

# 【建议收藏】Kafka 面试连环炮, 看看你能撑到哪一步? (上)

原创 王江华 华仔聊技术 2022-03-07 08:00

收录于话题

#消息队列 14 #kafka 14 #中间件 15 #面试 19 #大数据 14

本系列总共4万多字, 本文是上篇, 阅读本文大约需要 30 分钟。

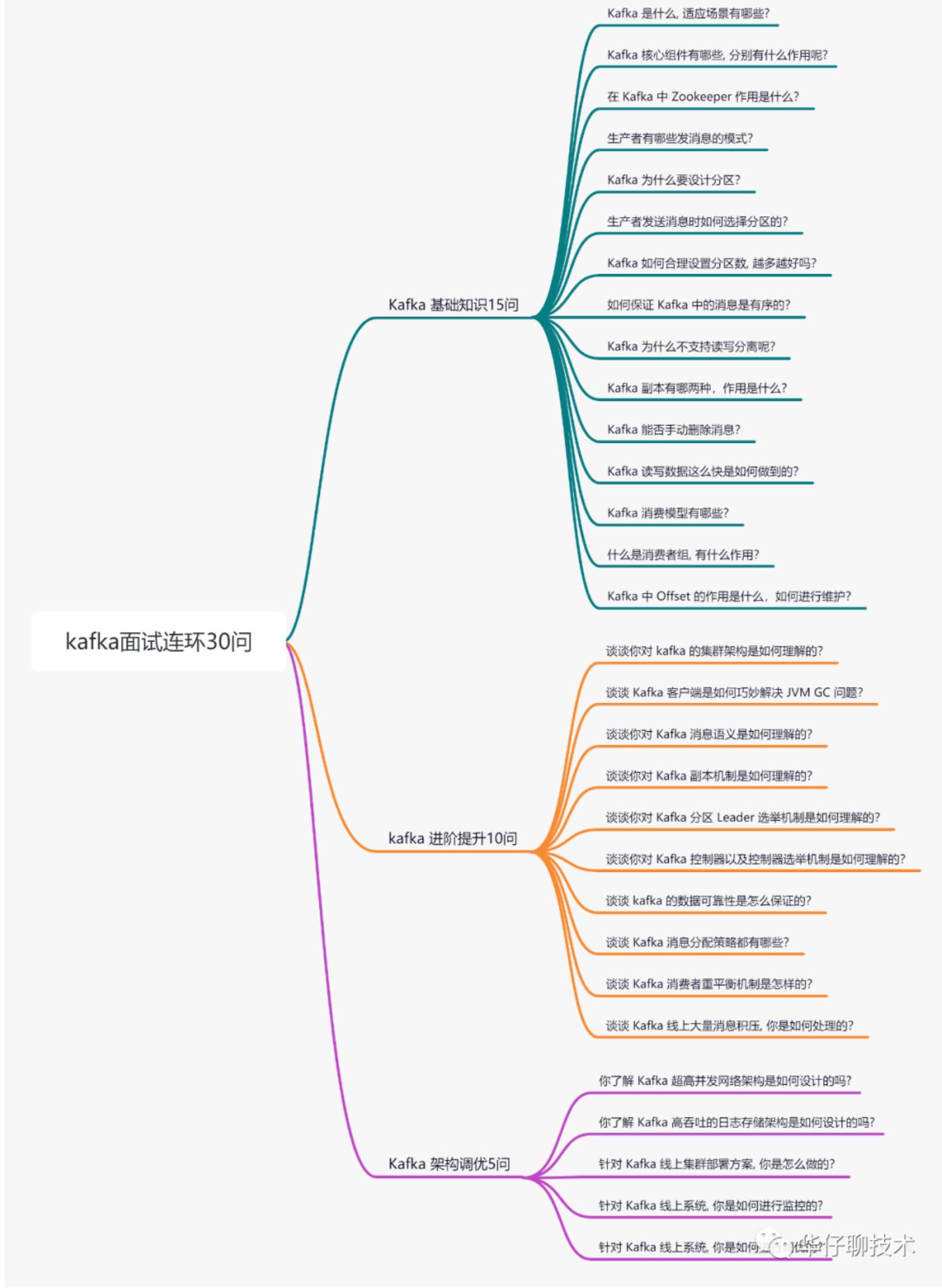
大家好, 我是 华仔, 又跟大家见面了。

之前有粉丝留言说能否总结和分享一些 Kafka 相关的面试题。

今天我们就来安排一期关于 Kafka 的核心面试题连环炮, 从「**基础知识**」、「**进阶提升**」、「**架构调优**」 三个方向梳理面试题, 希望在金三银四的关键节点可以帮助到大家。

由于内容很多, 打算拆分成「**上中下**」三篇, 本文是面试系列的上篇, 主要输出基础知识方面的面试题。

**这篇文章干货很多, 希望你可以耐心读完。**



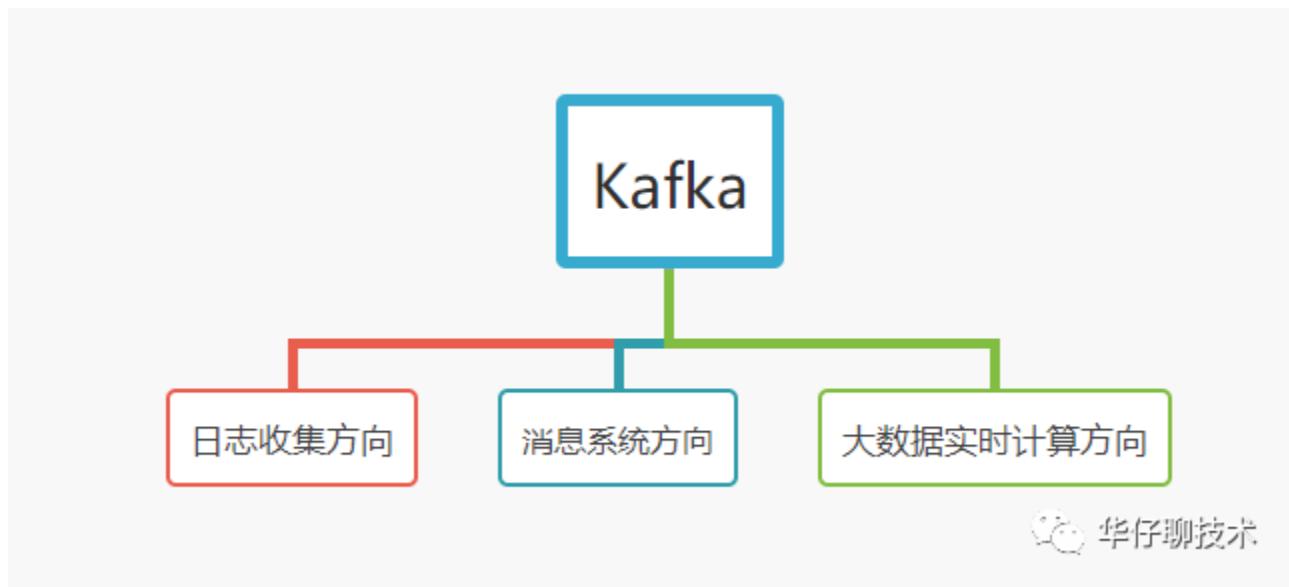
## Kafka 是什么, 适应场景有哪些?

Kafka 是一个分布式的流式处理平台, 用于实时构建流处理应用。主要应用在大数据实时处理领域。Kafka 凭借「高性能」、「高吞吐」、「高可用」、「低延迟」、「可伸缩」几大特性, 成为「消息队列」的首选。

其主要设计目标如下:

- 1) **高性能**: 以时间复杂度为  $O(1)$  的方式提供消息持久化能力, 即使对 TB 级以上数据也能保证常数时间的访问性能。
- 2) **高吞吐、低延迟**: 在非常廉价的机器上也能做到单机支持每秒几十万条消息的传输, 并保持毫秒级延迟。
- 3) **持久性、可靠性**: 消息最终被持久化到磁盘, 且提供数据备份机制防止数据丢失。
- 4) **容错性**: 支持集群节点故障容灾恢复, 即使 Kafka 集群中的某一台 Kafka 服务节点宕机, 也不会影响整个系统的功能 (若副本数量为 N, 则允许 N-1 台节点故障)。
- 5) **高并发**: 可以支撑数千个客户端同时进行读写操作。

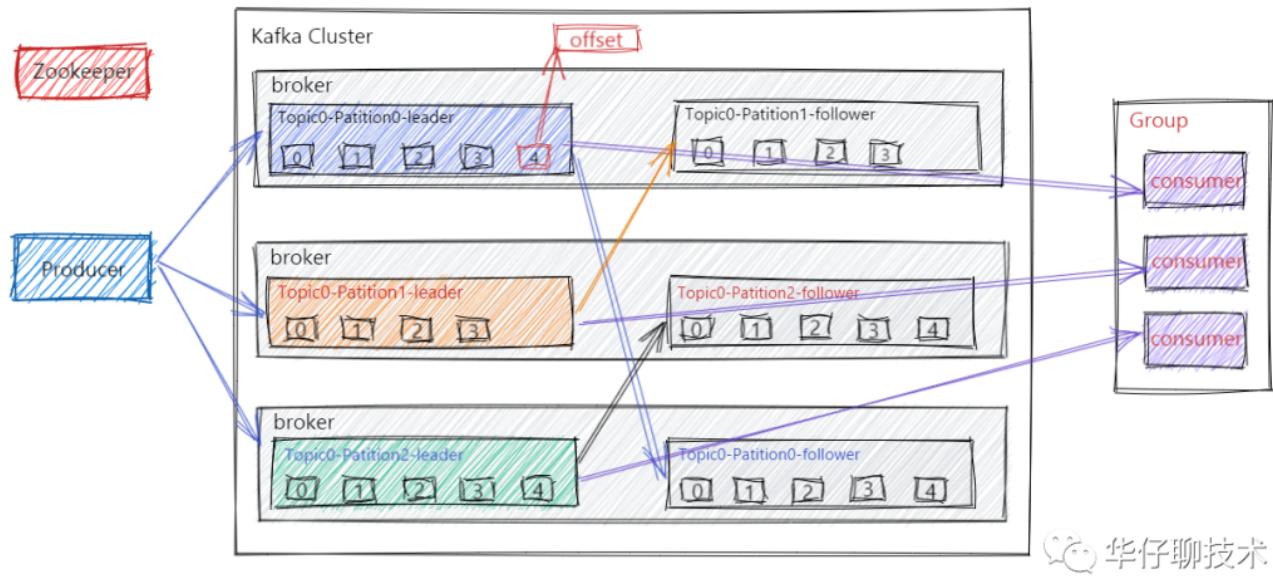
其适应场景主要有:



- 1) **日志收集方向**: 可以用 Kafka 来收集各种服务的 log, 然后统一输出, 比如日志系统 elk, 用 Kafka 进行数据中转。
- 2) **消息系统方向**: Kafka 具备系统解耦、副本冗余、流量削峰、消息缓冲、可伸缩性、容错性等功能, 同时还提供了消息顺序性保障以及消息回溯功能等。

3) 大数据实时计算方向: Kafka 提供了一套完整的流式处理框架, 被广泛应用到大数据处理, 如与 flink、spark、storm 等整合。

## Kafka 核心组件有哪些, 分别有什么作用呢?



### Kafka 核心组件的基础概念:

- 1) **Producer:** 即消息生产者, 向 Kafka Broker 发消息的客户端。
- 2) **Consumer:** 即消息消费者, 从 Kafka Broker 读消息的客户端。
- 3) **Consumer Group:** 即消费者组, 由多个 **Consumer** 组成。消费者组内每个消费者负责消费不同分区的数据, 以提高消费能力。一个分区只能由组内一个消费者消费, 不同消费者组之间互不影响。
- 4) **Broker:** 一台 Kafka 服务节点就是一个 **Broker**。一个集群是由1个或者多个 Broker 组成的, 且一个 Broker 可以容纳多个 Topic。
- 5) **Topic:** 一个逻辑上的概念, Topic 将消息分类, 生产者和消费者面向的都是同一个 Topic, 同一个 Topic 下的 Partition 的消息内容是不相同的。
- 6) **Partition:** 为了实现 Topic 扩展性, 提高并发能力, 一个非常大的 Topic 可以分布到多个 Broker 上, 一个 Topic 可以分为多个 Partition 进行存储, 且每个 Partition 是消息内容是有序的。
- 7) **Replica:** 即副本, 为实现数据备份的功能, 保证集群中的某个节点发生故障时, 该节点上的 Partition 数据不丢失。

