Kafka 消费者 Rebalance 的性能优化

概述

负载均衡是分布式系统中永恒的话题，在 Kafka 中也不例外。这篇文章将会简单介绍一下快手 Kafka 在消费者端的负载均衡，即 Consumer Group Rebalance 过程中遇到的性能问题，以及解决方案和具体的收益。

首先简单介绍一下相关概念。在 Kafka 中，主要角色可分为消息（Message）、生产者（Producer）、消费组（Consumer Group）这三大类。由生产者进行消息的生产，由消费组进行消息的订阅（Subscribe）和消费。在 Kafka 中，消息以主题（Topic）的格式进行组织，而 Topic 中包含多个分区（Partition），并且这些消息都存储在服务端的 broker 中。消费组则由若干个消费者组成。即从微观上来说，一个消费组内存在着若干组 Consumer - Partition 的对应关系，而在一定条件下，这个对应关系被破坏，此时由于需要保证消费组能够消费到其所订阅的所有消息，需要进行 **Consumer - Partition 对应关系的重新调整**，在 Kafka 中，这个重新调整对应关系的过程称为 Consumer Group Rebalance，简称 Rebalance。

在 Kafka 的服务端，存在一个名为 Group Coordinator 的角色，其作用是管理 Consumer Group，每一个 Group 对应一个 Server 端的 Coordinator。Coordinator 通过接收 Group 内 Consumer 定时发送的心跳来判断这个 Consumer 是否存活。在 Kafka 中，Rebalance 通常会发生在以下几种情况：

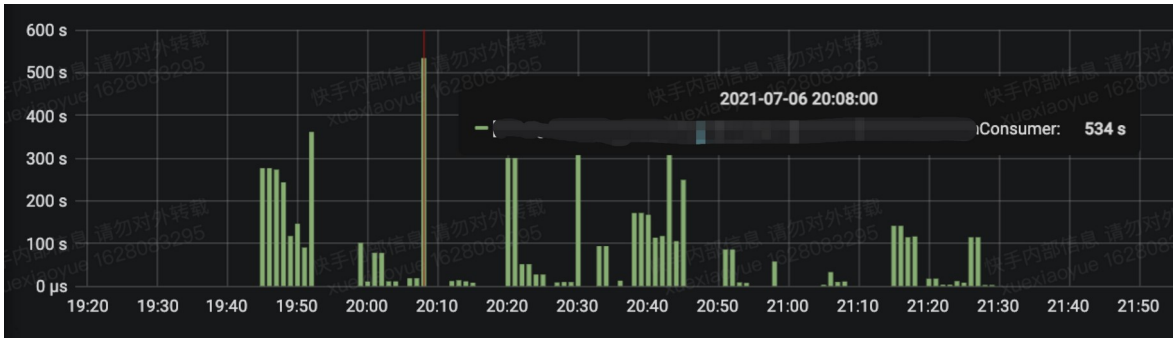
1. Consumer 与 Coordinator 心跳超时，Coordinator 将其移出 Group，按经验较常见的情况有网络抖动、宿主机故障、宿主机负载过高、持续 Full GC 等。
2. Consumer 主动加入或主动离开 Group，通常会发生在批量滚动重启操作中
3. Consumer 消费过慢，长时间未进行 poll() 操作，超过 max.poll.interval.ms 主动离开 Group 引发 Rebalance

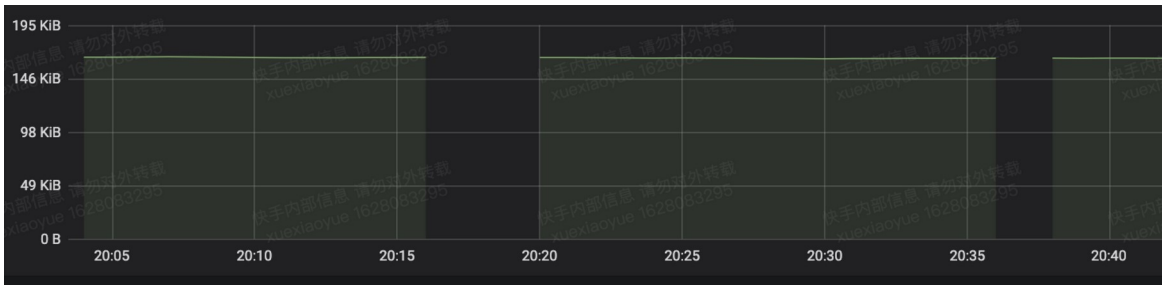
Kafka Rebalance 的性能问题

通常情况下，我们认为 Kafka 的 Rebalance 是一个代价很高的过程，为了保证其在 Rebalance 前后消费不产生重复与丢失，不可避免的会出现 **stop-the-world 现象**（与一些具有垃圾回收功能的编程语言类似），即一段时间内 Consumer 停止一切消费，等待具体的重分配方案。

随着业务的扩展，不可避免的会出现一些超大流量的 Consumer Group（Partition>2000，Consumer>2000）。这些 Consumer Group 一旦出现宿主机故障，例如容器云中游离的异常节点或负载过高的节点，并且未及时摘除时，便可能引发大规模的持续

Rebalance，如下图所示，从 19:45 开始频繁发生 Rebalance 过程，其中多次都在 300s 以上，最长的达到了 524 秒，在如此长时间的 Rebalance 过程中，消费者的暂停消费会对业务侧应用造成严重的影响，如下图所示，从监控可以看出曲线发生剧烈抖动甚至造成断流。





