图灵：楼兰
你的神秘技术宝藏

前面的部分我们都是为了快速的体验RocketMQ的搭建和使用。这一部分，我们慢下来，总结并学习下RocketMQ底层的一些概念以及原理，为后面的深入学习做准备。

# 一、基础概念：

这一部分我们先来总结下RocketMQ的一些重要的基础概念：

## 1 消息模型（Message Model）

RocketMQ主要由 Producer、Broker、Consumer 三部分组成，其中Producer 负责生产消息，Consumer 负责消费消息，Broker 负责存储消息。Broker 在实际部署过程中对应一台服务器，每个 Broker 可以存储多个Topic的消息，每个Topic的消息也可以分片存储于不同的 Broker。Message Queue 用于存储消息的物理地址，每个Topic中的消息地址存储于多个 Message Queue 中。ConsumerGroup 由多个Consumer 实例构成。

## 2 消息生产者（Producer）

负责生产消息，一般由业务系统负责生产消息。一个消息生产者会把业务应用系统里产生的消息发送到broker服务器。RocketMQ提供多种发送方式，同步发送、异步发送、顺序发送、单向发送。同步和异步方式均需要Broker返回确认信息，单向发送不需要。

生产者中，会把同一类Producer组成一个集合，叫做生产者组，这类Producer发送同一类消息且发送逻辑一致。如果发送的是事务消息且原始生产者在发送之后崩溃，则Broker服务器会联系同一生产者组的其他生产者实例以提交或回溯消费。

# 3 消息消费者（Consumer）

负责消费消息，一般是后台系统负责异步消费。一个消息消费者会从Broker服务器拉取消息、并将其提供给应用程序。从用户应用的角度而言提供了两种消费形式：拉取式消费、推动式消费。

- 拉取式消费的应用通常主动调用Consumer的拉消息方法从Broker服务器拉消息、主动权由应用控制。一旦获取了批量消息，应用就会启动消费过程。
- 推动式消费模式下Broker收到数据后会主动推送给消费端，该消费模式一般实时性较高。

消费者同样会把同一类Consumer组成一个集合，叫做消费者组，这类Consumer通常消费同一类消息且消费逻辑一致。消费者组使得在消息消费方面，实现负载均衡和容错的目标变得非常容易。要注意的是，消费者组的消费者实例必须订阅完全相同的Topic。RocketMQ 支持两种消息模式：集群消费（Clustering）和广播消费（Broadcasting）。

- 集群消费模式下,相同Consumer Group的每个Consumer实例平均分摊消息。
- 广播消费模式下，相同Consumer Group的每个Consumer实例都接收全量的消息。

# 4 主题（Topic）

表示一类消息的集合，每个主题包含若干条消息，每条消息只能属于一个主题，是RocketMQ进行消息订阅的基本单位。

同一个Topic下的数据，会分片保存到不同的Broker上，而每一个分片单位，就叫做MessageQueue。MessageQueue是生产者发送消息与消费者消费消息的最小单位。

# 5 代理服务器（Broker Server）

消息中转角色，负责存储消息、转发消息。代理服务器在RocketMQ系统中负责接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备。代理服务器也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等。

Broker Server是RocketMQ真正的业务核心，包含了多个重要的子模块：

- Remoting Module：整个Broker的实体，负责处理来自clients端的请求。
- Client Manager：负责管理客户端(Producer/Consumer)和维护Consumer的Topic订阅信息
- Store Service：提供方便简单的API接口处理消息存储到物理硬盘和查询功能。
- HA Service：高可用服务，提供Master Broker 和 Slave Broker之间的数据同步功能。
- Index Service：根据特定的Message key对投递到Broker的消息进行索引服务，以提供消息的快速查询。

而Broker Server要保证高可用需要搭建主从集群架构。RocketMQ中有两种Broker架构模式：

- 普通集群：

这种集群模式下会给每个节点分配一个固定的角色，master负责响应客户端的请求，并存储消息。slave则只负责对master的消息进行同步保存，并响应部分客户端的读请求。消息同步方式分为同步同步和异步同步。

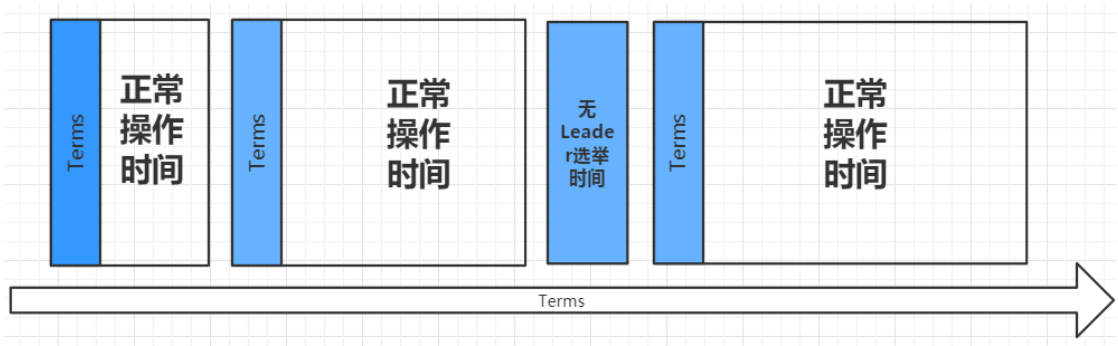
这种集群模式下各个节点的角色无法进行切换，也就是说，master节点挂了，这一组Broker就不可用了。