失，且 Kafka 仍然能够继续工作，为此 Kafka 提供了副本机制，一个 Topic 的每个 Partition 都有若干个副本，一个 Leader 副本和若干个 Follower 副本。

8) **Leader**: 即每个分区多个副本的主副本，生产者发送数据的对象，以及消费者消费数据的对象，都是 Leader。

9) **Follower**: 即每个分区多个副本的从副本，会实时从 Leader 副本中同步数据，并保持和 Leader 数据的同步。

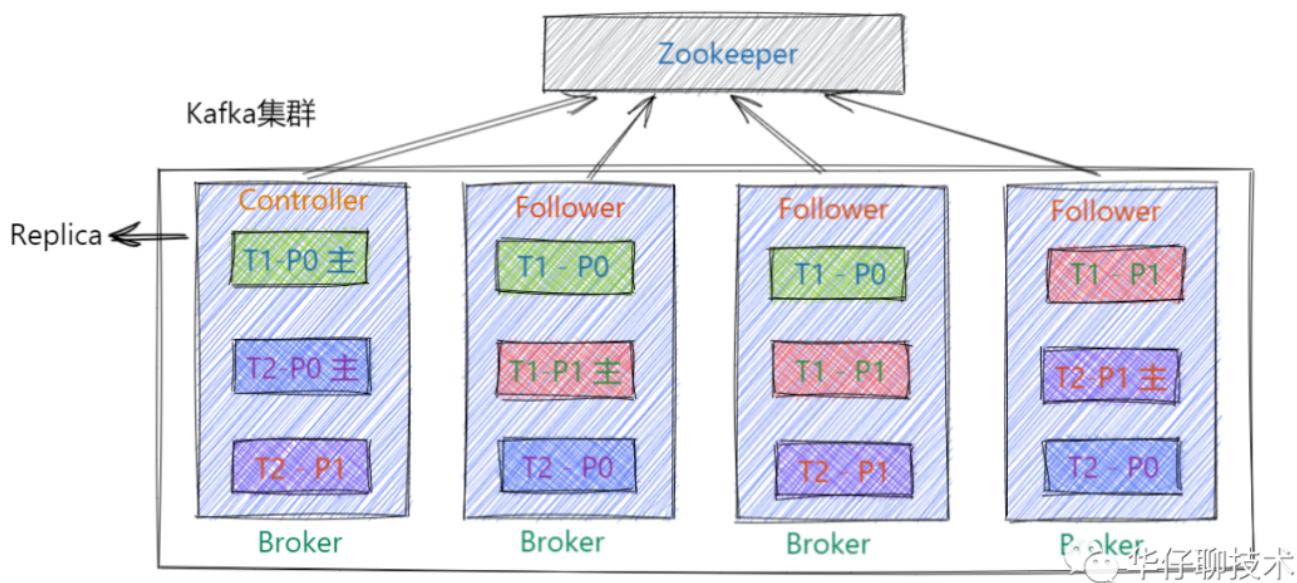
Leader 发生故障时，某个 Follower 还会被选举并成为新的 Leader，且不能跟 Leader 在同一个 Broker 上，防止崩溃数据可恢复。

10) **Offset**: 消费者消费的位置信息，监控数据消费到什么位置，当消费者挂掉再重新恢复的时候，可以从消费位置继续消费。

## 在 Kafka 中 Zookeeper 作用是什么？

Kafka 集群能够正常工作，**目前还是**需要依赖于 ZooKeeper，主要用来「负责 Kafka 集群元数据管理，集群协调工作」，在每个 Kafka 服务器启动的时候去连接并把自己注册到 Zookeeper，类似注册中心。

Kafka 使用 Zookeeper 存放「**集群元数据**」、「**集群成员管理**」、「**Controller 选举**」、「**其他管理类任务**」等。待 KRaft 提案完成后，Kafka 将完全不依赖 Zookeeper。



1) **集群元数据**: Topic 对应 Partition 的所有数据都存放在 Zookeeper 中，且以 Zookeeper 保存的数据为准。

2) **集群成员管理**: Broker 节点的注册、删除以及属性变更操作等。主要包括两个方面：**成员数量的管理**，主要体现在新增成员和移除现有成员；**单个成员的管理**，如变更单个 Broker 的数据等。

3) **Controller 选举**: 即选举 Broker 集群的控制器 Controller。其实它除了具有一般 Broker 的功能之外，还具有**选举主题分区 Leader 节点的功能**。在启动 Kafka 系统时，其中一个 Broker 会被选举为控制器，负责管理主题分区和副本状态，还会执行分区重新分配的管理任务。如果在 Kafka 系统运行过程中，当前的控制器出现故障导致不可用，那么 Kafka 系统会从其他正常运行的 Broker 中重新选举出新的控制器。

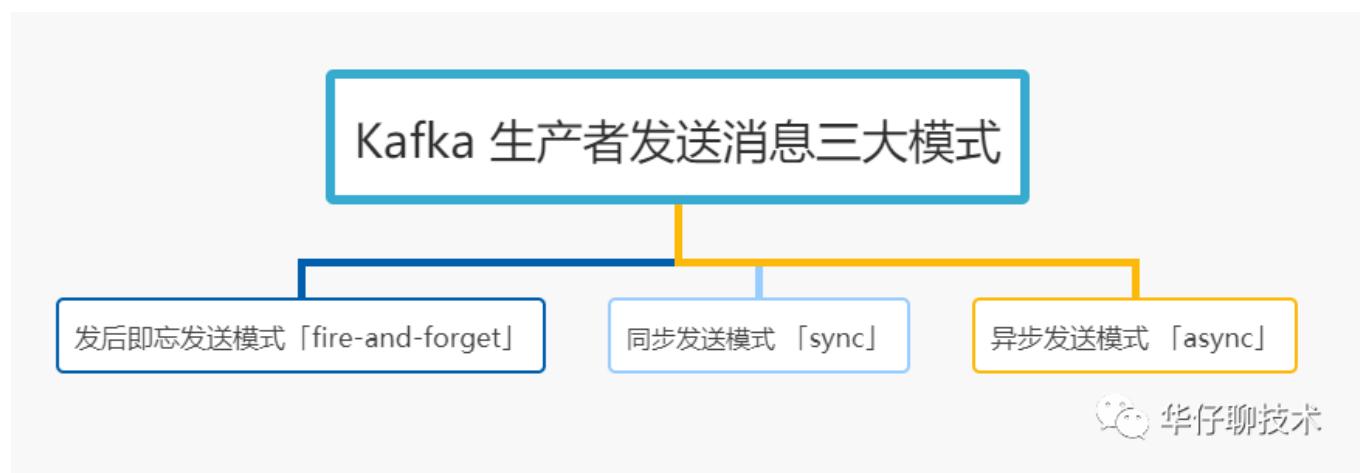
4) **其他管理类任务**: 包括但不限于 Topic 的管理、参数配置等等。

Kafka 3.X 「**2.8版本开始**」为什么移除 Zookeeper 的依赖的原因有以下2点：

- 1) **集群运维层面**: Kafka 本身就是一个分布式系统，如果还需要重度依赖 Zookeeper，集群运维成本和系统复杂度都很高。
- 2) **集群性能层面**: Zookeeper 架构设计并不适合这种高频的读写更新操作，由于之前的提交位移的操作都是保存在 Zookeeper 里面的，这样的话会严重影响 Zookeeper 集群的性能。

## 生产者有哪些发消息的模式？

Kafka 生产者发送消息主要有三种模式：



### 发后即忘发送模式 01

**发后即忘模式「fire-and-forget」**，它只管发送消息，并不需要关心消息是否发送成功。其本质上也是一种**异步发送**的方式，消息先存储在缓冲区中，达到设定条件后再批量进行发送。这是 **kafka 吞吐量最高的方式**，但同时也是**消息最不可靠的方式**，因为对于发送失败的消息并没有做任何处理，某些异常情况下会导致消息丢失。

```
1 ProducerRecord<k,v> record = new ProducerRecord<k,v>("this-topic", key, value);
2 try {
3     //fire-and-forget 模式
4     producer.send(record);
5 } catch (Exception e) {
6     e.printStackTrace();
7 }
```

## 同步发送模式 02

同步发送模式「sync」，调用 send() 方法会返回一个 Future 对象，再通过调用 Future 对象的 get() 方法，等待结果返回，根据返回的结果可以判断消息是否发送成功，**由于是同步发送会阻塞，只有当消息通过 get() 返回数据时，才会继续下一条消息的发送。**

```
1 ProducerRecord<k,v> record = new ProducerRecord<k,v>("this-topic", key, value);
2 try {
3     //sync 模式 调用future.get()
4     future = producer.send(record);
5     RecordMetadata metadata = future.get();
6 } catch (Exception e) {
7     e.printStackTrace();
8 }
9 producer.flush();
10 producer.close();
```

## 异步发送模式 03

异步发送模式「async」，在调用 send() 方法的时候指定一个 callback 函数，当 Broker 接收到返回的时候，该 callback 函数会被触发执行，**通过回调函数能够对异常情况进行处理，当调用了回调函数时，只有回调函数执行完毕生产者才会结束，否则一直会阻塞。**

```
1 Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback);
2 public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback)
3         //intercept the record, which can be potentially modified; this method doe
```

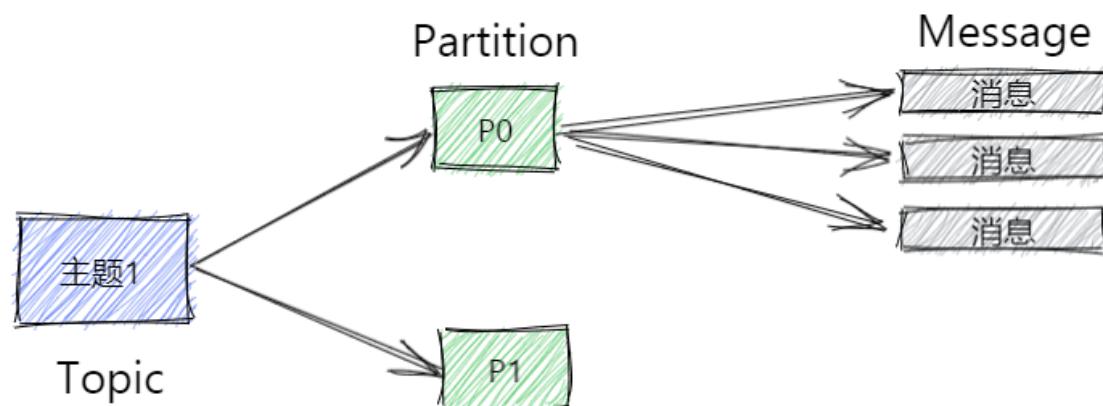
```
4 ProducerRecord<K, V> interceptedRecord = this.interceptors == null ? recor
5   return doSend(interceptedRecord, callback);
6 }
```

以上三种方式各有各的特点，具体还要看业务的应用场景适合哪一种：

- 1) **场景1：**如果业务只是关心消息的吞吐量，且允许少量消息发送失败，也不关注消息的发送顺序的话，那么可以使用发后即忘发送「**fire-and-forget**」的方式，配合参数 `acks = 0`，这样生产者并不需要等待服务器的响应，以网络能支持的最大速度发送消息。
- 2) **场景2：**如果业务要求消息必须是按**顺序发送**的话，且数据只能落在一个 `Partition` 上，那么可以使用同步发送「**sync**」的方式，并结合参数来设置 `retries` 的值让消息发送失败时可以进行多次重试「**retries > 0**」，再结合参数设置「**acks=all & max\_in\_flight\_requests\_per\_connection=1**」，可以控制生产者在收到服务器成功响应之前只能发送1条消息，在消息发送成功后立即进行 `flush`，从而达到控制消息顺序发送。
- 3) **场景3：**如果业务需要知道消息是否发送成功，但对消息的顺序并不关心的话，那么可以用「**异步async + 回调callback 函数**」的方式来发送消息，并配合参数 `retries=0`，待发送失败时将失败的消息记录到日志文件中进行后续处理。

