大厂学苑-大数据&人工智能

Kafka

版本：V1.0

More than 80% of all Fortune 100 companies trust, and use Kafka.

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

第1章 Kafka 基础

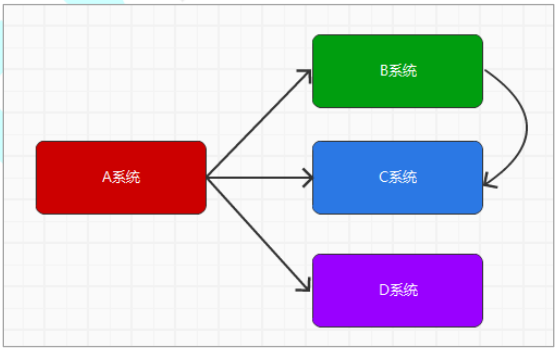
1.1 消息系统

1.1.1 消息队列

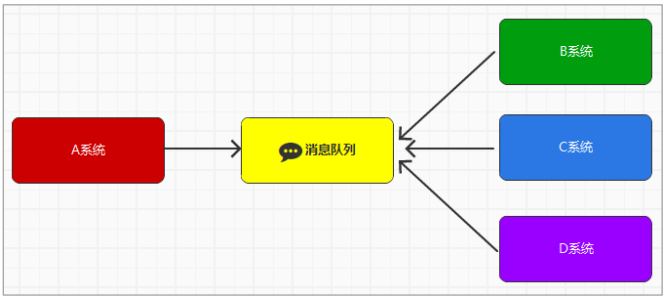
消息队列（Message Queue）是一种帮助开发人员解决系统间异步通信的中间件，常用于解决系统解耦和请求的削峰平谷的问题。它是一种中间件，意味着它是面向研发人员而非终端用户的产品，它的存在不能直接的创造价值，但可以有效的简化研发的开发工作。

一般情况下，我们不会在一个系统中完成所有功能，而是将系统拆分成几个子系统，然后根据业务，进行系统之间的相互调用。这里我们假设有 A，B，C 三个子系统，A 系统的数据会给 B 系统，B 系统的数据会给 C 系统，这种简单的系统间调用，逻辑清晰，架构简单，看起来是完全不需要消息队列的。

但是随着业务的发展，有了新的需求：A 系统的数据给 B 系统后，也要同时提供给 C 系统和 D 系统，那么此时，如果 A 系统的数据量急剧增长，就有可能会导致系统之间的调用延迟非常的高，并且可能会导致各个系统处理能力严重不足或急剧下降。



但是其实，从业务角度来讲，B 系统，C 系统，D 系统获取 A 系统的数据并没有严格意义上的先后顺序，且各个子系统对消息实时性要求并不高，所以如果在系统间增加一个用于进行缓冲或临时存储的消息队列，其实是一个好的选择。但是这里的消息队列需要考虑两种不同的消息模型：点对点（Point to Point，Queue）和发布/订阅（Publish/Subscribe，Topic），这两种模式主要区别或解决的问题就是发送到队列的消息能否重复消费(多订阅)



消息系统带来的好处是：

1) **解耦合：**

允许你独立的扩展或修改消息系统两边的处理过程，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) **顺序保证性**

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。

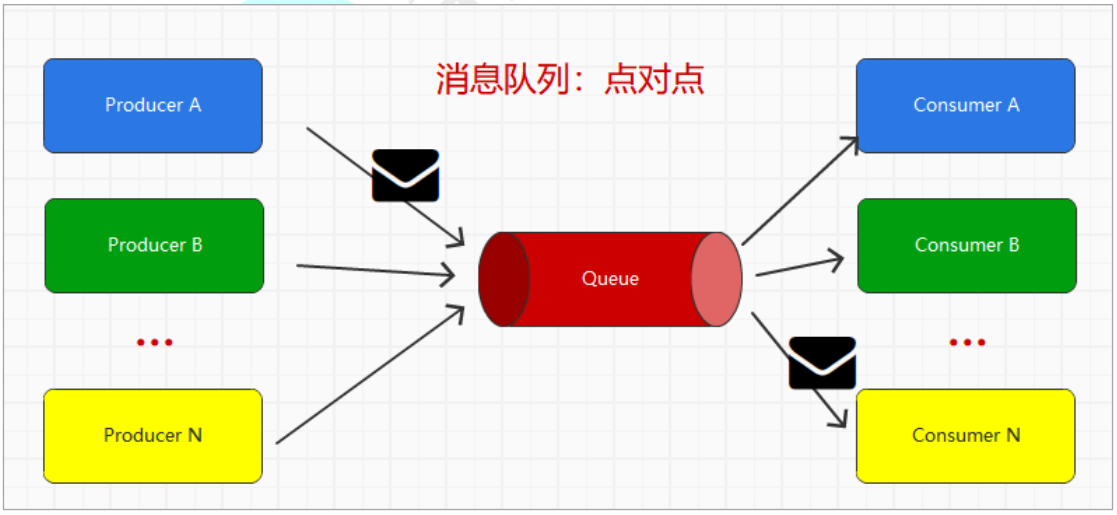
1.1.2 消息模型

Java 消息服务（Java Message Service，JMS）应用程序接口是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。点对点与发布订阅最初是由 JMS 定义的。这两种模式主要区别或解决的问题就是发送到队列的消息能否重复消费(多订阅)

传统企业型消息队列 ActiveMQ 遵循了 JMS 规范，实现了点对点和发布订阅模型，但其他流行的消息队列 RabbitMQ、Kafka 并没有遵循 JMS 规范。

