# 初识Kafka

Kafka是一个分布式的发布-订阅消息系统，能够支撑海量数据的数据传递。在离线和实时的消 息处理业务系统中，Kafka都有广泛的应用。Kafka将消息持久化到磁盘中，并对消息创建了备份保证了 数据的安全。Kafka在保证了较高的处理速度的同时，又能保证数据处理的低延迟和数据的零丢失。

## 特性

高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个主题可以 分多个分区, 消费组对分区进行消费操作；

可扩展性：kafka集群支持热扩展；

持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失；

容错性：允许集群中节点失败（若副本数量为n,则允许n-1个节点失败）；

高并发：支持数千个客户端同时读写；

## 使用场景

日志收集：一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放 给各种consumer，例如Hadoop、Hbase、Solr等；

消息系统：解耦和生产者和消费者、缓存消息等；

用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点 击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时 的监控分析，或者装载到Hadoop、数据仓库中做离线分析和挖掘；

运营指标：Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作 的集中反馈，比如报警和报告；

流式处理：比如spark streaming和storm；

## 核心概念

### Producer

生产者即数据的发布者，该角色将消息发布到Kafka的topic中。broker接收到生产者发送的消息后， broker将该消息追加到当前用于追加数据的segment文件中。生产者发送的消息，存储到一个partition 中，生产者也可以指定数据存储的partition。

### Consumer

消费者可以从broker中读取数据。消费者可以消费多个topic中的数据。

### Topic

在Kafka中，使用一个类别属性来划分数据的所属类，划分数据的这个类称为topic。如果把Kafka看做 为一个数据库，topic可以理解为数据库中的一张表，topic的名字即为表名。

### Partition

topic中的数据分割为一个或多个partition。每个topic至少有一个partition。每个partition中的数据使 用多个segment文件存储。partition中的数据是有序的，partition间的数据丢失了数据的顺序。如果 topic有多个partition，消费数据时就不能保证数据的顺序。在需要严格保证消息的消费顺序的场景下， 需要将partition数目设为1。

### Partition oﬀset

每条消息都有一个当前Partition下唯一的64字节的oﬀset，它指明了这条消息的起始位置。

### Replicas of partition

副本是一个分区的备份。副本不会被消费者消费，副本只用于防止数据丢失，即消费者不从为follower 的partition中消费数据，而是从为leader的partition中读取数据。副本之间是一主多从的关系。

### Broker

Kafka 集群包含一个或多个服务器，服务器节点称为broker。broker存储topic的数据。如果某topic有 N个partition，集群有N个broker，那么每个broker存储该topic的一个partition。如果某topic有N个 partition，集群有(N+M)个broker，那么其中有N个broker存储该topic的一个partition，剩下的M个 broker不存储该topic的partition数据。如果某topic有N个partition，集群中broker数目少于N个，那么 一个broker存储该topic的一个或多个partition。在实际生产环境中，尽量避免这种情况的发生，这种 情况容易导致Kafka集群数据不均衡。

### Leader

每个partition有多个副本，其中有且仅有一个作为Leader，Leader是当前负责数据的读写的 partition。

### Follower

Follower跟随Leader，所有写请求都通过Leader路由，数据变更会广播给所有Follower，Follower与 Leader保持数据同步。如果Leader失效，则从Follower中选举出一个新的Leader。当Follower与 Leader挂掉、卡住或者同步太慢，leader会把这个follower从“in sync replicas”（ISR）列表中删除，重 新创建一个Follower。

### Zookeeper

Zookeeper负责维护和协调broker。当Kafka系统中新增了broker或者某个broker发生故障失效时，由 ZooKeeper通知生产者和消费者。生产者和消费者依据Zookeeper的broker状态信息与broker协调数据 的发布和订阅任务。

