[1.1 为什么要学习zk？ 1](#_Toc75101615)

[1.1.1 . 安装zookeeper 1](#_Toc75101616)

[1.1.2 2-3.2 Zookeeper 1](#_Toc75101617)

[1.1.3 2-3.3 Zookeeper原理与架构 8](#_Toc75101618)

[1.1.4 ZAB协议 9](#_Toc75101619)

[1.2 消息广播模式 10](#_Toc75101620)

[1.3 崩溃恢复 12](#_Toc75101621)

[1.3.1 数据同步 12](#_Toc75101622)

[1.4 ZAB协议原理 13](#_Toc75101623)

[1.5 Zookeeper设计目标 13](#_Toc75101624)

## 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 安装

1.1 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)

1.2 安装zookeeper

zkServer.sh start stop

zkCli.sh

### 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

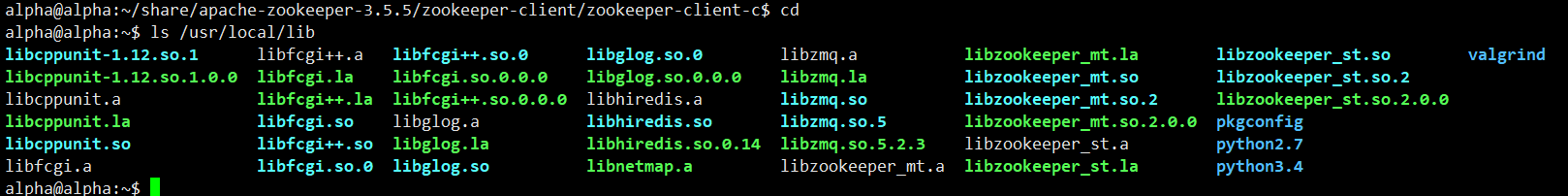
3、编译安装

./configure LDFLAGS='-ldl'

