RocketMQ 基础入门

1 RocketMQ 介绍

> RocketMQ作为一款纯java、分布式、队列模型的开源消息中间件，支持事务消息、顺序消息、批量消息、定时消息、消息回溯等。

前身是MetaQ，是阿里研发的一个队列模型的消息中间件，后开源给apache基金会成为了apache的顶级开源项目，具有高性能、高可靠、高实时、分布式特点。

RocketMQ 是阿里巴巴集团基于高可用分布式集群技术， 自主研发的云正式商用的专业消息中间件， 既可为分布式应用系统提供异步解耦和削峰填谷的能力，同时也具备互联网应用所需的海量消息堆积、高吞吐、可靠重试等特性， 是阿里巴巴双 11 使用的核心产品。

RocketMQ 原先阿里巴巴内部使用， 与 2017 年提交到 Apache 基金会成为 Apache 基金会的顶级开源项目， GitHub 代码库链接：https://github.com/apache/rocketmq.git

1.1 RocketMQ 特点

- 支持发布/订阅（Pub/Sub）和点对点（P2P）消息模型
- 在一个队列中可靠的先进先出（FIFO）和严格的顺序传递 （RocketMQ可以保证严格的消息顺序，而ActiveMQ无法保证）
- 支持拉（pull）和推（push）两种消息模式 （Push好理解，比如在消费者端设置Listener回调；而Pull，控制权在于应用，即应用需要主动的调用拉消息方法从Broker获取消息，这里面存在一个消费位置记录的问题（如果不记录，会导致消息重复消费））
- 单一队列百万消息的堆积能力 （RocketMQ提供亿级消息的堆积能力，这不是重点，重点是堆积了亿级的消息后，依然保持写入低延迟）
- 支持多种消息协议，如 JMS、MQTT 等
- 分布式高可用的部署架构,满足至少一次消息传递语义（RocketMQ原生就是支持分布式的，而ActiveMQ原生存在单点性）
- 提供 docker 镜像用于隔离测试和云集群部署
- 提供配置、指标和监控等功能丰富的 Dashboard

