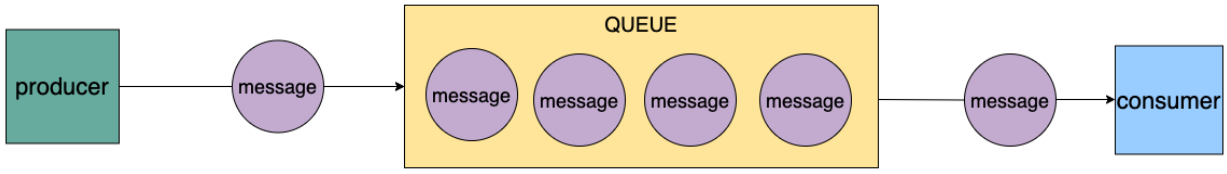
RocketMQ 基本理论及架构

1 MQ介绍

> 说明：

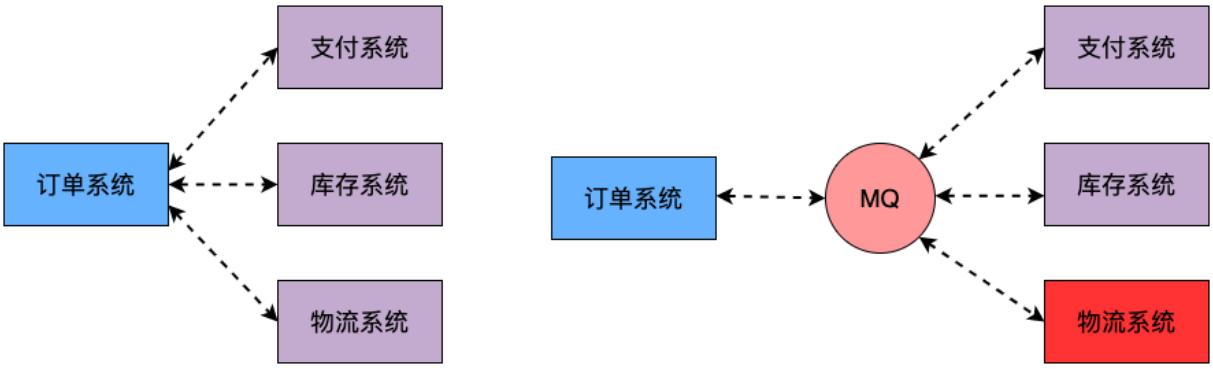
1.1 为什么要用MQ

MQ（Message Queue）消息队列，是基础数据结构中“先进先出”的一种数据结构。在消息的传输过程中保存消息的容器，生产者和消费者不直接通讯，依靠队列保证消息的可靠性，避免了系统间的相互影响。

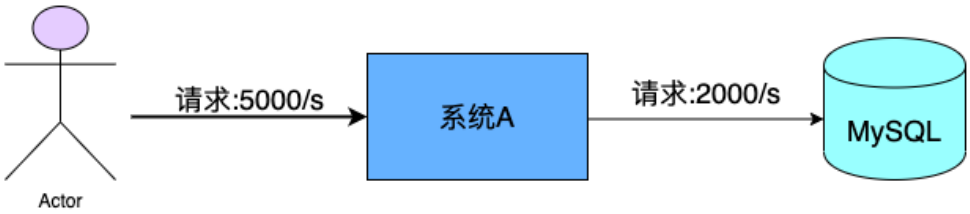


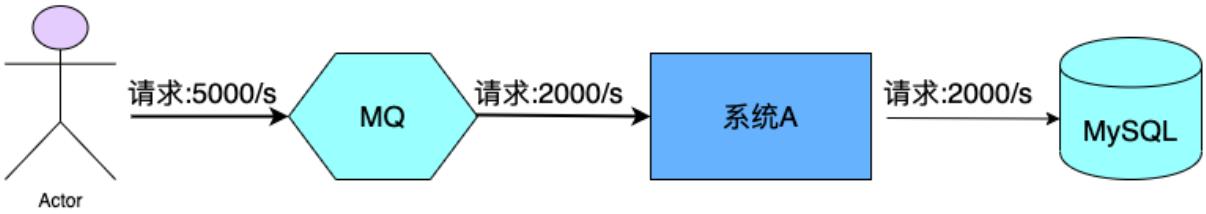
其应用场景主要包含以下：

1.1.1 异步解耦



1.1.2 削峰填谷





1.1.3 其他

- 顺序收发
- 分布式事务一致性
- 大数据分析
- 分布式缓存同步

1.2 MQ的缺点

- 不能完全代替RPC
- 系统可用性降低
- 系统的复杂度提高
- 一致性问题

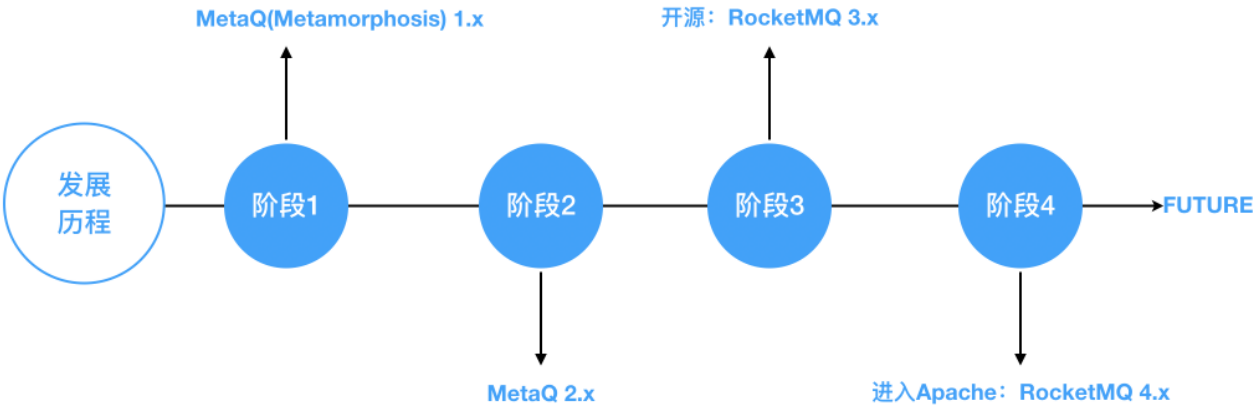
1.3 RocketMQ的前世今生

RocketMQ是一款阿里巴巴开源的消息中间件，在2017年9月份成为Apache的顶级项目，是国内首个互联网中间件在 Apache 上的顶级项目。

RocketMQ的起源受到另一款消息中间件Kafka的启发。最初，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和提升写入性能，需要在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，被Kafka无限消息堆积，高效的持久化速度所吸引。

不过当时Kafka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，例如：延迟消息，消费重试，事务消息，消息过滤等，这些都是一些企业级消息中间件需要具备的功能。为此，淘宝中间件团队重新用Java语言编写了RocketMQ，**定位于非日志的可靠消息传输**。不过随着RocketMQ的演进，现在也支持了日志流式处理。

目前RocketMQ经过了阿里多年双十一大促的考验，性能和稳定性得到了充分的验证。目前在业界被广泛应用在订单，交易，充值，流计算，消息推送，日志流式处理，binlog分发等场景。



阶段1：Metaq（Metamorphosis） 1.x

由开源社区 killme2008 维护，开源社区地址：https://github.com/killme2008/Metamorphosis，最后一次更新是在2017年1月份。

阶段2：Metaq 2.x

于 2012 年 10 月份上线，在淘宝内部被广泛使用。

阶段3：RocketMQ 3.x

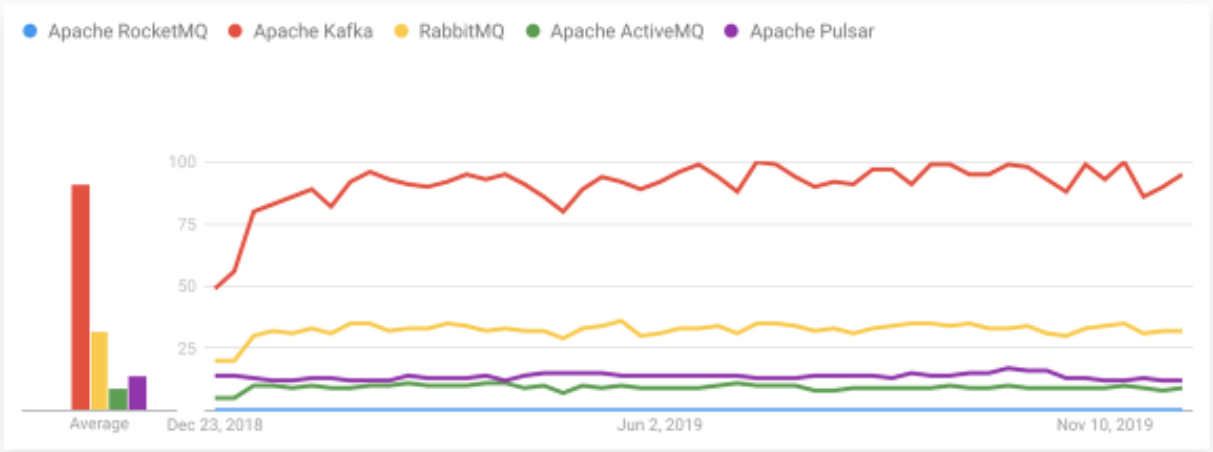
基于阿里内部开源共建原则， RocketMQ 项目只维护核心功能，且去除了所有其他运行时依赖，核心功能最简化。每个 BU 的个性化需求都在 RocketMQ 项目之上进行深度定制。 RocketMQ 向其他 BU 提供的仅仅是Jar 包，例如要定制一个 Broker，那么只需要依赖 rocketmq-broker 这个 jar 包即可，可通过 API 迕行交互，如果定制 client，则依赖 rocketmq-client 这个 jar 包，对其提供的 api 进行再封装。

