# Kafka

## 简单说说Kafka是啥？常用在哪些地方？

Kafka就是一个用来处理数据的分布式工具。

它主要有三个功能：

1. **消息队列**：就像一个邮局，负责把消息从一个地方发送到另一个地方。
2. **安全地保存消息**：Kafka会把消息存在硬盘上，这样就不用担心消息丢失了。
3. **实时处理数据**：在消息发出的同时进行处理，Kafka还提供了很多工具来帮助我们完成这个任务。

Kafka主要用在两个地方：

1. **消息队列**：在不同的系统或应用之间实时传输数据，保证数据安全可靠。
2. **数据处理**：实时处理数据流，用来改变或处理数据。

## Kafka比其他消息队列的优势？

当我们谈到Kafka时，总会觉得它是个很棒的消息队列，常常和RocketMQ、RabbitMQ等进行比较。Kafka相比其他消息队列有以下优点：

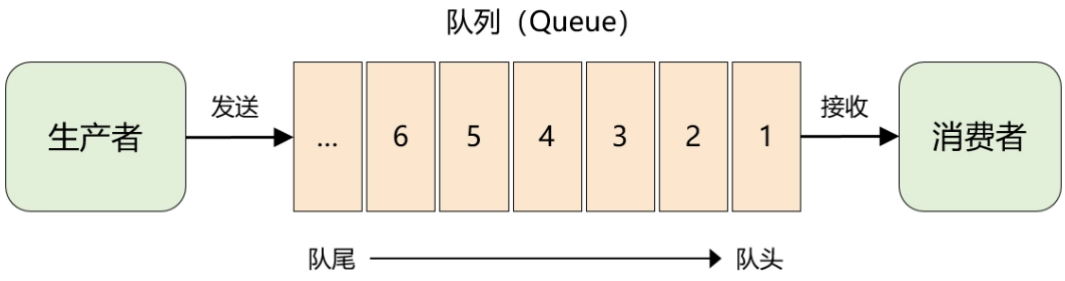
1. **高性能**：由Scala和Java开发，大量运用批处理和异步思想，每秒可以处理千万级别的消息。
2. **兼容性**：Kafka与其他生态系统的兼容性非常好，特别是在大数据和流计算领域。

其实早期的Kafka并不是个完美的消息队列，有些功能不够完善，比如丢失消息、消息可靠性不足等。这与最早LinkedIn开发Kafka是为了处理大量日志有关，本来就不是为了做消息队列的。不过，后来它在消息队列领域意外获得了成功。

随着时间的推移，Kafka逐渐修复了这些问题。所以，**说Kafka作为消息队列不可靠已经是过去式了！**现在的Kafka已经成为了一个非常可靠且功能强大的消息队列，受到了许多人的欢迎和青睐。

## 队列模型和Kafka的消息模型的区别

### 队列模型



队列模型就像一个传递消息的管道，生产者把消息放进队列，消费者从队列里拿消息。每条消息只能被一个消费者使用，消息会在队列里待着，直到被消费或过期。举个例子：生产者发了100条消息，两个消费者通常会各自消费一半的消息（轮流消费）。

