

JavaBlog电子“迷你书”

JMS与消息中间件专题



看图说话：

封面是今年(2010-11)在南京某咖啡店中留下的照片，因为店主的用心，那的布局和环境给我留下了深刻的影响。

照片上的那种花是那儿最多的装饰物了，虽然那样的花不够香，但很容易让人喜欢，花瓶就是一个很普通的玻璃杯，这杯家里有 4 个，去那的时候看见店主又更换着店内，店外别致的装饰物，还在琢磨着怎么安放，似乎我们每个人要知道最佳实践的答案，大多时候是我们经历了反复实践与不断思考才能得出的结论。

历史期刊

用最简洁的页面描述企业应用与Java艺术!

H.E.'s Blog 电子“迷你书”

性能优化专题



2010-2 第一期

JavaBlog 电子“迷你书”

系统架构专题



2010-4 第二期

JavaBlog电子“迷你书”

JMS与消息中间件专题



2010-11 第三期

目录

首篇语(Editor' s letter)	4
大型系统中使用 JMS 优化技巧--Sun OpenMQ.....	5
JMS 与 Java 消息中间件.....	12
JMS 集群 OpenMQ 的那些事	21
OpenMQ 中的 JMS 与 JMX.....	23
OpenMQ 命令	27
Sun OpenMQ Topic 消息收/发 一Tips	29
Linux 下 OpenMQ 集群的启动异常	33
GlassFish JMS 集群.....	34
Spring 集成 JMS OpenMQ.....	38
GlassFish OpenMQ JDBC.....	47

首篇语(Editor's letter)

本期内容主要对 JavaEE 中的 JMS 话题进行讲述，其中一些文章是我们经过了很多错/败与大量的测试总结出的经验和心得，所以拿来与大家分享。

在大型系统中不仅仅要考虑 JMS 服务器端或者 JMS 客户端消息收/发的效率和性能，在系统架构设计方面还需要考虑到 JMS 服务器的**可扩展性、伸缩性、容错性**。在设计时如何巧妙的将 JMS 的性能提高，如何充分发挥 JMS 所有的功能，如何提高 JMS 服务器的容错能力 等等，都是我们一线架构师们需要思考的话题。

对于 JMS 服务器的每一项参数必须在投入使用前理解的明明白白，做到一旦出现关于 JMS 的问题能迅速定位，提供开发团队解决问题的明确思路。我在此感谢我们技术团队的所有同仁，没有你们的投入与陪伴，不会让我有机会遇见各种形形色色的问题，也不会让我对 JMS 产品去深入的了解与提高。

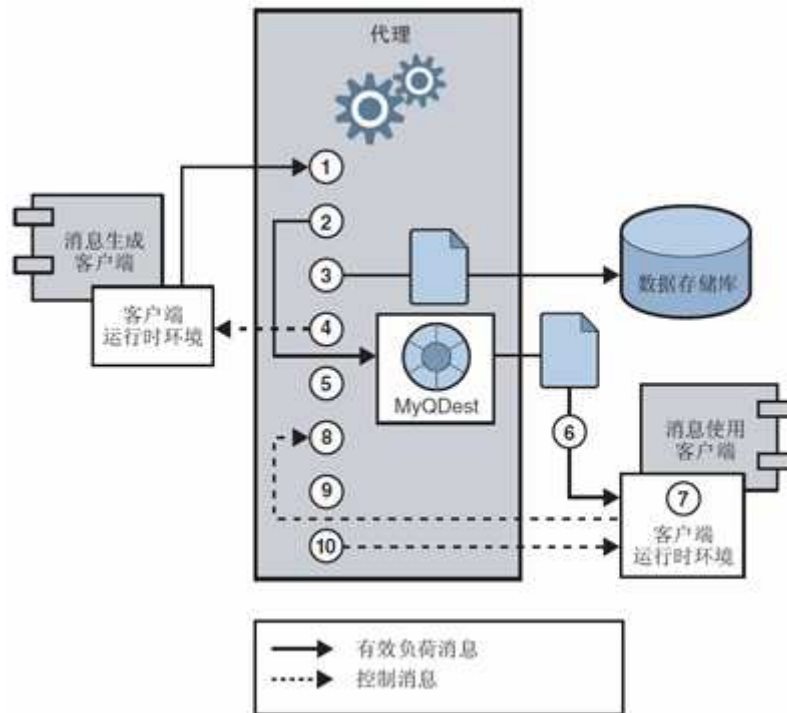
感谢，再一次感谢南京 AMT 技术团队所有成员。

读者内容反馈: njthnet@gmail.com

大型系统中使用 JMS 优化技巧--Sun OpenMQ

我们先来看看在 Sun OpenMQ 系统中 一个持久、可靠的方式传送消息的步骤是怎么样的，

如图所示：



在传送过程中，系统处理 JMS 消息分为以下两类：

- 有效负荷消息，由生成方发送给使用方的消息。
- 控制消息，代理与客户端运行时环境之间传送的私有消息，用于确保有效负荷消息成功传送和控制跨连接的消息流。

详细流程如下：

消息生成

1. 客户端运行时环境通过连接将消息从消息生成方传送到代理。

消息处理和路由

2. 代理从连接中读取消息并将此消息放入相应的目的地中。
3. 代理将（持久性）消息放入数据存储库中。
4. 代理向消息生成方的客户端运行时环境确认已收到消息。
5. 代理确定消息的路由。
6. 代理将消息从目的地写入适当的连接，并使用使用方的唯一标识符标记该消息。

消息使用

7. 消息使用方的客户端运行时环境将消息从连接传送到消息使用方。
8. 消息使用方的客户端运行时环境向代理确认消息已使用。

消息生命周期结束

9. 代理处理客户端确认，并在收到所有确认后删除（持久性）消息。
10. 代理向使用方的客户端运行时环境确认，告知客户端确认已得到处理。

如果管理员删除目的地中的消息，或者管理员删除或重新定义长期订阅，导致主题目的地中的消息未被传送即被删除，则代理可以在消息被使用前将它丢弃。在其他情况下，您可能希望代理将消息存储在称为停用消息队列的特殊目的地中，而不是将它们丢弃。在以下情况，消息会被放入停用消息队列中：消息过期时、消息因内存限制而被删除时，以及因客户端引发异常而导致传送失败时。通过将消息存储在停用消息队列中，您可以解决系统问题并在某些情况下恢复消息。

以下是针对 JMS 应用中的一些优化策略，H.E.在这里分为几点向大家进行介绍：

1.收发消息的属性

在接收端和发送端可以设置消息发送和接收的属性，对于消息发送时还需要注意客户端的消息确认模式一共有 3 种客户机确认模式：

■ 在 AUTO_ACKNOWLEDGE 模式下开销最大，可以保证消息逐条传送的可靠性，会话自动确认客户端使用的每条消息。会话线程会被阻止，以等待代理确认它已处理了每个已使用消息的客户端确认；

■ 在 CLIENT_ACKNOWLEDGE 模式下，在一条或多条消息被使用后，客户端通过调用消息对象的 acknowledge() 方法来显式确认。这样该会话就确认了自上次调用该方法后使用的所有消息。会话线程会被阻止，以等待代理确认它已处理了客户端确认。Message Queue 提供使客户端可仅仅确认收到一条消息的方法，从而扩展了该模式。因此需要的带宽开销较小；

■ 在 DUPS_OK_ACKNOWLEDGE 模式下，会话在使用了十条消息后进行确认。会话线程不会因等待代理确认而被阻止，因为在该模式下代理确认不是必需的。虽然该模式可保证不会丢失消息，但并不能保证不会收到重复的消息，因此名称为：DUPS_OK。

对于更关心性能而不是可靠性的客户端，Message Queue 服务通过提供 NO_ACKNOWLEDGE 模式来扩展 JMS API。在该模式下，代理不跟踪客户端确认，所以不保证使用方客户端已成功处理了消息。对于发送至非长期订户的非持久性消息，选择该模式可提高性能。

如果你有大量的消息需要进行发送，可以采用 DUPS_OK_ACKNOWLEDGE 模式，因为他是最快的。

代码示例

```
connection = connectionFactory.createTopicConnection();  
  
session=connection.createSession(false, Session.DUPS_OK_ACKNOWLEDGE);
```

2.慎用消息压缩

消息的大小对消息的效率是有影响的，跟 Http/Gzip 的道理一样，减少网络的负载，但是并不能提升你的运行效率，反而会将你的接收响应时间延时。以下是消息压缩的代码示例：

1.发送压缩消息的代码示例：

```
for (int i=0;i<10000;i++){ //循环 1W 次发送消息

    Javabloger_Msg msg=new Javabloger_Msg (); // 自定义的对象，进行实例
    化

    msg.setTime( System.currentTimeMillis() ); //赋值

    msg.setChat("This is JavablogerMsg Test msg.复制 300 次"); //赋值，此处省略
    1000 个字符

    myTextMsg.setObject( msg ); //放入消息对象

    myTextMsg.setBooleanProperty("JMS_SUN_COMPRESS", true); //设置是否进行
    压缩属性

    myMsgProducer.send(myTextMsg); // 发送消息

}
```

2.接收端会自己进行解压缩，无需人工干预，但是可以查看出消息经过压缩前后的大小：

```
// 获得压缩前的消息大小。
```

```
int
```

```
uncompressed=bytesMessage.getIntProperty( "JMS_SUN_UNCOMPRESSED_SIZE" );
```

```
//获得压缩后的消息大小。
```

```
int compressed=bytesMessage.getIntProperty( "JMS_SUN_COMPRESSED_SIZE" );
```

