<http://www.cnblogs.com/charlesblc/p/6045238.html>

消息队列的两种模式：http://blog.csdn.net/heyutao007/article/details/50131089

消息中间件的主要功能是消息的路由(Routing)和缓存(Buffering)。在AMQP中提供类似功能的两种域模型：Exchange 和 Message queue。

AMQP的更多内容可以看这里： <http://www.cnblogs.com/charlesblc/p/6058799.html>

一种分类是推和拉 。

还有一种分类是 Queue 和 Pub/Sub 。

先看的这一篇：<http://blog.csdn.net/heyutao007/article/details/50131089>

先讲了JMS和遵守JMS的ActiveMQ。Java Message Service，JMS，指的是面向消息中间件（MOM），用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

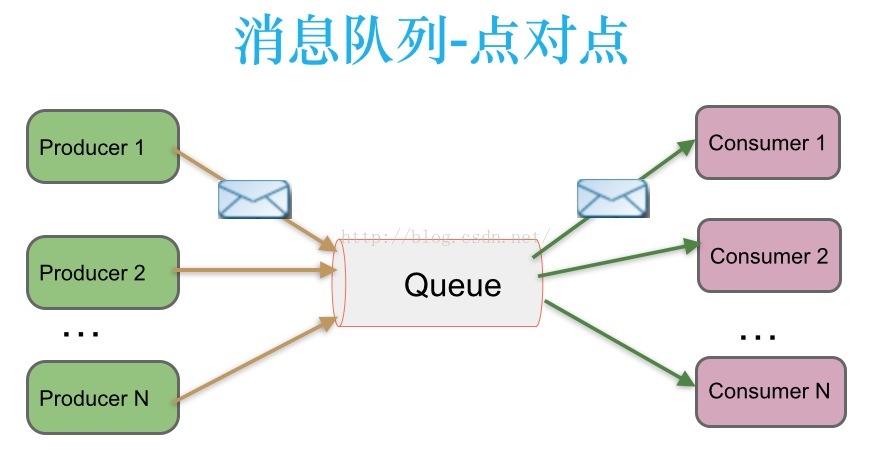
AMQP的原始用途只是为金融界提供一个可以彼此协作的消息协议，而现在的目标则是为通用消息队列架构提供通用构建工具。因此，面向消息的中间件（MOM）系统，例如发布/订阅队列，没有作为基本元素实现。反而通过发送简化的AMQ实体，用户被赋予了构建例如这些实体的能力。这些实体也是规范的一部分，形成了在线路层协议顶端的一个层级：AMQP模型。这个模型统一了消息模式，诸如之前提到的发布/订阅，队列，事务以及流数据，并且添加了额外的特性，例如更易于扩展，基于内容的路由。

JMS中定义了两种消息模型：点对点（point to point， queue）和发布/订阅（publish/subscribe，topic）。主要区别就是是否能重复消费。

## 点对点：Queue，不可重复消费

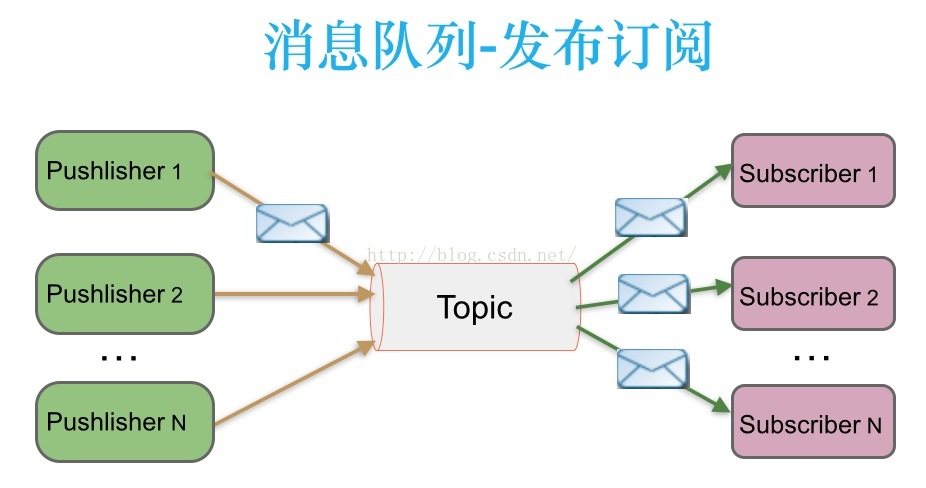
消息生产者生产消息发送到queue中，然后消息消费者从queue中取出并且消费消息。

消息被消费以后，queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。  
Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。  
注：Kafka不遵守JMS协议，所以Kafka实际应用中，很可能会需要ack，然后多个消费者能够会同时消费。。需要具体看。



