目录

[一、 使用场景](#_Toc3974_WPSOffice_Level1) [1](#_Toc3974_WPSOffice_Level1)

[二、 为什么是RocketMQ](#_Toc29212_WPSOffice_Level1) [1](#_Toc29212_WPSOffice_Level1)

[三、 整体介绍](#_Toc30007_WPSOffice_Level1) [2](#_Toc30007_WPSOffice_Level1)

[四、 消息队列要素](#_Toc25890_WPSOffice_Level1) [7](#_Toc25890_WPSOffice_Level1)

[五、 部署方式](#_Toc29280_WPSOffice_Level1) [11](#_Toc29280_WPSOffice_Level1)

[六、 消息类型及发送](#_Toc6547_WPSOffice_Level1) [12](#_Toc6547_WPSOffice_Level1)

[七、 场景硬伤](#_Toc17762_WPSOffice_Level1) [14](#_Toc17762_WPSOffice_Level1)

[八、 最佳实践](#_Toc29489_WPSOffice_Level1) [15](#_Toc29489_WPSOffice_Level1)

###### 使用场景

* **解耦**

解耦是消息队列的核心，一个流程要区分主要和次要节点，只关心核心的节点，次要并且不是那么重要的节点通知即可，不需要等待结果。例如交易流程，用户付款完成之后需要消息服务给用户发送邮件，该流程的主要节点是付款、更新订单、扣库存等，而发送邮件是次要节点，能否发送成功并不影响付款，所以解耦主要和次要节点，通知消息服务即可。

* **错峰**

应用服务能处理的请求是恒定的，出现峰值的时间段超出服务的能力，空闲状态资源都处于空置，不管是峰值或空闲都要保证服务的高质量，所以在流量高峰期要进行削峰，将耗资源、次要的节点延时异步化，以减少后端压力

* **广播**

消息生产者发送的每一条消息， Consumer 集群内所有的 Consumer 实例消费一次。例如商品变更后，索引服务和数据异构等服务都需要订阅该消息进行各自的处理，生产者只需将消息安全的发送出去，至于哪些服务需要订阅处理是不关心的。

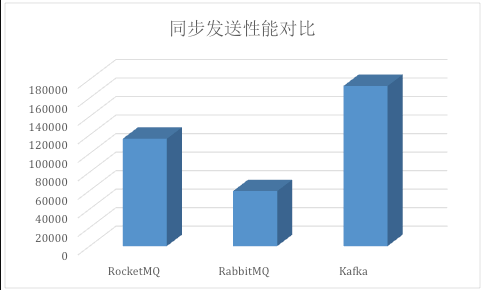
* **最终一致**

大多数互联网的场景只需保证数据的最终一致，将跨服务的大事务拆解成多个本地小事务+MQ，减少服务之间的交互，利用MQ的重试机制保证消息At-Least-Once消费，达到数据最终一致。例如用积分兑换付款后，账户服务需要扣除用户的积分，账户服务扣除积分与付款要么都成功要么都失败。

###### 为什么是RocketMQ

常见的三类消息中间件Kafka、RabbitMQ、RocketMQ

* Kafka是LinkedIn开源的分布式发布-订阅消息系统。Kafka主要特点是基于Pull的模式来处理消息消费，追求高吞吐量，一开始的目的就是用于日志收集和传输，不支持事务，对消息的重复、丢失、错误没有严格要求，适合产生大量数据的互联网服务的数据收集业务。
* RabbitMQ是使用Erlang语言开发的开源消息队列系统，基于AMQP协议来实现。AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。AMQP协议更多用在企业系统内，对数据一致性、稳定性和可靠性要求很高的场景，对性能和吞吐量的要求还在其次。
* RocketMQ是阿里开源的消息中间件，它是纯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，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。



