Kafka在美团数据平台承担着统一的数据缓存和分发的角色，随着数据量的增长，集群规模的扩大，Kafka面临的挑战也愈发严峻。本文分享了美团Kafka面临的实际挑战，以及美团针对性的一些优化工作，希望能给从事相关开发工作的同学带来帮助或启发。



# 1. 现状和挑战

## 1.1 现状

Kafka是一个开源的流处理平台，我们首先了解一下Kafka在美团数据平台的现状。

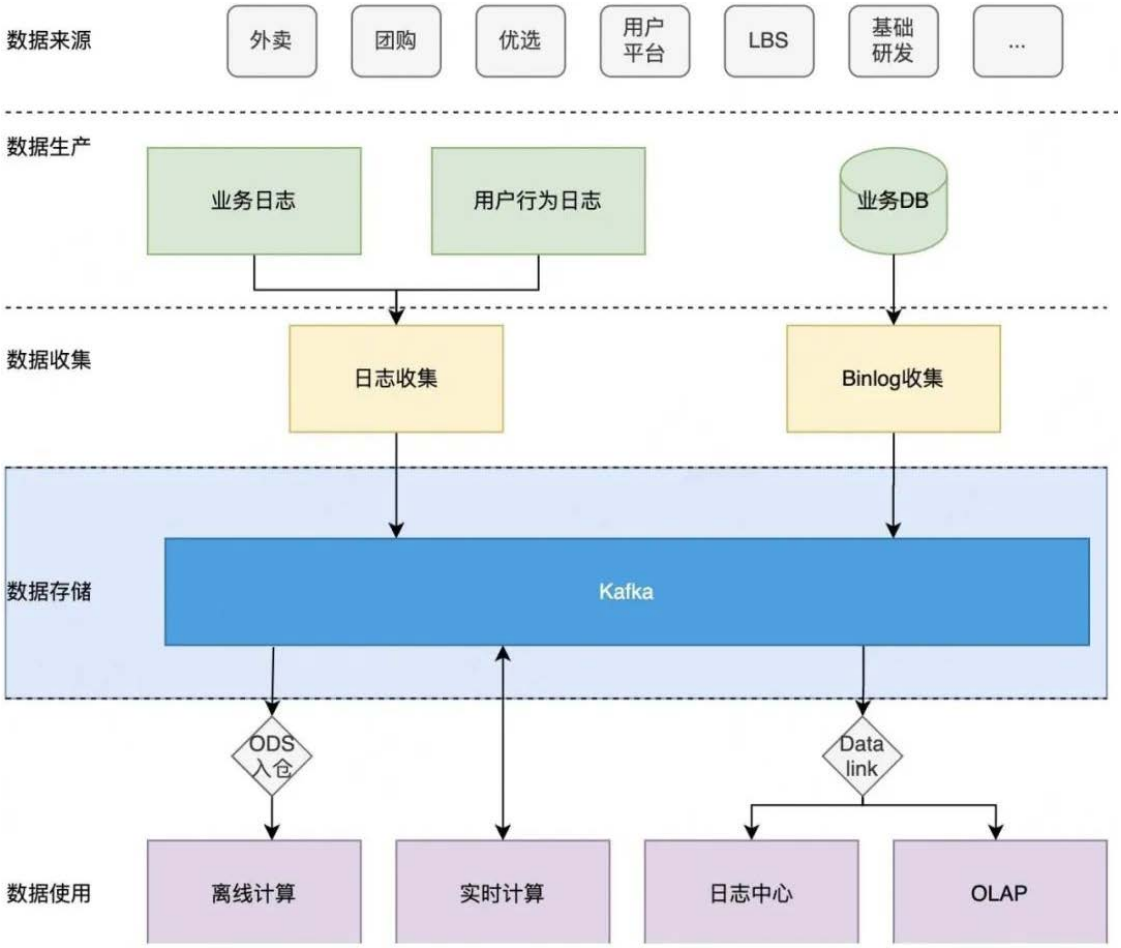


图1-1 Kafka在美团数据平台的现状

如图1-1所示，蓝色部分描述了Kafka在数据平台定位为流存储层。主要的职责是做数据的缓存和分发，它会将收集到的日志分发到不同的数据系统里，这些日志来源于系统日志、客户端日志以及业务数据库。下游的数据消费系统包括通过ODS入仓提供离线计算使用、直接供实时计算使用、通过DataLink同步到日志中心，以及做OLAP分析使用。

Kafka在美团的集群规模总体机器数已经超过了15000+台，单集群的最大机器数也已经到了2000+台。在数据规模上，天级消息量已经超过了30+P，天级消息量峰值也达到了4+亿/秒。不过随着集群规模的增大，数据量的增长，Kafka面临的挑战也愈发严峻，下面讲一下具体的挑战都有哪些。

## 1.2 挑战

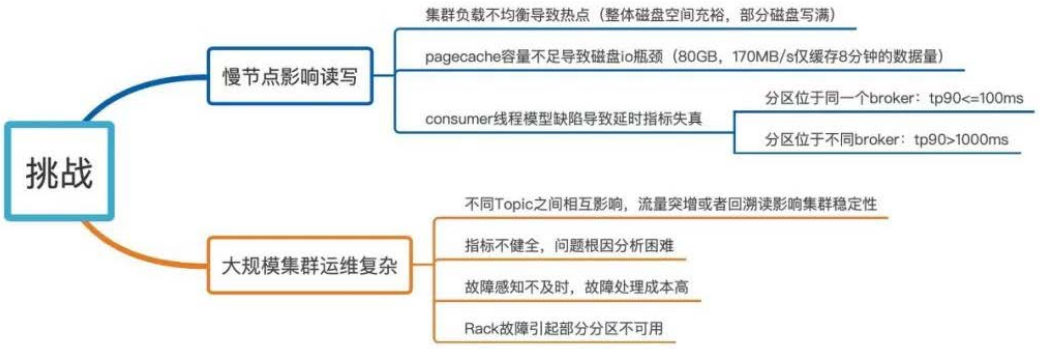


图1-2 Kafka在美团数据平台面临的挑战

如图1-2所示，具体的挑战可以概括为两部分：

**第一部分是慢节点影响读写**，这里慢节点参考了HDFS的一个概念，具体定义指的是读写延迟TP99大于300ms的Broker。造成慢节点的原因有三个：

1. 集群负载不均衡会导致局部热点，就是整个集群的磁盘空间很充裕或者ioutil很低，但部分磁盘即将写满或者ioutil打满。
2. PageCache容量，比如说，80GB的PageCache在170MB/s的写入量下仅能缓存8分钟的数据量。那么如果消费的数据是8分钟前的数据，就有可能触发慢速的磁盘访问。
3. Consumer客户端的线程模型缺陷会导致端到端延时指标失真。例如当Consumer消费的多个分区处于同一Broker时，TP90可能小于100ms，但是当多个分区处于不同Broker时，TP90可能会大于1000ms。

**第二部分是大规模集群管理的复杂性**，具体表现有4类问题：

1. 不同Topic之间会相互影响，个别Topic的流量突增，或者个别消费者的回溯读会影响整体集群的稳定性。
2. Kafka原生的Broker粒度指标不够健全，导致问题定位和根因分析困难。
3. 故障感知不及时，处理成本较高。
4. Rack级别的故障会造成部分分区不可用。

## 2. 读写延迟优化

接下来我们先介绍一下针对读写延迟问题，美团数据平台做了哪些优化。首先从宏观层面，我们将受影响因素分为应用层和系统层，然后详细介绍应用层和系统层存在的问题，并给出对应的解决方案，包括流水线加速、Fetcher隔离、迁移取消和Cgroup资源隔离等，下面具体介绍各种优化方案的实现。

### 2.1 概览

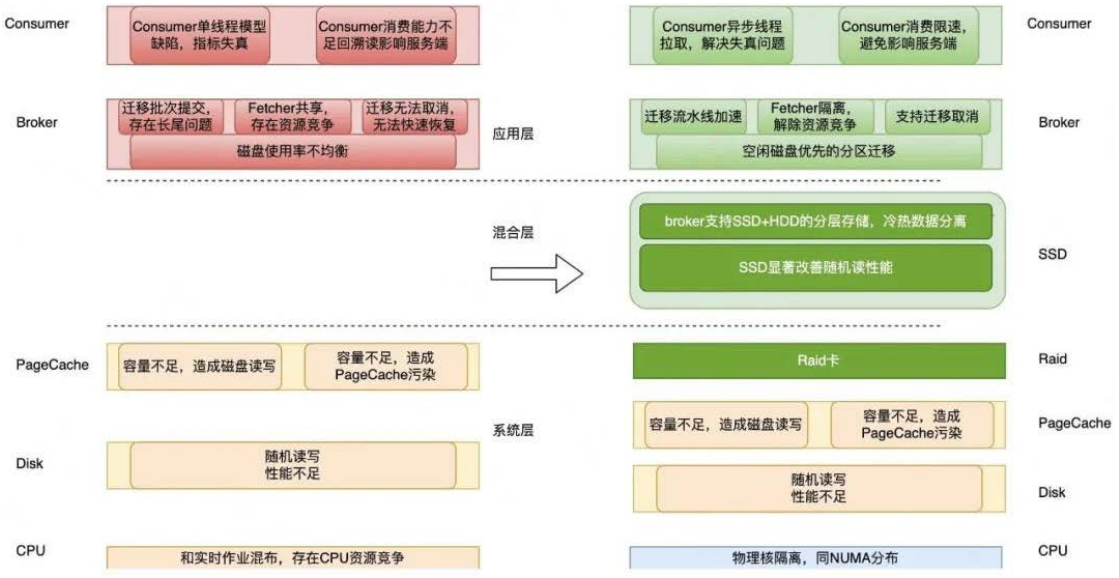


图2-1 Kafka读写延迟优化概览

图2-1是针对读写延迟碰到的问题以及对应优化方案的概览图。我们把受影响的因素分为应用层和系统层。

**应用层**主要包括3类问题：

1）Broker端负载不均衡，例如磁盘使用率不均衡、ioutil不均衡等问题。个别磁盘负载升高影响整个Broker的请求受到影响。

2）Broker的数据迁移存在效率问题和资源竞争问题。具体来讲，包括以下3个层面：

- 迁移只能按批次串行提交，每个批次可能存在少量分区迁移缓慢，无法提交下个批次，导致迁移效率受影响。
- 迁移一般在夜间执行，如果迁移拖到了午高峰还未完成，可能会显著影响读写请求。
- 迁移请求和实时拉取存在共用Fetcher线程的问题导致分区迁移请求可能会影响实时消费请求。

