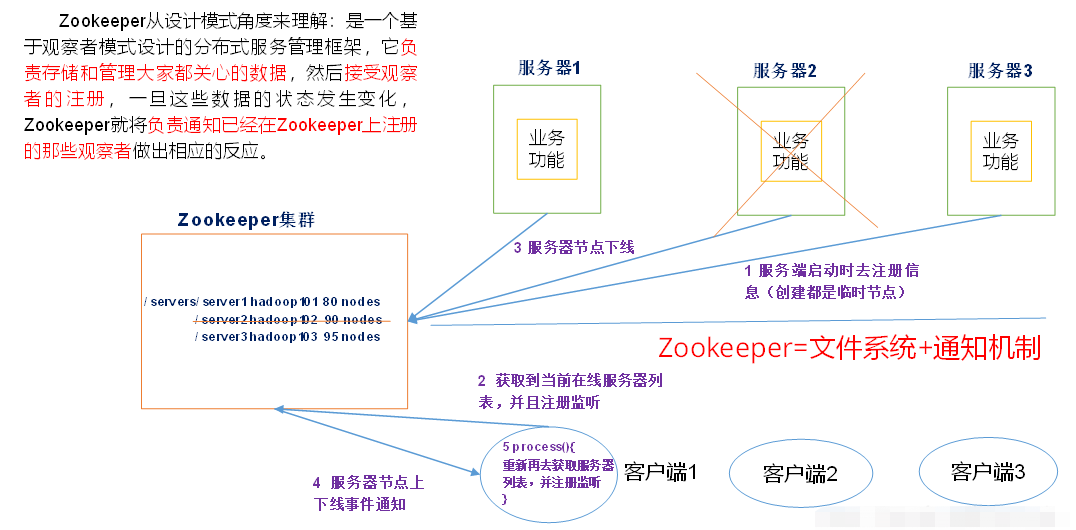
**大数据框架学习笔记**

# Zookeeper学习笔记

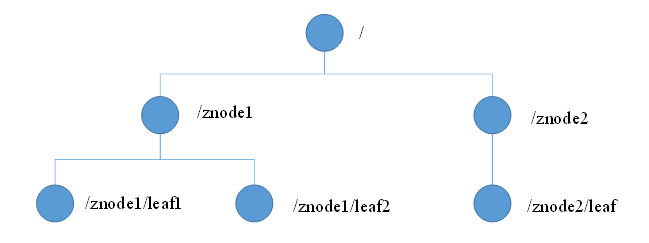
zookeeper的端口号：**2181**，客户端访问的时候使用的端口；

## Zookeeper的工作机制



Zookeeper集群中有一个Leader，多个follower；集群中只有要半数以上的节点存活，zookeeper就能提供服务；全局的数据是一致的，其数据结构和类Unix的文件系统很类似，每个路径下面可以存储数据，默认存储的大小是1M：

‘



## 小知识点：

Quorum：是仲裁的意思，启动zookeeper之后会有一个QuorumPeerMain进程。

Zookeeper的配置文件zoo.conf 中的两个选项：

dataDir，是数据文件的目录+树持久化路径，主要有用于保存zookeeper中的数据。

clientPort：服务器监听客户端的端口号。

集群模式下，zoo.conf 中的

server.2=hadoop102:2888:3888server.3=hadoop103:2888:3888server.4=hadoop104:2888:3888

server.2 是表示这是第几号服务器，2就是myid的值；hadoop101就是zookeeper服务器的地址；2888就是Leader和Follower信息交换的端口；3888 是在Leader挂了之后，用来选举的端口。

创建节点，并添加值：create /isea/gril “beauty”

获取节点的值：get /isea/gril

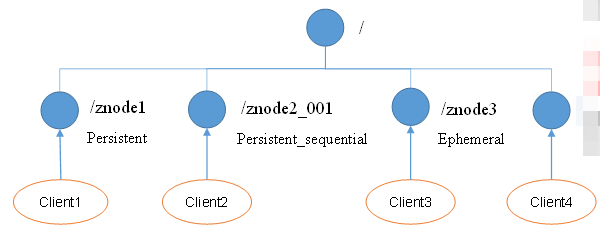
初此之外，还有ls delete

## Zookeeper的内部原理

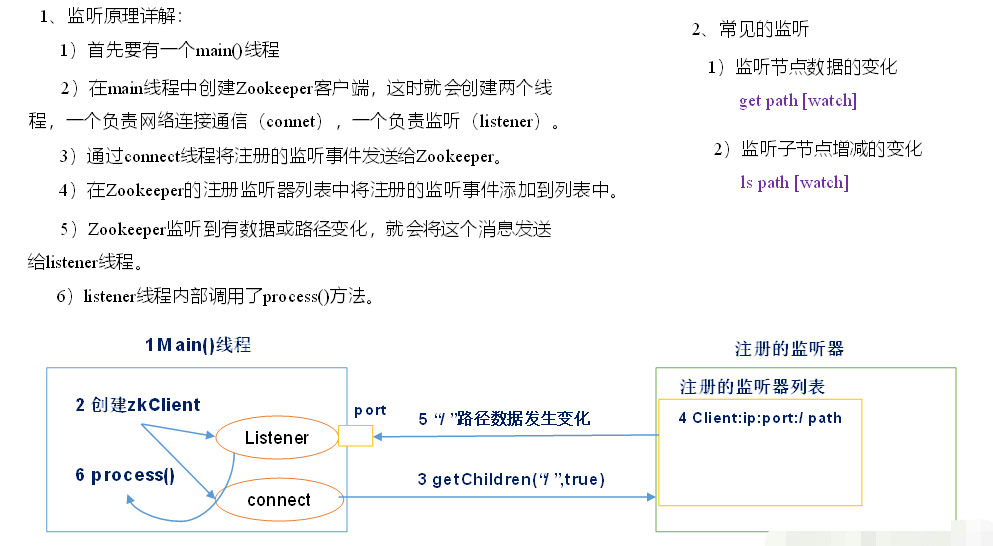
### 节点类型

持久（Persistent）客户端和服务器建立连接值断开后，创建的节点不删除

短暂（Ephemeral）客户端和服务器断开连接之后，创建的节点自己删除



### 监听器的原理：



这里的zookeeper的client是观察者；

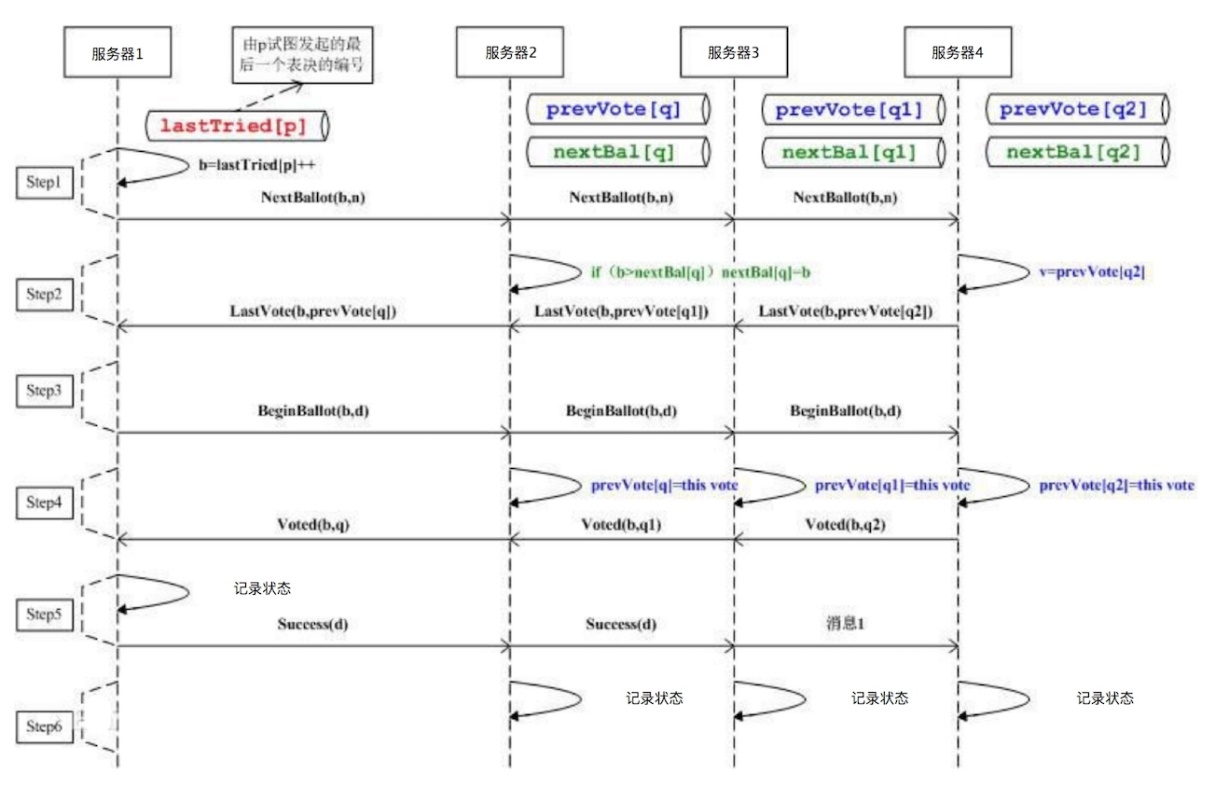
### Paxos保证数据一致性

分布式系统中一类非常常见的问题就是数据一致性的问题，比如在实现NameNode的高可用的时候，需要设置两台namenode，那么哪一台作为主，哪一台作为从呢？（如果判断的不同就会发生脑裂）所以需要一个“裁判”，通常在分布式的数据一致性方面，由zookeeper来当这个裁判。

