**Kafka简介**

作者：叶翠

Kafka基础名词解析 1

Kafka进阶名词解析 3

Kafka机制解析 7

Kafka场景分析 12

Kafka源码 15

## Kafka基础名词解析

**什么是Kafka，Broker、Topic、 Partition、Producer、Consumer又是什么？**

Kafka 是一种高性能、可扩展的分布式消息队列系统，常用于处理海量数据和实时数据流。

* **Broker（代理）** ：Kafka 集群由一个或多个独立的服务器节点组成，每个节点被称为 Broker。每个 Broker 负责消息的**存储、接收和转发(接化发)** 。它们一起形成了一个分布式消息传递系统。
* **Topic（主题）** ：主题是 Kafka 中消息的类别或数据流的名称。消息通过主题进行分类和发布。你可以将主题视为具有唯一标识符的消息队列。例如，你可以有一个主题用于接收日志消息，另一个主题用于接收用户活动消息。
* **Partition（分区）** ：每个主题可以分成一个或多个分区。分区是主题的物理划分，用于实现数据的并行处理和分布式存储。每个分区都是有序的、不可变的消息序列。分区在磁盘上以文件形式存储，并由 Broker 负责管理。
* **Producer（生产者）** ：生产者是生成消息并将其发布到 Kafka 主题的应用程序或系统。生产者负责将消息发送到特定的主题，并且可以选择将消息发送到特定的分区。生产者将消息发送到 Broker，然后由 Broker 将消息持久化并根据配置的复制策略复制到其他 Broker。
* **Consumer（消费者）** ：消费者是从 Kafka 主题中读取消息的应用程序或系统。消费者订阅一个或多个主题，**并从主题的一个或多个分区中读取消息**。每个消费者都有一个独立的消费者组标识符，以便多个消费者可以以并行的方式消费主题中的消息。

**简单总结一下**就是，Kafka是一个常用的高性能分布式消息队列，主要有代理、主题、分区、生产者和消费者这些概念。代理也就是Kafka集群中的每个服务器节点的代称，负责消息的存储、接收和转发。Kafka是基于发布订阅的消息系统，主题就是生产者和消费者建立联系的桥梁。每个主题可以划分至少一个分区，每个分区都是一组有序的消息，同时分区数也代表最大并行度。生产者是生成消息并将其发布到 Kafka 主题的应用程序或系统，消费者则是将消息从Kafka主题中读取的应用程序或系统。

补充一些小众知识：

1. **partition 的数据文件（offset，MessageSize，data）** 。partition中的每条 Message 包含了以下三个属性：offset，MessageSize，data，其中 offset 表示 Message 在这个 partition 中的偏移量，offset 不是该 Message 在 partition 数据文件中的实际存储位置，而是逻辑上一个值，它唯一确定了 partition 中的一条 Message，可以认为 offset 是 partition 中 Message 的 id；MessageSize 表示消息内容 data 的大小；data 为 Message 的具体内容。
2. **数据文件分段 segment（顺序读写、分段命名、二分查找）** 。partition 物理上由多个 segment 文件组成，每个 segment 大小相等，顺序读写。每个 segment 数据文件以该段中最小的 offset 命名，文件扩展名为.log。这样在**查找指定 offset 的 Message 的时候，用二分查找就可以定位到该 Message 在哪个 segment 数据文件中**。
3. **数据文件索引（分段索引、稀疏存储）** 。Kafka 为每个分段后的数据文件建立了索引文件，文件名与数据文件的名字是一样的，只是文件扩展名为.index。index 文件中并没有为数据文件中的每条 Message 建立索引，而是采用了稀疏存储的方式，每隔一定字节的数据建立一条索引。这样避免了索引文件占用过多的空间，从而可以将索引文件保留在内存中。

## Kafka进阶名词解析

### 控制器（Controller）

Kafka 启动是会往 Zookeeper 中注册当前Broker 信息. 谁先注册谁就是 Controller. 读取注册上来的从节点的数据(通过监听机制), **生成集群的元数据信息**, 之后把这些信息都分发给其他的服务器, 让其他服务器能感知到集群中其它成员的存在

### 协调者(Coordinator)

Kafka的Coordinator是负责协调和管理消费者组和生产者事务的关键组件。它**确保消费者组能够正确分配和消费分区，并管理生产者事务的正确提交和恢复**。

