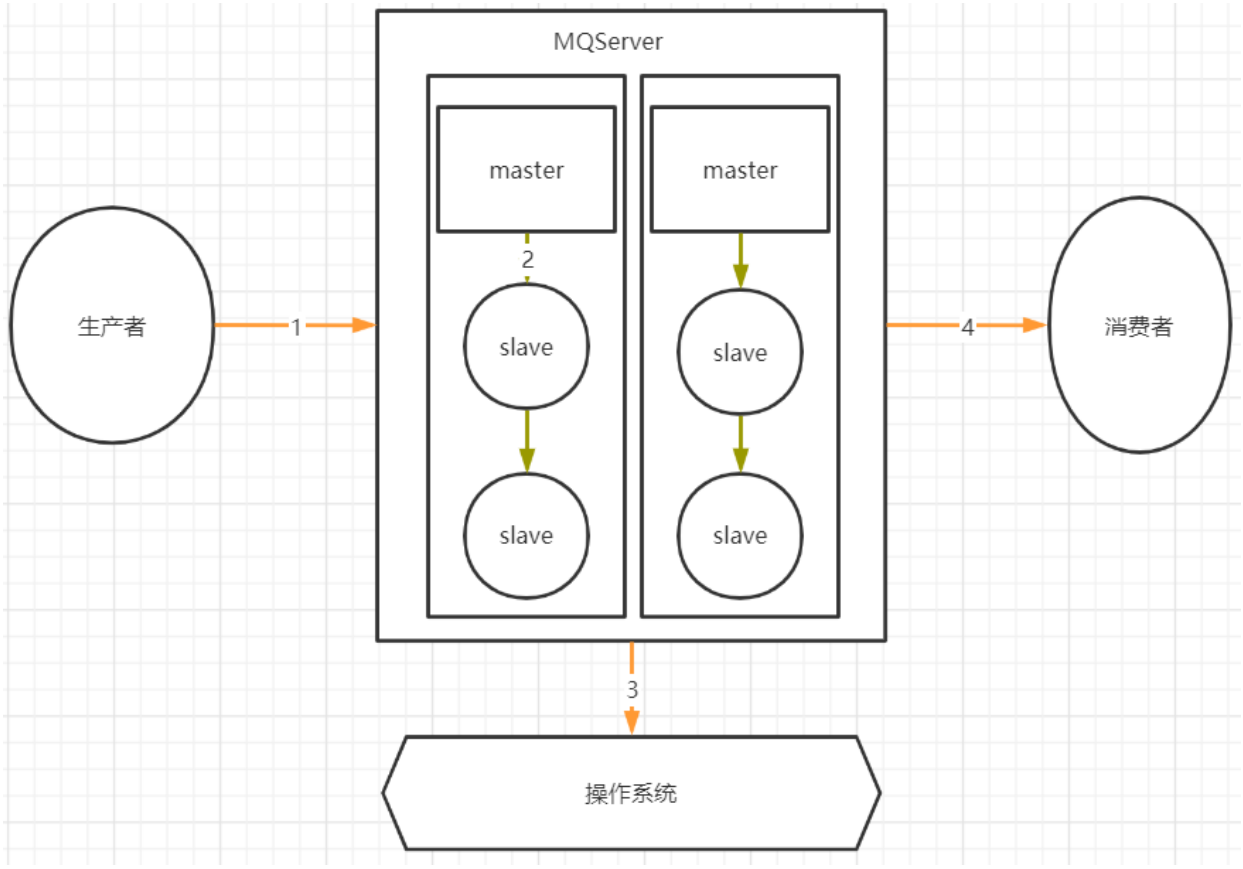
图灵:楼兰
RocketMQ生产环境常见问题分析
你的神秘技术宝藏

一、RocketMQ如何保证消息不丢失

这个是在面试时,关于MQ,面试官最喜欢问的问题。这个问题是所有MQ都需要面对的一个共性问题。大致的解决思路都是一致的,但是针对不同的MQ产品又有不同的解决方案。分析这个问题要从以下几个角度入手:

1、哪些环节会有丢消息的可能?