3）Consumer端单线程模型存在缺陷导致运维指标失真，并且单Consumer消费的分区数不受限制，消费能力不足就无法跟上实时最新的数据，当消费的分区数增多时可能会引起回溯读。

**系统层**也主要包括3类问题：

1）PageCache污染。Kafka利用内核层提供的ZeroCopy技术提升性能，但是内核层无法区分实时读写请求和回溯读请求，导致磁盘读可能污染PageCache，影响实时读写。

2）HDD在随机读写负载下性能差。HDD对于顺序读写友好，但是面对混合负载场景下的随机读写，性能显著下降。

3）CPU和内存等系统资源在混部场景下的资源竞争问题。在美团大数据平台，为了提高资源的利用率，IO密集型的服务（比如Kafka）会和CPU密集型的服务（比如实时计算作业）混布，混布存在资源竞争，影响读写延迟。

以上提到的问题，我们采取了针对性的策略。比如应用层的磁盘均衡、迁移流水线加速、支持迁移取消和Consumer异步化等。系统层的Raid卡加速、Cgroup隔离优化等。此外，针对HDD随机读写性能不足的问题，我们还设计并实现了基于SSD的缓存架构。

## 2.2 应用层

### ① 磁盘均衡

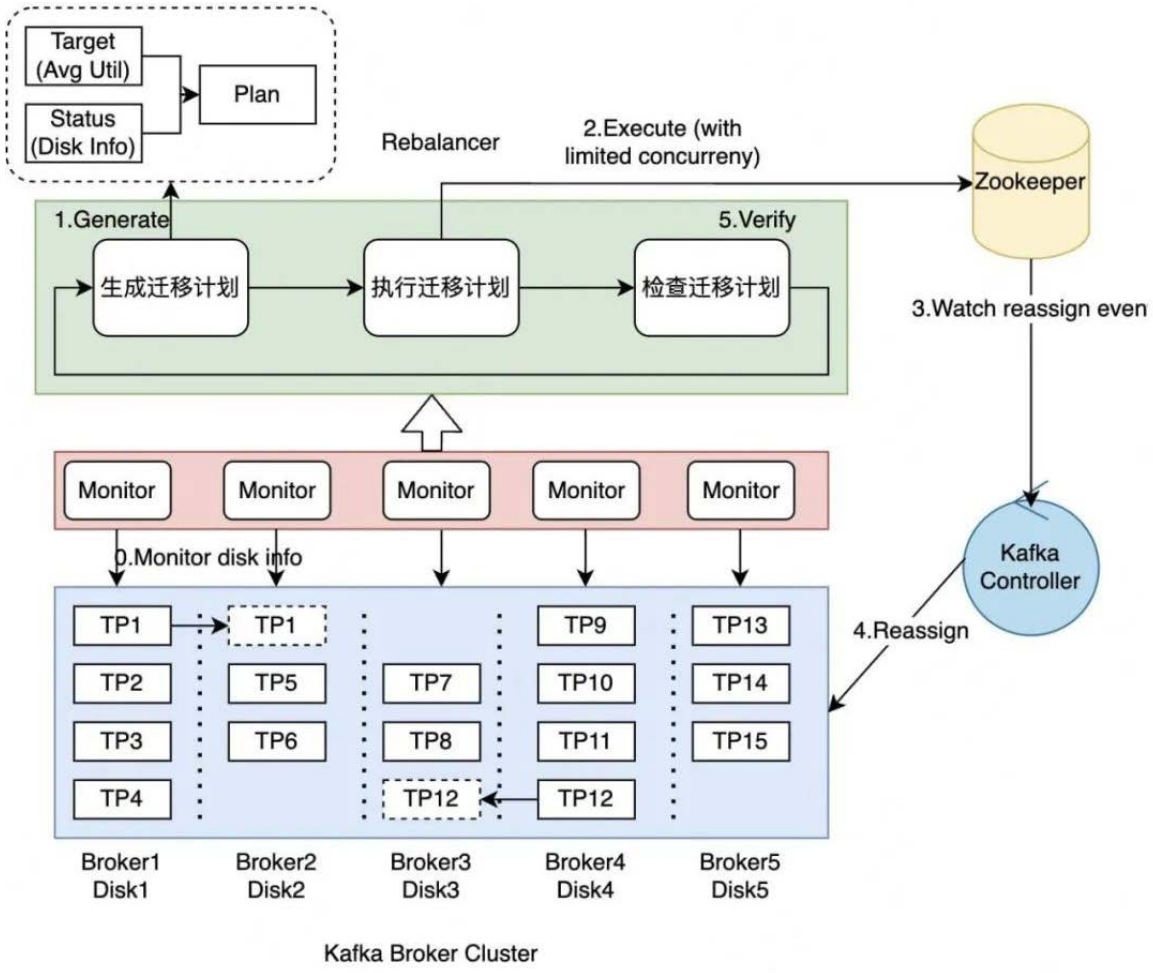


图2-2 Kafka应用层磁盘均衡

磁盘热点导致两个问题：

- 实时读写延迟变高，比如说TP99请求处理时间超过300ms可能会导致实时作业发生消费延迟问题，数据收集拥堵问题等。
- 集群整体利用率不足，虽然集群容量非常充裕，但是部分磁盘已经写满，这个时候甚至会导致某些分区停止服务。

针对这两个问题，我们采用了基于空闲磁盘优先的分区迁移计划，整个计划分为3步，由组件Rebalancer统筹管理：

1. 生成迁移计划。Rebalancer通过目标磁盘使用率和当前磁盘使用率（通过Kafka Monitor上报）持续生成具体的分区迁移计划。
2. 提交迁移计划。Rebalancer向Zookeeper的Reassign节点提交刚才生成的迁移计划，Kafka的Controller收到这个Reassign事件之后会向整个Kafka Broker集群提交Reassign事件。
3. 检查迁移计划。Kafka Broker负责具体执行数据迁移任务，Rebalancer负责检查任务进展。

如图2-2所示，每块Disk持有3个分区是一个相对均衡的状态，如果部分Disk持有4个分区，比如Broker1-Disk1和Broker4-Disk4；部分Disk持有2个分区，比如Broker2-Disk2，Broker3-Disk3，Reblanacer就会将Broker1-Disk1和Broker4-Disk4上多余的分区分别迁移到Broker2-Disk2和Broker3-Disk3，最终尽可能地保证整体磁盘利用率均衡。

**② 迁移优化**

虽然基于空闲磁盘优先的分区迁移实现了磁盘均衡，但是迁移本身仍然存在效率问题和资源竞争问题。接下来，我们会详细描述我们采取的针对性策略。

1. 采取流水线加速策略优化迁移缓慢引起的迁移效率问题。
2. 支持迁移取消解决长尾分区迁移缓慢引起的读写请求受影响问题。
3. 采取Fetcher隔离缓解数据迁移请求和实时读写请求共用Fetcher线程的问题。

**优化一，流水线加速**

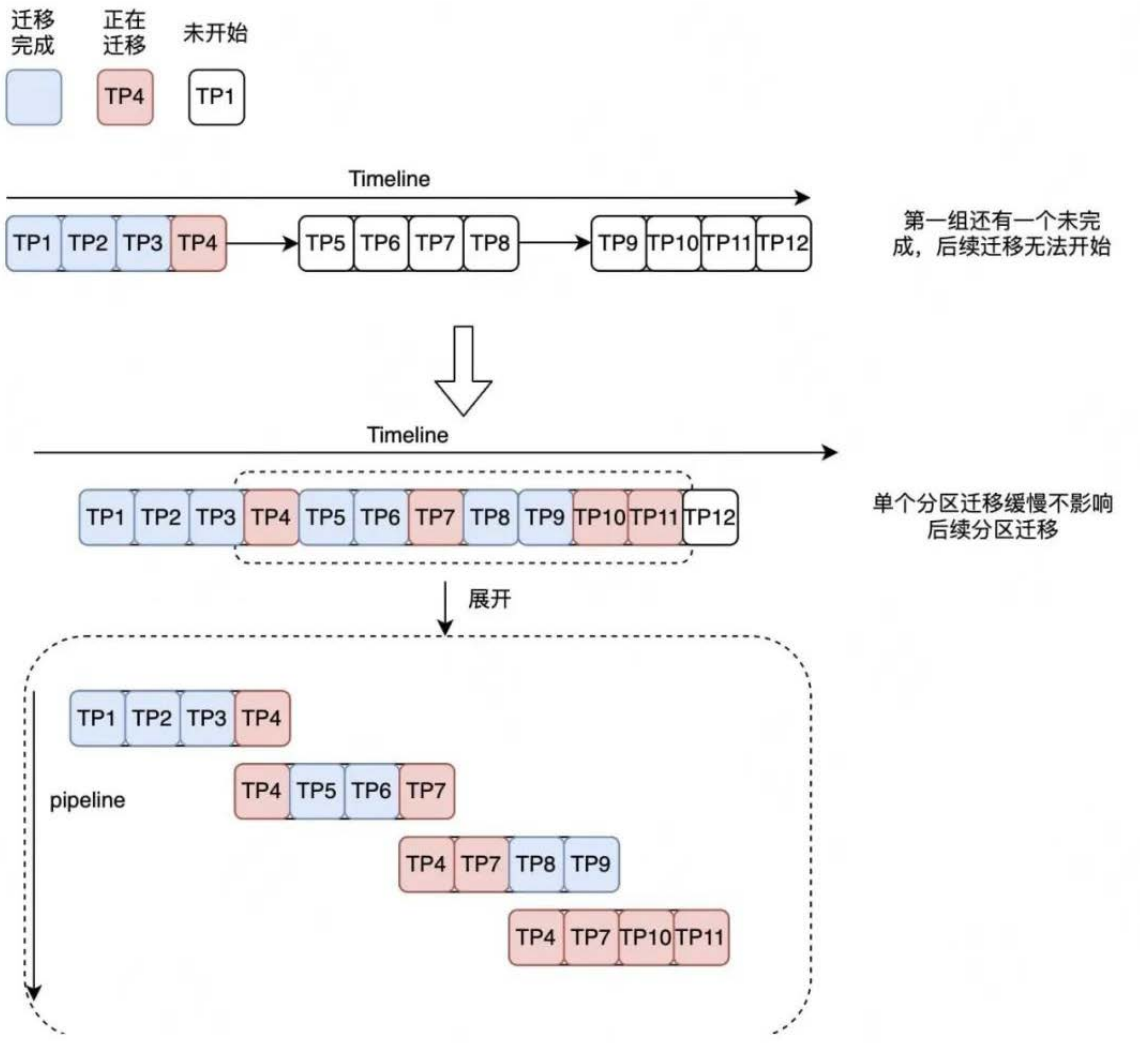


图2-3 流水线加速

如图2-3所示，箭头以上原生Kafka版本只支持按批提交，比如说一批提交了四个分区，当TP4这个分区一直卡着无法完成的时候，后续所有分区都无法继续进行。采用流水线加速之后，即使TP4这个分区还没有完成，可以继续提交新的分区。在相同的时间内，原有的方案受阻于TP4没有完成，后续所有分区都没办法完成，在新的方案中，TP4分区已经迁移到TP11分区了。图中虚线代表了一个无序的时间窗口，主要用于控制并发，目的是为了和原有的按组提交的个数保持一致，避免过多的迁移影响读写请求服务。

