初识Kafka

- 消息队列：系统解耦、异步通信、削峰填谷
- Kafka Streaming 流处理
- 消息队列两种类型：1.至多被消费者消费一次，即删除队列中的数据；2.没有限制：多个消费者消费多次，消息服务器相当于当数据库存数据用了

基础架构

- topic只是一个逻辑上的概念，topic由各个partition构成、数据存储在各个partition中。partition有备份，这很好理解，但是partition居然还有主（Leader）从（Follower）的概念，对比hdfs DataNode的block，这可没有主从的概念。

```
record: key/value/timestamp
topic
partition
broker:服务器实例
```

- 生产者可通过指定/round-robin/key hash % partition来将record发送至topic的哪个partition中。分区中每个record都被分配了唯一的序列编号称为是offset，offset越小说明进入消息队列时间早

```
log.retention.hours=168 //record默认持久化时间为7天
```

- kafka只能够保证分区内的数据是有序的，并不能保证分区之间的数据顺序。比如：partition-0的offset-0 与partition-1的offset-0的时间顺序就无法比较。局部FIFO
- 那kafka为什么要有分区这个概念呢？理论上你topic分区越多，分区所分配的Broker也越多，起到了一个动态扩容服务器的意思，你写入性能遇到瓶颈了，加分区->则提高写入性能，高并发场景写入，当然加分区肯定存储能力也扩大了。总结下来就是：1.高并发场景快速响应；2.大数据场景存储海量数据
- 消费者之间是相互独立的，消费者可以控制更改消费者的偏移量
- 消费者组（consume group）：topic中每个分区的数据，仅会发给消费者组中的一个消费者实例。这里的概念其实是和上面“消费者之间独立”这概念冲突的，除非把“消费者组”就是看成“一个逻辑的消费者”才说得通。分区越多，消费者组里面能够并行运行的消费者实例也就越多。所以说，分区才是kafka实现高吞吐的关键
- 高性能写关键：顺序写和mmap。磁盘的写入性能可以和内存比较
- mmap：内核空间PageCache与物理磁盘有直接的映射关系，用户空间程序只考虑把数据写入PageCache就成，就算写入成功了。由内核去控制何时把PageCache的数据刷盘。确定：内核不稳点，断电丢数据
- 高性能读关键：ZeryCopy（零拷贝）
- ZeryCopy：磁盘数据无需拷贝到用户空间，而是直接将数据通过内核空间传递出去
- kafka segment 1G ?

Kafka环境搭建和Topic管理

- rpm卸载jdk

```
rpm -e `rpm -qa | grep jdk`
```

- 开机启动项目

```
chkconfig --list
chkconfig iptables off
```

- kafka server.properties 配置参数修改三个地方

```
listeners=PLAINTEXT://node01:9092
log.dirs=/var/kafka/kafka-logs
zookeeper.connect=node01:2181/kafka
集群配置：zookeeper.connect=node01:2181,node02:2181,node03:2181/kafka
```

- 查看参数使用：kafka-topics.sh --help
- 注意：Kafka2.0以后使用的是--bootstrap-server node01:9092 而不是--zookeeper
- 消费者组内里面消费者有种均衡负载的效果：多个消费者共用一个组group01，在生产者生产消息，多个消费者轮训均衡负载，一个一个消费
- 消费者组内均衡负载、消费者组间广播机制
- linux 关机：shutdown -h now
- topic分区的修改，只能增加，不能减少

Topic管理API

Topic基本DML操作

生产者

- 注意：发送key-value数据之前，指定好key-value序列化规则

消费者 sub/assign

- 消费者一定是跟consume group 挂钩的

自定义分区

- consumer有两种消费方式：1.subscribe；2.assion(手动指定分区、那就不需要指定消费组了)
- kafka默认分区策略：有key就使用hash取模的方式，没有key就使用轮询策略
- Partitioner接口，自定义分区策略则自己实现这个类