阶段4：进入Apache

2016年11月28日，阿里巴巴向 Apache 软件基金会捐赠消息中间件 RocketMQ，成为 Apache 孵化项目。美国时间 2017 年 9 月 25 日，Apache 软件基金会（ASF）宣布 Apache®RocketMQ™ 已孵化成为 Apache 顶级项目（TLP ），是国内首个互联网中间件在 Apache 上的顶级项目。官网地址：http://rocketmq.apache.org/

1.4 各种MQ产品的比较

目前业界还有很多其他MQ，如Kafka、RabbitMQ、ActiveMQ、Apache Pulsar等。下图列出了全球范围内这些MQ在2018.12~2019.12一年时间内，在Google Trends的搜索频率，某种程度可以反映出这些中间件的火爆程度。



从这张图上，我们可以看出来，Kafka是一枝独秀，RabbitMQ紧接其后，ActiveMQ和Apache Pulsar也有一定的占比。而RocketMQ的搜索量可以说是微乎其微。

对于除了RocketMQ的其他几个MQ产品，可以根据这张图初步对比下流行程度。但是对于RocketMQ必须排除在外，因为一些原因，很多国内的用户无法通过Google进行搜索，因此关于RocketMQ的统计实际上是不准确的。

而在这里，主要对比的是RocketMQ与其他MQ有哪些功能特性上的差异。功能特性，主要取决于产品定位，如Kafka定位于高吞吐的流失日志和实时计算场景；ActiveMQ、RabbitMQ等则定位于企业级消息中间件，因此提供了很多企业开发时非常有用的功能，如延迟消息、事务消息、消息重试、消息过滤等，而这些特性Kafka都不具备，但是这类产品的吞吐量要明显的低于Kafka。

RocketMQ则是结合了Kafka和ActiveMQ、RabbitMQ的特性。在性能上，可以与Kafka抗衡；而在企业级MQ的特性上，则具备了很多ActiveMQ、RabbitMQ提供的特性。因此，企业在选择消息中间件选型时，RocketMQ是非常值得考虑的一款产品。

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级，吞吐量比RocketMQ和Kafka要低一个数量级	万级，吞吐量比RocketMQ和Kafka要低一个数量级	十万级，RocketMQ也是可以支撑高吞吐的一种MQ	十万级别，Kafka最大优点就是吞吐量大，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
Topic数量对吞吐量的影响	-	-	Topic可以达到几百、几千个的级别，吞吐量会有小幅度的下降。这是RocketMQ的一大优势，可在同等数量机器下支撑大量	Topic从几十个到几百个的时候，吞吐量会大幅下降。所以在同等机器数量下，Kafka尽量保证Topic数量不要过多。如果支撑大规模Topic需要增加更多的

			的Topic	机器
时效性	延迟在ms(毫秒)以内	微秒级，这是rabbitmq的一大特点，延迟是最低的	延迟在ms(毫秒)以内	延迟在ms(毫秒)以内
可用性	高，基于主从架构实现可用性	高，基于主从架构实现可用性	非常高，分布式架构	非常高，Kafka是分布式的，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据	-	经过参数优化配置，可以做到零丢失	经过参数配置，消息可以做到零丢失
功能支持	MQ领域的功能及其完备	基于erlang开发，所以并发性能极强，性能极好，延时低	MQ功能较为完备，分布式扩展性好	功能较为简单，主要支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广
优势	非常成熟，功能强大，在业内大量公司和项目中都有应用	erlang语言开发，性能极好、延时很低，吞吐量万级、MQ功能完备，管理界面非常好，社区活跃；互联网公司使用较多	接口简单易用，阿里出品有保障，吞吐量大，分布式扩展方便、社区比较活跃，支持大规模的Topic、支持复杂的业务场景，可以基于源码进行定制开发	超高吞吐量，ms级的时延，极高的可用性和可靠性，分布式扩展方便
劣势	偶尔有较低概率丢失消息，社区活跃度不高	吞吐量较低，erlang语音开发不容易进行定制开发，集群动态扩展麻烦	接口不是按照标准JMS规范走的，有的系统迁移要修改大量的代码，技术有被抛弃的风险	有可能进行消息的重复消费
应用	主要用于解耦和异步，较少用在大规模吞吐的场景中	都有使用	用于大规模吞吐、复杂业务中	在大数据的实时计算和日志采集中被大规模使用，是业界的标准
客户端语言		java、c++、python	java	java、c++、go、python

功能对比

对比项	Kafka	RocketMQ	RabbitMQ
顺序消息	支持	支持	支持
延时消息	不支持	只支持特定Level	不支持
事务消息	不支持	支持	不支持
消息过滤	支持	支持	不支持
消息查询	不支持	支持	不支持
消费失败重试	不支持	支持	支持
批量发送	支持	支持	不支持

总结

Kafka:系统间的数据流通道

RocketMQ:高性能可靠消息传输

RabbitMQ:可靠性传输

2 快速入门

官网地址:http://rocketmq.apache.org/

其主要功能有

1.灵活可扩展性

2.海量消息堆积能力

3.支持顺序消息

4.多种消息过滤方式

5.支持事务消息 等常用功能。

2.1 准备工作

最新版本: 4.9.0

当前版本: 4.8.0

2.2 安装RocketMQ

```
> unzip rocketmq-all-4.8.0-source-release.zip
 > cd rocketmq-all-4.8.0/
 > mvn -Prelease-all -DskipTests clean install -U
 > cd distribution/target/rocketmq-4.8.0/rocketmq-4.8.0
```

或者下载

https://mirrors.bfsu.edu.cn/apache/rocketmq/4.8.0/rocketmq-all-4.8.0-bin-release.zip

目录介绍

bin: 启动脚本，包含shell脚本和CMD脚本

conf:实时配置文件，包括broker配置文件，logback配置文件等

lib:依赖jar包，包含Netty、commons-lang、FastJSON等

2.3 启动RocketMQ

Start Name Server

```
 > nohup sh bin/mqnamesrv &
 > tail -f ~/logs/rocketmqlogs/namesrv.log
 The Name Server boot success...
```

Start Broker

```
 > nohup sh bin/mqbroker -n localhost:9876 &
 > tail -f ~/logs/rocketmqlogs/broker.log
 The broker[%s, 172.30.30.233:10911] boot success...
```

2.4 测试RocketMQ

```
> export NAMESRV_ADDR=localhost:9876
> sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
SendResult [sendStatus=SEND_OK, msgId= ...

> sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
ConsumeMessageThread_%d Receive New Messages: [MessageExt...
```

2.5 关闭RocketMQ

```
> sh bin/mqshutdown broker
The mqbroker(36695) is running...
Send shutdown request to mqbroker(36695) OK

> sh bin/mqshutdown namesrv
The mqnamesrv(36664) is running...
Send shutdown request to mqnamesrv(36664) OK
```

3 消息样例

```
<dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-client</artifactId>
        <version>4.8.0</version>
    </dependency>
```