经过测试 1000 个字符 压缩以后只有 95 个字符位的大小。

1000 个字符的消息发送 1W 个没有经过压缩， 收发在 300 毫秒以内完成，

1000 个字符的消息发送 1W 个经过压缩后， 收发在 1600 毫秒以内完成，也就是 1 秒多以内完成。

3.取消对消息收/发的验证

可以查阅我写的另外一篇文章

<http://www.javabloger.com/article/sun-glassfish-openmq-topic.html> 在文章中我提到，如果在发送端或者在接收端加上用户和密码的验证会大大降低系统运行效率的，所以请慎用 Sun OpenMQ 中的身份验证功能。

4.对服务器连接的开关

客户端对 JMS 服务器的开关连接对服务器的运行性能有很大的影响，所以：

- 1 我们可以采用容器提供的连接池，让容器来完成我们的连接资源管理。

- 2 如果使用非容器下的运行状态，那么每次发送完毕一个消息后下面还有消息的话，关闭 session 就可以了，无需关闭对 JMS 服务器的连接。

5.调整 JVM 虚拟机运行参数

100 个线程同时发送 100 个消息，一共 10w 个消息，运行到一半的时候出现

```
[29/Mar/2010:03:34:07 EDT] [B1089]: In low memory condition, Broker is attempting to free up resources
```

```
[29/Mar/2010:03:34:07 EDT] [B1088]: Entering Memory State YELLOW from previous state GREEN - allocated memory is 151214K, 80% of total memory used
```

错误，显然在说内存不够。经过 google 以后真实我的想法是正确的，参考文档：

http://docs.sun.com/app/docs/doc/819-4467/6n6k98bq2?l=zh_tw&a=view#aeokn

http://docs.sun.com/app/docs/doc/819-4467/6n6k98bqa?l=zh_tw&a=view

加大运行内存，将 JMS 服务器的内存使用率调整为 1G，效果非常明显，命令如下：

```
nohup imqbrokerd -tty -name myBroker -port 6769 -cluster
```

```
172.16.2.214:6769,172.16.2.215:6769 -D"imq.cluster.masterbroker=172.16.2.215:6769"
```

```
-vmargs "-Xms256m -Xmx1024m" &
```

6.慎用数据库存储消息

如果对于消息体内容较大，频率稍低的可以采用将消息体放入数据库进行存储。如果收发频率较高，并且消息体不算很大可以使用默认状态在内存中使用。也就是说 根据不同的应用场景进行应用，千万不要认为将消息体放入数据库中存储是最好的方案，那样只能提高消息的稳定性和消息的安全性。

7.开启多个队列分载

来看一个测试案例：

TopicA、B 2 个队列在同一台机器上 并发线程数 $100 \times 2 = 200$ 个线程，每个线程发 1000 个消息，2 个接收端平均每个接收端 40 秒以内处理完成。

TopicA、B 2 个队列在同一台机器上 并发线程数 $100 \times 2 = 200$ 个线程，每个线程发 500 个消息，2 个接收端平均每个接收端 10 秒以内处理完成。

TopicA、B、C、D 4 个队列在同一台机器上 并发线程数 $50 \times 4 = 200$ 个线程，每个线程发 250 个消息，4 个接收端平均每个接收端 5 秒以内处理完成。

经过以上测试可以表明，10W 条消息进行散列以后效率明显提高了，有人会问为什么需要进行散列，因为在消息接收端 `onMessage(Message msg)` 是单线程，对于 Topic 类型的消息，如果启动多线程也就是启动了多个 `onMessage` 触发，并不能提高运行效率。

8.队列的属性

无论是 Topic 还是 queue 类型的消息队列，默认状态都设置了上限的状态，如果消息量大的话将会出现这样的信息：

```
com.sun.messaging.jmq.jmsserver.util.BrokerException: [B4120]: Can not store
message 18727-172.16.2.19(8c:56:3e:4f:ba:71)-4100-1269861102326 on destination
TopicCase [Topic]. The destination message count limit (maxNumMsgs) of 100000
has been reached.
```

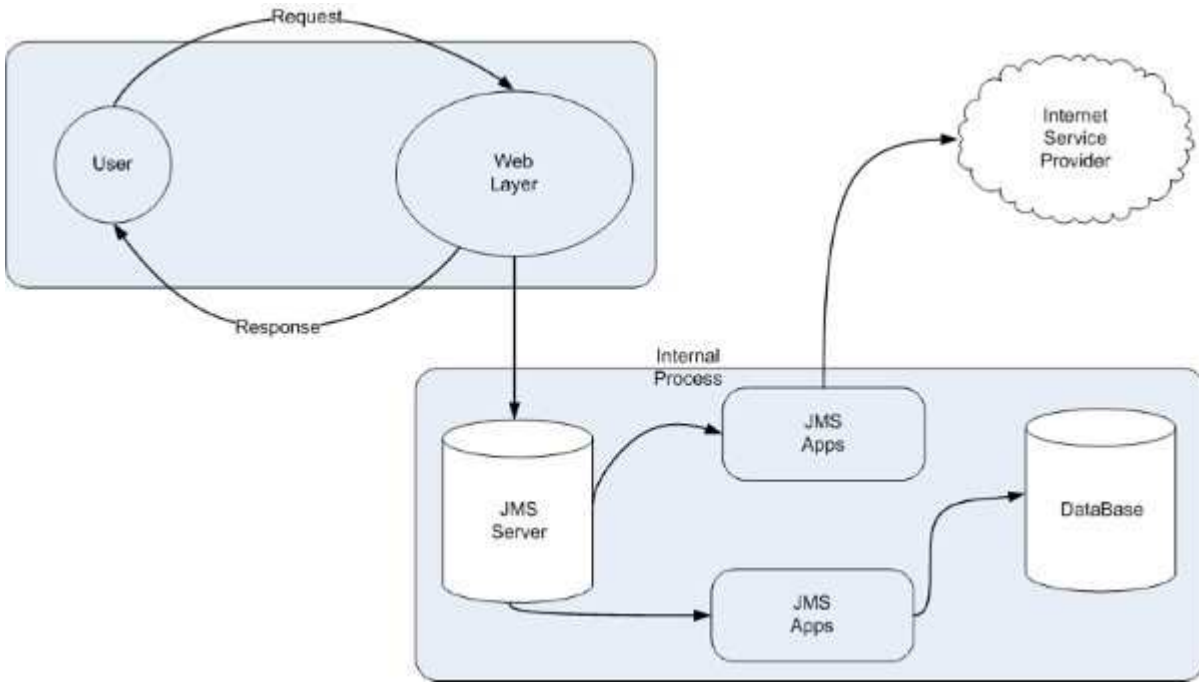
所以需要注意将队列中的状态，将队列的状态修改为没有限制的 **Unlimited** 选项。

JMS 与 Java 消息中间件

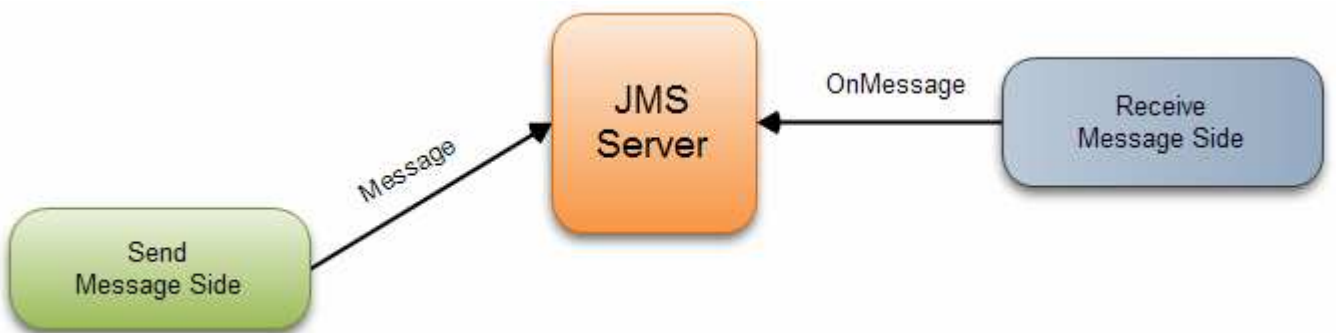
JMS 是一种企业消息传送的 API，并不是 MOM 消息中间件系统的全部，JMS 也是一种规范，类似于 JDBC，我们通 JMS API 访问 JMS 的服务器。目前 JMS 服务器的主要产品有 IBM WebSphere MQ、SonicMQ、Sun Open MQ、BEA WebLogic JMS、Oracle AQ 以及 我们最常用的 JBoss MQ 和 Apache 的 ActiveMQ.。而 JMS 服务器是 MQ(Message Queen)产品家族中的一种，Microsoft Message Queuing (mSMQ) 也是 MQ 产品类似于 JMS 服务器。

JMS 在 J2EE 系统中的应用场景

异步消息比同步消息操作更加便利。在 J2EE 系统中最常见的一个场景，前端浏览器 jsp/servlet 向服务器端发出一个请求，jsp/servlet 将请求传递给后端的应用程序处理业务逻辑，业务模块将响应的结果直接返回给客户 端，而不是真正的计算结果，例如一个网站的用户注册功能，一个用户点击注册以后，将会发送一份邮件给他当时的注册邮箱，如果需要等到邮件发送成功再返回给 用户结果的话，用户体验将会很差，所以将结果直接返回给用户，将用户注册的信息通过消息发送给后端程序慢慢处理。



谈到 JMS 一词 大体上有 3 个部分 1 消息发送端 2 中间件服务器 3 消息接收端 3 个组件缺一不可。 JMS 消息分为两种消息模式，点对点和发布者/订阅者。许多提供商支持这一因此，程序员可以在他们的分布式软件中实现面向消息的操作，这些操作将具有不同面向消息中间件产品的可移植性。



