**RabbitMQ**

▶**主流消息中间件及选型**



◇RabbitMQ

优点

• 轻量级，快速，部署使用方便

• 支持灵活的路由配置。RabbitMQ中，在生产者和队列之间有一个交换器模块。根据配置 的路 由规则，生产者发送的消息可以发送到不同的队列中。路由规则很灵活，还可以自 己实现。

• RabbitMQ的客户端支持大多数的编程语言

缺点

• 如果有大量消息堆积在队列中，性能会急剧下降

• RabbitMQ的性能在Kafka和RocketMQ中是最差的，每秒处理几万到几十万的消息。如 果应用要求高的性能，不要选择RabbitMQ。

• RabbitMQ是Erlang开发的，功能扩展和二次开发代价很高。

◇RocketMQ

RocketMQ主要用于有序，事务，流计算，消息推送，日志流处理，binlog分发等场景。

优点

RocketMQ几乎具备了消息队列应该具备的所有特性和功能。 java开发，阅读源代码、 扩展、二次开发很方便。 对电商领域的响应延迟做了很多优化。在大多数情况下，响应 在毫秒级。如果应用很关注响应时间，可以使用RocketMQ。 性能比RabbitMQ高一个 数量级，每秒处理几十万的消息。

缺点

跟周边系统的整合和兼容不是很好

◇Kafka

优点

• Kafka的可靠性，稳定性和功能特性基本满足大多数的应用场景。

• 跟周边系统的兼容性是数一数二的，尤其是大数据和流计算领域，几乎所有相关的开源 软件都支持 Kafka。 Kafka高效，可伸缩，消息持久化。

• 支持分区、副本和容错。

• 它的异步消息的发送和接收是三个中最好的，但是跟RocketMQ拉不开数量级，每秒处 理几十万的消息。 如果是异步消息，并且开启了压缩，Kafka最终可以达到每秒处理 2000w消息的级别。

缺点

由于是异步的和批处理的，延迟也会高，不适合电商场景。

▶**消息中间件应用场景**

◇流量削峰

削去秒杀场景下的峰值写流量

◇异步处理

通过异步处理简化秒杀请求中的业务流程

◇解耦

解耦，实现秒杀系统模块之间松耦合

▶**Exchange类型**

◇Fanout

会把所有发送到该交换器的消息路由到所有与该交换器绑定的队列中

◇Direct

direct类型的交换器路由规则很简单，它会把消息路由到那些BindingKey和RoutingKey完全匹配 的队列中

◇Topic

topic类型的交换器在direct匹配规则上进行了扩展，也是将消息路由到BindingKey和RoutingKey 相匹配的队列中，这里的匹配规则稍微不同，它约定： BindingKey和RoutingKey一样都是由"." 分隔的字符串；BindingKey中可以存在两种特殊字符“”和 “#”，用于模糊匹配，其中""用于 匹配一个单词，"#"用于匹配多个单词

◇Headers

headers类型的交换器不依赖于路由键的匹配规则来路由信息，而是根据发送的消息内容中的 headers属性进行匹配

▶**名词解释**

◇Broker

RabbbitMQ消息队列代理服务器实体。

◇Producer

发送消息的应用程序。

◇Consumer

接收消息的用户程序。

◇Exchange

交换器，生产者直接将消息发送给交换器。交换器将消息分发给指定的队列。它指定消息按什 么规则，路由到哪个队列。

◇Binding

绑定，指的是交换器和队列之间的关系。它的作用就是把exchange和queue按照路由规则绑定 起来。

◇Routing Key

路由关键字，exchange根据这个关键字进行消息投递。

◇vhost

虚拟主机，一个broker里可以开设多个vhost，用作不同用户的权限分离。

◇Channel

消息通道，包含了大量的API可用于编程。在客户端的每个连接里，可建立多个channel，每个 channel代表一个会话任务。

◇Connection

在客户创建一个到某个虚拟主机的连接。

◇ConnectionFactory

连接工厂类。可以创建一个连接。

▶**工作流程**

◇生产者发送消息的流程

• 生产者连接RabbitMQ，建立TCP连接( Connection)，开启信道（Channel）

• 生产者声明一个Exchange（交换器）并设置相关属性，比如交换器类型、是否持久化等

• 生产者声明一个队列井设置相关属性，比如是否排他、是否持久化、是否自动删除等

• 生产者通过 bindingKey （绑定Key）将交换器和队列绑定（ binding ）起来

• 生产者发送消息至RabbitMQ Broker，其中包含 routingKey （路由键）、交换器等信息

• 相应的交换器根据接收到的 routingKey 查找相匹配的队列。

• 如果找到，则将从生产者发送过来的消息存入相应的队列中。

• 如果没有找到，则根据生产者配置的属性选择丢弃还是回退给生产者

• 关闭信道、关闭连接。

◇消费者接收消息的过程

• 消费者连接到RabbitMQ Broker ，建立一个连接(Connection ) ，开启一个信道(Channel) 。

• 消费者向RabbitMQ Broker 请求消费相应队列中的消息，可能会设置相应的回调函数， 以及 做一些准备工作

• 等待RabbitMQ Broker 回应并投递相应队列中的消息， 消费者接收消息。

• 消费者确认( ack) 接收到的消息。

• RabbitMQ 从队列中删除相应己经被确认的消息。

• 关闭信道、关闭连接。

▶**消息可靠性保障**

◇发送端确认机制

RabbitMQ后来引入了一种轻量量级的方式，叫发送方确认(publisher confirm)机制。生产者将信 道设置成confirm(确认)模式，一旦信道进入confirm 模式，所有在该信道上发布的消息，都会被 指派 一个唯一的ID(从1 开始)，一旦消息被投递到所有匹配的队列之后，RabbitMQ 就会发送 一个确认(Basic.Ack)给生产者(包含消息的唯一 ID)，这样生产者就知道消息已经正确送达了。

confirm模式

普通confirm模式 代码：略

批量confirm模式 代码：略

异步confirm模式 代码：略

◇消息持久化机制

• Exchange的持久化。通过定义时设置durable参数为ture来保证Exchange相关的元数据不丢 失。

• Queue的持久化。也是通过定义时设置durable 参数为ture来保证Queue相关的元数据不丢 失。

• 消息的持久化。通过将消息的投递模式 (BasicProperties 中的 deliveryMode 属性)设置为 2 即可实现消息的持久化，保证消息自身不丢失

