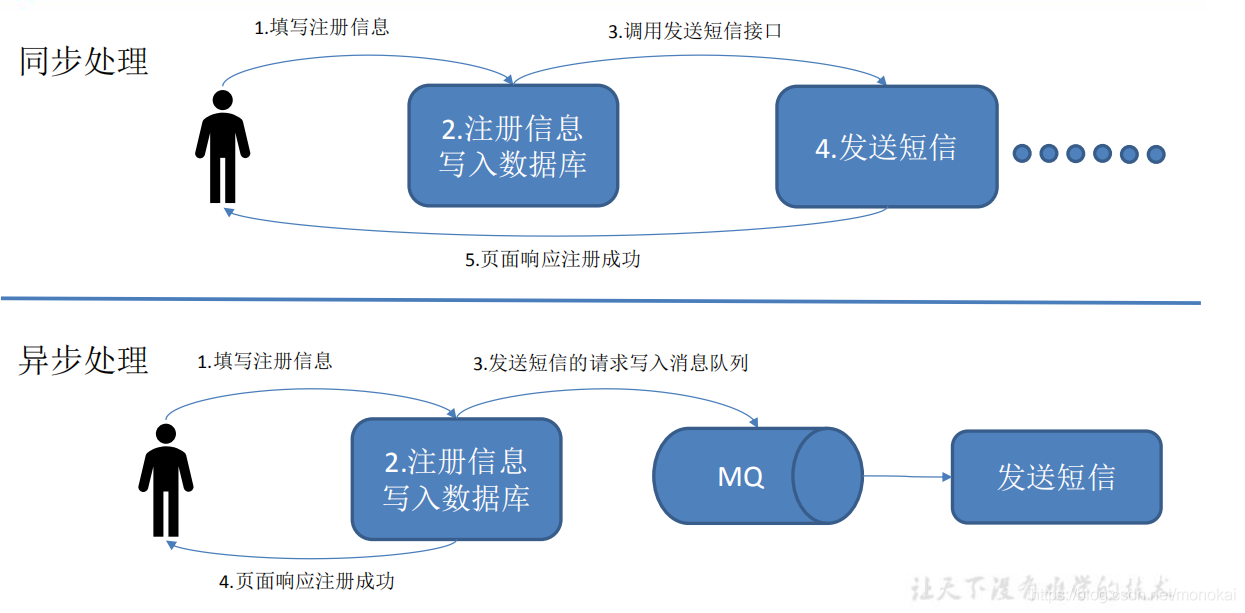
# **第 1 章 Kafka 概述**

## **1.1 定义**

Kafka 是一个分布式的基于发布/订阅模式的消息队列（Message Queue），主要应用于大数据实时处理领域。(官方定义Kafka是一个分布式 流处理平台，包含消息系统、文件存储系统、流处理（无穷数据集）、批处理。四大API，Producer API、Consumer API、Stream API、Connect API)

## **1.2 消息队列**

### **1.2.1 传统消息队列的应用场景**

  
****使用消息队列的好处****

1）解耦  
允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

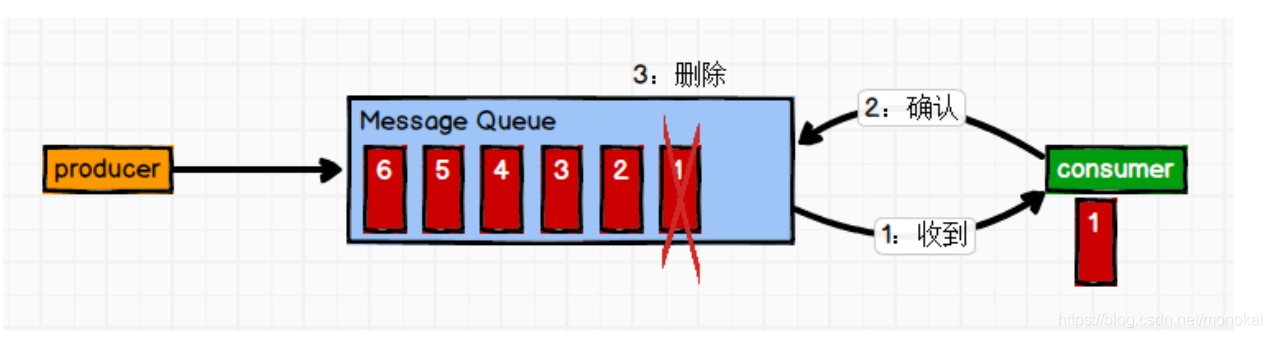
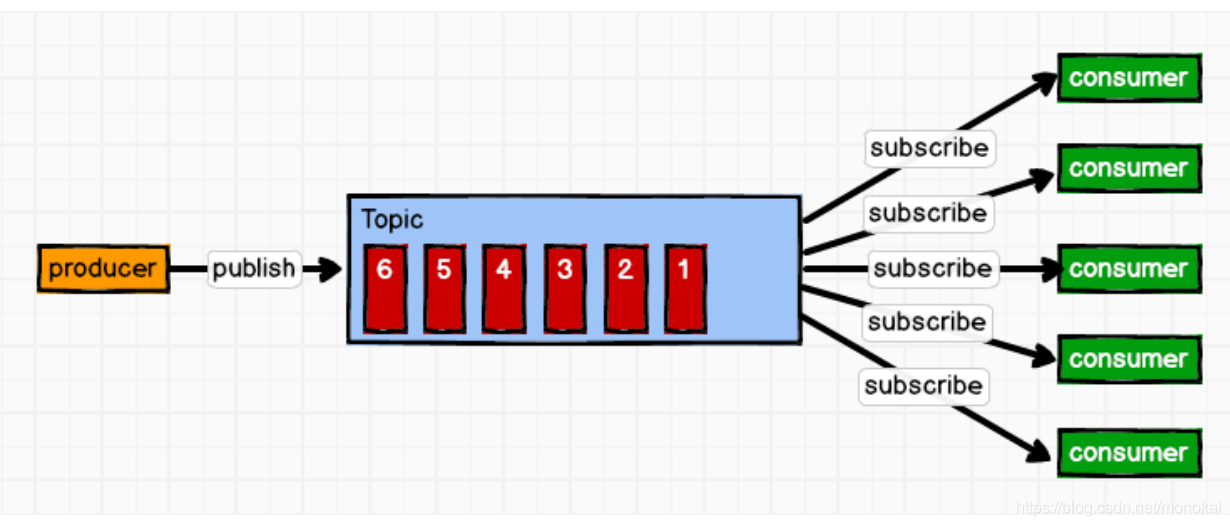
2）可恢复性  
系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，当有一个消费者的进程挂掉，消息（在队列中的）仍然可以在系统恢复后处理。

3）缓冲  
有助于控制和优化数据流经过系统的速度，解决生产消息和消费消息的处理速度不一致  
的情况。

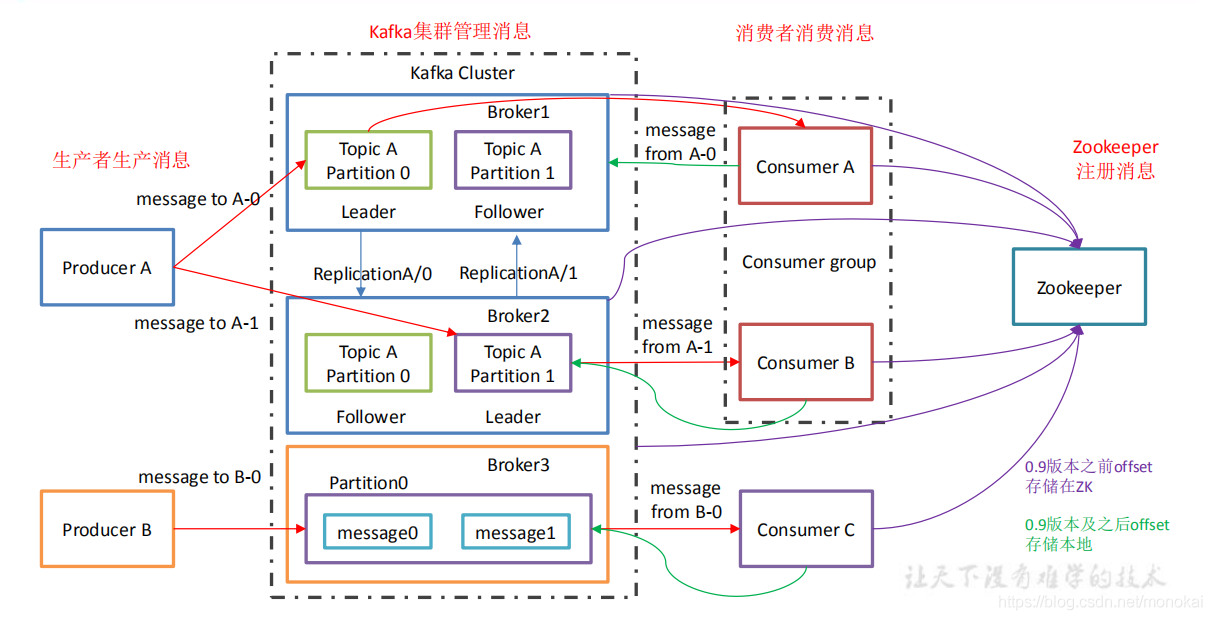
4）灵活性 & 峰值处理能力（削峰能力 不会炸掉）  
在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。  
如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列  
能够使关键组件顶住突发的访问压力（当突发流量进来后，会在消息队列中存储），而不会因为突发的超负荷的请求而完全崩溃。

5）异步通信（效率高，并行能力强）  
很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户  
把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要  
的时候再去处理它们。

### **1.2.2 消息队列的两种模式**

（1）点对点模式（一对一，消费者主动拉取数据，消息收到后消息清除）  
消息生产者生产消息发送到Queue中，然后消息消费者从Queue中取出并且消费消息。  
消息被消费以后，queue 中不再有存储，所以消息消费者不可能消费到已经被消费的消息。  
Queue 支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。  
（2）发布/订阅模式（一对多，消费者消费数据之后不会清除消息）  
消息生产者（发布）将消息发布到 topic 中，同时有多个消息消费者（订阅）消费该消  
息。和点对点方式不同，发布到 topic 的消息会被所有订阅者消费。

## **1.3 Kafka 基础架构**



1）Producer ：消息生产者，就是向 kafka broker 发消息的客户端；

2）Consumer ：消息消费者，向 kafka broker 取消息的客户端；

3）Consumer Group （CG）：消费者组，由多个 consumer 组成。消费者组内每个消费者负 责消费不同分区的数据，一个分区只能由一个组内消费者消费；消费者组之间互不影响。所  
有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

4）Broker ：一台 kafka 服务器就是一个 broker。一个集群由多个 broker 组成。一个 broker可以容纳多个 topic。 -

5）Topic ：可以理解为一个队列，生产者和消费者面向的都是一个 topic；

6）Partition：为了实现扩展性，一个非常大的 topic 可以分布到多个 broker（即服务器）上，一个 topic 可以分为多个 partition，每个 partition 是一个有序的队列；

7）Replica：副本，为保证集群中的某个节点发生故障时，该节点上的 partition 数据不丢失，且 kafka 仍然能够继续工作，kafka 提供了副本机制，一个 topic 的每个分区都有若干个副本，  
一个leader和若干个 follower。副本数指定是leader+follower的总和。

8）leader：每个分区多个副本的“主”，生产者发送数据的对象，以及消费者消费数据的对象都是 leader。

9）follower：每个分区多个副本中的“从”，实时从 leader 中同步数据，保持和 leader 数据的同步。leader 发生故障时，某个 follower 会成为新的 follower。

