[1.1 为什么要学习zk？ 1](#_Toc75650668)

[1.1.1 分布式环境特点 1](#_Toc75650669)

[1.1.2 分布式环境存在的问题 2](#_Toc75650670)

[1.1.3 CAP理论 2](#_Toc75650671)

[1.1.4 BASE理论 3](#_Toc75650672)

[1.1.5 部署节点为奇数? 3](#_Toc75650673)

[1.2 zookeeper 安装 3](#_Toc75650674)

[1.2.1 zookeeper步骤 3](#_Toc75650675)

[1.2.2 zookeeper的api接口 5](#_Toc75650676)

[1.3 Zookeeper原理与架构 6](#_Toc75650677)

[1.3.1 配置文件及基本命令 6](#_Toc75650678)

[1.3.2 zookeeper特性 7](#_Toc75650679)

[1.3.3 Zookeeper的数据结构 7](#_Toc75650680)

[1.3.4 Znode类型 7](#_Toc75650681)

[1.3.5 zookeeper⻆⾊、节点状态 8](#_Toc75650682)

[1.4 ZAB协议 9](#_Toc75650683)

[1.4.1 ZAB协议 9](#_Toc75650684)

[1.4.2 消息广播模式 10](#_Toc75650685)

[1.4.3 崩溃恢复 10](#_Toc75650686)

[1.4.4 数据同步 11](#_Toc75650687)

[1.4.5 ZAB协议原理 11](#_Toc75650688)

[1.4.6 Zookeeper设计目标 11](#_Toc75650689)

[1.5 分布式事务 11](#_Toc75650690)

[1.5.1 什么是事务 11](#_Toc75650691)

[1.5.2 本地事务 12](#_Toc75650692)

[1.5.3 分布式事务 12](#_Toc75650693)

[1.5.4 分布式事务产生的场景 12](#_Toc75650694)

[1.5.5 分布式事务基础理论 13](#_Toc75650695)

[1.5.6 分布式事务解决方案之2PC(两阶段提交) 15](#_Toc75650696)

## 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

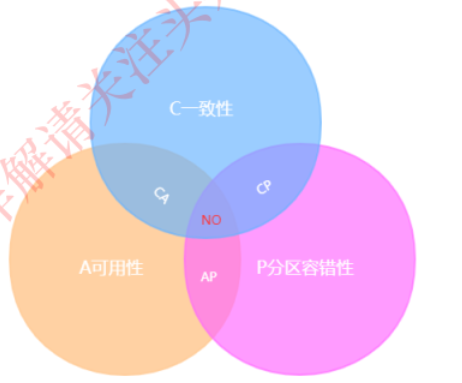
一致性是指数据在多个副本之间是否能够保持一致性的特性。假如现在的多个节点中的数据似乎保持一致的，当执行完某一个更新操作之后，应当要保证系统的数据然后处于一致性的状态，对于一个将数据副本分布在同的分布式节点上，如果对第一个节点的数据进行了更新的操作，并且更新成功之后，却没有让第二个节点得到相应的更新。当外部系统在去调用第二个节点时，获取到的依然是原始的数据，这就是分布式数据不一致的情况。在分布式系统中，如果能够做到针对一个数据项更新操作执行成功之后，所有的用户都可以读到最新的值，那么这样的系统就被认为是具有强一致性的。

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

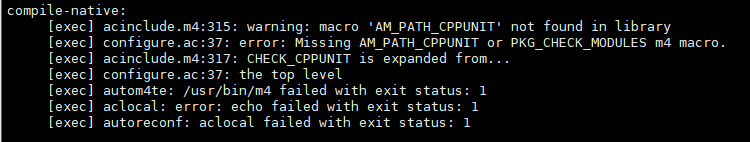
make check

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：

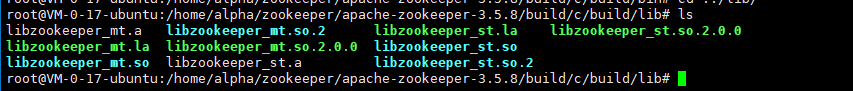
export LD\_LIBRARY\_PATH=/usr/local/lib:$LD\_LIBRARY\_PATH



