# 背景

有多种方式实现远程过程调用（RPC）——比如UNIX RPC、REST API、SOAP和thrift等架构。但是这些传统的RPC实现方法有共同之处，那就是客户端和服务端强耦合（thrift不是强耦合）。客户端直接连上服务端，发送一个请求，然后停下来等待服务器的应答。这种点对点性质的模式有很多好处，它使得在小范围内的拓扑变得简单，但是这种简单的拓扑也限制了灵活性，并且当需要纵向拓展的时候，增加了复杂度。

由于RPC的这些限制，故而在分布式通信中引入MQ。但是，并不是说消息队列就比RPC要好，这要根据具体的使用场景才能区分。

# 通信模型

## 同步通信模型





## 异步通信模型

http://www.zuidaima.com/blog/4398239732780032.htm

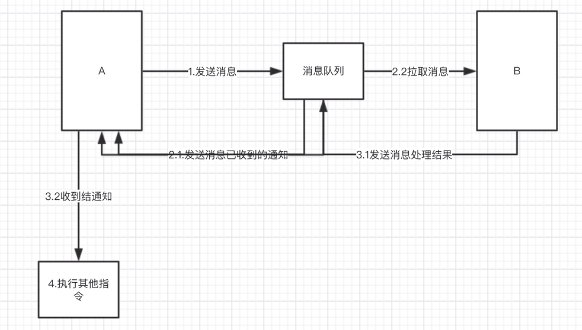
### 线程池



缺点：自己实现线程池，并且强耦合。

### 消息队列

消息队列使得同步通信变成异步通信：



这样就避免了前面发送消息和处理消息的耗时，这里只需要发送消息到消息队列，消息队列存储消息（不处理消息），然后迅速返回给客户端，然后才由接收端拉取消息处理。

#### 应用解耦

在分布式环境下，系统间的相互依赖，最终会会导致整个依赖关系混乱，特别在微服务环境下，会出现相互依赖，甚至是循环依赖的情况，对后期系统的拆分和优化都带来极大负担。那么我们就可以用MQ来进行处理。上游系统将数据投递到MQ，下游系统取MQ的数据进行消费，投递和消费可以用同步的方式处理，因为MQ接收数据的性能是非常高的，不会影响上游系统的性能。



场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口，如下：



传统模式的缺点：

1. 假如库存系统无法访问，则订单减库存将失败，从而导致下单失败；
2. 订单系统与库存系统耦合。

引入消息队列后的方案，如下图：



订阅系统：用户下单后，订阅系统完成持久化处理，将消息写入消息队列，返回用户，下单成功。

库存系统：订阅下单的消息，获取下单信息后，库存系统根据下单信息，进行库存操作。

假如再下单时库存系统不能正常工作，也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了，实现订单系统与库存系统的应用解耦。

#### 流量削峰

由于使用消息，我们的链路变成了生产者发送消息，消息中间件存储消息，最后消费者从消息中间件拉取消息的一个过程。而消息中间件的存储能力能够有效的帮助消费者进行缓冲。试想下，正常流量下消费者能够愉快的进行消费，瞬时高峰流量来的时候，消费者消费能力跟不上，刚好阻塞在消息中间件，等峰值过后，消费者又能很快的将阻塞的消息进行消费。



流量削峰也是消息队列中的常用场景，一般在**秒杀或团购**活动中广泛应用。

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列，作用如下：

1. 可以控制活动的人数；
2. 可以缓解短时间内高流量压垮应用。



用户的请求，服务器接收后，首先写入消息队列，加入消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

秒杀业务根据消息队列中的请求消息，再做后续处理。

#### 异步处理

如果采用同步的方式，系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？引入消息队列，将不必要的业务逻辑异步处理。

异步处理也可以引来并行处理的使用姿势。在工作中，我们基于消息开发了一个简单的分布式任务处理组件。该组件简单分为三块分别是切分、加载、执行三个阶段。

每个阶段都是以作为消费者，然后处理完毕后再作为生产者发送消息。消息消费无状态，可以按需无限拓容。



场景说明：用户注册后，需要发送注册邮件和注册短信，传统的做法如下：

将注册信息写入数据库成功后，发送注册邮件，再发送注册短信，以上三个任务全部完成后，返回给客户端。



引入消息队列，处理异步，改造后架构：



#### 数据分发

大部分开源的MQ中间件基本都支持一对多或者广播的模式，而且都可以根据规则选择分发的对象。这样上游的一份数据，众多下游系统中，可以根据规则选择是否接收这些数据，这样扩展性就很强了。

### 联系/区别

1、两者内部都使用了队列，如阻塞队列、优先级队列；

2、使用线程池时应用服务器既充当生产者又充当消费者，也是消息队列中间件的实现者，使用消息队列时中间件、生产者、消费者可以部署在不同的应用机器上（当然也可以部署在一台服务器上但很少有人这么用）；

3、出于第2点线程池更适合非分布式的系统，但在分布式架构下消息队列明显是更优项；

4、使用消息队列会带来额外的网络开销；

5、消息队列的耦合性更低，可扩展性更好，适用于弱一致性的场景，如对log日志的解耦；

6、消息队列自动实现消息的持久化，中间已经实现了大量功能，如消息转发、消息拒绝、消息重试，以及对消息的一些监控，例如消息的消费状态、消息的消费速率、消息内容查询等，使用线程池如果需要很多功能还要自己去实现，例如想要知道执行状态还需要打印队列数量、计算消息消费速度；

7、在不同系统间的服务调用（调用协议也可能不一致）线程池很难实现或开销很大，这时候消息队列可以屏蔽不同机器或不同协议的问题；

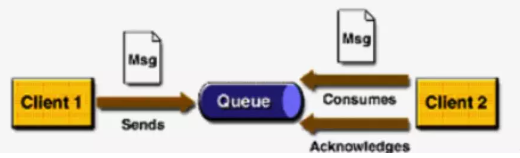
8、使用消息队列会提升系统的复杂度，网络抖动怎么办？最大队列长度怎么设置？超时时间又设置多少？Qos又设置为多少？消费者多少个比较合适？Channel cache size又该设置为多少？业务线可能都是用同一个Mq，你占资源太多，或者设计不当可能会导致整个Mq故障。

# JMS消息模型

JMS=Java Message Service

## P2P对点对模式

P2P（Point to Point）模式包含三个角色：消息队列（Queue），发送者（Sender），接收者（Reciver）。



每个消息都被发送到一个特定的队列，接收者从队列中获取消息。队列保留着消息，直到它们被消费或者超时。

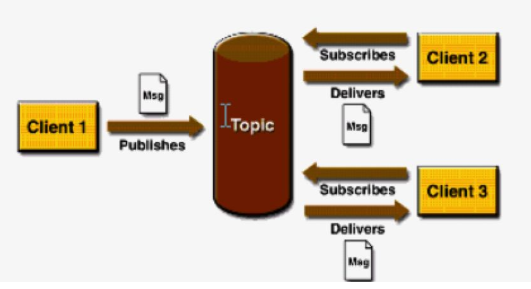
P2P的特点：

1. 每个消息只有一个消费者（即一旦被消费，消息就不再在消息队列中）；
2. 发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列；
3. 接收者在成功接收到消息之后需向队列应答成功，消息队列发现消息消费成功即做消息移除。

**如果希望发送的每个消息都会被成功处理的话，那么需要P2P模式**。

## Publish/Subscribe发布订阅模式

Pub/Sub模式包含三个角色：主体（Topic），发布者（Publisher），订阅者（Subscriber）。多个发布者将消息发送到Topic，系统将这些消息传递给多个订阅者。



Pub/Sub的特点：

1. 每个消息可以有多个消费者；
2. **发布者和订阅者之间有时间上的依赖性**，针对某个主体（topic）的订阅者，它必须创建一个订阅之后，才能消费发布者的消息；
3. 为了消费消息，订阅者必须保持运行的状态。

为了缓和这样严格的时间相关性，JMS允许订阅者创建一个可持久化的订阅。这样，即使订阅者没有被激活（运行），它也能接收到发布者的消息。

如果希望发送的消息可以被多个消费者处理的话，那么可以采用Pub/Sub模型。

通过这两种消息模式的灵活应用以及功能扩展，我们可以实现各种具体的消息应用场景，比如高并发下的订单异步处理，海量日志数据的分析处理等等。如果要总结一下消息队列在各类架构设计中能起到的作用，一般有如下几点：

1. 为系统增加了通用性的异步业务处理能力。
2. 降低系统间的耦合性，无论是开发期的引用关系依赖，还是运行期的调用关系依赖，都明显简化或降低了。通信的双方只需要定义好消息的数据格式（消息头有什么字段，消息体是什么格式的数据），就可以各自开发和测试，最后再各自上线即可集成到一起。
3. 提升了系统间通信可靠性，无论是从通信本身的可靠性上（请求响应机制、重试），还是业务意义上（处理顺序、事务、失败策略），都相比RPC等方式有所增强。
4. 提升了系统的业务缓冲能力，一般又叫削峰填谷，指的是经过MQ做为中间的缓冲，如果业务量突然增大时可以先把处理请求缓冲到队列中，再根据业务消费处理能力逐个消息处理，保障了系统不会因为突然爆发的大量请求而过载瘫痪，影响系统的连续服务能力。
5. 增强了系统的扩展能力，通过消息队列处理的业务，消费端的处理能力如果不够，一般可以随时多加几个消费者来处理，从而可以直接扩展系统的业务处理能力，而不需要额外的代价。

# 常见问题

## 消息阻塞

1、消息阻塞一般都是流量激增，超过消费者消费能力；

2、或者消费者出现逻辑问题，导致不断的重试或长时间等待。

第一种可以通过扩容解决；

第二种只能紧急修复问题，发布上线，在阻塞的过程中会造成大量的消息积压，这种情况也可以考虑临时扩容。

## 重复消费/幂等

### 背景

编程中的“幂等性”是指任意多次执行所产生的影响，与一次执行的影响相同。一个拥有幂等性设计的接口，保证无论一次或多次来调用接口，都能够得到相同的结果。接口的幂等性设计在某些场景下是必需的，例如用户下单的场景。

我们知道，服务之间的调用存在三种状态：成功、失败、超时。超时是一种未知的状态：被调服务是否执行成功，这个状态是未知的。上游服务调用下游服务超时时可能会进行重试。对于用户下单的场景的超时重试我们考虑以下问题：

是否会导致最终创建了两条一样的订单？

是否会扣除两遍库存？

是否会重复扣除用户的钱？

如果每一笔订单都携带唯一的序号，下单接口可以借助这个序号，来记录某次下单操作的状态。当下单的状态为成功时，就将重复的执行拦截住，避免出现上述的问题。这种方式是由下游被调方来保证幂等性。

除此之外，订单服务也可以提供查询订单状态的接口，上游在下单之前先进行查询，确认该笔订单并没有成功支付后，再重复进行下单操作。

一般来说，服务本身需要自己保证幂等性，而不应该将幂等性交给上游的调用方来做。

前端重复提交选中的数据，应该后台只产生对应这个数据的一个反应结果。我们发起一笔付款请求，应该只扣用户账户一次钱，当遇网络重发或系统bug重发，也应该只扣一次钱；发送消息，也应该只发一次，同样的短信发给用户，用户会哭的；创建业务订单，一次业务请求只能创建一个，创建多个就会出大问题。等等很多重要的情况，这些逻辑都需要幂等的特性来支持。

重复消费一般发生下消费端，比如消费者处理完毕，在准备进行ack的时候出现了问题，应用重启后，消息中间件以为该消息还未处理又推给了消费者，或者消费者拉取的时候重复。一般的做法是消费端做幂等。

