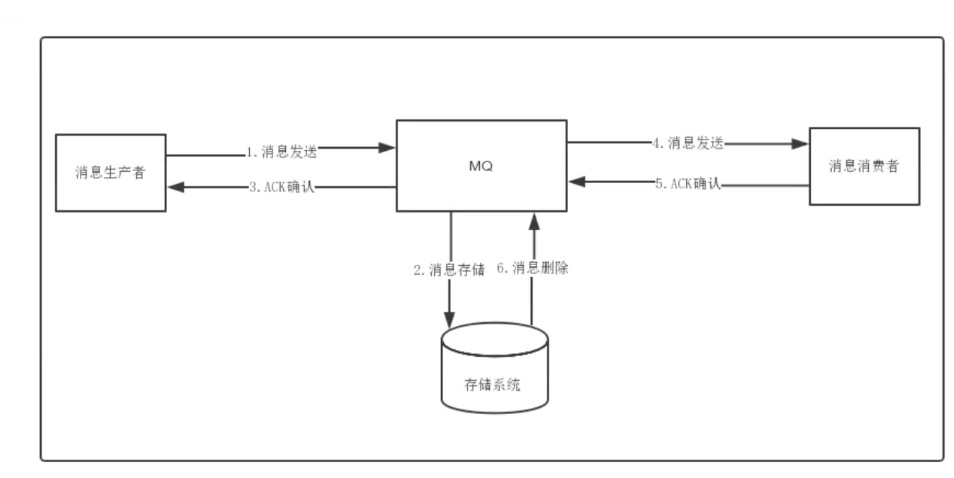
**消息存储**

分布式队列因为有高可靠性的要求，所以数据要进行持久化存储。



1. 消息生产者发送消息
2. mq收到消息，将消息进行持久化，在存储中增加一条记录
3. 返回ack 给生产者
4. mq push 消息给对应的消费者，然后等待消费者返回ack
5. 如果消息消费者在指定时间内成功返回ack，那么mq认为消息消费成功，在存储中删除消息，即执行第6 部；如果mq在指定时间内没有收到ack，则认为消息消费失败，会重新尝试push消息，重复执行4、5、6步骤。
6. mq删除消息

**存储介质**

**关系型数据库db**

apache下开源的另一款mq activemq 默认采用kahadb 做消息存储， 可选用jdbc 的方式来做消息持久化，通过简单的xml配置信息即可实现jdbc消息存储。 由于关系型数据库（mysql）在单表数据量达到千万级别的情况下，其io读写性能旺旺会出现瓶颈。 在可靠性方面，这种方案非常依赖DB，如果一旦DB出现故障，则mq的消息就无法裸盘存储会导致线上故障。

不适合消息量大的情况，不适合互联网

**文件系统存储**

目前业界较为常见的几款产品（rocketmq，kafka，rabbitmq） 均采用的是消息刷盘至所部属的物理机文件系统来做持久化（刷盘一般可以分为异步刷盘和同步刷盘）。消息刷盘为消息存储提供了一种高效率、高可靠和高性能的数据持久方式。除非部署mq机器本身或者本地磁盘挂了，否则一般不会出现无法持久化的故障问题。

**性能对比**

文件系统存储 > 关系型数据库存储

**消息的存储和发送**

**消息存储：** 磁盘如果使用得当，磁盘的速度完全可以匹配上网络的数据传输速度。 目前高性能的磁盘，顺序写速度可以达到600MB/s,超过了一般网卡的传输速度。但是磁盘随机写的书读大概只有100KB/s，和顺序写的性能相差6000倍。因此有如此巨大的速度差别，好的消息队列系统会比普通的消息队列系统速度多个数量级。rocketmq 的消息用顺序写，保证了消息存储的速度。

