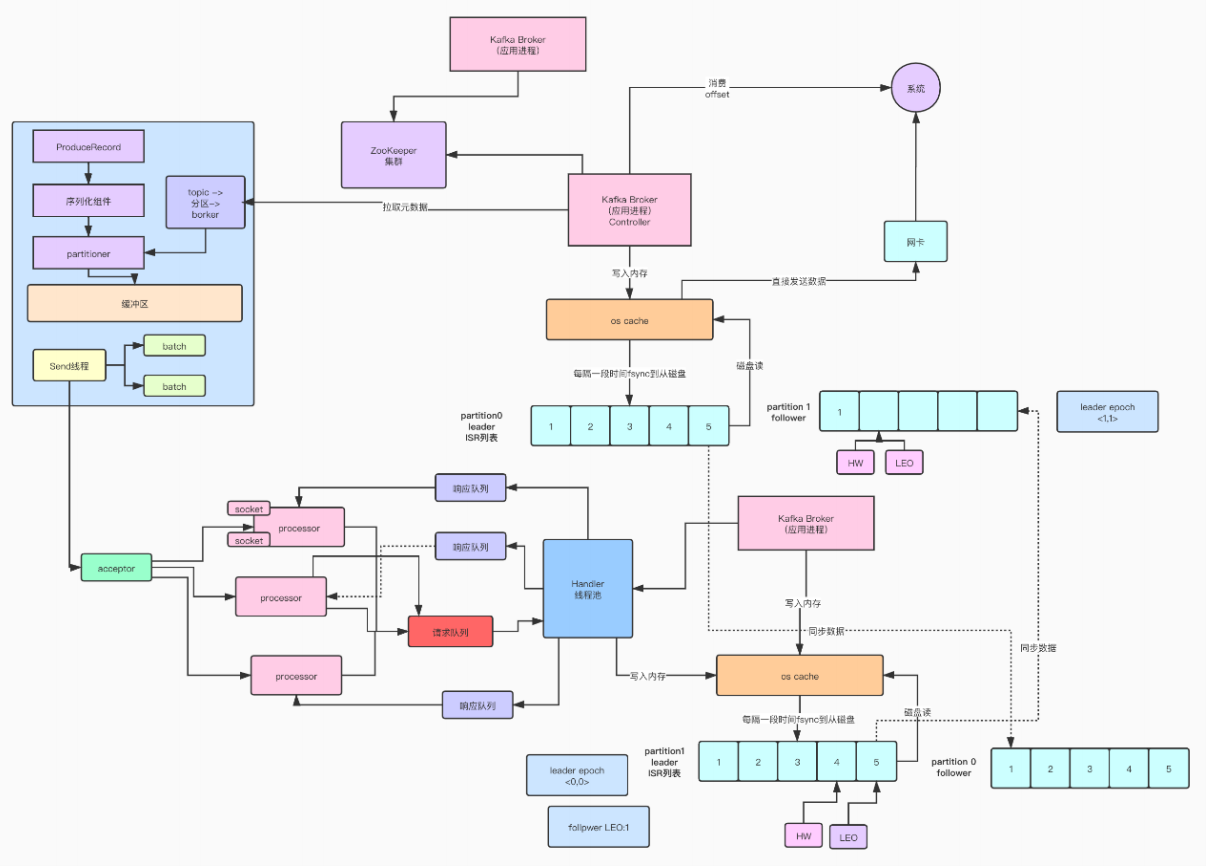


# Kafka概述

## Kafka 概述



### kafka

Kakfa起初是由LinkedIn公司开发的一个分布式的消息系统，后成为Apache的一部分，它使用Scala编写，以可水平扩展和高吞吐率而被广泛使用。目前越来越多的开源分布式处理系统如Cloudera、Apache Storm、Spark等都支持与Kafka集成。

Kafka凭借着自身的优势，越来越受到互联网企业的青睐，唯品会也采用Kafka作为其内部核心消息引擎之一。Kafka作为一个商业级消息中间件，消息可靠性的重要性可想而知。如何确保消息的精确传输？如何确保消息的准确存储？如何确保消息的正确消费？这些都是需要考虑的问题。本文首先从Kafka的架构着手，先了解下Kafka的基本原理，然后通过对kakfa的存储机制、复制原理、同步原理、可靠性和持久性保证等等一步步对其可靠性进行分析，最后通过benchmark来增强对Kafka高可靠性的认知。

Kafka是一款分布式消息发布和订阅系统，它的特点是高性能、高吞吐量。

### 设计目标

场景可以用于 web/nginx 日志、访问日志，消息服务等等。

1、行为跟踪。

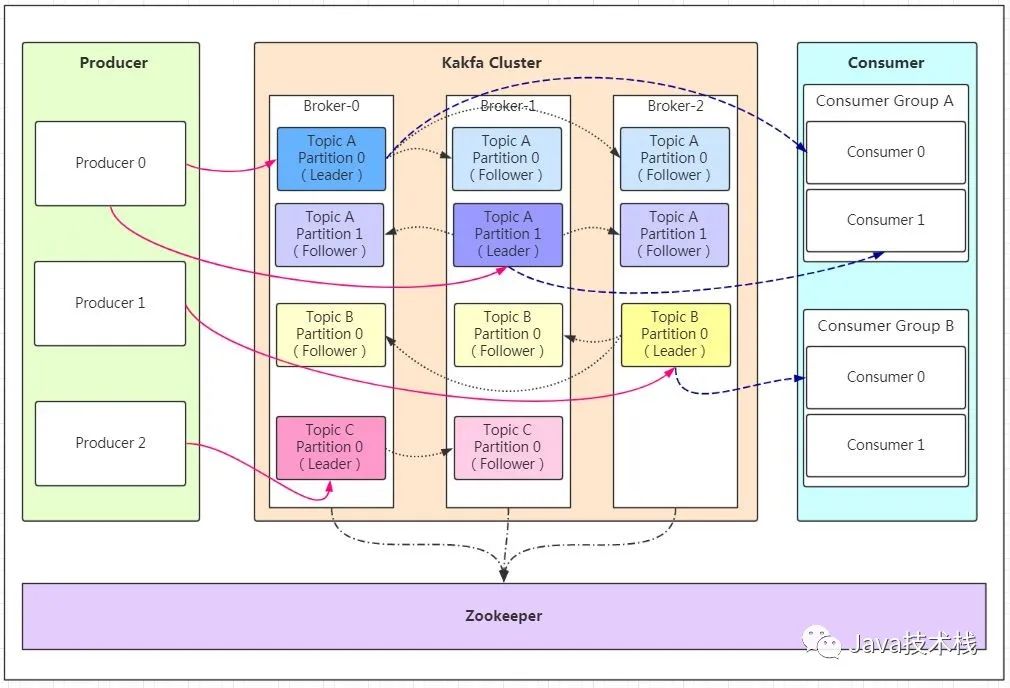
kafka 可以用于跟踪用户浏览页面、搜索及其他行为。通过发布-订阅模式实时记录到对应的Topic 中，通过后端大数据平台接入处理分析，并做更进一步的实时处理和监控

2、日志收集系统。

日志收集方面，有很多比较优秀的产品，比如 Apache Flume， 很多公司使用kafka 代理日志聚合。日志聚合表示从服务器上收集日志文件，然后放到一个集中的平台（文件服务器）进行处理。在实际应用开发中，我们应用程序的 log 都会输出到本地的磁盘上，排查问题的话通过Linux命令来搞定，如果应用程序组成了负载均衡集群，并且集群的机器有几十台以上，那么想通过日志快速定位到问题，就是很麻烦的的事了。所以一般都会做一个日志统一收集平台管理 log 日志用来快速定位到问题。所以很多公司的套路都是把应用日志集中到kafka上，然后分布导入到es 和 hdfs 上。用来做实时检索分析和离线统计数据备份等。而另一个方面，kafka 本身又提供了很好的api来集成日志并且做日志收集

3、消息系统。

## Kafka体系架构



Counsumer:

单个分区的消息只能由consumerGroup中某个consumer消费。

一个consumer可以消费多个partion，但是一个partion不能被多个consumer消费。

如上图所示，一个典型的Kafka体系架构包括若干Producer（可以是服务器日志，业务数据，页面前端产生的page view等等），若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干Consumer (Group)，以及一个Zookeeper集群。Kafka通过Zookeeper管理集群配置，选举leader，以及在consumer group发生变化时进行rebalance。Producer使用push(推)模式将消息发布到broker，Consumer使用pull(拉)模式从broker订阅并消费消息。

角色;

Producer：Producer即生产者，消息的产生者，是消息的入口。

kafka cluster：Broker：Broker是kafka实例，每个服务器上有一个或多个kafka的实例，我们姑且认为每个broker对应一台服务器。每个kafka集群内的broker都有一个不重复的编号，如图中的broker-0、broker-1等……

Topic：消息的主题，可以理解为消息的分类，kafka的数据就保存在topic。在每个broker上都可以创建多个topic。

Partition：Topic的分区，每个topic可以有多个分区，分区的作用是做负载，提高kafka的吞吐量。同一个topic在不同的分区的数据是不重复的，partition的表现形式就是一个一个的文件夹！

Replication:每一个分区都有多个副本，副本的作用是做备胎。当主分区（Leader）故障的时候会选择一个备胎（Follower）上位，成为Leader。在kafka中默认副本的最大数量是10个，且副本的数量不能大于Broker的数量，follower和leader绝对是在不同的机器，同一机器对同一个分区也只可能存放一个副本（包括自己）。

若创建 Topic1 和Topic2 两个Topic，且分别有13个和19个分区，则整个集群上相应会生成共32个文件夹

Message：每一条发送的消息主体。

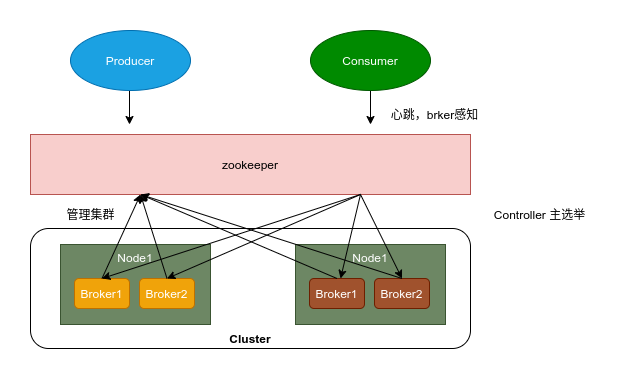
Consumer：消费者，即消息的消费方，是消息的出口。

Consumer Group：我们可以将多个消费组组成一个消费者组，在kafka的设计中同一个分区的数据只能被消费者组中的某一个消费者消费。同一个消费者组的消费者可以消费同一个topic的不同分区的数据，这也是为了提高kafka的吞吐量！

每个Consumer 属于一个特定的Consumer Group，可为每个Consumer 指定Group name，若不指定group name 则属于默认的group，每条消息只可以被Consumer Goup 组中中的一个Consumer消费，但是可以指定多个Consumer Group,所以一个消息在Consumer Group 里面只可以被消费一次。已确定！

Zookeeper：kafka集群依赖zookeeper来保存集群的的元信息，来保证系统的可用性。

### Zookeeper



Broker 注册：Broker 是分布式部署并且之间相互独立，Zookeeper 用来管理注册到集群的所有 Broker 节点。

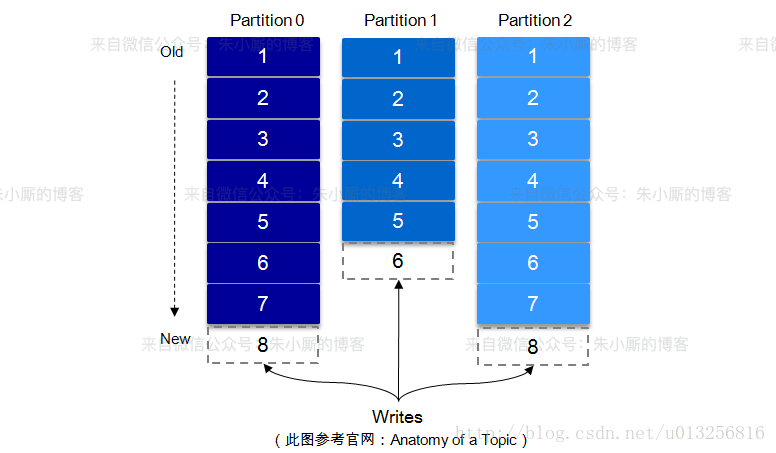
Topic 注册：在 Kafka 中，同一个 Topic 的消息会被分成多个分区并将其分布在多个 Broker 上，这些分区信息及与 Broker 的对应关系也都是由 Zookeeper 在维护。

生产者负载均衡：由于同一个 Topic 消息会被分区并将其分布在多个 Broker 上，因此，生产者需要将消息合理地发送到这些分布式的 Broker 上。

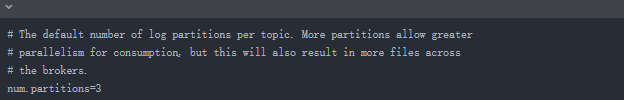
消费者负载均衡：与生产者类似，Kafka 中的消费者同样需要进行负载均衡来实现多个消费者合理地从对应的 Broker 服务器上接收消息，每个消费者分组包含若干消费者，每条消息都只会发送给分组中的一个消费者，不同的消费者分组消费自己特定的 Topic 下面的消息，互不干扰。

### Topic & Partition

一个topic可以认为一个一类消息，每个topic将被分成多个partition，每个partition在存储层面是append log文件。任何发布到此partition的消息都会被追加到log文件的尾部，每条消息在文件中的位置称为offset(偏移量)，offset为一个long型的数字，它唯一标记一条消息。每条消息都被append到partition中，是顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是Kafka高吞吐率的一个很重要的保证）。



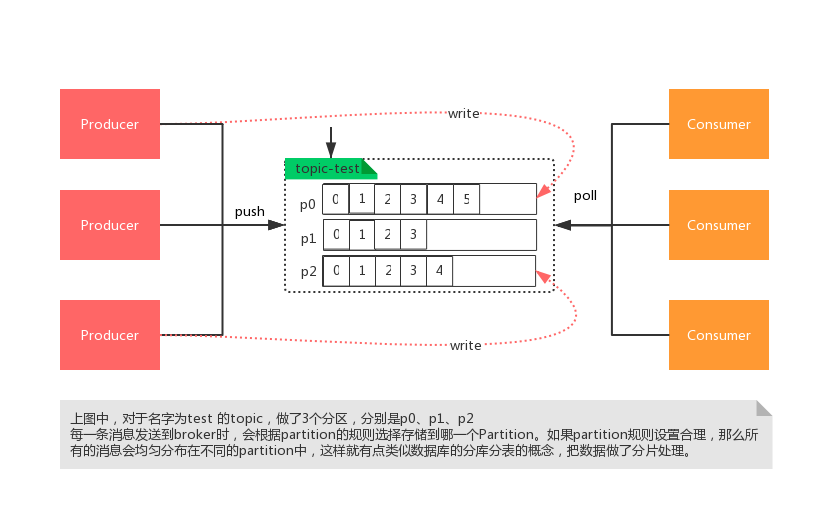
每一条消息被发送到broker中，会根据partition规则选择被存储到哪一个partition。如果partition规则设置的合理，所有消息可以均匀分布到不同的partition里，这样就实现了水平扩展。（如果一个topic对应一个文件，那这个文件所在的机器I/O将会成为这个topic的性能瓶颈，而partition解决了这个问题）。在创建topic时可以在$KAFKA\_HOME/config/server.properties中指定这个partition的数量（如下所示），当然可以在topic创建之后去修改partition的数量。