我们考虑一个通用的MQ场景:



其中，1，2，4三个场景都是跨网络的，而跨网络就肯定会有丢消息的可能。

然后关于3这个环节，通常MQ存盘时都会先写入操作系统的缓存page cache中，然后再由操作系统异步的将消息写入硬盘。这个中间有个时间差，就可能会造成消息丢失。如果服务挂了，缓存中还没有来得及写入硬盘的消息就会丢失。

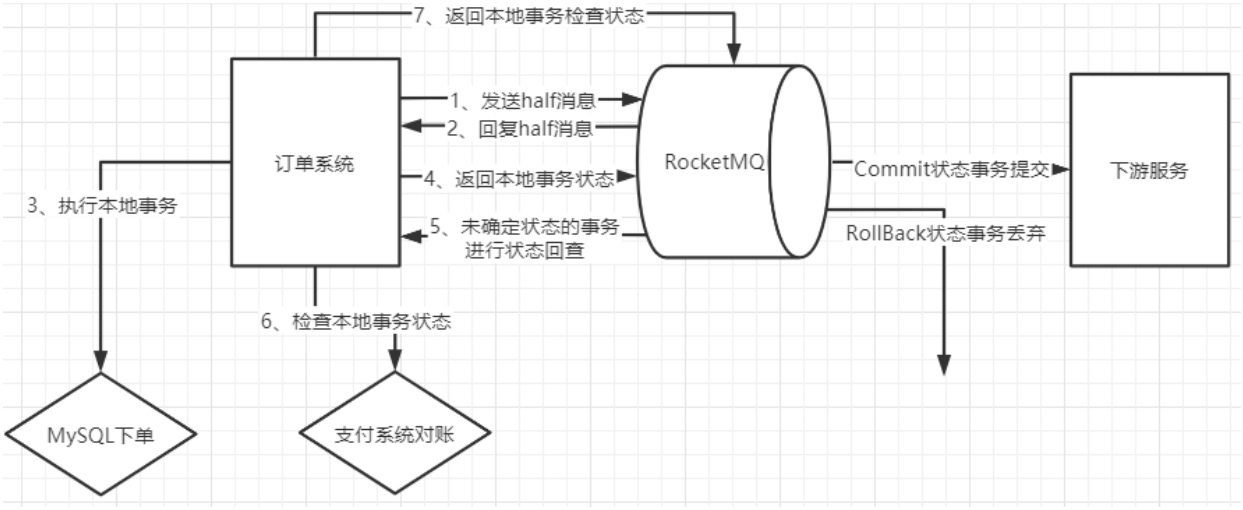
这个是MQ场景都会面对的通用的丢消息问题。那我们看看用RocketMQ时要如何解决这个问题

2、RocketMQ消息零丢失方案

1》 生产者使用事务消息机制保证消息零丢失

这个结论比较容易理解，因为RocketMQ的事务消息机制就是为了保证零丢失来设计的，并且经过阿里的验证，肯定是非常靠谱的。

但是如果深入一点的话，我们还是要理解下这个事务消息到底是不是靠谱。我们以最常见的电商订单场景为例，来简单分析下事务消息机制如何保证消息不丢失。我们看下下面这个流程图：



1、为什么要发送个half消息？有什么用?

这个half消息是在订单系统进行下单操作前发送，并且对下游服务的消费者是不可见的。那这个消息的作用更多的体现在确认RocketMQ的服务是否正常。相当于嗅探下RocketMQ服务是否正常，并且通知RocketMQ，我马上就要发一个很重要的消息了，你做好准备。

2.half消息如果写入失败了怎么办?

如果没有half消息这个流程，那我们通常是会在订单系统中先完成下单，再发送消息给MQ。这时候写入消息到MQ如果失败就会非常尴尬了。而half消息如果写入失败，我们就可以认为MQ的服务是有问题的，这时，就不能通知下游服务了。我们可以在下单时给订单一个状态标记，然后等待MQ服务正常后再进行补偿操作，等MQ服务正常后重新下单通知下游服务。

3.订单系统写数据库失败了怎么办?