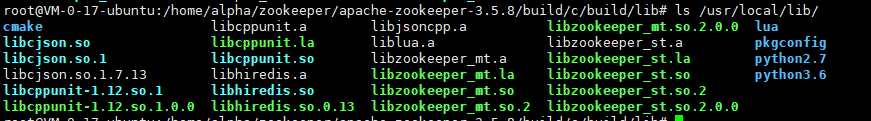
apt-get install libtool

 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

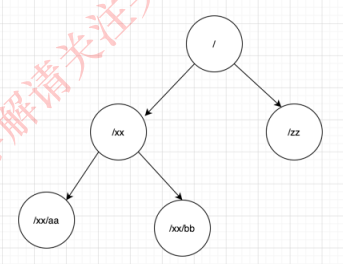
|  |  |
| --- | --- |
| 节点标示 | 描述 |
| cZxid | 创建节点事务标示 |
| ctime | 节点创建⽇期 |
| **mZxid** | 修该节点事务标示 |
| mtime | 修改⽇期 |
| pZxid | ⼦节点更新标示，包含增、删、改操作 |
| cversion | 直接⼦节点版本号(创建、删除) |
| dataVersion | 记录当前节点数据更新版本号 |
| aclVersion | 每次ACL发⽣改变，增加版本号 |
| ephemeralOwner | 增加临时节点，会将客户端sessionId记录在这 |
| dataLength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



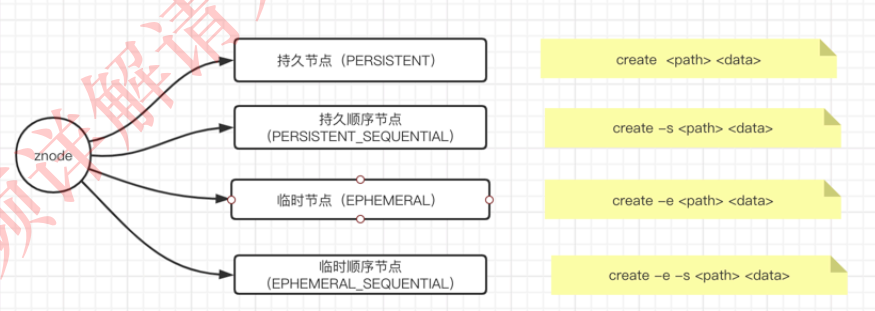
名称是由斜杠（/）分隔的⼀系列路径元素。 ZooKeeper命名空间中的每个节点都由路径进⾏唯⼀标识。

