原文链接：<http://blog.csdn.net/kimmking/article/details/8440150/>

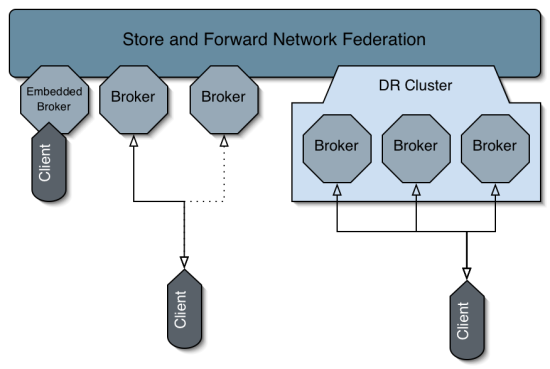
ActiveMQ集群：网络连接模式(network connector)详解

# 网络连接模式(network connector)

针对海量消息所要求的横向扩展性和系统的高可用性，ActiveMQ提供了网络连接模式的集群功能。简单的说，就是通过把多个不同的broker实例连接在一起，作为一个整体对外提供服务，从而提高整体对外的消息服务能力。通过这种方式连接在一起的broker实例之间，可以共享队列和消费者列表，从而达到分布式队列的目的。

## 拓扑结构

几种不同的ActiveMQ部署拓扑结构（嵌入、主从复制、网络连接）：



## 配置示例

在activemq.xml的broker节点内添加：

<networkConnectors>

      <networkConnectoruri="static:(tcp://localhost:62001)"/>

</networkConnectors>

uri也可以使用其他两种方式：

1.          multicast://default

2.          masterslave:(tcp://host1:61616,tcp://host2:61616,tcp://..)

其中masterslave方式的第一个url需要是master，其他是slave。

一个broker的实例内可以配置多个networkConnector，如果有两个以上的networkConnector指向同一个broker，则需要显式的指定name。

## 静态URI配置

使用静态URI方式可以指定多个URL，networkConnector将连接到每一个broker。

<networkConnectors>

      <networkConnector uri="static:(tcp://host1:61616,tcp://host2:61616,tcp://..)"/>

</networkConnectors>

URI的几个属性：

|  |  |  |
| --- | --- | --- |
| **属性** | **默认值** | **描述** |
| initialReconnectDelay | 1000 | 重连之前等待的时间(ms) (如果useExponentialBackOff为 false) |
| maxReconnectDelay | 30000 | 重连之前等待的最大时间(ms) |
| useExponentialBackOff | true | 每次重连失败时是否增大等待时间 |
| backOffMultiplier | 2 | 增大等待时间的系数 |

## networkConnector配置

### 配置参数

|  |  |  |
| --- | --- | --- |
| **属性** | **默认值** | **描述** |
| name | bridge | 名称 |
| dynamicOnly | 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。 |
| decreaseNetworkConsumerPriority | false | 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. |
| networkTTL | 1 | 消息和订阅在网络上通过的broker数量 |
| conduitSubscriptions | true | 多个网络消费者是否被当做一个消费者来对待。 |
| excludedDestinations | empty | 不通过网络转发的destination |
| dynamicallyIncludedDestinations | empty | 通过网络转发的destinations，注意空列表代表所有的都转发。 |
| staticallyIncludedDestinations | empty | 匹配的都将通过网络转发-即使没有对应的消费者 |
| duplex | false | 如果为true，则既可消费又可生产消息到网络broker |
| prefetchSize | 1000 | 设置网络消费者的[prefetch size](http://activemq.apache.org/what-is-the-prefetch-limit-for.html)参数。必须大于0，因为网络消费者不能自己轮询消息。 |
| suppressDuplicateQueueSubscriptions | false | (从5.3版本开始) 如果为true, 重复的订阅关系一产生即被阻止。 |
| bridgeTempDestinations | true | 是否广播advisory messages来创建临时destination。 |
| alwaysSyncSend | false | (从 5.6版本开始) 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。 |
| staticBridge | false | (从5.6版本开始) 如果为true，只有staticallyIncludedDestinations中配置的destination可以被处理。 |

networkConnector的实现原理是基于ActiveMQ的公告消息（AdvisoryMessage）机制的（参见[此处](http://activemq.apache.org/advisory-message.html)）。当broker2通过networkConnector、duplex方式指向broker1时，发生了什么呢？