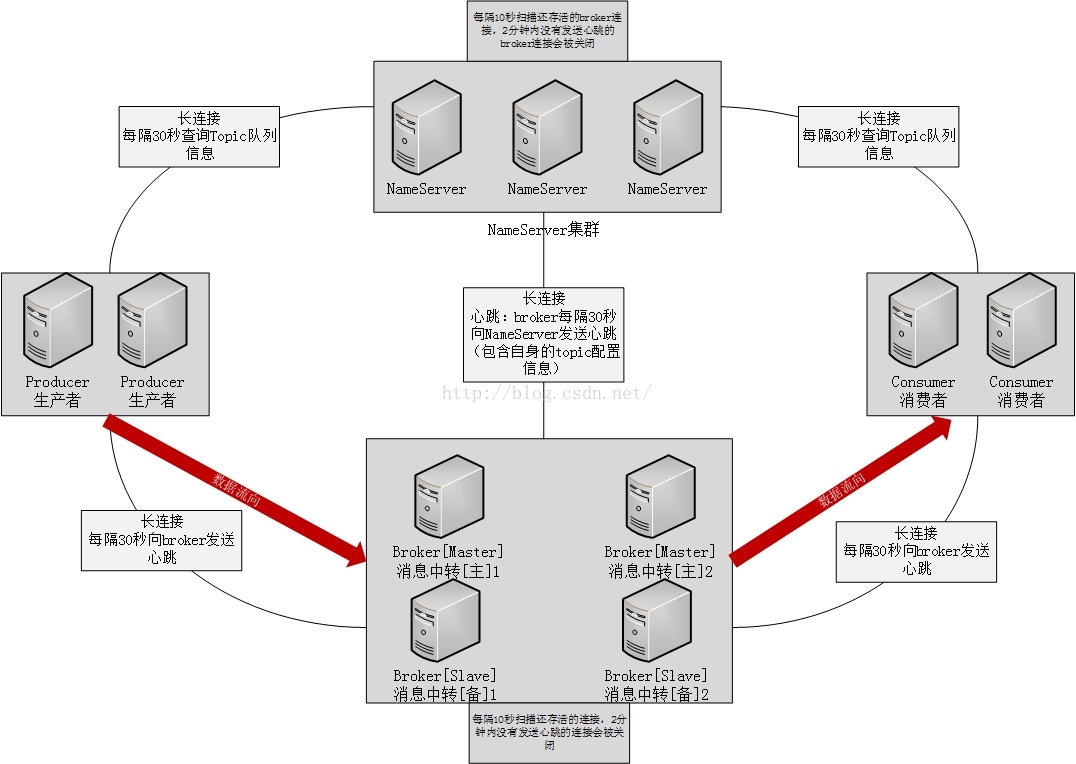
在同步发送场景中，三个消息中间件的表现区分明显：

* Kafka的吞吐量高达17.3w/s，这主要取决于它的队列模式保证了写磁盘的过程是线性IO，此时broker磁盘IO已达瓶颈。
* RocketMQ也表现不俗，吞吐量在11.6w/s，磁盘IO %util已接近100%。RocketMQ的消息写入内存后即返回ack，由单独的线程专门做刷盘的操作，所有的消息均是顺序写文件。
* RabbitMQ的吞吐量5.95w/s，CPU资源消耗较高。它支持AMQP协议，实现非常重量级，为了保证消息的可靠性在吞吐量上做了取舍，RabbitMQ在消息持久化场景下的性能测试，吞吐量在2.6w/s左右。

在服务端处理同步发送的性能上，Kafka>RocketMQ>RabbitMQ。

###### 整体介绍

整个RocketMQ包含了Name Server、Broker、Producer、Consumer，具体关系如图：



* **Name Server**

在MQ集群中是一个命名服务的角色，用于Borker、Producer和Consumer节点信息的注册和发现，同时还提供 Topic信息的查询。集群中每个Name Server都是一个无状态的节点，节点之间无任何信息同步。

* **Broker**

消息队列中的中转角色，负责消息的存储和转发，是RocketMQ的核心，主要功能如下：

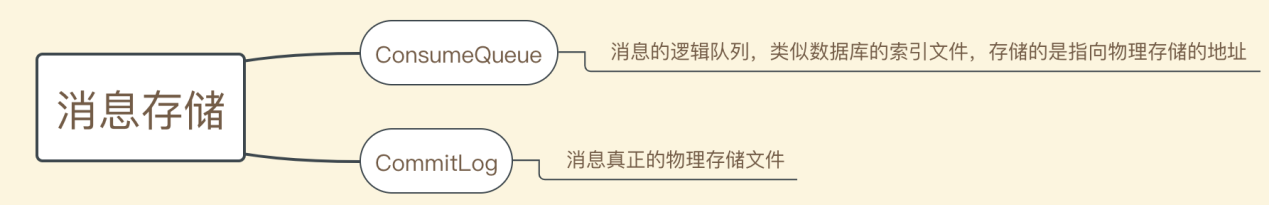
* 与Name Server保持通信
* 消息的持久化存储
* 接收Producer发送的消息
* 处理Consumer的消息消费

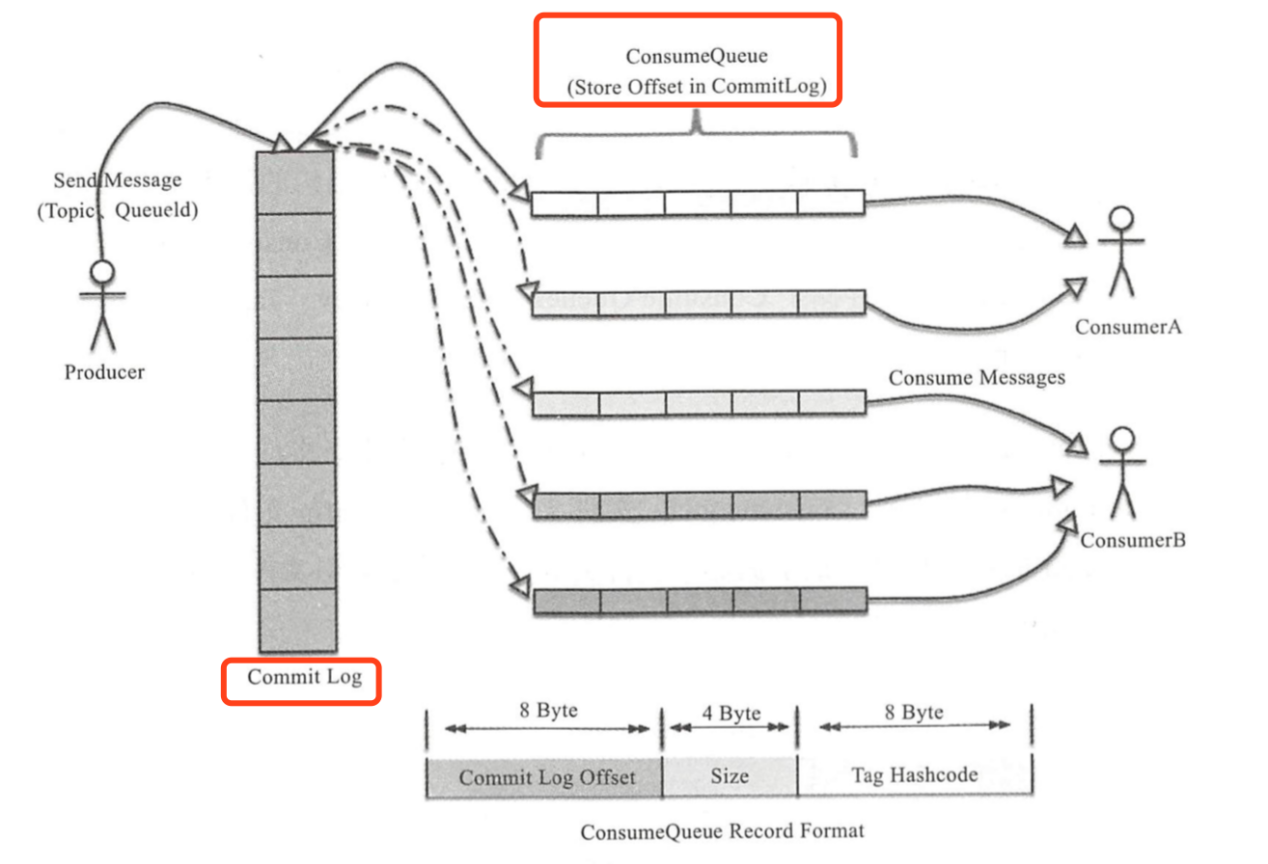
每个Broker与Name Server集群中的所有节点建立长连接，每隔30s向所有Name Server发送心跳，包含Topic信息

