followers 最近接收的事务 Proposal 。这个阶段的主要⽬的是发现当前⼤多数节点接收的最新 Proposal，并且准Leader ⽣成新的 epoch ，让 Followers 接收，更新它们的acceptedEpoch。

3). 同步阶段（Synchronization)

同步阶段主要是利⽤ Leader 前⼀阶段获得的最新 Proposal 历史，同步集群中所有的副本。只有当 quorum（超过半数的节点） 都同步完成，准 Leader 才会成为真正的Leader。 Follower 只会接受 zxid ⽐⾃⼰ lastZxid ⼤的 Proposal。

4). ⼴播阶段（Broadcast）

到了这个阶段， Zookeeper 集群才能正式对外提供事务服务，并且 Leader 可以进⾏消息⼴播。同时，如果有新的节点加⼊，还需要对新节点进⾏同步。 需要注意的是， Zab 提交事务并不像 2PC ⼀样需要全部 Follower 都 Ack，只需要

得到 quorum（超过半数的节点）的Ack 就可以。

ZAB与Paxos

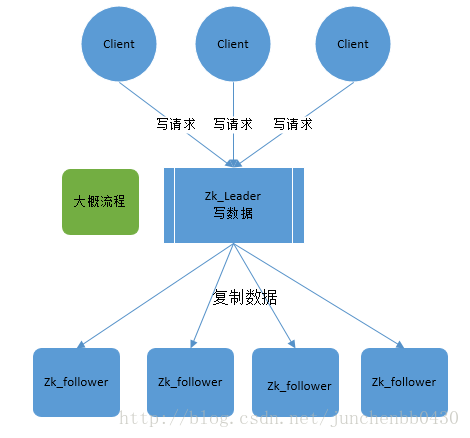
Paxos算法的确是不关⼼请求之间的逻辑顺序，⽽只考虑数据之间的全序，但很少有⼈直接使⽤paxos算法，都会经过⼀定的简化、优化。Google的粗粒度锁服务Chubby的设计开发者Burrows曾经说过： “所有⼀致性协议本质上要么是Paxos要么是其变体”。这句话还是有⼀定道理的， ZAB本质上就是Paxos的⼀种简化形式。

ZAB⼩结

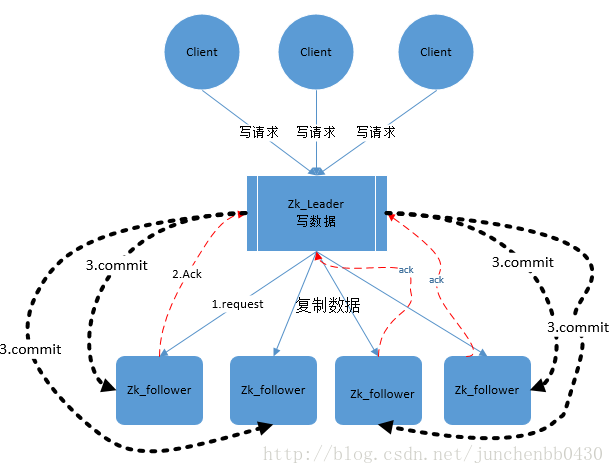
ZAB协议是个巧妙的设计，⽐如：为了加快收敛速度避免活锁引发的竞争引⼊了Leader⻆⾊，在正常情况下最多只有⼀个参与者扮演Leader⻆⾊，其他参与者扮演Acceptor；在这种优化算法中，只有Leader可以提出议案，从⽽避免了竞争使得算法能够快速地收敛⽽趋于⼀致；⽽为了保证Leader的健壮性，⼜引⼊了Leader选举，再考虑到同步的阶段，提出了消息⼴播和崩溃初始化同步以及恢复模式的两个原则

### ZAB协议

1. ZAB协议是专门为zookeeper实现分布式协调功能而设计。zookeeper主要是根据ZAB协议是实现分布式系统数据一致性。
2. zookeeper根据ZAB协议建立了主备模型完成zookeeper集群中数据的同步。这里所说的主备系统架构模型是指，在zookeeper集群中，只有一台leader负责处理外部客户端的事物请求(或写操作)，然后leader服务器将客户端的写操作数据同步到所有的follower节点中。



