

Kafka夺命连环15问，你能扛到几问？

苏三说技术 2022-03-27 18:40

以下文章来源于华仔聊技术，作者王江华



华仔聊技术

聊聊后端技术架构以及中间件源码



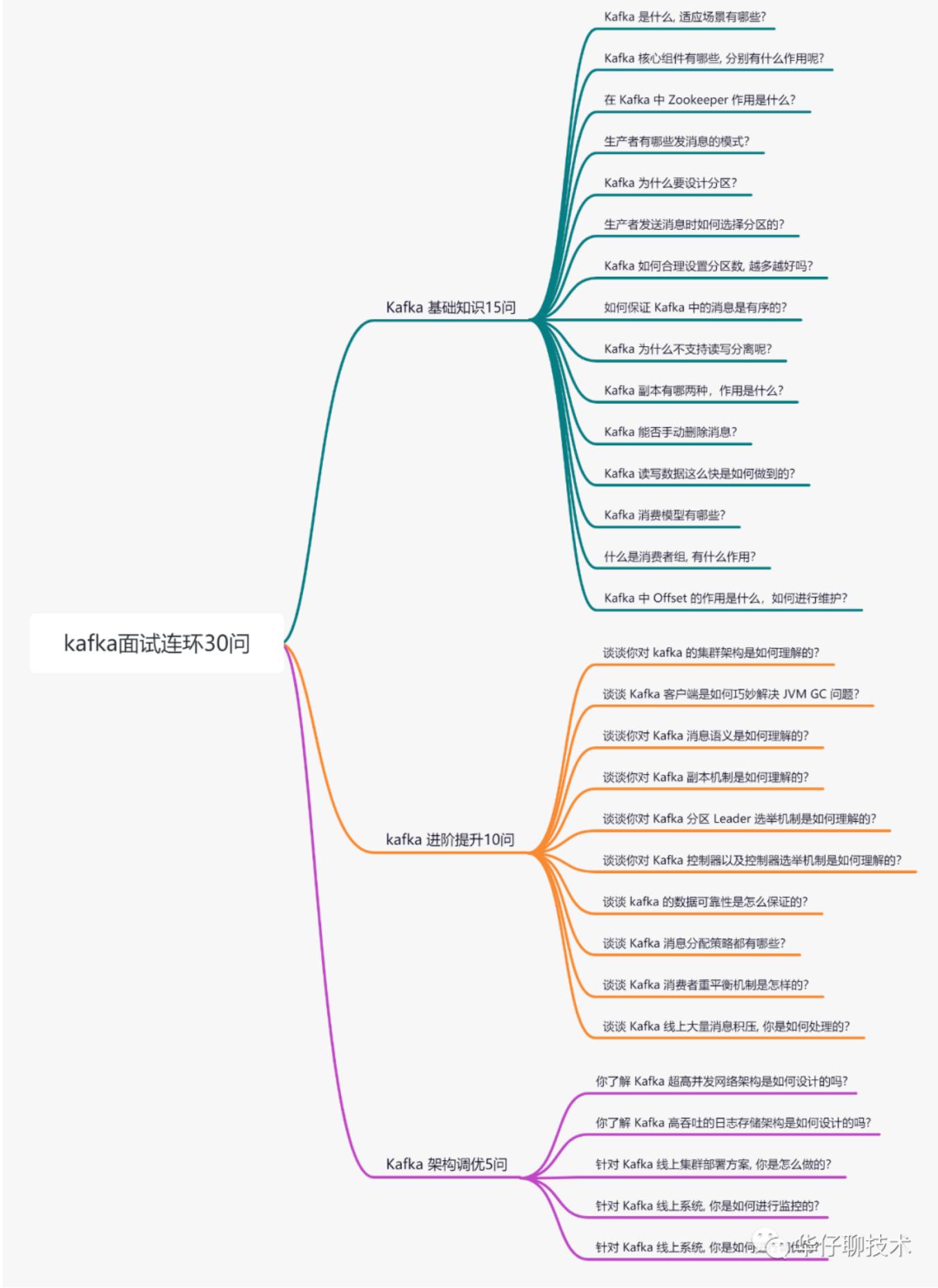
大家好，我是苏三，又跟大家见面了。

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，本文是面试系列的中篇。

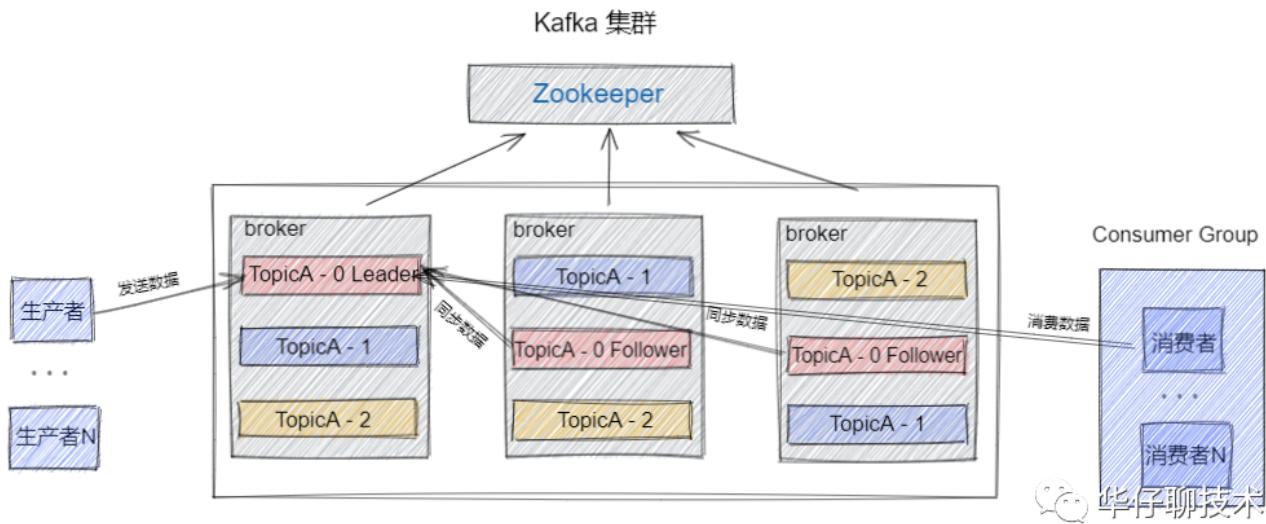
这篇文章干货很多，希望你可以耐心读完。



02 kafka 进阶提升10问

谈谈你对 kafka 的集群架构是如何理解的？

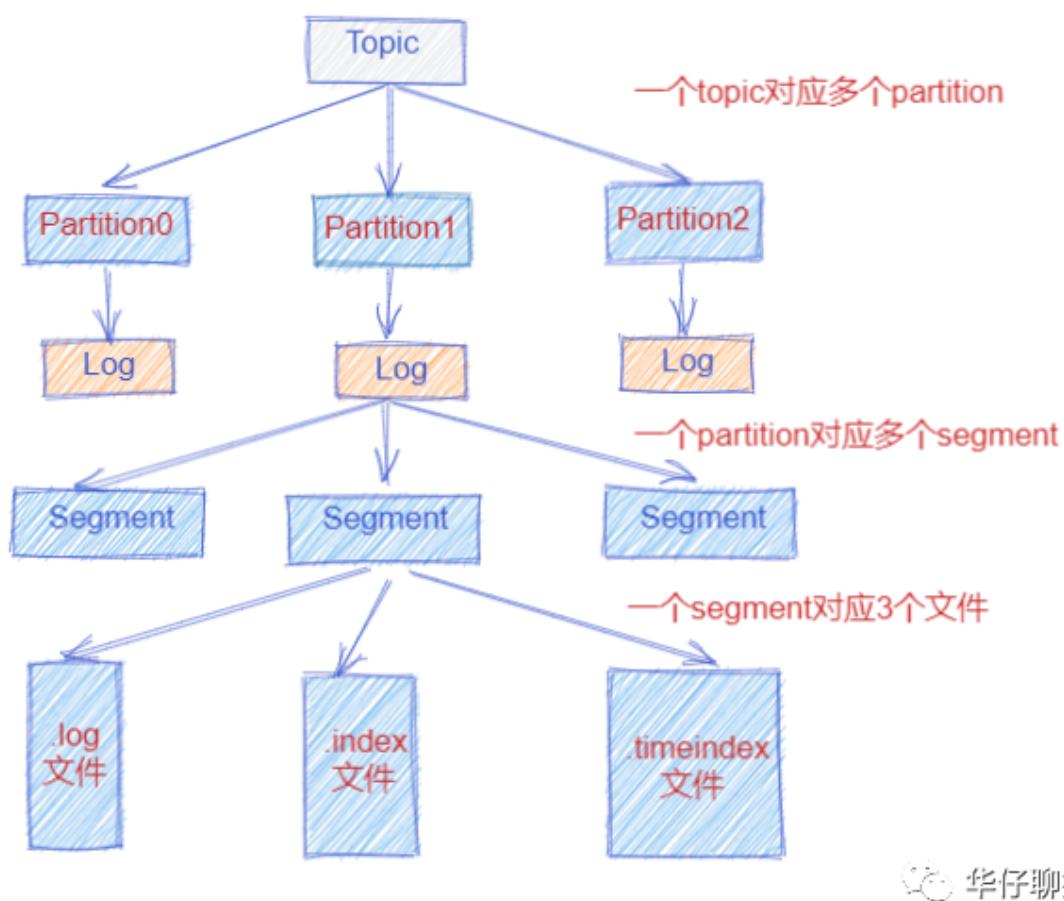
Kafka 整体架构图 01



一个典型的 Kafka 集群中包含若干 Producer，若干 Broker 「**Kafka支持水平扩展，一般 Broker 数量越多，集群吞吐率越高**」，若干 Consumer Group，以及一个 Zookeeper 集群。

Kafka 通过 Zookeeper 管理集群配置，选举 Leader，以及在 Consumer Group 发生变化时进行 Rebalance。Producer 使用 push 模式将消息发布到 Broker，Consumer 使用 pull 模式从 Broker 订阅并消费消息。

Kafka 存储机制 02



华仔聊技术

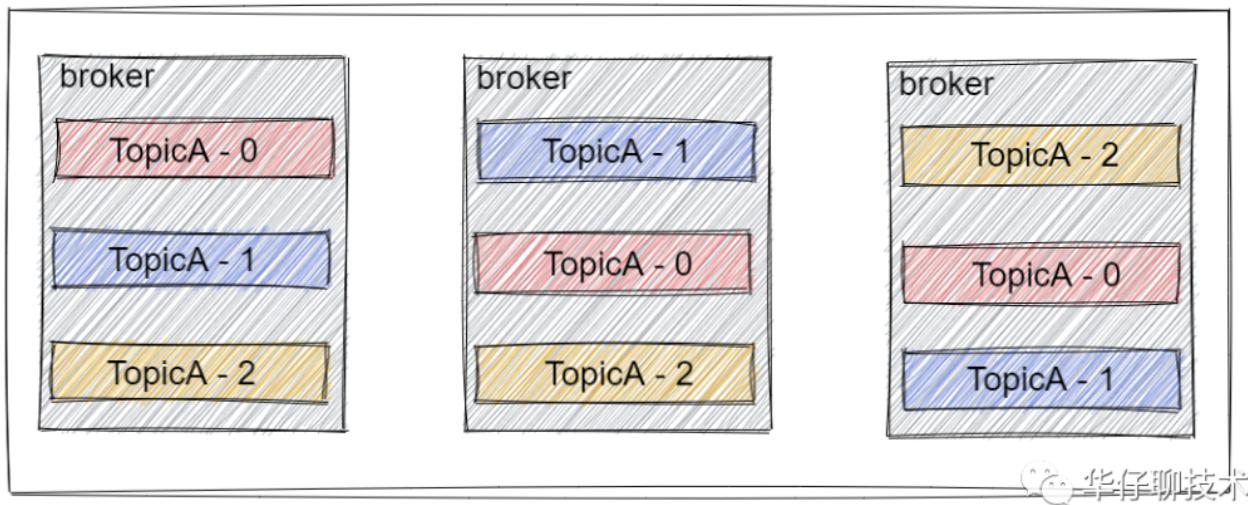
Producer 端生产的消息会不断追加到 log 文件末尾，这样文件就会越来越大，为了防止 log 文件过大导致数据定位效率低下，Kafka 采取了分片和索引机制。

它将每个 Partition 分为多个 Segment 每个 Segment 对应3个文件：

- 1) .index 索引文件
- 2) .log 数据文件
- 3) .timeindex 时间索引文件

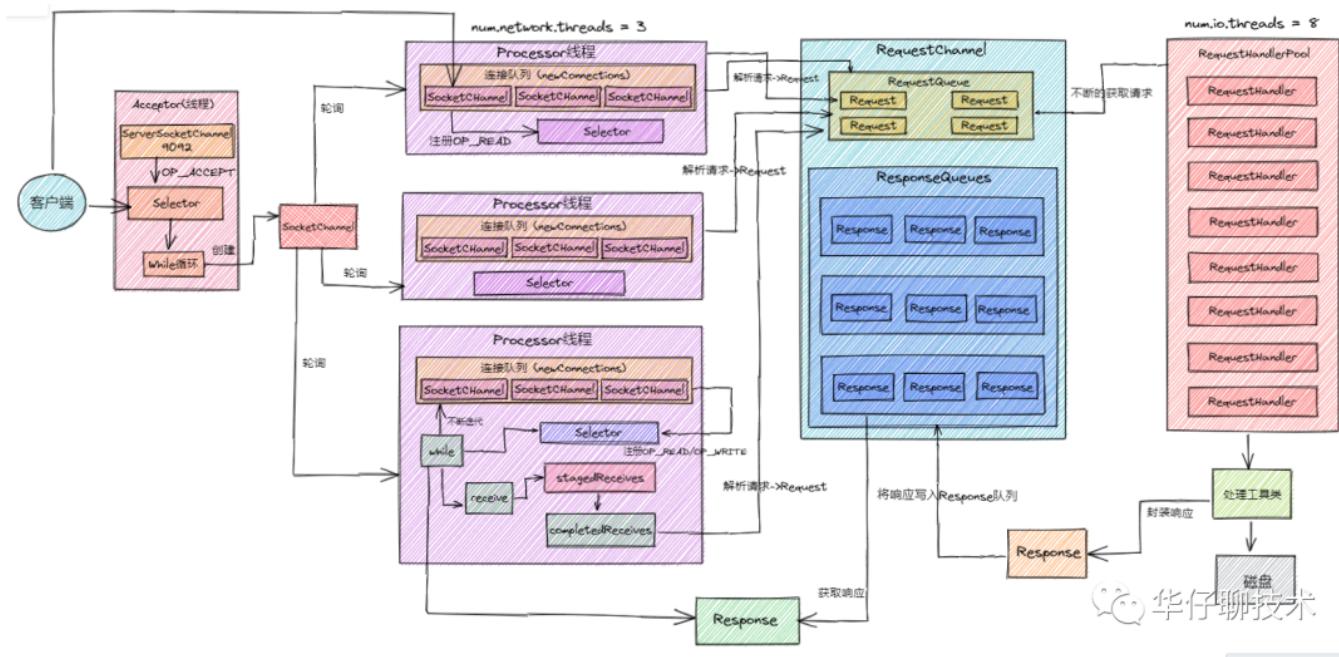
这些文件都位于同一文件夹下面，该文件夹的命名规则为：topic 名称-分区号。

Kafka 集群



Kafka中的 Partition 为了保证数据安全，每个 Partition 可以设置多个副本。此时我们对分区0,1,2分别设置3个副本。而且每个副本都是有「角色」之分的，**它们会选取一个副本作为 Leader 副本，而其他的作为 Follower 副本**，我们的 Producer 端在发送数据的时候，只能发送到Leader Partition 里面，然后 Follower Partition 会去 Leader Partition 自行同步数据，Consumer 消费数据的时候，也只能从 Leader 副本那去消费数据的。