1. **消费者组协调器（Group Coordinator）** ：每个消费者组都有一个消费者组协调器。它是一个Kafka broker，负责协调和管理消费者组的各种活动。消费者组协调器执行以下任务：

* **注册新的消费者**：当一个消费者加入消费者组时，它会将自己注册到消费者组协调器，并接收该消费者组的分配策略。
* **分配分区**：当消费者组中的消费者数量发生变化（如有新的消费者加入或旧的消费者离开），消费者组协调器负责重新分配分区给消费者。它使用分区分配策略（Partition Assignment Strategy）来决定如何将分区分配给消费者，以实现负载均衡和容错性。
* **管理消费者组的偏移量（offset）** ：消费者组协调器还负责跟踪和管理消费者组在每个分区上的消费偏移量。它会将消费者组的偏移量提交到Kafka中，以确保消费者组可以在断开连接或重新平衡后正确地从上次的偏移量继续消费。

1. **生产者事务协调器（Transaction Coordinator）** ：Kafka还有一个生产者事务协调器，它负责管理Kafka事务的协调工作。生产者事务协调器执行以下任务：

* **初始化事务**：当生产者开始一个事务时，它会与生产者事务协调器进行交互，以获取一个事务ID和对应的Leader副本。
* **提交事务**：生产者在事务完成时将事务的提交请求发送给生产者事务协调器。协调器确保事务中的所有消息都成功写入，并将事务的提交状态写入Kafka日志。
* **事务恢复**：如果生产者在事务期间失败或崩溃，生产者事务协调器负责协调恢复过程，以确保事务的完整性。

### 消费者组(Consumer Group)

消费者组（Consumer Group）用于组织和管理消息的消费者。官方的介绍，消费者组是Kafka提供的可扩展且具有容错性的消费者机制。消费者组中包含多个消费者实例，每个实例都可以并行地消费消息。消息队列系统将消息分发给组中的消费者实例，以便实现高吞吐量和负载均衡。需要注意的是超过分区数的同一消费者组的消费者无法消费消息，**所以一般设置消费者数等于分区数，当然也可以将多出来的消费者当作故障转移的备用**。

**值得一提的是**，Kafka用消费者组这个设计同时实现了两个消息队列模型。如果所有实例都属于同一个Group，是队列模型。如果所有实例分别属于不同的Group，那么它实现的就是发布/订阅模型（一对多）。

### 再平衡（Rebalancing）

再平衡（Rebalancing）是指当消费者组中的消费者实例发生变化时，消息队列系统会重新分配分发给各个消费者实例的消息的过程。**新消费者实例加入，或者老实例因为故障退出或者心跳超时离开，都会导致再平衡的发生**。再平衡过程中需要一定的时间，并可能导致一些消息在重新分配期间无法处理。

Rebalance过程分为两步：**Join和Sync**

1. Join阶段中，所有消费者实例都向Broker的消费协调者(Group Coordinator)发送加入请求。获取所有成功请求后，协调者随机选择一个消费者实例成为leader，并将组成员信息和Topic信息发给leader，由Consumer Leader负责消费分配方案的制定
2. Sync阶段中Consumer Leader开始分配消费方案, 即哪个 Consumer 负责消费哪些 Topic 的哪些Partition. 一旦完成分配, Leader 会将这个方案封装进 SyncGroup 请求中发给 Coordinator,非 Leader 也会发 SyncGroup 请求, 只是内容为空. Coordinator 接收到分配方案之后会把方案塞进SyncGroup的Response中发给各个Consumer. 这样组内的所有成员就都知道自己应该消费哪些分区

### 消息（Message）

一个 Kafka 的 Message 由一个固定长度的 header 和一个可变长度的消息体 body组成。

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，比如：

* Key（可选）：消息的键，用于分区和消息的顺序性。如果提供了键，Kafka将根据键的哈希值将消息路由到特定的分区，确保具有相同键的消息被写入和读取到同一个分区中。
* Value：实际的消息内容，通常是字节数组或字符串。这是需要传输和处理的数据。
* Offset：消息在分区中的唯一标识符，表示消息在分区中的位置。消费者可以使用偏移量来跟踪其消费进度，确保不会漏掉任何消息。
* Partition（可选）：消息所属的分区编号。如果未指定分区，Kafka将根据生产者配置的分区策略自动分配分区。
* Timestamp（可选）：消息的时间戳，表示消息被生产的时间。时间戳可以是消息实际被生产的时间，也可以是在生产者发送消息时显式设置的自定义时间。
* Headers（可选）：一组键值对，用于存储与消息相关的元数据。头部可以包含各种自定义信息，如消息的来源、类型、版本等。