Java 消息服务器是指，将数据通过消息作为载体在网络中从一个系统异步传送给另一个系统。这样的异步消息传送意味着：发送者不需要等待接收者接收 或处理该消息；它可以自由地发送消息并持续进行处理。这样一个异步式的架构主要依赖于一台消息服务器（message

server)。消息服务器，也称为消息路由器（message router）或代理（broker），它负责从一个消息传送客户端向其他消息传送客户端传送消息。

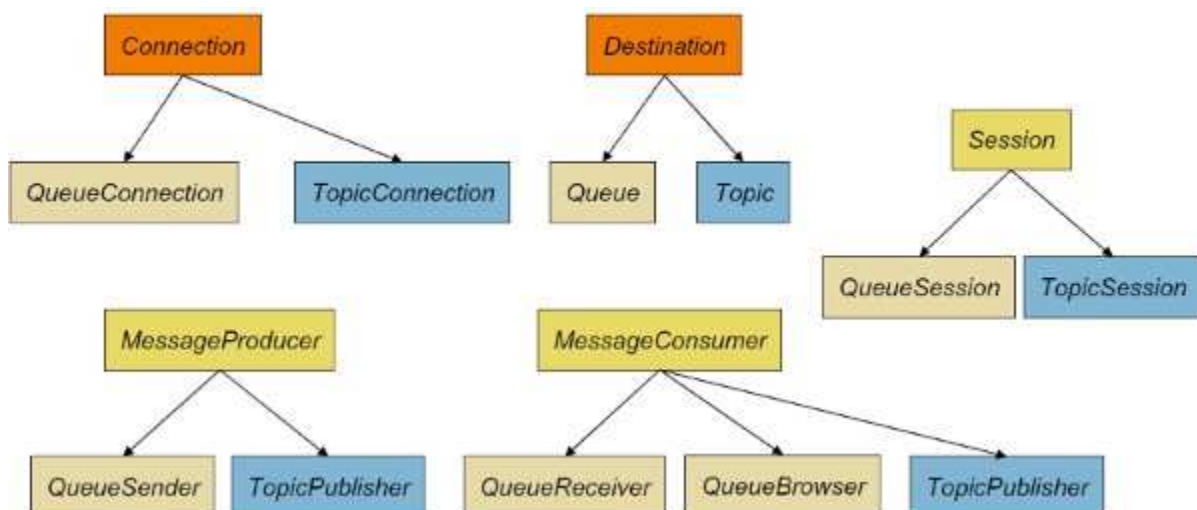
JMS 对与一个大型系统是必不可少的一个应用组件，可以利用 JMS 来实现 3 个目的：

- 1.提高可伸缩性(Increase Scalability),
- 2.可以利用 JMS 来缓解系统瓶颈(Reduce Bottlenecks)
- 3.提高系统对用户的响应能力

JMS 公共 API 内部，和发送和接收 JMS 消息有关的 JMS API 接口常用的有 7 个：

- ConnectionFactory
- Destination
- Connection
- Session
- Message
- MessageProducer
- MessageConsumer

其中关键的 5 个部件，如图所示：

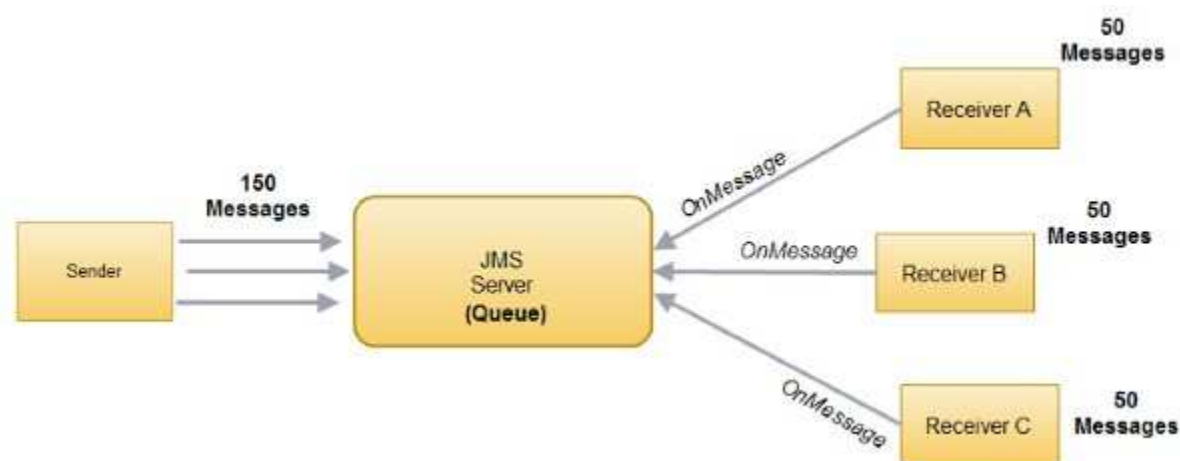


Topic 与 Queue 消息的区别

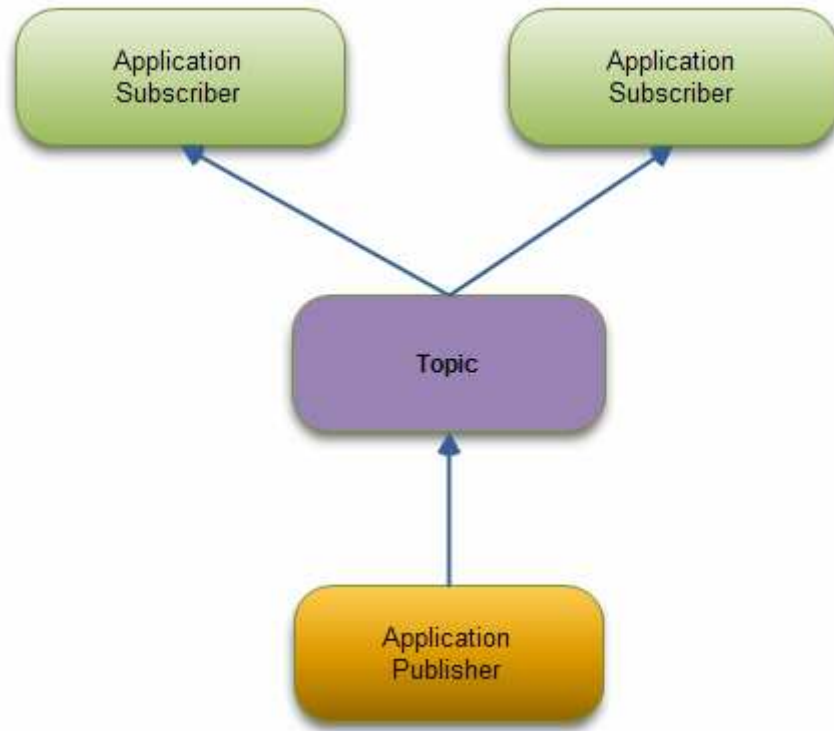
Queue 是一对一的消息传送，你可以看做是 QQ 应用程序中的一对一发送消息，一个人发出消息，只能有另外一个人阅读的到，中间需要通过一个队列支持 Queue 消息发送完毕后会保存在 JMS 服务器的队列中，如果接收端接收以后将从队列中摘除。**如图所示：**



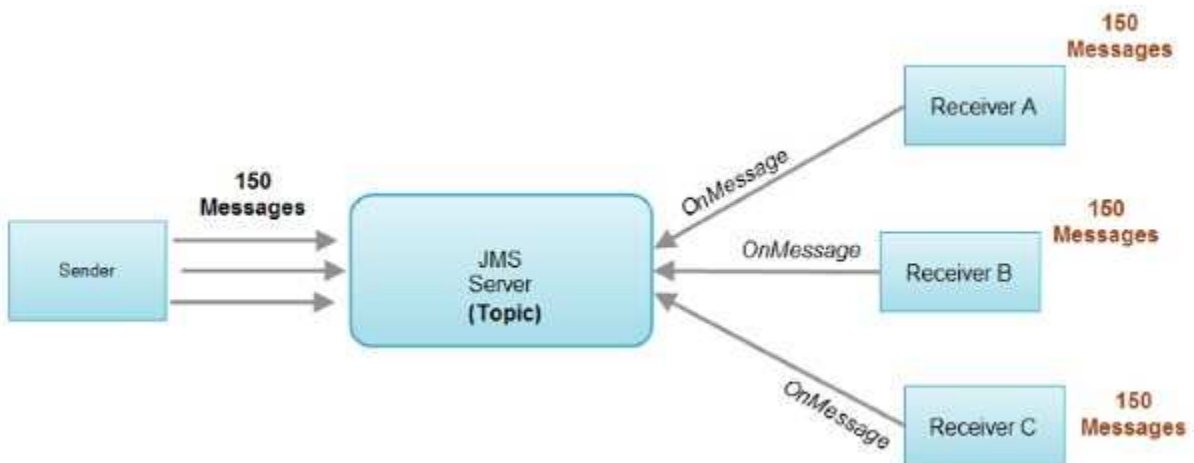
Queue 消息发送到服务器，接收端会平均接收到发送端发送过来的消息，如图所示，一次发送 150 个消息，3 个接收端每个收到 50 个



Topic 是一对多的消息传送，你可以看做是 QQ 应用程序 QQ 群 聊天中的一对多发送消息，一个人发出消息，可以有多个订阅的人阅读的到，需要有一个消息主题作为支柱.Topic 消息发送完毕以后 无论有没有客户端接收，JMS 服务器中的 Topic 消息都不会存在 JMS 服务器中。**如图所示：**