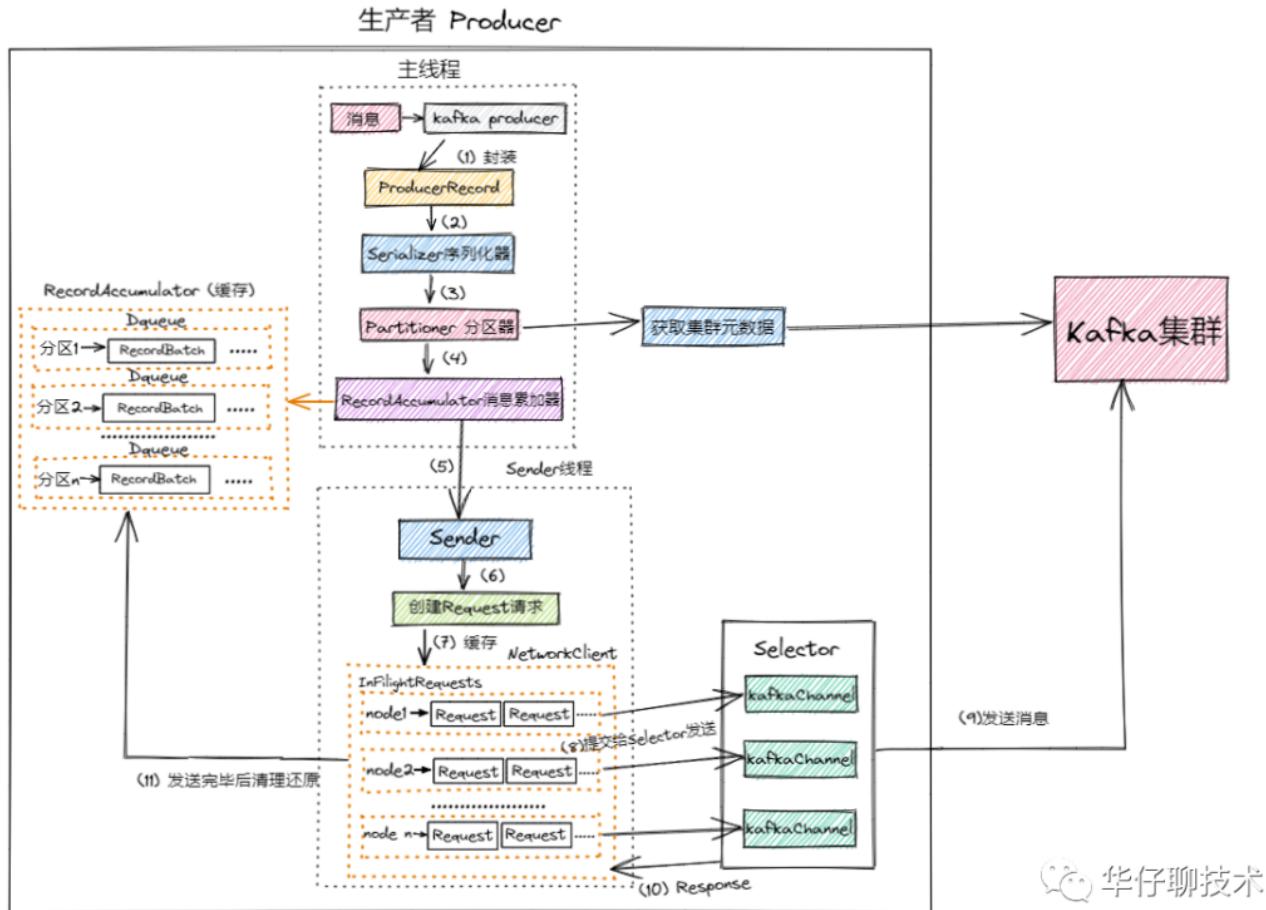
Kafka 网络模型 04



Kafka 采用多路复用方案，Reactor 设计模式，并引用 Java NIO 的方式更好的解决网络超高并发请求问题。

首先，大家知道的就是在客户端发送消息给 Kafka 服务端的时候，存在一个「**内存缓冲池机制**」的。即消息会先写入一个内存缓冲池中，然后直到多条消息组成了一个 Batch，达到一定条件才会一次网络通信把 Batch 发送过去。

整个发送过程图如下所示：



Kafka Producer 发送消息流程如下：

- 1) 进行 Producer 初始化，加载配置参数，开启网络线程。
- 2) 执行拦截器逻辑，预处理消息，封装 Producer Record。
- 3) 调用 Serializer.serialize() 方法进行消息的 key/value 序列化。
- 4) 调用 partition() 选择合适的分区策略，给消息体 Producer Record 分配要发送的 Topic 分区号。
- 5) 从 Kafka Broker 集群获取集群元数据 metadata。

华仔聊技术

6) 将消息缓存到 RecordAccumulator 收集器中，最后判断是否要发送。这个加入消息收集器，首先得从 Deque<RecordBatch> 里找到自己的目标分区，如果没有就新建一个 Batch 消息 Deque 加进入。

7) 当达到发送阈值，唤醒 Sender 线程，实例化 NetWorkClient 将 batch record 转换成 request client 的发送消息体，并将待发送的数据按 【Broker Id <=> List】的数据进行归类。

8) 与服务端不同的 Broker 建立网络连接，将对应 Broker 待发送的消息 List 发送出去。

9) 批次发送的条件为：缓冲区数据大小达到 batch.size 或者 linger.ms 达到上限，哪个先达到就算哪个。

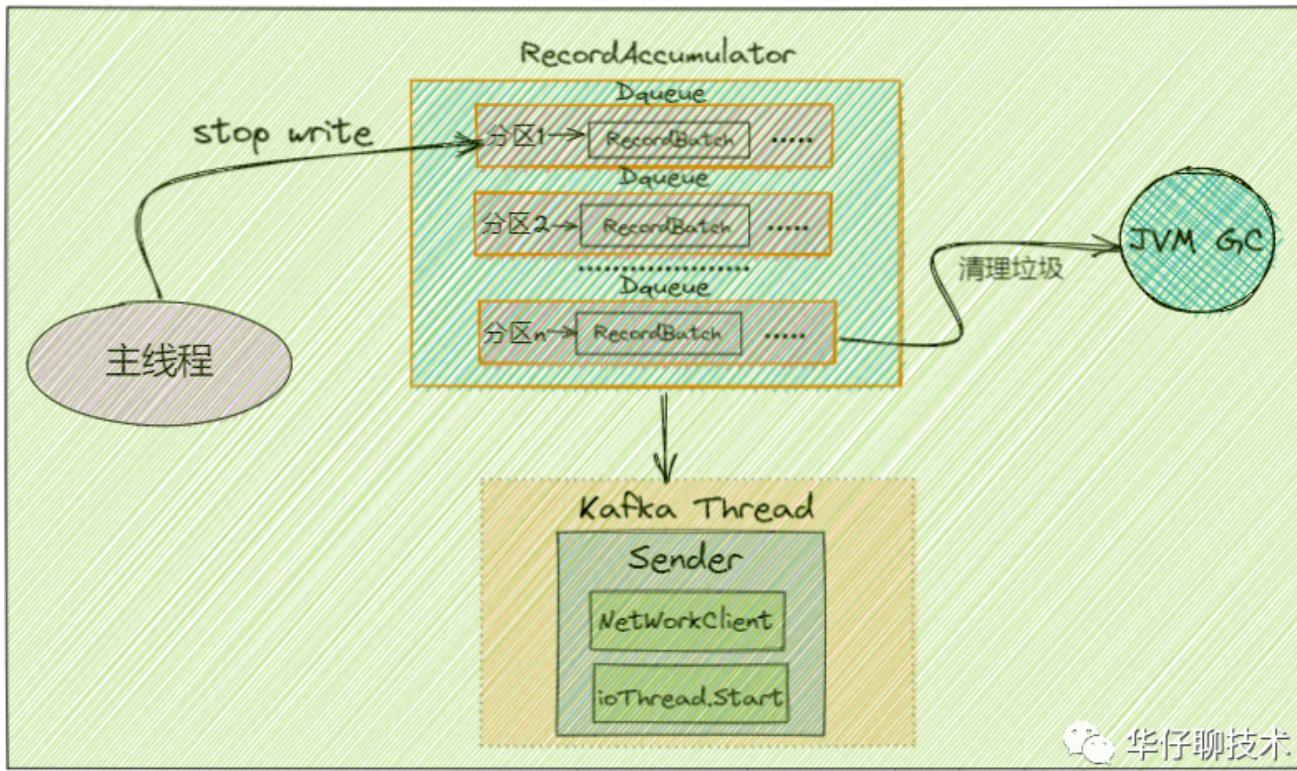
内存缓冲造成的频繁GC问题 02

内存缓冲机制说白了，其实就是**把多条消息组成一个Batch，一次网络请求就是一个Batch 或者多个 Batch**。这样避免了一条消息一次网络请求，从而提升了吞吐量。

那么问题来了，试想一下一个 Batch 中的数据取出来封装到网络包里，通过网络发送到达 Kafka 服务端。**此时这个 Batch 里的数据都发送过去了，里面的数据该怎么处理？**这些 Batch 里的数据还存在客户端的 JVM 的内存里！那么一定要避免任何变量去引用 Batch 对应的数据，然后尝试触发 JVM 自动回收掉这些内存垃圾。这样不断的让 JVM 进行垃圾回收，就可以不断的腾出来新的内存空间让后面新的数据来使用。

想法是挺好，但**实际生产运行的时候最大的问题，就是 JVM Full GC 问题**。JVM GC 在回收内存垃圾的时候，会有一个「**Stop the World**」的过程，即垃圾回收线程运行的时候，会导致其他工作线程短暂的停顿，这样可以踏踏实实的回收内存垃圾。

试想一下，在回收内存垃圾的时候，工作线程还在不断的往内存里写数据，那如何让JVM 回收垃圾呢？我们看看下面这张图就更加清楚了：



华仔聊技术

虽然现在 JVM GC 演进越来越先进，从 CMS 垃圾回收器到 G1 垃圾回收器，**核心的目标之一就是不断的缩减垃圾回收的时候，导致其他工作线程停顿的时间。**但是再先进的垃圾回收器这个停顿的时间还是存在的。

因此，如何尽可能在设计上避免 JVM 频繁的 Full GC 就是一个非常考验其设计水平了。

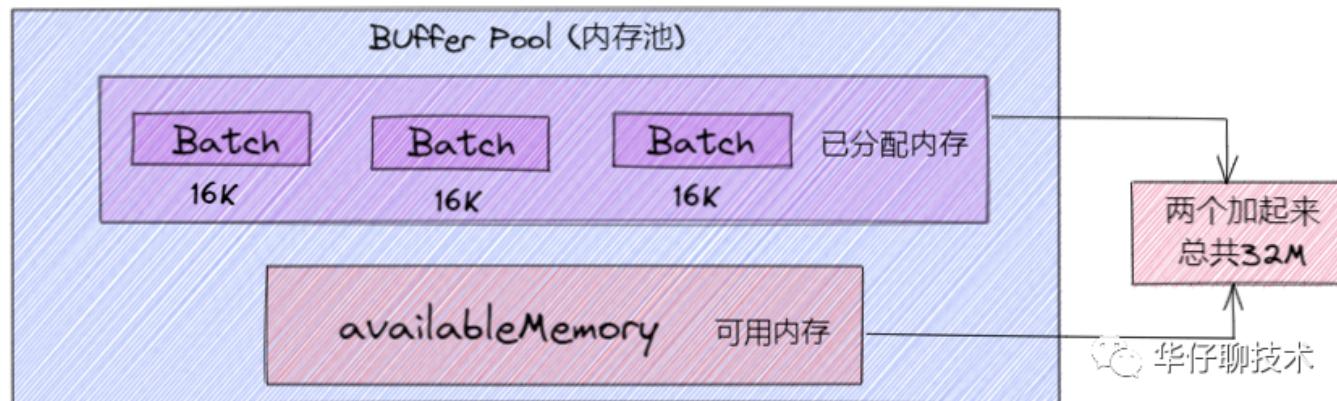
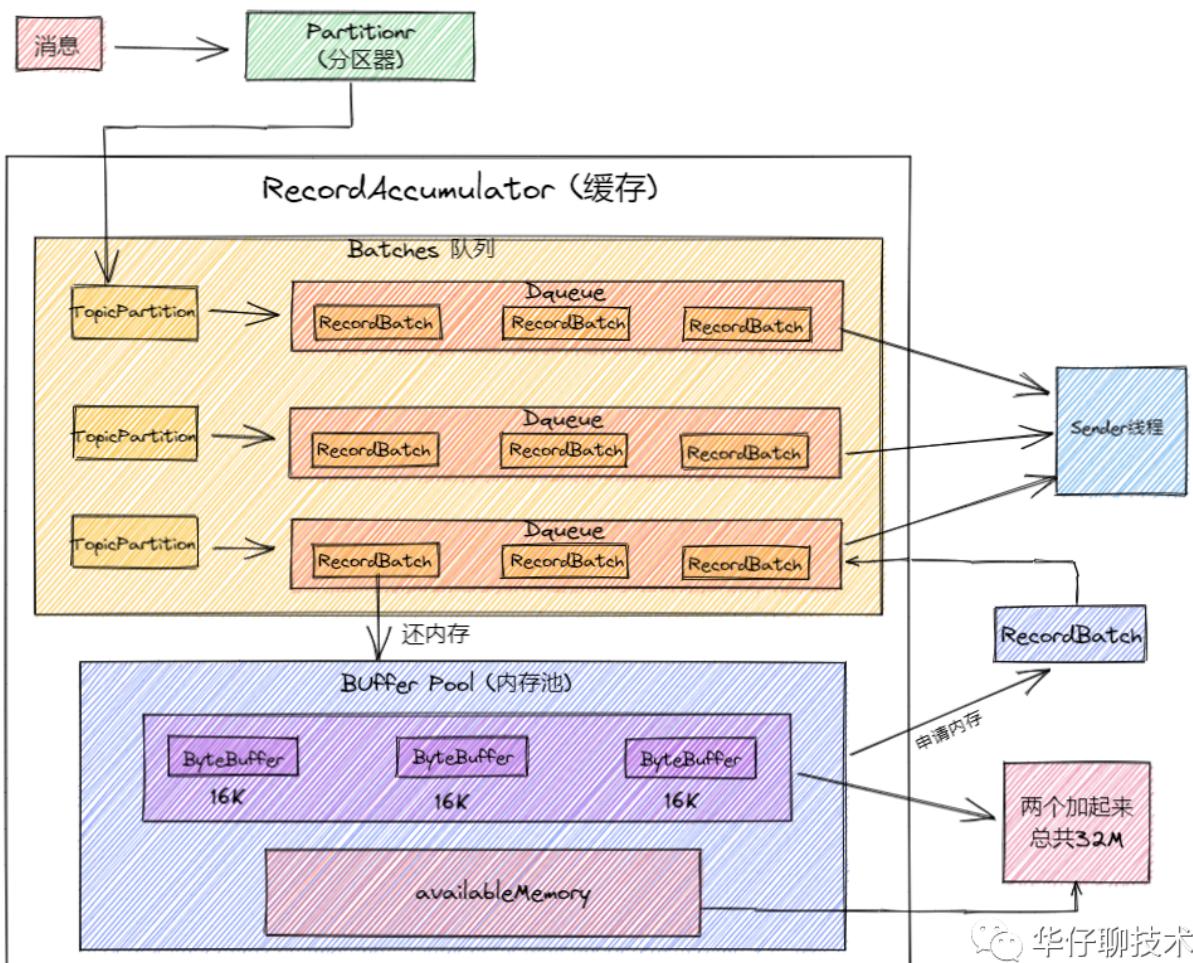
Kafka 实现的缓冲机制 03

在 Kafka 客户端内部，针对这个问题实现了一个非常优秀的机制，就是「**缓冲池机制**」。即每个 Batch 底层都对应一块内存空间，这个内存空间就是专门用来存放写进去的消息。

当一个 Batch 数据被发送到了 kafka 服务端，这个 Batch 的内存空间不再使用了。**此时这个 Batch 底层的内存空间先不交给 JVM 去垃圾回收，而是把这块内存空间放入一个缓冲池里。**

这个缓冲池里存放了很多块内存空间，下次如果有一个新的 Batch 数据了，那么直接从缓冲池获取一块内存空间是不是就可以了？然后如果一个 Batch 数据发送出去了之后，再把内存空间还回来是不是就可以了？以此类推，循环往复。

我们看看下面这张图就更加清楚了：



一旦使用了这个缓冲池机制之后，就不涉及到频繁的大量内存的 GC 问题了。

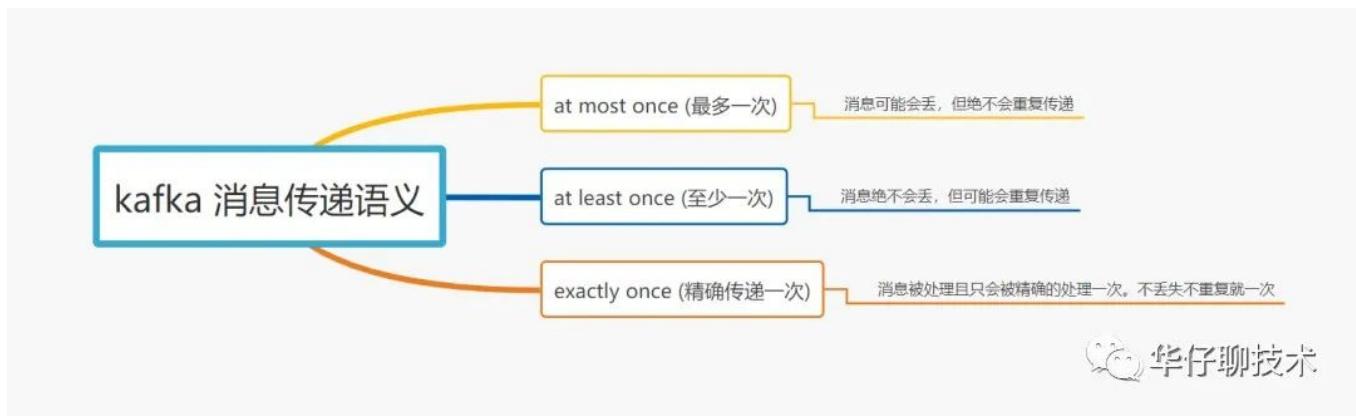
初始化分配固定的内存，即32MB。然后把 32MB 划分为 N 多个内存块，一个内存块默认是16KB，这样缓冲池里就会有很多的内存块。然后如果需要创建一个新的 Batch，就从缓冲池里取一个 16KB 的内存块就可以了。