同时zookeeper作为决策的，也面临着数据一致性的问题，和高可用的问题，所以zookeeper往往也会组成一个集群。

zookeeper可以说是一个提供锁服务的分布式系统，它由多台服务器构成一个集群对外提供锁服务，应用程序连接到任意一台服务器都可以获取或者是释放锁，因此这些服务器必须保持严格的状态一致，不能出现一台服务器将锁交给一台服务器，另外一台服务器将锁交给另外一台服务器。

paxos算法就是用来解决上述的问题（保证锁只交给一个应用）该算法的思路如下：

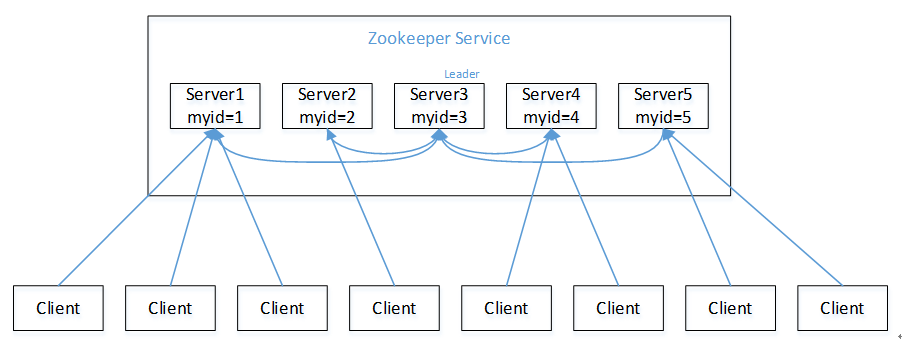
，

应用程序连接到任意一台服务器之后提起状态修改的请求（也可以是获得某个状态锁的请求）该服务器（也就是上图的服务器1）会将这个请求发送给集群中的其他服务器进行表决，如果其他的服务器也收到了状态修改的请求，他可能会拒绝服务器1的请求，并且自己也发起一个同样的表决请求，那么其他的服务器就会根绝服务器的排序顺序和时间戳进行表决

zookeeper中使用paxos算法的简化版；ZAB协议（原子消息广播协议）来保证zookeeper中数据的一致性。

### 选举机制

1. 半数机制，集群中半数以上的机器存活，集群可用，所以zookeeper适合安装奇数台机器
2. Zookeeper在配置文件中没有指定Master和Slave，zookeeper中的Leader是通过内部选举机制完成的。
3. 选举过程：



（1）服务器1启动，发起一次选举。服务器1投自己一票。此时服务器1票数一票，不够半数以上（3票），选举无法完成，服务器1状态保持为LOOKING；

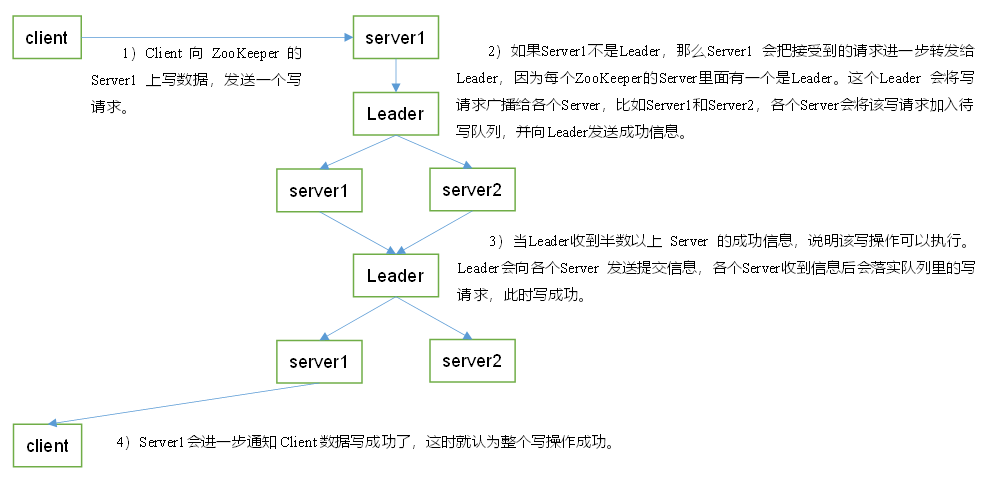
（2）服务器2启动，再发起一次选举。服务器1和2分别投自己一票并交换选票信息：此时服务器1发现服务器2的ID比自己目前投票推举的（服务器1）大，更改选票为推举服务器2。此时服务器1票数0票，服务器2票数2票，没有半数以上结果，选举无法完成，服务器1，2状态保持LOOKING

（3）服务器3启动，发起一次选举。此时服务器1和2都会更改选票为服务器3。此次投票结果：服务器1为0票，服务器2为0票，服务器3为3票。此时服务器3的票数已经超过半数，服务器3当选Leader。服务器1，2更改状态为FOLLOWING，服务器3更改状态为LEADING；

（4）服务器4启动，发起一次选举。此时服务器1，2，3已经不是LOOKING状态，不会更改选票信息。交换选票信息结果：服务器3为3票，服务器4为1票。此时服务器4服从多数，更改选票信息为服务器3，并更改状态为FOLLOWING；

（5）服务器5启动，同4一样当小弟。

### 写数据流程

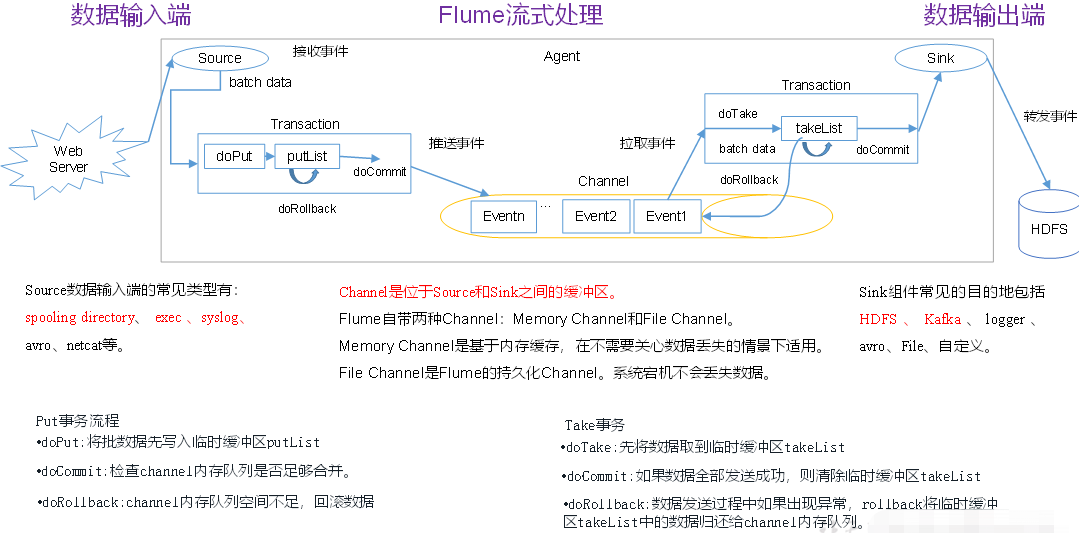


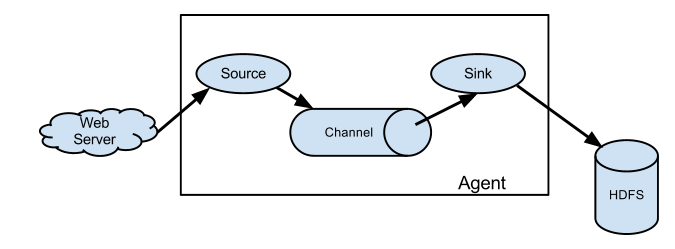
# Flume

Flume是分布式海量日志采集，聚合，传输系统。在大数据中最为广泛的应用是实时读取服务器本地磁盘的数据，为下一步传输做准备。

Flume使用两个独立的事务，分别负责source到channel，和从channel到sink的事件传递。一旦事务中的所有数据全部成功提交到channel，那么source才认为数据读取完毕，只有成功被sink写出去的数据，才会从channel中移除。

