



RCloud MQ ® V6

产品用户开发手册

(RCloud MQ Version 6.0)

北京中软国际信息技术有限公司

2014 年

目 录

第一章 概述.....	3
1.1 面向读者	3
1.2 开发准备	3
第二章 JMS 规范	3
2.1 JMS 的基本构件.....	3
2.1.1 连接工厂.....	3
2.1.2 连接.....	4
2.1.3 会话.....	4
2.1.4 目的地.....	4
2.1.5 消息生产者.....	5
2.1.6 消息消费者.....	5
2.1.7 消息.....	5
2.2 JMS 的可靠性机制.....	6
2.2.1 消息确认.....	6
2.2.2 消息持久化.....	6
2.2.3 优先级.....	7
2.2.4 消息过期.....	7
2.2.5 临时目的地.....	7
2.2.6 持久订阅.....	7
2.2.7 本地事务.....	8
2.3 JMS 规范的变迁.....	8
第三章 开发指导.....	9
3.1 JMS 开发流程.....	9
3.1.1 JMS 系统架构.....	9
3.1.2 点对点工作流程.....	10
3.1.3 发布订阅工作流程.....	11

3.1.4 JMS 开发步骤.....	12
3.1.5 JMS 开发示例.....	12
3.2 JMS 消息头介绍	15
3.3 传送大消息.....	16
3.4 设定消息优先级	17
3.5 集群模式下消息收发.....	19
3.6 消息转发	20
3.7 流量控制	22
3.7.1 消费速度控制.....	22
3.7.2 生产速度控制.....	24
第四章 开发建议.....	25
4.1 避免违背设计模式	25
4.2 避免使用繁琐的消息格式.....	26
4.3 不要为每个请求都创建新的临时队列	26
4.4 尽量不要使用 MDB	26

第一章 概述

1.1 面向读者

本手册详细描述了 RCloudMQ（以下简称 MQ）的客户端应用开发过程及核心 API 的使用方法，主要面向基于该产品的客户端产品开发人员、实施部署人员。

1.2 开发准备

在开始进行 JMS 客户端应用开发之前，需要参照该产品用户使用书册，使其满足下列先决条件：

- 1、完成 RCloudMQ 服务器的安装，有个一正常运行的服务端。
- 2、对 RCloudMQ 服务器按照业务需要进行配置。

提示：

因为 RCloudMQ 服务器的大多功能都是通过配置来保障，关于服务器端的配置，产品使用手册中有详细的说明，因此为了更好的进行开发，您需要配合产品使用手册中的内容。

第二章 JMS 规范

RCloudMQ 实现了 JMS1.1 和 JMS2.0 的规范，因此为了进行 JMS 的客户端开发，您需要全面掌握 JMS 相关的规范内容。本手册也不再涉及介绍使用 JMS 的接口进行开发的示例。

2.1 JMS 的基本构件

2.1.1 连接工厂

连接工厂是客户用来创建连接的对象，例如 RCloudMQ 提供的
RCloudMQConnectionFactory。

2.1.2 连接

JMS Connection 封装了客户与 JMS 提供者之间的一个虚拟的连接。

2.1.3 会话

JMS Session 是生产和消费消息的一个单线程上下文。会话用于创建消息生产者

者（producer）、消息消费者（consumer）和消息（message）等。会话提供了一个事务性的上下文，在这个上下文中，一组发送和接收被组合到了一个原子操作中。

2.1.4 目的地

目的地是客户用来指定它生产的消息的目标和它消费的消息的来源的对象。

JMS1.0.2 规范中定义了两种消息传递域：点对点（PTP）消息传递域和发布/订阅消息传递域。

点对点消息传递域的特点如下：

- 每个消息只能有一个消费者。
- 消息的生产者和消费者之间没有时间上的相关性。

无论消费者在生产者发送消息的时候是否处于运行状态，它都可以提取消息。

发布/订阅消息传递域的特点如下：

- 每个消息可以有多个消费者。

生产者和消费者之间有时间上的相关性。

- 订阅一个主题的消费者只能消费。

自它订阅之后发布的消息。JMS 规范允许客户创建持久订阅，这在一定程度上放松了时间上的相关性要求。持久订阅允许消费者消费它在未处于激活状态时发送的消息。

在点对点消息传递域中，目的地被成为队列（queue）；在发布/订阅消息传

递域中，目的地被成为主题（topic）。

2.1.5 消息生产者

消息生产者是由会话创建的一个对象，用于把消息发送到一个目的地。

2.1.6 消息消费者

消息消费者是由会话创建的一个对象，它用于接收发送到目的地的消息。消息的消费可以采用以下两种方法之一：

- 同步消费。

通过调用 消费者的 `receive` 方法从目的地中显式提取消息。`receive` 方法可以一直阻塞到消息到达。

- 异步消费。

客户可以为消费者注册一个消息监听器，以定义在消息到达时所采取的动作。

2.1.7 消息

JMS 消息由以下三部分组成：

- 消息头。

每个消息头字段都有相应的 `getter` 和 `setter` 方法。

- 消息属性。

如果需要除消息头字段以外的值，那么可以使用消息属性。

- 消息体。

JMS 定义的消息类型有 `TextMessage`、`MapMessage`、`BytesMessage`、`StreamMessage` 和 `ObjectMessage`。

2.2 JMS 的可靠性机制

2.2.1 消息确认

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：

- `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的确认消息的提交。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`。

2.2.2 消息持久化

JMS 支持以下两种消息提交模式：

- `PERSISTENT`。

指示 JMS provider 持久保存消息，以保证消息不会因为 JMSprovider 的失败而丢失。

- NON_PERSISTENT。

不要求 JMS provider 持久保存消息。

2.2.3 优先级

可以使用消息优先级来指示 JMS provider 首先提交紧急的消息。优先级分 10 个级别，从 0（最低）到 9（最高）。如果不指定优先级，默认级别是 4。需要注意的是，JMS provider 并不一定保证按照优先级的顺序提交消息。