# **第 2 章 Kafka 快速入门**

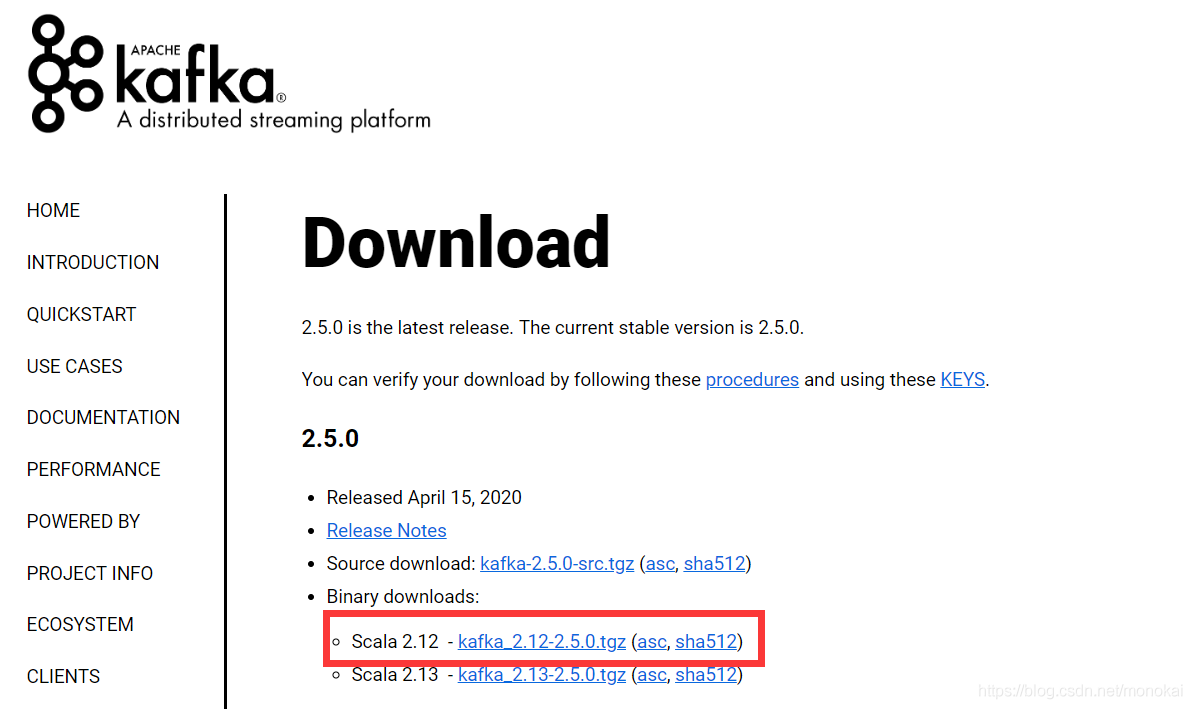
## **2.1 安装部署**

这里我们准备在三台主机上搭建kafka集群，

10.10.10.102，10.10.10.103，10.10.10.104

首先先安装zookeeper。

## **2.1.2 jar 包下载**

<http://kafka.apache.org/downloads.html>  


## **2.1.3 集群部署**

1）解压安装包

tar -zxvf kafka\_2.12-2.5.0.0.tgz -C /opt/module/

2）修改解压后的文件名称

$ mv kafka\_2.12-2.5.0.0/ kafka

3）在/opt/module/kafka 目录下创建 logs 文件夹

mkdir logs

4）修改配置文件

cd config/

vim server.properties

输入以下内容：

#broker 的全局唯一编号，不能重复

broker.id=0

#删除 topic 功能使能

delete.topic.enable=true

#处理网络请求的线程数量

num.network.threads=3

#用来处理磁盘 IO 的现成数量

num.io.threads=8

#发送套接字的缓冲区大小

socket.send.buffer.bytes=102400

#接收套接字的缓冲区大小

socket.receive.buffer.bytes=102400

#请求套接字的缓冲区大小

socket.request.max.bytes=104857600

#kafka 运行日志存放的路径

log.dirs=/opt/module/kafka/logs

#topic 在当前 broker 上的分区个数

num.partitions=1

#用来恢复和清理 data 下数据的线程数量

num.recovery.threads.per.data.dir=1

#segment 文件保留的最长时间，超时将被删除

log.retention.hours=168

#配置连接 Zookeeper 集群地址

zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181

5）配置环境变量

sudo vim /etc/profile

*#KAFKA\_HOME*export KAFKA\_HOME=/opt/module/kafkaexport PATH=$PATH:$KAFKA\_HOME/bin

source /etc/profil6）分发安装包

xsync kafka/

注意：分发之后记得配置其他机器的环境变量

7）分别在 剩下的机器 上修改配置文件/opt/module/kafka/config/server.properties  
中的 broker.id=1、broker.id=2

注：broker.id 不得重复

8）启动集群  
依次在 hadoop102、hadoop103、hadoop104 节点上启动 kafka

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

9）关闭集群

bin/kafka-server-stop.sh stop

10）kafka 群起脚本

for i in 10.10.10.102 10.10.10.103 10.10.10.104doecho "========== $i ==========" ssh $i '/opt/module/kafka/bin/kafka-server-start.sh -daemon

/opt/module/kafka/config/server.properties'done

## **2.2 Kafka 命令行操作**

1）查看当前服务器中的所有 topic

bin/kafka-topics.sh --zookeeper zk地址:2181 --list

2）创建 topic

bin/kafka-topics.sh --zookeeper zk地址:2181 --create --replication-factor 3 --partitions 1 --

topic first

选项说明：

* topic 定义 topic 名
* replication-factor 定义副本数
* partitions 定义分区数

3）删除 topic

bin/kafka-topics.sh --zookeeper zk地址:2181 --delete --topic first

需要 server.properties中设置 delete.topic.enable=true ****否则只是标记删除****。

4）发送消息

bin/kafka-console-producer.sh --broker-list 10.10.10.102:9092 --topic first

5）消费消息

bin/kafka-console-consumer.sh \

--zookeeper zk地址:2181 --topic first

bin/kafka-console-consumer.sh \

--bootstrap-server 10.10.10.102:9092 --topic first

bin/kafka-console-consumer.sh \

--bootstrap-server 10.10.10.102:9092 --from-beginning --topic first

from-beginning：会把主题中以往所有的数据都读取出来。

6）查看某个 Topic 的详情

bin/kafka-topics.sh --zookeeper

10.10.10.102:2181 --describe --topic first

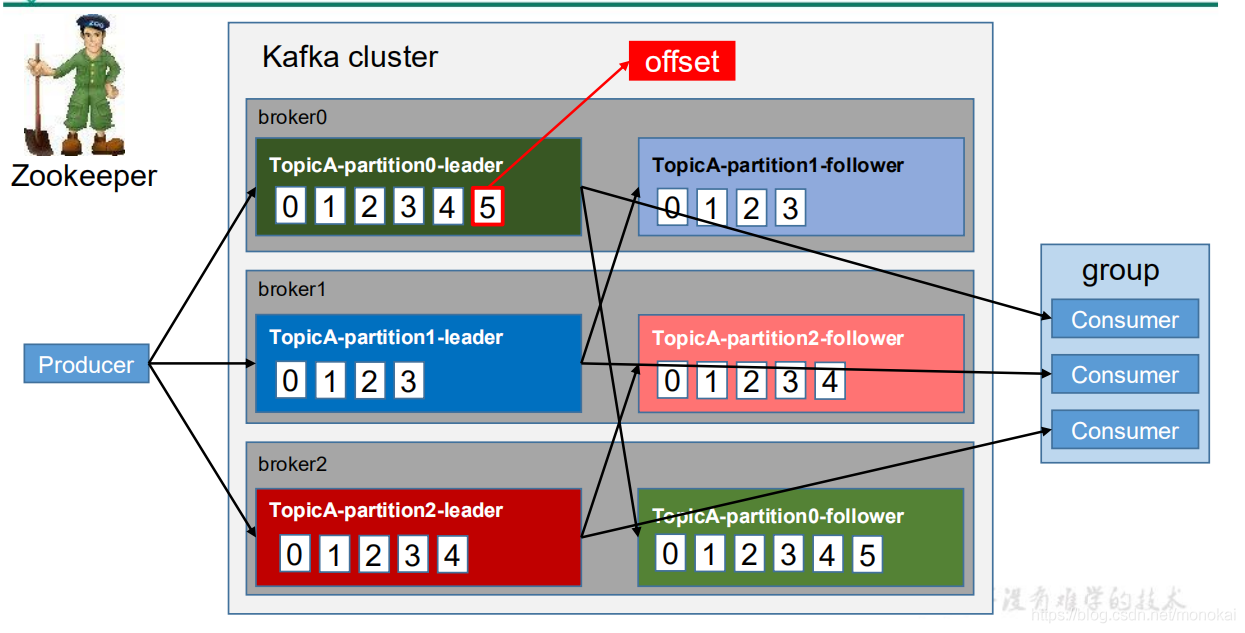
7）修改分区数

bin/kafka-topics.sh --zookeeper

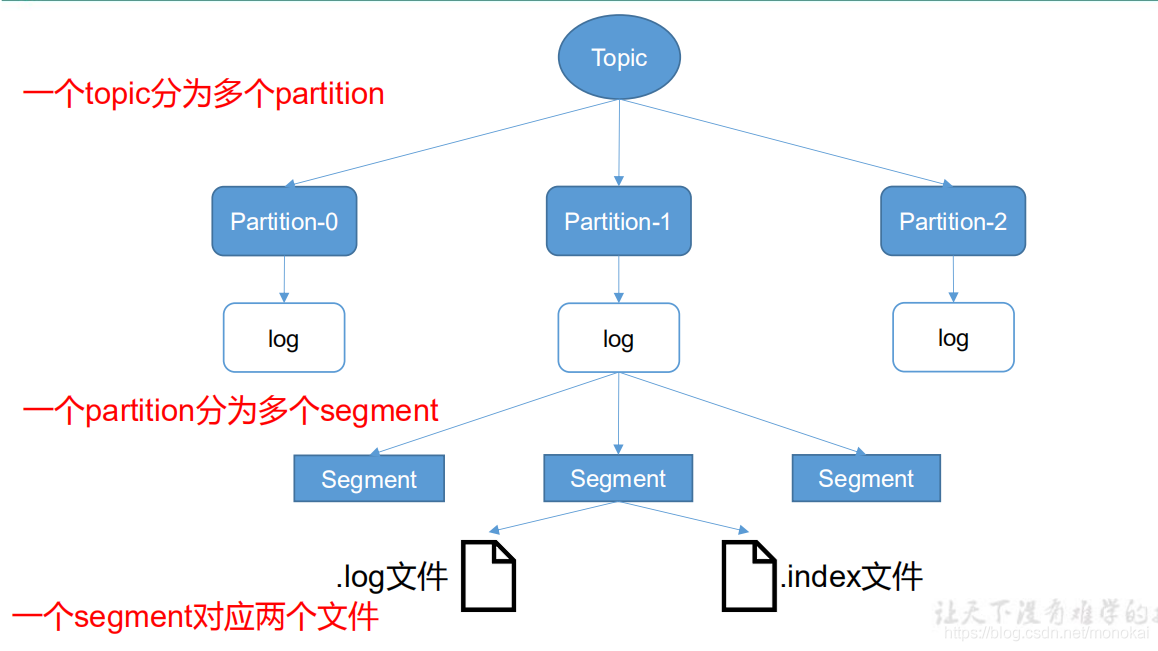
10.10.10.102:2181 --alter --topic first --partitions 6

# **第 3 章 Kafka 架构深入**

## **3.1 Kafka 工作流程及文件存储机制**



Kafka 中消息是以 topic 进行分类的，生产者生产消息，消费者消费消息，都是面向 topic  
的。topic 是逻辑上的概念，而 partition 是物理上的概念，****每个 partition 对应于一个 log 文件，该 log 文件中存储的就是 producer 生产的数据****。Producer 生产的数据会被不断追加到该log 文件末端，且每条数据都有自己的 offset。消费者组中的每个消费者，都会实时记录自己消费到了哪个 offset，以便出错恢复时，从上次的位置继续消费

Kafka文件存储机制  
  
由于生产者生产的消息会不断追加到 log 文件末尾，为防止 log 文件过大导致数据定位  
效率低下，Kafka 采取了分片和索引机制，****将每个 partition 分为多个 segment****。每个 segment对应两个文件——“.index”文件和“.log”文件。这些文件位于一个文件夹下，该文件夹的命名规则为：topic 名称+分区序号。  
例如，first 这个 topic 有三个分区，则其对应的文件夹为 first-0,first-1,first-2。

