MQ背景&选型

消息队列作为高并发系统的核心组件之一，能够帮助业务系统解构提升开发效率和系统稳定性。主要具有以下优势：

- 削峰填谷（主要解决瞬时写压力大于应用服务能力导致消息丢失、系统奔溃等问题）
- 系统解耦（解决不同重要程度、不同能力级别系统之间依赖导致一死全死）
- 提升性能（当存在一对多调用时，可以发一条消息给消息系统，让消息系统通知相关系统）
- 蓄流压测（线上有些链路不好压测，可以通过堆积一定量消息再放开来压测）

目前主流的MQ主要是Rocketmq、kafka、Rabbitmq，Rocketmq相比于Rabbitmq、kafka具有主要优势特性有：

- 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）

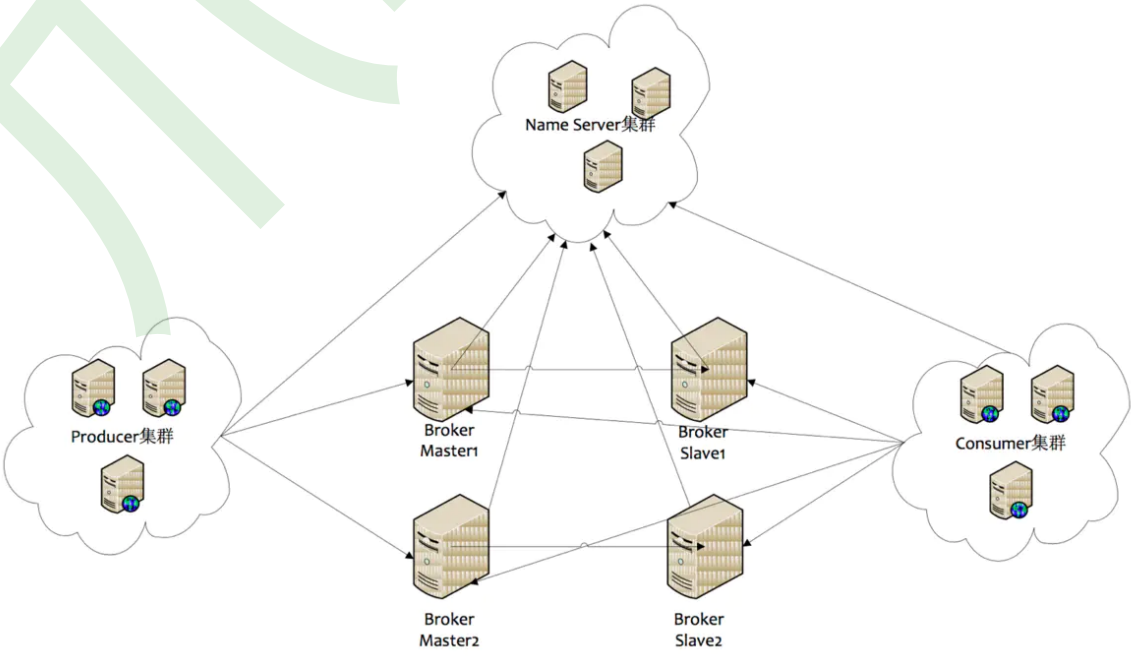
Rocketmq、kafka、Rabbitmq的详细对比，请参照下表格：

		kafka	RocketMQ	RabbitMQ
定位	设计定位	系统间的数据流管道，实时数据处理。 例如：常规的消息系统、网站活性跟踪，监控数据， 志收集、处理等	非日志的可靠消息传输。 例如：订单，交易，充值，流计算，消息推送，日 流式处理，binglog分发等	可靠消息传输。和RocketMQ类似。
基础对比	成熟度	日志领域成熟	成熟	成熟
	所属社区 / 公司	Apache	Alibaba开发，已加入到Apache下	Mozilla Public License
	社区活跃度	高	中	高
	API完备性	高	高	高
	文档完备性	高	高	高
	开发语言	Scala	Java	Erlang
	支持协议	一套自行设计的基于TCP的二进制协议	自己定义的一 套 (社区提供 JMS一不成熟)	AMQP
	客户端语言	C/C++、Python、Go、Erlang、.NET、Ruby、Node.js PHP等	Java	Java、C、 C++、 Python、 PHP、Perl 等
	持久化方式	磁盘文件	磁盘文件	内存、文件
可用性、 靠性比较	部署方式	单机 / 集群	单机 / 集群	单机 / 集群
	集群管理	zookeeper	name server	
	选主方式	从ISR中自动选举一个leader	不支持自动选主。通过设定brokername、brokerId 实现。brokername相同，brokerid=0时为maser， 他为slave	最早加入集群的broker
	可用性	非常高 分布式、主从	非常高 分布式、主从	高 主从，采用镜像模式实现，数据量大时可能产生性 能瓶颈
	主从切换	自动切换 N个副本，允许N-1个失效；master失效以后自动从is 中选择一个主；	不支持自动切换 master失效以后不能向master发送信息，consume 大概30s（默认）可以感知此事件，此后从slave消 费；如果master无法恢复，异步复制时可能出现部 分信息丢失	自动切换 最早加入集群的slave会成为master；因为新加入的 slave不会同步master之前的数据，所以可能会出现 部分数据丢失
	数据可靠性	很好 支持producer单条发送、同步刷盘、同步复制，但这 场景下性能明显下降。	很好 producer单条发送，broker端支持同步刷盘、异步 刷盘，同步双写，异步复制。	好 producer支持同步 / 异步ack。支持队列数据持久 化，镜像模式中支持主从同步
	消息写入性能	非常好 每条10个字节测试：百万条/s	很好 每条10个字节测试：单机单broker约7w/s，单机3 broker约12w/s	RAM约为RocketMQ的1/2， Disk的性能约为RAM性能的1/3
	性能的稳定性	队列/分区多时性能不稳定，明显下降。 消息堆积时性能稳定	队列较多、消息堆积时性能稳定	消息堆积时，性能不稳定、明显下降
	单机支持的队列数	单机超过64个队列/分区，Load会发生明显的飙高现 象，队列越多，load越高，发送消息响应时间变长	单机支持最高5万个队列，Load不会发生明显变化	依赖于内存
	堆积能力	非常好 消息存储在log中，每个分区一个log文件	非常好 所有消息存储在同一个commit log中	一般 生产者、消费者正常时，性能表现稳定；消费者不 消费时，性能不稳定
	复制备份	消息先写入leader的log，followers从leader中pull pull到数据以后先ack leader，然后写入log中。 ISR中维护与leader同步的列表，落后太多的follwer 被删除掉	同步双写 异步复制：slave启动线程从master中拉数据	普通模式下不复制； 镜像模式下：消息先到master，然后写到slave上。 入集群之前的消息不会被复制到新的slave上。
	消息投递实时性	毫秒级 具体由consumer轮询间隔时间决定	毫秒级 支持pull、push两种模式，延时通常在毫秒级	毫秒级
功能对比	顺序消费	支持顺序消费 但是一台Broker宕机后，就会产生消息乱序	支持顺序消费 在顺序消息场景下，消费失败时消费队列将会暂 停	支持顺序消费 但是如果一个消费失败，此消息的顺序会被打乱
	定时消息	不支持	开源版本仅支持定时Level	不支持
	事务消息	不支持	支持	不支持
	Broker端消息过滤	不支持	支持 通过tag过滤，类似于子topic	不支持
	消息查询	不支持	支持 根据MessageId查询 支持根据MessageKey查询消息	不支持
	消费失败重试	不支持失败重试 offset存储在consumer中，无法保证。 0.8.2版本后支持将offset存储在zk中	支持失败重试 offset存储在broker中	支持失败重试
	消息重新消费	支持通过修改offset来重新消费	支持按照时间来重新消息	