在发送一条消息时，可以指定这个消息的key，producer根据这个key和partition机制来判断这个消息发送到哪个partition。partition机制可以通过指定producer的partition.class这一参数来指定，该class必须实现kafka.producer.Partitioner接口。

### Partition

每个Topic 可以划分多个分区（每个Topic至少有一个分区），同一topic 不同分区包含不同分区包含的消息是不同的。每个消息在被添加到分区时，都会被分配一个offset （偏移量），它是消息在此分区中的唯一编号，kafka 通过offset 保证消息在分区内的顺序，offset 的顺序不跨分区，即kafka 只保证同一个分区内的消息是有序的。

、

## 高可靠性存储分析

### 高可靠性存储分析

Kafka的高可靠性的保障来源于其健壮的副本（replication）策略。通过调节其副本相关参数，可以使得Kafka在性能和可靠性之间运转的游刃有余。Kafka从0.8.x版本开始提供partition级别的复制,replication的数量可以在$KAFKA\_HOME/config/server.properties中配置（default.replication.refactor）。

这里先从Kafka文件存储机制入手，从最底层了解Kafka的存储细节，进而对其的存储有个微观的认知。之后通过Kafka复制原理和同步方式来阐述宏观层面的概念。最后从ISR，HW，leader选举以及数据可靠性和持久性保证等等各个维度来丰富对Kafka相关知识点的认知。

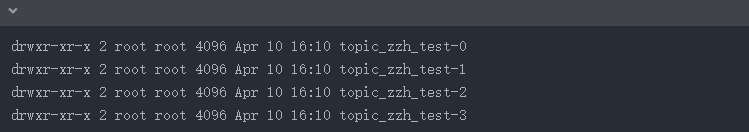
## Kafka文件存储机制

### 文件存储机制

Kafka中消息是以topic进行分类的，生产者通过topic向Kafka broker发送消息，消费者通过topic读取数据。然而topic在物理层面又能以partition为分组，一个topic可以分成若干个partition，那么topic以及partition又是怎么存储的呢？partition还可以细分为segment，一个partition物理上由多个segment组成，那么这些segment又是什么呢？下面我们来一一揭晓。

### partition

为了便于说明问题，假设这里只有一个Kafka集群，且这个集群只有一个Kafka broker，即只有一台物理机。在这个Kafka broker中配置（$KAFKA\_HOME/config/server.properties中）log.dirs=/tmp/kafka-logs，以此来设置Kafka消息文件存储目录，与此同时创建一个topic：topic\_zzh\_test，partition的数量为4（$KAFKA\_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 4 --topic topic\_zzh\_test --replication-factor 1）。那么我们此时可以在/tmp/kafka-logs目录中可以看到生成了4个目录：

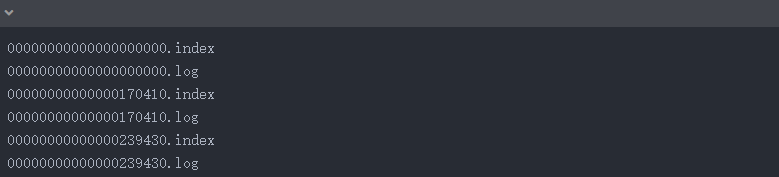


在Kafka文件存储中，同一个topic下有多个不同的partition，每个partiton为一个目录，partition的名称规则为：topic名称+有序序号，第一个序号从0开始计，最大的序号为partition数量减1，partition是实际物理上的概念，而topic是逻辑上的概念。

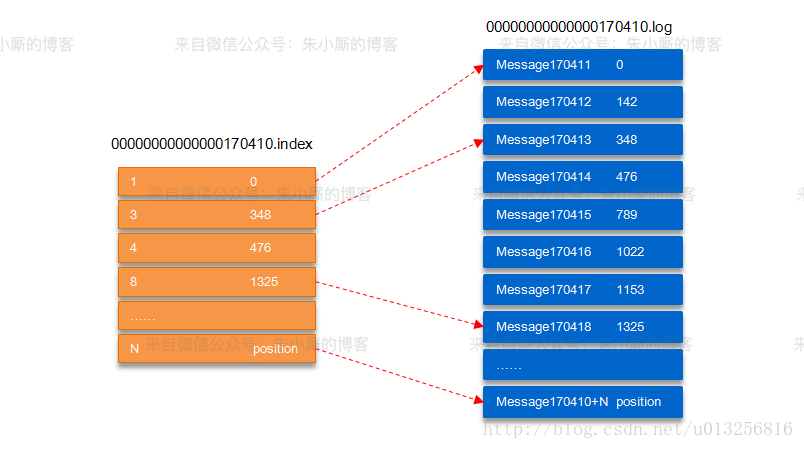
### segment

上面提到partition还可以细分为segment，这个segment又是什么？如果就以partition为最小存储单位，我们可以想象当Kafka producer不断发送消息，必然会引起partition文件的无限扩张，这样对于消息文件的维护以及已经被消费的消息的清理带来严重的影响，所以这里以segment为单位又将partition细分。每个partition(目录)相当于一个巨型文件被平均分配到多个大小相等的segment(段)数据文件中（每个segment 文件中消息数量不一定相等）这种特性也方便old segment的删除，即方便已被消费的消息的清理，提高磁盘的利用率。每个partition只需要支持顺序读写就行，segment的文件生命周期由服务端配置参数（log.segment.bytes，log.roll.{ms,hours}等若干参数）决定。

segment文件由两部分组成，分别为“.index”文件和“.log”文件，分别表示为segment索引文件和数据文件。这两个文件的命令规则为：partition全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值，数值大小为64位，20位数字字符长度，没有数字用0填充，如下：



以上面的segment文件为例，展示出segment：00000000000000170410的“.index”文件和“.log”文件的对应的关系，如下图



如上图，“.index”索引文件存储大量的元数据，“.log”数据文件存储大量的消息，索引文件中的元数据指向对应数据文件中message的物理偏移地址。其中以“.index”索引文件中的元数据[3, 348]为例，在“.log”数据文件表示第3个消息，即在全局partition中表示170410+3=170413个消息，该消息的物理偏移地址为348。

那么如何从partition中通过offset查找message呢？

以上图为例，读取offset=170418的消息，首先查找segment文件，其中00000000000000000000.index为最开始的文件，第二个文件为00000000000000170410.index（起始偏移为170410+1=170411），而第三个文件为00000000000000239430.index（起始偏移为239430+1=239431），所以这个offset=170418就落到了第二个文件之中。其他后续文件可以依次类推，以其实偏移量命名并排列这些文件，然后根据二分查找法就可以快速定位到具体文件位置。其次根据00000000000000170410.index文件中的[8,1325]定位到00000000000000170410.log文件中的1325的位置进行读取。

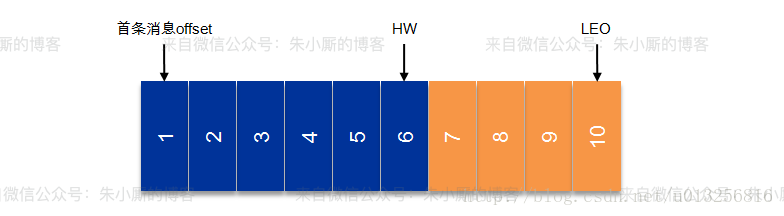
要是读取offset=170418的消息，从00000000000000170410.log文件中的1325的位置进行读取，那么怎么知道何时读完本条消息，否则就读到下一条消息的内容了？

这个就需要联系到消息的物理结构了，消息都具有固定的物理结构，包括：offset（8 Bytes）、消息体的大小（4 Bytes）、crc32（4 Bytes）、magic（1 Byte）、attributes（1 Byte）、key length（4 Bytes）、key（K Bytes）、payload(N Bytes)等等字段，可以确定一条消息的大小，即读取到哪里截止。

## 复制原理和同步方式

### 复制原理

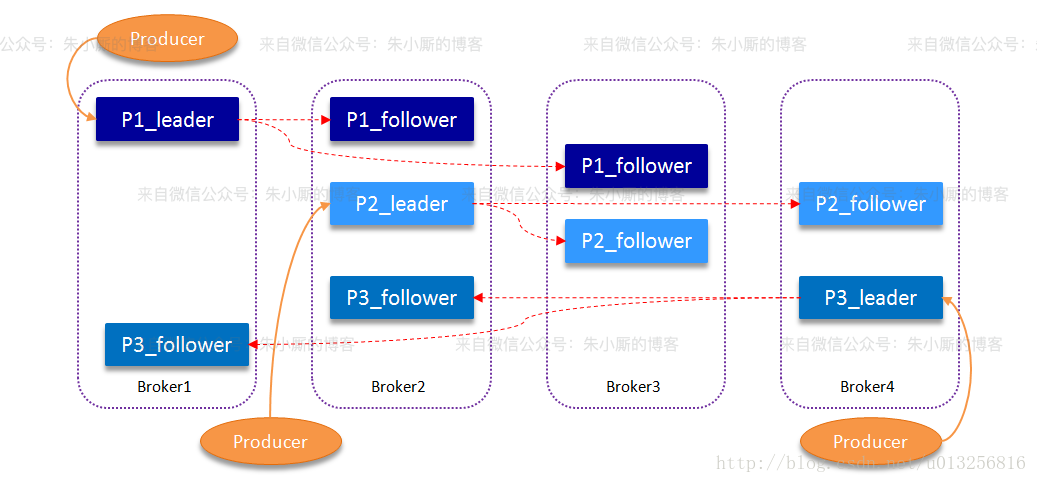
Kafka中topic的每个partition有一个预写式的日志文件，虽然partition可以继续细分为若干个segment文件，但是对于上层应用来说可以将partition看成最小的存储单元（一个有多个segment文件拼接的“巨型”文件），每个partition都由一些列有序的、不可变的消息组成，这些消息被连续的追加到partition中。



上图中有两个新名词：HW和LEO。这里先介绍下LEO，LogEndOffset的缩写，表示每个partition的log最后一条Message的位置。HW是HighWatermark的缩写，是指consumer能够看到的此partition的位置，这个涉及到多副本的概念，这里先提及一下，下节再详表。

言归正传，为了提高消息的可靠性，Kafka每个topic的partition有N个副本（replicas），其中N(大于等于1)是topic的复制因子（replica fator）的个数。Kafka通过多副本机制实现故障自动转移，当Kafka集群中一个broker失效情况下仍然保证服务可用。在Kafka中发生复制时确保partition的日志能有序地写到其他节点上，N个replicas中，其中一个replica为leader，其他都为follower, leader处理partition的所有读写请求，与此同时，follower会被动定期地去复制leader上的数据。

如下图所示，Kafka集群中有4个broker, 某topic有3个partition,且复制因子即副本个数也为3：



Kafka提供了数据复制算法保证，如果leader发生故障或挂掉，一个新leader被选举并被接受客户端的消息成功写入。Kafka确保从同步副本列表中选举一个副本为leader，或者说follower追赶leader数据。leader负责维护和跟踪ISR(In-Sync Replicas的缩写，表示副本同步队列，具体可参考下节)中所有follower滞后的状态。当producer发送一条消息到broker后，leader写入消息并复制到所有follower。消息提交之后才被成功复制到所有的同步副本。消息复制延迟受最慢的follower限制，重要的是快速检测慢副本，如果follower“落后”太多或者失效，leader将会把它从ISR中删除。

### ISR

上节我们涉及到ISR (In-Sync Replicas)，这个是指副本同步队列。副本数对Kafka的吞吐率是有一定的影响，但极大的增强了可用性。默认情况下Kafka的replica数量为1，即每个partition都有一个唯一的leader，为了确保消息的可靠性，通常应用中将其值(由broker的参数offsets.topic.replication.factor指定)大小设置为大于1，比如3。 所有的副本（replicas）统称为Assigned Replicas，即AR。ISR是AR中的一个子集，由leader维护ISR列表，follower从leader同步数据有一些延迟（包括延迟时间replica.lag.time.max.ms和延迟条数replica.lag.max.messages两个维度, 当前最新的版本0.10.x中只支持replica.lag.time.max.ms这个维度），任意一个超过阈值都会把follower剔除出ISR, 存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中。AR=ISR+OSR。

Kafka 0.9.0.0版本后移除了replica.lag.max.messages参数，只保留了replica.lag.time.max.ms作为ISR中副本管理的参数。为什么这样做呢？replica.lag.max.messages表示当前某个副本落后leader的消息数量超过了这个参数的值，那么leader就会把follower从ISR中删除。假设设置replica.lag.max.messages=4，那么如果producer一次传送至broker的消息数量都小于4条时，因为在leader接受到producer发送的消息之后而follower副本开始拉取这些消息之前，follower落后leader的消息数不会超过4条消息，故此没有follower移出ISR，所以这时候replica.lag.max.message的设置似乎是合理的。但是producer发起瞬时高峰流量，producer一次发送的消息超过4条时，也就是超过replica.lag.max.messages，此时follower都会被认为是与leader副本不同步了，从而被踢出了ISR。但实际上这些follower都是存活状态的且没有性能问题。那么在之后追上leader,并被重新加入了ISR。于是就会出现它们不断地剔出ISR然后重新回归ISR，这无疑增加了无谓的性能损耗。而且这个参数是broker全局的。设置太大了，影响真正“落后”follower的移除；设置的太小了，导致follower的频繁进出。无法给定一个合适的replica.lag.max.messages的值，故此，新版本的Kafka移除了这个参数。

注：ISR中包括：leader和follower。

上面一节还涉及到一个概念，即HW。HW俗称高水位，HighWatermark的缩写，取一个partition对应的ISR中最小的LEO作为HW，consumer最多只能消费到HW所在的位置。另外每个replica都有HW,leader和follower各自负责更新自己的HW的状态。对于leader新写入的消息，consumer不能立刻消费，leader会等待该消息被所有ISR中的replicas同步后更新HW，此时消息才能被consumer消费。这样就保证了如果leader所在的broker失效，该消息仍然可以从新选举的leader中获取。对于来自内部broker的读取请求，没有HW的限制。

下图详细的说明了当producer生产消息至broker后，ISR以及HW和LEO的流转过程：

由此可见，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。事实上，同步复制要求所有能工作的follower都复制完，这条消息才会被commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，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都还没有复制完，落后于leader时，突然leader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。

Kafka的ISR的管理最终都会反馈到Zookeeper节点上。具体位置为：/brokers/topics/[topic]/partitions/[partition]/state。目前有两个地方会对这个Zookeeper的节点进行维护：

Controller来维护：Kafka集群中的其中一个Broker会被选举为Controller，主要负责Partition管理和副本状态管理，也会执行类似于重分配partition之类的管理任务。在符合某些特定条件下，Controller下的LeaderSelector会选举新的leader，ISR和新的leader\_epoch及controller\_epoch写入Zookeeper的相关节点中。同时发起LeaderAndIsrRequest通知所有的replicas。

leader来维护：leader有单独的线程定期检测ISR中follower是否脱离ISR, 如果发现ISR变化，则会将新的ISR的信息返回到Zookeeper的相关节点中。