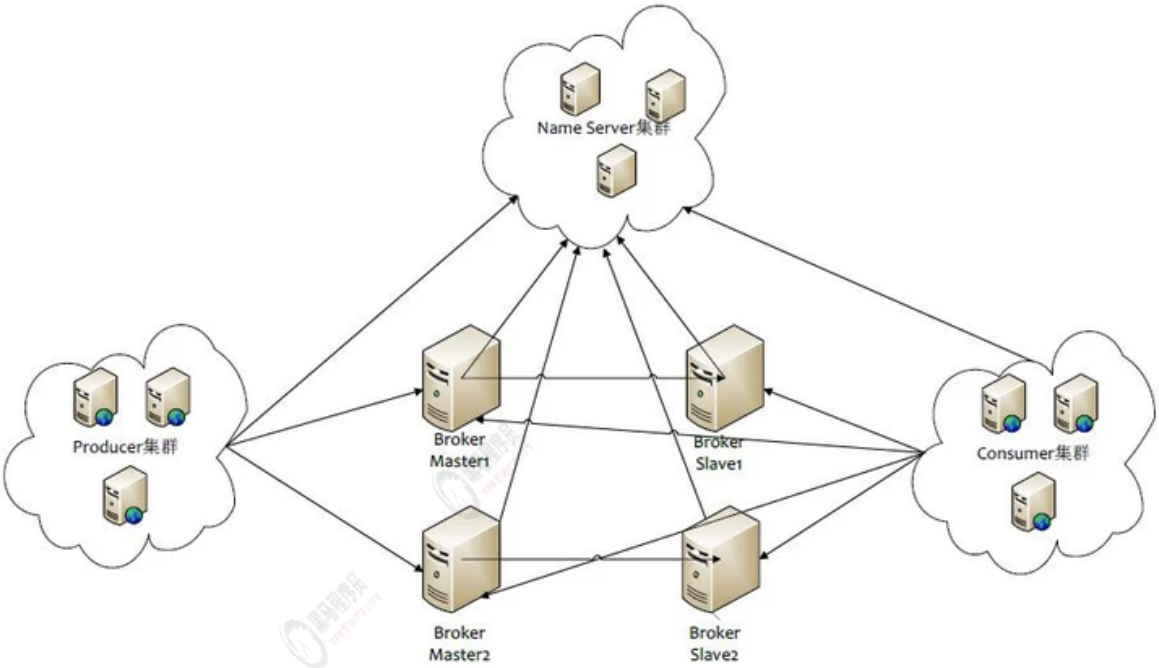
1.2 RocketMQ 优势

> 目前主流的 MQ 主要是 RocketMQ、kafka、RabbitMQ，其主要优势有：

- 支持事务型消息（消息发送和 DB 操作保持两方的最终一致性，RabbitMQ 和 Kafka 不支持）
- 支持结合 RocketMQ 的多个系统之间数据最终一致性（多方事务，二方事务是前提）
- 支持 18 个级别的延迟消息（RabbitMQ 和 Kafka 不支持）
- 支持指定次数和时间间隔的失败消息重发（Kafka 不支持，RabbitMQ 需要手动确认）
- 支持 Consumer 端 Tag 过滤，减少不必要的网络传输（RabbitMQ 和 Kafka 不支持）
- 支持重复消费（RabbitMQ 不支持，Kafka 支持）

2 RocketMQ 基本概念

> RocketMQ主要有四大核心组成部分：**NameServer**、**Broker**、**Producer**以及**Consumer**四部分。



2.1 NameServer

> Name Server是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。

NameServer 是整个 RocketMQ 的“大脑”，它是 RocketMQ 的服务注册中心，所以 RocketMQ 需要先启动 NameServer 再启动 Rocket 中的 Broker。

2.1.1 NameServer作用

名称服务器（NameServer）用来保存 Broker 相关元信息并给 Producer 和 Consumer 查找 Broker 信息。NameServer 被设计成几乎无状态的，可以横向扩展，节点之间相互之间无通信，通过部署多台机器来标记自己是一个伪集群。

每个 Broker 在启动的时候会到 NameServer 注册，Producer 在发送消息前会根据 Topic 到 NameServer 获取到 Broker 的路由信息，Consumer 也会定时获取 Topic 的路由信息。所以从功能上看应该是和 ZooKeeper 差不多，据说 RocketMQ 的早期版本确实是使用的 ZooKeeper ，后来改为了自己实现NameServer 。

2.1.2 和zk的区别

Name Server和ZooKeeper的作用大致是相同的，从宏观上来看，Name Server做的东西很少，就是保存一些运行数据，Name Server之间不互连，这就需要broker端连接所有的Name Server，运行数据的改动要发送到每一个Name Server来保证运行数据的一致性（这个一致性确实有点弱），这样就变成了Name Server很轻量级，但是broker端就要做更多的东西了。

而ZooKeeper呢，broker只需要连接其中的一台机器，运行数据分发、一致性都交给了ZooKeeper来完成。

2.1.3 高可用保障

Broker 在启动时向所有 NameServer 注册（主要是服务器地址等），生产者在发送消息之前先从 NameServer 获取 Broker 服务器地址列表（消费者一样），然后根据负载均衡算法从列表中选择一台服务器进行消息发送。

NameServer 与每台 Broker 服务保持长连接，并间隔 30S 检查 Broker 是否存活，如果检测到 Broker 宕机，则从路由注册表中将其移除，这样就可以实现 RocketMQ 的高可用。

2.2 Broker

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。它还存储与消息相关的元数据，包括用户组、消费进度偏移量、队列信息等。从部署结构图中可以看出 Broker 有 Master 和 Slave 两种类型，Master 既可以写又可以读，Slave 不可以写只可以读。

2.2.1 部署方式

Broker部署相对复杂，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，不同的Broker Id来定义，BrokerId为0表示Master，非0表示Slave。Master也可以部署多个。

从物理结构上看 Broker 的集群部署方式有四种：单 Master 、多 Master 、多 Master 多 Slave（同步刷盘）、多 Master多 Slave（异步刷盘）。

2.2.1.1 单 Master

这种方式一旦 Broker 重启或宕机会导致整个服务不可用，这种方式风险较大，所以显然不建议线上环境使用。

2.2.1.2 多 Master

所有消息服务器都是 Master ，没有 Slave 。这种方式优点是配置简单，单个 Master 宕机或重启维护对应用无影响。缺点是单台机器宕机期间，该机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受影响。

2.2.1.3 多 Master 多 Slave（异步复制）

每个 Master 配置一个 Slave，所以有多对 Master-Slave，消息采用异步复制方式，主备之间有毫秒级消息延迟。这种方式优点是消息丢失的非常少，且消息实时性不会受影响，Master 宕机后消费者可以继续从 Slave 消费，中间的过程对用户应用程序透明，不需要人工干预，性能同多 Master 方式几乎一样。缺点是 Master 宕机时在磁盘损坏情况下会丢失极少量消息。