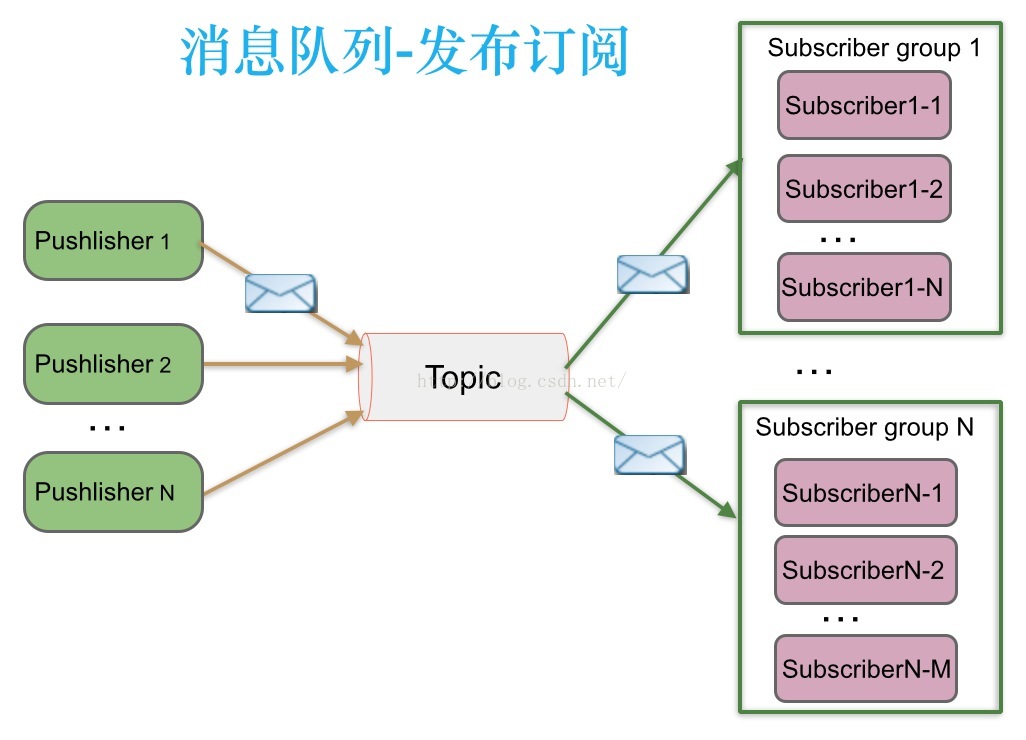
## 发布/订阅：Topic，可以重复消费

消息生产者（发布）将消息发布到topic中，同时有多个消息消费者（订阅）消费该消息。  
和点对点方式不同，发布到topic的消息会被所有订阅者消费。



支持订阅组的发布订阅模式：

发布订阅模式下，当发布者消息量很大时，显然单个订阅者的处理能力是不足的。  
实际上现实场景中是多个订阅者节点组成一个订阅组负载均衡消费topic消息即分组订阅，这样订阅者很容易实现消费能力线性扩展。



注：queue和topic在ActiveMQ里面的实现和对比，可以参考：《[ActiveMQ的queue以及topic两种消息处理机制分析](http://blog.csdn.net/studyforir/article/details/48340619)》

有完整queue和topic对比的代码可以看这里：<http://blog.csdn.net/zmx729618/article/details/51082844>

**可以看出区别 topic 是 session.createTopic("FirstTopic"); 而queue是 createQueue.**

# 流行模型比较

 传统企业型消息队列ActiveMQ遵循了JMS规范，实现了点对点和发布订阅模型，但其他流行的消息队列RabbitMQ、Kafka并没有遵循JMS规范。

## 3.1、RabbitMQ

RabbitMQ实现了AMQP协议，AMQP协议定义了消息路由规则和方式。

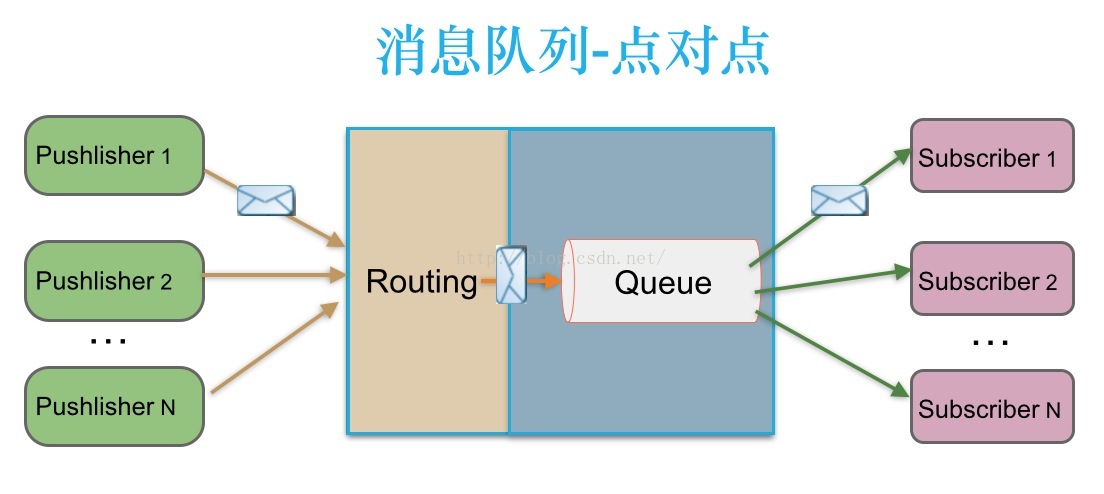
（更多AMQP内容，看这里：<http://www.cnblogs.com/charlesblc/p/6058799.html>）

