## 消息系统介绍及选型依据

消息系统目前已经广泛使用于互联网企业，各类业务系统都有它的身影，一方面是其传统的功能特点：系统间调用的异步解耦，减低系统的复杂度、流量的削峰填谷，便于业务弹性伸缩、易于实现最终一致性系统，避免分布式事务对性能的影响、支持P2P(点对点的调用)和pub/sub模式减少RPC的多次调用（广播通知机制）等。另外随着业务的快速增长，企业内部需要大量数据的同步传输，流式计算等应用都需要非常稳定高效的传输通道给予支持，消息系统在其中充当了重要的角色。

目前使用较为广泛的消息中间件（MOM）主要有：rabbitmq、kafka、RocketMQ(MetaQ)、ActiveMQ或ZeorMQ等产品；

1. **系统功能介绍**
   * 消息系统对企业异构系统集成有着重要的使用场景，各系统遵循协商的数据规范进行生成和消费消息，系统间通讯从传统RPC服务接口调用方式转换到数据协议的通讯方式，使得系统耦合度降低，责任边界更明确，同时有利于系统升级发布，可持续集成部署；
   * 消息通知机制使得系统异步处理变得简单，同步调用需要阻塞当前线程等待结果返回，会影响系统的并发能力；而消息通知方式是由上游业务将请求转化为消息的形式发往消息中间件(MOM)即可返回，无需等待后续业务处理，下游获取消息后异步的处理后续逻辑。
   * 业务流量的削峰填谷；传统RPC同步调用(PUSH模型)在应对突增性流量时,一般采用扩容/降级/熔断等机制来保护下游核心服务的正常运行； 而消息系统在调用中间层增加了一道缓冲机制，暴增流量发生时，通过消息积压的方式存储在broker中，下游服务根据自身处理能力，量力而行的拉取消息进行消费，不会给后续服务带来毁灭性冲击；
   * 消息系统易于业务实现最终一致性场景；传统的强一致性的事务，一般通过二阶段或三阶段提交方式来实现，但是性能较差容易造成性能瓶颈；通过消息系统的消息通知机制和补偿手段来达到分布式系统的数据一致，很大程度上改善了系统性能和吞吐能力；
   * 支持P2P和pub/sub模型；点到点的调用类似RPC的行为；另外一种是广播机制，比如缓存通知，需要更新所有本地缓存；RPC的方式实现，每接入一个新业务方需要调一次方法，难以动态接入和灵活性较差；消息系统绝大多数支持广播的方式，极大简化业务逻辑处理和提供灵活接入的能力；
   * 快速数据传输，为在线流式计算提供可靠的数据通道。分布式环境中各类异构数据源分散杂糅，高效快速的归集数据，并为实时数据分析模型高效地传导数据，对现代互联网企业有着至关重要的作用。
2. **核心技术剖析**
3. **消息协议支持**

消息协议分为即时通信协议和延迟通讯协议；

* 即时通信协议常用于RPC通信系统，端到端的进行传输采用push的方式，发送方需立即得到消费方的响应;如RMI，Thrift，Avro等协议；
* 延迟类通讯协议，消息一般经由中间层broker存储后，当条件满足时再分发至消费端；常见通信协议如：xmpp, stomp, mqtt, amqp或 binary私有协议；消息中间件采用的便属于此类延迟性协议；

xmpp, stomp, mqtt, amqp属于标准化通讯协议，被广泛使用在消息中间件设计之中，这类标准协议通常基于消息的状态机模型而设计，消息投递过程中的各种状态都从协议层面进行了精心设计和可靠性处理，有的协议在可靠性方面设计过于复杂，冗余信息也多，交互频率高，性能上会有大量损耗；因此在追求高性能大并发的消息系统中，也常见采用私有协议来进行交互，尽管在扩展性和灵活性上有所下降，但协议设计更紧凑，通信负载更轻量级；比如kafka或rocketmq等消息系统均采用自定义协议来进行设计；

1. **通信系统设计**

消息系统处于星型调用关系的中间节点，起着承上启下的关键作用；如何支持单节点达到高的连接数(cps)和高的吞吐量(qps)对通讯层的设计有很高的要求； 常常从以下角度来进行考虑：

* 建立TCP长链接，减少频繁握手的开销；
* 链路空闲(idle)和检活(heartbeat)处理, 回收空闲和非活的链接资源；
* 多路复用及非阻塞IO方式提高单链接多通道的通讯方式；
* io线程池隔离机制，io线程只负责req/resp的读写，各类业务线程池进行隔离避免重业务逻辑的资源池复用；
* 通讯协议设计，比如选用protocolbuf或者私有协议以降低包体大小；
* 对请求进行合并和批量发送的方式；
* 基于jvm的设计时，减少堆内存回收对网络io的影响，比如采用zero-copy的方式避免堆内存的申请；
* 具体实现上可选择netty框架(java语言)降低开发的复杂度。

1. **存储系统设计**

存储系统需要考虑：

* 消息物理存储格式设计

优秀的存储格式的设计，一方面节省存储资源,比如利用位域存储来实现字节的复用，利用族群分块和相对位移/增量存储降低存储开销；另一方面，存储格式的边界对齐考虑也有利于高效的读写性能；

同时要考虑数据的完整性检查，各类索引的创建以及如何快速检索定位消息；

* 消息存储介质选择
  + 日志文件存储方式

选择日志文件来持久化消息，需要考虑：

1. 如何保证高效的读写性能：比如写时采用AOF方式进行记录；利用os的pagecache来改善磁盘访问性能和安全性；
2. 日志文件的保留策略，磁盘是有限资源如何管理和制定妥善的清除策略对存储系统尤为关键；
3. 异常退出，通过日志的如何快速恢复系统；
4. 队列一般是无限长度的空间，需设计好与有限的存储文件之间的映射；
5. 不同业务队列采用单一的事务日志进行存储，还是不同业务采用物理隔离的日志进行存储；两种存储方式在不同的业务场景下各有优势，在不同的开源产品中也都有采用；
   * 数据库存储的方式

