## 配置队列与队列属性方式

1. **配置queue名称**

打开activemq.xml文件（目录位于conf目录下，以5.13版本为例，则该文件位于apache-activemq-5.13.0\conf目录下），在配置文件中加入queue标签。具体如下；

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

<destinations>

**<queue physicalName="FOO.1" name =”myQueue” />**

</destinations>

</broker>

其中physicalName是实际的队列名称，而name是别名。为了代码重用而准备

注：标签的位置必须位于destinations标签下；

1. **配置队列属性**

打开activemq.xml文件，在文件中加入policyEntry标签,开始队列属性的配置工作

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

<destinationPolicy>

<policyMap>

<policyEntries>

**<policyEntry queue=">" >**

**……**

**</policyEntry>**

</policyEntries>

</policyMap>

</destinationPolicy>

</broker>

注：

在activemq中policyEntry queue=">" 。”>” 表示通配符，即该规则适用于所有的queue，

即只需要配置一条所有的queue都必须满足的规则时使用。也可以指定某个特定队列。

## 队列属性配置内容

1. **配置死信队列**

死信队列用于存储过期未被接收以及重发多次仍失败的信息。缺省状态下所有队列的死信消息都被发送到同一个缺省死信队列，不便于管理，所以我们采用独立的死信处理，即每个队列都由自己的死信队列。

配置方法如下：

在activemq.xml配置文件中的policyEntry标签下增加deadLetterStrategy标签，具体如下：

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

<destinationPolicy>

<policyMap>

<policyEntries>

<policyEntry queue=">" >

**<deadLetterStrategy>**

**<individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true" processExpired="true" processNonPersistent="true" />**

**</deadLetterStrategy>**

</policyEntry>

</policyEntries>

</policyMap>

</destinationPolicy>

</broker>

**属性说明：**

individualDeadLetterStrategy：标签表示该队列的死信处理机制为独立的死信处理机制，即不和其他队列公用私信队列。

另：除此种方法外还有另外两种死信处理机制，具体如下：

1. SharedDeadLetterStrategy: 将所有的DeadLetter保存在一个共享的队列中，这是ActiveMQ broker端默认的策略。共享队列默认为“ActiveMQ.DLQ”，可以通过“deadLetterQueue”属性来设定具体如下：

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

<destinationPolicy>

<policyMap>

<policyEntries>

<policyEntry queue=">" >

<deadLetterStrategy>

<deadLetterStrategy>

**<sharedDeadLetterStrategy deadLetterQueue="DLQ-QUEUE"/></deadLetterStrategy>**

**</deadLetterStrategy>**

</policyEntry>

</policyEntries>

</policyMap>

</destinationPolicy>

</broker>

   （b）DiscardingDeadLetterStrategy: broker将直接抛弃DeadLeatter。如果开发者不需要关心DeadLetter，可以使用此策略。AcitveMQ提供了一个便捷的插件：DiscardingDLQBrokerPlugin，来抛弃DeadLetter。

<broker>

<plugins>

**<discardingDLQBrokerPlugin dropAll="true" dropTemporaryTopics="true" dropTemporaryQueues="true" />**

</plugins>

</broker>

deadLetterStrategy：在标签内配置死亡队列的相关属性。

queuePrefix：设定队列前缀，设定之后，如果某个队列的信息需要被存入死亡队列，则activemq将新建一个队列来存放信息，队列名为前缀拼接原队列名的。例如：原队列名为FirstQueue，如果此队列的信息需要进入死亡队列，则activemq将新建一个名为DLQ. FirstQueue的队列来存放相应信息。

processExpired：过期（超过设定时间仍未被派发）消息是否保存，如果需要保存则设定为true。如果设置为false则只保存重发一定次数后仍未被消费的信息

processNonPersistent：非持久化信息是否保存。默认不保存。

useQueueForQueueMessages: 设置使用队列保存死信，还可useQueueForTopicMessage即：创建一个Topic来保存死信。

注：重启后死信队列中的信息仍然存在，进入死信队列的信息可以通过view下的[Browse](http://localhost:8161/admin/browse.jsp?JMSDestination=DLQ.FirstQueue)，超链接进入显示页面点击相应信息进行内容查看。

如果只设置消息超时时间而不设置死亡队列，则消息将被丢弃。（但是从console页面中看和被正常消费尚未看出明显区别。）

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" >

<deadLetterStrategy>

<individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true" processExpired="true" processNonPersistent="true" />