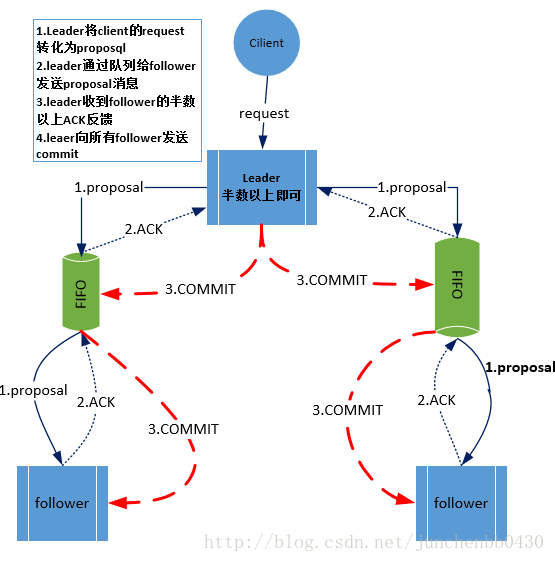
1. ZAB的协议核心是在整个zookeeper集群中只有一个节点即Leader将客户端的写操作转化为事物(或提议proposal)。Leader节点再数据写完之后，将向所有的follower节点发送数据广播请求(或数据复制)，等待所有的follower节点反馈。在ZAB协议中，只要超过半数follower节点反馈OK，Leader节点就会向所有的follower服务器发送commit消息。即将leader节点上的数据同步到follower节点之上。



1. ZAB协议中主要有两种模式，第一是消息广播模式；第二是崩溃恢复模式

### 消息广播模式

1. 在zookeeper集群中数据副本的传递策略就是采用消息广播模式。zookeeper中数据副本的同步方式与二阶段提交相似但是却又不同。二阶段提交的要求协调者必须等到所有的参与者全部反馈ACK确认消息后，再发送commit消息。要求所有的参与者要么全部成功要么全部失败。二阶段提交会产生严重阻塞问题。
2. ZAB协议中Leader等待follower的ACK反馈是指”只要半数以上的follower成功反馈即可，不需要收到全部follower反馈”
3. 图中展示了消息广播的具体流程图



1. **zookeeper中消息广播的具体步骤如下**：

4.1. 客户端发起一个写操作请求

4.2. Leader服务器将客户端的request请求转化为事物proposql提案，同时为每个proposal分配一个全局唯一的ID，即ZXID。

4.3. leader服务器与每个follower之间都有一个队列，leader将消息发送到该队列

4.4. follower机器从队列中取出消息处理完(写入本地事物日志中)毕后，向leader服务器发送ACK确认。

4.5. leader服务器收到半数以上的follower的ACK后，即认为可以发送commit

4.6. leader向所有的follower服务器发送commit消息。

1. **zookeeper采用ZAB协议的核心就是只要有一台服务器提交了proposal，就要确保所有的服务器最终都能正确提交proposal。这也是CAP/BASE最终实现一致性的一个体现**。
2. **leader服务器与每个follower之间都有一个单独的队列进行收发消息，使用队列消息可以做到异步解耦。leader和follower之间只要往队列中发送了消息即可。如果使用同步方式容易引起阻塞。性能上要下降很多**。

### 崩溃恢复

1. zookeeper集群中为保证任何所有进程能够有序的顺序执行，只能是leader服务器接受写请求，即使是follower服务器接受到客户端的请求，也会转发到leader服务器进行处理。
2. 如果leader服务器发生崩溃，则zab协议要求zookeeper集群进行崩溃恢复和leader服务器选举。
3. ZAB协议崩溃恢复要求满足如下2个要求：