生产端通过路由规则发送消息到不同queue，消费端根据queue名称消费消息。

RabbitMQ既支持内存队列也支持持久化队列，消费端为推模型，消费状态和订阅关系由**服务端**负责维护，消息消费完后立即删除，不保留历史消息。

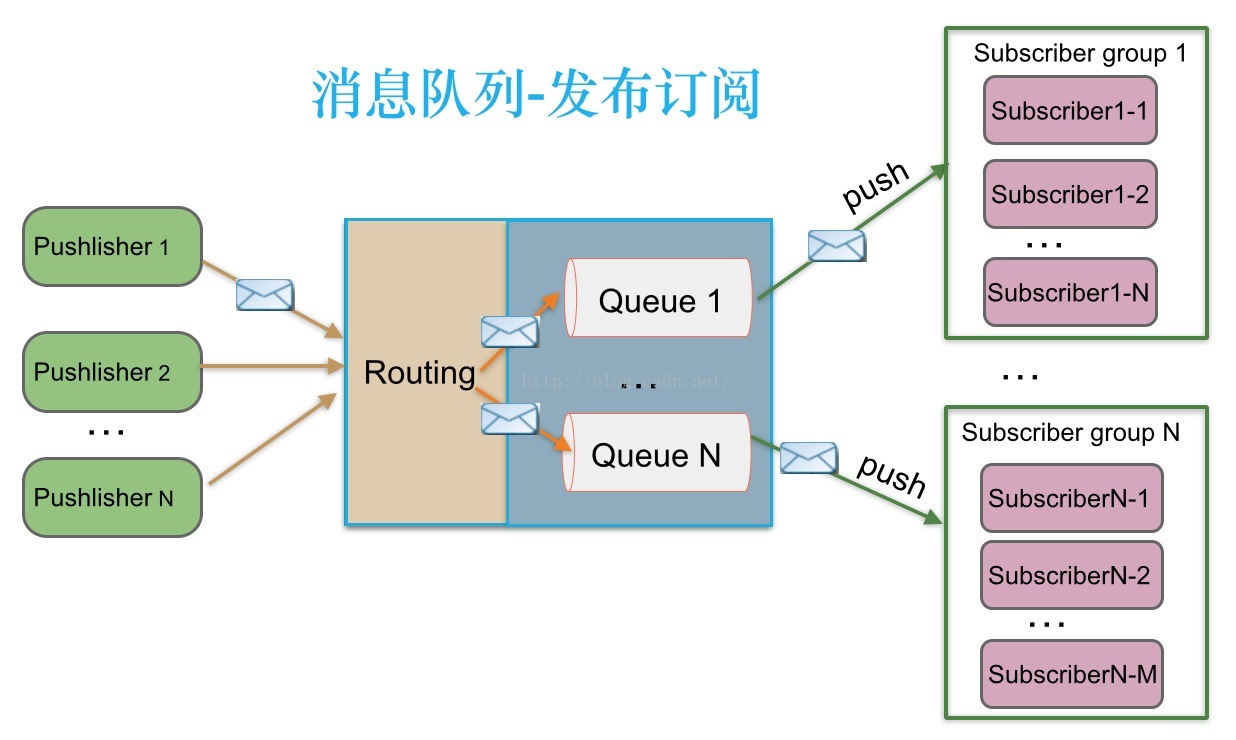
（1）点对点

生产端发送一条消息通过路由投递到Queue，只有一个消费者能消费到。



（2）多订阅

当RabbitMQ需要支持多订阅时，发布者发送的消息通过路由同时写到多个Queue，不同订阅组消费不同的Queue。  
所以支持多订阅时，消息会多个拷贝。



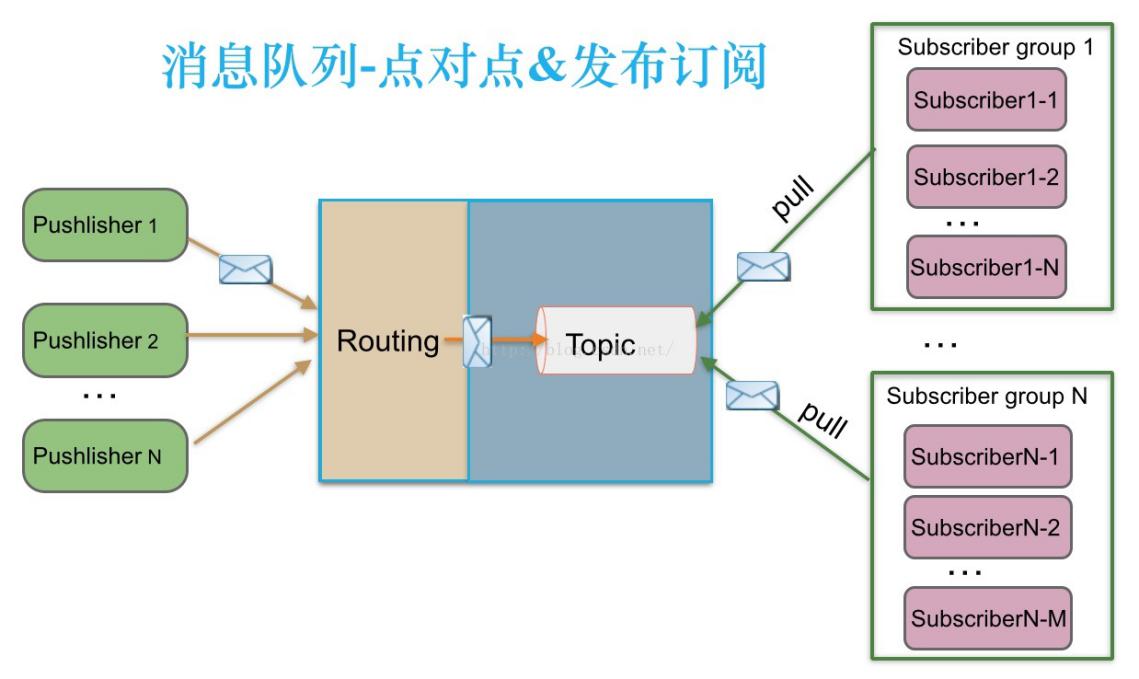
## 3.2、Kafka

Kafka只支持消息持久化，消费端为拉模型，消费状态和订阅关系由客户端端负责维护，消息消费完后不会立即删除，会保留历史消息。

因此支持多订阅时，消息只会存储一份就可以了。但是可能产生重复消费的情况。

（1）点对点&多订阅（因为**不删消息**，所以这两种就不区分了）

发布者生产一条消息到topic中，不同订阅组消费此消息。



上面是三种最流行MQ的比较（ActiveMQ, RabbitMQ, Kafka，没有涉及C++的zeorq）。

下面这篇文章针对ActiveMQ的推拉模型进行介绍。

<http://www.cnblogs.com/hapjin/p/5683648.html>

对于消费者而言有两种方式从消息中间件获取消息：

①Push方式：由消息中间件主动地将消息推送给消费者；  
②Pull方式：由消费者主动向消息中间件拉取消息。

看一段官网对Push方式的解释：

To be able to achieve high performance it is important to stream messages to consumers as fast as possible

so that the consumer always has a buffer of messages, in RAM, ready to process

- rather than have them explicitly pull messages from the server which adds significant latency per message.

