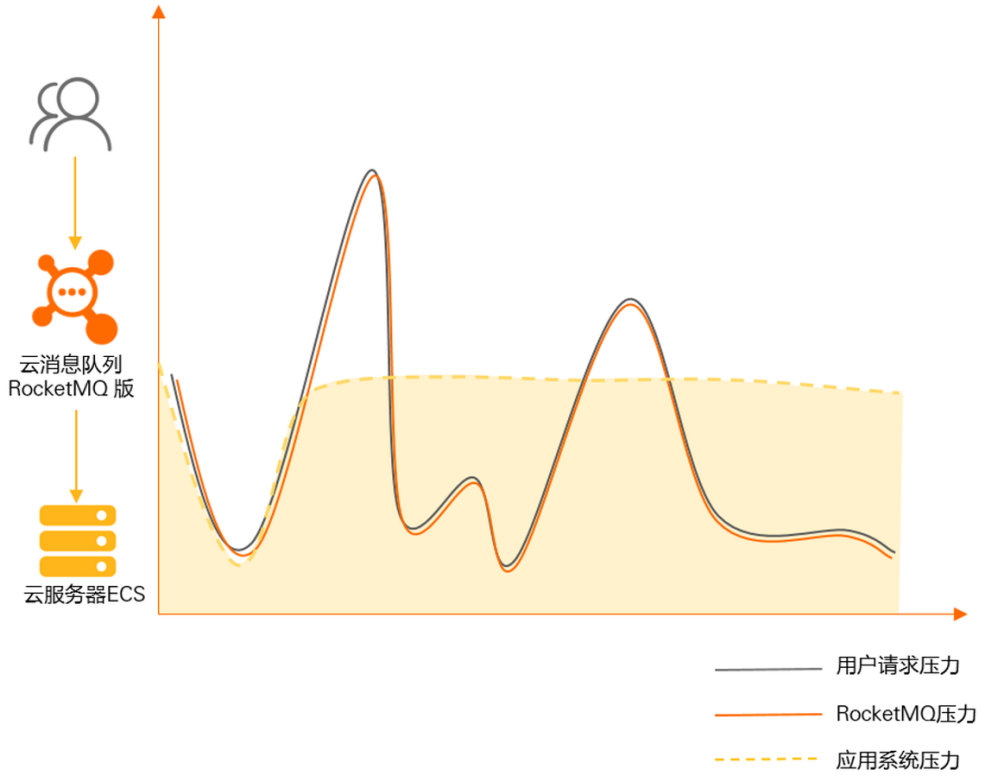
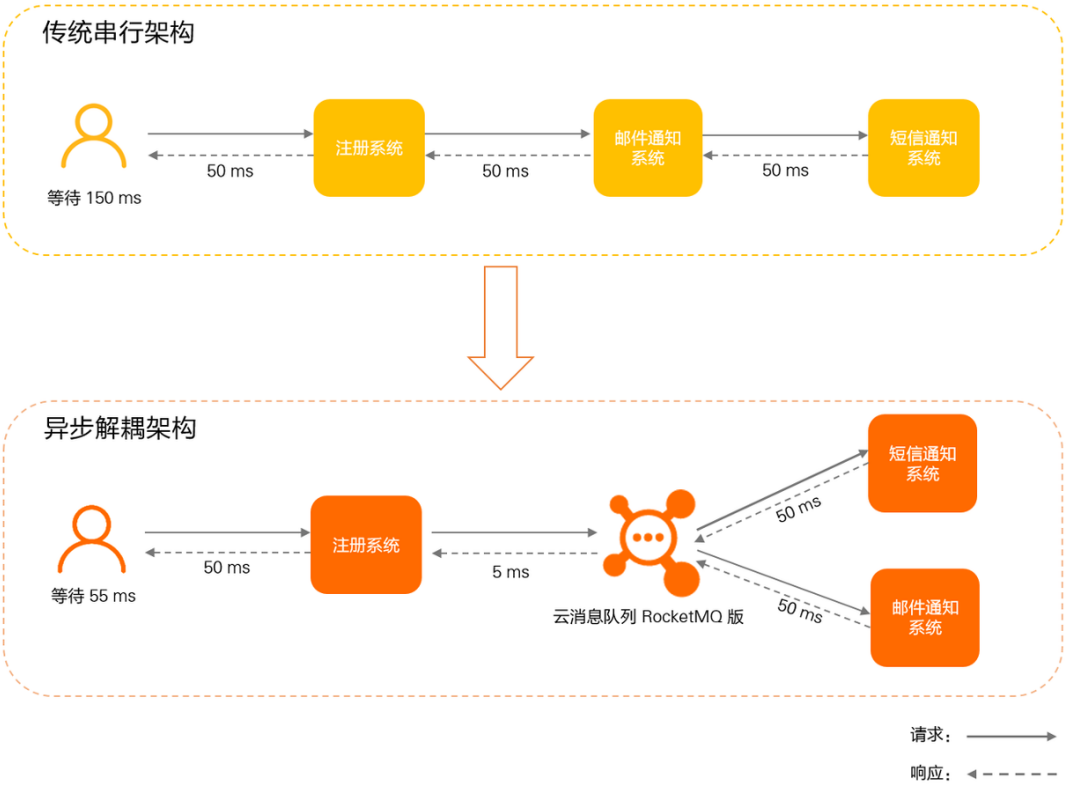
MetaQ技术分享

