大 师 签 名 系 列

Enterprise Integration Patterns

Design,Building,and Deploying Messaging Solutions

MASTER SIGNATURE SERIES 大师签名系列

企业集成模式

设计、构建及部署消息传递解决方案

[美] Gregor Hohpe Bobby Woolf 著

荆涛 王宇 杜枝秀 译

为本书作出贡献的还有

Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon

John Crupi 和 Martin Fowler 分别为本书作序



PEARSON Education

中国电力出版社
www.infopower.com.cn

MASTER SIGNATURE SERIES

大师签名系列

Enterprise Integration Patterns:
Designing, Building, and Deploying Messaging Solutions

企业集成模式
设计、构建及部署消息传递解决方案

- 你想使用统一的可视化符号绘制集成解决方案吗？请查看“模式列表”。
- 你想在发挥异步系统功用的同时克服它的不足吗？请阅读“引言”中的“异步思想”。
- 你想知道哪种应用集成方式最适合你的需要吗？请阅读第2章“集成方式”。
- 你想掌握并发处理消息的技术吗？请阅读第10章的“竞争消费者”和“消息分派器”。
- 你想知道当异步的消息流经分布式系统时如何跟踪它们吗？请阅读第11章中的“消息历史”和“消息存储库”。
- 你想了解如何用Java Web服务、.NET消息排队和基于TIBCO的发布－订购体系结构实现用集成模式设计的系统吗？请阅读第9章“中场演练：复合消息传递”。

凭借多年的实践，经验丰富的专家Gregor Hohpe和Bobby Woolf指出，异步消息传递已证明是成功实现企业集成的最佳方案。但是，为了构建和部署消息传递解决方案，开发人员要面对大量的问题。本书提供了意义非凡的65种模式，并用真实的解决方案演示了消息传递的强大功能，可以帮助你为自己的企业设计出高效的消息传递解决方案。

作者还列举了涉及各种不同集成技术的例子，如JMS、MSMQ、TIBCO ActiveEnterprise、Microsoft BizTalk、SOAP和XSL，其中债券交易系统的实例研究给出了模式的实际应用。另外本书还介绍了各种新兴的标准，以及企业集成的发展趋势。

对于采用了多种技术的大规模集成解决方案，本书为描述此类解决方案提供了统一的术语表和可视化的符号框架。它还详细介绍了异步消息传递体系结构的优点和限制条件。作者对于将应用连接到消息传递系统给出了实用的代码设计意见，并提供了丰富的信息，可以帮助你确定何时发送消息、如何把消息路由到正确的目标、如何监视消息传递系统的状态等等。在使用了消息传递系统后，如果你想知道如何进行管理、监视和维护，请购买本书。

Gregor Hohpe 负责ThoughtWorks公司的企业集成实践，该公司是应用开发和集成服务的专业提供商。根据为企业客户设计和实现集成解决方案的丰富经验，Gregor已经在企业集成、Web服务和面向服务的体系结构方面发表了大量的论文。他经常在世界各地的学术会议上发表演说。**Bobby Woolf** 是《The Design Patterns Smalltalk Companion》一书（1998年由Addison-Wesley出版）的作者之一，也是《IBM DeveloperWorks》杂志、《Java Developer's Journal》和其他杂志的撰稿人。他还在其他学术会议上做OOPSLA、JavaEdge和Smalltalk Solutions的教学推广。

责任编辑/牛贵华
封面设计/杜长清

ISBN 7-5083-4114-7

9 787508 341149 >

PEARSON Addison Wesley

www.PearsonEd.com

ISBN 7-5083-4114-7

定价：59.00元

大 师 签 名 系 列

Enterprise Integration Patterns
Design,Building,and Deploying Messaging Solutions

企业集成模式
设计、构建及部署消息传递解决方案

[美] Gregor Hohpe Bobby Woolf 著
荆涛 王宇 杜枝秀 译



中国电力出版社
www.infopower.com.cn

Enterprise Integration Patterns:Designing,Building,and Deploying Messaging Solutions (ISBN 0-321-20068-3)
Gregor Hohpe, Bobby Woolf
Copyright ©2004 by Pearson Education, Inc.
Original English Language Edition Published by Addison-Wesley.
All rights reserved.
Translation edition published by PEARSON EDUCATION ASIA LTD and CHINA ELECTRIC POWER PRESS, Copyright © 2006.

本书翻译版由 Pearson Education 授权中国电力出版社独家出版、发行。
未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 防伪标签，无标签者不得销售。
北京市版权局著作权合同登记号 图字：01-2005-3610 号

图书在版编目（CIP）数据
企业集成模式：设计、构建及部署消息传递解决方案 /（美）侯珀（Hohpe,G.），（美）沃尔夫（Woolf,B.）著；荆涛，王宇，杜枝秀译. —北京：中国电力出版社，2006
（大师签名系列）
书名原文：Enterprise Integration Patterns: Designing，Building，and Deploying Messaging Solutions
ISBN 7-5083-4114-7
Ⅰ. 企... Ⅱ. ①侯...②沃...③荆...④王...⑤杜... Ⅲ. 企业管理—应用软件 Ⅳ. F270.7
中国版本图书馆 CIP 数据核字（2006）第 009213 号

丛 书 名：大师签名系列
书　　名：企业集成模式：设计、构建及部署消息传递解决方案
编　　著：（美）Gregor Hohpe, Bobby Woolf
翻　　译：荆 涛 王 宇 杜枝秀
责任编辑：牛贵华
出版发行：中国电力出版社
　　地址：北京市三里河路 6 号　邮政编码：100044
　　电话：（010）88515918　传　真：（010）88518169
印　　刷：汇鑫印务有限公司
开本尺寸：185×233　印　张：32.25　字　数：756 千字
书　　号：ISBN 7-5083-4114-7
版　　次：2006 年 3 月北京第 1 版　2006 年 3 月第 1 次印刷
定　　价：59.00 元
版权所有　翻印必究

模式列表


聚合器：如何把独立但相关的结果消息组合起来，从而作为一个完整的消息来处理？

规范数据模型：当集成的应用使用不同的数据格式时，如何把它们的相互依赖减到最少？


通道适配器：应该如何把应用连接到消息传递系统，使之能发送和接收消息？


通道清洁者：如何防止通道中剩余的消息干扰测试和正在运行的系统？


声明标签：在不损害信息内容的前提条件下，如何减少跨系统发送的消息的数据量？


命令消息：怎样利用消息传递调用其他应用中的过程？


竞争消费者：消息传递客户端如何并发处理多个消息？


组合消息处理器：如果要处理的消息包含多个元素，而且每个元素需要不同的处理，那么如何维持整个消息流？


内容扩充器：如果消息的创建者无法提供所需的所有数据项，如何与其他系统通信？


内容过滤器：处理一个很大的消息时，如果只对其中的少量数据项感兴趣，应该如何简化？


基于内容的路由器：如果一个逻辑功能的实现需要跨越多个物理系统，这种情况应该如何应对？


控制总线：当消息传递系统分布在多种平台和很大的地理范围内时，如何有效地进行管理？


关联标识符：接收到应答的请求者如何知道这个应答对应哪一个请求？


数据类型通道：应用如何发送数据项才能让接收者知道怎样加以处理？


死文字通道：消息传递系统应该如何处理无法传送的消息？


旁路：如何才能使消息经过中间处理步骤，完成验证、测试或调试功能？


文档消息：如何使用消息传递在应用之间传输数据？


持久订购者：如何防止订购者丢失在其未监听期间发布的消息？


动态路由器：如何避免路由器对所有可能目标的依赖，同时还能保持其效率？


信封包装器：如果已有的系统对消息格式有特定的需求，诸如要求有消息首部字段或者要求加密，这样的系统如何参与消息传递交换？


事件消息：如何使用消息传递把事件从一个应用传送给另一个应用？


事件驱动的消费者：当消息到达时，如何让应用自动消费这些消息？


文件传输：如何集成多个应用，使它们能协同工作并交换信息？

格式指示符：如何设计消息的数据格式，使之能适应未来可能的变化？


可靠传输：发送者怎样才能确保即使消息传递系统失败，消息也能成功传送？

幂等接收者：消息接收者如何处理重复的消息？


非法消息通道：如果接收到无意义的消息，消息传递接收者如何妥善地处理？


消息代理：如何让消息的目标与消息的发送者解耦合，并保持对消息流的集中控制？


消息总线：应该采用哪种体系结构才能使单独的应用协同工作，但是又彼此解耦合，使得增加或修改某个应用不会影响其他应用？


消息通道：一个应用如何使用消息传递与另一个应用通信？


消息分派器：单个通道上的多个消费者如何才能协调其消息处理？


消息端点：应用如何连接到消息传递通道来发送和接收消息？

消息到期：发送者如何指示到期时间（超过这个时间则认为消息过时，不应再做处理）？

消息过滤器：如何避免组件接收到与之无关的消息？

消息历史：如何在一个松耦合的系统中有效地调试和分析消息流？

消息路由器：如何才能对各个处理步骤解耦合，使消息能按照一组条件传递到不同的过滤器？

消息序列：怎样使用消息传递传送任意大小的数据？

消息存储库：如何对消息的信息做出报告，而且不影响消息传递系统的松耦合和暂时性？

消息转换器：如何使用消息传递从而使得采用不同数据格式的系统之间实现通信？

消息：消息通道连接的两个应用如何交换一个信息？

消息传递桥：怎样把多个消息传递系统连接起来，使一个系统上的消息也能到达其他系统？

消息传递网关：如何把对消息传递系统的访问封装起来，使之独立于应用的其他部分？

消息传递映射器：如何实现领域对象与消息传递基础设施之间的数据传递，同时保持它们之间的相互独立？

消息传递：如何集成多个应用，使它们能协同工作并交换信息？

规范器：如何处理语义上等价但是格式不同的消息？

管道和过滤器：在保持独立性和灵活性的前提下，如何对消息完成复杂的处理？

点对点通道：调用者如何保证只有一个接收者能够接收文档或执行调用？

轮询消费者：应用准备好时，如何消费一个消息？

过程管理器：如果无法在设计时确定所需的处理步骤，或者处理步骤不是顺序执行的，应该如何通过多个处理步骤完成消息的路由？

发布-订购通道：发送者如何才能向所有感兴趣的接收者广播一个事件？

接收表：如果接收者列表会动态变化，如何把消息路由给这样一组接收者？

远程过程调用：如何集成多个应用，使它们能协同工作并交换信息？

请求/应答：应用发送一个消息后，如何才能获得接收者返回的响应？

重排器：如何把相互关联但是乱序的消息重新恢复成正确的顺序？

返回地址：应答者如何知道将应答消息发送到哪里？

路由表：如果设计时不清楚消息处理的步骤，而且每条消息的处理各不相同，那么如何让消息通过一系列处理步骤连续地得到路由？

分散收集器：如果必须把消息发送给多个接收者，而且每个接收者都会返回应答，如何维护整个消息流？

选择性消费者：消息的消费者如何选择想要接收的消息？

服务激励器：应用应该如何设计一个服务，从而既能通过各种消息传递技术调用，也能通过非消息传递技术调用？

共享数据库：如何集成多个应用，使它们能协同工作并交换信息？

智能代理：如果一个服务组件采用请求者指定的返回地址作为应答消息的返回通道，应该如何跟踪这样的消息？

分解器：如果消息中包含多个元素，每个元素要采用不同的方式处理，这种消息应该如何处理？

测试消息：组件正在处理消息时，如果因内部故障而导致输出消息被修改，会怎么样？

事务性客户端：客户端应该如何控制它与消息传递的事务？

线路分接器：如何检查点对点通道中传输的消息？

对本书的赞誉

《企业集成模式》一书为我们详细描绘了一种日益兴起的发展趋势，这种势头不仅会影响到金融服务领域中的所有软件，还会对这一领域的业务过程架构师产生影响，使我们在利用顾客原有系统投资的同时，能够更有创新性、更具竞争力。这本书的主题紧扣时代，以一种执行驱动（execution-driven）的简洁方式来介绍消息传递和工作流模式，这与当今信息量超大的事件驱动（event-driven）环境有紧密关系，而且可以在此环境中直接加以应用。

——Glenn Gameron，Thomson Financial 的中间件解决方案体系结构主管

除了介绍诸如发布-订购和可靠传输等基本概念，《企业集成模式》一书还为架构师提供了一个有关更高级模式的备查资源，这些模式描述了消息传递究竟要如何“使用”。这本书不单单介绍了集成，还花费大量笔墨描述和构建了基于消息传递的应用。利用这里介绍的模式，如路由表、聚合器或重排器，开发人员无论在集成领域还是在从头开始的应用开发项目中，都将有上佳表现。

——Paul Brown，FiveSight Technologies, Inc.的 CEO

《企业集成模式》一书可谓一个里程碑性的成果。一直以来，集成领域都存在一个困惑：集成所用的语言确实都缺乏一致性，更不用说软件或协议标准了。有了这本书，使我们终于有机会改变这种现状，使每一个人（包括生产商、顾问、开发人员和终端用户）都能通过一种共同的“语言”交流。如果说有人做出努力，着力将一些古老陈旧的集成最佳实践从我们的视线中移走而发起一场“复兴”运动，力图为集成领域建立一套正式的原则，这正是本书作者全力打造的！对于每一位 IT 架构师、开发人员和集成人员来说，这都是一本不可缺少的好书。

——John Schmidt，EAI 行业协会董事

《企业集成模式》一书为我们提供了一个坚实的基础，以便于理解集成的现状和未来。该书作者使用模式为我们清楚地展示了他们在这方面积累的智慧和经验，使我们能有幸分享这宝贵的资源。在审阅和阅读这本书的过程中，我学到了很多，并期盼着未来几年能把书中的建议付诸实践。

——Luke Honhmann，《Beyond Software Architecture》[①]的作者

这本书不光对如何使用消息传递来解决集成问题提供了极有价值的建议，还深入地展示了每种方法的意义何在。该书作者清晰地解释了消息传递在集成领域最常见的用法，还介绍了使用消息通道来解决复杂问题的简洁方法。

——Dave Chappell，Sonic Software 的副总裁和首席技术讲师，《Enterprise Service Bus》、《Java Web Services》和《Java Messaje Service》的作者

如果你参与过一个企业应用的运作或开发，肯定会有需要采用当前看好的消息传递方

---

① 该书中文版《超越软件架构》已由中国电力出版社出版。

法将你的应用与另一个应用集成之时。这一天到来时，这本书将成为最有价值的参考资料。Bobby 和 Gregor 在使用消息传递完成应用集成的方面做了非凡的工作，他们收集到软件开发行业中许多难得一见的智慧结晶，并把它们清晰地组织为一组模式，这是软件业人员之间交流设计心得的一种首选形式。由此我们得到了这样一个模式表和一组可行的解决方案，软件业人员可以利用这些模式来设计和讨论企业应用的集成。

—Randy Stafford，IQNavigator, Inc.首席架构师

献辞

献给我的家人，以及当我从本书中脱身时还记得我的所有朋友。

—Gregor

献给 Sharon，我的新婚妻子

—Bobby

序

John Crupi

当新的技术到来时，你会如何面对它？对我来说，就是尽快掌握这门技术。我研究过J2EE（由 Sun Microsystems 公司提出，它对我非常有用）。由于当时缺乏相关书籍，我只能阅读 J2EE 的规范，并把注意力集中在 EJB 技术方面。然而，掌握技术只是第一步，真正的目的应该是学会如何有效地应用技术。平台技术的优点在于它们能帮助你完成指定的任务。就技术本身而言，你可以随心所欲地用它做任何事情，可是一旦应用不当，就会带来很多麻烦。

在过去的 15 年里，我发现编程和设计这两个领域——确切地说，是有效的编程和设计这两个领域——一直困扰着软件开发人员。已经出版的大量书籍介绍了如何采用 Java 和 C#实现最有效率的编程，但是介绍如何进行更有效设计的书籍却很少，这也是创作这本书的初衷。我和 Deepak Alur、Dan Malks 编写《Core J2EE Patterns》的目的，就是为了帮助 J2EE 的开发人员设计更好的代码。我们的理想是让这些设计模式成为开发者最好的选择。正如 Sun 公司杰出的工程师 James Baty 指出的："模式就好比设计的甜点"。对此我十分赞同，幸运的是，Gregor 和 Bobby 也这么认为。

这本书集中讨论了一个热门的话题：消息传递的集成。这不仅是因为消息传递是集成的关键，同时也因为它在即将兴起的 Web 服务中也将占据支配地位。当前针对 Web 服务存在着太多的争议，仅仅为了确定关键的规范和技术就需要做出细致、艰辛的努力。然而，"软件帮助你解决问题"这一目标并未改变。正像刚刚推出 J2EE 和.NET 技术时一样，目前为 Web 服务提供的设计帮助也不多。很多人认为 Web 服务只是一种能解决现有集成问题的新的、开放的方法。尽管我也同意这种观点，但是我们并不知道如何设计 Web 服务，而这也正是《企业集成模式》这本书的精华所在。相信这本书能为你提供设计 Web 服务和其他集成系统时所需的许多模式。由于 Web 服务规范仍在修改过程中，所以 Bobby 和 Gregor 没有必要提供太多有关 Web 服务规范的例子。不过，当 Web 服务规范成为标准时，完全可以使用这本书介绍的模式来设计由这些标准实现的解决方案。甚至还能实现下一个集成目标：设计面向服务的体系结构。

请阅读这本书，并随时把它放在身边，它将使你的软件职业生涯不断提升。

John Crupi
Bethesda, MD
2003 年 8 月

序

Martin Fowler

我在编写自己的《Patterns of Enterprise Application Architecture》一书时，非常有幸，能以随意交流的方式从位于 Raleigh-Durham 的 Kyle 工作室那里获得 Kyle Brown 和 Rachel Reinitz 对企业集成模式的一些深刻评论。通过这些交流，使我认识到自己的书中最大的缺憾就是异步消息传递系统。

尽管我从没有想过要让自己的书成为企业开发模式的大全，但是异步消息传递方面的缺憾显得尤为突出，因为我们认为异步消息传递将在企业软件开发中，尤其是在企业集成中扮演越来越重要的角色。集成之所以重要是因为相互独立的应用是没有生命力的。我们需要一种技术能将在设计时并未考虑互操作的应用集成起来，打破它们之间的隔阂，获得比单个应用更多的效益。

为了解决集成问题，出现了各种各样的技术。而我们一致认为消息传递将成为最具希望的集成技术。我们所面对的挑战是如何实现更有效的消息传递。目前最大的挑战来自消息本身的异步特性，在异步环境中存在着多种明显不同的设计方式。

我没有时间、精力，或者更坦白地说，没有完全具备相应的知识涵盖《Patterns of Enterprise Application Architecture》一书中有关的内容。但是我们提出了弥补这一缺憾的更好的解决办法：寻找有能力的人完成这一任务。于是我们找到了 Gregor 和 Bobby，他们接受了挑战并写出这本书。

我为他们所做的工作感到高兴。如果你已经使用了消息传递系统，这本书将帮助你和其他人把通过艰苦途径掌握的知识更加系统化。如果你正准备使用消息传递系统，那么不管你要使用哪种消息传递技术，这本书都将为你提供难得的经验。

Martin Fowler
Melrose, MA
2003 年 8 月

前言

这是一本有关使用消息传递进行企业集成的书。这本书并不只是介绍某种特别的技术或产品，而是为使用各种消息传递产品和技术的开发人员以及集成人员准备的，这些消息传递产品和技术包括：

- 由 IBM（WebSphere MQ 系列）、Microsoft（BizTalk）、TIBCO、WebMethods、SeeBeyond、Vitria 以及其他一些开发商提供的面向消息的中间件（Message-Oriented Middleware，MOM）和 EAI 套件。
- 合并到商业和开放源代码 J2EE 应用服务器中的 Java 消息服务（Java Message Service，JMS）实现及独立产品。
- Microsoft 的消息排队（Microsoft's Message Queuing，MSMQ），可以通过多个 API（包括 Microsoft .NET 中的 System.Messaging 库）进行访问。
- 新兴的支持异步 Web 服务的 Web 服务标准（例如 WS-ReliableMessaging）以及有关的 API，如 Sun 的用于 XML 消息传递的 Java API（Java API for XML Messaging，JAXM），或 Microsoft 的 Web 服务扩展（Web Services Extensions，WSE）。

企业集成不仅仅能创建一个分布式 n 层应用结构，还能使一个应用分布到多台计算机上。在分布式应用中，单个的一层是无法自己运行的，而集成应用中的每一个应用都能独自运行，它们彼此之间通过松耦合完成功能。采用“发送并忽略（send and forget）”的方式进行消息传递能够帮助数据或指令通过网络传递，调用者发送完信息后可以继续完成其他操作，信息由消息传递系统负责传递。另外，通过回调函数，调用者事后能得到处理结果的通知。异步调用和回调比同步调用增加了设计的复杂度，但是异步调用可以反复执行直至调用成功，这使得通信更加可靠。异步消息传递还有其他一些优点，如节流请求、平衡负载等。

读者对象

本书适合以下应用开发人员和系统集成人员，可以帮助他们使用面向消息的集成工具来完成应用互连：

- 在设计和构建复杂的企业应用时，需要集成其他应用的应用架构师和开发人员。假设你正在使用现代企业应用平台（如 Java 2 平台企业版或 Microsoft .NET 框架）来开发你的应用，本书能帮助你把应用连接到消息传递层，通过消息传递层与其他应用交换信息。本书的重点在于应用的集成方面，而不是如何构建应用；如果你想了

解如何构建应用，请参考 Martin Fowler 所著《Patterns of Enterprise Application Architecture》一书。

- 为了连接已经封装或定制的应用，设计和构建集成解决方案的集成架构师和开发人员。属于这个群体的大多数读者已经有使用某种商业集成工具（如 IBM WebSphere MQ、TIBCO、 WebMethods、SeeBeyond 和 Vitria 等）的经验。这些工具中，大多数都提供了本书介绍的模式。因此本书能帮助读者理解这些模式的内涵，帮助他们作出独立于产品提供商的设计决策。
- 保持企业中软件和硬件资产“大局观”的企业架构师。本书提供了一种统一的语言，可以帮助架构师描述大规模的集成解决方案，即使这些方案是由许多不同的技术和局部解决方案构成的。采用这种语言能帮助企业架构师、集成与应用架构师和开发人员实现更有效的沟通。

本书内容

本书不会为企业应用集成建立商业案例，而是把重点放在如何实现应用的集成上。读者通过掌握以下知识可以学会如何集成企业应用：

- 与其他集成技术相比较，异步消息传递有哪些优点与局限性。
- 如何规划应用所需的消息通道，如何控制多个消费者能否得到相同的消息，如何处理非法消息。
- 发送一个消息时，消息中应该包含哪些内容，如何使用特殊的消息属性。
- 如何路由消息使之到达最终的目标，即便发送者不知道目标在哪里。
- 当消息的发送者与接收者不能使用共同的消息格式时，如何转换消息。
- 如何设计连接应用与消息传递系统的代码。
- 当消息传递系统作为企业应用的组成部分运行时，如何管理和监控。

本书未涉及的内容

任何在标题中冠以“企业”一词的书籍不外乎以下三类。其一是试图概括整个主题，但缺少如何实现具体解决方案的详细指南；其二是为开发具体解决方案提供详细的手册，但是却无法涵盖整个主题；其三是试图提供既完备又详尽的解决方案，但这样的书籍要么还未出版，要么就是出版得太晚。因此我们倾向于选择第二种方式，希望能编写一本书帮助人们设计更好的集成解决方案，即使书的内容会因此而受限。有一些主题我们本想介绍，但是为了避免陷入第三类情况，因此书中没有涉及，这包括安全性、复杂数据映射、工作流、规则引擎、可扩展性、健壮性和分布式事务处理（诸如 XA、Tuxedo 等）。我们选择异步消息传递作为本书的重点，这是因为存在着大量与之相关的设计问题和折衷方案，而且可以从各种集成应用提供商那里获得许多案例。

本书还是一本有关各种消息传递或中间件技术的指南。在书中你可以找到采用不同技术（如 JMS、MSMQ、TIBCO、BizTalk 和 XSL）的例子，这些例子能为读者演示如何把模式转换成真正的实现。如果你还想获得这些技术的更详细的说明，请阅读本书的参考文献或从在线帮助中获得相关的资源。

本书的组织结构

顾名思义，本书主要收集了各种模式。人们已经普遍认为，模式能把专家在不同领域中缺乏固定答案的知识捕获下来，这些领域包括应用体系结构、面向对象的设计或基于异步消息传递体系结构的集成解决方案。

每种模式都引出一个特殊的设计问题，通过介绍与问题相关的具体环境，提出一种能平衡各种矛盾或动因的解决方案。在大多数情况下，解决方案并不是一下就能想到的，而要经过多次实践不断提炼。因此，每种模式都融合了高级集成开发人员和架构师从反复构建的解决方案和挫折中总结出的经验。也就是说，本书并没有“发明”什么模式；模式是无法发明的，而是从相关领域的实践中发现和观察到的。

由于模式来源于实践者的实际应用，因此，如果你有机会接触过企业集成工具和异步消息传递体系结构，那么你对本书所介绍的许多模式就不会陌生。即便如此，这本书还是值得阅读的。它能帮助你确认一些晦涩的理解是否正确，这包括设计详细的解决方案时如何使用消息传递，以及你可能还没有注意到的各种模式之间的关系。本书还能帮助你将自己的知识更方便地传授给缺乏经验的同事。最后，这本书确定了模式的名称，这为同事之间更好地讨论各种集成设计方案提供了通用的术语。

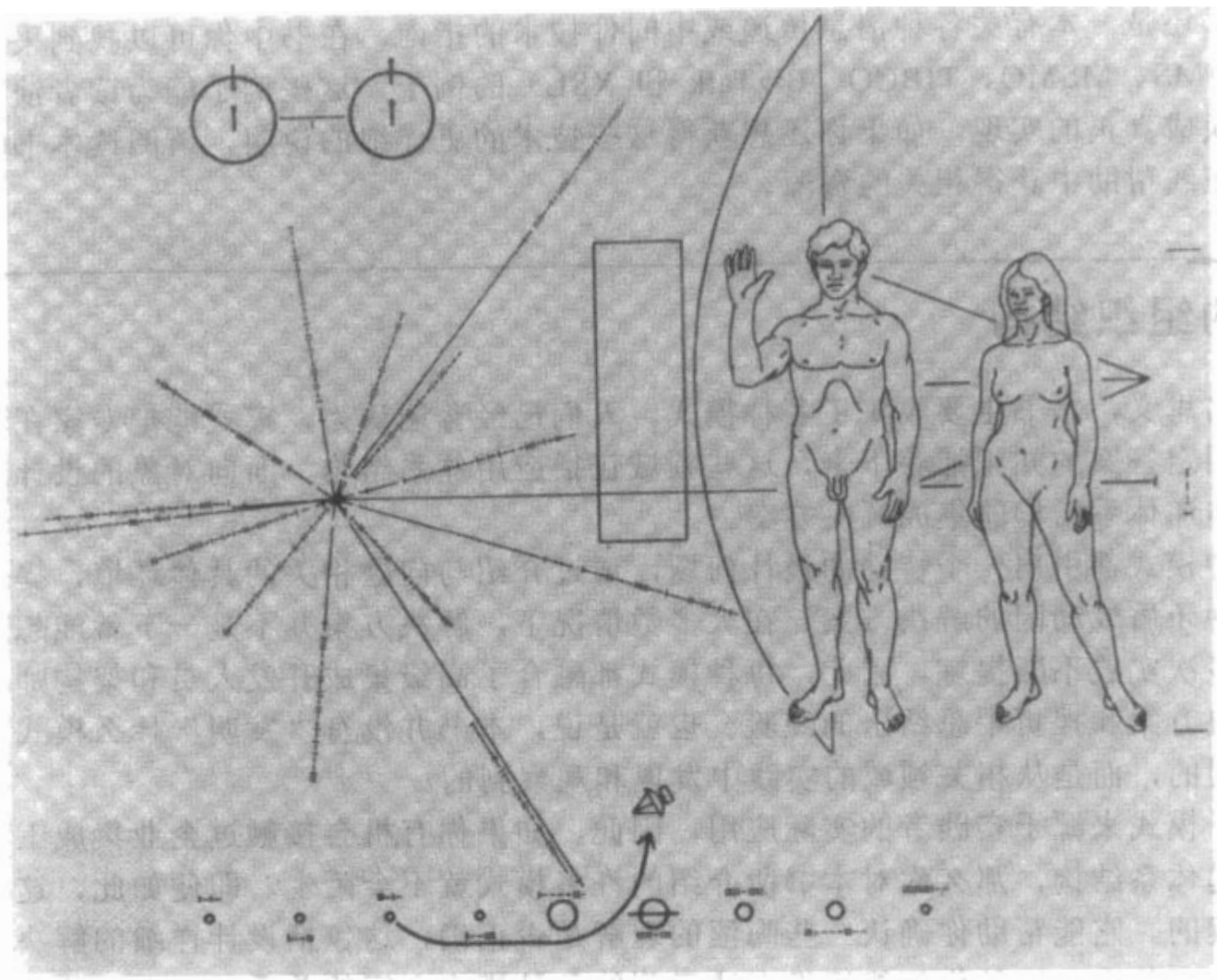
书中的模式可以应用于各种编程语言和开发平台。因此模式并不是一段可粘贴复制的代码，你必须在自己的实践环境中实现模式。为了使这种转换更加容易，书中列举了各种各样的例子，并采用常用的技术（如 JMS、MSMQ、TIBCO、BizTalk、XSL 等）来演示模式的各种实现。另外还提供了几个大型示例，展示如何将多个模式结合起来形成一个整体解决方案。