比较：

采用Push方式，可以尽可能快地将消息发送给消费者(stream messages to consumers as fast as possible)

而采用Pull方式，会增加消息的延迟，即消息到达消费者的时间有点长(adds significant latency per message)。

但是，Push方式会有一个坏处：

如果消费者的处理消息的能力很弱(一条消息需要很长的时间处理)，而消息中间件不断地向消费者Push消息，消费者的缓冲区可能会溢出。

ActiveMQ是怎么解决这个问题的呢？那就是 [**prefetch limit**](http://activemq.apache.org/what-is-the-prefetch-limit-for.html)

prefetch limit 规定了一次可以向消费者Push(推送)多少条消息。

Once the prefetch limit is reached, no more messages are dispatched to the consumer

until the consumer starts sending back acknowledgements of messages (to indicate that the message has been processed)

当推送消息的数量到达了perfetch limit规定的数值时，消费者还没有向消息中间件返回ACK，消息中间件将不再继续向消费者推送消息。

prefetch limit设置的大小根据场景而定：

[复制代码](javascript:void(0);)

那prefetch limit的值设置为多少合适？视具体的应用场景而定。

If you have very few messages and each message takes a very long time to process

you might want to set the prefetch value to 1 so that a consumer is given one message at a time.

如果消息的数量很少(生产者生产消息的速率不快)，但是每条消息 消费者需要很长的时间处理，那么prefetch limit设置为1比较合适。  
这样，消费者每次只会收到一条消息，当它处理完这条消息之后，向消息中间件发送ACK，此时消息中间件再向消费者推送下一条消息。

[复制代码](javascript:void(0);)

**prefetch limit 设置成0意味着什么？意味着变成 拉pull模式。**

Specifying a prefetch limit of zero means the consumer will poll for more messages, one at a time,

instead of the message being pushed to the consumer.

意味着此时，消费者去轮询消息中间件获取消息。不再是Push方式了，而是Pull方式了。即消费者主动去消息中间件拉取消息。

prefetch Limit>0即为prefetch，=0为Pull，看起来没有不prefetch的push，push都要设置prefetch。

另外，对于prefetch模式（，那么消费需要进行响应ACK。因为服务器需要知道consumer消费的情况。

perfetch limit是“消息预取”的值，这是针对消息中间件如何向消费者发消息 而设置的。  
与之相关的还有针对 消费者以何种方式向消息中间件返回确认ACK(响应)：  
比如消费者是每次消费一条消息之后就向消息中间件确认呢？还是采用“延迟确认”---即采用批量确认的方式(消费了若干条消息之后，统一再发ACK)。

这就是 [Optimized Acknowledge](http://activemq.apache.org/performance-tuning.html)

[引用 一段话](http://shift-alt-ctrl.iteye.com/blog/2020182)：

如果prefetchACK为true，那么prefetch必须大于0；当prefetchACK为false时，你可以指定prefetch为0以及任意大小的正数。

不过，当prefetch=0是，表示consumer将使用PULL(拉取)的方式从broker端获取消息，  
broker端将不会主动push消息给client端，直到client端发送PullCommand时；

当prefetch>0时，就开启了broker push模式，此后只要当client端消费且ACK了一定的消息之后，会立即push给client端多条消息。

**在程序中如何采用Push方式或者Pull方式呢？**

从是否阻塞来看，消费者有两种方式获取消息。同步方式和异步方式。

同步方式使用的是ActiveMQMessageConsumer的receive()方法。而异步方式则是采用消费者实现MessageListener接口，监听消息。

同步方式：

[复制代码](javascript:void(0);)

使用同步方式receive()方法获取消息时，prefetch limit即可以设置为0，也可以设置为大于0

prefetch limit为零 意味着：  
“receive()方法将会首先发送一个PULL指令并阻塞，直到broker端返回消息为止，这也意味着消息只能逐个获取(类似于Request<->Response)”

prefetch limit 大于零 意味着：  
“broker端将会批量push给client一定数量的消息(<= prefetch)，client端会把这些消息(unconsumed Message)放入到本地的队列中，  
只要此队列有消息，那么receive方法将会立即返回（并消费），  
当一定量的消息ACK之后，broker端会继续批量push消息给client端。”

[复制代码](javascript:void(0);)

异步方式：

当使用MessageListener异步获取消息时，prefetch limit必须大于零了。  
因为，prefetch limit 等于零 意味着消息中间件不会主动给消费者Push消息，而此时消费者又用MessageListener被动获取消息(不会主动去轮询消息)。  
这二者是矛盾的。

**此外，还有一个要注意的地方，即消费者采用同步获取消息(receive方法) 与 异步获取消息的方法(MessageListener) ，对消息的确认时机是不同的。**

**这里提到了这篇文章：**<http://shift-alt-ctrl.iteye.com/blog/2020182> 文章名《[ActiveMQ消息传送机制以及ACK机制详解](http://shift-alt-ctrl.iteye.com/blog/2020182)》

**ActiveMQ消息传送机制**

[复制代码](javascript:void(0);)