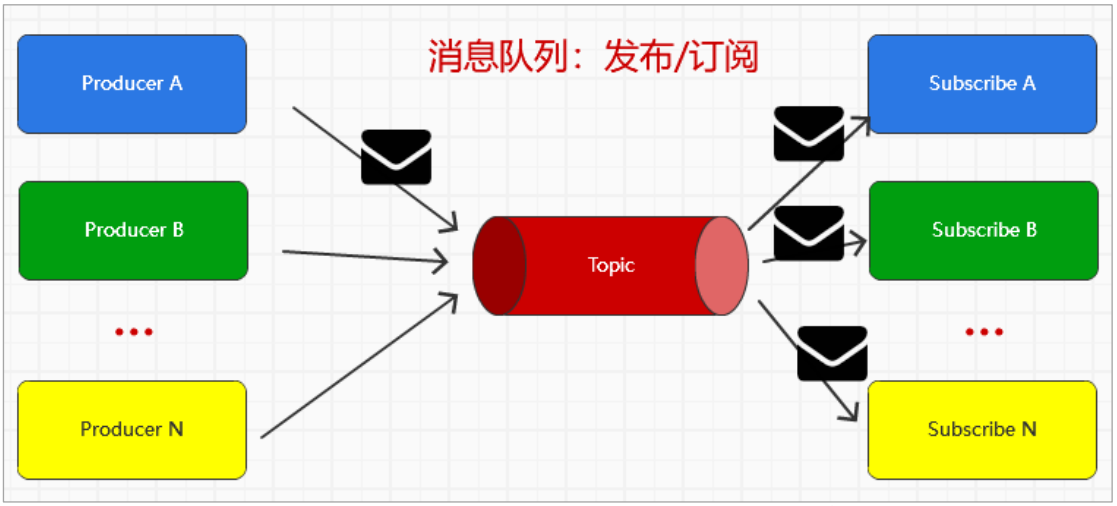
- **点对点**：Queue，不可重复消费

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。

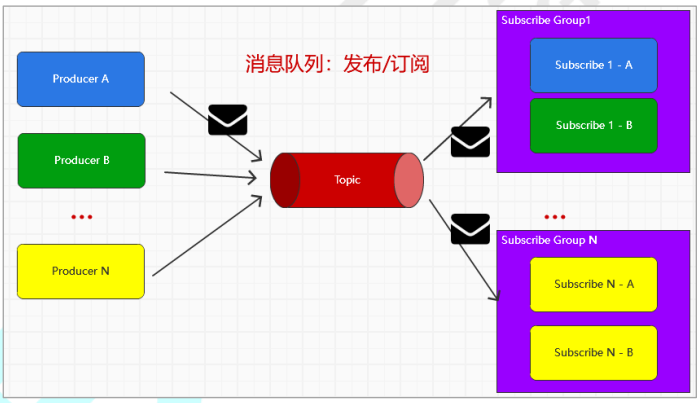


- **发布/订阅**：Topic，可以重复消费

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。



发布订阅模式下，当发布者消息量很大时，显然单个订阅者的处理能力是不足的。实际上现实场景中是多个订阅者节点组成一个订阅组负载均衡消费 Topic 消息即分组订阅，这样订阅者很容易实现消费能力线性扩展。可以看成是一个 Topic 下有多个 Queue，每个 Queue 是点对点的方式，Queue 之间是发布订阅方式。



1.1.3 为什么是 Kafka？

- RabbitMQ

老牌儿消息队列 RabbitMQ 实现了 AQMP 协议，AQMP 协议定义了消息路由规则和方式。生产端通过路由规则发送消息到不同 Queue，消费端根据 Queue 名称消费消息。RabbitMQ 既支持内存队列也支持持久化队列，消费端为推模型，消费状态和订阅关系由服务端负责维护，消息消费完后立即删除，不保留历史消息。

RabbitMQ 的特点：Messaging that just works，“开箱即用的消息队列”。也就是说，RabbitMQ 是一个相当轻量级的消息队列，非常容易部署和使用。

RabbitMQ 对消息堆积的支持并不好，当大量消息积压的时候，会导致 RabbitMQ 的

性能急剧下降。依据硬件配置的不同，它大概每秒钟可以处理几万到十几万条消息。其实，这个性能也足够支撑绝大多数的应用场景了，不过，如果你的应用对消息队列的性能要求非常高，那不要选择 RabbitMQ。

- RocketMQ

RocketMQ 是阿里巴巴在 2012 年开源的消息队列产品，后来捐赠给 Apache 软件基金会，2017 正式毕业，成为 Apache 的顶级项目。阿里内部也是使用 RocketMQ 作为支撑其业务的消息队列，经历过多次“双十一”考验，它的性能、稳定性和可靠性都是值得信赖的。作为优秀的国产消息队列，近年来越来越多的被国内众多大厂使用。RocketMQ 有非常活跃的中文社区，大多数问题你都可以找到中文的答案，也许会成为你选择它的一个原因。另外，RocketMQ 使用 Java 语言开发，它的贡献者大多数都是中国人，源代码相对也比较容易读懂，你很容易对 RocketMQ 进行扩展或者二次开发。RocketMQ 对在线业务的响应时延做了很多的优化，大多数情况下可以做到毫秒级的响应，如果你的应用场景很在意响应时延，那应该选择使用 RocketMQ。RocketMQ 的性能比 RabbitMQ 要高一个数量级，每秒钟大概能处理几十万条消息。RocketMQ 的一个劣势是，作为国产的消息队列，相比国外的比较流行的同类产品，在国际上还没有那么流行，与周边生态系统的集成和兼容程度要略逊一筹。

- Kafka

Kafka 最早是由 LinkedIn 开发，目前也是 Apache 的顶级项目。Kafka 最初的设计目的是用于处理海量的日志。

Kafka 使用 Scala 和 Java 语言开发，设计上大量使用了批量和异步的思想，这种设计使得 Kafka 能做到超高的性能。Kafka 的性能，尤其是异步收发的性能，是三者中最好的，但与 RocketMQ 并没有量级上的差异，大约每秒钟可以处理几十万条消息。。

