1、Kafka使用背景

在我们大量使用分布式数据库、分布式计算集群的时候，是否会遇到这样的一些问题：

a. 我们想分析下用户行为（pageviews），以便我们设计出更好的广告位

b. 我想对用户的搜索关键词进行统计，分析出当前的流行趋势

c. 有些数据，存储数据库浪费，直接存储硬盘效率又低

这些场景都有一个共同点：
数据是由上游模块产生，上游模块，使用上游模块的数据计算、统计、分析，这个时候就可以使用消息系统，尤其是分布式消息系统！

2、Kafka的定义

What is Kafka：它是一个分布式消息系统，由linkedin使用scala编写，用作LinkedIn的活动流（Activity Stream）和运营数据处理管道（Pipeline）的基础。具有高水平扩展和高吞吐量。

3.kafka的特点

Zookeeper是一种在分布式系统中被广泛用来作为：分布式状态管理、分布式协调管理、分布式配置管理、和分布式锁服务的集群。

kafka增加和减少服务器都会在Zookeeper节点上触发相应的事件kafka系统会捕获这些事件，进行新一轮的负载均衡，客户端也会捕获这些事件来进行新一轮的处理。

Kafka在底层摒弃了Java堆缓存机制，采用了操作系统级别的页缓存，同时将随机写操作改为顺序写，再结合Zero-Copy的特性极大地改善了IO性能。但是，这只是一个方面，毕竟单机优化的能力是有上限的。如何通过水平扩展甚至是线性扩展来进一步提升吞吐量呢？ Kafka就是使用了分区(partition)，通过将topic的消息打散到多个分区并分布保存在不同的broker上实现了消息处理(不管是producer还是consumer)的高吞吐量。

4.Kafka相关概念

1.AMQP协议

Advanced Message Queuing Protocol （高级消息队列协议）

The Advanced Message Queuing Protocol (AMQP)：是一个标准开放的应用层的消息中间件（Message Oriented Middleware）协议。AMQP定义了通过网络发送的字节流的数据格式。因此兼容性非常好，任何实现AMQP协议的程序都可以和与AMQP协议兼容的其他程序交互，可以很容易做到跨语言，跨平台。

上面说的3种比较流行的消息队列协议，要么支持AMQP协议，要么借鉴了AMQP协议的思想进行了开发、实现、设计。

2.什么是消息系统？

消息系统负责将数据从一个应用程序传输到另一个应用程序，因此应用程序可以专注于数据，但不必担心如何共享数据。 分布式消息传递基于可靠消息队列的概念。 消息在客户端应用程序和消息传递系统之间异步排队。 有两种类型的消息传递模式可用 - 一种是点对点的，另一种是发布 - 订阅(*pub-sub*)消息传递系统。 大多数消息传递模式遵循*pub-sub*。

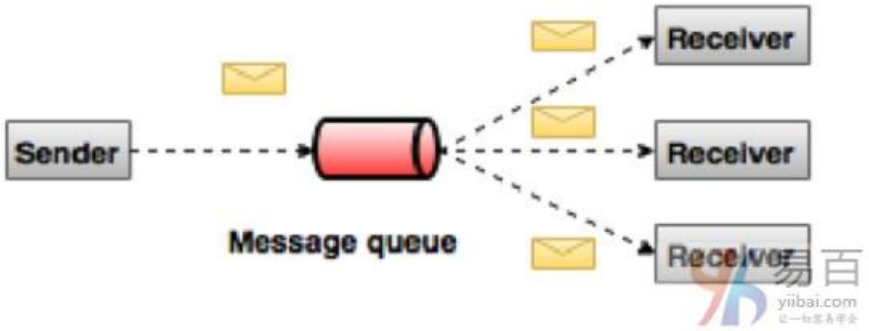
- 点对点消息系统

在点对点系统中，消息被保存在一个队列中。 一个或多个消费者可以消费队列中的消息，但是特定的消息只能由最多一个消费者消费。 一旦消费者在队列中读取消息，消息就从该队列中消失。 这个系统的典型例子是一个订单处理系统，其中每个

订单将由一个订单处理器处理，但是多订单处理器也可以同时工作。 下图描述了结构。

- 发布-订阅消息系统

在发布-订阅系统中，消息被保存在一个主题中。 与点对点系统不同，消费者可以订阅一个或多个主题并使用该主题中的所有消息。 在发布-订阅系统中，消息生产者称为发布者，消息消费者称为订阅者。 一个真实的例子是Dish TV，它发布体育，电影，音乐等不同的频道，任何人都可以订阅他们自己的一套频道，并在他们的订阅频道可用时获得内容。



