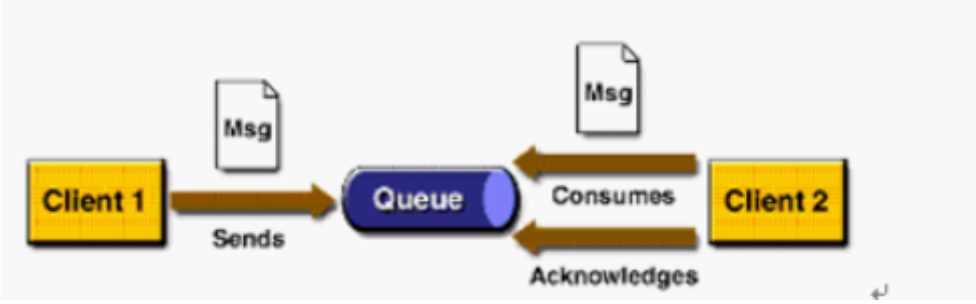
一、概述

1.JMS

jms(Java Messaging Service，Java消息服务)是Java平台上有关面向消息中间件（MOM）的技术规范，允许应用程序组建基于JavaEE平台创建、发送、接收和读取消息。它使分布式通讯耦合度更低，消息服务更加可靠以及异步性。

在JMS标准中，有两种消息模型P2Peye.com（Point to Point）和Publish/Subscribe（Pub/Sub）。

1.1 P2P模式

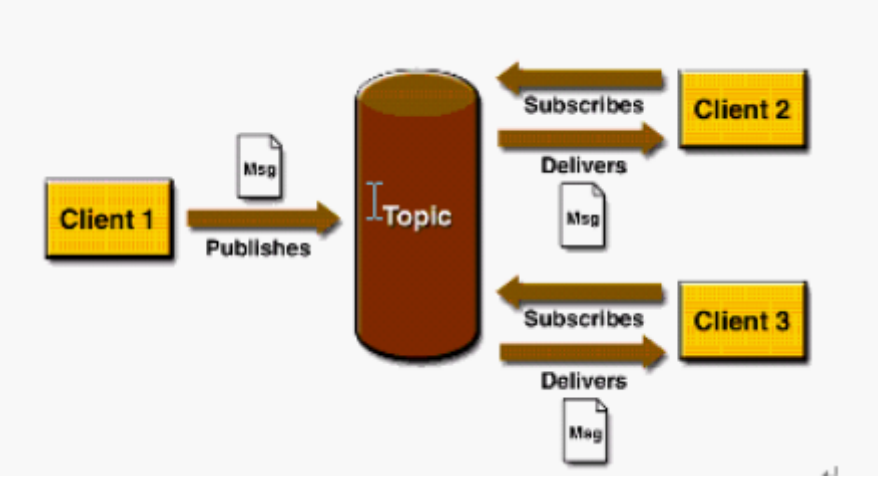


P2P模式包含三个角色：消息队列（Queue），发送者（Sender），接受者（Receiver）。每个消息都被发送到一个特定的队列，接受者从队列中获取消息。队列保留着消息，直到他们被消费或超时。

P2P的特点：

- 每个消息只有一个消费者（Consumer）（即一旦被消费，消息就不在消息队列中）
- 发送者和接受者在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有运行，它不会影响到消息被发送到队列
- 接收者在成功接收消息之后需向队列应答成功
- 如果希望发送的每个消息都被成功处理的话，那么需要P2P模式。

1.2Pub/Sub模式



包含三个角色：主题（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：

- 每个消息可以有多个消费者
- 发布者和订阅者之间有时间上的依赖性。针对某个主题（Topic）的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息，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不被任何处理、或者只被一个消费者处理、或者可以被多个消费者处理的话，可以采用Pub/Sub模型。

1.3消息消费

在JMS中，消息的生产和消费都是异步的。对于消费来说，JMS的消费者可以通过两种方式消费消息。

- 同步：订阅者或接收者通过receive方法来接收消息，receive方法在接收到消息之前（或超时之前）将一直阻塞。
- 异步：订阅者和接收者可以注册成为一个消息监听器。当消息到达之后，系统自动调用监听器的onMessage方法。

JNDI：Java命名和目录接口，是一种标准的 Java命名系统接口。可以在网络上查找和访问服务。通过制定一个资源名称，该系统对应于数据库或命名服务中的一个记录，同时返回资源连接建立所必须的信息。

JNDI在JMS中起到查找和访问发送目标或消息来源的作用。

2.JMX

JMX（Java Management Extensions，即Java管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架

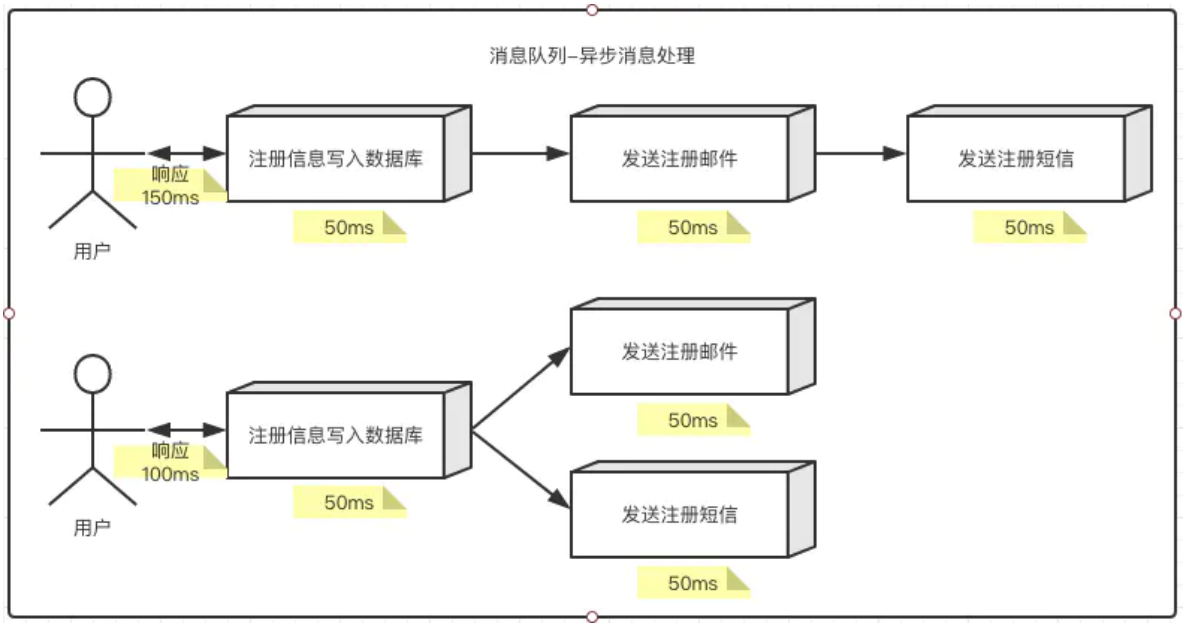
二、消息队列应用场景

消息中间件是分布式系统中重要的组件，主要实现异步消息，应用解耦，流量削峰以及消息通讯等功能。

1.异步处理

以用户注册，并且需要注册邮件和短信为例。

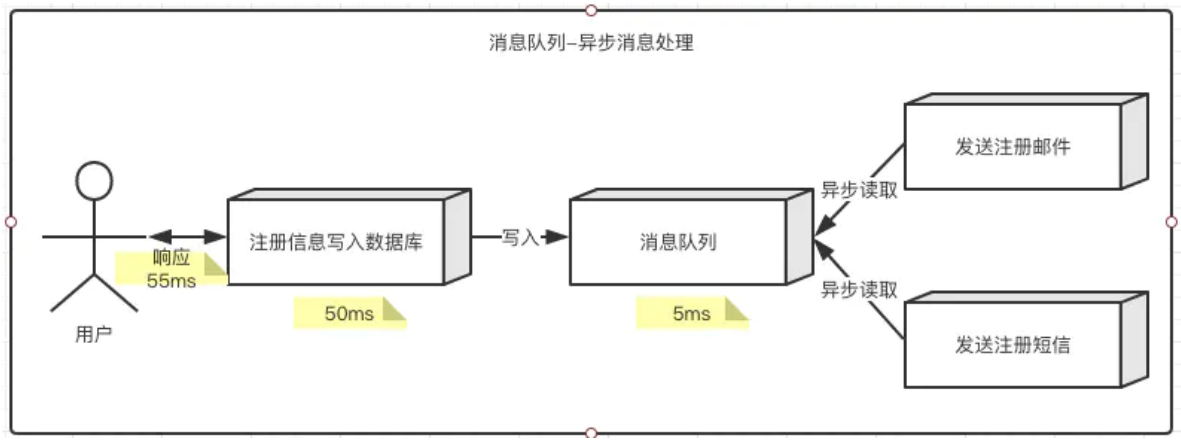
用户注册后，需要发送注册邮件和注册短信。传统方法有两种：串行和并行方式。如下图：



1）串行方式：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，返回客户端。

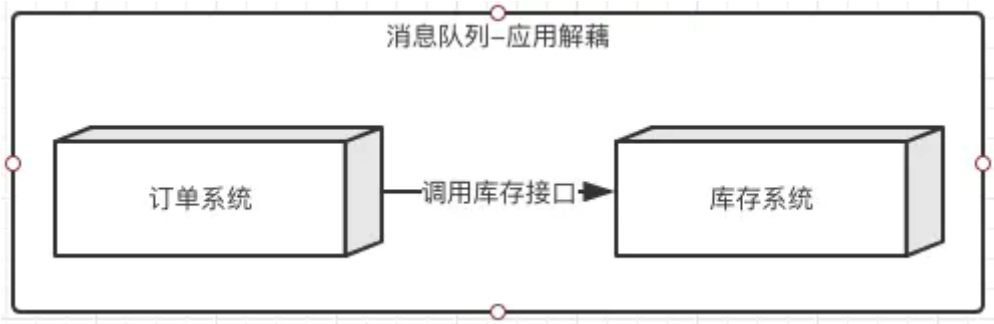
2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间。

引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



2.应用解耦

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，如下图：

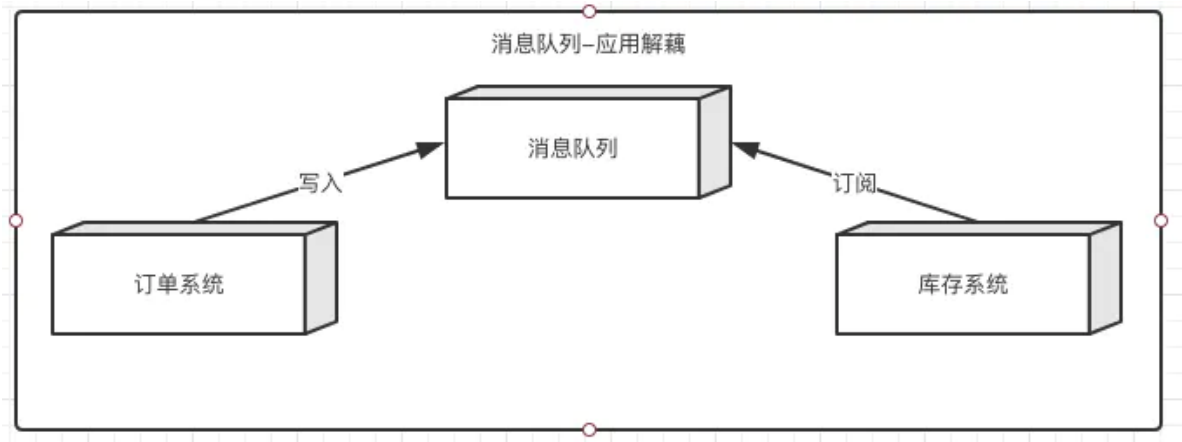


传统模式的缺点：

1）假如库存系统无法访问，则订单减库存将失败，从而导致订单失败。

2）订单系统与库存系统耦合。

引入消息队列后的方案：



1）订单系统：用户下单后，订单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。

2）库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息进行库存操作。

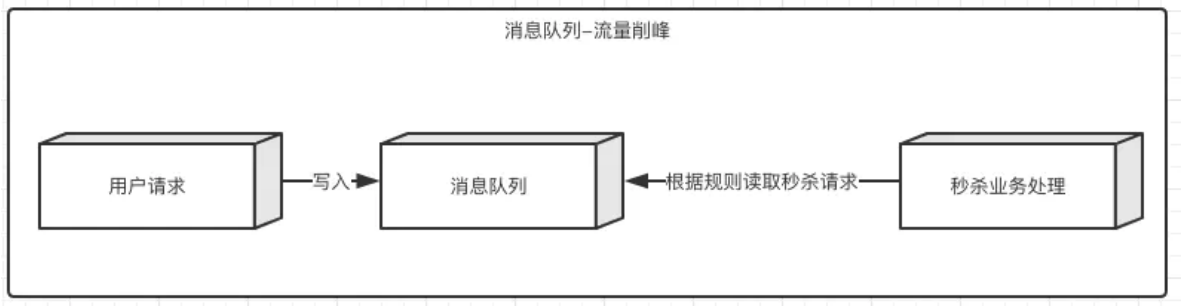
假如：下单时库存系统不能正常使用，也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