3.1. **确保已经被leader提交的proposal必须最终被所有的follower服务器提交**。

3.2. **确保丢弃已经被leader出的但是没有被提交的proposal**。

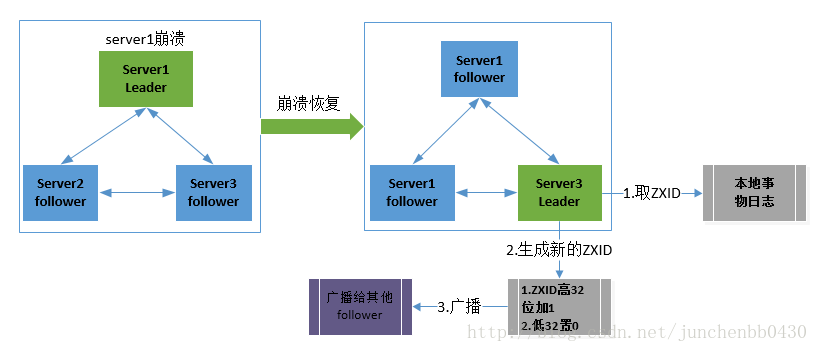
1. 根据上述要求，新选举出来的leader不能包含未提交的proposal，即新选举的leader必须都是已经提交了的proposal的follower服务器节点。同时，新选举的leader节点中含有最高的ZXID。这样做的好处就是可以避免了leader服务器检查proposal的提交和丢弃工作。
2. leader服务器发生崩溃时分为如下场景：

5.1. leader在提出proposal时未提交之前崩溃，则经过崩溃恢复之后，新选举的leader一定不能是刚才的leader。因为这个leader存在未提交的proposal。

5.2 leader在发送commit消息之后，崩溃。即消息已经发送到队列中。经过崩溃恢复之后，参与选举的follower服务器(刚才崩溃的leader有可能已经恢复运行，也属于follower节点范畴)中有的节点已经是消费了队列中所有的commit消息。即该follower节点将会被选举为最新的leader。剩下动作就是数据同步过程。

### 数据同步

1. 在zookeeper集群中新的leader选举成功之后，leader会将自身的提交的最大proposal的事物ZXID发送给其他的follower节点。follower节点会根据leader的消息进行回退或者是数据同步操作。最终目的要保证集群中所有节点的数据副本保持一致。
2. 数据同步完之后，zookeeper集群如何保证新选举的leader分配的ZXID是全局唯一呢？这个就要从ZXID的设计谈起。

2.1 ZXID是一个长度64位的数字，其中低32位是按照数字递增，即每次客户端发起一个proposal,低32位的数字简单加1。高32位是leader周期的epoch编号，至于这个编号如何产生(我也没有搞明白)，每当选举出一个新的leader时，新的leader就从本地事物日志中取出ZXID,然后解析出高32位的epoch编号，进行加1，再将低32位的全部设置为0。这样就保证了每次新选举的leader后，保证了ZXID的唯一性而且是保证递增的。 

### ZAB协议原理

1. ZAB协议要求每个leader都要经历三个阶段，即发现，同步，广播。
2. **发现**：即要求zookeeper集群必须选择出一个leader进程，同时leader会维护一个follower可用列表。将来客户端可以这follower中的节点进行通信。
3. **同步**：leader要负责将本身的数据与follower完成同步，做到多副本存储。这样也是体现了CAP中高可用和分区容错。follower将队列中未处理完的请求消费完成后，写入本地事物日志中。
4. **广播**：leader可以接受客户端新的proposal请求，将新的proposal请求广播给所有的follower。

### Zookeeper设计目标

1. zookeeper作为当今最流行的分布式系统应用协调框架，采用zab协议的最大目标就是建立一个高可用可扩展的分布式数据主备系统。即在任何时刻只要leader发生宕机，都能保证分布式系统数据的可靠性和最终一致性。
2. 深刻理解ZAB协议，才能更好的理解zookeeper对于分布式系统建设的重要性。以及为什么采用zookeeper就能保证分布式系统中数据最终一致性，服务的高可用性。