</deadLetterStrategy>

</policyEntry>

测试内容如下：

注：第一行中 <policyEntry queue=">" >的“>”为通配符。表示该配置内容对所有queue有效。

测试步骤如下：

1，启动activemq服务。

2，利用sender类创建队列并向该队列发送消息。

3，信息过期后进入死亡队列。

测试二：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" >

<deadLetterStrategy>

<individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true" processExpired="false" processNonPersistent="true" />

</deadLetterStrategy>

</policyEntry>

测试内容如下：

1，启动activemq服务。

2，利用sender类创建队列并向该队列发送消息。

3，信息过期后被丢弃，未能进入死信队列。

测试三：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" >

<deadLetterStrategy>

<individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true" processExpired="true" processNonPersistent="false" />

</deadLetterStrategy>

</policyEntry>

测试内容如下：

1，启动activemq服务。

2，利用sender类创建队列并向该队列发送消息。

3，信息过期后被丢弃，未能进入死信队列。

1. **设置消息队列属性**

配置方法如下：

在activemq.xml配置文件中的policyEntry标签中增加相应标签，具体如下：

**属性说明：**

**producerFlowControl**：表示是否控制生产者生产数据的速度，默认为true，如果设置为false，消息就会存在磁盘中以防止内存溢出；

**memoryLimit**表示在producerFlowControl=”true”的情况下，消息存储在内存中最大量，当消息达到这个值时，ActiveMQ会减慢消息生产者发送消息的速度甚至阻塞。

**注意内容**：自从ActiveMQ 5.x中引入新的文件游标之后，非持久化消息被分流到了临时文件存储中，以此来减少非持久化消息传送使用的内存总量。结果就是，发现一个队列的内存限制永远达不到，因为游标不需要使用太多的内存。真的想把所有的非持久化消息存放在内存中，

并在达到内存限制的时候停掉生产者，需要配置vmQueueCursor属性

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" producerFlowControl="true" memoryLimit="1mb" >

</policyEntry>

测试内容如下：

1，启动activemq服务。

2，利用sender类创建队列并向该队列发送消息（消息内容是一个大小为1mb的文件中的内容）。

3，可以循环发送几百次以上，不会发生阻塞现象。

测试二：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" producerFlowControl="true" memoryLimit="1mb" >

<pendingQueuePolicy><vmQueueCursor/></pendingQueuePolicy>

</policyEntry>

测试内容如下：

1，启动activemq服务。

2，利用sender类创建队列并向该队列发送消息（消息内容是一个大小为1mb的文件中的内容）。

3，只能发送一次就发生阻塞现象。

**queuePrefetch：**负责预取机制，表示队列能以预取最大消息数。

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" queuePrefetch="1" >

</policyEntry>

测试内容如下：

1，启动activemq服务。

2，启动启动receiver。长时间监听队列，且监听器中接收到信息后，不返回确认信息。

3，利用sender类创建队列并分多次向该队列发送消息。

4，Receiver只可以接受到一条信息。

测试二：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" >

</policyEntry>

测试内容如下：

1，启动activemq服务。

2，启动启动receiver。长时间监听队列，且监听器中接收到信息后，不返回确认信息。

3，利用sender类创建队列并分多次向该队列发送消息。

4，Receiver可以接受到sender类发送的每一条信息。

**usePrefetchExtension**：属性作用是预取功能的扩展，使用该功能能增加consumer预取信息条数。

注:该属性需要和代码中设置的consumer.prefetchSize属性配合使用。

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" usePrefetchExtension="false" >

</policyEntry>

另：在java代码中创建目的地时增加consumer.prefetchSize参数，具体如下：

destination = session.createQueue("FirstQueue?consumer.prefetchSize=2");

测试内容如下：

1，启动activemq服务。

2，启动启动receiver。长时间监听队列，且监听器中接收到信息后，不返回确认信息。

3，利用sender类创建队列并分多次向该队列发送消息。

4，Receiver只可以接受到两条信息。

测试二：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" usePrefetchExtension="true" >

</policyEntry>

另：在java代码部分不变

测试内容如下：

1，启动activemq服务。

2，启动启动receiver。长时间监听队列，且监听器中接收到信息后，不返回确认信息。

3，利用sender类创建队列并分多次向该队列发送消息。

4，Receiver接收到2条信息后，还可以接收点三条数据。

**特别注意**： **queuePrefetch设定的是消费者最多预取消息的条数。**usePrefetchExtension拓展的消息数是针对java代码中consumer.prefetchSize属性而言。

**expireMessagesPeriod**：排队信息检查是否过期的时间间隔，默认是30000，单位ms。

注：如果只设置expireMessagesPeriod属性则不会产生作用。必须和代码中为信息设置的超时时间相配合。设定这个属性后，acvtemq会间隔相应的时间来检测，队列中的信息是否过期。