来自〈https://www.yiibai.com/kafka/apache_kafka_introduction.html#article-start〉

3. 一些基本的概念

1、消费者（Consumer）：从消息队列中请求消息的客户端应用程序

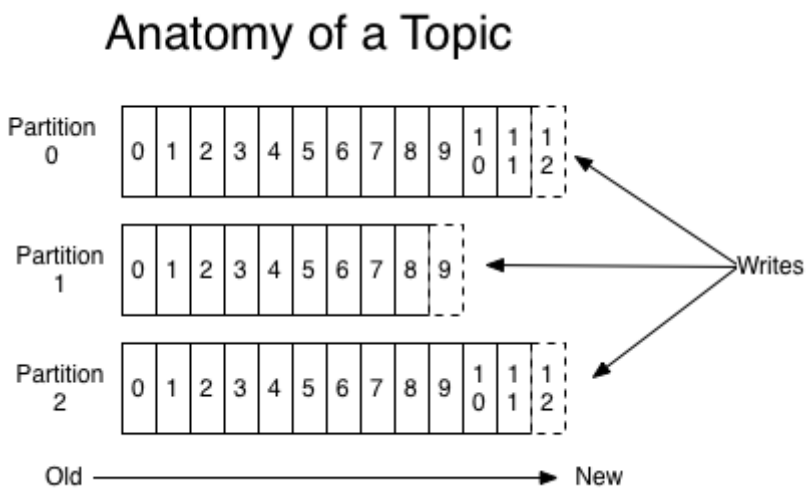
2、生产者（Producer） ：向broker发布消息的应用程序

3、AMQP服务端（broker）：用来接收生产者发送的消息并将这些消息路由给服务器中的队列，便于fafka将生产者发送的消息，动态的添加到磁盘并

给每一条消息一个偏移量，所以对于kafka一个broker就是一个应用程序的实例

4、复制因子：数据的备份数

5、偏移量(offset)： 来表示哪些消息是否被消费,偏移量唯一,消费后只是消费者的偏移量改变了,直到过期消息才会被清除(偏移量有很多,如下图） 来自 <http://orchome.com/5>



6 、key ： Kafka根据传递消息的key来进行分区的分配，即hash(key) % numPartitions ,这就保证了相同key的消息一定会被路由到相同的分区。如果你没有指定key,那Kafka几乎就是随机找一个分区发送无key的消息，然后把这个分区号加入到缓存中以备后面直接使用

7 、group ： 指消费群组,如果groupId一样(每个消费者都要指定groupId),说明是同一个群组,就变成了队列模式,如果每个groupId都不同,就成了发布订阅模式

8 、主题（Topic）：一个主题类似新闻中的体育、娱乐、教育等分类概念，在实际工程中通常一个业务一个主题。

9 、分区（Partition）：一个Topic中的消息数据按照多个分区组织，分区是kafka消息队列组织的最小单位，一个分区可以看作是一个,每个partition只能同一个group中的同一个consumer消费,一般情况下partition的数量大于等于broker的数量

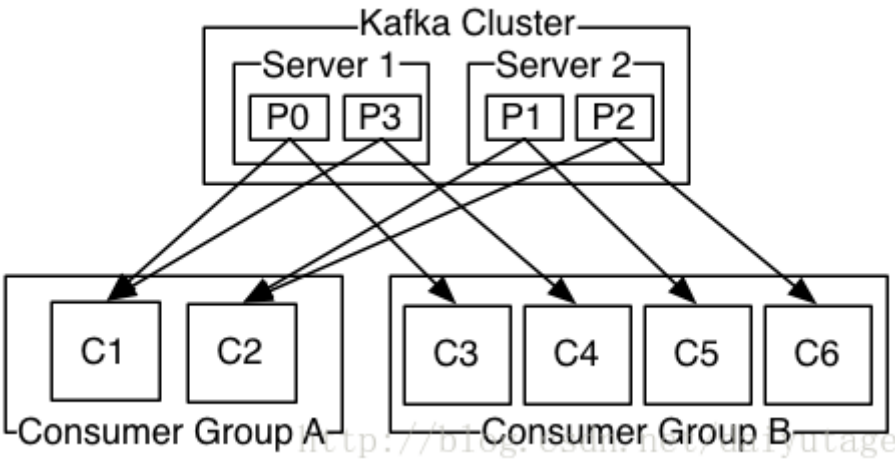
10、段(segment)：Partition包含多个segment，每个segment对应一个文件，segment可以手动指定大小，当segment达到阈值时，将不再写数据，每个segment都是大小相同的。segment由多个不可变的记录组成。记录只会被append

