# activeMQ

## 一 ActiveMQ简介

### 1.1 什么是ActiveMQ

ActiveMQ是Apache推出的，一款开源的，完全支持JMS1.1和J2EE 1.4规范的JMS Provider实现的消息中间件（Message Oriented Middleware，MOM），实际上为什么把MQ叫做消息中间件。它最初的来源当然是由于系统A与系统B之间有消息的传递。这个时候我们把系统A与系统B之间消息传递的过程打断。A与B通过MQ来间接通信的过程。所以这个时候的MQ就叫做消息中间件。

### 1.2 ActiveMQ的作用

最主要的功能就是：实现JMS Provider，用来帮助实现高可用、高性能、可伸缩、 易用和安全的企业级面向消息服务的系统。

### 1.3 ActiveMQ特点

完全支持JMS1.1和J2EE 1.4规范（持久化，XA消息，事务)

支持多种传送协议：in-VM,TCP,SSL,NIO,UDP,JGroups,JXTA

可插拔的体系结构，可以灵活定制，如：消息存储方式、安全管理等

很容易和Application Server集成使用

多种语言和协议编写客户端。语言: Java,C,C++,C#,Ruby,Perl,Python,PHP

从设计上保证了高性能的集群，客户端-服务器，点对点

可以很容易的和Spring结合使用

支持通过JDBC和journal提供高速的消息持久化

支持与Axis的整合。

### 1.5 ActiveMQ的主要功能

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。

### 1.6 ActiveMQ的主要特点

1：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。

2：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。

### 1.7 ActiveMQ的主要应用场景

在多个系统间进行整合和通讯的时候，通常会要求：

1：可靠传输，数据不能丢失，有的时候，也会要求不能重复传输；

2：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

### 1.8 比较知名的消息中间件

IBM MQSeries

BEA WebLogicJMS Server

Oracle AQ

Tibco

SwiftMQ

AcitveMQ：是免费的java实现的消息中间件

## 二 ActiveMQ安装与基本使用

注意：安装gcc,jdk等

### 2.1 安装解压

ActiveMQ服务器端 1：从http://activemq.apache.org/download.html下载最新的ActiveMQ



### 2.2 启动运行

1：普通启动：到ActiveMQ/bin下面，./activemq start

2：启动并指定日志文件 ./activemq start > /tmp/activemqlog

### 2.3 启动检查

ActiveMQ默认采用61616端口提供JMS服务，使用8161端口提供管理控制台服 务，执行以下命令以便检验是否已经成功启动ActiveMQ服务：

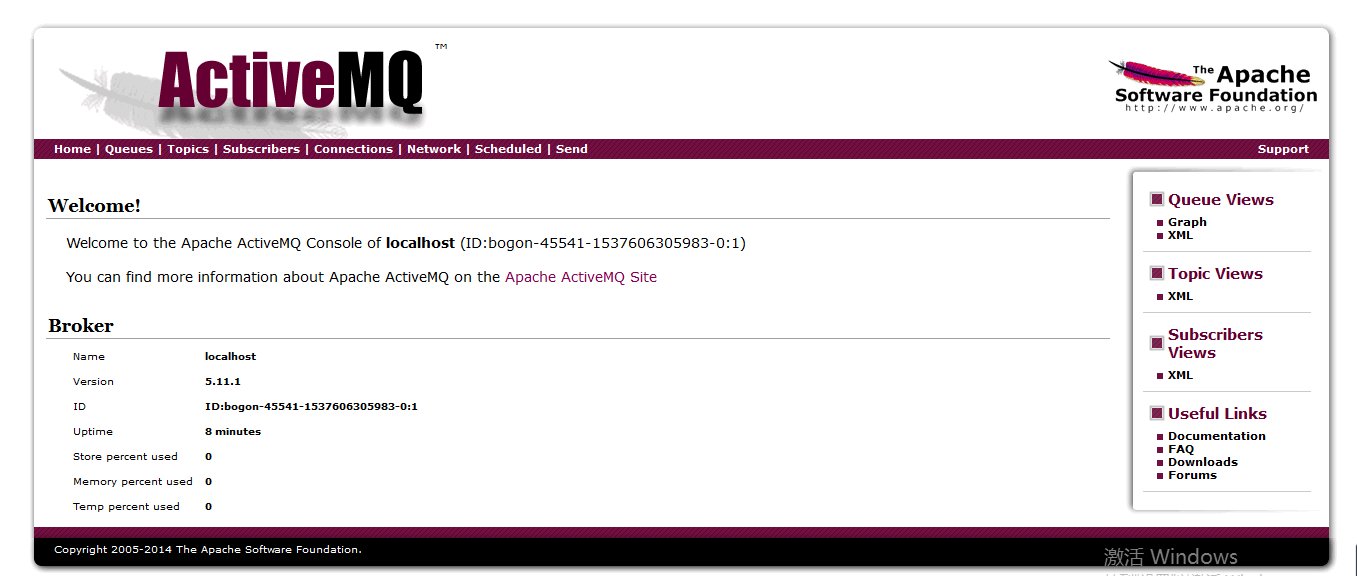
1：比如查看61616端口是否打开： netstat -an | grep 61616

2：也可以直接查看控制台输出或者日志文件

3：还可以直接访问ActiveMQ的管理页面：

<http://192.168.232.128:8161/>

默认的用户名和密码是admin/admin



### 2.4 停止ActiveMQ

可以用./activemq stop

暴力点的可以用ps aux| grep activemq 来得到进程号，然后kill掉

### 2.5 生产者

**public** **class** MsgSendder {

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

ActiveMQConnectionFactory ConnectionFactoryconnectionFactory = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection connection = ConnectionFactoryconnectionFactory.createConnection();

connection.start();

Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageProducer producer = session.createProducer(destination);

**for** (**int** i = 0; i < 3; i++) {

TextMessage message = session.createTextMessage("message--" + i);

Thread.*sleep*(1000);

producer.send(message);

}

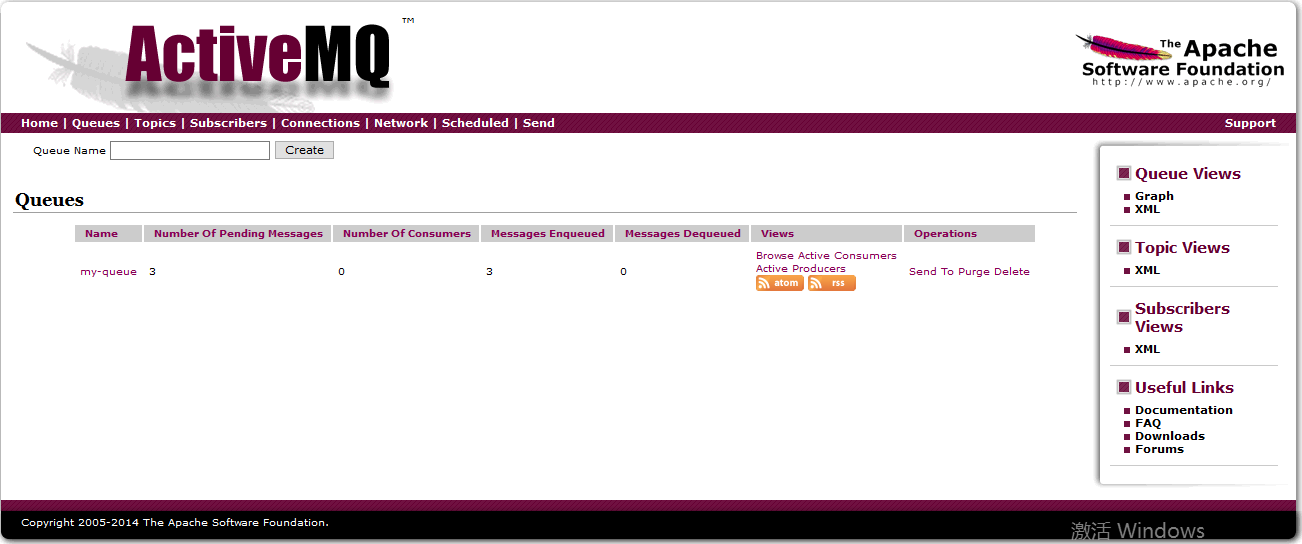
session.commit();

session.close();

connection.close();

}

}



### 2.6 消费者

**public** **class** MsgReceiver {

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

ConnectionFactory cf = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection connection = cf.createConnection();

connection.start();

**final** Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageConsumer consumer = session.createConsumer(destination);

**int** i = 0;

**while** (i < 3) {

i++;

TextMessage message = (TextMessage) consumer.receive();

session.commit();

System.***out***.println("收到消息：" + message.getText());