◇Broker端的高可用集群

镜像模式

集群模式

◇消费者确认机制

RabbitMQ在消费端会有Ack机制，即消费端消费消息后需要发送Ack确认报文给Broker端，告 知自己是否已消费完成，否则可能会一直重发消息直到消息过期（AUTO模式）。

• 采用AUTO（自动Ack）模式，不主动捕获异常，当消费过程中出现异常时会将消息放回 Queue 中，然后消息会被重新分配到其他消费者节点（如果没有则还是选择当前节点）重新 被消费， 默认会一直重发消息并直到消费完成返回Ack或者一直到过期

• 采用MANUAL（手动Ack）模式，消费者自行控制流程并手动调用channel相关的方法返回 Ack

◇消息幂等性

在消费端让我们消费消息的操作具备幂等性

▶**TLL机制**



在京东下单，订单创建成功，等待支付，一般会给30分钟的时间，开始倒计时。如果在这段时间内 用户没有支付，则默 认订单取消。

◇使用场景

订单失效

◇RabbitMQ 可以对消息和队列两个维度来设置TTL。

1. 通过Queue属性设置，队列中所有消息都有相同的过期时间。

2. 对消息自身进行单独设置，每条消息的TTL 可以不同

◇规则

如果不设置TTL，则表示此消息不会过期；

如果TTL设置为0，则表示除非此时可以直接将消息投递到消费者，否则该消息会被立即丢弃

▶**死信队列**

示例：用户下单，调用订单服务，然后订单服务调用派单系统通知外卖人员送单，这时候订单系统与 派单 系统 采用 MQ异步通讯。在定义业务队列时可以考虑指定一个 死信交换机，并绑定一个死信 队列。当消息变成死信时，该消 息就会被发送到该死信队列上，这样方便我们查看消息失败的原因。

◇导致消息变为死信条件

• 消息被拒绝（Basic.Reject/Basic.Nack），并且设置requeue参数为false；

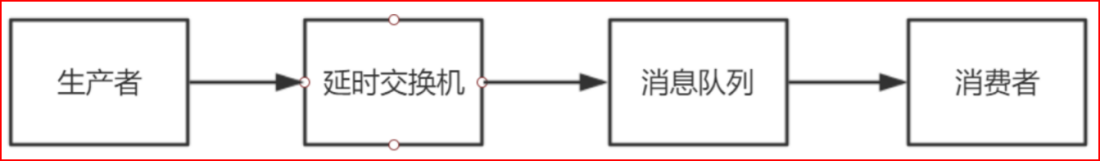
• 消息过期；

• 队列达到最大长度

▶**延迟队列**

使用场景：

在支付宝上面买电影票，锁定了一个座位后系统默认会帮你保留15分钟时 间，如果15分钟后 还没 付款那么不好意思系统会自动把座位释放掉



实现：

1. 生产者将消息(msg)和路由键(routekey)发送指定的延时交换机(exchange)上
2. 延时交换机(exchange)存储消息等待消息到期根据路由键(routekey)找到绑定自己的队列 (queue)并 把消息给它
3. 队列(queue)再把消息发送给监听它的消费者(customer）

▶**RabbitMQ集群**

◇集群模式

集群模式只能保证集群中的某个Node挂掉后应用程序还可以切换到其他Node上继续地发送和 消费消息，但并无法保证原有的消息不丢失，

◇镜像队列模式

• 将队列镜像（同步）到集群中的其他broker上，相当于是多副本冗余。如果集群中的一个节 点失效，队列能自动地切换到集群中的另一个镜像节点上以保证服务的可用性，而且消息不 丢失。

• 在RabbitMQ镜像队列中所谓的master和slave都仅仅是针对某个queue而言的，而不是node。 一个queue第一次创建所在的节点是它的master节点，其他节点为slave节点。如果master 由于某种原因失效，最先加入的slave会被提升为新的master。

master选举

最先创建队列

最先加入的slave

• 无论客户端请求到达master还是slave，最终数据都是从master节点获取。当请求到达master 节点时，master节点直接将消息返回给client，同时master节点会通过GM（Guaranteed Multicast）协议将queue的最新状态广播到slave节点。GM保证了广播消息的原子性，即要 么都更新要么都不更新。当请求到达slave节点时，slave节点需要将请求先重定向到master 节点，master节点将消息返回给client，同时master节点会通过GM协议将queue的最新状态 广播到slave节点。

▶**basicReject / basicNack / basicRecover区别**

◇channel.basicReject(deliveryTag, true);

basic.reject方法拒绝deliveryTag对应的消息，第二个参数是否requeue，true则重新入队列，否 则丢弃或者进入死信队列。该方法reject后，该消费者还是会消费到该条被reject的消息。

◇channel.basicNack(deliveryTag, false, true);

