研究 ActiveMQ 的文档

1. 背景

当前，CORBA、DCOM、RMI 等 RPC 中间件技术已广泛应用于各个领域。但是面对规模和复杂度都越来越高的分布式系统，这些技术也显示出其局限性：（1）同步通信：客户发出调用后，必须等待服务对象完成处理并返回结果后才能继续执行；（2）客户和服务对象的生命周期紧密耦合：客户进程和服务对象进程 都必须正常运行；如果由于服务对象崩溃或者网络故障导致客户的请求不可达，客户会接收到异常；（3）点对点通信：客户的一次调用只发送给某个单独的目标对象。

面向消息的中间件（Message Oriented Middleware，MOM）较好的解决了以上问题。发送者将消息发送给消息服务器，消息服务器将消息存放在若干队列中，在合适的时候再将消息转发给接收者。这种模式下，发送和接收是异步的，发送者无需等待；二者的生命周期未必相同：发送消息的时候接收者不一定运行，接收消息的时候发送者也不一定运行； 一对多通信：对于一个消息可以有多个接收者。

已有的 MOM 系统包括 IBM 的 MQSeries、Microsoft 的 MSMQ 和 BEA 的 MessageQ 等。由于没有一个通用的标准，这些系统很难实现互操作和无缝连接。**Java Message Service（JMS）是 SUN 提出的旨在统一各种 MOM 系统接口的规范，它包含点对点（Point to Point，PTP）和发布/订阅（Publish/Subscribe，pub/sub）两种消息模型，提供可靠消息传输、事务和消息过滤等机制。**

2．JMS 概述

2.1 JMS 规范

JAVA 消息服务(JMS)定义了 Java 中访问消息中间件的接口。JMS 只是接口，并没有给予实现，实现 JMS 接口的消息中间件称为 JMS Provider，例如 ActiveMQ。

2.2 术语

JMS Provider：实现 JMS 接口的消息中间件；
PTP：Point to Point，即点对点的消息模型；
Pub/Sub：Publish/Subscribe，即发布/订阅的消息模型；
Queue：队列目标；
Topic：主题目标；
ConnectionFactory：连接工厂，JMS 用它创建连接；
Connection：JMS 客户端到 JMS Provider 的连接；
Destination：消息的目的地；
Session：会话，一个发送或接收消息的线程；

MessageProducer：由 Session 对象创建的用来发送消息的对象；
MessageConsumer：由 Session 对象创建的用来接收消息的对象;
Acknowledge：签收；
Transaction：事务。

2.3 JMS 编程模型

在 JMS 编程模型中，JMS 客户端（组件或应用程序）通过 JMS 消息服务交换消息。消息生产者将消息发送至消息服务，消息消费者则从消息服务接收这些消息。这些消息传送操作是使用一组实现 JMS 应用编程接口（API）的对象（由 JMS Provide 提供）来执行的。

在 JMS 编程模型中，JMS 客户端使用 ConnectionFactory 对象创建一个连接，向消息服务发送消息以及从消息服务接收消息均是通过此连接来进行。Connection 是客户端与消息服务的活动连接。创建连接时，将分配通信资源以及验证客户端。这是一个相当重要的对象，大多数客户端均使用一个连接来进行所有的消息传送。

连接用于创建会话。Session 是一个用于生成和使用消息的单线程上下文。它用于创建发送的生产者和接收消息的消费者，并为所发送的消息定义发送顺序。会话通过大量确认选项或通过事务来支持可靠传送。

客户端使用 MessageProducer 向指定的物理目标（在 API 中表示为目标身份对象）发送消息。生产者可指定一个默认传送模式（持久性消息与非持久性消息）、优先级和有效期值，以控制生产者向物理目标发送的所有消息。

同样，客户端使用 MessageConsumer 对象从指定的物理目标（在 API 中表示为目标对象）接收消息。消费者可使用消息选择器，借助它，消息服务可以只向消费者发送与选择标准匹配的那些消息。

消费者可以支持同步或异步消息接收。异步使用可通过向消费者注册 MessageListener 来实现。当会话线程调用 MessageListener 对象的 onMessage 方法时，客户端将使用消息。

2.4 JMS 编程域

JMS 支持两种截然不同的消息传送模型：PTP（即点对点模型）和 Pub/Sub（即发布/订阅模型），分别称作：PTP Domain 和 Pub/Sub Domain。

PTP（使用 Queue 即队列目标） 消息从一个生产者传送至一个消费者。在此传送模型中，目标是一个队列。消息首先被传送至队列目标，然后根据队列传送策略，从该队列将消息传送至向此队列进行注册的某一个消费者，一次只传送一条消息。可以向队列目标发送消息的生产者的数量没有限制，但每条消息只能发送至、并由一个消费者成功使用。如果没有已经向队列目标注册的消费者，队列将保留它收到的消息，并在某个消费者向该队列进行注册时将消息传送给该消费者。

Pub/Sub（使用 Topic 即主题目标） 消息从一个生产者传送至任意数量的消费者。在此传送模型中，目标是一个主题。消息首先被传送至主题目标，然后传送至所有已订阅此主题的活动消费者。可以向主题目标发送消息的生产者的数量没有限制，并且

每个消息可以发送至任意数量的订阅消费者。主题目标也支持持久订阅的概念。持久订阅表示消费者已向主题目标进行注册，但在消息传送时此消费者可以处于非活动状态。当此消费者再次处于活动状态时，它将接收此信息。如果没有已经向主题目标注册的消费者，主题不保留其接收到的消息，除非有非活动消费者注册了持久订阅。

这两种消息传送模型使用表示不同编程域的 API 对象（其语义稍有不同）进行处理，如下所示：

基本类型（统一域）	PTP 域	Pub/Sub 域
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicPublisher
Destination（Queue 或 Topic）	Queue	Topic
MessageProducer	QueueSender	
MessageConsumer	QueueReceiver,QueueBrowser	TopicSubscriber

使用图表第一列中列出的统一域对象编写点对点和发布/订阅消息传送。这是首选方法（JMS 1.1 规范）。然而，为了符合早期的 JMS 1.02b 规范，可以使用 PTP 域对象编写点对点消息传送，使用 Pub/Sub 域对象编制发布/订阅消息传送。

2.5 JMS 消息结构

JMS 消息由以下几部分组成：消息头，属性和消息体。

2.5.1 消息头(Header)

消息头包含消息的识别信息和路由信息，消息头包含一些标准的属性如：JMSDestination，JMSMessageID 等。

消息头	由谁设置
JMSDestination	send 方法
JMSDeliveryMode	send 方法
JMSExpiration	send 方法
JMSPriority	send 方法
JMSMessageID	send 方法
JMSTimestamp	send 方法
JMSCorrelationID	客户
JMSReplyTo	客户
JMSType	客户

JMSRedelivered	JMS Provider

2.5.2 属性(Properties)

除了消息头中定义好的标准属性外，JMS 提供一种机制增加新属性到消息头中，这种新属性包含以下几种：

1. 应用需要用到的属性;
2. 消息头中原有的一些可选属性;
3. JMS Provider 需要用到的属性。

标准的 JMS 消息头包含以下属性：

消息头	描述
JMSDestination	消息发送的目的地
JMSDeliveryMode	传送模式， 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送的可靠性和吞吐量之间找到平衡点。
JMSExpiration	消息过期时间，等于 Destination 的 send 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。
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
JMSType	消息类型的识别符。
JMSRedelivered	如果一个客户端收到一个设置了 JMSRedelivered 属性的消息，则表示可能客户端曾经在早些时候收到过该消息，但并没有签收(acknowledged)。

2.5.3 消息体(Body)

JMS API 定义了 5 种消息体格式，也叫消息类型，可以使用不同形式发送接收数据并可以兼容现有的消息格式，下面描述这 5 种类型：

消息类型	消息体
TextMessage	java.lang.String 对象，如 xml 文件内容
MapMessage	名/值对的集合，名是 String 对象，值类型可以是 Java 任何基本类型
BytesMessage	字节流
StreamMessage	Java 中的输入输出流
ObjectMessage	Java 中的可序列化对象
Message	没有消息体，只有消息头和属性

2.6 PTP 模型

PTP(Point-to-Point)模型是基于队列的，生产者发消息到队列，消费者从队列接收消息，队列的存在使得消息的异步传输成为可能。和邮件系统中的邮箱一样，队列可以包含各种消息，JMS Provider 提供工具管理队列的创建、删除。JMS PTP 模型定义了客户端如何向队列发送消息，从队列接收消息，浏览队列中的消息。

下面描述 JMS PTP 模型中的主要概念和对象：

名称	描述
ConnectionFactory	客户端用 ConnectionFactory 创建 Connection 对象。
Connection	一个到 JMS Provider 的连接，客户端可以用 Connection 创建 Session 来发送和接收消息。
Session	客户端用 Session 创建 MessageProducer 和 MessageConsumer 对象。如果在 Session 关闭时，有一些消息已经被收到，但还没有被签收(acknowledged)，那么，当消费者下次连接到相同的队列时，这些消息还会被再次接收。
Destination（Queue 或 TemporaryQueue）	客户端用 Session 创建 Destination 对象。此处的目标为队列，队列由队列名识别。临时队列只能由创建它的 Connection 所创建的消费者消费，但是任何生产者都可向临时队列发送消息。
MessageProducer	客户端用 MessageProducer 发送消息到队列。
MessageConsumer	客户端用 MessageConsumer 接收队列中的消息，如果用户在 receive 方法中设定了消息选择条件，那么不符合条件的消息会留在队列中，不会被接收到。
可靠性(Reliability)	队列可以长久地保存消息直到消费者收到消息。消费者不需要因为担心消息会丢失而时刻和队列保持激活的连接状态，充分体现了异步传输模式的优势。