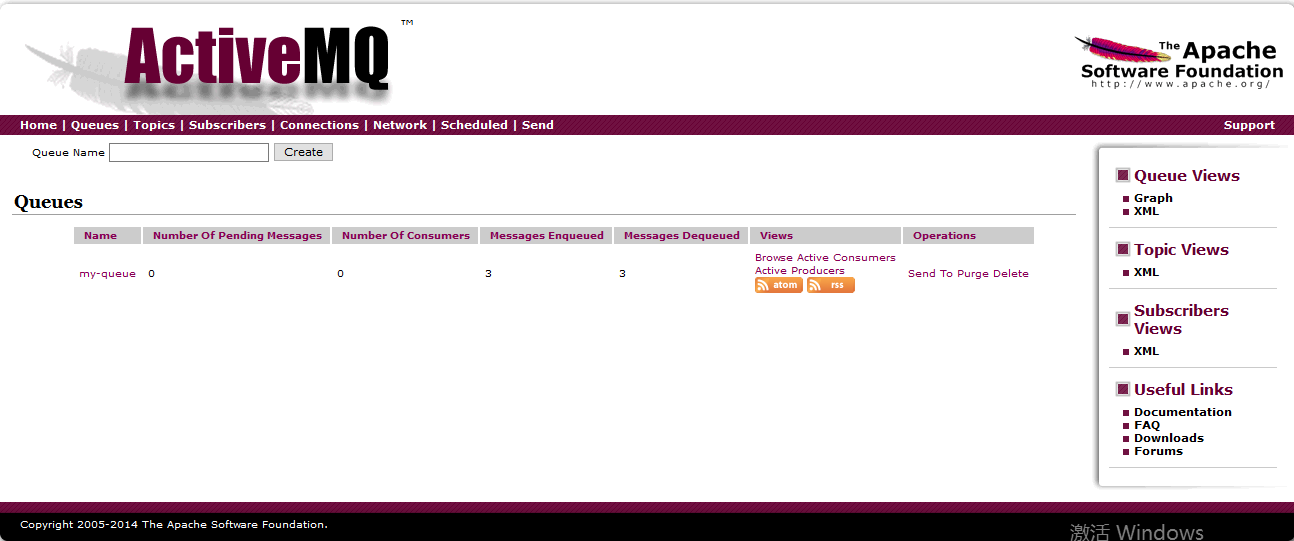
}

session.close();

connection.close();

}

}



## 三 JMS的基本概念

### 3.1 什么是jms

JMS Java Message Service，Java消息服务，是Java EE中的一个技术。

### 3.2 jms构成

JMS定义了Java 中访问消息中间件的接口，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

#### 3.2.1 JMS provider

实现JMS接口和规范的消息中间件

#### 3.2.2 JMS message

JMS的消息，JMS消息由以下三部分组成：

1：消息头：每个消息头字段都有相应的getter和setter方法

2：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

3：消息体：封装具体的消息数据

#### 3.2.3 JMS producer

消息生产者，创建和发送JMS消息的客户端应用

#### 3.2.4 JMS consumer

消息消费者，接收和处理JMS消息的客户端应用 消息的消费可以采用以下两种方法之一：

#### 3.2.4.1 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。

#### 3.2.4.2 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

### 3.3 jms消息传递模式

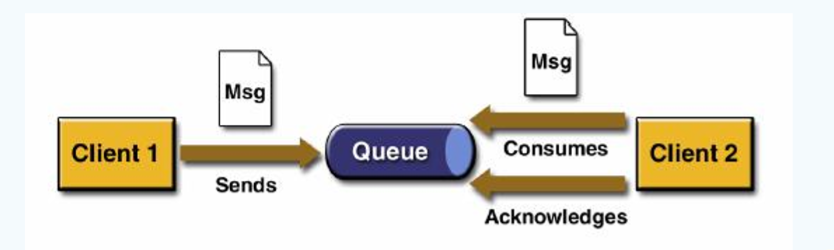
#### 3.3.1 点对点模式

JMS domains：消息传递域，JMS规范中定义了两种消息传递域：点对点 （point-topoint，简写成PTP）消息传递域和发布/订阅消息传递域（publish/subscribe，简写 成pub/sub）

(1) 每个消息只能有一个消费者

(2) 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。

(3) 目的地topic



##### PTP的一些特点

1：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

2：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

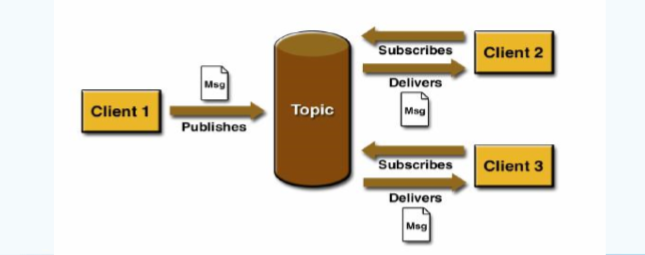
3：队列可以长久地保存消息直到消费者收到消息。消费者不需要因为担心消息会丢 失而时刻和队列保持激活的连接状态，充分体现了异步传输模式的优势

#### 3.3.2 发布订阅模式

(1)每个消息可以有多个消费者

(2)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

(3)目的地topic



##### Pub/Sub的一些特点

1：消息订阅分为非持久订阅和持久订阅 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。

2：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不会被接收

3：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。

4：当所有的消息必须被接收，则用持久订阅。当丢失消息能够被容忍，则用非持久订阅

### 3.4 jms的api类

(1)Connection factory：连接工厂，用来创建连接对象，以连接到JMS的provider

(2)JMS Connection：封装了客户与JMS 提供者之间的一个虚拟的连接

(3)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。

(4)Destination：消息发送到的目的地

(5)Acknowledge：签收

(6)Transaction：事务

(7)JMS client：用来收发消息的Java应用

(8)Non-JMS client：使用JMS provider本地API写的应用，用来替换JMS API实现收 发消息的功能，通常会提供其他的一些特性，比如：CORBA、RMI等。

(9)Administered objects：预定义的JMS对象，通常在provider规范中有定义，提 供给JMS客户端来访问，比如： ConnectionFactory和Destination

### 3.5 消息头结构

JMS 消息由以下几部分组成：消息头，属性和消息体

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下：

1：JMSDestination：由send方法设置

2：JMSDeliveryMode：由send方法设置

3：JMSExpiration：由send方法设置

4：JMSPriority：由send方法设置

5：JMSMessageID：由send方法设置

6：JMSTimestamp：由客户端设置

7：JMSCorrelationID ：由客户端设置

8：JMSReplyTo ：由客户端设置

9：JMSType ：由客户端设置

10：JMSRedelivered：由JMS Provider设置

标准的 JMS 消息头包含以下属性：

1：JMSDestination：消息发送的目的地：主要是指Queue和Topic，自动分配

2：JMSDeliveryMode：传送模式。有两种 ：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。自动分配

3：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。自动分配

4：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。自动分配

5：JMSMessageID：唯一识别每个消息的标识，由JMS Provider 产生。自动分配

6：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。自动分配

7：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标示的上一条消息的应答，不过，JMSCorrelationID可以是任何 值，不仅仅是JMSMessageID。由开发者设置

8：JMSReplyTo：提供本消息回复消息的目的地址。由开发者设置

9：JMSType：消息类型的识别符。由开发者设置

10：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。自动设置

### 3.6 消息体结构

消息体，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。包括：TextMessage、MapMessage、 BytesMessage、StreamMessage和ObjectMessage

消息属性，包含以下三种类型的属性：

1：应用程序设置和添加的属性，比如： Message.setStringProperty(“username”,username); 2：JMS定义的属性 使用“JMSX”作为属性名的前缀， connection.getMetaData().getJMSXPropertyNames()， 方法返回所有连接支持 的JMSX 属性的名字。

3：JMS供应商特定的属性

JMS定义的属性如下：

1：JMSXUserID：发送消息的用户标识，发送时提供商设置

2：JMSXAppID：发送消息的应用标识，发送时提供商设置

3：JMSXDeliveryCount：转发消息重试次数,第一次是1，第二次是2，… ，发送时 提供商设置

4：JMSXGroupID：消息所在消息组的标识，由客户端设置

5：JMSXGroupSeq：组内消息的序号第一个消息是1，第二个是2，…，由客户端设置

6：JMSXProducerTXID ：产生消息的事务的事务标识，发送时提供商设置

7：JMSXConsumerTXID ：消费消息的事务的事务标识，接收时提供商设置

8：JMSXRcvTimestamp ：JMS 转发消息到消费者的时间，接收时提供商设置

9：JMSXState：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。由提供商设置

### 3.7 可靠机制

#### 3.7.1 消息确认机制

消息的成功消费三个阶段：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：客户通过调用消息的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的确认消息的提交。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

#### 3.7.2 消息持久性

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

#### 3.7.3 消息优先级

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 n 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 n 消息的临时目的地 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

#### 3.7.2 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。

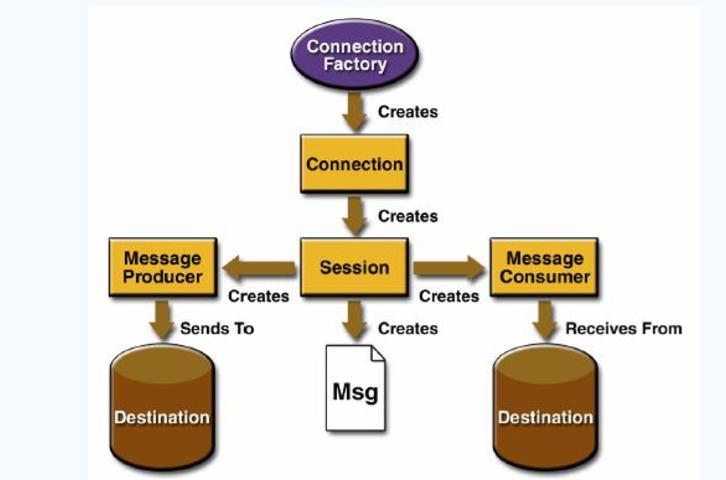
如果最初创建 持久订阅的客户或者任何其它客户，使用相同的连接工厂和连接的客户ID，相同 的主题和相同的订阅名，再次调用会话上的createDurableSubscriber方法，那 么该持久订阅就会被激活。JMS provider会向客户发送客户处于非激活状态时所 发布的消息。 持久订阅在某个时刻只能有一个激活的订阅者。持久订阅在创建之后会一 直保留，直到应用程序调用会话上的unsubscribe方法。也就是说，当订阅者不在线的时候，还能否在上线以后接口到下线时候没接收到的消息