到segment中，不会被单独删除或修改，当某个segment上的消息条数达到配置值或消息发布时间超过阈值时，segment上的消息会被flush到磁盘，segment达到一定的大小后将不会再往该segment写数据，broker会创建新的segment。segment中的数据，默认保留7天数据。

11、 Message:

- 消息是Kafka通讯的基本单位，有一个固定长度的消息头和一个可变长度的消息体（payload）构成。在Java客户端中又称之为记录(Record)。

> 消息结构各部分说明如下：
> * CRC32：CRC32校验和，4个字节。
> * magic：Kafka服务程序协议版本号，用于做兼容。1个字节。
> * attributes：该字段占1字节，其中低两位用来表示压缩方式，第三位表示时间戳类型（0表示LogCreateTime，1表示LogAppendTime），高四位为预留位置，暂无实际意义。
> * timestamp：消息时间戳，当magic>0 时消息头必须包含该字段。8个字节。
> * key-length：消息key长度，4个字节。
> * key：消息key实际数据。
> * payload-length：消息实际数据长度，4个字节。
> * payload：消息实际数据
>
> 在实际存储一条消息还包括12字节的额外开销（LogOverhead）：
> * 消息的偏移量：8字节，类似于消息的Id。
> * 消息的总长度：4字节
>
> https://mp.weixin.qq.com/s/17b-uA4vxnU_39xXdM9ihQ



如上图Consumer Group A中的consumer-C2挂掉，consumer-C1会接收P1,P2，即一个consumer Group中有其他consumer挂掉后能够重新平衡，反正数据不会丢，这是因为消费组会有选举机制，由组协调器负责(GroupCoordinator)，每次

有消费者加入或退出消费组,都会触发rebalance,重新分配分区(具体怎么分可看分区分配策略文章)

一般消息系统，consumer存在两种消费模型：

- push：优势在于消息实时性高。劣势在于没有考虑consumer消费能力和饱和情况，容易导致producer压垮consumer。
- pull：优势在可以控制消费速度和消费数量，保证consumer不会出现饱和。劣势在于当没有数据，会出现空轮询，消耗cpu。

kafka采用pull，并采用可配置化参数保证当存在数据并且数据量达到一定量的时候，consumer端才进行pull操作，否则一直处于block状态

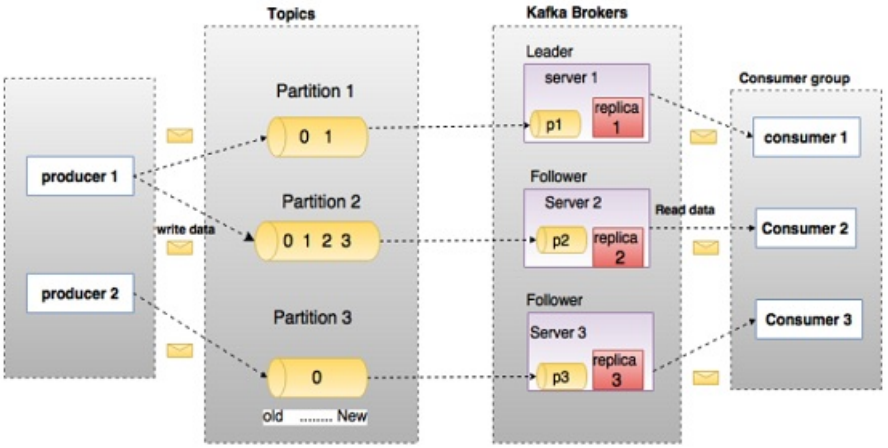
> 原文链接：https://blog.csdn.net/qq_29186199/article/details/80827085