假定broker1已经启动，这时候broker2开始启动。

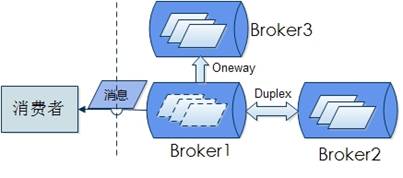
1.         broker2先启动自己的connector

2.         然后使用一个vm的connector，创建一个connection，把自己作为一个client，连接到broker1。

3.         通过订阅Advisory Message，拿到相互的Consumer与相应的Queue列表。

至此，双方建立关系。

然后通过broker1的转发，broker1上的消费者，就可以消费broker2的queue上的消息。这个过程可以看做一个消息被消费了两次，broker1作为消费者，消费掉broker2上的消息，broker1再作为broker，把消息投递给实际的消费者。



### 管道订阅(conduit subscription)

conduitSubscriptions选择决定网络消费者在所有消费者中的比重。假如有2个同一个远程的broker1上的网络消费者和一个broker2的本地消费者。

1.        conduitSubscriptions为true，则2个网络消费者只相当于一个消费者，broker1仅仅在broker2上注册了一个消费者。这时往broker2上发送300个消息，2个网络消费者各接收到75个消息，一个本地消费者接收到150 消息。

2.        conduitSubscriptions为false，则3个消费者平分所有消息，broker1在broker2上将注册了两个消费者。这时往broker2上发送300个消息，2个网络消费者和本地消费者一样，各接收到100个消息。

### 双向网络连接(duplex networkConnector)

默认NetworkConnector在需要转发消息时是单向连接的。当duplex=true时，就变成了双向连接，这时配置在broker2端的指向broker1的duplex networkConnector，相当于即配置了

broker2到broker1的网络连接，也配置了broker1到broker2的网络连接。（就是说不管broker1同意与否，都被绑架了。）当然，仅仅在broker1上配置也有同样的效果。

注意：可以在两个broker间建立两个以上的双向网络连接来增加吞吐量或对主题\队列分区，只需要指定他们使用不同的name即可。

### 指定和限制Destination

通过NetworkConnector共享的destination太多，传输的Advisory Message就会变的非常多，系统的拓扑结构将变得非常复杂，所有才有多种方式来限制这些destination配置：

1.        dynamicallyIncludedDestinations

ü  这里匹配到的destination，在需要时将被转发

2.        excludedDestinations

ü  这里匹配到的destination，将不会被转发

3.        staticallyIncludedDestinations

ü  如果指定了staticBridge为true，则只有这里匹配的destination可以被转发。此时本地broker完全被其他broker代理。并且本broker不会订阅其他broker上的AdvisoryMessage，也不会获取任何远程consumer信息。

这几个配置可以使用通配符，比如“>”，详见[wildcard](http://activemq.apache.org/wildcards.html)。

示例代码：

<networkConnectors>

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

         name="bridge"

         conduitSubscriptions="true"

         decreaseNetworkConsumerPriority="false">

         <dynamicallyIncludedDestinations>

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

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

         </dynamicallyIncludedDestinations>

         <excludedDestinations>

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

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

         </excludedDestinations>

        <staticallyIncludedDestinations>

                   <queue physicalName="always.include.queue"/>

                   <topic physicalName="always.include.topic"/>

         </staticallyIncludedDestinations>

      </networkConnector>

    </networkConnectors>

此外，从5.6版本起，可以在networkConnector上设置destinationFilter来指定感兴趣的Advisory Message将被传播。

<networkConnector uri="static:(tcp://host)" destinationFilter="Queue.include.test.foo,ActiveMQ.Advisory.Consumer.Topic.include.test.bar">