make

make install

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。

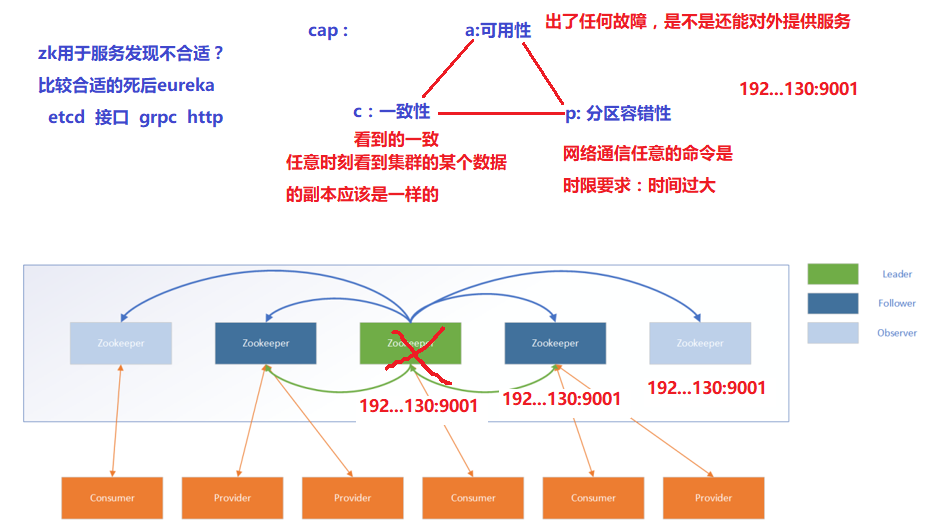
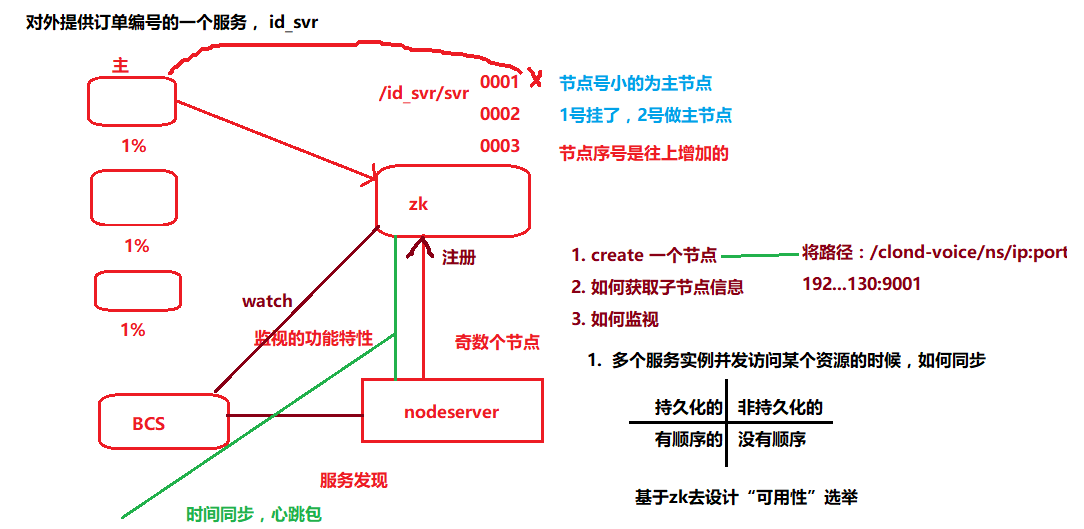
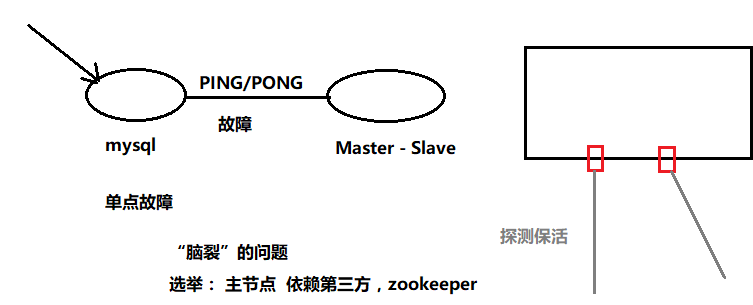