3.流量削峰

流量削峰也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛。秒刷活动，一般会因为流量过大，导致流量暴增，应用挂掉。为了解决这个问题，需要再应用前端加入消息队列。

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。

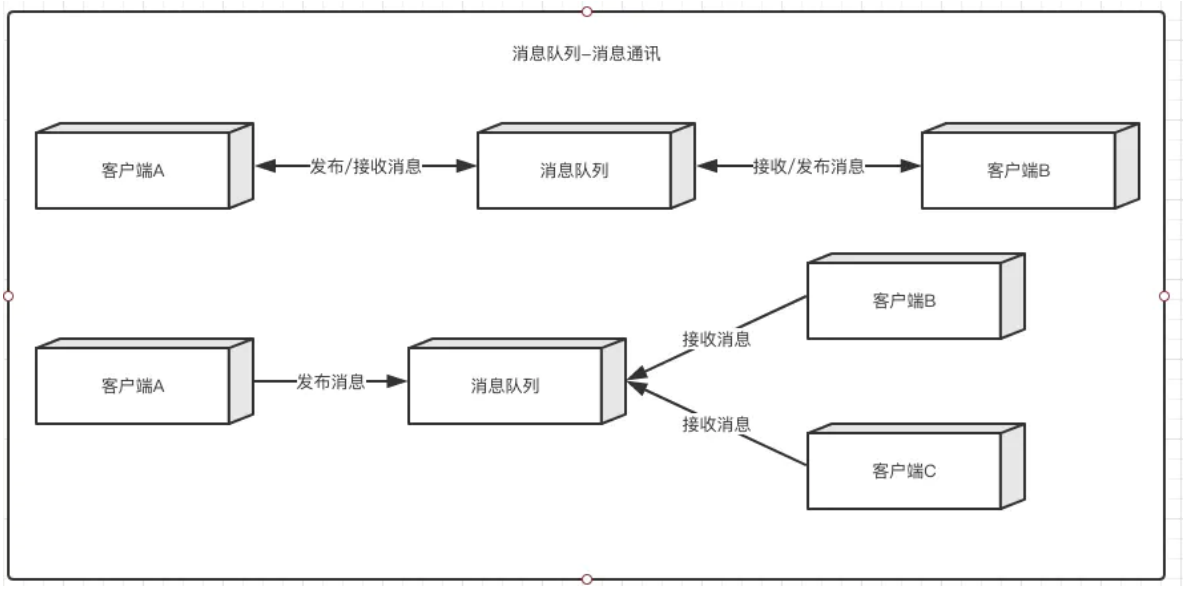


1）用户的请求服务器接受后，首先写入消息队列。加入消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

2）秒杀业务根据消息队列中的请求信息，再做后续处理。

4.消息通讯

消息通讯是指，消息队列一般都内置了高级的通讯机制，因此也可以用作消息通讯。比如实现点对点消息队列，或者聊天室等。

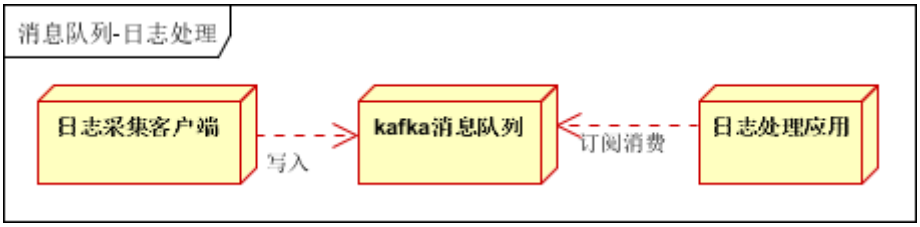


第一个是客户端A和客户端B使用同一队列，进行消息通讯；客户端A、客户端B、客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。

以上实际是消息队列的两种消息模式，点对点或发布订阅模式。

5.日志处理

日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。架构简化如下



日志采集客户端，负责日志数据采集，定时写受写入Kafka队列 Kafka消息队列，负责日志数据的接收，存储和转发 日志处理应用：订阅并消费kafka队列中的日志数据

5.常用消息队列（ActiveMQ、RabbitMQ、RocketMQ、Kafka）比较

- 生产者消费者模式（Producer-Consumer） ActiveMQ-支持，RabbitMQ-支持，RocketMQ-支持，Kafka-支持。
- 发布订阅模式（Publish-Subscribe） ActiveMQ-支持，RabbitMQ-支持，RocketMQ-支持，Kafka-支持。
- 请求回应模型（Request-Reply） ActiveMQ-支持，RabbitMQ-支持，RocketMQ-不支持，Kafka-不支持。
- API完备性 ActiveMQ-高，RabbitMQ-高，RocketMQ-高，Kafka-高。
- 多语言支持 ActiveMQ-支持，RabbitMQ-支持，RocketMQ-只支持JAVA，Kafka-支持。
- 单机吞吐量 ActiveMQ-万级，RabbitMQ-万级，RocketMQ-万级，Kafka-十万级。
- 消息延迟 ActiveMQ-无，RabbitMQ-微秒级，RocketMQ-毫秒级，Kafka-毫秒级。
- 可用性 ActiveMQ-高（主从），RabbitMQ-高（主从），RocketMQ-非常高（分布式），Kafka-非常高（分布式）。
- 消息丢失 ActiveMQ-低，RabbitMQ-低，RocketMQ-理论上不会丢失，Kafka-理论上不会丢失。
- 文档的完备性 ActiveMQ-高，RabbitMQ-高，RocketMQ-高，Kafka-高。

- 提供快速入门 ActiveMQ-有，RabbitMQ-有，RocketMQ-有，Kafka-有。
- 社区活跃度 ActiveMQ-高，RabbitMQ-高，RocketMQ-中，Kafka-高。
- 商业支持 ActiveMQ-无，RabbitMQ-无，RocketMQ-阿里云，Kafka-阿里云。

总体来说：

- ActiveMQ 历史悠久的开源项目，已经在很多产品中得到应用，实现了JMS1.1规范，可以和spring-jms轻松融合，实现了多种协议，不够轻巧（源代码比RocketMQ多），支持持久化到数据库，对队列数较多的情况支持不好。
- RabbitMQ 它比Kafka成熟，支持AMQP事务处理，在可靠性上，RabbitMQ超过Kafka，在性能方面超过ActiveMQ。
- RocketMQ RocketMQ是阿里开源的消息中间件，目前在Apache孵化，使用纯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，但并不是简单的复制，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，支撑了阿里多次双十一活动。 因为是阿里内部从实践到产品的产物，因此里面很多接口、API并不是很普遍适用。其可靠性毋庸置疑，而且与Kafka一脉相承（甚至更优），性能强劲，支持海量堆积。
- Kafka Kafka设计的初衷就是处理日志的，不支持AMQP事务处理，可以看做是一个日志系统，针对性很强，所以它并没有具备一个成熟MQ应该具备的特性。Kafka的性能（吞吐量、tps）比RabbitMQ要强，如果用来做大数据量的快速处理是比RabbitMQ有优势的。

6.ActiveMQ消息队列

三、消息队列中可能存在的问题及解决方案

https://blog.csdn.net/qq_36236890/article/details/81174504

1.可用性

（1）RabbitMQ

RabbitMQ有三种模式：单机模式，普通集群模式，镜像集群模式

- 单机模式

单机模式平常使用在开发或者本地测试场景，一般就是测试是不是能够正确的处理消息，生产上基本没人去用单机模式，风险很大。

- 普通集群模式

普通集群模式就是启动多个RabbitMQ实例。在你创建的queue，只会放在一个rabbtimq实例上，但是每个实例都同步queue的元数据。在消费的时候完了，上如果连接到了另外一个实例，那么那个实例会从queue所在实例上拉取数据过来。

这种方式确实很麻烦，也不怎么好，没做到所谓的分布式，就是个普通集群。因为这导致你要么消费者每次随机连接一个实例然后拉取数据，要么固定连接那个queue所在实例消费数据，前者有数据拉取的开销，后者导致单实例性能瓶颈。

而且如果那个放queue的实例宕机了，会导致接下来其他实例就无法从那个实例拉取，如果你开启了消息持久化，让RabbitMQ落地存储消息的话，消息不一定会丢，得等这个实例恢复了，然后才可以继续从这个queue拉取数据。

这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个queue的读写操作。

- 镜像集群模式

镜像集群模式是所谓的RabbitMQ的高可用模式，跟普通集群模式不一样的是，你创建的queue，无论元数据还是queue里的消息都会存在于多个实例上，然后每次你写消息到queue的时候，都会自动把消息到多个实例的queue里进行消息同步。

优点在于你任何一个实例宕机了，没事儿，别的实例都可以用。缺点在于性能开销太大和扩展性很低，同步所有实例，这会导致网络带宽和压力很重，而且扩展性很低，每增加一个实例都会去包含已有的queue的所有数据，并没有办法线性扩展queue。

开启镜像集群模式可以去RabbitMQ的管理控制台去增加一个策略，指定要求数据同步到所有节点的，也可以要求就同步到指定数量的节点，然后你再次创建queue的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。

（二）Kafka

kafka 0.8以后，提供了HA机制，就是replica副本机制。kafka会均匀的将一个partition的所有replica分布在不同的机器上，来提高容错性。每个partition的数据都会同步到吉他机器上，形成自己的多个replica副本。然后所有replica会选举一个leader出来，那么生产和消费都去leader，其他replica就是follower，leader会同步数据给follower。当leader挂了会自动去找replica，然后会再选举一个leader出来，这样就具有高可用性了。

写数据的时候，生产者就写leader，然后leader将数据落地写本地磁盘，接着其他follower自己主动从leader来pull数据。一旦所有follower同步好数据了，就会发送ack给leader，leader收到所有follower的ack之后，就会返回写成功的消息给生产者。（当然，这只是其中一种模式，还可以适当调整这个行为）

消费的时候，只会从leader去读，但是只有一个消息已经被所有follower都同步成功返回ack的时候，这个消息才会被消费者读到。

2.一致性

3.消息丢失

（一）RabbitMQ

（1）生产者弄丢了数据

- 使用AMQP提供的事务机制实现：生产者将数据发送到RabbitMQ的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。此时可以选择用RabbitMQ提供的事务功能，就是生产者发送数据之前开启RabbitMQ事务（channel.txSelect），然后发送消息，如果消息没有成功被RabbitMQ接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息；如果收到了消息，那么可以提交事务（channel.txCommit）。但是问题是，RabbitMQ事务机制一搞，基本上吞吐量会下来，因为太耗性能。
- 使用发送者确认模式实现：所以一般来说，如果你要确保说写RabbitMQ的消息别丢，可以开启confirm模式，在生产者那里设置开启confirm模式之后，你每次写的消息都会分配一个唯一的id，然后如果写入了RabbitMQ中，RabbitMQ会给你回传一个ack消息，告诉你说这个消息ok了。如果RabbitMQ没能处理这个消息，会回调你一个nack接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息id的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和confirm机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是confirm机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息RabbitMQ接收了之后会异步回调你一个接口通知你这个消息接收到了。

所以一般在生产者这块避免数据丢失，都是用confirm机制的。

（2）RabbitMQ弄丢了数据

就是RabbitMQ自己弄丢了数据，这个你必须开启RabbitMQ的持久化，就是消息写入之后会持久化到磁盘，哪怕是RabbitMQ自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，RabbitMQ还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。

设置持久化有两个步骤，第一个是创建queue的时候将其设置为持久化的，这样就可以保证RabbitMQ持久化queue的元数据，但是不会持久化queue里的数据；第二个是发送消息的时候将消息的deliveryMode设置为2，就是将消息设置为持久化的，此时RabbitMQ就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，RabbitMQ哪怕是挂了，再次重启，也会从磁盘上重启恢复queue，恢复这个queue里的数据。

而且持久化可以跟生产者那边的confirm机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者ack了，所以哪怕是在持久化到磁盘之前，RabbitMQ挂了，数据丢了，生产者收不到ack，你也是可以自己重发的。

哪怕是你给RabbitMQ开启了持久化机制，也有一种可能，就是这个消息写到了RabbitMQ中，但是还没来得及持久化到磁盘上，结果不巧，此时RabbitMQ挂了，就会导致内存里的一点点数据会丢失。

（3）消费端弄丢了数据

RabbitMQ如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ认为你都消费了，这数据就丢了。

这个时候得用RabbitMQ提供的ack机制，简单来说，就是你关闭RabbitMQ自动ack，可以通过一个api来调用就行，然后每次你自己代码里确保处理完的时候，再程序里ack一把。这样的话，如果你还没处理完，不就没有ack？那RabbitMQ就认为你还没处理完，这个时候RabbitMQ会把这个消费分配给别的consumer去处理，消息是不会丢的。

