zookeeper详解

「扫码关注我，面试、各种技术实践持续更新中～」

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

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数量来增强分布式系统整体的处理能力，以应对企业的业务增长带来的计算需求。

分布式

分布式系统中的堕胎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。

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。

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等，如何准确并高效地协调分布式并发操作也成为了分布式系统架构与设计中最大的挑战之一。

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消息来进行相互通信。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

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之间进行网络通信，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。

网络分区(脑裂)

当网络由于发生异常情况，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，而另一些节点则不能——我们将这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。

三态

我们已经了解到了在分布式环境下，网络可能会出现各式各样的我呢提，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、失败与超时**。超时出现的原因：1）由于网络原因，该请求（消息）并没有被成功的发送到接收方，而是在发送过程就发生了消息丢失现象。2）该请求（消息）成功的被接受方接收后，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。

节点故障

节点故障则是分布式环境下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。

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服务器的JVM上，所以无法通过本地事务的方式来保证ACID。

分布式理论

基于以上分布式符合当下互联网发展的趋势，但是又确实存在相应的问题，那么我们在设计分布式系统时需要基于哪些理论和算法呢?

其中重要的理论有：CAP，BASE理论。分布式一致性算法：Paxos,ZAB等，其中Paxos是理论算法，ZAB算法是对paxos的工业级实现。

CAP

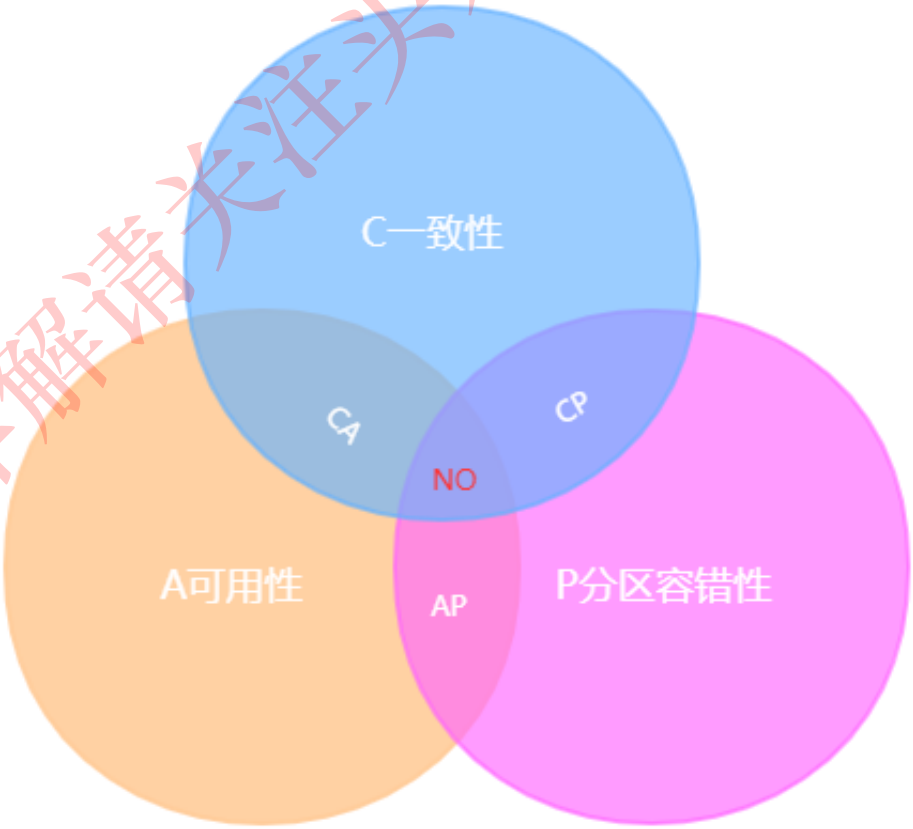
CAP是Consistency、Availablity和Partition-tolerance的缩写。分别是指：

- 一致性（Consistency）：每次读操作都能保证返回的是最新数据；
- 可用性（Availablity）：任何一个没有发生故障的节点，会在合理的时间内返回一个正常的结果；
- 分区容忍性（Partition-tolerance）：当节点间出现网络分区，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的特性**。假如现在的多个节点中的数据是保持一致的，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不同的分布式节点上，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。当外部系统再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了。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取到最新的值，那么这样的系统就被认为是具有强一致性的。

2.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。

3.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，因此在进行对cap定理的应用时，我们就需要去抛弃一项。

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。响应时间合理延长，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。

zookeeper集群搭建

搭建步骤

「zookeeper详解三」ZK集群搭建leader选举（上）：https://www.ixigua.com/i6832592490751590920/

「zookeeper详解四」ZK集群搭建（中）：https://www.ixigua.com/i6832620542932025859/

「zookeeper详解五」ZK集群搭建（下）：https://www.ixigua.com/i6832626201966674439/

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。

为什么ZK集群建议可参与选举节点数量为单数?

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原理概述

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

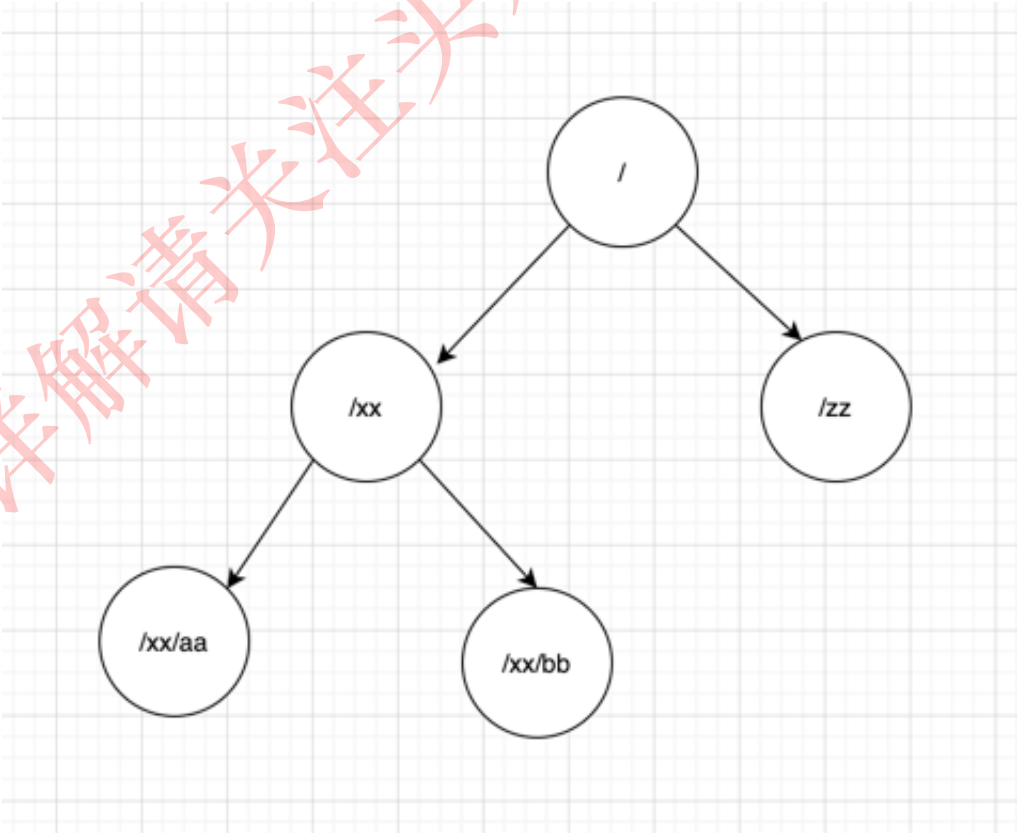
节点stat

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	节点存储数据的长度（B）
numChildren	直接子节点的个数

zookeeper特性

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。

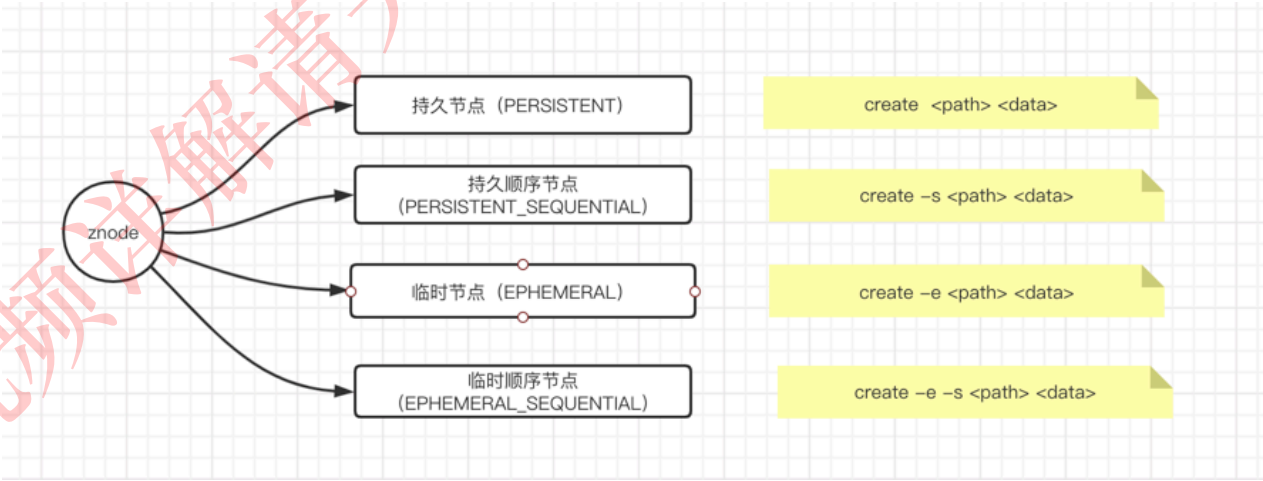
ZooKeeper的层次命名空间

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	客户端和服务器端建立的一个session，存放在临时节点上，如果客户端断开链接，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，最大时2的32次方减1

通过以上3种类型可以组合成4种节点



Znode的类型分为三类：

持久节点（persistent node）节点会被持久
临时节点（ephemeral node），客户端断开连接后，ZooKeeper会自动删除临时节点
顺序节点（sequential node），每次创建顺序节点时，ZooKeeper都会在路径后面自动添加上10位的数字，从1开始，最大是2147483647 （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，由Zookeeper的leader实例维护。

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角色、节点状态

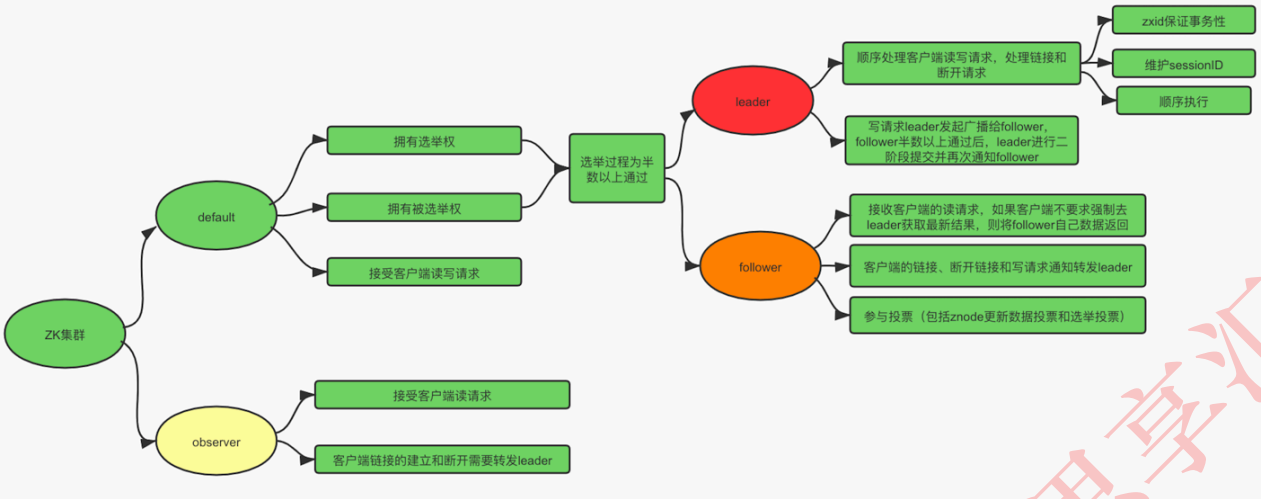
ZK集群服务节点角色

ZK集群中的 server 分为三种角色：`leader`，`follower`，`observer`。

那么角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` 角色的作用



- leader 是集群中最重要的角色。主要的工作任务有三项：
  1. 事务请求的唯一处理者，保证集群事务处理的顺序性，它会将每个状态更新请求进行顺序管理，以便保证整个集群内部消息处理的 FIFO，遵循了顺序一致性（Sequential Consistency）。
  2. 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），针对客户端连接，断开连接，节点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（transaction）操作。如果客户端有写操作，都会被 follower 统一转发给leader处理，然后leader会下发提案让follower投票，如果投票超过半数，则提交事务并通知follower和observer。
  3. leader内部维护 session ，来自客户端的连接和断开连接，都会被统一follower 或 observer 转发给leader处理。
- follower 具有选举权。

  1.处理客户端非事务请求、转发事务请求给 leader 服务器；

  2.参与事务请求 Proposal 的投票（需要半数以上服务器通过才能通知 leader commit 数据; Leader 发起的提案， 要求 Follower 投票）；

  3.参与 Leader 选举的投票 。
- observer没有选举权。Observer 是 zookeeper3.3 开始引入的一个全新的服务器 角色，从字面来理解，该角色充当了观察者的角色。 观察 zookeeper 集群中的最新状态变化并将这些状态变化同步到 observer 服务器上。observer 的工作原理与 follower 角色基本一致，而它和 follower 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 服务器只提供非事物请求服务，通常在于不影响集群事物处理能力的前提下提升集群非事物处理的能力 。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。