## Flume组成架构





Agent：是一个JVM进程，按照事件的形式将数据从源头送至目的地。是Flume传输数据的基本单元，主要有三部分组成：Source，Channel，Sink。

Source：负责接收数据到Flume Agent的组件。

Source类型：exec，实时读取本地文件，spooldir，实时读取目录文件

Avro：连接上一个Flume Agent的Avro Source，netcat 监听端口。

Channel：位于Source和Sink之间的缓冲区。

Sink：不断的轮询Channel中的时间，且批量的移除他们，并将这些事件批量写入存储或发送到另外一个Flume。

Sink类型：HDFS写到HDFS上，Avro ：连接下一个Flume的Agent的Avro。

Event：Flumet数据**传输单元**，有header和body两个部分。

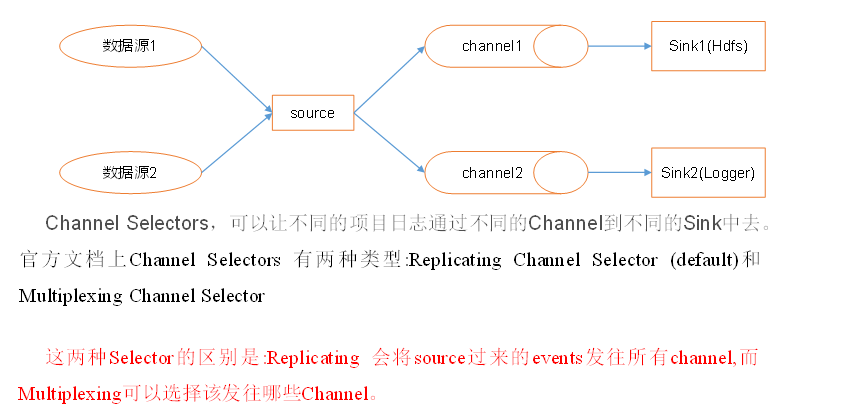
Flume中channel的数据是完整的数据，如果一个channel对应了多个sink，那么每个sink只是channel中的部分数据，但是分流的时候往往不这么做，通常的做法是会使用一个拦截器，将event拦截下来，然后打上特定的标签，然后再使用channel选择器将分往不同的channel，在进入channel之前就完成数据的分流，而不是使用sink。

玩Flume玩的是就是配置：

如何启动一个Flume进程：

|  |
| --- |
| bin/flume-ng agent -c conf/ -n a1 –f job/flume-netcat-logger.conf -Dflume.root.logger=INFO,console  对应的各个参数的信息：  --conf conf/ ：表示配置文件存储在conf/目录  --name a1 ：表示给agent起名为a1  --conf-file job/flume-netcat.conf ：flume本次启动读取的配置文件是在job文件夹下的flume-telnet.conf文件。  -Dflume.root.logger==INFO,console ：-D表示flume运行时动态修改flume.root.logger参数属性值，并将控制台日志打印级别设置为INFO级别。日志级别包括:log、info、warn、error。 |

Flume的channel selector选择器

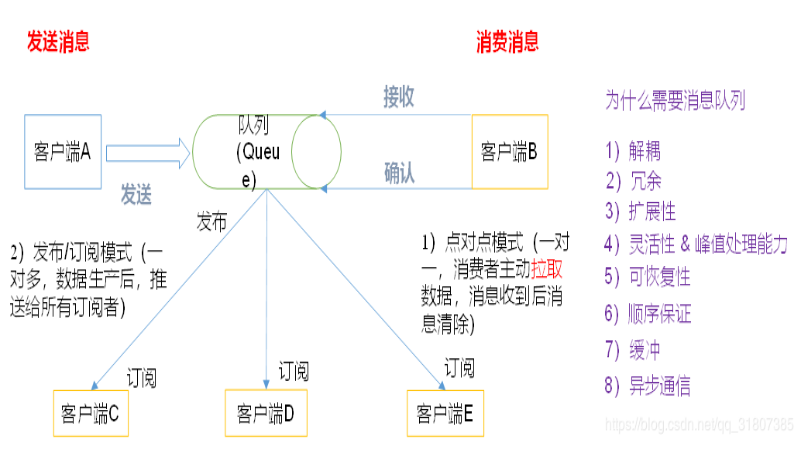


# Kafka

Kafka是一个分布式消息队列。

Kafka是用scala写的。

## 消息队列：



消息队列的好处，实现了生产者和消费者的解耦，生产者生产了数据，可以将消息放在消息队列中，然后自己继续生产，消费者消费与否，并影响生产。

## 相较于Flume，Kafka的优点

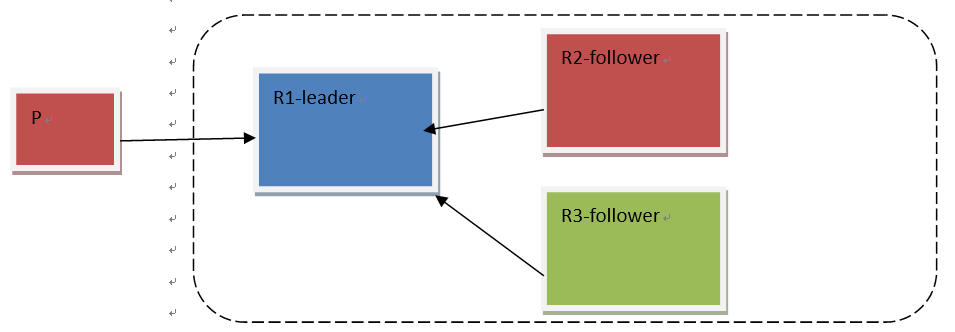
如果使用Flume进行数据的传输，无法较长时间的保存数据，但是Kafka可以较长时间的保存数据，这样一来消费者就可以衡量自己的消费能力消费。

Kafka增加消费者非常的方便，如果是Flume增加消费者，就得增加一个sink，由于sink只能获取channel中的部分数据，所以还需要增加一个channel，耦合度太高。

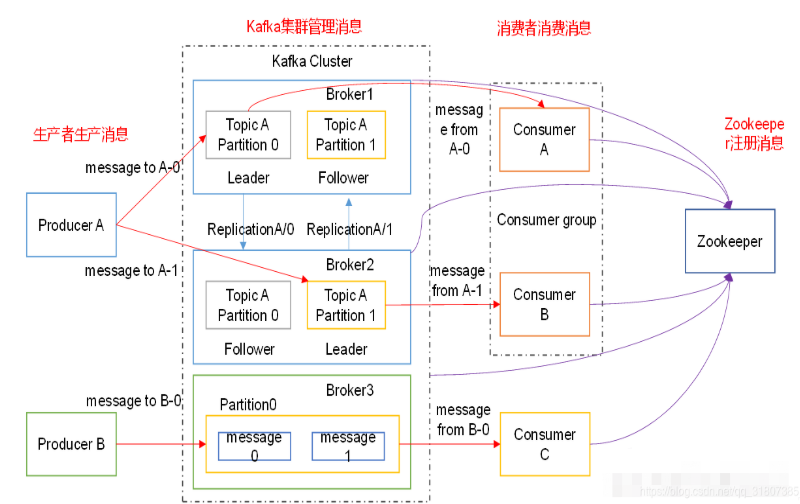
## Kafka与zookeeper

Kafka的server.properties配置文件中写入了zookeeper集群的地址信息。

0.9版本之后，在zookeeper中只是存放着主题，分区，leader，副本，kafka集群的等信息offset存放在kafka本地，所以在启动消费者进行数据的时候，不需要再连接zookeeper，也就不需要zookeeper参数了。生产者生产数据的时候，需要找到leader，连接Kafka就能将生产的数据推送到kafka。也即生产中是不会直接和zookeeper打交道的，只和kafka集群交涉，但是消费者在0.9之前要和zookeeper打交道以获取offset的信息，0,9之后便只是和Kafka集群交涉了。

redis的主从复制的时候，是master将向slave发送写的指令，为什么呢？因为redis的master读的多写的少；但是kafka就不是这样的设计了，因为kafka的leader既要读又要写，还要给flower发送数据，会太累的，所以不能这样做，所以是，flower去leader里面取数据。一段时间就轮询问一下， 如图：