这个问题我们同样比较下没有使用事务消息机制时会怎么办？如果没有使用事务消息，我们只能判断下单失败，抛出了异常，那就不往MQ发消息了，这样至少保证不会对下游服务进行错误的通知。但是这样的话，如果过一段时间数据库恢复过来了，这个消息就无法再次发送了。当然，也可以设计另外的补偿机制，例如将订单数据缓存起来，再启动一个线程定时尝试往数据库写。而如果使用事务消息机制，就可以有一种更优雅的方案。

如果下单时，写数据库失败(可能是数据库崩了，需要等一段时间才能恢复)。那我们可以另外找个地方把订单消息先缓存起来(Redis、文本或者其他方式)，然后给RocketMQ返回一个UNKNOWN状态。这样RocketMQ就会过一段时间来回查事务状态。我们就可以在回查事务状态时再尝试把订单数据写入数据库，如果数据库这时候已经恢复了，那就能完整正常的下单，再继续后面的业务。这样这个订单的消息就不会因为数据库临时崩了而丢失。

4.half消息写入成功后RocketMQ挂了怎么办?

我们需要注意下，在事务消息的处理机制中，未知状态的事务状态回查是由RocketMQ的Broker主动发起的。也就是说如果出现了这种情况，那RocketMQ就不会回调到事务消息中回查事务状态的服务。这时，我们就可以将订单一直标记为"新下单"的状态。而等RocketMQ恢复后，只要存储的消息没有丢失，RocketMQ就会再次继续状态回查的流程。

5.下单成功后如何优雅的等待支付成功?

在订单场景下，通常会要求下单完成后，客户在一定时间内，例如10分钟，内完成订单支付，支付完成后才会通知下游服务进行进一步的营销补偿。

如果不用事务消息，那通常会怎么办?

最简单的方式是启动一个定时任务，每隔一段时间扫描订单表，比对未支付的订单的下单时间，将超过时间的订单回收。这种方式显然是有很大问题的，需要定时扫描很庞大的一个订单信息，这对系统是个不小的压力。

那更进一步的方案是什么呢？是不是就可以使用RocketMQ提供的延迟消息机制。往MQ发一个延迟1分钟的消息，消费到这个消息后去检查订单的支付状态，如果订单已经支付，就往下游发送下单的通知。而如果没有支付，就再发一个延迟1分钟的消息。最终在第十个消息时把订单回收。这个方案就不用对全部的订单表进行扫描，而只需要每次处理一个单独的订单消息。

那如果使用上了事务消息呢？我们就可以用事务消息的状态回查机制来替代定时的任务。在下单时，给Broker返回一个UNKNOWN的未知状态。而在状态回查的方法中去查询订单的支付状态。这样整个业务逻辑就会简单很多。我们只需要配置RocketMQ中的事务消息回查次数(默认15次)和事务回查间隔时间(messageDelayLevel)，就可以更优雅的完成这个支付状态检查的需求。

