

课程大纲

- 1、ActiveMQ简介
- 2、ActiveMQ安装
- 3、原生JMS API操作ActiveMQ
- 4、Spring与ActiveMQ整合
- 5、SpringBoot与ActiveMQ整合
- 6、ActiveMQ消息组成与高级特性
- 7、ActiveMQ企业面试经典问题总结

01、ActiveMQ入门

消息中间件应用场景

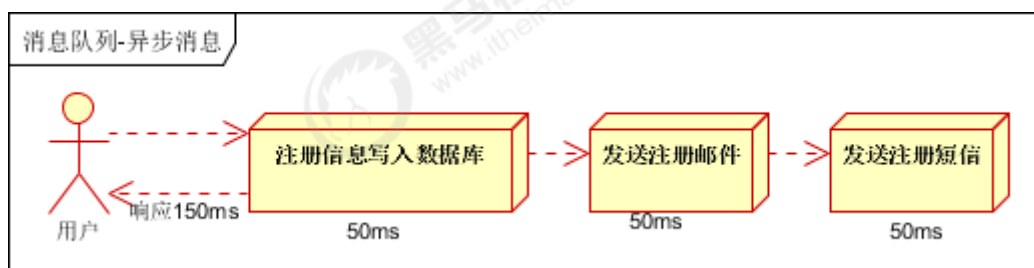
异步处理
应用解耦
流量削锋

异步处理

场景说明：用户注册，需要执行三个业务逻辑，分别为写入用户表，发注册邮件以及注册短信。

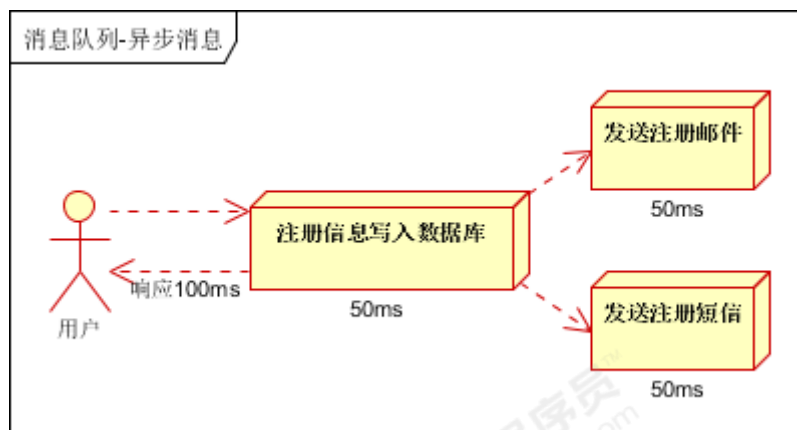
串行方式

将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。



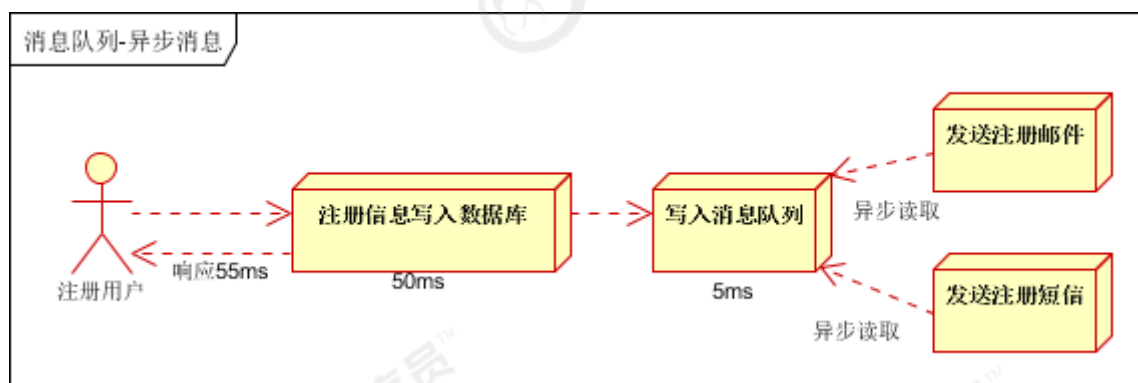
并行方式

将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间



异步处理

引入消息中间件，将部分的业务逻辑，进行异步处理。改造后的架构如下：

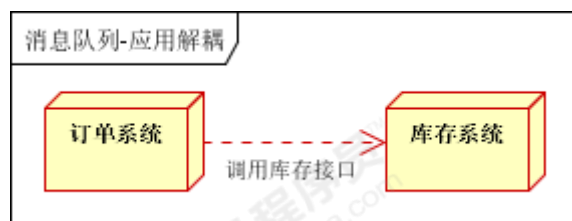


按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。因此架构改变后，系统的吞吐量提高啦，比串行提高了3倍，比并行提高了两倍。

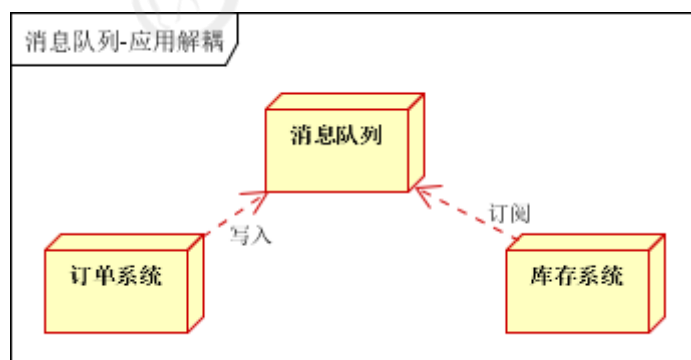
应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。

传统的做法是，订单系统调用库存系统的接口。如下图：



传统模式的缺点：假如库存系统无法访问，则订单减库存将失败，从而导致订单失败，订单系统与库存系统耦合。如何解决以上问题呢？引入应用消息队列后的方案，如下图：



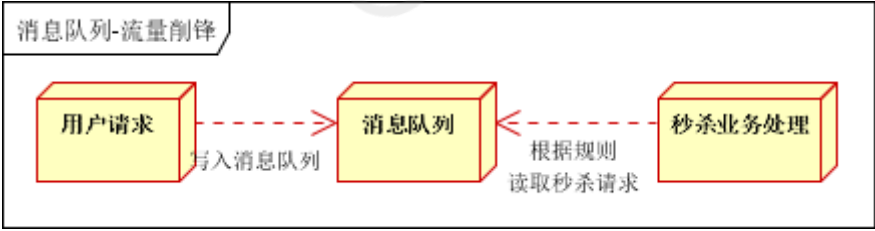
订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功 库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作 假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

流量消峰

流量削峰也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛。应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

通过加入消息队列完成如下功能：

- a、可以控制活动的人数
- b、可以缓解短时间内高流量压垮应用



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。秒杀业务根据消息队列中的请求信息，再做后续处理

常见的消息中间件产品对比

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
开发语言	Java	Erlang	Java	Scala
单击吞吐量	万级	万级	10万级	10万级
时效性	毫秒级	微秒级	毫秒级	毫秒级
可用性	高（支持主从架构）	高（支持主从架构）	非常高（分布式架构）	非常高（分布式架构）
功	成熟的产品，在很多	基于erlang开发，所以并	MQ功能比	像一些消息查 询、消息回溯等

能 特 性	公司得到应用；有较 多的文档；各种协议 支持较好	发能力很强，性能极其 好，延时很低；管理界面 较丰富	MQ功能比 较完备，扩 展性佳	同，消息回朔等 功能没有提供， 在大数据领域应 用广。
-------------	--------------------------------	----------------------------------	-----------------------	--------------------------------------

ActiveMQ简介及JMS



什么是ActiveMQ?

官网: <http://activemq.apache.org/>

ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ 是一个完全支持JMS1.1和J2EE 1.4规范的JMS Provider实现。我们在本次课程中介绍 ActiveMQ的使用。

什么是JMS?

消息中间件利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。它可以在分布式环境下扩展进程间的通信。对于消息中间件，常见的角色大致也就有 Producer（生产者）、Consumer（消费者）。

消息队列中间件是分布式系统中重要的组件，主要解决应用解耦，异步消息，流量削锋等问题，实现高性能，高可用，可伸缩和最终一致性架构。

JMS (Java Messaging Service) 是Java平台上有关面向消息中间件的技术规范，它便于消息系统中的Java应用程序进行消息交换,并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发。

JMS本身只定义了一系列的接口规范，是一种与厂商无关的 API，用来访问消息收发系统。它类似于 JDBC(Java Database Connectivity): 这里，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。许多厂商目前都支持 JMS，包括 IBM 的 MQSeries、BEA 的 Weblogic JMS service和 Progress 的 SonicMQ，这只是几个例子。JMS 使您能够通过消息收发服务（有时称为消息中介程序或路由器）从一个 JMS 客户机向另一个 JMS 客户机发送消息。消息是 JMS 中的一种类型对象，由两部分组成：报头和消息主体。报头由路由信息以及有关该消息的元数据组成。消息主体则携带着应用程序的数据或有效负载。

JMS消息模型

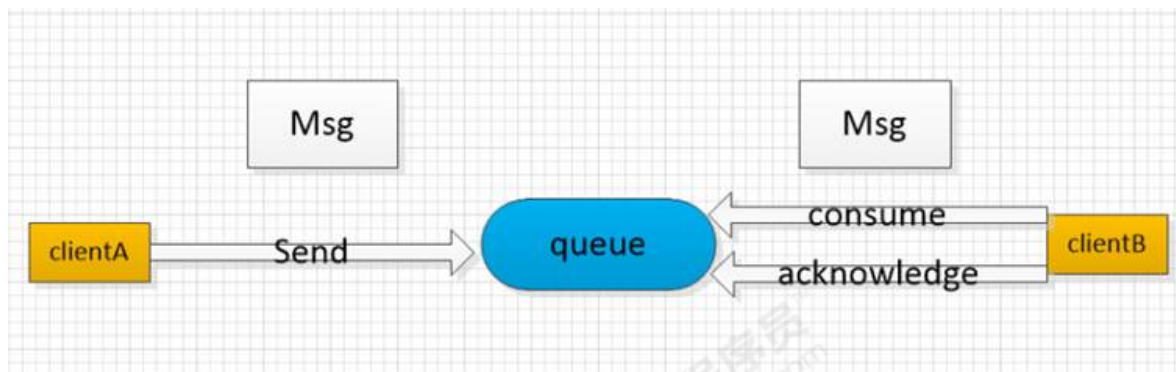
消息中间件一般有两种传递模式：点对点模式(P2P)和发布-订阅模式(Pub/Sub)。

(1) P2P (Point to Point) 点对点模型 (Queue队列模型)

(2) Publish/Subscribe(Pub/Sub) 发布/订阅模型(Topic主题模型)