2.2.4 消息过期

可以设置消息在一定时间后过期，默认是永不过期。

2.2.5 临时目的地

可以通过会话上的 `createTemporaryQueue` 方法和 `createTemporaryTopic` 方法来创建临时目的地。它们的存在时间只限于创建它们的连接所保持的时间。只有创建该临时目的地的连接上的消息消费者才能够从临时目的地中提取消息。

2.2.6 持久订阅

首先消息生产者必须使用 `PERSISTENT` 提交消息。客户可以通过会话上的 `createDurableSubscriber` 方法来创建一个持久订阅，该方法的第一个参数必须是一个 topic。第二个参数是订阅的名称。JMS provider 会存储发布到持久订阅对应的 topic 上的消息。如果最初创建持久订阅的客户或者任何其它客户使用相同的连接工厂和连接的客户 ID、相同的主题和相同的订阅名再次调用会话上的 `createDurableSubscriber` 方法，那么该持久订阅就会被激活。JMS provider 会向客户发送客户处于非激活状态时所发布的消息。

持久订阅在某个时刻只能有一个激活的订阅者。持久订阅在创建之后会一直保留，直到应用程序调用会话上的 `unsubscribe` 方法。

2.2.7 本地事务

在一个 JMS 客户端，可以使用本地事务来组合消息的发送和接收。JMS Session 接口提供了 `commit` 和 `rollback` 方法。事务提交意味着生产的所有消息被发送，消费的所有消息被确认；事务回滚意味着生产的所有消息被销毁，消费的所有消息被恢复并重新提交，除非它们已经过期。

事务性的会话总是牵涉到事务处理中，`commit` 或 `rollback` 方法一旦被调用，一个事务就结束了，而另一个事务被开始。关闭事务性会话将回滚其中的事务。需要注意的是，如果使用请求/回复机制，即发送一个消息，同时希望在同一个事务中等待接收该消息的回复，那么程序将被挂起，因为知道事务提交，发送操作才会真正执行。

需要注意的还有一个，消息的生产和消费不能包含在同一个事务中。

2.3 JMS 规范的变迁

JMS 的最新版本的是 2.0。它同 1.1 版本之间最大的差别是，JMS2.0 通过统一的消息传递域简化了消息传递。这不仅简化了 JMS API，也有利于开发人员灵活选择消息传递域，同时也有助于程序的重用和维护。

下表是不同消息传递域的相应接口：

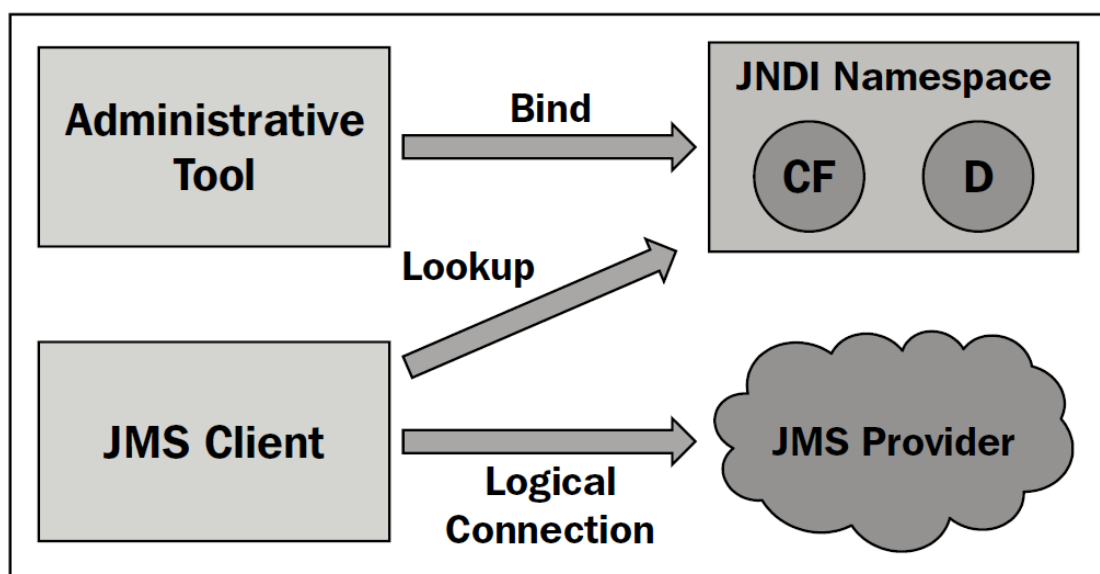
JMS 公共	点对点域	发布/订阅域
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