### AR(Assigned Replicas)

分区中所有的副本统称为AR。

### ISR(In-Sync Replicas)

所有与Leader部分保持一定程度的副（包括Leader副本在内）本组成ISR。

### OSR(Out-of-Sync-Replicas)

与Leader副本同步滞后过多的副本。

### HW(High Watermark)

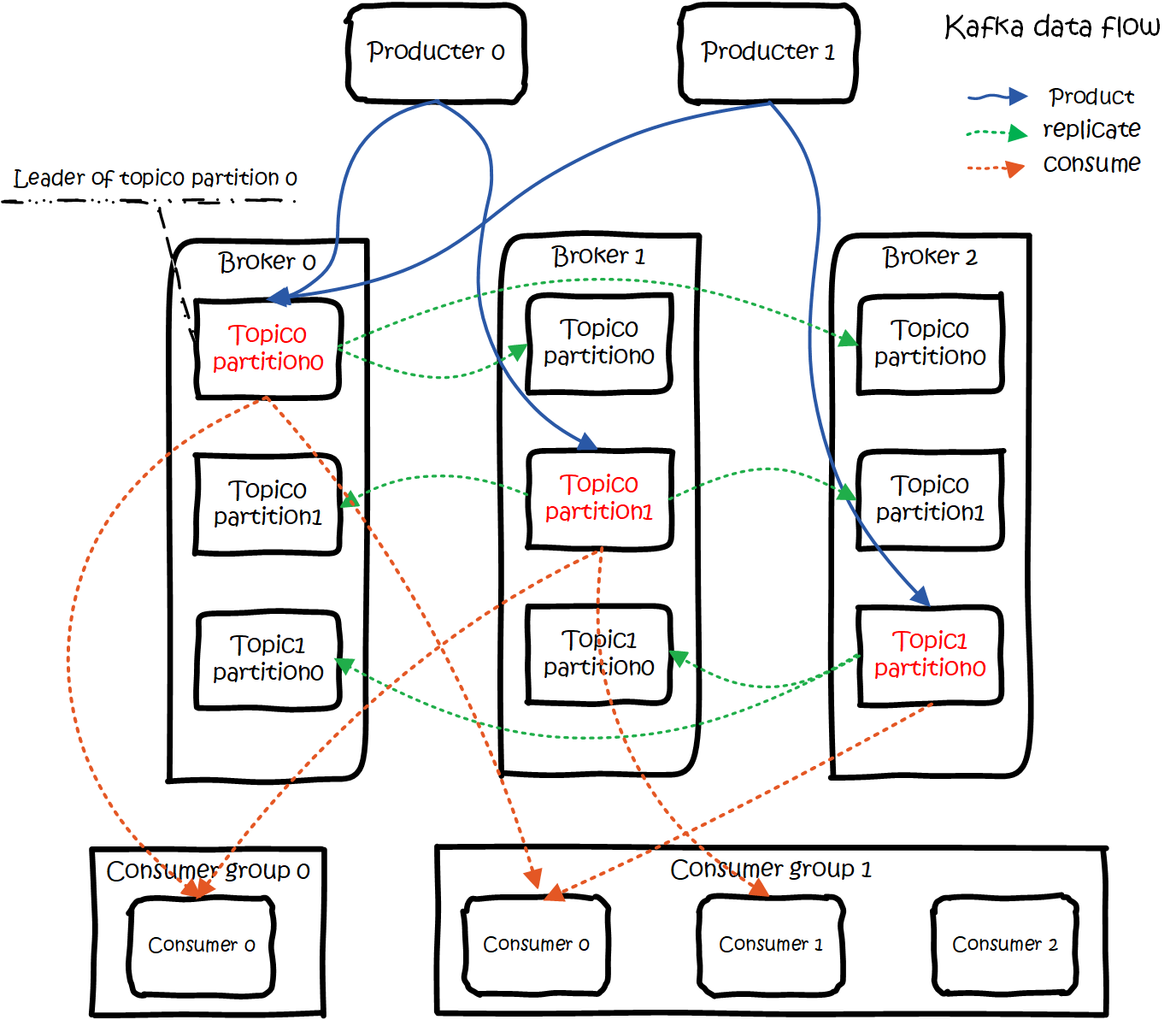
高水位，标识了一个特定的oﬀset，消费者只能拉取到这个oﬀset之前的消息。

### LEO(Log End Oﬀset)

即日志末端位移(log end oﬀset)，记录了该副本底层日志(log)中下一条消息的位移值。注意是下一条消 息！也就是说，如果LEO=10，那么表示该副本保存了10条消息，位移值范围是[0, 9]。