（二）Kafka

（1）消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你那个消费到了这个消息，然后消费者那边自动提交了offset，让kafka以为你已经消费好了这个消息，其实你刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

大家都知道kafka会自动提交offset，那么只要关闭自动提交offset，在处理完之后自己手动提交offset，就可以保证数据不会丢。但是此时确实还是会重复消费，比如你刚处理完，还没提交offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的kafka消费者消费到了数据之后是写到一个内存的queue里先缓冲一下，结果有的时候，你刚把消息写入内存queue，然后消费者会自动提交offset。

然后此时我们重启了系统，就会导致内存queue里还没来得及处理的数据就丢失了

（2）kafka弄丢了数据

这块比较常见的一个场景，就是kafka某个broker宕机，然后重新选举partiton的leader时。大家想想，要是此时其他的follower刚好还有些数据没有同步，结果此时leader挂了，然后选举某个follower成leader之后，他不就少了一些数据？这就丢了一些数据啊。

生产环境也遇到过，我们也是，之前kafka的leader机器宕机了，将follower切换为leader之后，就会发现说这个数据就丢了。

所以此时一般是要求起码设置如下4个参数：

给这个topic设置replication.factor参数：这个值必须大于1，要求每个partition必须有至少2个副本。 在kafka服务端设置min.insync.replicas参数：这个值必须大于1，这个是要求一个leader至少感知到有至少一个follower还跟自己保持联系，没掉队，这样才能确保leader挂了还有一个follower吧。 在producer端设置acks=all：这个是要求每条数据，必须是写入所有replica之后，才能认为是写成功了。 在producer端设置retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。 （3）生产者会不会弄丢数据

如果按照上述的思路设置了ack=all，一定不会丢，要求是，你的leader接收到消息，所有的follower都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

4.消息幂等性

其实消息重复消费的主要原因在于回馈机制（RabbitMQ是ack，Kafka是offset），在某些场景中我们采用的回馈机制不同，原因也不同，例如消费者消费完消息后回复ack，但是刚消费完还没来得及提交系统就重启了，这时候上来就pull消息的时候由于没有提交ack或者offset，消费的还是上条消息。

那么如何怎么来保证消息消费的幂等性呢？实际上我们只要保证多条相同的数据过来的时候只处理一条或者说多条处理和处理一条造成的结果相同即可，但是具体怎么做要根据业务需求来定，例如入库消息，先查一下消息是否已经入库啊或者说搞个唯一约束啊什么的，还有一些是天生保证幂等性就根本不用去管，例如redis就是天然幂等性。

还有一个问题，消费者消费消息的时候在某些场景下要放过消费不了的消息，遇到消费不了的消息通过日志记录一下或者搞个什么措施以后再来处理，但是一定要放过消息，因为在某些场景下例如spring-rabbitmq的默认回馈策略是出现异常就没有提交ack，导致了一直在重发那条消费异常的消息，而且一直还消费不了，这就尴尬了，后果你会懂的。

5.重复消费消息

根据具体的应用场景决定处理方式：

- 从数据库中查询是否已经有消息存在其中
- 借助第三方介质，如redis来存储消费状态，若被消费了则将redis中插入记录

6.消息的顺序性

RabbitMQ:

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

Kafka：

- 一个 topic，一个 partition，一个 consumer，内部单线程消费，单线程吞吐量太低，一般不会用这个。
- 写 N 个内存 queue，具有相同 key 的数据都到同一个内存 queue；然后对于 N 个线程，每个线程分别消费一个内存 queue 即可，这样就能保证顺序性。

7.消息队列中消息积压

几千万条数据在MQ里积压了七八个小时，从下午4点多，积压到了晚上很晚，10点多，11点多 这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复consumer的问题，让他恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条，1000多万条，所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概1小时的时间才能恢复过来。

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

1.先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉。 2.新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量。 3.然后写一个临时的分发数据的consumer程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，4.直接均匀轮询写入临时建立好的10倍数量的queue。 5.接着临时征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的数据。 6.这种做法相当于是临时将queue资源和consumer资源扩大10倍，以正常的10倍速度来消费数据。 7.等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的consumer机器来消费消息。

8.消息队列过期失效

假设你用的是rabbitmq，rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间就会被rabbitmq给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在mq里，而是大量的数据会直接搞丢。

这个情况下，就不是说要增加consumer消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。

这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入mq里面去，把白天丢的数据给他补回来。也只能是这样了。

假设1万个订单积压在mq里面，没有处理，其中1000个订单都丢了，你只能手动写程序把那1000个订单给查出来，手动发到mq里去再补一次。

9.消息队列满了

如果走的方式是消息积压在mq里，那么如果你很长时间都没处理掉，此时导致mq都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

四、Quartz

1.时间表达式

定义的时间格式为：<s, m, h, d, m, w(?), y(?)>，分别表示：秒 分 时 日 月 周 年

星号（*）：可用在所有字段中，表示对应时间域的每一时刻

问好（?）：该字段只用在日期和星期字段，它通常指“无意义的值”，相当于点位符

减号（-）：表达一个范围，如小时字段用“10-12”表示的是从10点到12点，即 10,11,12

逗号（,）：表达一个列表值，如在星期字段中使用“MON,WED,FRI”

斜杠（/）：X/Y表示一个等长序列，X为起始值，Y为增长步长值。如在分钟字段中使用0/15，则表示为0,15,30,45。使用*/Y等同于0/Y

名称	是否必须	允许值	特殊字符
秒	是	0-59	, - * /
分	是	0-59	, - * /
时	是	0-23	, - * /
日	是	1-31	, - * ? / L W C
月	是	1-12 或 JAN-DEC	, - * /
周	是	1-7 或 SUN-SAT	, - * ? / L C #
年	否	空 或 1970-2099	, - * /

2.Trigger

3.数据持久化

五、分布式事务

https://mp.weixin.qq.com/s/T3zeaQeq3GXlZxwP7kIEkQ

分布式事务指事务的操作位于不同的节点上，需要保证事务的 AICD 特性。

在分布式系统中，同时满足CAP定律中的一致性 Consistency、可用性 Availability和分区容错性 Partition Tolerance三者是不可能的。在绝大多数的场景，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证最终一致性。

CAP理解：

- Consistency：强一致性就是在客户端任何时候看到各节点的数据都是一致的（All nodes see the same data at the same time）。
- Availability：高可用性就是在任何时候都可以读写（Reads and writes always succeed）。
- Partition Tolerance：分区容错性是在网络故障、某些节点不能通信的时候系统仍能继续工作（The system continue to operate despite arbitrary message loss or

failure of part of the the system）。以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

一致性有三种策略（CAP指的是强一致性）：

- 强一致性：写操作完成后，后续的读操作都能看到最新数据；
- 弱一致性：能容忍部分或全部都看不到最新数据；
- 最终一致性：经过一段时间后，都能看到最新数据。

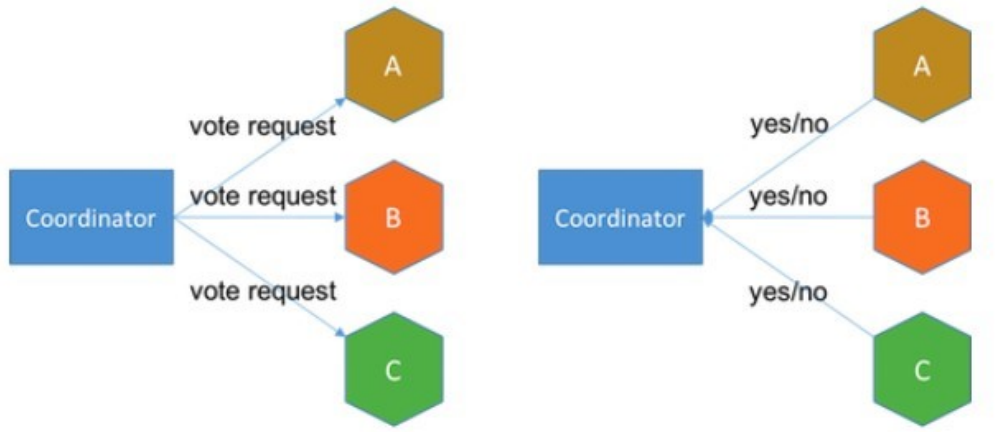
对于一个分布式系统来说，CAP三者中，

- P是基本要求，只能通过基础设施提升，无法通过降低 C/A 来提升；
- 然后在 C/A 两者之间权衡。

1.两阶段提交2PC

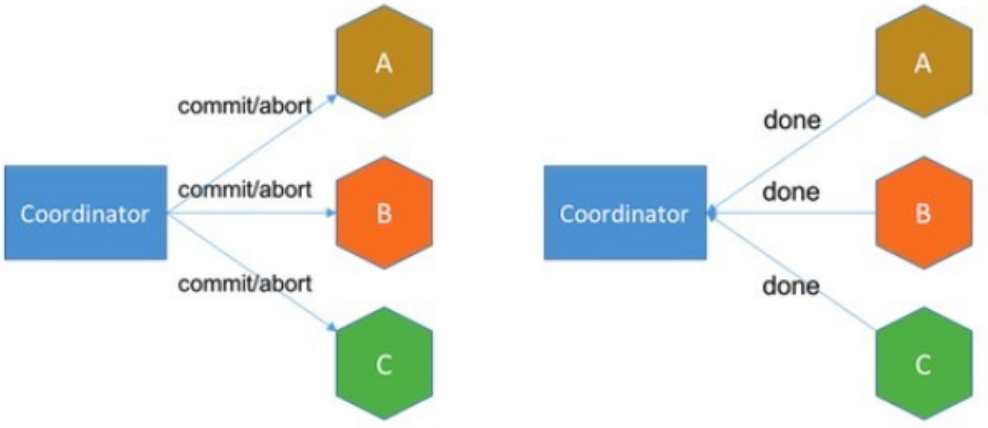
阶段1：请求阶段（commit-request phase，或称表决阶段，voting phase）

在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行故障）。



阶段2：提交阶段（commit phase）

在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。当且仅当所有的参与者同意提交事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。参与者在接收到协调者发来的消息后将执行响应的操作。



存在的问题：

- 同步阻塞：所有事务参与者在等待其它参与者响应的时候都处于同步阻塞状态，无法进行其它操作。
- 单点问题：协调者在 2PC 中起到非常大的作用，发生故障将会造成很大影响。特别是在阶段二发生故障，所有参与者会一直等待状态，无法完成其它操作。
- 数据不一致：在阶段二，如果协调者只发送了部分 Commit 消息，此时网络发生异常，那么只有部分参与者接收到 Commit 消息，也就是说只有部分参与者提交了事务，使得系统数据不一致。
- 太过保守：任意一个节点失败就会导致整个事务失败，没有完善的容错机制。

2.三阶段提交3PC

3PC，三阶段提交协议，是2PC的改进版本，即将事务的提交过程分为CanCommit、PreCommit、do Commit三个阶段来进行处理。

阶段1：**CanCommit**

1、协调者向所有参与者发出包含事务内容的CanCommit请求，询问是否可以提交事务，并等待所有参与者答复。 2、参与者收到CanCommit请求后，如果认为可以执行事务操作，则反馈YES并进入预备状态，否则反馈NO。

阶段2：**PreCommit，此阶段分两种情况：**

1、所有参与者均反馈YES，即执行事务预提交。

2、任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务。

事务预提交：（所有参与者均反馈YES时） 1、协调者向所有参与者发出PreCommit请求，进入准备阶段。 2、参与者收到PreCommit请求后，执行事务操作，将Undo和Redo信息记入事务日志中（但不提交事务）。 3、各参与者向协调者反馈Ack响应或No响应，并等待最终指令。

中断事务：（任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈时） 1、协调者向所有参与者发出abort请求。 2、无论收到协调者发出的abort请求，或者在等待协调者请求过程中出现超时，参与者均会中断事务。