## 分布式事务

### 什么是事务

什么是事务？举个生活中的例子：你去小卖铺买东西，“一手交钱，一手交货”就是一个事务的例子，交钱和交货必

须全部成功，事务才算成功，任一个活动失败，事务将撤销所有已成功的活动。

明白上述例子，再来看事务的定义：

事务可以看做是一次大的活动，它由不同的小活动组成，这些活动要么全部成功，要么全部失败

### 本地事务

在计算机系统中，更多的是通过关系型数据库来控制事务，这是利用数据库本身的事务特性来实现的，因此叫数据

库事务，由于应用主要靠关系数据库来控制事务，而数据库通常和应用在同一个服务器，所以基于关系型数据库的

事务又被称为本地事务。

回顾一下数据库事务的四大特性 ACID：

A（Atomic）：原子性，构成事务的所有操作，要么都执行完成，要么全部不执行，不可能出现部分成功部分失

败的情况。

C（Consistency）：一致性，在事务执行前后，数据库的一致性约束没有被破坏。比如：张三向李四转100元，

转账前和转账后的数据是正确状态这叫一致性，如果出现张三转出100元，李四账户没有增加100元这就出现了数

据错误，就没有达到一致性。

I（Isolation）：隔离性，数据库中的事务一般都是并发的，隔离性是指并发的两个事务的执行互不干扰，一个事

务不能看到其他事务运行过程的中间状态。通过配置事务隔离级别可以避脏读、重复读等问题。

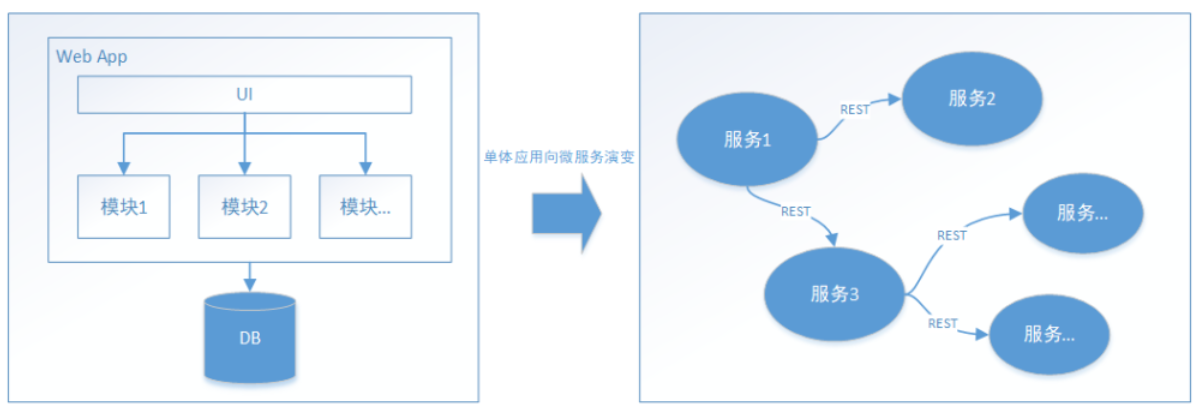
D（Durability）：持久性，事务完成之后，该事务对数据的更改会被持久化到数据库，且不会被回滚。

数据库事务在实现时会将一次事务涉及的所有操作全部纳入到一个不可分割的执行单元，该执行单元中的所有操作

要么都成功，要么都失败，只要其中任一操作执行失败，都将导致整个事务的回滚

### 分布式事务

随着互联网的快速发展，软件系统由原来的单体应用转变为分布式应用，下图描述了单体应用向微服务的演变：



分布式系统会把一个应用系统拆分为可独立部署的多个服务，因此需要服务与服务之间远程协作才能完成事务操作，这种分布式系统环境下由不同的服务之间通过网络远程协作完成事务称之为分布式事务，例如用户注册送积分事务、创建订单减库存事务，银行转账事务等都是分布式事务。

我们知道本地事务依赖数据库本身提供的事务特性来实现，因此以下逻辑可以控制本地事务：



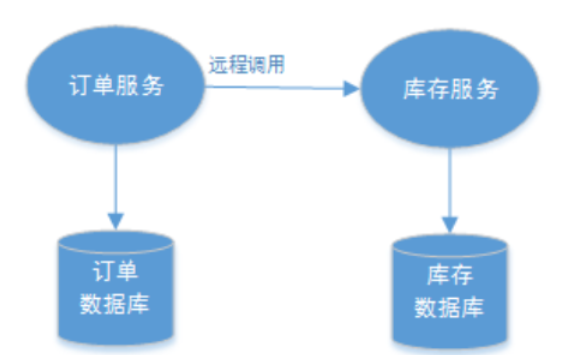
但是在分布式环境下，会变成下边这样：



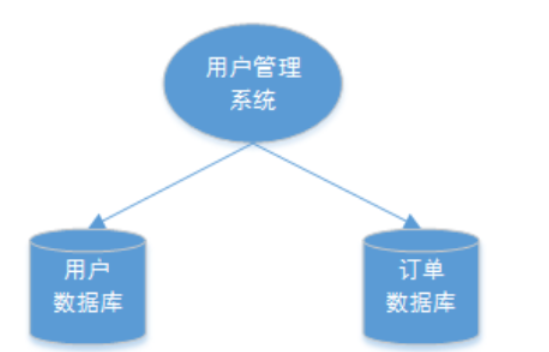
可以设想，当远程调用让李四增加金额成功了，由于网络问题远程调用并没有返回，此时本地事务提交失败就回滚了张三减少金额的操作，此时张三和李四的数据就不一致了。因此在分布式架构的基础上，传统数据库事务就无法使用了，张三和李四的账户不在一个数据库中甚至不在一个应用系统里，实现转账事务需要通过远程调用，由于网络问题就会导致分布式事务问题。