### 3数据可靠性和持久性保证

当producer向leader发送数据时，可以通过request.required.acks参数来设置数据可靠性的级别：

1（默认）：这意味着producer在ISR中的leader已成功收到数据并得到确认。如果leader宕机了，则会丢失数据。

0：这意味着producer无需等待来自broker的确认而继续发送下一批消息。这种情况下数据传输效率最高，但是数据可靠性确是最低的。

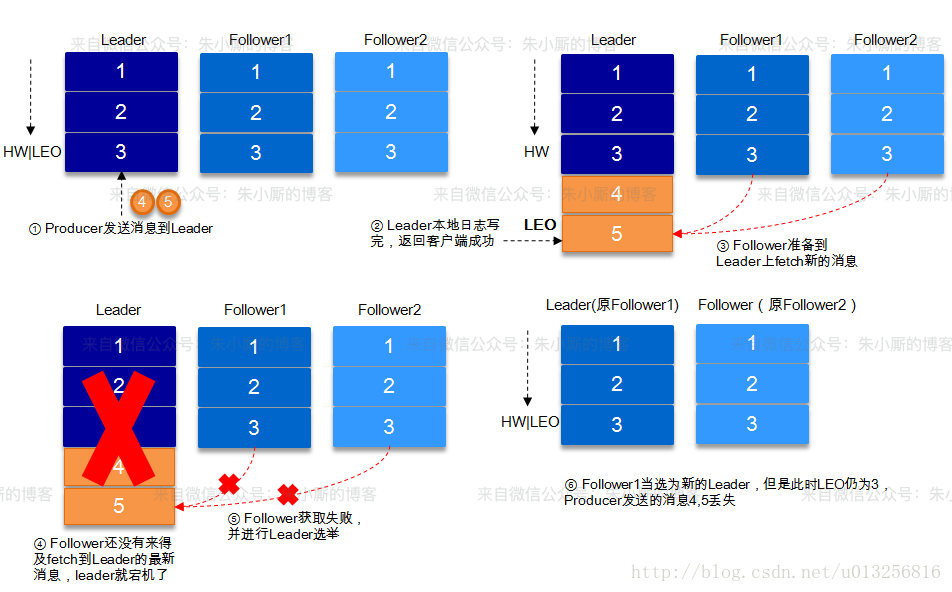
-1：producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。但是这样也不能保证数据不丢失，比如当ISR中只有leader时（前面ISR那一节讲到，ISR中的成员由于某些情况会增加也会减少，最少就只剩一个leader），这样就变成了acks=1的情况。

如果要提高数据的可靠性，在设置request.required.acks=-1的同时，也要min.insync.replicas这个参数(可以在broker或者topic层面进行设置)的配合，这样才能发挥最大的功效。min.insync.replicas这个参数设定ISR中的最小副本数是多少，默认值为1，当且仅当request.required.acks参数设置为-1时，此参数才生效。如果ISR中的副本数少于min.insync.replicas配置的数量时，客户端会返回异常：org.apache.kafka.common.errors.NotEnoughReplicasExceptoin: Messages are rejected since there are fewer in-sync replicas than required。

接下来对acks=1和-1的两种情况进行详细分析：

1. request.required.acks=1

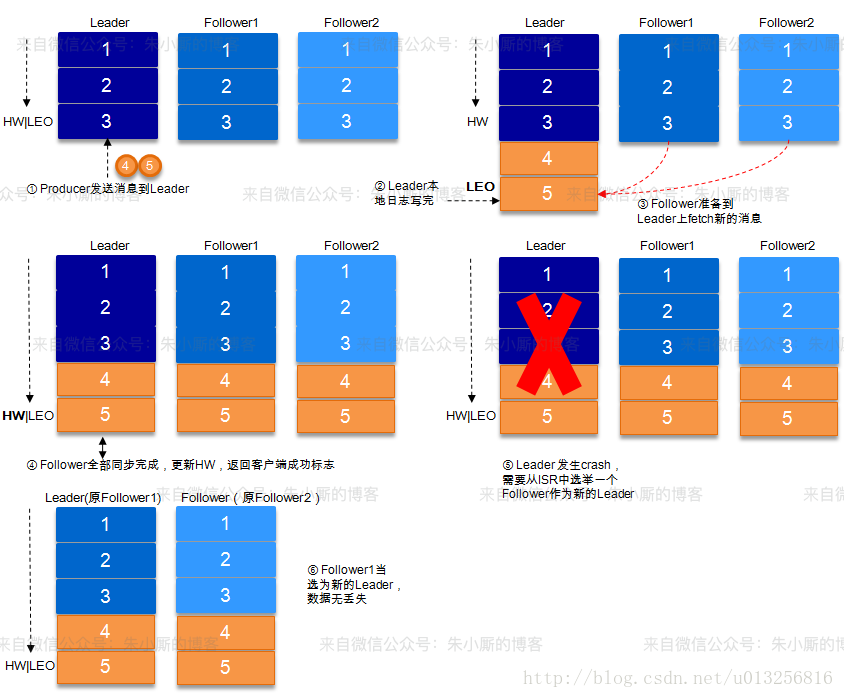
producer发送数据到leader，leader写本地日志成功，返回客户端成功；此时ISR中的副本还没有来得及拉取该消息，leader就宕机了，那么此次发送的消息就会丢失。



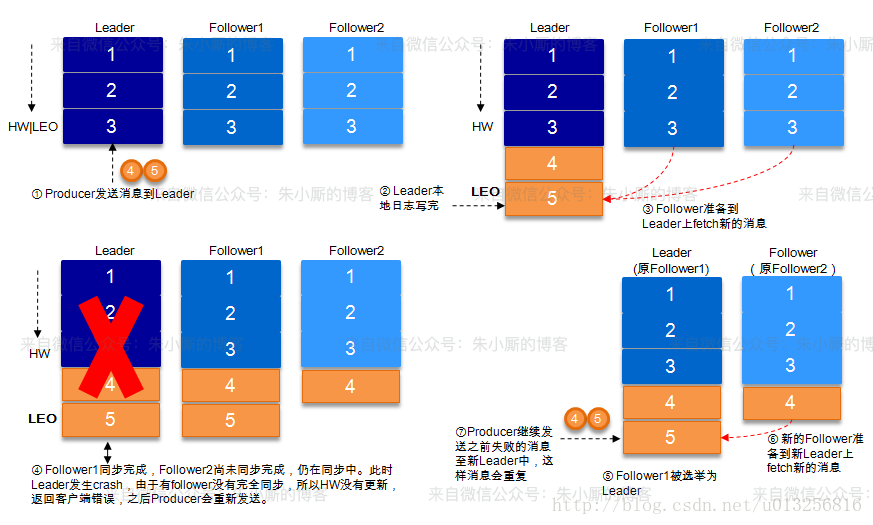
2. request.required.acks=-1

同步（Kafka默认为同步，即producer.type=sync）的发送模式，replication.factor>=2且min.insync.replicas>=2的情况下，不会丢失数据。

有两种典型情况。acks=-1的情况下（如无特殊说明，以下acks都表示为参数request.required.acks），数据发送到leader, ISR的follower全部完成数据同步后，leader此时挂掉，那么会选举出新的leader，数据不会丢失。



acks=-1的情况下，数据发送到leader后 ，部分ISR的副本同步，leader此时挂掉。比如follower1和follower2都有可能变成新的leader, producer端会得到返回异常，producer端会重新发送数据，数据可能会重复。



当然上图中如果在leader crash的时候，follower2还没有同步到任何数据，而且follower2被选举为新的leader的话，这样消息就不会重复。

注：Kafka只处理fail/recover问题,不处理Byzantine问题。

## PRO和CON的消息传输保障

前面已经介绍了Kafka如何进行有效的存储，以及了解了producer和consumer如何工作。接下来讨论的是Kafka如何确保消息在producer和consumer之间传输。有以下三种可能的传输保障（delivery guarantee）:

At most once: 消息可能会丢，但绝不会重复传输

At least once：消息绝不会丢，但可能会重复传输

Exactly once：每条消息肯定会被传输一次且仅传输一次

Kafka的消息传输保障机制非常直观。当producer向broker发送消息时，一旦这条消息被commit，由于副本机制（replication）的存在，它就不会丢失。但是如果producer发送数据给broker后，遇到的网络问题而造成通信中断，那producer就无法判断该条消息是否已经提交（commit）。虽然Kafka无法确定网络故障期间发生了什么，但是producer可以retry多次，确保消息已经正确传输到broker中，所以目前Kafka实现的是at least once。

consumer从broker中读取消息后，可以选择commit，该操作会在Zookeeper中存下该consumer在该partition下读取的消息的offset。该consumer下一次再读该partition时会从下一条开始读取。如未commit，下一次读取的开始位置会跟上一次commit之后的开始位置相同。当然也可以将consumer设置为autocommit，即consumer一旦读取到数据立即自动commit。如果只讨论这一读取消息的过程，那Kafka是确保了exactly once, 但是如果由于前面producer与broker之间的某种原因导致消息的重复，那么这里就是at least once。

考虑这样一种情况，当consumer读完消息之后先commit再处理消息，在这种模式下，如果consumer在commit后还没来得及处理消息就crash了，下次重新开始工作后就无法读到刚刚已提交而未处理的消息，这就对应于at most once了。

读完消息先处理再commit。这种模式下，如果处理完了消息在commit之前consumer crash了，下次重新开始工作时还会处理刚刚未commit的消息，实际上该消息已经被处理过了，这就对应于at least once。

要做到exactly once就需要引入消息去重机制。

# 源码

### Product

并不是说从broker入手来探索了,我们会从一条消息开始生产出去,首先是生产端的源码

运行的流程,最核心的是两块,第一块要学习一下katka客户端是如何去设计一个非常优秀的生产级的保证高吞吐的一个缓冲机制

要深入的分析一下,生产端的Sender线程他的网络通信的模块,这个是我们绝对重点中的

### 客户端通信

重点,必须要搞清楚kafka客户端是如何通过网络通信把一批消息发送到broker上去的,这个里面对于网络通信的很多细节,我们需要去深入的扣一下网络通信的设置一些对应的参数,应对网络故障,人家是怎么来做的

### broker

消息会到katka broker那块去,我们要学习很多重点的地方,首先就是网络通信里的server端的处理架构应该如何来设计,以及server端的网络通信的细节,包括里面的底层网各通信对应的一些参数的设置

### 磁盘读写

深入的学习他的磁盘读写这块是如何来实现的,他的消息是如何写入磁盘的,磁盘的存储结构,怎么去使用os page cahe,怎么实现磁盘文件的顺序写,非常的优秀的,我们要来学习里面的这些东西

### 集群和高可用架构

多副本冗余以及高可用的架构设计, leader和follower是如何同步的,副本是如何传输的,另外就是这个过程中,他的各种offset是如何变更的,如果leader所在的broker故障了,是如何进行leader和follower的切换的,高可用的架构

### 负载均衡和伸缩架构

负载均衡以及伸缩架构,他是如何保证数据均匀的分布在集群的各个broker机器上的,负载均衡,如何进行topic的partition的扩容,让一个topic可以通过partition扩容来使用集群里更多的broker的机器资源,另外一个就是说broker扩容,如何通过加更多的broker机器来扩容集群的存储资源以及网络资源,

### 消费端

消费端的原理,每个消费组的主控节点是如何来选择的, group coordinator如何选择,consumer group leader如何选择,分区分配的方法,分布式消费的实现机制,拉取消息的原理, offset 提交的原理

# 生产者

## 生产者

### 生产者

既然要从katka生产端开始研究他的源码,就得先看一下他的生产端涉及到哪些东西KakaProducer,里面包含了核心的资源,包括线程资源以及网备资源,他主要是通过一些线程实现了消息的异步发送,批处理机制,维护了跟各个broker的网络连接,然后可以通过网络连接发送消息到broker去可以整个系统全局就通过唯一的一个katkaProducer来发送消息也是可以的,多线程并发安全的,发送的消息是通过ProducerRecord来进行封装的,代表了你要生产发送的一条消息,交给KafkaProducer来进行发送即可。

### 核心组件

(1)核心组件1: Partitioner,后面用来决定,你发送的每条消息是路由到Topic的哪个分

区里去的

(2)核心组件2: Metadata,这个是对于生产端来说非常核心的一个组件,他是用来从broker集群去拉取元数据的Topics ( Topic > Partitions (LeadertFollowers, ISR)),后面如果写消息到Topic,才知道这个Topic有哪些Partitions, Partition Leader所在的Broker

后面肯定会每隔一小段时间就再次发送请求刷新元数据, metadata.max.age.ms,默认是5分钟,默认每隔5分钟一定会强制刷新一下

还有就是我们猜测,在发送消息的时候,如果发现你要写入的某个Topic对应的元数据不在本地,那么他是不是肯定会通过这个组件,发送请求到broker尝试拉取这个topic对应的元数据,如果你在集群里增加了一台broker,也会涉及到元数据的变化

(3)核心参数:

每个请求的最大大小(1mb),

缓冲区的内存大小, 32mb

重试时间间隔,缓冲区：60s

填满之后的阻塞时间: 60s

请求超时时间:30s.

默认情况下,如果光光是考虑batch的机制的话,那么必须要等到足够多的消息打包成一个batch,才能通过request发送到broker上去;但是有一个问题,如果你发送了一条消息,但是等了很久都没有达到一个bath大小

所以说要设置一个linger.ms,如果在指定时间范围内,都没凑出来一个batch把这条消息发

送出去,那么到了这个inger.ms指定的时间,比如说5ms,如果5ms还没凑出来一个batch,

那么就必须立即把这个消息发送出去

(6)核心组件:网络通信的组件, NetworkClient,一个网络连接最多空闲多长时间(9分

钟),每个连接最多有几个request没收到响应(5个),重试连接的时间间隔(50ms), Socket

发送缓冲区大小(128kb), Socket接收缓冲区大小(32kb)

(7)核心组件: Sender线程,负责从缓冲区里获取消息发送到broker上去, request最大

大小(1mb), acks (1,只要leader写入成功就认为成功),重试次数(o,无重试),请求超时时间30s,线程类叫做 kafkathread ，线程名字叫做”kafka-producer-network -thread.”