**优化二，迁移取消**

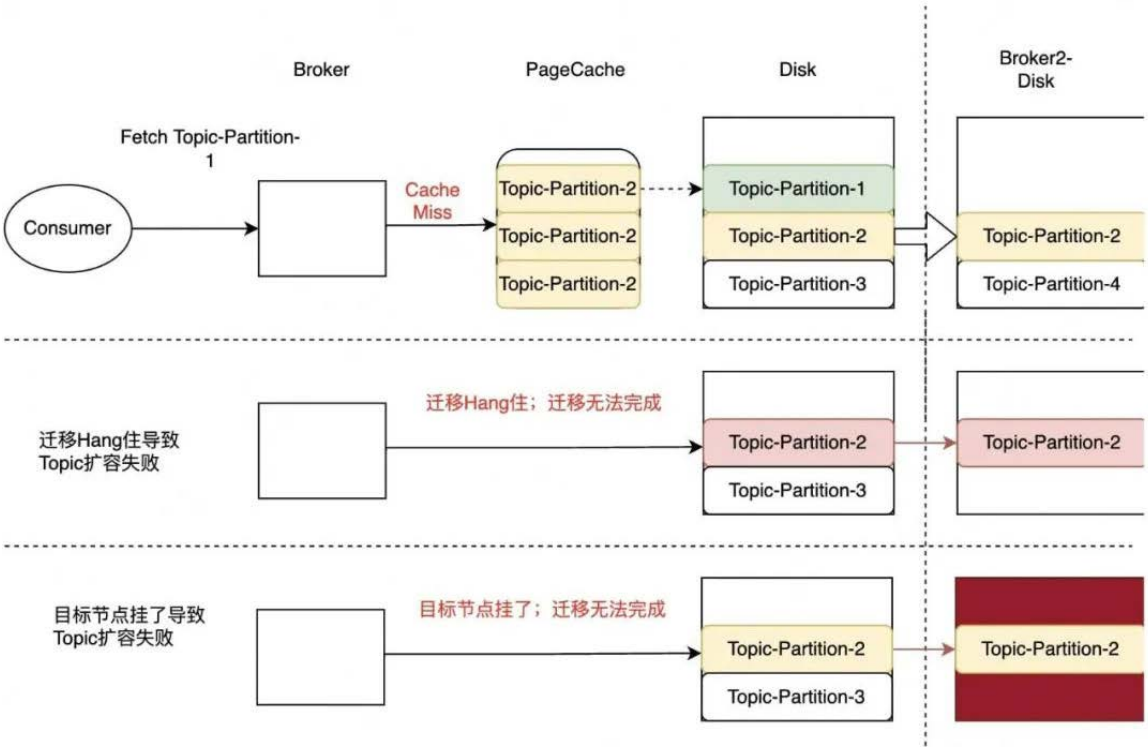


图2-4-1 迁移问题

如图2-4-1所示，箭头左侧描述了因为迁移影响的三种线上类型。第一种是因为迁移会触发最旧读，同步大量的数据，在这个过程中会首先将数据回刷到PageCache上引起PageCache污染，导致某个实时读的分区发生Cache Miss，触发磁盘度进而影响读写请求；第二种是当存在某些异常节点导致迁移Hang住时，部分运维操作无法执行，比如流量上涨触发的Topic自动扩分区。因为在Kafka迁移过程中这类运维操作被禁止执行。第三种和第二种类似，它的主要问题是当目标节点Crash，Topic扩分区也无法完成，用户可能一直忍受读写请求受影响。

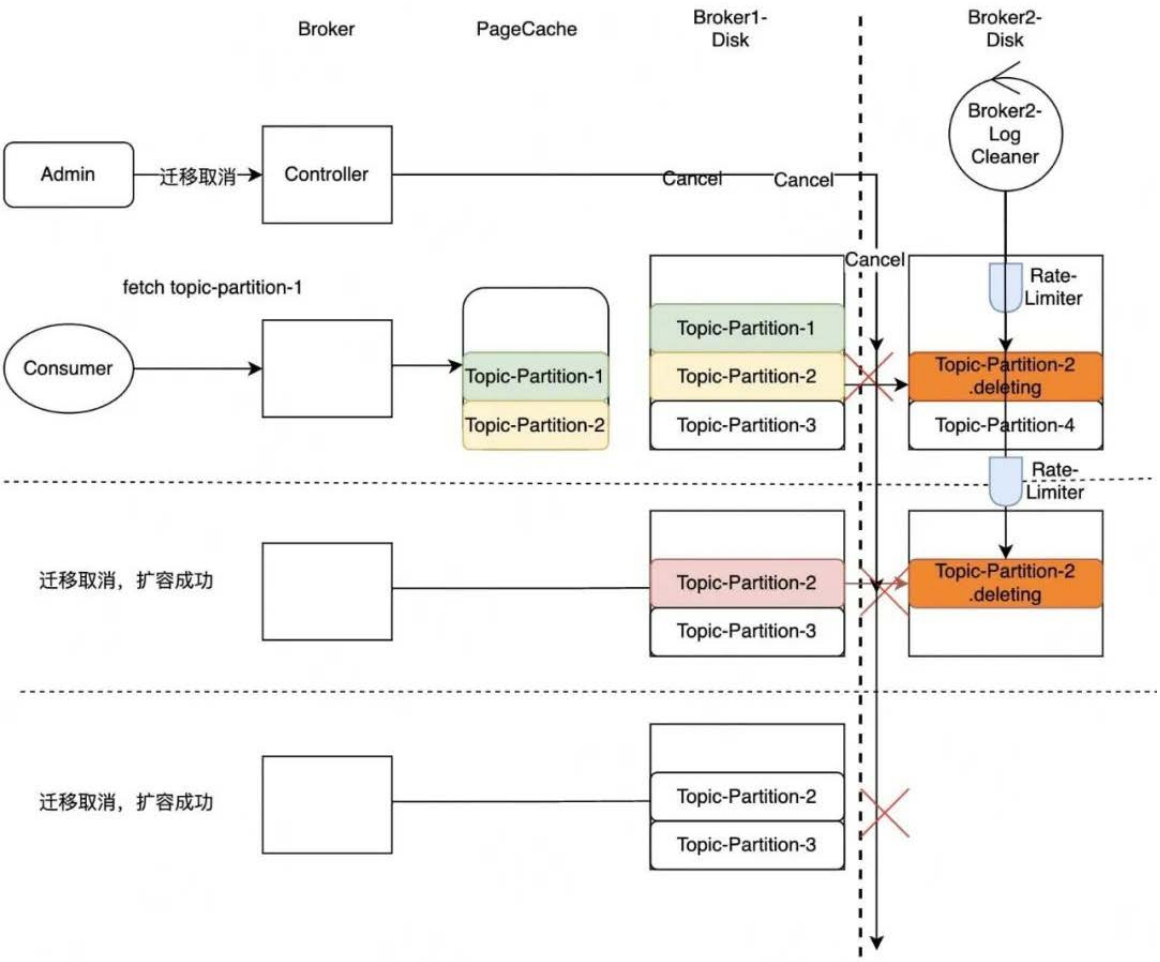


图2-4-2 迁移取消

针对上面提到的3种问题，我们支持了迁移取消功能。管理员可以调用迁移取消命令，中断正在迁移的分区，针对第一种场景，PageCache就不会被污染，实时读得以保证；在第二、三种场景中，因为迁移取消，扩分区得以完成。迁移取消会删除未完成迁移的分区，删除可能会导致磁盘IO出现瓶颈影响读写，因此我们通过支持平滑删除避免大量删除引起的性能问题。