采用数据库存储一般源于多维度查询的需求和长久保存消息方便消息审计的目的，利用数据库自身的索引机制给消息的检索带来便利；但需要解决数据库性能吞吐,数据库具有强事务性特点，系统开销会比较大；

1. **路由策略机制**

消息路由是消息系统的核心内容，包括路由元数据管理和消息数据分发策略的实现；

* 元数据管理：用以描述消息队列的物理存储地址，容量大小，分片情况；
* 消息分发策略：根据消息的路由键和系统的路由元数据，通过不同的路由策略算法计算获取落盘的具体物理地址；发送和消费可以共用相同的路由元数据，采用不同的路由算法机制，以达到各类消费的目的；

具体实现方式：有的路由策略放在server端实现；有的通过客户端来实现；

例如：

rabbitmq的路由逻辑放在服务端来实现；通过声明exchange和queue的绑定关系来确定路由情况，支持比较丰富的路由策略，direct/topic/fanout/header等方式以满足不同的业务需求，可以满足绝大部分的应用场景，不支持用户自定义策略；

kafka的路由逻辑在客户端实现；客户端会定时和特定情形下主动拉取路由元数据，通过客户端的路由算法来确定消息的存储地址并创建tcp链接与存储broker进行通信；路由算法在客户端有缺省实现同时支持用户实现接口的方式进行自定义，提供灵活的配置方案和动态调节能力；

1. **负载均衡策略**

负载均衡对分布式应用的稳定运行有极其重要的意义，如何防范流量不均衡导致集群雪崩效应的发生，又或是负载的不平衡造成系统的瓶颈，不能发挥最大的效能；请求负载能否自动化调节或是人工管理的方式进行灵活的分配调整，对系统健康度有着积极的作用，同样负载均衡对消息系统设计亦是如此。

负载均衡的设计，可以从客户端和服务端两方面来考虑：

如4中所言的路由策略，在一定意义上可以实现负载的平衡调节，不同的路由策略可以让流量进入到不同的集群节点进行处理；客户端通过调节路由算法来达到系统负载平衡；

随着集群不断的扩容，资源分布不均衡性逐步显现，资源分布如何动态进行迁移显的尤为关键, 服务端提供出灵活的监控和管理策略来调节节点资源分配就很有必要；比如多副本资源如何协调好leader节点在集群中的分配，对系统吞吐能力有很大的影响。

负载均衡策略在各个子系统中都需要进行细致设计和考虑，以充分发挥单节点和集群的整体性能。

1. **HA高可用设计**

系统高可用从技术和业务层面都需要进行考量；

比如：日志副本的复制/同步管理，采用怎样的同步机制能保证节点失效时能快速切换到新的副本节点继续提供服务；同时保证数据在多副本之间的一致性，以保障消息不丢；这里涉及到两个方面的问题：

1. Leader节点的选举机制；

2. 副本的同步机制，如何保证副本的一致性；

针对以上问题，目前有多种一致性协议来处理，如paxos,raft,pacificA以及私有同步协议来解决副本的一致性问题；协议目的是要解决以上2个重要问题，实现时有很多安全机制需要考虑去解决各类异常情形，比如网络分区，出现慢设备，网络抖动，反复选举，日志回滚安全机制等等问题；

同时业务层面采用ack机制去保证业务正常被处理；发送时如何保证消息安全被传输和存储；消费时如何保证业务正确被执行，失败时如何进行重试，并最终提供安全的ack方式；这些都需要在设计上提供友好的处理方式。

1. **流量控制设计**

消息系统承载着众多的业务数据，各业务重要程度不同，业务流量各异，遇到突增流量，对集群造成风险时；集群需在流量管理上进行流控处理以防止服务宕机，常采用阈值判断的方式来禁止请求进入；

比如：在网络io层或者业务reqest分发层来进行控制，采用blockingqueue的rejected策略，当内部业务线程处理延迟时，请求队列已经塞满的情况下，采取拒绝外部新请求来保证核心逻辑处理；

另外针对不同链接做细粒度的流量控制，对不同的业务流量采取有区别的控制方式，例如kafka根据clientid来区分具体客户端实例进行流量的细粒度控制；

除了server端的流量控制之外，也可在客户端进行流量监控和并发流量控制；客户端的流量在大促期间很容易出现流量尖刺，控制这些流量峰值对系统稳定性非常有价值，如何消除尖峰平稳流量，常常通过拥塞窗口的方式来进行控制，比如利用信号量来控制并发线程的网络io等等方式。

1. **安全认证和传输安全设计**

作为企业级消息总线，安全接入和信息传输的安全性至关重要；

接入认证对管理各类应用接入意义匪浅，例如通过Oauth认证来实现平台级管理客户端实例的功能；

信息的传输安全和系统性能常常不能兼得，信息加/解密操作会影响系统的吞吐，因此设计时根据具体场景来考虑；

1. **消息索引机制**

在提供平台级消息服务时，业务方常常需要对发送或者消费完的消息进行检索服务，进行快速定位、排查问题或是后期补偿措施；

消息队列通常数据量比较大，同时大部分场景使用完消息即可丢弃等特点，如何提供高效的创建索引机制、快速查询方案，以及索引的删除机制都有很大的挑战；

对于吞吐不大的业务需求，底层通过数据库进行存储，依赖数据进行索引创建来支持多维度的查询需求，是非常适合的场景，数据库的索引建立在B+tree结构上，查询性能logm(N)，同时btree是动态平衡树，记录的插入都会造成tree的调整都会带来较大的系统开销，性能欠佳；

