目　录

第1章　JMS 基本概念

1.1　什么是 JMS（Java Message Service）？

JMS 为 Java 程序提供了一种创建、发送、接收和读取企业消息系统中消息的通用方法。

企业消息产品（有时也被称为面向消息的中间件MOM-Message Oriented Middleware)，正成为一种用来整合公司内部操作的重要组件。它们使得分离的业务组件变成可靠而又灵活性的系统。Java语言编写的客户端以及中间层服务必须能够访问这些系统，JMS 为java语言访问这些消息系统提供了一种通用的方法。

JMS是一系列的接口及相关语义的集合，通过这些接口和语义定义了JMS客户端如何去访问消息系统。

1.2　什么是 JMS 中的消息（Messaging）？

消息这个术语在计算机系统中含义非常广泛，它被用来描述不同的操作系统概念，它被用来描述邮件和传真。**而在这里，它指的是用于企业应用间的异步通讯。**

这里所说的“消息”是指被企业应用而不是人所消费的异步的请求、报告以及事件。消息中包含了重要的用来系统间进行协作的信息。消息中包含了精确的数据格式以描述特定的业务活动，通过应用系统之间的消息交互，使得企业业务过程能够保持一致。

JMS应用由哪些部分组成？

- JMS客户端 ：用来发送和接收消息的Java语言程序。
- 非JMS客户端：这些客户端是用消息系统的本地客户端API编写的，而不是JMS。如果应用先于JMS出来之前，那么它可能会既包括JMS客户端，也包括非JMS客户端。
- Messages（消息）：每个应用定义了用于在客户端之间进行通讯的消息。
- JMS Provider （JMS提供者）：实现了JMS规范的消息系统，该系统还提供必须的用于管理和控制全方位的功能。
- Administered Objects（被管理的对象）：是预先配置的JMS对象，由系统管理员为使用JMS的客户端创建。

1.3 如何实现 JMS 客户端程序的跨平台性？

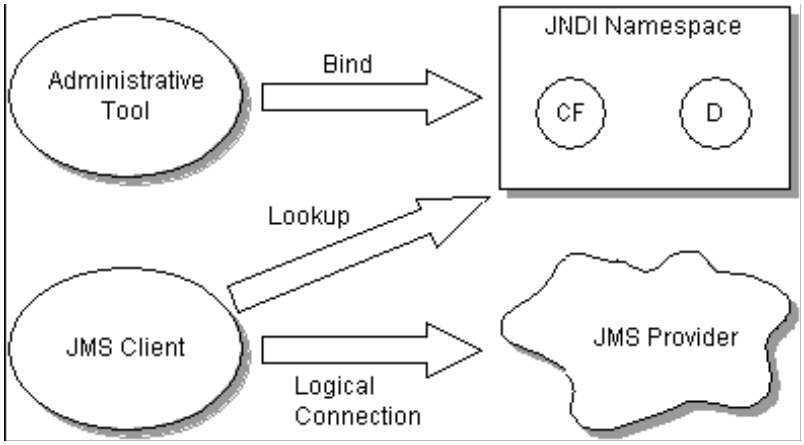
由于有很多 JMS 消息系统，它们的底层实现技术各不相同，比如 Sun MQ, IBM MQ,BEA MQ,Apache ActiveMQ,那么如何使得 JMS 客户端针对这些消息系统编程时能够隔离这些产品的变化而具有跨平台特性呢？那就是通过定义被管理的对象来实现。被管理的对象是由管理员通过使用 JMS 系统提供者的管理工具创建和定制，然后被 JMS 客户端使用。ＪＭＳ客户端通过接口来调用这些被管理的对象，从而具备跨平台特性。

主要有两个被管理的对象：

- ConnectionFactory： 这是客户端用来创建同 JMS 提供者之间的连接的对象。
- Destination ：这个对象是客户端用来指明消息被发送的目的地以及客户端接收消息的来源。

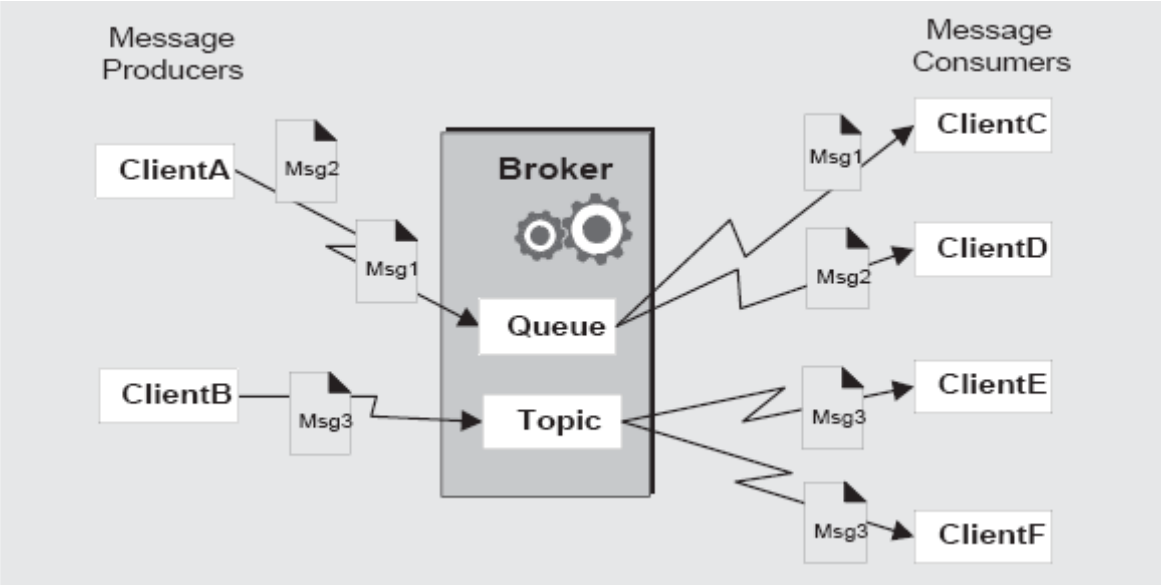
被管理的对象一般被管理员放在 JNDI 名字空间中，通常在ＪＭＳ客户端应用的文档中说明它所需要的ＪＭＳ被管理对象，以及应以何种 JNDI 名字来提供这些ＪＭＳ被管理对象。

下图 JMS 管理的一般工作顺序。



1.4 有关两种消息方式简介

消息应用能使用点对点(PTP)和发布订阅(Pub/Sub)的消息方式，在一个应用中也能混合使用两种消息方式



Clients A 和 B是消息生产者，以两种不同的目的地向Clients C, D, 和E 发送消息：

- ✓ 在clients A, C, 和 D之间的消息是点对点模式，使用这种模式，客户端发送消息到队列目的地，从这个队列里面只有一个消息接收者可以收到那个消息，其他访问同一目的地的接收者不会接收到消息。
- ✓ 在clients B, E, 和F之间的消息是发布 / 订阅模式。使用这种广播模式，一个客户端发送消息给主题目的地，任何数量的消费订阅者可以从这个主题目的地来接收它们。

这两种消息方式通常被称为消息域(messaging domains)。JMS提供这两个消息域，因为它们代表两种常用的消息模式。当使用JMS API的时候，开发者能使用接口和方法来支持这两种消息模式。当使用接口的时候，消息系统的行为可能会有所不同，因为，这两种消息域有不同的语义，稍后会详细介绍两种消息域的语义。

以上两个领域的消费者可以选择同步还是异步获取消息。同步消费者显式调用方法来获取消息，异步消费者指定一个回叫方法来处理消息。

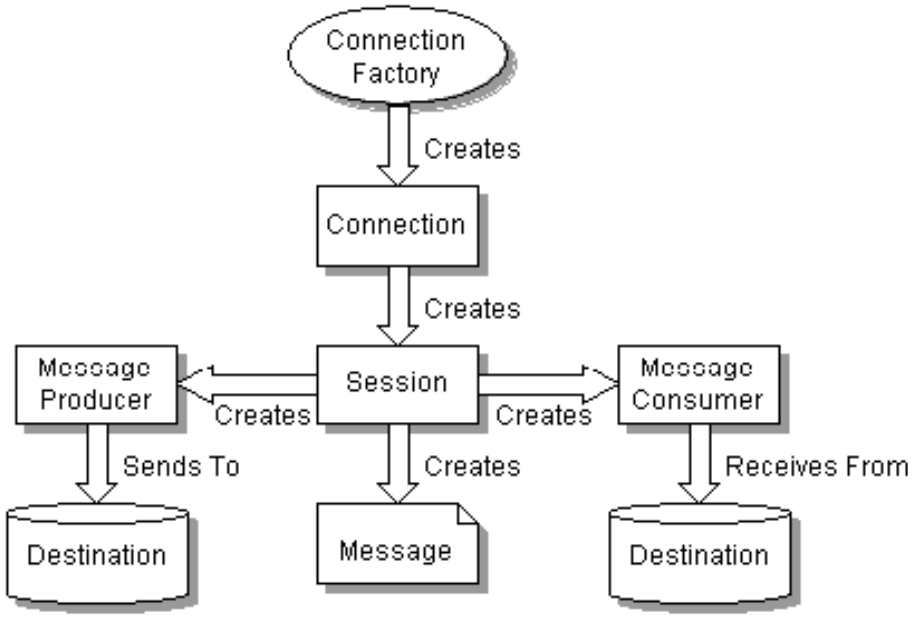
JMS 基于一套通用的消息概念。每个JMS消息域（PTP 和Pub/Sub）也都定义了一套自己概念的接口。JMS 通用接口则提供了不依赖于PTP和Pub/Sub消息域的能力。

JMS Common Interfaces	**PTP-specific Interfaces**	**Pub/Sub-specific interfaces**
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher

MessageConsumer	QueueReceiver QueueBrowser	TopicSubscriber

- ConnectionFactory ：被管理的对象，由客户端使用，用来创建一个连接。
- Connection：一个到JMS消息系统提供者的活动连接。
- Destination ：一个被管理对象，封装了消息目的地的标识。
- Session - a single：一个用来发送和接收消息的单独的线程上下文
- MessageProducer ：一个由Session对象创建的，用来发送消息的到目的地的对象。
- MessageConsumer ：一个由Session对象创建的，用来接收发送到某个目的地的消息的对象。

这些对象之间的如下图所示：



1.5 开发一个基于 JMS 的消息客户端应用的编程步骤

- 使用JNDI查找一个ConnectionFactory对象。
- 使用JNDI查找一个或者多个Destination对象。
- 使用ConnectionFactory创建一个JMS连接
- 使用连接创建一个或者多个JMS Sessions
- 使用Session and Destinations 创建所需的MessageProducers 和MessageConsumers
- 告知Connection 开始传送消息。

1.6 ＪＭＳ有关多线程方面的问题

JMS可能被要求所有它的对象都支持并发使用。由于支持并发访问通常会增加一些难度和复杂性，所以ＪＭＳ设计限定只有那些被多线程客户端可自然而然就共享的对象提供满足并发访问的需求，剩余对象被设计为一次只能有一个逻辑线程访问。下表列出了对JMS对并发访问的支持情况。

ＪＭＳ对象	是否支持并发访问
Destination	是
ConnectionFactory	是
Connection	是
Session	否
MessageProducer	否
MessageConsumer	否

第2章JMS 消息模型

2.1 背景：

企业级的消息产品将消息看做包括一个“头”（header）和一个“体”（body）的“轻量级”条目。“消息头”包含一些用于消息路由和消息识别的字段。“消息体”中包含了被发送的应用数据。在这种通常的格式下，不同的消息产品对于消息的定义会有很大的不同。这些不同主要是“消息头”的内容和语义。一些产品使用自我描述的、规范编码的消息数据，而其他则以完全不透明的方式处理数据。一些产品提供一个库来存储消息的描述，这些描述用于识别和解释消息的内容，而其他的产品则不会。这就使得 JMS 把握这些消息模型的常见冲突的尺度变得非常困难。

2.2 目标：

JMS消息模型目标如下：

- ✓ 提供一个单独，统一的消息API。
- ✓ 提供一个ＡＰＩ用于创建消息能够匹配已有的，非ＪＭＳ应用所使用的格式。
- ✓ 支持跨操作系统、机器结构以及计算机语言的异构应用的开发。
- ✓ 支持包含Java对象的消息。
- ✓ 支持包含可扩展标记语言（XML）页面的消息。

2.3 JMS 消息

JMS 消息包含以下组成部分：

- ➢ **消息头（Header）**：所有的消息都支持一套相同的头字段。头字段包含了客户端和提供者（provider）用来路由和识别消息的数据。
- ➢ **消息属性（Properties ）：**在标准头字段之外提供一种内建的设施用于给消息添加可选的头字段
  - ◆ **应用指定的属性：**提供一种给消息添加应用指定的头字段的机制。
  - ◆ **标准属性** ：JMS定义的一些标准属性，即一些有效的、可选的头字段。
  - ◆ **Provider指定的属性：**在集成JMS客户端和provider 内在客户端时可能需要使用Provider指定的属性，JMS 为这些定义了命名约定。

- **消息体（Body）**：JMS定义了几种类型的消息体，这些消息体覆盖目前常用的几种消息样式。

2.4 消息头字段

下面的各节描述了每个 JMS “消息头”，消息头的完整信息将被传递给所有接收到消息的 JMS 客户端。JMS 没有定义传递给非 JMS 客户端的消息头字段。

2.4.1 JMSDestination

JMSDestination 消息头字段包含了消息被发送到的目的地。当消息被发出的时候，这个字段被忽略。在消息发送结束后，消息持有了由发送方法指定的目的地对象。当一个消息被接收的时候，它目的地对象的值必须与它发出时候赋予的值相等。