**优化三，Fetcher隔离**

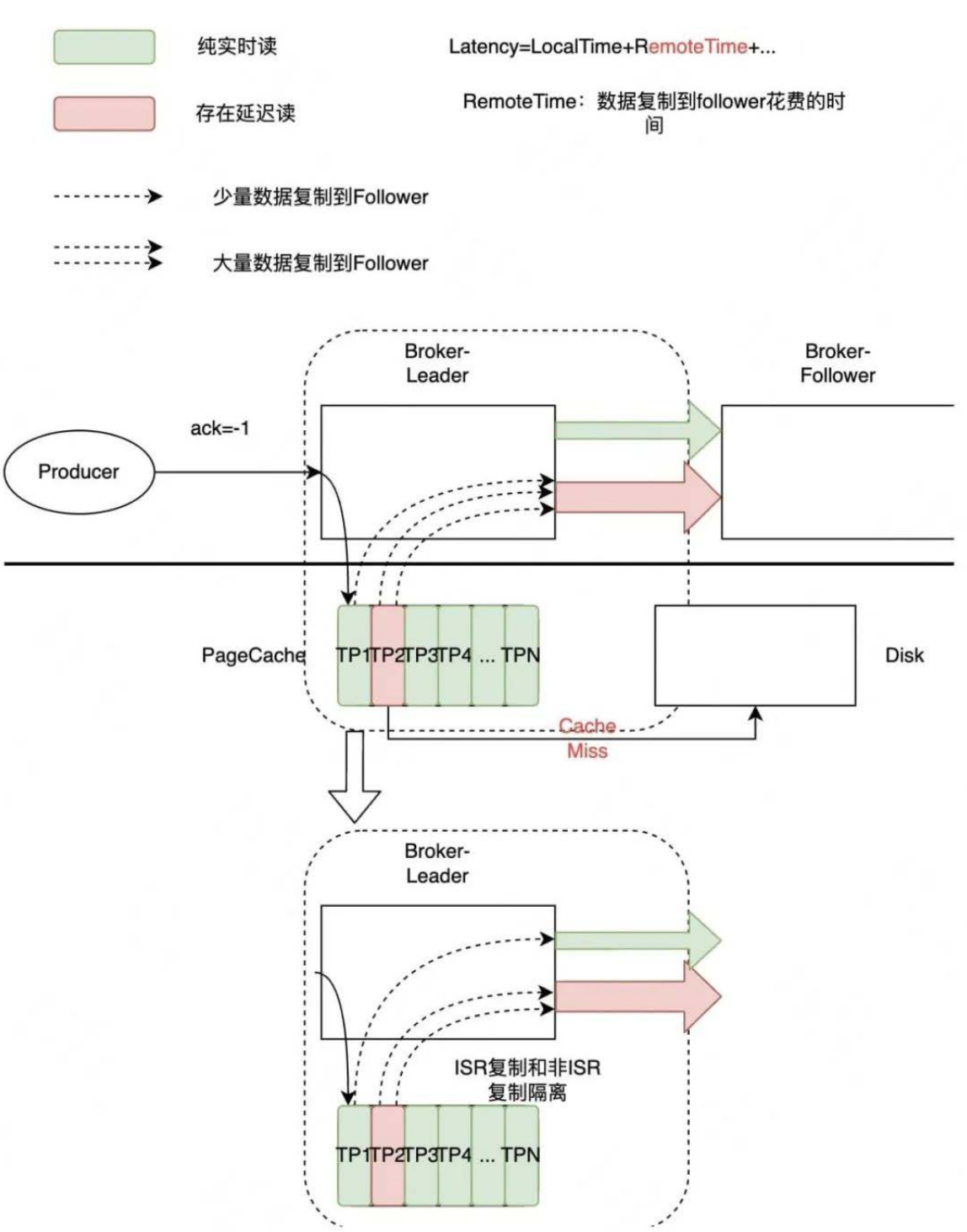


图2-5 Fetcher隔离

如图2-5，绿色代表实时读，红色代表延时读。当某一个Follower的实时读和延时读共享同一个Fetcher时，延时读会影响实时读。因为每一次延时读的数据量是显著大于实时读的，而且延时读容易触发磁盘读，可能数据已经不在PageCache中了，显著地拖慢了Fetcher的拉取效率。

针对这种问题，我们实施的策略叫Fetcher隔离。也就是说所有ISR的Follower共享Fetcher，所有非ISR的Follower共享Fetcher，这样就能保证所有ISR中的实时读不会被非ISR的回溯读所影响。

**③ Consumer异步化**

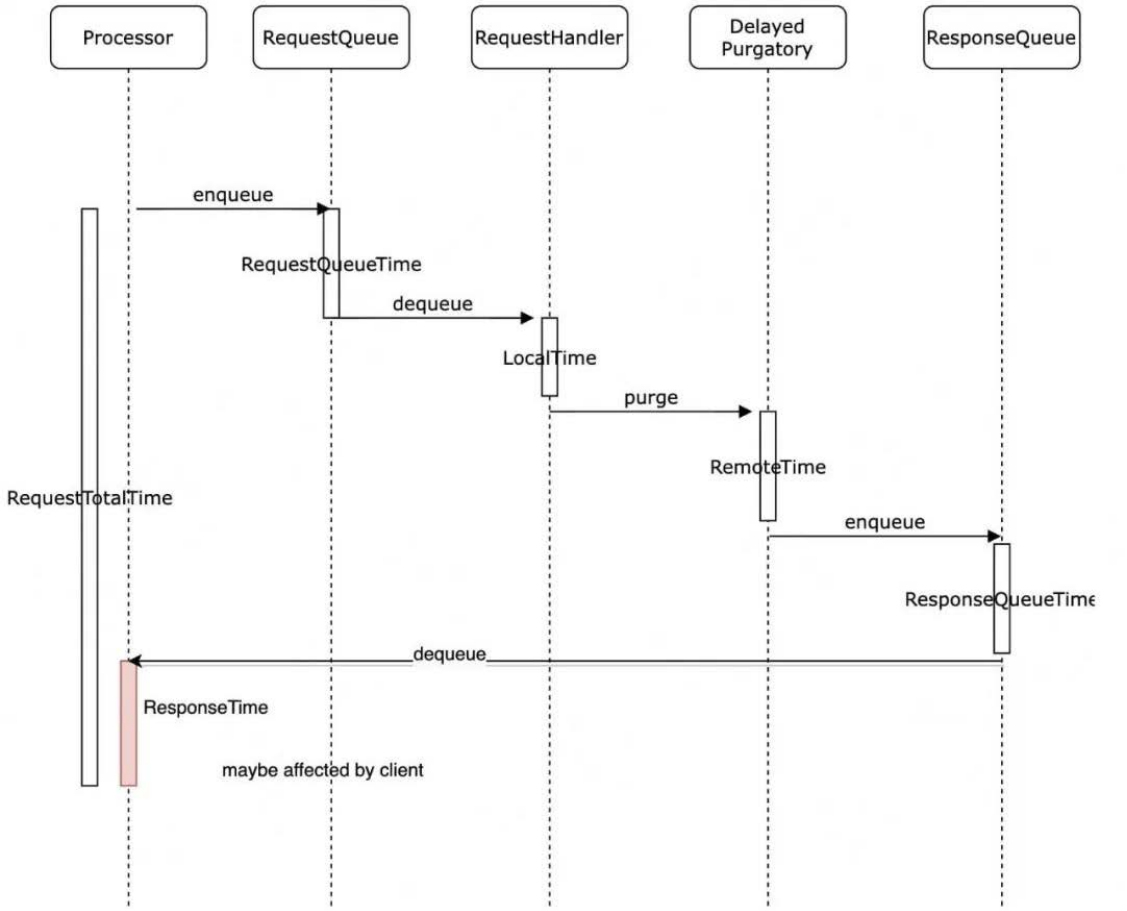


图2-6 Kafka-Broker分阶段延时统计模型

在讲述Consumer异步化前，需要解释下图2-6展示的Kafka-Broker分阶段延时统计模型。Kafka-Broker端是一个典型的事件驱动架构，各组件通过队列通信。请求在不同组件流转时，会依次记录时间戳，最终就可以统计出请求在不同阶段的执行耗时。

具体来说，当一个Kafka的Producer或Consumer请求进入到Kafka-Broker时，Processor组件将请求写入RequestQueue，RequestHandler从RequestQueue拉取请求进行处理，在RequestQueue中的等待时间是RequestQueueTime，RequestHandler具体的执行时间是LocalTime。当RequestHandler执行完毕后会将请求传递给DelayedPurgatory组件中，该组件是一个延时队列。

当触发某一个延时条件完成了以后会把请求写到ResponseQueue中，在DelayedPurgatory队列持续的时间为RemoteTime，Processor会不断的从ResponseQueue中将数据拉取出来发往客户端，标红的ResponseTime是可能会被客户端影响的，因为如果客户端接收能力不足，那么ResponseTime就会一直持续增加。从Kafka-Broker的视角，每一次请求总的耗时时RequestTotalTime，包含了刚才所有流程分阶段计时总和。

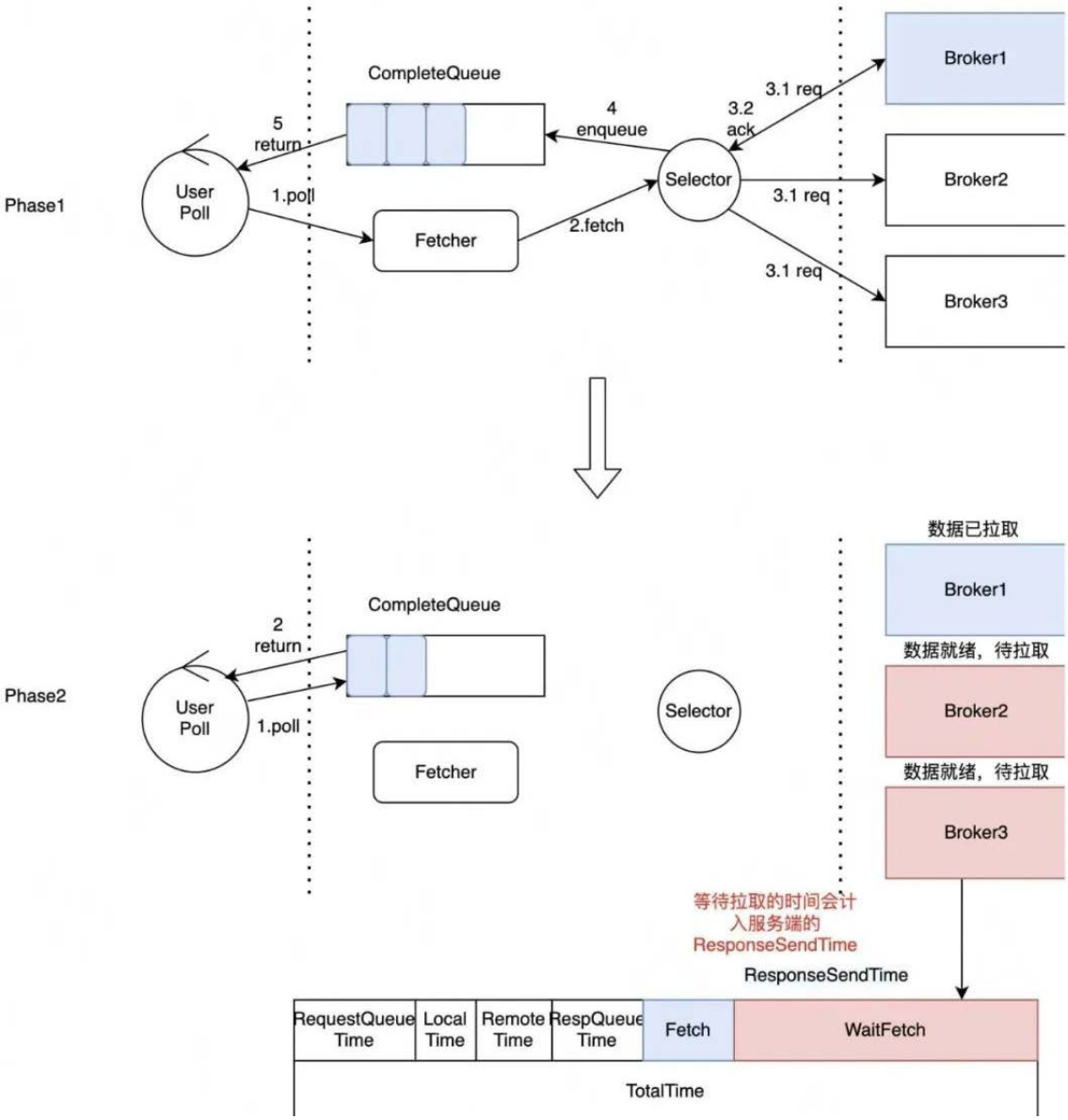


图2-7 Consumer异步化

ResponseTime持续增加的主要问题是因为Kafka原生Consumer基于NIO的单线程模型存在缺陷。如图2-7所示，在Phase1，User首先发起Poll请求，Kafka-Client会同时向Broker1、Broker2和Broker3发送请求，Broker1的数据先就绪时，Kafka Client将数据写入CompleteQueue，并立即返回，而不是继续拉取Broker2和Broker3的数据。后续的Poll请求会直接从CompleteQueue中读取数据，然后直接返回，直到CompleteQueue被清空。