00000000000000000000.index

00000000000000000000.log

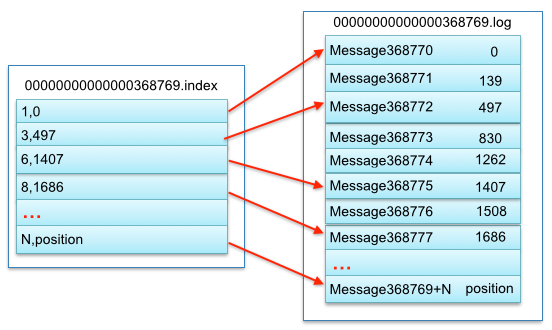
00000000000000170410.index

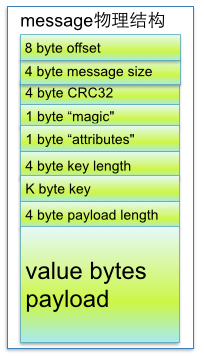
00000000000000170410.log

00000000000000239430.index

00000000000000239430.log

index 和 log 文件以当前 segment 的第一条消息的 offset 命名。下图为 index 文件和 log文件的结构示意图。





（取自美团官博）

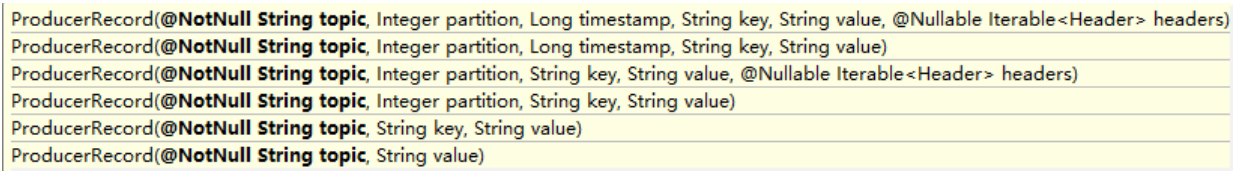
“.index”文件存储大量的索引信息，index的第一列是相对于文件名中消息的offset的相对偏移量，为了不使索引文件巨大采用了稀疏存储，“.log”文件存储大量的数据，索引文件中的元数据指向对应数据文件中 message 的物理偏移地址。

## **3.2 Kafka 生产者**

### **3.2.1 分区策略**

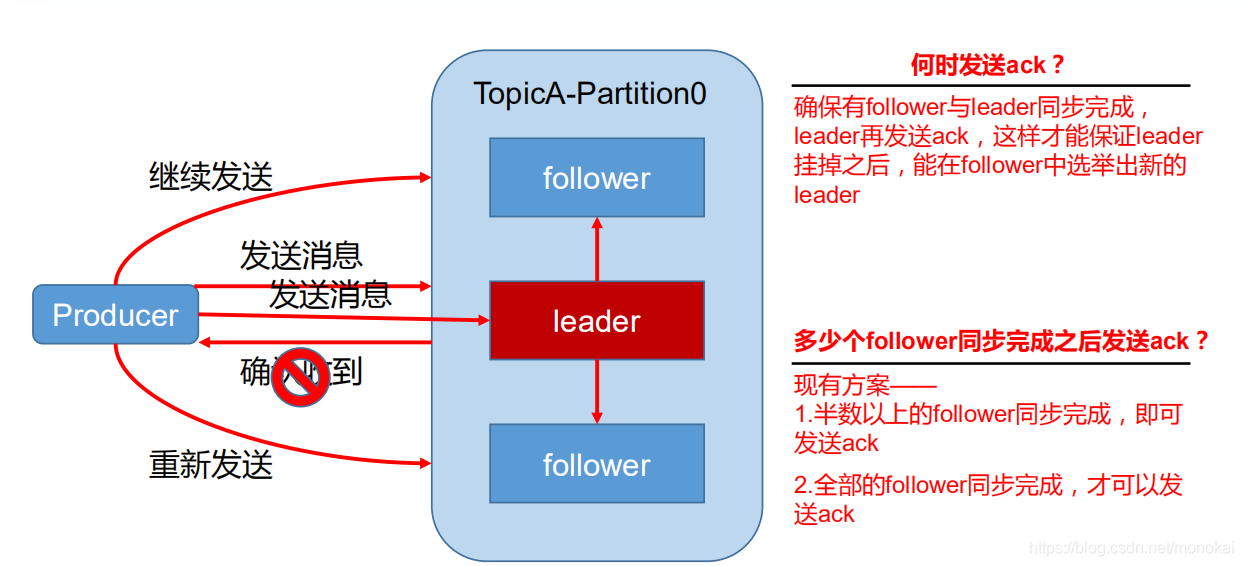
****1）分区的原因****

1. 方便在集群中扩展，每个 Partition 可以通过调整以适应它所在的机器，而一个 topic  
   又可以有多个 Partition 组成，因此整个集群就可以适应任意大小的数据了；
2. 可以提高并发，因为可以以 Partition 为单位读写了。

****2）分区的原则****  
我们需要将 producer 发送的数据封装成一个 ProducerRecord对象。  


1. 指明 partition 的情况下，直接将指明的值直接作为 partiton 值；
2. 没有指明 partition 值但有 key 的情况下，将 key 的 hash 值与 topic 的 partition数进行取余得到 partition 值；
3. 既没有 partition 值又没有 key 值的情况下，第一次调用时随机生成一个整数（后  
   面每次调用在这个整数上自增（轮询的方法）），将这个值与 topic 可用的 partition 总数取余得到 partition值，也就是常说的 round-robin 算法。

### **3.2.2 数据可靠性保证**

为保证 producer 发送的数据，能可靠的发送到指定的 topic，topic 的每个 partition 收到 producer 发送的数据后，都需要向 producer 发送 ack（acknowledgement 确认收到），如果 producer 收到 ack，就会进行下一轮的发送，否则重新发送数据。  


#### **1）副本数据同步策略**

副本同步有两种策略：

zk的半数策略，与假全数策略（自己的形容词）。

半数策略，当集群中有过半的服务器正常运行，则整个集群可以正常运行。相应的，当有n台机器故障，需要有2n+1台机器，才能保证集群的正常运行。

假全数策略，理论上是当有n台机器发生故障，只有要有n+1台机器就能保证正常运行。kafka采用了第二中。（但是实际上kafka维护了一个ISR集合，这个集合的判断条件在0.9.0之后变成，follow同步leader的时间是否在规定时间内，如果在的话就让这个follow 加入到ISR集合中。如果leader挂掉，新选举的leader从ISR中产生）

Kafka 选择了第二种方案，原因如下：

1. 同样为了容忍 n 台节点的故障，第一种方案需要 2n+1 个副本，而第二种方案只需要 n+1个副本，而 Kafka 的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。
2. 虽然第二种方案的网络延迟会比较高，但网络延迟对 Kafka 的影响较小。

#### **2）ISR**

采用第二种方案之后，设想以下情景：leader 收到数据，所有 follower 都开始同步数据，  
但有一个 follower，因为某种故障，迟迟不能与 leader 进行同步，那 leader 就要一直等下去，直到它完成同步，才能发送 ack。这个问题怎么解决呢？

Leader 维护了一个动态的 in-sync replica set (ISR)，意为和 leader 保持同步的 follower 集 合。当 ISR 中的 follower 完成数据的同步之后，leader 就会给 follower 发送 ack。如果 follower 长时间 未 向 leader 同 步 数 据 ， 则 该 follower 将 被 踢 出 ISR ， 该 时 间 阈 值 由replica.lag.time.max.ms 参数设定。Leader 发生故障之后，就会从 ISR 中选举新的 leader。

#### **3）ack 应答机制**

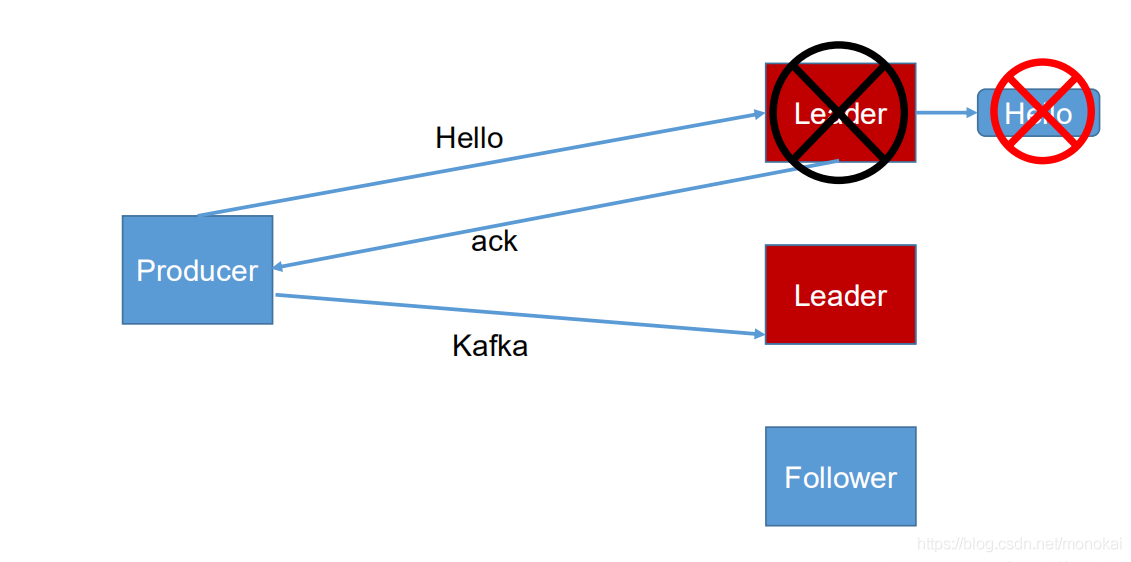
对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，  
所以没必要等 ISR 中的 follower 全部接收成功。所以 Kafka 为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡，选择以下的配置。

##### **acks 参数配置：**

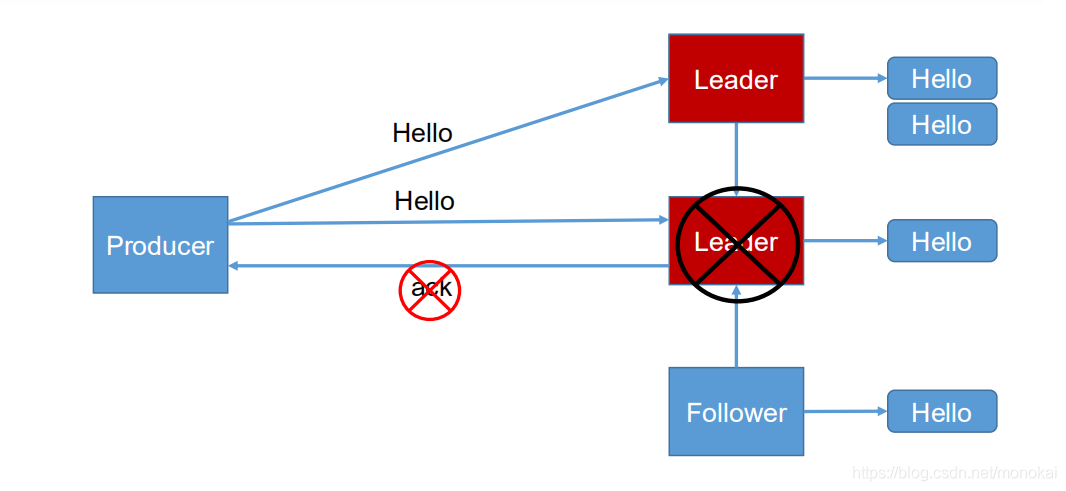
acks：

（1）“0”：producer 不等待 broker 的 ack，这一操作提供了一个最低的延迟，broker 一接收到还没有写入磁盘就已经返回，当 broker 故障时有可能丢失数据；