kafka的总体数据流是这样的：



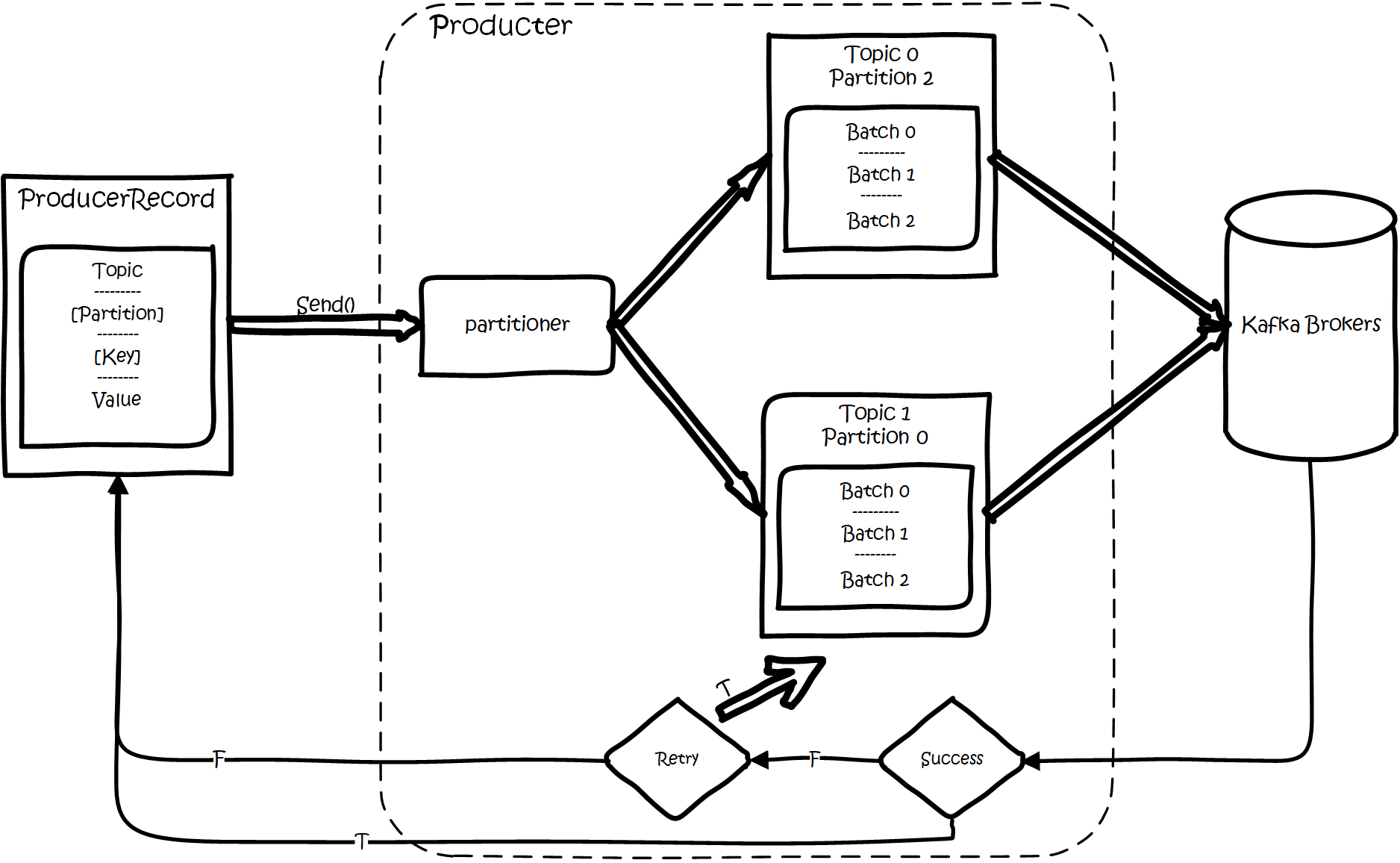
大概用法就是，Producers往Brokers里面的指定Topic中写消息，Consumers从Brokers里面拉去指定Topic的消息，然后进行业务处理。

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。

关于broker、topics、partitions的一些元信息用zk来存，监控和路由啥的也都会用到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，路由啥都替我们干了，用以来很简单。

