

1 JMS

在介绍 ActiveMQ 之前，首先简要介绍一下 JMS 规范。

1. 1 JMS 的基本构件

1. 1. 1 连接工厂

连接工厂是客户用来创建连接的对象，例如 ActiveMQ 提供的 `ActiveMQConnectionFactory`。

1. 1. 2 连接

`JMS Connection` 封装了客户与 JMS 提供者之间的一个虚拟的连接。

1. 1. 3 会话

`JMS Session` 是生产和消费消息的一个单线程上下文。会话用于创建消息生产者（producer）、消息消费者（consumer）和消息（message）等。会话提供了一个事务性的上下文，在这个上下文中，一组发送和接收被组合到了一个原子操作中。

1. 1. 4 目的地

目的地是客户用来指定它生产的消息的目标和它消费的消息的来源的对象。`JMS1.0.2` 规范中定义了两种消息传递域：点对点（PTP）消息传递域和发布/订阅消息传递域。

点对点消息传递域的特点如下：

- 每个消息只能有一个消费者。
- 消息的生产者和消费者之间没有时间上的相关性。无论消费者在生产者发送消息的时候是否处于运行状态，它都可以提取消息。

发布/订阅消息传递域的特点如下：

- 每个消息可以有多个消费者。
- 生产者和消费者之间有时间上的相关性。订阅一个主题的消费者只能消费自它订阅之后发布的消息。`JMS` 规范允许客户创建持久订阅，这在一定程度上放松了时间上的相关性要求。持久订阅允许消费者消费它在未处于激活状态时发送的消息。

在点对点消息传递域中，目的地被成为队列（queue）；在发布/订阅消息传递域中，目的地被成为主题（topic）。

1. 1. 5 消息生产者

消息生产者是由会话创建的一个对象，用于把消息发送到一个目的地。

1. 1. 6 消息消费者

消息消费者是由会话创建的一个对象，它用于接收发送到目的地的消息。消息的消费可以采用以下两种方法之一：

- 同步消费。通过调用消费者的 `receive` 方法从目的地中显式提取消息。`receive` 方法可以一直阻塞到消息到达。
- 异步消费。客户可以为消费者注册一个消息监听器，以定义在消息到达时所采取的动作。

1. 1. 7 消息

JMS 消息由以下三部分组成：

- 消息头。每个消息头字段都有相应的 `getter` 和 `setter` 方法。
- 消息属性。如果需要除消息头字段以外的值，那么可以使用消息属性。
- 消息体。JMS 定义的消息类型有 `TextMessage`、`MapMessage`、`BytesMessage`、`StreamMessage` 和 `ObjectMessage`。

1. 2 JMS 的可靠性机制

1. 2. 1 确认

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.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。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`。

1. 2. 2 持久性

JMS 支持以下两种消息提交模式：

- `PERSISTENT`。指示 JMS provider 持久保存消息，以保证消息不会因为 JMS provider 的失败而丢失。
- `NON_PERSISTENT`。不要求 JMS provider 持久保存消息。

1. 2. 3 优先级

可以使用消息优先级来指示 JMS provider 首先提交紧急的消息。优先级分 10 个级别，从 0（最低）到 9（最高）。如果不指定优先级，默认级别是 4。需要注意的是，JMS provider 并不一定保证按照优先级的顺序提交消息。

1. 2. 4 消息过期

可以设置消息在一定时间后过期，默认是永不过期。

1. 2. 5 临时目的地

可以通过会话上的 `createTemporaryQueue` 方法和 `createTemporaryTopic` 方法来创建临时目的地。它们的存在时间只限于创建它们的连接所保持的时间。只有创建该临时目的地的连接上的消息消费者才能够从临时目的地中提取消息。

1. 2. 6 持久订阅

首先消息生产者必须使用 PERSISTENT 提交消息。客户可以通过会话上的 `createDurableSubscriber` 方法来创建一个持久订阅，该方法的第一个参数必须是一个 topic。第二个参数是订阅的名称。

JMS provider 会存储发布到持久订阅对应的 topic 上的消息。如果最初创建持久订阅的客户或者任何其它客户使用相同的连接工厂和连接的客户 ID、相同的主题和相同的订阅名再次调用会话上的 `createDurableSubscriber` 方法，那么该持久订阅就会被激活。JMS provider 会象客户发送客户处于非激活状态时所发布的消息。

持久订阅在某个时刻只能有一个激活的订阅者。持久订阅在创建之后会一直保留，直到应用程序调用会话上的 `unsubscribe` 方法。

1. 2. 7 本地事务

在一个 JMS 客户端，可以使用本地事务来组合消息的发送和接收。JMS Session 接口提供了 `commit` 和 `rollback` 方法。事务提交意味着生产的所有消息被发送，消费的所有消息被确认；事务回滚意味着生产的所有消息被销毁，消费的所有消息被恢复并重新提交，除非它们已经过期。

事务性的会话总是牵涉到事务处理中，`commit` 或 `rollback` 方法一旦被调用，一个事务就结束了，而另一个事务被开始。关闭事务性会话将回滚其中的事务。需要注意的是，如果使用请求/回复机制，即发送一个消息，同时希望在同一个事务中等待接收该消息的回复，那么程序将被挂起，因为知道事务提交，发送操作才会真正执行。

需要注意的还有一个，消息的生产和消费不能包含在同一个事务中。

1. 3 JMS 规范的变迁

JMS 的最新版本的是 1.1。它和同 1.0.2 版本之间最大的差别是，JMS1.1 通过统一的消息传递域简化了消息传递。这不仅简化了 JMS API，也有利于开发人