2.7 PUB/SUB 模型

JMS Pub/Sub 模型定义了如何向一个内容节点发布和订阅消息，这些节点被称作主题(topic)。

主题可以被认为是消息的传输中介，发布者(publisher)发布消息到主题，订阅者(subscribe) 从主题订阅消息。主题使得消息订阅者和消息发布者保持互相独立，不需要接触即可保证消息的传送。

下面描述 JMS Pub/Sub 模型中的主要概念和对象：

名称	描述
订阅(subscription)	消息订阅分为非持久订阅(non-durable subscription)和持久订阅(durable subscrip-tion)，非持久订阅只有当客户端处于激活状态，也就是和 JMS Provider 保持连接状态才能收到发送到某个主题的消息，而当客户端处于离线状态，这个时间段发到主题的消息将会丢失，永远不会收到。持久订阅时，客户端向 JMS 注册一个识别自己身份的 ID，当这个客户端处于离线时，JMS Provider 会为这个 ID 保存所有发送到主题的消息，当客户再次连接到 JMS Provider 时，会根据自己的 ID 得到所有当自己处于离线时发送到主题的消息。
ConnectionFactory	客户端用 ConnectionFactory 创建 Connection 对象。
Connection	一个到 JMS Provider 的连接，客户端可以用 Connection 创建 Session 来发送和接收消息。
Session	客户端用 Session 创建 MessageProducer 和 MessageConsumer 对象。它还提供持久订阅主题，或使用 unsubscribe 方法取消消息的持久订阅。
Destination (Topic 和 TemporaryTopic)	客户端用 Session 创建 Destination 对象。此处的目标为主题，主题由主题名识别。临时主题只能由创建它的 Connection 所创建的消费者消费。临时主题不能提供持久订阅功能。JMS 没有给出主题的组织和层次结构的定义，由 JMS Provider 自己定义。
MessageProducer	客户端用 MessageProducer 发布消息到主题。
MessageConsumer	客户端用 MessageConsumer 接收发布到主题上的消息。可以在 receive 中设置消息过滤功能，这样，不符合要求的消息不会被接收。
恢复和重新派送(Recovery and Redelivery)	非持久订阅状态下，不能恢复或重新派送一个未签收的消息。只有持久订阅才能恢复或重新派送一个未签收的消息。
可靠性(Reliability)	当所有的消息必须被接收，则用持久订阅模式。当丢失消息能够被容忍，则用非持久订阅模式。

2.8 JMS 支持并发

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

3. ActiveMQ 安装

3.1 版本

jdk 版本：jdk1.5.0_11
ActiveMQ 版本：ActiveMQ 4.2 测试版
C++客户端版本：ActiveMQ CPP 1.1 Release

3.2 ActiveMQ 二进制安装

```
gunzip apache-activemq-4.2-20070328.130210-35.tar.gz
tar xvf apache-activemq-4.2-20070328.130210-35.tar
```

设置 ActiveMQ 环境变量：ACTIVEMQ_HOME=安装目录
设置CLASSPATH环境变量, CLASSPATH=$CLASSPATH:$ACTIVEMQ_HOME/ apache-activemq-4.2-SNAPSHOT.jar

3.3 ActiveMQ 移植

只需将$ACTIVEMQ_HOME 打包移植到新机器即可。

3.4 C++客户端编译

安装 perl-5.8.8.tar.gz

```
tar xzvf perl-5.8.8.tar.gz
cd perl-5.8.8
rm -f config.sh Policy.sh
```

```
sh Configure -de -Dprefix=/usr
make
make test
make install
reboot
```

系统重新启动，登录系统后可以执行 perl -v 查看 Perl 版本信息

检查/usr/bin/perl 或/usr/local/bin/perl 是否指向新版本的 perl

安装 m4-1.4.8.tar.gz

```
tar xzvf m4-1.4.8.tar.gz
cd m4-1.4.8
./configure
make
make install
```

安装 autoconf-2.59.tar.gz

```
tar xzvf autoconf-2.59.tar.gz
cd autoconf-2.59
./configure
make
make install
```

安装 automake-1.9.6.tar.gz

```
tar xzvf automake-1.9.6.tar.gz
cd automake-1.9.6
./configure
make
make install
```

安装 libtool-1.5.22.tar.gz

```
tar xzvf libtool-1.5.22.tar.gz
cd libtool-1.5.22
./configure
make
make install
```

安装 cppunit-1.10.2.tar.gz

```
tar xzvf cppunit-1.10.2.tar.gz
cd cppunit-1.10.2
./configure
make
```

```
make install
```

安装 e2fsprogs-1.38.tar.gz

注：此项仅 Solaris8 需要安装，Solaris9 已自带此 uuid 头文件和库文件

```
tar xzvf e2fsprogs-1.38.tar.gz
cd e2fsprogs-1.38
./configure
make
make install
```

安装 gcc-3.4.6

注：无需自行编译，直接从 http://www.sunfreeware.com 下载对应 solaris 版本和 cpu 的 gcc-3.4.6 包文件即可

```
gunzip gcc-3.4.6-sol8-sparc-local.gz
pkgadd –d gcc-3.4.6-sol8-sparc-local
```

安装 activemq-cpp-1.1.tar.gz

```
tar xzvf activemq-cpp-1.1.tar.gz
cd activemq-cpp-1.1
./autogen.sh
./configure
make
make install
复制 activemq-cpp-1.1/include/activemq-cpp-1.1 下文件至/usr/include
复制 activemq-cpp-1.1/lib/libactivemq-cpp.a 至/usr/lib
```

3.5 C++客户端移植

1. 在新机器安装 gcc-3.4.6；

2. 复制 activemq-cpp-1.1/include/activemq-cpp-1.1 下文件至新机器/usr/include，复制 activemq-cpp-1.1/lib/ libactivemq-cpp.a 至新机器/usr/lib；

如果新机器为 Solaris8 还需如下操作：复制在 Solaris8 下编译 e2fsprogs-1.38.tar.gz 产生的 uuid 头文件和库文件至新机器相应的目录(即/usr/include/uuid/uuid.h 和/usr/lib/ libuuid.a，此处复制需注意文件层次)

3.6 启动

```
cd $ACTIVEMQ_HOME/bin
./activemq
或者
./activemq > activemq.log  2>&1 &
```

3.6 停止

```
ps –ef|grep activemq
kill 进程号
```

4. ActiveMQ 编程

ActiveMQ 特色：

Supports a variety of Cross Language Clients and Protocols from Java, C, C++, C#, Ruby, Perl, Python, PHP

OpenWire for high performance clients in Java, C, C++, C#

Stomp support so that clients can be written easily in C, Ruby, Perl, Python, PHP to talk to ActiveMQ as well as any other popular Message Broker

Supports many advanced features such as Message Groups, Virtual Destinations, Wildcards and Composite Destinations

Fully supports JMS 1.1 and J2EE 1.4 with support for transient, persistent, transactional and XA messaging

Spring Support so that ActiveMQ can be easily embedded into Spring applications and configured using Spring's XML configuration mechanism

Tested inside popular J2EE servers such as Geronimo, JBoss 4, GlassFish and WebLogic

Includes JCA 1.5 resource adaptors for inbound & outbound messaging so that ActiveMQ should auto-deploy in any J2EE 1.4 compliant server

Supports pluggable transport protocols such as in-VM, TCP, SSL, NIO, UDP, multicast, JGroups and JXTA transports

Supports very fast persistence using JDBC along with a high performance journal

Designed for high performance clustering, client-server, peer based communication

REST API to provide technology agnostic and language neutral web based API to messaging

Ajax to support web streaming support to web browsers using pure DHTML, allowing web browsers to be part of the messaging fabric

Axis Support so that ActiveMQ can be easily dropped into Apache Axis runtimes to provide reliable messaging

Can be used as an in memory JMS provider, ideal for unit testing JMS

本文档仅描述最基本的使用方法，实际使用过程中请以官方文档为准。

1. ActiveMQ 官方网站：http://activemq.apache.org
2. JMS 官方网站：http://java.sun.com/products/jms

4.1 开发 JSM 的步骤

广义上说，一个 JMS 应用是几个 JMS 客户端交换消息，开发 JMS 客户端应用由以下几步构成：

用 JNDI 得到 ConnectionFactory 对象；
用 ConnectionFactory 创建 Connection 对象；
用 Connection 对象创建一个或多个 JMS Session；
用 JNDI 得到目标队列或主题对象，即 Destination 对象；
用 Session 和 Destination 创建 MessageProducer 和 MessageConsumer；
通知 Connection 开始传送消息。

4.2 编程模版

4.2.1 ConnectionFactory

要初始化 JMS，则需要使用连接工厂。客户端通过创建 ConnectionFactory 建立到 ActveMQ 的连接，一个连接工厂封装了一组连接配置参数，这组参数在配置 ActiveMQ 时已经定义，例如 brokerURL 参数，此参数传入的是 ActiveMQ 服务地址和端口，支持 openwire 协议的默认连接为 tcp://localhost:61616，支持 stomp 协议的默认连接为 tcp://localhost:61613。