异步 Server Rebalance 带来的收益

通过对快手某核心业务的数个 Consumer Group 进行新版异步 Server Rebalance 的切换后，我们观察到了明显的收益，选取两个 Group 进行具体的对比展示。需要注意的是下述 Rebalance 耗时指的是 Rebalance 过程中耗时最长的 Consumer Client 感知到的耗时。

Group1：
partition：1600
写入流量：30+G/s

Consumer 个数：1600+

	Rebalance 耗时(ms)								平均耗时(ms)
原生	3021	2989	2996	4433	2428	2892	3076	3126	3120.12
改进版	17	26	51	14	20	24	17	36	25.63

Group2：

partition：3000

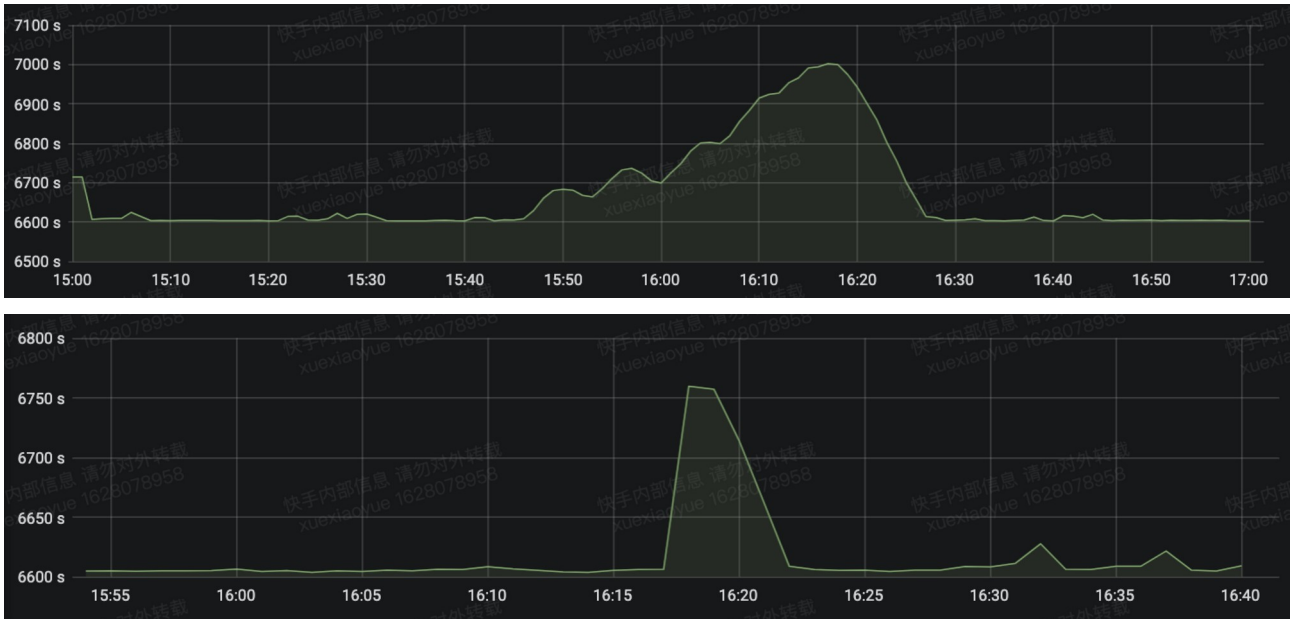
写入流量：30+G/s

Consumer 个数：3000+

	Rebalance 耗时(ms)								平均耗时(ms)
原生	6040	5340	9040	6790	9650	6230	7920	5680	7086.25
改进版	102	14	34	22	48	7.8	23	67	39.73

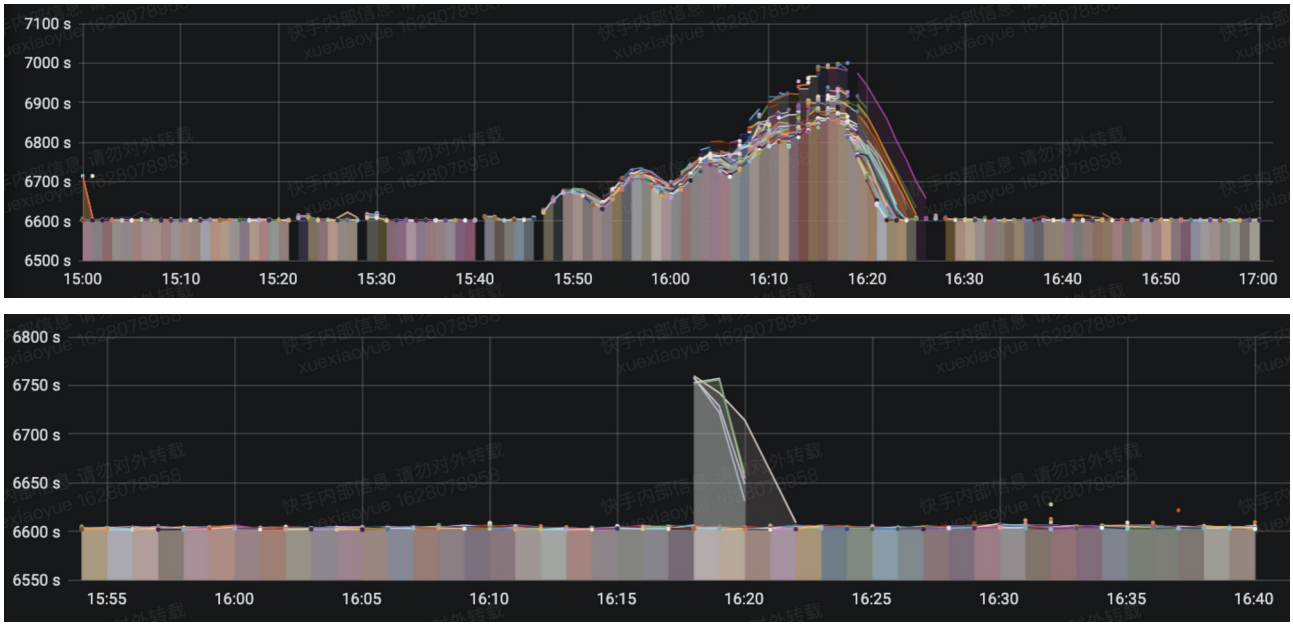
从表格中的数据可以明显看出切换异步 Server Rebalance 后在耗时方面的提升，耗时最长的 Consumer 平均可以减少 100 倍以上的时间。