第三章 开发指导

3.1 JMS 开发流程

3.1.1 JMS 系统架构

RCloudMQ 一般在分布式系统中使用，因此了解整个 JMS 的架构可以从全局指导客户端应用的开发。



整个系统分为四部分：JMS 服务端、JMS 客户端、JNDI 服务和管理平台。

首先，JMS 服务端发布消息服务；

其次，在 JNDI 服务上注册 JMS 服务信息；

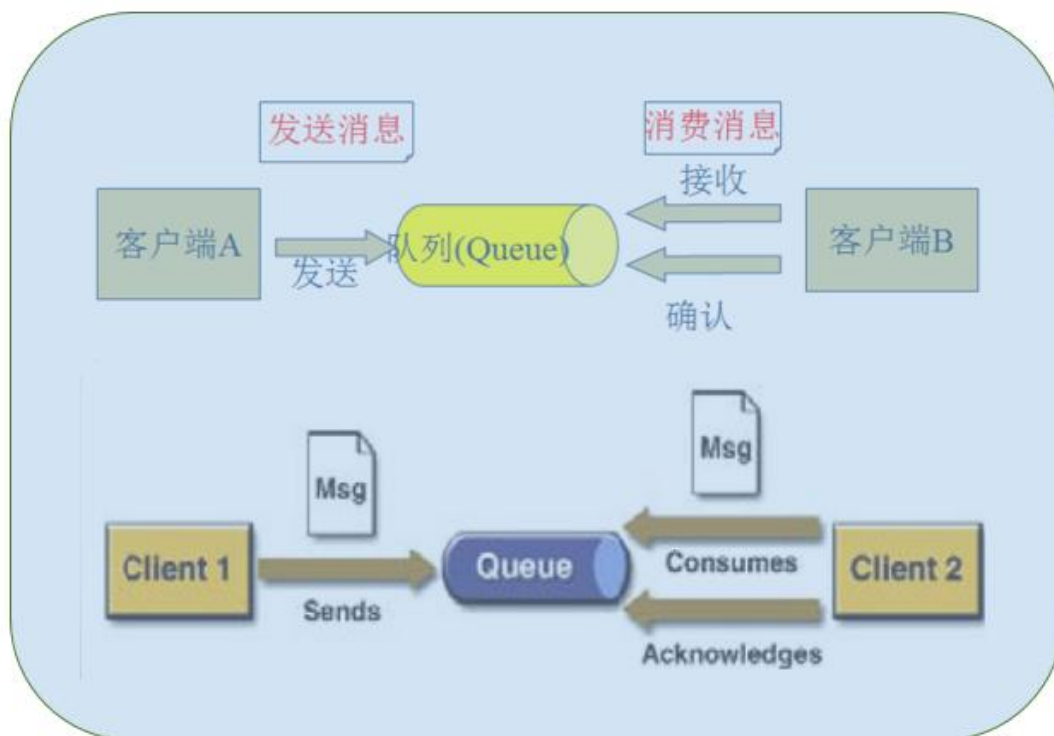
接着，JMS 客户端在 JNDI 上查找到 JMS 的连接信息；

最后，JMS 客户端使用连接信息和 JMS 服务器进行消息通信。

在整个架构中 JMS 客户端和 JMS 服务器完全独立，降低了整个系统的耦合性。

3.1.2 点对点工作流程

点对点是 JMS 的主要模型之一，只有一个消费者和生产者进行消息的交换。一般应用中都会使用该模式，下图是该模式下消息的流转过程。



首先客户端 1 向队列 Queue 发送消息；
接着客户端 2 从队列 Queue 接收到消息；
最后客户端 2 反馈一个消息接收的应答。

下面的 JMS 对象在点对点消息模式中是必须的：

- a. 队列(Queue) - 一个提供者命名的队列对象，客户端将会使用这个命名的队列对象。
- b. 队列链接工厂(QueueConnectionFactory) - 客户端使用队列链接工厂创建链接队列 ConnectionQueue 来取得与 JMS 点对点消息提供者的链接。
- c. 链接队列(ConnectionQueue) - 一个活动的链接队列存在在客户端与点对点消息提供者之间，客户用它创建一个或者多个 JMS 队列会话(QueueSession)。
- d. 队列会话(QueueSession) - 用来创建队列消息的发送者与接受者

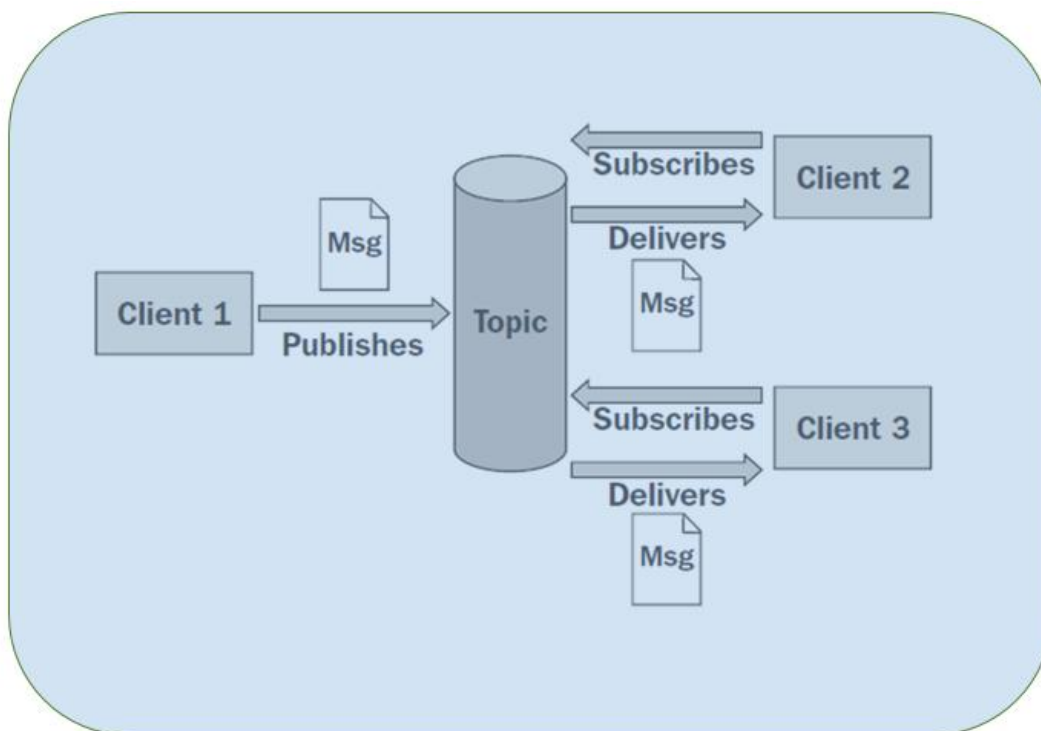
(QueueSender and QueueReceiver)。

e.消息发送者(QueueSender 或者 MessageProducer) - 发送消息到已经声明的队列。

f.消息接受者(QueueReceiver 或者 MessageConsumer) - 接受已经被发送到指定队列的消息。

3.1.3 发布订阅工作流程

发布订阅模式下,消费者可以先对感兴趣的主题进行订阅,生产者有消息后,通过该模式向订阅的消费者分发消息。



下面是在发布订阅模式中必须的对象：

a.主题 Topic(Destination) - 一个提供者命名的主题对象，客户端将会使用这个命名的主题对象。

b.主题链接工厂(TopciConnectionFactory) - 客户端使用主题链接工厂创建链接主题 ConnectionTopic 来取得与 JMS 消息 Pub/Sub 提供者的链接。