2.2.1.4 多 Master 多 Slave（同步双写）

每个 Master 配置一个 Slave，所以有多对 Master-Slave ，消息采用同步双写方式，主备都写成功才返回成功。这种方式优点是数据与服务都没有单点问题，Master 宕机时消息无延迟，服务与数据的可用性非常高。缺点是性能相对异步复制方式略低，发送消息的延迟会略高。

2.2.2 高可用保障

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分钟没有收到心跳，则Name Server断开与Broker的连接。

2.3 生产者（Producer）

> 也称为消息发布者，负责生产并发送消息至 Topic

生产者向brokers发送由业务应用程序系统生成的消息。RocketMQ提供了发送：同步、异步和单向（one-way）的多种范例。

2.3.1 同步发送

同步发送指消息发送方发出数据后会在收到接收方发回响应之后才发下一个数据包。一般用于重要通知消息，例如重要通知邮件、营销短信。

2.3.2 异步发送

异步发送指发送方发出数据后，不等接收方发回响应，接着发送下个数据包，一般用于可能链路耗时较长而对响应时间敏感的业务场景，例如用户视频上传后通知启动转码服务。

2.3.3 单向发送

单向发送是指只负责发送消息而不等待服务器回应且没有回调函数触发，适用于某些耗时非常短但对可靠性要求并不高的场景，例如日志收集。

2.3.4 生产者组

生产者组（Producer Group）是一类 Producer 的集合，这类 Producer 通常发送一类消息并且发送逻辑一致，所以将这些 Producer 分组在一起。从部署结构上看生产者通过 Producer Group 的名字来标记自己是一个集群。

2.3.5 高可用保障

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。

2.4 消费者（Consumer）

> 也称为消息订阅者，负责从 Topic 接收并消费消息

消费者从brokers那里拉取信息并将其输入应用程序。在用户应用的角度，提供了两种类型的消费者：

- **Pull**：Pull型消费者主动地从brokers那里拉取信息。只要批量拉取到消息，用户应用程序就会启动消费过程
- **Push**：Push型消费者封装消息的拉取、消费进度和维护内部的其他工作，将一个在消息到达时执行的回调接口留给终端用户来实现。

2.4.1 消费者组

消费者组（Consumer Group）一类 Consumer 的集合名称，这类 Consumer 通常消费同一类消息并且消费逻辑一致，所以将这些 Consumer 分组在一起。消费者组与生产者组类似，都是将相同角色的分组在一起并命名，分组是个很精妙的概念设计，RocketMQ 正是通过这种分组机制，实现了天然的消息负载均衡。消费消息时通过 Consumer Group 实现了将消息分发到多个消费者服务器实例，比如某个 Topic 有9条消息，其中一个 Consumer Group 有3个实例（3个进程或3台机器），那么每个实例将均摊3条消息，这也意味着我们可以很方便的通过加机器来实现水平扩展。

2.4.2 高可用保障

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。

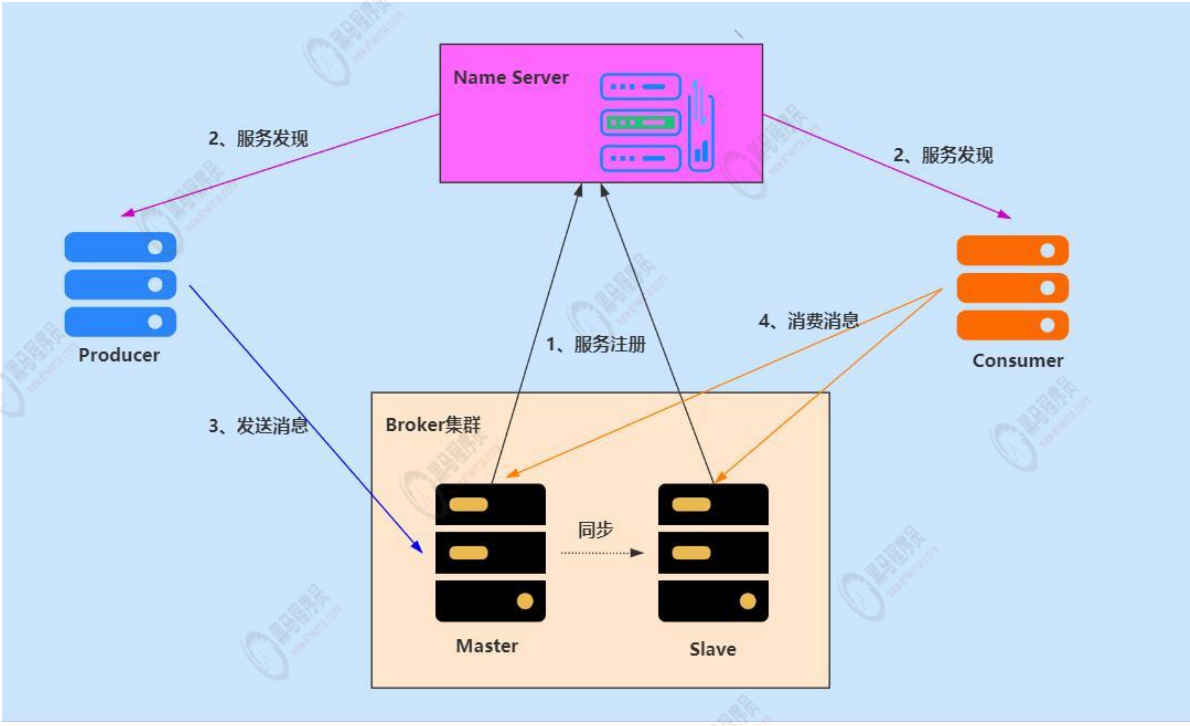
Consumer每隔30s（由ClientConfig中heartbeatBrokerInterval决定）向所有关联的broker发送心跳，Broker每隔10s扫描所有存活的连接，若某个连接2分钟内没有发送心跳数据，则关闭连接；并向该Consumer Group的所有Consumer发出通知，Group内的Consumer重新分配队列，然后继续消费。