3.1普通消息

3.1.1消息发送

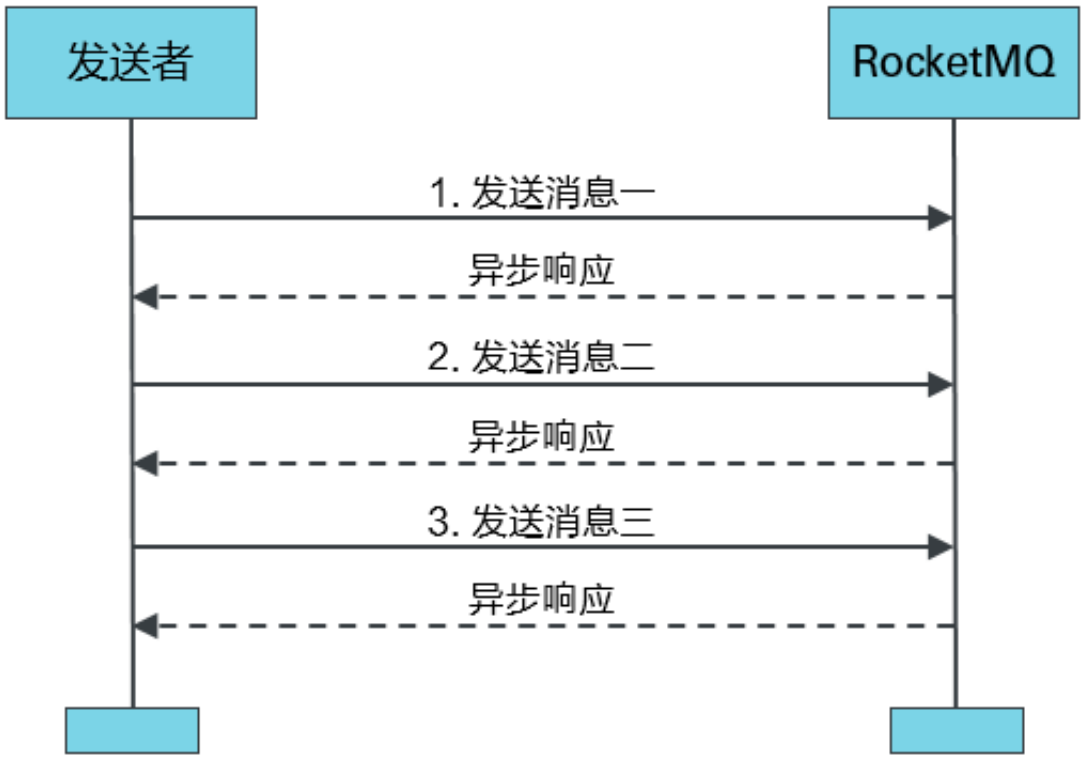
1)发送同步消息

应用场景:

此种方式应用场景非常广泛，例如重要通知邮件、报名短信通知、营销短信系统等。

```
public class SyncProducer {
    .....
}
```

2)发送异步消息



应用场景:

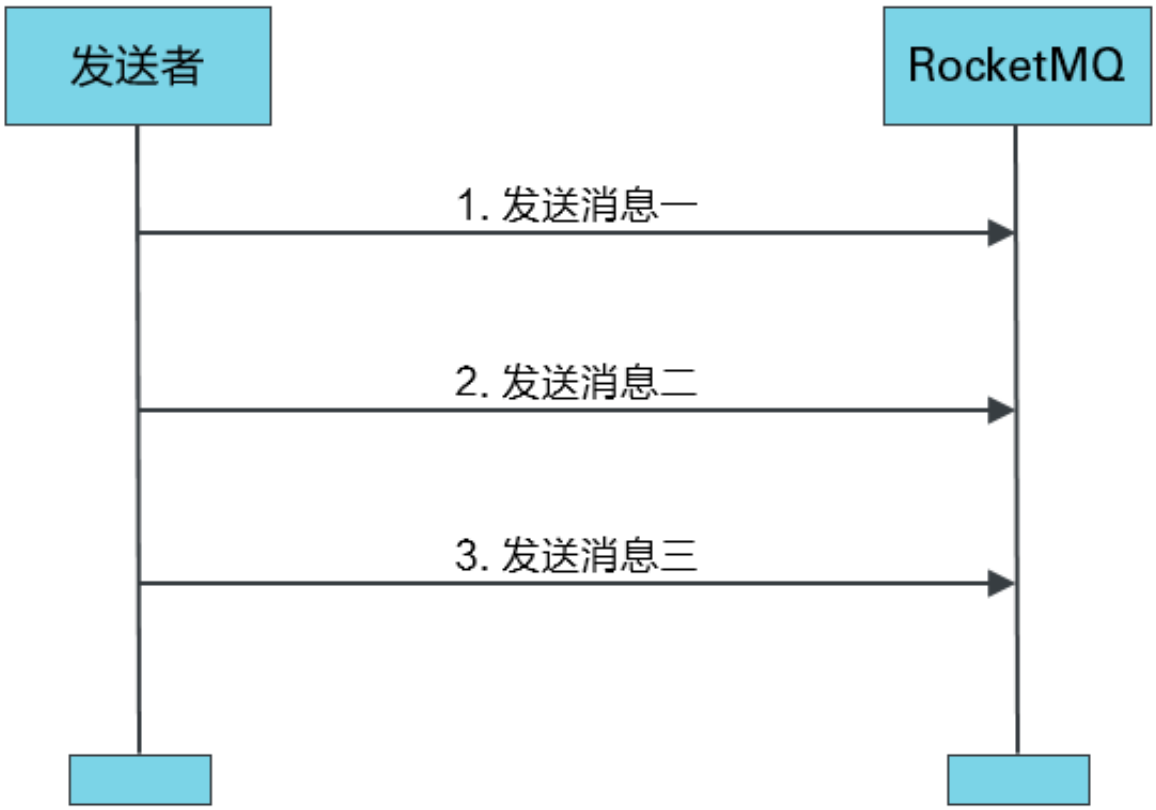
异步发送一般用于链路耗时较长，对响应时间较为敏感的业务场景，例如用户视频上传后通知启动转码服务，转码完成后通知推送转码结果等。

```
public class AsyncProducer {
    ......
}
```

```
 /**
     * It will be removed at 4.4.0 cause for exception handling and the wrong
Semantics of timeout. A new one will be
     * provided in next version
     *
     * @param msg
     * @param sendCallback
     * @param timeout the <code>sendCallback</code> will be invoked at most
time
     * @throws RejectedExecutionException
     */
```

- 超时时间的计算不准确:在run方法里进行时间判断（if (timeout > costTime)）实际上已经是开始执行当前线程的时间，而之前的排队时间没有算
- 这个异步方法还有提升空间，可以直接结合Netty、不必用到Executor
- 异常处理应该统一，而不应该是有的异常抛出，而有的异常通过回调方法返回客户端。

3)单向发送消息



应用场景:

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。

```
public class OnewayProducer {
    .....
}
```