### 概念

性概念幂等（idempotent、idempotence）是一个数学与计算机学概念，常见于抽象代数中。在编程中，一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。幂等函数，或幂等方法，是指可以使用相同参数重复执行，并能获得相同结果的函数。这些函数不会影响系统状态，也不用担心重复执行会对系统造成改变。例如，“getUsername()和setTrue()”函数就是一个幂等函数.更复杂的操作幂等保证是利用唯一交易号(流水号)实现.我的理解：幂等就是一个操作，不论执行多少次，产生的效果和返回的结果都是一样的。

### 设计

#### 唯一ID

要做到幂等性，就需要借助一个唯一的ID来标志每次交易。唯一ID的分配可以有几种方式：

1、由一个统一的ID分配中心来分配。

2、由上游服务来生成唯一ID，但必须保证不产生冲突的ID。

采用统一的分配中心来分配唯一ID时，业务方每次调用接口都多了一次调用分配中心获取唯一ID的请求。这多了额外的开销。获取唯一ID有一种方式，是借助mysql的自增索引，这其实也是一个ID分配中心。对服务性能有苛刻要求时，可以采用第二种方式，由主调服务本身来生成这个唯一ID。为了保持不会产生重复的ID，可以使用一下几种ID生成方法：

#### UUID

UUID的全称是Universally Unique Identifier，通用唯一识别码。具体可以看维基百科的介绍：https://en.wikipedia.org/wiki/Universally\_unique\_identifier

UUID是一个128bit的数字，用于标志计算机的信息，虽然UUID不能保证绝对不重复，但重复的概率小到可以被忽略。UUID的生成没有什么规律，为了保证UUID的唯一性，规范定义了包括网卡MAC地址、时间戳、名字空间（Namespace）、随机或伪随机数、时序等元素，以及从这些元素生成UUID的算法。这也就意味着：

1、128bit，占据了太多的内存空间

2、生成的ID不是人可以看懂的

3、无法保证ID的递增，某些场景需要按前后排序 无法满足。

这是一个在线生成UUID的网站：https://www.uuidgenerator.net/ 你可以直观感受一下UUID。

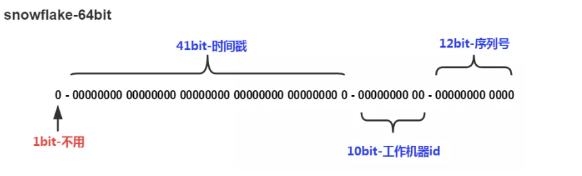
#### Snowflake

这是Twitter的一个开源项目，它是一个分布式ID的生成算法，它会产生一个long类型的唯一ID，其核心算法是：

时间部分：41bit作为毫秒数，大概可以使用69.7年

机器编号部分：10bit作为机器编号，支持1024个机器实例。

毫秒内的序列号：12bit，一毫米可以生成4096个序列号



实际上，redis 或是 mongoDB 的全局ID生成器的算法和Snowflake算法大同小异。这是基于redis的分布式ID生成器实现：https://github.com/hengyunabc/redis-id-generator

它的核心思想是：

使用41 bit来存放时间，精确到毫秒，可以使用41年。

使用12 bit来存放逻辑分片ID，最大分片ID是4095

使用10 bit来存放自增长ID，意味着每个节点，每毫秒最多可以生成1024个ID

#### 共享存储

如果我们的幂等性服务是分布式的，那么存储唯一ID也需要采用共享的存储，这样每个服务就是无状态的了。可以使用mysql来存储，也可以使用k-v存储例如redis。

#### 避免不必要的查询

并不是所有的请求都是重复的，生产环境下可能99%的请求都不是重复请求。如果每个请求在执行前都要去查询下唯一ID是否存在，可能会带来不必要的性能消耗。如果你使用mysql来存储唯一ID，那么可以直接进行insert，通过结果来判断是否插入记录成功，如果不成功则证明ID已经存在：

insert into ... values ... on DUPLICATE KEY UPDATE ...

而如果使用的是redis，也可以使用redis的setEx，设置成功则证明key不存在，否则key存在说明是重复请求。

### 技术方案

1. 查询操作

查询一次和查询多次，在数据不变的情况下，查询结果是一样的。select是天然的幂等操作

1. 删除操作

删除操作也是幂等的，删除一次和多次删除都是把数据删除。(注意可能返回结果不一样，删除的数据不存在，返回0，删除的数据多条，返回结果多个)

1. 唯一索引

防止新增脏数据。比如：支付宝的资金账户，支付宝也有用户账户，每个用户只能有一个资金账户，怎么防止给用户创建资金账户多个，那么给资金账户表中的用户ID加唯一索引，所以一个用户新增成功一个资金账户记录要点：唯一索引或唯一组合索引来防止新增数据存在脏数据（当表存在唯一索引，并发时新增报错时，再查询一次就可以了，数据应该已经存在了，返回结果即可）

1. token机制，防止页面重复提交

**业务要求：**页面的数据只能被点击提交一次

**发生原因：**由于重复点击或者网络重发，或者nginx重发等情况会导致数据被重复提交

**解决办法：**

集群环境：采用token加redis（redis单线程的，处理需要排队）

单JVM环境：采用token加redis或token加jvm内存

**处理流程：**数据提交前要向服务的申请token，token放到redis或jvm内存，token有效时间提交后后台校验token，同时删除token，生成新的token返回

**token特点：**要申请，一次有效性，可以限流

**注意：**redis要用删除操作来判断token，删除成功代表token校验通过，如果用select+delete来校验token，存在并发问题，不建议使用

1. 悲观锁

获取数据的时候加锁获取select \* from table\_xxx where id='xxx' for update;

注意：id字段一定是主键或者唯一索引，不然是锁表，会死人的悲观锁使用时一般伴随事务一起使用，数据锁定时间可能会很长，根据实际情况选用

1. 乐观锁

乐观锁只是在更新数据那一刻锁表，其他时间不锁表，所以相对于悲观锁，效率更高。 乐观锁的实现方式多种多样可以通过version或者其他状态条件：

1. 通过版本号实现u

pdate table\_xxx set name=#name#,version=version+1 where version=#version#

1. 通过条件限制

update tablexxx set avaiamount=avaiamount-#subAmount# where avaiamount-#subAmount# >= 0

要求：quality-#subQuality# >= ，这个情景适合不用版本号，只更新是做数据安全校验，适合库存模型，扣份额和回滚份额，性能更高注意：乐观锁的更新操作，最好用主键或者唯一索引来更新,这样是行锁，否则更新时会锁表，上面两个sql改成下面的两个更好

update tablexxx set name=#name#,version=version+1 where id=#id# and version=#version#

update tablexxx set avaiamount=avaiamount-#subAmount# where id=#id# and avai\_amount-#subAmount# >= 0

1. 分布式锁

还是拿插入数据的例子，如果是分布是系统，构建全局唯一索引比较困难，例如唯一性的字段没法确定这时候可以引入分布式锁，通过第三方的系统(redis或zookeeper)，在业务系统插入数据或者更新数据，获取分布式锁，然后做操作，之后释放锁这样其实是把多线程并发的锁的思路，引入多多个系统，也就是分布式系统中得解决思路。

要点：某个长流程处理过程要求不能并发执行，可以在流程执行之前根据某个标志(用户ID+后缀等)获取分布式锁，其他流程执行时获取锁就会失败，也就是同一时间该流程只能有一个能执行成功，执行完成后，释放分布式锁(分布式锁要第三方系统提供)

1. select + insert

并发不高的后台系统，或者一些任务JOB，为了支持幂等，支持重复执行，简单的处理方法是，先查询下一些关键数据，判断是否已经执行过，在进行业务处理，就可以了。

注意：核心高并发流程不要用这种方法。

1. 状态机幂等

在设计单据相关的业务，或者是任务相关的业务，肯定会涉及到状态机(状态变更图)，就是业务单据上面有个状态，状态在不同的情况下会发生变更，一般情况下存在有限状态机 如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不能够变更的，这样的话，保证了有限状态机的幂等。

注意：订单等单据类业务，存在很长的状态流转，一定要深刻理解状态机，对业务系统设计能力提高有很大帮助。

1. 对外提供接口的api如何保证幂等

如银联提供的付款接口：需要接入商户提交付款请求时附带：source来源，seq序列号source+seq在数据库里面做唯一索引，防止多次付款，(并发时，只能处理一个请求)。

重点对外提供接口为了支持幂等调用，接口有两个字段必须传，一个是来源source，一个是来源方序列号seq，这个两个字段在提供方系统里面做联合唯一索引这样当第三方调用时，先在本方系统里面查询一下，是否已经处理过，返回相应处理结果；没有处理过，进行相应处理，返回结果。注意，为了幂等友好，一定要先查询一下，是否处理过该笔业务，不查询直接插入业务系统，会报错，但实际已经处理了。

## 消息丢失

消息丢失一般分为生产者发送失败、消息中间件丢失、消费丢失。

生产者丢失：可能以为网络问题或者消息中间处理失败导致，消息遗漏。

消息中间的丢失：一般中间件可以设置丢弃策略，大部分MQ中间件产品可以保证数据不丢失，这种情况基本不用考虑。

消费丢失：有的消息中间件支持自动ack，当消费者消费到消息，消息中间件也不管是否消费成功自动ack。这时候一般选择消费者主动ack比较合适。

## 消息顺序性

消息顺序性一般通过MQ中间件保证，大部分MQ中间件只能做到局部有序，比如Kafka，只能保证单个partition队列有序。有些也会做到全局有序，但是成本比较高。

