RocketMQ 面试题

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 RocketMQ 的大保健。

而题目的难度，尽量按照从容易到困难的顺序，逐步下去。

> 友情提示：在开始阅读之前，胖友至少对 《RocketMQ —— 角色与术语详解》 有简单的了解。

另外，这个面试题是建立在胖友看过 《精尽【消息队列 】面试题》 。

RocketMQ 是什么?

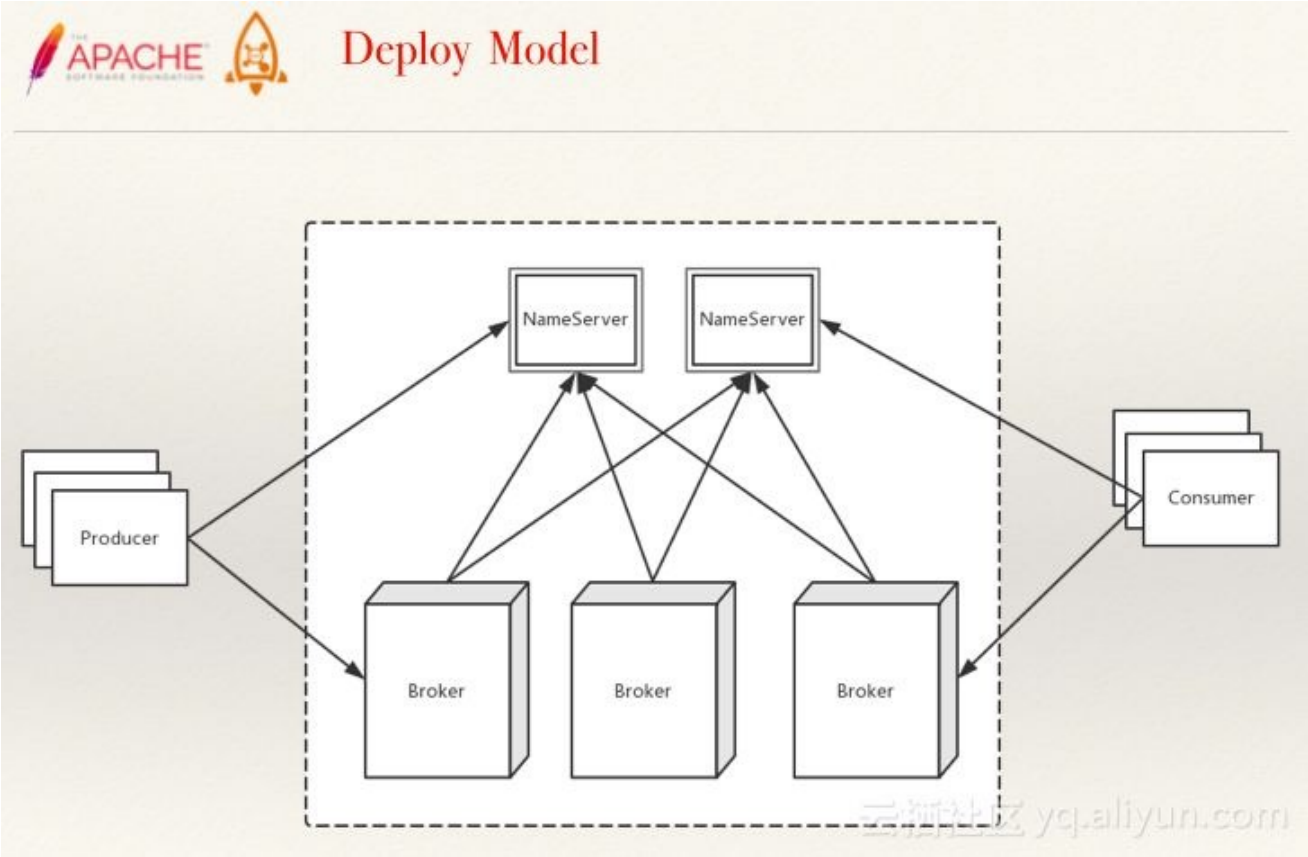
RocketMQ 是阿里巴巴在 2012 年开源的分布式消息中间件，目前已经捐赠给 Apache 软件基金会，并于 2017 年 9 月 25 日成为 Apache 的顶级项目。作为经历过多次阿里巴巴双十一这种“超级工程”的洗礼并有稳定出色表现的国产中间件，以其高性能、低延时和高可靠等特性近年来已经也被越来越多的国内企业使用。

如下是 RocketMQ 产生的原因：

> 淘宝内部的交易系统使用了淘宝自主研发的 Notify 消息中间件，使用 MySQL 作为消息存储媒介，可完全水平扩容，为了进一步降低成本，我们认为存储部分可以进一步优化，2011 年初，Linkin开源了 Kafka 这个优秀的消息中间件，淘宝中间件团队在对 Kafka 做过充分 Review 之后， Kafka 无限消息堆积，高效的持久化速度吸引了我们，但是同时发现这个消息系统主要定位于日志传输，对于使用在淘宝交易、订单、充值等场景下还有诸多特性不满足，为此我们重新用 Java 语言编写了 RocketMQ ，定位于非日志的可靠消息传输（日志场景也OK），目前 RocketMQ 在阿里集团被广泛应用在订单，交易，充值，流计算，消息推送，日志流式处理， binglog 分发等场景。