2.4.2 JMSDeliveryMode

*JMSDeliveryMode*头字段包含了消息发送时指定的传送模式信息。在消息被发送的时候，这个字段被忽略，当消息发送完成后，它持有了由发送方法所指定的传送模式。

2.4.3 JMSMessageID

The *JMSMessageID* 头字段包含了一个唯一的标识每一条由提供者发出消息的值 。
在消息被发送的时候，*JMSMessageID*被忽略，当send方法返回时，这个字段就包含了一个由提供者赋予的值。*JMSMessageID* 是一个 *String* 值，这个值用做消息在历史库中唯一键值。唯一性的确切范围由提供者定义，但它至少应当覆盖提供者的一个特定安装点上的所有消息。安装点就是被一系列消息路由器连接的地方。
所有的*JMSMessageID*值必须以前缀‘ID:’开头。不必保证消息ID值在跨越不同的提供者时也保持唯一。因为消息ID会增加消息的大小，所以如果JMS提供者被提示不在应用中使用消息ID时，可能会优化消息的开销。JMS产生者提出一个禁止消息ID的提示。当一个客户端设置消息产生者禁止消息ID，这就是说，它不依赖于它所产生消息的ID值。如果JMS提供者接受了这个提示，消息必须设置消息ID为null 。如果消息提供者忽略了这个提示，那么消息ID必须被设置为正常的唯一值。

2.4.4 JMSTimestamp

JMSTimestamp 头字段含有消息被交给提供者去发送的时间。这个时间不是消息实际

被传送的时间。因为事务或其他客户端对消息的排队导致实际传送消息可能会延迟。

在消息被发出的时候 ，*JMSTimestamp* 被忽略。当发送方法返回的时候，这个字段包含了在发送方法被调用和返回时间段之中的一个时间值。这是一个正规Java毫秒时间格式的时间值。由于timestamps对创建消息和消息的大小会产生影响，如果应用给消息提供者一个不使用timestamps的一个提示，那么消息提供者可以优化消息的开销。JMS提供者应提供禁止timestamps的功能。当客户端设置提供者禁止timestamps时，它所产生的消息不依赖于timestamp。如果JMS提供者接受了这个提示，消息中的timestamp值被设置为0。如果消息提供者忽略了这个提示，那么timestamp必须设置为正规的值。

2.4.5 JMSCorrelationID

客户端能用*JMSCorrelationID* 头字段将一个消息同另一个消息相联接。一个典型的用法就是将一个响应消息同它的请求消息相连接。*JMSCorrelationID* 能够持有以下中的一种：

- 提供者指定的消息ID.
- 应用指定的字符串。
- 提供者本地的字节值。

因为每个由JMS提供者发出的消息都被赋予一个消息ID值，这非常便于通过消息ID进行消息之间的连接。所有的消息ID值必须以’ID:’作为前缀。

在某些情况下，应用（由几个客户端组成）需要使用应用指定的值连接消息。例如：一个应用可以使用*JMSCorrelationID*去持有对一些外部信息的引用。应用指定的值不必以’ID:’作为前缀。以“ID:”作为前缀被保留给JMS提供者产生的消息ID 。

如果提供者支持本地概念上的correlation ID（相关联ID），JMS客户端可能需要赋予*JMSCorrelationID*一个能够匹配非JMS客户端要求的值。*byte[]* 值就是用于这个目的。没有本地correlation ID 值的JMS提供者不需要支持*byte[]* 值*。使用*byte[]* 导致应用不具备可移植性。

* 它们的*setJMSCorrelationIDAsBytes()* 和 *getJMSCorrelationIDAsBytes()方法将抛出异常lang.UnsupportedOperationException.*

2.4.6 JMSReplyTo

*JMSReplyTo*头字段包含一个由客户端在发送消息的时候提供的目的地信息 。这个目的地就是回复这个消息的目的地。发出的消息带有一个null *JMSReplyTo* 值可能表示一些事件或者它们只是一些发送者认为别人会感兴趣的数据。带有*JMSReplyTo* 值的消息通常期

望能有一个响应，这个响应是可选的，它由客户端决定。

2.4.7 JMSRedelivered

如果客户端接收的消息带有*JMSRedelivered*指示符设置，那么可能（但不一定），这个消息过去被发送了但是没有被确认。通常，如果消息被再次发送，提供者必须设置*JMSRedelivered*消息头字段。如果这个字段被设置为*true*，这对于消费它的应用来说就必须注意额外的重复处理的问题，因为这个消息过去曾经发送过。 这个头字段在发送时没有任何意义并且被发送方法设置为未赋值。

2.4.8 JMSType

*JMSType*头字段包含了由客户端在发送消息时提供的消息类型标识。一些消息提供者使用消息库来存储由应用发送的消息定义。*type*头字段可以引用提供者库中的消息定义。JMS没有定义一个标准的消息定义库，也没有定义这个库中所包含的各种定义的命名策略。一些消息系统要求每个被创建的应用消息都必须有一个消息类型定义，并且每个消息都指定它的类型。为了能够使JMS工作于这些消息系统提供者，无论应用是否使用，JMS客户端最好赋值*JMSType* ，这样可以保证为需要该头字段的提供者提供了正确的设置。为了保证移植性，JMS客户端应使用安装时在提供者消息库中定义的语义值作为*JMSType*的值。

2.4.9 JMSExpiration

在消息被发送的时候，它的过期时间是发送方法指定的 time-to-live 值加上当前的 GMT 值之和。在发送方法返回时，这个消息的 *JMSExpiration* 头字段就包含了这个值。当消息被接收时，它的 *JMSExpiration* 应含有相同的值。如果 time-to-live 被设置为 0，过期时间被设置为 0，则表明这个消息永不过期。当 GMT 晚于了一个未被发送的消息的过期时间时，这个消息将被销毁。JMS 没有定义消息过期通知，客户端不会接收到过期的消息，但是 JMS 不保证那样的事情不会发生。

2.4.10 JMSPriority

JMSPriority 头字段包含了消息的优先级。在消息被发送的时候，这个字段被忽略，当消息发送完成后，它持有了发送方法指定的值。JMS定义了10级的优先级，0作为最低级，9是最高级。除此之外，客户端可以认为0-4级是普通优先级，而5-9作为加速优先级。JMS不强迫提供者严格实现消息的优先级顺序，但是，最好实现加速消息先于普通消息。

2.4.11 对消息头信息如何被设置的总结

Header Fields	**Set By**
JMSDestination	Send Method
JMSDeliveryMode	Send Method
JMSExpiration	Send Method
JMSPriority	Send Method
JMSMessageID	Send Method
JMSTimestamp	Send Method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	Provider

2.4.12 重载消息头字段

JMS允许管理者配置JMS使得它能够重载客户端指定的*JMSDeliveryMode*, *JMSExpiration*及*JMSPriority*字段的值。如果对消息头字段进行重载，消息头字段值必须反映出管理设定的值。JMS没有定义管理者如何重载这些头字段值。JMS提供者不一定支持这个管理选项。

2.5 消息属性

除了头字段定义的信息以外，*Message* 接口包含了内置的设置来支持属性值。因而，这就提供了一种为消息增加可选头信息的机制。通过消息选择器，属性可以使客户端让JMS提供者按照应用指定的规则选择消息。

2.5.1 属性名

属性名称必须遵循消息选择器标识符的规则。

2.5.2 属性值

属性值可以是 *boolean, byte, short, int, long, float, double,* 以及*String*（Java语言基本类型和String）

2.5.3 使用属性

属性值在消息发送前指定，当一个客户接收到一个消息的时候，它的属性是“只读”模式的。如果一个客户端此时试图设置属性值，就会抛出一个*MessageNotWriteableException* 的异常。属性值可以复制一个消息体的值，也可以不必。尽管JMS没有定义什么应该成为属性，什么不该成为属性的策略，但是应用开发者应当注意JMS提供者可能处理消息体数据的效率要高于处理消息属性数据。为了获得最佳性能，应用应当只在它们确实需要自定义一个消息头的时候才使用消息属性，而自定义消息头的主要原因就是为了支持自定义消息选择。

2.5.4属性值的转换

属性支持以下的转换表。标记的情况必须被支持。未标记的情况必须抛出JMS *MessageFormatException*异常。在字符串向数字转换的时候，如果numeric *的valueOf()*方法不能接受一个字符串的值作为一个正确的表式，必须抛出*java.lang.NumberFormatException* 异常。试图读取一个null值作Java基本类型的值时，必须按照调用基本类型相应的 *valueOf(String)转换方法来处理null值。Row类型的值集合必须按照*column类型对待才能读取。

	boolean	byte	short	int	long	float	double	String
boolean	X							X
byte		X	X	X	X			X
short			X	X	X			X
int				X	X			X
long					X			X
float						X	X	X
double							X	X
String	X	X	X	X	X	X	X	X

2.5.5 属性值作为对象

除了有关属性的指定类型的set/get方法外，JMS提供了 *setObjectProperty/getObjectProperty* 方法。这些方法使用对象化的基本类型值支持相同系列的属性类型。它们的目的是允许在运行时确定属性类型，而不是在编译时。他们支持相

同的属性值转换。*setObjectProperty*方法接受*Boolean, Byte, Short, Integer, Long, Float, Double* 和 *String*类型值。试图使用其他类则必须抛出JMS异常*MessageFormatException*。*getObjectProperty* 方法只返回*null, Boolean, Byte, Short,Integer, Long, Float, Double* 和 *String*。如果指定的属性名不存在，则返回Null值。

2.5.6 属性迭代

没有定义属性值的顺序，要迭代消息的属性值，使用 *getPropertyNames* 去获取属性名称的枚举，然后使用不同属性的 *get* 方法去取得属性值。*getPropertyNames* 方法不返回 *JMS* 标准头字段的名字。

2.5.7 清空消息属性值

消息属性可通过*clearProperties*方法删除，这使消息只剩下空的属性集合。新的属性条目可以被创建和读取。

清空消息属性条目不会清空消息体的值。ＪＭＳ不支持一次只删除一条属性的方法。

2.5.8 不存在的属性

如果用一个名字获得一个还没有该名字属性的属性值时，那么它的处理看起来和该属性已存在了一样，只不过返回一个null值。

2.5.9 JMS 定义的属性

JMS为JMS定义的属性保留了‘JMSX’属性名前缀。全套的JMS定义的属性见下表。新的ＪＭＳ定义的属性可以被增加到JMS最近的版本中。除非另有通知，这些支持是可选的。*ConnectionMetaData.getJMSXPropertyNames()* 返回了所有连接所支持的ＪＭＳＸ属性名。无论JMSX属性是否被连接所支持，这些属性都可以被消息选择器引用。 如果JMSX属性没有出现在消息中，它们被按照所有未存在的属性一样对待。在特定消息中存在的JMS定义的属性是由JMS提供者根据自身如何控制这个属性的用途来进行设置。它可以选择让一些消息中包含这些属性，而在别的消息中去除这些属性，这些取决于管理或者其他规则

Nam	**Type**	**Set By**	**Use**
JMSXUserID	String	Provider on	用于标识发送消息的用户。

		Send	
JMSXAppID	String	Provider on Send	用于识别发送消息的应用。
JMSXDeliveryCount	int	Provider on Receive	这消息发送尝试的次数。第一次是1，然后是2…..
JMSXGroupID	String	Client	用来表示消息是哪个组的一部分。
JMSXGroupSeq	int	Client	表示消息在组内的顺序号第一个消息是1，第二个是2，……
JMSXProducerTXID	String	Provider on Send	事务标识符，用来表示消息是在哪个事务中产生的。
JMSXConsumerTXID	String	Provider on Receive	消息标识符，用来说明消息是在哪个事务中被消费的。
JMSXRcvTimestamp	long	Provider on Receive	消息被发送到消费者时的时间。
JMSXState	int	Provider	设想有一个消息仓库保存每个被发给消费者消息的拷贝，并且这些拷贝从消息开始发出之时就存在。每个拷贝的状态是如下： 1(waiting), 2(ready), 3(expired) or 4(retained). 因为产生者和消费者都不关心状态，所以这个属性只是用于在仓库中查询消息，并且JMS没有为其提供任何的API。

消息的生产者和消费者都能得到由提供者在发送消息时设置的JMSX属性。而提供者在消息接收时设置的JMSX属性只能被消费者得到。如果客户端想对消息进行分组，那么 *JMSXGroupID和 JMSXGroupSeq* 是标准的属性。所有的提供者必须支持这两个属性。这些JMSX属性名称的大小写必须按照上表的定义。除非特别说明，JMSX属性的值和语义是未定义的。

2.5.10 提供者指定的属性

*JMS保留了‘JMS*_<vendor_name>’作为提供者指定的属性名称前缀。每个提供者定义它们自己的<vendor_name>。这就是JMS提供者用来指定它能给JMS客户端带来的特定的消息服务的措施。提供者指定属性目的在于支持使用“提供者本地客户端”所需要的特性。这些属性不应被用于JMS或者JMS消息。

2.6 Message 确认

所有的JMS消息支持*acknowledge*方法，这个方法用于客户端已经指定ＪＭＳ消费者的消息被显示地确认。如果客户端使用自动确认，调用，acknowledge方法将被忽略。

2.7 消息接口

Message 是所有消息的根接口。它定义了 JMS 消息的头字段，属性设施和 acknowledge 方法。

2.8 Message 选择

很多消息应用需要过滤消息或对它们产生的消息进行分类。在消息只发给一个接收者的时候，将消息过滤规则加在消息上能够合理提高效率，使得接收的客户端不理会它不感兴趣的消息。