**消息发送：** linux 系统纷纷为用户态和内核态，文件操作、网络操作需要涉及这两种形态的切换，免不了进行数据复制。一台服务器吧本机磁盘文件的内容发送到客户端，一般分为两个步骤：

1. read；读取本地文件内容
2. write： 将读取的内容通过网络发送出去。

这两个看似见到的操作，实际执行了4次数据复制，分别是：

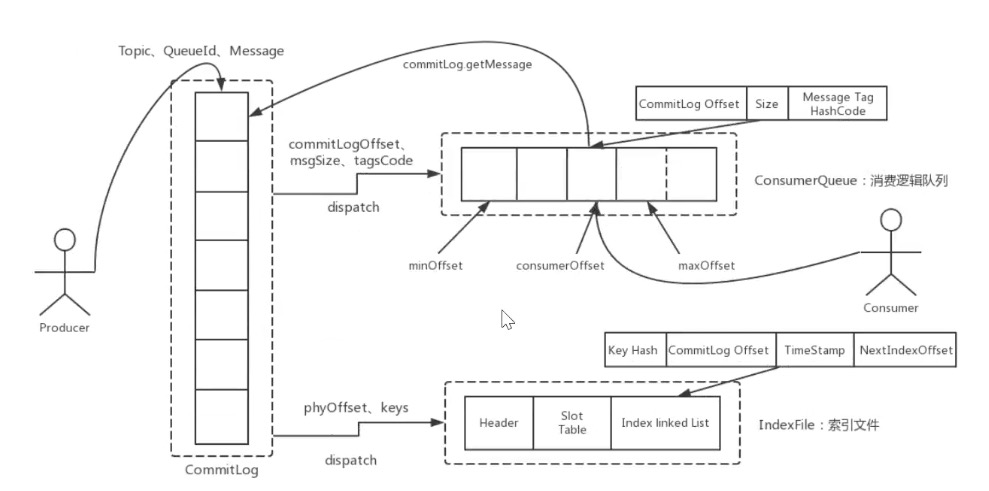
1. 从磁盘复制数据到内核态内存
2. 2. 从内核态内存复制到用户态内存；
3. 然后从用户态内存复制到网络驱动的内核态内存；
4. 最后是从网络驱动的内核态内存复制到网卡中进行传输。

通过使用mmap的方式，可以省去想用户态的内存复制，提高速度。这种机制在java中通过mappedbytebuffer实现的rocketmq 充分利用了上述特性。 也就是所谓的零拷贝技术。提高消息存盘的网络发送的速度。

这里需要主题的是，采用mappedbytebuffer这种内存映射的方式有几个限制，其中之一是一次只能映射1.5G到2G的文件至用户态的虚拟内存，这也是为何rocketmq 默认设置单个commitLog 日志数据问文件为1G的原因。

**消息存储结构**

rocketmq 消息的存储是有comsuequeue和commitlog配合完成的，消息真正的物理存储文件是commitLog，consumequeue是消息的逻辑队列，类似数据可的索引文件，存世的是指向物理内存的地址。 每个topic下的每个messagequeue都有一个对应的comsumequeue文件



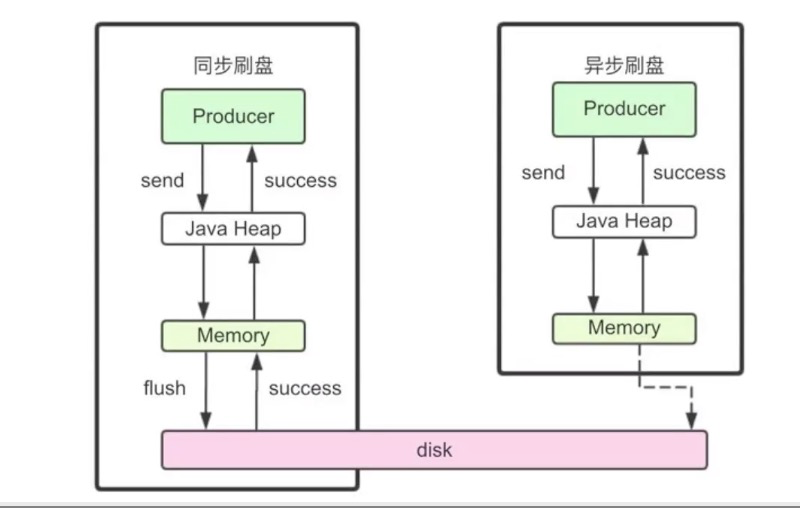
commitLog：存储消息的元数据，数据

consumerqueue： 存储消息在commitLog的索引。一个topic 下面会有多个messagequeue，这个跟messagequeue 是一一对应的；如果丢失了，可以通过commitlog 找回来。