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，并更新topic与Broker的关系，但是不会通知Producer和Consumer

**消息的存储结构**

RocketMQ的存储是由ConsumeQueue和CommitLog配合完成的，如图：





CommitLog以物理文件的方式存放，每个Broker上的ConsumeQueue指向当前Broker的CommitLog

在CommitLog中一个消息的存储长度是不固定的，RocketMQ采用了一些机制，尽量向CommitLog中顺序写随机读，CommitLog顺序写可以大大提高写入的效率；虽然是随机读，但是利用package机制可以批量地从磁盘读取，然后缓存到内存中，可加快读取速度。CommitLog里存储了Consume Queues、Message Queue、Tag等所有信息，即使ConsumeQueue丢失，也可以通过commitLog完全恢复出来。

* **Producer**

消息的生产者，Producer作为客户端发送消息时候，根据Msg的Topic从本地缓存的TopicPublishInfoTable获取路由信息，如果没有则从NameServer上重新拉取并更新路由信息。

* 与Name Server关系

Producer与Name Server集群中的其中一个节点(随机选择)建立长连接，定时从Name Server查询Topic路由信息，如果该Name Server宕机，生产者会自动与下一个Name Server建立连接，并且能自动重连。

默认情况，生产者每隔30S从Name server获取所有topic队列的最新信息，这意味着如果Broker不可用，生产者最多30s才能感知，在此期间内发往Broker的所有消息都会失败。

* 与Broker关系

生产者和该生产者关联的所有Broker保持长连接，每隔30秒向所有Broker发送心跳，该时间由DefaultMQProducer的heartbeatBrokerInterval参数决定，可手动配置。Broker每隔10秒钟(此时间无法更改)，扫描所有存活Broker的长连接，若超过2分钟(此时间无法更改)没有发送心跳数据，则关闭连接，并移除broker上的生产者信息。

* **Consumer**

消息的消费者，支持push和pull两种模式，同事也支持集群消费和广播消费，Consumer既可以从Master/Slave订阅消息，订阅规则由Broker配置决定。

* 与Name Server关系

Consumer与Name Server集群中的其中一个节点(随机选择)建立长连接，定时从Name Server查询Topic路由信息，如果该Name Server宕机，生产者会自动与下一个Name Server建立连接，并且能自动重连。

默认情况，消费者每隔30秒从nameserver获取所有topic队列的最新信息，这意味着如果Broker不可用，生产者最多30s才能感知。

* 与Broker关系

消费者与该消费者关联的所有Broker建立长连接，消每隔30秒向所有broker发送心跳，该时间由DefaultMQPushConsumer的heartbeatBrokerInterval参数决定，可手动配置。broker每隔10秒钟（此时间无法更改），扫描所有存活Broker的长连接，若超过2分钟内（此时间无法更改）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Broker，然后继续消费。

* **消费机制**

消费者有一个拉取线程【不间断】的从Broker拉取消息，消息拉取到本地队列，然后本地消费线程消费本地队列，这是一个异步过程，拉取线程不会等待本地消费线程，这种模式实时性非常高。对消费者对本地队列有一个保护，因此本地消息队列不能无限大，否则可能会占用大量内存，本地消息队列大小由DefaultMQPushConsumer的pullThresholdForQueue属性控制，默认1000，可手动设置。

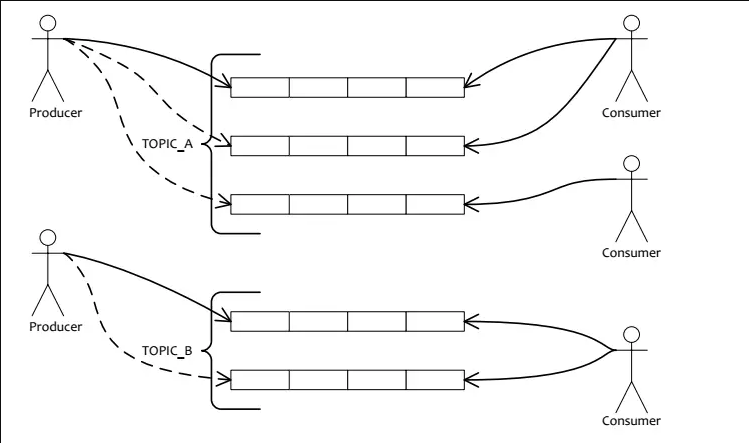
不间断的间隔时间由DefaultMQPushConsumer的pullInterval属性控制，默认为0。每次拉取默认1条消息，由DefaultMQPushConsumer的consumeMessageBatchMaxSize属性控制

* **消息主体**

一条消息由Topic、Tag、Key组成

通过 Topic 对不同的业务消息进行分类，比如商品消息、物流消息等，一般一个应用尽可能用一个Topic，消息子类型用tag来标识，只有发送消息设置了tag，消费方在订阅消息时，才可以利用tag 在broker做消息过滤。

* Topic表示消息主题，是最细粒度的订阅单位，一个topic可以分布在多个broker上，一个broker可以配置多个topic，消息量大的topic可以分布在多个broker上分摊broker的压力。
* Tag表示消息标签，是Topic下的二级分类，用于区分应用下的不同模块，例如商品消息可以下分为创建商品消息、变更商品消息等。
* Key是每个消息的唯一标识码，可以定位到唯一一条消息以及消息被谁消费，系统默认会生成key，也可自定义，但是一定要保证唯一。
* Message Queue是消息的物理管理单位，一个Topic下可以有多个Queue，Queue的引入使得消息的【存储】可以分布式集群化，具有了水平扩展能力。



