**1.1     JMS简介**

       JMS的全称是Java Message Service，即Java消 息服务。它主要用于在生产者和消费者之间进行消息传递，生产者负责产生消息，而消费者负责接收消息。把它应用到实际的业务需求中的话我们可以在特定的时候 利用生产者生成一消息，并进行发送，对应的消费者在接收到对应的消息后去完成对应的业务逻辑。对于消息的传递有两种类型，一种是点对点的，即一个生产者和 一个消费者一一对应；另一种是发布/订阅模式，即一个生产者产生消息并进行发送后，可以由多个消费者进行接收。

**1.2     Spring整合JMS**

       对JMS做了一个简要介绍之后，接下来就讲一下Spring整合JMS的具体过程。JMS只是一个标准，真正在使用它的时候我们需要有它的具体实现，这里我们就使用Apache的activeMQ来作为它的实现。所使用的依赖利用Maven来进行管理，具体依赖如下：

Xml代码  [收藏代码](javascript:void())

1. <dependencies>
2. <dependency>
3. <groupId>junit</groupId>
4. <artifactId>junit</artifactId>
5. <version>4.10</version>
6. <scope>test</scope>
7. </dependency>
8. <dependency>
9. <groupId>org.springframework</groupId>
10. <artifactId>spring-context</artifactId>
11. <version>${spring-version}</version>
12. </dependency>
13. <dependency>
14. <groupId>org.springframework</groupId>
15. <artifactId>spring-jms</artifactId>
16. <version>${spring-version}</version>
17. </dependency>
18. <dependency>
19. <groupId>org.springframework</groupId>
20. <artifactId>spring-test</artifactId>
21. <version>${spring-version}</version>
22. </dependency>
23. <dependency>
24. <groupId>javax.annotation</groupId>
25. <artifactId>jsr250-api</artifactId>
26. <version>1.0</version>
27. </dependency>
28. <dependency>
29. <groupId>org.apache.activemq</groupId>
30. <artifactId>activemq-core</artifactId>
31. <version>5.7.0</version>
32. </dependency>
33. </dependencies>

**1.2.1  activeMQ准备**

       既然是使用的apache的activeMQ作为JMS的实现，那么首先我们应该到apache官网上下载activeMQ（<http://activemq.apache.org/download.html>），进行解压后运行其bin目录下面的activemq.bat文件启动activeMQ。

**1.2.2配置ConnectionFactory**

       ConnectionFactory是用于产生到JMS服务器的链接的，Spring为我们提供了多个ConnectionFactory，有SingleConnectionFactory和CachingConnectionFactory。SingleConnectionFactory对于建立JMS服务器链接的请求会一直返回同一个链接，并且会忽略Connection的close方法调用。CachingConnectionFactory继承了SingleConnectionFactory，所以它拥有SingleConnectionFactory的所有功能，同时它还新增了缓存功能，它可以缓存Session、MessageProducer和MessageConsumer。这里我们使用SingleConnectionFactory来作为示例。

Xml代码  [收藏代码](javascript:void())

1. <bean id="connectionFactory" class="org.springframework.jms.connection.SingleConnectionFactory"/>

       这样就定义好产生JMS服务器链接的ConnectionFactory了吗？答案是非也。Spring提供的ConnectionFactory只是Spring用于管理ConnectionFactory的，真正产生到JMS服务器链接的ConnectionFactory还得是由JMS服务厂商提供，并且需要把它注入到Spring提供的ConnectionFactory中。我们这里使用的是ActiveMQ实现的JMS，所以在我们这里真正的可以产生Connection的就应该是由ActiveMQ提供的ConnectionFactory。所以定义一个ConnectionFactory的完整代码应该如下所示：

Xml代码  [收藏代码](javascript:void())

1. <!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供-->
2. <bean id="targetConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
3. <property name="brokerURL" value="tcp://localhost:61616"/>
4. </bean>
6. <!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->
7. <bean id="connectionFactory" class="org.springframework.jms.connection.SingleConnectionFactory">
8. <!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->
9. <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
10. </bean>

       ActiveMQ为我们提供了一个PooledConnectionFactory，通过往里面注入一个ActiveMQConnectionFactory可以用来将Connection、Session和MessageProducer池化，这样可以大大的减少我们的资源消耗。当使用PooledConnectionFactory时，我们在定义一个ConnectionFactory时应该是如下定义：

Xml代码  [收藏代码](javascript:void())

1. <!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供-->
2. <bean id="targetConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
3. <property name="brokerURL" value="tcp://localhost:61616"/>
4. </bean>
6. <bean id="pooledConnectionFactory" class="org.apache.activemq.pool.PooledConnectionFactory">
7. <property name="connectionFactory" ref="targetConnectionFactory"/>
8. <property name="maxConnections" value="10"/>
9. </bean>
11. <bean id="connectionFactory" class="org.springframework.jms.connection.SingleConnectionFactory">
12. <property name="targetConnectionFactory" ref="pooledConnectionFactory"/>
13. </bean>

**1.2.3配置生产者**

配置好ConnectionFactory之后我们就需要配置生产者。生产者负责产生消息并发送到JMS服务器，这通常对应的是我们的一个业务逻辑服务实现类。但是我们的服务实现类是怎么进行消息的发送的呢？这通常是利用Spring为我们提供的JmsTemplate类来实现的，所以配置生产者其实最核心的就是配置进行消息发送的JmsTemplate。对于消息发送者而言，它在发送消息的时候要知道自己该往哪里发，为此，我们在定义JmsTemplate的时候需要往里面注入一个Spring提供的ConnectionFactory对象。

Xml代码  [收藏代码](javascript:void())

1. <!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->
2. <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
3. <!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->
4. <property name="connectionFactory" ref="connectionFactory"/>
5. </bean>

       在真正利用JmsTemplate进行消息发送的时候，我们需要知道消息发送的目的地，即destination。在Jms中有一个用来表示目的地的Destination接口，它里面没有任何方法定义，只是用来做一个标识而已。当我们在使用JmsTemplate进行消息发送时没有指定destination的时候将使用默认的Destination。默认Destination可以通过在定义jmsTemplate bean对象时通过属性defaultDestination或defaultDestinationName来进行注入，defaultDestinationName对应的就是一个普通字符串。在ActiveMQ中实现了两种类型的Destination，一个是点对点的ActiveMQQueue，另一个就是支持订阅/发布模式的ActiveMQTopic。在定义这两种类型的Destination时我们都可以通过一个name属性来进行构造，如：

Xml代码  [收藏代码](javascript:void())

1. <!--这个是队列目的地，点对点的-->
2. <bean id="queueDestination" class="org.apache.activemq.command.ActiveMQQueue">
3. <constructor-arg>
4. <value>queue</value>
5. </constructor-arg>
6. </bean>
7. <!--这个是主题目的地，一对多的-->
8. <bean id="topicDestination" class="org.apache.activemq.command.ActiveMQTopic">
9. <constructor-arg value="topic"/>
10. </bean>

       假设我们定义了一个ProducerService，里面有一个向Destination发送纯文本消息的方法sendMessage，那么我们的代码就大概是这个样子：

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.springintejms.service.impl;
3. import javax.annotation.Resource;
4. import javax.jms.Destination;
5. import javax.jms.JMSException;
6. import javax.jms.Message;
7. import javax.jms.Session;
9. import org.springframework.jms.core.JmsTemplate;
10. import org.springframework.jms.core.MessageCreator;
11. import org.springframework.stereotype.Component;
13. import com.tiantian.springintejms.service.ProducerService;
15. @Component
16. public class ProducerServiceImpl implements ProducerService {
18. private JmsTemplate jmsTemplate;
20. public void sendMessage(Destination destination, final String message) {
21. System.out.println("---------------生产者发送消息-----------------");
22. System.out.println("---------------生产者发了一个消息：" + message);
23. jmsTemplate.send(destination, new MessageCreator() {
24. public Message createMessage(Session session) throws JMSException {
25. return session.createTextMessage(message);
26. }
27. });
28. }
30. public JmsTemplate getJmsTemplate() {
31. returnjmsTemplate;
32. }
34. @Resource
35. public void setJmsTemplate(JmsTemplate jmsTemplate) {
36. this.jmsTemplate = jmsTemplate;
37. }
39. }