#### 3.7.3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

### 3.8 jms是api结构

#### 3.8.1 api结构图



#### 3.8.2 api基本步骤

1：创建一个JMS connection factory

2：通过connection factory来创建JMS connection

3：启动JMS connection

4：通过connection创建JMS session

5：创建JMS destination

6：创建JMS producer,或者创建JMS message，并设置destination

7：创建JMS consumer，或者是注册一个JMS message listener

8：发送或者接受JMS message(s)

9：关闭所有的JMS资源(connection, session, producer, consumer等)

## 四 activeMQ消息通信

### 4.1 p2p的消息通信

#### 4.1.1 producer

**public** **class** MsgSendder {

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

ActiveMQConnectionFactory ConnectionFactoryconnectionFactory = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection connection = ConnectionFactoryconnectionFactory.createConnection();

connection.start();

Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageProducer producer = session.createProducer(destination);

**for** (**int** i = 0; i < 3; i++) {

TextMessage message = session.createTextMessage("message--" + i);

Thread.*sleep*(1000);

producer.send(message);

}

session.commit();

session.close();

connection.close();

}

}

#### 4.1.2 consumer

**public** **class** MsgReceiver {

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

ConnectionFactory cf = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection connection = cf.createConnection();

connection.start();

**final** Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageConsumer consumer = session.createConsumer(destination);

**int** i = 0;

**while** (i < 3) {

i++;

TextMessage message = (TextMessage) consumer.receive();

session.commit();

System.***out***.println("收到消息：" + message.getText());

}

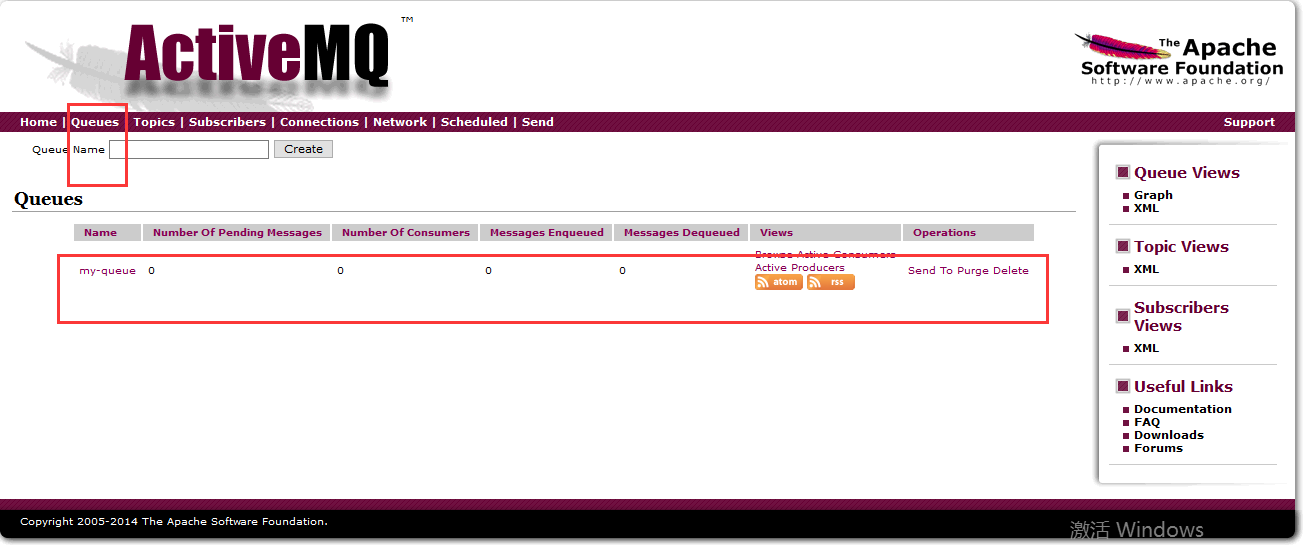
session.close();

connection.close();

}

}

#### 4.1.3 管理平台



### 4.2 非持久化消息

#### 4.2.1 producer

**public** **class** NonPersisiTopicSender {

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

ActiveMQConnectionFactory conFactory = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection createConnection = conFactory.createConnection();

createConnection.start();

Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Topic createTopic = createSession.createTopic("蜘蛛侠");

MessageProducer createProducer = createSession.createProducer(createTopic);

**for**(**int** i=0;i<3;i++){

TextMessage createTextMessage = createSession.createTextMessage("message"+i);

createProducer.send(createTextMessage);

}

createSession.commit();

createSession.close();

createConnection.close();

}

}

#### 4.2.2 consumer

**public** **class** NonPersisiTopicReceiver {

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

ActiveMQConnectionFactory activeMQConnectionFactory=**new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection createConnection = activeMQConnectionFactory.createConnection();

createConnection.start();

Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Topic createTopic = createSession.createTopic("蜘蛛侠");

MessageConsumer createConsumer = createSession.createConsumer(createTopic);

TextMessage message = (TextMessage)createConsumer.receive();

**while**(message!=**null**){

System.***out***.println(message.getText());

message = (TextMessage)createConsumer.receive();

}

createSession.commit();

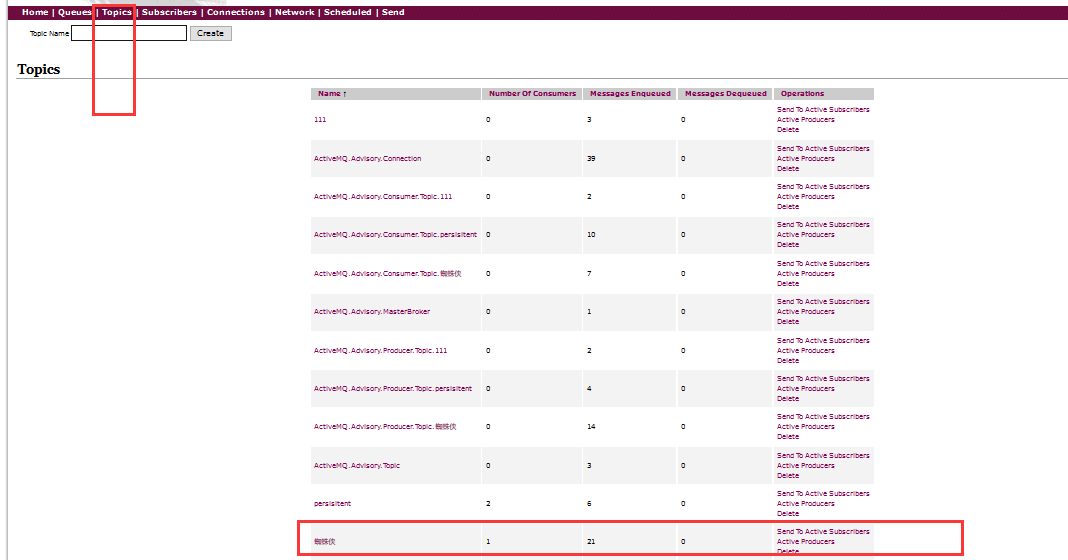
createSession.close();

createConnection.close();

}

}

#### 4.2.3 管理平台



### 4.3 持久化消息

#### 4.3.1 producer

**public** **class** PersisiTopicSender {

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

ActiveMQConnectionFactory conFactory = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection createConnection = conFactory.createConnection();

Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Topic createTopic = createSession.createTopic("persisitent");

MessageProducer createProducer = createSession.createProducer(createTopic);

createProducer.setDeliveryMode(DeliveryMode.***PERSISTENT***);

createConnection.start();

**for**(**int** i=0;i<3;i++){

TextMessage createTextMessage = createSession.createTextMessage("message"+i);

createProducer.send(createTextMessage);

}

createSession.commit();

createSession.close();

createConnection.close();

}

}

#### 4.3.2 consumer

**public** **class** PersisiTopicReceiver {

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

ActiveMQConnectionFactory activeMQConnectionFactory=**new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection createConnection = activeMQConnectionFactory.createConnection();

createConnection.setClientID("订阅者B\_ID");

createConnection.start();

Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Topic createTopic = createSession.createTopic("persisitent");

TopicSubscriber createDurableSubscriber = createSession.createDurableSubscriber(createTopic, "T1");

TextMessage message = (TextMessage)createDurableSubscriber.receive();

**while**(message!=**null**){

System.***out***.println(message.getText());

message = (TextMessage)createDurableSubscriber.receive();

}

createSession.commit();

