第一章 Kafka和消息队列概述

1.1 定义

- Kafka是一个分布式的流处理平台。
- 提供发布/订阅功能，充当**消息队列**；

1.2 Kafka特性

- 高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒；
- 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止丢失；
- 容错性：允许集群中的节点失败(若分区副本数量为n,则允许n-1个节点失败)；
- 高并发：单机可支持数千个客户端同时读写；
- 可扩展性：kafka集群支持热扩展；

1.3kafka的应用场景

- 日志收集：一个公司可以用Kafka收集各种服务的log，通过kafka以统一接口开放给各种消费端，例如hadoop、Hbase、Solr等。
- 消息系统：解耦生产者和消费者、缓存消息等。
- 用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索记录、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到hadoop、数据仓库中做离线分析和挖掘。
- 运营指标：Kafka也经常用来记录运营监控数据。
- 流式处理

1.4消息队列

要了解kafka先要了解消息队列。
在高并发的场景，大量的插入、更新请求同时到达数据库，会导致表会被锁住，请求堆积过多导致“连接数过多”异常。
所以在高并发场景需要一个缓冲机制，消息队列就是当作这个缓冲机制。

1.4.1消息队列定义

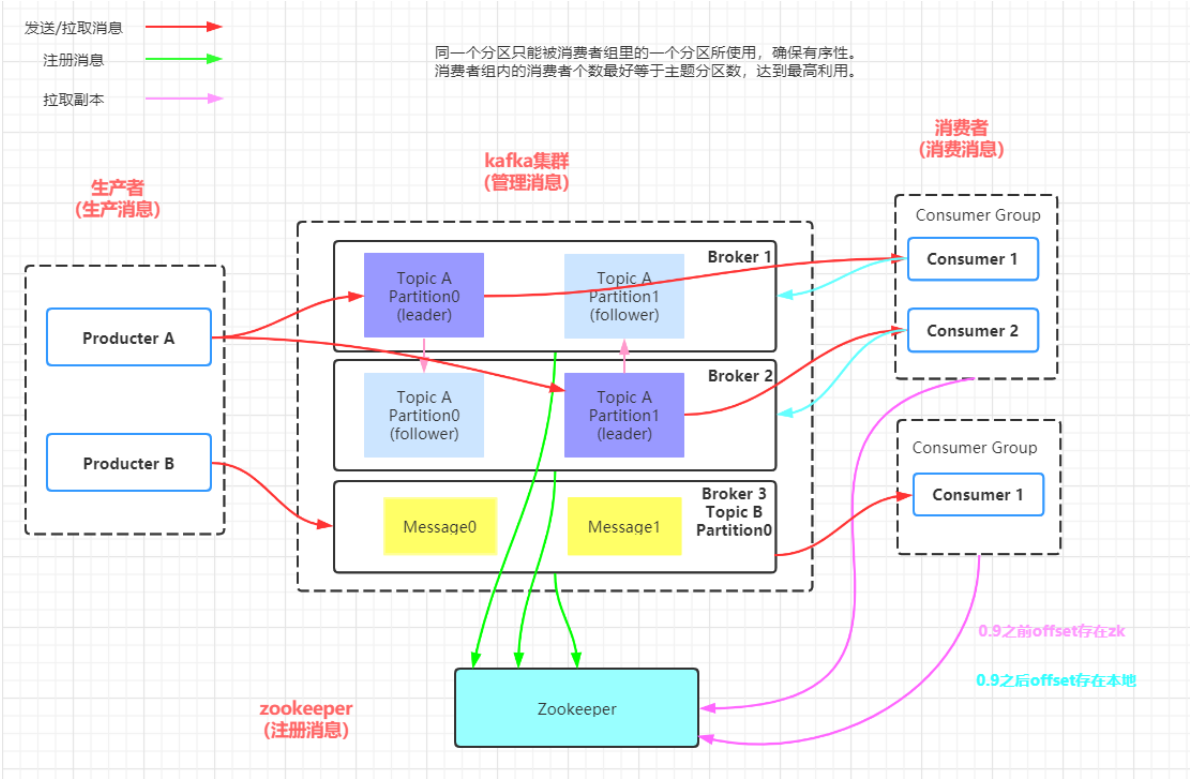
是一个传递消息的队列，是生产者和消费者之间的中间件,遵循先进先出的特性。

> **优点：解耦、异步、削峰/限流、消息通信**