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。

消费者对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对列数量*broker数量）。新挂载的消费者对列中拥有commitlog中的所有数据。

2.5 运转流程

> 上面介绍了RocketMQ的各个角色及其作用, 下面我们看一下各角色之间完整的交互过程。



1. NameServer 先启动
2. Broker 启动时向 NameServer 注册
3. 生产者在发送某个主题的消息之前先从 NamerServer 获取 Broker 服务器地址列表（有可能是集群），然后根据负载均衡算法从列表中选择一台Broker 进行消息发送。
4. NameServer 与每台 Broker 服务器保持长连接，并间隔 30S 检测 Broker 是否存活，如果检测到 Broker 宕机（使用心跳机制， 如果检测超过120S），则从路由注册表中将其移除。
5. 消费者在订阅某个主题的消息之前从 NamerServer 获取 Broker 服务器地址列表（有可能是集群），但是消费者选择从 Broker 中 订阅消息，订阅规则由 Broker 配置决定

2.6 名词解释

2.6.1 消息

消息（Message）就是要传输的信息。一条消息必须有一个主题（Topic），主题可以看做是你的信件要邮寄的地址。一条消息也可以拥有一个可选的标签（Tag）和额处的键值对，它们可以用于设置一个业务 key 并在 Broker 上查找此消息以便在开发期间查找问题。

2.6.2 主题

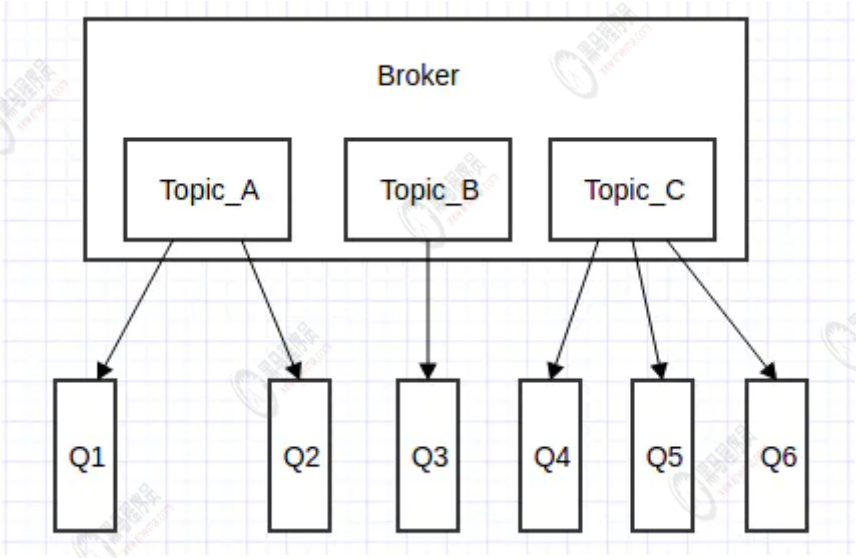
主题（Topic）可以看做消息的规类，它是消息的第一级类型。比如一个电商系统可以分为：交易消息、物流消息等，一条消息必须有一个 Topic 。Topic 与生产者和消费者的关系非常松散，一个 Topic 可以有0个、1个、多个生产者向其发送消息，一个生产者也可以同时向不同的 Topic 发送消息。一个 Topic 也可以被 0个、1个、多个消费者订阅。