       我们可以看到在sendMessage方法体里面我们是通过jmsTemplate来发送消息到对应的Destination的。到此，我们生成一个简单的文本消息并把它发送到指定目的地Destination的生产者就配置好了。

**1.2.4配置消费者**

生产者往指定目的地Destination发送消息后，接下来就是消费者对指定目的地的消息进行消费了。那么消费者是如何知道有生产者发送消息到指定目的地Destination了呢？这是通过Spring为我们封装的消息监听容器MessageListenerContainer实现的，它负责接收信息，并把接收到的信息分发给真正的MessageListener进行处理。每个消费者对应每个目的地都需要有对应的MessageListenerContainer。对于消息监听容器而言，除了要知道监听哪个目的地之外，还需要知道到哪里去监听，也就是说它还需要知道去监听哪个JMS服务器，这是通过在配置MessageConnectionFactory的时候往里面注入一个ConnectionFactory来实现的。所以我们在配置一个MessageListenerContainer的时候有三个属性必须指定，一个是表示从哪里监听的ConnectionFactory；一个是表示监听什么的Destination；一个是接收到消息以后进行消息处理的MessageListener。Spring一共为我们提供了两种类型的MessageListenerContainer，SimpleMessageListenerContainer和DefaultMessageListenerContainer。

SimpleMessageListenerContainer会在一开始的时候就创建一个会话session和消费者Consumer，并且会使用标准的JMS MessageConsumer.setMessageListener()方法注册监听器让JMS提供者调用监听器的回调函数。它不会动态的适应运行时需要和参与外部的事务管理。兼容性方面，它非常接近于独立的JMS规范，但一般不兼容Java EE的JMS限制。

大多数情况下我们还是使用的DefaultMessageListenerContainer，跟SimpleMessageListenerContainer相比，DefaultMessageListenerContainer会动态的适应运行时需要，并且能够参与外部的事务管理。它很好的平衡了对JMS提供者要求低、先进功能如事务参与和兼容Java EE环境。

**定义处理消息的MessageListener**

       要定义处理消息的MessageListener我们只需要实现JMS规范中的MessageListener接口就可以了。MessageListener接口中只有一个方法onMessage方法，当接收到消息的时候会自动调用该方法。

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.springintejms.listener;
3. import javax.jms.JMSException;
4. import javax.jms.Message;
5. import javax.jms.MessageListener;
6. import javax.jms.TextMessage;
8. public class ConsumerMessageListener implements MessageListener {
10. public void onMessage(Message message) {
11. //这里我们知道生产者发送的就是一个纯文本消息，所以这里可以直接进行强制转换
12. TextMessage textMsg = (TextMessage) message;
13. System.out.println("接收到一个纯文本消息。");
14. try {
15. System.out.println("消息内容是：" + textMsg.getText());
16. } catch (JMSException e) {
17. e.printStackTrace();
18. }
19. }
21. }

       有了MessageListener之后我们就可以在Spring的配置文件中配置一个消息监听容器了。

Xml代码  [收藏代码](javascript:void())

1. <!--这个是队列目的地-->
2. <bean id="queueDestination" class="org.apache.activemq.command.ActiveMQQueue">
3. <constructor-arg>
4. <value>queue</value>
5. </constructor-arg>
6. </bean>
7. <!-- 消息监听器 -->
8. <bean id="consumerMessageListener" class="com.tiantian.springintejms.listener.ConsumerMessageListener"/>
10. <!-- 消息监听容器 -->
11. <bean id="jmsContainer"        class="org.springframework.jms.listener.DefaultMessageListenerContainer">
12. <property name="connectionFactory" ref="connectionFactory" />
13. <property name="destination" ref="queueDestination" />
14. <property name="messageListener" ref="consumerMessageListener" />
15. </bean>

       我们可以看到我们定义了一个名叫queue的ActiveMQQueue目的地，我们的监听器就是监听了发送到这个目的地的消息。

       至此我们的生成者和消费者都配置完成了，这也就意味着我们的整合已经完成了。这个时候完整的Spring的配置文件应该是这样的：

Xml代码  [收藏代码](javascript:void())

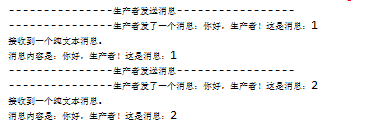
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
4. xmlns:jms="http://www.springframework.org/schema/jms"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/context
8. http://www.springframework.org/schema/context/spring-context-3.0.xsd
9. http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
10. http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">
12. <context:component-scan base-package="com.tiantian" />
14. <!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->
15. <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
16. <!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->
17. <property name="connectionFactory" ref="connectionFactory"/>
18. </bean>
20. <!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供-->
21. <bean id="targetConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
22. <property name="brokerURL" value="tcp://localhost:61616"/>
23. </bean>
25. <!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->
26. <bean id="connectionFactory" class="org.springframework.jms.connection.SingleConnectionFactory">
27. <!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->
28. <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
29. </bean>
31. <!--这个是队列目的地-->
32. <bean id="queueDestination" class="org.apache.activemq.command.ActiveMQQueue">
33. <constructor-arg>
34. <value>queue</value>
35. </constructor-arg>
36. </bean>
37. <!-- 消息监听器 -->
38. <bean id="consumerMessageListener" class="com.tiantian.springintejms.listener.ConsumerMessageListener"/>
39. <!-- 消息监听容器 -->
40. <bean id="jmsContainer"
41. class="org.springframework.jms.listener.DefaultMessageListenerContainer">
42. <property name="connectionFactory" ref="connectionFactory" />
43. <property name="destination" ref="queueDestination" />
44. <property name="messageListener" ref="consumerMessageListener" />
45. </bean>
46. </beans>

       接着我们来测试一下，看看我们的整合是否真的成功了，测试代码如下：

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.springintejms.test;
3. import javax.jms.Destination;
5. import org.junit.Test;
6. import org.junit.runner.RunWith;
7. import org.springframework.beans.factory.annotation.Autowired;
8. import org.springframework.beans.factory.annotation.Qualifier;
9. import org.springframework.test.context.ContextConfiguration;
10. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
11. import com.tiantian.springintejms.service.ProducerService;
13. @RunWith(SpringJUnit4ClassRunner.class)
14. @ContextConfiguration("/applicationContext.xml")
15. public class ProducerConsumerTest {
17. @Autowired
18. private ProducerService producerService;
19. @Autowired
20. @Qualifier("queueDestination")
21. private Destination destination;
23. @Test
24. public void testSend() {
25. for (int i=0; i<2; i++) {
26. producerService.sendMessage(destination, "你好，生产者！这是消息：" + (i+1));
27. }
28. }
30. }

       在上面的测试代码中我们利用生产者发送了两个消息，正常来说，消费者应该可以接收到这两个消息。运行测试代码后控制台输出如下：



       看，控制台已经进行了正确的输出，这说明我们的整合确实是已经成功了。

**1.3     消息监听器MessageListener**

       在Spring整合JMS的应用中我们在定义消息监听器的时候一共可以定义三种类型的消息监听器，分别是MessageListener、SessionAwareMessageListener和MessageListenerAdapter。下面就分别来介绍一下这几种类型的区别。

**1.3.1  MessageListener**

MessageListener是最原始的消息监听器，它是JMS规范中定义的一个接口。其中定义了一个用于处理接收到的消息的onMessage方法，该方法只接收一个Message参数。我们前面在讲配置消费者的时候用的消息监听器就是MessageListener，代码如下：

Java代码  [收藏代码](javascript:void())