> **缺点：为了实现高可用需要集群/分布式、消息队列挂掉数据丢失。**

1.4.2消息队列的两种模式

点对点（一对一）消息发送到Queue，消息只能消费一次
发布/订阅（一对多）消息发送到Topic，可以重复消费

1.5kafka基础架构

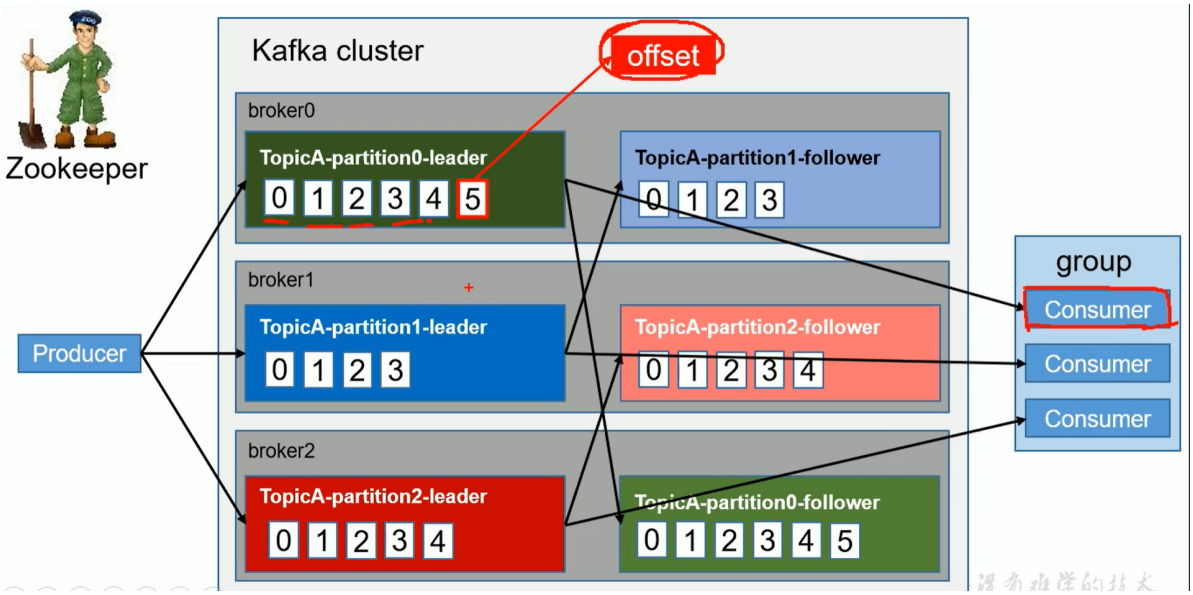


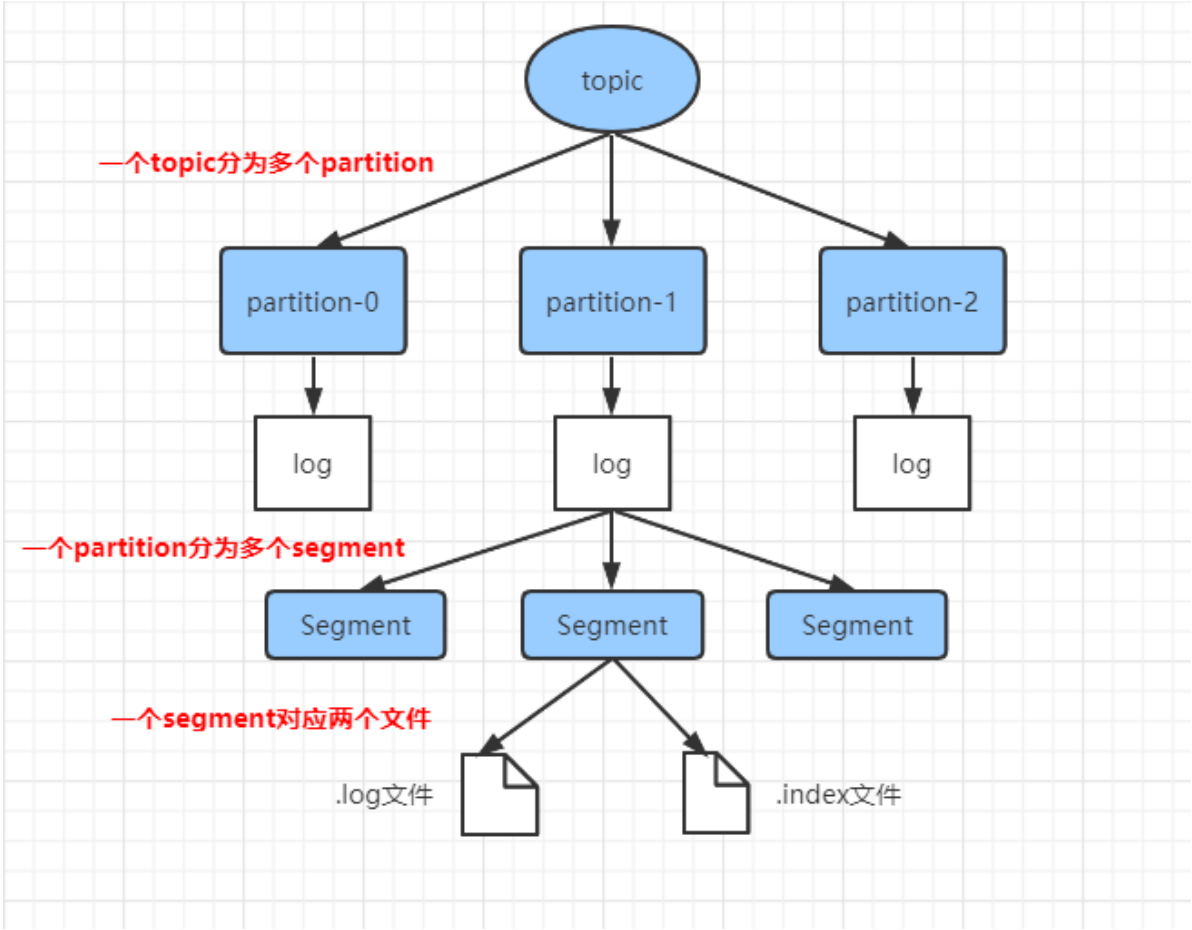
第二章 安装配置kafka

第三章 kafka架构深入

3.1 Kafka文件存储机制

每个分区都有独立的offset，不是全局的offset，保证区内有序。（如图建立3个分区，2个副本）



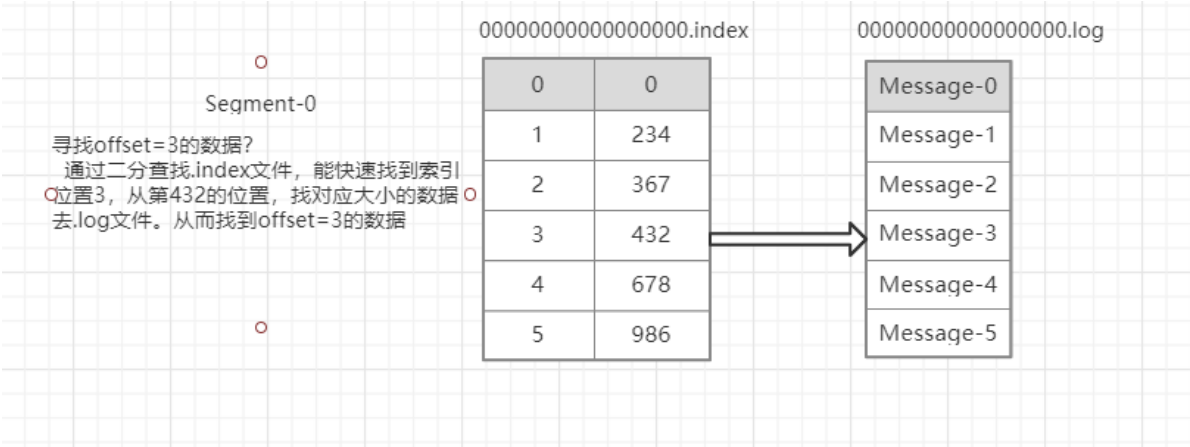


由于生产者生产的消息会不断追加到 log文件末尾，为防止 log 文件过大导致数据定位效率低下，Kafka采取了分片和索引机制，将每个partition分为多个segment。每个segment对应两个文件——".index"文件和".log"文件。这些文件位于一个文件夹下，该文件夹的命名规则为: topic名称+分区序号。例如，furst这个 topic有三个分区，则其对应的文件夹为first-0,first-1,first-2。

```
[root@node001 local]# cd kafka_2.13-2.6.0/
[root@node001 kafka_2.13-2.6.0]# cd logs/
[root@node001 logs]# ls
cleaner-offset-checkpoint  __consumer_offsets-16  __consumer_offsets-24  __consumer_offsets-34  __consumer_offsets-42  __consumer_offsets-8  kafkaServer-gc.