## Kafka的架构



Producer：向kafka broker发消息

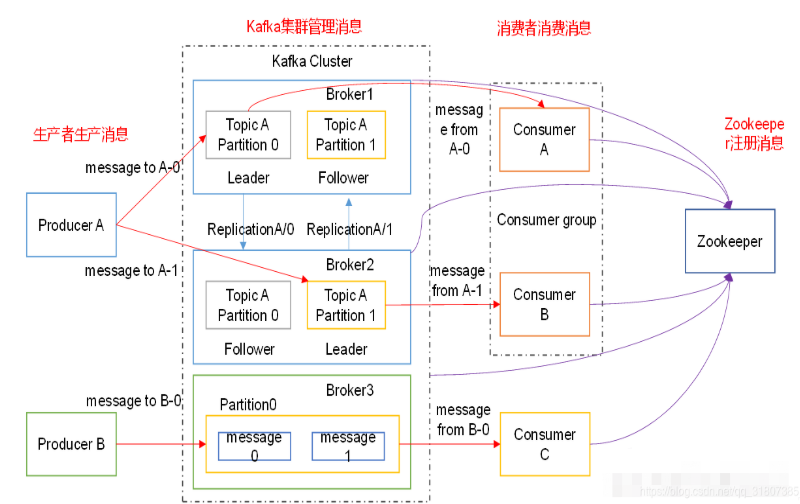
Consumer：从kafka broker中取消息

Kafka依赖zookeeper，kafka在启动的时候会写入broker 的id信息，分区信息，主题信息等到zookeeper中，Consumer还会将消费的偏移量offset记录到zookeeper中，记录当前消费到哪里了，下次消费的时候，从下一条数据开始消费。

Kafka为了提高负载，将数据分区存到不同的broker上，为了数据的安全性，将分区中的数据做了备份，进而产生了Leader和Follower，Follower只是负责备份，读写都由Leader负责。

同组中的消费者一般负责消费一个分区数据，如果分区数量是2，消费者组中的消费者数是3，那么将会有一个消费者处于空闲状态，kafka避免出现多个消费者消费一个分区的情况出现。在同一个分区中，消息是有序的。

## Kafka的工作流程



以上图的为例：多台机器构成kafka消息队列的集群，主要是为了负载，每一台服务器叫做broker，就是代理服务器的意思，多个服务broker在一起，就形成了一个集群，为什么这么设计？

①生产者生产数据的时候，理论上说一台机器来接收数据就OK了，但是如果生产者的产能很大，大量的数据涌向一台机器，这台，能干嘛？只能崩溃，就像情绪，小情绪反差出幸福，大情绪，让我们抑郁，所以我们这样做：增加多台机器，然后将大量的数据分区，将数据按照分区放到不同的机器上去，这样单台机制承担的数据压力就小了。这就是所谓的提高负载。

②如上图，生产者生产的消息，一部分分到了1分区，另外一部分分到了 0分区，如此一来，信息没有放在一块，数据量就少了，性能就会提升，当数据到达broker之后 ，可是发过来的数据到底是谁在关心呢？因为消费者不可能消费所有的消息，所以我们将消息划分逻辑主Topic，对不同的消息做一些分类，有了topic的概念，定义了主题之后，将相关的消息放进来就OK了，这样消费者消费的时候，就可以只是消费他关心的数据；这里分区是和主题相关的，比如订单（TopticA），订单的主题中有两个分区，所有就有Topic-A，partition-0，另外一个就是Topic-A，partition-1，这是两个不同的分区，但是从属于一个topic；

③将数据按照分区发送到不同的分区是没有任何的问题的，但是如果某一台机器宕机了，可见数据是不安全的，所以在这种情况下，我们应该给partition分区做一个备份，所以这里有一个replication的操作，将数据做一个副本，为了安全，这个副本应该被放置在不同的机器上，这样就有了leader和follower的概念，这里的副本仅仅是为了备份，读写都只有leader才能够进行，表现在图中，就是生产者的箭头，仅仅指向了leader，同理消费者；

④生产者往集群中存放消息，这个时候消费者就可以消费了，也就是从集群中取消息，这个时候涉及到一个组的概念，在kafka中，消费数据都是按照组为单位来消费的，这就意味着，TopicA中的所有数据都是组拿到的，组内的单个成员去消费不同的分区，这样得到的数据是不一样的，而整个组拿到TopicA的所有数据；对于本图而言，如果有A B C 三个消费者，A会读取p0，B会读取p1 ，C怎么办？这个C什么都不做，一般情况下，消费者中的消费者和分区的个数是保持一致的，这样不会浪费消费者，所以如何来增加消费能力呢？最基本的条件是增加分区。消费者个数和分区个数保持一致。图中的ConsumerC是一个消费者，但是也是一个消费者组；

⑤这里有分区0 和分区1，对于消费者组来说，他无法保证他是先读取到分区0中的数据，还是先读到分区1 中的数据，这里影响的主要因素是网络的原因，但是消费者能够保证的是：同一个分区中数据是有序的，因为消息的内部的实现方式是队列。所以在实际的业务中，如果有排序，就哟啊考虑往一个分区里面放，这里注意，往一个分区里面放，并不是说整个topic有一个分区，一个Topic可以有多个分区。

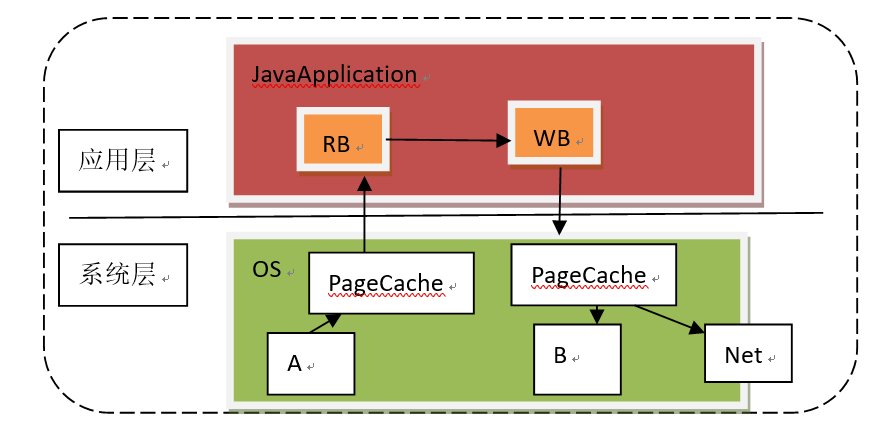
## Kafka的数据生产

## Kafka为什么很快？

### 顺写日志与零拷贝

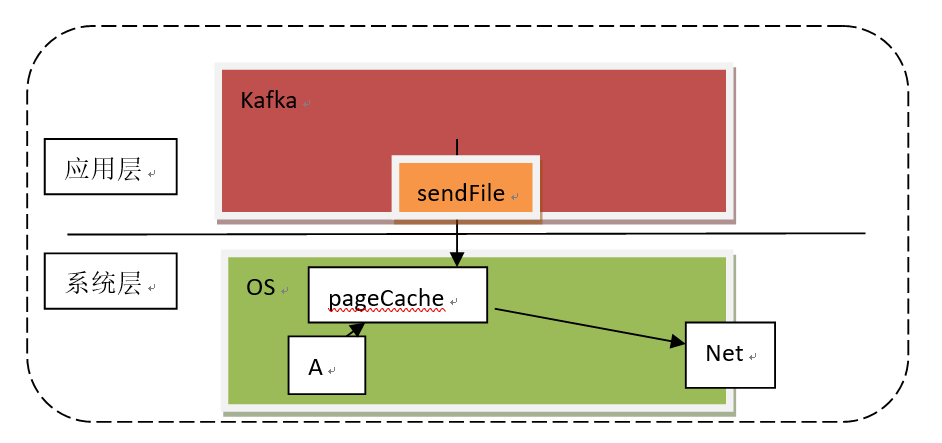
Java应用程序复制文件的原理：

使用java实现对文件A的拷贝，为B，A ,B两个文件都是属于系统的，所以呢这里由应用层的JavaApplication来完成这个操作，在java中创建BufferInputStream将A文件读取过来，记为RB（缓冲）所以在java的应用中有 一个缓冲区，如下图：



|  |
| --- |
| 实际上，我们的系统层在和别的层做交互的时候，为了提高效率，也会有一个缓冲区，因为系统层在和别的层级进行交互的时候涉及到两个层级的交互，效率是非常的慢的，这有些像CPU和内存的交互速度差异很大，cpu 和硬磁盘的交互也速度差异也很大，因此计算机的内存模型有四层（早期的时候CPU和硬盘直接交互的，后来，增加了内存，缓存等层级）计算机的内存模型如下： |

上面我们提到的OS的缓存我们称之为PageCache，文件A的内容先读到PageCache中，然后到达RB，此时java程序拿到了流，要进行写操作，所以此时有一个WB（writebuffer）然后..才到B文件，这就完成了文件的拷贝工作，我们假设这里不是文件的拷贝而是文件的传输，在 PageCache之后，还有一个NetCache（网卡的缓存）所以这里要拷贝四次，假如有10个消费者要读取数据，就4 \* 10 = 40次，这不可能快的起来。

kafka，对这个文件的缓存做了极大的改善，如下：kafka只是向OS发出了一个指令，告知OS这个文件要传输到哪里去，并不对数据进行拷贝。只是将A中的数据拷贝到PageCache中去，如下图：