6、事务消息机制的作用

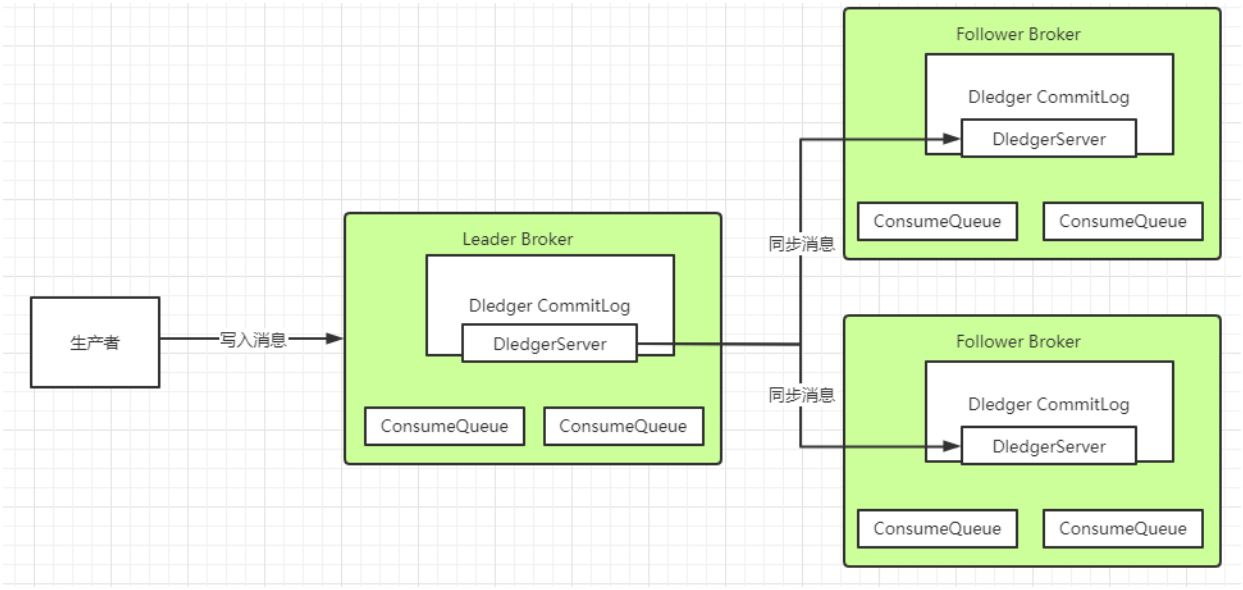
整体来说，在订单这个场景下，消息不丢失的问题实际上就还是转化成了下单这个业务与下游服务的业务的分布式事务一致性问题。而事务一致性问题一直以来都是一个非常复杂的问题。而RocketMQ的事务消息机制，实际上只保证了整个事务消息的一半，他保证的是订单系统下单和发消息这两个事件的事务一致性，而对下游服务的事务并没有保证。但是即便如此，也是分布式事务的一个很好的降级方案。目前来看，也是业内最好的降级方案。

2》RocketMQ配置同步刷盘+Dledger主从架构保证MQ主从同步时不会丢消息

1、同步刷盘

这个从我们之前的分析，就很好理解了。我们可以简单的把RocketMQ的刷盘方式 flushDiskType配置成同步刷盘就可以保证消息在刷盘过程中不会丢失了。

2、Dledger的文件同步



在使用Dledger技术搭建的RocketMQ集群中，Dledger会通过两阶段提交的方式保证文件在主从之间成功同步。

Dledger是由开源组织OpenMessage带入到RocketMQ中的一种高可用集群方案。Dledger的主要作用有两个，一是进行Broker自动选主。二是接管Broker的CommitLog文件写入过程。将单机的文件写入，转为基于多数同意机制的分布式消息写入。

简单来说，数据同步会通过两个阶段，一个是uncommitted阶段，一个是commited阶段。

Leader Broker上的Dledger收到一条数据后，会标记为uncommitted状态，然后他通过自己的DledgerServer组件把这个uncommitted数据发给Follower Broker的DledgerServer组件。

接着Follower Broker的DledgerServer收到uncommitted消息之后，必须返回一个ack给Leader Broker的Dledger。然后如果Leader Broker收到超过半数的Follower Broker返回的ack之后，就会把消息标记为committed状态。

再接下来， Leader Broker上的DledgerServer就会发送committed消息给Follower Broker上的DledgerServer，让他们把消息也标记为committed状态。这样，就基于Raft协议完成了两阶段的数据同步。

另外，在实现层面，Dledger还有很多细节机制，保证消息的安全性。

3》消费者端不要使用异步消费机制

正常情况下，消费者端都是需要先处理本地事务，然后再给MQ一个ACK响应，这时MQ就会修改Offset，将消息标记为已消费，从而不再往其他消费者推送消息。所以在Broker的这种重新推送机制下，消息是不会在传输过程中丢失的。但是也会有下面这种情况会造成服务端消息丢失：

```
 DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("please_rename_unique_group_name_4");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,

 ConsumeConcurrentlyContext context) {
                new Thread(){
                    public void run(){
                        //处理业务逻辑
                        System.out.printf("%s Receive New Messages: %s %n",
Thread.currentThread().getName(), msgs);
                    }
                };
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
```

这种异步消费的方式，就有可能造成消息状态返回后消费者本地业务逻辑处理失败造成消息丢失的可能。

4》RocketMQ特有的问题，NameServer挂了如何保证消息不丢失?