### TOPIC的TOPIC（\_\_consumer\_offsets）

在Kafka中，\_\_consumer\_offsets是一个内部主题（internal topic），用于存储消费者组的偏移量（offset）信息。它是用来跟踪消费者在特定主题的分区中消费的进度。\_\_consumer\_offsets主题中的每条消息都包含以下信息：

* 消费者组ID（consumer group ID）
* 主题名称（topic name）
* 分区ID（partition ID）
* 消费者组内消费者的偏移量（offset）
* 消息提交的时间戳（timestamp）

**通过维护这个内部主题，Kafka能跟踪和管理消费者组在分区中消费的进度，确保消费者能够正确地处理消息并实现故障恢复**。当消费者启动时，它会从该主题中读取偏移量信息，确定从哪个偏移量开始消费消息。随着消息的处理和消费者提交偏移量，这些信息也会相应地更新。同时，Kafka的消费者协调器（consumer coordinator）负责更新和维护\_\_consumer\_offsets主题。它会处理消费者加入和离开消费者组的请求，以及消费者提交和获取偏移量的操作。

### 数据文件（Data files）和日志文件（Log files）

数据文件（Data Files）：

在Kafka中，数据文件通常指的是Kafka服务器上**存储实际消息数据**的文件。Kafka使用分段存储的方式来管理消息，每个主题被分成多个分区（partitions），而每个分区又被细分为一系列的数据文件，这些数据文件用于存储消息的内容。当消息被生产者发送到Kafka集群后，它们会被追加到适当分区的数据文件中。消费者可以从这些数据文件中读取消息。

日志文件（Log Files）：

在Kafka中，日志文件通常指的是Kafka Broker上**用于持久化消息**的文件。Kafka使用日志来保持消息的持久性，并使用基于追加的日志存储模型。每个主题的每个分区都有一个相应的日志文件，用于按顺序存储消息。日志文件中的消息是以追加的方式进行写入和读取的。一旦日志文件达到一定的大小限制或时间限制，Kafka会将其关闭并创建一个新的日志文件，以便继续持久化消息。

### 日志索引

Kafka 能支撑 TB 级别数据, 在日志级别有两个原因: **基于顺序追加写日志 + 稀疏哈希索引**(和数据文件类似，都用了索引这个概念来优化)。**Kafka的稀疏哈希索引是一种以固定间隔记录关键消息偏移量的数据结构，用于加速消息的定位和检索，同时节省内存空间。**

当消息被写入Kafka的日志文件时，它们按照顺序追加到分区的日志末尾。每个分区都有一个对应的稀疏哈希索引，其中记录了一些重要的消息偏移量。这些索引并不会记录每条消息的偏移量，而是以固定的间隔（通常是一段连续的消息）记录一些关键消息的偏移量。使用稀疏哈希索引的好处在于，它能够显著减少索引的大小，从而节省了内存空间。

当消费者需要读取特定偏移量的消息时，它可以首先通过稀疏哈希索引找到最接近目标偏移量的记录，然后再线性地扫描日志文件，直到找到目标消息。这种方法虽然相比完整的索引查找要慢一些，但是稀疏哈希索引的存在使得整体的查找速度仍然非常高效。

## Kafka机制解析

### 详细说明一下Kafka的多副本(Replica)机制

Kafka的副本(Replica)提供了数据冗余和故障恢复的机制，确保Kafka节点故障时不会丢失数据，并支持高吞吐量的数据读写操作。

每个分区可以拥有多个副本，多个副本分布在不同的Broker节点上。一个分区的第一个副本称为领导者副本（Leader Replica），其他副本称为追随者副本（Follower Replica）。