Zookeeper 的存储

zookeeper中的znode数据都是在内存中优先维护和提供读服务，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]

```
wenqideMacBook-Pro:bin wenqi$ netstat -anp tcp | grep [23][23]8[789]
tcp4       0      0  127.0.0.1.2288         127.0.0.1.50587        ESTABLISHED
tcp4       0      0  127.0.0.1.50587        127.0.0.1.2288         ESTABLISHED
tcp4       0      0  127.0.0.1.2288         127.0.0.1.50487        ESTABLISHED
tcp4       0      0  127.0.0.1.50487        127.0.0.1.2288         ESTABLISHED
tcp4       0      0  127.0.0.1.2288         *.*                    LISTEN
tcp4       0      0  127.0.0.1.3388         127.0.0.1.50471        ESTABLISHED
tcp4       0      0  127.0.0.1.50471        127.0.0.1.3388         ESTABLISHED
tcp4       0      0  127.0.0.1.3387         127.0.0.1.50470        ESTABLISHED
tcp4       0      0  127.0.0.1.50470        127.0.0.1.3387         ESTABLISHED
tcp4       0      0  127.0.0.1.3389         *.*                    LISTEN
tcp4       0      0  127.0.0.1.3387         127.0.0.1.50361        ESTABLISHED
tcp4       0      0  127.0.0.1.50361        127.0.0.1.3387         ESTABLISHED
tcp4       0      0  127.0.0.1.3388         *.*                    LISTEN
tcp4       0      0  127.0.0.1.3387         *.*                    LISTEN
```

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。

TCP四次挥手的过程如下：

> 主动关闭端发送一个FIN包给被动关闭端；
>
> 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。

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。

TCP四次挥手的过程如下：

主动关闭端发送一个FIN包给被动关闭端；

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

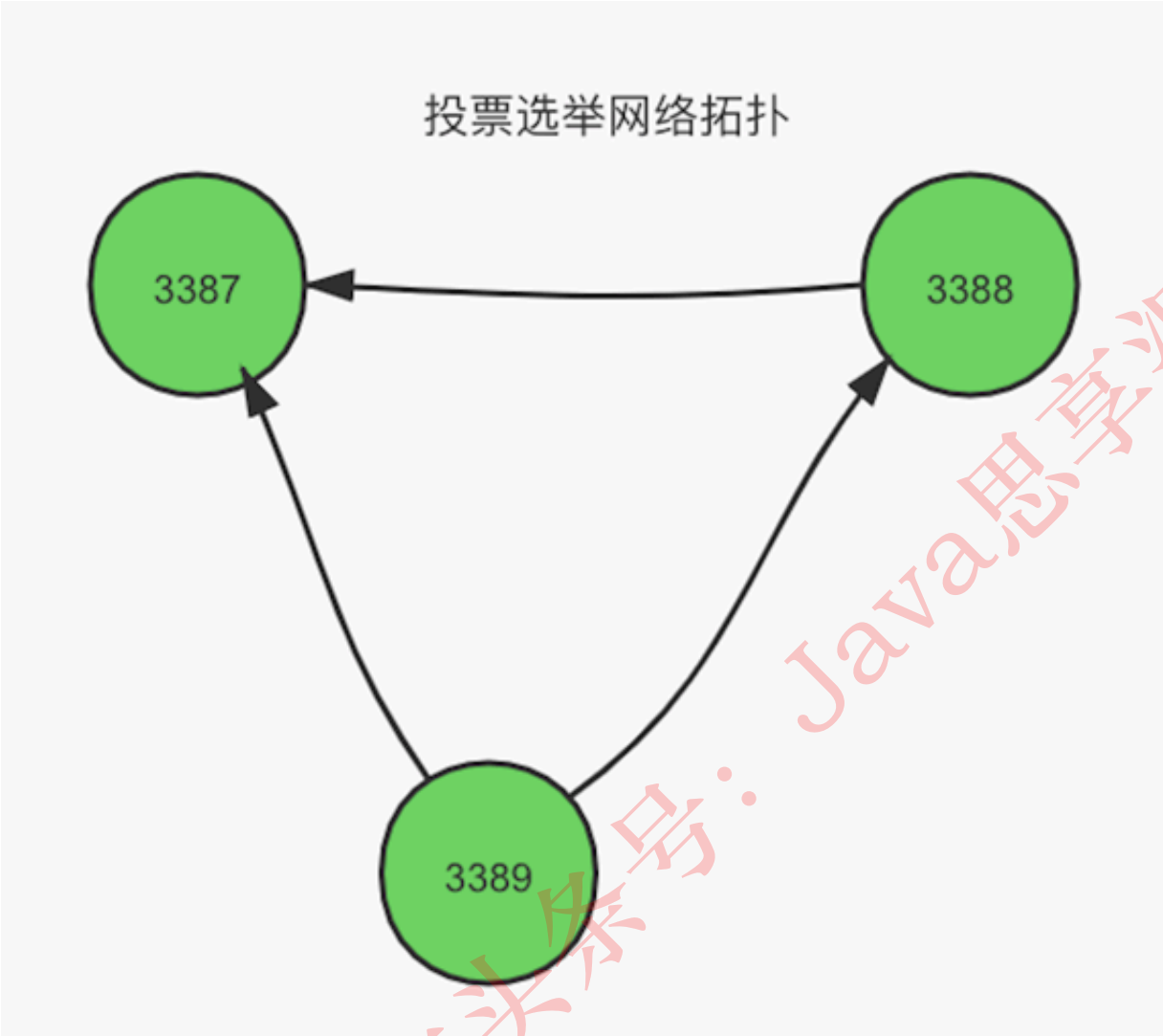
1） LISTEN：侦听状态，等待远程机器的连接请求；

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，

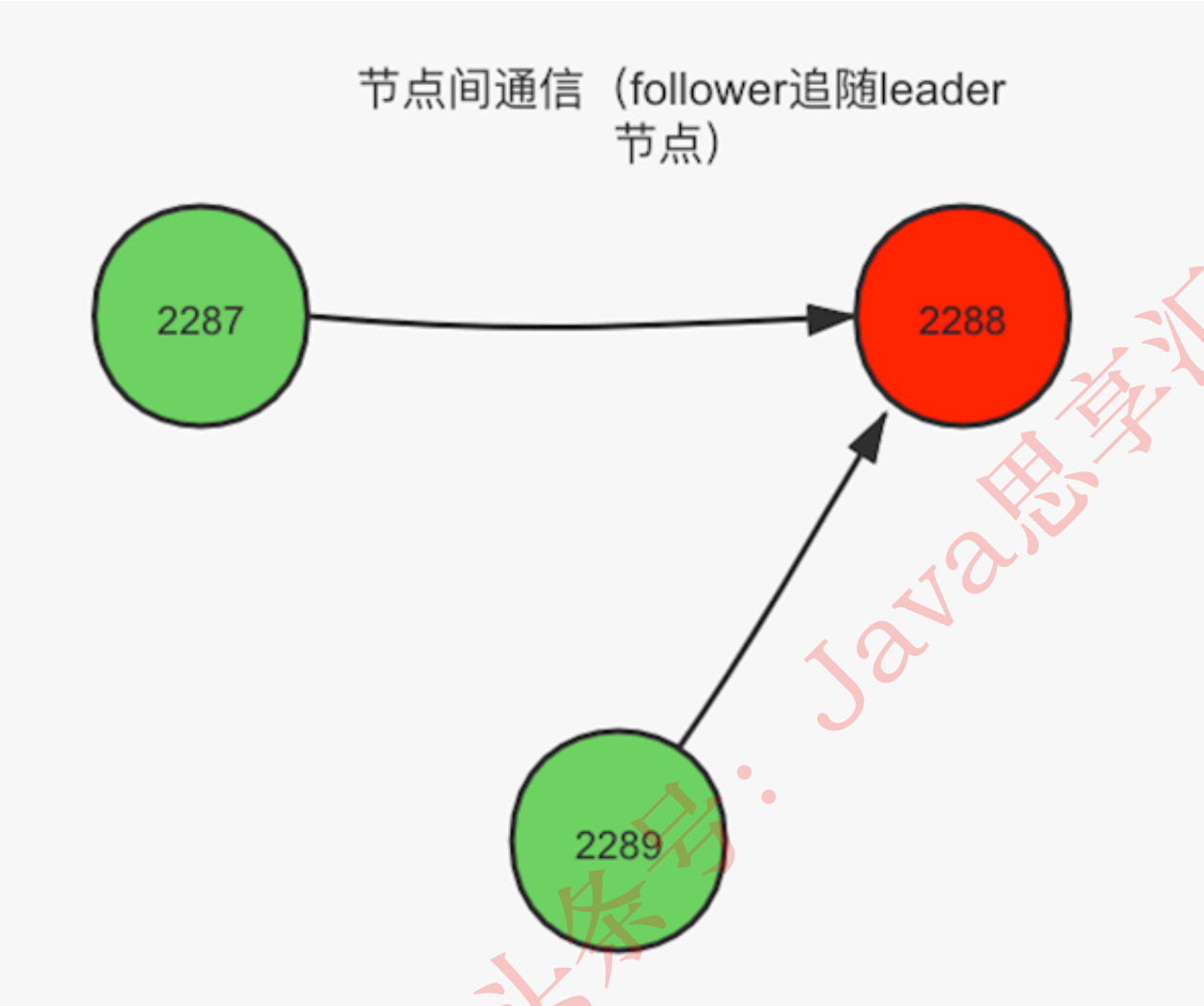
server.1=localhost:2287:**3387**
server.2=localhost:2288:**3388**
server.3=localhost:2289:**3389**



也就是3387、3388、3389，我们会发现以3387作为服务节点时，3388和3389建立连接进来；以3388作为服务节点时，3389建立链接进来，3389作为服务节点时，并没有链接进来，如上图所示，3个节点形成了两两相连的网络拓扑。

然后我们再分析配置文件zoo.cfg中的第1列端口号

server.1=localhost:**2287**:3387
server.2=localhost:**2288**:3388
server.3=localhost:**2289**:3389



也就是2287、2288、2289，其中2288作为leader节点，我们根据TCP通信图发现2287、2289都是追随2288（leader）节点的。

总结：zoo.cfg配置文件中ip后的第一列端口号为zk集群节点时follower节点和leader节点通信的端口号，主要为znode数据的同步、传送。zoo.cfg配置文件中ip后的第二列端口号为zk集群节点间选举leader和znode数据变更投票的端口号。

Paxos算法、ZAB、选主

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，Leslie Lamport在1990年提出，近几年被广泛应用于分布式计算中，Google的Chubby，Apache的Zookeeper都是基于它的理论来实现的，Paxos还被认为是到目前为止**唯一的分布式一致性算法**，其它的算法都是Paxos的改进或简化。有个问题要提一下，Paxos有一个前

提：没有**拜占庭将军**问题。就是说Paxos只有在一个可信的计算环境中才能成立，这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**，这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**的楼长同意了才可以生效。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，错过表决的提议不能再次表决，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，并不断的进行投票，更新新的提议编号。由于每次开会时会议室比较大，通知和修改提议编号有延迟情况，不能保证所有的楼长笔记本上的提议编号都是相同的。但是会议有一个目标：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。他首先看了一下记事本，嗯，当前提议编号是0，那么我的这个提议的编号就是1，于是他给所有楼长发消息：1号提议，将物业费调整为3元/平。其他楼长收到消息以后查了一下记事本，哦，当前提议编号是0，这个提议可接受，于是他记录下这个提议并回复：我接受你的1号提议，同时他在记事本上记录：当前提议编号为1。发起提议的楼长收到了**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诉对方：物业费为3元/平哈。

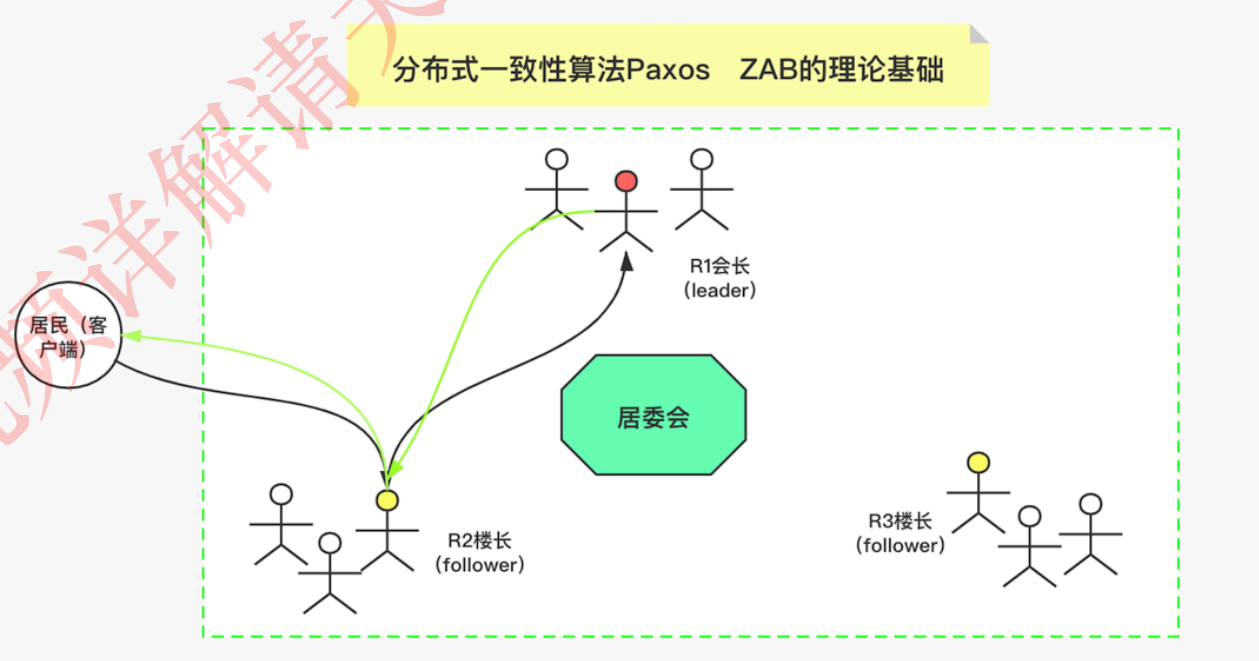
现在看冲突的解决：假设总共有三个楼长，R1，R2同时发起了一个天:1号提议，物业费调整为多少合适呢。R1想设为3元/平, R2想设为4元/平。结果R3先收到了R1的提议，于是他做了和前面同样的操作。紧接着他又收到了R2的提议，结果他一查记事本，咦，这个提议的编号小于等于我的当前编号1，于是他拒绝了这个提议：对不起，这个提议先前提过了。于是R2的提议被拒绝，R1正式发布了提议: 1号提议生效。R2向R1或者R3打听并更新了1号通过提议的内容，然后他可以选择继续发起自己的提议，但是编号增加为2号提议。