点对点模型

点对点模型 (Pointer-to-Pointer)：即生产者和消费者之间的消息往来。

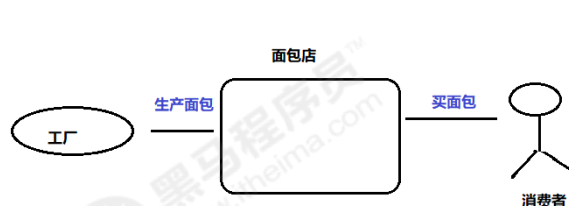


每个消息都被发送到特定的消息队列，接收者从队列中获取消息。队列保留着消息，直到他们被消费或超时。

点对点模型的特点：

- 每个消息只有一个消费者 (Consumer) (即一旦被消费，消息就不再在消息队列中)；
- 发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行，它不会影响到消息被发送到队列；
- 接收者在成功接收消息之后需向队列应答成功。

举例



点对点:

1. 生产的面包放入面包店，不会过期，直到被人买
2. 一个面包，只能卖给一个人。
3. 面包店的某一个面包卖掉后，就没了。

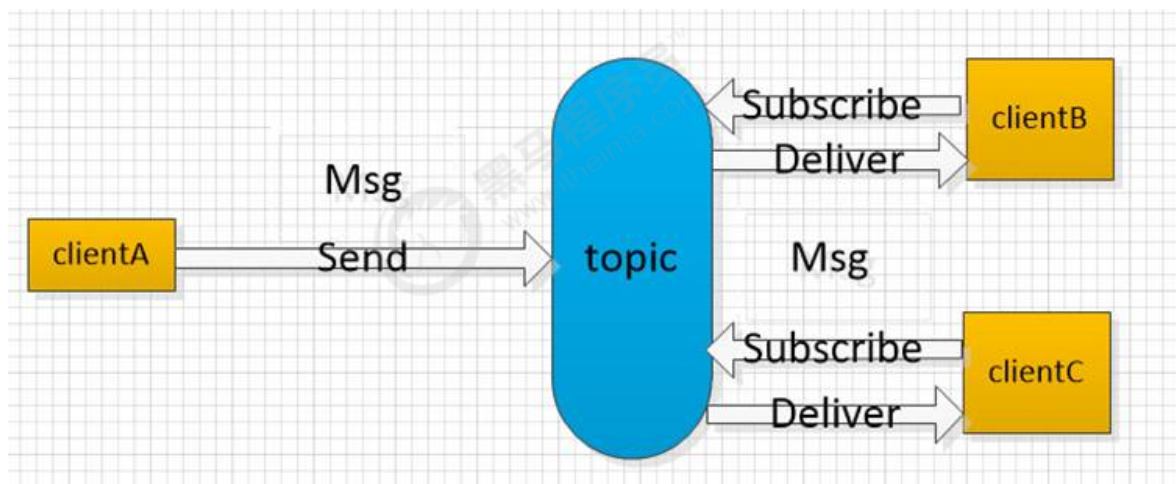
发布订阅模型:

1. 生产的一个面包放入面包店，此时必须有客户等待购买，否则就过期作废。
2. 一个面包，可以同时卖给多个人。
3. 面包店的某一个面包卖掉后，就没了。

发布/订阅模型

发布/订阅 (Publish-Subscribe)

包含三个角色：主题 (Topic)，发布者 (Publisher)，订阅者 (Subscriber)，多个发布者将消息发送到topic，系统将这些消息投递到订阅此topic的订阅者



发布者发送到topic的消息，只有订阅了topic的订阅者才会收到消息。topic实现了发布和订阅，当你发布一个消息，所有订阅这个topic的服务都能得到这个消息，所以从1到N个订阅者都能得到这个消息的拷贝。

发布/订阅模型的特点：

- 每个消息可以有多个消费者；
- 发布者和订阅者之间有时间上的依赖性（先订阅主题，再来发送消息）。
- 订阅者必须保持运行的状态，才能接受发布者发布的消息；

JMS编程API

要素	作用
Destination	表示消息所走通道的目标定义，用来定义消息从发送端发出后要走的通道，而不是接收方。Destination属于管理类对象
ConnectionFactory	顾名思义，用于创建连接对象，ConnectionFactory属于管理类的对象
Connection	连接接口，所负责的重要工作时创建Session
Session	会话接口，这是一个非常重要的对象，消息发送者、消息接收者以及消息对象本身，都是通过这个会话对象创建的
MessageConsume	消息的消费者，也就是订阅消息并处理消息的对象
MessageProducer	消息的生产者，也就是用来发送消息的对象

(1) ConnectionFactory

创建Connection对象的工厂，针对两种不同的jms消息模型，分别有QueueConnectionFactory和TopicConnectionFactory两种。

(2) Destination

Destination的意思是消息生产者的消息发送目标或者说消息消费者的消息来源。对于消息生产者来说，它的Destination是某个队列（Queue）或某个主题（Topic）；对于消息消费者来说，它的Destination也是某个队列或主题（即消息来源）。所以，Destination实际上就是两种类型的对象：Queue、Topic

(3) Connection

Connection表示在客户端和JMS系统之间建立的链接（对TCP/IP socket的包装）。Connection可以产生一个或多个Session

(4) Session

Session 是我们对消息进行操作的接口，可以通过session创建生产者、消费者、消息等。Session 提供了事务的功能，如果需要使用session发送/接收多个消息时，可以将这些发送/接收动作放到一个事务中。

(5) Producter

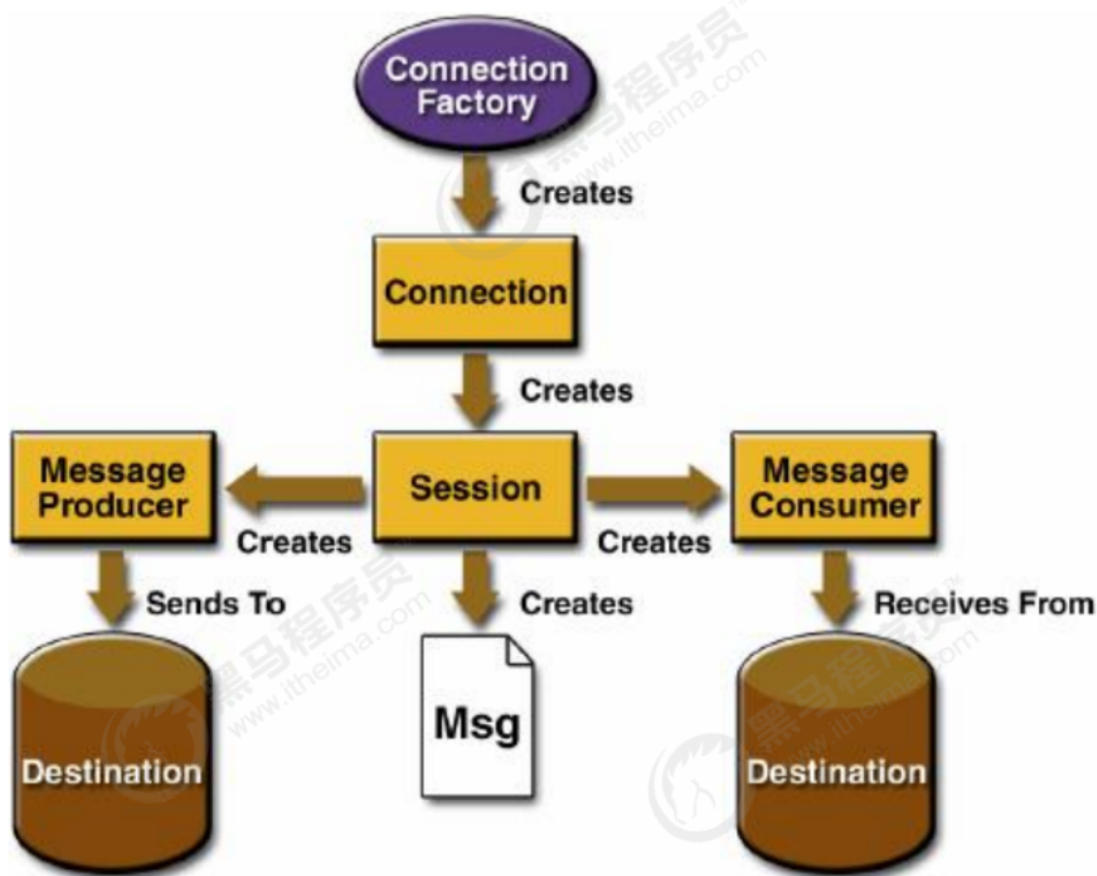
Producter（消息生产者）：消息生产者由Session创建，并用于将消息发送到Destination。同样，消息生产者分两种类型：QueueSender和TopicPublisher。可以调用消息生产者的方法（send或publish方法）发送消息。

(6) Consumer

Consumer（消息消费者）：消息消费者由Session创建，用于接收被发送到Destination的消息。两种类型：QueueReceiver和TopicSubscriber。可分别通过session的createReceiver(Queue)或createSubscriber(Topic)来创建。当然，也可以session的createDurableSubscriber方法来创建持久化的订阅者。

(7) MessageListener

消息监听器。如果注册了消息监听器，一旦消息到达，将自动调用监听器的onMessage方法。EJB中的MDB（Message-Driven Bean）就是一种MessageListener。



02、ActiveMQ的安装

安装

第一步：安装 jdk（略）

第二步：把 activemq的压缩包（apache-activemq-5.14.5-bin.tar.gz）上传到 linux 系统

第三步：解压缩压缩包

```
tar -zxvf apache-activemq-5.14.5-bin.tar.gz
```

第四步：进入apache-activemq-5.14.5的bin目录

```
cd apache-activemq-5.14.5/bin
```

第五步：启动 activemq

```
./activemq start （执行2次：第一次：生成配置信息；第二次：启动）
```

第六步：停止activemq:

```
./activemq stop
```

访问

<http://192.168.12.132:8161>

页面控制台: <http://ip:8161> (监控)

请求地址: <tcp://ip:61616> (java代码访问消息中间件)

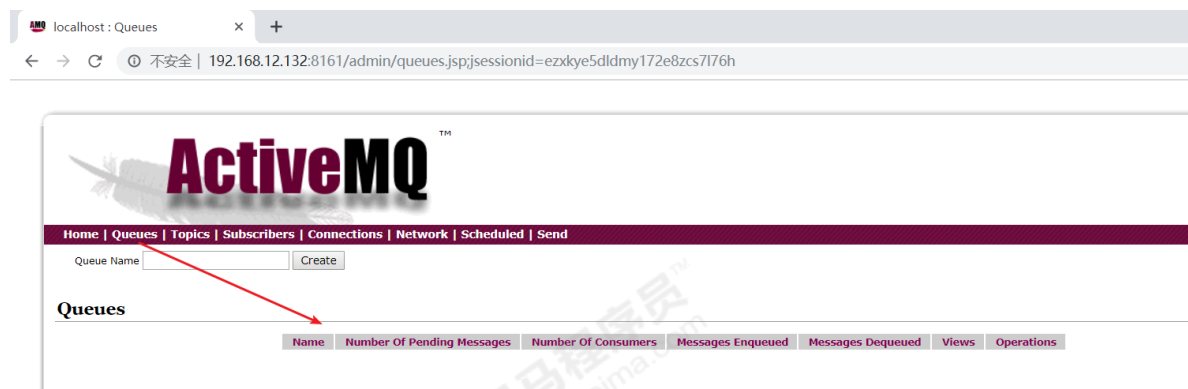
账号: admin

密码: admin

图1: 登陆:



图2: 点击Queues队列或者Topics主题消息



列表各列信息含义如下:

Number Of Pending Messages : 等待消费的消息 这个是当前未出队列的数量。

Number Of Consumers : 消费者 这个是消费者端的消费者数量

Messages Enqueued : 进入队列的消息 进入队列的总数量,包括出队列的。

Messages Dequeued : 出了队列的消息 可以理解为是消费这消费掉的数量。

03、原生JMS API操作ActiveMQ

PTP模式(生产者)

(1) 引入坐标

```
<dependencies>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-all</artifactId>
    <version>5.11.2</version>
  </dependency>
</dependencies>
```

(2) 编写生产消息的测试类 QueueProducer

步骤:

1. 创建连接工厂
2. 创建连接
3. 打开连接
4. 创建session
5. 创建目标地址 (Queue: 点对点消息, Topic: 发布订阅消息)
6. 创建消息生产者
7. 创建消息
8. 发送消息
9. 释放资源

```
package com.itheima.producer;

import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.*;

/**
 * 演示点对点模式 -- 消息生产者
 */
public class PTP_Producer {

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

        //1. 创建连接工厂
        ConnectionFactory factory
            = new ActiveMQConnectionFactory("tcp://192.168.66.133:61616");

        //2. 创建连接
        Connection connection = factory.createConnection();

        //3. 打开连接
        connection.start();

        //4. 创建session
        /**
         * 参数一: 是否开启事务操作
         */
    }
}
```

```

    * 参数二：消息确认机制
    */
    Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

    //5.创建目标地址（Queue:点对点消息，Topic：发布订阅消息）
    Queue queue = session.createQueue("queue01");

    //6.创建消息生产者
    MessageProducer producer = session.createProducer(queue);

    //7.创建消息
    //createTextMessage：文本类型
    TextMessage textMessage = session.createTextMessage("test message");

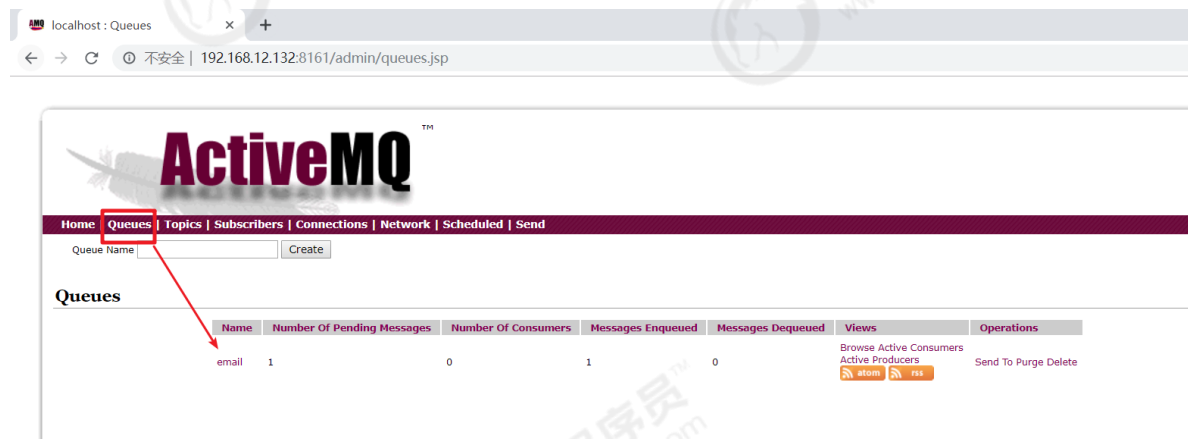
    //8.发送消息
    producer.send(textMessage);

    System.out.println("消息发送完成");

    //9.释放资源
    session.close();
    connection.close();
}
}

```

观察发送消息的结果：



PTP模式(消费者)

步骤：

1. 创建连接工厂
2. 创建连接
3. 打开连接
4. 创建session
5. 指定目标地址
6. 创建消息的消费者
7. 配置消息监听器

第一种消费者写法:

```
package com.itheima.consumer;

import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.*;

/**
 * 演示点对点模式- 消息消费者（第一种方案）
 */
public class PTP_Consumer1 {

    public static void main(String[] args) throws JMSEException {
        //1.创建连接工厂
        ConnectionFactory factory
            = new ActiveMQConnectionFactory("tcp://192.168.66.133:61616");

        //2.创建连接
        Connection connection = factory.createConnection();

        //3.打开连接
        connection.start();

        //4.创建session
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);

        //5.指定目标地址
        Queue queue = session.createQueue("queue01");

        //6.创建消息的消费者
        MessageConsumer consumer = session.createConsumer(queue);

        //7.接收消息
        while(true){
            Message message = consumer.receive();

            //如果已经没有消息了，结束啦
            if(message==null){
                break;
            }

            //如果还有消息，判断什么类型的消息
            if(message instanceof TextMessage){
                TextMessage textMessage = (TextMessage)message;

                System.out.println("接收的消息: "+textMessage.getText());
            }
        }
    }
}
```

第二种消费者写法（推荐）：

```
package com.itheima.consumer;

import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.*;

/**
 * 演示点对点模式- 消息消费者（第二种方案） -- 更加推荐
 */
public class PTP_Consumer2 {

    public static void main(String[] args) throws JMSEException {
        //1.创建连接工厂
        ConnectionFactory factory
            = new ActiveMQConnectionFactory("tcp://192.168.66.133:61616");

        //2.创建连接
        Connection connection = factory.createConnection();

        //3.打开连接
        connection.start();

        //4.创建session
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);

        //5.指定目标地址
        Queue queue = session.createQueue("queue01");

        //6.创建消息的消费者
        MessageConsumer consumer = session.createConsumer(queue);

        //7.设置消息监听器来接收消息
        consumer.setMessageListener(new MessageListener() {
            //处理消息
            @Override
            public void onMessage(Message message) {
                if(message instanceof TextMessage){
                    TextMessage textMessage = (TextMessage)message;

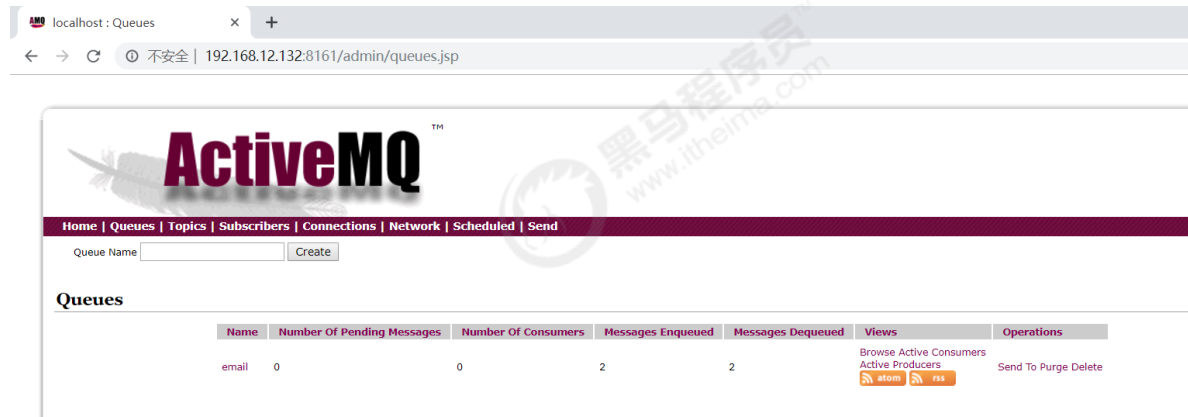
                    try {
                        System.out.println("接收的消息
(2) : "+textMessage.getText());
                    } catch (JMSEException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
    }
}
```

//注意：在监听器的模式下千万不要关闭连接，一旦关闭，消息无法接收

```
}
```

```
}
```

观察消费消息的结果：



Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
email	0	0	2	2		Browse Active Consumers Active Producers Send To Purge Delete

Pub/Sub模式(生成者)

1. 创建连接工厂
2. 创建连接
3. 打开连接
4. 创建session
5. 创建目标地址（Queue:点对点消息，Topic: 发布订阅消息）
6. 创建消息生产者
7. 创建消息
8. 发送消息
9. 释放资源

```
package cn.itcast.activemq;

import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.*;

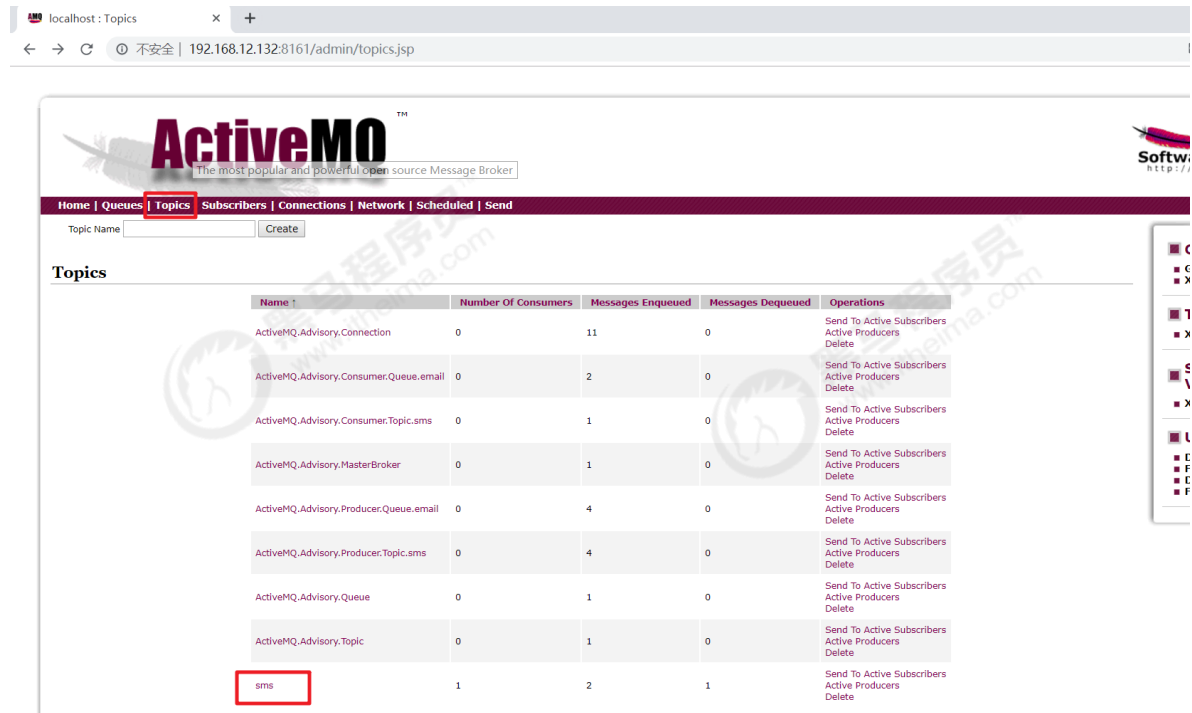
/**
 * 主题消息，消息的发送方
 */
public class TopicProducer {
    public static void main(String[] args) throws Exception {
        //1. 创建连接工厂
        ConnectionFactory factory = new
ActiveMQConnectionFactory("tcp://192.168.12.132:61616");
        //2. 创建连接
        Connection connection = factory.createConnection();
        //3. 打开连接
```

```

connection.start();
//4.创建session
Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
//5.创建目标地址（Queue:点对点消息，Topic:发布订阅消息）
Topic topic = session.createTopic("sms");
//6.创建消息生产者
MessageProducer producer = session.createProducer(topic);
//7.创建消息
TextMessage message = session.createTextMessage("发短信...");
//8.发送消息
producer.send(message);
System.out.println("发送消息: 发短信...");
session.close();
connection.close();
}
}