### 生产消息分发策略

消息是kafka 最基本的数据单元，在kafaka 中，一条消息由key、value 两部分组成。

在发送一条消息时，我们可以指定这个key，那么producer 会根据key 和partition 机制来判断当前这条消息应该发送并存储到哪个partition 中。我们可以根据需要进行扩展producer 和partition 机制。

### 消息默认的分发机制

默认情况下，kafka 采用的时hash 取模的分区算法(DefaultPartitioner)。

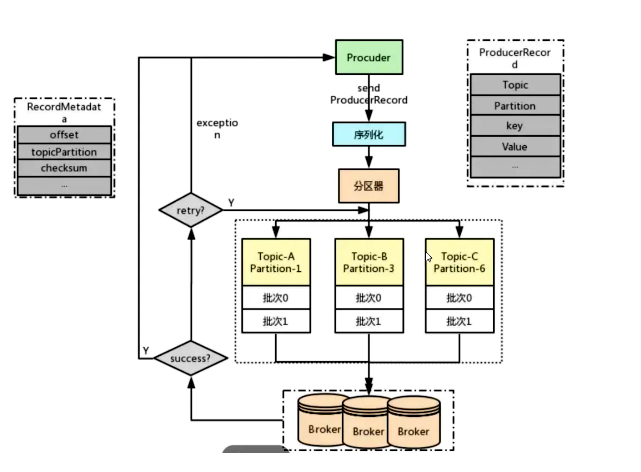
该默认的分区算法，如果key为null，则会随机分配一个分区，这个随机是在这个参数 “matadata.max.age.ms” 的时间范围内随机选择一个。对于这个时间段内，如果key为null，则只会发送到唯一的分区。这个值在默认情况下是10分钟更新一次。

关于Metadata类

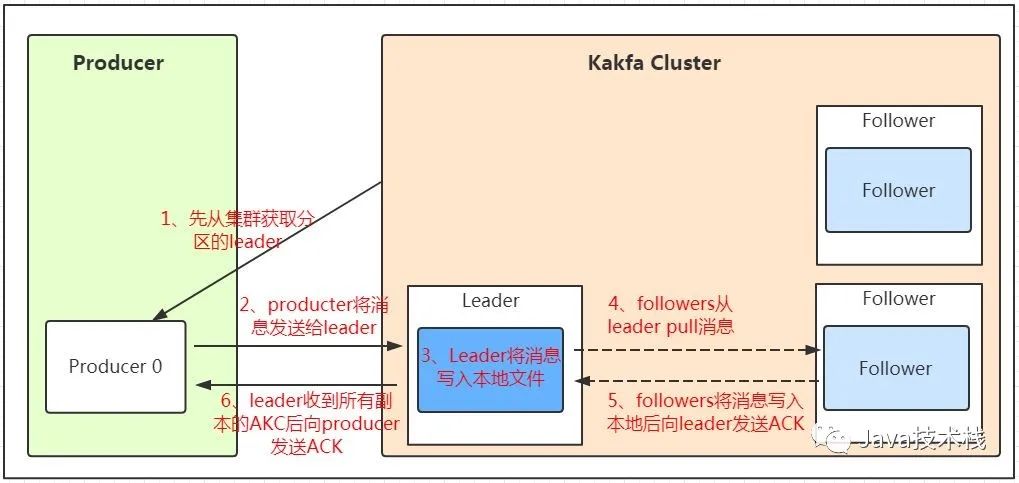
简单理解就是Topc 、 Partition 和 Broker 的映射关系，每个Topic 的每一个 Partition，需要知道对应的broker 列表是什么、Leader是谁、Follower是谁。这些信息都是存储在Metadata这个类的。

消费者的消费原理

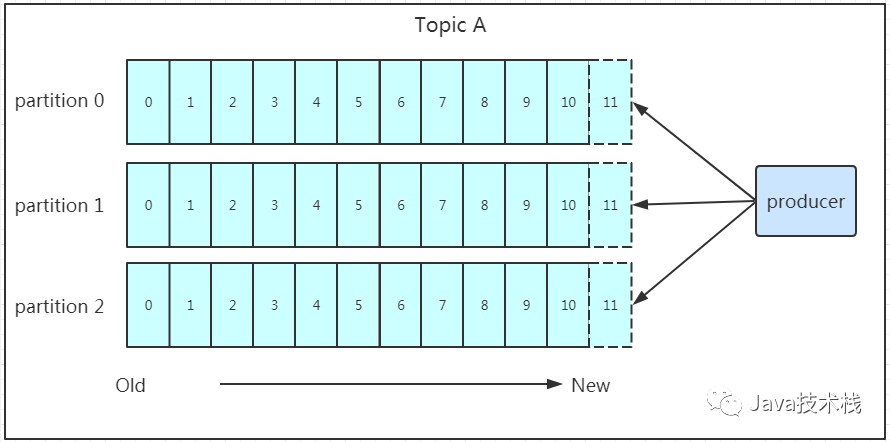
### 发送数据



Producer在写入数据的时候永远的找leader，不会直接将数据写入follower！那leader怎么找呢？



发送的流程就在图中已经说明了，就不单独在文字列出来了！需要注意的一点是，消息写入leader后，follower是主动的去leader进行同步的！producer采用push模式将数据发布到broker，每条消息追加到分区中，顺序写入磁盘，所以保证同一分区内的数据是有序的！写入示意图如下：



上面说到数据会写入到不同的分区，那kafka为什么要做分区呢？相信大家应该也能猜到，分区的主要目的是：

1、 方便扩展。因为一个topic可以有多个partition，所以我们可以通过扩展机器去轻松的应对日益增长的数据量。

2、 提高并发。以partition为读写单位，可以多个消费者同时消费数据，提高了消息的处理效率。

熟悉负载均衡的朋友应该知道，当我们向某个服务器发送请求的时候，服务端可能会对请求做一个负载，将流量分发到不同的服务器，那在kafka中，如果某个topic有多个partition，producer又怎么知道该将数据发往哪个partition呢？

kafka中有几个原则：

1、 partition在写入的时候可以指定需要写入的partition，如果有指定，则写入对应的partition。

2、 如果没有指定partition，但是设置了数据的key，则会根据key的值hash出一个partition。

3、 如果既没指定partition，又没有设置key，则会轮询选出一个partition。

保证消息不丢失是一个消息队列中间件的基本保证，那producer在向kafka写入消息的时候，怎么保证消息不丢失呢？

其实上面的写入流程图中有描述出来，那就是通过ACK应答机制！在生产者向队列写入数据的时候可以设置参数来确定是否确认kafka接收到数据，这个参数可设置的值为0、1、all。

0代表producer往集群发送数据不需要等到集群的返回，不确保消息发送成功。安全性最低但是效率最高。

1代表producer往集群发送数据只要leader应答就可以发送下一条，只确保leader发送成功。

all代表producer往集群发送数据需要所有的follower都完成从leader的同步才会发送下一条，确保leader发送成功和所有的副本都完成备份。安全性最高，但是效率最低。

最后要注意的是，如果往不存在的topic写数据，能不能写入成功呢？kafka会自动创建topic，分区和副本的数量根据默认配置都是1。

## 元数据的拉取

分析完毕了katkaProducer初始化的时候,都涉及到了哪些组件,每个组件对应的一些核心参数在源码里的注释和默认值,他的作用和效果,都进行了分析,到底会不会真实的去拉取集群的元数据呢?

wait(),释放锁,然后进入一个休眠等待再次被人唤醒获取锁的状态

此时如果有人获取锁之后,调用notifyAl1o,就会把之前调用wat(方法进入休眠的线程给唤醒,让他们再次尝试获取锁

在KatkaProducer初始化的时候,并没有真正的去某一个broker上去拉取元数据的,但是他肯定是对集群元数据做了一个初始化的,把你配置的那些broker地址转化为了Node,放在

Cluster对象实例里

List<Node>, Kafka Broker节点,一台机器

Listestring , unautorhizedopics,没有被授权访问的Topic的列表,就是kafka是可以支持权限控制的,如果你的客户端没有被授权访问某个Topic,那么就会放在这个列表里。

Map<TopicParittion, Partitioninfox> TopicPartition就代表了一个分区,里面就是他的toplc的名字,以及他在topic里的分区号; Partitininfo,就代表了分区的详细信息,属于哪个topic,分区号,每个分区都有多个副本, Leader在哪broker上, followers在哪些broker上, ISR列表,都在里面  
availablepartitionsyTopic, 个toplic哪些当前可用的分区,如果某个分区没有leader是存活的,此时那个分区就不可用了。

partitionsyNode,每个broker上放了哪些分区。

nodesByld, broker.id > Node

## 拦截器

### 拦截器

Kafka中的拦截器（Interceptor）是0.10.x.x版本引入的一个功能，一共有两种：

Kafka Producer端的拦截器

Kafka Consumer端的拦截器

本篇主要讲述的是Kafka Producer端的拦截器，它主要用来对消息进行拦截或者修改，也可以用于Producer的Callback回调之前进行相应的预处理。

### 使用

使用Kafka Producer端的拦截器非常简单，主要是实现ProducerInterceptor接口，此接口包含4个方法：

ProducerRecord<K, V> onSend(ProducerRecord<K, V> record)：Producer在将消息序列化和分配分区之前会调用拦截器的这个方法来对消息进行相应的操作。一般来说最好不要修改消息ProducerRecord的topic、key以及partition等信息，如果要修改，也需确保对其有准确的判断，否则会与预想的效果出现偏差。比如修改key不仅会影响分区的计算，同样也会影响Broker端日志压缩（Log Compaction）的功能。

void onAcknowledgement(RecordMetadata metadata, Exception exception)：在消息被应答（Acknowledgement）之前或者消息发送失败时调用，优先于用户设定的Callback之前执行。这个方法运行在Producer的IO线程中，所以这个方法里实现的代码逻辑越简单越好，否则会影响消息的发送速率。

void close()：关闭当前的拦截器，此方法主要用于执行一些资源的清理工作。

configure(Map<String, ?> configs)：用来初始化此类的方法，这个是ProducerInterceptor接口的父接口Configurable中的方法。

一般情况下只需要关注并实现onSend或者onAcknowledgement方法即可。下面我们来举个案例，通过onSend方法来过滤消息体为空的消息以及通过onAcknowledgement方法来计算发送消息的成功率。

### Demo

一般情况下只需要关注并实现onSend或者onAcknowledgement方法即可。下面我们来举个案例，通过onSend方法来过滤消息体为空的消息以及通过onAcknowledgement方法来计算发送消息的成功率

Demo

|  |
| --- |
| **public class** ProducerInterceptorDemo **implements** ProducerInterceptor<String,String> {  **private volatile long sendSuccess** = 0;  **private volatile long sendFailure** = 0;   @Override  **public** ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {  **if**(record.value().length()<=0)  **return null**;  **return** record;  }   @Override  **public void** onAcknowledgement(RecordMetadata metadata, Exception exception) {  **if** (exception == **null**) {  **sendSuccess**++;  } **else** {  **sendFailure** ++;  }  }   @Override  **public void** close() {  **double** successRatio = (**double**)**sendSuccess** / (**sendFailure** + **sendSuccess**);  System.***out***.println(**"[INFO] 发送成功率="**+String.*format*(**"%f"**, successRatio \* 100)+**"%"**);  }   @Override  **public void** configure(Map<String, ?> configs) {} } |

自定义的ProducerInterceptorDemo类实现之后就可以在Kafka Producer的主程序中指定，示例代码如下：

|  |
| --- |
| **public class** TestInterceptro {  **public static final** String ***brokerList*** = **"localhost:9092"**;  **public static final** String ***topic*** = **"hidden-topic"**;   **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  Properties properties = **new** Properties();  properties.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  properties.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  properties.put(**"bootstrap.servers"**, ***brokerList***);  properties.put(**"interceptor.classes"**, **"com.hidden.producer.ProducerInterceptorDemo"**);   Producer<String, String> producer = **new** KafkaProducer<String, String>(properties);   **for**(**int** i=0;i<100;i++) {  ProducerRecord<String, String> producerRecord = **new** ProducerRecord<String, String>(***topic***, **"msg-"** + i);  producer.send(producerRecord).get();  }  producer.close();  } } |

Kafka Producer不仅可以指定一个拦截器，还可以指定多个拦截器以形成拦截链，这个拦截链会按照其中的拦截器的加入顺序一一执行。比如上面的程序多添加一个拦截器，示例如下：

properties.put("interceptor.classes", "com.hidden.producer.ProducerInterceptorDemo,com.hidden.producer.ProducerInterceptorDemoPlus");

这样Kafka Producer会先执行拦截器ProducerInterceptorDemo，之后再执行ProducerInterceptorDemoPlus。

## 发送方式

### 消息发送方式

kafka有同步（sync）、异步（async）以及oneway这三种发送方式，某些概念上区分也可以分为同步和异步两种，同步和异步的发送方式通过“producer.type”参数指定，而oneway由“request.require.acks”参数指定。

### Sync vs async

producer.type的默认值是sync，即同步的方式。

这个参数指定了在后台线程中消息的发送方式是同步的还是异步的。如果设置成异步的模式，可以运行生产者以batch的形式push数据，这样会极大的提高broker的性能，但是这样会增加丢失数据的风险。

对于异步模式，还有4个配套的参数，

queue.buffering.max.ms 5000

启用异步模式时，producer缓存消息的时间。比如我们设置成1000时，它会缓存1s的数据再一次发送出去，这样可以极大的增加broker吞吐量，但也会造成时效性的降低。

queue.buffering.max.messages 10000

启用异步模式时，producer缓存队列里最大缓存的消息数量，如果超过这个值，producer就会阻塞或者丢掉消息。

queue.enqueue.timeout.ms -1

当达到上面参数时producer会阻塞等待的时间。如果设置为0，buffer队列满时producer不会阻塞，消息直接被丢掉；若设置为-1，producer会被阻塞，不会丢消息。

batch.num.messages 200

启用异步模式时，一个batch缓存的消息数量。达到这个数值时，producer才会发送消息。（每次批量发送的数量）

以batch的方式推送数据可以极大的提高处理效率，kafka producer可以将消息在内存中累计到一定数量后作为一个batch发送请求。batch的数量大小可以通过producer的参数（batch.num.messages）控制。通过增加batch的大小，可以减少网络请求和磁盘IO的次数，当然具体参数设置需要在效率和时效性方面做一个权衡。在比较新的版本中还有batch.size这个参数。

### Oneway

oneway是只顾消息发出去而不管死活，消息可靠性最低，但是低延迟、高吞吐，这种对于某些完全对可靠性没有要求的场景还是适用的，即request.required.acks设置为0。

### 一般配置

对于sync的发送方式：

producer.type=sync

request.required.acks=1

对于async的发送方式：

producer.type=async

request.required.acks=1

queue.buffering.max.ms=5000

queue.buffering.max.messages=10000

queue.enqueue.timeout.ms = -1

batch.num.messages=200

对于oneway的发送发送：

producer.type=async

request.required.acks=0

### 总结

producers可以一步的并行向kafka发送消息，但是通常producer在发送完消息之后会得到一个响应，返回的是offset值或者发送过程中遇到的错误。这其中有个非常重要的参数“request.required.acks”，这个参数决定了producer要求leader partition收到确认的副本个数，如果acks设置为0，表示producer不会等待broker的相应，所以，producer无法知道消息是否发生成功，这样有可能导致数据丢失，但同时，acks值为0会得到最大的系统吞吐量。若acks设置为1，表示producer会在leader partition收到消息时得到broker的一个确认，这样会有更好的可靠性，因为客户端会等待知道broker确认收到消息。若设置为-1，producer会在所有备份的partition收到消息时得到broker的确认，这个设置可以得到最高的可靠性保证。

## 序列化

### 序列化

Kafka Producer在发送消息时必须配置的参数为：

bootstrap.servers、

key.serializer

value.serializer。

序列化操作是在拦截器（Interceptor）执行之后并且在分配分区(partitions)之前执行的。

### Demo

|  |
| --- |
| **public class** ProducerJavaDemo {  **public static final** String ***brokerList*** = **"192.168.0.2:9092,192.168.0.3:9092,192.168.0.4:9092"**;  **public static final** String ***topic*** = **"hidden-topic"**;   **public static void** main(String[] args) {  Properties properties = **new** Properties();  properties.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  properties.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  properties.put(**"client.id"**, **"hidden-producer-client-id-1"**);  properties.put(**"bootstrap.servers"**, ***brokerList***);   Producer<String,String> producer = **new** KafkaProducer<String,String>(properties);   **while** (**true**) {  String message = **"kafka\_message-"** + **new** Date().getTime() + **"-edited by hidden.zhu"**;  ProducerRecord<String, String> producerRecord = **new** ProducerRecord<String, String>(***topic***,message);  **try** {  Future<RecordMetadata> future = producer.send(producerRecord, **new** Callback() {  **public void** onCompletion(RecordMetadata metadata, Exception exception) {  System.***out***.print(metadata.offset()+**" "**);  System.***out***.print(metadata.topic()+**" "**);  System.***out***.println(metadata.partition());  }  });  } **catch** (Exception e) {  e.printStackTrace();  }  **try** {  TimeUnit.***MILLISECONDS***.sleep(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } } |

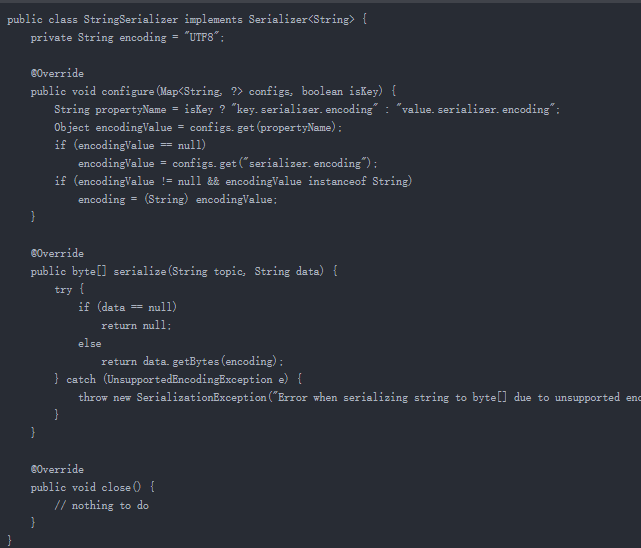
上面的程序中使用的是Kafka客户端自带的org.apache.kafka.common.serialization.StringSerializer，除了用于String类型的序列化器之外还有：ByteArray、ByteBuffer、Bytes、Double、Integer、Long这几种类型，它们都实现了org.apache.kafka.common.serialization.Serializer接口，此接口有三种方法：

public void configure(Map<String, ?> configs, boolean isKey)：用来配置当前类。

public byte[] serialize(String topic, T data)：用来执行序列化。

public void close()：用来关闭当前序列化器。一般情况下这个方法都是个空方法，如果实现了此方法，必须确保此方法的幂等性，因为这个方法很可能会被KafkaProducer调用多次。

下面我们来看看Kafka中org.apache.kafka.common.serialization.StringSerializer的具体实现，源码如下：



首先看下StringSerializer中的configure(Map<String, ?> configs, boolean isKey)方法，这个方法的执行是在创建KafkaProducer实例的时候调用的，即执行代码Producer<String,String> producer = new KafkaProducer<String,String>(properties)时调用，主要用来确定编码类型，不过一般key.serializer.encoding或serializer.encoding都不会配置，更确切的来说在Kafka Producer Configs列表里都没有此项，所以一般情况下encoding的值就是UTF-8。serialize(String topic, String data)方法非常的直观，就是将String类型的data转为byte[]类型即可。

如果Kafka自身提供的诸如String、ByteArray、ByteBuffer、Bytes、Double、Integer、Long这些类型的Serializer都不能满足需求，读者可以选择使用如Avro、JSON、Thrift、ProtoBuf或者Protostuff等通用的序列化工具来实现，亦或者是使用自定义类型的Serializer来实现。下面就以一个简单的例子来介绍下如何自定义类型的使用方法。

### 自定义序列化

数据

|  |
| --- |
| **public class** Company {  **private** String **name**;  **private** String **address**;  *//省略Getter, Setter, Constructor & toString方法* } |

示例中只修改了value.serializer,而key.serializer和value.serializer没有什么区别，如果有真实需要，修改以下也未尝不可。

自定义序列化类

|  |
| --- |
| **public class** DemoSerializer **implements** Serializer<Company> {  **public void** configure(Map<String, ?> configs, **boolean** isKey) {}  **public byte**[] serialize(String topic, Company data) {  **if** (data == **null**) {  **return null**;  }  **byte**[] name, address;  **try** {  **if** (data.getName() != **null**) {  name = data.getName().getBytes(**"UTF-8"**);  } **else** {  name = **new byte**[0];  }  **if** (data.getAddress() != **null**) {  address = data.getAddress().getBytes(**"UTF-8"**);  } **else** {  address = **new byte**[0];  }  ByteBuffer buffer = ByteBuffer.*allocate*(4+4+name.**length** + address.**length**);  buffer.putInt(name.**length**);  buffer.put(name);  buffer.putInt(address.**length**);  buffer.put(address);  **return** buffer.array();  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  **return new byte**[0];  }  **public void** close() {} } |

使用;

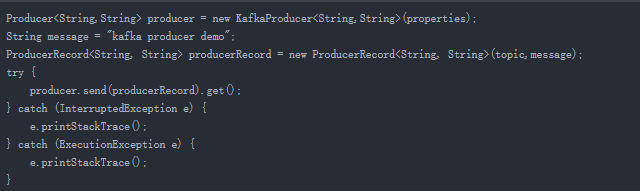
|  |
| --- |
| properties.put("value.serializer", "com.hidden.client.DemoSerializer");  //记得也要将相应的String类型改为Company类型，如：  //Producer<String,Company> producer = new KafkaProducer<String,Company>(properties);  //Company company = new Company();  //company.setName("hidden.cooperation-" + new Date().getTime());  //company.setAddress("Shanghai, China");  //ProducerRecord<String, Company> producerRecord = new ProducerRecord<String, Company>(topic,company); |

## 分区分配计算（分区器Partitions）

### 引入

KafkaProducer在调用send方法发送消息至broker的过程中，首先是经过拦截器Inteceptors处理，然后是经过序列化Serializer处理，之后就到了Partitions阶段，即分区分配计算阶段。

在某些应用场景下，业务逻辑需要控制每条消息落到合适的分区中，有些情形下则只要根据默认的分配规则即可。在KafkaProducer计算分配时，首先根据的是ProducerRecord中的partition字段指定的序号计算分区。读者有可能刚睡醒，看到这个ProducerRecord似曾相识，没有关系，先看段Kafka生产者的示例片段：



没错，ProducerRecord只是一个封装了消息的对象而已，ProducerRecord一共有5个成员变量，即：

private final String topic;//所要发送的topic

private final Integer partition;//指定的partition序号

private final Headers headers;//一组键值对，与RabbitMQ中的headers类似，kafka0.11.x版本才引入的一个属性

private final K key;//消息的key

private final V value;//消息的value,即消息体

private final Long timestamp;//消息的时间戳，可以分为Create\_Time和LogAppend\_Time之分，这个以后的文章中再表。

在KafkaProducer的源码（1.0.0）中，计算分区时调用的是下面的partition()方法：

|  |
| --- |
| **private int** partition(ProducerRecord<K, V> record, **byte**[] serializedKey, **byte**[] serializedValue, Cluster cluster) {  Integer partition = record.partition();  **return** partition != **null** ?  partition :  partitioner.partition(record.topic(), record.key(), serializedKey, record.value(), serializedValue, cluster); } |

可以看出的确是先判断有无指明ProducerRecord的partition字段，如果没有指明，则再进一步计算分区。上面这段代码中的partitioner在默认情况下是指Kafka默认实现的org.apache.kafka.clients.producer.DefaultPartitioner，其partition()方法实现如下：

|  |
| --- |
| **public int** partition(String topic, Object key, **byte**[] keyBytes, Object value, **byte**[] valueBytes, Cluster cluster) {  List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);  **int** numPartitions = partitions.size();  **if** (keyBytes == **null**) {  **int** nextValue = nextValue(topic);  List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);  **if** (availablePartitions.size() > 0) {  **int** part = Utils.*toPositive*(nextValue) % availablePartitions.size();  **return** availablePartitions.get(part).partition();  } **else** {  *// no partitions are available, give a non-available partition* **return** Utils.*toPositive*(nextValue) % numPartitions;  }  } **else** {  *// hash the keyBytes to choose a partition* **return** Utils.*toPositive*(Utils.*murmur2*(keyBytes)) % numPartitions;  } }  **private int** nextValue(String topic) {  AtomicInteger counter = topicCounterMap.get(topic);  **if** (**null** == counter) {  counter = **new** AtomicInteger(ThreadLocalRandom.*current*().nextInt());  AtomicInteger currentCounter = topicCounterMap.putIfAbsent(topic, counter);  **if** (currentCounter != **null**) {  counter = currentCounter;  }  }  **return** counter.getAndIncrement(); } |

由上源码可以看出partition的计算方式：

如果key为null，则按照一种轮询的方式来计算分区分配

如果key不为null则使用称之为murmur的Hash算法（非加密型Hash函数，具备高运算性能及低碰撞率）来计算分区分配。

### 自定义

KafkaProducer中还支持自定义分区分配方式，与

1、实现org.apache.kafka.clients.producer.Partitioner接口

2然后在KafkaProducer的配置中指定partitioner.class为对应的自定义分区器（Partitioners）即可，即：

properties.put("partitioner.class","com.hidden.partitioner.DemoPartitioner");

自定义DemoPartitioner主要是实现Partitioner接口的public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster)的方法。DemoPartitioner稍微修改了下DefaultPartitioner的计算方式，详细参考如下：

|  |
| --- |
| **public class** DemoPartitioner **implements** Partitioner {  **private final** AtomicInteger **atomicInteger** = **new** AtomicInteger(0);  @Override  **public void** configure(Map<String, ?> configs) {}  @Override  **public int** partition(String topic, Object key, **byte**[] keyBytes, Object value, **byte**[] valueBytes, Cluster cluster) {  List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);  **int** numPartitions = partitions.size();  **if** (**null** == keyBytes || keyBytes.**length**<1) {  **return atomicInteger**.getAndIncrement() % numPartitions;  }  *//借用String的hashCode的计算方式* **int** hash = 0;  **for** (**byte** b : keyBytes) {  hash = 31 \* hash + b;  }  **return** hash % numPartitions;  }   @Override  **public void** close() {} } |

这个自定义分区器的实现比较简单，读者可以根据自身业务的需求来灵活实现分配分区的计算方式，比如：一般大型电商都有多个仓库，可以将仓库的名称或者ID作为Key来灵活的记录商品信息。

## 消息发送保障机制

### 消息发送可靠性

当producer向leader发送数据时，可以通过request.required.acks参数来设置数据可靠性的级别：

1（默认）：这意味着producer在ISR中的leader已成功收到数据并得到确认。如果leader宕机了，则会丢失数据。

0：这意味着producer无需等待来自broker的确认而继续发送下一批消息。这种情况下数据传输效率最高，但是数据可靠性确是最低的。

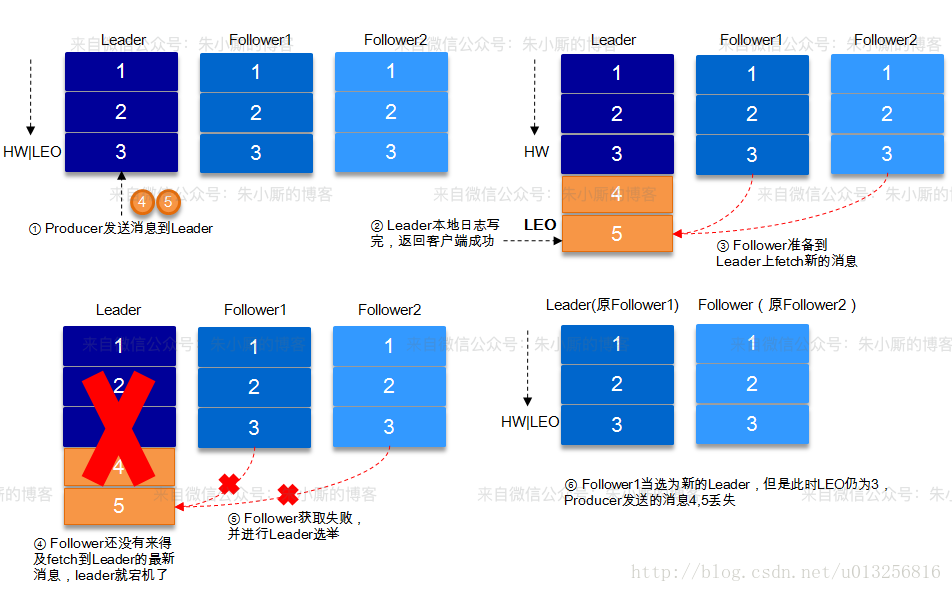
-1：producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。但是这样也不能保证数据不丢失，比如当ISR中只有leader时（前面ISR那一节讲到，ISR中的成员由于某些情况会增加也会减少，最少就只剩一个leader），这样就变成了acks=1的情况。

如果要提高数据的可靠性，在设置request.required.acks=-1的同时，也要min.insync.replicas这个参数(可以在broker或者topic层面进行设置)的配合，这样才能发挥最大的功效。min.insync.replicas这个参数设定ISR中的最小副本数是多少，默认值为1，当且仅当request.required.acks参数设置为-1时，此参数才生效。如果ISR中的副本数少于min.insync.replicas配置的数量时，客户端会返回异常：org.apache.kafka.common.errors.NotEnoughReplicasExceptoin: Messages are rejected since there are fewer in-sync replicas than required。

接下来对acks=1和-1的两种情况进行详细分析：

### request.required.acks=1

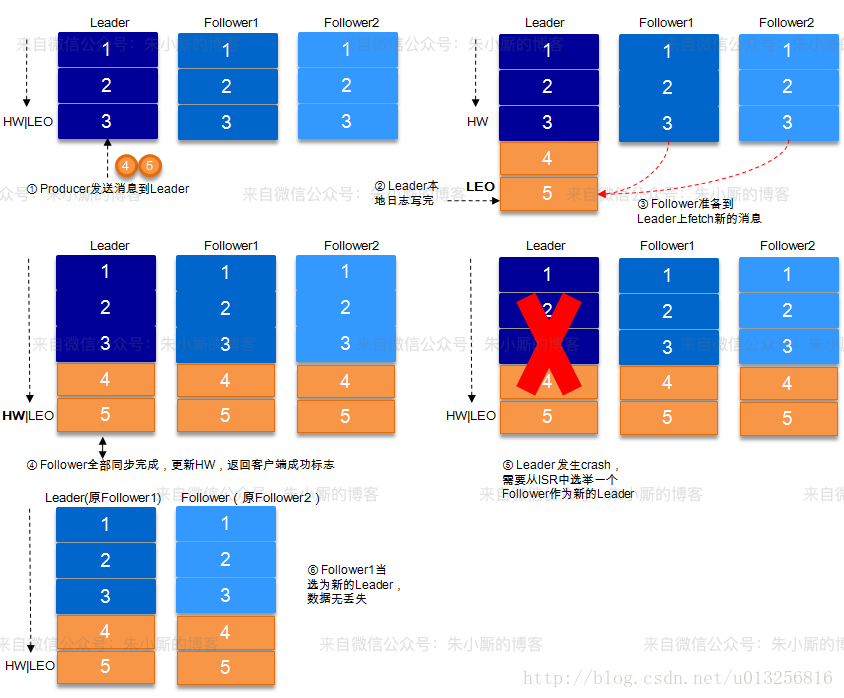
producer发送数据到leader，leader写本地日志成功，返回客户端成功；此时ISR中的副本还没有来得及拉取该消息，leader就宕机了，那么此次发送的消息就会丢失。



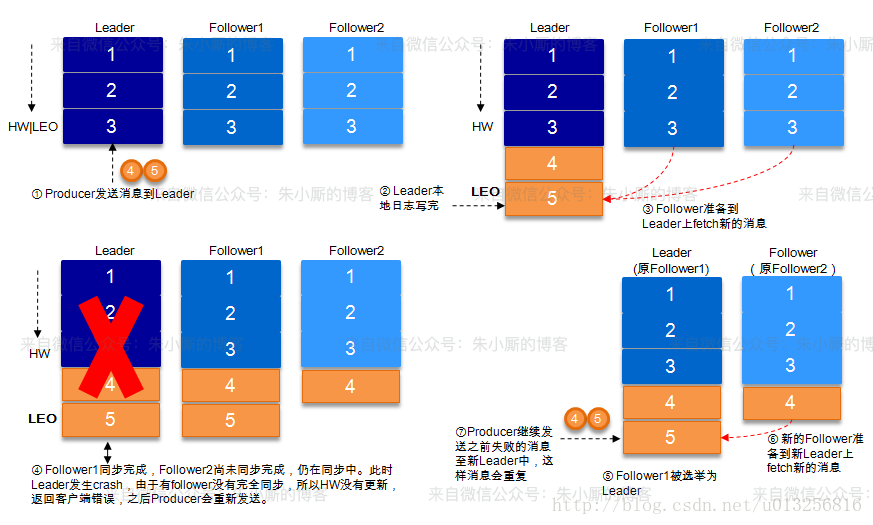
### request.required.acks=-1

同步（Kafka默认为同步，即producer.type=sync）的发送模式，replication.factor>=2且min.insync.replicas>=2的情况下，不会丢失数据。