还有simple api，offset啥的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

将所有Broker（假设共n个Broker）和待分配的Partition排序

将第i个Partition分配到第（i mod n）个Broker上 （这个就是leader）

将第i个Partition的第j个Replica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；

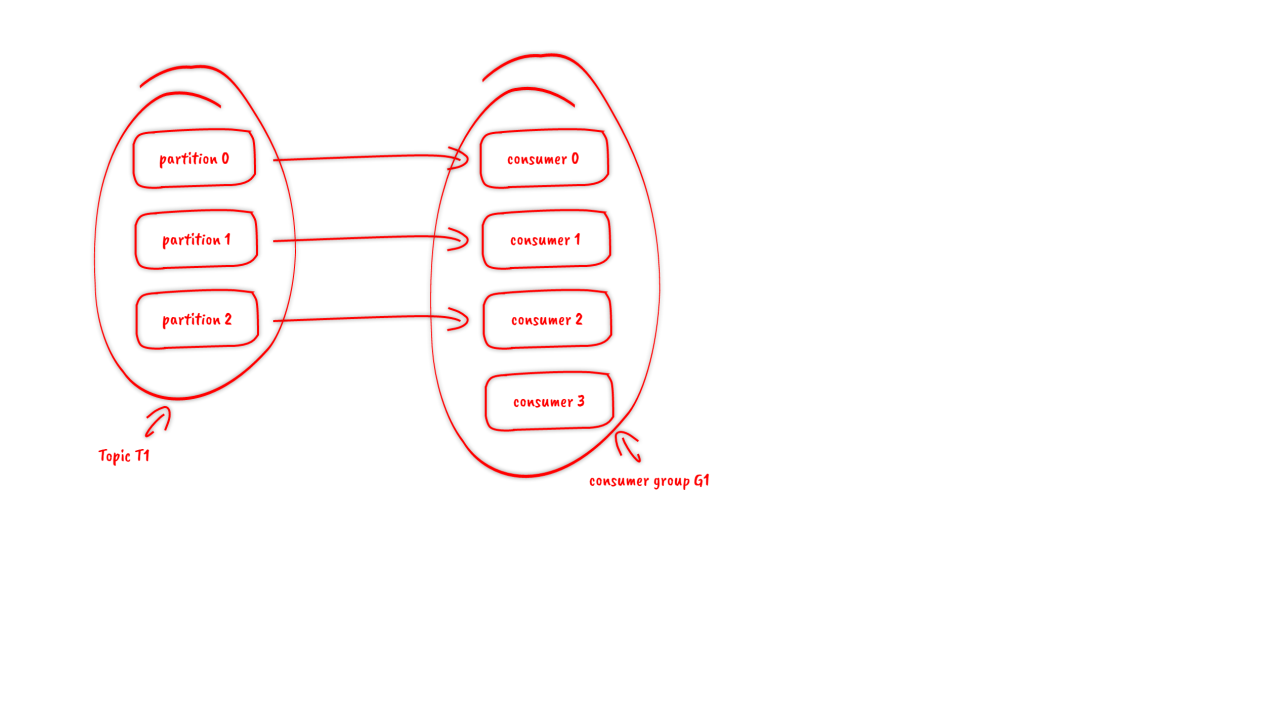
从leader的角度来说高水位的更新会延迟一轮，例如写入了一条新消息，ISR中的broker都fetch到了，但是ISR中的broker只有在下一轮的fetch中才能告诉leader。

**思考：**

当acks=-1时

1. 是follwers都来fetch就返回成功，还是等follwers第二轮fetch？
2. 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。