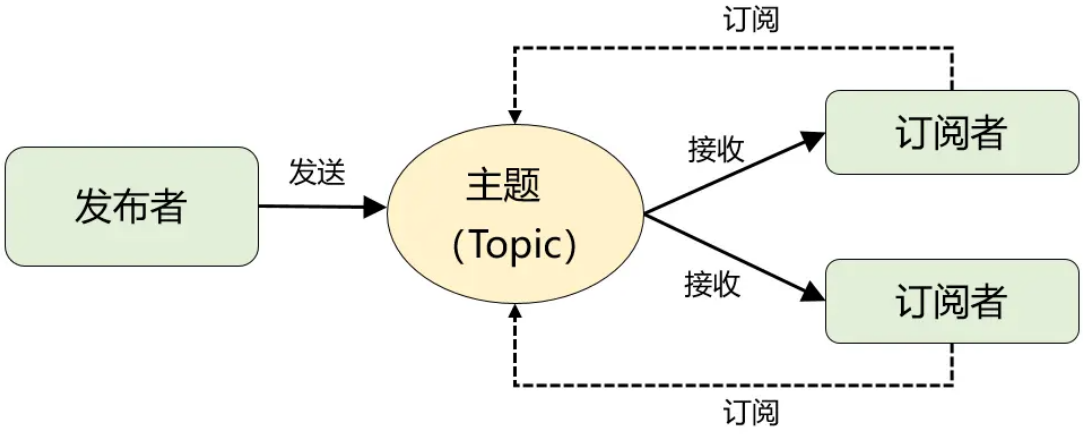
**队列模型的问题：**

想象这样一种情况：生产者产生的消息需要分发给多个消费者，每个消费者都能收到全部的消息内容。

队列模型在这种情况下就不太好用了。有人会想：为每个消费者创建一个单独的队列，让生产者发送多份。这样做很浪费资源，而且不符合使用消息队列的初衷。

## 发布-订阅模型：Kafka的消息模型

发布-订阅模型是为了解决队列模型存在的问题而提出的。




发布-订阅模型（Pub-Sub）通过**主题（Topic）进行消息传递，就像广播**一样。发布者发布一条消息，这条消息会通过主题传递给所有订阅者。**不过，如果用户在消息广播之后才订阅，那么他们将无法收到这条消息。**

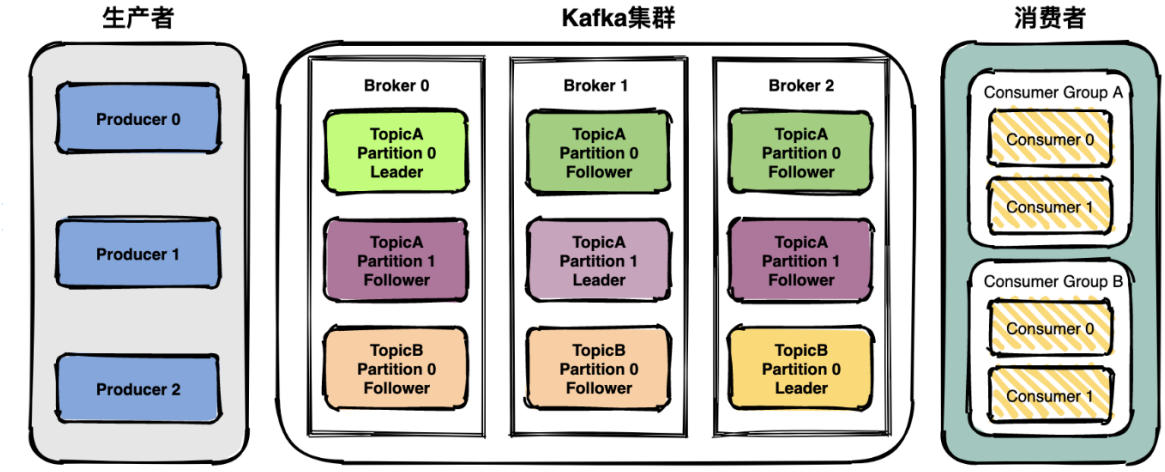
**在发布-订阅模型中，如果只有一个订阅者，那么它和队列模型其实就很相似了。因此，发布-订阅模型在功能上可以兼容队列模型。**

**Kafka正是采用的发布-订阅模型。**

> **RocketMQ的消息模型和Kafka基本相同。唯一的区别在于Kafka中没有队列这个概念，取而代之的是Partition（分区）。**

# Kafka的组件

包括：Producer、Consumer、Broker、Topic、Partition




Kafka系统里，生产者（Producer）会把消息发送到**主题（Topic）*中，*而需要这些消息的消费者（Consumer）则可以订阅这些*主题**（Topic），如下图所示：

在这个过程中，我们需要了解Kafka的几个重要概念：

1. **生产者（Producer）**：负责产生消息。
2. **消费者（Consumer）**：负责消费消息。
3. **代理（Broker）**：可以看作是一个独立的Kafka实例。多个Kafka代理组成一个Kafka集群。