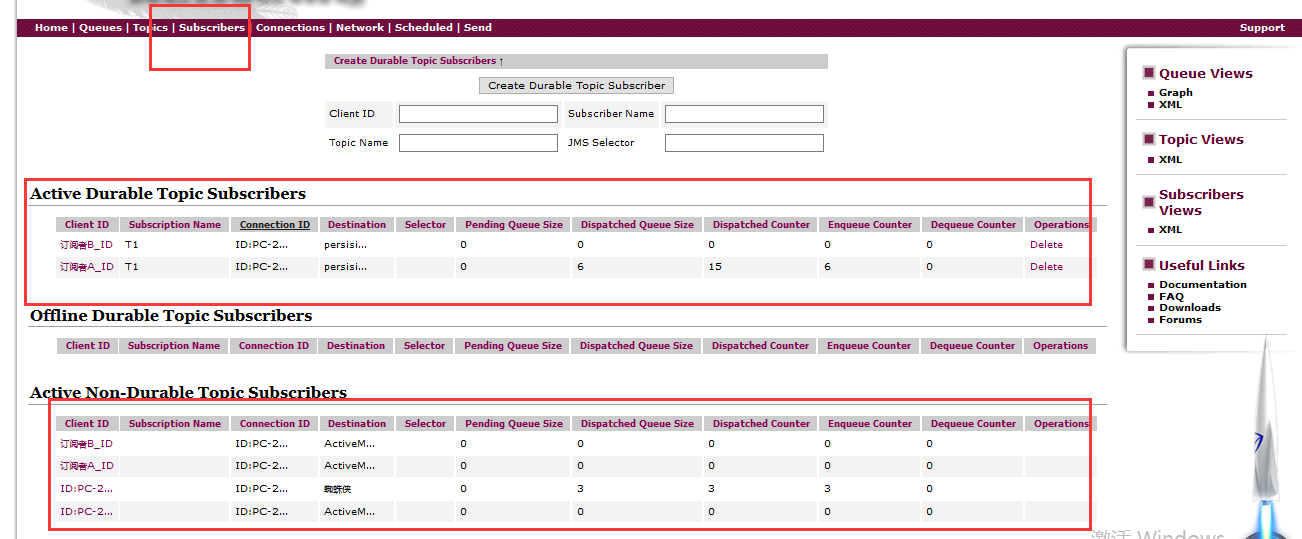
createSession.close();

createConnection.close();

}

}

#### 4.3.3 管理平台



### 4.4 总结

#### 4.4.1持久化消息

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。

#### 4.4.2非持久化消息

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； 如： producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);

2．使用send 方法为每一条消息设置传送模式

## 五 嵌入式的MQ实例

### 5.1 broker启动

一个broker相当于一个activeMQ实例。

1：activemqstart ：使用默认的activemq.xml来启动

2：activemqstart xbean:file:../conf/activemq-2.xml ：使用指定的配置文件 来启动

3：如果不指定file，也就是xbean:activemq-2.xml，那么xml必须在classpath下面

### 5.2 代码嵌入broker

#### 5.2.1 启动方式说明

1：Broker Service启动broker ，示例如下：

BrokerServicebroker = new BrokerService();

broker.setUseJmx(true);

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

broker.start();

2：BrokerFactory启动broker ，示例如下：

String Uri = "properties:broker.properties";

BrokerServicebroker1 = BrokerFactory.createBroker(newURI(Uri));

broker1.addConnector("tcp://localhost:61616");

broker1.start();

3：broker.properties的内容示例如下：

useJmx=true

persistent=false

brokerName=Cheese

#### 5.2.2 启动实例

##### (1)Broker服务

**public** **class** Snippet {

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

BrokerService broker = **new** BrokerService();

broker.setUseJmx(**true**);

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

broker.start();

}

}

##### (2)Sender

**public** **class** MsgSendder {

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

ActiveMQConnectionFactory ConnectionFactoryconnectionFactory = **new** ActiveMQConnectionFactory("tcp://localhost:61616");

Connection connection = ConnectionFactoryconnectionFactory.createConnection();

connection.start();

Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageProducer producer = session.createProducer(destination);

**for** (**int** i = 0; i < 3; i++) {

TextMessage message = session.createTextMessage("message--" + i);

Thread.*sleep*(1000);

producer.send(message);

}

session.commit();

session.close();

connection.close();

}

}

##### (3)Receiver

**public** **class** MsgReceiver {

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

ConnectionFactory cf = **new** ActiveMQConnectionFactory("tcp://localhost:61616");

Connection connection = cf.createConnection();

connection.start();

**final** Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageConsumer consumer = session.createConsumer(destination);

**int** i = 0;

**while** (i < 3) {

i++;

TextMessage message = (TextMessage) consumer.receive();

session.commit();

System.***out***.println("收到消息：" + message.getText());

}

session.close();

connection.close();

}

}

### 5.3 spring嵌入broker

#### 5.3.1 spring配置

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.2.xsd"*>

<bean id=*"broker"* class=*"org.apache.activemq.broker.BrokerService"*

init-method=*"start"* destroy-method=*"stop"*>

<property name=*"brokerName"* value=*"myBroker"* />

<property name=*"persistent"* value=*"false"* />

<property name=*"transportConnectorURIs"*>

<list>

<value>tcp://localhost:61616</value>

</list>

</property>

</bean>

</beans>

#### 5.3.2 sender

**public** **class** MsgSendder {

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

ActiveMQConnectionFactory ConnectionFactoryconnectionFactory = **new** ActiveMQConnectionFactory("tcp://localhost:61616");

Connection connection = ConnectionFactoryconnectionFactory.createConnection();

connection.start();

Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageProducer producer = session.createProducer(destination);

**for** (**int** i = 0; i < 3; i++) {

TextMessage message = session.createTextMessage("message--" + i);

Thread.*sleep*(1000);

producer.send(message);

}

session.commit();

session.close();

connection.close();

}

}

#### 5.3.3 receiver

**public** **class** MsgReceiver {

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

ConnectionFactory cf = **new** ActiveMQConnectionFactory("tcp://localhost:61616");

Connection connection = cf.createConnection();

connection.start();

**final** Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("my-queue");

MessageConsumer consumer = session.createConsumer(destination);

**int** i = 0;

**while** (i < 3) {

i++;

TextMessage message = (TextMessage) consumer.receive();

session.commit();

System.***out***.println("收到消息：" + message.getText());

}

session.close();

connection.close();

}

}

## 六 spring集成MQ

### 6.1 spring继承MQ(p2p)

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.2.xsd"*>

<context:property-placeholder location=*"classpath:resource/\*.properties"* />

<context:component-scan base-package=*"xiao.it.\*"*/>

<bean id=*"jmsFactory"* class=*"org.apache.activemq.pool.PooledConnectionFactory"*

destroy-method=*"stop"*>

<property name=*"connectionFactory"*>

<bean class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"*>

<value>tcp://192.168.232.128:61616</value>

</property>

</bean>

</property>

<property name=*"maxConnections"* value=*"100"*></property>

</bean>

<bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>

<property name=*"connectionFactory"* ref=*"jmsFactory"* />

<property name=*"defaultDestination"* ref=*"destination"* />

<property name=*"messageConverter"*>

<bean

class=*"org.springframework.jms.support.converter.SimpleMessageConverter"* />

</property>

</bean>

<bean id=*"destination"* class=*"org.apache.activemq.command.ActiveMQQueue"*>

<constructor-arg index=*"0"* value=*"spring-queue"* />

</bean>

<bean id=*"springSender"* class=*"xiao.it.jms.SpringSender"*></bean>

<bean id=*"springReceiver"* class=*"xiao.it.jms.SpringReceiver"*></bean>

</beans>

### 6.2 producer

**public** **class** SpringSender {

@Autowired

**private** JmsTemplate jmsTemplate= **null**;

**public** **static** **void** main(String[] args)

{

@SuppressWarnings("resource")

ApplicationContext ctx= **new** ClassPathXmlApplicationContext("spring-service.xml");

SpringSender ct = (SpringSender)ctx.getBean("springSender");

ct.jmsTemplate.send(**new** MessageCreator() {

**public** Message createMessage(Session s) **throws** JMSException {

TextMessage msg= s.createTextMessage("Springmsg===");

**return** msg;

}

});

}

}

### 6.3 consumer

@Service

**public** **class** SpringReceiver {

@Autowired

**private** JmsTemplate jmsTemplate= **null**;

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

@SuppressWarnings("resource")

ApplicationContext ctx= **new** ClassPathXmlApplicationContext("spring-service.xml");

SpringReceiver ct = (SpringReceiver)ctx.getBean("springReceiver");

String msg= (String)ct.jmsTemplate.receiveAndConvert();

System.***out***.println("msg==="+msg);

}

}

### 6.4 spring继承MQ(pub/sub)

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.2.xsd"*>

<context:property-placeholder location=*"classpath:resource/\*.properties"* />

<context:component-scan base-package=*"xiao.it.\*"*/>

<bean id=*"jmsFactory"* class=*"org.apache.activemq.pool.PooledConnectionFactory"*

destroy-method=*"stop"*>

<property name=*"connectionFactory"*>

<bean class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"*>

<value>tcp://192.168.232.128:61616</value>

</property>

</bean>

</property>

<property name=*"maxConnections"* value=*"100"*></property>

</bean>

<bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>

<property name=*"connectionFactory"* ref=*"jmsFactory"* />

<property name=*"defaultDestination"* ref=*"destinationTopic"* />

<property name=*"messageConverter"*>

<bean

class=*"org.springframework.jms.support.converter.SimpleMessageConverter"* />

</property>

</bean>

<bean id=*"destination"* class=*"org.apache.activemq.command.ActiveMQQueue"*>

<constructor-arg index=*"0"* value=*"spring-queue"* />

</bean>

<bean id=*"destinationTopic"* class=*"org.apache.activemq.command.ActiveMQTopic"*>