接着如果 Batch 数据被发送到 Kafka 服务端了，此时 Batch 底层的内存块就直接还回缓冲池就可以了。这样循环往复就可以利用有限的内存，那么就不涉及到垃圾回收了。没有频繁的垃圾回收，自然就避免了频繁导致的工作线程的停顿了，JVM Full GC 问题是不是就得到了大幅度的优化？

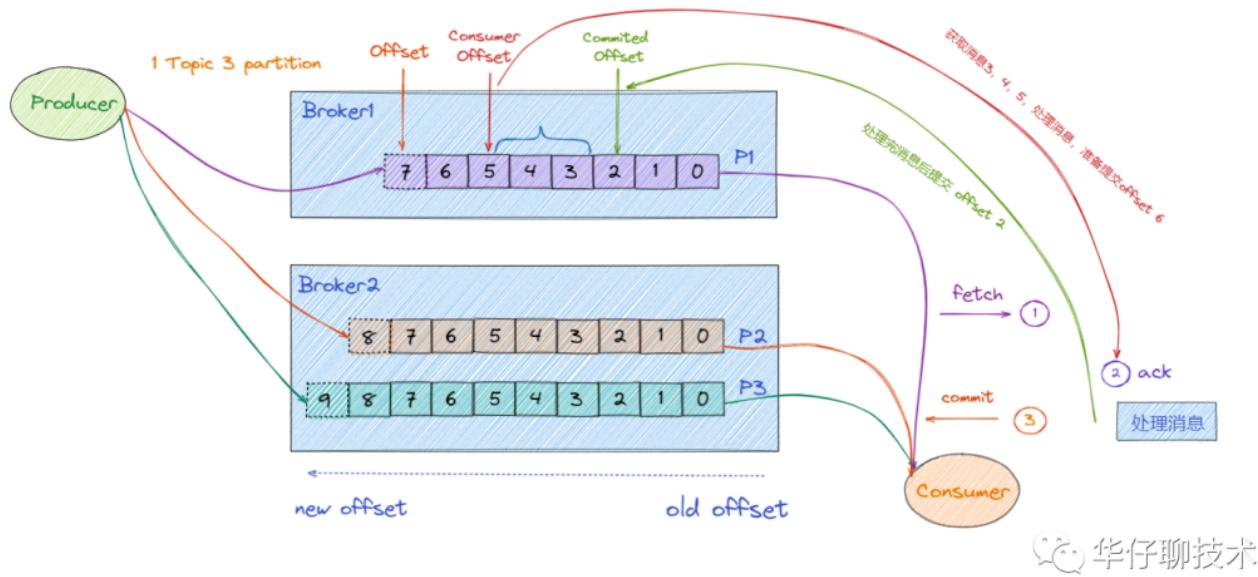
没错，正是这个设计思想让 Kafka 客户端的性能和吞吐量都非常的高，这里蕴含了大量的优秀的机制。

谈谈你对 Kafka 消息语义是如何理解的？

对于 Kafka 来说，当消息从 Producer 到 Consumer，有许多因素来影响消息的消费，因此「**消息传递语义**」就是 Kafka 提供的 Producer 和 Consumer 之间的消息传递过程中消息传递的保证性。主要分为三种，如下图所示：



对于这三种语义，我们来看一下可能出现的场景：



生产者发送语义：首先当 Producer 向 Broker 发送数据后，会进行消息提交，如果成功消息不会丢失。因此发送一条消息后，可能会有几种情况发生：

- 1) 遇到网络问题造成通信中断，导致 Producer 端无法收到 ack，Producer 无法准确判断该消息是否已经被提交，又重新发送消息，这就可能造成「at least once」语义。
- 2) 在 Kafka 0.11之前的版本，会导致消息在 Broker 上重复写入（保证至少一次语义），但在0.11版本开始，通过引入「**PID及Sequence Number**」支持幂等性，保证精确一次「**exactly once**」语义。

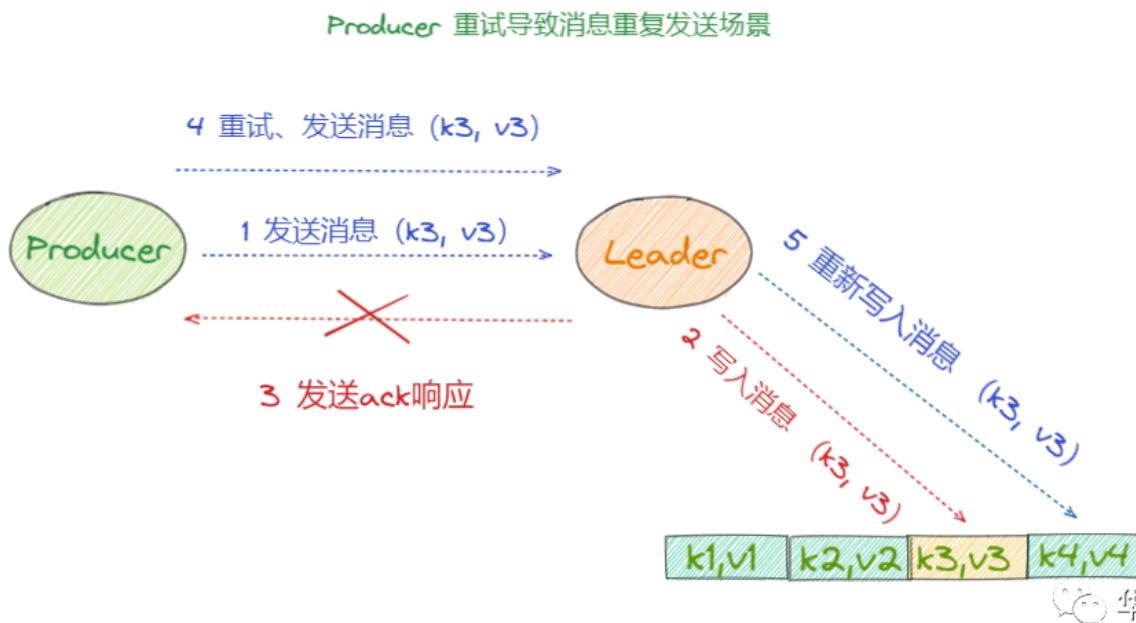
其中启用幂等传递的方法配置：enable.idempotence = true。**启用事务支持的方法配置：**设置属性 transctional.id = "指定值"。

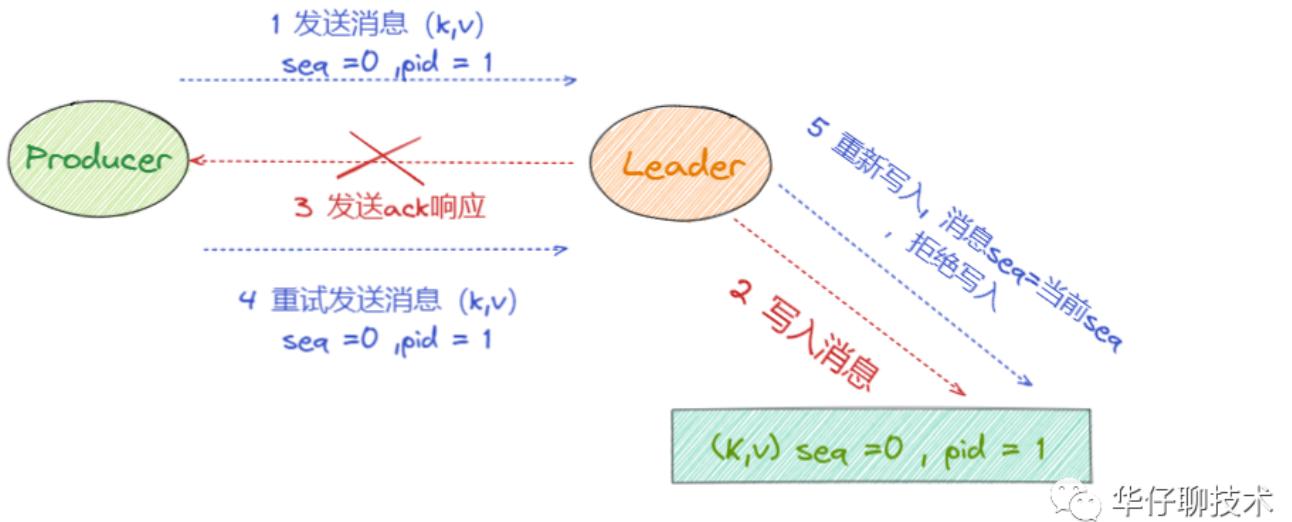
- 3) 可以根据 Producer 端 request.required.acks 的配置来取值。

acks = 0：由于发送后就自认为发送成功，这时如果发生网络抖动，Producer 端并不会校验 ACK 自然也就丢了，且无法重试。

acks = 1：消息发送 Leader Partition 接收成功就表示发送成功，这时只要 Leader Partition 不 Crash 掉，就可以保证 Leader Partition 不丢数据，保证「**at least once**」语义。

acks = -1 或者 all：消息发送需要等待 ISR 中 Leader Partition 和所有的 Follower Partition 都确认收到消息才算发送成功，可靠性最高，但也不能保证不丢数据，比如当 ISR 中只剩下 Leader Partition 了，这样就变成 acks = 1 的情况了，保证「**at least once**」语义。





Consumer端 02

消费者接收语义：从 Consumer 角度来剖析，我们知道 Offset 是由 Consumer 自己来维护的。

Consumer 消费消息时，有以下2种选择：

- 1) 读取消息 -> 提交offset -> 处理消息：如果此时保存 offset 成功，但处理消息失败，Consumer 又挂了，会发生 Rebalance，新接管的 Consumer 将从上次保存的 offset 的下一条继续消费，导致消息丢失，保证「**at most once**」语义。
- 2) 读取消息 -> 处理消息 -> 提交offset：如果此时消息处理成功，但保存 offset 失败，Consumer 又挂了，导致刚才消费的 offset 没有被成功提交，会发生 Rebalance，新接管的 Consumer 将从上次保存的 offset 的下一条继续消费，导致消息重复消费，保证「**at least once**」语义。

总结：默认 Kafka 提供「**at least once**」语义的消息传递，允许用户通过在处理消息之前保存 Offset 的方式提供「**at most once**」语义。如果我们可以自己实现消费幕等，理想情况下这个系统的消息传递就是严格的「**exactly once**」，也就是保证不丢失、且只会被精确的处理一次，但是这样是很难做到的。

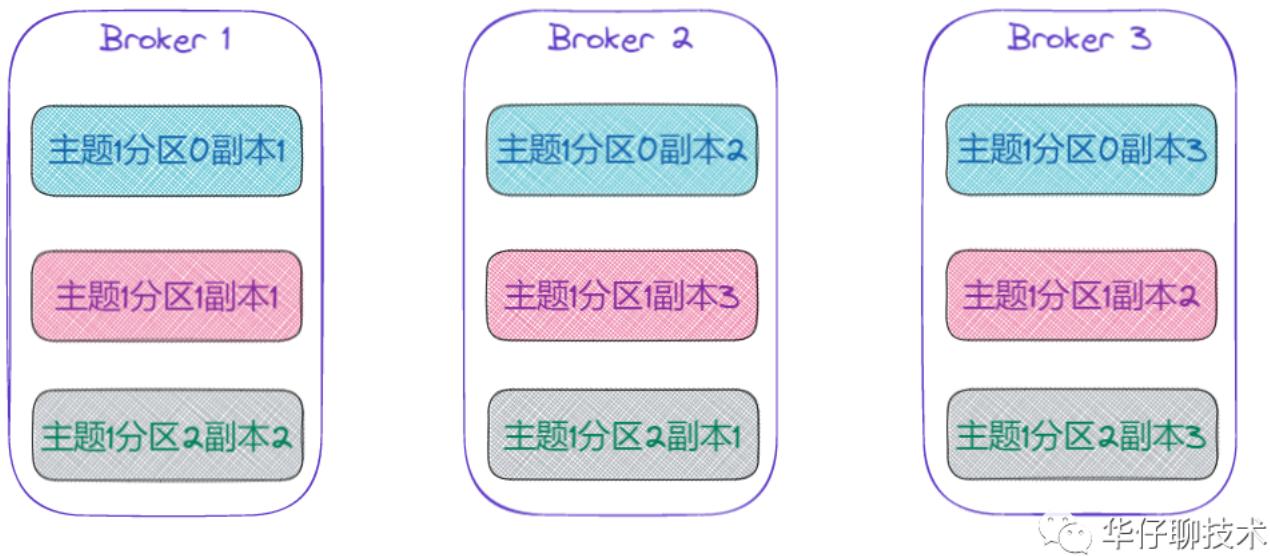
谈谈你对 Kafka 副本机制是如何理解的？

在上篇中，我们简单的分析了 Kafka 副本机制，这里我们再详细分析下 Kafka 的副本机制，说白了就是一个「**数据备份机制**」。

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，提高了系统可用性和数据持久性。

同一个分区下的所有副本保存相同的消息数据，这些副本分散保存在不同的 Broker 上，保证了 Broker 的整体可用性。

如下图所示：一个由 3 台 Broker 组成的 Kafka 集群上的副本分布情况。从这张图中，我们可以看到，主题 1 分区 1 的 3 个副本分散在 3 台 Broker 上，其他主题分区的副本也都散落在不同的 Broker 上，从而实现数据冗余。



华仔聊技术

副本同步机制 02

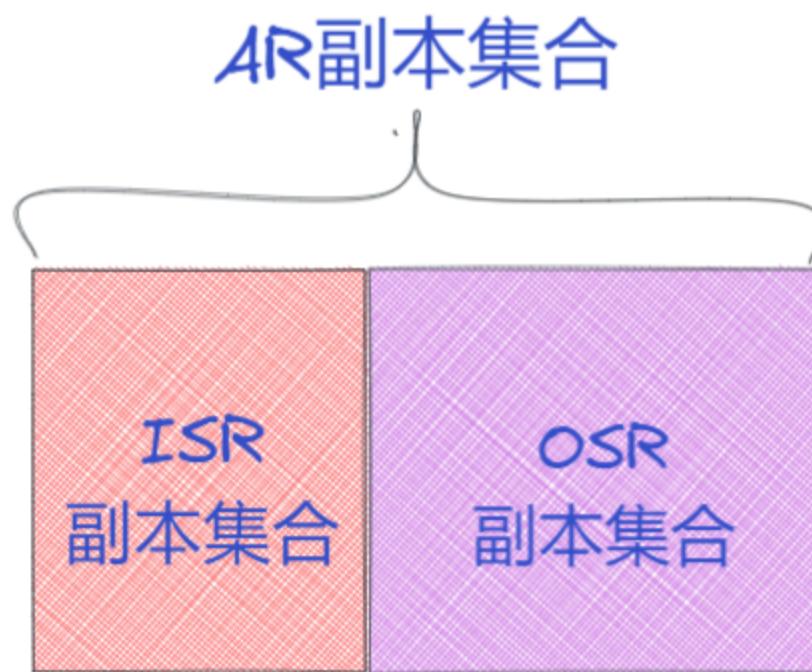
既然所有副本的消息内容相同，我们该如何保证副本中所有的数据都是一致的呢？当 Producer 发送消息到某个 Topic 后，消息是如何同步到对应的所有副本 Replica 中的呢？Kafka 中只有 Leader 副本才能对外进行读写服务，所以解决同步问题，Kafka 是采用基于 **Leader** 的副本机制来完成的。

1) 在 Kafka 中，一个 Topic 的每个 Partition 都有若干个副本，**副本分成两类：领导者副本「Leader Replica」和追随者副本「Follower Replica」**。每个分区在创建时都要选举一个副本作为领导者副本，其余的副本作为追随者副本。

2) 在 Kafka 中，Follower 副本是不对外提供服务的。也就是说，任何一个 Follower 副本都不能响应客户端的读写请求。**所有的读写请求都必须先发往 Leader 副本所在的 Broker，由该 Broker 负责处理。Follower 副本不处理客户端请求，它唯一的任务就是从 Leader 副本异步拉取消息，并写入到自己的提交日志中，从而实现与 Leader 副本的同步。**