三种发送方式的对比

性能和可靠性 平衡

发送方式	发送 TPS	发送结果反馈	可靠性
同步发送	快	有	不丢失
异步发送	快	有	不丢失
单向发送	最快	无	可能丢失

3.1.2消费消息

```
public class Consumer {

    .....
}
```

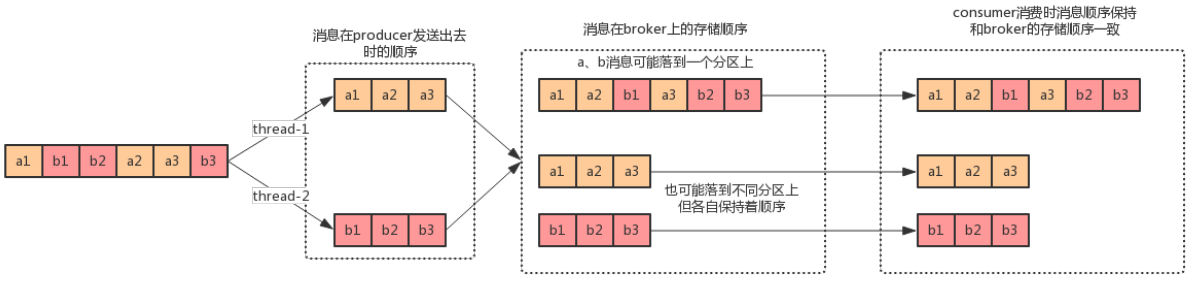
3.2 顺序消息

顺序消息（FIFO 消息）是消息队列提供的一种**严格按照顺序来发布和消费的消息**。

如何保证顺序

在MQ的模型中，顺序需要由3个阶段去保障：

1. 消息被发送时保持顺序，单线程发送
2. 消息被存储时保持和发送的顺序一致，队列的选择
3. 消息被消费时保持和存储的顺序一致，单线程接收

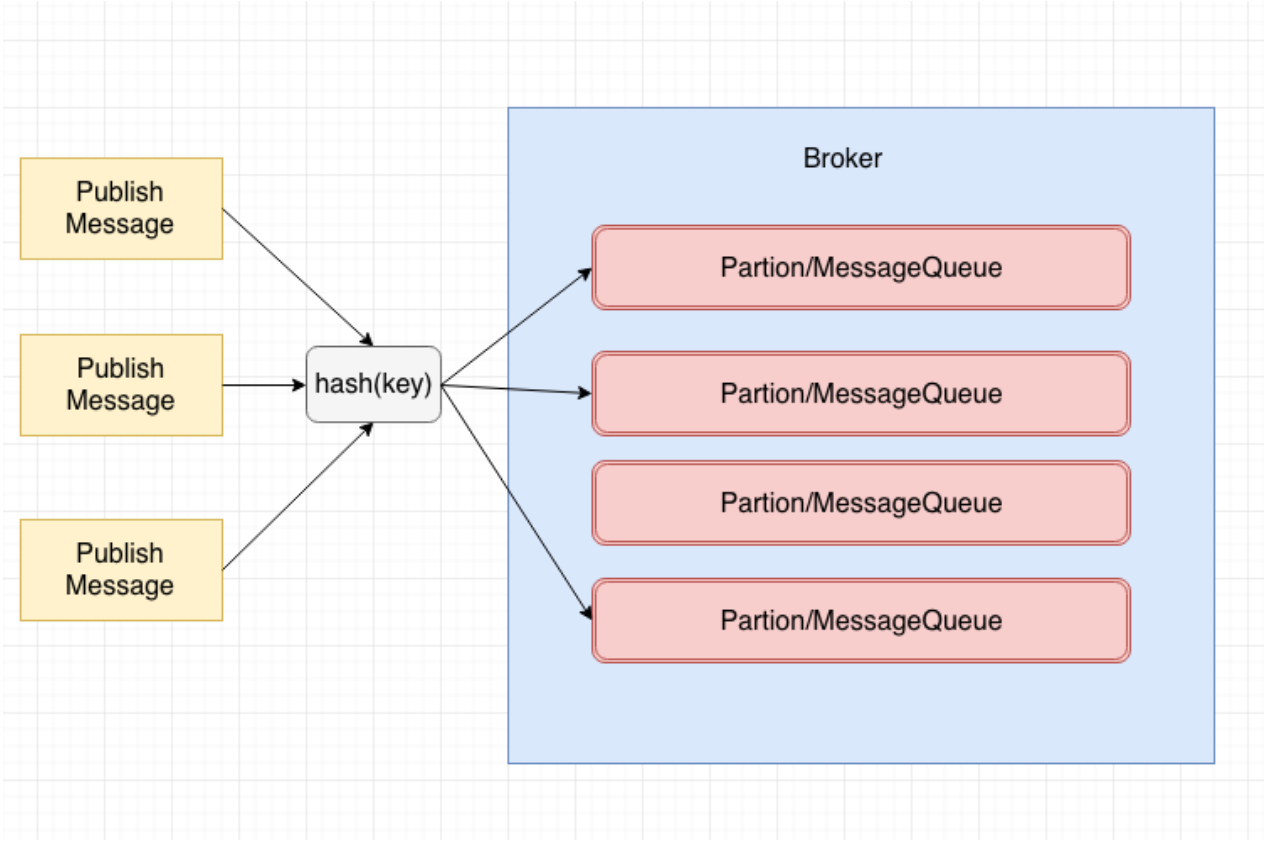


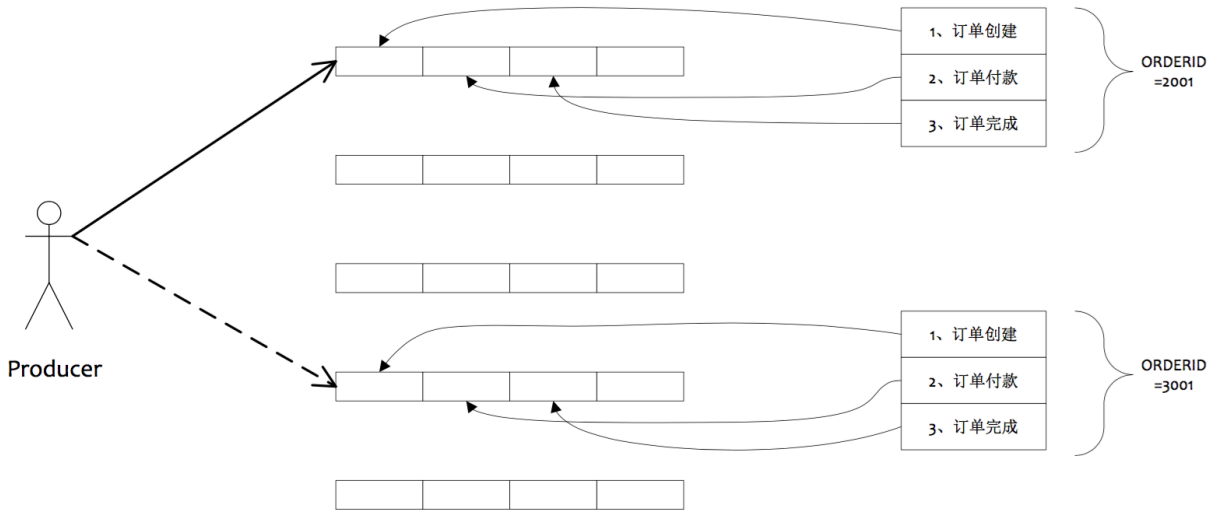
对于两个订单的消息的原始数据：a1、b1、b2、a2、a3、b3（绝对时间下发生的顺序）：

- a1、b1、b2、a2、a3、b3是可以接受的
- a1、a2、b1、b2、a3、b3也是可以接受的
- a1、a3、b1、b2、a2、b3是不能接受的

顺序的实现

顺序消息分为**全局顺序消息和分区顺序消息**。





消息生产示例

```
public class OrderedProducer {
    .....
}
```

消息消费

```
public class OrderedConsumer {
    .....
}
```

MessageListenerOrderly与MessageListenerConcurrently区别

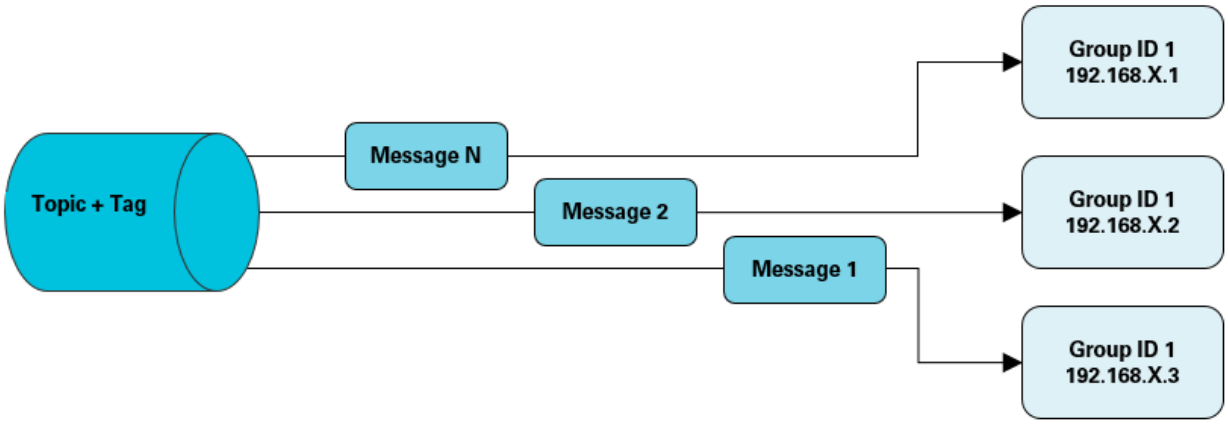
MessageListenerOrderly:有序消费，同一队列的消息同一时刻只能一个线程消费，可保证消息在同一队列严格有序消费

MessageListenerConcurrently:并发消费

3.3 广播消息

集群消费模式

适用于消费端集群化部署，每条消息只需要被处理一次的场景。此外，由于消费进度在服务端维护，可靠性更高。

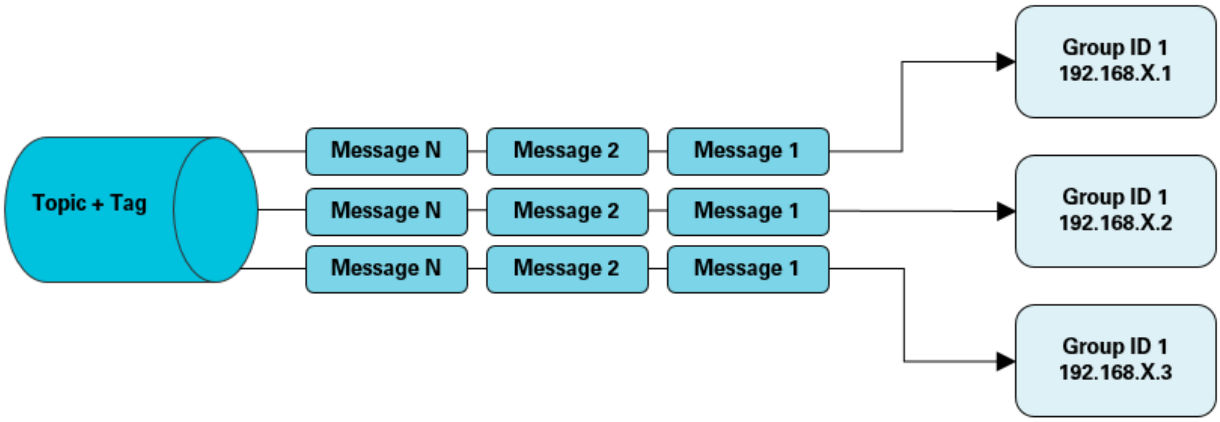


注意事项

- 集群消费模式下，每一条消息都只会被分发到一台机器上处理。如果需要被集群下的每一台机器都处理，请使用广播模式。
- 集群消费模式下，不保证每一次失败重投的消息路由到同一台机器上。