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

看offset保存在那个partition

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：

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。

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

consumer给coordinator发送JoinGroupRequest请求。

这时其他consumer发heartbeat请求过来时，coordinator会告诉他们，要reblance了。

其他consumer发送JoinGroupRequest请求。

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：

增加partition

增加消费者

1. 消费者主动关闭
2. 消费者宕机了
3. 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：

生产者重做导致重复写入消息----生产保证幂等性

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。

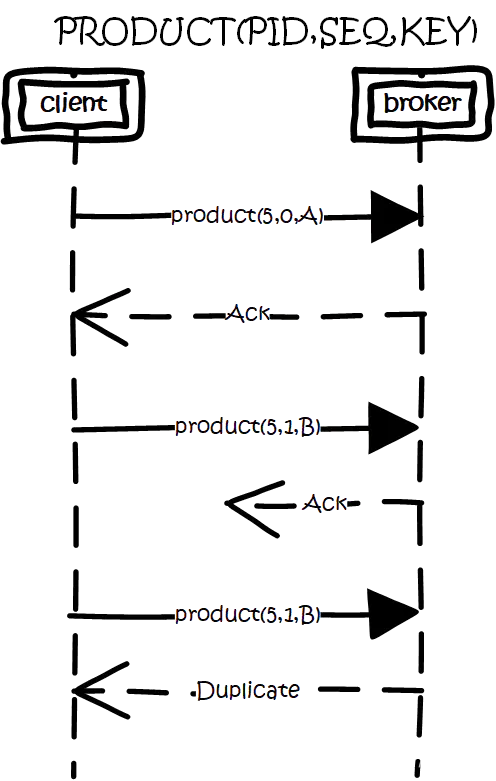
本来exactly once实现第1点就ok了。

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。

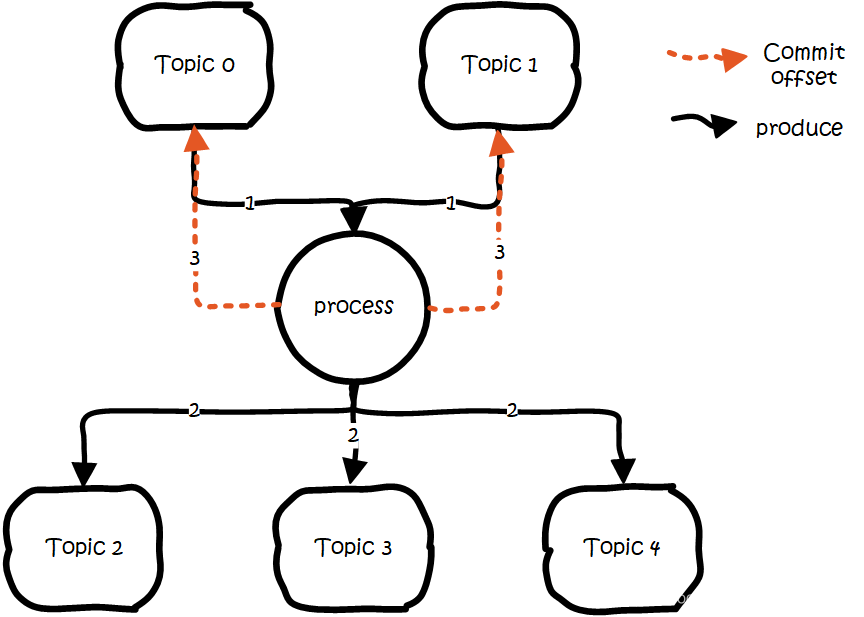


### 事务性/原子性广播

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：

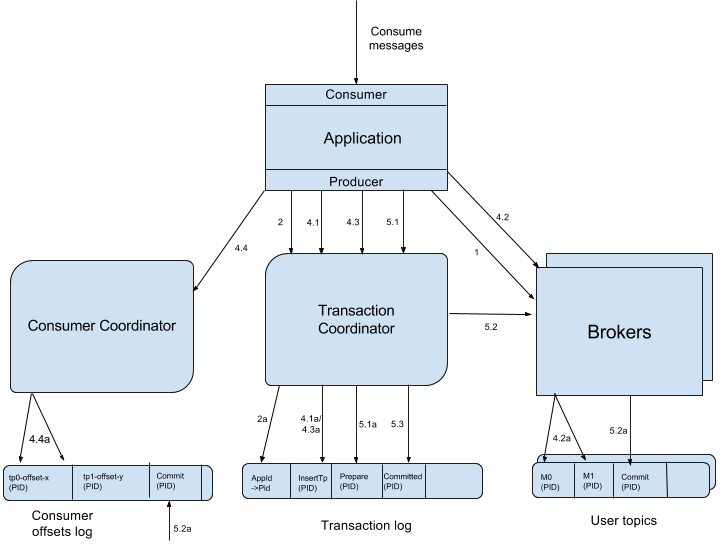
引入tid（transaction id），和pid不同，这个id是应用程序提供的，用于标识事务，和producer是谁并没关系。就是任何producer都可以使用这个tid去做事务，这样进行到一半就死掉的事务，可以由另一个producer去恢复。

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：



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marker4.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。