启动server

alpha@alpha:~/share/zookeeper-3.5.5-apache/bin$ sudo ./zkServer.sh start

启动client

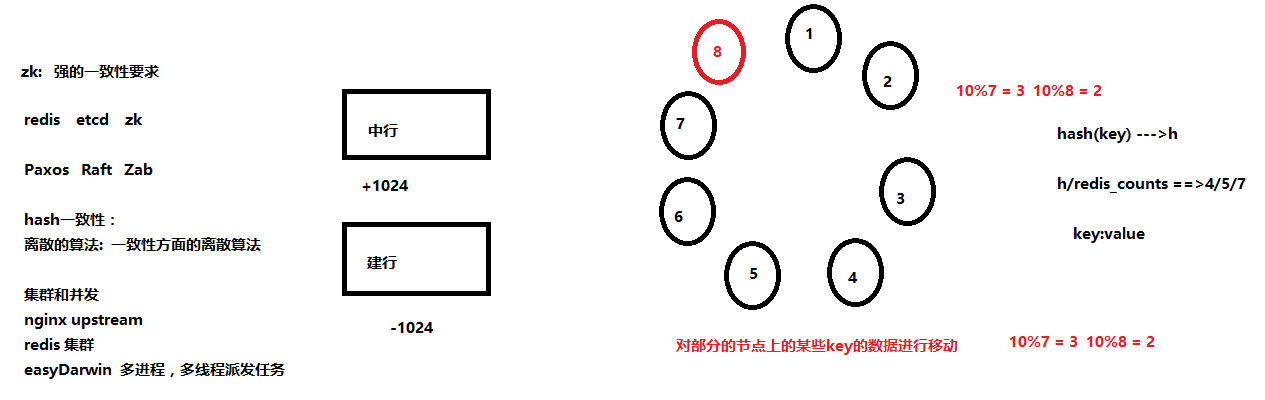
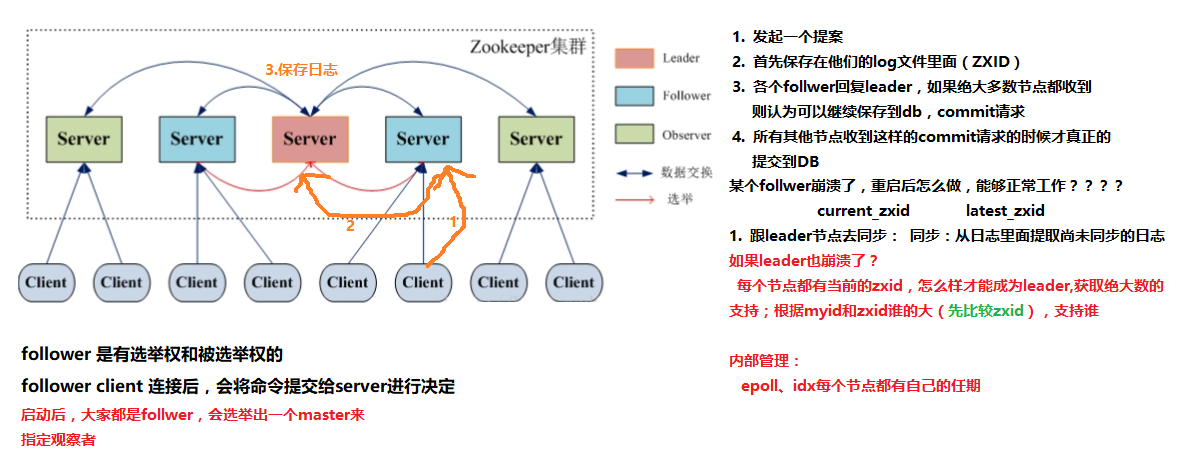
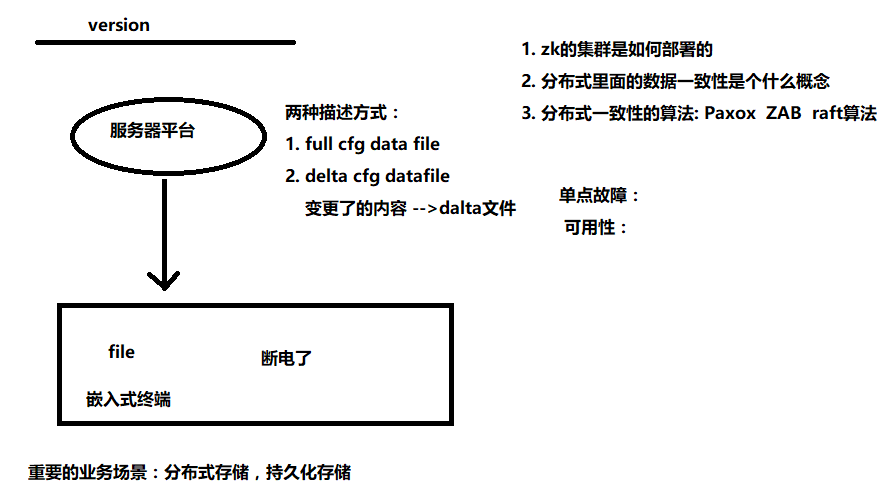
alpha@alpha:~/share/zookeeper-3.5.5-apache/bin$ ./zkCli.sh



### Zookeeper原理与架构

选举原理：

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



### 分布式系统设计遵循三原则

2000年7月，加州大学伯克利分校的Eric Brewer教授在ACM PODC会议上提出CAP猜想。2年后，麻省理工学院的Seth Gilbert和Nancy Lynch从理论上证明了CAP。之后，CAP理论正式成为分布式计算领域的公认定理。