**领导者副本**：每个分区的领导者副本负责处理该分区的所有读写请求。生产者将消息发送到领导者副本，消费者从领导者副本读取消息。领导者副本还负责将数据复制到追随者副本。

**追随者副本**：追随者副本是领导者副本的复制品。它们会从领导者副本同步数据。追随者副本不处理客户端的读写请求，仅用于提供冗余和故障转移。 （**值得一提的是**，很多分布式系统在设计类似副本概念的时候，会至少提供读的功能以分摊主节点压力，相比之下Kafka的副本机制更加严格）

**数据复制**：领导者副本将消息写入其本地日志（Log），然后通过复制机制将消息复制到追随者副本的日志中。复制可以使用两种模式：同步复制和异步复制。

* 同步复制：领导者副本等待所有追随者副本确认已成功复制消息后，才认为消息已提交。这种模式提供最强的数据保证，但会对写入延迟产生一定影响。
* 异步复制：领导者副本在将消息写入本地日志后立即返回成功响应，而不等待追随者副本的确认。这种模式可以提供更低的写入延迟，但在某些情况下可能会出现数据丢失。

**副本同步**：为了保持追随者副本与领导者副本的一致性，Kafka使用了基于日志的复制机制。追随者副本从领导者副本复制日志条目，并按顺序将它们追加到自己的日志中。复制过程采用高效的增量拉取方式，只拉取尚未复制的日志段。

**故障转移**：当领导者副本发生故障时，Kafka会自动选举一个追随者副本作为新的领导者。分区中的所有副本统称为 AR（Assigned Replicas），选举过程中，Kafka使用分区的ISR（In-Sync Replica）集合，即与领导者副本保持数据同步的追随者副本集合，延迟的追随者存入OSR(Outof-Sync Replicas)列表，新加入的follower也会先存放在OSR中。AR=ISR+OSR。只有在ISR中的追随者副本才有资格成为新的领导者。

* 如果故障的是领导者副本，Kafka会从ISR中选择一个追随者副本成为新的领导者。注意，如果ISR全部宕机，会从AR中选择第一个应答的副本当成leader，因此会造成消息丢失或者重复。
* 如果故障的是追随者副本，Kafka会将其从ISR中移除，并继续与其他追随者副本保持同步。

**简单总结一下**，Kafka的副本机制提供了数据冗余和故障恢复的功能，每个分区可以有多个副本并且可以分散在不同的节点上。副本的角色划分为领导者和追随者，领导者负责真实的读写以及将数据同步给追随者副本，有同步和异步两种模式。追随者仅负责数据冗余和故障转移恢复。所有副本统称AR，追随者和领导者数据同步一致的集合被称为ISR，其他为OSR，如果领导者出现故障则从ISR随机选择一个副本成为领导者，如果ISR挂了，就选择AR中第一个应答的副本。

### 副本分片规则

Kafka分配Replica的算法有两种: RangeAssignor 和 RoundRobinAssignor。默认为RangeAssignor，也可以自定义设置分片位置。

* 将所有Broker(假设共n个Broker)和待分配的Partition排序
* 将第i个Partition分配到第(i mod n)个Broker上
* 将第i个Partition的第j个Replica分配到第((i + j) mod n)个Broker上

### Kafka 的多分区（Partition）以及多副本（Replica）机制有什么好处呢？

1. **提高吞吐量**：多分区机制允许消息在多个分区上并行处理，从而提高了整体的吞吐量。每个分区可以在独立的消费者上进行并发处理，从而提高了系统的并行性和处理能力。
2. **实现水平扩展**：通过将数据分布在多个分区上，Kafka可以轻松地水平扩展。每个分区可以被部署在不同的服务器上，从而实现负载均衡和横向扩展，满足高吞吐量和大规模数据处理的需求。
3. **提高容错性**：多副本机制允许每个分区在多个副本之间进行复制。如果一个副本发生故障，Kafka可以自动将读写操作切换到其他可用的副本上，从而实现高可用性和容错性。此外，多副本机制还提供了数据冗余，即使某个副本损坏，数据仍然可用。
4. **实现数据持久化**：多副本机制确保数据在多个副本上进行复制，从而实现数据的持久化。即使在某个副本损坏或故障的情况下，数据仍然可以通过其他副本进行访问和恢复。
5. **支持消息的顺序性和分区**：多分区机制可以根据业务需求将消息分配到不同的分区中，并且每个分区可以保持消息的顺序。这对于需要保证消息顺序性的应用程序非常重要。同时，分区也可以用于实现消息的分组和隔离，从而更好地控制消息流。