### 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。

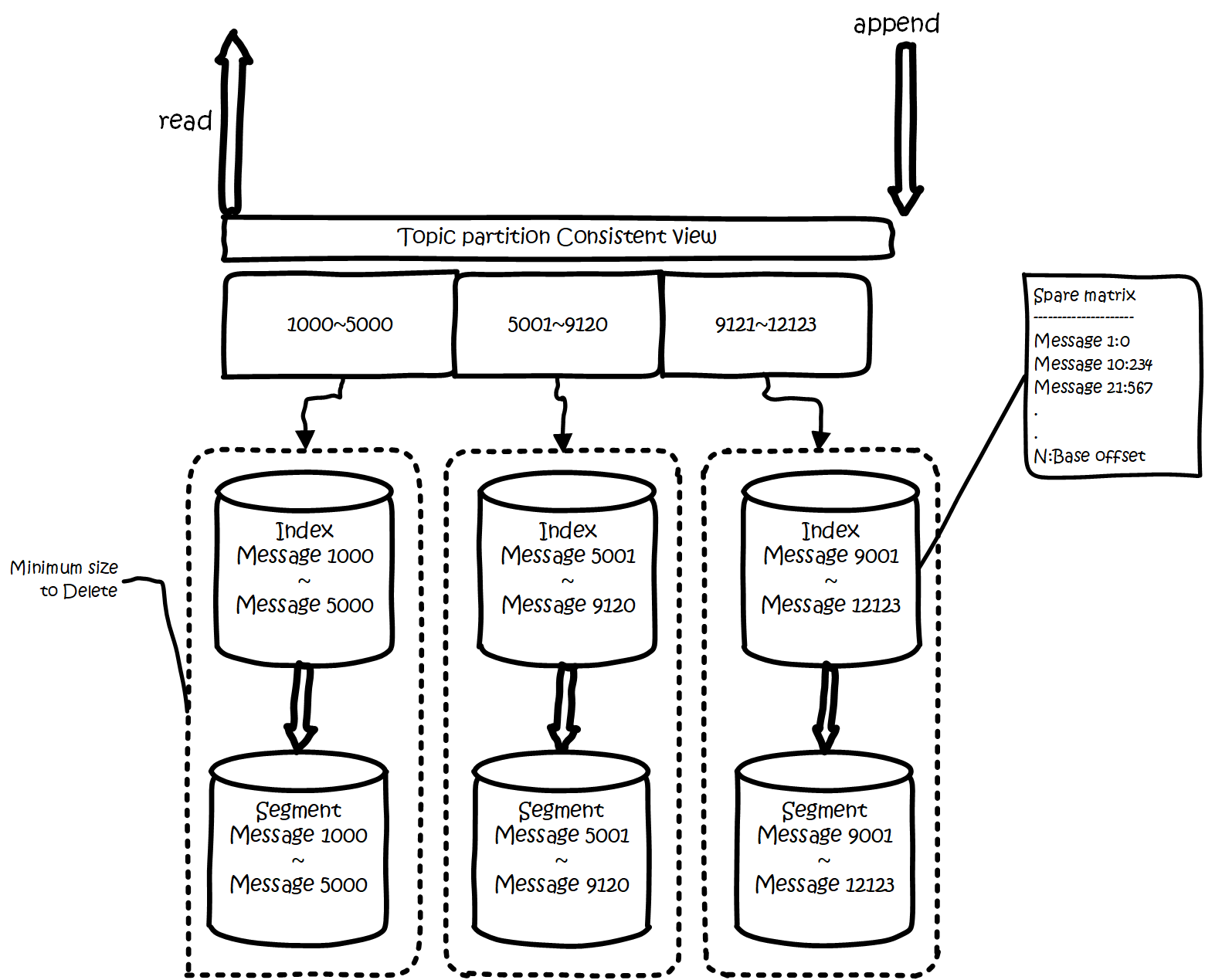
消费时，partition中会存在一些消息处于未commit状态，即业务方应该看不到的消息，需要过滤这些消息不让业务看到，kafka选择在消费者进程中进行过来，而不是在broker中过滤，主要考虑的还是性能。kafka高性能的一个关键点是zero copy，如果需要在broker中过滤，那么势必需要读取消息内容到内存，就会失去zero copy的特性。

# 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。

# Kafka与RabitMQ的区别

开源社区有许多优秀的队列中间件，比如RabbitMQ和Kafka，每个队列都貌似有其特性，在进行工程选择时，往往眼花缭乱，不知所措。对于RabbitMQ和Kafka，到底应该选哪个？