还有就是居委会一般都会选出一个**会长**，类似于zk集群的leader，其实Leader的概念也应该属于Paxos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”（指的是每个楼长都坚持自己的提出的提议）。Paxos的作者Lamport在他的文章”The Part-Time Parliament“中阐述了这个问题并给出了解决方案——在所有楼长中选出一个居委会会长，只有会长有权发出提议，如果楼长有自己的提议，必须发给会长并由会长来提出。好，我们又多了一个职务：会长。

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：全体居民
>
> 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：会长（myid选出）
>
> follower：楼长
>
> 除楼长外居民：observer

然后我们模拟客户端访问zk集群的场景



情况一：(查询操作)

居民A(Client)到某个楼长(ZK Server)那里询问(Get)某条正式提议的情况(ZNode的数据)，楼长毫不犹豫的拿出他的记事本(local storage)，查阅生效提议并告诉他结果，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一下再告诉你。

情况二：(修改操作)

居民A(Client)到某个楼长(ZK Server)那里交物业费100元，楼长让他在办公室等着，自己将问题反映给了会长，会长询问所有楼长的意见，多数楼长表示物业费可以收，于是会长发表声明，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，并将收据给了楼长，转交给了交钱的居民A。

情况三：(选举操作)

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

ZAB

ZAB协议是什么

Zab协议 的全称是 **Zookeeper Atomic Broadcast** （Zookeeper原子广播）。**Zookeeper 是通过 Zab 协议来保证分布式事务的最终一致性**。（AP）

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Paxos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（即Leader和Follower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模型，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 Proposal 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Follower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。

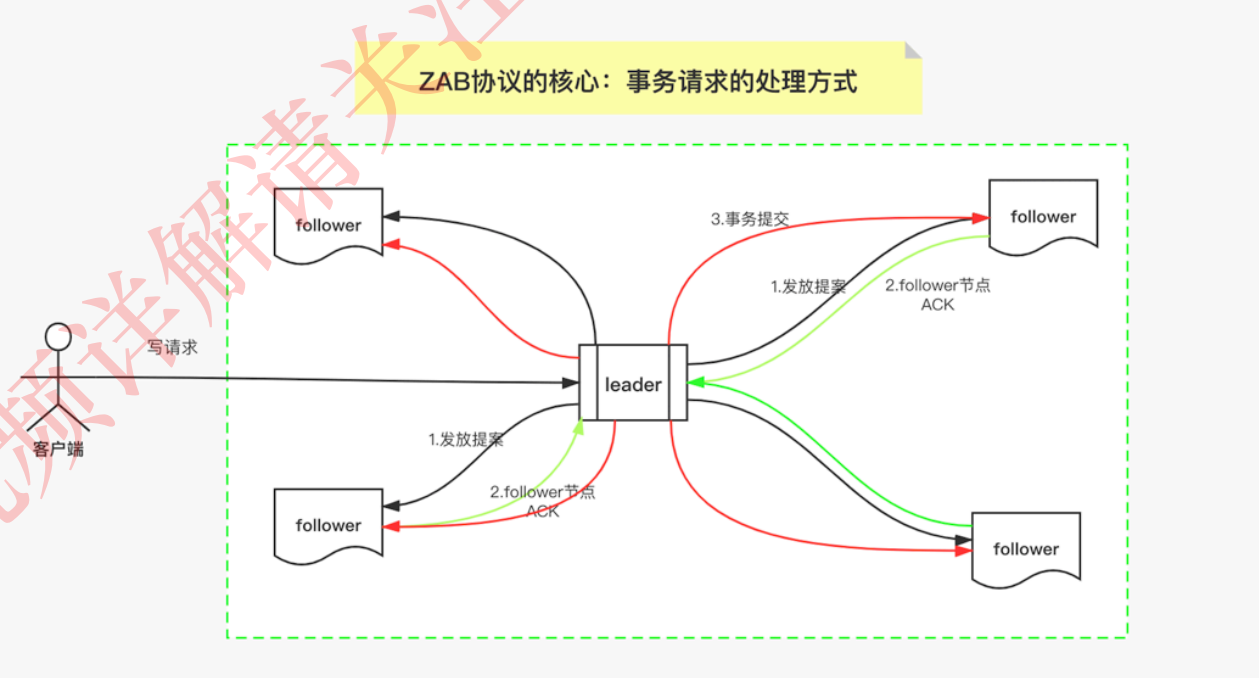
ZAB协议原理

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 Follower 可用客户端列表。将来客户端可以和这些 Follower节点进行通信。
2. **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Proposal请求，将新的Proposal请求广播给所有的 Follower。

ZAB协议核心

ZAB协议的核心：**定义了事务请求的处理方式**



1. 所有的事务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 **Follower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 分发给集群中所有的 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服务器的反馈（Ack请求），**在Zab协议中，只要超过半数的Follower服务器进行了正确的反馈**后（也就是收到半数以上的Follower的Ack请求），那么 Leader 就会再次向所有的 Follower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（Leader）来接收并处理客户端的事务请求**（也就是写请求），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proposal （事务提议）**的形式广播到所有的副本（Follower）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Leader的事务被apply之后，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。

ZAB协议内容

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，或者当 Leader 服务器出现网络中断、崩溃退出或重启等异常时，Zab协议就会 **进入崩溃恢复模式**，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 服务器完成了状态同步（即数据同步）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，如果有一台遵守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。

2.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。

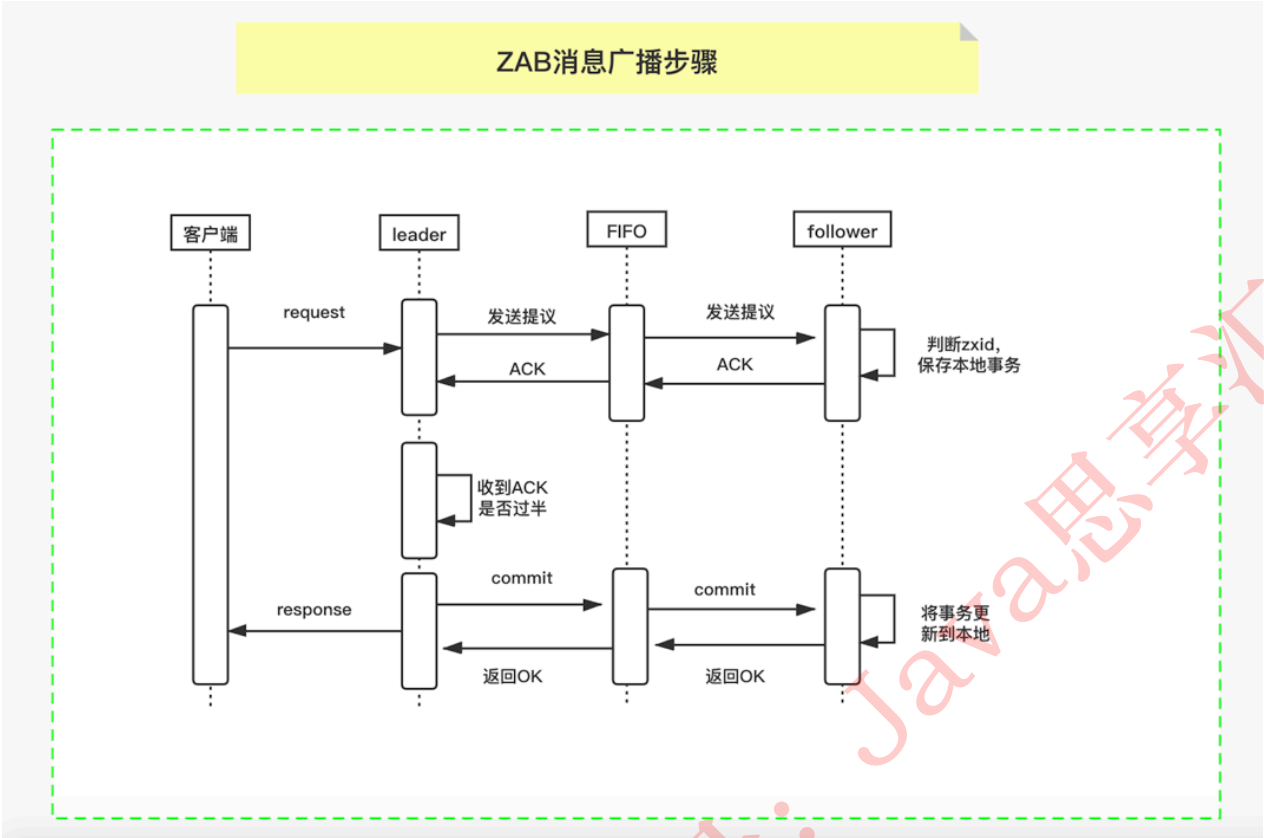
3.保证消息有序

在整个消息广播中，Leader会将每一个事务请求转换成对应的 proposal 来进行广播，并且在广播事务Proposal 之前，Leader服务器会首先为这个事务Proposal分配一个全局单递增的唯一ID，称之为事务ID（即zxid），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的先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与二段提交相似，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，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。二段提交会产生严重的阻塞问题。
2. Zab协议中 Leader 等待 Follower 的ACK反馈消息是指“只要半数以上的Follower成功反馈即可，不需要收到全部Follower反馈”。

消息广播具体步骤



1. 客户端发起一个写操作请求。
2. Leader 服务器将客户端的请求转化为事务 Proposal 提案，同时为每个 Proposal 分配一个全局的ID，即zxid。
3. Leader 服务器为每个 Follower 服务器分配一个单独的队列，然后将需要广播的 Proposal 依次放到队列中，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 Follower 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 Follower 广播 commit 消息，同时自身也会完成事务提交。Follower 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 实现最终一致性的一个体现。

leader 服务器与每一个 Follower 服务器之间都维护了一个单独的 FIFO 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 Follower 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 Follower 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，从而确保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选举产生的新 Leader 服务器。

崩溃恢复主要包括两部分：**Leader选举** 和 **数据恢复**

恢复模式的两个原则保证数据一致性

1. 假设两种异常情况：
   一个事务在 Leader 上提交了，并且过半的 Folower 都响应 Ack 了，但是 Leader 在 Commit 消息发出之前挂了。
2. 假设一个事务在 Leader 提出之后，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 Follower 的 ACKs 后，就向各个 Follower 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 Follower 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**，所以其并没有 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 Server 上都执行 了那些已经被部分 Server 执行过

的事务。

2. 被丢弃的消息不能再现

当在 Leader 新事务已经通过，其已经将该事务更新到了本地，但所有 Follower 还都没 有收到 COMMIT 之前，Leader 宕机了（比前面叙述的宕机更早），此时，所有 Follower 根本 就不知道该 Proposal 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成为了 Follower。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 Proposa 应该被丢弃。类似这样应该被丢弃的事务，是不能再次出现在集群中的， 应该被清除。

Zab 如何数据同步

1. 完成 Leader 选举后（新的 Leader 具有最高的zxid），在正式开始工作之前（接收事务请求，然后提出新的 Proposal），Leader 服务器会首先确认事务日志中的所有的 Proposal 是否已经被集群中过半的服务器 Commit。
2. Leader 服务器需要确保所有的 Follower 服务器能够接收到每一条事务的 Proposal ，并且能将所有已经提交的事务 Proposal 应用到内存数据中。等到 Follower 将所有尚未同步的事务 Proposal 都从 Leader 服务器上同步过啦并且应用到内存数据中以后，Leader 才会把该 Follower 加入到真正可用的 Follower 列表中。

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 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，其他 Follower 也不会听它的了，因为 Follower 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充最大编号 Proposal 的 zxid，并从 zxid 中解析得到对应的 epoch 编号，然后再对其加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 的异常情况。

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 Follower 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 进行比对，比对的结果肯定是 Leader 要求 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。

ZAB实现原理

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读请求；zk 集群中事务请求的唯一处理者，并负责发起决 议和投票，然后将通过的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（具有选举权与被选举权）；
- Observer：就是没有选举权与被选举权，且没有投票权的 Follower（临时工）。若 zk 集 群中的读压力很大，则需要增加 Observer，最好不要增加 Follower。因为增加 Follower 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。

这三类角色在不同的情况下又有一些不同的名称（这个为了下一篇阅读源码做准备，可以先了解即可）：

- Learner = Follower + Observer
- QuorumServer = Follower + Leader

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 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（可以理解为版本号）
- 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 编号，然后再对其加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zxid。

四种状态

zk 集群中的每一台主机，在不同的阶段会处于不同的状态。每一台主机具有四种状态。

- LOOKING：选举状态
- FOLLOWING：Follower 的正常工作状态，从 Leader 同步数据的状态
- LEADING：Leader 的正常工作状态，Leader 广播数据更新的状态
- OBSERVING：Observer 的正常工作状态，从 Leader 同步数据的状态

