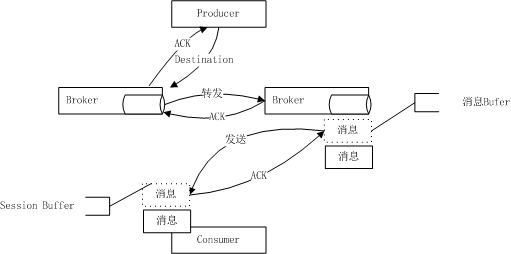
**ActiveMQ**

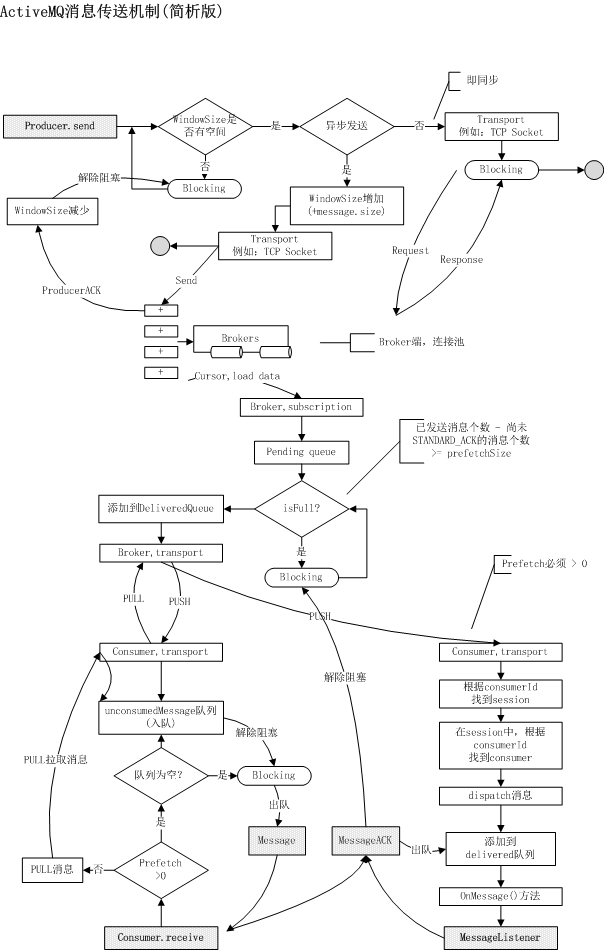
# 消息传送机制

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，在此不作详细介绍。

1. 消息的生命周期如下:



图片中简单的描述了一条消息的生命周期,不过在不同的架构环境中,message的流动行可能更加复杂.将在稍后有关broker的架构中详解..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



这是一张很复杂，而且有些凌乱的图片；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ACK机制有很大的帮助。

# optimizeACK

"可优化的ACK"，这是ActiveMQ对于consumer在消息消费时，对消息ACK的优化选项，也是consumer端最重要的优化参数之一，你可以通过如下方式开启:

1. 在brokerUrl中增加如下查询字符串：

Java代码 收藏代码

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

"jms.optimizeAcknowledge=true" +

"&jms.optimizeAcknowledgeTimeOut=30000" +

"&jms.redeliveryPolicy.maximumRedeliveries=6";

1. 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);

我们需要在brokerUrl指定optimizeACK选项，在destinationUri中指定prefetchSize(预获取)选项，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。此处需要注意"optimizeAcknowledgeTimeout"选项只能在brokerUrl中配置。

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。

此外，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。

到目前为止，或许你知道了optimizeACK和prefeth的大概意义，不过我们可能还会有些疑惑！！optimizeACK和prefetch配合，将会达成一个高效的消息消费模型：批量获取消息，并“延迟”确认(ACK)。prefetch表达了“批量获取”消息的语义，broker端主动的批量push多条消息给client端，总比client多次发送PULL指令然后broker返回一条消息的方式要优秀很多，它不仅减少了client端在获取消息时阻塞的次数和阻塞的时间，还能够大大的减少网络开支。optimizeACK表达了“延迟确认”的语义(ACK时机)，client端在消费消息后暂且不发送ACK，而是把它缓存下来(pendingACK)，等到这些消息的条数达到一定阀值时，只需要通过一个ACK指令把它们全部确认；这比对每条消息都逐个确认，在性能上要提高很多。由此可见，prefetch优化了消息传送的性能，optimizeACK优化了消息确认的性能。

当consumer端消息消费的速率很高(相对于producer生产消息)，而且消息的数量也很大时(比如消息源源不断的生产)，我们使用optimizeACK + prefetch将会极大的提升consumer的性能。不过反过来：

1) 如果consumer端消费速度很慢(对消息的处理是耗时的)，过大的prefetchSize，并不能有效的提升性能，反而不利于consumer端的负载均衡(只针对queue)；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。