	发送端负载均衡	可自由指定	可自由指定	需要单独loadbalancer支持
	消费并行度	消费并行度和分区数一致	顺序消费：消费并行度和分区数一致 乱序消费：消费服务器的消费线程数之和	镜像模式下其实也是从master消费
	消费方式	consumer pull	consumer pull / broker push	broker push
	批量发送	支持 默认producer缓存、压缩，然后批量发送	不支持	不支持
	消息清理	指定文件保存时间，过期删除	指定文件保存时间，过期删除	可用内存少于40%（默认），触发gc，gc时找到相邻的两个文件，合并right文件到left。
	访问权限控制	无	无	类似数据库一样，需要配置用户名密码
运维	系统维护	Scala语言开发，维护成本高	java语言开发，维护成本低	Erlang语言开发，维护成本高
	部署依赖	zookeeper	nameserver	Erlang环境
	管理后台	官网不提供，第三方开源管理工具可供使用；不用重新开发	官方提供，rocketmq-console	官方提供rabbitmqadmin
	管理后台功能	**Kafka Web Conslole** Brokers列表；Kafka 集群中 Topic列表，及对应的Partition、LogSize等信息；Topic对应的Consumer Groups、Offset、Lag等信息； 生产和消费流量图、消息预览 **KafkaOffsetMonitor：** Kafka集群状态；Topic、Consumer Group列表；图形展示topic和consumer之间的关系；图形化展示consumer的Offset、Lag等信息 **Kafka Manager** 管理几个不同的集群；监控集群的状态(topics, brokers，副本分布，分区分布)；产生分区分配(Generate partition assignments)基于集群的当前态；重新分配分区	Cluster、Topic、Connection、NameServ、Messa Broker、Offset、Consumer	overview、connections、channels、exchanges、queues、admin
总结	优点	1、在高吞吐、低延迟、高可用、集群热扩展、集群容错上有非常好的表现； 2、producer端提供缓存、压缩功能，可节省性能，提高效率。 3、提供顺序消费能力 4、提供多种客户端语言 5、生态完善，在大数据处理方面有大量配套的设施。	1、在高吞吐、低延迟、高可用上有非常好的表现，消息堆积时，性能也很好。 2、api、系统设计都更加适在业务处理的场景。 3、支持多种消费方式。 4、支持broker消息过滤。 5、支持事务。 6、提供消息顺序消费能力；consumer可以水平扩展，消费能力很强。 7、集群规模在50台左右，单日处理消息上百亿；经历过大数据量的考验，比较稳定可靠。	1、在高吞吐量、高可用上较前两者有所不如。 2、支持多种客户端语言；支持amqp协议。 3、由于erlang语言的特性，性能也比较好；使用RAM模式时，性能很好。 4、管理界面较丰富，在互联网公司也有较大规模的应用；
	缺点	1、消费集群数目受到分区数目的限制。 2、单机topic多时，性能会明显降低。 3、不支持事务	1、相比于kafka，使用者较少，生态不够完善。消堆积、吞吐率上也有所不如。 2、不支持主从自动切换，master失效后，消费者要一定的时间才能感知。 3、客户端只支持Java	1、erlang 语言难度较大。集群不支持动态扩展。 2、不支持事务、消息吞吐能力有限 3、消息堆积时，性能会明显降低

RocketMQ集群概述

1. RocketMQ集群部署结构



1) Name Server

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。

2) Broker

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。

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。

3) Producer

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。

4) Consumer

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。

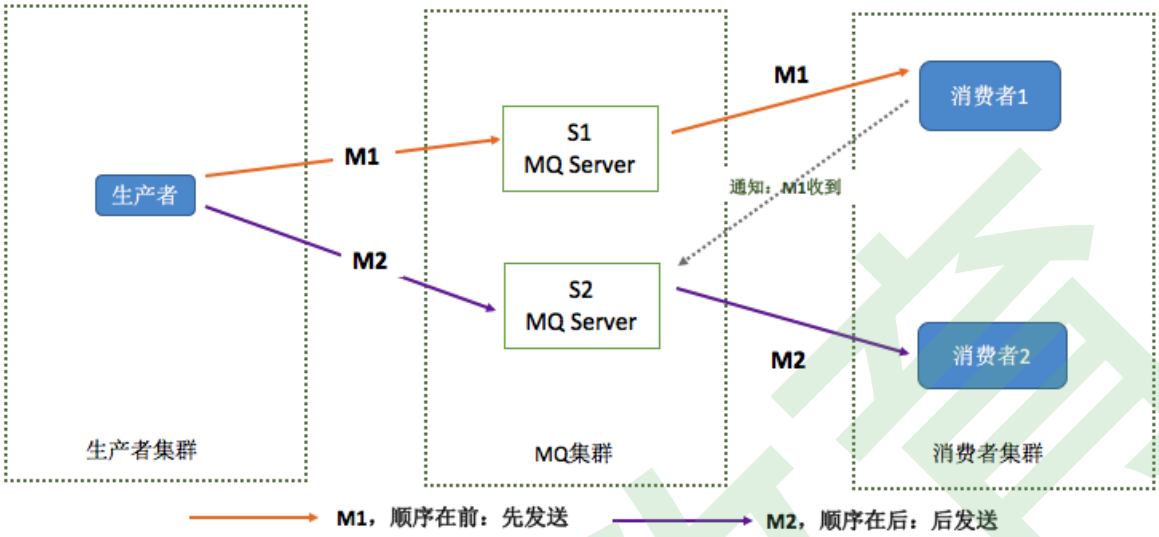
关键特性及其实现原理

顺序消费

消息有序指的是可以按照消息的发送顺序来消费。

例如：一笔订单产生了3条消息，分别是订单创建、订单付款、订单完成。消费时，必须按照顺序消费才有意义，与此同时多笔订单之间又是可以并行消费的。

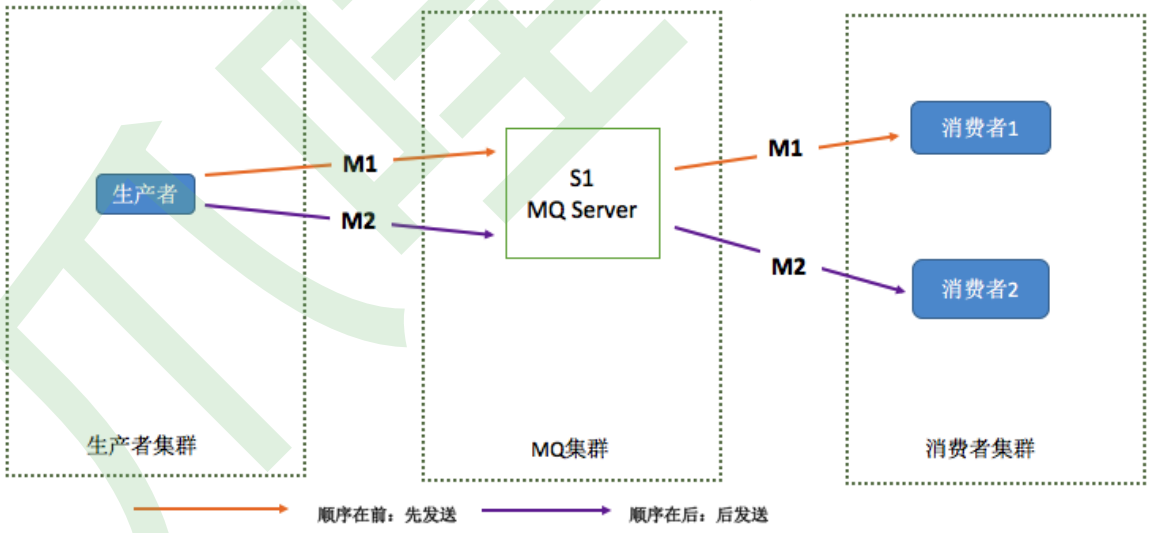
例如生产者差生了2条消息：M1、M2，要保证这两条消息的顺序，应该怎样做？可能脑中想到的是这样的：