> 代码实现中，多了一种状态：Observing 状态这是 Zookeeper 引入 Observer 之后加入的，Observer 不参与选举，是只读节点，实际上跟 Zab 协议没有关系。这里为了阅读源码加上此概念。

Zab 的四个阶段

- **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角色，在正常情况下最多只有一个参与者扮演Leader角色，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。

选主过程以及算法

https://blog.csdn.net/alyson_han/article/details/80044047

https://my.oschina.net/coderluo/blog/3106066

zookeeper-API

原生JavaAPI

添加依赖

```
<!-- 加入zk原生依赖包 -->
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.8</version>
</dependency>
```

测试链接zk集群

```
/**
 * @Description: 测试连接ZK集群
 * @Auther: 头条号：Java思享汇
 */
public class TestConnect {
    public static void main(String[] args) {
        final CountDownLatch countDownLatch=new
CountDownLatch(1);
        //集群的连接信息
        String
conn="127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183";

        long start = System.currentTimeMillis();

        try {
            ZooKeeper zooKeeper=new ZooKeeper(conn, 3000, new
Watcher() {
                @Override
                public void process(WatchedEvent watchedEvent)
{

  if(Event.KeeperState.SyncConnected==watchedEvent.getState()){
                        // 连接成功
                        countDownLatch.countDown();
                        System.out.println("建立连接成功");
                    }
                }
            });
```

```
            countDownLatch.await();
            System.out.println("链接状
态："+zooKeeper.getState());

            long total = System.currentTimeMillis() - start;
            System.out.println("总共花费时间："+total);

            zooKeeper.close();

        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

节点创建、修改、删除

```
/**
 * @Description: 模拟创建、修改、删除节点
 * @Auther: 头条号：Java思享汇
 */
public class TestCRUD {
    public static void main(String[] args) {
        final CountDownLatch countDownLatch=new
CountDownLatch(1);
        String
conn="127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183";

        long start = System.currentTimeMillis();

        try {
            ZooKeeper zooKeeper=new ZooKeeper(conn, 3000, new
Watcher() {
                @Override
                public void process(WatchedEvent watchedEvent)
{
```

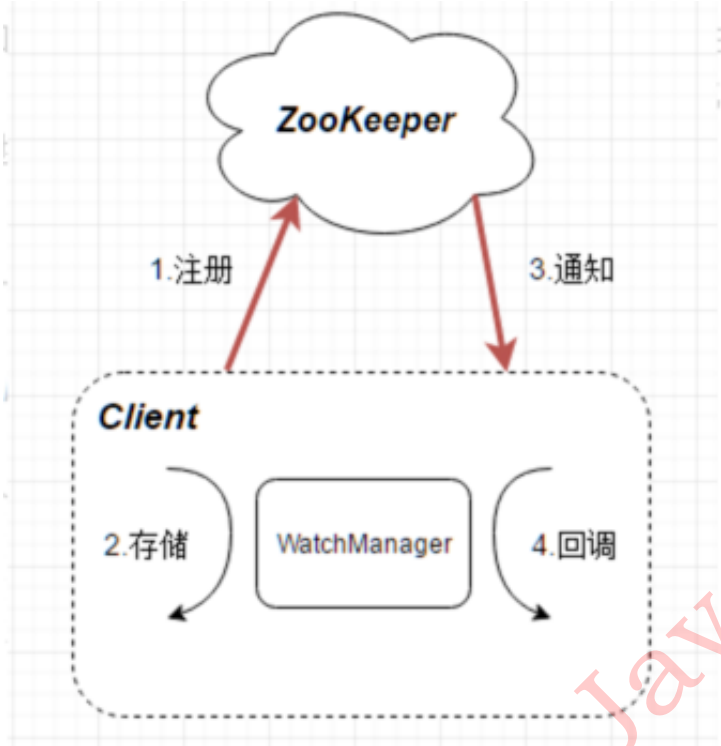
```
 if(Event.KeeperState.SyncConnected==watchedEvent.getState()){
                        // 连接成功
                        countDownLatch.countDown();
                        System.out.println("建立连接成功");
                    }
                }
            });

            countDownLatch.await();
            System.out.println("链接状
态："+zooKeeper.getState());

            long total = System.currentTimeMillis() - start;

            // 1.新增节点
            zooKeeper.create("/node","0".getBytes(),
ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);

            Thread.sleep(1000);
            Stat stat=new Stat();// 状态

            // 1.1.查看新增后的节点状态
            byte[] bytes=zooKeeper.getData("/node",null,stat);
            System.out.println("获取刚刚创建节点数据为："+new
String(bytes));

            Thread.sleep(8000);

            // 2.修改节点

  zooKeeper.setData("/node","1".getBytes(),stat.getVersion());

            // 2.2.查看修改后的节点状态
            byte[] byte2=zooKeeper.getData("/node",null,stat);
            System.out.println("获取刚刚修改节点数据为："+new
String(byte2));

            Thread.sleep(8000);
```

```
            // 3.删除节点
            zooKeeper.delete("/node",stat.getVersion());
            System.out.println("节点删除完成！");

            zooKeeper.close();

        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (KeeperException e) {
            e.printStackTrace();
        }
    }
}
```

watch原理

在zookeeper中，引入了watcher机制来通知客户端，服务端的节点信息发生了变化。其允许客户端向服务端注册一个watcher监听，当服务端的一些指定事件触发了这个watcher，就会向指定的客户端发送一个事件通知。

watch原理步骤



1. 客户端注册Watcher到ZooKeeper服务端获取想要监听的节点状态，同时客户端本地会存储该监听器相关的信息在WatchManager中;
2. ZooKeeper服务端发生数据变更;
3. ZooKeeper服务端会主动通知会话客户端数据变更;
4. 客户端回调Watcher处理变更应对逻辑;

注册事件机制

分为两步：**绑定事件、触发事件**。

第一步：通过`getDate`、`exists`、`getChildren`这三个操作来绑定事件。

第二步：凡是事务类型的操作，都会触发监听事件。

事务操作：`create`、`delete`、`setData`

```
/**
 * @Description: 测试watch
 * @Auther: 头条号：Java思享汇
 */
public class TestWatcher {
    public static void main(String[] args) {
        final CountDownLatch countDownLatch=new
CountDownLatch(1);
        //集群的连接信息
```

```
        String
conn="127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183";
        try {
            ZooKeeper zooKeeper=new ZooKeeper(conn, 3000, new
Watcher() {
                @Override
                public void process(WatchedEvent watchedEvent)
{
                    System.out.println("事
件: "+watchedEvent.getType());

 if(Event.KeeperState.SyncConnected==watchedEvent.getState()){
                        // 连接成功
                        countDownLatch.countDown();
                        System.out.println("建立连接成功");
                    }
                }
            });
            countDownLatch.await();
            System.out.println("链接状
态: "+zooKeeper.getState());

            // 1.创建节点

 zooKeeper.create("/node","1".getBytes(),ZooDefs.Ids.OPEN_ACL_U
NSAFE, CreateMode.PERSISTENT);

            // 2.绑定事件
            Stat stat=zooKeeper.exists("/node", new Watcher() {
                @Override
                public void process(WatchedEvent watchedEvent)
{
                    // TODO 对应业务逻辑
                    // watcher是一次性操作，只能看到setData操作带来
的变化，delete操作看不到变化。所以，要在绑定一次事件，来持续监听
                    System.out.println("事件类
型: "+watchedEvent.getType()+"-->"+watchedEvent.getPath());
                }
            });
```

```
            // 通过修改的事务类型操作来触发监听事件
            // 3.setData
            stat =
zooKeeper.setData("/node","2".getBytes(),stat.getVersion());

            Thread.sleep(1000);
            // 4.delete
            zooKeeper.delete("/node",stat.getVersion());

            System.in.read();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

zookeeper中的事件和状态

事件和状态构成了zookeeper客户端连接描述的两个维度。

客户端连接状态

连接状态	状态含义
Expired	客户端和服务器在ticktime的时间周期内，是要发送心跳通知的。这是租约协议的一个实现。客户端发送request，告诉服务器其上一个租约时间，服务器收到这个请求后，告诉客户端其下一个租约时间是哪个时间点。当客户端时间戳达到最后一个租约时间，而没有收到服务器发来的任何新租约时间，即认为自己下线（此后客户端会废弃这次连接，并试图重新建立连接）。这个过期状态就是Expired状态
Disconnected	就像上面那个状态所述，当客户端断开一个连接（可能是租约期满，也可能是客户端主动断开）这是客户端和服务器的连接就是Disconnected状态
SyncConnected	一旦客户端和服务器的某一个节点建立连接（注意，虽然集群有多个节点，但是客户端一次连接到一个节点就行了），并完成一次version、zxid的同步，这时的客户端和服务器的连接状态就是SyncConnected
AuthFailed	zookeeper客户端进行连接认证失败时，发生该状态

watch事件类型

事件类型	事件含义
NodeCreated	当node-x这个节点被创建时，该事件被触发
NodeChildrenChanged	当node-x这个节点的直接子节点被创建、被删除、子节点数据发生变更时，该事件被触发。
NodeDataChanged	当node-x这个节点的数据发生变更时，该事件被触发
NodeDeleted	当node-x这个节点被删除时，该事件被触发。
None	当zookeeper客户端的连接状态发生变更时，即KeeperState.Expired、KeeperState.Disconnected、KeeperState.SyncConnected、KeeperState.AuthFailed状态切换时，描述的事件类型为EventType.None。这些状态在触发时，所记录的事件类型都是：EventType.None。

Watch特性

1. Watch是一次性的，每次都需要重新注册，并且客户端在会话异常结束时不会收到任何通知，而快速重连接时仍不影响接收通知。
2. Watch的回调执行都是顺序执行的，并且客户端在没有收到关注数据的变化事件通知之前是不会看到最新的数据，另外需要注意不要在Watch回调逻辑中阻塞整个客户端的Watch回调。
3. Watch是轻量级的，WatchEvent是最小的通信单元，结构上只包含通知状态、事件类型和节点路径。ZooKeeper服务端只会通知客户端发生了什么，并不会告诉具体内容。

zookeeper会话状态

NOT_CONNECTED - > CONNECTING ->CONNECTED ->ClOSE

curator

Curator是Netflix公司开源的一套Zookeeper客户端框架。了解过Zookeeper原生API都会清楚其复杂度。Curator帮助我们在其基础上进行封装、实现一些开发细节，包括接连重连、反复注册Watcher和NodeExistsException等。目前已经作为Apache的顶级项目出现，是最流行的Zookeeper客户端之一。从编码风格上来讲，它提供了基于Fluent的编程风格支持。除此之外，Curator还提供了Zookeeper的各种应用场景：Recipe、共享锁服务、Master选举机制和分布式计数器等。

对应官网链接：http://curator.apache.org/index.html

curator framework增删改查

方法名	描述
create	开始创建操作。调用其他方法（模式或后台），并通过调用forPath（）完成操作
delete	删除操作。调用其他方法（版本或后台），并通过调用forPath（）完成操作
checkExists	操作以检查ZNode是否存在。调用其他方法（监视或后台）并通过调用forPath（）完成操作
getData	操作以获取ZNode的数据。调用其他方法（监视，后台或获取状态）并通过调用forPath（）完成操作
setData	设置ZNode数据的操作。调用其他方法（版本或后台），并通过调用forPath（）完成操作
getChildren	操作以获取ZNode的子ZNode列表。调用其他方法（监视，后台或获取状态）并通过调用forPath（）完成操作
transaction	事务性操作，可以结合以上创建、删除、修改节点数据等完成一个事务性操作

代码示例：

Pom.xml引入

```
        <!-- 基础框架 -->
```

```
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-framework</artifactId>
            <version>4.2.0</version>
        </dependency>
        <!-- 功能 jar 分布式锁、队列等 -->
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-recipes</artifactId>
            <version>4.2.0</version>
        </dependency>
        <!-- 客户端重试策略 -->
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-client</artifactId>
            <version>4.2.0</version>
        </dependency>
```