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：

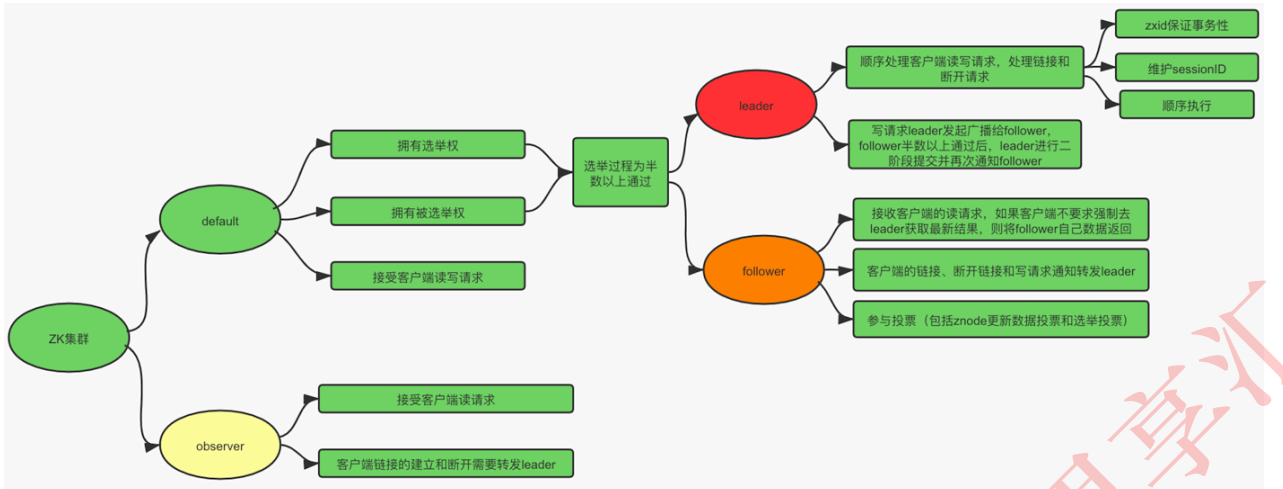
server.1=localhost:2287:3387

server.2=localhost:2288:3388

server.3=localhost:2289:3389

server.4=localhost:2290:3390:observer

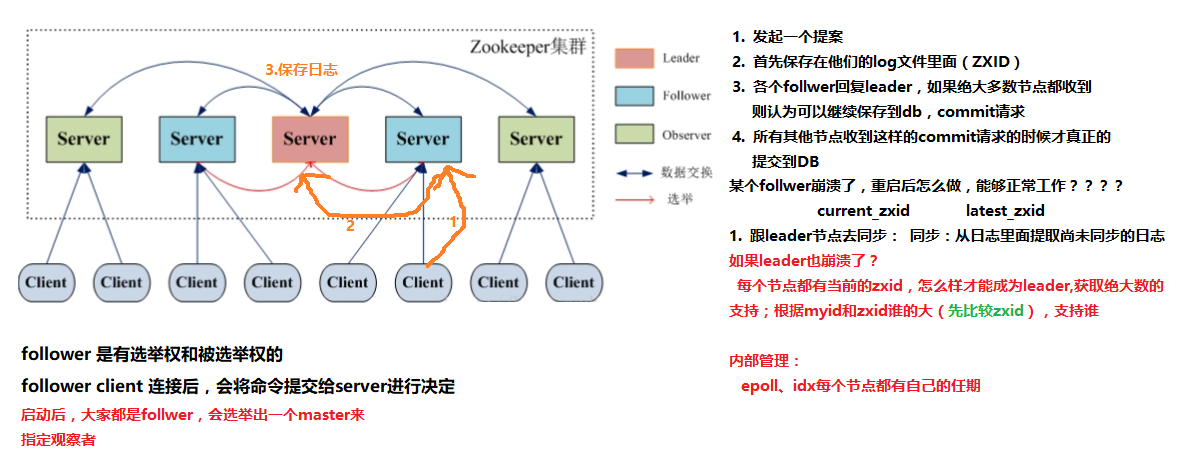
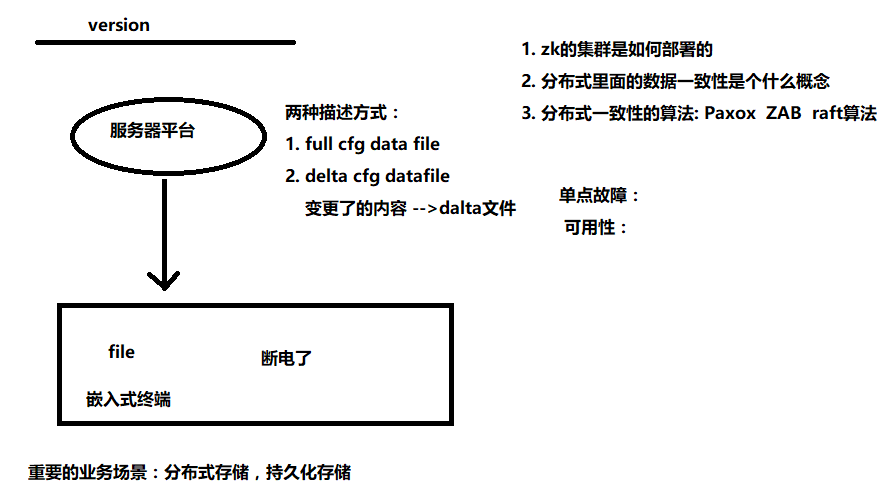
下⾯我们⽤⼀张图来总结下ZK集群节点中 leader , follower , observer ⻆⾊的作⽤