NameServer在RocketMQ中，是扮演的一个路由中心的角色，提供到Broker的路由功能。但是其实路由中心这样的功能，在所有的MQ中都是需要的。kafka是用zookeeper和一个作为Controller的Broker一起来提供路由服务，整个功能是相当复杂纠结的。而RabbitMQ是由每一个Broker来提供路由服务。而只有RocketMQ把这个路由中心单独抽取了出来，并独立部署。

这个NameServer之前都了解过，集群中任意多的节点挂掉，都不会影响他提供的路由功能。那**如果集群中所有的NameServer节点都挂了呢?**

有很多人就会认为在生产者和消费者中都会有全部路由信息的缓存副本，那整个服务可以正常工作一段时间。其实这个问题大家可以做一下实验，当NameServer全部挂了后，生产者和消费者是立即就无法工作了的。至于为什么，可以去源码中找找答案。

那再回到我们的消息不丢失的问题，在这种情况下，RocketMQ相当于整个服务都不可用了，那他本身肯定无法给我们保证消息不丢失了。我们只能自己设计一个降级方案来处理这个问题了。例如在订单系统中，如果多次尝试发送RocketMQ不成功，那就只能另外找给地方(Redis、文件或者内存等)把订单消息缓存下来，然后起一个线程定时的扫描这些失败的订单消息，尝试往RocketMQ发送。这样等RocketMQ的服务恢复过来后，就能第一时间把这些消息重新发送出去。整个这套降级的机制，在大型互联网项目中，都是必须要有的。

5》RocketMQ消息零丢失方案总结

完整分析过后，整个RocketMQ消息零丢失的方案其实挺简单

- 生产者使用事务消息机制。
- Broker配置同步刷盘+Dledger主从架构
- 消费者不要使用异步消费。
- 整个MQ挂了之后准备降级方案

那这套方案是不是就很完美呢？其实很明显，这整套的消息零丢失方案，在各个环节都大量的降低了系统的处理性能以及吞吐量。在很多场景下，这套方案带来的性能损失的代价可能远远大于部分消息丢失的代价。所以，我们在设计RocketMQ使用方案时，要根据实际的业务情况来考虑。例如，如果针对所有服务器都在同一个机房的场景，完全可以把Broker配置成异步刷盘来提升吞吐量。而在有些对消息可靠性要求没有那么高的场景，在生产者端就可以采用其他一些更简单的方案来提升吞吐，而采用定时对账、补偿的机制来提高消息的可靠性。而如果消费者不需要进行消息存盘，那使用异步消费的机制带来的性能提升也是非常显著的。

总之，这套消息零丢失方案的总结是为了在设计RocketMQ使用方案时的一个很好的参考。

二、使用RocketMQ如何快速处理积压消息?

1、如何确定RocketMQ有大量的消息积压?

在正常情况下，使用MQ都会要尽量保证他的消息生产速度和消费速度整体上是平衡的，但是如果部分消费者系统出现故障，就会造成大量的消息积累。这类问题通常在实际工作中会出现得比较隐蔽。例如某一天一个数据库突然挂了，大家大概率就会集中处理数据库的问题。等好不容易把数据库恢复过来了，这时基于这个数据库服务的消费者程序就会积累大量的消息。或者网络波动等情况，也会导致消息大量的积累。这在一些大型的互联网项目中，消息积压的速度是相当恐怖的。所以消息积压是个需要时时关注的问题。

对于消息积压，如果是RocketMQ或者kafka还好，他们的消息积压不会对性能造成很大的影响。而如果是RabbitMQ的话，那就惨了，大量的消息积压可以瞬间造成性能直线下滑。

对于RocketMQ来说，有个最简单的方式来确定消息是否有积压。那就是使用web控制台，就能直接看到消息的积压情况。

在Web控制台的主题页面，可以通过 Consumer管理 按钮实时看到消息的积压情况。

另外，也可以通过mqadmin指令在后台检查各个Topic的消息延迟情况。

还有RocketMQ也会在他的 ${storePathRootDir}/config 目录下落地一系列的json文件，也可以用来跟踪消息积压情况。

2、如何处理大量积压的消息?