- Dledger高可用集群：

Dledger是RocketMQ自4.5版本引入的实现高可用集群的一项技术。这个模式下的集群会随机选出一个节点作为master，而当master节点挂了后，会从slave中自动选出一个节点升级成为master。

Dledger技术做的事情：1、接管Broker的CommitLog消息存储 2、从集群中选举出master节点 3、完成master节点往slave节点的消息同步。

Dledger的关键部分是在他的节点选举上。Dledger是使用Raft算法来进行节点选举的。这里简单介绍下Raft算法的选举过程:

> 首先：每个节点有三个状态，Leader，follower和candidate(候选人)。正常运行的情况下，集群中会有一个leader，其他都是follower，follower只响应Leader和Candidate的请求，而客户端的请求全部由Leader处理，即使有客户端请求到了一个follower，也会将请求转发到leader。
>
> 集群刚启动时，每个节点都是follower状态，之后集群内部会发送一个timeout信号，所有follower就转成candidate去拉取选票，获得大多数选票的节点选为leader，其他候选人转为follower。如果一个timeout信号发出时，没有选出leader，将会重新开始一次新的选举。而Leader节点会往其他节点发送心跳信号，确认他的leader状态。
>
> -- 然后会启动定时器，如果在指定时间内没有收到Leader的心跳，就会转为Candidate状态，然后向其他成员发起投票请求，如果收到半数以上成员的投票，则Candidate会晋升为Leader。然后leader也有可能会退化成follower。
>
> 然后，在Raft协议中，会将时间分为一些任意时间长度的时间片段，叫做term。term会使用一个全局唯一，连续递增的编号作为标识，也就是起到了一个逻辑时钟的作用。
>
> 
> 
>
> 在每一个term时间片里，都会进行新的选举，每一个Candidate都会努力争取成为leader。获得票数最多的节点就会被选举为Leader。被选为Leader的这个节点，在一个term时间片里就会保持leader状态。这样，就会保证在同一时间段内，集群中只会有一个Leader。在某些情况下，选票可能会被各个节点瓜分，形成不了多数派，那这个term可能直到结束都没有leader，直到下一个term再重新发起选举，这也就没有了Zookeeper中的脑裂问题。而在每次重新选举的过程中，leader也有可能会退化成为follower。也就是说，在这个集群中， leader节点是会不断变化的。
>
> 然后，每次选举的过程中，每个节点都会存储当前term编号，并在节点之间进行交流时，都会带上自己的term编号。如果一个节点发现他的编号比另外一个小，那么他就会将自己的编号更新为较大的那一个。而如果leader或者candidate发现自己的编号不是最新的，他就会自动转成follower。如果接收到的请求term编号小于自己的编号，term将会拒绝执行。
>
> 在选举过程中，Raft协议会通过心跳机制发起leader选举。节点都是从follower状态开始的，如果收到了来自leader或者candidate的心跳RPC请求，那他就会保持follower状态，避免争抢成为candidate。而leader会往其他节点发送心跳信号，来确认自己的地位。如果follower一段时间(两个timeout信号)内没有收到Leader的心跳信号，他就会认为leader挂了，发起新一轮选举。