indexFile： 为了消息查询提供了一种通过key或者时间区间来查询消息的方法。 这种通过indexfile来查找消息的方法不影响发送与消费信息的主流程。

**刷盘机制**

rocketmq 的消息是存储到磁盘上的，这样既能保证断电后恢复，有可以让存储的消息量超过内存的限制。 rocketmq 为了提高性能，会尽可能地保证磁盘的顺序写，消息在通过producer 写入rocketmq的时候，有两种写磁盘的方式，分别是同步刷盘和异步刷盘



同步刷盘：

在返回写成功状态时，消息已经被写入磁盘。 具体流程是，消息写入内存的pagecache后，立刻通知刷盘线程刷盘，然后等待刷盘完成，刷盘线程执行完成后唤醒等待线程，返回消息写成功的状态。 保证消息的可靠性

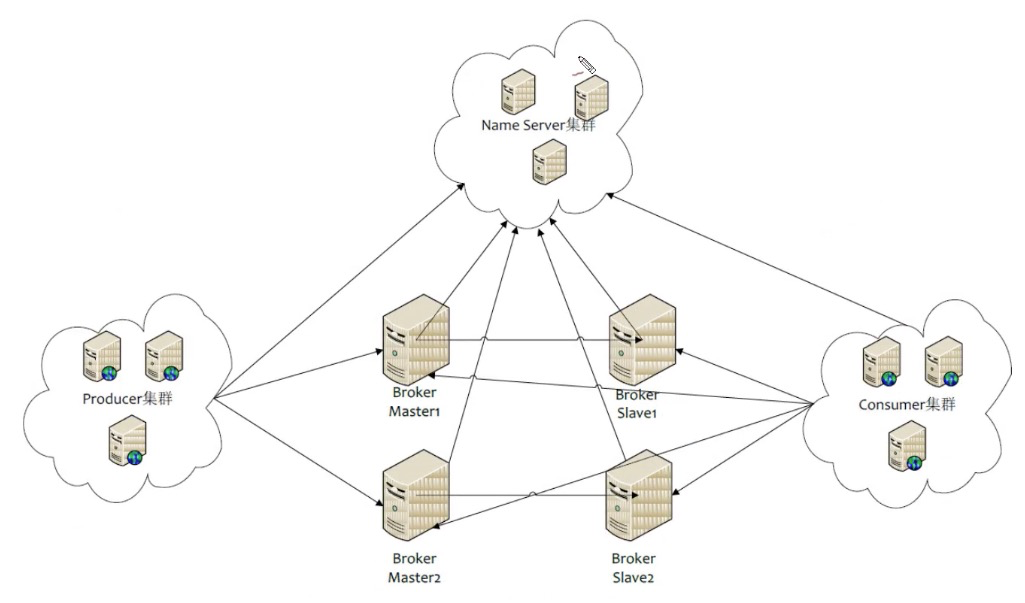
异步刷盘：

在返回写成功状态时，消息可能只是被写入了内存的pagecache，写操作的返回快，吞吐量大；当内存里的消息量积累到一定程度时，统一出发写磁盘操作，快速写入。

配置：

在broker的配置文件中配置。

**高可用性机制**



namesrv 是无状态的，当添加一个namesrc，broker 会自动上报自己的信息。

rocketmq 分布式集群是通过master和slave的配合达到高可用性的。

master和slave的却别是： 在broker的配置文件中，参数brokerid 为0 表示这个broker 是master，非0 表示这个broker 是slave，同时brokerrole 参数也会说明这个broker是master还是slave。