有两种典型情况。acks=-1的情况下（如无特殊说明，以下acks都表示为参数request.required.acks），数据发送到leader, ISR的follower全部完成数据同步后，leader此时挂掉，那么会选举出新的leader，数据不会丢失。



acks=-1的情况下，数据发送到leader后 ，部分ISR的副本同步，leader此时挂掉。比如follower1和follower2都有可能变成新的leader, producer端会得到返回异常，producer端会重新发送数据，数据可能会重复。



当然上图中如果在leader crash的时候，follower2还没有同步到任何数据，而且follower2被选举为新的leader的话，这样消息就不会重复。

# 消费者的消费原理

## 消费

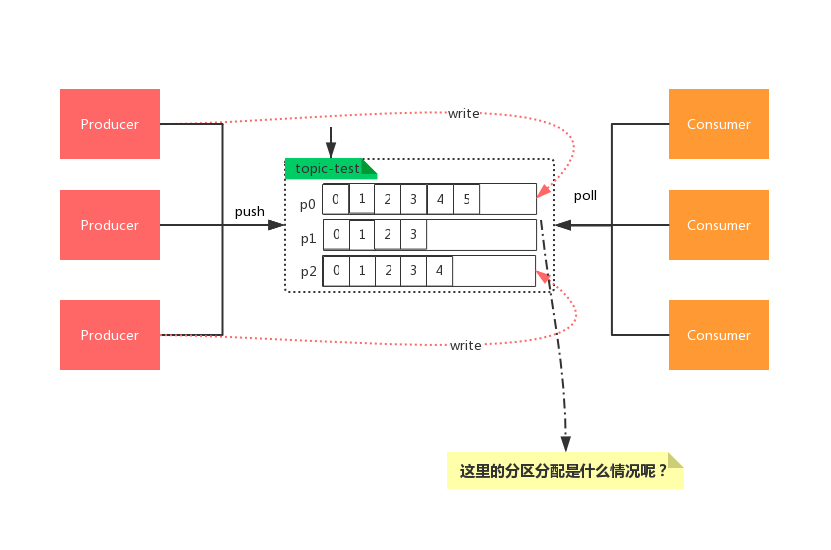
在实际的生产过程中，每个topic 都会有多个partitions。

多个partitions的好处在于：

1、一方面能够对broker 上的数据进行分片有效减少了消息的容量从而提升io性能。

2、另一方面，为了提高消费端的消费能力，一般会通过多个consumer去消费同一个topic，也就是消费端的负载均衡机制。

kafka存在consumer group的概念，组内的所有消费者协调在一起来消费订阅主题的所有分区。当然每一个分区只能由同一个消费组内的consumer 来消费，那么同一个consumer group 里面的consumer 是怎么分配该消费哪个分区的数据呢？如下图：



### 演示

a、环境演示

通过命令的方式创建3 分区 3 副本的topic

sh kafka-topics.sh --create --zookeeper 192.168.131.3:2181 --replication-factor 3 --partitions 3 --topic replis-test

登录三个消费者客户端，配置group 一样

sh kafka-console-consumer.sh --bootstrap-server 192.168.131.4:9092，192.168.131.5:9092，192.168.131.6:9092 --topic replis-test --from-beginning --group test\_group1

通过生产者发批量发送消息，记得用自己的随机算法来操作（利用key为null随机，不要用默认的消息分区算法，默认的不方便测试）然后批量发送消息，可以看到消息的确被分发到了不同的消费端

b、测试

3个partition 对应3consumer

结果：每个consumer 会消费一个分区

3个partition 对应2个consumer

结果：consumer1会消费partition0/partition1分区，consumer2会消费partition2分区

3个partition 对应4个或以上consumer

结果：任然只有3个consumer对应3个partition，其他的consumer无法消费消息

**c、结论**

**1、如果consumer 比 partition ,多是浪费，因为kafka的设计是在一个partition上是不允许并发的。所以consumer 数不要大于partition数**

**2、如果consumer 比partition 少,** **一个consumer会对应于多个partition，这里主要合理分配consumer数量的整数倍，否则会导致partition里面的数据被取的不均匀。最好partition数量是consumer数量的整数倍，所以partition 数量很重要，比如取18 就很容易设定consumer 数量。**

**3、如果consumer从多个partition读取到数据，不保证数据间的顺序性**

kafka只保证在一个partition上数据是有序的，但多个partition，根据你读的顺序会有不同

4、增减consumer，broker，partition会导致rebalance，所以rebalance后consumer对应的partition会发生变化

## 消费者分区分配策略

### 触发rebalance分配操作

1、同一个consumer group 内新增了消费者

2、消费者离开当前所属的consumer group，比如主机停机或宕机等

2、topic 新增了分区（也就是分区熟练发生了变化）

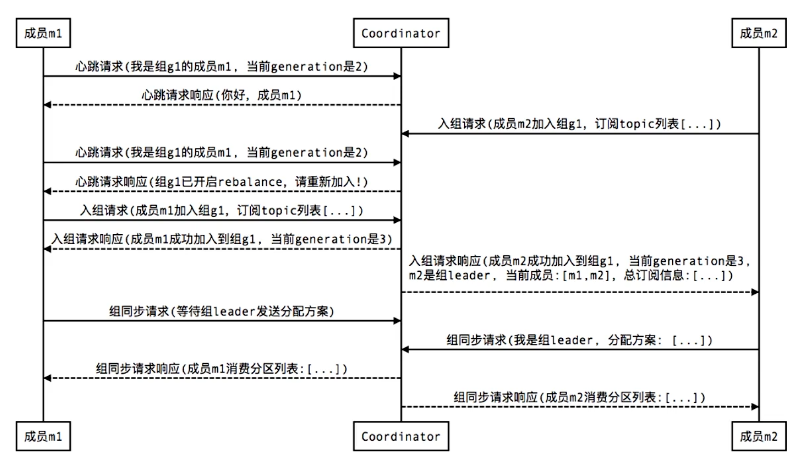
### 分配策略

Range（默认，范围）

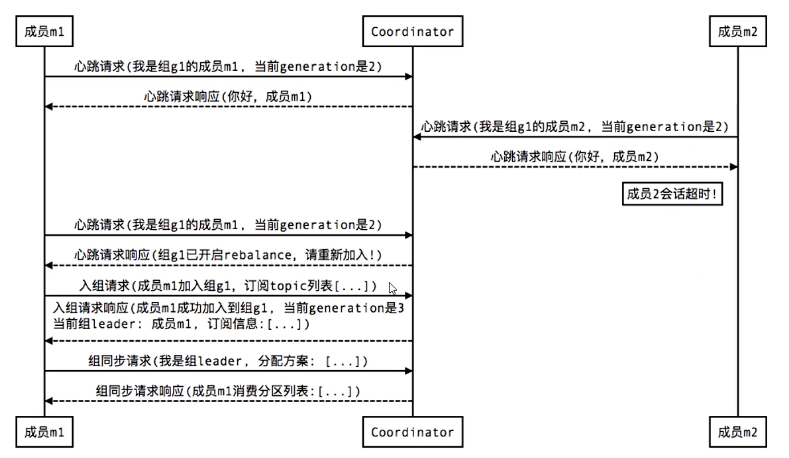
RoundRobin（轮询）

StickyAssignor（沾性）

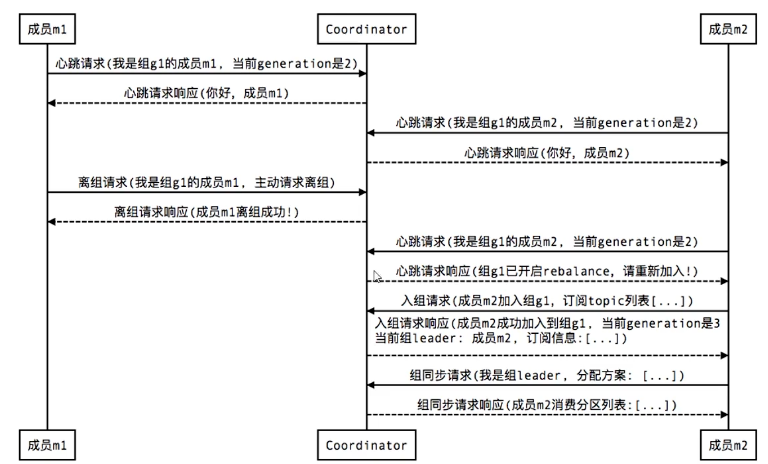
### 新成员加组

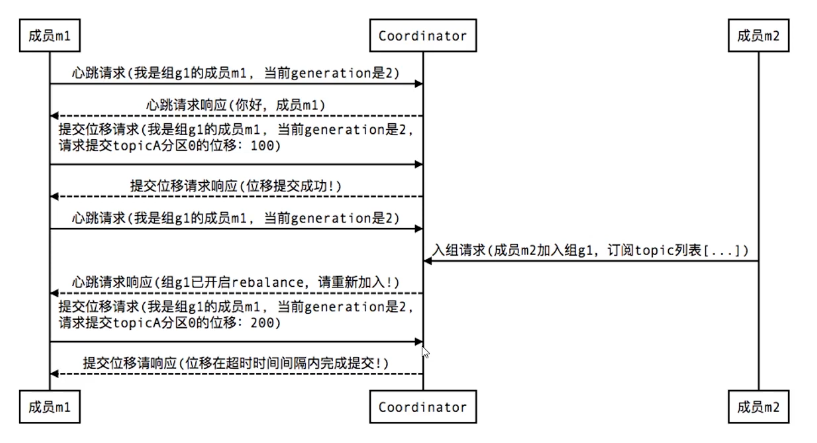


### 崩溃



### 主动离组





RangeAssignor：Range 策略是对每个主题而言的，首先对同一个主题里面的分区按照序号进行排序，并对消费者按照字符顺序进行排序。

假设 n = 分区数 / 消费数 ； m = 分区数 % 消费者数量

那么前 m 个消费者每个分配 n + 1 个分区

后面的（消费者数量 - m）个消费者每个分配n个分区

例如：11个分区，3个消费者

A1-0 将消费 0,1,2,3 分区

A2-0 将消费 4,5,6,7 分区

A3-0 将消费 8,9,10 分区

结果：A1-0 和 A2-0 多消费一个分区，没什么

例如：2个主题（T1和T2），分别有10个分区

A1-0 将消费T1主题的0,1,2,3 分区，以及T2主题的0,1,2,3分区

A2-0 将消费T1主题的4,5,6 分区，以及T2主题的4,5,6 分区

A3-0 将消费 T1主题的7,8,9 分区，以及T2主题的7,8,9 分区

结果：A1-0 比其他分区多了2个分区，这就是RangeAssignor很明显的弊端。

RoundRobinAssignor

轮询分区策略就是：

把所有partition 和所有consumer线程都列出来，然后按照hashcode进行排序

最后通过轮询算法分配partition给消费线程。

如果所有consumer实例的订阅是相同的，那么partition会均匀分布。

使用轮询分区策略，必须满足两个条件：

1、每个主题的消费者实例具有相同数量的流

2、每个消费者订阅的主题必须是相同的。

StrickyAssignor

kafka在0.11.x 版本支持了StrickyAssignor，翻译过来就是粘带策略，它有两个目的

1、分区的分配尽可能的均匀

2、分区的分配尽可能和上次分配保持相同

当两者冲突时，第一个目标优先于第二目标

假设消费组有3个消费者，C0、C1、C3，它们分别订阅了4个Topic（t0，t1，t2，t3），并且每个主题有两个分区（p0,p1），也就是说，整个消费组订阅了8个分区：t0p0、t0p1、t1p0、t1p1、t2p0、t2p1、t3p0、t3p1

那么最终的分配场景结果为：

C0: t0p0 t1p1 t3p0

C1: t0p1 t2p0 t3p1

C2: t1p0 t2p1

这样的分配策略类似轮询策略，但是其实不是，因为如果C1 这个消费者挂了，就必然会造成重新分区（reblance），如果是轮询，那么结果应该是

C0: t0p0 t1p1 t2p0 t3p0

C2: t0p1 t2p0 t2p1 t3p1

但是不是如此，因为他要满足粘带策略，所以他会满足 “分区的分配尽可能和上次分配保持相同”，所以分配结果是

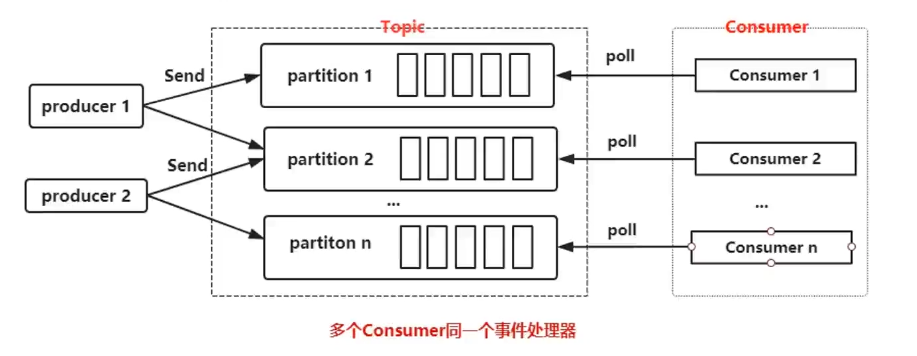
C0: t0p0 t1p1 t3p0 + t2p0

C2: t1p0 t2p1 + t0p1 t3p1

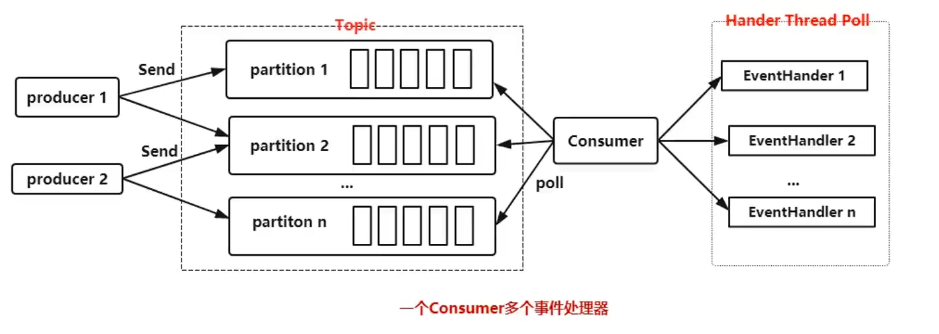
结论：也就是说保留了上一次的分配结果，并且把原挂掉的C1分区分配给了C0和C2，这中策略的好处，就是分区发生变化的时候，由于分区的沾性，减少了不要的分区移动。