很多公司常常会在 Kafka 和其他 MessageQueue 之间做选择，这是因为在实时流式架构中，消息用例可被分为两类：队列和流。Kafka 社区于 0.10.0.0 版本正式推出了流处理组件 Kafka Streams，也正是从这个版本开始，Kafka 正式“变身”为分布式的流处理平台，而不仅仅是消息引擎系统了。而 Kafka 在处理高速率实时场景中，表现是异常抢眼的。它全面具备：高吞吐量，低延迟，容错，持久性，可伸缩性，尤其是广为人知的高吞吐量，Kafka 每秒大约可以生产约 25 万消息（50MB），每秒处理 55 万消息（110MB），还有一个巨大的优势就是容错，它具备一个固有功能，可以自行应对集群中的节点故障。所以 Kafka 作为一款明星级产品，能彻底满足海量数据场景下的高吞吐，高并发需求，在短短的几年内，已

经被阿里，腾讯，百度，字节跳动等超一线大厂视为技术核心。

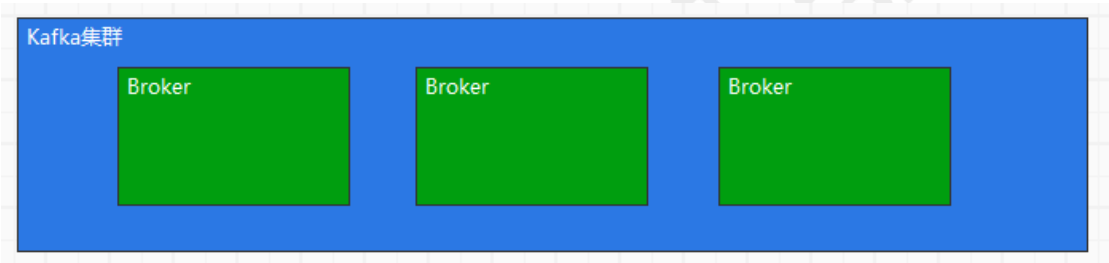
- Pulsar

Pulsar 是一个新兴的开源消息队列产品，最早是由 Yahoo 开发，。与其他消息队列最大的不同是，Pulsar 采用存储和计算分离的设计，未来消息队列的一个发展方向，建议你持续关注这个项目。

1.2 基本概念

1.2.1 Broker

Kafka 的服务器端由被称为 Broker 的服务进程构成，即一个 Kafka 集群由多个 Broker 组成，Broker 负责接收和处理客户端发送过来的请求，以及对消息进行持久化。

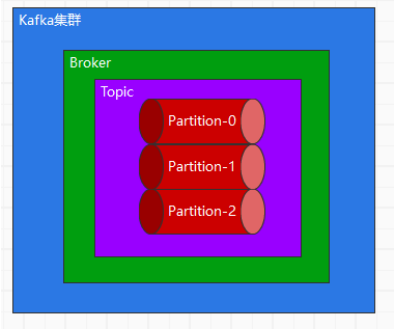


1.2.2 Topic

根据实际的业务场景，将数据按照业务进行分类，每一个分类，称之为 Topic，也叫主题。生产数据和消费数据都是面向主题的。Topic 是一个逻辑上的概念，并没有实际的物理存储。你可以为每个业务、每个应用甚至是每类数据都创建专属的主题。

1.2.3 Partition

kafka 中的分区机制指的是将每个主题划分成多个分区（Partition），每个分区是一组有序的消息日志。生产者生产的每条消息只会被发送到一个分区中，也就是说如果向一个双分区的主题发送一条消息，这条消息要么在分区 0 中，要么在分区 1 中。如你所见，Kafka 的分区编号是从 0 开始的，如果 Topic 有 100 个分区，那么它们的分区号就是从 0 到 99。



1.2.4 Replica

为保证 Kafka 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 仍然能够继续正常工作。Kafka 提供了备份机制（Replication）。备份的思想很简单，就是把相同的数据拷贝到多台机器上，而这些相同的数据拷贝在 Kafka 中被称为副本（Replica）。Kafka 定义了两类副本：领导者副本（Leader Replica）和追随者副本（Follower Replica）。前者对外提供服务，这里的对外指的是与客户端程序进行交互；而后者只是被动地追随领导者副本而已，不能与外界进行交互。为了能更好的保证数据安全，leader 副本和 follower 副本不会放置在同一节点中，所以创建 Topic 时，设置的副本数量不能超出 Broker 节点数量

1.2.5 Producer

向主题发布消息的客户端应用程序称为生产者（Producer），发送时，会将数据封装成一个 ProducerRecord 对象。生产者程序通常持续不断地向一个或多个主题发送消息。

1.2.6 Consumer

订阅主题消息的客户端应用程序就被称为消费者（Consumer），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。

1.2.7 Consumer Group

所谓的消费者组（Consumer Group），指的是多个消费者实例共同组成一个组来消费一组主题。这组主题中的每个分区都只会被组内的一个消费者实例消费，其他消费者实例不能消费它。为什么要引入消费者组呢？主要是为了提升消费者端的吞吐量。多个消费者实例同时消费，加速整个消费端的吞吐量（TPS）。每个消费者在消费消息的过程中必然需要有个字段记录它当前消费到了分区的哪个位置上，这个字段就是消费者位移（Consumer Offset）。消费者组里面的所有消费者实例不仅“瓜分”订阅主题的数据，而且更酷的是它们还能彼此协助。假设组内某个实例挂掉了，Kafka 能够自动检测到，然后把这个 Failed 实例之前负责的分区转移给其他活着的消费者。这个过程就是 Kafka 中大名鼎鼎的“重平衡”（Rebalance）。

1.2.8 Controller

从某种意义上来说，Controller 是 kafka 最核心的组件，一方面，它要为集群中的所有主题分区选取 Leader 副本；另一方面，它还承载着集群的全部元数据信息，并负责将这些元数据信息同步到其他 broker 节点上。