首先，如果消息不是很重要，那么RocketMQ的管理控制台就直接提供了跳过堆积的功能。

然后，如果消息很重要，又确实是因为消费者端处理能力不不够，造成消息大量积压，那么我们可以设计出一套消息转移的方案。

如果Topic下的MessageQueue配置得是足够多的，那每个Consumer实际上会分配多个MessageQueue来进行消费。这个时候，就可以简单的通过增加Consumer的服务节点数量来加快消息的消费，等积压消息消费完了，再恢复成正常情况。最极限的情况是把Consumer的节点个数设置成跟MessageQueue的个数相同。但是如果此时再继续增加Consumer的服务节点就没有用了。

而如果Topic下的MessageQueue配置得不够多的话，那就不能用上面这种增加Consumer节点个数的方法了。这时怎么办呢？ 这时如果要快速处理积压的消息，可以创建一个新的Topic，配置足够多的MessageQueue。然后把所有消费者节点的目标Topic转向新的Topic，并紧急上线一组新的消费者，只负责消费旧Topic中的消息，并转储到新的Topic中，这个速度是可以很快的。然后在新的Topic上，就可以通过增加消费者个数来提高消费速度了。之后再根据情况恢复成正常情况。

> 在官网中，还分析了一个特殊的情况。就是如果RocketMQ原本是采用的普通方式搭建主从架构，而现在想要中途改为使用Dledger高可用集群，这时候如果不想历史消息丢失，就需要先将消息进行对齐，也就是要消费者把所有的消息都消费完，再来切换主从架构。因为Dledger集群会接管RocketMQ原有的CommitLog日志，所以切换主从架构时，如果有消息没有消费完，这些消息是存在旧的CommitLog中的，就无法再进行消费了。这个场景下也是需要尽快的处理掉积压的消息。

三、打开RocketMQ的消息轨迹功能

RocketMQ默认提供了消息轨迹的功能，这个功能在排查问题时是非常有用的。

1、RocketMQ消息轨迹数据的关键属性：

Producer端	Consumer端	Broker端
生产实例信息	消费实例信息	消息的Topic
发送消息时间	投递时间,投递轮次	消息存储位置
消息是否发送成功	消息是否消费成功	消息的Key值
发送耗时	消费耗时	消息的Tag值