1. import javax.jms.JMSException;
2. import javax.jms.Message;
3. import javax.jms.MessageListener;
4. import javax.jms.TextMessage;
6. public class ConsumerMessageListener implements MessageListener {
8. public void onMessage(Message message) {
9. //这里我们知道生产者发送的就是一个纯文本消息，所以这里可以直接进行强制转换，或者直接把onMessage方法的参数改成Message的子类TextMessage
10. TextMessage textMsg = (TextMessage) message;
11. System.out.println("接收到一个纯文本消息。");
12. try {
13. System.out.println("消息内容是：" + textMsg.getText());
14. } catch (JMSException e) {
15. e.printStackTrace();
16. }
17. }
19. }

**1.3.2  SessionAwareMessageListener**

SessionAwareMessageListener是Spring为我们提供的，它不是标准的JMS MessageListener。MessageListener的设计只是纯粹用来接收消息的，假如我们在使用MessageListener处理接收到的消息时我们需要发送一个消息通知对方我们已经收到这个消息了，那么这个时候我们就需要在代码里面去重新获取一个Connection或Session。SessionAwareMessageListener的设计就是为了方便我们在接收到消息后发送一个回复的消息，它同样为我们提供了一个处理接收到的消息的onMessage方法，但是这个方法可以同时接收两个参数，一个是表示当前接收到的消息Message，另一个就是可以用来发送消息的Session对象。先来看一段代码：

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.springintejms.listener;
3. import javax.jms.Destination;
4. import javax.jms.JMSException;
5. import javax.jms.Message;
6. import javax.jms.MessageProducer;
7. import javax.jms.Session;
8. import javax.jms.TextMessage;
10. import org.springframework.jms.listener.SessionAwareMessageListener;
12. public class ConsumerSessionAwareMessageListener implements
13. SessionAwareMessageListener<TextMessage> {
15. private Destination destination;
17. public void onMessage(TextMessage message, Session session) throws JMSException {
18. System.out.println("收到一条消息");
19. System.out.println("消息内容是：" + message.getText());
20. MessageProducer producer = session.createProducer(destination);
21. Message textMessage = session.createTextMessage("ConsumerSessionAwareMessageListener。。。");
22. producer.send(textMessage);
23. }
25. public Destination getDestination() {
26. returndestination;
27. }
29. public void setDestination(Destination destination) {
30. this.destination = destination;
31. }
33. }

       在上面代码中我们定义了一个SessionAwareMessageListener，在这个Listener中我们在接收到了一个消息之后，利用对应的Session创建了一个到destination的生产者和对应的消息，然后利用创建好的生产者发送对应的消息。

       接着我们在Spring的配置文件中配置该消息监听器将处理来自一个叫sessionAwareQueue的目的地的消息，并且往该MessageListener中通过set方法注入其属性destination的值为queueDestination。这样当我们的SessionAwareMessageListener接收到消息之后就会往queueDestination发送一个消息。

Xml代码  [收藏代码](javascript:void())

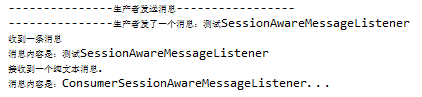
1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
4. xmlns:jms="http://www.springframework.org/schema/jms"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/context
8. http://www.springframework.org/schema/context/spring-context-3.0.xsd
9. http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
10. http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">
12. <context:component-scan base-package="com.tiantian" />
13. <!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->
14. <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
15. <!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->
16. <property name="connectionFactory" ref="connectionFactory"/>
17. </bean>
19. <!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供-->
20. <bean id="targetConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
21. <property name="brokerURL" value="tcp://localhost:61616"/>
22. </bean>
24. <!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->
25. <bean id="connectionFactory" class="org.springframework.jms.connection.SingleConnectionFactory">
26. <!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->
27. <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
28. </bean>
30. <!--这个是队列目的地-->
31. <bean id="queueDestination" class="org.apache.activemq.command.ActiveMQQueue">
32. <constructor-arg>
33. <value>queue</value>
34. </constructor-arg>
35. </bean>
36. <!--这个是sessionAwareQueue目的地-->
37. <bean id="sessionAwareQueue" class="org.apache.activemq.command.ActiveMQQueue">
38. <constructor-arg>
39. <value>sessionAwareQueue</value>
40. </constructor-arg>
41. </bean>
42. <!-- 消息监听器 -->
43. <bean id="consumerMessageListener" class="com.tiantian.springintejms.listener.ConsumerMessageListener"/>
44. <!-- 可以获取session的MessageListener -->
45. <bean id="consumerSessionAwareMessageListener" class="com.tiantian.springintejms.listener.ConsumerSessionAwareMessageListener">
46. <property name="destination" ref="queueDestination"/>
47. </bean>
48. <!-- 消息监听容器 -->
49. <bean id="jmsContainer"        class="org.springframework.jms.listener.DefaultMessageListenerContainer">
50. <property name="connectionFactory" ref="connectionFactory" />
51. <property name="destination" ref="queueDestination" />
52. <property name="messageListener" ref="consumerMessageListener" />
53. </bean>
55. <bean id="sessionAwareListenerContainer"
56. class="org.springframework.jms.listener.DefaultMessageListenerContainer">
57. <property name="connectionFactory" ref="connectionFactory" />
58. <property name="destination" ref="sessionAwareQueue" />
59. <property name="messageListener" ref="consumerSessionAwareMessageListener" />
60. </bean>
61. </beans>

       接着我们来做一个测试，测试代码如下：

Java代码  [收藏代码](javascript:void())

1. @RunWith(SpringJUnit4ClassRunner.class)
2. @ContextConfiguration("/applicationContext.xml")
3. public class ProducerConsumerTest {
5. @Autowired
6. private ProducerService producerService;
7. @Autowired
8. @Qualifier("sessionAwareQueue")
9. private Destination sessionAwareQueue;
11. @Test
12. public void testSessionAwareMessageListener() {
13. producerService.sendMessage(sessionAwareQueue, "测试SessionAwareMessageListener");
14. }
16. }

       在上述测试代码中，我们通过前面定义好的生产者往我们定义好的SessionAwareMessageListener监听的sessionAwareQueue发送了一个消息。程序运行之后控制台输出如下：



       这说明我们已经成功的往sessionAwareQueue发送了一条纯文本消息，消息会被ConsumerSessionAwareMessageListener的onMessage方法进行处理，在onMessage方法中ConsumerSessionAwareMessageListener就是简单的把接收到的纯文本信息的内容打印出来了，之后再往queueDestination发送了一个纯文本消息，消息内容是“ConsumerSessionAwareMessageListener…”，该消息随后就被ConsumerMessageListener处理了，根据我们的定义，在ConsumerMessageListener中也只是简单的打印了一下接收到的消息内容。

**1.3.3  MessageListenerAdapter**

MessageListenerAdapter类实现了MessageListener接口和SessionAwareMessageListener接口，它的主要作用是将接收到的消息进行类型转换，然后通过反射的形式把它交给一个普通的Java类进行处理。

       MessageListenerAdapter会把接收到的消息做如下转换：

       TextMessage转换为String对象；

       BytesMessage转换为byte数组；

       MapMessage转换为Map对象；

       ObjectMessage转换为对应的Serializable对象。

       既然前面说了MessageListenerAdapter会把接收到的消息做一个类型转换，然后利用反射把它交给真正的目标处理器——一个普通的Java类进行处理（如果真正的目标处理器是一个MessageListener或者是一个SessionAwareMessageListener，那么Spring将直接使用接收到的Message对象作为参数调用它们的onMessage方法，而不会再利用反射去进行调用），那么我们在定义一个MessageListenerAdapter的时候就需要为它指定这样一个目标类。这个目标类我们可以通过MessageListenerAdapter的构造方法参数指定，如：

Xml代码  [收藏代码](javascript:void())

1. <!-- 消息监听适配器 -->
2. <bean id="messageListenerAdapter" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
3. <constructor-arg>
4. <bean class="com.tiantian.springintejms.listener.ConsumerListener"/>
5. </constructor-arg>
6. </bean>

       也可以通过它的delegate属性来指定，如：

Xml代码  [收藏代码](javascript:void())

