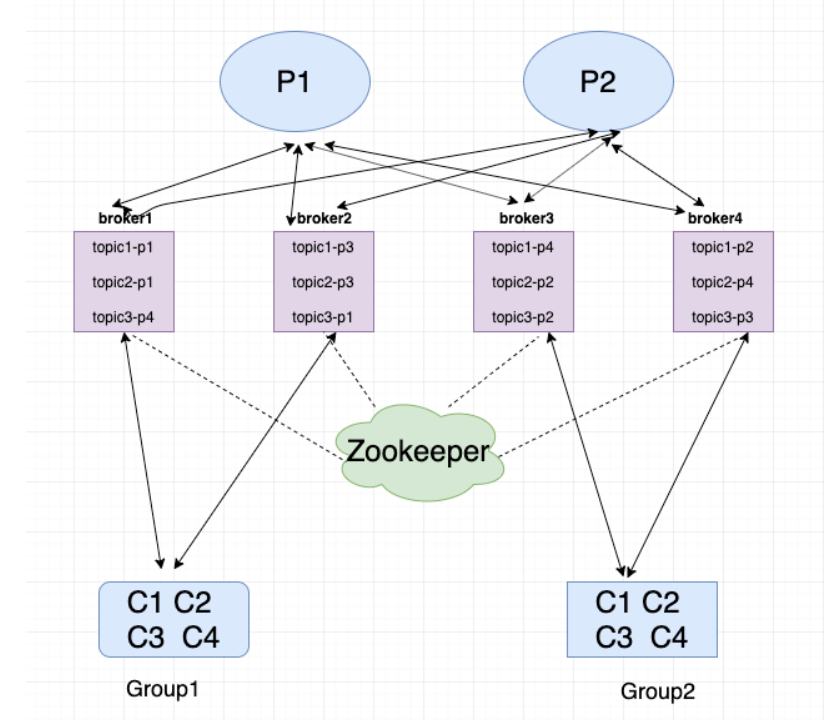
# 博客一

## 消息队列的作用

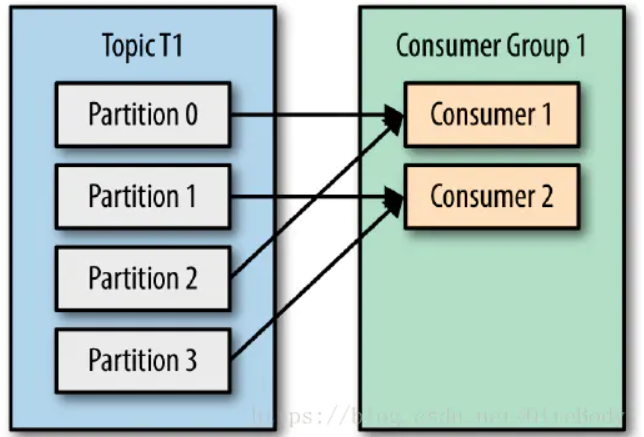
**解耦，削峰，异步，顺序性（在一定程度上保证）。**

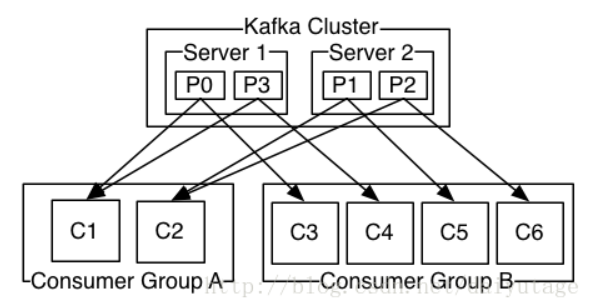
解偶  
快递员可以选择自己的时间，把快递放到柜子里，不需要关心小明是否在家。  
小明也不需要一直等待给快递员开门，两个人解耦了。  
异步  
快递员把快递放到柜子里发个信息就可以去送下一件，不需同步等待结果。  
削峰  
到了双十一，小明一天要到100个快递，由于小明一天只能消化10个快递，剩下的就放在了柜子里，等10天后才拿完。



## kafka如何支撑多线程消费?

设置一个组，这个组消费某个主题的消息，组内设置和分区数一样多的消费者，每个消费者消费一个分区。分区数决定了同组消费者个数的上限。如果你的分区数是N，那么最好线程数也保持为N，这样通常能够达到最大的吞吐量。超过N的配置只是浪费系统资源，因为多出的线程不会被分配到任何分区。