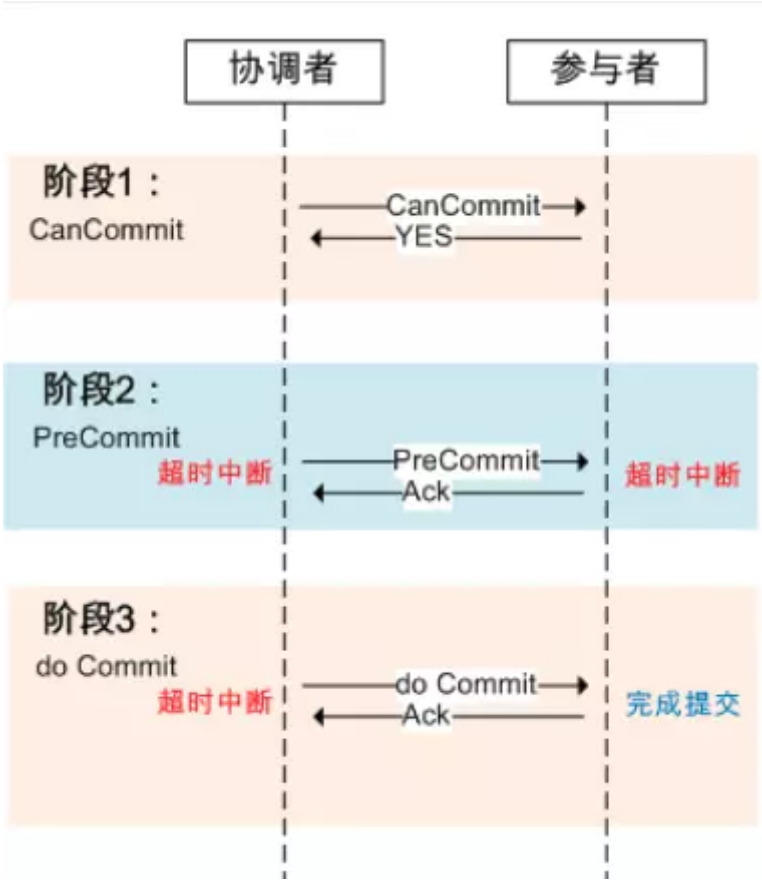
阶段3：**do Commit，此阶段也存在两种情况：**

1、所有参与者均反馈Ack响应，即执行真正的事务提交。 2、任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务。

提交事务：（所有参与者均反馈Ack响应时） 1、如果协调者处于工作状态，则向所有参与者发出do Commit请求。 2、参与者收到do Commit请求后，会正式执行事务提交，并释放整个事务期间占用的资源。 3、各参与者向协调者反馈Ack完成的消息。 4、协调者收到所有参与者反馈的Ack消息后，即完成事务提交。

中断事务：（任何一个参与者反馈NO，或者等待超时后协调者尚无法收到所有参与者的反馈时） 1、如果协调者处于工作状态，向所有参与者发出abort请求。 2、参与者使用阶段1中的Undo信息执行回滚操作，并释放整个事务期间占用的资源。 3、各参与者向协调者反馈Ack完成的消息。 4、协调者收到所有参与者反馈的Ack消息后，即完成事务中断。

注意：进入阶段三后，无论协调者出现问题，或者协调者与参与者网络出现问题，都会导致参与者无法接收到协调者发出的do Commit请求或abort请求。此时，参与者都会在等待超时之后，继续执行事务提交。



优点：降低了阻塞范围，在等待超时后协调者或参与者会中断事务。避免了协调者单点问题，阶段3中协调者出现问题时，参与者会继续提交事务。 缺陷：脑裂问题依然存在，即在参与者收到PreCommit请求后等待最终指令，如果此时协调者无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。

后记：无论2PC或3PC，均无法彻底解决分布式一致性问题。解决一致性问题，唯有Paxos。

六、RocketMQ使用

https://www.jianshu.com/p/2838890f3284

https://www.bilibili.com/video/av66702383?p=16

记一次rocketmq问题排查：https://www.jianshu.com/p/588ec604032f

一次 线上 线程数飙高 导致cpu飙高的解决思路，rocketmq：https://blog.csdn.net/jiewenenike/article/details/81543414

1.角色介绍

Tag：这个是干什么的?

producer：消息生产者，首先向Name Server发起请求询问将消息写入到哪个Broker中，然后再向对应的Broker中写入消息。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。

Producer每隔30s（由ClientConfig的pollNameServerInterval）从Name server获取所有topic队列的最新情况，这意味着如果Broker不可用，Producer最多30s能够感知，在此期间内发往Broker的所有消息都会失败。

Producer每隔30s（由ClientConfig中heartbeatBrokerInterval决定）向所有关联的broker发送心跳，Broker每隔10s中扫描所有存活的连接，如果Broker在2分钟内没有收到心跳数据，则关闭与Producer的连接。

consumer：消息消费者；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。

Consumer每隔30s从Name server获取topic的最新队列情况，这意味着Broker不可用时，Consumer最多最需要30s才能感知。

Consumer每隔30s（由ClientConfig中heartbeatBrokerInterval决定）向所有关联的broker发送心跳，Broker每隔10s扫描所有存活的连接，若某个连接2分钟内没有发送心跳数据，则关闭连接；并向该Consumer Group的所有Consumer发出通知，Group内的Consumer重新分配队列，然后继续消费。

当Consumer得到master宕机通知后，转向slave消费，slave不能保证master的消息100%都同步过来了，因此会有少量的消息丢失。但是一旦master恢复，未同步过去的消息会被最终消费掉。

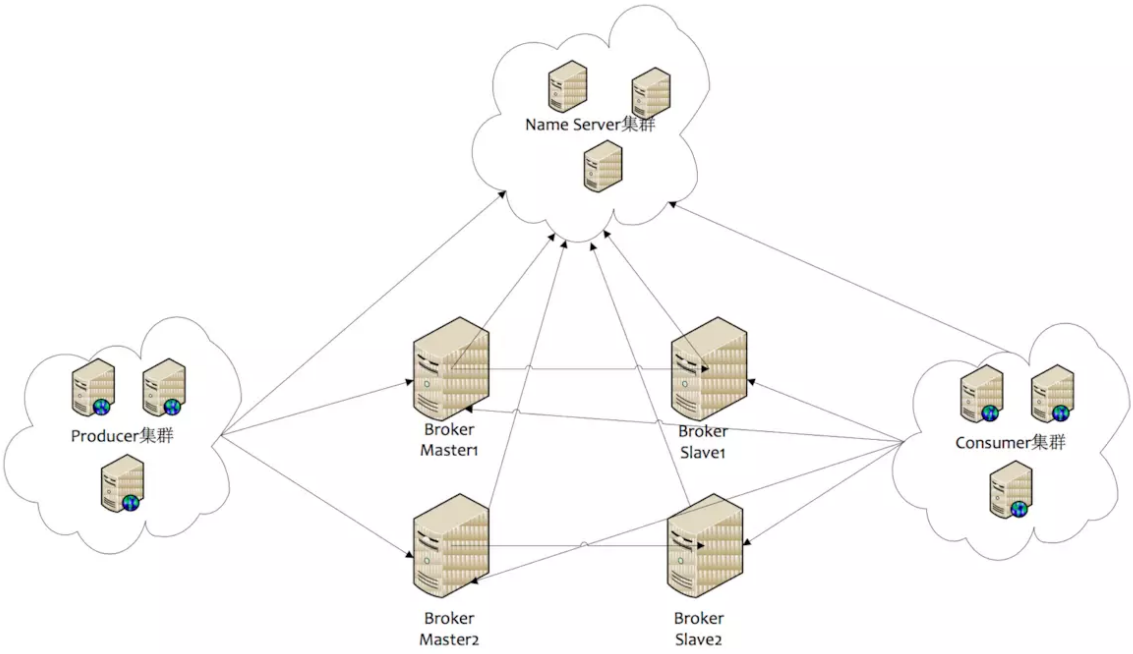
消费者队列是消费者连接之后（或者之前有连接过）才创建的。我们将原生的消费者标识由{IP}@{消费者group}扩展为 {IP}@{消费者group}{topic}{tag}，（例如 xxx.xxx.xxx.xxx@mqtest_producer-group_2m2sTest_tag-zyk）。任何一个元素不同，都认为是不同的消费端，每个消费端会拥有一份自己消费对列（默认是broker队列数量*broker数量）。新挂载的消费者对列中拥有commitlog中的所有数据。

Broker：暂存和传输消息；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Broker Name，然后再根据不同的Broker Id来定义Master和Slave，Broker Id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（每隔30s）注册Topic信息到所有Name Server。Name Server定时（每隔10s）扫描所有存活broker的连接，如果Name Server超过2分钟没有心跳，则Name Server断开与Broker的连接。Master节点主要处理写操作，Slave主要处理读操作

NameSrv：管理Broker，是一个几乎无状态节点，由于Broker会向所有Name server同步自身的信息，所以Name Server之间数据是没有差异的，可集群部署，节点之间无任何信息同步。

Topic：区分消息种类；一个发送者可以发送消息给一个或多个Topic；一个消息接收者可以订阅一个或多个Topic消息

Message Queue：相当于是Topic的分区；用于并行发送和接收消息



2.集群模式

（1）单Master模式

风险较大，一旦Broker重启或宕机时，会导致整个服务不可用。

（2）多Master模式

一个集群无Slave，全是Master，例如2个Master或者更多Master，这种模式的优缺点如下：

- 优点：配置简单，单个Master宕机或重启维护对应用无影响，在磁盘配置为RAID10时，即使机器宕机不可恢复情况下，由于RAID10磁盘非常可靠，消息也不会丢失（异步刷盘丢失少量消息，同步刷盘一条不丢），性能最高；
- 缺点：单台机器宕机期间，这台机器未被消费的消息在机器恢复之前不可订阅，消息实时性会受到影响。

（3）多Master多Slave模式（异步）

每个Master配置一个Slave，有多对Master-Slave，HA采用异步复制方式，只要Master写成功，即刻反馈给客户端写成功状态，主备有短暂消息延迟（毫秒级），优缺点如下：

- 优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不受影响，同时Master宕机后，消费者仍然可以从Slave消费，而且此过程对应用透明，不需要人工干预，性能同多Master模式几乎一样。
- 缺点：Master宕机，磁盘损坏情况下会丢失少量消息。
- 配置方式是：brokerRole=ASYNC_MASTER

（4）多Master多Slave模式（同步）

每个Master配置一个Slave，有多对Master-Slave，HA采用同步双写方式，即只有主备都写成功，才向应用返回成功，优缺点如下：

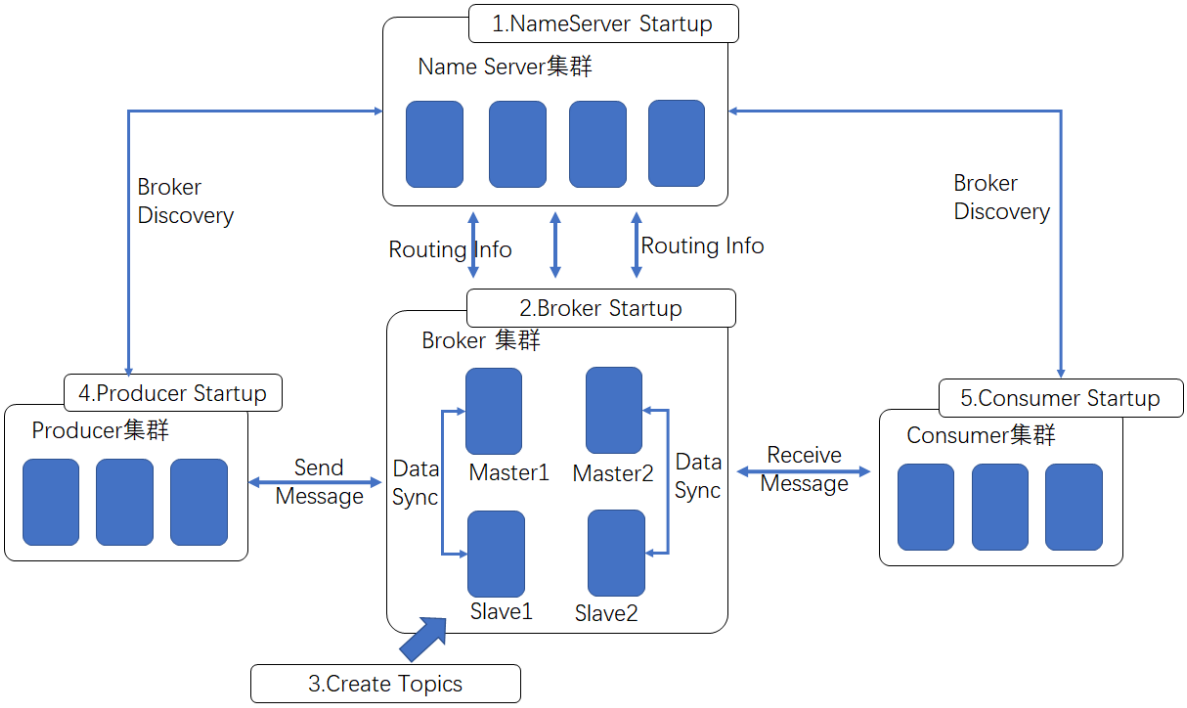
- 优点：数据与服务都无单点故障，Master宕机情况下，消息无延迟，服务可用性与数据可用性都非常高。
- 缺点：性能比异步复制模式略低（大约低10%左右），发送单个消息的RT会略高，且目前版本在主节点宕机后，备机不能自动切换为主机。
- 配置方式是：brokerRole=SYNC_MASTER

（5）配置刷盘和模式

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，由于频繁地触发磁盘写动作，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，主从之间配置成SYNC_MASTER的复制方式，即使有一台机器出现故障，仍然能保证数据不丢，是个不错的选择。