## Kafka 为什么要设计分区？

其实这个问题说来很简单，假如不进行分区的话就如同 MySQL 单表存储一样，发消息就会被集中存储，这样会导致某台 Kafka 服务器存储 Topic 消息过多，如果在写消息压力很大的情况下，最终会导致这台 Kafka 服务器吞吐量出现瓶颈，因此 Kafka 设计了分区的概念，同时也带来了「**负载均衡**」、「**横向扩展**」的能力，如下图所示：。



- 1) **负载均衡**: 发送消息时可以根据分区的数量进行数据均匀分布, 使其落在不同的分区上, 这样可以提高并发写性能; 同时消费的时候多个订阅者可以从一个或者多个分区中同时消费数据, 以支撑海量数据处理能力, 提高读消息性能。
- 2) **横向扩展**: 可以将一个 Topic 分成了多个 Partition, 将不同的 Partition 尽可能的部署在不同的物理节点上, 这样扩展起来非常方便, 另外一个消费者可以消费多个分区中的数据, 但是这样还是不能够充分的发挥横向扩展, **这时候消费者组就出现了**, 我们用消费者组, 来消费整个的 Topic, 一个消费者消费 Topic 中的一个分区。

## 生产者发送消息时如何选择分区的?

生产者发送消息的时候选择分区的策略方式主要有以下4种:

### Kafka 生产者发送消息选择分区的策略方式

轮询策略      消息key指定分区策略      随机策略      自定义策略

华仔聊技术

- 1) **轮询策略**: 顺序分配消息, 即按照消息顺序依次发送到某Topic下不同的分区, 它总是能保证消息最大限度地被平均分配到所有分区上, **如果消息在创建的时候 key 为 null, 那么Kafka 默认会采用这种策略**。
- 2) **消息key指定分区策略**: Kafka 允许为每条消息定义 key, 即消息在创建的时候 key 不为空, 此时 Kafka 会根据消息的 key 进行 hash, 然后根据 hash 值对 Partition 进行取模映射到指定的分区上, 这样的好处就是相同 key 的消息会发送到同一个分区上, 这样 **Kafka 虽然不能保证全局有序, 但是可以保证每个分区的消息是有序的, 这就是消息分区有序性**, 适应场景有下单支付的时候希望消息有序, 可以通过订单 id 作为 key 发送消息达到分区有序性。
- 3) **随机策略**: 随机发送到某个分区上, 看似也是将消息均匀打散分配到各个分区, 但是性能还是无法跟轮询策略比, 「**如果追求数据的均匀分布, 最好还是使用轮询策略**」。
- 4) **自定义策略**: 可以通过实现 org.apache.kafka.clients.producer.Partitioner 接口, 重写 partition 方法来达到自定义分区效果。

## Kafka 如何合理设置分区数,越多越好吗?

### 一、Kafka 如何合理设置分区数

首先我们要了解在 Partition 级别上达到负载均衡是实现高吞吐量的关键，合适的 Partition 数量可以达到并行读写和负载均衡的目的，需要根据每个分区的生产者和消费者的目标吞吐量进行估计。

此时我们可以遵循一定的步骤来计算确定分区数：

- 1) 首先根据某个 Topic 当前接收的数据量等经验来确定分区的初始值。
- 2) 然后针对这个 Topic，进行测试 Producer 端吞吐量和 Consumer 端的吞吐量。
- 3) 测试的结果，假设此时他们的值分别是  $T_p$  「Producer 端吞吐量」、 $T_c$  「Consumer 端吞吐量」，总的目标吞吐量是  $T_t$ ，单位是 MB/s，那么结果  $\text{numPartition} = T_t / \max(T_p, T_c)$ 。
- 4) **特殊说明：**测试  $T_p$  通常很容易的，因为它的逻辑非常简单，就是直接发送消息到 Kafka 就好了。而测试  $T_c$  通常与应用消费消息后进行其他什么处理有关，相对复杂一些。

### 二、分区设置越多越好吗？

首先 Kafka 高吞吐量的原因之一就是通过 Partition 将 Topic 中的消息均衡保存到 Kafka 集群中不同的 Broker 中。

「理论上说，如果一个 Topic 分区越多，整个集群所能达到的吞吐量就越大」。但是，实际生产中 Kafka Topic 的分区数真的配置越多越好吗？很显然不是！分区数过多会有什么弊端和问题呢，我们可以从下面4个方向进行深度分析：

# Kafka Partition 过多弊端和问题分析

使用内存方面分析

消耗文件句柄方面分析

端到端的延迟方面分析

高可用性方面分析

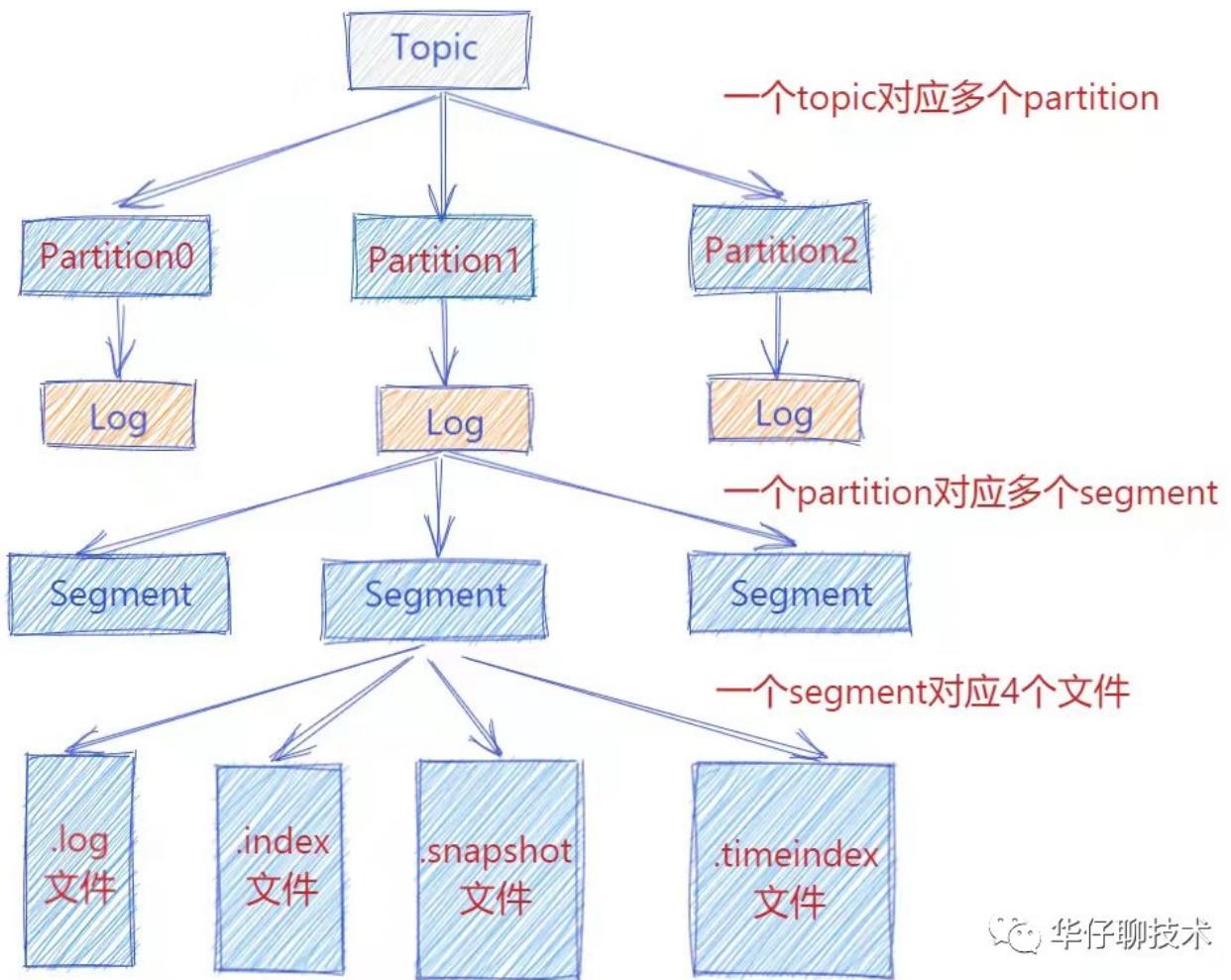
华仔聊技术

## 使用内存方面分析 01

- 1) **Broker端**: 有很多组件都在内存中维护了分区级别的缓存, 比如 Controller, FetcherManager 等, 因此分区数越多, 这类缓存的成本就越大。
- 2) **Producer端**: 比如参数 batch.size, 默认是16KB。它会为每个分区缓存消息, 在数据积累到一定大小或者足够的时间时, 累积的消息将会从缓存中移除并发往Broker 节点。这个功能是为了提高性能而设计, 但是随着分区数增多, 这部分缓存所需的内存占用也会更多。
- 3) **Consumer端**: 消费者数跟分区数是直接挂钩的, 在消费消息时的内存占用、以及为达到更高的吞吐性能需要开启的 Consumer 数也会随着分区数增加而增加。

## 消耗文件句柄方面分析 02

在 Kafka 的 Broker 中, 每个 Partition 都会对应磁盘文件系统中一个目录。在 Kafka 的日志文件目录中, 每个日志数据段都会分配三个文件, 两个索引文件和一个数据文件。每个 Broker 会为每个日志段文件打开两个 index 文件句柄和一个 log 数据文件句柄。因此, 随着 Partition 的增多, 所需要保持打开状态的文件句柄数也就越多, 最终可能超过底层操作系统配置的文件句柄数量限制。



华仔聊技术

### 端到端的延迟方面分析 03

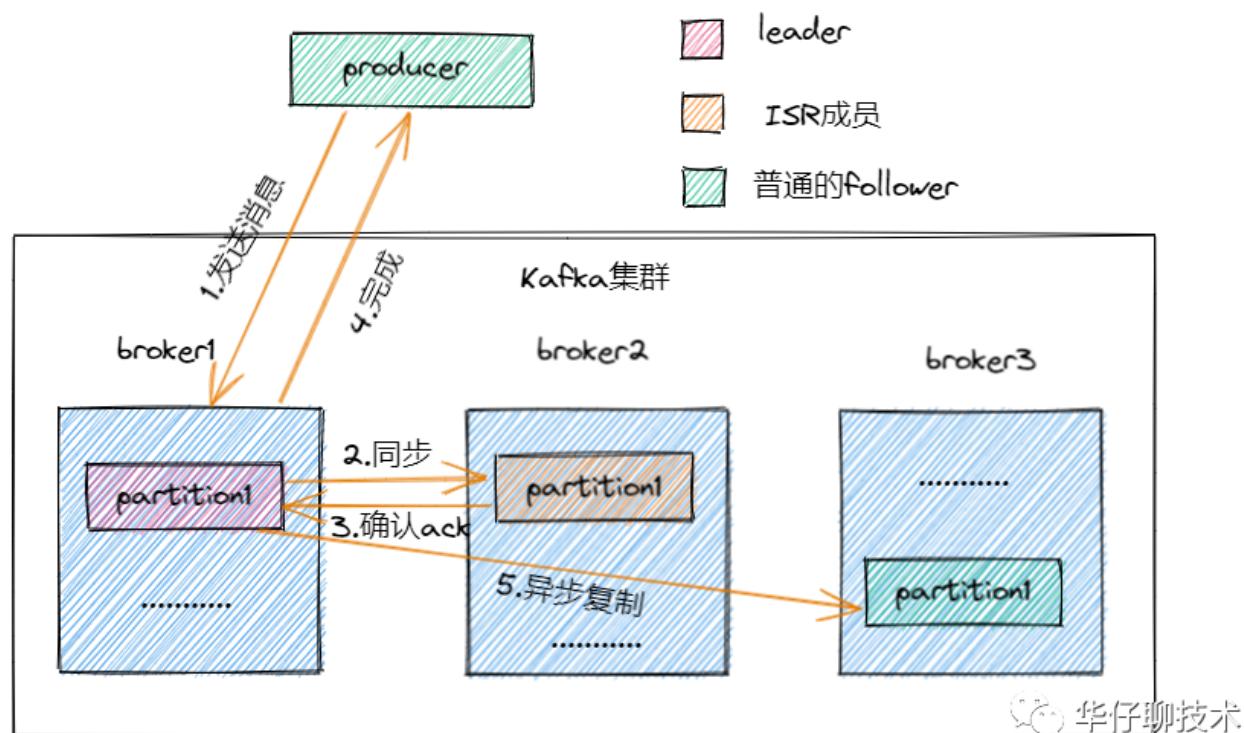
首先我们得先了解 Kafka 端对端延迟是什么？Producer 端发布消息到 Consumer 端接收消息所需要的时间，即 Consumer 端接收消息的时间减去 Producer 端发布消息的时间。