2) 如果consumer端消费速度很快，但是producer端生成消息的速率较慢，比如生产者10秒钟生成10条消息，但是consumer一秒就能消费完毕，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。

3) 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

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。

prefetch值默认为1000，当然这个值可能在很多场景下是偏大的；我们暂且不考虑ACK模式(参见下文)，通常情况下，我们只需要简单的统计出单个consumer每秒的最大消费消息数即可，比如一个consumer每秒可以处理100个消息，我们期望consumer端每2秒确认一次，那么我们的prefetchSize可以设置为100 \* 2 /0.65大概为300。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)

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:

Java代码 收藏代码

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。

# 消息确认机制

## 自动确认

AUTO\_ACKNOWLEDGE

默认的确认机制。

1. 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prefech\*0.65时确，当然我们可以指定prefetchSize=1来实现逐条消息确认。
2. 在”异步”(messageLinstener)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)。此外需要注意的是，消息的重发次数是有限制的。

最佳实践：使用默认确认机制时要用try-catch ,当异常的时候将消息转发到异常消息队列，以便维护跟踪

## 手动确认

CLIENT\_ACKNOWLEDGE

作用效果：

message.acknowledge():将当前session中所有consumer中尚未确认的消息都一起确认。

ActiveMQMessageConsumer.acknowledege()：只会对当前consumer中那些尚未确认的消息进行确认。开发者可以在合适的时机必须调用一次上述方法。

ActiveMQSession.acknowledge():同message.acknowledge()

使用场景:

通常会在基于Group(消息分组)情况下使用。我们将在一个组的消息序列接受完毕之后确认消息(组)。不过当你认为消息很重要，只有当消息被正确处理之后才能确认时，也可以使用。

如果开发者忘记调用acknowledge方法，将会导致当consumer重启后，会接受到重复消息，因为对于broker而言，那些尚未真正ACK的消息被视为”未消费”。

无论是同步还是异步，ActiveMQ都不会发送STANDARD\_ACK\_TYPE，直到message.acknowledege()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borker端可以继续push消息到client端。(参看PrefetchSubscription.acknwoledge)

在broker端，针对每个Consumer，都会保存一个因为"DELIVERED\_ACK\_TYPE"而“拖延”的消息个数，这个参数为prefetchExtension，事实上这个值不会大于prefetchSize \* 0.5,因为Consumer端会严格控制DELIVERED\_ACK\_TYPE指令发送的时机(参见ActiveMQMessageConsumer.ackLater方法)，broker端通过“prefetchExtension”与prefetchSize互相配合，来决定即将push给client端的消息个数，count = prefetchExtension + prefetchSize - dispatched.size()，其中dispatched表示已经发送给client端但是还没有“STANDARD\_ACK\_TYPE”的消息总量；由此可见，在CLIENT\_ACK模式下，足够快速的调用acknowledge()方法是决定consumer端消费消息的速率；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。

此外需要额外的补充一下：所有ACK指令都是依次发送给broker端，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。

## 消息可重复确认

DUPS\_OK\_ACKNOWLEDGE

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。

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。

## 事务确认

SESSION\_TRANSACTED

当session使用事务时，就是使用此模式。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，已避免rollback()或者commit()方法被多个consumer调用而造成的消息混乱。

当consumer接受到消息之后，首先检测TransactionContext是否已经开启，如果没有，就会开启并生成新的transactionId，并把信息发送给broker；此后将检测事务中已经消费的消息个数是否 >= prefetch \* 0.5,如果大于则补充发送一个“DELIVERED\_ACK\_TYPE”的确认指令；这时就开始调用onMessage()方法，如果是同步(receive),那么即返回message。上述过程，和其他确认模式没有任何特殊的地方。

当开发者决定事务可以提交时，必须调用session.commit()方法，commit方法将会导致当前session的事务中所有消息立即被确认；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。

当session.commit方法异常时，对于开发者而言通常是调用session.rollback()回滚事务(事实上开发者不调用也没有问题)，当然你可以在事务开始之后的任何时机调用rollback(),rollback意味着当前事务的结束，事务中所有的消息都将被重发。需要注意，无论是inner-rollback还是调用session.rollback()而导致消息重发，都会导致message.redeliveryCounter计数器增加，最终都会受限于brokerUrl中配置的"jms.redeliveryPolicy.maximumRedeliveries",如果rollback的次数过多，而达到重发次数的上限时，消息将会被DLQ(dead letter)。

## 单条消息确认

INDIVIDUAL\_ACKNOWLEDGE

单条消息确认，这种确认模式，我们很少使用，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模式的message.acknowledge()方法将导致整个session中所有消息被确认(批量确认)。

# 参考

源码参考类：

1) ActiveMQConnectionFactory,ActiveMQMessageConsumer,ActiveMQSession,MessageAck等

2) Queue,PrefetchSubscription,TransactionContext,TransactionStore

<http://shift-alt-ctrl.iteye.com/blog/2020182> ActiveMQ消息确认机制