# 消息中间件

消息中间件可以解决如下问题：异步处理，应用解耦，流量削峰，日志处理，从而实现高性能，高可用，可伸缩和最终一致性的架构。

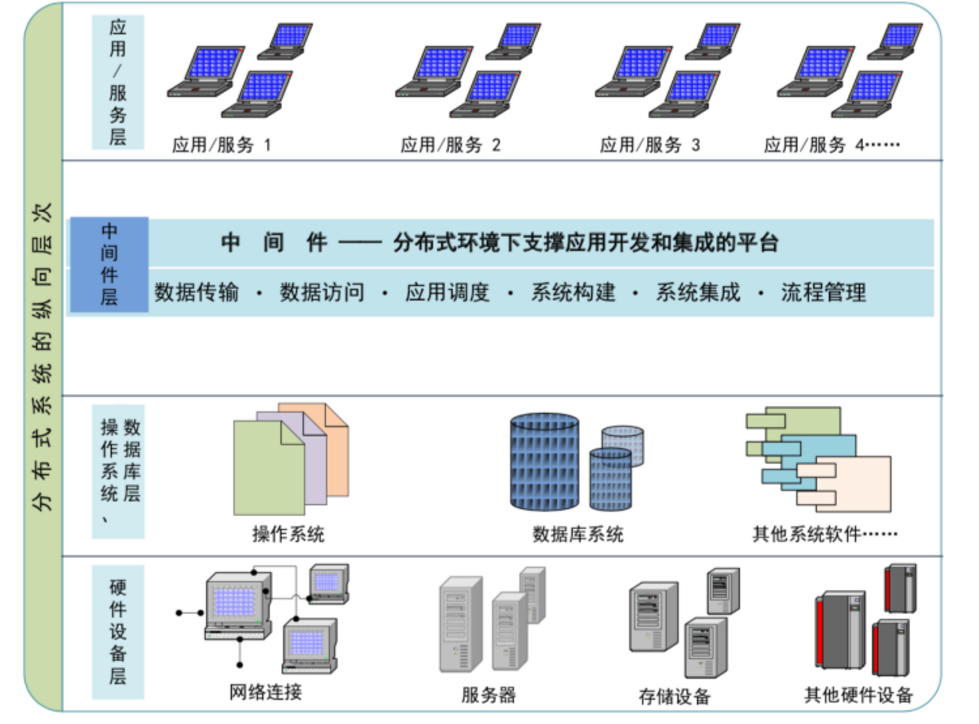
## 背景

生产者和消费者速度或稳定性等因素的不一致。

## 概述

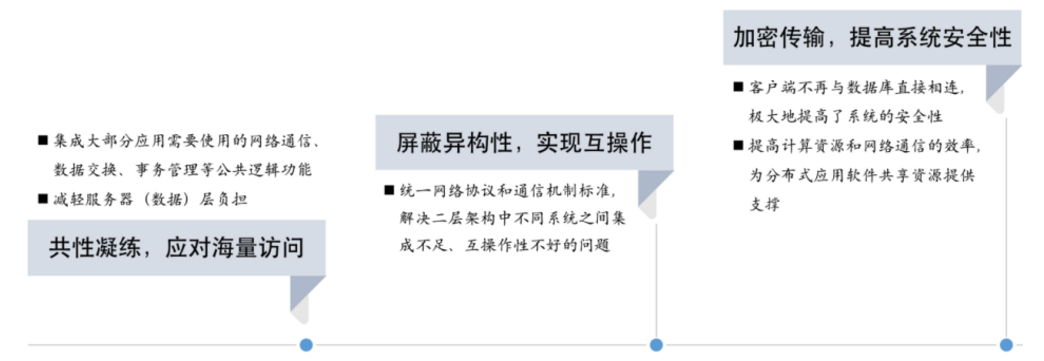
中间件，英文名称为Middleware，是一种应用于分布式系统的基础软件。从纵向层次来看，中间件位于各类应用/服务与操作系统/数据库系统以及其他系统软件之间，主要解决分布式环境下数据传输、数据访问、应用调度、系统构建和系统集成、流程管理等问题，是分布式环境下支撑应用开发、运行和集成的平台，能够实现系统之间的互联互通，帮助用户高效开发应用软件。

中间件在分布式系统中的作用：



中间件伴随分布式计算架构发展而演化。中间件发展和计算架构变化紧密相关，随着网络架构发展，许多应用软件需要在不同的硬件平台、网络协议环境下运行，传统的两层分布式架构，即客户端—服务器架构，面临着性能差、效率低的问题，无法满足需求，三层或多层分布式架构由此提出，具体业务逻辑与底层逻辑解耦，分离至中间层，最终由中间件统一接口和协议，将客户端与服务器进行组合。

中间件解决了二层架构中的缺陷：



MQ（Message Queue，消息队列）是一种应用程序对应用程序的通信方法。应用程序通过读写出入队列的消息（针对应用程序的数据）来通信，而无需专用链接来连接它们。

消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常是用于远程过程调用技术。排队指的是应用程序通过队列来通信，队列的使用出去了接收和发送应用程序同时执行的要求。

### Provider

消息生产者，就是投递消息的程序。

### Consumer

消息消费者，就是接受消息的程序。

没有使用消息队列时消息传递方式：



使用消息队列后消息传递方式：



### 队列

队列就像是存放了商品的仓库，是生产商品的工厂和购买商品的用户之间的中转站。

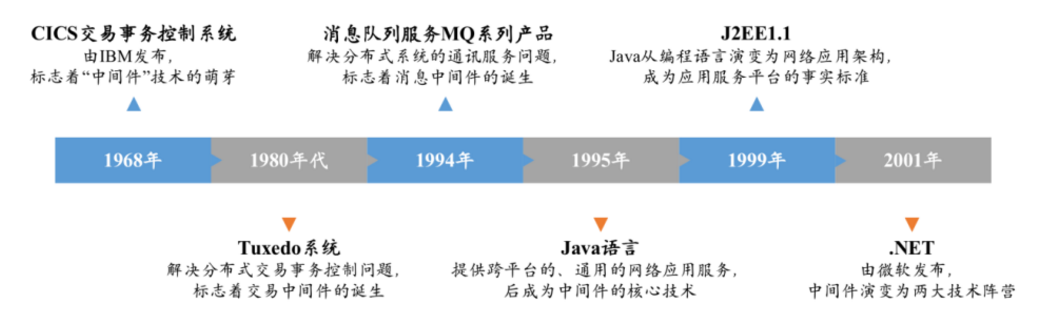
**队列里存储了什么？**

在RabbitMQ中，信息流从你的应用程序出发，来到RabbitMQ的队列，所有信息可以只存储在一个队列中。队列可以存储很多信息，因为它基本上是一个无限制的缓冲区，前提是你的机器有足够的存储空间。

**队列和应用程序的关系：**

多个生产者可以将消息发送到同一个队列中，多个消费者也可以只从同一个队列接收数据。

## 发展历程

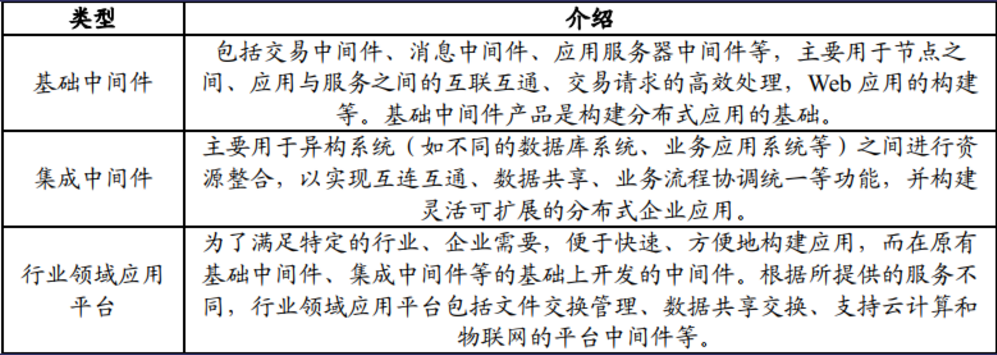


1968年，IBM发布CICS交易事务控制系统，使得应用软件与系统服务分离，带有“中间件”的技术思想，但由于不是分布式环境的产物，因此不将其作为正式的中间件。1980年代，AT&T的贝尔实验室开发了Tuxedo系统，标志着交易中间件的诞生，该系统早期只是实验室产品，后由BEA收购，最终归属于Oracle。1994年，IBM发布消息队列服务MQ系列产品，标志着消息中间件的诞生。

中间件主要分为两大技术阵营。Java语言诞生以来，特别是J2EE（后更名为JAVA EE）标准的发布，中间件的开发标准实现了统一。同时，IBM、Oracle等厂商积极参与J2EE标准制定，走的是开放路线，造就了强大的生命力。2001年，微软发布.NET，中间件演变为两大技术阵营。目前，Java阵营覆盖范围最广，而.NET阵营主要由微软及其伙伴使用。

## 分类

中间件的功能特点、自身定位决定了其分类的多样性。具体的，中间件可以分为基础中间件、集成中间件和行业领域应用平台。



基础中间件

 中间件产生之初主要解决分布式环境下软件性能和可靠性的问题，包括交易中间件、消息中间件等。此后，为满足应用软件在不同硬件平台、网络环境下运行的需求，应用服务器中间件随之出现。 交易中间件是面向对象技术与分布式计算技术结合的产物，其高效地传递交易（事务）请求，协调事务的各个分支，保证事务的完整性，调度应用程序的运行，实现整个系统运行的高效性。交易中间件适用于联机交易系统，如银行业务系统、订票系统等，在金融、财税、电信等行业中得到广泛落地。 消息中间件解决了分布式计算环境下多个子系统间的消息通信问题。其建立网络异步通信的通道，实现不同或同一计算机系统的应用通信，为网络环境下分布式应用系统的开发和运行提供灵活、易用的支撑平台，通常用来在各个系统或者组件间发送消息数据。消息队列是消息中间件的一种实现方式。

## 特性

**典型的特点：**

1、解耦

2、可靠投递

3、广播

4、最终一致性

5、流量削峰

6、消息投递保证

7、异步通信（支持同步）

8、提高系统吞吐、健壮性

**消息队列的特性：**

1. 业务无关：只做消息分发
2. FIFO：先投递先到达
3. 容灾：节点的动态增删和消息的持久化
4. 性能：吞吐量提升，系统内部通信效率提高

### 优点

1. 业务耦合；
2. 最终一致性（记录+补偿机制实现）；
3. 广播；
4. 错峰与流控。

### 缺点

1. 系统更复杂，多了一个MQ组件

MQ的加入大大增加了系统的复杂度，以前系统间是同步的远程调用，现在是通过MQ进行异步调用。

2）消息传递路径更长，延时会增加

3）消息可靠性和重复性互为矛盾，消息不丢不重难以同时保证

4）上游无法知道下游的执行结果，这一点是很致命的。

## 使用场景

<https://blog.csdn.net/weixin_41141219/article/details/80643461>

**什么时候不使用MQ？**

上游实时关注执行结果（比如数据库连接池，使用的是线程池而不是MQ）。

**什么时候使用MQ？**

1）数据驱动的任务依赖

2）上游不关心多下游执行结果

3）异步返回执行时间长

秒杀业务中利用MQ来实现流量削峰，以及应用解耦使用。

## ActiveMQ

### 概述

ActiveMQ是Apache出品，最流行的，能力强劲的开源消息总线，并且它一个完全支持JMS规范的消息中间件。

其丰富的API、多种集群构建模式使得他称为业界老牌消息中间件，在小型企业中应用广泛（**并发不是很大的场景**），但是对于高并发、大数据的场景就力不从心了。

MQ衡量指标：服务性能、数据存储、集群架构

### 原理

#### Topic消息失败重发

消息失败是指消息的消费方没有成功读取消息，此时需要重发消息，最终保持事务完整性和数据一致性。

JMS消息确认机制：

在session接口中定义的几个常量：

AUTO\_ACKNOWLEDGE = 1，自动确认

CLIENT\_ACKNOWLEDGE = 2，客户端手动确认

DUPS\_OK\_ACKNOWLEDGE = 3，自动批量确认

SESSION\_TRANSACTED = 0，事务提交并确认

代码实现：

消息消费端在创建session对象时需要指定应答模式位客户端手动应答，当消费者获取到消息并成功处理后需要调用message.achnowledge()方法进行应答，通知Broker消费成功。如果处理过程中出现异常，需要调用session.recover()通知Broker重复消息，默认最多重复6次。

#### Topic消息持久化订阅

##### 未持久化

必须先启动消费端，然后启动客户端发送消息到topic。如果客户端先向topic发送消息，然后启动消费端则无法消费该消息。此时发送端和消费端存在时间顺序。

##### 持久化到文件（默认）

基本步骤：

1. 在${activemq.base}/conf/activemq.xml文件中配置持久化适配器；
2. 在Java代码中发送消息时：
3. 消息订阅者方创建消费者对象时：

connection.setClientID()”client1-sub”;

TopicSubscriber consumer = session.createDurableSubscriber(topic,’client1-sub’);

##### 持久化到数据库

基本步骤：

1. 将MySQL的数据库驱动复制到ActiveMQ的lib目录下；
2. 在${activemq.base}/conf/activemq.xml文件中配置持久化适配器；
3. 在${activemq.base}/conf/activemq.xml文件中配置数据源。

### 架构

#### Master-Slave模式



#### Network模式



### Zookeeper+ActiveMQ

使用zookeeper实现的master-slave实现方式，是对ActiveMQ进行高可用的一种有效的解决方案。

高可用的原理：使用zookeeper（集群）注册所有的ActiveMQ Broker。只有其中的一个Broker可以对外提供服务（也就是Master节点），其他的Broker处于待机状态，被视为Slave。如果Master因故障而不能提供服务，则利用zookeeper的内部选举机制=从slave中选举出一个Broker充当Master节点，继续对外提供服务。通过zookeeper+activemq实现的集群，可以有效的排除单点故障引起的服务中断。