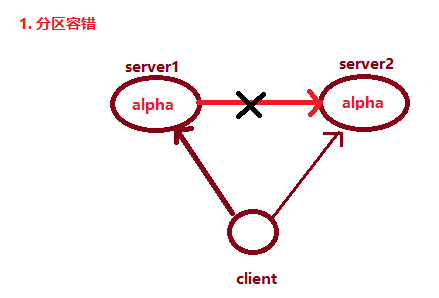
在理论计算机科学中，CAP定理（CAP theorem），又被称作布鲁尔定理（Brewer’s theorem），它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

一致性（Consistency） （等同于所有节点访问同一份最新的数据副本）

可用性（Availability）（每次请求都能获取到非错的响应——但是不保证获取的数据为最新数据）

分区容错性（Partition tolerance）（以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。）

#### 分区容错性

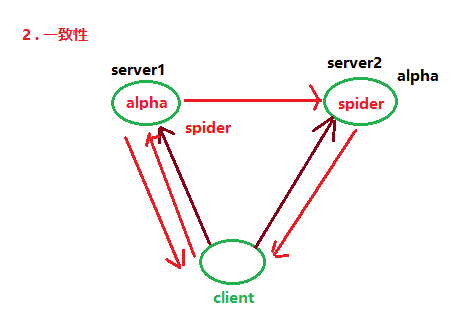


分区容错性指:即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。在分布式应用中，可能因为一些分布式的原因导致系统无法正常运转。好的分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。

比如现在的分布式系统中有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，或者是机器之间有网络异常，将分布式系统分隔未独立的几个部分，各个部分还能维持分布式系统的运作，这样就具有好的分区容错性。

重点是服务器节点之间的通讯中断，而不是与客户之间的中断。server1、server2都能对外提供服务只是两个服务器节点之间无法同步数据。

#### 一致性



对于一致性，可以分为从客户端和服务端两个不同的视角。

从客户端来看: 主要指的是多并发访问时更新过的数据如何获取的问题。

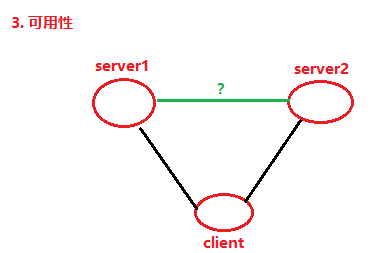
从服务端来看: 更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。

如果能容忍后续的部分或者全部访问不到，则是弱一致性。

如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

#### 可用性



我理解为：集群的节点一部分宕机后，其它机器是否还能正常响应客户端的读写请求。

满足分区容错性的分布式系统，只能在一致性和可用性两者中，选择其中一个。

CA without P

这种情况在分布式系统中几乎是不存在的。首先在分布式环境下，网络分区（通讯故障）是一个自然的事实。因为分区是必然的，所以如果舍弃P，意味着要舍弃分布式系统。

比如我们熟知的关系型数据库，如My Sql和Oracle就是保证了可用性和数据一致性，但是他并不是个分布式系统。一旦关系型数据库要考虑主备同步、集群部署等就必须要把P也考虑进来。

CP without A

如果一个分布式系统不要求强的可用性，即容许系统停机或者长时间无响应的话，就可以在CAP三者中保障CP而舍弃A。一个保证了CP而一个舍弃了A的分布式系统，一旦发生网络故障或者消息丢失等情况，就要牺牲用户的体验，等待服务器数据恢复后才会提供服务。

设计成CP的系统其实也不少，其中最典型的就是很多分布式数据库，他们都是设计成CP的。在发生极端情况时，优先保证数据的强一致性，代价就是舍弃系统的可用性。如Redis、HBase等，还有分布式系统中常用的Zookeeper也是在CAP三者之中选择优先保证CP的。

无论是像Redis、HBase这种分布式存储系统，还是像Zookeeper这种分布式协调组件。数据的一致性是他们最最基本的要求。一个连数据一致性都保证不了的分布式存储要他有何用？

AP wihtout C

要高可用并允许分区，则需放弃一致性。一旦网络问题发生，节点之间可能会失去联系。为了保证高可用，需要在用户访问时可以马上得到返回，则每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。