第2章 Kafka 核心

2.1 基础架构

2.1.1 部署环境

生产环境中的 Kafka 集群方案该怎么做。既然是集群，那必然就要有多个 Kafka 节点机器，因为只有单台机器构成的 Kafka 伪集群只能用于日常测试之用，根本无法满足实际的线上生产需求。而真正的线上环境需要仔细地考量各种因素，结合自身的业务需求而制定。下面我就分别从操作系统、磁盘、磁盘容量和带宽等方面来讨论一下。

- 操作系统

常见的操作系统有 3 种：Linux、Windows 和 macOS，考虑操作系统与 Kafka 的适配性，Linux 系统显然要比其他两个特别是 Windows 系统更加适合部署 Kafka。

- 磁盘

Kafka 使用磁盘的方式多是顺序读写操作，一定程度上规避了机械磁盘最大的劣势，即随机读写操作慢。从这一点上来说，使用 SSD 似乎并没有太大的性能优势，还有就是 Kafka 在存储这方面提供了越来越便捷的高可靠性方案，因此在线上环境使用 RAID 似乎变得不是那么重要了。

- 磁盘容量

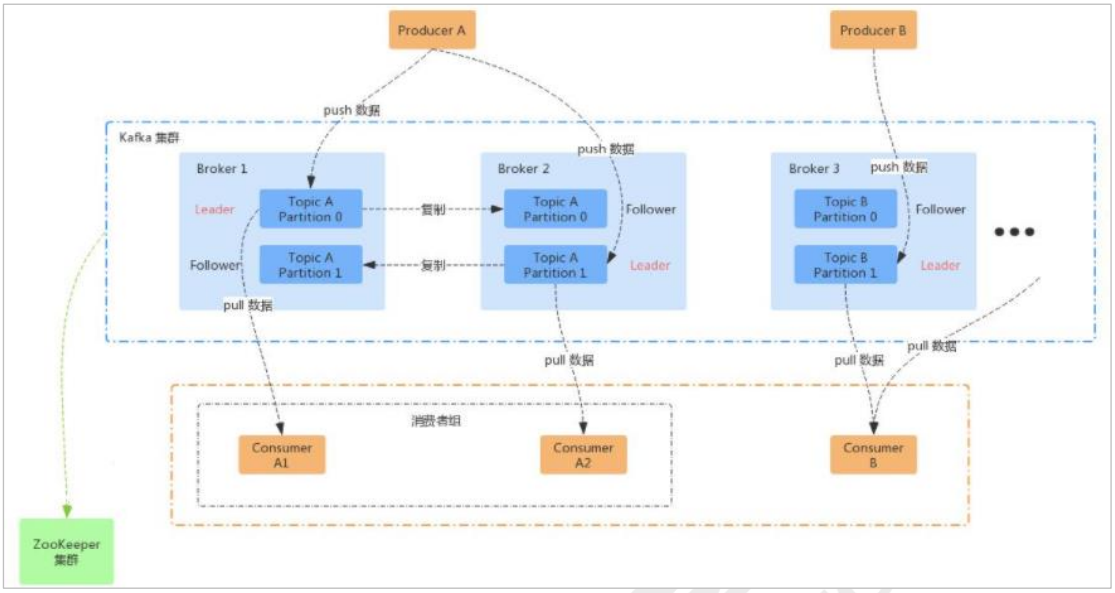
规划磁盘容量时你需要考虑下面这几个元素：

  - 新增消息数
  - 消息留存时间
  - 平均消息大小
  - 备份数
  - 是否启用压缩

- 带宽

对于 Kafka 这种通过网络大量进行数据传输的框架而言，带宽特别容易成为瓶颈。对于 Kafka 这种通过网络大量进行数据传输的框架而言，带宽特别容易成为瓶颈。

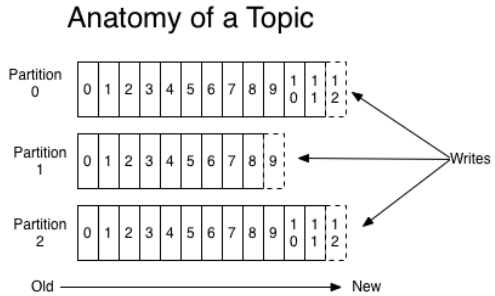
2.1.2 架构



2.2 核心概念

2.2.1 分区策略

我们在使用 Apache Kafka 生产和消费消息的时候，希望能够将数据均匀地分配到所有服务器上。但是在企业中，集群环境会产生大量的数据，每分钟产生的日志量都能以 GB 数，因此如何将这么大的数据量均匀地分配到 Kafka 的各个 Broker 上，就成为一个非常重要的问题。Kafka 有主题（Topic）的概念，它是承载真实数据的逻辑容器，而在主题之下还分为若干个分区，也就是说 Kafka 的消息组织方式实际上是三级结构：主题 - 分区 - 消息。主题下的每条消息只会保存在某一个分区中，而不会在多个分区中被保存多份。官网上的这张图非常清晰地展示了 Kafka 的三级结构，如下所示：



那么 Kafka 这么设计有什么好处呢？为什么使用分区的概念而不是直接使用多个主题呢？

其实分区的作用就是提供负载均衡的能力，或者说对数据进行分区的主要原因，就是为了实现系统的高伸缩性（Scalability）。不同的分区能够被放置到不同节点的机器上，而数据的读写操作也都是针对分区这个粒度而进行的，这样每个节点的机器都能独立地执行各自分区的读写请求处理。并且，我们还可以通过添加新的节点机器来增加整体系统的吞吐量。

下面我们说说 Kafka 生产者的分区策略。所谓分区策略是决定生产者将消息发送到哪个分区的算法。Kafka 为我们提供了默认的分区策略，同时它也支持你自定义分区策略。