 <dynamicallyIncludedDestinations>

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

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

 </dynamicallyIncludedDestinations>

</networkConnector>

### 被卡住的消息

一个很有意思的场景是，broker1和broker2通过networkConnector连接。一些个consumers连接到broker1，消费broker2上的消息。消息先被broker1从broker2上消费掉，然后转发给这些consumers。不幸的是转发部分消息的时候broker1重启了，这些consumers发现broker1连接失败，通过failover连接到broker2上去了，但是有一部分他们还没有消费的消息被broker2已经分发到了broker1上去了。这些消息，就好像是消失了，除非有消费者重新连接到broker1上来消费。怎么办呢？

办法就是从5.6版本destinationPolicy上新增的选项replayWhenNoConsumers。这个选项使得broker1上有需要转发的消息但是没有消费者时，把消息回流到它原始的broker。同时把enableAudit设置为false，为了防止消息回流后被当做重复消息而不被分发。

<destinationPolicy>

      <policyMap>

        <policyEntries>

          <policyEntry queue="TEST.>" >

            <conditionalNetworkBridgeFilterFactory replayWhenNoConsumers="true" enableAudit="false"/>

          </policyEntry>

        </policyEntries>

      </policyMap>

    </destinationPolicy>

更详细的讨论见这里：

<http://tmielke.blogspot.de/2012/03/i-have-messages-on-queue-but-they-dont.html>

### 其他说明

1.        NetworkConnector基于AdvisoryMessage机制，如果broker的advisorySupport选型被禁用，则NetworkConnector将不起作用。

2.        用作转发的broker中入列出列这些统计信息只记录其转发的数据。

3.        用作转发的broker中无法看到远程broker的相同队列中的数据（browse消息列表为空，queuesize为0）。

## 使用示例

下载ActiveMQ 5.7版本，其中带了包含static network connector的例子。

即配置文件activemq-static-network-broker1.xml和activemq-static-network-broker2.xml。

他们分别使用端口 tcp://localhost:61616和tcp://localhost:61618

static network connector在第二个文件里。

分别使用这两个配置文件启动两个broker实例（先启动broker1，再启动broker2）。

在broker2的控制台看到：

INFO | Establishing network connection fromvm://static-broker2?async=false&network=true to tcp://localhost:61616

 INFO |Connector vm://static-broker2 Started

 INFO |Network Connector DiscoveryNetworkConnector:NC:BrokerService[static-broker2]Started

 INFO |Apache ActiveMQ 5.7.0 (static-broker2, ID:kimmking-2270-1356502079016-0:1)started

 INFO |For help or more information please see: http://activemq.apache.org

 INFO |Network connection between vm://static-broker2#0 andtcp://localhost/127.0.0.1:61616 @2271(static-broker1) has been established.

在broker1的控制台看到：

INFO | Network connection betweenvm://static-broker1#0 and tcp:///127.0.0.1:1710@61616 (static-broker2) has beenestablished.

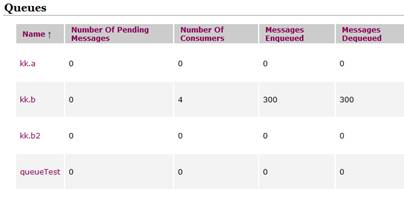
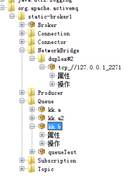
在命令行输入jconsole，然后分别通过下列url连接jmx控制台来管理broker：