此外，每个代理（Broker）里包含了主题（Topic）和分区（Partition）这两个重要概念：

- **主题（Topic）**：生产者把消息发送到特定的主题，消费者通过订阅特定的主题来消费消息。
- **分区（Partition）**：分区是主题的一部分。一个主题可以有多个分区，同一个主题下的分区可以分布在不同的代理（Broker）上，这也意味着一个主题可以跨越多个代理（Broker）。就像上面的图示一样。

  > 重点提示：在Kafka中，分区（Partition）实际上可以看作是消息队列中的队列。

# Kafka的多副本机制是什么？它有什么优点？

Kafka为分区（Partition）引入了多副本（Replica）机制。在一个分区里，有一个副本称为leader，其他副本称为follower。我们发送的消息会先发给leader副本，然后follower副本从leader那里同步消息。

> 生产者和消费者只和leader副本交互。其他副本就像leader副本的备份，它们存在是为了保证消息存储的安全性。如果leader出问题了，系统会从follower中选一个新的leader，但是如果follower和leader同步程度不够，它就不能参选。

**Kafka的多分区（Partition）和多副本（Replica）机制有哪些优点呢？**

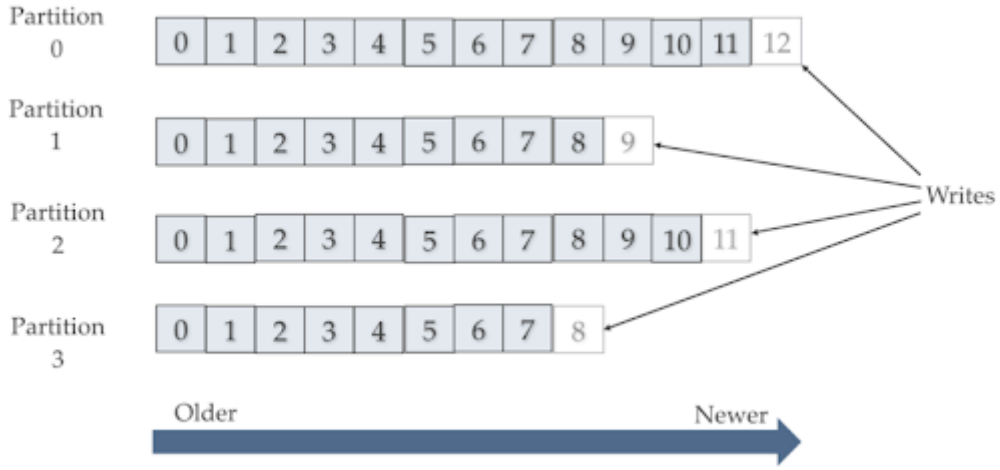
1. Kafka可以为特定的主题指定多个分区，而这些分区可以分布在不同的代理（Broker）上，从而提供很好的并发能力（负载均衡）。
2. 分区可以指定对应的副本数量，这大大提高了消息存储的安全性，增强了容灾能力，但相应地也增加了所需的存储空间。

# 如何确保Kafka中消息的消费顺序？

在使用消息队列时，我们经常遇到需要严格按照顺序消费消息的业务场景。例如，有两条消息对应的数据库操作分别是：

1. 修改用户会员等级。
2. 根据会员等级计算订单价格。

如果这两条消息的消费顺序不同，最终结果可能完全不同。




我们知道，Kafka的Partition（分区）是真正存储消息的地方。而Partition（分区）存在于Topic（主题）这个概念中，我们可以为特定的Topic指定多个Partition。

每次向Partition（分区）添加消息时，都会采用追加方式。**Kafka只能保证Partition（分区）中的消息是有序的。**

> 消息在追加到Partition（分区）时，都会分配一个特定的偏移量（offset）。Kafka通过偏移量（offset）来确保分区内消息的顺序性。

因此，我们有一种简单的方法来保证消息消费的顺序：**一个Topic只对应一个Partition**。虽然这样可以解决问题，但它破坏了Kafka的设计初衷。

在Kafka中发送一条消息时，可以指定topic、partition、key和data（数据）这四个参数。如果在发送消息时指定了Partition，那么所有消息都会被发送到指定的Partition。此外，具有相同key的消息可以确保只发送到同一个partition，我们可以使用表/对象的ID作为key。