### Zookeeper 在 Kafka 中的作用知道吗？

1. **协调器选举**：Zookeeper负责选举Kafka集群中的协调器（Coordinator）和控制器（Controller），两者可以由一个Broker同时担任。控制器负责管理整个Kafka集群的状态和元数据，并负责分配分区给各个Broker以实现负载均衡。
2. **配置管理**：Kafka的集群配置信息（如主题、分区、副本等）和消费者组的偏移量等元数据都被存储在Zookeeper中。Kafka的各个Broker和消费者通过与Zookeeper交互来获取最新的集群配置信息。
3. **Broker注册和发现**：Kafka的Broker在启动时会向Zookeeper注册自己的信息，包括主机名、端口号等。同时，消费者可以通过Zookeeper来发现可用的Broker节点。
4. **分区分配和重平衡**：当新的Broker加入集群或者旧的Broker下线时，Zookeeper协助控制器进行分区的重新分配和重平衡。它维护了分区的分配方案，并将这些信息通知给各个Broker，以确保数据的高可用性和负载均衡。
5. **副本管理**：Zookeeper跟踪和管理Kafka分区的副本信息。它监控副本的状态，并在副本发生故障时负责进行副本的重新分配和恢复。
6. **客户端会话管理**：Kafka的消费者和生产者通过Zookeeper维护与集群的会话连接。Zookeeper可以检测到客户端的活动状态，并处理连接的故障和恢复。

总的来说，Zookeeper在Kafka中承担了协调、配置管理、分区分配、副本管理和客户端会话管理等重要的角色，确保了Kafka集群的稳定运行和数据一致性。

### Kafka 为什么那么快

1. **分布式架构**：Kafka采用分布式架构，可以在多个节点上分布数据和负载，以实现并行处理和扩展性。消息被分割成多个分区，每个分区可以在集群中的不同节点上进行处理，从而实现水平扩展和负载均衡。
2. **零拷贝技术**：Kafka使用零拷贝技术，在读写消息时避免了不必要的数据拷贝操作。Kafka通过操作系统的文件映射（mmap）机制，直接将磁盘上的消息映射到内存中，减少了数据在内存和磁盘之间的复制开销，提高了读写性能。
3. **批量处理**：Kafka支持批量处理消息。生产者可以将多个消息一起发送到Kafka，消费者也可以批量拉取多个消息进行处理。批量处理减少了网络开销和系统调用次数，提高了吞吐量和效率。
4. **高效的磁盘顺序写**：Kafka的消息是追加写入到磁盘的，而不是随机写入。这种顺序写的方式使得磁盘的读写效率更高，减少了寻道时间和磁盘碎片，提高了磁盘的利用率和性能。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。
5. **基于内存的存储和缓存**：Kafka使用操作系统的页缓存（page cache）来缓存消息，以提高读写性能。热门的消息会被缓存在内存中，减少了磁盘访问的次数，加快了消息的读取速度。 如果Kafka 的写速率和消费速率差不多, 那么整个生产和消费过程是不会经过磁盘 IO. 全部都是内存操作
6. **高效的复制机制**：Kafka的复制机制采用了流式复制的方式，通过异步复制和批量复制的方式来提高复制的效率和吞吐量。同时，Kafka还使用了多个副本和ISR（In-Sync Replicas）机制，保证了数据的可靠性和高可用性。

简单总结下，一是多分区的分布式结构使得多线程的利用十分充分。二是利用操作系统的文件映射机制，直接将磁盘数据映射到内存，减少了一次复制开销，提升了数据读写性能。三是支持批量处理，生产者和消费者同时支持，减少了网络开销和请求次数。四是磁盘顺序写，磁盘写的效率比随机写更高。五是热门消息存在内存，提高读写性能。

分享一点冷门小知识

**Kafka为什么不自己管理缓存, 而非要用page cache?**

1. JVM中一切皆对象, 数据的对象存储会带来所谓 object overhead 浪费空间
2. 如果由JVM来管理缓存, 会受到GC的影响, 并且过大的堆也会拖累GC的效率, 降低吞吐量
3. 一旦程序崩溃, 自己管理的缓存数据会全部丢