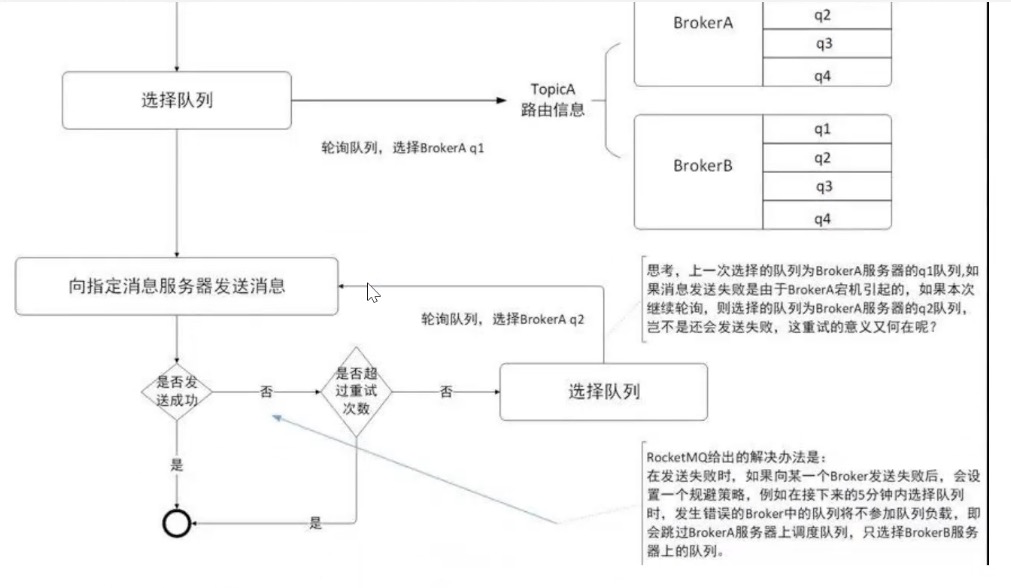
master 角色的broker 只是读和写，slave 仅仅支持度，也就是producer 只能和master 角色的broker 连接写入消息，consumer可以连接master或者slave

**消息消费高可用**

在consumer 的配置文件中，并不需要设置是从master读还是slave 读，当master不可用或者繁忙时，consumer会被自动切换到从slave读。 有了自动切换consumer这种机制，当一个master角色的机器出现故障后，consumer仍然可以从slave读取消息，不影响consumer程序。 这就达到了消费端的高可用。

**消息发送高可用**

在创建topic的时候，把topic的多个messagequeue 创建在多个broker组上（相同broker名称，不通brokerid的机器组成一个broker组），这样当一个broker组的master不可用后，其他组的master仍然可用，producer扔然可以发送消息。rocketmq目前还不支持slave自动转成master，如果机器资源不足，需要把slave转成master，则要手动停止slave角色的broker，更改配置文件，用写的配置文件启动broker。