3) 在 Kafka 2.X 版本中，当 Leader 副本所在的 Broker 宕机时，ZooKeeper 提供的监控功能能够实时感知到，并立即开启新一轮的 Leader 选举，从 ISR 副本集合中 Follower 副本中选一个作为新的 Leader，当旧的 Leader 副本重启回来后，只能作为 Follower 副本加入到集群中。3.x 的选举后续会有单篇去介绍。

副本管理 03



华仔聊技术

- 1) **AR 副本集合:** 分区 Partition 中的所有 Replica 组成 AR 副本集合。
- 2) **ISR 副本集合:** 所有与 Leader 副本能保持一定程度同步的 Replica 组成 ISR 副本集合，其中也包括 Leader 副本。
- 3) **OSR 副本集合:** 与 Leader 副本同步滞后过多的 Replica 组成 OSR 副本集合。

这里我们重点来分析下 **ISR 副本集合**。

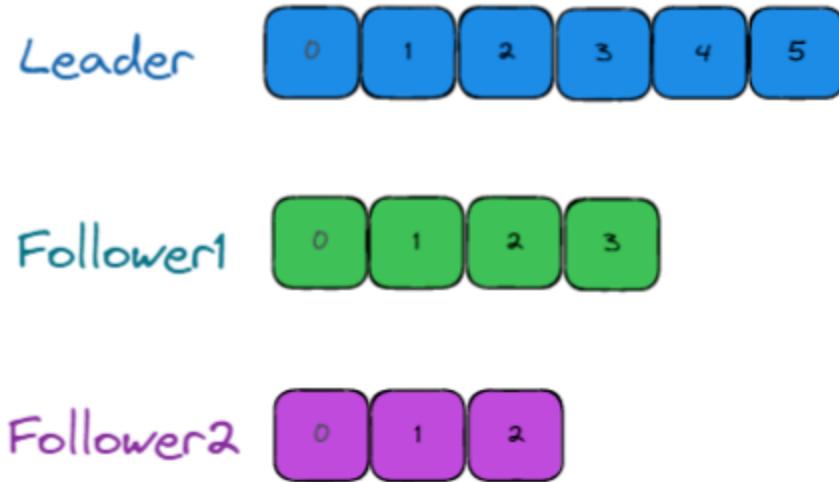
ISR 副本集合 04

上面强调过，Follower 副本不提供服务，只是定期地异步拉取 Leader 副本中的数据。既然是异步的，就一定会存在不能与 Leader 实时同步的情况出现。

Kafka 为了解决这个问题，引入了「**In-sync Replicas**」机制，即 ISR 副本集合。要求 ISR 副本集合中的 Follower 副本都是与 Leader 同步的副本。

那么，到底什么样的副本能够进入到 ISR 副本集合中呢？

首先要明确的，Leader 副本天然就在 ISR 副本集合中。也就是说，ISR 不只是有 Follower 副本集合，它必然包括 Leader 副本。另外，能够进入到 ISR 副本集合的 Follower 副本要满足一定的条件。



华仔聊技术

图中有 3 个副本：1 个 Leader 副本和 2 个 Follower 副本。Leader 副本当前写入了 6 条消息，Follower1 副本同步了其中的 4 条消息，而 Follower2 副本只同步了其中的 3 条消息。那么，对于这 2 个 Follower 副本，你觉得哪个 Follower 副本与 Leader 不同步？

事实上，这2个 Follower 副本都有可能与 Leader 副本同步，但也可能不与 Leader 副本同步，这个完全依赖于 Broker 端参数 `replica.lag.time.max.ms` 参数值。

这个参数是指 **Follower** 副本能够落后 **Leader** 副本的最长时间间隔，当前默认值是 10 秒，从 2.5 版本开始，默认值从 10 秒增加到 30 秒。即只要一个 Follower 副本落后 Leader 副本的时间不连续超过 30 秒，Kafka 就认为该 Follower 副本与 Leader 是同步的，即使 Follower 副本中保存的消息明显少于 Leader 副本中的消息。

此时如果这个副本同步过程的速度持续慢于 Leader 副本的消息写入速度的时候，那么在 `replica.lag.time.max.ms` 时间后，该 Follower 副本就会被认为与 Leader 副本是不同步的，因此 Kafka 会自动收缩，将其踢出 ISR 副本集合中。后续如果该副本追上了 Leader 副本的进度的话，那么它是能够重新被加回 ISR 副本集合的。

在默认情况下，当 Leader 副本发生故障时，只有在 ISR 副本集合中的 Follower 副本才有资格被选举为新 Leader，而 OSR 中副本集合的副本是没有机会的（可以通过 `unclean.leader.election.enable` 进行配置执行脏选举）。

总结：ISR 副本集合是一个动态调整的集合。

谈谈你对Kafka Leader选举机制是如何理解？

这里所说的 Leader 选举是指分区 Leader 副本的选举，**它是由 Kafka Controller 负责具体执行的，当创建分区或分区副本上线的时候都需要执行 Leader 的选举动作。**

有以下场景可能会触发选举：

- 1) 当 Controller 感知到分区 Leader 下线需要执行 Leader 选举。

此时的选举策略是：Controller 会从 AR 副本集合（同时也在ISR 副本集合）中按照副本的顺序取出第一个存活的副本作为 Leader。

一个分区的 AR 副本集合在分配的时候就被指定，并且只要不发生重分配集合内部副本的顺序是保持不变的，而分区的 ISR 副本集合中副本的顺序可能会改变。

注意这里是根据 AR 副本集合的顺序而不是 ISR 副本结合的顺序进行选举的。

此时如果 ISR 副本集合中没有可用的副本，还需要再检查一下所配置的 unclean.leader.election.enable 参数「**默认值为false**」。**如果这个参数配置为true，那么表示允许从非 ISR 副本集合中选举 Leader，从 AR 副本集合列表中找到第一个存活的副本即为 Leader。**

- 2) 当分区进行重分配的时候也需要进行 Leader 选举。

此时的选举策略是：**从重分配的 AR 副本集合中找到第一个存活的副本，且这个副本在当前的 ISR 副本集合中。当发生优先副本的选举时，直接将优先副本设置为 Leader 即可，AR 副本集合中的第一个副本即为优先副本。**

- 3) 当某节点执行 ControlledShutdown 被优雅地关闭时，位于这个节点上的 Leader 副本都会下线，所以与此对应的分区需要执行 Leader 的选举。

此时的选举策略是：**从 AR 副本集合中找到第一个存活的副本，且这个副本在当前的 ISR 副本集合中，同时还要确保这个副本不处于正在被关闭的节点上。**

谈谈你对Kafka控制器及选举机制是如何理解？

所谓的控制器「**Controller**」就是通过 ZooKeeper 来管理和协调整个 Kafka 集群的组件。集群中任意一台 Broker 都可以充当控制器的角色，但是在正常运行过程中，只能有一个 Broker 成为控制器。

控制器的职责主要包括：

- 1) 集群元信息管理及更新同步 (Topic路由信息等)。
- 2) 主题管理 (创建、删除、增加分区等)。
- 3) 分区重新分配。
- 4) 副本故障转移、 Leader 选举、ISR 变更。
- 5) 集群成员管理 (通过 watch 机制自动检测新增 Broker、Broker 主动关闭、Broker 宕机等)。

控制器机制 01

我们知道 Kafka 2.X 版本是依赖 Zookeeper 来维护集群成员的信息：

- 1) Kafka 使用 Zookeeper 的临时节点来选举 Controller。
- 2) Zookeeper 在 Broker 加入集群或退出集群时通知 Controller。
- 3) Controller 负责在 Broker 加入或离开集群时进行分区 Leader 选举。

控制器数据分布 02

分类	数据描述
Broker 相关	当前存活的 broker 列表
	正在关闭中的 broker 列表
	获取某个 broker 上的所有分区
	某组 broker 上的所有副本
Topic 相关	topic 列表
	某个 topic 的所有分区和所有副本
	移除某个 topic 的所有信息

运维任务 副本相关	正在进行的 Leader 选举的分区
	当前存活的所有副本
	分配给每个分区的副本列表
	正在进行重分配的分区列表
	某组分区下的所有副本

从上面表格可以看出, 存储的大概有3大类:

- 1) 所有topic信息: 包括具体的分区信息, 比如 Leader 副本是谁, ISR 集合中有哪些副本等。
- 2) 所有 Broker 信息: 包括当前都有哪些运行中的 Broker, 哪些正在关闭中的 Broker 等。
- 3) 涉及运维任务的副本分区: 包括当前正在进行 Leader 选举以及分区重分配的分区列表等。

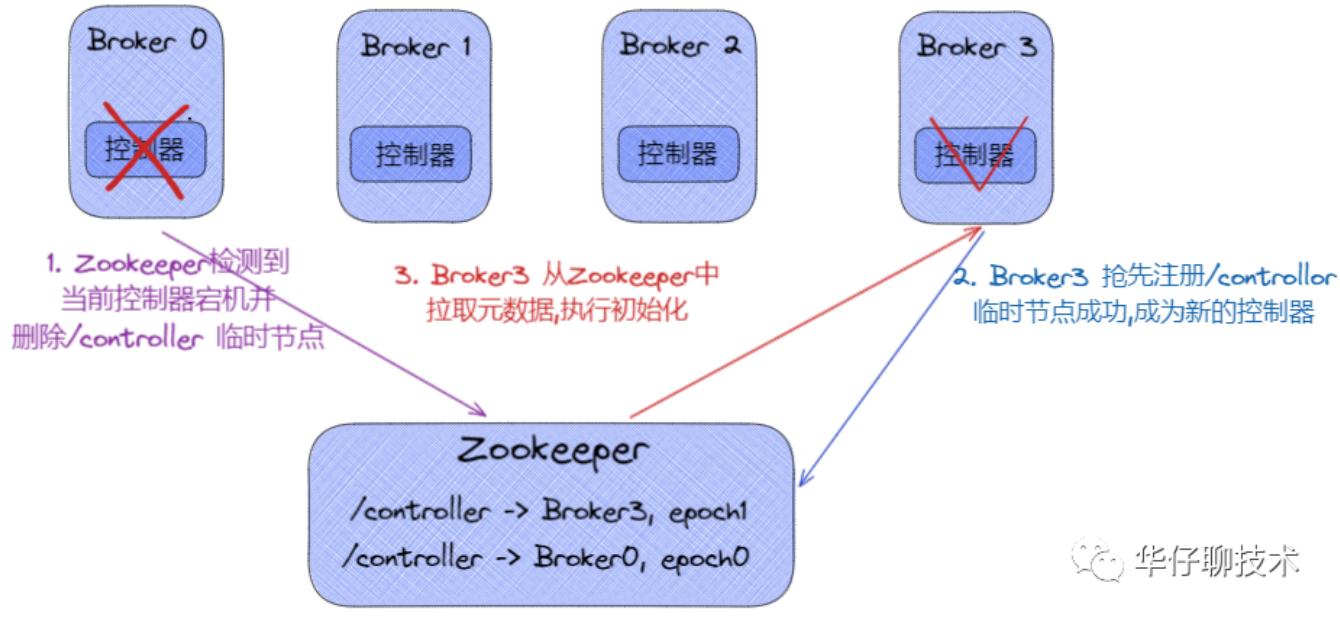
控制器故障转移

03

在 Kafka 集群运行过程中, 只能有一台 Broker 充当控制器的角色, 存在「单点故障」的风险, Kafka 如何应对单点故障呢?

其实 Kafka 为控制器提供故障转移功能「Failover」。指当运行中的控制器突然宕机时, Kafka 能够快速地感知到, 并立即启用备用控制器来代替之前失败控制器的过程。

下面通过一张图来展示控制器故障转移的过程:



控制器触发选举场景 04

至此你一定想知道控制器是如何被选出来的？前面说过，每台 Broker 都能充当控制器，当集群启动后，Kafka 是如何确认控制器在哪台 Broker 呢？

实际上这个问题很简单，即 **Broker 启动时，会尝试去 ZooKeeper 中创建 /controller 节点，第一个成功创建 /controller 节点的 Broker 会被选为控制器。**

接下来我们看下**触发 Controller 选举的场景**有哪些？

场景一、集群首次启动时：

集群首次启动时，Controller 还未被选举出来，因此 Broker 启动后，会干4件事：

- 1) 先注册 Zookeeper 状态变更监听器，用来监听 Broker 与 Zookeeper 之间的会话是否过期。
- 2) 然后将 Startup 这个控制器事件写入到事件队列中。
- 3) 然后开始启动对应的控制器事件处理线程即「**ControllerEventThread**」、以及「**ControllerChangeHandler**」 Zookeeper 监听器，开始处理事件队列中Startup 事件。
- 4) 最后依赖事件处理线程来选举 Controller。

场景二、Broker 监听 /controller 节点消失时：

集群运行过程中，当 Broker 监听到 /controller 节点消失时，就表示此时当前整个集群中已经没有 Controller 了。所有监听到 /controller 节点消失的 Broker，此时都会开始执行竞选操作。

那么 Broker 是如何监听到 ZooKeeper 上的变化呢？主要依赖 ZooKeeper 监听器提供的功能，所以 Kafka 是依赖 ZooKeeper 来完成 Controller 的选举。

对于 Kafka 3.X 版本中，内部实现一个类似于 Raft 的共识算法来选举 Controller。

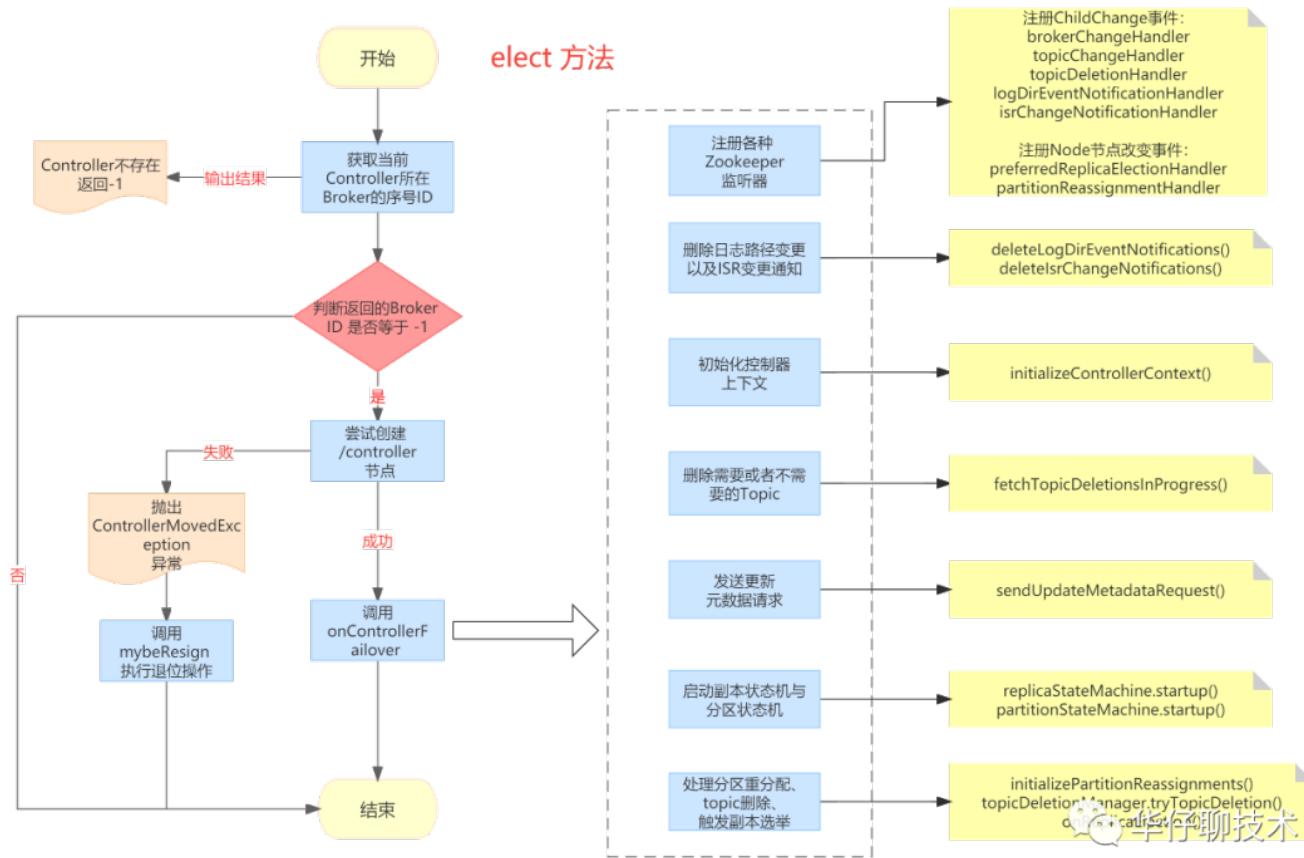
场景三、Broker 监听 /controller 节点数据变化时：

集群运行过程中，当 Broker 检测到 /controller 节点数据发生变化，此时 Controller 可能已经被「易主」了，这时有以下两种情况：

- 1) 假如 Broker 是 Controller，那么该 Broker 需要首先执「退位」操作，然后再尝试进行竞选 Controller。
- 2) 假如 Broker 不是 Controller，那么，该 Broker 直接去竞选新 Controller。