当消息广播给很多客户端时，将效率规律规则加在消息头上更加有用，以便JMS提供者可以知道这个规则。这使得提供者可以处理这些大量的过滤和路由工作，否则，这些工作就要由应用程序自己去做。

JMS提供了这种机制让客户端将“消息选择”代理给它们的JMS提供者。这简化了客户端的工作，并且让JMS提供者减少了因发送给客户端不需要的消息而浪费的时间和带宽。

客户端通过使用消息属性来制定应用指定的消息选择规则。客户端通过消息选择器表达式来指定消息的选择规则。

2.8.1 Message 选择器

消息选择器可由客户端通过消息头指定它所感兴趣的消息。只有消息头和属性能够匹配的消息才能传送给客户端。根据消息消费者的不同，“不传送”的语义会有一点不同。参见“QueueReceiver”和“TopicSubscriber”来获取详细信息。消息选择器不涉及消息体

的值。用消息头字段和属性值替换消息选择器中的标识符后，如果求值为true的，表示消息同消息选择器相匹配。

2.8.2 消息选择语法

消息选择器是一个字符串，这个字符串的语法是基于条件表达式语法SQL92*的一个子集。 如果消息选择器的值是一个空字符串，这个值被按照null处理，并且说明没有为消息消费者设定消息选择器。消息选择器的取值顺序是从左到右，括号能够改变这个顺序。在这里用大写来表示预先定义的选择器的文字和操作符。但是，他们实际上是大小写无关的。一个选择器包括如下：

- Literals:文字
  - 字符串文字由一个单引号括起来表示，字符串中的单引号用两个单引号表示。例如：‘literal’ and ‘literal’’s’.如同Java *String*文字一样，都使用Unicode字符编码。
  - 精确的数字文字是不带有小数点的数字值，例如57, -957, +62;支持Java long的取值范围。
  - 近似的数字文本是一个使用科学技术法表示的值，例如： 7E3 and -57.9E2，或者带有小数的数字，例如7., -95.7, and +6.2;支持Java *double*的取值范围。近似文字使用Java浮点数文字语法。
  - 布尔文字是*TRUE* 和 *FALSE*.
- Identifiers:标识符
  - 标识符是没有长度限制的字符串，必须以Java标识符字符开头。所有允许的字符必须是Java标识符允许的字符。标识符的开始字符是任何Character.isJavaIdentifierStart返回为true的值。这包括‘_’ 和‘$’。表示符的组成字符是任何Character.isJavaIdentifierPart返回为true的字符。
  - 标识符不能是*NULL*, *TRUE*, 或 *FALSE*.
  - 标识符不能是*NOT*, *AND*, *OR*, *BETWEEN*, *LIKE*, *IN*, *IS*, 或者*ESCAPE*.
  - 标识符可以是头字段或者属性的引用，消息选择器中的属性值类型应该与设置属性时使用的类型一致。如果消息中不存在的属性被引用了，它的值是NULL。在消息选择器中求NULL值的语义在后面的“Null Values.”中详细描述。
  - 属性get方法中的转换不适用于消息选择器表达式，例如：把一个属性设置为一个字符串值，如下：

```
myMessage.setStringProperty("NumberOfOrders", "2");
```

下面选择器表达式的取值为false,因为字符串不能被用于数学表达式。

```
"NumberOfOrders > 1"
```

- 标识符是大小写敏感的。
- 可引用的消息头字段仅限于： *JMSDeliveryMode,JMSPriority, JMSMessageID, JMSTimestamp, JMSCorrelationID,* and*JMSType.*而且 *JMSType* 值可以是*null，如果 JMSType* 值是null,那么被当作NULL处理。
- 任何以‘JMSX’开始的名字都是JMS定义属性名。
- 任何以‘JMS_’开头的名字都是提供者指定的属性名。
- 任何不是以‘JMS’开头的名字都是应用指定的属性名。

■ 空格同Java中定义的一样:空格，水平tab，制表符和行结束符。

■ • Expressions 表达式

- 选择器是条件表达式，选择器表达式求值为true表示匹配，为false 或者unknown表示不匹配。
- 数学表达式由数学操作符，带有数字值的标识符和数字文本组成。
- 条件表达式由比较操作符，逻辑操作符，带有布尔值的标识符以及布尔文本组成。

■ 支持用标准的括号()来改变求值顺序。

■ 逻辑操作符的优先顺序为：NOT, AND, OR

■ 比较操作符： =, >, >=, <, <=, <> (不等于)

- 只有相同类型的值可以进行比较，一个例外就是，比较确且的数字和近似数字是正确的（需要的类型转换是由JAVA数字说明（numeric promotion）规则定义的）。如果试图比较类型不一样的值，那么操作的结果就是false。如果两个类型中任何一个值是NULL,那么比较的结果就是unknown.
- *String* 和 *Boolean* 的比较仅限于= 和 <>只有两个字符串包含相同序列的字符时才是相等的。

■ 数学操作符的优先顺序是：

- +, - *(一元操作符)*
- *, / *(乘和除)*
- +, - *(加和减)*
- 数学操作符必须使用Java numeric promotion.

■ arithmetic-expr1 [NOT] BETWEEN arithmetic-expr2 AND arithmetic-expr3(比较操作符)

- *age BETWEEN 15 AND 19”等价于 “age >= 15 AND age <= 19”*

- *age NOT BETWEEN 15 AND 19” 等价于 “age < 15 OR age >19”*

■ identifier [NOT] IN (string-literal1, string-literal2,...) (比较操作符，这里 identifier 是一个String or NULL 值)

- *表达式“Country IN (’ UK’, ’US’, ’France’)”对于Country值为‘UK’返回true,‘Peru’返回false,相当于表达式 ”(Country = ’ UK’) OR (Country = ’ US’) OR (Country = ’ France’)”*
- 表达式 “Country NOT IN (’ UK’, ’US’, ’France’)” 对于*Country值为‘UK’返回* false ， 而‘Peru’则返回true,它等价于表达式 “NOT ((Country = ’ UK’) OR (Country = ’ US’) OR (Country = ’ France’))”
- 如果IN 或者NOT IN 操作的*identifier*是NULL,表达式返回值将是unknown.

■ identifier [NOT] LIKE pattern-value [ESCAPE escape-character] (比较操作符，这里 identifier 有一个String值; pattern-value is 是一个字符串文本，在这个文本中‘_’ 代表任何单个字符； ‘%’ 代表任何字符串,包括空串和所以其他串，可选的escape-character是一个单字符（single-characte）的string文本，它的字符用于规避pattern-value 中‘_’ 和‘%’ 的含义)。

- “phone LIKE ‘12%3’” 当phone是123’或‘12993’的时候返回true,当phone是‘1234’的时候返回false。
- “word LIKE ‘l_se’” 当work是‘lose’返回true， 当‘loose’ 时返回 false。
- “underscored LIKE ‘_%’ ESCAPE ‘\’”当underscored值为‘_foo’返回true,值为‘bar’返回false.
- “phone NOT LIKE ‘12%3’”当phone值为‘123’和 ‘12993’ 返回false，当phone值为‘1234’返回true。
- 如果LIKE 或 NOT LIKE中的标识符是NULL值，那么操作结果为unkonwn。

■ • identifier IS NULL (比较操作符，用来测试空的头字段值或者丢失的属性值)

- “prop_name IS NULL”

• *identifier* IS NOT NULL (比较操作符，用来测试非空的头字段值存在或者数值的存在。)

- “prop_name IS NOT NULL”

当选择器出现的时候，JMS提供者不必验证消息选择器的语法正确性。一个提供错误语法选择器的方法将导致一个JMS InvalidSelectorException异常。JMS提供者也可以在选择器出现时，有选择性地提供一些语义检查。不是所有的语法检查都可在消息选择器出现时能够进行，因为属性类型是未知的。下面的语法选择器选择了消息类型为car且color为blue

且weight大于2500 lbs的消息：

```
"JMSType = 'car' AND color = 'blue' AND weight > 2500"
```

2.8.3 Null Values 空值

如上所述，头字段和属性值可以为NULL。 包含NULL值的选择器求值遵循SQL92 NULL语义的定义。现在简单描述这些语义。ＳＱＬ将NULL值看作unknown。带有unknown值的比较或者数学操作通常得到unknown结果。IS NULL 及 IS NOT NULL 操作符将unknown的头或者属性值转化为相应的TRUE 和FALSE值。布尔操作符使用按照下表所定义的“三值”逻辑：

操作符的定义

AND	**T**	**F**	**U**
T	T	F	U
F	F	F	F
U	U	F	U

OR 操作符定义

OR	**T**	**F**	**U**
T	T	T	T
F	T	F	U
U	T	U	U

NOT 操作符定义

NOT	
T	F
F	T
U	U

2.8.4 特别说明.

当*JMSDeliveryMode*被用在消息选择器中时，它被按照有‘PERSISTENT’ 及‘NON_PERSISTENT’值处理。

日期(Date)和时间(Time)值使用标准的long 型Java毫秒值。当一个日期或者时间文本被包含在一个消息表达式中时，它应当是一个毫秒的整数文本。产生毫秒值的标准的方式是使用*java.util.Calendar*。尽管ＳＱＬ支持固定的小数比较或者数学计算，但JMS 消息选择器并不支持，这也是仅支持那些没有小数的精确的数字文本的原因（并且数字与小数相加是表示近似数的另一种方法）。

SQL注释被支持。

2.9 访问被发送的消息

在消息被发送后，客户端可以保留并且更改这个消息而不影响已被发出的消息。相同的消息对象可以被多次发送。在执行sending方法期间，消息不能被客户端更改，如果它被更改了，那么发送结果就是未定义。

2.10 改变已接收的消息的值

当消息被接收到时，它的头字段的值能被改变。但是它的属性和它的body内容是只读的。正如在本章中提到的，只读限制的原因是它给JMS providers在如何实现接收到消息的管理方面更多的自由。比如，它们返回了一个消息对象，这个消息对象引用了内部的消息队列中的属性条目和Body值，而不是被强迫实现一个拷贝。

在调用*clearBody* 或*clearProperties*方法之后使得*body*或者属性可写，从而消费者能够更改接收到的消息。如果消费者更改了接收的消息，并且消息然后被再次传送，那么被再次传送的消息必须是原始的，未被改变的消息（作为重新传送的结果，除了被JMS provider 更改的头和属性之外，例如*JMSRedelivered* 头和*MSXDeliveryCount*属性，其他属性和头信息都不能被更改）。

2.11 JMS 消息体

JMS提供五种格式的消息体。每种格式都通过一个消息接口定义。

• StreamMessage – 消息体包含了一个Java primitive 流，这个流被顺序地填充和读取。

• MapMessage – 消息体包含了一系列的名字-值对。名字是*Strings，而值则是*Java primitive类型。消息体中的条目可以被enumerator按照顺序访问，也可以自由访问。条目的顺序没有定义。

• TextMessage – 消息体包含了一个*java.lang.String*. 包含这种这种消息类型是考虑到String消息将被广泛使用的前提。另外一个原因就是XML将可能变成一种用来表示JMS消息内容的主流机制。

• ObjectMessage –消息包含了一个可序列化的Java对象.如果需要Java对象集合（collection），可以是JDK 1.2提供的集合类型中的任何一种。

• BytesMessage – 消息包含了一个不间断的字节流。这个消息类型是用来以文字方式编码一个消息体以匹配存在的消息格式。在很多种情况下，它可能被用于以下用途：*自定义的消息类型，尽管JMS允许使用带有字节的消息属性，但它们通常不能被使用，因为包含的属性可以影响格式。*

2.11.1 清空消息体

*clearBody*方法重置消息体为*'empty'*初始消息值，如同这个消息类型被*Session*提供的create方法创建时一样。清空消息体不会清空它的属性条目。

2.11.2 “只读消息体”

当消息被接收到的时候，消息体是只读的。如果试图改变消息体。*MessageNotWriteableException*将被抛出。如果它的消息体被清空，那么消息同它被创建时是一样的。

2.11.3 由 StreamMessage 和 MapMessage 提供的转换功能

StreamMessage和MapMessage都支持相同的primitive数据类型。类型可通过使用每种类型的方法显式地读写。例如，调用*MapMessage.setInt(“foo”, 6)*方法等价于*MapMessage.setObject(“foo”, newInteger(6))*。两种方法都被提供是因为显示格式便于静态编程，并且当类型在编译时不知道的时候就可以使用对象格式。

*StreamMessage*和 *MapMessage* 都支持下面的转换表。打标记的情况必须被支持。未打标记的情况必须抛出JMS *MessageFormatException*。如果数字的*valueOf() 方法不能接受*字符串的值作正确的表达式，那么*String*到 numeric 的转换必须抛出 j*ava.lang.NumberFormatException* .

	boolean	**byte**	**short**	**char**	**int**	**long**	**float**	**double**	**String**	**byte[]**
boolean	X								X	
byte		X	X		X	X			X	

short			X		X	X			X	
char				X					X	
int					X	X			X	
long						X			X	
float							X	X	X	
double								X	X	
String	X	X	X		X	X	X	X	X	
byte[]										X

. 试图读一个空值作为Java主类型必须被按照调用主类型相应的带有null值的 *valueOf(String)*转换方法来对待。因为char不支持String转换，所以试图读取一个null值作为一个char必须抛出NullPointerException异常。

通过字段名获得MapMessage的字段值，而这个字段值还没有被设置，这种情况按照字段已经存在，其值为null来对待。

如果*StreamMessage* 或*BytesMessage* 的读取方法抛出*MessageFormatException* 或 *NumberFormatException*, 当前读取指针的位置不能被增加。顺序读取必须能够从读取数据作为不同类型的异常处恢复。当作row类型写入的值可以由column类型读取。

第3章　JMS 通用设施

这一章主要描述由PTP和Pub/Sub共享的JMS设施。

3.1 Administered Objects 被管理的对象

JMS被管理的对象是包含ＪＭＳ配置信息的对象，这些对象由JMS管理者创建，并且最终由JMS客户端使用。它们使管理员在企业中实践ＪＭＳ应用。尽管管理对象接口没有显示说明依赖于JNDI, JMS建立了一种惯例，那就是JMS客户端通过使用JNDI在命名空间中查找管理对象。

管理员可以将被管理的对象放在任何一个命名空间中。JMS 没有定义命名策略。这种将 JMS 和管理分离的策略提供了几种好处：

•它为JMS客户端隐藏了JMS提供者规定的配置细节。

•它将JMS管理信息抽象为Java对象，这样可以很容易地通过一个通用的管理控制台来组织和管理这些对象。

•使用JNDI来提供通用的命名服务，这意味着，JMS提供者能够发布一个可到处运行的管理实现.