2.6.3 标签

标签（Tag）可以看作子主题，它是消息的第二级类型，用于为用户提供额外的灵活性。使用标签，同一业务模块不同目的的消息就可以用相同 Topic 而不同的 Tag 来标识。比如交易消息又可以分为：交易创建消息、交易完成消息等，一条消息可以没有 Tag 。标签有助于保持您的代码干净和连贯，并且还可以为 RocketMQ 提供的查询系统提供帮助。

2.6.4 消息队列

消息队列（Message Queue），主题被划分为一个或多个子主题，即消息队列。一个 Topic 下可以设置多个消息队列，发送消息时执行该消息的 Topic ，RocketMQ 会轮询该 Topic 下的所有队列将消息发出去。下图 Broker 内部消息情况：



2.6.5 消息消费模式

> 消息消费模式有两种：集群消费（Clustering）和广播消费（Broadcasting）

默认情况下就是集群消费，该模式下一个消费者集群共同消费一个主题的多个队列，一个队列只会被一个消费者消费，如果某个消费者挂掉，分组内其它消费者会接替挂掉的消费者继续消费。而广播消费消息会发给消费者组中的每一个消费者进行消费。

2.6.6 消息顺序

> 消息顺序（Message Order）有两种：顺序消费（Orderly）和并行消费（Concurrently）

顺序消费表示消息消费的顺序同生产者为每个消息队列发送的顺序一致，所以如果正在处理全局顺序是强制性的场景，需要确保使用的主题只有一个消息队列。并行消费不再保证消息顺序，消费的最大并行数量受每个消费者客户端指定的线程池限制。

3 设计理念

> RocketMQ 的设计基于主题的发布与订阅模式，其核心功能包括消息发送、消息存储(Broker)、消息消费，整体设计追求简单与性能第一，主要体现在以下三个方面：

3.1 NameServer设计及其简单

RocketMQ摒弃了业界常用的zookeeper作为注册中心，而是使用自研的NameServer来实现元数据的管理，因为Topic路由信息无须在集群间保持强一致性，追求最终一致性，并且能容忍分钟级的不一致，所以RocketMQ的NameServer集群间互不通信，极大降低了设计的复杂度，降低了对网络的要求，提升性能。

3.2 高效的IO存储机制

RocketMQ追求消息发送的高吞吐量，RocketMQ消息存储文件设计成文件组的概念，组内单个文件大小固定，方便引入内存映射机制。

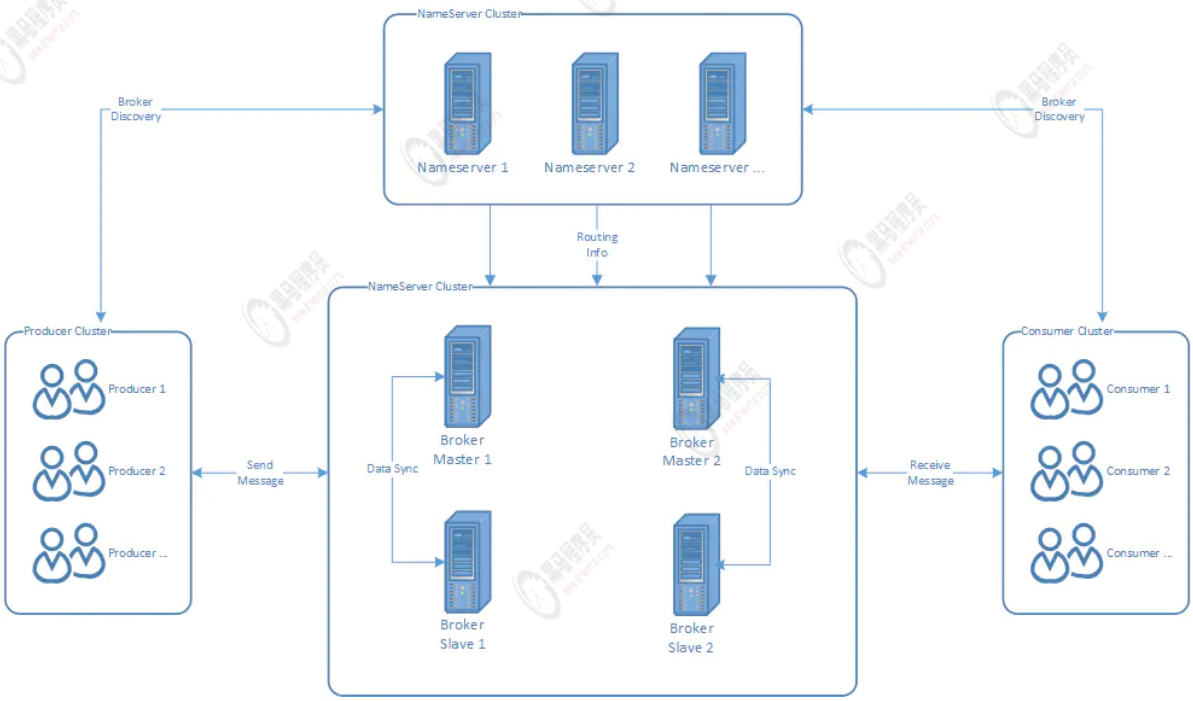
所有主题的消息存储基于顺序写，提升写性能，同时为了兼顾消息消费与消息查找，引入了消息消费队列文件与索引文件。