## Kafka

### 概述

Kafka是LinkedIn开源的分布式分布-订阅消息系统，目前归属于Apache顶级项目。

Kafka主要特点是基于Pull的模式来处理消息消费，追求高吞吐量，一开始的目的就是用于日志收集和传输（大数据场景）。0.8版本开始支持复制，不支持事务，对消息的重复、丢失、错误没有严格要求，适合产生大量数据的互联网服务的数据收集业务。

### 集群模式



### 特点

优点：性能快，吞吐量大，并且高于其他消息队列一个水平。即使在消息量巨大的情况下还能保持高性能。

缺点：不支持消费失败重试，不支持事务消息，不支持定时消息，不支持不支持消息查询等。这些都需要业务上进行额外的开发以实现或者补偿。

### 应用

kafka适合那种简单没有太多业务要求，能容忍一些失败消息，并且消息量大的场景。比如：日志处理，大数据框架数据的收集。但是，如果是在一个金融交易业务上，使用kafka，那么就需要做很多额外的开发来弥补kafka的不足之处。

## RocketMQ

### 概述

RocketMQ是阿里开源的消息中间件，目前已经孵化为Apache顶级项目，它是纯Java开发，具有高吞吐量、高可用性、适合大规模分布式系统应用的特点。

RocketMQ思路起源于Kafka，它对消息的可靠传输及事务性做了优化，目前在阿里集团被广泛应用于交易、充值、流计算、消息推送、日志流式处理、binglog分发等场景。

### 集群拓扑



### 应用

可以按照以下的思路选择消息队列的解决方案：

如果是一个不大的系统，不一定要用消息队列引擎，库就能解决，先选择性能快的kafka，思考kafka能不能满足业务需求，kafka不能，则选择mq。

近两年RocketMQ流行起来，性能优异，并且有大规模生产实践的例子，会是不错的选择。

## RabbitMQ

### 概述

RabbitMQ是使用Erlang语言开发的开源消息队列系统，基于AMQP协议来实现。AMQP的主要特征是面向消息、队列、路由（包括点对点和分布订阅）、可靠性、安全。AMQP协议更多用在企业系统内，对数据**一致性、稳定性和可靠性**要求很高的场景，对性能和吞吐量的要求还在其次。

RabbitMQ 在有大量消息堆积的情况下性能会下降，Kafka不会。毕竟AMQP设计的初衷不是用来持久化海量消息的，而Kafka一开始是用来处理海量日志的。

在RabbitMQ中，一个消息的大致流程是这样的：

1. 连接到RabbitMQ
2. 获取信道
3. 声明交换器
4. 创建消息
5. 发布消息
6. 关闭信道
7. 关闭连接

### 集群



### 应用

RabbitMQ遵循AMQP协议，由内在高并发的erlanng语言开发，用在实时的对可靠性要求比较高的消息传递上，适合企业级的消息发送订阅，也是比较受到大家欢迎的。

**AMQP协议更多用在企业系统内，对数据一致性、稳定性和可靠性要求很高的场景，对性能和吞吐量还在其次。**

# 线程池与消息队列

1、两者内部都使用了队列，如阻塞队列、优先级队列；

2、使用线程池时应用服务器既充当生产者又充当消费者，也是消息队列中间件的实现者，使用消息队列时中间件、生产者、消费者可以部署在不同的应用机器上（当然也可以部署在一台服务器上但很少有人这么用）；

3、出于第2点线程池更适合非分布式的系统，但在分布式架构下消息队列明显是更优项；

4、使用消息队列会带来额外的网络开销；

5、消息队列的耦合性更低，可扩展性更好，适用于弱一致性的场景，如对log日志的解耦；

6、消息队列自动实现消息的持久化，中间已经实现了大量功能，如消息转发、消息拒绝、消息重试，以及对消息的一些监控，例如消息的消费状态、消息的消费速率等，使用线程池如果需要很多功能还要自己去实现，例如需要执行状态需要打印队列数量、计算消息消费速度；

7、在不同系统间的服务调用（调用协议也可能不一致）线程池很难实现或开销很大，这时候消息队列可以屏蔽不同机器或不同协议的问题；

8、使用消息队列会提升系统的复杂度，网络抖动怎么办？最大队列长度怎么设置？超时时间又设置多少？QOS又设置为多少？消费者多少个比较合适？Channel cache size又该设置为多少？业务线可能都是用同一个MQ，你占资源太多，或者设计不当可能会导致整个MQ故障。

# 对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 特性 | ActiveMQ | RabbitMQ | RocketMQ | Kafka |
| 开发语言 | Java | Erlang | Java | Scala |
| 单机吞吐量 | 万级 | 万级 | 十万级 | 十万级 |
| 时效性 | ms级 | us级 | ms级 | ms级以内 |
| 可用性 | 高（主从架构） | 高（主从架构） | 非常高（分布式架构） | 非常高（分布式架构） |
| 功能特性 | 成熟产品，很多公司应用；文档较多；各种协议支持好 | 基于erlang开发，所以并发性能好，延时很低；管理界面较为丰富 | MQ比较完备，拓展性佳 | 只支持主要的MQ功能，没有提供消息查询、消息回溯等功能，大数据领域应用广 |

## 资料文档

Kafka：中。有Kafka作者自己写的书，网上资料也有一些。

RabbitMQ：多。有一些不错的书，网上资料多。

ZeroMQ：少。没有专门写ZeroMQ的书，网上的资料多是一些代码的实现和简单介绍。

RocketMQ：少。没有专门写RocketMQ的书，网上的资料良莠不齐，官方文档很简洁，但是对技术细节没有过多的描述。

ActiveMQ：多。没有专门写ActiveMQ的书，网上资料多。

## 开发语言

Kafka：Scala

RabbitMQ：Erlang

ZeroMQ：C

RocketMQ：java

ActiveMQ：java

## 支持的协议

Kafka：自己定义的一套（基于TCP）

RabbitMQ：AMQP

ZeroMQ：TCP、UDP

RocketMQ：自己定义的一套

ActiveMQ：OpenWire、STOMP、REST、XMPP、AMQP

## 消息事务

Kafka：支持

RabbitMQ：支持。客户端将信道设置为事务模式，只有当消息被RabbitMQ接收，事务才能提交成功，否则在捕获异常后进行回滚。使用事务会使得性能有所下降

ZeroMQ：不支持

RocketMQ：支持

ActiveMQ：支持

## 负载均衡

Kafka：支持负载均衡。

kafka采用zookeeper对集群中的broker、consumer进行管理，可以注册topic到zookeeper上；通过zookeeper的协调机制，producer保存对应topic的broker信息，可以随机或者轮询发送到broker上；并且producer可以基于语义指定分片，消息发送到broker的某分片上。

1、一个broker通常就是一台服务器节点。对于同一个Topic的不同分区，Kafka会尽力将这些分区分布到不同的Broker服务器上，zookeeper保存了broker、主题和分区的元数据信息。分区首领会处理来自客户端的生产请求，Kafka分区首领会被分配到不同的broker服务器上，让不同的broker服务器共同分担任务。

每一个broker都缓存了元数据信息，客户端可以从任意一个broker获取元数据信息并缓存起来，根据元数据信息知道要往哪里发送请求。

2、Kafka的消费者组订阅同一个topic，会尽可能地使得每一个消费者分配到相同数量的分区，分摊负载。

3、当消费者加入或者退出消费者组的时候，还会触发再均衡，为每一个消费者重新分配分区，分摊负载。

Kafka的负载均衡大部分是自动完成的，分区的创建也是Kafka完成的，隐藏了很多细节，避免了繁琐的配置和人为疏忽造成的负载问题。

4、发送端由topic和key来决定消息发往哪个分区，如果key为null，那么会使用轮询算法将消息均衡地发送到同一个topic的不同分区中。如果key不为null，那么会根据key的hashcode取模计算出要发往的分区。

RabbitMQ：对负载均衡的支持不好。

RabbitMQ的负载均衡需要单独的loadbalancer进行支持。

1、消息被投递到哪个队列是由交换器和key决定的，交换器、路由键、队列都需要手动创建。

RabbitMQ客户端发送消息要和broker建立连接，需要事先知道broker上有哪些交换器，有哪些队列。通常要声明要发送的目标队列，如果没有目标队列，会在broker上创建一个队列，如果有，就什么都不处理，接着往这个队列发送消息。假设大部分繁重任务的队列都创建在同一个broker上，那么这个broker的负载就会过大。（可以在上线前预先创建队列，无需声明要发送的队列，但是发送时不会尝试创建队列，可能出现找不到队列的问题，RabbitMQ的备份交换器会把找不到队列的消息保存到一个专门的队列中，以便以后查询使用）

使用镜像队列机制建立RabbitMQ集群可以解决这个问题，形成master-slave的架构，master节点会均匀分布在不同的服务器上，让每一台服务器分摊负载。slave节点只是负责转发，在master失效时会选择加入时间最长的slave成为master。

当新节点加入镜像队列的时候，队列中的消息不会同步到新的slave中，除非调用同步命令，但是调用命令后，队列会阻塞，不能在生产环境中调用同步命令。

2、当RabbitMQ队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，RabbitMQ不再向这个消费者发送任何消息。

3、对于RabbitMQ而言，客户端与集群建立的TCP连接不是与集群中所有的节点建立连接，而是挑选其中一个节点建立连接。

但是RabbitMQ集群可以借助HAProxy、LVS技术，或者在客户端使用算法实现负载均衡，引入负载均衡之后，各个客户端的连接可以分摊到集群的各个节点之中。

客户端均衡算法：

1)轮询法。按顺序返回下一个服务器的连接地址。

2)加权轮询法。给配置高、负载低的机器配置更高的权重，让其处理更多的请求；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载。

3)随机法。随机选取一个服务器的连接地址。

4)加权随机法。按照概率随机选取连接地址。

5)源地址哈希法。通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算。

6)最小连接数法。动态选择当前连接数最少的一台服务器的连接地址。

ZeroMQ：去中心化，不支持负载均衡。本身只是一个多线程网络库。

RocketMQ：支持负载均衡。

一个broker通常是一个服务器节点，broker分为master和slave,master和slave存储的数据一样，slave从master同步数据。

1、nameserver与每个集群成员保持心跳，保存着Topic-Broker路由信息，同一个topic的队列会分布在不同的服务器上。

2、发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。发送消息指定topic、tags、keys，无法指定投递到哪个队列（没有意义，集群消费和广播消费跟消息存放在哪个队列没有关系）。

tags选填，类似于 Gmail 为每封邮件设置的标签，方便服务器过滤使用。目前只支 持每个消息设置一个 tag，所以也可以类比为 Notify 的 MessageType 概念。

keys选填，代表这条消息的业务关键词，服务器会根据 keys 创建哈希索引，设置后， 可以在 Console 系统根据 Topic、Keys 来查询消息，由于是哈希索引，请尽可能 保证 key 唯一，例如订单号，商品 Id 等。

3、RocketMQ的负载均衡策略规定：Consumer数量应该小于等于Queue数量，如果Consumer超过Queue数量，那么多余的Consumer 将不能消费消息。这一点和Kafka是一致的，RocketMQ会尽可能地为每一个Consumer分配相同数量的队列，分摊负载。

