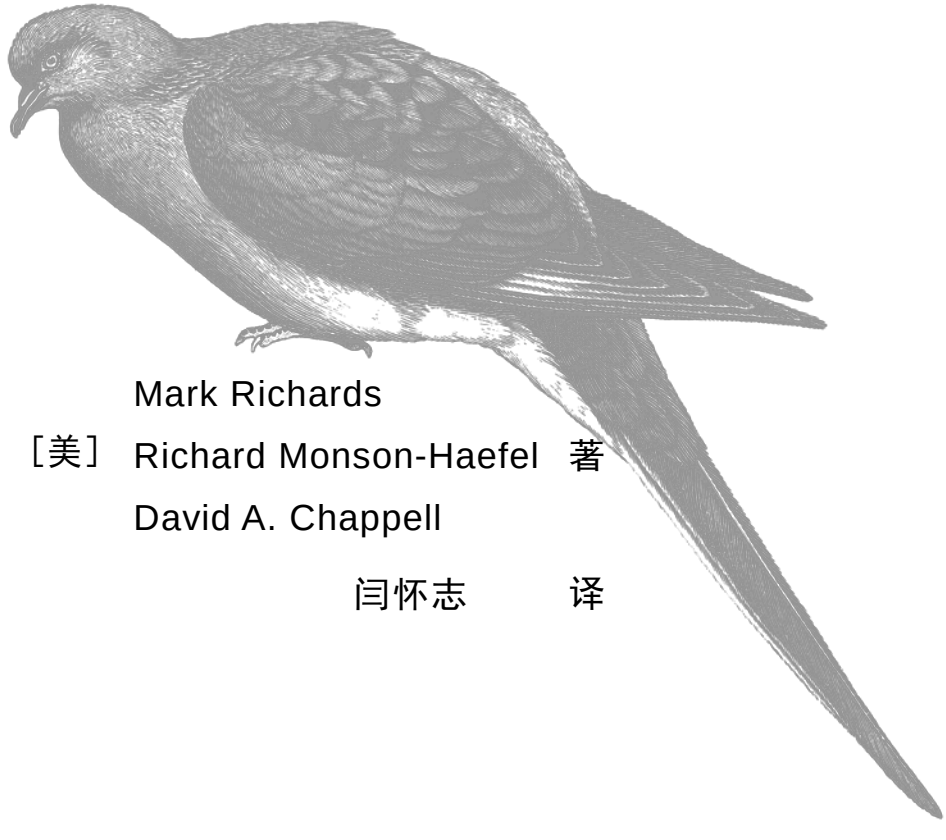
Java 消息服务（第 2 版）

Java Message Service,2nd Edition



Mark Richards

［美］ Richard Monson-Haefel 著

David A. Chappell

闫怀志 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书深入浅出地讲解了 JMS1.1 规范的底层技术、Java 类和接口、编程模型及其不同实现等 Java 消息服务（JMS）和消息传送机制关键技术。通过对支持点对点和发布/订阅“消息传送”的标准 API 的完全解读及具体实例，介绍了如何利用“厂商无关”的 JMS 来解决许多体系结构面临的挑战。本书适用于掌握 Java 语言并有业务解决方案开发经验的读者，或者需要学习消息传送技术的读者。

版权贸易合同登记号　图字：01-2009-3823

图书在版编目（CIP）数据

Java 消息服务：第 2 版 /（美）理查兹（Richards,M.），（美）蒙森-哈斐尔（Monson-Haefel,R.），（美）查普尔（Chappell,D.A.）著；闫怀志译.—北京：电子工业出版社，2010.1

书名原文: Java Message Service，2/e

ISBN 978-7-121-10050-5

Ⅰ. J…　Ⅱ.①理…②蒙…③查…④闫…　Ⅲ.JAVA 语言－程序设计　Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2009）第 224822 号

责任编辑：周筠

封面设计：Karen Montgomery，张健

印　　刷：北京市天竺颖华印刷厂

装　　订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

　　北京市海淀区万寿路 173 信箱　邮编 100036

开　　本：787×980　1/16　印张：21　字数：400 千字

印　　次：2010 年 1 月第 1 次印刷

定　　价：59.80 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 *zlts@phei.com.cn*，盗版侵权举报请发邮件至 *dbqq@phei.com.cn*。服务热线：（010）88258888。

译者序

获邀担纲《Java 消息服务（第 2 版 》的翻译，我的思绪一下回到了 10 年前。当时，不仅是初出茅庐的我，甚至很多资深的同行，都在为不同系统之间的信息交换而头疼不已：一方面，数据集成和系统整合的需求非常旺盛；另一方面，信息交换的技术实现不仅难度很大，而且实现方式也因各有利弊而难于选择。在技术路线的选择上，我们究竟何去何从？这在“雾里看花”的当年，真不啻于一场赌博。因为后来的事实证明，即便是其中较为常用的 CORBA、DCOM、RMI 等远程过程调用（RPC）中间件技术，虽可以解决一些问题，但它们在健壮性、性能和可伸缩性方面的表现很难让人满意。

期间，从我一位留美的同学处，偶然获悉了 Java 消息服务（JMS）机制，令我闻之一震——它很好地解决了让我困惑良久的在不同系统间进行数据和功能共享的问题。我不仅自己如获至宝，还因为职业的关系，在我的领地——三尺讲台上竭力传播这一思路。

简而言之，JMS 是 1999 年由 Sun Microsystems 领衔开发的一种访问消息系统的方法，也就是供 Java 程序员使用的面向消息的中间件（MOM）。这种基于消息传送的异步处理模型，具有非阻塞的调用特性。发送者将消息发送给消息服务器，服务器会在合适的时候再将消息转发给接收者；发送和接收采用异步方式，这就意味着发送者无须等待，发送者和接收者的生命周期也无需相同，而且发送者还可以将消息传给多个接收者。如此一来，这种异步处理方式就大大提升了应用程序的健壮性、性能和可伸缩性，使数据集成和系统整合工作变得易如反掌，特别是在分布式应用上让同步处理方式望尘莫及。Java 消息服务作为一个与具体平台无关的 API，已经得到了绝大多数 MOM 提供商的支持。

本书深入浅出地讲解了 JMS 规范的底层技术、Java 类和接口、编程模型及其不同实现。尤其值得指出的是，当年我学习 Java 消息服务的启蒙教材正是 O’Reilly 在 2000 年出版的本书第 1 版。所以看到第 2 版，真是一见如故。较之 9 年前的第 1 版，除了著名的 Richard Monson-Haefel 和 David A. Chappell 之外，又有消息传送机制、面向服务体系结构和事务管理领域的公认权威 Mark Richards 来担纲第一作者。我在仔细研读之后发现，本版继承了初版的优点，但在内容及结构方面有了很大变化，某些地方甚至可以说是“质”变，究其原因主要有二：其一，初版两年后（即 2002 年 3 月）发布了 JMS 1.1，统一了消息域，新版必须为适用 JMS 1.1 而修订；其二，经过 10 年的发展，Java 平台和消息传送机制日臻完善，消息驱动 bean、Spring 消息框架、事件驱动体系结构（EDA）、面向服务体系结构（SOA）、RESTful JMS 接口，以及企业服务总线（ESB）等新的消息传送技术也层出不穷。这些亮点不仅是第 2 版的精华所在，它们本身也代表了 Java 消息服务技术的发展方向。

当然，任何一项技术都是有利有弊的，也都会经历发展、鼎盛直至消亡的生命旅程，Java 消息服务也不例外。毋庸讳言，近些年来在异构系统集成领域，Java 消息服务就遭到了 Web 服务的强力挑战。不过，在 Web 服务解决可靠性等问题之前，Java 消息服务仍然是异构系统集成的不二之选。我相信，如果您读完了本书，并切实领会了 JMS 的精髓和要义之后，会十分认同这个观点。

翻译一本书，弄不好就会是“出力不讨好的苦差”，因为无论是技术本身、原著水平、读者领悟力等哪个环节出了问题，板子通常都会打在翻译者身上。当然就本书而言，JMS 技术本身、原著的水平和读者的领悟力都不会有什么问题。所以，既然我愿意做这样的知识传播者，那就要勇于承担所有的责任。套用我所敬仰的一位长者之言：我只希望在我翻译完以后，全体读者能说一句，他是一个负责的译者，不是敷衍了事的译者，我就很满意了。如果他们再慷慨一点，说闫怀志某些地方译得还不错，我就谢天谢地了。

本书翻译期间得到了电子工业出版社博文视点公司徐定翔、白爱萍、杨绣国、陈元玉、许莹等编辑的热心帮助。书中绝大部分术语采用了 Sun 公司的标准译法，在此一并致谢。最后，我要感谢家人的支持，否则完成这样一项工作几乎是不可想象的。由于译者水平有限，加之时间较紧，虽已尽力避免错误，难免仍有疏漏，恳请广大读者将意见和建议发至：*bityhz001@sina.com*，不胜感激。

闫怀志

2009 年秋于北京中关村

目录

Contents

推荐序
Foreword

近十年来，我逐渐变成了基于消息传送系统的铁杆粉丝。这种系统所提供的可靠性、灵活性、扩展性及模块化程度，实为传统 RPC 或分布式对象系统所不及。使用基于消息传送的系统只须些许调整，因为它们的行为方式与体系结构师或设计师所期望的传统 n 层系统截然不同。这并不是说基于消息传送的系统更好抑或更差，它们仅仅是不同而已。这时候不再是直接调用对象的方法（这时对象能够保持会话状态或上下文），而是要求消息自身必须是自包含的而且是状态完备的。

这就提出了一个重要问题。

任何特定的开发人员使用任何特定的技术，都会有 4 个明显不同的阶段。

首先是门外汉（Ignorant）阶段。此时，我们只知道该技术存在与否，除此以外，对其能力一无所知。它充其量是经常和其他技术一起提及的一个字母组合，既可能和我们的日常工作十分相关，也可能无关紧要。

第二是探索者（Explorer）阶段。无论自愿与否，由于有某些东西激发了我们的好奇心和求知欲。我们开始迈出丛林探险的第一步，可能是下载一个实现，也可能是阅读几篇文章。开始了解这项技术在更大范围内的基本框架定位及其大致工作方式，不过，我们的实战经验通常仅限于诸如“Hello World”和为数不多的其他例子而已。

第三是熟手（Journeyman）阶段。在运行了一些例子和阅读过几篇文章后，我们对它有了些基本了解，并尝试使用它编写代码。感到用它生成代码并调试所犯的低级错误会非常轻松。无论如何，我们不是专家，但是至少能够写出个东西来编译，并且它在大多数时间可以运行。

最后是大师阶段。在构建一些系统并了解它们的实际运行情况后，我们会对该工具或技术有全面、深入的认识，甚至经常是没有运行代码也能够预知它会如何作用。我们会看到它如何与其他技术取长补短，并了解如何实现一些令人不可思议的效果，比如能够抵御网络

运行中断或机器失效等。回溯到 1999 年 Java 消息服务（JMS）API 初次发布时，它并无任何非商业/开源实现可用，我还清晰地记得当时我边看边想：“不错，它看起来令人很感兴趣，但是没有真正实现，我就没法使用”，并把该规范印刷本放在一边供以后研读。几年之后，我进入 JMS 的探索者和熟手阶段，此时我开始领悟消息传送系统的功能，这一方面是因为有了一些实现，另一方面是因为我自己对其他消息传送系统（最多的是 MSMQ 和 Tibco）的探索，但是主要应归功于 Java 消息服务第 2 版的作者。

时至今日，我依然愧称大师。所幸您和我都知道谁不是大师。

Mark Richards已经在消息传送领域花费了数年心血，他既是架构师和实现者，又是领袖和导师：这首先是因为他的顾问身份，其次是他作为No Fluff Just Stuff（NFJS）专题研讨会定期演讲者的地位 （译注 1）。他对构建基于消息传送的系统的来龙去脉及其要义了然于胸，而且是他提议近乎完全重写Richard Monson-Haefel和Dave Chappell的第 1 版。即使您仍是JMS的门外汉，Mark对消息传送基础知识、实现及设计利弊的完全攻略，会带您快速进入熟手阶段，并使您很快就可了解达到大师阶段所必需的知识结构。

而且，我的朋友是咨询本书的最佳人选。

妙哉，消息传送机制！

—Ted Neward
ThoughtWorks 首席顾问
2008 年 12 月 10 日，于比利时安特卫普 （译注 2）

译注 1：No Fluff Just Stuff 是一个 Java 和开源技术会议，网址是 *http://www.nofluffjuststuff.com*。
译注 2：安特卫普：比利时北部一港口城市，位于布鲁塞尔以北的斯海尔德河边。

前言
Preface

获悉有机会修订《Java 消息服务》，我为此雀跃不已。2000 年，该书第 1 版由 O’Reilly 出版后非常畅销，而且在当时毫无疑问是 JMS 和消息传送的权威参考资料。编写第 2 版，这种机会非常令人兴奋，它赋予这样的名作以新的生命，并为其增加了当今使用消息传送机制的有关新内容。我接手这个项目时完全没有想到，过去 10 年中，消息传送机制（或者更确切地说，我们使用消息传送机制的方式）发生了这么巨大的变化。现在已经研发出若干种消息传送新技术，仅举几个为例：消息驱动 bean（作为 EJB 规范的一部分）、Spring 消息框架、事件驱动体系结构（Event-Driven Architecture，EDA）、面向服务体系结构（Service-Oriented Architecture，SOA）、RESTful JMS 接口，以及企业服务总线（ESB）等。我最初规划的这本小书很快变成了一个庞大的图书项目。

我最初设想在新版中尽量保留原来的内容。不过，根据第 1 版编写以来 JMS 规范的变化，以及消息传送新技术的研发情况，只好大量缩减原来的内容。结果您会发现，第 2 版中有 75%左右是新内容，或者是修订的内容。

在本书第 1 版付印后两三年内，JMS 规范就升级到了 1.1 版。尽管新版本对 JMS 规范并没有做重大修改，它仍然在解决原 JMS 规范某些缺陷方面迈出了重要的一步。该规范最大的变化之一是在统一的公共 API 基础上增加了队列和主题 API，允许队列和主题共享同一事务性工作单元。不过，单单规范的发展并不是促成本书修订出第 2 版的唯一因素。由于 Java 平台已经成熟，相应地，消息传送的方式也已经成熟。从新的消息传送技术和框架，到复杂的集成和吞吐量需求，消息传送机制已经改变了我们考虑和设计系统的方式，特别是在过去的 10 年中更是如此。正是这些因素，加上 JMS 规范的发展，促成了本书第 2 版的问世。

除了第 2 章中的 Chat 应用程序以外，全书所有示例代码均已修改，以反映更新的消息传送用例和阐释第 1 版中未曾包括的一些 JMS 附加特性。

由于显而易见的原因，我在新版中增加了几章内容。您会发现第 1 章新加了几小节，主要是关于 JMS API、更新的消息传送用例及讨论消息传送如何改变系统设计方式。您还会发现关于消息过滤、Java EE 和消息驱动 bean、Spring JMS、消息驱动 POJO 及消息传送设计的几章新内容。

除了增加新章之外，我还对现有各章做了大幅修订。由于更新了全书多个要点的示例代码，所以，我必须逐一大量重写对应的正文。这恰好为我提供了增加另外的小节和议题的机会，尤其是对于第 4 章的点对点消息传送模型和第 5 章的发布/订阅消息传送模型来说，更是如此。我还调整了第 1 版中这两章的顺序，因为我相信，使用队列的点对点消息传送模型更容易深入领会消息传送的概念，而不是使用主题和订阅者的发布/订阅消息传送模型。

我衷心希望本书新版会有助于您更加全面地理解 Java 消息服务和消息传送机制。

—Mark Richards

谁应该阅读本书
Who Should Read This Book?