序列化

- 自定义序列化

拦截器

- 自定义Producer拦截器，实现拦截器接口Interceptor

Kafka高级API

- 消费者：offset
- 生产者：ack/retry

offset自动控制

- 默认的offset策略是，如果生产者先启动，再启动消费者的话，那么消费者只能消费最新的latest。这里有个很重要的前置条件：系统并没有存储该消费者的消费分区信息的时候，也就是消费者首次消费的时候。前面说过了，消费者消费后，会把自己的偏移量offset提交给kafka系统，kafka系统会记录的
- 消费者消费后，会把自己消费的offset+1提交给Kafka系统，默认是每隔5秒提交一次，这一点很关键，决定了消费者的消费位置是从哪儿开始的

Kafka架构模型思考

- 由性能问题->Topic引入Partition，分而治之，将数据分散到多台机器，那么问题来了，将数据聚合起来的时候，如何保证顺序？ 要明白数据之间是否需要有关联的顺序性，有关联的顺序性的数据那么只能放在一个Partition里面了，无关联的可以分散到不同Partition无所谓
- 单点问题->Kafka的Partition复制是主备模型，主做读写，从不处理（数据的一致性问题）
- AKF（X轴、Y轴、Z轴）
- Offset：每个消费者组的offset肯定不一样，因为消费者组之间彼此毫无关联，可以都消费Kafka中的数据
- 方法论：回归本质的一种抽象
- Broker:可以理解为物理机上的一个JVM进程，分布式，那么必然多个物理机多个Broker，那么哪个Broker才是真正的Master来控制Topic，Partition呢?
- 想一想为什么Kafka中会有zookeeper？-> 分布式协调，选主功能（选出一个Broker作为主的Controller）
- admin-api：创建Topic、创建Partition个数
- 新版本（0.10以后）Kafka producer不需要从zookeeper中获取Broker列表了，老版本需要。

> 为什么需要这样做？不要给ZK增加负担，把Kafka自身的metadata（Broker、Topic、Partition）维护在自己的broker里，broker之间同步

- 数据的顺序性不是一个简单的问题，想一想，多个Producer，怎么规定好顺序推送到同一个Partition?
- 一个Partition不能对应两个Consumer，这是绝对不行的。Partition：Consumer只能是1:1 或者N：1

- Consumer消费的时候挂了该怎么办?
- offset存的位置在哪里?
- 老版本：消费者维护的offset放在zookeeper中；新版本：消费者的offset放在Kafka自带的一个Topic中；也可存在第三方，如Redis、Mysql
- offset提交（持久化）的频率如何控制?
- 异步提交：例如每隔5s提交一次，假设一种场景，4s的时候consumer挂了，那么就会造成已经消费过得数据没有提交。造成重复消费的问题
- 同步提交：没有控制好顺序，offset持久化提交了，但是业务写失败了。造成数据丢失的问题
- 顺序读写？Kafka、HBase、ES、Mysql myism

Kafka集群搭建

```
kafka-topics.sh --zookeeper node02:2181,node03:2181/kafka --create --topic ooxx --
partitions 2 --replication-factor 2
```

- 想一想这里为什么要填zk地址？因为topic是由kafka controller（相当于Master）的Broker创建，这个主从分布式协调信息存储在zk
- 消费者group自带负载均衡效果（消息默认在消费者组里面轮训），这就意味着消费者可以无限制拓展下去了，性能提高
- Topic在ZK上是能看到一些信息的

```
ZK路径:/kafka/brokers/topics/_consumer_offsets //这是consumer offset的提交会维护在一个topic
中
```

- message是KV，相同的Key一定到同一个Partition中去，但是相同的K不一定是紧靠着一起排列，因为一个Partition可能会有不同的Key
- 消息队列Push（server端主动推送） VS Pull（client端主动拉取）：Push可能遇到的问题，client网卡已经撑爆了，吃不下了，这时候server端得调控自己不push；Pull client端主动按需要去拉取数据
- 颗粒度：按batch pull数据还是一个个pull
- consumer怎么消费一个batch多条数据？1.单线程处理，每条数据按顺序处理，每成功处理一条数据，提交一次offset。那么offset提交频率成本有点高，同时导致CPU网卡资源浪费。2.多线程流式处理，其实取决于处理流程中可以用多线程则用多线程，但要保证最后有一个事务是单线程，要么同时成功，要么同时失败，batch是要么整体成功，要么整体失败，成功则提交offset，失败则不提交offset