## kafka 会不会出现不按顺序消费的情况？

**乱序场景一**  
因为一个topic可以有多个partition，kafka只能保证partition内部有序，partition数量=同一个消费者组中消费者数量时，可能需要顺序的数据分布到了不同的partition，导致处理时乱序

解决方案

1、可以设置topic 有且只有一个partition。  
2、同一topic下，再分为各需要顺序消费的业务小类，每个小类消息指定到同一个partition。

**乱序场景二**  
对于同一业务进入了一个消费者组之后，用了多线程来处理消息，会导致消息的乱序  
解决方案

消费者内部保持与线程等量的内存队列，相同的业务数据，放到同一个内存队列中，然后线程从对应的内存队列中取出并操作。

组并发：一个组可以并发消费多个分区。

分区负载均衡：一个分区可以被多个组消费。

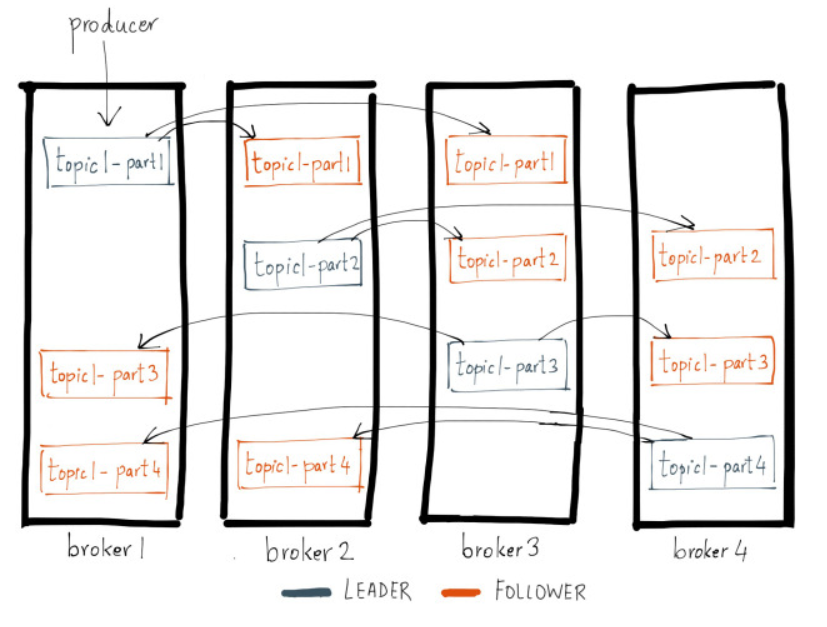
组内分工明确：组内消费者只能消费不同的分区（比如一个组关注多个topic，每个topic是多个分区，那么组内消费者每个消费者都可以消费其中0个或多个分区，但是同一分区只能一个消费者来消费）。

一个消息对一个组，只会发一次。（对多个组可能发多次，注意重复消费的问题）

**broker到消费者采用pull模式：**  
push模式的目标是尽可能快地传递消息，这样很容易造成消费者来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式可以根据consumer的消费能力以适当的速率消费消息。  
pull模式缺点在于，如果broker没有数据，消费者会轮询，忙等待数据直到数据到达，为了避免这种情况，我们允许消费者在pull请求时候使用“long poll”进行阻塞，直到数据到达 。

一个组对每条休息只能消费一次。

## Leader和Follower分区



 producer 只能往 leader 分区上写数据，消费也只能从leader分区上读，followers 只按顺序从 leader 上复制日志。

## **幂等性发送**

为实现Producer的幂等性，Kafka引入了Producer ID（即PID）和Sequence Number。对于每个PID，该**Producer**发送消息的每个<Topic, Partition>都对应一个单调递增的Sequence Number。

同样，Broker端也会为每个<PID, Topic, Partition>维护一个序号，并且每Commit一条消息时将其对应序号递增。对于接收的每条消息，如果其序号比Broker维护的序号大1，则Broker会接受它，否则将其丢弃。

如果消息序号比Broker维护的序号差值比一大，说明中间有数据尚未写入，即乱序，此时Broker拒绝该消息，Producer抛出InvalidSequenceNumber  
如果消息序号小于等于Broker维护的序号，说明该消息已被保存，即为重复消息，Broker直接丢弃该消息，Producer抛出DuplicateSequenceNumber  
Sender发送失败后会重试，这样可以保证每个消息都被发送到broker。

## **生产者**

**生产者发送消息的三种方式：**

1. producer.send(record);//Fire-and-forget异步发送，不管发送结果。

2. RecordMetadata result = producer.send(record).get(); //同步发送。

3. producer.send(record,new MyProducerCallBack());//带回调的异步发送

一定需要处理callback的成功或失败逻辑：

private static class MyProducerCallBack implements Callback{

@Override

public void onCompletion(RecordMetadata recordMetadata, Exception e) {

if(null != e){

e.printStackTrace();

return;

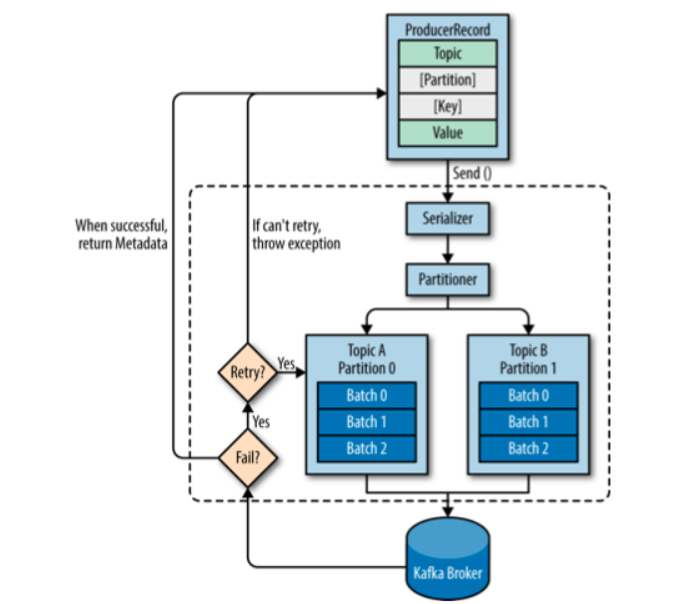
}

System.out.println("时间戳，主题，分区，位移: " + recordMetadata.timestamp() + ", " + recordMetadata.topic() + "," + recordMetadata.partition() + " " + recordMetadata.offset());

}

}

如果ProducerRecord中指定了Partition，则Partitioner不做任何事情；否则，Partitioner根据消息的key得到一个Partition。这是生产者就知道向哪个Topic下的哪个Partition发送这条消息。