注：由于 C++客户端暂时仅支持 stomp 协议，所以需要使用 tcp://localhost:61613。

ConnectionFactory 支持并发。

Java 客户端：

ActiveMQConnectionFactory 构造方法：

```
ActiveMQConnectionFactory();
ActiveMQConnectionFactory(String brokerURL);
    ActiveMQConnectionFactory(String userName, String password, String brokerURL) ;
    ActiveMQConnectionFactory(String userName, String password, URI brokerURL) ;
ActiveMQConnectionFactory(URI brokerURL);
```

其中 brokerURL 为 ActiveMQ 服务地址和端口。

例如：

```
    ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://192.168.0.135:61616");
    或者
    ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory();
    connectionFactory. setBrokerURL("tcp://192.168.0.135:61616");
```

C++客户端：

ActiveMQConnectionFactory 构造函数：

```
ActiveMQConnectionFactory(void)；
ActiveMQConnectionFactory( const std::string& url,
     const std::string& username = "",
     const std::string& password = "",
     const std::string& clientId = "" );
```

例如：

```
    ActiveMQConnectionFactory* connectionFactory = new ActiveMQConnectionFactory("tcp://192.
168.0.135:61613");
    或者
    ActiveMQConnectionFactory* connectionFactory = new ActiveMQConnectionFactory();
    connectionFactory-> setBrokerURL("tcp://192.168.0.135:61613");
```

4.2.2 Connection

在成功创建正确的 ConnectionFactory 后，下一步将是创建一个连接，它是 JMS 定义的一个接口。ConnectionFactory 负责返回可以与底层消息传递系统进行通信的 Connection 实现。通常客户端只使用单一连接。根据 JMS 文档，Connection 的目的是“利用 JMS 提供者封装开放的连接”，以及表示“客户端与提供者服务例程之间的开放 TCP/IP 套接字”。该文档还指出 Connection 应该是进行客户端身份验证的地方，除了其他一些事项外，客户端还可以指定惟一标志符。

当一个 Connection 被创建时，它的传输默认是关闭的，必须使用 start 方法开启。

一个 Connection 可以建立一个或多个的 Session。

当一个程序执行完成后，必须关闭之前创建的 Connection，否则 ActiveMQ 不能释放资源，关闭一个 Connection 同样也关闭了 Session，MessageProducer 和 MessageConsumer。

Connection 支持并发。

4.2.2.1 创建 Connection

Java 客户端：

ActiveMQConnectionFactory 方法：

```
Connection createConnection()；
Connection createConnection(String userName, String password);
```

例如：

```
    Connection connection = connectionFactory.createConnection();
```

C++客户端：

函数原型：

```
cms::Connection* ActiveMQConnectionFactory::createConnection(void)
```

```
    throw ( cms::CMSException );
cms::Connection* ActiveMQConnectionFactory::createConnection(
    const std::string& username,
    const std::string& password,
    const std::string& clientId )
        throw ( cms::CMSException );
```

例如：

```
Connection* connection = connectionFactory->createConnection();
```

4.2.2.2 开启 Connection

Java 客户端：

ActiveMQConnection 方法：

```
void start();
```

例如：

```
Connection.start();
```

C++客户端：

函数原型：

```
void ActiveMQConnection::start(void) throw ( cms::CMSException );
```

例如：

```
connection->start();
```

4.2.2.3 关闭 Connection

Java 客户端：

ActiveMQConnection 方法：

```
void close();
```

例如：

```
Connection.close();
```

C++客户端：

函数原型：

```
void ActiveMQConnection::close(void) throw ( cms::CMSException );
```

例如：

```
connection->close();
```

4.2.3 Session

一旦从 ConnectionFactory 中获得一个 Connection，就必须从 Connection 中创建一个或者多个 Session。Session 是一个发送或接收消息的线程，可以使用 Session 创建 MessageProducer，MessageConsumer 和 Message。

Session 可以被事务化，也可以不被事务化，通常，可以通过向 Connection 上的适当创建方法传递一个布尔参数对此进行设置。

Java 客户端：

ActiveMQConnection 方法：

```
Session createSession(boolean transacted, int acknowledgeMode);
```

其中 transacted 为使用事务标识，acknowledgeMode 为签收模式。

例如：

```
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

C++客户端：

函数原型：

```
cms::Session* ActiveMQConnection::createSession(void);
cms::Session* ActiveMQConnection::createSession(
   cms::Session::AcknowledgeMode ackMode );
```

例如：

```
    Session* session = connection->createSession( Session::AUTO_ACKNOWLEDGE );
```

4.2.4 Destination

Destination 是一个客户端用来指定生产消息目标和消费消息来源的对象。

在 PTP 模式中，Destination 被称作 Queue 即队列；在 Pub/Sub 模式，Destination 被称作 Topic 即主题。在程序中可以使用多个 Queue 和 Topic。

Java 客户端：

ActiveMQSession 方法：

```
Queue createQueue(String queueName);
TemporaryQueue createTemporaryQueue();
Topic createTopic(String topicName);
TemporaryTopic createTemporaryTopic();
```

例如：

```
    Destination destination = session.createQueue("TEST.FOO");

    或者

    Destination destination = session.createTopic("TEST.FOO");
```

C++客户端：

函数原型：

```
cms::Queue* ActiveMQSession::createQueue( const std::string& queueName
)
    throw ( cms::CMSException );
cms::TemporaryQueue* ActiveMQSession::createTemporaryQueue(void)
    throw ( cms::CMSException );
cms::Topic* ActiveMQSession::createTopic( const std::string& topicName
)
```

```
    throw ( cms::CMSException );
cms::TemporaryTopic* ActiveMQSession::createTemporaryTopic(void)
    throw ( cms::CMSException );
```

例如：

```
Destination* destination = session->createQueue( "TEST.FOO" );
或者
Destination* destination = session->createTopic( "TEST.FOO" );
```

4.2.5 MessageProducer

MessageProducer 是一个由 Session 创建的对象，用来向 Destination 发送消息。

4.2.5.1 创建 MessageProducer

Java 客户端：

ActiveMQSession 方法：

```
MessageProducer createProducer(Destination destination);
```

例如：

```
MessageProducer producer = session.createProducer(destination);
```

C++客户端：

函数原型：

```
cms::MessageProducer* ActiveMQSession::createProducer(
    const cms::Destination* destination );
```

例如：

```
MessageProducer* producer = session->createProducer( destination );
```

4.2.5.2 发送消息

Java 客户端：

ActiveMQMessageProducer 方法：

```
void send(Destination destination, Message message);
void send(Destination destination, Message message, int deliveryMode, in
    t priority,
long timeToLive);
void send(Message message);
void send(Message message, int deliveryMode, int priority, long timeToL
ive);
```

其中 deliveryMode 为传送模式，priority 为消息优先级，timeToLive 为消息过期时间。

例如：

```
    producer.send(message);
```

C++客户端：

函数原型：

```
void ActiveMQProducer::send( cms::Message* message )
    throw ( cms::CMSException );
void ActiveMQProducer::send( cms::Message* message, int deliveryMode,
    int priority,
    long long timeToLive )
        throw ( cms::CMSException );
void ActiveMQProducer::send( const cms::Destination* destination,
    cms::Message* message) throw ( cms::CMSException );
void ActiveMQProducer::send( const cms::Destination* destination,
    cms::Message* message, int deliveryMode,
    int priority, long long timeToLive)
        throw ( cms::CMSException );
```

例如：

```
    producer->send( message );
```

4.2.6 MessageConsumer

MessageConsumer 是一个由 Session 创建的对象，用来从 Destination 接收消息。

4.2.6.1 创建 MessageConsumer

Java 客户端：

ActiveMQSession 方法：

```
MessageConsumer createConsumer(Destination destination);
MessageConsumer createConsumer(Destination destination, String messageS
elector);
MessageConsumer createConsumer(Destination destination, String messageSe
    lector, boolean noLocal);
TopicSubscriber createDurableSubscriber(Topic topic, String name);
TopicSubscriber createDurableSubscriber(Topic topic, String name, String
     messageSelector, boolean noLocal);
```

其中 messageSelector 为消息选择器；noLocal 标志默认为 false，当设置为 true 时限制消费者只能接收和自己相同的连接（Connection）所发布的消息，此标志只适用于主题，不适用于队列；name 标识订阅主题所对应的订阅名称，持久订阅时需要设置此参数。

例如：

```
    MessageConsumer consumer = session.createConsumer(destination);
```

C++客户端：

函数原型：

```
cms::MessageConsumer* ActiveMQSession::createConsumer(
    const cms::Destination* destination );
cms::MessageConsumer* ActiveMQSession::createConsumer(
    const cms::Destination* destination,
    const std::string& selector )
        throw ( cms::CMSException );
cms::MessageConsumer* ActiveMQSession::createConsumer(
    const cms::Destination* destination,
    const std::string& selector,
    bool noLocal )
        throw ( cms::CMSException );
cms::MessageConsumer* ActiveMQSession::createDurableConsumer(
    const cms::Topic* destination,
    const std::string& name,
    const std::string& selector,
    bool noLocal )
        throw ( cms::CMSException );
```

例如：