2、消息轨迹配置

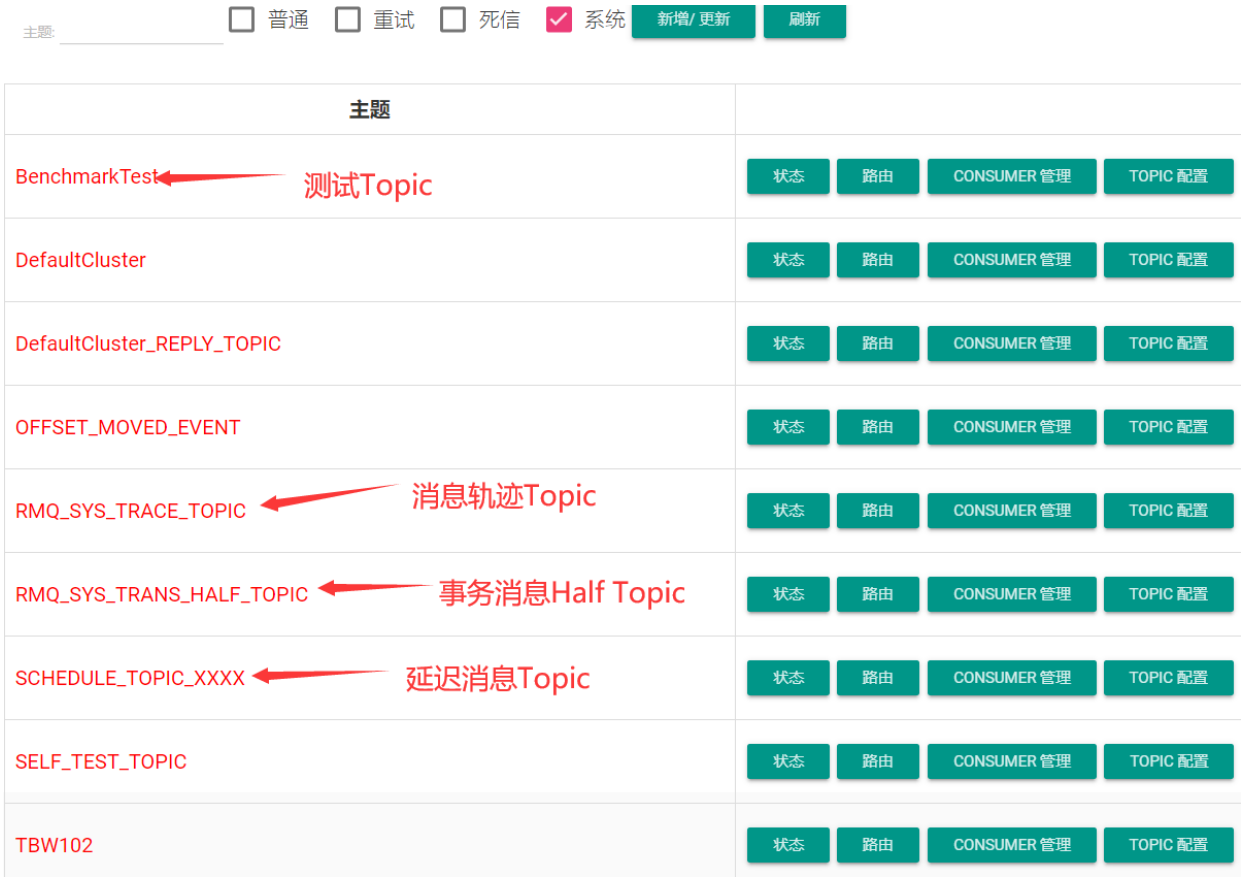
打开消息轨迹功能，需要在broker.conf中打开一个关键配置：

```
traceTopicEnable=true
```