__consumer_offsets-0       __consumer_offsets-18  __consumer_offsets-26  __consumer_offsets-36  __consumer_offsets-44  controller.log        kafkaServer.out
__consumer_offsets-10      __consumer_offsets-2   __consumer_offsets-28  __consumer_offsets-38  __consumer_offsets-46  first-1               log-cleaner.log
__consumer_offsets-12      __consumer_offsets-20  __consumer_offsets-30  __consumer_offsets-4   __consumer_offsets-48  kafka-authorizer.log  log-start-offse
__consumer_offsets-14      __consumer_offsets-22  __consumer_offsets-32  __consumer_offsets-40  __consumer_offsets-6   kafka-request.log     meta.properties
[root@node001 logs]# cd first-1/
[root@node001 first-1]# ls
00000000000000000000.index  00000000000000000000.log  00000000000000000000.timeindex  00000000000000000001.snapshot  leader-epoch-checkpoint
[root@node001 first-1]# ll
总用量 16
-rw-r--r--. 1 root root  0 12月  2 10:41 00000000000000000000.index
-rw-r--r--. 1 root root 73 12月  2 10:37 00000000000000000000.log
-rw-r--r--. 1 root root 12 12月  2 10:41 00000000000000000000.timeindex
-rw-r--r--. 1 root root 10 12月  2 10:41 00000000000000000001.snapshot
-rw-r--r--. 1 root root  8 12月  2 10:36 leader-epoch-checkpoint
[root@node001 first-1]#
```

00000000000000000000.log文件 存放的是topic数据片段 默认存储的旧数据时长是7天 默认最大容量是1G

00000000000000000000.index文件 每个log文件的偏移量



index和 log文件以当前segment的第一条消息的offset命名。如果log数据大于1g，就分成多个segment。

```
00000000000000000000.log
00000000000000000000.index
00000000000000000212.log
00000000000000002122120.index
00000000000000132132121.log
00000000000000132132121.index
```