```
    MessageConsumer* consumer = session->createConsumer( destination );
```

4.2.6.2 消息的同步和异步接收

消息的同步接收是指客户端主动去接收消息，客户端可以采用 MessageConsumer 的 receive 方法去接收下一个消息。

消息的异步接收是指当消息到达时，ActiveMQ 主动通知客户端。客户端可以通过注册一个实现 MessageListener 接口的对象到 MessageConsumer。MessageListener 只有一个必须实现的方法 —— onMessage，它只接收一个参数，即 Message。在为每个发送到 Destination 的消息实现 onMessage 时，将调用该方法。

Java 客户端：

ActiveMQMessageConsumer 方法：

```
Message receive()
Message receive(long timeout)
Message receiveNoWait()
```

其中 timeout 为等待时间，单位为毫秒。

或者

实现 MessageListener 接口，每当消息到达时，ActiveMQ 会调用 MessageListener 中的 onMessage 函数。

例如：

```
    Message message = consumer.receive();
```

C++客户端：

函数原型：

```
cms::Message* ActiveMQConsumer::receive() throw ( cms::CMSException )
cms::Message* ActiveMQConsumer::receive( int millisecs )
    throw ( cms::CMSException );
cms::Message* ActiveMQConsumer::receiveNoWait(void)
    throw ( cms::CMSException );
```

或者

实现 MessageListener 接口，每当消息到达时，ActiveMQ 会调用 MessageListener 中的 onMessage 函数。

例如：

```
Message *message = consumer->receive();
或者
consumer->setMessageListener( this );
virtual void onMessage( const Message* message ){
 //process message
}
```

4.2.6.3 消息选择器

JMS 提供了一种机制，使用它，消息服务可根据消息选择器中的标准来执行消息过滤。生产者可在消息中放入应用程序特有的属性，而消费者可使用基于这些属性的选择标准来表明对消息是否感兴趣。这就简化了客户端的工作，并避免了向不需要这些消息的消费者传送消息的开销。然而，它也使得处理选择标准的消息服务增加了一些额外开销。

消息选择器是用于 MessageConsumer 的过滤器，可以用来过滤传入消息的属性和消息头部分（但不过滤消息体），并确定是否将实际消费该消息。按照 JMS 文档的说法，消息选择器是一些字符串，它们基于某种语法，而这种语法是 SQL-92 的子集。可以将消息选择器作为 MessageConsumer 创建的一部分。

Java 客户端：

例如：

```
public final String SELECTOR = “ JMSType = ‘ TOPIC_PUBLISHER ’ ”;
```

该选择器检查了传入消息的 JMSType 属性，并确定了这个属性的值是否等于 TOPIC_PUBLISHER。如果相等，则消息被消费；如果不相等，那么消息会被忽略。

4.2.7 Message

JMS 程序的最终目的是生产和消费的消息能被其他程序使用，JMS 的 Message 是一个既简单又不乏灵活性的基本格式，允许创建不同平台上符合非 JMS 程序格式的消息。Message 由以下几部分组成：消息头，属性和消息体。

Java 客户端：

ActiveMQSession 方法：

```
BlobMessage createBlobMessage(File file)
BlobMessage createBlobMessage(InputStream in)
BlobMessage createBlobMessage(URL url)
BlobMessage createBlobMessage(URL url, boolean deletedByBroker)
BytesMessage createBytesMessage()
MapMessage createMapMessage()
Message createMessage()
ObjectMessage createObjectMessage()
ObjectMessage createObjectMessage(Serializable object)
TextMessage createTextMessage()
TextMessage createTextMessage(String text)
```

例如：

下例演示创建并发送一个 TextMessage 到一个队列：

```
    TextMessage message = queueSession.createTextMessage();
    message.setText(msg_text); // msg_text is a String
    queueSender.send(message);
```

下例演示接收消息并转换为合适的消息类型：

```
    Message m = queueReceiver.receive();
    if (m instanceof TextMessage) {
     TextMessage message = (TextMessage) m;
     System.out.println("Reading message: " + message.getText());
} else {
     // Handle error
}
```

C++客户端：

函数原型：

```
cms::Message* ActiveMQSession::createMessage(void)
    throw ( cms::CMSException )
cms::BytesMessage* ActiveMQSession::createBytesMessage(void)
    throw ( cms::CMSException )
cms::BytesMessage* ActiveMQSession::createBytesMessage(
    const unsigned char* bytes,
    unsigned long long bytesSize )
        throw ( cms::CMSException )
cms::TextMessage* ActiveMQSession::createTextMessage(void)
    throw ( cms::CMSException )
cms::TextMessage* ActiveMQSession::createTextMessage( const std::string
& text )
    throw ( cms::CMSException )
cms::MapMessage* ActiveMQSession::createMapMessage(void)
```

throw (cms::CMSException)

例如：

下例演示创建并发送一个 TextMessage 到一个队列：

```
TextMessage* message = session->createTextMessage( text ); // text is a string

producer->send( message );

delete message;
```

下例演示接收消息：

```
Message *message = consumer->receive();

const TextMessage* textMessage = dynamic_cast< const TextMessage* >( message );

string text = textMessage->getText();

printf( "Received: %s\n", text.c_str() );

delete message;
```

4.3 可靠性机制

发送消息最可靠的方法就是在事务中发送持久性的消息，ActiveMQ 默认发送持久性消息。结束事务有两种方法：提交或者回滚。当一个事务提交，消息被处理。如果事务中有一个步骤失败，事务就回滚，这个事务中的已经执行的动作将被撤销。

接收消息最可靠的方法就是在事务中接收信息，不管是从 PTP 模式的非临时队列接收消息还是从 Pub/Sub 模式持久订阅中接收消息。

对于其他程序，低可靠性可以降低开销和提高性能，例如发送消息时可以更改消息的优先级或者指定消息的过期时间。

消息传送的可靠性越高，需要的开销和带宽就越多。性能和可靠性之间的折衷是设计时要重点考虑的一个方面。可以选择生成和使用非持久性消息来获得最佳性能。另一方面，也可以通过生成和使用持久性消息并使用事务会话来获得最佳可靠性。在这两种极端之间有许多选择，这取决于应用程序的要求。

4.3.1 基本可靠性机制

4.3.1.1 控制消息的签收（Acknowledgment）

客户端成功接收一条消息的标志是这条消息被签收。成功接收一条消息一般包括如下三个阶段：

1. 客户端接收消息；
2. 客户端处理消息；
3. 消息被签收。签收可以由 ActiveMQ 发起，也可以由客户端发起，取决于 Session 签收模式的设置。

在带事务的 Session 中，签收自动发生在事务提交时。如果事务回滚，所有已经接收的消息将会被再次传送。

在不带事务的 Session 中，一条消息何时和如何被签收取决于 Session 的设置。

1. Session.AUTO_ACKNOWLEDGE

当客户端从 receive 或 onMessage 成功返回时，Session 自动签收客户端的这条消息的收条。在 AUTO_ACKNOWLEDGE 的 Session 中，同步接收 receive 是上述三个阶段的一个例外，在这种情况下，收条和签收紧随在处理消息之后发生。

2. Session.CLIENT_ACKNOWLEDGE

客户端通过调用消息的 acknowledge 方法签收消息。在这种情况下，签收发生在 Session 层面：签收一个已消费的消息会自动地签收这个 Session 所有已消费消息的收条。

3. Session.DUPS_OK_ACKNOWLEDGE

此选项指示 Session 不必确保对传送消息的签收。它可能引起消息的重复，但是降低了 Session 的开销，所以只有客户端能容忍重复的消息，才可使用（如果 ActiveMQ 再次传送同一消息，那么消息头中的 JMSRedelivered 将被设置为 true）。

Java 客户端：

签收模式分别为：

1. Session.AUTO_ACKNOWLEDGE
2. Session.CLIENT_ACKNOWLEDGE
3. Session.DUPS_OK_ACKNOWLEDGE

ActiveMQConnection 方法：

```
Session createSession(boolean transacted, int acknowledgeMode);
```

例如：

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

C++客户端：

签收模式分别为：

1. Session::AUTO_ACKNOWLEDGE
2. Session::CLIENT_ACKNOWLEDGE
3. Session::DUPS_OK_ACKNOWLEDGE
4. Session::SESSION_TRANSACTED

函数原型：

```
cms::Session* ActiveMQConnection::createSession(
   cms::Session::AcknowledgeMode ackMode )
      throw ( cms::CMSException )
```

例如：

```
Session* session = connection->createSession( Session::AUTO_ACKNOWLEDGE );
```

对队列来说，如果当一个 Session 终止时它接收了消息但是没有签收，那么 ActiveMQ 将保留这些消息并将再次传送给下一个进入队列的消费者。

对主题来说，如果持久订阅用户终止时，它已消费未签收的消息也将被保留，直到再次传送给这个用户。对于非持久订阅，AtiveMQ 在用户 Session 关闭时将删除这些消息。

如果使用队列和持久订阅，并且 Session 没有使用事务，那么可以使用 Session 的 recover 方法停止 Session，再次启动后将收到它第一条没有签收的消息，事实上，重

启后 Session 一系列消息的传送都是以上一次最后一条已签收消息的下一条为起点。如果这时有消息过期或者高优先级的消息到来，那么这时消息的传送将会和最初的有所不同。对于非持久订阅用户，重启后，ActiveMQ 有可能删除所有没有签收的消息。

4.3.1.2 指定消息传送模式

ActiveMQ 支持两种消息传送模式：PERSISTENT 和 NON_PERSISTENT 两种。

1. PERSISTENT（持久性消息）

这是 ActiveMQ 的默认传送模式，此模式保证这些消息只被传送一次和成功使用一次。对于这些消息，可靠性是优先考虑的因素。可靠性的另一个重要方面是确保持久性消息传送至目标后，消息服务在向消费者传送它们之前不会丢失这些消息。这意味着在持久性消息传送至目标时，消息服务将其放入持久性数据存储。如果消息服务由于某种原因导致失败，它可以恢复此消息并将此消息传送至相应的消费者。虽然这样增加了消息传送的开销，但却增加了可靠性。

2. NON_PERSISTENT（非持久性消息）

保证这些消息最多被传送一次。对于这些消息，可靠性并非主要的考虑因素。此模式并不要求持久性的数据存储，也不保证消息服务由于某种原因导致失败后消息不会丢失。

有两种方法指定传送模式：

1. 使用 setDeliveryMode 方法，这样所有的消息都采用此传送模式；
2. 使用 send 方法为每一条消息设置传送模式；

Java 客户端：

传送模式分别为：

1. DeliveryMode.PERSISTENT
2. DeliveryMode.NON_PERSISTENT

ActiveMQMessageProducer 方法：

