



01689179

Apache ActiveMQ 笔记

当前产品版本号	v5.2		
最初发布日期			
最新修订日期	2010 年 3 月 2 日		
审核者	邓建利	日期	
批准者	Apache	日期	



译者序

Java 是当前IT 领域中比较流行的技术之一。J2EE 是当前比较流行的企业级应用架构。本人一直致力于J2EE 架构的学习和研究，但是总是对英文文档有不可言语的恐惧。我想很多J2EE 爱好者和我有同样的感觉。这样就影响了我们深入学习J2EE 原始规范的兴趣。但是J2EE 原始的规范文档对我们深入理解J2EE 有很大的帮助，因为它阐述了规范的来龙去脉，以及违反了规范会造成什么样的影响。了解了这些缘由和影响，会使我们对J2EE 架构有更深层次的理解。这也是我做该笔记是该规范的动力所在。

邓建利 【Alisd】

- ActiveMQ 官方网站: <http://activemq.apache.org>
- JMS 官方网站: <http://java.sun.com/products/jms>

第一章. 概述



背景

当前，CORBA、DCOM、RMI 等RPC 中间件技术已广泛应用于各个领域。但是面对规模和复杂度都越来越高的分布式系统，这些技术也显示出其局限性：（1）同步通信：客户发出调用后，必须等待服务对象完成处理并返回结果后才能继续执行；（2）客户和服务器对象的生命周期紧密耦合：客户进程和服务对象进程都必须正常运行；如果由于服务对象崩溃或者网络故障导致客户的请求不可达，客户会接收到异常；（3）点对点通信：客户的一次调用只发送给某个单独的目标对象。

面向消息的中间件（Message Oriented Middleware, MOM）较好的解决了以上问题。发送者将消息发送给消息服务器，消息服务器将消息存放在若干队列中，在合适的时候再将消息转发给接收者。这种模式下，发送和接收是异步的，发送者无需等待；

二者的生命周期未必相同：发送消息的时候接收者不一定运行，接收消息的时候发送者也不一定运行；一对多通信：对于一个消息可以有多个接收者。

已有的 MOM 系统包括IBM 的MQSeries、Microsoft 的MSMQ 和BEA 的MessageQ 等。由于没有一个通用的标准，这些系统很难实现互操作和无缝连接。Java Message Service (JMS) 是SUN 提出的旨在统一各种MOM 系统接口的规范，它包含点对点（Point to Point, PTP）和发布/订阅（Publish/Subscribe, pub/sub）两种消息模型，提供可靠消息传输、事务和消息过滤等机制。

1. 什么是消息中间件

面向消息的中间件：Message-oriented Middleware, MOM

基本功能：将信息以消息的形式，从一个应用程序传送到另一个或多个应用程序。

主要特点：

- 消息异步接受，类似手机短信的行为，消息发送者不需要等待消息接受者的响应，减少软件多系统集成的耦合度；
- 消息可靠接收，确保消息在中间件可靠保存，只有接收方收到后才删除消息，多个消息也可以组成原子事务。



消息中间件的主要应用场景：

公司在发展过程中，开发（或者购买了）多套企业信息系统，比如财务系统，人事系统，在线销售系统，运营系统等。

这些系统生产/消费公司的各种业务数据。

公司需要将这些系统集成（整合），比如让在线销售系统的订单数据输入到财务系统中。

类似应用的一般系统需求是：

- 可靠传输，数据不能丢失，有的时候，也会要求不能重复传输；
- 异步传输，否则各个系统同步发送接受数据，互相等待，造成系统瓶颈。

公司可以为此开发自己的软件服务，代价会比较大，现在一般使用已经成型的**消息中间件**。

目前比较知名的消息中间件：

- IBM MQSeries
- BEA WebLogic JMS Server
- Oracle AQ
- Tibco
- SwiftMQ
- AcitveMQ：是免费的 java 实现的消息中间件

什么是 **JMS**

JMS Java Message Service，Java消息服务。



JMS 概述

1.1.1. JMS 规范

JAVA 消息服务(JMS)定义了Java 中访问消息中间件的接口。JMS 只是接口，并没有给予实现，实现JMS 接口的消息中间件称为**JMS Provider**，例如 ActiveMQ。

1.1.2. 术语

JMS Provider: 实现JMS 接口的消息中间件；

PTP: Point to Point, 即点对点的消息模型；

Pub/Sub: Publish/Subscribe, 即发布/订阅的消息模型；

Queue: 队列目标；

Topic: 主题目标；

ConnectionFactory: 连接工厂，JMS 用它创建连接；

Connection: JMS 客户端到JMS Provider 的连接；

Destination: 消息的目的地；

Session: 会话，一个发送或接收消息的线程；

MessageProducer: 由Session 对象创建的用来发送消息的对象；

MessageConsumer: 由Session 对象创建的用来接收消息的对象；

Acknowledge: 签收；

Transaction: 事务。

1.1.3. JMS 编程模型

在 JMS 编程模型中，JMS 客户端（组件或应用程序）通过 JMS 消息服务交换消息。

消息生产者将消息发送至消息服务，消息消费者则从消息服务接收这些消息。这些消息

传送操作是使用一组实现 JMS 应用编程接口 (API) 的对象(由 JMS Provide 提供) 来

执行的。

在 JMS 编程模型中，JMS 客户端使用 ConnectionFactory 对象创建一个连接，向

消息服务发送消息以及从消息服务接收消息均是通过此连接来进行。**Connection** 是客

户端与消息服务的活动连接。创建连接时，将分配通信资源以及验证客户端。这是一个

相当重要的对象，大多数客户端均使用一个连接来进行所有的消息传送。

连接用于创建会话。**Session** 是一个用于生成和使用消息的单线程上下文。它用于

创建发送的生产者和接收消息的消费者，并为所发送的消息定义发送顺序。会话通过大



量确认选项或通过事务来支持可靠传送。

客户端使用 **MessageProducer** 向指定的物理目标（在 API 中表示为目标身份对象）

发送消息。生产者可指定一个默认传送模式（持久性消息与非持久性消息）、优先级和

有效期值，以控制生产者向物理目标发送的所有消息。

同样，客户端使用 **MessageConsumer** 对象从指定的物理目标（在 API 中表示为目

标对象）接收消息。消费者可使用消息选择器，借助它，消息服务可以只向消费者发送

与选择标准匹配的那些消息。

消费者可以支持同步或异步消息接收。异步使用可通过向消费者注册

MessageList

ener 来实现。当会话线程调用 **MessageListener** 对象的 **onMessage** 方法时，客户端

将使用消息。

1.1.4. JMS 编程域

JMS 支持两种截然不同的消息传送模型：PTP（即点对点模型）和Pub/Sub（即发布/订阅模型），分别称作：PTP Domain 和Pub/Sub Domain。

PTP（使用**Queue** 即队列目标） 消息从一个生产者传送至一个消费者。在此传送模型中，目标是一个队列。消息首先被传送至队列目标，然后根据队列传送策略，从

该队列将消息传送至向此队列进行注册的某一个消费者，一次只传送一条消息。可以向

队列目标发送消息的生产者的数量没有限制，但每条消息只能发送至、并由一个消费者

成功使用。如果没有已经向队列目标注册的消费者，队列将保留它收到的消息，并在某

个消费者向该队列进行注册时将消息传送给该消费者。

Pub/Sub（使用**Topic** 即主题目标） 消息从一个生产者传送至任意数量的消费者。在此传送模型中，目标是一个主题。消息首先被传送至主题目标，然后传送至所有

已订阅此主题的活动消费者。可以向主题目标发送消息的生产者的数量没有限制，并且

每个消息可以发送至任意数量的订阅消费者。主题目标也支持持久订阅的概念。持久订

阅表示消费者已向主题目标进行注册，但在消息传送时此消费者可以处于非活动状态。

当此消费者再次处于活动状态时，它将接收此信息。如果没有已经向主题目标注册的消



费者，主题不保留其接收到的消息，除非有非活动消费者注册了持久订阅。这两种消息传递模型使用表示不同编程域的 API 对象（其语义稍有不同）进行处理，如下所示：

基本类型（统一域）	PTP 域	Pub/Sub 域
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicPublisher
Destination (Queue 或 Topic)	Queue	Topic
MessageProducer	QueueSender	
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

使用图表第一列中列出的统一域对象编写点对点和发布/订阅消息传递。这是首选方法（JMS 1.1 规范）。然而，为了符合早期的 JMS 1.02b 规范，可以使用 PTP 域对象

编写点对点消息传递，使用 Pub/Sub 域对象编制发布/订阅消息传递。

JMS 消息结构

JMS 消息由以下几部分组成：消息头，属性和消息体。

1.1.5. 消息头(Header)

消息头包含消息的识别信息和路由信息，消息头包含一些标准的属性如：

JMSDestination, JMSSessageID 等。

如何设置消息头的字段：

消息头	由谁设置
JMSDestination	send方法
JMSDeliveryMode	send方法
JMSExpiration	send方法
JMSPriority	send方法
JMSSessageID	send方法
JMSTimestamp	客户端
JMSCorrelationID	客户端
JMSReplyTo	客户端
JMSType	客户端
JMSRedelivered	JMS Provider

标准的 JMS 消息头包含以下属性：



消息头	描述	分配方式
JMSDestination	消息发送的目的地：主要是指Queue和Topic。	自动
JMSDeliveryMode	传送模式有两种模式：持久模式和非持久模式。一条持久性的消息应该被传送“一次仅仅一次”，这就意味者如果JMS提供者出现故障，该消息并不会丢失，它会在服务器恢复之后再次传递。一条非持久的消息最多会传送一次，这意味这服务器出现故障，该消息将永远丢失。	自动
JMSExpiration	消息过期时间，等于Destination 的send方法中的timeToLive值加上发送时刻的GMT 时间值。如果timeToLive值等于零，则JMSExpiration 被设为零，表示该消息永不过期。如果发送后，在消息过期时间之后消息还没有被发送到目的地，则该消息被清除。	自动
JMSPriority	消息优先级，从0-9 十个级别，0-4 是普通消息，5-9 是加急消息。JMS 不要求JMS Provider 严格按照这十个优先级发送消息，但必须保证加急消息要先于普通消息到达。默认是4级。	自动
JMSMessageID	唯一识别每个消息的标识，由JMS Provider 产生。	自动
JMSTimestamp	一个JMS Provider在调用send()方法时自动设置的。它是消息被发送和消费者实际接收的时间差。	自动
JMSCorrelationID	用来连接到另外一个消息，典型的应用是在回复消息中连接到原消息。在大多数情况下，JMSCorrelationID用于将一条消息标记为对JMSMessageID表示的上一条消息的应答，不过，JMSCorrelationID可以是任何值，不仅仅是JMSMessageID。	开发者设置
JMSTimestamp	一个消息被提交给JMS	自动



	Provider 到消息被发出的时间。	
JMSReplyTo	提供本消息回复消息的目的地址。	开发者设置
JMSType	消息类型的识别符。	开发者设置
JMSRedelivered	如果一个客户端收到一个设置了 JMSRedelivered 属性的消息，则表示可能客户端曾经在早些时候收到过该消息，但并没有签收(acknowledged)。如果该消息被重新传送，JMSRedelivered=true 反之，JMSRedelivered =false。	自动

1.1.6. 消息体(Body)

JMS API 定义了5 种消息体格式，也叫消息类型，可以使用不同形式发送接收数据并可以兼容现有的消息格式，下面描述这5 种类型：

消息类型	消息体
TextMessage	java.lang.String 对象，如xml 文件内容。
MapMessage	名/值对的集合，名是String 对象，值类型可以是Java 任何基本类型。
BytesMessage	字节流。
StreamMessage	Java 中的输入输出流。
ObjectMessage	Java 中的可序列化对象。
Message	没有消息体，只有消息头和属性。

1.1.7. 消息属性

包括以下三中类型的属性

A. 应用程序特定的属性。例如：

```
TextMessage message=session.createTextMessage();
```

```
Message.setStringProperty("username",username);
```

B. JMS 定义的属性

JMS 保留了“JMSX”作为JMS 属性名的前缀。新的JMS 定义的属性可能在后续版本中增



加。除非说明否则支持这些属性是可选的。`ConnectionMetaData.getJMSXPropertyNames()`方法返回所有连接支持的JMSX 属性的名字。无论连接是否支持JMSX 属性，它们都可以在消息选择器中使用。如果消息中没有这些属性，那么它们与其他缺席属性一样看待。在特定消息中，存在JMS 定义的属性，它们是由JMS 提供商根据如何控制属性的使用来设置的。根据管理或其它规则，可以在某些消息中包含它们在其他消息中忽略它们。

JMS 定义的属性

名字	类型	设置者	用法
JMSXUserID	String	发送时提供商设置	发送消息的用户标识
JMSXAppID	String	发送时提供商设置	发送消息的应用标识
JMSXDeliveryCount	int	发送时提供商设置	转发消息重试次数，第一次是1，第二次是2，...
JMSXGroupID	String	客户端	消息所在消息组的标识
JMSXGroupSeq	int	客户端	组内消息的序号第一个消息是1，第二个是2，...
JMSXProducerTXID	String	发送时提供商设置	产生消息的事务的事务标识
JMSXConsumerTXID	String	接收时提供商设置	消费消息的事务的事务标识
JMSXRcvTimestamp	long	接收时提供商设置	JMS 转发消息到消费者的时间
JMSXState	int	提供商	假定存在一个消息仓库，它存储了每个消息的单独拷贝，且这些消息从原始消息被发送时开始。每个拷贝的状态有：1（等待），2（准备），3（到期）或4（保留）。由于状态与生产者和消费者无关，所以它不是由它们来提供。它只和在仓库中查找消息相关，因此JMS 没有提供这种API。

C. 提供者特定的属性



1.1.8. 消息的确认

如果会话是事务性的，那么消息确认自动由commit 处理，且恢复自动由 rollback 处理。如果会话不是事务性的，有三个确认选择，且手工处理恢复。

- DUPS_OK_ACKNOWLEDGE——这个选项告诉会话懒惰确认消息的传递。如果JMS 失败，这很可能造成传递重复消息，因此这个选项只用于可以忍受重复消息的消费者。它的好处是减少了会话为防止重复所做的工作。
- AUTO_ ACKNOWLEDGE——使用这个选项，当消息被成功地从调用接收返回或处理消息的MessageListener 成功返回时，会话自动确认客户端的消息接收。
- CLIENT_ ACKNOWLEDGE——使用这个选项，客户端通过调用消息的acknowledge 方法来确认消息。确认一个被消费的消息会自动确认被该会话转发的所有消息。当使用CLIENT_ ACKNOWLEDGE 模式时，客户端可以在处理它们时产生大量未确认消息。JMS 提供商应当为管理员提供限制客户端超量运行的途径，以便客户端不会造成资源耗尽并保证当它们使用的资源被临时阻塞时造成失败。会话的recover 方法用于停止一个会话然后使用第一个未确认消息来重新启动它。事实上，会话的被转发消息序列被重新设置到最后一个确认消息之后。现在转发的消息序列可以与起初转发的消息序列不同，因为消息到期和收到更高优先级的消息。会话必须设置消息的redelivered 标记，表示它是由恢复而被重新转发。

PTP 模型

PTP(Point-to-Point)模型是基于队列的，生产者发消息到队列，消费者从队列接收消息，队列的存在使得消息的异步传输成为可能。和邮件系统中的邮箱一样，队列可

以包含各种消息，JMS Provider 提供工具管理队列的创建、删除。JMS PTP 模型定义

了客户端如何向队列发送消息，从队列接收消息，浏览队列中的消息。

下面描述 JMS PTP 模型中的主要概念和对象：

名称	描述
ConnectionFactory	客户端用 ConnectionFactory 创建 Connection 对象。
Connection	一个到 JMS Provider 的连接，客户端可以用
Session	创建 Session 来发送和接收消息。
MessageConsumer	客户端用 Session 创建 MessageProducer 和
Destination (Queue 或 Topic)	对象。如果在 Session 关闭时，有一些消息已经被收到，但还没有被签收 (acknowledged)，那么，当消费者下次连接到相同的队列时，这些消息还会被再次接收。
Destination (Queue 或 Topic)	客户端用 Session 创建 Destination 对象。



TemporaryQueue	象。此处的目标为队列，队列由队列名识别。临时队列只能由创建它的 Connection 所创建的消费者消费，但是任何生产者都可向临时队列发送消息。
MessageProducer	客户端用 MessageProducer 发送消息到队列。
MessageConsumer	客户端用 MessageConsumer 接收队列中的消息，如果用户在 receive 方法中设定了消息选择条件，那么不符合条件的消息会留在队列中，不会被接收到。
可靠性(Reliability)	队列可以长久地保存消息直到消费者收到消息。消费者不需要因为担心消息会丢失而时刻和队列保持激活的连接状态，充分体现了异步传输模式的优势。

PUB/SUB 模型

JMS Pub/Sub 模型定义了如何向一个内容节点发布和订阅消息，这些节点被称作主题(topic)。

主题可以被认为是消息的传输中介，发布者(publisher)发布消息到主题，订阅者(subscribe) 从主题订阅消息。主题使得消息订阅者和消息发布者保持互相独立，不需要接触即可保证消息的传送。

下面描述 JMS Pub/Sub 模型中的主要概念和对象：

名称	描述
订阅(subscription)	消息订阅分为非持久订阅(non-durablesubscription)和持久订阅(durablesubscription)，非持久订阅只有当客户端处于激活状态，也就是和JMS Provider 保持连接状态才能收到发送到某个主题的消息，而当客户端处于离线状态，这个时间段发到主题的消息将会丢失，永远不会收到。持久订阅时，客户端向JMS 注册一个识别自己身份的 ID，当这个客户端处于离线时，JMS Provider 会为这个ID 保存所有发送到主题的消息，当客户再次连接到JMS Provider 时，会根据自己的ID 得到所有当自己处于离线时发送到主题的消息。
ConnectionFactory	客户端用 ConnectionFactory 创建 Connection 对象。
Connection	一个到 JMS Provider 的连接，客户端



	可以用 Connection 创建 Session 来发送和接收消息。
Session	客户端用 Session 创建 MessageProducer 和 MessageConsumer 对象。它还提供持久订阅主题，或使用 unsubscribe 方法取消消息的持久订阅。
Destination (Topic和TemporaryTopic)	客户端用 Session 创建 Destination 对象。此处的目标为主题，主题由主题名识别。临时主题只能由
创建它的 Connection 所创建的消费者消费。临时主题不能提供持久订阅功能。 JMS 没有给出主题的组织和层次结构的定义，由 JMS Provider 自己定义。	
MessageProducer	客户端用 MessageProducer 发布消息到主题。
MessageConsumer	客户端用 MessageConsumer 接收发布到主题上的消息。可以在 receive 中设置消息过滤功能，这样，不符合要求的消息不会被接收。
恢复和重新派送(Recovery andRedelivery)	非持久订阅状态下，不能恢复或重新派送一个未签收的消息。只有持久订阅才能恢复或重新派送一个未签收的消息。
可靠性(Reliability)	当所有的消息必须被接收，则用持久订阅模式。当丢失消息能够被容忍，则用非持久订阅模式

。

2.8 JMS 支持并发

JMS	对象是否支持并发
Destination	是
ConnectionFactoRy	是
Connection	是
Session	否
MessageProducer	否
MessageConsumer	否



第二章. 如何配置 ActiveMQ

2. ActiveMQ 的链接

如何配置传输链接

2.1.1. 格式配置如下：

```
<!-- The transport connectors ActiveMQ will listen to -->
<transportConnectors>
    <transportConnector name="openwire" uri="tcp://localhost:61616"
        discoveryUri="multicast://default"/>
    <transportConnector name="ssl"      uri="ssl://localhost:61617"/>
    <transportConnector name="stomp"   uri="stomp://localhost:61613"/>
    <transportConnector name="xmpp"    uri="xmpp://localhost:61222"/>
</transportConnectors>
```

2.1.2. Transmission Control Protocol(TCP)

Transmission Control Protocol(TCP)对于人类可能是最重要的传输协议。作为 Internet Protocols 基础，几乎所有的在线通信都适用了 TCP Pocotol。

ActiveMQ 的 Broker 和 Client 之间需要一个高稳定性的通信，不难发现，TCP 是一个理想的实现。所以我们并不惊讶在 ActiveMQ 频繁的使用 TCP 协议。

在做数据交互之前，我们需要知道必须序列化数据，消息是如何通过一个叫 wire protocol 的来序列化成字节流。默认情况下，**ActiveMQ 把 wire protocol 叫做 OpenWire**。它的目的是促使网络上的效率和数据快速交互等。

默认的 Broker 配置，TCP 的 Client 监听端口是 61616。

```
tcp://hostname:port?key=value&key=value
```

黑体部分是必须写的，后面的是一些选项参数。

是 ActiveMQ 最常用的传输协议。

经常应用于可靠性高，稳定性强的场景中。例如：Email。



它包括以下优点：

1. 高效性：字节流方式传递，效率很高。
2. 有效性、可用性：应用广泛，支持任何平台。
3. 可靠性、稳定性：数据不会丢失。

2.1.3. New I/O API Protocol (NIO)

New I/O(NIO)API 已经在 java SE 1.4 里已经支持了在 util 包中已经存在的 I/O API。NIO 并不是传统的 I/O API。

NIO 协议和 TCP 协议类似，nio 更侧重于底层的访问操作。它不同与传统的 I/O 流操作。允许开发人员对同一资源可有更多的 client 调用和服务端有更多的负载。

从 Client 来看，NIO 是和标准的 TCP 很类似，尽管 TCP 是应用在 network protocol 之下和 OpenWire 协议作为消息序列化协议。

这里以下场合使用 NIO 更为合适：

- 你可能有大量的 Client 去链接到 Broker 上。

一般情况下，大量的 Client 去链接 Broker 是被操作系统的线程数所限制的。因此，NIO 的实现比 TCP 需要更少的线程去运行，所以，建议你使用 NIO 协议。

- 你可能对于 Broker 有一个很迟钝的网络传输。

NIO 比 TCP 提供更好的性能。



2.1.3.1. 格式配置如下：

```
nio://hostname:port?key=value
```

Now take a look at the following configuration snippet:

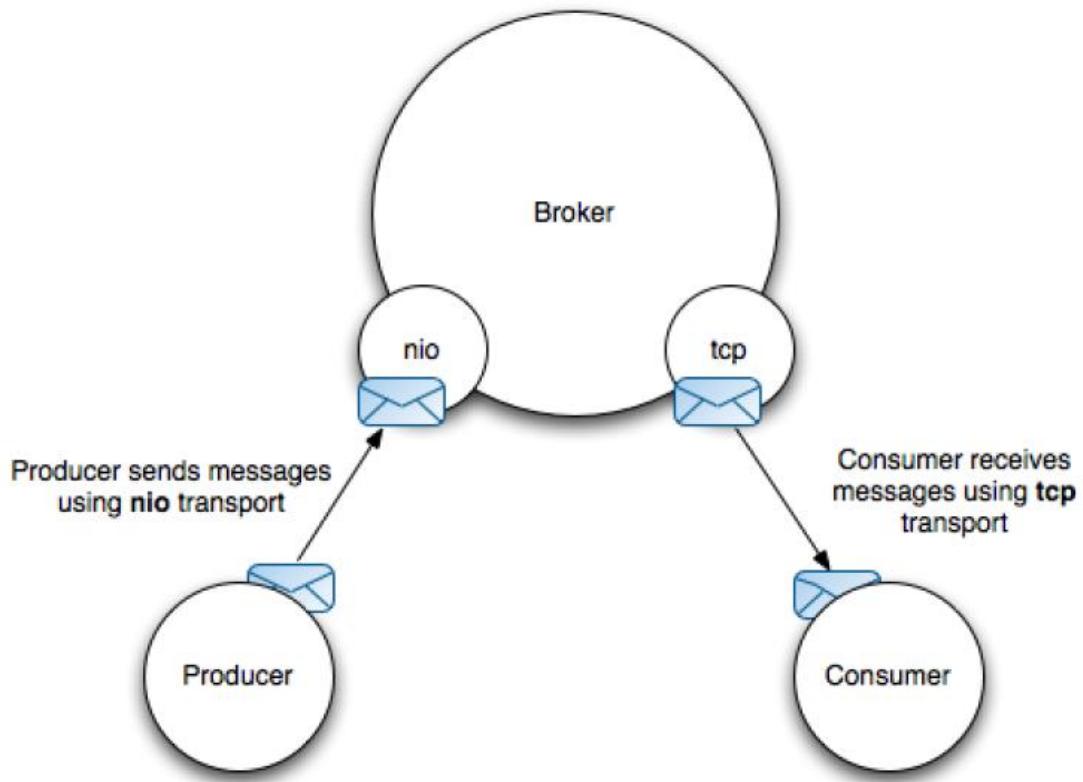
```
<transportConnectors>
    <transportConnector
        name="tcp"
        uri="tcp://localhost:61616?trace=true" />❶
    <transportConnector
        name="nio"
        uri="nio://localhost:61618?trace=true" />❷
</transportConnectors>
```

一个 TCP 协议监听 61616 端口

一个 NIO 协议监听 61618 端口

运行原理如下图：

Producer 通过 NIO 协议发送数据到 broker，Consumer 通过 TCP 协议接收数据。





2.1.4. User Datagram Protocol (UDP)

UDP 协议和 TCP 协议组成了 Internet protocols。这两种协议的初衷是相同的，即通过网络发送和接受数据包。以下有两个方面的不同：

- TCP 是一个原始流的传递协议，意味着数据包是有保证的，换句话说，数据包是不会被复制和丢失的。UDP，另一方面，它是不会保证数据包的传递的。
- TCP 也是一个稳定可靠的数据包传递协议，意味着数据在传递的过程中不会被丢失。这样确保了在发送和接收之间能够可靠的传递。相反，UDP 仅仅是一个链接协议，所以它没有可靠性之说。

所以，从上面可以得出：TCP 是被用在稳定可靠的场景中使用的，然而，UDP 通常用在快速数据传递和不怕数据丢失的场景中使用的。

配置如下：

```
udp://hostname:port?key=value
```

何时使用 UDP？

- ActiveMQ 通过防火墙时，你只能用 UDP。
- 如果你想尽可能的减少传递延迟，快速的传递数据。

2.1.4.1. 格式配置如下：

```
<transportConnectors>
    <transportConnector
        name="tcp"
        uri="tcp://localhost:61616?trace=true"
    />
    <transportConnector
        name="udp"
        uri="udp://localhost:61618?trace=true"
    />
</transportConnectors>
```

2.1.5. TCP、UDP 的区别：

主要有两个方面的不同：



1. TCP 是一个原始的流协议，意味着数据包的传递是有保证的。UDP 则没有保证。
2. TCP 同样也是一个可靠性高的传递协议。意味着数据包不会丢失，反之，UDP 则不能保证。

2.1.6. Secure Sockets Layer Protocol (SSL)

基于 TCP 之上的安全协议

2.1.7. Hypertext Transfer Protocol (HTTP/HTTPS)

在很多场合，像 web 和 email 等服务需要通过防火墙来访问的，所以，http 可以使用这种场合。Hypertext Transfer Protocol(HTTP)当初是被设计用来传递 hypertext(HTML)pages 基于 web。http 协议是建立在 TCP 协议之上和添加了一些额外的逻辑来通信的。在 Internet 普及后，web 框架和 http 协议将用于 web services 的应用。在 web 服务应用中的不同主要是传递 xml 格式的数据通过 HTTP 协议。

书写格式：

```
http://hostname:port?key=value
```

配置格式如下：

```
<transportConnectors>
    <transportConnector
        name="tcp"
        uri="tcp://localhost:61616?trace=true"
    />
    <transportConnector
        name="http"
        uri="http://localhost:8080?trace=true"
    />
</transportConnectors>
```

2.1.8. VM Protocol (VM)

VM transport 允许在 VM 内部通信，从而避免了网络传输的开销。这时候采用的连接不是 socket 连接，而是直接地方法调用。第一个创建 VM 连接的客户会启动一个 embed VM broker，接下来所有使用相同的 broker name 的 VM 连接都会使用这个 broker。当这个 broker 上所有的连接都关闭的时候，这个 broker 也会自动



关闭。

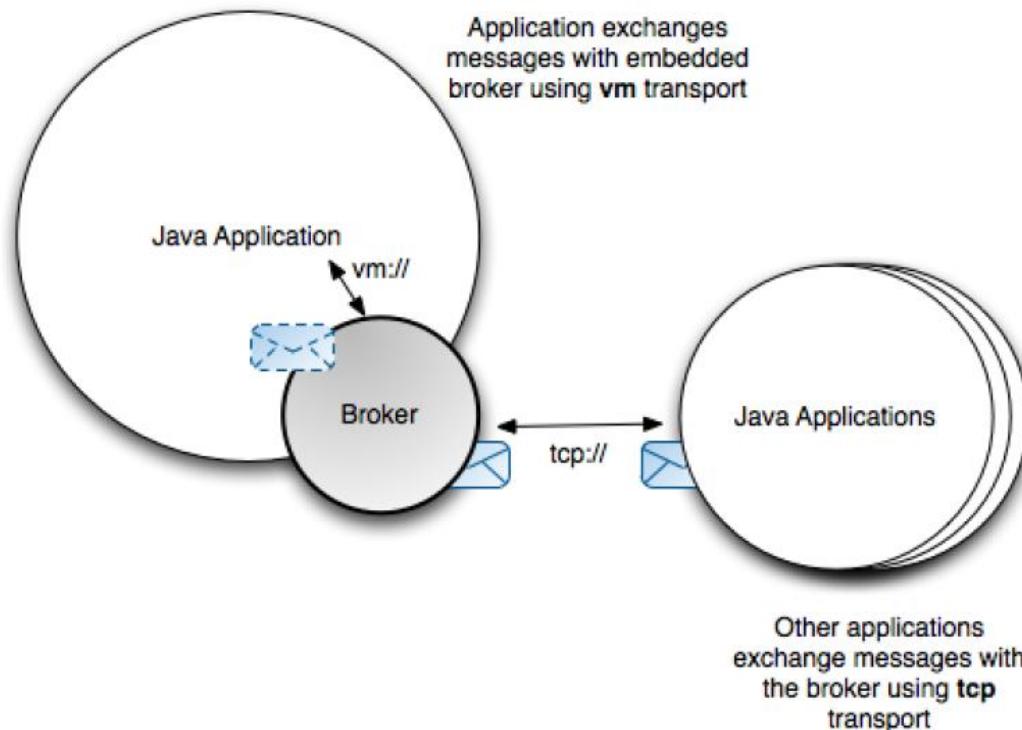
格式配置如下：

```
vm://brokerName?transportOptions
```

Java 中嵌入是方式：

```
vm:broker:(tcp://localhost:6000)?brokerName=embeddedbroker&persistent=false
```

