# ActiveMQ

## 一 容错机制

Failover Protocol

前面讲述的都是Client配置链接到指定的broker上。但是，如果Broker的链接失败怎么办呢？此 时，Client有两个选项：要么立刻死掉，要么去连接到其它的broker上。 Failover协议实现了自动重新链接的逻辑。这里有两种方式提供了稳定的brokers列表对于 Client链接。第一种方式：提供一个static的可用的Brokers列表。第二种方式：提供一个dynamic 发 现的可用Brokers。

Failover Protocol 的配置方式

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

Failover Protocol 的默认配置

默认情况下，这种协议用于随机的去选择一个链接去链接，如果链接失败了，那么会链接到其他 的Broker上。默认的配置定义了延迟重新链接，意味着传输将会在10秒后自动的去重新链接可用的 broker。当然所有的重新链接参数都可以根据应用的需要而配置。

Failover Protocol 的使用示例，在客户端程序里面：

ConnectionFactoryconnectionFactory= new

ActiveMQConnectionFactory("failover:(tcp://192.168.1.106:61679,tcp://192.168.1.106:61819) ?randomize=false");

简单的说：容错机制就是客户端可以在服务器端，一个服务器挂了之后，通过failover协议连接到另外的服务器端。

Failover Protocol 可用的配置参数：

1：initialReconnectDelay：在第一次尝试重连之前等待的时间长度（毫秒），默认10

2：maxReconnectDelay：最长重连的时间间隔（毫秒），默认30000

3：useExponentialBackOff：重连时间间隔是否以指数形式增长，默认true

4：backOffMultiplier：递增倍数，默认2.0

5：maxReconnectAttempts： 默认-1|0，自版本5.6起：-1为默认值，代表不限重试次数；0代表从不重试 （只尝试连接一次，并不重连），5.6以前的版本：0为默认值，代表不限重试次数所有版本：如果设置 为大于0的数，代表最大重试次数

6：startupMaxReconnectAttempts：初始化时的最大重连次数。一旦连接上，将使用maxReconnectAttempts 的配置，默认0

7：randomize：使用随机链接，以达到负载均衡的目的，默认true

8：backup：提前初始化一个未使用连接，以便进行快速失败转移，默认false

9：timeout：设置发送操作的超时时间（毫秒），默认-1

10：trackMessages：设置是否缓存[故障发生时]尚未传送完成的消息，当broker一旦重新连接成功，便将 这些缓存中的消息刷新到新连接的代理中，使得消息可以在broker切换前后顺利传送，默认false

11：maxCacheSize：当trackMessages启用时，缓存的最大字节，默认为128\*1024bytes

12：updateURIsSupported：设定是否可以动态修改broker uri（自版本5.4起），默认true

## 二 集群

### 2.1 Queue consumer 集群

ActiveMQ支持Consumer对消息高可靠性的负载平衡消费，如果一个Consumer死掉， 该消息会转发到其它的Consumer消费的Queue上。如果一个Consumer获得消息比其它 Consumer快，那么他将获得更多的消息。因此推荐ActiveMQ的Broker和Client使用

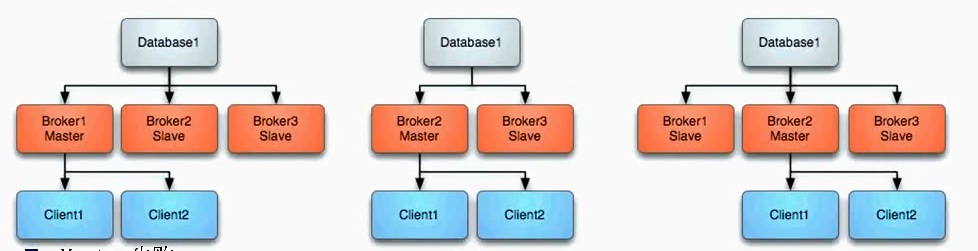
failover：//transport的方式来配置链接。 如何实现消费者的负载均衡呢，这个就要更改networkConnector的属性配置conduitSubscriptions ：默认true，是否把同一个broker的多个consumer当做一个来处理,保证在一个消费者端的多个消费者不被当成一个，按比例分配。