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

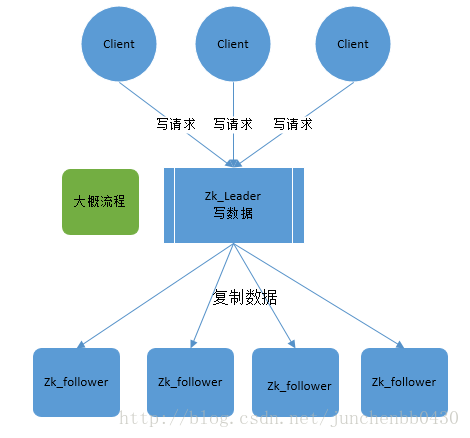
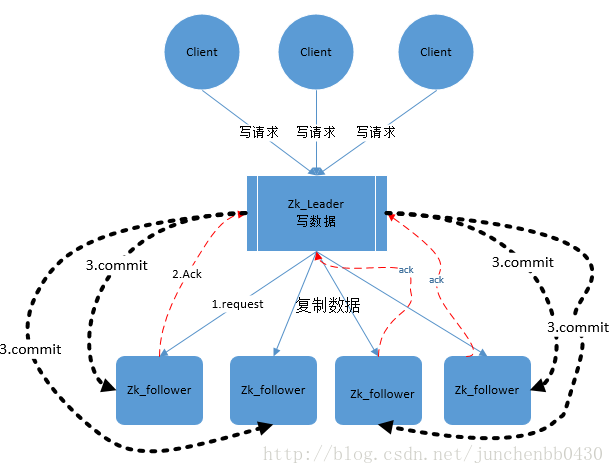
选举原理：

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

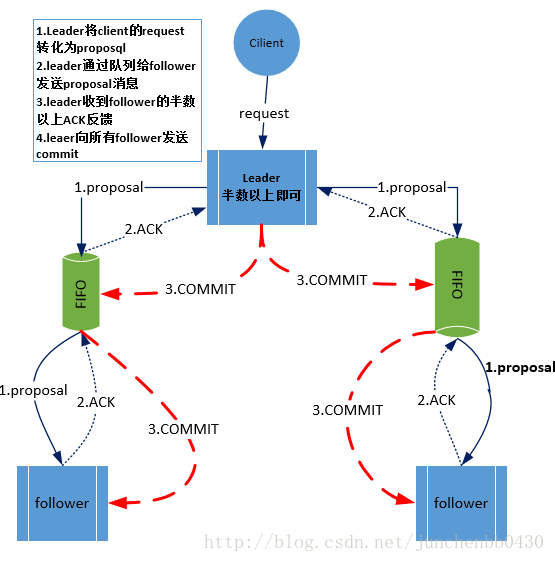


## ZAB协议

### 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。 
3. 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。 
4. 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

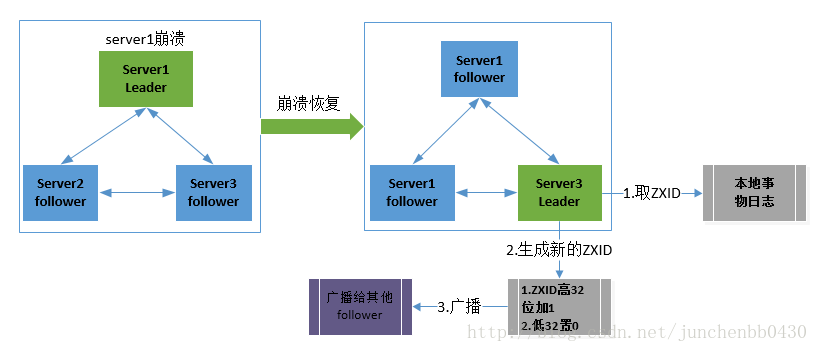
### 消息广播模式

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 
4. **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。
5. **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**。
6. **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**。
4. 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。
5. 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。