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

ü  service:jmx:rmi:///jndi/rmi://localhost:1100/jmxrmi

也可以再这两个xml中配置jetty来使用web控制台查看和管理。

然后可以通过代码在两个broker中通过static network connector存取消息。



测试代码：

**[java]** [view plaincopy](http://blog.csdn.net/kimmking/article/details/8440150/)

1. **package** org.qsoft.activemq.test;
3. **import** java.util.concurrent.atomic.AtomicInteger;
5. **import** javax.jms.JMSException;
6. **import** javax.jms.Message;
7. **import** javax.jms.MessageListener;
8. **import** javax.jms.MessageProducer;
9. **import** javax.jms.Queue;
10. **import** javax.jms.QueueConnection;
11. **import** javax.jms.QueueConnectionFactory;
12. **import** javax.jms.QueueReceiver;
13. **import** javax.jms.QueueSession;
14. **import** javax.jms.Session;
15. **import** javax.jms.TextMessage;
17. **import** org.apache.activemq.ActiveMQConnectionFactory;
18. **import** org.apache.activemq.command.ActiveMQQueue;
20. **public** **class** TestReceiver {
22. /\*\*
23. \* @param args
24. \*/
25. **public** **static** **void** main(String[] args) {
26. **try** {
27. // init connection factory with activemq
28. QueueConnectionFactory factoryA = **new** ActiveMQConnectionFactory("tcp://127.0.0.1:61616");
29. // specify the destination
30. Queue queueB = **new** ActiveMQQueue("kk.b");
31. // create connection,session,consumer and receive message
32. QueueConnection connA = factoryA.createQueueConnection();
33. connA.start();
35. // first receiver on broker1
36. QueueSession sessionA1 = connA.createQueueSession(**false**, Session.AUTO\_ACKNOWLEDGE);
37. QueueReceiver receiverA1 = sessionA1.createReceiver(queueB);
38. **final** AtomicInteger aint1 = **new** AtomicInteger(0);
39. MessageListener listenerA1 = **new** MessageListener(){
40. **public** **void** onMessage(Message message) {
41. **try** {
42. System.out.println(aint1.incrementAndGet()+" => A1 receive from kk.b: " + ((TextMessage)message).getText());
43. } **catch** (JMSException e) {
44. e.printStackTrace();
45. }
46. }};
47. receiverA1.setMessageListener(listenerA1 );
49. // second receiver on broker1
50. QueueSession sessionA2 = connA.createQueueSession(**false**, Session.AUTO\_ACKNOWLEDGE);
51. QueueReceiver receiverA2 = sessionA2.createReceiver(queueB);
52. **final** AtomicInteger aint2 = **new** AtomicInteger(0);
53. MessageListener listenerA2 = **new** MessageListener(){
54. **public** **void** onMessage(Message message) {
55. **try** {
56. System.out.println(aint2.incrementAndGet()+" => A2 receive from kk.b: " + ((TextMessage)message).getText());
57. } **catch** (JMSException e) {
58. e.printStackTrace();
59. }
60. }};
61. receiverA2.setMessageListener(listenerA2 );
63. // a fake one on broker1
64. QueueReceiver receiverA3 = sessionA2.createReceiver(queueB);
65. **final** AtomicInteger aint3 = **new** AtomicInteger(0);
66. MessageListener listenerA3 = **new** MessageListener(){
67. **public** **void** onMessage(Message message) {
68. **try** {
69. System.out.println(aint3.incrementAndGet()+" => A3 receive from kk.b: " + ((TextMessage)message).getText());
70. } **catch** (JMSException e) {
71. e.printStackTrace();
72. }
73. }};
74. receiverA3.setMessageListener(listenerA3 );
76. QueueConnectionFactory factoryB = **new** ActiveMQConnectionFactory("tcp://127.0.0.1:61618");
77. Queue queueB1 = **new** ActiveMQQueue("kk.b");
78. QueueConnection connB = factoryB.createQueueConnection();
79. connB.start();
81. // one receiver on broker2
82. QueueSession sessionB1 = connB.createQueueSession(**false**, Session.AUTO\_ACKNOWLEDGE);
83. QueueReceiver receiverB1 = sessionB1.createReceiver(queueB);
84. **final** AtomicInteger bint1 = **new** AtomicInteger(0);
85. MessageListener listenerB1 = **new** MessageListener(){
86. **public** **void** onMessage(Message message) {
87. **try** {
88. System.out.println(bint1.incrementAndGet()+" => B1 receive from kk.b: " + ((TextMessage)message).getText());
89. } **catch** (JMSException e) {
90. e.printStackTrace();
91. }
92. }};
93. receiverB1.setMessageListener(listenerB1 );
95. // producer  on broker2
96. QueueSession sessionBp = connB.createQueueSession(**false**, Session.AUTO\_ACKNOWLEDGE);
97. MessageProducer producer = sessionBp.createProducer(queueB1);
98. **int** index = 0;
99. **while**(index++<300){
100. TextMessage message = sessionBp.createTextMessage(index + " from kk.b on broker2");
101. producer.send(message);
102. }

105. } **catch** (Exception e) {
106. e.printStackTrace();
107. System.exit(1);
108. }
110. }
112. }