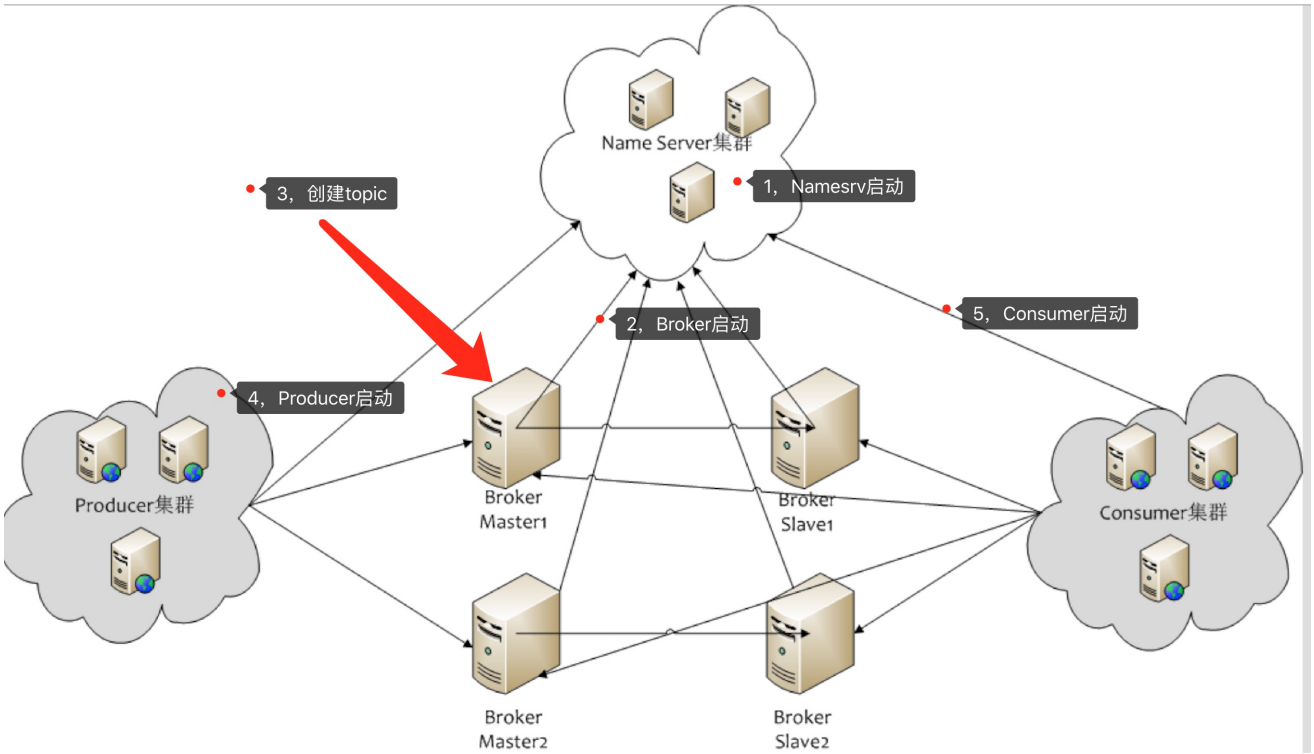
RocketMQ 由哪些角色组成?

如下图所示：



- 生产者（Producer）：负责产生消息，生产者向消息服务器发送由业务应用程序系统生成的消息。
- 消费者（Consumer）：负责消费消息，消费者从消息服务器拉取信息并将其输入用户应用程序。
- 消息服务器（Broker）：是消息存储中心，主要作用是接收来自 Producer 的消息并存储， Consumer 从这里取得消息。
- 名称服务器（NameServer）：用来保存 Broker 相关 Topic 等元信息并给 Producer ，提供 Consumer 查找 Broker 信息。

请描述下 RocketMQ 的整体流程?



- 1、启动 **Namesrv**，Namesrv起 来后监听端口，等待 Broker、Producer、Consumer 连上来，相当于一个路由控制中心。
- 2、**Broker** 启动，跟所有的 Namesrv 保持长连接，定时发送心跳包。

  > 心跳包中，包含当前 Broker 信息(IP+端口等)以及存储所有 Topic 信息。 注册成功后，Namesrv 集群中就有 Topic 跟 Broker 的映射关系。

  - 3、收发消息前，先创建 Topic 。创建 Topic 时，需要指定该 Topic 要存储在 哪些 Broker上。也可以在发送消息时自动创建Topic。
- 4、**Producer** 发送消息。

  > 启动时，先跟 Namesrv 集群中的其中一台建立长连接，并从Namesrv 中获取当前发送的 Topic 存在哪些 Broker 上，然后跟对应的 Broker 建立长连接，直接向 Broker 发消息。

- 5、**Consumer** 消费消息。

  > Consumer 跟 Producer 类似。跟其中一台 Namesrv 建立长连接，获取当前订阅 Topic 存在哪些 Broker 上，然后直接跟 Broker 建立连接通道，开始消费消息。

> ：下面，我们先逐步对 RocketMQ 每个角色进行介绍。
>
> 对不了解 RocketMQ 的胖友来说，可能概念会有点多。淡定~

请说说你对 Namesrv 的了解?