使用异步消息传递体系结构来集成多个应用是一个富有挑战而又有趣味的研究领域。希望你能像作者一样喜爱阅读这本书。

Gregor Hohpe
San Francisco, California

Bobby Woolf
Raleigh, North Carolina

2003 年 9 月
www.enterpriseintegrationpatterns.com



Carl Sagan 博士绘制的超现代图
这是向宇宙生物发出的一个消息

致谢

和大多数书籍一样，本书的创作经历了很长一段时间。撰写基于消息的集成模式的想法可追溯到 2001 年的夏天，那时 Martin Fowler 正在撰写《Patterns of Enterprise Application Architecture》（P of EAA）。Kyle Brown 发现，尽管那本书介绍了很多如何创建应用的方法，但是对如何集成这些应用只作了简单的描述。这一想法促使 Martin 与 Kyle 进行了一系列的交流，Rachel Reinitz、John Crupi 和 Mark Weitzel 也参与了进来。Bobby 在 2001 年秋，Gregor 在 2002 年初也加入了他们的讨论。同年夏天这个小组在程序模式语言（Pattern Languages of Programs, PLoP）会议上发表了两篇论文，其中一篇是由 Bobby 和 Kyle 合作撰写的，另一篇是由 Gregor 撰写的。在这次会议后，Kyle 和 Martin 把精力集中在他们自己的出书计划上，Gregor 和 Bobby 把他们的论文合并在一起构成了本书的雏形。同时，他们建立了 www.enterpriseintegrationpatterns.com 网站，让全世界的集成架构师和开发人员能参与到本书的写作进程中。当 Gregor 和 Bobby 撰写这本书时，邀请了志愿者帮助审阅本书的内容。在 Kyle 提出最初想法的两年后，本书的终稿才交付出版社。

本书是许多人集体努力的结晶。许多同事与朋友（有的是通过这本书认识的）提供了本书的实例，确保了技术内容的正确性，并给出了大量反馈意见和评价。他们的付出对本书最后的形式与内容产生了很大的影响。很高兴能在这里感谢他们的贡献，并表达我们的谢意。

特别值得指出的是，Kyle Brown 和 Martin Fowler 为本书的创作奠定了基础。如果不是 Martin 编写了《Patterns of Enterprise Application Architecture》一书，不是 Kyle 组建了一个讨论消息传递模式的小组来帮助 Martin 完善他的作品，这本书也不会成稿。

很高兴有 Conrad F. D'Cruz、Sean Neville、Michael J. Rettig 和 Jonathan Simon 这几位作者帮助编写了本书的重要章节。这些章节展示了如何在实践中应用模式。

2002 年 PLoP 会议上的同仁，包括 Ali Arsanjani、Kyle Brown、John Crupi、Eric Evans、Martin Fowler、Brian Marick、Toby Sarver、Jonathan Simon、Bill Trudell 和 Marek Vokac 是第一批为本书提供反馈意见的读者，他们帮助我们选择了正确的写作方向。

还要感谢审阅小组抽出时间审查本书的初稿并提供了宝贵的反馈意见和建议，他们是：

Richard Helm

Luke Hohmann

Dragos Manolescu

David Rice

Russ Rufer 及硅谷模式小组

Matthew Short

特别要感谢 Russ 在硅谷模式小组（Silicon Valley Patterns Group）编排了本书的初稿。还要感谢以下人员的工作：Robert Benson、Tracy Bialik、Jeffrey Blake、Azad Bolour、John Brewer、Bob Evans、Andy Farlie、Jeff Glaza、Phil Goodwin、Alan Harriman、Ken Hejmanowski、Deborah Kaddah、Rituraj Kirti、Jan Looney、Chris Lopez、Jerry Louis、Tao-hung Ma、Jeff Miller、Stilian Pandev、John Parello、Hema Pillay、Russ Ruffer、Rich Smith、Carol Thistlethwaite、Debbie Utley、Walter Vannini、David Vydra 和 Ted Young。

我们提供了公开的邮件讨论列表，欢迎大家对 www.enterpriseintegrationpatterns.com 上的资料发表自己的观点与看法。特别要感谢 Bill Trudell 对邮件列表的热心参与。其他积极参与讨论的人员包括 Venkateshwar Bommineni、Duncan Cragg、Hohn Crupi、Fokko Degenaar、Shailesh Gosavi、Christian Hall、Ralph Johnson、Paul Julius、Orjan Lundberg、Dragos Manolescu、Rob Mee、Srikanth Narasimhan、Sean Neville、Rob Patton、Kirk Pepperdine、Matthew Pryor、Somik Raha、Michael Rettig、Frank Sauer、Jonathan Simon、Frderico Spinazzi、Randy Stafford、Marek Vokac、Joe Walnes 和 Mark Weitzel。

感谢 Martin Fowler 邀请我们把本书加入其签名的出版系列中。Martin 的认可为我们完成本书增加了信心与动力。

感谢 John Crupi 为本书作序。他从一开始就关注本书的创作，并自始至终不失幽默地给予我们指导。

最后，要特别感谢 Addison-Wesley 的编辑与出版小组，这个小组由主编 Mike Hendrickson 领导，由制作统筹 Amy Fleischer、项目经理 Kim Arney Mulcahy、文字编辑 Carol J. Lallier、校对 Rebecca Rider、索引编辑 Sharon Hilgenberg，以及 Jacquelyn Doucette、John Fuller 和 Bernard Gaffney 组成。

对那些本该致谢但没有提及的其他人员，我们在此一并表达歉意。不管前面有没有提到，我们要感谢所有为使本书变得更好的人员所给予的帮助。希望你也能如我们一样为本书感到自豪。

引言

让人们感兴趣的应用很少是独立存在的。销售应用必须要与库存应用打交道，采购应用必须与某个拍卖站点连接，或者你的个人数字助理（PDA）上的日历必须与公司的日历服务器同步，似乎所有的应用通过与其他应用集成后都会变得更好。

所有的集成解决方案必须应对以下几个基本挑战：

- 网络是不可靠的。集成解决方案必须通过网络把数据从一台计算机传送到另一台计算机上。与运行在同一台计算机上的进程相比，分布式计算必须应对更多可能出现的问题。很多情况下，若需要集成的两个系统位于不同的大陆，它们之间的数据必须通过电话线、局域网段、路由器、交换机、公共网络和卫星链路传输。其中每个环节都会带来延迟或中断。
- 网络速度慢。通过网络发送数据要比执行一个本地方法调用慢多个数量级。如果使用设计单一应用的方法设计一个广域分布式解决方案，可能会对性能造成灾难性的影响。
- 任何两个应用都是不同的。集成解决方案需要在使用不同编程语言、不同操作平台和不同数据格式的系统之间传送信息。因此，集成解决方案要能与这些不同的技术打交道。
- 改变是难免的。应用会随着时间改变。一个集成解决方案必须跟上与之相关连的应用变化的步伐。集成解决方案应该能轻易捕获改变所带来的雪崩效应：如果一个系统改变了，所有其他系统也会受到影响。通过在应用间使用松耦合连接，集成解决方案需要尽可能减少系统之间的依赖关系。

随着时间的推移，开发人员使用以下四种主要方法克服了上述挑战：

1. 文件传输——一个应用写文件，之后另一个应用读这个文件。为此，应用之间需要协商文件名、文件的位置、文件的格式、文件读写的时间以及谁负责删除这个文件。

2. 共享数据库——多个应用共享相同的数据库，这个数据库位于独立的物理数据库中。由于不存在重复保存的数据资料，因此不必将数据从一个应用传送给另一个应用。

3. 远程过程调用——一个应用开放其部分功能，使得其他应用能够远程访问这些过程。它们之间的通信是实时、同步的。

4. 消息传递——一个应用向公共消息通道中发布一个消息，其他应用可以在之后某个时间从通道中获得这个消息。应用之间必须协商建立通道以及消息的格式。这种通信是异步的。

尽管上述四种方法从本质上解决的是相同的问题，但每种方法均有其独特的优点和不足。实际上，应用之间可通过多种方法集成，使得每个集成点都能充分利用最合适的方法。

什么是消息传递？

这是一本介绍如何使用消息传递来集成应用的书。为了更简单地理解消息传递的作用，不妨考虑电话系统。电话呼叫采用了同步通信方式。只有被呼叫的另一方正好在位，才能与之通话。而语音邮件却能支持异步通信。使用语音邮件时，如果接收者没有应答，呼叫者可给他留下消息，之后接收者在方便的时候可以收听存储在邮箱消息队列中的消息。利用语音邮件，呼叫者能将消息留下，让接收者在以后收听，这要比同时让呼叫者与接收者通过电话交换信息方便许多。语音邮件把电话呼叫转换成消息和队列，以便事后使用，这就是消息传递的基本原理。

消息传递是一种能支持高速、异步、程序到程序间可靠通信的技术。程序之间通过发送数据包进行通信，这种数据包即称为消息（message）。通道（channel），也称为队列，是连接程序并传递消息的逻辑路径。通道的行为类似于消息集合或消息数组，但是它能在多台计算机之间神奇地共享，同时被多个应用使用。发送消息的应用被称为发送者（sender）或生产者（producer），它把消息写到通道中。从通道中读取（和删除）消息的应用称为接收者（receiver）或消费者（consumer）。

消息本身是一种简单的数据结构——如字符串、字节数组、记录或对象。可以把它看作是数据、接收方所调用的命令，或者是发送方所发生事件的描述。消息实际上是由两部分组成的：消息首部和消息体。消息首部包含消息的元信息——谁发送的消息、其目标是哪里等等。这一信息为消息传递系统所使用，使用消息的应用大多会忽略这些信息。消息体包含实际传送的应用数据，这些信息通常会被消息传递系统忽略。在讨论中，如果使用消息传递的应用开发人员谈到消息，那么他所指的是消息体中的数据。

异步消息传递体系结构的功能非常强大，但是在使用时需要再三斟酌开发方法。与其他三种集成方法相比，很少有开发人员尝试使用消息传递和消息系统。因此，应用开发人员对这种通信平台的相关术语与特性还不太熟悉。

什么是消息传递系统？

消息传递能力是由一种单独的称为消息传递系统或面向消息中间件（Message-Oriented Middleware，MOM）的软件系统提供的。消息传递系统管理消息传递的方式就如同数据库系统管理数据的持久性。就像管理员必须根据应用数据的模式（schema）来组装数据库一样，要使用通道部署消息传递系统，通道定义了应用之间的通信途径。消息传递系统负责协调和管理消息的发送与接收。数据库系统的主要目的是确保每条数据记录都是安全持久的，类似地，消息传递系统的主要任务是为了把消息从发送者的计算机以可靠的方式传递给接收者的计算机。

消息传递系统之所以要把消息从一台计算机转移到另一台计算机上，是因为计算机和连接计算机的网络本来就不可靠。一个应用准备好发送数据，并不意味着另一个应用已经

准备好接收数据。即使两个应用都准备好了，网络可能并不通畅，或者无法正确地传递数据。消息传递系统通过反复尝试传递消息直到成功，可以克服上述限制。在理想的情况下，消息在第一次尝试时就会传递成功，但是实际的情况则往往并不理想。

从本质上讲，消息传递会经过五个步骤：

1. 创建——发送者创建消息并组装其数据。

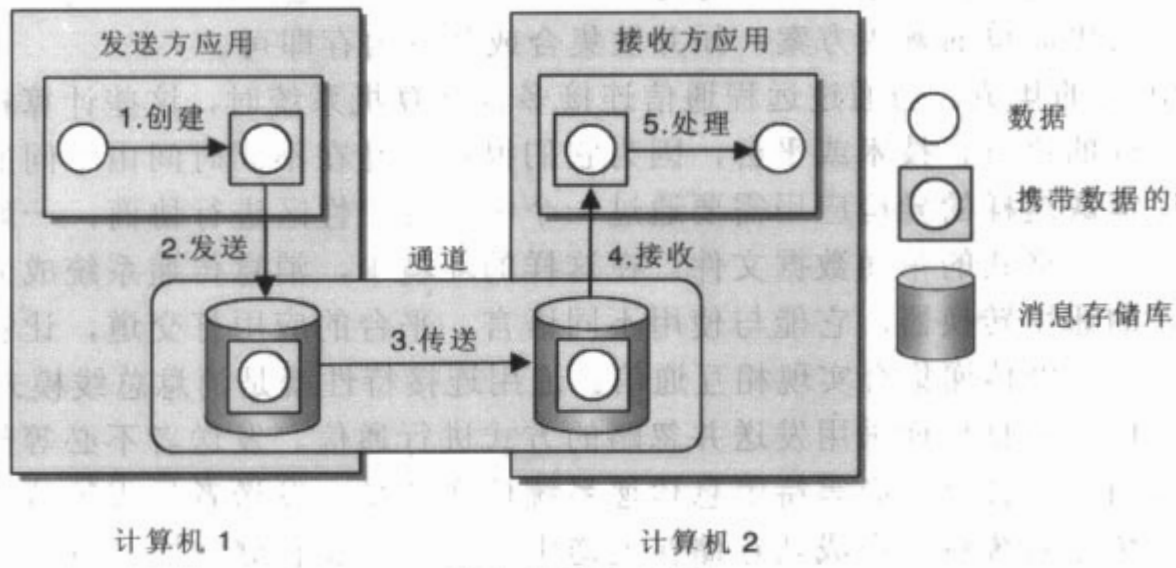
2. 发送——发送者把消息添加到通道中。

3. 传送——消息传递系统把消息从发送者的计算机转移到接收者的计算机，使接收者能获得这个消息。

4. 接收——接收者从通道中读取消息。

5. 处理——接收者从消息中提取数据。

下图说明了消息传递的五个步骤，从中可以看到计算机执行了哪些步骤，以及消息传递系统执行了哪些步骤。



消息传递的步骤

这个图还表达了两个重要的消息传递概念：

1. 发送并忽略——在第 2 步，发送方应用把消息发送到消息通道中。一旦发送完毕，发送者就可以继续其他工作，而让消息传递系统在后台传递消息。发送者可以肯定接收者最终能够获得消息，因此它不必等待。

2. 存储并转发——在第 2 步，当发送方应用把消息发送到通道中后，消息传递系统把消息存储在发送者的计算机内存或磁盘中。在第 3 步，消息传递系统把消息从发送者的计算机转发给接收者的计算机，然后再次把消息存储在接收者的计算机中。当消息从一台计算机转移到另一台计算机上时，这一存储转发过程会重复多次，直到消息到达接收者的计算机。

创建、发送、接收和处理这些步骤好像是多余的。为什么不直接把数据发送给接收者，而是让应用把数据作为消息封装并存储在消息传递系统中，然后委托消息传递系统负责发送数据呢？这是因为，数据封装为原子消息，传送操作可以反复执行，直到这一操作成功，这样接收者就能确信收到正确的数据副本。

为何使用消息传递？

既然了解了消息传递是什么，下一步就要问到为何使用消息传递？对于任何复杂的解决方案，要给出简单的答案并不容易。简要地说，消息传递要比文件传输更直接，对数据的封装要比共享数据库更好，传输可靠性要比远程过程调用高。但是，这些只是刚开始使用消息传递系统就能得到的好处。

使用消息传递的特殊优点包括：

- 远程通信。消息传递允许不同的应用能够通信与传递数据。位于相同进程中的两个对象可以完全共享内存中相同的数据。而把数据从一台计算机发送到另一台计算机比这复杂得多，数据需要复制。这意味着对象必须是“可串行化的（serializable）”，也就是说，它们能被转换成简单的字节流，并通过网络传送。消息传递会负责这种转换，因此应用无需考虑这个问题。如果不需要远程通信，就不需要消息传递系统。可以采用更简单的解决方案，如并发集合或共享内存即可。
- 平台/语言的集成。当通过远程通信连接多台计算机系统时，这些计算机系统可能使用了不同的语言、技术或平台，因为它们很可能是在不同时间由不同的小组开发完成的。集成这样的异构应用需要通过一个中间件中性区进行协商，一般使用最小公共集，如无格式的平面数据文件。在这样的环境中，消息传递系统成为应用之间彼此协作的通用转换器，它能与使用不同语言、平台的应用打交道，让它们能采用公共的消息范例传递集合实现相互通信。通用连接特性就是消息总线模式的核心。
- 异步通信。消息传递采用发送并忽略的方式进行通信。发送者不必等待接收者接收和处理消息，甚至不必等待消息传递系统传送消息。发送者只需等待消息发送，即由消息传递系统将消息成功存储在通道中。一旦消息存储完毕，发送者就可以执行其他操作，同时消息由后台传送。接收者可能要向发送者返回消息的应答或处理结果，这是通过另一个消息完成的，发送者通过回调机制检测此消息是否到来。
- 可变的定时机制。采用同步通信时，调用者必须等待接收者完成调用处理才能接收到结果并继续运行。使用这种方法，调用者调用的速度是由接收者的执行速度决定的。与之相比，异步通信让发送者可以按照自己的速度把请求批量发送给接收者，接收者也可按照自己的速度处理请求。这就使得所有应用均能以最大吞吐量运行，不会因等待彼此而浪费时间（至少在接收者处理完所有消息前）。
- 节流。远程过程调用（remote procedure call，RPC）最大的缺点就是当单个接收者需要同时处理过多的调用时，可能负载过大。这会导致性能的下降，甚至让接收者崩溃。异步通信使接收者能控制处理请求的速度，使其不会因过多的并发请求过载。这种节流机制对调用者产生的副作用是很小的，因为通信是异步的，调用者不会因等待接收者而阻塞。
- 可靠的通信。消息传递提供了远程过程调用所不具备的可靠传输。合理的消息传递比 RPC 更可靠，因为消息传递使用了存储转发方式传送消息。数据被打包成消

息，它们是原子的、独立的单元。当发送者发送一个消息时，消息传递系统把消息保存起来，然后把消息转发给接收者所在的计算机，并再次保存。把消息存储在发送者和接收者的计算机上，这就增强了通信的可靠性（为了更为可靠，消息可以存储在磁盘中，而不是内存中，请参考“可靠传输”的介绍）。真正不可靠的是将消息从发送者的计算机上转移到接收者的计算机上，因为接收者或网络可能无法正常工作。消息传递系统通过重传消息直至成功的方式来克服这一问题。通过自动重试，使消息传递系统能够克服网络方面的问题，发送者和接收者不必再担心这些细节。

- 无连接运行。一些应用通过专门设计，能够在网络不连通的情况下运行，当网络连通时再与服务器完成数据同步。这样的应用配置在笔记本电脑、PDA 和汽车仪表盘这样的平台上。消息传递系统特别适合这些应用进行数据同步——要同步的数据创建后首先排队，等待应用重新连接网络。
- 仲裁。如仲裁者模式（mediator pattern [GoF]）所示，消息传递系统在所有能够发送与接收消息的应用中扮演了仲裁者的角色。应用可以使用消息传递系统作为与其他应用或服务进行集成的指南。如果一个应用与其他应用断开了连接，它只需与消息传递系统重建连接，而不必与其他所有进行消息传递的应用建立连接。消息传递系统可以为共享资源（如数据库）提供大量分布式连接。消息传递系统还可以使用冗余资源提供高可用性、负载平衡、失败网络连接的重新路由，并能调整服务的性能和质量。
- 线程管理。异步通信意味着，一个应用在等待其他应用完成任务时，除非有必要，自己不必阻塞。调用者可以使用回调函数，通过回调获得应答消息的通知（请参见请求/应答模式）。如果存在大量阻塞的线程，或者线程被阻塞很长时间，这些都会带来问题。太多的阻塞线程会导致应用再没有可用的线程来执行真正的任务。如果有大量阻塞线程的应用崩溃了，那么当应用重新启动并恢复以前的状态时，将很难重建这些线程。而采用回调函数，只是少量等待响应的监听线程会阻塞。这就为完成其他任务留下了大量可用的线程，并且当程序崩溃后，重建这些数量确定的监听线程并不困难。

因此，许多理由可以证明，应用或企业可以从消息传递得到好处。有些理由是与应用开发人员密切相关的技术细节，还有一些理由是得到企业架构师共鸣的战略决策。到底哪个理由最重要，是由特定应用所处的当前环境决定的。这些都能充分说明为什么使用消息传递，不论参考哪一条都能获得最大的收益。

异步消息传递带来的挑战

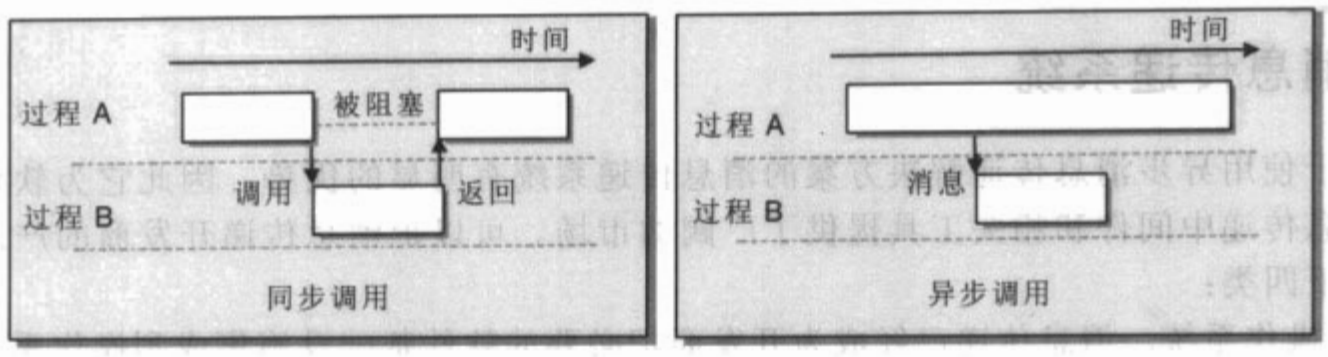
异步消息传递并不能解决所有集成问题。它能很好地解决集成异构系统时面临的大量难题，但是也带来了新的挑战。其中一些挑战是异步通信模式中固有的，而另外一些挑战与消息传递系统的特定实现相关。

- 复杂的编程模式。异步消息传递需要开发人员采用事件驱动的编程模式。应用逻辑不再只写在一个方法中（并由这个方法调用所有其他方法）；如今，应用逻辑被分解到多个事件处理方法中，由这些方法来响应接收到的消息。这样的系统变得更加复杂，更加难于开发与调试。例如，一个简单的方法调用变成请求消息、请求通道、应答消息、应答通道、关联标识符和一个非法消息队列的组合（详见请求/应答模式）。
- 序列问题。消息通道可以保证消息的传递，但是不能保证传递消息的时机。这就会导致发送时有序的消息最后可能乱序。当消息之间相互关联时，重建消息序列则尤其重要（详见重排模式）。
- 同步问题。并不是所有的应用都能按照发送并忽略方式运行。如果用户要查询机票，他们希望能立即看到机票价格，而不是再等待不确定的时间。因此，许多消息传递系统必须综合采用同步与异步解决方案。
- 性能。消息传递系统确实会增加通信的开销。它要把应用数据封装到消息中并发送，还要接收并处理消息。如果需要传递大块的数据，那么把数据分割成无数的小消息并不是明智之举。例如，某集成解决方案必须同步两个已有系统的信息，首先它要把所有相关的信息从一个系统复制到另一个系统中。对于大块数据的复制操作，ETL（提取、转换和加载）工具要比消息传递更有效。消息传递更适合用于初始数据复制完成后的系统同步。
- 有限的平台支持。许多专用的消息传递系统不能适用于所有平台。通常，通过 FTP 发送文件是唯一可行的集成方案，因为目标平台可能不支持消息传递系统。
- 开发商锁定。许多消息传递系统的实现依赖于专用的协议。即使是公共的消息传递规范，如 JMS，也没有规定解决方案的具体实现。因此，不同的消息传递系统通常不能互连。这就给我们带来了新的集成挑战：需要集成多种集成解决方案！（详见消息传递桥模式）。

总之，异步消息传递并不能解决所有问题，甚至会带来新的问题。在决定哪些问题需要用消息传递来解决时，请记住它们可能带来的后果。

异步思想

消息传递是一种异步技术，消息可以反复传送直至成功。与此不同，大多数应用采用的是同步函数调用。例如，一个过程调用子过程，一个方法调用另一个方法，或者一个过程通过 RPC（如 CORBA 和 DCOM）远程调用另一个过程。同步调用是指调用进程在子进程执行被调函数时停止运行。即使在 RPC 中，被调用的子过程在其他进程中执行，调用者也会阻塞，直到子过程把控制权（和结果）返回给调用者。反之，当使用异步消息传递时，调用者使用“发送并忽略”的方法，因此它能在发送完消息后续续执行。因而当调用子过程后，调用过程会继续运行（见下图）。



同步与异步调用语义

异步通信有许多含义。首先，不再只是一个执行线程。多个线程使得子过程能同时执行，从而极大提高了性能，并可以确保当一些子进程等待外部结果时另外的子进程可继续执行。但是，并发的线程也增加了调试的困难。其次，结果（如果有的话）是通过回调函数返回的。这就使调用者能在结果返回前继续执行，从而改善了性能。但是，如果结果返回时调用者正在执行其他任务，也必须停下来处理结果，并且要记住发起调用时的进程上下文。再者，异步子进程的执行顺序是不确定的。而且，一个子过程无法执行并不意味着另外的子过程不能执行。这就要求子进程可以按任意的顺序独立运行，调用者应当能确定哪些结果来自于哪子进程，并将它们合并起来。因此，异步通信有诸多优点，但是需要重新考虑过程如何调用其子过程（译注 1）。

分布式应用与集成

本书介绍的是企业集成，即如何集成独立的应用，使它们能协调工作。一个企业应用通常有 n 层体系结构（属于更加复杂的客户/服务器体系结构），所以它能分布在多台不同的计算机上。即使这种结构会导致不同机器上的进程相互通信，但它还是属于应用分布，而不是应用集成。

为什么 n 层体系结构被认为是应用分布，而不是应用集成呢？首先，它的通信部件是紧耦合的，它们之间相互直接依赖，其中任何一层离开其他层都会失去作用。其次，层间一般采用同步通信。再者，（n 层或基本）应用的用户一般只接受快速的系统响应。