本书讲解并演示了 Java 消息服务的基础知识。它对于 JMS 规范的底层技术、Java 类和接口、编程模型及其不同实现的讲解，既直观、简洁，又不失严谨、扎实。

虽然本书着重于讲解基础知识，但它并不是一本“傻瓜式”书籍。尽管 JMS API 易于学习，但 API 抽象却是相当复杂的企业级技术。在阅读本书之前，您应该熟练掌握 Java 语言，并有过一些开发业务解决方案的实践经验。我们并不要求您具备消息传送系统方面的相关经验，但是必须掌握 Java 语言的应用知识。

组织结构
Organization

全书共分为 11 章和 4 个附录。第 1 章介绍了消息传送系统、消息传送用例、集中式和分布式体系结构及 JMS 的重要性所在。第 2 章到第 6 章深入探讨了使用点对点和发布/订阅这两种消息传送模型开发 JMS 客户端的细节，包括如何使用消息选择器来过滤消息。第 7 章和第 10 章应被视为“高级议题”，其范围涵盖了消息传送系统的部署和管理等部分。第 8 章是有关 JMS 的 Java 2 企业版（Java EE）概览，包括作为企业级 JavaBean（Enterprise

JavaBeans，EJB）3.0 规范一部分的消息驱动 bean。第 9 章介绍了和消息传送有关的 Spring 框架（Spring Framework）。最后，第 11 章提供了对消息传送相关设计注意事项和反模式等诸多问题的深入思考。

第 1 章：消息传送机制基础

定义了企业消息传送和消息提供者使用的通用体系结构。本章还定义和讲解了 JMS 及其两种编程模型：发布/订阅模型和点对点模型。本章描述了消息传送机制的多个用例和实际场景，同时还介绍了 JMS API 的基础知识。

第 2 章：编写一个简单的示例程序

带领读者一起开发一个简单的发布/订阅 JMS 客户端。

第 3 章：深入剖析一条 JMS 消息

详细分析了 JMS API 最为重要的部分——JMS 消息。

第 4 章：点对点消息传送模型

通过开发一个简单的借方和贷方 JMS 应用程序，深入分析了点对点消息传送模型。本章还介绍了点对点消息传送模型的一些核心亮点，包括消息关联、动态队列、负载均衡及队列浏览等。

第 5 章：发布/订阅消息传送模型

通过强化第 4 章中开发的借方和贷方应用程序，深入分析了发布/订阅消息传送模型。本章还包括持久订阅者、非持久订阅者、动态持久订阅者及临时主题等。

第 6 章：消息过滤

详细讨论了如何使用消息选择器过滤消息。

第 7 章：保证消息传送和事务

全面深入地讲解了有关高级议题，包括保证消息传送、事务、确认、消息编组及失效等。

第 8 章：Java EE 和消息驱动 bean

提供了有关 JMS 的 Java 2 企业版（Java EE）3.0 版本概览，而且还包括了消息驱动 bean（MDB）。

第 9 章：Spring 和 JMS

详细讲解了提供有关 JMS 的 Spring 框架（Spring Framework），包括 Spring JMS 模板（Spring JMS Template）和消息驱动 POJO（MDB）。

第 10 章：部署注意事项

系统深入地分析了选择提供者和部署 JMS 应用程序时应该注意的若干特点和问题。

第 11 章：消息传送设计注意事项

提供了对若干设计注意事项的深入思考和解释说明，包括内部目的地和外部目的地的使用、请求/应答处理，另外还讨论了某些更常见的消息传送反模式。

附录 A：Java 消息服务 API

提供 JMS 包定义的类和接口的快速参考。

附录 B：消息头

提供有关消息头的具体信息。

附录 C：消息属性

提供有关消息属性的具体信息。

附录 D：安装和配置 ActiveMQ

提供运行本书示例所需的 ActiveMQ 安装和配置的具体信息。

软件和版本
Software and versions

本书内容涵盖了 Java 消息服务 1.1 版（Java Message Service version 1.1，JMS 1.1）。它使用来自 Java 6 平台的 Java 语言特性。由于本书重点是开发“厂商无关”的 JMS 客户端和应用程序，因此不会讨论依赖于某个厂商的私有扩展和惯用法。无论使用哪种 JMS 兼容提供者，都可以阅读本书，不过，您应该非常熟悉该提供者的特定安装、部署和运行时管理过程，只有这样才能更好地运行本书的示例。如果要查找某个特定 JMS 提供者 JMS 客户端的安装和运行细节，请查阅该 JMS 提供者的文档；JMS 规范中并未包括这些细节。本书附录 D 提供了使用流行的开源 JMS 提供者 ActiveMQ 运行示例的具体细节。

第 8 章的示例源代码与解释参考了企业级 Java-Beans 3.0（EJB 3）规范。第 9 章的示例源代码与解释参考了 Spring 框架 2.5 版。

要获得本书开发的示例，可以访问 *http://oreilly.com/catalog/9780596522049/examples*。这些例子是按章组织的。另外还提供了为特定厂商而修改的专用源代码。这些特定厂商的示例包括一个 readme.txt 文件，它指向用于下载和安装 JMS 提供者的文档，以及为每个示例安装相应提供者的特定使用说明。

本书使用的约定
Conventions Used in This Book

本书使用如下排版约定：

等宽字体（`Constant Width`）

用于示例代码和代码段、类、变量、方法名称、文本内使用的 Java 关键字、SQL 命令、表名、列名及 XML 元素和标记。

等宽黑体（**`Constant width bold`**）

在某些代码示例中用于强调。

本图标代表提示、建议或一般性注释。

术语 JMS **提供者**是指实现了 JMS API以提供到其企业消息传送服务连接的厂商。术语 JMS 客户端是指使用 JMS API 和 JMS 提供者发送和接收消息的 Java 组件或应用程序。JMS 应用程序是指为提供一个软件解决方案而一起工作的任何 JMS 客户端组合。

使用示例代码
Using Code Examples

本书的目的是帮助您更好地完成工作。一般而言，您可以在自己的程序和文档中使用书中代码。除非您原封不动地大量引用代码，否则您不需要征得我们的许可。例如，编写程序时引用本书中的若干代码片段，这并不需要获得许可，而销售或发布 O'Reilly 图书的示例光盘则需要许可。通过引用本书内容及示例代码来解答问题并不需要许可。将本书中的大量示例代码加入到您的产品文档之中则需要得到许可。

如果您在引用时注明出处，我们将不胜感激，但我们并不强求。标注引用通常应包括书名、作者、出版商及 ISBN（国际标准书号）。例如："Java Message Service，Second Edition，by Mark Richards，Richard Monson-Haefel，and David A. Chappell. Copyright 2009 Mark Richards，978-0-596-52204-9。"

如果您觉得示例代码的使用超出了合理使用或上述许可范围，请随时通过 *permissions@oreilly.com* 和我们取得联系。

如何联系我们

How to Contact Us

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下的联系方式与我们联系。

奥莱理软件（北京）有限公司
北京市 西城区 西直门 南大街2号 成铭大厦C座807室
邮政编码：100080
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

与本书有关的在线信息如下所示。

http://www.oreilly.com/catalog/978059622049（原书）
http://www.oreilly.com.cn/book.php?bn=978-7-121-10050-5（中文版）

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特1号 湖北信息产业科技大厦1402室
邮政编码：430074
电话： (027)87690813　　传真：(027)87690813转817
读者服务网页：*http://bv.csdn.net*
E-mail：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com（投稿信箱）

致谢

Acknowledgments

这些致谢来自 Mark Richards，而且是指本书第 2 版。

写出一本书绝非一人之功；恰恰相反，它是许多人一起不辞辛劳、努力工作的最终结果。我要感谢许多人在此项目期间的辛勤工作及他们的大力支持。

首先，我要赞赏和感谢 Julie Steele 编辑在本项目期间对我的宽容，以及为本书付印所做的编辑、协调和所有这一切不平凡的工作。我还要感谢 Richard Monson-Haefel（和 David Chappell 一起）所做的编写本书第 1 版这样伟大的工作，以及为我提供编写第 2 版的机会。

我要感谢好友和同事 Ted Neward 在繁忙的旅途中为本书作序，并在此项目期间一直为我领路指航。他的建议和指导使得本书新版得以完成。我还要感谢以下朋友：Neal Ford、Scott Davis、Venkat Subramaniam、Brian Sletten、David Bock、Nate Shutta、Stuart Halloway、Jeff Brown、Ken Sipe，以及所有的 No Fluff Just Stuff（NFJS）伙伴，感谢他们在 NFJS 会议内外持之以恒的支持、踊跃的讨论，以及真挚的情谊。他们是最伟大的。

我还要感谢帮助确保资料技术准确性的许多专业技术评论者，他们是：超级软件工程师和技术专家 Ben Messer；August Technology Group，LLC 首席软件开发者和所有者 Tim Berglund；Collaborative Consulting，LLC 首席技术体系结构师 Christian Kenyeres，最后还有（但绝非最不重要）Ken Yu 和 Igor Polevoy。我知道在假期编辑和审查原稿并不容易（对我来说恐怕是很糟糕的时间安排），但是他们的实战经验、建议、注释、意见及技术编辑，使得本书日臻完美。

感谢 Scotland Macallan Distillery 的亲属们，他们做出了世界上口味最淡的苏格兰威士忌。它减轻了我写作时那些漫漫长夜的痛苦，特别是隆冬季节。

最后，我要感谢可爱的妻子 Rebecca 对我写书持之以恒的支持。Rebecca，你对我来说意味着整个世界，直至永远。

第 1 版致谢

Acknowledgments from the First Edition

这些致谢摘录自本书第 1 版，来自原作者 Richard Monson-Haefel 和 David A.Chappell。

虽然封面仅有两位作者署名，但是，本书成书和面世的荣耀应由多人共享。Michael Loukides 编辑对本书的成功至关重要。没有他的丰富经验、专业技能和悉心指导，写出本书将几无可能。

许多专业技术评论者的帮助，确保了资料的技术准确性，并且真正阐释了 Java 消息服务的精髓。在此要特别鸣谢：Sun Microsystems 公司的 Joseph Fialli、Anne Thomas Manes 和 Chris Kasso，Progress 的 Andrew Neumann 和 Giovanni Boschi，Softwired 的 Thomas Haas，International Systems Group 的 Mikhail Rizkin，以及 ExoLab 的 Jim Alateras 等。这些技术专家的贡献对保证本书技术和概念的正确性至关重要。他们实现了行业标准和实战经验二者的融合，并使得本书成为迄今出版的 JMS 图书当中最好的一本。

我还要感谢 Sun Microsystems 公司 Java 2 企业版的首席体系结构师 Mark Hapner，他为我们解答了最为棘手的几个问题。感谢 Sun Microsystems 公司主办的 JMS-INTEREST 邮件

列表的所有参与者，感谢他们热情的、内涵丰富的邮件。

我要特别感谢 SonicMQ 技术写作团队的 George St.Maurice 参与组织 O’Reilly 网站的示例。最后，我们要把最真挚的谢意献给我们的家庭。Richard Monson-Haefel 要感谢妻子 Hollie 对他接连出书的一贯支持和帮助。她的爱使得一切皆有可能。David Chappell 要感谢妻子 Wendy 和孩子 Dave、Amy 和 Chris 在他奋力写作期间对他的宽容。

David Chappell 还要感谢 Progress SonicMQ 团队的部分成员：Bill Wood、Andy Neumann、Giovanni Boschi、Christine Semeniuk、David Grigglestone、Bill Cullen、Perry Yin、Kathy Guo、Mitchell Horowitz、Greg O’Connor、Mike Theroux、Ron Rudis、Charlie Nuzzolo、Jeanne Abmayr、Oriana Merlo 和 George St.Maurice，感谢他们帮助确定合适的议题，并确保对这些议题的正确阐释。另外，David Chappell 还要特别感谢 George Chappell 在“分裂不定式”方面的帮助。

第 1 章

消息传送机制基础

Messaging Basics

近些年来，系统的复杂性和先进性增长非常显著。现在对系统的可靠性、可伸缩性和灵活性等的要求要比以前更高，这种需要已经促成了更为复杂的先进体系结构的出现。为了适应这种对更好更快的系统日益增长的需求，体系结构师、设计师和开发者已经开始利用消息传送机制（messaging），作为解决这些复杂问题的一种方式。

自从 2000 年本书第 1 版问世以来，消息传送机制已经取得了长足的进展，特别是和 Java 平台有关的消息传送机制更是如此。尽管 Java 消息服务（Java Message Service，JMS）API 自 1999 年推出以来，未发生显著的改变，而消息传送机制使用的方式则有了很大的变化。解决可靠性和可伸缩性问题，已经广泛使用了消息传送机制。与此同时，它还被用于解决许多商业应用程序和非商业应用程序遇到的大量其他问题。

异构集成（heterogeneous integration）是消息传送机制在其中起关键作用的一个领域。无论它的成因是合并、并购、业务需求，或者仅仅是技术方向上的一个变化，越来越多的公司都正面临着在企业内部、跨企业集成异构系统和应用程序的问题。在一家公司或部门内部，遇到由 Java EE、Microsoft 公司的.NET、Tuxedo、Oryes，甚至是大型机上的 CICS 所组成的多种技术和平台，这种情况毫不足奇。

消息传送机制还具有异步处理请求的能力，它为系统体系结构师和开发者提供的解决方案，能够减轻或消除系统瓶颈，并提高最终用户的生产率和系统的整体可伸缩性。由于消息传送机制能够实现组件之间的高度去耦，因此，使用这种机制的系统还具有高度的体系结构灵活性和敏捷性。

应用程序到应用程序（application-to-application）类型的消息传送系统，在用于业务系统时，通常称为企业消息传送系统（enterprise messaging system），或者称为面向消息的中间件（message-Oriented Middleware，MOM）。企业消息传送系统允许两个或更多的应用程序以消息的形式来交换信息。这时，一条消息就是业务数据和网络路由头的一个自包含

（self-contained）数据包。消息中包含的业务数据可以是任何内容，这取决于业务场景，而且，它通常包含了有关某些业务事务的信息。在企业消息传送系统当中，消息会将另一个系统中的某些事务或事件通知给一个应用程序。

通过使用面向消息的中间件，消息通过网络从一个应用程序传送到另一个应用程序之中。企业中间件产品能够确保消息在应用程序中间的正确分发。此外，对于那些需要可靠地大量交换消息的企业，这些产品通常为它们提供了容错和负载均衡、可伸缩性和事务性的支持。

企业消息传送系统厂商（enterprise messaging vendor）在交换消息时，使用不同的消息格式和网络协议，但是它们的基本语义是相同的。一个 API 可用于创建一条消息、加载应用程序数据（消息有效负载）、分配路由信息和发送这条消息。接收其他应用程序生产的消息也使用相同的 API。

在所有的现代企业消息传送系统中，应用程序通过虚拟通道来交换消息，这些虚拟通道称为目的地。发送一条消息时，是发送到一个目的地（也就是队列或主题），而不是发送到某个特定的应用程序。在这个目的地中，订阅或注册对该目的地感兴趣的所有应用程序都可以接收这条消息。这样一来，接收消息的应用程序和发送消息的应用程序就能够实现去耦。发送者和接收者并非在任何情况下都相互绑定在一起，而且，它们可以在自己认为合适的时候发送和接收消息。

所有的企业消息传送系统厂商都为应用程序开发者提供了一个 API，用于发送和接收消息。当消息传送系统厂商各自实现自己的网络协议、路由及管理工具时，由不同厂商提供的开发者 API 的基本语义是相同的。正是 API 的这种相似性使得 Java 消息服务成为可能。

JMS 是一种厂商无关（vendor-agnostic）的 Java API，它可以供多个不同的企业消息厂商使用。JMS 与 JDBC 非常相似，应用程序开发者能够重用同样的 API 来访问多种不同的系统。如果厂商提供了 JMS 兼容的服务提供程序，我们就可以使用 JMS API 来向其发送消息，或者从该厂商处接收消息。举例来说，您要是使用 IBM 的 WebSphere MQ，您就可以使用相同的 JMS API 来发送使用 SonicMQ 的消息。本书的目的正是要说明企业消息传送系统是如何工作的，特别是 JMS 是如何与这些系统一起使用的。本书第 2 版将重点讨论 2002 年 3 月推出的 JMS 规范的最新版本 JMS 1.1。

本章的其余部分将探讨企业消息传送和 JMS 的更多细节，以便您在随后各章中学习 JMS API 和消息传送概念时，具备足够扎实的基础。我们在本书中所作的唯一假设，就是您已经对 Java 编程语言非常熟悉。

1.1 消息传送机制的优点
The Advantages of Messaging

正如本章开始时所言，消息传送机制能够解决诸多体系结构性挑战，比如说异构集成、可伸缩性、系统瓶颈、并发处理，以及整体体系结构灵活性和敏捷性等。本节将简要介绍 JMS 和消息传送机制的更多优点和用途。