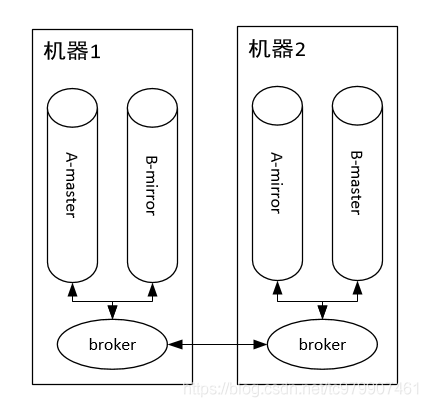
## RabbitMQ架构

RabbitMQ是一个分布式系统，这里面有几个抽象概念。

1：broker：每个节点运行的服务程序，功能为维护该节点的队列的增删以及转发队列操作请求。

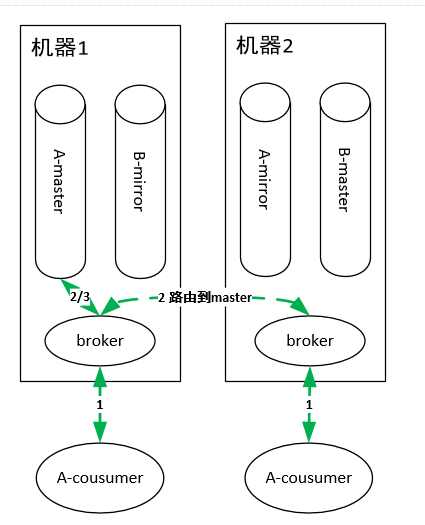
2：master queue：每个队列都分为一个主队列和若干个镜像队列。

3：mirror queue：镜像队列，作为master queue的备份。在master queue所在节点挂掉之后，系统把mirror queue提升为master queue，负责处理客户端队列操作请求。注意，mirror queue只做镜像，设计目的不是为了承担客户端读写压力。



如上图所示，集群中有两个节点，每个节点上有一个broker，每个broker负责本机上队列的维护，并且borker之间可以互相通信。集群中有两个队列A和B，每个队列都分为master queue和mirror queue（备份）。那么队列上的生产消费怎么实现的呢？

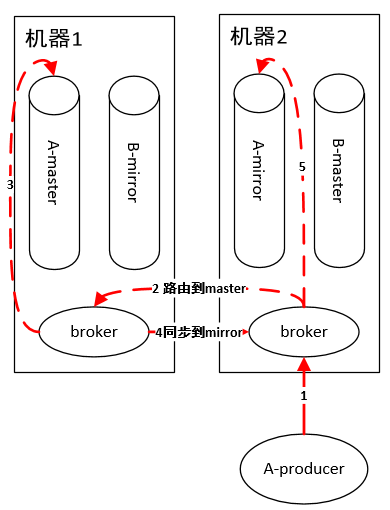
### 队列消费



如上图有两个consumer消费队列A，这两个consumer连在了集群的不同机器上。RabbitMQ集群中的任何一个节点都拥有集群上所有队列的元信息，所以连接到集群中的任何一个节点都可以，主要区别在于有的consumer连在master queue所在节点，有的连在非master queue节点上。

因为mirror queue要和master queue保持一致，故需要同步机制，正因为一致性的限制，导致所有的读写操作都必须都操作在master queue上（想想，为啥读也要从master queue中读？和数据库读写分离是不一样的。），然后由master节点同步操作到mirror queue所在的节点。即使consumer连接到了非master queue节点，该consumer的操作也会被路由到master queue所在的节点上，这样才能进行消费。

### 队列生产



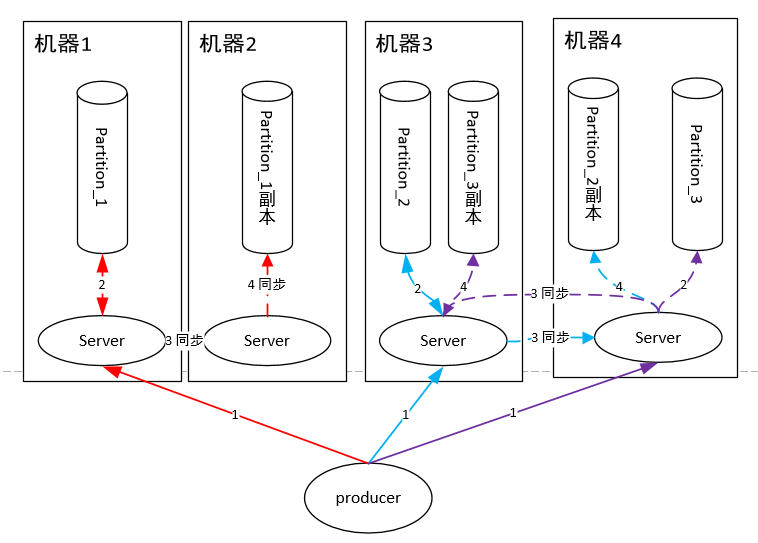
原理和消费一样，如果连接到非 master queue 节点，则路由过去。

所以，到这里小伙伴们就可以看到 RabbitMQ的不足：由于master queue单节点，导致性能瓶颈，吞吐量受限。虽然为了提高性能，内部使用了Erlang这个语言实现，但是终究摆脱不了架构设计上的致命缺陷。