## Consumer的多线程消费

一个线程第一个consumer。



|  |
| --- |
| **public class** ConsumerThreadSample {  **private final static** String ***TOPIC\_NAME***=**"jiangzh-topic"**;   */\*  这种类型是经典模式，每一个线程单独创建一个KafkaConsumer，用于保证线程安全  \*/* **public static void** main(String[] args) **throws** InterruptedException {  KafkaConsumerRunner r1 = **new** KafkaConsumerRunner();  Thread t1 = **new** Thread(r1);   t1.start();   Thread.*sleep*(15000);   r1.shutdown();  }   **public static class** KafkaConsumerRunner **implements** Runnable{  **private final** AtomicBoolean **closed** = **new** AtomicBoolean(**false**);  **private final** KafkaConsumer **consumer**;   **public** KafkaConsumerRunner() {  Properties props = **new** Properties();  props.put(**"bootstrap.servers"**, **"192.168.220.128:9092"**);  props.put(**"group.id"**, **"test"**);  props.put(**"enable.auto.commit"**, **"false"**);  props.put(**"auto.commit.interval.ms"**, **"1000"**);  props.put(**"session.timeout.ms"**, **"30000"**);  props.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  props.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);   **consumer** = **new** KafkaConsumer<>(props);   TopicPartition p0 = **new** TopicPartition(***TOPIC\_NAME***, 0);  TopicPartition p1 = **new** TopicPartition(***TOPIC\_NAME***, 1);   **consumer**.assign(Arrays.*asList*(p0,p1));  }    **public void** run() {  **try** {  **while**(!**closed**.get()) {  *//处理消息* ConsumerRecords<String, String> records = **consumer**.poll(Duration.*ofMillis*(10000));   **for** (TopicPartition partition : records.partitions()) {  List<ConsumerRecord<String, String>> pRecord = records.records(partition);  *// 处理每个分区的消息* **for** (ConsumerRecord<String, String> record : pRecord) {  System.***out***.printf(**"patition = %d , offset = %d, key = %s, value = %s%n"**,  record.partition(),record.offset(), record.key(), record.value());  }   *// 返回去告诉kafka新的offset* **long** lastOffset = pRecord.get(pRecord.size() - 1).offset();  *// 注意加1* **consumer**.commitSync(Collections.*singletonMap*(partition, **new** OffsetAndMetadata(lastOffset + 1)));  }   }  }**catch**(WakeupException e) {  **if**(!**closed**.get()) {  **throw** e;  }  }**finally** {  **consumer**.close();  }  }   **public void** shutdown() {  **closed**.set(**true**);  **consumer**.wakeup();  }  } } |



一个consumer，使用线程池异步处理。

适合流数据处理。

|  |
| --- |
| **public class** ConsumerRecordThreadSample {  **private final static** String ***TOPIC\_NAME*** = **"jiangzh-topic"**;   **public static void** main(String[] args) **throws** InterruptedException {  String brokerList = **"192.168.220.128:9092"**;  String groupId = **"test"**;  **int** workerNum = 5;   CunsumerExecutor consumers = **new** CunsumerExecutor(brokerList, groupId, ***TOPIC\_NAME***);  consumers.execute(workerNum);   Thread.*sleep*(1000000);   consumers.shutdown();   }   *// Consumer处理* **public static class** CunsumerExecutor{  **private final** KafkaConsumer<String, String> **consumer**;  **private** ExecutorService **executors**;   **public** CunsumerExecutor(String brokerList, String groupId, String topic) {  Properties props = **new** Properties();  props.put(**"bootstrap.servers"**, brokerList);  props.put(**"group.id"**, groupId);  props.put(**"enable.auto.commit"**, **"true"**);  props.put(**"auto.commit.interval.ms"**, **"1000"**);  props.put(**"session.timeout.ms"**, **"30000"**);  props.put(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  props.put(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**);  **consumer** = **new** KafkaConsumer<>(props);  **consumer**.subscribe(Arrays.*asList*(topic));  }   **public void** execute(**int** workerNum) {  **executors** = **new** ThreadPoolExecutor(workerNum, workerNum, 0L, TimeUnit.***MILLISECONDS***,  **new** ArrayBlockingQueue<>(1000), **new** ThreadPoolExecutor.CallerRunsPolicy());   **while** (**true**) {  ConsumerRecords<String, String> records = **consumer**.poll(200);  **for** (**final** ConsumerRecord record : records) {  **executors**.submit(**new** ConsumerRecordWorker(record));  }  }  }   **public void** shutdown() {  **if** (**consumer** != **null**) {  **consumer**.close();  }  **if** (**executors** != **null**) {  **executors**.shutdown();  }  **try** {  **if** (!**executors**.awaitTermination(10, TimeUnit.***SECONDS***)) {  System.***out***.println(**"Timeout.... Ignore for this case"**);  }  } **catch** (InterruptedException ignored) {  System.***out***.println(**"Other thread interrupted this shutdown, ignore for this case."**);  Thread.*currentThread*().interrupt();  }  }    }   *// 记录处理* **public static class** ConsumerRecordWorker **implements** Runnable {   **private** ConsumerRecord<String, String> **record**;   **public** ConsumerRecordWorker(ConsumerRecord record) {  **this**.**record** = record;  }   @Override  **public void** run() {  *// 假如说数据入库操作* System.***out***.println(**"Thread - "**+ Thread.*currentThread*().getName());  System.***err***.printf(**"patition = %d , offset = %d, key = %s, value = %s%n"**,  **record**.partition(), **record**.offset(), **record**.key(), **record**.value());  }   } } |

### Consimer的限流

## 反序列化

# Broker

## Replica

### 简介

Kafka从0.8.x版本开始引入副本机制，这样可以极大的提高集群的可靠性和稳定性。不过这也使得Kafka变得更加复杂起来，失效副本就是所要面临的一个难题。

通常情况下，Kafka中的每个分区（partition）都会分配多个副本（replica），具体的副本数量由Broker级别参数default.replication.factor（默认大小为1）指定，也可以在创建topic的时候通过 –replication-factor ${num} 显式指定副本的数量（副本因子）。一般情况下，将前者default.replication.factor设置为大于1的值，这样在参数auto.create.topic.enable为true的时候，自动创建的topic会根据default.replication.factor的值来创建副本数；或者更加通用的做法是使用后者而指定大于1的副本数。

每个分区的多个副本称之为AR（assigned replicas），包含至多一个leader副本和多个follower副本。与AR对应的另一个重要的概念就是ISR（in-sync replicas），ISR是指与leader副本保持同步状态的副本集合，当然leader副本本身也是这个集合中的一员。而ISR之外，也就是处于同步失败或失效状态的副本，副本对应的分区也就称之为同步失效分区，即under-replicated分区。

### 失效副本的判定

## Leader选举

一条消息只有被ISR中的所有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。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中有一个或者以上的follower，一条被commit的消息就不会丢失。

有一个很重要的问题是当leader宕机了，怎样在follower中选举出新的leader，因为follower可能落后很多或者直接crash了，所以必须确保选择“最新”的follower作为新的leader。一个基本的原则就是，如果leader不在了，新的leader必须拥有原来的leader commit的所有消息。这就需要做一个折中，如果leader在一个消息被commit前等待更多的follower确认，那么在它挂掉之后就有更多的follower可以成为新的leader，但这也会造成吞吐率的下降。

一种非常常用的选举leader的方式是“少数服从多数”，Kafka并不是采用这种方式。这种模式下，如果我们有2f+1个副本，那么在commit之前必须保证有f+1个replica复制完消息，同时为了保证能正确选举出新的leader，失败的副本数不能超过f个。这种方式有个很大的优势，系统的延迟取决于最快的几台机器，也就是说比如副本数为3，那么延迟就取决于最快的那个follower而不是最慢的那个。“少数服从多数”的方式也有一些劣势，为了保证leader选举的正常进行，它所能容忍的失败的follower数比较少，如果要容忍1个follower挂掉，那么至少要3个以上的副本，如果要容忍2个follower挂掉，必须要有5个以上的副本。也就是说，在生产环境下为了保证较高的容错率，必须要有大量的副本，而大量的副本又会在大数据量下导致性能的急剧下降。这种算法更多用在Zookeeper这种共享集群配置的系统中而很少在需要大量数据的系统中使用的原因。HDFS的HA功能也是基于“少数服从多数”的方式，但是其数据存储并不是采用这样的方式。

实际上，leader选举的算法非常多，比如Zookeeper的Zab、Raft以及Viewstamped Replication。而Kafka所使用的leader选举算法更像是微软的PacificA算法。

Kafka在Zookeeper中为每一个partition动态的维护了一个ISR，这个ISR里的所有replica都跟上了leader，只有ISR里的成员才能有被选为leader的可能（unclean.leader.election.enable=false）。在这种模式下，对于f+1个副本，一个Kafka topic能在保证不丢失已经commit消息的前提下容忍f个副本的失败，在大多数使用场景下，这种模式是十分有利的。事实上，为了容忍f个副本的失败，“少数服从多数”的方式和ISR在commit前需要等待的副本的数量是一样的，但是ISR需要的总的副本的个数几乎是“少数服从多数”的方式的一半。

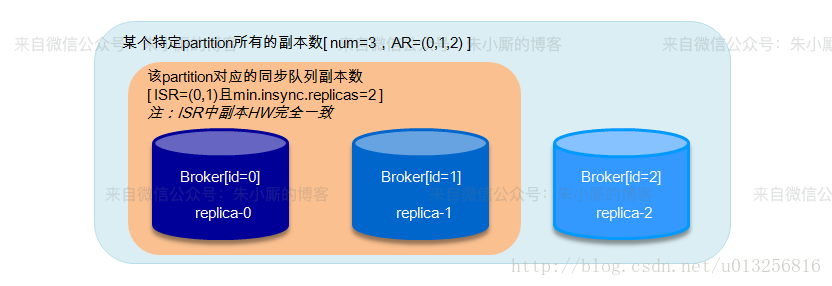
上文提到，在ISR中至少有一个follower时，Kafka可以确保已经commit的数据不丢失，但如果某一个partition的所有replica都挂了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

等待ISR中任意一个replica“活”过来，并且选它作为leader

选择第一个“活”过来的replica（并不一定是在ISR中）作为leader

这就需要在可用性和一致性当中作出一个简单的抉择。如果一定要等待ISR中的replica“活”过来，那不可用的时间就可能会相对较长。而且如果ISR中所有的replica都无法“活”过来了，或者数据丢失了，这个partition将永远不可用。选择第一个“活”过来的replica作为leader,而这个replica不是ISR中的replica,那即使它并不保障已经包含了所有已commit的消息，它也会成为leader而作为consumer的数据源。默认情况下，Kafka采用第二种策略，即unclean.leader.election.enable=true，也可以将此参数设置为false来启用第一种策略。

unclean.leader.election.enable这个参数对于leader的选举、系统的可用性以及数据的可靠性都有至关重要的影响。下面我们来分析下几种典型的场景。



如果上图所示，假设某个partition中的副本数为3，replica-0, replica-1, replica-2分别存放在broker0, broker1和broker2中。AR=(0,1,2)，ISR=(0,1)。

设置request.required.acks=-1, min.insync.replicas=2，unclean.leader.election.enable=false。这里将broker0中的副本也称之为broker0起初broker0为leader，broker1为follower。

当ISR中的replica-0出现crash的情况时，broker1选举为新的leader[ISR=(1)]，因为受min.insync.replicas=2影响，write不能服务，但是read能继续正常服务。此种情况恢复方案：

尝试恢复(重启)replica-0，如果能起来，系统正常；

如果replica-0不能恢复，需要将min.insync.replicas设置为1，恢复write功能。

当ISR中的replica-0出现crash，紧接着replica-1也出现了crash, 此时[ISR=(1),leader=-1],不能对外提供服务，此种情况恢复方案：

尝试恢复replica-0和replica-1，如果都能起来，则系统恢复正常；

如果replica-0起来，而replica-1不能起来，这时候仍然不能选出leader，因为当设置unclean.leader.election.enable=false时，leader只能从ISR中选举，当ISR中所有副本都失效之后，需要ISR中最后失效的那个副本能恢复之后才能选举leader, 即replica-0先失效，replica-1后失效，需要replica-1恢复后才能选举leader。保守的方案建议把unclean.leader.election.enable设置为true,但是这样会有丢失数据的情况发生，这样可以恢复read服务。同样需要将min.insync.replicas设置为1，恢复write功能；

replica-1恢复，replica-0不能恢复，这个情况上面遇到过，read服务可用，需要将min.insync.replicas设置为1，恢复write功能；

replica-0和replica-1都不能恢复，这种情况可以参考情形2.

当ISR中的replica-0, replica-1同时宕机,此时[ISR=(0,1)],不能对外提供服务，此种情况恢复方案：尝试恢复replica-0和replica-1，当其中任意一个副本恢复正常时，对外可以提供read服务。直到2个副本恢复正常，write功能才能恢复，或者将将min.insync.replicas设置为1。

# Partition 的副本机制

## Partition

### Partition

每个topic 都可以分为多个partition，并且多个partition 会均匀分布在集群的各个节点下。虽然这种方式能够有效的对数据进行分片，但是对于每个partition来说 ，都是单点的，当其中一个partition不可用的时候，那么这部分消息就没办法消费。所以kafka为了提高partition的可靠性而提供了副本的概念（Replica)，通过副本机制来实现冗余备份。

每个分区可以有多个副本，并且在副本集合中会存在一个leader的副本，所有的读写请求都是由leader副本来进行处理。剩余的其他副本都作为Follower副本，Follower副本会从leader副本同步消息日志。这个有点类似Zookeeper中leader和Follower的概念，但是具体实现方式还是有很大的差异。所以副本集会存在一主多从的关系。

一般情况下，同一个分区的多个副本会被均匀分配到集群中的不同broker上，当leader副本所在broker出现故障后，可以重新选举新的leader副本继续对外提供服务。通过这样的副本机制提高了kafka集群的可用性。

注意：kafka集群中的一个broker中最多只能有一个副本，leader副本所在的broker节点的分区叫leader节点，foolower副本所在的broker节点的分区叫follower节点

# Steam

# kafka安装和配置

## 安装

### 安装jdk

1、在/usr/local 创建java目录，将jdk解压。

2、在/etc/profile文件设置环境变量

|  |
| --- |
| export JAVA\_HOME=/usr/local/java/jdk1.8.0\_191  export JRE\_HOME=${JAVA\_HOME}/jre  export PATH=${PATH}:${JAVA\_HOME}/bin:${JRE\_HOME}/bin  export CLASSPATH=.:${JAVA\_HOME}/lib:${JRE\_HOME}/lib |

3、刷新配置文件

Source /etc/profile

4、测试

java或java -version

### 安装Zookeeper

1、上传解压zk

解压文件：tar -zxvf zookeeper-3.4.12

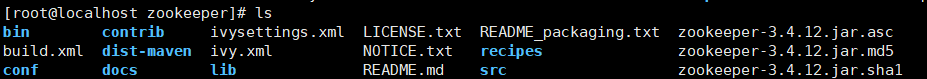
2、设置环境变量

在/etc/profile文件设置环境变量

|  |
| --- |
| export ZOOKEEPER\_HOME=/usr/local/zookeeper/apache-zookeeper-3.5.9-bin  export PATH=$PATH:$ZOOKEEPER\_HOME/bin |

source /etc/profile

3、文件目录介绍



bin:可执行文件

lib：依赖

conf：配置文件

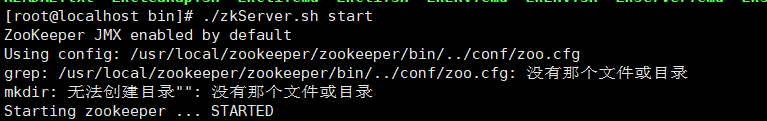
4、zookeeper运行

命令： zkServer.sh start|stop|status|restart

进入bin目录：cd bin



执行文件: ./ zkServer.sh



报错原因：没有配置文件，进入conf文件夹中，将zoo\_sample.cfg修改为zoo.cfg



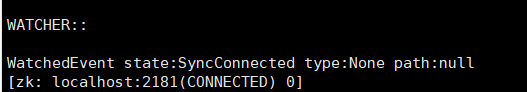
mv zoo\_sample.cfg zoo.cfg

运行zookeeper服务器：./zkServer.sh start

查看是否启动成功：jps



运行zookeeper 客户端：./zkCli.sh



默认端口：2181

### 关闭防火墙

firewall-cmd –state

systemctl stop firewalld.service

systemctl disable firewalld.service

### Kafka

1、安装

tar -zxvf kafka\_2.11-2.4.0.tgz

命令

1、启动Kafka

bin/kafka-server-start.sh config/server.properties &

2、停止Kafka

bin/kafka-server-stop.sh

3、创建Topic

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic jiangzh-topic

4、查看已经创建的Topic信息

bin/kafka-topics.sh --list --zookeeper localhost:2181

5、发送消息(producer)