3.3 容忍存在的设计缺陷

> 适当将某些工作下放给RocketMQ使用者。消息中间件的实现者经常会遇到一个难题：

如何保证消息一定能被消息消费者消费，并且保证只消费一次。RocketMQ的设计者给出的解决办法是不解决这个难题，而是退而求其次，只保证消息被消费者消费，但设计上允许消息被重复消费，这样极大地简化了消息中间件的内核，使得实现消息发送高可用变得非常简单与高效，消息重复问题由消费者在消息消费时实现幂等。

4 RocketMQ 架构



> RocketMQ 架构图中展示了四个集群：

4.1 NameServer 集群

提供轻量级的服务发现及路由，每个 NameServer 记录完整的路由信息，提供相应的读写服务，支持快速存储扩展。有些其它开源中间件使用 ZooKeeper 实现服务发现及路由功能，如 Apache Kafka。

> NameServer是一个功能齐全的服务器，主要包含两个功能：

1. Broker 管理，接收来自 Broker 集群的注册请求，提供心跳机制检测 Broker 是否存活
2. 路由管理，每个 NameServer 持有全部有关 Broker 集群和客户端请求队列的路由信息

4.2 Broker 集群

通过提供轻量级的 Topic 和Queue 机制处理消息存储。同时支持推（Push）和拉（Pull）两种模型，包含容错机制。提供强大的峰值填充和以原始时间顺序累积数千亿条消息的能力。此外还提供灾难恢复，丰富的指标统计数据和警报机制，这些都是传统的消息系统缺乏的。

> Broker 有几个重要的子模块：

1. 远程处理模块，Broker 入口，处理来自客户端的请求
2. 客户端管理，管理客户端（包括消息生产者和消费者），维护消费者的主题订阅
3. 存储服务，提供在物理硬盘上存储和查询消息的简单 API
4. HA 服务，提供主从 Broker 间数据同步
5. 索引服务，通过指定键为消息建立索引并提供快速消息查询

4.3 Producer 集群

消息生产者支持分布式部署，分布式生产者通过多种负载均衡模式向 Broker 集群发送消息。

4.4 Consumer 集群

消息消费者也支持 Push 和 Pull 模型的分布式部署，还支持集群消费和消息广播。提供了实时的消息订阅机制，可以满足大多数消费者的需求。

> 有关架构图中集群间交互方式的说明：

1. Broker Master 和 Broker Slave 是主从结构，会执行数据同步 Data Sync
2. 每个 Broker 与 NameServer 集群中所有节点建立长连接，定时注册 Topic 信息到所有 NameServer
3. Producer 与 NameServer 集群中的其中一个节点（随机）建立长连接，定期从 NameServer 获取 Topic 路由信息，并与提供 Topic 服务的 Broker Master 建立长连接，定时向 Broker 发送心跳
4. Producer 只能将消息发送到 Broker Master，但是 Consumer 同时和提供 Topic 服务的 Master 和 Slave 建立长连接，既可以从 Master 订阅消息，也可以从 Slave 订阅消息。

5 设计目标

5.1 架构模式

RocketMQ与大部分消息中间件一样，采用发布订阅模式，基本的参与组件主要包括：消息发送者、消息服务器（消息存储）、消息消费、路由发现。

5.1.1 顺序消息

> 顺序消息（FIFO：First Input First Output）是一种严格按照顺序进行发布和消费的消息类型。要求消息的发布和消息消费都按照顺序进行，RocketMQ可以严格保证消息有序

RocketMQ可以严格的保证消息有序。但这个顺序，不是全局顺序，只是分区（queue）顺序。要全局顺序只能一个分区，但是同一条queue里面，RocketMQ的确是能保证FIFO的。