员灵活选择消息传递域，同时也有助于程序的重用和维护。
 以下是不同消息传递域的相应接口：

JMS 公共	点对点域	发布/订阅域
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

2 ActiveMQ

2. 1 Broker

2. 1. 1 Running Broker

ActiveMQ5.0 的二进制发布包中 bin 目录中包含一个名为 `activemq` 的脚本，直接运行这个脚本就可以启动一个 broker。


此外也可以通过 Broker Configuration URI 或 Broker XBean URI 对 broker 进行配置，以下是一些命令行参数的例子：

Example	Description
<code>activemq</code>	Runs a broker using the default 'xbean:activemq.xml' as the broker configuration file.
<code>activemq xbean:myconfig.xml</code>	Runs a broker using the file <code>myconfig.xml</code> as the broker configuration file that is located in the classpath.
<code>activemq xbean:file:./conf/broker1.xml</code>	Runs a broker using the file <code>broker1.xml</code> as the broker configuration file that is located in the relative file path <code>./conf/broker1.xml</code>
<code>activemq xbean:file:C:/ActiveMQ/conf/broker2.xml</code>	Runs a broker using the file <code>broker2.xml</code> as the broker configuration file that is located in the absolute

	file path C:/ActiveMQ/conf/broker2.xml
activemq broker:(tcp://localhost:61616, tcp://localhost:5000)?useJmx=true	Runs a broker with two transport connectors and JMX enabled.
activemq broker:(tcp://localhost:61616, network:tcp://localhost:5000)?persistent=false	Runs a broker with 1 transport connector and 1 network connector with persistence disabled.


2. 1. 2 Embedded Broker

可以通过在应用程序中以编码的方式启动 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();

如果希望在同一个 JVM 内访问这个 broker，那么可以使用 VM Transport，URI 是：vm://brokerName。关于更多的 broker 属性，可以参考 Apache 的官方文档。

此外，也可以通过 BrokerFactory 来创建 broker，例如：

Java 代码 


1. BrokerService broker = BrokerFactory.createBroker(new URI(someURI));

someURI 的可选值如下：

URI scheme	Example	Description
xbean:	xbean:activemq.xml	Searches the classpath for an XML


		document with the given URI (activemq.xml in this case) which will then be used as the Xml Configuration
file:	file:foo/bar/activemq.xml	Loads the given file (in this example foo/bar/activemq.xml) as the Xml Configuration
broker:	broker:tcp://localhost:61616	Uses the Broker Configuration URI to configure the broker

当使用 XBean 的配置方式的时候，需要指定一个 xml 配置文件，例如：

Java 代码 

```
1. BrokerService broker = BrokerFactory.createBroker(new URI("xbean:com/test/activemq.xml"));
```

使用 Spring 的配置方式如下：


Xml 代码 

```
1. <bean id="broker" class="org.apache.activemq.xbean.BrokerFactoryBean">
2.   <property name="config" value="classpath:org/apache/activemq/xbean/activemq.xml" />
3.   <property name="start" value="true" />
4. </bean>
```

2. 1. 3 Monitoring Broker

2. 1. 3. 1 JMX

在使用 JMX 监控 broker 之前，首先要启用 broker 的 JMX 监控功能，例如在配置文件中设置 useJmx="true"，如下：

Xml 代码 

```
1. <broker useJmx="true" brokerName="broker1">
2.   <managementContext>
3.     <managementContext createConnector="true"/>
4.   </managementContext>
5.   ...
6. </broker>
```

接下来运行 JDK 自带的 jconsole。在运行了 jconsole 后，它会弹出对话框来选择需要连接到的 agent。如果是在启动 broker 的主机上运行 jconsole，那么 ActiveMQ broker 会出现在 jconsole 的 Local 标签中。如果要连接到远程的 broker，那么可以在 Advanced 标签中指定 JMX URL，以下是一个连接到本机的 JMX URL：

```
service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
```

在 jconsole 的 MBeans 标签中，可以查看详细信息，也可以执行相应的 operation。需要注意的是，在 jconsole 连接到 broker 的时候，并不需要输入用户名和密码，如果这存在潜在的安全问题，那么就需要为 JMX Connector 配置密码保护（需要使用 1.5 以上版本的 JDK）。

首先要禁止 ActiveMQ 创建自己的 connector，例如：

Xml 代码 

```
1. <broker xmlns="http://activemq.org/config/1.0" brokerName="localhost" useJmx="true">
2.   <managementContext>
3.     <managementContext createConnector="false"/>
4.   </managementContext>
5. </broker>
```

然后在 ActiveMQ 的 conf 目录下创建一个访问控制文件和密码文件，如下：
conf/jmx.access:

```
# The "monitorRole" role has readonly access.
# The "controlRole" role has readwrite access.
monitorRole readonly
controlRole readwrite
```

conf/jmx.password:

```
# The "monitorRole" role has password "abc123".
# The "controlRole" role has password "abcd1234".
monitorRole abc123
controlRole abcd1234
```

然后修改 ActiveMQ 的 bin 目录下 activemq 的启动脚本，查找包含“SUNJMX=”的一行如下：

```
REM set SUNJMX=-Dcom.sun.management.jmxremote.port=1616
-Dcom.sun.management.jmxremote.authenticate=false
```

```
-Dcom.sun.management.jmxremote.ssl=false
    把它替换成
set SUNJMX=-Dcom.sun.management.jmxremote.port=1616
-Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.password.file=%ACTIVEMQ_BASE%/conf/jmx.
password
-Dcom.sun.management.jmxremote.access.file=%ACTIVEMQ_BASE%/conf/jmx.a
ccess
```

