# 第三版：kafka 18 道

## 如何获取 topic 主题的列表

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

## 生产者和消费者的命令行是什么?

**生产者在主题上发布消息：**

**1、** bin/Kafka-console-producer.sh --broker-list 192.168.43.49:9092 --topic Hello-Kafka

**2、** 注意这里的 IP 是 server.properties 中的 listeners 的配置。接下来每个新行就是输入一条新消息。

**3、** 消费者接受消息：

**4、** bin/Kafka-console-consumer.sh --zookeeper localhost:2181 --topic Hello-Kafka --from-beginning

## consumer 是推还是拉?

**1、** Kafka 最初考虑的问题是，customer 应该从 brokes 拉取消息还是 brokers 将消息推送到 consumer，也就是 pull 还 push。在这方面，Kafka 遵循了一种大部分消息系统共同的传统的设计：producer 将消息推送到 broker，consumer 从 broker 拉取消息。

**2、** 一些消息系统比如 Scribe 和 Apache Flume 采用了 push 模式，将消息推送到下游的 consumer。这样做有好处也有坏处：由 broker 决定消息推送的速率，对于不同消费速率的 consumer 就不太好处理了。消息系统都致力于让 consumer 以最大的速率最快速的消费消息，但不幸的是，push 模式下，当 broker 推送的速率远大于 consumer 消费的速率时，consumer 恐怕就要崩溃了。最终 Kafka 还是选取了传统的 pull 模式。

**3、** Pull 模式的另外一个好处是 consumer 可以自主决定是否批量的从 broker 拉取数据。Push 模式必须在不知道下游 consumer 消费能力和消费策略的情况下决定是立即推送每条消息还是缓存之后批量推送。如果为了避免 consumer 崩溃而采用较低的推送速率，将可能导致一次只推送较少的消息而造成浪费。Pull 模式下，consumer 就可以根据自己的消费能力去决定这些策略。

**4、** Pull 有个缺点是，如果 broker 没有可供消费的消息，将导致 consumer 不断在循环中轮询，直到新消息到 t 达。为了避免这点，Kafka 有个参数可以让 consumer 阻塞知道新消息到达(当然也可以阻塞知道消息的数量达到某个特定的量这样就可以批量发送)。

## 讲讲 Kafka 维护消费状态跟踪的方法

**1、** 大部分消息系统在 broker 端的维护消息被消费的记录：一个消息被分发到 consumer 后 broker 就马上进行标记或者等待 customer 的通知后进行标记。这样也可以在消息在消费后立马就删除以减少空间占用。

**2、** 但是这样会不会有什么问题呢？如果一条消息发送出去之后就立即被标记为消费过的，一旦 consumer 处理消息时失败了（比如程序崩溃）消息就丢失了。为了解决这个问题，很多消息系统提供了另外一个个功能：当消息被发送出去之后仅仅被标记为已发送状态，当接到 consumer 已经消费成功的通知后才标记为已被消费的状态。这虽然解决了消息丢失的问题，但产生了新问题，首先如果

consumer 处理消息成功了但是向 broker 发送响应时失败了，这条消息将被消费两次。第二个问题时，broker 必须维护每条消息的状态，并且每次都要先锁住消息然后更改状态然后释放锁。这样麻烦又来了，且不说要维护大量的状态数据，比如如果消息发送出去但没有收到消费成功的通知，这条消息将一直处于被锁定的状态，

**3、** Kafka 采用了不同的策略。Topic 被分成了若干分区，每个分区在同一时间只被一个 consumer 消费。这意味着每个分区被消费的消息在日志中的位置仅仅是一个简单的整数：offset。这样就很容易标记每个分区消费状态就很容易了，仅仅需要一个整数而已。这样消费状态的跟踪就很简单了。

**4、** 这带来了另外一个好处：consumer 可以把 offset 调成一个较老的值，去重新消费老的消息。这对传统的消息系统来说看起来有些不可思议，但确实是非常有用的，谁规定了一条消息只能被消费一次呢？

## 为什么需要消息系统，MySQL 不能满足需求吗?

**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、** 冗余：

消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的”插入-获取-删除”范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

**3、** 扩展性：

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。

**4、** 灵活性 & 峰值处理能力：

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，而不会因为突发的超负荷的请求而完全崩溃。

**5、** 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，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

**6、** 顺序保证：

在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。（Kafka 保证一个 Partition 内的消息的有序性）

**7、** 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。

**8、** 异步通信：

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。

## Zookeeper 对于 Kafka 的作用是什么?

1、 Zookeeper 是一个开放源码的、高性能的协调服务，它用于 Kafka 的分布式应用。

2、 Zookeeper 主要用于在集群中不同节点之间进行通信

3、 在 Kafka 中，它被用于提交偏移量，因此如果节点在任何情况下都失败了，它都可以从之前提交的偏移量中获取

4、 除此之外，它还执行其他活动，如: leader 检测、分布式同步、配置管理、识别新节点何时离开或连接、集群、节点实时状态等等。

## 数据传输的事务定义有哪三种?

**和 MQTT 的事务定义一样都是 3 种**

1、 最多一次: 消息不会被重复发送，最多被传输一次，但也有可能一次不传输

2、 最少一次: 消息不会被漏发送，最少被传输一次，但也有可能被重复传输.

