限流比降 10 倍：百炼网关如何用 RocketMQ LiteTopic 重构大模型限流

原创 不铭、文婷、稚柳 千问AI平台 2026年7月10日 11:35 浙江

阿里妹导读

文章内容基于作者个人技术实践与独立思考，旨在分享经验，仅代表个人观点。

百炼是阿里云的大模型服务平台，承载百万级用户同时调用千问等数十种大模型。作为国内最大的公有云模型推理入口之一，百炼网关每天处理海量异构推理请求——**每一次限流误判，都意味着用户侧的超时、重试，甚至业务中断；每一次过载放行，都意味着稀缺 GPU 算力的浪费和其他租户的体验劣化。**

当平台用户从千级增长到百万级，限流系统面临的已不再是“防刷”这种老问题，而是“**如何在有限 GPU 池中实现百万租户各自独立、按需弹性的精细化流量治理**”。百炼网关团队用一次端到端重构回答了这个问题——核心技术选型是 RocketMQ LiteTopic。方案上线后，限流比降低了 10 倍，限流导致的用户可感知异常大幅收敛。

以下是这次重构的完整技术复盘。

1. 大模型时代，限流从粗放走向精细化

过去十年，CPU、内存、带宽都能在分钟级甚至秒级弹起来，互联网应用的限流主要解决“防刷”和“防雪崩”，处理比较直接。

进入大模型时代，**GPU 扩容周期长、单价高，供给还受制于硬件交付节奏**。对公有云模型服务而言，“GPU 池子有多大”几乎直接决定“今天能服务多少客户、能承诺多高的 SLA”。

作为模型服务的流量入口，百炼网关要在有限的 GPU 池里同时兑现三件事：

- **租户级流量隔离**：各租户请求流量相互隔离，有效防止多租户间的资源争抢与相互干扰，避免个别租户的突发流量影响其他租户的正常使用。

- **精细化配额限流**：支持租户维度的差异化配额管理，仅对超额请求进行限流退避，未达配额的租户不受任何限制。

- **平滑突发与资源统筹**：通过流量整形削峰填谷，在有限的GPU资源池内进行合理统筹与分配，保障服务体验平稳、低抖动。

要同时做好这三件事，传统“一个限流器卡在网关上”的范式已经不够用。百炼网关团队在过去几个月对限流算法到底层消息队列做了一次**端到端重构——把 RocketMQ LiteTopic 作为漏桶（Leaky Bucket）的物理载体**，构建出一套面向大模型场景的精细化限流体系。本文分享其中的设计取舍与工程经验。

2. 为什么传统限流模式不够用了？

百炼面对的并非单一限流场景，而是三类叠加：

- **基于 SLA 承诺的基础限流**：需要稳定、可计量、可观测；

- **用户之间及客户内部多账号的细粒度管控**：要在 User + Model 维度做交叉约束；

- **超大客户的扩容与突发承接**：额度调到很大之后，一旦突发真的来，GPU 在扩容到位前根本顶不住——硬限会大面积返回 503 影响体验，放过去又会让 GPU 过载、殃及其他租户。

三者交织，意味着限流不只是“决定放不放过”，更要回答“放过去之后以什么节奏喂给后端”。百炼网关在算法上最终选定**固定窗口 + 漏桶**：

- **固定窗口而非滑动窗口**：滑动窗口更加严格，边界处的突发也会被精确计算；固定窗口则对短期波动更加宽容。

- **漏桶而非令牌桶**：漏桶“匀速放行”更契合 GPU 这种稳态友好、尖峰敏感的下游，而令牌桶天然允许突发，恰恰是 GPU 最怕的形态。

方案选定后，新问题随之而来：超大客户的突发流量如果堆在网关进程内存里，**几十万请求会立刻把网关本身打挂**。漏桶必须**从进程内搬到进程外**，用一个足够大、足够隔离的外部存储承接缓冲，把“接住请求”和“按节奏放行”在物理上拆开。这是 LiteTopic 进入视野的起点。