- 1、 Namesrv 用于存储 Topic、Broker 关系信息，功能简单，稳定性高。
  - 多个 Namesrv 之间相互没有通信，单台 Namesrv 宕机不影响其它 Namesrv 与集群。

    > 多个 Namesrv 之间的信息共享，**通过 Broker 主动向多个 Namesrv 都发起心跳。**正如上文所说，Broker 需要跟所有 Namesrv 连接。

    - 即使整个 Namesrv 集群宕机，已经正常工作的 Producer、Consumer、Broker 仍然能正常工作，但新起的 Producer、Consumer、Broker 就无法工作。

      > 这点和 Dubbo 有些不同，不会缓存 Topic 等元信息到本地文件。

- 2、 Namesrv 压力不会太大，平时主要开销是在维持心跳和提供 Topic-Broker 的关系数据。但有一点需要注意，Broker 向 Namesr 发心跳时，会带上当前自己所负责的所有 Topic 信息，如果 Topic 个数太多（万级别），会导致一次心跳中，就 Topic 的数据就几十 M，网络情况差的话，网络传输失败，心跳失败，导致 Namesrv 误认为 Broker 心跳失败。

  > 当然，一般公司，很难达到过万级的 Topic ，因为一方面体量达不到，另一方面 RocketMQ 提供了 Tag 属性。
  >
  > 另外，内网环境网络相对是比较稳定的，传输几十 M 问题不大。同时，如果真的要优化，Broker 可以把心跳包做压缩，再发送给 Namesrv 。不过，这样也会带来 CPU 的占用率的提升。

如何配置 Namesrv 地址到生产者和消费者?

将 Namesrv 地址列表提供给客户端(生产者和消费者)，有四种方法：

- 编程方式，就像 `producer.setNamesrvAddr("ip:port")` 。
- Java 启动参数设置，使用 `rocketmq.namesrv.addr` 。
- 环境变量，使用 `NAMESRV_ADDR` 。
- HTTP 端点，例如说：`http://namesrv.rocketmq.xxx.com` 地址，通过 DNS 解析获得 Namesrv 真正的地址。

请说说你对 Broker 的了解?

- 1、 **高并发读写服务。**Broker的高并发读写主要是依靠以下两点：
    - 消息顺序写，所有 Topic 数据同时只会写一个文件，一个文件满1G ，再写新文件，真正的顺序写盘，使得发消息 TPS 大幅提高。
    - 消息随机读，RocketMQ 尽可能让读命中系统 Pagecache ，因为操作系统访问 Pagecache 时，即使只访问 1K 的消息，系统也会提前预读出更多的数据，在下次读时就可能命中 Pagecache ，减少 IO 操作。
- 2、 **负载均衡与动态伸缩。**
    - 负载均衡：Broker 上存 Topic 信息，Topic 由多个队列组成，队列会平均分散在多个 Broker 上，而 Producer 的发送机制保证消息尽量平均分布到所有队列中，最终效果就是所有消息都平均落在每个 Broker 上。
    - 动态伸缩能力（非顺序消息）：Broker 的伸缩性体现在两个维度：Topic、Broker。
        - Topic 维度：假如一个 Topic 的消息量特别大，但集群水位压力还是很低，就可以扩大该 Topic 的队列数， Topic 的队列数跟发送、消费速度成正比。

          > Topic 的队列数一旦扩大，就无法很方便的缩小。因为，生产者和消费者都是基于相同的队列数来处理。 如果真的想要缩小，只能新建一个 Topic ，然后使用它。 不过，Topic 的队列数，也不存在什么影响的，淡定。

        - Broker 维度：如果集群水位很高了，需要扩容，直接加机器部署 Broker 就可以。Broker 启动后向 Namesrv 注册，Producer、Consumer 通过 Namesrv 发现新Broker，立即跟该 Broker 直连，收发消息。

> 新增的 Broker 想要下线，想要下线也比较麻烦，暂时没特别好的方案。大体的前提是，消费者消费完该 Broker 的消息，生产者不往这个 Broker 发送消息。

- 3、 **高可用 & 高可靠。**
  - 高可用：集群部署时一般都为主备，备机实时从主机同步消息，如果其中一个主机宕机，备机提供消费服务，但不提供写服务。
  - 高可靠：所有发往 Broker 的消息，有同步刷盘和异步刷盘机制。
    - 同步刷盘时，消息写入物理文件才会返回成功。
    - 异步刷盘时，只有机器宕机，才会产生消息丢失，Broker 挂掉可能会发生，但是机器宕机崩溃是很少发生的，除非突然断电。