## kafka消费如何保证顺序性

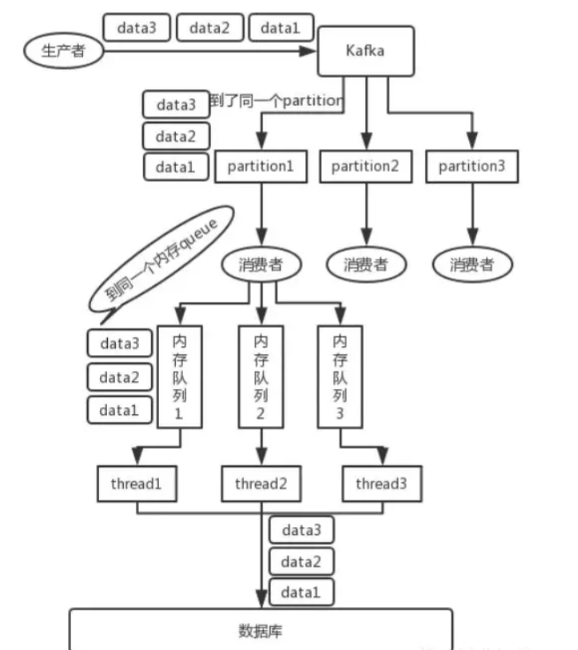
一个 topic，一个 partition，一个 consumer，内部单线程消费，这样的状态数据消费是有序的。但由于单线程吞吐量太低，在数据庞大的实际场景很少采用。

1、生产者在写的时候，可以指定一个 key，比如说我们指定了某个订单 id 作为 key，那么这个订单相关的数据，一定会被分发到同一个 partition 中去，而且这个 partition 中的数据一定是有顺序的。（需要保证顺序的几条消息，发送到同一个分区。局部有序）

2、消费者从 partition 中取出来数据的时候，也一定是有顺序的。

3、但是消费者里可能会有多个线程来并发来处理消息。因为如果消费者是单线程消费数据，那么这个吞吐量太低了。而多个线程并发的话，顺序可能就乱掉了。

4、写N个queue，将具有相同key的数据都存储在同一个queue，然后对于N个线程，每个线程分别消费一个queue即可。



producer发送消息时具体到topic的哪一个partition分区，提供了三种方式  
1）指定分区  
2）不指定分区，有指定key 则根据key的hash值与分区数进行运算后确定发送到哪个partition分区  
3）不指定分区，不指定key，则轮询各分区发送

# 博客二

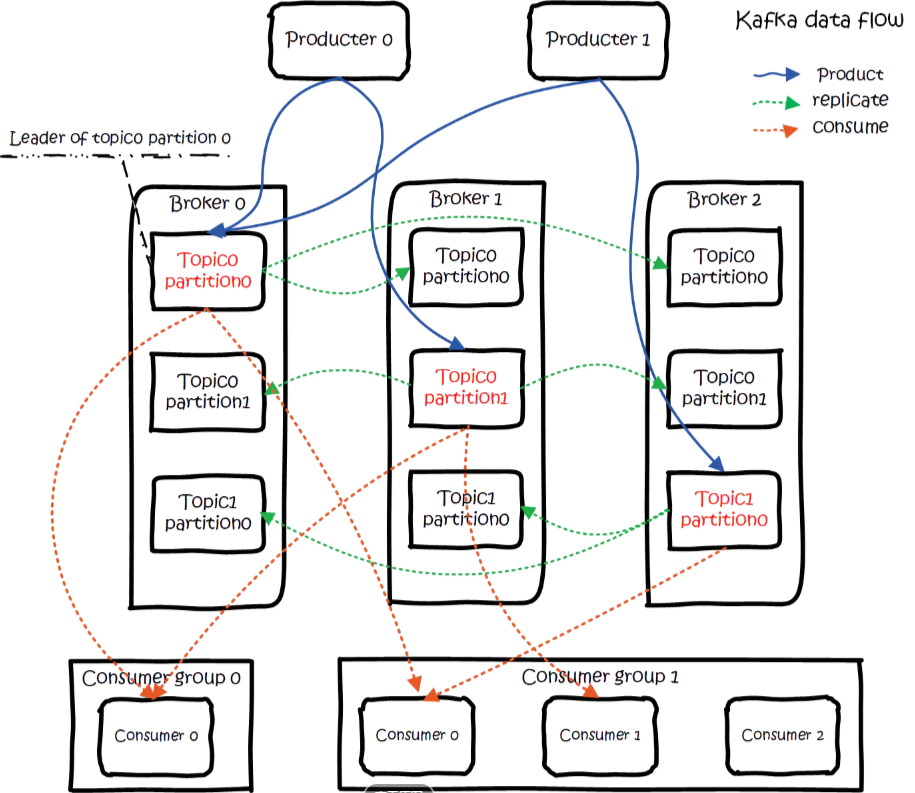
## 简介

kafka是一个分布式消息队列。具有高性能、持久化、多副本备份、横向扩展能力。生产者往队列里写消息，消费者从队列里取消息进行业务逻辑。一般在架构设计中起到解耦、削峰、异步处理的作用。

kafka对外使用topic的概念，生产者往topic里写消息，消费者从topic读消息。为了做到水平扩展，一个topic实际是由多个partition(分区)组成的，遇到瓶颈时，可以通过增加partition的数量来进行横向扩容。单个parition内是保证消息有序。