下面再来从 Lag Time 来观察一下 Rebalance 过程中产生消息积压的情况，选用上述 Group2 进行观察切换异步 Server Rebalance 前后对比，得到以下监控记录。

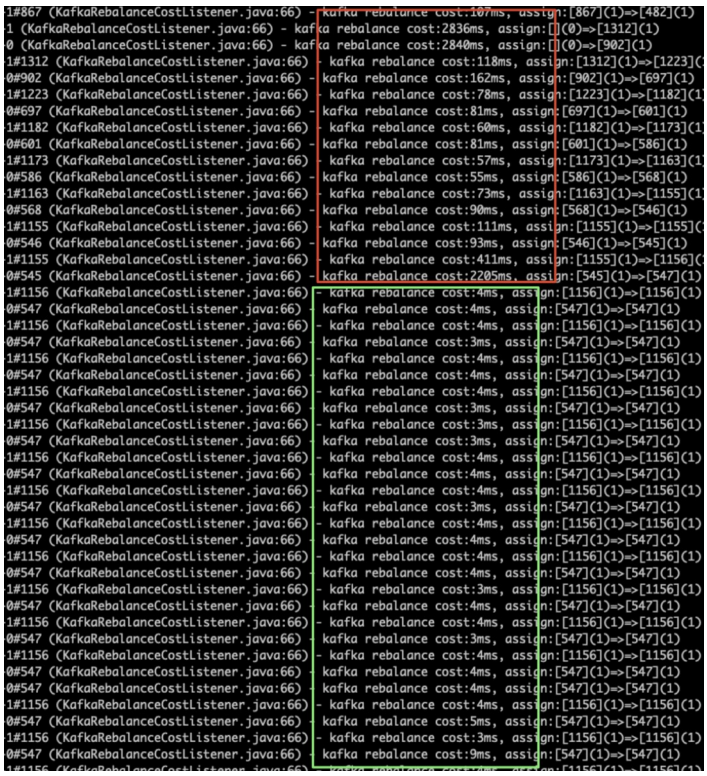


从以上两图中我们可以看出，对于原生 Kafka Rebalance 时，Lag Time 从 6600s 上升至 7000s 左右，并且整体上升以及恢复时间较长，约为 15:45-16:25，耗时 40min 才恢复正常。

对比切换异步 Server Rebalance 后的 Rebalance 过程来看，切换后 Lag Time 从 6600s 上升至 6760s 左右，相较于原生的 Lag Time 有着明显的降低，并且整体持续时间较短，约为 16:17-16:22，耗时约 5min 就恢复正常。



切换至 Partition 维度的监控，我们从上图不难看出，原生的 Kafka Rebalance 过程中，所有的 Partition 都参与了 Rebalance 的过程，导致所有 Partition 的 Lag Time 都出现了大幅度的上涨。而对比异步 Server Rebalance 的过程，只有少数几个 Partition 感知到了 Rebalance 过程，其余 Partition 的 Lag Time 与正常消费时无变化。



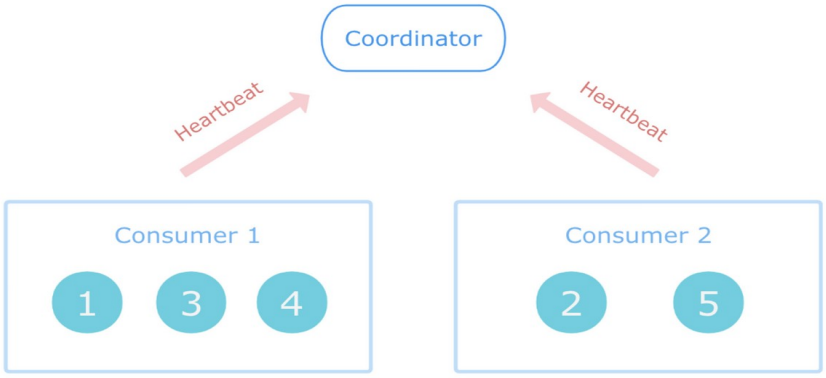
从上图业务侧提供的某两次客户端日志中也可以看出，切换异步 Server Rebalance 后客户端对与 Rebalance 的感知时间有着明显的减少，并且从右侧 assign 结果可以看出，原生 Rebalance 很容易会将几乎所有的 Partition 重新分配，而新版异步 Server Rebalance 则将大多数分区分配方案保持不变，只有极少数感知到分区方案的变化。