5.1.2 消息过滤

> 消息过滤是指在消息消费时，消息消费者可以对同一主题下的消息按照规则只消费自己感兴趣的消息。RocketMQ消息过滤支持在服务端与消费端的消息过滤机制。

1. 消息在Broker端过滤，Broker只将消息消费者感兴趣的消息发送给消息消费者。
2. 消息在消息消费端过滤，消息过滤方式完全由消息消费者自定义，但缺点是有很多无用的消息会从Broker传输到消费端。

5.1.3 消息存储

> 消息中间件的一个核心实现是消息的存储，对消息存储一般有如下两个维度的考量：

消息堆积能力和消息存储性能。RocketMQ追求消息存储的高性能，引入内存映射机制，所有主题的消息顺序存储在同一个文件中。同时为了避免消息无限在消息存储服务器中累积，引入了消息文件过期机制与文件存储空间报警机制。

5.2 消息高可用性

> 通常影响消息可靠性的有以下几种情况

1. Broker正常关机。
2. Broker异常宕机。
3. 操作系统宕机。
4. 机器断电，但是能立即恢复供电情况。
5. 机器无法开机（可能是CPU、主板、内存等关键设备损坏）。
6. 磁盘设备损坏。

针对上述情况，情况1,4的RocketMQ在同步刷盘机制下可以确保不丢失消息，在异步刷盘模式下，会丢失少量消息。情况5,6属于单点故障，一旦发生，该节点上的消息全部丢失，如果开启了异步复制机制，RoketMQ能保证只丢失少量消息，RocketMQ在后续版本中将引入双写机制，以满足消息可靠性要求极高的场合。

5.2.1 消息到消费低延迟

RocketMQ在消息不发生消息堆积时，以长轮询模式实现准实时的消息推送模式。

5.2.2 确保消息必须被消费一次

RocketMQ通过消息消费确认机制（ACK）来确保消息至少被消费一次，但由于ACK消息有可能丢失等其他原因，RocketMQ无法做到消息只被消费一次，有重复消费的可能。

5.2.3 回溯消息

回溯消息是指消息消费端已经消费成功的消息，由于业务要求需要重新消费消息。RocketMQ支持按时间回溯消息，时间维度可精确到毫秒，可以向前或向后回溯。

5.2.4 消息堆积

消息中间件的主要功能是异步解耦，必须具备应对前端的数据洪峰，提高后端系统的可用性，必然要求消息中间件具备一定的消息堆积能力。RocketMQ消息存储使用磁盘文件（内存映射机制），并且在物理布局上为多个大小相等的文件组成逻辑文件组，可以无限循环使用。RocketMQ消息存储文件并不是永久存储在消息服务器端，而是提供了过期机制，默认保留3天。

5.2.5 定时消息

定时消息是指消息发送到Broker后，不能被消息消费端立即消费，要到特定的时间点或者等待特定的时间后才能被消费。如果要支持任意精度的定时消息消费，必须在消息服务端对消息进行排序，势必带来很大的性能损耗，故RocketMQ不支持任意进度的定时消息，而只支持特定延迟级别。（一个说法是任意精度的定时消息会带来性能损耗，但是阿里云版本的 RocketMQ却提供这样的功能，充值收费优先策略？ ）

5.2.6 消息重试机制

消息重试是指消息在消费时，如果发送异常，消息中间件需要支持消息重新投递，RocketMQ支持消息重试机制。

6 RocketMQ安装

> 参考： https://baiyp.ren/RocketMQ%E5%AE%89%E8%A3%85.html

7 工程实例

7.1 Java 访问 RocketMQ 实例

> RocketMQ 目前支持 Java、C++、Go 三种语言访问，按惯例以 Java 语言为例看下如何用 RocketMQ 来收发消息的。

7.2 引入依赖

> 添加 RocketMQ 客户端访问支持，具体版本和安装的 RocketMQ 版本一致即可。

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.7.1</version>
</dependency>
```

7.3 消息生产者

```
public class Producer {

    public static void main(String[] args) throws Exception {
        //创建一个消息生产者，并设置一个消息生产者组
        DefaultMQProducer producer = new
DefaultMQProducer("niwei_producer_group");

        //指定 NameServer 地址
        producer.setNamesrvAddr("localhost:9876");

        //初始化 Producer，整个应用生命周期内只需要初始化一次
        producer.start();

        for (int i = 0; i < 100; i++) {
            //创建一条消息对象，指定其主题、标签和消息内容
              Message msg = new Message(
                    "topic_rocket_example" /* 消息主题名 */,
                    "TagA" /* 消息标签 */,
                    ("Hello Java demo RocketMQ " +
i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* 消息内容 */
            );