Producer客户端使用来发送消息的， Consumer客户端用来消费消息；

它们的协同中心就是ActiveMQ broker,broker也是让producer和consumer调用过程解耦的工具，最终实现了异步RPC/数据交换的功能。

随着ActiveMQ的不断发展，支持了越来越多的特性，也解决开发者在各种场景下使用ActiveMQ的需求。

比如producer支持异步调用；

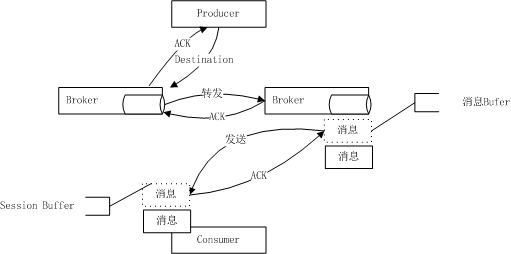
使用flow control机制让broker协同consumer的消费速率；

consumer端可以使用prefetchACK来最大化消息消费的速率；

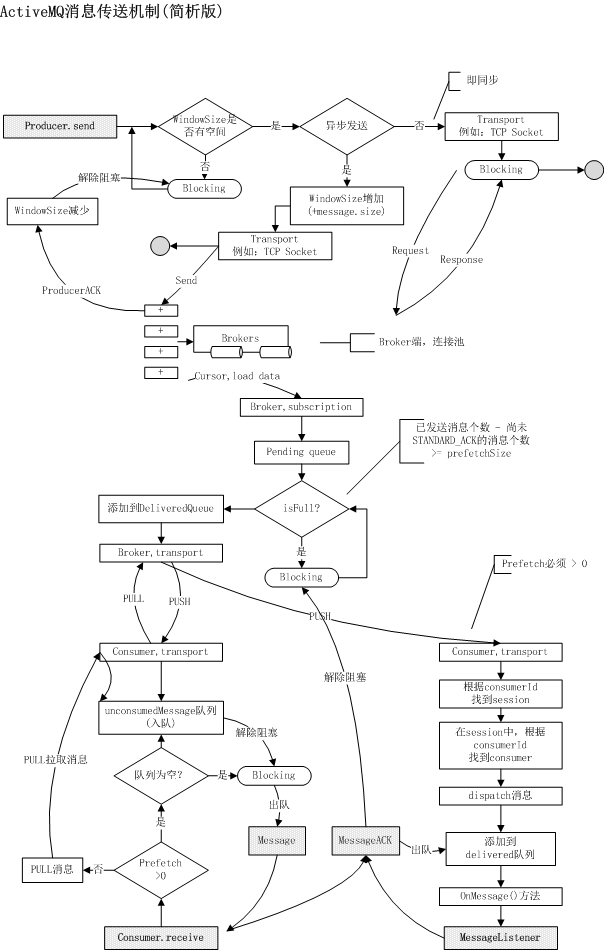
提供"重发策略"等来提高消息的安全性等。

[复制代码](javascript:void(0);)

一条消息的生命周期如下:



一条消息从producer端发出之后，一旦被broker正确保存，那么它将会被consumer消费，然后ACK，broker端才会删除；  
不过当消息过期或者存储设备溢出时，也会终结它。



上面的图里面写的很清晰。

上半部分是producer的流程，下半部分consumer的流程分为两块，同步的consumer.receive和异步的MessageListener。从图中可以看出异步的MessageLister也是一条一条处理的，由delivered队列控制的。

这张图片中简单的描述了:1)producer端如何发送消息 2) consumer端如何消费消息 3) broker端如何调度。  
如果用文字来描述图示中的概念，恐怕一言难尽。  
图示中，提及到prefetchAck，以及消息同步、异步发送的基本逻辑；这对你了解下文中的ACK机制将有很大的帮助。

**Prefetch和optimizeACK**

我们需要在brokerUrl指定optimizeACK选项，在destinationUri中指定prefetchSize(预获取)选项。

[复制代码](javascript:void(0);)

其中brokerUrl参数选项是全局的，即当前factory下所有的connection/session/consumer都会默认使用这些值；  
而destinationUri中的选项，只会在使用此destination的consumer实例中有效；  
如果同时指定，brokerUrl中的参数选项值将会被覆盖。  
optimizeAck表示是否开启“优化ACK”，只有在为true的情况下，  
prefetchSize(下文中将会简写成prefetch)以及optimizeAcknowledgeTimeout参数才会有意义。（prefetch依赖于optimizeAck？看起来是笔误）

此处需要注意"optimizeAcknowledgeTimeout"选项只能在brokerUrl中配置。

prefetch值建议在destinationUri中指定，因为在brokerUrl中指定比较繁琐；  
在brokerUrl中，queuePrefetchSize和topicPrefetchSize都需要单独设定：  
"&jms.prefetchPolicy.queuePrefetch=12&jms.prefetchPolicy.topicPrefetch=12"等来逐个指定。

1) 在brokerUrl中增加如下查询字符串：

String brokerUrl = "tcp://localhost:61616?" +

"jms.optimizeAcknowledge=true" +

"&jms.optimizeAcknowledgeTimeOut=30000" +

"&jms.redeliveryPolicy.maximumRedeliveries=6";

ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory(brokerUrl);

2) 在destinationUri中，增加如下查询字符串：

String queueName = "test-queue?customer.prefetchSize=100";

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

Destination queue = session.createQueue(queueName);

[复制代码](javascript:void(0);)

关于prefetchAck、同步、异步api（上面讲过了，温习一下）：

[复制代码](javascript:void(0);)

如果prefetchACK为true，那么prefetch必须大于0；  
当prefetchACK为false时，你可以指定prefetch为0以及任意大小的正数。  
不过，当prefetch=0是，表示consumer将使用PULL(拉取)的方式从broker端获取消息，  
broker端将不会主动push消息给client端，直到client端发送PullCommand时；  
当prefetch>0时，就开启了broker push模式，此后只要当client端消费且ACK了一定的消息之后，会立即push给client端多条消息。

当consumer端使用receive()方法同步获取消息时，prefetch可以为0和任意正值；  
当prefetch=0时，那么receive()方法将会首先发送一个PULL指令并阻塞，  
直到broker端返回消息为止，这也意味着消息只能逐个获取(类似于Request<->Response)，这也是Activemq中PULL消息模式；  
当prefetch > 0时，broker端将会批量push给client 一定数量的消息(<= prefetch),client端会把这些消息(unconsumedMessage)放入到本地的队列中，  
只要此队列有消息，那么receive方法将会立即返回，当一定量的消息ACK之后，broker端会继续批量push消息给client端。

