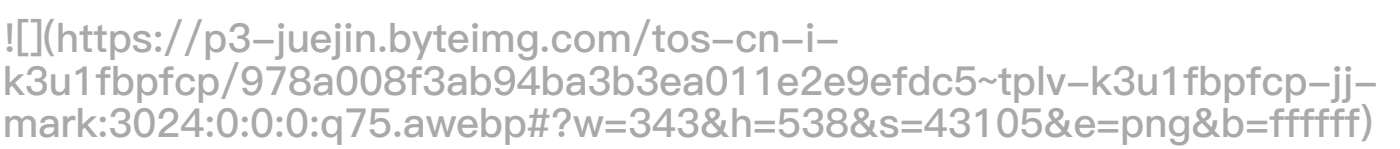
消息队列的七种经典应用场景

在笔者心中，**消息队列**，**缓存**，**分库分表**是高并发解决方案三剑客。在职业生涯中，笔者曾经使用过 ActiveMQ 、RabbitMQ 、Kafka 、RocketMQ 这些知名的消息队列 。

这篇文章，笔者结合自己的真实经历，和大家分享消息队列的七种经典应用场景。



1 异步&解耦

笔者曾经负责某电商公司的用户服务，该服务提供用户注册，查询，修改等基础功能。用户注册成功之后，需要给用户发送短信。 

图中，**新增用户**和**发送短信**都揉在用户中心服务里，这种方式缺点非常明显：

0. 短信渠道不够稳定，发送短信会达到 5 秒左右，这样用户注册接口耗时很大，影响前端用户体验;
1. 短信渠道接口发生变化，用户中心代码就必须修改了。但用户中心是核心系统。每次上线都必要谨小慎微。这种感觉很别扭，非核心功能影响到核心系统了。

为了解决这个问题，笔者采用了消息队列进行了重构。

mark:3024:0:0:0:q75.awebp#?w=1052&h=153&s=7368&e=webp&b=f9eee4)

* **异步**

用户中心服务保存用户信息成功后，发送一条消息到消息队列 ,立即将结果返回给前端，这样能避免总耗时比较长，从而影响用户的体验的问题。
* **解耦**

任务服务收到消息调用短信服务发送短信，将核心服务与非核心功能剥离，显著的降低了系统间的耦合度。

2 消峰
====

高并发场景下，面对突然出现的请求峰值，非常容易导致系统变得不稳定，比如大量请求访问数据库，会对数据库造成极大的压力，或者系统的资源 CPU 、IO 出现瓶颈。

笔者曾服务于神州专车订单团队，在订单的载客生命周期里，订单的修改操作先修改订单缓存，然后发送消息到 MetaQ ，订单落盘服务消费消息，并判断订单信息是否正常（比如有无乱序），若订单数据无误，则存储到数据库中。

当面对请求峰值时，由于消费者的并发度在一个阈值范围内，同时消费速度相对均匀，因此不会对数据库造成太大的影响，同时真正面对前端的订单系统生产者也会变得更稳定。

3 消息总线
======

所谓总线，就是**像主板里的数据总线一样， 具有数据的传递和交互能力，各方不直接通信，使用总线作为标准通信接口**。

笔者曾经服务于某彩票公司订单团队，在彩票订单的生命周期里，经过创建，拆分子订单，出票，算奖等诸多环节。 每一个环节都需要不同的服务处理，每个系统都有自己独立的表，业务功能也相对独立。假如每个应用都去修改订单主表的信息，那就会相当混乱了。

因此，公司的架构师设计了**调度中心**的服务，调度中心维护订单的信息，但它不与子服务通讯，而是通过**消息队列**和出票网关，算奖服务等系统传递和交换信息。

消息总线这种架构设计，可以让系统更加解耦，同时也可以让每个系统各司其职。

4 延时任务
======

用户在美团 APP 下单，假如没有立即支付，进入订单详情会显示倒计时，如果超过支付时间，订单就会被自动取消。

非常优雅的方式是：**使用消息队列的延时消息**。

订单服务生成订单后，发送一条延时消息到消息队列。消息队列在消息到达支付过期时间时，将消息投递给消费者，消费者收到消息之后，判断订单状态是否为已支付，假如未支付，则执行取消订单的逻辑。

RocketMQ 4.X 生产者发送延迟消息代码如下：

```
Message msg = new Message();
msg.setTopic(”TopicA”);
```

```
msg.setTags(”Tag”);
msg.setBody(”this is a delay message”.getBytes());
//设置延迟level为5，对应延迟1分钟
msg.setDelayTimeLevel(5);
producer.send(msg);
```