假定M1-->S1，M2-->S2，如果要保证M1比M2想消费，那么M1到达消费端被消费后，通知S2，然后S2再将M2发送到消费端。

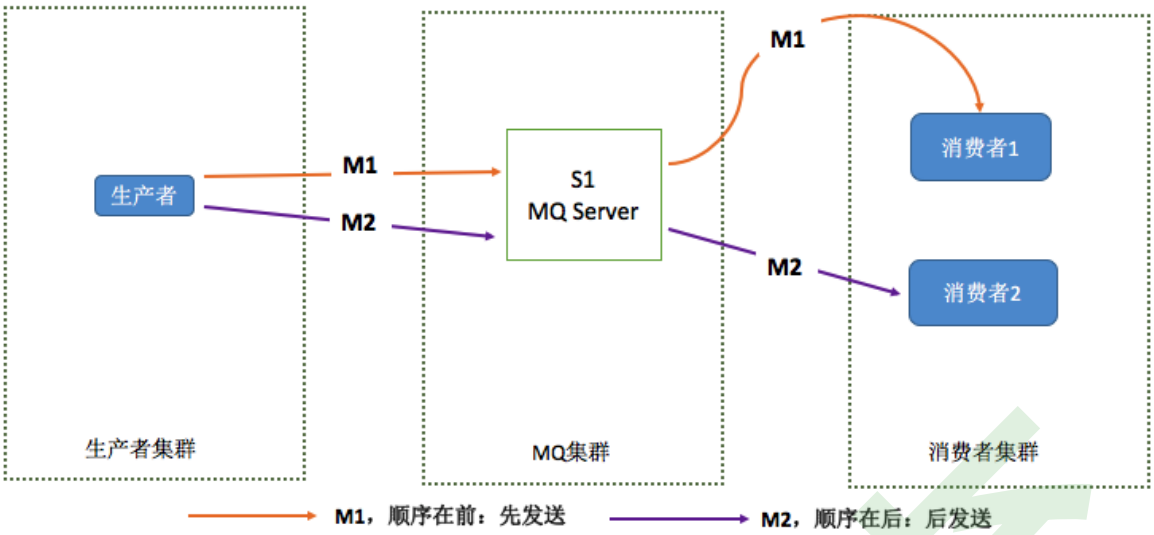
但是这个模型存在的问题是：如果M1和M2分别发送到两台Server上，就不能保证M1先到达MQ集群，也不能保证M1被先消费。换个角度看，如果M2先与M1到达MQ集群，甚至M2被消费后，M1才到达消费端，这时候消息就乱序了，说明以上模型是不能保证消息的顺序的。

如何才能在MQ集群保证消息的顺序？一种简单的方式就是将M1、M2发送到同一个Server上：



上面这样可以保证M1先与M2达到MQ集群（Producer等待M1发送成功后再发送M2），根据先到达先被消费的原则，M1会先于M2被消费，这样就保证了消息的顺序。

但是这个模型也仅仅是在理论上可以保证消息的顺序，在实际场景中可能会遇到下面的问题：网络延迟问题

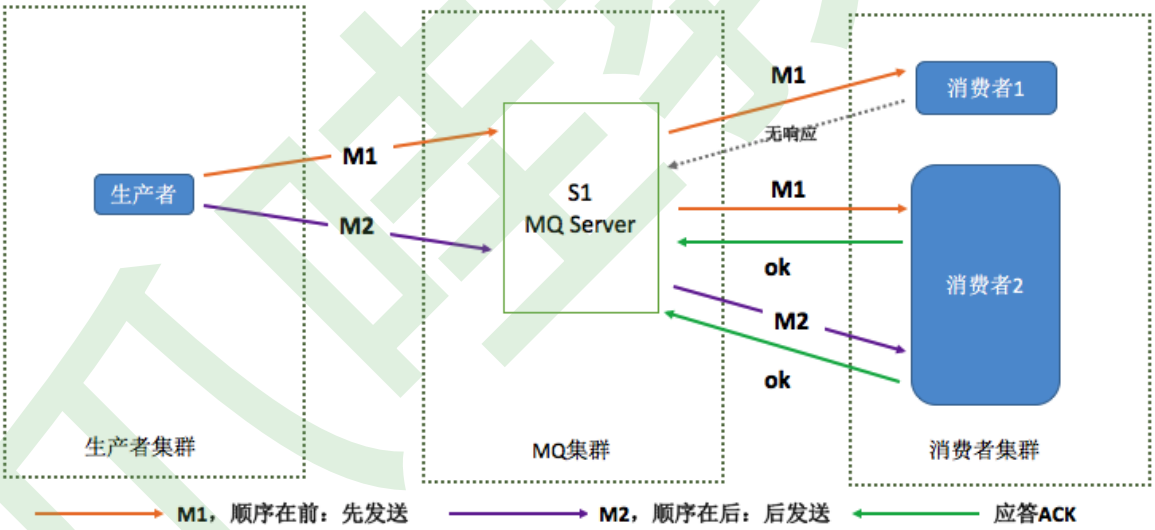


只要将消息从一台服务器发往另一台服务器，就会存在网络延迟问题，如上图所示，如果发送M1耗时大于发送M2耗时，那么仍然存在可能M2被先消费，仍然不能保证消息的顺序，即使M1和M2同时到达消费端，由于不清楚消费端1和消费端2的负载情况，仍然可能出现M2先于M1被消费的情况。

那如何解决这个问题呢?

将M1和M2发往同一个消费者，且发送M1后，需要消费端响应成功后才能发送M2。

但是这里又会存在另外的问题：如果M1被发送到消费端后，消费端1没有响应，那么是继续发送M2呢，还是重新发送M1？一般来说为了保证消息一定被消费，肯定会选择重发M1到另外一个消费端2，如下图，保证消息顺序的正确方式：



这样的模型就可以严格保证消息的顺序，但是仍然会有问题：消费端1没有响应Server时，有两种情况，一种是M1确实没有到达（数据可能在网络传输中丢失），另一种是消费端已经消费M1并且已经发回响应消息，但是MQ Server没有收到。如果是第二种情况，会导致M1被**重复消费**。

回过头来看消息顺序消费问题，严格的顺序消息非常容易理解，也可以通过文中所描述的方式来简化处理，总结起来，要实现严格的顺序消息，简单可行的办法就是：

```
保证 生产者-MQServer-消费者  是“一对一对一”的关系。
```

这样的设计虽然简单易行，但是也存在一些很严重的问题，比如：

```
1.并行度会成为消息系统的瓶颈（吞吐量不够）
2.产生更多的异常处理。比如：只要消费端出现问题，就会导致整个处理流程阻塞，我们不得不花费更多的精力来解决阻塞的问题。
```

我们最终的目标是要集群的高容错性和高吞吐量，这似乎是一对不可调和的矛盾，那么阿里是如何解决的呢?

```
世界上解决一个计算机问题最简单的方法：“恰好”不需要解决它！—— 沈询
```

有些问题，看起来很重要，但实际上我们可以通过合理的设计将问题分解来规避，如果硬要把时间花在解决问题本身，实际上不仅效率低下，而且也是一种浪费。从这个角度来看消息的顺序问题，可以得出两个结论：

```
1.不关注乱序的应用大量存在
2.队列无序并不意味着消息无序
```

所以从业务层面来保证消息的顺序，而不仅仅是依赖于消息系统，是不是我们更应该寻求的一种合理的方式?

最后从源码角度分析RocketMQ怎么实现发送顺序消息。

RocketMQ通过轮询所有队列的方式来确定消息被发送到哪一个队列（负载均衡策略）。

比如下面的例子中，订单号相同的消息挥别先后发送到同一个队列中:

```
// RocketMQ通过MessageQueueSelector中实现的算法来确定消息发送到哪一个队列上
// RocketMQ默认提供了两种MessageQueueSelector实现：随机/Hash
// 当然你可以根据业务实现自己的MessageQueueSelector来决定消息按照何种策略发送到消息队列中
SendResult sendResult = producer.send(msg, new MessageQueueSelector() {
    @Override
    public MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg)
{
        Integer id = (Integer) arg;
        int index = id % mqs.size();
        return mqs.get(index);
    }
}, orderId);
```

在获取到路由信息以后，会根据MessageQueueSelector实现的算法来选择一个队列，同一个OrderId获取到的肯定是同一个队列。

```
private SendResult send()  {
    // 获取topic路由信息
    TopicPublishInfo topicPublishInfo =
this.tryToFindTopicPublishInfo(msg.getTopic());
    if (topicPublishInfo != null && topicPublishInfo.ok()) {
        MessageQueue mq = null;
        // 根据我们的算法，选择一个发送队列
        // 这里的arg = orderId
        mq = selector.select(topicPublishInfo.getMessageQueueList(), msg, arg);
        if (mq != null) {
            return this.sendKernelImpl(msg, mq, communicationMode, sendCallback,
timeout);
        }
    }
}
```