最后重启 ActiveMQ 和 jconsole，这时候需要强制 login。如果在启动 activemq 的过程中出现以下错误，那么需要为这个文件增加访问控制。Windows 平台上的具体解决方法请参考如下网址：

<http://java.sun.com/j2se/1.5.0/docs/guide/management/security-windows.html>

Error: Password file read access must be restricted:

D:\apache-activemq-5.0.0\bin\../conf/jmx.password

2. 1. 3. 2 Web Console

Web Console 被集成到了 ActiveMQ 的二进制发布包中，因此缺省访问 <http://localhost:8161/admin> 即可访问 Web Console。

在配置文件中，可以通过修改 nioConnector 的 port 属性来修改 Web console 的缺省端口：

Xml 代码 

```
1. <jetty xmlns="http://mortbay.com/schemas/jetty/1.0">
2.   <connectors>
3.     <nioConnector port="8161" />
4.   </connectors>
5.   ...
6. </jetty>
```

出于安全性或者可靠性的考虑，Web Console 可以被部署到不同于 ActiveMQ 的进程中。例如把 activemq-web-console.war 部署到一个单独的 web 容器中（Tomcat，Jetty 等）。在 ActiveMQ5.0 的二进制发布包中不包含 activemq-web-console.war，因此需要下载 ActiveMQ 的源码，然后进入到 `${activemq.base}/src/activemq-web-console` 目录中执行 `mvn install`。如果一切正常，那么缺省会在 `${activemq.base}/src/activemq-web-console/target` 目录中生成 `activemq-web-console-5.0.0.war`。然后将 `activemq-web-console-5.0.0.war` 拷贝到 Tomcat 的 webapps 目录中，并重命名成 `activemq-web-console.war`。

需要注意的是,要将activemq-all-5.0.0.jar拷贝到WEB-INF\lib目录中(可能还需要拷贝jms.jar)。还要为Tomcat设置以下五个系统属性(修改catalina.bat文件):

```
set JAVA_OPTS=%JAVA_OPTS% -Dwebconsole.type="properties"
set JAVA_OPTS=%JAVA_OPTS% -Dwebconsole.jms.url="tcp://localhost:61616"
set JAVA_OPTS=%JAVA_OPTS%
-Dwebconsole.jmx.url="service:jmx:rmi:///jndi/rmi://localhost:1099/jm
xrmi"
set JAVA_OPTS=%JAVA_OPTS% -Dwebconsole.jmx.role=""
set JAVA_OPTS=%JAVA_OPTS% -Dwebconsole.jmx.password=""
```

如果JMX没有配置密码保护,那么webconsole.jmx.role和webconsole.jmx.password设置成""即可。如果broker被配置成了Master/Slave模式,那么可以配置成使用failover transport,例如:

```
-Dwebconsole.jms.url=failover:(tcp://serverA:61616,tcp://serverB:61616)
```

顺便说一下,由于webconsole.type属性是properties,因此实际上起作用的Web Console的配置文件是WEB-INF/webconsole-properties.xml。最后启动被监控的ActiveMQ,访问http://localhost:8080/activemq-web-console/,查看显示是否正常。

2. 1. 3. 3 Advisory Message

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

Advisory Topics	Description
ActiveMQ.Advisory.Connection	Connection start & stop messages

ActiveMQ.Advisory.Producer.Queue	Producer start & stop messages on a Queue
ActiveMQ.Advisory.Producer.Topic	Producer start & stop messages on a Topic
ActiveMQ.Advisory.Consumer.Queue	Consumer start & stop messages on a Queue
ActiveMQ.Advisory.Consumer.Topic	Consumer start & stop messages on a Topic

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。

Destination and Message based advisories

Advisory Topics	Description
ActiveMQ.Advisory.Queue	Queue create & destroy
ActiveMQ.Advisory.Topic	Topic create & destroy
ActiveMQ.Advisory.TempQueue	Temporary Queue create & destroy
ActiveMQ.Advisory.TempTopic	Temporary Topic create & destroy
ActiveMQ.Advisory.Expired.Queue	Expired messages on a Queue
ActiveMQ.Advisory.Expired.Topic	Expired messages on a Topic
ActiveMQ.Advisory.NoConsumer.Queue	No consumer is available to process messages being sent on a Queue
ActiveMQ.Advisory.NoConsumer.Topic	No consumer is available to process messages being sent on a Topic

以上的这些 destnations 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、producderID 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.>。

org.apache.activemq.advisory.AdvisorySupport 类上有如下的 helper methods，用来在程序中得到 advisory destination objects。

Java 代码

1. `AdvisorySupport.getConsumerAdvisoryTopic()`
2. `AdvisorySupport.getProducerAdvisoryTopic()`
3. `AdvisorySupport.getDestinationAdvisoryTopic()`
4. `AdvisorySupport.getExpiredTopicMessageAdvisoryTopic()`
5. `AdvisorySupport.getExpiredQueueMessageAdvisoryTopic()`
6. `AdvisorySupport.getNoTopicConsumersAdvisoryTopic()`
7. `AdvisorySupport.getNoQueueConsumersAdvisoryTopic()`

以下是段使用 Advisory Messages 的程序代码:

Java 代码

1. `Destination advisoryDestination = AdvisorySupport.getProducerAdvisoryTopic(destination)`
2. `MessageConsumer consumer = session.createConsumer(advisoryDestination);`
3. `consumer.setMessageListener(this);`
4. `...`
5. `public void onMessage(Message msg) {`
6. `if (msg instanceof ActiveMQMessage) {`
7. `try {`
8. `ActiveMQMessage aMsg = (ActiveMQMessage)msg;`
9. `ProducerInfo prod = (ProducerInfo) aMsg.getDataStructure();`
10. `} catch (JMSEException e) {`
11. `log.error("Failed to process message: " + msg);`
12. `}`
13. `}`
14. `}`

2. 1. 3. 4 Command Agent

在介绍 Command Agent 前首先简要介绍一下 XMPP(Jabber)协议, XMPP 是一种基于 XML 的即时通信协议, 它由 Jabber 软件基金会开发。在配置文件中通过增加 `transportConnector` 来支持 XMPP 协议:

Xml 代码

1. `<broker xmlns="http://activemq.org/config/1.0">`
2. `<transportConnectors>`
3. `...`

4. `<transportConnector name="xmpp" uri="xmpp://localhost:61222"/>`
5. `</transportConnectors>`
6. `</broker>`

ActiveMQ 提供了 ActiveMQ messages 和 XMPP 之间的双向桥接:

- 如果客户加入了一个聊天室, 那么这个聊天室的名字会被映射到一个 JMS topic。
- 尝试在聊天室内发送消息会导致一个 JMS 消息被发送到这个 topic。
- 呆在一个聊天室中意味着这将保持一个对相应 JMS topic 的订阅。因此发送到这个 topic 的 JMS 消息也会被发送到聊天室。

推荐 XMPP 客户端 Spark(<http://www.igniterealtime.org/>)。

从 4.2 版本起, ActiveMQ 支持 Command Agent。在配置文件中, 通过设置 commandAgent 来启用 Command Agent:

Xml 代码 

1. `<beans>`
2. `<broker useJmx="true" xmlns="http://activemq.org/config/1.0">`
3. `...`
4. `</broker>`
5. `<commandAgent xmlns="http://activemq.org/config/1.0"/>`
6. `</beans>`

启用了 Command Agent 的 broker 上会有一个来自 Command Agent 的连接, 它同时订阅 topic: ActiveMQ.Agent。在你启动 XMPP 客户端, 加入到 ActiveMQ.Agent 聊天室后, 就可以同 broker 进行交谈了。通过在 XMPP 客户端中键入 help, 可以得到帮助信息。

需要注意的是, ActiveMQ5.0 版本有个小 bug, 如果 broker 没有采用缺省的用户名和密码, 那么 Command Agent 便无法正常启动。Apache 官方文档说, 此 bug 已经被修正, 预定在 5.2.0 版本上体现。修改方式如下:


Xml 代码 

1. `<commandAgent xmlns="http://activemq.org/config/1.0" brokerUser="user" brokerPassword="password"/>`

2. 1. 3. 5 Visualization plugin

ActiveMQ 支持以 broker 插件的形式生成 DOT 文件(可以用 agrviewer 来查

看)，以图表的方式描述 connections、sessions、producers、consumers、destinations 等信息。配置方式如下：

Xml 代码 

```
1. <broker xmlns="http://activemq.org/config/1.0" brokerName="localhost" useJmx="true">
2.     ...
3.     <plugins>
4.         <connectionDotFilePlugin file="connection.dot"/>
5.         <destinationDotFilePlugin file="destination.dot"/>
6.     </plugins>
7. </broker>
```

需要注意的是，笔者认为 ActiveMQ5.0 版本的 Visualization Plugin 尚不稳定，存在诸多问题。例如：如果使用 connectionDotFilePlugin，那么 brokerName 必须是 localhost；如果使用 destinationDotFilePlugin 可能会导致 ArrayStoreException。

2. 2 Transport

ActiveMQ 目前支持的 transport 有：VM Transport、TCP Transport、SSL Transport、Peer Transport、UDP Transport、Multicast Transport、HTTP and HTTPS Transport、Failover Transport、Fanout Transport、Discovery Transport、ZeroConf Transport 等。以下简单介绍其中的几种，更多请参考 Apache 官方文档。

2. 2. 1 VM Transport

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。

以下是配置语法：

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

```
例如：vm://broker1?marshal=false&broker.persistent=false
```

Transport Options 的可选值如下：

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

Marshal	false	If true, forces each command sent over the transport to be marshlled and unmarshlled 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 allready 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

以下是高级配置语法：

```
vm:(broker:(tcp://localhost)?brokerOptions)?transportOptions
```

```
vm:broker:(tcp://localhost)?brokerOptions
```

例如：

```
vm:(broker:(tcp://localhost:6000)?persistent=false)?marshal=false
```

Transport Options 的可选值如下：

Option Name	Default Value	Description
marshal	false	If true, forces each command sent over the transport to be marshlled and unmarshlled 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

使用配置文件的配置语法：

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

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

使用 Spring 的配置:

Xml 代码 

1. `<bean id="broker" class="org.apache.activemq.xbean.BrokerFactoryBean">`
2. `<property name="config" value="classpath:org/apache/activemq/xbean/activemq.xml" />`
3. `<property name="start" value="true" />`
4. `</bean>`
- 5.
6. `<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory" depends-on="broker">`
7. `<property name="brokerURL" value="vm://localhost"/>`
8. `</bean>`

如果 persistent 是 true, 那么 ActiveMQ 会在当前目录下创建一个缺省值是 activemq-data 的目录用于持久化保存数据。需要注意的是, 如果程序中启动了多个不同名字的 VM broker, 那么可能会有如下警告: Failed to start jmx connector: Cannot bind to URL [rmi://localhost:1099/jmxrmi]: javax.naming.NameAlreadyBoundException...可以通过在 transportOptions 中追加 broker.useJmx=false 来禁用 JMX 来避免这个警告。