1.1.1 异构集成
Heterogeneous Integration

异构平台的通信和集成可能是消息传送机制最为典型的使用范例。使用消息传送机制，您可以向在完全不同的平台上实现的应用程序和系统请求调用服务。许多开源消息传送系统和商业消息传送系统使用了一种集成消息桥（message bridge），该桥能够将使用 JMS 的一条消息转换为通用的内部消息格式，以此来实现 Java 和其他语言和平台之间的无缝连接。这些消息传送系统的例子有 ActiveMQ（开源系统）和 IBM WebSphere MQ（商业系统）。这两种消息传送系统都支持 JMS，不过，它们都还开放了一个本机 API，供非 Java（比如 C 和 C++）消息传送客户端使用。这里的关键之处在于，根据厂商的不同，使用 JMS 实现和非 Java 或非 JMS 的消息传送客户端进行通信，是可能的。

从历史上看，在应对异构系统的集成问题方面，已经有许多方法。较早期的一些解决方案，多通过FTP或其他文件传输手段来传输信息，包括使用将一个磁盘或磁带从一台机器拷贝到另一台机器上的经典“人力网络（Sneakernet）”方 法 （译注 1）。使用数据库在两个异构系统或应用程序之间来共享信息，这是另一种迄今仍在广泛应用的常见方法。远程过程调用（Remote Procedure Call），或 者简称RPC，也是在不同系统之间共享数据和功能的另一种方法。不过，这些解决方案各自都有优缺点，只有消息传送机制提供的去耦解决方案，能够真正实现跨应用程序或子系统共享数据和功能。不久前，Web服务（Web Services）已经作为异构系统集成的另一种可能的解决方案脱颖而出。不过，Web服务在可靠性方面的欠缺，使得消息传送机制成为一种更佳的集成选择。

1.1.2 缓解系统瓶颈
Reduce System Bottlenecks

无论何时，只要您的某个进程跟不上对它访问请求的速度，那么就会产生系统和应用程序的瓶颈问题。系统瓶颈的一个经典例子是：在一个拙劣优化（poorly tuned）的数据库中，应用程序和进程在一直等待，直到数据库连接可用或数据库锁被释放为止。在系统的某些地方，系统将出现阻塞，响应会越来越慢，直到最终请求出现超时现象。

关于系统瓶颈的一个很好的类比就是向漏斗中注水。由于漏斗只能够允许一定的水量通过，因此，它就成为了一个瓶颈。由于进入漏斗的水量在不断增加，漏斗中的水最终会溢

译注 1：sneakernet 是人力网络，指由人将资料拷贝到磁盘，然后送到另一台机器上拷贝，用这种方法来传送资料。

出，因为这些水无法足够快地流出漏斗，以使得该漏斗能够处理越来越多的水流。IT 系统的工作方式与此差不多相同：某些组件只能处理数量有限的请求，而且它将很快会变成系统瓶颈。

再回过头来看我们的例子：如果单个漏斗每分钟能够“处理”一升水，而同时却有两升水注入漏斗，那么漏斗最终会阻塞，漏斗中的水也会溢出。不过，如果再增加两个漏斗来处理水流，那么在理论上，我们现在每分钟就能够“处理”3 升水，从而满足需求。同样地，在 IT 系统内部，消息传送机制可以用于缓解乃至消除系统瓶颈。与一个同步组件处理众多请求时，众多请求一个接一个地积聚阻塞不同，这时候请求会发送到一个消息传送系统，该系统将该请求分发给多个消息侦听器组件。如此一来，就缓解了单独采用点对点同步连接带来的系统瓶颈，在某些情况下，甚至可以完全消除这些瓶颈。

1.1.3 提高可伸缩性
Increase Scalability

和缓解系统瓶颈的方式非常相似，消息传送机制还可以用于提高系统的整体可伸缩性和吞吐量，同时，它还能够有效地缩短响应时间。通过引入能够并发处理不同消息的多个消息接收者，消息传送系统的可伸缩性得以实现。由于消息排队等候处理，队列中的消息数量，或者称为**队列深度**（queue depth），开始逐渐增大。随着队列深度的增大，系统响应时间开始变长，与此同时，吞吐量也会下降。提高系统可伸缩性的一种方法就是，向队列中添加多个并发消息侦听器（和我们在上面的例子中增加漏斗相类似），以便并发处理更多的请求。

提高系统整体可伸缩性的另一种方法，就是要尽可能地利用系统的异步方式。如此一来，按照这种方式的组件去耦就会允许系统水平增长，而此时硬件资源则成为了主要的限制因素。虽然这看起来可能像一剂灵丹妙药（silver bullet），不过，中间件只能在系统的另一个主要瓶颈——数据库——的实质性限制之内进行水平扩展。您可以在单独的队列中拥有数百个甚至数千个消息侦听器，以此来提供同时处理多条消息的能力，但是，数据库却可能只能处理数量有限的并发请求。尽管在应对数据库瓶颈问题方面，已经有多种复杂的技术，然而现实却是：能将中间件层扩展到什么程度，始终会受到数据库这种实质性的限制。

1.1.4 提高最终用户生产率
Increase End User Productivity

使用异步消息传送机制还能够提高最终用户的生产率。让我们设想一下这样一种情景：最终用户通过基于 web 的用户界面或桌面用户界面，向系统发出一个请求，这个接口要花好几分钟来运行。在此期间，最终用户一直在等待结果，而无法完成其他任何工作。通过使

用异步消息传送机制，最终用户能够向系统发出一个请求，并立即得到回应，表明该请求已被接收。现在，当执行长时间运行请求时，最终用户可以在系统上继续做其他工作。一旦该请求处理完毕，就立即告知最终用户，并将处理结果回传给最终用户。通过使用消息传送机制，最终用户就能够以更短的等待时间来完成更多的工作，使得最终用户拥有更高的生产率。

许多前台（front-office)交易系统在交易应用程序和后端系统之间使用这种消息传送策略。这种基于消息传送的体系结构类型，允许交易者执行其他工作而无须等待系统响应。不过，作为这种灵活性和生产率提升之间的折衷平衡，它增加了系统的复杂性。一名优秀的体系结构师会始终寻求机会，使得系统的不同方面均采用异步方式，无论是在用户界面和系统之间，还是在系统内部各组件之间。

1.1.5 体系结构灵活性和敏捷性
Architecture Flexibility and Agility

使用消息传送机制作为企业体系结构整体解决方案的一部分，这为未来的体系结构灵活性和敏捷性留有了更大的余地。这些优良特性是通过使用抽象和去耦来实现的。使用消息传送机制，各个子系统、组件，乃至服务都能够被抽象出来，甚至可以达到在对其知之甚少，甚至是一无所知的情况下，即可用客户端组件所取代的程度。

体系结构敏捷性是对不断变化的环境快速响应的能力。通过使用消息传送机制来抽象和去耦组件，就能够快速地响应软件、硬件，甚至是业务的变化。更替一个系统、更换一个技术平台，甚至是改变一家厂商的解决方案，而不会影响到客户端应用程序，这种能力可以使用消息传送机制的抽象而获得。使用消息传送机制方式，消息生产者或是客户端组件都不会知道接收组件使用的是哪种编程语言或平台，组件或服务位于何处，组件或服务实现的名称是什么，甚至用于访问该组件或服务的是哪种协议。正是借助于这些不同级别的抽象，我们才能够更加容易地替换组件和子系统，从而提高了体系结构的敏捷性。

1.2 企业消息传送
Enterprise Messaging

企业消息传送并不是什么新概念。消息传送产品，比如 IBM WebSphere MQ、SonicMQ、Microsoft Message Queuing（mSMQ）及 TIBCO Rendezvous 等，已经存在了很多年。最近，ActiveMQ 等几种开源消息传送产品也已经进入市场，并且正在企业生产环境中使用。此外，面向服务体系结构（Service-Oriented Architecture，SOA）的推出，已经促生了称为企业服务总线（Enterprise Service Bus，ESB）的新型消息传送产品。尽管大多数企业服务总线允许使用基于 HTTP 的通信，基于消息传送的系统则仍然继续保持了在大多数生产企业系统当中的标准地位。

企业消息传送的一个关键概念就是：消息是通过网络从一个系统**异步**传送给其他系统的。异步传送一条消息意味着：发送者不需要等待接收者接收或处理该消息；它可以自由地发送消息并持续进行处理。异步消息可作为独立的自主单元（autonomous unit），也就是说，每条消息都是自包含的，它带有处理其业务逻辑所需的所有数据和状态信息。

在异步消息传送机制中，应用程序使用一个简单的 API 来构建一条消息，然后再将该消息转发给面向消息的中间件，以便传送给一个或多个的预定接收者（参见图 1-1）。一条消息就是一个业务数据包，它通过网络从一个应用程序发送给其他应用程序。消息应该是自描述的（Self-describing），因为它应该包含所有必要的上下文，以便允许接收者独立地完成它们的工作。

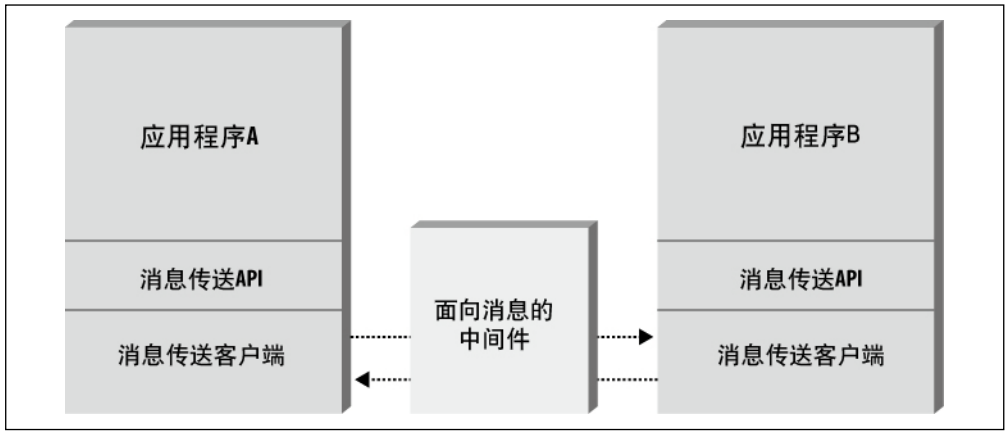


图 1-1：面向消息的中间件

现在，面向消息的中间件体系结构在其实现方面，范围从依赖消息服务器来执行路由选择的集中式体系结构，一直到将“服务器”处理分发给客户端机器的分散式体系结构，形式可以说是多种多样。在网络传输层，它使用了包括 TCP/IP、HTTP、SSL 和 IP 组播在内的众多协议。根据使用模式的不同，一些消息传送产品则是采用二者混合的方法。

很有必要解释一下术语客户端（client）的含义。消息传送系统由消息传送客户端（messaging client）和几种消息传送中间件服务器所组成。客户端向消息传送服务器发送消息，该服务器随后再将那些消息分发给其他客户端。客户端是使用消息传送 API 的一个业务应用程序或组件（具体到本书来说就是 JMS）。

1.2.1 集中式体系结构
Centralized Architectures

使用集中式体系结构的企业消息传送系统，依赖于一台**消息服务器**（message server）。消息服务器，也称为消息路由器（message router）或代理（broker），它负责从一个消息传送客户端向其他消息传送客户端传送消息。消息服务器可以实现一个发送客户端和其他接收客户端之间的去耦。客户端仅仅会看到消息传送服务器，而不会看到其他客户端，这将允许在不会影响系统整体的情况下添加和删除客户端。

通常，集中式体系结构使用的是一种星型（hub-and-spoke）拓扑结构。最简单的例子就是，只有一台集中式消息服务器及其相连的所有客户端。如图 1-2 所示，星型体系结构适合于一个最小数量的网络连接，同时，它仍然允许系统的任何部分和其他部分进行通信。

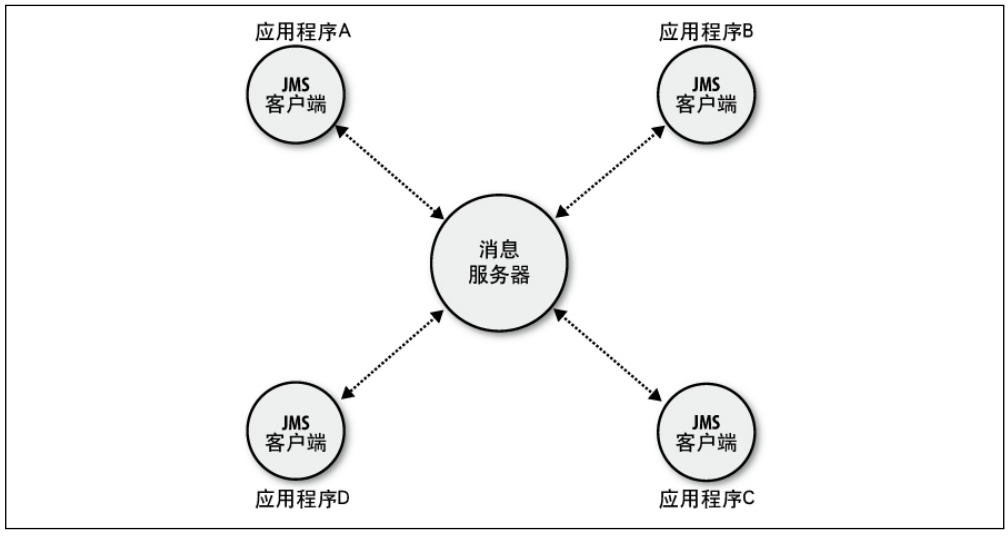


图 1-2：集中式星型体系结构

实际上，集中式消息服务器可以是按照逻辑单元（logical unit)方式运行的一个分布式服务器集群（cluster）。

1.2.2 分散式体系结构
Decentralized Architectures

当前，所有的分散式体系结构都在使用网络层 IP 组播。基于组播的消息传送系统没有集中式服务器。一些服务器功能（持久性、事务和安全性）作为一个客户端的本地部分嵌入进来，而此时消息路由则利用 IP 组播协议委托给网络层。

IP 组播允许应用程序加入到一个或多个 IP 组播组（IP multicast group）之中；每个组播组使用一个 IP 网络地址，它将接收到的所有消息重新发布（redistribute)给组内的所有成员。这样，应用程序就能够向一个 IP 组播地址发送消息，并期望网络层正确地重新发布这些消息（参见图 1-3）。与集中式体系结构不同，分布式体系结构不需要用于消息路由的服务器，因为网络会自动地处理路由。然而，每个客户端仍然需要具有像服务器那样的其他功能，比如说消息持久性和“一次而且仅仅一次”传送（delivery）之类的消息传送语义等。

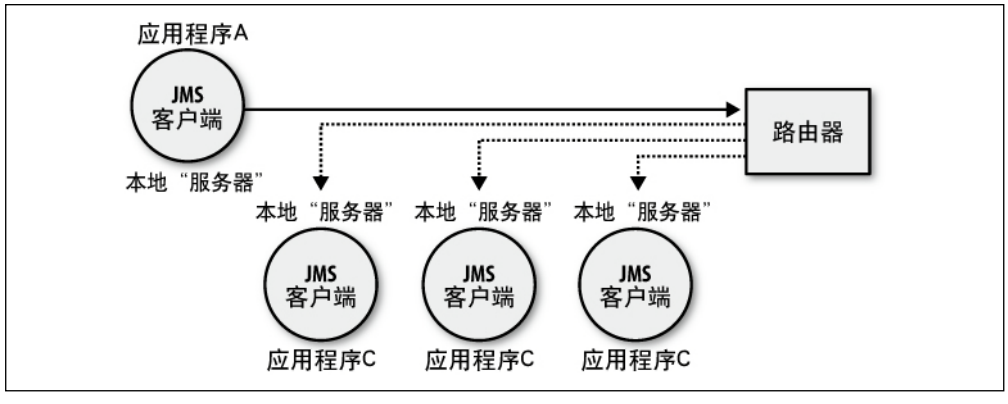


图 1-3：分散式 IP 组播体系结构

1.2.3 混合体系结构
Hybrid Architectures

一个分散式体系结构通常意味着正在使用 IP 组播协议，而一个集中式体系结构则意味着 TCP/IP 在不同组件之间实现通信的基础。一个消息传送系统厂商的体系结构还可能会将这两种方法结合起来。客户端可以使用 TCP/IP 连接到一个守护进程（daemon process），它使用 IP 组播组依次和其他守护进程实现通信。

