[**Spring整合JMS（四）——事务管理**](http://elim.iteye.com/blog/1983532)

**博客分类：**

* [Spring](http://elim.iteye.com/category/156657)
* [Jms](http://elim.iteye.com/category/347649)

[Spring](http://www.iteye.com/blogs/tag/Spring)[JMS](http://www.iteye.com/blogs/tag/JMS)[事务](http://www.iteye.com/blogs/tag/%E4%BA%8B%E5%8A%A1)[sessionTransacted](http://www.iteye.com/blogs/tag/sessionTransacted)[JtaTransactionManager](http://www.iteye.com/blogs/tag/JtaTransactionManager)

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. }

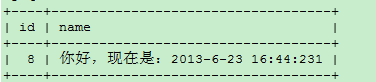
这里我们使用支持JtaTransactionManager的Weblogic来进行测试，因为是Web容器，所以我们这里定义了一个Controller来进行消息的发送，具体代码如下：

**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。

* [Spring整合JMS——基于ActiveMQ实现（一）](http://elim.iteye.com/blog/1893038)
* [Spring整合JMS(二)——三种消息监听器](http://elim.iteye.com/blog/1893676)
* [Spring整合JMS（三）——MessageConverter介绍](http://elim.iteye.com/blog/1900937)