## Kafka部分

1. kafka集群扩容和数据迁移？

集群扩容：

利用kafka-ressign-partitions.sh工具

①新启动brokers

注意brokerid以及/etc/hosts的添加和修改，此时仅仅是增加了broker，但是topic数据并不会均分到新增的节点上

②生成分配计划generage

./bin/kafka-reassign-partitions.sh --zookeeper 192.168.2.225:2183 --topics-to-move-json-file migration-push-token-topic.json --broker-list "101,,102,103,104,105,106" --generate

关于脚本内容网上有，这一步的目的是把原来101 102 103上的指定的topics数据均分到扩容后的集群。

③执行分配execute

根据第一步生成的分配计划

kafka-reassign-partitions.sh --zookeeper $ZK\_CONNECT --reassignment-json-file topic-reassignment.json --execute

④检查分配状态verify

kafka-reassign-partitions.sh --zookeeper $ZK\_CONNECT --reassignment-json-file topic-reassignment.json --verify

数据迁移分两种情况：

第一：是否是均分的数据迁移，例如123 -> 23456

第二：是否是单一broker数据迁移，比如更换某一台broker例如 123 -> 234

第一种情况解决方案：完全可以按照上述集群扩容的方式做，只需要改动将101,102,103,104,105,106中去掉不再使用的broker即可，比如102,103,104,105,106，去掉了某一台服务器，剩下的数据均分到5台上，当然也可以不去除服务器仅仅是将某些topic方到456上等等各种情况都属于第一种

第二种情况解决方案：不需要generate、execute、verify三部，只需要后两步，只需要写一个ressign.json文件规定将1上面所有的topic+partiion全部迁移到指定brokerid如4上即可。然后直接运行--execute。就可以完成kafka单台服务器的替换过程。

1. kafka如何保证数据高可用，副本是如何进行数据同步的，leader是如何进行选举的？

注意这个问题：如果只答出副本冗余机制的话那么面试可能会被pass哦！

①replicas副本冗余机制

②failover失败转移

其一：多副本冗余机制

说白了就是partition的replications，一个leader，多个follower

注意：任何一个partition只有leader才能进行对外的读写！！

Follower会不断的向leader拉取最新的数据，当leader有新数据的时候就进行同步。

其二：Failover失败转移

极端情况，当某个partition的leader不可用时怎么办，因为数据的读写是请求leader的啊，这种情况下肯定是要从follower中选举leader的。而这个过程是怎样的的呢

选举机制：

①普通选举

如果知识leader不可用，那么只需要从zk上的ISR列表中找一个follower作为leader即可

②脏选举机制

这种情况发生在所有的副本都不可用了，才是着分区停止服务，此时进行脏选举机制，也就是说选取任何一个激励可用follower作为leader，这种方式不能保证一致性

③禁止脏选举机制

等待ISR列表中副本复活，虽然可以保证一致性，但是需要很长时间

实现：在实际实现代码中，并没有这么简单，而是通过controller来实现，因为在实际业务中可能存在成千上万的partition，如果一台服务器宕机，可能出现很多leader不可用的情况，所以需要从ontroller，而controller是kafka随机选择的一个broker作为controller leader，作用如下：

关于controller还是再总结一下把，在下面

1. 你知道ISR吗，什么情况下会导致ISR列表中的follower见少？

先说明一点，无论是leader和follower都叫做副本！！不然创建topic的时候指定副本个数为1时难道就没有副本吗，leader也是副本

ISR——in sync replicas：指的就是数据保证最新的那些副本，保存在zk上：其中肯定有leader，还有跟leader同步的follower，为什么会有ISR，因为我们不能把保证所有的follower都是实时很快同步数据，因为他们可能处于GC或者网络等问题...无法及时同步数据，当然了正常情况下总会同步的。

所以机制是：正常情况下ISR列表中保存着所有副本，而当其中一个follower非同步时，leader就会将这个follower剔除ISR列表，等待回复同步之后再加入列表。所以后面介绍的acks参数针对的是ISR列表而不是真正所有的follower，不然的话延时就大了！