每新写一条消息，kafka就是在对应的文件append写，所以性能非常高。

kafka的总体数据流是这样的：



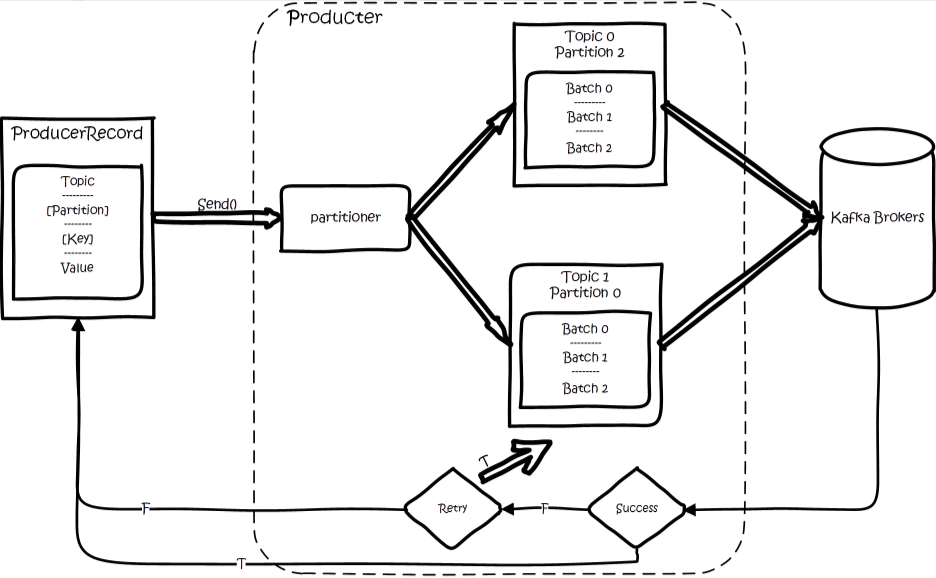
大概用法就是，Producers往Brokers里面的指定Topic中写消息，Consumers从Brokers里面拉去指定Topic的消息，然后进行业务处理。  
图中有两个topic，topic 0有两个partition，topic 1有一个partition，三副本备份。可以看到consumer gourp 1中的consumer 2没有分到partition处理，这是有可能出现的，下面会讲到。

producters生产的详细针对的是broker里的topic，topic接收到消息会同步到其它的两套副本里。消费者组中的消费者，去订阅topic，取的为什么都是接收broker中的topic，没去副本topic中取呢,？这样效率不低么?

关于broker、topics、partitions的一些元信息用zk来存，监控和路由等也都会用到zk。

## 生产

基本流程是这样的：



创建一条记录，记录中一个要指定对应的topic和value，key和partition可选。 先序列化，然后按照topic和partition，放进对应的发送队列中。kafka produce都是批量请求，会积攒一批，然后一起发送，不是调send()就进行立刻进行网络发包。  
如果partition没填，那么情况会是这样的：

* key有填：  
  按照key进行哈希，相同key去一个partition。（如果扩展了partition的数量那么就不能保证了）
* key没填：  
  round-robin来选partition

这些要发往同一个partition的请求按照配置，攒一波，然后由一个单独的线程一次性发过去。

### API

有high level api，替我们把很多事情都干了，offset，路由等都替我们干了，用起也来很简单。  
还有simple api，offset等都是要我们自己记录。

### partition

当存在多副本的情况下，会尽量把多个副本，分配到不同的broker上。**kafka会为partition选出一个leader，之后所有该partition的请求，实际操作的都是leader，然后再同步到其他的follower。**当一个broker歇菜后，所有leader在该broker上的partition都会重新选举，选出一个leader。（这里不像分布式文件存储系统那样会自动进行复制保持副本数）

然后这里就涉及两个细节：怎么分配partition，怎么选leader。

**关于partition的分配，还有leader的选举，总得有个执行者。在kafka中，这个执行者就叫controller。**kafka使用zk在broker中选出一个controller，用于partition分配和leader选举。

#### partition的分配

1. 将所有Broker（假设共n个Broker）和待分配的Partition排序
2. 将第i个Partition分配到第（i mod n）个Broker上 （这个就是leader）
3. 将第i个Partition的第j个Replica(副本)分配到第（(i + j) mode n）个Broker上

#### leader容灾

controller会在Zookeeper的/brokers/ids节点上注册Watch，一旦有broker宕机，它就能知道。当broker宕机后，controller就会给受到影响的partition选出新leader。

controller从zk的/brokers/topics/[topic]/partitions/[partition]/state中，读取对应partition的ISR（in-sync replica已同步的副本）列表，选一个出来做leader。  
选出leader后，更新zk，然后发送LeaderAndISRRequest给受影响的broker，让它们改变知道这事。为什么这里不是使用zk通知，而是直接给broker发送rpc请求，我的理解可能是这样做zk有性能问题吧。

如果ISR列表是空，那么会根据配置，随便选一个replica做leader，或者干脆这个partition就是歇菜。

如果ISR列表的有机器，但是也歇菜了，那么还可以等ISR的机器活过来。

#### 多副本同步

这里的策略，服务端这边的处理是follower从leader批量拉取数据来同步。

但是具体的可靠性，是由生产者来决定的。  
生产者生产消息的时候，通过request.required.acks参数来设置数据的可靠性。

| **acks** | **what happen** |
| --- | --- |
| 0 | which means that the producer never waits for an acknowledgement from the broker.  发过去就完事了，不关心broker是否处理成功，可能丢数据。 |
| 1 | which means that the producer gets an acknowledgement after the leader replica has received the data.  当写Leader成功后就返回,其他的replica都是通过fetcher去同步的,所以kafka是异步写，主备切换可能丢数据。 |
| -1 | which means that the producer gets an acknowledgement after all in-sync replicas have received the data.  要等到isr里所有机器同步成功，才能返回成功，延时取决于最慢的机器。强一致，不会丢数据。 |