      > 如果 Broker 挂掉，未同步到硬盘的消息，还在 Pagecache 中呆着。

- 4、 **Broker 与 Namesrv 的心跳机制。**
  - 单个 Broker 跟所有 Namesrv 保持心跳请求，心跳间隔为30秒，心跳请求中包括当前 Broker 所有的 Topic 信息。
  - Namesrv 会反查 Broker 的心跳信息，如果某个 Broker 在 2 分钟之内都没有心跳，则认为该 Broker 下线，调整 Topic 跟 Broker 的对应关系。但此时 Namesrv 不会主动通知Producer、Consumer 有 Broker 宕机。也就说，只能等 Producer、Consumer 下次定时拉取 Topic 信息的时候，才会发现有 Broker 宕机。

从上面的描述中，我们也已经发现 Broker 是 RocketMQ 中最最最复杂的角色，主要包括如下五个模块：

Broker 组件图

- 远程处理模块：是 Broker 的入口，处理来自客户的请求。
- Client Manager ： 管理客户端（生产者/消费者），并维护消费者的主题订阅。
- Store Service ： 提供简单的 API 来存储或查询物理磁盘中的消息。
- HA 服务：提供主节点和从节点之间的数据同步功能。
- 索引服务：通过指定键为消息建立索引，并提供快速的消息查询。

Broker 如何实现消息的存储?

关于 Broker 如何实现消息的存储，这是一个很大的话题，所以建议直接看如下的资料，保持耐心。

- 《读懂这篇文章，你的阿里技术面就可以过关了 | Apache RocketMQ》的如下部分：
  - 「三、RocketMQ的存储模型」
- 《RocketMQ 原理简介》

  的如下部分：
  - 「6.3 数据存储结构」
  - 「6.4 存储目录结构」
  - 「7.1 单机支持 1 万以上持久化队列」
  - 「7.2 刷盘策略」
- 癫狂侠
  - 《消息中间件 —— RocketMQ消息存储（一）》
  - 《消息中间件 —— RocketMQ消息存储（二）》

请说说你对 Producer 的了解?

- 1、**获得 Topic-Broker 的映射关系。**
  - Producer 启动时，也需要指定 Namesrv 的地址，从 Namesrv 集群中选一台建立长连接。如果该 Namesrv 宕机，会自动连其他 Namesrv ，直到有可用的 Namesrv 为止。
  - 生产者每 30 秒从 Namesrv 获取 Topic 跟 Broker 的映射关系，更新到本地内存中。然后再跟 Topic 涉及的所有 Broker 建立长连接，每隔 30 秒发一次心跳。
  - 在 Broker 端也会每 10 秒扫描一次当前注册的 Producer ，如果发现某个 Producer 超过 2 分钟都没有发心跳，则断开连接。
- 2、**生产者端的负载均衡。**
  - 生产者发送时，会自动轮询当前所有可发送的broker，一条消息发送成功，下次换另外一个broker发送，以达到消息平均落到所有的broker上。

    > 这里需要注意一点：假如某个 Broker 宕机，意味生产者最长需要 30 秒才能感知到。在这期间会向宕机的 Broker 发送消息。当一条消息发送到某个 Broker 失败后，会自动再重发 2 次，假如还是发送失败，则抛出发送失败异常。
    >
    > 客户端里会自动轮询另外一个 Broker 重新发送，这个对于用户是透明的。

Producer 发送消息有几种方式?

Producer 发送消息，有三种方式：

1. 同步方式
2. 异步方式
3. Oneway 方式

其中，方式 1 和 2 比较常见，具体使用哪一种方式需要根据业务情况来判断。而方式 3 ，适合大数据场景，允许有一定消息丢失的场景。

请说说你对 Consumer 的了解?

- 1、获得 Topic-Broker 的映射关系。
  - Consumer 启动时需要指定 Namesrv 地址，与其中一个 Namesrv 建立长连接。消费者每隔 30 秒从 Namesrv 获取所有Topic 的最新队列情况，这意味着某个 Broker 如果宕机，客户端最多要 30 秒才能感知。连接建立后，从 Namesrv 中获取当前消费 Topic 所涉及的 Broker，直连 Broker 。
  - Consumer 跟 Broker 是长连接，会每隔 30 秒发心跳信息到Broker 。Broker 端每 10 秒检查一次当前存活的 Consumer ，若发现某个 Consumer 2 分钟内没有心跳，就断开与该 Consumer 的连接，并且向该消费组的其他实例发送通知，触发该消费者集群的负载均衡。
- 2、**消费者端的负载均衡。**根据消费者的消费模式不同，负载均衡方式也不同。

  > 消费者有两种消费模式：集群消费和广播消费。
  >
  > - 集群消费：一个 Topic 可以由同一个消费这分组(Consumer Group)下所有消费者分担消费。 具体例子：假如 TopicA 有 6 个队列，某个消费者分组起了 2 个消费者实例，那么每个消费者负责消费 3 个队列。如果再增加一个消费者分组相同消费者实例，即当前共有 3 个消费者同时消费 6 个队列，那每个消费者负责 2 个队列的消费。

  - 广播消费：每个消费者消费 Topic 下的所有队列。

消费者消费模式有几种?