发送 Topic 消息无论连接在 JMS 服务器上有多少个接收端, 将会收到同样的消息, 而且 Topic 发送完毕以后不会保留在 JMS 服务器, 否则将会和 Topic 消息的设计思想相互冲突。



JMS 消息的属性

JMS 的每条消息分为 **Header**、**Properties** 、**Body** 3 个属性：

Header、**Properties** 中包含可设置的参数，分别是：

JMSDestination 消息发送的目的地

JMSDeliveryMode 传递模式，有两种模式：**PERSISTENT** 和 **NON_PERSISTENT**，

PERSISTENT 表示该消息一定要被送到目的地，否则会导致应用错误。**NON_PERSISTENT** 表示偶然丢失该消息是被允许的，这两种模式使开发者可以在消息传递的可靠性和吞吐量之间找到平衡点。

JMSMessageID 唯一识别每个消息的标识，由 JMS Provider 产生。

JMSTimestamp 一个消息被提交给 JMS Provider 到消息被发出的时间。

JMSCorrelationID 用来连接到另外一个消息，典型的应用是在回复消息中连接到原消息。

JMSReplyTo 提供本消息回复消息的目的地址

JMSRedelivered 如果一个客户端收到一个设置了 **JMSRedelivered** 属性的消息，则表示可能该客户端曾经在早些时候收到过该消息，但并没有签收(acknowledged)。

JMSType 消息类型的识别符。

JMSExpiration 消息过期时间，等于 **QueueSender** 的 **send** 方法中的 **timeToLive** 值或 **TopicPublisher** 的 **publish** 方法中的 **timeToLive** 值加上发送时刻的 GMT 时间值。如果

timeToLive 值等于零，则 JMSExpiration 被设为零，表示该消息永不过期。如果发送后，在消息过期时间之后消息还没有被发送到目的地，则该消息被清除。

JMSPriority 消息优先级，从 0-9 十个级别，0-4 是普通消息，5-9 是加急消息。JMS 不要求 JMS Provider 严格按照这十个优先级发送消息，但必须保证加急消息要先于普通消息到达。

Body 分为以下 5 种：

TextMessage

这种类型携带了一个 java.lang.String 作为有效负载。它可以用于简单的文本消息交换，还可以用于更复杂的字符数据交换，比如 XML 文档等。

ObjectMessage

这种类型携带了一个可序列化 Java 对象作为有效负载。它可以用于 Java 对象交换。

BytesMessage

这种类型携带了一组原始类型字节流 (primitive byte) 作为有效负载。它可以使用应用程序的本机格式 (native format) 来交换数据，这种格式可能不兼容其他现有的 Message 类型。当 JMS 纯粹用于两个系统之间的消息传送时，也可以使用这种类型，而且该消息的有效负载对 JMS 客户端来说是不透明的。

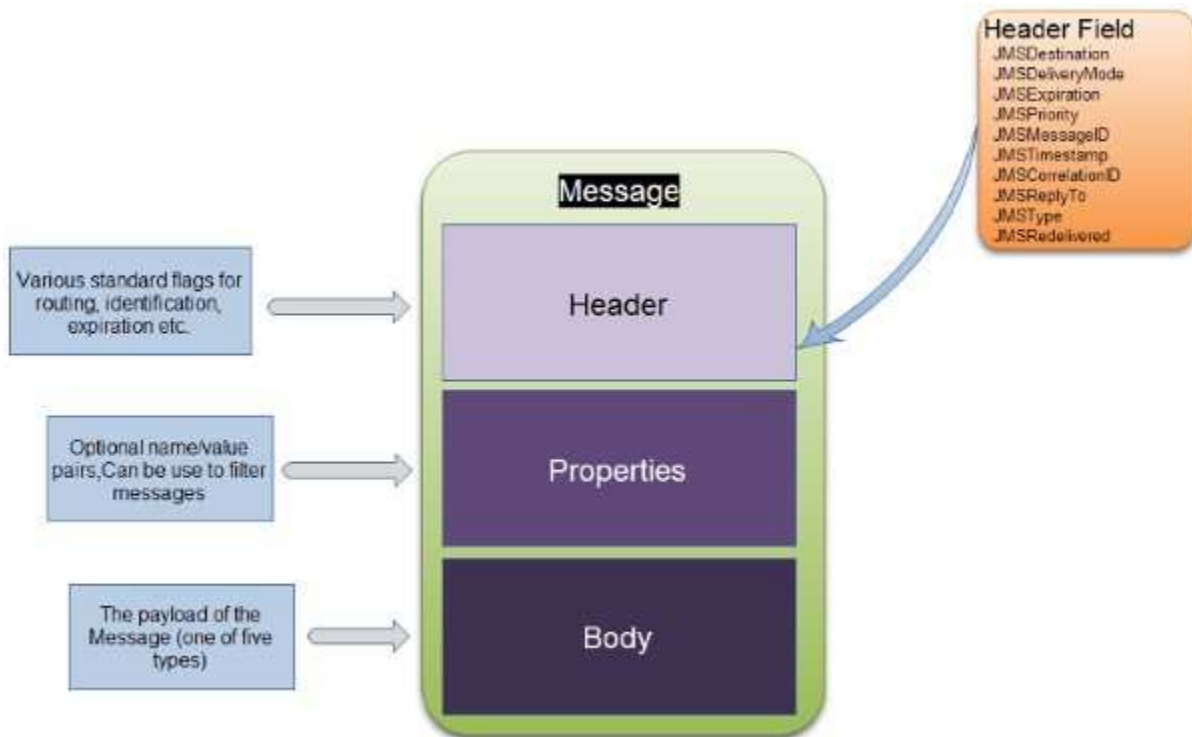
StreamMessage

这种类型携带了一个 Java 原始数据类型流 (int、double、char 等) 作为有效负载。它提供了一套将格式化字节流映射为 Java 原始数据类型的简便方法。在以固定顺序进行原始应用数据交换时，这种模型非常易于编程实现。

MapMessage

这种类型携带了一组名/值对 (name-value pair) 作为有效负载。有效负载类似于一个 `java.util.Properties` 对象，除了有效负载值必须是 Java 原始数据类型或它们的包装器之外。MapMessage 可以用于传送键入的数据。

如图所示：



JMS 的消息存储

基于内存 最快

基于文件 最慢

基于数据库 比较慢

JMS 的消息过滤功能

这里 selector 是一个字符串，用来过滤消息。也就是说，这种方式可以创建一个可以只接收特定消息的一个消费者。Selector 的格式是类似于 SQL-92 的一种语法。可以用来比较消息头信息和属性。例如：发送消息属性中指定 ip 地址位数是奇数的接收，而 ip 地址位数是偶数的绝对不会接收到消息，具体实现方法请 Google 关键字 “jms selector”

JMS 消息的顺序

乱序接收，顺序处理。也就是说，消息在发送，传输和接收过程中可能是乱序的，但消费者在接收到消息之后，并不立即处理，而是先将消息排序，然后在处理。JMS 消息头部的 JMSCorrelationID 可以帮助我们完成这个工作。JMSCorrelationID 存放了另一个消息的 id。消息的发送者，如果要保证消息的顺序性，要将后发送的消息的 JMSCorrelationID 设置成前一个消息的 id。消费者接收消息后，如果发现其头部有 JMSCorrelationID，则查看该消息是否已被处理过，如果没有，则等待该消息，至到该消息被处理后，才处理这个消息。这一工作需要发送者和接收者都记住已经发送和接收过的消息，以便于给后来的消息参考。由此可见，在 JMS 中设置 JMS 的 head 还是能起到不少作用的。

口水：

本文图绘采用国产软件 《亿图图示专家》所绘制，感谢《亿图图示专家》的所有开发人员制造出这么好的工具。

JMS 集群 OpenMQ 的那些事

我的废话:

JMS、OpenMQ 这 2 个词不断的在我的空间里出现，因为我们对于 JMS 服务器的重视程度很高，JMS 在整个系统中的角色是系统组件之间通讯的桥梁，我们利用 JMS 服务器构建“低耦合，高内聚”的重要手段之一，曾经我们想过采用 http、ejb 来实现，但是这 2 个手段无法满足以下 4 个需求：

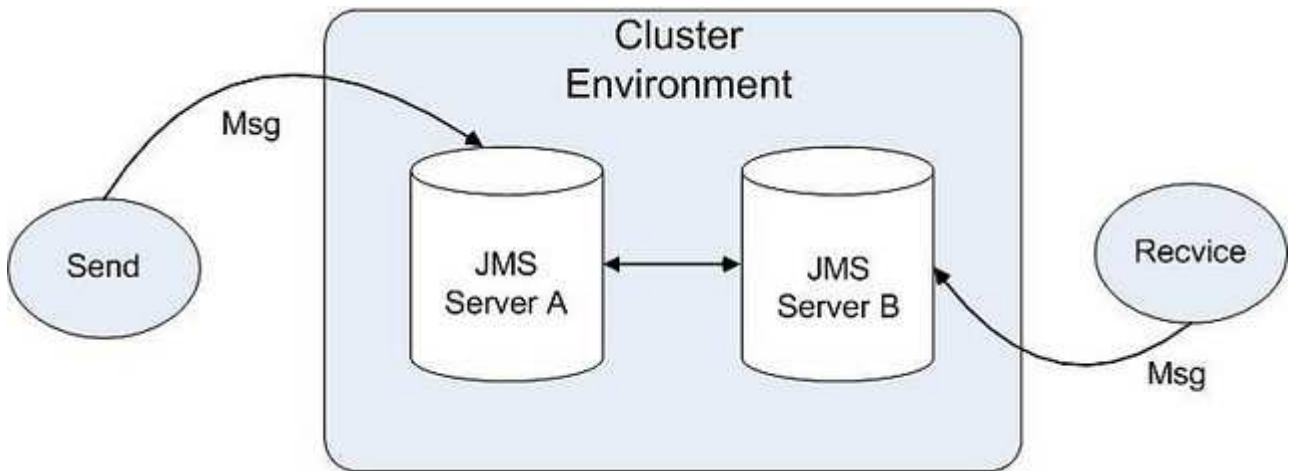
- 1.不停机,不改变集群中任何节点的状态，可以对集群中的节点任意添加，进行压力分载，
- 2.发出一个请求，集群中的所有应用程序节点都能接收相同数量的请求，
- 3.通信中组件的接收端失效，当生效后再运行时能继续接收到上次失效前的请求，
- 4.请求中有多种不同的类型，下面的集群节点可以按照不同的类型接收，可以动态的指定集群节点中的机器完成不同的任务。