与之相反，所集成的应用都是独立的应用，它们能各自运行，还能采用松耦合的方式相互配合。这使得每个应用都能集中完成一部分具体的功能，并将相关的功能委托给其他应用来完成。所集成的应用采用异步通信方式，它们不必等待响应就能继续执行；或者并发地执行其他任务，直到应答返回。集成的应用对时限的要求更宽松，这就使它们能够处理其他任务直到有调用结果返回，因此要比很多实时等待结果的用户更有耐心。

译注 1：原文中“subprocess”和“subprocedure”交替出现，分别译为“子进程”和“子过程”，相对来讲，“子过程”和“过程”是一种静态概念，“子进程”和“进程”还涵盖动态行为。不过，在这里二者并没有显著差别。

商业消息传递系统

由于使用异步消息传递解决方案的消息传递系统有明显的优势，因此它为软件开发商创建消息传递中间件和相关工具提供了广阔的市场。可以把消息传递开发商的产品大致划分为以下四类：

1．操作系统。消息传递已经成为开发商把必要的软件基础设施集成到操作系统或数据库平台中的共同需要。例如，Microsoft Windows 2000 和 Windows XP 操作系统包含了 Microsoft 消息排队（Microsoft Message Qneuing，MSMQ）服务软件。这个服务可通过许多 API（包括 COM 组件和 System.Messaging 名字空间）访问，它属于 Microsoft .NET 平台的组成部分。类似地，Oracle 提供了 Oracle AQ，作为其数据库平台的组成部分。

2．应用服务器。Sun Microsystems 公司首先是在 J2EE 1.2 版规范中集成了 Java 消息传递服务（Java Messaging Service，JMS）。从那以后，所有的 J2EE 应用服务器（如 IBM WebSphere 及 BEA WebLogic）都提供了该规范的具体实现。而且，Sun 还在 J2EE JDK 中提供了 JMS 的参考实现。

3．EAI 套件。由这些开发商提供的产品是专用的，但是包含了丰富的功能，如消息传递、业务流程自动化、工作流、信息汇总等等。在这个市场中 IBM WebSphere MQ、Microsoft BizTalk、TIBCO、WebMethods、SeeBeyond、Vitria、CrossWolds 和其他一些公司扮演了重要的角色。许多产品都包含了 JMS，把它作为向客户提供的一部分 API，而其他像 Sonic，Software 和 Fiorano 这样的开发商则把重点集中在实现与 JMS 兼容的消息传递基础设施上。

4．Web 服务工具集。Web 服务已经在企业集成社群中引起了极大关注。各个标准组织和协会正在对使用 Web 服务进行可靠的消息传递制定通用标准（例如 WS-Reliability、WS- ReliableMessaging 和 ebMS）。越来越多的开发商开始提供工具来实现路由、转换以及基于 Web 服务的解决方案的管理。

本书介绍的模式独立于具体的开发商，适用于大多数消息传递解决方案。遗憾的是，每个开发商在描述消息传递解决方案时定义了各自的术语。本书所使用的模式名称将在技术和产品方面，甚至在描述方面尽可能保持中立，以利于交流。

许多消息传递开发商在其产品中融合了本书所介绍的部分模式的特性，这些产品简化了模式的应用，促进了解决方案的发展。对于熟悉特定开发商术语的读者来说，很可能已经了解本书介绍的许多概念。为了帮助这些读者将模式语言与开发商的特定术语关联起来，下表列出了大多数通用的模式名称及在消息传递产品中广泛使用的与之相应的产品特性名称。

企业集成模式	Java 消息服务（JMS）	Microsoft MSMQ	WebSphere MQ
消息通道（Message Channel）	目标（Destination）	消息队列（MessageQueue）	队列（Queue）
点对点通道 （Point-to- Point Channel）	队列（Queue）	消息队列（MessageQueue）	队列（Queue）

续表

企业集成模式	Java 消息服务（JMS）	Microsoft MSMQ	WebSphere MQ
发布-订购通道（Publish-Subscribe Channel）	主题（Topic）	—	—
消息（Message）	消息（Message）	消息（Message）	消息（Message）
消息端点（Message Endpoint）	消息生产者，消息消费者（MessageProducer，MessageConsumer）		

企业集成模式	TIBCO	WebMethods	SeeBeyond	Vitria
消息通道（Message Channel）	主体（Subject）	队列（Queue）	智能队列（Intelligent Queue）	通道（Channel）
点对点通道（Point-to-Point Channel）	分布式队列（Distributed Queue）	传送动作（Deliver Action）	智能队列（Intelligent Queue）	通道（Channel）
发布-订购通道（Publish-Subscribe Channel）	主体（Subject）	发布-订购动作（Publish-Subscribe Action）	智能队列（Intelligent Queue）	发布-订购通道（Publish-Subscribe Channel）
消息（Message）	消息（Message）	文档（Document）	事件（Event）	事件（Event）
消息端点（Message Endpoint）	发布者、订购者（Publisher, Subscriber）	发布者、订购者（Publisher, Subscriber）	发布者、订购者（Publisher, Subscriber）	发布者、订购者（Publisher, Subscriber）

模式的描述形式

本书介绍的是一种模式语言，它是由一组模式组成的。和《Design Patterns》、《Pattern Oriented Software Architecture》、《Core J2EE Patterns》和《Patterns of Enterprise Application Architecture》这些书一样，本书旨在普及如何运用模式来统一计算机编程技术。Christopher Alexander 首先在他的专著《A Pattern Language》和《A Timeless Way of Building》中提出了模式及其模式语言的概念。每种模式代表了一种决定，以及做出这一决定需要考虑的相关事项。模式语言将相关的模式编织成网，使每种模式均能与其他模式相关，帮助读者做出决策。采用这种方式可以将专家的经验记录下来，让其他人能方便地理解和应用这些经验。

针对确定问题空间中不断涌现的问题，模式语言向你提供了加以解决的方法。由于每次要解决的问题均不相同，因此如何应用这些模式以及应用模式的方式也是不同的。所以本书所面向的是在各种应用中使用各种消息传递工具的人，对于你以及你所面对的特定消息传递应用都有帮助。

如果只在书中列出模式的描述，这显然还不能让读者获得丰富的知识。简单地说“当你面对这个问题时使用这个解决方案”是远远不够的。对真正想掌握模式的读者来说，必须说明解决问题的难点在哪里，要介绍各种在实际应用中可能不是很好的解决方案，解释为什么这里提出的解决方案是最好的。同样地，还要把模式之间的关系讲清楚，帮助读者从一个问题深入到下一个问题。用这种方法，模式的描述形式不但能用于教会读者如何选择已有的解决方案，更重要的是还提供了方法，你可以拿这些方法来解决作者尚未遇到的新问题。这些就是我们创作本书的目的。

模式应该是规范的，即它们应该能告诉你做什么。模式不但要描述问题，描述解决问题的方法，还要告诉你具体的解决步骤。每种模式代表了一种你要做出的决定：“我应该使用消息传递吗？”“采用命令消息能给我提供帮助吗？”。模式和模式语言的关键是帮助你做出解决特定问题的适当决定，即使作者并不了解这些特殊的问题是什么，即使你还不具备自己开发解决方案的知识和经验，也能从中获益。

不存在通用的模式描述形式，不同的书使用的描述结构是不同的。我们采用了与Alexandrian相似的描述形式，这种形式首先是由Kent Beck在计算机编程书《Smalltalk Best Practice Patterns》中提出的。之所以使用Alexandrian描述形式是因为用它描述模式不会受太多的约束。尽管介绍每种模式采用了良定义的相同结构，但不必为每个独立的部分设定标题，因此不会打断所要叙述的内容。为了便于阅读，这种格式使用了下划线、缩排和图片等排版方法，以帮助读者快速找出重要的信息。

每种模式采用了以下的结构来描述：

- 名称——模式的标识符，指示了这种模式会做什么。我们精心选择了模式名，以便在描述中使用，使设计人员能在交流中引用所述模式的概念。
- 图标——大多数模式不但有名称，还有相关的图标。因为许多架构师往往采用图形进行可视化的交流，所以除了书面语言外，我们还提供了一种可视化的语言。这种可视化的语言增强了模式的组合能力，在描述大型、复杂的模式解决方案时可将多个模式图标组合起来。
- 上下文——解释遇到哪些情况时可以采用这种模式帮助你解决问题。上下文给出了问题的解决步骤，而且指出了可能用到的其他模式。
- 问题——以提问的方式描述可能碰到的困难。读者应该阅读这些提问，并尽快确定这种模式是否与你的工作有关。我们把问题用一句话描述，并用水平线区分出来。
- 冲突——冲突是指导致问题难以解决的限制因素。在这一部分往往会考虑一些看起来很有希望，但是实际不可行的替代解决方案，由此可以体现出真正的解决方案的价值。
- 解决方案——这一部分给出了解决问题的方法。它不但适用于特殊情况，还给出了在与这个问题相关的各种环境中该怎么做。如果理解了模式要解决的问题和解决方案，就能理解模式本身。解决方案采用了与问题相同的描述方式，以便读者能从书中快速找出这些问题和解决方案。

- 草图——Alexandrian 描述形式的最大优点在于每种模式均包含一幅说明解决方案的草图。在很多情况下，通过查看模式名和它的草图就能理解模式的基本含义。因此，我们在给出每种模式的解决方案后，尽量插入一幅解决方案的图片或草图来保持这种描述风格。
- 结果——在解决方案的基础上进一步给出如何使用解决方案的详细方法，以及如何应对可能的冲突。同时还提出采用这种模式可能面对的新挑战。
- 下一步——给出在应用当前模式后需要考虑的其他模式。模式不是独立存在的，使用一种模式常常会导致新的问题，这些问题需要其他模式解决。正是因为模式之间存在着这种关系，所以模式构成的是一个语言，而不只是一个模式名录。
- 其他选项——这一部分更详细地讨论模式的技术问题或变化。我们把这部分与其他部分明显区分出来，如果这些内容与你的特定应用无关，就能很容易跳过这个部分。
- 实例——每种模式往往包含一个或多个正在使用或已经使用过的模式例子。简单的例子可能只是对我们原本已经熟悉的用法指定一个模式名，而详细的例子可能包含一大段代码。由于存在大量可用的消息传递技术，我们并不指望读者能熟悉实例中使用的每种技术。因此，我们对模式做了精心的设计，即使你不看例子，只需阅读模式的介绍，也不会遗漏任何关键的内容。

采用模式描述解决方案的优点在于，它不但能帮助你解决书中所介绍的具体问题，还能帮助你进行创新设计，解决作者也未遇到过的问题。因此，这些消息传递模式不但描述了已有的消息传递系统，还能应用于本书出版后才创建的新的消息传递系统中。

图形符号

集成解决方案是由许多不同的部分组成的：应用、数据库、通信端点、通道、消息、路由器等。如果要描述一个集成解决方案，必须定义一种涵盖所有这些不同组件的符号。据我们所知，目前还没有一种广泛使用的符号表示能够描述集成解决方案的所有方面。统一建模语言（Unified Modeling Language，UML）对描述面向对象的系统非常有用，它提供了类图和交互图，但是并没有提供描述消息传递解决方案的语义。针对 EAI 的 UML 规范[UMLEAI]增加了协作图语义，能够描述组件之间的消息流。作为一种精确的系统可视化描述语言，这种符号非常有用，可以作为代码生成的基础，这是模型驱动的体系结构（Model-Driven Architecture，MDA）的一部分。我们之所以不使用这种符号有两个原因。第一，UML 规范无法涵盖用模式语言描述的所有模式。第二，我们不想创建精确的可视化描述，而只是采用大致能满足模式描述要求的草图。希望这些图片能够让读者立即领会模式的内涵，就像 Alexander 的“sketch（草图）”一样。所以我们创建了自己的符号。幸运的是，与其他更正规的符号不同，采用这些自定义的符号不需要阅读大量的参考手册。只用一个类似如下所示的简单的图就够了。



消息传递解决方案的可视化符号

这种简单的图片表现出通过通道把消息发送给组件的过程。在这里，组件的含义非常广泛——它可以代表要集成的应用、负责转换或路由应用间消息的中间部件，或一个应用的一个特定部分。有时，为了突出通道本身，还会把通道描绘成三维的管道。组件往往是我们感兴趣的重点，因此，通道可以简单地绘制成带箭头的线段，这两种表示都是等同的。消息可以表示成以圆形作为根，以嵌套的方框作为其元素的小树。因为许多消息传递系统允许消息中包含树型结构的数据，例如 XML 文档，在特定的模式中，树中的元素还可以着色或增添阴影，以突出其用途。因此，采用这种描述方式就能为转换模式提供快速的可视化描述——能方便地显示出向消息中增加、重排、删除字段的模式。

当我们描述应用设计时——如消息传递端点或采用 C#或 Java 编写的例子——可以使用标准的 UML 类图和顺序图来描述类的层次结构及其对象之间的交互。这是因为 UML 符号是一种受到广泛认可的标准，可以用于描述这种解决方案（如果你是 UML 的初学者，请参考[UML]）。

实例和中场演练

为了尽可能扩大这些模式的应用范围，书中包含了使用各种集成技术的实例。这种做法有一个不足，就是读者可能并不熟悉例子中使用的某种技术。我们也不要求读者必须阅读这些例子，在模式的描述中已经包含了所有关键内容。因此，读者跳过这些例子并不会遗漏任何重要的细节。另外，在可能的情况下，我们还采用了不同的技术实现了实例的不同版本。

提供实例代码时，我们关注的是代码的可读性而不是可运行性。代码段可以帮助读者消除对解决方案描述的模糊认识，许多应用开发人员和架构师更愿意阅读 30 行左右的代码，而不愿阅读大量的文本。为了满足这些需要，我们往往只给出了大型解决方案中与模式最相关的方法或类。同时我们省略了大量的错误检查代码，以突出代码所实现的核心功能。很多代码段不包含行间注释，因为已经在代码前后的段落做出了解释。

为单个集成模式提供一个有意义的例子并不容易。企业集成解决方案通常包括许多不同类型的组件，这些组件分布在多个系统上。同样地，大多数集成模式并不是独立运行的，而是与其他模式一起构成实际的解决方案。为了突出多个模式之间的协作关系，我们在本书的主要章节之间提供了更完整的例子作为中场演练（见第 6 章、第 9 章和第 12 章）。这些解决方案给出了设计更全面的消息传递解决方案时所做的大量折衷考虑。

目录

第 1 章　运用模式解决集成问题 1
1.1　集成的需要 1
1.2　集成面临的挑战 2
1.3　集成模式有何帮助 3
1.4　集成世界 4
1.5　松耦合 6
1.6　EAI 简介 7
1.7　一个松耦合的集成解决方案 10
1.8　Widget & Gadgets'R Us 公司：示例 11
1.9　小结 26

第 2 章　集成方式 27
2.1　引言 27
2.2　文件传输 29
2.3　共享数据库 31
2.4　远程过程调用 33
2.5　消息传递 35

第 3 章　消息传递系统 38
3.1　引言 38
3.2　消息通道 39
3.3　消息 43
3.4　管道和过滤器 46
3.5　消息路由器 52

3.6 消息转换器 57
3.7 消息端点 65

第 4 章 消息传递通道 67
4.1 引言 67
4.2 点对点通道 69
4.3 发布-订购通道 71
4.4 数据类型通道 75
4.5 非法消息通道 77
4.6 死文字通道 80
4.7 可靠传输 82
4.8 通道适配器 85
4.9 消息传递桥 89
4.10 消息总线 91

第 5 章 消息构造 95
5.1 引言 95
5.2 命令消息 96
5.3 文档消息 97
5.4 事件消息 100
5.5 请求/应答 102
5.6 返回地址 105
5.7 关联标识符 107
5.8 消息序列 112
5.9 消息到期 116
5.10 格式指示符 118

第 6 章 中场演练：简单的消息传递 121
6.1 引言 121
6.2 JMS 请求/应答示例 122
6.3 .NET 请求/应答示例 132
6.4 JMS 发布-订购示例 139

第 7 章 消息路由 155
7.1 引言 155
7.2 基于内容的路由器 158
7.3 消息过滤器 163

7.4 动态路由器 167
7.5 接收表 171
7.6 分解器 179
7.7 聚合器 187
7.8 重排器 199
7.9 复合消息处理器 208
7.10 分散收集器 210
7.11 路由表 212
7.12 过程管理器 219
7.13 消息代理 226

第 8 章 消息转换 230
8.1 引言 230
8.2 信封包装器 232
8.3 内容扩充器 236
8.4 内容过滤器 240
8.5 声明标签 243
8.6 规范器 247
8.7 规范数据模型 249

第 9 章 中场演练：复合消息传递 254
9.1 贷款中介示例 254
9.2 使用 Web 服务的同步实现方案 260
9.3 使用 MSMQ 的异步实现方案 287
9.4 使用 TIBCO ActiveEnterprise 的异步实现方案 327

第 10 章 消息传递端点 342
10.1 引言 342
10.2 消息传递网关 344
10.3 消息传递映射器 352
10.4 事务性客户端 356
10.5 轮询消费者 364
10.6 事件驱动的消费者 366
10.7 竞争消费者 370
10.8 消息分派器 374
10.9 选择性消费者 380
10.10 持久订购者 385

10.11 幂等接收者 389
10.12 服务激励器 392

第 11 章 系统管理 395
11.1 引言 395
11.2 控制总线 396
11.3 旁路 399
11.4 线路分接器 400
11.5 消息历史 402
11.6 消息存储库 405
11.7 智能代理 407
11.8 测试消息 416
11.9 通道清洁者 417

第 12 章 中场演练：系统管理示例 421
12.1 贷款中介系统的管理 421

第 13 章 实战集成模式 443
13.1 案例研究：债券定价系统 443

第 14 章 结束语 458
14.1 企业集成的新兴标准和展望 458

参考文献 475

第 1 章

运用模式解决集成问题

本章介绍如何使用书中的模式解决各种集成问题。为此，首先要考虑各种常见的集成场景，并给出一个完整的集成示例。在设计这个示例的集成解决方案时，我们会用书中包含的模式加以表述。读完本章你会对二十多个集成模式有所了解。

1.1 集成的需要

企业一般都由成百上千个应用组成，这些应用要么定制，要么由第三方获得，要么是遗留系统的一部分，也可能是上述应用的组合，它们运行在不同操作系统平台的多个层面上。在企业中完全有可能发现 30 个不同的 Web 站点、3 个 SAP 实例和无数的部门解决方案，这种情况并不少见。

也许我们会问：是什么原因造成了这种混乱局面？企业的体系结构如此混乱不堪，难道没有哪个 CIO 会因此被解聘吗？不错，凡事均事出有因。

首先，创建商业应用很不容易。创建一个独立的、大型的应用来完成一个企业的所有业务处理几乎是不可能的。ERP（Enterprise Resource Planning，企业资源计划）开发商有过一些开发更大商业应用的成功案例。但是，即使是重量级的开发商，如 SAP、Oracle、Peoplesoft 等，也只是实现了典型企业所需的部分业务功能。ERP 系统是目前企业中最常见的集成系统之一，从这一点也很容易发现创建商业应用的艰辛。

其次，把业务功能分散到多个应用中，这为企业提供了很大的灵活性，可以选择“最好”的财会开发包、“最好”的客户关系管理系统以及“最好”的订单处理系统来满足自己的需要。IT 组织通常对那种“一站式”的企业应用并不感兴趣，况且企业的需求名目繁多，要想提供这样一个“无所不能”的应用也不太可能。

开发商开始学着迎合这种需要，着力提供围绕一个特定核心功能的针对性应用。但是，客户不断提出要对已有的软件包添加新的功能，这就造成了一些集成商业应用的功能泛滥。比如，许多账单管理系统开始加入客户关怀和财会功能。同样，客户关怀软件的开发者会在软件中加入一些简单的账单管理功能，如处理账单纠纷或调解。要在系统之间划定清晰的功能界限是很困难的：如果客户对账单有异议，处理这种纠纷到底应该算是客户关怀功能还是账单管理功能呢？

用户（如客户、商业合作伙伴和内部用户）在使用商业应用时，一般并不考虑系统的

边界。他们只是执行业务功能，而不管这种业务功能是由多少个内部系统实现的。例如，某客户可能要求修改他（或她）的地址，并查看最近的付款是否到账。在许多企业中，这种简单的请求可能会涉及客户关怀和账单管理系统。类似地，客户下新的订单可能需要多个系统的协作。企业需要验证客户 ID，检查客户的信用等级，检查库存，履行订单，获得货运报价，核算营业税，交付账单等等。这个过程通常会跨越五到六个不同的系统。但从客户的角度看，这只是一个单独的商业交易。

为了支持常见的商业过程和多个应用间的数据共享，必须把这些应用集成起来。应用集成必须在多个企业应用之间提供高效、可靠和安全的数据交换。

1.2 集成面临的挑战

遗憾的是，企业集成不是一项简单的任务。由定义可知，企业集成要与运行在不同地点、不同操作平台上的多个应用交互，称之为“简单集成”显然是有矛盾的。软件开发商提供了 EAI（Enterprise Application Integration，企业应用集成）套件，能提供跨平台、跨语言的集成，并能与许多封装好的（packaged）流行商业应用交互。但是，企业集成如此错综复杂，这项技术作为基础设施只解决了其中的一小部分。实现跨行业、跨技术的集成才是真正的挑战。

- 企业集成要求公司策略大幅调整。商业应用一般会强调某个特定的功能领域，如客户关系管理（Customer Relationship Management，CRM）、账单管理、金融管理等。这一点似乎是对著名的 Conway 定律的扩展：“设计系统的组织会根据组织的通信结构来得出设计”。因此，许多 IT 小组就是依据这些功能领域组建的。成功的企业集成不但要在多个计算机系统之间建立通信，还要在业务单位和 IT 部门之间建立通信——在一个集成的企业应用中，特定的应用不再由小组控制，因为每个应用已经成为整个集成应用和服务流的一部分。
- 由于集成的覆盖面很广，通常会对商业应用产生深远影响。一旦把最重要的业务功能加入到一个集成方案中，这个解决方案能否正确地履行功能对商业应用至关重要。如果集成方案失败，或者表现不正常，有可能使企业丢失订单、支付有误，引起客户的不满，因而损失几百万美元。
- 开发集成解决方案最大的限制在于，集成开发人员对所集成的各个应用只有有限的控制。在大多数情况下，这些应用都是“遗留”系统或者是封装好的应用，不会因为要连接到集成方案而作出修改。这就迫使集成开发人员不得不修补应用存在的缺陷，增加新的特性，或者弥补它们之间存在的差异。如果能在应用端点内部实现部分解决方案会更容易一些，但是出于政治或技术上的原因，这通常是做不到的。
- 尽管对集成解决方案存在着广泛的需求，但是这个领域内的标准还很少。XML、XSL 和 Web 服务的出现是集成解决方案向标准化迈进的重要标志。但是，对 Web

服务的大肆宣传也给市场带来了新的格局，这又造成关于标准的新的“扩展”和“实现”到处泛滥。需要注意的是，尽管 CORBA 力图为系统集成提供一个全面的技术解决方案，但是与标准兼容的产品之间缺乏互操作性，已经成为 CORBA 的主要绊脚石之一。

- 已有的 XML Web 服务标准只是解决了一部分集成问题。例如，通常会标榜 XML 是系统集成的万能钥匙，这其实存在着误导。把所有数据交换都标准化为采用 XML 格式，这就如同将所有文档都采用一种通用字母表（如罗马字母）来书写。即使字母表是相同的，但仍然可以用来表示不同的语言和方言，所以并非所有读者都能读懂。企业集成也会面临同样的问题。采用相同的表示方式（如 XML）并不代表一定提供相同的语义。“账户（account）”这个词在所集成的各个系统中可能有不同的含义和语义，存在不同的限制和假设。要在系统之间克服语义方面的差异，这是一项非常困难而耗时的任务，因为它涉及到重大的商业和技术决策。
- 开发一个 EAI 解决方案本身就是一项艰难的任务，运行和维护这样的解决方案可能会更加困难。在此会混合大量不同的技术，再加上 EAI 解决方案的分布特性，因此，要完成复杂任务的部署、监控和故障排除需要各种各样的技能。在大多数情况下，这些技能是由不同的人掌握的，甚至并非 IT 行业内部的人。

任何经历过 EAI 部署的人员都知道，EAI 解决方案是当前企业策略的一个关键部分，但是它并没有让 IT 人员更轻松，而是更增加了工作难度。从企业集成的高层目标（用直通处理、T+1、敏捷企业等术语定义）到完成具体的实现（例如 System.Messaging.XmlMessage-Formatter 还要取哪些参数？），这中间有很长的一段路要走。

1.3 集成模式有何帮助

如何实现企业集成，对此没有简单的答案。在我们看来，如果有人说集成很容易，这些人要么是难得的天才（至少比我们这些人更聪明），要么就是少有的白痴（这还是乐观的说法），也有可能他们想让你相信集成很简单，以便向你兜售他们的产品而从中获利。

尽管集成是一项覆盖面广、难度很大的工作，仍然有些人确实比其他人做得好。为什么这些人掌握的知识比别人多？显然所谓“21 天集成速成”是不可能的（本书肯定做不到！），因此这些人也不可能知道所有集成问题的答案。但是，他们曾经解决过大量集成问题，能把新问题与已经解决过的老问题进行比较。他们了解问题的“模式”及其相应的解决方案。他们经过长时间的摸索，通过尝试与失败，或者从其他有经验的集成架构师那里取经，才学到了这些模式。

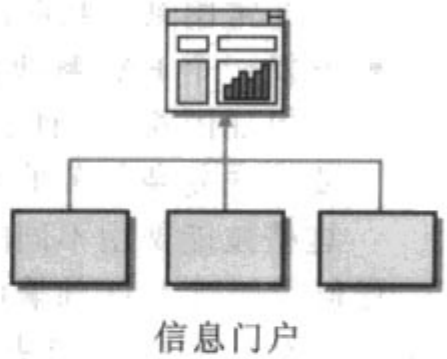
模式并不是可以复制粘贴的示例代码，也不是塑膜包装的组件，而是一些宝贵的建议，描述了针对一些频繁出现的问题有哪些解决方案。如果使用得当，集成模式有助于填补集成的高层目标与具体系统实现之间的鸿沟。

1.4 集成世界

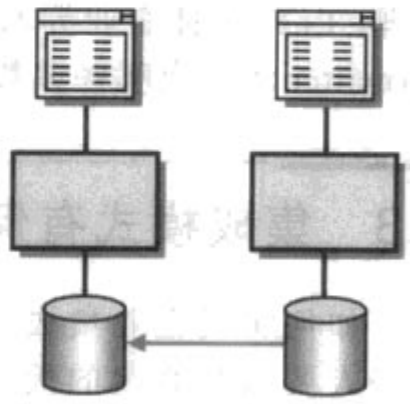
我们有意给集成下一个很宽泛的定义。对我们来说，集成（integration）就是连接计算机系统、公司或人。采用这种宽泛的定义，就能把感兴趣的内容添加到本书中，还有助于近距离地观察一些最常见的集成方案。在许多集成项目中，我们反复地遇到以下六种类型的集成：

- 信息门户
- 数据复制
- 共享的业务功能
- 面向服务的体系结构
- 分布式的业务过程
- 企业到企业（B2B）的集成

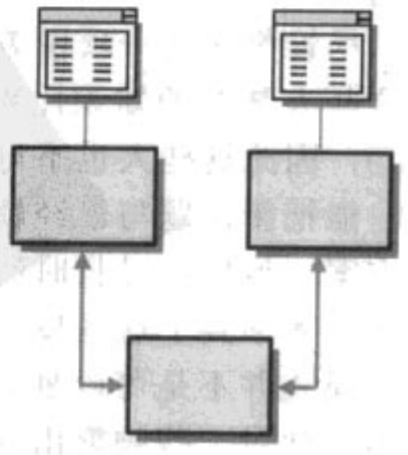
上述清单并没有全面地列出各种类型的集成，但是这确实可以说明集成架构师建立的解决方案。许多集成项目都结合了多种类型的集成。例如，为了把应用组合成一个单个的、分布式的业务过程，往往需要引用数据复制。