消费者消费模式有两种：集群消费和广播消费。

📣 **1. 集群消费**

消费者的一种消费模式。一个 Consumer Group 中的各个 Consumer 实例分摊去消费消息，即一条消息只会投递到一个 Consumer Group 下面的一个实例。

- 实际上，每个 Consumer 是平均分摊 Message Queue 的去做拉取消费。例如某个 Topic 有 3 个队列，其中一个 Consumer Group 有 3 个实例（可能是 3 个进程，或者 3 台机器），那么每个实例只消费其中的 1 个队列。
- 而由 Producer 发送消息的时候是轮询所有的队列，所以消息会平均散落在不同的队列上，可以认为队列上的消息是平均的。那么实例也就平均地消费消息了。
- 这种模式下，消费进度的存储会持久化到 Broker 。
- 当新建一个 Consumer Group 时，默认情况下，该分组的消费者会从 min offset 开始重新消费消息。

📣 **2. 广播消费**

消费者的一种消费模式。消息将对一 个Consumer Group 下的各个 Consumer 实例都投递一遍。即即使这些 Consumer 属于同一个Consumer Group ，消息也会被 Consumer Group 中的每个 Consumer 都消费一次。

- 实际上，是一个消费组下的每个消费者实例都获取到了 Topic 下面的每个 Message Queue 去拉取消费。所以消息会投递到每个消费者实例。
- 这种模式下，消费进度会存储持久化到实例本地。

消费者获取消息有几种模式?

消费者获取消息有两种模式：推送模式和拉取模式。

📣 **1. PushConsumer**

推送模式（虽然 RocketMQ 使用的是长轮询）的消费者。消息的能及时被消费。使用非常简单，内部已处理如线程池消费、流控、负载均衡、异常处理等等的各种场景。

- 长轮询，就是我们在 《 精尽【消息队列 】面试题》 提到的，push + pull 模式结合的方式。

📣 **2. PullConsumer**

拉取模式的消费者。应用主动控制拉取的时机，怎么拉取，怎么消费等。主动权更高。但要自己处理各种场景。

决绝绝大多数场景下，我们只会使用 PushConsumer 推送模式。😈 至少目前，暂时还没用过 PullConsumer 。

如何对消息进行重放?

消费位点就是一个数字，把 Consumer Offset 改一下，就可以达到重放的目的了。

什么是顺序消息？如何实现?

消费消息的顺序要同发送消息的顺序一致。由于 Consumer 消费消息的时候是针对 Message Queue 顺序拉取并开始消费，且一条 Message Queue 只会给一个消费者（集群模式下），所以能够保证同一个消费者实例对于 Queue 上消息的消费是顺序地开始消费（不一定顺序消费完成，因为消费可能并行）。

- Consumer ： 在 RocketMQ 中，顺序消费主要指的是都是 Queue 级别的局部顺序。这一类消息为满足顺序性，必须 Producer 单线程顺序发送，且发送到同一个队列，这样 Consumer 就可以按照 Producer 发送的顺序去消费消息。

- Producer：生产者发送的时候可以用 MessageQueueSelector 为某一批消息（通常是有相同的唯一标示id）选择同一个 Queue，则这一批消息的消费将是顺序消息（并由同一个consumer完成消息）。或者 Message Queue 的数量只有 1，但这样消费的实例只能有一个，多出来的实例都会空跑。

当然，上面的文字比较绕，总的来说，RocketMQ 提供了两种顺序级别：

- 普通顺序消息： Producer 将相关联的消息发送到相同的消息队列。
- 严格顺序消息： 在【普通顺序消息】的基础上，Consumer 严格顺序消费。

> 也就说，顺序消息包括两块：Producer 的顺序发送，和 Consumer 的顺序消费。

📣 1. 普通顺序消息

顺序消息的一种，正常情况下可以保证完全的顺序消息，但是一旦发生异常，Broker 宕机或重启，由于队列总数发生发化，消费者会触发负载均衡，而默认地负载均衡算法采取哈希取模平均，这样负载均衡分配到定位的队列会发化，使得队列可能分配到别的实例上，则会短暂地出现消息顺序不一致。

如果业务能容忍在集群异常情况（如某个 Broker 宕机或者重启）下，消息短暂的乱序，使用普通顺序方式比较合适。

📣 2. 严格顺序消息

顺序消息的一种，无论正常异常情况都能保证顺序，但是牺牲了分布式 Failover 特性，即 Broker 集群中只要有一台机器不可用，则整个集群都不可用，服务可用性大大降低。

如果服务器部署为同步双写模式，此缺陷可通过备机自动切换为主避免，不过仍然会存在几分钟的服务不可用。（依赖同步双写，主备自动切换，自动切换功能目前并未实现）

📣 小结

目前已知的应用只有数据库 binlog 同步强依赖严格顺序消息，其他应用绝大部分都可以容忍短暂乱序，推荐使用普通的顺序消息。

📣 实现原理

顺序消息的实现，相对比较复杂，想要深入理解的胖友，可以看看 [《RocketMQ 源码分析 —— Message 顺序发送与消费》](#) 。

顺序消息扩容的过程中，如何在不停写的情况下保证消息顺序?