在CompleteQueue被清空之前，即使Broker2和Broker3的端的数据已经就绪，也不会得到及时拉取。如图中Phase2，因为单线程模型存在缺陷导致WaitFetch这部分时长变大，导致Kafka-Broker的RespnseTime延时指标不断升高，带来的问题是无法对服务端的处理瓶颈进行精准的监控与细分。

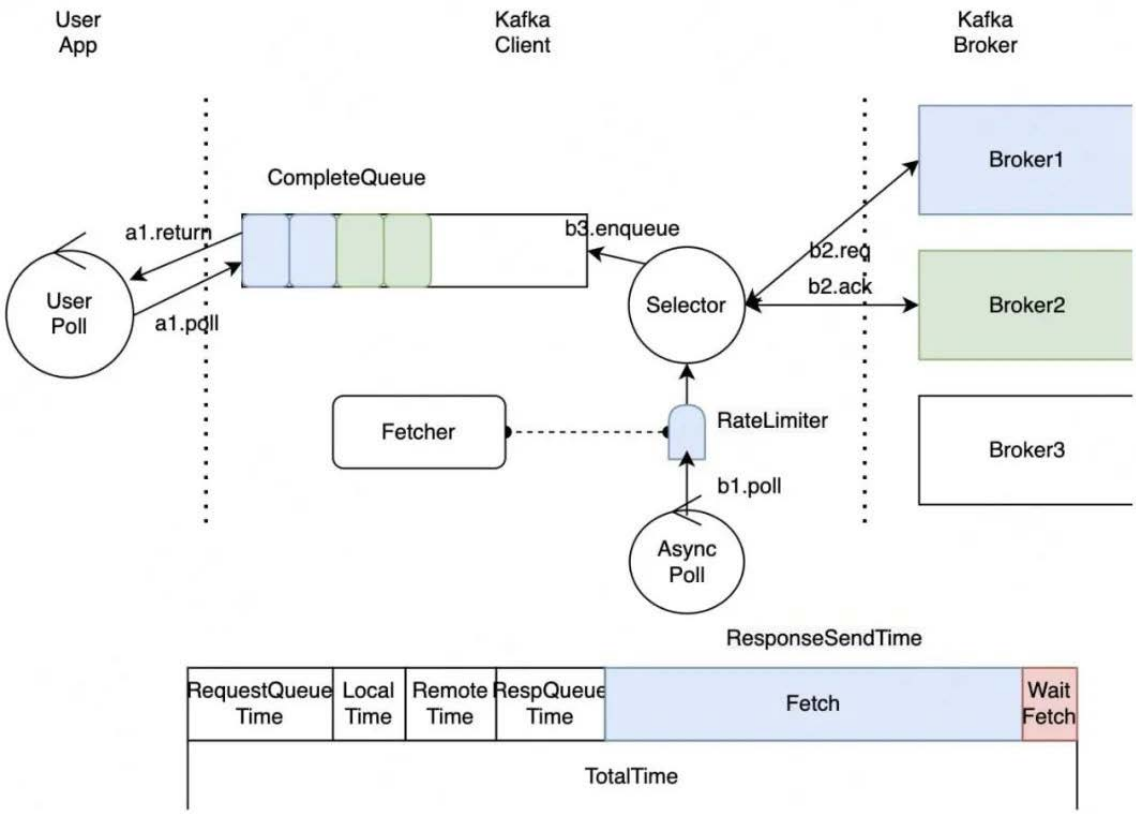


图2-8 引入异步拉取线程

针对这个问题，我们的改进是引入异步拉取线程。异步拉取线程会及时地拉取就绪的数据，避免服务端延时指标受影响，而且原生Kafka并没有限制同时拉取的分区数，我们在这里做了限速，避免GC和OOM的发生。异步线程在后台持续不断地拉取数据并放到CompleteQueue中。

## 2.3 系统层

### ① Raid卡加速

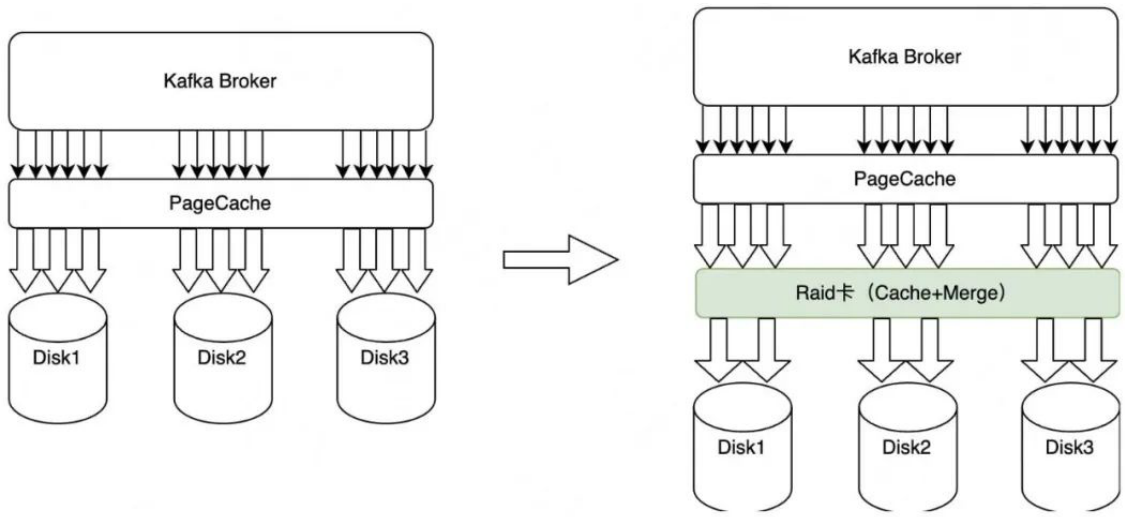


图2-9 Raid卡加速

HDD存在随机写性能不足的问题，表现为延时升高，吞吐降低。针对这个问题我们引入了Raid卡加速。Raid卡自带缓存，与PageCache类似，在Raid这一层会把数据Merge成更大的Block写入Disk，更加充分利用顺序写HDD的

带宽，借助Raid卡保证了随机写性能。

### ② Cgroup隔离优化

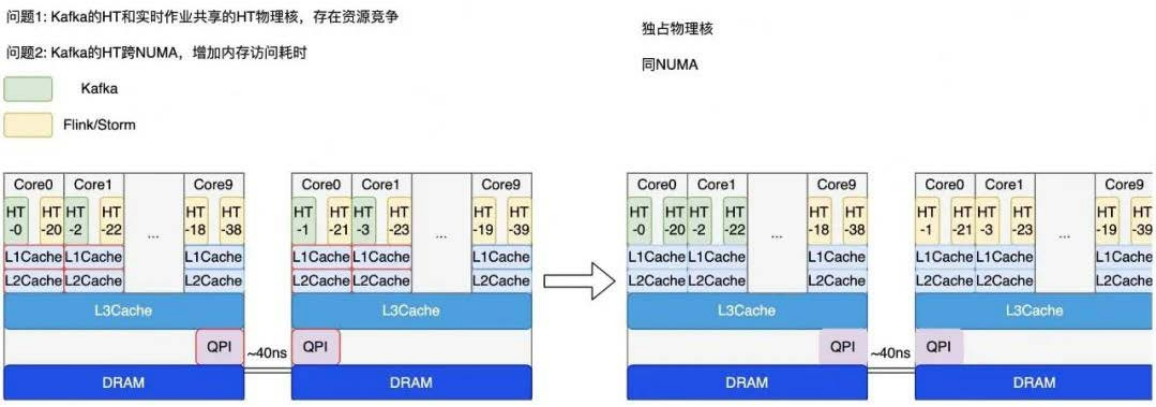


图2-10 Cgroup隔离

为了提高资源利用率，美团数据平台将IO密集型应用和CPU密集型应用混合部署。IO密集型应用在这里指的就是Kafka，CPU密集型应用在这里指的是Flink和Storm。但是原有的隔离策略存在两个问题：首先是物理核本身会存在资源竞争，在同一个物理核下，共享的L1Cache和L2Cache都存在竞争，当实时平台CPU飙升时会导致Kafka读写延时受到影响；其次，Kafka的HT跨NUMA，增加内存访问耗时，如图2-10所示，跨NUMA节点是通过QPI去做远程访问，而这个远程访问的耗时是40ns。

针对这两个问题，我们改进了隔离策略，针对物理核的资源竞争，我们新的混布策略保证Kafka独占物理核，也就是说在新的隔离策略中，不存在同一个物理核被Kafka和Flink同时使用；然后是保证Kafka的所有超线程处于同一侧的NUMA，避免Kafka跨NUMA带来的访问延时。通过新的隔离策略，Kafka的读写延时不再受Flink　CPU飙升的影响。