3.安装与启动

- 启动Namesrv：mqnamesrv
- 启动Broker：mqbroker -n localhost:9876 [可选参数：autoCreateTopicEnable=true]
- 启动Broker，指定配置文件：mqbroker -c /...../broker-b-s.properties
- 关闭Namesrv：mqshutdown namesrv
- 关闭Broker：mqshutdown broker
- 可以使用JVM虚拟机自带的jps命令查看启动状态。
- nameserver默认使用9876端口；master默认使用10911端口；slave默认使用11011端口
- 在RocketMQ的conf文件夹下给出了双主双从的配置示例2m-2s-sync和2m-2s-async两个文件夹



配置文件详解如下：

```
# 所属集群名字
brokerClusterName=rocketmq-cluster
# broker名字，注意此处不同的配置文件填写的不一样，同一组Master和Slave这个值要相同
brokerName=broker-a
# 0表示Master，>0 表示slave
brokerId=0
#nameServer地址，分号分割
namesrvAddr=rocketmq-nameserver1:9876;rocketmq-nameserver2:9876
#在发送消息时，自动创建服务器不存在的topic，默认创建的队列数
defaultTopicQueueNums=4
# 是否允许Broker自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
```

```
# 是否允许Broker自动创建订阅组，建议线下开启，线上关闭
autoCreateSubscriptionGroup=true
# Broker对外服务的监听端口
listenPort=10911
# 删除文件时间点，默认凌晨 4点
deleteWhen=04
# 文件保留时间，默认 48小时
fileReservedTime=120
# commitLong每个文件的大小默认1G
mapedFileSizeCommitLog=1073741824
# CousumeQueue每个文件默认存30w条，根据业务情况调整
mapedFileSizeConsumeQueue=300000

# 消息存储路径
storePathRootDir=/usr/local/rocketmq/store
# commitLog存储路径
storePathCommitLog=/usr/local/rocketmq/store/commitlog
# 消费队列存储路径
storePathConsumeQueue=/usr/local/rocketmq/store/consumequeue
# 消息索引存储路径
storePathIndex=/usr/local/rocketmq/store/index
# checkpoint文件存储路径
storeCheckpoint=/usr/local/rocketmq/store/checkpoint
# abort文件存储路径
abortFile=/usr/local/rocketmq/store/abort
# 限制消息大小
maxMessageSize=65536

# Broker角色
# - ASYNC_MASTER 异步复制Master
# - SYNC_MASTER 同步双写Master
# - SLAVE
brokerRole=SYNC_MASTER

# 刷盘方式
# - ASYNC_FLUSH 异步刷盘
# - SYNC_FLUSH 同步刷盘
flushDiskType=SYNC_FLUSH

# 发消息线程池数量
# sendMessageThreadPoolNums=128
# 拉消息线程池数量
# pullMessageThreadPoolNums=128
```

4.mqadmin管理工具

在bin目录下执行./mqadmin {command} {args}

```
The most commonly used mqadmin commands are:
   updateTopic          Update or create topic
   deleteTopic          Delete topic from broker and NameServer.
   updateSubGroup       Update or create subscription group
   deleteSubGroup       Delete subscription group from broker.
   updateBrokerConfig   Update broker's config
```

```
   updateTopicPerm      Update topic perm
   topicRoute           Examine topic route info
   topicStatus          Examine topic Status info
   topicClusterList     Get cluster info for topic
   brokerStatus         Fetch broker runtime status data
   queryMsgById         Query Message by Id
   queryMsgByKey        Query Message by Key
   queryMsgByUniqueKey  Query Message by Unique key
   queryMsgByOffset     Query Message by offset
   queryMsgByUniqueKey  Query Message by Unique key
   printMsg             Print Message Detail
   sendMsgStatus        Send msg to broker.
   brokerConsumeStats   Fetch broker consume stats data
   producerConnection   Query producer's socket connection and client version
   consumerConnection   Query consumer's socket connection, client version and subscription
   consumerProgress     Query consumers's progress, speed
   consumerStatus       Query consumer's internal data structure
   cloneGroupOffset     Clone offset from other group.
   clusterList          List all of clusters
   topicList            Fetch all topic list from name server
   updateKvConfig       Create or update KV config.
   deleteKvConfig       Delete KV config.
   wipeWritePerm        Wipe write perm of broker in all name server
   resetOffsetByTime    Reset consumer offset by timestamp(without client restart).
   updateOrderConf      Create or update or delete order conf
   cleanExpiredCQ       Clean expired ConsumeQueue on broker.
   cleanUnusedTopic     Clean unused topic on broker.
   startMonitoring      Start Monitoring
   statsAll             Topic and Consumer tps stats
   syncDocs             Synchronize wiki and issue to github.com
   allocateMQ           Allocate MQ
   checkMsgSendRT       Check message send response time
   clusterRT            List All clusters Message Send RT
```

https://blog.csdn.net/Howie_zhw/article/details/52669759

查看消息堆积情况：mqadmin consumerProgress -g consumerGroup -n localhost:9876

#Topic	#Broker Name	#QID	#Broker Offset	#Consumer Offset	#Client IP	#Diff	#LastTime
%DLQ%consumerGroup	broker-a	0	10	10	N/A	0	2020-03-22 17:14:45
%RETRY%consumerGroup	broker-a	0	10	10	N/A	0	2020-03-22 17:14:35

Broker Offset：为生产的条数

Consumer Offset：为消费的条数

Diff：为堆积的条数

max offset：字面上可以理解为这是标识message queue中的max offset表示消息的最大offset。但是从源码上看，这个offset实际上是最新消息的offset+1，即：下一条消息的offset。

min offset：标识现存在的最小offset。而由于消息存储一段时间后，消费会被物理地从磁盘删除，message queue的min offset也就对应增长。这意味着比min offset要小的那些消息已经不在broker上了，无法被消费。

查看Topic状态：mqadmin topicStatus -n localhost:9876 -t testTopic

（1）Topic相关

updateTopic 创建新Topic配置

-b Broker地址，表示topic所在Broker，只支持单台Broker，地址为ip:port -c cluster名称，表示topic所在集群（集群可通过clusterList查询） -h 打印帮助 -n NameServer服务地址，格式ip:port -p 指定新Topic的读写权限（W=2|R=4|WR=6） -r 可读队列数（默认为8） -w 可写队列数（默认为8） -t topic名称（名称只能使用字符^[a-zA-Z0-9_-]+$） deleteTopic 删除Topic

-c cluster名称，表示删除某集群下的某个topic（集群可通过clusterList查询） -h 打印帮助 -n NameServer服务地址，格式ip:port -t topic名称（名称只能使用字符^[a-zA-Z0-9_-]+$） topicList 查看Topic列表信息

-h 打印帮助 -c 不配置-c只返回topic列表，增加-c返回clusterName，topic，consumerGroup信息，即topic的所属集群和订阅关系，没有参数 -n NameServer服务地址，格式ip:port topicRoute 查看Topic路由信息

-t topic名称 -h 打印帮助 -n NameServer服务地址，格式ip:port topicStatus 查看Topic消息队列offset

-t topic名称 -h 打印帮助 -n NameServer服务地址，格式ip:port topicClusterList 查看Topic所在集群列表

-t topic名称 -h 打印帮助 -n NameServer服务地址，格式ip:port updateTopicPerm 更新topic读写权限

-t topic名称 -h 打印帮助 -n NameServer服务地址，格式ip:port -b Broker地址，表示topic所在Broker，只支持单台Broker，地址为ip:port -p 指定新Topic的读写权限（W=2|R=4|WR=6） -c cluster名称，表示topic所在集群（集群可通过clusterList查询）,-b优先，如果没有-b，则对集群中所有Broker执行命令 updateOrderConf 从NameServer上创建、删除、获取特定命名空间的kv配置，目前还未启用

-h 打印帮助 -n NameServer服务地址，格式ip:port -t topic，键 -v orderConf，值 -m method,可选个体、put、delete allocateMQ 以平均负载算法计算消费者列表负载消息队列的负载结果

（2）集群相关

（3）Broker相关

（4）消息相关

（5）消费者、消费组相关

（6）连接相关

（7）NameServer相关

（8）其他

5.发送消息

导入RocketMQ客户端依赖

```
<!-- 消息队列 -->
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.5.1</version>
</dependency>
```

消息发送者步骤分析

```
1.创建消息生产者producer，并制定生产者组名
2.指定Nameserver地址
3.启动producer
4.创建消息对象，指定主题Topic、Tag和消息体
5.发送消息
6.关闭生产者producer
```

消息消费者步骤分析

```
1.创建消费者Consumer，指定消费者组名
2.指定Nameserver地址
3.订阅主题Topic和Tag
4.设置回调函数，处理消息
5.启动消费者Consumer
```

（1）消息发送

1）发送同步消息

这种可靠性同步的发送方式使用比较广泛，如：重要的消息通知，短信通知。

```
// 1.创建消息生产者producer，并制定生产者组名
DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
// 2.指定Nameserver地址
producer.setNamesrvAddr("localhost:9876");
// 3.启动producer
producer.start();
// 4.创建消息对象，指定主题Topic、Tag和消息体
Message msg = new
Message("Topic","Tag","messageBody".getBytes(RemotingHelper.DEFAULT
_CHARSET));
// 5.发送消息
SendResult sendResult = producer.send(msg);
System.out.println(sendResult);
// 6.关闭生产者producer
producer.shutdown();
```

2）发送异步消息

异步消息通常在对响应时间敏感的业务场景，即发送端不能容忍长时间地等待Broker的响应，消息可靠性没有发送同步消息高。!!!!不可以立即关闭producer，否则会报错。

```
// 1.创建消息生产者producer，并制定生产者组名
DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
// 2.指定Nameserver地址
producer.setNamesrvAddr("localhost:9876");
// 3.启动producer
producer.start();
// 4.创建消息对象，指定主题Topic、Tag和消息体
Message msg = new
Message("Topic","Tag","messageBody".getBytes(RemotingHelper.DEFAULT
_CHARSET));
// 5.发送消息
producer.send(msg,new SendCallBack(){
    @override
    public void onSuccess(SendResult sendResult){
        System.out.println(sendResult);
    }
    @override
    public void onException(Throwable e){
        System.out.printf("%-10d Exception %s %n",index,e);
        e.printStackTrace();
    }
});
// 6.关闭生产者producer!!!!不可以立即关闭producer，否则会报错。
producer.shutdown();
```

3）单向发送消息

这种方式主要用在不关心发送结果的场景，例如日志发送。

```
// 1.创建消息生产者producer，并制定生产者组名
DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
// 2.指定Nameserver地址
producer.setNamesrvAddr("localhost:9876");
// 3.启动producer
producer.start();
// 4.创建消息对象，指定主题Topic、Tag和消息体
Message msg = new
Message("Topic","Tag","messageBody".getBytes(RemotingHelper.DEFAULT
_CHARSET));
// 5.发送消息
producer.sendOneway(msg);
// 6.关闭生产者producer
producer.shutdown();
```

（2）消费消息

默认消费模式是负载均衡。

1）负载均衡模式

多个消费者共同消费同一队列中的数据，但不重复消费

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer();
consumer.setNamesrvAddr("localhost:9876;127.0.0.1:9876");
consumer.subscribe("testTopic","Tag2");
// 设置成负载均衡模式
consumer.setMessageModel(MessageModel.CLUSTERING);
consumer.registerMessageListener(new MessageListenerConcurrently()
{
    @Override
    public ConsumeConcurrentlyStatus
consumeMessage(List<MessageExt> list, ConsumeConcurrentlyContext
consumeConcurrentlyContext) {
        // 消费
        ret urn ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
consumer.start();
```

2）广播模式

每个消费者都消费了一遍消息。

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer();
consumer.setNamesrvAddr("localhost:9876;127.0.0.1:9876");
consumer.subscribe("testTopic","Tag2");
// 设置成广播模式
consumer.setMessageModel(MessageModel.BROADCASTING);
consumer.registerMessageListener(new MessageListenerConcurrently()
{
    @Override
    public ConsumeConcurrentlyStatus
consumeMessage(List<MessageExt> list, ConsumeConcurrentlyContext
consumeConcurrentlyContext) {
        // 消费
        ret urn ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
consumer.start();
```

（3）顺序消息

消息有序指的是可以按照消息的发送顺序来消费（FIFO）。RocketMQ可以严格的保证消息有序，可以区分为分区有序或者全局有序。

顺序消费的原理解析，在默认的情况下消息发送会采取Round Robin轮询方式把消息发送到不同的queue（分区队列）；而消费消息的时候从多个queue上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个queue中，消费的时候只从这个queue上依次拉取，则保证了顺序。当发送和消费参与的queue只有一个，则是全局有序；如果多个queue参与，则为分区有序，即相对每个queue，消息都是有序的。

下面用订单进行分区有序的示例。一个订单的顺序流程是：创建、付款、推送、完成，订单号相同的消息会被先后发送到同一队列中，消费时，同一个OrderId获取到的肯定是同一队列。

1）顺序消息生产

```
DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
producer.setNamesrvAddr("localhost:9876");
producer.start();
Order order= getOrder();
Message msg = new
Message("orderTopic","orderTag",order.getOrderId().getBytes());
/**
* 参数一：消息对象
* 参数二：消息队列选择器
* 参数三：选择队列的业务标识（订单ID）
*/
producer.send(msg,new MessageQueueSelector(){
    /**
    * 参数一：队列集合
    * 参数二：消息对象
    * 参数三：业务标识的参数
    */
    public MessageQueue select(List<MessageQueue> mqs,Message
msg,Object arg){
```

```
        long orderId = (long) arg;
        long index = orderId % mqs.size();
        return mqs.get((int)index);
    }
},order.getOrderId());
producer.shutdown();
```

2）顺序消息消费

```
DefaultMQPushConsumer consumer = new DefaultMQPushConsumer();
consumer.setNamesrvAddr("localhost:9876;127.0.0.1:9876");
consumer.subscribe("orderTopic","orderTag");
// 设置成负载均衡模式
consumer.setMessageModel(MessageModel.CLUSTERING);
consumer.registerMessageListener(new MessageListenerOrderly(){
    // 一个队列一个线程消费，保证队列中消息的绝对顺序性
    public ConsumerOrderlyStatus consumeMessage(List<MessageExt>
msgs,ConsumerOrderlyContex context){
        for(MessageExt msg:msgs){
            System.out.println(Thread.currentThread.getName() + new
String(msg.getBody()));
        }
        return ConsumerOrderlyStatus.SUCCESS;
    }
});
consumer.start();
```

（3）延时消息

比如电商里，提交了一个订单就可以发送一个延时消息，1h后再去检查这个订单的状态，如果还是未付款就取消订单释放库存。到底是先发送到队列中再延迟消费还是在producer中就延迟了呢?