经过多番比较最后决定采用 JMS 作为系统中组件与组件通讯的重要手段。废话少说，进入正题，介绍在实践中使用 JMS 的一些经验，和大家分享一下：

失效转发

有 2 台 Sun 的 OpenMQ 服务器集群后，并不是一个 work，另外一个 standby，如果你认为是这样，那么只是一些表面的现象。我们可以做一个测试，如果你向其中一个 jms 服务器发送消息 Topic 或者是 Queue 类型的消息，那么你也可以在另外一台服务器上接收到消息。说明他们之间在集群后内部是完全相通的，这样集群中无论其中任何一个 JMS 的服务器 down 机消息都会失效转发到另外一台上，因为当你向其中一台发送消息的时候 JMS 集群内部就完成了消息 copy。

如图所示：

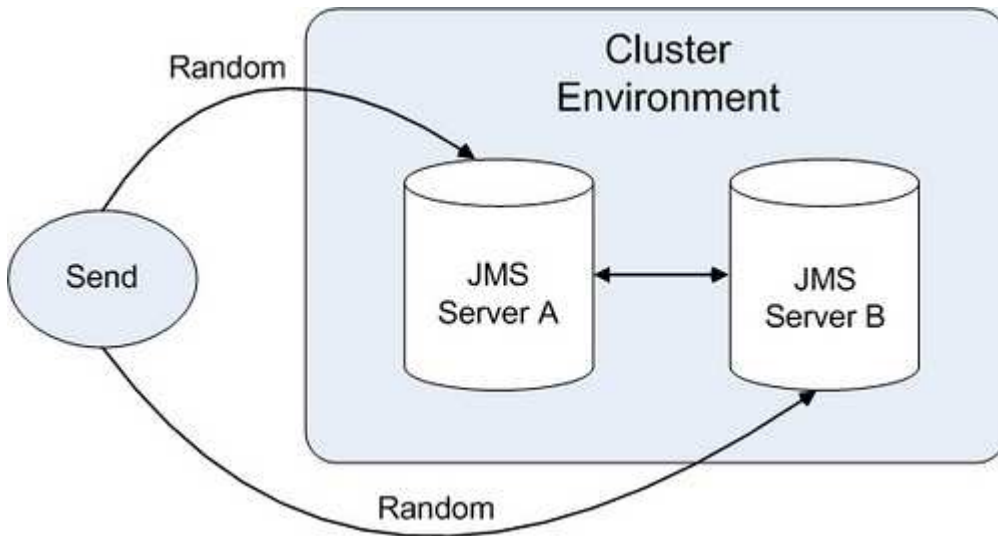


BWT:如果你测试在 JMS 失效转发的过程中是否生效,那么请注意,如果你发送的是大对象(2M 以上的数据)会导致失效转发失败,因为默认反复尝试的时间 间隔是 5000 毫秒,你需要提高 `imqReconnectInterval` 的值(毫秒)作为时间间隔反复尝试连接每个地址的之间时间间隔的长短。

压力分载

如果你有 2 台以上的 JMS 服务器作为集群的节点,首先你需要在你的发送端和接收端的主机地址中写上这些 JMS 服务器的 ip 地址,其次还需要注意,在发送端 和接收端的连接属性中设置 `imqAddressListBehavior` 指定了尝试连接指定地址的顺序。如果将此属性设置为字符串 `PRIORITY`,将按地址在地址列表中的显示顺序尝试连接。如果属性值为 `RANDOM`,将按随机顺序尝试连接地址;这种方式非常有用,例如,当许多 `Message Queue` 客户端共享同一个连接工厂对象时,这种方式有助于防止所有客户端尝试连接到同一个代理地址。也就是说如果你不配置,那么发送端的压力始终只会指向一台,就是你写在在代理地址列表 (`imqAddressList`) 中的第一个主机。

如图所示：



OpenMQ 中的 JMS 与 JMX

我的废话：

为了能够满足强大的 Java 计算环境与分布式系统中管理的问题，Sun 公司对 Java 基础类库进行了扩充开发了专用的 Java 管理扩展类库

(Java Management Extensions)。利用 JMX 的标准可以对 JEE 服务器的资源进行性能监控，管理和统计各类应用服务器运行时的信息。

JMX 中有很多内容，本文只针对如何操作 JMS 服务器 OpenMQ 产品进行阐述，讨论 JMX 层次与架构的相关话题暂时不是本文的重点。



OpenMQ 提供了一套非常完备的 JMX MBeans 管理接口，使用 MBeans 可以对 OpenMQ 服务器进行监控就像我们通常使用 OpenMQ 中的命令 `imqcmd` 一样方便，还记得 JDK 中的 JConsole 工具吗？可以对 Java JDK 运行的状态进行监控和管理。在之前的文章里我提到可以通过 `imqcmd` 命令对 OpenMQJMS 服务器进行管理监控，但客户提出要一个 web 界面下管理多台集群下的 JMS 服务器工具，并且还需要和现有的系统管理工具进行集成，此时还不得不去使用 JMX MBeans 对 OpenMQ JMS 服务器进行管理监控，因为我们找不到比这个更好的办法。

可以使用 JEE 规范中的 JMX 协议进行管理 OpenMQ 中的集群管理节点(managing brokers)、服务(admin/jms)、消息队列(destinations), 接收者(consumers), 生产者(producers), 还可以 JMS 服务器监控连接状态，查看发送的消息内容。

我们可以通过 OpenMQ 提供的 `imqjmx.jar` API 编写代码调用 JMX 对 OpenMQ 服务器进行性能监控，在这个基础上还可以对 JMS 服务器的配置进行优化，还可以写一个 OpenMQ JMX 的客户端对服务器进行自动化的监控和自动维护，例如，发现某个消息队列堵塞消息可以立刻通知管理员，或者发现堵塞消息后可以采用其他方式将消息拿下来再次发送一遍，使用这样的方式可以比原来的 `imqcmd` 命令对 JMS 服务器进行管理和监控更进一步。

下面我们来看看通过 OpenMQ 提供的 API 结合下面的 Java 代码可以将正在运行的 JMS 服务器停止，代码如下：

```
AdminConnectionFactory acf = new AdminConnectionFactory();           // #1
acf.setProperty(AdminConnectionConfiguration.imqAddress,
"192.168.1.1:7677"); // #2
```

```

JMXConnector jmxc = acf.createConnection("admin", "admin");      // #3
MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();  // #4
ObjectName serviceConfigName =
MQObjectName.createServiceConfig("jms");                       // #5
mbsc.invoke(serviceConfigName, ServiceOperations.PAUSE, null,
null);                                                          // #6
// mbsc.invoke(serviceConfigName, ServiceOperations.RESUME, null,
null);                                                          // #6
jmxc.close();                                                  // #7

```

执行以上代码需要引入 `imqjmx.jar` 这个就是通过 JMX 调用 OpenMQ 的 API，这个文件在 OpenMQ 服务器的 lib 文件下面，也就是说在运行的 OpenMQ 服务器上肯定有这个 jar 包，不要再去 google 或者去 sun 网站下载 jar 包了。

#1 首先我们初始化一个管理的连接工厂，`AdminConnectionFactory` 这个程序在 API 的 jar 包中，

#2 写入被管理的 OpenMQ 的服务器地址和端口号，例如 `127.0.0.1:7676`

#3 还需要写上 OpenMQ 服务器的管理员用户名和密码，

#4 开始通过 MBean server 连接 OpenMQ 服务器

#5 指定一个 OpenMQ 服务器上的服务名称，默认名称是 `jms`，

#6 对 OpenMQ 服务器进行具体的操作，`ServiceOperations.PAUSE` 和 `ServiceOperations.RESUME` 是对服务器执行的 2 种不同的命令

#7 关闭连接

如图所示，第一行第 3 个字段中 出现了 “PAUSED” 的字样，说明刚刚执行的程序生效了：



来看另外一个示例，通过 OpenMQ 的 JMX API 获取某个队列中生产者最大的连接数量，

```
AdminConnectionFactory acf = new AdminConnectionFactory();           // #1
acf.setProperty(AdminConnectionFactoryConfiguration.imqAddress,
"192.168.1.100:7676"); // #2
JMXConnector jmxc = acf.createConnection("admin", "admin");         // #3
MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();      // #4
ObjectName destConfigName=
MQObjectName.createDestinationConfig(DestinationType.QUEUE, "sms"); // #5
Integer attrValue=
(Integer)mbsc.getAttribute(destConfigName, DestinationAttributes.MAX_NUM_PRODUCERS); // #6
System.out.println("Maximum number of producers: " + attrValue);
jmxc.close(); // #7
```