## 2.4 混合层-SSD新缓存架构

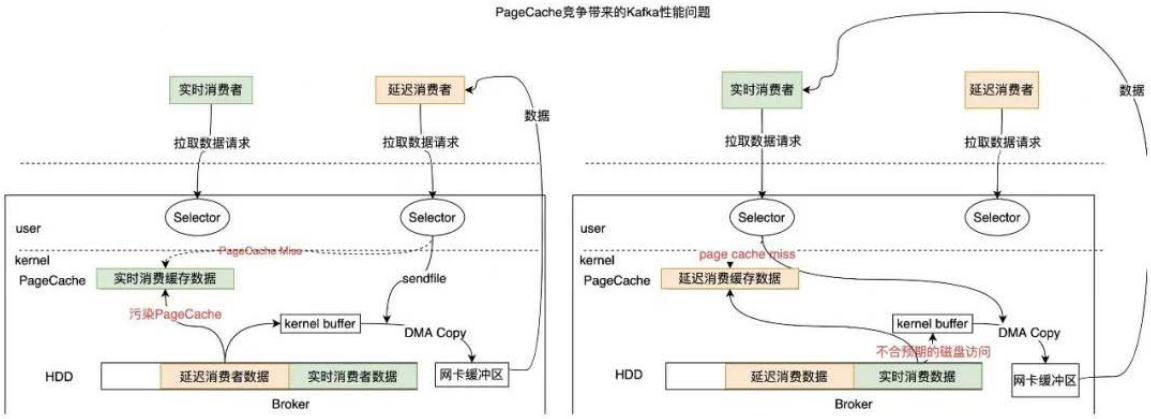


图2-11 Page污染引起的性能问题

### 背景和挑战

Kafka利用操作系统提供的ZeroCopy技术处理数据读取请求，PageCache容量充裕时数据直接从PageCache拷贝到网卡，有效降低了读取延时。但是实际上，PageCache的容量往往是不足的，因为它不会超过一个机器的内

存。容量不足时，ZeroCopy就会触发磁盘读，磁盘读不仅显著变慢，还会污染PageCache影响其他读写。

如图2-11中左半部分所示，当一个延迟消费者去拉取数据时，发现PageCache中没有它想要的数据，这个时候就会触发磁盘读。磁盘读后会将数据回写到PageCache，导致PageCache污染，延迟消费者消费延迟变慢的同时也会导致另一个实时消费受影响。因为对于实时消费而言，它一直读的是最新的数据，最新的数据按正常来说时不应该触发磁盘读的。

**选型和决策**

针对这个问题，我们这边在做方案选型时提供了两种方案：

**方案一**，读磁盘时不回写PageCache，比如使用DirectIO，不过Java并不支持；

**方案二**，在内存和HDD之间引入中间层，比如SSD。众所周知，SSD和HDD相比具备良好的随机读写能力，非常适合我们的使用场景。针对SSD的方案我们也有两种选型：

| 决策 | 优势 | 不足 |
| --- | --- | --- |
| 基于操作系统内核层实现 | 1.数据路由对应用层透明，对应用代码改动量小。<br>2.开源软件自身的健壮性由社区维护，可用性较好（前提：社区比较活跃）。 | 1.FlashCache/OpenCAS每种模式下都会将数据回刷到SSD缓存中，与PageCache相似，都会发生缓存污染。<br>2.发生Cache Miss时会多一次对设备的访问，延迟增加<br>3.所有的Meta数据都由操作系统维护，内核消耗的内存会增加，在与其他引擎混布的场景下会导致其他服务可申请的内存减少。 |
| Kafka应用内部实现 | 1.设计缓存策略时充分考虑了Kafka的读写特性，确保近实时的数据消费请求全部落在SSD上，保证这部分请求处理的低延迟，同时从HDD读取的数据不会回刷到SSD防止缓存污染。<br>2.由于每个日志段都有唯一明确的状态，因此每次请求的查询路径最短，不存在因Cache Miss带来的额外性能开销。 | 1.需要在Server端代码上进行改进，涉及的开发及测试工作量较大。<br>2.随社区大版本升级，也需要迭代上这些改进的代码。但可将相关代码贡献社区，解决迭代问题。 |

**方案一**，可以基于操作系统的内核实现，这种方案SSD与HDD存储空间按照固定大小分块，并且SSD与HDD建立映射关系，同时会基于数据局部性原理，Cache Miss后数据会按LRU和LFU替换SSD中部分数据，业界典型方案包括OpenCAS和FlashCache。其优势是数据路由对应用层透明，对应用代码改动量小，并且社区活跃可用性好；但是问题在于局部性原理并不满足Kafka的读写特性，而且缓存空间污染问题并未得到根本解决，因为它会根据LRU和LFU去替换SSD中的部分数据。

**方案二**，基于Kafka的应用层去实现，具体就是Kafka的数据按照时间维度存储在不同设备上，对于近实时数据直接放在SSD上，针对较为久远的数据直接放在HDD上，然后Leader直接根据Offset从对应设备读取数据。这种方案的优势是它的缓存策略充分考虑了Kafka的读写特性，确保近实时的数据消费请求全部落在SSD上，保证这部分请求处理的低延迟，同时从HDD读取的数据不回刷到SSD防止缓存污染，同时由于每个日志段都有唯一明确的状态，因此每次请求目的明确，不存在因Cache Miss带来的额外性能开销。同时劣势也很明显，需要在Server端代码上进行改进，涉及的开发以及测试的工作量较大。

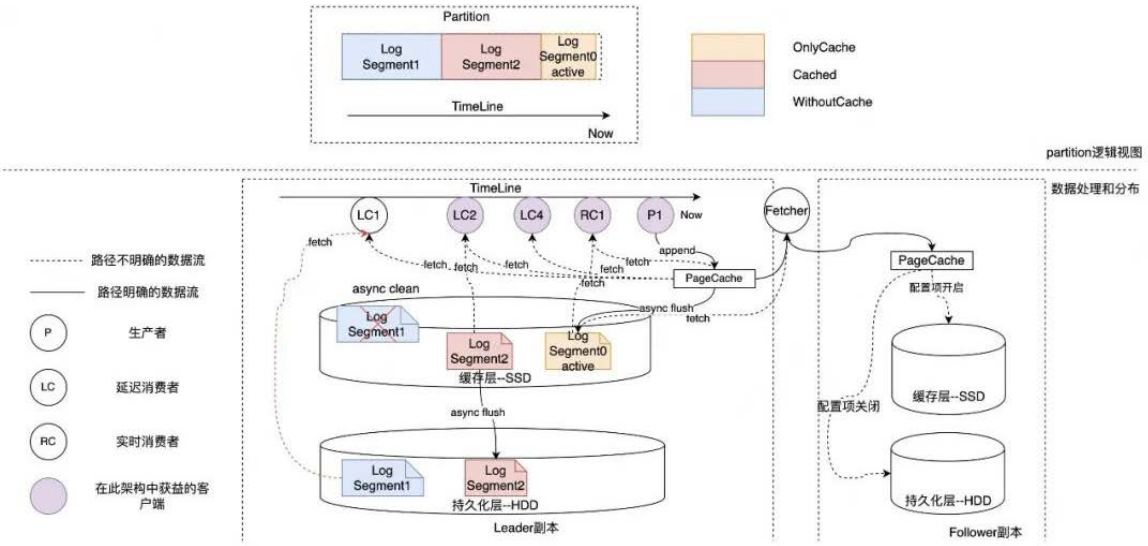


图2-13 KafkaSSD新缓存架构

## 具体实现

下面来介绍一下SSD新缓存架构的具体实现。

1. 首先新的缓存架构会将Log内的多个Segment按时间维度存储在不同的存储设备上，如图2-14中的红圈1，新缓存架构数据会有三种典型状态，一种叫Only Cache，指的是数据刚写进SSD，还未同步到HDD上；第2个是Cached，指数据既同步到了HDD也有一部分缓存在SSD上；第三种类型叫WithoutCache，指的是同步到了HDD但是SSD中已经没有缓存了。
2. 然后后台异步线程持续地将SSD数据同步到HDD上。
3. 随着SSD的持续写入，当存储空间达到阈值后，会按时间顺序删除距当前时间最久的数据，因为SSD的数据空间有限。
4. 副本可根据可用性要求灵活开启是否写入SSD。
5. 从HDD读取的数据是不会回刷到SSD上的，防止缓存污染。

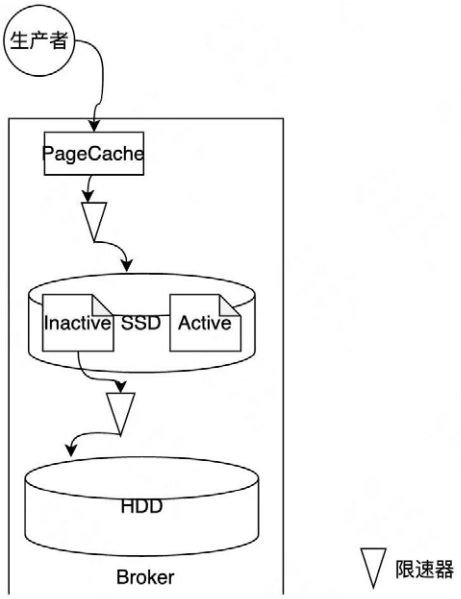


图2-14 SSD新缓存架构细节优化

**细节优化**

介绍了具体实现之后，再来看一下细节优化。

1. 首先是关于日志段同步，就是刚才说到的Segment，只同步Inactive的日志段，Inactive指的是现在并没有在写的日志段，低成本解决数据一致性问题。
2. 其次是做同步限速优化，在SSD向HDD同步时是需要限速的，同时保护了两种设备，不会影响其他IO请求的处理。

# 3. 大规模集群管理优化

## 3.1 隔离策略

美团大数据平台的Kafka服务于多个业务，这些业务的Topic混布在一起的话，很有可能造成不同业务的不同Topic之间相互影响。此外，如果Controller节点同时承担数据读写请求，当负载明显变高时，Controller可能无法及时控制类请求，例如元数据变更请求，最终可能会造成整个集群发生故障。