第一部分：消息队列基础介绍

- **定义与作用**：消息队列，作为一种中间件技术，是一种在分布式系统中实现应用程序间异步通信的数据结构。它允许程序将需要处理的任务封装成消息并放入队列中，而无需立即处理。队列中的消息随后会被相应的接收者（消费者）异步地取出并处理。这一机制有效解决了高并发处理、系统解耦、流量削峰填谷等多种挑战。



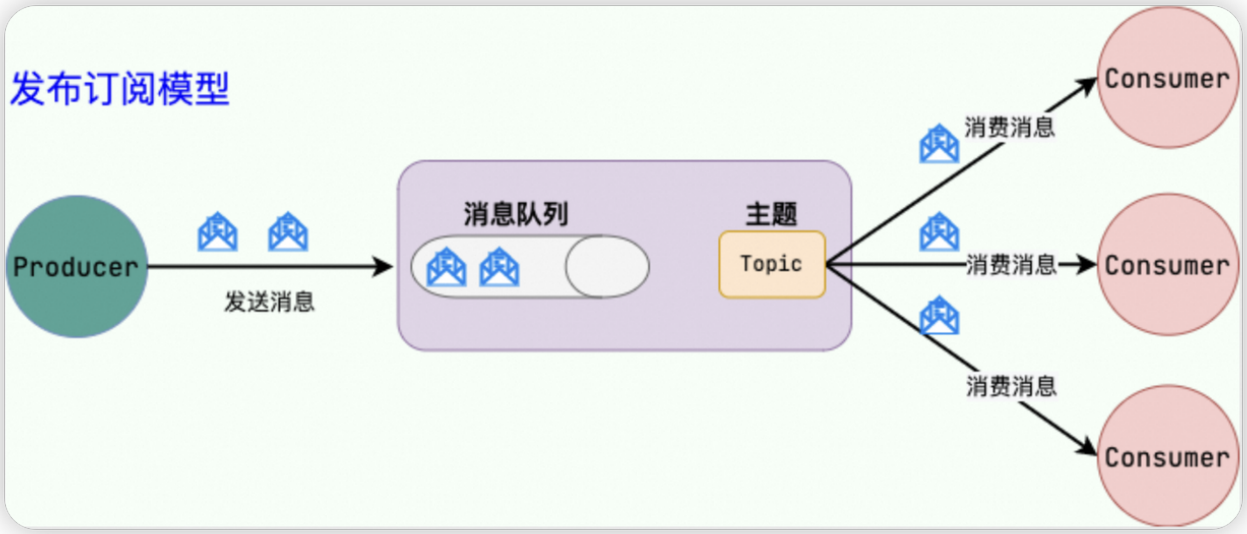
流量削峰填谷



异步解耦

- **应用场景**：
  a. **订单处理系统**：在电商系统中，用户下单后，订单系统通过消息队列将订单信息发送给库存系统、物流系统等，各系统异步处理，保证了高并发下单时的系统稳定性。
  b. **用户通知服务**：当需要向大量用户发送电子邮件、短信或推送通知时，应用服务器将通知消息投递到消息队列，由专门的消费者服务负责发送，避免了因发送操作耗时导致的服务器响应延迟。
  c. **数据同步与备份**：在大数据处理场景中，消息队列用于收集各类数据变更事件，异步同步至数据分析平台或备份系统，确保数据的一致性和完整性。
- **核心概念**：
  a. **生产者(Producer)**：创建并发送消息的应用程序或服务。它负责生成消息并将消息放入消息队列中。
  b. **消费者(Consumer)**：从消息队列中接收并处理消息的应用程序或服务。一个消息可以被单个消费者消费，也可以被多个消费者（通过发布/订阅模式）共享处理。
  c. **消息(Message)**：生产者和消费者之间传递的数据单元，通常包含数据本身及一些元数据（如消息ID、发送时间等）。
  d. **队列(Queue)**：一种先进先出（FIFO）的数据结构，用于存储等待处理的消息。生产者向队列中添加消息，消费者从队列中取出并处理消息。

e. **主题（Topic）**：一种分类标识，用来对消息进行分类和路由。生产者将消息发布到特定的主题上，而不是直接发送到队列或某个特定的消费者。消费者则通过订阅他们感兴趣的主题来接收消息。



发布订阅模型

第二部分：常用消息队列框架及性能对比

- **概述**：在众多消息队列技术中，RabbitMQ、Apache Kafka、RocketMQ（MetaQ）是当前市场上最为广泛应用的几款产品，各自有着鲜明的特点和适用场景：
  a. **RabbitMQ**：基于AMQP（Advanced Message Queuing Protocol）标准的消息队列服务，提供了丰富的路由策略和灵活的消息确认机制，适用于需要复杂消息路由逻辑和高可用性的应用场景。
  b. **Apache Kafka**：最初由LinkedIn开发，专为高吞吐量实时处理场景设计，尤其适合日志收集、流处理和大规模事件驱动架构。Kafka采用发布/订阅模型，并支持数据持久化，具有极高的数据吞吐能力。
  c. **RocketMQ(MetaQ)**：起源于阿里巴巴，现为Apache顶级项目，专为大规模分布式系统设计，强调高吞吐量、低延迟和高可用性。特别适合金融、电商等领域中对消息可靠性有严格要求的场景。
- **性能指标**：
  a. **吞吐量**：单位时间内系统能够处理的消息数量，直接关系到系统处理能力。
  b. **延迟**：消息从生产到被消费的时间差，影响实时性需求。
  c. **可靠性**：确保消息不丢失、不重复，是金融、支付等关键领域的重要考量。
- **对比分析**：