```

查看主题消息：



The screenshot shows the ActiveMQ web console interface. The 'Topics' tab is selected in the top navigation bar. Below the navigation bar, there is a search field for 'Topic Name' and a 'Create' button. The main content area displays a table of topics. The 'sms' topic is highlighted with a red box. The table has the following columns: Name, Number Of Consumers, Messages Enqueued, Messages Dequeued, and Operations.

Name	Number Of Consumers	Messages Enqueued	Messages Dequeued	Operations
ActiveMQ.Advisory.Connection	0	11	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Consumer.Queue.email	0	2	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Consumer.Topic.sms	0	1	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.MasterBroker	0	1	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Producer.Queue.email	0	4	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Producer.Topic.sms	0	4	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Queue	0	1	0	Send To Active Subscribers Active Producers Delete
ActiveMQ.Advisory.Topic	0	1	0	Send To Active Subscribers Active Producers Delete
sms	1	2	1	Send To Active Subscribers Active Producers Delete

Pub/Sub模式(消费者)

1. 创建连接工厂
2. 创建连接
3. 打开连接
4. 创建session
5. 指定目标地址
6. 创建消息的消费者
7. 配置消息监听器


```

package cn.itcast.activemq;

import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.*;

/**
 * 主题消息，消息的消费方
 */
public class TopicConsumer {
    public static void main(String[] args) throws Exception {
        //1.创建连接工厂
        ConnectionFactory factory = new
ActiveMQConnectionFactory("tcp://192.168.12.132:61616");
        //2.创建连接
        Connection connection = factory.createConnection();
        //3.打开连接
        connection.start();
        //4.创建session
        Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
        //5.创建目标地址（Queue:点对点消息，Topic:发布订阅消息）
        Topic topic = session.createTopic("sms");
        //6.创建消息的消费者
        MessageConsumer consumer = session.createConsumer(topic);
        //7.配置消息监听器
        consumer.setMessageListener(new MessageListener() {
            @Override
            public void onMessage(Message message) {
                TextMessage textMessage = (TextMessage) message;
                try {
                    System.out.println("消费消息: " + textMessage.getText());
                } catch (JMSEException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}

```

04、Spring与ActiveMQ整合

消息生产者

```

<dependencies>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-all</artifactId>
        <version>5.11.2</version>
    </dependency>

```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jms</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>

<dependency>
```

```

        <groupId>javax.jms</groupId>
        <artifactId>javax.jms-api</artifactId>
        <version>2.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.apache.xbean</groupId>
        <artifactId>xbean-spring</artifactId>
        <version>3.7</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>

```

2. 编写Spring整合ActiveMQ配置： applicationContext-producer.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:amp="http://activemq.apache.org/schema/core"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://activemq.apache.org/schema/core
http://activemq.apache.org/schema/core/activemq-core.xsd">

    <!--1. 创建连接工厂对象-->
    <amp:connectionFactory
        id="connetionFactory"
        brokerURL="tcp://192.168.66.133:61616"
        userName="admin"
        password="admin"
    />

    <!--2. 创建缓存连接工厂-->
    <bean id="cachingConnectionFactory"
        class="org.springframework.jms.connection.CachingConnectionFactory">
        <!--注入连接工厂-->
        <property name="targetConnectionFactory" ref="connetionFactory"/>
        <!--缓存消息数据-->
        <property name="sessionCacheSize" value="5"/>
    </bean>

    <!--3. 创建用于点对点发送的JmsTemplate-->
    <bean id="jmsQueueTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <!--注入缓存连接工厂-->
        <property name="connectionFactory" ref="cachingConnectionFactory"/>
        <!--指定是否为发布订阅模式-->
        <property name="pubSubDomain" value="false"/>
    </bean>

    <!--4. 创建用于发布订阅发送的JmsTemplate-->
    <bean id="jmsTopicTemplate"
        class="org.springframework.jms.core.JmsTemplate">

```

```

        <!--注入缓存连接工厂-->
        <property name="connectionFactory" ref="cachingConnectionFactory"/>
        <!--指定是否为发布订阅模式-->
        <property name="pubSubDomain" value="true"/>
    </bean>
</beans>

```

3. 编写测试类，实现发送消息

```

package com.itheima.producer;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;

/**
 * 演示Spring与ActiveMQ整合
 */
@RunWith(SpringJUnit4ClassRunner.class) // junit与spring整合
@ContextConfiguration("classpath:applicationContext-producer.xml") // 加载spring
配置文件
public class SpringProducer {

    //点对点模式
    @Autowired
    @Qualifier("jmsQueueTemplate")
    private JmsTemplate jmsQueueTemplate;

    //发布订阅模式
    @Autowired
    @Qualifier("jmsTopicTemplate")
    private JmsTemplate jmsTopicTemplate;

    /**
     * 点对点发送
     */
    @Test
    public void ptpSender(){
        /**
         * 参数一：指定队列的名称
         * 参数二： MessageCreator接口，我们需要提供该接口的匿名内部实现
         */
        jmsQueueTemplate.send("spring_queue", new MessageCreator() {

```

```

//我们只需要返回发送的消息内容即可
@Override
public Message createMessage(Session session) throws JMSException {
    //创建文本消息
    TextMessage textMessage = session.createTextMessage("spring test
message");
    return textMessage;
}
});
System.out.println("消息发送已完成");
}

/**
 * 发布订阅发送
 */
@Test
public void psSender(){
    jmsTopicTemplate.send("spring_topic", new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSException {
            //创建文本消息
            TextMessage textMessage = session.createTextMessage("spring test
message--topic");
            return textMessage;
        }
    });
    System.out.println("消息发送已完成");
}
}

```

消息消费者

1. 编写监听器：监听主题消息、队列消息

```

@Component
public class EmailMessageListener implements MessageListener {
    @Override
    public void onMessage(Message message) {
        MapMessage mapMessage = (MapMessage) message;
        try {
            String email = mapMessage.getString("email");
            System.out.println("消费消息: " + email);
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}

```

2. 编写Spring整合ActiveMQ配置：applicationContext-consumer.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:amq="http://activemq.apache.org/schema/core"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx.xsd
           http://www.springframework.org/schema/jms
           http://www.springframework.org/schema/jms/spring-jms.xsd
           http://activemq.apache.org/schema/core
           http://activemq.apache.org/schema/core/activemq-core.xsd">

    <!-- 1. 创建ActiveMQ连接工厂 -->
    <amq:connectionFactory
        id="amqConnectionFactory"
        userName="admin" password="admin"
        brokerURL="tcp://192.168.12.132:61616"/>

    <!-- 2. 创建缓存工厂 -->
    <bean id="cachingConnectionFactory"
        class="org.springframework.jms.connection.CachingConnectionFactory">
        <!-- 注入 连接工厂-->
        <property name="targetConnectionFactory" ref="amqConnectionFactory">
    </property>
        <!-- session缓存数目 -->
        <property name="sessionCacheSize" value="5"></property>
    </bean>

    <!--开启注解扫描-->
    <context:component-scan base-
package="cn.itcast.spring_activemq_consumer"/>
    <!--
        配置消息监听器类，监听队列或主题消息模型中的消息。从而实现消费消息。
        jms:listener-container
            destination-type 监听的JMS消息类型（queue、topic）
            connection-factory Spring的缓存连接工厂
        jms:listener
            destination 对应MQ中队列名称或主题名称
            ref 消息监听器类（实现MessageListener接口）
    -->

    <!-- 3.1 监听指定名称(email)的队列中的消息-->
    <jms:listener-container destination-type="queue" connection-
factory="cachingConnectionFactory">
        <jms:listener destination="email" ref="emailMessageListener"/>
    </jms:listener-container>

```



```

    <!-- 3.2 监听指定名称(email)的主题中的消息 -->
    <jms:listener-container destination-type="topic" connection-
factory="cachingConnectionFactory">
        <jms:listener destination="sms" ref="smsMessageListener"/>
    </jms:listener-container>

</beans>

```

3. 编写测试类，实现发送消息

```

/**
 * Spring整合ActiveMQ消费消息
 */
public class Consumer {
    public static void main(String[] args) throws IOException {
        ApplicationContext ac =
            new ClassPathXmlApplicationContext("applicationContext-
activemq-consumer.xml");
        System.in.read();
    }
}

```

05、SpringBoot与ActiveMQ整合

消息生产者

```

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.1.RELEASE</version>
    <relativePath/>
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>

```

```

        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-activemq</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

```

配置:

```

server:
  port: 9001 #端口
spring:
  application:
    name: activemq-producer # 服务名称

# springboot与activemq整合配置
activemq:
  broker-url: tcp://192.168.66.133:61616 # 连接地址
  user: admin # activemq用户名
  password: admin # activemq密码

# 指定发送模式 （点对点 false , 发布订阅 true）
jms:
  pub-sub-domain: false

```

编写启动类

```

package com.itheima.producer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * 生产者启动类
 */
@SpringBootApplication
public class ProducerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class,args);
    }

}

```

编写生产者

```
package com.itheima.producer;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * 演示SpringBoot与ActiveMQ整合- 消息生产者
 */
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = ProducerApplication.class)
public class SpringBootProducer {

    //JmsMessagingTemplate: 用于工具类发送消息
    @Autowired
    private JmsMessagingTemplate jmsMessagingTemplate;