发送延时消息:

```
DefaultMQProducer producer = new
DefaultMQProducer("ExampleProducerGroup");
producer.setNamesrvAddr("localhost:9876");
producer.start();
Message msg = new Message("testTopic","tag","content".getBytes());
// 设置延时等级3，这个消息将在10s之后发送（现在只支持固定的几个时间，详情看
delayTimeLevel）
message.setDelayTimeLevel(3);
producer.send(msg);
producer.shutdown();
```

可选的延迟时间有：1s、5s 、10s、30s、1m、2m、3m、4m、5m、6m、7m、8m、9m、10m、20m、30m、1h、2h。从1s到2h对应着等级1到18。不能设置任意时间。消费方式还是和之前的相同

（4）发送批量消息

批量发送消息能显著提高传递小消息的性能。限制是这些批量消息都应有相同topic，相同的waitStoreMsgOK，而且不能是延时消息。此外，这一批消息的总大小不应超过4MB。

发送批量消息：

```
...
// 这批消息的总大小不超过4MB
String topic = "batchTest";
List<Message> messages = new ArrayList<>();
messages.add(new Message(topic,"TagA","content1".getBytes()));
messages.add(new Message(topic,"TagA","content2".getBytes()));
messages.add(new Message(topic,"TagA","content3".getBytes()));
try{
    SendResult result = producer.send(messages);
} catch(Exception e){
    e.printStackTrace();
}

...
```

如果这一批的消息总大小超过4MB，那么最好把消息进行分割。

```
public class ListSplitter implements Iterator<List<Message>>{
    private final int SIZE_LIMIT= 1024 * 1024 *4;
    private final List<Message> messages;
    private int currIndex;
    public ListSplitter(List<Message> messages){
        this.messages = messages;
    }
    public boolean hasNext(){
        return currIndex < messages.size();
    }
    public List<Message> next(){
        int nextIndex = currIndex;
        int totalSize=0;
        for(;nextIndex < messages.size();nextIndex++){
            Message message = messages.get(nextIndex);
            int tmpSize = message.getTopic().length() +
message.getBody().length();
            // 额外属性
            Map<String,String> properties =
message.getProperties();
            for(Map.Entry<String,String>
entry:properties.entrySet()){
                tmpSize += entry.getKey().length() +
entry.getValue().length();
            }
            tmpSize += 20;// 增加日志的开销20字节
            if(tmpSize > SIZE_LIMIT){
                // 单个消息超过了最大的限制
                // 忽略，否则会阻塞分裂的进程
                if(nextIndex - currIndex == 0){
                    // 假如下一个子列表没有元素，则添加这个字列表然后退
出循环，否则只是退出循环
                    nextIndex ++;
                }
                break;
```

```
            }
            if(tmpSize + totalSize > SIZE_LIMIT){
                break;
            } else {
                totalSize += tmpSize;
            }
        }
        List<Message> subList =
messages.subList(currIndex,nextIndex);
        currIndex = nextIndex;
        return subList;
    }
}

// 在使用时
ListSplitter splitter = new Splitter(message);
while(splitter.hasNext()){
    try{
        List<Message> listItem = splitter.next();
        producer.send(listItem);
    }catch (Exception e){
        e.printStackTrace();
        // 处理error
    }
}
```

(5) 过滤消息

- 针对队列使用不同的tag过滤，可以使用*代表消费当前队列的所有tag，或者使用tag1 || tag2来消费两个tag下的值。
- 使用sql进行过滤。

```
在生产者中：msg.putUserProperty("i",String.valueOf(1))
在消费者中：
    consumer.subscribe("topic",MessageSelector.bySql("i>5"));
    consumer.subscribe("topic",MessageSelector.bySql("i between
1 and 5"));
```

RocketMQ只定义了一些基本语法来支持这个特性，sql语法为：

- 数值比较：>,>=,<,<=,BETWEEN,=
- 字符比较：=,<>,IN
- IS NULL 或者IS NOT NULL
- 逻辑符号：AND, OR, NOT

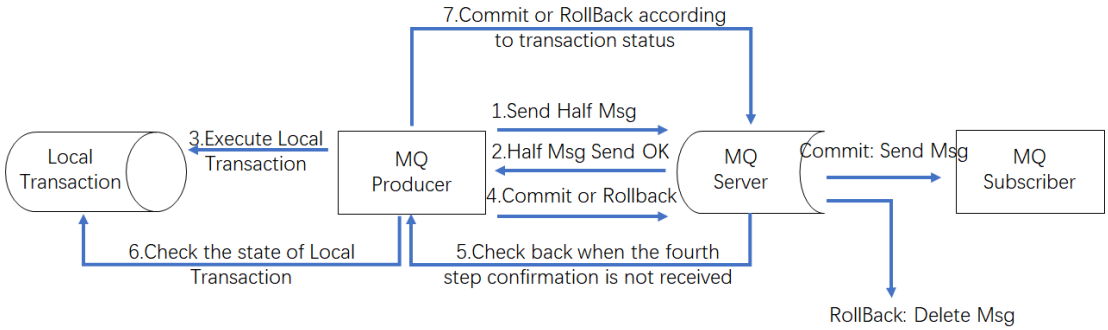
常量支持类型为：

- 数值：比如：123,3.1415
- 字符，'abc'，必须用单括号包裹起来
- NULL，特殊的常量
- 布尔值，TRUE或FALSE

只有在push模式的消费者才能使用SQL92标准的sql语句，接口如下：

```
public void subscribe(finalString topic,final MessageSelector
messageSelector)
```

（6）事务消息



上图说明了事务消息的大致方案，其中分为两个流程：正常消息的发送及提交、事务消息的补偿流程。

1）事务消息发送及提交

1. 发送消息（half消息）
2. 服务端响应消息写入结构
3. 根据发送结果执行本地事务（如果写入失败，此时half消息对业务不可见，本地逻辑不执行）
4. 根据本地事务状态执行commit或者rollback（commit操作生成消息索引，消息对消费者可见）

2）事务补偿

1. 对没有Commit/Rollback的事务消息（pending状态的消息），从服务端发起一次“回查”
2. Producer收到回查消息，检查回查消息对应的本地事务的状态
3. 根据本地事务状态，重新Commit或者Rollback

其中，补偿阶段用于解决消息Commit或者Rollback发生超时或者失败的情况

3）事务消息状态

事务消息状态有三种，提交状态、回滚状态、中间状态：

- TransactionStatus.CommitTransaction：提交事务，它允许消费者消费此消息
- TransactionStatus.RollbackTransaction：回滚事务，它代表该消息将被删除，不允许被消费
- TransactionStatus.Unknown：中间状态，它代表需要检查消息队列来确定状态

```
1  // 1.创建消息生产者producer，并制定生产者组名
2  TransactionMQProducer producer = new
   TransactionMQProducer("testGroup");
3  // 2.指定Nameserver地址
4  producer.setNamesrvAddr("127.0.0.1:9876");
5  producer.setTransactionListener(new TransactionListener() {
6      /***
7      * 执行本地事务的入口
8      * @param message
9      * @param o
10     * @return
11     */
12     @Override
13     public LocalTransactionState executeLocalTransaction(Message
   message, Object o) {
14         if("TAGA".equals(String.valueOf(message.getTags()))){
```

```
            // 模拟事务执行成功
            return LocalTransactionState.COMMIT_MESSAGE;
        } else if("TAGB".equals(String.valueOf(message.getTags()))){
            // 模拟事务回滚
            return LocalTransactionState.ROLLBACK_MESSAGE;
        }
        // 回查
        return LocalTransactionState.UNKNOW;
    }

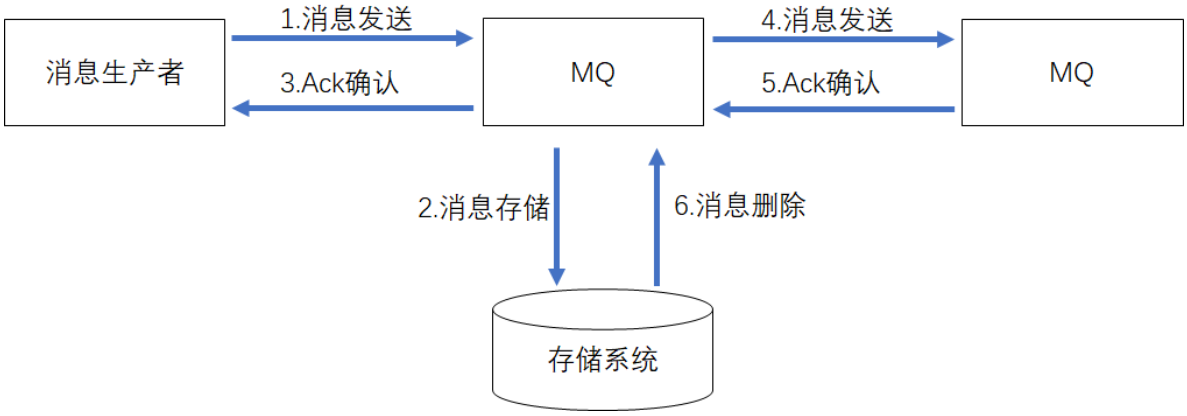
    /***
    * 是MQ进行消息事务状态回查
    * @param messageExt
    * @return
    */
    @Override
    public LocalTransactionState checkLocalTransaction(MessageExt messageExt) {
        return LocalTransactionState.COMMIT_MESSAGE;
    }
});
// 3.启动producer
producer.start();
// 4.创建消息对象，指定主题Topic、Tag和消息体
Message msg = new Message("testTopic","testTag","messageBody".getBytes(RemotingHelper.DEFAULT_CHARSET));
// 5.发送消息，将事务应用到整个producer上
producer.sendMessageInTransaction(msg,null);
// 6.关闭生产者producer
//        producer.shutdown();
```

注意:

- 不支持延迟消息和批量消息。
- broker回查次数默认为15次，可通过transactionCheckMax参数设置。
- broker在transactionTimeout时间后进行回查，可以在发送事务消息是通过CHECK_IMMUNITY_TIME_IN_SECONDS设置回查时间
- 事务性消息可能不止一次被检查或消费，在消费方做好幂等性检查
- 条给用户的目标主题可能会失效，目前这依日志的记录而定。它的高可用性通过RocketMQ本身的高可用性机制来保证，如果希望确保事务消息不丢失，并且事务完整性得到保证，建议使用同步的双重写入机制。
- ActiveMQ也支持事务消息

6.高级功能

(1) 消息存储



1）关系型数据库DB

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。由于普通关系型数据库在单表数量达到千万级别的情况下，其IO读写性能往往会出现瓶颈，在可靠性方面，该种方案非常依赖DB，如果一旦DB出现故障，则MQ的消息就无法落盘存储会导致线上故障。