部署

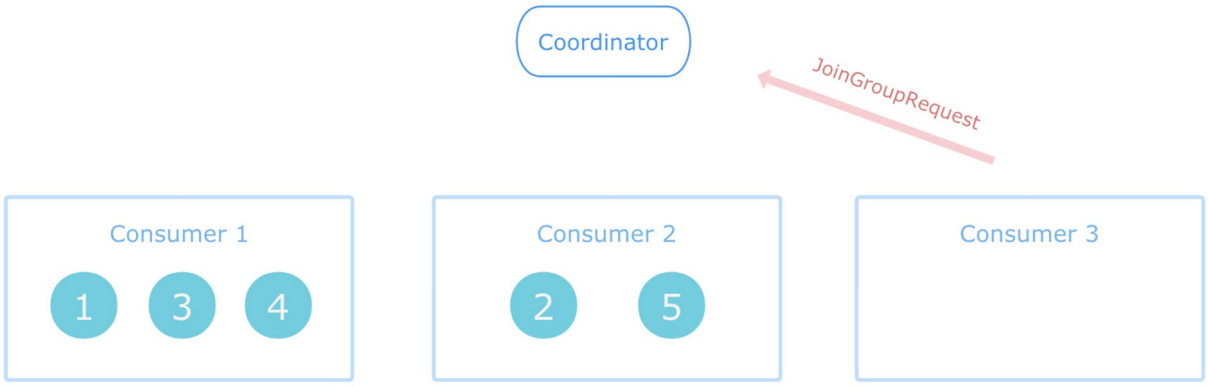
目前公司 Kafka 客户端默认使用的方式还是原生的 rebalance 方式，若需要开启新 Server rebalance 方式需确保为 Java 客户端进行消费，其它语言客户端暂不支持。更多详情可咨询 Kafka oncall

原生 Kafka Rebalance 流程简述

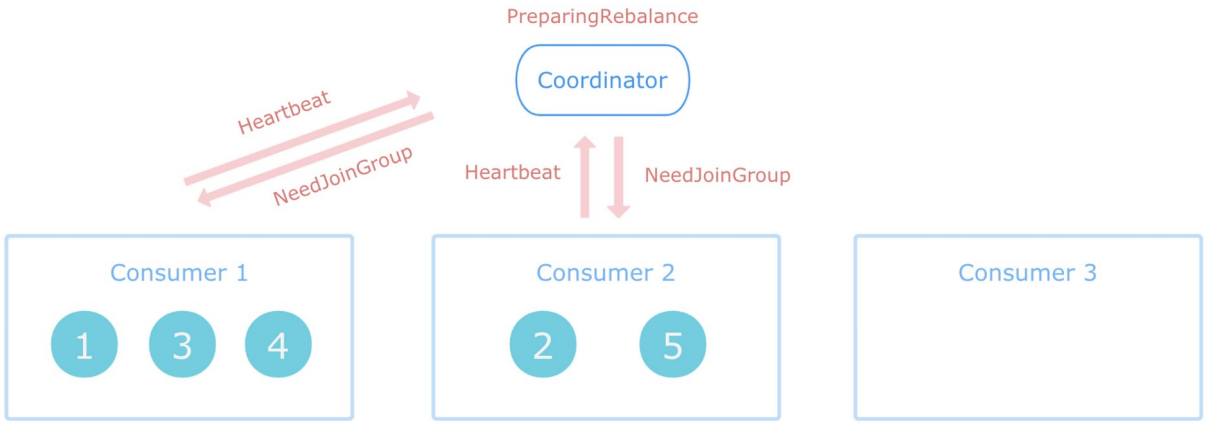
现在让我们用一个实际的例子来具体看一下 Kafka 在原生的 Rebalance 过程中发生了什么。假设目前有一个 2 个 Consumer 的 Consumer Group 以及 5 个 Partition，Consumer1 被分配了 Partition1、3、4，Consumer2 被分配了 Partition2、5。两个 Consumer 通过定时心跳来通知 Coordinator 自己当前的存活状态。



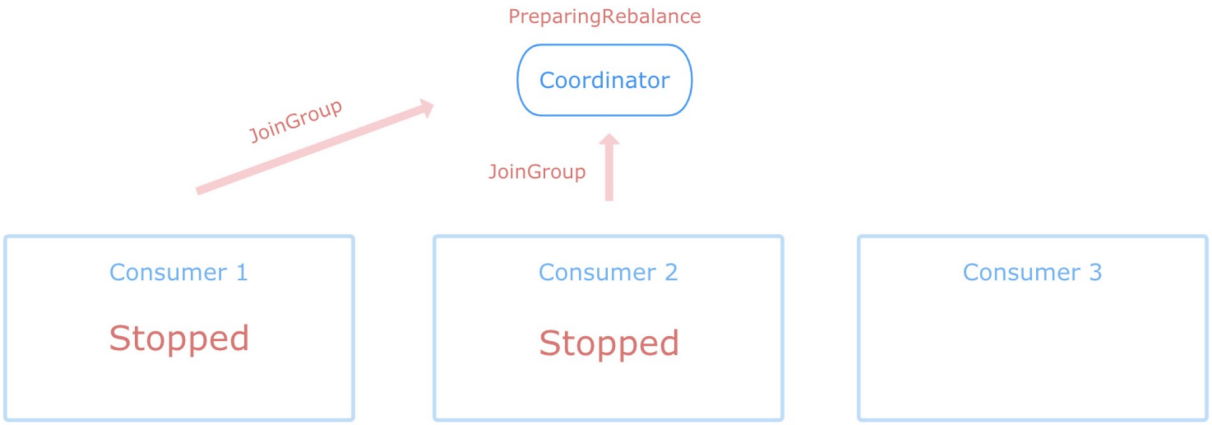
此时有一个新的 Consumer3 准备加入该 Group，Consumer3 向 Coordinator 发送一个 JoinGroup 请求。