在 Kafka 中只对「已提交的消息做最大限度的持久化保证不丢失」，因此 Kafka 只有在消息提交之后，才会将消息暴露给消费者。此时如果分区越多那么副本之间需要同步的数据就会越多，假如消息需要在所有 ISR 副本集合列表同步复制完成之后才能进行暴露。因此 ISR 副本集合间复制数据所花时间将是 kafka 端对端延迟的主要部分。

此时我们可以通过加大 kafka 集群来进行缓解。比如，我们将 100 个分区 Leader 放到一个 Broker 节点和放到 10 个 Broker 节点，它们之间的延迟是不一样的。如在 10 个 Broker 节点的集群中，每个 Broker 节点平均只需要处理 10 个分区的数据复制。此时端对端的延迟将会变成原来的十分之一。

因此根据实战经验，如果你特别关心消息延迟问题的话，此时限制每个 Broker 节点的 Partition 数量是一个非常不错的主意：对于 N 个 Broker 节点和复制副本因子「replication-factor」为 F 的 Kafka 集群，那么整个

Kafka 集群的 Partition 数量最好不超过 「**100 \* N \* F**」 个，即单个 Broker 节点 Partition 的 Leader 数量不超过100。



#### 高可用性方面分析 04

我们知道 Kafka 是通过多副本复制技术来实现集群的高可用和稳定性的。每个 Partition 都会有多个数据副本，每个副本分别存在于不同的 Broker 上。所有的数据副本中，其中一个数据副本为 Leader，其他的数据副本为 Follower。

在Kafka集群内部，所有的数据副本采用自动化的方式管理且会确保所有副本之间的数据是保持同步状态的。当 Broker 发生故障时，对于 Leader 副本所在 Broker 的所有 Partition 将会变得暂不可用。Kafka 将自动在其它副本中选择出一个 Leader，用于接收客户端的请求。这个过程由 Kafka Controller 节点 Broker 自动选举完成。

正常情况下，当一个 Broker 在有计划地停止服务时候，那么 Controller 会在服务停止之前，将该 Broker上的所有 Leader 副本一个个地移走。对于单个 Leader 副本的移动速度非常快，从客户层面看，有计划的服务停服只会导致系统很短时间窗口不可用。

但是，当 Broker 不是正常停止服务时「**kill -9 强杀方式**」，系统的不可用时间窗口将会与受影响的 Partition 数量有关。如果此时发生宕机的 Broker 是 Controller 节点时，这时 Controller 节点故障恢复会自动的进行，但是新的 Controller 节点需要从 Zookeeper 中读取每一个 Partition 的元数据信息用于初始化数据。假设一个

Kafka 集群存在10000个 Partition，从 Zookeeper 中恢复元数据时每个 Partition 大约花费2ms，则 Controller 恢复将会增加约20秒的不可用时间窗口。

总之，通常情况下 Kafka 集群中越多的 Partition 会带来越高的吞吐量。但是，如果 Kafka 集群中 Partition 总量过大或者单个 Broker 节点 Partition 过多，都可能会对系统的可用性和消息延迟带来潜在的负面影响，需要引起我们的重视。

## 如何保证 Kafka 中的消息是有序的？

我们知道在 Kafka 中，并不保证消息全局有序，但是可以**保证分区有序性**，分区与分区之间是无序的。**那么如何保证 Kafka 中的消息是有序的呢？** 可以从以下三个方面来入手分析：

### 生产端 Producer

01

在第4道题「**生产者有哪些发送模式**」的最后的场景分析里面简单的说明了下，这里再详细的进行分析下：

首先 Kafka 的 Producer 端发送消息，如果是不对默认参数进行任何设置且网络没有抖动的情况下，消息是可以一批批的按消息发送的顺序被发送到 Kafka Broker 端。但是，一旦有网络波动了，则消息就可能出现乱序。

所以，要严格保证 Kafka 发消息有序，首先要考虑用同步的方式来发送消息，两种同步发送的方式如下：

- 1) 设置消息响应参数 `acks = all & max.in.flight.requests.per.connection = 1`：发送端将会在一条消息发出后，响应必须满足 `acks` 设置的参数后，才会发送下一条消息。虽然在使用时还是异步发送的方式，其实底层已经是一条接一条的发送了。
- 2) **Sync发送方式**：当调用 KafkaProducer 的 `send()` 后，返回的 Future 对象的 `get` 方式阻塞等待结果。根据返回的结果可以判断是否发送成功，**由于是同步发送会阻塞，只有当消息通过 `get()` 返回数据时，才会继续下一条消息的发送。**

通过上面方式还可能出现消息重发和幂等问题：

- 1) **重发问题**：Kafka 在消息发送出现问题时，通过判断是否可以自动重试恢复，如果是可以自动恢复的问题，设置 `retries > 0`，让 Kafka 自动重试。
- 2) **幂等问题**：Kafka 1.0 之后的版本，Producer 端引入了幂等特性。设置 `enable.idempotence = true`，幂等特性可以给消息添加序列号，即每次发送会把序列号递增 1。**开启了 Kafka Producer 端的幂等特性后，我们就可以通过设置参数 `max.in.flight.requests.per.connection = 5` 「默认值」**，这样当 Kafka 发消息的时候，由于消息有了序列

号当发送消息出现错误的时候，Kafka 底层会通过获取服务器端的最近几条日志的序列号和发送端需要重新发送的消息序列号做对比，如果是连续的，那么就可以继续发送消息，保证消息顺序。

## 服务端 Broker 02

在 Kafka 中，Topic 只是一个逻辑上的概念，而组成 Topic 的分区 Partition 才是真正存消息的地方。

Kafka 只保证单分区内的消息是有序的，所以如果要保证业务全局严格有序，就要设置 Topic 为单分区的方式。不过对业务来说一般不需要考虑全局有序的，只需要保证业务中不同类别的消息有序即可。

但是这里有个必须要受到重视的问题，就是当我们对分区 Partition 进行数量改变的时候，由于是简单的 Hash 算法会把以前可能分到相同分区的消息分到别的分区上。这样就不能保证消息顺序了。面对这种情况，就需要在动态变更分区的时候，考虑对业务的影响。有可能需要根据业务和当前分区需求，重新划分消息类别。

## 消费端 Consumer 03

在 Consumer 端，根据 Kafka 的模型，一个 Topic 下的每个分区只能从属于这个 Topic 的消费者组中的某一个消费者。

当消息被发送分配到同一个 Partition 中，消费者从 Partition 中取出来数据的时候，也一定是有顺序的，没有错乱。

但是消费者可能会有多个线程来并发来消费消息。如果单线程消费数据，吞吐量太低了，而多个线程并发消费的话，顺序可能就乱掉了。

此时可以通过写多个内存队列，将相同 key 的消息都写入同一个队列，然后对于多个线程，每个线程分别消息一个队列即可保证消息顺序。

## Kafka 为什么不支持读写分离呢？

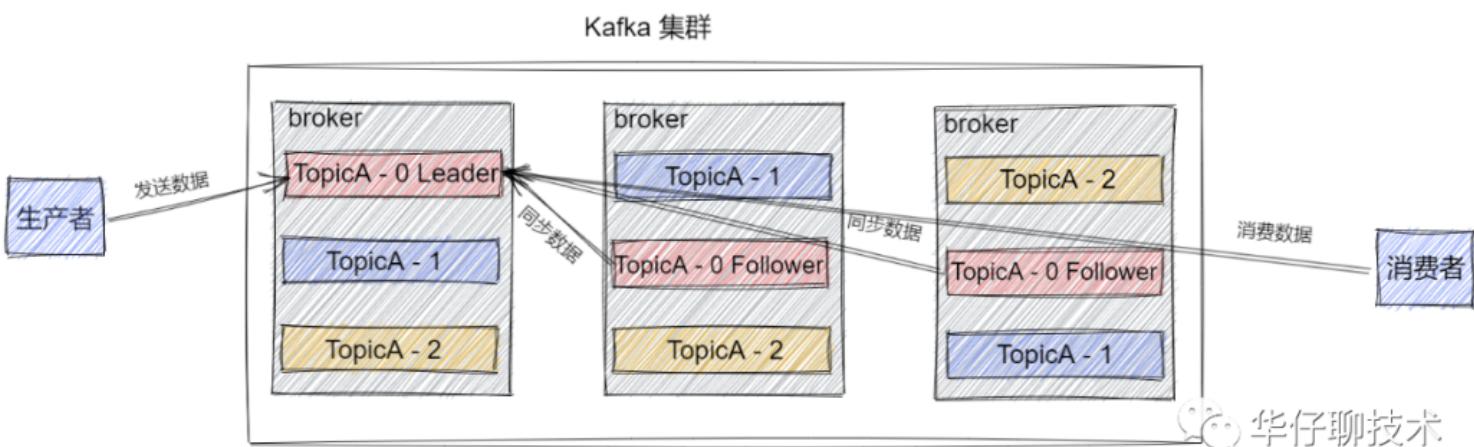
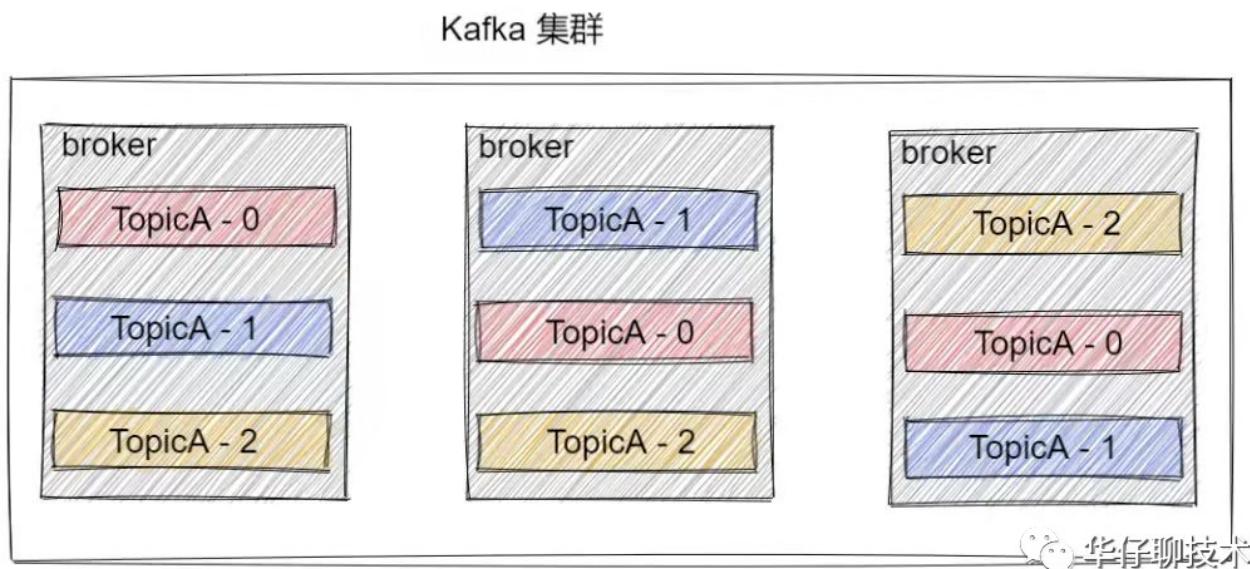
在很多主从模型系统中，是允许从节点可以对外提供读服务的，只不过 Kafka 当初为了避免数据不一致的问题，而采用了通过主节点来统一提供服务的方式。