#1 和 #2 填写一些必要的连接参数，例如，连接地址、端口和用户名、密码

#3 开始创建一个 JMX 的连接

#4 得到了一个到 OpenMQ 的 MBean 服务器的连接

#5 获得 Queue 属性的 smsQueue 消息队列上的属性

#6 通过 DestinationAttributes.MAX_NUM_PRODUCERS 命令查看 sms 消息队列上的生产者最大的连接数，

#7.关闭连接

如图所示：



OpenMQ 命令

整理了一些查看 OpenMQ JMS Server 的 命令，仅供自己参考，学的东西太多，怕忘，先放这里记着。

首先创建一个 password 的文件，这样就不需要每次都 输入密码了，怎么创建，password 文件里面的内容，自己看看 OpenMQ imqcmd 里面的命令也就知道了，这里就不多说了。

查看当前 service 的运行状态，默认名称是 jms

```
/opt/mq/bin/imqcmd -u admin -passfile pass -b 127.0.0.1:7677 -t q -n jms query svc
```

不断的跳动，显示状态

```
/opt/mq/bin/imqcmd -b 127.0.0.1:7677 -t q -n imagent metrics dst
```

同上

```
imqcmd -b 127.0.0.1:7677 metrics bkr -m rts -int 1 -u admin -passfile pass
```

查看整个服务器的运行属性

```
imqcmd -u admin -passfile pass -b 127.0.0.1:7677 query bkr
```

查看消息队列中的消息

```
/opt/mq/bin/imqcmd -u admin -passfile pass -b 127.0.0.1:7677 list msg -t q -n
```

```
imagent -nocheck
```

查看消息队列中的连接者

```
/opt/mq/bin/imqcmd -u admin -passfile pass -b 127.0.0.1:7677 list cxn
```

查看整个队列状态

```
/opt/mq/bin/imqcmd -u admin -passfile pass -b 127.0.0.1:7677 list dst
```

查看某个队列的状态

```
/opt/mq/bin/imqcmd -b 127.0.0.1:7677 -t q -n agent
```

查询某个消息在某个队列中的状态

```
imqcmd -b 127.0.0.1:7677 query msg -t q -n agent -msgID
```

```
"ID:1625-127.0.0.1(ef:f5:f4:5c:46:69)-60828-1282050952257" -nocheck
```

删除某个队列中的消息

```
imqcmd -b 127.0.0.1:7677 destroy msg -t q -n agent -msgID
```

```
"ID:205-127.0.0.1(83:c2:1d:63:77:b1)-44516-1282050052264" -nocheck
```

启动 共享 连接

```
/opt/mq/bin/imqbrokerd -tty -name myBroker -port 7677 -cluster
```

```
10.101.102.180:7677,10.101.102.179:7677
```

```
-Dimq.cluster.masterbroker=10.101.102.179:7677 -Dimq.jms.max_threads=10240
```

```
-Dimq.jms.threadpool_model=shared -vmargs "-Xms6144m -Xmx6144m"
```

启动 连接

```
/opt/mq/bin/imqbrokerd -tty -name myBroker -port 7677 -cluster
```

```
10.101.102.221:7677,10.101.102.235:7677
```

```
-Dimq.cluster.masterbroker=10.101.102.221:7677 -Dimq.jms.max_threads=10240
```

```
-Dimq.jms.threadpool_model=shared -vmargs "-Xms6144m -Xmx6144m"
```

Sun OpenMQ Topic 消息收/发 —Tips

Story1

现象

项目需要采用 JMS 消息中间件 OpenMQ 来发送、接收一对多消息。前期需要进行对性能进行模拟测试，开始测试时发送端没有问题循环 1000 次发送 Topic 消息顺利结束，但是 Topic

接收端表现异常，间隔 50 个一次的接收消息进行处理，如果把 1000 个消息处理完成大概需要 30 分钟，更别说下面还要启用 100 个线程发送 1000 次了。

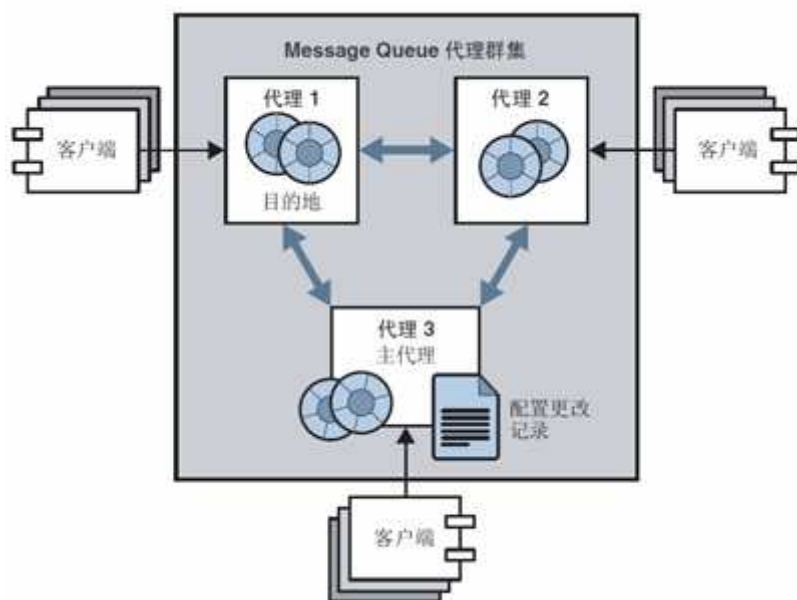
环境

3 台 JMS 消息中间件 OpenMQ 集群，Linux

1 台消息发送端，Windows

1 台消息接收端，Windows

如图所示：



代码

发送端代码 (下载 url)

接收端代码 (下载 url)

需要的 Jar 包



经过几番测试，猜测问题多数是由于接收端与服务器端连接的配置而导致的，仔细检查发现多写了几行代码，接收端验证的部分，也就是说对于 JMS 的消息收发加上验证是对效率有影响的，去掉用户名、密码验证一切正常。

Story2

100 个线程同时发送 100 个消息，一共 10w 个消息，运行到一半的时候出现

```
[29/Mar/2010:03:34:07 EDT] [B1089]: In low memory condition, Broker is attempting to free up resources
```

```
[29/Mar/2010:03:34:07 EDT] [B1088]: Entering Memory State YELLOW from previous state GREEN - allocated memory is 151214K, 80% of total memory used
```

显然在说内存不够。经过 google 以后证明我的想法是正确的，参考文档：

http://docs.sun.com/app/docs/doc/819-4467/6n6k98bq2?l=zh_tw&a=view#aeokn

http://docs.sun.com/app/docs/doc/819-4467/6n6k98bqa?l=zh_tw&a=view

加大运行内存，命令如下：

```
nohup imqbrokerd -tty -name myBroker -port 6769 -cluster
```

```
172.16.2.214:6769,172.16.2.215:6769 -D"imq.cluster.masterbroker=172.16.2.215:6769"
```

```
-vmargs "-Xms256m -Xmx1024m" &
```

话外音

1. 可以用 `imqcmd metrics bkr -m cxn -b 127.0.0.1:6769` 命令查看 JVM 空闲值，最大值，最小值和当前 OpenMQ 的运行状态。

2. 对于消息发送时还需要注意客户端的消息确认模式一共有 3 种客户机确认模式，根据需要的不同级别的处理和带宽开销进行选择：

AUTO_ACKNOWLEDGE 模式的开销最大，可以保证消息逐条传送的可靠性；

CLIENT_ACKNOWLEDGE 模式按批次发送确认，因此需要的带宽开销较小；

DUPS_OK_ACKNOWLEDGE 模式的开销最小，但允许重复传送消息。

Linux 下 OpenMQ 集群的启动异常

今天需要部署 2 台 Sun OpenMQ 集群在的真实生产环境中，同事 GuoHai.Yang 在部署的过程中遇到一些小问题，之后解决了，现在拿来和大家分享一下。

首先和往常一样先从 GlassFishV3 目录下 Copy MQ 出来到另外一台机器上让 OpenMQ JSM 消息服务独立运行，远程 copy 完成，然后键入命令

```
nohup /opt/mq/bin/imqbrokerd -tty -name myBroker -port 6000 -cluster  
10.101.101.223:6000,101.101.224:6000  
-D"imq.cluster.masterbroker=10.101.101.223:6000" &
```

现如下错误异常信息：

```
"Invalid broker address for this broker to run in cluster: Loopback IP address is not  
allowed in broker address
```

```
mq://127.0.0.1:6000/?instName=myBroker&brokerSessionUID=7340910598241782272  
for cluster"
```

分析了一下异常的信息，开始动手：

1.查看 linux 主机名

键入 hostname 命令

2.修改 /etc/hosts 文件

ip (按下 tab 键) 主机名 写入你的 ip 地址和主机，这一步非常重要，因为问题就是没有指定主机名导致的启动错误。

3.你可以重启，也可以不重启

```
source /etc/hosts
```

4.启动 OpenMQ 服务

```
nohup /opt/mq/bin/imqbrokerd -tty -name myBroker -port 6000 -cluster  
10.101.101.223:6000,101.101.224:6000  
-D"imq.cluster.masterbroker=10.101.101.223:6000" &
```

另外，说明一下启动 OpenMQ JMS 集群 命令 的含义

myBroker 实例名称

-port 6000 本机实例的端口号

-cluster 10.101.101.223:6000,101.101.224:6000 表示 2 个被集群进来的机器地址和端口号

-D"imq.cluster.masterbroker=10.101.101.223:6000" 集群的主管理节点

GlassFish JMS 集群

Story

最近项目中一直在使用 GlassFish 作为应用服务器,使用到 GlassFish 中 JMS/MDB(消息驱动 Bean)的部分, 项目对 JMS 服务的要求比较高不仅仅需要在单台服务器上运行, 还需要在集群环境运行 JMS 服务。