> 选举开始后，每个follower会增加自己当前的term，并将自己转为candidate。然后向其他节点发起投票请求，请求时会带上自己的编号和term，也就是说都会默认投自己一票。之后candidate状态可能会发生以下三种变化：
>
> - **赢得选举，成为leader**： 如果它在一个term内收到了大多数的选票，将会在接下的剩余term时间内称为leader，然后就可以通过发送心跳确立自己的地位。(每一个server在一个term内只能投一张选票，并且按照先到先得的原则投出)
> - **其他节点成为leader**： 在等待投票时，可能会收到其他server发出心跳信号，说明其他leader已经产生了。这时通过比较自己的term编号和RPC过来的term编号，如果比对方大，说明leader的term过期了，就会拒绝该RPC,并继续保持候选人身份; 如果对方编号不比自己小,则承认对方的地位,转为follower。
> - **选票被瓜分,选举失败**: 如果没有candidate获取大多数选票, 则没有leader产生, candidate们等待超时后发起另一轮选举. 为了防止下一次选票还被瓜分,必须采取一些额外的措施, raft采用随机election timeout(随机休眠时间)的机制防止选票被持续瓜分。通过将timeout随机设为一段区间上的某个值, 因此很大概率会有某个candidate率先超时然后赢得大部分选票。
>
> 所以以三个节点的集群为例，选举过程会是这样的：
>
> 1. 集群启动时，三个节点都是follower，发起投票后，三个节点都会给自己投票。这样一轮投票下来，三个节点的term都是1，是一样的，这样是选举不出Leader的。
> 2. 当一轮投票选举不出Leader后，三个节点会进入随机休眠，例如A休眠1秒，B休眠3秒，C休眠2秒。
> 3. 一秒后，A节点醒来，会把自己的term加一票，投为2。然后2秒时，C节点醒来，发现A的term已经是2，比自己的1大，就会承认A是Leader，把自己的term也更新为2。实际上这个时候，A已经获得了集群中的多数票，2票，A就会被选举成Leader。这样，一般经过很短的几轮选举，就会选举出一个Leader来。
> 4. 到3秒时，B节点会醒来，他也同样会承认A的term最大，他是Leader，自己的term也会更新为2。这样集群中的所有Candidate就都确定成了leader和follower.
> 5. 然后在一个任期内，A会不断发心跳给另外两个节点。当A挂了后，另外的节点没有收到A的心跳，就会都转化成Candidate状态，重新发起选举。

Dledger还会采用Raft协议进行多副本的消息同步：

> 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。

最后，关于Dledger以及Raft协议的更底层的详细资料，后续会有一个分布式一致性协议的专题，将会结合其他分布式一致性算法做统一讲解，这里就不深入展开了。

# 6 名字服务（Name Server）

名称服务充当路由消息的提供者。Broker Server会在启动时向所有的Name Server注册自己的服务信息，并且后续通过心跳请求的方式保证这个服务信息的实时性。生产者或消费者能够通过名字服务查找各主题相应的Broker IP列表。多个Namesrv实例组成集群，但相互独立，没有信息交换。

这种特性也就意味着NameServer中任意的节点挂了，只要有一台服务节点正常，整个路由服务就不会有影响。当然，这里不考虑节点的负载情况。

# 7 消息（Message）

消息系统所传输信息的物理载体，生产和消费数据的最小单位，每条消息必须属于一个主题Topic。RocketMQ中每个消息拥有唯一的Message ID，且可以携带具有业务标识的Key。系统提供了通过Message ID和Key查询消息的功能。

并且Message上有一个为消息设置的标志，Tag标签。用于同一主题下区分不同类型的消息。来自同一业务单元的消息，可以根据不同业务目的在同一主题下设置不同标签。标签能够有效地保持代码的清晰度和连贯性，并优化RocketMQ提供的查询系统。消费者可以根据Tag实现对不同子主题的不同消费逻辑，实现更好的扩展性。

> 关于Message的更详细字段，在源码的docs/cn/best_practice.md中有详细介绍。

# 二、消息存储

## 1、何时存储消息

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. MQ收到一条消息后，需要向生产者返回一个ACK响应，并将消息存储起来。
2. MQ Push一条消息给消费者后，等待消费者的ACK响应，需要将消息标记为已消费。如果没有标记为消费，MQ会不断的尝试往消费者推送这条消息。
3. MQ需要定期删除一些过期的消息，这样才能保证服务一直可用。

## 2、消息存储介质

RocketMQ采用的是类似于Kafka的文件存储机制，即直接用磁盘文件来保存消息，而不需要借助MySQL这一类索引工具。

### 2.1磁盘保存文件慢吗?

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！因为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快多个数量级。RocketMQ的消息用顺序写,保证了消息存储的速度。

### 2.2零拷贝技术加速文件读写

Linux操作系统分为【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 把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。

这两个看似简单的操作，实际进行了4 次数据复制，分别是：

1. 从磁盘复制数据到内核态内存；
2. 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。

而通过使用mmap的方式，可以省去向用户态的内存复制，提高速度。这种机制在Java中是通过NIO包中的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，提高消息存盘和网络发送的速度。

> 这里需要注意的是，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1.5~2G 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日志数据文件为1G的原因了