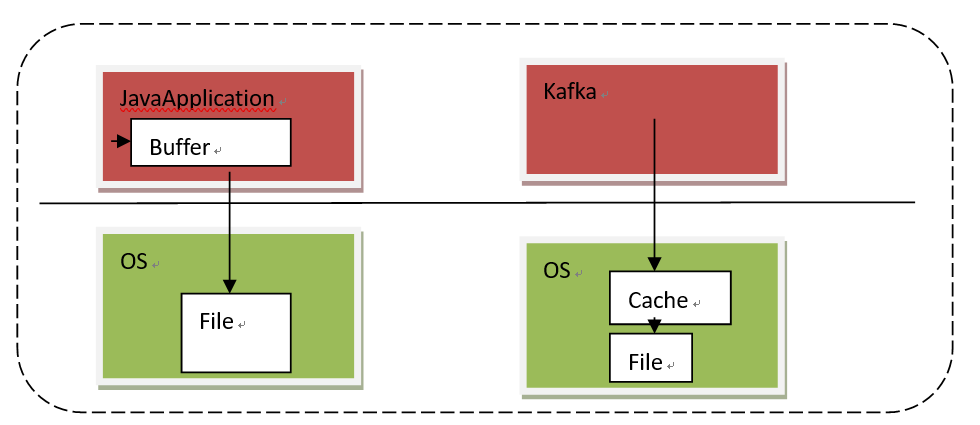
这叫做kafka的顺写日志，由于kafka没有对数据 进行复制，这涉及到②零复制，kafka发送一个操作系统级别的指令，sendFile，（和redis中的fork指令一样，就是在redis在创建rdb的时候，redis会创建一个分支进程，这个进程会将当前的缓存数据保存到一个文件中，这个过程靠的就是fork这个底层的指令 ）很多的框架为了提到效率和我们的OS 打交道，这样的话性能是很高的，比如juc的copyOnWriteAarrayList所以在kafka中，生产者生产完数据之后，不是OS产生数据，在到Java，在到OS，kafka会直接直接顺写到磁盘，即生产完数据之后，直接发指令到OS将数据写到磁盘中去。

### 分段日志：

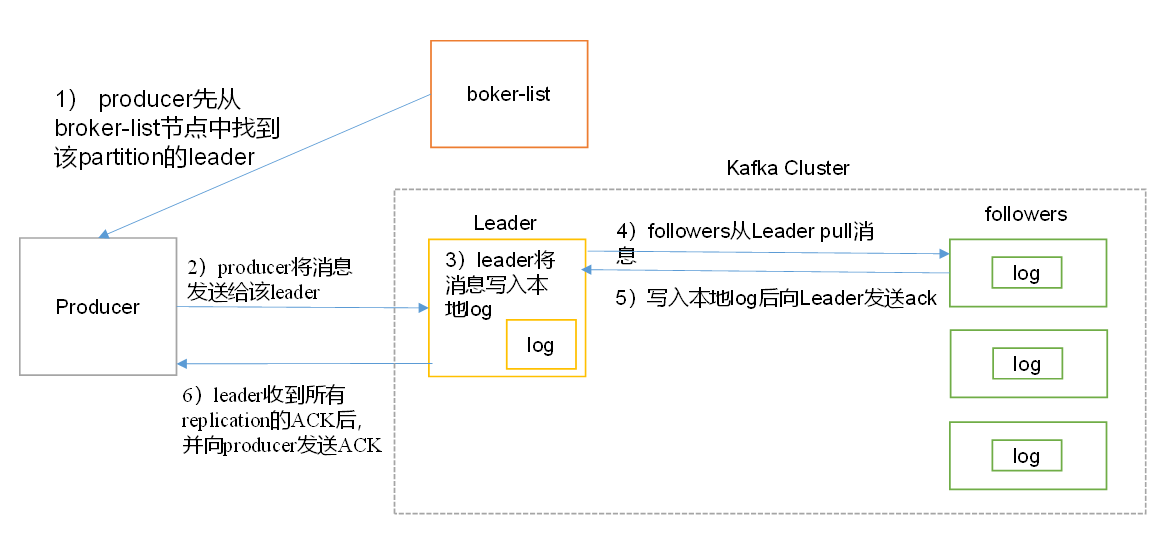
分段日志 segment（其实在juc中也类似的应用，叫做currentHashMap使用的就是分段锁，分段锁就是把每一个小格子加了一把锁，来提到并发访问的能力）在kafka形成日志的时候，实际上也是在以分区的形式保存着数据信息，在分区中的信息如下：├── first-0│ ├── 00000000000000000000.index│ ├── 00000000000000000000.log│ ├── 00000000000000000000.timeindex│ └── leader-epoch-checkpoint├── first-1│ ├── 00000000000000000000.index│ ├── 00000000000000000000.log│ ├── 00000000000000000000.timeindex│ └── leader-epoch-checkpoint├── first-2│ ├── 00000000000000000000.index│ ├── 00000000000000000000.log│ ├── 00000000000000000000.timeindex│ └── leader-epoch-checkpoint00000000000000000000.log里面存放着数据（共20个零），如：[isea@hadoop108 first-2]$ strings 00000000000000000000.log kafka在分区中，这个00000000000000000000.log会随着数据的增加，而产生新的文件，比如，文件记录10条数据会产生一个新文件，00000000000000000000.log 00000000000000000010.log00000000000000000020.log假如我现在要找第12条数据，那么消费者就不需要从头开始遍历，而是直接定位在（10<12<20）00000000000000000010.log里面找。另外，每一个文件还有一个index文件：这个文件专门来记录索引值的，索引值能帮助我们快速的定位到我们数据的位置00000000000000000000.log 00000000000000000000.index00000000000000000010.log00000000000000000010.index 00000000000000000020.log00000000000000000020.index假设这里面我们寻找第12条数据，首先按照文件的名字计算10<12<20那么在00000000000000000010.log中，大范围确定了，但是具体在哪里呢？假设00000000000000000010.log中数据是这样存储的，1 iseayou 2 isea3 ilvoeyou这种形式是为了好表示 ，实际上数据是这样的1iseayou2isea3ilvoeyou那么这个时候，我们怎么知道数据在哪里呢？所以我们使用00000000000000000010.index 这个文件中存放着偏移量和所对应的字节的位置，是一份键值对的形式这里是1 02 93 14找2，那么直接从9的位置开始读，所以就会非常的快，所以追加到partition log中的每一个消息都有一个消息编号来表示这是第多少条消息，还有都会有一个唯一的offset的值，就是这里的0，9 ，14在深究一下，还有一个问题，从9开始读，读多长或者说是读到哪里停止呢？这个时候涉及到kafka对于数据存储，kafka是按照k,v存储数据的，消息有固定的格式，这个格式中可以根据key，v的长度信息来计算出value的长度，从而做消费数据的时候，读到读完该条消息就停止了。我们发现kafka将数据分成了好几个文件，形成如下的形式：00000000000000000000.log 00000000000000000010.log00000000000000000020.log而不是将一个数据作为一个整体，这就是分段日志。

### 预读和预写

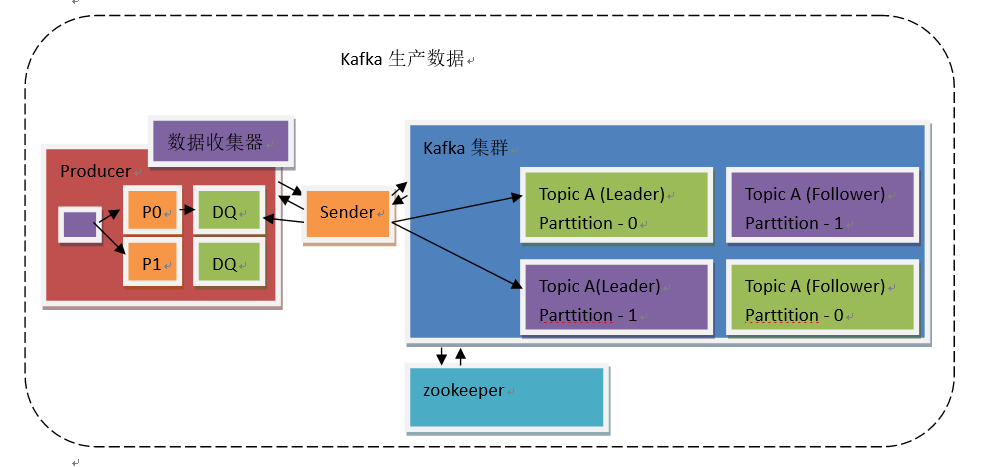
read ahead， write behind，预读就是在读取某一条数据的时候，并不是单纯的将数据读完，因为（读数据，也可能读周边的数据，也即局部性原理）于是在读本条数据的同时就顺便把周边的数据也读出来放到缓存中；后写：java生成文件是先放到缓存中，然后在到OS系统的文件中，而kafka会直接将数据写到OS的缓冲中，再由OS自己来决定什么时候把数据变成文件。两者都是讲数据放到文件中，区别在于两个层级的交互：java程序中，数据先到buffer，然后在跨层级写到OS的文件中（这个过程是很慢的额）而kafka是将数据直接传输到OS的缓存，然后OS做同一个层级的写交互，是非常快的。