另外一种如rocketmq的索引方案通过预分配hash文件的方式来创建文件hash表，利用文件系统的pagecache来达到内存级别的O(1)访问速度；对于hash键碰撞问题未能提供re-hash的处理机制，预分配时是已经确定了最大hash槽位，检索碰撞键值时退化到链表的查找性能，需业务上精心设计查询键值；

也可接入外部索引方案如ES系统，通过消费的方式对所有集群的消息进行索引创建，可以只考虑索引的存储，避免索引过大，带来系统的开销和检索性能的下降。

1. **消息的幂等处理**

幂等处理是业务层面考虑的重要议题，如何在消息技术层面降低重复消息的影响，通常需要从上下游综合考虑；

1. 发送端发出的消息如何保证在broker端进行唯一存储，这是上游发送需要考虑的幂等问题；由于网络不稳定或丢包造成发送ack包丢失，业务层面采取重试后就易造成消息的重复存储；在server端进行消息去重，要求server端能缓存和识别消息的标识码，进而实施去重操作；这种标识需要在集群层面全局唯一，最有效的处理方式，是集群server能提供生成唯一标示码的服务，发送端发送之前需要首先从server端获取标识消息的标识码，进而发送，server端通过预申请的标识来过滤重复消息；
2. 消费端去重处理，消息系统在投递消息时一般支持at-least-once的投递保证；当消费客户端发生rebalance或是集群元数据异动后都有可能造成消息重新投递至客户端；客户端如何识别重复消息并进行去重处理，一般做法采用缓存某个时间窗口的消息标识，继而来过滤短时间内的大量重复情形；这依赖本地缓存或是中心化缓存的大小，本地缓存问题在于发送客户端rebalance之后，缓存失效问题，中心化缓存会引入第三方存储的复杂度；不引入第三方缓存服务亦可基于消息系统本身来实现中心化缓存的目的；以上考虑只能去除短时期内的重复问题，因为内存大小的限制；要彻底解决幂等性问题还需要业务上配合进行更严格的幂等判断，比如实现业务上的状态机，避免相同状态反复处理问题；
3. **事务性消息支持**

事务性消息有两个层面的含义：

1. 业务操作和消息发送的同步成功或同时失败；例如：数据库的修改事务完成后，消息才能同步投递消费端；这种情况为什么采用事务消息来处理， 事务消息在实现时一般采用2阶段commit方式来完成；pre-commit阶段消息已经发送至server端并持久化存储，该阶段有较重的业务逻辑，第二阶段进行commit或rollback来判定消息是否对外发布，处理逻辑相对轻量级，出问题的概率小很多；如果不采用事务消息的方式发送，有可能数据库修改完成了，但是发送消息逻辑一直处于失败之中，造成业务逻辑处理的不一致；而采用事务消息发送时，消息先进行了存储，之后进行数据库的操作，根据数据库操作的成败来决定消息的提交或回滚；
2. 事务性消息需要保证在单事务内所有消息的原子性和顺序性；意味着事务开始后，所有该事务内发送的消息，在提交完成后，消息同时对外进行发布及消费者能够消费该批消息，如果回滚该事务，则此类消息对外不可见；并且事务内的消息顺序需要保证；

事务性消息设计需要考虑，消息的二阶段提交过程，在此过程中需要考虑第二阶段过程出现异常后，比如commit/rollback丢包或这阶段发送实例crash，本地事务并未完成时，server端如何进行远程回查并完结该事务；另外该事务内的批量消息的顺序如何保证。

1. **顺序性消息支持**

顺序消息一般通过队列来实现，依赖队列先进先出的机制来保证消息投递的顺序性；

单队列的吞吐有限，如何提升性能，自然想到的一个方式增加队列数，支持更多并发线程来处理多个队列消息；

队列之间并不能保证顺序，消息的全局顺序性是无法保证的；通常结合业务特征，对顺序消息进行细分，将顺序消息负载到不同的队列上支持更好的并发度；

多个队列提供顺序服务时，需要考虑消费端的rebalance机制，例如缺省的负载分配方式，会造成rebalance阶段出现短暂的消息混乱；因此需要精心设计该部分逻辑。

1. **消息优先级**

消息队列实现排序一般通过额外索引来实现，基于数据库存储支持排序实现相对简单，数据库自身的索引机制就能支持，但随着吞吐量的增高和消息量的增加，动态索引创建会给系统带了很大的负荷；

基于文件存储的消息队列做优先级排序实现更为困难，即便使用额外索引文件做排序方案，随着消息量的增加，索引文件排序必然导致文件外排序，索引项的不断调整会引起频繁的io，性能会产生瓶颈；

另外的方式，对消息设定有限的优先级别比如最大128个优先级；通过桶排序的方式，将不同优先等级的消息存储到不同级别的桶中，投递消息时按照桶的优先级进行顺序投递；基于这种思想，消息队列并非需要支持优先级排序功能，也可从业务使用的场景上出发，通过申请多个消息队列，队列有优先级顺序，在发送消费时做好优先级别分配也能达到同样的效果。

1. **消息回溯**

消息回溯在各项业务中都有广泛的应用，比如索引恢复/本地缓存重建，有些业务补偿方案也可采用回溯的方式来实现；

消息回溯需要解决的是查找回溯的基点，通常按时间维度和绝对位点来进行消息起点的定位；消息队列如果实现了多维度查询亦可按照查询维度定位消息的位点后再重置回溯起点；

在PUSH消费模式下，除更新server端存储的消费进度外，同时也需通知所有对应的消费端重置pull消息的位置信息，重新进行消息拉取动作；

1. **消息广播**

大部分消息系统支持pub/sub的订阅方式，不同的消费group能全量消费订阅队列的所有消息，组内实例间采用并行互斥的方式消费消息；组间的广播常常会造成业务部署时的苦恼，每个环境下需要申请不同的group名来进行配置，遗漏或错误配置都会引起生产事故；在这种场景下，实现组内的广播机制，对业务会有很大的帮助，在设计客户端rebalance机制和位点持久化方面需要精心考虑。