    @Test
    public void ptpSender(){
        /**
         * 参数一: 队列的名称或主题名称
         * 参数二: 消息内容
         */
        jmsMessagingTemplate.convertAndSend("springboot_queue","spring boot message");
    }

}
```

消息消费者

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.1.RELEASE</version>
  <relativePath/>
</parent>
```

```

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

配置:

```

server:
  port: 9002 #端口
spring:
  application:
    name: activemq-consumer # 服务名称

# springboot与activemq整合配置
activemq:
  broker-url: tcp://192.168.66.133:61616 # 连接地址
  user: admin # activemq用户名
  password: admin # activemq密码

# 指定发送模式 （点对点 false , 发布订阅 true）
jms:
  pub-sub-domain: false

activemq:
  name: springboot_queue

```

编写启动类

```
package com.itheima.consumer;
```

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * 消息消费者启动类
 */
@SpringBootApplication
public class ConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class,args);
    }

}

```

```

package com.itheima.consumer.listener;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.TextMessage;

/**
 * 用于监听消息类（既可以用于队列的监听，也可以用于主题监听）
 */
@Component // 放入IOC容器
public class MsgListener {

    /**
     * 用于接收消息的方法
     * destination: 队列的名称或主题的名称
     */
    @JmsListener(destination = "${activemq.name}")
    public void receiveMessage(Message message){
        if(message instanceof TextMessage){
            TextMessage textMessage = (TextMessage)message;
            try {
                System.out.println("接收消息: "+textMessage.getText());
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }

}

```

06、ActiveMQ消息组成与高级特性

JMS消息组成详解

JMS消息组成格式

整个JMS协议组成结构如下：

结构	描述
JMS Provider	消息中间件/消息服务器
JMS Producer	消息生产者
JMS Consumer	消息消费者
JMS Message	消息（重要）

JMS Message消息由三部分组成：

- 1) 消息头
- 2) 消息体
- 3) 消息属性

JMS消息头

JMS消息头预定义了若干字段用于客户端与JMS提供者之间识别和发送消息，预编译头如下：

- **红色**为重要的消息头

名称	描述
JMSDestination	消息发送的 Destination，在发送过程中由提供者设置
JMSMessageID	唯一标识提供者发送的每一条消息。这个字段是在发送过程中由提供者设置的，客户机只能在消息发送后才能确定消息的 JMSMessageID
JMSDeliveryMode	消息持久化。包含值 DeliveryMode.PERSISTENT 或者 DeliveryMode.NON_PERSISTENT。
JMSTimestamp	提供者发送消息的时间，由提供者在发送过程中设置
JMSExpiration	消息失效的时间，毫秒，值 0 表明消息不会过期，默认值为0
	消息的优先级，由提供者在发送过程中设置。优先级 0 的优先级最低，优先级 9 的优先级最高。0-4为普通消息，5-9为加急消息。ActiveMQ不保证

JMSPriority 名称	描述 优先级高就一定先发送，只保证了加急消息必须先于普通消息发送。默认值为4
JMSCorrelationID	通常用来链接响应消息与请求消息，由发送消息的 JMS 程序设置。
JMSReplyTo	请求程序用它来指出回复消息应发送的地方，由发送消息的 JMS 程序设置
JMSType	JMS 程序用它来指出消息的类型。
JMSRedelivered	消息的重发标志，false，代表该消息是第一次发生，true，代表该消息为重发消息

不过需要注意的是，在传送消息时，消息头的值由JMS提供者来设置，**因此开发者使用以上 setJMSXXX()方法分配的值就被忽略了**，只有以下几个值是可以由开发者设置的：

JMSCorrelationID, JMSReplyTo, JMSType

JMS消息体

在消息体中，JMS API定义了五种类型的消息格式，让我们可以以不同的形式发送和接受消息，并提供了对已有消息格式的兼容。不同的消息类型如下：

JMS 定义了五种不同的消息正文格式，以及调用的消息类型，允许你发送并接收一些不同形式的数据，提供现有消息格式的一些级别的兼容性。

- TextMessage--一个字符串对象 *
- MapMessage--一套名称-值对
- ObjectMessage--一个序列化的 Java 对象 *
- BytesMessage--一个字节的数据流 *
- StreamMessage -- Java原始值的数据流

TextMessage:

写出：

```
/**
 * 发送TextMessage消息
 */
@Test
public void testMessage(){

    jmsTemplate.send(name, new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSException {
            TextMessage textMessage = session.createTextMessage("文本消息");

            return textMessage;
        }
    });
}
```

读取:

```
/**
 * 接收TextMessage的方法
 */
@JmsListener(destination = "${activemq.name}")
public void receiveMessage(Message message){
    if(message instanceof TextMessage){
        TextMessage textMessage = (TextMessage)message;

        try {
            System.out.println("接收消息: "+textMessage.getText());
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}
```

MapMessage:

发送:

```
/**
 * 发送MapMessage消息
 */
@Test
public void mapMessage(){

    jmsTemplate.send(name, new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSEException {
            MapMessage mapMessage = session.createMapMessage();
            mapMessage.setString("name", "张三");
            mapMessage.setInt("age", 20);

            return mapMessage;
        }
    });
}
```

接收:

```
@JmsListener(destination = "${activemq.name}")
public void receiveMessage(Message message){
    if(message instanceof MapMessage){
        MapMessage mapMessage = (MapMessage)message;

        try {
```

```

        System.out.println("名称: "+mapMessage.getString("name"));
        System.out.println("年龄: "+mapMessage.getString("age"));
    } catch (JMSException e) {
        e.printStackTrace();
    }
}
}
}

```

ObjectMessage:

```

//发送ObjectMessage消息
@Test
public void test2(){
    jmsTemplate.send(name, new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSException {
            User user = new User();
            user.setName("小苍");
            user.setAge(18);

            ObjectMessage objectMessage = session.createObjectMessage(user);
            return objectMessage;
        }
    });
}
}

```

接收:

```

@JmsListener(destination = "${activemq.name}")
public void receiveMessage(Message message){
    if(message instanceof ObjectMessage){
        ObjectMessage objectMessage = (ObjectMessage)message;

        try {
            User user = (User)objectMessage.getObject();
            System.out.println(user.getUsername());
            System.out.println(user.getPassword());
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

注意：ActiveMQ5.12后，为了安全考虑，ActiveMQ默认不接受自定义的序列化对象，需要将自定义的加入到受信任的列表。

```
spring:
  activemq:
    broker-url: tcp://192.168.66.133:61616
    user: admin
    password: admin
    packages:
      trust-all: true # 添加所有包到信任列表
```

BytesMessage:

写出:

```
//发送BytesMessage消息
@Test
public void test3(){
    jmsTemplate.send(name, new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSException {

            BytesMessage bytesMessage = session.createBytesMessage();

            try {
                File file = new File("d:/spring.jpg");
                FileInputStream in = new FileInputStream(file);
                byte[] bytes = new byte[(int)file.length()];
                in.read(bytes);

                bytesMessage.writeBytes(bytes);
            } catch (Exception e) {
                e.printStackTrace();
            }

            return bytesMessage;
        }
    });
}
```

读取:

```
@JmsListener(destination="${activemq.name}")
public void receiveMessage(Message message) throws Exception {
    BytesMessage bytesMessage = (BytesMessage)message;

    FileOutputStream out = new FileOutputStream("d:/abc.jpg");
    byte[] buf = new byte[(int)bytesMessage.getBodyLength()];
    bytesMessage.readBytes(buf);

    out.write(buf);
    out.close();
}
```

StreamMessage:

写出:

```
//发送StreamMessage消息
@Test
public void test4(){
    jmsTemplate.send(name, new MessageCreator() {
        @Override
        public Message createMessage(Session session) throws JMSException {
            StreamMessage streamMessage = session.createStreamMessage();
            streamMessage.writeString("你好, ActiveMQ");
            streamMessage.writeInt(20);

            return streamMessage;
        }
    });
}
```

读取:

```
@JmsListener(destination="${activemq.name}")
public void receiveMessage(Message message) throws Exception {
    StreamMessage streamMessage = (StreamMessage)message;
    String str = streamMessage.readString();
    int i = streamMessage.readInt();
    System.out.println(str);
    System.out.println(i);
}
```

JMS消息属性

我们可以给消息设置自定义属性，这些属性主要是提供给应用程序的。对于实现消息过滤功能，消息属性非常有用，JMS API定义了一些标准属性，JMS服务提供者可以选择性的提供部分标准属性。