## Kafka场景分析

**Kafka如何保证高可用**

1. **分布式架构**：Kafka采用分布式架构，将数据分散存储在多个Broker节点上。这使得即使某个Broker节点发生故障，其他正常运行的Broker节点仍然能够继续提供服务。
2. **多副本机制**：Kafka使用多副本机制来保证数据的冗余和可用性。每个分区都可以配置多个副本，这些副本分布在不同的Broker节点上。如果某个副本所在的Broker发生故障，其他副本仍然可以接管服务，确保数据的可用性。
3. **控制器（Controller）** ：Kafka集群中的控制器负责管理整个集群的状态和元数据。当某个Broker节点发生故障或下线时，控制器会监测到这个变化，并进行相应的操作，例如重新分配分区和副本的领导权，以保证数据的高可用性和一致性。
4. **自动故障恢复**：Kafka具有自动故障恢复的能力。当Broker节点发生故障或下线时，控制器会自动触发副本的重新分配和恢复过程，将副本分配到其他可用的Broker节点上，以保证数据的冗余和可用性。
5. **心跳和会话过期**：Kafka与客户端之间通过心跳机制维持连接，并定期检测客户端的活跃状态。如果某个消费者或生产者客户端长时间未发送心跳，控制器会认为该客户端的会话过期，并将其分区重新分配给其他活跃的消费者。
6. **监控和报警**：Kafka提供了丰富的监控指标和报警机制，可以实时监测集群的状态、分区的健康情况和消费者的进度等。这有助于及时发现和解决潜在的问题，提高系统的可用性和稳定性。

简单来说，分为四点。一是Kafka的分布式结构，多分区分布在不同节点，部分节点宕机依然能正常提供服务。二是多副本机制通过数据冗余来保障高可用。三是Kafka集群的控制器角色会管理整个集群的状态和元数据，一旦某个节点上线或者下线被监听到，会进行分区和副本领导权的重分配。四是Kafka集群中的协调者，在生产端保障数据的事务操作，并在消费端通过再平衡机制确保数据被消费者组顺利消费。

**Kafka如何保证消息不重复消费**

1. **消费者偏移量（Consumer Offset）** ：Kafka维护了每个消费者组消费的偏移量，即已经处理的消息的位置。消费者定期提交当前的偏移量，表明它已经成功消费了该消息。Kafka会保存这个偏移量，并在重启后将其还原，从而确保消费者可以继续从上次提交的偏移量处进行消费。
2. **消费者组协调器（Consumer Group Coordinator）** ：Kafka中的消费者组协调器负责跟踪和管理消费者组的偏移量。它会将每个消费者的偏移量信息存储在内部或外部的存储系统中（如Zookeeper或Kafka自身的内部主题\_\_consumer\_offsets）。通过协调器的管理，Kafka确保了消费者组内的每个消费者都可以正确地从上次的偏移量处继续消费。
3. **提交消费位移（Committing Consumer Offsets）** ：消费者可以选择在消费一批消息后手动提交偏移量，或者通过自动提交配置来定期自动提交偏移量。手动提交偏移量可以确保在消费失败或错误处理时，不会提交偏移量，从而避免重复消费。自动提交偏移量需要谨慎使用，确保处理消息的操作是幂等的。
4. **Exactly-Once语义**：Kafka引入了事务性生产者和消费者API，使得应用程序能够实现“精确一次”（exactly-once）语义。事务性生产者可以将消息写入Kafka事务，并在确认事务提交后才将消息暴露给消费者。事务性消费者会在处理消息后，通过事务提交来确保偏移量的原子性提交和消息处理的一致性。

简单来说，Kafka主要是通过消费者偏移量来确保消息不被重复消费。Kafka内部通过协调者和内部TOPIC(\_\_consumer\_offsets)来确保消费者实例不会重复消费。开发者则可以通过**手动提交消费偏移量**的方法，来保证消费失败或者出现异常时，主动不提交偏移量，从而避免重复消费。开发者还可以通过在消费端进行**幂等处理**来规避重复消费。

**Kafka如何保证消息的消费顺序？**

Kafka通过分区（partitions）和分区内的偏移量（offsets）来保证消息的顺序。主题可以是多个分区，但是每个分区中的数据是顺序的，生产者发送的消息顺序追加至分区末尾，消费者也是顺序消费数据。