1. **常见消息系统对比**

本文主要从几种常见的消息系统来进行比较，唯品会目前主要使用两种消息系统：kafka和rabbitmq，阿里开源消息系统RocketMQ，本文主要分析这三个消息系统的不同特性：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 消息系统  特征 | **特性类别** | **Kafka** | **RabbitMQ** | **RocketMQ** | **评估标准** | **指标 重要性** | **kafka score** | **rabbitmq score** | **rocketmq score** |
| **集群部署** | **可用性** | 1. 支持大规模集群部署,分区容错性非常好; 2. kafka元数据通过zookeeper来进行持久化,保证访问的一致性； 3. 支持跨机架均衡分布策略; | 1. 对网络延迟和稳定性较为敏感，相比之下**容易发生集群脑裂**; 2. 跨机房大规模部署时依赖第三方插件如federation等,分区容错性较差; | 1. 不支持大规模的部署,对于脑裂情况的处理相对较弱; 2. 元数据管理通过namenode节点存储, 可能出现元数据不一致的情况; namenode的可用性比较高，不受网络分区等情况影响； | 1. 支持的集群规模大小; 2. 元数据管理的一致性问题; 3. 集群可用性对网络延迟的敏感性问题; 4. 资源负载自动调节问题; | 高 | 优秀 | 较弱 | 中等 |
| **数据一致性保证** | **可用性** | 1.支持pacificA协议保证副本复制; 通过ISR来保证副本之间的同步性； 2.支持强一致性语义； 3.leader节点由集群的controller选举,支持动态调节leader分布情况以均衡负载； 4. Leader/follower是partition级别，主备切换只影响该分区; 5.读/写发生在leader节点上; | 1. 基于rabbitmq镜像队列同步框架 的方式; 2. 支持强一致性语义; 3. Leader/follower是queue级别，主备切换只影响该队列; 4. 读/写发生在leader节点; | 1. 通过私有协议来进行主备同步； 2. 不支持在线的主备切换，通过配置方式决定主备实例； 3. 主备角色是实例级别的，切换时会引起节点上所有的队列不可用； 4. 写事件发生在leader节点，读事件可以在follower节点执行； | 1.是否支持强一致性语义; 2.同时是否支持弱一致的模式; 3.副本同步的延迟考量; 4.failover机制，恢复能力; 5.主备切换的可用性; 6.一致性协议的复杂性; 7. 同步的效率问题; | 高 | 优秀 | 良好 | 较弱 |
| **路由策略支持** | **功能性** | 1. 支持topic路由方式； 2. Server端保存和维护topic的分区partition的存储信息； 3. 客户端实现路由算法(默认采用hash算法来选择存储partition地址），客户端定时拉取server路由表通过路由算法来计算存储地址并发送请求； | 1. 支持exchange的多种路由策略，如direct/topic/fanout/head等； 2. 元数据存储在server端，路由策略在server端执行，通过exchange进行分发； 3. 客户端在消息中设置路由键routekey后，发送消息到任意server上对应的exchange即完成发送。 | 1. 同kafka 的topic路由方式 | 1. 路由策略满足各类业务需求的情况; 2. 路由策略执行位置，服务器或客户端; 3. 路由策略带来的负荷问题; 4. 路由策略的效率问题; 5. 是否支持用户自定义路由规则; 6. 路由机制对系统资源的开销影响; | 高 | 中等 | 良好 | 中等 |
| **日志保留策略** | **功能性** | 1. 支持按时间和文件大小及指定offset的清除日志的方式，根据保留时间阈值扫描日志的最后修改时间进行清除动作，批量清理处理效率极高；不考虑消费方积压情况，设置不合理容易引起客户端重置进度和丢失数据； 2. 支持compact压缩的日志清除策略，按K-V来存储最新版本的消息，永久保留内存中的全量数据 | 1. RabbitMQ清除策略有些类似jvm垃圾回收，采用标记清除方式定时清除标记为已消费的消息并进行清理；清除逻辑比较复杂。 | 1. 同kafka按时间清除的策略 | 1. 日志文件保留策略多样性; 2. 日志保留策略对系统资源的的保护情况; 3. 日志保留对业务消息回溯的支持情况; 4. compact log保留策略的支持; 5. 清除日志的性能问题; 6. 清除日志对运行系统性能影响问题; | 中 | 优秀 | 中等 | 良好 |
| **消息索引机制** | **功能性** | 1. 支持按offset的查询方式； 2. 支持按时间timestamp的查询方式； 3. 每个partition目录会产生2个独立索引文件用于支持按offset和按时间的检索方式； | 1. 内部不支持消息的检索 | 1．支持按offset的查询方式； 2．支持按时间timestamp的查询方式； 3．支持按业务设置的key进行检索消息方式； 4．系统按业务设置的key创建hash索引文件，检索时通过hash方式检索消息； | 1. 支持的索引方式; 2. 索引查询的性能; 3. 创建索引成本问题,效率/size等; 4. 多维度索引需求的支持问题; 5. 查询接口使用的难易程度; | 中 | 中等 | 差 | 良好 |
| **延时队列支持** | **功能性** | 1．kafka服务并未从系统内部支持延时队列的实现； 2．每个消息都携带时间属性，可按发送时间或存储时间进行选择； 3．通过外部服务的方式，结合队列延时要求，实现延时队列的支持，存在一定的开发工作量 | 1. RabbitMQ支持队列设置消息expired时间，结合DLX的路由策略来实现延时队列的投递； 2. 客户端通过声明相关exchange和queue及相应绑定关系便能实现延时队列的功能； | 1.内部支持18个等级的延时队列； 2.用户发送延时消息时，指定消息的延时等级，server会按延时等级先存储至不同的内部延时队列； 3.在延时到期时，从内部队列发布到实际用户队列中； | 1. 延时队列支持情况; 2. 延时的时间粒度问题; 3. 延时的精度问题; 4. 延时对消息堵塞的影响问题; 5. 用户使用的难易程度; | 高 | 较弱 | 良好 | 良好 |
| **队列的幂等处理** | **功能性** | 支持在producer生命周期内发送消息的幂等性；server端会按事务id进行去重处理。 消费端不支持幂等处理 | 不支持发送消息的幂等处理 不支持消费端的幂等 | 不支持发送消息的幂等 不支持消费端幂等 | 1.发送端是否支持幂等; 2.消费端是否支持幂等; 3.幂等实现逻辑的复杂性; 4.幂等处理对性能的影响； 5.幂等对空间复杂度的耗用; 6.幂等客户端编码的复杂度问题; | 中 | 良好 | 较弱 | 较弱 |