信息门户

许多商业用户必须访问多个系统才能得到某个问题的答案，或者完成一项商业活动。例如，为了检查一个订单的状态，客户服务代表必须访问主机上的订单管理系统，还要登录管理网上订单的系统。信息门户能把信息从多个信息源集中到一个显示画面上，以避免用户访问多个系统才能得到信息。简单的信息门户把屏幕划分成多个区，每个区显示来自不同系统的信息。更为复杂的系统还会在区之间提供有限的交互：比如，当用户从位于A区的某个列表中选择一项时，B区将刷新显示与这一项相关的详细信息。其他的信息门户还能提供更复杂的用户交互，这使得信息门户与集成应用之间的界限不那么泾渭分明。

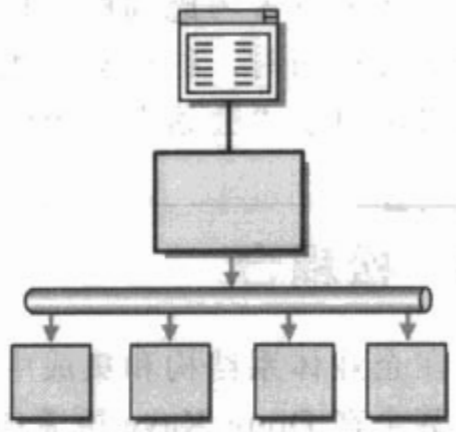

数据复制

许多商业系统需要访问相同的数据。例如，客户地址信息可能会同时用在客户关怀系统（当客户要求变更地址时）、账户管理系统（用于计算营业税）、货运系统（在发货单中填写）和账单管理系统（用于开具发货单）中。许多这样的系统都会有自己的数据存储库，用来存储与客户相关的信息。当客户要求变更其地址时，所有这些系统都需要各自修改客户地址的副本。基于数据复制来实现集成策略，就可以做到这一点。


共享的业务功能

要实现数据复制，有许多不同的方法。例如，一些数据库开发商在数据库中增加了复制功能；另外，可以把数据导出到文件中，再把它们重新导入到其他系统中，或者还可以使用面向消息的中间件利用消息传递数据记录。

许多商业应用不但保存有冗余的数据，类似地，它们可能还实现了冗余的功能。很多系统都要检查社会保险号是否合法、地址是否与相应的邮编匹配、指定的某个商品是否有库存。由此可见，把这些功能提供为共享的业务功能很有意义，这样只需实现一次，这些功能就能为其他系统服务。

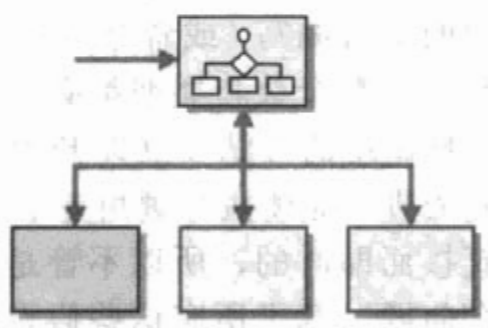


面向服务的体系结构

共享的业务功能类似于数据复制，可以满足同样的一些需要。例如，实现一个“取得客户地址”的业务功能后，其他系统就可以在需要时请求客户的地址，而不必永久地保存一个冗余的副本。具体采用哪种方法取决于很多因素，如我们对系统的控制能力（与把数据加载到数据库中相比，调用一个共享功能往往需要更多的权限），或者数据更新的频率（地址不常改变，但是要经常查询）。

共享的业务功能一般被称作服务。服务是一种定义明确的功能，它对外开放并响应来自“服务客户”的请求。一旦企业集成了一些有用的服务，管理这些服务就成为一项重要的功能。首先，应用需要某种形式的服务目录，它是所有可用服务的一个集中列表。其次，每个服务都要以某种方式提供接口描述，以便应用能与服务协商建立通信。这两项功能，即服务目录和协商功能，是建立面向服务的体系结构（Service-Oriented Architecture，SOA）的关键要素。

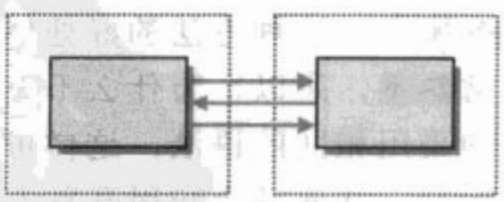
SOA 使集成与分布式应用之间的界限更加模糊。可以使用已有的远程服务（由其他应用提供）开发新的应用。因此，调用服务可以看作是两个应用之间的集成。然而，大多数 SOA 都提供了一些工具，利用这些工具，调用外部服务几乎与调用本地方法一样简单（暂不考虑性能），这样一来，基于 SOA 开发应用的过程就像是建立一个分布式应用。



分布式业务过程

单个业务交易往往涉及多个不同的系统，这是推动集成的一个关键因素。前面的例子指出，就连“下订单”这样的简单业务功能也很可能涉及六到七个系统。在大多数情况下，已有的应用中已经加入了所有相关的功能，缺少的只是这些应用之间的协调。因此，可以增加一个业务过程管理组件，由它管理如何跨越多个已有系统执行一个业务功能。

面向服务的体系结构与分布式业务之间的界限可能很模糊。例如，可以把所有相关的业务功能提供为服务，然后在一个应用中编写业务过程，通过 SOA 来访问所有服务。



企业到企业的集成

到目前为止，我们主要讨论的是单个企业内部应用与业务功能之间的交互。在很多情况下，外部供应商或商业合作伙伴也会提供业务功能。比如，货运公司可能会为客户提供一项服务，可以计算货运成本或跟踪发货。或者，商家

可以使用一个外部供应商来计算营业税率。同样地，在商业合作伙伴之间也经常出现集成。一个客户可能与某零售商联系，查询一件商品的价格以及是否有货。作为响应，零售商可能向供货商询问缺货商品的供货情况。

上述的许多考虑同样可以应用于企业到企业的集成。但是，通过 Internet 或其他一些网络进行通信时，往往会产生与传输协议和安全相关的新问题。而且，由于许多商业伙伴可能采用电子“会话”的合作方式，因此数据格式标准化非常重要。

1.5 松耦合

在企业体系结构和集成中使用最多的词就是松耦合（loose coupling）。这个术语非常流行，甚至在 Doug Kaye 所著的一本书[Kaye]中这个词贯穿全书。松耦合的好处早已被人们了解，但是直到 Web 服务体系结构迅速普及时，它们才真正得到重视。

松耦合有一个基本的核心，就是在协作双方（组件、应用、服务、程序、用户）交换信息时要减少对对方做出的假设。双方对对方及公共协议做出的假设越多，通信的效率就越高，但是这种解决方案往往无法忍受服务的中断或变化，因为通信双方被紧密耦合在了一起。

本地方法调用就是紧耦合的一个很好的例子。在应用中调用一个本地方法时，需要对调用和被调用的例程做许多假设。它们都必须运行在同一个进程（如虚拟机）中，并采用相同的语言编写（或者至少使用相同的中间语言或字节码）。调用方法必须准确提供所需的参数，参数个数和类型都必须符合约定。调用是即时的，也就是说，在调用方法做出调用后，被调用的方法要立即开始执行（这意味着，调用是同步的）。方法调用结束后，会自动恢复处理，继续执行调用方法中的下一条语句（即方法调用后面的语句）。方法之间的通信是直接而即时的，所以不管是调用方法还是被调用的方法，都不必考虑第三方窃听这样的安全问题。基于所有这些假设，编写良构的应用会非常容易，可以把功能划分到单个的方法中，再由其他方法调用。这样就会得到许多小方法，可以提高灵活性和重用性。

许多集成方法的目标是把远程数据交换打包，从而采用本地方法调用同样的语义，以简化远程通信。基于这种策略，远程过程调用（RPC）或远程方法调用（Remote Method Invocation，RMI）得以出现，许多流行的框架和平台都已经提供了对此的支持，包括 CORBA（见[Zahavi]）、Microsoft DCOM、.NET Remoting、Java RMI 以及最近出现的 RPC 风格的 Web 服务。这种方法的好处体现在两个方面。第一，应用开发人员对同步方法调用的形式非常熟悉，所以，为什么不充分利用已知的知识呢？第二，对本地方法调用和远程方法调用均采用相同的语法，这样可以延迟到部署时再决定哪些组件要在本地运行而哪些应该远程运行，从而减少应用开发人员的负担。

所有这些方法都要面对一个难题，由于存在远程通信，所以本地方法调用所依据的许多假设不再成立。因此，把远程通信抽象成一个简单的方法调用，会造成混乱和误解。Waldo 及其同事在 1994 年曾指出，“与在同一个地址空间中交互的对象相比，在分布式系统中交

互的对象在处理上存在本质的区别”[Waldo]。例如，如果我们调用远程服务完成某个功能，是不是真的想对这个服务采用的编程语言加以限制呢？非得采用我们使用的语言吗？一个跨网络的调用往往比本地调用慢好几个数量级。调用方法是否确实需要等到被调方法完成后才能执行呢？如果网络中断，被调方法暂时不可达怎么办？到底应该等待多长时间？怎样才能保证是与正确的一方通信，而不是冒充的第三方？如何防止窃听？如果被调方法的方法签名（即所需的参数列表）改变了怎么办？如果远程方法由第三方或商业伙伴维护，我们就无法控制这种改变。那么是让方法调用失败，还是尝试寻找最有可能的参数间映射继续执行调用呢？显然，远程集成带来了本地方法调用永远不会遇到的很多问题。

总之，尝试把远程通信描述成一种本地方法调用是自找麻烦。这种紧耦合的体系结构很容易导致失败，也很难维护和扩展。许多 Web 服务的先驱都从痛苦的经历中（再三）发现了这一事实。

1.6 EAI 简介

为了说明紧耦合依赖关系的影响以及如何加以解决，下面来分析如何用一种简单的方法来连接两个系统。假设我们要构建一个在线银行系统，为客户提供银行账户的跨行转账功能。为了实现这一功能，前端的 Web 应用必须与后端管理资金转账的金融系统集成。

要连接两个系统，最容易的方法是采用 TCP/IP 协议。过去 15 年中开发的各种知名操作系统或程序开发库都肯定包含有 TCP/IP 堆栈。连接到 Internet 和本地网络上的几百万台计算机间进行数据传输时都普遍使用 TCP/IP 作为通信协议。那么为什么不使用最普遍的网络协议来实现应用间的通信呢？

为简单起见，我们假设把钱存入个人账户的远程函数只取人名和金额作为参数。下面的几行代码将基于 TCP/IP 来调用该函数（这里采用的是 C#语言，但是看起来与 C 或 Java 代码基本相同）。

```
String hostName = "finance.bank.com";
int port = 80;

IPHostEntry hostInfo = Dns.GetHostByName(hostName);
IPAddress address = hostInfo.AddressList[0];

IPEndPoint endpoint = new IPEndPoint(address, port);

Socket socket = new Socket(address.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
socket.Connect(endpoint);

byte[] amount = BitConverter.GetBytes(1000);
byte[] name   = Encoding.ASCII.GetBytes("Joe");
```

```
int bytesSent = socket.Send(amount);
bytesSent    += socket.Send(name);

socket.Close();
```

这段代码打开与地址 finace.bank.com 的一个 Socket 连接，然后通过网络发送两项数据（金额和客户的姓名）。这里不需要昂贵的中间件：不需要 EAI 工具、RPC 工具集，只需要 10 行代码而已。运行这段代码时，它会告诉我们“发送了 7 个字节”。不错啊！谁说集成很难？

但是采用这种集成方式带来很多严重的问题。TCP/IP 协议的一个主要优势是它得到了广泛的支持，因此我们可以与其他大量连接到网络上的计算机建立连接，而不必考虑它们采用的操作系统或编程语言是什么。但是，这种平台无关性只适用于非常简单的消息：字节流。为了把数据转换成字节流，我们使用了 BitConverter 类。这个类能把任何数据转换成一个字节数组，会使用数据类型的内部内存表示。问题在于，整数的内部表示会随着计算机系统的不同而有所变化。例如，.NET 使用 32 位整数，而其他系统使用 64 位表示。刚才这个示例通过网络传送 4 字节，表示一个 32 位的整数。如果是一个使用 64 位整数的系统，就会从网络上读取 8 字节的数据，这样将把整个消息（包括客户的姓名）都解释为一个整数。

同样地，有的计算机系统采用 big-endian 格式存储数字，而另外一些系统则采用 little-endian 格式。big-endian 格式先存储最高位，采用 little-endian 格式的系统则按照最低位优先的方式存储数字。PC 机采用的是 little-endian 格式，因此上述代码会在网络上传送以下 4 个字节：

```
232 3 0 0
```

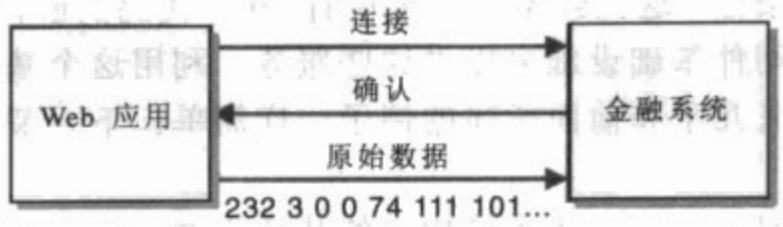
$232 + 3 * 2^8$ 等于 1000。使用 big-endian 数字的系统可能会把这个消息理解为 $232 * 2^{24} + 3 * 2^{16} = 3\ 892\ 510\ 720$。这么一来，Joe 一下子就会成为一个富翁！因此所有相连的计算机必须采用相同的内部格式表示数字，只有在这个假设前提下这种解决方法才可行。

采用这种简单的方法还会带来另一个问题，在此我们指定了远程计算机的地址（在示例中即为 finance.bank.com）。动态名字解析服务（Dynamic Naming Service，DNS）在域名与 IP 地址之间提供一个间接层，但是如果我们想要把函数转移到不同域中的另一台计算机上，或者计算机出现故障，需要启动另外一台计算机，再或者如果我们想把信息发送给多台计算机，该怎么办？上述每种情况都要求我们修改代码。如果使用了大量的远程函数，这项工作将会很繁琐。因此应该找到一种方法，使通信与网络上特定的计算机无关。

前面这个简单的 TCP/IP 示例还在两台计算机之间建立了临时的依赖关系。TCP/IP 是一种面向连接的协议。在传输任何数据前，首先必须建立连接。建立 TCP 连接则需要在发送者和接收者之间来回传递 IP 数据包。这就要求两台计算机和网络必须同时可用。如果三者之中的任何一方出现故障，或者由于负载过大而不可用，数据就无法发送。

最后，这个简单的通信还有赖于非常严格的数据格式。我们要发送 4 个字节的金额数

据，然后是一个定义客户账户的字符序列。如果还要插入第三个参数，比如货币名，就需要修改发送者和接收者的代码，使用新的数据格式。



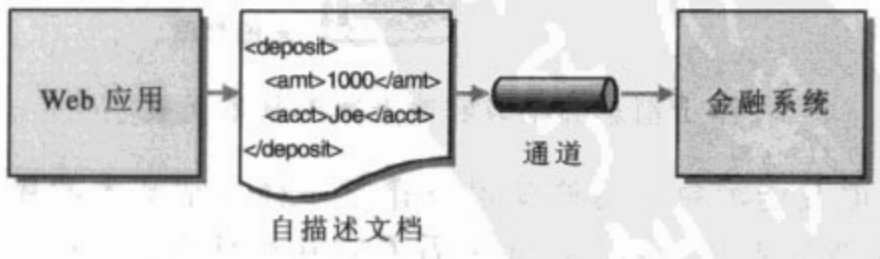
紧耦合的交互

这个最小集成解决方案确实很容易构建，且成本很低，但是它也是一个非常脆弱的解决方案，因为参与通信的双方要做出如下假设：

- 平台技术——数字和对象的内部表示
- 位置——硬编码的机器地址
- 时间——所有组件必须同时可用
- 数据格式——参数表以及参数的类型必须匹配

本章一开始就介绍过，耦合可以作为一个标准来衡量通信双方对彼此做了多少假设。前面的简单解决方案需要参与者做大量的假设。因此，这种解决方案属于紧耦合。

为了使该解决方案放松耦合，必须尽可能消除这些依赖关系。可以使用标准的数据格式，能够自描述，而且与平台无关，如 XML。可以采用把数据发送给一个可寻址的“通道”，而不是把数据直接发送给指定的计算机。通道是一种逻辑地址，发送者和接收者可以在不了解对方身份的情况下协商使用相同的通道。使用通道可以解决位置依赖问题，但是，如果通道采用面向连接的协议来实现，那么在使用时仍然需要所有组件同时可用。为了消除这种暂时的依赖问题，可以改进通道，当网络和接收系统未准备好时，先把发送请求放在队列中等待。现在发送者就可以向通道发送请求并继续处理，而不必担心数据的实际传送。为了在通道的内部实现请求排队，需要把数据封装成自包含的消息，这样一来，在任何时刻通道都可以知道需要缓存和传送多少数据。最后，这两个系统还依赖于一种共同的数据格式，但可以在通道中完成数据格式的转换，从而消除这种依赖。如果某个系统的格式改变了，我们只需要修改转换器，而不必修改参与的其他系统。当有许多应用向相同的通道发送数据时，这种做法尤其有用。



松耦合的交互

利用公共数据格式、通过排队通道异步通信和转换器之类的机制，有助于把紧耦合的解决方案转变成松耦合的解决方案。发送者不再依赖于接收者的内部数据格式，也不再依

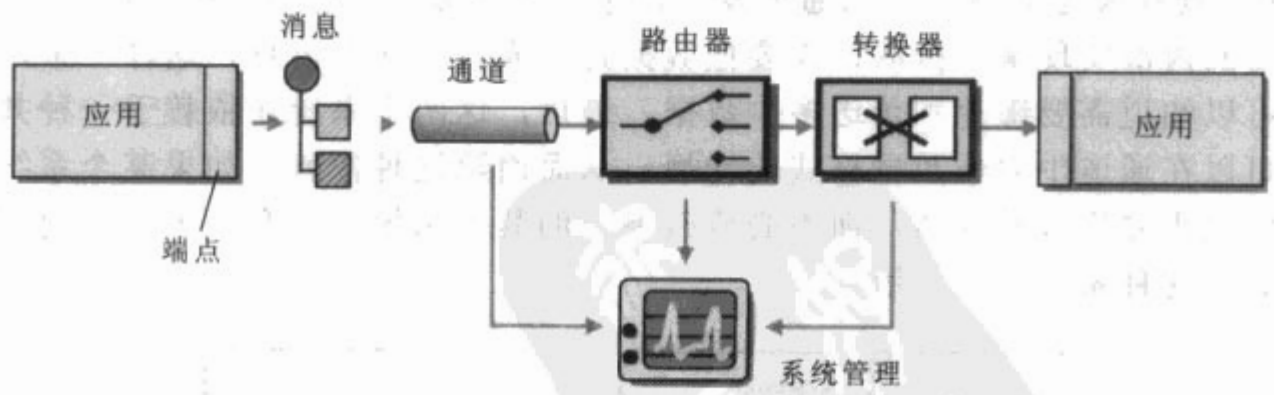
赖接收者的位置，甚至不必关心其他计算机是否已经准备好接收请求。通过消除这些系统间的依赖，可以使整个解决方案更能适应变化，这正是松耦合的主要优点。松耦合方式的主要缺陷是它会增加复杂性。这已经不是一个仅有 10 行代码的解决方案了！因此，我们要使用一个面向消息的中间件基础设施来提供这些服务。利用这个基础设施，可以采用松耦合的方式交换数据，而且几乎和前面所述的例子一样简单。下一节将介绍这种中间件解决方案的组成组件。

松耦合就是万能的吗？与企业体系结构中的其他方面一样，对此并没有一个最佳的答案。松耦合提供了很多重要的优点，如灵活性和可扩展性，但是它也带来了更复杂的编程模型，使得设计、构建和调试解决方案变得更加困难。

1.7 一个松耦合的集成解决方案

要通过集成解决方案连接两个系统，为此需要解决许多问题。这些功能构成了所谓的中间件（middleware）——也就是应用之间的粘合剂。

通常，必须把一些数据从一个应用传递给下一个应用。这些数据可能是需要复制的地址记录，也可能是对远程服务的一个调用，或者是用于门户显示的一段 HTML。无论是哪种类型的数据，都必须能被通信的双方理解，并能通过网络传送。为了实现这一基本功能，需要两个元素：通道和消息。通信通道能把信息从一个应用传送给另一个应用，它可能是一系列 TCP/IP 连接、一个共享文件、一个共享数据库，甚至是在计算机间传递数据的一个软盘（这就是所谓声名狼藉的“sneakernet”，即“偷偷会面”）。在这个通道里，要放置一个消息——这是一段数据，而且对要集成的应用来说，消息的含义是一致的。这段数据可能很小，比如是某个客户修改的电话号码；也可能很大，例如所有客户及其相应地址的完整列表。



基于消息的集成解决方案的基本元素

既然可以通过通道发送消息，这样就可以建立一种非常基本的集成。但是，我们已经说过，“简单集成”显然是矛盾的，所以先来看看它缺少什么。前面说过，集成解决方案对所集成的应用只有有限的控制，比如无法修改应用使用的内部数据格式。举例来说，可能有一种数据格式把客户的名字存储在两个字段中，分别是 FIRST_NAME 和 LAST_NAME，而其他系统可能只使用一个 Customer_Name 字段来存储客户的名字。同样地，也许一个系

统能够支持多个客户地址，而其他系统只支持一个客户地址。由于应用的内部数据格式通常无法改变，因此中间件必须提供某种机制，把一个应用的数据格式转换成另一个应用的数据格式。这个步骤称为转换（translation）。

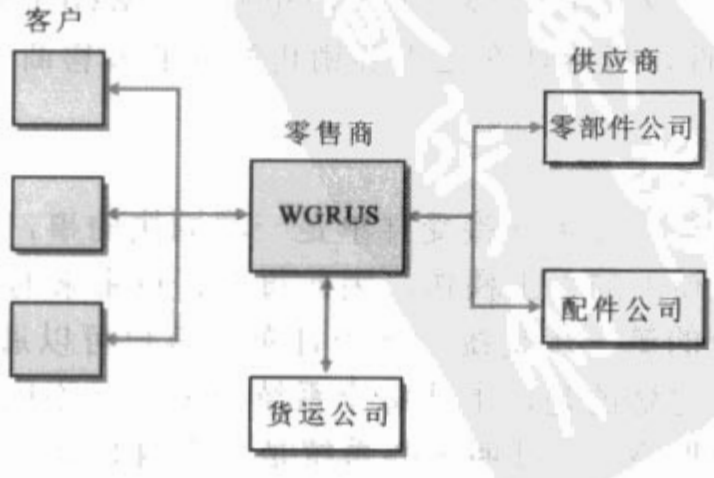
至此，我们可以把数据从一个系统发送到另一个系统，并能应付不同的数据格式。如果要集成更多的系统，会出现什么情况呢？数据要发送到哪里？我们可能希望每个应用都明确地指定数据经通道要发送到哪个目标系统（或者哪些目标系统）。例如，如果客户关怀系统中的客户地址发生了改变，可以让该系统负责将这个数据发送给其他所有存储有客户地址副本的系统。随着系统数量的增加，这会变得非常麻烦，并且要求发送方掌握所有目标系统的信息。每增加一个新系统，客户关怀系统都要相应调整，以适应新环境。如果由中间件负责把消息发送到正确的位置，就会容易得多。这就是路由组件的作用，如消息代理（message broker）。

由于要处理多个应用，另外要处理数据格式、通道、路由及转换等诸多方面，集成解决方案很快就会变得相当复杂。所有这些元素可能位于多个操作平台，或者处在不同的地理位置。为了掌握系统内部的运行机制，必须提供一个系统管理功能。这个子系统要监控数据流，确保所有应用和组件都在正常运行，并能向中心节点报告出错情况。

现在，我们的集成解决方案已经基本完成。可以把数据从一个系统转移到另一个系统，并支持不同的数据格式，可以把数据路由传送到相应的目标系统，并能监控解决方案的性能。到目前为止，有一个前提条件，这就是应用要把数据作为消息发送给通道。但是，大多数封装好的应用和遗留系统都没有打算要参与集成，而且许多定制应用也没有考虑过集成问题。为此，我们需要采用消息端点（message endpoint）把这些系统显式地连接到集成解决方案中。端点可能是一段特殊的代码，也可能是一个由集成软件开发商提供的通道适配器（Channel Adapter）。

1.8 Widgets & Gadgets 'R Us 公司：示例

要理解基于消息的集成解决方案，最好的办法就是分析一个具体的例子。下面以 Widgets & Gadgets' R Us（简称 WGRUS）公司为例，这是一家在线零售商，从制造商那里购买零部件及配件，然后销售给客户。



WGRUS Ecosystem

在这个例子中，假设解决方案需要支持以下需求。当然，为了节省篇幅，我们对这些需求做了简化，不过在实际业务中经常会出现这样一些需求。

- **接受订单**：客户可以通过网络、电话或传真下订单。
- **处理订单**：处理订单涉及多步操作，包括检查库存情况、发货以及为客户开具发货单。
- **检查状态**：客户可以检查订单的状态。
- **修改地址**：客户可以通过 Web 前端修改他们的账单和货运地址。
- **更新目录**：供应商要定期地更新商品目录。WGRUS 必须根据新的商品目录来更新商品的定价和存货情况。
- **发布公告**：客户可以有选择地订购 WGRUS 发布的公告。
- **测试与监控**：操作人员应当能够监控所有组件以及组件之间的消息流。

下面将分别处理上述各个需求，并使用本书介绍的模式语言来描述各种替代解决方案和折衷考虑。首先从简单的消息流体系结构开始，然后逐步解决更复杂的需求，在此过程中，将会引入更多复杂的概念，如过程管理器（Process Manager）。

1.8.1 内部系统

与大多数集成应用环境一样，WGRUS 并不是所谓的“green field”实现，而是基于一个已有的 IT 基础设施实现的集成，其中包括大量封装好的应用和定制应用。由于必须与已有的应用打交道，这往往使集成工作很负有挑战性。在这个例子中，WGRUS 运行着以下系统（如右图所示）。



WGRUS IT 基础设施

WGRUS 采用 4 个不同的通道与客户交互。客户可以访问公司的 Web 网站，打电话给呼叫中心的客户服务代表，或者通过传真提交订单。客户还可以通过电子邮件接收通知。

WGRUS 的内部系统由财会系统和货运系统组成，财会系统还包括账单管理功能，货运系统能够计算运费并与货运公司协调。出于历史原因，WGRUS 有两个库存和目录系统。WGRUS 过去只销售零部件，但是现在还需要销售配件的零售商。