被管理对象不应持有任何远程的资源，它的查找不应使用除了JNDI本身之外的任何远程资源。客户端应将被管理对象看作为本地的Java对象。　查找这些被管理的对象不应有任何隐含的副作用或者使用数量惊人的本地资源。

JMS定义了两个被管理对象。*Destination*和*ConnectionFactory*.期望JMS提供者能够提供工具，使得管理员能够在JNDI名字空间中创建和配置被管理的对象。JMS提供者实现的被管理对象应当是*javax.naming.Referenceable* and *java.io.Serializable* ，这样它们可以被存储在所有的JNDI命名环境中。除此之外，建议这些实现遵循JavaBeansTM设计模式。

3.1.1 Destination 目的地

JMS没有定义一个标准的地址语法，尽管这个提议曾被考虑过。但是现存消息产品中的地址语义太宽广，无法用一个统一的语义来表示，因而,JMS定义了*Destination*对象，这个对象封装了提供者定义的地址。

因为*Destination*是一个被管理的对象，它可以包含一个提供者指定的配置信息以及它的地址。JMS也支持客户端使用提供者指定的地址名。*Destination*对象支持并发使用。

3.1.2 ConnectionFactory 连接工厂。

连接工厂封装了一系列由管理员定义的连接。客户端使用它创建与JMS提供者之间的连接，*ConnectionFactory*对象支持并发使用。

3.2 Connection 连接

JMS连接是一个连接到JMS提供者的活动连接，它通常会占用Java虚拟机之外的提供者资源。连接对象支持并发使用。连接服务于以下几种目的：

• 它封装了一个到ＪＭＳ提供者的开放连接。它通常表现为一个客户端和提供者守候服务之间的开放的TCP/IP套接。
•它的创建是在客户端认证发生地方。
•它指定了一个唯一的户端标识。
•它创建Session对象。
•它提供*ConnectionMetaData*。
•它支持可选的*ExceptionListener*。

因为认证和通信的建立是在*Connection*被创建时完成的，所以连接是一个相对重量级的JMS对象。大多数客户端将使用一个连接完成它们所有的消息通信。其他更高级的应用可以使用几个连接。JMS不存在由于架构方面的原因来使用多个连接（而不是由一个客户端扮演两个不同提供者之间的网关 ）但是，可能（应用）会因为操作方面的原因来使用多个连接。

3.2.1 Authentication 认证

当创建一个连接的时候，一个客户端可以指定它的身份，例如name/password。如果没有身份信息被设置，当前线程的身份则被使用。在这一点上，JDK没有定义线程缺省的身份。但是，可能将来会被定义。目前，正在运行的JMS客户端使用的用户身份将被使用。

3.2.2 Client Identifier 客户端标识

指定客户端标识的一个比较好的方法就是将其配置在一个客户端指定的ConnectionFactory中，并且将其透明地赋值给客户端所创建的连接。另一种方法就是客户端能够使用提供者指定的值来设置连接的客户端标识。显示设置客户端标识的设施不是用

来覆盖管理配置的机制。它用于不存在管理指定的标识时提供客户端标识。如果不存在客户端标识，试图通过设置来改变它会抛出*IllegalStateException*异常。

如果客户端显示地设置它，那么必须在创建连接后立即进行，并且要在任何连接行为被执行之前。如果不是在此时进行，那么设置客户端标识就是一个程序错误，并会抛出*IllegalStateException*.异常。客户端标识的目的是将连接及其对象与代表提供者客户端的状态相关联。通过定义，由客户端标识所表示的客户端的状态在同一时刻只能由同一个客户端所使用。JMS提供者必须阻止并发执行的客户端使用它。这种阻止的形式可以采用抛出JMSExceptions异常的方式，这会导致讨厌的客户端阻塞，或者一些其他的结果。JMS提供者必须保证这些试图“共享”一个单独客户端状态的企图不会导致消息被丢失或者重复处理。这个唯一的由JMS标识的客户端状态被用来支持持久化的定阅。

3.2.3 Connection Setup 连接的建立

JMS客户端通常创建一个*Connection*，一个或者多个*Sessions*以及大量的*MessageProducers* 和 *MessageConsumers*，那意味着没有消息被发送给它。

直到建立完成，连接才能离开被停止的模式。在连接的start()方法被调用时刻起，消息才能被到达连接的消费者。这个创建约定可将客户端的混乱减至最少，这些混乱会导致客户端尚且处于建立它自身的过程中，就会有异步消息传送。

一个连接可以被立即启动(start)，而建立(setup) 过程可以继续进行。客户端如果这样做就必须准备好在它们建立的过程中处理异步消息传送。

认识到这样一点很重要，就是在连接没有被启动(start)之前不会有消息被传送。JMS提供者必须保证这一点。

3.2.4 Pausing Delivery of Incoming Messages 停止传送即将到来的消息

通过使用它的stop()方法，连接可以临时停止即将到来的消息的传送，连接也能使用start()方法重新启动。当连接被停止了，给连接的所有*MessageConsumer*s的消息发送都会被禁止。同步接收块以及消息不会被传送给*MessageListener*s。

停止一个连接不会影响它发送消息的能力。停止一个已经被停止连接以及启动一个已启动的连接的操作都会被忽略。Stop方法的调用直到所有消息的传送被停止后才会返回。这意味着客户端可以信赖这样的一个事实：不会有任何消息监听器会被调用，并且，所有控制等待接受返回的线程不会返回任何消息，直到连接被重新启动。被停止的连接的接收计时器仍在继续向前，因此，当连接被停止时，消息接收可能会超时，并且返回一个null

消息。

如果消息监听器（MessageListeners）在连接停止时正在运行。Stop方法必须等待所有的消息监听器返回后它才能返回，这时，这些消息监听器（MessageListeners）都已经完成，它们必须完成可用连接的全部服务。

3.2.5 Closing a Connection 关闭连接

因为提供者通常占用由连接所代表的位于JVM之外的重要资源，客户端应当在这些资源不再需要时关闭它们。依赖于垃圾回收机制通常是不够及时的。

连接的关闭操作会结束该连接会话中消费者的所有未完成的（pending）的消息接受工作。消息接收操作可以返回一个消息或者一个null值，这取决于在关闭时是否有消息可以得到。注意，在这种情况下，如果消费者在处理最后的消息时试图使用现在已经关闭了的连接中的设施，那么可能会得到一个异常。当写一个消息的消费者时，开发者必须考虑“最后的消息”。它要承受这样的重复（循环），即：消息消费者不能够依靠一个返回的空值来表示这是“最后的消息”。

在连接关闭被调用的时候，如果有一个或者多个连接的会话的消息监听器正在处理一个消息，所有的连接设施以及会话必须保持让这些监听器得到，直到这些监听器将控制返回给JMS提供者。

当连接关闭被调用的时候，应当直到消息处理被停止才能返回。这意味着，所有的消息监听器可以被运行并且有返回，并且所有未决（未完成的）的消息接收工作已经被返回。

如果连接被关闭了，没有必要关闭它的组成对象。连接关闭足以通知ＪＭＳ提供者所有该连接的资源应被释放。

关闭连接必须回滚它的事务性会话中正在处理的事务。关闭时连接不要强行确认“客户端-确认”的会话。调用来自于已关闭连接的会话中的已接收消息的*acknowledge*方法必须抛出一个*IllegalStateException*异常。这些语义确保关闭连接不会导致队列中或者持久化订阅的消息丢失，以满足JMS客户端后续执行的处理是可靠的。

一旦连接被关闭，试图使用连接或者连接的会话，或者它们的消息消费者以及消息的生产者必须抛出*IllegalStateException*异常，（调用这些对象的close方法将被忽略）。除了已接收消息对象的*acknowledge*方法外，继续使用被创建的消息对象或者通过连接已接收的消息对象是合法的。

关闭一个已关闭的消息不必抛出异常。

3.2.6 Sessions 会话

连接是创建生产和消费消息的会话“工厂”。

＊术语“事务性会话”指的是这种情况，即，一个会话的提交和回滚方法被用来界定一个事务是否位于这个会话。在会话的工作由一个外部事务管理器来协调处理的时候。不需要调用会话的提交和回滚方法，并且，关闭会话工作的结果最终由事务管理器决定。

3.2.7 ConnectionMetaData

连接提供了一个*ConnectionMetaData*对象。这个对象提供了由提供者所支持的JMS的最新版本，以及提供者产品名称和版本。它也提供了一个由连接所支持的JMS定义的属性名称列表。

3.2.8 ExceptionListener 异常监听器

如果JMS提供者发现了连接的问题，如果有异常监听器被注册，它将通知给连接的*ExceptionListener*，。要获取异常监听器，JMS提供者则调用连接的 *getExceptionListerer()*方法。这个方法返回连接的*ExceptionListener*。 如果没有异常监听器被注册，将返回null值。因而连接能通过调用监听器的*onException()*方法使用监听器，将描述问题的*JMSException*异常传递给监听器。

这个机制使得客户端可以异步地查看问题。一些连接只消费消息，因此他们没有途径知道它们的连接已经有失败发生。

连接顺序执行它们的*ExceptionListener*。

JMS提供者应当试图在通知给客户端之前试图解决连接中发生的问题。

发送给*ExceptionListener*的异常应是那些没有地方报告的异常。如果由JMS调用导致的异常，那么按照定义，这个异常不必发送给*ExceptionListener*。（换句话说，*ExceptionListener*不是为了监视所有由连接抛出的异常）

3.3 Session 会话

JMS 会话是一个用来生产和消费消息的单线程的上下文。尽管它在Java虚拟机之外分配了提供者的资源，它被认为是一个轻量级的ＪＭＳ对象。

会话服务于以下几个目的：

- 它是*MessageProducer*s和*MessageConsumer*s 的工厂。

- 它是*TemporaryTopics*和*TemporaryQueues*的工厂。
- 它为那些需要动态操作提供者指定目的名称的客户端创建*Queue* 或*Topic*对象。
- 它提供了提供者优化的消息工厂。
- 它支持独立序列的事务，将扩越会话的生产者和消费者的工作组成一个原子单元。
- 它为它所消费的消息以及产生的消息定义了序列顺序。
- 它持有它所消费的消息，直到这些消息已经被确认。
- 它是*QueueBrowsers*的工厂。

3.3.1 Closing a Session 关闭会话

因为提供者可能会为每个Session分配JVM以外的资源，客户端应当在不需要的时候关闭他们。依靠垃圾回收机制证明并不及时可靠。这对于由会话所创建的*MessageProducer*s和 *MessageConsumer*s 也是一样的。

Session关闭结束了这个会话中的所有正在进行的消息处理工作。它必须处理消费者未决接收的停止或者正在运行的消息监听器的停止。如同关闭连接里面描述的一样。

***这里没有限制能够使用一个会话对象的线程数量，也没有限制线程能够创建会话的数量。限制就是会话的资源不能够被多个线程并发使用。它要求用户来保证这种并发限制被满足。最简单的方法就是使用一个线程。在异步传送的情况下，使用一个线程用来在停止模式下建立会话，然后启动异步传送。在更复杂的情况下，用户必须提供显式的同步。**

当会话的close方法被调用后，它不会立即返回，直到会话正在进行的消息处理被依次停止后才能返回。这意味着没有消息监听器在运行，并且如果没有未决的接收，它会返回一个null或者一个消息。

当会话被关闭的时候，没有必要关闭它所包含的消息生产者和消息消费者。会话关闭足以通知JMS提供者去释放所有与会话相关的资源。

关闭一个事物性的会话，必须回滚正在处理的事务。关闭一个“客户端确认”类型的会话不会强制产生确认。

一旦会话被关闭，试图使用它的消息消费者和产生着必须抛出*IllegalStateException*异常。（调用这些对象的close方法必须被忽略）。继续使用被创建的消息或者通过Session接收的消息是合法的，除了接收对象的*acknowledge*方法以外。

关闭一个已经关闭的会话不必抛出异常。

3.3.2 MessageProducer 和 MessageConsumer 的创建

. 会话能创建和服务于多个*MessageProducer*s 和*MessageConsumer*s.尽管一个会话可以创建多个生产者和消费者，他们被限制为串行使用。实际上，只有单个本地控制线程能够使用它们。这在以后会详细讨论。