不支持读写分离的原因有2点：

- 1) **场景不一致**: 读写分离架构适用于那种读操作负载很大, 但写操作相对不频繁的场景, 但是 Kafka 显然不适合这种场景。
- 2) **延迟问题**: Kafka 通过 PULL 方式来实现数据同步, 因此 Leader 副本和 Follower 副本存在数据不一致的情况, 如果允许 Follower 副本提供读服务的话, 就会带来消息滞后的问题。

## Kafka 副本有哪两种, 作用是什么?

在 Kafka 中, 为实现「**数据备份**」的功能, 保证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 仍然能够继续工作, **为此 Kafka 提供了副本机制, 一个 Topic 的每个 Partition 都有若干个副本, 一个 Leader 副本和若干个 Follower 副本**。



1) Leader 主副本负责对外提供读写数据服务。

2) Follower 从副本只负责和 Leader 副本保持数据同步，并不对外提供任何服务。

## Kafka 能否手动删除消息？

首先 Kafka 是支持手动删除消息的，当然它本身提供了消息留存策略，能够自动删除过期的消息。

Kafka 将消息存储到磁盘中，随着写入数据不断增加，磁盘占用空间越来越大，为了控制占用空间就需要对消息做一定的清理操作。Kafka 存储日志结构分析中每一个分区副本（Replica）都对应一个 Log，而 Log 又可以分为多个日志分段（LogSegment），这样就便于 Kafka 对日志的清理操作。

- 1) 普通消息：我们可以使用 Kafka-delete-records 命令或者通过程序调用 Admin.deleteRecords 方法来删除消息。两者底层都是调用 Admin 的 deleteRecords 的方法，通过将分区的 LEO 值抬高来间接删除消息。
- 2) 设置key且参数 cleanup.policy=delete/campact 的消息：可以依靠 Log Cleaner 组件提供的功能删除该 Key 的消息。

**日志删除（Log Retention）**：按照一定的保留策略直接删除不符合条件的日志分段（LogSegment）。

**日志压缩（Log Compaction）**：针对每个消息的key进行整合，对于有相同key的不同value值，只保留最后一个版本。

### 日志删除 01

Kafka 的日志管理器（LogManager）中有一个专门的日志清理任务通过周期性检测和删除不符合条件的日志分段文件（LogSegment），这里我们可以通过设置 Kafka Broker 端的参数

「**log.retention.check.interval.ms**」，默认值为300000，即5分钟。

在 Kafka 中一共有3种保留策略：

#### 基于时间策略

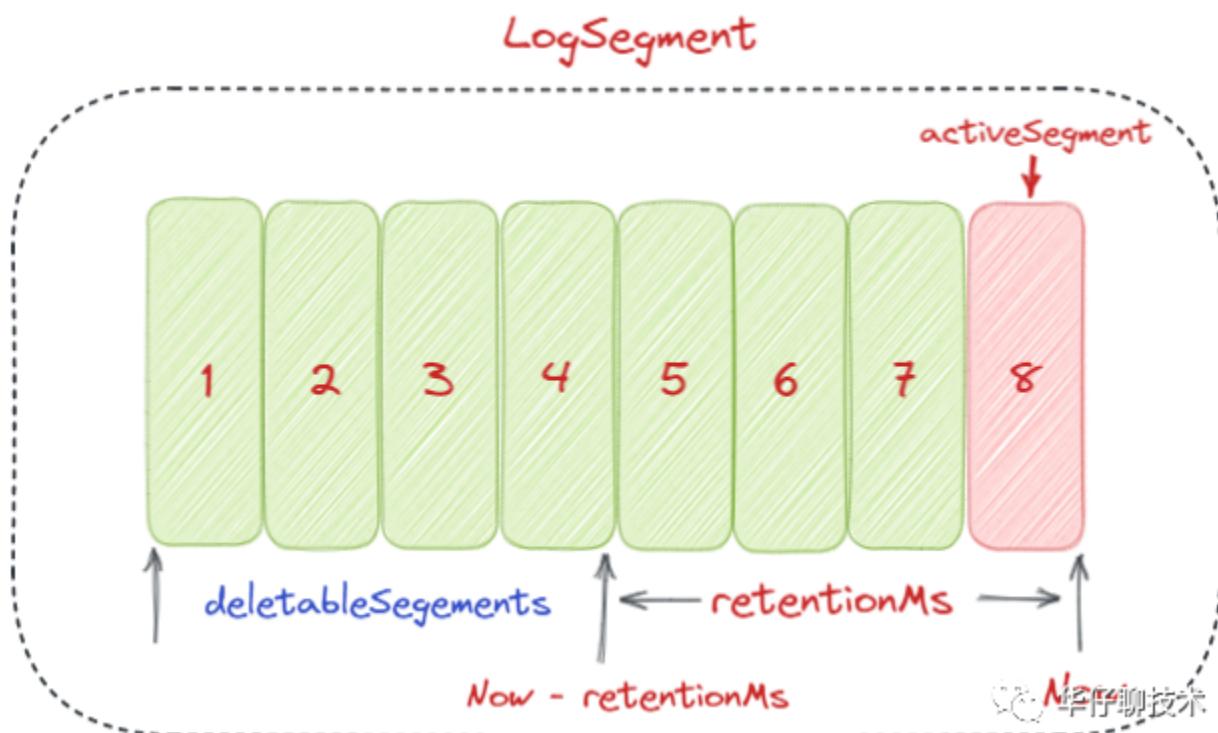
日志删除任务会周期检查当前日志文件中是否有保留时间超过设定的阈值(**retentionMs**) 来寻找可删除的日志段文件集合(**deletableSegments**)。

其中 **retentionMs** 可以通过 Kafka Broker 端的这几个参数的大小判断的  
log.retention.ms > log.retention.minutes > log.retention.hours 优先级来设置，**默认情况只会配置 log.retention.hours 参数，值为168即为7天。**

这里需要注意：删除过期的日志段文件，并不是简单的根据该日志段文件的修改时间计算的，而是要根据该日志段中最大的时间戳 largestTimeStamp 来计算的，首先要查询该日志分段所对应的时间戳索引文件，查找该时间戳索引文件的最后一条索引数据，如果时间戳值大于0，则取值，否则才会使用最近修改时间 (lastModifiedTime) 。

#### 【删除步骤】：

1. 首先从 Log 对象所维护的日志段的跳跃表中移除要删除的日志段，用来确保已经没有线程来读取这些日志段。
2. 将日志段所对应的所有文件，包括索引文件都添加上".deleted"的后缀。
3. 最后交给一个以"delete-file"命名的延迟任务来删除这些以" .deleted "为后缀的文件。默认1分钟执行一次，可以通过 file.delete.delay.ms 来配置。



#### 基于日志大小策略

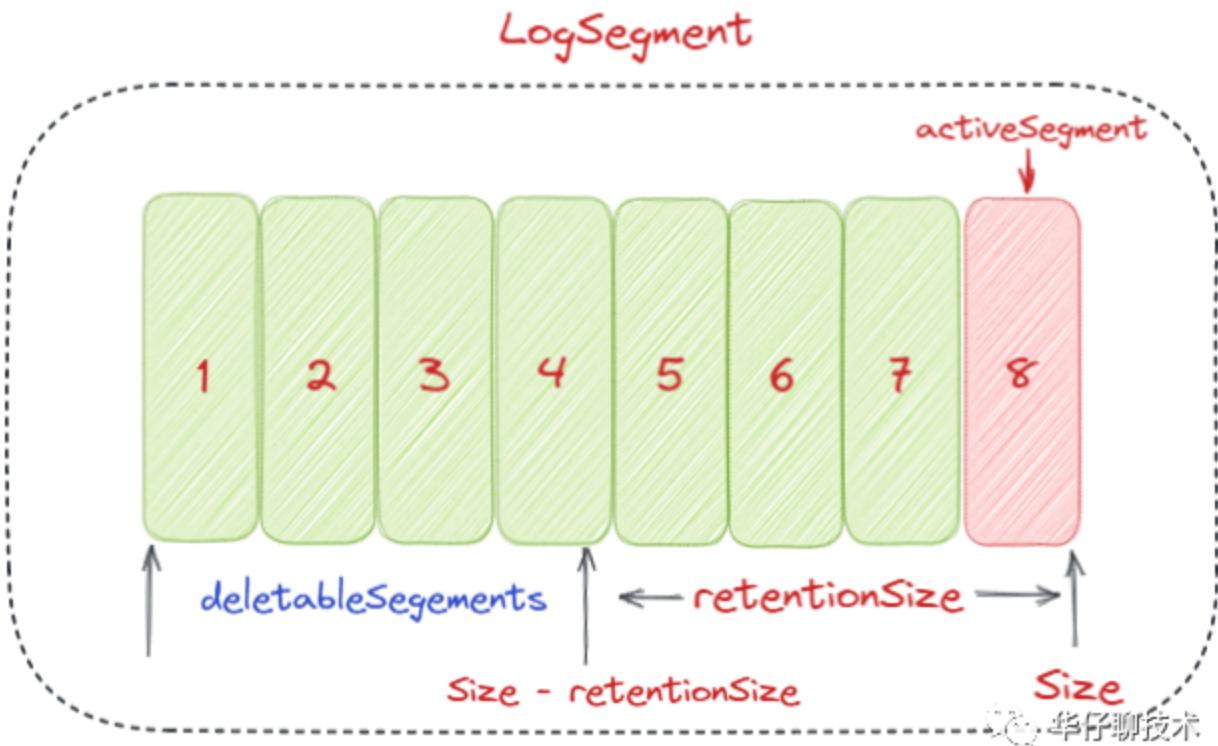
日志删除任务会周期检查当前日志大小是否超过设定的阈值(**retentionSize**) 来寻找可删除的日志段文件集合 (**deletableSegments**)。

其中**retentionSize**这里我们可以通过 Kafka Broker 端的参数log.retention.bytes 来设置， 默认值为-1，即无穷大。

这里需要注意的是 log.retention.bytes 设置的是Log中所有日志文件的大小，而不是单个日志段的大小。单个日志段可以通过参数 log.segment.bytes 来设置，默认大小为1G。

### 【删除步骤】：

1. 首先计算日志文件的总大小Size和 retentionSize 的差值，即需要删除的日志总大小。
2. 然后从日志文件中的第一个日志段开始进行查找可删除的日志段的文件集合(deletableSegments)
3. 找到后就可以进行删除操作了。