查阅大量资料, 网上 N 多讲的都是 Jboss 的 JMS 集群, 很少有简述 GlassFish 集群 JMS 的, 无意找到了一篇印度阿三的写的文章,经过反复推敲, 证明这样的实施方案虽然是 SUN

公司比较推崇的，实施起来成本比较大，将来需要投入的维护成本也不小。印度阿三提出的做法给人感觉：要吃块牛肉而已，现在 牵头牛过来给我了。请看原文 。

将此方法放弃，直接使用 OpenMQ JMS 服务，OpenMQ 也是 SUN 的产品，只不过现在和 GlassFish 整合包含在 GlassFish 里面，也可以独立使用。

在项目测试环境中运行了 OpenMQ 的集群，没有用 GlassFish 里面提供的 MDB 解决方案，欢快的使用着 OpenMQ JMS 集群功能，2 台机器上的 JMS 队列也在欢快的 进行消息同步。

不久后问题来了。。。。。

OpenMQ 和我们自己写的客户端程序建立的是 socket 连接，也就是说服务端 OpenMQ JMS 服务一旦 down 掉，那么 JMS 客户端程序就跟白痴一样还在等待接收消息，却不知道 OpenMQ JMS 服务端已经 down 掉了。系统中没有体验到失效转发。

项目组的一位仁兄 写了一个程序 通过轮询检测 我们自己写的 JMS 客户端程序 是否抛出异常判定 OpenMQ JMS 服务端是否还活着。

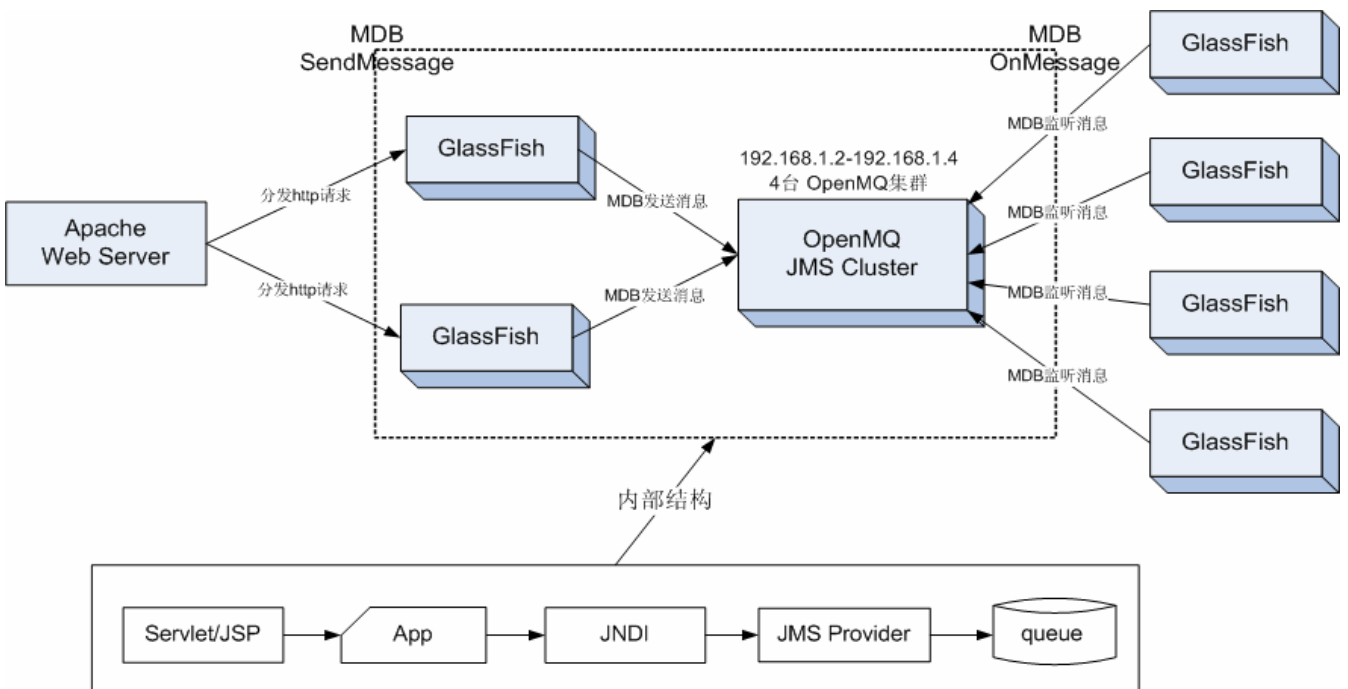
嗯，这位仁兄的方法果然有效 原先我们自己写的客户端程序不是呆子了，智能了一些，知道服务端 down 掉以后该做什么。

这样的方法虽然达到了效果，但是比较简陋，性能上有种说不出的痛。不知道将来会出现什么问题，还是需要依靠自己去解决，会有一定程度的风险。

How

接下来的几天,我 google 到一份讲述 J2EE 服务器原理的书籍,通过理论上的知识让我知道,通过 J2EE 容器发送 JMS 消息,跟写通过容器中 JNDI 整合 JDBC 方式向数据库做操作一样。首先要查找数据源,建立数据库连接。JMS 程序也是一样。可以通过 API 调用 JMS 服务器直接操作,也可以通过容器 JNDI 的方式操作。和 JDBC 的区别就是分为 发送者/接受者 (Queue), 发送者/订阅者 (Topic)。

那么换个思维方式,也就是说在 GlassFish 中配置一个 JNDI 作为一个别名,实际服务器的目标地址可以配置成本地/远程 JMS 服务器,可以是一个也可以是多个服务器。很多事情就可以交给 GlassFish 容器帮我们去做了,比如:超时重连,切换失效主机,事务等。这样可以对项目进行重新架构,详见下图:



说明:

1.客户端收/发程序 ->部署在容器中 ->容器 JNDI—>JMS Server—>Message

Queue—>Message 。

2.发送者向 JMS 服务器中的消息队列发送 100 个消息 , 通过 JMS 集群 接收端如果有

2 台服务器每台服务器会接收 50 个，有 4 台每台接收 25 个，以此类推，这样达到了压力分载的效果。

3.无论在发送端还是在接收端任意一台服务器 Down 掉，JMS 集群服务器会自动分配负载。

拓展话题: 1.JMS 与 EJB 中 MDB 的关系到底是什么？ 2.SUN 公司推出的 JMS1.0 和 JMS1.1 在功能上、标准上有什么区别？

口水: “没有 100%最佳的实践方案，往往最佳实践就是最折中的一种方法。”

参考资料:

<http://www.novell.com/documentation/extend52/Docs/help/MP/jms/admin/clustering.html>

参考资料: <http://www.wnetw.net/jclub/technology/read.jsp?itemid=762>

参考资料: <http://today.java.net/article/2008/01/18/jms-messaging-using-glassfish>

参考资料: <http://developers.sun.com.cn/Java/jms-messaging-using-glassfish.html>

Spring 集成 JMS OpenMQ

前端时间采用 JMS API 直接访问 OpenMQ JMS 服务器会出现一个现象，当 JMS 服务器 down 掉以后或者重启以后，JMS 的接收端将无法工作，如果将程序改成 MDB 的方式将违背了我们的初衷，所以采用一个相对折中的办法，采用 Spring 整合 JMS OpenMQ。

不仅可以解决我们现在存在的问题，并且有以下优势：

- 1、占用资源少，对硬件配置要求低
- 2、部署简单、灵活，不限制于某种特定的 J2EE 容器
- 3、依托微容器管理扩展性强

代码如下：

连接器

```
package com.javabloger.jms;

import java.util.Enumeration;
import java.util.Properties;

import javax.jms.XAConnectionFactory;

public class OpenMqConnectionFactory {

    private Properties props;
```

```

public void setProperties(Properties props) {
    this.props = props;
}

public XAConnectionFactory createConnectionFactory(){
    com.sun.messaging.XAConnectionFactory cf =
        new com.sun.messaging.XAConnectionFactory();
    try{
        Enumeration<?> keys = props.propertyNames();
        while (keys.hasMoreElements()) {
            String name = (String)keys.nextElement();
            String value = props.getProperty(name);
            cf.setProperty(name, value);
        }
    } catch (Exception e){
        throw new RuntimeException(
            "MQConnectionFactoryFactory.createConnectionFactory() failed: "+
            e.getMessage(), e);
    }
    return cf;
}
}

```

配置文件

```
<?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:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <bean id="connectionfactoryfactory"
        class="com.javabloger.jms.OpenMqConnectionFactory">

    <property name="properties">

      <props>

        <prop
          key="imqAddressList">192.168.20.210:7677,192.168.20.211:7677</prop>

      </props>

    </property>

  </bean>

</beans>
```

```
</bean>
```

```
<bean id="mqConnectionFactory" factory-bean="connectionfactoryfactory"  
    factory-method="createConnectionFactory" />
```

```
<bean id="testmq" class="com.sun.messaging.Queue">  
    <constructor-arg type="java.lang.String" value="testmq" />
```

```
</bean>
```

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">  
    <property name="connectionFactory" ref="mqConnectionFactory" />  
    <property name="defaultDestination" ref="testmq" />  
    <property name="receiveTimeout" value="20000" />
```

```
</bean>
```

```
<bean id="messageListener1"  
    class="org.springframework.jms.listener.adapter.MessageListenerAdapter">  
    <constructor-arg>  
        <bean class="com.javabloger.jms.SimpleMessageListener" />  
    </constructor-arg>  
    <property name="defaultListenerMethod" value="receive" />  
    <property name="messageConverter">  
        <null />
```