- 指明 partition 的情况下，直接将指明的值直接作为 partition 值
- 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；
- 既没有 partition 值又没有 key 值的情况下，kafka 采用 Sticky Partition(黏性分区器)，会随机选择一个分区，并尽可能一直使用该分区，待该分区的 batch 已满或者过了间隔时间，kafka 再随机一个分区进行使用

接下来我们说说 Kafka 消费者的分区策略，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 有三种分配策略：

- RoundRobinAssignor 分配策略
- RangeAssignor（默认）
- StickyAssignor 分配策略

2.2.2 数据可靠性保证

Kafka 只对“已提交”的消息（committed message）做有限度的持久化保证。

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。

副本数据同步策略:

方案	优点	缺点
半数以上完成同步，就发送ack	延迟低	选举新的leader时，容忍n台节点的故障，需要2n+1个副本
全部完成同步，才发送ack	选举新的leader时，容忍n台节点的故障，需要n+1个副本	延迟高

副本数据同步策略 Kafka 选择了第二种方案，原因如下:

- 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。
- 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

2.2.3 Exactly Once 语义

Kafka 是一个高度可扩展的分布式消息系统，在海量数据处理生态中占据着重要的地位。数据处理的一个关键特性是数据的一致性。具体到 Kafka 的领域中，也就是生产者生产的数据和消费者消费的数据之间一对一的一致性。在各种类型的失败普遍存在的分布式系统环境下，保证业务层面一个整体的消息集合被原子的发布和恰好一次处理，是数据一致性在 Kafka 生态系统的实际要求。Kafka 0.11.x 版本，该版本引入了 exactly-once 语义和事务机制。

分布式系统中最难解决的两个问题是：

- 消息顺序保证（Guaranteed order of messages）。
- 消息的精确一次投递（Exactly-once delivery）。

在一个分布式发布订阅消息系统中，组成系统的计算机总会由于各自的故障而不能工作。在 Kafka 中，一个单独的 broker，可能会在生产者发送消息到一个 topic 的时候宕机，或者出现网络故障，从而导致生产者发送消息失败。根据生产者如何处理这样的失败，产生了不同的语义：

- 至少一次语义（**At least once** semantics）：如果生产者收到了 Kafka broker 的确认（acknowledgement，ack），并且生产者的 acks 配置项设置为 all（或-1），这就意味着消息已经被精确一次写入 Kafka topic 了。然而，如果生产者接收 ack 超时或者收到了

错误，它就会认为消息没有写入 Kafka topic 而尝试重新发送消息。如果 broker 恰好在消息已经成功写入 Kafka topic 后，发送 ack 前，出了故障，生产者的重试机制就会导致这条消息被写入 Kafka 两次，从而导致同样的消息会被消费者消费不止一次。每个人都喜欢一个兴高采烈的给予者，但是这种方式会导致重复的工作和错误的结果。

- 至多一次语义（**At most once** semantics）：如果生产者在 ack 超时或者返回错误的时候不重试发送消息，那么消息有可能最终并没有写入 Kafka topic 中，因此也就不会被消费者消费到。但是为了避免重复处理的可能性，我们接受有些消息可能被遗漏处理。
- 精确一次语义（**Exactly once** semantics）： 即使生产者重试发送消息，也只会让消息被发送给消费者一次。精确一次语义是最令人满意的保证，但也是最难理解的。因为它需要消息系统本身和生产消息的应用程序还有消费消息的应用程序一起合作。比如，在成功消费一条消息后，你又把消费的 offset 重置到之前的某个 offset 位置，那么你将收到从那个 offset 到最新的 offset 之间的所有消息。这解释了为什么消息系统和客户端程序必须合作来保证精确一次语义。

2.2.4 offset 的维护

kafka 消费者在会保存其消费的进度，也就是 offset，存储的位置根据选用的 kafka api 不同而不同。

- 消费者如果是根据 javaapi 来消费，也就是【kafka.javaapi.consumer.ConsumerConnector】，我们会配置参数【zookeeper.connect】来消费。这种情况下，消费者的 offset 会更新到 zookeeper 的【consumers/{group}/offsets/{topic}/{partition}】目录下
- 如果是根据 kafka 默认的 api 来消费，即

  【org.apache.kafka.clients.consumer.KafkaConsumer】，我们会配置参数【bootstrap.servers】来消费。而其消费者的 offset 会更新到一个 kafka 自带的 topic【__consumer_offsets】下面，查看当前 group 的消费进度，则要依靠 kafka 自带的工具

  【kafka-consumer-offset-checker】

offset 更新的方式，不区分是用的哪种 api，大致分为两类：

- 自动提交，设置 enable.auto.commit=true，更新的频率根据参数【auto.commit.interval.ms】来定。这种方式也被称为【at most once】，fetch 到消息后就可以更新 offset，无论是否消费成功。

- 手动提交，设置 enable.auto.commit=false，这种方式称为【at least once】。fetch 到消息后，等消费完成再调用方法【consumer.commitSync()】，手动更新 offset；如果消费失败，则 offset 也不会更新，此条消息会被重复消费一次。

2.3 事务

Kafka 从 0.11 版本开始支持了事务机制。Kafka 事务机制支持了跨分区的消息原子写功能。具体来说，Kafka 生产者在同一个事务内提交到多个分区的消息，要么同时成功，要么同时失败。这一保证在生产者运行时出现异常甚至宕机重启之后仍然成立。

此外，同一个事务内的消息将以生产者发送的顺序，唯一地提交到 Kafka 集群上。也就是说，事务机制从某种层面上保证了消息被恰好一次地提交到 Kafka 集群。众所周知，恰好一次送达在分布式系统中是不可能实现的。这个论断有一些微妙的名词重载问题，但大抵没错，所有声称能够做到恰好一次处理的系统都在某个地方依赖了幂等性。