控制器选举机制 05

其实选举最终都是通过调用底层的 elect 方法进行选举，如下图所示：

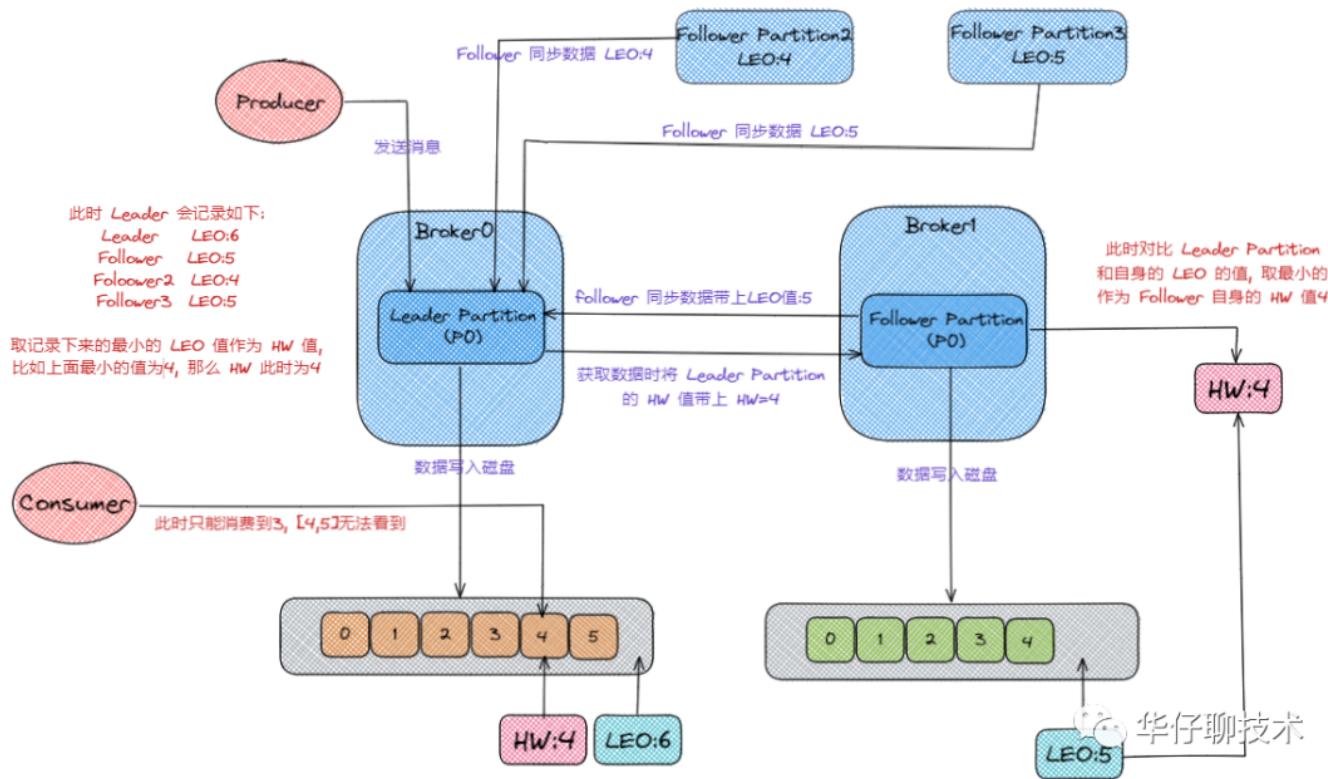


谈谈 kafka 的数据可靠性是怎么保证的?

开始数据可靠性之前先看几个重要的概念：AR、OSR、ISR、HW、LEO，前面已经讲了 AR、OSR、ISR。这里我们重点讲下 HW、LEO。

HW: 全称「**Hign WaterMark**」，即高水位，它标识了一个特定的消息偏移量 offset，消费者只能拉取到这个水位 offset 之前的消息。

LEO: 全称「**Log End Offset**」，它标识当前日志文件中下一条待写入的消息的 offset，在 ISR 副本集合中的每个副本都会维护自身的LEO。



从上图可以看出 HW 和 LEO 的作用：

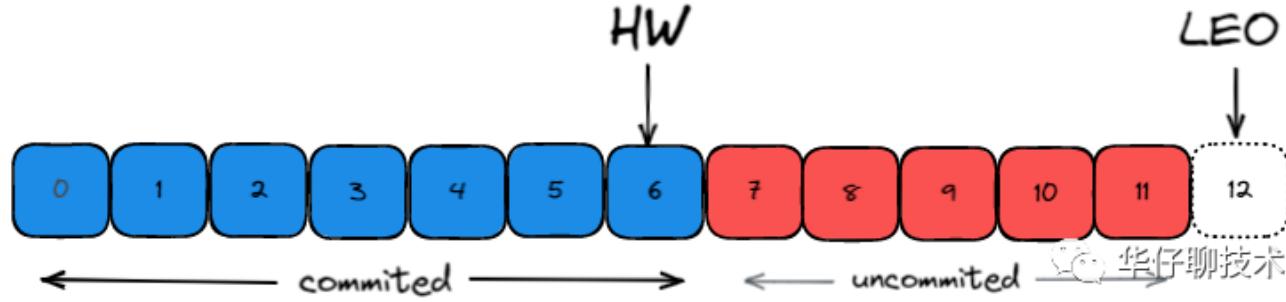
HW 作用：

- 1) 用来标识分区下的哪些消息是可以被消费者消费的。
- 2) 协助 Kafka 完成副本数据同步。

LEO 作用：

- 1) 如果 Follower 和 Leader 的 LEO 数据同步了，那么 HW 就可以更新了。

2) HW 之前的消息数据对消费者是可见的，属于 committed 状态，HW 之后的消息数据对消费者是不可见的。



HW 更新是需要一轮额外的拉取请求才能实现，Follower 副本要拉取 Leader 副本的数据，也就是说，Leader 副本 HW 更新和 Follower 副本 HW 更新在时间上是存在错配的。这种错配是很多“数据丢失”或“数据不一致”问题的根源。因此社区在 0.11 版本正式引入了「Leader Epoch」概念，来规避因 HW 更新错配导致的各种不一致问题。

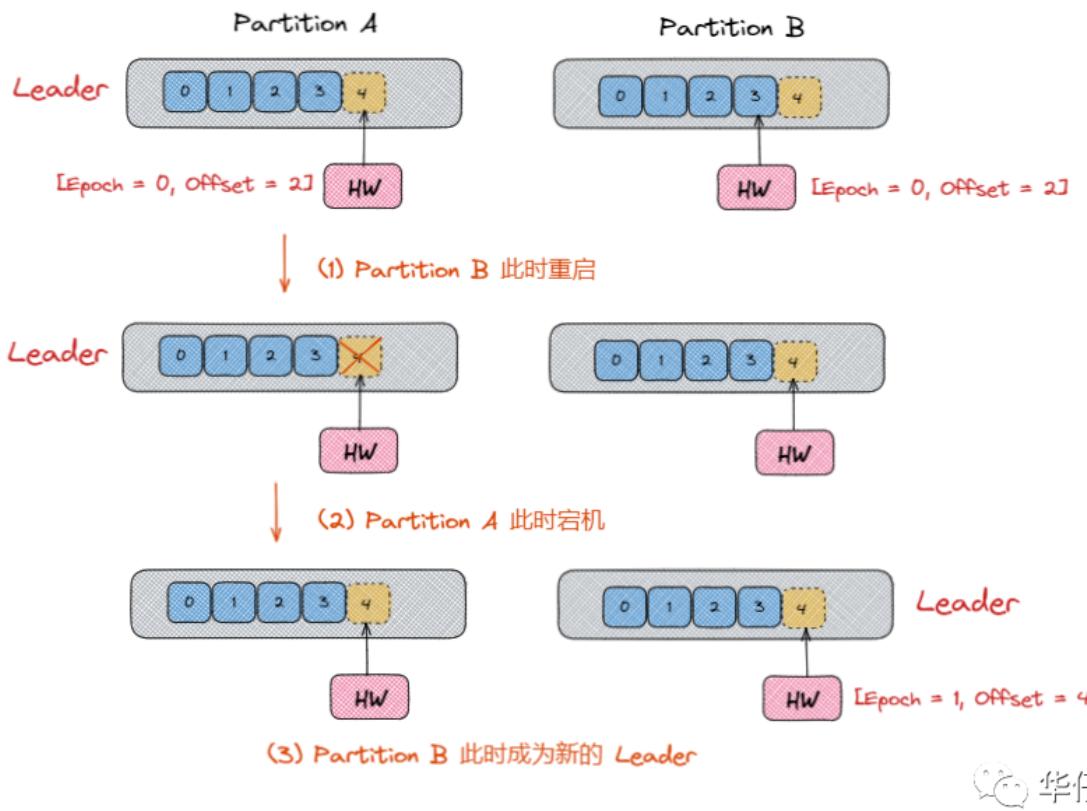
所谓 Leader Epoch，我们大致可以认为是 Leader 版本。它由两部分数据组成：

- 1) **Epoch**: 一个单调递增的版本号。每当副本 Leader 权力发生变更时，都会增加该版本号。小版本号的 Leader 被认为是过期 Leader，不能再行使 Leader 权力。
- 2) **起始位移 (Start Offset)** : Leader 副本在该 Epoch 值上写入的首条消息的位移。

Kafka Broker 会在内存中为每个分区都缓存 Leader Epoch 数据，同时它还会定期地将这些信息持久化到一个 checkpoint 文件中。当 Leader Partition 写入消息到磁盘时，Broker 会尝试更新这部分缓存。如果该 Leader 是首次写入消息，那么 Broker 会向缓存中增加一个 Leader Epoch 条目，否则就不做更新。这样，每次有 Leader 变更时，新的 Leader 副本会查询这部分缓存，取出对应的 Leader Epoch 的起始位移，以避免数据丢失和不一致的情况。

严格来说，这个场景发生的前提是 **Broker 端参数 min.insync.replicas 设置为 1**。此时一旦消息被写入到 Leader 副本的磁盘，就会被认为是 committed 状态，但因存在时间错配问题导致 Follower 的 HW 更新是有滞后的。如果在这个短暂的滞后时间内，接连发生 Broker 宕机，那么这类数据的丢失就是无法避免的。

接下来，我们来看下如何利用 Leader Epoch 机制来规避这种数据丢失。如下图所示：



华仔聊技术

因此 Kafka 只对 「已提交」 的消息做 「最大限度的持久化保证不丢失」。由于篇幅详细请看：[刨根问底：Kafka 到底会不会丢数据](#)

谈谈 Kafka 消息分配策略都有哪些？

Kafka Partition 分配策略机制

RangeAssignor 分配策略

RoundRobinAssignor 分配策略

StickyAssignor 分配策略

华仔聊技术

这里主要说的是消费的分区分配策略，我们知道一个 Consumer Group 中有多个 Consumer，一个 Topic 也有多个 Partition，所以必然会有 Partition 分配问题「确定哪个 Partition 由哪个 Consumer 来消费的问题」。

Kafka 客户端提供了3 种分区分配策略：RangeAssignor、RoundRobinAssignor 和 StickyAssignor，前两种

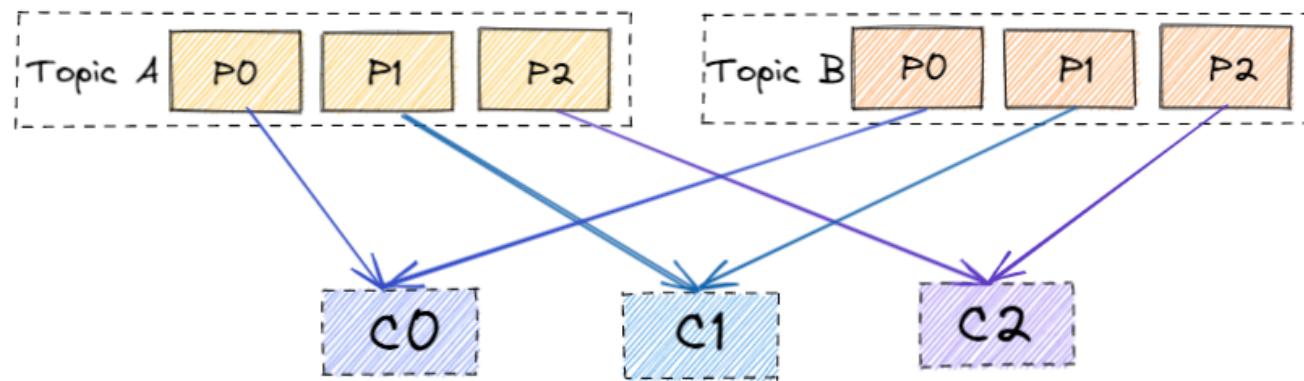
分配方案相对简单一些StickyAssignor 分配方案相对复杂一些。

RangeAssignor

01

RangeAssignor 是 Kafka 默认的分区分配算法，它是按照 Topic 的维度进行分配的，首先对 每个Topic 的 Partition 按照分区ID进行排序，然后对订阅该 Topic 的 Consumer Group 的 Consumer 按名称字典进行排序，之后尽量均衡的按照范围区段将分区分配给 Consumer。此时也可能会造成先分配分区的 Consumer 任务过重（分区数无法被消费者数量整除）。

分区分配场景分析如下图所示（同一个消费者组下的多个 consumer）：



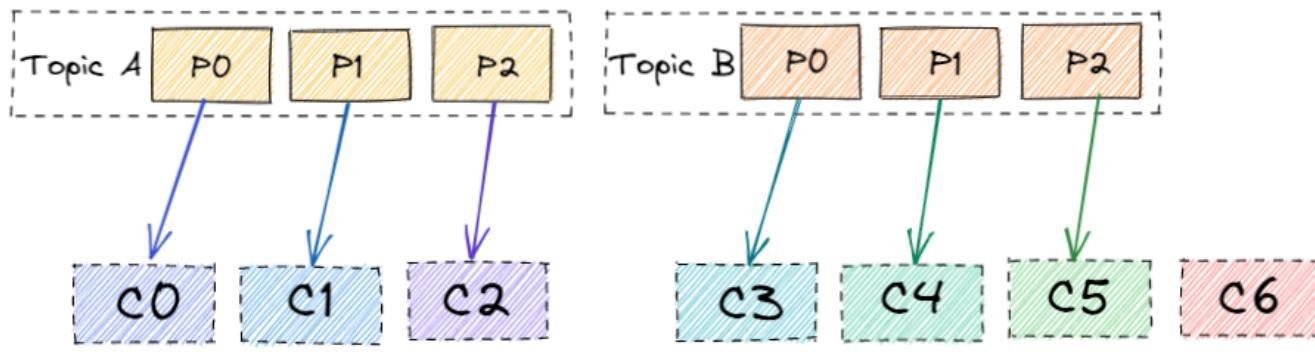
2个Topic 6个Partition, 3个Consumer的场景
分配结果如下：

C0 : [Topic A P0, Topic B P0]

C1 : [Topic A P1, Topic B P1]

C2 : [Topic A P2, Topic B P2]

华仔聊技术



2个Topic 6个Partition, 7个Consumer的场景

分配结果如下:

- C0: [Topic A P0]
- C1: [Topic A P1]
- C2: [Topic A P2]
- C3: [Topic B P0]
- C4: [Topic B P1]
- C5: [Topic B P2]
- C6: [空]

当Consumer数量超过Partition数量的场景，排序靠前的consumer优先分配到partition，排序靠后的consumer可能分配不到partition

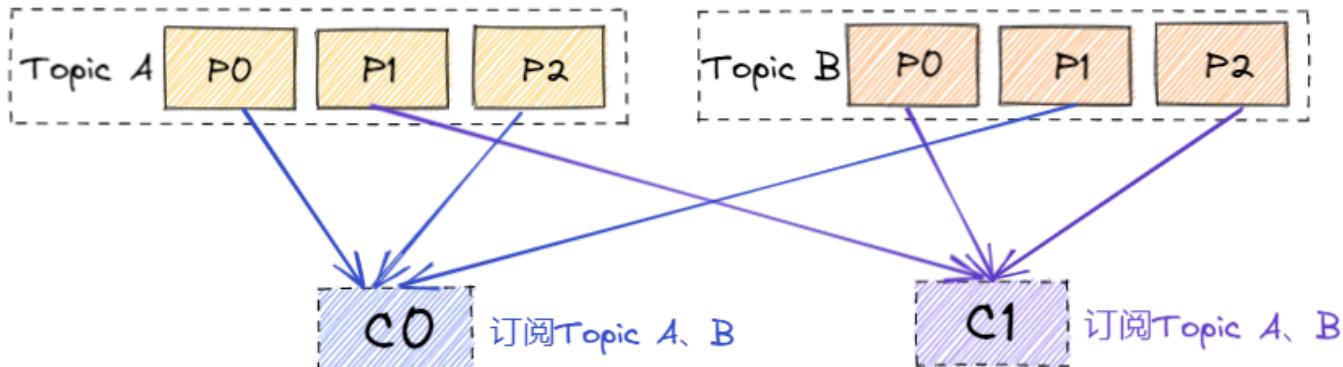
总结：该分配方式明显的问题就是随着消费者订阅的Topic的数量的增加，不均衡的问题会越来越严重。