3.2 kafka生产者

生产者负责生产数据，提交到broker的指定topic的partition上。

3.2.1 分区策略

1. **分区的原因**

   方便在集群中扩展

   可以提高并发

2. **分区的策略**

我们需要将producer发送的数据封装成一个ProducerRecord对象

```
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, Long timestamp, String key, String value)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value, @Nullable Iterable<Header> headers)
ProducerRecord(@NotNull String topic, Integer partition, String key, String value)
ProducerRecord(@NotNull String topic, String key, String value)
ProducerRecord(@NotNull String topic, String value)
```

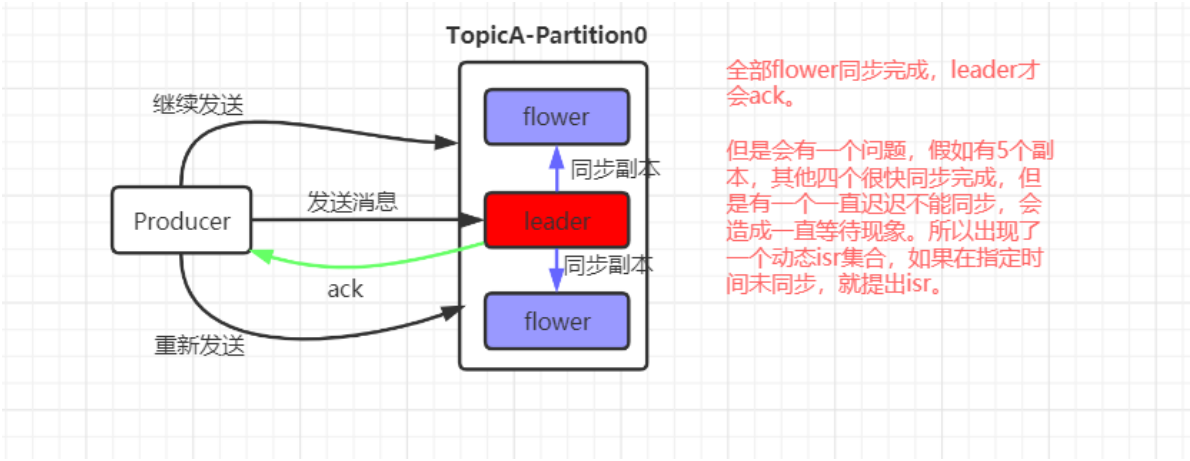
（1）指明 partition 的情况下，直接将指明的值直接作为 partiton 值；

（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），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