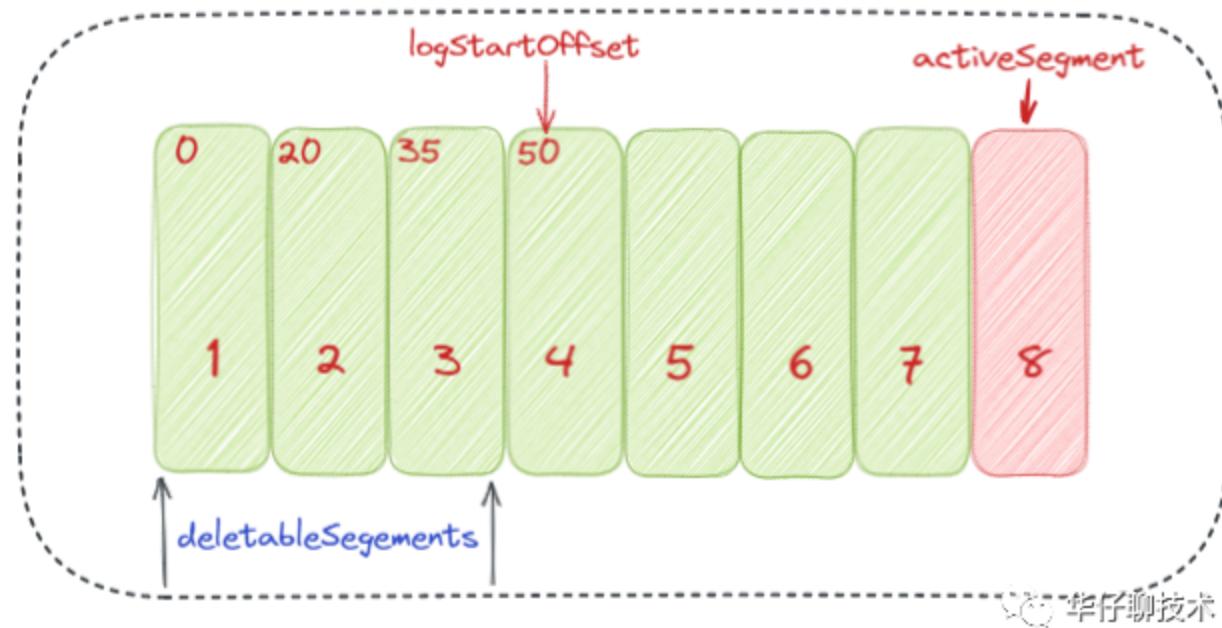
### 基于日志起始偏移量

该策略判断依据是日志段的下一个日志段的起始偏移量 baseOffset 是否小于等于 logStartOffset，如果是，则可以删除此日志分段。

### 【如下图所示 删除步骤】：

1. 首先从头开始遍历每个日志段，日志段 1 的下一个日志分段的起始偏移量为 20，小于 logStartOffset 的大小，将日志段1加入deletableSegments。
2. 日志段2的下一个日志偏移量的起始偏移量为 35，也小于 logStartOffset 的大小，将日志分段2页加入deletableSegments。
3. 日志段3的下一个日志偏移量的起始偏移量为 50，也小于 logStartOffset 的大小，将日志分段3页加入deletableSegments。
4. 日志段4的下一个日志偏移量通过对比后，在 logStartOffset 的右侧，那么从日志段4开始的所有日志段都不会加入 deletableSegments。
5. 待收集完所有的可删除的日志集合后就可以直接删除了。

## LogSegment

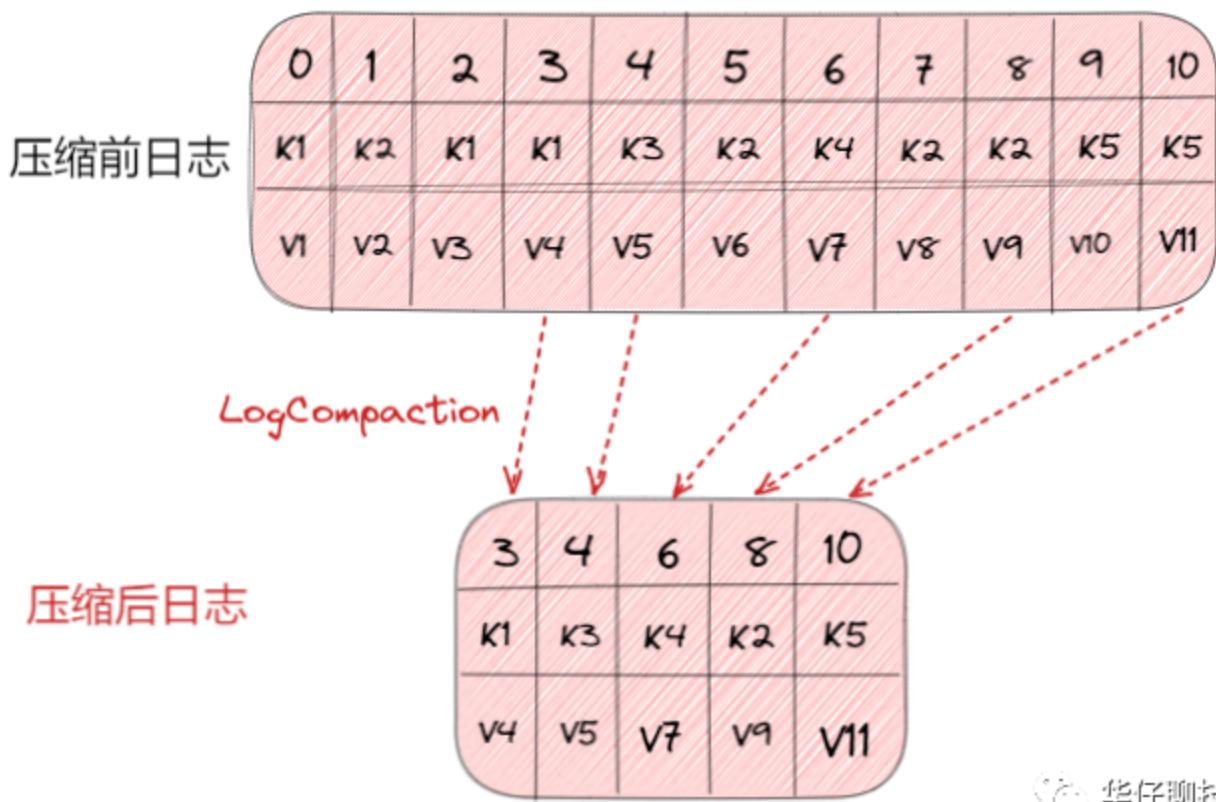


© 华仔聊技术

## 日志压缩 02

**日志压缩 Log Compaction** 对于有相同key的不同value值，只保留最后一个版本。如果应用只关心 key 对应的最新 value 值，则可以开启 Kafka 相应的日志清理功能，Kafka 会定期将相同 key 的消息进行合并，只保留最新的 value 值。

Log Compaction 可以类比 Redis 中的 RDB 的持久化模式。我们可以想象下，如果每次消息变更都存 Kafka，在某一时刻，Kafka 异常崩溃后，如果想快速恢复，可以直接使用日志压缩策略，这样在恢复的时候只需要恢复最新的数据即可，这样可以加快恢复速度。

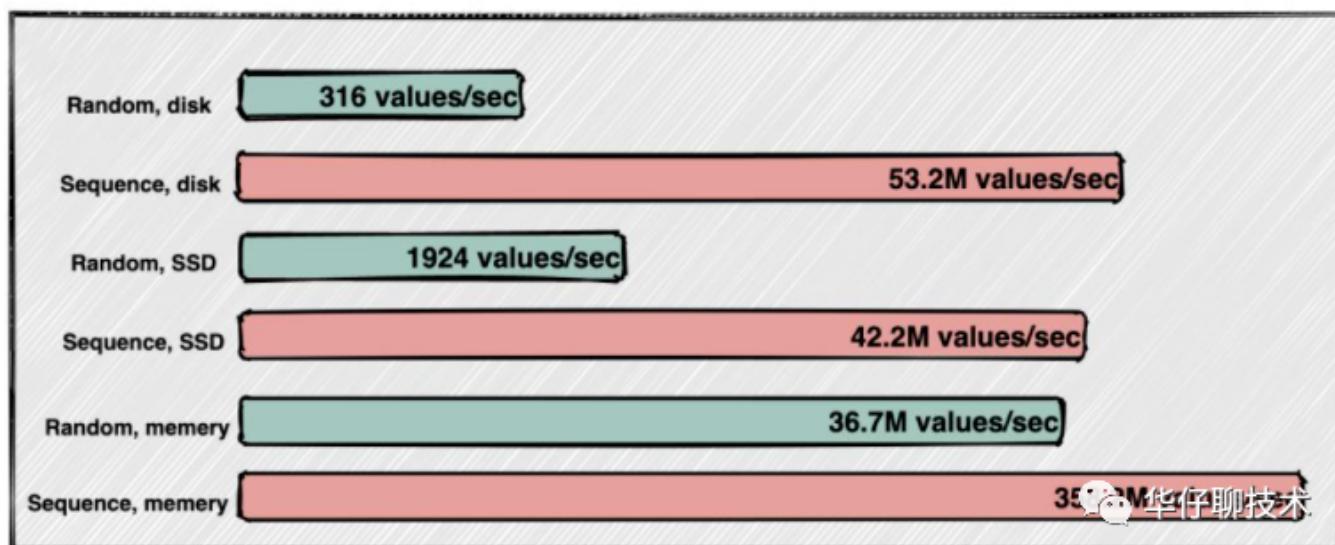


华仔聊技术

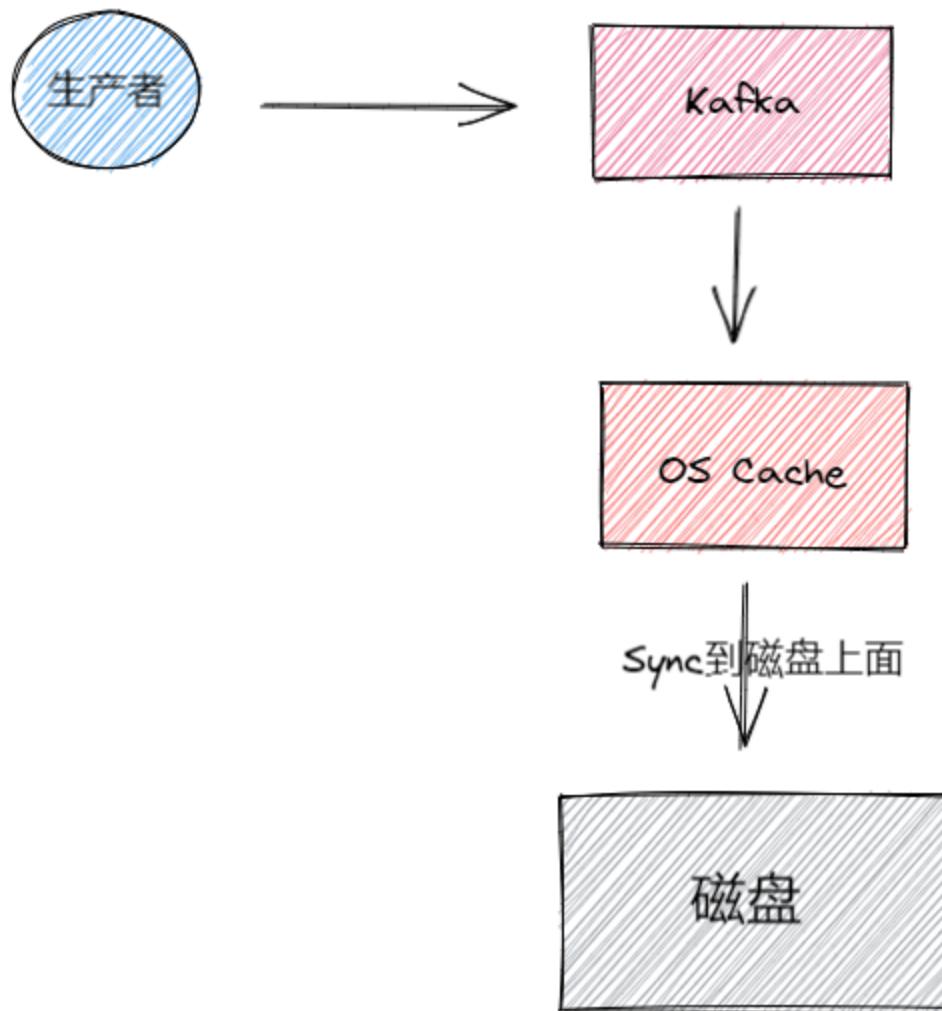
## Kafka 读写数据这么快是如何做到的？

### 顺序追加写 01

kafka 在写数据时是以「**磁盘顺序写**」的方式来进行落盘的，即将数据追加到文件的末尾。对于普通机械磁盘，如果是随机写的话，涉及到磁盘寻址的问题，导致性能极低，**但是如果只是按照顺序的方式追加文件末尾的话，这种磁盘顺序写的性能基本可以跟写内存的性能差不多的**。下图所示普通机械磁盘的顺序I/O性能指标是53.2M values/s。