ActiveMQ：支持负载均衡。可以基于zookeeper实现负载均衡。

## 集群方式

Kafka：天然的‘Leader-Slave’无状态集群，每台服务器既是Master也是Slave。

分区首领均匀地分布在不同的Kafka服务器上，分区副本也均匀地分布在不同的Kafka服务器上，所以每一台Kafka服务器既含有分区首领，同时又含有分区副本，每一台Kafka服务器是某一台Kafka服务器的Slave，同时也是某一台Kafka服务器的leader。

Kafka的集群依赖于zookeeper，zookeeper支持热扩展，所有的broker、消费者、分区都可以动态加入移除，而无需关闭服务，与不依靠zookeeper集群的mq相比，这是最大的优势。

RabbitMQ：支持简单集群，'复制'模式，对高级集群模式支持不好。

RabbitMQ的每一个节点，不管是单一节点系统或者是集群中的一部分，要么是内存节点，要么是磁盘节点，集群中至少要有一个是磁盘节点。

在RabbitMQ集群中创建队列，集群只会在单个节点创建队列进程和完整的队列信息（元数据、状态、内容），而不是在所有节点上创建。

引入镜像队列，可以避免单点故障，确保服务的可用性，但是需要人为地为某些重要的队列配置镜像。

ZeroMQ：去中心化，不支持集群。

RocketMQ：常用 多对'Master-Slave' 模式，开源版本需手动切换Slave变成Master

Name Server是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。

Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，Master与Slave的对应关系通过指定相同的BrokerName，不同的BrokerId来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。每个Broker与Name Server集群中的所有节点建立长连接，定时注册Topic信息到所有Name Server。

Producer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。

Consumer与Name Server集群中的其中一个节点（随机选择）建立长连接，定期从Name Server取Topic路由信息，并向提供Topic服务的Master、Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，订阅规则由Broker配置决定。

客户端先找到NameServer, 然后通过NameServer再找到 Broker。

一个topic有多个队列，这些队列会均匀地分布在不同的broker服务器上。RocketMQ队列的概念和Kafka的分区概念是基本一致的，Kafka同一个topic的分区尽可能地分布在不同的broker上，分区副本也会分布在不同的broker上。

RocketMQ集群的slave会从master拉取数据备份，master分布在不同的broker上。

ActiveMQ：支持简单集群模式，比如'主-备'，对高级集群模式支持不好。

## 管理界面

Kafka：一般

RabbitMQ：好

ZeroMQ：无

RocketMQ：无

ActiveMQ：一般

## 可用性

Kafka：非常高（分布式），kafka的broker支持主备模式。

RabbitMQ：高（主从），RabbitMQ支持miror的queue，主queue失效，miror queue接管。

ZeroMQ：高。

RocketMQ：非常高（分布式）

ActiveMQ：高（主从）

## 吞吐量TPS

Kafka：极大 Kafka按批次发送消息和消费消息。发送端将多个小消息合并，批量发向Broker，消费端每次取出一个批次的消息批量处理。

kafka具有高的吞吐量，内部采用消息的批量处理，zero-copy机制，数据的存储和获取是本地磁盘顺序批量操作，具有O(1)的复杂度，消息处理的效率很高。

RabbitMQ：比较大。rabbitMQ在吞吐量方面稍逊于kafka，他们的出发点不一样，rabbitMQ支持对消息的可靠的传递，支持事务，不支持批量的操作；基于存储的可靠性的要求存储可以采用内存或者硬盘。

ZeroMQ：极大

RocketMQ：大 RocketMQ接收端可以批量消费消息，可以配置每次消费的消息数，但是发送端不是批量发送。

ActiveMQ：比较大

## 并发度

Kafka：高

一个线程一个消费者，Kafka限制消费者的个数要小于等于分区数，如果要提高并行度，可以在消费者中再开启多线程，或者增加consumer实例数量。

RabbitMQ：极高

本身是用Erlang语言写的，并发性能高。

可在消费者中开启多线程，最常用的做法是一个channel对应一个消费者，每一个线程把持一个channel，多个线程复用connection的tcp连接，减少性能开销。

当RabbitMQ队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，RabbitMQ不再向这个消费者发送任何消息。

ZeroMQ：高

RocketMQ：高

1、RocketMQ限制消费者的个数少于等于队列数，但是可以在消费者中再开启多线程，这一点和Kafka是一致的，提高并行度的方法相同。

修改消费并行度方法

a) 同一个 ConsumerGroup 下，通过增加 Consumer 实例数量来提高并行度，超过订阅队列数的 Consumer实例无效。

b) 提高单个 Consumer 的消费并行线程，通过修改参数consumeThreadMin、consumeThreadMax

2、同一个网络连接connection，客户端多个线程可以同时发送请求，连接会被复用，减少性能开销。

ActiveMQ：高

单个ActiveMQ的接收和消费消息的速度在1万笔/秒（持久化 一般为1-2万， 非持久化 2 万以上），在生产环境中部署10个ActiveMQ就能达到10万笔/秒以上的性能，部署越多的ActiveMQ broker 在MQ上latency也就越低，系统吞吐量也就越高。

## 消息存储

Kafka：内存、磁盘、数据库。支持大量堆积。

Kafka的最小存储单元是分区，一个topic包含多个分区，Kafka创建主题时，这些分区会被分配在多个服务器上，通常一个broker一台服务器。 分区首领会均匀地分布在不同的服务器上，分区副本也会均匀的分布在不同的服务器上，确保负载均衡和高可用性，当新的broker加入集群的时候，部分副本会被移动到新的broker上。 根据配置文件中的目录清单，Kafka会把新的分区分配给目录清单里分区数最少的目录。 默认情况下，分区器使用轮询算法把消息均衡地分布在同一个主题的不同分区中，对于发送时指定了key的情况，会根据key的hashcode取模后的值存到对应的分区中。

RabbitMQ：内存、磁盘。支持少量堆积。

RabbitMQ的消息分为持久化的消息和非持久化消息，不管是持久化的消息还是非持久化的消息都可以写入到磁盘。 持久化的消息在到达队列时就写入到磁盘，并且如果可以，持久化的消息也会在内存中保存一份备份，这样可以提高一定的性能，当内存吃紧的时候会从内存中清除。非持久化的消息一般只存在于内存中，在内存吃紧的时候会被换入到磁盘中，以节省内存。

引入镜像队列机制，可将重要队列“复制”到集群中的其他broker上，保证这些队列的消息不会丢失。配置镜像的队列，都包含一个主节点master和多个从节点slave,如果master失效，加入时间最长的slave会被提升为新的master，除发送消息外的所有动作都向master发送，然后由master将命令执行结果广播给各个slave，RabbitMQ会让master均匀地分布在不同的服务器上，而同一个队列的slave也会均匀地分布在不同的服务器上，保证负载均衡和高可用性。

ZeroMQ：消息发送端的内存或者磁盘中。不支持持久化。

RocketMQ：磁盘。支持大量堆积。

commitLog文件存放实际的消息数据，每个commitLog上限是1G，满了之后会自动新建一个commitLog文件保存数据。ConsumeQueue队列只存放offset、size、tagcode，非常小，分布在多个broker上。ConsumeQueue相当于CommitLog的索引文件，消费者消费时会从consumeQueue中查找消息在commitLog中的offset，再去commitLog中查找元数据。

ConsumeQueue存储格式的特性，保证了写过程的顺序写盘（写CommitLog文件），大量数据IO都在顺序写同一个commitLog，满1G了再写新的。加上RocketMQ是累计4K才强制从PageCache中刷到磁盘（缓存），所以高并发写性能突出。

ActiveMQ：内存、磁盘、数据库。支持少量堆积。

## 订阅形式和消息分发

Kafka：基于topic以及按照topic进行正则匹配的发布订阅模式。

【发送】

发送端由topic和key来决定消息发往哪个分区，如果key为null，那么会使用轮询算法将消息均衡地发送到同一个topic的不同分区中。如果key不为null，那么会根据key的hashcode取模计算出要发往的分区。

【接收】

1>consumer向群组协调器broker发送心跳来维持他们和群组的从属关系以及他们对分区的所有权关系，所有权关系一旦被分配就不会改变除非发生再均衡(比如有一个consumer加入或者离开consumer group)，consumer只会从对应的分区读取消息。

2>Kafka限制consumer个数要少于分区个数,每个消息只会被同一个 Consumer Group的一个consumer消费（非广播）。

3>Kafka的 Consumer Group订阅同一个topic，会尽可能地使得每一个consumer分配到相同数量的分区，不同 Consumer Group订阅同一个主题相互独立，同一个消息会被不同的 Consumer Group处理。

RabbitMQ：提供了4种：direct, topic ,Headers和fanout。

【发送】

先要声明一个队列，这个队列会被创建或者已经被创建，队列是基本存储单元。

由exchange和key决定消息存储在哪个队列。

direct>发送到和bindingKey完全匹配的队列。

topic>路由key是含有"."的字符串，会发送到含有“\*”、“#”进行模糊匹配的bingKey对应的队列。

fanout>与key无关，会发送到所有和exchange绑定的队列

headers>与key无关，消息内容的headers属性（一个键值对）和绑定键值对完全匹配时，会发送到此队列。此方式性能低一般不用

【接收】

RabbitMQ的队列是基本存储单元，不再被分区或者分片，对于我们已经创建了的队列，消费端要指定从哪一个队列接收消息。

当RabbitMQ队列拥有多个消费者的时候，队列收到的消息将以轮询的分发方式发送给消费者。每条消息只会发送给订阅列表里的一个消费者，不会重复。

这种方式非常适合扩展，而且是专门为并发程序设计的。

如果某些消费者的任务比较繁重，那么可以设置basicQos限制信道上消费者能保持的最大未确认消息的数量，在达到上限时，RabbitMQ不再向这个消费者发送任何消息。

ZeroMQ：点对点(p2p)

RocketMQ：基于topic/messageTag以及按照消息类型、属性进行正则匹配的发布订阅模式

【发送】

发送消息通过轮询队列的方式发送，每个队列接收平均的消息量。发送消息指定topic、tags、keys，无法指定投递到哪个队列（没有意义，集群消费和广播消费跟消息存放在哪个队列没有关系）。

tags选填，类似于 Gmail 为每封邮件设置的标签，方便服务器过滤使用。目前只支 持每个消息设置一个 tag，所以也可以类比为 Notify 的 MessageType 概念。

keys选填，代表这条消息的业务关键词，服务器会根据 keys 创建哈希索引，设置后， 可以在 Console 系统根据 Topic、Keys 来查询消息，由于是哈希索引，请尽可能 保证 key 唯一，例如订单号，商品 Id 等。

【接收】

1>广播消费。一条消息被多个Consumer消费，即使Consumer属于同一个ConsumerGroup，消息也会被ConsumerGroup中的每个Consumer都消费一次。

2>集群消费。一个 Consumer Group中的Consumer实例平均分摊消费消息。例如某个Topic有 9 条消息，其中一个Consumer Group有3个实例，那么每个实例只消费其中的 3 条消息。即每一个队列都把消息轮流分发给每个consumer。

ActiveMQ：点对点(p2p)、广播（发布-订阅）

点对点模式，每个消息只有1个消费者；

发布/订阅模式，每个消息可以有多个消费者。

【发送】

点对点模式：先要指定一个队列，这个队列会被创建或者已经被创建。

发布/订阅模式：先要指定一个topic，这个topic会被创建或者已经被创建。

【接收】

点对点模式：对于已经创建了的队列，消费端要指定从哪一个队列接收消息。

发布/订阅模式：对于已经创建了的topic，消费端要指定订阅哪一个topic的消息。

## 顺序消息

Kafka：支持。

设置生产者的max.in.flight.requests.per.connection为1，可以保证消息是按照发送顺序写入服务器的，即使发生了重试。

Kafka保证同一个分区里的消息是有序的，但是这种有序分两种情况

1、key为null，消息逐个被写入不同主机的分区中，但是对于每个分区依然是有序的

2、key不为null , 消息被写入到同一个分区，这个分区的消息都是有序。

RabbitMQ：不支持

ZeroMQ：不支持

RocketMQ：支持

ActiveMQ：不支持

## 消息确认

Kafka：支持。

1、发送方确认机制

ack=0，不管消息是否成功写入分区

ack=1，消息成功写入首领分区后，返回成功

ack=all，消息成功写入所有分区后，返回成功。

2、接收方确认机制

自动或者手动提交分区偏移量，早期版本的Kafka偏移量是提交给Zookeeper的，这样使得zookeeper的压力比较大，更新版本的Kafka的偏移量是提交给Kafka服务器的，不再依赖于zookeeper群组，集群的性能更加稳定。