1. 成倍扩容，实现扩容前后，同样的 key，hash 到原队列，或者 hash 到新扩容的队列。
2. 扩容前，记录旧队列中的最大位点。
3. 对于每个 Consumer Group，保证旧队列中的数据消费完，再消费新队列，也即：先对新队列进行禁读即可。

什么是定时消息？如何实现?

定时消息，是指消息发到 Broker 后，不能立刻被 Consumer 消费，要到特定的时间点或者等待特定的时间后才能被消费。

目前，开源版本的 RocketMQ 只支持固定延迟级别的延迟消息，不支持任一时刻的延迟消息。如下表格：

延迟级别	时间
1	1s
2	5s
3	10s
4	30s
5	1m
6	2m
7	3m
8	4m
9	5m
10	6m
11	7m
12	8m
13	9m
14	10m
15	20m
16	30m
17	1h
18	2h

- 可通过配置文件，自定义每个延迟级别对应的延迟时间。当然，这是全局的。
- 如果胖友想要实现任一时刻的延迟消息，比较简单的方式是插入延迟消息到数据库中，然后通过定时任务轮询，到达指定时间，发送到 RocketMQ 中。

实现原理

- 1、 定时消息发送到 Broker 后，会被存储 Topic 为 `SCHEDULE_TOPIC_XXXX` 中，并且所在 Queue 编号为延迟级别 - 1 。

  > 需要 -1 的原因是，延迟级别是从 1 开始的。如果延迟级别为 0 ，意味着无需延迟。

- 2、Broker 针对每个 `SCHEDULE_TOPIC_XXXX` 的队列，都创建一个定时任务，**顺序**扫描到达时间的延迟消息，重新存储到延迟消息**原始**的 Topic 的**原始** Queue 中，这样它就可以被 Consumer 消费到。此处会有两个问题：
  - 为什么是“**顺序**扫描到达时间的延迟消息”？因为先进 `SCHEDULE_TOPIC_XXXX` 的延迟消息，在其所在的队列，意味着先到达延迟时间。

- 会不会存在重复扫描的情况？每个 `SCHEDULE_TOPIC_XXXX` 的扫描进度，会每 10s 存储到 `config/delayOffset.json` 文件中，所以正常情况下，不会存在重复扫描。如果异常关闭，则可能导致重复扫描。

详细的，胖友可以看看 《RocketMQ 源码分析 —— 定时消息与消息重试》 。

什么是消息重试？如何实现？

消息重试，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 会将消费失败的消息发回 Broker，进入延迟消息队列。即，消费失败的消息，不会立即消费。
- 也就是说，消息重试是构建在定时消息之上的功能。

📣 消息重试的主要流程

1. Consumer 消费失败，将消息发送回 Broker 。
2. Broker 收到重试消息之后置换 Topic ，存储消息。
3. Consumer 会拉取该 Topic 对应的 retryTopic 的消息。
4. Consumer 拉取到 retryTopic 消息之后，置换到原始的 Topic ，把消息交给 Listener 消费。

这里，可能有几个点，胖友会比较懵逼，简单解释下：

1. Consumer 消息失败后，会将消息的 Topic 修改为 `%RETRY%` + Topic 进行，添加 `"RETRY_TOPIC"` 属性为原始 Topic ，然后再返回给 Broker 中。
2. Broker 收到重试消息之后，会有两次修改消息的 Topic 。
    - 首先，会将消息的 Topic 修改为 `%RETRY%` + ConsumerGroup ，因为这个消息是当前消费这分组消费失败，只能被这个消费组所重新消费。😈 注意噢，消费者会默认订阅 Topic 为 `%RETRY%` + ConsumerGroup 的消息。
    - 然后，会将消息的 Topic 修改为 `SCHEDULE_TOPIC_XXXX` ，添加 `"REAL_TOPIC"` 属性为 `%RETRY%` + ConsumerGroup ，因为重试消息需要延迟消费。
3. Consumer 会拉取该 Topic 对应的 retryTopic 的消息，此处的 retryTopic 为 `%RETRY%` + ConsumerGroup 。
4. Consumer 拉取到 retryTopic 消息之后，置换到原始的 Topic ，因为有消息的 `"RETRY_TOPIC"` 属性是原始 Topic ，然后把消息交给 Listener 消费。