消息重复

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？

```
1.消费端处理消息的业务逻辑要保持幂等性。
2.保证每条数据都有唯一编号，且保证消息处理成功与去重表的日志同时出现。
```

第1条很好理解，只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。

第2条原理就是利用一张日志表来记录已经处理成功的消息的ID，如果新到的消息ID已经在日志表中，那么久不在处理这条消息。

第1条解决方案，很明显应该在消费端实现，不属于消息系统要实现的功能。

第2条可以由消息系统实现，也可以由业务端实现。正常情况下出现重复消息的概率其实很小，如果由消息系统来实现的话，肯定会对消息系统的吞吐量和高可用有影响，所以最好还是由业务端自己处理消息重复的问题，这也是RocketMQ不解决消息重复问题的原因。

RocketMQ不保证消息不重复，如果你的业务系统需要保证严格的不重复消息，需要你自己在业务端去重。

事务消息

RocketMQ除了支持普通消息，顺序消息，另外还支持事务消息。首先讨论一下什么是事务消息以及支持事务消息的必要性。我们以一个转账的场景为例来说明这个问题：Bob向Smith转账100元。

在单机环境下，执行事务的情况大概是下面这个样子：（单机环境下转账事务示意图）

操作	耗时	总耗时
锁定Bob账户	0.01ms	5.04ms
锁定Smith账户	0.01ms	
检查Bob账户是否有100块	1ms	
Bob账户减去100块	2ms	
Smith账户加上100块	2ms	
解锁Smith账户	0.01ms	
解锁Bob账户	0.01ms	

事物时间顺序

当用户增长到一定的程度，Bob和Smith各自的账户和余额信息不再同一台服务器上了，那么上面的流程就会变成这样：（集群环境下转账事务示意图）

操作	耗时	总耗时
锁定Bob账户	0.01ms	11.04ms
通过网络锁定Smith账户	2+0.01ms	
检查Bob账户是否有100块	1ms	
Bob账户减去100块	2ms	
通过网络Smith账户加上100块	2+2ms	
通过网络解锁Smith账户	2+0.01ms	
解锁Bob账户	0.01ms	

事物时间顺序

这时候会发现，同样的一个转账业务，在集群环境下，耗时会成倍地增长，这显然是不能接受的，那么如何来规避这个问题？

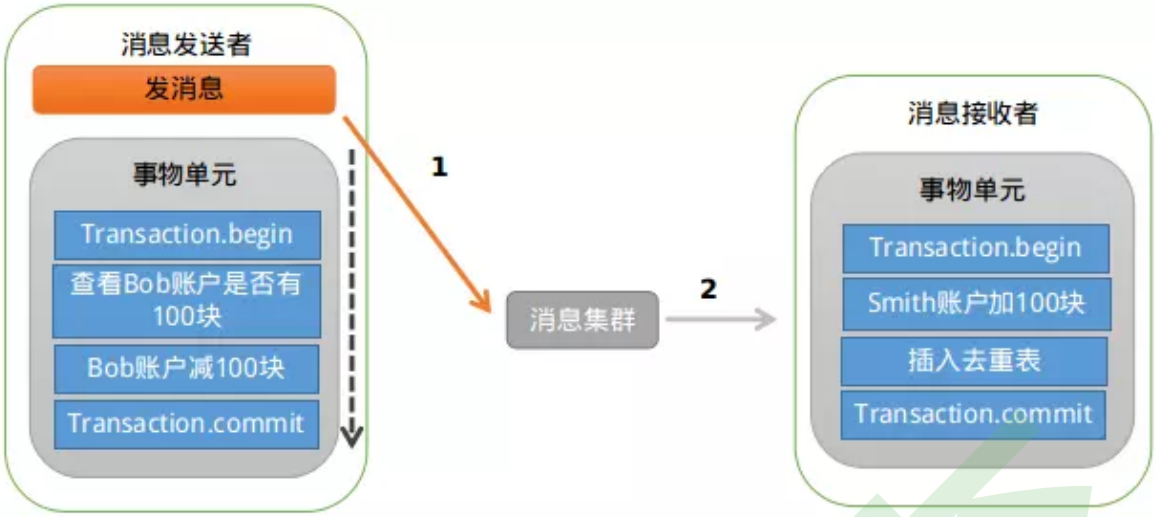
```
大事务 = 小事务 + 异步
```

将大事务拆分成多个小事务异步执行。这样基本上能够将跨机事务的执行效率优化到与单机一致。

转账的事务可以分解成如下两个小事务：（小事务+异步消息）

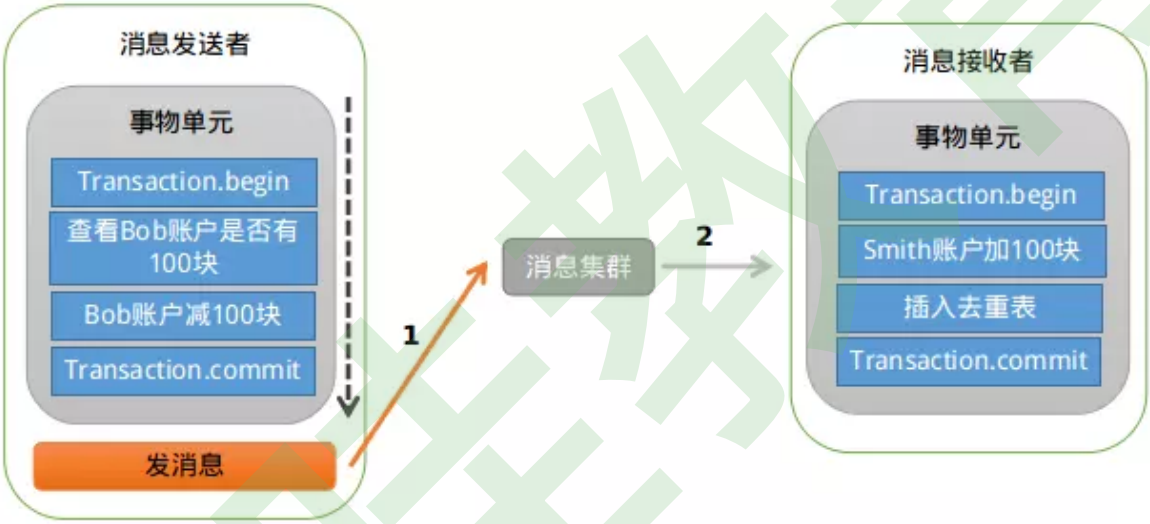
图中执行本地事务（Bob账户扣款）和发送异步消息应该保证同步成功或者同步失败，也就是扣款成功了，发送消息也一定要成功，如果扣款失败了，就不能发送消息。问题来了，我们是先扣款还是先发送消息呢？