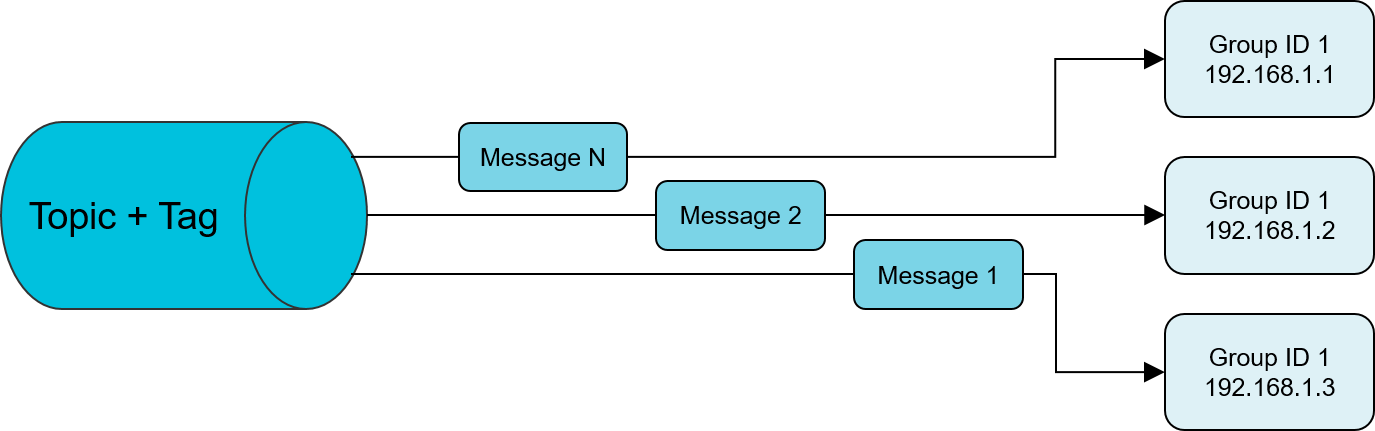
在 RocketMQ 中所有消息队列都是持久化，长度无限的数据结构，所谓长度无限是指队列中的每个存储单元都是定长，访问其中的存储单元使用 Offset 来访问，也可以认为 Message Queue 是一个长度无限的数组，Offset 就是下标。

* **消费模式**

RocketMQ有两种消费模式：BROADCASTING广播模式，CLUSTERING集群模式，默认的是集群消费模式。

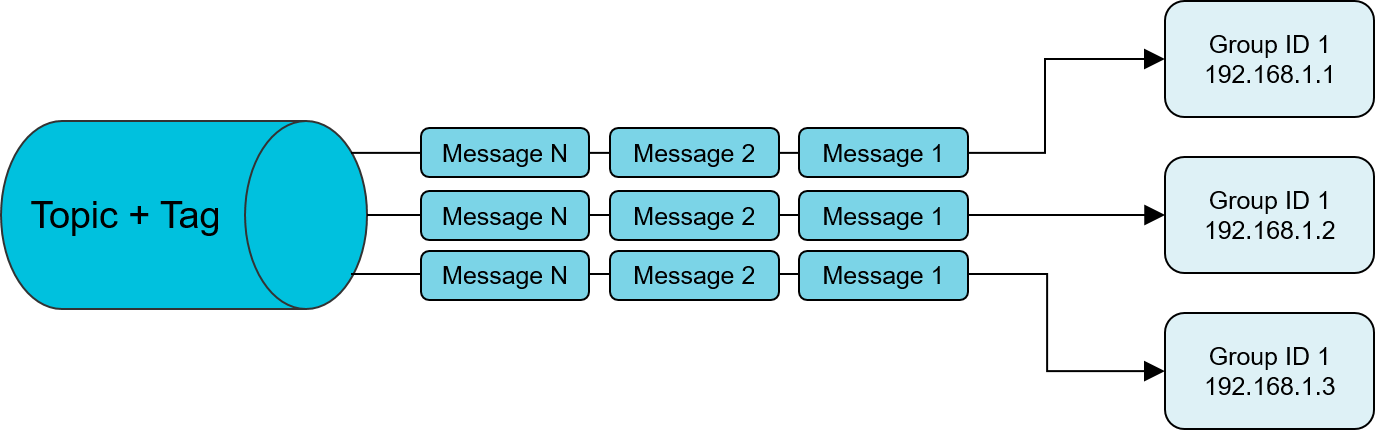
消息队列 RocketMQ 是基于发布/订阅模型的消息系统。消息的订阅方订阅关注的 Topic，以获取并消费消息。由于订阅方应用一般是分布式系统，以集群方式部署有多台机器。因此消息队列 RocketMQ 约定以下概念

* 集群消费



当使用集群消费模式时，消息队列 RocketMQ 认为任意一条消息只需要被集群内的任意一个Consumer消费即可，RocketMQ默认使用平均分配算法决定Queue由哪个Consumer消费。例如某个Topic有9条消息，ConsumerGroup有3个实例，那么每个实例只消费其中部分，消费完的消息不能被其他实例消费。

* 广播消费



当使用广播消费模式时，消息队列 RocketMQ 会将每条消息推送给集群内所有注册过的客户端，保证消息至少被每台机器消费一次，一条消息被多个consumer消费，即使这些consumer属于同一个ConsumerGroup,消息也会被ConsumerGroup中的每个Consumer都消费一次。