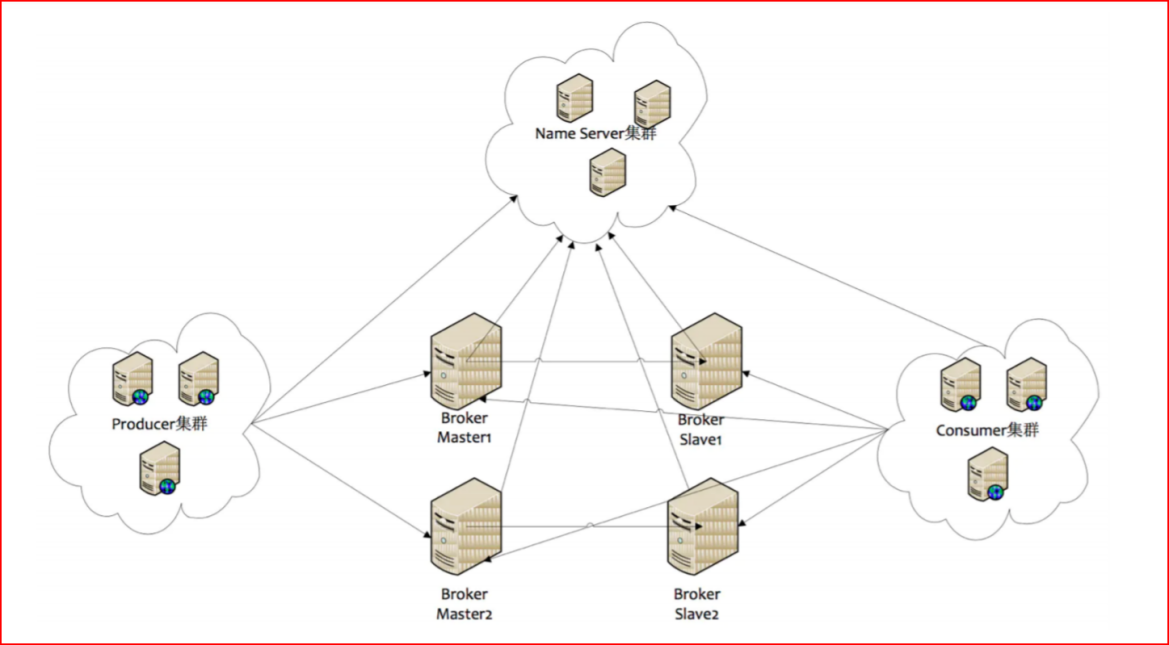
basic.nack方法为不确认deliveryTag对应的消息，第二个参数是否应用于多消息，第三个参数是 否requeue，与basic.reject区别就是同时支持多个消息，可以nack该消费者先前接收未ack的 所有消息。nack后的消息也会被自己消费到。

◇channel.basicRecover(true);

basic.recover是否恢复消息到队列，参数是是否requeue，true则重新入队列，并且尽可能的将 之前recover的消息投递给其他消费者消费，而不是自己再次消费。false则消息会重新被投递给 自己。

**RocketMQ**

▶**RocketMQ的角色介绍**



◇NameServer

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。

◇Producer

Producer与NameServer集群中的其中一个节点建立长连接，定期从 NameServer获取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。

◇Consumer

Consumer与NameServer集群中的其中一个节点建立长连接，定期从 NameServer获取Topic路 由信息

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

消费者在向Master拉取消息时，Master服务器会根据拉取偏移量与最大偏移量的距离（判断是 否读老消息，产生读I/O），以及从服务器是否可读等因素建议下一次是从Master还 是Slave拉 取

◇ProducerGroup

同一类Producer的集合，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

◇ConsumerGroup

同一类Consumer的集合，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）。

◇Broker

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BrokerName，不同的BrokerId来定义，BrokerId为0 表示Master，非0表示Slave。Master也可以部署多个。

每个Broker与NameServer集群中的所有节点建立长连接，定时注册Topic信息到所有 NameServer。 注意：当前RocketMQ版本在部署架构上支持一Master多Slave，但只有 BrokerId=1 的从服务器才会参与消息的读负载。

◇Topic

Topic表示消息的第⼀级类型，⽐如⼀个电商系统的消息可以分为：交易消息、物流消息等。⼀ 条消息必须有⼀个Topic。

◇Tag

Tag表示消息的第⼆级类型，⽐如交易消息⼜可以分为：交易创建消息，交易完成消息等。 RocketMQ提供2级消息分类，⽅便灵活控制。

◇Message Queue

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，⻓度⽆限的数据结构，所谓⻓度⽆限是指队列中 的每个存储单元都是定⻓，访问其中的存储单元使⽤。Offset 来访问， offset 为 java long 类型， 64 位

▶**执行流程**

◇启动NameServer，NameServer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。

◇Broker启动，跟所有的NameServer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，NameServer集群中就有Topic 跟Broker的映射关系。

◇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。

◇Producer发送消息，启动时先跟NameServer集群中的其中一台建立长连接，并从 NameServer中获 取当前发送的Topic存在哪些Broker上，轮询从队列列表中选择一个队列， 然后与队列所在的Broker 建立长连接从而向Broker发消息。

◇Consumer跟Producer类似，跟其中一台NameServer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

▶**相关术语**

◇广播消费

一条消息被多个 Consumer 消费，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，广播消费中的 Consumer Group 概念可以 认为在消息划分方面无意义。

◇集群消费

一个 Consumer Group 中的 Consumer 实例平均分摊消费消息。例如某个 Topic 有 9 条消息， 其中一个 Consumer Group 有 3 个实例(可能是 3 个进程，或者3台机器)，那举每个实例只消 费其中的 3条消息。

◇顺序消息

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。

• 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，但是一旦发生通信异常，Broker 重 启，由于队列总数发生发化，哈希取模后定位的队列会变化，产生短暂的消息顺序不一致。 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。

• 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，自动切换功能目前还未实现

◇消息队列

在 RocketMQ 中，所有消息队列都是持久化的，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，offset 为 java long 类型， 64 位， 理论上在 100 年内不会溢出，所以认为为是长度无限，另外队列中只保存最近几天的 数据，之前的数据 会按照过期时间来删除。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

• Push

Push方式里，消费者把长轮询的动作封装，并注册MessageListener监听器，取到消息后，

唤醒MessageListener的consumeMessage()来消费，对用户而言，感觉消息是被推送过来的

Push本质都是采用消费端主动拉取的方式，即consumer轮询从broker拉取消息。

• Pull

Pull方式里，取消息的过程需要用户自己主动调用，首先通过Topic拿到消息队列的集合， 遍历消息队列集合，然后针对每个消息队列批量取消息，一次取完后，记录该队列下一次 要取的开始offset，直到取完了，再换另一个消息队列。

▶**集群模式**

◇单Master模式

这种方式风险较大，一旦Broker重启或者宕机时，会导致整个服务不可用。不建议线上环境使 用,可以用于本地测试。

◇多Master模式

优点：配置简单，单个Master宕机或重启维护对应用无影响

缺点：单台机器宕机期间，这台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 会受到影响。

◇多Master多Slave模式（异步）

优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，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。