定义了一个嵌入的 broker 名称为 embeddedbroker 以及配置了一个 tcptransprotconnector 在监听端口 6000 上。



使用一个加载一个配置文件来启动 broker。

```
vm://localhost?brokerConfig=xbean:activemq.xml
```

属性：

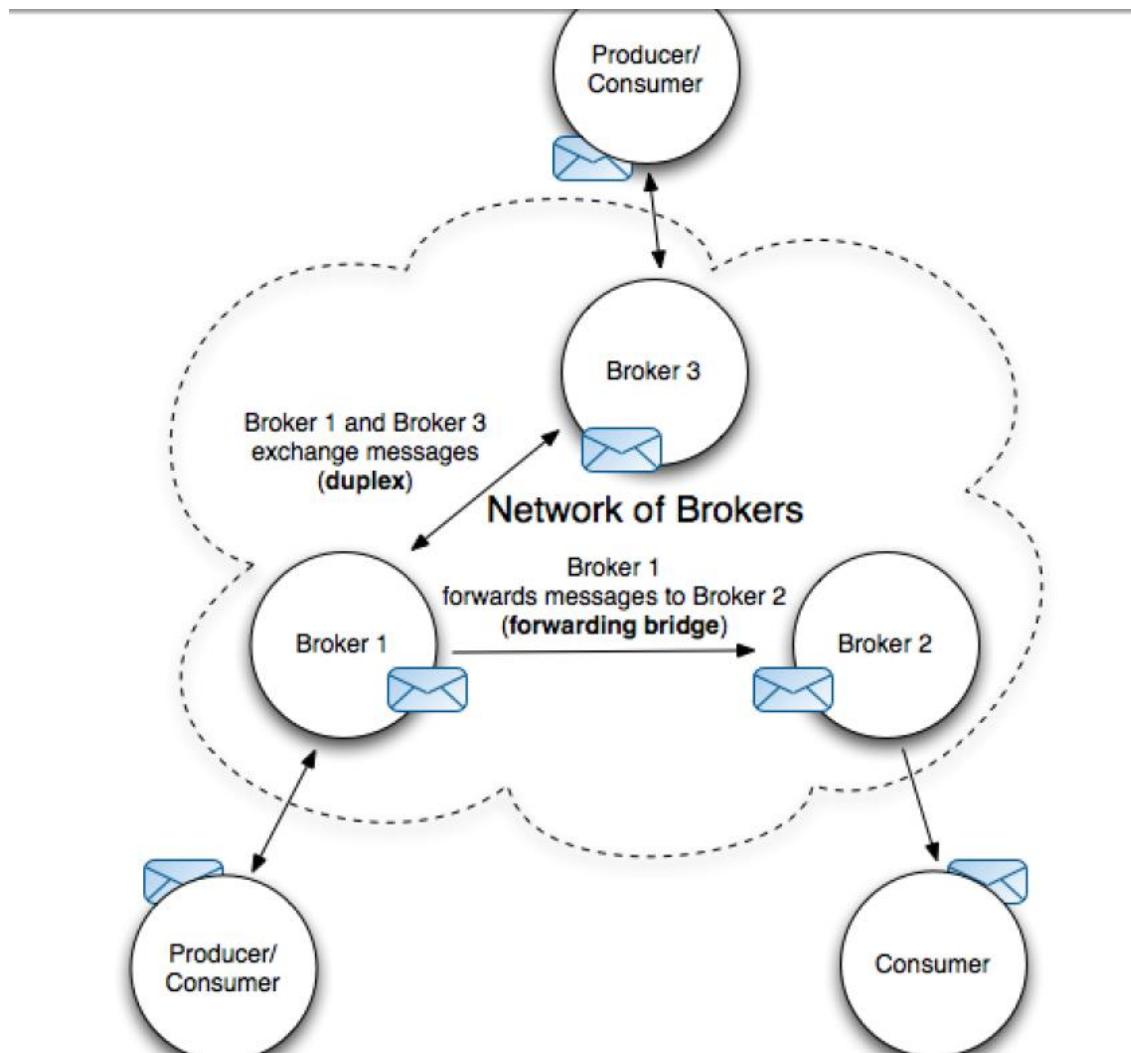
Option Name	Default Value	Description
-------------	---------------	-------------



Marshal	false	If true, forces each command sent over the transport to be marshalled and unmarshalled using a WireFormat
wireFormat	default	The name of the WireFormat to use
wireFormat.*		All the properties with this prefix are used to configure the wireFormat
create	true	If the broker should be created on demand if it does not already exist. Only supported in ActiveMQ 4.1
broker.*		All the properties with this prefix are used to configure the broker. See Configuring Wire Formats for more information

如何配置网络连接

一个 ActiveMQ Broker 对于你的所有的应用来说是非常实用的。但是，某些场景需要一些高级的特性。例如：高稳定性和大传输量。典型的就是用在网络中。Broker 集群在多个 ActiveMQ 的实例中大大优化了消息的传递。这章主要是解释网络链接在 ActiveMQ 的应用，Broker 到 Broker 的通信。网络链接是以通道的形式将一个 Broker 和其他的 Broker 链接起来通信。网络链接默认是单向的，然而，一个 Broker 在一端发送消息，在另一 Broker 在另一端接收消息。这就是所谓的“桥接”。然而，你可能想创建一个双向的通道对于两个 Broker。他将不仅发送消息而且也能从相同的通道来接收消息。ActiveMQ 支持这种双向链接，通常作为 duplex connector 来映射。如下图：



网络链接配置通过 ActiveMQ XML 配置如下：

```
<!-- The store and forward broker networks ActiveMQ will listen to -->
<networkConnectors>
    <!-- by default just auto discover the other brokers -->
    <networkConnector name="default-nc" uri="multicast://default"/>
    <!--
    <networkConnector name="host1 and host2"
        uri="static://(tcp://host1:61616,tcp://host2:61616)"/>
    -->
</networkConnectors>
```

Broker 的网络链接配置用<networkConnector>元素，它包括一个以上的链接用<networkConnector>元素表示。主要是 name 和 uri。其他的属性被用于在链接上的附加特性。

这一章中，各种 ActiveMQ 的协议和技术将讨论如何用于配置 ActiveMQ 的网络



链接。在我们开始前，我要重点介绍一个关于 ActiveMQ 的一个术语概念“discovery”。一般情况下，discovery 是被用来发现远程的服务。客户端通常想去发现所有可利用的 brokers，另一方的解释，它是基于现有的网络 Broker 去发现其他可用的 Brokers。

这里用两种配置 Client 到 Broker 的链接方式，一种方式：Client 通过 Statically 配置的方式去连接 Broker，一种方式：Client 通过 discovery agents 来 dynamically 的发现 Brokers。

2.1.9. 如何配置 Dynamic Networks

2.1.9.1. Multicast 协议

Multicast：多点传播

ActiveMQ 使用 Multicast 协议将一个 Service 和其他的 Broker 的 Service 连接起来。IP multicast 是一个被用于网络中传输数据到其它一组接收者的技术。Ip multic 传统的概念称为组地址。组地址是 ip 地址在 224.0.0.0 到 239.255.255.255 之间的 ip 地址。

ActiveMQ broker 使用 multicast 协议去建立服务与远程的 broker 的服务的网络链接。

格式配置如下：

```
multicast://address:port?transportOptions
```

Transport Options 属性

Option Name	Default Value	Description
group	default	表示唯一的组名称



minimumWireFormatVersion	0	The minimum version wireformat that is allowed
trace	false	Causes all commands that are sent over the transport to be logged
useLocalHost	true	如果 true , 表是本地机器的名称为 localhost
datagramSize	4 * 1024	特定的数据大小
timeToLive	-1	消息的生命周期
loopBackMode	false	是否启用 loopback 模式
wireFormat	default	默认用 wireFormat 命名
wireFormat.*		前缀是 wireFormat

Uri 的例子

Default

请注意， 默认情况下是不可靠的多播， 数据包可能会丢失。

```
multicast://default
```

特定的 ip 和端口

```
multicast://224.1.2.3:6255
```

特定的 ip 和端口以及组名

```
multicast://224.1.2.3:6255?group=mygroupname
```



Activemq 默认的使用 multicast 协议的配置格式如下：

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="multicast"
    dataDirectory="${activemq.base}/data">

    <networkConnectors>

        <networkConnector name="default-nc" uri="multicast://default"/>
    </networkConnectors>

    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61616"
            discoveryUri="multicast://default"/>
    </transportConnectors>

</broker>
```

以上图示中，“default” =是 activemq 默认的 ip， 默认动态的寻找地址。

以下地址具体的描述了该协议的内容：

<http://activemq.apache.org/multicast-transport-reference.html>

这里有两个很重要的属性：

“discoveryUri” =是指在 transport 中用 multicast 的 default 的地址传递。

“uri” =指动态寻找可利用的地址。

防止自动的寻找地址

- 1) 名称为 openwire 的 transport， 移除 discoveryUri="multicast://default" 即可。传输链接用默认的名称 openwire 来配置 broker 的 tcp 多点链接，这将允许其它 broker 能够自动发现和链接到可用的 broker 中。

```
<transportConnector name="openwire" uri="tcp://localhost:61616"
    discoveryUri="multicast://default"/>
```

去停止基于 multicast 的链接协议， 仅仅需要改变移除 discoveryUri 属性即可， 如下：

```
<transportConnector name="openwire" uri="tcp://localhost:61616" />
```

- 2) 名称为 “default-nc”的 networkConnector， 注释掉或者删除即可。



ActiveMQ 默认的 networkConnector 基于 multicast 协议的链接的默认名称是 default-nc，而且自动的去发现其他 broker。去停止这种行为，只需要注销或者删除掉 default-nc 网络链接。

```
<networkConnector name="default-nc" uri="multicast://default"/>
```

```
<!--networkConnector name="default-nc" uri="multicast://default"-->
```

- 3) 使 brokerName 的名字唯一。默认是 localhost。是为了唯一识别 Broker 的实例。

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost" dataDirectory="${activemq.base}/data">
```

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="broker1234" dataDirectory="${activemq.base}/data">
```

Multicast 协议和普通的 tcp 协议没有多大的不同，仅仅不同的是能够自动的发现其他 broker 从而替代了使用 static 功能列表 brokers。用 multicast 协议可以在网络中频繁的添加和删除 ip 不会有影响。

用这种 multicast 协议的好处是能够自动发现 broker，

优点：能够适应动态变化的地址。

缺点：自动的链接地址和过度的消耗网络资源。

2.1.9.2. Discovery 协议

Discovery 是在 multicast 协议的功能上定义的。功能类似与 failover 功能。它将动态的发现 multicast 协议的 broker 的链接并且随机的链接其中一个 broker。

这种协议的格式如下：

```
discovery: (discoveryAgentURI) ?key=value
```

具体的内容参考如下地址：

<http://activemq.apache.org/discovery-transport-reference.html>



格式配置如下：

discovery:(discoveryAgentURI)?transportOptions

or

discovery:discoveryAgentURI

Transport Options

Option Name	Default Value	Description
reconnectDelay	10	再次寻址等待时间
initialReconnectDelay	10	初始化设定再次寻址等待时间
maxReconnectDelay	30000	最大寻址等待时间
useExponentialBackOff	true	Should an exponential backoff be used between reconnect attempts
backOffMultiplier	2	The exponent used in the exponential backoff attempts
maxReconnectAttempts	0	如果异常，最大的重新链接个数
group	default	组唯一的地址

Example URI

```
discovery:(multicast://default)?initialReconnectDelay=100
```

以下默认的 xml 配置：

```
<broker name="foo">
  <transportConnectors>
```



```
<transportConnector uri="tcp://localhost:0"
discoveryUri="multicast://default"/>
</transportConnectors>

...
</broker>
```

2.1.9.3. Peer 协议

网络中 broker 的嵌入将是一个非常好的概念，对于你根据你的应用需求来调整 broker。当然了，我们可能创建一个嵌入式的 broker，但是这将会对于配置 broker 将完全繁琐。然而，这就是为什么 ActiveMQ 提出了 peer transport connector 作为为你更加容易的去嵌入 broker 中网络中。它将创建一个优于 vm 链接的 p2p 网络链接。

默认格式如下：

```
peer://peergroup/brokerName?key=value
```

具体的内容参考如下地址：

<http://activemq.apache.org/peer-transport-reference.html>

当我们启动了用 peer 协议时，应用将自动的启动内嵌 broker。**但是它也将会自动的去配置其它 broker 来建立链接**，当然了，前提是必须属于一个组。

配置如下：

```
peer://groupa/broker1?persistent=false
```

生产者和消费者都各自链接到嵌入到自己应用的 broker，并且在在本地的同一个组名中相互访问数据。

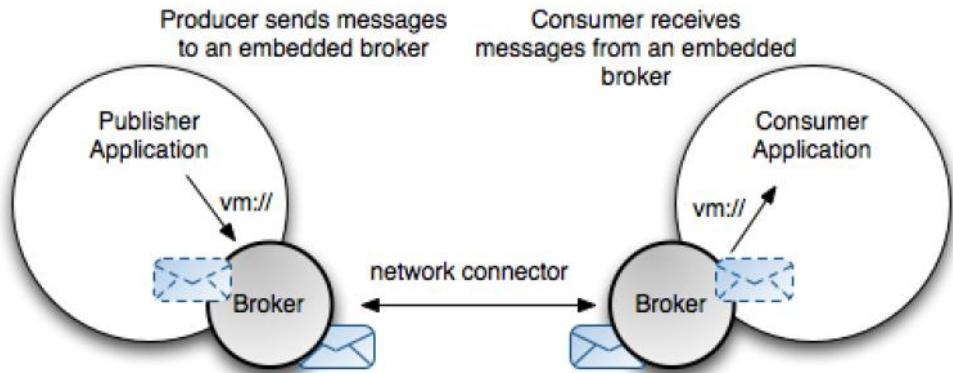


Figure 4.5. Two applications communicating using embedded brokers over peer protocol

在本地机器断网的情况下，本地的 client 访问本地 brokerA 将任然正常。在断网的情况下发送消息到本地 brokerA，然后网路链接正常后，所有的消息将重新发送并链接到 brokerB。

2.1.9.4. Fanout 协议

Fanout 协议是将同时链接多个 broker。

默认格式如下：

```
fanout: (fanoutURI) ?key=value
```

更多的内容参考如下地址：

<http://activemq.apache.org/fanout-transport-reference.html>

其中 fanoutURI，可以是个 static URI 或者是一个 multicast URI.

格式如下：

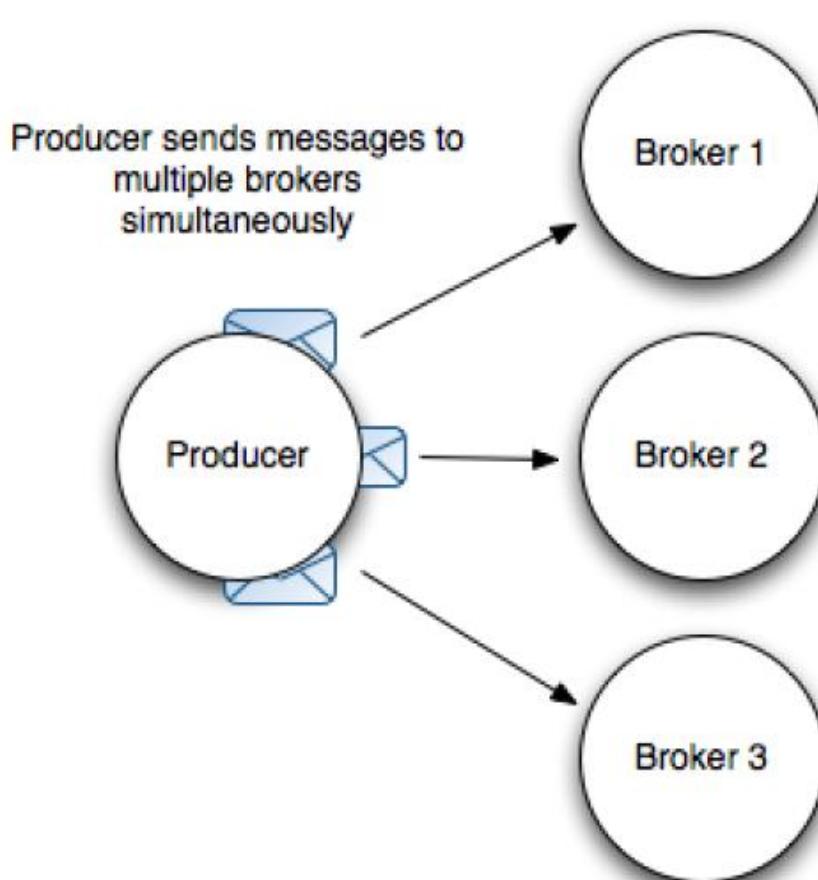
```
fanout: (static:(tcp://host1:61616,tcp://host2:61616,tcp://host3:61616))
```

以上示图说明，client 将试图链接到三个 static 列表中定义的三个 URI。

或者



```
fanout:(multicast://default)
```



配置方式如下：

```
fanout:(discoveryURI)?transportOptions
```

or

```
fanout:discoveryURI
```

Transport Options

Option Name	Default Value	Description



initialReconnectDelay	10	重新链接的等待时间
maxReconnectDelay	30000	最大重新链接的等待时间
useExponentialBackOff	true	Should an exponential backoff be used between reconnect attempts
backOffMultiplier	2	The exponent used in the exponential backoff attempts
maxReconnectAttempts	0	试图重新链接的个数
fanoutQueues	false	True, 表示 topic 消息转换 queue 消息
minAckCount	2	Broker 链接的最小数



Activemq 不推荐使 Consumer 使用 fanout 协议。当 Provider 发送消息到多个 broker 中，测试 Consumer 可能收到重复的消息。

Example URI

```
fanout:(static:(tcp://localhost:61616, tcp://remotehost:61616))?initialReconnectDelay=100
```

2.1.10. 如何配置 Static Networks

2.1.10.1. Static Protocol

Static network connector 是用于创建一个静态的配置对于网络中的多个 Broker。这种协议用于复合 url，一个复合 url 包括多个 url 地址。

格式如下：



```
static:(uri1,uri2,uri3,...)?key=value
```

你将从下面的地址中获得关于这种传输说明：

<http://activemq.apache.org/static-transport-reference.html>

如下例子：

```
<networkConnectors>
    <networkConnector name="local network"
        uri="static://(tcp://remotehost1:61616,tcp://remotehost2:61616)"/>
</networkConnectors>
```

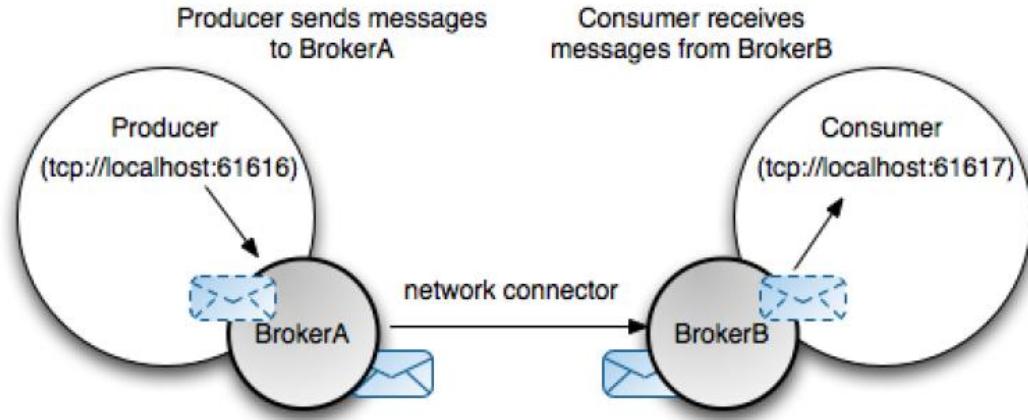
假设你配置 Broker 在 localhost 和 remotehost1 和 remotehost2 上运行，启动 broker 是你将看到如下信息：

```
...
INFO DiscoveryNetworkConnector - Establishing network connection between
from vm://localhost to tcp://remotehost1:61616
INFO TransportConnector - Connector vm://localhost Started
INFO DiscoveryNetworkConnector - Establishing network connection between
from vm://localhost to tcp://host2:61616
INFO DemandForwardingBridge - Network connection between vm://localhost#0
and tcp://remotehost1:61616 has been established.
INFO DemandForwardingBridge - Network connection between vm://localhost#2
and tcp://remotehost2:61616 has been established.
...
```

以上的输出指出了 localhost 已经成功的配置和其它两个 brokers 主机桥接。总之，消息发送到 localhost 此时将转发到 remotehost1 和 remotehost2 的 broker 上。

让我们论证以下关于 Static networks 用于股票的例子在网络桥接中。以下视图提供了一个关于 Broker 拓扑的看法。

运行原理如图：



关于以上视图，两个 Brokers 是网络链接的。Brokers 通过一个 static 的协议来创建一个链接。一个 Consumer 链接到 brokerB 的一个地址上，当 Producer 在 brokerA 上以相同的地址发送消息时，此时它将被转移到 brokerB 上。这种情况下，BrokerA 转发消息到 BrokerB 上。

让我们做个例子吧，首先我们需要创建两个网络 Brokers，让我们开始配置 BrokerB：

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="BrokerB"
        dataDirectory="${activemq.base}/data">

    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61617" />
    </transportConnectors>

</broker>
```

以上这个简单的配置启动 broker 在端口 61616 监听，

现在让我们配置 BrokerA：



```

<broker xmlns="http://activemq.apache.org/schema/core" brokerName="BrokerA"
        dataDirectory="${activemq.base}/data">

    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61616" />
    </transportConnectors>

    <networkConnectors>
        <networkConnector uri="static:(tcp://localhost:61617)" />
    </networkConnectors>

</broker>

```

以上配置 broker 监听端口在 61617 上, 它定义了一个 networks 链接到 BrokerB 上。

以上例子, 消息在 BrokerA 上被发送, 消息被转发到 BrokerB 上, 此时通过 Consumer 来消费该消息。

属性

Option Name	Default Value	Description
initialReconnectDelay	10	How long to wait before the first reconnect attempt (in ms)
maxReconnectDelay	30000	The maximum amount of time we ever wait between reconnect attempts (in ms)
useExponentialBackOff	true	Should an exponential backoff be used btween reconnect attempts
backOffMultiplier	2	The exponent used in the exponential backoff attempts
maxReconnectAttempts	0	If not 0, then this is the maximum number of reconnect attempts before an error is



		sent back to the client
minConnectTime	500	If a connection fails faster than this amount of time then it is considered a connection failure

2.1.10.2. Failover Protocol

以上例子，都是 Client 配置链接到指定的 broker 上。但是，如果 Broker 的链接失败怎么办呢？此时，你的 Client 有两个选项：要么立刻死掉要么去连接到其它的 broker 上。你可能猜想，股票例子运行在正中情况下是多么不稳定啊。

Failover 协议实现了自动重新链接的逻辑。这里有两种方式提供了稳定的 brokers 列表对于 Client 链接。第一种方式：提供一个 static 的可用的 Brokers 列表。第二种方式：提供一个 dynamic 发现的可用 Brokers。

其 failover 的 URI 的书写格式很类似与之前的 Static 网络链接 URI。这里有两种配置方式：

```
failover:(uri1,...,uriN) ?key=value
```

或者

```
failover:uri1,...,uriN
```

关于该协议的具体说明请参考如下地址：

<http://activemq.apache.org/failover-transport-reference.html>

默认情况下，这种协议用于随机的去选择一个链接去链接。如果链接失败了，那么会链接到其他的 Broker 上。默认的配置也将实现重新链接延迟逻辑，意味着传输将会在 10 秒后自动的去重新链接可用的 broker。当然了，所有的重新链接参数经根据你的需要而配置。

主要是指在 networks 失败时，自动链接其它 url。

Failover Transport 是一种重新连接的机制，它工作于其它 transport 的上层，用



于建立可靠的传输。它的配置语法允许制定任意多个复合的 URI。Failover transport 会自动选择其中的一个 URI 来尝试建立连接。如果没有成功，那么会选择一个其它的 URI 来建立一个新的连接。

配置格式如下：

```
failover:(tcp://localhost:61616)
```

属性：

属性

Option Name	Default Value	Description
initialReconnectDelay	10	How long to wait before the first reconnect attempt (in ms)
maxReconnectDelay	30000	The maximum amount of time we ever wait between reconnect attempts (in ms)
useExponentialBackOff	true	Should an exponential backoff be used btween reconnect attempts
backOffMultiplier	2	The exponent used in the exponential backoff attempts
maxReconnectAttempts	0	If not 0, then this is the maximum number of reconnect attempts before an error is sent back to the client
randomize	true	use a random algorithm to choose the the URI to use for reconnect from the list



		provided
backup	false	initialize and hold a second transport connection - to enable fast failover
timeout	-1	Enables timeout on send operations (in milliseconds) without interruption of reconnection process

对于重新链接机制的能力，他有很高的优点对于用 failover 协议。对于同一个地址重新的链接， broker 将会断掉。

```
failover:(tcp://localhost:61616)
```

对于 broker 失败 client 不需要重新去启动，而是自动的去连接其它可用的 broker。Failover 扮演一个非常重要的角色，例如：高稳定性和负载平衡。

3. 消息存储持久化

消息的持久化不仅支持 persistent 和 non-persistent 两种方式，而其 ActiveMQ 还支持消息的 recovery[恢复]方式。

对于 ActiveMQ 的消息存储必须知道它的存储原理是非常必要的，正如在它的配置和基于持久消息的传递必须清楚其原理 。消息发送到 Queue 和 Topic 的存储原理和结构是不同的。

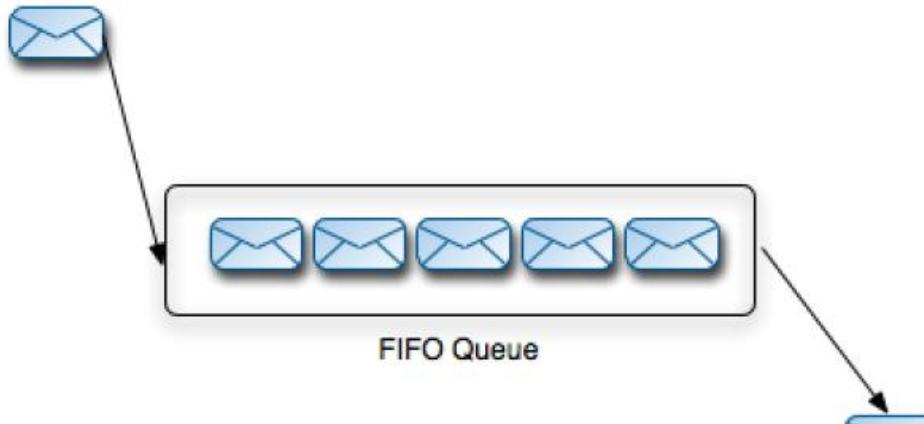
3.1.1. Point-to-point

Queue 的存储是很简单的。

如图下：



Message in



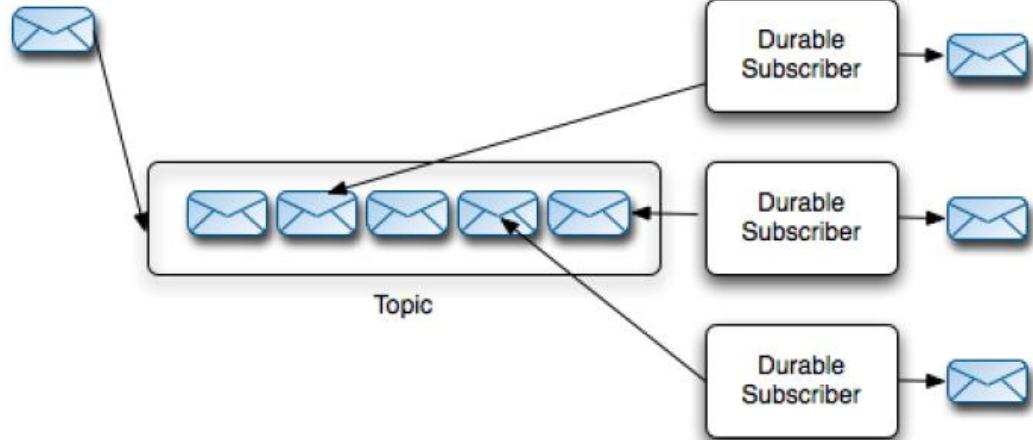
Message Out

3.1.2. Published/subscribe

对于持久化订阅主题，每一个消费者将获得一个消息的复制。

Message in

Messages Out



3.1.3. 有效的消息存储

ActiveMQ 提供了一个插件式的消息存储类似于消息的多点传播，主要实现了



如下几种存储：

- AMQ 消息存储-默认的消息存储
- KahaDB 消息存储-提供了容量的提升和恢复能力（5.3 以上采用）
- JDBC 消息存储-消息基于 JDBC 存储的。
- Memory 消息存储-基于内存的消息存储。

3.1.3.1. AMQ Message Store

AMQ Message Store 是 ActiveMQ5.0 缺省的持久化存储。它是一个基于文件、事务存储设计为快速消息存储的一个结构。AMQ 消息存储的初衷是尽可能简单的应用。它用了一个基于文件存储的消息数据库并且不依赖与第三方数据库。ActiveMQ 不会下载数据和不会运行很长时间。反之，AMQ 存储结构是以流的形式来进行消息交互的。

如果 AMQ 消息存储没有配置，那么它会使用默认的配置参数。选择性的使用消息存储或者改变 AMQ 消息存储的默认行为。当然了，一个<persistanceAdapter>元素必须被配置。

Message commands 被保存到 transactional journal (由 rolling data logs 组成)。Messages 被保存到 data logs 中，同时被 reference store 进行索引以提高存取速度。Date logs 由一些单独的 data log 文件组成，缺省的文件大小是 32M，如果某个消息的大小超过了 data log 文件的大小，那么可以修改配置以增加 data log 文件的大小。如果某个 data log 文件中所有的消息都被成功消费了，那么这个 data log 文件将会被标记，以便在下一轮的清理中被删除或者归档。以下是其配置的一个例子：

Xml 代码

```
1. <broker brokerName="broker" persistent="true" useShutdownHook="false">
2.   <persistanceAdapter>
3.     <amqPersistenceAdapter directory="${activemq.base}/data" maxFileLength="32mb"/>
4.   </persistanceAdapter>
```