a. **吞吐量**：Kafka以其分布式架构和高效的磁盘I/O处理，在高吞吐量场景下表现突出；RocketMQ也具备高吞吐特性；相比之下，RabbitMQ更侧重于提供丰富功能和高可用性，吞吐量相对较低。

b. **延迟**：RocketMQ和Kafka都能达到毫秒级的低延迟，但在特定配置下Kafka的端到端延迟更低，特别适合实时数据处理。RabbitMQ 在这三者中拥有最低的延迟，它的延迟可以达到微秒级。

c. **可靠性**：RocketMQ和RabbitMQ在消息可靠性方面提供了丰富的保障机制，如事务消息、消息确认等，适合对数据一致性要求极高的场景；Kafka通过副本机制也保证了较高的消息持久性。

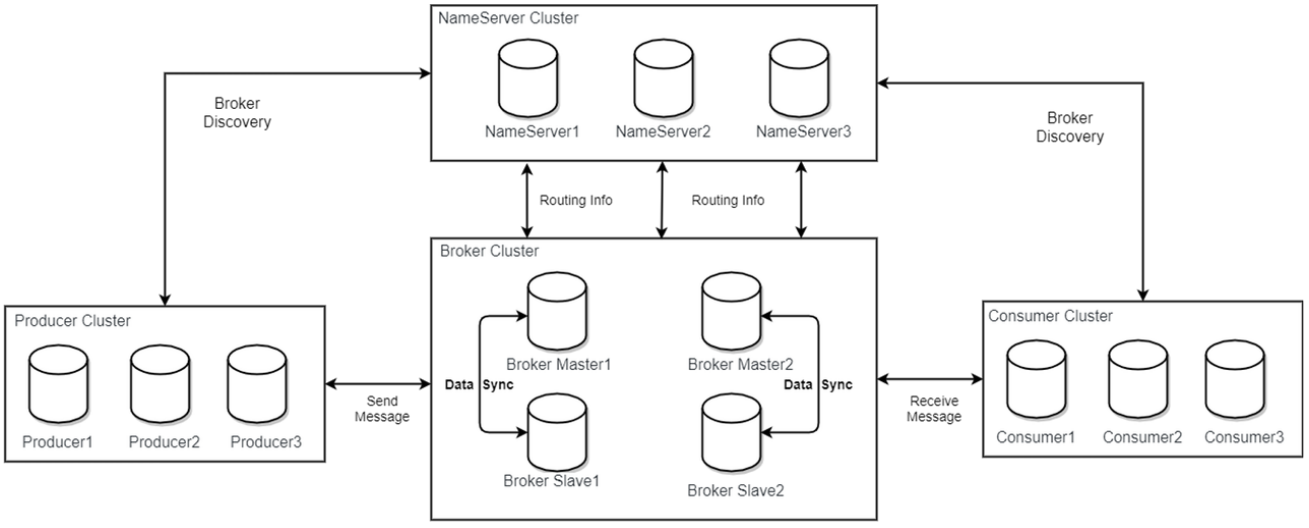
- **选择考量**：

a. **对于金融级的高可靠性需求**，RocketMQ或RabbitMQ可能是更好的选择，它们提供了更全面的消息确认机制和事务支持。

b. **针对大数据处理和实时流处理场景**，Apache Kafka凭借其高吞吐量和低延迟特性成为首选。

c. **需要灵活的消息路由和复杂的业务逻辑**时，RabbitMQ的AMQP协议支持和丰富的路由策略使其成为优选。

第三部分：MetaQ性能特性解析



MetaQ部署架构

- **技术架构**：MetaQ作为一款分布式消息中间件，其设计充分考虑了 高性能、高可用及可扩展性需求，旨在为企业级应用提供稳定可靠的消息服务。其架构设计核心要点如下：