```
message.setStringProperty("Property", Property);    //自定义属性
```

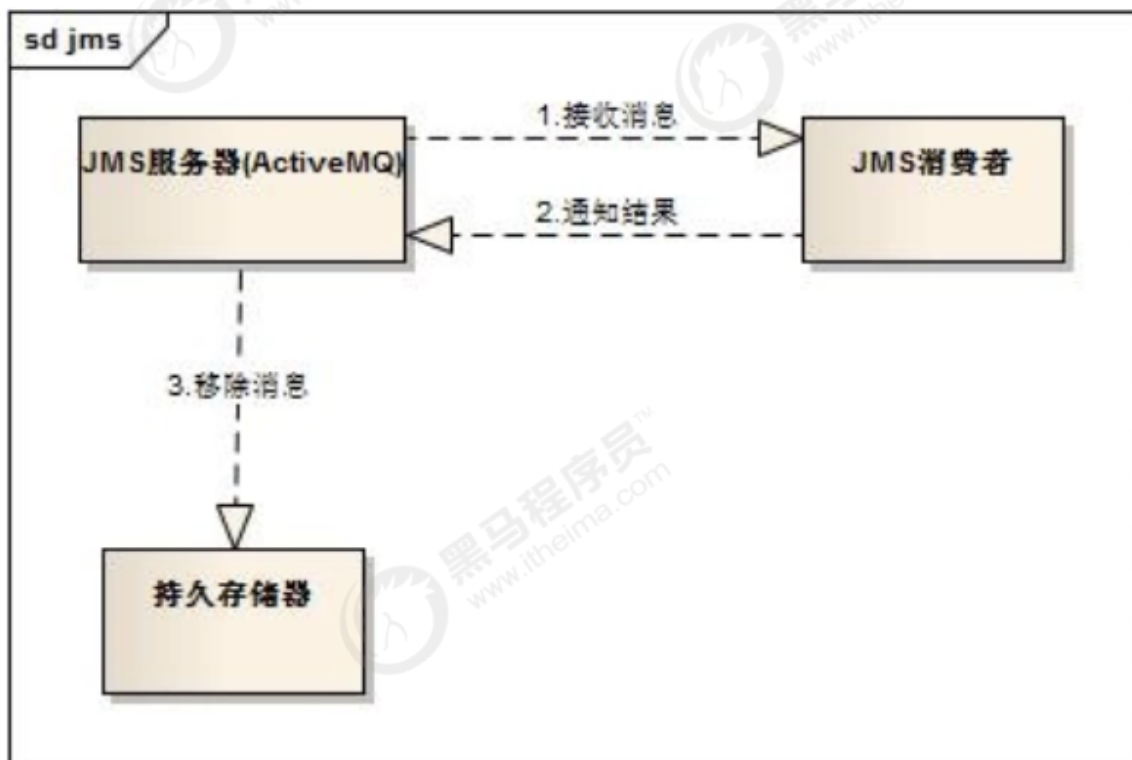
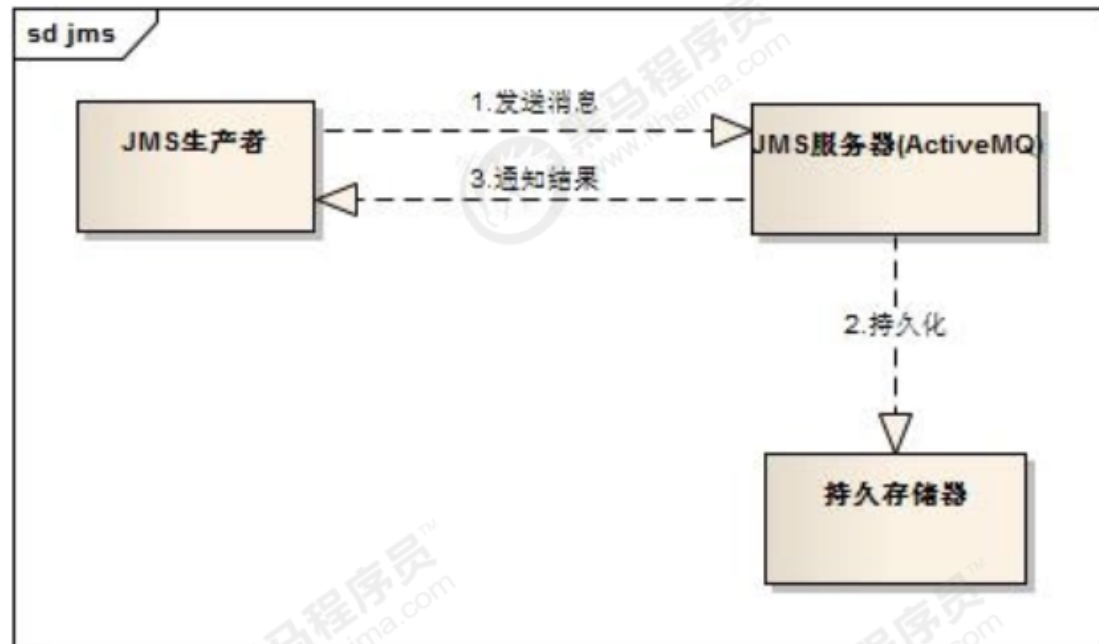
消息持久化

消息持久化是保证消息不丢失的重要方式!!!

ActiveMQ提供了以下三种的消息存储方式:

- (1) Memory 消息存储-基于内存的消息存储。
- (2) 基于日志消息存储方式, KahaDB是ActiveMQ的默认日志存储方式, 它提供了容量的提升和恢复能力。
- (3) 基于JDBC的消息存储方式-数据存储于数据库 (例如: MySQL) 中。

ActiveMQ持久化机制流程图:



====JDBC消息存储=====

1) application.yml

```
server:
  port: 9001
spring:
  activemq:
    broker-url: tcp://192.168.66.133:61616
    user: admin
    password: admin
  jms:
    pub-sub-domain: false # false: 点对点队列模式, true: 发布/订阅模式
    template:
      delivery-mode: persistent # 持久化
  activemq:
    name: springboot-queue01
```

2) 修改activemq.xml

```
<!--配置数据库连接池-->
<bean name="mysql-ds" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url"
value="jdbc:mysql://192.168.66.133:3306/db_activemq" />
  <property name="username" value="root" />
  <property name="password" value="123456"/>
</bean>

<!--JDBC Jdbc用于master/slave模式的数据库分享 -->
<persistenceAdapter>
  <jdbcPersistenceAdapter dataSource="#mysql-ds"/>
</persistenceAdapter>
```

3) 拷贝mysql及durid数据源的jar包到activemq的lib目录下

4) 重启activemq

消息事务

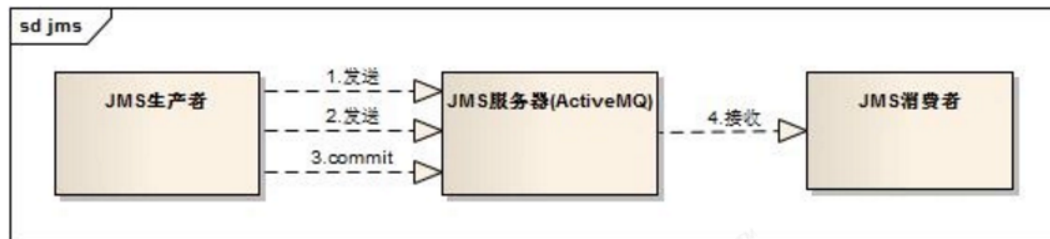
消息事务，是保证消息传递原子性的一个重要特征，和JDBC的事务特征类似。

一个事务性发送，其中一组消息要么能够全部保证到达服务器，要么都不到达服务器。

生产者、消费者与消息服务器直接都支持事务性；

ActionMQ的事务主要偏向在生产者的应用。

ActionMQ消息事务流程图:



一、生产者事务:

方式一:

```
/**
 * 事务性发送--方案一
 */
@Test
public void sendMessageTx(){
    //获取连接工厂
    ConnectionFactory connectionFactory =
jmsMessagingTemplate.getConnectionFactory();

    Session session = null;
    try {
        //创建连接
        Connection connection = connectionFactory.createConnection();

        /**
         * 参数一: 是否开启消息事务
         */
        session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);

        //创建生产者
        MessageProducer producer =
session.createProducer(session.createQueue(name));

        for(int i=1;i<=10;i++){
            //模拟异常
            if(i==4){
                int a = 10/0;
            }

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

            producer.send(textMessage);
        }

        //注意: 一旦开启事务发送, 那么就必须使用commit方法进行事务提交, 否则消息无法到达
MQ服务器
        session.commit();
    }
}
```

```

    } catch (JMSEException e) {
        e.printStackTrace();
        //消息事务回滚
        try {
            session.rollback();
        } catch (JMSEException e1) {
            e1.printStackTrace();
        }
    }
}
}

```

方式二:

配置类:

```

package com.itheima;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.RedeliveryPolicy;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.SimpleJmsListenerContainerFactory;
import org.springframework.jms.connection.JmsTransactionManager;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.transaction.PlatformTransactionManager;

import javax.jms.ConnectionFactory;
import javax.jms.Session;

/**
 *
 */
@Configuration
public class ActiveMqConfig {

    @Bean
    public PlatformTransactionManager transactionManager(ConnectionFactory
connectionFactory) {
        return new JmsTransactionManager(connectionFactory);
    }
}

```

生产者业务类:

```

package com.itheima.producer;

```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

/**
 * 消息发送的业务类
 */
@Service
public class MessageService {

    @Autowired
    private JmsMessagingTemplate jmsMessagingTemplate;
    @Value("${activemq.name}")
    private String name;

    @Transactional // 对消息发送加入事务管理（同时也对JDBC数据库的事务生效）
    public void sendMessage(){
        for(int i=1;i<=10;i++) {
            //模拟异常
            if(i==4){
                int a = 10/0;
            }

            jmsMessagingTemplate.convertAndSend(name, "消息---"+i);
        }
    }
}

```

测试发送方法：

```

@Autowired
private MessageService messageService;

/**
 * 事务性发送--方案二： Spring的JmsTransactionManager功能
 */
@Test
public void sendMessageTx2(){
    messageService.sendMessage();
}

```

二、消费者事务

```

package com.itheima.consumer;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

```

```

import org.springframework.transaction.annotation.Transactional;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;

/**
 * 消息消费者
 */
@Component
public class Consumer {

    /**
     * 接收消息的方法
     */

    @JmsListener(destination="${activemq.name}",containerFactory =
    "jmsQueryListenerFactory")
    public void receiveMessage(TextMessage textMessage,Session session) throws
    JMSEException {
        try {
            System.out.println("消息内容: " + textMessage.getText() + ",是否重发: "
            + textMessage.getJMSRedelivered());

            int i = 100/0; //模拟异常

            session.commit();//提交事务
        } catch (JMSEException e) {
            try {
                session.rollback();//回滚事务
            } catch (JMSEException e1) {
            }
            e.printStackTrace();
        }
    }
}

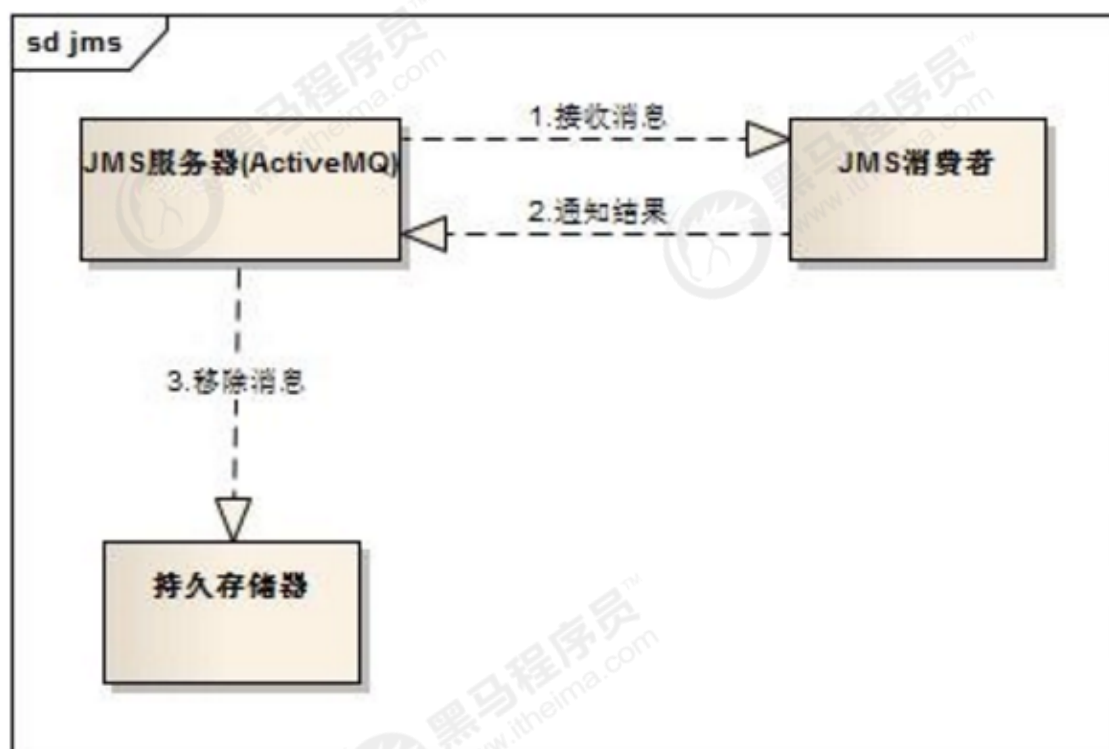
```