这种舍弃强一致性而保证系统的分区容错性和可用性的场景和案例非常多。对于很多业务系统来说，比如淘宝的购物，12306的买票。都是在可用性和一致性之间舍弃了一致性而选择可用性。

你在12306买票的时候肯定遇到过这种场景，当你购买的时候提示你是有票的（但是可能实际已经没票了），你也正常的去输入验证码，下单了。但是过了一会系统提示你下单失败，余票不足。这其实就是先在可用性方面保证系统可以正常的服务，然后在数据的一致性方面做了些牺牲，会影响一些用户体验。

但是，我们说很多网站牺牲了一致性，选择了可用性，这其实也不准确的。就比如上面的买票的例子，其实舍弃的只是强一致性，退而求其次保证了最终一致性。也就是说，虽然下单的瞬间，关于车票的库存可能存在数据不一致的情况，但是过了一段时间，还是要保证最终一致性的。

redis符合AP原则

zookeeper 符合CP原则

base：基本可用 弱状态 最终一致性

核心思想：分布式系统中即使无法做到强一致性，采用适当的方式让分布式系统达到最终一致性

基本可用：响应时间 & 功能

弱状态： 允许同步的延时

zk里面是有事务日志的 == aof / zk内部也是一种内存数据库

ZooKeeper是什么 ？ ZooKeeper 的由来

下面这段内容摘自《从Paxos到Zookeeper 》第四章第一节的某段内容，推荐大家阅读以下：

Zookeeper最早起源于雅虎研究院的一个研究小组。在当时，研究人员发现，在雅虎内部很多大型系统基本都需要依赖一个类似的系统来进行分布式协调，但是这些系统往往都存在分布式单点问题。所以，雅虎的开发人员就试图开发一个通用的无单点问题的分布式协调框架，以便让开发人员将精力集中在处理业务逻辑上。

关于“ZooKeeper”这个项目的名字，其实也有一段趣闻。在立项初期，考虑到之前内部很多项目都是使用动物的名字来命名的（例如著名的Pig项目),雅虎的工程师希望给这个项目也取一个动物的名字。时任研究院的首席科学家RaghuRamakrishnan开玩笑地说：“在这样下去，我们这儿就变成动物园了！”此话一出，大家纷纷表示就叫动物园管理员吧！因为各个以动物命名的分布式组件放在一起，雅虎的整个分布式系统看上去就像一个大型的动物园了，而Zookeeper正好要用来进行分布式环境的协调一一于是，Zookeeper的名字也就由此诞生了。

ZooKeeper是什么

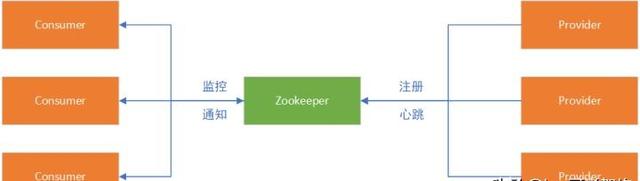
ZooKeeper 是一个开源的分布式协调服务，ZooKeeper框架最初是在“Yahoo!"上构建的，用于以简单而稳健的方式访问他们的应用程序。 后来，Apache ZooKeeper成为Hadoop，HBase和其他分布式框架使用的有组织服务的标准。 例如，Apache HBase使用ZooKeeper跟踪分布式数据的状态。ZooKeeper 的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用。

原语： 操作系统或计算机网络用语范畴。是由若干条指令组成的，用于完成一定功能的一个过程。具有不可分割性即原语的执行必须是连续的，在执行过程中不允许被中断。

ZooKeeper 是一个典型的分布式数据一致性解决方案，分布式应用程序可以基于 ZooKeeper 实现诸如**数据发布/订阅**、**负载均衡**、**服务发现**、**分布式协调/通知**、**集群管理**、**Master 选举**、**分布式锁**和**分布式队**列等功能。