RabbitMQ：支持。

1、发送方确认机制，消息被投递到所有匹配的队列后，返回成功。如果消息和队列是可持久化的，那么在写入磁盘后，返回成功。支持批量确认和异步确认。

2、接收方确认机制，设置autoAck为false，需要显式确认，设置autoAck为true，自动确认。

当autoAck为false的时候，RabbitMQ队列会分成两部分，一部分是等待投递给consumer的消息，一部分是已经投递但是没收到确认的消息。如果一直没有收到确认信号，并且consumer已经断开连接，RabbitMQ会安排这个消息重新进入队列，投递给原来的消费者或者下一个消费者。

未确认的消息不会有过期时间，如果一直没有确认，并且没有断开连接，RabbitMQ会一直等待，RabbitMQ允许一条消息处理的时间可以很久很久。

ZeroMQ：支持。

RocketMQ：支持。

ActiveMQ：支持。

## 消息回溯

Kafka：支持指定分区offset位置的回溯。

RabbitMQ：不支持

ZeroMQ：不支持

RocketMQ：支持指定时间点的回溯。

ActiveMQ：不支持

## 消息重复

Kafka：支持at least once、at most once

RabbitMQ：支持at least once、at most once

ZeroMQ：只有重传机制，但是没有持久化，消息丢了重传也没有用。既不是at least once、也不是at most once、更不是exactly only once

RocketMQ：支持at least once

ActiveMQ：支持at least once

## 消息重试

Kafka：不支持，但是可以实现。

Kafka支持指定分区offset位置的回溯，可以实现消息重试。

RabbitMQ：不支持，但是可以利用消息确认机制实现。

RabbitMQ接收方确认机制，设置autoAck为false。

当autoAck为false的时候，RabbitMQ队列会分成两部分，一部分是等待投递给consumer的消息，一部分是已经投递但是没收到确认的消息。如果一直没有收到确认信号，并且consumer已经断开连接，RabbitMQ会安排这个消息重新进入队列，投递给原来的消费者或者下一个消费者。

ZeroMQ：不支持，

RocketMQ：支持。

消息消费失败的大部分场景下，立即重试99%都会失败，所以RocketMQ的策略是在消费失败时定时重试，每次时间间隔相同。

1、发送端的 send 方法本身支持内部重试，重试逻辑如下：

a)至多重试3次；

b)如果发送失败，则轮转到下一个broker；

c)这个方法的总耗时不超过sendMsgTimeout 设置的值，默认 10s，超过时间不在重试。

2、接收端。

Consumer 消费消息失败后，要提供一种重试机制，令消息再消费一次。Consumer 消费消息失败通常可以分为以下两种情况：

由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被

注销，无法充值）等。定时重试机制，比如过 10s 秒后再重试。

由于依赖的下游应用服务不可用，例如 db 连接不可用，外系统网络不可达等。

即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况可以 sleep 30s，再消费下一条消息，减轻 Broker 重试消息的压力。

ActiveMQ：不支持

# 选型

## 概述

中小型公司，技术实力较为一般，技术挑战不是特别高，用RabbitMQ是不错的选择；大型公司，基础架构研发实力较强，用RocketMQ是很好的选择。

如果是大数据领域的实时计算、日志采集等场景，用Kafka是业内标准。

Redis是一个key-value缓存中间件，而不是一个消息队列中间件。但事实上它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。

实验表明：入队时，当数据比较小时Redis的性能要高于RabbitMQ，而如果数据大小超过了10K，Redis则慢的无法忍受；出队时，无论数据大小，Redis都表现出非常好的性能，而RabbitMQ的出队性能则远低于Redis。

但在实际应用中，大家在考虑消息中间件的时候一般都不考虑 Redis。主要有两个原因，一方面是数据大小超过10K速度很慢，另一个问题是Redis给人的印象就是做缓存的。基于上面这两点原因，**Redis更适合用来做很小规模、业务简单的消息队列场景**。 如果业务复杂、业务规模大，一般情况下Redis就会被排除。

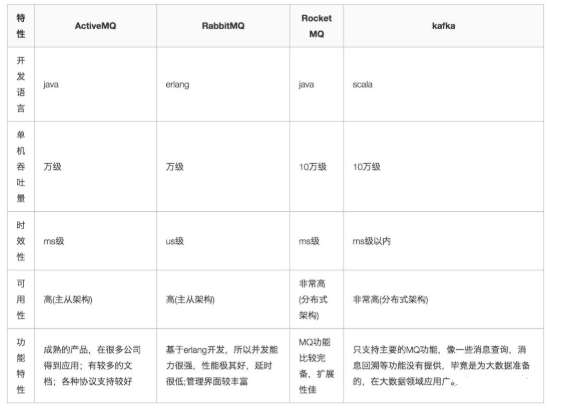
ActiveMQ是Apache下的一个子项目。类似于ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于RabbitMQ，它少量代码就可以高效地实现高级应用场景。

RabbitMQ是使用**Erlang**编写的一个开源的消息队列，本身支持很多的协议：**AMQP，XMPP，SMTP，STOMP**，也正因如此，它非常重量级，更适合于**企业级的开发**。同时实现了**Broker构架**，这意味着消息在发送给客户端时先在中心队列排队。对路由，负载均衡或者数据持久化都有很好的支持。

Kafka是Apache下的一个子项目，是一个**高性能跨语言分布式发布/订阅消息队列系统**。它具有以下特性：快速持久化，可以在O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统，**Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡**；支持Hadoop数据并行加载，对于像Hadoop的一样的日志数据和离线分析系统，但又要求**实时处理**的限制，这是一个可行的解决方案。

RocketMQ是阿里巴巴开源的一个项目，目前已经纳入Apache基金会。其是在Kafka 的基础上发展起来的，起因是随着阿里巴巴业务的发展，他们发现Kafka对于具体业务场景的支持不完善，所以才有了RocketMQ的诞生。

与Kafka比起来，RocketMQ很多方面都极其相似。唯一的不同是RocketMQ对于业务特性的支持更完善，所以**更适用于业务场景**。



从上面的表格我们可以看出几个简单的结论：

无论是在单机吞吐量还是可用性方面，ActiveMQ和RabbitMQ都差不多，而RocketMQ和Kafka差不多。

在功能特性方面，ActiveMQ、RabbitMQ、RocketMQ功能比较完善。Kafka功能性较弱。

衡量一款消息中间件是否符合需求需要从多个维度进行考察，首要的就是功能维度，这个直接决定了你能否最大程度上的实现开箱即用，进而缩短项目周期、降低成本等。如果一款消息中间件的功能达不到想要的功能，那么就需要进行二次开发，这样会增加项目的技术难度、复杂度以及增大项目周期等。

## 功能维度

功能维度又可以划分个多个子维度，大致可以分为以下这些。

### 优先级队列

优先级队列不同于先进先出队列，优先级高的消息具备优先被消费的特权，这样可以为下游提供不同消息级别的保证。

不过这个优先级也是需要有一个前提的：如果消费者的消费速度大于生产者的速度，并且消息中间件服务器（一般简单的称之为 Broker）中没有消息堆积，那么对于发送的消息设置优先级也就没有什么实质性的意义了，因为生产者刚发送完一条消息就被消费者消费了，那么就相当于 Broker 中至多只有一条消息，对于单条消息来说优先级是没有什么意义的。

### 延迟队列

当你在网上购物的时候是否会遇到这样的提示：“三十分钟之内未付款，订单自动取消”？这个是延迟队列的一种典型应用场景。

延迟队列存储的是对应的延迟消息，所谓“延迟消息”是指当消息被发送以后，并不想让消费者立刻拿到消息，而是等待特定时间后，消费者才能拿到这个消息进行消费。

延迟队列一般分为两种：基于消息的延迟和基于队列的延迟。基于消息的延迟是指为每条消息设置不同的延迟时间，那么每当队列中有新消息进入的时候就会重新根据延迟时间排序，当然这也会对性能造成极大的影响。实际应用中大多采用基于队列的延迟，设置不同延迟级别的队列，比如 5s、10s、30s、1min、5mins、10mins 等，每个队列中消息的延迟时间都是相同的，这样免去了延迟排序所要承受的性能之苦，通过一定的扫描策略（比如定时）即可投递超时的消息。

### 死信队列

由于某些原因消息无法被正确的投递，为了确保消息不会被无故的丢弃，一般将其置于一个特殊角色的队列，这个队列一般称之为死信队列。

与此对应的还有一个“回退队列”的概念，试想如果消费者在消费时发生了异常，那么就不会对这一次消费进行确认（Ack）, 进而发生回滚消息的操作之后消息始终会放在队列的顶部，然后不断被处理和回滚，导致队列陷入死循环。为了解决这个问题，可以为每个队列设置一个回退队列，它和死信队列都是为异常的处理提供的一种机制保障。实际情况下，回退队列的角色可以由死信队列和重试队列来扮演。

### 重试队列

重试队列其实可以看成是一种回退队列，具体指消费端消费消息失败时，为防止消息无故丢失而重新将消息回滚到 Broker中。

与回退队列不同的是重试队列一般分成多个重试等级，每个重试等级一般也会设置重新投递延时，重试次数越多投递延时就越大。举个例子：消息第一次消费失败入重试队列Q1，Q1的重新投递延迟为5s，在5s过后重新投递该消息；如果消息再次消费失败则入重试队列 Q2，Q2的重新投递延迟为10s，在10s过后再次投递该消息。以此类推，重试越多次重新投递的时间就越久，为此需要设置一个上限，超过投递次数就入死信队列。重试队列与延迟队列有相同的地方，都是需要设置延迟级别，它们彼此的区别是：延迟队列动作由内部触发，重试队列动作由外部消费端触发；延迟队列作用一次，而重试队列的作用范围会向后传递。

### 消费模式

消费模式分为推（push）模式和拉（pull）模式。推模式是指由Broker主动推送消息至消费端，实时性较好，不过需要一定的流制机制来确保服务端推送过来的消息不会压垮消费端。而拉模式是指消费端主动向Broker端请求拉取（一般是定时或者定量）消息，实时性较推模式差，但是可以根据自身的处理能力而控制拉取的消息量。

### 广播消费

消息一般有两种传递模式：点对点（P2P，Point-to-Point）模式和发布/订阅（Pub/Sub）模式。

对于点对点的模式而言，消息被消费以后，队列中不会再存储，所以消息消费者不可能消费到已经被消费的消息。虽然队列可以支持多个消费者，但是一条消息只会被一个消费者消费。

发布订阅模式定义了如何向一个内容节点发布和订阅消息，这个内容节点称为主题（topic），主题可以认为是消息传递的中介，消息发布者将消息发布到某个主题，而消息订阅者则从主题中订阅消息。主题使得消息的订阅者与消息的发布者互相保持独立，不需要进行接触即可保证消息的传递，发布/订阅模式在消息的一对多广播时采用。