```
void setDeliveryMode(int newDeliveryMode);
```

或者

```
void send(Destination destination, Message message, int deliveryMode, in
    t priority,
long timeToLive);
 void send(Message message, int deliveryMode, int priority, long timeToL
ive);
```

其中 deliveryMode 为传送模式，priority 为消息优先级，timeToLive 为消息过期时间。

例如：

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

C++客户端：

传送模式分别为：

1. DeliveryMode::PERSISTANT
2. DeliveryMode::NON_PERSISTANT

函数原型：

```
void setDeliveryMode( int mode );
```

或者

```
void ActiveMQProducer::send( cms::Message* message, int deliveryMode,
    int priority,
    long long timeToLive )
    throw ( cms::CMSException );
void ActiveMQProducer::send( const cms::Destination* destination,
    cms::Message* message, int deliveryMode,
    int priority, long long timeToLive)
        throw ( cms::CMSException );
```

例如：

```
producer->setDeliveryMode( DeliveryMode::NON_PERSISTANT );
```

如果不指定传送模式，那么默认是持久性消息。如果容忍消息丢失，那么使用非持久性消息可以改善性能和减少存储的开销。

4.3.1.3 设置消息优先级

通常，可以确保将单个会话向目标发送的所有消息按其发送顺序传送至消费者。然而，如果为这些消息分配了不同的优先级，消息传送系统将首先尝试传送优先级较高的消息。

有两种方法设置消息的优先级：

1. 使用 setDeliveryMode 方法，这样所有的消息都采用此传送模式；
2. 使用 send 方法为每一条消息设置传送模式；

Java 客户端：

ActiveMQMessageProducer 方法：

```
void setPriority(int newDefaultPriority);
```

或者

```
void send(Destination destination, Message message, int deliveryMode, in
    t priority,
long timeToLive);
 void send(Message message, int deliveryMode, int priority, long timeToL
ive);
```

其中 deliveryMode 为传送模式，priority 为消息优先级，timeToLive 为消息过期时间。

例如：

```
producer.setPriority(4);
```

C++客户端：

函数原型：

```
void setPriority( int priority );
```

或者

```
void ActiveMQProducer::send( cms::Message* message, int deliveryMode,
    int priority,
    long long timeToLive )
        throw ( cms::CMSException );
void ActiveMQProducer::send( const cms::Destination* destination,
    cms::Message* message, int deliveryMode,
    int priority, long long timeToLive)
        throw ( cms::CMSException );
```

例如：

```
producer-> setPriority(4);
```

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。如果不指定优先级，则默认为 4。JMS 不要求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。

4.3.1.4 允许消息过期

默认情况下，消息永不会过期。如果消息在特定周期内失去意义，那么可以设置过期时间。

有两种方法设置消息的过期时间，时间单位为毫秒：

1. 使用 setTimeToLive 方法为所有的消息设置过期时间；
2. 使用 send 方法为每一条消息设置过期时间；

Java 客户端：

ActiveMQMessageProducer 方法：

```
void setTimeToLive(long timeToLive);
```

或者

```
void send(Destination destination, Message message, int deliveryMode, in
    t priority,
long timeToLive);
void send(Message message, int deliveryMode, int priority, long timeToL
ive);
```

其中 deliveryMode 为传送模式，priority 为消息优先级，timeToLive 为消息过期时间。

例如：

```
producer.setTimeToLive(1000);
```

C++客户端：

函数原型：

```
void setTimeToLive( long long time );
```

或者

```
void ActiveMQProducer::send( cms::Message* message, int deliveryMode,
    int priority,
    long long timeToLive )
```

```
        throw ( cms::CMSException );
void ActiveMQProducer::send( const cms::Destination* destination,
    cms::Message* message, int deliveryMode,
    int priority, long long timeToLive)
        throw ( cms::CMSException );
```

例如：

```
Producer->setTimeToLive(1000);
```

消息过期时间，send 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。

4.3.1.5 创建临时目标

ActiveMQ 通过 createTemporaryQueue 和 createTemporaryTopic 创建临时目标，这些目标持续到创建它的 Connection 关闭。只有创建临时目标的 Connection 所创建的客户端才可以从临时目标中接收消息，但是任何的生产者都可以向临时目标中发送消息。如果关闭了创建此目标的 Connection，那么临时目标被关闭，内容也将消失。

Java 客户端：

ActiveMQSession 方法：

```
TemporaryQueue createTemporaryQueue();
TemporaryTopic createTemporaryTopic();
```

C++客户端：

函数原型：

```
cms::TemporaryQueue* ActiveMQSession::createTemporaryQueue(void)
    throw ( cms::CMSException );
cms::TemporaryTopic* ActiveMQSession::createTemporaryTopic(void)
    throw ( cms::CMSException );
```

某些客户端需要一个目标来接收对发送至其他客户端的消息的回复。这时可以使用临时目标。Message 的属性之一是 JMSReplyTo 属性，这个属性就是用于这个目的的。可以创建一个临时的 Destination，并把它放入 Message 的 JMSReplyTo 属性中，收到该消息的消费者可以用它来响应生产者。

Java 客户端：

如下所示代码段，将创建临时的 Destination，并将它放置在 TextMessage 的 JMSReplyTo 属性中：

```
// Create a temporary queue for replies...

Destination tempQueue = session.createTemporaryQueue();

// Set ReplyTo to temporary queue...

msg.setJMSReplyTo(tempQueue);
```

消费者接收这条消息时，会从 JMSReplyTo 字段中提取临时 Destination，并且会通过应用程序构造一个 MessageProducer，以便将响应消息发送回生产者。这展示了如何

使用 JMS Message 的属性，并显示了私有的临时 Destination 的有用之处。它还展示了客户端可以既是消息的生产者，又可以是消息的消费者。

```
// Get the temporary queue from the JMSReplyTo

// property of the message...

Destination tempQueue = msg.getJMSReplyTo();

...

// create a Sender for the temporary queue

MessageProducer Sender = session. createProducer(tempQueue);

TextMessage msg = session.createTextMessage();

msg.setText(REPLYTO_TEXT);

...

// Send the message to the temporary queue...

sender.send(msg);
```

4.3.2 高级可靠性机制

4.3.2.1 创建持久订阅

通过为发布者设置 PERSISTENT 传送模式，为订阅者时使用持久订阅，这样可以保证 Pub/Sub 程序接收所有发布的消息。

消息订阅分为非持久订阅(non-durable subscription)和持久订阅(durable subscription)，非持久订阅只有当客户端处于激活状态，也就是和 ActiveMQ 保持连接状态才能收到发送到某个主题的消息，而当客户端处于离线状态，这个时间段发到主题的消息将会丢失，永远不会收到。持久订阅时，客户端向 ActiveMQ 注册一个识别自己身份的 ID，当这个客户端处于离线时，ActiveMQ 会为这个 ID 保存所有发送到主题的消息，当客户端再次连接到 ActiveMQ 时,会根据自己的 ID 得到所有当自己处于离线时发送到主题的消息。持久订阅会增加开销，同一时间在持久订阅中只有一个激活的用户。

建立持久订阅的步骤：

1. 为连接设置一个客户 ID；
2. 为订阅的主题指定一个订阅名称；
   上述组合必须唯一。

4.3.2.1.1 创建持久订阅

Java 客户端：

ActiveMQConnection 方法：

void setClientID(String newClientID)

和

ActiveMQSession 方法：

```
TopicSubscriber createDurableSubscriber(Topic topic, String name)
TopicSubscriber createDurableSubscriber(Topic topic, String name, Strin
g
    messageSelector, boolean noLocal)
```

其中 messageSelector 为消息选择器；noLocal 标志默认为 false，当设置为 true 时限制消费者只能接收和自己相同的连接（Connection）所发布的消息，此标志只适用于主题，不适用于队列；name 标识订阅主题所对应的订阅名称，持久订阅时需要设置此参数。

C++客户端：

函数原型：

```
virtual void setClientId( const std::string& clientId );
```

和

```
cms::MessageConsumer* ActiveMQSession::createDurableConsumer(
        const cms::Topic* destination,
        const std::string& name,
        const std::string& selector,
        bool noLocal )
        throw ( cms::CMSException )
```

4.3.2.1.2 删除持久订阅

Java 客户端：

ActiveMQSession 方法：