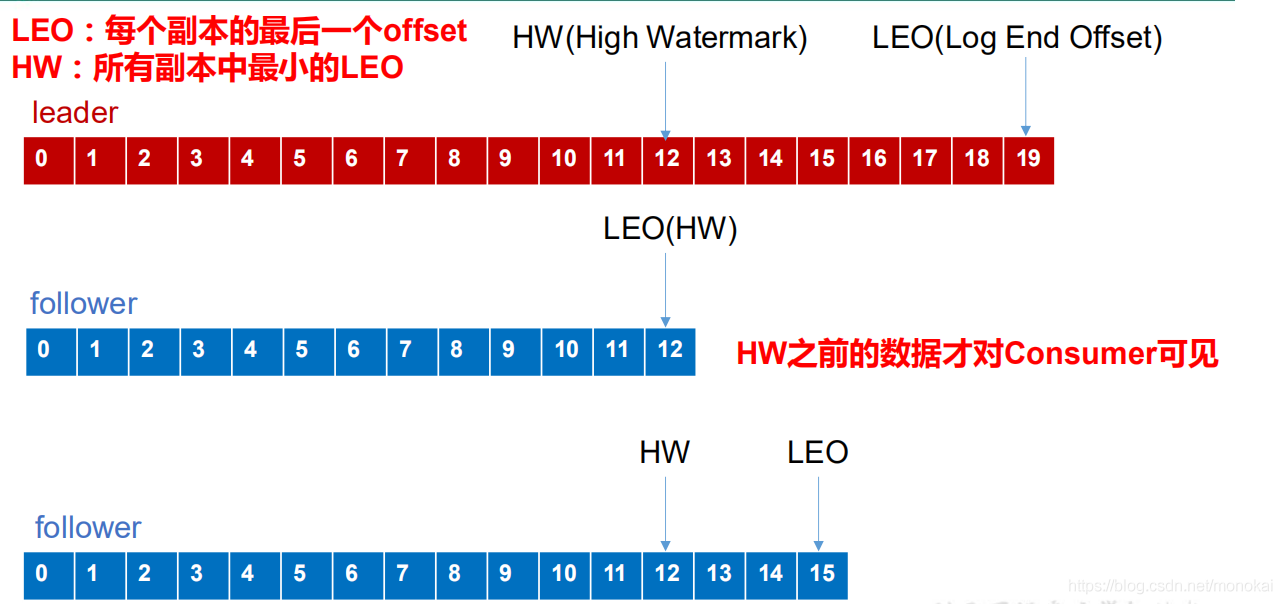
（2）‘’1‘’：producer 等待 broker 的 ack，partition 的 leader 落盘成功后返回 ack，如果在 follower同步成功之前 leader 故障，那么将会丢失数据；

acks = 1 数据丢失案例  


（3）“-1”（all）：producer 等待 broker 的 ack，partition 的 leader 和 follower 全部落盘成功后才返回 ack。但是如果在 follower 同步完成后，broker 发送 ack 之前，leader 发生故障，那么会造成数据重复。

acks = -1 数据重复案例  


#### **4）故障处理细节**

Log中HW和LEO相关保证数据的一致性

* LEO：指的是每个副本最大的 offset；
* HW：指的是消费者能见到的最大的 offset，ISR 队列中最小的 LEO。

##### **（1）follower 故障**

follower 发生故障后会被临时踢出 ISR，待该 follower 恢复后，follower 会读取本地磁盘。记录的上次的 HW，并将 log 文件高于 HW 的部分截取掉，从 HW 开始向 leader 进行同步。等该 follower 的 LEO 大于等于该Partition 的 HW，即 follower 追上 leader 之后，就可以重新加入 ISR 了。

##### **（2）leader 故障**

leader 发生故障之后，会从 ISR 中选出一个新的 leader，之后，为保证多个副本之间的数据一致性，其余的 follower 会先将各自的 log 文件高于 HW 的部分截掉，然后从新的 leader同步数据

注意：这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

### **3.2.3 Exactly Once 语义**

将服务器的 ACK 级别设置为-1，可以保证 Producer 到 Server 之间不会丢失数据，即 At Least Once 语义。相对的，将服务器 ACK 级别设置为 0，可以保证生产者每条消息只会被发送一次，即 At Most Once 语义。

At Least Once 可以保证数据不丢失，但是不能保证数据不重复；相对的，At Least Once  
可以保证数据不重复，但是不能保证数据不丢失。但是，对于一些非常重要的信息，比如说 交易数据，下游数据消费者要求数据既不重复也不丢失，即 Exactly Once 语义。在 0.11 版本以前的 Kafka，对此是无能为力的，只能保证数据不丢失，再在下游消费者对数据做全局去重。对于多个下游应用的情况，每个都需要单独做全局去重，这就对性能造成了很大影响。0.11 版本的 Kafka，引入了一项重大特性：幂等性。所谓的幂等性就是指 Producer 不论向 Server 发送多少次重复数据，Server 端都只会持久化一条。幂等性结合 At Least Once 语义，就构成了 Kafka 的 Exactly Once 语义。即：

* At Least Once + 幂等性 = Exactly Once

要启用幂等性，只需要将 Producer 的参数中 enable.idompotence 设置为 true 即可。Kafka的幂等性实现其实就是将原来下游需要做的去重放在了数据上游。开启幂等性的 Producer 在初始化的时候会被分配一个 PID，发往同一 Partition 的消息会附带 Sequence Number（发送消息的标识）。而Broker 端会对<PID, Partition, SeqNumber>做缓存，当具有相同主键的消息提交时，Broker 只会持久化一条。但是 PID 重启就会变化，同时不同的 Partition 也具有不同主键，所以幂等性无法保证跨分区跨会话的 Exactly Once。

## **3.3 Kafka 消费者**

### **3.3.1 消费方式**

* consumer 采用 pull（拉）模式从 broker 中读取数据。
* push（推）模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。

它的目标是尽可能以最快速度传递消息，但是这样很容易造成 consumer 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 模式则可以根据 consumer 的消费能力以适当的速率消费消息。pull 模式不足之处是，如果 kafka 没有数据，消费者可能会陷入循环中，一直返回空数 据。针对这一点，Kafka 的消费者在消费数据时会传入一个时长参数 timeout，如果当前没有数据可供消费，consumer 会等待一段时间之后再返回，这段时长即为 timeout。（虽然没有解决掉空轮询的问题，但是缓解了）

### **3.3.2 分区分配策略**

一个 consumer group 中有多个 consumer，一个 topic 有多个 partition，所以必然会涉及到 partition 的分配问题，即确定那个 partition 由哪个 consumer 来消费。Kafka 有两种分配策略，一是 RoundRobin，一是 Range。

#### **RoundRobin**

#### 分区分配策略之RoundRobin（轮询分配） IMG_269

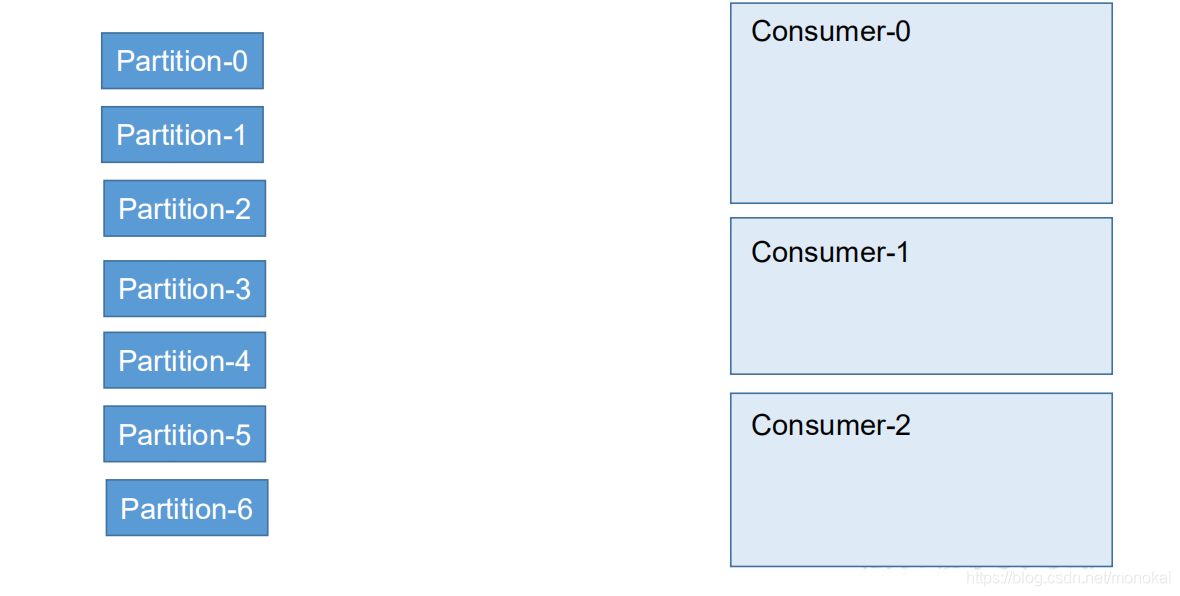
Consumer0: 0、3、6

Consumer1: 1、4

Consumer2: 2、5

依次分配给同一个消费者组中的每一个消费者。不确定是哪个消费者会先开始。

#### **2）Range**

分区分配策略之Range  


range策略，右侧为同一主题下的分区，右侧为同一分组下的三个消费者都订阅了topic A。

Consumer0: 0,1,2

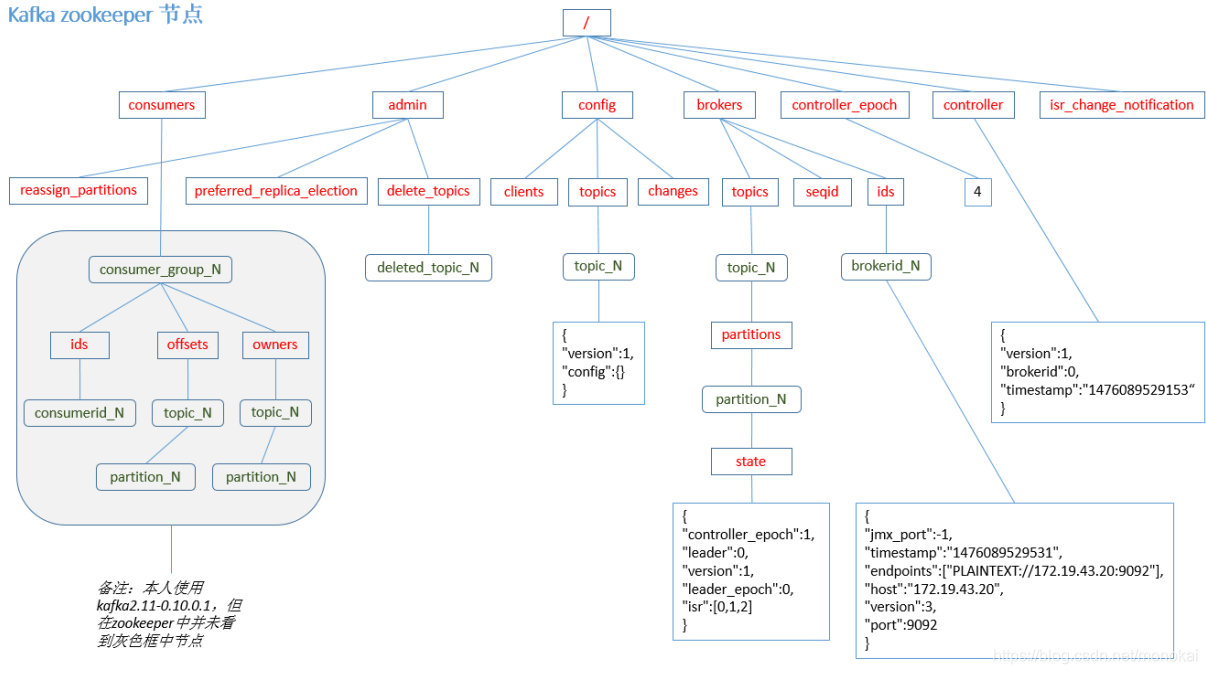
Consumer1: 3,4

Consumer2: 5,6

计算分区总数,计算同一组下的consumer总数 。Partition/Consumer每个分区有2余1

会随机给一个消费者订阅三个分区。不一定是上面我写的Consumer0消费三个分区的。

### **3.3.3 offset 的维护**

由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故  
障前的位置的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢复后继续消费。  
Kafka 0.9 版本之前，consumer 默认将 offset 保存在 Zookeeper 中，从 0.9 版本开始， consumer 默认将 offset 保存在 Kafka 一个内置的 topic 中，该 topic 为\_\_consumer\_offsets。

****1）修改配置文件 consumer.properties****

exclude.internal.topics=false

* 1

****2）读取 offset****  
0.11.0.0 之前版本:

bin/kafka-console-consumer.sh --topic \_\_consumer\_offsets --

zookeeper hadoop102:2181 --formatter "kafka.coordinator.GroupMetadataManager\$OffsetsMessageFormatter"