1. <!-- 消息监听适配器 -->
2. <bean id="messageListenerAdapter" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
3. <property name="delegate">
4. <bean class="com.tiantian.springintejms.listener.ConsumerListener"/>
5. </property>
6. <property name="defaultListenerMethod" value="receiveMessage"/>
7. </bean>

       前面说了如果我们指定的这个目标处理器是一个MessageListener或者是一个SessionAwareMessageListener的时候Spring将直接利用接收到的Message对象作为方法参数调用它们的onMessage方法。但是如果指定的目标处理器是一个普通的Java类时Spring将利用Message进行了类型转换之后的对象作为参数通过反射去调用真正的目标处理器的处理方法，那么Spring是如何知道该调用哪个方法呢？这是通过MessageListenerAdapter的defaultListenerMethod属性来决定的，当我们没有指定该属性时，Spring会默认调用目标处理器的handleMessage方法。

       接下来我们来看一个示例，假设我们有一个普通的Java类ConsumerListener，其对应有两个方法，handleMessage和receiveMessage，其代码如下：

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.springintejms.listener;
3. public class ConsumerListener {
5. public void handleMessage(String message) {
6. System.out.println("ConsumerListener通过handleMessage接收到一个纯文本消息，消息内容是：" + message);
7. }
9. public void receiveMessage(String message) {
10. System.out.println("ConsumerListener通过receiveMessage接收到一个纯文本消息，消息内容是：" + message);
11. }
13. }

       假设我们要把它作为一个消息监听器来监听发送到adapterQueue的消息，这个时候我们就可以定义一个对应的MessageListenerAdapter来把它当做一个MessageListener使用。

Xml代码  [收藏代码](javascript:void())

1. <!-- 消息监听适配器 -->
2. <bean id="messageListenerAdapter" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
3. <property name="delegate">
4. <bean class="com.tiantian.springintejms.listener.ConsumerListener"/>
5. </property>
6. <property name="defaultListenerMethod" value="receiveMessage"/>
7. </bean>

       当然，有了MessageListener之后我们还需要配置其对应的MessageListenerContainer，这里配置如下：

Xml代码  [收藏代码](javascript:void())

1. <!-- 消息监听适配器对应的监听容器 -->
2. <bean id="messageListenerAdapterContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
3. <property name="connectionFactory" ref="connectionFactory"/>
4. <property name="destination" ref="adapterQueue"/>
5. <property name="messageListener" ref="messageListenerAdapter"/><!-- 使用MessageListenerAdapter来作为消息监听器 -->
6. </bean>

       在上面的MessageListenerAdapter中我们指定了其defaultListenerMethod属性的值为receiveMessage，所以当MessageListenerAdapter接收到消息之后会自动的调用我们指定的ConsumerListener的receiveMessage方法。

       针对于上述代码我们定义测试代码如下：

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.springintejms.test;
3. import javax.jms.Destination;
5. import org.junit.Test;
6. import org.junit.runner.RunWith;
7. import org.springframework.beans.factory.annotation.Autowired;
8. import org.springframework.beans.factory.annotation.Qualifier;
9. import org.springframework.test.context.ContextConfiguration;
10. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
12. import com.tiantian.springintejms.service.ProducerService;
14. @RunWith(SpringJUnit4ClassRunner.class)
15. @ContextConfiguration("/applicationContext.xml")
16. public class ProducerConsumerTest {
18. @Autowired
19. @Qualifier("adapterQueue")
20. private Destination adapterQueue;
22. @Test
23. public void testMessageListenerAdapter() {
24. producerService.sendMessage(adapterQueue, "测试MessageListenerAdapter");
25. }
27. }

       这时候我们会看到控制台输出如下：

http://dl2.iteye.com/upload/attachment/0086/1540/ed428c5f-4b65-31fa-bc6c-6b5940df5b99.png  
        如果我们不指定MessageListenerAdapter的defaultListenerMethod属性，那么在运行上述代码时控制台会输出如下结果：

http://dl2.iteye.com/upload/attachment/0086/1542/d0270ffa-b3e3-3405-8181-83e9cd80574b.png  
        MessageListenerAdapter除了会自动的把一个普通Java类当做MessageListener来处理接收到的消息之外，**其另外一个主要的功能是可以自动的发送返回消息**。

     当我们用于处理接收到的消息的方法的返回值不为空的时候，Spring会自动将它封装为一个JMS Message，然后自动进行回复。那么这个时候这个回复消息将发送到哪里呢？这主要有两种方式可以指定。  
       第一，可以通过发送的Message的setJMSReplyTo方法指定该消息对应的回复消息的目的地。这里我们把我们的生产者发送消息的代码做一下修改，在发送消息之前先指定该消息对应的回复目的地为一个叫responseQueue的队列目的地，具体代码如下所示：

Java代码  [收藏代码](javascript:void())

1. package com.tiantian.springintejms.service.impl;
3. import javax.jms.Destination;
4. import javax.jms.JMSException;
5. import javax.jms.Message;
6. import javax.jms.Session;
7. import javax.jms.TextMessage;
9. import org.springframework.beans.factory.annotation.Autowired;
10. import org.springframework.beans.factory.annotation.Qualifier;
11. import org.springframework.jms.core.JmsTemplate;
12. import org.springframework.jms.core.MessageCreator;
13. import org.springframework.stereotype.Component;
15. import com.tiantian.springintejms.service.ProducerService;
17. @Component
18. public class ProducerServiceImpl implements ProducerService {
20. @Autowired
21. private JmsTemplate jmsTemplate;
23. @Autowired
24. @Qualifier("responseQueue")
25. private Destination responseDestination;
27. public void sendMessage(Destination destination, final String message) {
28. System.out.println("---------------生产者发送消息-----------------");
29. System.out.println("---------------生产者发了一个消息：" + message);
30. jmsTemplate.send(destination, new MessageCreator() {
31. public Message createMessage(Session session) throws JMSException {
32. TextMessage textMessage = session.createTextMessage(message);
33. textMessage.setJMSReplyTo(responseDestination);
34. return textMessage;
35. }
36. });
37. }
39. }

       接着定义一个叫responseQueue的队列目的地及其对应的消息监听器和监听容器。

Xml代码  [收藏代码](javascript:void())

1. <!-- 用于测试消息回复的 -->
2. <bean id="responseQueue" class="org.apache.activemq.command.ActiveMQQueue">
3. <constructor-arg>
4. <value>responseQueue</value>
5. </constructor-arg>
6. </bean>
8. <!-- responseQueue对应的监听器 -->
9. <bean id="responseQueueListener" class="com.tiantian.springintejms.listener.ResponseQueueListener"/>
11. <!-- responseQueue对应的监听容器 -->
12. <bean id="responseQueueMessageListenerContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
13. <property name="connectionFactory" ref="connectionFactory"/>
14. <property name="destination" ref="responseQueue"/>
15. <property name="messageListener" ref="responseQueueListener"/>
16. </bean>

       ResponseQueueListener的定义如下所示：

Java代码  [收藏代码](javascript:void())

1. public class ResponseQueueListener implements MessageListener {
3. public void onMessage(Message message) {
4. if (message instanceof TextMessage) {
5. TextMessage textMessage = (TextMessage) message;
6. try {
7. System.out.println("接收到发送到responseQueue的一个文本消息，内容是：" + textMessage.getText());
8. } catch (JMSException e) {
9. e.printStackTrace();
10. }
11. }
12. }
14. }

        接着把我们接收消息的ConsumerListener的receiveMessage方法改为如下：

Java代码  [收藏代码](javascript:void())

1. /\*\*
2. \* 当返回类型是非null时MessageListenerAdapter会自动把返回值封装成一个Message，然后进行回复
3. \* @param message
4. \* @return
5. \*/
6. public String receiveMessage(String message) {
7. System.out.println("ConsumerListener通过receiveMessage接收到一个纯文本消息，消息内容是：" + message);
8. return "这是ConsumerListener对象的receiveMessage方法的返回值。";
9. }