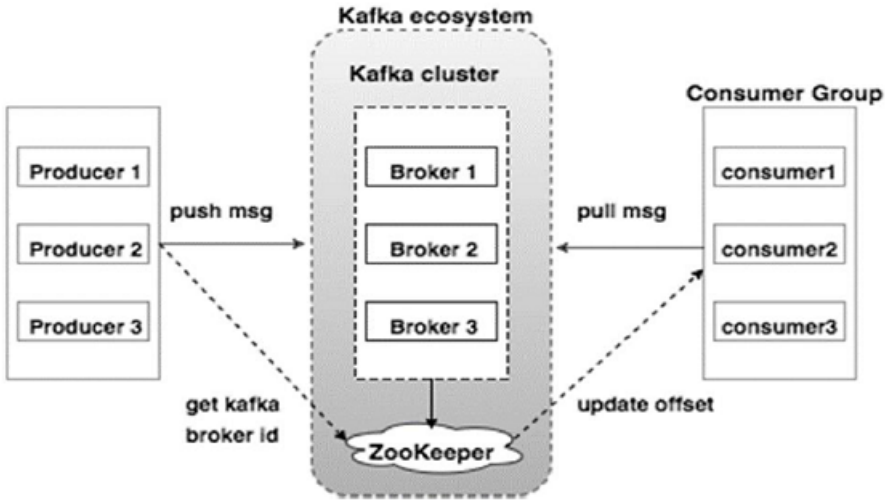
kafka支持的客户端语言：Kafka客户端支持当前大部分主流语言，包括：C、C++、Erlang、Java、.net、perl、PHP、Python、Ruby、Go、Javascript
可以使用以上任何一种语言和kafka服务器进行通信（即辨析自己的consumer从kafka集群订阅消息也可以自己写producer程序）

4. Kafka架构

生产者生产消息、kafka集群、消费者获取消息这样一种架构,
kafka集群中的消息，是通过Topic（主题）来进行组织的
主题（Topic）：一个主题类似新闻中的体育、娱乐、教育等分类概念，在实际工程中通常一个业务一个主题。
分区（Partition）：一个Topic中的消息数据按照多个分区组织，分区是kafka消息队列组织的最小单位，一个分区可以看作是一个FIFO（ First Input First Output的缩写，先入先出队列）的队列。
kafka分区是提高kafka性能的关键所在，当你发现你的集群性能不高时，常用手段就是增加Topic的分区，分区里面的消息是按照从新到老的顺序进行组织，消费者从队列头订阅消息，生产者从队列尾添加消息。
备份（Replication）：为了保证分布式可靠性，kafka0.8开始对每个分区的数据进行备份（不同的Broker上），防止其中一个Broker宕机造成分区上的数据不可用。

kafka0.7是一个很大的改变：1、增加了备份2、增加了控制借点概念，增加了集群领导者选举 。





5.Kafka原理

5.1 消费

5.1.1 消费组

什么是consumer group? 一言以蔽之，consumer group是kafka提供的可扩展且具有容错性的消费者机制。既然是一个组，那么组内必然可以有多个消费者或消费者实例(consumer instance)，它们共享一个公共的ID，即group ID。组内的所有消费者协调在一起来消费订阅主题(subscribed topics)的所有分区(partition)。当然，每个分区只能由同一个消费组内的一个consumer来消费。个人认为，理解consumer group记住下面这三个特性就好了：

- consumer group下可以有一个或多个consumer instance，consumer instance可以是一个进程，也可以是一个线程
- group.id是一个字符串，唯一标识一个consumer group

- consumer group下订阅的topic下的每个分区只能分配给某个group下的一个consumer(当然该分区还可以被分配给其他group)

5.1.2 coordinator

Kafka提供了一个角色：coordinator来执行对于consumer group的管理。坦率说kafka对于coordinator的设计与修改是一个很长的故事。最新版本的coordinator也与最初的设计有了很大的不同。这里我只想提及两次比较大的改变。

首先是0.8版本的coordinator，那时候的coordinator是依赖zookeeper来实现对于consumer group的管理的。Coordinator监听zookeeper的/consumers/<group>/ids的子节点变化以及/brokers/topics/<topic>数据变化来判断是否需要进行rebalance。group下的每个consumer都自己决定要消费哪些分区，并把自己的决定抢先在zookeeper中的/consumers/<group>/owners/<topic>/<partition>下注册。很明显，这种方案要依赖于zookeeper的帮助，而且每个consumer是单独做决定的，没有那种“大家属于一个组，要协商做事情”的精神。

基于这些潜在的弊端，0.9版本的kafka改进了coordinator的设计，提出了group coordinator——每个consumer group都会被分配一个这样的coordinator用于组管理和位移管理。这个group coordinator比原来承担了更多的责任，比如组成员管理、位移提交保护机制等。当新版本consumer group的第一个consumer启动的时候，它会去和kafka server确定谁是它们组的coordinator。之后该group内的所有成员都会和该coordinator进行协调通信。显而易见，这种coordinator设计不再需要zookeeper了，性能上可以得到很大的提升。后面的所有部分我们都将讨论最新版本的coordinator设计。

上面简单讨论了新版coordinator的设计，那么consumer group如何确定自己的coordinator是谁呢？ 简单来说分为两步：

- 确定consumer group位移信息写入__consumers_offsets的哪个分区。具体计算公式：
    - __consumers_offsets partition# = Math.abs(groupId.hashCode() % groupMetadataTopicPartitionCount) 注意：groupMetadataTopicPartitionCount由offsets.topic.num.partitions指定，默认是50个分区。
- 该分区leader所在的broker就是被选定的coordinator