<constructor-arg index=*"0"* value=*"spring-topic"* />

</bean>

<bean id=*"springSender"* class=*"xiao.it.jms.SpringSender"*></bean>

<!-- <bean id="springReceiver" class="xiao.it.jms.SpringReceiver"></bean> -->

</beans>

### 6.5 producer

**public** **class** SpringSender {

@Autowired

**private** JmsTemplate jmsTemplate= **null**;

**public** **static** **void** main(String[] args)

{

@SuppressWarnings("resource")

ApplicationContext ctx= **new** ClassPathXmlApplicationContext("spring-service.xml");

SpringSender ct = (SpringSender)ctx.getBean("springSender");

ct.jmsTemplate.send(**new** MessageCreator() {

**public** Message createMessage(Session s) **throws** JMSException {

TextMessage msg= s.createTextMessage("Springmsg===");

**return** msg;

}

});

}

}

### 6.6 consumer

@Service

**public** **class** SpringReceiver {

@Autowired

**private** JmsTemplate jmsTemplate= **null**;

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

@SuppressWarnings("resource")

ApplicationContext ctx= **new** ClassPathXmlApplicationContext("spring-service.xml");

SpringReceiver ct = (SpringReceiver)ctx.getBean("springReceiver");

String msg= (String)ct.jmsTemplate.receiveAndConvert();

System.***out***.println("msg==="+msg);

}

}

### 6.7 spring继承MQ(消息监听器)

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.2.xsd*

*http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.2.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-3.2.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.2.xsd"*>

<context:property-placeholder location=*"classpath:resource/\*.properties"* />

<context:component-scan base-package=*"xiao.it.\*"* />

<bean id=*"jmsFactory"* class=*"org.apache.activemq.pool.PooledConnectionFactory"*

destroy-method=*"stop"*>

<property name=*"connectionFactory"*>

<bean class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"*>

<value>tcp://192.168.232.128:61616</value>

</property>

</bean>

</property>

<property name=*"maxConnections"* value=*"100"*></property>

</bean>

<bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>

<property name=*"connectionFactory"* ref=*"jmsFactory"* />

<property name=*"defaultDestination"* ref=*"destinationTopic"* />

<property name=*"messageConverter"*>

<bean

class=*"org.springframework.jms.support.converter.SimpleMessageConverter"* />

</property>

</bean>

<bean id=*"destinationTopic"* class=*"org.apache.activemq.command.ActiveMQTopic"*>

<constructor-arg index=*"0"* value=*"spring-topic"* />

</bean>

<!-- <bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">

<constructor-arg index="0" value="spring-queue" />

</bean>

<bean id="springReceiver" class="xiao.it.jms.SpringReceiver"></bean> -->

<bean id=*"jmsContainer"*

class=*"org.springframework.jms.listener.DefaultMessageListenerContainer"*>

<property name=*"connectionFactory"* ref=*"jmsFactory"* />

<property name=*"destination"* ref=*"destinationTopic"* />

<property name=*"messageListener"* ref=*"messageListener"* />

</bean>

<bean id=*"messageListener"* class=*"xiao.it.jms.MyMessageListener"*>

</bean>

</beans>

### 6.8 producer

**public** **class** SpringSender {

@Autowired

**private** JmsTemplate jmsTemplate= **null**;

**public** **static** **void** main(String[] args)

{

@SuppressWarnings("resource")

ApplicationContext ctx= **new** ClassPathXmlApplicationContext("spring-service.xml");

SpringSender ct = (SpringSender)ctx.getBean("springSender");

ct.jmsTemplate.send(**new** MessageCreator() {

**public** Message createMessage(Session s) **throws** JMSException {

TextMessage msg= s.createTextMessage("Springmsg===");

**return** msg;

}

});

}

}

### 6.9 consumer

前提是已经通过spring进行初始化，并且注入

**public** **class** MyMessageListener **implements** MessageListener

{

**public** **void** onMessage(Message message) {

TextMessage msg= (TextMessage)message;

**try** {

System.***out***.println("receivetxt msg==="+msg.getText());

} **catch** (JMSException e) {

e.printStackTrace();

}

}

}

### 6.10 最佳实践和建议

1：Spring消息发送的核心架构是JmsTemplate，隔离了像打开、关闭Session和 Producer的繁琐操作，因此应用开发人员仅仅需要关注实际的业务逻辑。但是 JmsTemplate损害了ActiveMQ的PooledConnectionFactory对session和消息 producer的缓存机制而带来的性能提升。

2：新的Spring里面，可以设置 org.springframework.jms.connection.CachingConnectionFactory的 sessionCacheSize ，或者干脆使用ActiveMQ的PooledConnectionFactory

3：不建议使用JmsTemplate的receive()调用，因为在JmsTemplate上的所有调用都 是同步的，这意味着调用线程需要被阻塞，直到方法返回，这对性能影响很大

4：请使用DefaultMessageListenerContainer，它允许异步接收消息并缓存session 和消息consumer，而且还可以根据消息数量动态的增加或缩减监听器的数量

## 七 ActiveMQ的通信协议

client-broker通讯协议如下：

1：TCP：这个也是缺省使用的协议 2：NIO 3：UDP 4：SSL 5：Http（s）

### 7.1 TCP协议Transmission Control Protocol (TCP)

1：这是默认的Broker配置，TCP的Client监听端口是61616。

2：在网络传输数据前，必须要序列化数据，消息是通过一个叫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。

3：TCP连接的URI形式：tcp://hostname:port?key=value&key=value，加粗部分是必须的

4：TCP传输的优点：

（1） TCP协议传输可靠性高，稳定性强

（2）高效性：字节流方式传递，效率很高

（3）有效性、可用性：应用广泛，支持任何平台

5：所有关于Transport协议的可配置参数，可以参见： <http://activemq.apache.org/configuring-version-5-transports.html>

### 7.2 New I/O API Protocol（NIO）

1：NIO协议和TCP协议类似，但NIO更侧重于底层的访问操作。它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

2：适合使用NIO协议的场景：

（1）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，所以建议使用NIO协议

（2）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

3：NIO连接的URI形式：nio://hostname:port?key=value

4：Transport Connector配置示例：

<transportConnectors>

<transportConnector name="tcp" uri="tcp://localhost:61616?trace=true" />

<transportConnectorname="nio"uri="nio://localhost:61618?trace=true"/> </transportConnectors>

上面的配置，示范了一个TCP协议监听61616端口，一个NIO协议监听61618端口

### 7.3 User Datagram Protocol（UDP)

1：UDP和TCP的区别

（1）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

（2）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

2：从上面可以得出：TCP是被用在稳定可靠的场景中使用的；UDP通常用在快速数据传递和不 怕数据丢失的场景中，还有ActiveMQ通过防火墙时，只能用UDP

3：UDP连接的URI形式：udp://hostname:port?key=value

4：Transport Connector配置示例：

<transportConnectors>

<transportConnector name="udp" uri="udp://localhost:61618?trace=true" />

</transportConnectors>

### 7.4 Secure Sockets Layer Protocol (SSL)

1：连接的URI形式：ssl://hostname:port?key=value

2：Transport Connector配置示例：

<transportConnectors>

<transportConnectorname="ssl"uri="ssl://localhost:61617?trace=true"/> </transportConnectors>

### 7.5 Hypertext Transfer Protocol (HTTP/HTTPS)

1：像web和email等服务需要通过防火墙来访问的，Http可以使用这种场合

2：连接的URI形式：http://hostname:port?key=value或者 https://hostname:port?key=value 3：Transport Connector配置示例：

<transportConnectors>

<transportConnectorname="http"uri="http://localhost:8080?trace=true"/> </transportConnectors>

### 7.6 VM Protocol（VM）

1：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，而是直接的方法调用。

2：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。

3：连接的URI形式：vm://brokerName?key=value

4：Java中嵌入的方式：

vm:broker:(tcp://localhost:6000)?brokerName=embeddedbroker&persistent=fal se ， 定义了一个嵌入的broker名称为embededbroker以及配置了一个 tcptransprotconnector在监听端口6000上

5：使用一个加载一个配置文件来启动broker

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

## 八 ActiveMQ数据持久化

### 8.1 ActiveMQ数据持久化方式

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： 1：AMQ消息存储-基于文件的存储方式，是以前的默认消息存储

2：KahaDB消息存储-提供了容量的提升和恢复能力，是现在的默认存储方式

3：JDBC消息存储-消息基于JDBC存储的

4：Memory 消息存储-基于内存的消息存储

### 8.2 AMQ持久化

AMQ Message Store概述 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，该结构是以流的形式来进行消息交互的。 这种方式中，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。

AMQ Message Store配置示例

<broker brokerName="broker" persistent="true" useShutdownHook="false">

<persistenceAdapter>

<amqPersistenceAdapterdirectory="${activemq.base}/data"maxFileLength="32mb"/> </persistenceAdapter>

</broker>

### 8.3 KahaDB 持久化

KahaDB Message Store概述 KahaDB是目前默认的存储方式，可用于任何场景，提高了性能和恢复能力。消息存储使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。 KahaDB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。