### 2.2 Broker 集群

大部情况下是使用一系列的Broker和Client链接到一起。如果一个Broker死掉了， Client可以自动链接到其它Broker上。实现以上行为需要用failover协议作为Client。 如果启动了多个Broker，Client可以使用static discover或者 Dynamic discovery 容易的从一个broker到另一个broker直接链接。 这样当一个broker上没有Consumer的话，那么它的消息不会被消费的，然而该 broker会通过存储和转发的策略来把该消息发到其它broker上。 特别注意：ActiveMQ默认的两个broker，static链接后是单方向的，broker-A可以 访问消费broker-B的消息，如果要支持双向通信，需要在netWorkConnector配置的时候， 设置duplex=true，并且配置上消息回流机制就可以了。

#### 2.2.1 方式一 JDBC

利用数据库作为数据源，采用Master/Slave模式，其中在启动的时候Master首先获 得独有锁，其它Slaves Broker则等待获取独有锁。 推荐客户端使用Failover来链接Brokers。 具体如下图所示：



如果Master失败，则它释放独有锁，其他Slaver则获取独有锁，其它Slaver立即获得独有锁后此时它将变成Master，并且启动所有的传输链接。同时，Client将停止链接之前的Master和将会轮询链接到其他可以利用的Broker即新Master。如上中图所示

Master重启 任何时候去启动新的Broker，即作为新的Slave来加入集群，如上右图所示

JDBC Master Slave的配置 使用<jdbcPersistenceAdapter/>来配置消息的持久化，自动就会使用JDBC Master Slave的方式。

#### 2.2.2 方式二 桥接

##### 2.2.2.1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"

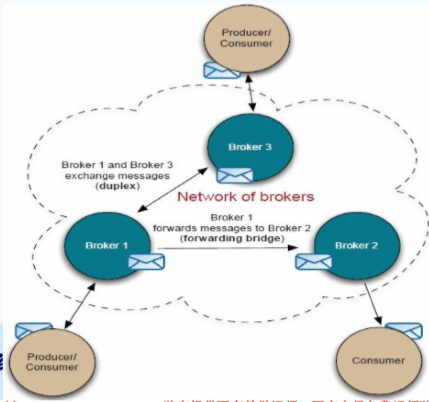
(3)修改端口，里面有个tcp的61616的端口，要改成不一样的，最好跟activemq.xml里面的tcp的端口一致