当consumer端使用MessageListener异步获取消息时，这就需要开发设定的prefetch值必须 >=1,即至少为1；  
在异步消费消息模式中，设定prefetch=0,是相悖的，也将获得一个Exception。

[复制代码](javascript:void(0);)

重发选项：

我们还可以brokerUrl中配置“redelivery”策略，比如当一条消息处理异常时，broker端可以重发的最大次数；  
和下文中提到REDELIVERED\_ACK\_TYPE互相协同。当消息需要broker端重发时，  
consumer会首先在本地的“deliveredMessage队列”(Consumer已经接收但还未确认的消息队列)删除它，  
然后向broker发送“REDELIVERED\_ACK\_TYPE”类型的确认指令，  
broker将会把指令中指定的消息重新添加到pendingQueue(亟待发送给consumer的消息队列)中，直到合适的时机，再次push给client。

consumer消费快慢，决定了架构和设计如何处理：

按照良好的设计准则，  
当consumer消费速度很慢时，我们通常会部署多个consumer客户端，并使用较小的prefetch，同时关闭optimizeACK，  
可以让消息在多个consumer间“负载均衡”(即均匀的发送给每个consumer)；  
如果较大的prefetchSize，将会导致broker一次性push给client大量的消息，但是这些消息需要很久才能ACK(消息积压)，  
而且在client故障时，还会导致这些消息的重发。

其他情景：

如果consumer端消费速度很快，但是producer端生成消息的速率较慢，而且我们还部署了多个consumer，  
这种场景下，建议开启optimizeACK，但是需要设置的prefetchSize不能过大；  
这样可以保证每个consumer都能有"活干"，否则将会出现一个consumer非常忙碌，但是其他consumer几乎收不到消息。

如果消息很重要，特别是不愿意接收到”redelivery“的消息，那么我们需要将optimizeACK=false，prefetchSize=1

错误处理与重发：

既然optimizeACK是”延迟“确认，那么就引入一种潜在的风险：  
在消息被消费之后还没有来得及确认时，client端发生故障，  
那么这些消息就有可能会被重新发送给其他consumer，那么这种风险就需要client端能够容忍“重复”消息。

从上面的图可以看出，没有ACK的情况下，队列是blocking的。

无论如何设定此值，client持有的消息条数最大为：prefetch + “DELIVERED\_ACK\_TYPE消息条数”(DELIVERED\_ACK\_TYPE参见下文)

**optimizeACK其他注意：**

[复制代码](javascript:void(0);)

即使当optimizeACK为true，也只会当session的ACK模式为AUTO\_ACKNOWLEDGE时才会生效，即在其他类型的ACK模式时consumer端仍然不会“延迟确认”，即:

consumer.optimizeAck = connection.optimizeACK && session.isAutoAcknowledge()

当consumer.optimizeACK有效时，如果客户端已经消费但尚未确认的消息(deliveredMessage)达到prefetch \* 0.65，consumer端将会自动进行ACK；  
同时如果离上一次ACK的时间间隔，已经超过"optimizeAcknowledgeTimout"毫秒，也会导致自动进行ACK。

此外简单的补充一下，批量确认消息时，只需要在ACK指令中指明“firstMessageId”和“lastMessageId”即可，即消息区间，  
那么broker端就知道此consumer(根据consumerId识别)需要确认哪些消息。

[复制代码](javascript:void(0);)

**ACK模式与类型介绍**

JMS API中约定了Client端可以使用四种ACK模式

[复制代码](javascript:void(0);)

在javax.jms.Session接口中:

AUTO\_ACKNOWLEDGE = 1 自动确认

CLIENT\_ACKNOWLEDGE = 2 客户端手动确认

DUPS\_OK\_ACKNOWLEDGE = 3 自动批量确认

SESSION\_TRANSACTED = 0 事务提交并确认

此外AcitveMQ补充了一个自定义的ACK模式:

INDIVIDUAL\_ACKNOWLEDGE = 4 单条消息确认

[复制代码](javascript:void(0);)

对于broker而言，只有接收到ACK指令,才会认为消息被正确的接收或者处理成功了,通过ACK，可以在consumer（/producer）与Broker之间建立一种简单的“担保”机制.

[复制代码](javascript:void(0);)

Client端指定了ACK模式,但是在Client与broker在交换ACK指令的时候,还需要告知ACK\_TYPE,ACK\_TYPE表示此确认指令的类型，  
不同的ACK\_TYPE将传递着消息的状态，broker可以根据不同的ACK\_TYPE对消息进行不同的操作。

比如Consumer消费消息时出现异常,就需要向broker发送ACK指令,ACK\_TYPE为"REDELIVERED\_ACK\_TYPE",那么broker就会重新发送此消息。  
在JMS API中并没有定义ACT\_TYPE,因为它通常是一种内部机制,并不会面向开发者。ActiveMQ中定义了如下几种ACK\_TYPE(参看MessageAck类):

DELIVERED\_ACK\_TYPE = 0 消息"已接收"，但尚未处理结束

STANDARD\_ACK\_TYPE = 2 "标准"类型,通常表示为消息"处理成功"，broker端可以删除消息了

POSION\_ACK\_TYPE = 1 消息"错误",通常表示"抛弃"此消息，比如消息重发多次后，都无法正确处理时，消息将会被删除或者DLQ(死信队列)