广播消费模式

适用于消费端集群化部署，每条消息需要被集群下的每个消费者处理的场景。



注意事项

- 广播消费模式下不支持顺序消息。
- 广播消费模式下不支持重置消费位点。
- 每条消息都需要被相同订阅逻辑的多台机器处理。
- 广播模式下服务端不维护消费进度，消费进度在客户端维护
- 广播模式下，消息队列 RocketMQ 版保证每条消息至少被每台客户端消费一次，但是并不会重投消费失败的消息，因此业务方需要关注消费失败的情况。
- 广播模式下，客户端每一次重启都会从最新消息消费。客户端在被停止期间发送至服务端的消息将会被自动跳过，请谨慎选择。
- 广播模式下，每条消息都会被大量的客户端重复处理，因此推荐尽可能使用集群模式。

发送消息

```
public class BroadcastProducer {
    .....
}
```

消费消息

```
public class BroadcastConsumer {
    ......
}
```

3.4 延时消息

3.4.1 介绍

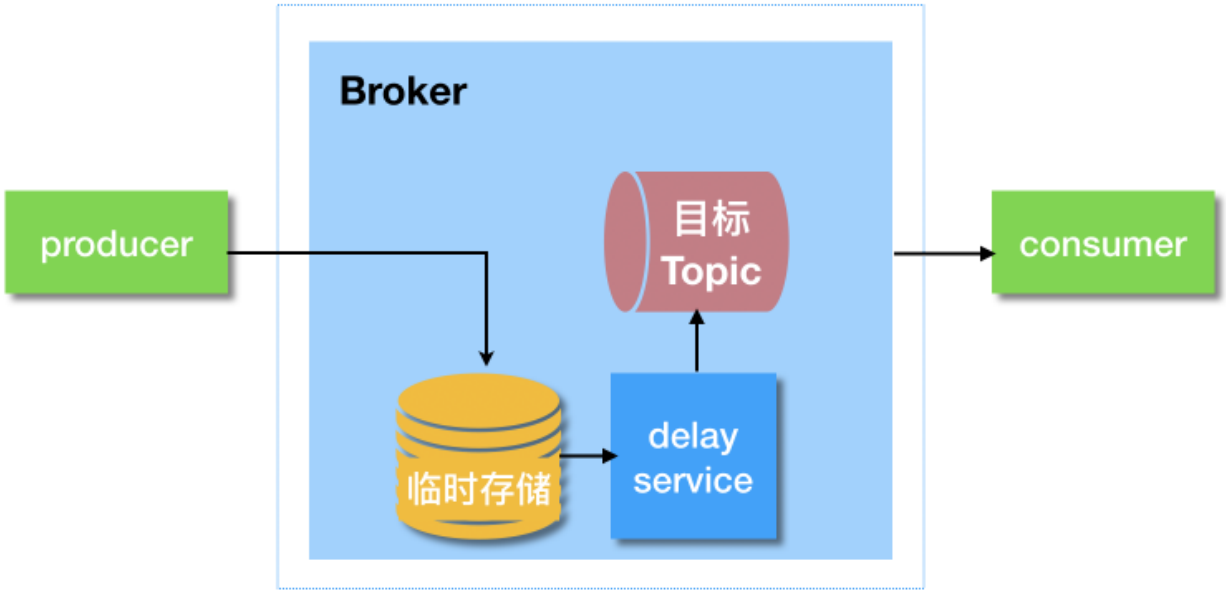
实现这类需求通常有两种方式：

- 轮询定时任务

- 延时消息

> 1) Sleep
>
> 不靠谱，工作线程sleep导致普通任务也sleep
>
> 2) 定时扫描
>
> 增大数据库压力(扫描频率高)
>
> 不精准(扫描频率低)

一些消息中间件的Broker端内置了延迟消息支持的能力，核心实现思路都是一样：**将延迟消息通过一个临时存储进行暂存，到期后才投递到目标Topic中**。



显然，临时存储模块和延迟服务模块，是延迟消息实现的关键。上图中，临时存储和延迟服务都是在Broker内部实现，对业务透明。

第三方存储选型要求：

对于第三方临时存储，其需要满足以下几个特点：

高性能：

写入延迟要低，在选择临时存储时，写入性能必须要高，关系型数据库通常不满足需求。

高可靠：

延迟消息写入后，不能丢失，需要进行持久化，并进行备份

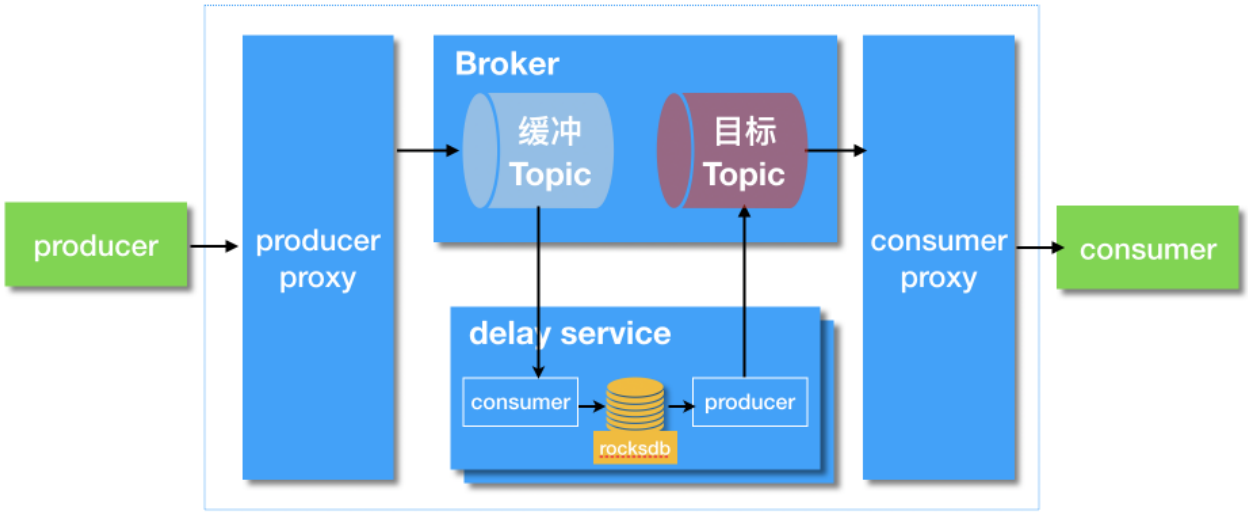
支持排序：

支持按照某个字段对消息进行排序，对于延迟消息需要按照时间进行排序。普通消息通常先发送的会被先消费，延迟消息与普通消息不同，需要进行排序。例如先发一条延迟10s的消息，再发一条延迟5s的消息，那么后发送的消息需要被先消费。

支持长时间保存

3.4.2 DDMQ

滴滴开源的消息中间件DDMQ，底层消息中间件的基础上加了一层代理，独立部署延迟服务模块，使用rocksdb进行临时存储。rocksdb是一个高性能的KV存储，并支持排序。



说明如下：

```
1 生产者将发送给producer proxy，proxy判断是延迟消息，将其投递到一个缓冲Topic中；

2 delay service启动消费者，用于从缓冲topic中消费延迟消息，以时间为key，存储到rocksdb中；

3 delay service判断消息到期后，将其投递到目标Topic中。

4 消费者消费目标topic中的数据
```

这种方式的好处是，因为delay service的延迟投递能力是独立于broker实现的，不需要对broker做任何改造，对于任意MQ类型都可以提供支持延迟消息的能力，例如DDMQ对RocketMQ、Kafka都提供了秒级精度的延迟消息投递能力，RocketMQ虽然支持延迟消息，但不支持秒级精度。

3.4.3 示例

RocketMQ不支持任意时间的延时，支持一定数量

```
String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m
30m 1h 2h";
```

你想支持1天的延迟，修改最后一个level的值为1d，这个时候依然是18个level；也可以增加一个1d，这个时候总共就有19个level。

```
messageDelayLevel=1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h
1d
```

```
public class ScheduledMessageConsumer {

    ......
}
```