从ISR剔除follower的情况：

①增加副本

②GC

③IO瓶颈

④follower失效

1. 好了问你一个重要的问题：acks参数你用过吧，你觉得它的意义何在，你顺便说说kafka同步复制和异步复制的流程吧？

首先：acks参数并不是kafka broker的参数，而是producer的参数

其次：acks参数有三个选项，分别是0,1，all

当acks=0时，表示kafkaproducer客户端只要把消息发送出去就完活，不管这条message是否落到partition的leader的磁盘上，默认发送成功

当acks=1时，这个是默认的设置！只要partition leader接受到小时而且写入本地磁盘了就认为成功了！不管follower是否同步这条message

当acks=all时，这个意思很明了了，就是leader接受到消息之后还必须要求ISR列表里跟leader保持同步的那些follwer都要把消息同步过去才能认为是成功的。

需要注意的是这三种情况都不能保证数据一定持久化/接收成功，看起来第三种acks=all是可以的，但是当副本数=1时，leader挂了根本没有follower，所以要是用acks=all必须要保证副本是>=2！

同步复制就是acks=all，异步复制就是acks=1的情况。

我们已经知道了ISR，这个列表中的副本是需要同步的partition。

同步复制：

①producer联系这块，找到leader

②想leader发送笑死

③leader收到消息并写入本地log

④follower从leader pull消息

⑤follower想本地写log

⑥follower向leader发送acks消息

⑦leader收到所有follower的ack消息

⑧leader向producer回传ack

异步复制：

.......leader写完本地log之后直接向producer恢复ack消息

1. 请问kakfa客户端缓冲机制你了解吗，kafka是如何优化这个缓冲池的？它是用来解决什么问题的？你还能列举类似的应用吗？

其实缓冲池大家并不陌生，随处可见，操作系统中更是常用，比如缓冲、缓存，socket的输入缓冲区、输出缓冲区，输入流输出流管道，sparkflink都有record的缓冲区，kafka的客户端发送数据同样也要缓冲池，namenode记录edits log时使用双缓冲机制和异步持久化机制，不是一条一条发送，而是先写入缓冲池——batch，然后一个batch一个batch的发送，那么现在就有一个问题——————GC问题，message是很多的，发送完就面临GC，那么频繁GC肯定会导致stop theworld，阻塞进程。

Kafka是如何做的呢？

其实它的做法很类似线程池、python的内存管理，当batch这块内存空间不需要的时候不会让垃圾收集器的算法标记可回收而是将其放入缓冲池中！比如给内存池32M，每个batch16kb，那么当有新数据进来时直接从内存池中拿16kb的batch即可而不用在申请内存空间了。

1. kafka是如何实现消息队列和订阅模式的？

看到这个问题，实际上也要回答一下其他消息队列如rabbitmq是如何实现的

①kafka是通过consumer group来实现的，比如消息队列的实现就是一个consumer group，同一个消费组不能重复消费一个patition中的数据，这就保证了一个消费组中所有consumer只消费一次数据，不重复；而订阅的方式实现肯定就是不同的conumer group了。

②rabbitmq则不然，它通过exchange的规则的方式将数据写入不同的queue，实现复杂路由，比如把相同数据写入多个queue，比如副歌一定规则的数据写入制定queue，这中间关键点在于routing-key和binding-key的匹配！，当然也可以直接指定queue！而consumer消费的时候并不是消费组的概念而是queue的概念，不管有多少consumer，只需要指定queue即可，如果是队列模式，那么只需要消费一个queue即可，如果是订阅模式那么多个consumer对应多个queue即可，由exchange来实现数据的copy。

另外说一下：kafka注重吞吐量，而rabbitmq注重复杂路由机制，kakfa为大数据而生，rabbitmq则是为复杂逻辑而生。

1. 既然你说有partition副本的机制保证高可用，那么你知道kafka是如何向broker上分配partitionid的吗？

注：kakfa集群中，每个broker都有均等的机会分配partition的leader机会

①将所有N歌broker和待分配i partition排序

②将第i个partition分配到i mod n的broker上

③将第i个partition的第j个副本分配到(i+j) mod n的broker上

所以partition的副本分配看上就像是一个等概论循环分配。

注意：这是理想的情况哦，也就是当topic刚刚创建时是这样分配的，但是当某台broker宕机之后这台上的leader全部都会变更，而在此恢复之后，就全部编程follower了，这样的话就不均衡了。