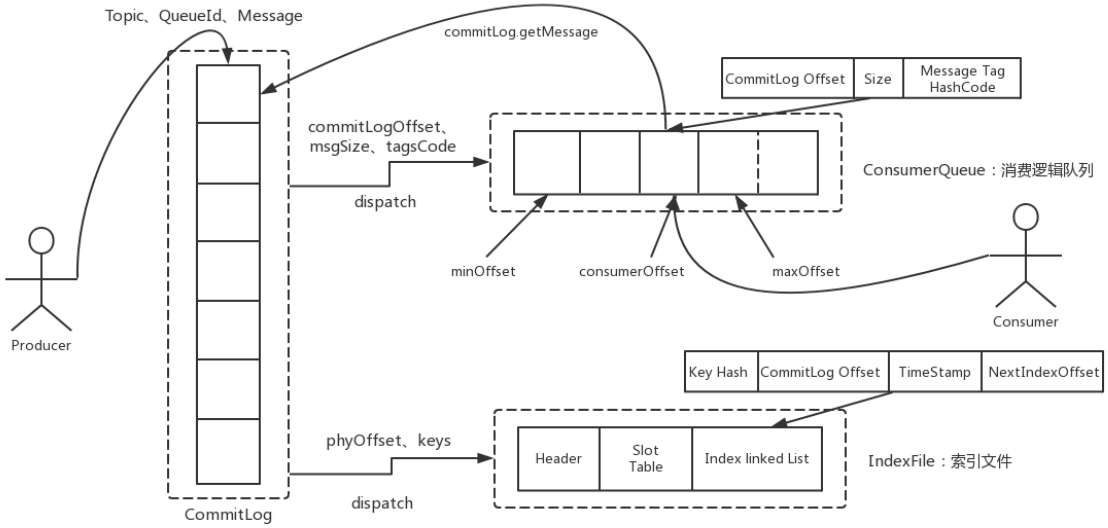
> 关于零拷贝，JAVA的NIO中提供了两种实现方式，mmap和sendfile，其中mmap适合比较小的文件，而sendfile适合传递比较大的文件。同学们自行回顾下这部分的内容。

# 3 消息存储结构

RocketMQ消息的存储分为三个部分：

- CommitLog：存储消息的元数据。所有消息都会顺序存入到CommitLog文件当中。CommitLog由多个文件组成，每个文件固定大小1G。以第一条消息的偏移量为文件名。
- ConsumerQueue：存储消息在CommitLog的索引。一个MessageQueue一个文件，记录当前MessageQueue被哪些消费者组消费到了哪一条CommitLog。
- 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

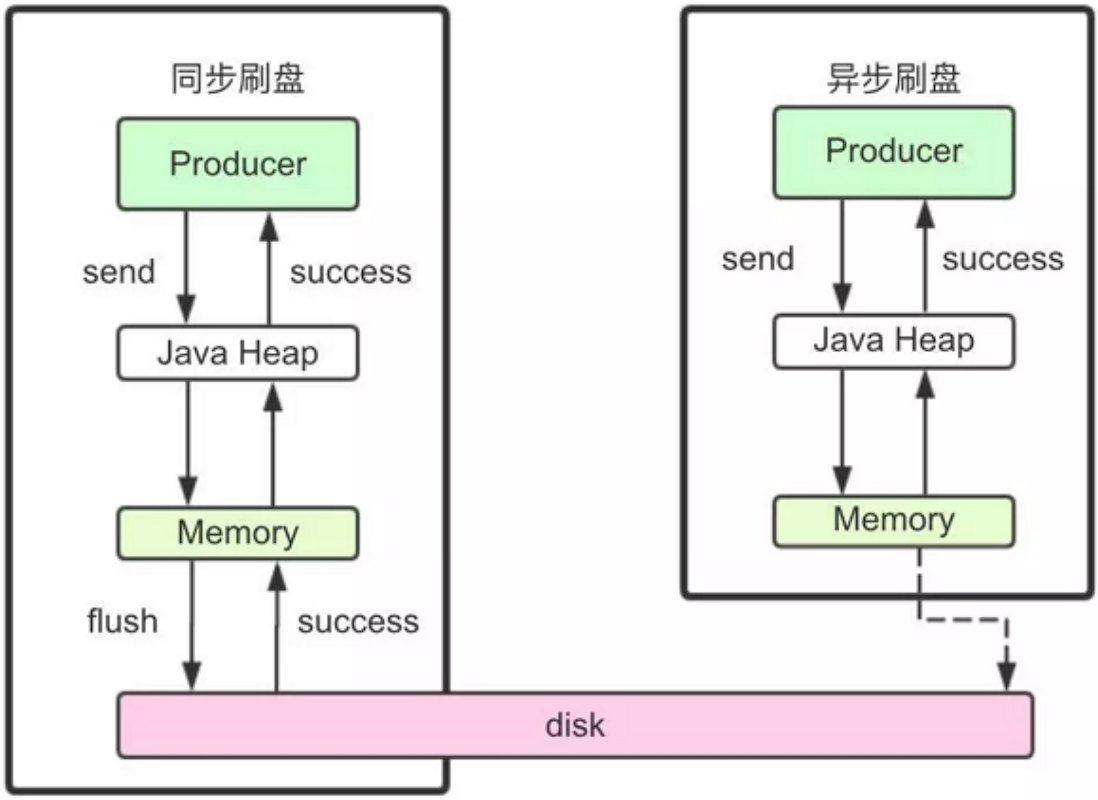
整体的消息存储结构如下图：




> 还记得我们在搭建集群时都特意指定的文件存储路径吗？现在可以上去看看这些文件都是什么样子。还有哪些落盘的文件？
>
> 另外还有几个文件可以了解下。
>
> abort：这个文件是RocketMQ用来判断程序是否正常关闭的一个标识文件。正常情况下，会在启动时创建，而关闭服务时删除。但是如果遇到一些服务器宕机，或者kill -9这样一些非正常关闭服务的情况，这个abort文件就不会删除，因此RocketMQ就可以判断上一次服务是非正常关闭的，后续就会做一些数据恢复的操作。
>
> checkpoint：数据存盘检查点
>
> config/*.json：这些文件是将RocketMQ的一些关键配置信息进行存盘保存。例如Topic配置、消费者组配置、消费者组消息偏移量Offset 等等一些信息。