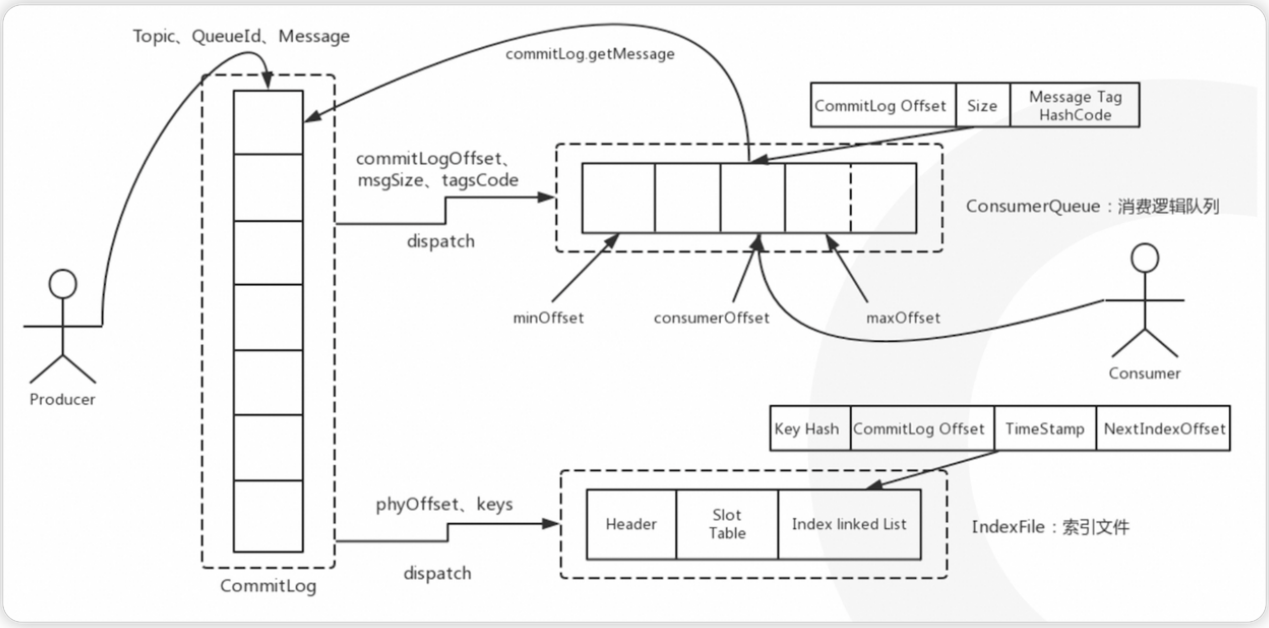
a. **集群部署**：MetaQ采用分布式集群部署模式，每个节点既是生产者也是消费者，能够均衡负载，提高系统整体处理能力。集群通过NameServer等协调服务进行元数据管理与配置同步，确保各节点状态的一致性。

b. **高可用机制**：为确保服务的连续性，MetaQ实施了多维度的高可用策略。这包括但不限于主备切换机制、分区副本机制以及自动故障转移。每个主题的消息会被切分为多个分区，并在不同节点上保存副本，即使单个节点故障也不会影响消息的正常处理。

c. **数据持久化策略**：MetaQ支持多种数据持久化方式，包括磁盘存储与内存缓存相结合的策略，以平衡性能与可靠性。消息在被确认消费之前会持久化到磁盘，数据持久化操作提升了消息堆积的能力，内存缓存机制进一步提升了消息处理的即时性。

d. **低延时保障**：通过消息预取、异步刷盘及智能路由等策略，MetaQ能在高吞吐的同时保持较低的消息端到端延迟。预取机制允许消费者提前拉取消息，减少等待时间；异步刷盘则保证了消息的快速确认，不影响发送流程。

- **数据存储模型:**



MetaQ数据存储模型

a. Commitlog

CommitLog是MetaQ存储消息的主体部分，所有主题（Topic）的消息内容都被顺序写入这个文件中，实现了高度顺序写磁盘操作，这对于提升写入性能至关重要。由于消息的长度不固定，CommitLog中的每条记录长度也不定长。每个CommitLog文件的默认大小为1GB（1073741824字节），文件名长度为20位，左侧用零填充，剩余部分表示该文件中消息的起始偏移量。例如，第一个文件名为 `00000000000000000000` ，起始偏移量为0；当这个文件写满后，下一个文件名为 `00000000001073741824` ，起始偏移量为1GB。这种顺序写入的方式非常有利于磁盘I/O性能的优化。

b. ConsumeQueue

ConsumeQueue，也称为消费队列，是为了提高消息消费的效率而设计的。每个主题下的每个消息队列（Queue）都有一个对应的ConsumeQueue文件。与CommitLog不同，ConsumeQueue中的记录是定长的，通常每个记录包含20个字节的消息存储偏移量、16个字节的消息长度和8个字节的Tag Hashcode。这样的设计使得Consumer可以从ConsumeQueue中快速定位到CommitLog中的消

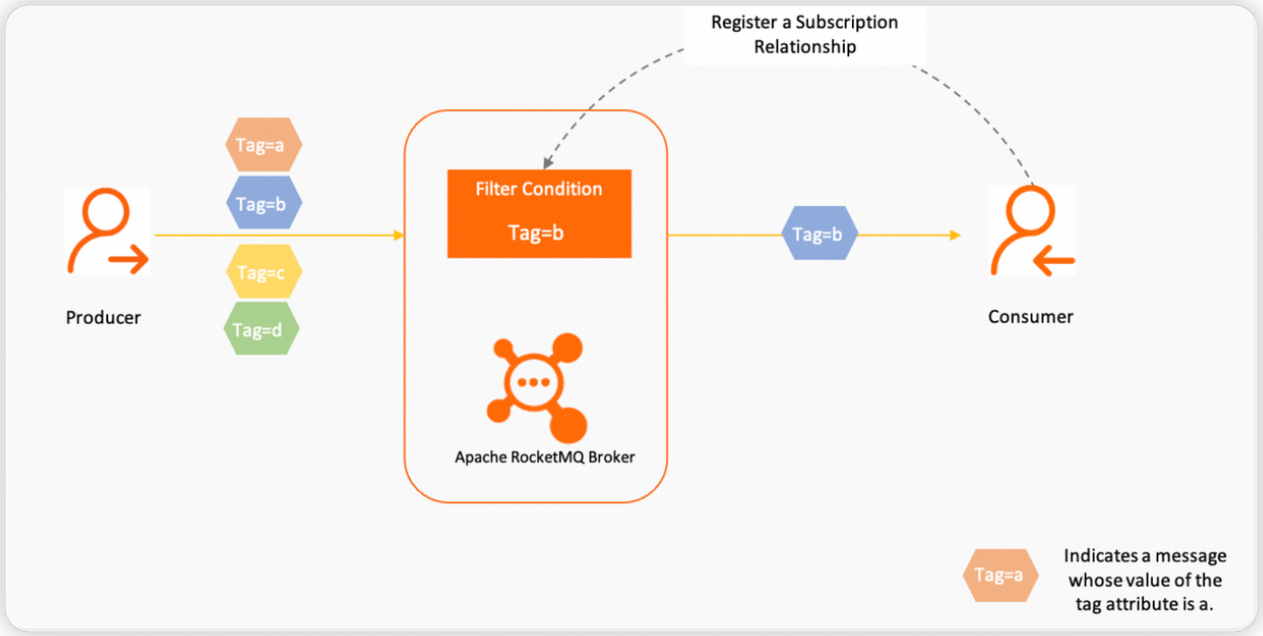
息内容，而无需遍历整个CommitLog文件。ConsumeQueue的存在大大加速了消息的检索过程，尤其是对于海量消息的场景。