RoundRobinAssignor 02

该分区分配策略是将 Consumer Group 订阅的所有 Topic 的 Partition 及所有 Consumer 按照字典进行排序后尽量均衡的挨个进行分配。如果 Consumer Group 内，每个 Consumer 订阅都订阅了相同的Topic，那么分配结果是均衡的。如果订阅 Topic 是不同的，那么分配结果是不保证「**尽量均衡**」的，因为某些 Consumer 可能不参与一些 Topic 的分配。

分区分配场景分析如下图所示：

1) 当组内每个 Consumer 订阅的相同 Topic :



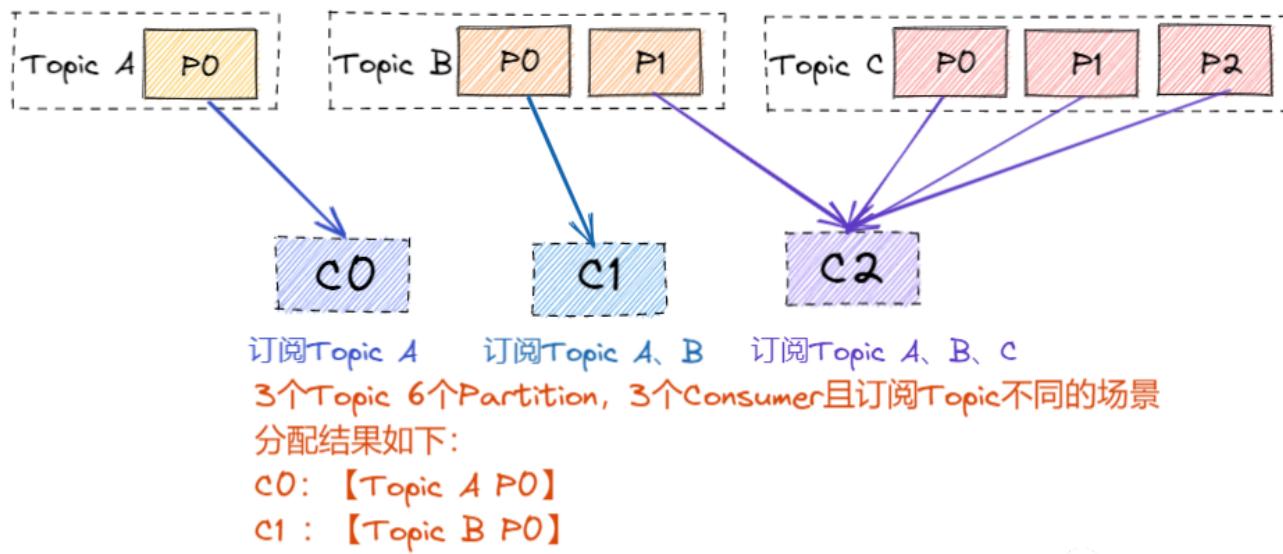
2个Topic 6个Partition, 2个Consumer且订阅Topic相同的场景
分配结果如下:

C0: [Topic A P0, P2, Topic B P1]

C1: [Topic A P1, Topic B P0, P2]

华仔聊技术

2) 当组内每个订阅的不同的 Topic, 这样就可能会造成本区订阅的倾斜:



订阅Topic A 订阅Topic A、B 订阅Topic A、B、C

3个Topic 6个Partition, 3个Consumer且订阅Topic不同的场景

分配结果如下:

C0: [Topic A P0]

C1: [Topic B P0]

C2: [Topic B P1, Topic C P0, P1, P2]

华仔聊技术

StickyAssignor 03

该分区分配算法是最复杂的一种, 可以通过 `partition.assignment.strategy` 参数去设置, 从 0.11 版本开始引入, 目的就是在执行新分配时, 尽量在上一次分配结果上少做调整, 其主要实现了以下2个目标:

1、Topic Partition 的分配要尽量均衡。

2、当 Rebalance 发生时, 尽量与上一次分配结果保持一致。

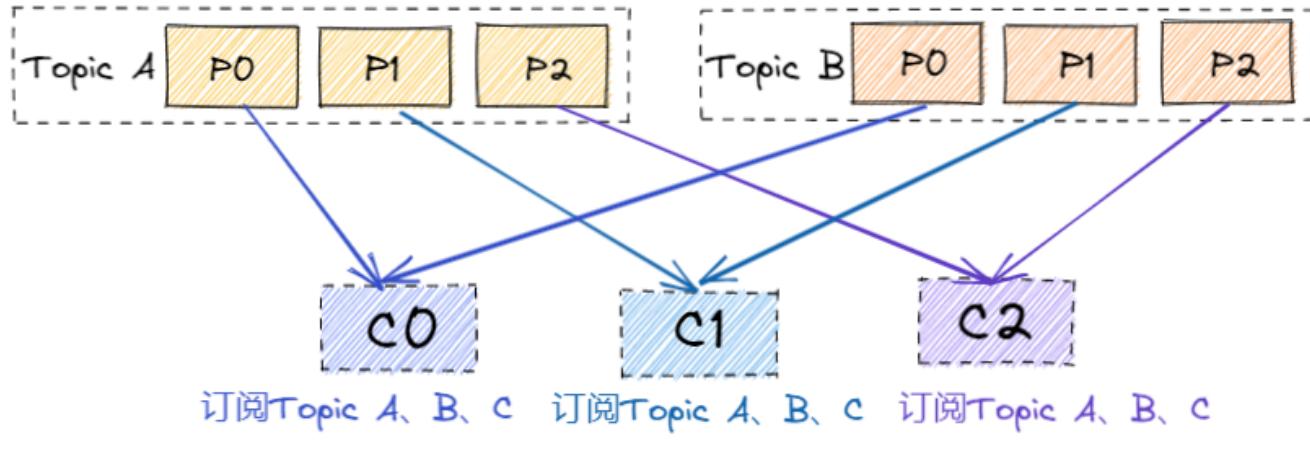
注意: 当两个目标发生冲突的时候, 优先保证第一个目标, 这样可以使分配更加均匀, 其中第一个目标是3种分配

策略都尽量去尝试完成的，而第二个目标才是该算法的精髓所在。

下面我们看看该策略与RoundRobinAssignor策略的不同：

分区分配场景分析如下图所示：

1) 组内每个 Consumer 订阅的相同的 Topic , RoundRobinAssignor 跟StickyAssignor 分配一致：



2个Topic 6个Partition, 3个Consumer且订阅Topic相同的场景
分配结果如下：

C0: [Topic A P0, Topic B P0]

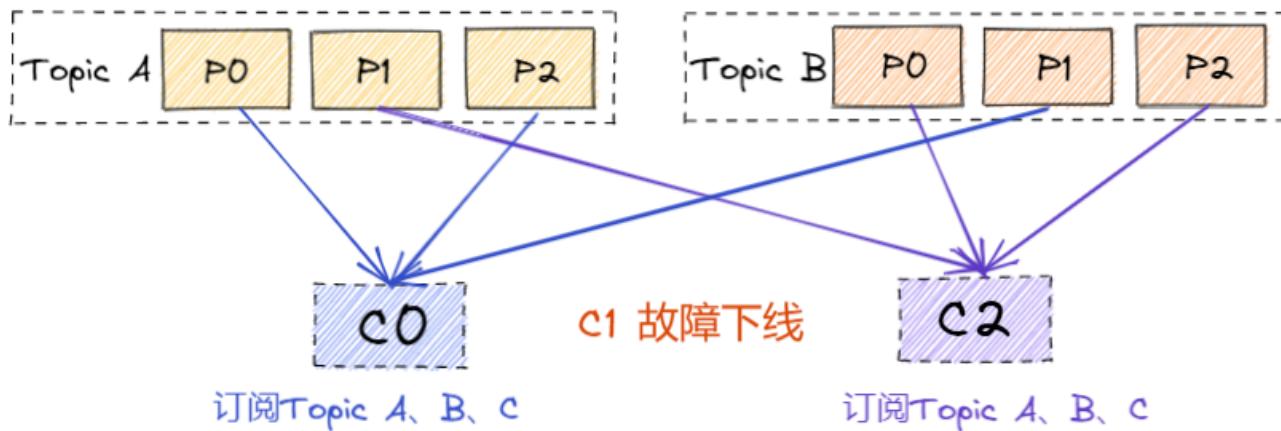
C1 : [Topic A P1, Topic B P1]

C2 : [Topic A P2, Topic B P2]

华仔聊技术

当发生 Rebalance 情况后，可能分配会不太一样，假如这时候C1发生故障下线：

RoundRobinAssignor:



2个Topic 6个Partition, 2个Consumer且订阅Topic相同的场景

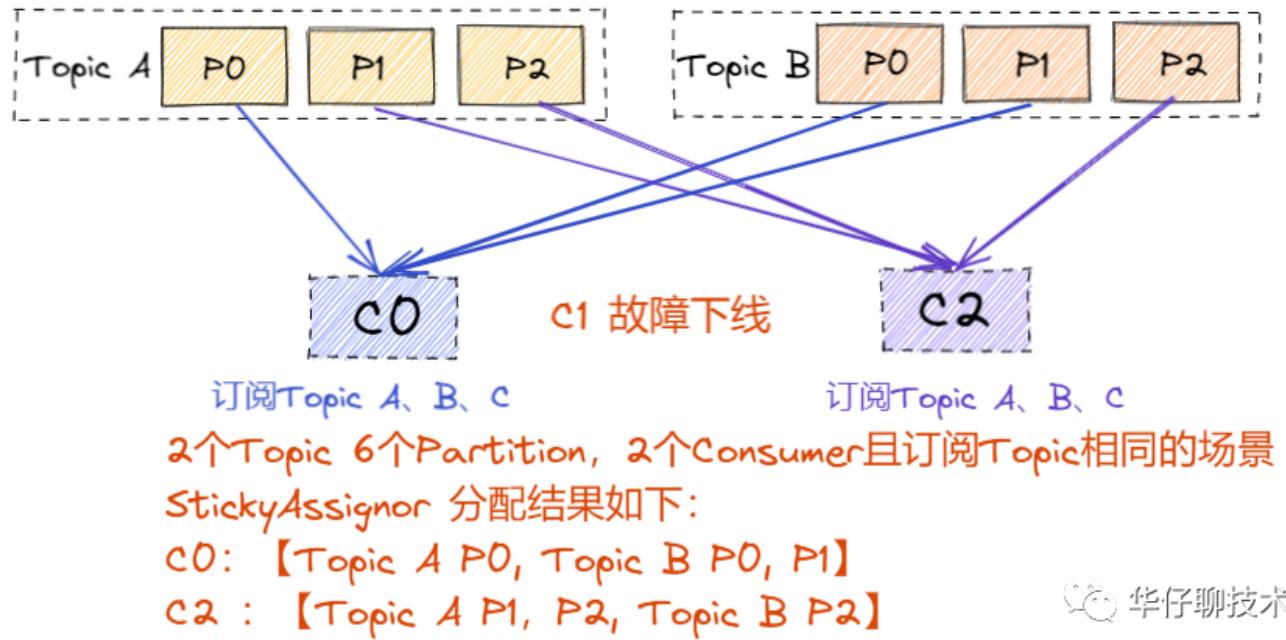
RoundRobinAssignor 分配结果如下：

C0: [Topic A P0, P2, Topic B P1]

C2 : [Topic A P1, Topic B P0, P2]

华仔聊技术

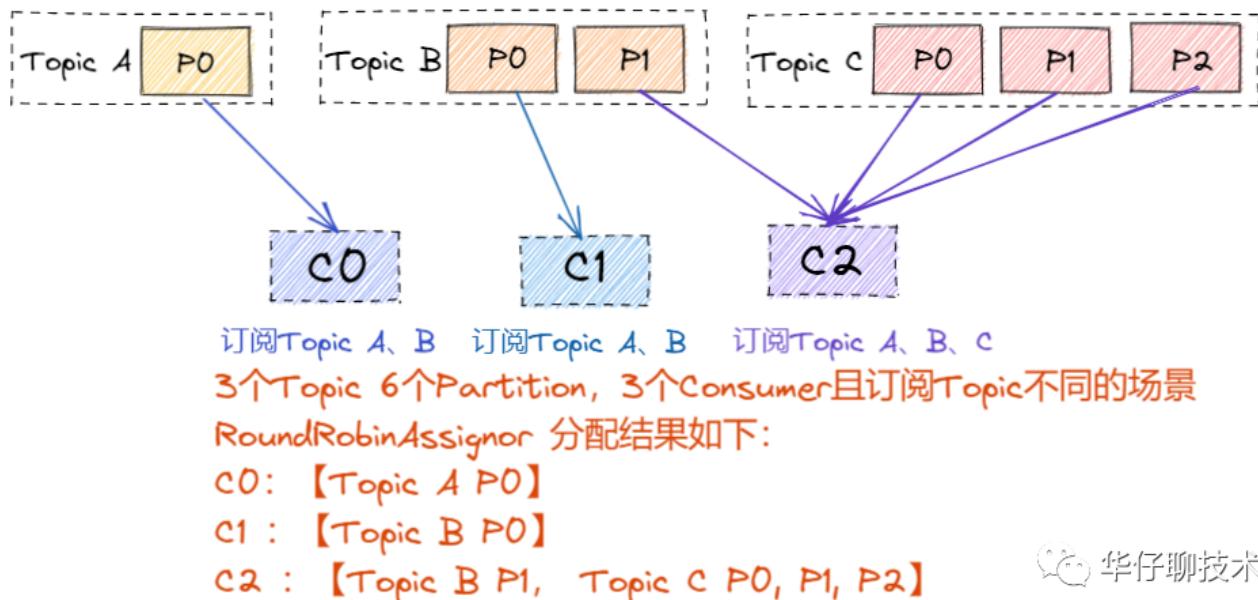
StickyAssignor:



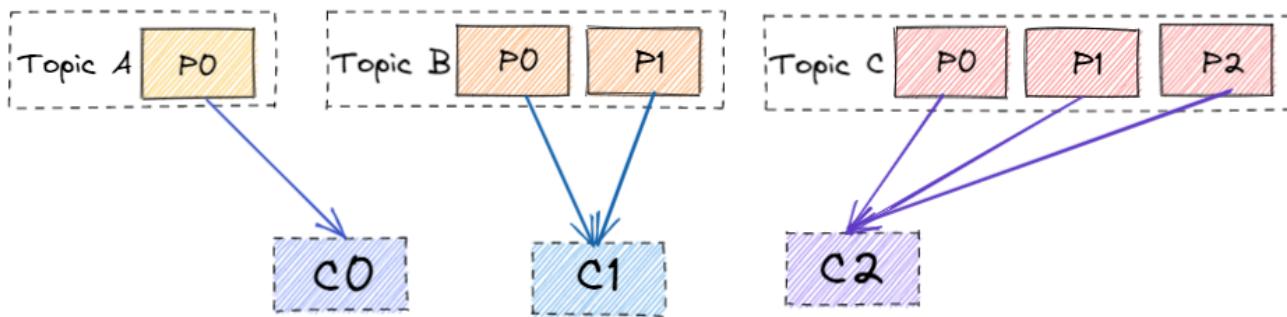
结论: 从上面 Rebalance 发生后的结果可以看出, 虽然两种分配策略最后都是均匀分配的, 但是 RoundRobinAssignor 分区分配策略 完全是重新分配了一遍, 而 StickyAssignor 则是在原先的基础上达到了均匀的状态。

2) 当组内每个 Consumer 订阅的 Topic 是不同情况:

RoundRobinAssignor:



StickyAssignor:



订阅Topic A、B 订阅Topic A、B 订阅Topic A、B、C

3个Topic 6个Partition, 3个Consumer且订阅Topic不同的场景

StickyAssignor 分配结果如下:

C0: [Topic A P0]