针对这些相互影响的问题，我们从业务、角色和优先级三个维度来做隔离优化。

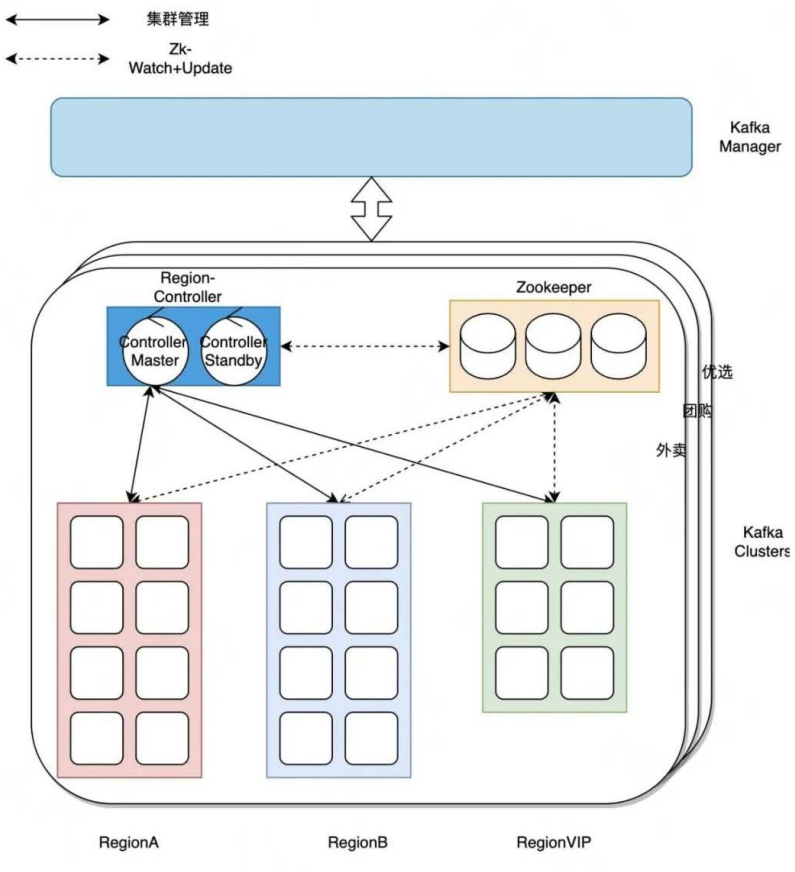


图3-1 隔离优化

- 第一点是业务隔离，如图3-1所示，每一个大的业务会有一个独立的Kafka集群，比如外卖、到店、优选。
- 第二点是分角色隔离，这里Kafka的Broker和Controller以及它们依赖的组件Zookeeper是部署在不同机器上的，避免之间相互影响。
- 第三点是分优先级，有的业务Topic可用性等级特别高，那么我们就可以给它划分到VIP集群，给它更多的资源冗余去保证其可用性。

## 3.2 全链路监控

随着集群规模增长，集群管理碰到了一系列问题，主要包括两方面：

- Broker端延时指标无法及时反应用户问题。
  - 随着请求量的增长，Kafka当前提供的Broker端粒度的TP99甚至TP999延时指标都可能无法反应长尾延时。
  - Broker端的延时指标不是端到端指标，可能无法反应用户的真实问题。
- 故障感知和处理不及时。

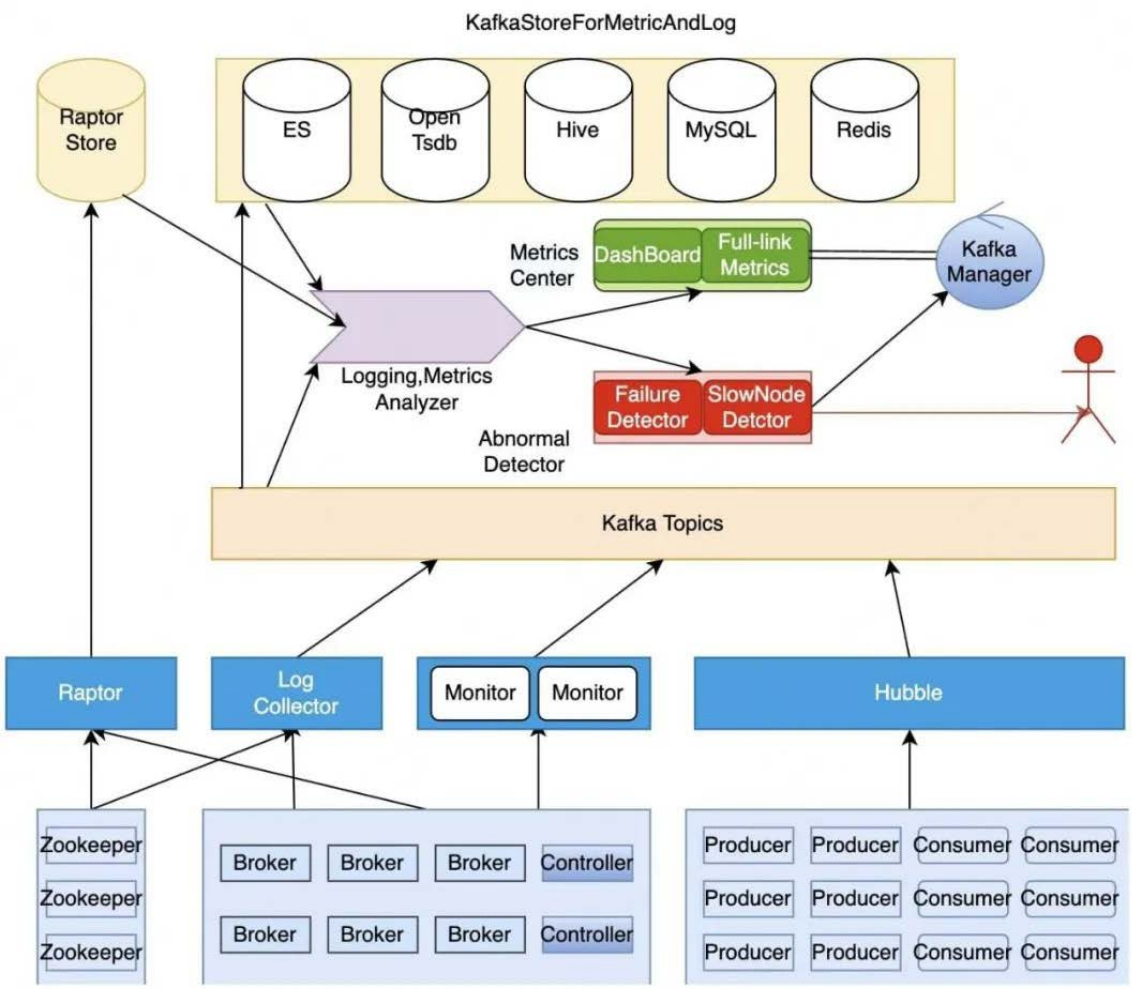


图3-2 全链路监控

针对这两个问题，我们采取的策略是全链路监控。全链路监控收集和监控Kafka核心组件的指标和日志。全链路监控架构如图3-2所示。当某一个客户端读写请求变慢时，我们通过全链路监控可以快速定位到具体慢在哪个环节，全链路指标监控如图3-3所示。

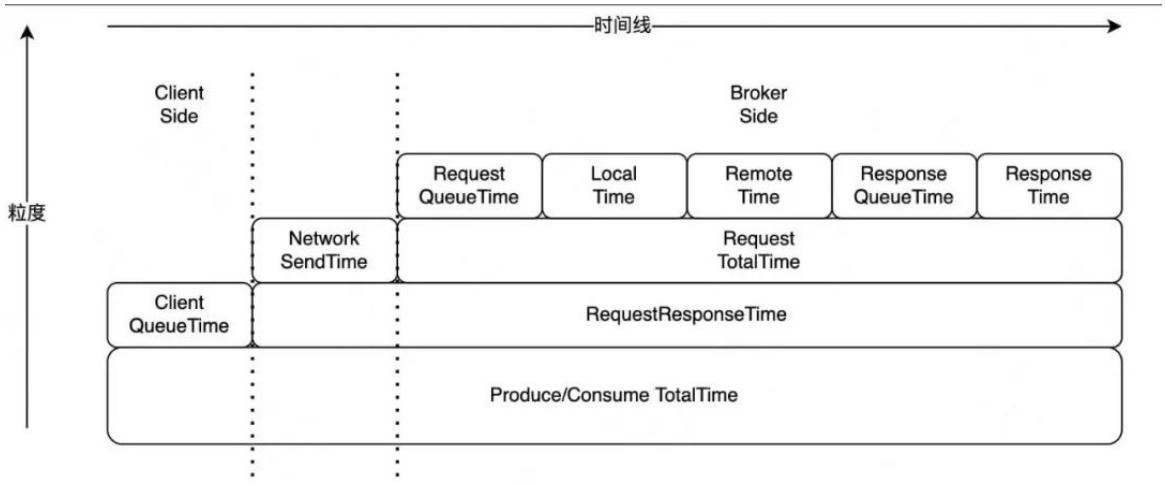


图3-3 全链路指标监控

图3-4是一个根据全链路指标定位请求瓶颈的示例，可以看出服务端RemoteTime占比最高，这说明耗时主要花费在数据复制。日志和指标的解析服务可以自动实时感知故障和慢节点，大部分的故障（内存、磁盘、Raid卡以及网卡等）和慢节点都已经支持自动化处理，还有一类故障是计划外的故障，比如分区多个副本挂掉导致的不可用，迁