首先看下线发送消息的情况，大致的示意图如下：（事务消息：先发送消息）



存在的问题是：如果消息发送成功，但是扣款失败，消费端就会消费此消息，进而向Smith账户加钱。

先发消息不行，那就先扣款吧，大致的示意图如下：（事务消息：先扣款）

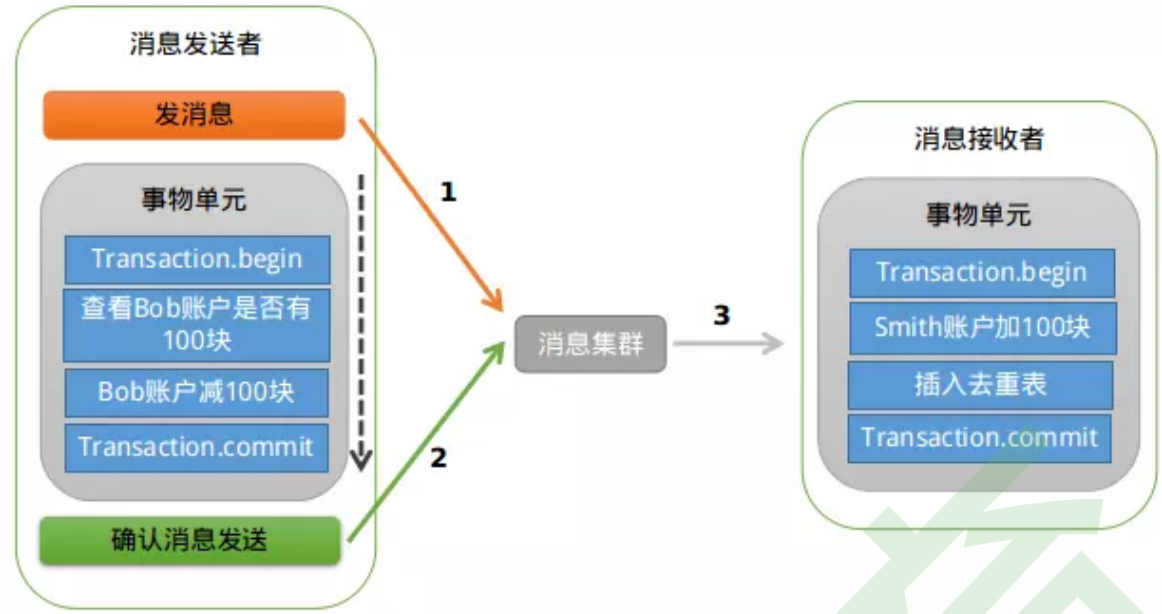


存在的问题和上面类似：如果扣款成功，发送消息失败，就会出现Bob扣钱了，但是Smith账户未加钱。

可能还会想到别的办法来解决这个问题：直接将发消息放到Bob扣款的扣款的事务中去，如果发送失败，就抛出异常，事务回滚。这样的处理方式也符合“恰好”不需要解决的原则。

> 这里需要声明一下：如果使用Spring来管理事务的话，大可以将发送消息的逻辑放到本地事务中去，发送消息失败就抛出异常，Spring捕捉到异常后就会回滚此事务，以此来保证本地事务与发送消息的原子性。

RocketMQ支持事务消息，下面来看看RocketMQ是怎样来实现发送事务消息的：



RocketMQ分三个阶段：

第一阶段发送Prepared消息时，会拿到消息的地址。

第二阶段执行本地事务。

第三阶段通过第一阶段拿到的地址去访问消息，并修改消息的状态。

这里又会发现问题，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事务消息，如果发现了Prepared消息，它会向Producer发送端确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚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。

我们来看下RocketMQ的源码，是如何处理事务消息的。Producer发送事务消息的部分。（完整的代码请查看：rocketmq-example工程下的com.alibaba.rocketmq.example.transaction.TransactionProducer）

```
// ============================发送事务消息的一系列准备工作
=====================================
// 未决事务，MQ服务器回查客户端
// 也就是上文所说的，当RocketMQ发现`Prepared消息`时，会根据这个Listener实现的策略来决断事务
TransactionCheckListener transactionCheckListener = new
TransactionCheckListenerImpl();// 构造事务消息的生产者
TransactionMQProducer producer = new TransactionMQProducer("groupName");
// 设置事务决断处理类
producer.setTransactionCheckListener(transactionCheckListener);
// 本地事务的处理逻辑，相当于示例中检查Bob账户并扣钱的逻辑
TransactionExecuterImpl tranExecuter = new TransactionExecuterImpl();
producer.start()
// 构造MSG，省略构造参数
Message msg = new Message(......);
// 发送消息
SendResult sendResult = producer.sendMessageInTransaction(msg, tranExecuter,
null);
producer.shutdown();
```

接着查看**sendMessageInTransaction**方法的源码，总共分为三个阶段：

1.发送Prepared消息

2.执行本地事务

3.发送确认消息

```
//  ================================事务消息的发送过程
==========================================
public TransactionSendResult sendMessageInTransaction(.....)  {
    // 逻辑代码，非实际代码
    // 1.发送消息
    sendResult = this.send(msg);
    // sendResult.getSendStatus() == SEND_OK
    // 2.如果消息发送成功，处理与消息关联的本地事务单元
if(sendResult.getStatus==SEND_OK)
    LocalTransactionState localTransactionState =
tranExecuter.executeLocalTransactionBranch(msg, arg);
    // 3.结束事务
    this.endTransaction(sendResult, localTransactionState, localException);
}
```

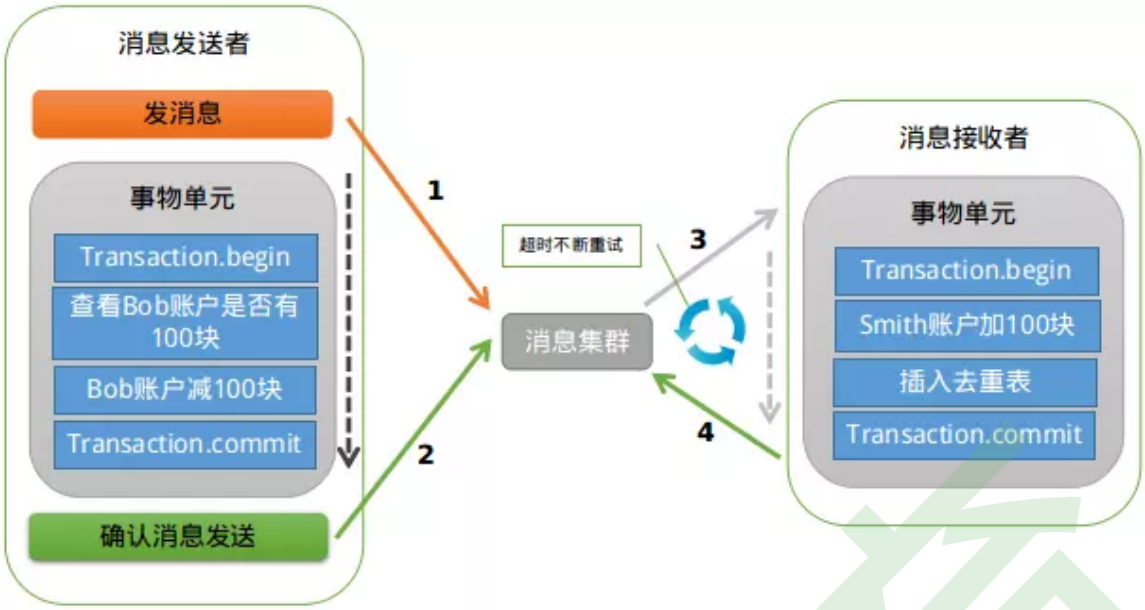
endTransaction()方法会将请求发往broker(mq server)去更新事务消息的最终状态，如下：

1.根据sendResult找到Prepared消息，sendResult包含事务消息的ID。

2.根据location更新消息的最终状态。

如果**endTransaction()**方法执行失败，数据没有发送发到broker,导致事务消息的状态更新失败，broker会有回查线程定时（默认1分钟）扫描每个事务状态的表格文件，如果已经提交或者回滚消息则直接跳过，如果是prepared状态的则会向Producer发起checkTransaction请求，Producer会调用DefaultMQProducerImpl.checkTransactionState()方法来处理broker的定时回调请求，而checkTransactionState会调用我们的事务设置的决断方法来决定是回滚事务还是继续执行，最后调用endTransactionOneway让broker来更新消息的最终状态。