1.8.2 接受订单

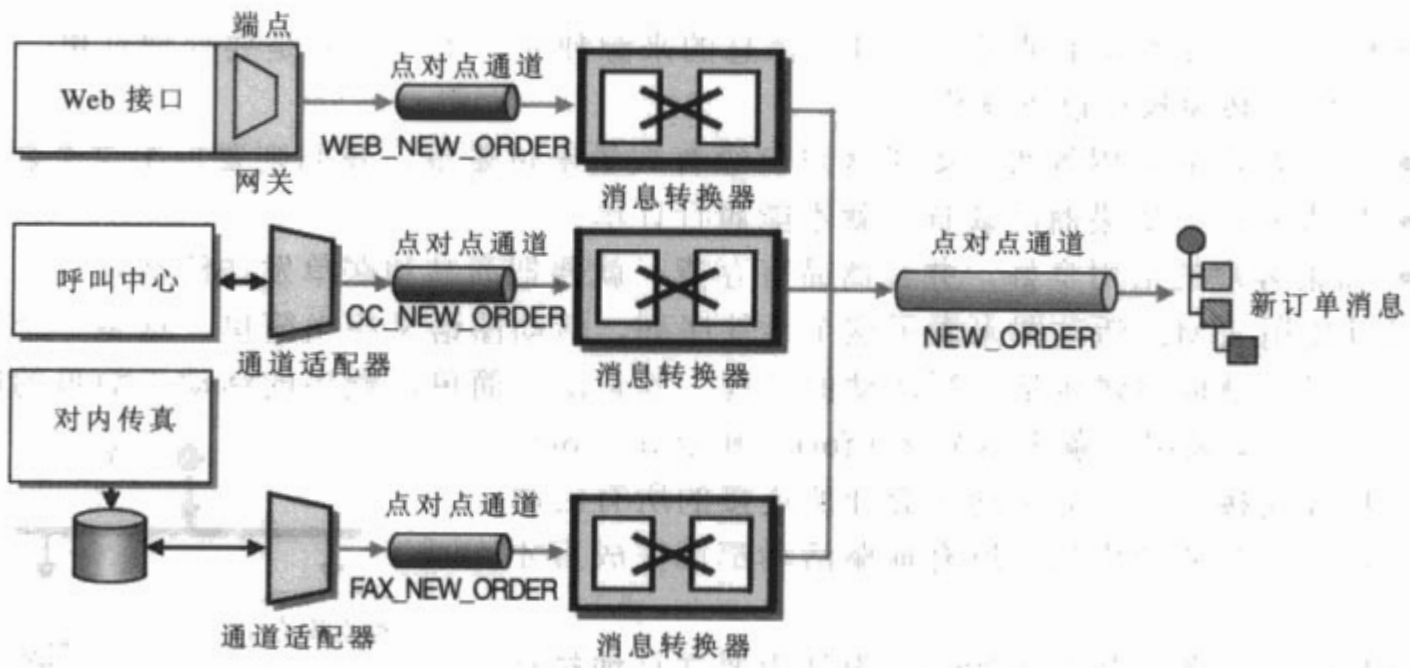
首先要实现的功能是接受订单。接受订单是一项愉快的事，因为订单能带来收入。但是，下订单目前还是一个烦人的手工操作，因此每个订单带来的开销是很高的。

实现流水线订单处理的第一步是统一接受订单。客户可以通过三种通道下订单：Web 站点、呼叫中心或传真。遗憾的是，上述每个系统都采用了不同的实现技术，并且把接受到的订单保存为不同的数据格式。呼叫中心系统是一个封装好的应用，而 Web 站点是一个定制 J2EE 应用。对内传真系统需要把数据项手工录入到一个小型的 Microsoft Access 应用

中。我们必须平等对待所有订单，而不考虑它们的来源。例如，客户可以通过呼叫中心下订单，然后通过 Web 站点检查订单的状态。

由于下订单是一个异步的过程，连接了多个系统，所以我们决定实现一个面向消息的中间件解决方案来完成订单输入过程的流水线处理。封装好的呼叫中心应用在开发时没有考虑集成问题，因此需要使用通道适配器把它连接到消息系统。通道适配器是一种能与应用关联的组件，只要应用内部有事件发生，它就能将消息发布到一个消息通道。对于某些通道适配器，应用甚至不会察觉到它们的存在。例如，数据库适配器可以为某些表添加触发器，每次应用插入一行数据时，就会有一个消息发送到消息通道。通道适配器还反方向工作，从消息通道中取出消息，并在应用内部触发一个动作作为响应。

对内传真应用中使用了相同的方法，将通道适配器连接到应用数据库。由于 Web 应用是定制构建的，所以我们在 Web 应用内部实现了消息端点代码，并使用消息传递网关（Messaging Gateway）把应用代码与特定于消息传递的代码相隔离。



从 3 种不同的通道接受订单

由于每个系统对接收到的订单使用了不同的数据格式，因此要使用三个消息转换器将不同的数据格式转换为一种公共的新订单消息，这种消息要遵循一个规范化数据模型（Canonical Data Model）。规范化数据模型定义了独立于特定应用的消息格式，因此所有应用都能采用这种公共的格式相互通信。如果一个应用的内部格式有变化，只是这个应用与公共消息通道之间的消息转换器需要修改，所有其他应用和消息转换器都不会受到影响。使用规范化数据模型意味着要处理两种类型的消息：规范（公共）的消息和特定于应用（私有）的消息。特定于某应用的消息只能由该应用和相关消息转换器处理，任何其他组件均不能处理。为了突出这一点，我们在特定于应用的消息通道的名称前加上了应用名，例如，WEB_NEW_ORDER 通道（表示特定于 Web 应用）。与之对应，传送规范消息的通道则根据消息的作用来取名，不加任何前缀，例如 NEW_ORDER。

之所以通过一个点对点通道将各个通道适配器与消息转换器相连接，是为了保证每个订单消息只被处理一次。如果把转换功能放到消息传递网关中实现，可以不对 Web 接口使用消息转换器。但是，将转换功能硬编码写入程序，不但繁琐还容易出错，因此最好采用一致的方法。附加的消息转换器还可以防止 Web 接口数据格式的微小变化对新订单消息流的影响。所有的消息转换器都把消息发布到同一个的 NEW_ORDER 点对点通道，这样不必考虑订单的来源，都能从这个通道统一地处理订单。

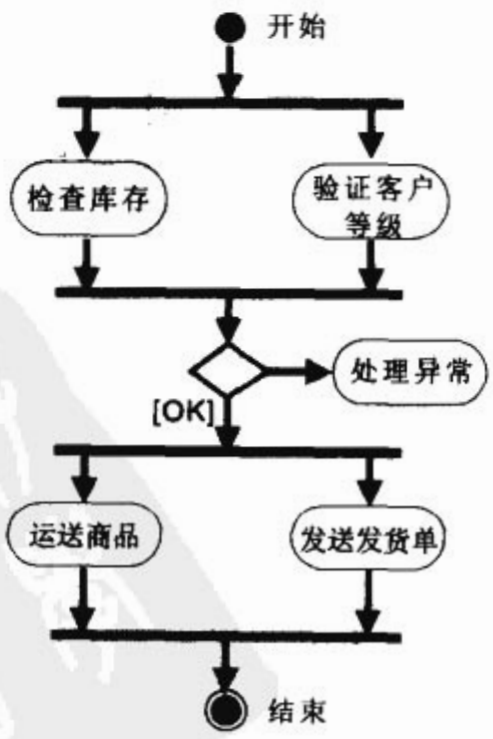
NEW_ORDER 消息通道也被称为数据类型通道（Datatype Channel），因为它只传送一种类型的消息：新订单。这就使消息的消费者很容易知道会得到何种类型的消息。新订单消息本身被设计为一个文档消息（Document Message）。这个消息的目的不是为了指示接收者完成某个特定的动作，而是将一个文档传送给所有感兴趣的接收者，至于如何处理这个文档，则由接收者自行决定。

1.8.3 处理订单

既然有了一致的订单消息，并且与消息的来源独立，下一步就是要处理订单。为了履行一个订单，必须执行以下操作：

- 检查客户的信用等级。如果客户已经有大量未付账单，我们则要拒绝这个新订单。
- 检查库存。如果商品缺货，就不能履行订单。
- 如果客户的信用良好，并且商品有存货，就要把商品和账单发送给客户。

可以使用 UML 活动图来表示这个事件序列。活动图语义相当简单，这是一个很好的工具，可以清楚地描述包括并行活动的过程。其记法很简单，顺序的活动由简单的箭头连接，并行的活动采用一条表示分叉（fork）和合并（join）动作的粗线连接。一个分叉动作会让所连接的所有活动同时开始，而合并动作只在所有前驱活动都已完成后才能继续。



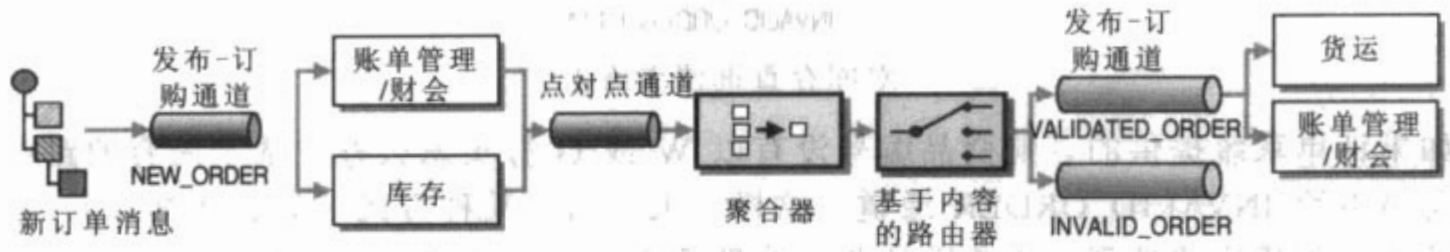
订单处理的活动图

我们的活动图（如右图所示）设计为要并行地执行检查库存任务和验证客户信用等级任务。合并线意味着等待，即前面两项活动都完成后才允许下一个活动开始。下一项活动是检查前两项活动的结果：是否有库存，以及客户的信用是否良好？如果这两个条件都满足，就可以继续履行订单。否则，转向异常处理活动。例如，可以打电话提醒客户支付最近的账单，或者发送一封邮件，告诉他（或她）订单会被延期处理。因为本书的重点是面向消息的集成的设计方面，而不是工作流建模，因此异常处理过程的细节就不再介绍了。工作流体系结构和工作流建模的详细讨论可参阅[Leyman]和[Sharp]。

看起来活动图中的活动与 WGRUS IT 部门中的系统能很好地对应。财会系统检查客户的信用等级，库存系统检查库存，货运系统启动实际的

商品运送。财会系统还承担了账单管理系统的功能，并要发送发货单。由此可见，订单处理正是分布式业务过程的一个典型实现。

为了把逻辑活动图转换为一个集成设计方案，可以使用发布-订购通道（Publish-Subscribe Channel）实现分叉动作，使用聚合器（Aggregator）实现合并动作。发布-订购通道把消息发送给所有活动的消费者；聚合器接收多个输入的消息，并把它们合并成一个发出的消息（如下图所示）。



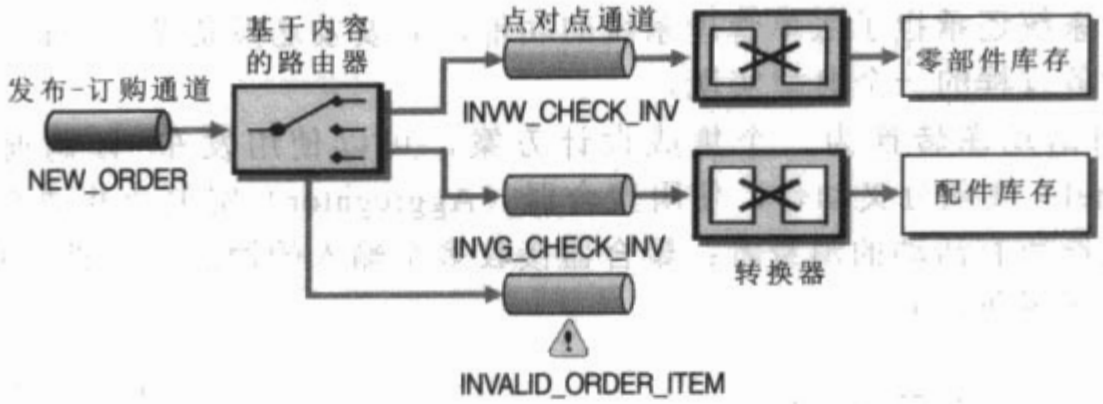
使用异步消息传递实现订单处理

在这个例子中，发布-订购通道把新订单消息同时发送给财会系统和库存系统。聚合器把两个系统处理的结果消息合并，然后将合并后的消息传递给基于内容的路由器（Content-Based Router）。基于内容的路由器处理（消费）一个消息，并且不加修改地把它发布给其他所选的通道，这里要依据编写在路由器中的规则来选择通道。基于内容的路由器等同于 UML 活动图中的分支（branch）。在这个例子中，如果库存检查和信用检查都已通过，那么基于内容的路由器就会把消息发送给 VALIDATED_ORDER 通道。该通道是一个发布-订购通道，因此合法的订单会同时传递给货运系统和账单管理系统。如果客户的信用不佳，或者当前商品缺货，基于内容的路由器就会把消息发送给 INVALID_ORDER 通道。一个异常处理进程（图中没有显示）将监听此通道上的消息，并通知客户订单被拒绝。

在建立了整个消息流后，下面具体来看看库存系统的功能是如何实现的。在需求部分已知 WGRUS 有两个库存系统：一个是用于零部件库存的管理，另一个用于配件库存的管理。因此，必须把库存查询请求发送给正确的系统。为了对其他系统隐藏库存系统的特性，我们插入了另一个基于内容的路由器，它根据所订购商品的类型把消息发送给正确的库存系统（如下图所示）。例如，只要收到的消息中商品编号以 W 打头，就把该消息发送给零部件库存系统处理，凡是商品编号以 G 打头的消息，则统统发送给配件库存系统处理。

请注意位于基于内容的路由器和库存系统之间的点对点通道，其中消息的作用与前面的通道有所不同。这些通道包含的是命令消息（Command Message），即指示系统执行指定命令的消息，在这个例子中就是检查某个商品项的库存情况。

因为零部件库存系统和配件库存系统使用了不同的内部数据格式，所以还要插入消息转换器（Message Translator），把规范的新订单消息格式转换成相应系统的特定格式。

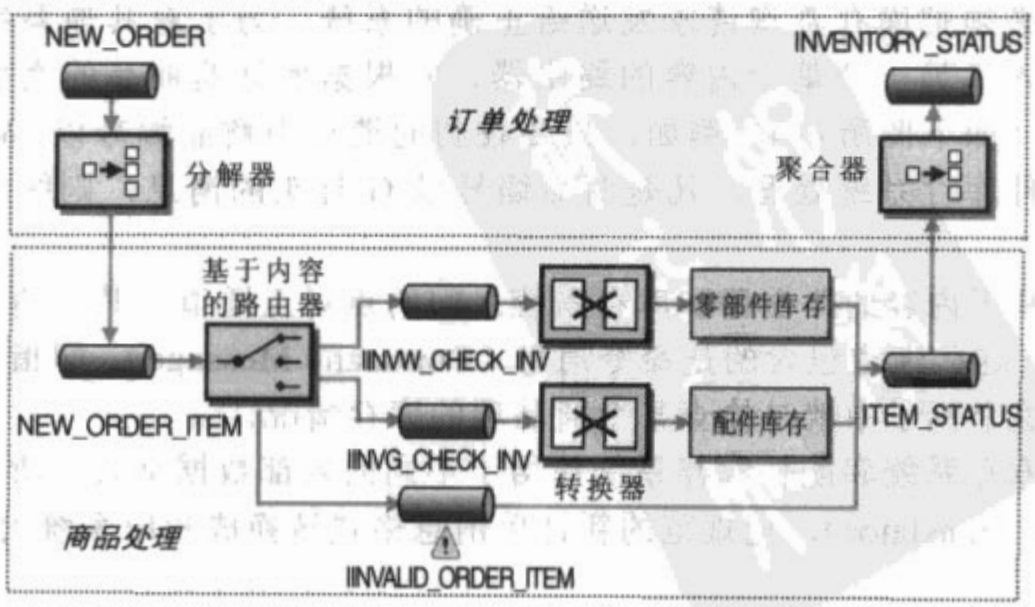


对库存查询请求路由

如果订单系统提供的订购商品编号没有以 W 或 G 打头怎么办？基于内容的路由器会把消息路由给 INVALID_ORDER 通道，这样一来，通过某种方式（如通知客户），非法的订单就会得到相应的处理。该通道是非法消息通道（Invalid Message Channel）的一个典型实例。它强调了一个事实，即取决于消息所在的通道，消息的含义会有所变化。NEW_ORDER 通道和 INVALID_ORDER 通道都传递相同类型的消息，但在其中一个通道上会处理一个新订单，而在另一个通道上该订单则被认为是非法的。

到目前为止，都只是假设每个订单只包含一件商品。对客户来说，这样做很不方便，因为他们必须为每件商品下一个新的订单。而且，对同一个客户的多个订单，我们要分别发货，这就增加了不必要的货运成本。但是，如果允许接受具有多项商品的订单，那么应该由哪个库存系统来检查订单的库存情况呢？可以使用一个发布-订购通道把订单发送给每个库存系统，让它们挑出能够处理的商品。但是对非法订单应该怎么处理呢？怎样才能发现两个库存系统都不能处理的订购商品？我们想保持基于内容的路由器所提供的集中控制功能，但是还需要一个分别路由各个订购商品的新功能。

因此，需要插入一个分解器（Splitter），这个组件能把一个消息划分为多个单个的消息。在此例中，分解器把一个订单消息划分为多个订购商品（Order Item）消息。每个订购商品消息可以被前面的基于内容的路由器路由到正确的库存系统，如下图所示。



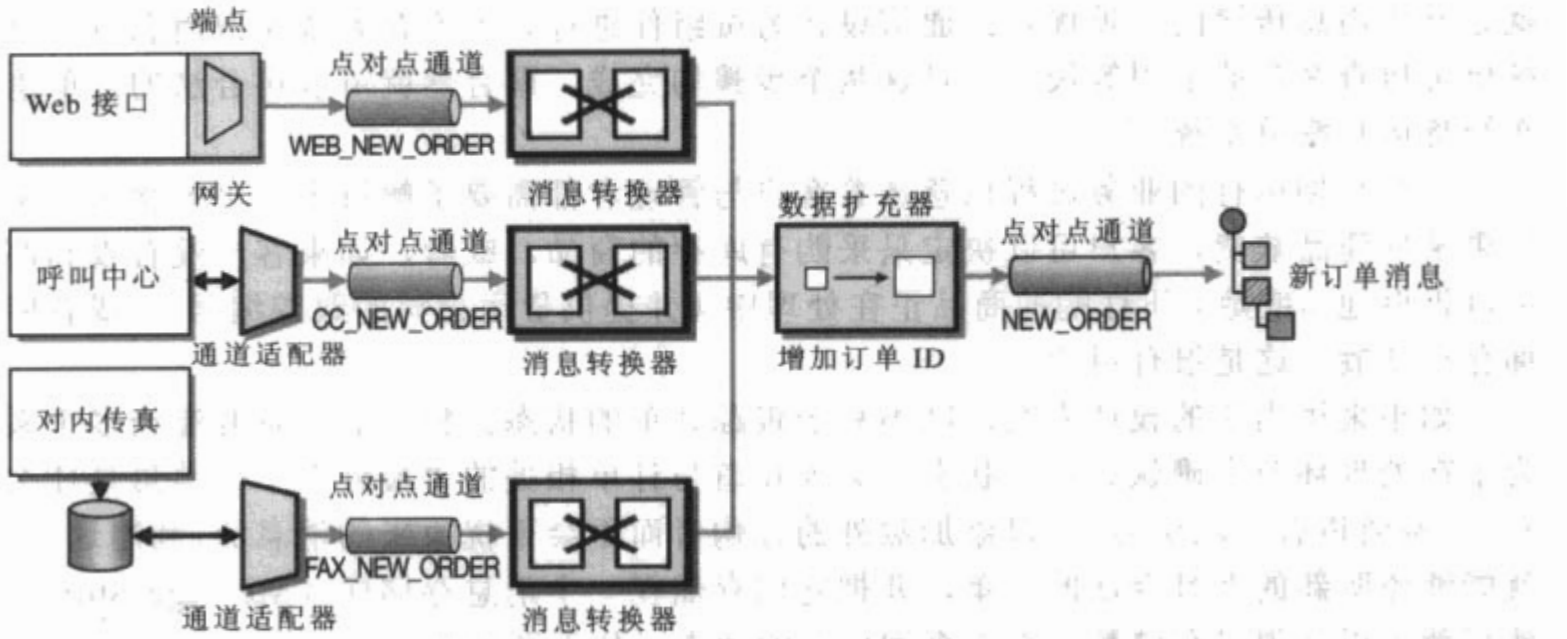
单独处理每个订购商品

很自然，当所有商品的库存情况都已查明后，就要把这些消息重新合并成一个消息。前面已经介绍过，能把多个消息组合成一个消息的组件是聚合器。同时使用分解器和聚合器，可以从逻辑上把完整的订单消息流划分为单个的订购商品消息流。

当设计一个聚合器时，必须做出三个关键决定：

- 哪些消息可以合并（相关性）？
- 如何确定所有消息都已接收（完备性条件）？
- 如何把单个的消息合并成一个结果消息（聚合算法）？

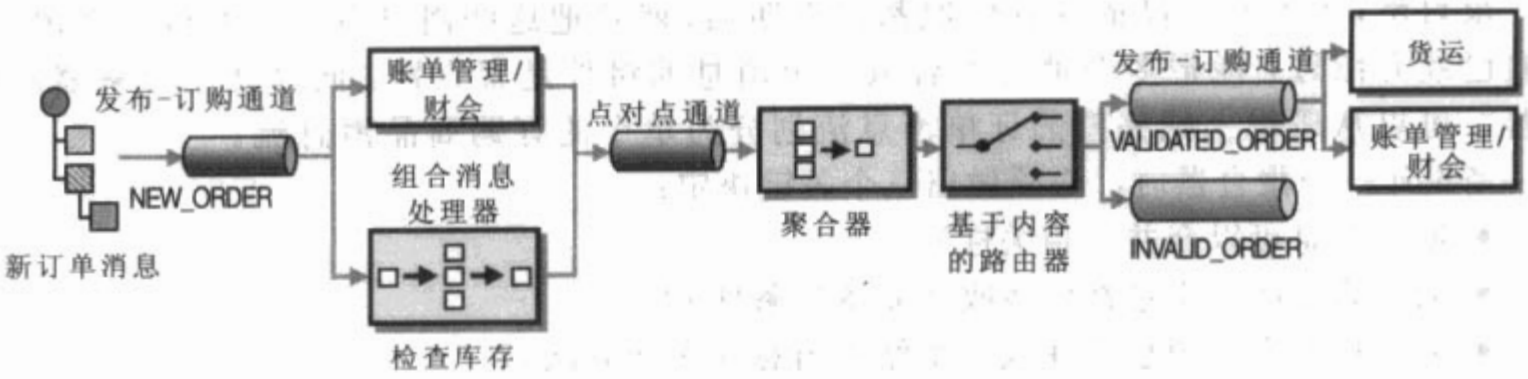
下面让我们逐个地解决这些问题。首先，不能根据客户 ID 来合并订购商品（建立订购商品的相关性），因为客户可能会短时间内连续下多个订单。因此，每个订单必须有唯一的订单 ID。在接受订单的解决方案中插入一个内容扩充器（Content Enricher）就能实现这个功能（如下图所示）。内容扩充器组件可以把缺少的数据项添加到输入的消息中。在这个例子中，内容扩充器把一个唯一的订单 ID 添加到了消息中。



使用扩充器接受订单

既然已经有一个订单 ID 来建立订购商品消息的相关性，下面需要为订购商品聚合器定义完备性条件和聚合算法。由于要把所有消息（包括非法的订购商品）路由到聚合器，聚合器可以简单地使用订单中商品的个数（订单消息中的一个字段）来统计是否所有订购商品都已到达。聚合算法同样很简单。聚合器把所有的订购商品消息重新连接成一个订单消息，并把它发布给 VALIDATED_ORDER 通道。

把分解器、消息路由器和聚合器结合起来是很常见的一种用法。我们把它称为组合消息处理器（Composed Message Processor）。为了使绘图简化，我们在原来的消息流图中插入了一个“组合消息处理器”符号。



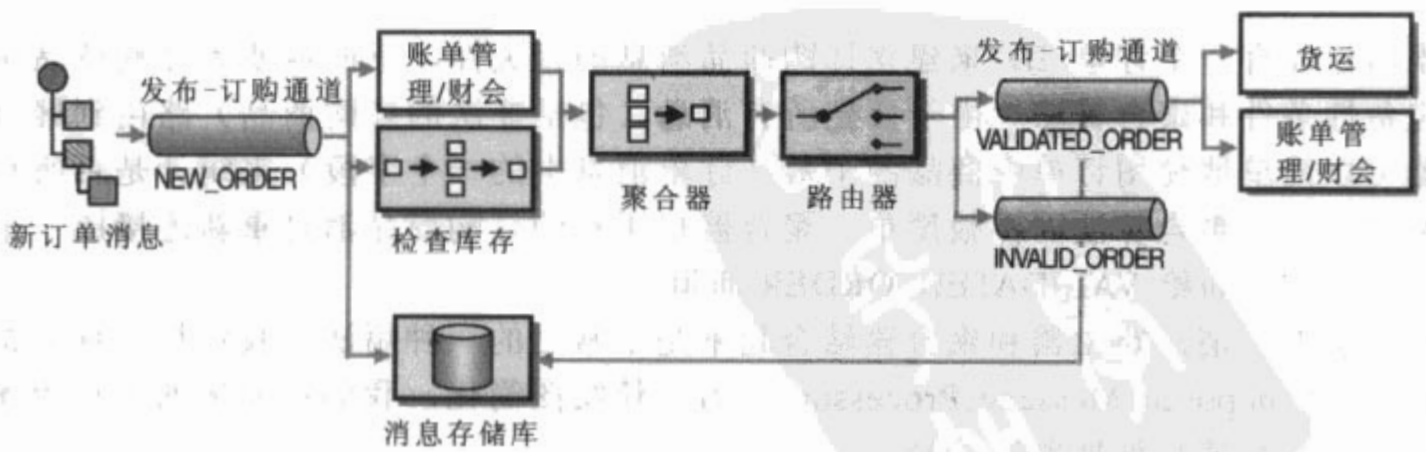
修改后的订单处理实现

1.8.4 检查状态

除了通过消息通道连接系统需要花费时间外，履行订单也要花费一定的时间。例如，所需的某项商品可能缺货，库存系统可以保留库存检查消息，直到新的订购商品到来。这就是异步消息传递的一项优点：通信设计为按组件进行。当库存系统保留消息时，财会系统仍可检查客户的信用等级。一旦这两个步骤均完成，聚合器就可以将合法的订单消息发布给货运和账单系统。

一个长期运行的业务过程也意味着客户与管理者都需要了解特定订单的状态。例如，如果某项商品缺货，客户可以决定只采购有库存的商品。或者，如果客户没有收到商品，可以告诉他（或她）所订购的商品正在处理中（并提供货运公司的跟踪编号），或者库房内部存在耽搁，这是很有用的。

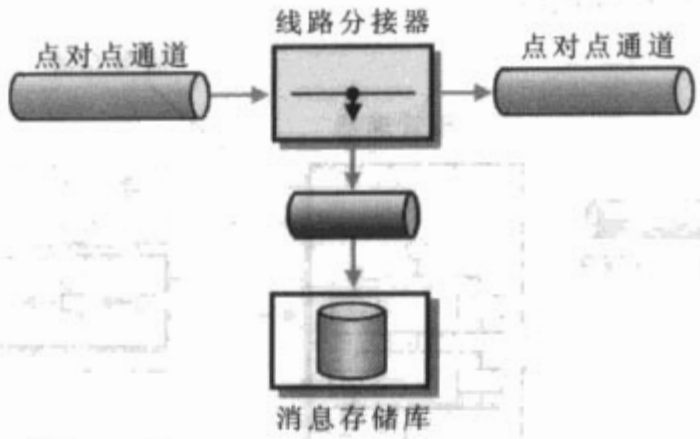
如果采用当前的设计方案，则不利于跟踪订单的状态。相关的消息要流经多个系统。为了在处理环节中确认订单的状态，必须知道与订单相关的“最近”一个消息是什么。发布-订购通道有一项优点，可以添加额外的订购者而不会干扰原来的消息流。利用这一特性，就能够监听新的而且合法的订单，并把它们存储在一个消息存储库（Message Store）中。然后就可以从消息存储数据库中查询订单的状态（如下图所示）。