            //发送消息并返回结果
            SendResult sendResult = producer.send(msg);

            System.out.printf("%s%n", sendResult);
        }

        // 一旦生产者实例不再被使用则将其关闭，包括清理资源，关闭网络连接等
        producer.shutdown();
    }
}
```

示例中用 DefaultMQProducer 类来创建一个消息生产者，通常一个应用创建一个 DefaultMQProducer 对象，所以一般由应用来维护生产者对象，可以其设置为全局对象或者单例。该类构造函数入参 producerGroup 是消息生产者组的名字，无论生产者还是消费者都必须给出 GroupName ，并保证该名字的唯一性，ProducerGroup 发送普通的消息时作用不大，后面介绍分布式事务消息时会用到。

接下来指定 NameServer 地址和调用 start 方法初始化，在整个应用生命周期内只需要调用一次 start 方法。

初始化完成后，调用 send 方法发送消息，示例中只是简单的构造了100条同样的消息发送，其实一个 Producer 对象可以发送多个主题多个标签的消息，消息对象的标签可以为空。send 方法是同步调用，只要不抛异常就标识成功。

最后应用退出时调用 shutdown 方法清理资源、关闭网络连接，从服务器上注销自己，通常建议应用在 JBOSS、Tomcat 等容器的退出钩子里调用 shutdown 方法。

7.4 消息消费者

```
public class Consumer {

    public static void main(String[] args) throws Exception {
        //创建一个消息消费者，并设置一个消息消费者组
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("niwei_consumer_group");
        //指定 NameServer 地址
        consumer.setNamesrvAddr("localhost:9876");
        //设置 Consumer 第一次启动时从队列头部开始消费还是队列尾部开始消费

 consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
        //订阅指定 Topic 下的所有消息
        consumer.subscribe("topic_rocket_example", "*");

        //注册消息监听器
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
list, ConsumeConcurrentlyContext context) {
                //默认 list 里只有一条消息，可以通过设置参数来批量接收消息
                if (list != null) {
                    for (MessageExt ext : list) {
                        try {
                            System.out.println(new Date() + new
String(ext.getBody(), "UTF-8"));
                        } catch (UnsupportedEncodingException e) {
                            e.printStackTrace();
                        }
                    }
                }
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });

        // 消费者对象在使用之前必须要调用 start 初始化
        consumer.start();
        System.out.println("消息消费者已启动");
    }
}
```

示例中用 DefaultMQPushConsumer 类来创建一个消息消费者，通生产者一样一个应用一般创建一个 DefaultMQPushConsumer 对象，该对象一般由应用来维护，可以其设置为全局对象或者单例。该类构造函数入参 consumerGroup 是消息消费者组的名字，需要保证该名字的唯一性。

接下来指定 NameServer 地址和设置消费者应用程序第一次启动时从队列头部开始消费还是队列尾部开始消费。

接着调用 subscribe 方法给消费者对象订阅指定主题下的消息，该方法第一个参数是主题名，第二个擦书是标签名，示例表示订阅了主题名 topic_example_java 下所有标签的消息。

最主要的是注册消息监听器才能消费消息，示例中用的是 Consumer Push 的方式，即设置监听器回调的方式消费消息，默认监听回调方法中 List<MessageExt> 里只有一条消息，可以通过设置参数来批量接收消息。

最后调用 start 方法初始化，在整个应用生命周期内只需要调用一次 start 方法。

7.5 启动 Name Server

```
nohup sh bin/mqnamesrv &
tail -f ~/logs/rocketmqlogs/namesrv.log
```

RocketMQ 核心的四大组件中 Name Server 和 Broker 都是由 RocketMQ 安装包提供的，所以要启动这两个应用才能提供消息服务。首先启动 Name Server，先确保你的机器中已经安装了与 RocketMQ 相匹配的 JDK，并设置了环境变量 JAVA_HOME，然后在 RocketMQ 的安装目录下执行 bin 目录下的 mqnamesrv，默认会将该命令的执行情况输出到当前目录的 nohup.out 文件，最后跟踪日志文件查看 Name Server 的实际运行情况。

7.6 启动 Broker

```
nohup sh bin/mqbroker -n localhost:9876 &
tail -f ~/logs/rocketmqlogs/broker.log
```

同样也要确保你的机器中已经安装了与 RocketMQ 相匹配的 JDK，并设置了环境变量 JAVA_HOME，然后在 RocketMQ 的安装目录下执行 bin 目录下的 mqbroker，默认会将该命令的执行情况输出到当前目录的 nohup.out 文件，最后跟踪日志文件查看 Broker 的实际运行情况。