```
void unsubscribe(String name);
```

4.3.2.2 使用本地事务

在事务中生成或使用消息时，ActiveMQ 跟踪各个发送和接收过程，并在客户端发出提交事务的调用时完成这些操作。如果事务中特定的发送或接收操作失败，则出现异常。客户端代码通过忽略异常、重试操作或回滚整个事务来处理异常。在事务提交时，将完成所有成功的操作。在事务进行回滚时，将取消所有成功的操作。

本地事务的范围始终为一个会话。也就是说，可以将单个会话的上下文中执行的一个或多个生产者或消费者操作组成一个本地事务。

不但单个会话可以访问 Queue 或 Topic （任一类型的 Destination ），而且单个会话实例可以用来操纵一个或多个队列以及一个或多个主题，一切都在单个事务中进行。这意味着单个会话可以（例如）创建队列和主题中的生产者，然后使用单个事务来同时发送队列和主题中的消息。因为单个事务跨越两个目标，所以，要么队列和主题的消息都得到发送，要么都未得到发送。类似地，单个事务可以用来接收队列中的消息并将消息发送到主题上，反过来也可以。

由于事务的范围只能为单个的会话，因此不存在既包括消息生成又包括消息使用的端对端事务。（换句话说，至目标的消息传送和随后进行的至客户端的消息传送不能放在同一个事务中。）

4.3.2.2.1 使用事务

Java 客户端：

ActiveMQConnection 方法：

Session createSession(boolean transacted, int acknowledgeMode);

其中 transacted 为使用事务标识，acknowledgeMode 为签收模式。

例如：

```
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

C++客户端：

函数原型：

```
cms::Session* ActiveMQConnection::createSession(
   cms::Session::AcknowledgeMode ackMode );
```

其中 AcknowledgeMode ackMode 需指定为 SESSION_TRANSACTED。

例如：

```
Session* session = connection->createSession( Session:: SESSION_TRANSACTED );
```

4.3.2.2.2 提交

Java 客户端：

ActiveMQSession 方法：

void commit()；

例如：

```
try {
 producer.send(consumer.receive());
 session.commit();
}
catch (JMSException ex) {
 session.rollback();
}
```

C++客户端：

函数原型：

```
void ActiveMQSession::commit(void) throw ( cms::CMSException )
```

4.3.2.2.3 回滚

Java 客户端：

ActiveMQSession 方法：

void rollback();

C++客户端：

函数原型：

```
void ActiveMQSession::rollback(void) throw ( cms::CMSException )
```

4.4 高级特征

4.4.1 异步发送消息

ActiveMQ 支持生产者以同步或异步模式发送消息。使用不同的模式对 send 方法的反应时间有巨大的影响，反映时间是衡量 ActiveMQ 吞吐量的重要因素，使用异步发送可以提高系统的性能。

在默认大多数情况下，AcitveMQ 是以异步模式发送消息。例外的情况：在没有使用事务的情况下，生产者以 PERSISTENT 传送模式发送消息。在这种情况下，send 方法都是同步的，并且一直阻塞直到 ActiveMQ 发回确认消息：消息已经存储在持久性数据存储中。这种确认机制保证消息不会丢失，但会造成生产者阻塞从而影响反应时间。

高性能的程序一般都能容忍在故障情况下丢失少量数据。如果编写这样的程序，可以通过使用异步发送来提高吞吐量（甚至在使用 PERSISTENT 传送模式的情况下）。

Java 客户端：

使用 Connection URI 配置异步发送:

```
cf = new ActiveMQConnectionFactory("tcp://locahost:61616?jms.useAsyncSend=true");
```

在 ConnectionFactory 层面配置异步发送:

```
((ActiveMQConnectionFactory)connectionFactory).setUseAsyncSend(true);
```

在 Connection 层面配置异步发送，此层面的设置将覆盖 ConnectionFactory 层面的设置:

```
((ActiveMQConnection)connection).setUseAsyncSend(true);
```

4.4.2 消费者特色

4.4.2.1 消费者异步分派

在 ActiveMQ4 中，支持 ActiveMQ 以同步或异步模式向消费者分派消息。这样的意义：可以以异步模式向处理消息慢的消费者分配消息；以同步模式向处理消息快的消费者分配消息。

ActiveMQ 默认以同步模式分派消息，这样的设置可以提高性能。但是对于处理消息慢的消费者，需要以异步模式分派。

Java 客户端：

在 ConnectionFactory 层面配置同步分派:

```
((ActiveMQConnectionFactory)connectionFactory).setDispatchAsync(false);
```

在 Connection 层面配置同步分派，此层面的设置将覆盖 ConnectionFactory 层面的设置：

```
((ActiveMQConnection)connection).setDispatchAsync(false);
```

在消费者层面以 Destination URI 配置同步分派，此层面的设置将覆盖 ConnectionFactory 和 Connection 层面的设置：

```
queue = new ActiveMQQueue("TEST.QUEUE?consumer.dispatchAsync=false");

consumer = session.createConsumer(queue);
```

4.4.2.2 消费者优先级

在 ActveMQ 分布式环境中，在有消费者存在的情况下，如果更希望 ActveMQ 发送消息给消费者而不是其他的 ActveMQ 到 ActveMQ 的传送，可以如下设置：

Java 客户端：

```
queue = new ActiveMQQueue("TEST.QUEUE?consumer.prority=10");

consumer = session.createConsumer(queue);
```

4.4.2.3 独占的消费者

ActiveMQ 维护队列消息的顺序并顺序把消息分派给消费者。但是如果建立了多个 Session 和 MessageConsumer，那么同一时刻多个线程同时从一个队列中接收消息时就并不能保证处理时有序。

有时候有序处理消息是非常重要的。ActiveMQ4 支持独占的消费。ActiveMQ 挑选一个 MessageConsumer，并把一个队列中所有消息按顺序分派给它。如果消费者发生故障，那么 ActiveMQ 将自动故障转移并选择另一个消费者。可以如下设置：

Java 客户端：

```
queue = new ActiveMQQueue("TEST.QUEUE?consumer.exclusive=true");

consumer = session.createConsumer(queue);
```

4.4.2.4 再次传送策略

在以下三种情况中，消息会被再次传送给消费者：

1. 在使用事务的 Session 中，调用 rollback()方法；
2. 在使用事务的 Session 中，调用 commit()方法之前就关闭了 Session;
3. 在 Session 中使用 CLIENT_ACKNOWLEDGE 签收模式，并且调用了 recover()方法。

可以通过设置 ActiveMQConnectionFactory 和 ActiveMQConnection 来定制想要的再次传送策略。

属性	默认值	描述
collisionAvoidanceFactor	0.15	The percentage of range of collision avoidance if enabled
maximumRedeliveries	6	Sets the maximum number of times a message will be redelivered before it is considered a **poisoned pill** and

		returned to the broker so it can go to a Dead Letter Queue
initialRedeliveryDelay	1000L	The initial redelivery delay in milliseconds
useCollisionAvoidance	false	Should the redelivery policy use collision avoidance
useExponentialBackOff	false	Should exponential back-off be used (i.e. to exponentially increase the timeout)
backOffMultiplier	5	The back-off multiplier

4.4.3 目标特色

4.4.3.1 复合目标

在 1.1 版本之后，ActiveMQ 支持混合目标技术。它允许在一个 JMS 目标中使用一组 JMS 目标。

例如可以利用混合目标在同一操作中用向 12 个队列发送同一条消息或者在同一操作中向一个主题和一个队列发送同一条消息。

在混合目标中，通过“ , ”来分隔不同的目标。

Java 客户端：

例如：

```
// send to 3 queues as one logical operation

Queue queue = new ActiveMQQueue("FOO.A,FOO.B,FOO.C");

producer.send(queue, someMessage);
```

如果在一个目标中混合不同类别的目标，可以通过使用“ queue:// ”和“ topic:// ”前缀来识别不同的目标。

例如：

```
// send to queues and topic one logical operation

Queue queue = new ActiveMQQueue("FOO.A,topic://NOTIFY.FOO.A");

producer.send(queue, someMessage);
```

4.4.3.2 目标选项

属性	默认值	描述
consumer.prefetchSize	variable	The number of message the consumer will prefetch.
consumer.maximumPendingMessageLimit	0	Use to control if messages are dropped if a slow consumer situation exists.

consumer.noLocal	false	Same as the noLocal flag on a Topic consumer. Exposed here so that it can be used with a queue.
consumer.dispatchAsync	false	Should the broker dispatch messages asynchronously to the consumer.
consumer.retroactive	false	Is this a Retroactive Consumer.
consumer.selector	null	JMS Selector used with the consumer.
consumer.exclusive	false	Is this an Exclusive Consumer.
consumer.priority	0	Allows you to configure a Consumer Priority.