# 4 刷盘机制

RocketMQ需要将消息存储到磁盘上，这样才能保证断电后消息不会丢失。同时这样才可以让存储的消息量可以超出内存的限制。RocketMQ为了提高性能，会尽量保证磁盘的顺序写。消息在写入磁盘时，有两种写磁盘的方式，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的线程，返回消息写 成功的状态。

- 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，统一触发写磁盘动作，快速写入。

- 配置方式：

  刷盘方式是通过Broker配置文件里的flushDiskType 参数设置的，这个参数被配置成SYNC_FLUSH、ASYNC_FLUSH中的 一个。

# 5 消息主从复制

如果Broker以一个集群的方式部署，会有一个master节点和多个slave节点，消息需要从Master复制到Slave上。而消息复制的方式分为同步复制和异步复制。

- 同步复制：

同步复制是等Master和Slave都写入消息成功后才反馈给客户端写入成功的状态。

在同步复制下，如果Master节点故障，Slave上有全部的数据备份，这样容易恢复数据。但是同步复制会增大数据写入的延迟，降低系统的吞吐量。

- 异步复制：

异步复制是只要master写入消息成功，就反馈给客户端写入成功的状态。然后再异步的将消息复制给Slave节点。

在异步复制下，系统拥有较低的延迟和较高的吞吐量。但是如果master节点故障，而有些数据没有完成复制，就会造成数据丢失。

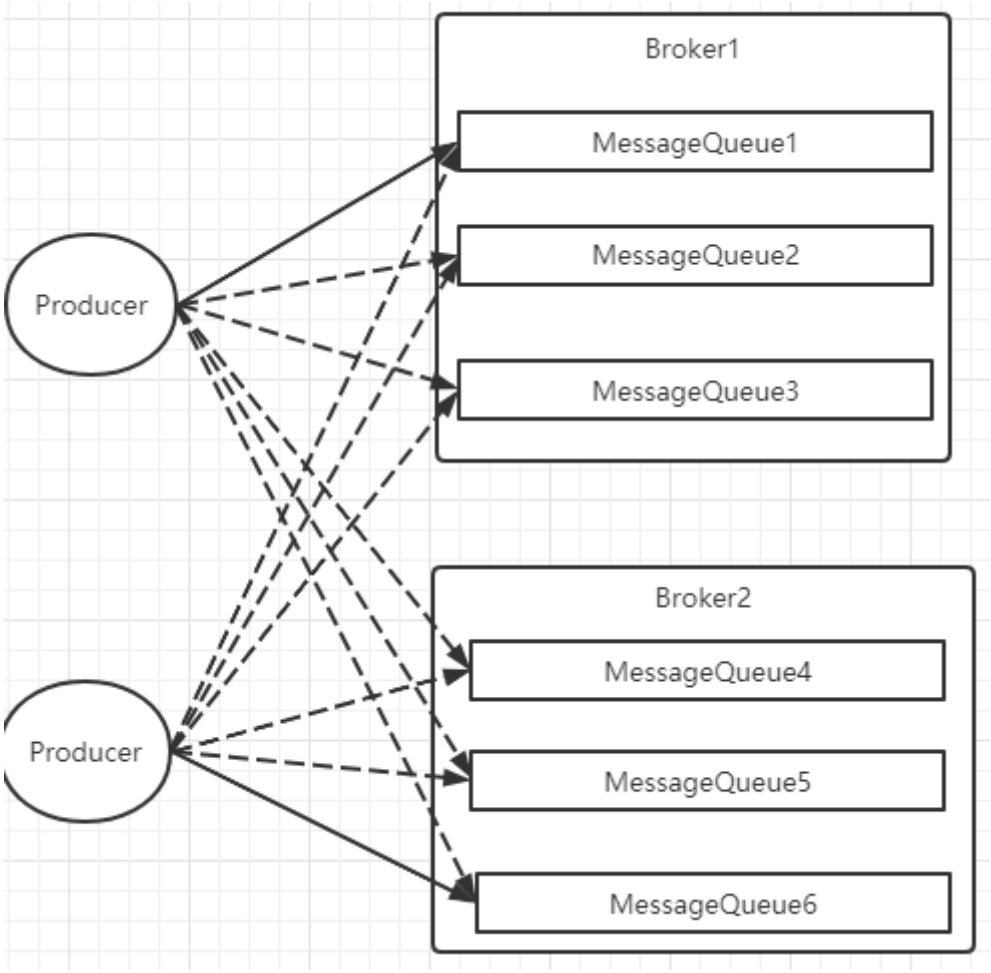
- 配置方式：

消息复制方式是通过Broker配置文件里的brokerRole参数进行设置的，这个参数可以被设置成ASYNC_MASTER、 SYNC_MASTER、SLAVE三个值中的一个。