2. 2. 2 TCP Transport

TCP transport 允许客户端通过 TCP socket 连接到远程的 broker。以下是配置语法:

tcp://hostname:port?transportOptions

Transport Options 的可选值如下:

Option Name	Default Value	Description
minnumWireFormatVersion	0	The minimum version wireformat that is allowed
trace	false	Causes all commands that are sent over the transport to be logged
useLocalHost	true	When true, it causes the local machines name to resolve to "localhost".
socketBufferSize	64 * 1024	Sets the socket buffer size in bytes
soTimeout	0	sets the socket timeout in milliseconds
connectionTimeout	30000	A non-zero value specifies the connection timeout in milliseconds. A

		zero value means wait forever for the connection to be established. Negative values are ignored.
wireFormat	default	The name of the WireFormat to use
wireFormat.*		All the properties with this prefix are used to configure the wireFormat. See Configuring Wire Formats for more information

例如: tcp://localhost:61616?trace=false

2. 2. 3 Failover Transport

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。以下是配置语法:

failover:(uril,...,uriN)?transportOptions

failover:uril,...,uriN

Transport Options 的可选值如下:

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 between 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 URI to use for reconnect from the list provided
backup	false	initialize and hold a second transport connection - to enable fast failover

例如：

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

2. 2. 4 Discovery transport

Discovery transport 是可靠的 transport。它使用 Discovery transport 来定位用来连接的 URI 列表。以下是配置语法：

```
discovery:(discoveryAgentURI)?transportOptions
```

```
discovery:discoveryAgentURI
```

Transport Options 的可选值如下：

Option Name	Default Value	Description
initialReconnectDelay	10	How long to wait before the first reconnect attempt
maxReconnectDelay	30000	The maximum amount of time we ever wait between reconnect attempts
useExponentialBackOff	true	Should an exponential backoff be used between 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

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

为了使用 Discovery 来发现 broker，需要为 broker 启用 discovery agent。以下是 XML 配置文件中的一个例子：


Xml 代码 

```
1. <broker name="foo">
2.   <transportConnectors>
3.     <transportConnector uri="tcp://localhost:0" discoveryUri="
   multicast://default"/>
4.   </transportConnectors>
5.   ...
6. </broker>
```

在使用 Failover Transport 或 Discovery transport 等能够自动重连的 transport 的时候，需要注意的是：设想有两个 broker，它们都启用 AMQ Message

Store 作为持久化存储，有一个 producer 和一个 consumer 连接到某个 queue。当因其中一个 broker 失效时而切换到另一个 broker 的时候，如果失效的 broker 的 queue 中还有未被 consumer 消费的消息，那么这个 queue 里的消息仍然滞留在失效 broker 的中，直到失效的 broker 被修复并重新切换回这个被修复的 broker 后，之前被保留的消息才会被 consumer 消费掉。如果被处理的消息有时序限制，那么应用程序就需要处理这个问题。另外也可以通过 ActiveMQ 集群来解决这个问题。

在 transport 重连的时候，可以在 connection 上注册 TransportListener 来获得回调，例如：


Java 代码 

```
1. (ActiveMQConnection)connection).addTransportListener(new TransportListener() {
2.     public void onCommand(Object cmd) {
3.     }
4.
5.     public void onException(IOException exp) {
6.     }
7.
8.     public void transportInterrupted() {
9.         // The transport has suffered an interruption from which it hopes to recover.
10.    }
11.
12.    public void transportResumed() {
13.        // The transport has resumed after an interruption.
14.    }
15.});
```

2. 3 Persistence

2. 3. 1 AMQ Message Store

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 


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

Property name	Default value	Comments
directory	activemq-data	the path to the directory to use to store the message store data and log files
useNIO	true	use NIO to write messages to the data logs
syncOnWrite	false	sync every write to disk
maxFileLength	32mb	a hint to set the maximum size of the message data logs
persistentIndex	true	use a persistent index for the message logs. If this is false, an in-memory structure is maintained
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
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

2. 3. 2 Kaha Persistence

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" maxDa
   taFileLength="33554432"/>
4.     </persistenceAdapter>
5. </broker>

```


2. 3. 3 JDBC Persistence

目前支持的数据库有 Apache Derby, Axion, DB2, HSQL, Informix, MaxDB, MySQL, Oracle, Postgresql, SQLServer, Sybase。

如果你使用的数据库不被支持，那么可以调整 StatementProvider 来保证使用正确的 SQL 方言 (flavour of SQL)。通常绝大多数数据库支持以下 adaptor：

- org.activemq.store.jdbc.adapter.BlobJDBCAdapter
- org.activemq.store.jdbc.adapter.BytesJDBCAdapter
- org.activemq.store.jdbc.adapter.DefaultJDBCAdapter
- org.activemq.store.jdbc.adapter.ImageJDBCAdapter

也可以在配置文件中直接指定 JDBC adaptor，例如：


Xml 代码 

```

1. <jdbcPersistenceAdapter adapterClass="org.apache.activemq.stor
   e.jdbc.adapter.ImageBasedJDBCAdapter"/>

```

以下是其配置的一个例子：

Xml 代码 