在acks=-1的时候，如果ISR少于min.insync.replicas指定的数目，那么就会返回不可用。

这里ISR列表中的机器是会变化的，根据配置replica.lag.time.max.ms，多久没同步，就会从ISR列表中剔除。以前还有根据落后多少条消息就踢出ISR，在1.0版本后就去掉了，因为这个值很难取，在高峰的时候很容易出现节点不断的进出ISR列表。

从ISA中选出leader后，follower会从把自己日志中上一个高水位后面的记录去掉，然后去和leader拿新的数据。因为新的leader选出来后，follower上面的数据，可能比新leader多，所以要截取。

这里高水位的意思，对于partition和leader，就是所有ISR中都有的最新一条记录。消费者最多只能读到高水位；

从leader的角度来说高水位的更新会延迟一轮，例如写入了一条新消息，ISR中的broker都fetch到了，但是ISR中的broker只有在下一轮的fetch中才能告诉leader。

也正是由于这个高水位延迟一轮，在一些情况下，kafka会出现丢数据和主备数据不一致的情况，0.11开始，使用leader epoch来代替高水位。

（https://cwiki.apache.org/confluence/display/KAFKA/KIP-101+-+Alter+Replication+Protocol+to+use+Leader+Epoch+rather+than+High+Watermark+for+Truncation#KIP-101-AlterReplicationProtocoltouseLeaderEpochratherthanHighWatermarkforTruncation-Scenario1:HighWatermarkTruncationfollowedbyImmediateLeaderElection）

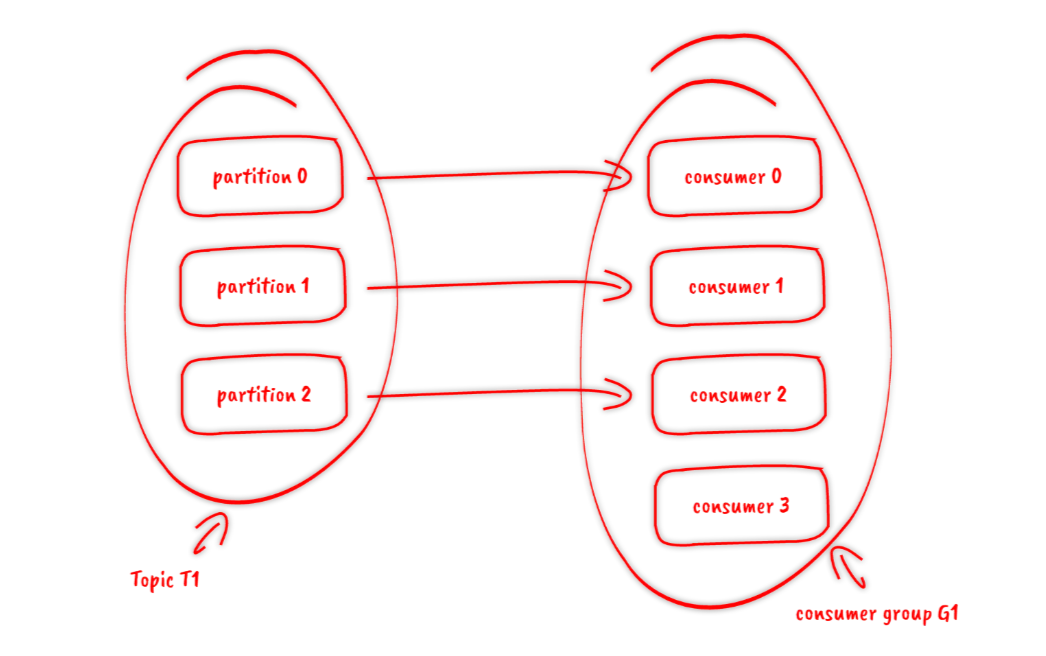
**思考：**  
当acks=-1时

是follwers都来fetch就返回成功，还是等follwers第二轮fetch？

leader已经写入本地，但是ISR中有些机器失败，那么怎么处理呢？

## 消费

订阅topic是以一个消费组来订阅的，一个消费组里面可以有多个消费者。同一个消费组中的两个消费者，不会同时消费一个partition。换句话来说，**就是一个partition，只能被消费组里的一个消费者消费**，但是可以同时被多个消费组消费。因此，如果消费组内的消费者如果比partition多的话，那么就会有个别消费者一直空闲。



### API

订阅topic时，可以用正则表达式，如果有新topic匹配上，那能自动订阅上。

### offset的保存

一个消费组消费partition，需要保存offset记录消费到哪，以前保存在zk中，由于zk的写性能不好，以前的解决方法都是consumer每隔一分钟上报一次。这里zk的性能严重影响了消费的速度，而且很容易出现重复消费。  
在0.10版本后，kafka把这个offset的保存，从zk总剥离，保存在一个名叫

\_\_consumeroffsets topic的topic中。写进消息的key由groupid、topic、partition组成，value是偏移量offset。topic配置的清理策略是compact。总是保留最新的key，其余删掉。

一般情况下，每个key的offset都是缓存在内存中，查询的时候不用遍历partition，如果没有缓存，第一次就会遍历partition建立缓存，然后查询返回。

确定consumer group位移信息写入\_\_consumers\_offsets的哪个partition，具体计算公式：

**\_\_consumers\_offsets partition =Math.abs(groupId.hashCode() % groupMetadataTopicPartitionCount)**

//groupMetadataTopicPartitionCount由offsets.topic.num.partitions指定，默认是50个分区。

**思考：**  
如果正在跑的服务，修改了offsets.topic.num.partitions，那么offset的保存是不是就乱套了？

### 分配partition--reblance

生产过程中broker要分配partition，**消费过程这里，也要分配partition给消费者。类似broker中选了一个controller出来，消费也要从broker中选一个coordinator，用于分配partition。**  
下面从顶向下，分别阐述一下