解决不均衡方法：

可以配置自动触发，也可以手动触发

kafka-preferred-replica-election.sh用来均衡leader

具体实现：

当最初创建topic的时候在ISR列表中的第一个副本编号就是leader，当broker宕机，leader变更之后，这个编号是不变的！！！这就是关键，所以当运行这个脚本的时候会让controller把这个编号的副本提升为leader，从而实现均衡操作。

1. 你知道kafka的持久化机制的吗？顺序写的优化你是怎么理解的，它是怎么实现的？

Kafka是基于磁盘而不是内存来存储信息的，这种方式乍一看感觉效率肯定低啊，但是实则不然，要知道顺序磁盘写的效率比随机内存写的效率要高。

①磁盘便宜，内存贵

②顺序读+预读取操作，提高缓存命中

③配合操作系统的缓存cache，这个缓存有预读取和回写计数，消费从cache读，写到cache就返回，操作系统自动flush

④内存占用太大

⑤基于文件的顺序读写

⑥持久化数据结构上选择了queue而不是btree，因为kafka只支持根据offset读取和append操作，所以基于queue的操作时O(1)，而btree时O(logn)的

⑦在大量文件读写时，基于queue的read和append只需要一次磁盘寻址，而Btree会设计多次，磁盘寻址大大降低了读写性能，因为Btree的数据存储中是根据索引确定数据存储。

1. 问了你那么多kafka的基础知识，我想问问你到底这些东西是怎么跟zk交互的？或者直接一点请问controller manager你知道吗？

Controller的目的是防止broker集群对zk的大量操作，通过这个中间件可以简化操作

功能：

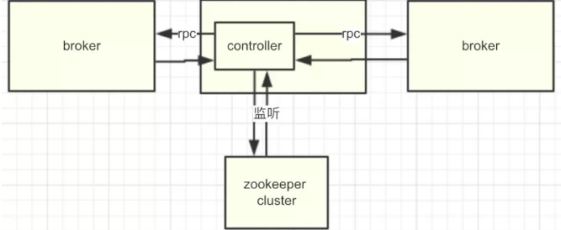
①增加删除topic

②更新分区副本

③leader选举

④集群broker增加和宕机的调整

⑤自身的选择controller leader的功能



先看一下存储在zk上的信息吧：

brokers列表：ls /brokers/ids

某个broker信息：get /brokers/ids/0

topic信息：get /brokers/topics/kafka10-topic-20170924

partition信息：get /brokers/topics/kafka10-topic-20170924/partitions/0/state

controller中心节点变更次数：get /controller\_epoch

conrtoller leader信息：get /controller

内部设计：

当controller启动时会跟集群中所有的broker建立socket连接

1. kafka的负载均衡体现在哪些方面？

①partition的分配

刚才说过了，根据分配策略，以及leader的分配和重分配

②producer的message进入分区时的负载均衡

进入策略首先：如果指定分区则进入指定分区，其次：如果没指定分区，但是有key，那么会根据key的hash进入分区，最后：如果key为空那么久轮训。

③consumer消费partition

第一种情况：如果分区大于或者等于消费者数，那么很简单无非就是一个消费者负责多个分区而已

第二种情况：如果分区数大于消费者数怎么办？默认是一对一消费，有些消费者是空闲的！！

那么如果出现多个consumer负责同一个分区会出现什么情况？就类似与多个线程操作一个对象一样，可能出现由于线程进度的问题导致消费顺序的问题，这也就是为什么kafka采用pull的方式让consumer自己拉数据而不是push，目的就是根据consumer的能力自己来消费方式压力太大或者消费顺序打乱问题。

针对第一种情况，如果分区数大于消费者，那么就有一个负载均衡的问题，到底分区怎么分配给少量的consumers？

策略一：range分配策略

简单说就是分区按数字排序，消费者按字典排序，然后基于每个topic的分区对consumer数量进行整除，不整除的话前面的消费者多分配一个分区，比如topicA的三个分区p0p1p2和topicB的p0p1p2，有两个consumer：C1和C2，那么分配则是C1：topicAp0 topicAp1 topicBp0 topicBp1，而C2：topicAp2 topicBp2，不要以为是取余！！