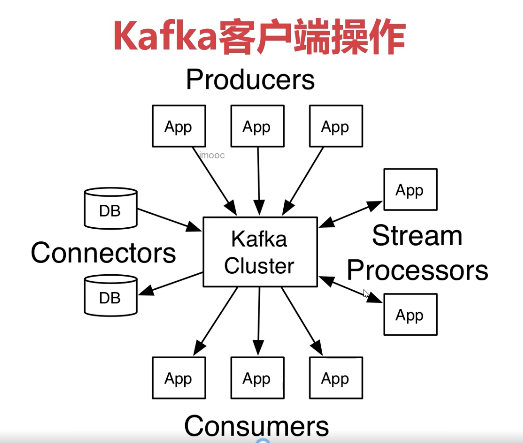
bin/kafka-console-producer.sh --broker-list 192.168.205.134:9092 --topic jiangzh-topic

6、接收消息(consumer)

bin/kafka-console-consumer.sh --bootstrap-server 192.168.205.134:9092 --topic jiangzh-topic --from-beginning

{"orderId":"002","price":"80"}

## 客户端操作



AdminClient APl:允许管理和检测Topic, broker以及其它Kafka对象

Producer API:发布消息到1个或多个topic

Consumer API:订阅一个或多个topic,并处理产生的消息

,Streams API:高效地将输入流转换到输出流

Connector APl:从一些源系统或应用程序中拉取数据到kafka

# 集群

## 集群

## 集群安装

### 修改配置

broker.id:同一个集群中的每个机器的id必须唯一，修改三台

broker.id=0

zookeeper连接配置，修改三台

zookeeper.connect=192.168.165.3:2181

继续修改Listeners配置，修改三台

listeners=PLAINTEXT://192.168.165.4:9092

2、启动

bin/kafka-server-start.sh config/server.properties &

3、关闭

bin/kafka-server-stop.sh

## Clinent



Properties properties = new Properties();

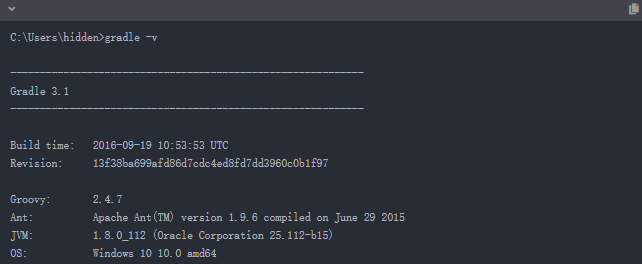
## 源码搭建

### 安装和配置JDK

确认JDK版本至少为1.7，最好是1.8及以上。使用java -version命令来查看当前JDK的版本，示例如下：

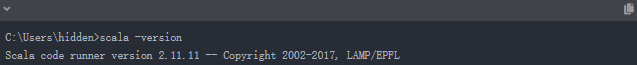
### 下载并安装配置Gradle

下载地址为：https://gradle.org/releases/，笔者使用的版本是3.1。一般只需要将下载的包解压，然后再将\$GRADLE\_HOME/bin的路径添加到环境变量Path中即可，其中\$GRADLE\_HOME指的是Gradle的根目录。可以使用gradle -v命令来验证Gradle是否已经配置完成，示例如下：



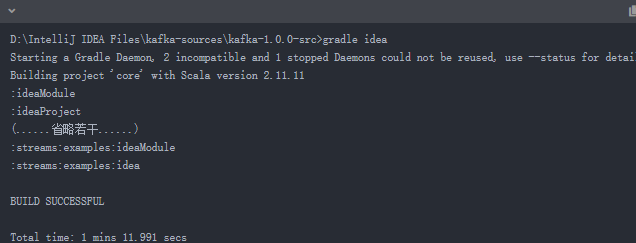
### 下载并安装配置Scala

下载地址为：[http://www.scala-lang.org/download/all.html，目前最新的版本是2.12.4，不过笔者这里使用的版本是2.11.11。如Gradle一样，只需要解压并将\$SCALA\_HOME/bin的路径添加到环境变量Path即可，其中\$SCALA\_HOME指的是Scala的根目录。可以使用scala](http://www.scala-lang.org/download/all.html%EF%BC%8C%E7%9B%AE%E5%89%8D%E6%9C%80%E6%96%B0%E7%9A%84%E7%89%88%E6%9C%AC%E6%98%AF2.12.4%EF%BC%8C%E4%B8%8D%E8%BF%87%E7%AC%94%E8%80%85%E8%BF%99%E9%87%8C%E4%BD%BF%E7%94%A8%E7%9A%84%E7%89%88%E6%9C%AC%E6%98%AF2.11.11%E3%80%82%E5%A6%82Gradle%E4%B8%80%E6%A0%B7%EF%BC%8C%E5%8F%AA%E9%9C%80%E8%A6%81%E8%A7%A3%E5%8E%8B%E5%B9%B6%E5%B0%86/$SCALA_HOME/bin%E7%9A%84%E8%B7%AF%E5%BE%84%E6%B7%BB%E5%8A%A0%E5%88%B0%E7%8E%AF%E5%A2%83%E5%8F%98%E9%87%8FPath%E5%8D%B3%E5%8F%AF%EF%BC%8C%E5%85%B6%E4%B8%AD/$SCALA_HOME%E6%8C%87%E7%9A%84%E6%98%AFScala%E7%9A%84%E6%A0%B9%E7%9B%AE%E5%BD%95%E3%80%82%E5%8F%AF%E4%BB%A5%E4%BD%BF%E7%94%A8scala) -version命令来验证scala是否已经配置完成，示例如下：

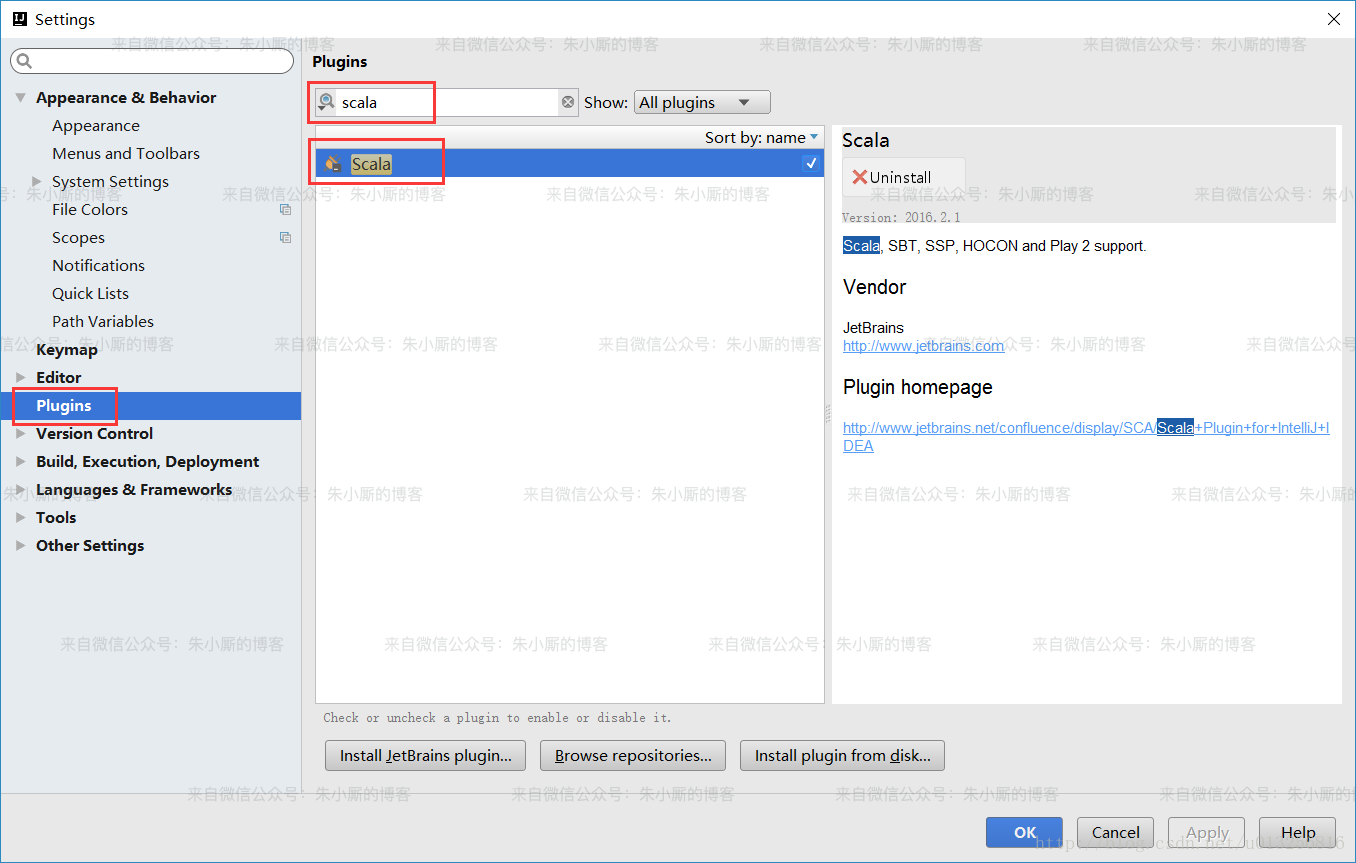


### 构建Kafka源码环境

Kafka下载地址为：http://kafka.apache.org/downloads，目前最新的版本是1.0.0。将下载的压缩包解压，并在Kafka的根目录执行gradle idea命令进行构建，如果你使用的是Eclipse，则只需采用gradle eclipse命令构建即可。构建细节如下所示：



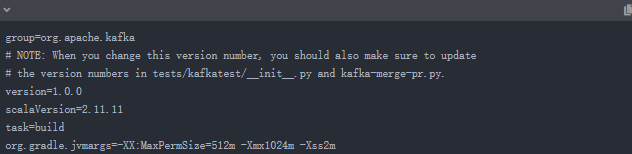
之后将Kafka导入到IDEA中即可。不过这样还没有结束，对于IDEA而言，还需要安装Scala插件，在Setting->Plugin中搜索scala并安装，可以参考下图，笔者这里是已经安装好的状态：



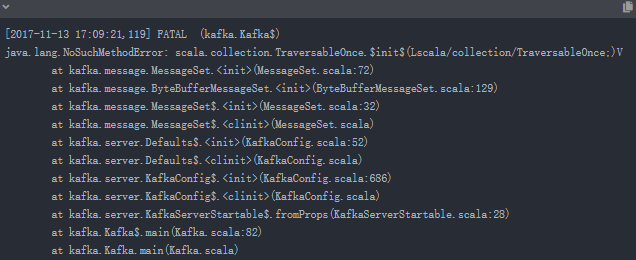
### 配置Kafka源码环境

前面几个步骤执行完成后就可以很舒适的阅读Kafka的源码，但是如果需要启动Kafka的服务还需要一些额外的步骤。

首先确保gradle.properties配置文件中的scalaVersion与安装的一致。gradle.properties配置文件的细节如下：

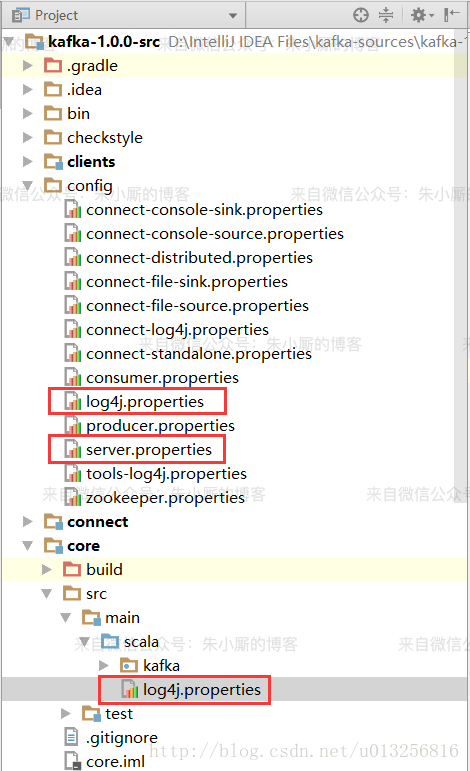


如果更改了scalaVersion，需要重新执行gradle idea命令来重新构建。虽然很多时候在操作系统中安装其他版本的Scala也并没有什么问题，比如安装2.12.4版本。但是有些情况下运行Kafka时会出现一些异常，而这些异常却又是由于Scala版本不一致而引起的，比如会出现下面示例中的报错：

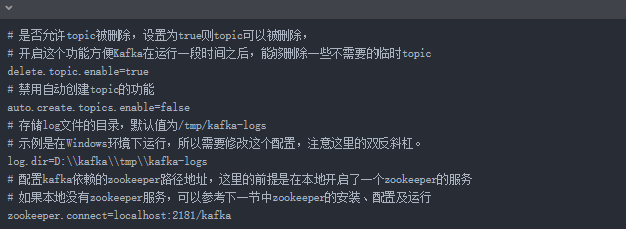


所以为了省去一些不必要的麻烦，还是建议读者在安装Scala版本之前先查看下Kafka源码中gradle.properties文件中配置的scalaVersion。

再确保了scalaVersion之后，需要将config目录下的log4j.properties文件拷贝到core/src/main/scala目录下，这样可以让Kafka在运行时能够输出日志信息，可以参考下图：



之后还需要配置server.properties文件，一般只需要修改以下一些配置项：



之后配置Kafka的启动参数，详细参考下图：

# Kafka的监控

# Kafka生产问题

### 发送超过broker限定大小

Kafka发送超过broker限定大小的消息时Client和Broker端各自会有什么异常？

前几天遇到一个bug，查看发送日志发现java.io.IOException: Broken pipe的错误，通过深入了解发现当kafka producer发送的消息体大于Broker配置的默认值时就会报这个异常。如果仅发送一次是不会报这个异常的，要连续发送才会报这个异常。

Kafka Broker Configs中有一个参数：message.max.bytes——用来指定消息的大小。



当Producer向Broker发送一个比Kafka Broker配置的阈值还要大的一个消息时,Producer端和Broker端会有什么异常情况。

**Prodcude**: 注意输出中的：java.io.IOException: 你的主机中的软件中止了一个已建立的连接。（ps:如果没有中文，这里会出现“java.io.IOException: Broken pipe”的报错。）

**Broker:** kafka.common.MessageSizeTooLargeException: Message size is 1048602 bytes which exceeds the maximum configured message size of 1000012.这句。