◇多Master多Slave模式（同步）

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。

▶**消费发送**

◇消息发送步骤

• 设置GroupName。

• 设置InstanceName

当一个Jvm需要启动多个Producer的时候，通过设置不同的InstanceName来区分，不设置 的话系统使用默认名称“DEFAULT”。

• 设置发送失败重试次数

当网络出现异常的时候，这个次数影响消息的重复投递次数。想保证不丢消息，可以设置 多重试几次。

• 设置NameServer地址

• 组装消息并发送

◇消息发送返回状态

• FLUSH\_DISK\_TIMEOUT

同步刷盘超时

表示没有在规定时间内完成刷盘（需要Broker的刷盘策略被设置成SYNC\_FLUSH才会报这 个错误），一般异步刷盘

• FLUSH\_SLAVE\_TIMEOUT

主从同步超时

表示在主备方式下，并且Broker被设置成SYNC\_MASTER方式，

没有在设定时间内完成主从同步。

• SLAVE\_NOT\_AVAILABLE

没有可用的从节点

这个状态产生的场景和FLUSH\_SLAVE\_TIMEOUT类似，表示在主备方式下，并且Broker被设 置成SYNC\_MASTER，但是没有找到被配置成Slave的Broker

• SEND\_OK

表示发送成功，

发送成功的具体含义，比如消息是否已经被存储到磁盘？消息是否被同步到了Slave上？消 息在Slave上是否被写入磁盘？需要结合所配置的刷盘策略、主从策略来定。这个状态还可 以简单理解为，没有发生上面列出的三个问题状态就是SEND\_OK

◇提升写入的性能

• 增加Producer的并发量

使用多个Producer同时发送

• 采用Oneway方式发送

在一些对速度要求高，但是可靠性要求不高的场景下，比如日志收集类应用， 可以采用 Oneway方式发送

▶**消息消费**

◇提高Consumer的处理能力

• 提高消费并行度

在同一个ConsumerGroup下（Clustering方式），可以通过增加Consumer实例的数量来提

高并行度。

通过提高单个Consumer实例中的并行处理的线程数，可以在同一个Consumer内增加

并行度来提高吞吐量（设置方法是修改consumeThreadMin和consumeThreadMax）。

• 以批量方式进行消费

可以通过批量方式消费来提高消费的吞吐量。实现方法是设置Consumer的

consumeMessageBatchMaxSize这个参数，默认是1，如果设置为N，在消息多的时候每次

收到的是个长度为N的消息链表

▶**消息存储**

◇存储结构

CommitLog：

消息存储文件，所有消费主题的消息都存储在CommitLog文件中。

CommitLog里面一个MappedFileQueue，MappedFileQueue里面多个MappedFile

ConsumeQueue：逻辑消费队列，保存了指定Topic下的队列消息在CommitLog中的起始物理偏 移量offset，消息大小size和消息Tag的HashCode值

IndexFile：消息索引文件，主要存储消息Key 与 Offset的对应关系，查找消息的方法

Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog，1G）来存储

每次读取消息队列先读取consumerQueue,然后再通过consumerQueue去commitLog 中拿到消息主体。

◇Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog，1G）来存储

◇每次读取消息队列先读取consumerQueue,然后再通过consumerQueue去commitLog中拿到消息主 体。

▶**过滤消息**

消费者订阅消息时再做消息过滤的

Tag过滤方式

Consumer端在订阅消息时除了指定Topic还可以指定TAG，如果一个消息有多个TAG，可以用|| 分隔

▶**同步复制和异步复制**

◇同步复制

同步复制方式是等Master和Slave均写成功后才反馈给客户端写成功状态

在同步复制方式下，如果Master出故障，Slave上有全部的备份数据，容易恢复，但是同步复制 会增大数据写入延迟，降低系统吞吐量。

◇异步复制

异步复制方式是只要Master写成功 即可反馈给客户端写成功状态。

在异步复制方式下，系统拥有较低的延迟和较高的吞吐量，但是如果Master出了故障，有些数 据因为没有被写 入Slave，有可能会丢失；

◇同步复制和异步复制是通过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

◇同步双写

主从之间配置成SYNC\_MASTER的复制方式，刷盘方式配置成ASYNC\_FLUSH

也就是同步复制，异步双写

▶**刷盘机制**

RocketMQ的所有消息都是持久化的，先写入系统 PageCache，然后刷盘。 访问时，直接从页缓存读 取。

刷盘方式

同步刷盘

异步刷盘

▶**消息重试**

◇顺序消息的重试

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）

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。

◇无序消息的重试

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

消息队列 RocketMQ 默认允许每条消息最多重试 16 次

▶**死信队列**

◇RocketMQ中消息重试超过一定次数后（默认16次）就会被放到死信队列中

◇死信消息具有以下特性

不会再被消费者正常消费。

有效期为 3 天，3 天后会被自动删除。

◇死信队列具有以下特性

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，消息队列 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。

▶**延迟消费**

◇延迟消费的原理

每一个延迟消息的主题都被暂时更改为延迟消费主题SCHEDULE\_TOPIC\_XXXX，并且通过 ScheduleMessageService根据延迟级别延迟消息变更了新的队列Id

◇缺点

时间精度不够：不能精确时间消费

不支持任意时间段的消息延迟。只支持上述18个时间的延迟级别；

延迟消费最长时间为2h

◇broker有配置项messageDelayLevel，默认值为“1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m