| **系统吞吐/性能** | **性能** | 1 .kafka单机的吞吐性能非常好，在批量压缩的模式下，可以跑满系统的资源，比如网络或磁盘的io, 可以达到近10w+的qps； 2.网络的多路复用，异步分堵塞，以及server端的zero-copy等机制让其性能表现非常优越； 3.kafka 分区性非常好，很容易进行水平扩容，并保持一定的线性吞吐增长. | 1. rabbitmq性能方面低于kafka，在高可用的方案下，单queue的吞吐大概在8k左右； 水平扩容方面比较麻烦，单queue的扩容几乎不可能，需要进一步的封装逻辑queue来达到分片的目的，以支持多物理queue的持久化. | 1. 和kafka类似的设计方案，但是缺乏批量发送等功能，在性能方面在特殊场景下会低于kafka；但在单机大量partition的情况下，性能表现比较平稳，优于kafka; 2. 水平扩容方式比较简单，非常容易进行线性扩容。 | 1. 发送不同size消息的QPS情况; 2. 高并发情况下性能瓶颈问题; 3. 系统自身保护问题; 4. 系统负载分配情况; 5. 系统容量的水平扩展问题; | 高 | 优秀 | 中等 | 良好 |
| **租户管理** | **可靠性** | 1．支持用户实例级别的访问权限管理； 2．支持用户实际级别设置流量控制； | 1．支持用户级别的访问权限管理； 2．流量控制只能在集群层面设置对每个链接按类似信号量的方式(credit)进行限制流量 3．另外按照内存容量或磁盘容量设定阈值来进行限流； 4．支持vhost的资源隔离级别，在能在资源的命名空间上做隔离，并未在物理资源分配上做到隔离 | 1．系统不支持用户权限和限流处理 2．可以接入第三方系统来实现 | 1. 用户管理，认证和授权; 2. 实例容量管理，限流/扩容/缩容管理; 3. 资源监控和告警; 4. 用户资源的隔离机制; 5. 用户信息传输加密处理; | 中 | 中等 | 良好 | 较弱 |
| **集群扩容能力** | **可用性** | 1. Kafka水平扩容能力较为突出，基本能够到达线性容量提升的水平; 在linkedin实践中也有部署超过千台设备的kafka集群； 2. Topic的设计极易于业务容量扩容，topic是逻辑层概念，物理层面上关联了一系列分散在不同设备上的partition资源信息，而partition是实际数据传输通道；因此当扩容时只需增加partition数目，客户端可无感知的将数据分散到新的分区partition上进行服务，增加了带宽。 | 1．RabbitMQ对网络延迟和稳定性较为敏感，易发生脑裂情况，尽管通过federation或shovel插件能降低脑裂发生的情况；但在实际生产上，并未见rabbitmq部署在大规模集群中使用； 2．RabbitMQ针对单业务容量扩容，从设计上是有一定欠缺，客户实例容易扩容，依靠rabbitmq内部负载均衡机制便能达到；但是受限于单queue通道的容量大小，该queue在原生系统中很难简单的扩展上去； | 1．RocketMQ的设计理念和kafka类似，也采取了topic的路由策略，通过partition分区极易扩展容量； 2．在实际生产上，并未见大规模部署集群的案例；通过有限的扩容实验来看，其水平扩容也能达到线性增长的程度。 | 1. 发送者动态扩缩容能力; 2. 消费者动态扩缩容能力; 3. 服务端broker资源的水平扩缩容能力; 4. 扩缩容后流量负载的均衡分配能力; 5. 扩缩容对业务的影响情况; | 高 | 优秀 | 较弱 | 良好 |
| **版本兼容性** | **可用性** | 1. Kafka设计上充分考虑了生产环境中不同版本的兼容问题，从协议层面进行很好的设计以支持跨版本的访问情况； 2. 集群升级时也提供了不同版本升级方案，以无损无感知的方式继续提供消息服务； | 1．RabbitMQ采用的标准协议，协议保持不变的情况下server端和客户端并没有版本的耦合，因此极少考虑兼容性的问题； 2．Server端升级也提供比较完善的升级方案，以支持版本的不断迭代。 | 1． RocketMQ在兼容性方面相对薄弱，底层交互的协议变化会对应用方造成比较麻烦的处理； 2． 基于mqtt协议的客户端相对影响比较小 | 1. 不同版本客户端或服务端的兼容性问题; 2. 平滑升级版本的情况; 3. 升级客户端对业务代码的影响情况; 4. 数据协议的兼容性问题; 5. 不同版本对系统性能的影响情况; | 中 | 良好 | 优秀 | 中等 |
| **监控运维** | **可用性** | 1. Kafka监控和运维工具较多，每个工具专注方向不同如kafka manager， kafka web console，kafka offset monito，Cruise Controller等； 2. Kafka服务自身提供了metrics的各类指标，方便用户通过jmx的方式去拉取指标去做平台级的监控服务；在运维管理方面也可通过admin-api进行集群的运维和管理，为达到更高层次的监控和治理能力可以集成开源产品的服务功能； | 1． RabbitMQ主要基于rabbitmq-management插件进行集群的监控和管理； 2． 该插件提供丰富的rest-api，方便集成至公司的消息平台来实现各项管理能力，提高公司消息平台的服务能力； | 1．RocketMQ监控和运维平台主要是rocketmq-console，通过rocketmq的admin-tools工具可以获取rocketmq内部的各类状态，同时提供管理的api可方便的对集群和客户端进行监控和运维管理 | 1. 监控运维工具是否齐备; 2. 是否支持动态调整参数; 3. 资源的动态创建，扩缩容和迁移; 4. 客户端/服务端各类系统指标的获取和展示; 5. 操作步骤审计 | 中 | 良好 | 良好 | 中等 |