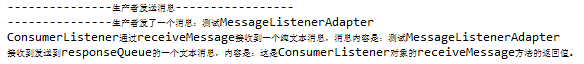
        我们可以看到在上述负责接收消息的receiveMessage方法有一个非空的返回值。

       接着我们运行我们的测试代码，利用生产者往我们定义好的MessageListenerAdapter负责处理的adapterQueue目的地发送一个消息。测试代码如下所示：

Java代码  [收藏代码](javascript:void())

1. @RunWith(SpringJUnit4ClassRunner.class)
2. @ContextConfiguration("/applicationContext.xml")
3. public class ProducerConsumerTest {
5. @Autowired
6. private ProducerService producerService;
8. @Qualifier("adapterQueue")
9. @Autowired
10. private Destination adapterQueue;
12. @Test
13. public void testMessageListenerAdapter() {
14. producerService.sendMessage(adapterQueue, "测试MessageListenerAdapter");
15. }
17. }

       运行上述测试代码之后，控制台输出如下：

  
        这说明我们的生产者发送消息被MessageListenerAdapter处理之后，MessageListenerAdapter确实把监听器的返回内容封装成一个Message往原Message通过setJMSReplyTo方法指定的回复目的地发送了一个消息。对于MessageListenerAdapter对应的监听器处理方法返回的是一个null值或者返回类型是void的情况，MessageListenerAdapter是不会自动进行消息的回复的，有兴趣的网友可以自己测试一下。

       第二，通过MessageListenerAdapter的defaultResponseDestination属性来指定。这里我们也来做一个测试，首先维持生产者发送消息的代码不变，即发送消息前不通过Message的setJMSReplyTo方法指定消息的回复目的地；接着我们在定义MessageListenerAdapter的时候通过其defaultResponseDestination属性指定其默认的回复目的地是“defaultResponseQueue”，并定义defaultResponseQueue对应的消息监听器和消息监听容器。

Xml代码  [收藏代码](javascript:void())

1. <!-- 消息监听适配器 -->
2. <bean id="messageListenerAdapter" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
3. <!-- <constructor-arg>
4. <bean class="com.tiantian.springintejms.listener.ConsumerListener"/>
5. </constructor-arg> -->
6. <property name="delegate">
7. <bean class="com.tiantian.springintejms.listener.ConsumerListener"/>
8. </property>
9. <property name="defaultListenerMethod" value="receiveMessage"/>
10. <property name="defaultResponseDestination" ref="defaultResponseQueue"/>
11. </bean>
13. <!-- 消息监听适配器对应的监听容器 -->
14. <bean id="messageListenerAdapterContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
15. <property name="connectionFactory" ref="connectionFactory"/>
16. <property name="destination" ref="adapterQueue"/>
17. <property name="messageListener" ref="messageListenerAdapter"/><!-- 使用MessageListenerAdapter来作为消息监听器 -->
18. </bean>
20. !-- 默认的消息回复队列 -->
21. <bean id="defaultResponseQueue" class="org.apache.activemq.command.ActiveMQQueue">
22. <constructor-arg>
23. <value>defaultResponseQueue</value>
24. </constructor-arg>
25. </bean>
27. <!-- defaultResponseQueue对应的监听器 -->
28. <bean id="defaultResponseQueueListener" class="com.tiantian.springintejms.listener.DefaultResponseQueueListener"/>
30. <!-- defaultResponseQueue对应的监听容器 -->
31. <bean id="defaultResponseQueueMessageListenerContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
32. <property name="connectionFactory" ref="connectionFactory"/>
33. <property name="destination" ref="defaultResponseQueue"/>
34. <property name="messageListener" ref="defaultResponseQueueListener"/>
35. </bean>

       DefaultResponseQueueListener的代码如下所示：

Java代码  [收藏代码](javascript:void())

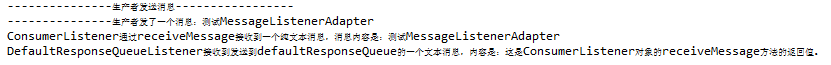
1. package com.tiantian.springintejms.listener;
3. import javax.jms.JMSException;
4. import javax.jms.Message;
5. import javax.jms.MessageListener;
6. import javax.jms.TextMessage;
8. public class DefaultResponseQueueListener implements MessageListener {
10. public void onMessage(Message message) {
11. if (message instanceof TextMessage) {
12. TextMessage textMessage = (TextMessage) message;
13. try {
14. System.out.println("DefaultResponseQueueListener接收到发送到defaultResponseQueue的一个文本消息，内容是：" + textMessage.getText());
15. } catch (JMSException e) {
16. e.printStackTrace();
17. }
18. }
19. }
21. }

       这时候运行如下测试代码：

Java代码  [收藏代码](javascript:void())

1. @Test
2. public void testMessageListenerAdapter() {
3. producerService.sendMessage(adapterQueue, "测试MessageListenerAdapter");
4. }

       控制台将输出如下内容：

  
        这说明MessageListenerAdapter会自动把真正的消息处理器返回的非空内容封装成一个Message发送回复消息到通过defaultResponseDestination属性指定的默认消息回复目的地。

       既然我们可以通过两种方式来指定MessageListenerAdapter自 动发送回复消息的目的地，那么当我们两种方式都指定了而且它们的目的地还不一样的时候会怎么发送呢？是两个都发还是只发其中的一个呢？关于这部分的测试我 这里就不赘述了，有兴趣的网友可以自己进行。这里我可以直接的告诉大家，当两种方式都指定了消息的回复目的地的时候使用发送消息的setJMSReplyTo方法指定的目的地将具有较高的优先级，MessageListenerAdapter将只往该方法指定的消息回复目的地发送回复消息。

**1.4     消息转换器MessageConverter**

MessageConverter的作用主要有两方面，一方面它可以把我们的非标准化Message对象转换成我们的目标Message对象，这主要是用在发送消息的时候；另一方面它又可以把我们的Message对象转换成对应的目标对象，这主要是用在接收消息的时候。

下面我们就拿发送一个对象消息来举例，假设我们有这样一个需求：我们平台有一个发送邮件的功能，进行发送的时候我们只是把我们的相关信息封装成一个JMS消息，然后利用JMS进行发送，在对应的消息监听器进行接收到的消息处理时才真正的进行消息发送。

假设我们有这么一个Email对象：

Java代码  [收藏代码](javascript:void())

1. public class Email implements Serializable {
3. private static final long serialVersionUID = -658250125732806493L;
5. private String receiver;
6. private String title;
7. private String content;
9. public Email(String receiver, String title, String content) {
10. this.receiver = receiver;
11. this.title = title;
12. this.content = content;
13. }
15. public String getReceiver() {
16. return receiver;
17. }
19. public void setReceiver(String receiver) {
20. this.receiver = receiver;
21. }
23. public String getTitle() {
24. return title;
25. }
27. public void setTitle(String title) {
28. this.title = title;
29. }
31. public String getContent() {
32. return content;
33. }
35. public void setContent(String content) {
36. this.content = content;
37. }
39. @Override
40. public String toString() {
41. StringBuilder builder = new StringBuilder();
42. builder.append("Email [receiver=").append(receiver).append(", title=")
43. .append(title).append(", content=").append(content).append("]");
44. return builder.toString();
45. }
47. }

       这个Email对象包含了一个简单的接收者email地址、邮件主题和邮件内容。我们在发送的时候就把这个对象封装成一个ObjectMessage进行发送。代码如下所示：

Java代码  [收藏代码](javascript:void())

1. public class ProducerServiceImpl implements ProducerService {
3. @Autowired
4. private JmsTemplate jmsTemplate;
6. public void sendMessage(Destination destination, final Serializable obj) {
7. jmsTemplate.send(destination, new MessageCreator() {
9. public Message createMessage(Session session) throws JMSException {
10. ObjectMessage objMessage = session.createObjectMessage(obj);
11. return objMessage;
12. }
14. });
15. }
17. }