😈 有一丢丢复杂，胖友可以在思考思考~详细的，胖友可以看看 《RocketMQ 源码分析 —— 定时消息与消息重试》 。

多次消费失败后，怎么办？

默认情况下，当一条消息被消费失败 16 次后，会被存储到 Topic 为 `"%DLQ%"` + ConsumerGroup 到死信队列。

为什么 Topic 是 `"%DLQ%"` + ConsumerGroup 呢？因为，是这个 ConsumerGroup 对消息的消费失败，所以 Topic 里要以 ConsumerGroup 为维度。

后续，我们可以通过订阅 `"%DLQ%"` + ConsumerGroup ，做相应的告警。

什么是事务消息？如何实现？

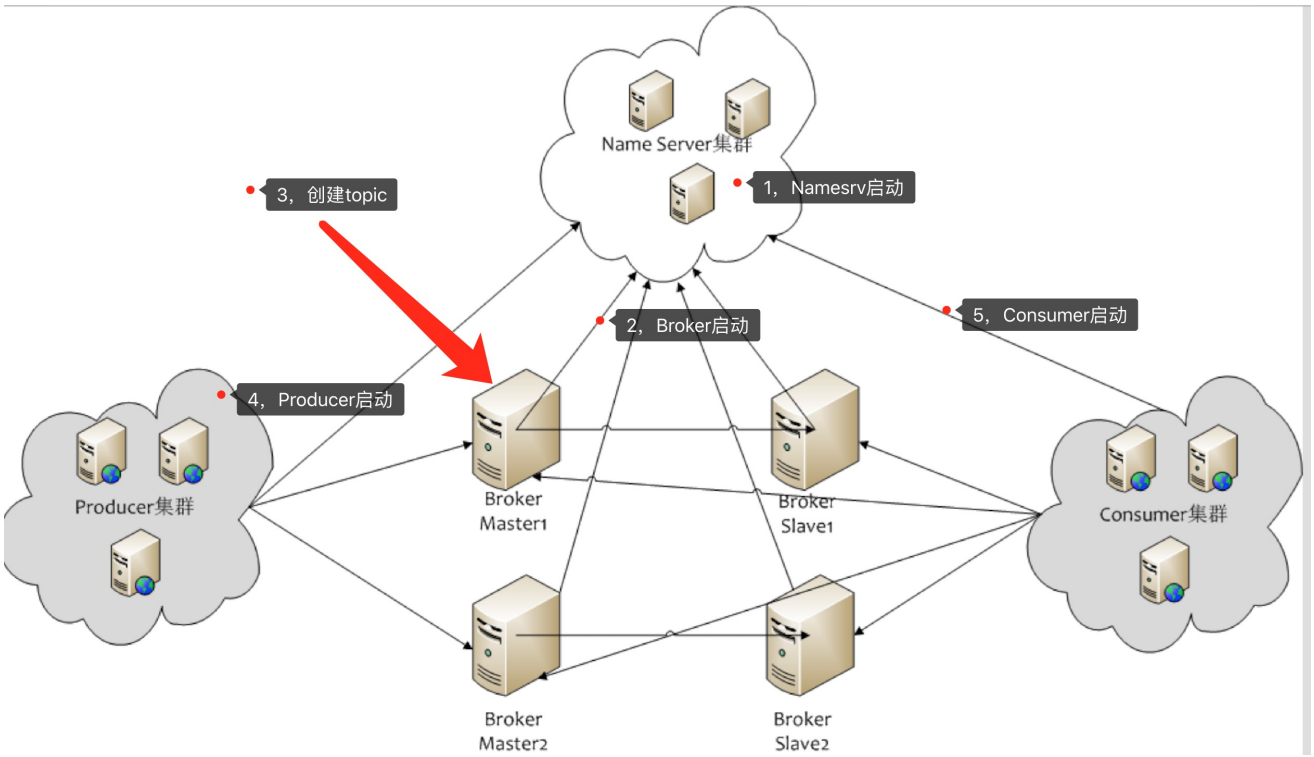
关于事务消息的概念和原理，胖友可以看看官方对这块的解答，即 《RocketMQ 4.3 正式发布，支持分布式事务》 的 「四 事务消息」 小节。

16 年的时候，基于 RocketMQ 早期的版本，写了 《RocketMQ 源码分析 —— 事务消息》 文章，虽然 RocketMQ 版本不太一样，但是大体的思路是差不多的，可以帮助胖友更容易的读懂事务消息相关的源码。

- 简单看了下最新版本的 RocketMQ 的事务代码，新增了 `RMQ_SYS_TRANS_HALF_TOPIC` 和 `RMQ_SYS_TRANS_OP_HALF_TOPIC` 两个队列。
  - Producer 发送 PREPARED Message 到 Broker 后，先存储到 `RMQ_SYS_TRANS_HALF_TOPIC` 队列中。
  - Producer 提交或回滚 PREPARED Message 时，会添加一条消息到 `RMQ_SYS_TRANS_OP_HALF_TOPIC` 队列中，标记这个消息已经处理。
  - Producer 提交 PREPARED Message 时，会将当前消息存储到原 Topic 的队列中，从而该消息能够被 Consumer 拉取消费。

如何实现 RocketMQ 高可用?