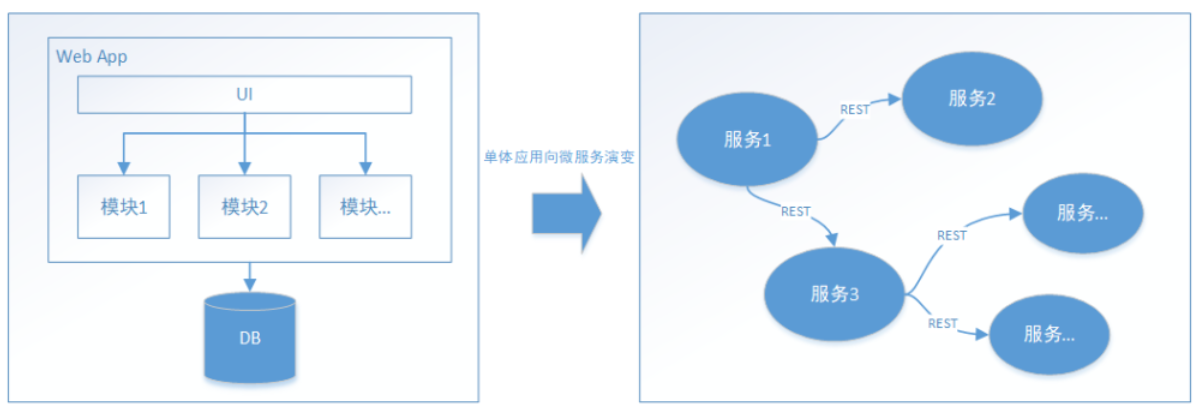
D（Durability）：持久性，事务完成之后，该事务对数据的更改会被持久化到数据库，且不会被回滚。

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：



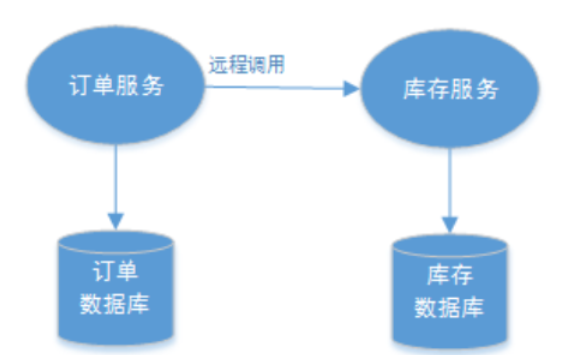
但是在分布式环境下，会变成下边这样：



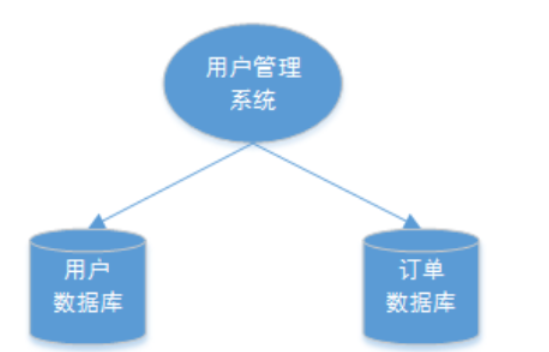
可以设想，当远程调用让李四增加金额成功了，由于网络问题远程调用并没有返回，此时本地事务提交失败就回滚了张三减少金额的操作，此时张三和李四的数据就不一致了。因此在分布式架构的基础上，传统数据库事务就无法使用了，张三和李四的账户不在一个数据库中甚至不在一个应用系统里，实现转账事务需要通过远程调用，由于网络问题就会导致分布式事务问题。