Curaator测试类

```
/**
 * @Description: 测试Curator
 * @Auther: Java思享汇
 */
public class CuratorTest {

    //设置重置策略
    /**
     * ExponentialBackoffRetry：构造器含有三个参数
ExponentialBackoffRetry(int baseSleepTimeMs, int maxRetries,
int maxSleepMs)
     *      baseSleepTimeMs：初始的sleep时间，用于计算之后的每次重试
的sleep时间，计算公式：当前sleep时间=baseSleepTimeMs*Math.max(1,
random.nextInt(1<<(retryCount+1)))
     *      maxRetries：最大重试次数
     *      maxSleepMs：最大sleep时间，如果上述的当前sleep计算出来比
这个大，那么sleep用这个时间
     */
```

```
    RetryPolicy retryPolicy=new
ExponentialBackoffRetry(2000,5);
    //设置连接的字符串
    String
conneString="127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183";
    //工厂创建客户端
    CuratorFramework
client=CuratorFrameworkFactory.newClient(conneString,retryPolic
y);

    //工厂创建客户端的另一种方式
    CuratorFramework client2=CuratorFrameworkFactory
            .builder()
            .connectString(conneString) //zk的server地址，多个
server之间使用英文逗号分隔开
            .connectionTimeoutMs(3000) //连接超时时间，如上是
30s，默认是15s
            .retryPolicy(retryPolicy)   //失败重试策略
            .sessionTimeoutMs(3000)     //会话超时时间，如上是
50s，默认是30s
            .build();
    //开启客户端
    public CuratorTest() {
        client.start();
    }

    /**
     * 在path路径下新增节点及数据
     * @param path
     * @param data
     */
    public void create(String path, byte[] data) throws
Exception {
        String res =
client.create().creatingParentsIfNeeded().forPath(path, data);
    }

    /**
     * 删除path路径下的节点
```

```
     * @param path
     */
    public void delete(String path) throws Exception {
        client.delete().forPath(path);
    }
    /**
     * 获取path路径下的节点数据
     * @param path
     */
    public String getData(String path) throws Exception {
        byte[] res = client.getData().forPath(path);
        return new String(res);
    }
    /**
     * 设置path路径下的节点数据
     * @param path
     */
    public void setData(String path,byte[] data) throws
Exception {
        client.setData().forPath(path,data);
    }
    /**
     * path 路径下节点是否存在
     * @param path
     * stat是对znode节点的一个映射，stat=null表示节点不存在
     */
    public Stat checkExists(String path) throws Exception {
        Stat res = client.checkExists().forPath(path);
        return res;
    }
    /**
     * 获取子节点
     * @param path
     */
    public List<String> getChildren(String path) throws
Exception {
        List<String> pathList =
client.getChildren().forPath(path);
        return pathList;
```

```
    }
    //关闭客户端的连接
    public void close() {
        client.close();
    }

    //测试
    public static void main(String[] args) throws Exception {
        String path="/node";
        CuratorTest curatorTest=new CuratorTest();
        if(null!=curatorTest.checkExists(path)) {
            curatorTest.delete(path);
            System.out.println("删除路径为："+path);
        }
        //递归创建 可以支持赋值

 curatorTest.create(path+"/node1"+"/node2","hi".getBytes());
        //获取path下面的节点
        List<String> list= curatorTest.getChildren("/");
        for(String chipath:list){
            System.out.println("打印遍历的节点数据"+chipath);
        }
        curatorTest.setData(path,"hello".getBytes());
        System.out.println("打印修改完的节点数据："+new
String(curatorTest.getData(path)));

        curatorTest.close();
        System.out.println("连接客户端关闭！");
    }
}
```

三种Watcher的监听

- Path Cache

  监视一个路径下1）子节点的创建、2）删除，3）以及节点数据的更新。产生的事件会传递给注册的PathChildrenCacheListener。

- Node Cache

  监视一个节点的创建、更新、删除，并将节点的数据缓存在本地。

- Tree Cache
  Path Cache和Node Cache的“合体”，监视路径下的创建、更新、删除事件，并缓存路径下所有孩子节点的数据。

缓存的监听模式

Cache的监听模式，可以理解为利用了一个缓存的机制，同时Cache提供了反复注册的功能。简单来说Cache就是在客户端缓存了指定节点的状态，当感知到ZNode的节点变化的时候，就会触发event事件，注册的监听器就会处理这个事件。

代码示例：

```
/**
 * @Description: 测试Curator方式Watcher
 * @Auther: Java思享汇
 */
public class CuratorWatcherTest {
    static String
connectString="127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183";
    //设置重连策略
    static RetryPolicy retryPolicy=new
ExponentialBackoffRetry(3000,3);
    //使用工厂创建客户端
    static CuratorFramework client=
CuratorFrameworkFactory.newClient(connectString,retryPolicy);

    //node cache
    public static void testNodeCacheUpdate(String nodePath,
CuratorFramework curatorFramework) throws Exception {
        //1.初始化一个nodeCache
        NodeCache nodeCache = new NodeCache(curatorFramework,
nodePath, false);

        //2.初始化一个NodeCacheListener
        NodeCacheListener nodeCacheListener = new
NodeCacheListener() {
            @Override
            public void nodeChanged() throws Exception {
```

```
                ChildData childData =
nodeCache.getCurrentData();
                System.out.println("ZNode 节点的状态变化,path=" +
childData.getPath());
                System.out.println("ZNode 节点的状态变化,data=" +
new String(childData.getData(), "utf-8"));
                System.out.println("ZNode 节点的状态变化,stat=" +
childData.getStat());
            }
        };


 nodeCache.getListenable().addListener(nodeCacheListener);
        /**
         * 唯一的一个参数buildInitial代表着是否将该节点的数据立即进行缓
存。
         * 如果设置为true的话，在start启动时立即调用NodeCache的
getCurrentData方法就能够得到对应节点的信息ChildData类，
         * 如果设置为false的就得不到对应的信息。
         *
         * 使用NodeCache来监听节点的事件
         */
        nodeCache.start();//start

        // 第1次变更节点数据
        curatorFramework.setData().forPath(nodePath, "第1次更改
内容".getBytes());
        Thread.sleep(1000);

        // 第2次变更节点数据
        curatorFramework.setData().forPath(nodePath, "第2次更改
内容".getBytes());

        Thread.sleep(1000);

        // 第3次变更节点数据
        curatorFramework.setData().forPath(nodePath, "第3次更改
内容".getBytes());
        Thread.sleep(1000);
```

```
        System.in.read();

        // 第4次变更节点数据
        // client.delete().forPath(workerPath);
    }

    /**
     * 测试pathChildrenCache
     * PathChildrenCache子节点缓存用于子节点的监听，监控本节点的子节点被创建、更新或者删除
     * （1）只能监听子节点，监听不到当前节点
     * <p>
     * （2）不能递归监听，子节点下的子节点不能递归监控
     * <p>
     * 简单说下Curator的监听原理，无论是PathChildrenCache，还是TreeCache，
     * 所谓的监听，都是进行Curator本地缓存视图和ZooKeeper服务器远程的数据节点的对比
     * <p>
     * 以节点增加事件NODE_ADDED为例，所在本地缓存视图开始的时候，本地视图为空，
     * 在数据同步的时候，本地的监听器就能监听到NODE_ADDED事件。
     * 这是因为，刚开始本地缓存并没有内容，然后本地缓存和服务器缓存进行对比，发现ZooKeeper服务器有节点而本地缓存没有，
     * 这才将服务器的节点缓存到本地，就会触发本地缓存的NODE_ADDED事件。
     *
     * @param curatorFramework
     */
    //path cache
    public static void testPathChildrenCache(String nodePath, CuratorFramework curatorFramework) throws Exception {

        PathChildrenCache pathChildrenCache = new PathChildrenCache(curatorFramework, nodePath, true);
        PathChildrenCacheListener pathChildrenCacheListener = new PathChildrenCacheListener() {
            @Override
```

```
            public void childEvent(CuratorFramework
curatorFramework, PathChildrenCacheEvent
pathChildrenCacheEvent) throws Exception {
                ChildData childData =
pathChildrenCacheEvent.getData();
                switch (pathChildrenCacheEvent.getType()) {
                    case CHILD_ADDED:
                        System.out.println("子节点增加，path=" +
childData.getPath() + ",data=" + new
String(childData.getData(), "utf-8"));
                        break;
                    case CHILD_UPDATED:
                        System.out.println("子节点更新，path=" +
childData.getPath() + ",data=" + new
String(childData.getData(), "utf-8"));
                        break;
                    case CHILD_REMOVED:
                        System.out.println("子节点删除，path=" +
childData.getPath() + ",data=" + new
String(childData.getData(), "utf-8"));
                        break;
                    default:
                        break;
                }
            }
        };


 pathChildrenCache.getListenable().addListener(pathChildrenCach
eListener);

 pathChildrenCache.start(PathChildrenCache.StartMode.BUILD_INIT
IAL_CACHE);
        Thread.sleep(1000);
        for (int i = 0; i < 3; i++) {
            String childPath = nodePath + "/" + i;
            byte[] data = childPath.getBytes();
            curatorFramework.create().forPath(childPath, data);
        }
```

```
        Thread.sleep(1000);
        for (int i = 0; i < 3; i++) {
            String childPath = nodePath + "/" + i;
            curatorFramework.delete().forPath(childPath);
        }
    }

    //tree cache
    /**
     * Tree Cache可以看做是上两种的合体,
     * Tree Cache观察的是当前ZNode节点的所有数据。
     * 而TreeCache节点树缓存是PathChildrenCache的增强,
     * 不光能监听子节点，也能监听节点自身。
     *
     * @param curatorFramework
     */
    public static void testTreeCacheNode(String nodePath,
CuratorFramework curatorFramework) throws Exception {

        TreeCache treeCache = new TreeCache(curatorFramework,
nodePath);
        TreeCacheListener treeCacheListener = new
TreeCacheListener() {
            @Override
            public void childEvent(CuratorFramework
curatorFramework, TreeCacheEvent treeCacheEvent) throws
Exception {
                ChildData data = treeCacheEvent.getData();
                switch (treeCacheEvent.getType()) {
                    case NODE_ADDED:
                        System.out.println("[TreeNode]节点增加,
path=" + data.getPath() + ",data=" + new String(data.getData(),
"utf-8"));
                        break;
                    case NODE_UPDATED:
                        System.out.println("[TreeNode]节点更新,
path=" + data.getPath() + ",data=" + new String(data.getData(),
"utf-8"));
```

```
                        break;
                    case NODE_REMOVED:
                        System.out.println("[TreeNode]节点删除,
path=" + data.getPath() + ",data=" + new String(data.getData(),
"utf-8"));
                        break;
                    default:
                        break;
                }
            }
        };


 treeCache.getListenable().addListener(treeCacheListener);

        treeCache.start();
        Thread.sleep(1000);
        for (int i = 0; i < 3; i++) {
            String childPath = nodePath + "/" + i;
            byte[] data = childPath.getBytes();

 curatorFramework.create().creatingParentsIfNeeded().forPath(ch
ildPath, data);
        }

        Thread.sleep(1000);
        for (int i = 0; i < 3; i++) {
            String childPath = nodePath + "/" + i;
            curatorFramework.delete().forPath(childPath);
        }

        curatorFramework.setData().forPath(nodePath, "update
parent".getBytes());
    }


    public static void main(String[] args) throws Exception {
        client.start();
        String nodePath = "/watcher";
```

```
        //node cache
//
CuratorWatcherTest.testNodeCacheUpdate(nodePath,client);

        //path cache
//
CuratorWatcherTest.testPathChildrenCache(nodePath,client);

        //tree cache
        CuratorWatcherTest.testTreeCacheNode(nodePath,client);
    }
}
```

zookeeper应用场景

Watcher实现事件监听

事件监听主要是通过Watcher机制来实现的，上一期已经讲过，主要有三种方式：NodeCache、PathChildrenCache、TreeCache。

分布式计数器

DistributedAtomicLong 相对于java中的 AtomicLong。主要的方法有：

1. `get()`：获取当前值
2. `increment()`： 加一
3. `decrement()`：减一
4. `add()`： 增加特定的值
5. `subtract()`：减去特定的值
6. `trySet()`：尝试设置计数值
7. `forceSet()`：强制设置计数值

需要检查设置完数据的返回结果的succeeded()， 它代表此操作是否成功。 如果操作成功， preValue()代表操作前的值， postValue()代表操作后的值。

代码示例：

```
/**
 * @Description: zk实现分布式计数器
 * @Auther: Java思享汇
 */
public class DistributedAtomicLongTest {
    private static final int CLIENT_COUNT = 5;
    private static final String PATH = "/counter";

    public static void main(String[] args) throws Exception {
        try{
            CuratorFramework client =
CuratorFrameworkFactory.newClient("127.0.0.1:2181,127.0.0.1:218
2,127.0.0.1:2183",3000,3000, new ExponentialBackoffRetry(1000,
3,Integer.MAX_VALUE));
            client.start();

            List<DistributedAtomicLong> examples = new
ArrayList<>();
            ExecutorService service =
Executors.newFixedThreadPool(CLIENT_COUNT);

            for (int i = 0; i < CLIENT_COUNT; ++i) {
                System.out.println("第"+i+"次执行！");
                //类似于JUC AtomicLong CAS乐观锁的方式
                final DistributedAtomicLong count = new
DistributedAtomicLong(client, PATH, new RetryNTimes(10, 10));
                examples.add(count);
                Callable<Void> task = new Callable<Void>() {
                    @Override
                    public Void call() throws Exception {
                        try {
                            AtomicValue<Long> value =
count.increment();
                            System.out.println("操作结果： " +
value.succeeded());
                            if (value.succeeded()){
                                System.out.println("操作成功：操
作前的值为： " + value.preValue() + " 操作后的值为： " +
value.postValue());
```

```
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    return null;
                }
            };
            service.submit(task);
        }
        service.shutdown();
        service.awaitTermination(10, TimeUnit.MINUTES);
    }catch (Exception e){
        e.printStackTrace();
    }
  }
}
```