1.2.4 以集中式体系结构作为模型
Centralized Architecture As a Model

无论是分散式体系结构还是集中式体系结构，二者最终都会在企业消息传送中获得用武之地。第 10 章会更加详细地讨论分布式体系结构相对集中式体系结构的优缺点。同时，我们需要一个通用模型，用于讨论企业消息传送的其他方面。为简化讨论起见，本书使用集中式体系结构，作为企业消息传送的逻辑视角。这仅仅是为了方便，并非支持集中式体系

结构优于分散式体系结构的观点。本书中常用的术语**消息服务器**是指负责路由选择和发布消息的底层基础体系结构。在集中式体系结构中，消息服务器是一个中间件服务器或服务器集群。而在分散式体系结构中，服务器则是指类似于本地服务器的客户端工具。

1.3 消息传送模型
Messaging Models

JMS 支持两类消息传送模型：点对点模型和发布/订阅模型。有时候，又称这些消息传送模型为**消息传送域**。点对点消息传送模型和发布/订阅消息传送模型经常分别缩写为 p2p 和 Pub/Sub。在本书中，这二者的原文和缩写都会用到。

简而言之，发布/订阅模型设计用于一对多（one-to-many）消息广播，而点对点模型则设计用于一对一（one-to-one）消息传送（参见图 1-4）。

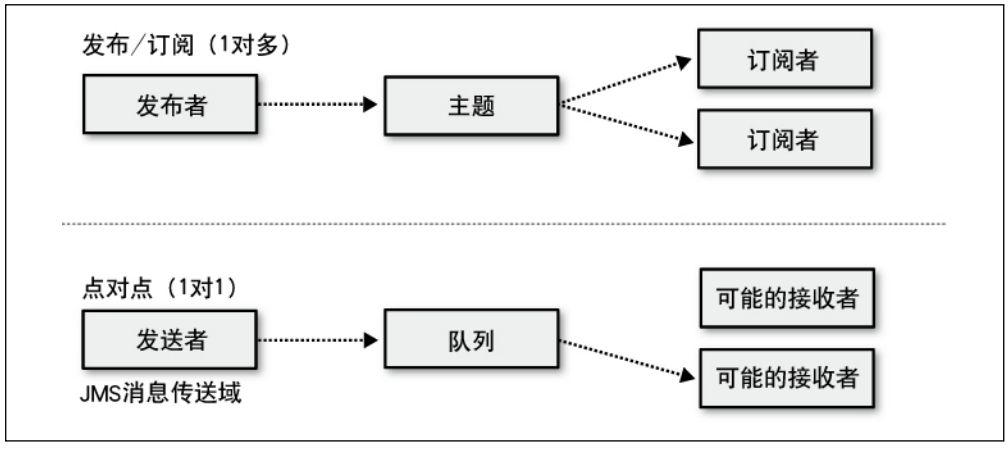


图 1-4：JMS 消息传送域

从 JMS 的视角来看，消息传送客户端称为 JMS 客户端（JMS client），而消息传送系统则称为 JMS **提供者**（JMS provider）。一个 JMS **应用程序**是由多个 JMS 客户端和（通常是）一个 JMS 提供者所组成的业务系统。

此外，生产消息的 JMS 客户端称为**消息生产者**（message producer），而接收消息的 JMS 客户端则称为**消息消费者**（message consumer）。一个 JMS 客户端可以既是消息生产者，又是消息消费者。当我们使用术语**消费者**（consumer）或**生产者**（producer）时，我们分别是指消费或生产消息的一个 JMS 客户端。在全书中，我们都会用到这个术语。

1.3.1 点对点模型
Point-to-Point

点对点消息传送模型允许 JMS 客户端通过**队列**（queue）这个虚拟通道来同步和异步发送、接收消息。在点对点模型中，消息生产者称为**发送者**（Sender），而消息消费者则称为**接收者**（receiver）。传统上，点对点模型是一个基于拉取（Pull）或基于轮询（polling）的消息传送模型，这种模型从队列中请求消息，而不是自动地将消息推送到客户端。点对点消息传送模型的一个突出特点就是：发送到队列的消息被一个而且仅仅一个接收者所接收，即使可能有多个接收者在一个队列中侦听同一消息时，也是如此。

点对点消息传送模型既支持异步“即发即弃（fire and forget）”消息传送方式，又支持同步请求/应答消息传送方式。点对点消息传送模型比发布/订阅模型具有更强的耦合性，发送者通常会知道消息将被如何使用，而且也会知道谁将接收该消息。举例来说，发送者可能会向一个队列发送一个证券交易订单并等待响应，响应中应包含一个交易确认码。这样一来，消息发送者就会知道消息接收者将要处理交易订单。另一个例子就是一个生成长时间运行报告的异步请求。发送者发出报告请求，而当该报告准备就绪时，就会给发送者发送一条通知消息。在这种情况下，发送者就会知道消息接收者将要处理该消息并创建报告。

点对点模型支持负载均衡，它允许多个接收者侦听同一队列，并以此来分配负载。如图 1-4 所示，JMS 提供者负责管理队列，确保每条消息被组内下一个可用的接收者消费一次，而且仅仅一次。JMS 规范没有规定在多个接收者中间分发消息的规则，尽管某些 JMS 厂商已经选择实现此规则来提升负载均衡能力。点对点模型还具有其他优点，比如说，队列浏览器允许客户端在消费其消息之前查看队列内容——在发布/订阅模型中，并没有这种浏览器的概念。第 4 章会详细介绍点对点消息传送模型的更多细节。

1.3.2 发布/订阅模型
Publish-and-Subscribe

在发布/订阅模型中，消息会被发布到一个名为**主题**（topic）的虚拟通道中。消息生产者称为**发布者**（publisher），而消息消费者则称为**订阅者**（subscriber）。与点对点模型不同，使用发布/订阅模型发布到一个主题的消息，能够由多个订阅者所接收。有时候，也称这项技术为**广播**（broadcasting）消息。每个订阅者都会接收到每条消息的一个副本。总地来说，发布/订阅消息传送模型基本上是一个基于推送（push）的模型，其中消息自动地向消费者广播，它们无须请求或轮询主题来获得新消息。

发布/订阅模型的去耦能力要比 p2p 模型更强，消息发布者通常不会意识到有多少订阅者或那些订阅者如何处理这些消息。举例来说，假定每次在 Java 应用程序发生异常时，向一个主题发布一条消息。发布者的责任仅仅是广播发生了一个异常。该发布者不会知道，或者说通常也不关心如何使用该消息。例如，有可能是订阅者根据该异常向开发人员或支持人员发送了一封电子邮件，也有可能是订阅者收集不同类型的异常数目用于生成报告，甚至是订阅者根据异常的类型，使用这个信息来通知随叫随到（on-call）的技术支持人员。

在发布/订阅消息传送模型内部，有多种不同类型的订阅者。非持久订阅者是临时订阅类型，它们只是在主动侦听主题时才接收消息。而另一方面，持久订阅者将接收到发布的每条消息的一个副本，即便在发布消息，它们处于“离线”状态时也是如此。另外还有动态持久订阅者和受管的持久订阅者等类型。第 2 章和第 5 章会详细讨论发布/订阅消息传送模型的更多细节。

1.4 JMS API

JMS 是 Sun Microsystems 公司通过 JSR-914 项目创建的一种企业消息传送 API。JMS 自身并不是一种消息传送系统；它是消息传送客户端和消息传送系统通信时所需接口和类的一个抽象。与 JDBC 抽象（JDBC abstract）访问关系数据库、JNDI 抽象访问命名和目录服务的方式一样，JMS 抽象可以访问消息提供者。使用 JMS，应用程序的消息传送客户端可以实现跨消息服务器产品的移植。

JMS 的创立是整个行业努力的结果。Sun Microsystems 公司领衔 JMS 规范的制订工作，并自始至终和消息传送系统厂商保持着密切的合作关系。它们的最初目标只是提供一个 Java API，用于企业消息传送系统的连接。然而，这个目标很快就变得更加宏大：支持消息传送机制，使其成为一流的 Java 分布式计算范例，并且可以和 CORBA 及企业级 JavaBean（Enterprise JavaBeans）等基于 RPC 的系统相提并论。Sun Microsystems 公司 JMS 规范项目主管 Mark Hapne 就此解释说：

> 很多 MOM（面向消息的中间件）厂商参与了 JMS 的创建工作。它得益于整个行业，而不仅仅只是 Sun 公司的努力。Sun 公司担当了 JMS 规范的领导角色(spec lead)，并且确确实实起到了保驾护航的作用，但是，如果没有消息传送系统厂商的直接参与，这项工作就不会取得成功。尽管我们最初的目标是为连接 MOM 系统提供一个 Java API，但是这个工作目标已经变得更加宏大：支持消息传送机制，使其成为一个和 RPC 具有同等地位的一流 Java 分布式计算范例。

上述努力的结果就是形成了一个具有单项优势（best-of-breed）的、健壮（robust）的规范，它包括了一组丰富的消息传送语义，并和简单而灵活的API相结合，用于将消息传送合并到应用程序之中。它的设计初衷是，除了新增的厂商之外，现有的消息传送系统厂商也会支持JMS API（译注 2）。

JMS API 可以分为 3 个主要部分：公共 API、点对点 API 和发布/订阅 API。在 JMS1.1 中，公共 API 可被用于向一个队列或一个主题发送消息，或从其中接收消息。点对点 API 专门用于使用队列的消息传送，而发布/订阅 API 则专门用于使用主题的消息传送。

在 JMS 公共 API 内部，和发送和接收 JMS 消息有关的 JMS API 接口主要有 7 个：

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

在这些公共接口中，`ConnectionFactory` 和 `Destination` 必须使用 JNDI（遵照 JMS 规范要求）从提供者处获得。其他接口则可以通过工厂方法在不同的 API 接口中创建。举例来说，一旦有了一个 `ConnectionFactory`，就可以创建一个 `Connection`。一旦有了一个 `Connection`，就可以创建一个 `Session`。而一旦有了一个 `Session`，就可以创建一个 `Message`、`MessageProducer` 和 `MessageConsumer`。这 7 个主要的 JMS 公共 API 接口之间的关系如图 1-5 所示。

在 JMS 中，是 `Session` 对象保存着用于消息传送的事务性工作单元（transactional unit），而不是 `Connection` 对象。这和 JDBC 不同，JDBC 中是 `Connection` 对象保存事务性工作单元。这就意味着在使用 JMS 时，一个应用程序通常只会有一个 `Connection` 对象，但是它可以有一个 `Session` 对象池。

另外，还有和异常处理、消息优先级及消息持久性有关的其他接口。这些接口和其他 API 接口将在全书和附录 A 中详细讨论。

译注 2：单项优势（best-of-breed）是近年来计算机系统开发的一种新理念，由用户挑选最好的单项系统后组合成新系统。较之以往的集成系统，这种开发理念具有更大的灵活性。

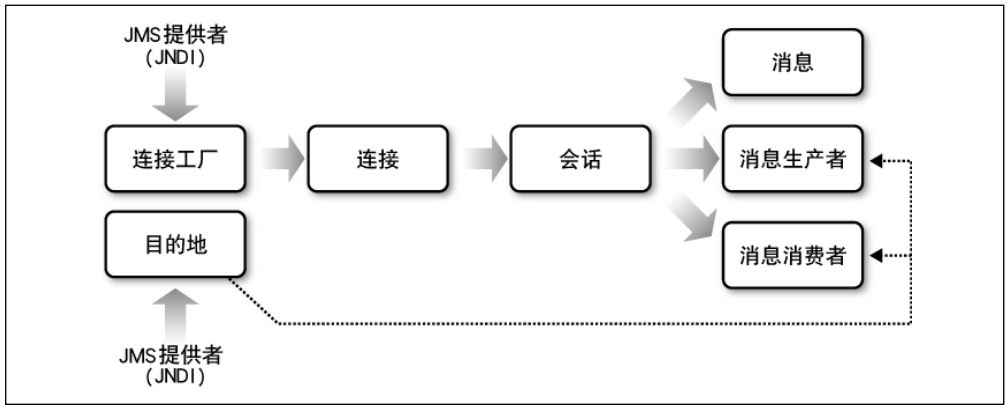


图 1-5：JMS 公共 API 核心接口

1.4.1 点对点 API
Point-to-Point API

一旦弄懂了 JMS 的公共 API，JMS API 的其余部分就非常容易类推和理解了。点对点消息传送模型 API 特指 JMS API 之内基于队列的接口。下面是用于向一个队列发送和从一个队列接收消息的接口：

- `QueueConnectionFactory`
- `Queue`
- `QueueConnection`
- `QueueSession`
- `Message`
- `QueueSender`
- `QueueReceiver`

与在 JMS 公共 API 中一样，`QueueConnectionFactory` 和 `Queue` 对象必须通过 JNDI 从 JMS 提供者处获得（按照 JMS 规范的要求）。请注意：大多数接口名称仅仅是在公共 API 接口名称之前添加 `Queue` 一词而已。不同之处在于，这里是称为 `Queue` 的 `Destination` 接口，而 `MessageProducer` 和 `MessageConsumer` 接口则分别称为 `QueueSender` 和 `QueueReceiver`。图 1-6 显示了基于队列的 JMS API 接口之间的流程和关系。

一般来说，使用点对点消息传送模型的应用程序将使用基于队列的 API，而不是使用公共 API。

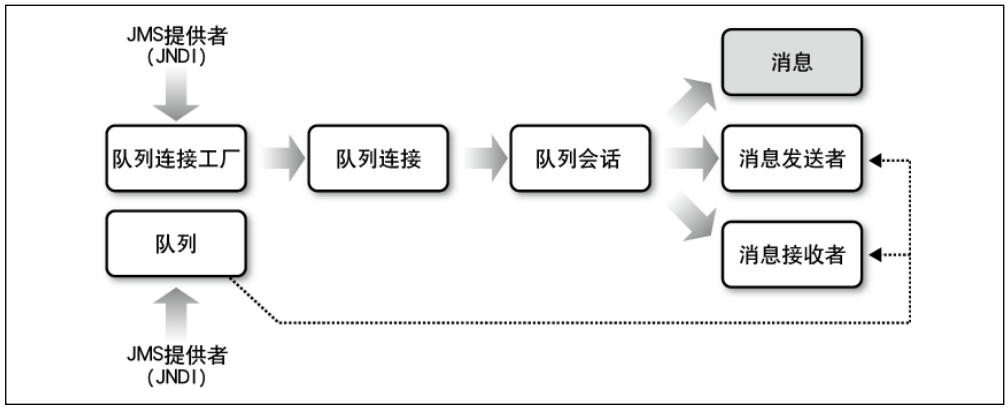


图 1-6：JMS 点对点 API 核心接口

1.4.2 发布/订阅 API
Publish-and-Subscribe API

由于基于主题的 JMS API 类似于基于队列的 API，因此在大多数情况下，`Queue` 这个词会由 `Topic` 取代。发布/订阅消息传送模型内部使用的接口如下：

- `TopicConnectionFactory`
- `Topic`
- `TopicConnection`
- `TopicSession`
- `Message`
- `TopicPublisher`
- `TopicSubscriber`

请注意：除了 `TopicPublisher` 和 `TopicSubscriber` 不同以外，发布/订阅域中的接口和 p2p 域中的那些接口名称基本类似。JMS API 在这一点上非常明显。正如本章开始时所述的那样，发布/订阅模型使用**主题**、**发布者**和**订阅者**这些术语，而 p2p 模型使用的则是**队列**、**发送者**和**接收者**。请注意这些术语是如何和 API 接口名称相匹配的。基于主题的 JMS API 接口的关系和流程如图 1-7 所示。