在 「RocketMQ 由哪些角色组成？」 中，我们看到 RocketMQ 有四个角色，需要考虑每个角色的高可用。



1. Producer

- 1、Producer 自身在应用中，所以无需考虑高可用。
- 2、Producer 配置多个 Namesrv 列表，从而保证 Producer 和 Namesrv 的连接高可用。并且，会从 Namesrv 定时拉取最新的 Topic 信息。
- 3、Producer 会和所有 Consumer 直连，在发送消息时，会选择一个 Broker 进行发送。如果发送失败，则会使用另外一个 Broker 。
- 4、Producer 会定时向 Broker 心跳，证明其存活。而 Broker 会定时检测，判断是否有 Producer 异常下线。

2. Consumer

- 1、Consumer 需要部署多个节点，以保证 Consumer 自身的高可用。当相同消费者分组中有新的 Consumer 上线，或者老的 Consumer 下线，会重新分配 Topic 的 Queue 到目前消费分组的 Consumer 们。

- 2、Consumer 配置多个 Namesrv 列表，从而保证 Consumer 和 Namesrv 的连接高可用。并且，会从 Consumer 定时拉取最新的 Topic 信息。
- 3、Consumer 会和所有 Consumer 直连，消费相应分配到的 Queue 的消息。如果消费失败，则会发回消息到 Broker 中。
- 4、Consumer 会定时向 Broker 心跳，证明其存活。而 Broker 会定时检测，判断是否有 Consumer 异常下线。

📣 3. Namesrv

- 1、Namesrv 需要部署多个节点，以保证 Namesrv 的高可用。
- 2、Namesrv 本身是无状态，不产生数据的存储，是通过 Broker 心跳将 Topic 信息同步到 Namesrv 中。
- 3、多个 Namesrv 之间不会有数据的同步，是通过 Broker 向多个 Namesrv 多写。

📣 4. Broker

- 1、多个 Broker 可以形成一个 Broker 分组。每个 Broker 分组存在一个 Master 和多个 Slave 节点。
  - Master 节点，可提供读和写功能。Slave 节点，可提供读功能。
  - Master 节点会不断发送新的 CommitLog 给 Slave节点。Slave 节点不断上报本地的 CommitLog 已经同步到的位置给 Master 节点。
  - Slave 节点会从 Master 节点拉取消费进度、Topic 配置等等。
- 2、多个 Broker 分组，形成 Broker 集群。
  - Broker 集群和集群之间，不存在通信与数据同步。
- 3、Broker 可以配置同步刷盘或异步刷盘，根据消息的持久化的可靠性来配置。

📣 总结

目前官方提供三套配置：

- **2m-2s-async**

brokerClusterName	brokerName	brokerRole	brokerId
DefaultCluster	broker-a	ASYNC_MASTER	0
DefaultCluster	broker-a	SLAVE	1
DefaultCluster	broker-b	ASYNC_MASTER	0
DefaultCluster	broker-b	SLAVE	1

- **2m-2s-sync**

brokerClusterName	brokerName	brokerRole	brokerId
DefaultCluster	broker-a	SYNC_MASTER	0
DefaultCluster	broker-a	SLAVE	1
DefaultCluster	broker-b	SYNC_MASTER	0
DefaultCluster	broker-b	SLAVE	1

- **2m-noslave**

brokerClusterName	brokerName	brokerRole	brokerId
DefaultCluster	broker-a	ASYNC_MASTER	0
DefaultCluster	broker-b	ASYNC_MASTER	0

相关的源码解析，胖友可以看看 《RocketMQ 源码分析 —— 高可用》 。

如何保证消费者的消费消息的幂等性?

在 《精尽【消息队列 】面试题》 中，已经解析过该问题。当然，我们有几点要补充下：

- Producer 在发送消息时，默认会生成消息编号(`msgId`)，可见 `org.apache.rocketmq.common.message.MessageClientExt` 类。
- Broker 在存储消息时，会生成结合 offset 的消息编号(`offsetMsgId`)。
- Consumer 在消费消息失败后，将该消息发回 Broker 后，会产生新的 `offsetMsgId` 编号，但是 `msgId` 不变。

重点补充说明

RocketMQ 涉及的内容很多，能够问的问题也特别多，但是我们不能仅仅为了面试，应该是为了对 RocketMQ 了解更多，使用的更优雅。所以，强烈胖友认真撸下如下三个 PDF：

- 《RocketMQ 用户指南》 基于 RocketMQ 3 的版本。
- 《RocketMQ 原理简介》 基于 RocketMQ 3 的版本。
- 《RocketMQ 最佳实践》 基于 RocketMQ 3 的版本。

666. 彩蛋

RocketMQ 能够问的东西，真的挺多的，中间也和一些朋友探讨过。如果胖友有什么想要问的，可以在星球给留言。

参考与推荐如下文章：

- javahongxi 《RocketMQ 架构模块解析》
- 薛定谔的风口猪 《RocketMQ —— 角色与术语详解》
- 阿里中间件小哥 《一文讲透 Apache RocketMQ 技术精华》