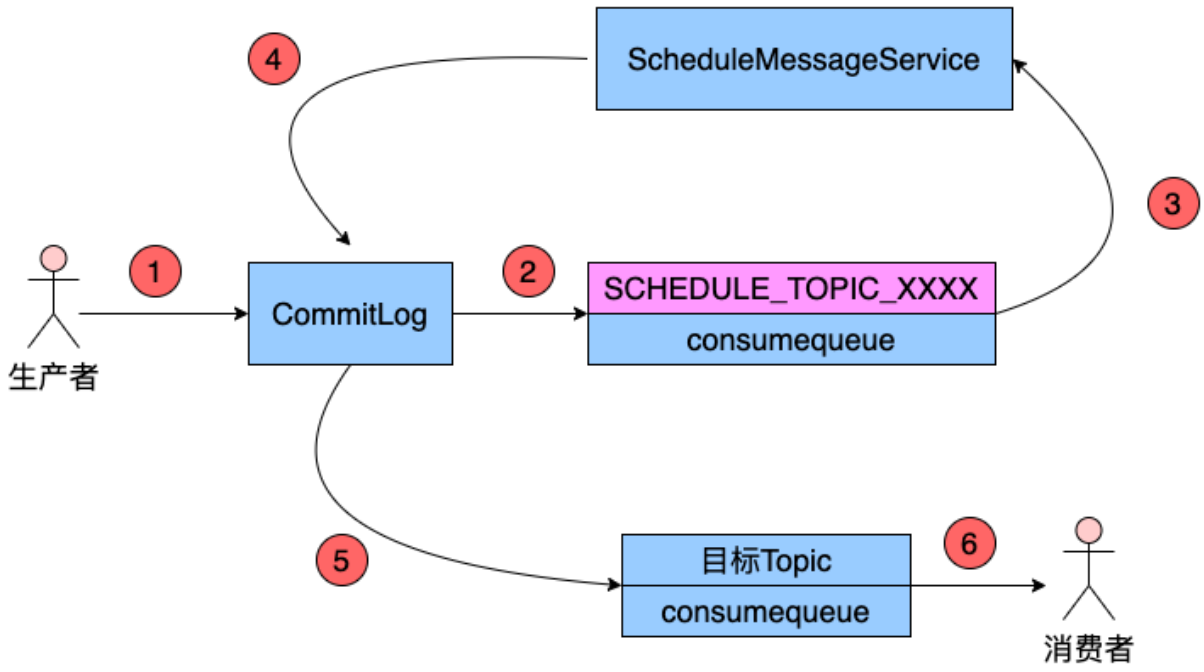
```
public class ScheduledMessageProducer {

    .....

}
```

3.4.4 原理

延迟队列的核心思路是：所有的延迟消息由producer发出之后，都会存放到同一个topic（**SCHEDULE_TOPIC_XXXX**）下，根据延迟level的个数，创建对应数量的队列，也就是说18个level对应了18个队列。



1. 修改消息Topic名称和队列信息
2. 转发消息到延迟主题SCHEDULE_TOPIC_XXXX的CosumeQueue中
3. 延迟服务消费SCHEDULE_TOPIC_XXXX消息
4. 将信息重新存储到CommitLog中

5. 将消息投递到目标Topic中
6. 消费者消费目标topic中的数据

第一步：修改消息Topic名称和队列信息

org.apache.rocketmq.store.CommitLog#putMessage

```
public PutMessageResult putMessage(final MessageExtBrokerInner msg) {
...
    // 如果是延迟消息
    if (msg.getDelayTimeLevel() > 0) {
        //如果设置的级别超过了最大级别，重置延迟级别
        if (msg.getDelayTimeLevel() >
this.defaultMessageStore.getScheduleMessageService()

.getMaxDelayLevel()) {

 msg.setDelayTimeLevel(this.defaultMessageStore.getScheduleMessageService()

.getMaxDelayLevel());
        }

        //修改Topic的投递目标为内部主题SCHEDULE_TOPIC_XXXX
        topic = ScheduleMessageService.SCHEDULE_TOPIC;
        //根据delayLevel，确定将消息投递到SCHEDULE_TOPIC_XXXX内部的哪个队列中
        queueId =
ScheduleMessageService.delayLevel2QueueId(msg.getDelayTimeLevel());

        // 记录原始topic，queueId
        MessageAccessor.putProperty(msg, MessageConst.PROPERTY_REAL_TOPIC,
msg.getTopic());
        MessageAccessor.putProperty(msg, MessageConst.PROPERTY_REAL_QUEUE_ID,
                                    String.valueOf(msg.getQueueId()));

 msg.setPropertiesString(MessageDecoder.messageProperties2String(msg.getProper
ties()));

        //更新消息投递目标为SCHEDULE_TOPIC_XXXX和queueId
        msg.setTopic(topic);
        msg.setQueueId(queueId);
    }
...
```

第二步：转发消息到延迟主题的CosumeQueue中

在转发过程中，会对延迟消息进行特殊处理，主要是计算这条延迟消息需要在什么时候进行投递。

```
投递时间=消息存储时间(storeTimestamp) + 延迟级别对应的时间
```

- Commit Log Offset：记录在CommitLog中的位置。
- Size：记录消息的大小
- Message Tag HashCode：记录消息Tag的哈希值，用于消息过滤。**特别的，对于延迟消息，这个字段记录的是消息的投递时间戳。**这也是为什么java中hashCode方法返回一个int型，只占用4个字节，而这里Message Tag HashCode字段确设计成8个字节的原因。

CommitLog#checkMessageAndReturnSize

```
public DispatchRequest checkMessageAndReturnSize(java.nio.ByteBuffer
byteBuffer, final boolean checkCRC,
    final boolean readBody) {
  ...
  // Timing message processing
  {
      //如果消息需要投递到延迟主题SCHEDULE_TOPIC_XXX中
      String t = propertiesMap.get(MessageConst.PROPERTY_DELAY_TIME_LEVEL);
      if (TopicValidator.RMQ_SYS_SCHEDULE_TOPIC.equals(topic) && t != null) {
          int delayLevel = Integer.parseInt(t);

          if (delayLevel >
this.defaultMessageStore.getScheduleMessageService().getMaxDelayLevel()) {
              delayLevel =
this.defaultMessageStore.getScheduleMessageService().getMaxDelayLevel();
          }
          //如果延迟级别大于0，计算目标投递时间，并将其当做tag哈希值
          if (delayLevel > 0) {
              tagsCode = this.defaultMessageStore.getScheduleMessageService()

.computeDeliverTimestamp(delayLevel,storeTimestamp);
          }
      }
  }
  ...
```

第三步：延迟服务消费SCHEDULE_TOPIC_XXXX消息

Broker内部有一个ScheduleMessageService类，其充当延迟服务，消费SCHEDULE_TOPIC_XXXX中的消息，并投递到目标Topic中。

ScheduleMessageService在启动时，其会创建一个定时器Timer，并根据延迟级别的个数，启动对应数量的TimerTask，每个TimerTask负责一个延迟级别的消费与投递。

ScheduleMessageService#start

```
public void start() {
        if (started.compareAndSet(false, true)) {
                //1 创建定时器Timer
            this.timer = new Timer("ScheduleMessageTimerThread", true);
            //2 针对每个延迟级别，创建一个TimerTask
        //2.1 迭代每个延迟级别：delayLevelTable是一个Map记录了每个延迟级别对应的延迟时间
            for (Map.Entry<Integer, Long> entry :
this.delayLevelTable.entrySet()) {
                        //2.2 获得每个每个延迟级别的level和对应的延迟时间
                Integer level = entry.getKey();
                Long timeDelay = entry.getValue();
                Long offset = this.offsetTable.get(level);
                if (null == offset) {
                    offset = 0L;
                }
//2.3 针对每个级别创建一个对应的TimerTask
                if (timeDelay != null) {
                    this.timer.schedule(new
DeliverDelayedMessageTimerTask(level, offset), FIRST_DELAY_TIME);
                }
            }
```

需要注意的是，每个TimeTask在检查消息是否到期时，首先检查对应队列中尚未投递第一条消息，如果这条消息没到期，那么之后的消息都不会检查。如果到期了，则进行投递，并检查之后的消息是否到期。

第四步：将信息重新存储到CommitLog中

由于之前Message Tag HashCode字段存储的是消息的投递时间，这里需要重新计算tag的哈希值后再存储。

DeliverDelayedMessageTimerTask#messageTimeup方法

```
private MessageExtBrokerInner messageTimeup(MessageExt msgExt) {
            MessageExtBrokerInner msgInner = new MessageExtBrokerInner();
            msgInner.setBody(msgExt.getBody());
            msgInner.setFlag(msgExt.getFlag());
            MessageAccessor.setProperties(msgInner, msgExt.getProperties());
```

```
            TopicFilterType topicFilterType =
MessageExt.parseTopicFilterType(msgInner.getSysFlag());
          //由于之前Message Tag HashCode字段存储的是消息的投递时间，这里需要重新计算tag
的哈希值后再存储。
            long tagsCodeValue =
                MessageExtBrokerInner.tagsString2tagsCode(topicFilterType,
msgInner.getTags());
            msgInner.setTagsCode(tagsCodeValue);

 msgInner.setPropertiesString(MessageDecoder.messageProperties2String(msgExt.g
etProperties()));

            msgInner.setSysFlag(msgExt.getSysFlag());
            msgInner.setBornTimestamp(msgExt.getBornTimestamp());
            msgInner.setBornHost(msgExt.getBornHost());
            msgInner.setStoreHost(msgExt.getStoreHost());
            msgInner.setReconsumeTimes(msgExt.getReconsumeTimes());

            msgInner.setWaitStoreMsgOK(false);
            MessageAccessor.clearProperty(msgInner,
MessageConst.PROPERTY_DELAY_TIME_LEVEL);
//设置真正的topic

 msgInner.setTopic(msgInner.getProperty(MessageConst.PROPERTY_REAL_TOPIC));

            String queueIdStr =
msgInner.getProperty(MessageConst.PROPERTY_REAL_QUEUE_ID);
            int queueId = Integer.parseInt(queueIdStr);
            msgInner.setQueueId(queueId);

            return msgInner;
        }
```