### 分布式事务产生的场景

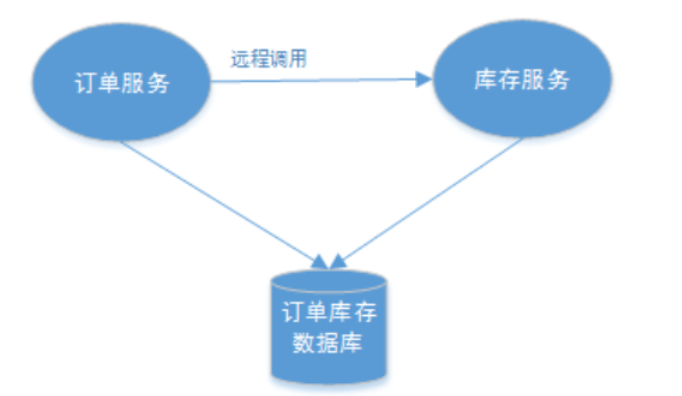
1、典型的场景就是微服务架构 微服务之间通过远程调用完成事务操作。 比如：订单微服务和库存微服务，下单的同时订单微服务请求库存微服务减库存。 简言之：跨JVM进程产生分布式事务



2、单体系统访问多个数据库实例 当单体系统需要访问多个数据库（实例）时就会产生分布式事务。 比如：用户信息和订单信息分别在两个MySQL实例存储，用户管理系统删除用户信息，需要分别删除用户信息及用户的订单信息，由于数据分布在不同的数据实例，需要通过不同的数据库链接去操作数据，此时产生分布式事务。 简言之：跨数据库实例产生分布式事务。



3、多服务访问同一个数据库实例 比如：订单微服务和库存微服务即使访问同一个数据库也会产生分布式事务，原因就是跨JVM进程，两个微服务持有了不同的数据库链接进行数据库操作，此时产生分布式事务



### 分布式事务解决方案之2PC(两阶段提交)

前面已经学习了分布式事务的基础理论，以理论为基础，针对不同的分布式场景业界常见的解决方案有2PC、TCC、可靠消息最终一致性、最大努力通知这几种。

#### 什么是2PC

2PC即两阶段提交协议，是将整个事务流程分为两个阶段，准备阶段（Prepare phase）、提交阶段（commit phase），2是指两个阶段，P是指准备阶段，C是指提交阶段。

举例：张三和李四好久不见，老友约起聚餐，饭店老板要求先买单，才能出票。这时张三和李四分别抱怨近况不如意，囊中羞涩，都不愿意请客，这时只能AA。只有张三和李四都付款，老板才能出票安排就餐。但由于张三和李四都是铁公鸡，形成了尴尬的一幕：

准备阶段：老板要求张三付款，张三付款。老板要求李四付款，李四付款。

提交阶段：老板出票，两人拿票纷纷落座就餐。

例子中形成了一个事务，若张三或李四其中一人拒绝付款，或钱不够，店老板都不会给出票，并且会把已收款退回。

整个事务过程由事务管理器和参与者组成，店老板就是事务管理器，张三、李四就是事务参与者，事务管理器负责决策整个分布式事务的提交和回滚，事务参与者负责自己本地事务的提交和回滚。

在计算机中部分关系数据库如Oracle、MySQL支持两阶段提交协议，如下图：

1. 准备阶段（Prepare phase）：事务管理器给每个参与者发送Prepare消息，每个数据库参与者在本地执行事务，并写本地的Undo/Redo日志，此时事务没有提交。（Undo日志是记录修改前的数据，用于数据库回滚，Redo日志是记录修改后的数据，用于提交事务后写入数据文件）

2. 提交阶段（commit phase）：如果事务管理器收到了参与者的执行失败或者超时消息时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据事务管理器的指令执行提交或者回滚操作，并释放事务处理过程中使用的锁资源。注意:必须在最后阶段释放锁资源。

下图展示了2PC的两个阶段，分成功和失败两个情况说明：

成功情况：