3.3.3 Creating Temporary Destinations 创建临时目的地

尽管会话被用于创建临时目的地。这只是一种常规约定。它们（临时目的地）的范围实际上是整个连接。它们的生命周期存在于整个连接，并且连接的任何会话都被允许去为这些目的地创建*MessageConsumer*。

临时目的地（*TemporaryQueue或者TemporaryTopic* 对象）是唯一的系统产生的连接中的目的地。只有临时目的地自己的连接被允许创建*MessageConsumer*s。临时目的地典型用法是作为服务请求的*JMSReplyTo目的地*。

每个*TemporaryQueue* 或 *TemporaryTopic*目的地都是唯一的。它不能被拷贝。 因为临时目的地可以分配ＪＶＭ以外的资源，所以应在不用的时候删除这些临时目的地。当连接关闭时，它们会自动被删除。

3.3.4 Creating Destination Objects 创建目的地对象。

大多数客户端将会通过在JNDI中查找的方式来使用*Destination*s这种JMS被管理对象，这是最具可移植性方法。一些特定的客户端需要通过使用提供者指定目的地名称来动态生成目的地对象，Sesssion提供了一个JMS“提供者指定的”方法来完成这种工作。

3.3.5 Optimized Message Implementations 优化消息的实现。

会话提供的消息创建方法使用了“提供者优化”的实现，这使得提供者能够使用最小的开销来处理消息。

会话一定能够发出所有JMS消息而不考虑它们是如何实现的。

3.3.6 Conventions for Using a Session 使用 Session 的常规

会话被设计为只能被一个线程在同一个时刻顺序使用。只有一个例外就是当会话或者它的连接被顺序关闭时。见4.3.5“关闭连接”和4.4.1“关闭会话”获取详细信息。

一个典型的用法是对同步的*MessageConsumer*使用线程阻塞，直到消息达到。线程可以使用一个或者多个会话的*MessageProducer*s。

如果已经有一个客户线程控制等待接收消息，这时客户端在同一个会话中使用另一个控制线程去试图同步接收消息会产生错误。

另一个典型的用法是使用一个线程通过创建它的生产者和一个或者多个异步消费者的方式来建立一个会话。在这种情况下，消息生产者以专有(排他) 方式使用消费者的消息监听器。因为会话顺序地执行消费者的*MessageListener*s，他们能够安全地共享会话中的资源。

当连接中的会话被建立，如果连接停留在被停止的模式下，在客户端充分准备处理消息之前，不必处理消息的接收。这是最好的策略，因为，它减少了建立会话和消息处理之间发生不可预期的冲突的可能性。**当连接正在接收消息时创建和建立会话是可能的，在这种情况下，要非常谨慎，保证会中的*MessageProducer*s、*MessageConsumer*s和*MessageListener*s按照正确的顺序创建。**例如：一个错误的顺序可以导致*MessageListener*去使用一个还没有被创建的*MessageProducer*；或者由于*MessageListener*s注册的顺序导致的消息到达顺序错误。如果客户端需要使一个线程创建消息而其他线程消费这些消息，那么客户端应当让创建线程使用单独的会话。

一旦连接被启动， 它所有带有已注册的消息监听器的会话被用于这些会话发送消息的控制线程。如果客户端在另一个控制线程中使用这样的会话则会产生错误。只有一种情况例外，那就是使用会话或者连接的close方法不会出错。

这种会话的“单线程控制”限制的**一个推论就是：一个带有消息监听器的会话不能被用于同步接收消息**。这样的会话要么专用于向消息监听器发送消息的控制线程，要么专用于客户端代码初始化的控制线程。如果试图在一个会话中完成两个工作，就会出现错误。**另一个推论就是：在建立带有一个或者多个消息监听器会话时，连接必须处于“停止”模式。**原因就是当连接正在发送消息时，一旦会话的第一个消息监听器被注册，会话则会被向其传送消息的线程所控制，这导致客户端控制线程不能进行进一步的会话配置工作。

对于大多数客户端而言，分离会话的工作是很自然的事情。这种模式允许客户端启动并能根据并发的需要逐步增加消息处理的复杂度。

3.3.7 Transactions 事务

会话可以根据需要被指定为“事务性”的。每个事务性的会话支持单独序列事务。每个事务将一系列产生的消息和一些列消费的消息打包成一个原子性的工作单元。实际上，事务组织将一个会话中的输入消息流和输出消息流组织成为序列的原子单元。当时事务提交时，输入的原子单元被确认并且与之相关的输出原子单元被发出。当事务回滚时，它产

生的消息被销毁，并且它消费的消息被自动恢复。查看更多关于消息恢复的内容请见4.4.11 “Message Acknowledgment” 。

可用会话的*commit()* 或者 *rollback()*方法来结束事务，会话当前事务的完成会导致下一个事务自动开始。其结果就是：事务性的会话总是有一个当前事务，工作是在这个事务中完成。

JTS或者一些其他事务监控设施可以将会话的事务同其他资源的事务相组合（数据库或者其他ＪＭＳ事务等等）。因为Java分布式事务是有ＪＴＡ　事务API所控制，在这种（分布事务）上下文中使用会话的*commit*或者*rollback*方法会抛出JMS *TransactionInProgressException*异常。

3.3.8 Distributed Transactions 分布事务

JMS部要求提供者支持分布式事务，但是它也要求一旦提供者支持分布事务，那么应当支持JTA *XAResource* API

JMS 提供者也可以是一个分布式事务监控器，如果这样，它应当通过JTA API提供事务控制。

尽管JMS客户端可能直接处理分布式事务，我们仍然建议JMS客户端避免自己处理分布式事务。JMS客户端使用后面“JMS Application ServerFacilities” 中描述的基于XA接口进行开发可能在不同JMS实现中难以移植，因此这些接口是可选的选项。在JMS中支持JTA是那些要把JMS集成到他们自己应用服务器中厂商的目标。

3.3.9 Multiple Sessions 多会话

客户端可以创建多个会话，每个会话都是相互独立的消息生产者和消费者。

对于发布/订阅模型来说，如果两个会话，每个都有一个*TopicSubscriber*订阅相同的*Topic*，每个订阅者都会给一个消息。向一个订阅者传送消息不会阻塞其他后面的订阅者。

对于点对点(PTP)模式，JMS没有指出相同*Queue*的并发*QueueReceivers*的规范，但是JMS不禁止提供者支持并发。因此，消息是否传送给多个*QueueReceivers*取决于JMS提供者的实现，这种依赖会导致应用的不可移植。

3.3.10 Message Order 消息顺序

JMS 客户端需要了解它们什么时候可以依赖消息的顺序，而什么时候不能。

3.3.10.1Order of Message Receipt 消息接收顺序

由会话消费的消息定义了一个序列顺序。这个顺序非常重要，因为它定义了消息确认的结果，详细信息见3.4.11“消息确认”。会话中所有消费者的消息都被插入到一个会话输入消息流中。JMS定义：由一个会话向一个目的地发出的消息必须按照发出的顺序接收。这样“部分地”定义了会话输入消息流中的顺序约束。

JMS没有定义跨越多个目的地的消息接收顺序，或者跨越目的地的消息来自多个会话的接收顺序。会话在接收消息流的顺序方面是“时间依赖”。它不在应用的控制之下。

3.3.10.2 Order of Message Sends 消息发出顺序

尽管客户端松散地将它们在一个会话中产生的消息看作发出消息流的形式。流中的全部顺序并不重要。对接收客户端看到的顺序只是消息向特定目的地发出的顺序。几个事情能够影响这个顺序。

- 高优先级的消息排在低优先级的消息之前。
- 如果JMS提供者失败，则客户端不接受NON_PERSISTENT消息。
- 如果PERSISTENT和NON_PERSISTENT被发往一个目的地，那么只保证在传递模式中的顺序。也就是说，一个迟后的PERSISTENT可以在NON_PERSISTENT消息之前到达。但是，他不会在较早的同一优先级的NON_PERSISTENT消息之前到达。
- 客户端可以使用事务性的会话去将发送的消息放在一个原子单元中（JMS事务的生产者组件）。到相同目的地的事务消息顺序很重要。跨越目的地消息的发送顺序不重要。

3.3.11 Message Acknowledgment 消息确认

如果会话是事务性的，消息确认在*commit*时自动处理，并且恢复在*rollback*时自动处理。如果会话不是事务性的，则有三种确认选项，并且恢复消息需要手工处理：

• DUPS_OK_ACKNOWLEDGE – 这个选项指令会话延迟发送消息的确认，如果JMS失败，则可能导致消息重复发送，因此只能在消息的消费者容忍重复消息时使用。它的好处在于通过最小化会话所作的工作来减少了会话的负荷。

• AUTO_ACKNOWLEDGE -带有这个选项，当会话成功从receive调用返回或者它所调用的*MessageListener*处理消息成功地返回时，会话自动确认客户端对消息的接收，

• CLIENT_ACKNOWLEDGE -带有这个选项，客户端通过调用消息的*acknowledge*方法来对消息进行确认。这个确认已消费消息的操作会自动地确认被传送给会话的所有消息。

当CLIENT_ACKNOWLEDGE模式被使用，在需要处理消息的时候，客户端会阻塞大

量的未确认的消息。JMS提供者应提供管理者一种途径去限制客户端溢出。这样客户端不会资源耗尽且在一些使用的资源被临时阻塞时确保失败。

会话的*recover*方法被用来停止会话并且带着会话第一未被确认的消息来重新启动会话。实际上，会话发送消息的序列在它最后确认消息的一刻被重置。由于消息过期以及更高优先级消息的到达，此时被发出的消息同那些原始传递的消息不同。

3.3.12 Duplicate Delivery of Messages 重复的消息传送

JMS 提供者不必传送已确认消息的另一个拷贝。

当客户端使用AUTO_ACKNOWLEDGE模式时，它不是直接控制消息的确认，因为这些客户端不知道特定的消息是否已经被确认了，他们必须准备重新传送最后消费的消息。这些是因为客户端完成它的工作恰好先于防止消息确认发生失败之前导致的。只有会话最后被消费的消息容易不明确。在这样的情况下，一个被重新发送的消息其*JMSRedelivered*消息头字段将被设置。

3.3.13 Duplicate Production of Messages 重复的消息生产

JMS提供者不必生产重复消息，这意味着客户端生产消息可以依赖它的JMS提供者，以保证消息的消费者只接受一次消息。客户端的错误能不会导致提供者重复消息。

如果在客户端提交Session工作和提交方法返回之间发生错误，客户端不能确定事务是否被提交还是被回滚。同样的模棱两可在非事务性的PERSISTENT消息发出和返出方法返回之间也同样存在。

由ＪＭＳ应用决定处理这些模棱两可。有些情况下，这会导致客户端生产功能性重复的消息。由于会话恢复导致的重新发送的消息不被认为是重复的消息。

3.3.14 Serial Execution of Client Code 顺序执行客户端代码

尽管Java语言提供内建的多线程支持，编写多线程程序仍然比单线程程序困难。

因为这个原因，JMS不会导致并发执行客户端的代码，除非客户端明确地要求这样做。这样做的一个办法就是定义一个会话来序列化所有异步消息的发送。

为了异步接受消息，客户端注册一个对象，这个对象实现JMS *MessageListener*和*MessageConsumer*接口。实际上，会话使用单独线程运行所有它的*MessageListeners.*当线程忙于执行一个监听器，所有其他被异步发送给会话的消息必须等待。

3.3.15 Concurrent Message Delivery 并发消息传送

需要并发发送消息的客户端可以使用多个会话。实际上，是每个会话的监听器线程并发运行。当一个会话上的监听器在执行的时候，另一个会话上的监听器也可以执行。

注意，JMS自身不提供并发处理一个主题的消息的设施。客户端可以使用单个消费者并且实现所需并发处理的多线程逻辑。但是，这样做不可能可靠。因为JMS没有处理并发事务这种需求的事务措施。

3.4 MessageConsumer 消息消费者

客户端使用*MessageConsumer*接受来自于目的地的消息，*MessageConsumer*通过向*Session*的*createConsumer*方法传递*Queue*或*Topic*来创建。

消费者可以被带有消息选择器的方式来创建。这使得客户端可以限制传送给消费者的消息必须同选择器相匹配。客户端既可以同步获取消费者的消息，也可以使提供者在消息到达时异步传送消息。

3.4.1 Synchronous Delivery 同步传送

客户端可以使用*MessageConsumer*的*receive*方法请求下一个来自于*MessageConsumer*的消息。Receive有几种变化允许客户端poll或者wait下一个消息。

3.4.2 Asynchronous Delivery 异步传送

客户端可以注册一个用*MessageConsumer*来实现JMS *MessageListener接口的*对象.当消息达到了消费者时，提供者通过调用监听器的*onMessage*方法来传送它们。可能监听器会抛出*RuntimeException*异常，但是这主要考虑到的事客户端程序错误。良好的监听器应当捕捉这些异常并且尝试将这些消息转向发给一些应用指定的某些形式的“不可处理消息”的目的地。

监听器抛出*RuntimeException*的结果取决于会话的确认模式：

• AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE – 消息将被立即重发。在放弃之前的重发的次数取决于提供商。在这种情况下，*JMSRedelivered* 消息头字段将设置在被重发的消息中。

• CLIENT_ACKNOWLEDGE –监听器的下一个消息将被传送。如果客户端希望是前面未确认的消息重新发送，它必须手工恢复会话。

• Transacted Session – 监听器的下一个消息被发送，客户端可以提交或者回滚会话。（换句话说，*RuntimeException*不会导致会话的自动回滚）JMS提供者应当将消息监听器抛出异常的客户端标记为“可能的障碍”。