8m 9m 10m 20m 30m 1h 2h”，18个level。可以配置自定义messageDelayLevel。注意，

messageDelayLevel是broker的属性，不属于某个topic。

◇发消息时，设置delayLevel等级即可：

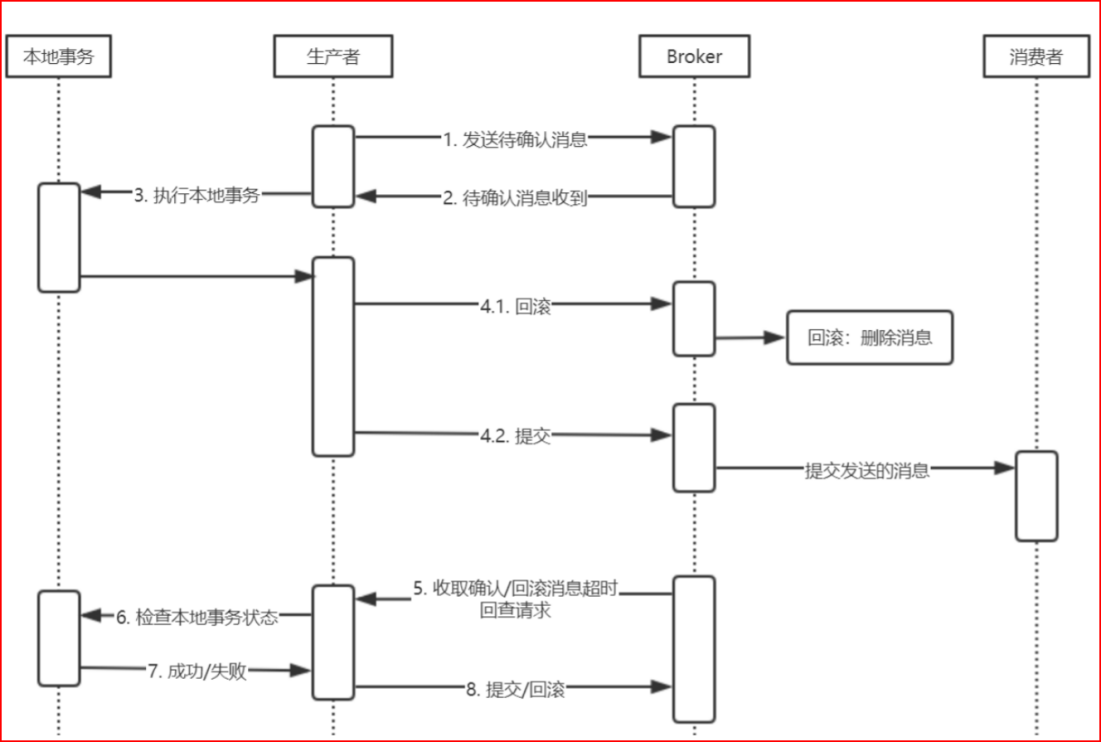
msg.setDelayLevel(level)。level有以下三种情况：

level == 0，消息为非延迟消息

1<=level<=maxLevel，消息延迟特定时间，例如level==1，延迟1s

level > maxLevel，则level== maxLevel，例如level==20，延迟2h

▶**事务消息**



◇RocketMQ采用两阶段提交的方式实现事务消息

◇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果。

(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）

◇补偿流程

(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

▶**限流**

消费端

设置最大消费线程数

每次拉取消息条数等

Sentinel

▶**RocketMQ Broker中的消息被消费后会立即删除吗？**

不会，每条消息都会持久化到CommitLog中，每个Consumer连接到Broker后会维持消费进度信息， 当有消息消费后只是当前Consumer的消费进度（CommitLog的offset）更新了

▶**那么消息会堆积吗？什么时候清理过期消息？**

4.6版本默认48小时后会删除不再使用的CommitLog文件，72小时

检查这个文件最后访问时间，判断是否大于过期时间

指定时间删除，默认凌晨4点

▶**RocketMQ消费模式有几种？**

◇集群消费

一条消息只会被同Group中的一个Consumer消费

◇广播消费

消息将被 一个Consumer Group 下的各个 Consumer 实例都消费一遍。

▶**消费消息是push还是pull？**

RocketMQ没有真正意义的push，都是pull，虽然有push类，实际底层实现采用的是长轮询机制，即 拉取方式。

▶**如何让RocketMQ保证消息的顺序消费**

同一topic，同一个QUEUE，发消息的时候一个线程去发送消息，消费的时候 一个线程去消费一个 queue里的消息。

▶**怎么保证消息发到同一个queue**

Rocket MQ提供MessageQueueSelector接口，可以自己重写select方法

举个最简单的例子：判断i % 2 == 0，那就都放到queue1里，否则放到queue2里。

▶**RocketMQ如何保证消息不丢失**

◇生产者

同步发送send()

发送失败后可以重试，设置重试次数。默认3次。

◇Broker

非集群，修改刷盘策略为同步刷盘。默认情况下是异步刷盘的

集群，主从模式，异步刷盘

◇消费者

手动ack确认

▶**如果Consumer和Queue不对等，上线了多台也在短时间内无法消费完堆积的消息怎么办？**

◇准备一个临时的topic

◇上线一台Consumer做消息的搬运工，把原来Topic中的消息挪到新的Topic里，不做业务逻辑处理， 只是挪过去

◇上线N台Consumer同时消费临时Topic中的数据

◇改bug

◇恢复原来的Consumer，继续消费之前的Topic

▶**堆积的消息会不会进死信队列？**

消息在消费失败后会进入重试队列（%RETRY%+ConsumerGroup）,才会进入死信队列 （%DLQ%+ConsumerGroup）。

▶**RocketMQ为何不用ZooKeeper**

RocketMQ不需要进行Master选举，只需要一个轻量级的元数据服务器就足够了。

NameServer并没有提供类似Zookeeper的watcher机制， 而是采用了每30s心跳机制。

▶**RocketMQ心跳机制**

◇单个Broker跟所有NameServer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。NameServer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NameServer不会主动通知Producer、 Consumer有Broker宕机。

◇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(rebalance)。

◇Producer每30秒从NameServer获取Topic跟Broker的映射关系，更新到本地内存中。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。

▶**RocketMQ的高性能**

◇顺序写

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！

◇零拷贝

◇页缓存

◇跳跃读(尽量命中PageCache)，

▶**设计分布式消息中间件**

快速扩容

集群

持久化

高可用性

数据0丢失

▶**高吞吐量下如何优化生产者和消费者的性能?**

◇开发

同一个组，多机部署，并行消费

消费速度最好快于生产速度

•增加消费者

•增加单个消费者消费速度，线程池

•增加单个消费者消费消息数目

◇运维

网卡调优

Jvm调优

多线程与cpu调优

Page Cache

▶**任何一台Broker突然宕机了怎么办**

◇Broker主从架构以及多副本策略。Master收到消息后会同步给Slave，这样一条消息就不止一份了， Master宕机了还有slave中的消息可用，保证了MQ的可靠性和高可用性。

◇RocketMQ 4.5 版本发布后，可以采用 RocketMQ on DLedger 方式进行部署。DLedger commitlog 代 替了原来的 commitlog，使得 commitlog 拥有了选举复制能力