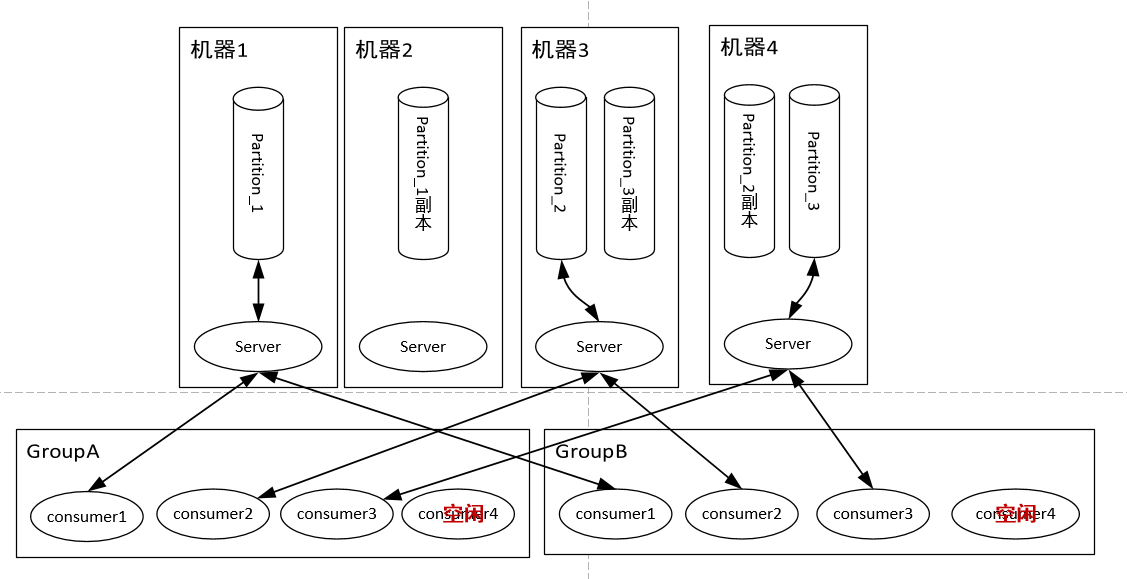
## Kafka

说实话，Kafka我觉得就是看到了RabbitMQ这个缺陷才设计出的一个改进版，改进的点就是：把一个队列的单一master变成多个master，即一台机器扛不住qps，那么我就用多台机器扛qps，把一个队列的流量均匀分散在多台机器上不就可以了么？注意，多个master之间的数据没有交集，即一条消息要么发送到这个master queue，要么发送到另外一个master queue。

这里面的每个master queue 在Kafka中叫做Partition，即一个分片。一个队列有多个主分片，每个主分片又有若干副分片做备份，同步机制类似于RabbitMQ。



如上图，我们省略了不同的queue，假设集群上只有一个queue（Kafka中叫Topic）。每个生产者随机把消息发送到主分片上，之后主分片再同步给副分片。



队列读取的时候虚拟出一个Group的概念，一个Topic内部的消息，只会路由到同Group内的一个consumer上，同一个Group中的consumer消费的消息是不一样的；Group之间共享一个Topic，看起来就是一个队列的多个拷贝。所以，为了达到多个Group共享一个Topic数据，Kafka并不会像RabbitMQ那样消息消费完毕立马删除，而是必须在后台配置保存日期，即只保存最近一段时间的消息，超过这个时间的消息就会从磁盘删除，这样就保证了在一个时间段内，Topic数据对所有Group可见（这个特性使得Kafka非常适合做一个公司的数据总线）。队列读同样是读主分片，并且为了优化性能，消费者与主分片有一一的对应关系，如果消费者数目大于分片数，则存在某些消费者得不到消息。

由此可见，Kafka绝对是为了高吞吐量设计的，比如设置分片数为100，那么就有100台机器去扛一个Topic的流量，当然比RabbitMQ的单机性能好。

# 总结

1：吞吐量较低：Kafka和RabbitMQ都可以。

2：吞吐量高：Kafka。

参考资料：

https://www.cnblogs.com/qingyunzong/category/1212387.html

https://www.jianshu.com/p/d3e963ff8b70

https://www.jianshu.com/p/3d15798b88d2