策略二：roundrobin轮训

这种方式就是取余的均匀轮训分配，比如上面的例子C1：topicpA0 topicAp2 topicBp1 C2:topicAp1 topicBp0 topicBp2，这种方式不局限于topic同时采用更均匀方式，当然条件是消费者们都是订阅同一个主题，如果出现订阅不相同，其实道理一样，这不过就出现不均匀的情况了。

1. 很好，最后你说一下kafka与rabbitmq的区别吧？

|  |  |  |
| --- | --- | --- |
|  | Kafka | Rabbitmq |
| 语言 | Java+scala | Erlang |
| 消息协议 | 自定义通信协议 | AMQP/MQTT/STOMP |
| 消息过滤 | Topic+partition | 交换机路由 |
| 消息堆积 | 磁盘 | 内存，也就是证明了前段时间为什么mq总蹦的问题 |
| 消息传递模式 | Pub/sub | 典型的p2p |
| 消费模式 | Pull | Push+pull |
| 消息回溯 | Offset+timestamp（文件命名） | 消费即删除 |
| 流量控制 | 对producer和consumer的设置 | Credit-based算法作用于producer |
| 消息顺序 | 支持但分区的顺序 | 单线程的发送和消费，很顺序 |
| QPS | 单机维持数十万，甚至百万 | 单机万级别 |

## Zookeeper部分

1. 请问尽可能说一说你对zookeeper的了解？

简单说：是一个典型的基于Pub/Sub模式的分布式管理和协调系统

可以用作：数据的发布/订阅、复杂均衡、统一命名空间、集群管理、master选举、分布式锁和分布式队列等功能。

我们都知道分布式系统要保持CAP定理，关于CAP中C和A的取舍可以自行百度。Zk在cap基础上添加了顺序一致性，也就是有序性是zk的一大特性，所有的更新都是全局有序的，都有唯一的时间戳——zxid，等一下会介绍zxid。

Zk提供四种类型的数据节点znode（两大类）：

①持久节点persistent

除非手动删除，否则一直存在

②临时节点ephemeral

临时节点的生命周期与客户端回话绑定，一旦客户端会话失效（注意断开连接不是失效），那么这个客户端创建的所有临时节点都会被移除

③持久顺序节点persistent\_sequental

基本特定同持久节点，知识增加的顺序属性，节点名后面会追加一个由父节点维护的自增整形数字

④临时顺序节点ephemeral\_sequental

基本特性同临时节点，增加了顺序属性，节点后面追加一个由父节点维护的自增整形数字

Zookeeper实现很多功能如分布式锁、统一配置、统一命名空间、集群管理都是基于持久节点和临时节点完成的！

服务器角色：

①leader

事务请求的唯一调度者，保证集群事务处理的顺序性，集群内部个服务的协调

②follower

处理客户端的非事务请求，转发事务请求给leader，参与事务请求proposal的投票，参与leader的选举

③observer

3.3.0版本引入的角色，在不影响集群事务处理能力的基础上提升非事务处理能力，负责处理非事务请求，转发事务请求给leader，不参数任何形式的投票

1. 调用zkServer.sh status后都有哪些工作状态？

①leadering

领导者状态

②follower

跟随者状态

③looking

寻找leader状态，表示没有leader需要进行选举

④observering

观察者状态，表明当前服务器是observer

1. zk的选举机制流程？选举算法？

说zk选举之前先说一下问什么选举或者什么状态下选举，选举的条件是什么：

两种情况下进行leader选举：

①服务器初始化启动

②服务器运行期间无法和leader保持连接

细说两种选举流程之前先说一下zxid吧，因为选举中要用到这个zxid，zxid是全局有序的，zxid有三种：

①cZxid：节点创建的全局id

②mZxid：节点修改的全局id（仅限于本节点）

③pZxid：本节点和本节点的子阶段的增/删（跟孙节点无关）

所以我们可以认为，理想情况下所有几点上的相同节点的zxid是完全一致的。如果发现两次zxid不一致，那么肯定说明了zxid小的慢与zxid的那次操作！

zookeeper使用zxid来保证事务的全局顺序一致性：

当有事务提交时，会把这个事务提交到proposal的事务提交缓存队列中，每个事务都会加上一个zxid，zxid是一个64位的数字，高32位是epoch——表示leader的迭代次数，每次选举都会自增（它和electionepoch还不一样，electionepoch是投票迭代次数），低32位用来递增计数的。所以这就保证了全局事务的顺序一致性。