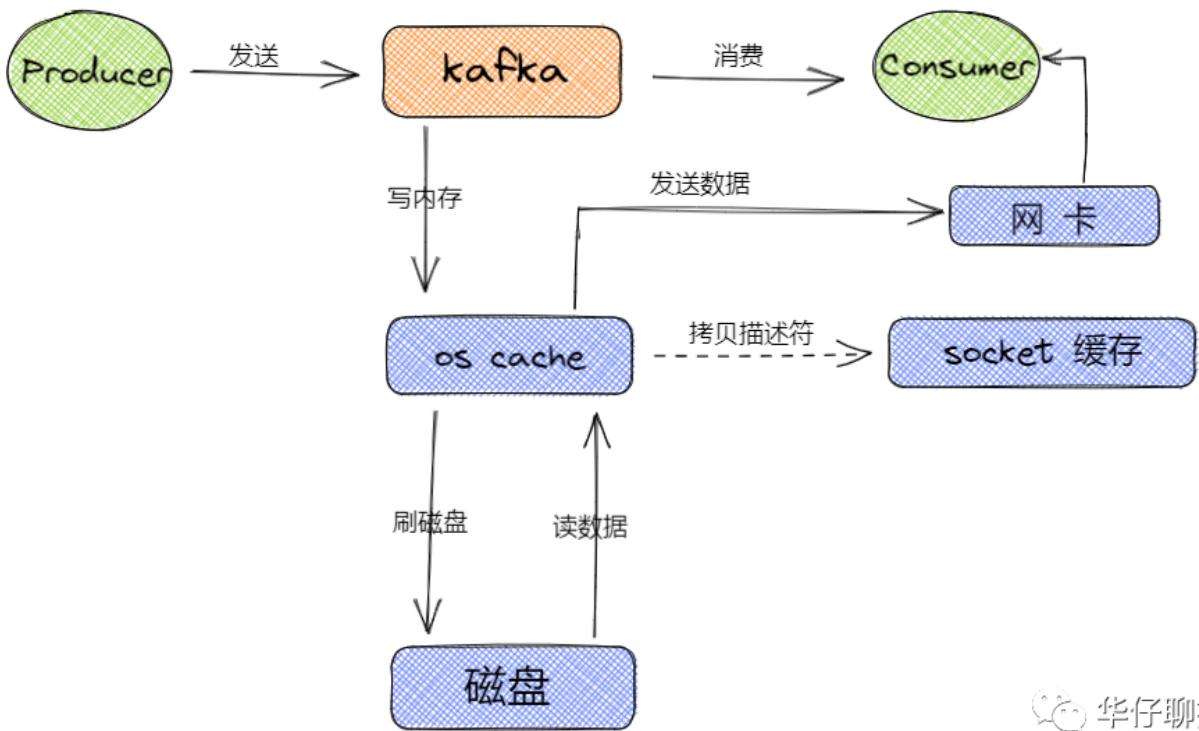


首先 Kafka 为了保证磁盘写入性能，通过 mmap 内存映射的方式利用操作系统的 Page Cache 异步写入。也可以称为 os cache，意思就是操作系统自己管理的缓存。那么在写磁盘文件的时候，就可以先直接写入 os cache 中，接下来由操作系统自己决定什么时候把 os cache 里的数据真正刷入到磁盘中，这样大大提高写入效率和性能。如下图所示：



华仔聊技术

Kafka 为了解决内核态和用户态数据不必要 Copy 这个问题，在读取数据的时候就引入了「零拷贝技术」。即让操作系统的 os cache 中的数据直接发送到网卡后传出给下游的消费者，中间跳过了两次拷贝数据的步骤，从而减少拷贝的 CPU 开销，减少用户态内核态的上下文切换次数，从而优化数据传输的性能，而 Socket 缓存中仅仅会拷贝一个描述符过去，不会拷贝数据到 Socket 缓存，如下图所示：



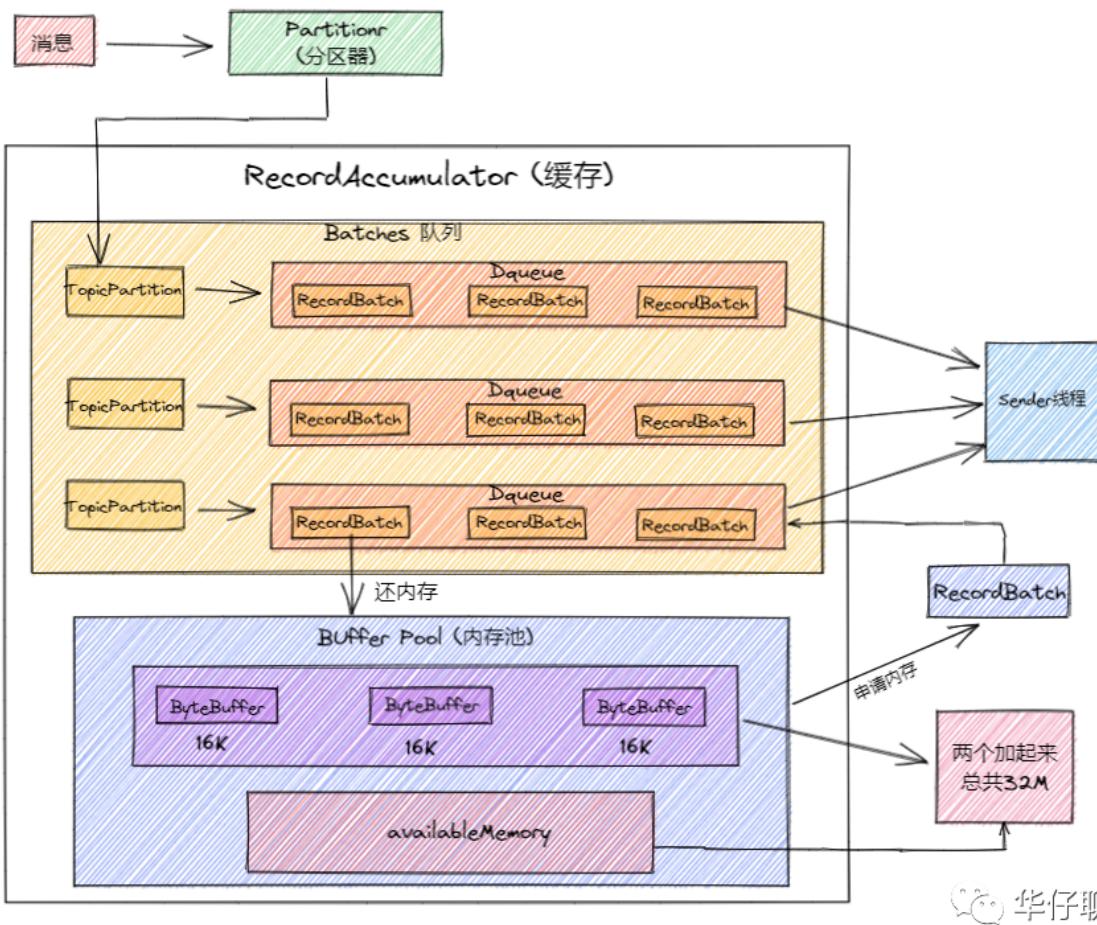
华仔聊技术

在 Kafka 中主要有以下两个地方使用到了「零拷贝技术」：

- 1) 基于 mmap 机制实现的索引文件：首先索引文件都是基于 **MappedByBuffer** 实现，即让用户态和内核态来共享内核态的数据缓冲区，此时数据不需要 Copy 到用户态空间。虽然 mmap 避免了不必要的 Copy，但是在不同操作系统下，其创建和销毁成功是不一样的，不一定都能保证高性能。所以在 Kafka 中只有索引文件使用了 mmap。
- 2) 基于 **sendfile** 机制实现的日志文件读写：在 Kafka 传输层接口中有个 **TransportLayer** 接口，它的实现类中有使用了 Java **FileChannel** 中 **transferTo** 方法。该方法底层就是使用 **sendfile** 实现的零拷贝机制，目前只是在 I/O 通道是普通的 **PLAINTEXT** 的时候才会使用到零拷贝机制。

### 消息批量发送 04

Kafka 在发送消息的时候并不是一条条的发送的，而是会**把多条消息合并成一个批次Batch 进行处理发送**，消费消息也是同样，一次拉取一批次的消息进行消费。如下图所示：



华仔聊技术

## 数据压缩 05

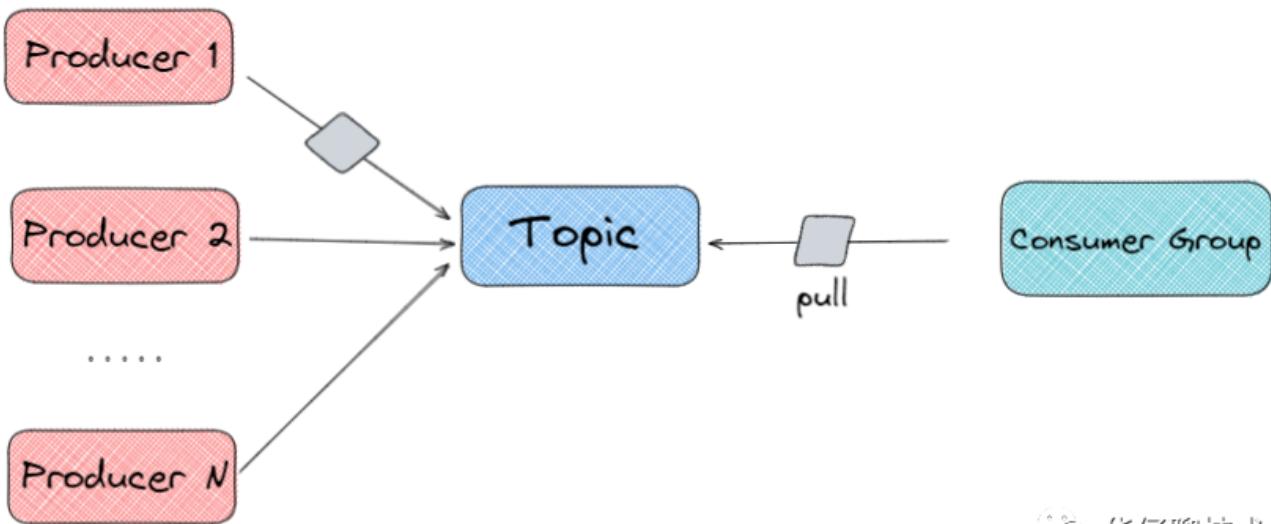
在 Kafka 中三个端都使用了优化后的压缩算法，**压缩有助于提高吞吐量，降低延迟并提高磁盘利用率**。Kafka 底层支持多种压缩算法: **Iz4, snappy, gzip**，从 Kafka 2.1.0 开始新增了 **ZStandard** 算法，该算法是 Facebook 开源的压缩算法，能提供超高的压缩比。

在 Kafka 中，压缩可能会发生在两个地方：**生产者端和Broker端**。一句话总结下压缩和解压缩，即 **Producer 端压缩，Broker 端保持，Consumer 端解压缩**，这样可以节省大量的网络和磁盘开销。

## Kafka 消费模型有哪些？

一般情况下消息消费总共有两种模式：「**推模型**」和「**拉模型**」。在 Kafka 中的消费模型是属于「**拉模型**」，此模式的消息消费方式实现有两种：「**点对点方式**」和「**发布订阅方式**」。

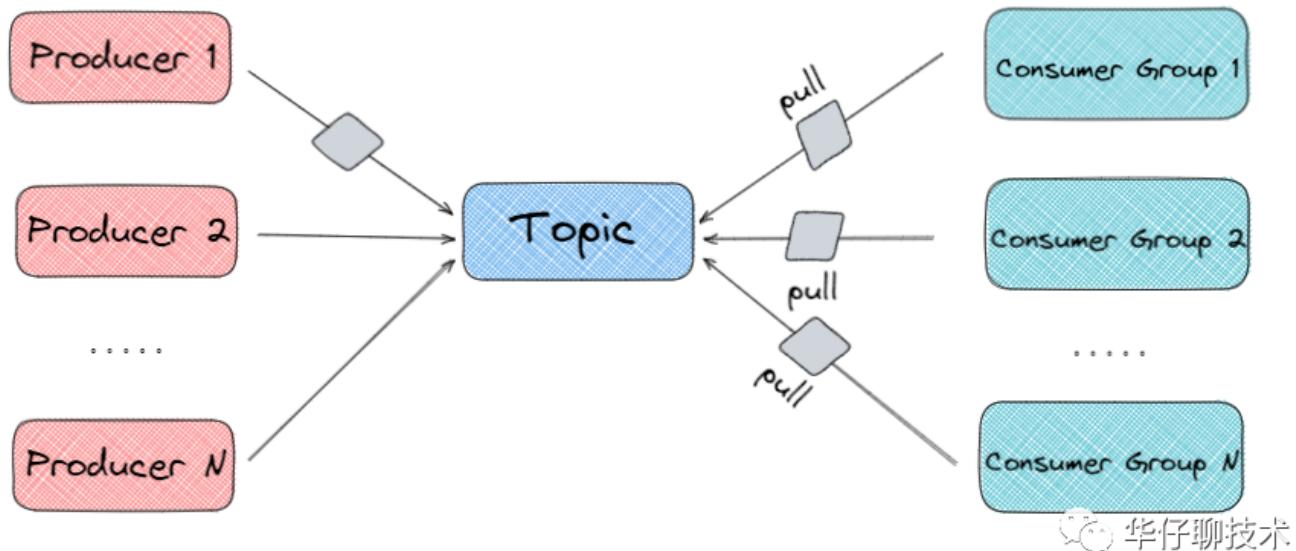
## 点对点方式 01



华仔聊技术

**点对点方式：**假如所有消费者都同属于一个消费组的话，此时所有的消息都会被分配给每一个消费者，**但是消息只会被其中一个消费者进行消费。**

## 发布订阅方式 02



华仔聊技术

**发布订阅：**假如所有消费者属于不同的消费组，此时所有的消息都会被分配给每一个消费者，**每个消费者都会收到该消息。**

什么是消费者组,有什么作用?

