

# Kafka 连环20问

苏三说技术 今天

以下文章来源于微观技术，作者TomGE



**微观技术**

前阿里P7技术专家，研究生，出过专利。负责过电商交易、社区团购、流量营销等业务...

大家好，我是苏三~

Kafka作为一款开源的消息引擎，很多人并不陌生，但深入其源码的同学估计不多，除非你是中间件团队消息系统维护者。但术业有专攻，市面上那么多开源框架且每个框架又经常迭代升级，花精力深入了解每一个框架源码不太现实，本文会以业务视角罗列工作中大家需要熟知的一些知识

**本篇文章的目录：**



## 首先, 为什么使用kafka?

- 削峰填谷。缓冲上下游瞬时突发流量, 保护“脆弱”的下游系统不被压垮, 避免引发全链路服务“雪崩”。
- 系统解耦。发送方和接收方的松耦合, 一定程度简化了开发成本, 减少了系统间不必要的直接依赖。

## Kafka 名词术语, 一网打尽

## Kafka名词术语 (公众号：微观技术)

Broker

主题

分区

消息

消息位移

副本

生产者

消费者

消费者位移

消费者组

重平衡

微观技术

- Broker：接收客户端发送过来的消息，对消息进行持久化
- 主题：Topic。主题是承载消息的逻辑容器，在实际使用中多用来区分具体的业务。
- 分区：Partition。一个有序不变的消息序列。每个主题下可以有多个分区。
- 消息：这里的消息就是指 Kafka 处理的主要对象。
- 消息位移：Offset。表示分区中每条消息的位置信息，是一个单调递增且不变的值。
- 副本：Replica。Kafka 中同一条消息能够被拷贝到多个地方以提供数据冗余，这些地方就是所谓的副本。副本还分为领导者副本和追随者副本，各自有不同的角色划分。每个分区可配置多个副本实现高可用。一个分区的N个副本一定在N个不同的Broker上。
- 生产者：Producer。向主题发布新消息的应用程序。
- 消费者：Consumer。从主题订阅新消息的应用程序。
- 消费者位移：Consumer Offset。表示消费者消费进度，每个消费者都有自己的消费者位移。offset保存在broker端的内部topic中，不是在clients中保存

- 消费者组：Consumer Group。多个消费者实例共同组成的一个组，同时消费多个分区以实现高吞吐。
- 重平衡：Rebalance。消费者组内某个消费者实例挂掉后，其他消费者实例自动重新分配订阅主题分区

## ZooKeeper 在里面的职责是什么？

它是一个分布式协调框架，负责协调管理并保存 Kafka 集群的所有元数据信息，比如集群都有哪些 Broker 在运行、创建了哪些 Topic，每个 Topic 都有多少分区以及这些分区的 Leader 副本都在哪些机器上等信息。

## 消息传输的格式

纯二进制的字节序列。当然消息还是结构化的，只是在使用之前都要将其转换成二进制的字节序列。

## 消息传输协议

- 点对点模型。系统 A 发送的消息只能被系统 B 接收，其他任何系统都不能读取 A 发送的消息
- 发布/订阅模型。该模型也有发送方和接收方，只不过提法不同。发送方也称为发布者（Publisher），接收方称为订阅者（Subscriber）。和点对点模型不同的是，这个模型可能存在多个发布者向相同的主题发送消息，而订阅者也可能存在多个，它们都能接收到相同主题的消息。

## 消息压缩

生产者程序中配置 `compression.type` 参数即表示启用指定类型的压缩算法。

`props.put("compression.type", "gzip")`，它表明该 Producer 的压缩算法使用的是 GZIP。这样 Producer 启动后生产的每个消息集合都是经 GZIP 压缩过的，故而能很好地节省网络传输带宽以及 Kafka Broker 端的磁盘占用。

但如果 Broker 又指定了不同的压缩算法，如：Snappy，会将生产端的消息解压然后按自己的算法重新压缩。

各压缩算法比较：吞吐量方面：LZ4 > Snappy > zstd 和 GZIP；而在压缩比方面，zstd > LZ4 > GZIP > Snappy。

kafka默认不指定压缩算法。

## 消息解压缩

当 Consumer pull消息时，Broker 会原样发送出去，当消息到达 Consumer 端后，由 Consumer 自行解压缩还原成之前的消息。

## 分区策略

编写一个类实现 `org.apache.kafka.clients.Partitioner` 接口。实现内部两个方法：`partition()` 和 `close()`。然后显式地配置生产者端的参数 `partitioner.class`