第五步：将消息投递到目标Topic中

第六步：消费者消费目标topic中的数据

延迟消息存放

```
consumequeue
├── SCHEDULE_TOPIC_XXXX
│   ├── 0
│   │   └── 00000000000000000000
│   ├── 1
│   │   └── 00000000000000000000
│   ├── 2
│   │   └── 00000000000000000000
│   ├── 3
│   │   └── 00000000000000000000
│   ├── 4
│   │   └── 00000000000000000000
    .....
    .....
├── DelayTopicTest
│   ├── 0
│   │   └── 00000000000000000000
│   ├── 1
│   │   └── 00000000000000000000
│   ├── 2
│   │   └── 00000000000000000000
│   └── 3
│       └── 00000000000000000000
```

其中不同的延迟级别放在不同的队列序号下（**queueId=delayLevel-1**）。每一个延迟级别对应的延迟消息转换为普通消息的位置标识存放在~/store/config/delayOffset.json文件内。

key为对应的延迟级别，value对应不同延迟级别转换为普通消息的offset值。

```
{
        "offsetTable":{3:202,4:2,5:2,6:2,7:2,8:2,9:2,10:2,11:2
        }
```

3.5 批量消息

批量发送可以提高发送性能，但有一定的限制：

- topic 相同
- waitStoreMsgOK 相同，消息发送时是否等消息存储完成后再返回。（首先消息的iswaitstoremsgok=true(默认为true), 如果没有异常,我们将始终收到"OK", org.apache.rocketmq.common.message.Message#isWaitStoreMsgOK）
- 不支持延时发送
- 一批消息的大小不能大于 4M(DefaultMQProducer.maxMessageSize)

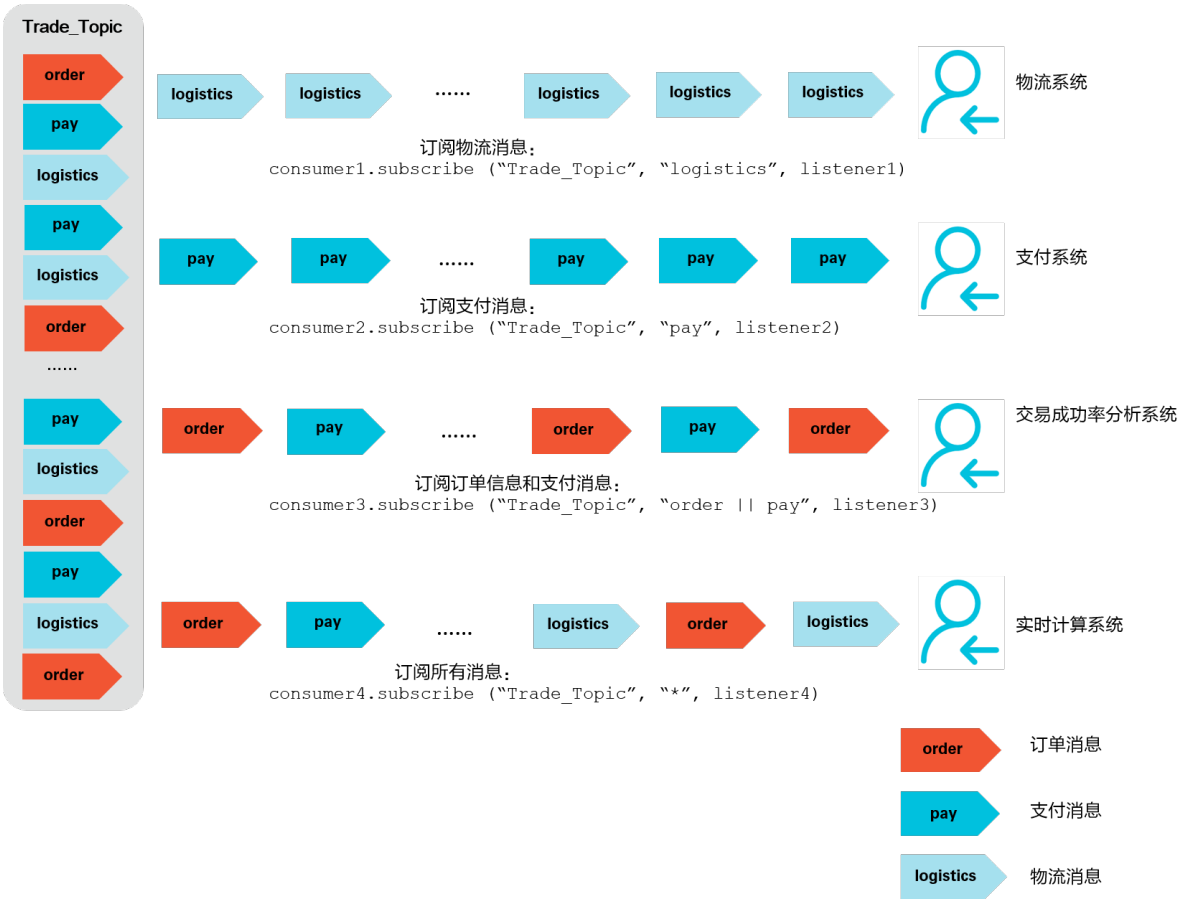
大小限制需要特殊注意，因为消息是动态的，不注意的话就可能超限，就会报错：

计算消息的大小

```
//计算消息的大小 = (topic + body +  (key + value) * N) * 吞吐量
int tmpSize = message.getTopic().length() + message.getBody().length;
Map<String, String> properties = message.getProperties();
for (Map.Entry<String, String> entry : properties.entrySet()) {
    tmpSize += entry.getKey().length() + entry.getValue().length();
}
```

```
public class BatchProducer {
  ....
}
```

3.6 过滤消息



3.6.1 TAG模式过滤

```
public class FilterByTagProducer {
  ....
}
```

```
public class FilterByTagConsumer {
...
}
```

3.6.2 SQL表达式过滤

SQL92表达式消息过滤，是通过**消息的属性**运行SQL过滤表达式进行条件匹配，消息发送时需要设置用户的属性putUserProperty方法设置属性。

支持的语法：

1. 数值比较, 如`>`, `>=`, `<`, `<=`, `BETWEEN`, `=`;
2. 字符比较, 如`=`, `<>`, `IN`;
3. `IS NULL` or `IS NOT NULL`;
4. 逻辑连接符`AND`, `OR`, `NOT`;

支持的类型：

1. 数值型, 如123, 3.1415;
2. 字符型, 如 'abc', 必须用单引号;
3. `NULL`, 特殊常数;
4. 布尔值, `TRUE` or `FALSE`;

```
 CODE: 1  DESC: The broker does not support consumer to filter message by
SQL92

enablePropertyFilter = true
```

```
public class FilterBySQLProducer {
  ....
}
```

```
public class FilterBySQLConsumer {
  ....
}
```

3.6.3 类过滤模式（基于4.2.0版本）

RocketMQ通过定义消息过滤类的接口实现消息过滤

3.7 事务消息

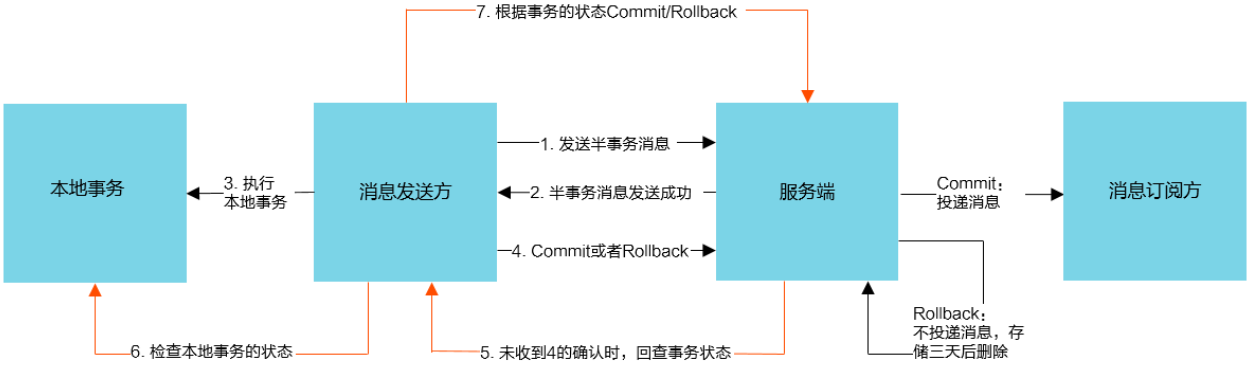
消息队列 RocketMQ 版提供的事务消息适用于所有对数据最终一致性有强需求的场景。