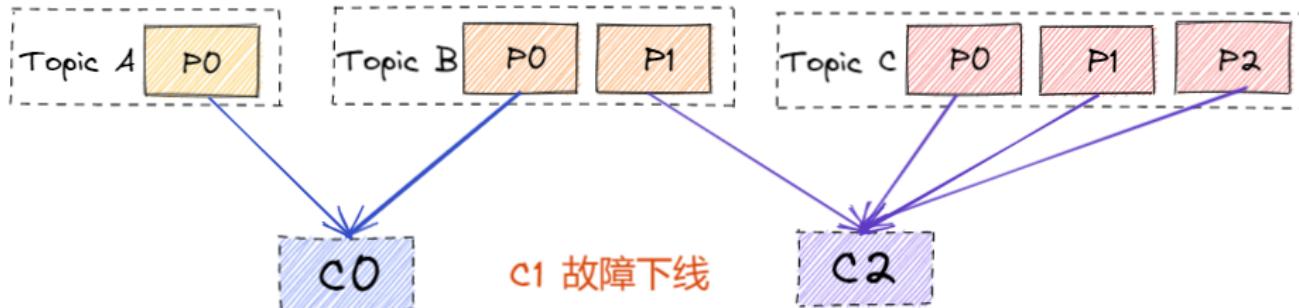
C1: [Topic B P0, P1]

C2: [Topic C P0, P1, P2]

华仔聊技术

当发生 Rebalance 情况后, 可能分配会不太一样, 假如这时候C1发生故障下线:

RoundRobinAssignor:



订阅Topic A、B

订阅Topic A、B、C

3个Topic 6个Partition, 2个Consumer且订阅Topic不同的场景

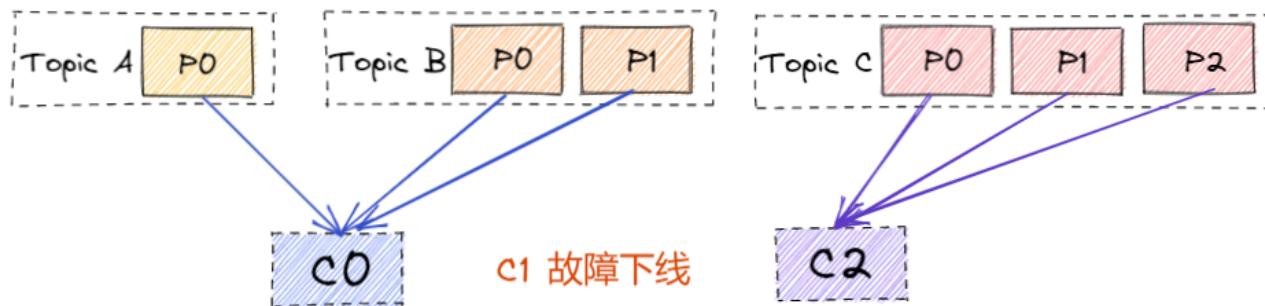
RoundRobinAssignor 分配结果如下:

C0: [Topic A P0, Topic B P0]

C2: [Topic B P1, Topic C P0, P1, P2]

华仔聊技术

StickyAssignor:



订阅 Topic A、B
3个Topic 6个Partition, 2个Consumer且订阅Topic不同的场景
StickyAssignor 分配结果如下:
C0: [Topic A P0, Topic B P0, P1]
C2 : [Topic C P0, P1, P2]

华仔聊技术

从上面结果可以看出, RoundRobin 的分配策略在 Rebalance 之后造成了严重的分配倾斜。因此在生产环境上如果想要减少重分配带来的开销, 可以选用 StickyAssignor 的分区分配策略。

谈谈 Kafka 消费者重平衡机制是怎样的?

所谓的消费者组的重平衡目的就是让组内所有的消费者实例对消费哪些主题分区达成一致。

对于 Consumer Group 来说, 可能随时都会有 Consumer 加入或退出, 那么 Consumer 列表的变化必定会引起 Partition 的重新分配。我们将这个分配过程叫做 Consumer Rebalance, 但是这个分配过程需要借助 Broker 端的 Coordinator 协调者组件, 在 Coordinator 的帮助下完成整个消费者组的分区重分配, 也是通过监听ZooKeeper 的 /admin/reassign_partitions 节点触发的。

Rebalance 触发与通知 01

Rebalance 的触发条件有三种:

- 1) 当 Consumer Group 组成员数量发生变化(主动加入或者主动离组, 故障下线等)。
- 2) 当订阅主题数量发生变化。
- 3) 当订阅主题的分区数发生变化。

Rebalance 触发后如何通知其他 Consumer 进程?

Rebalance 的通知机制就是靠 **Consumer** 端的心跳线程，它会定期发送心跳请求到 Broker 端的 Coordinator 协调者组件，当协调者决定开启 Rebalance 后，它会将「**REBALANCE_IN_PROGRESS**」封装进心跳请求的响应中发送给 Consumer，当 Consumer 发现心跳响应中包含了「**REBALANCE_IN_PROGRESS**」，就知道是 Rebalance 开始了。

Rebalance 协议说明 02

其实 Rebalance 本质上也是一组协议，Consumer Group 与 Coordinator 共同使用它来完成 Consumer Group 的 Rebalance。

下面我看看这5种协议完成了什么功能：

- 1) **Heartbeat 请求**: Consumer 需要定期给 Coordinator 发送心跳来证明自己还活着。
- 2) **LeaveGroup 请求**: 主动告诉 Coordinator 要离开 Consumer Group。
- 3) **SyncGroup 请求**: Group Leader Consumer 把分配方案告诉组内所有成员。
- 4) **JoinGroup 请求**: 成员请求加入组。
- 5) **DescribeGroup 请求**: 显示组的所有信息，包括成员信息，协议名称，分配方案，订阅信息等。通常该请求是给管理员使用。

Coordinator 在 Rebalance 的时候主要用到了前面4种请求。

Consumer Group 状态机 03

如果 Rebalance 一旦发生，就会涉及到 Consumer Group 的状态流转，此时 Kafka 为我们设计了一套完整的状态机机制，来帮助 Broker Coordinator 完成整个重平衡流程。

了解整个状态流转过程可以帮助我们深入理解 Consumer Group 的设计原理。5种状态，定义分别如下：

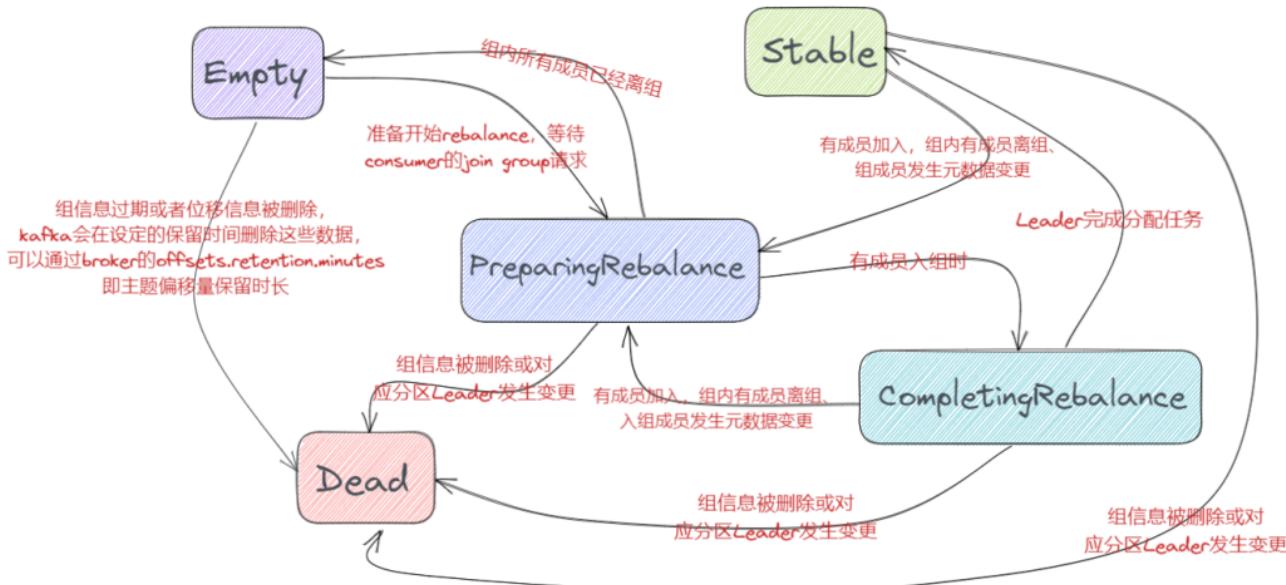
- 1) **Empty 状态**: 表示当前组内无成员，但是可能存在 Consumer Group 已提交的位移数据，且未过期，这种状态只能响应 JoinGroup 请求。。
- 2) **Dead 状态**: 表示组内已经没有任何成员的状态，组内的元数据已经被 Broker Coordinator 移除，这种状态响应各种请求都是一个Response: UNKNOWN_MEMBER_ID。

3) PreparingRebalance 状态：表示准备开始新的 Rebalance，等待组内所有成员重新加入组内。

4) CompletingRebalance 状态：表示组内成员都已经加入成功，正在等待分配方案，旧版本中叫「AwaitingSync」。

5) Stable 状态：表示 Rebalance 已经完成，组内 Consumer 可以开始消费了。

5种状态流转图如下：



华仔聊技术

Rebalance 流程分析 04

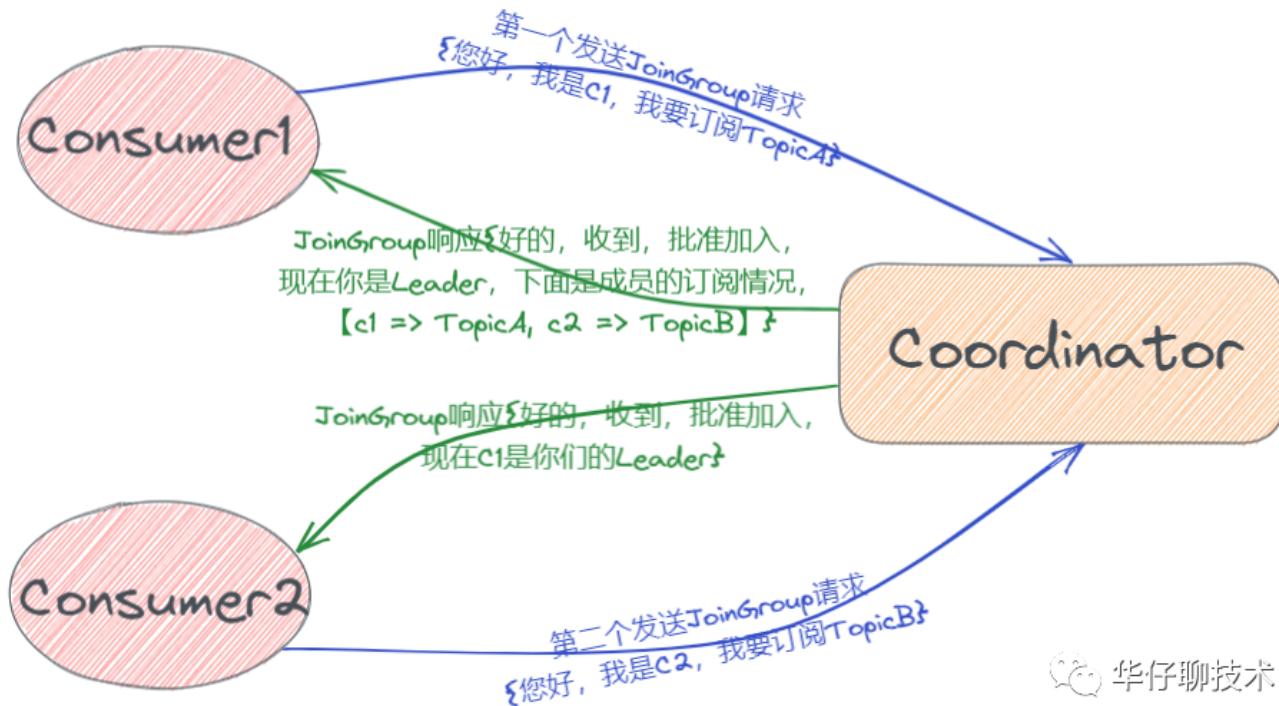
通过上面5种状态可以看出，Rebalance 主要分为两个步骤：加入组「JoinGroup 请求」和等待 Consumer 分配方案「SyncGroup 请求」。

JoinGroup 请求

组内所有成员向 Coordinator 发送 JoinGroup 请求，请求加入组，顺带会上报自己订阅的 Topic，这样 Coordinator 就能收集到所有成员的 JoinGroup 请求和订阅 Topic 信息，Coordinator 就会从这些成员中选择一个担任这个 Consumer Group 的 Leader 「一般情况下，第一个发送请求的 Consumer 会成为 Leader」。

这里说的 Leader 是指具体的某一个 Consumer，它的任务就是收集所有成员的订阅 Topic 信息，然后制定具体的消费分区分配方案。待选出 Leader 后，Coordinator 会把 Consumer Group 的订阅 Topic 信息封装进

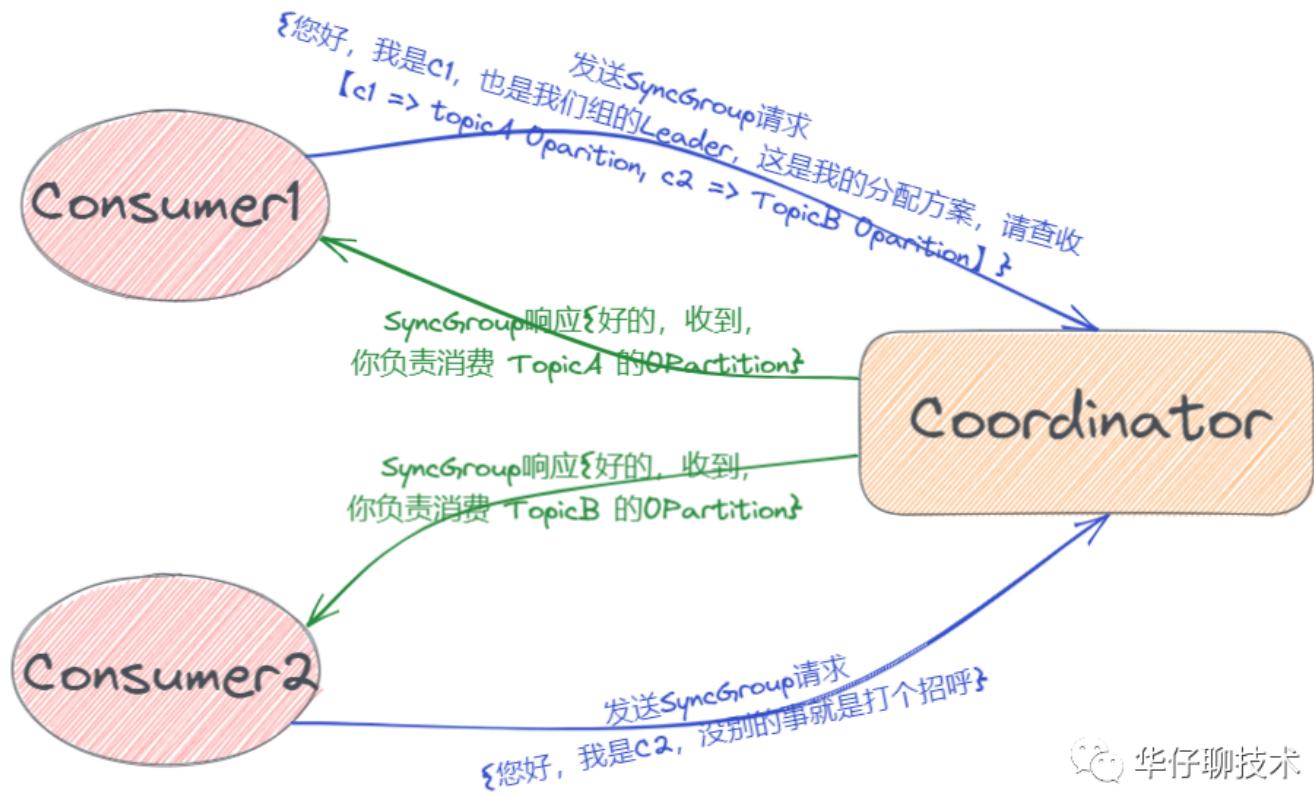
JoinGroup 请求的 Response 中，然后发给 Leader，然后由 Leader 统一做出分配方案后，进入到下一步



SyncGroup 请求

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 会将这个分配方案封装进 SyncGroup 请求中发给 Coordinator，其他成员也会发 SyncGroup 请求，只是内容为空，待 Coordinator 接收到分配方案之后会把方案封装进 SyncGroup 的 Response 中发给组内各成员，这样各自就知道应该消费哪些 Partition 了。