RocketMQ 4.X 版本默认支持 18 个 level 的延迟消息， 通过 broker 端的 messageDelayLevel 配置项确定的。

RocketMQ 5.X 版本支持任意时刻延迟消息，客户端在构造消息时提供了 3 个 API 来指定延迟时间或定时时间。

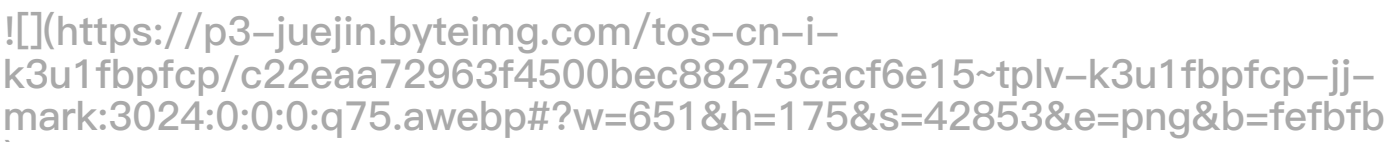
5 广播消费
======

广播消费：当使用广播消费模式时，每条消息推送给集群内所有的消费者，保证消息至少被每个消费者消费一次。

广播消费主要用于两种场景：**消息推送**和**缓存同步**。

01 消息推送

下图是专车的司机端推送机制，用户下单之后，订单系统生成专车订单，派单系统会根据相关算法将订单派给某司机，司机端就会收到派单推送消息。



推送服务是一个 TCP 服务（自定义协议），同时也是一个消费者服务，消息模式是广播消费。

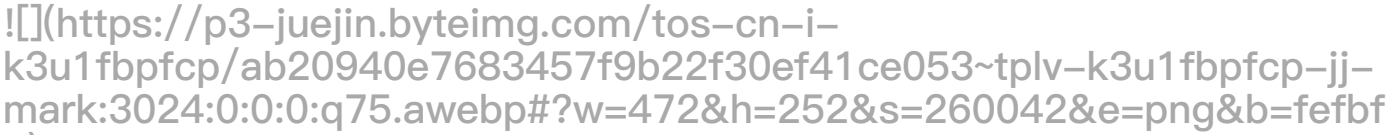
司机打开司机端 APP 后，APP 会通过负载均衡和推送服务创建长连接，推送服务会保存 TCP 连接引用 （比如司机编号和 TCP channel 的引用）。

派单服务是生产者，将派单数据发送到 MetaQ，每个推送服务都会消费到该消息，推送服务判断本地内存中是否存在该司机的 TCP channel， 若存在，则通过 TCP 连接将数据推送给司机端。

02 缓存同步

高并发场景下，很多应用使用本地缓存，提升系统性能 。

本地缓存可以是 HashMap 、ConcurrentHashMap ，也可以是缓存框架 Guava Cache 或者 Caffeine cache 。

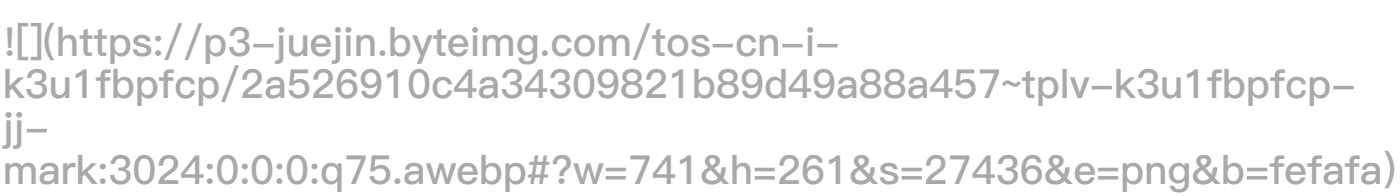


如上图，应用A启动后，作为一个 RocketMQ 消费者，消息模式设置为广播消费。为了提升接口性能，每个应用节点都会将字典表加载到本地缓存里。

当字典表数据变更时，可以通过业务系统发送一条消息到 RocketMQ ，每个应用节点都会消费消息，刷新本地缓存。

6 分布式事务
=======

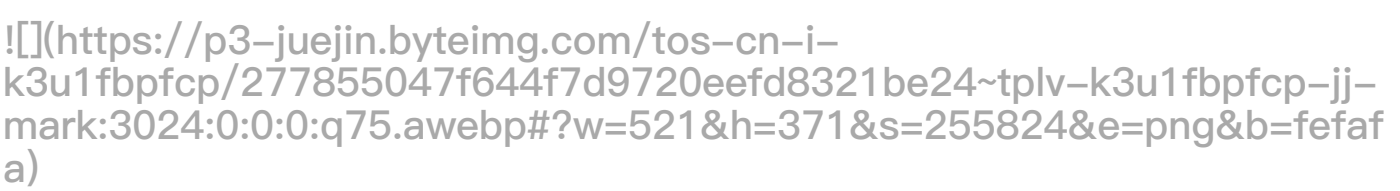
以电商交易场景为例，**用户支付订单**这一核心操作的同时会涉及到下游物流发货、积分变更、购物车状态清空等多个子系统的变更。



1、传统XA事务方案：性能不足

为了保证上述四个分支的执行结果一致性，典型方案是基于 XA 协议的分布式事务系统来实现。将四个调用分支封装成包含四个独立事务分支的大事务。基于 XA 分布式事务的方案可以满足业务处理结果的正确性，但最大的缺点是多分支环境下资源锁定范围大，并发度低，随着下游分支的增加，系统性能会越来越差。