### 分布式事务产生的场景

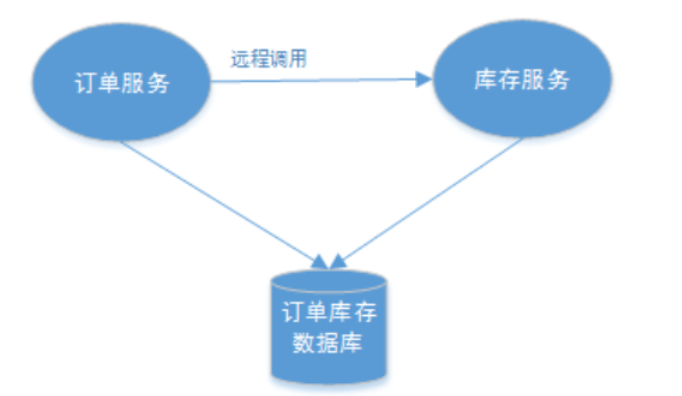
1、典型的场景就是微服务架构 微服务之间通过远程调用完成事务操作。 比如：订单微服务和库存微服务，下单的同时订单微服务请求库存微服务减库存。 简言之：跨JVM进程产生分布式事务



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



### 分布式事务基础理论

通过前面的学习，我们了解到了分布式事务的基础概念。与本地事务不同的是，分布式系统之所以叫分布式，是因为提供服务的各个节点分布在不同机器上，相互之间通过网络交互。不能因为有一点网络问题就导致整个系统无法提供服务，网络因素成为了分布式事务的考量标准之一。因此，分布式事务需要更进一步的理论支持，接下来，我们先来学习一下分布式事务的CAP理论。

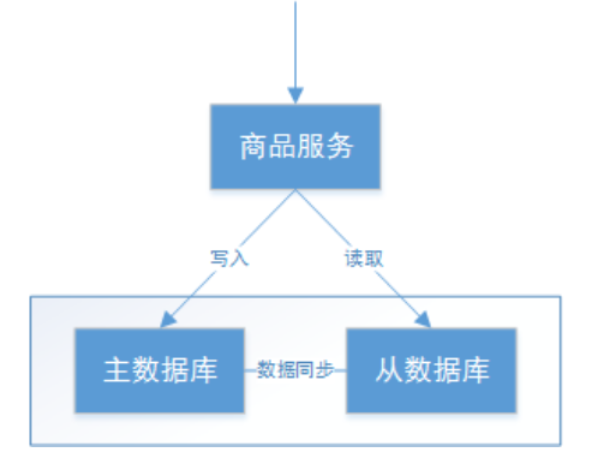
在讲解分布式事务控制解决方案之前需要先学习一些基础理论，通过理论知识指导我们确定分布式事务控制的目标，从而帮助我们理解每个解决方案。

#### 理解CAP

CAP是 Consistency、Availability、Partition tolerance三个词语的缩写，分别表示一致性、可用性、分区容忍性。 下边我们分别来解释：