RabbitMQ是一种典型的点对点模式，而Kafka是一种典型的发布订阅模式。但是RabbitMQ中可以通过设置交换器类型来实现发布订阅模式而达到广播消费的效果，Kafka 中也能以点对点的形式消费，你完全可以把其消费组（consumer group）的概念看成是队列的概念。不过对比来说，Kafka中因为有了消息回溯功能的存在，对于广播消费的力度支持比RabbitMQ的要强。

### 消息回溯

一般消息在消费完成之后就被处理了，之后再也不能消费到该条消息。消息回溯正好相反，是指消息在消费完成之后，还能消费到之前被消费掉的消息。对于消息而言，经常面临的问题是“消息丢失”，至于是真正由于消息中间件的缺陷丢失还是由于使用方的误用而丢失一般很难追查，如果消息中间件本身具备消息回溯功能的话，可以通过回溯消费复现“丢失的”消息进而查出问题的源头之所在。消息回溯的作用远不止与此，比如还有索引恢复、本地缓存重建，有些业务补偿方案也可以采用回溯的方式来实现。

### 消息堆积+持久化

流量削峰是消息中间件的一个非常重要的功能，而这个功能其实得益于其消息堆积能力。从某种意义上来讲，如果一个消息中间件不具备消息堆积的能力，那么就不能把它看做是一个合格的消息中间件。

消息堆积分内存式堆积和磁盘式堆积。

RabbitMQ 是典型的内存式堆积，但这并非绝对，在某些条件触发后会有换页动作来将内存中的消息换页到磁盘（换页动作会影响吞吐），或者直接使用惰性队列来将消息直接持久化至磁盘中。

Kafka 是一种典型的磁盘式堆积，所有的消息都存储在磁盘中。一般来说，磁盘的容量会比内存的容量要大得多，对于磁盘式的堆积其堆积能力就是整个磁盘的大小。从另外一个角度讲，消息堆积也为消息中间件提供了冗余存储的功能。援引 纽约时报的案例，其直接将 Kafka 用作存储系统。

### 消息追踪

对于分布式架构系统中的链路追踪（trace）而言，大家一定不会陌生。对于消息中间件而言，消息的链路追踪（以下简称消息追踪）同样重要。对于消息追踪最通俗的理解就是要知道消息从哪来，存在哪里以及发往哪里去。基于此功能下，我们可以对发送或者消费完的消息进行链路追踪服务，进而可以进行问题的快速定位与排查。

### 消息过滤

消息过滤是指按照既定的过滤规则为下游用户提供指定类别的消息。

就以 kafka 而言，完全可以将不同类别的消息发送至不同的 topic 中，由此可以实现某种意义的消息过滤，或者 Kafka 还可以根据分区对同一个 topic 中的消息进行分类。不过更加严格意义上的消息过滤应该是对既定的消息采取一定的方式按照一定的过滤规则进行过滤。

同样以 Kafka 为例，可以通过客户端提供的 ConsumerInterceptor 接口或者 Kafka Stream 的 filter 功能进行消息过滤。

### 多租户

也可以称为多重租赁技术，是一种软件架构技术，主要用来实现多用户的环境下公用相同的系统或程序组件，并且仍可以确保各用户间数据的隔离性。

RabbitMQ 就能够支持多租户技术，每一个租户表示为一个 vhost，其本质上是一个独立的小型 RabbitMQ 服务器，又有自己独立的队列、交换器及绑定关系等，并且它拥有自己独立的权限。vhost 就像是物理机中的虚拟机一样，它们在各个实例间提供逻辑上的分离，为不同程序安全保密地允许数据，它既能将同一个 RabbitMQ 中的众多客户区分开，又可以避免队列和交换器等命名冲突。

### 多协议支持

消息是信息的载体，为了让生产者和消费者都能理解所承载的信息（生产者需要知道如何构造消息，消费者需要知道如何解析消息），它们就需要按照一种统一的格式描述消息，这种统一的格式称之为消息协议。有效的消息一定具有某种格式，而没有格式的消息是没有意义的。一般消息层面的协议有 AMQP、MQTT、STOMP、XMPP 等（消息领域中的 JMS 更多的是一个规范而不是一个协议），支持的协议越多其应用范围就会越广，通用性越强，比如 RabbitMQ 能够支持 MQTT 协议就让其在物联网应用中获得一席之地。还有的消息中间件是基于其本身的私有协议运转的，典型的如 Kafka。

### 跨语言支持

对很多公司而言，其技术栈体系中会有多种编程语言，如 C/C++、JAVA、Go、PHP 等，消息中间件本身具备应用解耦的特性，如果能够进一步的支持多客户端语言，那么就可以将此特性的效能扩大。跨语言的支持力度也可以从侧面反映出一个消息中间件的流行程度。

### 流量控制

流量控制（flow control）针对的是发送方和接收方速度不匹配的问题，提供一种速度匹配服务抑制发送速率使接收方应用程序的读取速率与之相适应。通常的流控方法有 Stop-and-wait、滑动窗口以及令牌桶等。

### 消息顺序性

顾名思义，消息顺序性是指保证消息有序。这个功能有个很常见的应用场景就是CDC（Change Data Chapture），以MySQL为例，如果其传输的 binlog 的顺序出错，比如原本是先对一条数据加 1，然后再乘以 2，发送错序之后就变成了先乘以 2 后加 1 了，造成了数据不一致。

### 安全机制

在 Kafka 0.9 版本之后就开始增加了身份认证和权限控制两种安全机制。身份认证是指客户端与服务端连接进行身份认证，包括客户端与 Broker 之间、Broker 与 Broker 之间、Broker 与 ZooKeeper 之间的连接认证，目前支持 SSL、SASL 等认证机制。

权限控制是指对客户端的读写操作进行权限控制，包括对消息或 Kafka 集群操作权限控制。权限控制是可插拔的，并支持与外部的授权服务进行集成。对于 RabbitMQ 而言，其同样提供身份认证（TLS/SSL、SASL）和权限控制（读写操作）的安全机制。

### 消息幂等性

对于确保消息在生产者和消费者之间进行传输而言一般有三种传输保障（delivery guarantee）：

At most once，至多一次，消息可能丢失，但绝不会重复传输；

At least once，至少一次，消息绝不会丢，但是可能会重复；

Exactly once，精确一次，每条消息肯定会被传输一次且仅一次。

对于大多数消息中间件而言，一般只提供 At most once 和 At least once 两种传输保障，对于第三种一般很难做到，由此消息幂等性也很难保证。

Kafka 自 0.11 版本开始引入了幂等性和事务，Kafka 的幂等性是指单个生产者对于单分区单会话的幂等，而事务可以保证原子性地写入到多个分区，即写入到多个分区的消息要么全部成功，要么全部回滚，这两个功能加起来可以让 Kafka 具备 EOS（Exactly Once Semantic）的能力。

不过如果要考虑全局的幂等，还需要与从上下游方面综合考虑，即关联业务层面，幂等处理本身也是业务层面所需要考虑的重要议题。以下游消费者层面为例，有可能消费者消费完一条消息之后没有来得及确认消息就发生异常，等到恢复之后又得重新消费原来消费过的那条消息，那么这种类型的消息幂等是无法有消息中间件层面来保证的。如果要保证全局的幂等，需要引入更多的外部资源来保证，比如以订单号作为唯一性标识，并且在下游设置一个去重表。

### 事务性消息

事务本身是一个并不陌生的词汇，事务是由事务开始（Begin Transaction）和事务结束（End Transaction）之间执行的全体操作组成。支持事务的消息中间件并不在少数，Kafka 和 RabbitMQ 都支持，不过此两者的事务是指生产者发生消息的事务，要么发送成功，要么发送失败。消息中间件可以作为用来实现分布式事务的一种手段，但其本身并不提供全局分布式事务的功能。

下表是对 Kafka 与 RabbitMQ 功能的总结性对比及补充说明：







## 性能

功能维度是消息中间件选型中的一个重要的参考维度，但这并不是唯一的维度。有时候性能比功能还要重要，况且性能和功能很多时候是相悖的，鱼和熊掌不可兼得，Kafka 在开启幂等、事务功能的时候会使其性能降低，RabbitMQ 在开启 rabbitmq\_tracing 插件的时候也会极大的影响其性能。

消息中间件的性能一般是指其吞吐量，虽然从功能维度上来说，RabbitMQ 的优势要大于 Kafka，但是 Kafka 的吞吐量要比 RabbitMQ 高出 1 至 2 个数量级，一般 RabbitMQ 的单机 QPS 在万级别之内，而 Kafka 的单机 QPS 可以维持在十万级别，甚至可以达到百万级。

消息中间件的吞吐量始终会受到硬件层面的限制。就以网卡带宽为例，如果单机单网卡的带宽为 1Gbps，如果要达到百万级的吞吐，那么消息体大小不得超过 (1Gb/8)/100W，即约等于 134B，换句话说如果消息体大小超过 134B，那么就不可能达到百万级别的吞吐。这种计算方式同样可以适用于内存和磁盘。

时延作为性能维度的一个重要指标，却往往在消息中间件领域所被忽视，因为一般使用消息中间件的场景对时效性的要求并不是很高，如果要求时效性完全可以采用 RPC 的方式实现。消息中间件具备消息堆积的能力，消息堆积越大也就意味着端到端的时延也就越长，与此同时延时队列也是某些消息中间件的一大特色。那么为什么还要关注消息中间件的时延问题呢？消息中间件能够解耦系统，对于一个时延较低的消息中间件而言，它可以让上游生产者发送消息之后可以迅速的返回，也可以让消费者更加快速的获取到消息，在没有堆积的情况下可以让整体上下游的应用之间的级联动作更加高效，虽然不建议在时效性很高的场景下使用消息中间件，但是如果所使用的消息中间件的时延方面比较优秀，那么对于整体系统的性能将会是一个不小的提升。

## 可靠性 + 可用性

消息丢失是使用消息中间件时所不得不面对的一个同点，其背后消息可靠性也是衡量消息中间件好坏的一个关键因素。尤其是在金融支付领域，消息可靠性尤为重要。然而说到可靠性必然要说到可用性，注意这两者之间的区别，消息中间件的可靠性是指对消息不丢失的保障程度；而消息中间件的可用性是指无故障运行的时间百分比，通常用几个 9 来衡量。

从狭义的角度来说，分布式系统架构是一致性协议理论的应用实现，对于消息可靠性和可用性而言也可以追溯到消息中间件背后的一致性协议。对于 Kafka 而言，其采用的是类似 PacificA 的一致性协议，通过 ISR（In-Sync-Replica）来保证多副本之间的同步，并且支持强一致性语义（通过 acks 实现）。对应的 RabbitMQ 是通过镜像环形队列实现多副本及强一致性语义的。多副本可以保证在 master 节点宕机异常之后可以提升 slave 作为新的 master 而继续提供服务来保障可用性。Kafka 设计之初是为日志处理而生，给人们留下了数据可靠性要求不要的不良印象，但是随着版本的升级优化，其可靠性得到极大的增强，详细可以参考 KIP101。就目前而言，在金融支付领域使用 RabbitMQ 居多，而在日志处理、大数据等方面 Kafka 使用居多，随着 RabbitMQ 性能的不断提升和 Kafka 可靠性的进一步增强，相信彼此都能在以前不擅长的领域分得一杯羹。

同步刷盘是增强一个组件可靠性的有效方式，消息中间件也不例外，Kafka 和 RabbitMQ 都可以支持同步刷盘，但是笔者对同步刷盘有一定的疑问：绝大多数情景下，一个组件的可靠性不应该由同步刷盘这种极其损耗性能的操作来保障，而是采用多副本的机制来保证。