--consumer.config config/consumer.properties --from-beginning

0.11.0.0 之后版本(含):

bin/kafka-console-consumer.sh --topic \_\_consumer\_offsets --

zookeeper hadoop102:2181 --formatter "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageForm

atter" --consumer.config config/consumer.properties --frombeginning

### **3.3.4 消费者组案例**

1）需求：测试同一个消费者组中的消费者，同一时刻只能有一个消费者消费。

2）案例实操

1. 10.10.10.102、10.10.10.103上修改/opt/module/kafka/config/consumer.properties 配置文件中的 group.id 属性为任意组名。

vim consumer.properties

group.id=atguigu

1. 在 10.10.10.102、10.10.10.103上分别启动消费者

bin/kafka-console-consumer.sh \

--zookeeper zk地址:2181 --topic first --consumer.config

config/consumer.properties

bin/kafka-console-consumer.sh --

bootstrap-server 10.10.10.102:9092 --topic first --consumer.config

config/consumer.properties

1. 在 10.10.10.104上启动生产者

bin/kafka-console-producer.sh \

--broker-list 10.10.10.102:9092 --topic first

>hello world

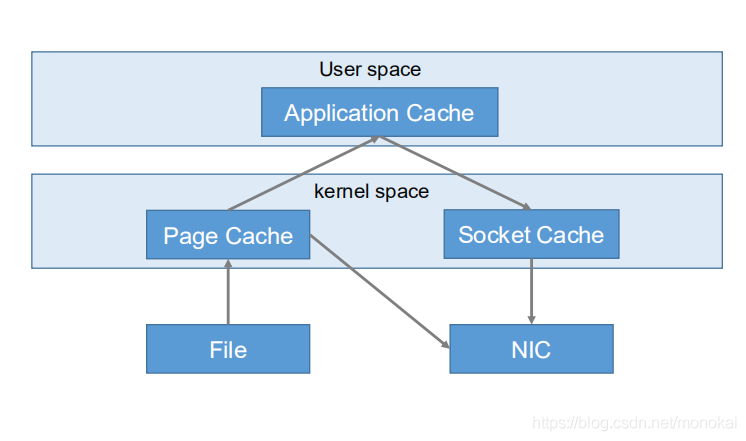
1. 查看 10.10.10.102和 10.10.10.103的接收者。同一时刻只有一个消费者接收到消息。

## **3.4 Kafka 高效读写数据**

### **1）顺序写磁盘**

Kafka 的 producer 生产数据，要写入到 log 文件中，写的过程是一直追加到文件末端，  
为顺序写。官网有数据表明，同样的磁盘，顺序写能到 600M/s，而随机写只有 100K/s。这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

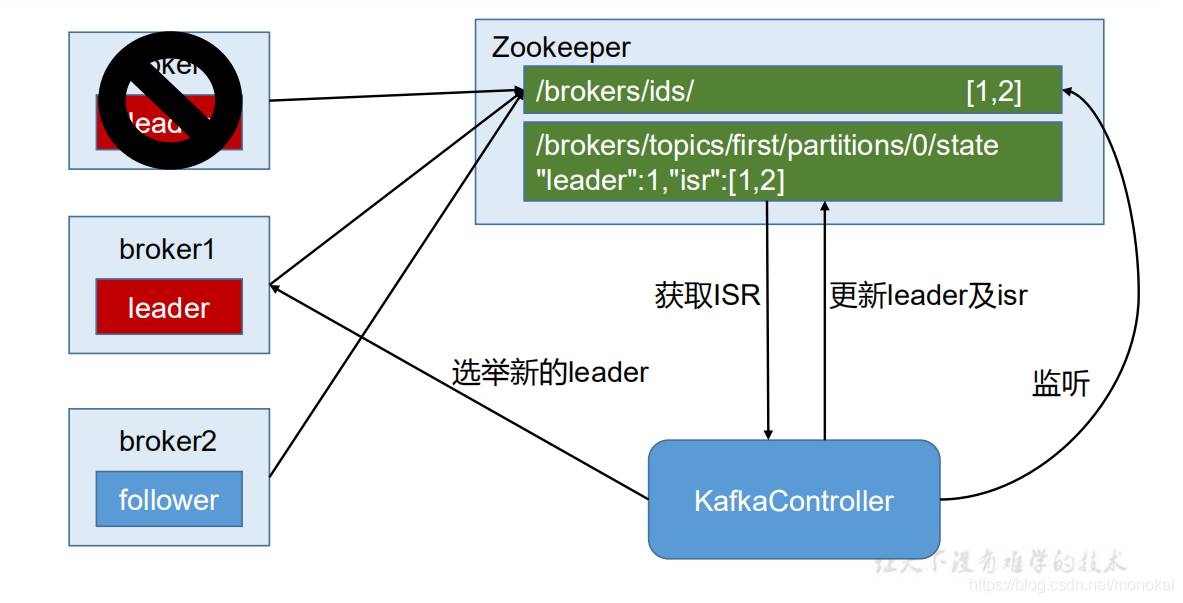
### **2）零复制技术**

零拷贝，这个不太懂。没有去深究（如果描述有错，希望指正）。大概是，我们的工作空间是User space,正常的IO流我们从内核到取文件，经过操作系统的一个kemel space,经过拷贝等操作写回kemel space再给内核。但是kafka，不经过 user space 直接在操作内核完成就是kemel space完成，效率高。  


## **3.5 Zookeeper 在 Kafka 中的作用**

Kafka 集群中有一个 broker 会被选举为 Controller，负责管理集群 broker 的上下线，所  
有 topic 的分区副本分配和leader 选举等工作。

****Controller 的管理工作都是依赖于 Zookeeper 的。****

以下为 partition 的 leader 选举过程：

## **3.6 Kafka 事务**

Kafka 从 0.11 版本开始引入了事务支持。事务可以保证 Kafka 在 Exactly Once 语义的基础上，生产和消费可以跨分区和会话，要么全部成功，要么全部失败。

### **3.6.1 Producer 事务**

为了实现跨分区跨会话的事务，需要引入一个全局唯一的 Transaction ID，并将 Producer  
获得的PID 和Transaction ID 绑定。这样当Producer 重启后就可以通过正在进行的Transaction ID 获得原来的 PID。为了管理 Transaction，Kafka 引入了一个新的组件Transaction Coordinator。Producer 就是通过和 Transaction Coordinator 交互获得 Transaction ID 对应的任务状态。Transaction Coordinator 还负责将事务所有写入 Kafka 的一个内部 Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行。

### **3.6.2 Consumer 事务**

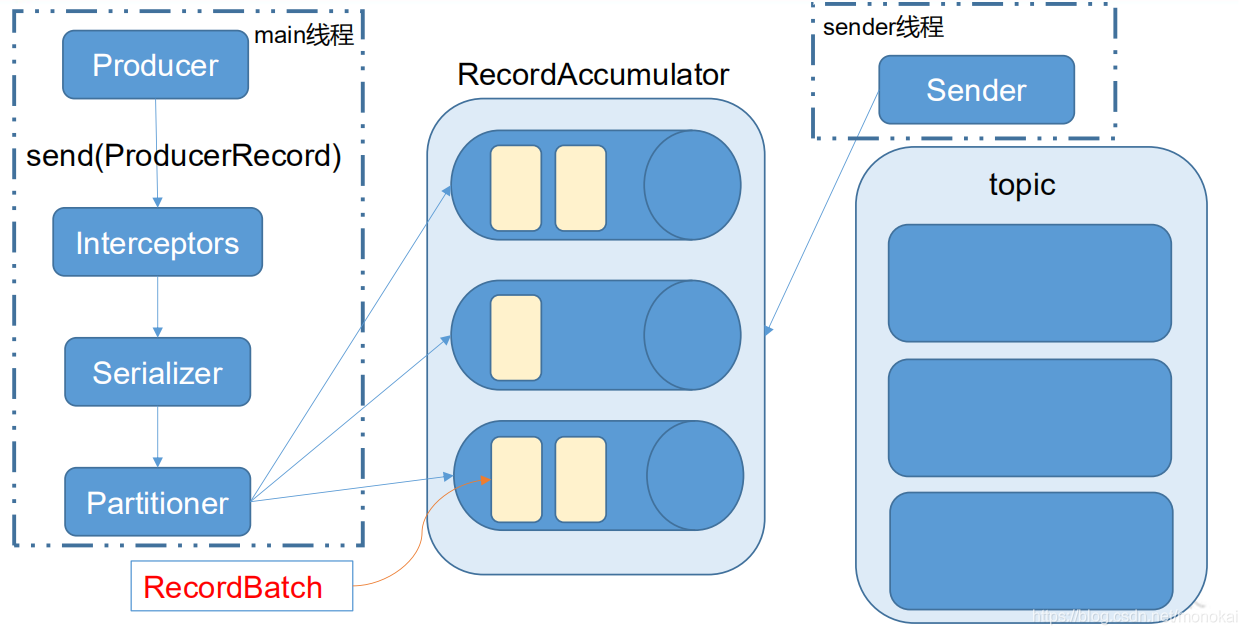
上述事务机制主要是从 Producer 方面考虑，对于 Consumer 而言，事务的保证就会相对较弱，尤其时无法保证 Commit 的信息被精确消费。这是由于 Consumer 可以通过 offset 访问任意信息，而且不同的 Segment File 生命周期不同，同一事务的消息可能会出现重启后被删除的情况。

# **第 4 章 Kafka API**

## **4.1 Producer API**

### **4.1.1 消息发送流程**

Kafka 的 Producer 发送消息采用的是异步发送的方式。在消息发送的过程中，涉及到了  
两个线程——main 线程和 Sender 线程，以及一个线程共享变量——RecordAccumulator。main 线程将消息发送给 RecordAccumulator，Sender 线程不断从 RecordAccumulator 中拉取消息发送到 Kafka broker。

KafkaProducer 发送消息流程如下:  
  
****相关参数****：

* batch.size：只有数据积累到 batch.size 之后，sender 才会发送数据。
* linger.ms：如果数据迟迟未达到 batch.size，sender 等待 linger.time 之后就会发送数据。

### **4.1.2 异步发送 API**

#### **1）导入依赖**

<dependency><groupId>org.apache.kafka</groupId><artifactId>kafka-clients</artifactId><version>0.11.0.0</version></dependency>

#### **2）编写代码**

需要用到的类：

* KafkaProducer：需要创建一个生产者对象，用来发送数据
* ProducerConfig：获取所需的一系列配置参数
* ProducerRecord：每条数据都要封装成一个 ProducerRecord 对象

##### **1.不带回调函数的 API**

package com.atguigu.kafka;import org.apache.kafka.clients.producer.\*;import java.util.Properties;import java.util.concurrent.ExecutionException;public class CustomProducer {

public static void main(String[] args) throws ExecutionException,

InterruptedException {

Properties props = new Properties();

*//kafka 集群，broker-list*

props.put("bootstrap.servers", "hadoop102:9092");

props.put("acks", "all");

*//重试次数*

props.put("retries", 1);

*//批次大小*

props.put("batch.size", 16384);

*//等待时间*

props.put("linger.ms", 1);

*//RecordAccumulator 缓冲区大小*

props.put("buffer.memory", 33554432);

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

for (int i = 0; i < 100; i++) {

producer.send(new ProducerRecord<String, String>("first",

Integer.toString(i), Integer.toString(i)));

}

producer.close();

} }

##### **2.带回调函数的 API**

回调函数会在 producer 收到 ack 时调用，为异步调用，该方法有两个参数，分别是  
RecordMetadata 和 Exception，如果 Exception 为 null，说明消息发送成功，如果  
Exception 不为 null，说明消息发送失败。

注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。

package com.atguigu.kafka;import org.apache.kafka.clients.producer.\*;import java.util.Properties;import java.util.concurrent.ExecutionException;public class CustomProducer {public static void main(String[] args) throws ExecutionException,

InterruptedException {

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");*//kafka 集*

群，broker-list

props.put("acks", "all");

props.put("retries", 1);*//重试次数*

props.put("batch.size", 16384);*//批次大小*

props.put("linger.ms", 1);*//等待时间*

props.put("buffer.memory", 33554432);*//RecordAccumulator 缓*

冲区大小

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

for (int i = 0; i < 100; i++) {

producer.send(new ProducerRecord<String, String>("first",

Integer.toString(i), Integer.toString(i)), new Callback() {

*//回调函数，该方法会在 Producer 收到 ack 时调用，为异步调用*

@Override

public void onCompletion(RecordMetadata metadata,

Exception exception) {

if (exception == null) {

System.out.println("success->" +

metadata.offset());