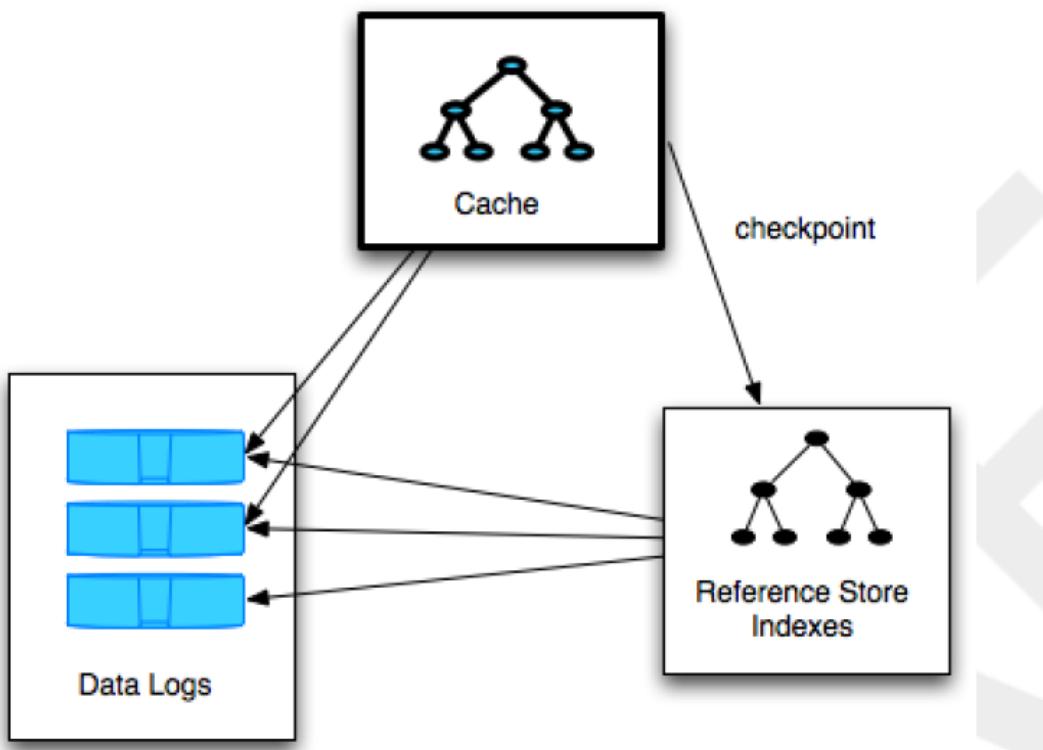


5. </broker>

它是 ActiveMQ 默认的消息存储，是在消息存储实现中 最快的消息存储了。它势必会产生快速事务持久、高优化消息索引 id 和内存消息缓存。

AMQ Message 内部存储结构

AMQ Message 存储结构图：



以上示图提供了一个 AMQ 存储的结构部分，主要包括以下：

对于理解基于 File 目录存储的 AMQ 存储很重要的，当我们使用 ActiveMQ 的时候，能够帮助去配置和解决问题。

1. Journal=包括一些基于消息和命令存储的回滚日志文件。当数据文件到达最大值时，新的数据文件将会被创建。数据文件中的消息都会被映射，如果一些消息如果不长时间使用的话，他将会被移除或者被存档。
Journal 仅仅是把当前的最新消息链接到该数据文件上即可，所以存储

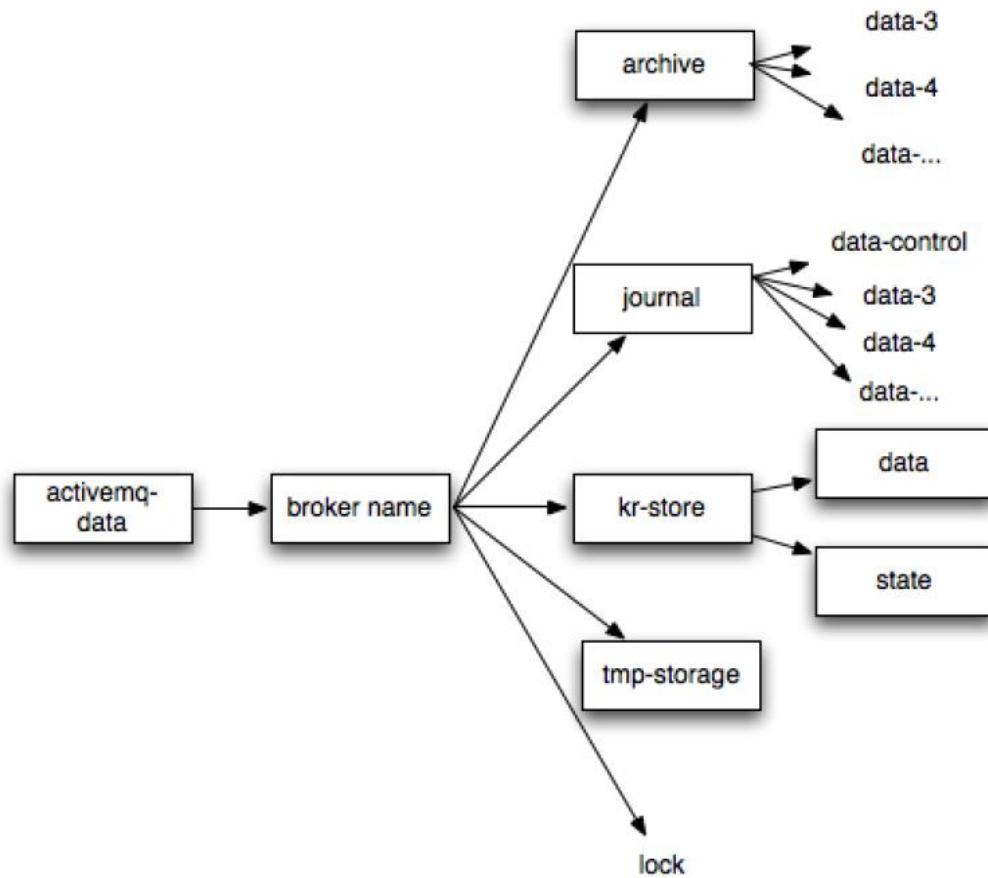


相当的快。

2. Cache=能够快速的是存储在 journal 目录中的数据恢复到内存中。Cache 还更新当前消息 id 和定位消息在 journal 中存储的最新映射。映射存储被更新后，消息才被安全的从缓存中移除。在缓存更新到映射存储这个范围段内我们需要配置属性 checkpointInterval。如果 ActiveMQ 消息 broker 在内存达到上限时 checkpoint 将会起作用。
3. Reference=在 journal 中通过消息的 id 来映射消息。它能够准确的从 FIFO 中映射其队列的数据结构和通过指针指向持久化订阅的主题消息。索引指针指的是 hash 索引。它也可以用一个内存 HashMap 类型的。

AMQ Message 存储目录结构

启动 ActiveMQ 时， 默认在本地创建一下目录结构。





⚠️ 警告:

每一个 **broker** 最好创建一个唯一的名称。

目录描述:

Data 目录=包括索引和在 **journal** 目录中用于映射消息的集合, 如果 **broker** 没有关闭彻底, 这个数据目录被删除和重建作为恢复的一部分。你会在启动 **broker** 之前通过手动的方式删除这个目录来恢复。

State 目录=包括持久化订阅的主题的消费者的信息。

Lock 目录=是在任何时候只确保仅仅一个 **broker** 能够一个进入数据库, 其他的 **broker** 等待。

Temp-storage 目录=存储 **non-persistent** 的消息数据。在 **broker** 的内存中不长期存储的消息, 这些消息是典型传递给慢消费者。

Kr-store 目录=主要是存储 **AMQ Message stroe** 的映射部分。它默认用 **Kaha** 数据库去索引和存储映射消息在 **journal** 目录中。

Journal 目录=存储数据的文件。数据文件是被映射的, 以至于所有的消息被传递, 数据文件也将会被删除和存档。

Data-control 目录=一些数据 **meta** 信息

Archive 目录=该目录是根据需要来使用的, 默认情况下地位于 **journal** 目录, 它是被用于一个独立部分或者磁盘, 该目录将用于存储来自 **journal** 目录下被移除而替换删除的消息。它将尽可能重载消息在 **archive** 目下。去重载消息移到 **archive** 目录下的数据日志到一个新的 **journal** 目录下和启动 **broker**, 此时 **broker** 将会自动重载数据在 **journal** 目录下。

属性

Property name	Default value	Comments
---------------	---------------	----------



directory	activemq-data	消息存储目录
useNIO	true	是否用 NIO 协议写入消息到 log 中
syncOnWrite	false	是否同步写入磁盘
maxFileLength	32mb	消息日志文件的最大存储
persistentIndex	true	用一个持久的消息索引，如果 false，这将会在内存中维护消息的索引
maxCheckpointMessageAddSize	4kb	the maximum number of messages to keep in a transaction before automatically committing
cleanupInterval	30000	多长时间 (ms) 检查不常用和移除的消息
indexBinSize	1024	default number of bins used by the index. The bigger the bin size – the better the relative performance of the index
indexKeySize	96	the size of the index key – the key is the message id
indexPageSize	16kb	the size of the index page – the bigger the page – the better the write performance of the index
directoryArchive	archive	the path to the directory to use to store discarded data logs
archiveDataLogs	false	if true data logs are moved to the archive directory instead of being deleted

Xml 配置方式：

AMQ Message Store 是 ActiveMQ5.0 缺省的持久化存储。Message commands 被保存到 transactional journal (由 rolling data logs 组成)。Messages 被保存到 data logs 中，同时被 reference store 进行索引以提高存取速度。Data logs 由一些单独的 data log 文件组成，缺省的文件大小是 32M，如果某个消息的大小超过了 data log 文件的大小，那么可以修改配置以增加 data log 文件的大小。如果某个 data log 文件中所有的消息都被成功消费了，那么这个 data log 文件将会被标记，以便在下一轮的清理中被删除或者归档。以下是其配置的一个例子：



Xml 代码

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker xmlns="http://activemq.apache.org/schema/core">
        <persistenceAdapter>
            <amqpersistenceAdapter
                directory="target/Broker2-data/activemq-data"
                syncOnWrite="true"
                indexPageSize="16kb"
                indexMaxBinSize="100"

                maxFileLength="10mb" />
        </persistenceAdapter>
    </broker>
</beans>
```

嵌入 java 代码方式:

```
public class EmbeddedBrokerUsingAMQStoreExample {

    public void main(String[] args) throws Exception {

        BrokerService broker = new BrokerService();❶

        PersistenceAdapterFactory persistenceFactory =
            new AMQPersistenceAdapterFactory();❷
        persistenceFactory.setMaxFileLength(1024*16);
        persistenceFactory.setPersistentIndex(true);
        persistenceFactory.setCleanupInterval(10000);

        broker.setPersistenceFactory(persistenceFactory);❸

        broker.addConnector("tcp://localhost:61616");❹

        broker.start();❺
    }
}
```

1: 表示初始化 broker

2: 表示创建一个 AMQ Stroe 的实例

3: 设置 broker 使用 AMQ 作为存储结构

4: 创建 transprotconnector 的地址

5: 启动 broker

注意:



每一个 **ActiveMQ** 都实现了 **PersistenceAdapter** 这个类。

何时使用 AMQ 消息存储

AMQ 是 ActiveMQ 的默认消息存储，它提供了在执行中的负载平衡，事实上这种存储是已经在嵌入 broker 和配置 xml 的 broker 中是最理想的存储方式对于用户。

它对于独立的和嵌入式的 ActiveMQ 来说是由可靠的持久性依赖于持久的事务处理和高效的索引来存储使其最优化程度高

AMQ 存储的这用易操作性意味这它将被大部分应用所使用，从高输出应用到存储大数据量的消息。

3.1.3.2. KahaDB Message Store

提升了容量和恢复能力（5.3 以及以上）。KahaDB 用于任何情况下。

KahaDB 是一种新的消息消息存储，而且解决了 AMQ 的一些不足，提高了性能。AMQ 消息存储用两个分离的文件对于每一个索引和如果 broker 没有彻底关闭则恢复很麻烦，所有的索引文件需要重新构建，broker 需要遍历所有的消息日志文件。

为了克服以上限制，KahaDB 消息存储对于它的索引用一个事务日志和仅仅用一个索引文件来存储它所有的地址。不同于 AMQ。而且在生成环境测试链接数到 10000，而且每一个链接对应一个队列。

Kaha Persistence 是一个专门针对消息持久化的解决方案。它对典型的消息使用模式进行了优化。在 Kaha 中，数据被追加到 data logs 中。当不再需要 log 文件中的数据的时候，log 文件会被丢弃。以下是其配置的一个例子：

Xml 代码

```
1. <broker brokerName="broker" persistent="true" useShutdownHook="false">
2.           <persistenceAdapter>
3.               <kahaPersistenceAdapter directory="activemq-data" maxDataFileLength="33554432"/>
```

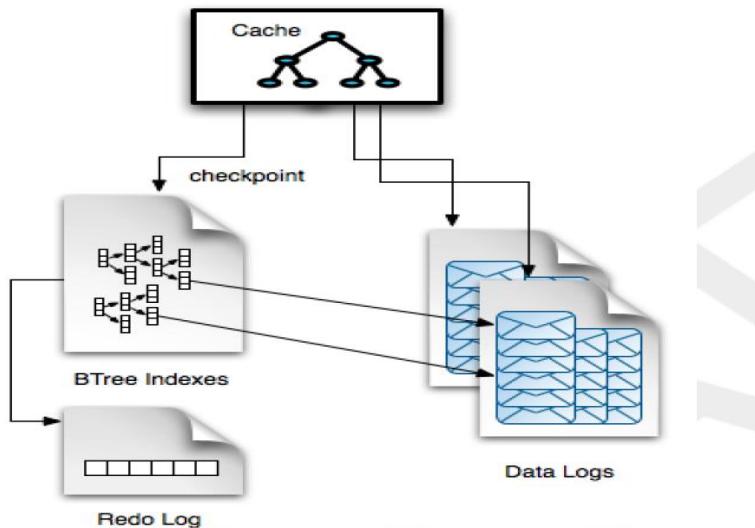


```

4.          </persistenceAdapter>
5.  </broker>

```

KahaDB 存储结构:



KahaDB 的存储结构和 AMQ 的存储结构类似。

也包括 Cache、Reference Indexes、message Journal，所有的索引文件更新的记录存在 Redo Log 中，这样就不用更新没有变化的索引数据了，仅仅更新变化的数据。额外的，KahaDB 消息存储用了一个 B-Tree 布局恰恰和 AMQ 消息存储相反，KahaDB 消息存储保持所有的索引在一个持久的 hash 表中，然而 hash 索引在时刻的变化，KahaDB 在这方面已经有了很好的性能特征。

KahaDB 配置方式如下：

```

<broker brokerName="broker" persistent="true"
useShutdownHook="false">
    <persistenceAdapter>
        <kahaDB directory="activemq-data"
journalMaxFileLength="32mb"/>
    </persistenceAdapter>
    <transportConnectors>
        <transportConnector uri="tcp://localhost:61616"/>
    </transportConnectors>

```



```
</broker>
```

KahaDB 属性

property name	default value	Comments
directory	activemq-data	the path to the directory to use to store the message store data and log files
indexWriteBatchSize	1000	number of indexes written in a batch
indexCacheSize	100	number of index pages cached in memory
enableIndexWriteAsync	false	if set, will asynchronously write indexes
journalMaxFileLength	32mb	a hint to set the maximum size of the message data logs
enableJournalDiskSyncs	true	ensure every non transactional journal write is followed by a disk sync (JMS durability requirement)
maxCheckpointMessageAddSize	4kb	the maximum number of messages to keep in a transaction before automatically committing
cleanupInterval	30000	time (ms) before checking for a discarding/moving message data logs that are no longer



		used
checkpointInterval	5000	time (ms) before checkpointing the journal

何时使用 KahaDB 消息存储

KahaDB 相比与 AMQ 消息存储能够应用到任何情况下。如果你需要一个高输出和大量的地址已经 broker 不超过 500 个，那么 AMQ 消息存储是最好的选择。

必须澄清 KahaDB 消息存储和 AMQ 消息存储是不完全一样的。两种之间是不能通过某种工具来互相转换的。如果你已经使用了一个 AMQ 消息存储，此时如果你想改变使用 KahaDB 存储，那么你必须移除 AMQ.

虽然 KahaDB 性能非常好，但是它并不能像 JDBC 存储那样简单实用，意味 KahaDB 实在是太新了。

3.1.3.3. JDBC Message Store

消息存储基于 JDBC。

ActiveMQ 插件式的消息存储具有的不同的消息存储实现而且具有很强的易用性。最常见的和最原始的消息存储 JDBC 存储也被 AactiveMQ 支持。

当我们使用 JDBC 消息存储默认的驱动使用 Apache Derby 数据库。同时也支持其它关系数据库： MySQL、Oracle、SQLServer、Sybase、Informix、MaxDB。

很多用户用一个关系数据库对于消息持久来说可以简单的查询去验证消息等功能。讨论以下一个话题：

在 ActiveMQ 中使用 Apache Derby。

Apache Derby 是一个 ActiveMQ 默认的数据库用于 JDBC 存储。只是因为它是一个很棒的数据库。不仅仅是它由 100%java 写的，而且它被设计成一个嵌入式的数据库。Derby 提供了一个很全的功能特征，性能很好和提供了一个很小容量，然



而，它对于 ActiveMQ 用户来说仅仅这能一个人可以使用，用 Derby 的感受，它在虚拟内存中提供了一个垃圾回收机制，它将代替在数据库中删除存储的消息，Derby 在它的 jvm 实例中允许 ActiveMQ 执行更加优化。

JDBC 消息存储提供了三张表，其中两种表是用于存储消息和第三张表是用于类似与排他锁似的，这样确保 ActiveMQ 仅仅由一个用户进入数据库。

消息表默认的名称 ACTIVEMQ_MSGS.

column name	default type	description
ID	INTEGER	The sequence id used to retrieve the message
CONTAINER	VARCHAR(250)	The destination of the message
MSGID_PROD	VARCHAR(250)	The id of the message producer
MSGID_SEQ	INTEGER	The producer sequence number for the message. This together with the MSGID_PROD is equivalent to the JMSMessageID
EXPIRATION	BIGINT	The time in milliseconds when the message will expire
MSG	BLOB	The serialized message itself

消息（队列和主题）存进 ACTIVEMQ_MSGS 表中。

ACTIVEMQ_ACKS 表存储持久订阅的信息和最后一个持久订阅接收的消息 ID。

column name	default type	description
CONTAINER	VARCHAR(250)	The destination
SUB_DEST	VARCHAR(250)	The destination of the



		durable subscriber (can be different from the container if using wild cards)
CLIENT_ID	VARCHAR(250)	The clientId of the durable subscriber
SUB_NAME	VARCHAR(250)	The subscriber name of the durable subscriber
SELECTOR	VARCHAR(250)	The selector of the durable subscriber
LAST_ACKED_ID	Integer	The sequence id of last message received by this subscriber

配置方式如下：

链接 MySQL 和 Oracle 两个 xml 配置如下：

图一：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker" xmlns="http://activemq.apache.org/schema/core">

        <persistenceAdapter>
            <jdbcPersistenceAdapter dataSource="#mysql-ds"/>
        </persistenceAdapter>
    </broker>

    <bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost/activemq?relaxAutoCommit=true"/>
        <property name="username" value="activemq"/>
        <property name="password" value="activemq"/>
        <property name="maxActive" value="200"/>
        <property name="poolPreparedStatements" value="true"/>
    </bean>
</beans>

```

图二：



```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker" xmlns="http://activemq.apache.org/schema/core">
        <persistenceAdapter>
            <jdbcPersistenceAdapter dataSource="#oracle-ds"/>
        </persistenceAdapter>
    </broker>

    <bean id="oracle-ds" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    
```

```

        <property name="url" value="jdbc:oracle:thin:@localhost:1521:AMQDB"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
        <property name="maxActive" value="200"/>
        <property name="poolPreparedStatements" value="true"/>
    </bean>

</beans>
```

3.1.3.4. JDBC Message Store with ActiveMQ Journal

它克服了 jdbc store 的不足。快速的缓存写入技术，大大提高了性能。

配置方式如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker" xmlns="http://activemq.apache.org/schema/core">
        <persistenceAdapter>
            <journalizedJDBC dataDirectory="${activemq.base}/data" dataSource="#derby-ds"/>
        </persistenceAdapter>
    </broker>

    <bean id="derby-ds" class="org.apache.derby.jdbc.EmbeddedDataSource">
        <property name="databaseName" value="derbydb"/>
        <property name="createDatabase" value="create"/>
    </bean>

</beans>
```

JDBC、 JDBC With Jouunal 区别：

1. Jdbc with journal 的性能优于 jdbc。
2. Jdbc 用于 master/slave 模式的数据库分享。
3. Jdbc with journal 不能用于 master/slave 模式。
4. 一般情况下， 推荐使用 jdbc with journal



何时使用 JDBC 消息存储

为什么使用 JDBC 消息存储最通俗的原因是关系数据库已经存在了。JDBC 持久的性能并不是很好，然而，用共享数据库是一个关于多个 brokers 的特殊的应用与 master/slave 拓扑结构。当一组 ActiveMQ brokers 是配置应用于一个共享数据库，此时他们将试图去链接数据库和获得一个排它锁。然而，仅仅一个 broker 成功链接才能成为一个 master。其他 broker 等待该 master 失败。这是比较常见的应用场景。

何时使用 JDBC 持久消息存储

JDBC message store with the journal

Journal 提供相当大的执行优势对于一个标准的 JDBC 消息存储，并且建议使用 JDBC 持久消息存储，仅仅当使用共享数据库的 master/slave 配置是不能使用 JDBC message store with the journal。例如：如果消息在提交之前被持久，那么该消息肯能会丢失。

3.1.3.5. Memory Message Store

内存消息存储主要是存储所有的持久化的消息在内存中。这里没有动态的缓存存在，所以你必须注意设置你的 broker 在 JVM 和内存限制。

消息存储基于 Memory，所有的消息都存储在内存里。

配置方式如下：

Xml 的配置方式：

将 broker 的属性 persistent 设置为 false 即可。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker"
        persistent="false"
        xmlns="http://activemq.apache.org/schema/core">
        <transportConnectors>
            <transportConnector uri="tcp://localhost:61635"/>
        </transportConnectors>
    </broker>
</beans>
```



Java 代码的配置方式：

```
import org.apache.activemq.broker.BrokerService;
public void createEmbeddedBroker() throws Exception {
    BrokerService broker = new BrokerService();
    //configure the broker to use the Memory Store
    broker.setPersistent(false);
    //Add a transport connector
    broker.addConnector("tcp://localhost:61616");
    //now start the broker
    broker.start();
}
```

3.1.4. KahaDB、AMQ 区别：

1. KahaDB 克服了 AMQ 的一些不足。
2. kahaDB 性能上优于 AMQ。
3. KahaDB 用于大量的 broker 【500 个以上】。
4. AMQ 用于独立和嵌入式的 broker 比较好。
5. AMQ 在执行的性能和索引方面都比较不错。
6. kahaDB 和 AMQ 两者是独立的，谁也不包括谁。

3.1.5. 何时使用内存消息存储

如果你的 broker 仅仅对于一组消息的消费是很快的，那么可以使用内存消息存储，但是他通常用于内部简单的消息测试，而不花费很多时间。或者能够在测试消息后清除消息。

第三章. 如何用 ActiveMQ 构建应用

4. 用 ActiveMQ 创建 JAVA 应用

这里主要将用 ActiveMQ Broker 作为独立的消息服务器来构建 JAVA 应用。ActiveMQ 也支持在 vm 中通信基于嵌入式的 broker。它能够无缝的集成其它 java 应用。



嵌入式 Broker 启动

4.1.1. Broker Service 启动 broker

配置方式如下：

```
BrokerService broker = new BrokerService();
broker.setUseJmx(true);
broker.addConnector("tcp://localhost:61616");
broker.start();
```

4.1.2. BrokerFactory 启动 broker

配置方式如下：

```
String Uri = "properties:resources/broker.properties";
//默认不自动启动。
// BrokerService broker1 = BrokerFactory.createBroker(Uri);
BrokerService broker1 = BrokerFactory.createBroker(new URI(Uri));
//表示自动启动。反之，不启动。
// BrokerService broker2 = BrokerFactory.createBroker(Uri,true);
broker1.addConnector("tcp://localhost:61616");
broker1.start();
```

利用 Spring 集成 Broker

如果你想把 ActiveMQ 嵌入到你的应用中的话，那么最后是用一种框架来开发，作为目前最流行的集成框架 Spring 框架。可以采用它来作为嵌入应用。ActiveMQ 集成了 Spring 框架。

java 配置方式如下：

```
protected static ApplicationContext context;
```



```

protected static BrokerService service;

private Connection connection;

public static void main(String args[])throws Exception {

    context = new
ClassPathXmlApplicationContext("resources/spring/spring2.xml");

    service = (BrokerService) context.getBean("broker");

    // already started

    service.start();
}

```

XML 配置方式如下：

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:amq="http://activemq.org/config/1.0"                      #1
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://activemq.org/config/1.0                         #2
                           http://activemq.apache.org/schema/activemq-core.xsd">      #2

<amq:broker
  brokerName="localhost" dataDirectory="${activemq.base}/data">

  <!-- The transport connectors ActiveMQ will listen to -->
  <amq:transportConnectors>
    <amq:transportConnector name="openwire"
      uri="tcp://localhost:61616" />
  </amq:transportConnectors>
  <amq:plugins>
    <amq:simpleAuthenticationPlugin>
      <amq:users>
        <amq:authenticationUser username="admin"
          password="password"
          groups="admins,publishers,consumers"/>
        <amq:authenticationUser username="publisher"
          password="password"
          groups="publishers,consumers"/>
        <amq:authenticationUser username="consumer"
          password="password"
          groups="consumers"/>
        <amq:authenticationUser username="guest"
          password="password"
          groups="guests"/>
      </amq:users>
    </amq:simpleAuthenticationPlugin>
  </amq:plugins>
</amq:broker>

```

这里通过 spring 来发送和介绍消息省略，因为 spring 提供了自己的 aip 基于 activemq。



5. ActiveMQ 嵌入到其他应用服务器中

指 Jboss、Jetty、Apache Tomcat、Weblogic，Apache Geronimo 等等。

//TODO

启动 ActiveMQ

有两种方式来启动：

第一种：Embedded Broke

可以通过在应用程序中以编码的方式来启动 broker，例如：

Java 代码

1. BrokerService broker = new BrokerService();
2. broker.addConnector("tcp://localhost:61616");
3. broker.start();

如果需要启动多个 broker，那么需要为 broker 设置一个名字。例如：

Java 代码

1. BrokerService broker = new BrokerService();
2. broker.setName("fred");
3. broker.addConnector("tcp://localhost:61616");
4. broker.start();

第二种：Spring 启动

```
<broker useJmx="true" xmlns="http://activemq.apache.org/schema/core"
persistent="true">
    <transportConnectors>
        <transportConnector
            uri="tcp://localhost:61616" discoveryUri="multicast://default"/>
    </transportConnectors>
    <networkConnectors>
        <networkConnector uri="multicast://default"/>
    </networkConnectors>
```



```
</broker>
```

6. 其他语言链接 ActiveMQ

7. ActiveMQ 的安全

权限认证、授权

7.1.1. 简单的授权插件

Simple Authentication Plugin 适用于简单的认证需求，或者用于建立测试环境。它允许在 XML 配置文件中指定用户、用户组和密码等信息。

Xml 如下配置：

```
<broker useJmx="true" persistent="false"
xmlns="http://activemq.apache.org/schema/core"
populateJMSXUserID="true">

    <transportConnectors>
        <transportConnector uri="tcp://localhost:61616"/>
    </transportConnectors>

    <destinations>
        <queue physicalName="TEST.Q" />
    </destinations>

    <plugins>
        <simpleAuthenticationPlugin>
            <users>
                <authenticationUser username="system" password="manager"
groups="users,admins"/>
                <authenticationUser username="user" password="password"
groups="users"/>
                <authenticationUser username="guest" password="password"
groups="guests"/>
            </users>
        </simpleAuthenticationPlugin>
    <!-- 以上是三种权限来连接broker服务器。根据以上的权限来执行下面的生明的队列和主题的操作。 -->
```



```
<!-- lets configure a destination based authorization mechanism -->
<authorizationPlugin>
    <map>
        <authorizationMap>
            <authorizationEntries>
                <authorizationEntry queue=">" read="admins" write="admins"
admin="admins" />
                <authorizationEntry queue="USERS.>" read="users"
write="users" admin="users" />
                <authorizationEntry queue="GUEST.>" read="guests"
write="guests,users" admin="guests,users" />

                <authorizationEntry queue="TEST.Q" read="guests"
write="guests" />

                <authorizationEntry topic=">" read="admins" write="admins"
admin="admins" />
                <authorizationEntry topic="USERS.>" read="users"
write="users" admin="users" />
                <authorizationEntry topic="GUEST.>" read="guests"
write="guests,users" admin="guests,users" />

                <authorizationEntry topic="ActiveMQ.Advisory.>"
read="guests,users" write="guests,users" admin="guests,users"/>
            </authorizationEntries>
        </authorizationMap>
    </map>
</authorizationPlugin>
</plugins>
</broker>
```

7.1.2. 基于 JAAS 授权插件

JAAS Authentication Plugin 依赖标准的 JAAS 机制来实现认证。通常情况下，你需要通过设置 `java.security.auth.login.config` 系统属性来配置 `login modules` 的配置文件。如果没有指定这个系统属性，那么 JAAS Authentication Plugin 会缺省使用 `login.config` 作为文件名。以下是一个 `login.config` 文件的例子：

```
activemq-domain {
    org.apache.activemq.jaas.PropertiesLoginModule required debug=true
```



```
org.apache.activemq.jaas.properties.user="users.properties"
```

```
org.apache.activemq.jaas.properties.group="groups.properties";
};
```

这个 login.config 文件中设置了两个属性:

org.apache.activemq.jaas.properties.user 和

org.apache.activemq.jaas.properties.group 分别用来指向 user.properties 和 group.properties 文件。需要注意的是，PropertiesLoginModule 使用本地文件的查找方式，而且查找时采用的 base directory 是 login.config 文件所在的目录。因此这个 login.config 说明 user.properties 和 group.properties 文件存放在跟 login.config 文件相同的目录里。

以下是 ActiveMQ 配置的一个例子:



第一步：创建 `login.config` 这个文件。存放到 src 下即可。

文件内容如下：

```
activemq-domain {
    org.apache.activemq.jaas.PropertiesLoginModule required
        debug=true
    org.apache.activemq.jaas.properties.user=
"resources/jaasAuthenticationPlugin/users.properties"

    org.apache.activemq.jaas.properties.group="resources/jaasAuthenticationPlugin/groups.
properties";
};
```



```
admins=system
tempDestinationAdmins=system,user
users=system,user
```

`guests=guest`



```
system=manager
user=password
guest=password
```



配置 xml 如下：

```
jaas-broker.xml:

<broker useJmx="true" persistent="true"
xmlns="http://activemq.apache.org/schema/core"
populateJMSXUserID="true">

    <transportConnectors>
        <transportConnector uri="tcp://localhost:61616"/>
    </transportConnectors>

    <plugins>
        <!-- use JAAS to authenticate using the login.config file on the
classpath to configure JAAS -->
        <jaasAuthenticationPlugin configuration="activemq-domain" />

        <!-- lets configure a destination based authorization mechanism -->
        <authorizationPlugin>
            <map>
                <authorizationMap>
                    <authorizationEntries>
                        <authorizationEntry queue=">" read="admins" write="admins"
admin="admins" />
                        <authorizationEntry queue="USERS.>" read="users"
write="users" admin="users" />
                        <authorizationEntry queue="GUEST.>" read="guests"
write="guests,users" admin="guests,users" />

                        <authorizationEntry topic=">" read="admins" write="admins"
admin="admins" />
                        <authorizationEntry topic="USERS.>" read="users"
write="users" admin="users" />
                        <authorizationEntry topic="GUEST.>" read="guests"
write="guests,users" admin="guests,users" />