```
1. <persistence>
2.     <jdbcPersistence dataSourceRef="mysql-ds"/>
3. </persistence>
4.
5. <bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSou
   rce" destroy-method="close">
6.     <property name="driverClassName" value="com.mysql.jdbc.Dr
   iver"/>
7.     <property name="url" value="jdbc:mysql://localhost/active
   mq?relaxAutoCommit=true"/>
8.     <property name="username" value="activemq"/>
9.     <property name="password" value="activemq"/>
10.    <property name="poolPreparedStatements" value="true"/>
11.</bean>
```

需要注意的是，如果使用 MySQL，那么需要设置 relaxAutoCommit 标志为 true。

2. 3. 4 Disable Persistence

以下是其配置的一个例子：

Xml 代码 


```
1. <broker persistent="false">
2. </broker>
```

2. 4 Security

ActiveMQ 支持可插拔的安全机制，用以在不同的 provider 之间切换。

2. 4. 1 Simple Authentication Plugin

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。以下是 ActiveMQ 配置的一个例子：

Xml 代码 

```
1. <plugins>
2.     ...
3.     <simpleAuthenticationPlugin>
4.         <users>
5.             <authenticationUser username="system" password="manager"
   groups="users, admins"/>
```

```

6.      <authenticationUser username="user" password="password" g
roups="users"/>
7.      <authenticationUser username="guest" password="password"
groups="guests"/>
8.      </users>
9.      </simpleAuthenticationPlugin>
10. </plugins>

```

2. 4. 2 JAAS Authentication Plugin

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.pr
operties"           org.apache.activemq.jaas.properties.group="groups.p
roperties";
};

```

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：

Xml 代码 

```

1. <plugins>
2.   ...
3.   <jasAuthenticationPlugin configuration="activemq-domain" />
4. </plugins>


```

基于以上的配置，在 JAAS 的 `LoginContext` 中会使用 `activemq-domain` 中配置的 `PropertiesLoginModule` 来进行登陆。

ActiveMQ JAAS 还支持 `LDAPLoginModule`、`CertificateLoginModule`、`TextFileCertificateLoginModule` 等 login module。

2. 4. 3 Custom Authentication Implementation

可以通过编码的方式为 ActiveMQ 增加认证功能。例如编写一个类继承自 XBeanBrokerService。

Java 代码 

```
1. package com.yourpackage;
2.
3. import java.net.URI;
4. import java.util.HashMap;
5. import java.util.Map;
6.
7. import org.apache.activemq.broker.Broker;
8. import org.apache.activemq.broker.BrokerFactory;
9. import org.apache.activemq.broker.BrokerService;
10. import org.apache.activemq.security.SimpleAuthenticationBroker;
11.
12. import org.apache.activemq.xbean.XBeanBrokerService;
13.
14. public class SimpleAuthBroker extends XBeanBrokerService {
15.     //
16.     private String user;
17.     private String password;
18.
19.     @SuppressWarnings("unchecked")
20.     protected Broker addInterceptors(Broker broker) throws Exception {
21.         broker = super.addInterceptors(broker);
22.         Map passwords = new HashMap();
23.         passwords.put(getUser(), getPassword());
24.         broker = new SimpleAuthenticationBroker(broker, passwords, new HashMap());
25.         return broker;
26.     }
27.
28.     public String getUser() {
29.         return user;
30.     }
31.
32.     public void setUser(String user) {
33.         this.user = user;
34.     }
35. }
```

```

35.     public String getPassword() {
36.         return password;
37.     }
38.
39.     public void setPassword(String password) {
40.         this.password = password;
41.     }
42. }

```

以下是 ActiveMQ 配置文件的一个例子：

Xml 代码 

```

1. <beans>
2.   ...
3.   <auth:SimpleAuthBroker
4.     xmlns:auth="java://com.yourpackage"
5.     xmlns="http://activemq.org/config/1.0" brokerName="SimpleAuthBroker1" user="user" password="password" useJmx="true">
6.
7.     <transportConnectors>
8.       <transportConnector uri="tcp://localhost:61616"/>
9.     </transportConnectors>
10.  </auth:SimpleAuthBroker>
11.  ...
12. </beans>

```

在这个配置文件中增加了一个 namespace auth, 用于指向之前编写的哪个类。同时为 SimpleAuthBroker 注入了两个属性值 user 和 password, 因此在被 SimpleAuthBroker 改写的 addInterceptors 方法里, 可以使用这两个属性进行认证了。ActiveMQ 提供的 SimpleAuthenticationBroker 类继承自 BrokerFilter (可以简单的看成是 Broker 的 Adaptor), 它的构造函数中的两个 Map 分别是 userPasswords 和 userGroups。 SimpleAuthenticationBroker 在 addConnection 方法中使用 userPasswords 进行认证, 同时会把 userGroups 的信息保存到 ConnectionContext 中。

2. 4. 4 Authorization Plugin

可以通过 Authorization Plugin 为认证后的用户授权, 以下 ActiveMQ 配置文件的一个例子：

Xml 代码 

```

1. <plugins>
2.   <jaasAuthenticationPlugin configuration="activemq-domain"/>

```



```

3.
4.     <authorizationPlugin>
5.         <map>
6.             <authorizationMap>
7.                 <authorizationEntries>
8.                     <authorizationEntry queue="" read="admins" write="ad
mins" admin="admins" />
9.                     <authorizationEntry queue="USERS.>" read="users" writ
e="users" admin="users" />
10.                    <authorizationEntry queue="GUEST.>" read="guests" wri
te="guests,users" admin="guests,users" />
11.
12.                    <authorizationEntry topic="" read="admins" write="ad
mins" admin="admins" />
13.                    <authorizationEntry topic="USERS.>" read="users" writ
e="users" admin="users" />
14.                    <authorizationEntry topic="GUEST.>" read="guests" wri
te="guests,users" admin="guests,users" />
15.
16.                    <authorizationEntry topic="ActiveMQ.Advisory.>" read=
"guests,users" write="guests,users" admin="guests,users"/>
17.                </authorizationEntries>
18.            </authorizationMap>
19.        </map>
20.    </authorizationPlugin>
21.</plugins>