| **产品社区活跃度** | **可用性** | 1.最近Kafka版本更新很快，提交频率非常高，不断的推出新的特性，生态环境也越来越完善，流计算，数据同步等都有很好的支持； 2.Kafka后期的改进规划(KIP事项)非常清晰，针对可靠性，可用性和一致性提高方面都有很好的讨论和时间规划； 3.Kafka各类文档资源丰富，任何问题很容易在wik或各论坛邮件都能得到快速回复。 4.围绕kafka的各类开源项目众多，大家容易参考和借鉴相关经验。 5. 语言支持比较广泛，各类跨语言的应用产品容易搭建； 6. 针对大数据的应用，数据同步，比如同步数据库或缓存等业务的场景，在kafka生态立很容易找到同类产品。 | 1．Rabbitmq官方文档比较丰富和完善，因其erlang语言开发，对大多数开发者而言不太容易深入学习，出现线上bug自己修复的可能性很低； 2．论坛社区针对各类问题的回答相对欠缺，基本依赖官方解答或者bug修复； 3．生态产品基本是官方出品，第三方提供的相关产品比较少见； | 1．RocketMQ今年刚升级为apache顶级项目，但是其文档和问答内容还是十分欠缺；出现问题时比较难找到解决方案； 2．相比其他产品属于比较新的系统，因为其对业务支持相对完善，越来越多的业界开始尝试使用；和阿里内部使用版本的还是有一定差距也是外界诟病的地方； 3．围绕rocketmq的生态产品目前还不太多。 | 1. 介绍/设计文档资料丰富; 2. wiki和论坛活跃; 3. 代码提交频繁，版本更新快速; 4. 实际生产，用户数量参考; 5. 问题解答快速，网络容易查找解决方案: | 中 | 优秀 | 良好 | 中等 |
| **客户端对比** |  |  |  |  |  |  |  |  |  |
| **顺序性消息发送** | **功能性** | 支持顺序性消息发送，可自定义路由策略来指定partition进行发送； | 不支持顺序消息 | 同kafka | 1. 是否支持; 2. 顺序支持的限制条件; 3. 顺序消息的吞吐性能; | 中 | 良好 | 较弱 | 良好 |
| **延时消息发送** | **功能性** | 1.原生kafka不支持延时消息； | 1.基于rabbitmq队列消息过期和DLX来实现延时队列的支持 | 原生客户端api和server端服务支持延时消息的投递 | 1.是否支持延时队列发送接口; 2.延时队列发送的确认机制; 3. 是否需要客户端逻辑来实现; | 中 | 较弱 | 良好 | 中等 |
| **资源名隔离机制** | **可用性** | 1. Kafka原生不支持资源的命名空间隔离； 2.可参照rocketmq机制来实现命名空间的资源隔离 | 支持vhost的资源命名隔离;相同资源名在不同的命名空间里属于不同的物理资源；针对不同环境复用或有灰度使用场景的需求极为有用 | Rocketmq支持virtual的命名空间隔离资源功能(类似rabbitmq的vhost)，此功能在客户端进行了封装，通过环境变量作为前缀命名的方式来区分物理资源；同样可以达到rabbitmq对资源隔离的效果 | 1.物理资源名隔离机制; 2. 数据的分流处理逻辑; 3. 灰度下发的支持情况; | 高 | 较弱 | 较弱 | 中等 |
| **消费模式支持** | **功能性** | 1.支持P2P的消费方式； 2.支持组间广播消费，组内实例间互斥消费模式; 3.pull + push 模型 | 同上 | 1. 同kafka消费模式； 2.同时支持组内的广播方式，每个实例亦能消费全量消息； | 1. 消费方式的支持情况; 2. 不同模式下，消费ack的处理方式; 3. 不同模式下的水平扩展方式; | 中 | 良好 | 中等 | 良好 |
| **消费Rebalance机制** | **功能性** | 1. 支持默认的平均分配机制 2. 支持自定义负载均衡机制； 比如：a.保证严格顺序对partition的严格分配策略，消费实例固定在partition上； b.按idc机房的分配策略以支持多机房多活的处理逻辑 | 1. 仅Roundrobin的负载分配机制，缺乏灵活性； 2. 但是通过客户端比较复杂的逻辑，并结合 rabbitmq服务端exchange的路由策略可以定义丰富的负载机制，但是实现逻辑负载。 | 1. 与kafka的rebalance机制类似；  2. 但在网络分区情况下，会造成一定的分配故障，kafka分配策略在这种情况下会比rocketmq更严格。 | 1. 消费rebalance自定义策略的支持; 2. rebalance对消费状态的影响; 3. rebalance对扩缩容的影响; 4. 网络分区对rebalance的影响状况; | 中 | 良好 | 中等 | 良好 |
| **消费ack机制** | **可靠性** | 1.提供手动和自动ack机制 2.自动ack机制情况下，按kafka投递消息的语义，并不能完全可靠的保证消息是否成功消费； | 1.支持手动和自动提交 | 1.定时自动提交方式，能保证可靠的投递和消费，根据消费状态来进行消息的ack机制； | 1. 消费ack的可靠性和安全机制; 2. 消费ack的支持模式; 3. ack的持久化方式; 4. 失败后的恢复机制; | 高 | 中等 | 良好 | 优秀 |
| **消费的重试机制** | **可用性** | 1. Kafka客户端不提供消费重试，业务自己考虑本地重试方式；在故障或异常情况发生时，不能很好地解决消息失败后的处理逻辑； | 1. 原生客户端不支持重试 | 1.客户端支持重试投递的机制，用户可根据时延等级来制定下次消费的时间间隔；  2.重试并不会阻塞原生队列的消费，而是通过内部重试队列来充分解决并发问题。 | 1. 重试队列的实现方式; 2. 重试队列的使用方式: 3. 重试队列的订阅情况； 4. 重试队列对性能的影响; 5. 对并发情况及吞吐的影响; | 高 | 较弱 | 中等 | 良好 |
| **消息回溯** | **功能性** | 1.支持按消息offset进行回溯 2.按消息时间戳进行回溯 3.回溯只在单实例内进行，并不能通知所有实例统一进行； | 不支持 | 1.支持按offset方式回溯  2.支持按时间戳方式回溯  3.支持按业务设置的key进行消息回溯； 4.回溯能通知到所有组内实例同时进行; | 1. 支持的回溯消息的方式; 2. 查找回溯地址的方式; 3. 通知客户端回溯消费的机制; 4. 回溯失败等异常处理机制; | 高 | 良好 | 较弱 | 良好 |
| **消息过滤机制** | **功能性** | 1.不支持server端过滤方式 | 1. 不支持消息server端的过滤方式 | 1. 支持tags属性的过滤方式，性能很高，server端执行过滤; 2. 支持编写java class插件的过滤方式，server端执行过滤; | 1. 支持消息过滤的方式； 2. 是否server端过滤，降低传输流量； 3. 过滤的性能影响； | 中 | 中等 | 中等 | 良好 |
| **单队列并行消费能力** | **可用性** | 1.默认kafka客户端不支持单队列的并行消费，自身的线程分配模型决定； 2.提供缺省的并发能力依赖分区partition数量的增加和消费实例数的扩容； | 1.支持单队列并发能力，所有实例处于平等地位； | 1.支持kafka的默认的并发模型； 2.同时支持单队列的并行消费，降低单队列堵塞情况； | 1. 对单队列并行线程消费的支持; 2. 批量预取数据的能力； 3. 对消息消费完后并行ack的安全处理机制; | 高 | 中等 | 良好 | 良好 |
|  |  |  |  |  |  |  | **63.04** | **52.61** | **59.57** |