也就是说，如果一个主题有多个分区，消息的顺序只在分区内得到保证，而不同分区之间的消息顺序无法被Kafka保证。如果应用程序的逻辑需要严格的全局顺序，那么所有相关的消息应该被发送到同一个分区，或者应用程序在消费者端进行额外的处理来实现全局顺序。

**Kafka 如何保证消息不丢失**

1. **持久化存储**：Kafka将消息持久化地存储在磁盘上，而不仅仅保存在内存中。每个主题的消息被写入多个分区（partition）中，每个分区又可以有多个副本（replica）。这样即使某个Broker或磁盘发生故障，消息仍然可以从其他副本中恢复，避免了消息的丢失。
2. **复制机制**：Kafka使用复制机制来提供高可用性和容错性。每个分区的消息可以在多个Broker上进行复制，形成副本集。副本集中的一个副本被指定为领导者（leader），负责读写操作，而其他副本则是追随者（follower）。当领导者副本发生故障时，追随者中的一个会被选举为新的领导者，确保消息的持久性和可用性。
3. **写入确认（Acknowledgement）** ：生产者在将消息发送到Kafka时，可以选择等待消息写入成功的确认。生产者可以配置等待的副本数，只有当指定数量的副本成功写入后，生产者才会收到确认。这种机制可以确保消息被写入到足够多的副本中，从而减少消息丢失的风险。
4. **副本同步**：Kafka使用副本同步机制来确保分区的副本保持同步。在写入消息时，领导者副本会将消息复制到所有追随者副本，并等待它们的确认。只有当所有追随者副本都成功复制并确认了消息后，领导者副本才会认为消息写入成功。这种同步机制确保了数据的一致性，减少了数据丢失的可能性。
5. **消费者位移提交**：消费者在消费消息时，会定期将消费的位移（offset）提交给Kafka。这样即使消费者出现故障或重新加入消费者组，它可以根据上次提交的位移来恢复消费。通过位移的提交，Kafka可以跟踪消费者的进度，并确保消息不会被重复消费或丢失。

简单总结下，Kafka多副本冗余数据的机制可以在多个节点拥有多个备份使用。并且ACK机制，确保生产者发送消息后，可以选择部分或者所有副本都收到消息，才确认结束请求，类似于事务机制。消费者通过偏移量的主动提交，同样能避免消息不被重复消费或丢失

**消息积压怎么处理**

1. 提高消费者的处理能力：您可以通过增加消费者实例的数量，提升消费者的并发处理能力。这样可以更快地消费消息，减少积压。
2. 调整消费者的处理逻辑：如果消费者的处理逻辑较为复杂，可能导致处理速度较慢。您可以评估和优化消费者的代码逻辑，提高其处理效率。
3. 增加分区数量：增加主题的分区数量可以提高消息的并行性。这样每个分区中的消息会被均匀地分配给不同的消费者实例，从而减轻积压问题。
4. 调整生产者的发送速率：如果消息挤压是由于生产者发送速率过快引起的，您可以调整生产者的发送速率，降低消息的产生速度，使消费者有足够的时间来处理消息。
5. 增加Kafka集群的资源：如果Kafka集群的资源（例如磁盘、网络带宽等）不足，可能会导致消息处理变慢。您可以考虑增加集群的资源，以提高整体的处理能力。

简单总结下，一般情况下都是消费压力大，可以通过临时扩充分区和消费者数量的方式提高消费速度。后续按照我自己的经历[消息积压问题难？思路代码优化细节全公开](https://juejin.cn/post/7209657931124555834" \o "https://juejin.cn/post/7209657931124555834" \t "_blank)，进行项目难点总结。

## Kafka源码

**生产端**

**NIO 网络通信模块**：Kafka 是如何基于Java NIO 实现一套工业生产级的网络底层通信模块

**内存缓冲池设计**：Kafka 客户端是如何设计一套支撑百万并发的高吞吐量的缓冲机制

**Sender发送线程**：重点的重点，Kafka 客户端是如何通过网络通信把一批批消息发送到 Kafka Broker 端上去的？这里涉及到很多网络通信的细节，一些参数设置，应对网络故障时的处理

**集群元数据拉取和更新机制**：集群元数据拉取组件以及拉取时机；元数据如何在客户端缓存的；如何对Topic元数据支持细粒度的按需加载和同步等待。

**服务端**

**集群架构**：Kafka Broker 的集群架构是如何实现的；各个 Broker启动起来后是如何组成一个集群的；集群的 Controller 是如何被选举出来的；故障恢复高可用架构方式是如何实现的等等。

**服务端网络通信模块**：Kafka 服务端网络通信模块是如何实现的、了解 Reactor 设计模型以及 Kafka 基于 Reactor 实现的支撑超高并发的网络架构、深度剖析 Acceptor 线程、Processor 线程、RequestChannel、IO线程池等网络底层通信组件实现以及请求处理全流程源码。