服务注册与发现

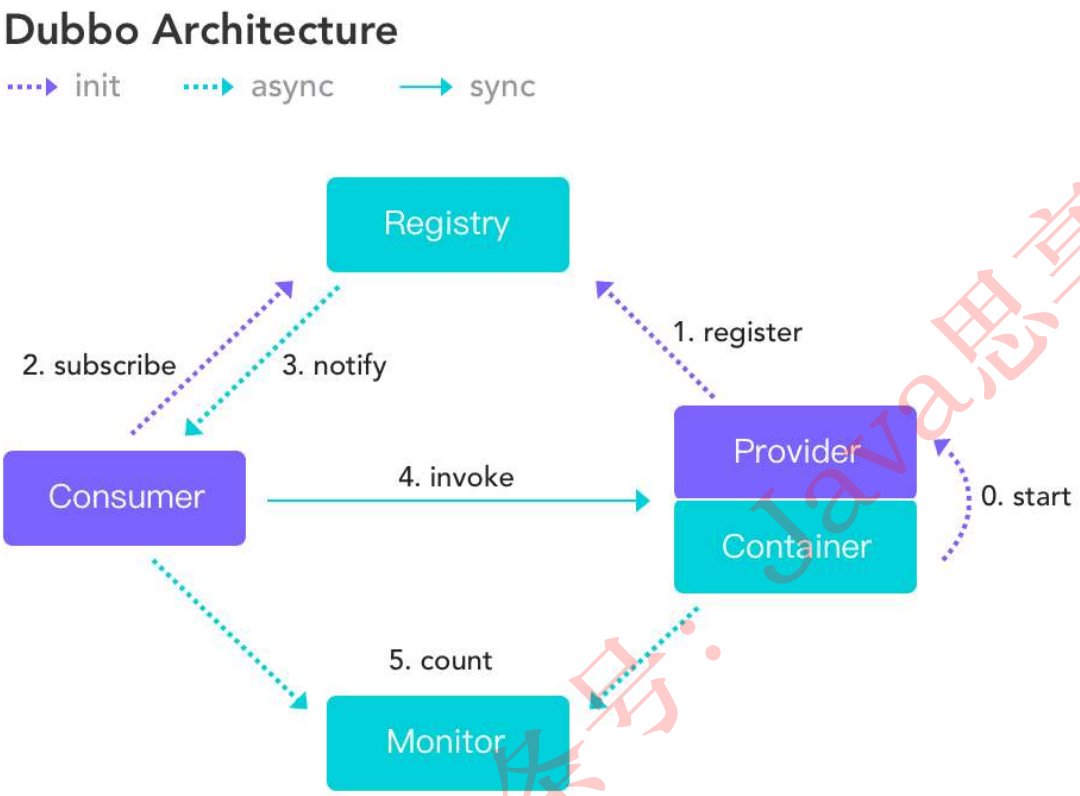
随着业务增加,以前简单的系统已经变得越来越复杂，单纯的提升服务器性能也不是办法,而且代码也是越来越庞大，维护也变得越来越困难，这一切都催生了新的架构设计风格 – 微服务架构的出现。

微服务给我们带来了很多好处，例如：独立可扩展、易维护。但是随着应用的分解，微服务的引入，服务越来越多，业务系统与服务系统之间的调用，该架构的问题暴露出来：最明显的问题是所有的请求都需要nginx来转发，随着内部服务的越来越多，服务上线都需要修改nginx的配置。一旦内部网络调整，nginx upstream也需要做对应的配置调整，并且每个服务都还需要维护nginx的地址。所以,服务注册中心诞生了。

什么是服务注册中心

注册中心可以对服务上下线做统一管理。每个工作服务器都可以作为数据的发布方向集群注册自己的基本信息，而让某些监控服务器作为订阅方，订阅工作服务器的基本信息，当工作服务器的基本信息发生改变如上下线、服务器角色或服务范围变更，监控服务器可以得到通知并响应这些变化。服务自动注册与发现后,不再需要写死服务提供方地址,注册中心基于接口名查询服务提供者的IP地址,并且能够平滑添加或删除服务提供者。

dubbo结构图



Consumer服务消费者，Provider服务提供者。Container服务容器。消费当然是invoke提供者了，invoke这条实线按照图上的说明当然同步的意思了。但是在实际调用过程中，Provider的位置对于Consumer来说是透明的，上一次调用服务的位置（IP地址）和下一次调用服务的位置，是不确定的。这个地方就需要使用注册中心来实现软负载。

服务提供者先启动start，然后注册register服务。消费订阅subscribe服务，如果没有订阅到自己想获得的服务，它会不断的尝试订阅。新的服务注册到注册中心以后，注册中心会将这些服务通过notify到消费者。这里的注册中心用到的就是zookeeper。

核心类

我们通常在调用服务的时候，需要知道服务的地址，端口，或者其他一些信息，通常情况下，我们是把他们写到程序里面，但是随着服务越来越多，维护起来也越来越费劲，更重要的是，由于地址都是在程序中配置的，我们根本不知道远程的服务是否可用，当我们增加或者删除服务，我们又需要到配置文件

中配置么？ 这时候，Zookeeper帮大忙了，我们可以把我们的服务注册到Zookeeper中，创建一个临时节点（当连接断开之后，节点将被删除），存放我们的服务信息（url，ip，port等信息），把这些临时节点都存放在以serviceName命名的节点下面，这样我们要获取某个服务的地址，只需要到Zookeeper中找到这个path，然后就可以读取到里面存放的服务信息，这时候我们就可以根据这些信息调用我们的服务。这样，通过Zookeeper我们就做到了动态的添加和删除服务，做到了一旦一个服务时效，就会自动从Zookeeper中移除。

Curator Service Discovery就是为了解决这个问题而生的，它对此抽象出了ServiceInstance、ServiceProvider、ServiceDiscovery三个接口，通过它我们可以很轻易的实现Service Discovery。

ServiceInstance

Curator中使用ServiceInstance作为一个服务实例，ServiceInstances具有名称，ID，地址，端口和/或ssl端口以及可选的payload(用户定义）。
ServiceInstances以下列方式序列化并存储在ZooKeeper中：

```
base path
       |_______ service A name
                    |__________ instance 1 id --> (serialized ServiceInstance)
                    |__________ instance 2 id --> (serialized ServiceInstance)
                    |__________ ...
       |_______ service B name
                    |__________ instance 1 id --> (serialized ServiceInstance)
                    |__________ instance 2 id --> (serialized ServiceInstance)
                    |__________ ...
       |_______ ...
```


ServiceProvider

ServiceProvider是主要的抽象类。它封装了发现服务为特定的命名服务和提供者策略。提供者策略方案是从一组给定的服务实例选择一个实例。有三个捆绑策略:轮询调度、随机和粘性(总是选择相同的一个)。
ServiceProviders是使用ServiceProviderBuilder分配的。消费者可以从从ServiceDiscovery获取ServiceProviderBuilder（参见下文）。
ServiceProviderBuilder允许您设置服务名称和其他几个可选值。
必须通过调用start()来启动ServiceProvider 。完成后，您应该调用close()。
ServiceProvider中有以下两个重要方法：

```
/** 获取一个服务实例 */
public ServiceInstance<T> getInstance() throws Exception;
/** 获取所有的服务实例 */
public Collection<ServiceInstance<T>> getAllInstances() throws
Exception;
```

注意：在使用curator 2.x(ZooKeep3.4.x)时，服务提供者对象必须由应用程序缓存并重用。因为服务提供者添加的内部NamespaceWatcher对象无法在ZooKeep3.4.x中删除，所以为每个对相同服务的调用创建一个新的服务提供者最终将耗尽JVM的内存。

ServiceDiscovery

为了创建ServiceProvider,你必须有一个ServiceDiscovery。它是由一个ServiceDiscoveryBuilder创建。开始必须调用start()方法。当使用完成应该调用close()方法。
如果特定实例有I/O错误，等等。您应该调用ServiceProvider.NoteError()，并传入实例。ServiceProvider将临时将有错误的实例视为“关闭”。实例的阈值和超时是通过DownInstancePolicy设置的，该策略可以传递给ServiceProviderBuilder(注意：如果没有指定默认DownInstancePolicy，则使用默认DownInstancePolicy)。

API介绍

注册/注销服务

通常，将应用程序的服务描述符传递给ServiceDiscovery构造函数，它将自动注册/注销服务。但是，如果需要手动执行此操作，请使用以下方法：

```
/** 注册服务 */
public void registerService(ServiceInstance<T> service) throws
Exception;
/** 注销服务 */
public void unregisterService(ServiceInstance<T> service)
throws Exception;
```

查询服务

您可以查询所有服务名称、特定服务的所有实例或单个服务实例。

```
/** 查询所有服务名称 */
public Collection<String> queryForNames() throws Exception;
/** 查询特定服务的所有实例 */
public Collection<ServiceInstance<T>>  queryForInstances(String
name) throws Exception;
/** 查询单个服务实例 */
public ServiceInstance<T> queryForInstance(String name, String
id) throws Exception;
```

服务缓存

上述每个查询方法都直接调用ZooKeeper。如果经常查询服务，还可以使用ServiceCache。它在内存中缓存特定服务的实例列表。它使用Watcher监听使列表保持最新。
可以通过ServiceDiscovery.serviceCacheBuilder()返回的构建器分配ServiceCache 。通过调用start()启动ServiceCache对象，完成后，应调用close()。然后可以通过调用以下内容获取服务的当前已知实例列表：

```
/** 获取缓存服务列表 */
public List<ServiceInstance<T>> getInstances();
ServiceCache支持在Watcher更新实例列表时收到通知的侦听器：
/**
 * Listener for changes to a service cache
 */
public interface ServiceCacheListener extends
ConnectionStateListener {
    /**
     * Called when the cache has changed (instances
added/deleted, etc.)
     */
    public void cacheChanged();
}
```

代码示例

pom添加依赖

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-x-discovery</artifactId>
    <version>4.2.0</version>
</dependency>
```

服务定义信息类

该类中可以自定义一些自己想要的属性,例如方法需要的参数,方法的描述等等。

```
/**
 * @Description: 服务定义信息
 * @Auther: Java思享汇
 */
public class InstanceDetails {
    public static final String ROOT_PATH = "/service";

    /** 该服务拥有哪些方法 */
    public Map<String,Service> services = new HashMap<>();

    /** 服务描述 */
    private String serviceDesc;

    public InstanceDetails(){
        this.serviceDesc = "";
    }

    public InstanceDetails(String serviceDesc){
        this.serviceDesc = serviceDesc;
    }

    public String getServiceDesc() {
        return serviceDesc;
    }

    public void setServiceDesc(String serviceDesc) {
        this.serviceDesc = serviceDesc;
    }
```

```
    public Map<String, Service> getServices() {
        return services;
    }

    public void setServices(Map<String, Service> services) {
        this.services = services;
    }

    public static class Service {
        /** 方法名称 */
        private String methodName;

        /** 方法描述 */
        private String desc;

        /** 方法所需要的参数列表 */
        private List<String> params;

        public String getMethodName() {
            return methodName;
        }

        public void setMethodName(String methodName) {
            this.methodName = methodName;
        }

        public String getDesc() {
            return desc;
        }

        public void setDesc(String desc) {
            this.desc = desc;
        }

        public List<String> getParams() {
            return params;
        }

        public void setParams(List<String> params) {
```

```
            this.params = params;
        }
    }
}
```

服务生产者1

```
/**
 * @Description: 服务提供者1
 * @Auther: Java思享汇
 */
public class ProviderService1 {
    public static void main(String[] args) throws Exception {
        CuratorFramework client =
CuratorFrameworkFactory.newClient("127.0.0.1:2181,127.0.0.1:218
2,127.0.0.1:2183",
                2000,2000, new ExponentialBackoffRetry(1000,
3));
        client.start();
        client.blockUntilConnected();

        //服务构造器
        ServiceInstanceBuilder<InstanceDetails> sib =
ServiceInstance.builder();
        //该服务中所有的接口
        Map<String,InstanceDetails.Service> services = new
HashMap<>();

        // 添加订单服务接口
        //服务所需要的参数
        ArrayList<String> addOrderParams = new ArrayList<>();
        addOrderParams.add("createTime");
        addOrderParams.add("state");
        InstanceDetails.Service addOrderService = new
InstanceDetails.Service();
        addOrderService.setDesc("添加订单");
        addOrderService.setMethodName("addOrder");
        addOrderService.setParams(addOrderParams);
        services.put("addOrder",addOrderService);
```

```
        //添加删除订单服务接口
        ArrayList<String> delOrderParams = new ArrayList<>();
        delOrderParams.add("orderId");
        InstanceDetails.Service delOrderService = new
InstanceDetails.Service();
        delOrderService.setDesc("删除订单");
        delOrderService.setMethodName("delOrder");
        delOrderService.setParams(delOrderParams);
        services.put("delOrder",delOrderService);

        //服务的其他信息
        InstanceDetails payload = new InstanceDetails();
        payload.setServiceDesc("订单服务");
        payload.setServices(services);

        //将服务添加到 ServiceInstance
        ServiceInstance<InstanceDetails> orderService =
sib.address("127.0.0.1")
                .port(8080)
                .name("OrderService")
                .payload(payload)
                .uriSpec(new UriSpec("{scheme}://{address}:
{port}"))
                .build();

        //构建 ServiceDiscovery 用来注册服务
        ServiceDiscovery<InstanceDetails> serviceDiscovery =
ServiceDiscoveryBuilder.builder(InstanceDetails.class)
                .client(client)
                .serializer(new
JsonInstanceSerializer<InstanceDetails>(InstanceDetails.class))
                .basePath(InstanceDetails.ROOT_PATH)
                .build();
        //服务注册
        serviceDiscovery.registerService(orderService);
        serviceDiscovery.start();

        System.out.println("第一台服务注册成功......");
```

```
        TimeUnit.SECONDS.sleep(Integer.MAX_VALUE);

        serviceDiscovery.close();
        client.close();
    }
}
```

服务生产者2

服务生产者2和服务生产者1代码基本一致，唯一不同的地方只是端口号改了一下，将端口号设置为8081,用来模拟两台不同的机器。同时将启动打印日志改了下。

```
sib.address("127.0.0.1").port(8081)
```

```
System.out.println("第二台服务注册成功.....");
```

服务消费者

```
/**
 * @Description: 服务消费者
 * @Auther: Java思享汇
 */
public class ConsumerClient {
    public static void main(String[] args) throws Exception{
        CuratorFramework client =
CuratorFrameworkFactory.newClient("127.0.0.1:2181,127.0.0.1:218
2,127.0.0.1:2183", new ExponentialBackoffRetry(1000, 3));
        client.start();
        client.blockUntilConnected();

        ServiceDiscovery<InstanceDetails> serviceDiscovery =
ServiceDiscoveryBuilder.builder(InstanceDetails.class)
                .client(client)
                .basePath(InstanceDetails.ROOT_PATH)
                .serializer(new
JsonInstanceSerializer<InstanceDetails>(InstanceDetails.class))
                .build();
```

```
        serviceDiscovery.start();

        boolean flag = true;