# 6 负载均衡

## 6.1Producer负载均衡

Producer发送消息时，默认会轮询目标Topic下的所有MessageQueue，并采用递增取模的方式往不同的MessageQueue上发送消息，以达到让消息平均落在不同的queue上的目的。而由于MessageQueue是分布在不同的Broker上的，所以消息也会发送到不同的broker上。




同时生产者在发送消息时，可以指定一个MessageQueueSelector。通过这个对象来将消息发送到自己指定的MessageQueue上。这样可以保证消息局部有序。

## 6.2 Consumer负载均衡

Consumer也是以MessageQueue为单位来进行负载均衡。分为集群模式和广播模式。

### 1、集群模式

在集群消费模式下，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的数量和实例的数量平均分配queue给每个实例。

每次分配时，都会将MessageQueue和消费者ID进行排序后，再用不同的分配算法进行分配。内置的分配的算法共有六种，分别对应AllocateMessageQueueStrategy下的六种实现类，可以在consumer中直接set来指定。默认情况下使用的是最简单的平均分配策略。

- AllocateMachineRoomNearby： 将同机房的Consumer和Broker优先分配在一起。

这个策略可以通过一个machineRoomResolver对象来定制Consumer和Broker的机房解析规则。然后还需要引入另外一个分配策略来对同机房的Broker和Consumer进行分配。一般也就用简单的平均分配策略或者轮询分配策略。

> 感觉这东西挺鸡肋的，直接给个属性指定机房不是挺好的吗。

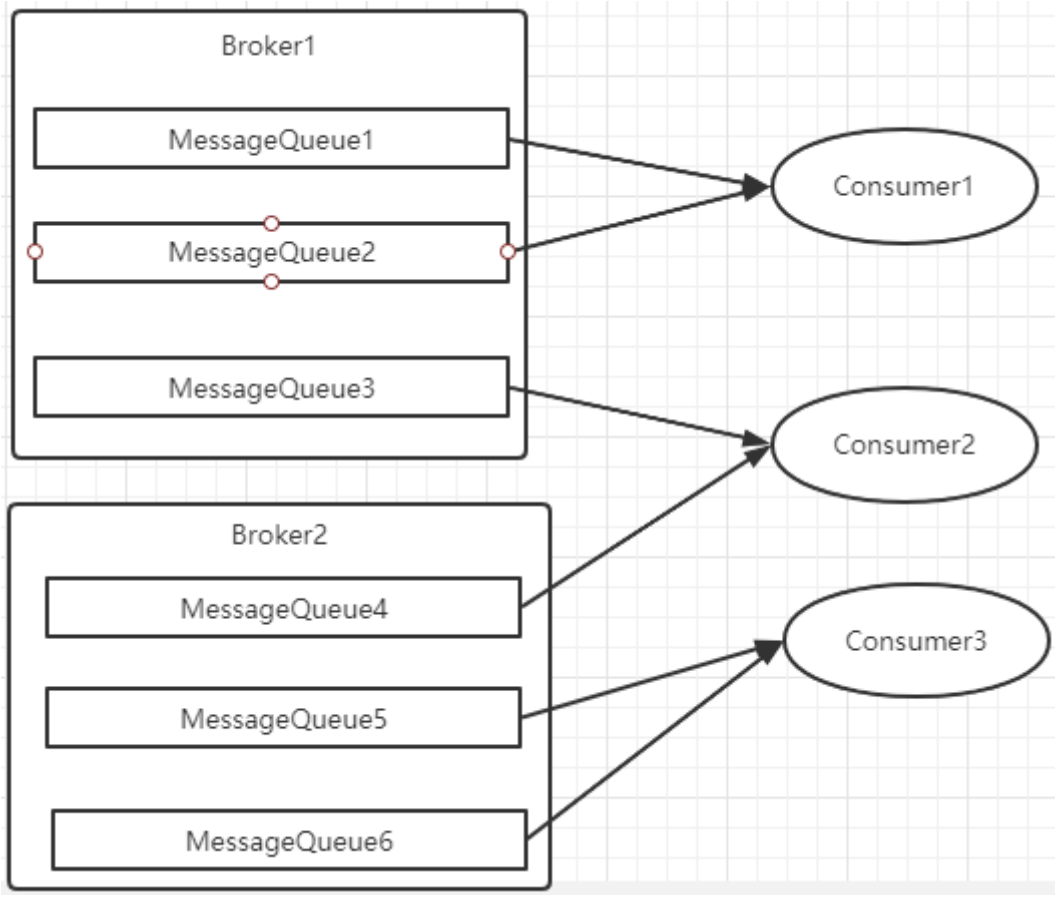
源码中有测试代码AllocateMachineRoomNearByTest。

在示例中：Broker的机房指定方式： messageQueue.getBrokerName().split("-")[0]，而Consumer的机房指定方式：clientID.split("-")[0]

clinetID的构建方式：见ClientConfig.buildMQClientId方法。按他的测试代码应该是要把clientIP指定为IDC1-CID-0这样的形式。