发送消息给leader如果leader接收消息，副本拉取成功，返回给producer一个成功ack，继续发送其他消息，否则重新发送。



1）数据同步策略

当所有的副本同步完成才会返回ack。

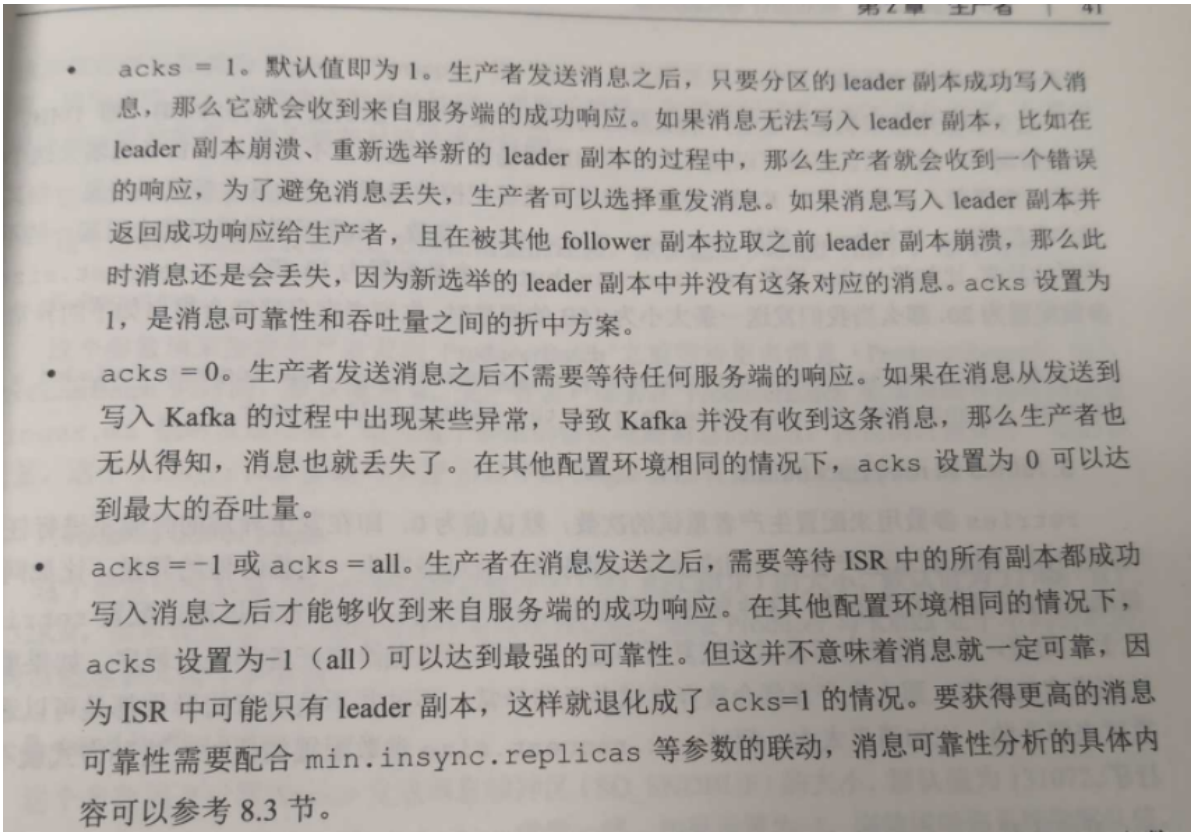
2）同步副本策略ISR

当leader收到消息后，需要同步副本，假如有5个副本，其他四个都很快同步完成，只有一个因故障迟迟不能同步，这会导致ack一直不能发送。为了解决这个问题，leader维护了一个isr，是leader保持同步的follower集合，当出现某一个副本不能够再默认时间内同步成功，就会踢出ISR集合，该时间由**replica.lag.time.max.ms**决定。leader发送故障就会从ISR中从新先去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。