第一种情况：初始化时选举

进行选举至少要两台服务器，当启动server1时，只有一台服务选举，当启动server2时。两台可以互相通信，都试图找到leader，于是进行leader选举的过程：

①发起投票：每个server发出一个投票(myid, zxid)——server1的(1, 0)和server2的(2,0)，将投票发送给其他服务器，zk集群中，两两之间都会建立一个连接，用于消息的发送接收，监听3888

②接收投票：接收到其他服务器的投票，需要验证投票的有效性，检查electionEpoch轮次，判断是不是本轮投票，这个electionepoch是自增的，每次投票都+1，是否处于选举中的locking状态，如果接受到的选票的electionepoch大于自己的轮次，那么就更新自己轮次，重新广播自己的投票，如果小于自己的轮次，则不做任何处理，如果相等的话就进行下一步...

③处理投票：先看zxid，大的作为leader（上面说了，zxid是全局有序，如果出现zxid不一致，那么说明zxid大的是最新的操作，所以把zxid大的那次投票的结果作为），如果zxid相同则myid大的作为leader，由于zookeeper是少数服从多数的机制，所以必须要过半的支持率才可以，所以这就是为什么zk是奇数个。此时(1, 0) (2, 0)zxid相同，那么server2应该赢了，此时server1要更新自己的投票，改投(2, 0)，server2不变，然后再次投票

④统计投票：根据过半准则：少数服务从多数

⑤改变服务器状态，一旦确定leader，每个服务器都改变状态，如leadering和followering

第二种情况：服务器运行期间选举

运行期间，follower宕机不会影响服务，但是leader宕机，那么服务停止必须进行leader选举，过程其实和初始化时选举差不多，假设server1 server2 server3，此时作为leader的server3挂了。

①变更状态：集群中follower的状态从followering全部变为locking状态

②发起投票：每个server都发出一个投票，如上

③接收投票：如上

④处理投票：如上

⑤统计投票

⑥改变服务器状态

FastLeaderElection：zk选举算法：

其实该算法的核心就在与变更投票的过程！

变更投票上leader选举的第三步左右进行，每台服务器自己的投票与接收到的其他服务器发送的投票进行对比，然后判断是否要变更投票的过程！是否变更刚才已经说过了。

有个问题就是，zxid不是全局有序并一致的吗？为什么可能出现zxid不一致的情况？

答：zxid肯定是全局一致，并且每台服务器上的某个节点的所有zxid一致，但是投票过程中的zxid并不是投票行为的zxid而是当前节点所正进行事务zxid，投票所携带的zxid如果大，说明正在执行的事务是最新的，这样的话让他作为leader，那么数据恢复不就是最新的嘛！，所以不要把投票携带的zxid看成是操作的zxid！

1. zk的数据同步？请问zk如何保证的主从节点的状态同步？

概念：集群leader确认之后，follower和observer会向leader注册自己，注册完毕后就需要进行数据同步，最后由leader确定同步完成。

zookeeper的数据同步通常分为四类：

①直接差异化同步

②先回滚再差异化同步

③仅回滚同步

④全量同步

他们各自使用于不同的情况，其实说白了就是zxid事务id的区别，目的是保证数据的一致性！介绍这四种同步方式应用情况之前先说一下follower和leader之前的注册和信息交换：

在进行数据同步之前，leader服务器会完成数据同步的初始化：

peerLastZxid：follower服务器最后完成的事务id：zxid，follower向leader注册时发送ackepoch消息中携带的

minCommittedLog：leader上proposal——事务缓存队列中已经提交的事务中最小的Zxid，也就是代表leader能保存的最早的信息了

maxCommittedLog：leader上proposal缓存队列中已提交事务的最大Zxid，代表着最新的信息

①直接差异化同步

场景：当follower上报的peerlastzxid介于minCXommittedLog和MaxCommitedLog时，也就是说执行一下maxCommittedLog - peerLastZxid之间的事务就ok

②先回滚再差异化同步

场景：发现follower上有一条事务信息leader上没有，那么此时需要让follower回滚至跟leader相同，然后再补上新的数据即可。目的是剔除那条follower上有单leader上没有的事务。