} else {

exception.printStackTrace();

}

}

});

}

producer.close();

} }

### **4.1.3 同步发送 API**

同步发送的意思就是，一条消息发送之后，会阻塞当前线程，直至返回 ack。

由于 send 方法返回的是一个 Future 对象，根据 Futrue 对象的特点，我们也可以实现同步发送的效果，只需在调用 Future 对象的 get 方发即可。

package com.atguigu.kafka;import org.apache.kafka.clients.producer.KafkaProducer;import org.apache.kafka.clients.producer.Producer;import org.apache.kafka.clients.producer.ProducerRecord;import java.util.Properties;import java.util.concurrent.ExecutionException;public class CustomProducer {public static void main(String[] args) throws ExecutionException,

InterruptedException {

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");*//kafka 集*

群，broker-list

props.put("acks", "all");

props.put("retries", 1);*//重试次数*

props.put("batch.size", 16384);*//批次大小*

props.put("linger.ms", 1);*//等待时间*

props.put("buffer.memory", 33554432);*//RecordAccumulator 缓*

冲区大小

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

for (int i = 0; i < 100; i++) {

producer.send(new ProducerRecord<String, String>("first",

Integer.toString(i), Integer.toString(i))).get();

}

producer.close();

} }

## **4.2 Consumer API**

Consumer 消费数据时的可靠性是很容易保证的，因为数据在 Kafka 中是持久化的，故  
不用担心数据丢失问题。

由于 consumer 在消费过程中可能会出现断电宕机等故障，consumer 恢复后，需要从故  
障前的位置的继续消费，所以 consumer 需要实时记录自己消费到了哪个 offset，以便故障恢  
复后继续消费。

所以 offset 的维护是 Consumer 消费数据是必须考虑的问题。

### **4.2.1 自动提交 offset**

#### **1）导入依赖**

<dependency><groupId>org.apache.kafka</groupId><artifactId>kafka-clients</artifactId><version>0.11.0.0</version></dependency>

#### **2）编写代码**

需要用到的类：

* KafkaConsumer：需要创建一个消费者对象，用来消费数据
* ConsumerConfig：获取所需的一系列配置参数
* ConsuemrRecord：每条数据都要封装成一个 ConsumerRecord 对象

为了使我们能够专注于自己的业务逻辑，Kafka 提供了自动提交 offset 的功能。

自动提交 offset 的相关参数：

* enable.auto.commit：是否开启自动提交 offset 功能
* auto.commit.interval.ms：自动提交 offset 的时间间隔  
  以下为自动提交 offset 的代码：

package com.atguigu.kafka;import org.apache.kafka.clients.consumer.ConsumerRecord;import org.apache.kafka.clients.consumer.ConsumerRecords;import org.apache.kafka.clients.consumer.KafkaConsumer;import java.util.Arrays;import java.util.Properties;public class CustomConsumer {public static void main(String[] args) {

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");

props.put("group.id", "test");

props.put("enable.auto.commit", "true");

props.put("auto.commit.interval.ms", "1000");

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = newKafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("first"));

while (true) {

ConsumerRecords<String, String> records =

consumer.poll(100);

for (ConsumerRecord<String, String> record : records)

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

}

} }

### **4.2.2 手动提交 offset**

虽然自动提交 offset 十分简介便利，但由于其是基于时间提交的，开发人员难以把握  
offset 提交的时机。因此 Kafka 还提供了手动提交 offset 的 API。

手动提交 offset 的方法有两种：分别是 commitSync（同步提交）和 commitAsync（异步 提交）。

* 两者的相同点是，都会将本次 poll 的一批数据最高的偏移量提交；
* 不同点是，commitSync 阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；而 commitAsync 则没有失败重试机制，故有可能提交失败。

#### **1）同步提交 offset**

由于同步提交 offset 有失败重试机制，故更加可靠，以下为同步提交 offset 的示例。

package com.atguigu.kafka.consumer;import org.apache.kafka.clients.consumer.ConsumerRecord;import org.apache.kafka.clients.consumer.ConsumerRecords;import org.apache.kafka.clients.consumer.KafkaConsumer;import java.util.Arrays;import java.util.Properties;public class CustomComsumer {

public static void main(String[] args) {

Properties props = new Properties();*//Kafka 集群*

props.put("bootstrap.servers", "hadoop102:9092");*//消费者组，只要 group.id 相同，就属于同一个消费者组*

props.put("group.id", "test");

props.put("enable.auto.commit", "false");*//关闭自动提交 offset*

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("first"));*//消费者订阅主题*

while (true) {*//消费者拉取数据*

ConsumerRecords<String, String> records =

consumer.poll(100);

for (ConsumerRecord<String, String> record : records) {

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

}*//同步提交，当前线程会阻塞直到 offset 提交成功*

consumer.commitSync();

}

} }

#### **2）异步提交 offset**

虽然同步提交 offset 更可靠一些，但是由于其会阻塞当前线程，直到提交成功。因此吞  
吐量会收到很大的影响。因此更多的情况下，会选用异步提交 offset 的方式。

以下为异步提交 offset 的示例：

package com.atguigu.kafka.consumer;import org.apache.kafka.clients.consumer.\*;import org.apache.kafka.common.TopicPartition;import java.util.Arrays;import java.util.Map;import java.util.Properties;public class CustomConsumer {

public static void main(String[] args) {

Properties props = new Properties();

*//Kafka 集群*

props.put("bootstrap.servers", "hadoop102:9092");

*//消费者组，只要 group.id 相同，就属于同一个消费者组*

props.put("group.id", "test");*//关闭自动提交 offset*

props.put("enable.auto.commit", "false");

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("first"));*//消费者订阅主题*

while (true) {

ConsumerRecords<String, String> records =

consumer.poll(100);*//消费者拉取数据*

for (ConsumerRecord<String, String> record : records) {

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

}*//异步提交*

consumer.commitAsync(new OffsetCommitCallback() {

@Override

public void onComplete(Map<TopicPartition,

OffsetAndMetadata> offsets, Exception exception) {

if (exception != null) {

System.err.println("Commit failed for" +

offsets);

}

}

});

}

} }

#### **3） 数据漏消费和重复消费分析**

无论是同步提交（虽然有失败重试）还是异步提交 offset，都有可能会造成数据的漏消费或者重复消费。先提交 offset 后消费，有可能造成数据的漏消费；而先消费后提交 offset，有可能会造成数据的重复消费。（大概情况可能是，同步或者异步在消费完消息后，向kafka提交最新的offset的是 GG掉，这时kafka存储的还是上次的offset 会再次拉取相同的消息消费，造成重复消费）（漏消费是，取得消息但是，消费时间太长，已经将最新的offset提交给kafka但是此时服务器挂掉，Kafka以为你已经消费了，所以导致了数据漏消费）

### **4.2.3 自定义存储 offset**

Kafka 0.9 版本之前，offset 存储在 zookeeper，0.9 版本及之后，默认将 offset 存储在 Kafka的一个内置的 topic \_consumer\_offset 中。除此之外，Kafka 还可以选择自定义存储 offset。

offset 的维护是相当繁琐的，因为需要考虑到消费者的 Rebalace。

当有新的消费者加入消费者组、已有的消费者推出消费者组或者所订阅的主题的分区发 生变化，就会触发到分区的重新分配，重新分配的过程叫做 Rebalance。

消费者发生 Rebalance 之后，每个消费者消费的分区就会发生变化。因此消费者要首先获取到自己被重新分配到的分区，并且定位到每个分区最近提交的 offset 位置继续消费。

要实现自定义存储 offset，需要借助 ConsumerRebalanceListener，以下为示例代码，其  
中提交和获取 offset 的方法，需要根据所选的 offset 存储系统自行实现。

package com.atguigu.kafka.consumer;import org.apache.kafka.clients.consumer.\*;import org.apache.kafka.common.TopicPartition;import java.util.\*;public class CustomConsumer {

private static Map<TopicPartition, Long> currentOffset = new HashMap<>();public static void main(String[] args) {*//创建配置信息*

Properties props = new Properties();*//Kafka 集群*

props.put("bootstrap.servers", "hadoop102:9092");*//消费者组，只要 group.id 相同，就属于同一个消费者组*

props.put("group.id", "test");*//关闭自动提交 offset*

props.put("enable.auto.commit", "false");

*//Key 和 Value 的反序列化类*

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

*//创建一个消费者*

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

*//消费者订阅主题*

consumer.subscribe(Arrays.asList("first"), new ConsumerRebalanceListener() {

*//该方法会在 Rebalance 之前调用*

@Override

public void onPartitionsRevoked(Collection<TopicPartition> partitions) {

commitOffset(currentOffset);

}

*//该方法会在 Rebalance 之后调用*

@Override

public void onPartitionsAssigned(Collection<TopicPartition> partitions) {currentOffset.clear();

for (TopicPartition partition : partitions) {

consumer.seek(partition, getOffset(partition));*//*

定位到最近提交的 offset 位置继续消费

}

}

});

while (true) {

ConsumerRecords<String, String> records =

consumer.poll(100);*//消费者拉取数据*

for (ConsumerRecord<String, String> record : records) {

System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

currentOffset.put(new TopicPartition(record.topic(),

record.partition()), record.offset());

}

commitOffset(currentOffset);*//异步提交*

}

}

*//获取某分区的最新 offset*

private static long getOffset(TopicPartition partition) {

return 0;

}

*//提交该消费者所有分区的 offset*

private static void commitOffset(Map<TopicPartition, Long>

currentOffset) {

} }

## **4.3 自定义 Interceptor**

### **4.3.1 拦截器原理**

Producer 拦截器(interceptor)是在 Kafka 0.10 版本被引入的，主要用于实现 clients 端的定制化控制逻辑。对于 producer 而言，interceptor 使得用户在消息发送前以及 producer 回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。同时，producer 允许用户指定多个 interceptor按序作用于同一条消息从而形成一个拦截链(interceptor chain)。Intercetpor 的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括：

（1）configure(configs)  
获取配置信息和初始化数据时调用。

（2）onSend(ProducerRecord)：  
该方法封装进 KafkaProducer.send 方法中，即它运行在用户主线程中。Producer 确保在消息被序列化以及计算分区前调用该方法。用户可以在该方法中对消息做任何操作，但最好 保证不要修改消息所属的 topic 和分区，否则会影响目标分区的计算。

（3）onAcknowledgement(RecordMetadata, Exception)：  
该方法会在消息从 RecordAccumulator 成功发送到 Kafka Broker 之后，或者在发送过程 中失败时调用。并且通常都是在 producer 回调逻辑触发之前。onAcknowledgement 运行在  
producer 的 IO 线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢 producer 的消息发送效率。  
(4）close：  
关闭 interceptor，主要用于执行一些资源清理工作

如前所述，interceptor 可能被运行在多个线程中，因此在具体实现时用户需要自行确保  
线程安全。另外倘若指定了多个 interceptor，则 producer 将按照指定顺序调用它们，并仅仅是捕获每个 interceptor 可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

### **4.3.2 拦截器案例**

#### **1）需求：**

实现一个简单的双 interceptor 组成的拦截链。第一个 interceptor 会在消息发送前将时间戳信息加到消息 value 的最前部；第二个 interceptor 会在消息发送后更新成功发送消息数或失败发送消息数。

Kafka拦截器

#### **2）案例实操**

（1）增加时间戳拦截器

package com.atguigu.kafka.interceptor;import java.util.Map;import org.apache.kafka.clients.producer.ProducerInterceptor;import org.apache.kafka.clients.producer.ProducerRecord;import org.apache.kafka.clients.producer.RecordMetadata;public class TimeInterceptor implements ProducerInterceptor<String, String> {@Overridepublic void configure(Map<String, ?> configs) {}@Overridepublic ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {*// 创建一个新的 record，把时间戳写入消息体的最前部*return new ProducerRecord(record.topic(),

record.partition(), record.timestamp(), record.key(),

System.currentTimeMillis() + "," +

record.value().toString());}@Overridepublic void onAcknowledgement(RecordMetadata metadata,