- AllocateMessageQueueAveragely：平均分配。将所有MessageQueue平均分给每一个消费者
- AllocateMessageQueueAveragelyByCircle： 轮询分配。轮流的给一个消费者分配一个MessageQueue。
- AllocateMessageQueueByConfig： 不分配，直接指定一个messageQueue列表。类似于广播模式，直接指定所有队列。
- AllocateMessageQueueByMachineRoom：按逻辑机房的概念进行分配。又是对BrokerName和ConsumerIdc有定制化的配置。
- AllocateMessageQueueConsistentHash。源码中有测试代码AllocateMessageQueueConsitentHashTest。这个一致性哈希策略只需要指定一个虚拟节点数，是用的一个哈希环的算法，虚拟节点是为了让Hash数据在换上分布更为均匀。

例如平均分配时的分配情况是这样的：




### 2、广播模式

广播模式下，每一条消息都会投递给订阅了Topic的所有消费者实例，所以也就没有消息分配这一说。而在实现上，就是在Consumer分配Queue时，所有Consumer都分到所有的Queue。

# 7、消息重试

首先对于广播模式的消息， 是不存在消息重试的机制的，即消息消费失败后，不会再重新进行发送，而只是继续消费新的消息。

而对于普通的消息，当消费者消费消息失败后，你可以通过设置返回状态达到消息重试的结果。

## 1、如何让消息进行重试

集群消费方式下，消息消费失败后期望消息重试，需要在消息监听器接口的实现中明确进行配置。可以有三种配置方式：

- 返回Action.ReconsumeLater-推荐
- 返回null
- 抛出异常

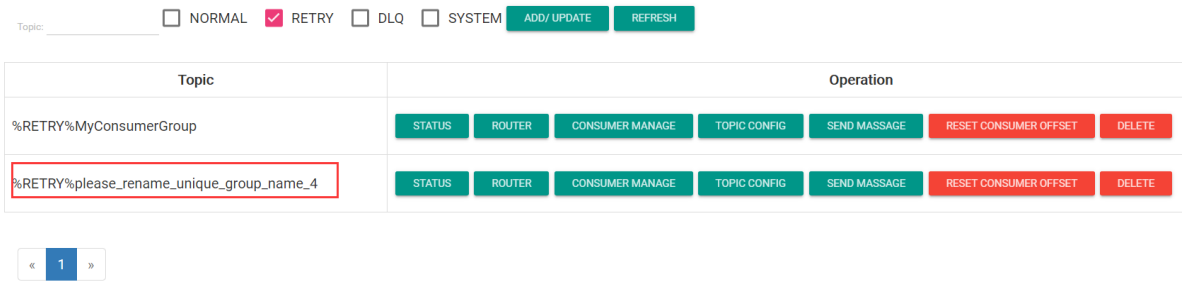
```java
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        //处理消息
        doConsumeMessage(message);
        //方式1：返回 Action.ReconsumeLater，消息将重试
        return Action.ReconsumeLater;
        //方式2：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 exceotion");
    }
}
```

如果希望消费失败后不重试，可以直接返回Action.CommitMessage。

```java
public class MessageListenerImpl implements MessageListener {
    @Override
    public Action consume(Message message, ConsumeContext context) {
        try {
            doConsumeMessage(message);
        } catch (Throwable e) {
            //捕获消费逻辑中的所有异常，并返回 Action.CommitMessage;
            return Action.CommitMessage;
        }
        //消息处理正常，直接返回 Action.CommitMessage;
        return Action.CommitMessage;
    }
}
```

## 2、重试消息如何处理

重试的消息会进入一个 “%RETRY%”+ConsumeGroup 的队列中。




然后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：

| 重试次数 | 与上次重试的间隔时间 | 重试次数 | 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 |

> 这个重试时间跟延迟消息的延迟级别是对应的。不过取的是延迟级别的后16级别。
>
> messageDelayLevel=1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
>
> 这个重试时间可以将源码中的org.apache.rocketmq.example.quickstart.Consumer里的消息监听器返回状态改为RECONSUME_LATER测试一下。

**重试次数：**

如果消息重试16次后仍然失败，消息将不再投递。转为进入死信队列。

另外一条消息无论重试多少次，这些重试消息的MessageId始终都是一样的。

然后关于这个重试次数，RocketMQ可以进行定制。例如通过consumer.setMaxReconsumeTimes(20);将重试次数设定为20次。当定制的重试次数超过16次后，消息的重试时间间隔均为2小时。