1. 怎么选coordinator。
2. 交互流程。
3. reblance的流程。

#### 选coordinator

1. 看offset保存在那个partition
2. 该partition leader所在的broker就是被选定的coordinator

这里我们可以看到，consumer group的coordinator，和保存consumer group offset的partition leader是同一台机器。

#### 交互流程

把coordinator选出来之后，就是要分配了  
整个流程是这样的：

1. consumer启动、或者coordinator宕机了，consumer会任意请求一个broker，发送ConsumerMetadataRequest请求，broker会按照上面说的方法，选出这个consumer对应coordinator的地址。
2. consumer 发送heartbeat请求给coordinator，返回IllegalGeneration的话，就说明consumer的信息是旧的了，需要重新加入进来，进行reblance。返回成功，那么consumer就从上次分配的partition中继续执行。

#### reblance流程

1. consumer给coordinator发送JoinGroupRequest请求。
2. 这时其他consumer发heartbeat请求过来时，coordinator会告诉他们，要reblance了。
3. 其他consumer发送JoinGroupRequest请求。
4. 所有记录在册的consumer都发了JoinGroupRequest请求之后，coordinator就会在这里consumer中随便选一个leader。然后回JoinGroupRespone，这会告诉consumer你是follower还是leader，对于leader，还会把follower的信息带给它，让它根据这些信息去分配partition
5. consumer向coordinator发送SyncGroupRequest，其中leader的SyncGroupRequest会包含分配的情况。
6. coordinator回包，把分配的情况告诉consumer，包括leader。

当partition或者消费者的数量发生变化时，都得进行reblance。  
列举一下会reblance的情况：

1. 增加partition
2. 增加消费者
3. 消费者主动关闭
4. 消费者宕机了
5. coordinator自己也宕机了

## 消息投递语义

kafka支持3种消息投递语义  
At most once：最多一次，消息可能会丢失，但不会重复  
At least once：最少一次，消息不会丢失，可能会重复  
Exactly once：只且一次，消息不丢失不重复，只且消费一次（0.11中实现，仅限于下游也是kafka）

在业务中，常常都是使用At least once的模型，如果需要可重入的话，往往是业务自己实现。

### At least once

先获取数据，再进行业务处理，业务处理成功后commit offset。  
1、生产者生产消息异常，消息是否成功写入不确定，重做，可能写入重复的消息  
2、消费者处理消息，业务处理成功后，更新offset失败，消费者重启的话，会重复消费

### At most once

先获取数据，再commit offset，最后进行业务处理。  
1、生产者生产消息异常，不管，生产下一个消息，消息就丢了  
2、消费者处理消息，先更新offset，再做业务处理，做业务处理失败，消费者重启，消息就丢了

### Exactly once

思路是这样的，首先要保证消息不丢，再去保证不重复。所以盯着At least once的原因来搞。 首先想出来的：

1.生产者重做导致重复写入消息----生产保证幂等性

2.消费者重复消费---消灭重复消费，或者业务接口保证幂等性重复消费也没问题

**由于业务接口是否幂等，不是kafka能保证的，所以kafka这里提供的exactly once是有限制的，消费者的下游也必须是kafka。**所以一下讨论的，没特殊说明，消费者的下游系统都是kafka（注:使用kafka conector，它对部分系统做了适配，实现了exactly once）。

生产者幂等性好做，没啥问题。

解决重复消费有两个方法：

1.下游系统保证幂等性，重复消费也不会导致多条记录。

2.把commit offset和业务处理绑定成一个事务。

本来exactly once实现第1点就ok了。

但是在一些使用场景下，我们的数据源可能是多个topic，处理后输出到多个topic，这时我们会希望输出时要么全部成功，要么全部失败。**这就需要实现事务性。**既然要做事务，那么干脆**把重复消费的问题从根源上解决，把commit offset和输出到其他topic绑定成一个事务。**

#### 生产幂等性

思路是这样的，为每个producer分配一个pid，作为该producer的唯一标识。

producer会为每一个<topic,partition>维护一个单调递增的seq。

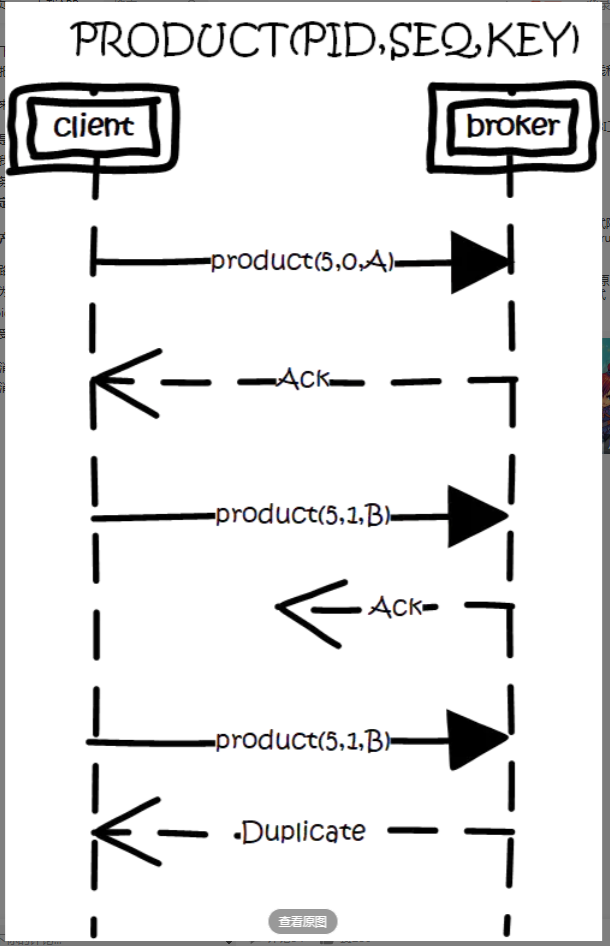
类似的，broker也会为每个<pid,topic,partition>记录下最新的seq。