c.链接主题(ConnectionTopic) - 一个活动的链接主题存在发布者与订阅者之间。

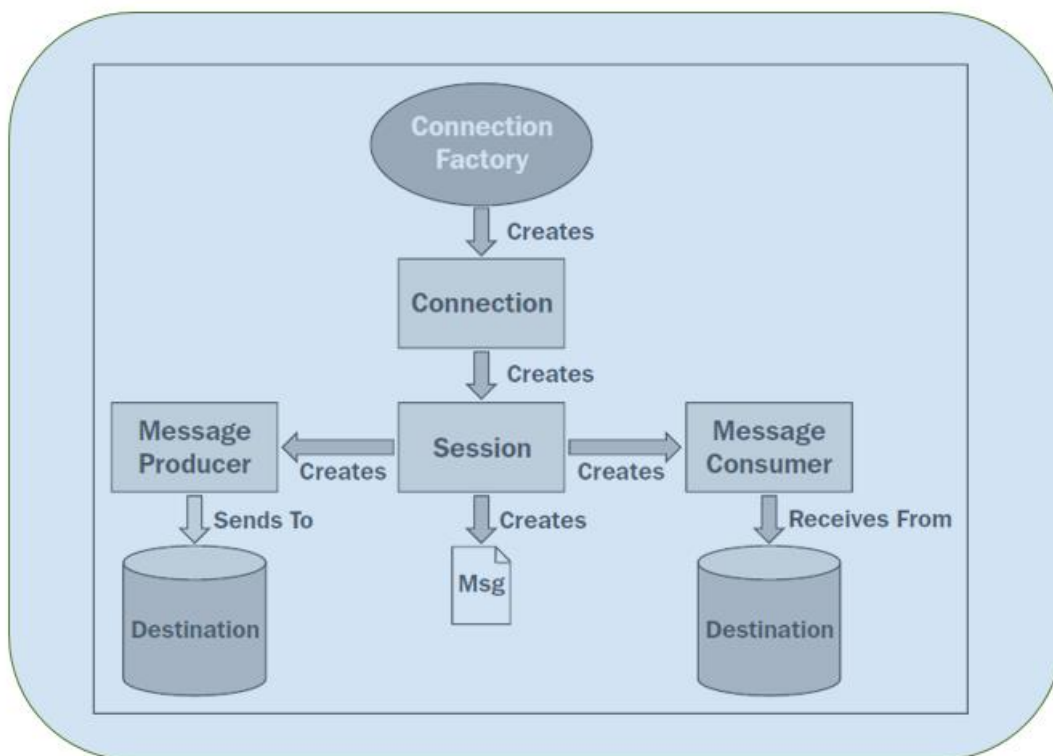
d 会话 (TopicSession) - 用来创建主题消息的发布者与订阅者 (TopicPublisher and TopicSubscribers)。

e.消息发送者 MessageProducer) - 发送消息到已经声明的主题。

f.消息接受者(MessageConsumer) - 接受已经被发送到指定主题的消息。

3.1.4 JMS 开发步骤

在使用 JMS 标准 API 进行编程时，基本过程是先过去连接工厂，然后创建会话，再创建生产者和消费者。最后创建消息进行发送和接收消息。



3.1.5 JMS 开发示例

下面是一个通用的使用 JMS 标准 API 写的点对点发送消息的示例。

```

//1、创建连接工厂类
ConnectionFactory factory = new MQConnectionFactory(URL);
//2、创建工厂
Connection connection = factory.createConnection();
connection.start();
//3、建立会话session,

```

```

/*
 * 第一个参数是是否使用事务，第二个参数是消费者向发送者确认消息已经接收的方式
 * AUTO_ACKNOWLEDGE(自动通知);
 * CLIENT_ACKNOWLEDGE(客户端自行决定通知时机);
 * DUPS_OK_ACKNOWLEDGE(延时//批量通知)
 */
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//4、创建目的地 ( Destination ) 和消息提供者 ( MessageSender )
Destination queueDestination = session.createQueue(Queue.NAME);
MessageProducer queueProducer = session.createProducer(queueDestination);
queueProducer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);//发送方式
//5、创建消息
TextMessage txt = session.createTextMessage("Hello RCloud MQ!!!");
//6、发送消息
queueProducer.send(txt);
//7、关闭对象
queueProducer.close();
session.close();
connection.close();

```

消息接收代码的编写也基本类似。

```

//1、创建连接工厂类
MQConnectionFactory factory = new MQConnectionFactory(URL);

//2、创建工厂
Connection connection = factory.createConnection();
connection.start();
//3、建立会话session,
/*
 * 第一个参数是是否使用事务，第二个参数是消费者向发送者确认消息已经接收的方式
 * AUTO_ACKNOWLEDGE(自动通知);
 * CLIENT_ACKNOWLEDGE(客户端自行决定通知时机);
 * DUPS_OK_ACKNOWLEDGE(延时//批量通知)
 */
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//4、创建目的地 ( Destination ) 和消息消费者 ( MessageConsumer )
Destination queueDestination = session.createQueue(Queue.NAME);
MessageConsumer queueConsumer = session.createConsumer(queueDestination);
//5、接受消息
Message message = queueConsumer.receive();
//6、解析消息
if(message instanceof TextMessage){

```

```
String txt = ((TextMessage)message).getText();
System.out.println("接收的消息 : " + txt);
}
//关闭对象
queueConsumer.close();
session.close();
connection.close();
```

发布订阅模式的代码也很简单。

```
ConnectionFactory factory=new MQConnectionFactory("vm://localhost");
Connection connection=factory.createConnection();
connection.start();
//创建一个Topic
Topic topic=new ActiveMQTopic("testTopic");
Session session=connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//注册消费者1
MessageConsumer consumer1=session.createConsumer(topic);
consumer1.setMessageListener(new MessageListener(){
    public void onMessage(Message m) {
        try {
            System.out.println("Consumer1 get: " + ((TextMessage)m).getText());
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
});
//注册消费者2
MessageConsumer consumer2=session.createConsumer(topic);
consumer2.setMessageListener(new MessageListener(){
    public void onMessage(Message m) {
        try {
            System.out.println("Consumer2 get: " + ((TextMessage)m).getText());
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
});
//创建一个生产者，然后发送多个消息。
MessageProducer producer=session.createProducer(topic);
for(int i=0;i<10;i++){
    producer.send(session.createTextMessage("Message:" + i));
}
}
```