消息确认机制

JMS消息只有在被确认之后，才认为已经被成功地消费了。消息的成功消费通常包含三个阶段：客户接收消息、客户处理消息和消息被确认。**在事务性会话中，当一个事务被提交的时候，确认自动发生。**在非事务性会话中，消息何时被确认取决于创建会话时的应答模式（acknowledgement mode）。该参数有以下三个可选值：

值	描述
Session.AUTO_ACKNOWLEDGE	当客户成功的从receive方法返回的时候，或者从MessageListener.onMessage方法成功返回的时候，会话自动确认客户收到的消息
Session.CLIENT_ACKNOWLEDGE	客户通过消息的acknowledge方法确认消息。需要注意的是，在这种模式中，确认是在会话层上进行：确认一个被消费的消息将自动确认所有已被会话消费的消息。例如，如果一个消息消费者消费了10个消息，然后确认第5个消息，那么所有10个消息都被确认
Session.DUPS_ACKNOWLEDGE	该选择只是会话迟钝确认消息的提交。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

注意：消息确认机制与事务机制是冲突的，只能选其中一种。所以演示消息确认前，先关闭事务。



1) auto_acknowledge 自动确认

添加配置类：

```
package com.itheima;
```

```

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.RedeliveryPolicy;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.SimpleJmsListenerContainerFactory;
import org.springframework.jms.core.JmsTemplate;

import javax.jms.ConnectionFactory;
import javax.jms.Session;

/**
 *
 */
@Configuration
public class ActiveMqConfig {

    @Bean(name="jmsQueryListenerFactory")
    public DefaultJmsListenerContainerFactory
jmsListenerContainerFactory(ConnectionFactory connectionFactory){
        DefaultJmsListenerContainerFactory factory=new
DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setSessionTransacted(false); // 不开启事务操作
        factory.setSessionAcknowledgeMode(1); //自动确认
        return factory;
    }
}

```

消费者:

```

package com.itheima.consumer;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;

/**
 * 消息消费者
 */
@Component
public class Consumer {

    /**
     * 接收消息的方法
     */
}

```

```

    @JmsListener(destination="${activemq.name}", containerFactory =
    "jmsQueryListenerFactory")
    public void receiveMessage(TextMessage textMessage){

        try {
            System.out.println("消息内容: " + textMessage.getText() + ", 是否重发: "
+ textMessage.getJMSRedelivered());
            throw new RuntimeException("test");
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }

}

```

如果消费方接收消息失败，JMS服务器会重发消息，默认重发6次。

2) dups_ok_acknowledge

类似于 auto_acknowledge 确认机制，为自动批量确认而生，而且具有“延迟”确认的特点，ActiveMQ 会根据内部算法，在收到一定数量的消息自动进行确认。在此模式下，可能会出现重复消息，如果消费方不允许重复消费，不建议使用！

3) client_acknowledge 手动确认

配置类：

```

package com.itheima;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.RedeliveryPolicy;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.SimpleJmsListenerContainerFactory;
import org.springframework.jms.core.JmsTemplate;

import javax.jms.ConnectionFactory;
import javax.jms.Session;

/**
 *
 */
@Configuration
public class ActiveMqConfig {

    @Bean(name="jmsQueryListenerFactory")

```



```

    public DefaultJmsListenerContainerFactory
    jmsListenerContainerFactory(ConnectionFactory connectionFactory){
        DefaultJmsListenerContainerFactory factory=new
    DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setSessionTransacted(false); // 不开启事务操作
        factory.setSessionAcknowledgeMode(4); //手动确认
        return factory;
    }
}

```

消费者:

```

package com.itheima.consumer;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;

/**
 * 消息消费者
 */
@Component
public class Consumer {

    /**
     * 接收消息的方法
     */

    @JmsListener(destination="${activemq.name}",containerFactory =
    "jmsQueryListenerFactory")
    public void receiveMessage(TextMessage textMessage){

        try {
            System.out.println("消息内容: " + textMessage.getText() + ",是否重发: "
+ textMessage.getJMSRedelivered());
            textMessage.acknowledge(); // 确认收到消息,一旦消息确认,消息不会重新发送
            throw new RuntimeException("test");
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }

}

```

消息投递方式

异步投递

1、异步投递 vs 同步投递

同步发送:

消息生产者使用持久 (Persistent) 传递模式发送消息的时候, `Producer.send()` 方法会被阻塞, 直到 broker 发送一个确认消息给生产者(`ProducerAck`), 这个确认消息暗示broker已经成功接收到消息并把消息保存到二级存储中。

异步发送:

如果应用程序能够容忍一些消息的丢失, 那么可以使用异步发送。异步发送不会在受到broker的确认之前一直阻塞 `Producer.send`方法。

想要使用异步, 在brokerURL中增加 `jms.alwaysSyncSend=false&jms.useAsyncSend=true`属性

1) 如果设置了`alwaysSyncSend=true`系统将会忽略`useAsyncSend`设置的值都采用同步 2) 当`alwaysSyncSend=false`时, "NON_PERSISTENT"(非持久化)、事务中的消息将使用"异步发送" 3) 当`alwaysSyncSend=false`时, 如果指定了`useAsyncSend=true`, "PERSISTENT"类型的消息使用异步发送。如果`useAsyncSend=false`, "PERSISTENT"类型的消息使用同步发送。

总结: 默认情况(`alwaysSyncSend=false,useAsyncSend=false`), 非持久化消息、事务内的消息均采用异步发送; 对于持久化消息采用同步发送!!!

2、配置异步投递的方式

1. 在连接上配置

```
new ActiveMQConnectionFactory("tcp://localhost:61616?jms.useAsyncSend=true");
```

2. 通过ConnectionFactory

```
((ActiveMQConnectionFactory)connectionFactory).setUseAsyncSend(true);
```

3. 通过connection

```
((ActiveMQConnection)connection).setUseAsyncSend(true);
```

注意: 如果是Spring或SpringBoot项目, 通过修改JmsTemplate的默认参数实现异步或同步投递

```
@Configuration
public class ActiveConfig {
```

```

/**
 * 配置用于异步发送的非持久化JmsTemplate
 */
@Autowired
@Bean
public JmsTemplate asyncJmsTemplate(PooledConnectionFactory
pooledConnectionFactory) {
    JmsTemplate template = new JmsTemplate(pooledConnectionFactory);
    template.setExplicitQosEnabled(true);
    template.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    return template;
}

/**
 * 配置用于同步发送的持久化JmsTemplate
 */
@Autowired
@Bean
public JmsTemplate synJmsTemplate(PooledConnectionFactory
pooledConnectionFactory) {
    JmsTemplate template = new JmsTemplate(pooledConnectionFactory);
    return template;
}

```

异步投递如何确认发送成功？

异步投递丢失消息的场景是：生产者设置 UseAsyncSend=true，使用 producer.send (msg) 持续发送消息。

由于消息不阻塞，生产者会认为所有 send 的消息均被成功发送至 MQ。如果 MQ 突然宕机，此时生产者端内存中尚未被发送至 MQ 的消息都会丢失。

这时，可以给异步投递方法接收回调，以确认消息是否发送成功！

```

/**
 * 异步投递，回调函数
 * @return
 */
@RequestMapping("/send")
public String sendQueue(){
    Connection connection = null;
    Session session = null;
    ActiveMQMessageProducer producer = null;
    // 获取连接工厂
    ConnectionFactory connectionFactory =
jmsMessagingTemplate.getConnectionFactory();

    try {
        connection = connectionFactory.createConnection();

        session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);

```

```

Queue queue = session.createQueue(name);
int count = 10;

producer = session.createProducer(queue);

producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

long start = System.currentTimeMillis();
for (int i = 0; i < count; i++) {

    //创建需要发送的消息
    TextMessage textMessage = session.createTextMessage("Hello");
    //设置消息唯一ID
    String msgid = UUID.randomUUID().toString();
    textMessage.setJMSMessageID(msgid);

    producer.send(textMessage, new AsyncCallback() {
        @Override
        public void onSuccess() {
            // 使用msgid标识来进行消息发送成功的处理
            System.out.println(msgid+" 消息发送成功");
        }

        @Override
        public void onException(JMSEException exception) {
            // 使用msgid表示进行消息发送失败的处理
            System.out.println(msgid+" 消息发送失败");
            exception.printStackTrace();
        }
    });

    session.commit();
} catch (Exception e) {
    e.printStackTrace();
}

return "ok";
}

```

延迟投递

生产者提供两个发送消息的方法，一个是即时发送消息，一个是延时发送消息。

1、修改activemq.xml

```
<broker xmlns="http://activemq.apache.org/schema/core" ...
  schedulerSupport="true" >
  .....
</broker>
```

注意：添加schedulerSupport="true"配置

2、在代码中设置延迟时长

```
/**
 * 延时投递
 *
 * @return
 */
@Test
public String sendQueue() {
    Connection connection = null;
    Session session = null;
    ActiveMQMessageProducer producer = null;
    // 获取连接工厂
    ConnectionFactory connectionFactory =
        JmsMessagingTemplate.getConnectionFactory();

    try {
        connection = connectionFactory.createConnection();
        session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
        Queue queue = session.createQueue(name);
        int count = 10;

        producer = (ActiveMQMessageProducer) session.createProducer(queue);

        producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
        //创建需要发送的消息
        TextMessage textMessage = session.createTextMessage("Hello");

        //设置延时时长(延时10秒)
        textMessage.setLongProperty(ScheduledMessage.AMQ_SCHEDULED_DELAY,
10000);

        producer.send(textMessage);

        session.commit();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return "ok";
}
```

定时投递

1、启动类添加定时注解

```
@SpringBootApplication
@EnableScheduling // 开启定时功能
public class MyActiveMQApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyActiveMQApplication.class, args);
    }

}
```

2、在生产者添加@Scheduled设置定时

```
/**
 * 消息生产者
 */
@Component
public class ProducerController3 {

    @Value("${activemq.name}")
    private String name;

    @Autowired
    private JmsMessagingTemplate jmsMessagingTemplate;

    /**
     * 延时投递
     *
     * @return
     */
    //每隔3秒定投
    @Scheduled(fixedDelay = 3000)
    public void sendQueue() {
        jmsMessagingTemplate.convertAndSend(name, "消息ID:" +
        UUID.randomUUID().toString().substring(0,6));
        System.out.println("消息发送成功...");
    }

}
```

死信队列

DLQ-Dead Letter Queue, 死信队列, 用来保存处理失败或者过期的消息

出现以下情况时，消息会被重发：

A transacted session is used and rollback() is called.
A transacted session is closed before commit is called.
A session is using CLIENT_ACKNOWLEDGE and Session.recover() is called.