3. 传统 Topic/Group 无法撑起平台级漏桶

只看“用 MQ 做漏桶”这一步，传统 RocketMQ Topic 也能做——百炼最早就是这么落地的：为每个 User + Model 单独建 Topic 与 Consumer Group，再部署一组消费机器专门负责这条链路。但随着头部客户陆续接入，三个痛点很快暴露。

- **元数据重、生效慢**：传统 Topic/Group 需要预先创建，生效时间在几十秒量级，新用户突发流量到来时会出现短时间异常。
- **机器利用率低、成本随客户数膨胀**：共享消费模型要求同一个 Consumer Group 下的客户端订阅完全相同的 Topic 集合，否则会出现订阅不一致导致的堆积甚至丢失。要隔离客户流量，就必须给每个客户单独建 Group、甚至单独切机器；哪怕客户今天一条消息都没有，名下的那组机器仍要常驻。
- **单 Topic 暴涨的株连效应**：若硬把多客户数据塞进同一 Topic，一旦某个用户消息激增，几乎所有消费线程都被它占据，其他用户消息全部被阻塞。团队最终不得不退回到“一客户一组机器”，用机器规模换隔离强度。

事实是：传统方案能服务“少数几个头部突发客户”，但无法支撑“任意中大型客户都能自动接入漏桶”。**当限流要从“个别 VIP 待遇”变成“平台基础能力”时，底层架构必须重新设计。**

4. LiteTopic：把限流的“重资产”变成“轻资产”

LiteTopic 是 RocketMQ 5.x 引入的轻量级队列形态，与传统 Topic 最关键的三个差异正好对应上面那三个痛点。

- **轻量元数据**：单 Broker 支撑**百万级 LiteTopic**，运行时按需创建、按 TTL 自动回收。客户端往不存在的 LiteTopic 上发消息即可自动建出，无活动一段时间后由 Broker 自动清理。
- **差异化订阅**：同一 Consumer Group 下，不同消费实例可订阅不同的 LiteTopic 子集，且不会引发堆积或丢失。订阅关系从“刚性约束”变成“按需路由”。
- **Suspend 消费控制**：业务在消费回调中返回 Suspend N（毫秒），即可让 Broker 在 N 毫秒内暂停对该 LiteTopic 的拉取，且**不影响其他 LiteTopic 的拉取节奏**。这是漏桶在 Broker 层落地的关键开关。

基于这三点，百炼网关限流架构被重构如下：

- **发送侧**：通过基础限流校验后，按 User + Model 写入对应 LiteTopic，User 与 Model 信息直接编进名字，**天然多租户物理隔离**。
- **消费侧**：将原来按客户切分的几十组机器合并为统一一组 Pod，通配符订阅全部 LiteTopic，新增客户时发送侧自动建、消费侧自动拉，**无需改代码、无需重启、无需扩机器**。

- **限流逻辑**：收敛到消费线程内几行判断——命中则返回 Suspend N，否则转发模型并 ACK。Broker 在指定毫秒内不再投递该 LiteTopic，到期自动恢复。

可以把这套机制理解为一个**“分布式漏桶矩阵”**：每个 User + Model 拥有一个独立漏桶，桶容量由 LiteTopic 堆积上限与 TTL 决定，出水速率由消费侧 Suspend 时长动态决定。整个矩阵共享同一组消费机器，资源池随总流量伸缩，而非按客户数线性增长。

5. LiteTopic 为何是漏桶的天然载体?

把 LiteTopic 当作漏桶的物理载体，并不只是“换一个队列实现”，它在三个维度上恰好对上了“GPU 稀缺 + 多租户 + 突发尖峰”的场景。

- **桶的容量在工程意义上近似无限**：传统漏桶以进程内队列实现，容量是一个写死的数字，桶满即拒。LiteTopic 数据持久化到 Broker 磁盘，单实例承载百万级队列，堆积上限远超进程内队列，且彼此物理隔离。客户尖峰被“**接住、攒起来、按节奏放行**”，“等几秒再处理”几乎在所有场景下都优于“被 429”。