1.5 实际场景
Real-World Scenarios

到目前为止，我们对于企业消息的讨论显得稍微有点抽象。本节力图通过一些实际的场景，让您对企业消息传送系统能够解决的问题类型有一个更为深入的理解。

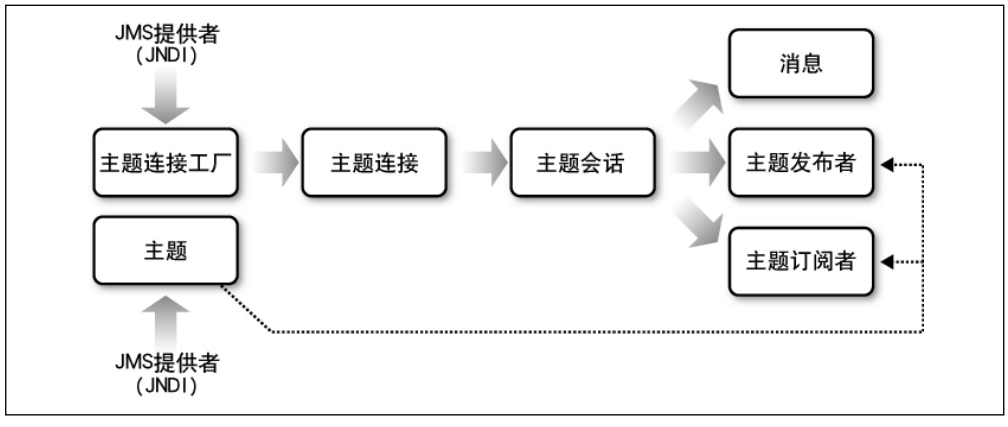


图 1-7：JMS 发布/订阅 API 核心接口

1.5.1 面向服务体系结构

Service-Oriented Architecture

面向服务体系结构（Service-Oriented Architecture，SOA）作为一种体系结构体系，定义了从对应的企业服务实现中抽象出来的业务服务。因此，SOA 已经促生了一种称为企业服务总线（ESB）的新类型中间件。在 SOA 发展的早期阶段，大多数 ESB 是作为消息代理实现的，由此，消息传送层内部的组件会用于执行某种智能路由选择，或者用于在传送消息之前进行消息转换。这些早期的消息代理，已经发展成为成熟的商业 ESB 产品和开源 ESB 产品，它们在核心之处使用了消息传送机制。尽管某些 ESB 产品能够支持传统的非 JMS 型 HTTP 传送，但是，大多数企业级产品的实现仍然采用消息传送作为通信协议。

在需要将业务服务从其底层实现中完全抽象出来的 SOA 内部构建抽象层时，消息传送是一种极好的手段。使用消息传送机制，业务服务无须关心对应的实现服务位于何处、使用哪种语言编程、在哪种平台上部署，甚至不需要关心实现服务的名称。消息传送机制还提供了在 SOA 环境内部所需的可伸缩性，另外，它还对进、出 ESB 的请求提供了一种健壮的监控手段。现在，几乎所有可用的商业 ESB 产品和开源 ESB 产品都支持 JMS 消息传送作为一种通信协议——最值得注意的例外就是 Microsoft 公司的消息传送产品系列（例如，BizTalk 和 MSMQ）。

业内对于 SOA 的热衷和应用，反过来又掀起了普遍使用消息传送解决方案的热潮。尽管完全成熟的 SOA 实现尚在不断发展之中，许多公司就已经开始转向消息传送解决方案，将其作为迈向 SOA 的其中一步。

1.5.2 事件驱动体系结构
Event-Driven Architecture

事件驱动体系结构（EDA）作为一种体系结构体系，它建立在下列前提之上：进程和事件编排是动态的和非常复杂的，因而通过一个中央编排组件来控制或实现是不可行的。当系统中发生了一个活动时，该进程将向整个系统发送一个事件，以此指示发生了一个活动（一个事件）。这个事件接下来可能会启动（kick off）其他进程，这些进程又可以依次启动附加的进程，所有进程都会相互去耦。

在 EDA 的一些优秀范例中，包括了保险领域和养老金固定收益领域。这些行业领域都是由发生在该系统中的事件来驱动的。像修改您的地址这样一些简单的事情，也会影响到保险领域的许多方面，其中包括政策、报价及消费者记录等。在这个例子中，保险应用程序中的驱动事件就是修改地址。然而，修改地址模块并无责任去知道该事件所导致的一切后果。因此，该修改地址模块仅仅发送一条事件消息，让系统知道修改了某个地址。报价系统将获知这个事件，并对提供给该消费者的所有未执行报价进行调整。同时，政策系统将获知这个地址修改事件，并调整该消费者的费率和适用政策。

EDA 的另一个例子是在养老金固定收益领域。结婚或变换工作会触发系统中的某个事件，它使您有资格对自己的健康和退休金信息进行某些修改。许多系统会使用 EDA，以此来避免使用一个庞大、复杂而且难以维护的中央处理引擎来控制与特定的“合格事件”有关的所有活动。

消息传送机制是基于事件驱动体系结构系统的基础。通常来说，事件一般是以空负载（empty payload）消息的模式来实现的，这些消息会在消息头中包含和事件有关的一些信息，尽管某些消息会将应用程序数据作为事件的一部分进行传送。毫不足奇的是，基于 EDA 的体系结构大多使用发布/订阅模型，作为在系统内部广播事件的手段。

1.5.3 异构平台集成
Heterogeneous Platform Integration

大多数公司，经由合并、收购、移植或错误决策的诸多因素组合，最终会拥有种类繁多的异构业务支撑平台、产品和语言。显然，集成这些平台是一项挑战性极强的任务，特别是相关标准也在持续不断地发展和演变之中。消息传送机制在这些异构平台相互通信之中起到了关键作用，无论这些平台是 Java EE 或 Microsoft .NET、Java EE 和 CICS，还是 Java EE 和 Tuxedo C++等。

尽管像 Java 这样的平台可以使用 JMS API，而其他平台，比如.NET 或 C++却都无法使用（其原因显而易见）。许多商业消息传送系统厂商和开源消息传送系统厂商都会支持 JMS API 和一个本机 API。这些提供者一般都有一个内置的消息传送桥，它允许提供者将一条

JMS 消息转换为内部消息，反之亦然。某些平台，比如说.NET，可能会需要一个外部消息传送桥，将一条 JMS 消息转换为 MSMQ 消息（这取决于您所使用的消息提供者）。例如，ActiveMQ 就提供了一个消息传送桥，用于将 MSMQ 消息转换为 JMS 格式（反之亦然）。这种低级的平台集成已经引发了范围更广的集成热潮，比如著名的企业应用集成（Enterprise Application Integration）等。

1.5.4 企业应用集成
Enterprise Application Integration

大多数成熟的组织都同时拥有遗留（legacy）应用系统和新的应用系统，这些系统都是独立实现的，而且无法实现互操作。很多时候，各个组织都会有将这些应用系统集成起来的强烈需求，以便它们能够在大规模企业运行中共享信息并实现协作。这些应用系统的集成通常称为企业应用集成（Enterprise Application Integration，EAI）。

虽然 EAI 使用了大量的厂商提供的和自己开发的解决方案，但是，企业消息传送系统仍然是大多数解决方案的主流。企业消息传送系统允许烟囱式（stovepipe）应用系统（由异构产品、技术与组件等组成）和事件进行通信并交换数据，同时还保持物理上的独立。数据和事件可以通过主题或队列以消息的形式进行交换，它们提供了可以实现各个参与应用系统去耦的一个抽象。

举例来说，一个消息传送系统可能会被用于实现一个因特网订单处理系统和像 SAP 这样的企业资源规划（Enterprise Resource Planning，ERP）系统的集成。该因特网系统使用 JMS 向一个主题传送有关新订单的业务数据。而一个 ERP 网关应用程序，它通过本机 API 访问一个 SAP 应用程序，能够订阅该订单主题。当新订单广播到该主题时，网关就会接收到这个订单，并将它们纳入 SAP 应用程序的处理之中。

1.5.5 企业到企业
Business-to-Business

历史上，企业曾使用电子数据交换（Electronic Data Interchange，EDI）系统来交换数据。数据严格使用固定的格式，通过专用增值网络（Value-Added Network，VAN）来实现交换。这种接入的成本很高，而且数据通常是以批处理的方式，而不是以实时业务事件的方式来进行交换的。

因特网、XML 和现代消息传送系统已经从根本上改变了在如今称为企业到企业（Business-to-Business，B2B）的系统中，如何进行业务数据交换和交互的状况。使用消息传送系统是现代 B2B 解决方案的主流趋势，因为它允许各个组织相互协作，而无须将它们的业务系统紧密集成起来。此外，它还降低了接入门槛，因为细粒度（finer-grained）的参与已经成为可能。根据和企业相结合的队列和主题的不同，它们既可以加入到 B2B 之中，也可以自由退出。

举个例子，一个制造商能够建立一个关于原料招标广播请求的主题。各家供应商都可以订阅该主题，并通过向该制造商队列生产消息而实现响应。制造商可以根据意愿添加和删除供应商，而且，关于各类存货和原料的新主题和队列，也可以用以正确区分各个系统。

1.5.6 地理分散
Geographic Dispersion

如今，很多公司在地理上都是分散的。所有的砖墙加灰泥式的（brick-and-mortar）传统实体公司、鼠标加灰泥式的（click-and-mortar）新型公司及时兴的网络化（dot-coms）公司都面临着和企业系统地理分散有关的诸多问题。比如，远程仓库中的存货系统需要和位于公司总部的集中式内部 ERP 系统进行通信；由各个子公司本地管理的敏感雇员数据需要和总公司实现同步等。JMS 消息传送系统能够确保地理上分散的业务数据交换的安全性和可靠性。

1.5.7 信息广播
Information Broadcasting

拍卖网站、股票报价服务和证券交易，所有这些应用都必须将数据以一对多的方式推送给堪称海量的接收者。在许多情况下，广播信息都需要各个接收者逐一选择路由和过滤。当输出信息要以一对多的方式进行传送时，经常要将对这种信息的响应发回给广播者。这是企业消息传送非常适用的另一种情况，因为发布/订阅模型可以用于发布消息，而 p2p 模型则可以用于响应。

在这些情况下，传送可靠性的选择成了关键因素。举例来说，在广播股票报价时绝对地保证信息传送，这可能并不是最重要的，因为同一股票代码很可能会在较短的时间间隔内进行另一次广播。然而，在交易者通过购买订单对报价做出响应的情况下，以保证的（guaranteed）方式返回响应是至关重要的。此时，您需要综合利用消息传送机制的可靠性，因为发布/订阅模型分发速度很快但并不可靠，而使用 p2p 模型从交易者处购买订单则是非常可靠的。JMS 和企业消息传送为发布/订阅和 p2p 这两种模型都提供了不同程度的可靠性。

1.5.8 构建动态系统
Building Dynamic Systems

在 JMS 中，发布/订阅主题和 p2p 队列是集中管理的，并都称为 JMS **受管对象**。您的应用程序和另一个应用程序通信时，并不用知道各个主题或队列的网络位置；它仅仅将主题和队列对象作为标识符使用而已。使用主题和队列为 JMS 应用程序提供了一定程度的位置透明性和灵活性，这使得在一个企业系统中添加和删除参与者成为可能。

举例来说，一个系统管理员能够在按需添加的基础上，动态地向特定主题添加订阅者。一个常见的场景可能会是：您是否发现需要为特定的消息而不是其他消息添加一个审计跟踪（audit-trail）机制。图 1-8 显示了您如何只通过订阅您所感兴趣的主题，来插入一个专用的审计和日志记录 JMS 客户端。这个客户端唯一的工作就是跟踪特定的消息。

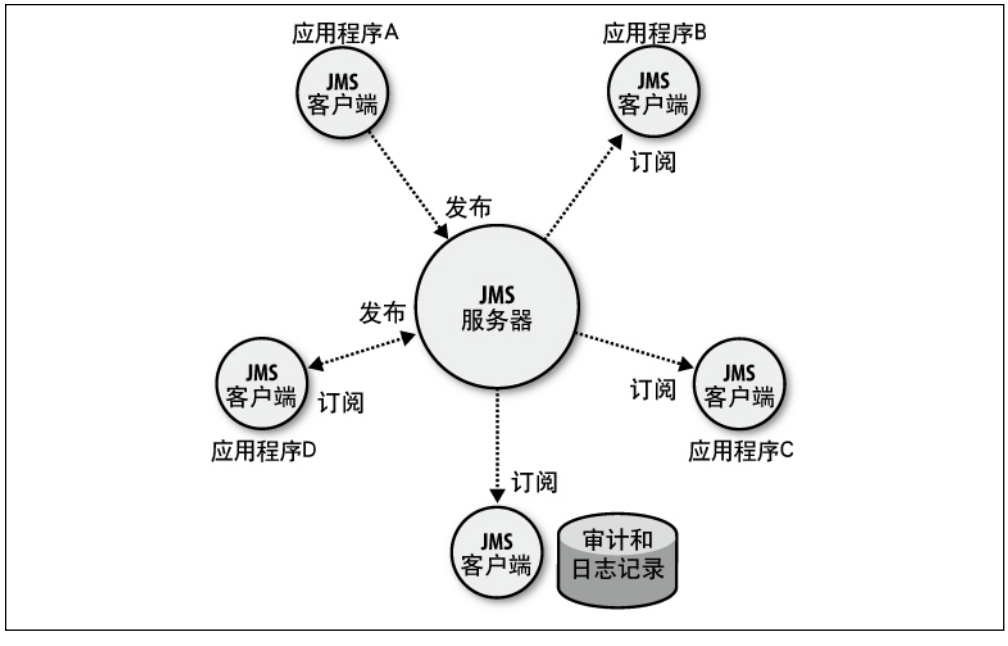


图 1-8：使用发布/订阅动态添加审计和日志记录

企业系统有了添加和删除生产者、消费者的能力，就可以在一个已经部署好的环境中，动态改变消息路由并重新选择路由。

我们也可以在前面讨论的 EAI 场景上来构建另外一个例子。在这个例子中，一个网关接收输入的采购订单，将它们转换为适用于遗留 ERP 系统的格式，并调入 ERP 系统进行处理（参见图 1-9）。

在图 1-8 中，其他 JMS 应用程序（A 和 B）还订阅了采购订单主题，并独立进行处理。应用程序 A 可能是公司内的一个遗留应用程序，而应用程序 B 则可能是另一家公司的业务系统，以此来表示 B2B 集成。

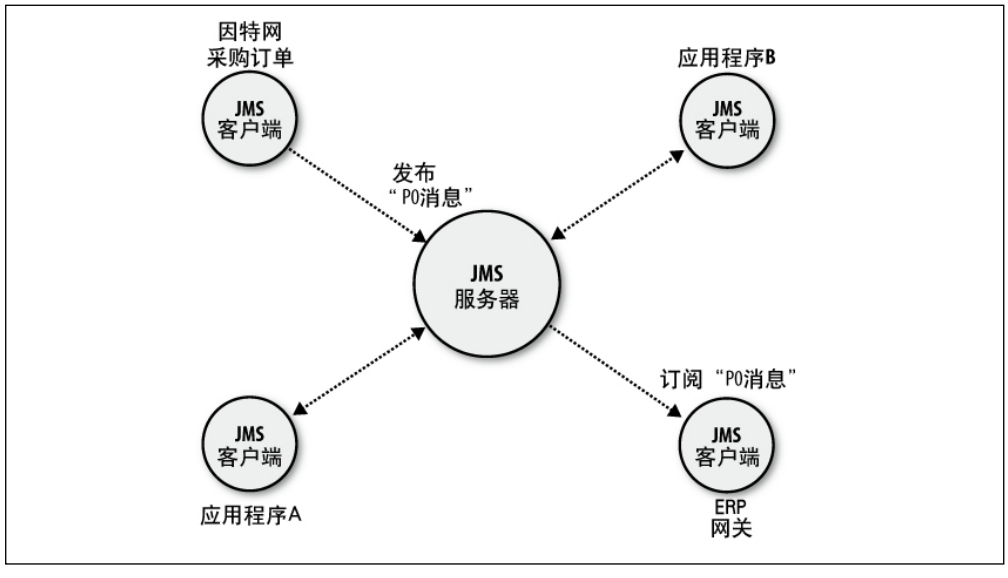


图 1-9：将一个采购订单系统和一个 ERP 系统相集成

使用 JMS 从这个流程中添加和删除应用程序，这是非常容易的。例如，如果需要处理来源不同的采购订单，比如一个来自基于因特网的系统，而另一个则来自遗留的 EDI 系统，只需要将遗留采购订单系统加入混合系统之中即可（参见图 1-10）。