目前唯品会消息中心(VMS)使用了RabbitMQ和Kafka两套消息引擎；根据最初建立消息系统时的情况，分别定义了各引擎的适用范围：RabbitMQ适用于电商平台重要业务，对消息可靠性要求比较高的场景；另外，kafka主要适用于可靠性要求不高但流量(QPS)比较大的场景。

实际生产中, RabbitMQ集群曾发生过多次脑裂的情况，造成系统可用性下降，影响到业务的正常运行；并且脑裂发生后运维难度比较大，存在消息丢失的情况，对重要业务的影响颇大；另外若出现broker源码的bug，因erlang语言的因素，常常只有等待官方的bug-fixed，不能及时的解决生产问题；

另外，随着kafka近几年的快速发展，其在设计上做了较多改善，在可靠性方面也做了很多优化，同时加强和新增了对业务处理的诸多Features；从最新发布的kafka版本1.0来看，比如支持了事务消息的发送，消息发送幂等处理，负载均衡leader角色调整，新增offset管理角色(减少对zookeeper依赖),支持按时间回溯消息索引，副本同步协议实现方式优化，用户接口层面都使用非阻塞可中断的机制来实现等，再加上系统能非常好的支持超大流量，以及便捷的水平扩容能力，这些让kafka适用范围越来越广泛，对各类可靠性要求高的业务场景也能较好的支撑；

基于生产中存在的问题和开源产品演进的特征，从目前业务方使用情况来分析，若能统一消息引擎，从系统开发，运营/运维角度考虑无疑会极大降低成本，同时降低了系统的复杂度，增强系统的可用性和可靠性；以下将从唯品会电商业务使用功能进行梳理，统一到kafka引擎能否满足目前所有的需求进行论证。

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | RabbitMQ支持，Kafka原生不支持 | Kafka解决方案 | 备注 |
| 1 | 延时队列的支持 | 1. 发送消息到特定的topic，通过后端服务消费该topic并判断消息的时间戳以及延时等级或间隔并发回到用户队列来实现。 2. 需考虑不同延时等级或gap的消息相互堵塞的问题，可以通过多级topic的方式来避免该情况的发生； | 延时队列是指消息无需立刻进入消费状态，而是等待一定的延时之后再提供给消费端消费。 |
| 2 | 重试队列的支持 | Vms封装的功能，kafka实现可采用相同逻辑 |  |
| 3 | 死信队列 | Vms封装的功能，kafka实现可采用相同逻辑 |  |
| 4 | 路由功能：   1. 同一exchange根据rk绑定路由到不同queue； 2. Topic的模糊匹配的路由方式 | 发向同一个topic资源的消息，不同消费组订阅后可采用客户端过滤机制来实现；订阅关系中新增消息filter的接口，可通过完全匹配的键值或模糊匹配的方式来进行过滤 |  |
| 5 | 单条消息消费的ack机制； | 1. 可通过滑动窗口，min\_offset的提交方式，来保证单条消息的不丢，保证单消息的可靠性。 |  |

综合所述，目前rabbitmq的提供的功能特性方面，通过在kafka-client基础上进行二次开发可以满足现有的业务需求；同时依赖1.0版本的kafka，可以实现更多的业务特性，比如消息的发送幂等，消息的回溯，消息的查询等。