- **每个 User + Model 各拥有独立漏桶**：LiteTopic 把租户隔离变成简单的“命名规则”问题——客户 A 的 Model X 进入 liteTopic-A-X，客户 B 的 Model X 进入 liteTopic-B-X，桶与桶在 Broker 层物理隔离，A 链路堵了不会通过任何共享资源传染到 B。租户隔离从“靠业务代码维护”变成“**靠基础设施天然提供**”。

- **每个漏桶速率可独立、动态调整**：这是最关键的一项能力。Suspend N 完全由业务策略实时计算，百炼据此可在毫秒级调整任意一个漏桶的放行速度：重要客户在模型负载高时仍能拿到较高节奏；模型扩容到位后矩阵速率同步上调；某款模型出现拥塞苗头，单独把对应那一列漏桶收紧即可，不拖累其他模型。整个过程**不重启消费组、不改 LiteTopic 配置、不需要客户感知**。

三点结合，本质上把漏桶从“被动的削峰工具”升级为**多租户友好、容量近似无限、速率可调度**的基础设施。

6. 最小改造：发送、订阅、限流各一段代码

核心改动可以浓缩成三段（以下为简化示意，展示核心调用逻辑）：

1.发送：构造消息时设置 LiteTopic，按客户维度自动路由，LiteTopic 不存在时由 Broker 自动创建。

```
// 网关侧：把请求按 user + model 路由到对应的 LiteTopic
Message msg = new Message(parentTopic, payload);
msg.setLiteTopic(buildLiteTopicName(user, model)); // 新增的一行
producer.send(msg);
```

2.消费：启动时通配符订阅 ParentTopic 下的全部 LiteTopic，新客户与新 LiteTopic 由消费组自动感知，无需维护清单、无需重启。

```
// 消费侧：一次订阅，覆盖当前和未来所有 LiteTopic
PushConsumer consumer = new PushConsumer("bailian-rate-limit-group");
consumer.subscribe("bailian-rate-limit-parent", "*"); // 通配符订阅
consumer.start();
```

3.限流逻辑：消费回调里几行代码，命中策略则 Suspend（N），未命中则正常调用模型并 ACK。

```
@Override
public ConsumeResult consume(MessageView msg) {
    String liteTopic = msg.getLiteTopic();
    long suspendMs = rateLimitPolicy.acquireOrSuspend(liteTopic);
    if (suspendMs > 0) {
        // 仅暂停该 LiteTopic 的拉取，其余 LiteTopic 不受影响
        return ConsumeResult.Suspend(suspendMs);
    }
    invokeModel(msg);
    return ConsumeResult.SUCCESS;
}
```

7. 几个关键的工程技术细节

1.海量LiteTopic下的消费性能

同时几十万 User + Model 并发是日常态。若沿用“对每个订阅的 LiteTopic 各发起一个独立 Pull 循环”的传统做法，开销随订阅数线性增长——本质上和 select/poll 一样：每次调用都要遍历全量集合，大部分遍历无效。

LiteTopic 在 Broker 侧引入了**就绪集合（Ready Set）**事件驱动机制，思路类似 epoll：只有真正有新消息写入的 LiteTopic 才进入就绪集合，Pull 时 Broker 只读取就绪 LiteTopic，集合为空则长轮询等待事件触发（新消息到达、ACK 后仍有未消费消息、顺序锁释放等）。Pull 开销因此只与“**当前时刻就绪的 LiteTopic 数量**”成正比，而非“总订阅数”。

POC 压测中，单 Broker 承载 **200 客户端 × 1 万 LiteTopic（共 200 万队列）**，平均消费延迟稳定在 **12ms**；同等订阅量下传统逐 Topic 轮询，消息稀疏时 CPU 高出数倍且随订阅数恶化。

2.Suspend的精度

当前最小粒度为 **30 毫秒**，更细的速率控制需要消费侧自己做局部 Sleep。这个台阶对绝大多数模型推理场景够用，推理本身就是百毫秒到秒级；但若业务目标速率非常高，需要把 Suspend 步长和并发线程数放在一起算实际放行速率，避免阶梯感传导到 P99 RT。