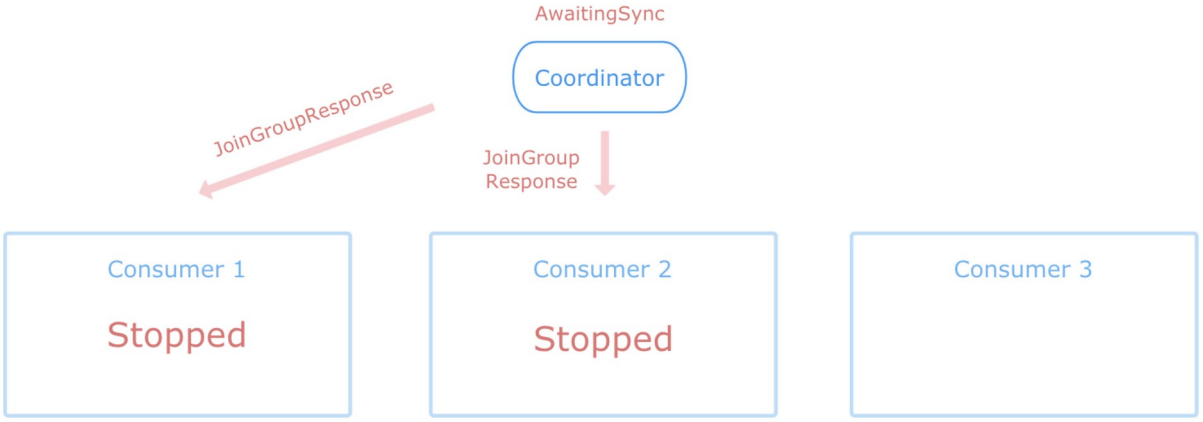
Step1：Coordinator 收到 Consumer3 的 JoinGroup 请求后，将该 group 状态标记为 PreparingRebalance，表示准备开始 Rebalance 操作。此后该 Group 内的成员向 Coordinator 发送心跳时，Coordinator 就会告知该 Consumer 需要发送一个 JoinGroup 请求来重新加入该 group。



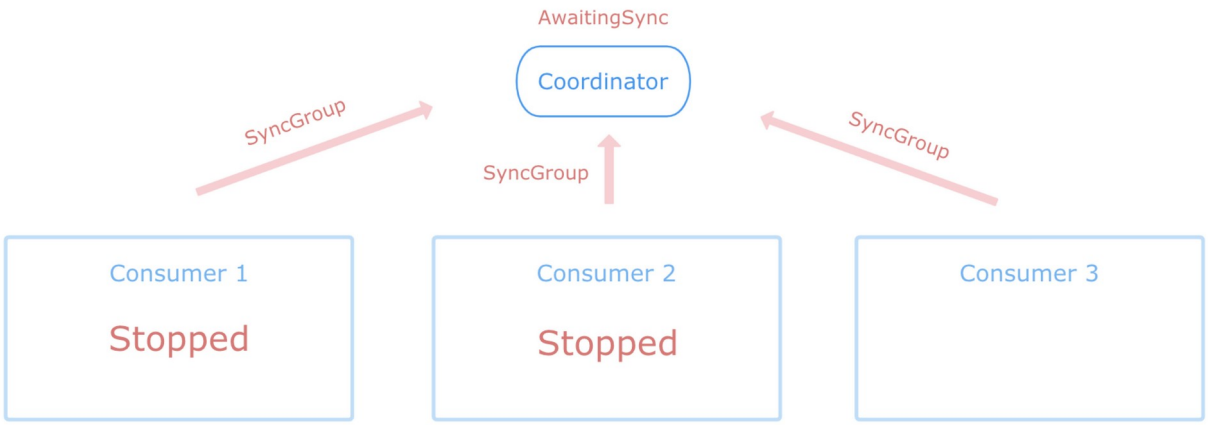
Step2：Consumer1 和 Consumer2 被 Coordinator 告知需要发送 JoinGroup 请求，这两个 Consumer 需要**立刻放弃当前所订阅的所有 Partition**，并向 Coordinator 发送 JoinGroup 请求。从这个时刻开始，该 Consumer Group 中所有 Consumer 就停止消费，**进入 stop-the-world 状态**。