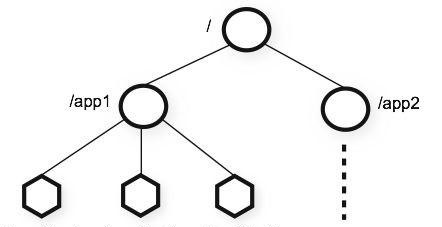
Zookeeper 一个最常用的使用场景就是用于担任服务生产者和服务消费者的注册中心。服务生产者将自己提供的服务注册到Zookeeper中心，服务的消费者在进行服务调用的时候先到Zookeeper中查找服务，获取到服务生产者的详细信息之后，再去调用服务生产者的内容与数据。

如下图所示，使用 Zookeeper 担任了注册中心这一角色。



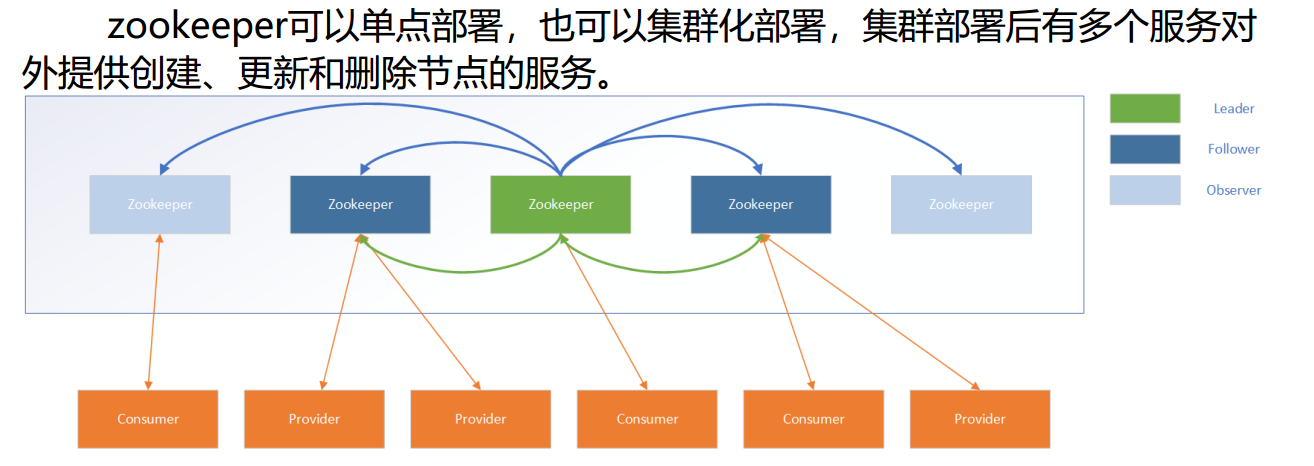
zookeeper内部的数据模型

在zookeeper内部维护了一个树的结构，如下图所示，跟unix的文件目录类似，但是它的每一个节点下还有很多的属性，这一点在接下来的内容中会分享的。



ZooKeeper是集群化部署的

zookeeper可以单点部署，也可以集群化部署，集群部署后有多个服务对外提供创建、更新和删除节点的服务。



为什么要学习zookeeper？

ZooKeeper的特点

顺序一致性： 从同一客户端发起的事务请求，最终将会严格地按照顺序被应用到 ZooKeeper 中去。

原子性： 所有事务请求的处理结果在整个集群中所有机器上的应用情况是一致的，也就是说，要么整个集群中所有的机器都成功应用了某一个事务，要么都没有应用。

单一系统映像 ： 无论客户端连到哪一个 ZooKeeper 服务器上，其看到的服务端数据模型都是一致的。

可靠性： 一旦一次更改请求被应用，更改的结果就会被持久化，直到被下一次更改覆盖，即使在这个过程中，即使有一些zookeeper服务挂掉了。

我们可以利用这些特点，在系统中提供**数据发布/订阅**、**负载均衡**、**服务发现**、**分布式协调/通知**、**集群管理**、**Master 选举**、**分布式锁**和**分布式队**列等功能，那么这些功能实现原理是什么，并且如何编码的？