###### 消息队列要素

* 推拉模式

Consumer消费消息支持PUSH和PULL两种模式

* PUSH是Broker主动向Consumer推消息，但其本质还是pull，consumer把轮询过程封装了，并注册MessageListener监听器，取到消息后唤醒MessageListener的consumeMessage()来消费。由于PUSH的主动权是Broker，所以Broker不会考虑消费者的消费能力，只要有消息就会推送给Consumer，会导致消费端缓冲区溢出或异常。
* PULL是Consumer主动的从Broker拉取消息，可以实现批量的消费消息。Pull方式中取消息的过程需要自行维护，首先通过打算消费的Topic拿到MessageQueue的集合，遍历MessageQueue集合，然后针对每个MessageQueue批量取消息，一次取完后记录该队列下一次要取的开始offset,直到取完了，再换另一个MessageQueue。

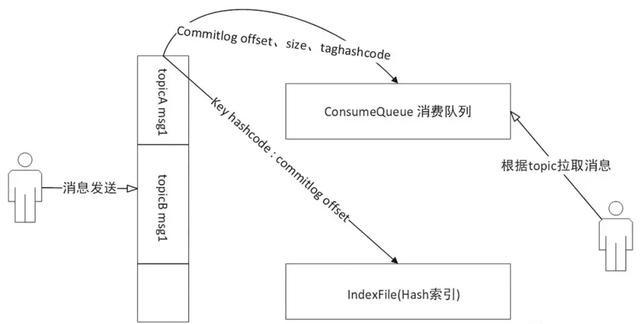
优点：Pull模式consumer可以根据自身的消费能力消费，比较合适消费者能力比较弱的场景。

缺点：由于主动权在消费者，消费者无法准确地决定何时去拉取最新的消息，所以解决的方案是长轮询，消费者如果尝试拉取失败不是直接return，而是把连接挂在那里wait,服务端如果有新的消息就会通知连接，但海量的长连接Broker对系统的开销还是不容小觑的，需要合理的设置时间间隔。

* 持久化

消息的存储是消息队列中最复杂和重要的部分，消息中间件应该尽量减少对于外部第三方中间件的依赖，依赖的外部系统越多会导致本身的设计越复杂，所以RocketMQ采用的是消息刷盘至所部署虚拟机/物理机的文件系统来做持久化，除非部署MQ机器本身或是本地磁盘挂了。但是频繁的刷盘会导致磁盘IO性能拖累系统性，那么RocketMQ是怎么解决呢？

存储设计图



RocketMQ的存储主要有三个文件

* CommitLog：消息存储文件，所有的消息存在这里，Broker单个实例下所有的Message Queue共用一个日志数据文件CommitLog来存储，每个文件大小默认1G
* ConsumeQueue：消费队列文件，消息在存储到CommitLog后，会将消息所在CommitLog偏移量、大小、tag的hashcode异步转发到消费队列文件存储，其类似于数据库的索引文件，存储的是指向物理存储的地址。每个Topic下的每个Message Queue都有一个对应的ConsumeQueue文件，目的在于提高Topic与ConsumeQueue检索消息的速度。
* Index：索引文件，消息在存储到CommitLog后，会将消息key与消息所在CommitLog偏移量转发到索引文件存储，可根据key查询到消息的信息

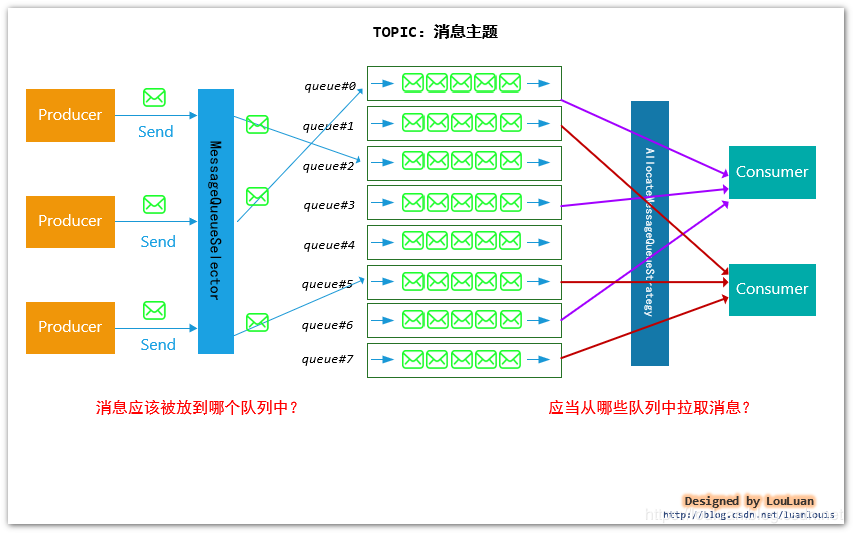
从图中可看出，Producer发送消息至Broker端，然后Broker端对消息刷盘持久化，保存至CommitLog中，只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。

每个Topic的消息由CommitLog的多个Message Queue分别存储，每个Message Queue对应单独的Consumer Queue文件。Consumer端先从ConsumeQueue读取持久化消息的起始物理位置偏移量offset、大小size和消息Tag的HashCode值，根据偏移量offset定位到消息所在的commitLog物理文件，再从CommitLog中读取待拉取消息的实体内容消费。

RocketMQ采用了全部消息都存入一个CommitLog文件中，并且对写操作加锁(putMessageLock)，保证串行顺序写入消息，避免磁盘竟争导致IO wait增高提高写入效率。

* 可靠投递

RocketMQ的消息投递分分为两种：一种是生产者往MQ Broker中投递；另外一种则是Broker往消费者投递。



一个Topic可能对应多个实际的消息队列(MessgeQueue)

在底层实现上，为了提高MQ的可用性和灵活性，一个Topic在实际存储的过程中，采用了多队列的方式，每个消息队列在使用中应当保证先入先出（FIFO,First In First Out）的方式进行消费。

生产者在发送相同Topic的消息时，消息体应当被存储到哪一个消息队列(MessageQueue)中是基于Producer投递消息的策略。

Producer默认基于Queue队列轮询算法，保证每一个Queue队列的消息投递数量尽可能均匀。

但是有些Queue队列可能由于投递的过程比较长，对于这样的Queue队列会影响后续投递的效果，基于该问题RocketMQ在每发送一个MQ消息后，都会统计一下消息投递的时间延迟，根据这个时间延迟可以知道往哪些Queue队列投递的速度快。在这种场景下，会优先使用消息投递延迟最小的策略，如果没有生效再使用Queue队列轮询的方式。