Java 客户端：

例如：

```
    queue = new ActiveMQQueue("TEST.QUEUE?consumer.dispatchAsync=false&consumer.prefetchSize=10");

    consumer = session.createConsumer(queue);
```

4.4.4 消息预取

ActiveMQ 的目标之一就是高性能的数据传送，所以 ActiveMQ 使用“ 预取限制 ”来控制有多少消息能及时的传送给任何地方的消费者。

一旦预取数量达到限制，那么就不会有消息被分派给这个消费者直到它发回签收消息（用来标识所有的消息已经被处理）。

可以为每个消费者指定消息预取。如果有大量的消息并且希望更高的性能，那么可以为这个消费者增大预取值。如果有少量的消息并且每条消息的处理都要花费很长的时间，那么可以设置预取值为 1，这样同一时间，ActiveMQ 只会为这个消费者分派一条消息。

Java 客户端：

在 ConnectionFactory 层面为所有消费者配置预取值：

```
    tcp://localhost:61616?jms.prefetchPolicy.all=50
```

在 ConnectionFactory 层面为队列消费者配置预取值：

```
    tcp://localhost:61616?jms.prefetchPolicy.queuePrefetch=1
```

使用“ 目标选项 ”为一个消费者配置预取值：

```
    queue = new ActiveMQQueue("TEST.QUEUE?consumer.prefetchSize=10");

    consumer = session.createConsumer(queue);
```

4.4.5 配置连接 URL

ActiveMQ 支持通过 Configuration URI 明确的配置连接属性。

例如：当要设置异步发送时，可以通过在 Configuration URI 中使用 jms.$PROPERTY 来设置。

```
tcp://localhost:61616?jms.useAsyncSend=true
```

以下的选项在 URI 必须以“ **jms.** ”为前缀。

属性	默认值	描述
alwaysSessionAsync	true	If this flag is set then a seperate thread is not used for dispatching messages for each Session in the Connection. However, a separate thread is always used if there is more than one session, or the session isn't in auto acknowledge or dups ok mode
clientID	null	Sets the JMS clientID to use for the connection
closeTimeout	15000 (milliseconds)	Sets the timeout before a close is considered complete. Normally a close() on a connection waits for confirmation from the broker; this allows that operation to timeout to save the client hanging if there is no broker.
copyMessageOnSend	true	Should a JMS message be copied to a new JMS Message object as part of the send() method in JMS. This is enabled by default to be compliant with the JMS specification. You can disable it if you do not mutate JMS messages after they are sent for a performance boost.
disableTimeStampsByDefault	false	Sets whether or not

		timestamps on messages should be disabled or not. If you disable them it adds a small performance boost.
dispatchAsync	false	Should the broker dispatch messages asynchronously to the consumer.
nestedMapAndListEnabled	true	Enables/disables whether or not Structured Message Properties and MapMessages are supported so that Message properties and MapMessage entries can contain nested Map and List objects. Available since version 4.1 onwards
objectMessageSerializationDefered	false	When an object is set on an ObjectMessage, the JMS spec requires the object to be serialized by that set method. Enabling this flag causes the object to not get serialized. The object may subsequently get serialized if the message needs to be sent over a socket or stored to disk.
optimizeAcknowledge	false	Enables an optimised acknowledgement mode where messages are acknowledged in batches rather than individually. Alternatively, you could use Session.DUPS_OK_ACKNOWLEDGE acknowledgement mode for the consumers which can often be faster. **WARNING** enabling this issue could cause some issues with auto-acknowledgement on

		reconnection
optimizedMessageDispatch	true	If this flag is set then an larger prefetch limit is used - only applicable for durable topic subscribers
useAsyncSend	false	Forces the use of Async Sends which adds a massive performance boost; but means that the send() method will return immediately whether the message has been sent or not which could lead to message loss.
useCompression	false	Enables the use of compression of the message bodies
useRetroactiveConsumer	false	Sets whether or not retroactive consumers are enabled. Retroactive consumers allow non-durable topic subscribers to receive old messages that were published before the non-durable subscriber started.

4.5 优化

优化部分请参阅：http://devzone.logicblaze.com/site/how-to-tune-activemq.html

5. ActiveMQ 配置

5.1 配置文件

ActiveMQ 配置文件：$AcrtiveMQ/conf/activemq.xml

5.2 配置 ActiveMQ 服务 IP 和端口

```
        <transportConnectors>
           <transportConnector name="openwire" uri="tcp://localhost:61616" discoveryUri="mu
lticast://default"/>
           <transportConnector name="ssl"     uri="ssl://localhost:61617"/>
           <transportConnector name="stomp"   uri="stomp://localhost:61613"/>
        </transportConnectors>
```

在 transportConnectors 标识中配置 ActiveMQ 服务 IP 和端口，其中 name 属性指定协议的名称，uri 属性指定协议所对应的协议名，IP 地址和端口号。上述 IP 地址和端口可以根据实际需要指定。Java 客户端默认使用 openwire 协议，所以 ActiveMQ 服务地址为 tcp://localhost:61616；目前 C++客户端仅支持 stomp 协议，所以 ActiveMQ 服务地址为 tcp://localhost:61613。

5.3 分布式部署

分布式部署请参阅：http://activemq.apache.org/networks-of-brokers.html

5.4 监控 ActiveMQ

本节将使用 JXM 和 JXM 控制台（JDK1.5 控制台）监控 ActiveMQ。

5.4.1 配置 JXM

```
    <broker brokerName="emv219" useJmx="true" xmlns="http://activemq.org/config/1.0">
    ...
    <managementContext>
        <managementContext connectorPort="1099" jmxDomainName="org.apache.activemq"/>
    </managementContext>
    ...
    </broker>
```

配置 JXM 步骤如下：

1. 设置 broker 标识的 useJmx 属性为 true；
2. 取消对 managementContext 标识的注释（系统默认注释 managementContext 标识），监控的默认端口为 1099。

5.4.2 在 Windows 平台监控

进入%JAVA_HOME%/bin，双击 jconsole.exe 即出现如下画面，在对话框中输入 ActiveMQ 服务主机的地址，JXM 的端口和主机登陆帐号。

6. 目前存在问题

6.1 C++客户端丢失消息问题

ActiveMQ 版本：ActiveMQ 4.1.1SNAPSHOT
C++客户端版本：ActiveMQ CPP 1.1 Release
测试中发现，当 C++客户端异常退出时（即没有正常调用 close 函数关闭连接），ActiveMQ 并不能检测到 C++客户端的连接已经中断，这时如果向队列中发送消息，那么第一条消息就会丢失，这时 ActiveMQ 才能检测到这个连接是中断的。
在 ActiveMQ 论坛反应此问题后，开发人员答复并建议使用 CLIENT_ACKNOWLEDGE 签收模式。但是此模式会造成消息重复接收。
测试 ActiveMQ 4.2SNAPSHOT 时并未发现上述问题。

6.2 队列消息堆积过多后有可能阻塞程序

默认 activemq.xml 中配置的内存是 20M，这就意味着当消息堆积超过 20M 后，程序可能出现问题。在 mial list 中其他用户对此问题的描述是：send 方法会阻塞或抛出异常。ActiveMQ 开发人员的答复：The memory model is different for ActiveMQ 4.1 in that for Queues, only small references to the Queue messages are held in memory. This means that the Queue depth can be considerably bigger than for ActiveMQ 3.2.x.However, our next major release (5.0 nee 4.2) has a more robust model in that Queue messages are paged in from storage only when space is available - hence Queue depth is now limited by how much disk space you have.

6.3 目前版本的 C++客户端仅支持 stomp 协议

目前版本的 C++客户端程序（ActiveMQ CPP 1.1 Release）仅支持 stomp 协议，因此传输消息的速度应该没有使用 openwire 协议的 Java 客户端快。ActiveMQ 网站显示不久将会有支持 openwire 协议的 C++客户端程序发布。

6.4 分布式部署问题

ActiveMQ 版本：ActiveMQ 4.1.1SNAPSHOT 和 ActiveMQ 4.2SNAPSHOT

测试选用上述两个未正式发布的版本，未选用正式发布的 ActiveMQ 4.1.0 Release 版本是因为此版本 bug 较多。

在测试中发现，如果重启其中一台机器上的 ActiveMQ，其他机器的 ActiveMQ 有可能会打印：

```
        java.io.EOFException
            at java.io.DataInputStream.readInt(DataInputStream.java:358)
            at org.apache.activemq.openwire.OpenWireFormat.unmarshal(OpenWireFormat.java:267)
            at org.apache.activemq.transport.tcp.TcpTransport.readCommand(TcpTransport.java:156)
            at org.apache.activemq.transport.tcp.TcpTransport.run(TcpTransport.java:136)
            at java.lang.Thread.run(Thread.java:595)
    WARN  TransportConnection            - Unexpected extra broker info command received: BrokerInfo {commandId = 6, responseRequired = false, brokerId = ID:emv219n-33945-1174458770157-1:0, brokerURL = tcp://emv219n:61616, slaveBroker = false, masterBroker = false, faultTolerantConfiguration = false, networkConnection = false, duplexConnection = false, peerBrokerInfos = [], brokerName = emv219, connectionId = 0}.
    INFO  FailoverTransport              - Transport failed, attempting to automatically reconnect due to: java.io.EOFException。
```