3、 精确的一次（Exactly once）：不会漏传输也不会重复传输,每个消息都传输被一次而且仅仅被传输一次，这是大家所期望的

## Kafka 判断一个节点是否还活着有那两个条件?

1、 节点必须可以维护和 ZooKeeper 的连接，Zookeeper 通过心跳机制检查每个节点的连接

2、 如果节点是个 follower,他必须能及时的同步 leader 的写操作，延时不能太久

## Kafka 与传统 MQ 消息系统之间有三个关键区别

1、 Kafka 持久化日志，这些日志可以被重复读取和无限期保留

2、 Kafka 是一个分布式系统：它以集群的方式运行，可以灵活伸缩，在内部通过复制数据提升容错能力和高可用性

3、 Kafka 支持实时的流式处理

## 讲一讲 Kafka 的 ack 的三种机制

request.required.acks 有三个值 0 1 -1(all)

0:生产者不会等待 broker 的 ack，这个延迟最低但是存储的保证最弱当 server 挂掉的时候就会丢数据。

1：服务端会等待 ack 值 leader 副本确认接收到消息后发送 ack 但是如果 leader 挂掉后他不确保是否复制完成新 leader 也会导致数据丢失。

-1(all)：服务端会等所有的 follower 的副本受到数据后才会受到 leader 发出的 ack，这样数据不会丢失

## 消费者如何不自动提交偏移量，由应用提交?

将 auto.commit.offset 设为 false，然后在处理一批消息后 commitSync() 或者异步提交 commitAsync()

即：

```
ConsumerRecords<> records = consumer.poll();
for (ConsumerRecord<> record : records){
。。。
tyr{
consumer.commitSync()
}
。。。
}
```

## 消费者故障，出现活锁问题如何解决?

出现“活锁”的情况，是它持续的发送心跳，但是没有处理。为了预防消费者在这种情况下一直持有分区，我们使用 max.poll.interval.ms 活跃检测机制。 在此基础上，如果你调用的 poll 的频率大于最大间隔，则客户端将主动地离开组，以便其他消费者接管该分区。 发生这种情况时，你会看到 offset 提交失败（调用 commitSync（）引发的 CommitFailedException）。这是一种安全机制，保障只有活动成员能够提交 offset。所以要留在组中，你必须持续调用 poll。

**消费者提供两个配置设置来控制 poll 循环：**

**1、** max.poll.interval.ms：增大 poll 的间隔，可以为消费者提供更多的时间去处理返回的消息（调用 poll(long)返回的消息，通常返回的消息都是一批）。缺点是此值越大将会延迟组重新平衡。

**2、** max.poll.records：此设置限制每次调用 poll 返回的消息数，这样可以更容易的预测每次 poll 间隔要处理的最大值。通过调整此值，可以减少 poll 间隔，减少重新平衡分组的

**3、** 对于消息处理时间不可预测地的情况，这些选项是不够的。 处理这种情况的推荐方法是将消息处理移到另一个线程中，让消费者继续调用 poll。 但是必须注意确保已提交的 offset 不超过实际位置。另外，你必须禁用自动提交，并只有在线程完成处理后才为记录手动提交偏移量（取决于你）。 还要注意，你需要 pause 暂停分区，不会从 poll 接收到新消息，让线程处理完之前返回的消息（如果你的处理能力比拉取消息的慢，那创建新线程将导致你机器内存溢出）。

## 如何控制消费的位置

Kafka 使用 seek(TopicPartition, long)指定新的消费位置。用于查找服务器保留的最早和最新的 offset 的特殊的方法也可用（seekToBeginning(Collection) 和 seekToEnd(Collection)）

## Kafka 分布式（不是单机）的情况下，如何保证消息的顺序消费?

**1、** Kafka 分布式的单位是 partition，同一个 partition 用一个 write ahead log 组织，所以可以保证 FIFO 的顺序。不同 partition 之间不能保证顺序。但是绝大多数用户都可以通过 message key 来定义，因为同一个 key 的 message 可以保证只发送到同一个 partition。

**2、** Kafka 中发送 1 条消息的时候，可以指定(topic, partition, key) 3 个参数。partiton 和 key 是可选的。如果你指定了 partition，那就是所有消息发往同 1 个 partition，就是有序的。并且在消费端，Kafka 保证，1 个 partition 只能被 1 个 consumer 消费。或者你指定 key（比如 order id），具有同 1 个 key 的所有消息，会发往同 1 个 partition。

## Kafka 的高可用机制是什么?

这个问题比较系统，回答出 Kafka 的系统特点，leader 和 follower 的关系，消息读写的顺序即可。

https://www.cnblogs.com/qingyunzong/p/9004703.html

https://www.tuicool.com/articles/BNRza2E

https://yq.aliyun.com/articles/64703

## Kafka 如何不消费重复数据？比如扣款，我们不能重复的扣。

**其实还是得结合业务来思考，我这里给几个思路：**

**1、** 比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。

**2、** 比如你是写 Redis，那没问题了，反正每次都是 set，天然幂等性。

**3、** 比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没

有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。

**4、** 比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。