2、基于普通消息方案：一致性保障困难



该方案中消息下游分支和订单系统变更的主分支很容易出现不一致的现象，例如：

* 消息发送成功，订单没有执行成功，需要回滚整个事务。
* 订单执行成功，消息没有发送成功，需要额外补偿才能发现不一致。
* 消息发送超时未知，此时无法判断需要回滚订单还是提交订单变更。

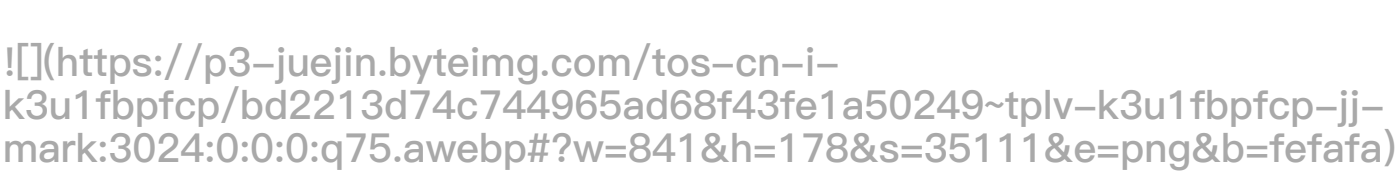
3、基于 RocketMQ 分布式事务消息：支持最终一致性

上述普通消息方案中，普通消息和订单事务无法保证一致的原因，本质上是由于普通消息无法像单机数据库事务一样，具备提交、回滚和统一协调的能力。

而基于 RocketMQ 实现的分布式事务消息功能，在普通消息基础上，支持二阶

段的提交能力。将二阶段提交和本地事务绑定，实现全局提交结果的一致性。

RocketMQ 事务消息是支持在分布式场景下**保障消息生产和本地事务的最终一致性**。交互流程如下图所示：



1、生产者将消息发送至 Broker 。

2、Broker 将消息持久化成功之后，向生产者返回 Ack 确认消息已经发送成功，此时消息被标记为”**暂不能投递**”，这种状态下的消息即为**半事务消息**。

3、生产者开始**执行本地事务逻辑**。

4、生产者根据本地事务执行结果向服务端**提交二次确认结果**（ Commit 或是 Rollback ），Broker 收到确认结果后处理逻辑如下：

* 二次确认结果为 Commit ： Broker 将半事务消息标记为可投递，并投递给消费者。
* 二次确认结果为 Rollback ： Broker 将回滚事务，不会将半事务消息投递给消费者。

5、在断网或者是生产者应用重启的特殊情况下，若 Broker 未收到发送者提交的二次确认结果，或 Broker 收到的二次确认结果为 Unknown 未知状态，经过固定时间后，服务端将对消息生产者即生产者集群中任一生产者实例发起**消息回查**。

0. 生产者收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
1. 生产者根据检查到的本地事务的最终状态**再次提交二次确认**，服务端仍按照步骤4对半事务消息进行处理。

7 数据中转枢纽
========

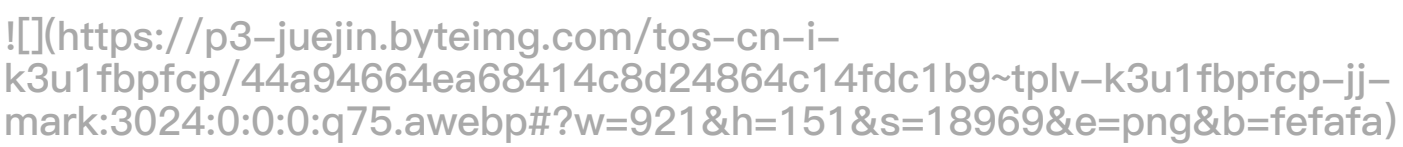
近10多年来，诸如 KV 存储（HBase）、搜索（ElasticSearch）、流式处理（Storm、Spark、Samza）、时序数据库（OpenTSDB）等专用系统应运而生。这些系统是为单一的目标而产生的，因其简单性使得在商业硬件上构建分布式系统变得更加容易且性价比更高。

通常，同一份数据集需要被注入到多个专用系统内。

例如，当应用日志用于离线日志分析时，搜索单个日志记录同样不可或缺，而构建各自独立的工作流来采集每种类型的数据再导入到各自的专用系统显然不切实际，利用消息队列 Kafka 作为数据中转枢纽，同份数据可以被导入到不同专用系统中。

日志同步主要有三个关键部分：日志采集客户端，Kafka 消息队列以及后端的日志处理应用。

1. 日志采集客户端，负责用户各类应用服务的日志数据采集，以消息方式将日志“批量”“异步”发送Kafka客户端。 Kafka客户端批量提交和压缩消息，对应用服务的性能影响非常小。
2. Kafka 将日志存储在消息文件中，提供持久化。
3. 日志处理应用，如 Logstash，订阅并消费Kafka中的日志消息，最终供文件搜索服务检索日志，或者由 Kafka 将消息传递给 Hadoop 等其他大数据应用系统化存储与分析。



如果我的文章对你有所帮助，还请帮忙**点赞、在看、转发**一下，你的支持会激励我输出更高质量的文章，非常感谢！

原文链接: https://juejin.cn/post/7350947906566815778