**消息主从复制**

如果一个broker组有master和slave，消息需要从master复制到slave上，有同步和异步复制两种方式。

同步复制

同步复制方式是等master和slave 均写成功后才反馈给客户端写成功状态。

在同步复制方式下，如果master出现故障，slave上有全部的备份数据，容易恢复，但是同步复制会增加数据写入谈事，降低系统吞吐量。

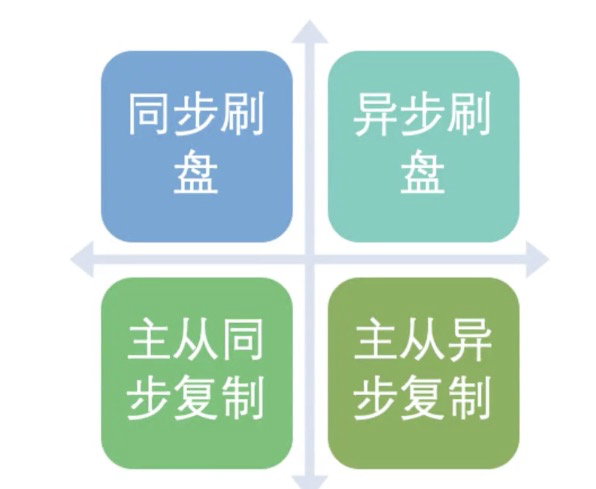
异步复制

异步复制当时是只要master写成功，即可反馈给客户端写成功状态。

在与不复制方式下，系统拥有较低的延时和较高的吞吐量。但是如果master除了故障，有些数据因为没有被写入slave有可能会丢失。

配置：

通过broker 里面的brokerrole 参数进行设置。

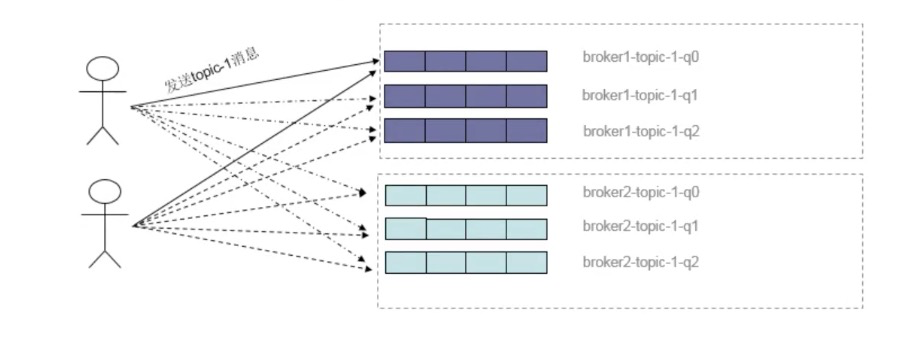


在实际应用中要结合业务场景，合理设置刷盘方式和主从复制方式，尤其是sync\_flush方式，由于频繁触发磁盘写操作，会明显降低性能。通常情况下，应该把master和slave配置成Async\_FLUSH的刷盘方式。只从之间配置成SYNC\_master 复制方式，这样即使有一台机器出故障，仍然能保证数据不丢，是个不错的选择。

**负载均衡**

**producer负载均衡**

pruducer端，每个势力在发消息的时候，默认会轮询所有messagequeue发送，以达到让消息平均落在不通的queue上。 而由于queue可以散落在不同的broker，所以消息就发送到不通的broker下。



体重箭头线条上的标号代表顺序，发布方会把第一条消息发送至queue 0，然后把第二条消息发送到queue 1，一次类推

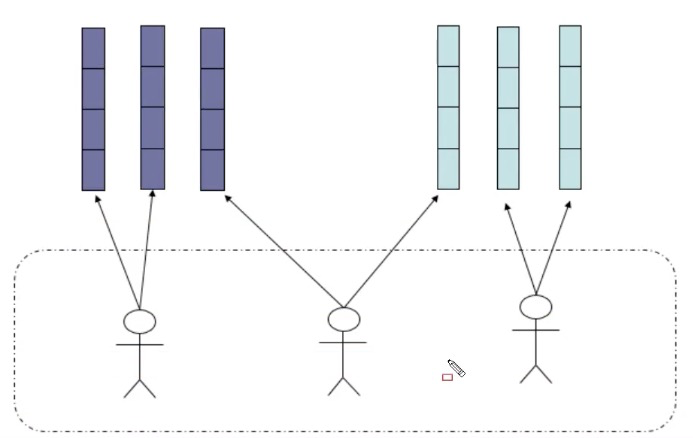
**consumer负载均衡**

**集群消费模式**

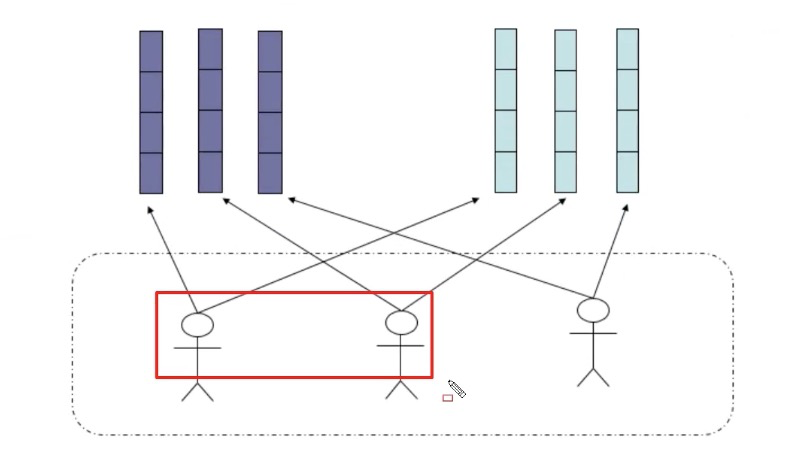
在集群消费模式(也就是负载均衡)下，每条消息只需要投递到订阅这个topic的consumer group下的每一个实例即可。 rocketmq 采用注定拉取的方式消费消息，在拉取的时候需要明确指定拉取哪一条message queue。