1. **选型参考依据**

如何构建企业级消息总线，是基于现有开源引擎还是自力开发消息系统去满足企业内部需求；从开发周期、技术力量、人力资源以及功能特征上去分析，一般采用已有的开源产品来构建企业的消息总线收益更大；具体选择哪种消息引擎，可以从以下角度进行判断：

选择条件：

* 开源产品的活跃度，学习资源和各类问题解答获取的途径；
* 开源产品开发语言的学习成本；
* 消息引擎支持的功能特性：
  + 数据可靠性，一致性保证；
  + 集群水平扩容能力，分区有效性；
  + 集群兼容性和平滑升级的能力；
  + 消息路由策略的灵活性；
  + 消息日志的持久化方案；
  + 消息支持高级特性方面，顺序消息，延迟消息，事务消息，幂等性处理，索引机制等；
  + 运维治理能力，接入认证，流量控制，日志清除方案；
  + 压力性能测试结果。
* 客户端功能支持：
  + 客户端发送消费可靠性机制，比如并发ack机制，重试消费等能力；
  + 客户端实例水平扩展能力及负载均衡机制
  + 客户端多线程并发能力；
  + 实例级别治理管控能力；
  + Push和pull客户端的支持能力；
  + 消息高级特征的支持情况；
  + 灰度及资源隔离情况支持；
  + 消息回溯或补偿方式；

从kafka,rabbitmq和rocketmq的各类特性比较来看，kafka在以上条件对比中，都有比较好的支持和功能实现，有些功能特性原生kafka还未能支持，但是在其之上进行功能二次开发，能够很好的满足现有的业务需求；同时从性能测试角度考量，kafka批量压缩异步等特性使得kafka能支持单机10w之上的吞吐，优于其他消息引擎；从管理运维角度，kafka大规模部署场景下，运维治理能力也十分优秀；

Kafka引擎之上可实现的功能特性之总结：

|  |  |  |  |
| --- | --- | --- | --- |
| 基于kafka引擎功能  特征支持情况 | kafka\_V0.8.x | kafka\_v1.0.0 | 原生kafka基础上  新增和优化  (默认支持原生态) |
| 顺序消息发送 | 支持 | 支持 | 同上 |
| 批量压缩模式 | 支持 | 支持 | 同上 |
| Ack机制 | -1,0,1  -1 代表副本全同步,  0 代表请求送达，  1 代表leader持久化 | 同上 | 同上 |
| 异步发送回调机制 | 不支持 | 支持 | 同v1.0.0 |
| 发送限流控制 | 不支持 | 支持按客户端实例方式 | 同v1.0.0 |
| 发送路由策略 | 缺省支持:  hash(key)取模 | 缺省支持:  hash和roundrobin方式 | 丰富的路由策略：  读配置方式路由，按机房路由，按权重方式等 |
| 支持扩展header信息 | 不支持 | 支持 | 同v1.0.0 |
| 发送幂等性 | 不支持 | 支持 | 同v1.0.0 |
| 发送事务性消息 | 不支持 | 支持 | 同v1.0.0 |
| 延迟消息 | 不支持 | 不支持 | 通过内部队列的转发机制实现延时功能，需开发； |
| 重试队列 | 不支持 | 不支持 | 基于java客户端可实现重试队列；  可提升消费的并发度和降低消费阻塞情况； |
| 死信队列 | 不支持 | 不支持 | 开发实现；方便用户审计以及消息的补偿 |
| 支持push和pull模式 | 两种模式均支持 | 支持pull模式 | 基于v1.0.0版本增加支持push模式 |
| Push模式下消费ack的高可靠机制 | 不支持 | 不支持 | 采用滑动窗口的方式来保证单queue下并发消费的offset覆盖问题，确保单消息的安全性； |
| 支持多线程并发消费单queue的能力 | 不支持 | 不支持 | 在原生支持线程模型之上，增加单queue对多线程的支持 |
| 消费端负载均衡策略 | Sdk仅支持RangeAssignor方式 | 支持Range/RoundRobin/Sticky的分配模式 | 支持v1.0.0所有负载模式；  实现丰富分配策略，比如idc机房的分配策略，走配置文件的分配方式等 |
| 消息的广播模式 | 支持组间广播,带来组管理的复杂性以及客户端配置问题的繁琐 | 同v0.8.x模式 | 在原有广播模式基础上，改进组内广播模式，解决组间广播的痛点问题 |
| 灰度发送和消费 | 不支持 | 不支持 | 实现vhost的资源隔离机制，满足业务同资源的分流问题和解决发布灰度问题； |
| 消息回溯机制 | Push模式下不支持  Pull模式下支持offset回溯 | Push模式下不支持  支持offset和time的回溯 | 可实现push模式下的按offset和time以及业务埋点信息的回溯方式 |
| 消息查询 | offset查询 | offset查询  timestamp的查询 | 同v1.0.0 依赖外部索引系统，提供更多元化的查询功能 |
| 发送和消费审计信息埋点 | 不支持 | 不支持 | 基于v1.0.0进行埋点信息收集并进行线下准实时计算可对消息按业务维度审计。增强消息的可靠性。 |
| 单实例指定链接补偿机制 | 不支持 | 不支持 | 需开发，对于指向性的消息补偿 |
| 状态汇报，运维治理  监控告警平台级支持 | 不支持 | 不支持 | 可封装并结合企业内部监控或告警平台进行配合治理 |
| 客户端配置平台化管理 | 客户度自配置 | 客户度自配置 | 结合消息平台统一管理能力，可统一业务的所有实例级参数和集中化变更 |

1. **总结**

通过分析消息系统关键技术及对比不同消息系统的功能特征；结合目前公司内部各业务团队的功能需求，综合考虑各个选型依据，选择kafka引擎来实现公司内部的消息总线是十分经济，基于已有消息功能并结合新业务需求也易于开发新的功能点，未来基于kafka的服务在可靠性，扩展性和可用性上都会有更好的发展。