增加消息存储库来跟踪订单的状态

在使用点对点通道的情况下，不能简便地向通道增加订购者，因为点对点通道要保证每个消息只能由一个订购者消费。但是，我们可以插入一个线路分接器（Wire Tap），这是

一个简单的组件，能从一个通道中取出一个消息并把它发布给两个通道。然后就可以使用第二个通道把消息发送给消息存储库，如下图所示。



使用线路分接器跟踪消息

把消息存储到一个中心数据库中还有另一个显著的优点。在原来的设计中，每个消息必须携带额外的数据才能继续沿线处理。例如，验证客户信用等级（Validate Customer Standing）消息不得不携带所有的客户数据，尽管只会用到客户 ID。这些额外的数据是必要的，这样结果消息中才能仍然包含有原始订单消息中的所有数据。在消息流的起始处把新订单消息保存在一个消息存储库中，后续的组件通过查询消息存储库就可以获得重要的消息数据，这样中间环节就不用携带这些数据了。我们把这一功能称为声明标签（Claim Check）——消息可以“登记”数据，以便事后查询。这种方法的缺点是访问中心数据库不像采用异步消息通道发送消息那么可靠。

现在，消息存储库负责维护与新消息相关的数据，以及处理过程中改变的消息。利用消息存储库提供的充分信息，可以确定下一步要执行的处理动作，而不必把组件连接到固定的消息通道上。例如，如果数据库中包含有来自库存系统和账单管理系统的应答消息，就可以做出订单已经得到验证的结论，并且可以向货运系统和账单管理系统发送一个消息。采用消息存储库，可以代替一个单独的聚合组件做出正确的决定。为了提高效率，可将消息存储库转换为一个过程管理器（Process Manager）。

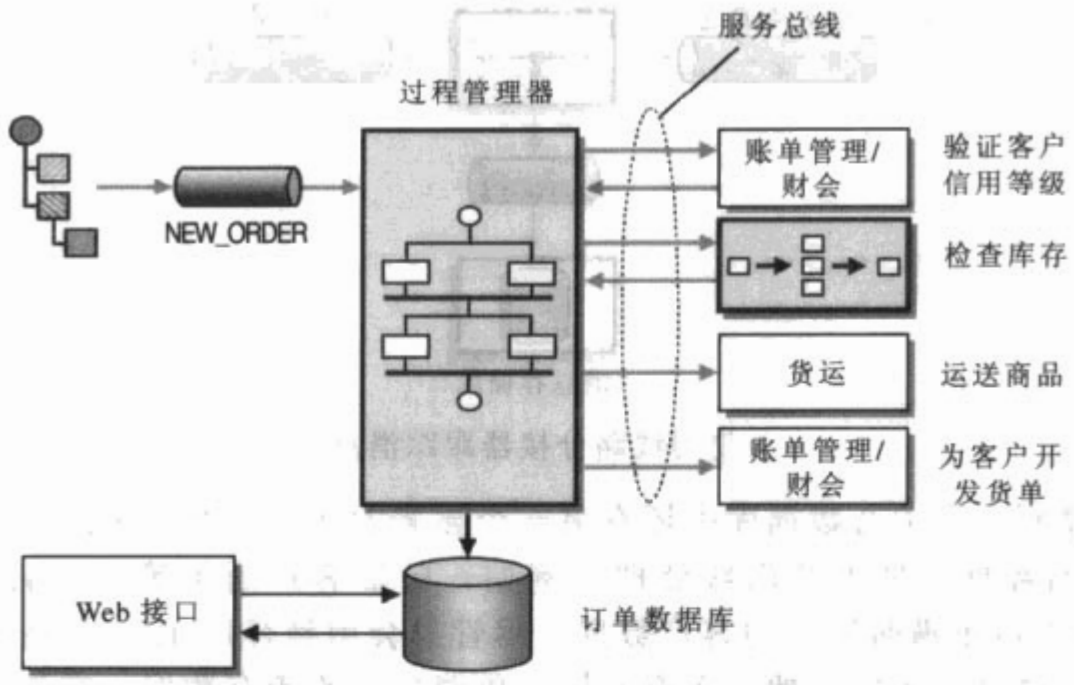
过程管理器是一个系统中的中心组件，用以管理消息流。它提供两个主要功能：

- 保存消息之间（一个“过程实例”中）的数据
- 跟踪进度，并确定下一步的动作（通过使用一个“过程模板”）

该体系结构把单个的系统（如库存系统）转变成共享的业务功能，从而能被其他组件访问，这就增加了重用性，并简化了系统维护。这些服务可以通过消息流连接起来（例如，使用组合消息处理器检查每个订购商品的库存状态），或者通过过程管理器协调。需要改变消息流时，使用过程管理器要比以前的方法更简单。

新的体系结构把所有的服务连接到一个公共的服务总线上，从而能被其他任何组件调用。通过增加发现功能，即从中心服务注册库中查找（或“发现”）服务，可以把 WGRUS IT 基础设施转换成一个 SOA。为了加入这个 SOA，每个服务必须提供额外的功能。例如，每个服务必须提供一个接口合约，用来描述这个服务的功能。每个请求/应答服务还需要支

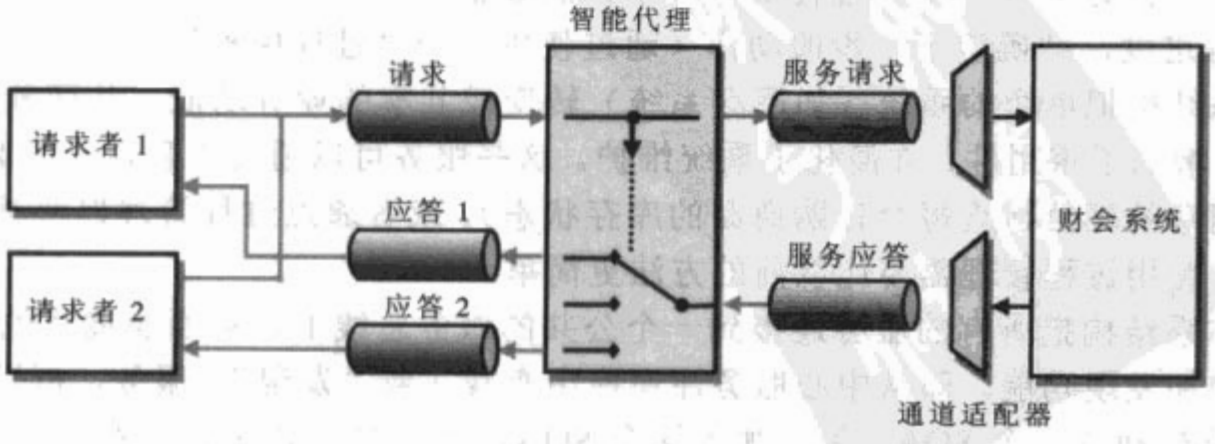
持返回地址（Return Address）的概念。基于返回地址，调用者（服务消费者）可以指定一个通道，服务要在这里发送应答消息。这对于服务在不同环境中重用也很重要，每种环境可能需要不同的通道来发送应答消息。



使用过程管理器处理订单

过程管理器本身使用一个持久存储库（采用文件或关系数据库）来保存与每个过程实例相关的数据。为了让 Web 接口查询订单的状态，可以向过程管理器或订单数据库发送一个消息。但是，检查状态是一个同步过程——客户希望能立即获得响应。由于 Web 接口是一个定制的应用，所以我们采用直接访问订单数据库的方式查询订单的状态。这种共享数据库是最简单、效率最高的方法，并能保证 Web 接口始终显示最新的状态。该方法的潜在不足是 Web 接口与数据库属于紧耦合关系，但这还是可以接受的。

要把系统提供为服务存在一个难点，那就是许多遗留系统原来并没有考虑返回地址这样的特性。因此，必须采用智能代理（Smart Proxy）“包装”对遗留系统的访问。智能代理利用附加功能来改进基本系统服务，使之能加入到 SOA 中。为了达到这个目标，智能代理会拦截发送到基本服务的所有请求和来自于基本服务的应答消息（如下图所示）。



插入智能代理，以把遗留系统转换成一个共享服务

智能代理可以存储来自请求消息中的信息（例如由请求者指定的返回地址），并使用这一信息处理应答消息（例如把消息路由到正确的应答通道）。智能代理对于跟踪外部服务的服务质量（如响应时间）也很有用。

1.8.5 修改地址

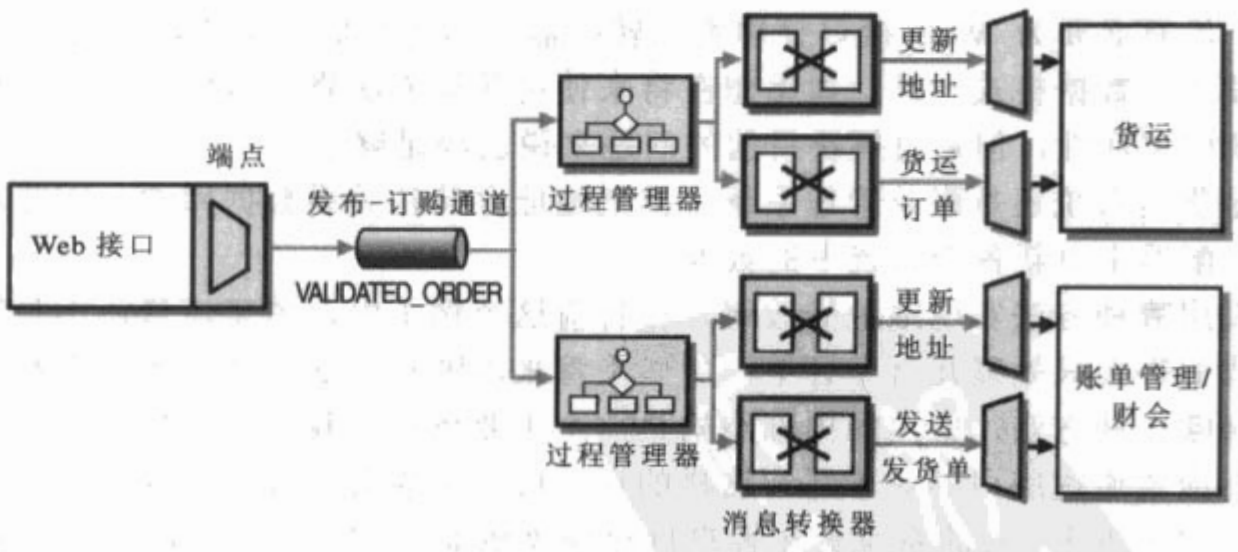
WGRUS 需要处理大量地址。例如，必须把发货单按照付账地址发送给客户，并把商品运送到发货地址。我们希望客户能通过 Web 接口维护这些地址，以减少不必要的手工步骤。

可以采用两种基本的方法把正确的付账地址和发货地址交给账单管理系统和货运系统：

- 在每条新订单消息中包含地址数据
- 在每个系统中保存地址数据，并且当地址改变时进行复制

第一种方法的优点是可以使用已有的集成通道传递额外的信息。其潜在不足是这些额外的数据要流经中间件基础设施，即使地址很少被修改，也要携带在每个订单中进行传送。

实现第一种方法时，必须考虑到账单管理系统和货运系统属于已经封装好的应用，它们设计时很可能没有考虑到集成。因此，它们很可能无法接受新订单中的地址，而是要使用存储在本地数据库中的地址。为了使系统能按照新订单消息更新地址，必须在账单管理系统（和货运系统）中实现两个功能：首先要更新地址，然后发送账单（或运送商品）。由于这两个消息的顺序非常关键，我们插入了一个简单的过程管理器组件，用它接收新订单消息（包含有当前的发货地址和付账地址），并向账单管理（或货运）系统发布两个单独的消息（如下图所示）。



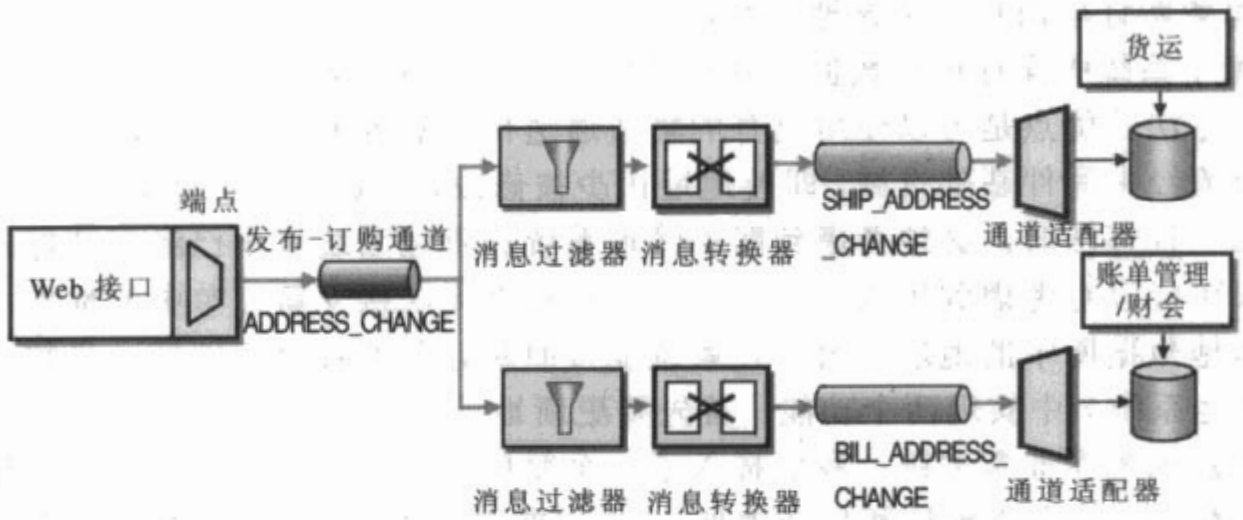
在新订单消息中包含地址数据

需要牢记一点，通道适配器要求消息必须格式化为应用所需的格式（使用所谓的私有消息）。因为新订单消息采用了规范消息格式，所以必须在两种格式之间完成转换（即规范格式与私有格式的转换）。可以把转换功能放在过程管理器中实现，不过采用外部消息转换器更好一些，这样的话，过程管理器中的逻辑就不会受到应用所需复杂数据格式的影响。

第二种方法是使用数据复制把修改后的地址传播给所有受影响的系统，这与新订单处

理无关。不管在什么时候通过 Web 接口修改了地址信息，都要使用一个发布-订购通道把修改后的地址发送给所有相关的系统。系统在内部存储更新后的地址，并在订单消息到达时使用这个地址。这种方法能减少消息流量(假设客户修改地址的频率比下订单的频率低)。它还能减少系统之间的耦合。任何用到地址的系统都可以订购 ADDRESS_CHANGE 通道，而不会影响其他系统。

由于我们处理的地址有多种类型(发货地址和付账地址)，因此必须确保每个系统中保存的地址都是正确的。要避免把付账地址的修改消息发送给货运系统。为此可以使用消息过滤器（Message Filter）对消息进行过滤，只让符合指定标准的消息通过（如下图所示）。



通过一个单独的发布-订购通道传播地址修改消息

还可以使用消息转换器把普通的地址修改消息转换成应用所用的特殊消息格式。在这个例子中，没有必要为 Web 接口使用消息转换器，因为我们定义的规范数据模型和 Web 接口应用使用的数据格式一样。如果想在将来使用其他方法修改地址，目前这种做法可能会限制系统的灵活性，但是对解决目前的问题来说已经足够了。

不管是货运系统还是账单管理系统，都把地址存储在关系数据库中，因此我们使用数据库通道适配器来更新各个系统中的数据。

应该采用哪种方法实现地址修改呢？在目前这个例子中，消息流量还不是很突出的问题，因为我们每天只处理几百个订单，所以不管采用哪种方法，从效率上看都是合理的。要决定选择哪一种方法，应用的内部结构是一个主要依据。我们不能把地址直接插入到数据库中，而应该通过应用的业务层。这样的话，应用可能会完成额外的验证动作，并把地址修改活动记录下来。有的系统会在客户每次修改地址后，向客户发送一个消息确认邮件。如果在每个订单中都携带地址更新信息，就会给客户带来很大的麻烦。为此最好采用专门的消息来传播地址修改信息，而且这些消息只在客户确实修改地址时才发送。

我们一般都喜欢明确定义的自包含业务动作，如“修改地址”和“下订单”，因为这对实现业务过程的协调提供了更多的灵活性。现在的问题是，如何确定这些业务动作的粒度及其相关的代价。细粒度的接口需要执行过多的远程调用或发送大量的消息，因而会造成系统运行缓慢。例如，设想一个接口为修改各个地址字段分别提供了单独的方法。如果通信发生在同一个应用中，采用这种方法效率很高，只需更新那些有所改变的字段就可以。

而在一个集成环境下，为了更新一个地址，如果需要发送六到七个消息，这就存在一个很大的开销，不仅如此，我们还要处理消息之间的同步。细粒度的接口还会导致紧耦合。如果修改了地址的格式，在其中添加了一个新的字段，就必须定义新的消息格式，并且要修改其他所有应用，使它们能发送这样一种新消息。

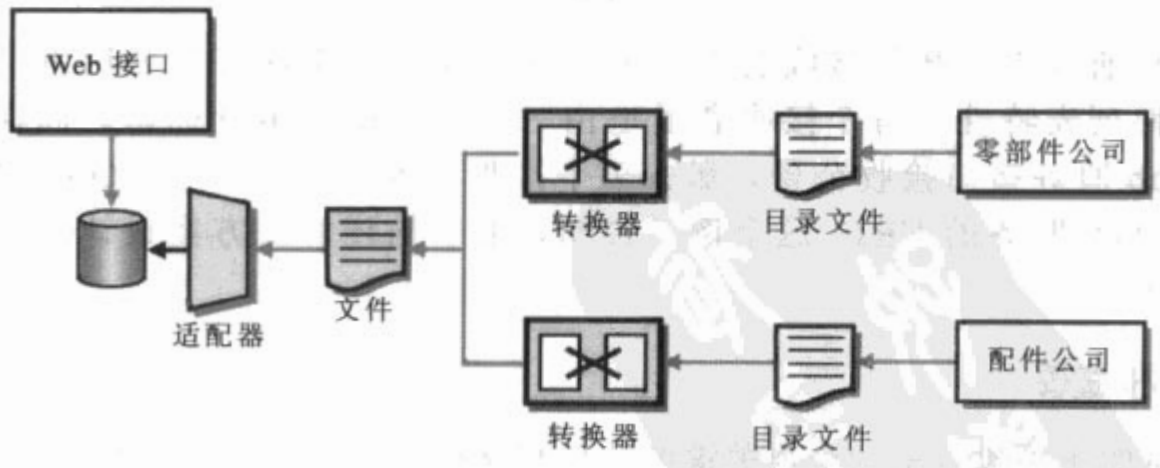
粗粒度的接口克服了上述问题，但是也要付出代价。发送的消息越少，系统的效率就会越高，而且耦合度也越低。但是，粒度太粗的接口会限制系统的灵活性。如果把发送发货单与修改地址的功能结合到一个外部函数中，那么每次修改地址时都要发送一次账单。因此，最好的做法是根据现实中具体的工作来权衡决定接口的粒度。

1.8.6 更新目录

为了下订单，客户首先要查看当前能提供的商品及其价格。WGRUS 的商品目录是由不同的供应商提供的。但是，WGRUS 会向客户提供一项服务，使他们能够在同一网站上浏览所有零部件和配件，并能在同一个订单中订购这两种类型的商品。这项功能是信息门户的一个例子，要把多个信息源的信息结合到一个视图中。

由于这两种类型的供应商都是每隔三个月才更新一次商品目录。因此，在 WGRUS 中，如果采用实时消息传递基础设施来广播供应商对商品目录所做的修改，这没有太大的意义。相反，我们使用文件传输（File Transfer）集成把商品目录数据从供应商传送到 WGRUS。使用文件还有一个优点，可以使用 FTP 或类似的协议，通过公共网络方便、高效地传输文件。与之相比，大多数异步消息传递基础设施在公共 Internet 上的表现则不佳。

同样地，还可以使用转换器和适配器把数据转换成内部目录格式。但是，这些转换器一次要处理整个商品目录，而不是一个商品。当我们处理大批量相同格式的数据时，采用这种方法效率会更高。



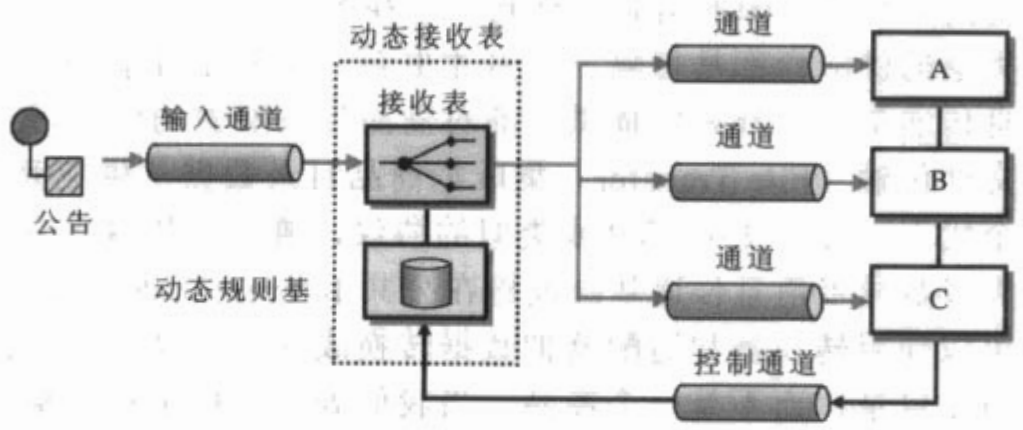
通过文件传输更新商品的种类数据

1.8.7 发布公告

为了改进业务，每隔一定时期需要向客户发布特定的公告。为了避免骚扰客户，系统允许每个客户指定他们感兴趣的消息。另外还要提供适当的功能，向特定的客户群发送特

定消息。例如，只向对某些商品有需求的客户发布这些商品的信息。需要向多个接收者传送信息时，可能会立即想到采用发布-订购通道。但是，发布-订购通道也存在一些缺点。首先，任何订购者都可以监听发布的消息，而不需知道发布者的情况。例如，我们不希望购买量小的客户收到为大客户发布的消息。其次，发布-订购通道只能在局域网中保证工作效率。如果要通过广域网发送数据，就必须为每个接收者发送一个单独的消息副本。如果接收者对这个消息不感兴趣，则会造成不必要的网络流量。

因此，必须寻找一种解决方案，能让订购者决定他们需要的订购首选项，并把独立的消息发送给感兴趣的（授权）客户。为了实现这项功能，我们使用了动态接收表（Dynamic Recipient List）。动态接收表结合了两个消息路由模式。接收表（Recipient List）就是一个路由器，它能把一个消息广播给一组接收者。接收表与发布-订购通道的主要区别在于，接收表会为每个接收者提供专门的处理，因此对于谁接收消息能有严格的控制。动态路由器（Dynamic Router）是一个能根据控制消息改变路由算法的路由器。这些控制消息可能是订购者指定的订购首选项。动态接收表就是由这两种模式组合得到的。



采用动态接收表发送公告

如果客户通过电子邮件接收公告，那么这些模式的实现可以使用电子邮件系统通常提供的邮件列表特性。每个接收者通道由一个电子邮件地址标识。同样的，如果客户想通过 Web 服务接口接收公告，那么每个接收者通道则由一个 SOAP 请求实现，通道的地址是 Web 服务的 URI。这个例子说明，用于描述解决方案设计的模式与特定的传输技术无关。

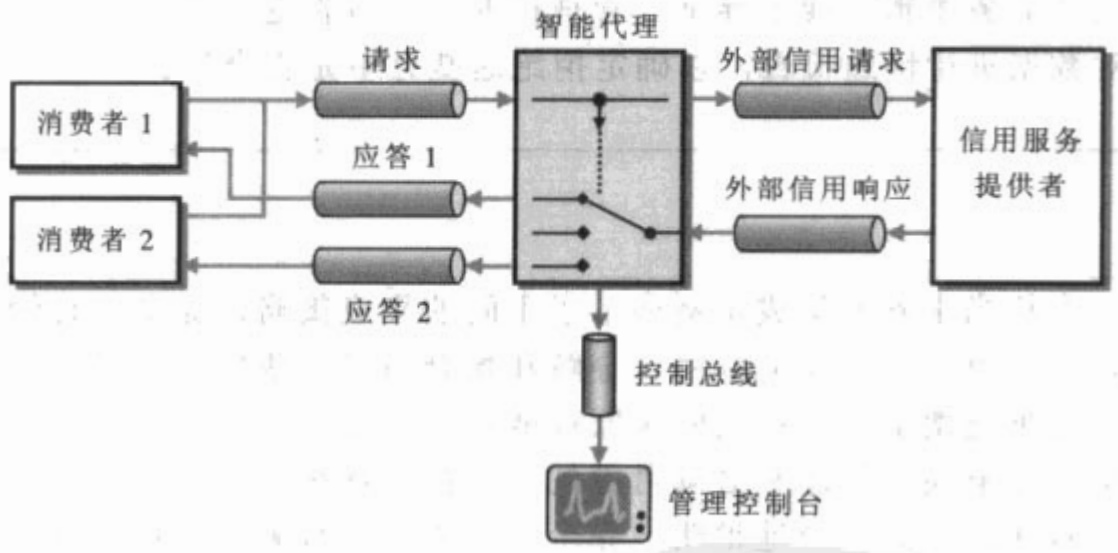
1.8.8 测试和监控

监控消息的正确执行是一个很关键的动作和支持功能。消息存储库可以提供一些重要的业务度量（metrics）信息，如履行一个订单的平均时间。但是，为了保证集成解决方案能成功运行，我们还需要了解更详细的信息。假设在改进的解决方案中，需要访问一家外部信用中介机构，以便更好地评估客户的信用等级。即使一个客户没有透支记录，但是如果这个客户的信用等级很差，我们同样很可能会拒绝该客户的订单。对于没有任何支付记录的新客户来说，这种做法非常有用。由于这项服务是由外部提供者提供的，所以我们必须为此付费。为了检查提供者的发货单是否正确，可能要跟踪自己的实际使用情况，并把

使用情况报告与发货单进行比较。不能只是检查订单个数，因为对于长期客户，业务逻辑可能不需要申请外部信用检查。另外，还要与外部服务提供者达成服务质量（QoS）协议。例如，如果响应时间超过了指定的时间，可能就不必为请求支付费用。