                        <authorizationEntry topic="ActiveMQ.Advisory.>"
read="guests,users" write="guests,users" admin="guests,users"/>
                    </authorizationEntries>
                </authorizationMap>
            </map>
        </authorizationPlugin>
    </plugins>
```



```

<!-- let's assign roles to temporary destinations. comment this
entry if we don't want any roles assigned to temp destinations -->
<tempDestinationAuthorizationEntry>
    <tempDestinationAuthorizationEntry
        read="tempDestinationAdmins" write="tempDestinationAdmins"
        admin="tempDestinationAdmins"/>
    </tempDestinationAuthorizationEntry>
</authorizationMap>
</map>
</authorizationPlugin>
</plugins>
</broker>

```

Java 代码如下：

```

BrokerService broker;
broker =
createBroker("resources/jaasAuthenticationPlugin/jaas-broker.xml");
System.out.println("从xml文件中获得内容是:
"+broker.getBrokerName()+"\n"+broker.isUseJmx());

```



```

broker.start();

```

第四章。 ActiveMQ 特点

8. Broker 的拓扑结构

集群概述

集群是一个很泛的概念，所谓仁者见仁，智者见智。如何用 ActiveMQ 来进行集群列出了以下几个方面：

- Queue consumer clusters

我们支持 Consumer 对消息高可靠的负载平衡消费，如果一个 Consumer 死掉，该消息会转发到其它的 Consumer 消费的 Queue 上。如果一个 Consumer 获得消息比其它 Consumer 快，那么他将获得更多的消息。如果一个 Consumer 消费缓慢，则其它 Consumer 会替换它。这样就有一个可靠的负载平衡对一





个 Queue 的 Consumer。所以几乎 ActiveMQ 的 Broker 和 Client 使用 failover:

//transport 来配置链接。

- **Broker clusters**

大部情况下是使用一系列的 Broker 和 Client 链接到一起。如果一个 Broker 死掉了，Client 可以自动链接到其它 Broker 上。实现以上行为我们需要用 failover:// 协议作为 Client，如果启动了多个 Broker，Client 可以使用 static discover 或者

Dynamic discovery 容易的从一个 broker 到另一个 broker 直接链接。这样的当一个 broker 上没有 Consumer 的话，那么它的消息不会被消费的，然而该 broker 会通过存储和转发的策略来把该消息发到其它 broker 上。这里的 brokers 没有网络链接各个 brokers。

- **Discovery of brokers**

ActiveMQ 支持使用 static discovery 或者 dynamic discovery 来动态发现 brokers，以至于 Client 能够链接到其它发现的 broker，从而形成网络。

- **Networks of brokers**

如果你用 client/sever 或者 hub/spoke 模式来构建拓扑结构网路并且使用大量的 Client 和大量的 Brokers，这里使用网络链接各个 brokers 的。这里一个 broker 只有一个 Producer 没有 Consumer，以至于消息不被消费，为了避免这种问题，ActiveMQ 支持 Brokers 的 Networks 利用存储转发移动消息到有 Consumer 的 Broker 上。这里支持 queue 和 topic。



● Master Slave

运行独立应用部署的 Brokers 由一个问题，它在任何时候其 broker 在物理上都是独有的。如果一个 broker 断掉，那么你必须等待重新复制消息才能启动 broker。必须是 persistent=true 的情况下成立。例如：这里指的是 Shared Nonthing Master/Slave 模式，就是说如果 master 断掉了，消息的备份会存储在 slave 中。重新拷贝 slave 目录下的数据到 master 的目录下即可，重启 master 了。还有一种就是启动 slave 来替换 master。

● Replicated Message Stores

Master/Slave 模式的另一种方式就是基于 Shared File 和 DataBase 的共享模式。当 master 断掉以后，消息应经存储到硬盘上了，此时 slave 获得锁替换换了 Master。

Broker 的有效性、高效性、可靠性以及集群

8.1.1. Master/Slave

如果你仍然没有一个高性能高可靠性高稳定性的系统架构的话，那么建议使用 Pure MasterSlave 配置，它有高有效性和容错特性，activemq4.0 以后版本都支持该特性。

目前，activemq 包括以下几种不同类型的 master/slave 配置

Master Slave Type	Requirements	Pros	Cons



Pure Master Slave	None	No central point of failure	Requires manual restart to bring back a failed master and can only support 1 slave
Shared File System Master Slave	A Shared File system such as a SAN	Run as many slaves as required. Automatic recovery of old masters	Requires shared file system
JDBC Master Slave	A Shared database	Run as many slaves as required. Automatic recovery of old masters	Requires a shared database. Also relatively slow as it cannot use the high performance journal

如果你想用共享网络文件系统作为一个高性能，高可靠性，建议使用 shared file master slave。如果你不需要 高性能，那么请使用 jdbc master slave 共享文件系统的主从。

8.1.1.1. Shared Nothing(Pure) Master/Slave—集群

Pure master slave 配置提供了一种基本的非共享、全复制的技术不同于 shared file master slave 和 shared jdbc master slave。

How Pure Master Slave works

Pure master slave 工作原理



Master 和 slave 各自有自己的存储目录。

首先，必须 master 先启动 broker，然后 slave 不会自动链接其他 networkConnector。如图：

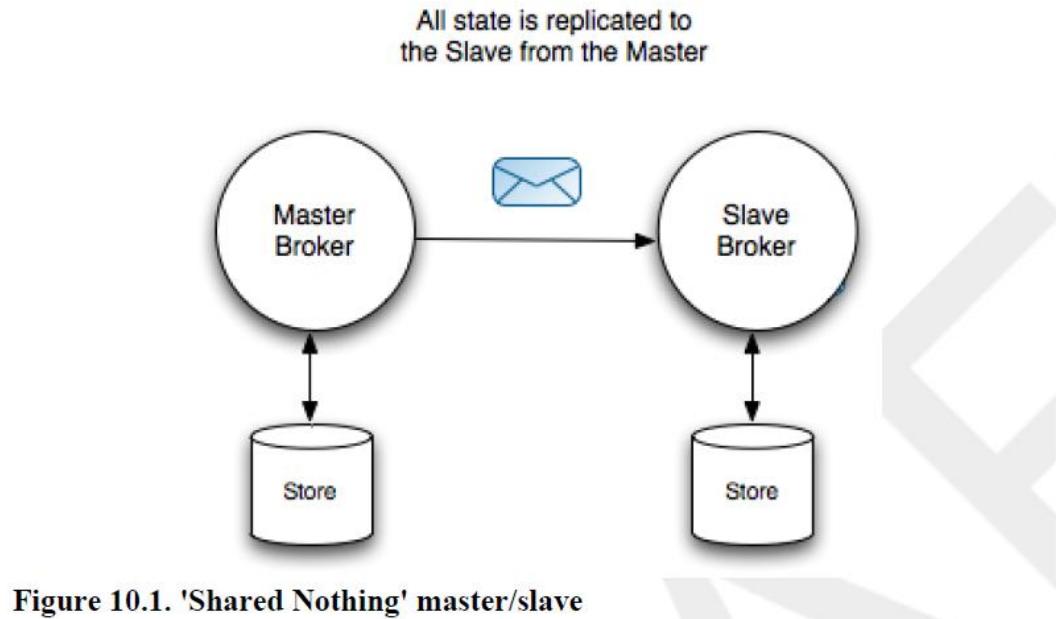


Figure 10.1. 'Shared Nothing' master/slave

运行原理：

- Slave broker 消费 master broker 上所有的消息状态，例如消息、确认和事务状态等。只要 slave broker 连接到了 master broker，它不会（也不被允许）启动任何 network connectors 或者 transport connectors，所以唯一的目的就是复制 master broker 的状态。
- Master broker 只有在消息成功被复制到 slave broker 之后才会响应客户。例如，客户的 commit 请求只有在 master broker 和 slave broker 都处理完毕 commit 请求之后才会结束。
- 当 master broker 失效的时候，slave broker 有两种选择，一种是 slave broker 启动所有的 network connectors 和 transport connectors，这允许客户端切换到 slave broker；另外一种是 slave broker 停止。这种情况下，slave broker 只是复制了 master broker 的状态。
- 客户应该使用 failover transport 并且应该首先尝试连接 master broker。例如：
`failover://(tcp://masterhost:61616, tcp://slavehost:61615)?randomize=false`
 设置 randomize 为 false 就可以让客户总是首先尝试连接 master broker (slave broker 并不会接受任何连接，直到它成为了 master broker)。



Pure Master Slave 具有以下限制:

- 只能有一个 slave broker 连接到 master broker。
- 在因 master broker 失效而导致 slave broker 成为 master 之后，之前的 master broker 只有在当前的 master broker (原 slave broker) 停止后才能重新生效。
- Master broker 失效后而切换到 slave broker 后，最安全的恢复 master broker 的方式是人工处理。首先要停止 slave broker (这意味着所有的客户也要停止)。然后把 slave broker 的数据目录中所有的数据拷贝到 master broker 的数据目录中。然后重启 master broker 和 slave broker。

Master broker 不需要特殊的配置。Slave broker 需要进行以下配置

Xml 代码

```

1. <broker masterConnectorURI="tcp://masterhost:62001" shutdownOn
   MasterFailure="false">
2. ...
3.   <transportConnectors>
4.     <transportConnector uri="tcp://slavehost:61616"/>
5.   </transportConnectors>
6. </broker>
```

其中的 masterConnectorURI 用于指向 master broker，
shutdownOnMasterFailure 用于指定 slave broker 在 master broker 失效的时候是否需要停止。此外，也可以使用如下配置:

Xml 代码

```

1. <broker brokerName="slave" useJmx="false" deleteAllMessages
   OnStartup="true" xmlns="http://activemq.org/config/1.0">
2. ...
3.   <services>
4.     <masterConnector remoteURI="tcp://localhost:62001" u
   serName="user" password="password"/>
5.   </services>
```

</broker>

Pure Master Slave 的限制



- 01: 仅仅只能有一个 slave 可以连接 master
- 02: master 失败后不能自动重新启动，除非是 slave 关闭
- 03: broker 之间没有自动同步。

Pure master slave 数据恢复策略

这里使用手动处理，如果 master 失败，仅仅一种方式来保证数据的同步，

- 01: 关闭 slave broker (可以不关闭 client，因为 client 将一直等待中，知道连接上)
- 02: 拷贝 slave 的数据目录到 master 的数据目录下覆盖即可
- 03: 重启 maste slave

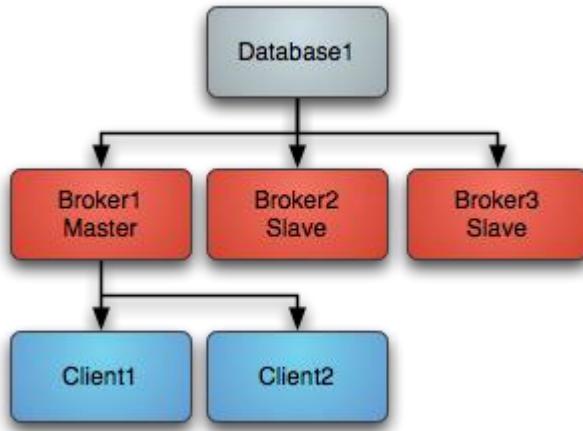
8.1.1.2. JDBC DataBase master/slave—集群

该功能在 ActiveMQ4.1 以上支持。

如果你用单纯的 JDBC 而没有用高性能的持久此时你将依赖于你的数据库作为单独的持久存储引擎。如果你没有高性能的要求，类似这种数据库作为存储引擎可以应用很多场景。

启动

利用数据库作为数据源，采用 Master/Slave 模式，其中在启动的时候 Master 首先获得独有锁，其它 Slaves Broker 则等待获取独有锁。如下图所示：



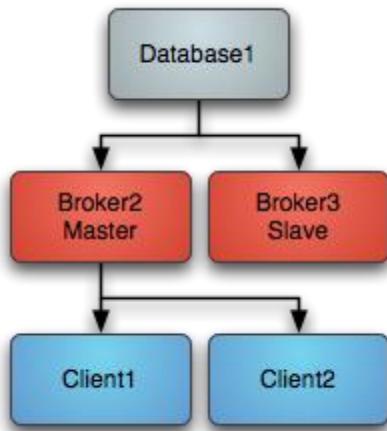
Client 将用 Failover 来链接可用的 Brokers。例如：如下写法：

```
failover:(tcp://broker1:61616, tcp://broker2:61616, tcp://broker3:61616)
```

在启动是仅仅 Master 获得链接，所以 Client 仅仅能过链接到 Master 上。

Master 失败

如果 Master 失败，则它释放独有锁，其他 Slaver 则获取独有锁。如下拓扑图所示：



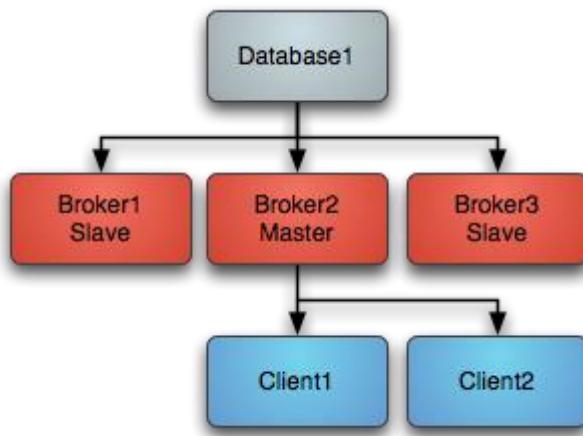
其它 Slaver 立即获得独有锁后此时它将变成 Master，并且启动所有的传输链接。同时，Client 将停止链接

之前的 Master 和将会轮询链接到其他可以利用的 Broker 即新 Master。



Master 重启

任何时候去启动新的 Broker 即新的 Master，它以集群方式接入和启动。如下图，重新建立的 Master 和之前断掉的 Master 的拓扑结构图：



配置 JDBC Master Slave

默认使用<**jdbcPersistenceAdapter**>作为 JDBC Master Slave，但是它没有较高性能。

```

<beans>

    <!-- Allows us to use system properties as variables in this
        configuration file -->
    <bean
        class="org.springframework.beans.factory.config.PropertyPlaceholderC
        onfigurer"/>

    <broker xmlns="http://activemq.apache.org/schema/core">

        <destinationPolicy>
            <policyMap><policyEntries>

                <policyEntry topic="FOO.>">
                    <dispatchPolicy>
                        <strictOrderDispatchPolicy />
                    </dispatchPolicy>
                </policyEntry>
            </policyEntries>
        </policyMap>
    </destinationPolicy>
</broker>
  
```



```
<subscriptionRecoveryPolicy>
    <lastImageSubscriptionRecoveryPolicy />
</subscriptionRecoveryPolicy>
</policyEntry>

</policyEntries></policyMap>
</destinationPolicy>

<persistenceAdapter>
    <jdbcPersistenceAdapter
dataDirectory="${activemq.base}/activemq-data"/>

    <!--
        <jdbcPersistenceAdapter dataDirectory="activemq-data"
dataSource="#oracle-ds"/>
    -->
</persistenceAdapter>

<transportConnectors>
    <transportConnector name="default"
uri="tcp://localhost:61616"/>
</transportConnectors>

</broker>

<!-- This xbean configuration file supports all the standard spring
xml configuration options -->

<!-- Postgres DataSource Sample Setup -->
<!--
<bean id="postgres-ds"
class="org.postgresql.ds.PGPoolingDataSource">
    <property name="serverName" value="localhost"/>
    <property name="databaseName" value="activemq"/>
    <property name="portNumber" value="0"/>
```



```
<property name="user" value="activemq"/>
<property name="password" value="activemq"/>
<property name="dataSourceName" value="postgres"/>
<property name="initialConnections" value="1"/>
<property name="maxConnections" value="10"/>
</bean>
-->

<!-- MySql DataSource Sample Setup --&gt;
&lt;!--
&lt;bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close"&gt;
    &lt;property name="driverClassName" value="com.mysql.jdbc.Driver"/&gt;
    &lt;property name="url"
value="jdbc:mysql://localhost/activemq?relaxAutoCommit=true"/&gt;
    &lt;property name="username" value="activemq"/&gt;
    &lt;property name="password" value="activemq"/&gt;
    &lt;property name="poolPreparedStatements" value="true"/&gt;
&lt;/bean&gt;
--&gt;

<!-- Oracle DataSource Sample Setup --&gt;
&lt;!--
&lt;bean id="oracle-ds"
class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close"&gt;
    &lt;property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/&gt;
    &lt;property name="url"
value="jdbc:oracle:thin:@localhost:1521:AMQDB"/&gt;
    &lt;property name="username" value="scott"/&gt;
    &lt;property name="password" value="tiger"/&gt;
    &lt;property name="poolPreparedStatements" value="true"/&gt;
&lt;/bean&gt;
--&gt;</pre>
```



```
<!-- Embedded Derby DataSource Sample Setup -->
<!--
<bean id="derby-ds"
class="org.apache.derby.jdbc.EmbeddedDataSource">
    <property name="databaseName" value="derbydb"/>
    <property name="createDatabase" value="create"/>
</bean>
-->

</beans>
```

JDBC Master Slave 的工作原理跟 Shared File System Master Slave 类似，只是采用了数据库作为持久化存储。

仅仅一个 broker 可以访问数据库,其他等待..

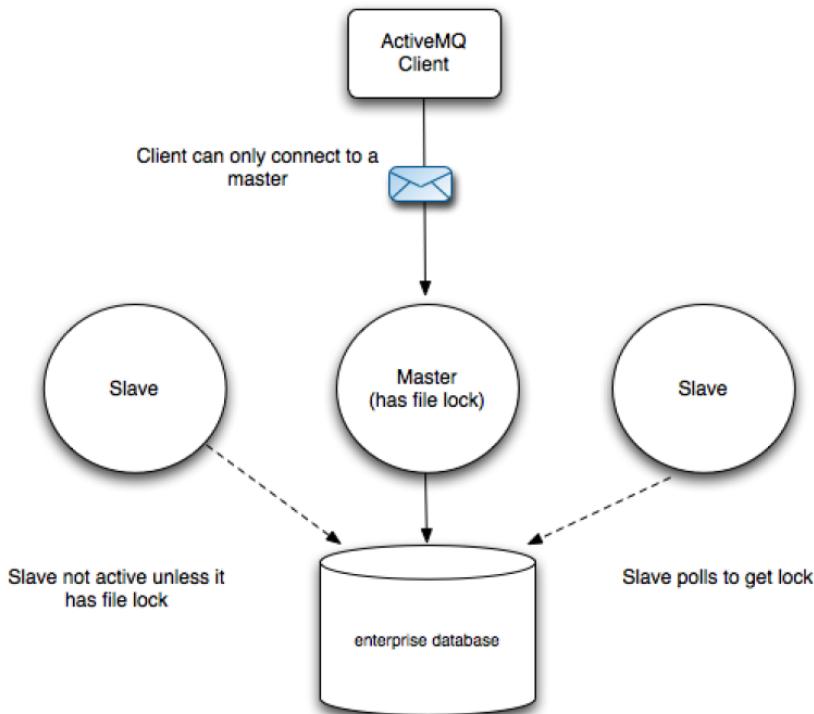


figure 10.2. Shared Database master/slave

8.1.1.3. Shared File master/slave—集群

该功能仅仅在 ActiveMQ4.1 版本以上才有。



如果你用 Shared File System, 它会带了高可用性, 例如: 一个 Broker 死掉, 另一个 Broker 会链接上。

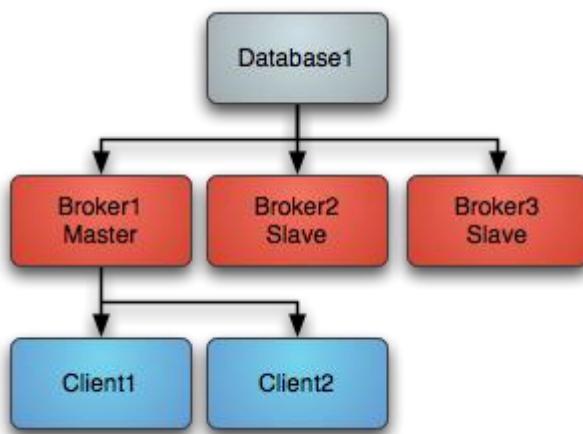
希望基于相同的存储目录下运行多个 Broker, 首先, 该 Broker Master 会首先获得排它锁。如果该 Master Broker 死掉了会释放掉排它锁, 则其它的 Slave Broker 会替换它。Slave 是一个轮询的从 Master 那里获得排它锁。

以下是如何配置 Shared File System Master/Slave

```
<broker useJmx="false" xmlns="http://activemq.org/config/1.0">  
  
    <persistenceAdapter>  
        <journalizedJDBC dataDirectory="/sharedFileSystem/broker"/>  
    </persistenceAdapter>
```

A. Master 启动

在一个 Master 获得一个排它锁基于 File 目录, 那么所有的其他 Broker 都是在等待获取排它锁。如下图:





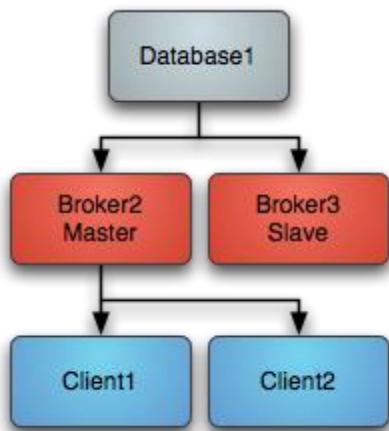
ActiveMQ Client 的调用方式用 Failover 来链接可利用的 Broker。例如：如下 URL 的写法：

```
failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)
```

在第一次启动是，仅仅 Master 可以链接上 File System，所以 Client 仅仅能链接 Master。

B. Master 失败

如果 Master 断掉或者失败，则它立即释放排它锁，其他 Slave 则会链接到 File System 上以及拓扑切换如下图：



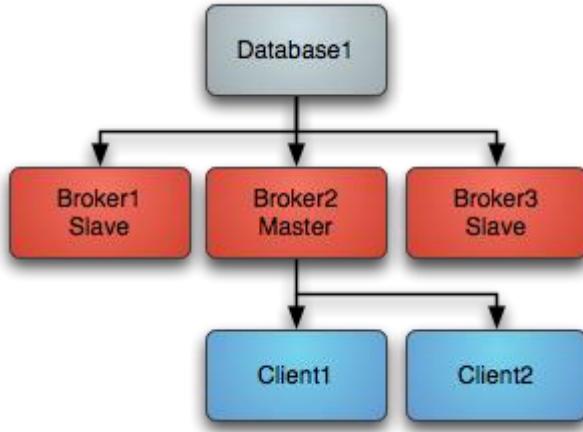
如果其他 Slave 获得排它锁，则它会变成 Master 的角色，以及启动所有的链接传输。同样 Client 失去了 Master 的链接，此时会重寻可用的地址去链接 File System。

C. Master 再次启动

如果 Master 失败了，则其它的 Slave 则会以集群方式接入和启动它并且等待变成 Master。以下图展现了创建一个新的 Master 和以前 failer 的 Master 的一个



拓扑结构图：



总之： 如果你使用 SAN 或者共享文件系统，那么你可以使用 Shared File System Master Slave。基本上，你可以运行多个 broker，这些 broker 共享数据目录。当第一个 broker 得到文件上的排他锁之后，其它的 broker 便会在循环中等待获得这把锁。客户端使用 failover transport 来连接到可用的 broker。当 master broker 失效的时候会释放这把锁，这时候其中一个 slave broker 会得到这把锁从而成为 master broker。

其运行原理和共享数据库原理一样.

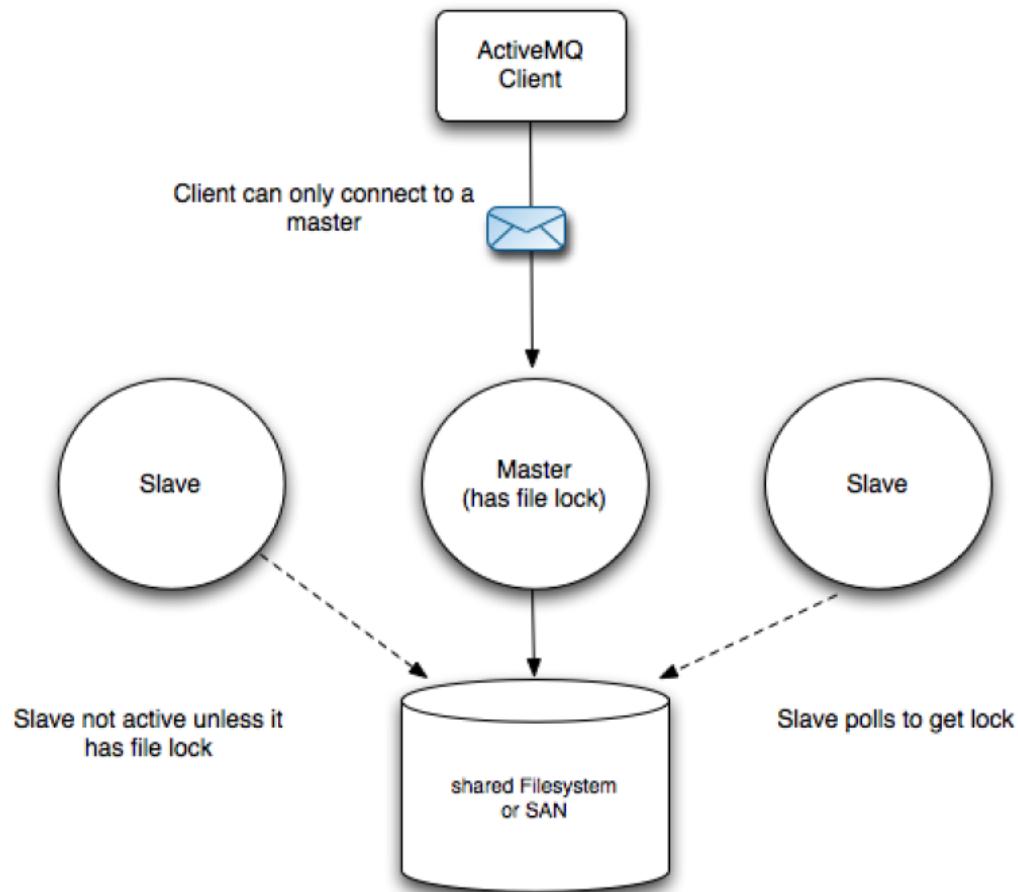


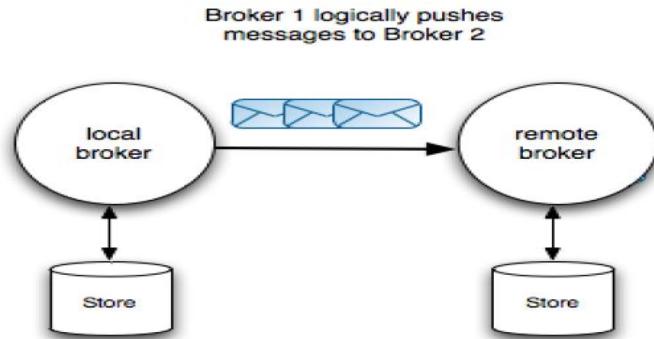
Figure 10.3. Shared File System master/slave

8.1.2. Network of brokers

8.1.2.1. 存储和转发—集群

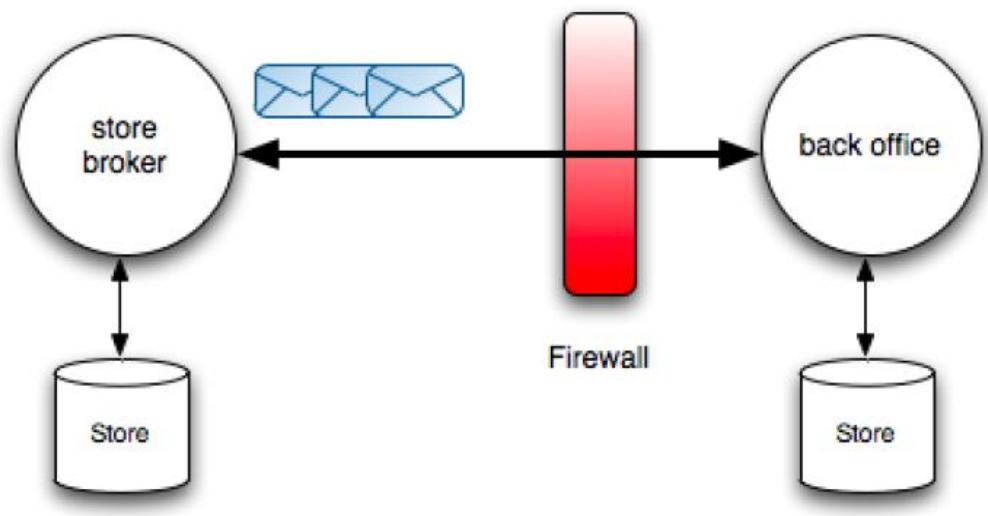
ActiveMQ 默认的两个 broker 链接后是单方向的，broker-A 可以访问消费 broker-B 的消息，

如图一：单向的消息传递.(默认是单向传递的)

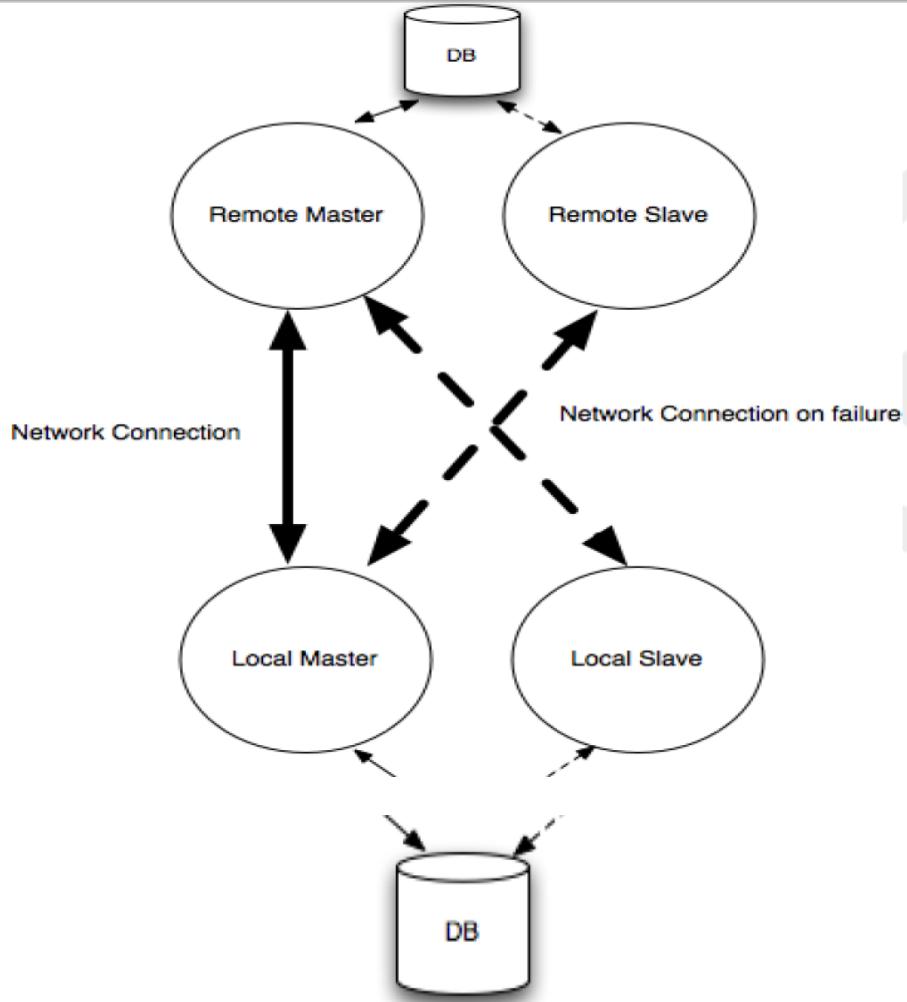


详情参考示例。

如图二:双向的消息传递.



如图三:



配置格式如下：

Example 10.3. Configuring a Store Network Broker