(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。

##### 2.2.2.2 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



###### 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

###### 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：

#### 2.2.3 <broker>中配置

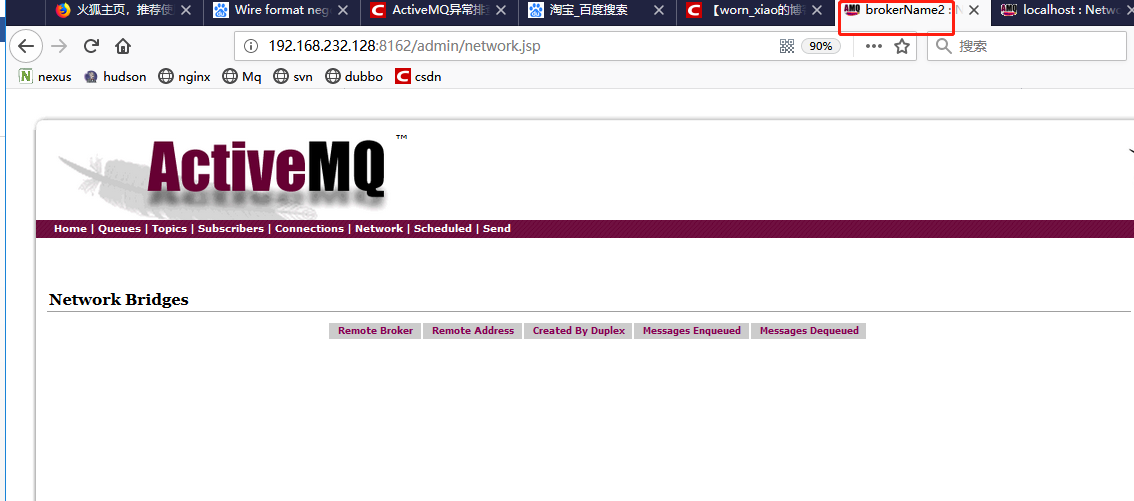
<networkConnectors>

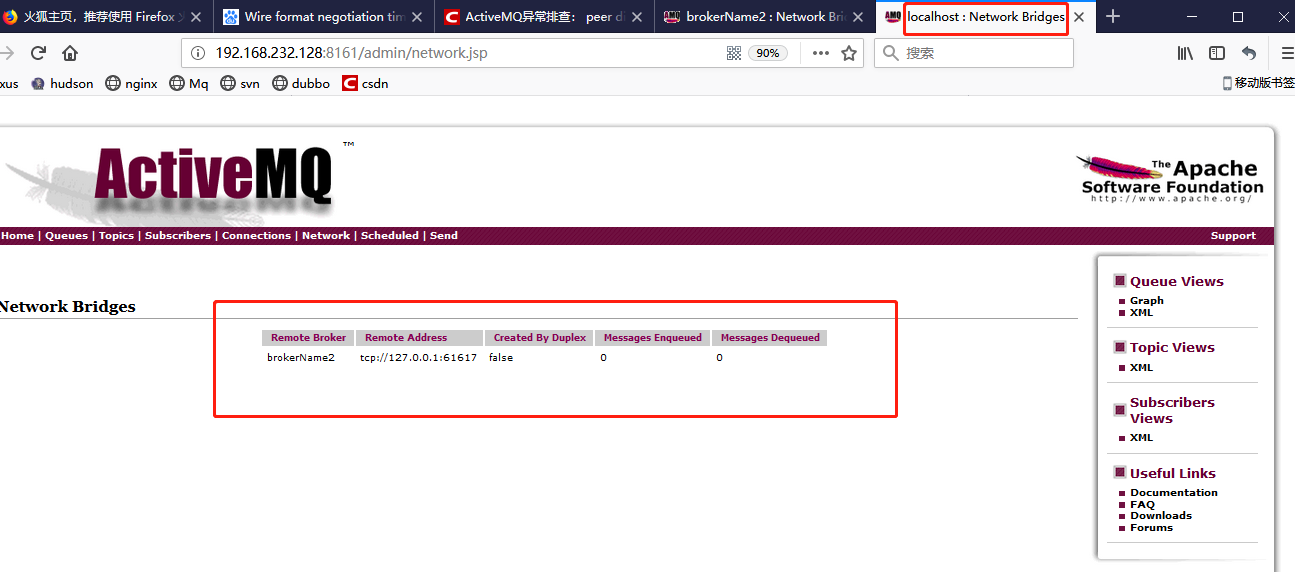
<networkConnector name="localnetwork" uri="static://(tcp://localhost:61616,tcp:// localhost:61617)"/>

</networkConnectors>

#### 2.2.4 演示结果

Broker2

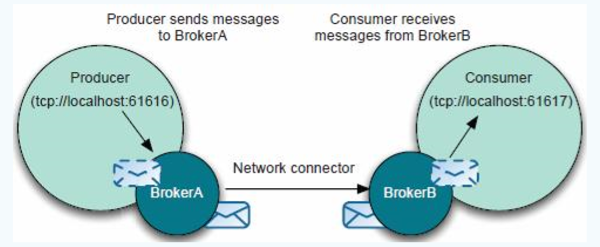




可以从上面的两个看出已经搭建起来了一个伪集群了。

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);

}

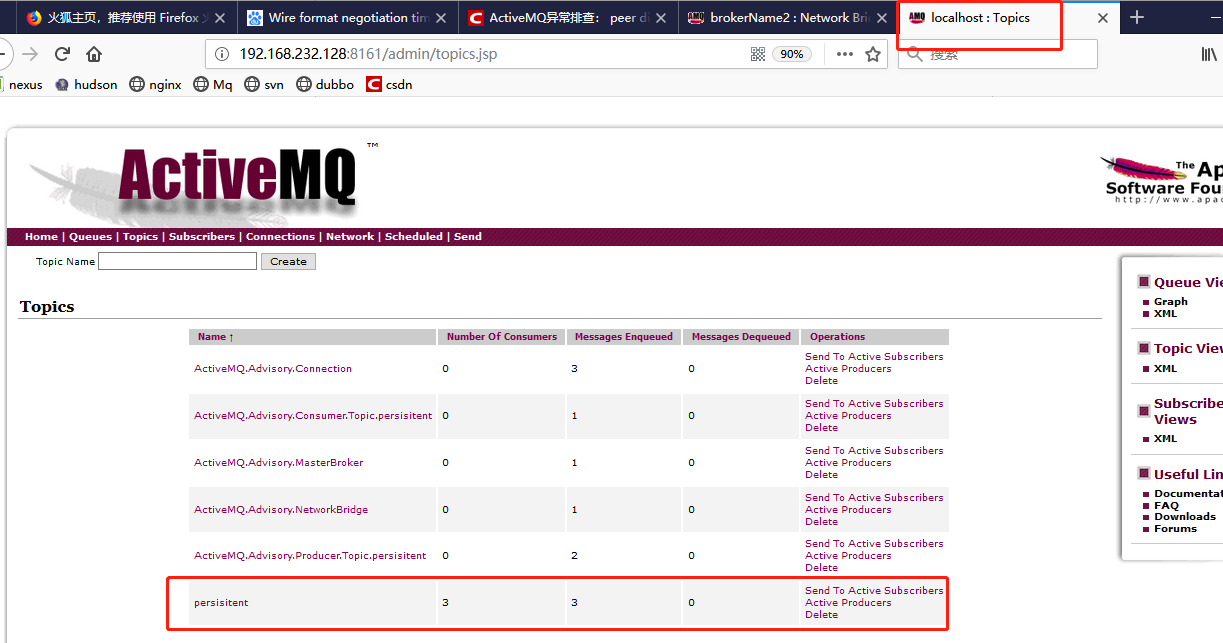
createSession.commit();