在使用zookeeper过程中，可能会遇到很多的坑，这些坑又如何避免呢？

进程线程竞争资源

当有一个线程或进程对资源进行操作时，其它线程或进程都不可以对这个资源进行操作，直到该线程或进程完成操作，其它线程或进程才能对该资源进行操作，而其它线程或进程又处于等待状态。

线程进程间同步的方式和机制

临界区 互斥量 信号量 事件

分布式锁的实现：

基于数据库实现分布式锁

基于缓存实现分布式锁

基于zookeeper实现分布式锁

Zookeeper介绍

公式解释： zookeeper = 文件系统 + 监听通知机制

Zookeeper 节点类型

persistent – 持久化目录节点

客户端与zookeeper断开连接后，该节点依旧存在

persistent\_sequential – 持久化顺序编号目录节点

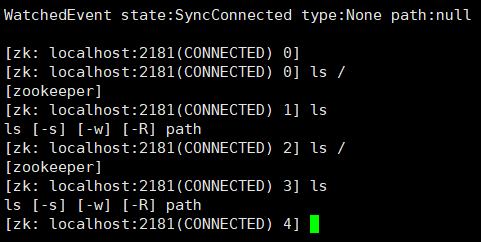
客户端与zookeeper断开连接后，该节点依旧存在，只是zookeeper给该节点名称进行顺序编号

EPHEMERAL –临时目录节点

客户端与zookeeper断开连接后，该节点被删除

EPHEMERAL\_sequential ---- 临时顺序编号目录节点

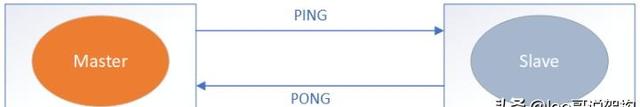
客户端与zookeeper断开连接后，该节点被删除，只是zookeeper给该节点进行顺序编号



分布式系统中普遍存在的问题

可用性

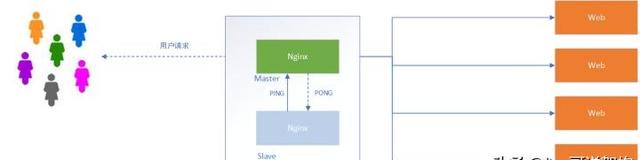
现在很多时候我们的服务需要7\*24小时工作，假如一台机器挂了，我们希望能有其它机器顶替它继续工作。此类问题现在多采用master-salve模式，也就是常说的主从模式，正常情况下主机提供服务，备机负责监听主机状态，当主机异常时，可以自动切换到备机继续提供服务(这里有点儿类似于数据库主库跟备库，备机正常情况下只监听，不工作)，这个切换过程中选出下一个主机的过程就是master选举。



举个栗子，比如我们利用nginx作为代理服务器，把用户的访问代理到Web服务器上，如下图：

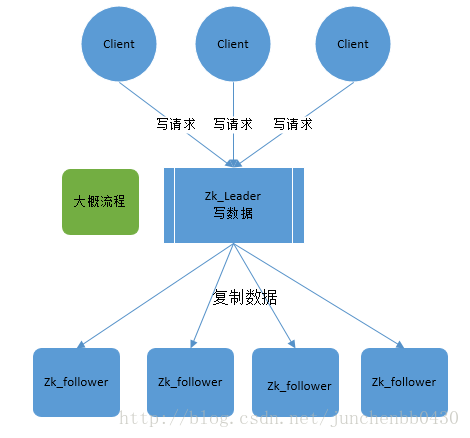
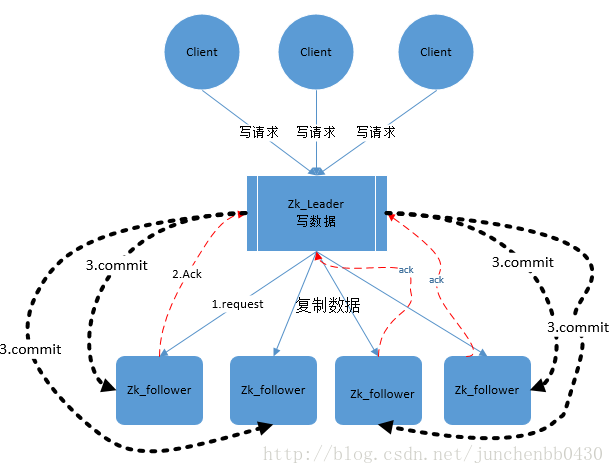