3.2 JMS 消息头介绍

一个消息对象分为三部分：消息头(Headers)，属性 (Properties) 和消息体 (Payload)。对于 StreamMessage 和 MapMessage，消息本身就有特定的结构，而对于 TextMessage，ObjectMessage 和 BytesMessage 是无结构的。一个消息可以包含一些重要的数据或者仅仅是一个事件的通知。

消息的 Headers 部分通常包含一些消息的描述信息，它们都是标准的描述信息。包含下面一些值：

1) JMSDestination

消息的目的地，Topic 或者是 Queue。

2) JMSDeliveryMode

消息的发送模式：persistent 或 nonpersistent。前者表示消息在被消费之前，如果 JMS 提供者 DOWN 了，重新启动后消息仍然存在。后者在这种情况下表示消息会被丢失。可以通过下面的方式设置：

```
Producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

3) JMSTimestamp

当调用 send()方法的时候，JMSTimestamp 会被自动设置为当前事件。可以通过下面方式得到这个值：

```
long timestamp = message.getJMSTimestamp();
```

4) JMSExpiration

表示一个消息的有效期。只有在这个有效期内，消息消费者才可以消费这个消息。默认值为 0，表示消息永不过期。可以通过下面的方式设置：

```
producer.setTimeToLive(3600000);
```

//有效期 1 小时 （1000 毫秒 * 60 秒 * 60 分）

5) JMSPriority

消息的优先级。0-4 为正常的优先级，5-9 为高优先级。可以通过下面方式设置：

```
producer.setPriority(9);
```


6) JMSMessageID

一个字符串用来唯一标示一个消息。

7) JMSReplyTo

有时消息生产者希望消费者回复一个消息，JMSReplyTo 为一个 Destination，表示需要回复的目的地。当然消费者可以不理睬它。

8) JMSCorrelationID

通常用来关联多个 Message。例如需要回复一个消息，可以把 JMSCorrelationID 设置为所收到的消息的 JMSMessageID。

9) JMSType

表示消息体的结构，和 JMS 提供者有关。

10) JMSRedelivered

如果这个值为 true，表示消息是被重新发送了。因为有时消费者没有确认他已经收到消息或者 JMS 提供者不确定消费者是否已经收到。

除了 Header，消息发送者可以添加一些属性(Properties)。这些属性可以是应用自定义的属性，JMS 定义的属性和 JMS 提供者定义的属性。我们通常只适用自定义的属性。

3.3 传送大消息

RCloudMQ 支持超大消息的传输，服务器端的配置参加产品使用手册相关的配置，下面是发送代码示例。

```
//获取连接信息
Properties p = new java.util.Properties();
p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,"org.jnp.
interfaces.NamingContextFactory");
p.put(javax.naming.Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.
interfaces");
p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");
ic = new javax.naming.InitialContext(p);
cf = (javax.jms.ConnectionFactory)ic.lookup("/ConnectionFactory");
//获取队列
queue = (javax.jms.Queue)ic.lookup("queue/exampleQueue");
```

```

connection = cf.createConnection();
session = connection.createSession(false, javax.jms.Session.AUTO_ACKNOWLEDGE);
//构建生产者
MessageProducer producer = session.createProducer(queue);
//构建消息
BytesMessage message = session.createBytesMessage();
FileInputStream fileInputStream = new FileInputStream("/tmp/hl7.xml");
BufferedInputStream bufferedInput = new BufferedInputStream(fileInputStream);
message.setObjectProperty("JMS_HQ_InputStream", bufferedInput);
System.out.println("Sending the message ." + new Date());
producer.send(message);
System.out.println("Large Message sent on " + new Date());

```

消息的接收主要代码示例如下：

```

BytesMessage messageReceived = (BytesMessage)messageConsumer.receive(120000);
File outputFile = new File("/tmp/reveic");
FileOutputStream fileOutputStream = new FileOutputStream(outputFile);
BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOutputStream);

messageReceived.setObjectProperty("JMS_HQ_SaveStream", bufferedOutput);

```

3.4 设定消息优先级

RCloudMQ 支持消息的优先级传输，共分为 0-9 级 10 个级别，下面是发送和接收示例，以及运行结果。

```

//获取连接信息
Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");
ConnectionFactory cf = (ConnectionFactory)initialContext.lookup("/ConnectionFactory");
connection = cf.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(queue);

//构建3个消息，设置不同的优先级
TextMessage[] messagewithPriorities = new TextMessage[3];
messagewithPriorities[0] = session.createTextMessage("1;31/01/2014 19:35:34.000;180");
messagewithPriorities[1] = session.createTextMessage("1;31/01/2014 19:35:35.000;200");
messagewithPriorities[1] = session.createTextMessage("1;31/01/2014 19:35:36.000;220");

```

```
for (int i = 0; i < 3; i++) {
    System.out.println("message " + (i+1) + " created with priority "
+ messagewithPriorities[i].getJMSPriority());
}