这时分布式的消息传输就会出现问题，此问题目前还没找到原因。

7. 附录

7.1 完整的 Java 客户端例子

```
    import org.apache.activemq.ActiveMQConnectionFactory;


    import javax.jms.Connection;
    import javax.jms.DeliveryMode;
    import javax.jms.Destination;
    import javax.jms.ExceptionListener;
    import javax.jms.JMSException;
    import javax.jms.Message;
    import javax.jms.MessageConsumer;
    import javax.jms.MessageProducer;
    import javax.jms.Session;
    import javax.jms.TextMessage;
```

```
/**
 * Hello world!
 */
public class App {

    public static void main(String[] args) throws Exception {
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        Thread.sleep(1000);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        Thread.sleep(1000);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldProducer(), false);
        Thread.sleep(1000);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldConsumer(), false);
        thread(new HelloWorldProducer(), false);
    }
```

```
        public static void thread(Runnable runnable, boolean daemon) {
            Thread brokerThread = new Thread(runnable);
            brokerThread.setDaemon(daemon);
            brokerThread.start();
        }

        public static class HelloWorldProducer implements Runnable {
            public void run() {
                try {
                    // Create a ConnectionFactory
                    ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory
("tcp://localhost:61616");

                    // Create a Connection
                    Connection connection = connectionFactory.createConnection();
                    connection.start();

                    // Create a Session
                    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDG
E);

                    // Create the destination (Topic or Queue)
                    Destination destination = session.createQueue("TEST.FOO");

                    // Create a MessageProducer from the Session to the Topic or Queue
                    MessageProducer producer = session.createProducer(destination);
                    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

                    // Create a messages
                    String text = "Hello world! From: " + Thread.currentThread().getName() + " :
 " + this.hashCode();
                    TextMessage message = session.createTextMessage(text);

                    // Tell the producer to send the message
                    System.out.println("Sent message: "+ message.hashCode() + " : " + Thread.cu
rrentThread().getName());
                    producer.send(message);
```

```
                    // Clean up
                    session.close();
                    connection.close();
                }
                catch (Exception e) {
                    System.out.println("Caught: " + e);
                    e.printStackTrace();
                }
            }
        }

        public static class HelloWorldConsumer implements Runnable, ExceptionListener {
            public void run() {
                try {

                    // Create a ConnectionFactory
                    ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory
("tcp://localhost:61616");

                    // Create a Connection
                    Connection connection = connectionFactory.createConnection();
                    connection.start();

                    connection.setExceptionListener(this);

                    // Create a Session
                    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDG
E);

                    // Create the destination (Topic or Queue)
                    Destination destination = session.createQueue("TEST.FOO");

                    // Create a MessageConsumer from the Session to the Topic or Queue
                    MessageConsumer consumer = session.createConsumer(destination);
```

```
                // Wait for a message
                Message message = consumer.receive(1000);

                if (message instanceof TextMessage) {
                    TextMessage textMessage = (TextMessage) message;
                    String text = textMessage.getText();
                    System.out.println("Received: " + text);
                } else {
                    System.out.println("Received: " + message);
                }

                consumer.close();
                session.close();
                connection.close();
            } catch (Exception e) {
                System.out.println("Caught: " + e);
                e.printStackTrace();
            }
        }

        public synchronized void onException(JMSException ex) {
            System.out.println("JMS Exception occured.  Shutting down client.");
        }
    }
}
```

7.2 完整的 C++客户端例子

```
#include <activemq/concurrent/Thread.h>
#include <activemq/concurrent/Runnable.h>
#include <activemq/core/ActiveMQConnectionFactory.h>
#include <activemq/util/Integer.h>
#include <cms/Connection.h>
#include <cms/Session.h>
#include <cms/TextMessage.h>
#include <cms/ExceptionListener.h>
```

```
#include <cms/MessageListener.h>
#include <stdlib.h>
#include <iostream>

using namespace activemq::core;
using namespace activemq::util;
using namespace activemq::concurrent;
using namespace cms;
using namespace std;

class HelloWorldProducer : public Runnable {
private:

    Connection* connection;
    Session* session;
    Destination* destination;
    MessageProducer* producer;
    int numMessages;
    bool useTopic;

public:

    HelloWorldProducer( int numMessages, bool useTopic = false ){
        connection = NULL;
        session = NULL;
        destination = NULL;
        producer = NULL;
        this->numMessages = numMessages;
        this->useTopic = useTopic;
    }

    virtual ~HelloWorldProducer(){
        cleanup();
    }

    virtual void run() {
```

```
            try {
                // Create a ConnectionFactory
                ActiveMQConnectionFactory* connectionFactory = new ActiveMQConnectionFactory("t
cp://127.0.0.1:61613");

                // Create a Connection
                connection = connectionFactory->createConnection();
                connection->start();

                // Create a Session
                session = connection->createSession( Session::AUTO_ACKNOWLEDGE );

                // Create the destination (Topic or Queue)
                if( useTopic ) {
                    destination = session->createTopic( "TEST.FOO" );
                } else {
                    destination = session->createQueue( "TEST.FOO" );
                }

                // Create a MessageProducer from the Session to the Topic or Queue
                producer = session->createProducer( destination );
                producer->setDeliveryMode( DeliveryMode::NON_PERSISTANT );

                // Create the Thread Id String
                string threadIdStr = Integer::toString( Thread::getId() );

                // Create a messages
                string text = (string)"Hello world! from thread " + threadIdStr;

                for( int ix=0; ix<numMessages; ++ix ){
                    TextMessage* message = session->createTextMessage( text );

                    // Tell the producer to send the message
                    printf( "Sent message from thread %s\n", threadIdStr.c_str() );
                    producer->send( message );
```

```
                delete message;
            }

        }catch ( CMSException& e ) {
            e.printStackTrace();
        }
    }

private:

    void cleanup(){

            // Destroy resources.
            try{
                if( destination != NULL ) delete destination;
            }catch ( CMSException& e ) {}
            destination = NULL;

            try{
                if( producer != NULL ) delete producer;
            }catch ( CMSException& e ) {}
            producer = NULL;

            // Close open resources.
            try{
                if( session != NULL ) session->close();
                if( connection != NULL ) connection->close();
            }catch ( CMSException& e ) {}

            try{
                if( session != NULL ) delete session;
            }catch ( CMSException& e ) {}
            session = NULL;

            try{
                if( connection != NULL ) delete connection;
```

```
            }catch ( CMSException& e ) {}
            connection = NULL;
        }
    };

    class HelloWorldConsumer : public ExceptionListener,
                               public MessageListener,
                               public Runnable {

    private:

        Connection* connection;
        Session* session;
        Destination* destination;
        MessageConsumer* consumer;
        long waitMillis;
        bool useTopic;

    public:

        HelloWorldConsumer( long waitMillis, bool useTopic = false ){
            connection = NULL;
            session = NULL;
            destination = NULL;
            consumer = NULL;
            this->waitMillis = waitMillis;
            this->useTopic = useTopic;
        }
        virtual ~HelloWorldConsumer(){
            cleanup();
        }

        virtual void run() {

            try {
```

```
            // Create a ConnectionFactory
            ActiveMQConnectionFactory* connectionFactory =
                new ActiveMQConnectionFactory( "tcp://127.0.0.1:61613" );

            // Create a Connection
            connection = connectionFactory->createConnection();
            delete connectionFactory;
            connection->start();

            connection->setExceptionListener(this);

            // Create a Session
            session = connection->createSession( Session::AUTO_ACKNOWLEDGE );

            // Create the destination (Topic or Queue)
            if( useTopic ) {
                destination = session->createTopic( "TEST.FOO" );
            } else {
                destination = session->createQueue( "TEST.FOO" );
            }

            // Create a MessageConsumer from the Session to the Topic or Queue
            consumer = session->createConsumer( destination );

            consumer->setMessageListener( this );

            // Sleep while asynchronous messages come in.
            Thread::sleep( waitMillis );

        } catch (CMSException& e) {
            e.printStackTrace();
        }
    }

    // Called from the consumer since this class is a registered MessageListener.
    virtual void onMessage( const Message* message ){
```

```
        static int count = 0;

        try
        {
            count++;
            const TextMessage* textMessage =
                dynamic_cast< const TextMessage* >( message );
            string text = textMessage->getText();
            printf( "Message #%d Received: %s\n", count, text.c_str() );
        } catch (CMSException& e) {
            e.printStackTrace();
        }
    }

    // If something bad happens you see it here as this class is also been
    // registered as an ExceptionListener with the connection.
    virtual void onException( const CMSException& ex ) {
        printf("JMS Exception occured.  Shutting down client.\n");
    }

private:

    void cleanup(){

        //**************************************************
        // Always close destination, consumers and producers before
        // you destroy their sessions and connection.
        //**************************************************

        // Destroy resources.
        try{
            if( destination != NULL ) delete destination;
        }catch (CMSException& e) {}
        destination = NULL;
```

```
            try{
                if( consumer != NULL ) delete consumer;
            }catch (CMSException& e) {}
            consumer = NULL;

            // Close open resources.
            try{
                if( session != NULL ) session->close();
                if( connection != NULL ) connection->close();
            }catch (CMSException& e) {}

            // Now Destroy them
            try{
                if( session != NULL ) delete session;
            }catch (CMSException& e) {}
            session = NULL;

            try{
                if( connection != NULL ) delete connection;
            }catch (CMSException& e) {}
            connection = NULL;
        }
    };

    int main(int argc, char* argv[]) {

        std::cout << "=====================================================\n";
        std::cout << "Starting the example:" << std::endl;
        std::cout << "-----------------------------------------------------\n";

        //============================================================
        // set to true to use topics instead of queues
        // Note in the code above that this causes createTopic or
        // createQueue to be used in both consumer an producer.
        //============================================================
        bool useTopics = false;
```

```
    HelloWorldProducer producer( 1000, useTopics );
    HelloWorldConsumer consumer( 8000, useTopics );

    // Start the consumer thread.
    Thread consumerThread( &consumer );
    consumerThread.start();

    // Start the producer thread.
    Thread producerThread( &producer );
    producerThread.start();

    // Wait for the threads to complete.
    producerThread.join();
    consumerThread.join();

    std::cout << "-----------------------------------------------------\n";
    std::cout << "Finished with the example, ignore errors from this"
              << std::endl
              << "point on as the sockets breaks when we shutdown."
              << std::endl;
    std::cout << "=====================================================\n";
}
```