5.1.3 Rebalance

rebalance本质上是一种协议，规定了一个consumer group下的所有consumer如何达成一致来分配订阅topic的每个分区。比如某个group下有20个consumer，它订阅了一个具有100个分区的topic。正常情况下，Kafka平均会为每个consumer分配5个分区。这个分配的过程就叫rebalance。

这也是经常被提及的一个问题。rebalance的触发条件有三种：

- 组成员发生变更(新consumer加入组、已有consumer主动离开组或已有consumer崩溃了——离开组是主动地发起rebalance；而崩溃则是被动地发起rebalance)
- 订阅主题数发生变更——这当然是可能的，如果你使用了正则表达式的方式进行订阅，那么新建匹配正则表达式的topic就会触发rebalance
- 订阅主题的分区数发生变更

Kafka新版本consumer默认提供了两种分配策略：range 和 round-robin。当然Kafka采用了可插拔式的分配策略，你可以创建自己的分配器以实现不同的分配策略。

rebalance分为2步：Join和Sync

1. Join， 顾名思义就是加入组。这一步中，所有成员都向coordinator发送JoinGroup请求，请求入组。一旦所有成员都发送了JoinGroup请求，coordinator会从中选择一个consumer担任leader的角色，并把组成员信息以及订阅信息发给leader——注意leader和coordinator不是一个概念。leader负责消费分配方案的制定。
2. Sync，这一步leader开始分配消费方案，即哪个consumer负责消费哪些topic的哪些partition。一旦完成分配，leader会将这个方案封装进SyncGroup请求中发给coordinator，非leader也会发SyncGroup请求，只是内容为空。coordinator接收到分配方案之后会把方案塞进SyncGroup的response中发给各个consumer。这样组内的所有成员就都知道自己应该消费哪些分区了。

consumer group的分区分配方案是在客户端执行的！Kafka将这个权利下放给客户端主要是因为这样做可以有更好的灵活性。比如这种机制下我可以实现类似于Hadoop那样的机架感知(rack-aware)分配方案，即为consumer挑选同一个机架下的分区数据，减少网络传输的开销

> 来自: https://www.cnblogs.com/songanwei/p/9202803.html

5.2 Data Replication（副本策略）

5.2.1 消息传递同步策略

Producer在发布消息到某个Partition时，先通过ZooKeeper找到该Partition的Leader，然后无论该Topic的Replication Factor为多少，Producer只将该消息发送

到该Partition的Leader。Leader会将该消息写入其本地Log。每个Follower都从Leader pull数据。这种方式上，Follower存储的数据顺序与Leader保持一致。Follower在收到该消息并写入其Log后，向Leader发送ACK。一旦Leader收到了ISR中的所有Replica的ACK，该消息就被认为已经commit了，Leader将增加HW并且向Producer发送ACK。

为了提高性能，每个Follower在接收到数据后就立马向Leader发送ACK，而非等到数据写入Log中。因此，对于已经commit的消息，Kafka只能保证它被存于多个Replica的内存中，而不能保证它们被持久化到磁盘中，也就不能完全保证异常发生后该条消息一定能被Consumer消费。

Consumer读消息也是从Leader读取，只有被commit过(所有同步副本（ISR 中所有副本）都保存了)的消息才会暴露给Consumer。

5.2.2 ACK前需要保证有多少个备份

对于Kafka而言，定义一个Broker是否“活着”包含两个条件：

- 一是它必须维护与ZooKeeper的session（这个通过ZooKeeper的Heartbeat机制来实现）。
- 二是Follower必须能够及时将Leader的消息复制过来，不能“落后太多”。

Leader会跟踪与其保持同步的Replica列表，该列表称为ISR（即in-sync Replica,由zookeeper维护）。如果一个Follower超过一定时间未向leader发送fetch请求，或者落后太多，Leader将把它从ISR中移除。

> 这里所描述的“落后太多”指:
> Follower复制的消息落后于Leader后的条数超过预定值（该值可在$KAFKA_HOME/config/server.properties中通过replica.lag.max.messages配置，其默认值是4000,offest值）　或者
> Follower超过一定时间（该值可在$KAFKA_HOME/config/server.properties中通过replica.lag.time.max.ms来配置，其默认值是10000）未向Leader发送fetch请求。

一个副本可以不同步Leader有如下几个原因:

- 慢副本：在一定周期时间内follower不能追赶上leader。最常见的原因之一是I / O瓶颈导致follower追加复制消息速度慢于从leader拉取速度。
- 卡住副本：在一定周期时间内follower停止从leader拉取请求。follower replica卡住了是由于GC暂停或follower失效或死亡。
- 新启动副本：当用户给主题增加副本因子时，新的follower不在同步副本列表中，直到他们完全赶上了leader日志。

一个partition的follower落后于leader足够多时，被认为不在同步副本列表或处于滞后状态。在Kafka-0.8.2.x中,副本滞后判断依据是副本落后于leader最大消息数量(replica.lag.max.messages)或replicas响应partition leader的最长等待时间(replica.lag.time.max.ms)。前者是用来检测缓慢的副本,而后者是用来检测失效或死亡的副本

> 原文链接：https://blog.csdn.net/lizhitao/article/details/51718185 (含图解)

Kafka的复制机制既不是完全的同步复制，也不是单纯的异步复制(一半一半,follower像leader pull消息,只要有一个follower复制成功了就行,所以才出现leader从ISR中移除不同步的副本)。

- 完全同步复制要求所有能工作的Follower都复制完，这条消息才会被认为commit，这种复制方式极大的影响了吞吐率（高吞吐率是Kafka非常重要的一个特性）。
- 异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，这种情况下如果Follower都复制完都落后于Leader，而如果Leader突然宕机，则会丢失数据。

而Kafka的这种使用ISR的方式则很好的均衡了确保数据不丢失以及吞吐率。Follower可以批量的从Leader复制数据，这样极大的提高复制性能（批量写磁盘），极大减少了Follower与Leader的差距。

需要说明的是，Kafka只解决fail/recover，不处理“Byzantine”（“拜占庭”）问题。一条消息只有被ISR里的所有Follower都从Leader复制过去才会被认为已提交。这样就避免了部分数据被写进了Leader，还没来得及被任何Follower复制就宕机了，而造成数据丢失（Consumer无法消费这些数据）。而对于Producer而言，它可以选择是否等待消息commit，这可以通过request.required.acks来设置。这种机制确保了只要ISR有一个或以上的Follower，一条被commit的消息就不会丢失。

5.2 生产

5.2.1 写入方式

producer 采用 push 模式将消息发布到 broker，每条消息都被 append 到 patition 中，属于顺序写磁盘（顺序写磁盘效率比随机写内存要高，保障 kafka 吞吐率,kafka快的原因之一）。

5.2.2 消息路由

producer 发送消息到 broker 时，会根据分区算法选择将其存储到哪一个 partition。其路由机制为：

1. 指定了 patition，则直接使用；
2. 未指定 patition 但指定 key，通过对 key 的 value 进行hash 选出一个 patition
3. patition 和 key 都未指定，使用轮询选出一个 patition。

5.3 broker

5.3.1 存储方式

物理上把 topic 分成一个或多个 patition（对应 server.properties 中的 num.partitions=3 配置），每个 patition 物理上对应一个文件夹（该文件夹存储该 patition 的所有消息和索引文件），如下：

```
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-0/
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-1/
drwxr-xr-x  2 root root 4096 Oct 10 01:54 demoTopic2-2/
-rw-r--r--  1 root root    0 Oct 10 00:48 .lock
-rw-r--r--  1 root root   54 Oct 10 00:48 meta.properties
-rw-r--r--  1 root root 1254 Oct 10 04:56 recovery-point-offset-checkpoint
-rw-r--r--  1 root root 1256 Oct 10 04:57 replication-offset-checkpoint
feng@ubuntu:/tmp/kafka/kafka-logs-1$ cd demoTopic2-0
feng@ubuntu:/tmp/kafka/kafka-logs-1/demoTopic2-0$ ll
total 12
drwxr-xr-x  2 root root     4096 Oct 10 01:54 ./
drwxr-xr-x 56 root root     4096 Oct 10 04:57 ../
-rw-r--r--  1 root root 10485760 Oct 10 01:54 00000000000000000000.index
-rw-r--r--  1 root root      188 Oct 10 02:28 00000000000000000000.log
feng@ubuntu:/tmp/kafka/kafka-logs-1/demoTopic2-0$
```