```

<networkConnectors>
    <networkConnector uri="static://(tcp://backoffice:61617)"
        name="bridge"
        duplex="true"
        conduitSubscriptions="true"
        decreaseNetworkConsumerPriority="false">
    </networkConnector>
</networkConnectors>

```

Duplex="true" 表示双发可以通信。

ConduitSubscriptions=“false” 表示每个 Consumer 上都会收到所有的发送的消息。

Name=“bridge” 默认的。



8.1.2.2. Broker 连接(网络寻址 network discovery)--集群

一个常见的场景是有多个 JMS broker，有一个客户连接到其中一个 broker。如果这个 broker 失效，那么客户会自动重新连接到其它的 broker。在 ActiveMQ 中使用 failover:// 协议来实现这个功能。ActiveMQ3.x 版本的 reliable:// 协议已经变更为 failover://。

如果某个网络上有多个 brokers 而且客户使用静态发现（使用 Static Transport 或 Failover Transport）或动态发现（使用 Discovery Transport），那么客户可以容易地在某个 broker 失效的情况下切换到其它的 brokers。然而，stand alone brokers 并不了解其它 brokers 上的 consumers，也就是说如果某个 broker 上没有 consumers，那么这个 broker 上的消息可能会因得不到处理而积压起来。目前的解决方案是使用 Network of brokers，以便在 broker 之间存储转发消息。ActiveMQ 在未来会有更好的特性，用来在客户端处理这个问题。

从 ActiveMQ1.1 版本起，ActiveMQ 支持 networks of brokers。它支持分布式的 queues 和 topics。一个 broker 会相同对待所有的订阅（subscription）：不管他们是来自本地的客户连接，还是来自远程 broker，它都会递送有关的消息拷贝到每个订阅。远程 broker 得到这个消息拷贝后，会依次把它递送到其内部的本地连接上。有两种方式配置 Network of brokers，一种是使用 static transport，如下：

Xml 代码 [查看](#)

```

1. <broker brokerName="receiver" persistent="false" useJmx="false">
2.   <transportConnectors>
3.     <transportConnector uri="tcp://localhost:62002"/>
4.   </transportConnectors>
5.   <networkConnectors>
6.     <networkConnector uri="static:( tcp://localhost:61616,
    tcp://remotehost:61616 )"/>
7.   </networkConnectors>
8. ...
9. </broker>
```

另外一种是使用 multicast discovery，如下：

Xml 代码 [查看](#)

```

1. <broker name="sender" persistent="false" useJmx="false">
2.   <transportConnectors>
3.     <transportConnector uri="tcp://localhost:0" discoveryU
    ri="multicast://default"/>
```



```

4.    </transportConnectors>
5.    <networkConnectors>
6.        <networkConnector uri="multicast://default"/>
7.    </networkConnectors>
8.    ...
9. </broker>

```

Network Connector 有以下属性:

Property	Default Value	Description
name	bridge	name of the network – for more than one network connector between the same two brokers – use different names
dynamicOnly	false	if true, only forward messages if a consumer is active on the connected broker
decreaseNetworkConsumerPriority	false	decrease the priority for dispatching to a Queue consumer the further away it is (in network hops) from the producer
networkTTL	1	the number of brokers in the network that messages and subscriptions can pass through
conduitSubscriptions	true	multiple consumers subscribing to the same destination are treated as one consumer by the network
excludedDestinations	empty	destinations matching this list won't be forwarded across the network
dynamicallyIncludedDestinations	empty	destinations that match this list will be forwarded across the network n.b. an empty list means all destinations not in the excluded list will be forwarded
staticallyIncludedDestinations	empty	destinations that match will always be passed across the network – even if no consumers have ever registered an interest



duplex	false	if true, a network connection will be used to both produce AND Consume messages. This is useful for hub and spoke scenarios when the hub is behind a firewall etc.
--------	-------	--

关于 conduitSubscriptions 属性，这里稍稍说明一下。设想有两个 brokers，分别是 brokerA 和 brokerB，它们之间用 forwarding bridge 连接。有一个 consumer 连接到 brokerA 并订阅 queue: Q.TEST。有两个 consumers 连接到 brokerB，也是订阅 queue: Q.TEST。这三个 consumers 有相同的优先级。然后启动一个 producer，它发送了 30 条消息到 brokerA。如果 conduitSubscriptions=true，那么 brokerA 上的 consumer 会得到 15 条消息，另外 15 条消息会发送给 brokerB。此时负载并不均衡，因为此时 brokerA 将 brokerB 上的两个 consumers 视为一个；如果 conduitSubscriptions=false，那么每个 consumer 上都会收到 10 条消息。

8.1.3. Queue Consumer--集群

ActiveMQ 支持订阅同一个 queue 的 consumers 上的集群。如果一个 consumer 失效，那么所有未被确认（unacknowledged）的消息都会被发送到这个 queue 上其它的 consumers。如果某个 consumer 的处理速度比其它 consumers 更快，那么这个 consumer 就会消费更多的消息。

需要注意的是，笔者发现 AcitveMQ5.0 版本的 Queue consumer clusters 存在一个 bug：采用 AMQ Message Store，运行一个 producer，两个 consumer，并采用如下的配置文件：

Xml 代码

```

1. <beans>
2.   <broker  xmlns="http://activemq.org/config/1.0"  brokerName=
   "BugBroker1"  useJmx="true">
3.
4.   <transportConnectors>
5.     <transportConnector  uri="tcp://localhost:61616"/>
6.
7.   </transportConnectors>
8.   <persistenceAdapter>
9.     <amqPersistenceAdapter  directory="activemq-data/Bug
   Broker1"  maxFileLength="32mb"/>

```



```
10.      </persistenceAdapter>
11.
12.      </broker>
13. </beans>
```

8.1.4. 网络配置

8.1.4.1. Network Property: dynamicOnly

默认是 false，如果是 true，消息将被动态的转接的在其他 broker 的 consumer 上。

8.1.4.2. Network Property: prefetchSize

默认是 1000，指定消息的数量。

8.1.4.3. Network Property: conduitSubscriptions

关于 conduitSubscriptions 属性，这里稍稍说明一下。设想有两个 brokers，分别是 brokerA 和 brokerB，它们之间用 forwarding bridge 连接。有一个 consumer 连接到 brokerA 并订阅 queue: Q.TEST。有两个 consumers 连接到 brokerB，也是订阅 queue: Q.TEST。这三个 consumers 有相同的优先级。然后启动一个 producer，它发送了 30 条消息到 brokerA。如果 conduitSubscriptions=true，那么 brokerA 上的 consumer 会得到 15 条消息，另外 15 条消息会发送给 brokerB。此时负载并不均衡，因为此时 brokerA 将 brokerB 上的两个 consumers 视为一个；如果 conduitSubscriptions=false，那么每个 consumer 上都会收到 10 条消息。

8.1.4.4. Network Property: excludedDestinations

指定排除的地址。

如下图：

1. `networkConnectors>`
 2. `<networkConnector uri="static://(tcp://localhost:61617)">`
-



```
3.           name="bridge"  dynamicOnly="false"  conduitSubscriptio
ns="true"
4.           decreaseNetworkConsumerPriority="false">
5.           <excludedDestinations>
6.               <queue  physicalName="exclude.test.foo"/>
7.               <topic  physicalName="exclude.test.bar"/>
8.           </excludedDestinations>
9.           <dynamicallyIncludedDestinations>
10.              <queue  physicalName="include.test.foo"/>
11.              <topic  physicalName="include.test.bar"/>
12.          </dynamicallyIncludedDestinations>
13.          <staticallyIncludedDestinations>
14.              <queue  physicalName="always.include.queue"/>
15.              <topic  physicalName="always.include.topic"/>
16.          </staticallyIncludedDestinations>
17.      </networkConnector>
18. </networkConnectors>
```

8.1.4.5. Network Property: dynamicallyIncludedDestinations

类似于 excludedDestinations。包括的地址

如下：

```
<dynamicallyIncludedDestinations>

    <queue physicalName="include.test.foo"/>

    <topic physicalName="include.test.bar"/>

</dynamicallyIncludedDestinations>
```

8.1.4.6. Network Property: staticallyIncludedDestinations

静态的包括消息地址。类似于 excludedDestinations

如图下：



```

<networkConnectors>
    <networkConnector uri="static:(tcp://remote:61617)?useExponentialBackOff=false"/>
        <staticallyIncludedDestinations>
            <queue physicalName="management.queue-1"/>
            <queue physicalName="management.queue-2"/>
            <queue physicalName="global.>"/>
            <topic physicalName="global.>"/>
        </staticallyIncludedDestinations>
    </networkConnector>

```

8.1.4.7. Network Property: decreaseNetworkConsumerPriority

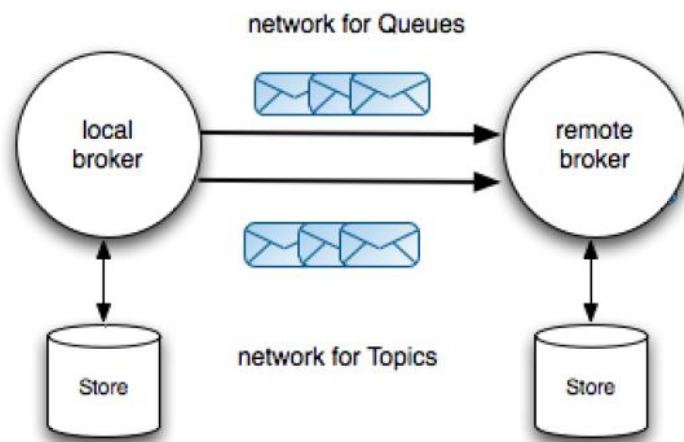
默认是 false。设定消费者优先权。

8.1.4.8. Network Property: networkTTL

默认是 1.

8.1.4.9. Network Property: name

默认是“bridge”表示在多个 broker 之间来标示。如下图：



```

<networkConnectors>
    <networkConnector uri="static://(tcp://remotehost:77171)">
        Name="queue_only"
        Duply="ture"
    </networkConnector>

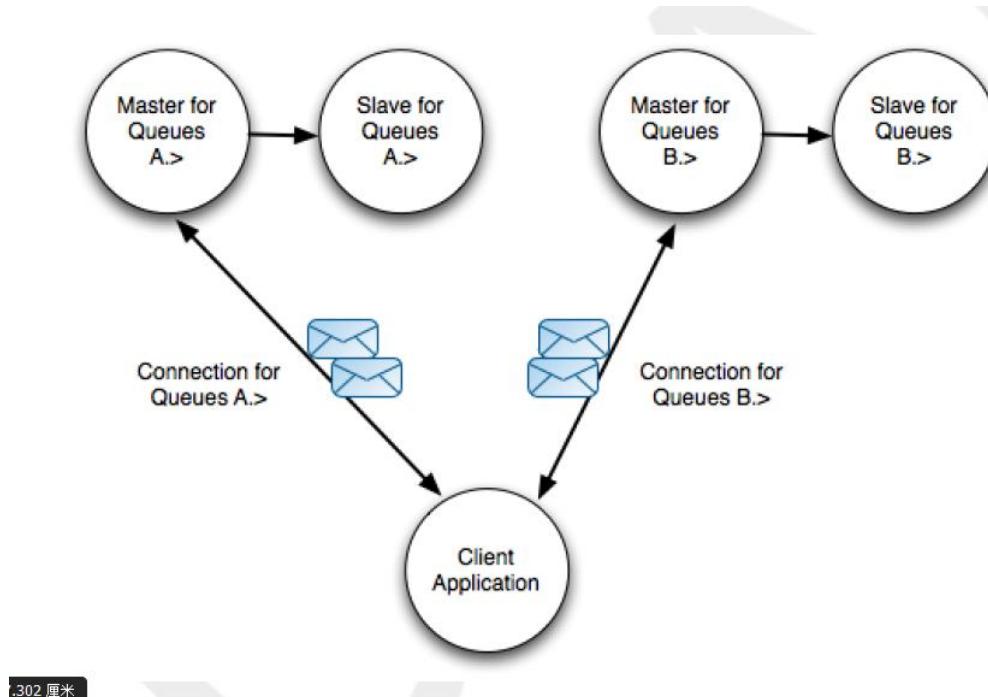
    <networkConnector uri="static://(tcp://remotehost:77171)">
        Name="topic_only"
        Duply="ture"
    </networkConnector>
</networkConnectors>

```



8.1.5. 应用缩小

8.1.5.1. 通信两条路



8.1.5.2. 水平收缩

消息的并发小了。

由于大量的 broker 在 network 中，这里用一个 client 去链接一 brokers 的集群，并且随机的选择一个 broker。

```
failover://(tcp://broker1:61616,tcp://broker2:61616)?randomize=true
```

水平收缩配置：

```
<networkConnector uri="static://(tcp://remotehost:61617)"  
    name="bridge"  
    dynamicOnly="true"  
    prefetchSize="1"  
</networkConnector>
```

8.1.5.3. 垂直收缩

存储性能提高了。



通过 NIO 来提高传递性能，一个链接就是一个线程。

```
<broker>
  <transportConnectors>
```

```
    <transportConnector name="nio" uri="nio://localhost:61616"/>
  </transportConnectors>
</broker>
```

设置内存：

```
<systemUsage>
  <systemUsage>
    <memoryUsage>
      <memoryUsage limit="20 mb"/>
    </memoryUsage>
    <storeUsage>
      <storeUsage limit="1 gb" name="foo"/>
    </storeUsage>
    <tempUsage>
      <tempUsage limit="100 mb"/>
    </tempUsage>
  </systemUsage>
</systemUsage>
```

设置优化的分发消息：

```
<destinationPolicy>
  <policyMap>
    <policyEntries>
```



```
<policyEntry queue="">"memoryLimit="5mb"  
optimizedDispatch= "true" />  
  
</policyEntry>  
  
</policyEntries>  
  
</policyMap>  
  
</destinationPolicy>
```

9. ActivMQ 的先进特性

Destination Features

9.1.1. Composite Destinations

从 1.1 版本起, ActiveMQ 支持 composite destinations。它允许用一个虚拟的 destination 代表多个 destinations。例如你可以通过 composite destinations 在一个操作中同时向 12 个 queue 发送消息。在 composite destinations 中, 多个 destination 之间采用","分割。例如:

Java 代码 [\[剪贴板\]](#)

```
1. Queue queue = new ActiveMQQueue("FOO.A,FOO.B,FOO.C");
```

如果你希望使用不同类型的 destination,那么需要加上前缀如 queue:// 或 topic://, 例如:

Java 代码 [\[剪贴板\]](#)

```
1. Queue queue = new ActiveMQQueue("FOO.A,topic://NOTIFY.FOO.A");
```

以下是 ActiveMQ 配置文件进行配置的一个例子:

Xml 代码 [\[剪贴板\]](#)

```
1. <destinationInterceptors>  
2.   <virtualDestinationInterceptor>  
3.     <virtualDestinations>  
4.       <compositeQueue name="MY.QUEUE">
```



```

5.           <forwardTo>
6.               <queue physicalName="FOO" />
7.               <topic physicalName="BAR" />
8.           </forwardTo>
9.       </compositeQueue>
10.      </virtualDestinations>
11.      </virtualDestinationInterceptor>

</destinationInterceptors>

```

9.1.2. Mirrored Queues

每个 queue 中的消息只能被一个 consumer 消费。然而，有时候你可能希望能够监视生产者和消费者之间的消息流。你可以通过使用 Virtual Destinations 来建立一个 virtual queue 来把消息转发到多个 queues 中。但是为系统中每个 queue 都进行如此的配置可能会很麻烦。

ActiveMQ 支持 Mirrored Queues。Broker 会把发送到某个 queue 的所有消息转发到一个名称类似的 topic，因此监控程序可以订阅这个 mirrored queue topic。为了启用 Mirrored Queues，首先要将 BrokerService 的 useMirroredQueues 属性设置成 true，然后可以通过 destinationInterceptors 设置其它属性，如 mirror topic 的前缀，缺省是“VirtualTopic.Mirror.”。以下是 ActiveMQ 配置文件的一个例子：

Xml 代码

```

1. <broker xmlns="http://activemq.org/config/1.0" brokerName="MirroredQueuesBroker1" useMirroredQueues="true">
2.
3.     <transportConnectors>
4.         <transportConnector uri="tcp://localhost:61616"/>
5.     </transportConnectors>
6.     <destinationInterceptors>
7.         <mirroredQueue copyMessage = "true" prefix="Mirror.Topic"/>
8.     </destinationInterceptors>
9. ...
10. </broker>

```

内嵌 broker 启动 mirrored queue 代码：

```
protected BrokerService createBroker() throws Exception {
```



```

BrokerService answer = new BrokerService();

answer.setUseMirroredQueues(true);

answer.setPersistent(isPersistent());

answer.addConnector(bindAddress);

return answer;

}

```

9.1.3. Wildcards Desitination

Wildcards 用来支持联合的名字分层体系（federated name hierarchies）。它不是 JMS 规范的一部分，而是 ActiveMQ 的扩展。ActiveMQ 支持以下三种 wildcards：

- “.” 用于作为路径上名字间的分隔符。
- “*” 用于匹配路径上的任何名字。
- “>” 用于递归地匹配任何以这个名字开始的 destination。

作为一种组织事件和订阅感兴趣那部分信息的一种方法，这个概念在金融市场领域已经流行了一段时间了。设想你有以下两个 destination：

- PRICE. STOCK. NASDAQ. IBM （IBM 在 NASDAQ 的股价）
- PRICE. STOCK. NYSE. SUNW （SUN 在纽约证券交易所的股价）

订阅者可以明确地指定 destination 的名字来订阅消息，或者它也可以使用 wildcards 来定义一个分层的模式来匹配它希望订阅的 destination。例如：

Subscription	Meaning
PRICE. >	Any price for any product on any exchange
PRICE. STOCK. >	Any price for a stock on any exchange
PRICE. STOCK. NASDAQ. *	Any stock price on NASDAQ
PRICE. STOCK. *. IBM	Any IBM stock price on any exchange

9.1.4. Virtual Topic

虚拟主题，它创建一个逻辑地址并且使用户在客户端可以使用生产者和消费者消息在多个物理地址上关联起来，提高更解耦的消息配置。



最好的场景就是应用在发布/订阅上，允许生产者根本无需知道订阅该消息的消费者的数量。

Jms 持久主题订阅有不足之处，一个持久的消费者的创建必须与 jms 的 clientID 以及持久订阅名称绑定。对于任何时候仅仅有一个 jms 消费者处理和 clientID 以及它的订阅名称绑定的消息。例如：从单个逻辑地址仅仅有一个线程来处理，这将意味着我们不能实现如下：

01：一个消费者失效，则不能转接到其它的订阅上。

02：消息负载均衡

Jms 定义队列其提供了在多个消费者中的有效负载均衡-运行多线程、多个机器处理消息。此时，我们想到了像消息组技术来处理负载平衡。

另一方面，通过订阅物理的逻辑地址来监控队列信息和系统执行情况。

Virtual Topics 讨论

最好的场景就是应用在发布/订阅上，消费者仅仅根据 jms 规范语义来订阅消息，而不需要关心其虚拟地址。

如果消费主题是虚拟的，那么，消费者需要从其对应的逻辑地址上订阅消息，运行多个消费者去运行在多个机器上，多个线程以达到负载均衡。

例如：有一个主题为 “**VirtualTopic.Orders**.” (前缀是 VirtualTopic. 表示虚拟主题)。

我们发送订单到系统 A 和系统 B 上，常规下，我们需要创建 jms 消费者绑定 ClientID_A 和”A”以及 ClientID_B 和”B”。但是，如果使用了 virtual topic 就能够正确的消费队列上的消息。

创建如下消费者即可消费以上消息：

01： **Consumer.A.VirtualTopic.Orders //消费系统 A**

02： **Consumer.B.VirtualTopic.Orders //消费系统 B**

虚拟主题。消费者可以继续使用 jms 规范中的主题语义，但是，如果主题是虚拟的，消费者可以从消费的逻辑主题订阅物理队列，是许多消费者应当在众多的机器和线程运行的负载平衡。



默认虚拟主题是前缀必须是 **VirtualTopic.>**

自定义消费虚拟地址默认格式： **Consumer.*.VirtualTopic.>**

原理如下图：

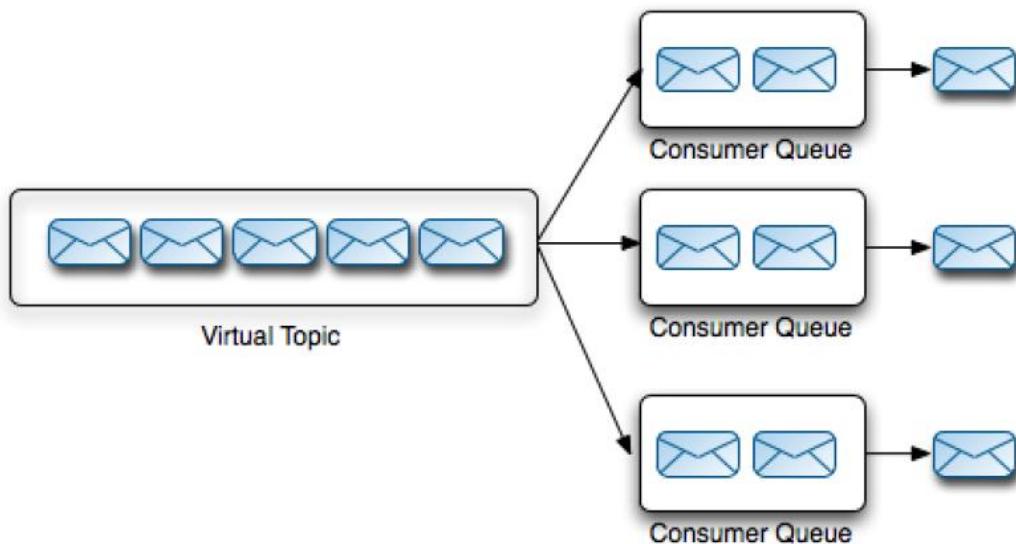


Figure 11.1. Virtual Topics

Xml 配置： 定义匹配所以 Topic， 是其为虚拟主题。

```
<beans>

    <bean
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

    <broker xmlns="http://activemq.apache.org/schema/core">
        <destinationInterceptors>
            <virtualDestinationInterceptor>
                <virtualDestinations>
                    <virtualTopic name=">" prefix="VirtualTopicConsumers.*." />
                </virtualDestinations>
            </virtualDestinationInterceptor>
        </destinationInterceptors>
    </broker>
</beans>
```



```
</broker>
```

```
</beans>
```

Java 代码如下：

```
String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection consumerConnection = connectionFactory.createConnection();
consumerConnection.start();
```

```
Session consumerSessionA = consumerConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue consumerAQueue = consumerSessionA.createQueue("Consumer.A.VirtualTopic.orders");
MessageConsumer consumerA = consumerSessionA.createConsumer(consumerAQueue);

Session consumerSessionB = consumerConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue consumerBQueue = consumerSessionB.createQueue("Consumer.B.VirtualTopic.orders");
MessageConsumer consumerB = consumerSessionB.createConsumer(consumerAQueue);

//setup the sender
Connection senderConnection = connectionFactory.createConnection();
senderConnection.start();
Session senerSession = senderConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic ordersDestination = senerSession.createTopic("VirtualTopic.orders");
MessageProducer producer = senerSession.createProducer(ordersDestination);
```

9.1.5. Configure startup destinations

如果在启动时， 默认创建队列或主题。如下配置：

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://activemq.apache.org/schema/core
    http://activemq.apache.org/schema/core/activemq-core.xsd
    http://camel.apache.org/schema/spring
    http://camel.apache.org/schema/spring/camel-spring.xsd">

    <bean
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />
```



```
<broker xmlns="http://activemq.apache.org/schema/core">
    <destinations>
        <queue physicalName="FOO.BAR" />
        <topic physicalName="SOME.TOPIC" />
    </destinations>

</broker>
```

9.1.6. Destination options

选项

Option Name	Default Value	Description
consumer.prefetchSize	variable	The number of message the consumer will prefetch .
consumer.maximumPendingMessageLimit	0	Use to control if messages for non-durable topics are dropped if a slow consumer situation exists.
consumer.noLocal	false	Same as the noLocal flag on a Topic consumer. Exposed here so that it can be used with a queue.
consumer.dispatchAsync	true	Should the broker dispatch messages asynchronously to the consumer.
consumer.retroactive	false	Is this a Retroactive Consumer .
consumer.selector	null	JMS Selector used with the consumer.



consumer.exclusive	false	Is this an Exclusive Consumer .
consumer.priority	0	Allows you to configure a Consumer Priority .

示例

```
queue = new
ActiveMQQueue("TEST.QUEUE?consumer.dispatchAsync=false&consumer.prefetchSize=10");
consumer = session.createConsumer(queue);
```

9.1.7. Per destination policies

支持多种不同的策略对于单独的地址(队列、主题)或占位符的队列或主题都可以应用。

默认如下属性:

property	default	description
producerFlowControl	true	the producer will slow down and eventually block if no resources(e.g. memory) are available on the broker. If this is off messages get off-lined to disk to prevent memory exhaustion
enableAudit	true	tracks duplicate messages (which can occur in failover for non-persistent messages)
useCache	true	persistent messages are cached for fast retrieval from store
maxPageSize	200	maximum number of persistent messages to page from store at a time



maxBrowsePageSize	400	maximum number of persistent messages to page from store for a browser
memoryLimit	n/a	The memory limit for a given destination. This acts as a child to the overall broker memory specified by the <systemUsage>'s memoryLimit attribute. There is no default for this value; it simply acts as a child to the overall broker memory until the broker memory is exhausted.
minimumMessageSize	1024	for non-serialized messages (embedded broker) - the assumed size of the message used for memory usage calculation. Serialized messages used the serialized size as the basis for the memory calculation
cursorMemoryHighWaterMark	70%	the tipping point at which a system memory limit will cause a cursor to block or spool to disk
storeUsageHighWaterMark	100%	the tipping point at which a system usage store limit will cause a sent to block
prioritizedMessages	false	have the store respect message priority
advisoryForConsumed	false	send an advisory message when a message is consumed by a client
advisoryForDelivered	false	send an advisory message when a message is sent to a client
advisoryForDiscardedMessages	false	send an advisory when a message is discarded
advisoryForSlowConsumers	false	send an advisory message if a consumer is deemed



		slow
advisoryForFastProducers	false	send an advisory message if a producer is deemed fast
advisoryWhenFull	false	send an advisory message when a limit (memory,store,temp disk) is full

Additional properties for a Queue

property	default	description
useConsumerPriority	true	use the priority of a consumer when dispatching messages from a Queue
strictOrderDispatch	false	ignore least loaded and always round robin dispatch
optimizedDispatch	false	don't use a separate thread for dispatching from a Queue
lazyDispatch	false	only page in from store the number of messages that can be dispatched at time
consumersBeforeDispatchStarts	0	when the first consumer connects, wait for specified number of consumers before message dispatching starts
timeBeforeDispatchStarts	0	when the first consumer connects, wait for specified time (in ms) before message dispatching starts



queuePrefetch	n/a	sets the prefetch for consumers that are using the default value
expireMessagesPeriod	30000	the period (in ms) of checks for message expiry on queued messages, value of 0 disables

Additional properties for a Topic

property	default	description
topicPrefetch	n/a	sets the prefetch for topic consumers that are using the default value
durableTopicPrefetch	n/a	sets the prefetch for durable topic consumers that are using the default value

以下示例是基于 dispatch policies

```

<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:amq="http://activemq.apache.org/schema/core"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://activemq.apache.org/schema/core
    http://activemq.apache.org/schema/core/activemq-core.xsd">

    <bean
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"/>

        <broker persistent="false" brokerName="${brokername}"
        xmlns="http://activemq.apache.org/schema/core">

```



```
<!-- lets define the dispatch policy -->
<destinationPolicy>
    <policyMap>
        <policyEntries>
            <policyEntry topic="FOO.">
                <dispatchPolicy>
                    <roundRobinDispatchPolicy />
                </dispatchPolicy>
                <subscriptionRecoveryPolicy>
                    <lastImageSubscriptionRecoveryPolicy />
                </subscriptionRecoveryPolicy>
            </policyEntry>

            <policyEntry topic="ORDERS.">
                <dispatchPolicy>
                    <strictOrderDispatchPolicy />
                </dispatchPolicy>
                <!-- 1 minutes worth -->
                <subscriptionRecoveryPolicy>
                    <timedSubscriptionRecoveryPolicy
recoverDuration="60000" />
                </subscriptionRecoveryPolicy>
            </policyEntry>

            <policyEntry topic="PRICES.">
                <!-- lets force old messages to be discarded for slow
consumers -->
                <pendingMessageLimitStrategy>
                    <constantPendingMessageLimitStrategy limit="10"/>
                </pendingMessageLimitStrategy>

                <!-- 10 seconds worth -->
                <subscriptionRecoveryPolicy>
                    <timedSubscriptionRecoveryPolicy
recoverDuration="10000" />
                </subscriptionRecoveryPolicy>
            </policyEntry>
        </policyEntries>
    </policyMap>
</destinationPolicy>
```



```
</subscriptionRecoveryPolicy>

</policyEntry>
<policyEntry tempTopic="true" advisoryForConsumed="true" />

<policyEntry tempQueue="true" advisoryForConsumed="true" />
</policyEntries>
</policyMap>
</destinationPolicy>
</broker>
```

Message Dispatching Features

9.1.8. Message Cursors

消息游标

当 producer 发送的持久化消息到达 broker 之后, broker 首先会把它保存在持久存储中。接下来, 如果发现当前有活跃的 consumer, 而且这个 consumer 消费消息的速度能跟上 producer 生产消息的速度, 那么 ActiveMQ 会直接把消息传递给 broker 内部跟这个 consumer 关联的 dispatch queue; 如果当前没有活跃的 consumer 或者 consumer 消费消息的速度跟不上 producer 生产消息的速度, 那么 ActiveMQ 会使用 Pending Message Cursors 保存对消息的引用。在需要的时候, Pending Message Cursors 把消息引用传递给 broker 内部跟这个 consumer 关联的 dispatch queue。以下是两种 Pending Message Cursors:

- VM Cursor。在内存中保存消息的引用。
- File Cursor。首先在内存中保存消息的引用, 如果内存使用量达到上限, 那么会把消息引用保存到临时文件中。

在缺省情况下, ActiveMQ 5.0 根据使用的 Message Store 来决定使用何种类型的 Message Cursors, 但是你可以根据 destination 来配置 Message Cursors。

对于 topic, 可以使用的 pendingSubscriberPolicy 有 vmCursor 和 fileCursor。可以使用的 PendingDurableSubscriberMessageStoragePolicy 有 vmDurableCursor 和 fileDurableSubscriberCursor。以下是 ActiveMQ 配置文件的一个例子:

Xm1 代码

1. <destinationPolicy>
2. <policyMap>



```
3.      <policyEntries>
4.          <policyEntry topic="org.apache.">
5.              <pendingSubscriberPolicy>
6.                  <vmCursor />
7.              </pendingSubscriberPolicy>
8.          <PendingDurableSubscriberMessageStoragePolicy>

9.          <vmDurableCursor/>
10.     </PendingDurableSubscriberMessageStoragePolicy>

11.    </policyEntry>
12.  </policyEntries>
13. </policyMap>
14. </destinationPolicy>
```

对于 queue，可以使用的 pendingQueuePolicy 有 vmQueueCursor 和 fileQueueCursor。以下是 ActiveMQ 配置文件的一个例子：

Xml 代码

```
1. <destinationPolicy>
2.   <policyMap>
3.     <policyEntries>
4.       <policyEntry queue="org.apache.">
5.           <pendingQueuePolicy>
6.               <vmQueueCursor />
7.           </pendingQueuePolicy>
8.       </policyEntry>
9.     </policyEntries>
10.   </policyMap>
11. </destinationPolicy>
```

9.1.9. Strict Order Dispatch Policy

严格调度策略



有时候需要保证不同的 topic consumer 以相同的顺序接收消息。通常 ActiveMQ 会保证 topic consumer 以相同的顺序接收来自同一个 producer 的消息。然而，由于多线程和异步处理，不同的 topic consumer 可能会以不同的顺序接收来自不同 producer 的消息。例如有两个 producer，分别是 P 和 Q。差不多是同一时间内，P 发送了 P1、P2 和 P3 三个消息；Q 发送了 Q1 和 Q2 两个消息。两个不同的 consumer 可能会以以下顺序接收到消息：

```
consumer1: P1 P2 Q1 P3 Q2
consumer2: P1 Q1 Q2 P2 P3
```

Strict order dispatch policy 会保证每个 topic consumer 会以相同的顺序接收消息，代价是性能上的损失。以下是采用了 strict order dispatch policy 后，两个不同的 consumer 可能以以下的顺序接收消息：

```
consumer1: P1 P2 Q1 P3 Q2
consumer2: P1 P2 Q1 P3 Q2
```

以下是 ActiveMQ 配置文件的一个例子：

Xm1 代码

```
1. <destinationPolicy>
2.   <policyMap>
3.     <policyEntries>
4.       <policyEntry topic=""></policyEntry>
5.         <dispatchPolicy>
6.           <strictOrderDispatchPolicy />
7.         </dispatchPolicy>
8.       </policyEntry>
9.     </policyEntries>
10.   </policyMap>
11. </destinationPolicy>
```

9.1.10. Round Robin Dispatch Policy

轮转调度策略

介绍过 ActiveMQ 的 prefetch 机制，ActiveMQ 的缺省参数是针对处理大量消息时的高性能和高吞吐量而设置的。所以缺省的 prefetch 参数比较大，而且缺省的 dispatch policies 会尝试尽可能快的填满 prefetch 缓冲。然而在有些情况下，例如只有少量的消息而且单个消息的处理时间比较长，那么在缺省的 prefetch 和 dispatch policies 下，这些少量的消息总是倾向于被分发到个别的 consumer 上。这样就会因为负载的不均衡分配而导致处理时间的增加。

Round robin dispatch policy 会尝试平均分发消息，以下是 ActiveMQ 配置文件的一个例子：



Xml 代码

```
1. <destinationPolicy>
2.   <policyMap>
3.     <policyEntries>
4.       <policyEntry topic="FOO.">
5.         <dispatchPolicy>
6.           <roundRobinDispatchPolicy />
7.         </dispatchPolicy>
8.       </policyEntry>
9.     </policyEntries>
10.   </policyMap>
11. </destinationPolicy>
```

9.1.11. Configure Startup Destinations

启动时 JMS 地址配置

配置如下：

```
<beans>
  <bean
    class="org.springframework.beans.factory.config.PropertyPlaceholderC
onfigurer" />
  <broker xmlns="http://activemq.apache.org/schema/core">
    <destinations>
      <queue physicalName="FOO.BAR" />
      <topic physicalName="SOME.TOPIC" />
    </destinations>
  </broker>
</beans>
```

9.1.12. Async Sends

Activemq 支持异步和同步发送消息。在 ActiveMQ4.0 以上，所有的异步或同步对于 Consumer 来说是变得可配置了。默认是在 ConnectionFactory、Connection、Connection URI 等方面配置对于一个基于 Destination 的 Consumer 来说。

众所周知，如果你想传递给 Slow Consumer 那么你可能使用异步的消息传递，



但是对于 Fast Consumer 你可能使用同步发送消息。(这样可以避免同步和上下文切换额外的增加 Queue 堵塞花费。如果对于一个 Slow Consumer，你使用同步发送消息可能出现 Producer 堵塞等显现。

ActiveMQ 默认设置 `dispatcheAsync=true` 是最好的性能设置。如果你处理的是 Slow Consumer 则使用 `dispatcheAsync=false`，反之，那你使用的是 Fast Consumer 则使用 `dispatcheAsync=false`。

用 Connection URI 来配置 Async 如下：

```
ActiveMQConnectionFactory("tcp://localhost:61616?jms.useAsyncSend=true");
```

用 ConnectionFactory 配置 Async 如下：

```
((ActiveMQConnectionFactory)connectionFactory).setUseAsyncSend(true);
```

用 Connection 配置 Async 如下：

```
((ActiveMQConnection)connection).setUseAsyncSend(true);
```

Connection URI 配置

```
cf = new ActiveMQConnectionFactory("tcp://localhost:61616?jms.useAsyncSend=true");
```

ConnectionFactory 配置

```
((ActiveMQConnectionFactory)connectionFactory).setUseAsyncSend(true);
```

Connection 配置

```
((ActiveMQConnection)connection).setUseAsyncSend(true);
```

9.1.13. Optimized Acknowledgement

ActiveMQ 缺省支持批量确认消息。由于批量确认会提高性能，因此这是缺省的确认



方式。如果希望在应用程序中禁止经过优化的确认方式，那么可以采用如下方法：

Java代码

```
1. cf = new ActiveMQConnectionFactory  
("tcp://localhost:61616?jms.optimizeAcknowledge=false");  
  
2.  
((ActiveMQConnectionFactory)connectionFactory).setOptimizeAcknowledge(false);  
  
3. ((ActiveMQConnection)connection).setOptimizeAcknowledge(false);
```

9.1.14. Producer Flow Control

同步发送消息的producer会自动使用producer flow control；对于异步发送消息的producer，要使用producer flow control，你先要为connection配置一个ProducerWindowSize参数，如下：

Java代码

```
1. ((ActiveMQConnectionFactory)cf).setProducerWindowSize(1024000);
```

ProducerWindowSize 是 producer 在发送消息的过程中，收到 broker 对于之前发送消息的确认之前，能够发送消息的最大字节数

你也可以禁用producer flow control，以下是ActiveMQ配置文件的一个例子：

Java代码

```
1. <destinationPolicy>  
2. <policyMap>  
3. <policyEntries>  
4. <policyEntry topic="FOO." producerFlowControl="false">  
5. <dispatchPolicy>  
6. <strictOrderDispatchPolicy/>  
7. </dispatchPolicy>  
</policyEntry>  
9. </policyEntries>  
10. </policyMap>
```



```
11.</destinationPolicy>
```

前面介绍关于 activemq5.*在内存到达一定数量时，非持久信息将会存储在一个临时文件中。如果你想保存你的非持久消息在内存中，那么就设置内存限制。你的内存限制一旦达到，生产者就停止发送消息。

配置如下：

```
<policyEntry queue= ">" producerFlowControl= "true" memoryLimit= "1mb" >
    <pendingQueuePolicy>
        <vmQueueCursor/>
    </pendingQueuePolicy>
</policyEntry>
```

客户端的异常配置

异常配置是对 send()方法上的引入的异常配置的一种提升，可以通过配置属性 sendFailIfNoSpace= “true” 来代替 send()方法抛出的异常，而且抛出的异常信息在 client。以下配置信息如下：

```
<systemUsage>
    <systemUsage sendFailIfNoSpace= " true " >
        <memoryUsage>
            <memoryUsage limit= "20 mb" />
        </memoryUsage>
    </systemUsage>
</systemUsage>
```

这种属性异常的好处是在异常抛出来，等待一会继续发送执行 send()方法而并不是立即处理异常。

sendFailIfNoSpaceAfterTimeout 属性在 activemq5.3.1 中引入，该属性不仅是对 send()方法引入异常的替换，而且还指定了在等待给定的时间到达时抛出异常。

如果在等待的时间中，broker 中可以再次接收新的消息，则执行 send()，反之，



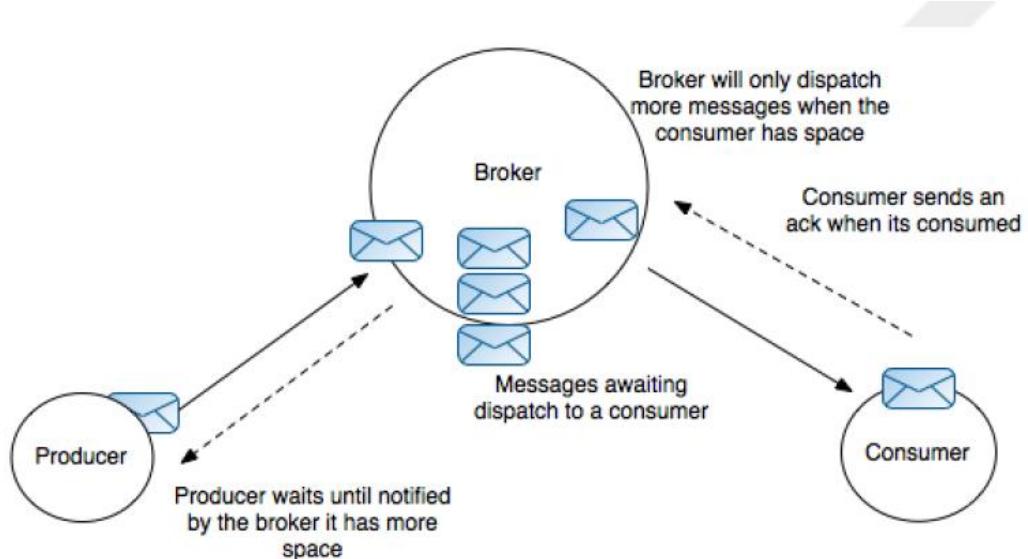
报错。等待在循环执行下去。

以下配置信息如下：

```
<systemUsage>
  <systemUsage sendFailIfNoSpaceAfterTimeout= "3000" >
    <memoryUsage>
      <memoryUsage limit= "20 mb" />
    </memoryUsage>
  </systemUsage>
</systemUsage>
```

这种适合于慢的消费者，大量的消息暂时存储到内存中，然后慢慢的 dispatch。

运行原理如图下：



Java 代码如下：

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setProducerWindowSize(1024000);
```

Xml 配置的策略如下：



```

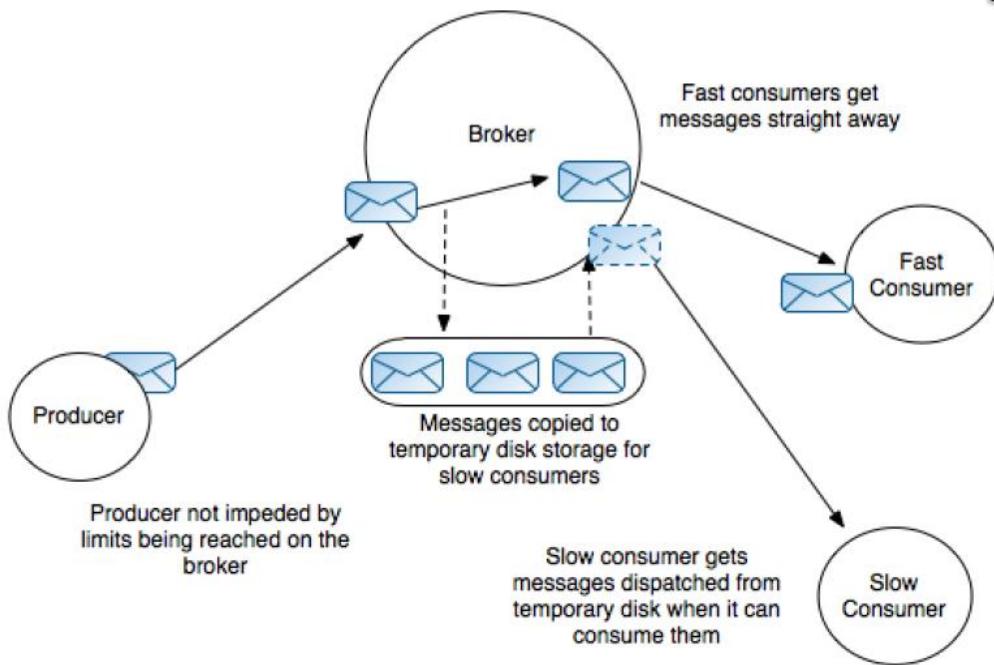
<destinationPolicy>
    <policyMap>
        <policyEntries>

            <policyEntry topic="FOO.>" producerFlowControl="false" memoryLimit="10mb">
                <dispatchPolicy>
                    <strictOrderDispatchPolicy/>
                </dispatchPolicy>
                <subscriptionRecoveryPolicy>
                    <lastImageSubscriptionRecoveryPolicy/>
                </subscriptionRecoveryPolicy>
            </policyEntry>

        </policyEntries>
    </policyMap>
</destinationPolicy>

```

Disabled Producer Flow Control 运行原理:



9.1.15. Strict Order Dispatch Policy

有时候需要保证不同的 topic consumer 以相同的顺序接收消息。通常 ActiveMQ 会保证 topic consumer 以相同的顺序接收来自同一个 producer 的消息。然而，由于多线程和异步处理，不同的 topic consumer 可能会以不同的顺序接收来自不同 producer 的消息。例如有两个 producer，分别是 P 和 Q。差不多是同一时间内，P 发送了 P1、P2 和 P3 三个消息；Q 发送了 Q1 和 Q2 两个消息。两个不同的 consumer 可能会以以下顺序接收到消息：



consumer1: P1 P2 Q1 P3 Q2

consumer2: P1 Q1 Q2 P2 P3 Strict order dispatch policy 会保证每个 topic consumer 会以相同的顺序接收消息，代价是性能上的损失。以下是采用了 strict order dispatch policy 后，两个不同的 consumer 可能以以下的顺序接收消息：
 consumer1: P1 P2 Q1 P3 Q2 consumer2: P1 P2 Q1 P3 Q2

以下是 ActiveMQ 配置文件的一个例子：

Xml 代码

```

1. <destinationPolicy>
2. <policyMap>
3. <policyEntries>
4. <policyEntry topic=""F00.></policyEntry>
5. <dispatchPolicy>
6. <strictOrderDispatchPolicy />
7. </dispatchPolicy>
8. </policyEntry>
9. </policyEntries>
10. </policyMap>
11. </destinationPolicy>
```

Message Features

9.1.16. ActiveMQ Message Properties

Acivemq 默认支持以下属性

Property Name	type	default value	description
JMSDestination	javax.jms.Destination	set by the producer	Destination used by the producer



JMSReplyTo	javax.jms.Destination	null	user defined
JMSType	String	empty	user defiend
JMSDeliveryMode	int	DeliveryMode.PERSISTENT	indicator if messages should be persisted
JMSPriority	int	4	value from 0-9 (nb priorities aren't currently supported)
JMSMessageID	String	unique	unique identifier for the message
JMSTimestamp	long	time the message was sent	time in milliseconds
JMSCorrelationID	String	null	user defined
JMSExpiration	long	0	time in milliseconds to expire the message - 0 means never expire
JMSRedelivered	boolean	false	true if the message is being resent to the consumer

JMS 定义:

Property Name	type	default value	description
---------------	------	---------------	-------------



JMSXDeliveryCount	int	0	number of attempts to send the message
JMSXGroupID	String	null	identity of the message group
JMSXGroupSeq	int	0	sequence number of the message
JMSXProducerTXID	String	null	transaction identifier

ActiveMQ 定义:

Property Name	type	default value	description
JMSActiveMQBrokerInTime	long	0	timestamp (milliseconds) for when the message arrived at the broker
JMSActiveMQBrokerOutTime	long	0	timestamp (milliseconds) for when the message left the broker

9.1.17. Blob Message

配置其地址：包括其上传的 url 的地址。

```
tcp://localhost:61616?jms.blobTransferPolicy.uploadUrl=http://foo.com
```

发送一个 blob 类型的消息如图下：

方式一：

```
BlobMessage message = session.createBlobMessage(new URL("http://some.shared.site.com"));
producer.send(message);
// lets use a local file
```

方式二：

```
BlobMessage message = session.createBlobMessage(new File("/foo/bar"));
producer.send(message);
```

方式三：



```
// lets use a stream
InputStream in = ...;
BlobMessage message = session.createBlobMessage(in);
producer.send(message);
```

接收一个 blob 类型的消息:

```
public class MyListener implements MessageListener {
    public void onMessage(Message message) {
        if (message instanceof BlobMessage) {
            BlobMessage blobMessage = (BlobMessage) message;
            InputStream in = blobMessage.getInputStream();
            // process the stream...
        }
    }
}
```

9.1.18. Advisory Message

ActiveMQ 自身的系统消息地址.我们可以监听该地址来获取 activemq 的系统信息.

ActiveMQ 支持 Advisory Messages, 它允许你通过标准的 JMS 消息来监控系统。目前的 Advisory Messages 支持:

- consumers, producers and connections starting and stopping
- temporary destinations being created and destroyed
- messages expiring on topics and queues
- brokers sending messages to destinations with no consumers.
- connections starting and stopping

Advisory Messages 可以被想象成某种的管理通道, 通过它你可以得到关于 JMS provider、producers、consumers 和 destinations 的信息。Advisory topics 都使用 ActiveMQ.Advisory. 这个前缀, 以下是目前支持的 topics:

Client based advisories

注意:

Advisory topics: advisory message 简写名称

Properties: advisory message 包括的 key 属性

Data structure: 要转换的类型



Advisory Topics	Description	properties	Data Structure
ActiveMQ.Advisory.Connection	Connection start & stop messages		
ActiveMQ.Advisory.Producer.Queue	Producer start & stop messages on a Queue	String='producerCount' – the number of producers	ProducerInfo
ActiveMQ.Advisory.Producer.Topic	Producer start & stop messages on a Topic	String='producerCount' – the number of producers	ProducerInfo
ActiveMQ.Advisory.Consumer.Queue	Consumer start & stop messages on a Queue	String='consumerCount' – the number of Consumers	ConsumerInfo
ActiveMQ.Advisory.Consumer.Topic	Consumer start & stop messages on a Topic	String='consumerCount' – the number of Consumers	ConsumerInfo

基于 destinations 的 advisories

Advisory Topics	Description	properties	Data Structure	default	PolicyEntry property
ActiveMQ.Advisory.Queue	Queue create & destroy	null	null	true	none
ActiveMQ.Advisory.Topic	Topic create & destroy	null	null	true	none
ActiveMQ.Advisory.TempQueue	Temporary Queue create & destroy	null	null	true	none
ActiveMQ.Advisory.TempTopic	Temporary Topic create & destroy	null	null	true	none
ActiveMQ.Advisory.Expired.Queue	Expired messages on a Queue	String='or signalMessageId' – the	Message	true	none



		expired id			
ActiveMQ.Advisory.Expired.Topic	Expired messages on a Topic	String='signalMessageId' – the expired id	Message	true	none
ActiveMQ.Advisory.NoConsumer.Queue	No consumer is available to process messages being sent on a Queue	null	Message	false	sendAdvisoryIfNoConsumers
ActiveMQ.Advisory.NoConsumer.Topic	No consumer is available to process messages being sent on a Topic	null	Message	false	sendAdvisoryIfNoConsumers

在消费者启动/停止的 Advisory Messages 的消息头中有个 consumerCount 属性，他用来指明目前 desination 上活跃的 consumer 的数量。

5.2 版本以上的特点。

Advisory.Topics	Description	properties	Data Structure	default	PolicyEntry property
ActiveMQ.Advisory.SlowConsumer.Queue	Slow Queue Consumer	String='consumerId' – the consumer id	ConsumerInfo	false	advisoryForSlowConsumers
ActiveMQ.Advisory.SlowConsumer.Topic	Slow Topic Consumer	String='consumerId' – the consumer id	ConsumerInfo	false	advisoryForSlowConsumers
ActiveMQ.Advisory.FastProducer.Queue	Fast Queue producer	String='producerId' – the producer id	ProducerInfo	false	advisoryForFastProducers



		producer id			
ActiveMQ.Advisory.FastProducer.Topic	Fast Topic producer	String=' consumerId' - the producer id	ProducerInfo	false	advisoryForFastProducers
ActiveMQ.Advisory.MessageDiscarded.Queue	Message discarded	String=' originalMessageId' - the discarded id	Message	false	advisoryForDiscardingMessages
ActiveMQ.Advisory.MessageDiscarded.Topic	Message discarded	String=' originalMessageId' - the discarded id	Message	false	advisoryForDiscardingMessages
ActiveMQ.Advisory.MessageDelivered.Queue	Message delivered to the broker	String=' originalMessageId' - the delivered id	Message	false	advisoryForDelivery
ActiveMQ.Advisory.MessageDelivered.Topic	Message delivered to the broker	String=' originalMessageId' - the delivered id	Message	false	advisoryForDelivery
ActiveMQ.Advisory.MessageConsumed.Queue	Message consumed by a client	String=' originalMessageId' - the delivered id	Message	false	advisoryForConsumed
ActiveMQ.Advisory.MessageConsumed.To	Message	String=' originalMessageId'	Message	false	advisoryForConsumed



pic	consumed by a client	lMessageId' - the delivered id			
ActiveMQ.Advisory.FULL	A Usage resource is at its limit	String='usageName' - the name of Usage resource	null	false	advisoryWhenFull
ActiveMQ.Advisory.MasterBroker	A broker is now the master in a master/slave configuration	null	null	true	none

New Advisories in 5.4

Advisory Topics	Description	properties	Data Structure	default	PolicyEntry property
ActiveMQ.Advisory.MessageDLQd.Queue	Message sent to DLQ	String='originalMessageId' - the delivered id	Messa ge	Always on	advisoryForConsumed
ActiveMQ.Advisory.MessageDLQd.Topic	Message sent to DLQ	String='originalMessageId' - the delivered id	Messa ge	Always on	advisoryForConsumed



New Advisories in 5.4

Advisory Topics	Description	properties	Advisory Topics	Description
ActiveMQ.Advisory.NetworkBridge	Network bridge being stopped or started	Boolean="started" – true if bridge is started, false if it is stopped Boolean="createdByDuplex" – true if the bridge is created by remote network connector	BrokerInfo – provides data of the remote broker	Always on

以上的这些 destinations 都可以用来作为前缀，在其后面追加其它的重要信息，例如 topic、queue、clientId、producerID 和 consumerID 等。这令你可以利用 Wildcards 和 Selectors 来过滤 Advisory Messages（关于 Wildcard 和 Selector 会在稍后介绍）。

例如，如果你希望订阅 FOO.BAR 这个 queue 上 Consumer 的 start/stop 的消息，那么可以订阅 ActiveMQ.Advisory.Consumer.Queue.FOO.BAR；如果希望订阅所有 queue 上的 start/stop 消息，那么可以订阅 ActiveMQ.Advisory.Consumer.Queue.>；如果希望订阅所有 queue 或者 topic 上的 start/stop 消息，那么可以订阅 ActiveMQ.Advisory.Consumer.>。

举个例子：例如我们想监听所有以 ActiveMQ.Advisory.Consumer.Topic. 开头的 TOPIC. 那么我们在服务端发布发布的 TOPIC 就必须以 ActiveMQ.Advisory.Consumer.Topic.XX 的形式写，然后客户端就可以通过接受 ActiveMQ.Advisory.Consumer.Topic.> 的形式来接受所有这类似的 TOPIC.

任何 advisory message 有 “**ActiveMQ.Advisory.**” 前缀的消息都有以下该属性

property name	type	description	version
originBrokerId	StringProperty	the id of the broker	5.x



		where the advisory originated	
originBrokerName	StringProperty	the name of the broker where the advisory originated	5.x
originBrokerURL	StringProperty	the first URL of the broker where the advisory originated	5.2

通过以下方式获:

```
Message msg = advisoryConsumer.receive(1000);
ActiveMQMessage msg1 = (ActiveMQMessage) msg;
System.out.println(msg1.getProperty("originBrokerId"));
```

帮助方法

通过以下方法获得 advisory desitination，不需要输入上面的前缀地址了

```
AdvisorySupport.getConsumerAdvisoryTopic()
AdvisorySupport.getProducerAdvisoryTopic()
AdvisorySupport.getExpiredTopicMessageAdvisoryTopic()
AdvisorySupport.getExpiredQueueMessageAdvisoryTopic()
AdvisorySupport.getNoTopicConsumersAdvisoryTopic()
AdvisorySupport.getNoQueueConsumersAdvisoryTopic()
AdvisorySupport.getDestinationAdvisoryTopic()
AdvisorySupport.getExpiredQueueMessageAdvisoryTopic()
AdvisorySupport.getExpiredTopicMessageAdvisoryTopic()
AdvisorySupport.getNoQueueConsumersAdvisoryTopic()
AdvisorySupport.getNoTopicConsumersAdvisoryTopic()
```

```
//Version 5.2 onwards
AdvisorySupport.getSlowConsumerAdvisoryTopic()
AdvisorySupport.getFastProducerAdvisoryTopic()
AdvisorySupport.getMessageDiscardedAdvisoryTopic()
AdvisorySupport.getMessageDeliveredAdvisoryTopic()
```



```
AdvisorySupport.getMessageConsumedAdvisoryTopic()
AdvisorySupport.getMasterBrokerAdvisoryTopic()
AdvisorySupport.getFullAdvisoryTopic()
```

示例 1:

```
.....
    ActiveMQQueue destination = (ActiveMQQueue)
session.createQueue(subject);
    Destination advisoryDestination =
AdvisorySupport.getProducerAdvisoryTopic(destination)
    MessageConsumer consumer =
session.createConsumer(advisoryDestination);
    consumer.setMessageListener( this );
.....
public void onMessage(Message msg) {
    if (msg instanceof ActiveMQMessage) {
        try {
            ActiveMQMessage aMsg = (ActiveMQMessage)msg;
            ProducerInfo prod = (ProducerInfo)
aMsg.getDataStructure();
            } catch (JMSException e) {
                log.error( "Failed to process message: " + msg);
            }
        }
    }
}
```

示例 2:

```
//获得 AdvisoryMessage 的 producer

ActiveMQQueue destination = (ActiveMQQueue)
session.createQueue(subject);
ActiveMQDestination destinations =
AdvisorySupport.getProducerAdvisoryTopic(destination);
MessageConsumer consumers = session.createConsumer(destinations);
ActiveMQMessage m1 = (ActiveMQMessage) consumers.receive(9000);
System.out.println("发送者个数: "+m1.getProperty("producerCount"));
DataStructure data=m1.getDataStructure();
if(data.getDataStructureType()==ProducerInfo.DATA_STRUCTURE_TYPE) {
ProducerInfo producerinfo=(ProducerInfo) data;
System.out.println("发送者开始: "+producerinfo);}
```



```

if(data.getDataStructureType()==RemoveInfo.DATA_STRUCTURE_TYPE) {
    RemoveInfo removeinfo=(RemoveInfo) data;
    System.out.println("发送者结束: "+removeinfo);
}

```

示例 3:

```

//获得 AdvisoryMessage 的 destination
//*****AdvisoryMessage
START*****
ActiveMQQueue destination = (ActiveMQQueue)
session.createQueue(subject);
ActiveMQDestination destinations =
AdvisorySupport.getDestinationAdvisoryTopic(destination);
MessageConsumer consumerss =
session.createConsumer(destinations);
ActiveMQMessage m1s = (ActiveMQMessage)
consumerss.receive(9000);
DataStructure datas=m1s.getDataStructure();

if(datas.getDataStructureType()==DestinationInfo.DATA_STRUCTURE_TYPE) {
    DestinationInfo destinationinfo=(DestinationInfo)
datas;
    System.out.println("地址开始: "+destinationinfo);
}

```

示例 4:

```

Topic advisoryTopic =
AdvisorySupport.getMessageDeliveredAdvisoryTopic((ActiveMQDestination)
queue);
MessageConsumer advisoryConsumer =
s.createConsumer(advisoryTopic);
//start throwing messages at the consumer
MessageProducer producer = s.createProducer(queue);
BytesMessage m = s.createBytesMessage();
m.writeBytes(new byte[1024]);
producer.send(m);
Message msg = advisoryConsumer.receive(1000);
ActiveMQMessage msg1 = (ActiveMQMessage) msg;

System.out.println("tttttttt"+msg1.getProperty("originBrokerId"));
DataStructure datas=msg1.getDataStructure();

```



```

System.out.println( msg1.getDataStructure());;

if(datas.getDataStructureType ()==ActiveMQBytesMessage.DATA_STRUCTURE_T
YPE) {
    ActiveMQBytesMessage message=(ActiveMQBytesMessage) datas;
    System.out.println(message.getDestination());
}
}

```

示例 5

```

Topic advisoryTopic =
AdvisorySupport.getExpiredMessageTopic((ActiveMQDestination) queue);
    MessageConsumer advisoryConsumer =
s.createConsumer(advisoryTopic);
    //start throwing messages at the consumer
    MessageProducer producer = s.createProducer(queue);
    producer.setTimeToLive(1);
    for (int i = 0; i < MESSAGE_COUNT; i++) {
        BytesMessage m = s.createBytesMessage();
        m.writeBytes(new byte[1024]);
        producer.send(m);
    }
    Message msg = advisoryConsumer.receive(2000);
    ActiveMQMessage msg1 = (ActiveMQMessage) msg;

System.out.println("tttttttt"+msg1.getProperty("originBrokerId"));

System.out.println("tttttttt"+msg1.getProperty("originalMessageId"));
    DataStructure datas=msg1.getDataStructure();
    System.out.println( msg1.getDataStructure());

if(datas.getDataStructureType ()==ActiveMQBytesMessage.DATA_STRUCTURE_T
YPE) {
    ActiveMQBytesMessage message=(ActiveMQBytesMessage) datas;
    System.out.println(message.getDestination());
}
assertNotNull(msg);
}

```

🔴注：

禁止使用 advisory 需要是 advisorySupprt 属性为 false, 在 broker 或者在 xml 的配置中。



Xml 中：