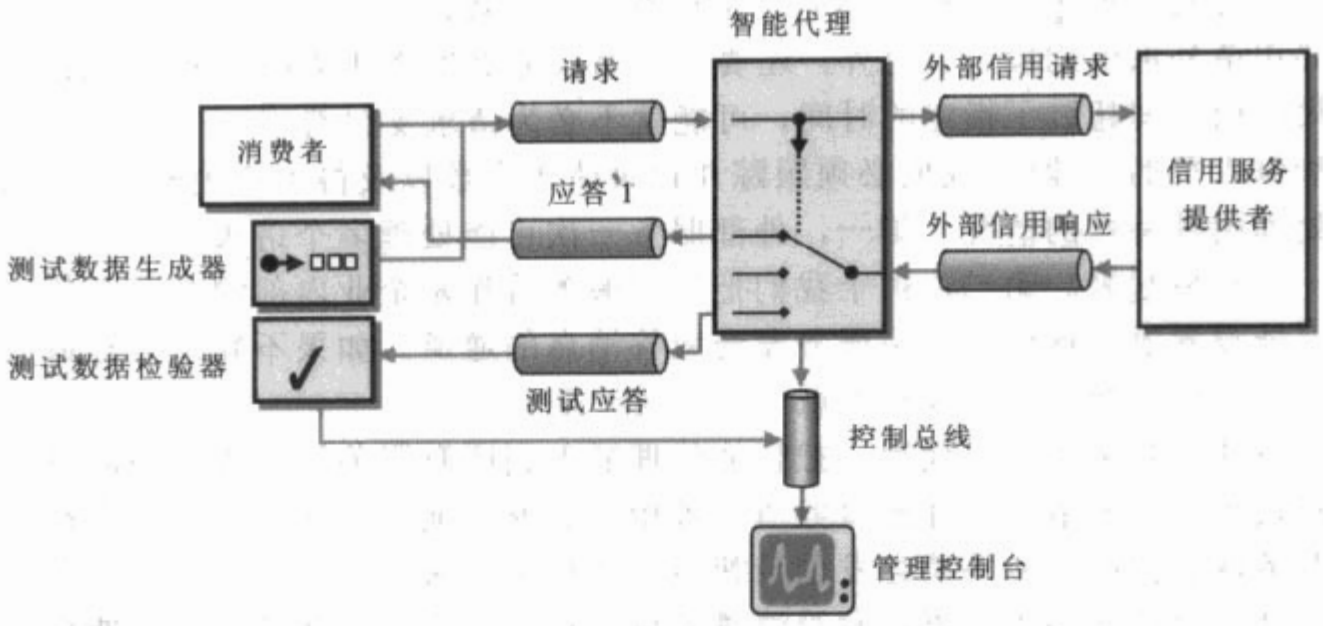
为了确保收费的合理性，我们必须跟踪并记录请求个数以及得到相关响应所需的时间。为此必须处理两种特殊的情况。第一，外部服务一次可能处理多个请求，因此要把请求和应答消息一一对应起来。第二，由于我们把外部服务看作是企业内部的共享服务，因此服务消费者要能设置返回地址，这是服务发送应答消息的通道。如果不知道应答消息应该发给哪个通道，就很难把请求与应答消息一一对应起来。

解决的方法还是采用智能代理。把智能代理插入到所有服务消费者与外部服务之间。智能代理拦截每个服务请求，并把服务消费者指定的返回地址替换为固定的应答通道。这样一来，服务就会把所有应答消息发送给智能代理指定的通道。由于代理存储有原始的返回地址，因此它会把应答消息再转发给消费者最初指定的通道。智能代理还能测量服务的耗用时间，即对外部服务发出请求到得到外部请求应答之间所需要的时间，并把这个数据发布到控制总线。控制总线与管理控制台连接，管理控制台负责收集来自不同组件的度量信息。



插入智能代理来跟踪响应时间

除了要跟踪外部信用服务的使用外，还要确保服务能正常工作。智能代理可以向管理控制台报告在指定的超时间隔内没有接收到应答消息的情况。有时，外部服务返回了应答消息，但是消息中包含的结果不正确，这种情况更难检测。例如，如果外部服务出现了错误，为每个客户返回的信用等级都是零，我们就会拒绝每一个客户的订单。为了防止这种情况的发生，可以采取两项措施。首先，要周期性地在请求消息流中插入一个测试消息（Test Message）。这个测试消息请求某个已知客户的信息等级。然后使用一个测试数据检验器（test data verifier）检查不仅能收到应答，而且应答消息的内容要正确。因为智能代理支持返回地址，测试数据生成器（test data generator）可以为测试应答指定一个特殊的应答通道，把测试应答与正常应答区分开来（如下图所示）。



插入测试消息以检查结果的正确性

另一种检测服务异常的有效策略是做统计采样。例如，我们估计因客户信息等级过低而造成的订单被拒概率平均不会超过 10%。如果连续拒绝的订单数超过五个，可能就意味着外部服务或某些业务逻辑出现了异常。管理控制台可以把这五个订单用电子邮件发送给管理员，他会对数据进行快速检查，以确定拒绝这些订单是否合理。

1.9 小结

本章通过一个相当丰富的集成示例展示了不同的集成策略，如文件传输、共享数据库和异步消息传递。这里介绍了如何路由、分解和聚合消息。我们引入了过程管理器来得到更大的灵活性，另外还增加了一些功能来监控解决方案是否正常运行。尽管这个例子的需求有所简化，但是要解决的问题和必须考虑的设计折衷都很实际。本章给出了一些解决方案的图形表示和描述，由此充分地说明，完全可以采用一种独立于特定开发商和特定技术的语言来描述各个解决方案，而且比高级顺序图更为准确。

本章介绍的集成示例主要针对的是如何连接已有的应用。对于如何发布和消费来自定制应用内部的消息，请参考第 6 章“中场演练：简单的消息传递”和第 9 章“中场演练：复合消息传递”。

对于这个解决方案设计中用到的各种模式以及许多相关的模式，本书后面的部分将提供详细的描述和代码实例。这些模式按其主要作用分为基本模式、通道模式、消息模式、路由模式、转换模式、端点模式和系统管理模式等。这种安排有利于按顺序阅读各种模式，也能方便查找个别的模式。

第2章

集成方式

2.1 引言

企业集成是指让不同的应用协同工作，提供一组统一的功能。这些应用可以是内部定制开发的，也可以从第三方开发商购买。它们可能运行在多台计算机上，分别有不同的平台，甚至在地理位置上也是分散的。有些应用可能由企业外的商业合作伙伴或客户运行。还有些应用在设计时可能没有考虑集成问题，并且也很难修改。由于存在这些问题以及诸如此类的其他问题，使得应用的集成变得非常复杂。本章探讨的各种集成方法有助于克服这些难题。

2.1.1 应用集成的标准

怎样才是好的应用集成？如果集成的需求始终保持不变，可能只存在一种集成方式。然而，与所有复杂的技术工作一样，应用集成涉及许多注意事项，在任何集成条件下都应该考虑到这些问题和相应的后果。

最基本的一条是究竟是否采用应用集成。如果只需要开发一个独立的应用、不与其他任何应用协作，就可以避免所有的集成问题。但是在现实情况中，即使是一个简单的企业也会存在许多应用，这些应用需要协同运行，以便为企业的员工、合作伙伴和客户提供统一的服务。

下面是另外一些主要的决策标准。

应用耦合——应该尽量减少所集成应用之间的依赖性，使每个应用的变化不会对其他应用造成影响。在第1章“运用模式解决集成问题”中介绍过，紧耦合的应用会对其他应用如何运行做出大量的假设。当应用发生变化，这些假设条件不再成立时，应用之间的集成就会中止。因此，集成应用的接口应该足够特定，可以提供所需的功能，但同时也要有足够的一般性，允许应用根据需要修改。

干扰性——当把某个应用集成到企业中时，开发人员会尽可能减少对应用的修改，同时尽量减少集成代码的数量。然而，还是要对应用做一定的修改以及编写新的代码，这是提供良好的集成功能所必需的，如果采用对应用影响最少的方法，也许无法为企业提供最佳的集成。

技术选择——不同的集成技术需要不同的专门软件和硬件，数量也不同。这些工具可

能价格不菲，且一旦采用可能很难再采用其他开发商的产品，还可能增加开发人员的学习曲线。但是从另一方面看，从头开始构建集成解决方案常常要比最初的设想付出更多的努力，做出重复性的工作。

数据格式——所集成的应用必须协商好所交换数据的格式。为了使用统一的数据格式而修改已有的应用可能会很困难，甚至不可能。而采用中间转换器就可以把仍然使用不同数据格式的应用连接起来。与之相关的问题是数据格式的演化及其可扩展性——数据格式如何随时间改变，以及这些改变会对应用造成怎样的影响。

数据的时间特性——当某个应用决定共享一些数据，同时其他应用拥有这些数据时，集成要尽可能缩短共享的时间。采用频繁交换少量数据的方式可以满足这一需求。但是，把一个大的数据集划分为小的数据块传送会降低系统效率。数据共享造成的延迟会影响集成设计。理想的情况是，接收方的应用在共享数据准备好时，能够立即得到通知。共享的时间越长，应用之间不同步的几率就越大，集成就会变得更复杂。

数据或功能——许多集成解决方案不但允许应用共享数据，还能共享功能，因为共享功能可以在应用之间提供更好的抽象。尽管调用远程应用中的功能与调用本地功能看上去一样，但其工作方式截然不同，这对集成工作的表现有很大的影响。

远程通信——计算机处理一般是同步的，也就是说，进程会等待子进程执行完毕后再执行。但是，调用远程子进程要比调用本地子进程慢得多，因此进程可能不希望等待子进程完成；相反地，可能希望采用异步方式调用子进程，也就是说，启动子进程后继续完成自己的处理。异步特性可以提高解决方案的效率，但是这种解决方案的设计、开发和调试更为复杂。

可靠性——与本地函数调用相比，远程连接不仅慢，而且更不可靠。当一个进程调用同一应用内部的子进程时，能够确保子进程是可用的。但是当进行远程通信时，则没有肯定的把握；远程应用可能根本没有运行，或者网络出现了临时故障。可靠的异步通信能够保证源应用继续处理其他工作，并相信远程应用以后总会运行。

因此，正如你所看到的，在选择和设计集成方式时，有好多不同的标准需要考虑。现在的问题是，到底哪种集成方式能最好地适应这些标准？

2.1.2 应用集成的选择

没有哪一种集成方式能很好地实现上述所有标准。因此，随着时间的推移，很多集成应用的方法都得到了改进。归纳起来可分为以下四种主要方式。

文件传输——让每个应用生成共享数据文件，供其他应用使用，并使用其他应用生成的共享数据文件。

共享数据库——让应用把要共享的数据存储在一个公共数据库中。

远程过程调用——让每个应用公开提供自己的一些过程，使它们能够被远程调用，应用通过调用这些过程来执行操作并交换数据。

消息传递——让每个应用连接到一个公共的消息传递系统上，并通过消息来交换数据和调用行为。

本章把每种方式作为一种模式。这四种模式面对的是相同的问题（需要集成应用），并且所面对的集成环境也非常相似。之所以会存在这些不同的集成模式，这是由要寻找一种更好的解决方案带来的。每种模式都建立在前一种模式的基础上，目的是为了找到一种更成熟的方法，来解决以前的集成方法所存在的问题。因此，模式的演化顺序也反映出其成熟度的增加，但同时复杂性也呈上升趋势。

选择集成方式的技巧是，不要每次都选择相同的方式，而是要根据特殊的集成需求来选择最好的方式。每种方式都有自己的优点和缺点。应用的集成可以使用多种方式，让每种集成方式各尽所能，使每个集成环节都能充分发挥相应集成方式的优点。同样地，一个应用可以使用不同的方式与不同的应用集成，应当选择最适合其他应用的方式。因此，许多集成方法都可以被看成是多个集成方式的组合。为了支持这种类型的集成，许多集成产品和 EAI 中间件产品都采用了多种集成方式，所有这些都被有效地隐藏在产品的实现中。

本书后面介绍的模式详细描述了消息传递集成方式。之所以把重点放在消息传递上，这是因为我们相信，它既能为各种集成标准提供一个很好的平衡点，同时也是一种最难实现的集成方式。目前，消息传递仍然是我们了解最少的集成方式，也是一种随着模式发展逐步成熟的技术，这些模式能很好地解释如何让消息传递充分发挥作用。最后一点，消息传递是许多商业 EAI 产品的基础，因此介绍如何使用消息传递对于学习如何使用这些产品也有很大的帮助。本章的重点是强调应用集成所涉及的问题，以及如何利用消息传递解决这些问题。

2.2 文件传输

作者：Martin Fowler



企业往往有多个应用，分别采用不同的语言和平台独立构建而成。

如何集成多个应用，使它们能协同工作并交换信息？

在理想情况下，你可能设想一个企业只运行一个完整的软件，从设计到开始工作都采用一种统一、一致的方式。然而在实际情况中，即使是最小的操作也不会是这样。存在许多软件分别处理企业中不同方面的问题，这是由许多原因造成的。

- 人们购买了由外部组织开发的软件包。
- 不同的系统是在不同时间建立的，因此它们采用了不同的技术。
- 不同的系统是由不同的人开发的，这些人的经验和偏好促使他们采用了不同的方法

来构建应用。

- 把应用开发出来使之能发挥作用，这比保证它的集成特性重要得多，尤其是集成并不能给正在开发的应用增加任何价值时，更是如此。

因此，要在截然不同的应用之间共享信息，对任何组织来说都是一件头疼的事。这些应用可能采用了不同的编程语言，工作在不同的平台上，并且对如何进行业务操作设定了不同的假设条件。

把这样的应用联系在一起，需要从业务和技术两个层面对如何连接应用有深入的理解。如果不需要了解各个应用如何工作（或者尽量减少这个必要），集成起来就要容易得多。

真正需要的是一种公共的数据传输机制，可以由各种语言和平台使用，而且对每种语言和平台来说都很自然。这样的机制尽可能不需要特殊的硬件和软件，而只是充分使用企业已有的资源。

文件是一种通用的存储机制，各种企业操作系统都提供了这种机制，而且得到了所有企业语言的支持。最简单的方法莫过于使用文件来集成应用。

由各个应用产生文件，其中包含供其他应用使用的信息。集成人员只是负责把文件转换成不同的格式。要根据业务自身的特点定期地产生文件。



使用文件的关键在于确定文件的格式。一个应用的输出正好是另一个应用所需要的，这种情况很少碰到，因此在传输过程中必须对文件进行一定的处理；也就是说，使用文件的应用不但要读它，还要使用处理工具加以处理。因此，经过一段时间后，标准的文件格式诞生了。大型机系统一般使用 COBOL 文件系统格式的数据输入；UNIX 系统则使用基于文本的文件。当前采用的方法是使用 XML 格式的文件。目前已经有针对各种文件格式的阅读器、书写器和转换工具，这个行业已成气候。

与文件相关的另一个问题是何时生产与消费文件。由于创建和处理文件都需要付出一定的努力，因此人们都不大愿意太过频繁地使用文件。一些常规的业务只是每晚、每周、每个季度这样定期地创建和处理文件。得到一个新文件时，应用往往无甚举动，而是在合适的时间才处理这个文件。

文件系统具有的最大优点是集成人员不必了解应用的内部细节。一般由应用小组自己来提供文件。如果使用了软件包，选择的余地则是有限的，尽管如此，还是需要与集成人员协商确定文件的内容与格式。随后由集成人员负责处理其他应用所需的转换，或者把文件交给使用文件的应用，由它们自己决定如何处理和读取文件。因此，不同的应用之间关

系非常松散。每个应用可以自由地进行内部修改，而不会影响其他应用，只要能保持所生产的文件内容及格式不变就可以。这样，文件就有效地成为各个应用的公共接口。

文件传输之所以简单，部分原因是不再需要额外的处理工具或集成软件包，但是开发人员必须自己完成大量的处理工作。应用必须协商建立共同的文件命名约定，还要确定文件放在哪些目录中。文件的书写器必须实现一个策略，以保证文件名是唯一的。应用必须协商好由谁负责删除旧文件，而且负责删除文件的应用必须清楚什么时候文件才会过期，不会再使用。应用还要实现锁定机制或遵循某种定时约定，以确保一个应用不会在另一个应用正在写文件时读这个文件。如果所有的应用无法存取相同的磁盘，那么必须有一些应用负责将文件从一个磁盘传输到另一个磁盘。

采用文件传输最明显的问题是很少完成更新，因此系统很可能不同步。假设客户管理系统每个晚上会处理地址的修改并产生一个文件，但是在同一天账单管理系统可能会将账单发送到老地址。有时不同步并不会造成大的问题。对人，甚至对计算机来说，获取返回的信息允许有一定的延迟。但是在某些情况下，使用过时的信息就会造成灾难。决定何时生产文件时，必须把客户对信息的刷新需求考虑在内。

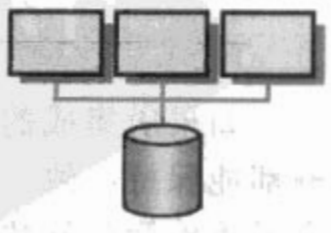
实际上，文件过时的最大问题往往在软件开发人员自己身上，他们必须经常与不完全正确的数据打交道。由此带来的不一致性是最难解决的。如果客户在同一天用两个不同的系统修改了他的地址，但是其中一次修改出现了一个错误，把街道名弄错了，那么对客户来说，他就有两个不同的地址。开发人员必须找出一种办法解决这样的问题。文件传输间的周期越长，这种问题造成的危害可能就越大。

当然，你可以把产生文件的频率提高。实际上，也可以把消息传递看作是文件传输，只要应用中出现数据修改就产生一个文件。现在的问题是如何管理已生成的文件，使它们都能得到读取而不会丢失。基于文件的集成方法是做不到这一点的，这主要是因为处理一个文件需要付出昂贵的资源代价，所以不允许在短时间内产生大量的文件（否则就有可能丢失）。如果要使用非常细粒度的文件，更简单的做法是采用消息传递。

为了加快数据的共享，并且保证一致的数据格式，可以使用共享数据库。要把应用的功能（而不是数据）集成起来，可以使用远程过程调用。为了能够频繁交换少量数据，以前一般使用远程调用，现在可以使用消息传递。

2.3 共享数据库

作者：Martin Fowler



一个企业往往有多个应用，分别采用不同的语言和平台独立构建而成。企业需要信息

能被快速、一致地共享。

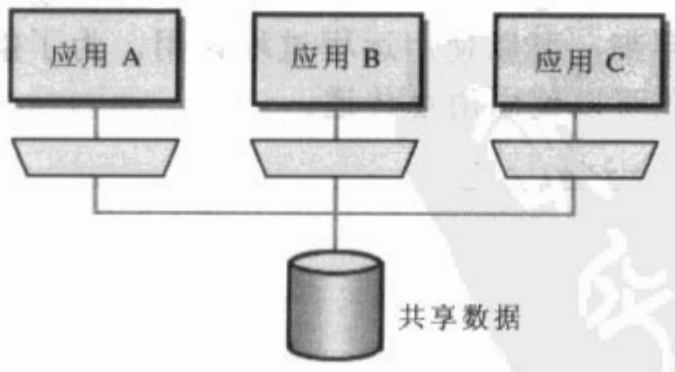
怎样才能把多个应用集成起来，使它们能协同工作并交换信息？

利用文件传输，应用能共享数据，但是缺乏时间特性（timeliness）——然而集成的时间特性往往是很关键的。如果数据修改不能迅速传递给所有的应用，那么由于数据的过时，可能会造成错误。对现代商业来说，必须让每个人都能获得最新的数据。这不但能减少错误，还能增强人们对数据本身的信任。

快速更新还能更好地解决不一致问题。同步的越频繁，那么造成不一致的可能性就越小，而且处理不一致性的代价就会越低。但是无论更新的速度有多快，仍然会存在其他问题。如果在很短的时间内地址被不一致地更新了，那么怎样才能确定哪个地址是对的呢？针对每份数据，你可以指出某个应用是产生该数据的主要来源，但是你还必须记住主要由哪个应用处理哪些数据。

文件传输可能不会对数据格式做足够的要求。许多集成问题就是由于分析数据的方法不兼容造成的。它们往往表现为细微的业务差异，但是会带来严重的影响。一个地质数据库可以把油井定义为能够或不能生产石油的单眼钻孔。一个产品数据库可能把油井定义为由一台设备管理的多个钻孔。这些语义上的不一致会造成问题，这些问题比数据格式不一致带来的问题更难解决（对此问题的更深入讨论，请参阅《Data and Readlity》[Kent]）。我们真正需要的是一个集中、一致的数据存储库，能让所有应用共享，每个应用都能在需要的时候访问任何共享数据。

通过把应用的数据存储在一个共享数据库中来集成应用，并制定数据库模式来处理不同应用的各种需求。



如果所集成的所有应用都依赖于相同的数据库，那么可以肯定地说这些应用在任何时刻都能保持一致。从不同的应用对同一数据进行同时更新时，数据库的事务管理系统能非常妥善地管理好数据的一致性。由于更新的时间间隔很小，任何错误都能更容易地发现和更正。

基于 SQL 的关系数据库得到了广泛使用，因此更适合作为共享数据库。几乎所有应用开发平台都支持 SQL，而且往往配备有完善的开发工具，因此就没有必要再为多种文件格式操心。由于所有的应用都会使用 SQL，开发人员也不必再去掌握另一门技术。

因为每个应用使用的是相同的数据库，这就避免了语义上的不一致问题。在软件运行并产生大量不一致数据之前，开发人员就必须面对语义的不一致问题并想办法解决，而不是任由这些问题泛滥，再想用转换来解决就很困难了。

共享数据库面临的最大困难之一是，要为共享数据库提出一个合适的设计。要提出一个统一的数据库模式来满足多个应用的需要，是一项非常艰难的任务，设计出的模式往往让应用程序员感到难以着手。即使设计一个统一模式的技术难度不算太大，在行政方面还可能存在着更严重的困难。如果一个关键应用要使用统一模式，而为此很可能延期，它往往就要承受压力与其他应用分离。不同部门人员之间的分歧常常会加剧这一问题。

另外，对共享数据库的更大限制来自外部的已封装应用。大多数封装好的应用只使用自己的数据库模式。即使提供了一定的调整空间，但是仍然会给集成人员带来意想不到的更多限制。更糟糕的是，软件开发商通常在软件的每个新版本中都会修改数据库模式。

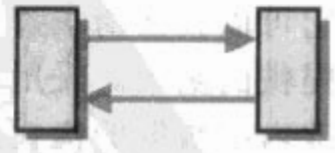
这个问题还会延伸到开发之后的集成。即使你已经把所有的应用组织起来，如果出现公司合并，同样的集成问题还会产生。

多个应用可能通过共享数据库频繁地读取和修改相同的数据，这会使数据库成为一个性能瓶颈，并且当每个应用都对数据完成互斥操作（即不允许其他应用访问数据）时，还会引起死锁。当应用分布在不同的位置、通过广域网访问同一个共享数据库时，访问速度可能过慢，以至于并不实用。分布式数据库允许各个应用通过本地网络连接访问数据库，但是它也带来了一个问题：应该把数据存储在哪台计算机上。分布式数据库如果存在锁定冲突，很容易在性能方面带来灾难。

要集成应用的功能而不是数据，可以使用远程过程调用。为了支持少量数据的频繁交换，其中每种数据类型均使用不同的格式，而没有采用统一的数据模式，则可以使用消息传递。

2.4 远程过程调用

作者：Martin Fowler



一个企业往往有多个应用，分别采用不同的语言和平台独立构建而成。企业需要以一种响应的方式共享数据和过程。

如何集成多个应用，使它们能协同工作并交换信息？

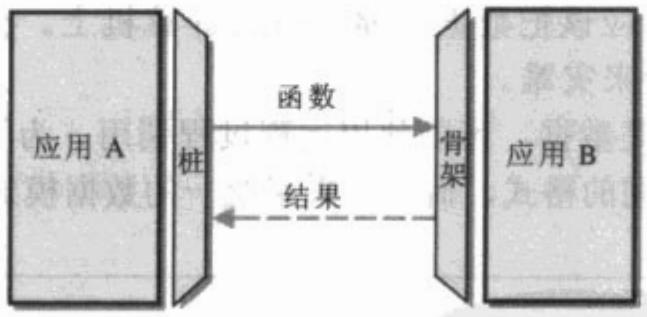
通过文件传输和共享数据库，应用能够共享它们的数据，这是应用集成的一个重要部分，但是仅仅是共享数据往往还不够。数据的改变往往需要不同的应用采取相应的动作。例如，改变地址可能只需要简单地修改数据，也可能会触发相应的注册和法律程序，按照不同的法律权限执行不同的法规。要让一个应用直接调用其他应用中的过程，它必须深入了解其他应用的内部细节。

这就给应用的设计带来一个经典的两难境地。程序设计最强大的结构化机制之一是封装，模块通过函数调用接口来隐藏其数据。采用这种方式，当数据发生改变时，模块可以截获数据的变化，并完成各种相应的操作。共享数据库提供了一个很大的、未封装的数据结构，因此较难做到数据隐藏。文件传输允许应用在处理文件时对数据的变化作出响应，但是这个过程是滞后的。

共享数据库采用的是未封装的数据结构，这一点增加了维护集成应用的难度。应用中的许多修改都会触发数据库的改变，而数据库的改变又会对其他应用产生连锁反应。因此，使用共享数据库的组织一般不愿意修改数据库，也就是说应用的开发工作很少考虑业务的改变需求。

我们需要这样一种机制，能让一个应用调用另一个应用的函数、传递要共享的数据，并调用函数来告诉接收应用如何处理数据。

把每个应用都开发为一个大规模的对象或组件，其中包含封装的数据。要提供一个接口，使其他应用可以与当前应用交互。



远程过程调用运用封装原则来集成应用。如果某个应用需要其他应用拥有的一些信息，它可以直接请求那个应用。如果某个应用需要修改另一个应用的信息，可以调用那个应用提供的函数。采用这种方式，每个应用都能维护自己数据的完整性。此外，各应用可以修改内部数据的格式，而不会影响其他应用。

许多技术，如 CORBA、COM、.NET Remoting 和 Java RMI 都实现了远程过程调用。分别有多少系统支持这些方法以及使用这些方法的难易程度都是不同的。其中一些技术还提供了额外的功能，如事务处理。从应用范围来看，目前最受欢迎的是 Web 服务，它使用了诸如 SOAP 和 XML 等标准。Web 服务的最大特点是，可以很容易地利用 HTTP 协议，而这种协议能够轻松地穿过防火墙。

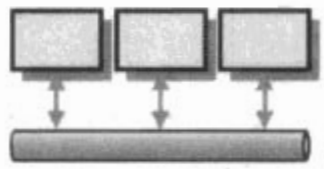
通过方法把数据封装起来，就能更好地解决语义不一致问题。应用可以为相同的数据提供不同的接口，使不同的客户看到不同的数据样式，甚至更新也可以使用多个接口。相对于关系视图，这样更能从多个角度提供视图。但是，要让集成人员增加转换组件就太麻烦了，所以每个应用都必须与其“相邻”应用协商接口。

由于软件开发人员已经习惯了过程调用，所以对远程过程调用并不感觉陌生。但是，这可能正是采用这种方法的缺点而不是优点。远程过程调用和本地过程调用在性能和可靠性方面存在显著的差异。如果开发人员不了解这些差异，远程过程调用很可能会导致缓慢而且不可靠的系统（见[Waldo]、[EAA]）。

利用封装，可以不需要大的共享数据结构，从而有助于减少应用的耦合，尽管如此，应用还是紧密地耦合在一起。每个系统提供的远程调用企图把不同的系统组成一体。特别是存在时序问题，即要按特定的顺序完成某个工作，这使得独立地修改系统变得相当困难。这些类型的问题放在单个应用中并不突出，但是放到集成应用中就很严重了。开发人员往往按照设计单个应用的方式来设计集成系统，而没有意识到相应的规则已经发生了很大的变化。

为了以更加松耦合的异步方式来集成应用，可以使用消息传递来频繁交换少量的数据，而以前这一般采用远程过程调用实现。

2.5 消息传递



一个企业往往有多个应用，分别采用不同的语言和平台独立构建而成。企业需要以一种响应的方式共享数据和过程。

如何集成多个应用，使它们能协同工作并交换信息？

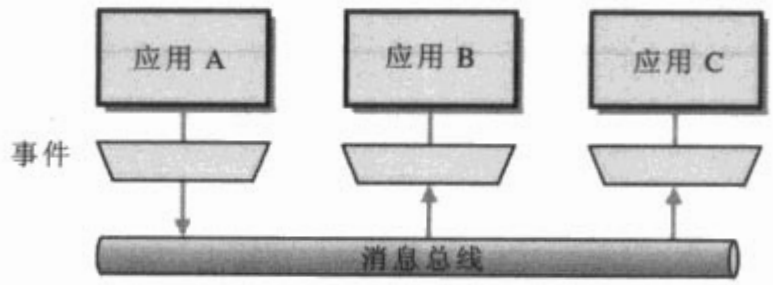
利用文件传输和共享数据库，应用能够共享它们的数据，但不能共享功能。远程过程调用使应用能够共享功能，但是这会让应用紧耦合。集成的难点在于，要让不同的系统尽可能及时地协作，而且无论从应用执行还是应用开发来看，都不会使所集成的系统变得不可靠。