        //死循环来不停的获取服务列表,查看是否有新服务发布
        while(flag){

            //根据名称获取服务
            Collection<ServiceInstance<InstanceDetails>>
services = serviceDiscovery.queryForInstances("OrderService");
            if(services.size() == 0){
                System.out.println("当前没有发现服务");
                Thread.sleep(5 * 1000);
                continue;
            }

            for(ServiceInstance<InstanceDetails> service :
services) {

                //获取请求的scheme 例如：http://127.0.0.1:8080
                String uriSpec = service.buildUriSpec();
                //获取服务的其他信息
                InstanceDetails payload = service.getPayload();

                //服务描述
                String serviceDesc = payload.getServiceDesc();
                //获取该服务下的所有接口
                Map<String, InstanceDetails.Service> allService
= payload.getServices();
                Set<Map.Entry<String, InstanceDetails.Service>>
entries = allService.entrySet();

                for (Map.Entry<String, InstanceDetails.Service>
entry: entries) {
                    System.out.println(serviceDesc +uriSpec
                            + "/" + service.getName()
                            + "/" + entry.getKey()
                            + " 该方法需要的参数为："
```

```
                                +
entry.getValue().getParams().toString());
                    }
                }
                System.out.println("----------------------");
                Thread.sleep(10*1000);
            }
        }
}
```

测试结果：

先启动服务消费者,当前没有发现服务,启动服务生产者1，日志打印生产者1的服务列表, 再启动服务生产者2 日志打印生产者1 和 生产者2的服务列表,然后停止生产者1 服务,日志只打印生产者2的服务列表,最后全停掉,打印当前没有服务。

实现分布式锁

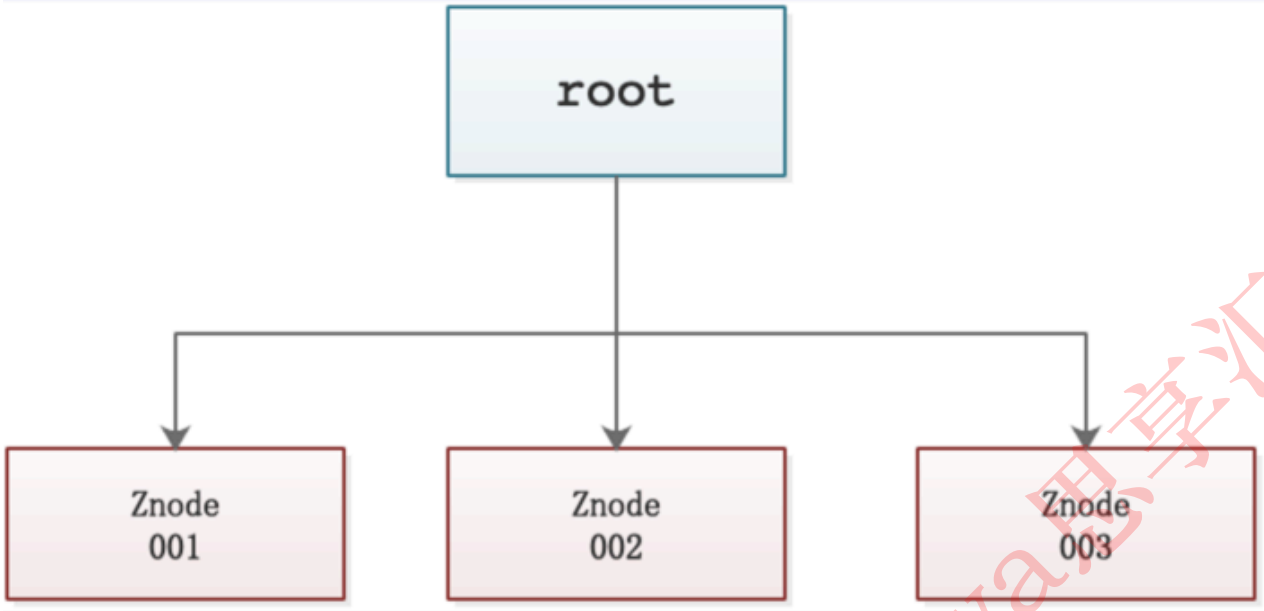
首先我们在将分布式锁应用的时候需要先了解zk节点的四种类型，Zookeeper的数据存储结构是一种树状结构，其有节点（Znode）组成，要实现分布式锁，先了解一下Znode的几种类型：持久节点、持久顺序节点、临时节点、临时顺序节点

1、持久节点

默认的节点类型。创建节点的客户端与zookeeper断开连接后，该节点依旧存在 。

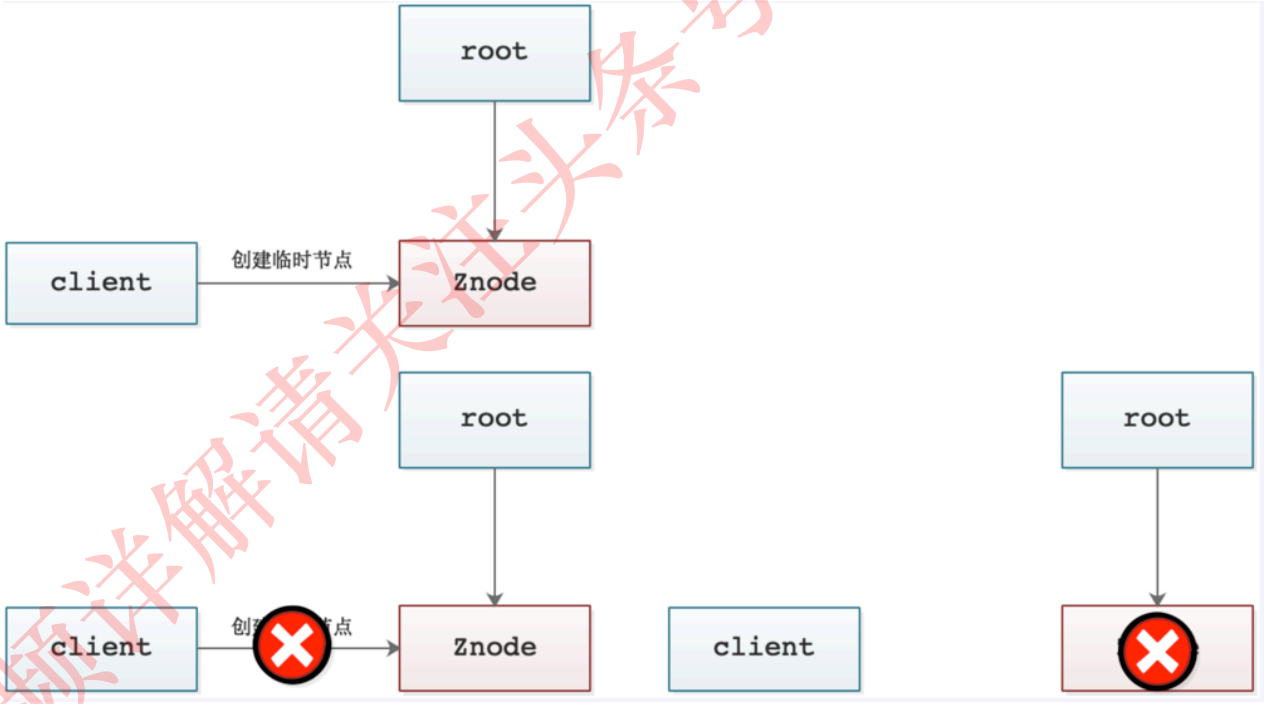
2、持久顺序节点

在创建节点时，Zookeeper根据创建的时间顺序给该节点名称进行编号。



3、临时节点

和持久节点相反，当创建节点的客户端与zookeeper断开连接后，临时节点会被删除。



4、临时顺序节点

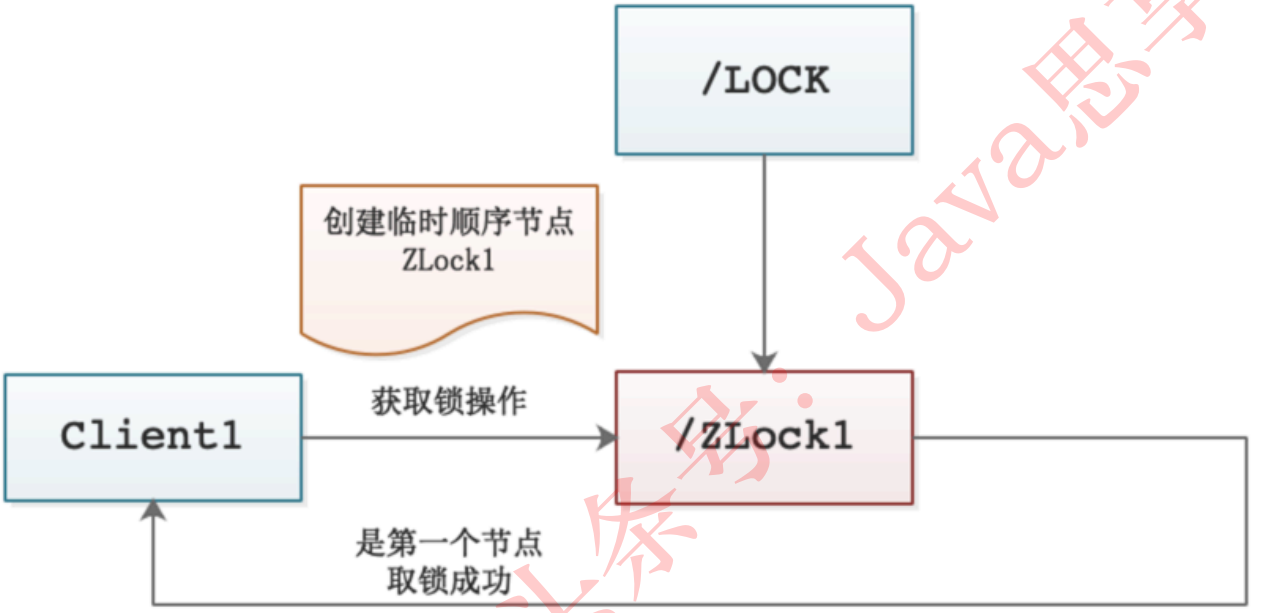
临时顺序节点结合临时节点和顺序节点的特点：在创建节点时，Zookeeper根据创建的时间顺序给节点名称进行编号；当创建节点的客户端与Zookeeper断开连接后，临时节点会被删除。

ZK实现分布式锁的原理

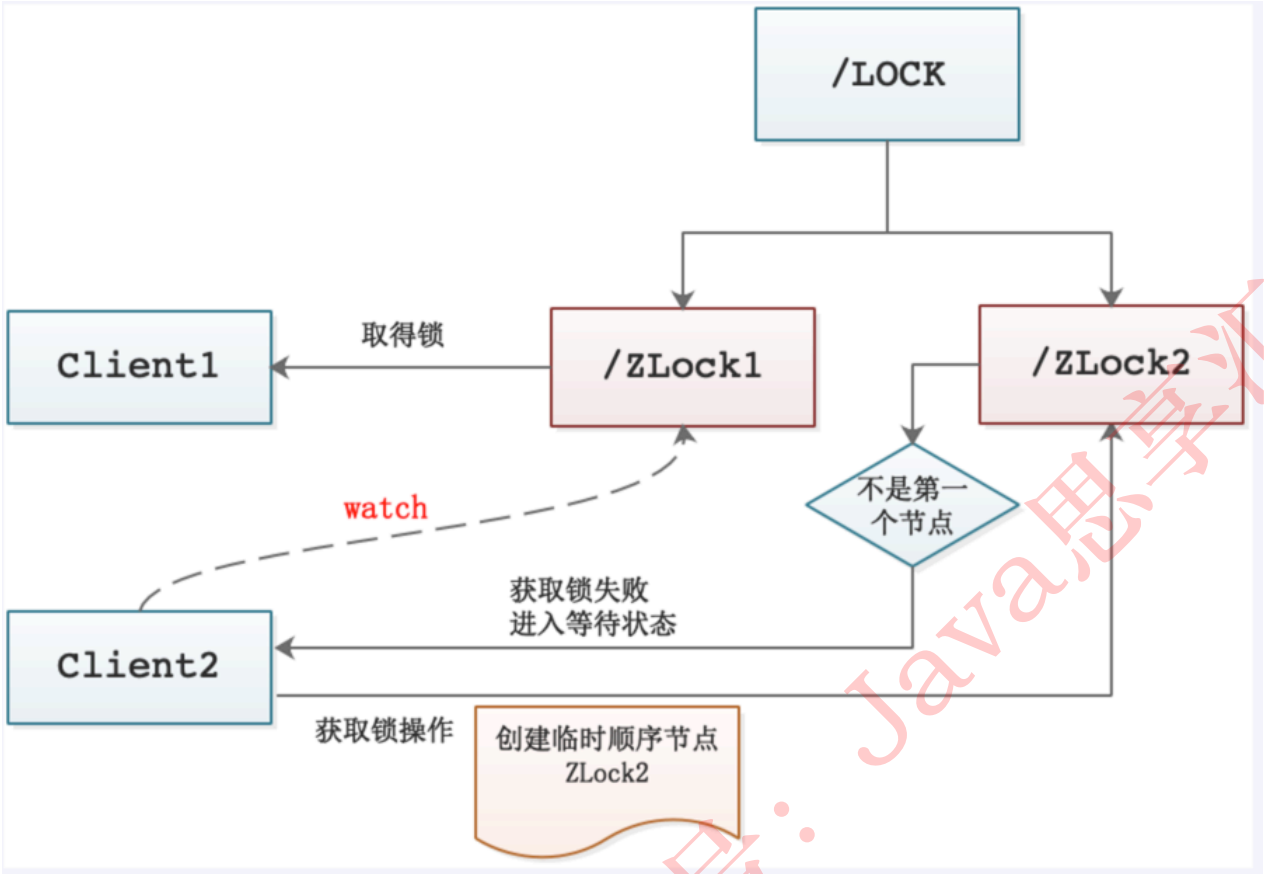
其应用了临时顺序节点的特性（zk客户端断开后，临时节点则删除）

获取锁步骤如下：

1、先创建一个持久节点LOCK，所有要获得锁的Client，需要在LOCK节点上创建一个临时顺序节点。例如：Client1要获取锁，需要在LOCK上创建一个临时顺序节点ZLock1；Client1查找Lock下面所有的临时顺序节点判断自己所创建的节点Lock1是不是最靠前的一个。如果是第一个节点，则成功获得锁。



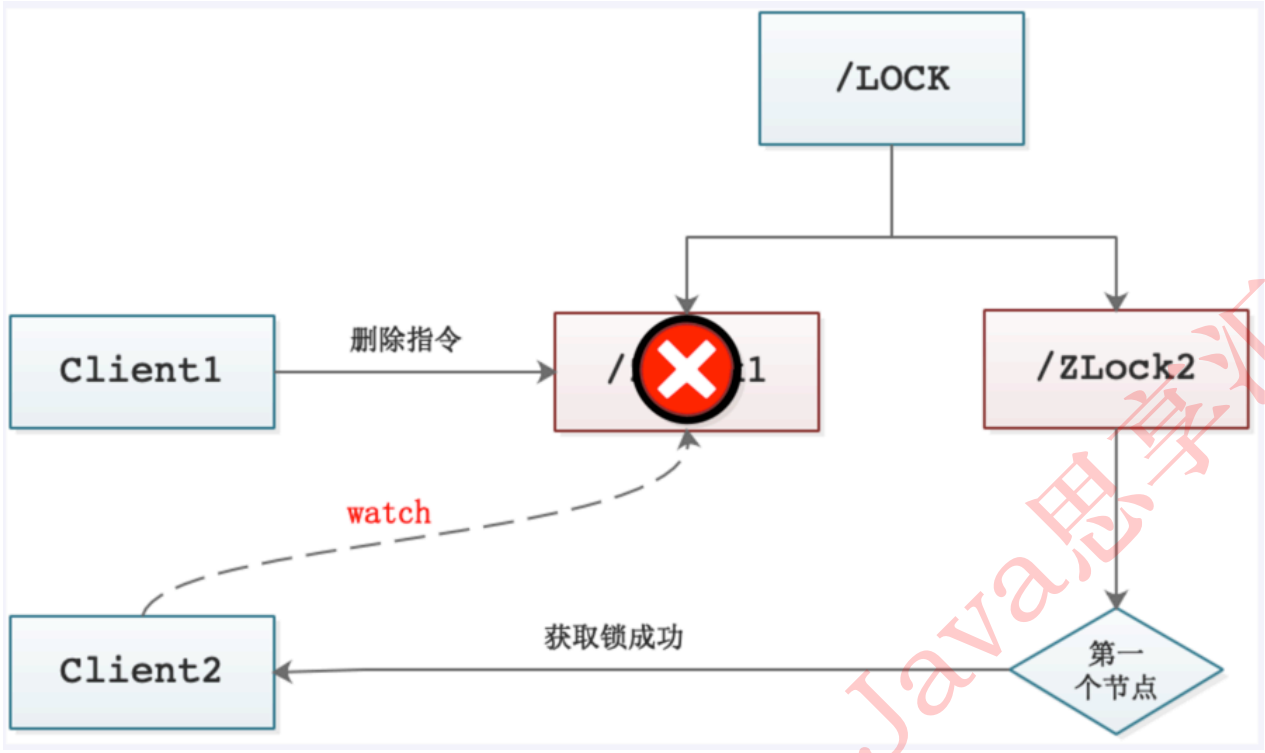
2、当Client2获取锁时，同样在LOCK节点上创建一个临时顺序节点ZLock2；Client2查找LOCK下面所有的临时顺序节点，判断自己所创建的节点Lock2是不是顺序最靠前的一个，结果不是；于是，Client2向排序仅比它靠前的节点ZLock1注册Watcher，用于监听ZLock1节点是否存在。这意味着Client2获取锁失败，进入了等待状态。



这样一来，Client1得到了锁，Client2监听了ZLock1，取锁的过程大致就是这样，那么其如何释放锁呢--删除对应节点就可以。

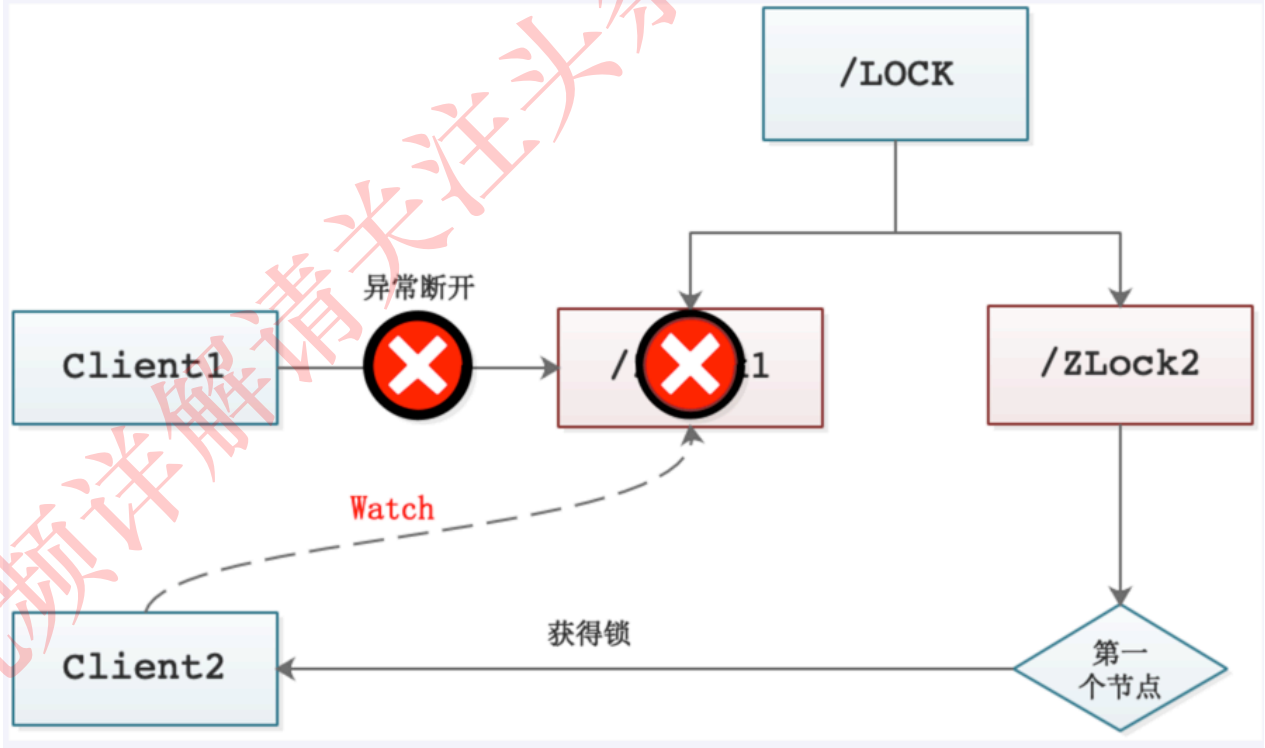
释放锁有两种情况：

1、正常情况，任务完成，Client端主动释放
当任务完成时，Client1会主动调用删除节点ZLock1的指令，以此释放锁。



2、异常情况，Client异常断开

获得锁的Client1在任务执行过程中，如果出现异常断开与Zookeeper服务端的链接。根据临时节点的特性，相关联的节点ZLock1会自动删除。



不管哪种情况，由于Client2一直监听着ZLock1的存在状态，当ZLock1节点被删除，Client2会立刻收到通知。这时候Client2会再次查询Lock下面的所有节点，确认自己创建的节点ZLock2是不是第一个节点。如果是，则Client2就会成功获得锁。

Curator代码实现

使用

创建**InterProcessMutex**实例
InterProcessMutex提供了两个构造方法，传入一个CuratorFramework实例和一个要使用的节点路径，InterProcessMutex还允许传入一个自定义的驱动类，默认是使用StandardLockInternalsDriver。