Exception exception) {}@Overridepublic void close() {} }

（2）统计发送消息成功和发送失败消息数，并在 producer 关闭时打印这两个计数器

package com.atguigu.kafka.interceptor;import java.util.Map;import org.apache.kafka.clients.producer.ProducerInterceptor;import org.apache.kafka.clients.producer.ProducerRecord;import org.apache.kafka.clients.producer.RecordMetadata;public class CounterInterceptor implements ProducerInterceptor<String, String>{

private int errorCounter = 0;

private int successCounter = 0;@Overridepublic void configure(Map<String, ?> configs) {}@Overridepublic ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {return record;}@Overridepublic void onAcknowledgement(RecordMetadata metadata,

Exception exception) {*// 统计成功和失败的次数*

if (exception == null) {

successCounter++;

} else {

errorCounter++;

} }@Overridepublic void close() {

*// 保存结果*

System.out.println("Successful sent: " + successCounter);

System.out.println("Failed sent: " + errorCounter);} }

（3）producer 主程序

package com.atguigu.kafka.interceptor;import java.util.ArrayList;import java.util.List;import java.util.Properties;import org.apache.kafka.clients.producer.KafkaProducer;import org.apache.kafka.clients.producer.Producer;import org.apache.kafka.clients.producer.ProducerConfig;import org.apache.kafka.clients.producer.ProducerRecord;public class InterceptorProducer {public static void main(String[] args) throws Exception {*// 1 设置配置信息*

Properties props = new Properties();

props.put("bootstrap.servers", "hadoop102:9092");

props.put("acks", "all");

props.put("retries", 3);

props.put("batch.size", 16384);

props.put("linger.ms", 1);

props.put("buffer.memory", 33554432);

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");*// 2 构建拦截链*

List<String> interceptors = new ArrayList<>();

interceptors.add("com.atguigu.kafka.interceptor.TimeInterce

ptor");

interceptors.add("com.atguigu.kafka.interceptor.CounterInterceptor");

props.put(ProducerConfig.INTERCEPTOR\_CLASSES\_CONFIG,

interceptors);

String topic = "first";

Producer<String, String> producer = new KafkaProducer<>(props);*// 3 发送消息*for (int i = 0; i < 10; i++) {

ProducerRecord<String, String> record = new ProducerRecord<>(topic, "message" + i);

producer.send(record);}*// 4 一定要关闭 producer，这样才会调用 interceptor 的 close 方法*

producer.close();} }

#### **3）测试**

（1）在 kafka 上启动消费者，然后运行客户端 java 程序

bin/kafka-console-consumer.sh \

--bootstrap-server hadoop102:9092 --from-beginning --topic first

1501904047034,message0

1501904047225,message1

1501904047230,message2

1501904047234,message3

1501904047236,message4

1501904047240,message5

1501904047243,message6

1501904047246,message7

1501904047249,message8

1501904047252,message9

# **第 5 章 Kafka 监控**

## **5.1 Kafka Eagle**

****1.修改 kafka 启动命令****  
修改kafka-server-start.sh命令中

if [ "x$KAFKA\_HEAP\_OPTS" = "x" ]; then

export KAFKA\_HEAP\_OPTS="-Xmx1G -Xms1G"fi

if [ "x$KAFKA\_HEAP\_OPTS" = "x" ]; then

export KAFKA\_HEAP\_OPTS="-server -Xms2G -Xmx2G -XX:PermSize=128m

-XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=8 -

XX:ConcGCThreads=5 -XX:InitiatingHeapOccupancyPercent=70"

export JMX\_PORT="9999"

*#export KAFKA\_HEAP\_OPTS="-Xmx1G -Xms1G"*fi

注意：修改之后在启动 Kafka 之前要分发之其他节点

****2.上传压缩包 kafka-eagle-bin-1.3.7.tar.gz 到集群/opt/software 目录****

****3.解压到本地****

tar -zxvf kafka-eagle-bin-1.3.7.tar.gz

****4.进入刚才解压的目录****

ll

总用量 82932

-rw-rw-r--. 1 atguigu atguigu 84920710 8 月 13 23:00 kafka-eagleweb-1.3.7-bin.tar.gz

****5.将 kafka-eagle-web-1.3.7-bin.tar.gz 解压至/opt/module****

tar -zxvf kafka-eagleweb-1.3.7-bin.tar.gz -C /opt/module/

****6.修改名称****

mv kafka-eagle-web-1.3.7/ eagle

****7.给启动文件执行权限****

cd bin/

ll

总用量 12

-rw-r--r--. 1 atguigu atguigu 1848 8 月 22 2017 ke.bat

-rw-r--r--. 1 atguigu atguigu 7190 7 月 30 20:12 ke.shchmod 777 ke.sh

****8.修改配置文件****

######################################

# multi zookeeper&kafka cluster list

######################################

kafka.eagle.zk.cluster.alias=cluster1

cluster1.zk.list=hadoop102:2181,hadoop103:2181,hadoop104:2181

######################################

# kafka offset storage

######################################

cluster1.kafka.eagle.offset.storage=kafka

######################################

# enable kafka metrics

######################################

kafka.eagle.metrics.charts=true

kafka.eagle.sql.fix.error=false

######################################

# kafka jdbc driver address

######################################

kafka.eagle.driver=com.mysql.jdbc.Driver

kafka.eagle.url=jdbc:mysql://hadoop102:3306/ke?useUnicode=true&ch

aracterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull

kafka.eagle.username=root

kafka.eagle.password=000000

****9.添加环境变量****

export KE\_HOME=/opt/module/eagleexport PATH=$PATH:$KE\_HOME/bin

``

注意：source /etc/profile

\*\*10.启动\*\*```bash

$ bin/ke.sh start... ...... ...

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*

\* Kafka Eagle Service has started success.

\* Welcome, Now you can visit 'http://192.168.9.102:8048/ke'

\* Account:admin ,Password:123456

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*

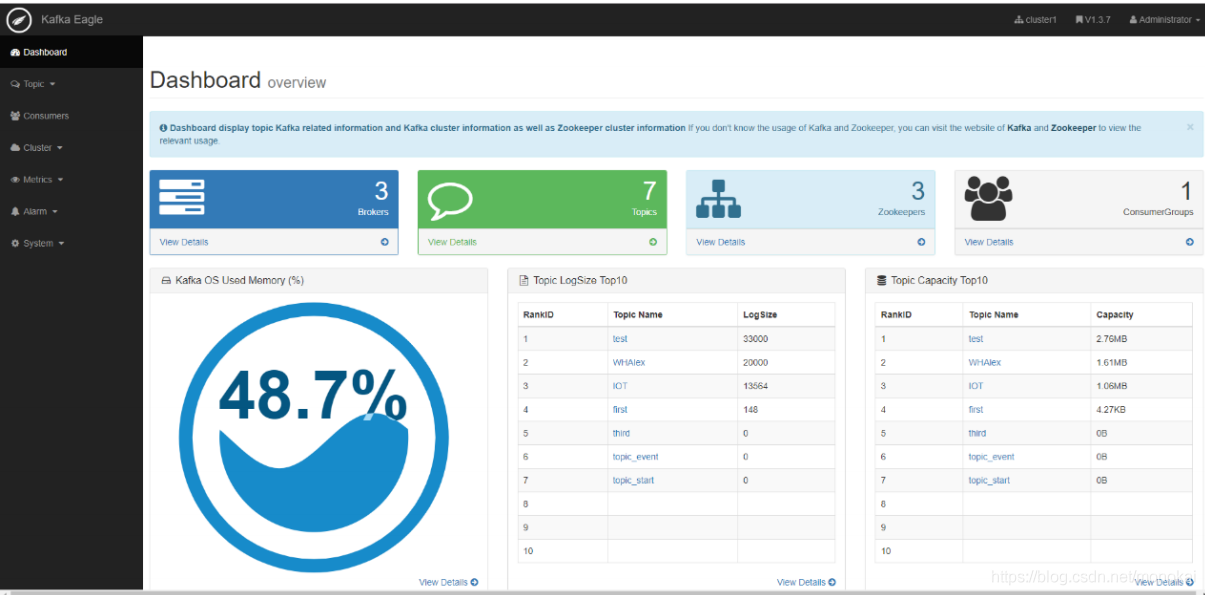
\* <Usage> ke.sh [start|status|stop|restart|stats] </Usage>

\* <Usage> https://www.kafka-eagle.org/ </Usage>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*

注意：启动之前需要先启动 ZK 以及 KAFKA

****11.登录页面查看监控数据****  
http://10.10.10.102:8048/ke  


# **第 6 章 Flume 对接 Kafka**

****1）配置 flume(flume-kafka.conf)****

# define

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# source

a1.sources.r1.type = exec

a1.sources.r1.command = tail -F -c +0 /opt/module/data/flume.log

a1.sources.r1.shell = /bin/bash -c

# sink

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.kafka.bootstrap.servers =

hadoop102:9092,hadoop103:9092,hadoop104:9092

a1.sinks.k1.kafka.topic = first

a1.sinks.k1.kafka.flumeBatchSize = 20

a1.sinks.k1.kafka.producer.acks = 1

a1.sinks.k1.kafka.producer.linger.ms = 1

# channel

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# bind

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

****3） 进入 flume 根目录下，启动 flume****

$ bin/flume-ng agent -c conf/ -n a1 -f jobs/flume-kafka.conf

* 1

****4） 向 /opt/module/data/flume.log 里追加数据，查看 kafka 消费者消费情况****

$ echo hello >> /opt/module/data/flume.log

* 1

# **第 7 章 Kafka 面试题**

## **7.1 面试问题**

1. Kafka 中的 ISR(InSyncRepli)、OSR(OutSyncRepli)、AR(AllRepli)代表什么？

ISR：在设定的时间内完成与leader同步的follow集合

OSR：那么没有完成的follow存在哪里 肯定就是OSR啊

AR：应该是指副本总和 ISR+OSR

1. Kafka 中的 HW、LEO 等分别代表什么？

HW：高水位，leader与Follow同步时，因为同步有差异，各个follow同步的消息数目不一致，取最小的LEO。最后一个位置+1

LEO：每个follow中最大的offset+1(紧紧能保证分区有序性)

LSO：事务中引入LSO<=HW<=LEO

在事务完成时LSO=HW ,其他情况下为事务的第一条数据

1. Kafka 中是怎么体现消息顺序性的？

kafka不能够保证全局消息的有序性，仅能保证分区消息有序性。依次进入队列，每个分区独立维护自己的offset。因为在不同的Broker上（这句话不太严密）。

1. Kafka 中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？
2. 拦截器：内有四个方法，config、onsend(在分区前进行处理，能够对数据体进行操作，但是不建议动分区、主题)、onAcknowledgement（回调函数，可以做成功回调，失败回调逻辑处理，但是不建议有重逻辑，会使得提交消息缓慢）、close、
3. 序列化器：将数据序列化（在消费者端再反序列化）
4. 分区器：（根据一定的分区策略，确定自己进入哪个分区）
5. Kafka 生产者客户端的整体结构是什么样子的？使用了几个线程来处理？分别是什么？

客户端整体架构，main 线程producer->拦截器->序列化器->分区器->recordAccoumend缓存区

Sender线程 从缓存区拉取数据 给kafka。

1. “消费组中的消费者个数如果超过 topic 的分区，那么就会有消费者消费不到数据”这句话是否正确？

正确。

1. 消费者提交消费位移时提交的是当前消费到的最新消息的 offset 还是 offset+1？

Offset+1

1. 有哪些情形会造成重复消费？

重复消费，一定发生在消费者端。消费者端，同步或者异步，在接受到消息后进行处理，处理已经完成，向kafka写最新的offset时，服务器挂掉。这时因为kafka中存储的该offset还是旧的就会造成重复消费。造成这种的原因很简单，kafka自动提交offset，是根据时间处理提交，并不是根据处理结果提交。

（解决方案）

关闭自动提交，在完成自己的业务逻辑处理后提交。但是好像个人感觉也无法避免重复消费问题。但是要比自动提交好的多，比如你设定自动提交offset 3s提交 ,业务逻辑只用了0.5s完成处理，在2s时服务器炸了，这时就会重复消费，但是你改了自定义提交。除非运气超级差，处理完写入kafka的时候直接炸，都能避免问题。

1. 那些情景会造成消息漏消费？

消息漏消费，消息体大，消费者还正在处理，但是已经将最新获取 到的offset写给了，kafka但是这个时候 服务器GG了，那么这个消息kafka认为你已经消费了，不会重复给你发送，就造成了消息的漏消费问题。造成这种的原因很简单，kafka自动提交offset，是根据时间处理提交，并不是根据处理结果提交。