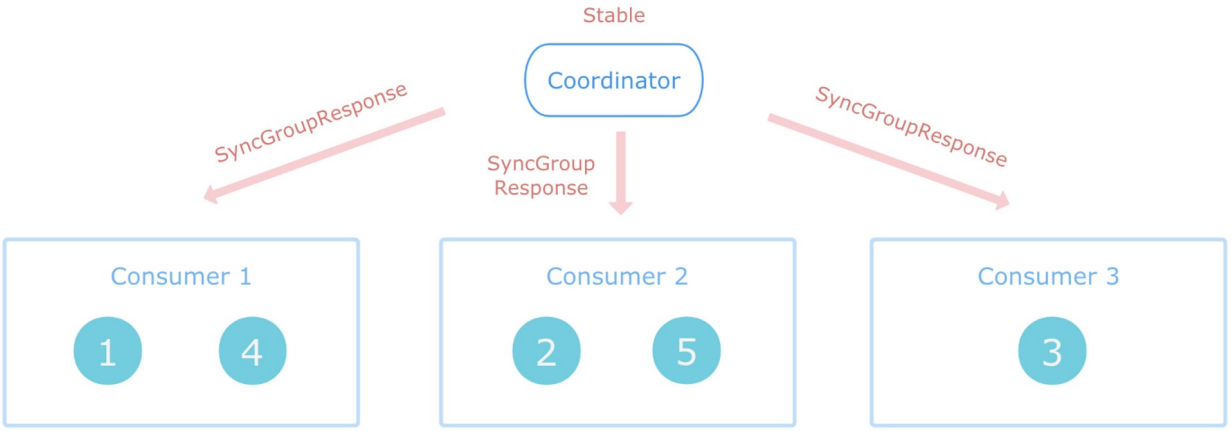
Step3：Coordinator 端在**收到组内所有 Consumer 的 JoinGroup 请求后**，会选出一个 Leader Consumer，将组内所有 Consumer 的订阅策略以及一些元数据信息返回给该 Leader Consumer，**由该 Leader Consumer 进行具体的分区分配**，而不是在 Coordinator 端进行分配。



Step4：所有的 Consumer 发送 SyncGroup 请求给 Coordinator，其中 Leader Consumer 会将自己分配好的所有分区方案包含在这一次请求中，其它 Consumer 节点这次请求不包含实质性信息。



Step5：Coordinator 在收到 Leader Consumer 发送的所有分区分配方案后，将每个 Consumer 被分配到的分区元数据作为 SyncGroup 的 Response 返回给 Consumer，接收到该 Response 的 Consumer 即可根据分配方案恢复正常的消费。



从上述 Rebalance 的流程中，我们可以发现如下几个问题：

1. Consumer Group 内所有 Consumer 从 Step2 发送 JoinGroup 请求开始便停止一切消费，直到 Step5 中 Coordinator 返回分区方案后才恢复正常消费。由于在这个过程中 Producer 仍在不断写入，在流量较大的情况下可能会导致**严重的消息积压（Lag）**。
2. 在 Step2 - Step3 过程中，Coordinator 需要等待组内**所有的 Consumer 都发送 JoinGroup** 后才能进行下一步的选举 Leader 以及下发工作，第一个发送 JoinGroup 的 Consumer 可能需要进行较长一段时间的等待直到 Coordinator 接受到最后一个加入的 JoinGroup 请求，若出现网络波动这个等待时间则会更长，等待过程中所有消费仍是停止的。
3. 在 Step4 的 SyncGroup 过程中，若在此时出现新的 Consumer 希望发送 JoinGroup 希望加入 Group，或出现原先的 Group 成员心跳超时掉线，为保证分配结果的正确性，需要**重新触发一轮 Rebalance** 过程，由于在这个过程中所有消费者都是处于停止消费状态，出现这种情况时会**大大增长这次 Rebalance 需要的整体时间**，放大 case1 中提及的消息积压。

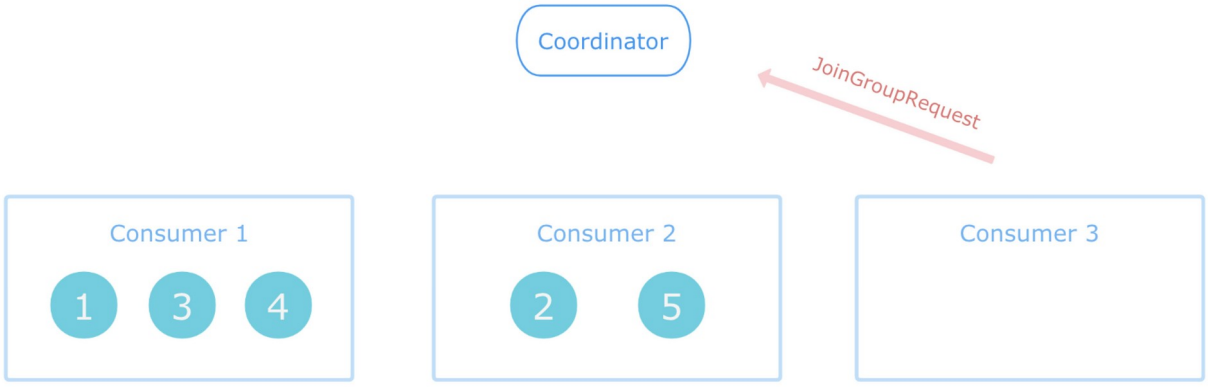
4. 注意到上述流程中，Consumer2 在 Rebalance 前和 Rebalance 后，**分配的 Partition 其实并未发生改变**。在这个例子中读者可能觉得是一个巧合，但设想在大规模的 Consumer Group 中，如包含超过 2000 Consumer 和订阅超过 2000 Partition 的一个 Group 出现新增 Consumer 或减少 Consumer，绝大多数的 Partition 分配结果应该是可以和原本的分配方案差别不大的。