Kafka代码实践

- log-end-offset是什么？为什么我生产者生产消息以后，没有消费者消费，但是我的log-end-offset也到了末尾。这与我理解的offset有出入，我理解的offset只与consumer有关
- 编写kafka代码，很少用自动提交的，太容易造成数据重复消费和数据丢失
- consumer rebanlance的意思是，我一个消费组里面增加consumer了，那么会导致Partition重新分配的问题，Partition到底归哪个consumer管

- 原来Partition还是仅能给（消费者组）一个consumer去消费，1:1或者N：1，这一点很重要
- 当多个Partition对应一个consumer的时候，N：1，这时候consumer每次poll是可以取多个Partition的数据的，并且可以针对Partition做Partition的数据分区
- 手动提交offset，是要考虑提交的offset的粒度的。1.每消费一条数据，提交一次；2.每消费完一个分区的数据提交一次；3.每消费一个批次数据提交一次
- 多线程消费kafka数据一般这样做：kafka消费数据，poll一个批量的数据，批量数据按partition分组，每个partition的数据开辟一个线程去处理
- offset是挂在partition上去更新维护的
- 把每次poll看做一个job，job之间必须是串行的
- 多分区rebanlance会不会重复消费？

内部机制

- 数据可靠性问题
- 有想过Kafka数据的流向吗？producer->broker（jvm）->pagecache(kernel)->磁盘，通常kafka写进pagecache就算写成功了
- 分布式：单机磁盘的持久化可靠性转移到集群多机可靠性
- pagecache(kernel)->磁盘，牵扯到顺序写的问题，顺序写肯定比随机写要快。磁盘文件方式保存数据有三部分：1.data 2.index offset 3.index timestamp
- consumer消费数据：pagecache(kernel)->sendfile(in,offset,out)->consumer,零拷贝，就是说数据没有从kernel到broker的过程了，直接由sendfile发送给consumer
- kafka对数据只是发送，没有加工
- ACK机制，控制producer发送消息的可靠性级别，同时影响producer发送的吞吐量

```
ACK=0，producer消息扔向broker就算成功
ACK=1（默认值），producer消息由kenerl刷到磁盘才算成功
ACK=-1，分布式才用得上，最严格，ISR里面消息进度要同步
```

- ISR（In sync replicas）：partition所对应的broker数目，存活着的，它们之间连通活跃的broker数目
- OSR（Out sync replicas）：超过阈值（10s），没有心跳的broker数目
- AR（Assigned replicas）：创建topic给出的partition副本数，broker和partition之间有对应关系，该分区broker的集合数目
- AR = ISR+OSR
- 上面这种机制是数据最终一致性的另外一种方案。以前一致性方案是过半通过，则表示写入成功，这种方案是硬性的，是死的规则
- 数据的可靠性->牵扯出数据的一致性问题
- 当ACK=-1时，ISR中多个broker的消息进度是一致的
- 当ACK=1时，ISR中多个broker的消息进度可以不一致，因为ACK=1只管在master partition写入数据成功就返回了，我又不管你standy partition是否同步成功
- LEO(log end offset)：原来就是指producer在master partition上生产最新消息的后一个位置
- consumer只能消费到High Watermark的位置，而不一定是LEO的位置。想一想木桶原理，consumer只能消费到最短的那块板

- consumer只消费最完整的那一部分数据
- 要一致性可靠性还是要吞吐？这是需要trade off的
- 基本思想：任何DB或者数据存储，拿磁盘做持久化，都有两种选择，1.是不是写进kenerl的pageache就算成功了；2.绝对写入磁盘
- kafka可以当作存储层，但尽量不要当作全量历史数据的存储层
- kafka是会数据裁剪的，原来的很久没用的数据会裁剪掉，所以就有个Low Watermark的概念