总之，关于如何确保Kafka中消息消费顺序，我们有以下两种方法：

1. 一个Topic仅对应一个Partition。
2. （推荐）在发送消息时指定key/Partition。

# Kafka如何确保消息不会丢失

## 生产者可能会丢失消息的情况

当生产者（Producer）调用 `send` 方法发送消息时，由于网络问题，消息可能没有成功发送。我们不能默认调用 `send` 方法后消息就发送成功了。为了确定消息发送成功，我们需要判断发送结果。但请注意，Kafka生产者使用 `send` 方法发送消息实际上是异步操作。我们可以通过 `get()` 方法获取调用结果，但这会使其变为同步操作。通常，我们不推荐这样做，而是添加回调函数。如果消息发送失败，我们可以在检查失败原因后重新发送。此外，建议为生产者的 `retries` 设置一个合理的重试次数（通常为3），以及重试间隔，以避免消息丢失。

## 消费者可能会丢失消息的情况

Kafka通过偏移量（offset）确保消息在分区内的顺序性。消费者在拉取到分区中的某个消息后，会自动提交offset。为了避免消费者在消费消息前崩溃导致的消息丢失，我们可以关闭自动提交offset，并在真正消费完消息后手动提交offset。但这样可能导致消息被重复消费。例如，在消费完消息后还未提交offset时，消费者崩溃，这样消息实际上会被消费两次。

### Kafka可能丢失消息的情况

Kafka通过引入多副本（Replica）机制来保证分区（Partition）中的消息安全性。分区中的多个副本之间会有一个称为leader的副本，其他副本称为follower。我们发送的消息会被发送到leader副本，然后follower副本从leader副本中拉取消息进行同步。生产者和消费者只与leader副本交互。

假设leader副本所在的broker突然崩溃，此时需要从follower副本中选出一个新的leader。但如果leader的数据尚未被follower副本完全同步，就会导致消息丢失。为了避免这种情况，我们可以设置**acks = all**，表示所有ISR列表中的副本都收到消息时，生产者才认为消息发送成功。这是最安全的模式，但会增加延迟。

另外，我们通常为Topic设置 **replication.factor >= 3**，以确保每个分区（Partition）至少有3个副本。虽然这会导致数据冗余，但它确保了数据的安全性。

我们还需要设置 **min.insync.replicas > 1**，这意味着消息至少要被写入到2个副本才算发送成功。在实际生产中，应尽量避免将**min.insync.replicas**的默认值设为1。为了保证Kafka服务的高可用性，你需要确保**replication.factor > min.insync.replicas**。我们通常建议设置为**replication.factor = min.insync.replicas + 1**。

此外，为了减少消息丢失的可能性，我们可以设置 **unclean.leader.election.enable = false**。这样，在leader副本出现故障时，Kafka不会从与leader同步程度达不到要求的follower副本中选择新的leader。这有助于降低消息丢失的风险。

## 如何避免Kafka中的消息重复消费?

在Kafka中，消息可能会被重复消费。原因包括：

- 已消费的数据没有成功提交offset（根本原因）。
- 由于服务器处理业务时间过长或网络连接等原因，Kafka认为服务已经失效，从而触发了分区重新平衡（rebalance）。

为了解决这个问题，我们可以采取以下措施：

- 对消费消息进行幂等性检查，例如使用Redis的set或MySQL的主键等自然具有幂等性的功能。这种方法是最有效的。
- 将`enable.auto.commit`参数设置为false，关闭自动提交，并在代码中手动提交offset。这样会引入一个问题：何时提交offset合适?
  - 在处理完消息后提交：这样仍然存在消息重复消费的风险，与自动提交类似。
  - 在拉取到消息后立即提交：这将导致消息丢失的风险。在允许消息延迟的场景下，通常会采用这种方法。接着，通过定时任务在业务不繁忙（例如凌晨）的时候进行数据补充操作。这样可以在保证系统性能的同时，尽量避免消息的重复消费。

总之，在Kafka中避免消息重复消费需要权衡不同方案的优缺点。可以根据实际业务场景和需求，结合幂等性检查、手动提交offset以及定时任务进行数据补充等策略，来降低消息重复消费的风险。同时，确保系统的稳定性和性能不受影响。