       这是对应的在没有使用MessageConverter的时候我们需要new一个MessageCreator接口对象，然后在其抽象方法createMessage内部使用session创建一个对应的消息。在使用了MessageConverter的时候我们在使用JmsTemplate进行消息发送时只需要调用其对应的convertAndSend方法即可。如：

Java代码  [收藏代码](javascript:void())

1. public void sendMessage(Destination destination, final Serializable obj) {
2. //未使用MessageConverter的情况
3. /\*jmsTemplate.send(destination, new MessageCreator() {
5. public Message createMessage(Session session) throws JMSException {
6. ObjectMessage objMessage = session.createObjectMessage(obj);
7. return objMessage;
8. }
10. });\*/
11. //使用MessageConverter的情况
12. jmsTemplate.convertAndSend(destination, obj);
13. }

这样JmsTemplate就会在其内部调用预定的MessageConverter对我们的消息对象进行转换，然后再进行发送。

       这个时候我们就需要定义我们的MessageConverter了。要定义自己的MessageConverter很简单，只需要实现Spring为我们提供的MessageConverter接口即可。我们先来看一下MessageConverter接口的定义：

Java代码  [收藏代码](javascript:void())

1. public interface MessageConverter {
3. Message toMessage(Object object, Session session) throws JMSException, MessageConversionException;
5. Object fromMessage(Message message) throws JMSException, MessageConversionException;
7. }

       我们可以看到其中一共定义了两个方法fromMessage和toMessage，fromMessage是用来把一个JMS Message转换成对应的Java对象，而toMessage方法是用来把一个Java对象转换成对应的JMS Message。因为我们已经知道上面要发送的对象就是一个Email对象，所以在这里我们就简单地定义一个EmailMessageConverter用来把Email对象和对应的ObjectMessage进行转换，其代码如下：

Java代码  [收藏代码](javascript:void())

1. import javax.jms.JMSException;
2. import javax.jms.Message;
3. import javax.jms.ObjectMessage;
4. import javax.jms.Session;
6. import org.springframework.jms.support.converter.MessageConversionException;
7. import org.springframework.jms.support.converter.MessageConverter;
9. public class EmailMessageConverter implements MessageConverter {
11. public Message toMessage(Object object, Session session)
12. throws JMSException, MessageConversionException {
13. return session.createObjectMessage((Serializable) object);
14. }
16. public Object fromMessage(Message message) throws JMSException,
17. MessageConversionException {
18. ObjectMessage objMessage = (ObjectMessage) message;
19. return objMessage.getObject();
20. }
22. }

       这样当我们利用JmsTemplate的convertAndSend方法发送一个Email对象的时候就会把对应的Email对象当做参数调用我们定义好的EmailMessageConverter的toMessage方法。

       定义好我们的EmailMessageConverter之后就需要指定我们用来发送Email对象的JmsTemplate对象的messageConverter为EmailMessageConverter，这里我们在Spring的配置文件中定义JmsTemplate bean的时候就指定：

Xml代码  [收藏代码](javascript:void())

1. <!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->
2. <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
3. <!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->
4. <property name="connectionFactory" ref="connectionFactory"/>
5. <!-- 消息转换器 -->
6. <property name="messageConverter" ref="emailMessageConverter"/>
7. </bean>
8. <!-- 类型转换器 -->
9. <bean id="emailMessageConverter" class="com.tiantian.springintejms.converter.EmailMessageConverter"/>

       到此我们的MessageConverter就定义好了，也能够进行使用了，接着我们来进行测试一下，定义测试代码如下所示：

Java代码  [收藏代码](javascript:void())

1. @Test
2. public void testObjectMessage() {
3. Email email = new Email("zhangsan@xxx.com", "主题", "内容");
4. producerService.sendMessage(destination, email);
5. }

       上面destination对应的接收处理的MessageListener方法如下所示：

Java代码  [收藏代码](javascript:void())

1. public class ConsumerMessageListener implements MessageListener {
3. public void onMessage(Message message) {
5. if (message instanceof ObjectMessage) {
6. ObjectMessage objMessage = (ObjectMessage) message;
7. try {
8. Object obj = objMessage.getObject();
9. Email email = (Email) obj;
10. System.out.println("接收到一个ObjectMessage，包含Email对象。");
11. System.out.println(email);
12. } catch (JMSException e) {
13. e.printStackTrace();
14. }
15. }
16. }
18. }

       之前说了MessageConverter有两方面的功能，除了把Java对象转换成对应的Jms Message之外还可以把Jms Message转换成对应的Java对象。我们看上面的消息监听器在接收消息的时候接收到的就是一个Jms Message，如果我们要利用MessageConverter来把它转换成对应的Java对象的话，只能是我们往里面注入一个对应的MessageConverter，然后在里面手动的调用，如：

Java代码  [收藏代码](javascript:void())

1. public class ConsumerMessageListener implements MessageListener {
3. private MessageConverter messageConverter;
5. public void onMessage(Message message) {
7. if (message instanceof ObjectMessage) {
8. ObjectMessage objMessage = (ObjectMessage) message;
9. try {
10. /\*Object obj = objMessage.getObject();
11. Email email = (Email) obj;\*/
12. Email email = (Email) messageConverter.fromMessage(objMessage);
13. System.out.println("接收到一个ObjectMessage，包含Email对象。");
14. System.out.println(email);
15. } catch (JMSException e) {
16. e.printStackTrace();
17. }
19. }
20. }
22. public MessageConverter getMessageConverter() {
23. return messageConverter;
24. }
26. public void setMessageConverter(MessageConverter messageConverter) {
27. this.messageConverter = messageConverter;
28. }
30. }

       当我们使用MessageListenerAdapter来作为消息监听器的时候，我们可以为它指定一个对应的MessageConverter，这样Spring在处理接收到的消息的时候就会自动地利用我们指定的MessageConverter对它进行转换，然后把转换后的Java对象作为参数调用指定的消息处理方法。这里我们再把前面讲解MessageListenerAdapter时定义的MessageListenerAdapter拿来做一个测试，我们指定它的MessageConverter为我们定义好的EmailMessageConverter。

Xml代码  [收藏代码](javascript:void())

1. <!-- 消息监听适配器 -->
2. <bean id="messageListenerAdapter" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
3. <property name="delegate">
4. <bean class="com.tiantian.springintejms.listener.ConsumerListener"/>
5. </property>
6. <property name="defaultListenerMethod" value="receiveMessage"/>
7. <property name="messageConverter" ref="emailMessageConverter"/>
8. </bean>
9. <!-- 消息监听适配器对应的监听容器 -->
10. <bean id="messageListenerAdapterContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
11. <property name="connectionFactory" ref="connectionFactory"/>
12. <property name="destination" ref="adapterQueue"/>
13. <property name="messageListener" ref="messageListenerAdapter"/><!-- 使用MessageListenerAdapter来作为消息监听器 -->
14. </bean>

       然后在我们的真正用于处理接收到的消息的ConsumerListener中添加一个receiveMessage方法，添加后其代码如下所示：

Java代码  [收藏代码](javascript:void())

1. public class ConsumerListener {
3. public void receiveMessage(String message) {
4. System.out.println("ConsumerListener通过receiveMessage接收到一个纯文本消息，消息内容是：" + message);
5. }
7. public void receiveMessage(Email email) {
8. System.out.println("接收到一个包含Email的ObjectMessage。");
9. System.out.println(email);
10. }
12. }

       然后我们定义如下测试代码：

Java代码  [收藏代码](javascript:void())

1. @Test
2. public void testObjectMessage() {
3. Email email = new Email("zhangsan@xxx.com", "主题", "内容");
4. producerService.sendMessage(adapterQueue, email);
5. }