当req\_seq == broker\_seq+1时，broker才会接受该消息。

因为：

1.消息的seq比broker的seq大超过时，说明中间有数据还没写入，即乱序了。

2.消息的seq不比broker的seq小，那么说明该消息已被保存。



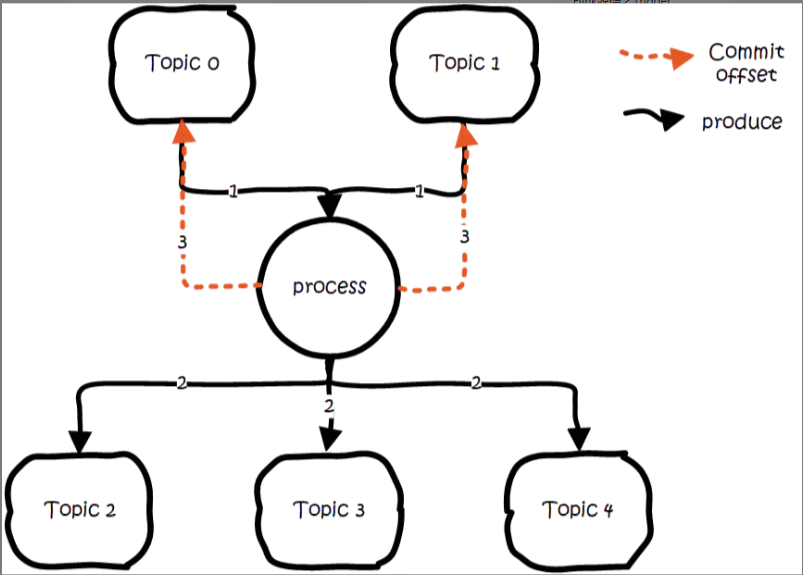
解决重复生产

#### 事务性/原子性广播

场景是这样的：

1. 先从多个源topic中获取数据。
2. 做业务处理，写到下游的多个目的topic。
3. 更新多个源topic的offset。

其中第2、3点作为一个事务，要么全成功，要么全失败。这里得益与offset实际上是用特殊的topic去保存，这两点都归一为写多个topic的事务性处理。

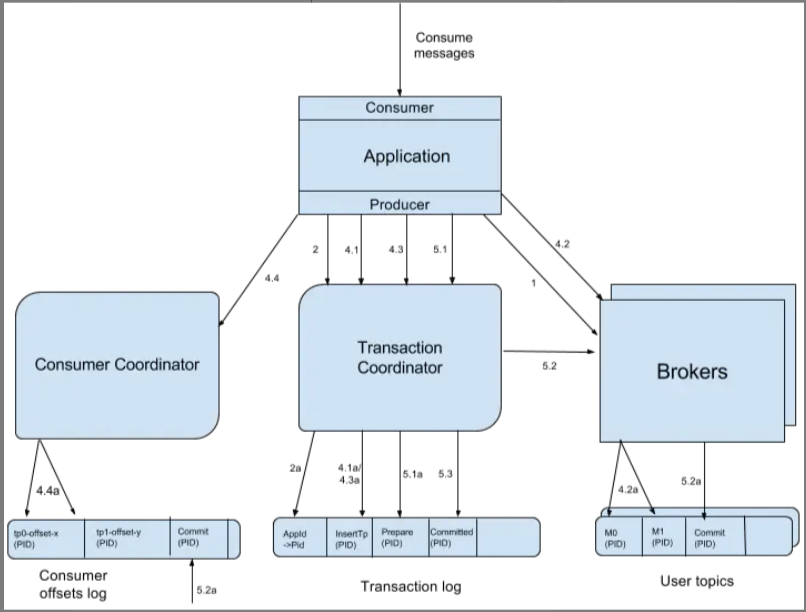


基本思路是这样的：  
引入tid（transaction id），和pid不同，这个id是应用程序提供的，用于标识事务，和producer是谁并没关系。就是任何producer都可以使用这个tid去做事务，这样进行到一半就死掉的事务，可以由另一个producer去恢复。  
同时为了记录事务的状态，类似对offset的处理，引入transaction coordinator用于记录transaction log。在集群中会有多个transaction coordinator，每个tid对应唯一一个transaction coordinator。  
注：transaction log删除策略是compact，已完成的事务会标记成null，compact后不保留。

做事务时，先标记开启事务，写入数据，全部成功就在transaction log中记录为prepare commit状态，否则写入prepare abort的状态。之后再去给每个相关的partition写入一条marker（commit或者abort）消息，标记这个事务的message可以被读取或已经废弃。

成功后在transaction log记录下commit/abort状态，至此事务结束。

数据流：



Kafka Transactions Data Flow

1. 首先使用tid请求任意一个broker（代码中写的是负载最小的broker），找到对应的transaction coordinator。
2. 请求transaction coordinator获取到对应的pid，和pid对应的epoch，这个epoch用于防止僵死进程复活导致消息错乱，当消息的epoch比当前维护的epoch小时，拒绝掉。tid和pid有一一对应的关系，这样对于同一个tid会返回相同的pid。
3. client先请求transaction coordinator记录<topic,partition>的事务状态，初始状态是BEGIN，如果是该事务中第一个到达的<topic,partition>，同时会对事务进行计时；client输出数据到相关的partition中；client再请求transaction coordinator记录offset的<topic,partition>事务状态；client发送offset commit到对应offset partition。
4. client发送commit请求，transaction coordinator记录prepare commit/abort，然后发送marker给相关的partition。全部成功后，记录commit/abort的状态，最后这个记录不需要等待其他replica的ack，因为prepare不丢就能保证最终的正确性了。