（解决方案）

offset手动提交，业务成功处理后，提交offset。

1. 生产者消息丢失处理方案？消息乱序的可能性处理？

异步消息丢失，为了防止缓冲区满，可以在配置文件设置不限制阻塞超时时间，当缓存区满时让生产者一直处理阻塞状态。同步就简单了，ack设置为-1。

设置max.in.flight.request.per.connection=1对一个broker来讲，如果上一个请求没有响应，不允许client像同一个Broker发送请求。

1. 当你使用 kafka-topics.sh 创建（删除）了一个 topic 之后，Kafka 背后会执行什么逻辑？  
   1）会在 zookeeper 中的/brokers/topics 节点下创建一个新的 topic 节点，如：  
   /brokers/topics/first  
   2）触发 Controller 的监听程序  
   3）kafka Controller 负责 topic 的创建工作，并更新 metadata cache
2. topic 的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

可以增加，就是用kafka-topics.sh动态管理分区。

1. topic 的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

但是绝对不可能减少。减少的问题。首先，该分区的数据 存储到哪里，如果存到其他分期，立马将各个分区的消息顺序破坏了。删除的话，存在数据丢失风险。

1. Kafka 有内部的 topic 吗？如果有是什么？有什么所用？

kafka有内部的topic么？没有看太懂这个问题的想问什么。Topic是一个逻辑概念，主题相关性，消息发布订阅的一个主体。

1. Kafka 分区分配的概念？

分区分配：应该指的是

生产者如何传消息到哪个分区：有指定走指定，有Key走key，都没 随机选择一个开始轮询分配。

消费者如何确定自己消费哪个分区的消息：消费者组订阅主题，根据range策略分配分区。

1. 简述 Kafka 的日志目录结构？

日志目录结构，即使kafka文件存储系统。

在topic下分区后，进行了分段segment

每个segment都有.index与.log，这里的log不是日志，而是数据，上文中Message结构。

1. 如果我指定了一个 offset，Kafka Controller 怎么查找到对应的消息？

指定一个offset，二分查找找到属于哪个segment，确定之后进入索引，二分查找找到最想相近的两个offset（稀疏存储的原因，也可能直接找到），然后再该范围内从前向后读取到自己想要的数据。（稀疏存储就是用一点点时间，换取空间的效率）

1. 聊一聊 Kafka Controller 的作用？

Kafka Controller 是一个broker 负责分区分配和leader选举。管理集群的上下线。

1. Kafka 中有那些地方需要选举？这些地方的选举策略又有哪些？

<https://blog.csdn.net/yanshu2012/article/details/54894629>

新leader,新controller，好像就这两个（如果ISR那个集合不算）

选举策略

leader在zk上创建一个临时节点，所有的follow都对该节点注册监听，当leader GG 时，所有的follow都尝试创建该节点，创建成功的成为Leader。

controller同理，但是controller控制选举leader，但是很复杂，思路如上。

两种情况 ：在ISR中选一个成为leader;（数据可靠性）

服务全挂了，允许第一个活过来的服务成为leader不一定是isr；（高可用性）

1. 失效副本是指什么？有那些应对措施？

失效副本，指的时没有在设定时间内完成同步的副本，失效副本一直在追赶leader副本

应对措施让他滚蛋，哈哈。其实不是，有一个优先副本选举。

1. Kafka 的哪些设计让它有如此高的性能？

Cache Filesystem Cache PageCache缓存；顺序写 由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。；Zero-copy 零拷技术减少拷贝次数；Batching of Messages 批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。；Pull 拉模式 使用拉模式进行消息的获取消费，与消费端处理能力相符。

1. kafka中的broker 是干什么的

broker 是消息的代理，Producers往Brokers里面的指定Topic中写消息，Consumers从Brokers里面拉取指定Topic的消息，然后进行业务处理，broker在中间起到一个代理保存消息的中转站。

1. kafka中的 zookeeper 起到什么作用，可以不用zookeeper么

zookeeper 是一个分布式的协调组件，早期版本的kafka用zk做meta信息存储，consumer的消费状态，group的管理以及 offset的值。考虑到zk本身的一些因素以及整个架构较大概率存在单点问题，新版本中逐渐弱化了zookeeper的作用。新的consumer使用了kafka内部的group coordination协议，也减少了对zookeeper的依赖，但是broker依然依赖于ZK，zookeeper 在kafka中还用来选举controller 和 检测broker是否存活等等。

1. kafka follower如何与leader同步数据

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。完全同步复制要求All Alive 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。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leader挂掉，会丢失数据，kafka使用ISR的方式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，而且Leader充分利用磁盘顺序读以及send file(zero copy)机制，这样极大的提高复制性能，内部批量写磁盘，大幅减少了Follower与Leader的消息量差。

1. kafka producer如何优化打入速度

增加线程、提高 batch.size、增加更多 producer 实例、增加 partition 数、设置 acks=-1 时，如果延迟增大：可以增大 num.replica.fetchers（follower 同步数据的线程数）来调解；跨数据中心的传输：增加 socket 缓冲区设置以及 OS tcp 缓冲区设置。

1. kafka  unclean 配置代表啥，会对 spark streaming 消费有什么影响

unclean.leader.election.enable 为true的话，意味着非ISR集合的broker 也可以参与选举，这样有可能就会丢数据，spark streaming在消费过程中拿到的 end offset 会突然变小，导致 spark streaming job挂掉。如果unclean.leader.election.enable参数设置为true，就有可能发生数据丢失和数据不一致的情况，Kafka的可靠性就会降低；而如果unclean.leader.election.enable参数设置为false，Kafka的可用性就会降低。

1. 如果leader crash时，ISR为空怎么办

kafka在Broker端提供了一个配置参数：unclean.leader.election,这个参数有两个值：  
true（默认）：允许不同步副本成为leader，由于不同步副本的消息较为滞后，此时成为leader，可能会出现消息不一致的情况。  
false：不允许不同步副本成为leader，此时如果发生ISR列表为空，会一直等待旧leader恢复，降低了可用性。

1. kafka的message格式是什么样的

一个Kafka的Message由一个固定长度的header和一个变长的消息体body组成

header部分由一个字节的magic(文件格式)和四个字节的CRC32(用于判断body消息体是否正常)构成。

当magic的值为1的时候，会在magic和crc32之间多一个字节的数据：attributes(保存一些相关属性，

比如是否压缩、压缩格式等等);如果magic的值为0，那么不存在attributes属性

body是由N个字节构成的一个消息体，包含了具体的key/value消息

1. kafka中consumer group 是什么概念

同样是逻辑上的概念，是Kafka实现单播和广播两种消息模型的手段。同一个topic的数据，会广播给不同的group；同一个group中的worker，只有一个worker能拿到这个数据。换句话说，对于同一个topic，每个group都可以拿到同样的所有数据，但是数据进入group后只能被其中的一个worker消费。group内的worker可以使用多线程或多进程来实现，也可以将进程分散在多台机器上，worker的数量通常不超过partition的数量，且二者最好保持整数倍关系，因为Kafka在设计时假定了一个partition只能被一个worker消费（同一group内）。

1. Kafka中的消息是否会丢失和重复消费？

要确定Kafka的消息是否丢失或重复，从两个方面分析入手：消息发送和消息消费。

1、消息发送

         Kafka消息发送有两种方式：同步（sync）和异步（async），默认是同步方式，可通过producer.type属性进行配置。Kafka通过配置request.required.acks属性来确认消息的生产：

0---表示不进行消息接收是否成功的确认；

1---表示当Leader接收成功时确认；

-1---表示Leader和Follower都接收成功时确认；

综上所述，有6种消息生产的情况，下面分情况来分析消息丢失的场景：

（1）acks=0，不和Kafka集群进行消息接收确认，则当网络异常、缓冲区满了等情况时，消息可能丢失；

（2）acks=1、同步模式下，只有Leader确认接收成功后但挂掉了，副本没有同步，数据可能丢失；

2、消息消费

Kafka消息消费有两个consumer接口，Low-level API和High-level API：

Low-level API：消费者自己维护offset等值，可以实现对Kafka的完全控制；

High-level API：封装了对parition和offset的管理，使用简单；

如果使用高级接口High-level API，可能存在一个问题就是当消息消费者从集群中把消息取出来、并提交了新的消息offset值后，还没来得及消费就挂掉了，那么下次再消费时之前没消费成功的消息就“诡异”的消失了；

解决办法：

        针对消息丢失：同步模式下，确认机制设置为-1，即让消息写入Leader和Follower之后再确认消息发送成功；异步模式下，为防止缓冲区满，可以在配置文件设置不限制阻塞超时时间，当缓冲区满时让生产者一直处于阻塞状态；

        针对消息重复：将消息的唯一标识保存到外部介质中，每次消费时判断是否处理过即可。

消息重复消费及解决参考：<https://www.javazhiyin.com/22910.html>

1. 为什么Kafka不支持读写分离？

在 Kafka 中，生产者写入消息、消费者读取消息的操作都是与 leader 副本进行交互的，从 而实现的是一种主写主读的生产消费模型。

Kafka 并不支持主写从读，因为主写从读有 2 个很明 显的缺点:

(1)数据一致性问题。数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间 窗口会导致主从节点之间的数据不一致。某一时刻，在主节点和从节点中 A 数据的值都为 X， 之后将主节点中 A 的值修改为 Y，那么在这个变更通知到从节点之前，应用读取从节点中的 A 数据的值并不为最新的 Y，由此便产生了数据不一致的问题。

(2)延时问题。类似 Redis 这种组件，数据从写入主节点到同步至从节点中的过程需要经 历网络→主节点内存→网络→从节点内存这几个阶段，整个过程会耗费一定的时间。而在 Kafka 中，主从同步会比 Redis 更加耗时，它需要经历网络→主节点内存→主节点磁盘→网络→从节 点内存→从节点磁盘这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。

1. Kafka中是怎么体现消息顺序性的？

kafka每个partition中的消息在写入时都是有序的，消费时，每个partition只能被每一个group中的一个消费者消费，保证了消费时也是有序的。  
整个topic不保证有序。如果为了保证topic整个有序，那么将partition调整为1.

1. kafka如何实现延迟队列？

Kafka并没有使用JDK自带的Timer或者DelayQueue来实现延迟的功能，而是基于时间轮自定义了一个用于实现延迟功能的定时器（SystemTimer）。JDK的Timer和DelayQueue插入和删除操作的平均时间复杂度为O(nlog(n))，并不能满足Kafka的高性能要求，而基于时间轮可以将插入和删除操作的时间复杂度都降为O(1)。时间轮的应用并非Kafka独有，其应用场景还有很多，在Netty、Akka、Quartz、Zookeeper等组件中都存在时间轮的踪影。

底层使用数组实现，数组中的每个元素可以存放一个TimerTaskList对象。TimerTaskList是一个环形双向链表，在其中的链表项TimerTaskEntry中封装了真正的定时任务TimerTask.

Kafka中到底是怎么推进时间的呢？Kafka中的定时器借助了JDK中的DelayQueue来协助推进时间轮。具体做法是对于每个使用到的TimerTaskList都会加入到DelayQueue中。Kafka中的TimingWheel专门用来执行插入和删除TimerTaskEntry的操作，而DelayQueue专门负责时间推进的任务。再试想一下，DelayQueue中的第一个超时任务列表的expiration为200ms，第二个超时任务为840ms，这里获取DelayQueue的队头只需要O(1)的时间复杂度。如果采用每秒定时推进，那么获取到第一个超时的任务列表时执行的200次推进中有199次属于“空推进”，而获取到第二个超时任务时有需要执行639次“空推进”，这样会无故空耗机器的性能资源，这里采用DelayQueue来辅助以少量空间换时间，从而做到了“精准推进”。Kafka中的定时器真可谓是“知人善用”，用TimingWheel做最擅长的任务添加和删除操作，而用DelayQueue做最擅长的时间推进工作，相辅相成。

参考：<https://blog.csdn.net/u013256816/article/details/80697456>

1. .Kafka中的事务是怎么实现的？

参考：<https://blog.csdn.net/u013256816/article/details/89135417>

参考：

1、尚硅谷kafka学习视频

2、<https://blog.csdn.net/monokai/article/details/106969361>