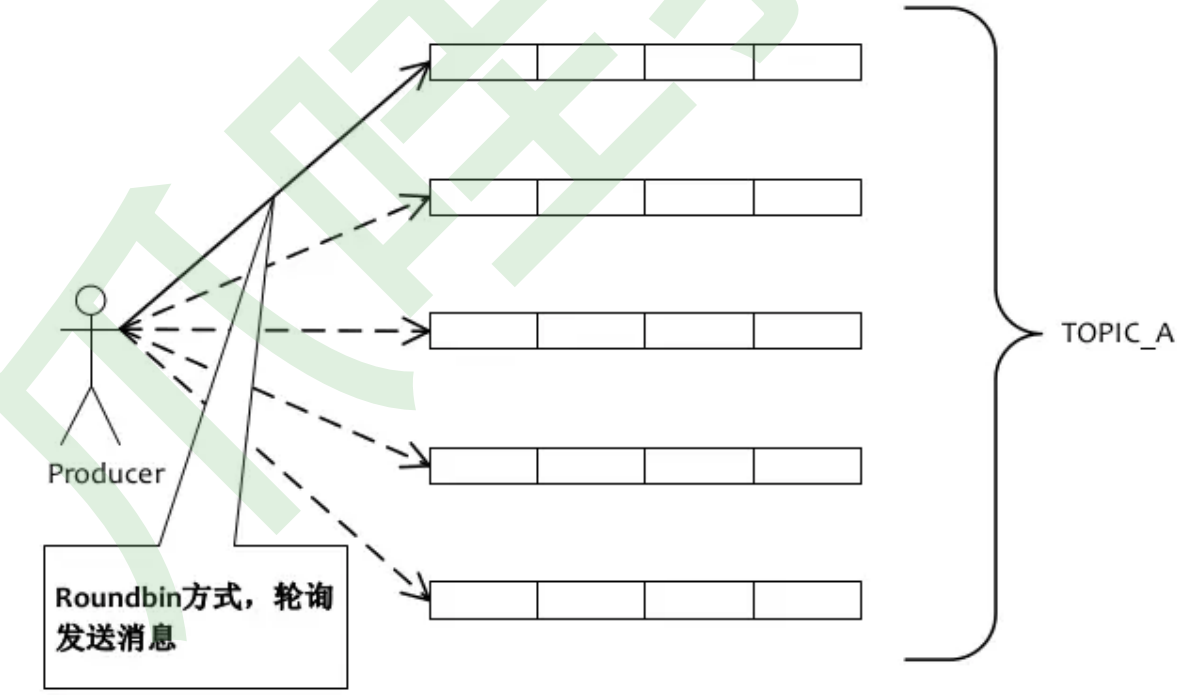
再回到转账的例子，如果Bob的账户余额已经减少，且消息发送成功，Smith端开始消费这条消息，这个时候回出现消费失败和消费超时两个问题，解决超时问题的思路就是一直重试，直到消费端消费消息成功，真个过程可能会有重复消费的问题，按照上面的思路解决即可。



这样基本上可以解决消费端超时的问题，但是消费失败了怎么办？爱丽缇提供的解决方法是：人工解决。我们可以考虑一下，按照事务的流程，因为某种原因Smith加款失败，那么需要回滚整个流程。如果消息系统要实现这个回滚流程的话，系统的复杂度将大大提升，且很容易出现bug，估计出现Bug的概率会比消费失败的概率大很多，这也是RocketMQ目前暂时没有解决这个问题的原因。在设计实现消息系统时，我们需要衡量是否花这么大的代价来解决一个出现概率非常小的问题。

Producer如何发送消息

Producer轮询某Topic下所有队列的方式来实现发送方的负载均衡，如下图所示：



RocketMQ的客户端发送消息的源码：

```
// 构造Producer
DefaultMQProducer producer = new DefaultMQProducer("ProducerGroupName");
// 初始化Producer，整个应用生命周期内，只需要初始化1次
producer.start();
// 构造Message
Message msg = new Message("TopicTest1",// topic
                          "TagA",         // tag：给消息打标签,用于区分一类消息，可为null
                          "OrderID188",  // key：自定义Key，可以用于去重，可为null
                          ("Hello MetaQ").getBytes());// body：消息内容
// 发送消息并返回结果
SendResult sendResult = producer.send(msg);
// 清理资源，关闭网络连接，注销自己
producer.shutdown();
```

在整个生命周期内，生产者需要调用一次start方法来初始化，初始化主要完成的任务有：

```
1.如果没有指定namesrv地址，将会自动寻址
2.启动定时任务：更新namesrv地址、从namesrv更新Topic路由信息、清理已经挂掉的broker、向所有的
broker发送心跳...
3.启动负载均衡的服务
```

初始化完成后，开始发送消息，发送消息的主要代码如下：

```
private SendResult sendDefaultImpl(Message msg,......) {
    // 检查Producer的状态是否是RUNNING
    this.makeSureStateOK();
    // 检查msg是否合法：是否为null、topic,body是否为空、body是否超长
    Validators.checkMessage(msg, this.defaultMQProducer);
    // 获取topic路由信息
    TopicPublishInfo topicPublishInfo =
this.tryToFindTopicPublishInfo(msg.getTopic());
    // 从路由信息中选择一个消息队列
    MessageQueue mq = topicPublishInfo.selectOneMessageQueue(lastBrokerName);
    // 将消息发送到该队列上去
    sendResult = this.sendKernelImpl(msg, mq, communicationMode, sendCallback,
timeout);
}
```

代码中需要关注两个方法**tryToFindTopicPublishInfo**和**selectOneMessageQueue**。

前面说过在Producer初始化时，会启动定时任务获取路由信息并更新到本地缓存，所以**tryToFindTopicPublishInfo**会首先从缓存中获取Topic路由信息，如果没有获取到，则会自己去namesrv获取路由信息，**selectOneMessageQueue**方法通过轮询的方式，返回一个队列，以达到负载的目的。

如果Producer发送消息失败，会自动重试，重试的策略：

1.重试次数<retryTimesWhenSendFailed（可配置）

2.总的耗时（包含重试n次的耗时）<sendMsgTimeout（发送消息传入的参数）

3.同时满足上面两个条件后，Producer会选择另外一个队列发送消息。

消息存储

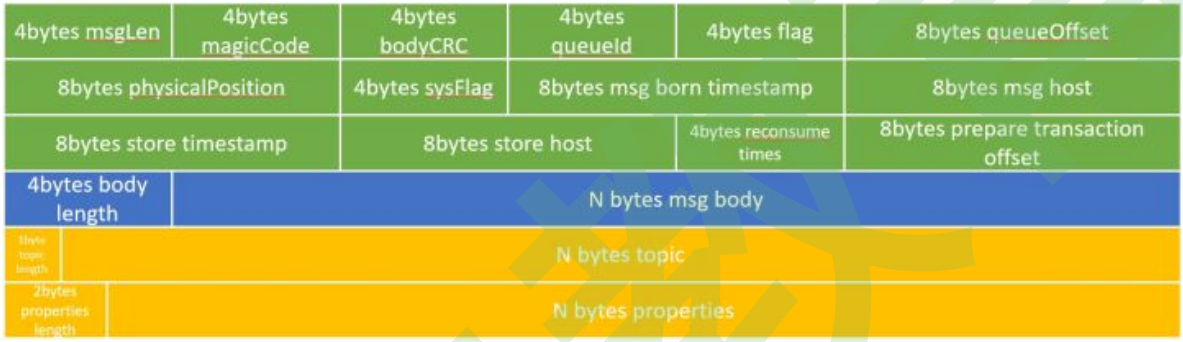
RocketMQ的消息存储是由comsume queue 和 cimmit log配合完成的。

consume queue是消息的逻辑队列，相当于字典的目录，用来指定消息在物理文件commit log上的位置。

CommitLog

要想知道RocketMQ如何存储消息，我们先看看CommitLog。在RocketMQ中，所有topic的消息都存储在一个称为CommitLog的文件中，该文件默认最大为1GB，超过1GB后会轮到下一个CommitLog文件。通过CommitLog，RocketMQ将所有消息存储在一起，以顺序IO的方式写入磁盘，充分利用了磁盘顺序写减少了IO争用提高数据存储的性能，

消息在CommitLog中的存储格式如下：



- 4字节表示消息的长度，消息的长度是整个消息体所占用的字节数的大小
- 4字节的魔数，是固定值，有MESSAGE_MAGIC_CODE和BLANK_MAGIC_CODE
- 4字节的CRC，是消息体的校验码，用于防止网络、硬件等故障导致数据与发送时不一样带来的问题
- 4字节的queueId，表示消息发到了哪个MessageQueue(逻辑上相当于kakka的partition)
- 4字节的flag，flag是创建Message对象时由生产者通过构造器设定的flag值
- 8字节的queueOffset，表示在queue中的偏移量
- 8字节的physicalPosition，表示在存储文件中的偏移量
- 4字节sysFlag，是生产者相关的信息标识，具体生产逻辑可以看相关代码
- 8字节消息创建时间
- 8字节消息生产者的host
- 8字节消息存储时间
- 8字节消息存储的机器的host
- 4字节表示重复消费次数
- 8字节消息事务相关偏移量
- 4字节表示消息体的长度
- 消息休，不是固定长度，和前面的4字节的消息体长度值相等
- 1字节表示topic的长度，因此topc的长度最多不能超过127个字节，超过的话存储会出错（有前置校验）
- Topic，存储topic，因为topic不是固定长度，所以这里所占的字节是不固定的，和前一个表示topic长度的字节的值相等
- 2字节properties的长度，properties是创建消息时添加到消息中的，因此，添加在消息中的poperties不能太多太大，所有的properties的kv对在拼接成string后，所占的字节数不能超过2^15-1