```
<broker advisorySupport="false">...
```

Java 代码：

```
BrokerService broker = new BrokerService();
broker.setAdvisorySupport(false);
...
broker.start();
```

以及 ActiveMQConnectionFactory 设置

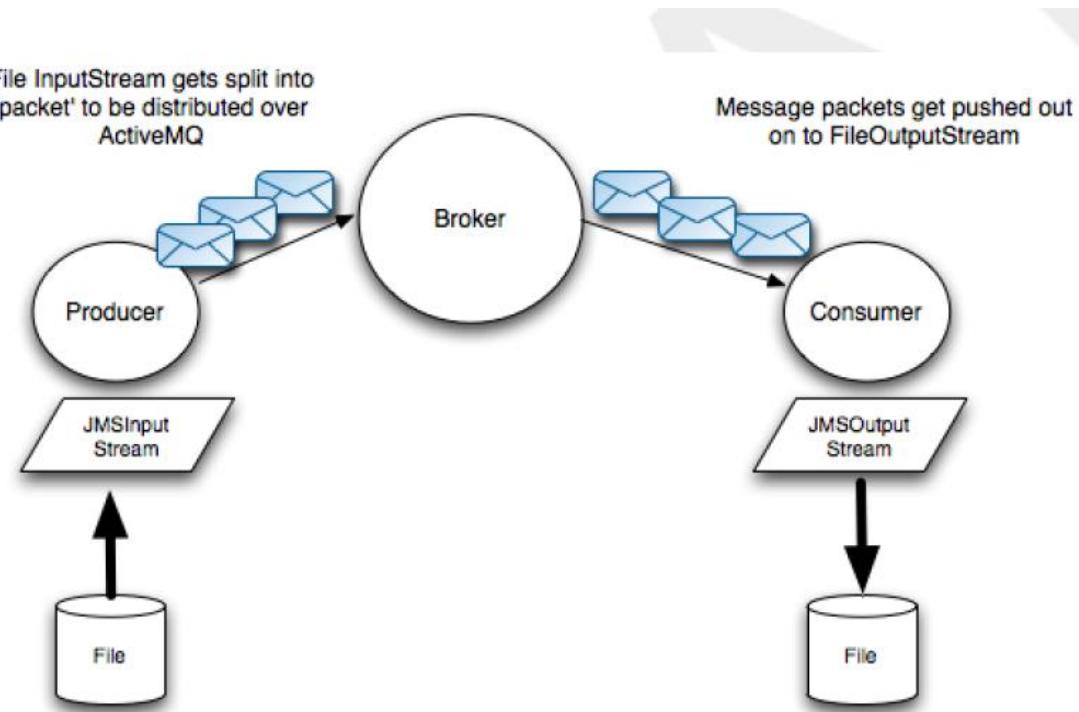
```
"tcp://localhost:61616?jms.watchTopicAdvisories=false"
```

或者

```
ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory();
factory.setWatchTopicAdvisories(false);
```

9.1.19. ActiveMQ Stream

运行原理如图下：





发送一个 Stream 如图下：

```
//source of our large data
FileInputStream in = new FileInputStream("largetextfile.txt");

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
ActiveMQConnection connection = (ActiveMQConnection) connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue destination = session.createQueue(QUEUE_NAME);

OutputStream out = connection.createOutputStream(destination);

//now write the file on to ActiveMQ
byte[] buffer = new byte[1024];
while(true){
    int bytesRead = in.read(buffer);
    if (bytesRead== -1) {
        break;
    }
    out.write(buffer, 0, bytesRead);
}
//close the stream so the receiving side knows the stream is finished
out.close();
```

消费一个 Stream 如图下：

```
//destination of our large data
FileOutputStream out = new FileOutputStream("copied.txt");

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
ActiveMQConnection connection = (ActiveMQConnection) connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//we want to be an exclusive consumer
String exclusiveQueueName= QUEUE_NAME + "?consumer.exclusive=true";
Queue destination = session.createQueue(exclusiveQueueName);

InputStream in = connection.createInputStream(destination);

//now write the file from ActiveMQ
byte[] buffer = new byte[1024];
while(true){
    int bytesRead = in.read(buffer);
    if (bytesRead== -1) {
        break;
    }
    out.write(buffer, 0, bytesRead);
}
out.close();
```



9.1.20. Transformer message

有时候需要在 JMS provider 内部进行 message 的转换。从 4.2 版本起，ActiveMQ 提供了一个 MessageTransformer 接口用于进行消息转换，如下：

Java 代码

```
1. public interface MessageTransformer {  
    2.     Message producerTransform(Session session, MessageProducer producer,  
        Message message) throws JMSException;  
    3.     Message consumerTransform(Session session, MessageConsumer consumer,  
        Message message) throws JMSException;  
    4. }
```

通过在以下对象上通过调用 setTransformer 方法来设置 MessageTransformer：

- ActiveMQConnectionFactory
- ActiveMQConnection
- ActiveMQSession
- ActiveMQMessageConsumer
- ActiveMQMessageProducer

MessageTransformer 接口支持：

- 在消息被发送到 JMS provider 的消息总线前进行转换。通过 producerTransform 方法。
- 在消息到达消息总线后，但是在 consumer 接收到消息前进行转换。通过 consumerTransform 方法。

以下是个简单的例子：



Java 代码

```
public class SimpleMessage implements Serializable {  
    3. private static final long serialVersionUID = 2251041841871975105L;  
    6. private String id;  
    7. private String text;  
    9. public String getId() {  
        10. return id;  
    11. }  
    12. public void setId(String id) {  
        13. this.id = id;  
    14. }  
    15. public String getText() {  
        16. return text;  
    17. }  
    18. public void setText(String text) {  
        19. this.text = text;  
    20. }  
    21.}
```

在 producer 内发送 ObjectMessage，如下：

Java 代码

```
1. SimpleMessage sm = new SimpleMessage();  
2. sm.setId("1");
```



```
3. sm.setText("this is a sample message");

4. ObjectMessage message = session.createObjectMessage();

5. message.setObject(sm);

6. producer.send(message);
```

在 consumer 的 session 上设置一个 MessageTransformer 用于将 ObjectMessage 转换成 TextMessage，如下：

Java 代码

```
1. ((ActiveMQSession)session).setTransformer(new MessageTransformer() {

2.     public Message consumerTransform(Session session, MessageConsumer consumer, Message message) throws JMSEException {

3.         ObjectMessage om = (ObjectMessage)message;

4.         XStream xstream = new XStream();

5.         xstream.alias("simple message", SimpleMessage.class);

6.         String xml = xstream.toXML(om.getObject());

7.         return session.createTextMessage(xml);

8.     }

10.    public Message producerTransform(Session session, MessageProducer consumer, Message message) throws JMSEException {

11.        return null;

12.    }

13.});
```



Consumer Features

9.1.21. Exclusive Consumer(Exclusive Queue)

独有消费者或者独有队列

Queue 中的消息是按照顺序被分发到 consumers 的。然而，当你有多个 consumers 同时从相同的 queue 中提取消息时，你将失去这个保证。因为这些消息是被多个线程并发的处理。有的时候，保证消息按照顺序处理是很重要的。例如，你可能不希望在插入订单操作结束之前执行更新这个订单的操作。

ActiveMQ 从 4. x 版本起开始支持 Exclusive Consumer (或者说 Exclusive Queues)。Broker 会从多个 consumers 中挑选一个 consumer 来处理 queue 中所有的消息，从而保证了消息的有序处理。如果这个 consumer 失效，那么 broker 会自动切换到其它的 consumer。

可以通过 Destination Options 来创建一个 Exclusive Consumer，如下：

Java 代码

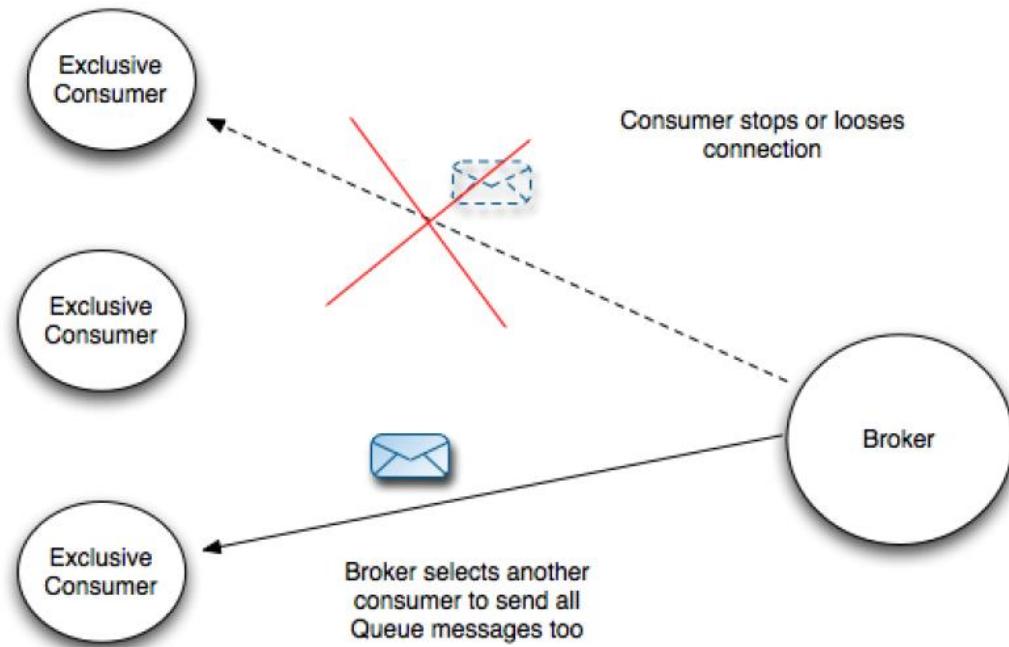
1. `queue = new ActiveMQQueue("TEST.QUEUE?consumer.exclusive=true");`
2. `consumer = session.createConsumer(queue);`

顺便说一下，可以给 consumer 设置优先级，以便针对网络情况(如 network hops) 进行优化，如下：

Java 代码

1. `queue = new ActiveMQQueue("TEST.QUEUE?consumer.exclusive=true&consumer.priority=10");`

运行原理如下图：



如果存在 Exclusive Consumer 和普通的 Consumer，那么 Broker 会首先把消息发送给 Exclusive Consumer。除非该独有消费者死亡。

9.1.22. Message Group

用 Apache 官方文档的话说，Message Groups rock！它是 Exclusive Consumer 功能的增强。逻辑上，Message Groups 可以看成是一种并发的 Exclusive Consumer。跟所有的消息都由唯一的 consumer 处理不同，JMS 消息属性 JMSXGroupID 被用来区分 message group。Message Groups 特性保证所有具有相同 JMSXGroupID 的消息会被分发到相同的 consumer（只要这个 consumer 保持 active）。另外一方面，Message Groups 特性也是一种负载均衡的机制。

在一个消息被分发到 consumer 之前，broker 首先检查消息 JMSXGroupID 属性。如果存在，那么 broker 会检查是否有某个 consumer 拥有这个 message group。如果没有，那么 broker 会选择一个 consumer，并将它关联到这个 message group。此后，这个 consumer 会接收这个 message group 的所有消息，直到：

- Consumer 被关闭。
- Message group 被关闭。通过发送一个消息，并设置这个消息的 JMSXGroupSeq 为-1。

运行原理如下：

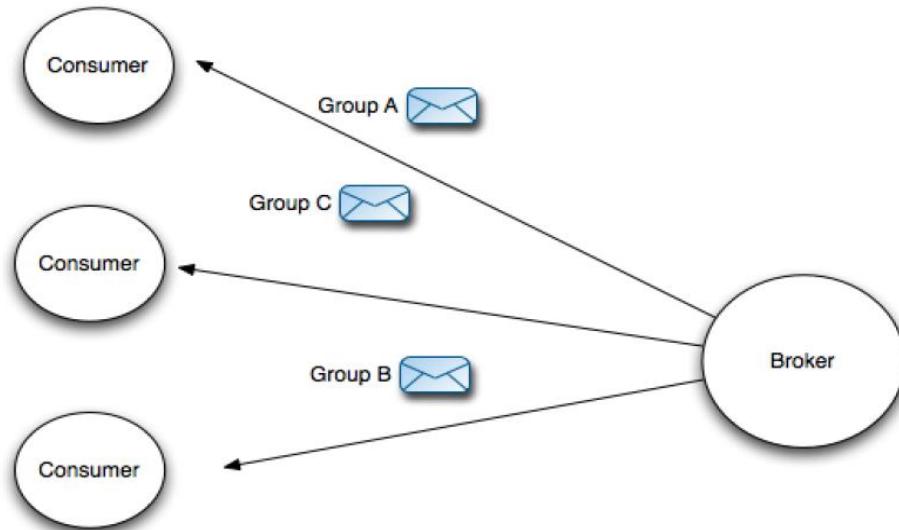


Figure 12.3. Message Groups

创建一个 Message Group:

```

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("group.queue");
MessageProducer producer = session.createProducer(queue);
Message message = session.createTextMessage(
<foo>test</foo>
");
message.setStringProperty("JMSXGroupID", "TEST_GROUP_A");
producer.send(message);
  
```

关闭一个 Message Group:

```

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("group.queue");
MessageProducer producer = session.createProducer(queue);
<foo></foo>

Message message = session.createTextMessage(
<foo>close</foo>
");
message.setStringProperty("JMSXGroupID", "TEST_GROUP_A");
message.setIntProperty("JMSXGroupSeq", -1);
producer.send(message);
  
```

Message Group 策略配置:

如果你的 broker 中已经存在了接收的消息了，但是此时，你可能要后新增一个消费者，最好的方式就是设置消息的延迟分发，直到消费者准备好了（至少给予足够的时间去准备订阅消息）。如果你不想将所有的消息分发给第一个消费者，那么你可以使用 consumersBeforeDispatchStarts(消息分发启动之前的消费者



个数)和 timeBeforeDispatchStarts(消息分发启动之前等待时间)这两个消息策略来设置。

```
<destinationPolicy>
  <policyMap>
    <policyEntries>
      <policyEntry queue= ">" consumersBeforeDispatchStarts= "2"
timeBeforeDispatchStarts= "2000" />
    </policyEntries>
  </policyMap>
</destinationPolicy>
```

消息组的接收:

```
String group = msg.getStringProperty("JMSXGroupID");
        boolean first =
msg.getBooleanProperty("JMSXGroupFirstForConsumer");
        if("A".equals(group)) {
            --counters[0];
            update(group);
            Thread.sleep(500);
        }
        else if("B".equals(group)) {
            --counters[1];
            update(group);
            Thread.sleep(100);
        }
```

9.1.23. Message Selectors

JMS Selectors 用于在订阅中，基于消息属性和 Xpath 语法对进行消息的过滤。

JMS Selectors 由 SQL92 语义定义。以下是个 Selectors 的例子：

Java 代码

1. consumer = session.createConsumer(destination, "JMSType = 'car' AND weight > 2500");



在 JMS Selectors 表达式中，可以使用 IN、NOT IN、LIKE 等，例如：

LIKE '12%3' ('123' true, '12993' true, '1234' false)

LIKE 'l_se' ('lose' true, 'loose' false)

LIKE '\%' ESCAPE '\' ('_foo' true, 'foo' false)

需要注意的是，JMS Selectors 表达式中的日期和时间需要使用标准的 long 型毫秒值。另外表达式中的属性不会自动进行类型转换，例如：

Java 代码

```
1. myMessage.setStringProperty("NumberOfOrders", "2");
```

"NumberofOrders > 1" 求值结果是 false。关于 JMS Selectors 的详细文档请参考 javax.jms.Message 的 javadoc。

上一小节介绍的 Message Groups 虽然可以保证具有相同 message group 的消息被唯一的 consumer 顺序处理，但是却不能确定被哪个 consumer 处理。在某些情况下，Message Groups 可以和 JMS Selector 一起工作，例如：

设想有三个 consumers 分别是 A、B 和 C。你可以在 producer 中为消息设置三个 message groups 分别是"A"、"B"和"C"。然后令 consumer A 使用"JMXGroupID = 'A'"作为 selector。B 和 C 也同理。这样就可以保证 message group A 的消息只被 consumer A 处理。需要注意的是，这种做法有以下缺点：

- producer 必须知道当前正在运行的 consumers，也就是说 producer 和 consumer 被耦合到一起。
- 如果某个 consumer 失效，那么应该被这个 consumer 消费的消息将会一直被积压在 broker 上。

9.1.24. Message Redelivery and Dead-letter Queues

消息的重新传递和死信队列。

ActiveMQ 需要重新传递消息需要 Client 有以下几种操作：

1. Client 用了 transactions 和调用了 rollback() 在 session 中。
2. Client 用了 transactions 和在调用 commit() 之前关闭。
3. Client 在 CLIENT_ACKNOWLEDGE 的传递模式下在 session 中调用了 recover()。

Redelivery 属性：



Property	Default Value	Description
collisionAvoidanceFactor	0.15	The percentage of range of collision avoidance if enabled
maximumRedeliveries	6	Sets the maximum number of times a message will be redelivered before it is considered a poisoned pill and returned to the broker so it can go to a Dead Letter Queue (use value -1 to define infinite number of redeliveries)
initialRedeliveryDelay	1000L	The initial redelivery delay in milliseconds
useCollisionAvoidance	false	Should the redelivery policy use collision avoidance
useExponentialBackOff	false	Should exponential back-off be used (i.e. to exponentially increase the timeout)
backOffMultiplier	5	The back-off multiplier

只有最后一个事物提交后，消息才能发送到 broker 上，事物没有提交前，整个传递消息仍处于事物中。一旦回滚，恢复以前情况。在 broker 端不知道消息是否处于重新传递状态，这将会造成消息分发开销。因此，重新传递和死队列将重新设计在 4.0 以后。

Client 通过 ActiveMQConnection.getRedeliveryPolicy()方法来设置 redelivery 策略；

```
ActiveMQConnectionFactory answer =new
```



```
ActiveMQConnectionFactory("vm://localhost?broker.persistent=false");
    RedeliveryPolicy policy = new RedeliveryPolicy();
    policy.setMaximumRedeliveries(3);
    policy.setBackOffMultiplier((short) 1);
    policy.setInitialRedeliveryDelay(10);
    policy.setUseExponentialBackOff(false);

    answer.setRedeliveryPolicy(policy);
```

当消息试图被传递的次数是配置中 maximumRedeliveries 属性的值时，那么， broker 会认为该消息是一个死消息，并被发送到死队列中。

默认，activemq 中死队列被声明为“ActivemQ.DLQ”，所有不能消费的消息被传递到该死队列中。

你可以在 activemq.xml 中配置 individualDeadLetterStrategy 属性

xml 配置如下：

```
<broker...>
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <!-- Set the following policy on all queues using the '>' wildcard -->
        <policyEntry queue=">">
          <deadLetterStrategy>
            <!--
              Use the prefix 'DLQ.' for the destination name, and
              make
                the DLQ a queue rather than a topic
            -->
            <individualDeadLetterStrategy
              queuePrefix="DLQ."
              useQueueForQueueMessages="true"
            />
          </deadLetterStrategy>
        </policyEntry>
      </policyEntries>
    </policyMap>
```



```
</destinationPolicy>  
...  
</broker>
```

- 自动删除过期消息

有时需要直接删除过期的消息而不需要发送到死队列中，xml 可以使用属性 processExpired=false 来设置。

```
<broker...>  
  <destinationPolicy>  
    <policyMap>  
      <policyEntries>  
        <!-- Set the following policy on all queues using the '>' wildcard -->  
        <policyEntry queue= ">" >  
          <!--  
          Tell the dead letter strategy not to process expired messages  
          so that they will just be discarded instead of being sent to  
          the DLQ  
          -->  
          <deadLetterStrategy>  
            <sharedDeadLetterStrategy processExpired= "false" />  
          </deadLetterStrategy>  
        </policyEntry>  
      </policyEntries>  
    </policyMap>  
  </destinationPolicy>  
...  
</broker>
```

- 存放非持久消息到死队列中

默认情况下，Activemq 不会把非持久的死消息发送到死队列中。

非持久性如果你想把非持久的消息发送到死队列中，需要设置属性 processNonPersistent= “true”

```
<broker...>
```



```
<destinationPolicy>
  <policyMap>
    <policyEntries>
      <!-- Set the following policy on all queues using the '>' wildcard -->
      <policyEntry queue= ">" >
        <!--
            Tell the dead letter strategy to also place non-persisted messages
            onto the dead-letter queue if they can't be delivered.
        -->
        <deadLetterStrategy>
          <sharedDeadLetterStrategy processNonPersistent="true" />
        </deadLetterStrategy>
      </policyEntry>
    </policyEntries>
  </policyMap>
</destinationPolicy>
...
</broker>
```

```
<destinationPolicy>
  <policyMap>
    <policyEntries>
      <policyEntry queue=">">
        <deadLetterStrategy>
          <individualDeadLetterStrategy queuePrefix="DLQ."
            useQueueForQueueMessages="true"
            processExpired="false"
            processNonPersistent="false"/>
        </deadLetterStrategy>
      </policyEntry>
    </policyEntries>
  </policyMap>
</destinationPolicy>
```

9.1.25. Consumer Priority

众所周知, JMS **JMSPriority** 定义了十个消息优先级值, 0 是最低的优先级, 9 是



最高的优先级。另外，客户端应当将0-4 看作普通优先级，5-9 看作加速优先级。

然而，如何定义Consumer Priority的优先级呢？

配置如下：

```
queue = new ActiveMQQueue("TEST.QUEUE?consumer.priority=10");
consumer = session.createConsumer(queue);
```

Consumer 的 Priority 的划分为 0~127 个级别，127 是最高的级别，0 是最低的也是 ActiveMQ 默认的。

这种配置可以是 Broker 根据 Consumer 的优先级来发送消息先到较高的优先级的 Consumer 上，如果某个较高的 Consumer 的缓存预先被消息装载慢，则 Broker 会把消息发送到仅次于它优先级的 Consumer 上。

9.1.26. Slow Consumer Handling

首先简要介绍一下prefetch机制。ActiveMQ通过prefetch机制来提高性能，这意味着客户端的内存里可能会缓存一定数量的消息。缓存消息的数量由prefetch limit来控制。当某个consumer的prefetch buffer已经达到上限，那么broker不会再向consumer分发消息，直到consumer向broker发送消息的确认。可以通过在 ActiveMQConnectionFactory或者ActiveMQConnection上设置 ActiveMQPrefetchPolicy对象来配置prefetch policy。也可以通过connection options或者destination options来配置。例如：

```
tcp://localhost:61616?jms.prefetchPolicy.all=50
tcp://localhost:61616?jms.prefetchPolicy.queuePrefetch=1
```

queue = new ActiveMQQueue("TEST.QUEUE?consumer.prefetchSize=10"); prefetch size 的缺省值如下：

- persistent queues (default value: 1000)
- non-persistent queues (default value: 1000)
- persistent topics (default value: 100)
- non-persistent topics (default value: Short.MAX_VALUE -1)

慢消费者会在非持久的 topics 上导致问题：一旦消息积压起来，会导致 broker 把大量消息保存在内存中，broker 也会因此而变慢。未来 ActiveMQ 可能会实现磁盘缓存，



但是这也还是会存在性能问题。目前 ActiveMQ 使用 Pending Message Limit Strategy 来解决这个问题。除了 prefetch buffer 之外，你还要配置缓存消息的上限，超过这个上限后，新消息到来时会丢弃旧消息。通过在配置文件的 destination map 中配置 PendingMessageLimitStrategy，可以为不用的 topic namespace 配置不同的策略。

A:Pending Message Limit Strategy（等待消息限制策略）目前有以下两种：

01: ConstantPendingMessageLimitStrategy

Limit 可以设置 0、>0、-1 三种方式：

0 表示：不额外的增加其预存大小。

>0 表示：在额外的增加其预存大小。

-1 表示：不增加预存也不丢弃旧的消息。

这个策略使用常量限制

```
<constantPendingMessageLimitStrategy limit="50"/>
```

02: PrefetchRatePendingMessageLimitStrategy

这种策略是利用 Consumer 的之前的预存的大小乘以其倍数等于现在的预存大小。

```
<prefetchRatePendingMessageLimitStrategy multiplier="2.5"/>
```

在以上两种方式中，如果设置 0 意味着除了 prefetch 之外不再缓存消息；如果设置-1 意味着禁止丢弃消息。

此外，你还可以配置消息的丢弃策略，目前有以下两种：

- oldestMessageEvictionStrategy。这个策略丢弃最旧的消息。
- oldestMessageWithLowestPriorityEvictionStrategy。这个策略丢弃最旧的，而且具有最低优先级的消息。

以下是个ActiveMQ配置文件的例子：

Xml代码

```
1. <broker persistent="false" brokerName="${brokername}"  
      xmlns="http://activemq.org/config/1.0">  
2. <destinationPolicy>  
3. <policyMap>  
4. <policyEntries>  
5. <policyEntry topic="PRICES.">  
6. <!-- 10 seconds worth -->
```



```
7. <subscriptionRecoveryPolicy>
8. <timedSubscriptionRecoveryPolicy recoverDuration="10000" />
9. </subscriptionRecoveryPolicy>
10.
11. <!-- lets force old messages to be discarded for slow consumers -->
12. <pendingMessageLimitStrategy>
13. <constantPendingMessageLimitStrategy limit="10"/>
14. </pendingMessageLimitStrategy>
15. </policyEntry>
16. </policyEntries>
17. </policyMap>
18. </destinationPolicy>
19. ...
20.</broker>
```

9.1.27. Consumer Dispatch Async

在 activemq4.0 以后, 你肯能选择 broker 执行同步或异步的方法消息给消费者。

默认是 true

ConnectionFactory 中配置

```
((ActiveMQConnectionFactory)connectionFactory).setDispatchAsync( false );
```

Connection 中配置

```
((ActiveMQConnection)connection).setDispatchAsync( false );
```

Destination URI 中配置

```
queue = new ActiveMQQueue( "TEST.QUEUE?consumer.dispatchAsync= false" );
consumer = session.createConsumer(queue);
```



9.1.28. Retroactive Consumer

消费者追溯消息。

ActiveMQ 支持 6 种恢复策略，可以自行选择使用不同的策略

1. <fixedCountSubscriptionRecoveryPolicy>:

这种策略限制在基于一个静态的计数中对于主题 (Topic) 消息缓存的数量。

property name	default value	description
maximumSize	100	the number of messages allowed in the Topics cache

2. <fixedSizedSubscriptionRecoveryPolicy>:

这种策略限制在内存使用量中对于主题 (Topic) 消息缓存的数量。这是 ActiveMQ 的默认持久恢复策略。你可以选择设置 cache 的大小来应用与所有的主题 [Topic]。

property name	default value	description
maximumSize	6553600	The memory size in bytes for this cache
useSharedBuffer	true	If true - the amount of memory allocated will be used across all Topics



```
<broker persistent="false"
xmlns="http://activemq.apache.org/schema/core">↓
↓
<destinationPolicy>↓
  <policyMap>↓
    <policyEntries>↓
      <policyEntry topic="org.apache.activemq.test.">↓
        <subscriptionRecoveryPolicy>↓
          <fixedSizedSubscriptionRecoveryPolicy
useSharedBuffer="false" maximumSize="600000"/>↓
        </subscriptionRecoveryPolicy>↓
      </policyEntry>↓
    </policyEntries>↓
  </policyMap>↓
</destinationPolicy>↓
</broker>↓
```

3. <lastImageSubscriptionRecoveryPolicy>: only keep the last message.

这种策略仅仅保持发送到主题（Topic）的最后一个消息。

```
<broker persistent="false"
xmlns="http://activemq.apache.org/schema/core">↓
↓
<destinationPolicy>↓
  <policyMap>↓
    <policyEntries>↓
      <policyEntry topic="org.apache.activemq.test.">↓
        <subscriptionRecoveryPolicy>↓
          <lastImageSubscriptionRecoveryPolicy/>↓
        </subscriptionRecoveryPolicy>↓
      </policyEntry>↓
    </policyEntries>↓
  </policyMap>↓
</destinationPolicy>↓
↓
</broker>↓
```

4. <noSubscriptionRecoveryPolicy>:

这种策略是不保存主题消息，不需要任何配置。



5. <queryBasedSubscriptionRecoveryPolicy>:

这种策略基于一个 JMS 属性选择器应用到所有的消息来设置其消息缓存的大小

property name	default value	description
query	null	caches only messages that match the query

```

<broker persistent="false"
xmlns="http://activemq.apache.org/schema/core">↓
↓
<destinationPolicy>↓
  <policyMap>↓
    <policyEntries>↓
      <policyEntry topic="org.apache.activemq.test.>">↓
        <subscriptionRecoveryPolicy>↓
          <queryBasedSubscriptionRecoveryPolicy
query="#myQuery" />↓
        </subscriptionRecoveryPolicy>↓
      </policyEntry>↓
    </policyEntries>↓
  </policyMap>↓
</destinationPolicy>↓
</broker>↓

```

6. <timedSubscriptionRecoveryPolicy>

这种策略是基于应用到每个消息的过期时间来限制其消息缓存数量。提示这种消息的生命周期时间来源于消息发送者设置其 timeToLive 参数。

property name	default value	description
recoverDuration	60000	the time in milliseconds to keep messages in the cache

```

<policyEntry topic="PRICES.>">
  <!-- 10 seconds worth -->
  <subscriptionRecoveryPolicy>
    <timedSubscriptionRecoveryPolicy recoverDuration="10000" />
  </subscriptionRecoveryPolicy>

```



```

<broker persistent="false"
xmlns="http://activemq.apache.org/schema/core">↓
↓
<destinationPolicy>↓
<policyMap>↓
<policyEntries>↓
<policyEntry topic="org.apache.activemq.test.">↓
<subscriptionRecoveryPolicy>↓
<timedSubscriptionRecoveryPolicy />↓
</subscriptionRecoveryPolicy>↓
</policyEntry>↓
</policyEntries>↓
</policyMap>↓
</destinationPolicy>↓
↓
</broker>↓
.

```

故名思议，就是消费者失去消费上一次生产者发送的消息，然而它想重新获取它。那么 ActiveMQ 默认的会缓存默认大小是 64kb 的消息数据以备恢复它。

原理是调用了 ActiveMQ 的恢复策略，`FixedSizedSubscriptionRecoveryPolicy` 这个类。

举例 Java 代码配置如下：

Example 11.7. Setting A Retroactive Consumer

```

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection connection = connectionFactory.createConnection();
connection.start();

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic topic = session.createTopic("soccer.division1.leeds?consumer.retroactive=true");
MessageConsumer consumer = session.createConsumer(topic);
Message result = consumer.receive();

```

Xml 配置如下：



```
<broker brokerName="test-broker"
       persistent="true"
       useShutdownHook="false"
       deleteAllMessagesOnStartup="true"
       xmlns="http://activemq.apache.org/schema/core">
  <transportConnectors>
    <transportConnector uri="tcp://localhost:61635"/>
  </transportConnectors>
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic="Topic.FixedSizedSubs">
          <subscriptionRecoveryPolicy>
            <fixedSizeSubscriptionRecoveryPolicy maximumSize="2000000"
              useSharedBuffer="false"/>
          </subscriptionRecoveryPolicy>❶
        </policyEntry>