```


2. 5 Clustering

ActiveMQ 从多种不同的方面提供了集群的支持。

2. 5. 1 Queue consumer clusters

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 ActiveMQ 5.0 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="Bu
gBroker1" useJmx="true">
3.
4.         <transportConnectors>

```

```

5.     <transportConnector uri="tcp://localhost:61616"/>
6. </transportConnectors>
7.
8.     <persistenceAdapter>
9.         <amqpPersistenceAdapter directory="activemq-data/BugBroker
10. 1" maxFileLength="32mb"/>
11.     </persistenceAdapter>
12.
13. </broker>
14. </beans>

```

那么经过一段时间后可能会报出如下错误:

```

ERROR [ActiveMQ Transport: tcp:///127.0.0.1:1843 -
RecoveryListenerAdapter.java:58 - RecoveryListenerAdapter] Message id
ID:versus-1837-1203915536609-0:2:1:1:419 could not be recovered from the
data store!

```


Apache 官方文档说, 此 bug 已经被修正, 预定在 5.1.0 版本上体现。

2. 5. 2 Broker clusters

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 

```

1. <broker brokerName="receiver" persistent="false" useJmx="false"
2. >
3.     <transportConnectors>
4.         <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 

```

1. <broker name="sender" persistent="false" useJmx="false">
2.   <transportConnectors>
3.     <transportConnector uri="tcp://localhost:0" discoveryUri="m
      ulticast://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。以下是关于 NetworkConnector 属性的一个例子:

Xml 代码 

```

1. <networkConnectors>
2.   <networkConnector uri="static://(tcp://localhost:61617)"
3.     name="bridge" dynamicOnly="false" conduitSubscriptions="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>

```

2. 5. 3 Master Slave

在一个网络内运行多个 brokers 或者 stand alone brokers 时存在一个问题，这就是消息在物理上只被一个 broker 持有，因此当某个 broker 失效，那么你能等待直到它重启后，这个 broker 上的消息才能够被继续发送（如果没有设置持久化，那么在这种情况下，消息将会丢失）。Master Slave 背后的想法是，消息被复制到 slave broker，因此即使 master broker 遇到了像硬件故障之类的错误，你也可以立即切换到 slave broker 而不丢失任何消息。

Master Slave 是目前 ActiveMQ 推荐的高可靠性和容错的解决方案。以下是几种不同的类型：

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

2. 5. 3. 1 Pure Master Slave


Pure Master Slave 的工作方式如下：

- 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；另外一种选择是 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:61616)?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" shutdownOnMasterFailure="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" deleteAllMessagesOnStartup="true" xmlns="http://activemq.org/config/1.0">
2.   ...
3.   <services>
4.     <masterConnector remoteURI="tcp://localhost:62001" username="user" password="password"/>
5.   </services>
6. </broker>
```

需要注意的是,笔者认为 ActiveMQ5.0 版本的 Pure Master Slave 仍然不够稳定。

2. 5. 3. 2 Shared File System Master Slave

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。以下是 ActiveMQ 配置的一个例子:

Xml 代码 

```
1. <broker useJmx="false" xmlns="http://activemq.org/config/1.0">
2.   <persistenceAdapter>
3.     <journalJDBC dataDirectory="/sharedFileSystem/broker"/>
4.   </persistenceAdapter>
5.   ...
6. </broker>
```

2. 5. 3. 3 JDBC Master Slave

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。以下是 ActiveMQ 配置的一个例子:

Xml 代码

```
1. <beans>
2.   <broker xmlns="http://activemq.org/config/1.0" brokerName="JdbcMasterBroker">
3.     ...
4.     <persistenceAdapter>
5.       <jdbcPersistenceAdapter dataSource="#mysql-ds"/>
6.     </persistenceAdapter>
7.
8.   </broker>
9.
10.  <bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
11.    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
12.    <property name="url" value="jdbc:mysql://localhost:3306/test?relaxAutoCommit=true"/>
13.    <property name="username" value="username"/>
14.    <property name="password" value="password"/>
15.    <property name="poolPreparedStatements" value="true"/>
16.  </bean>
17.</beans>
```

需要注意的是，如果你使用 MySQL 数据库，需要首先执行以下三条语句：
(Apache 官方文档说，此 bug 已经被修正，预定在 5.1.0 版本上体现)

Sql 代码

```
1. ALTER TABLE activemq_acks ENGINE = InnoDB;
2. ALTER TABLE activemq_lock ENGINE = InnoDB;
3. ALTER TABLE activemq_msgs ENGINE = InnoDB;
```

2. 6 Features


ActiveMQ 包含了很多功能强大的特性，下面简要介绍其中的几个。

2. 6. 1 Exclusive Consumer

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");
```

2. 6. 2 Message Groups


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 0。

从 4.1 版本开始，ActiveMQ 支持一个布尔字段 JMSXGroupFirstForConsumer。当某个 message group 的第一个消息被发送到 consumer 的时候，这个字段被设置。如果客户使用 failover transport 连接到 broker。在由于网络问题等造成客户重新连接到 broker 的时候，相同 message group 的消息可能会被分发到不同与之前的 consumer，因此 JMSXGroupFirstForConsumer 字段也会被重新设置。

以下是使用 message groups 的例子：


Java 代码 