c. **IndexFile**

用于提供基于消息Key的快速查询能力，进一步丰富了其存储模型的功能。

- **特色功能**：

a. **消息过滤**：MetaQ支持消息过滤功能，允许消费者基于消息属性或内容进行灵活的过滤操作，而无需在生产者端进行预处理。这一特性极大地方便了消费者侧的逻辑处理，使得消费者能够只接收并处理与其相关或符合特定条件的消息，从而降低了不必要的消息处理开销，提高了系统效率。



接入MetaQ，业务消息的拆分可以基于topic进行，也可以基于消息tag和属性进行。

遵循三个基本原则：

i. 消息类型是否一致：不同类型的消息，如普通消息和顺序消息需要用不同的主题进行拆分，无法通过Tag标签进行分类

ii. 业务域是否相同：不同业务域和部门的消息应该拆分为不同的主题。例如物流消息和支付消息应该使用两个不同的主题；同样是一个主题内的物流消息，普通物流消息和加急物流消息则可以通过不同的tag进行区分

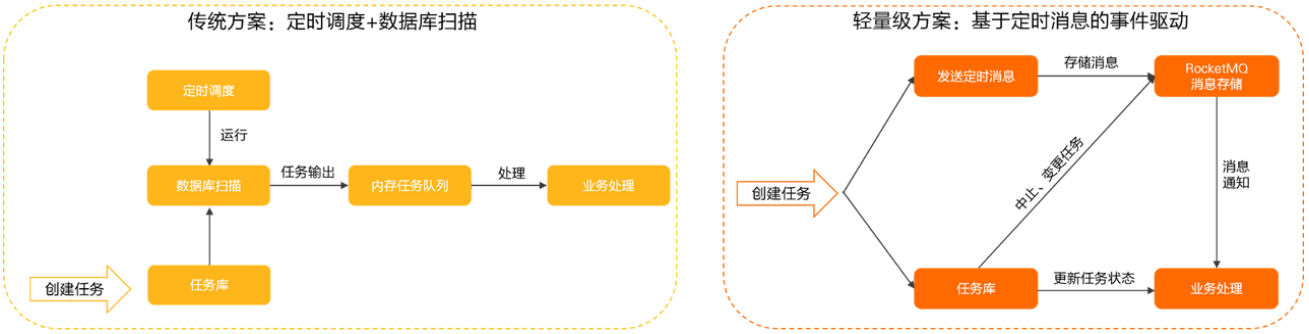
iii. 消息量级和重要性是否一致：如果消息的量级规模存在巨大差异，或者说消息的链路重要程度存在差异，则应该使用不同的主题进行隔离拆分

b. **延时消息：**延时消息功能允许生产者在发送消息时指定一个延迟时间，消息不会立即投递给消费者，而是在指定时间后才被激活并进行分发。这一特性适用于诸如定时任务调度、订单超时未支付提醒等场景，为业务提供了灵活的时间驱动消息处理机制。