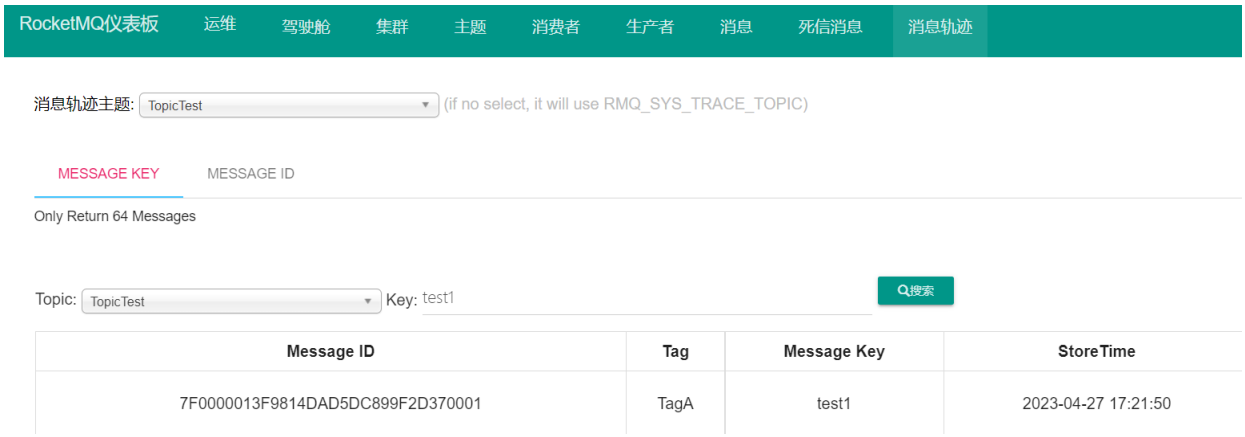
这个配置的默认值是false。也就是说默认是关闭的。

3、消息轨迹数据存储

默认情况下，消息轨迹数据是存于一个系统级别的Topic ,RMQ_SYS_TRACE_TOPIC。这个Topic在Broker节点启动时，会自动创建出来。



打开消息轨迹后，在管理控制台就可以根据Key或者MessageId，查询消息在RocketMQ内的处理过程。



另外，也支持客户端自定义轨迹数据存储的Topic。

在客户端的两个核心对象 DefaultMQProducer和DefaultMQPushConsumer，他们的构造函数中，都有两个可选的参数来打开消息轨迹存储

- **enableMsgTrace**：是否打开消息轨迹。默认是false。
- **customizedTraceTopic**：配置将消息轨迹数据存储到用户指定的Topic 。

四、RocketMQ服务器配置优化

1 JVM选项

推荐使用最新发布的JDK 1.8版本。通过设置相同的Xms和Xmx值来防止JVM调整堆大小以获得更好的性能。简单的JVM配置如下所示：

```
-server -Xms8g -Xmx8g -Xmn4g
```

如果您不关心RocketMQ Broker的启动时间，还有一种更好的选择，就是通过"预触摸"Java堆以确保在JVM初始化期间每个页面都将被分配。那些不关心启动时间的人可以启用它： -XX:+AlwaysPreTouch禁用偏置锁定可能会减少JVM暂停， -XX:-UseBiasedLocking。至于垃圾回收，建议使用带JDK 1.8的G1收集器。

```
-XX:+UseG1GC -XX:G1HeapRegionSize=16m
-XX:G1ReservePercent=25
-XX:InitiatingHeapOccupancyPercent=30
```

这些GC选项看起来有点激进，但事实证明它在我们的生产环境中具有良好的性能。另外不要把-XX:MaxGCPauseMillis的值设置太小，否则JVM将使用一个小的年轻代来实现这个目标，这将导致非常频繁的minor GC，所以建议使用rolling GC日志文件：

```
-XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFiles=5
-XX:GCLogFileSize=30m
```

如果写入GC文件会增加代理的延迟，可以考虑将GC日志文件重定向到内存文件系统：

```
-Xloggc:/dev/shm/mq_gc_%p.log123
```

2 Linux内核参数

RocketMQ的启动脚本中提供了一个os.sh脚本，在这个脚本中列出了许多建议的内核参数，可以进行微小的更改然后用于生产用途。下面的参数需要注意，更多细节请参考/proc/sys/vm/*的文档

- **vm.extra_free_kbytes**，告诉VM在后台回收（kswapd）启动的阈值与直接回收（通过分配进程）的阈值之间保留额外的可用内存。RocketMQ使用此参数来避免内存分配中的长延迟。（与具体内核版本相关）
- **vm.min_free_kbytes**，如果将其设置为低于1024KB，将会巧妙的将系统破坏，并且系统在高负载下容易出现死锁。
- **vm.max_map_count**，限制一个进程可能具有的最大内存映射区域数。RocketMQ将使用mmap加载CommitLog和ConsumeQueue，因此建议将为此参数设置较大的值。（agressiveness --> aggressiveness）
- **vm.swappiness**，定义内核交换内存页面的积极程度。较高的值会增加攻击性，较低的值会减少交换量。建议将值设置为10来避免交换延迟。
- **File descriptor limits**，RocketMQ需要为文件（CommitLog和ConsumeQueue）和网络连接打开文件描述符。我们建议设置文件描述符的值为655350。
- Disk scheduler，RocketMQ建议使用I/O截止时间调度器，它试图为请求提供有保证的延迟。

> 有道云笔记链接：https://note.youdao.com/s/NCReOZ6y