3.Suspend vs消费线程Sleep

多租户共享消费组下，所有 LiteTopic 共用同一线程池（POC 中 50 线程），限流方式直接决定其他 LiteTopic 是否会被“误伤”。

POC 对比：订阅 50 个 LiteTopic、前 5 个随机触发 300ms 限流。

- Thread.sleep(300)时：被限流的 LiteTopic 各占住一个线程不释放，当被限流数增至 40 个时线程池几乎被耗尽，剩下 10 个正常 LiteTopic 拿不到线程而全部堆积——限流“传染”到了不该被限流的租户。

- Suspend 300时：消费线程返回后**立即释放回线程池**，转头服务其他 LiteTopic；被 Suspend 的 LiteTopic 由 Broker 在 N 毫秒后重投，整个过程**不占客户端任何线程**，结果是仅被限流的 5 个堆积、其余 45 个正常。

本质区别：Sleep 是**线程级阻塞**，会通过共享线程池扩散到无关租户；Suspend 是 **Broker 级拉取流控**，与其他漏桶在线程层面完全解耦。在“大量用户同时被限流、少数用户正常放行”的常态下，Suspend 的非阻塞特性保证了公平性——无论多少 User + Model 正在被限流，剩余用户始终有空闲线程可用。

8. 写在最后

回过头看这次重构，百炼网关从“进程内漏桶”走到“分布式漏桶矩阵”，核心不是算法多新颖，而是找到了一个能让**隔离、调速、弹性**同时成立的工程载体。

过程中有几条经验，值得同样面对大模型限流挑战的团队参考：

- **限流不是单一算法，而是分层组合**。固定窗口管硬上限、漏桶管放行节奏、消息队列管承接缓冲，三层串联缺一不可。

- **漏桶的物理载体决定了上限**。当限流额度大到突发流量可能压垮网关进程，漏桶必须从进程内搬到进程外。MQ 是天然候选，但要进一步追问：能否支持百万级队列、能否运行时按需创建、流控粒度是否足够精细。

- **LLM 场景对租户隔离的要求更高，但隔离的成本必须更低**。大模型推理一次请求消耗数秒 GPU 时间，被挤占损失的不只是成功率，还有已消耗的昂贵算力，隔离因此成为刚需。但若靠“每个租户切一组独立机器”，成本随客户数乘法膨胀，公有云无法持续。LiteTopic 的方案是：每个 User + Model 拥有物理隔离的队列，所有队列共享同一组消费 Pod，资源池随总流量伸缩而非随客户数膨胀，隔离强度不降、成本从乘法变回加法。

- **Suspend 与 Sleep 各有适用场景，关键在于“谁为等待买单”**。Suspend 适合粗粒度、长时间的限流等待——暂停整个 LiteTopic 拉取、线程立即释放、不影响其他租户；Sleep 适合短时间的处理等待——比如 Suspend 30ms 精度不够时做亚毫秒级节奏控制，持续时间极短、对线程池影响可控。百炼实际是两者组合：Suspend 承担主要漏桶调速，Sleep 在极少数亚 30ms 精度场景中补充。

从业务结果看，LiteTopic 提供的“**轻量队列 + 差异化订阅 + Suspend 控速**”三件套，让百炼网关的**限流比降低了 10 倍**。

而这套方案并非百炼独有的定制工程——任何大模型平台或推理服务，只要面对“GPU 稀缺 + 海量租户 + 突发尖峰”这三个约束，都会遇到同样的限流困境。**百万级物理隔离队列 + 动态速率调度 + 零运维弹性**，是一套可直接复用的基础设施范式。

大模型时代，限流不再是一道防御题，而是一道资源调度题。当 GPU 成为最稀缺的生产要素，谁能把有限算力更精细、更公平、更弹性地分配给每一个租户，谁就能在体验和成本之间找到最优解。这正是 RocketMQ LiteTopic 在这个时代的价值所在。

点击阅读原文即可体验！

阅读原文