注意，这种情况和仅回滚同步是不一样的

③仅回滚同步

场景：场景：peerLastZxid大于maxCommittedLog，也就是leader发现follower比自己还新，那么只需要回滚就OK了。

④全量同步

场景1：peerLastZxid 小于 minCommittedLog，因为再怎么恢复也不行了，只能全量同步

场景2：leader服务器上没有proposal缓存队列并且peerLastZxid不等于lastProcessZxid

关于第二个问题，实际上如果只回答数据同步还是不够严谨的，因为数据同步发生在恢复模式，而恢复模式是leader选举过程中的模式，zk还有另外一个中模式是广播模式，正常情况下zk试运行在广播模式下的！

Zookeeper的核心是原子广播模式，这个机制保证各个server之间的同步。实现这个机制的协议叫做ZAB协议。

当整个zookeeper集群刚刚启动或者Leader服务器宕机、重启或者网络故障导致不存在过半的服务器与Leader服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的Leader服务器，然后集群中Follower服务器开始与新的Leader服务器进行数据同步，当集群中超过半数机器与该Leader服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

1. zookeeper中watcher机制是什么，都有哪些特点呢，用它可以来干嘛，为什么watcher不能是永久的？

聊watcher的话那就说明已经不是单纯的zk集群了，而是涉及zkClient客户端与zk服务端的操作了。（当然了集群中的节点也可能会监听某个znode）

Zookeeper Watcher机制————数据变更分布式通知！

解释：zk允许client向服务端的某个znode注册一个watcher监听，当这个znode发生一些指定的事件触发这个watcher时，zk就会向client发送一个时间通知来实现分布式通知功能，然后客户端根据watcher童稚状态和事件类型做出相应的改变。

①客户端注册watcher

②服务端处理watcher

③客户端回调watcher

watcher特点总结：

①一次性

无论是客户端还是服务端，一旦一个watcher触发，zk就会将其移除，目的是减轻服务端压力，假如更新频繁的znode，难道每次都要想client通知吗，这样显而易见是不行的，这也就解释了为什么watcher不能是永久的

②客户端串行执行

客户端watcher回调的过程是一个串行同步的过程

③轻量级

zk的watcher通知很简单，只会告诉cleint发生的时间而不是事件具体内容，客户端注册watcher时，zk并不会吧这个watcher对象保存下来，而是在客户端请求中使用boolean类型属性进行了标记

④watcher最终一致性，不保证强一致性

event异步发送watcher的通知事件，也就是server向client发送通知是异步的，这就有一个问题就是可能存在通知失败的情况zk却不知道，而client只有收到通知事件之后才能感知到znode的变化，所以只能保证最终一致性，不能保证强一致性

⑤注册watcher getData、exists、getChildren

⑥触发watcher create、dalete、setData

⑦重新注册

通常情况下：client注册完watcher后断开连接，后来重新连接时会重新注册先前注册过的watcher，所以如果在断开期间发现znode的数据变化、子节点变化、znode的存在与否都会被通知。

但是存在一种情况——exists watcher，当断开client与server之间连接的期间，有一个znode创建了又被删除了，然后client重新连接server再注册watcher时并没有通知！

下面说一下watcher机制的实现吧：

客户端watcher实现：

①调用getData()/getChildren()/exists()三个API，传入watcher对象

②标记请求request，封装watcher到warcherRegistion

③封装Packet对象，向服务端发送request

④收到服务端响应后，将watcher注册到ZKWarcherManager中进行管理

⑤请求返回，完成注册

.......

⑥....如果client的SendThread线程接收到了server事件通知，那么交给EventThread线程回调Watcher

服务端处理watcher实现：

①服务端接收watcher并存储，接收client请求，判断是否需要注册watcher，需要的话将数据节点的节点路径和ServerCnxn(ServerCnxn代表一个cleint与server的连接，实现了Watcher的process接口，可以看成一个watcher对象）储存在watcherManager的WatcherTable的watch2paths中

②Watcher触发，比如当server收到某个客户端的setData()操作，它会触发NodeDataChanged事件，zk把这个事件封装为WatcherEvent，包括童稚状态、事件类型、路径znode；然后从watcherTable中查找是否有这个znode的监听watcher，没找到的话无所谓，如果找到了就从WatcherTable和Watch2Paths中删除这个watcher