3.5 MessageProducer 消息生产者

客户端使用*MessageProducer*向目的地发送消息。用*Queue*或者*Topic*对象作为参数来调用session对象的*createProducer*方法来创建*MessageProducer*。

客户端也可以选择创建没有目的地的生产者。这种情况下，目的地对象必须传给每个发送操作。这种方式的一个典型用法就是生产者被用来发送回复请求时，使用请求的*JMSReplyTo目的地*。

客户端可指定由生产者发出的消息的缺省的传送模式、优先级、存活周期。每次客户端创建*MessageProducer*，它就定义了新的消息系列，这些消息与以前发送的消息没有顺序关系。

3.6 Message Delivery Mode 消息传送模式

JMS支持两种消息传送模式：

- NON_PERSISTENT 模式是开销最低的传输模式，因为它不需要消息被记录到稳定的存储中。JMS提供者失败会导致NON_PERSISTENT 被丢失。
- PERSISTENT 模式指令JMS提供者特别关注“不能由于JMS 提供者的失败而导致的消息的丢失”。JMS提供者最多只能将NON_PERSISTENT消息发送一次。这意味着，它可能会丢失消息，但决不会发送两次。JMS提供者必须保证PERSISTENT发送一次且只有一次。这意味着，ＪＭＳ提供者的失败不会导致消息丢失，而且它不会被发送两次。

PERSISTENT (一次且只有一次) 和NON_PERSISTENT (最多一次)模式是JMS客户端在ＪＭＳ死机时可以丢失消息还是确保JMS失败时花费额外的努力将消息从失败中拯救出来的传输技术的选择。这种选择通常是性能/可靠性之间的权衡。当客户端选择NON_PERSISTENT传输模式，则表明性能要求高于可靠性。当用户选择PERSISTENT模式时，则表明要求相反的权衡。

使用*PERSISTENT*消息不能保证所有的消息都会发送给每个符合条件的客户端。看3.10“可靠性”来讨论这个话题。

3.7 Message Time-To-Live 消息存活时间

客户端可以以毫秒为单位指定它发出的每个消息的存活时间。这个值指定了消息的过期时间，用发送时的GMT时间加上存活时间就是过期时间。（对于事务性的消息发送，发送时间是指客户端发送消息的时间，不是事务提交的时间）。

JMS提供者最好让过期时间精确，但是，JMS没有定义精确的程度。简单忽略存活时间是不可接受的。

3.8 Exceptions 异常

JMSException 是所有JMS异常的基类。

3.9 Reliability 可靠性

大多数客户端应使用生产者来生产PERSISTENT消息。这保证消息从队列或者持久化的订阅传送一次且只有一次的。在有些情况下，应用需要最多传送一次的消息，这就需要发布.NON_PERSISTENT消息。这些消息通常开销很低，但是，它们可能在JMS提供者失败时丢失。 PERSISTENT和NON_PERSISTENT消息都能发给相同的目的地。

通常，消费者在确认消息前完整地处理每个消息，在机器故障时可保证JMS不会丢弃部分处理。例如，消费者通过事务性的或者CLIENT_ACKNOWLEDGE会话来完成这个工作。由于系统失败而导致重发的未确认消息必须带有JMS提供者设置的*JMSRedelivered*头字段。

如果NON_PERSISTENT消息被发送给一个持久化的订阅或者队列，如果持久化的订阅变得不活动（也就是说如果没有当前的订阅者）或者如果JMS提供者关闭重起，则不能保证发送确实成功。

我们期望重要的消息在事务中被以PERSISTENT模式生产，并且将被事务中的nontemporaryde(非临时) 队列或者持久化订阅消费。

当这么做的时候，应用在“消息被正确生产、可靠传输、正确消费”方面有最高级别的保证。非事务性的生产和消费也能达到相同级别的保证。但是，这需要仔细编程。

JMS提供者会用资源限制来制约可被高容量目的地或者“非响应”客户端持有的消息数量。如果因为资源限制导致消息被砍掉，这通常是一个需要关注的管理问题。JMS正常工作需要客户端能够响应并且要保证充足的资源来提供服务。

本规范中描述的“有且只有一次”的传输模式有一个重要的警告，那就是它不适用于由于过期或者因其他管理方面的销毁准则而销毁的消息。它也不适用于因为资源限制导致的丢失。为JMS应用配制充足的资源和处理能力是管理员的一项工作，管理员必须了解JMS提供者的可靠性特征。

NON_PERSISTENT消息，非持久化的订阅，以及临时性的目的地被定义为不可靠。JMS提供者关闭或者失败将导致NON_PERSISTENT的消息的丢失，也会导致临时目的和非持久化订阅所持有的消息丢失。应用的结束很可能导致由非持久化的订阅或者临时目的地所持有的消息丢失。

第4章 JMS 点对点传输模式

4.1 Overview 概述

点对点系统是用队列来处理消息，它之所以是“点对点”是因为客户端向特定的队列发送消息。一些PTP系统由于提供了能够自动分发消息的系统客户端从而混淆了PTP和Pub/Sub之间的区别。

对于客户端来说，所有的消息被发送给一个单独的队列是很常见的。如同任何普通邮箱一样，队列能够包含消息的混合。并且像实际的邮箱一样，创建和维护每个队列是有相当的开销。多数队列由管理创建并且被当作客户端的静态资源。

JMS点对点模式定义客户端如何同“队列”工作：如何发现队列，如何向队列发送消息。如何从队列获取消息。本章描述点对点模式的语义。支持点对点模式的JMS提供者必须交付这里所描述的语义。

无论JMS客户端程序使用PTP特定领域接口还是在第 3 章“JMS通用设置”中描述通用接口，客户端程序必须保证相同的行为。

表 4 -1 显示了专用于PTP领域的接口以及JMS 通用接口。最好使用JMS通用接口创建应用，因为他们与领域无关。

Table 4 -1 PTP Domain Interfaces and JMS Common Interfaces

PTP Domain Interfaces	**JMS Common Interfaces**
QueueConnectionFactory	ConnectionFactory
QueueConnection	Connection
Queue	Destination
QueueSession	Session
QueueSender	MessageProducer
QueueReceiver	MessageConsumer

4.2 Queue Management（队列管理）

大多数客户端使用静态定义的队列，，JMS没有定义创建、管理、删除长期存活的队列（它确实提供了这样一种*TemporaryQueue*s队列的机制）的措施，因为这没有任何问题。

4.3 Queue（队列）

*Queue*对象封装了提供者定义的队列名称，它是客户端向JMS方法说明队列标识的一种方法。JMS没有定义消息被队列持有的时间长短以及资源溢出后如何处理。

4.4 TemporaryQueue

TemporaryQueue 是为*Connection*或者*QueueConnection*持久化而创建的唯一的*Queue*对象，它由“系统定义”的且只能被创建它的*Connection*或者*QueueConnection*所消费。

4.5 QueueConnectionFactory

客户端用*QueueConnectionFactory*创建到JMS PTP提供者之间的*QueueConnections*连接。

4.6QueueConnection

A *QueueConnection* 是到JMS PTP提供之间的活动连接，客户端使用*QueueConnection* 来创建一个或者多个*QueueSessions*，以生产和消费消息。

4.7 QueueSession

QueueSession 提供创建*QueueReceivers*, *QueueSenders*,*QueueBrowsers*,以及*TemporaryQueues*的方法。

当*QueueSession*结束时，如果存在接收到而未被确认的消息，这些消息必须被保持，并且在消费者下次访问队列时重新发送。

4.8 QueueReceiver

客户端使用*QueueReceiver*来接收已经发送给队列的消息。尽管一个队列可能有两个带有*QueueReceiver*的会话，但是ＪＭＳ没有定义消息如何在*QueueReceiver*之间分发。

如果指定了*QueueReceiver*消息选择器，没有被选择的消息仍然留在队列中。通过定义，消息选择器可以让*QueueReceiver*忽略一些消息。这意味着当被忽略的消息最后被读取时，整体的阅读顺序没有遵照消息生产者局部定义的顺序。只有当*QueueReceivers*不带有消息

选择器，它才能按照消息生产者定义的顺序阅读消息。详细信息见3.5, “MessageConsumer.” ，如果*MessageConsumer* 正从*Queue*中消费消息，它的行为必须按照4.8, “QueueReceiver.”描述进行。客户端使用*MessageProducer* 或者 *QueueSender*向*Queue*发送消息。详细信息参见3.6, “MessageProducer.”

4.9 QueueBrowser

客户端使用*QueueBrowser*去查看队列中的消息而不会移走消息。可从*Session* 或者*QueueSession*来创建*QueueBrowser*。

浏览方法返回*java.util.Enumeration*对象，这个对象被用来去扫描队列的消息。可以枚举整个队列的内容，也可以只包含同选择器相匹配的消息。

当进行扫描时，消息可以到达或过期。JMS不要求枚举的内容是静态的队列内容快照。队列内容变化是否在扫描时可见，取决于JMS提供者。

4.10 QueueRequestor

JMS 提供了一个*QueueRequestor*帮助器类用来简化产生服务请求。*QueueRequestor*的构造器要求一个*QueueSession*和一个目的地队列作为参数，它创建一个*TemporaryQueue*用来响应和提供请求方法发送请求消息并且等待它的回复。这一个基本的请求／回复抽象，可以满足大多数用途，JMS提供者和客户端可以创建更复杂的用途。

4.11 Reliability 可靠性

队列通常由管理员创建，并且长时间存在。它通常会持有发送给他的消息，而无论消费消息的客户端是否活动。因此，客户端不必采用特别的预防措施去保证消息不会丢失。

第5章 JMS 发布／订阅（Publish/Subscribe）模式

5.1 Overview 概述

JMS Pub/Sub模式定义JMS客户端如何发布消息，并且从一个众所周知的基于内容结构的节点来定义消息。JMS将这些节点称之为主题(topics)。

在本节中，术语发布（*publish*）和定阅（*subscrib*）用于代替前面提到的更为通用的术语生产（*produce*）和消费（*produce*）。

主题（topic）可以被认为是一个小的消息调度器,这个调度器用来收集和发布发送给它的消息。依靠主题作为中介，消息发布同订阅者之间保持独立。反之亦然。

当代表它们的Java对象存在时，发布者和订阅者是活动。JMS也支持可选的“持久化能力”的订阅者以“记住”它们活动时的存在。

本章描述的是发布／订阅模型的语义。支持发布／订阅模式的JMS提供者必须交付这里描述的语义。

无论JMS客户端程序使用发布／订阅领域特定接口还是使用第 3 章“JMS Common Facilities”描述的通用接口，客户端程序都必须保证相同的行为。

表 5-1 显示了专用于pub/sub领域的接口以及JMS 通用接口。最好使用JMS通用接口创建应用，因为他们与领域无关。

表 5-1

Pub/Sub Domain interfaces	**JMS Common interfaces**
TopicConnectionFactory	ConnectionFactory
TopicConnection	Connection
Topic	Destination
TopicSession	Session
TopicPublisher	MessageProducer
TopicSubscriber	MessageConsumer

5.2 Pub/Sub Latency 延迟

因为在所有的发布/定阅 系统中都存在延迟，订阅者什么时候看到消息取决于JMS提

供者发送消息给已在订阅者的速度，以及提供者在传输中保持消息的时间长短。

例如，一些来自远程的发布者的消息可能被错过，因为这个消息向系统范围内给新建订阅者传播会花费一定时间。当一个新的订阅者被创建时，它可以接收早期发送的消息，因为提供者会使这些消息可以被得到。

JMS does not define the exact semantics that apply during the interval when a pub/sub provider is adjusting to a new client. JMS semantics only apply once the provider has reached a ‘steady state’ with respect to a new client.