以电商交易场景为例，订单下单后暂未支付，此时不可以直接关闭订单，而是需要等待一段时间后才能关闭订单。使用MetaQ定时消息可以实现超时任务的检查触发。

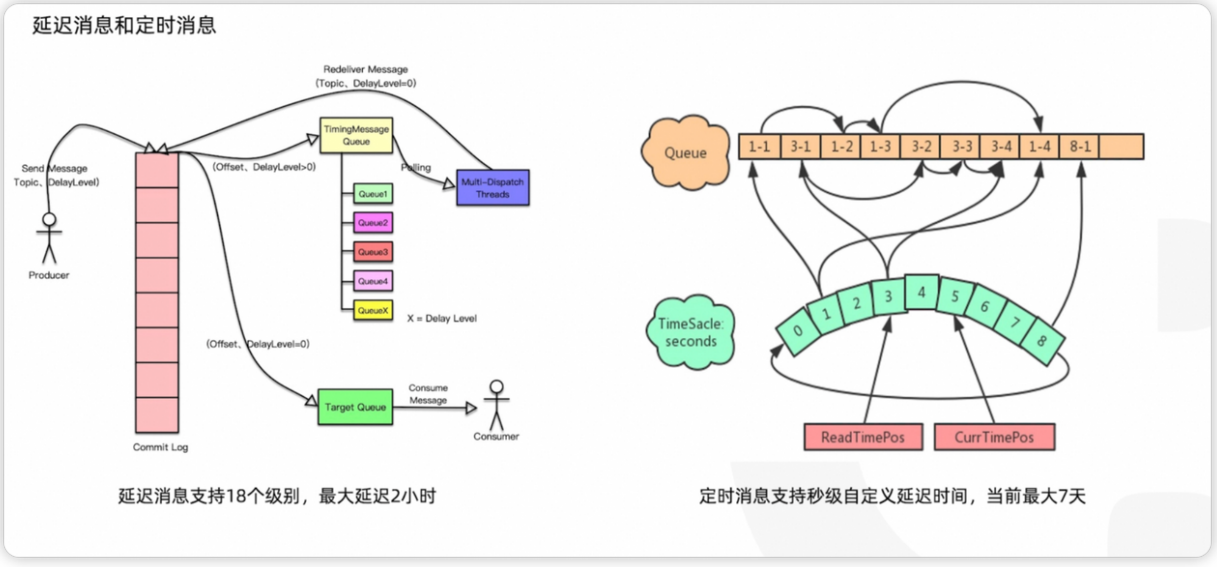
基于定时消息的超时任务处理具备如下优势：

- 精度高、开发门槛低：基于消息通知方式不存在定时阶梯间隔。可以轻松实现任意精度事件触发，无需业务去重。
- 高性能可扩展：传统的数据库扫描方式较为复杂，需要频繁调用接口扫描，容易产生性能瓶颈。MetaQ的定时消息具有高并发和水平扩展的能力。



延时消息与传统定时方案对比

延时消息是如何实现的？MetaQ在内部维护了18个不同级别的延时队列，延时消息会根据设置的延时级别被放入1到18个队列中，然后根据不同的时间间隔定时轮询这18个队列，将当前已到期的消息从延迟队列中拉出，写入commitlog中构建commserqueue供消费者消费。



延时消息实现原理

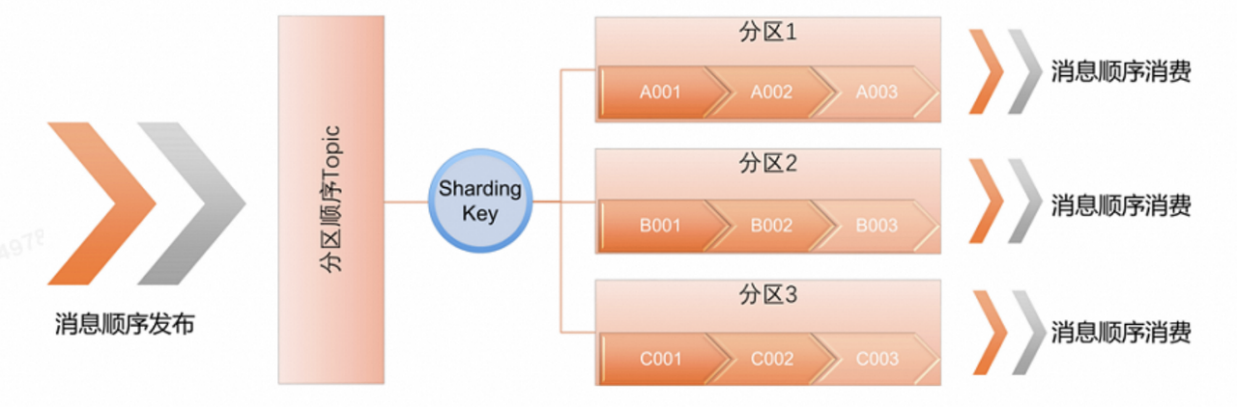
c. **顺序消息：**为了满足某些业务场景下严格的顺序要求，MetaQ支持顺序消息功能。生产者按照一定的顺序发送消息，MetaQ确保这些消息按照发送顺序被同一个消费者消费，或者在分区级别上保持消息的顺序性。这对于需要按序处理的业务流程（如账务系统中的交易记录处理）至关重要，保证了业务逻辑的正确执行。

全局顺序：对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费。

全局顺序消息

全局顺序消息适用于性能要求不高，所有的消息严格按照 FIFO 原则进行消息发布和消费的场景。

分区顺序：对于指定的一个 Topic，所有消息根据 sharding key 进行区块分区。同一个分区内的消息按照严格的 FIFO 顺序进行发布和消费。Sharding key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。