KahaDB基本配置例子

<persistenceAdapter>

<kahaDB directory="${activemq.data}/kahadb"/>

</persistenceAdapter>

可用的属性有：

1：director：KahaDB存放的路径，默认值activemq-data

2：indexWriteBatchSize： 批量写入磁盘的索引page数量，默认值1000

3：indexCacheSize：内存中缓存索引page的数量，默认值10000

4：enableIndexWriteAsync：是否异步写出索引，默认false

5：journalMaxFileLength：设置每个消息data log的大小，默认是32MB

6：enableJournalDiskSyncs：设置是否保证每个没有事务的内容，被同步写入磁盘，JMS持久化的时候需 要，默认为true

7：cleanupInterval：在检查到不再使用的消息后，在具体删除消息前的时间，默认30000

8：checkpointInterval：checkpoint的间隔时间，默认5000

9：ignoreMissingJournalfiles：是否忽略丢失的消息日志文件，默认false

10：checkForCorruptJournalFiles：在启动的时候，将会验证消息文件是否损坏，默认false

11：checksumJournalFiles：是否为每个消息日志文件提供checksum，默认false

12：archiveDataLogs：是否移动文件到特定的路径，而不是删除它们，默认false

13：directoryArchive：定义消息已经被消费过后，移动data log到的路径，默认null

14：databaseLockedWaitDelay：获得数据库锁的等待时间 (used by shared master/slave)，默认 10000

15：maxAsyncJobs：设置最大的可以存储的异步消息队列，默认值10000，可以和concurrent MessageProducers 设置成一样的值

16：concurrentStoreAndDispatchTransactions：是否分发消息到客户端，同时事务存储消息，默认 true

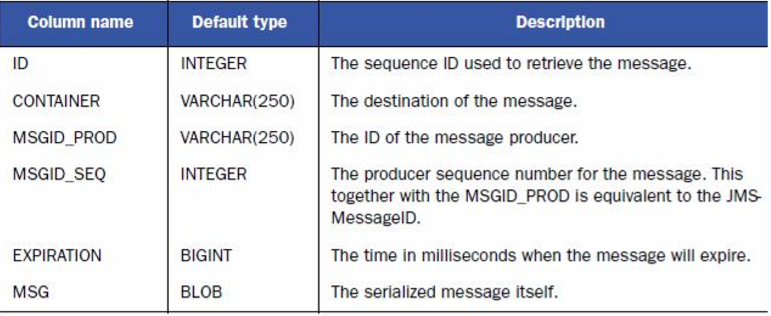
17：concurrentStoreAndDispatchTopics：是否分发Topic消息到客户端，同时进行存储，默认true

18：concurrentStoreAndDispatchQueues：是否分发queue消息到客户端，同时进行存储，默认true

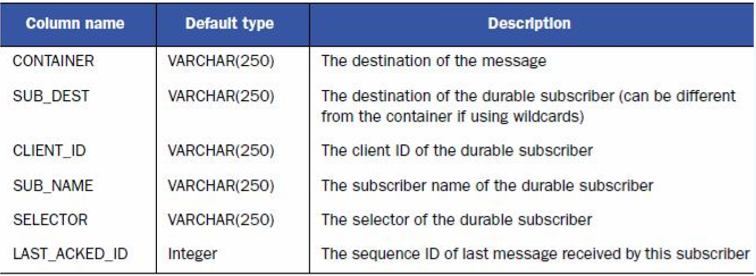
### 8.4 JDBC持久化

#### 8.4.1 建表

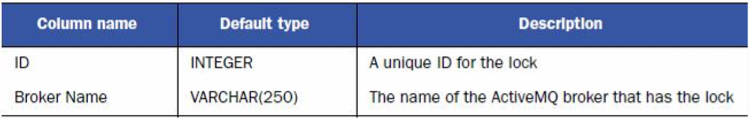
ACTIVEMQ\_MSGS 消息表



ACTIVEMQ\_ACKS 确认表



ACTIVEMQ\_LOCK 锁表



#### 8.4.2 配置

<beans>

<broker

brokerName="test-broker" persistent=true xmlns="http://activemq.apache.org/schema/core"> <persistenceAdapter>

<jdbcPersistenceAdapterdataSource=“#mysql-ds"/>

</persistenceAdapter>

</broker>

<bean name="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName">

<value>org.gjt.mm.mysql.Driver</value>

</property>

<property name="url">

<value>jdbc:mysql://192.168.1.100:3306/test?useUnicode=true&amp;characterEncodi ng=UTF-8</value>

</property>

<property name="username">

<value>root</value>

</property>

<property name="password" value="cc"/>

</bean>

### 8.5 JDBC Message Store with ActiveMQ Journal

这种方式克服了JDBC Store的不足，使用快速的缓存写入技术，大大提高了性能。 配置示例如下

#### 8.5.1 配置

<beans>

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

<persistenceFactory>

<journalPersistenceAdapterFactory journalLogFiles="4" journalLogFileSize="32768" useJournal="true" useQuickJournal="true" dataSource="#derby-ds" dataDirectory="activemq-data" />

</persistenceFactory>

</broker>

</beans>

#### 8.5.2 JDBC Store和JDBCMessage Store with ActiveMQJournal的区别

1：Jdbcwith journal的性能优于jdbc

2：Jdbc用于master/slave模式的数据库分享

3：Jdbcwith journal不能用于master/slave模式

4：一般情况下，推荐使用jdbcwith journal

### 8.6 Memory持久化

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所在的JVM和内存限制

#### 8.5.1 配置

<beans>

<broker brokerName="test-broker" persistent="false" xmlns="http://activemq.apache.org/schema/core">

<transportConnectors>

<transportConnectoruri="tcp://localhost:61635"/>

</transportConnectors>

</broker>

</beans>

#### 8.5.2 内嵌使用Broker

public void createEmbeddedBroker() throws Exception {

BrokerServicebroker = new BrokerService();

broker.setPersistent(false);

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

broker.start();

}

## 九 ActiveMQ伪集群

### 9.1创建ACtiveMQ实例

步骤如下：

1:把整个conf文件夹复制一份，比如叫做conf2

2:修改里面的activemq.xml文件

(1)里面的brokerName 不能跟原来的重复

(2)数据存放的文件名称不能重复，比如： <kahaDBdirectory="${activemq.data}/kahadb\_2"/>

(3)所有涉及的transportConnectors 的端口，都要跟前面的不一样

3:修改jetty.xml，主要就是修改端口，比如： <property name=“port” value=“8181”/> 端口必须和前面的不一样

4:到bin下面，复制一个activemq，比如叫做activemq2：

(1)修改程序的id，不能和前面的重复 ACTIVEMQ\_PIDFILE="$ACTIVEMQ\_DATA/activemq2-`hostname`.pid"

(2)修改配置文件路径 ACTIVEMQ\_CONF="$ACTIVEMQ\_BASE/conf2"

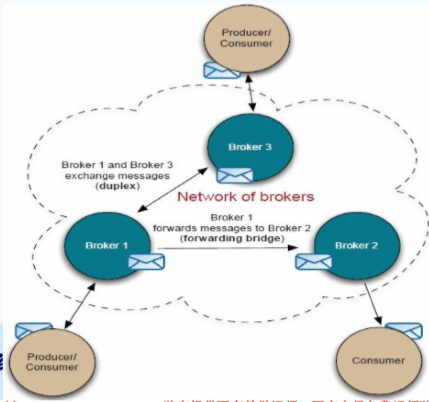
(3)修改端口，里面有个tcp的61616的端口，要改成不一样的，最好跟activemq.xml里面的tcp的端口一致

(4)然后就可以执行了，如果执行没有权限的话，就授权：chmod751 activemq2

(5) 如果(3),(4)两步没有也没必要补足。

### 9.2 ACtiveMQ broker网络

ActiveMQ的networkConnector是什么 在某些场景下，需要多个ActiveMQ的Broker做集群，那么就涉及到Broker到Broker的通信，这个被称为ActiveMQ的networkConnector。 ActiveMQ的networkConnector默认是单向的，一个Broker在一端发送消息，另一Broker在另一端接收消息。这就是所谓的“桥接”。 ActiveMQ也支持双向链接，创建一个双向的通道对于两个Broker，不仅发送消息而且也能从相同的通道来接收消息，通常作为duplex connector来映射，如下



#### 9.2.1 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；另一层意思，它是基于现有的网络Broker去发现其他可用的Brokers。 有两种配置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