而当每个实例的数量有变更，都会触发一次所有实例的负载均衡，这时候会按照queue的数量和实例的数量平均分配queue给每个实例。

默认的分配算法是Allocatemessagequeueaveragely



还有一种平均的算法是allocatemessagequeueAveragelyByCircle，也就是平均分摊每一条queue，只是一换装轮流分queue的形式：



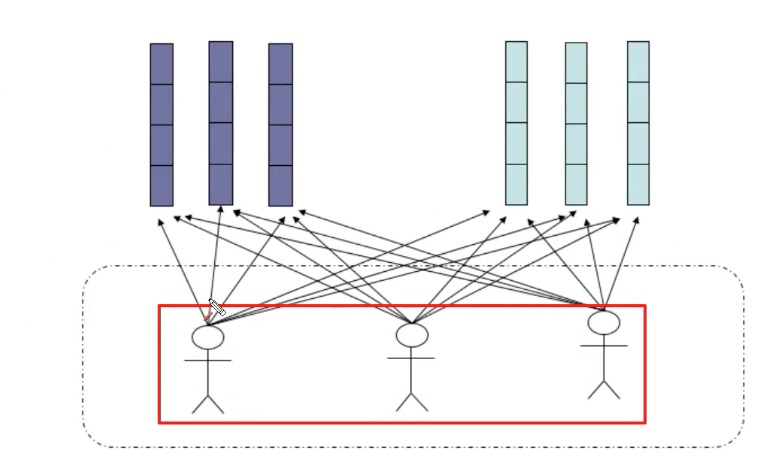
需要注意的是，集群模式下，queue都是只允许分配至一个实例，这是由于如果多个实例同时消费一个queue的消息，由于拉取哪些消息是consumer主动控制的，那样会导致同一个消息在不通的实例下被消费多次，所以算法上都是一个queue只分给一个consumer实例，一个consumer实例可以允许同时分到不同的queue。

通过增加consumer实例去分摊queue的消费，可以起到水平扩展的消费能力的作用。而有实例下线的时候，会重新出发负载均衡，这时候原来分配到queue分配到其他实例上继续消费。

但是如果consumer 实例的数量比messagequeue 的总数还要多的话，多出来的consumer实例将无法分到queue，也就是无法消费到消息，也就无法起到分摊负载的作用了。 所以需要控制让queue的总数大于consumer的数量。

**广播模式**

由于广播模式下要求一条消息投递到一个消费组下面所有的消费者实例，所以也就没有消息被分摊消费的说法。在实现上，其中一个不同就是在consumer在分配queue的时候，所有consumer都分到所有的queue



**消息重试**

**顺序消息重试**

对于顺序消息，当消费者消费消息失败后，消息队列rocketmq 会自动不断进行消息重试，每次间隔为1秒，这是应用汇出现消息消费被阻塞的情况。 因此，在使用顺序消息时，务必保证应用能够及时监控并处理消费失败的情况，避免阻塞现象的发生。

**无序消息重试**

对于无序消息（普通，定时、延时、事务消息），当消费者消费消息失败时，可以通过设置返回状态达到消息重试的结果。 无序消息的重试只针对集群消费方式生效，广播方式不提供失败重试特性，即消费失败后，失败消息不再重试，继续消费新的消息。