▶**RocketMQ如何做负载均衡**

◇生产者

Rocket MQ提供MessageQueueSelector接口

• hash

• 随机

• machine room

◇消费者

平均分配算法

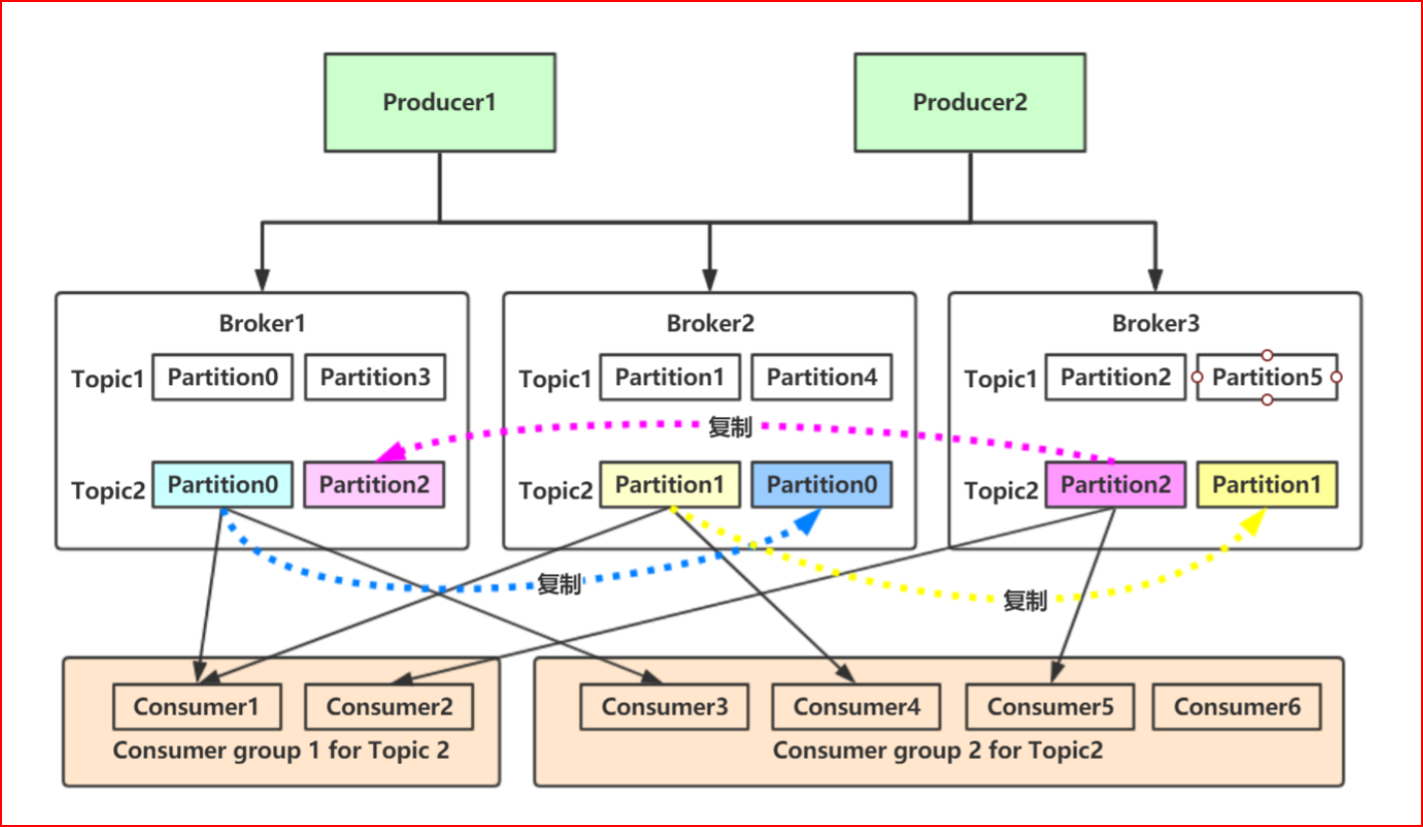
当消费负载均衡时候，consumer和queue不对等的时候会发生什么？

• 如果Consumer少于queue的个数，则会存在部分Consumer消费多个queue的情况

• 如果Consumer等于queue的个数，那就是一个Consumer消费一个queue

• 如果Consumer个数大于queue的个数，那么会有部分Consumer空余出来，白白的浪费 了。

**Kafka**



▶**优点**

◇高吞吐量

单机每秒处理几十上百万的消息量。即使存储了许多TB的消息，它也保持稳定的性能。

◇高性能

单节点支持上千个客户端

零停机

零数据丢失

◇持久化

零拷贝

顺序读写

页缓存

◇易扩展

所有的Producer、Broker和Consumer都会有多个，均为分布式 的。无需停机即可扩展机器。 多个Producer、Consumer可能是不同的应用。

◇可靠性

Kafka是分布式，分区，复制和容错的。

◇支持多种客户端语言

Kafka支持Java、.NET、PHP、Python等多种语言。

▶**应用场景**

◇日志收集

一个公司可以用Kafka可以收集各种服务的Log，通过Kafka以统一接口服务的方式开放 给各种 Consumer；

◇消息系统

解耦生产者和消费者、缓存消息等；

◇用户活动跟踪

Kafka经常被用来记录Web用户或者App用户的各种活动，如浏览网页、搜索、点击 等活动， 这些活动信息被各个服务器发布到Kafka的Topic中，然后消费者通过订阅这些Topic来做实时的

◇监控分析

亦可保存到数据库； 运营指标：Kafka也经常用来记录运营监控数据。包括收集各种分布式应用 的数据，生产各种操作的 集中反馈，比如报警和报告；

◇流式处理

比如Spark Streaming和Storm

▶**核心概念**

◇Producer

• 生产者创建消息，该角色将消息发布到Kafka的topic中。broker接收到生产者发送的消息后， broker将该消息追加到 当前用于追加数据的 segment 文件中

• 一般情况下，一个消息会被发布到一个特定的主题上

1、默认情况下通过轮询把消息均衡地分布到主题的所有分区上。

2、在某些情况下，生产者会把消息直接写到指定的分区。这通常是通过消息键和分区器来 实现的，分区器为键生成一个散列值，并将其映射到指定的分区上。这样可以保证包含 同一个键的 消息会被写到同一个分区上。