当一个消息被重发超过6(缺省为6次)次数时，会给broker发送一个"Poison ack"，这个消息被认为是a poison pill，这时broker会将这个消息发送到死信队列，以便后续处理。

注意两点：

- 1) 缺省持久消息过期，会被送到DLQ，非持久消息不会送到DLQ
- 2) 缺省的死信队列是ActiveMQ.DLQ，如果没有特别指定，死信都会被发送到这个队列。

可以通过配置文件(activemq.xml)来调整死信发送策略。

1、修改activemq.xml

为每个队列建立独立的死信队列

```
<destinationPolicy>
  <policyMap>
    <policyEntries>
      <policyEntry queue="">
        <deadLetterStrategy>
          <individualDeadLetterStrategy queuePrefix="DLQ."
            useQueueForQueueMessages="true" />
        </deadLetterStrategy>
      </policyEntry>

      <policyEntry topic="" >
        <pendingMessageLimitStrategy>
          <constantPendingMessageLimitStrategy limit="1000"/>
        </pendingMessageLimitStrategy>
      </policyEntry>
    </policyEntries>
  </policyMap>
</destinationPolicy>
```

2、RedeliveryPolicy重发策略设置

修改启动类

```
package com.itheima;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.RedeliveryPolicy;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
```

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.SimpleJmsListenerContainerFactory;
import org.springframework.jms.connection.JmsTransactionManager;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.transaction.PlatformTransactionManager;

import javax.jms.ConnectionFactory;
import javax.jms.Session;

/**
 *
 */
@Configuration
public class ActiveMqConfig {

    //RedeliveryPolicy重发策略设置
    @Bean
    public RedeliveryPolicy redeliveryPolicy(){
        RedeliveryPolicy redeliveryPolicy= new RedeliveryPolicy();
        //是否在每次尝试重新发送失败后,增长这个等待时间
        redeliveryPolicy.setUseExponentialBackOff(true);
        //重发次数,默认为6次 这里设置为10次
        redeliveryPolicy.setMaximumRedeliveries(10);
        //重发时间间隔,默认为1秒
        redeliveryPolicy.setInitialRedeliveryDelay(1);
        //第一次失败后重新发送之前等待500毫秒,第二次失败再等待500 * 2毫秒,这里的2就是value
        redeliveryPolicy.setBackOffMultiplier(2);
        //是否避免消息碰撞
        redeliveryPolicy.setUseCollisionAvoidance(false);
        //设置重发最大拖延时间-1 表示没有拖延只有UseExponentialBackOff(true)为true时生效
        redeliveryPolicy.setMaximumRedeliveryDelay(-1);

        return redeliveryPolicy;
    }

    @Bean
    public ActiveMQConnectionFactory activeMQConnectionFactory
    (@Value("${spring.activemq.broker-url}")String url,RedeliveryPolicy
    redeliveryPolicy){
        ActiveMQConnectionFactory activeMQConnectionFactory =
            new ActiveMQConnectionFactory(
                "admin",
                "admin",
                url);
        activeMQConnectionFactory.setRedeliveryPolicy(redeliveryPolicy);
        return activeMQConnectionFactory;
    }

    @Bean
    public PlatformTransactionManager transactionManager(ConnectionFactory
    connectionFactory) {
        return new JmsTransactionManager(connectionFactory);
    }
}

```



```

@Bean(name="jmsQueryListenerFactory")
public DefaultJmsListenerContainerFactory
jmsListenerContainerFactory(ConnectionFactory
connectionFactory,PlatformTransactionManager transactionManager){
    DefaultJmsListenerContainerFactory factory=new
DefaultJmsListenerContainerFactory ();
    factory.setTransactionManager(transactionManager);
    factory.setConnectionFactory(connectionFactory);
    factory.setSessionTransacted(true); // 开启事务
    factory.setSessionAcknowledgeMode(1);
    return factory;
}
}

```

07、ActiveMQ企业面试经典问题

问题：ActiveMQ宕机了怎么办？

1) ActiveMQ主从集群方案：Zookeeper集群+ Replicated LevelDB + ActiveMQ集群

官网链接：<http://activemq.apache.org/replicated-leveldb-store>



2) 集群信息概览

Zookeeper端口	ActiveMQweb端口	ActiveMQ协议端口
2181	8161	61616
2182	8162	61617
2183	8163	61618

3) 先搭建Zookeeper集群

1) 上传zookeeper-3.4.6.tar.gz到linux

2) 解压: `tar -xzf zookeeper-3.4.6.tar.gz`

3) 创建根目录: `mkdir /root/zookeeper`

4) 创建节点目录及数据, 日志存放目录:

```
mkdir -p zookeeper/218{1,2,3}/{data,datalogs}
```

3个节点的子文件夹为: 2181,2182,2183

5) 复制Zookeeper到每个节点目录下

```
cp -r zookeeper-3.4.6 zookeeper/2181
```

```
cp -r zookeeper-3.4.6 zookeeper/2182
```

```
cp -r zookeeper-3.4.6 zookeeper/2183
```

6) 移除原始目录

```
rm -rf zookeeper-3.4.14/
```

7) 修改2181节点的zoo.cfg

```
cd zookeeper/2181/zookeeper-3.4.6/conf/
```

```
cp zoo_sample.cfg zoo.cfg
```

```
vi zoo.cfg
```

内容如下:

```
clientPort=2181
```

```
dataDir=/root/zookeeper/2181/data
```

```
dataLogDir=/root/zookeeper/2181/datalogs
```

```
server.1=192.168.66.133:2881:3881
```

```
server.2=192.168.66.133:2882:3882
```

```
server.3=192.168.66.133:2883:3883
```

8) 相同方式修改2182及2183节点的zoo.cfg

2182配置为:

```
clientPort=2182
```

```
dataDir=/root/zookeeper/2182/data
```

```
dataLogDir=/root/zookeeper/2182/datalogs
```

```
server.1=192.168.66.133:2881:3881
```

```
server.2=192.168.66.133:2882:3882
```

```
server.3=192.168.66.133:2883:3883
```

2183配置为:

```
clientPort=2183
```

```
dataDir=/root/zookeeper/2183/data
```

```
dataLogDir=/root/zookeeper/2183/datalogs
```

```
server.1=192.168.66.133:2881:3881
```

```
server.2=192.168.66.133:2882:3882
```

```
server.3=192.168.66.133:2883:3883
```

9) 每个节点必须有myid配置文件，记录节点的唯一标识，必须放在dataDir文件夹下。而且id值必须与上面配置的server.x中的x对应

```
touch 2181/data/myid && echo "1" > 2181/data/myid
touch 2182/data/myid && echo "2" > 2182/data/myid
touch 2183/data/myid && echo "3" > 2183/data/myid
```

查看是否创建成功:

```
more 218*/data/myid
```

10) 分别启动三台Zookeeper

启动:

```
cd 2181/ && zookeeper-3.4.6/bin/zkServer.sh start
cd 2182/ && zookeeper-3.4.6/bin/zkServer.sh start
cd 2183/ && zookeeper-3.4.6/bin/zkServer.sh start
```

查看状态:

```
2181/zookeeper-3.4.6/bin/zkServer.sh status
2182/zookeeper-3.4.6/bin/zkServer.sh status
2183/zookeeper-3.4.6/bin/zkServer.sh status
```

看到Mode: leader的Zookeeper为主节点，其他为从节点。

4) 搭建ActiveMQ集群

1) 上传apache-activemq-5.15.9-bin.tar.gz到linux

2) 解压: tar -xzf apache-activemq-5.15.9-bin.tar.gz

3) 创建三个节点目录

```
mkdir activemq
mkdir -p activemq/816{1,2,3}
```

4) 复制activemq到每个节点目录

```
cp -r apache-activemq-5.15.9 activemq/8161
cp -r apache-activemq-5.15.9 activemq/8162
cp -r apache-activemq-5.15.9 activemq/8163
```

5) 修改每个节点的activemq.xml

必须使用相同的集群名称

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

添加配置:

```
61616:
<persistenceAdapter>
<replicatedLevelDB
```

```

directory="${activemq.data}/leveldb"
replicas="3"
bind="tcp://0.0.0.0:61616"
zkAddress="192.168.66.133:2181,192.168.66.133:2182,192.168.66.133:2183"
hostname="192.168.66.133"
zkPath="/activemq/leveldb-stores" />
</persistenceAdapter>

```

```

61617:
<persistenceAdapter>
<replicatedLevelDB
directory="${activemq.data}/leveldb"
replicas="3"
bind="tcp://0.0.0.0:61617"
zkAddress="192.168.66.133:2181,192.168.66.133:2182,192.168.66.133:2183"
hostname="192.168.66.133"
zkPath="/activemq/leveldb-stores" />
</persistenceAdapter>

```

```

61618:
<persistenceAdapter>
<replicatedLevelDB
directory="${activemq.data}/leveldb"
replicas="3"
bind="tcp://0.0.0.0:61618"
zkAddress="192.168.66.133:2181,192.168.66.133:2182,192.168.66.133:2183"
hostname="192.168.66.133"
zkPath="/activemq/leveldb-stores" />
</persistenceAdapter>

```

6) 修改jetty.xml

```

<bean id="jettyPort" class="org.apache.activemq.web.WebConsolePort" init-
method="start">
    <!-- the default port number for the web console -->
    <property name="host" value="0.0.0.0"/>
    <property name="port" value="8161"/>
</bean>

```

分别为8181,8182,8183

7) 分别启动每台activemq

可以使用ZooInspector工具查看ActiveMQ是否注册成功

5) 生产者和消费者的broker-url需要修改

```
server:
  port: 9001
spring:
  activemq:
    broker-url: failover:
      (tcp://192.168.66.133:61616,tcp://192.168.66.133:61617,tcp://192.168.66.133:6161
      68)
    user: admin
    password: admin
```

问题：如何防止消费方消息重复消费？

解决消费方幂等性问题！

如果因为网络延迟等原因，MQ无法及时接收到消费方的应答，导致MQ重试。在重试过程中造成重复消费的问题。

解决思路：1) 如果消费方是做数据库操作，那么可以把消息的ID作为表的唯一主键，这样在重试的情况下，会触发主键冲突，从而避免数据出现脏数据。

2) 如果消费方不是做数据库操作，那么可以借助第三方的应用，例如Redis，来记录消费记录。每次消息被消费完成时候，把当前消息的ID作为key存入redis，每次消费前，先到redis查询有没有该消息的消费记录。

问题：如何防止消息丢失？

以下手段可以防止消息丢失：

- 1) 在消息生产者和消费者使用事务
- 2) 在消费方采用手动消息确认（ACK）
- 3) 消息持久化，例如DBC或日志

问题：什么是死信队列？

前面已经讲过该问题