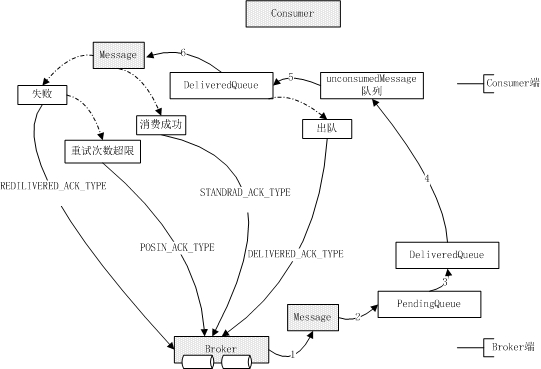
REDELIVERED\_ACK\_TYPE = 3 消息需"重发"，比如consumer处理消息时抛出了异常，broker稍后会重新发送此消息

INDIVIDUAL\_ACK\_TYPE = 4 表示只确认"单条消息",无论在任何ACK\_MODE下

UNMATCHED\_ACK\_TYPE = 5 在Topic中，如果一条消息在转发给“订阅者”时，发现此消息不符合Selector过滤条件，那么此消息将 不会转发给订阅者，  
消息将会被存储引擎删除(相当于在Broker上确认了消息)。

[复制代码](javascript:void(0);)

ACK的基本流程见下图：



Consumer消费消息的风格有2种: 同步/异步. 使用consumer.receive()就是同步，使用messageListener就是异步。

在同一个consumer中，我们不能同时使用这2种风格，比如在使用listener的情况下，当调用receive()方法将会获得一个Exception。

两种风格下，消息确认时机有所不同。

[复制代码](javascript:void(0);)

"同步"伪代码：

//receive伪代码---过程

Message message = sessionMessageQueue.dequeue();

if(message != null){

ack(message);

}

return message

同步调用时，在消息从receive方法返回之前，就已经调用了ACK；因此如果Client端没有处理成功，此消息将丢失(可能重发，与ACK模式有关)。

"异步"伪代码：

//基于listener

Session session = connection.getSession(consumerId);

sessionQueueBuffer.enqueue(message);

Runnable runnable = new Ruannale(){

run(){

Consumer consumer = session.getConsumer(consumerId);

Message md = sessionQueueBuffer.dequeue();

try{

consumer.messageListener.onMessage(md);

ack(md);//

}catch(Exception e){

redelivery();//sometime，not all the time;

}

}

//session中将采取线程池的方式，分发异步消息

//因此同一个session中多个consumer可以并行消费

threadPool.execute(runnable);

基于异步调用时，消息的确认是在onMessage方法返回之后，如果onMessage方法异常，会导致消息不能被ACK，会触发重发。

[复制代码](javascript:void(0);)

**ACK模式详解**

**AUTO\_ACKNOWLEDGE :**

[复制代码](javascript:void(0);)

自动确认,这就意味着消息的确认时机将有consumer择机确认.  
"择机确认"似乎充满了不确定性,这也意味着,开发者必须明确知道"择机确认"的具体时机,否则将有可能导致消息的丢失,或者消息的重复接收.  
那么在ActiveMQ中,AUTO\_ACKNOWLEDGE是如何运作的呢?

1) 对于consumer而言，optimizeAcknowledge属性只会在AUTO\_ACK模式下有效。

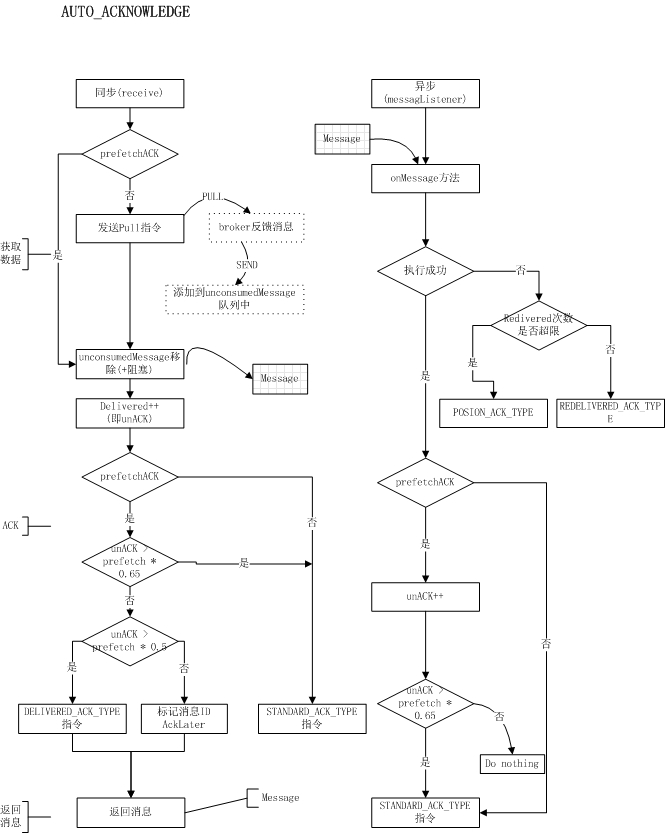
2) 其中DUPS\_ACKNOWLEGE也是一种潜在的AUTO\_ACK,只是确认消息的条数和时间上有所不同。

3) 在“同步”(receive)方法返回message之前,会检测optimizeACK选项是否开启，如果没有开启，此单条消息将立即确认，  
所以在这种情况下，message返回之后，如果开发者在处理message过程中出现异常，会导致此消息也不会redelivery,即"潜在的消息丢失"；  
如果开启了optimizeACK，则会在unAck数量达到prefetch \* 0.65时确认，当然我们可以指定prefetchSize = 1来实现逐条消息确认。

4) 在"异步"(messageListener)方式中,将会首先调用listener.onMessage(message),此后再ACK,  
如果onMessage方法异常,将导致client端补充发送一个ACK\_TYPE为REDELIVERED\_ACK\_TYPE确认指令；  
如果onMessage方法正常,消息将会正常确认(STANDARD\_ACK\_TYPE)。  
此外需要注意，消息的重发次数是有限制的，每条消息中都会包含“redeliveryCounter”计数器，用来表示此消息已经被重发的次数，  
如果重发次数达到阀值，将会导致发送一个ACK\_TYPE为POSION\_ACK\_TYPE确认指令,这就导致broker端认为此消息无法消费,  
此消息将会被删除或者迁移到"dead letter"通道中。