Kafka 的事务机制被广泛用于现实世界中复杂业务需要保证一个业务领域中原子的概念被原子地提交的场景。

2.3.1 生产者事务

事务型 Producer 能够保证消息原子性地写入到多个分区中。这批消息要么全部成功，要么全部失败。另外，事务型 Producer 也不惧进程重启，Producer 重启后，Kafka 依然保证它们发送消息的精确一次处理。

设置事务型 Producer 需要满足两要求：

- 和幂等性 Producer 一样，开启 enable.idempotence = true;
- 设置 Producer 端参数 transaction.id ,最好为其设置一个有意义的名字。

在 Producer 代码中也需要做调整

```
producer.initTransactions();
try {
          producer.beginTransaction();
          producer.send(record1);
          producer.send(record2);
          producer.commitTransaction();
} catch (KafkaException e) {
          producer.abortTransaction();
}
```

和普通的 Producer 代码相比，事务型 Producer 的显著特点是调用了一些事务 API 如：initTransaction（事务初始化）、beginTransaction（事务开始）、commitTranscation（事务提

交）、abortTransaction（事务终止）。

这段代码中能够保证 record 1 和 record 2 被当做一个事务统一提交到 Kafka，要么全部提交成功，要么全部提交失败。

事务型 Producer 是基于两阶段提交，实际上写入失败时，Kafka 也会把消息写入到日志中，也就是说 Consumer 有可能会看到，写入失败数据，所以读取事务消息 Consumer 也需要做一些变更，即 设置 isoIation.level 参数，当前这个参数有两个取值：

- read_uncommitted:这是默认值，表明 Consumer 能够读取到 Kafka 写入的任何消息，在用事务型 Producer 时，对应的 consumer 就不要使用这个值。
- read_committed:表明 Consumer 只会读取事务型 Producer 成功提交事务写入的消息。当然它也能看到非事务型 Producer 写入的消息。

2.3.2 消费者事务

生产者应该开启事务

```
//开启事务
properties.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,"transaction-id"+
UUID.randomUUID());
```

消费者应该设置提交事务的隔离级别

```
//设置事务的隔离级别 为已提交
properties.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG,"read_committed");
```

生产者提交数据失败 控制台报错 消费者隔离级别是 read_committed 只能读取成功提交的数据。生产者提交数据失败 控制台报错 消费者隔离级别是 read_uncommitted 可以读取未提交的数据。

2.4 监控

消费 Kafka 集群中的消息时，数据的变动是我们所关心的，当业务并不复杂的前提下，我们可以使用 Kafka 提供的命令工具，配合 Zookeeper 客户端工具，可以很方便的完成我们的工作。随着业务的复杂化，Group 和 Topic 的增加，此时我们使用 Kafka 提供的命令工具，已预感到力不从心，这时候 Kafka 的监控系统此刻便尤为显得重要，我们需要观察消费应用的详情。 监控系统业界有很多杰出的开源监控系统。我们在早期，有使用 KafkaMonitor 和 Kafka Manager 等，不过随着业务的快速发展，以及互联网公司特有的一些需求，现有的开源的监控系统在性能、扩展性、和 DEVS 的使用效率方面，已经无法满足了。 因此，我们在过去的时间里，从互联网公司的一些需求出发，从各位 DEVS 的使

用经验和反馈出发，结合业界的一些开源的 Kafka 消息监控，用监控的一些思考出发，设计开发了现在 Kafka 集群消息监控系统：Kafka Eagle。

Kafka Eagle 用于监控 Kafka 集群中 Topic 被消费的情况。包含 Lag 的产生，Offset 的变动，Partition 的分布，Owner ，Topic 被创建的时间和修改的时间等信息。下载地址如下所示

下载地址：http://download.smartloli.org

2.4.1 上传文件

将压缩文件 kafka-eagle-bin-1.2.3.tar.gz 复制到 Linux

```
tar -zxvf kafka-eagle-bin-2.0.5.tar.gz -C /opt/module/
mv kafka-eagle-bin-2.0.5/  kafka-eagle
cd /opt/module/kafka-eagle
tar -zxvf kafka-eagle-web-2.0.5-bin.tar.gz -C ./
```

2.4.2 修改配置文件

修改 conf/system-config.properties 配置文件

```
cd /opt/module/kafka-eagle/kafka-eagle-web-2.0.5/conf/

# 修改配置文件
# multi zookeeper&kafka cluster list
kafka.eagle.zk.cluster.alias=cluster1
cluster1.zk.list=zk 机器:2181/kafka
kafka.zk.limit.size=25
kafka.eagel.webui.port=8048
kafka.eagle.url=jdbc:sqlite://opt/module/kafka-eagle/db/ke.db
```

2.4.3 配置环境变量

修改/etc/profile.d/my_env.sh 配置文件

```
cd /opt/module/kafka-eagle/kafka-eagle-web-2.0.5/conf/

#增加环境变量
export KE_HOME=/opt/module/kafka-eagle/kafka-eagle-web-2.0.5/

cd /etc/profile.d
Source my_env.sh
```