基于以上问题，我们产生了以下想法：

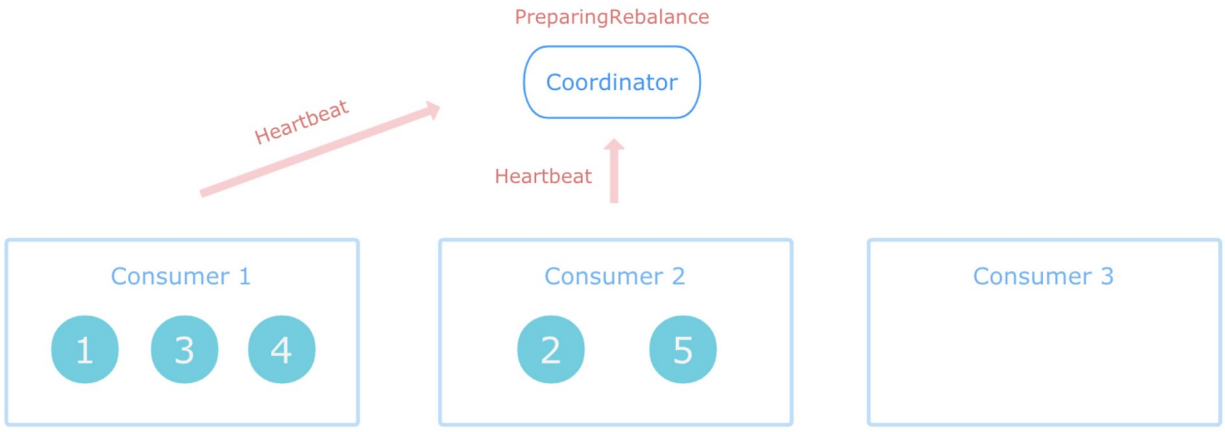
1. 是否可以由 Coordinator 端直接进行分配，在分配过程中 Consumer 仍然可以正常消费。
2. Coordinator 端进行分配时，能否尽可能的保持原先的分区分配方案，进行最少的调整，使 Rebalance 过程中影响的分区降到最低。
3. Coordinator 分配完成后，能否单独通知需要调整分区的 Consumer 进行调整，而其它没有调整的 Consumer 则照常进行消费。

异步 Server Rebalance 方案

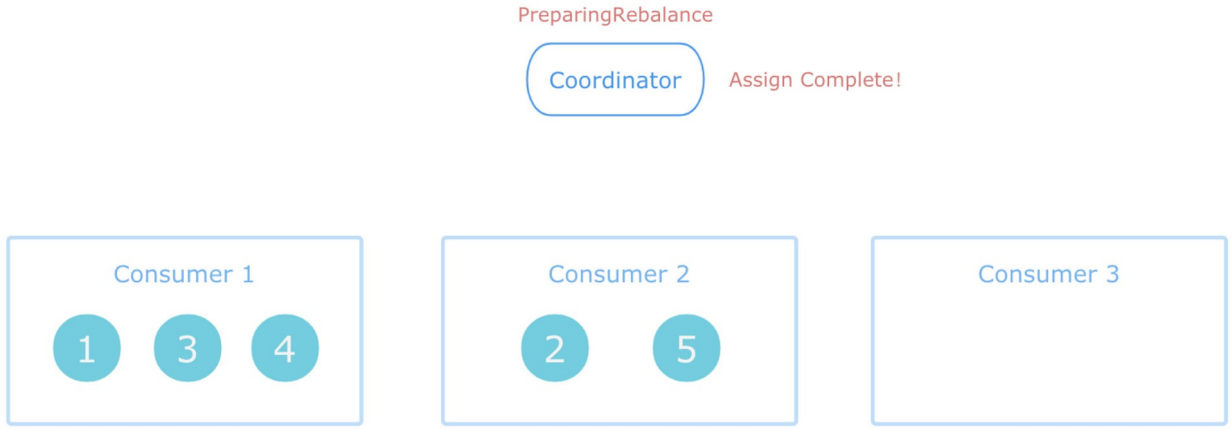
下面使用我们开发的新版异步 Server Rebalance 方案模拟一下相同的 Rebalance 流程。与上述例子中的情况相同，Consumer Group 中存在 2 个 Consumer 订阅了 5 个 Partition，Consumer1 被分配了 Partition1、3、4，Consumer2 被分配了 Partition2、5。此时有一个新的 Consumer3 发送 JoinGroup 希望加入 Group。



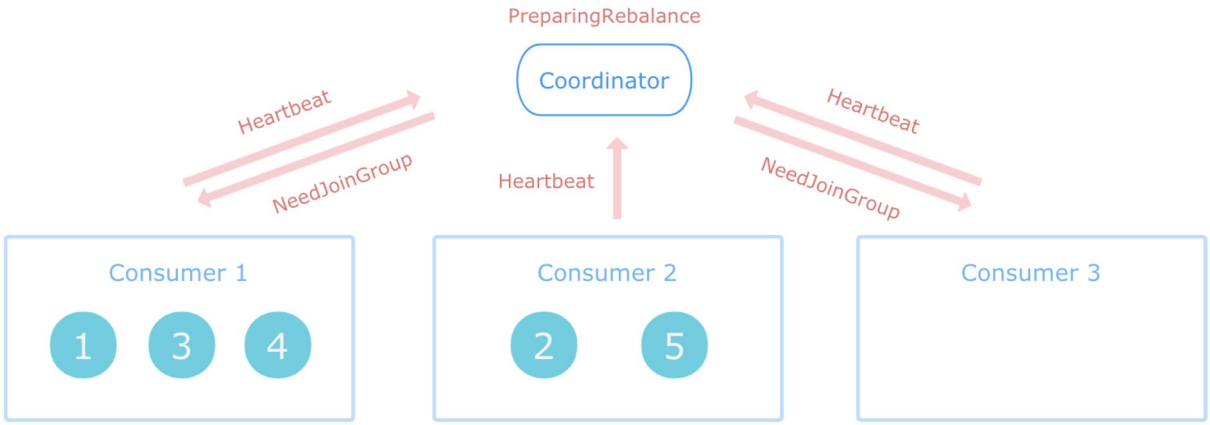
Step1：Coordinator 将该 Group 标记为 PreparingRebalance，此后接受到 Consumer 端发送的心跳后，将其当前分配的分区信息进行记录。