试想，nginx绑定了一个VIP（虚拟IP地址），然后把用户请求代理到Web服务器上，那么如果nginx出现了故障，不能转发客户请求给Web服务器了，于是整个系统就无法对外处理用户的请求了，系统处于一种不可用的状态，这就是称之为**单点故障**。面临这种单点故障，我们应该怎么解决呢？也就是我们必须对外提供高可用（High Availability）的服务。一种解决办法就是nginx部署多台（例如2台）服务器，当一台故障后另一台对外提供服务，这就是经典的Master-Slave的机制。如下图，他们之间使用一个PING/PONG机制，去探测对方是否存在，如果收不到对方的PING或者PONG信息，那么就认为对端挂掉了，比如Slave收不到PONG消息，则认为对端挂了，此时Slave的nginx则设置VIP，然后对外提供服务，看起来十分完美，但是如果是因为网络问题而收不到PONG消息呢？这个时候Slave也会认为Master断开了，Slave nginx也会自我选举成为一个Master nginx，于是系统中就出现了两个Master同时对外提供服务，这就会乱套了，像这么一个情况，我们称之为**脑裂**。那如何解决这种脑裂问题呢？其实，上面我们依赖了一种PING/PONG的机制在选举Master，这种选举机制有缺陷，那么比较好的办法就是引入一个第三方，比如zookeeper，去选举Master。那如何利用zookeeper去做呢？预知后事如何，且听下文分解！

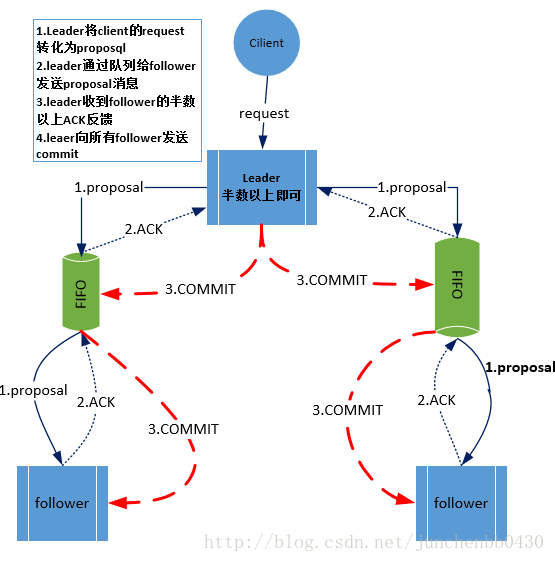


## ZAB协议

### 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

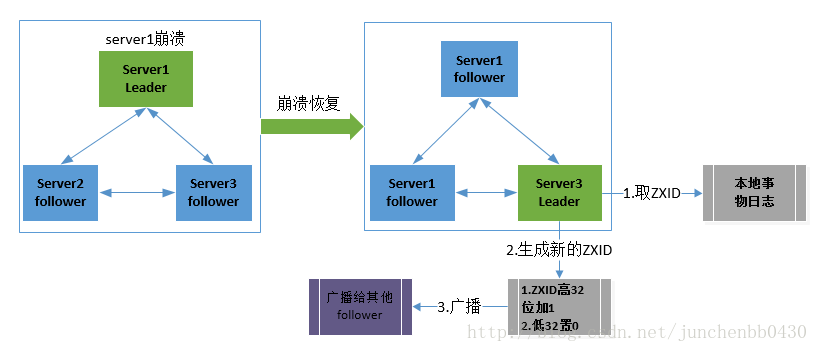
### 消息广播模式

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。