常见的策略：

- 轮询策略（默认）。保证消息最大限度地被平均分配到所有分区上。
- 随机策略。随机策略是老版本生产者使用的分区策略，在新版本中已经改为轮询了。
- 按key分区策略。key可能是uid或者订单id，将同一标志位的所有消息都发送到同一分区，这样可以保证一个分区内的消息有序
- 其他分区策略。如：基于地理位置的分区策略

## 生产者管理TCP连接

在new KafkaProducer 实例时，生产者应用会在后台创建并启动一个名为 Sender 的线程，该 Sender 线程开始运行时首先会创建与 Broker 的连接。此时还不知道给哪个topic发消息，所以Producer 启动时会发起与所有的 Broker 的连接。

Producer 通过 `metadata.max.age.ms` 参数定期地去更新元数据信息，默认值是 300000，即 5 分钟，不管集群那边是否有变化，Producer 每 5 分钟都会强制刷新一次元数据以保证它是最新的数据。

## Producer 发送消息：

Producer 使用带回调通知的发送 API，`producer.send(msg, callback)`。

设置 `acks = all`。Producer 的一个参数，表示所有副本都成功接收到消息，该消息才算是“已提交”，最高等级，acks的其它值说明。`min.insync.replicas > 1`，表示消息至少要被写入到多少个副本才算是“已提交”

retries 是 Producer 的参数。当出现网络的瞬时抖动时，消息发送可能会失败，此时配置了 retries > 0 的 Producer 能够自动重试消息发送，避免消息丢失。

## 幂等性 Producer

设置参数 `props.put("enable.idempotence", true)`，Producer 自动升级成幂等性 Producer，其他所有的代码逻辑都不需要改变。Kafka 自动帮你做消息的重复去重。

原理很简单，就是经典的空间换时间，即在 Broker 端多保存一些字段。当 Producer 发送了具有相同字段值的消息后，Broker 能够自动知晓这些消息已经重复了，可以在后台默默地把它们“丢弃”掉。

只能保证单分区、单会话上的消息幂等性。一个幂等性 Producer 能够保证某个topic的一个分区上不出现重复消息，但无法实现多个分区的幂等性。比如采用轮询，下一次提交交换了一个分区就无法解决

## 事务型 Producer

能够保证将消息原子性地写入到多个分区中。这批消息要么全部写入成功，要么全部失败。能够保证跨分区、跨会话间的幂等性。

```
producer.initTransactions();
try {
    producer.beginTransaction();
    producer.send(record1);
    producer.send(record2);
    //提交事务
    producer.commitTransaction();
} catch (KafkaException e) {
    //事务终止
    producer.abortTransaction();
}
```

实际上即使写入失败，Kafka 也会把它们写入到底层的日志中，也就是说 Consumer 还是会看到这些消息。要不要处理在 Consumer 端设置 `isolation.level`，这个参数有两个值：

- read\_uncommitted：这是默认值，表明 Consumer 能够读取到 Kafka 写入的任何消息
- read\_committed：表明 Consumer 只会读取事务型 Producer 成功提交事务写入的消息

## Kafka Broker 是如何存储数据？



Kafka 使用消息日志（Log）来保存数据，一个日志就是磁盘上一个只能追加写（Append-only）消息的物理文件。因为只能追加写入，故避免了缓慢的随机 I/O 操作，改为性能较好的顺序 I/O 写操作，这也是实现 Kafka 高吞吐量特性的一个重要手段。

不过如果你不停地向一个日志写入消息，最终也会耗尽所有的磁盘空间，因此 Kafka 必然要定期地删除消息以回收磁盘。怎么删除呢？

简单来说就是通过日志段（Log Segment）机制。在 Kafka 底层，一个日志又进一步细分成多个日志段，消息被追加写到当前最新的日志段中，当写满了一个日志段后，Kafka 会自动切分出一个新的日志段，并将老的日志段封存起来。Kafka 在后台还有定时任务会定期地检查老的日志段是否能够被删除，从而实现回收磁盘空间的目的。