以上两种投递方式【轮询策略】【延迟最小策略】属于对消息投递的顺序没有要求的场景，这种投递的速度和效率比较高，而在有些场景下，是需要保证同类型消息投递的顺序性，可以发送【[顺序消息](#_顺序消息)】

如果消息投递失败，默认情况重试2次，可通过retryTimesWhenSendFailed定义重试次数，通过设置sendMsgTimeout来定义超时时间，消息最终能否可靠投递到Broker并落地，是依靠Broker的刷盘机制，确保刷盘成功才返回客户端结果。

* 消费确认

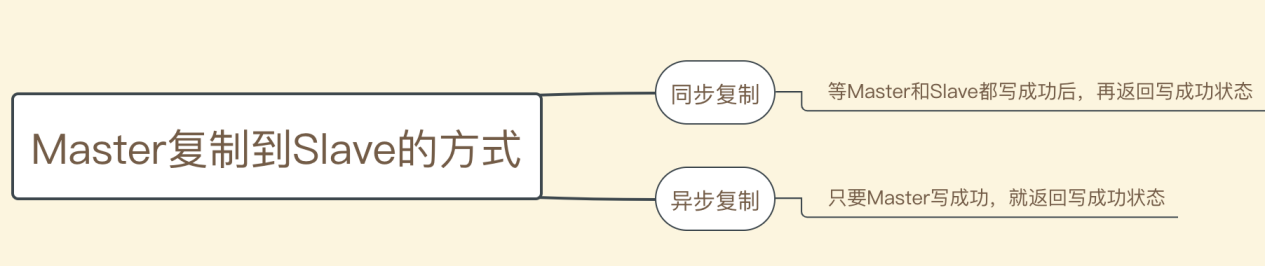
当Broker把消息投递给Producer后，默认自动 ack，但是消费者并不能保证消费一定成功，但是Producer也可以在消费成功后手动ack保证【At least Once】，任何的异常情况(包含业务异常、断电等)都会默认ConsumeConcurrentlyStatus.RECONSUME\_LATER可以重复投递消费，为了保证消息至少消费成功一次，RocketMQ会把消息重发回Broker的Retry Topic，在延迟的某个时间点后，再次投递到这个ConsumerGroup消费，而这条失败的消息即使立刻重试消费，99%也不成功，所以最好提供一种定时重试机制，即过10s秒后再重试，而如果一直这样重复消费都持续失败到一定次数(16次)，就会投递到DLQ死信队列，可以监控死信队列来做人工干预。但是一般的业务场景消费失败就算重试16次也是会失败，既浪费时间也浪费资源，可以设置重试多少次message.getReconsumeTimes()后不再重试

但是消息的重试消费会导致业务数据的错乱，例如Producer已经处理完业务事务，在发起ack的那一刻宕机，Producer会收到Broker的重复消息投递。这就需要Producer保证业务的幂等性，可利用状态机、唯一请求流水号等机制保证幂等性。

* 复制和刷盘

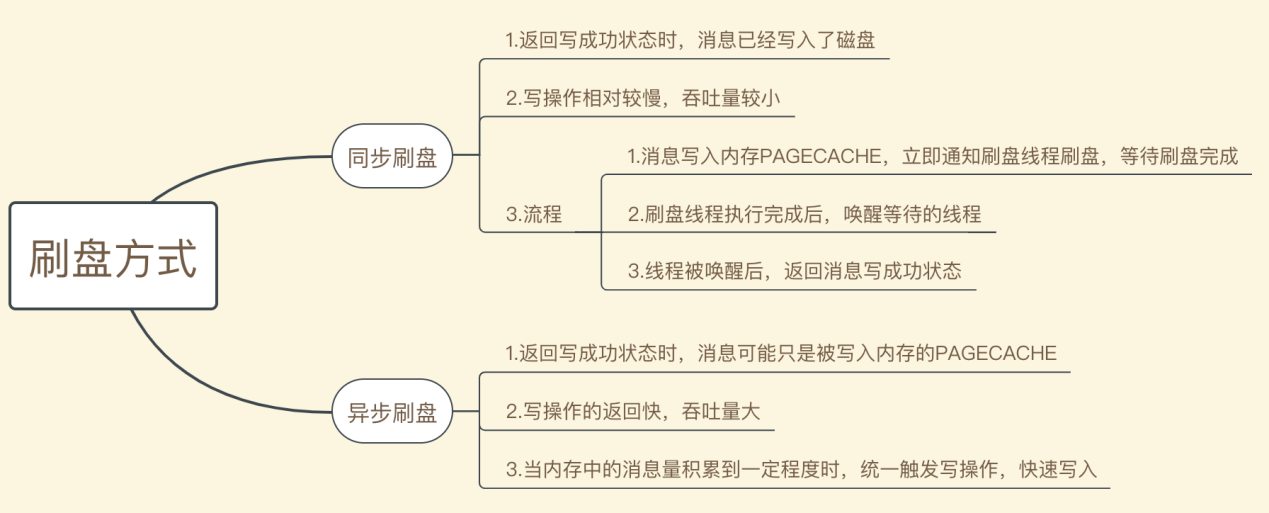
复制

复制是消息从Master到Slave的同步过程，复制方式属性brokerRole分为同步复制(SYNC\_MASTER)和异步复制(ASYNC\_MASTER)，默认为异步同步，具体区别如图：