//使用延迟发送
producer.send(sentMessages[0]);
System.out.println("Message sent: " + sentMessages[0].getText() + " with priority: "
+ sentMessages[0].getJMSPriority());
producer.send(sentMessages[1], DeliveryMode.NON_PERSISTENT, 5, 0);
System.out.println("Message sent: " + sentMessages[1].getText() + " with priority: "
+ sentMessages[1].getJMSPriority());
producer.send(sentMessages[2], DeliveryMode.NON_PERSISTENT, 9, 0);
System.out.println("Message sent: " + sentMessages[2].getText() + " with priority: "
+ sentMessages[2].getJMSPriority());
```

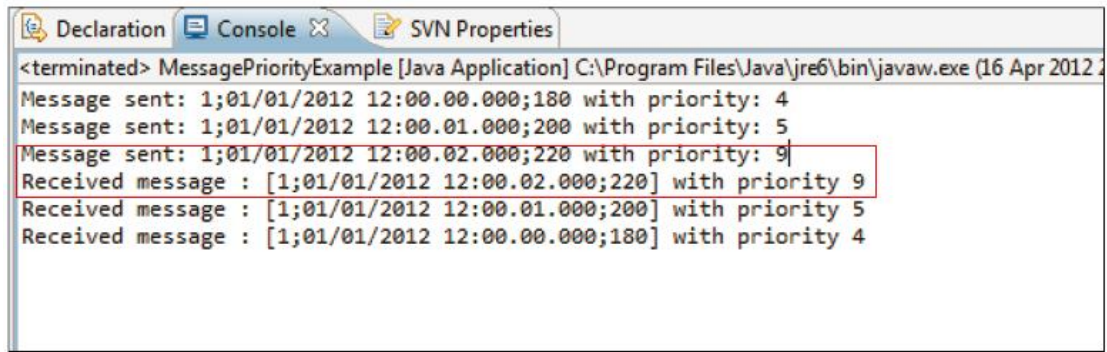
下面是接收消息，使用异步监听方式。

```
//消费者接收信息
MessageConsumer Consumer = session.createConsumer(queue);
Consumer.setMessageListener(new SimpleMessageListener());
connection.start();
Thread.sleep(5000);
```

下面是消息监听器的代码

```
public void onMessage(final Message msg)
{
    TextMessage textMessage = (TextMessage)msg;
    try
    {
        System.out.println("Received message : [" + textMessage.getText() + "] with
priority " + textMessage.getJMSPriority());
    } catch (JMSException e)
    {
        result = false;
    }
}
```

下面是运行结果调试输出信息，可以看出，后发的消息因为优先级高而先接收到。



3.5 集群模式下消息收发

关于 RCloudMQ 的集群配置，请参见产品使用手册，下面的代码示例我们怎么在集群环境中进行消息的收发。

```
//两个节点
Connection connectiona = null;
Connection connectionb = null;
InitialContext initialContexta = null;
InitialContext initialContextb = null;
String ECG_TEXT = "1;31/01/2012 15:45:01.100;1021;1022;1023";
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:1099");
initialContexta = new javax.naming.InitialContext(p);
ConnectionFactory cfa = (ConnectionFactory)initialContexta.lookup("/ConnectionFactory");
p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "jnp://localhost:2099");
initialContextb = new javax.naming.InitialContext(p);
ConnectionFactory cfb =
(ConnectionFactory)initialContexta.lookup("/ConnectionFactory");

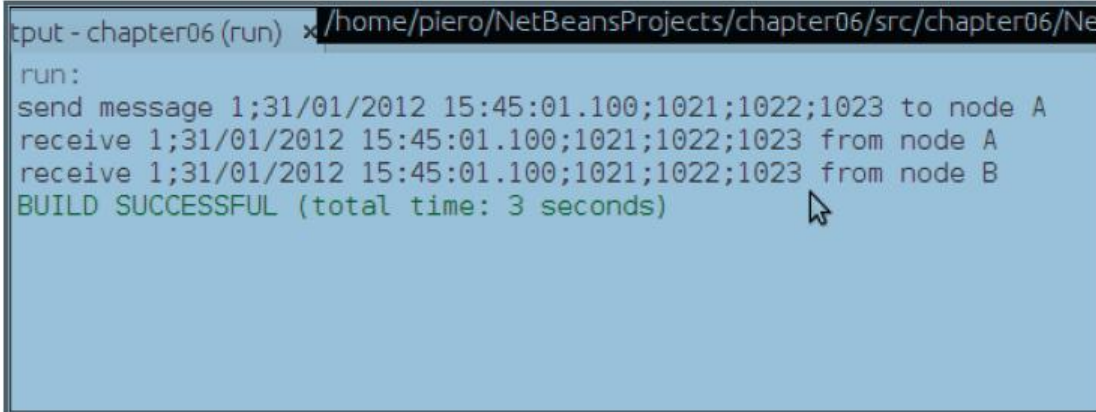
//获取话题，和连接会话信息
Topic topic = (Topic)initialContexta.lookup("/topic/ECGTopic");
connectiona = cfa.createConnection();
connectionb = cfb.createConnection();
connectiona.start();
```

```
connectionb.start();
Session sessiona = connectiona.createSession(false, Session.AUTO_ACKNOWLEDGE);
Session sessionb = connectionb.createSession(false, Session.AUTO_ACKNOWLEDGE);

//A节点发送一个消息
MessageConsumer messageConsumera = sessiona.createConsumer(queue);
MessageProducer producer = sessiona.createProducer(topic);
TextMessage message = sessiona.createTextMessage(ECG_TEXT);
producer.send(message);

//B节点来接收消息
MessageConsumer messageConsumerb = sessionb.createConsumer(topic);
```

下面是运行的结果，A 节点广播一个消息，A 节点和 B 节点的订阅者都收到了该主题的消息。