5.3.2 存储策略

无论消息是否被消费，kafka 都会保留所有消息。有两种策略可以删除旧数据：

1. 基于时间：log.retention.hours=168
2. 基于大小：log.retention.bytes=1073741824

5.4 高效速度

Kafka会把收到的消息都写入到硬盘中，它绝对不会丢失数据。为了优化写入速度Kafka主要采用了两个技术， 顺序写入 和 MMFile 。

5.4.1 写入数据

5.4.1.1 顺序写入磁盘

磁盘读写的快慢取决于你怎么使用它，也就是顺序读写或者随机读写。在顺序读写的情况下，某些优化场景磁盘的读写速度可以和内存持平（注：此处有疑问， 不推敲细节，参

考 http://searene.me/2017/07/09/Why-is-Kafka-so-fast/）。

因为硬盘是机械结构，每次读写都会寻址->写入，其中寻址是一个“机械动作”，它是最耗时的。所以硬盘最讨厌随机I/O，最喜欢顺序I/O。为了提高读写硬盘的速度，Kafka就是使用顺序I/O。

而且Linux对于磁盘的读写优化也比较多，包括read-ahead和write-behind，磁盘缓存等。如果在内存做这些操作的时候，一个是JAVA对象的内存开销很大，另一个是随着堆内存数据的增多，JAVA的GC时间会变得很长，使用磁盘操作有以下几个好处：

- 磁盘顺序读写速度超过内存随机读写
- JVM的GC效率低，内存占用大。使用磁盘可以避免这一问题
- 系统冷启动后，磁盘缓存依然可用

因此kafka写数据时在后面追加的(参考offset图解)

5.4.1.2 Memory Mapped Files(内存映射文件)

即便是顺序写入硬盘，硬盘的访问速度还是不可能追上内存。所以Kafka的数据并 不是实时的写入硬盘 ，它充分利用了现代操作系统 分页存储 来利用内存提高I/O效率。

Memory Mapped Files(后面简称mmap)也被翻译成 内存映射文件 ，在64位操作系统中一般可以表示20G的数据文件，它的工作原理是直接利用操作系统的Page来实现文件到物理内存的直接映射。完成映射之后你对物理内存的操作会被同步到硬盘上（操作系统在适当的时候）。通过mmap，进程像读写硬盘一样读写内存（当然是虚拟机内存），也不必关心内存的大小有虚拟内存为我们兜底。

使用这种方式可以获取很大的I/O提升， 省去了用户空间到内核空间 复制的开销（调用文件的read会把数据先放到内核空间的内存中，然后再复制到用户空间的内存中。）也有一个很明显的缺陷——不可靠， 写到mmap中的数据并没有被真正的写到硬盘，操作系统会在程序主动调用flush的时候才把数据真正的写到硬盘。 Kafka提供了一个参数——producer.type来控制是不是主动flush，如果Kafka写入到mmap之后就立即flush然后再返回Producer叫 同步 (sync)；写入mmap之后立即返回Producer不调用flush叫 异步 (async)。

5.4.2 读取数据

5.4.2.1 Zero Copy(零拷贝)

基于sendfile实现Zero Copy

传统模式下，当需要对一个文件进行传输的时候，其具体流程细节如下：

1. 调用read函数，文件数据被copy到内核缓冲区
2. read函数返回，文件数据从内核缓冲区copy到用户缓冲区
3. write函数调用，将文件数据从用户缓冲区copy到内核与socket相关的缓冲区。
4. 数据从socket缓冲区copy到相关协议引擎。

以上细节是传统read/write方式进行网络文件传输的方式，我们可以看到，在这个过程当中，文件数据实际上是经过了四次copy操作：

硬盘一>内核buf一>用户buf一>socket相关缓冲区一>协议引擎
而sendfile系统调用则提供了一种减少以上多次copy，提升文件传输性能的方法。
在内核版本2.1中，引入了sendfile系统调用，以简化网络上和两个本地文件之间的数据传输。
sendfile的引入不仅减少了数据复制，还减少了上下文切换。

```
sendfile(socket, file, len);
```

运行流程如下:

1. sendfile系统调用，文件数据被copy至内核缓冲区
2. 再从内核缓冲区copy至内核中socket相关的缓冲区
3. 最后再socket相关的缓冲区copy到协议引擎

相较传统read/write方式，2.1版本内核引进的sendfile已经减少了内核缓冲区到user缓冲区，再由user缓冲区到socket相关缓冲区的文件copy，而在内核版本2.4之后，文件描述符结果被改变，sendfile实现了更简单的方式，再次减少了一次copy操作。
在apache，nginx，lighttpd等web服务器当中，都有一项sendfile相关的配置，使用sendfile可以大幅提升文件传输性能。
Kafka把所有的消息都存放在一个一个的文件中，当消费者需要数据的时候Kafka直接把文件发送给消费者，配合mmap作为文件读写方式，直接把它传给sendfile。