③调用process方法触发watcher，主要是通过ServerCnxn对应的TCP连接发送Watcher事件通知。

1. 请列举zookeeper的几个典型的应用场景？

zk说白了就是利用四种（两类）znode+watcher完成各种功能：

①数据pub/sub——统一配置

把数据存在znode上，当启动服务时向zk读取配置，并且注册一个watcher，当发生改变时就动态改变配置

②统一命名空间

命名服务就是通过制定的名字来获取资源或者服务的地址，利用zk创建一个全局的路径，这个路径就可以作为一个名字指向集群、提供服务的地址

④分布式协调/通知

⑤集群管理

无外乎两点：是否有机器的加入和退出、master选举

实现起来很简单，用临时节点即可，一旦有机器加入就在指定目录创建临时节点，并且对该节点的父节点进行监听，所以有加入或者退出所有节点都会受到通知，master选举也很容易，通过临时节点+编号（时间戳即可）选取最小的作为master，备用节点监听这个临时节点，如果master宕机，那么所有备用节点就是竞争（重新注册临时节点），选取编号最小的作为master

⑦分布式锁

独占锁、共享锁

独占锁：实现是竞争创建持久节点，床架成功的就获取锁，用完删除

共享锁：仍然是跟master选举一直，都去指定目录下注册带有顺序的临时节点，最好带上自己的标识（目的是可重入性），按照编号从小到大依次获取锁和释放锁

⑧分布式队列

同步队列：用watcher监听某目录下子节点数量，当达到要求时进行消费

FIFO：与分布式锁、master选举基本一样，创建顺序的节点，按照编号消费，消费完的删除

## flume部分

1. 请问flume常用的source、channel、sink举几个例子？

source:

用于接收event并且将这个event放入一个或者多个channel缓存，这个写入过程是事务的！

①avro source

支持avro协议，实际上是avro rpc，其实就是传输协议啦，最常用于级联flume的agent端的sink和server端的source

②thrift source

支持thrift协议，实际上是thrift rpc，也是用于级联的....关于avro 和 thrift这两种source都可以用作sink，作为source时是将avro event和thrift event转换为正常的数据，作为sink的时候就是将数据转换为avro event和thrift event。

③exec source

unix系统中的command命令的标准输出作为source数据

④JMS source

从JMS系统中读取数据

⑤spooling dictionary

监控指定目录内数据变更

⑥netcat source

监听某个端口

⑦http source

基于http 的get或者post的数据源

⑧taildir source

监听一批文件的实时动态，那么exec 的tail -f(F)、spooling、taildir他们三个怎么区别，实际上前两个已经不用了，因为taildir是他俩的优化版，比如exec的tail -f，如果agent挂了再启动，那么可能存在数据丢失或者数据重复消费，spooling dir是监控一个目录下的所有文件，但是只支持新增文件不支持文件的动态追加，而taildir则可以监控一批文件的动态追加，可以记录每个文件的offset保证不会重复和丢失数据。

还有其他如kafka source、自定义source等

channel：

channel是线程安全的，可以同时被多个source写入和多个sink读取

①memory channel

event存入内存队列中，这种情况是不考虑部分数据丢失的情况下用的

②file channel

将所有的event写到磁盘，不会丢失

③jdbc channel

event数据存储在持久化数据库中，flume自带Derby

④spillable memory channel

event存储在内存和磁盘上，当内存队列满了会持久化到磁盘

⑤kafka channel

效率略低哦

sink：

不断轮询channel中的event并批量的移除它们，将这些时间批量吸入到存储或索引李彤或者发送到级联的flume，这个从channel到sink结束过程是事务的，所以在真正删除之前需要事务提交成功。