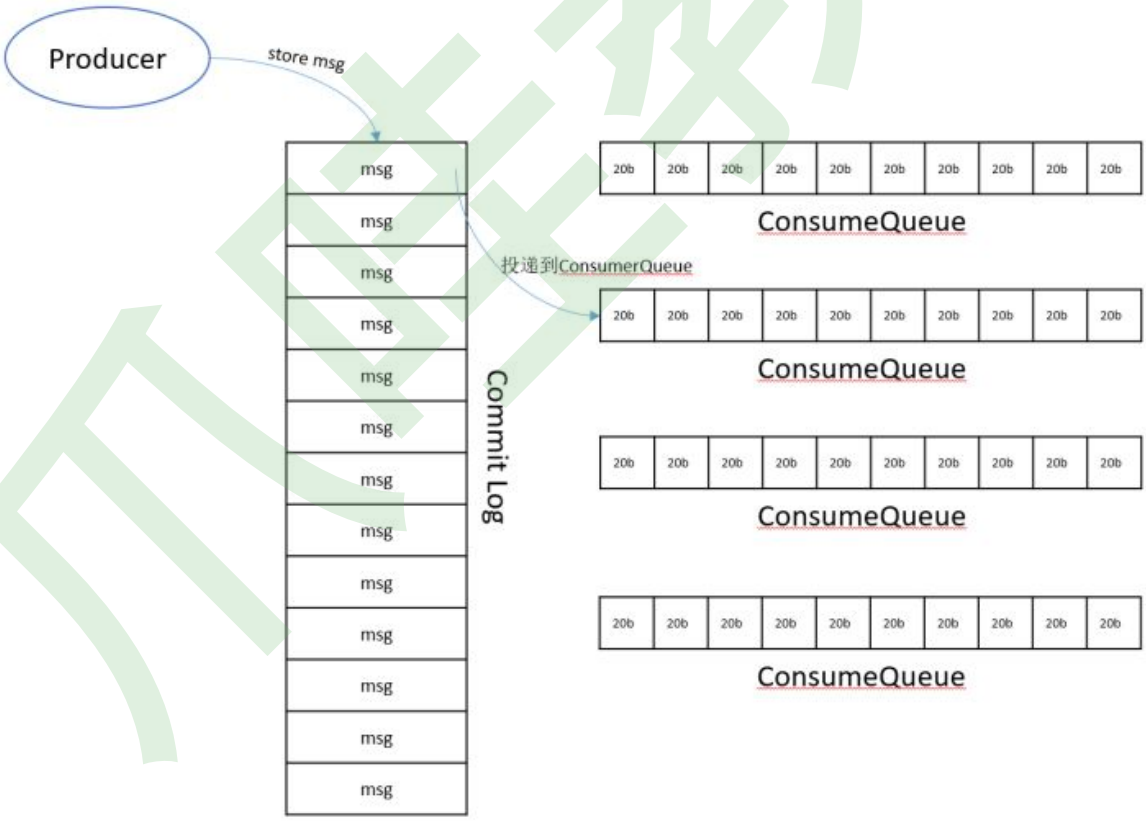
- Properties的内容，也不是固定长度，和前面的2字节properties长度的值相同

ConsumeQueue

一个ConsumeQueue表示一个topic的一个queue，类似于kafka的一个partition，但是rocketmq在消息存储上与kafka有着非常大的不同，RocketMQ的ConsumeQueue中不存储具体的消息，具体的消息由CommitLog存储，ConsumeQueue中只存储路由到该queue中的消息在CommitLog中的offset，消息的大小以及消息所属的tag的hash（tagCode），一共只占20个字节，整个数据包如下：

消息存储方式

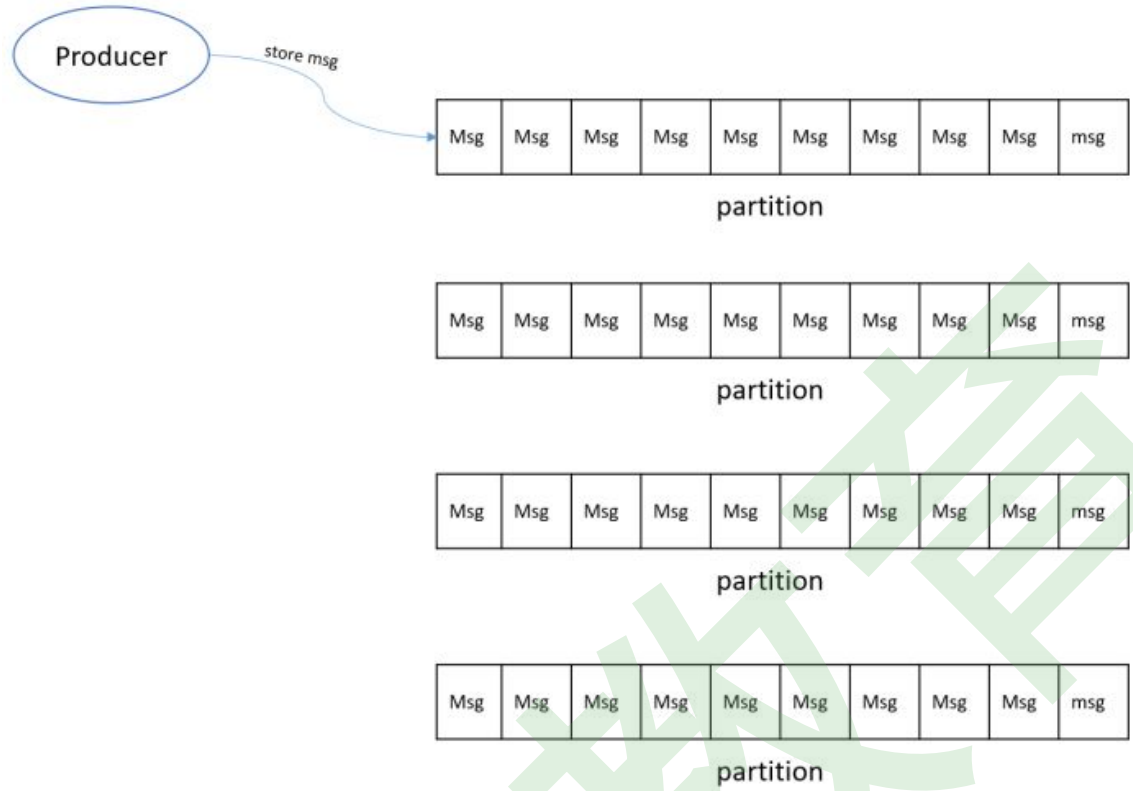
前文已经描述过，RocketMQ的消息存储由CommitLog和ConsumeQueue两部分组成，其中CommitLog用于存储原始的消息，而ConsumeQueue用于存储投递到某一个queue中的消息的位置信息，消息的存储如下图所示：



消费者在读取消息时，先读取ConsumeQueue，再通过ConsumeQueue中的位置信息读取CommitLog，得到原始的消息。

消息存储与kafka的对比

Kafka中，每个partition有独立的消息存储，投递到每个partition的消息，存储在partition自己的存储文件中，示意图如下：



在消息的存储上，RocketMQ与Kafka的主要区别在于，RocketMQ将所有消息存储在同一个CommitLog中且ConsumeQueue中每个消息只存储20个字节的消息位置信息，而Kafka将每个partition的消息分开存储，这导致RocketMQ单个broker能支持更多的topic和partition。

因为在RocketMQ中，所有消息都存储在同一个文件中，这使得RocketMQ的消息存储是磁盘的顺序写，而kafka将消息按partition存储在不同的文件中，因此kafka在消息存储上是随机IO，磁盘的顺序IO要比随机IO快得多，顺序IO可以接近内存的速度。将partition的数量非常大时，kafka中的随机IO将非常多，这将导致kafka在所有topic的partition变大了之后broker性能会明显下降。