移Hang住以及非预期的错误日志等，需要人工介入处理。

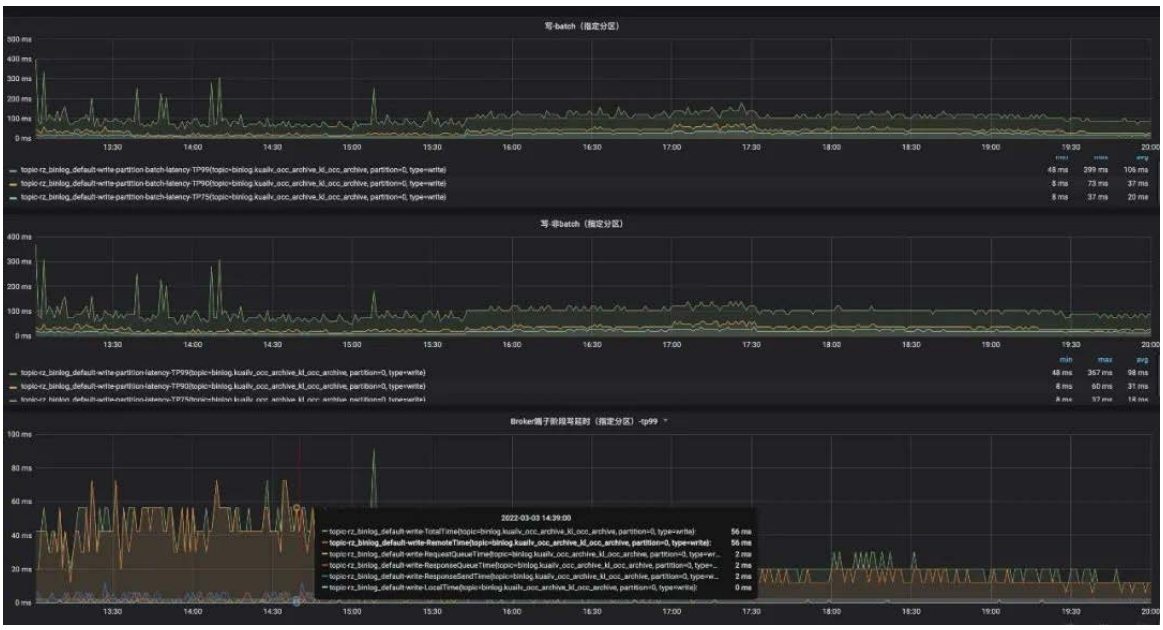

图3-4 全链路监控指标示例

## 3.3 服务生命周期管理

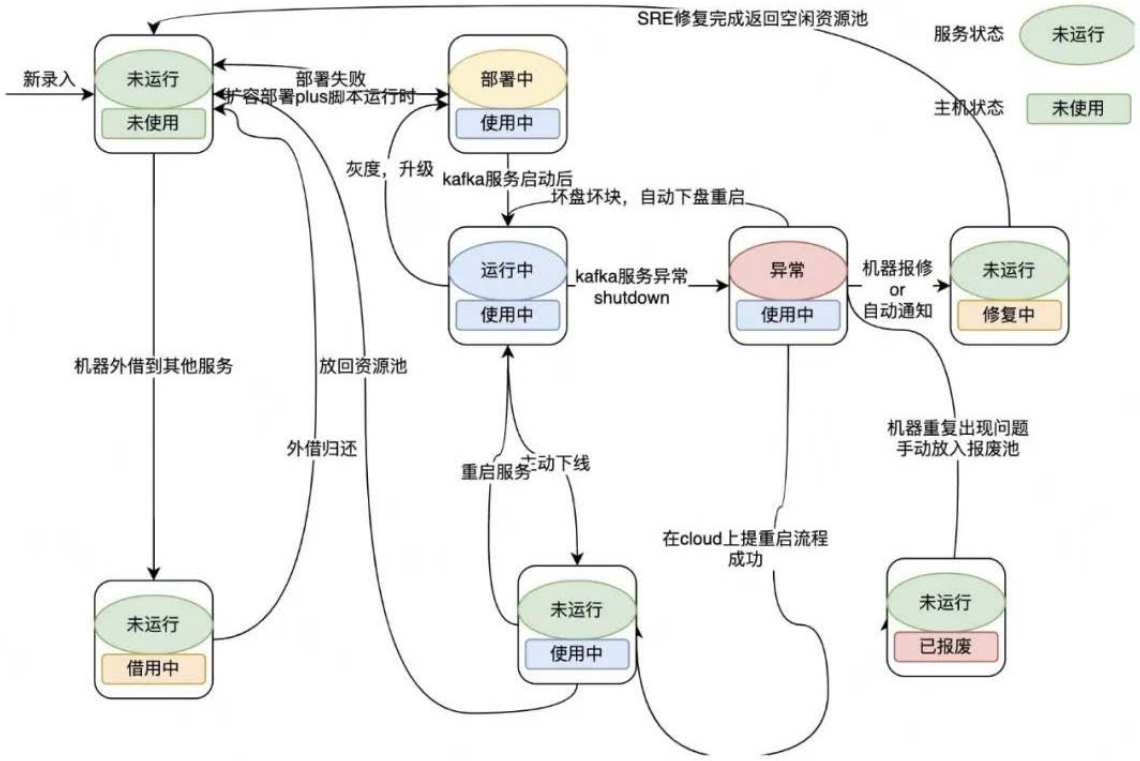


图3-5 服务生命周期管理

美团线上Kafka的服务器规模在万级别，随着服务规模的增长，我们对服务和机器本身的管理，也在不断迭代。我们的自动化运维系统能够处理大部分的机器故障和服务慢节点，但对于机器和服务本身的管理是割裂的，导致存在两类问题：

1. 状态语义存在歧义，无法真实反映系统状态，往往需要借助日志和指标去找到真实系统是否健康或者异常。
2. 状态不全面，异常Case需人工介入处理，误操作风险极大。

为了解决这两类问题，我们引入了生命周期管理机制，确保能够真实反映系统状态。生命周期管理指的是从服务开始运行到机器报废停止服务的全流程管理，并且做到了服务状态和机器状态联动，无需人工同步变更。而且新的生命周期管理机制的状态变更由特定的自动化运维触发，禁止人工变更。

## 3.4 TOR容灾

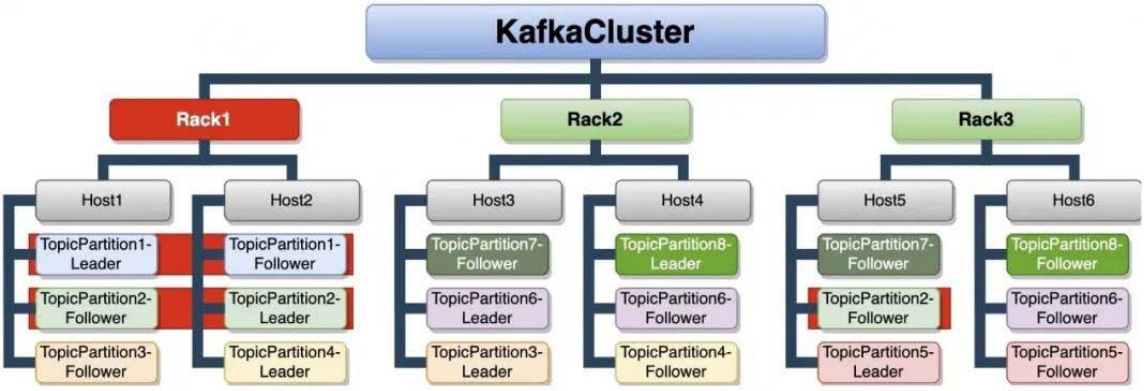


图3-6 TOR容灾挑战

我们从工程实现的角度，归纳总结了当前主流图神经网络模型的基本范式，实现一套通用框架，以期涵盖多种GNN模型。以下按照图的类型（同质图、异质图和动态图）分别讨论。

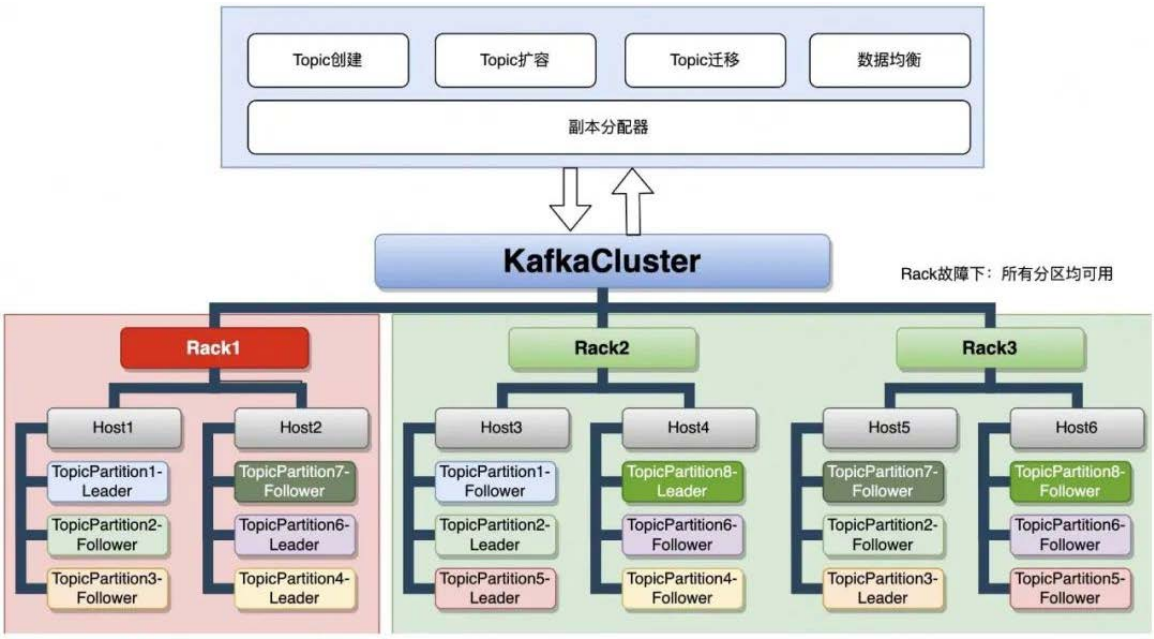


图3-7 TOR容灾

TOR容灾保证同一个分区的不同副本不在同一个Rack下，如图3-7所示，即使Rack1整个发生故障，也能保证所有分区可用。

# 4 未来展望

过去一段时间，我们围绕降低服务端的读写延迟做了大量的优化，但是在服务高可用方面，依然有一些工作需要完成。未来一段时间，我们会将重心放在提升鲁棒性和通过各种粒度的隔离机制缩小故障域。比如，让客户端主动对

一些故障节点进行避让，在服务端通过多队列的方式隔离异常请求，支持服务端热下盘，网络层主动反压与限流等等。

另外，随着美团实时计算业务整体的发展，实时计算引擎（典型如Flink）和流存储引擎（典型如Kafka）混合部署的模式越来越难以满足业务的需求。因此，我们需要在保持当前成本不变的情况下对Kafka进行独立部署。这就意味着需要用更少的机器（在我们的业务模式下，用原来1/4的机器）来承载不变的业务流量。如何在保障服务稳定的情况下，用更少的机器扛起业务请求，也是我们面临的挑战之一。

最后，随着云原生趋势的来临，我们也在探索流存储服务的上云之路。

## 5 作者简介

海源、仕禄、肖恩、鸿洛、启帆、胡荣、李杰等，均来自美团数据科学与平台部。