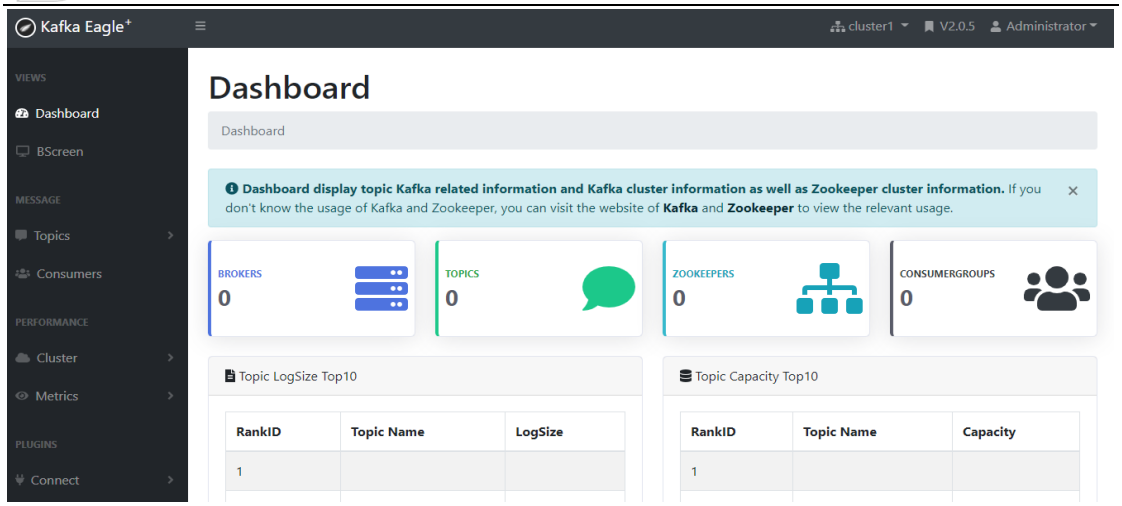
2.4.4 启动服务

```
bin/ke.sh start
```

2.4.5 访问 web 服务

访问网址：http://虚拟机地址:8048/

账号：admin，密码：123456



第3章　Kafka 扩展

3.1　kafka 为什么这么快？

3.1.1 批处理

kafka，一次发送多条消息，微批处理。

```
生产者发送消息，需要 2 次 rpc：
        发送消息；
        broker 返回 ACK 信号，表示已经接收消息；
消费者消费消息，3 次 rpc：
        消费者请求接收消息；
        broker 返回消息；
        消费者返回 ACK 信号，表示已经消费
```

3.1.2 客户端优化

新版客户端摒弃单线程，采用双线程模式——主线程+Sender 线程。主线程负责将消息置入客户端缓存(缓存会将多个消息聚合为 1 个批次)；Sender 线程将缓存中聚合好的批次消息发送到 Broker。

3.1.3 设计优良的日志消息格式

新版本(从 kafka 0.11.0 版本开始)的日志消息格式，引用了变长字段 Varints 和 ZigZag 编码，有效降低了附加字段占用的空间，降低了网络传输、日志存盘占用开销。

3.1.4 消息压缩

Kafka 支持多种消息压缩方式（gzip、snappy、lz4）。对消息进行压缩可以极大地减少网络传输 量、降低网络 I/O，从而提高整体的性能。消息压缩是一种使用时间换空间的优化方式，如果对 时延有一定的要求，则不推荐对消息进行压缩。

3.1.5 分区

Kafka 对消息进行分区，提高了数据生产与消费的并行度，有效的提升了数据的吞吐量。

3.1.6 索引

kafka 为每个日志分段文件提供了 2 个索引文件(偏移量索引文件.index、时间戳索引文件.timeindex)，提高了消息的查询效率

3.1.7 顺序写盘

Kafka 在设计时采用了文件追加的方式来写入消息，即只能在日志文件的尾部追加新的消息，并且也不允许修改已写入的消息，这种方式属于典型的顺序写盘的操作，而操作系统可以针对线性读写做深层次的优化，比如预读(read-ahead，提前将一个比较大的磁盘块读入内存) 和后写(write-behind，将很多小的逻辑写操作合并起来组成一个大的物理写操作)技术，所以就算 Kafka 使用磁盘作为存储介质，它所能承载的吞吐量也不容小觑。

3.1.8 页缓存

Kafka 中大量使用了页缓存，这是 Kafka 实现高吞吐的重要因素之一。页缓存是操作系统实现的一种主要的磁盘缓存，采用页缓存的主要优点：

- 减少对磁盘 I/O 的操作(具体来说，就是把磁盘中的数据缓存到内存中，把对磁盘的访问变为对内存的访问)
- 维护页缓存和文件之间的一致性交由操作系统来负责，比进程内维护更加安全有效

3.1.9 零拷贝

Kafka 使用了 Zero Copy 技术提升了消费的效率。前面所说的 Kafka 将消息先写入页缓存，如果消费者在读取消息的时候如果在页缓存中可以命中，那么可以直接从页缓存中读取，这样又节省了一次从磁盘到页缓存的 copy 开销。

Zero-Copy 技术省去了将操作系统的 read buffer 拷贝到程序的 buffer，以及从程序 buffer 拷贝到 socket buffer 的步骤，直接将 read buffer 拷贝到 socket buffer. Java NIO 中的 FileChannal.transferTo()方法就是这样的实现，这个实现是依赖于操作系统底层的 sendFile()实现的。

3.2 kafka 优化

所谓的优化，其实就是在了解软件整体架构后，对其中的组件，参数进行调整所进行的配置

调整。

3.2.1 partition 数量配置

partition 数量由 topic 的并发决定，并发少则 1 个分区就可以，并发越高，分区数越多，可以提高吞吐量。

3.2.2 日志保留策略设置

当 kafka broker 的被写入海量消息后，会生成很多数据文件，占用大量磁盘空间，kafka 默认是保留 7 天，建议根据磁盘情况配置，避免磁盘撑爆。

```
# server.properties
# Kafka 的消息保存时间（单位：小时）
log.retention.hours=72
```

段文件配置 1GB，有利于快速回收磁盘空间，重启 kafka 加载也会加快(如果文件过小，则文件数量比较多，kafka 启动时是单线程扫描目录(log.dir)下所有数据文件)

```
# Kafka 的消息保留字节数
# 段文件配置 1GB，有利于快速回收磁盘空间，重启 kafka 加载也会加快(如果文件过小，则文件数量比较多，kafka 启动时是单线程扫描目录(log.dir)下所有数据文件)
log.retention.bytes=1024
```

3.2.3 文件刷盘策略

为了大幅度提高 producer 写入吞吐量，需要定期批量写文件。建议配置：

```
# 每当 producer 写入 10000 条消息时，刷数据到磁盘
log.flush.interval.messages=10000
# 每间隔 1 秒钟时间，刷数据到磁盘
log.flush.interval.ms=1000
```