## 生产者写数据流程



这里producer不会直接和zookeeper直接打交道，而是先访问kafka集群，kafka集群要想做操作一定要和zookeeper连接，所以这个关于leader的，主题，副本的信息，kafka一定能够拿到，也就是说producer，生产了数据想要知道数据应该放在哪里，只需要连接集群就够了，集群里面会自动的从zookeeper中获取分区的leader，从而生产者知道了谁是leader，有了leader之后，就能够把数据直接放到leader里就完事了，所以这里不是直接和zookeeper直接打交道，**zookeeper一直在和kafka连接，producer连接kafka就能获得副本（使用Kafka的时候必须要开启zookeeper）**，分区等信息，效率会非常的高而不是去连接zookeeper（那样发一次数据先和zookeeper建立连接，在和kafka建立连接，这很没有效率），如图：比如在我们在命令行连接生产者的时候，就是直接连接的集群。而消费的时候我们连接的是zookeeper集群，要获取offset信息，0.9之后，连zookeeper都不用了。



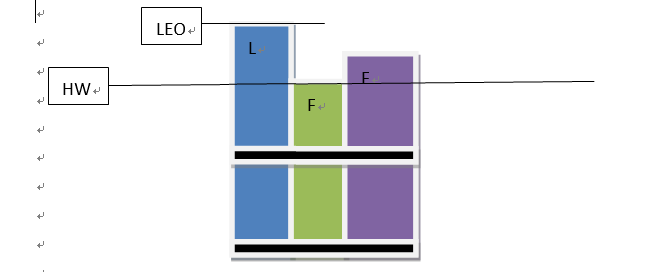
kafka生产数据的流程，producer要生产数据，需要存放在Kafka集群上，需要获取kafka的分区信息，主题等信息，这些信息存储在zookeeper上，zookeeper一直和kafka连着，producer会获得分区的信息，leader信息，producer和kafka连接的时候使用的**sender 发送器**。此时producer生产了数据，要发往到TopicA，kafka集群快告诉我leader在哪？以sender发送，然后返回，消费者知道了有多少个分区，有几个leader，假设此时，产生了一个消息，这个消息放在哪个分区呢？分区算法来实现，要是指定了，就放进指定的分区，如果制定了key，计算hash得到一个分区，如果都没有，轮询，可能这个放0，下个放在1里。但是这个时候数据还是不会发送，一条一条发，多累啊，多慢啊，所以可以使用批处理的机制，这个时候呢在这里加一个队列，但是这个对列式双端队列，队列满了就交给sender发送，所以sender的作用就是不断的周期性的轮询DQ（验证两个条件：是否满了，是否到时间了）sender会将数据往Partition 0的Leader或者Partition1 的Leader发送数据，图中数据收集器的作用是决定了DQ中的数据如何做操作， DQ中的数据到达了Kafka之后，producer怎么知道有没有成功呢？（保证数据的安全）这就有了ACK应答机制。。。。这样收到了数据，完成了kafka的数据写入，这里的DQ：一头放，sender接触的那端又放，又读，这样sender发送失败了，还会继续把数据放回来，这样在消费的时候，还是有顺序的。这就完成了数据的写入。

## ACK应答机制

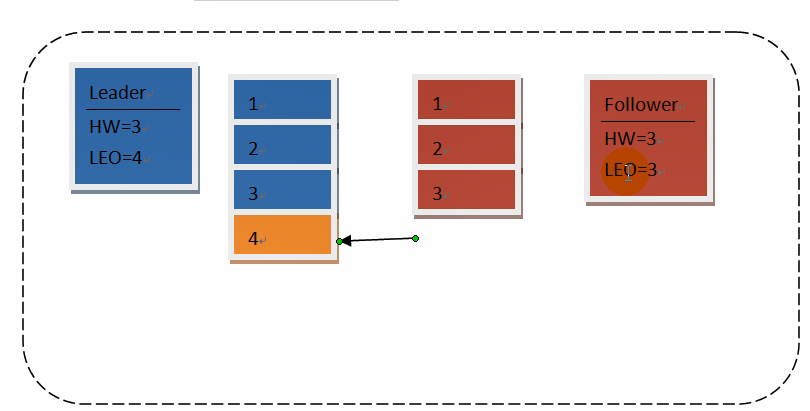
produce向leader发送了数据之后，怎么知道我的数据发送成功了呢？

|  |  |
| --- | --- |
| ACK=0 | Producer直接向leader发送数据，不问kafka的leader有没有收到 |
| ACK=1 | Producer发送数据之后需要等待Leader的Ack确认 |
| ACK=-1（all） | Leader等待follower的ACK |

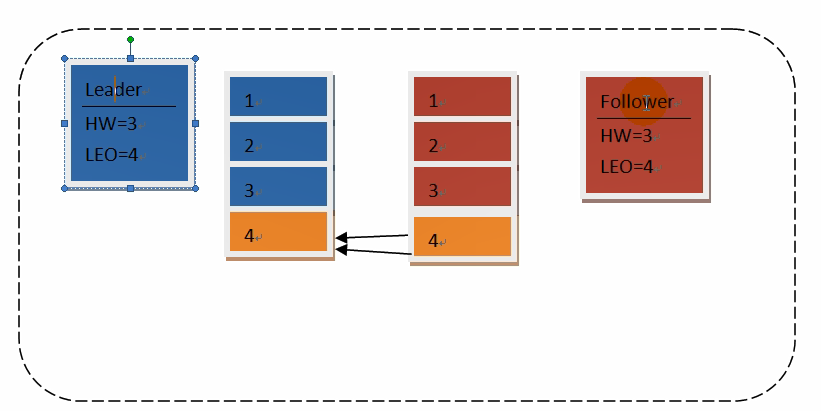
数据的保存就涉及到了上面说的kafka的高并发的四个概念。kafka拿到数据，发送sendFile指令，余下的交给OS来负责。分页日志，零复制，顺写日志，预读后写；这就很快完成了数据的写入和存储，但是这里有一个问题，就是ack为1的时候：leader挂掉了，follower还没有获取到leader的数据，此时follower反客为主，成为leader，但是数据不全怎么办？HW：High WaterMark :最高水位，HW是所有副本中最小的LEO值；LEO：Log End Offset ：最后的偏移量，用户看不到这个数据。



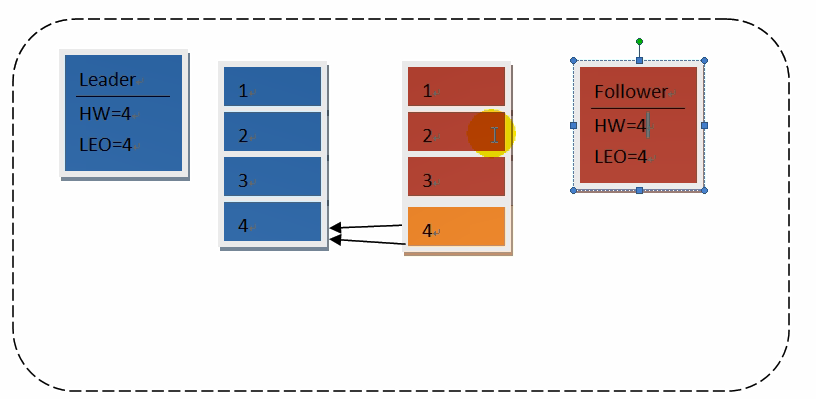
ack为1的时候，这里一共有两种情况，①



此时follower发现leader多了一条数据，于是将4数据取过来，保存，此时LED = 4 但是这个时候，但是这个时候水位是没有变化的，因为还需要提交告诉leader保证数据收到了，只有leader收到了ack，leader的水位变化之后，follower才会变化，但恰巧，刚读过来的数据，follower还没有提交，leader挂了，这个时候等于数据没有取过来，水LEO还是3，此时follower反客为主，为leader了，对于用户来说数据并没有丢失，因为之前看到三条数据，现在任然还是三条数据， ②，



数据拿过来了，提交ack的时候，发现leader挂了，follower变为leader，现在只有一个机器了，HW变为4 ，此时数据还是没有丢。③



ISR表示的是当前还有哪几个副本可以同步数据

# Sqoop

Sqoop主要用于Hadoop（Hive）与传统的数据库（MySQL）之间进行数据的传递，将关系型数据库中的数据导入到HDFS（Hive），有可以将HDFS中的数据导入到关系型数据库中。

## Sqoop原理

将导入或导出命令翻译成MapReduce程序来实现。

Sqoop是需要连接关系型数据库，所以需要将mysql的驱动拷贝到sqoop的lib目录下。

# Kylin

Kylin是一个开源的分布式分析引擎，提供Hadoop、Spark之上的SQL查询接口和多维OLAP能力以支持超大规模的数据，能够做到亚秒级查询巨大的Hive表。