**关于MessageId：**

在老版本的RocketMQ中，一条消息无论重试多少次，这些重试消息的MessageId始终都是一样的。

但是在4.7.1版本中，每次重试MessageId都会重建。

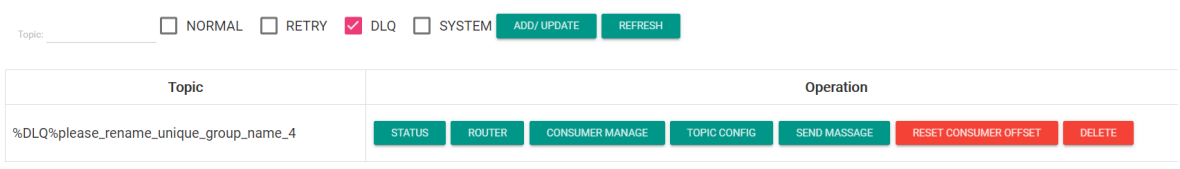
**配置覆盖：**

消息最大重试次数的设置对相同GroupID下的所有Consumer实例有效。并且最后启动的Consumer会覆盖之前启动的Consumer的配置。

# 8、死信队列

当一条消息消费失败，RocketMQ就会自动进行消息重试。而如果消息超过最大重试次数，RocketMQ就会认为这个消息有问题。但是此时，RocketMQ不会立刻将这个有问题的消息丢弃，而会将其发送到这个消费者组对应的一种特殊队列：死信队列。

死信队列的名称是%DLQ%+ConsumGroup




**死信队列的特征：**

- 一个死信队列对应一个ConsumGroup，而不是对应某个消费者实例。
- 如果一个ConsumeGroup没有产生死信队列，RocketMQ就不会为其创建相应的死信队列。
- 一个死信队列包含了这个ConsumeGroup里的所有死信消息，而不区分该消息属于哪个Topic。
- 死信队列中的消息不会再被消费者正常消费。
- 死信队列的有效期跟正常消息相同。默认3天，对应broker.conf中的fileReservedTime属性。超过这个最长时间的消息都会被删除，而不管消息是否消费过。

通常，一条消息进入了死信队列，意味着消息在消费处理的过程中出现了比较严重的错误，并且无法自行恢复。此时，一般需要人工去查看死信队列中的消息，对错误原因进行排查。然后对死信消息进行处理，比如转发到正常的Topic重新进行消费，或者丢弃。

> 注：默认创建出来的死信队列，他里面的消息是无法读取的，在控制台和消费者中都无法读取。这是因为这些默认的死信队列，他们的权限perm被设置成了2:禁读(这个权限有三种 2:禁读，4:禁写,6:可读可写)。需要手动将死信队列的权限配置成6，才能被消费(可以通过mqadmin指定或者web控制台)。

# 9、消息幂等

## 1、幂等的概念

在MQ系统中，对于消息幂等有三种实现语义：

- at most once 最多一次：每条消息最多只会被消费一次
- at least once 至少一次：每条消息至少会被消费一次
- exactly once 刚刚好一次：每条消息都只会确定的消费一次

这三种语义都有他适用的业务场景。

其中，at most once是最好保证的。RocketMQ中可以直接用异步发送、sendOneWay等方式就可以保证。

而at least once这个语义，RocketMQ也有同步发送、事务消息等很多方式能够保证。

而这个exactly once是MQ中最理想也是最难保证的一种语义，需要有非常精细的设计才行。RocketMQ只能保证at least once，保证不了exactly once。所以，使用RocketMQ时，需要由业务系统自行保证消息的幂等性。

> 关于这个问题，官网上有明确的回答：
>
> ### 4. Are messages delivered exactly once?
>
> RocketMQ ensures that all messages are delivered at least once. In most cases, the messages are not repeated.

## 2、消息幂等的必要性

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以下情况：

- 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，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

- 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，消费者后续会收到两条内容相同并且 Message ID 也相同的消息。

- 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。

## 3、处理方式

从上面的分析中，我们知道，在RocketMQ中，是无法保证每个消息只被投递一次的，所以要在业务上自行来保证消息消费的幂等性。

而要处理这个问题，RocketMQ的每条消息都有一个唯一的MessageId，这个参数在多次投递的过程中是不会改变的，所以业务上可以用这个MessageId来作为判断幂等的关键依据。

但是，这个MessageId是无法保证全局唯一的，也会有冲突的情况。所以在一些对幂等性要求严格的场景，最好是使用业务上唯一的一个标识比较靠谱。例如订单ID。而这个业务标识可以使用Message的Key来进行传递。

> 有道云分享链接：
>
> 文档：VIP03-RocketMQ高级原理.md
> 链接：http://note.youdao.com/noteshare?id=dbb91903836704056882 1f7ac4d8438c&sub=6BBCDC2E47D04863A9536179A6AFBEBF