3.2.4 网络和 io 操作线程配置优化

一般 num.network.threads 主要处理网络 io，读写缓冲区数据，基本没有 io 等待，配置线程数量为 cpu 核数加 1

```
# broker 处理消息的最大线程数
num.network.threads=xxx
```

num.io.threads 主要进行磁盘 io 操作，高峰期可能有些 io 等待，因此配置需要大些。配置线程数量为 cpu 核数 2 倍，最大不超过 3 倍。

```
# broker 处理磁盘 IO 的线程数
num.io.threads=xxx
```

加入队列的最大请求数,超过该值，network thread 阻塞

```
queued.max.requests=5000
```

server 使用的 send buffer 大小。

```
socket.send.buffer.bytes=1024000
```

server 使用的 recive buffer 大小。

```
socket.receive.buffer.bytes=1024000
```

3.2.5 异步提交（kafka.javaapi.producer）

```
request.required.acks=0
producer.type=async
##在异步模式下，一个 batch 发送的消息数量。producer 会等待直到要发送的消息数量达到这个值，之后才会发送。但如果消息数量不够，达到 queue.buffer.max.ms 时也会直接发送。
batch.num.messages=100
##默认值：200，当使用异步模式时，缓冲数据的最大时间。例如设为 100 的话，会每隔 100 毫秒把所有的消息批量发送。这会提高吞吐量，但是会增加消息的到达延时
queue.buffering.max.ms=100
##默认值：5000，在异步模式下，producer 端允许 buffer 的最大消息数量，如果 producer 无法尽快将消息发送给 broker，从而导致消息在 producer 端大量沉积，如果消息的条数达到此配置值，将会导致 producer 端阻塞或者消息被抛弃。
queue.buffering.max.messages=1000 ##发送队列缓冲长度
##默认值：10000，当消息在 producer 端沉积的条数达到 queue.buffering.max.meesages 时，阻塞一定时间后，队列仍然没有 enqueue(producer 仍然没有发送出任何消息)。此时 producer 可以继续阻塞或者将消息抛弃，此 timeout 值用于控制阻塞的时间，如果值为-1（默认值）则 无阻塞超时限制，消息不会被抛弃；如果值为 0 则立即清空队列，消息被抛弃。
queue.enqueue.timeout.ms=100
compression.codec=gzip
```

3.2.6 producer 版本

使用新 producer 发送少量消息时丢失

- 使用 producer 时必须调用 producer.close()，且在发送后 Thread.sleep 适当时间，则不会丢失数据。否则会造成资源泄露，导致数据丢失。
- 当使用多个 producer 进行发送时（使用 apache 线程池），当同时有多个 producer 并发发送时，依然会造成数据丢失。sleep 后有好转，但仍然丢失。
- 使用老 producer，且 compression.codec（压缩编码 压缩格式）不为 snappy 时，不会造成数据丢失。使用线程池也不会丢失。

3.2.7 compact 策略

通过压缩算法对日志文件进行压缩，而是对重复的日志进行清理来达到目的。在日志清理过程中，会清理重复的 key，最后只会保留最后一条 key，可以理解为 map 的 put 方法。在清理完后，一些 segment 的文件大小就会变小，这时候，kafka 会将那些小的文件再合并成一个大的 segment 文件。

3.2.8 开启压缩

在 kafka/config/producer.properties 中配置

```
compression.type=none
```

3.2.9 内存

在 kafka/bin/kafka-server-start.sh 中配置

```
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
```

```
    export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
fi
```

默认 1G，生产中的经验值为 4 到 6 G。最高 6G，超过 6G 效果不明显，这时建议加服务器

3.2.10 消息大小

kafka 对于消息体的大小默认为单条最大值是 1M 但是在我们应用场景中，常常会出现一条消息大于 1M，如果不对 kafka 进行配置。则会出现生产者无法将消息推送到 kafka 或消费者无法去消费 kafka 里面的数据

```
#producer.properties:
#也可以在创建 topic 时动态设置
max.request.size=5242880（5M）
#server.properties
message.max.bytes=6291456（6M）
#consumer.properties:
fetch.max.bytes=7340032（7M）


#max.request.size < message.max.bytes < fetch.max.bytes
```

3.3 kafka 面试题

一线互联网大厂，面试所考到的知识可不止 Kafka，当然多拥有一个技术点，对自己的面试也是一件好事,而 kafka 作为一个流行，成熟的消息队列，在大厂中被广泛使用，所以不同大厂的面试问题也不会有太大的差异，面试官是否满意也就区别于你回答的内容上。

3.3.1 什么是 Kafka？为什么 Kafka 技术很重要？

3.3.2 描述 Kafka 中的各个组件？

3.3.3 解释偏移量的作用？

3.3.4 什么是消费者组？

3.3.5 ZooKeeper 的作用是什么？

3.3.6 是否可以在没有 ZooKeeper 的情况下使用 Kafka？

3.3.7 领导者和跟随者是什么意思？

3.3.8 副本服务器和 ISR 扮演什么角色？

3.3.9 为什么复制在 Kafka 中至关重要？

3.3.10 如果副本长时间不在 ISR 中，则表示什么？

3.3.11 连接器 API 的作用是什么？

3.3.12 consumer 是推还是拉？

3.3.13 讲一下主从同步？

3.3.14 在生产者中，何时发生 QueueFullException？

3.3.15 为什么需要消息系统，mysql 不能满足需求吗？

3.3.16 数据传输的事务定义有哪三种？

3.3.17 讲一讲 kafka 的 ack 的三种机制？

3.3.18 消费者故障，出现活锁问题如何解决？

3.3.19 kafka 分布式（不是单机）的情况下，如何保证消息的顺序消费？

3.3.20 kafka 如何不消费重复数据？比如扣款，我们不能重复的扣。