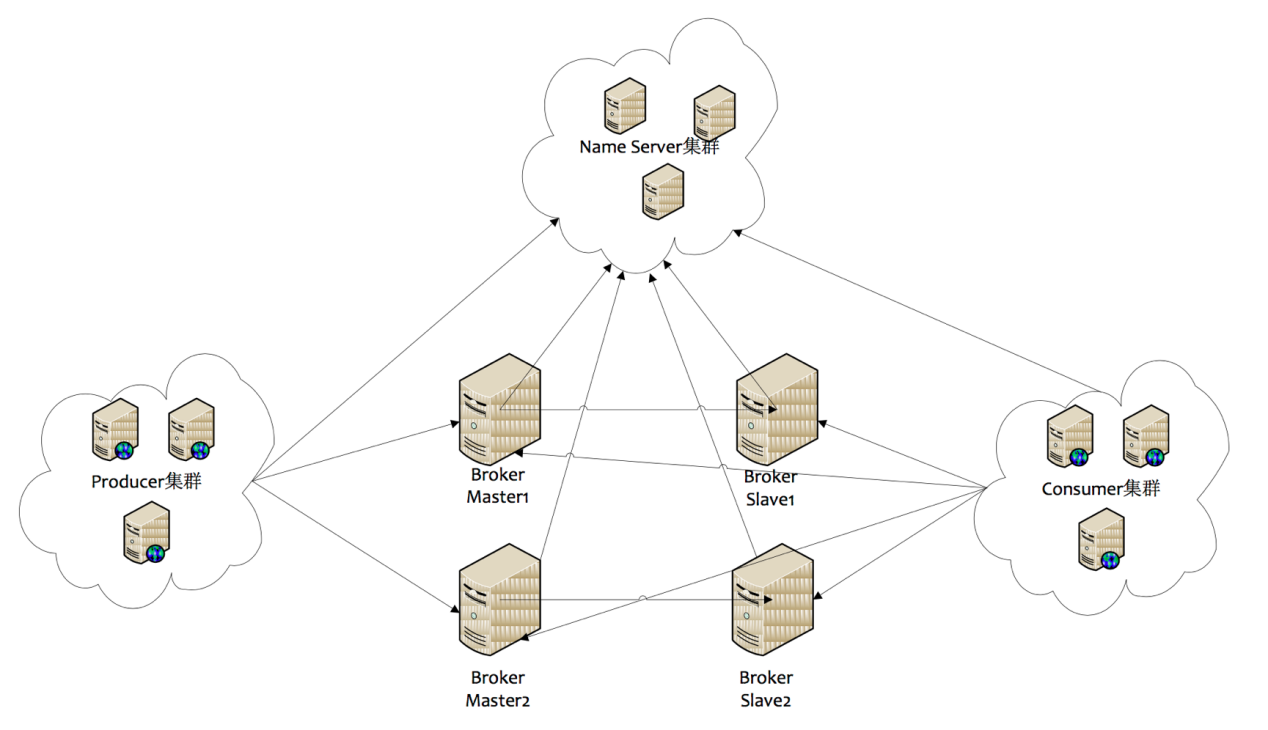
刷盘

刷盘是消息持久化的操作，刷盘方式属性flushDiskType分为同步(SYNC\_FLUSH)和异步(ASYNC\_FLUSH)，默认为异步刷盘，具体区别如图：



在实际场景中要合理的设置复制和刷盘策略，需要权衡吞吐量和数据安全，通常会设置为同步复制、异步刷盘。

###### 部署方式



* 单Master

不可靠，一旦这个 master 节点宕机，那么整个服务就不可用

* 多Master

多个 master 节点组成集群，单个 master 节点宕机或者重启对应用没有影响，但是单个 master 节点宕机期间，未被消费的消息在节点恢复之前消息不能被消费，同时Topic 相对应的 Message Queue应该分布在集群中各个节点，否则该节点宕机会对订阅该 topic 的Producer造成影响。

* 多Master多Slave模式

异步复制

在每个 master 节点都有至少一个对应的 slave，master节点可读可写，但是 slave 只能读不能写。集群采用异步复制方式，主从有毫秒级延迟。在 master 宕机时，消费者可以从 slave 读取消息，但是异步复制的同步方式有可能会有消息丢失的问题。

同步双写

同多 master 多 slave 异步复制模式类似，区别在于 master 和 slave 之间的数据同步是同步双写，这样能保证数据不丢失，但是消息 RT 会略长，性能相比异步复制低10%左右。

RocketMQ建议两主两从异步或同步复制数据

###### 消息类型及发送

* 消息类型
* 普通消息

普通消息是指消息队列 RocketMQ 中无特性的消息，区别于有特性的定时/延时消息、顺序消息和事务消息。

* 顺序消息

消息（FIFO 消息）是消息队列 RocketMQ 提供的一种严格按照顺序进行发布和消费的消息类型，顺序消息指消息发布和消息消费都按顺序进行。

顺序发布：对于指定的一个 Topic，客户端将按照一定的先后顺序发送消息。

顺序消费：对于指定的一个 Topic，按照一定的先后顺序接收消息，即先发送的消息一定会先被客户端接收到。

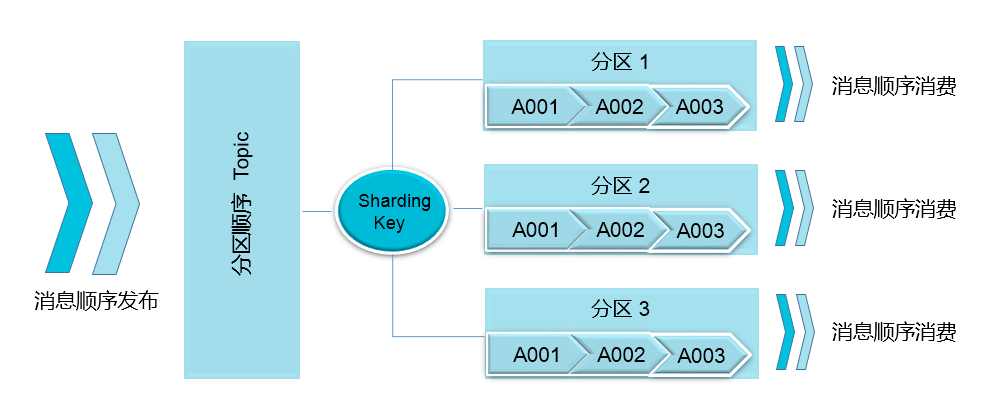
顺序消息分为全局顺序和分区顺序

* 全局顺序



全局顺序消息对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费。适用于性能要求不高，所有的消息严格按照 FIFO 原则进行消息发布和消费的场景。