```
tput - chapter06 (run) x /home/piero/NetBeansProjects/chapter06/src/chapter06/Ne
run:
send message 1;31/01/2012 15:45:01.100;1021;1022;1023 to node A
receive 1;31/01/2012 15:45:01.100;1021;1022;1023 from node A
receive 1;31/01/2012 15:45:01.100;1021;1022;1023 from node B
BUILD SUCCESSFUL (total time: 3 seconds)
```

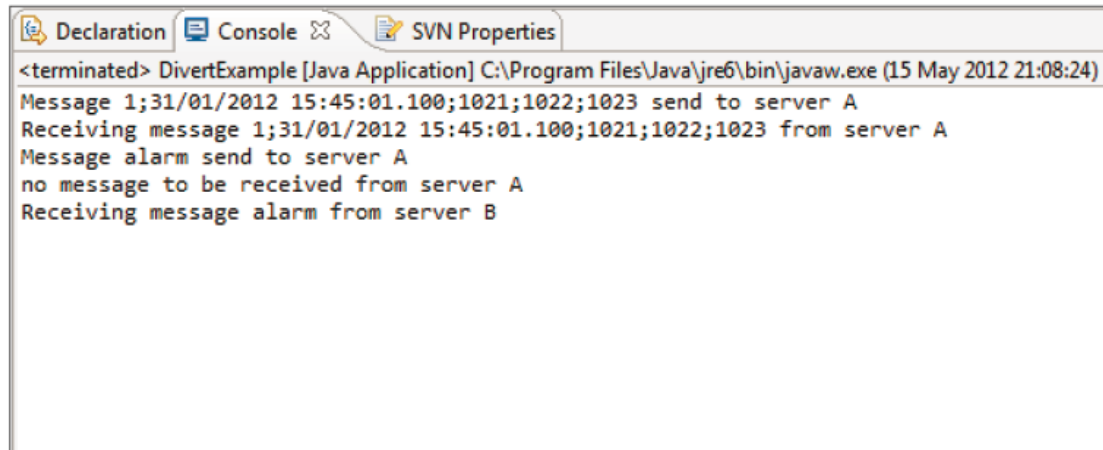
3.6 消息转发

RCloudMQ 支持消息的转发功能，关于消息转发的服务器端配置请参见产品使用手册。下面是一个消息转发的示例代码：

```
//定义两个节点
Queue ECGQueue = (Queue)initialContextA.lookup("/queue/ECQQueue");
Queue AlertQueue = (Queue)initialContextB.lookup("/queue/AlertQueue");
ConnectionFactory cfA = (ConnectionFactory)initialContextA.lookup("/ConnectionFactory");
ConnectionFactory cfB = (ConnectionFactory)initialContextB.lookup("/ ConnectionFactory");
connectionA = cfA.createConnection();
connectionB = cfB.createConnection();
Session sessionA = connectionA.createSession(false, Session.AUTO_ACKNOWLEDGE);
Session sessionB = connectionB.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

```
//一个生产者，两个消费者
MessageProducer ECGProducer = sessionA.createProducer(ECGQueue);
MessageConsumer ECGConsumer = sessionB.createConsumer(ECGQueue);
MessageConsumer AlertConsumer = sessionB.createConsumer(AlertQueue);
connectionA.start();
connectionB.start();
//发送普通消息，不需要转发
String ECG_TEXT = "1;31/01/2012 15:45:01.100;1021;1022;1023";
TextMessage ECGMessage = sessionA.createTextMessage(ECG_TEXT);
ECGProducer.send(ECGMessage);
System.out.println("Message " + ECGMessage.getText() + " send to server A" );
TextMessage receivedMessageA = (TextMessage)ECGConsumer.receive(5000);
System.out.println("Receiving message " + receivedMessageA.getText() + " from server A" );
//发送一个告警消息，转发到告警队列里
String ALERT_TEXT = "this is an alert";
ECGMessage = sessionA.createTextMessage(ALERT_TEXT);
ECGProducer.send(ECGMessage);
System.out.println("Message " + ECGMessage.getText() + " send to server A" );
//主队列没有收到该消息
TextMessage receivedMessageA = (TextMessage)ECGConsumer.receive(5000);
if (receivedMessageA == null)
{
System.out.println("no message to be received from server A" );
}
String ALERT_TEXT = "alert";
ECGMessage = sessionA.createTextMessage(ECG_TEXT);
ECGProducer.send(ECGMessage);
System.out.println("Message " + ECGMessage.getText() + " send to server A" );
//收到告警消息
TextMessage receivedMessageB = (TextMessage)AlertConsumer.receive(5000);
System.out.println("Receiving message " + receivedMessageB.getText() + " from server B" );
```

下面是允许结果。



3.7 流量控制

RCloudMQ 支持发送者的流量控制，也支持接收者的流量控制。

3.7.1 消费速度控制

有两种方式来控制消费者的速度。

一是使用工厂来设定。

//工厂里设定消费者最大速度

```
ClientSessionFactory.setConsumerMaxRate(int consumerMaxRate)
```

二是定义消费者时设定。

//创建消费者时设定消费者最大速度

```
createConsumer(SimpleString queueName,
               SimpleString filter,
               int windowSize,
               int maxRate,
               boolean browseOnly)
```

下面一个示例来说明如何限制消费者的速度。

//构建工厂连接和会话

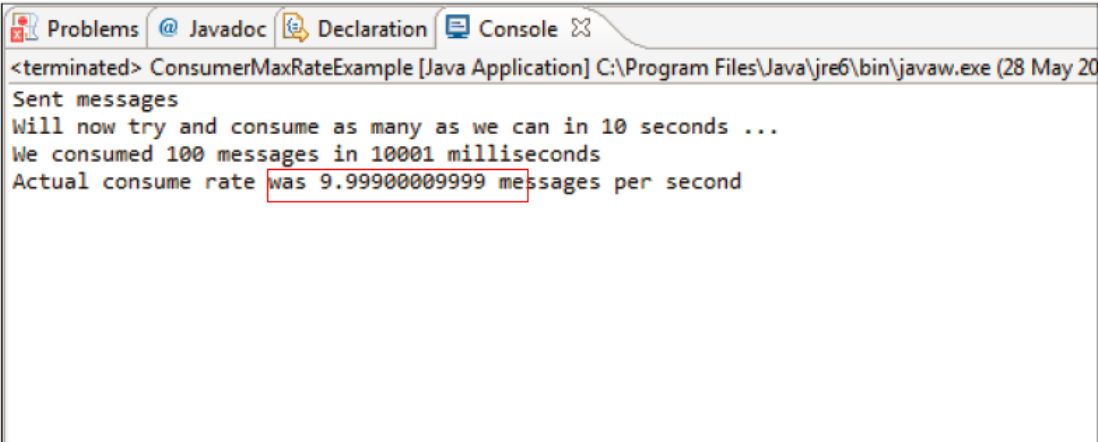
```
String ECG_TEXT = "1;02/20/2012 14:01:59.010;1020,1021,1022";
java.util.Properties p = new java.util.Properties();
p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,"org.jnp.
interfaces.NamingContextFactory");
p.put(javax.naming.Context.URL_PKG_PREFIXES, "org.jboss.naming:org. jnp.interfaces");
p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");
```

```

initialContext = new javax.naming.InitialContext(p);
Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");
ConnectionFactory cf = (ConnectionFactory)initialContext.lookup("/ConnectionFactory");
connection = cf.createConnection();
Session session = connection.createSession(false, Session.AUTOACKNOWLEDGE);
MessageProducer producer = session.createProducer(queue);
MessageConsumer consumer = session.createConsumer(queue);
connection.start();
//发送消息，不限速
final int numMessages = 150;
for (int i = 0; i < numMessages; i++){
    TextMessage message = session.createTextMessage(ECG_TEXT);
    producer.send(message);
}
System.out.println("Sent messages");
System.out.println("Will now try and consume as many as we can in 10 seconds ...");
final long duration = 10000;
int i = 0;
long start = System.currentTimeMillis();
//接收消息
while (System.currentTimeMillis() - start <= duration){
    TextMessage message = (TextMessage)consumer.receive(2000);
    i++;
}
long end = System.currentTimeMillis();
double rate = 1000 * (double)i / (end - start);
System.out.println("We consumed " + i + " messages in " + (end - start) + " milliseconds");
System.out.println("Actual consume rate was " + rate + " messages per second");