5.3 Durable Subscription 持久化的订阅

非持久化订阅持续到它们订阅对象的生命周期。这意味着，客户端只能在订阅者活动时看到相关主题发布的消息。如果订阅者不活动，它会错过相关主题的消息。

如果花费较大的开销，订阅者可以被定义为durable（持久化的）。持久化的订阅者注册一个带有JMS保持的唯一标识的持久化订阅（subscription）。带有相同标识的后续订阅者会再续前一个订阅者的订阅状态。如果持久化订阅没有活动的订阅者，JMS会保持订阅消息，直到消息被订阅接收或者过期。

所有的ＪＭＳ提供者必须能够运行JMS应用来动态创建和删除持久化的订阅。除此之外，一些JMS提供者可以提供管理这些持久化订阅的设施。如果持久化订阅被管理性配置，**I t is valid for it to silently override the subscription specified by the client.**

不活动的持久化订阅是存在的，但是当前它没有消息消费者。

5.4 Topic Management 主题管理

一些产品需要主题被与相关的授权控制列表等一起静态定义，其他产品甚至没有主题管理的概念。

JMS没定义主题创建、管理和删除的设施。要提供一个特定类型的主题 *TemporaryTopic* ，它用来创建对应*TopicConnection*唯一的主题。

5.5 Topic 主题

主题对象封装了提供者指定的主题名称。它是客户端将主题标识指定给JMS方法的一种途径。很多发布／订阅提供者将主题分成层次，并且提供不同订阅选项来订阅层次中不

同的部分。JMS对于主题对象表达方式没有限制。它可以是主题层次中的叶子，也可以是层次中更大的部分（用于订阅通用类的信息）。

主题的组织和订阅主题的粒度是发布 / 订阅　应用结构的重要部分。JMS没有制定关于如何处理这个方面的任何策略。如果应用利用提供者指定的主题分组机制，它应当有此方面的文档。如果应用被安装使用另外的提供者，那就应该由管理员去构建等价的主题结构和创建等价的主题对象。

5.6 TemporaryTopic

*TemporaryTopic*是唯一一个用于*Connection* 或 *TopicConnection*持久的主题对象。它是系统定义的主题，只能被创建它的*Connection* 或 *TopicConnection*消费。通过定义，我们发现为临时主题创建持久化订阅没有意义。如果这样做，则JMS提供者可以检测这个错误也可以不检测这个错误。

5.7 TopicConnectionFactory

客户端使用 *TopicConnectionFactory* 去创建到JMS 发布/订阅提供者的*TopicConnection*s

5.8 TopicConnection

TopicConnection 是用来连接到JMS 发布/订阅提供者的一个活动连接。客户端使用*TopicConnection*创建一个或者多个*TopicSession*s来生产和消费消息。

5.9 TopicSession

*TopicSession*提供创建*TopicPublisher*s、*TopicSubscriber*s以及*TemporaryTopic*s的方法。它也提供了*unsubscribe*方法用来删除它的客户端的持久化订阅。

当*TopicSession*结束时，如果消息被接收但是没有被确认，那么持久化的*TopicSubscriber*必须保持这些消息并且重新发送它们。

5.10 TopicPublisher

客户端使用来*TopicPublisher*将消息发布到主题。*TopicPublisher*是JMS *MessageProducer*在发布/订阅模式下的变量。消息也可通过*MessageProducer*发布到主题上。

5.11TopicSubscriber

客户端使用*TopicSubscriber*来接收已经发布到主题的消息。

普通的*TopicSubscribers*不是持久化的。他们只能在活动时接收已发布的消息。

被订阅者消息过滤器过滤掉的消息将不会被发送到订阅者。

被订阅者消息选择器过滤掉的消息不会再次被发送给订阅者。从订阅者的视角来看，它们仿佛不存在。

在某些情况下，一个连接可以同时发布和订阅一个主题。订阅者*NoLocal*属性允许订阅者禁止发送它自己连接所发布的消息。

*TopicSession*允许每个目的地创建多个*TopicSubscribers*,它将目的地的消息发送给每个符合条件的接收订阅者。每个消息的拷贝都被看作完整的独立的消息，对一个拷贝的处理不会影响其他的拷贝。确认一个拷贝不会确认其他的，一个消息可以被立即发送，而另外一个消息可能在前面等待消息者处理。

5.11.1Durable TopicSubscriber 持久化的主题订阅

如果客户端需要接收主题上所有发布的消息，包括那些订阅者不活动期间发布的消息，那么就要使用持久化的*TopicSubscriber*.持久化的*TopicSubscriber*可以被*Session*或者被*TopicSession*创建。JMS保持这个持久化订阅的记录并且确保所有来自于主题发布者的消息都会被保持到被持久化订阅者确认，或者过期。

带有持久化订阅的会画必须提供相同的客户端标识。另外，每个客户端必须指定一个名字来唯一标识（在客户端标识范围内）它所创建的每个持久化订阅。对于特定的持久化订阅，在同一时刻只有一个会话在可以有一个*TopicSubscriber*。*参见*Section 3.3.2, “Client Identifier,”获取更多的信息。

客户端通过创建相同名字以及新的主题和（或）消息选择器以及NoLocal 属性的*TopicSubscriber*，可以改变已经存在持久化订阅。更改持久化订阅相当于删除后重建。

Sessions 和TopicSessions提供unsubscribe方法来删除由它们的客户端创建持久化订阅。这样做会删除提供者为持久化订阅而保留的状态信息。客户端删除持久化订阅而这个持久化订阅有一个活动的TopicSubscriber，或者已接收到的消息是当前事务的一部分或者还没有在会话中被确认时，就会出现错误。

5.12 Recovery and Redelivery 恢复和重发

非持久化订阅者的未确认消息应当可以在非持久化订阅者生命周期中被恢复。当非持久化的订阅者终结时，等待它的消息可能被取消而无论消息是否已经被确认。

只有持久化订阅能够可靠地恢复未确认的消息。

发送带有PERSISTENT发送模式的消息到主题不会改变这种恢复和重新发送的模式。为了保证发送，*TopicSubscriber*应当建立持久化的订阅。

5.13 Administering Subscriptions 管理定阅

理想情况下，当发布者和订阅者被创建时，它们向提供者的注册是动态的。从客户端的视角来看，这是通常情况。从管理者的视角来看，其他任务需要支持发布者和订阅者的创建。

JMS没有定义消息存储分配的资源数量和资源溢出时的后续处理。

5.14TopicRequestor

JMS提供了一个*TopicRequestor*帮助者类来简化服务请求的开发。*TopicRequestor*构造器需要一个*TopicSession*和destination对象。它为响应创建了一个TemporaryTopic，并提供request()方法来发送请求消息并等待消息的回复。

这是个基本的“请求/回复”模型，可以满足大多数情况。JMS提供者和客户端可以自由发挥以创建更多复杂应用。

5.15 Reliability 可靠性

当主题所有的消息都要被获取，应当使用持久化订阅（**Durable Subscriber**）。JMS保证在持久化的订阅者不活动时发布的消息可以被JMS保持，并且当订阅者后来变为活动时发送给它。

非持久化的订阅者应当只在可以容忍消息丢失的情况下使用。

Table 5-2 Pub/Sub可靠性

How Published	**Nondurable Subscriber**	**Durable Subscriber**
NON_PERSISTENT	at-most-once (missed if inactive)	at-most-once

PERSISTENT	once-and-only-once (missed if inactive)	once-and-only-once

第6章 SUN MQ 安装及配置

6.1 安装注意事项

- ✓ 安装 Sun Message Queue 2005 sp1(3.6)时，要保证 JRE 运行环境为 JRE1.5,如果为 JRE1.6 则安装后，运行 MQ 服务或者管理工具会出现错误，导致安装的服务和管理工具无法运行。
- ✓ 在 Windows2003 或者 Windows XP 下安装 Sun Message Queue 2005 sp1(3.6)时，确保安装前注册表中没有以下条目，否则安装程序会挂起：
  - ➢ KEY_LOCAL_MACHINE\SOFTWARE\ Sun Microsystems\ EntSys
  - ➢ KEY_LOCAL_MACHINE\SOFTWARE\ Sun Microsystems\ MessageQueue

6.2 JMS 服务管理代理并创建各种目的地对象

6.2.1 创建 JMS 服务管理代理

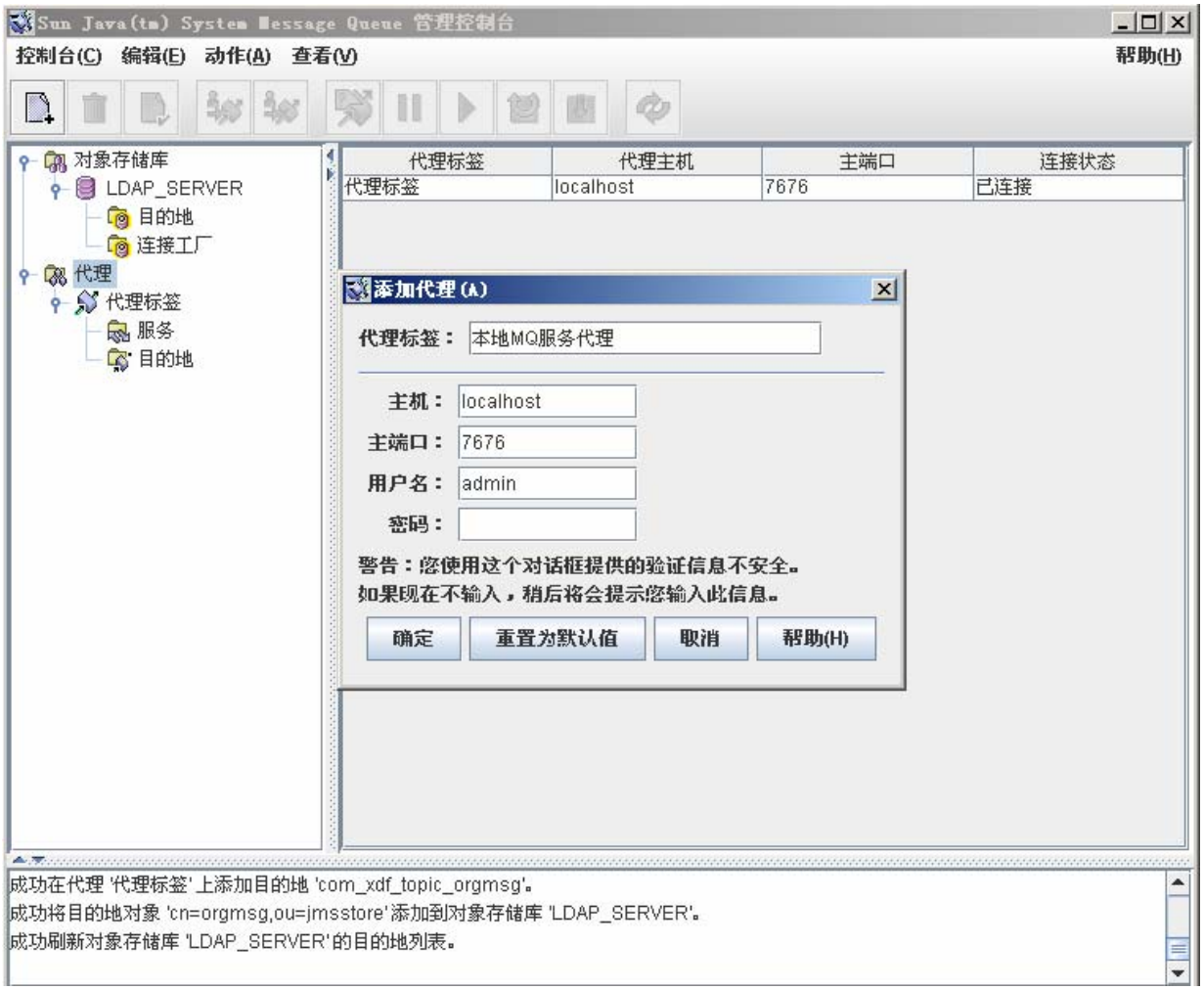
SUN MQ 管理工具可以管理多个 MQ 服务器，因此，每当要管理一个 MQ 服务器，就要在管理工具中创建一个代理，用来指明要管理的 MQ 服务器在哪台计算机上(IP 地址)，端口号是多少，连接该 MQ 服务器的用户名和密码是多少。如下图所示：

代理标签：可以是任意的名字。

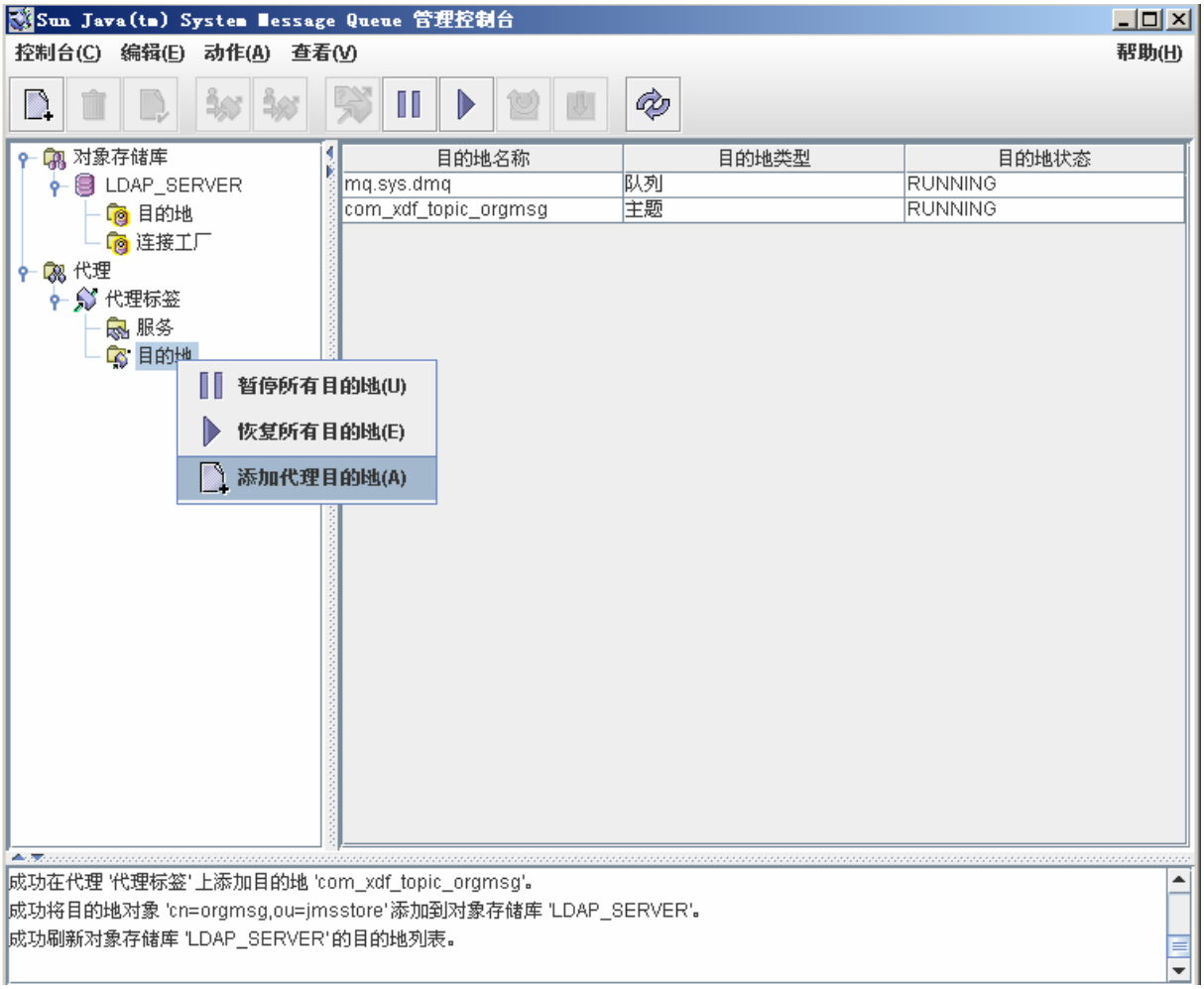
主机：可以是计算机名或者 IP 地址。

主端口：MQ 服务器的端口号，默认是 7676 。

用户名：用于连接 MQ 服务器的用户名。如果不知道用户名和密码，请使用 MessageQueue\bin 目录下的 imqusermgr.exe 命令行工具来创建用户，或者更改密码等。