3、生产者也可以使用自定义的分区器，根据不同的业务规则将消息映射到分区。只需要实 现Partitioner接口

◇Consumer

• 消费者读取消息，消费者订阅一个或多个主题，并按照消息生成的顺序读取它们。

• 消费者通过检查消息的偏移量来区分已经读取过的消息。偏移量是一种元数据，它是一个不 断递增的整数值，消费者把每个分区最后读取的消息偏移量保存在Zookeeper 或Kafka 上， 如果消费者关闭或重启，它的读取状态不会丢失。

• 消费者是消费组的一部分。群组保证每个分区只能被一个消费者使用。如果一个消费者失效， 消费组里的其他消费者可以接管失效消费者的工作，再平衡，分区重新分配

◇Broker

一个独立的Kafka 服务器被称为broker。

broker 为消费者提供服务，对读取分区的请求作出响应，返回已经提交到磁盘上的消息。

1. 如果某topic有N个partition，集群有N个broker，那么每个broker存储该topic的一个 partition。
2. 如果某topic有N个partition，集群有(N+M)个broker，那么其中有N个broker存储该topic 的一 个partition，剩下的M个broker不存储该topic的partition数据。

3. 如果某topic有N个partition，集群中broker数目少于N个，那么一个broker存储该topic的一 个或多个partition。在实际生产环境中，尽量避免这种情况的发生，这种情况容易导致 Kafka 集群数据不均衡。

◇Topic

发布到Kafka集群的消息都有一个Topic。 物理上不同Topic的消息分开存储。Topic就好比数据 库的表，尤其是分库分表之后的逻辑表

◇Partition

• 主题可以被分为若干个分区，一个分区就是一个提交日志。

• 消息以追加的方式写入分区，然后以先入先出的顺序读取。

• 无法在整个主题范围内保证消息的顺序，但可以保证消息在单个分区内的顺序。

• 在需要严格保证消息的消费顺序的场景下，需要将partition数目设为1

• Kafka 通过分区来实现数据冗余和伸缩性。

• Partition和Consumer关系

partition = consumer（3个partiton --> 3个consumer）：一对一

partition > consumer（3个partiton --> 2个consumer）

consumer1会消费partition0/partition1分区

consumer2会消费partition2分区

partition < consumer（3个partition --> 4个或以上consumer）

仍然只有3个consumer对应3个partition，其他的consumer无法消费消息

• 分配策略

一种是Range(默认)、 一种是RoundRobin（轮询）、 一种是StickyAssignor(粘性)。

◇Replicas

Kafka 使用主题来组织数据，每个主题被分为若干个分区，每个分区有多个副本。那些副本被保 存在broker 上，每个broker可以保存成百上千个属于不同主题和分区的副本。

• 副本类型

首领副本

每个分区都有一个首领副本。为了保证一致性，所有生产者请求和消费者请求都会经 过这个副本。

跟随者副本

首领以外的副本都是跟随者副本。跟随者副本不处理来自客户端的请求，它们唯一的 任务就是从首 领那里复制消息，保持与首领一致的状态。如果首领发生崩溃，其中 的一个跟随者会被提升为新首领。

• 副本AR

分区中的所有副本统称为AR（Assigned Repllicas）。AR=ISR+OSR

○ISR（In-Sync Replicas）

所有与leader副本保持一定程度同步的副本（包括Leader）

○OSR(Out-Sync Relipcas)

与leader副本同步滞后过多的副本（不包括leader）副本

◇Offset

• 每个topic可以划分多个分区，同一topic下的不同分区包含的消息是不同的。

• 每个消息在被添加到分区时，都会被分配一个 oﬀset，它是消息在此分区中的唯一编号，Kafka 通过oﬀset保证消息在分区内的顺序，oﬀset的顺序不跨分区，即Kafka只保证在同一个分区 内 的消息是有序的；

• 每次消费一个消息并且提交后，会保存当前消费到的最近的一个oﬀset。

• 老版本的kafka中，offet是保存在zookeeper上。而现在的kafka中，提供了一个\_\_consumer\_o ﬀsets 的一个topic，保存每个consumer group某一时刻提交的oﬀset信息。

▶**生产者 数据生产流程**

1. Producer创建时，会创建一个Sender线程并设置为守护线程。
2. 生产消息时，内部其实是异步流程；生产的消息先经过拦截器->序列化器->分区器，然后将消息缓 存在缓冲区（该缓冲区也是在Producer创建时创建）。
3. 批次发送的条件为：缓冲区数据大小达到batch.size或者linger.ms达到上限，哪个先达到就算哪个。
4. 批次发送后，发往指定分区，然后落盘到broker；如果生产者配置了retrires参数大于0并且失 败原因允许重试，那么客户端内部会对该消息进行重试。
5. 落盘到broker成功，返回生产元数据给生产者。
6. 元数据返回有两种方式：一种是通过阻塞直接返回，另一种是通过回调返回。

▶**消费者**

◇位移提交

• 自动提交

○开启自动提交： enable.auto.commit=true

○配置自动提交间隔：Consumer端： auto.commit.interval.ms ，默认 5s

○Kafka 会保证在开始调用 poll 方法时，提交上次 poll 返回的所有消息。从顺序上来说， poll 方法的逻辑是先提交上一批消息的位移，再处理下一批消息，因此它能保证不出现 消费丢失的情况

○自动提交位移的一个问题在于，它可能会出现重复消费

○我认为还是会造成数据丢失，poll方法后数据入库失败，还是会数据丢失

• 同步提交

○使用 KafkaConsumer#commitSync()：会提交 KafkaConsumer#poll() 返回的最新 offset

○该方法为同步操作，等待直到 offset 被成功提交才返回

○问题

重复消费，除非消费一次提交一次

会影响 TPS

• 异步提交

KafkaConsumer#commitAsync()