因此当我们使用messageListener方式消费消息时，通常建议在onMessage方法中使用try-catch,这样可以在处理消息出错时记录一些信息，  
而不是让consumer不断去重发消息；  
如果你没有使用try-catch,就有可能会因为异常而导致消息重复接收的问题,需要注意你的onMessage方法中逻辑是否能够兼容对重复消息的判断。

[复制代码](javascript:void(0);)



**CLIENT\_ACKNOWLEDGE :**

客户端手动确认，这就意味着AcitveMQ将不会“自作主张”的为你ACK任何消息，开发者需要自己择机确认。

无论是“同步”/“异步”，ActiveMQ都不会发送STANDARD\_ACK\_TYPE，直到message.acknowledge()调用。  
如果在client端未确认的消息个数达到prefetchSize \* 0.5时，会补充发送一个ACK\_TYPE为DELIVERED\_ACK\_TYPE的确认指令，  
这会触发broker端可以继续push消息到client端。

注意防止不ack而hang住：

如果client端因为某种原因导致acknowledge方法未被执行，将导致大量消息不能被确认，  
broker端将不会push消息，事实上client端将处于“假死”状态，而无法继续消费消息。  
  
我们要求client端在消费1.5\*prefetchSize个消息之前，必须acknowledge()一次；  
通常我们总是每消费一个消息调用一次，这是一种良好的设计。

broker依据ack速率进行负载平衡：

[复制代码](javascript:void(0);)

在CLIET\_ACK模式下，消息在交付给listener之前，都会首先创建一个DELIVERED\_ACK\_TYPE的ACK指令，  
直到client端未确认的消息达到"prefetchSize \* 0.5"时才会发送此ACK指令，如  
果在此之前，开发者调用了acknowledge()方法，会导致消息直接被确认(STANDARD\_ACK\_TYPE)。  
  
broker端通常会认为“DELIVERED\_ACK\_TYPE”确认指令是一种“slow consumer”信号，  
如果consumer不能及时的对消息进行acknowledge而导致broker端阻塞，那么此consumer将会被标记为“slow”，  
此后queue中的消息将会转发给其他Consumer。

[复制代码](javascript:void(0);)

**DUPS\_OK\_ACKNOWLEDGE :**

"消息可重复"确认，意思是此模式下，可能会出现重复消息，并不是一条消息需要发送多次ACK才行。  
它是一种潜在的"AUTO\_ACK"确认机制，为批量确认而生，而且具有“延迟”确认的特点。  
  
对于开发者而言，这种模式下的代码结构和AUTO\_ACKNOWLEDGE一样，不需要像CLIENT\_ACKNOWLEDGE那样调用acknowledge()方法来确认消息。

发生作用的时机：

[复制代码](javascript:void(0);)

1) 在ActiveMQ中，如果在Destination是Queue通道，我们真的可以认为DUPS\_OK\_ACK就是“AUTO\_ACK + optimizeACK + (prefetch > 0)”  
这种情况，在确认时机上几乎完全一致；  
此外在此模式下，如果prefetchSize =1 或者没有开启optimizeACK，也会导致消息逐条确认，从而失去批量确认的特性。

2) 如果Destination为Topic，DUPS\_OK\_ACKNOWLEDGE才会产生JMS规范中诠释的意义，  
即无论optimizeACK是否开启，都会在消费的消息个数>=prefetch \* 0.5时，批量确认(STANDARD\_ACK\_TYPE),  
在此过程中，不会发送DELIVERED\_ACK\_TYPE的确认指令,这是DUPS和AUTO\_ACK的最大的区别。

这也意味着，当consumer故障重启后，那些尚未ACK的消息会重新发送过来。

[复制代码](javascript:void(0);)

**SESSION\_TRANSACTED :**

当session使用事务时，就是使用此模式。

[复制代码](javascript:void(0);)

在事务开启之后，和session.commit()之前，所有消费的消息，要么全部正常确认，要么全部redelivery。  
这种严谨性，通常在基于GROUP(消息分组)或者其他场景下特别适合。  
  
在SESSION\_TRANSACTED模式下，optimizeACK并不能发挥任何效果,因为在此模式下，optimizeACK会被强制设定为false，  
不过prefetch仍然可以决定DELIVERED\_ACK\_TYPE的发送时机。

因为Session非线程安全，那么当前session下所有的consumer都会共享同一个transactionContext；  
同时建议，一个事务类型的Session中只有一个Consumer，以避免rollback()或者commit()方法被多个consumer调用而造成的消息混乱。

[复制代码](javascript:void(0);)

确认过程，以及确认ACK的发送时机：

事务的确认过程中，首先把本地的deliveredMessage队列中尚未确认的消息全部确认(STANDARD\_ACK\_TYPE)；  
此后向broker发送transaction提交指令并等待broker反馈，  
  
如果broker端事务操作成功，那么将会把本地deliveredMessage队列清空，新的事务开始；  
如果broker端事务操作失败(此时broker已经rollback)，那么对于session而言，将执行inner-rollback，  
这个rollback所做的事情，就是将当前事务中的消息清空并要求broker重发(REDELIVERED\_ACK\_TYPE),同时commit方法将抛出异常。

**INDIVIDUAL\_ACKNOWLEDGE :**

很少使用，它的确认时机和CLIENT\_ACKNOWLEDGE几乎一样

当消息消费成功之后，需要调用message.acknowledege来确认此消息(单条)，  
而CLIENT\_ACKNOWLEDGE模式先message.acknowledge()方法将导致整个session中所有消息被确认(批量确认)。

（完）