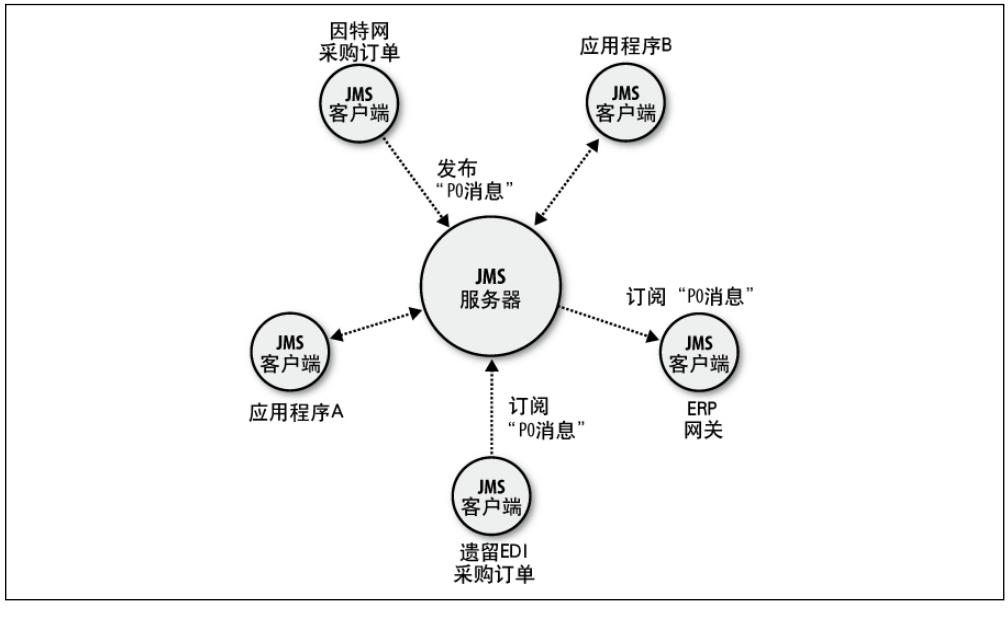


图 1-10：将两个不同的采购订单系统和一个 ERP 系统相集成

这个例子有意思之处就在于，ERP 网关并未意识到，它是从两个完全不同的来源接收到采购订单消息。遗留 EDI 系统可以是一个比较旧的内部系统，或者是一个业务伙伴或新近并购的一家子公司的主系统。此外，还可以动态地将遗留 EDI 系统添加进去，而不需要关机和重组整个系统。企业消息传送系统使得这种灵活性成为可能，同时，JMS 还允许 Java 客户端使用相同的 Java 编程模型来访问多个不同的消息传送系统。

1.6 RPC 和异步消息传送
RPC Versus Asynchronous Messaging

RPC（Remote Procedure Call，**远程过程调用**）是通常用于描述分布式计算模型的术语，现在 Java 和.NET 这两种平台都在使用这个术语。基于组件的体系结构，比如企业级 JavaBean(Enterprise JavaBeans，EJB)，就是建立在这个模型基础之上的。对于许多应用程序来说，基于 RPC 的技术已经是，并且将继续是切实可行的解决方案。不过，企业消息传送模型在特定类型的分布式应用程序中表现更为出色。在本节中，我们将讨论每种模型的优缺点。

1.6.1 紧密耦合的 RPC
Tightly Coupled RPC

紧密耦合的 RPC 模型最为成功的一个领域就是构建 3 层或 n 层应用程序。在这个模型中，表示层（第 1 层）使用 RPC 和中间层（第 2 层）的业务逻辑进行通信，访问位于后端（第 3 层）的数据。Sun Microsystems 公司的 J2EE 平台和 Microsoft 公司的.NET 平台是这种体系结构最为先进的范例。

使用 J2EE、JSP 和 servlet 描述的是表示层，而企业级 JavaBean(EJB）则是中间层。抛开平台不论，这些系统使用的核心技术是基于成为定义通信范例的 RPC 的中间件。

RPC 试图模仿在一个进程中运行的某个系统的行为。在调用一个远程过程时，调用者将被阻塞，直到该过程完成并将控制权返回给调用者。从开发者的角度看，这种同步模型使得该系统就好像运行在一个进程当中。这些工作会依次完成，同时确保以预定顺序完成。RPC 同步的本质特性，将客户端（进行调用的软件）和服务器（为该调用服务的软件）二者紧密耦合在一起。因为客户端已被阻塞，所以它无法继续进行工作，直到服务器做出响应为止。

RPC 紧密耦合的本质特性导致出现了相互高度依赖的系统，其中一个系统的失效会对其他系统产生立竿见影的弱化影响。例如，在 J2EE 中，如果期望使用企业级 bean 的 servlet 顺利工作，EJB 服务器就必须正常地发挥功能。

虽然 RPC 在许多场景中表现优秀，但是在系统对系统（system-to-system）的处理过程当中，它的同步、紧密耦合等本质特性却是一个严重的缺陷，因为“系统对系统”有很多垂直的应用程序集成在一起。在系统对系统场景中，垂直系统之间的通信线路不仅数量众多，而且方向也是错综复杂的，如图 1-11 所示。

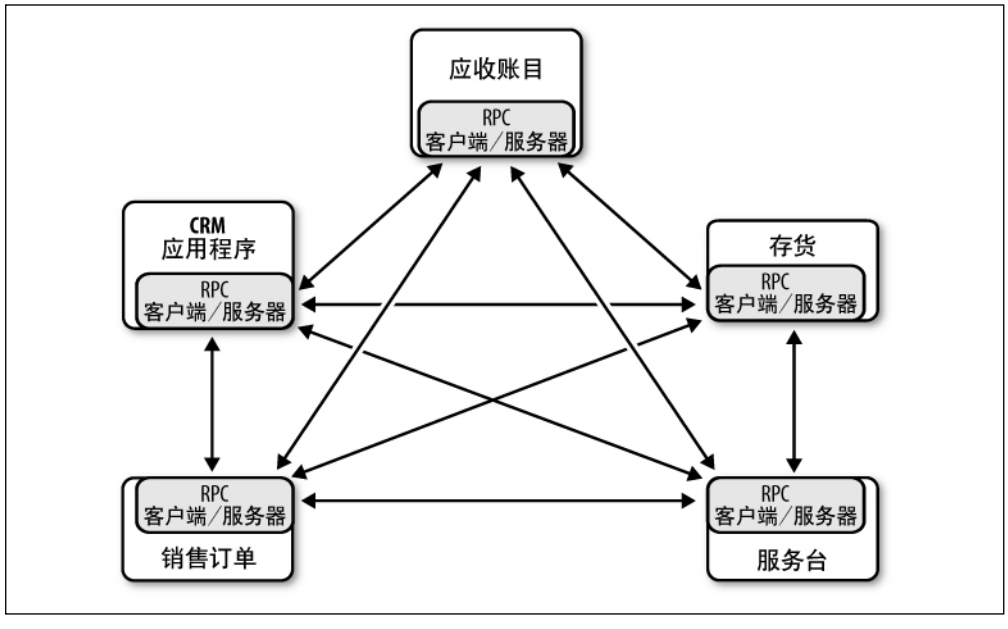


图 1-11：与同步 RPC 紧密耦合

让我们设想一下使用紧密耦合的 RPC 机制实现这种基础设施所面临的挑战。这些系统之间的连接管理是多对多（many-to-many）的问题。当您向混合系统中加入另一个应用程序时，您不得不回过头来让其余所有的系统都知道它，而且，这些系统也会崩溃（crash）。它们仍然需要预定停工时间（scheduled downtime），而且对象的接口也需要升级。

当该系统的一部分中断运行时，一切都得停止。当您向一个订单输入系统添加订单时，它要对其他系统逐个进行同步调用。这会导致订单输入系统发生阻塞，并一直等待，直到每个系统都处理完该订单时为止。（注 1）

正是 RPC 系统的同步、紧密耦合、相互依赖等本质特性，使得子系统中出现的故障最终会导致整个系统的失效。就像在“系统对系统”场景中那样，当 RPC 紧密耦合的本质特性不再适用时，消息传送机制为此提供了另一种选择方案。

注 1：像 CORBA 单向调用这种多线程的、松散的 RPC 机制也是一种选择，不过，这些解决方案自身非常复杂，而且它们还需要成熟完善的开发。当没有“明智地”使用线程时，它们的开销很大，而且在出现故障的情况下，CORBA 单向调用仍然需要进行应用程序级（application-level）错误处理。

1.6.2 企业消息传送
Enterprise Messaging

各个子系统在可用性方面存在的问题，并不是使用面向消息的中间件所带来的后果。消息传送机制的一个基本思想就是：规定应用程序之间的通信应该采用异步方式。将各部分连接在一起的代码会假定这是一条**单向**消息，它不需要立即从另一个应用程序那里得到响应。换句话说，也就是没有出现阻塞现象。一旦一条消息被发出，消息传送客户端就能够转向其他任务；它不必等待对这条消息的响应。这是 RPC 和异步消息传送之间的主要区别，而且，它对于理解消息传送系统的优点来说至关重要。

在一个异步消息传送系统当中，每个子系统（收款、存货等）都不存在和其他系统的耦合（参见图 1-12）。它们通过消息传送服务器进行通信，因此，某个子系统出现故障，并不会妨碍其他子系统的运行。

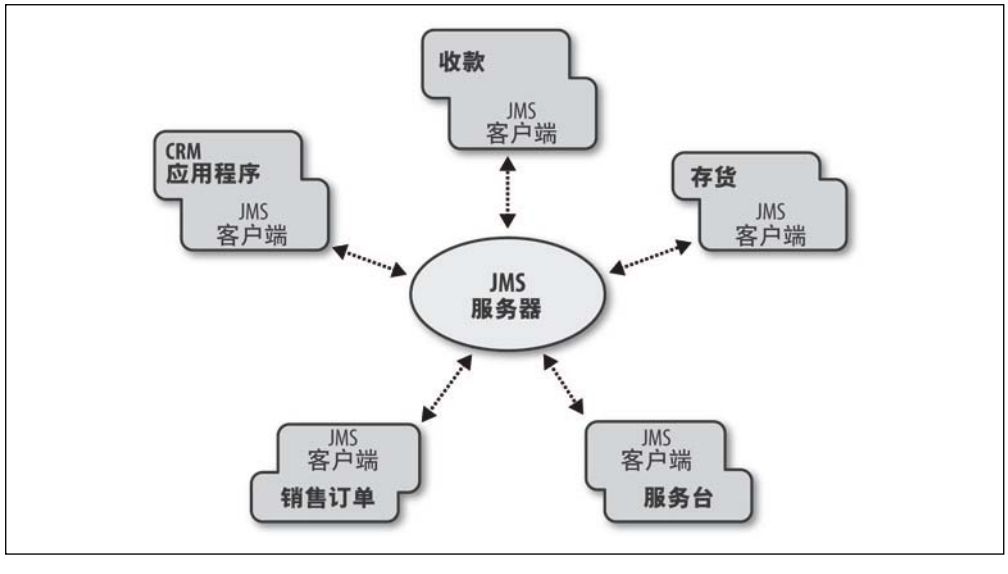


图 1-12：JMS 提供了一个松散耦合的环境，其中，系统组件的局部故障并不会妨碍系统的整体可用性

在网络化系统中会出现局部故障，这是一个不可避免的事实。其中的一个系统，可能会在其连续运行期间的某个时刻，发生不可预测的故障，或者需要停机。这种现象可能会由于内部系统和合作系统地理上的分散而被进一步放大。考虑到这个因素，JMS 提供了**保证传送**（guaranteed delivery）方式，它可以确保即便发生了局部故障，预定消费者最终也会接收到这条消息。

保证传送使用的是一种“**保存并转发**（store-and-forward）”的机制，这就意味着，如果预定消费者当前并不可用，底层消息服务器就会将输入的消息写到一个持久存储器（persistent store）之中。随后，当该接收应用程序变为可用时，“保存并转发”机制会把预定消费者在不可用时错过的所有消息传送给它们（参见图 1-13）。

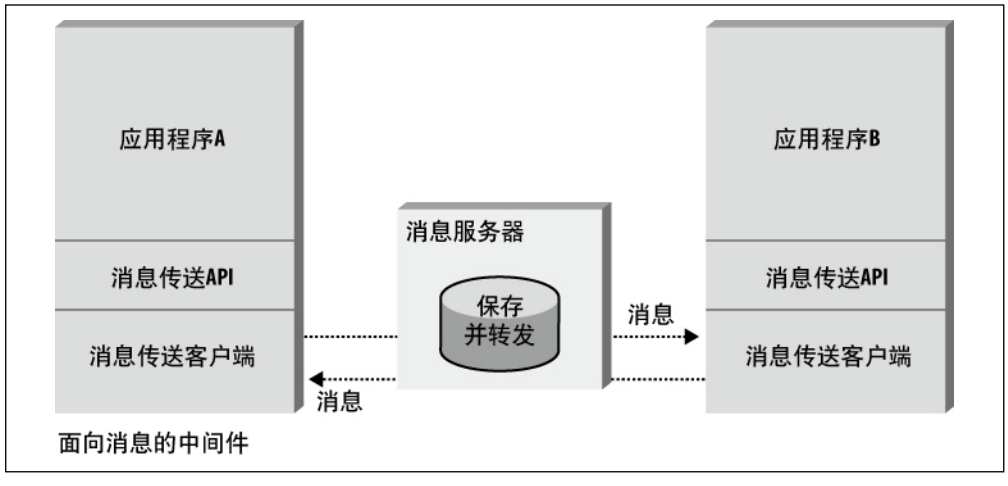


图 1-13：底层的“保存并转发”机制保证了消息的传送

概括来说，JMS 不仅仅是另外一种事件服务。它的设计涵盖了范围极广的企业应用程序，包括 EAI、B2B 和推送模型等。通过异步处理、“保存并转发”及“保证传送”机制，它为保持业务应用程序连续运行并实现不间断服务提供了很高的可用性。它还通过发布/订阅功能和点对点功能，提供了集成灵活性。通过位置透明和管理控制，它提供了一种健壮的、基于服务的体系结构。而且，最重要的是，它非常易于学习和使用。下一章，我们将通过构建第一个 JMS 应用程序，来看一看它是多么地简单。

第 2 章

编写一个简单的示例程序
Developing a Simple Example

前面已经了解了面向消息的中间件和 JMS 的一些概念，现在就可以编写第一个 JMS 应用程序了。尽管 通过应用更为广泛的点对点模型来阐释一个简单的例子，可能会更容易一些，不过，发布/订阅模型却提供了一个更令人感兴趣的例子。因此，本章将使用发布/订阅消息传送模型，对 JMS 做一个简要的介绍。您会在实践中了解 JMS，并学习到一些基本的类与接口。第 4 章将通过一个实际的（real-world）例子来详细讨论点对点模型，随后第 5 章将利用同一示例来分析发布/订阅消息传送模型。

与本书中所有例子一样，可以通过 O'Reilly 公司的网址 *http://oreilly.com/catalog/9780596522049/examples* 来下载一些特定厂商的示例代码和使用说明。安装 并配置 JMS 提供者需要参考有关厂商提供的使用说明。为了说明一家示例厂商的配置情况，我们将使用一个常用的开源 JMS 提供者：ActiveMQ 5.2 版（参见 *http://activemq.apache.org*），它不仅健壮（robust），而且质量很高（production-quality）。在本书附录 D 中还可以找到执行示例代码所需的基本安装使用说明和配置设置。

2.1 聊天应用程序
The Chat Application

因特网聊天为学习 JMS 发布/订阅消息传送模型提供了一种有趣的应用。由于基于 web 的聊天应用程序主要用于娱乐，所以可以在数以千计的网站上看到它的踪影。在这类应用中，人们可以加入虚拟聊天室来和其他人“聊天”。

为了说明 JMS 是如何工作的，我们将使用 JMS 发布/订阅 API 来构建一个简单的聊天应用程序。因特网聊天的需求可以完美地映射到发布/订阅消息传送模型之上。在这个模型中，生产者可以通过将消息传送给单个主题，实现向多个消费者发送一条消息。消息生产者还可以称为**发布者**，同时，消息消费者也可以称为**订阅者**。