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" expireMessagesPeriod="60000" >

<deadLetterStrategy>

<individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true" processExpired="false" processNonPersistent="true" />

</deadLetterStrategy>

</policyEntry>

另：java代码中不设定消息过期时间

1，启动activemq服务。

2，利用sender类创建队列并分发送消息，但不使用consumer接收信息。

3，则消息永不过期，不会进入死信队列中

测试二：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" expireMessagesPeriod="1000" >

<deadLetterStrategy>

<individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true" processExpired="false" processNonPersistent="true" />

</deadLetterStrategy>

</policyEntry>

另：在java代码中设定消息过期时间。producer.setTimeToLive(6000);

1，启动activemq服务。

2，利用sender类创建队列并分发送消息，但不使用consumer接收信息。

3，则消息6秒后进入死信队列中

测试三：

Activemq.xml的关键配置信息如下：

<policyEntry queue="FirstQueue" expireMessagesPeriod="60000" >

<deadLetterStrategy>

<individualDeadLetterStrategy queuePrefix="DLQ." useQueueForQueueMessages="true" processExpired="false" processNonPersistent="true" />

</deadLetterStrategy>

</policyEntry>

另：在java代码中设定消息过期时间。producer.setTimeToLive(6000);

另：在java代码中设定消息过期时间。producer.setTimeToLive(10000);

1，启动activemq服务。

2，利用sender类创建队列并分发送消息，但不使用consumer接收信息。

3，则消息60秒后进入死信队列中

注意事项：expireMessagesPeriod的值应该小于代码中设定的消息过期时间。

**slowConsumerStrategy**：用于对处理信息缓慢的消费者进行处理。默认为null。

使用方式如下：

1. AbortSlowConsumerStrategy: 中断慢速消费者，慢速消费将会被关闭。

Activemq.xml关键配置如下

<policyEntry queue="test" …>

……

<slowConsumerStrategy>

<abortSlowConsumerStrategy abortConnection="false"/><!-- 此种方式不关闭底层链接 -->

</slowConsumerStrategy>

</policyEntry>

1. AbortSlowConsumerStrategy: 如果慢速消费者最后一个ACK距离现在的时间间隔超过阀值，则中断慢速消费者。

Activemq.xml关键配置如下：

<policyEntry queue="test" …>

……

<slowConsumerStrategy>

<abortSlowConsumerStrategy maxTimeSinceLastAck="30000"/><!-- 30秒滞后 -->

</slowConsumerStrategy>

</policyEntry>

**maxDestinations**：在一个目的地下建立虚拟目的地的最大数量。

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" maxDestinations="5" >

......

</policyEntry>

1，启动activemq服务，进入console页面。

2，在queue页面上建立新的队列。当建立第六个队列时，activemq报错，无法再新建队列。

测试二：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" >

......

</policyEntry>

1，启动activemq服务，进入console页面。

2，在queue页面上建立新的队列。当建立第六个队列时，activemq不会报错，可以新建任意多个队列。

**gcInactiveDestinations**：是回收空闲Destination，默认是false；

**inactiveTimoutBeforeGC**：表示被回收前Destination不活动时间。默认是5000,单位ms。

注意：如果要使用空闲Destination回收功能，不但要使用以上两项配置，还必须在broker中设置检查时间schedulePeriodForDestinationPurge="1000" ，单位ms。

这个功能只删除空队列，不删除有信息的队列。

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost" dataDirectory="${activemq.data}" schedulePeriodForDestinationPurge="1000">

<policyEntry queue=">" gcInactiveDestinations="true" inactiveTimoutBeforeGC="15000" >

......

</policyEntry>

</broker>

1, 启动activemq服务，进入console页面。

2，在queue页面上新建队列，但是不向队列发送任何信息。

3，到达配置的超时时间后，Destination被回收。

测试二：

Activemq.xml的关键配置信息如下：

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

<policyEntry queue=">" gcInactiveDestinations="true" inactiveTimoutBeforeGC="15000" >

......