2）文件系统

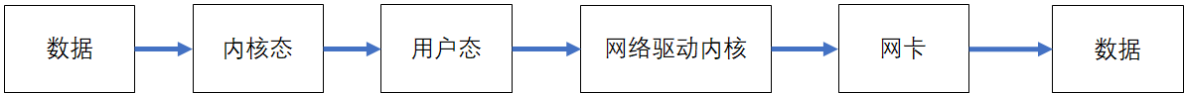
目前业界较为常见的几款产品（RocketMQ/Kafka/RabbitMQ）均采用的是消息刷盘至部署虚拟化/物理机的文件系统来做持久化（刷盘一般可以分为异步刷盘和同步刷盘两种模式）。消息刷盘为消息存储提供了一种高效率、高可靠性和高性能的数据持久化方式。除非部署MQ机器本身或者本地磁盘挂了，否则一般是不会出现无法持久化的故障问题。性能比关系型数据库更好。

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。目前的高性能磁盘，顺序写速度可以达到600MB/s，超过了一般网卡的传输速度。但是磁盘随机写的速度只有大概100kb/s，和顺序写的性能相差6000倍。RocketMQ的消息用顺序写，保证了消息存储的速度。

3）消息发送

Linux操作系统分为用户态和内核态，文件操作、网络操作需要设计这两种形态的切换，免不了进行数据复制。详情见：Java IO的笔记。一台服务器把本地磁盘文件的内容发送到客户端，一般分为两个步骤：

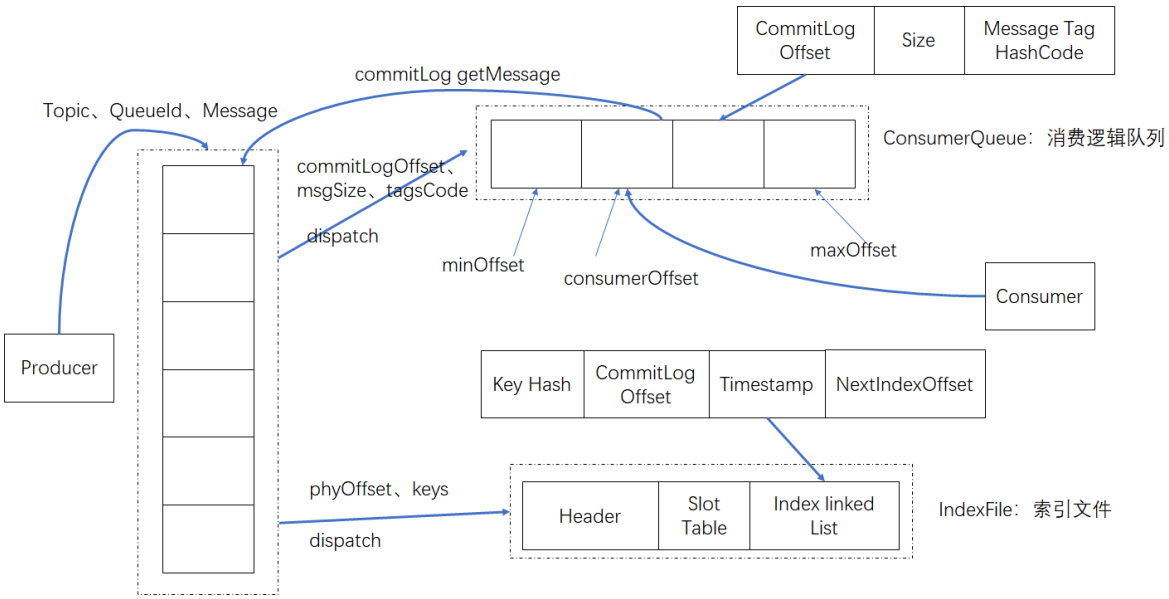
- read：读取本地文件内容；
- write：将读取的内容通过网络发送出去



通过mmap的方式，可以省去从用户态的内存复制，提高速度。这种机制在Java中是通过MappedByteBuffer实现的。RocketMQ充分利用了上述特征，也就是所谓的“零拷贝”技术，提高消息存盘和网络发送的速度。

> *这里需要注意的是，采用MapperByteBuffer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1.5~2G的文件至用户态的虚拟内存，这也就是为何RocketMQ默认设置单个CommitLog日志数据文件为1G的原因了。*

4）消息存储结构



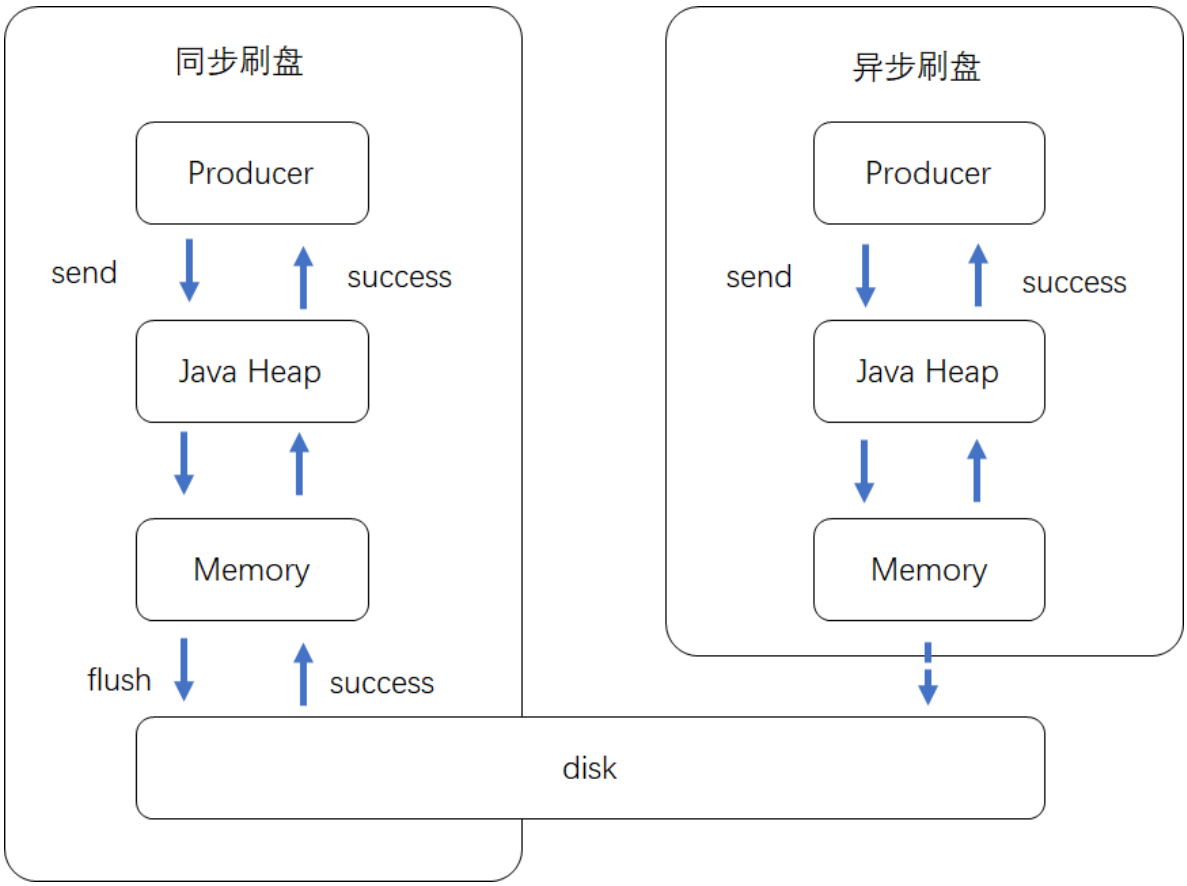
commitLog：存储消息的元数据

ConsumerQueue：存储消息在CommitLog的索引，如果它丢了，可以从commitLog中重新生成。

IndexFile：为了消息查询提供了一种通过key或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消息的主流程。

5）刷盘机制

RocketMQ的消息是存储在磁盘上的，这样既能保证断电后恢复，又可以让存储的消息量超出内存的限制。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：在返回写成功状态后，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的线程，返回消息写成功的状态。

异步刷盘：在返回写成功时，消息可能只是被写入内存的PAGECACHE，写操作的返回快，吞吐量大；当内存里的消息累积到一定程度时，统一触发磁盘写动作，快速写入。

配置：同步刷盘还是异步刷盘，都是通过Broker配置文件里的flushDiskType参数设置的，可选项有SYNC_FLUSH、ASYNC_FLUSH中的一个。

（2）高可用机制

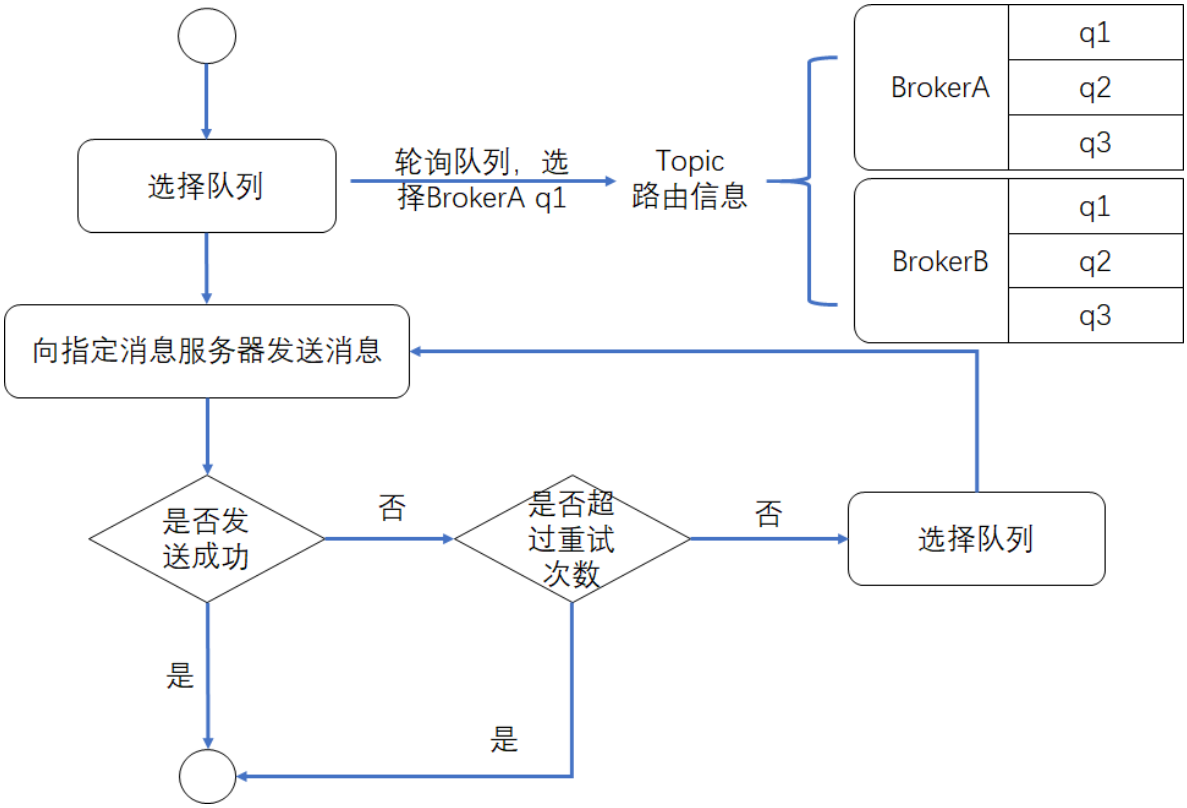
1）消息消费高可用

在Consumer的配置文件中，并不需要设置是从Master读还是从Slave读，当Master不可用或者繁忙的时候，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性。

2）消息生产高可用

在创建Topic的时候，把Topic的多个Message Queue创建在多个Broker组上（相同Broker名称，不同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仍然可以发送消息。RocketMQ目前还不支持把Slave自动转成Master，如果机器资源不足，把Slave转成Master，则要手动停止Slave角色的Broker，更改配置文件，用新的配置文件启动Broker。