但是RocketMQ的ConsumeQueue也是随机IO，为何相比kafka能支持更多的partition呢，原因是RocketMQ通过MappedFile的方式读写ConsumeQueue，操作系统对内存映射文件有page cache而ConsumeQueue中的数据都非常小（只有20bytes），读写几乎都是page cache的操作，因此虽然是随机IO但效率也非常高。

消息订阅

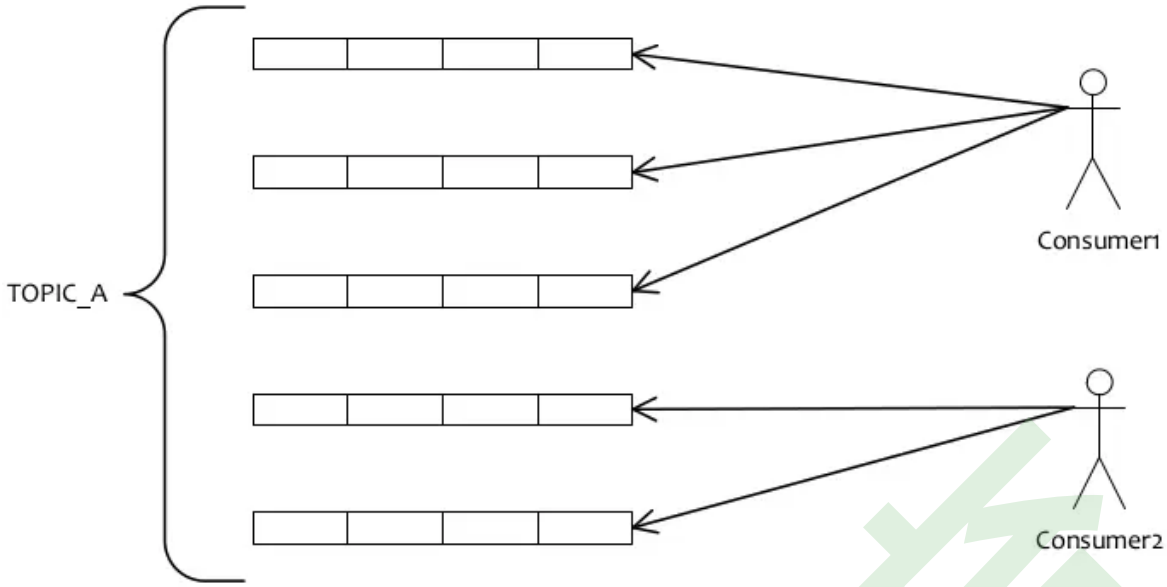
RocketMQ消息订阅有两种模式：

一种是push模式，即MQServer主动向消费端推送。

另一种是Pull模式，即消费端在需要时，主动到MQServer拉取。

但是再具体实现时，Push和Pull模式都是采用消费端主动拉取的方式。

首先看下消费端的负载均衡：



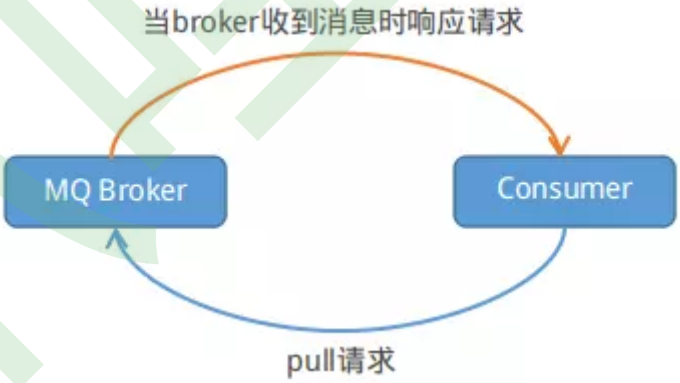
消费端会通过RebalanceService线程，10s做一次基于Topic下的所有队列负载：

```
1.遍历Consumer下所有的Topic，然后根据Topic订阅所有的消息
2.获取同一Topic和Consume Group下的所有Consumer
3.然后根据具体的分配策略来分配消费队列，分配的策略包含：平均分配、消费端配置等
```

如上图所示，如果有5个队列，2个Consumer，那么第一个Consumer消费3个队列，第二个Consumer消费2个队列。这里采用的就是平均分配策略。它类似于分页的过程，Topic下面所有的Queue就是记录，Consumer的个数就相当于总的页数，那么每页有多少条记录，就类似于Consumer会消费哪些队列。

通过这样的策略来达到大体上的平均消费，这样的设计也可以很方便地水平扩展来提高Consumer的消费能力。

消费端的Push模式是通过长轮询的模式来实现的，就如同下图：（Push模式示意图）



Consumer端每隔一段时间主动向broker发送拉消息请求，broker在收到Pull请求后，如果有消息就立即返回数据，Consumer端收到返回的消息后，再回调消费者设置的Listener方法。如果broker在收到Pull请求时，消息队列里没有数据，broker端会阻塞请求指导有数据传递或超时才返回。

当然，Consumer端是通过一个线程将阻塞队列LinkBlockingQueue中的PullRequest发送到broker拉取消息，以防止Consumer一直被阻塞。而Broker端，在接收到Consumer的PullRequest时，如果发现没有消息，就会把PullRequest扔到ConcurrentHashMap中缓存起来。broker在启动时，会启动一个线程不停地从ConcurrentHashMap取出PullRequest检查，直到有数据返回。

RocketMQ的其他特性

前面的6个特性都是基本上点到为止，想要深入了解，还需要多多查看源码，多多在实际中运用欧冠。除了上面提到的特性，RocketMQ还支持：

1.定时消息

2.消息的刷盘策略

3.主动同步策略：同步双写、异步复制

4.海量消息堆积能力

5.高效通信

……

其中涉及到的很多设计思路和解决方法都值得深入研究：

1.消息的存储设计：既要满足海量消息的堆积能力，又要满足极快的查询效率，还要保证写入的效率。

2.高效的通信组件设计：高吞吐量，毫秒级的消息投递能力都离不开高效的通信。

.....

Rocket的最佳实践

Producer最佳最佳实践

1.一个应用尽可能用一个Topic，消息子类型用tags来标识，tags可以由应用自由设置。只有发送消息设置了tags，消费方在订阅消息时，才可以利用tags在broker做消息过滤。

2.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。由于是哈希索引，请务必保证key尽可能唯一，这样可以避免潜在的哈希冲突。

3.消息发送成功或者失败，要打印消息日志，务必打印sendResult和key字段

4.对于消息不可丢失应用，务必要有消息重发机制。例如：消息发送失败，存储到数据库，能有定时程序尝试重发或者人工触发重发。

5.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最佳实践

1.消费过程要做到幂等

2.尽量使用批量方式消费，可以很大程度上提高消费吞吐量。

3.优化每条消息的消费过程

其他配置

线上应该关闭autoCreateTopicEnable，即在配置文件中将其设置为false。

RocketMQ在发送消息时，会首先获取路由信息。如果是新的消息，由于MQServer上面还没有创建对应的Topic，这个时候，如果上面的配置打开的话（autoCreateTopicEnable=true），会返回默认Topic的路由信息（RocketMQ会在每台Broker上面创建名为TBW102的Topic），然后Producer会选择一台Broker发送消息，选中的Broker在存储消息时，发现消息的Topic还没有创建，就会自动创建Topic。后果就是：以后所有该Topic的消息，都将发送到这台Broker上，达不到负载均衡的目的。

所以基于目前RocketMQ的设计，建议关闭自动创建Topic的功能，然后根据消息量的大小，手动创建Topic。

RocketMQ设计相关

RocketMQ的设计假定：

```
每台PC机器都可能宕机不可服务
任意集群都有可能处理能力不足
最坏的情况一定会发生
内网需要低延迟来提供最佳用户体验
```

RocketMQ的关键设计

```
分布式集群化
强数据安全
海量数据堆积
毫秒级投递延迟（推拉模式）
```

这是RocketMQ在设计时的假定前提以及需要到达的效果。我想这些假定适用于所有的系统设计。随着我们系统的服务的增多，每位开发者都要注意自己的程序是否存在单点故障，如果挂了应该怎么恢复、能不能很好的水平扩展、对外的接口是否足够高效、自己管理的数据是否足够安全..... 多多规范自己的设计，才能开发出高效健壮的程序。