```
</property>
```

```
</bean>
```

```
<bean id="consumercontainer"
```

```
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
```

```
  <property name="connectionFactory" ref="mqConnectionFactory"/>
```

```
  <property name="destination" ref="testmq"/>
```

```
  <property name="messageListener" ref="messageListener1"/>
```

```
  <property name="transactionTimeout" value="180000"/>
```

```
  <property name="receiveTimeout" value="180000"/>
```

```
  <property name="sessionTransacted" value="true" />
```

```
</bean>
```

```
<!--
```

```
<bean id="jmsContainer"
```

```
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
```

```
  <property name="connectionFactory" ref="mqConnectionFactory" />
```

```
  <property name="destination" ref="testmq" />
```

```
  <property name="messageListener" ref="messageListener1" />
```

```
</bean>
```

```
-->
```

```
</beans>
```

接收端

```
package com.javabloger.jms;

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

public class SimpleMessageListener implements MessageListener {

    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            try {
                System.out.println("Received message: "+
                    ((TextMessage) message).getText());
            }
            catch (JMSEException ex) {
                System.out.println(
                    "SimpleMessageListener.onMessage(): got exception: "+ex.getMessage());
                ex.printStackTrace();
                throw new RuntimeException(ex);
            }
        }
        else {
```

```
        throw new IllegalArgumentException(
            "MessageListener.onMessage(): Message must be of type TextMessage");
    }
}
}
```

启动 Spring 与 JMS 集成的客户端代码

```
package com.javabloger.jms;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MessageConsumer {

    public static void main(String[] args) {

        ApplicationContext context = new
ClassPathXmlApplicationContext("jms.xml");

        System.out.println( context.getId() );

    }
}
```

发送端代码

```
package com.javabloger.jms;

import javax.jms.Connection;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import com.sun.messaging.ConnectionConfiguration;
import com.sun.messaging.ConnectionFactory;
import com.sun.messaging.Queue;

public class QueueSender {

    /**
     * @param args
     * @throws JMSException
     */
    public static void main(String[] args) throws JMSException {

        ConnectionFactory myConnFactory;

        myConnFactory = new com.sun.messaging.ConnectionFactory();

        myConnFactory.setProperty(ConnectionConfiguration.imqAddressList, "mq:/
```

```

/192.168.20.211:7677");

    myConnFactory.setProperty(ConnectionConfiguration.imqReconnectEnable
d, "true");

    Connection myConn = myConnFactory.createConnection();

    myConn.start();

    //Step 4:

    //Create a session within the connection.

    Session mySess = myConn.createSession(false, Session.AUTO_ACKNOWLEDGE);

    Queue myTopic=new com.sun.messaging.Queue("testmq");

    MessageProducer myMsgProducer = mySess.createProducer(myTopic);

    TextMessage myTextMsg = mySess.createTextMessage();

    for (int i=0;i<10;i++){

        myTextMsg.setText("Queue Msg ID: "+i+" "+ new java.util.Date() );

        System.out.println( myTextMsg.getText());

        myMsgProducer.send(myTextMsg);

    }

    mySess.close();

    myConn.close();

}

}

```

GlassFish OpenMQ JDBC

OpenMQ 是一个开源的消息中间件，类似 IBM 的 WebSphere MQ(MQSeries)，现在被集成到 GlassFish 和 OpenESB 的高质量且开放源代码的 JMS 应用中，网上关于 OpenMQ 配置成 JDBC 存储方式的**中文文章**比较少，而且官方英文文档讲述的也不够直观，所以 Javabloger 在此写一篇关于 OpenMQ 是怎么配置成 JDBC 存储方式的。

1.修改 x:\glassfish\domains\domainEE\imq\instances\imqbroker\props 路径下的 config.properties 文件

内容如下：

```
imq.instanceconfig.version=300  
imq.persist.jdbc.mysql.user=root  
imq.persist.jdbc.password=www.javabloger.com  
imq.persist.jdbc.dbVendor=mysql  
imq.brokerid=_broker  
imq.persist.jdbc.mysql.property.url=jdbc\:mysql\://127.0.0.1/test  
imq.persist.jdbc.mysql.needpassword=true  
imq.jms.max_threads=1000  
imq.persist.store=jdbc  
imq.message.expiration.interval=90
```

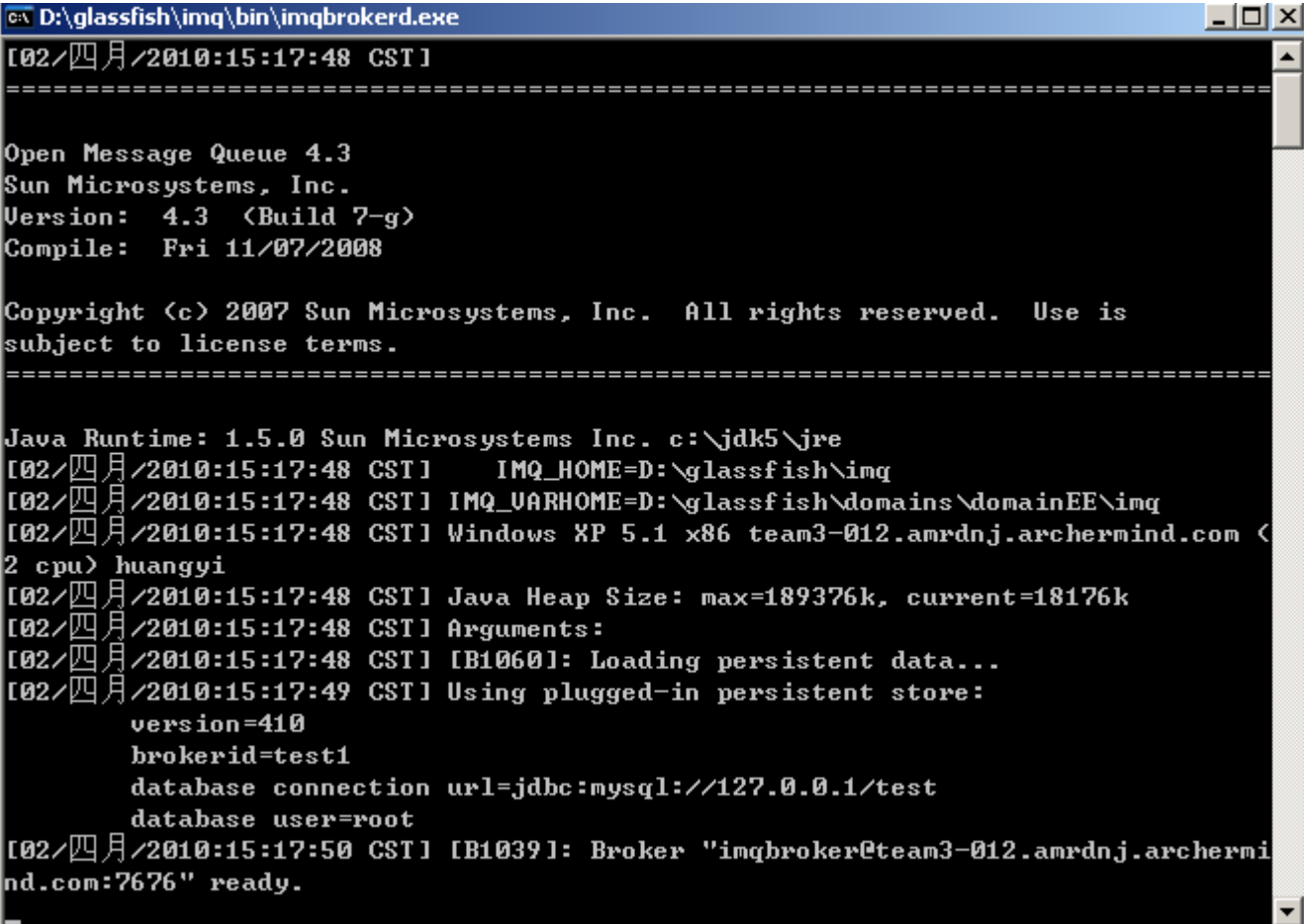
2.复制 JDBC 驱动到指定目录下

将 JDBC 驱动复制到 D:\glassfish\domains\domainEE\lib\ext 目录下

3.启动 Sun OpenMQ JMS 服务

运行 x:\glassfish\imq\bin\imqbrokerd.exe

4.运行成功，如图所示：



```
C:\D:\glassfish\imq\bin\imqbrokerd.exe
[02/四月/2010:15:17:48 CST]
=====
Open Message Queue 4.3
Sun Microsystems, Inc.
Version: 4.3 <Build 7-g>
Compile: Fri 11/07/2008

Copyright (c) 2007 Sun Microsystems, Inc. All rights reserved. Use is
subject to license terms.
=====

Java Runtime: 1.5.0 Sun Microsystems Inc. c:\jdk5\jre
[02/四月/2010:15:17:48 CST] IMQ_HOME=D:\glassfish\imq
[02/四月/2010:15:17:48 CST] IMQ_VARHOME=D:\glassfish\domains\domainEE\imq
[02/四月/2010:15:17:48 CST] Windows XP 5.1 x86 team3-012.amrdnj.archermind.com <
2 cpu> huangyi
[02/四月/2010:15:17:48 CST] Java Heap Size: max=189376k, current=18176k
[02/四月/2010:15:17:48 CST] Arguments:
[02/四月/2010:15:17:48 CST] [B1060]: Loading persistent data...
[02/四月/2010:15:17:49 CST] Using plugged-in persistent store:
    version=410
    brokerid=test1
    database connection url=jdbc:mysql://127.0.0.1/test
    database user=root
[02/四月/2010:15:17:50 CST] [B1039]: Broker "imqbroker@team3-012.amrdnj.archermi
nd.com:7676" ready.
```