分区顺序消息

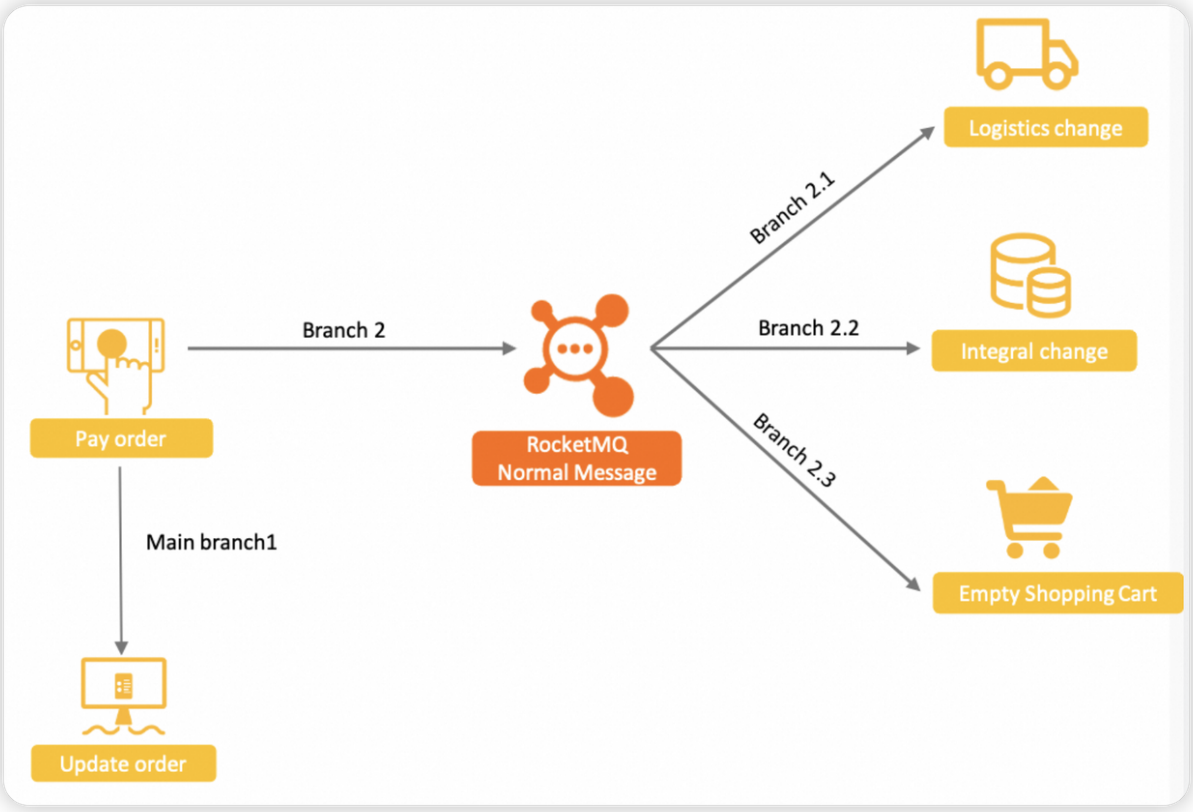
分区顺序消息适用于性能要求高，以 sharding key 作为分区字段，在同一个区块中严格的按照 FIFO 原则进行消息发布和消费的场景。举例说明：

【例一】用户注册需要发送发验证码，以用户 ID 作为 sharding key， 那么同一个用户发送的消息都会按照先后顺序来发布和订阅。

【例二】电商的订单创建，以订单 ID 作为 sharding key，那么同一个订单相关的创建订单消息、订单支付消息、订单退款消息、订单物流消息都会按照先后顺序来发布和订阅。

集团内部电商系统均使用此种分区顺序消息，既保证业务的顺序，同时又能保证业务的高性能。

d. **事务消息：**事务消息机制确保了分布式事务的一致性。它允许生产者在执行业务操作前后分别提交半事务消息的“预备”和“提交/回滚”状态，只有当事务操作成功完成时，消息才会最终被投递到消费者，否则将进行回滚处理。这一特性对于需要跨服务边界的事务操作，如库存扣减与订单创建的原子性执行，提供了强大的支持，确保了数据的一致性和完整性。

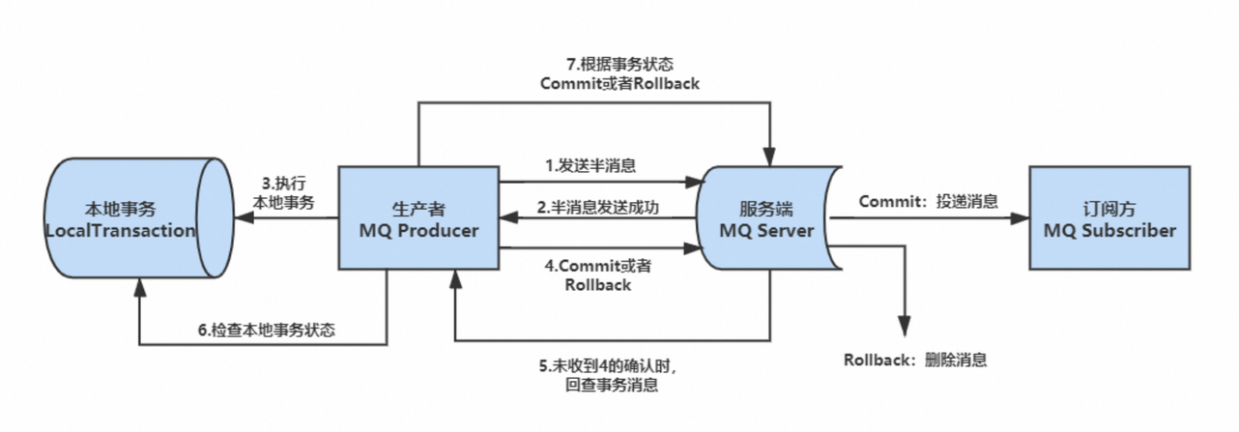


为什么需要事务消息？以电商交易场景为例，用户支付订单这一核心操作的同时会涉及到下游物流发货、积分变更、购物车状态清空等多个子系统的变更。当前业务的处理分支包括：

- 主分支订单系统状态更新：由未支付变更为支付成功。
- 物流系统状态新增：新增待发货物流记录，创建订单物流记录。
- 积分系统状态变更：变更用户积分，更新用户积分表。
- 购物车系统状态变更：清空购物车，更新用户购物车记录。