#### 9.2.2 Static networks

Static networkConnector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 1：

#### 9.2.3 <broker>中配置

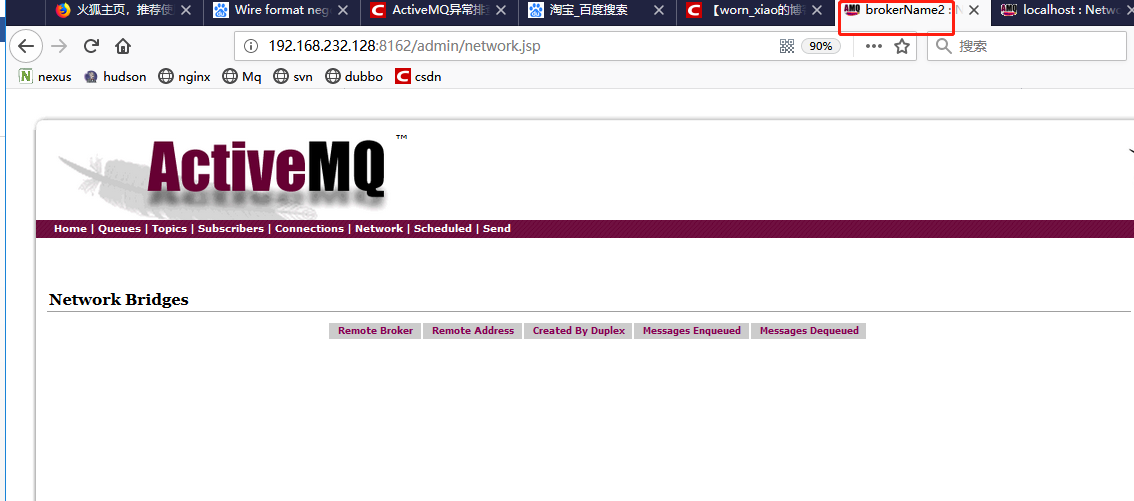
<networkConnectors>

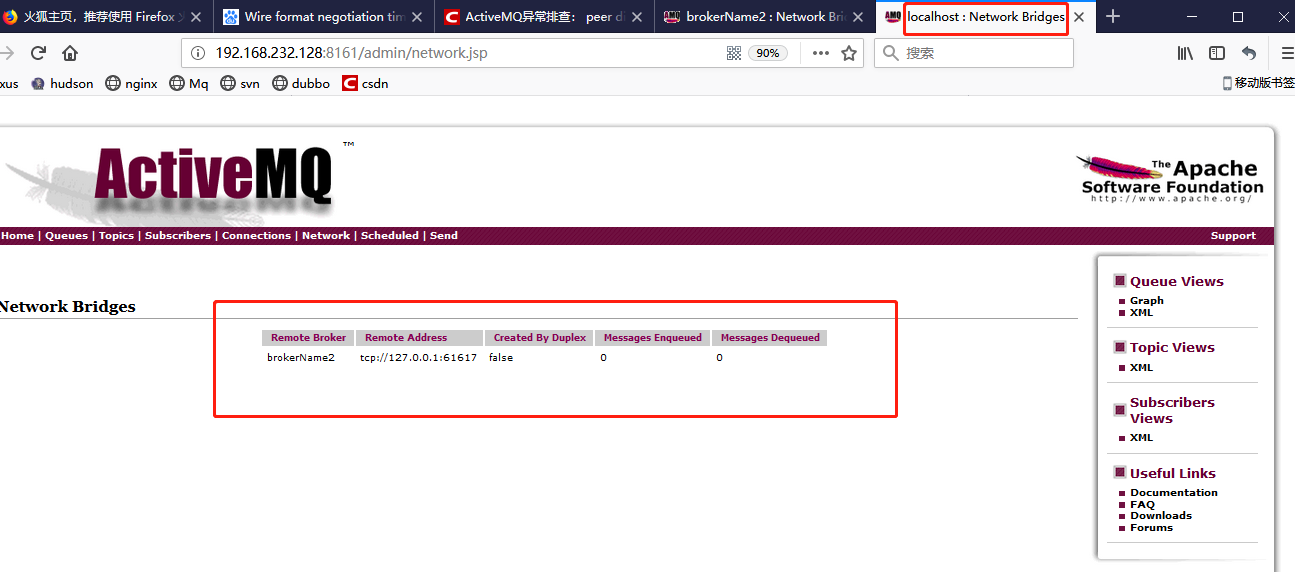
<networkConnector name="localnetwork" uri="static://(tcp://localhost:61616,tcp:// localhost:61617)"/>

</networkConnectors>

#### 9.2.4 演示结果

Broker2

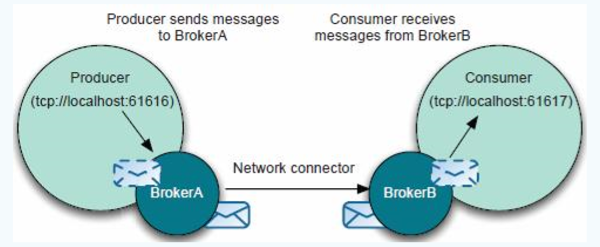




可以从上面的两个看出已经搭建起来了一个伪集群了。

Static networkConnector的基本原理示意图：

上图中，两个Brokers是通过一个static的协议来网络链接的。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。也就是，BrokerA会转发消息到 BrokerB上。



现在生产者向61616生成数据

执行发送者代码

**public** **class** PersisiTopicSender {

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

ActiveMQConnectionFactory conFactory = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection createConnection = conFactory.createConnection();

Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Topic createTopic = createSession.createTopic("persisitent");

MessageProducer createProducer = createSession.createProducer(createTopic);

createProducer.setDeliveryMode(DeliveryMode.***PERSISTENT***);

createConnection.start();

**for**(**int** i=0;i<3;i++){

TextMessage createTextMessage = createSession.createTextMessage("message"+i);

createProducer.send(createTextMessage);

}

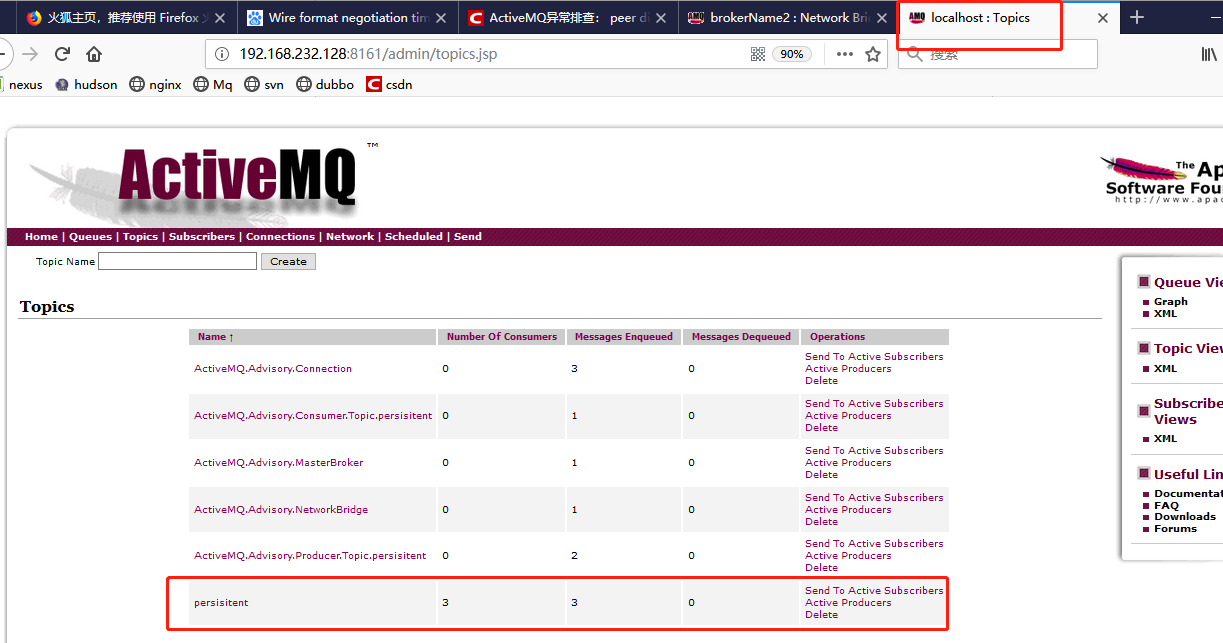
createSession.commit();

createSession.close();

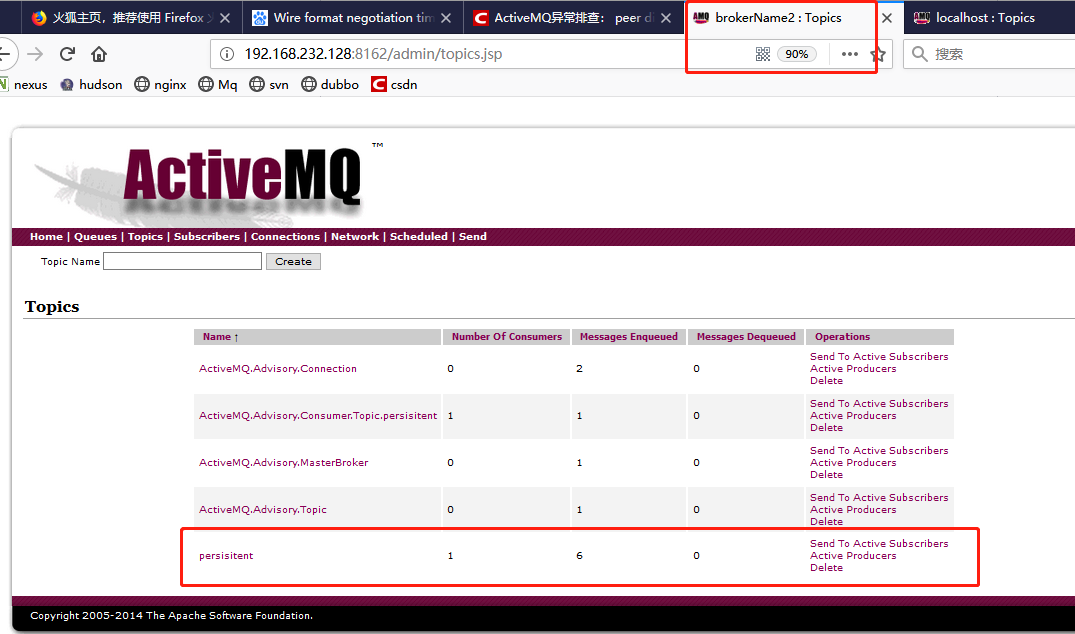
createConnection.close();