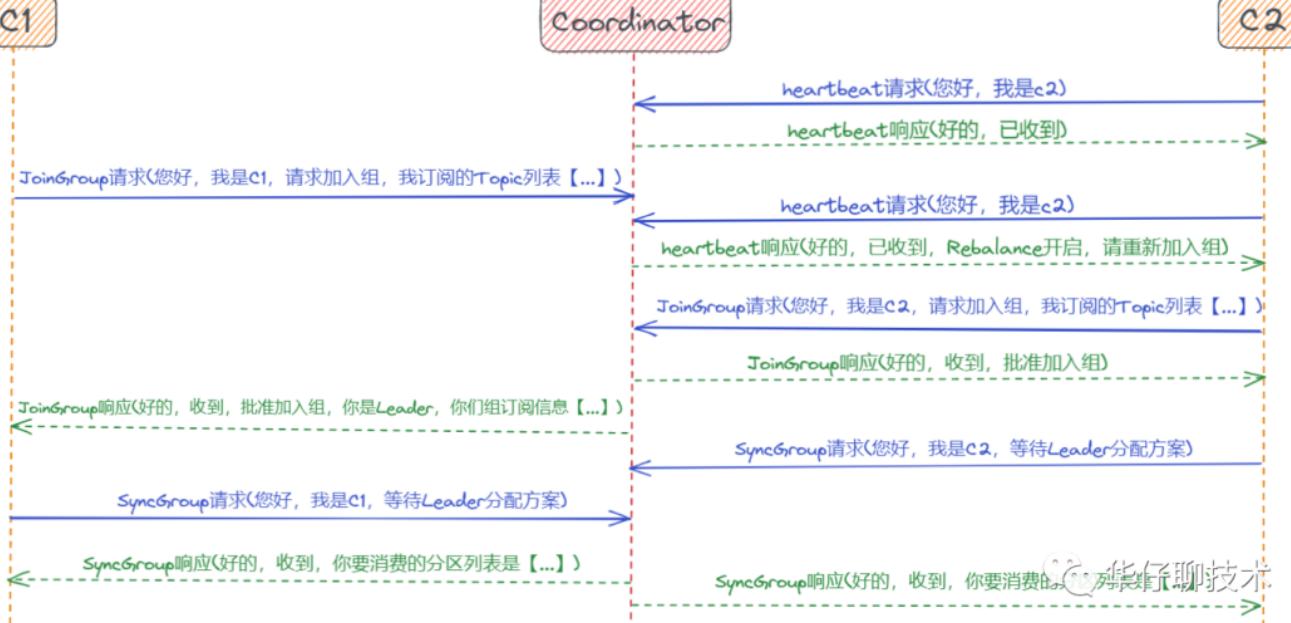
华仔聊技术

Rebalance 场景分析 05

刚刚详细的分析了关于 Rebalance 的状态流转，接下来我们通过时序图来重点分析几个场景来加深对 Rebalance 的理解。

场景一：新成员(c1)加入组

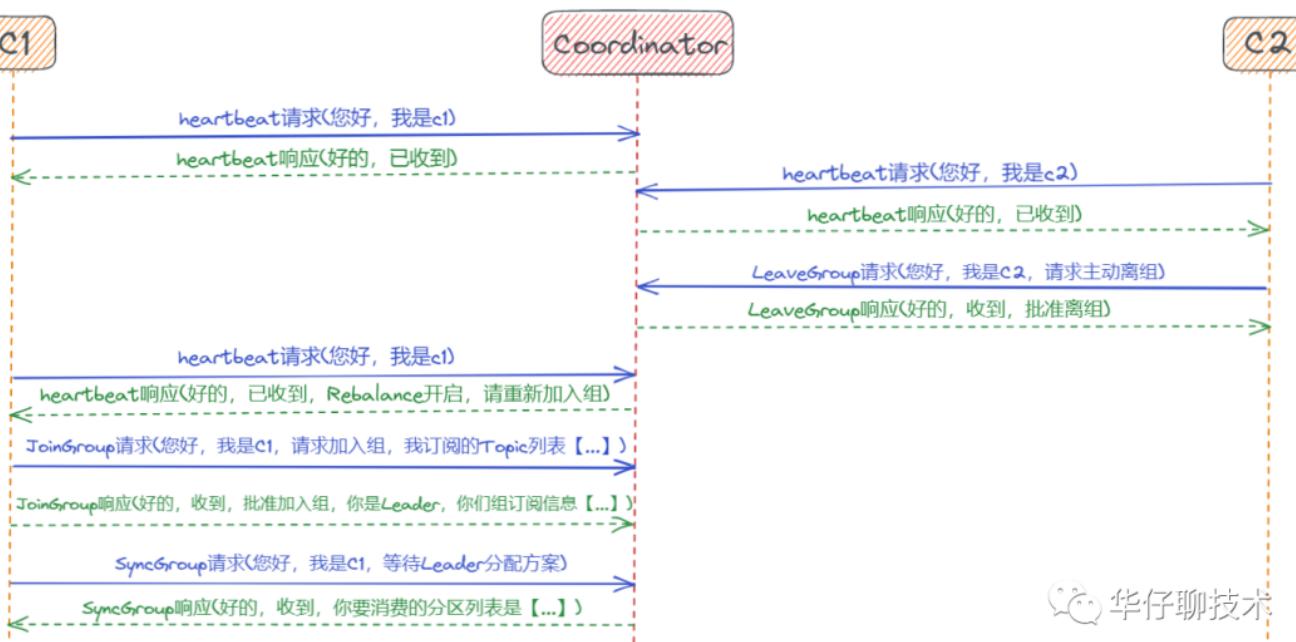
这里新成员加入组是指组处于 Stable 稳定状态后，有新成员加入的情况。当协调者收到新的 JoinGroup 请求后，它会通过心跳机制通知组内现有的所有成员，强制开启新一轮的重平衡。



场景二：成员(c2)主动离组

这里主动离组是指消费者所在线程或进程调用 `close()` 方法主动通知协调者它要退出。当协调者收到 `LeaveGroup` 请求后，依然会以心跳机制通知其他成员，强制开启新一轮的重平衡。

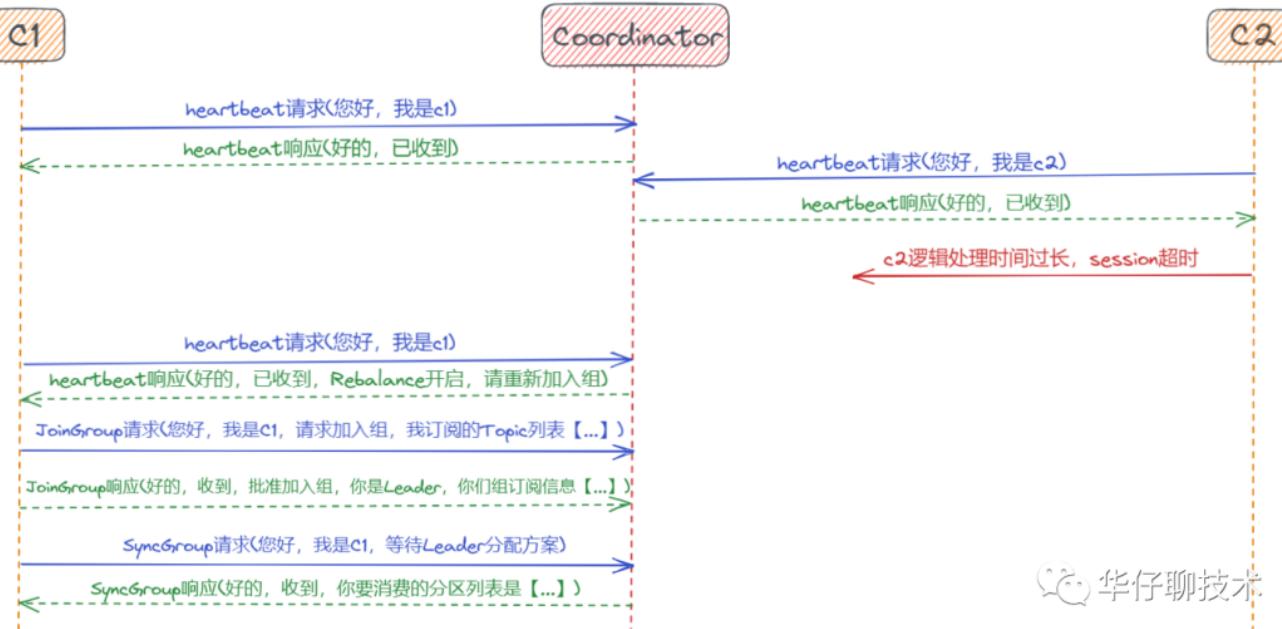
成员(c2)主动离组



场景三：成员(c2)超时被踢出组

这里超时被踢出组是指消费者实例出现故障或者处理逻辑耗时过长导致的离组。此时离组是被动的，协调者需要等待一段时间才能感知到，一般是由消费者端参数 `session.timeout.ms` 控制的。

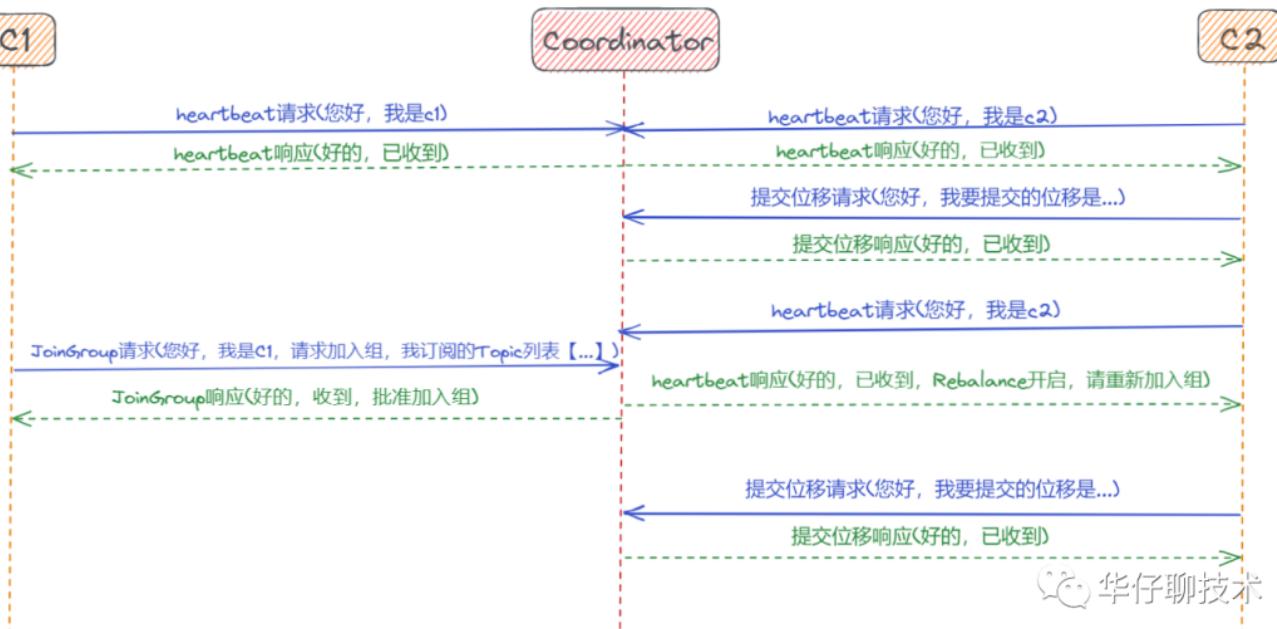
成员(c2)超时被踢出组



场景四：成员(c2)提交位移数据

当重平衡开启时，协调者会要求组内成员必须在这段缓冲时间内快速地提交自己的位移信息，然后再开启正常的JoinGroup/SyncGroup 请求发送。

成员(c2)提交位移数据



谈谈Kafka线上大量消息积压你是如何处理的？

消息大量积压这个问题，直接原因一定是某个环节出现了性能问题，来不及消费消息，才会导致消息积压。接着就比较坑爹了，此时假如 Kafka 集群的磁盘都快写满了，都没有被消费，这个时候怎么办？或者消息积压了几个小时，这个时候怎么办？生产环境挺常见的问题，一般不出问题，而一出就是大事故。

所以，我们先来分析下，在使用 Kafka 时如何来优化代码的性能，避免出现消息积压。如果你的线上 Kafka 系统出现了消息积压，该如何进行紧急处理，最大程度地避免消息积压对业务的影响。

优化性能来避免消息积压

01

对于 Kafka 性能的优化，主要体现在生产者和消费者这两部分业务逻辑中。而 Kafka 本身不需要多关注的主要原因是，对于绝大多数使用 Kafka 的业务来说，Kafka 本身的处理能力要远大于业务系统的处理能力。Kafka 单个节点，消息收发的性能可以达到每秒钟处理几十万条消息的水平，还可以通过水平扩展 Broker 的实例数成倍地提升处理能力。

对于业务系统处理的业务逻辑要复杂一些，单个节点每秒钟处理几百到几千次请求，已经非常不错了，所以我们应该更关注的是消息的收发两端。

1. 发送端性能优化

发送端即生产者业务代码都是先执行自己的业务逻辑，最后再发送消息。**如果说性能上不去，需要你优先检查一下，是不是发消息之前的业务逻辑耗时太多导致的。**

对于发送消息的业务逻辑，只需要注意设置合适的并发和批量大小，就可以达到很好的发送性能。我们知道 Producer 发消息给 Broker 且收到消息并返回 ack 响应，假设这一次过程的平均时延是 1ms，它包括了下面这些步骤的耗时：

- 1) 发送端在发送网络请求之前的耗时。
- 2) 发送消息和返回响应在网络传输中的耗时。
- 3) Broker 端处理消息的时延。

假设此时你的发送端是单线程，每次只能发送 1 条消息，那么每秒只能发送 1000 条消息，这种情况下并不能发挥出 Kafka 的真实性能。此时无论是增加每次发送消息的批量大小，还是增加并发，都可以成倍地提升发送性能。

如果当前发送端是在线服务的话，比较在意请求响应时延，此时可以采用并发方式来提升性能。

如果当前发送端是离线服务的话，更关注系统的吞吐量，发送数据一般都来自数据库，此时更适合批量读取，批量发送来提升性能。

另外还需要关注下消息体是否过大，如果消息体过大，势必会增加 IO 的耗时，影响 Kafka 生产和消费的速度，也可能会造成消息积压。

2. 消费端性能优化

而在使用 Kafka 时，大部分的性能问题都出现在消费端，如果消费的速度跟不上发送端生产消息的速度，就会造成消息积压。如果只是暂时的，那问题不大，只要消费端的性能恢复之后，超过发送端的性能，那积压的消息是可以逐渐被消化掉的。

要是消费速度一直比生产速度慢，时间长了系统就会出现问题，比如 Kafka 的磁盘存储被写满无法提供服务，或者消息丢失，对于整个系统来说都是严重故障。

所以我们在设计的时候，**一定要保证消费端的消费性能要高于生产端的发送性能，这样的系统才能健康的持续运行。**

消费端的性能优化除了优化业务逻辑外，也可以通过水平扩容，增加消费端的并发数来提升总体的消费性能。需要注意的是，在扩容 Consumer 的实例数量的同时，必须同步扩容主题中的分区数量，确保 Consumer 的实例数和分区数量是相等的，如果 Consumer 的实例数量超过分区数量，这样的扩容实际上是没有效果的。

消息积压后如何处理 02

日常系统正常时候，没有积压或者只有少量积压很快就消费掉了，但某时刻，突然开始积压消息且持续上涨。这种情况下需要你在短时间内找到消息积压的原因，迅速解决问题。

导致消息积压突然增加，有两种：发送变快了或者消费变慢了。

假如赶上大促或者抢购时，短时间内不太可能优化消费端的代码来提升消费性能，**此时唯一的办法是通过扩容消费端的实例数来提升总体的消费能力**。如果短时间内没有足够的服务器资源进行扩容，只能降级一些不重要的业务，减少发送方发送的数据量，最低限度让系统还能正常运转，保证重要业务服务正常。

假如通过内部监控到消费变慢了，需要你检查消费实例，分析一下是什么原因导致消费变慢？

1、优先查看日志是否有大量的消费错误。

2、此时如果没有错误的话，可以通过打印堆栈信息，看一下你的消费线程是不是卡在哪里「**触发死锁或者卡在某些等待资源**」。

最后说一句，欢迎加苏三技术交流群，里面大佬很多，关键是有许多技术资料分享。



Lzh

深入理解 Java 虚拟机：JVM...特性与最



28.32MB



Lzh

分享一个电子书



Lzh

里面文字能直接复制的，不是一个
个图片

2月17日 08:03



Lzh

nacos 架构与原理.pdf



12.83MB



悟空



这个群真是宝藏

悟空



扫描下发二维码，回复：加群，即可拉你进群。

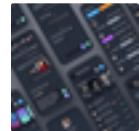


喜欢此内容的人还喜欢

MySQL通过bin log恢复数据 | 手撕MySQL | 对线面试官
程序员白泽



Jetpack Compose 写一个聊天 App，附赠源码（上）
字节数组



瞧瞧人家，那后端API接口写得，那叫一个优雅！
首席架构师专栏