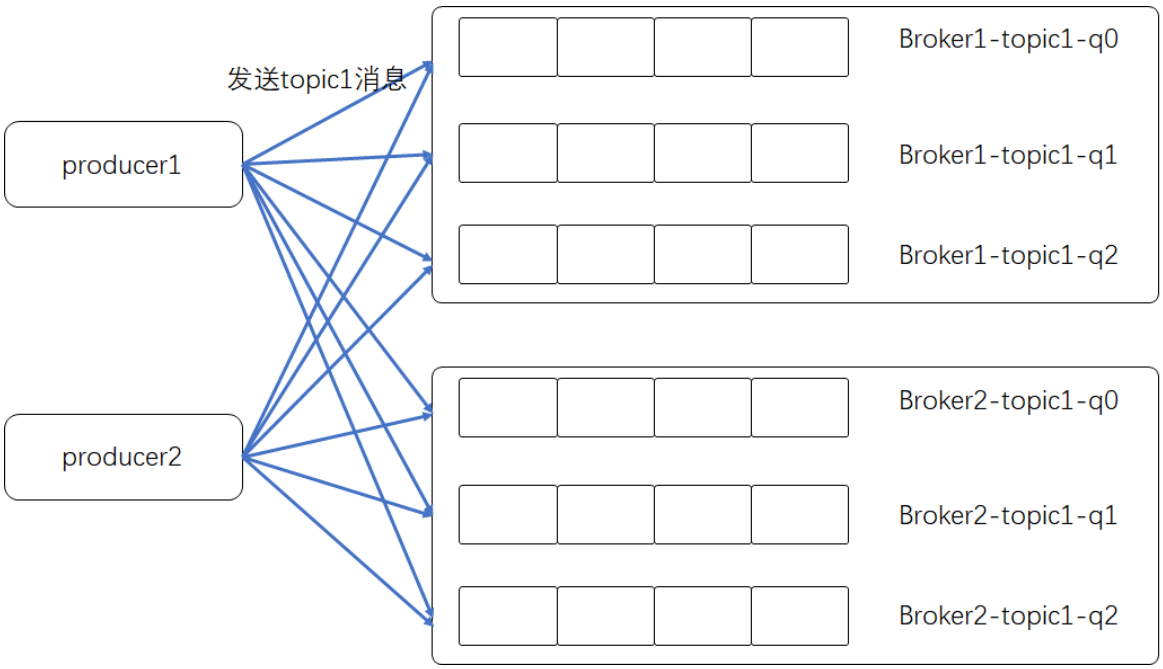
> *上一次选择的队列为BrokerA服务器的q1队列，如果消息发送失败是由于BrokerA宕机引起的，如果本次继续轮询，则选择的队列为BrokerA服务器的q2队列，岂不是还会发送失败，这重试有什么意义呢?*
>
> *RocketMQ的解决办法是：在发送失败时，如果向某一个Broker发送失败后，会设置一个规避策略，例如在接下来的5分钟内选择队列时，发生错误的Broker中的队列将不参加队列负载，即会跳过BrokerA服务器上调度队列，只选择BrokerB服务器上的队列。*



（3）负载均衡

1）Producer负载均衡

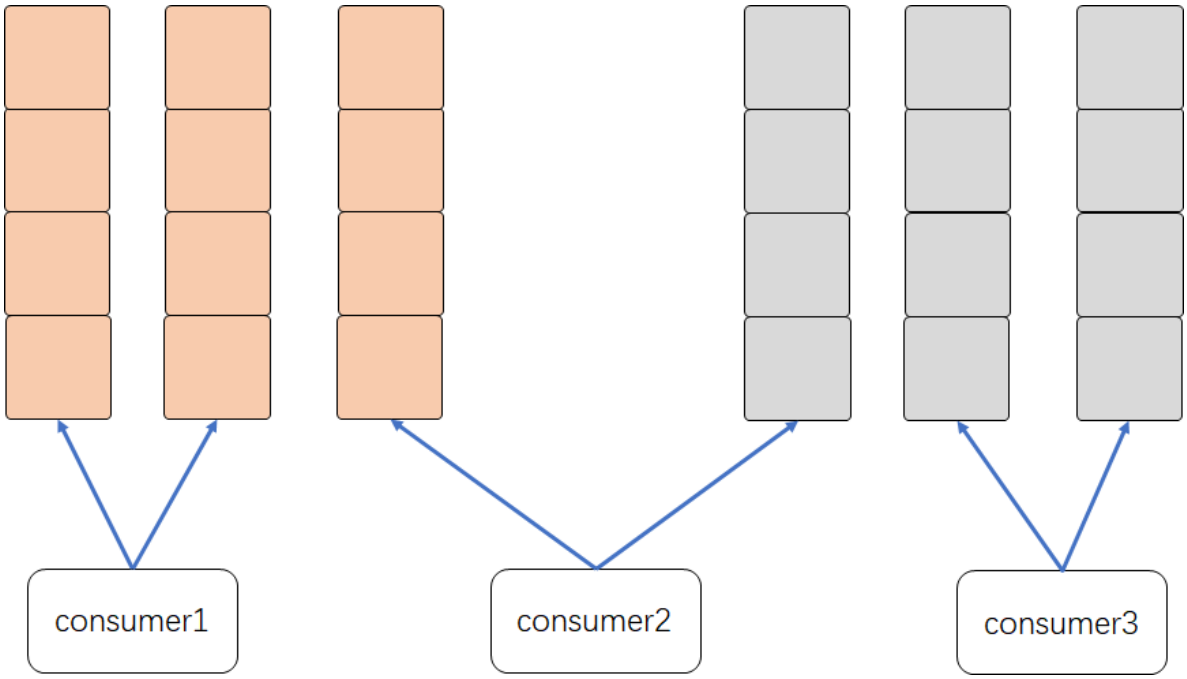
Producer端，每个实例在发送消息的时候，默认会轮询所有的message queue发送，以达到让消息平均落在不同的queue上。而由于queue可以散落在不同的broker，所以消息就发送到不同的broker下，如下图：



每个producer默认采用Round Robin方式轮询发送每个Queue，producer会把第一条消息发送至Queue0，然后第二条消息发送至Queue 1，以此类推。只需要发就行，上述是RocketMQ中的内部实现。

2）Consumer负载均衡

集群模式时：每条消息只需要投递到订阅这个topic的Consumer Group下的一个实例即可。RocketMQ采用主动拉取的方式拉取并消费消息，在拉取的时候需要明确指定拉取哪一条message queue。而每当实例的数量有变更，都会触发一次所有实例的负载均衡，这时候会按照queue的数量和实例数量平均分配queue给每个实例。默认的分配算法是AllocateMessageQueueAveragely，每个consumer实例平均分配每个consumer queue，如下图：



还有另外一种平均算法是AllocateMessageQueueAveragelyByCircle，也是平均分摊每一条queue，只是以环状轮流分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同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consume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，会重新触发负载均衡，这时候原来分配到的queue将分配到其他实例上继续消费。

但是如果consumer实例的数量比message queue的总数量还多的话，多出来的consumer实例将无法分到queue，也就无法消费到消息，也就无法起到分摊负载的作用了，所以需要控制让queue的总数量大于等于consumer的数量。

广播模式时：由于广播模式下要求一条消息需要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分配queue的时候，所有consumer都分到所有的queue。

（4）消息重试

1）顺序消息的重试

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（每次间隔时间为1秒），这时，应用会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。

2）无序消息的重试

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，每次重试的间隔时间如下：

第几次重试	与上次重试的间隔时间	第几次重试	与上次重试的间隔时间
1	10秒	9	7分钟
2	30秒	10	8分钟
3	1分钟	11	9分钟
4	2分钟	12	10分钟
5	3分钟	13	20分钟
6	4分钟	14	30分钟
7	5分钟	15	1小时
8	6分钟	16	2小时

如果消息重试16次后仍然失败，消息将不再传递。如果严格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分钟之内进行16次重试，超过这个时间范围消息将不再重试投递。

注意：一条消息无论重试多少次，这些重试消息的Message ID不会改变。

？？那未被消费的消息将如何处理呢？？？

消费失败后，重试配置方式：

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

- 返回Action.ReconsumerLater（推荐）
- 返回NULL
- 抛出异常

```
1  public class MessageListenerImpl implements MessageListener{
2      @Override
3      public Action consume(Message message,ConsumerContext
   context){
4          // 处理消息
5          doConsumerMessage(message);
6          // 方式1：返回Action.ReconsumerLater，消息将重试
7          return Action.ReconsumerLater;
```

```
        // 方式2：返回null，消息将重试
        return null;
        // 方式3：直接抛出异常，消息将重试
        throw new RuntimeException("Consumer Message exception");
    }
}
// 现在好像变成了ConsumeConcurrentlyStatus
new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) {
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
        return ConsumeConcurrentlyStatus.RECONSUME_LATER;
    }
}
```

消费失败后，不重试配置方式：

集群消费方式下，消息失败后期望消息不充实，需要补货逻辑中的异常，最终返回Action.CommitMessage后，这条消息将不会再重试。

```
public class MessageListenerImpl implements MessageListener{
    @Override
    public Action consume(Message message,ConsumerContext context){
        // 处理消息
        try{
            doConsumerMessage(message);
        } catch(Exception e){
        }
        return Action.CommitMessage;
    }
}
// 现在好像变成了ConsumeConcurrentlyStatus
new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> list, ConsumeConcurrentlyContext consumeConcurrentlyContext) {
        ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
}
```

如果一个consumer挂掉后，在消息消费次数范围之内，消息将被分配给其他消费者进行消费，曾经消费过的次数依次递增。

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重试次数，重试时间间隔按照如下策略：

- 最大重试次数小于等于16次，则重试时间间隔同上表表述。
- 最大重试次数大于16次，超过16次的重试时间间隔均为每次2小时。

- ```
  consumer.setMaxReconsumeTimes(2);
  ```

注意：

- 消息最大重试次数的设置对相同Group ID下的所有Consumer实例有效。
- 如果只对相同**Group ID**下两个**Consumer**实例中的其中一个设置了**MaxReconsumeTimes**，那么该配置对这两个**Consumer**实例均生效。
- 配置采用覆盖的方式生效，即最后启动的Consumer实例会覆盖之前的启动实例的配置。

获取消息重试次数：消费者收到消息后，可按照如下方式获取消息的重试次数：

```
ext.getReconsumeTimes()
```

（5）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地消费该消息，此时，消费队列RocketMQ不会立刻将消息丢弃，而是将其发送该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 Message），存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。

1）死信特征

死信消息具有以下特性：

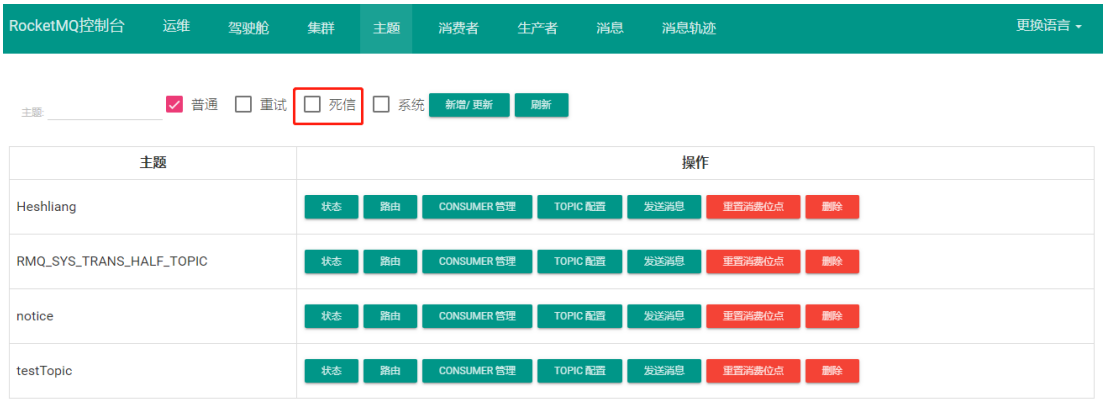
- 不会再被消费者正常消费
- 有效期与正常消息相同，均为3天，3天后被自动删除。因此，请在死信消息产生后的3天内及时处理。

死信队列有以下特性：

- 一个死信队列对应一个Group Id，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，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Group ID产生的所有死信消息，**不论该消息属于哪个Topic**。

2）查看死信消息

- 在控制台查询出现死信队列的主题信息



- 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该消息，因此，通常需要对其进行特殊处理。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。把死信队列当做普通队列进行消费。

```
// 其余配置为完全相同
consumer.subscribe("%DLQ%consumerGroup","*");
```

（6）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。

1）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，这个重复简答可以概括为以下情况：

- 发送消息时重复：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，消费者后续会收到两条内容相同并且Message ID也想通的消息。
- 投递时消息重复：消息消费场景下，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，消费者后续会收到两条内容相同并且Message ID也相同的消息。
- 负载均衡消息重复（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或客户端重启、扩容或缩容时，会触发Rebalance，此时消费者可能会收到重复消息。

2）处理方式

因为Message ID有可能出现冲突（重复）的情况，所以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。

```
// 消费者
Message msg = new Message();
msg.setKey("ORDERID_100");
SendResult sendResult = producer.send(message);
// 生产者
consumer.subscribe("topic","tag",new MessageListener(){
    public Action consume(Message message, ConsumeContext context){
        String key = message.getKey();
        // 业务唯一标识的key做幂等处理
        ...
    }
});
```

7.源码分析

8.RocketMQ console

（1）查看消费者消费情况：“消费者”-> 选择某个订阅组“消费详情”，查看对应消费者消费情况，包括正常消费和重试消费。

（2）查看topic消费情况：“主题”->勾选“普通”->“CONSUMER管理”，查看消费的到哪儿了。

（3）修改死信队列权限：“主题”-> 勾选“死信”->“Topic配置”->修改“perm”为对应值并提交。

（4）查看消费者的死信队列：“主题”-> 勾选“死信”->“CONSUMER管理”，就可以查看消费者对死信队列的消费情况。