```

下面是运行结果。



```

<terminated> ConsumerMaxRateExample [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (28 May 20
Sent messages
Will now try and consume as many as we can in 10 seconds ...
We consumed 100 messages in 10001 milliseconds
Actual consume rate was 9.99900009999 messages per second

```


3.7.2 生产速度控制

生产者速度的控制和消费者的控制类似，下面是示例代码。

```
java.util.Properties p = new java.util.Properties();
p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
p.put(javax.naming.Context.URL_PKG_PREFIXES,"org.jboss.naming:org.jnp.interfaces");
p.put(javax.naming.Context.PROVIDER_URL, "jnp://localhost:1099");
initialContext = new javax.naming.InitialContext(p);
Queue queue = (Queue)initialContext.lookup("/queue/ECGQueue");
QueueConnectionFactory cf =
(QueueConnectionFactory)initialContext.lookup("/ConnectionFactory");
connection = cf.createQueueConnection();
connection.start();
QueueSession session = connection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(queue);
//发送消息
for (int i = 0; i<1000; i++) {
    BytesMessage message = session.createBytesMessage();
    message.writeBytes(new byte[10 * 1024]);
    producer.send(message);
    System.out.println("Sent message: " + i + " " + new Date());
}
```

下面是理论上的发送流量截图。

```
Sent message: 0 Tue May 29 22:49:14 CEST 2012
Total bytes sent 1024
Sent message: 1 Tue May 29 22:49:14 CEST 2012
Total bytes sent 2048
Sent message: 2 Tue May 29 22:49:14 CEST 2012
Total bytes sent 3072
Sent message: 3 Tue May 29 22:49:14 CEST 2012
Total bytes sent 4096
Sent message: 4 Tue May 29 22:49:14 CEST 2012
Total bytes sent 5120
Sent message: 5 Tue May 29 22:49:14 CEST 2012
Total bytes sent 6144
Sent message: 6 Tue May 29 22:49:14 CEST 2012
Total bytes sent 7168
Sent message: 7 Tue May 29 22:49:14 CEST 2012
Total bytes sent 8192
Sent message: 8 Tue May 29 22:49:14 CEST 2012
Total bytes sent 9216
Sent message: 9 Tue May 29 22:49:14 CEST 2012
Total bytes sent 10240
```

我们看接收端的代码。

```
for (int i = 0; i < 1000; i++) {
    BytesMessage message = session.createBytesMessage();
    message.writeBytes(new byte[10 * 1024]);
    producer.send(message);
    System.out.println("Sent message: " + i + " " + new Date());
    message = (BytesMessage)messageConsumer.receive(3000);
    System.out.println("consuming message " + i);
}
```

下面是实际接收到的消息速度截图，可以发现生产者的速度实际受到了限制。

```
Total bytes sent 1016832
consuming message 992
Sent message: 993 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1017856
consuming message 993
Sent message: 994 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1018880
consuming message 994
Sent message: 995 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1019904
consuming message 995
Sent message: 996 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1020928
consuming message 996
Sent message: 997 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1021952
consuming message 997
Sent message: 998 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1022976
consuming message 998
Sent message: 999 Tue May 29 22:57:23 CEST 2012
Total bytes sent 1024000
consuming message 999
```

第四章 开发建议

4.1 避免违背设计模式

重用连接 / 会话 / 接收者 / 发送者。

最常见的错误恐怕就是每发送 / 接收一个消息都要创建一个新的连接 / 会话 / 发送者或接收者。这样非常浪费资源。这些对象的创建要占用时间和网络带宽。它们应该进行重用。

有些常用的框架如 Spring JMS Template 在使用 JMS 时违背了设计模式。如果你在使用了它后性能可能会受到影响。Spring 的 JMS 模板只有与能缓存 JMS

会话的应用服务器一起使用 才是安全的，并且只能是用于发送消息。使用它在应用服务器中同步接收消息是不安全的。

4.2 避免使用繁锁的消息格式

发送消息应尽量使用轻量级的格式，如 JSON。尽量避免使用 XML，它会使数据量变大进而降低性能。所以应该尽量避免在消息体中使用 XML。

4.3 不要为每个请求都创建新的临时队列

临时队列通常用于请求—响应模式的消息应用。在这个模式中消息被发往一个目的，它带有一个 `reply-to` 的头属性指向一个本地的临时队列的地址。当消息被收到后，接收方将响应做为消息发 往那个 `reply-to` 指定的临时的地址。如果每发一个消息都创建一个临时队列，那么性能将会受很大影响。正确的作法是在发送消息时重用临时队列。

4.4 尽量不要使用 MDB

使用 MDB，消息的接收过程要比直接接收复杂得多，要执行很多应用服务器内部的代码。在设计应用时要问一下是否真的需要 MDB，可不可以直接使用消息接收者完成同样的任务。