问题：出现问题不会自动重试

◇再均衡

• 触发条件

○消费者组内成员发生变更

这个变更包括了增加和减少消费者，比如消费者宕机退出消费组。

○主题的分区数发生变更

kafka目前只支持增加分区，当增加的时候就会触发重平衡

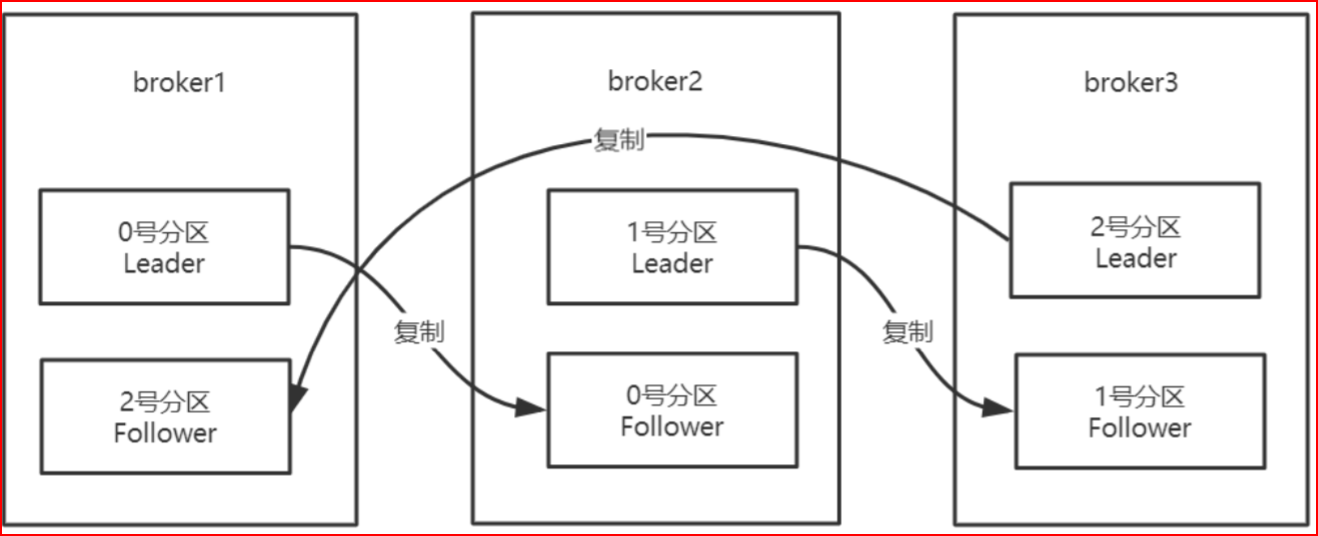
○订阅的主题发生变化

当消费者组使用正则表达式订阅主题，而恰好又新建了对应的主题，就会触发重平衡

• 问题

重平衡过程中，消费者无法从kafka消费消息，这对kafka的TPS 影响极大。

▶**分区** Leader副本选举



Kafka中Leader分区选举，通过维护一个动态变化的ISR集合来实现，一旦Leader分区丢掉，则从 ISR 中随机挑选一个副本做新的Leader分区

为什么不用少数服从多数的方法

ISR副本本身和所有与leader副本保持一定程度同步

所有的ISR副本都失败了怎么办

可以等待ISR中的副本任何一个恢复，接着对外提供服务，需要时间等待。

从OSR中选出一个副本做Leader副本，此时会造成数据丢失

▶**偏移量**

auto.offset.reset值含义解释

◇earliest

当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，从头开始消费

◇latest

当各分区下有已提交的offset时，从提交的offset开始消费；无提交的offset时，消费新产生的 该分区下的数据

◇none

topic各分区都存在已提交的offset时，从offset后开始消费；只要有一个分区不存在已提交的 offset，则抛出异常

▶**物理存储**

◇日志存储

每个主题又可以分为一个或多个分区。

每个分区各自存在一个记录消息数据的日志文件

• 日志存储时间

默认保存数据时间为7天或者大小1G

◇磁盘存储

• 零拷贝

将磁盘文件读取到操作系统内核缓冲区

将内核缓冲区的数据copy到应用缓冲区

应用缓冲区的数据copy到socket网络发送缓冲区

将socket发送缓冲区的数据copy到网络协议栈

kafka的两个过程：

1. 网络数据持久化到磁盘 (Producer 到 Broker)
2. 磁盘文件通过网络发送（Broker 到 Consumer）

数据落盘通常都是非实时的，Kafka的数据并不是实时的写入硬盘，它充分利用了现代操作系统分页 存储来利用内存提高I/O效率。

• 页缓存

Memory Mapped Files(后面简称mmap)内存映射文件

工作原理是直接利用操作系统的Page来实现文件到物理内存的直接映射，对物理内存的操 作会被同步到硬盘上

磁盘中的数据缓存到内存中，把对磁盘的访问变为对内存的访问。

• 顺序写入

Kafka 在设计时采用了文件追加的方式来写入消息，即只能在日志文件的尾部追加新的消 息，并且也不允许修改已写入的消息，这种方式属于典型的顺序写盘的操作，所以就算Kafka 使用磁盘作为存储介质，也能承载非常大的吞吐量

▶**消息可靠性保障**

◇生产消息确认投递

配置request.required.acks属性来确认消息投递方式

0---不进行消息接收成功的确认；

1---分区主副本接收成功时确认；

-1---所有的分区副本都接收成功时确认；

◇消费消息确认消费

使用同步提交

enable.auto.commit 设置自动提交false

▶**消费重复消费**

原因

手动同步提交批量数据

解决

每次消费时幂等校验

▶**MQ积压几百万条数据怎么办**

◇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

◇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 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，直接均匀轮询写入临时建立好的10倍数量的queue

◇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 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 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

▶**Kafka为什么速度这么快？**

写

顺序写入

页缓存Memory Mapped Files

读

零拷贝

批量压缩

通过压缩的形式传输，并且在日志中也可以保持压缩格式，直到被消费者解压缩

压缩协议

Gzip

Snappy

▶**Zookeeper在Kafka中的应用**

◇Broker注册

Broker服务器列表记录

◇Topic注册

分区信息及与Broker的对应关系

◇消费者注册

对 消费者分组 中的 消费者 的变化注册监听

◇生产者负载均衡

生产者需要将消息合理地发送到这些分布式的Broker上

◇消费者负载均衡

每个消费者分组包含若干消费者，每条消息都只会发送给分组中的一个消费者，不同的消费者 分组消费自己特定的Topic下面的消息，互不干扰。

◇分区与消费者的关系

在 Zookeeper 上记录 消息分区 与 Consumer 之间的关系，每个消费者一旦确定了对一个消息 分区的消费权力，需要将其Consumer ID 写入到 Zookeeper 对应消息分区的临时节点上

◇消费进度Offset 记录

需要定时地将分区消息的消费进度Offset记录到Zookeeper上