</policyEntry>

</broker>

注：即不配置schedulePeriodForDestinationPurge监测时间。

1, 启动activemq服务，进入console页面。

2，在queue页面上新建队列，但是不向队列发送任何信息。

3，到达配置的超时时间后，Destination并未回收。

注意事项：使用空闲目的地回收功能时。如果在配置文件中配置的的目的地达到超时时间未被使用，也会被回收。但是当activemq服务再次启动时。配置文件中配置的队列仍会出现。

**advisoryForConsumed** ： 当客户端消费完信息后发送一条报告信息。

**advisoryForDelivery** ： 当信息发送给客户端后发送一条报告；

**advisoryForSlowConsumers** : 当一个消费者被认定缓慢之后，发一条公告。

**advsioryForFastProducers** : 当一个生产者被认定过快之后，发一条公告。

**advisoryWhenFull** : 当存储被占满之后发一条公告。

**includeBodyForAdvisory**: 保留原始报文的主体部分，最为报告信息的一部分。默认为false，发送报告时会将原报文的主题删除。

注：**以下配置用于监听activemq上的动态，大同小异，并未一一进行测试。测试案例围绕advisoryForConsumed和includeBodyForAdvisory属性进行。此功能是a**

**测试案例**

测试一：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" **advisoryForConsumed="true"** ......>

......

</policyEntry>

1，启动activemq服务。

1. 启动监听报告的程序，启动receiver，并调用sender发送报文
2. 当receiver正确接收报文后，报告监听程序接到消息被消费的报告；但报告中并未包含被消费报文的主题内容；

测试二：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" advisoryForConsumed="true" includeBodyForAdvisory="true" ......>

......

</policyEntry>

1，启动activemq服务。

1. 启动监听报告的程序，启动receiver，并调用sender发送报文
2. 当receiver正确接收报文后，报告监听程序接到消息被消费的报告；而且报告中包含被消费报文的主题内容；

测试三：

Activemq.xml的关键配置信息如下：

<policyEntry queue=">" advisoryForConsumed="false" includeBodyForAdvisory="true" ......>

......

</policyEntry>

1，启动activemq服务。

1. 启动监听报告的程序，启动receiver，并调用sender发送报文
2. 当receiver正确接收报文后，报告监听程序无法接收到消息被消费的报告。

Java代码如下：

package com.tgz.activemq;

import javax.jms.Connection;

import javax.jms.ConnectionFactory;

import javax.jms.Destination;

import javax.jms.Message;

import javax.jms.MessageConsumer;

import javax.jms.MessageListener;

import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;

import org.apache.activemq.advisory.AdvisorySupport;

import org.apache.activemq.command.ActiveMQMessage;

public class TestAdv {

public static void testAdvisory4() throws Exception {

ConnectionFactory factory = new ActiveMQConnectionFactory(

"tcp://localhost:61616");

Connection connection = factory.createConnection();

connection.start();

final Session session = connection.createSession(false/\* 支持事务 \*/,

Session.AUTO\_ACKNOWLEDGE);

Destination queue = AdvisorySupport

.getMessageConsumedAdvisoryTopic(session

.createQueue("FirstQueue"));

MessageConsumer consumer = session.createConsumer(queue);

consumer.setMessageListener(new MessageListener() {

@Override

public void onMessage(Message message) {

System.out.println(message);

ActiveMQMessage consumedMessage = (ActiveMQMessage) ((ActiveMQMessage) message)

.getDataStructure();

System.out.println("队列：[" + consumedMessage.getDestination()

+ "]，消息：[id="

+ consumedMessage.getMessageId() + "]被成功接收。");

}

});

}

public static void main(String[] args) {

try {

TestAdv.testAdvisory4();

} catch (Exception e) {

System.out.println("程序错误，请检查 ！！！");

}

}

}

**其他**

**consumersBeforeDispatchStarts**：第一个消费者连接后，等待到指定消费者都连接后派送信息。

**timeBeforeDispatchStarts** ：第一个消费者连接后，等待到指定时间后派送信息。

**lazyDispatch**:增加控制使用当前消费者列表的预取值；大数据传输使用

**optimizedDispatch**：调度优化，不使用独立线程从队列调度信息。默认情况下从某个queue中调度信息都是用单独的线程。

**strictOrderDispatch**：如果设定为true，消息将会一直派发给一个消费者，直到它的预取缓存器被占满。

**useConsumerPriority**：根据消费者的优先级派送信息

**prioritizedMessages**：根据消息的优先级进行存储

**storeUsageHighWaterMark**：信息存储允许使用的最大百分比。

**cursorMemoryHighWaterMark**：系统存储的临界点，超过会引起阻塞或者暂时将信息写到磁盘。

**minimumMessageSize**：对于非序列化信息，信息假设大小，用于存储计算。序列化信息使用序列化的大小作为存储计算的依据。

**maxBrowsePageSize**：增加控制使用当前消费者列表的预取值，用于大数据传输

**maxPageSize**：控制进入内存中的消息数量。用于大数据传输

**useCache**：缓存持久化的信息 以便能尽快从存储提取，根据处理机制不同，信息有可能写入临时文件，然后在内存中保存一个文件游标。默认为true。

**enableAudit**：；和另一个属性配合使用replayWhenNoConsumers，选项**replayWhenNoConsumers**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。