        <policyEntry topic="Topic.LastImageSubs">
          <subscriptionRecoveryPolicy>
            <lastImageSubscriptionRecoveryPolicy/>
          </subscriptionRecoveryPolicy>❷
        </policyEntry>

        <policyEntry topic="Topic.NoSubs">
          <subscriptionRecoveryPolicy>
            <noSubscriptionRecoveryPolicy/>
          </subscriptionRecoveryPolicy>❸
        </policyEntry>

        <policyEntry topic="Topic.TimedSubs">
          <subscriptionRecoveryPolicy>
            <timedSubscriptionRecoveryPolicy recoverDuration="25000"/>
          </subscriptionRecoveryPolicy>❹
        </policyEntry>
      </policyEntries>
    </policyMap>
  </destinationPolicy>
</broker>
```

10. 优化 ActiveMQ 性能

一般技术

10.1.1. Persistent vs Non-Persistent Message

持久化和非持久化传递

1. PERSISTENT (持久性消息)

这是 ActiveMQ 的默认传送模式，此模式保证这些消息只被传送一次和成功使用一次。对于这些消息，可靠性是优先考虑的因素。可靠性的另一个重要方面是确

保持持久性消息传送至目标后，消息服务在向消费者传送它们之前不会丢失这些消息。这



意味着在持久性消息传送至目标时，消息服务将其放入持久性数据存储。如果消息服务

由于某种原因导致失败，它可以恢复此消息并将此消息传送至相应的消费者。虽然这样

增加了消息传送的开销，但却增加了可靠性。

2. NON_PERSISTENT（非持久性消息）

保证这些消息最多被传送一次。对于这些消息，可靠性并非主要的考虑因素。

此模式并不要求持久性的数据存储，也不保证消息服务由于某种原因导致失败后消息不会丢失。

有两种方法指定传送模式：

1. 使用 `setDeliveryMode` 方法，这样所有的消息都采用此传送模式；

2. 使用 `send` 方法为每一条消息设置传送模式；

方法一： `void send(Destination destination, Message message, int deliveryMode, int priority, long timeToLive);`

方法二： `void send(Message message, int deliveryMode, int priority, long timeToLive);`

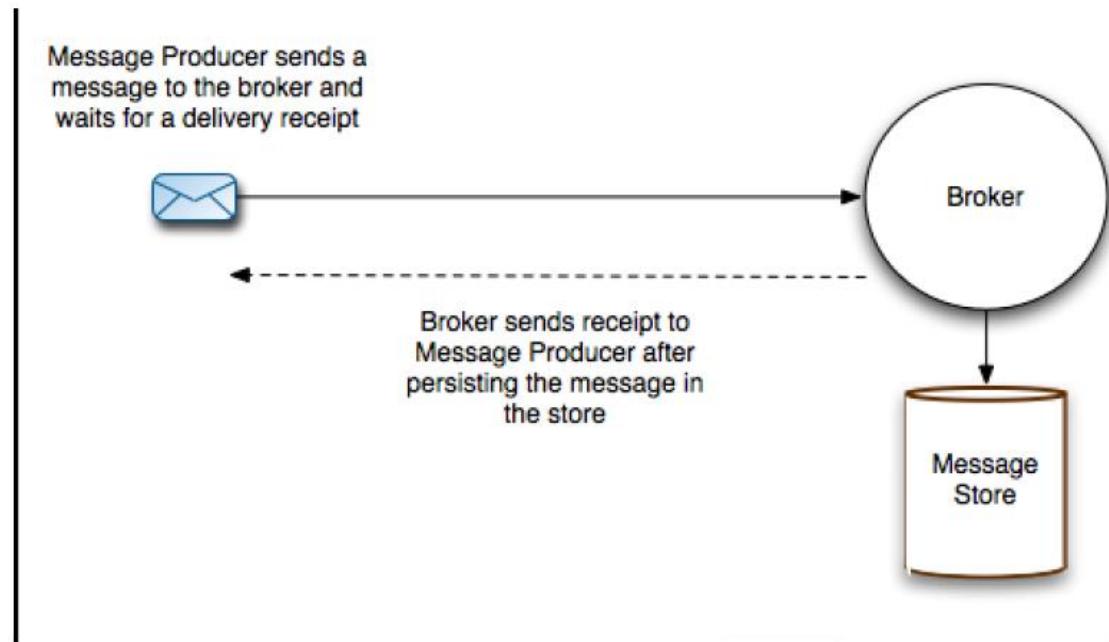
其中 `deliveryMode` 为传送模式，`priority` 为消息优先级，`timeToLive` 为消息过期时间。

方法三： `producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);`

JMS 规范 1.1 允许消息传递包括 Persistent 和 Non-Persistent。

Non-persistent 传递消息比 Persistent 传递消息速度更快，原因如下：

- 1) Non-persistent 发送消息是异步的，Producer 不需要等待 Consumer 的 receipt 消息。如下图：
- 2) Persistent 传递消息是需要把消息存储起来。然后在传递，这样很慢。



10.1.2. Transactions

事务

以下例子说明了 Transaction 比 Non-transaction 的性能高。

Transaction 和 Non-transaction 代码如下：

```
public void sendTransacted() throws JMSException {
    //create a default connection - we'll assume a broker is running
    //with its default configuration
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    Connection connection = cf.createConnection();
    connection.start();

    //create a transacted session

    Session session = connection.createSession(true, Session.SESSION_TRANSACTED)
    Topic topic = session.createTopic("Test.Transactions");
    MessageProducer producer = session.createProducer(topic);
    int count =0;
    for (int i =0; i < 1000; i++) {
        Message message = session.createTextMessage("message " + i);
        producer.send(message);

        //commit every 10 messages
        if (i!=0 && i%10==0) {
            session.commit();
        }
    }
}
```



```
        session.commit();
    }
}

public void sendNonTransacted() throws JMSException {
    //create a default connection - we'll assume a broker is running
    //with its default configuration

    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    Connection connection = cf.createConnection();
    connection.start();

    //create a default session (no transactions)

    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Topic topic = session.createTopic("Test.Transactions");
    MessageProducer producer = session.createProducer(topic);
    int count =0;
    for (int i =0; i < 1000; i++) {
        Message message = session.createTextMessage("message " + i);
        producer.send(message);
    }
}
```

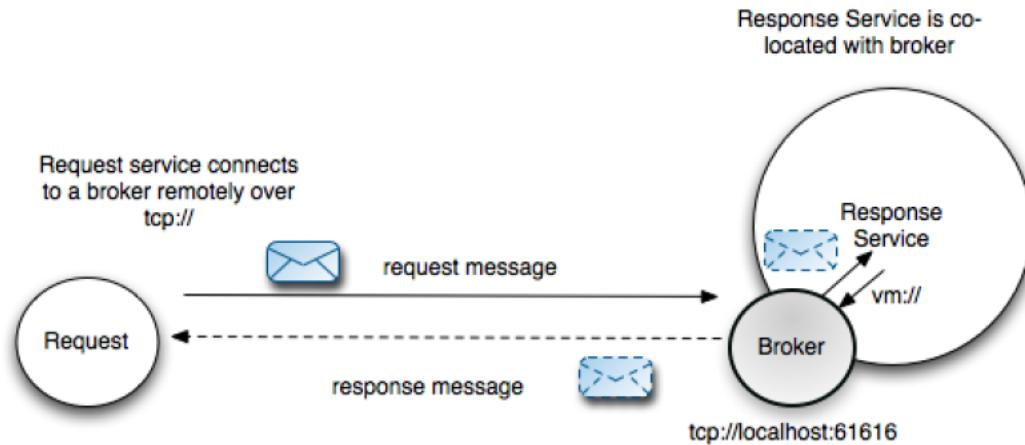
10.1.3. 超快回应消息

内嵌 broker; 如下图:

```
BrokerService broker = new BrokerService();
broker.setBrokerName("service");
broker.setPersistent(false);
broker.addConnector("tcp://localhost:61616");
broker.start();
```

下面以 Co-locate (合作定位)with a broker 为例。

其运行原理如下图:



Java 代码如下：

创建一个 queue 服务：

```
//By default a broker always listens on vm://<broker name>
//so we don't need to set up an explicit connector for
//vm:// connections - just the tcp connector

BrokerService broker = new BrokerService();
broker.setBrokerName("service");
broker.setPersistent(false);
broker.addConnector("tcp://localhost:61616");
broker.start();

ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("vm://service");
cf.setCopyMessageOnSend(false);
Connection connection = cf.createConnection();
connection.start();
```



```

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//we will need to respond to multiple destinations - so use null
//as the destination this producer is bound to

final MessageProducer producer = session.createProducer(null);

//create a Consumer to listen for requests to service

Queue queue = session.createQueue("service.queue");
MessageConsumer consumer = session.createConsumer(queue);
consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message msg) {
        try {
            TextMessage textMsg = (TextMessage)msg;
            String payload = "REPLY: " + textMsg.getText();
            Destination replyTo;
            replyTo = msg.getJMSReplyTo();
            textMsg.clearBody();
            textMsg.setText(payload);
            producer.send(replyTo, textMsg);
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
});

```

创建一个 queueRequestor:

```

ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
QueueConnection connection = cf.createQueueConnection();
connection.start();
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("service.queue");
QueueRequestor requestor = new QueueRequestor(session,queue);
for(int i = 0; i < 10; i++) {
    TextMessage msg = session.createTextMessage("test msg: " + i);
    TextMessage result = (TextMessage)requestor.request(msg);
    System.err.println("Result = " + result.getText());
}

```

注意:

```

ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setCopyMessageOnSend(false);

```

设置发送的消息不需要 copy。

10.1.4. Tuning the OpenWire protocol

跨语言协议

//TODO



10.1.5. Tuning the TCP Transport

TCP 协议是 ActiveMQ 使用最常见的协议。

有以下两点影响 TCP 协议性能：

- 1) socketBufferSize=缓存， 默认是 65536。
- 2) tcpNoDelay=默认是 false，

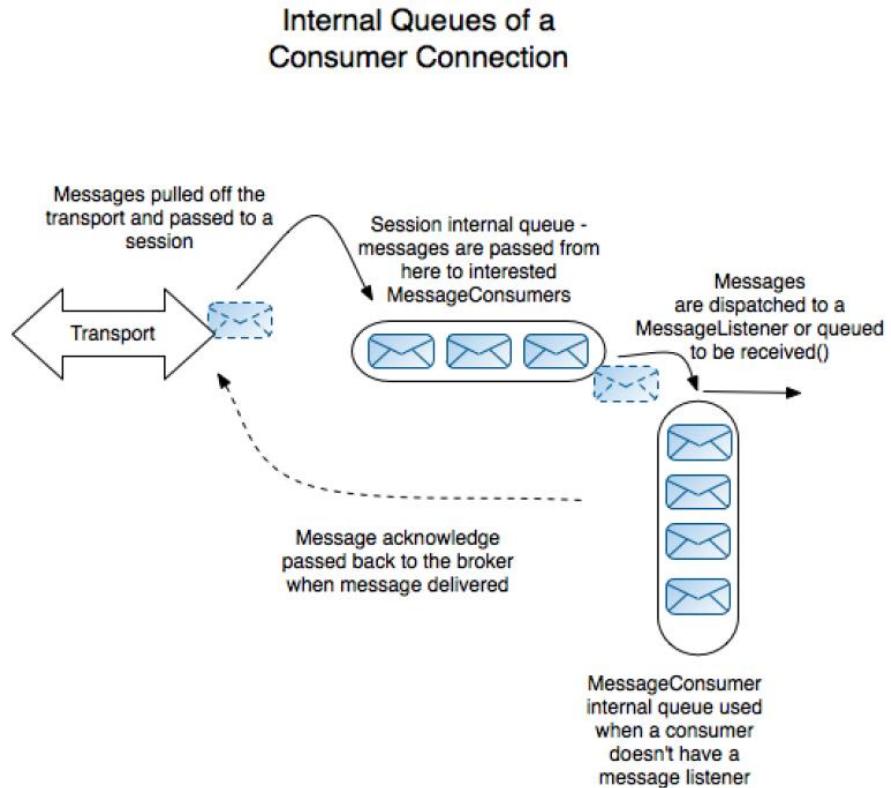
示例如下：

```
String url = "failover://(tcp://localhost:61616?tcpNoDelay=true)";
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory(url);
cf.setAlwaysSyncSend(true);
```

优化消息发送

优化消息消费者

消息消费的内部流程结构如下：



10.1.6. Prefetch Limit

ActiveMQ 默认的 prefetch 大小不同的：

Queue Consumer 默认大小=1000

Queue Browser Consumer 默认大小=500

Persistent Topic Consumer 默认大小=100

Non-persistent Topic Consumer 默认大小=32766

Prefecth policy 设置如下：

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();

Properties props = new Properties();
props.setProperty("prefetchPolicy.queuePrefetch", "1000");
props.setProperty("prefetchPolicy.queueBrowserPrefetch", "500");
props.setProperty("prefetchPolicy.durableTopicPrefetch", "100");
props.setProperty("prefetchPolicy.topicPrefetch", "32766");

cf.setProperties(props);
```



设置 prefetch policy 在 Destinations 上:

```
Queue queue = new ActiveMQQueue("TEST.QUEUE?consumer.prefetchSize=10");
MessageConsumer consumer = session.createConsumer(queue);
```

10.1.7. Delivery and Acknowledgement of messages

传递和回执消息。

建议使用 Session.DUPS_ACKNOWLEDGE。

JMS 消息只有在被确认之后，才认为已经被成功地消费了。消息的成功消费通常包含三个阶段：客户接收消息、客户处理消息和消息被确认。

在事务性会话中，当一个事务被提交的时候，确认自动发生。在非事务性会话中，消息何时被确认取决于创建会话时的应答模式(acknowledgement mode)。该参数有以下三个可选值：

- Session.AUTO_ACKNOWLEDGE。当客户成功的从 receive 方法返回的时候，或者从 MessageListener.onMessage 方法成功返回的时候，会话自动确认客户收到的消息。
- Session.TRANSACTION。用 session.commit() 回执确认。
- Session.CLIENT_ACKNOWLEDGE。客户通过消息的 acknowledge 方法确认消息。需要注意的是，在这种模式中，确认是在会话层上进行：确认一个被消费的消息将自动确认所有已被会话消费的消息。例如，如果一个消息消费者消费了 10 个消息，然后确认第 5 个消息，那么所有 10 个消息都被确认。
- Session.DUPS_ACKNOWLEDGE。该选择只是会话迟钝第确认消息的提交。当消息到达一定数量后，才开始消费该消息。如果 JMS provider 失败，那么可能会导致一些重复的消息。如果是重复的消息，那么 JMS provider 必须把消息头的 JMSRedelivered 字段设置为 true。

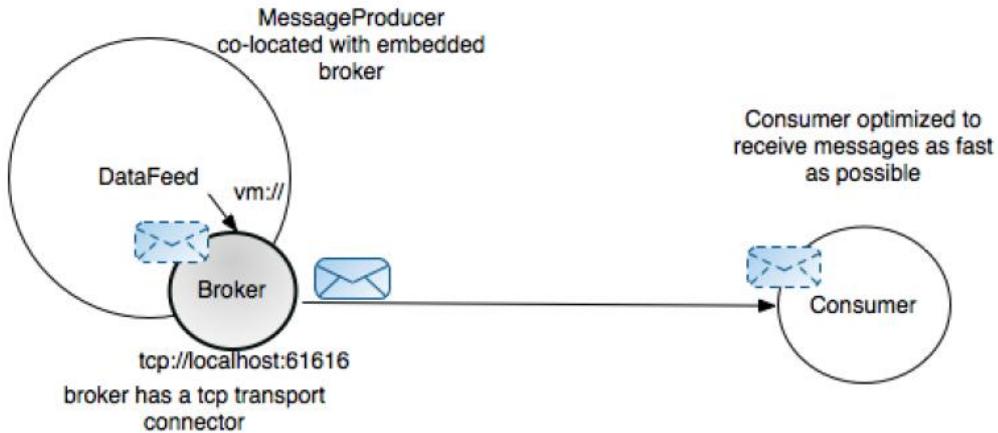
优化回执：

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setOptimizeAcknowledge(true);
```

超大批量快速发送到 broker

请参考：“快速回应消息”这一节。

运行原理如下：



创建 embeded broker:

```

import org.apache.activemq.broker.BrokerService;
import org.apache.activemq.broker.region.policy.PolicyEntry;
import org.apache.activemq.broker.region.policy.PolicyMap;
...
//By default a broker always listens on vm://<broker name>

BrokerService broker = new BrokerService();
broker.setBrokerName("fast");
broker.getSystemUsage().getMemoryUsage().setLimit(64*1024*1024);

//Set the Destination policies

PolicyEntry policy = new PolicyEntry();

//set a memory limit of 4mb for each destination

policy.setMemoryLimit(4 * 1024 *1024);

//disable flow control

policy.setProducerFlowControl(false);

PolicyMap pMap = new PolicyMap();

//configure the policy

pMap.setDefaultEntry(policy);

broker.setDestinationPolicy(pMap);
broker.addConnector("tcp://localhost:61616");
broker.start();

```

创建 Producer:



```
//tell the connection factory to connect to an embedded broker named fast.  
//if the embedded broker isn't already created, the connection factory will  
//create a default embedded broker named "fast"  
  
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("vm://fast");  
  
//disable message copying  
  
cf.setCopyMessageOnSend(false);  
  
Connection connection = cf.createConnection();  
connection.start();  
  
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);  
Topic topic = session.createTopic("test.topic");  
final MessageProducer producer = session.createProducer(topic);  
  
//send non-persistent messages  
  
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);  
for (int i =0; i < 1000000;i++) {  
    TextMessage message = session.createTextMessage("Test:"+i);  
    producer.send(message);  
}  
}
```

创建接收者：

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("failover://(tcp://localhost:61616)?jms.prefetchSize=32766&jms.reconnectInterval=5000");

//configure the factory to create connections
//with straight through processing of messages
//and optimized acknowledgement

cf.setAlwaysSessionAsync(false);
cf.setOptimizeAcknowledge(true);

Connection connection = cf.createConnection();
connection.start();

//use the default session
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//set the prefetch size for topics - by parsing a configuration parameter in
// the name of the topic

Topic topic = session.createTopic("test.topic?consumer.prefetchSize=32766");

MessageConsumer consumer = session.createConsumer(topic);

//setup a counter - so we don't print every message

final AtomicInteger count = new AtomicInteger();

consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            //only print every 10,000th message
            if (count.incrementAndGet()%10000==0)
                System.err.println("Got = " + textMessage.getText());
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
});
```



11. 管理和监控

JMX 支持

配置如下：

```
<broker brokerName="emv219" useJmx="true" xmlns="http://activemq.org/config/1.0">
...
<managementContext>
<managementContext connectorPort="1099" jmxDomainName="org.apache.activemq"/>
</managementContext>
...
</broker>
```

配置JXM 步骤如下：

1. 设置 broker 标识的useJmx 属性为true;
2. 取消对 managementContext 标识的注释（系统默认注释 managementContext 标

识），监控的默认端口为 1099。

Java 代码如下：

```
public class Stats {
    public static void main(String[] args) throws Exception {
        JMXServiceURL url = new JMXServiceURL(
            "service:jmx:rmi:///jndi/rmi://localhost:2011/jmxrmi");
        JMXConnector connector = JMXConnectorFactory.connect(url, null);
        connector.connect();
        MBeanServerConnection connection = connector.getMBeanServerConnection();❶

        ObjectName name = new ObjectName(
            "my-broker:BrokerName=localhost,Type=Broker");
        BrokerViewMBean mbean = (BrokerViewMBean) MBeanServerInvocationHandler
            .newProxyInstance(connection, name, BrokerViewMBean.class, true);❷

        System.out.println("Statistics for broker " + mbean.getBrokerId()
            + " - " + mbean.getBrokerName());
        System.out.println("\n-----\n");
        System.out.println("Total message count: " + mbean.getTotalMessageCount() + "\n");
        System.out.println("Total number of consumers: " + mbean.getTotalConsumerCount());
        System.out.println("Total number of Queues: " + mbean.getQueues().length);❸
    }
}
```



```

        for (ObjectName queueName : mbean.getQueues()) {
            QueueViewMBean queueMbean = (QueueViewMBean) MBeanServerInvocationHandler
                .newProxyInstance(connection, queueName,
                    QueueViewMBean.class, true);
            System.out.println("\n-----\n");
            System.out.println("Statistics for queue " + queueMbean.getName());
            System.out.println("Size: " + queueMbean.getQueueSize());
            System.out.println("Number of consumers: " + queueMbean.getConsumerCount());
        }
    }
}

```

日志插件

Xml 配置如下：

```

<broker useJmx="false" persistent="false"
xmlns="http://activemq.apache.org/schema/core">↓
↓
<plugins>↓
↓
<!-- lets enable detailed logging in the broker -->↓
<loggingBrokerPlugin/>↓
↓
</plugins>↓
</broker>↓

```

12. 附上 java 和 c++Client 示例：

JAVA 客户端示例：

```

import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.Connection;

import javax.jms.DeliveryMode;

import javax.jms.Destination;

import javax.jms.ExceptionListener;

```



```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

/**
 * Hello world!
 */

public class JavaClient {

    public static void main(String[] args) throws Exception {
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        Thread.sleep(1000);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        Thread.sleep(1000);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
    }
}
```



```
thread(new HelloWorldConsumer(), false);

thread(new HelloWorldProducer(), false);

thread(new HelloWorldProducer(), false);

Thread.sleep(1000);

thread(new HelloWorldProducer(), false);

thread(new HelloWorldConsumer(), false);

thread(new HelloWorldConsumer(), false);

thread(new HelloWorldProducer(), false);

}

public static void thread(Runnable runnable, boolean daemon) {

    Thread brokerThread = new Thread(runnable);

    brokerThread.setDaemon(daemon);

    brokerThread.start();

}

public static class HelloWorldProducer implements Runnable {

    public void run() {
```



```
try {

    // Create a ConnectionFactory

    ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory

("tcp://localhost:61616");

    // Create a Connection

    Connection connection = connectionFactory.createConnection();

    connection.start();

    // Create a Session

    Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

    // Create the destination (Topic or Queue)

    Destination destination = session.createQueue("TEST.FOO");

    // Create a MessageProducer from the Session to the Topic or Queue

    MessageProducer producer = session.createProducer(destination);

    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

    // Create a messages

    String text = "Hello world! From: " + Thread.currentThread().getName() + " :"
+ this.hashCode();

    TextMessage message = session.createTextMessage(text);

    // Tell the producer to send the message

    System.out.println("Sent message: " + message.hashCode() + " : " +
Thread.currentThread().getName());

    producer.send(message);

    // Clean up
```



```
        session.close();

        connection.close();

    }

    catch (Exception e) {

        System.out.println("Caught: " + e);

        e.printStackTrace();

    }

}

}

public static class HelloWorldConsumer implements Runnable,
ExceptionListener {

    public void run() {

        try {

            // Create a ConnectionFactory

            ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory

                ("tcp://localhost:61616");

            // Create a Connection

            Connection connection = connectionFactory.createConnection();

            connection.start();

            connection.setExceptionListener(this);

            // Create a Session

            Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
```



```
// Create the destination (Topic or Queue)

Destination destination = session.createQueue("TEST.FOO");

// Create a MessageConsumer from the Session to the Topic or Queue

MessageConsumer consumer = session.createConsumer(destination);

// Wait for a message

Message message = consumer.receive(1000);

if (message instanceof TextMessage) {

    TextMessage textMessage = (TextMessage) message;

    String text = textMessage.getText();

    System.out.println("Received: " + text);

} else {

    System.out.println("Received: " + message);

}

consumer.close();

session.close();

connection.close();

} catch (Exception e) {

    System.out.println("Caught: " + e);

    e.printStackTrace();

}

}

public synchronized void onException(JMSEException ex) {

    System.out.println("JMS Exception occurred. Shutting down client.");
}
```



```
}
```

```
}
```

```
}
```

C++客户端示例：

```
#include <activemq/concurrent/Thread.h>
#include <activemq/concurrent/Runnable.h>
#include <activemq/core/ActiveMQConnectionFactory.h>
#include <activemq/util/Integer.h>
#include <cms/Connection.h>
#include <cms/Session.h>
#include <cms/TextMessage.h>
#include <cms/ExceptionListener.h>
#include <cms/MessageListener.h>
#include <stdlib.h>
#include <iostream>

using namespace activemq::core;
using namespace activemq::util;
using namespace activemq::concurrent;
using namespace cms;
using namespace std;

class HelloWorldProducer : public Runnable {

private:
    Connection* connection;
    Session* session;
    Destination* destination;
    MessageProducer* producer;
    int numMessages;
    bool useTopic;

public:
    HelloWorldProducer( int numMessages, bool useTopic = false ) {
        connection = NULL;
        session = NULL;
        destination = NULL;
        producer = NULL;
        this->numMessages = numMessages;
        this->useTopic = useTopic;
    }
    virtual ~HelloWorldProducer() {
        cleanup();
    }
}
```



```
}

virtual void
connection = NULL;
session = NULL;
destination = NULL;
producer = NULL;
this->numMessages = numMessages;
this->useTopic = useTopic;
}

virtual ~HelloWorldProducer() {
cleanup();
}

virtual void run() {
try {
// Create a ConnectionFactory
ActiveMQConnectionFactory* connectionFactory = new
ActiveMQConnectionFactory("t
cp://127.0.0.1:61613");
// Create a Connection
connection = connectionFactory->createConnection();
connection->start();
// Create a Session
session =
connection->createSession( Session::AUTO_ACKNOWLEDGE );
// Create the destination (Topic or Queue)
if( useTopic ) {
destination = session->createTopic( "TEST.FOO" );
} else {
destination = session->createQueue( "TEST.FOO" );
}
// Create a MessageProducer from the Session to the Topic or Queue
producer = session->createProducer( destination );
producer->setDeliveryMode( DeliveryMode::NON_PERSISTANT );
// Create the Thread Id String
string threadIdStr = Integer::toString( Thread::getId() );
// Create a messages
string text = (string) "Hello world! from thread " + threadIdStr;
for( int ix=0; ix<numMessages; ++ix ){
TextMessage* message = session->createTextMessage( text );
// Tell the producer to send the message
printf( "Sent message from thread %s\n", threadIdStr.c_str() );
producer->send( message );
}
```



```
        delete message;
    }
} catch ( CMSEException& e ) {
    e.printStackTrace();
}
}

private:
void cleanup() {
    // Destroy resources.
    try{
        if( destination != NULL ) delete destination;
    } catch ( CMSEException& e ) {}
    destination = NULL;
    try{
        if( producer != NULL ) delete producer;
    } catch ( CMSEException& e ) {}
    producer = NULL;
    // Close open resources.
    try{
        if( session != NULL ) session->close();
        if( connection != NULL ) connection->close();
    } catch ( CMSEException& e ) {}
    try{
        if( session != NULL ) delete session;
    } catch ( CMSEException& e ) {}
    session = NULL;
    try{
        try{
            if( connection != NULL ) delete connection;
        } catch ( CMSEException& e ) {}
        connection = NULL;
    }
};

class HelloWorldConsumer : public ExceptionListener,
public MessageListener,
public Runnable {
private:
    Connection* connection;
    Session* session;
    Destination* destination;
    MessageConsumer* consumer;
    long waitMillis;
    bool useTopic;
```



```
public:
    HelloWorldConsumer( long waitMillis, bool useTopic =
false ) {
        connection = NULL;
        session = NULL;
        destination = NULL;
        consumer =
    }

    consumer = NULL;
    this->waitMillis = waitMillis;
    this->useTopic = useTopic;
}

virtual ~HelloWorldConsumer() {
    cleanup();
}

virtual void run() {
try {
    // Create a ConnectionFactory
    ActiveMQConnectionFactory* connectionFactory =
new
ActiveMQConnectionFactory( "tcp://127.0.0.1:61613" );
    // Create a Connection
    connection = connectionFactory->createConnection();
    delete connectionFactory;
    connection->start();
    connection->setExceptionListener(this);
    // Create a Session
    session =
connection->createSession( Session::AUTO_ACKNOWLEDGE );
    // Create the destination (Topic or Queue)
    if( useTopic ) {
        destination = session->createTopic( "TEST.FOO" );
    } else {
        destination = session->createQueue( "TEST.FOO" );
    }
    // Create a MessageConsumer from the Session to the
Topic or Queue
    consumer = session->createConsumer( destination );
    consumer->setMessageListener( this );
    // Sleep while asynchronous messages come in.
    Thread::sleep( waitMillis );
} catch (CMSEException& e) {
```



```
        e.printStackTrace();
    }
}

// Called from the consumer since this class is a
registered MessageListener.

virtual void onMessage( const Message* message ) {
    static int count = 0;
    try
    {
        count++;
        const TextMessage* textMessage =
        dynamic_cast< const TextMessage* >( message );
        string text = textMessage->getText();
        printf( "Message #%-d Received: %s\n", count,
text.c_str() );
    } catch (CMSEException& e) {
        e.printStackTrace();
    }
}

// If something bad happens you see it here as this
class is also been

// registered as an ExceptionListener with the
connection.

virtual void onException( const CMSEException& ex )
{
    printf("JMS Exception occurred. Shutting down
client.\n");
}

private:
void cleanup()

//*****
// Always close destination, consumers and
producers before
// you destroy their sessions and connection.

//*****
// Destroy resources.

try{
if( destination != NULL ) delete destination;
}catch (CMSEException& e) {}
destination = NULL;
try{
```



```
        if( consumer != NULL ) delete consumer;
    }catch (CMSEException& e) {}
    consumer = NULL;
    // Close open resources.
    try{
        if( session != NULL ) session->close();
        if( connection != NULL ) connection->close();
    }catch (CMSEException& e) {}
    // Now Destroy them
    try{
        if( session != NULL ) delete session;
    }catch (CMSEException& e) {}
    session = NULL;
    try{
        if( connection != NULL ) delete connection;
    }catch (CMSEException& e) {}
    connection = NULL;
    }
};

int main(int argc, char* argv[]) {
    std::cout <<
=====
    std::cout << "Starting the example:" <<
    std::endl;
    std::cout <<
-----
//=====
// set to true to use topics instead of
queues
// Note in the code above that this causes
createTopic or
// createQueue to be used in both consumer
an producer.

//=====
bool useTopics = false;
}
HelloWorldProducer producer( 1000,
useTopics );
HelloWorldConsumer consumer( 8000,
useTopics );
```



```
// Start the consumer thread.  
Thread consumerThread( &consumer );  
consumerThread.start();  
// Start the producer thread.  
Thread producerThread( &producer );  
producerThread.start();  
// Wait for the threads to complete.  
producerThread.join();  
consumerThread.join();  
std::cout <<  
"-----\n";  
std::cout << "Finished with the example,  
ignore errors from this"  
<< std::endl  
<< "point on as the sockets breaks when we  
shutdown."  
<< std::endl;  
std::cout <<  
"=====\\n";  
}  
}
```