* 分区消息



分区顺序消息对于指定的一个 Topic，所有消息根据 sharding key 进行区块分区，同一个分区内的消息按照严格的 FIFO 顺序进行发布和消费。适用于性能要求高，以 sharding key 作为分区字段，在同一个区块中严格的按照 FIFO 原则进行消息发布和消费的场景。

但是顺序消息有以下几个缺点：

1. 顺序消息不支持异步发送方式，否则将无法严格保证顺序
2. 顺序消息暂不支持广播模式
3. 顺序消息无法利用集群Fail Over特性消费顺序消息的并行度依赖于队列数量队列热点问题，个别队列由于哈希不均导致消息过多，消费速度跟不上，产生消息堆积问题遇到消息失败的消息，无法跳过，当前队列消费暂停。

全局顺序与分区顺序对比

【消息类型对比】

| **Topic 的消息类型** | **支持事务消息** | **支持定时/延时消息** | **性能** |
| --- | --- | --- | --- |
| 无序消息（普通、事务、定时/延时消息） | 是 | 是 | 最高 |
| 分区顺序消息 | 否 | 否 | 高 |
| 全局顺序消息 | 否 | 否 | 一般 |

【发送方式对比】

| **消息类型** | **支持可靠同步发送** | **支持可靠异步发送** | **Oneway 发送** |
| --- | --- | --- | --- |
| 无序消息（普通、事务、定时/延时消息） | 是 | 是 | 是 |
| 分区顺序消息 | 是 | 否 | 否 |
| 全局顺序消息 | 是 | 否 | 否 |

* 延时消息

Producer 将消息发送到消息队列 RocketMQ 服务端，但并不期望这条消息立马投递，而是延迟一定时间后才投递到 Consumer 进行消费，该消息即延时消息。延时消息时需要设定一个延时时间长度，消息将从当前发送时间点开始延迟固定时间之后才开始投递。

* 定时消息

Producer 将消息发送到消息队列 RocketMQ 服务端，但并不期望这条消息立马投递，而是推迟到在当前时间点之后的某一个时间投递到 Consumer 进行消费，该消息即定时消息。定时消息需要明确指定消息发送时间点之后的某一时间点作为消息投递的时间点。

* 普通消息发送方式
* 可靠同步

同步发送是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的通讯方式。

* 可靠异步

异步发送是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的通讯方式。 消息队列 RocketMQ 的异步发送，需要用户实现异步发送回调接口（SendCallback）。发送方通过回调接口接收服务器响应，并对响应结果进行处理。异步发送一般用于链路耗时较长，对 RT 响应时间较为敏感的业务场景，例如签约服务完成签约后通知业务应用签约结果，业务应用根据结果做后续的处理。

* 单向

单向（Oneway）发送特点为发送方只负责发送消息，不等待服务器回应且没有回调函数触发，即只发送请求不等待应答。 此方式发送消息的过程耗时非常短，一般在微秒级别。

###### 场景硬伤

RocketMQ作为一款高性能、高吞吐量的消息中间件，消息的重复问题和消息事务是不可避免的三大问题。

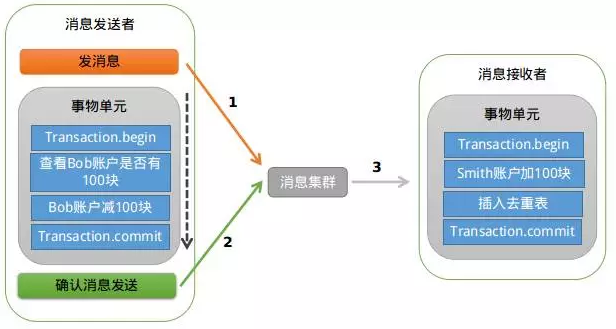
* 重复消费

造成消息重复的根本原因是：网络不可达。只要通过网络交换数据，就无法避免这个问题。所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理？

* 消费端处理消息的业务逻辑保持幂等性
* 保证每条消息都有唯一编号且保证消息处理成功与去重表的日志同时出现

RocketMQ不保证消息不重复，如果你的业务需要保证严格的不重复消息，需要在业务端去重。

* 事务消息



其实RocketMQ总共分为3个阶段：发送Prepared消息、执行本地事务、发送确认消息。

RocketMQ第一阶段发送Prepared消息时拿到消息的地址，第二阶段执行本地事物，第三阶段通过消息的地址去访问消息并修改消息的状态。

如果确认消息发送失败了怎么办？RocketMQ会定期(默认1分钟)扫描消息集群中的事物消息，如果发现了Prepared消息，它会向消息Producer发起CheckTransaction确认本地事务是否执行成功，Producer再根据事务设置的策略来决定是回滚事务还是继续发送确认消息,这样就保证了消息发送与本地事务同时成功或同时失败。

在一个跨系统的大事务处理中，一般利用【大事务 = 多个小事务 + 异步】的方式，通过异步重试机制保证多个小事务的一致性。

###### 最佳实践

* Producer
* 一个应用尽可能用一个 Topic，消息子类型用 tags 来标识
* 每个消息在业务层面的唯一标识码，要设置到 keys 字段，方便将来定位消息丢失问题
* 对于消息不可丢失应用，务必要有人工重发机制
* 某些应用如果不关注消息是否发送成功，请直接使用sendOneWay方法发送消息
* Consumer
* 消费过程要做到幂等
* 尽量使用批量方式消费方式，可以很大程度上提高消费吞吐量
* 线上应该关闭autoCreateTopicEnable，即在配置文件中将其设置为false