重试次数

消息队列rocketmq 默认允许每条消息最多重试16 次，每次重试的时间间隔如下。



如果消息重试16次后仍然失败，消息将不再投递，如果沿河按照上述重试时间间隔计算，某条消息在一直消费失败的前提下，将会在接下来的4小时46分钟之内重试16次，超过这个时间消息将不再重试投递

一条消息无论重试多少次，这些重试消息的messageid 都不会改变

**配置方式**

**消费失败后，重试方式配置**

集群消费方式下，消息消费失败后期望消息重试，需要在消息监听接口的实现中明确进行配置（三种方式任选一种）

返回 action.reconsumerlater

返回null

跑出异常

**消费失败后，不充实的配置方式**

集群消费方式下，消息失败后期望消息不充实，需要补货消费逻辑中可能抛出的异常，最终返回action.commitmessage,此后，这条消息将不会被重试

**自定义消息最大重试次数**

消息队列rocketmq允许consumer启动的时候设置最大的重试次数，重试时间间隔将按照如下策略：

最大重试次数小于等于16，则重试时间间隔同上表描述

超过16，时间间隔均为2小时

消息最大重试次数的设置，对相同groupid 下的所有consumer实例有效。

如果只对相同groupid下两个consumer实例中的其中一个设置了maxreconsumetimes，那么该配置对两个consumer实例均生效。

配置采用覆盖的房方式生效，即最后启动的consumer实例会覆盖之前的启动实例

**获取消息重试次数**

**死信队列**

当一条消息初次消费失败，消息队列rocketmq会自动进行消息重试；达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确的消费改消息，此时，消息队列rocketmq不会立刻将消息丢弃，而是将其发送到改消费者对应的特殊队列中。

在消息队列rocketmq 中，这种正常情况下无法被消费的消息称为死信消息（dead-letter mesage），存储死信消息的特殊队列称为死信队列

**死信消息具有以下特性。**

不会再被消费者正常消费

有效期与正常消息相同，均为3天，3天后会被自动删除。 因此，请在死信消息产生后的3天内及时处理。

**死信队列具有以下特性：**

一个死信队列对应一个groupid，而不是对应单个消费者实例

如果一个group id 未产生死信消息，消息队列rocketmq不会为其创建相应的死信队列

一个死信队列包含了对应groupid产生的所有死信消息，不论改消息属于哪个topic

**查看死信信息**

1. 在控制台查询
2. 在消息界面根据主题查询死信消息
3. 选择重新发送消息

一条消息进入死信队列，意味着某些因素导致消费者无法正常消费改消息。 因此，通常需要您对其进行特殊处理，排查可以因素并解决问题后，可以在消息队列rocketmq控制台重新发送该消息，让消费者重新消费一次。

**消费幂等**

消息队列rocketmq 消费者在接收到消息以后，有必要根据业务上的唯一key对消息做幂等处理的必要性

**消费幂等的必要性**

在互联网应用中，尤其在网络不稳定的情况下，消息队列rocketmq的消息有可能会出现重复，这个重复简单可以概括为一下情况：

1. 发送时消息重复

当一条消息已被成功发送到服务端并完成持久化，此时出现了网络闪断或者客户端宕机，导致服务端对客户端应答失败。如果此时生产者意识到消息发送失败并尝试再次发送消息，消费者后续会受到两条内容完全相同并且messageid 也相同的消息

1. 投递时消息重复

消息消费的场景下，消息已投递到消费者并完成业务处理，当客户端给服务端反馈应答的时候网络闪断。 为了保证消息至少被消费一次，消息队列rocketmq的服务端将在网络恢复后再次尝试投递之前已被处理过的消息。消费者后续会受到两条内容相同并且messageid也相同的消息。

1. 负载均衡时消息重复（包括但不限于网络抖动，broker重启以及订阅方应用重启）

当消息队列rocketmq的broker 或者客户端重启、扩容、缩容时，会触发rebalance，此时消费者可能会受到重复消息

**处理方式**

因为message id 有可能出现冲突（重读）的 情况，所以真正安全的幂等处理，不建议以messageid 作为处理依据。 最好的方式是以业务唯一表示作为幂等处理的关键依据，而业务的唯一表示可以通过消息key 进行设置：