如果基于普通消息实现，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。



事务消息执行流程

事务消息执行流程：

1. 生产者将消息发送至MetaQ服务端。
2. MetaQ服务端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），服务端收到确认结果后处理逻辑如下：
   - 二次确认结果为Commit：服务端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：服务端将回滚事务，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，若服务端未收到发送者提交的二次确认结果，或服务端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。
6. 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。
7. 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。

事务消息适用于对数据一致性要求非常高的场景，如金融支付、电商等。

第四部分：MetaQ最佳实践

- **消息幂等处理**

在互联网应用中，尤其在网络不稳定的情况下，消息有可能会出现重复，这个重复简单可以概括为以下三种情况：

1.发送时消息重复（消息 Message ID 不同）

2.消费时消息重复（消息 Message ID 相同）

3.负载均衡重新平衡过程中导致的消息重复（不同机器收到消息 Message ID 相同的消息）

幂等性设计：鼓励业务端实现幂等性处理逻辑，即同一个消息被多次消费时，业务逻辑的处理结果保持一致，不会因重复消费而导致数据异常。基于上述第一种原因，内容相同的消息 Message ID 可能会不同，真正安全的幂等处理，不建议以 Message ID 作为处理依据。最好的方式是以业务唯一标识作为幂等处理的关键依据，而业务的唯一标识可以通过消息 Key 进行设置

普通场景或其他系统设计原因，可以不做幂等；但如下类似场景强烈建议做好幂等

a. 单条消息处理时间较长、下游链路较多

b. 单条消息消耗系统资源较多（CPU、内存）

c. 1和2皆有的情况（大量重复可能导致影响系统）

具体实现幂等消费的方式多种多样，较为直接的一种方法是利用数据库的内在机制，如主键约束或联合唯一索引，甚至借助Redis中的setnx命令等工具，确保同一操作在任何情况下执行结果均保持不变。另一种策略则是借鉴乐观锁的理念，在更新数据时设定特定的前提条件，即只有当某一字段维持某个特定值时，才允许进行相应操作。若发现该字段已被修改，那么当前的操作自然无效，此时便需要重新处理该条消息。总之，我们运用幂等性的智慧，巧妙地化解了消息重复带来的难题，虽然重复现象在网络世界中难以完全规避，但通过巧妙设计与严谨实践，我们得以有效管理和控制这一现象。

- **消息堆积处理：**

消息堆积是指消息生产速率超过了消费速率，导致消息在Broker上累积，无法及时被消费者处理。这种情况可能由多种因素引起：

a. 普通消费模式：

消费者的消费逻辑复杂、资源限制（CPU、内存、网络带宽等）或消费者实例数量不足，都可能导致消息处理速度跟不上生产速度，如果单条消息处理时间过长，即使有足够的消费者实例，也可能出现消息积压。消费者进程异常终止、长时间阻塞或网络不稳定导致断连，都会影响消息的正常消费。

解决方案：

i. 排查有没有机器异常下线或者断连的情况

ii. 优化消费逻辑，降低单条消息的处理时间

iii. 增加消费者实例，提高整体消费能力（需要保证消费者实例小于等于队列数，队列数小于消费者集群机器数量时，会存在部分机器分配不到队列，处于闲置状态，需要联系metaq团队对队列进行扩容）

b. 顺序消费模式：

顺序消费模式，某队列某条消息如果消费失败会不断重试，不会跳过，如果一直失败则导致当前队列消费block（并非bug，是保证顺序的设计），所以会出现当若干条消息处理失败时，有个别队列一直堆积的情况。

解决方案：

i. 设置合理的重试次数，并在最后一次重试前对数据进行落库或记录日志处理，避免消息异常重试阻塞队列，后续再对异常消息进行人工补偿。

- **大消息处理：**

作为一款消息中间件产品，MetaQ一般传输的是都是业务事件数据。单个原子消息事件的数据大小需要严格控制，如果单条消息过大（超过128k）容易造成网络传输层压力，不利于异常重试和流量控制。对于较大的消息，推荐应用对消息进行拆分，这样做的原因如下：

a. MetaQ通信层没有对大的请求做优化，采用的是典型的RPC方式，不适合大的请求传递，可能会导致网络层的Buffer异常
b. MetaQ的服务器存储是一个典型的LRU CACHE系统，过大的消息会占用较多Cache，对于其他应用Cache命中率产生影响
c. MetaQ的磁盘资源通常比较紧张
d. MetaQ暂不解决大消息存储问题

解决方案：

建议生产者在发送消息前对消息体的大小做校验，如果有大消息的情况，对消息进行数据分片。生产环境中如果需要传输超大负载的消息，建议按照固定大小做报文拆分，或者结合文件存储等方法进行传输。

参考链接

Apache RocketMQ 官方文档 https://rocketmq.apache.org/zh/docs/introduction/02concepts

MetaQ Wiki https://aliyuque.antfin.com/mqdevops/metaq_wiki

阿里云 云消息队列 产品介绍 https://help.aliyun.com/zh/apsaramq-for-rocketmq/?spm=a2c4g.11186623.0.0.35a724cc2qsTLb

RocketMQ 架构原理解析 https://www.cnblogs.com/xijiu/p/15565733.html