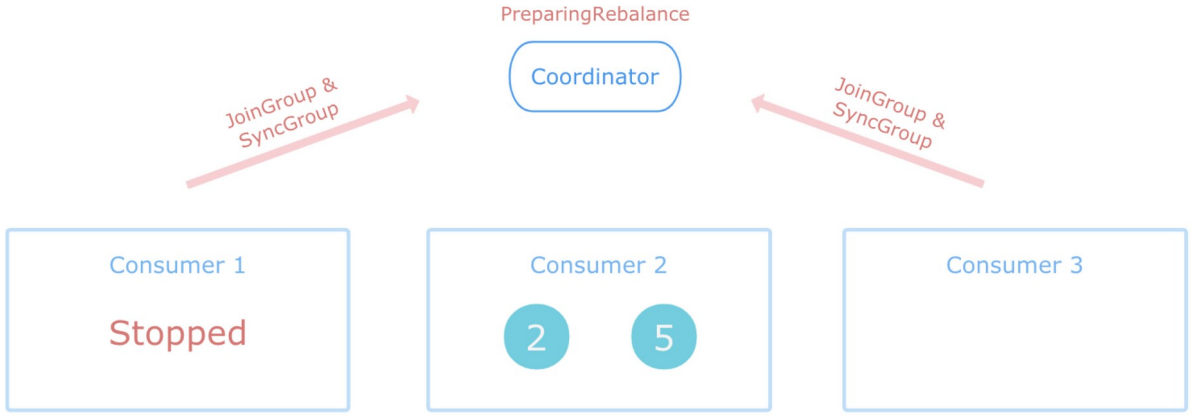
Step2：Coordinator 接受到 Group 内所有 Consumer 的心跳后，也即拥有了所有的原先分配分区方案，在 Coordinator 端使用 StickyAssignor 进行重新分配计算，简单来说就是进行最小程度的分区调整，保证 Group 中需要进行调整的 Consumer 数降至最低。将需要调整的 Consumer 标记为 needRebalance，准备对该 Consumer 进行单独通知。



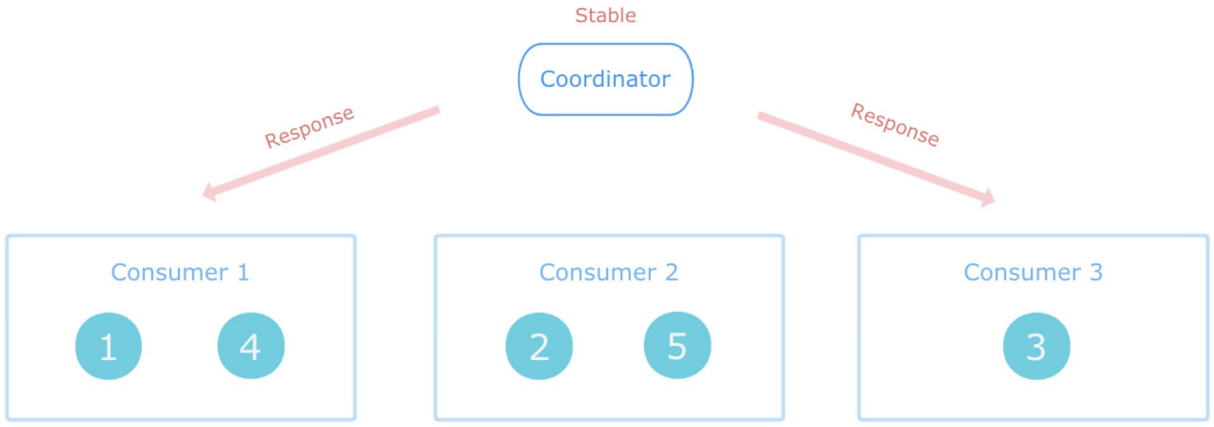
Step3：在下一次接收到 Consumer 的心跳时，若 Coordinator 端记录了该 Consumer 为 needRebalance，则通知其需要发送 JoinGroup 请求重新加入。



Step4：此时感知到自己为 needRebalance 状态的 Consumer 才需要停止消费，等待 Coordinator 下发重新分配后的分区方案。



Step5：Coordinator 返回分区方案后，Consumer 即可正常恢复消费。



可以看到，在我们新版 Server Rebalance 的模式下，触发一次 Rebalance 有了以下几点变化：

1. 直接在 Coordinator 端进行重分配方案，减少一次与客户端的 RPC 交互。

2. 单独通知需要修改分区方案的 Consumer 进行调整，对于需要调整的 Consumer，只需要在 Step4 时进行 stop-the-world 操作，其余时间都可以正常消费，耗时远小于原生 Rebalance 的等待时间。
3. 对于上述例子中的 Consumer2，其不需要进行订阅分区的调整，因此整个 Rebalance 流程对其透明，无需停止消费以及释放分区。

总结

Apache Kafka 在快手是一个被广泛使用的消息中间件，对于原生 Kafka 在超大规模 Consumer Group 上 Rebalance 速度慢、影响分区多、消息积压大等问题，我们提出了异步 Server Rebalance 的方案，在确保服务稳定性的前提下显著提升了 Rebalance 的性能。