# HBase

HBase的原型是Google的BigTable，可**以支持结构化的数据存储**。HBase是一个高可靠，高性能，**面向列**，可伸缩的分布式存储系统，利用**HBase可以实现在廉价PC上搭建大规模结构化存储集群**。

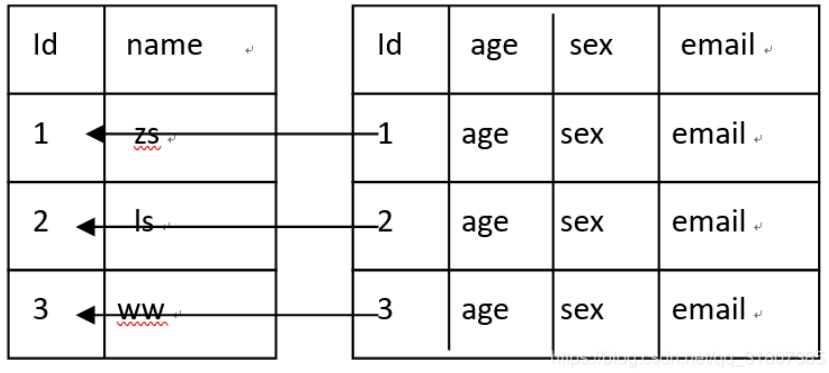
在HBASE中使用zookeeper实现HMaster的高可用，google的大表使用的是Chubby。

面向列存储指的是面向列簇存储，HBase根据列簇来存储数据，列簇下面可以指定很多的列，列簇在建立表的时候，就必须指定。

MySQL数据库是面向行存储的，取数据的时候，也是按照行来存储的。

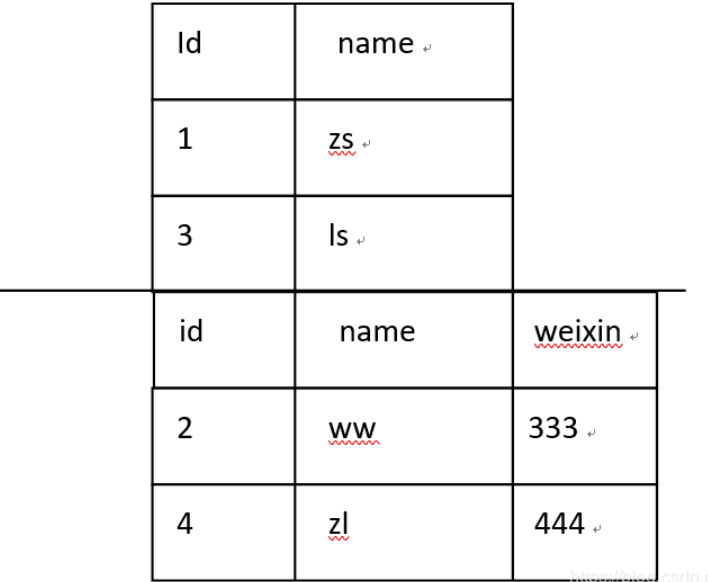
HBase的优点（为什么不使用MySQL集群而是使用，HBase存储）：因为HBase有若干的优点，①：海量存储，适合存储PB级别的数据，并且基于廉价的机器，能够在几十毫秒内相应数据；②：稀疏性，HBASE面向列族进行数据的存储，如果列的数据为空，不会占用存储空间；③：易扩展，基于HDFS，还可以横向扩展RegionServer，④：高并发

MySQL的宽表优化：表中的数据越多，查询的效率就会越慢，因为MySQL在进行查询的时候，每次都要查询所有的字段，所以如果表很宽的话，就会影响查询的速率。所以在设计表的时候，需要把两张表拆分成一张表，来提升查询效率。

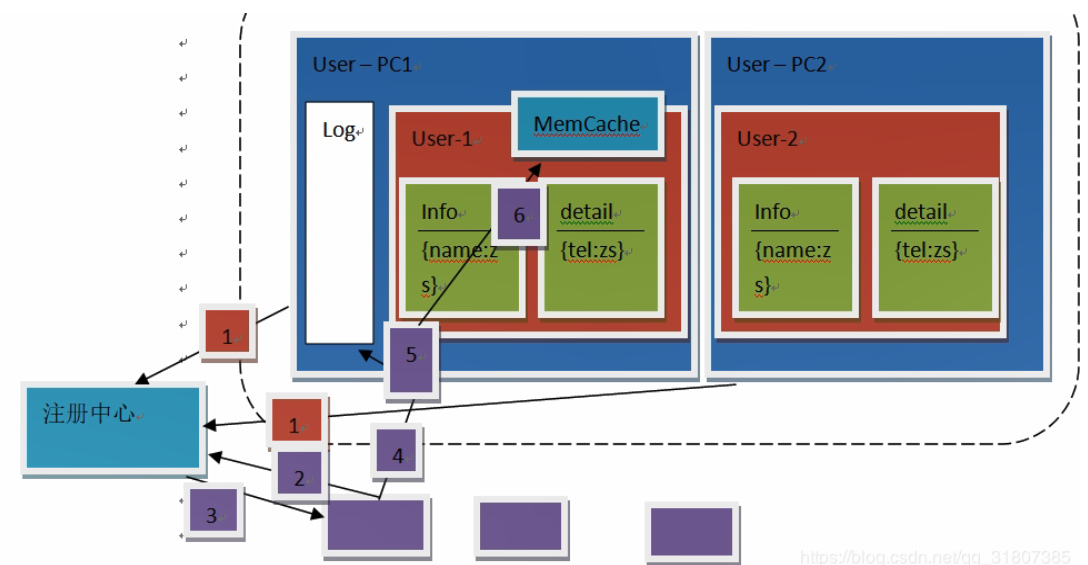


宽表的优化：

如果数非常的深，查询的效率也会收到影响，高表的优化策略是水平拆分，分成两个部分，在插入和查询数据的时候，使用分区的策略，如下图：

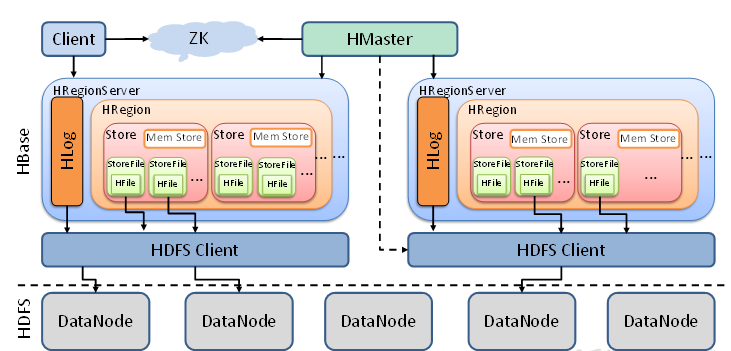


由于大数据要处理的数据量非常的大，所以HBase在设计之初，就考虑到了高表和宽表拆分问题，另外为了提高效率，HBase会将写操作放置在缓存中，对于新产生的安全性的问题，又产生了log日志，最后HBase的设计模型大约是下面的这个样子：



<https://blog.csdn.net/qq_31807385/article/details/84994267>

## HBase的架构



Region是HBase表的分片，HBase表会根据RowKey的值被切分成不同的region，存储在RegionServer中，在一个RegionServer中可以有多个不同的region。

Store，HFile存储在Store中，一个store对应着Hbase表中的一个列族。一个region中可以有多个store。

Memstore：内存存储，用来保存当前的数据操作所以当数据存储在WAL中之后，RegionServer会在内存中存储键值对。

Write-Ahead logs：HBase的修改记录，当对HBase读写数据的时候，数据不是直接写进磁盘，它会在内存中一段时间，但是由于这样可能会导致数据丢失，数据会先写在这个文件中，然后在写入内存。也叫HLog。

StoreFile是原始数据的实际的物理文件，是实际的存储文件，StoreFile是以Hfile的形式存储在HDFS上的。Hfile是一种文件格式，类似TXT。

### Client

Client包含了访问HBase的接口，还维护了对应的cache来加速对HBase的访问

### Zookeeper

1. HBase通过zookeeper来做Master的高可用，保证集群中只有一个Master在运行，如果Master异常，会通过竞争机制产生新的Master
2. 通过zookeeper来监控RegionServer的状态，当RegionServer有异常的时候，通过回调的形式通知Master RegionServer上下线的信息
3. 通过zookeeper存储元数据的统一入口地址。

### Master

1. 为RegionServer分配Region
2. 维护整个集群的负载均衡
3. 维护集群的元数据信息
4. 发现失效的Region，并将失效的Region分配到正常的RegionServer上。
5. RegionServer失效的时候，协调对应的HLog的拆分