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。

**persistJMSRedelivered**属性：这个属性和JMSRedelivered属性配合使用。在**消息重发**时发挥作用。

## Activemq中jndi的应用

Activemq可以与JNDI提供者配合使用，而且其本身也提供了jndi功能，使得客户端可以在缺少jndi提供者的情况下享受jndi编程带来的好处。但是即使activemq提供了这样的能力，也不能把activemq的jndi功能当做jndi提供者来使用，毕竟它不是为了提供专业的jndi能力而存在的。

**测试案例**

**测试一：**

Activemq.xml的关键配置信息如下：

......

<destinations>

<queue physicalName="MyTestQueue" name ="test" />

</destinations>

......

Java代码如下：

package activemqJNDI;

import javax.jms.Connection;

import javax.jms.DeliveryMode;

import javax.jms.JMSException;

import javax.jms.MessageProducer;

import javax.jms.Session;

import javax.jms.TextMessage;

import javax.jms.TopicSession;

public class Sender {

public static void sendMessage(Session session, MessageProducer producer,

int i) {

try {

String words = "shine message NO." + i;

TextMessage message = session.createTextMessage(words);

System.out.println("发送的信息" + words);

producer.send(message);

} catch (JMSException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

public static void main(String[] args) {

// JMS客户端到JMSProvider的链接；

Connection connection = null;

// Session 一个发送或接收信息的线程

Session session;

// 消息发送者

MessageProducer producer;

try {

// create a new intial context, which loads from jndi.properties

// file

javax.naming.Context ctx = new javax.naming.InitialContext();

// lookup the connection factory

javax.jms.QueueConnectionFactory factory = (javax.jms.QueueConnectionFactory) ctx

.lookup("ConnectionFactory");

// create a new TopicConnection for pub/sub messaging

javax.jms.QueueConnection conn = factory.createQueueConnection();

// lookup an existing topic

javax.jms.Queue myQueue = (javax.jms.Queue) ctx.lookup("MyQueue");

// create a new TopicSession for the client

session = conn.createQueueSession(false,

TopicSession.AUTO\_ACKNOWLEDGE);

// 生成消息发送者

producer = session.createProducer(myQueue);

// 设置是否持久化??

producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);

conn.start();

// 发送消息

sendMessage(session, producer, 10000);

} catch (Exception e) {

e.printStackTrace();

} finally {

try {

if (null != connection) {

connection.close();

}

} catch (JMSException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

}

jndi.properties配置文件内容如下：

## ---------------------------------------------------------------------------

## Licensed to the Apache Software Foundation (ASF) under one or more

## contributor license agreements. See the NOTICE file distributed with

## this work for additional information regarding copyright ownership.

## The ASF licenses this file to You under the Apache License, Version 2.0

## (the "License"); you may not use this file except in compliance with

## the License. You may obtain a copy of the License at

##

## http://www.apache.org/licenses/LICENSE-2.0

##

## Unless required by applicable law or agreed to in writing, software

## distributed under the License is distributed on an "AS IS" BASIS,

## WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

## See the License for the specific language governing permissions and

## limitations under the License.

## ---------------------------------------------------------------------------

# START SNIPPET: jndi

java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory

# use the following property to configure the default connector

java.naming.provider.url = tcp://localhost:61616

# use the following property to specify the JNDI name the connection factory

# should appear as.

#connectionFactoryNames = connectionFactory, queueConnectionFactory, topicConnectionFactry

# register some queues in JNDI using the form

# queue.[jndiName] = [physicalName]

queue.MyQueue = MyTestQueue

# register some topics in JNDI using the form

# topic.[jndiName] = [physicalName]

topic.MyTopic = example.MyTopic

# END SNIPPET: jndi

1，启动activemq服务。

1. 调用sender发送报文
2. 登录activemq的console页面查看队列状态，信息已经正确发送到activemq服务；

## 注意内容：

1，activemq.Xml中关于所有队列公共设置项和单个队列的设置无法叠加使用。设置公共属性，或者设置每一个队列的属性只能选择一个。 所以决定设置完每个queue的属性，请把公共规则删除，否则会引起冲突。