3.7.1 介绍

- 事务消息：消息队列 RocketMQ 版提供类似 X/Open XA 的分布式事务功能，通过消息队列 RocketMQ 事务消息能达到分布式事务的最终一致。
- **半事务消息**：暂不能投递的消息，发送方已经成功地将消息发送到了消息队列 RocketMQ 版服务端，但是服务端未收到生产者对该消息的二次确认，此时该消息被标记成“**暂不能投递**”状态，处于该种状态下的消息即半事务消息。
- 消息回查：由于网络闪断、生产者应用重启等原因，导致某条事务消息的二次确认丢失，消息队列 RocketMQ 版服务端通过扫描发现某条消息长期处于“半事务消息”时，需要**主动**向消息生产者询问该消息的最终状态（Commit 或是 Rollback），该询问过程即消息回查。

3.7.2 交互流程

RocketMQ的事务消息是基于两阶段提交实现的，**prepared和commited**。



事务消息发送：

1. 发送方将半事务消息发送至消息队列 RocketMQ 服务端。
2. 服务端将消息持久化成功之后，向发送方返回 Ack 确认消息已经发送成功，此时消息为半事务消息。
3. 发送方开始执行本地事务逻辑。
4. 发送方根据本地事务执行结果向服务端提交二次确认（Commit 或是 Rollback），服务端收到 Commit 状态则将半事务消息标记为可投递，订阅方最终将收到该消息；服务端收到 Rollback 状态则删除半事务消息，订阅方将不会接受该消息。

事务消息回查：

1. 在断网或者是应用重启的特殊情况下，上述步骤 4 提交的二次确认最终未到达服务端，经过固定时间后服务端将对该消息发起消息回查。
2. 发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
3. 发送方根据检查得到的本地事务的最终状态再次提交二次确认，服务端仍按照步骤 4 对半事务消息进行操作。

3.7.3 TransactionListener

```
public interface TransactionListener {
    /**
     * When send transactional prepare(half) message succeed, this method will
be invoked to execute local transaction.
```

```
     * 执行本地事务
     * @param msg Half(prepare) message
     * @param arg Custom business parameter
     * @return Transaction state
     */
    LocalTransactionState executeLocalTransaction(final Message msg, final
Object arg);

    /**
     * When no response to prepare(half) message. broker will send check
message to check the transaction status, and this
     * method will be invoked to get local transaction status.
     *消息回查后，需要检查对应消息的本地事务执行的最终结果
     * @param msg Check message
     * @return Transaction state
     */
    LocalTransactionState checkLocalTransaction(final MessageExt msg);
}
```

进入prepared状态以后，就要执行executeLocalTransaction方法，这个方法的返回值有3个

- LocalTransactionState.COMMIT_MESSAGE：提交消息，消息由prepared状态进入到commited状态，消费者可以消费这个消息；
- LocalTransactionState.ROLLBACK_MESSAGE：回滚，这个消息将被删除，消费者不能消费这个消息；
- LocalTransactionState.UNKNOW：未知，如果返回这个状态，这个消息既不提交，也不回滚，还是保持prepared状态，而最终决定这个消息命运的，是checkLocalTransaction这个方法。

当executeLocalTransaction方法返回UNKNOW以后，broker会每隔一段时间调用一次checkLocalTransaction，这个方法的返回值决定着这个消息的最终归宿。那么checkLocalTransaction这个方法多长时间调用一次呢?

BrokerConfig类

```
/**
     * Transaction message check interval.
     */
    @ImportantField
    private long transactionCheckInterval = 60 * 1000;
```

那么会检查多少次呢?

```
 /**
     * The maximum number of times the message was checked, if exceed this
value, this message will be discarded.
     */
    @ImportantField
    private int transactionCheckMax = 15;
```

这个是检查的最大次数，超过这个次数，如果还返回UNKNOW，这个消息将被删除。

3.7.4 示例

```
public class TransactionProducer {
    .....
}
```

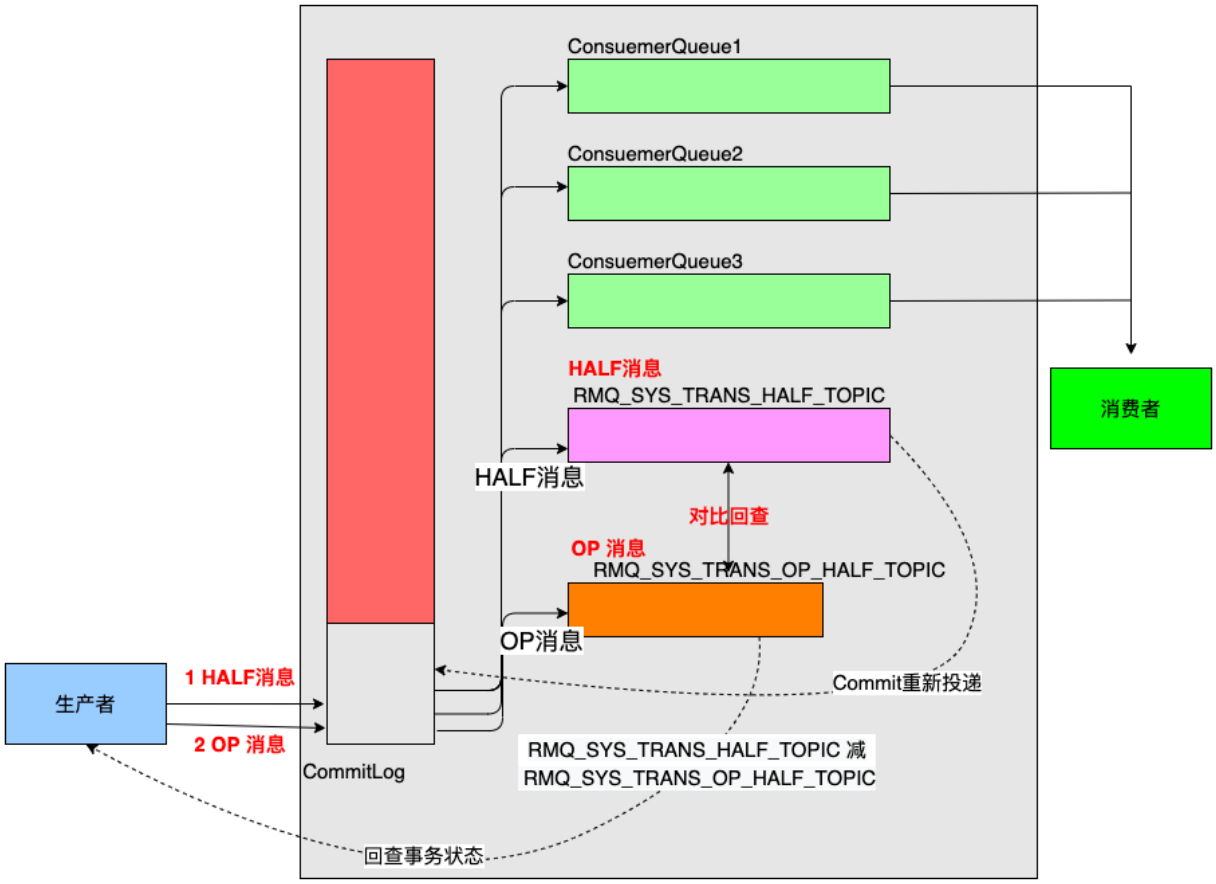
```
public class TransactionConsumer {
.....

}
```

3.7.5 注意事项

1. **事务消息不支持延时消息和批量消息。**
2. 为了避免单个消息被检查太多次而导致半队列消息累积，我们默认将单个消息的检查次数限制为 15 次，但是用户可以通过 Broker 配置文件的 **transactionCheckMax**参数来修改此限制。如果已经检查某条消息超过 N 次的话（ N = transactionCheckMax ） 则 Broker 将丢弃此消息，并在默认情况下同时打印错误日志。
3. 事务消息将在 Broker 配置文件中的参数 transactionTimeout 这样的特定时间长度之后被检查。当发送事务消息时，用户还可以通过设置用户属性 CHECK_IMMUNITY_TIME_IN_SECONDS 来改变这个限制，该参数优先于 transactionTimeout 参数。
4. 事务性消息可能不止一次被检查或消费，做好幂等性的检查
5. 提交给用户的目标主题消息可能会失败，目前这依日志的记录而定。它的高可用性通过 RocketMQ 本身的高可用性机制来保证，如果希望确保事务消息不丢失、并且事务完整性得到保证，建议使用同步的双重写入机制。
6. **事务消息的生产者 ID 不能与其他类型消息的生产者 ID 共享**。与其他类型的消息不同，事务消息允许反向查询、MQ服务器能通过它们的生产者 ID 查询到消费者。

3.7.6 事务消息原理



HALF消息:RMQ_SYS_TRANS_HALF_TOPIC(临时存放消息信息)

事务消息替换主题，保存原主题和队列信息

半消息对Consumer不可见，不会被投递

OP消息: RMQ_SYS_TRANS_OP_HALF_TOPIC(记录二阶段操作)

Rollback:只做记录

Commit:根据备份信息重新构造消息并投递

回查:

对比HALF消息和OP消息进行回查

```
RMQ_SYS_TRANS_HALF_TOPIC-RMQ_SYS_TRANS_OP_HALF_TOPIC = 消息回查消息
```