```
public InterProcessMutex(CuratorFramework client, String path);
public InterProcessMutex(CuratorFramework client, String path,
LockInternalsDriver driver);
```

获取锁

使用acquire方法获取锁,acquire方法有两种:

```
public void acquire() throws Exception;
```

获取锁，一直阻塞到获取到锁为止。获取锁的线程在获取锁后仍然可以调用acquire() 获取锁(可重入)。 锁获取使用完后，调用了几次acquire(),就得调用几次release()释放。

```
public boolean acquire(long time, TimeUnit unit) throws
Exception;
```

与acquire()类似，等待time * unit时间获取锁，如果仍然没有获取锁，则直接返回false。

释放锁

使用release() 方法释放锁
线程通过acquire()获取锁时，可通过release()进行释放，如果该线程多次调用了acquire()获取锁，则如果只调用 一次release()该锁仍然会被该线程持有。

注意：同一个线程中InterProcessMutex实例是可重用的，也就是不需要在每次获取锁的时候都new一个InterProcessMutex实例，用同一个实例就好。

锁撤销

InterProcessMutex 支持锁撤销机制，可通过调用makeRevocable()将锁设为可撤销的，当另一线程希望你释放该锁时，实例里的listener会被调用。 撤销机制是协作的。

```
public void makeRevocable(RevocationListener<T> listener);
```

如果你请求撤销当前的锁， 调用Revoker类中的静态方法attemptRevoke()要求锁被释放或者撤销。如果该锁上注册有RevocationListener监听，该监听会被调用。

```
public static void  attemptRevoke(CuratorFramework client,
String path) throws Exception;
```

使用场景

起五个线程，即五个窗口卖票，五个客户端分别有50张票可以卖，先是尝试获取锁，操作资源后，释放锁。

代码示例：

定义火车票资源类

```
/**
 * @Description: 火车票资源
 * @Auther: Java思享汇
 */
public class FakeLimitedResource {
    //总共250张火车票
    private Integer ticket = 250;

    public void use() throws InterruptedException {
        try {
            System.out.println("火车票还剩"+(--ticket)+"张！");
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

无锁多线程测试类

```
/**
 * @Description: 无锁情况下测试
 * @Auther: Java思享汇
 */
public class TestThread {
    private static final int QTY = 5;
    private static final int REPETITIONS = 50;

    public static void main(String[] args) {
        //模拟或者票一次只能由一个进程访问
        final FakeLimitedResource resource = new
FakeLimitedResource();
        //定义默认初始化为5个线程的线程池
        ExecutorService service =
Executors.newFixedThreadPool(QTY);
        for ( int i = 0; i < QTY; ++i ){
            Callable<Void> task = new Callable<Void>() {
                @Override
                public Void call() throws Exception {
                    //模拟一个线程买50张票
                    for ( int j = 0; j < REPETITIONS; ++j ) {
                        resource.use();
                    }
                    return null;
                }
            };
            service.submit(task);
        }
        service.shutdown();
    }
}
```

Curator实现并发操作类

```
/**
 * @Description: 并发操作类
 * @Auther: Java思享汇
```

```
 */
public class LockingExample {
    private static final int QTY = 5;
    private static final int REPETITIONS = 50;
    private static final String CONNECTION_STRING =
"127.0.0.1:2181,127.0.0.1:2182,127.0.0.1:2183";
    private static final String PATH = "/lock";

    public static void main(String[] args) throws Exception {

        //模拟或者票一次只能由一个进程访问
        final FakeLimitedResource resource = new
FakeLimitedResource();
        //定义默认初始化为5个线程的线程池
        ExecutorService service =
Executors.newFixedThreadPool(QTY);
        try {
            for ( int i = 0; i < QTY; ++i ){
                final int index = i;
                Callable<Void> task = new Callable<Void>() {
                    @Override
                    public Void call() throws Exception {
                        //获取zk集群连接
                        CuratorFramework client =
CuratorFrameworkFactory.newClient(CONNECTION_STRING,
                                new
ExponentialBackoffRetry(1000, 3,Integer.MAX_VALUE));
                        try {
                            client.start();
                            ExampleClientThatLocks example =
new ExampleClientThatLocks(client, PATH, resource, "Client " +
index);
                            //模拟一个线程买50张票
                            for ( int j = 0; j < REPETITIONS;
++j ) {
                                example.doWork(10,
TimeUnit.SECONDS);
                            }
                        }catch (Exception e ){
```

```
                            e.printStackTrace();
                        }finally{

 CloseableUtils.closeQuietly(client);
                        }
                        return null;
                    }
                };
                service.submit(task);
            }
            service.shutdown();
            service.awaitTermination(10, TimeUnit.MINUTES);
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

客户端获取锁

```
/**
 * @Description: 客户端锁操作
 * @Auther: Java思享汇
 */
public class ExampleClientThatLocks {
    // 锁对象
    private final InterProcessMutex lock;
    //火车票共享资源
    private final FakeLimitedResource resource;
    //客户端名称
    private final String clientName;

    public ExampleClientThatLocks(CuratorFramework client,
String lockPath, FakeLimitedResource resource, String
clientName) {
        this.resource = resource;
        this.clientName = clientName;
        lock = new InterProcessMutex(client, lockPath);
    }
```

```
    public void doWork(long time, TimeUnit unit) throws
Exception {
        //尝试获取锁
        if (!lock.acquire(time, unit) ) {
            throw new IllegalStateException(clientName + "
could not acquire the lock");
        }
        try {
            System.out.println(clientName + " has the lock");
            //操作资源
            resource.use();
        } finally {
            System.out.println(clientName + " releasing the
lock");
            //释放锁
            lock.release();
        }
    }
}
```

至此小结～

对应视频合集：https://www.ixigua.com/6832814381701530116?id=6832542802853757448&logTag=_tN2AaBhF5LWIMl18PPOU