### RegionServer

RegionServer直接对接用户的读写请求，是真正干活的节点。

1. 管理Master为其分配的Region
2. 处理来自客户端的读写请求
3. 负责和底层HDFS的交互，将数据存储到HDFS上
4. 负责Region变大之后的拆分
5. 负责Storefile的合并工作。

## HBase的数据结构

### RowKey

数据按照Rowkey的字段序在HBase表中排序存储。可以是任意字符串。实际常用的是60-100字节。

### ColumnFamily

列族：HBase表中的每个列，都归属于某个列族，列族是表的一部分。

### Cell

Rowkey，column Family，column，version。确定唯一的单元。Cell中的数据是没有类型的。

cell的数据存储格式是字节码。

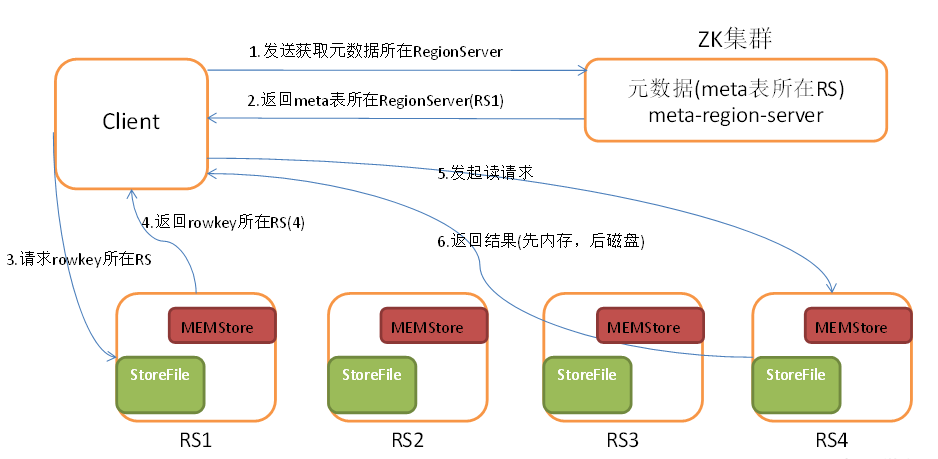
### TimeStamp

Rowkey，列族，列名，版本确定唯一的单元，在单元中，都保存一份数据的额多个版本。版本的信息就是时间戳来确定的。最新的数据排列在最前面。

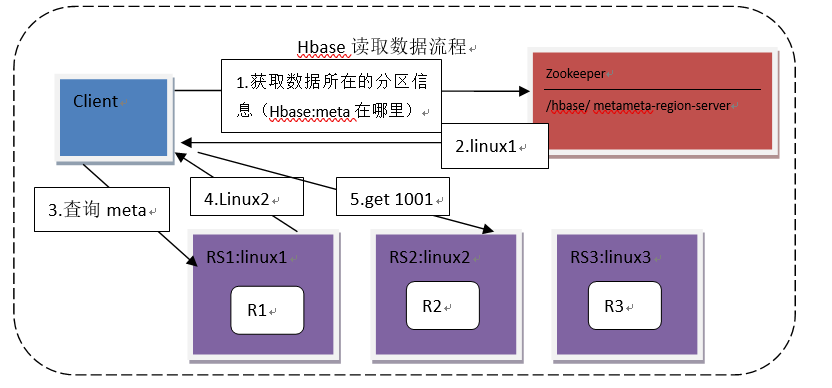
HBase常用的命令，scan扫描表，put添加数据，get获取数据，delete，删除数据，还有超级强大的help

## HBase的原理

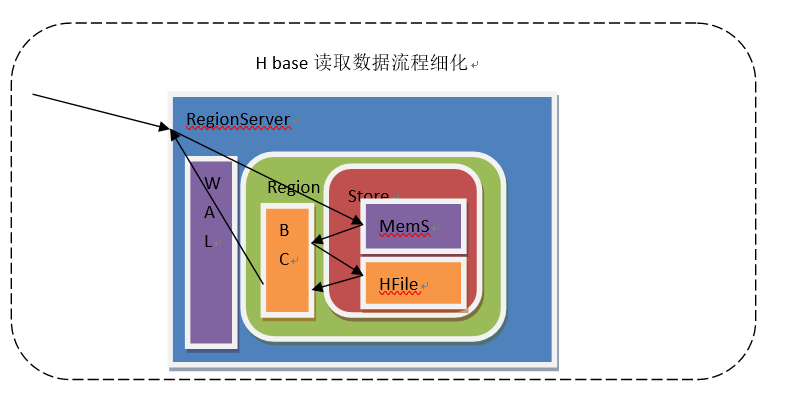
### HBase读流程



1. client先访问zookeeper，获取meta表所在的region的位置，然后读取meta表中的数据。
2. 根据namespace，表名，rowkey在meta表中找到对应的region的信息
3. 找到region对应的RegionServer，并找到对应的region
4. 先从Memstore找数据，如果没有，在到BlockCache里面读
5. BlockCache还没有读到，再到storeFile上读数据，如果在StoreFile中读到了数据，并不是直接返回给客户端，而是先写入到BlockCache，在返回给客户端。

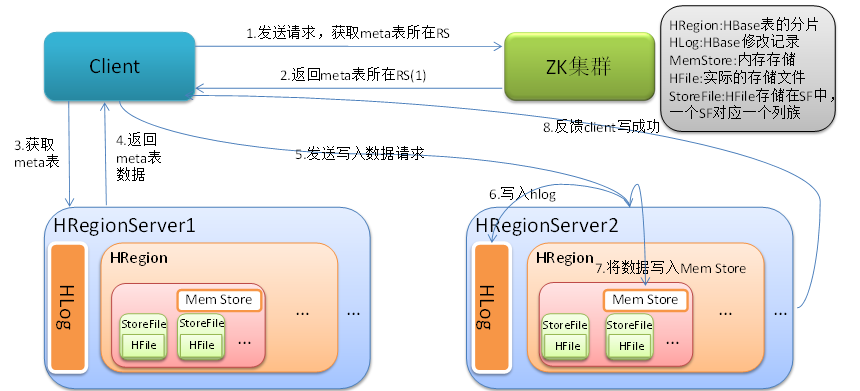


以上client就获取了数据在哪台个RegionServer的哪个Region里面，获取到该数据阿按照下面的步骤；



1. 读取MemStore（写缓存）
2. 读取BlockCache(读缓存)
3. 读取HFile(真正的数据文件)
4. 将查询结果放置到BlockCache当中
5. 将结果返回给客户

### HBase写流程



1. client向RegionServer发送写请求
2. RegionServer将数据写到HLog（write ahead log）为了数据的持久化和恢复
3. RegionServer将数据写到内存。

### 数据刷写

当满足一定的条件的时候，RegionServer会将数据刷写到磁盘，将内存中的数据删除，同时删除HLog中的历史数据。

并将数据存储到HDFS中

在HLog中做标记点。

### 数据合并

数据的合并，在数据刷写的过程中，会产生很多的文件，当数据块到达了3块，maser即会触发合并操作，region将数据加载到本地，进行合并我们需要合并小文件。当数据合并的容量超过了256M的话，要将数据进行拆分。拆分后的Region分配给不同的RegionServer进行管理，当RegionServer宕机之后，还会将RegionServer上的hlog，然后分配给不同的RegionServer进行管理。

**合并和拆分和master有关，数据的读写和mater没有关联**

合并的小文件只能同一个列族的，也即只能合并同一个store的。如果列族设置的过多的话，就会有多个Memstore，一旦触发了region的内存的最大设置的话，会一起刷写到磁盘，从而产生多个小文件，而这些小文件从属于不同的列族，他门并不会合并。

当合并的数据超过一定的大小，还要进行拆分，将拆分后的region分配给不同的RegionServer

### HBase的优化

#### 高可用

Master负责监控RegionServer的生命周期，均衡RegionServer的负载，如果master挂掉了，那么整个集群将进入不健康的状态。

在HBase的conf目录下的的backup-masters中添加备用的master节点

#### 预分区

每一个region都维护这startRow和endRow，如果加入的数据符合某个region维护的rowkey的范围，则将该数据交给该region维护。按照这个原则，我们可以将所要放置的分区，提前规划好，以提高HBase的性能。

#### Rowkey的设计

一条数据唯一的标识就是rowkey，那么这条数据存储到哪个分区，取决于rowkey处于哪个分区的区间，设计rowkey的主要目的就是为了让数据均匀的分布于所有的region，防止数据倾斜。

设计原则：

1. 唯一性原则：rowkey就是关系型数据库中的主键
2. 长度原则：64k，推荐10-100byte，一般是60-80字节，一般是8的倍数，在满足查询的条件下，越短越好。
3. 散列原则：希望将rowkey所代表的数据均匀的分布到不同的分区中。盐值散列，一般放置在rowkey的前面或者使用字符串翻转或者添加分区号。