5.4.2.2 批量压缩

在很多情况下，系统的瓶颈不是CPU或磁盘，而是网络IO，对于需要在广域网上的数据中心之间发送消息的数据流水线尤其如此。进行数据压缩会消耗少量的CPU资源,不过对于kafka而言，网络IO更应该需要考虑。

- 如果每个消息都压缩，但是压缩率相对很低，所以Kafka使用了批量压缩，即将多个消息一起压缩而不是单个消息压缩
- Kafka允许使用递归的消息集合，批量的消息可以通过压缩的形式传输并且在日志中也可以保持压缩格式，直到被消费者解压缩
- Kafka支持多种压缩协议，包括Gzip和Snappy压缩协议

> 来自 :https://www.cnblogs.com/binyue/p/10308754.html

6.Kafka集群搭建

1、软件环境

1、linux一台或多台，大于等于2
2、已经搭建好的zookeeper集群
3、软件版本kafka_2.11-0.10.1.0.tgz

2、创建目录并下载安装软件

tar -zxvf kafka_2.11-0.10.1.0.tgz

3、修改配置文件

进入到config目录主要关注：server.properties 这个文件即可，

修改配置文件：

#每台服务器的broker.id都不能相同
broker.id=0

#hostname
host.name=192.168.7.100

#在log.retention.hours=168 下面新增下面三项
message.max.byte=5242880
default.replication.factor=2
replica.fetch.max.bytes=5242880

#设置zookeeper的连接端口
zookeeper.connect=192.168.7.100:2181,192.168.7.101:2181,192.168.7.107:2181
# 消息存放的目录 ,记得创建文件夹
log.dirs=/usr/software/kafka_2.11-0.10.2.1/logs

下面是参数的解释
broker.id=0 #当前机器在集群中的唯一标识，和zookeeper的myid性质一样
port=19092 #当前kafka对外提供服务的端口默认是9092
host.name=192.168.7.100 #这个参数默认是关闭的，在0.8.1有个bug，DNS解析问题，失败率的问题。
num.network.threads=3 #这个是borker进行网络处理的线程数
num.io.threads=8 #这个是borker进行I/O处理的线程数
log.dirs=/opt/kafka/kafkalogs/ #消息存放的目录，这个目录可以配置为“，”逗号分割的表达式，上面的num.io.threads要大于这个目录的个数这个目录，如果配置多个目录，新创建的topic他把消息持久化的地方是，当前以逗号分割的目录中，那个分区数最少就放那一个
socket.send.buffer.bytes=102400 #发送缓冲区buffer大小，数据不是一下子就发送的，先回存储到缓冲区了到达一定的大小后在发送，能提高性能
socket.receive.buffer.bytes=102400 #kafka接收缓冲区大小，当数据到达一定大小后在序列化到磁盘
socket.request.max.bytes=104857600 #这个参数是向kafka请求消息或者向kafka发送消息的请请求的最大数，这个值不能超过java的堆栈大小
num.partitions=1 #默认的分区数，一个topic默认1个分区数
log.retention.hours=168 #默认消息的最大持久化时间，168小时，7天
message.max.byte=5242880 #消息保存的最大值5M
default.replication.factor=2 #kafka保存消息的副本数，如果一个副本失效了，另一个还可以继续提供服务
replica.fetch.max.bytes=5242880 #取消息的最大直接数

log.segment.bytes=1073741824 #这个参数是：因为kafka的消息是以追加的形式落地到文件，当超过这个值的时候，kafka会新起一个文件
log.retention.check.interval.ms=300000 #每隔300000毫秒去检查上面配置的log失效时间（log.retention.hours=168 ），到目录查看是否有过期的消息如果有，删除
log.cleaner.enable=false #是否启用log压缩，一般不用启用，启用的话可以提高性能
zookeeper.connect=192.168.7.100:2181,192.168.7.101:2181,192.168.7.107:1218 #设置zookeeper的连接端口

zookeeper.properties,修改/增加日志文件路径(记得创建文件夹)

dataDir=/usr/software/kafka_2.11-0.10.2.1/zookeeper/dataDir
dataLogDir=/usr/software/kafka_2.11-0.10.2.1/zookeeper/dataLogDir

4、启动Kafka集群并测试

```
# 每台都要启动,需先启动zookeeper
 nohup sh kafka-server-start.sh -daemon  ../config/server.properties >/dev/null &
```

> kafka-server-stop.sh 脚本好像不能正确关闭kafka,需要改动命令,如下

```
PIDS=$(jps -lm | grep -i 'kafka.Kafka'| awk '{print $1}')
```

> 关闭kafka后,用jps查看好像会延时,等待一会查看