createSession.close();

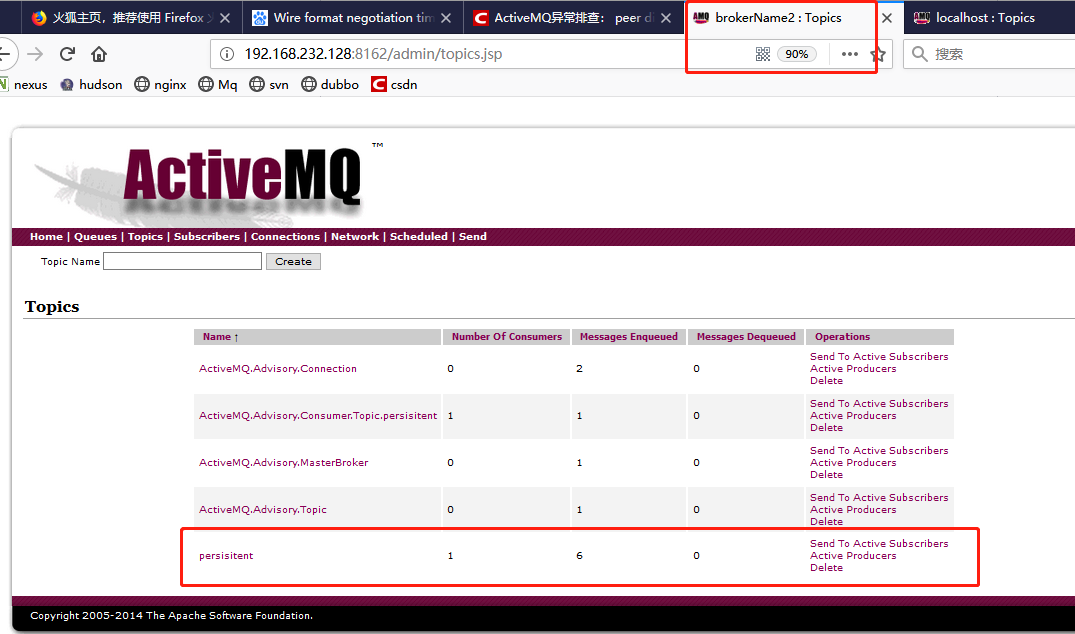
createConnection.close();

}

}



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();

}

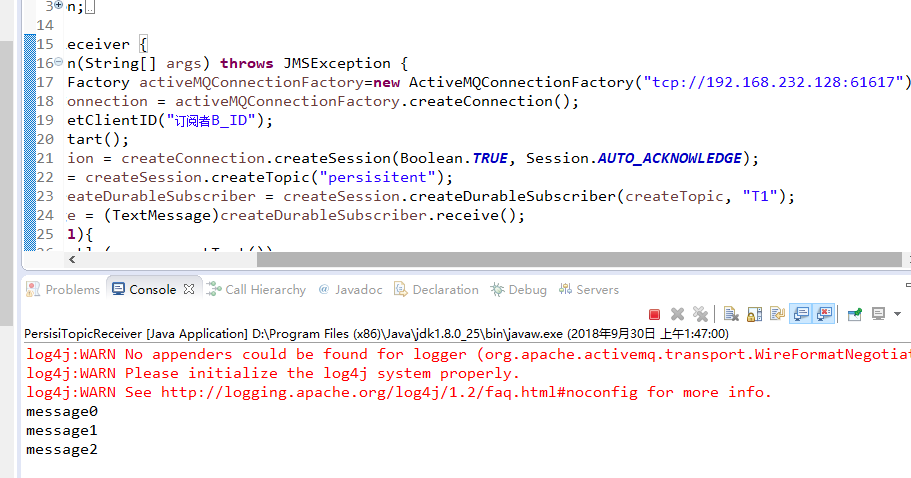
createSession.commit();