为了方便对CAP理论的理解，我们结合电商系统中的一些业务场景来理解CAP。

如下图，是商品信息管理的执行流程：



整体执行流程如下：

1、商品服务请求主数据库写入商品信息（添加商品、修改商品、删除商品）

2、主数据库向商品服务响应写入成功。

3、商品服务请求从数据库读取商品信息。

**C - Consistency：**

一致性是指写操作后的读操作可以读取到最新的数据状态，当数据分布在多个节点上，从任意结点读取到的数据都是最新的状态。

上图中，商品信息的读写要满足一致性就是要实现如下目标：

1、商品服务写入主数据库成功，则向从数据库查询新数据也成功。

2、商品服务写入主数据库失败，则向从数据库查询新数据也失败。

如何实现一致性？

1、写入主数据库后要将数据同步到从数据库。

2、写入主数据库后，在向从数据库同步期间要将从数据库锁定，待同步完成后再释放锁，以免在新数据写入成功后，向从数据库查询到旧的数据。

分布式系统一致性的特点：

1、由于存在数据同步的过程，写操作的响应会有一定的延迟。

2、为了保证数据一致性会对资源暂时锁定，待数据同步完成释放锁定资源。

3、如果请求数据同步失败的结点则会返回错误信息，一定不会返回旧数据。

**A - Availability ：**

可用性是指任何事务操作都可以得到响应结果，且不会出现响应超时或响应错误。

上图中，商品信息读取满足可用性就是要实现如下目标：

1、从数据库接收到数据查询的请求则立即能够响应数据查询结果。

2、从数据库不允许出现响应超时或响应错误。

如何实现可用性？

1、写入主数据库后要将数据同步到从数据库。

2、由于要保证从数据库的可用性，不可将从数据库中的资源进行锁定。

3、即时数据还没有同步过来，从数据库也要返回要查询的数据，哪怕是旧数据，如果连旧数据也没有则可以按照约定返回一个默认信息，但不能返回错误或响应超时。

分布式系统可用性的特点：

1、 所有请求都有响应，且不会出现响应超时或响应错误。

**P - Partition tolerance ：**

通常分布式系统的各各结点部署在不同的子网，这就是网络分区，不可避免的会出现由于网络问题而导致结点之间通信失败，此时仍可对外提供服务，这叫分区容忍性。

上图中，商品信息读写满足分区容忍性就是要实现如下目标：

1、主数据库向从数据库同步数据失败不影响读写操作。

2、其一个结点挂掉不影响另一个结点对外提供服务。

如何实现分区容忍性？

1、尽量使用异步取代同步操作，例如使用异步方式将数据从主数据库同步到从数据，这样结点之间能有效的实现松耦合。

2、添加从数据库结点，其中一个从结点挂掉其它从结点提供服务。

分布式分区容忍性的特点：

1、分区容忍性分是布式系统具备的基本能力。

1、上边商品管理的例子是否同时具备 CAP呢？

在所有分布式事务场景中不会同时具备CAP三个特性，因为在具备了P的前提下C和A是不能共存的。

通过上面我们已经学习了CAP理论的相关知识，CAP是一个已经被证实的理论：一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Partition tolerance）这三项中的两项。它可以作为我们进行架构设计、技术选型的考量标准。对于多数大型互联网应用的场景，结点众多、部署分散，而且现在的集群规模越来越大，所以节点故障、网络故障是常态，而且要保证服务可用性达到N个9（99.99..%），并要达到良好的响应性能来提高用户体验，因此一般都会做出如下选择：保证P和A，舍弃C强一致，保证最终一致性。

#### BASE理论

1、理解强一致性和最终一致性

CAP理论告诉我们一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Partition tolerance）这三项中的两项，其中AP在实际应用中较多，AP即舍弃一致性，保证可用性和分区容忍性，但是在实际生产中很多场景都要实现一致性，比如前边我们举的例子主数据库向从数据库同步数据，即使不要一致性，但是最终也要将数据同步成功来保证数据一致，这种一致性和CAP中的一致性不同，CAP中的一致性要求在任何时间查询每个结点数据都必须一致，它强调的是强一致性，但是最终一致性是允许可以在一段时间内每个结点的数据不一致，但是经过一段时间每个结点的数据必须一致，它强调的是最终数据的一致性。

2、Base理论介绍

BASE 是 Basically Available(基本可用)、Soft state(软状态)和 Eventually consistent (最终一致性)三个短语的缩写。BASE理论是对CAP中AP的一个扩展，通过牺牲强一致性来获得可用性，当出现故障允许部分不可用但要保证核心功能可用，允许数据在一段时间内是不一致的，但最终达到一致状态。满足BASE理论的事务，我们称之为“柔性事务”。

基本可用:分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。如，电商网站交易付款出现问题了，商品依然可以正常浏览。

软状态:由于不要求强一致性，所以BASE允许系统中存在中间状态（也叫软状态），这个状态不影响系统可用性，如订单的"支付中"、“数据同步中”等状态，待数据最终一致后状态改为“成功”状态。

最终一致:最终一致是指经过一段时间后，所有节点数据都将会达到一致。如订单的"支付中"状态，最终会变为“支付成功”或者"支付失败"，使订单状态与实际交易结果达成一致，但需要一定时间的延迟、等待。

### 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：