```
1. Message message = session.createTextMessage("<foo>hey</foo>");
2. message.setStringProperty("JMSXGroupID", "IBM_NASDAQ_20/4/05");
```

3. ...
4. `producer.send(message);`

2. 6. 3 JMS Selectors

JMS Selectors 用于在订阅中，基于消息属性对进行消息的过滤。JMS Selectors 由 SQL92 语法定义。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"JMGroupID = '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。

2. 6. 4 Pending Message Limit Strategy

首先简要介绍一下 prefetch 机制。ActiveMQ 通过 prefetch 机制来提高性能，这意味这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, 可以为不同的 topic namespace 配置不同的策略。目前有以下两种:

- ConstantPendingMessageLimitStrategy。这个策略使用常量限制。
例如: `<constantPendingMessageLimitStrategy limit="50"/>`
- PrefetchRatePendingMessageLimitStrategy。这个策略使用 prefetch size 的倍数限制。
例如: `<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 sl
    ow consumers -->
12.        <pendingMessageLimitStrategy>
13.            <constantPendingMessageLimitStrategy limit="10"/
    >
14.        </pendingMessageLimitStrategy>
15.    </policyEntry>
16. </policyEntries>
17. </policyMap>
18. </destinationPolicy>
19. ...
20. </broker>

```


2. 6. 5 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 


```
1. Queue queue = new ActiveMQQueue("F00. A, F00. B, F00. 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 

```
1. Queue queue = new ActiveMQQueue("F00. A, topic://NOTIFY. F00. A");
```

以下是 ActiveMQ 配置文件进行配置的一个例子:

Xml 代码 

```

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>
12.</destinationInterceptors>

```

可以在转发前，先通过 JMS Selector 判断一个消息是否需要转发，例如：

Xml 代码 

```

1. <destinationInterceptors>
2.     <virtualDestinationInterceptor>
3.         <virtualDestinations>
4.             <compositeQueue name="MY.QUEUE">
5.                 <forwardTo>
6.                     <filteredDestination selector="odd = 'yes'" queue="FOO" />
7.                     <filteredDestination selector="i = 5" topic="BAR"/>
8.                 </forwardTo>
9.             </compositeQueue>
10.        </virtualDestinations>
11.    </virtualDestinationInterceptor>
12.</destinationInterceptors>

```

2. 6. 6 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.
7.   <destinationInterceptors>
8.     <mirroredQueue copyMessage = "true" prefix="Mirror.Topic"
9.   />
10.  </destinationInterceptors>
11. ...
12. </broker>

```

假如某个 producer 向名为 Foo.Bar 的 queue 中发送消息，那么你可以通过订阅名为 Mirror.Topic.Foo.Bar 的 topic 来获得发送到 Foo.Bar 中的所有消息。

2. 6. 7 Wildcards

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

2. 6. 8 Async Sends

ActiveMQ 支持以同步 (sync) 方式或者异步 (async) 方式向 broker 发送消息。使用何种方式对 send 方法的延迟有巨大的影响。对于生产者来说, 既然延迟是决定吞吐量的重要因素, 那么使用异步发送方式会极大地提高系统的性能。

ActiveMQ 缺省使用异步传输方式。但是按照 JMS 规范, 当在事务外发送持久化消息的时候, ActiveMQ 会强制使用同步发送方式。在这种情况下, 每一次发送都是同步的, 而且阻塞到收到 broker 的应答。这个应答保证了 broker 已经成功地将消息持久化, 而且不会丢失。但是这样作也严重地影响了性能。

如果你的系统可以容忍少量的消息丢失, 那么可以在事务外发送持久消息的时候, 选择使用异步方式。以下是几种不同的配置方式:

Java 代码 

1. `cf = new ActiveMQConnectionFactory("tcp://localhost:61616?jms.useAsyncSend=true");`
2. `((ActiveMQConnectionFactory)connectionFactory).setUseAsyncSend(true);`
3. `((ActiveMQConnection)connection).setUseAsyncSend(true);`

2. 6. 9 Dispatch Policies

2. 6. 9. 1 Round Robin Dispatch Policy

在 2. 6. 4 小节介绍过 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="F00.">`
5. `<dispatchPolicy>`
6. `<roundRobinDispatchPolicy />`
7. `</dispatchPolicy>`
8. `</policyEntry>`
9. `</policyEntries>`

10. `</policyMap>`
11. `</destinationPolicy>`

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

2. 6. 10 Message Cursors

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:

Xml 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>

2. 6. 11 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.o
ptimizeAcknowledge=false");
2. ((ActiveMQConnectionFactory)connectionFactory).setOptimizeAckno
wledge(false);
3. ((ActiveMQConnection)connection).setOptimizeAcknowledge(false);


2. 6. 12 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="F00.>" producerFlowControl="false">
5. <dispatchPolicy>
6. <strictOrderDispatchPolicy/>
7. </dispatchPolicy>

```
8.         </policyEntry>
9.     </policyEntries>
10. </policyMap>
11.</destinationPolicy>
```

2. 6. 13 Message Transformation

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 JMSEException;
3.     Message consumerTransform(Session session, MessageConsumer
        consumer, Message message) throws JMSEException;
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 


```
1. public class SimpleMessage implements Serializable {
2.     //
```

```

3.     private static final long serialVersionUID = 22510418418719
       75105L;
4.
5.     //
6.     private String id;
7.     private String text;
8.
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 MessageTransforme
   r() {
2.     public Message consumerTransform(Session session, MessageConsum
       er 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. }
9.
10. public Message producerTransform(Session session, MessageProducer consumer, Message message) throws JMSException {
11.     return null;
12. }
13. });
```