**分区与副本**：多副本冗余以及高可用架构如何实现；Leader 和 Follower 数据如何同步、副本如何传输、之间的HW和LEO如何变更；Leader 所在 Broker 故障宕机后， 如何保证系统高可用；副本管理器如何管理副本；Broker是如何异步更新元数据缓存的等等。

**负载均衡与伸缩架构**：如何保证数据均衡的分布在集群的 Broker 机器上的；如何进行 Topic 的 Partition 的扩缩容以及 Broker 的扩缩容。

**日志存储架构**：Kafka 是如何高效存储的；磁盘读写是如何实现的；日志的存储结构是怎样的；如何利用 OS Cache、零拷贝、稀疏索引、顺序写等优秀设计来支撑超高吞吐量的存储架构的。

**消费端**

**消费流程**：消费端是如何初始化的；如何与服务端进行通信的；Consumer 的 poll() 方法是如何进行数据消费的。

**消费者组管理**：Consumer Group 概念、状态机流转；Consumer启动后是如何加入到 Group 的；Consumer Group 管理全流程；Consumer Group 元数据管理设计原理等等。

**Coordinator 机制**：Consumer Coordinator 工作原理；Consumer Coordinator 是如何选举出来 Consumer Leader 的；Consumer Leader 如何制定分区分配方案；Consumer Coordinator 又是如何下发分区分配方案的； 以及 Consumer 是如何定时发送心跳给 Consumer Coordinator 的等等。

**消息重平衡机制**：几种重平衡场景；以及重平衡源码流程分析。

**\_\_consumer\_offsets**：topic中的topic。

**订阅状态与Offset操作**：消费端订阅状态是如何保存和追踪Topic Partition 和 Offset 的对应关系；以及了解 Offset 如何获取以及提交方式等等。

**如何设计一个消息队列？**

设计一个消息队列时，您可以考虑以下关键方面：

1. 定义需求和用途：明确消息队列的用途、预期的吞吐量、延迟要求、数据大小限制等需求。了解您要解决的问题和实现的目标，这将有助于确定合适的设计方向。
2. 选择适当的消息队列中间件：根据需求和用途选择适合的消息队列中间件。有许多开源和商业的消息队列中间件可供选择，如Kafka、RabbitMQ、ActiveMQ等。考虑它们的特性、可靠性、性能、可扩展性和社区支持等方面。
3. 定义消息格式和协议：设计消息的格式和协议，包括消息的结构、数据字段以及可能的消息头和元数据。这有助于消息的一致性和可扩展性，并提供给消费者有用的信息。
4. 考虑消息持久化和可靠性：确定是否需要将消息持久化到磁盘，以防止消息丢失。某些消息队列中间件提供持久化选项，确保消息在故障发生时不会丢失。
5. 考虑消息的顺序性：如果消息的顺序对您的应用程序很重要，需要选择一个支持有序消息传递的消息队列中间件，并根据需求设计适当的消息分区和顺序保证机制。
6. 考虑消息的分区和扩展：如果您预计有大量的消息和高吞吐量，需要考虑如何对消息进行分区和扩展。这可以提高并行性和系统的扩展性，确保消息能够有效地处理。
7. 实施适当的安全机制：考虑消息的安全性，包括身份验证、访问控制、数据加密等方面。根据需求选择适当的安全机制，保护消息的机密性和完整性。
8. 监测和管理：设计监测和管理机制，以便能够实时监控消息队列的性能、吞吐量和延迟等指标。这有助于及时发现问题并进行故障排除。

设计一个消息队列需要综合考虑多个因素，并根据实际需求和应用场景做出决策。这些指导原则可帮助您开始设计一个可靠、高性能和可扩展的消息队列系统。