首先我来看看什么是「**消费者组**」：

消费者组 Consumer Group，顾名思义就是由多个 Consumer 组成，且拥有一个公共且唯一的 Group ID。组内每个消费者负责消费不同分区的数据，**但一个分区只能由一个组内消费者消费，消费者组之间互不影响。**

## 为什么 Kafka 要设计 Consumer Group，只有 Consumer 不可以吗？

我们知道 Kafka 是一款高吞吐量，低延迟，高并发，高可扩展的消息队列产品，那么如果某个 Topic 拥有数百万到数千万的数据量，仅仅依靠 Consumer 进程消费，消费速度可想而知，**所以需要一个扩展性较好的机制来保障消费进度，这个时候 Consumer Group 应运而生，Consumer Group 是 Kafka 提供的可扩展且具有容错性的消费者机制。**

Kafka Consumer Group 特点如下：

- 1) 每个 Consumer Group 有一个或者多个 Consumer。
- 2) 每个 Consumer Group 拥有一个公共且唯一的 Group ID。
- 3) Consumer Group 在消费 Topic 的时候，Topic 的每个 Partition 只能分配给组内的某个 Consumer，只要被任何 Consumer 消费一次，那么这条数据就可以认为被当前 Consumer Group 消费成功。

## Kafka 中 Offset 的作用是什么，如何进行维护？

在 Kafka 中每个 Topic 分区下面的每条消息都被赋予了一个唯一的ID值，用来标识它在分区中的位置。这个ID值就被称为位移「**Offset**」或者叫**偏移量**，一旦消息被写入到日志分区中，它的位移值将不能被修改。

### 位移 Offset 管理方式 01

Kafka 旧版本（0.9版本之前）是把位移保存在 ZooKeeper 中，减少 Broker 端状态存储开销。

鉴于 Zookeeper 不适合频繁写更新，而 Consumer Group 的位移提交又是高频写操作，这样会拖慢 ZooKeeper 集群的性能，于是在新版 Kafka 中，社区采用了将位移保存在 Kafka 内部「**Kafka Topic 天然支持高频写且持久化**」，这就是所谓大名鼎鼎的\_\_consumer\_offsets。

**\_\_consumer\_offsets**：用来保存 Kafka Consumer 提交的位移信息，另外它是由 Kafka 自动创建的，和普通的 Topic 相同，它的消息格式也是 Kafka 自己定义的，我们无法进行修改。

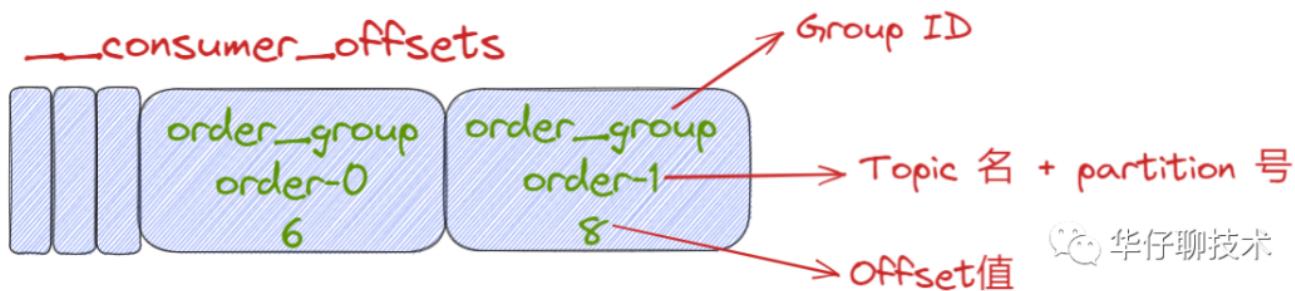
## \_\_consumer\_offsets 有3种消息格式：

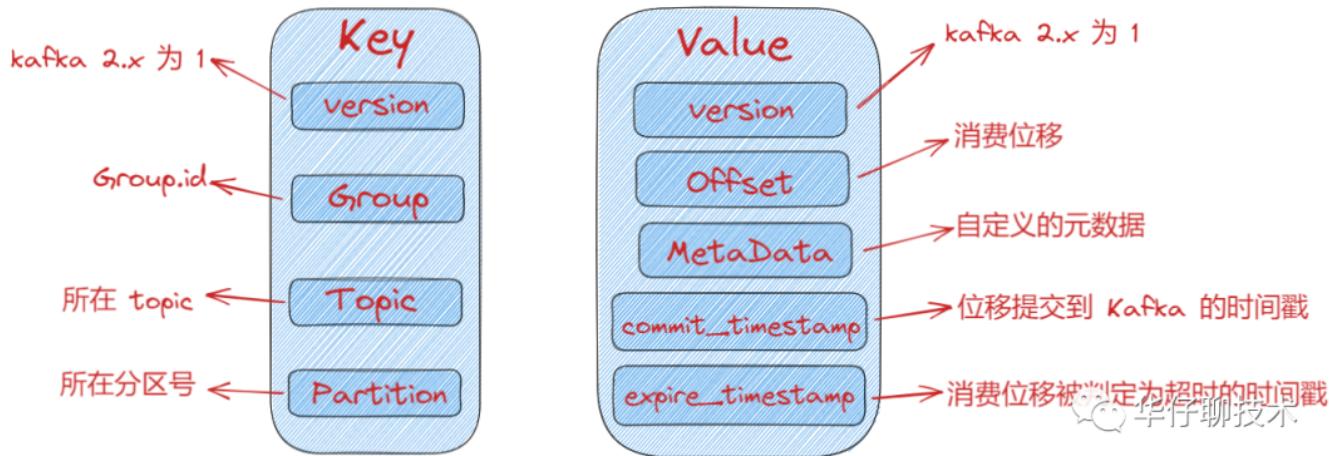
- 1) 用来保存 Consumer Group 信息的消息。
- 2) 用来删除 Group 过期位移甚至是删除 Group 的消息，也可以称为 tombstone 消息，即墓碑消息，它的主要特点是空消息体，一旦某个 Consumer Group 下的所有 Consumer 位移数据都已被删除时，Kafka会向 \_\_consumer\_offsets 主题的对应分区写入 tombstone 消息，表明要彻底删除这个 Group 的信息。
- 3) 用来保存位移值。

## \_\_consumer\_offsets 消息格式分析揭秘：

- 1) 消息格式我们可以简单理解为是一个 KV 对。Key 和 Value 分别表示消息的键值和消息体。
- 2) 那么 Key 存什么呢？既然是存储 Consumer 的位移信息，在 Kafka 中，Consumer 数量会很多，必须有字段来标识位移数据是属于哪个 Consumer 的，怎么来标识 Consumer 字段呢？我们知道 Consumer Group 会共享一个公共且唯一的 Group ID，那么只保存它就可以了吗？我们知道 Consumer 提交位移是在分区的维度进行的，很显然，key 中还应该保存 Consumer 要提交位移的分区。
- 3) 总结：位移主题的 Key 中应该保存 3 部分内容：**<Group ID, 主题名, 分区号>**
- 4) value 可以简单认为存储的是 offset 值，当然底层还存储其他一些元数据，帮助 Kafka 来完成一些其他操作，比如删除过期位移数据等。

## \_\_consumer\_offsets 消息格式示意图：





## \_\_consumer\_offsets 创建 02

`__consumer_offsets` 是怎么被创建出来的呢？当 Kafka 集群中的第一个 Consumer 启动时，Kafka 会自动创建`__consumer_offsets`。

它就是普通的 Topic，也有对应的分区数，如果由 Kafka 自动创建的，那么分区数又是怎么设置的呢？

这个依赖 Broker 端参数主题分区位移个数即「`offsets.topic.num.partitions`」默认值是50，因此 Kafka 会自动创建一个有 50 个分区的 `__consumer_offsets`。既然有分区数，必然就会有分区对应的副本个数，这个是依赖Broker 端另外一个参数来完成的，即 「`offsets.topic.replication.factor`」默认值为3。

总结一下，`__consumer_offsets` 由 Kafka 自动创建的，那么该 Topic 的分区数是 50，副本数是 3，而具体 Consumer Group 的消费情况要存储到哪个 Partition，根据`abs(groupId.hashCode()) % NumPartitions` 来计算的，这样就可以保证 Consumer Offset 信息与 Consumer Group 对应的 Coordinator 处于同一个 Broker 节点上。如下图所示：



wangjianghua@DESKTOP-F37NME5:/tmp/kafka-logs\$ tree

```
|- consumer_offsets-0
|   |- 00000000000000000000000000000000.index
|   |- 00000000000000000000000000000000.log
|   |- 00000000000000000000000000000000.timeindex
|   |- leader-epoch-checkpoint
|- consumer_offsets-1
|   |- 00000000000000000000000000000000.index
|   |- 00000000000000000000000000000000.log
|   |- 00000000000000000000000000000000.timeindex
|   |- leader-epoch-checkpoint
|- consumer_offsets-10
|   |- 00000000000000000000000000000000.index
|   |- 00000000000000000000000000000000.log
|   |- 00000000000000000000000000000000.timeindex
|   |- leader-epoch-checkpoint
|- consumer_offsets-11
|   |- 00000000000000000000000000000000.index
|   |- 00000000000000000000000000000000.log
|   |- 00000000000000000000000000000000.timeindex
|   |- leader-epoch-checkpoint
|- consumer_offsets-12
|   |- 00000000000000000000000000000000.index
|   |- 00000000000000000000000000000000.log
|   |- 00000000000000000000000000000000.timeindex
|   |- leader-epoch-checkpoint
|- consumer_offsets-13
|   |- 00000000000000000000000000000000.index
|   |- 00000000000000000000000000000000.log
|   |- 00000000000000000000000000000000.timeindex
|   |- leader-epoch-checkpoint
|- consumer_offsets-14
```

华仔聊技术

如果我的文章对你有所帮助，还请帮忙点赞、在看、转发一下，非常感谢！

坚持总结，持续输出高质量文章 关注我：华仔聊技术



微信搜一搜

华仔聊技术

华仔聊技术



# 华仔聊技术

聊聊后端技术架构以及中间件源码

22篇原创内容

公众号

点个“赞”和“在看”鼓励一下嘛~

收录于话题 #消息队列 14

上一篇

【建议收藏】Kafka 面试连环炮，看看你能撑到哪一步？（中）

下一篇

源码系列第1弹 | 带你快速攻略Kafka源码之旅入门篇

文章已于2022-03-11修改

喜欢此内容的人还喜欢

kafka的数据结构和算法

Anryg是码农



简单计算机知识小科普——什么是数据仓库？

数据小钟



用户研究：用户研究和数据分析的根本联系与区别

AYGNIX