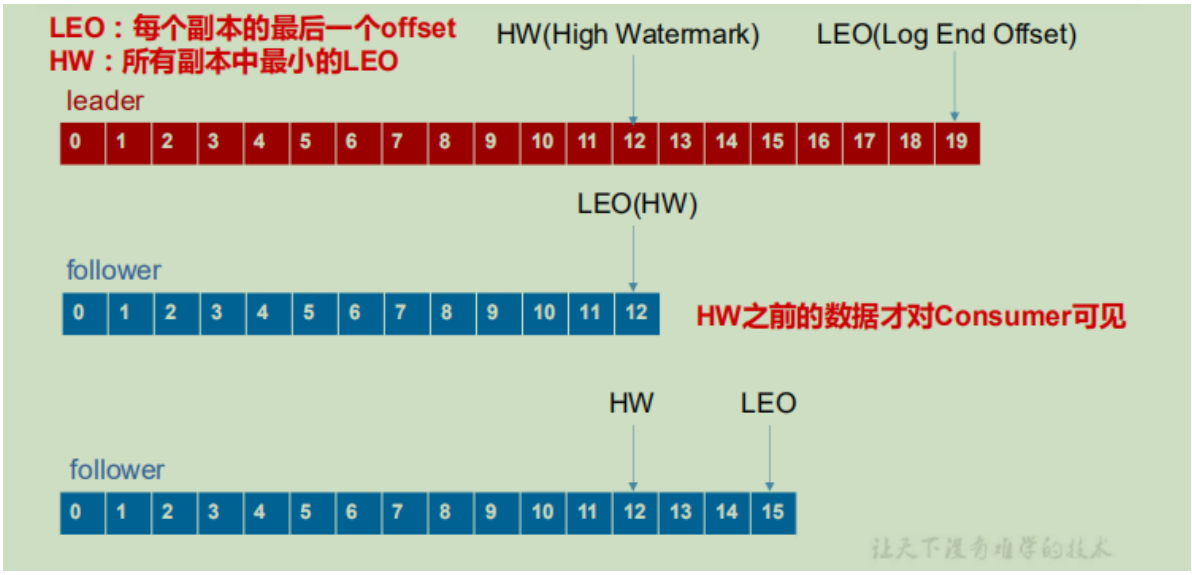
> ACK=0 : producer 不等待 broker 的 ack，这一操作提供了一个最低的延迟，broker 一接收到还没有写入磁盘就已经返回，当 broker 故障时有可能**丢失数据**；
>
> ACK=1： producer 等待 broker 的 ack，partition 的 leader 写入成功后返回 ack。如果leader写入成功并返回ack给生产者，在同步副本的时候挂掉了，这时就会先取新的leader会造成**数据丢失**。
>
> ACK=-1(all):producer 等待 broker 的 ack，partition 的 leader 和 follower 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。默认值即为1。生产者发送消息之后，只要分区的leader副本成功写入消息，那么它就会收到来自服务端的成功响应。如果消息无法写入leader副本，比如在leader副本崩溃、重新选举新的leader副本的过程中，那么生产者就会收到一个错误的响应，为了避免消息丢失，生产者可以选择重发消息。如果消息写入leader副本并返回成功响应给生产者，且在被其他follower副本拉取之前leader副本崩溃，那么此时消息还是会丢失，因为新选举的leader副本中并没有这条对应的消息。acks设置为1，是消息可靠性和吞吐量之间的折中方案。
- acks =0。生产者发送消息之后不需要等待任何服务端的响应。如果在消息从发送到写入Kafka的过程中出现某些异常，导致Kafka并没有收到这条消息，那么生产者也无从得知，消息也就丢失了。在其他配置环境相同的情况下，acks设置为0可以达到最大的吞吐量。
- acks =-1或acks =all。生产者在消息发送之后，需要等待ISR中的所有副本都成功写入消息之后才能够收到来自服务端的成功响应。在其他配置环境相同的情况下，acks设置为-1（all）可以达到最强的可靠性。但这并不意味着消息就一定可靠，因为ISR中可能只有leader副本，这样就退化成了acks=1的情况。要获得更高的消息可靠性需要配合min.insync.replicas等参数的联动，消息可靠性分析的具体内容可以参考8.3节。

这是《深入理解kafka》书中的解释。

4)故障处理细节



> **（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。（消费一致性）
>
> **（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。（存储一致性）

ACK=all 确保producer数据不丢失，但会有重复问题，HW确保数据一致性问题。（消费一致性、存储一致性）

> 为了确保数据不丢失，也不重复。0.11版本之后，引入了幂等性，
>
> 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。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。

幂等性原理解释：https://www.cnblogs.com/smartloli/p/11922639.html

3.3 kafka消费者

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。

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。

3.3.1 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。

1)Range（平铺）

按照主题来分。接入两个主题T1,T2，分别有10个分区。

```
C1：T1（0，1，2，3） T2（0，1，2，3）
C2：T1（4，5，6） T2（4，5，6）
C3：T1（7，8，9） T2（7，8，9）
```

这样会造成 C1多消费2个分区。如果有多个主题，C1的消费压力会越来越大。

2)RoundRobin（轮询）

将所有消费主题打散，均匀的分配给所有消费者，不同消费者最多出现一个分区的消费差别。

3.3.2 消费宕机问题

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。

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**。

查询消费信息 如果使用 --zookeeper 则保存在zookeeper中zh是 组+主题+分区，要是使用--bootstrap-server则保存kafka本地--consumer--offsets

3.4 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。