下面是一个基于 JMS 的聊天客户端的完整源代码清单。在一个聊天会话中，每个聊天参与者都使用这个 Chat 程序，加入一个特定的聊天室（主题）之中，并向该聊天室传送消息，以及从该聊天室中接收消息。本章将深入剖析这个示例，并讲解此源代码清单中使用的各种 API 调用。有关点对点模型和发布/订阅模型的更多细节，将在第 4 章和第 5 章中详细阐述。

```
package ch02.chat;

import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class Chat implements javax.jms.MessageListener {
    private TopicSession pubSession;
    private TopicPublisher publisher;
    private TopicConnection connection;
    private String username;

    /*用于初始化 Chat （聊天）的构造函数*/
    public Chat(String topicFactory, String topicName,String username)
        throws Exception {

        //使用 jndi.properties 文件获得一个 JNDI 连接
        InitialContext ctx = new InitialContext();

        //查找一个 JMS 连接工厂并创建连接
        TopicConnectionFactory conFactory =
             (topicConnectionFactory)ctx.lookup(topicFactory);
        TopicConnection connection = conFactory.createTopicConnection();

        //创建两个 JMS 会话对象
        TopicSession pubSession = connection.createTopicSession(
            false, Session.AUTO_ACKNOWLEDGE);
        TopicSession subSession = connection.createTopicSession(
            false, Session.AUTO_ACKNOWLEDGE);

        //查找一个 JMS 主题
        Topic chatTopic =(Topic)ctx.lookup(topicName);

        //创建一个 JMS 发布者和订阅者。createSubscriber 中附加的参数是一个消息
        //选择器（null）和 noLocal 标记的一个真值，它表明这个发布者生产的消息不
        //应被它自己所消费。
        TopicPublisher publisher =
            pubSession.createPublisher(chatTopic);
        TopicSubscriber subscriber =
            subSession.createSubscriber(chatTopic, null, true);

        //设置一个 JMS 消息侦听器
        subscriber.setMessageListener(this);

        //初始化 Chat 应用程序变量
        this.connection = connection;
```

```
        this.pubSession = pubSession;
        this.publisher = publisher;
        this.username = username;

        // 启动 JMS 连接；允许传送消息
        connection.start();
    }

    /*接收来自 TopicSubscriber 的消息*/
    public void onMessage(Message message){
        try {
            TextMessage TextMessage =(textMessage)message;
            System.out.println(textMessage.getText());

        } catch（JMSException jmse){ jmse.printStackTrace(); }
    }

    /*使用发布者创建并发送消息*/
    protected void writeMessage(String text)throws JMSException {
        TextMessage message = pubSession.createTextMessage();
        message.setText(username+"："+text);
        publisher.publish(message);


}

    /*关闭 JMS 连接*/
    public void close()throws JMSException {
        connection.close();
    }

    /*运行聊天客户端*/
    public static void main(String [] args){
        try {
            if(args.length!=3)
                System.out.println("Factory,Topic,or username missing");

            //args[0]=topicFactory；args[1]=topicName；args[2]=username
            Chat chat = new Chat(args[0],args[1],args[2]);

            //从命令行读取
            BufferedReader commandLine = new
              java.io.BufferedReader（new InputStreamReader(System.in));

        //循环，直到键入“exit”为止
        while(true){
           String s = commandLine.readLine();
           if(S.equalsIgnoreCase ("exit")){
              chat.close();
              System.exit(0);
            } else
              chat.writeMessage(S);
        }
    } catch（Exception e){ e.printStackTrace(); }
  }
}
```

请注意：刚才给出的代码使用的是 createSubscriber()方法，它采用 3 个参数而不仅仅是一个参数。这样就可以设置 noLocal 标记（第 3 个参数），使得这个类发布的消息不会被该类自己所消费。第 2 参数用于一个消息选择器。由于并不对主题做任何过滤，这个值就被设置为 null。如果使用单个参数的方法调用来创建订阅者，就会在控制台显示器上看到发送的消息。

2.1.1 从 Chat 示例开始
Getting Started with the Chat Example

如果要运行 Chat 应用程序，就需要使用支持 JNDI 和 JMS 1.1 的 JMS 提供者。为了说明示例代码中的某些细节和配置情况，我们将使用一种流行的开源 JMS 提供者：ActiveMQ。您需要查阅 JMS 厂商的文档，来了解 Chat 应用程序的 TopicConnectionFactory 和 Topic 的相关配置信息。在我们的示例中，已经将它们分别命名为 TopicCF 和 topic1。例如，使用 ActiveMQ，可以在类路径中创建一个 jndi.properties 文件，为 Chat Application（聊天应用程序）设置 TopicConnectionFactory 名称和一个 Topic（如下所示）。

```
java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
java.naming.provider.url = tcp://localhost:61616
java.naming.security.principal=system
java.naming.security.credentials=manager

connectionFactoryNames = TopicCF
topic.topic1 = jms.topic1
```

jndi.properties 文件还包含了 JMS 提供者的 JNDI 连接信息。您需要设置连接到 JMS 服务器所需要的初始上下文工厂类、提供者 URL、用户名及密码等。每家厂商都将拥有一个不同的上下文工厂类和 URL 名称，用于连接到服务器。要获得这些值，需要查阅特定的 JMS 提供者或 Java EE 容器的相关文档。例如，在刚才所示的 jndi.properties 文件中，对于 ActiveMQ，应将初始上下文工厂设置为 org.apache.activemq.jndi. ActiveMQInitial-ContextFactory，并将 提供者 URL 设置为 tcp://localhost:61616（ActiveMQ 的默认协议、主机和端口）。附录 D 中附有 ActiveMQ 安装和配置的更多细节。

在配置并启动 JMS 服务器之后，您就要编译 Chat 应用程序。除了 jms-11.jar 文件之外，还须将 JMS 提供者需要的所有 JAR 文件包含进类路径之中（对于 ActiveMQ 5.2，只须在类路径中包含 activemq-all-5.2.0.jar 文件）。

Chat 类包括一个 main()方法，因此它可以作为一个独立的 Java 应用程序来运行。显然，您会打开多个命令窗口，以至于能够模拟和多人同时聊天。Chat 类可以通过命令行或一个 shell 脚本执行：

```
java ch02.chat.Chat topicConnectionFactory topicName username
```

例如，在前面列出的 OpenJMS 配置中，我们已经定义了一个名为 `TopicCF` 的主题连接工厂（Topic Connection Factory），以及名为 `Topic1` 的一个 Topic（主题）。因此，要为一个名为 `Fred` 的用户和另一个名为 `Wilma` 的用户执行 Chat 应用程序，要使用下面的命令：

```
java ch02.chat.Chat TopicCF topic1 Fred
java ch02.chat.Chat TopicCF topic1 Wilma
```

在单独的命令窗口中，至少运行两个聊天客户端，并尝试在其中一个客户端中输入，您会看到键入的文本显示在了另一个客户端上。

在分析详细源代码之前，让我们快速浏览一下该代码的功能，可能会有一些帮助。聊天客户端为一个特定的主题创建一个 JMS 发布者和订阅者。该主题就代表了聊天室。JMS 服务器注册了所有想要发布或订阅一个特定主题的 JMS 客户端。在一个聊天客户端的命令行中输入文本时，它会发布给消息传送服务器。消息传送服务器会识别出和该发布者有关的主题，并将消息传送给已经订阅该主题的所有 JMS 客户端。如图 2-1 所示，由任何一个 JMS 客户端发布的消息，都会被传送给该主题的所有 JMS 订阅者。

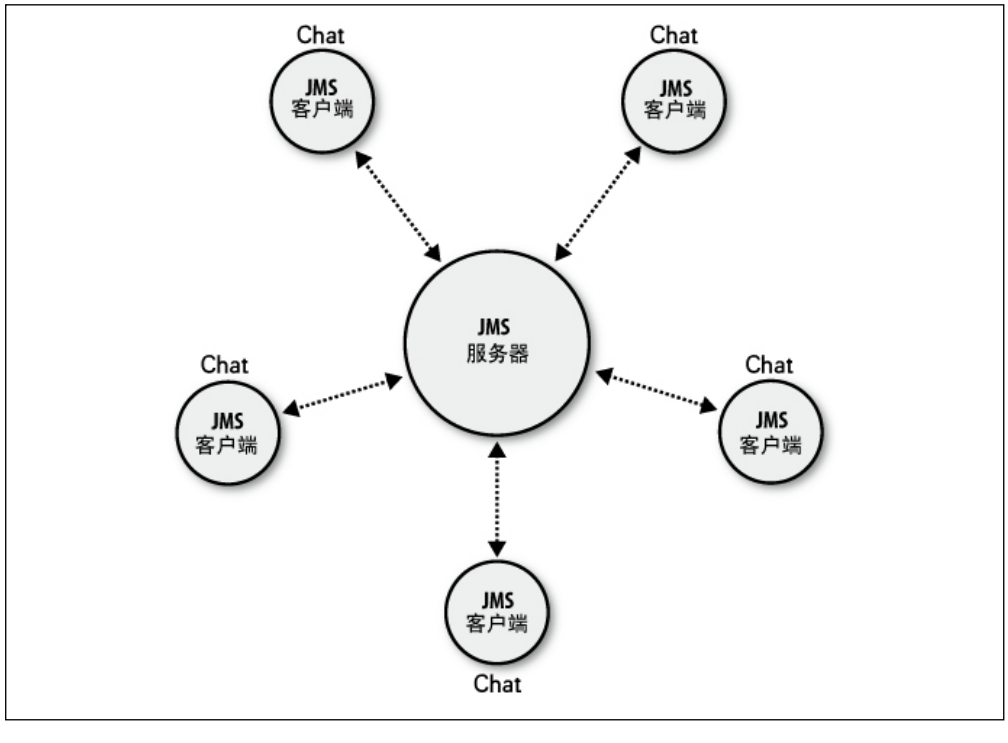


图 2-1：Chat 应用程序

2.1.2 分析源代码

Examining the Source Code

在一对命令窗口中运行 Chat 示例程序，来说明 Chat 应用程序的功能。本章的其余部分将分析 Chat 应用程序的源代码，因此，您可以看到 Chat 应用程序是**如何**工作的。

引导 JMS 客户端

main()方法将引导（bootstrap）聊天客户端，并提供一个命令行接口。一旦创建了一个 Chat 类实例，main()方法就将剩余时间用于读取从命令行键入的文本，并用该实例的 writeMessage()方法将该文本传送到 Chat 实例之中。

Chat 实例连接到主题之上，并接收和传送消息。Chat 实例在构造函数中启动它的生命周期，构造函数会完成连接到主题所需的全部工作，并建立用于传送和接收消息的 TopicPublisher 和 TopicSubscribers 主题。

获得一个 JNDI 连接

获得一个到 JMS 消息传送服务器的 JNDI 连接，聊天客户端就开始启动。JNDI 是一个与具体实现无关的（implementation-independent）API，用于目录和命名服务系统。JMS 客户端可以使用一个目录服务来访问 ConnectionFactory 和 Destination（主题和队列）对象。ConnectionFactory 和 Destination 对象是绝无仅有的使用 JMS API 无法获得的对象，在这一点上，它和连接、会话、生产者、消费者及消息不同，后者是在 JMS API 内部使用工厂模式生产的。JNDI 为获得 ConnectionFactory 和 Destination 对象提供了一种方便、位置透明、可配置并且可移植的机制，这些对象也称为 JMS 受管（JMS-administered）对象，因为它们是由一个系统管理员建立和配置的。

使用 JNDI，JMS 客户端就可以通过首先查找一个 ConnectionFactory 来实现对一个 JMS 提供者的访问。ConnectionFactory 用于创建 JMS 连接，而该连接随后就可以用于发送和接收消息。也可以通过 JNDI 来获得 Destination 对象，也就是 JMS 当中的虚拟通道（主题和队列），并由 JMS 客户端使用。目录服务还可以由系统管理员来配置，用以提供 JMS 受管（JMS-administered）对象，如此一来，JMS 客户端就无须使用专有代码来访问 JMS 提供者了。

JMS 服务器既可以和一个独立的目录服务（例如 LDAP）结合使用，也可以单独提供支持 JNDI API 的目录服务。有关 JNDI 的更多细节，请参见第 31 页的“解读 JNDI”。

Chat 类的构造函数通过连接 JMS 服务器使用的 JNDI 命名服务开始启动：

```
//获得一个 JNDI 连接使用 jndi.properties 文件
InitialContext ctx = new InitialContext();
```

解读 JNDI

JNDI 是一个标准的 Java 扩展，它提供了一个统一的 API，用于访问多种目录和命名服务。在这方面，它和 JDBC 有点类似。使用 JDBC 编写的代码能够访问不同的关系数据库，比如说 Oracle、SQLSERVER 或 Sybase 等；而使用 JNDI 编写的代码，则能够访问不同的目录和命名服务，比如说 LDAP、NDS、CORBA 命名服务，以及 JMS 服务器提供的私有命名服务。

在 JMS 中，JNDI 主要用于命名服务来定位受管对象。受管对象就是由系统管理员创建和配置的 JMS 对象。受管对象包括 JMS `ConnectionFactory` 和 `Destination` 对象，比如说主题和队列等。

受管对象和命名服务中的一个名称相互绑定在一起。一个命名服务将名称和分布式对象、文件和设施关联起来，以便它们可以使用简单的名称而不是密码似的网络地址实现在网络上的定位。DNS 就是命名服务的一个例子，它将像 *www.oreilly.com* 这样的因特网主机名转换为一个网络地址，浏览器则会使用该地址连接到 Web 服务器。还有许多其他的命名服务，比如 CORBA 中的 CosNaming 和 Java RMI 注册（registry）等。命名服务允许打印机、分布式对象、JMS 受管对象和名称相互绑定，并按照和文件系统相类似的层次进行组织。目录服务是一种更为复杂的命名服务类型。

JNDI 提供一个隐藏了命名服务细节的抽象，使得客户端应用程序的可移植性更高。使用 JNDI，JMS 客户端能够浏览一个命名服务，并在不知道命名服务细节或它如何实现的情况下，引用受管对象。通常，JMS 服务器会和一个标准 JNDI 驱动器（也称为服务提供者），以及类似于轻量级目录访问协议（Lightweight Directory Access Protocol，LDAP）这样的目录服务结合使用，或者提供一个私有的 JNDI 服务提供者和目录服务。

JNDI 既是虚拟的，又是动态的。说它是虚拟的，是因为它允许一个命名服务和另一个命名服务相互连接。您可以使用 JNDI，通过文件、打印机、JMS 受管对象的目录，以及命名服务虚拟连接之后的其他资源目录，实现向下钻取（drill down）。用户并不知道或并不关心该目录的实际位置。作为一名管理员，您可以创建跨多个物理位置的多种不同服务的虚拟目录。

说 JNDI 是动态的，是因为它允许特定类型的命名和目录服务所使用的 JNDI 驱动器在运行时动态加载。驱动器将一个特定类型的命名服务或目录服务映射成标准的 JNDI 类接口。用于 LDAP、Novell NetWare NDS、Sun Solaris NIS+、CORBA CosNaming，以及许多其他类型的命名和目录服务（包括私有服务）的各个驱动器都已经创建完毕。动态地加载 JNDI 驱动器（服务提供者），使得一个客户端在预先不知道可能会发现哪种服务的情况下，实现跨任意目录服务导航。

要创建一个到 JNDI 命名服务的连接，就需要先创建一个 javax.naming.InitialContext 对象。一个 InitialContext 就是所有 JNDI 查找的起始点，它和文件系统根目录的概念很相似。InitialContext 提供了一个到目录服务的网络连接，这个目录服务就发挥访问 JMS 受管对象的根目录的作用。用于创建 InitialContext 的属性取决于所用的 JMS 目录服务类型。可以使用 Properties Object(属性对象)直接在源代码中配置 InitialContext 属性，或者使用位于应用程序类路径中的一个外部 jndi.properties 文件。我们在例子中使用的是 ActiveMQ，jndi.properties 文件看起来就像是：

```
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory
java.naming.provider.url = tcp://localhost:61616
java.naming.security.principal=system
java.naming.security.credentials=manager
...
```

使用 Properties 对象的相应源代码如下：

```
Properties env = new Properties();
env.put(context.SECURITY_PRINCIPAL,"system");
env.put(context.SECURITY_CREDENTIALS,"manager");
env.put(context.INITIAL_CONTEXT_FACTORY,
  "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
env.put(context.PROVIDER_URL,"tcp://localhost:61616");

InitialContext ctx = new InitialContext(env);
```