## Kafka 的备份机制

相同的数据拷贝到多台机器上。副本的数量是可以配置的。Kafka 中 **follower** 副本不会对外提供服务。

副本的工作机制也很简单：生产者总是向 **leader** 副本写消息；而消费者总是从 **leader** 副本读消息。至于 **follower** 副本，它只做一件事：向 **leader** 副本以异步方式发送 **pull** 请求，请求 **leader** 把最新的消息同步给它，必然有一个时间窗口导致它和 **leader** 中的数据是不一致的，或者说它是落后于 **leader**。

## 为什么要引入消费者组？

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）。

在一个消费者组下，一个分区只能被一个消费者消费，但一个消费者可能被分配多个分区，因而在提交位移时也就能提交多个分区的位移。如果1个topic有2个分区，消费者组有3个消费者，有一个消费者将无法分配到任何分区，处于idle状态。

理想情况下，Consumer 实例的数量应该等于该 Group 订阅topic（可能多个）的分区总数。

## 消费端拉取（批量）、ACK

消费端先拉取并消费消息，然后再ack更新offset。

1) 消费者程序启动多个线程，每个线程维护专属的 **KafkaConsumer** 实例，负责完整的消息拉取、消息处理流程。一个 **KafkaConsumer** 负责一个分区，能保证分区内的消息消费顺序。

缺点：线程数受限于 Consumer 订阅topic的总分区数。

2) 任务切分成了消息获取和消息处理两个部分。消费者程序使用单或多线程拉取消息，同时创建专门线程池执行业务逻辑。优点：可以灵活调节消息获取的线程数，以及消息处理的线程数。

缺点：无法保证分区内的消息消费顺序。另外引入了多组线程，使得整个消息消费链路被拉长，最终导致正确位移提交会变得异常困难，可能会出现消息的重复消费或丢失。

## 消费端offset管理

1) 老版本的 Consumer组把位移保存在 ZooKeeper 中，但很快发现zk并不适合频繁的写更新。

2) 在新版本的 Consumer Group 中，Kafka 社区重新设计了 Consumer组的位移管理方式，采用了将位移保存在 Broker端的内部topic中，也称为“位移主题”，由kafka自己来管理。原理很简单，Consumer的位移数据作为一条条普通的 Kafka 消息，提交到 `__consumer_offsets` 中。它的消息格式由 Kafka 自己定义，用户不能修改。位移主题的 Key 主要包括 3 部分内容：<Group ID, topic名, 分区号 >

Kafka Consumer 提交位移的方式有两种：自动提交位移和手动提交位移。

Kafka 使用Compact策略来删除位移主题中的过期消息，避免该topic无限期膨胀。提供了专门的后台线程定期地巡检待 Compact 的主题，看看是否存在满足条件的可删除数据。

## Rebalance 触发条件

1) 组成员数发生变更。比如有新的 Consumer 实例加入组或者离开组，又或是有 Consumer 实例崩溃被“踢出”组。（99%原因是由它导致）

2) 订阅topic数发生变更。Consumer Group 可以使用正则表达式的方式订阅topic，比如 `consumer.subscribe(Pattern.compile("t.*c"))` 就表明该 Group 订阅所有以字母 t 开头、字母 c 结尾的topic。在 Consumer Group 的运行过程中，你新创建了一个满足这样条件的topic，那么该 Group 就会发生 Rebalance。

3) 订阅topic的分区数发生变化。Kafka 目前只允许增加topic的分区数。当分区数增加时，也会触发订阅该topic的所有 Group 开启 Rebalance。

## 消息的顺序性



Kafka的设计中多个分区的话无法保证全局的消息顺序。如果一定要实现全局的消息顺序，只能单分区。

方法二：通过有key分组，同一个key的消息放入同一个分区，保证局部有序

## 历史数据清理策略

- 基于保存时间， `log.retention.hours`
- 基于日志大小的清理策略。通过 `log.retention.bytes` 控制
- 组合方式

---

关于我：前阿里架构师，出过专利，竞赛拿过奖，CSDN博客专家，负责过电商交易、社区生鲜、营销、金融等业务，多年团队管理经验，爱思考，喜欢结交朋友。



微观技术

前阿里P7技术专家，研究生，出过专利。负责过电商交易、社区团购、流量营销等业务...

111篇原创内容

---

公众号

喜欢此内容的人还喜欢

Mysql 连环20问

苏三说技术

---

Python 协程 asyncio 极简入门与爬虫实战

重科sugon瑞翼大数据

---

mysql数据库迁移至Oracle数据库

全栈数据