内部机制实践演示

演示kafka data

- 找数据都有个index的概念：1.偏移量index；2.time index

> 直接取kafka存数据目录去看就知道了

- 命令：lsof -Pnp 进程号，查看文件对应文件描述符的类型
- 下面命令去看kafka数据，普通vim命令是看不到的

```
命令：kafka-dump-log.sh --files 0000000000000000.index
```

- 为什么kafka data里面.log数据不用mmap?

```
IO相关的知识
普通IO级别的write 数据只到达pagecache，性能快，容易丢失数据
```

- kafka data里面的index文件是类似下面的：offset与真正数据的字节数组位置有一个映射

```
offset：54  position:4158
```

- time index -> index -> data ； 时间戳找到offset，offset找到position，最后找到data

演示ACK等级ISR连通性

- master node 数据同步到 slave node 的时候，切断 slave 到 master的 ack 确认

```
route -n
把通向master的route转向本地
route add -host 192.168.150.13 gw 127.0.0.1
```

- 只有ACK=-1时，切断 slave 到 master的 ack 确认，会影响生产者生产数据。但也就是仅仅影响10s，IRS会把这个slave踢出去，生产者又可以生产数据了

演示通过timestamp去seek指定时间的offset

生产者参数配置、源码原理剖析

- 生产者是怎么推送数据的？是每产生一条数据就推送一次，还是按批次推送数据吗？
- 生产者向服务器推送数据有连接池这个概念吗？
- 源码里面Partition分区器的构造：有个默认分区器，根据key是否为null，决定数据是轮询发送还是固定取key做hash发送
- 池化结构：线程池，内存池
- 源码分析：大致分为三块，worker线程（这里是可以多线程的）、KafkaProducer（对象，线程安全的，也可以new多个Producer对象，但是多个对象配上多个worker线程事情就会变得很复杂）、IO Thread（发送数据的线程）
- 调用流程：producer.send(msg)->Interceptor->Serializable->TopicPartition->RecordAccumulator
- RecordAccumulator : 这是一个缓存数据的buffer，由BUFFER_MEMORY_CONFIG参数控制，默认是32M，里面维护着各种Topic-Partition的双端队列DEQUE，里面有一个个BATCH（默认16k，每个队列）。BUFFER_MEMORY_CONFIG这个值可以稍微设大点
- BATCH值（默认16K），尽量保证你的一个MSG<BATCH的大小，因为BATCH本身是利用了内存池的池化技术的，如果你的一条数据大小>BATCH，那么就得重新申请内存空间了，内存碎片化
- 阻塞一条条发送数据

```
RecordMetadata metadata = producer.send(record).get();
```

- 非阻塞走批次发送数据

```
Future<RecordMetadata> send = producer.send(record);
```

- LINGER_MS_CONFIG：这个配置其实用于非阻塞这种情况，LINGER_MS_CONFIG=30，意味着如果batch还没满的话，我可以等30s再通过IO Thread发送一个批次的数据
- IOThread : 这里有个NetworkClient，这里面包含一个nioSelector，注意这其实是java的selector，没有用Netty的NIO
- MAX_REQUEST_SIZE_CONFIG 这是IOThread从RecordAccumulator里面取数据的大小，默认是1M
- InFlightRequests,这里有个参数MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION(默认值5)，这个参数意味着NetworkClient往broker真正发送数据，broker要给client响应的，如果5次都还没有响应，那么久不发送了
- SEND_BUFFER_CONFIG和RECEIVE_BUFFER_CONFIG，这是TCP层面发送和接收数据包的大小了，默认是32k。 -1 代表使用系统的默认值。用下面命令查看

```
cat /proc/sys/net/core/rmem_max
cat /proc/sys/net/core/wmem_max
```