①HDFS sink

直接写入hdfs

②hive sink

直接写入表

③logger sink

④avro sink thrift sink

将数据封装成avro thrift event发送到指定RPC端口

⑤IRC sink

数据在IRC上进行回放

⑥file roll sink

存放在本地文件系统

⑦hbase sink

写入hbase

⑧solr sink es sink

发送到solr集群或者es集群

⑨kafka sink http sink等

1. flume是如何保证数据不会丢失的？

是通过两个事务实现的——put事务和take事务。

①put事务

doput：将批量数据吸入临时缓冲区putlist

docommit：检查channel内存队列是否足够合并

dorollback：channel内存队列空间不足，回滚数据

②take事务：

dotake：将数据读取到临时缓冲区takelist

docommit：如果数据全部发送成功，则清除临时缓冲区takelist

dorollback：数据发送过程中如果出现异常rollback将临时缓冲区的takelist中的数据归还给channel内存队列

1. channel Interceptors、 channel selector是什么东西？什么作用？

①channel interceptor拦截器

目的是对event进行处理！！！注意不要狭义的理解为过滤，过滤仅仅是一个功能，它还可以对event中的header中的key进行操作。

1. timestamp interceptor：在event的header中添加一个key叫做timestamp，value就是当前时间戳，很重要哦
2. host interceptor：在event的header中添加一个key叫做host：value，value就是histname合作而ip
3. static interceptor：可以再在evetn的header中自定义key：value
4. regex filter interceptor：这个才是过滤功能，通过震泽清洗或包含匹配的event
5. regex extrctor interceptor：通过正则表达式来在header中添加指定的key,value则为正则匹配的部分

②channel selector

首先明确一点，不管是source还是channel还是sink都可以是多个！多个source可以指定一个channel，这种情况下所有数据没有任何分别，一个source可以指定多个channel，那么这种情况下就涉及到channel selector的问题了，到底是把所有event发往所有channel还是分别发往呢（说白了就是备不备份）所以涉及channel selector的情况只有当一个source对应多个channel的时候！

1. replicating channel selector：只要source指定的channel，所有该source中的event全部同样发送，类似于备份
2. mutiplexing channel selector：根据一定的规则（event的header中的key：value）来判断该event应该发往哪个channel

如果要理解拦截器和channel选择器的话需要理解flume agent的工作流程

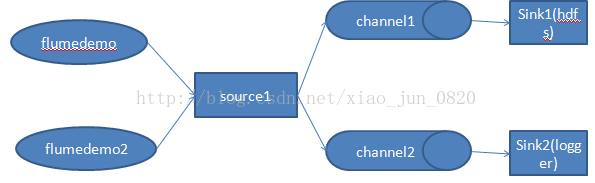
①一个或多个source将数据封装为event发往channel

②channel处理器首先会将event传入拦截器进行过滤，然后将每个事件传给channel selector，channel 拦截器的作用也挺重要的不能忽略，可以chain链式的添加多个拦截器，内置的有添加时间戳拦截器，添加hostname拦截器，添加自定义keyvalue拦截器，过滤拦截器等

③channel selector会根据配置并根据header中keyvalue判断返回需要写入的channel的列表，这样channel就知道把event写入哪个或者哪些channel里面了

④sink 处理器就比较简单了，根据每个sink的配置去安全的读取（channel是线程安全的）对应的channel数据并sink到指定位置

可以看出channel selector就是判断event写入哪个channel或者哪些个channel是根据配置来的，selectort分为：replicating channel和multiplexing channel，其中replicating channel是source的所有event都全量发往不同的channel，而multiplexing channel selector则是根据规则发送到指定的channel，只有一份。其实这样说不够完美，下面介绍一个实际例子：



目的是demo1个数据发往hdfs，demo2的数据发往logger，两种情况：

首先明确一点，channel selector是根据event中header的key值来判断发往哪个channel的！！

①demo1和demo2在不同的服务器上跑

现在我们认为这个flume agent是二级级联的，因为是一个source嘛。此时由于demo1和demo2是两个服务器上的，可以在第一层flume的source中添加host interceptor，将hostname添加到header中，然后指定channel的selector是multiplexing的，然后根据header的mapping值指定channel即可，这就达到了区分两类数据的目的

②demo1和demo2在同一台服务器上跑

其实说实话解决办法还是蛮多的，比如

1. fileHeaderKey这个东西可以在header中加入文件名，这样的话你懂的，当然了条件是文件名不一致，也就是数据不混在一起
2. 在写日志的时候带上自己的标识（当然这种方式不是特别好），有了标识时候就可以根据正则匹配创建header中的key vlaue，然后根据selector进行选择channel
3. 也可以自定义source