Sun Java(tm) System Message Queue 管理控制台
控制台(C) 编辑(E) 动作(A) 查看(V) 帮助(H)
对象存储库
LDAP_SERVER
目的地
连接工厂
代理
代理标签
服务
目的地
代理标签 代理主机 主端口 连接状态
代理标签 localhost 7676 已连接
添加代理(A)
代理标签： 本地MQ服务代理
主机： localhost
主端口： 7676
用户名： admin
密码：
警告：您使用这个对话框提供的验证信息不安全。
如果现在不输入，稍后将会提示您输入此信息。
确定 重置为默认值 取消 帮助(H)
成功在代理 '代理标签' 上添加目的地 'com_xdf_topic_orgmsg'。
成功将目的地对象 'cn=orgmsg,ou=jmsstore' 添加到对象存储库 'LDAP_SERVER'。
成功刷新对象存储库 'LDAP_SERVER' 的目的地列表。

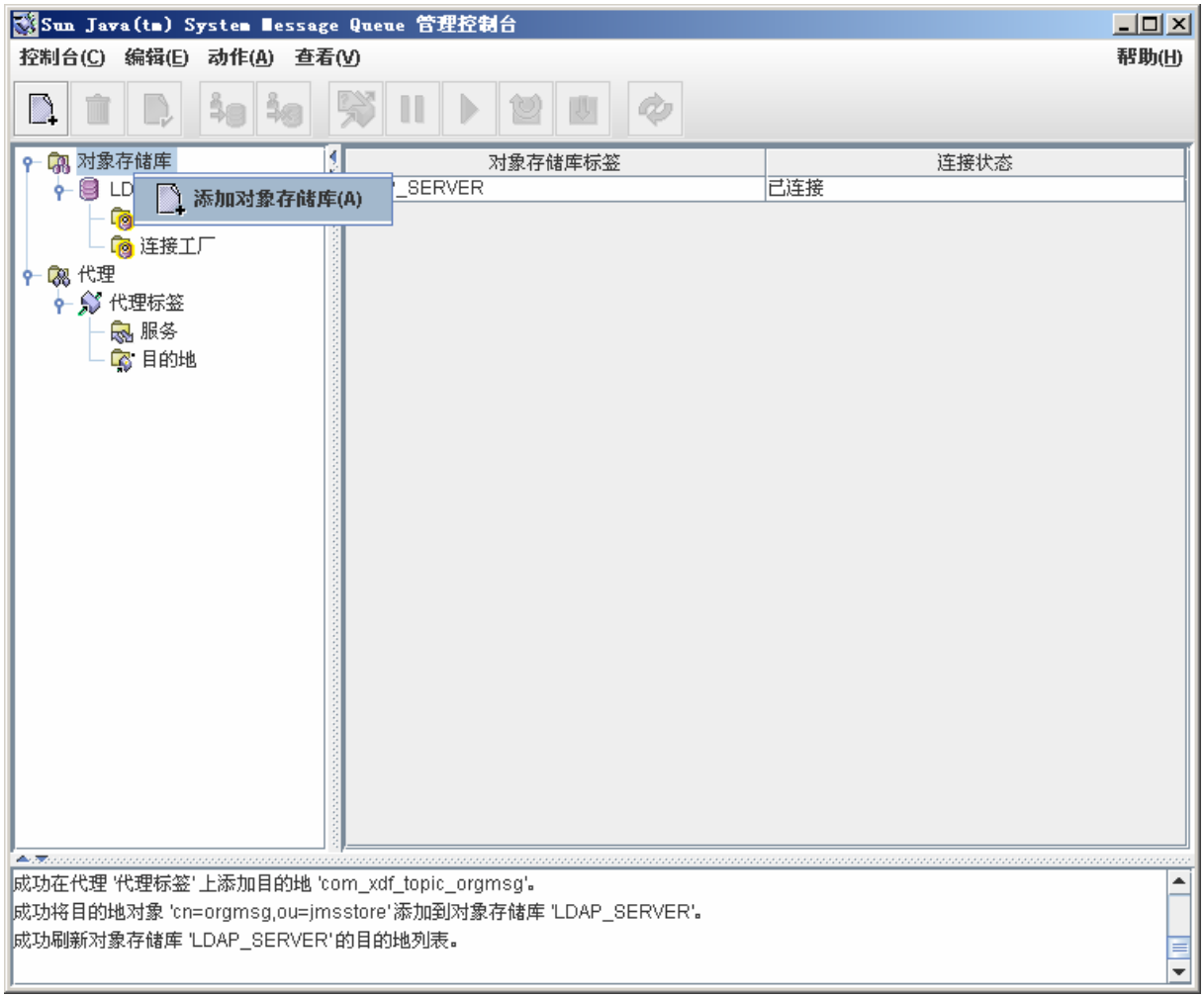
6.2.2 创建目的地

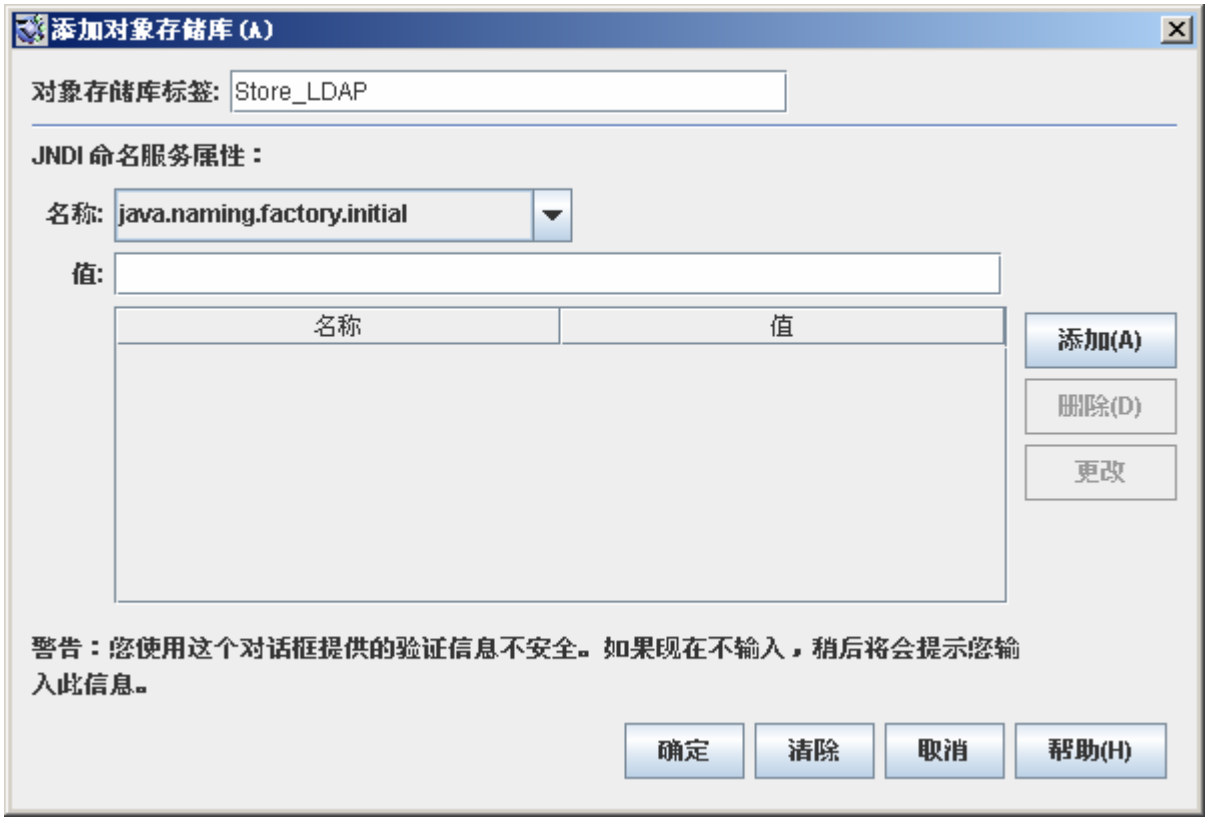


6.3 配置开发所需环境

6.3.1 建立基于 JNDI 的管理对象存储环境

如果想实现使 JMS 客户端应用程序不依赖于具体 JMS 提供者提供的连接及消息目的地的配置，那么应该用将 JMS 连接工厂和目的地对象存储于基于 JNDI 的名字空间中。常见 JNDI 的名字空间实现一般有两种：文件目录系统和 LDAP 系统。这里主要介绍如何通过目录服务器（LDAP）和目录文件系统来存储 JMS 连接工厂和目的地对象。


Sun Java(tm) System Message Queue 管理控制台
控制台(C) 编辑(E) 动作(A) 查看(V) 帮助(H)
对象存储库
添加对象存储库(A)
连接工厂
代理
代理标签
服务
目的地
对象存储库标签
连接状态
_SERVER
已连接
成功在代理 '代理标签' 上添加目的地 'com_xdf_topic_orgmsg'。
成功将目的地对象 'cn=orgmsg,ou=jmsstore' 添加到对象存储库 'LDAP_SERVER'。
成功刷新对象存储库 'LDAP_SERVER' 的目的地列表。


添加对象存储库 (A)
对象存储库标签: Store_LDAP
JNDI 命名服务属性:
名称: java.naming.factory.initial
值:
名称
值
添加(A)
删除(D)
更改
警告：您使用这个对话框提供的验证信息不安全。如果现在不输入，稍后将会提示您输入此信息。
确定
清除
取消
帮助(H)

这里：

对象存储库标签的名字可以任意，用来帮助记忆将 JMS 被管理对象存储于何种 JNDI 名字空间中。

JNDI 命名服务属性是一系列的值对，用来表示连接到命名服务的参数。下面两个表分别给出连接到用 LDAP 和文件系统的参数。

表 6-1 LDAP 存储被管理对象所需要的参数设置

参数名称	含义	参数值举例
java.naming.factory.initial	表示初始化 JNDI 存储服务的类，如果使用 LDAP 存储，必须使用 com.sun.jndi.ldap.LdapCtxFactory	com.sun.jndi.ldap.LdapCtxFactory
java.naming.provider.url	表示连接到 JNDI 服务的 url	ldap://lantian2.xindongfang.com:2922/dc=xindongfang,dc=com
java.naming.security.authentication	表示连接 LDAP 服务的认证方式，可选值有三种： none 表示匿名，不用认证 Simple：表示用户名密码方式的认证 Strong: 使用证书方式方式认证。	Simple
java.naming.security.principal	表示用于认证的身份标识，在 LDAP 中指用户户的 DN	uid=admin,ou=administrators,ou=topologymanagement,o=netscaperoot
java.naming.security.credentials	用来表示给定身份标识的认证对象。如果对于 simple 模式，则指的是密码。	111111

注： 用 LDAP 来存储 JMS 连接工厂和目的对象，我们应当知道 LDAP 服务器访问的 url,

认证方式，用户名以及密码（或证书，根据认证的形式不同而不同）。

表 6-2 用文件系统存储被管理对象所需的参数设置

参数名称	含义	参数值举例
java.naming.factory.initial	表示初始化 JNDI 存储服务的类，如果使用 LDAP 存储，必须使用 com.sun.jndi.fscontext.RefFSContextFactory	com.sun.jndi.fscontext.RefFSContextFactory
java.naming.provider.url	表示连接到 JNDI 服务的 url	file:///C:/myapp/mqobjs

6.3.2 在 LDAP 中存储目的地和连接工厂

一旦能够连上 JNDI 存储服务，那么存储目的地和连接工厂则变得相对简单。
创建目的地窗口如下：

这里需要注意的是：

- **查找名**：必须是 JND 名字空间中，相对于存储对象定义中 url 名称：例如：存储对象 java.naming.provider.url 参数值为：ldap://lantian2.xindongfang.com:2922/dc=xindongfang, dc=com 如果想将目的地对象在 LDAP 中 DN 为 cn=myTopic,ou=jsmStore, dc=xindongfang, dc=com，则查找表必须是：cn=myTopic,ou=jsmStore 。
- **目的地名称**必须与管理代理中定义的目的地名称相一致。

第7章 基于发布/订阅模式的应用范例

7.1 背景

建立一个主题，通过页面向主题中发送消息，同时定义两个消息监听器来异步消费发布到主题的消息，以对同一个消息分别实现两种不同的消息处理。

7.2 实现

用一个 servlet 来接受用户发送消息的请求。同时在该 Servlet 初始化时连接到消息服务器，并注册两个消息监听器。

Servlet： `MessageBroker`

两个消息监听器分别是：

`MessageListenerForOrgMsg`

`MessageListenerForOrgMsg2`