这里prepare的状态主要是用于事务恢复，例如给相关的partition发送控制消息，没发完就宕机了，备机起来后，producer发送请求获取pid时，会把未完成的事务接着完成。

当partition中写入commit的marker后，相关的消息就可被读取。所以kafka事务在prepare commit到commit这个时间段内，消息是逐渐可见的，而不是同一时刻可见。

详细细节可看：

https://cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging#KIP-98-ExactlyOnceDeliveryandTransactionalMessaging-TransactionalGuarantees

#### 消费事务

前面都是从生产的角度看待事务。还需要从消费的角度去考虑一些问题。  
消费时，partition中会存在一些消息处于未commit状态，即业务方应该看不到的消息，需要过滤这些消息不让业务看到，kafka选择在消费者进程中进行过来，而不是在broker中过滤，主要考虑的还是性能。kafka高性能的一个关键点是zero copy，如果需要在broker中过滤，那么势必需要读取消息内容到内存，就会失去zero copy的特性。

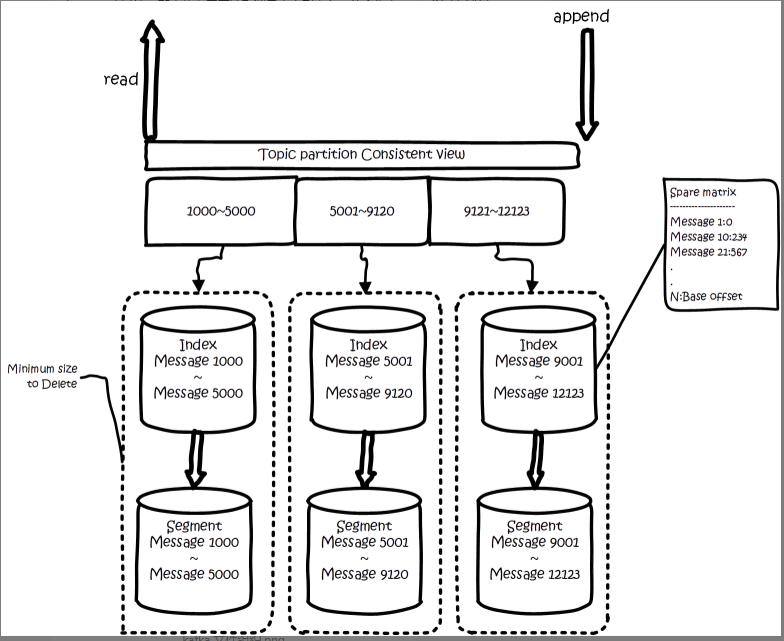
## 文件组织

kafka的数据，实际上是以文件的形式存储在文件系统的。topic下有partition，partition下有segment，segment是实际的一个个文件，topic和partition都是抽象概念。

在目录/${topicName}-{$partitionid}/下，存储着实际的log文件（即segment），还有对应的索引文件。

每个segment文件大小相等，文件名以这个segment中最小的offset命名，文件扩展名是.log；segment对应的索引的文件名字一样，扩展名是.index。有两个index文件，一个是offset index用于按offset去查message，一个是time index用于按照时间去查，其实这里可以优化合到一起，下面只说offset index。

总体的组织是这样的：



为了减少索引文件的大小，降低空间使用，方便直接加载进内存中，这里的索引使用稀疏矩阵，不会每一个message都记录下具体位置，而是每隔一定的字节数，再建立一条索引。 索引包含两部分，分别是baseOffset，还有position。

baseOffset：意思是这条索引对应segment文件中的第几条message。这样做方便使用数值压缩算法来节省空间。例如kafka使用的是varint。

position：在segment中的绝对位置。

查找offset对应的记录时，会先用二分法，找出对应的offset在哪个segment中，然后使用索引，在定位出offset在segment中的大概位置，再遍历查找message。

## 常用配置项

### broker配置

| **配置项** | **作用** |
| --- | --- |
| broker.id | broker的唯一标识 |
| auto.create.topics.auto | 设置成true，就是遇到没有的topic自动创建topic。 |
| log.dirs | log的目录数，目录里面放partition，当生成新的partition时，会挑目录里partition数最少的目录放。 |

### topic配置

| **配置项** | **作用** |
| --- | --- |
| num.partitions | 新建一个topic，会有几个partition。 |
| log.retention.ms | 对应的还有minutes，hours的单位。日志保留时间，因为删除是文件维度而不是消息维度，看的是日志文件的mtime。 |
| log.retention.bytes | partion最大的容量，超过就清理老的。注意这个是partion维度，就是说如果你的topic有8个partition，配置1G，那么平均分配下，topic理论最大值8G。 |
| log.segment.bytes | 一个segment的大小。超过了就滚动。 |
| log.segment.ms | 一个segment的打开时间，超过了就滚动。 |
| message.max.bytes | message最大多大 |

关于日志清理，默认当前正在写的日志，是怎么也不会清理掉的。  
还有0.10之前的版本，时间看的是日志文件的mtime，但这个指是不准确的，有可能文件被touch一下，mtime就变了。因此在0.10版本开始，改为使用该文件最新一条消息的时间来判断。  
按大小清理这里也要注意，Kafka在定时任务中尝试比较当前日志量总大小是否超过阈值至少一个日志段的大小。如果超过但是没超过一个日志段，那么就不会删除。