       因为我们给MessageListenerAdapter指定了一个MessageConverter，而且是一个EmailMessageConverter，所以当MessageListenerAdapter接收到一个消息后，它会调用我们指定的MessageConverter的fromMessage方法把它转换成一个Java对象，根据定义这里会转换成一个Email对象，然后会把这个Email对象作为参数调用我们通过defaultListenerMethod属性指定的默认处理器方法，根据定义这里就是receiveMessage方法，但是我们可以看到在ConsumerListener中我们一共定义了两个receiveMessage方法，因为是通过转换后的Email对象作为参数进行方法调用的，所以这里调用的就应该是参数类型为Email的receiveMessage方法了。上述测试代码运行后会输出如下结果：  
http://dl2.iteye.com/upload/attachment/0086/6469/ee3a2dc6-82ce-3d4c-a7a5-30f5adca1499.png  
        说到这里可能有读者就会有疑问了，说我们在之前讲解MessageListenerAdapter的时候不是没有指定对应的MessageConverter，然后发送了一个TextMessage，结果Spring还是把它转换成一个String对象，调用了ConsumerListener参数类型为String的receiveMessage方法吗？那你这个MessageConverter在MessageListenerAdapter进行消息接收的时候也没什么用啊。

       其实还是有用的，在我们使用MessageListenerAdapter时，在对其进行初始化也就是调用其构造方法时，它会默认new一个Spring已经为我们实现了的MessageConverter——SimpleMessageConverter作为其默认的MessageConverter，这也就是为什么我们在使用MessageListenerAdapter的时候不需要指定MessageConverter但是消息还是会转换成对应的Java对象的原因。所以默认情况下我们使用MessageListenerAdapter时其对应的MessageListener的处理器方法参数类型必须是一个普通Java对象，而不能是对应的Jms Message对象。

       那如果我们在处理Jms Message的时候想使用MessageListenerAdapter，然后又希望处理最原始的Message，而不是经过MessageConverter进行转换后的Message该怎么办呢？这个时候我们只需要在定义MessageListenerAdapter的时候指定其MessageConverter为空就可以了。

Xml代码  [收藏代码](javascript:void())

1. <!-- 消息监听适配器 -->
2. <bean id="messageListenerAdapter" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
3. <property name="delegate">
4. <bean class="com.tiantian.springintejms.listener.ConsumerListener"/>
5. </property>
6. <property name="defaultListenerMethod" value="receiveMessage"/>
7. <property name="messageConverter">
8. <null/>
9. </property>
10. </bean>

       那么这个时候我们的真实MessageListener的处理器方法参数类型就应该是Jms Message或对应的Jms Message子类型了，不然就会调用不到对应的处理方法了。这里因为我们发送的是一个ObjectMessage，所以这里就添加一个对应的参数类型为ObjectMessage的receiveMessage方法了。

Java代码  [收藏代码](javascript:void())

1. public void receiveMessage(ObjectMessage message) throws JMSException {
2. System.out.println(message.getObject());
3. }

       刚刚讲到Spring已经为我们实现了一个简单的MessageConverter，即org.springframework.jms.support.converter.SimpleMessageConverter，其实Spring在初始化JmsTemplate的时候也指定了其对应的MessageConverter为一个SimpleMessageConverter，所以如果我们平常没有什么特殊要求的时候可以直接使用JmsTemplate的convertAndSend系列方法进行消息发送，而不必繁琐的在调用send方法时自己new一个MessageCreator进行相应Message的创建。

这里我们也来看一下SimpleMessageConverter的定义，如果觉得它不能满足你的要求，那我们可以对它里面的部分方法进行重写，或者是完全实现自己的MessageConverter。

Java代码  [收藏代码](javascript:void())

1. public class SimpleMessageConverter implements MessageConverter {
3. public Message toMessage(Object object, Session session) throws JMSException, MessageConversionException {
4. if (object instanceof Message) {
5. return (Message) object;
6. }
7. else if (object instanceof String) {
8. return createMessageForString((String) object, session);
9. }
10. else if (object instanceof byte[]) {
11. return createMessageForByteArray((byte[]) object, session);
12. }
13. else if (object instanceof Map) {
14. return createMessageForMap((Map) object, session);
15. }
16. else if (object instanceof Serializable) {
17. return createMessageForSerializable(((Serializable) object), session);
18. }
20. else {
21. throw new MessageConversionException("Cannot convert object of type [" +
22. ObjectUtils.nullSafeClassName(object) + "] to JMS message. Supported message " +
23. "payloads are: String, byte array, Map<String,?>, Serializable object.");
24. }
25. }
27. public Object fromMessage(Message message) throws JMSException, MessageConversionException {
28. if (message instanceof TextMessage) {
29. return extractStringFromMessage((TextMessage) message);
30. }
31. else if (message instanceof BytesMessage) {
32. return extractByteArrayFromMessage((BytesMessage) message);
33. }
34. else if (message instanceof MapMessage) {
35. return extractMapFromMessage((MapMessage) message);
36. }
37. else if (message instanceof ObjectMessage) {
38. return extractSerializableFromMessage((ObjectMessage) message);
39. }
40. else {
41. return message;
42. }
43. }
45. protected TextMessage createMessageForString(String text, Session session) throws JMSException {
46. return session.createTextMessage(text);
47. }
49. protected BytesMessage createMessageForByteArray(byte[] bytes, Session session) throws JMSException {
50. BytesMessage message = session.createBytesMessage();
51. message.writeBytes(bytes);
52. return message;
53. }
55. protected MapMessage createMessageForMap(Map<?, ?> map, Session session) throws JMSException {
56. MapMessage message = session.createMapMessage();
57. for (Map.Entry entry : map.entrySet()) {
58. if (!(entry.getKey() instanceof String)) {
59. throw new MessageConversionException("Cannot convert non-String key of type [" +
60. ObjectUtils.nullSafeClassName(entry.getKey()) + "] to JMS MapMessage entry");
61. }
62. message.setObject((String) entry.getKey(), entry.getValue());
63. }
64. return message;
65. }
67. protected ObjectMessage createMessageForSerializable(Serializable object, Session session) throws JMSException {
68. return session.createObjectMessage(object);
69. }

72. protected String extractStringFromMessage(TextMessage message) throws JMSException {
73. return message.getText();
74. }
76. protected byte[] extractByteArrayFromMessage(BytesMessage message) throws JMSException {
77. byte[] bytes = new byte[(int) message.getBodyLength()];
78. message.readBytes(bytes);
79. return bytes;
80. }
82. protected Map extractMapFromMessage(MapMessage message) throws JMSException {
83. Map<String, Object> map = new HashMap<String, Object>();
84. Enumeration en = message.getMapNames();
85. while (en.hasMoreElements()) {
86. String key = (String) en.nextElement();
87. map.put(key, message.getObject(key));
88. }
89. return map;
90. }
92. protected Serializable extractSerializableFromMessage(ObjectMessage message) throws JMSException {
93. return message.getObject();
94. }
96. }

Spring提供了一个JmsTransactionManager用于对JMS ConnectionFactory做事务管理。这将允许JMS应用利用Spring的事务管理特性。JmsTransactionManager在执行本地资源事务管理时将从指定的ConnectionFactory绑定一个ConnectionFactory/Session这样的配对到线程中。JmsTemplate会自动检测这样的事务资源，并对它们进行相应操作。

在Java EE环境中，ConnectionFactory会池化Connection和Session，这样这些资源将会在整个事务中被有效地重复利用。在一个独立的环境中，使用Spring的SingleConnectionFactory时所有的事务将公用一个Connection，但是每个事务将保留自己独立的Session。

JmsTemplate可以利用JtaTransactionManager和能够进行分布式的 JMS ConnectionFactory处理分布式事务。

       在Spring整合JMS的应用中，如果我们要进行本地的事务管理的话非常简单，只需要在定义对应的消息监听容器时指定其sessionTransacted属性为true，如：

Xml代码  [收藏代码](javascript:void())

1. <bean id="jmsContainer"
2. class="org.springframework.jms.listener.DefaultMessageListenerContainer">
3. <property name="connectionFactory" ref="connectionFactory" />
4. <property name="destination" ref="queueDestination" />
5. <property name="messageListener" ref="consumerMessageListener" />
6. <property name="sessionTransacted" value="true"/>
7. </bean>