TopicConnectionFactory

一旦实例化一个 JNDI InitialContext 对象，就可以使用它在消息传送服务器的命名服务中查找 TopicConnectionFactory：

```
TopicConnectionFactory conFactory =
  (topicConnectionFactory)ctx.lookup(topicFactory);
```

javax.jms.TopicConnectionFactory 用于创建到一个消息服务器的连接。一个 TopicConnectionFactory 就是一类受管对象，这就意味着它的属性和行为要由负责消息传送服务器的系统管理员来配置。由于每家厂商对 TopicConnectionFactory 的实现不同，因此可供系统管理员使用的配置选项也会因产品而异。举例来说，一个连接工厂可能会被配置成为创建使用特定协议、安全模式（scheme）、集群策略等的连接。一个系统管理员可能会选择部署几个不同的 TopicConnectionFactory 对象，每个对象都使用它自己的 JNDI 查找名来配置。

TopicConnectionFactory 提供了两个重载版本的 createTopicConnection()方法：

```
package javax.jms;

public interface TopicConnectionFactory extends ConnectionFactory {
```

```
public TopicConnection createTopicConnection()
   throws JMSException, JMSSecurityException;
public TopicConnection createTopicConnection(String username,
   String password) throws JMSException, JMSSecurityException;
}
```

这些方法用于创建TopicConnection对象。无参数（no-arg）方法的行为取决于JMS提供者。一些JMS提供者假定JMS客户端在匿名安全性上下文之内连接，而其他提供者则可能假定凭证（credential）从JNDI或当前的线程获得 （注 1）。第 2 种方法为客户端提供了一个用户名/密码认证凭证，用于连接认证。我们在代码中使用的是无参数方法，在创建连接时，该方法将使用默认用户身份。

The TopicConnection

TopicConnection 由 TopicConnectionFactory 所创建：

```
//查找一个 JMS 连接工厂并创建连接
TopicConnectionFactory conFactory =
    (topicConnectionFactory)ctx.lookup(topicFactory);
TopicConnection connection = conFactory.createTopicConnection();
```

TopicConnection表示和消息服务器的一个连接。从TopicConnectionFactory中创建的每个TopicConnection就是和该服务器的唯一连接 （注 2）。一个JMS客户端可能会选择从同一连接工厂创建多个连接，但是这种情况并不常见，因为这些连接的开销相对较大（每个连接都需要一个网络套接字、I/O流和内存等）。一般认为从同一连接中创建多个session对象（本章后面会讨论）效率更高，因为会话共享了对同一连接的访问。TopicConnection是javax.jms.Connection接口的一个扩展接口。它定义了供TopicConnection客户端使用的几种通用方法，包括start()、stop()和close()方法：

```
public interface Connection {
    public void start()throws JMSException;
    public void stop()throws JMSException;
    public void close()throws JMSException;
    ...
}

public interface TopicConnection extends Connection {
    public TopicSession createTopicSession(boolean transacted,
                                           int acknowledgeMode)
    throws JMSException;
```

注 1：线程特定的(thread-specific)存储器将和 Java 认证与授权服务（Java Authentication and Authorization Service，JAAS）一起使用，从而保证了安全凭证在资源和应用程序之间的透明传播。

注 2：实际的物理网络连接可能唯一，也可能不是，这取决于厂商的不同。不过，连接被认为是逻辑唯一的，因此，认证和连接控制可以分别通过其他连接来管理。

```
    ...
}
```

start()、stop()和close()方法允许一个客户端直接管理连接。start()方法将到达的消息流变为“on”状态，允许这些消息被该客户端接收。这个方法被用在Chat类中构造函数的结尾：

```
//开始JMS连接；允许传送消息
connection.start();
```

在建立订阅者**之后**再启动连接，这是一个明智之举，因为start()一旦被调用，消息就能够从主题开始流入。

stop()方法阻塞了到达的消息流，直到再次调用start()方法时为止。close()方法用于关闭连接消息服务器的TopicConnection。这个方法应在客户端使用完TopicConnection后再调用；关闭连接节约了客户端和服务器的资源。在Chat类中，main()方法在命令行键入了“exit”时才调用Chat.close()。接下来，Chat.close()方法再依次调用TopicConnection.close()方法：

```
public void close()throws JMSException {
   connection.close();
}
```

关闭TopicConnection将关闭和该连接有关的所有对象，包括TopicSession、TopicPublisher和TopicSubscriber等。

TopicSession

在获得TopicConnection之后，它被用于创建TopicSession对象：

```
//创建两个JMS会话对象
TopicSession pubSession = connection.createTopicSession(
    false, Session.AUTO_ACKNOWLEDGE);

TopicSession subSession = connection.createTopicSession(
    false, Session.AUTO_ACKNOWLEDGE);
```

TopicSession对象是用于创建Message、TopicPublisher和TopicSubscriber对象的工厂。它还用作JMS内部的事务性工作单元。一个客户端可以创建多个TopicSession对象，对发布者、订阅者及其相关事务提供粒度更细的控制。在这个例子中，我们创建了两个TopicSession对象：pubSession和subSession。由于JMS中的线程限制，我们需要两个对象，本章稍后即将讨论这两个对象。

createTopicSession()方法中的boolean参数用来表明Session对象是不是事务性的。一个事务性Session自动管理着一个事务内部的输出和输入消息。虽然事务很重要，但对于这里的讨论来说，它并不是必需的，因此该参数被设置为false，这就意味着TopicSession将不是事务性的。有关事务的更多细节将在第7章中详细讨论。

第 2 个参数表明了 JMS 客户端使用的确认模式。确认就是通知消息服务器：客户端已经接收到消息。这个例子中选用 AUTO_ACKNOWLEDGE，这就意味着消息将在客户端接收之后自动确认。

TopicSession 对象用于创建 TopicPublisher 和 TopicSubscriber。TopicPublisher 和 TopicSubscriber 对象创建时带有一个 Topic 标识符，而且，这两个对象专属于创建它们的 TopicSession，它们的运行受控于特定的 TopicSession：

```
TopicPublisher publisher =
    pubSession.createPublisher(chatTopic);
TopicSubscriber subscriber =
    subSession.createSubscriber(chatTopic);
```

TopicSession 还用于创建传送给主题的 Message 对象。pubSession 用于在 writeMessage()方法中创建 Message 对象。在命令行中键入文本时，main()方法将读取该文本，并调用 writeMessage()，将该 文本传送给 Chat 实例。writeMessage()方法（如下例所示）使用 pubSession 对象产生一个 TextMessage 对象，该对象能够将文本传送给主题：

```
protected void writeMessage(String text)throws JMSException {

   TextMessage message = pubSession.createTextMessage();
   message.setText(username+" ： "+text);
   publisher.publish(message);
}
```

一个 TopicSession 可以创建若干类 Message。最常用的一类是 TextMessage。

Topic

JNDI 用于定位一个 Topic 对象，它是类似于 TopicConnectionFactory 的一个受管对象：

```
InitialContextjndi = new InitialContext(env)；
...

//查找一个 JMS 主题
Topic chatTopic =(topic)jndi.lookup(topicName);
```

Topic 对象是消息传送服务器上一个**实际主题**的句柄（handle）或标识符（identifier），该主题称为物理主题（physical topic）。一个物理主题就是多个客户端订阅和发布消息的一条电子通道。一个主题类似于一个新闻组或邮件列表服务器：当一条消息被发送到一个新闻组或邮件列表服务器时，它就会传送给所有的订阅者。同样地，当 JMS 客户端向一个主题传送 Message 对象时，订阅该主题的所有客户端都会接收到该 Message。

Topic 对象封装了一个厂商指定的（vendor-specific）名称，用来 在消息传送服务器中标识一个物理主题。Topic 对象有一个 getTopicName()方法，它将该名称标识符返回给它所

代表的物理主题。由 Topic 对象封装的名称是厂商指定的，并且因产品而异。例如，一家厂商可能会使用以点（.）分开的主题名称，比如“oreilly.jms.chat”，而另一家厂商则可能会使用一种完全不同的命名系统，类似于 LDAP 命名“o=oreilly，cn= chat”。如果直接使用主题名称，会导致客户端应用程序不能跨 JMS 服务器品牌移植。Topic 对象隐藏了来自客户端的主题名称，使得客户端的可移植性更强。

作为约定，我们将一个物理主题称为一个**主题**，而且，只有在表明它和 Topic 对象的区别非常重要时，才使用 “物理主题”这个术语。

TopicPublisher

一个 TopicPublisher 是使用 pubSession 和 chatTopic 来创建的：

```
//查找一个 JMS 主题
Topic chatTopic =(topic)ctx.lookup(topicName);

//创建一个 JMS 发布者和订阅者
TopicPublisher publisher =
    pubSession.createPublisher(chatTopic);
```

TopicPublisher 用于将消息传送给一个消息服务器上的特定主题。在 createPublisher()方法中使用的 Topic 对象，确定了将从 TopicPublisher 接收消息的主题 在 Chat 一例中,在命令行键入的任何文本都会传送给 Chat 类的 writeMessage()方法。这种方法使用 TopicPublisher 向该主题传送一条消息：

```
/*使用发布者创建并发送消息*/
protected void writeMessage(String text)throws JMSException {
   TextMessage message = pubSession.createTextMessage();
   message.setText(username+": "+text);
   publisher.publish(message);
}
```

TopicPublisher 将消息异步传送给该主题。异步传送和消费消息是面向消息中间件的一个关键特性；TopicPublisher 不会一直阻塞或等到所有的订阅者都接收到该消息为止，取而代之的是，只要消息服务器一接收到消息，TopicPublisher 就会从 publish()方法返回。它依赖消息服务器将消息传送给该主题的所有订阅者。

TopicSubscriber

TopicSubscriber 是使用 subSession 和 chatTopic 来创建的：

```
//查找一个 JMS 主题
Topic chatTopic =(topic)ctx.lookup(topicName);

//创建一个 JMS 发布者和订阅者
TopicSubscriber subscriber =
    subSession.createSubscriber(chatTopic, null, true);
```

TopicSubscriber 从特定的主题接收消息。在 createSubscriber()方法中使用的 Topic 对象参数，确定了 TopicSubscriber 将要从哪一个主题中接收消息。第 2 个参数包含了消息选择器，它用于过滤出我们想要接收的那些基于特定标准（criteria）的消息。在这个例子中，我们将这个参数值设置为 null，表明我们想要接收所有的消息。第 3 个参数包含了一个布尔值，表明我们是否想要接收自己发布的消息。在这个例子中，我们将该值设置为 true（真 ，这就表明，作为该主题的一个订阅者，我们并不想看到自己发布的消息。

TopicSubscriber 从消息服务器（连续地）一次接收一条消息。消息服务器将这些消息异步推送给 TopicSubscriber，这就意味着 TopicSubscriber 不必轮询消息服务器。在我们的例子中，每个聊天客户端都会接收到任何其他聊天客户端发布的所有消息。当一个用户在命令行输入文本时，这个文本消息就会被传送给订阅同一主题的其余所有聊天客户端。

JMS中的发布/订阅消息传送模型包括了一个用于处理输入消息的进程内（in-process）Java 事件模型。这类似于Java-Beans使用的事件驱动模型 （注 3）。一个对象非常简单地实现了侦听器接口（在本例中就是指MessageListener），然后使用TopicSubscriber进行注册。一个 TopicSubscriber 只能有一个 MessageListener 对象。在 JMS 中使用的 MessageListener接口定义如下：

```
package javax.jms;

public interface MessageListener {
    public void onMessage(Message message);
}
```

当 TopicSubscriber 从它的主题接收一条消息时，调用了它的 MessageListener 对象的 onMessage()方法。Chat 类自身实现了 MessageListener 接口和 onMessage()方法：

```
public class Chat implements javax.jms.MessageListener {
    ...
    public void onMessage(Message message){
        try{
            TextMessage TextMessage =(textMessage)message;
            System.out.println(textMessage.getText());
        } catch（JMSException jmse){ jmse.printStackTrace(); }
    }
...
}
```

Chat 类是 MessageListener 类型，因此，它使用 TopicSubscriber 在其构造函数中自我注册：

注 3：尽管 TopicSubscriber 使用的进程内（in-process）事件模型类似于 JavaBeans 中使用的事件模型，但 JMS 自身也是一个 API，而且它定义的接口并不是 JavaBeans。

```
TopicSubscriber subscriber = subSession.createSubscriber(chatTopic);
subscriber.setMessageListener(this);
```

当消息服务器将一条消息推向 TopicSubscriber 时，TopicSubscriber 将调用 Chat 对象的 onMessage()方法。

提示：很容易把 Java 消息服务和它的 Java 事件模型应用这二者混为一谈。Java 消息服务（JMS）是一个 API，用于在网络上跨进程和跨机器异步传送分布式企业消息。Java 事件模型则用于通过调用一个或多个对象上的方法来传送事件，这些（个）对象处于同一过程之中，而且已经注册为侦听器。JMS 发布/订阅模型使用 Java 事件模型，以便 TopicSubscriber 通知它在同一过程中的 MessageListener 对象：一条来自消息服务器的消息已经到达。

Message

在聊天的例子中，TextMessage 类用于封装我们发送和接收的消息。TextMessage 包含一个 java.lang.String 作为它的消息体，而且它也是最为常用的消息类型。onMessage()方法接收来自 TopicSubscriber 的 TextMessage 对象。同样地，writeMessage()方法使用 TopicSubscriber 来创建和发布 TextMessage 对象：

```
/*接收来自主题订阅者的消息*/
public void onMessage(Message message){
  try {
    TextMessage TextMessage =(textMessage)message;
    String text = TextMessage.getText();
    System.out.println(text);
  } catch（JMSException jmse){ jmse.printStackTrace(); }
}

/*使用发布者创建并发送消息*/
protected void writeMessage(String text)throws JMSException {
   TextMessage message = pubSession.createTextMessage();
   message.setText(username+": "+text);
   publisher.publish(message);
}
```

一条消息可分为 3 部分：**消息头**、**属性**和**有效负载**。消息头由用于标识消息、声明消息属性及提供路由信息的特殊字段（special field）组成。消息的属性区包含了和该消息有关的附加元数据，这个元数据由应用程序开发者进行设置，或者在某些情况下，由 JMS 提供者进行设置（第 3 章会详述）。消 息类型之间的区别在很大程度上取决于它们的有效负载（也就是，该消息包含的应用程序数据类型）。Message 类作为所有消息对象的超类，没有有效负载。它是一个轻量级消息，虽然并不传送有效负载，却能用作一个简单的事件通知。其他消息类型都会有决定它们类型和用途的特定有效负载：

`Message`

这种类型不含有效负载。它可以用于简单的事件通知。

`TextMessage`

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

`ObjectMessage`

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

`BytesMessage`

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

`StreamMessage`

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

`MapMessage`

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

2.1.3 会话和线程
Sessions and Threading

`Chat` 应用程序分别为发布者和订阅者设置了两个单独的会话：`pubSession` 和 `subSession`。其原因在于 JMS 强制实行的线程限制。按照 JMS 规范的规定，一个会话不能同时在一个以上的线程中运行。在我们的例子中，有两个控制线程是活动的（active）：`Chat` 应用程序默认的主线程和调用 `onMessage()`处理程序（handler）的线程。调用 `onMessage()`处理器的线程属于 JMS 提供者所有。由于 `onMessage()`处理器是异步调用的，所以当主线程（main thread）在 `writeMessage()`方法中发布一条消息时，就可以调用 `onMessage()`方法。如果发布者和订阅者都是由同一个会话创建的，那么，这两个线程就能够同时在这些方法上运行。实际上，它们是能够在同一个 `TopicSession` 上并发运行的，不过，这种情况被禁止了。

JMS 规范的一个目标就是要避免将一个内部体系结构强加在 JMS 提供者上。特别是要避免要求 JMS 提供者对 `Session` 对象的实现具有安全处理多个线程的能力。这主要是考虑了 JMS 的预定用途——JMS API 是现有消息传送系统的一个包装器，它可能在客户端不具有多线程传送的能力。

对 JMS 提供者提出的要求就是，要能够连续地依次发送消息和异步接收消息。对发布/订阅模型来说，使用相同的会话是可能的，但这仅限于应用程序是由 `onMessage()`处理器内部发布的情况。关于这一点，第 5 章会给出一个具体示例。