这里还要提及的一个方面是扩展能力，这里我狭隘地将此归纳到可用性这一维度，消息中间件的扩展能力能够增强其用可用能力及范围，比如前面提到的 RabbitMQ 支持多种消息协议，这个就是基于其插件化的扩展实现。还有从集群部署上来讲，归功于 Kafka 的水平扩展能力，其基本上可以达到线性容量提升的水平，在 LinkedIn 实践介绍中就提及了有部署超过千台设备的 Kafka 集群。

## 运维管理

在消息中间件的使用过程中难免会出现各式各样的异常情况，有客户端的，也有服务端的，那么怎样及时有效的进行监测及修复。业务线流量有峰值又低谷，尤其是电商领域，那么怎样前进行有效的容量评估，尤其是大促期间？脚踢电源、网线被挖等事件层出不穷，如何有效的做好异地多活？这些都离不开消息中间件的衍生产品——运维管理。

运维管理也可以进行进一步的细分，比如：申请、审核、监控、告警、管理、容灾、部署等。

申请、审核很好理解，在源头对资源进行管控，既可以进行有效校正应用方的使用规范，配和监控也可以做好流量统计与流量评估工作，一般申请、审核与公司内部系统交融性较大，不适合使用开源类的产品。

监控、告警也比较好理解，对消息中间件的使用进行全方位的监控，即可以为系统提供基准数据，也可以在检测到异常的情况配合告警，以便运维、开发人员的迅速介入。除了一般的监控项（比如硬件、GC 等）之外，对于消息中间件还需要关注端到端时延、消息审计、消息堆积等方面。对于 RabbitMQ 而言，最正统的监控管理工具莫过于 rabbitmq\_management 插件了，但是社区内还有 AppDynamics, Collectd, DataDog, Ganglia, Munin, Nagios, New Relic, Prometheus, Zenoss 等多种优秀的产品。Kafka 在此方面也毫不逊色，比如：Kafka Manager, Kafka Monitor, Kafka Offset Monitor, Burrow, Chaperone, Confluent Control Center 等产品，尤其是 Cruise 还可以提供自动化运维的功能。

不管是扩容、降级、版本升级、集群节点部署、还是故障处理都离不开管理工具的应用，一个配套完备的管理工具集可以在遇到变更时做到事半功倍。故障可大可小，一般是一些应用异常，也可以是机器掉电、网络异常、磁盘损坏等单机故障，这些故障单机房内的多副本足以应付。如果是机房故障就要涉及异地容灾了，关键点在于如何有效的进行数据复制，对于 Kafka 而言，可以参考 MirrorMarker、uReplicator 等产品，而 RabbitMQ 可以参考 Federation 和 Shovel。

## 社区力度及生态发展

对于目前流行的编程语言而言，如 Java、Python，如果你在使用过程中遇到了一些异常，基本上可以通过搜索引擎的帮助来得到解决，因为一个产品用的人越多，踩过的坑也就越多，对应的解决方案也就越多。对于消息中间件也同样适用，如果你选择了一种“生僻”的消息中间件，可能在某些方面运用的得心应手，但是版本更新缓慢、遇到棘手问题也难以得到社区的支持而越陷越深；相反如果你选择了一种“流行”的消息中间件，其更新力度大，不仅可以迅速的弥补之前的不足，而且也能顺应技术的快速发展来变更一些新的功能，这样可以让你以“站在巨人的肩膀上”。在运维管理维度我们提及了 Kafka 和 RabbitMQ 都有一系列开源的监控管理产品，这些正是得益于其社区及生态的迅猛发展。

## 误区总结

在进行消息中间件选型之前可以先问自己一个问题：是否真的需要一个消息中间件？在搞清楚这个问题之后，还可以继续问自己一个问题：是否需要自己维护一套消息中间件？很多初创型公司为了节省成本会选择直接购买消息中间件有关的云服务，自己只需要关注收发消息即可，其余的都可以外包出去。

很多人面对消息中间件时会有一种自研的冲动，你完全可以对 Java 中的 ArrayBlockingQueue 做一个简单的封装，你也可以基于文件、数据库、Redis 等底层存储封装而形成一个消息中间件。消息中间件做为一个基础组件并没有想象中的那么简单，其背后还需要配套的管理运维整个生态的产品集。自研还有会交接问题，如果文档不齐全、运作不规范将会带给新人噩梦般的体验。是否真的有自研的必要？如果不是 KPI 的压迫可以先考虑下这 2 个问题：1. 目前市面上的消息中间件是否都真的无法满足目前业务需求？ 2. 团队是否有足够的能力、人力、财力、精力来支持自研？

很多人在做消息中间件选型时会参考网络上的很多对比类的文章，但是其专业性、严谨性、以及其政治立场问题都有待考证，需要带着怀疑的态度去审视这些文章。比如有些文章会在没有任何限定条件及场景的情况下直接定义某款消息中间件最好，还有些文章没有指明消息中间件版本及测试环境就来做功能和性能对比分析，诸如此类的文章都可以唾弃之。

消息中间件犹如小马过河，选择合适的才最重要，这需要贴合自身的业务需求，技术服务于业务，大体上可以根据上一节所提及的功能、性能等 6 个维度来一一进行筛选。更深层次的抉择在于你能否掌握其魂，笔者鄙见：RabbitMQ 在于 routing，而 Kafka 在于 streaming，了解其根本对于自己能够对症下药选择到合适的消息中间件尤为重要。

消息中间件选型切忌一味的追求性能或者功能，性能可以优化，功能可以二次开发。如果要在功能和性能方面做一个抉择的话，那么首选性能，因为总体上来说性能优化的空间没有功能扩展的空间大。然而对于长期发展而言，生态又比性能以及功能都要重要。

很多时候，对于可靠性方面也容易存在一个误区：想要找到一个产品来保证消息的绝对可靠，很不幸的是这世界上没有绝对的东西，只能说尽量趋于完美。想要尽可能的保障消息的可靠性也并非单单只靠消息中间件本身，还要依赖于上下游，需要从生产端、服务端和消费端这 3 个维度去努力保证，《RabbitMQ 消息可靠性分析》这篇文章就从这 3 个维度去分析了 RabbitMQ 的可靠性。

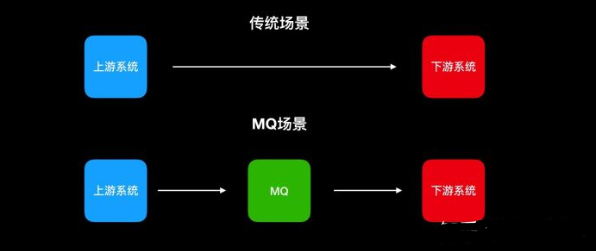
消息中间件选型还有一个考量标准就是尽量贴合团队自身的技术栈体系，虽然说没有蹩脚的消息中间件只有蹩脚的程序员，但是让一个 C 栈的团队去深挖 PhxQueue 总比去深挖 Scala 编写的 Kafka 要容易的多。

消息中间件大道至简：一发一存一消费，没有最好的消息中间件，只有最合适的消息中间件。

# 应用

基于上文所述的特点，那么MQ就衍生出了中的使用场景，在大型的系统中，应用非常广泛，这里我们就列举一下常见的应用场景。

## 应用解耦（异步）



系统之间进行数据交互的时候，在时效性和稳定性之间我们都需要进行选择。基于线程的异步处理，能确保用户体验，但是极端情况下可能会出现异常，影响系统的稳定性，而同步调用很多时候无法保证理想的性能，那么我们就可以用MQ来进行处理。上游系统将数据投递到MQ，下游系统取MQ的数据进行消费，投递和消费可以用同步的方式处理，因为MQ接收数据的性能是非常高的，不会影响上游系统的性能，那么下游系统的及时率能保证吗？当然可以，不然就不会有下面的一个应用场景。

## 通知

这里就用到了前文一个重要的特点，发布订阅，下游系统一直在监听MQ的数据，如果MQ有数据，下游系统则会按照先进先出这样的规则，逐条进行消费，而上游系统只需要将数据存入MQ里，这样就既降低了不同系统之间的耦合度，同时也确保了消息通知的及时性，而且也不影响上游系统的性能。

## 限流

上文有说了一个非常重要的特性，MQ数据是只有一条数据在使用中。 在很多存在并发，而又对数据一致性要求高，而且对性能要求也高的场景，如何保证，那么MQ就能起这个作用了。不管多少流量进来，MQ都会让你遵守规则，排除处理，不会因为其他原因，导致并发的问题，而出现很多意想不到脏数据。

## 数据分发

MQ的发布订阅肯定不是只是简单的一对一，一个上游和一个下游的关系，MQ中间件基本都是支持一对多或者广播的模式，而且都可以根据规则选择分发的对象。这样上游的一份数据，众多下游系统中，可以根据规则选择是否接收这些数据，这样扩展性就很强了。 PS:上文中的上游和下游，在MQ更多的是叫做生产者（producer）和消费者（consumer）。

## 分布式事务

分布式事务是我们开发中一直尽量避免的一个技术点，但是，现在越来越多的系统是基于微服务架构开发，那么分布式事务成为必须要面对的难题，解决分布式事务有一个比较容易理解的方案，就是二次提交。基于MQ的特点，MQ作为二次提交的中间节点，负责存储请求数据，在失败的情况可以进行多次尝试，或者基于MQ中的队列数据进行回滚操作，是一个既能保证性能，又能保证业务一致性的方案，当然，这个方案的主要问题就是定制化较多，有一定的开发工作量。

## 应用示例

为了更加直观的展示MQ的应用场景，这里我们就用一个常见的电商系统中的几个业务，来具体说明下MQ在实际开发中应用场景。

我们的实际场景大概是一个基于微服务架构的电商系统，分为用户微服务、商品微服务、订单微服务、促销微服务等。基于微服务模式开发的系统，MQ的使用场景更多，下面我们逐一说明：

1. 注册后我们可能需要做很多初始化的操作，如：调用邮件服务器发送邮件、调用促销服务赠送优惠劵、下发用户数据到客户关系系统等。那么这时候我们将这些操作去监听MQ，当用户注册成功过后，通过MQ通知其他业务进行操作。确保注册用户的性能。
2. 后台发布商品的时候，商品数据需要从数据库中转换成搜索引擎数据（基于elasticsearch），那么我们应该将商品写入数据库后，再写入到MQ，然后通过监听MQ来生成elasticsearch对应的数据。
3. 用户下单后，24小时未支付，需要取消订单。以前我们可能是定时任务循环查询，然后取消订单。实际上，我更推荐类似延迟MQ的方式，避免了很多无效的数据库查询，将一个MQ设置为24小时后才让消费者消费掉，这样很大程度上能减轻服务器压力。
4. 支付完成后，需要及时的通知子系统（进销存系统发货，用户服务积分，发送短信）进行下一步操作，但是，支付回调我们都是需要保证高性能的，所以，我应该直接修改数据库状态，存入MQ，让MQ通知子系统做其他非实时的业务操作。这样能保证核心业务的高效及时。

## 注意事项

其实，还有非常多的业务场景，是可以考虑用MQ方式的，但是很多时候，也会存在滥用的情况，我们需要清楚认识我们的业务场景： 发验证码短信、邮件，这种过分依赖外部，而且时效性可以接收几十秒延迟的，其实更好的方式是多线程异步处理，而不是过多依赖MQ。 秒杀抢购确保库存不为负数，更多的依赖高性能缓存（如redis），以及强制加锁，千万不要依赖消费者最终的返回结果。（实际工作中已经看到好几个这样的案例了）上游-下游 这种直接的处理方式效率肯定是比 上游-MQ-下游 方式要高，MQ效率高，是因为，我只是上游-MQ 这个阶段就当做已经成功了。