createSession.close();

createConnection.close();

}

}

如上已经消费到消息了。

#### 2.2.5 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

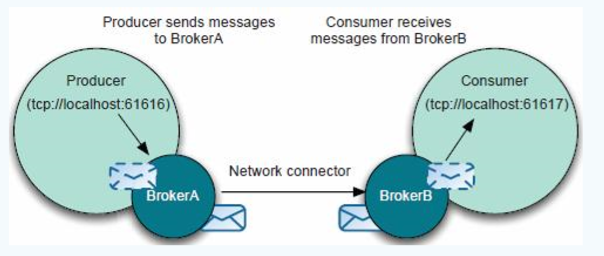
13：suppressDuplicateQueueSubscriptions：默认false，如果为true, 重复的订 阅关系一产生即被阻止

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。

#### 2.2.6 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。

##### 2.2.6.1 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();

}

}

##### 2.2.6.2 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();

}

}

##### 2.2.6.3 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。

## 三 ActiveMQ数据持久化

### 3.1 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

### 3.2 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>

### 3.3 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

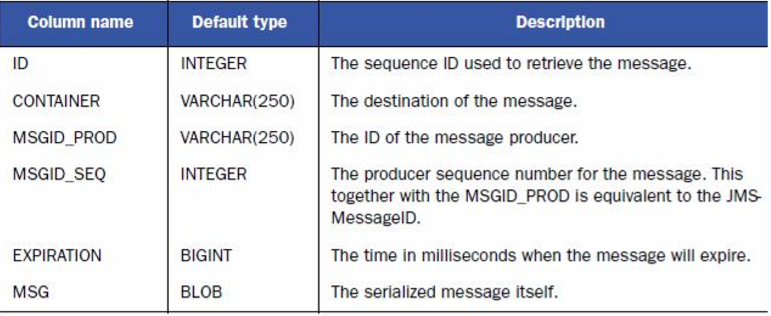
17：concurrentStoreAndDispatchTopics：是否分发Topic消息到客户端，同时进行存储，默认true

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

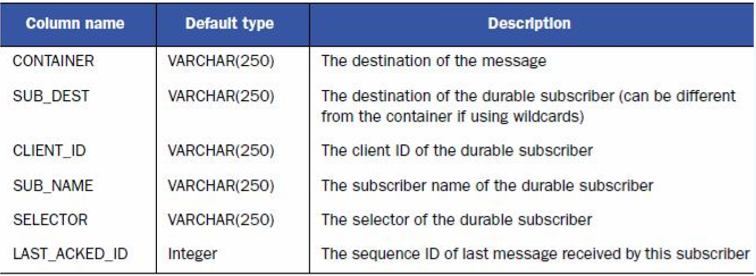
### 3.4 JDBC持久化

#### 3.4.1 建表

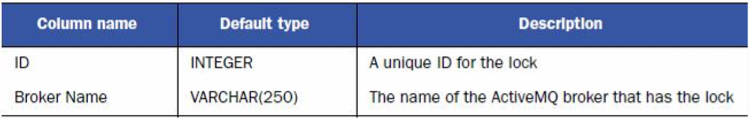
ACTIVEMQ\_MSGS 消息表



ACTIVEMQ\_ACKS 确认表



ACTIVEMQ\_LOCK 锁表



#### 3.4.2 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>

### 3.5 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

#### 3.5.1 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>

#### 3.5.2 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

### 3.6 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

#### 3.5.1 配置

<beans>

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

<transportConnectors>

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

</transportConnectors>

</broker>

</beans>

#### 3.5.2 内嵌使用Broker

public void createEmbeddedBroker() throws Exception {

BrokerServicebroker = new BrokerService();

broker.setPersistent(false);

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

broker.start();

}