       该属性值默认为false，这样JMS在进行消息监听的时候就会进行事务控制，当在接收消息时监听器执行失败时JMS就会对接收到的消息进行回滚，对于SessionAwareMessageListener在接收到消息后发送一个返回消息时也处于同一事务下，但是对于其他操作如数据库访问等将不属于该事务控制。

这里我们可以来做一个这样的测试：我们如上配置监听在queueDestination的消息监听容器的sessionTransacted属性为true，然后把我们前面提到的消息监听器ConsumerMessageListener改成这样：

Java代码  [收藏代码](javascript:void())

1. public class ConsumerMessageListener implements MessageListener {
3. public void onMessage(Message message) {
4. //这里我们知道生产者发送的就是一个纯文本消息，所以这里可以直接进行强制转换，或者直接把onMessage方法的参数改成Message的子类TextMessage
5. TextMessage textMsg = (TextMessage) message;
6. System.out.println("接收到一个纯文本消息。");
7. try {
8. System.out.println("消息内容是：" + textMsg.getText());
9. if (1 == 1) {
10. throw new RuntimeException("Error");
11. }
12. } catch (JMSException e) {
13. e.printStackTrace();
14. }
15. }
17. }

我们可以看到在上述代码中我们的ConsumerMessageListener在进行消息接收的时候抛出了一个RuntimeException，根据我们上面说的，因为我们已经在对应的监听容器上定义了其sessionTransacted属性为true，所以当这里抛出异常的时候JMS将对接收到的消息进行回滚，即下次进行消息接收的时候该消息仍然能够被接收到。为了验证这一点，我们先执行一遍测试代码，往queueDestination发送一个文本消息，这个时候ConsumerMessageListener在进行接收的时候将会抛出一个RuntimeException，已经接收到的纯文本消息将进行回滚；接着我们去掉上面代码中抛出异常的语句，即ConsumerMessageListener能够正常的进行消息接收，这个时候我们再运行一次测试代码，往ConsumerMessageListener监听的queueDestination发送一条消息。如果之前在接手时抛出了异常的那条消息已经回滚了的话，那么这个时候将能够接收到两条消息，控制台将输出接收到的两条消息的内容。具体结果有兴趣的朋友可以自己验证一下。

       如果想接收消息和数据库访问处于同一事务中，那么我们就可以配置一个外部的事务管理同时配置一个支持外部事务管理的消息监听容器（如DefaultMessageListenerContainer）。要配置这样一个参与分布式事务管理的消息监听容器，我们可以配置一个JtaTransactionManager，当然底层的JMS ConnectionFactory需要能够支持分布式事务管理，并正确地注册我们的JtaTransactionManager。这样消息监听器进行消息接收和对应的数据库访问就会处于同一数据库控制下，当消息接收失败或数据库访问失败都会进行事务回滚操作。

Xml代码  [收藏代码](javascript:void())

1. <bean id="jmsContainer"
2. class="org.springframework.jms.listener.DefaultMessageListenerContainer">
3. <property name="connectionFactory" ref="connectionFactory" />
4. <property name="destination" ref="queueDestination" />
5. <property name="messageListener" ref="consumerMessageListener" />
6. <property name="transactionManager" ref="jtaTransactionManager"/>
7. </bean>
9. <bean id="jtaTransactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

当给消息监听容器指定了transactionManager时，消息监听容器将忽略sessionTransacted的值。

       关于使用JtaTransactionManager来管理上述分布式事务，我们这里也可以来做一个试验。

       首先：往Spring配置文件applicationContext.xml中添加如下配置：

Xml代码  [收藏代码](javascript:void())

1. <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
2. <property name="dataSource" ref="dataSource"/>
3. </bean>
5. <jee:jndi-lookup jndi-name="jdbc/mysql" id="dataSource"/>
6. <bean id="jtaTransactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
8. <tx:annotation-driven transaction-manager="jtaTransactionManager"/>

       我们可以看到，在这里我们引入了一个jndi数据源，定义了一个JtaTransactionManager，定义了Spring基于注解的声明式事务管理，定义了一个Spring提供的进行Jdbc操作的工具类jdbcTemplate。

       接下来把我们的ConsumerMessageListener改为如下形式：

Java代码  [收藏代码](javascript:void())

1. public class ConsumerMessageListener implements MessageListener {
3. @Autowired
4. private TestDao testDao;
6. private int count = 0;
8. public void onMessage(Message message) {
9. //这里我们知道生产者发送的就是一个纯文本消息，所以这里可以直接进行强制转换，或者直接把onMessage方法的参数改成Message的子类TextMessage
10. TextMessage textMsg = (TextMessage) message;
11. System.out.println(new Date().toLocaleString() + "接收到一个纯文本消息。");
12. try {
13. String text = textMsg.getText();
14. System.out.println("消息内容是：" + text);
15. System.out.println("当前count的值是：" + count);
16. testDao.insert(text + count);
17. if (count == 0) {
18. count ++;
19. throw new RuntimeException("Error! 出错啦！");
20. }
21. } catch (JMSException e) {
22. e.printStackTrace();
23. }
24. }
26. }

       我们可以看到，在ConsumerMessageListener中我们定义了一个实例变量count，其初始值为0；在onMessage里面，我们可以看到我们把接收到的消息内容作为参数调用了testDao的insert方法；当count值为0，也就是进行第一次消息接收的时候会将count的值加1，同时抛出一个运行时异常。那么我们这里要测试的就是进行第一次接收的时候testDao已经把相关内容插入数据库了，接着在onMessage里面抛出了一个异常同时count加1，我们预期的结果应该是此时数据库进行回滚，同时JMS也回滚，这样JMS将继续尝试接收该消息，此时同样会调用testDao的insert方法将内容插入数据库，再接着count已经不为0了，所以此时将不再抛出异常，JMS成功进行消息的接收，testDao也成功的将消息内容插入到了数据库。要证明这个预期我们除了看数据库中插入的数据外，还可以看控制台的输出，正常情况控制台将输出两次消息接收的内容，且第一次时count为0，第二次count为1。

       TestDao是一个接口，其TestDaoImpl对insert的方法实现如下：

Java代码  [收藏代码](javascript:void())

1. @Transactional(readOnly=false)
2. public void insert(final String name) {
4. jdbcTemplate.update("insert into test(name) values(?)", name);
6. }

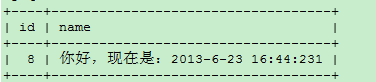
这里我们使用支持JtaTransactionManager的Weblogic来进行测试，因为是Web容器，所以我们这里定义了一个Controller来进行消息的发送，具体代码如下：

Java代码  [收藏代码](javascript:void())

1. @Controller
2. @RequestMapping("test")
3. public class TestController {
5. @Autowired
6. @Qualifier("queueDestination")
7. private Destination destination;
9. @Autowired
10. private ProducerService producerService;
12. @RequestMapping("first")
13. public String first() {
14. producerService.sendMessage(destination, "你好，现在是：" + new Date().toLocaleString());
15. return "/test/first";
16. }
18. }

接下来就是启用Weblogic服务器，进入其控制台，定义一个名叫“jdbc/mysql”的JNDI数据源，然后把该项目部署到Weblogic服务器上并进行启动。接下来我们就可以访问/test/first.do访问到上述first方法。之后控制台会输出如下信息：

        我们可以看到当count为0时接收了一次，并随后抛出了异常，之后count为1又接收了一次，这说明在count为0时抛出异常后我们的JMS进行回滚了，那么我们的数据库是否有进行回滚呢？接着我们来看数据库中的内容：

  
        我们可以看到数据库表中只有一条记录，而且最后一位表示count的值的为1，这说明在JMS进行消息接收抛出异常时我们的数据库也回滚了。关于使用JtaTransactionManager进行分布式事务管理的问题就说到这里了，有兴趣的朋友可以自己试验一下。