}

}



现在我到61617这个broker实例上去消费



此时在61616上已经可以看到相同的队列，并且运行一次消费者代码注册了一次了。

此时把消费者代码停止了，再运行一次就可以消费消息了

**public** **class** PersisiTopicReceiver {

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

ActiveMQConnectionFactory activeMQConnectionFactory=**new** ActiveMQConnectionFactory("tcp://192.168.232.128:61617");

Connection createConnection = activeMQConnectionFactory.createConnection();

createConnection.setClientID("订阅者B\_ID");

createConnection.start();

Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Topic createTopic = createSession.createTopic("persisitent");

TopicSubscriber createDurableSubscriber = createSession.createDurableSubscriber(createTopic, "T1");

TextMessage message = (TextMessage)createDurableSubscriber.receive();

**while**(message!=**null**){

System.***out***.println(message.getText());

message = (TextMessage)createDurableSubscriber.receive();

}

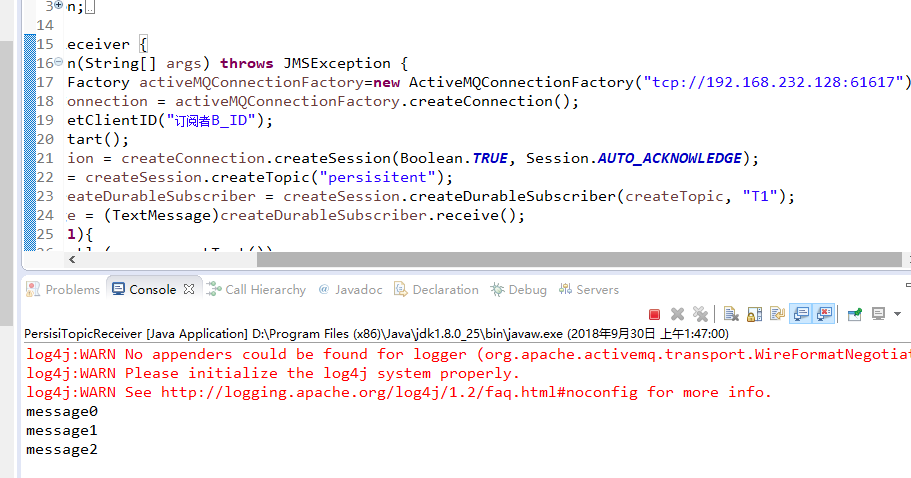
createSession.commit();

createSession.close();

createConnection.close();

}

}

如上已经消费到消息了。

#### 9.2.5 静态网络属性

networkConnector配置的可用属性：

1：name：默认是bridge

2：dynamicOnly：默认是false，如果为true, 持久订阅被激活时才创建对应的网路持久订阅。默认是启动时激活

3：decreaseNetworkConsumerPriority：默认是false。设定消费者优先权，如果为true，网络的消费者优先级降低 为-

5。如果为false，则默认跟本地消费者一样为0 4：networkTTL ：默认是1 ，网络中用于消息和订阅消费的broker数量

5：messageTTL ：默认是1 ，网络中用于消息的broker数量

6：consumerTTL：默认是1 ，网络中用于消费的broker数量

7：conduitSubscriptions ：默认true，是否把同一个broker的多个consumer当做一个来处理

8：dynamicallyIncludedDestinations ：默认为空，要包括的动态消息地址，类似于excludedDestinations，如： <dynamicallyIncludedDestinations> <queue physicalName="include.test.foo"/> <topic physicalName="include.test.bar"/> </dynamicallyIncludedDestinations>

9：staticallyIncludedDestinations ：默认为空，要包括的静态消息地址。类似于excludedDestinations，如： <staticallyIncludedDestinations> <queue physicalName="always.include.queue"/> </staticallyIncludedDestinations>

10：excludedDestinations ：默认为空，指定排除的地址，示例如下： <networkConnectors> <networkConnectoruri="static://(tcp://localhost:61617)" name="bridge" dynamicOnly="false" conduitSubscriptions="true" decreaseNetworkConsumerPriority="false"> <excludedDestinations> <queue physicalName="exclude.test.foo"/> <topic physicalName="exclude.test.bar"/> </excludedDestinations> <dynamicallyIncludedDestinations> <queue physicalName="include.test.foo"/> <topic physicalName="include.test.bar"/> </dynamicallyIncludedDestinations> <staticallyIncludedDestinations> <queue physicalName="always.include.queue"/> <topic physicalName="always.include.topic"/> </staticallyIncludedDestinations> </networkConnector> </networkConnectors>

11：duplex ：默认false，设置是否能双向通信

12：prefetchSize ：默认是1000，持有的未确认的最大消息数量，必须大于0，因 为网络消费者不能自己轮询消息

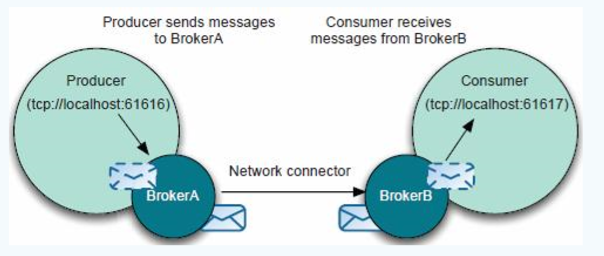
13：suppressDuplicateQueueSubscriptions：默认false，如果为true, 重复的订 阅关系一产生即被阻止

14：bridgeTempDestinations ：默认true，是否广播advisory messages来创建临 时destination 15：alwaysSyncSend ：默认false，如果为true，非持久化消息也将使用 request/reply方式代替oneway方式发送到远程broker。

16：staticBridge ：默认false，如果为true，只有 staticallyIncludedDestinations中配置的destination可以被处理。

如上属性所示，11就是让borker1与broker2可以互相通信的属性配置，实际上之前的演示消息是只能从61616流转至61617的，如果要让消息可以回流行程一个网，那么我们要在networkConnector上配置duplex这样消息队列才会形成一个网状的集群。

#### 9.2.6 消息回流



配置完duplex以后，我们会认为他就成为了一个集群了，实际上不是这样的，duplex只是决定了，生产者不管是在61616,还是61617我们都可以在网络连接的另一方通过消费者消费到消息。但是当消息从61616流转带61617以后，没有被消费完的这部分消息是不会因为消费者在61616消费而回流的。这就是我们所说的消息丢失。这是一个严重的问题。

<destinationPolicy>

<policyMap>

<policyEntries>

<policyEntry queue=">"

enableAudit="false">

<networkBridgeFilterFactory>

<conditionalNetworkBridgeFilterFactory replayWhenNoConsumers="true"/> </networkBridgeFilterFactory>

</policyEntry>

</policyEntries>

</policyMap>

</destinationPolicy>

此时就要求我们在activemq.xml中进行配置，能够让消息进行回流，真正的行程一个集群。

加上上面这一段，尽量两台机都要进行配置。

##### 9.2.6.1 发送者

**public** **class** MsgSendder {

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

ActiveMQConnectionFactory ConnectionFactoryconnectionFactory = **new** ActiveMQConnectionFactory("tcp://192.168.232.128:61617");

Connection connection = ConnectionFactoryconnectionFactory.createConnection();

connection.start();

Session session = connection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

Destination destination = session.createQueue("persisitent");

MessageProducer producer = session.createProducer(destination);

**for** (**int** i = 0; i < 20; i++) {

TextMessage message = session.createTextMessage("message--" + i);

producer.send(message);

}

session.commit();

session.close();

connection.close();

}

}

##### 9.2.6.2 61616消费者

**public** **class** PersisiTopicReceiver {

**public** **static** **void** main(String[] args) **throws** JMSException, InterruptedException {

ActiveMQConnectionFactory activeMQConnectionFactory=**new** ActiveMQConnectionFactory("tcp://192.168.232.128:61616");

Connection createConnection = activeMQConnectionFactory.createConnection();

createConnection.setClientID("B\_ID");

createConnection.start();

**final** Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

**for**(**int** i=0;i<30;i++){

Queue destination = createSession.createQueue("persisitent");

MessageConsumer consumer = createSession.createConsumer(destination);

consumer.setMessageListener(**new** MessageListener() {

**public** **void** onMessage(Message message) {

TextMessage msg = (TextMessage)message;

System.***out***.println(msg);

**try** {

Thread.*sleep*(2000);

createSession.commit();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (JMSException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

});

}

createSession.close();

createConnection.close();

}

}

##### 9.2.6.3 61617消费者

**public** **class** PersisiTopicReceiver2 {

**public** **static** **void** main(String[] args) **throws** JMSException, InterruptedException {

ActiveMQConnectionFactory activeMQConnectionFactory=**new** ActiveMQConnectionFactory("tcp://192.168.232.128:61617");

Connection createConnection = activeMQConnectionFactory.createConnection();

createConnection.setClientID("B\_ID2");

createConnection.start();

**final** Session createSession = createConnection.createSession(Boolean.***TRUE***, Session.***AUTO\_ACKNOWLEDGE***);

**for**(**int** i=0;i<30;i++){

Queue destination = createSession.createQueue("persisitent");

MessageConsumer consumer = createSession.createConsumer(destination);

consumer.setMessageListener(**new** MessageListener() {

**public** **void** onMessage(Message message) {

TextMessage msg = (TextMessage)message;

System.***out***.println(msg);

**try** {

createSession.commit();

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (JMSException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

});

}

createSession.close();

createConnection.close();

}

}

如上所示的配置，加上测试代码，就可以实现消息的互相通信与消息的回流了，这就形成了一个消息队列的伪集群了。