文件传输使得能够很好地保持应用的解耦合，但是在时间特性方面要付出代价。系统不能保持彼此的同步。协作行为太过缓慢。共享数据库能按照响应方式把数据集中在一起，但是也要付出代价，这样会把所有应用与数据库耦合。同样无法处理协作行为。

面对这些问题，远程过程调用看起来是一种更好的选择。但是把单个应用的模型扩展到应用集成中，会产生很多其他的缺陷。这些缺陷都是由分布式开发造成的。尽管远程过程调用看起来像本地调用，但是表现完全不同。远程调用更慢，而且更容易失败。当多个应用跨企业通信时，我们不希望一个应用的失败引发所有其他应用也失败。而且，不希望在假设调用速度很快的前提下设计系统，另外不想让每个应用都知道其他应用的细节，甚至只是接口方面的细节。

我们真正需要的有些类似于文件传输，大量的小数据包能迅速地产生，并且方便地传送，当有新的数据包到达时，接收应用会自动得到通知。数据传输需要一种重传机制，以确保传送的成功。存储数据的任何磁盘结构或数据库细节要对应用隐藏，这样一来，与共享数据库不同，存储模式和细节可以很容易地修改，从而反映企业不断变化的需求。类似于远程过程调用，一个应用可以向另一个应用发送一个数据包，从而调用其他应用的相应操作，不过这里不那么容易失败。要让发送者不必等待接收者，尤其是必须要重传数据时，应该采用异步传输。

运用消息传递，可以使用可定制的格式频繁地、立即地、可靠地、异步地传输数据包。



异步消息传递从根本上是为了解决分布式系统面临的实际问题。发送一个消息时，不需要通信双方同时在线或者都准备好。而且，采用异步通信方式有助于开发人员认识到，与远程应用协作会更慢，促使他们设计出高内聚（把大量操作本地化）和低耦合（有选择地执行远程操作）的组件。

消息传递系统同样能保持文件传输的解耦合性。消息能在传送中得到转换，发送者与接收者都不用了解具体的转换。解耦合有助于集成人员选择不同的消息传递方式，如把消息广播给多个接收者，或把消息路由给多个接收者之一或者其他的拓扑结构等等。这样就能把集成设计与应用开发分离开来。从人的角度讲，把应用开发与应用集成分开考虑更符合人性。

转换（transformation）是指，不同的应用可能有差别很大的概念模型，因此会出现语义上的不一致问题。但是，消息传递的观点是，共享数据库中用来避免语义不一致的策略太复杂了，很难具体实现。而且，语义不一致问题往往发生在第三方应用上，或者出现在由于公司合并而增加的应用上，因此消息传递方法就是要解决这样的问题，而不是设计应用来避免这种问题。

通过频繁发送小消息，应用不仅能共享数据，还可以在操作上相互协作。如果要求接收到保险索赔声明的同时启动某个过程，可以在接收到声明时立即发送一个消息，请求所需的信息，并迅速作出响应。尽管这种协作方式不像远程过程调用那么快，但是在处理消息以及返回响应时，调用者不需要等待。而且消息传递并不像很多人想像得那样慢，许多消息解决方案最初都应用于金融服务行业，这些行业要求消息传递系统每秒内要传送成千上万的股票报价或交易信息。

这本书是有关消息传递的，因此你可以肯定，我们认为消息传递通常是实现企业应用集成的最佳途径。但是，你不要认为采用消息传递不存在任何问题。尽管频繁的消息传递能减少文件传输遇到的许多不一致问题，但是并没有完全根除。如果系统未能同时更新，也会带来一些滞后问题。大多数开发人员没有学过异步设计，因此取而代之，会采用许多不同的规则和技术。与异步应用环境（如 X Windows）相比，基于消息传递上下文编写程序更为容易，尽管如此，掌握异步仍然有一定的学习曲线。另外，在这种环境中，测试和调试也更加困难。

利用消息传递能够转换消息，这有一个明显的好处，即有助于应用相互解耦合，而不像远程过程调用那样。但是这种独立性也意味着，集成人员必须编写大量麻烦的代码才能把不同的系统组合起来。

一旦决定使用消息传递方法集成系统，还要考虑许多新的问题以及相应的做法。

如何传输数据包？

发送者通过一个连接发送者和接收者的消息通道来发送一个消息，从而向接收者发送数据。

如何确定消息的发送地址？

如果发送者不知道消息发送到哪里，它可以把数据发送给一个消息路由器，由它把数据转发给适当的接收者。

怎样确定需要使用的数据格式？

如果发送者和接收者不能达成一致的数据格式，发送者可以把数据直接发送给消息转换器，由它把数据转换成接收者的格式，并进一步将数据转发给接收者。

如果你是一位应用开发人员，该如何把自己的应用与消息传递系统连接？

想要使用消息传递的应用必须采用消息端点来完成具体的发送和接收。

第3章

消息传递系统

3.1 引言

在第2章“集成方式”中，我们讨论了可以采用哪些方法把应用相互连接起来，其中包括消息传递。消息传递通过采用异步通信方式可以实现应用的松耦合，由于所集成的两个应用不一定要同时运行，这就使通信变得更加可靠。基于消息传递，消息传递系统会负责把数据从一个应用传输到另一个应用，因此应用可以把重点放在需要共享什么数据，而不是如何共享数据。

3.1.1 消息传递的基本概念

和大多数技术一样，消息传递也包含一些基本概念。一旦理解了这些概念，即使你对如何使用的细节不完全清楚，也能搞懂这项技术的含义。以下是消息传递的基本概念。

通道——消息传递应用通过消息通道来传送数据，通道是一个虚拟管道，将一个发送者连接到一个接收者。新安装的消息传递系统通常不包含任何通道；首先必须确定应用要如何通信，然后再创建与之相适应的通道。

消息——消息是能通过通道传送的一个原子数据包。因此，为了传送数据，应用必须把数据分解为一个或多个数据包，把每个数据包包装为消息，然后通过通道发送消息。同样地，接收者的应用在接收到消息后，必须从中取出数据并进行处理。消息系统将会不断尝试发送消息（从发送者传送给接收者），直到发送成功。

管道和过滤器——在最简单的情况下，消息传递系统会把消息直接从发送者的计算机传送给接收者的计算机。但是，在消息从最初的发送者那里发出，直到最后的接收者接收到以前，往往需要对消息完成某些动作。例如，假设接收者所期望的消息格式与发送者使用的消息格式不同，那么必须对消息进行验证和转换。管道和过滤器体系结构描述了如何使用通道将多个处理环节链接起来。

路由——在一个大型企业中，可能拥有许多应用，还有许多连接这些应用的通道，消息在到达最终目标前可能要经过好几个通道。消息经过的路由可能非常复杂，以至于最初的发送者无法确定采用哪个通道能将消息传送给最终的接收者。因此，最初的发送者将把消息发送给消息路由器，这是一个应用组件，能够取代管道和过滤器体系结构中的过滤器。路由器会决定如何在通道拓扑结构中“导航”，把消息发送给最终的接收者，或者至少把消

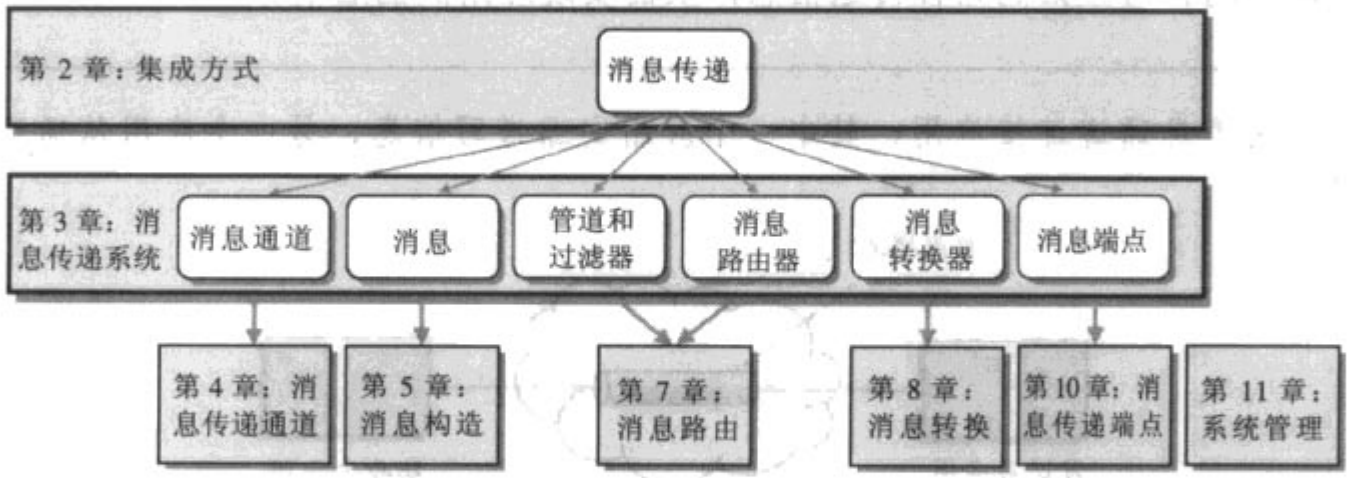
息发送给下一个路由器。

转换——各种应用可能对相同的概念数据采用了不同的数据格式：发送者采用一种消息格式，但接收者却希望采用另一种消息格式。为了调解这一矛盾，消息必须经过一个中间过滤器，即消息转换器，由它把消息从一种格式转换为另一种格式。

端点——大多数应用没有提供与消息传递系统交互的内置功能。与此不同，它们必须包含一层代码，这些代码不仅知道应用如何工作，还了解消息传递系统如何工作，从而能够在应用和消息传递系统之间架桥，使它们能协同工作。这种桥接代码就是一组协同的消息端点，支持应用发送和接收消息。

3.1.2 本书的组织

本章介绍的模式提供了基本的术语，能帮助你理解如何使用消息传递实现企业集成。后面的各章都建立在这里的某个基本模式基础之上，并对相关主题进行更深入的讨论。



根模式与各章的关系

通读本章可以了解消息传递的主要内容。如果想更详细地了解这些主题，可直接阅读与特定模式相关的章节。

3.2 消息通道

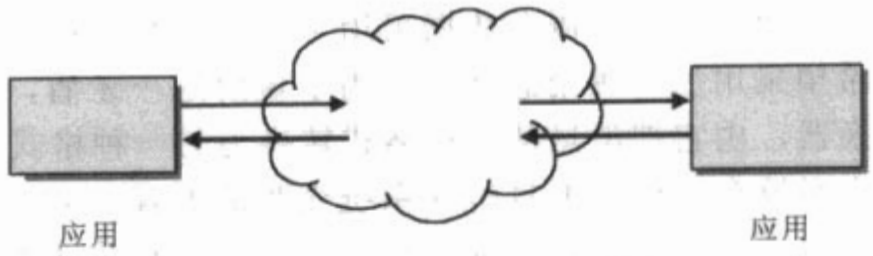


一个企业有两个不同的应用，需要使用消息传递进行通信。

一个应用如何使用消息传递与另一个应用通信？

一旦一组应用可以使用一个消息传递系统，你很可能会认为：只要需要，在任何时刻，任何应用都能与其他应用通信。然而，消息传递系统并没有把所有的应用魔术般地

连接起来。



应用被魔术般地连接起来

同样地，应用不是随机地把信息抛给消息传递系统，由其他应用随机地抓取所碰到的信息（即使这种方式可行，其效率也会非常低）。相反，发送信息的应用知道信息的类型是什么，接收信息的应用并不是查找所有类型的信息，而只是查找它们能使用的指定类型的信息。因此消息传递系统不是一个大桶，任由应用向它倾倒和抓取信息。它是一组连接，按照预先确定的、可预知的路径传送信息，从而实现应用间的通信。

使用消息通道连接应用，其中一个应用向通道写信息，另一个应用从通道读信息。



当一个应用有信息需要传送时，它并不是把信息注入到消息传递系统中，而是把信息添加到一个特定的消息通道中。接收信息的应用不是仅从消息传递系统中随机地获取信息，而是从一个特定的消息通道中获取信息。

发送信息的应用并不需要知道最终由哪个应用获取信息，但是它可以确信获取信息的应用肯定对信息感兴趣。这是因为，对于应用想传送的各种类型的信息，消息传递系统分别有不同的消息通道。一个应用发送信息时，它并不是把信息随机地添加到任何可用的通道上，而是把信息添加到与其类型匹配的特定通道。同样地，对于想接收特定信息的应用，它并不是随机地选择某个通道获取信息，而是根据所需的信息类型选择相应的通道。

通道是消息传递系统中的逻辑地址。通道如何具体实现，依赖于消息传递系统产品及其产品本身的实现。也许每个消息端点都与其他任何端点有一个直接的连接，也可能是通过一个中心集线器进行连接。也许多个独立的逻辑通道被配置为同一个物理通道，但是该通道知道具体每个消息的目标是什么。所定义的逻辑通道组对应用隐藏了这些配置细节。

消息传递系统不会自动为应用通信预先配置所有必需的消息通道，而要由设计应用和

应用之间通信的开发人员来决定，究竟需要什么通信通道。然后，由负责安装消息传递系统软件的系统管理员进行配置，建立应用所需的通道。尽管某些消息传递系统实现支持在应用运行时也能创建新通道，但是除了创建通道的应用外，其他应用也必须知道新的通道，这样才能使用这个新通道，所以这种实时创建通道的做法没有太大的用处。通道的个数及其用途一般在部署阶段就已经确定了（当然也存在例外情况，请参见第 4 章“消息传递通道”）。

消息传递术语浅谈

通过消息通道进行通信的应用称作什么？目前对它有很多种叫法，但含义是相同的。最常用的叫法是发送者（sender）和接收者（receiver），应用把消息发送给消息通道，并由另一个应用接收该消息。其他常用的叫法是生产者（producer）和消费者（consumer）。你还会看到发布者（publisher）和订购者（subscriber）的说法，但是这样的应用一般与发布-订购通道连接，并且以一般方式使用。有时可以称某个应用监听（listen）某个通道，另一个应用则与该通道会话（talk）。在使用 Web 服务时，常常会谈到请求者（requester）和提供者（provider）。这些术语通常是指请求者向提供者发送一个消息，并接收其响应。在更早的时候，我们把它们称为客户（client）和服务器（server），尽管这些术语的含义相同，但是称为“客户”与“服务器”不算很好。

现在情况变得更容易混淆了。与 Web 服务打交道时，把消息发送给服务提供者的应用一般被称为服务的消费者，尽管它是在发送请求消息。可以这样来理解，消费者向提供者发送一个消息，然后消费返回的响应。这种含义的术语只是在远程过程调用的环境中使用。发送或接收消息的应用可以被称为消息传递系统的客户（client），更专用的术语是端点（endpoint）或消息端点（message endpoint）。

当开发人员第一次使用消息传递系统时，常常不知道应该如何创建一个通道。开发人员可以编写 Java 代码，调用 JMS API 中定义的 createQueue 方法；也可以编写.NET 代码，其中包含 new MessageQueue 语句，但是这两种代码都不会真正在消息传递系统中分配一个新的队列资源。相反，它们只是简单地实例化一个运行时对象，通过它访问消息传递系统中已经创建的资源，该资源是通过消息传递系统的管理工具创建的。

为消息传递系统设计通道时，还要记住一个问题：通道尽管很廉价，但并不是免费的。应用需要多个通道传送不同类型的信息，而且要把相同的信息传送给大量其他应用。每个通道都需要内存来存储消息，持久通道还需要占用磁盘空间。即使一个企业系统有无限的内存与磁盘空间，任何消息传递系统实现都会对通道数量做出严格或实用的限制（只有不超过此限制的情况下，才能提供一致的服务）。因此，当应用需要新通道时，应该拟定一个创建计划，但是当应用需要数千个通道或者需要扩展到数千个通道时，就需要选择一个扩展性很强的消息传递系统实现，并测试其可扩展性，以确保满足你的需要。

通 道 名

如果通道是逻辑地址，那么这些地址具体是什么样的呢？在大多数情况下，具体的答案要由消息传递系统实现来决定。不过，通道一般采用字母数字组成的名字命名，如MyChannel。许多消息传递系统支持层次性的通道命名机制，从而能像文件系统的文件夹或子文件夹那样组织通道。例如，MyCorp/Prod/OrderProcessing/NewOrders 表示一个在 MyCorp 公司的生产应用中使用的通道，其中包含新订单。

有两种不同类型的消息通道：点对点通道和发布-订购通道。把不同的数据类型混合在一个通道上会引起许多混乱。为了避免这种情况，可以使用不同的数据类型通道。选择性消费者（Selective Consumer）使一个物理通道扮演多个逻辑通道。对于使用消息的应用，如果有一个对应非法消息的特殊通道，即非法消息通道（Invalid Message Channel），往往很有帮助。那些希望使用消息传递但是无法访问消息传递客户的应用，仍然可以使用通道适配器连接到消息传递系统。一组得到良好设计的通道可以形成一个消息总线，这相当于为所有应用提供的一个消息传递 API。

例：股票交易

当股票交易应用进行一项交易时，它把请求发送到对应交易请求的消息通道中。另一个负责处理交易请求的应用将在同一个消息通道中查找所能处理的消息。如果发出请求的应用需要获取股票的报价，则可能会使用另外一个消息通道，即专门为股票报价设计的通道，这样报价请求就能与交易请求分开。

例：J2EE JMS 参考实现

现在让我们来看看如何用 JMS 创建一个消息通道。J2EE SDK 提供了 J2EE 服务的一个参考实现，其中包括 JMS。参考服务器可以用 j2ee 命令启动。消息通道必须使用 j2eeadmin 工具来配置。这个工具既可以配置队列（Queue），也可以配置主题（Topic）。

```
j2eeadmin -addJmsDestination jms/mytopic topic
j2eeadmin -addJmsDestination jms/myqueue queue
```

一旦通道得到管理（创建），就可由 JMS 客户代码访问。

```
Context jndiContext = new InitialContext();
Queue myQueue = (Queue) jndiContext.lookup("jms/myqueue");
Topic myTopic = (Topic) jndiContext.lookup("jms/mytopic");
```

JNDI 查找不会创建队列（或主题），队列（或主题）已经由 j2eeadmin 命令创建。JNDI 查找只是在 Java 中创建一个 Queue 实例，它对消息传递系统中的队列结构建模，并提供对此队列的访问。

例：IBM WebSphere MQ

如果消息传递系统实现是 IBM 的 WebSphere MQ for Java（实现了 JMS），那么可以使

用 WebSphere MQ JMS 管理工具来创建目标。下面的代码将创建一个名为 myQueue 的队列。

```
DEFINE Q(myQueue)
```

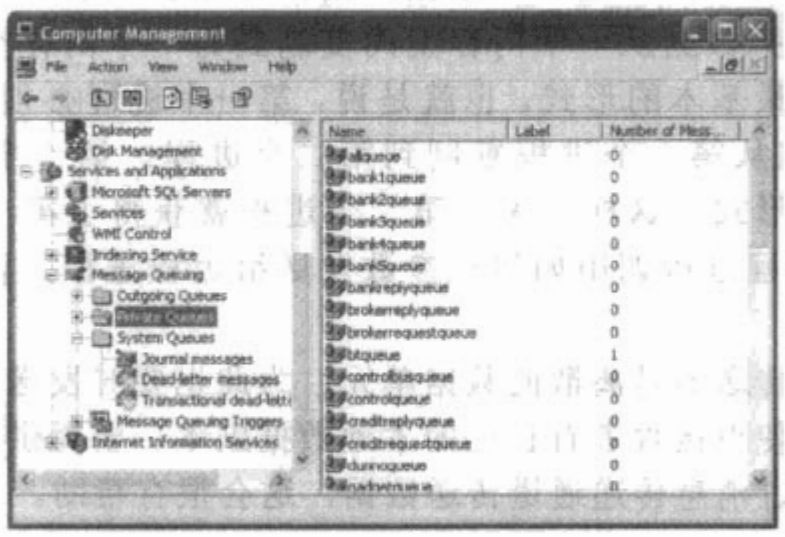
一旦 WebSphere MQ 中有了这个队列，应用就可以访问了。

如果不带完整的 WebSphere 应用服务器，WebSphere MQ 就不包含 JNDI 实现，因此不能像 J2EE 示例那样使用 JNDI 来查找队列，而必须通过一个 JMS 会话访问队列：

```
Session session = //创建会话
Queue queue = session.createQueue("myQueue");
```

例：Microsoft MSMQ

MSMQ 提供了许多不同的方式来创建消息通道，又称为队列。可以使用 Microsoft 消息队列浏览器（Message Queue Explorer）或计算机管理控制台（如下图所示）创建一个队列。在此可以设置队列属性或删除队列。



也可以使用下面的代码创建队列。

```
using System.Messaging;
...
MessageQueue.Create("MyQueue");
```

队列一旦创建，应用就可以通过创建一个 MessageQueue 实例来访问该队列，创建时必须传递队列名。

```
MessageQueue mq = p new MessageQueue("MyQueue");
```

3.3 消息



一个企业有两个不同应用通过消息传递进行通信，它们采用一个消息通道相互连接。

消息通道连接的两个应用如何交换一个信息？

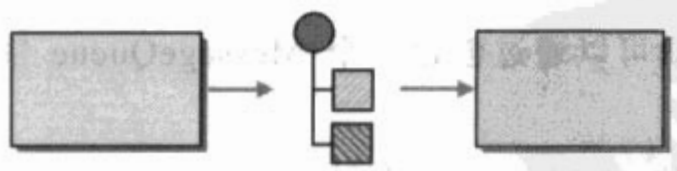
消息通道一般被看成是一个管道，它从一个应用连接到另一个应用。就像水一样，很自然地，可以把数据理解为从管道的一端流入，再从另一端流出。但是大多数应用的数据并不像水流那样连续不断，数据由单元组成，如记录、对象、数据库行，等等。因此通道必须传送数据单元。

“传送”数据的含义是什么？在一个函数调用中，调用者可以传递一个指向内存中数据地址的指针，从而按引用传递参数，这种做法是可行的，因为调用者和函数共享相同的内存堆。同样地，相同进程中的两个线程可以通过传递指针的方式传递记录或对象，因为它们都共享相同的内存空间。

要在两个不同的进程之间传递一些数据，则需要做更多的工作。因为它们都有自己的内存空间，因此必须把数据从一个内存空间复制到另一个内存空间。数据一般作为字节流传送，这是数据的最基本的形式。也就是说，第一个进程必须把数据编组（marshal）为字节形式，然后把它从第一个进程复制到第二个进程。第二个进程还要把数据解组（unmarshal）成原来的形式，这样一来，第二个进程就获得了第一个进程的原始数据副本。所谓编组，就是远程过程调用如何把参数发送给远程进程，以及远程进程如何返回结果。

同样地，消息传递传送的是离散的数据单元，为此也要对发送者的数据编组，并在接收方完成数据的解组，使得接收者有自己的本地数据副本。要提供一种简单的方法来包装数据单元，从而适于通过消息传递通道传送数据，这会很有帮助。

把信息封装到消息中，消息是一个数据记录，消息传递系统可以通过消息通道传送消息。



因此，任何想要通过消息传递系统传送的数据都必须转换成一个或多个消息，从而通过消息传递通道发送。

一个消息由两个基本部分组成。

1. **消息首部**——由消息传递系统使用的信息，用于描述所传送的信息、其信源和信宿（目标）等等。

2. **消息体**——被传送的数据，一般被消息传递系统忽略，只是原封不动地传送。

消息的概念不只是用在消息传递中，邮局服务邮件和电子邮件都会以离散的邮件消息发送数据。以太网按数据包发送数据，Internet 使用的 TCP/IP 协议簇中 IP 部分也传送数据包。Internet 上的流媒体实际上就是一系列数据包。

对消息传递系统来说，所有的消息都是一样的：要按照消息首部描述的信息传送消息体中的数据。但是对应用程序员来说，则有不同类型的消息，也就是说，不同的应用使用消息的方式也不同。使用命令消息可以调用另一个应用中的过程。使用文档消息可以把一组数据传递给另一个应用。使用事件消息可以向另一个应用通知本应用中发生的变化。如果其他应用必须返回一个应答，则使用请求/应答消息。

如果应用发送的信息在一个消息中容纳不下，可以把它划分为更小的单元，将各个部分作为一个消息序列（Message Sequence）发送。如果数据只是在一定的时间期限内有用，可以指定这样一个消息到期时间（Message Expiration）。由于消息的各种接收者和发送者必须对消息中的数据格式达成一致，可以把数据格式指定为规范数据类型（Canonical Data Model）。

例：JMS 消息

在 JMS 中，消息由 Message 类型表示，它有几个子类型。每种子类型的首部结构都是相同的，只是消息体格式随类型的不同而不同。

1. TextMessage——最常用的消息类型。消息体是一个字符串，比如文本或 XML 文档。textMessage.getText()将消息体返回为一个 String。

2. BytesMessage——最简单、最通用的消息类型。消息体是一个字节数组。bytesMessage.readBytes(byteArray)把消息内容复制到指定的字节数组中。

3. ObjectMessage——消息体是一个 Java 对象，它实现了 java.io.Serializable 接口，从而支持对象的编组和解组。objectMessage.getObject()返回 Serializable。

4. StreamMessage——消息体是一个 Java 基本类型数据流。接收者使用诸如 readBoolean()、readChar()和 readDouble()等方法从消息中读取数据。

5. MapMessage——消息体相当于 java.util.Map，其中主键是 String。接收者使用 getBoolean("isEnabled")和 getInt("numberOfItems")等方法从消息中读数据。

例：.NET 消息

在.NET 中，Message 类实现了消息类型。它有一个属性 Body，其中包含消息的内容，并作为一个 Object；BodyStream 把消息的内容存储为 Stream。另一个属性 BodyType 指定了消息体所包含数据的类型，如字符串、日期、货币、数字或其他对象。

例：SOAP 消息

在 SOAP 协议[SOAP1.1]中，SOAP 消息就是消息的一个例子。SOAP 消息是一个 XML 文档，该文档是一个信封（根 SOAP-ENV:Envelope 元素），包含一个可选的首部（SOAP-ENV:Header 元素）和一个必要的体（SOAP-ENV:Body 元素）。这个 XML 文档是一

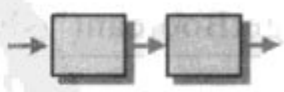
个可传送的原子数据记录（常用的传输协议为 HTTP），因此它是一个消息。

下面是一个 SOAP 消息的例子，这是从 SOAP 规范中摘取出来的，显示了一个包含首部和体的信封。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
      <t:Transaction
         xmlns:t="some-URI"
         SOAP-ENV:mustUnderstand="1">
            5
      </t:Transaction>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
         <symbol>DEF</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP 还展示出消息的递归性，因为 SOAP 消息可以由消息传递系统传送，这说明消息传递系统的消息对象（例如 JMS 中类型为 javax.jms.Message 的对象，或.NET 中类型为 System.Messaging.Message 的对象）包含 SOAP 消息（XML SOAP-ENV:Envelope 文档）。在这种情况下，采用的传输协议不再是 HTTP，而是消息传递系统内部的协议（它可能又会使用 HTTP 或其他网络协议传送数据，但是消息传递系统可以保证传输的可靠性）。对于通过一个不同的消息传递系统传送消息，要了解有关的更多内容，请见信封包装器（Envelope Wrapper）。

3.4 管道和过滤器



在许多企业集成案例中，一个事件会触发一系列的处理步骤，每个步骤都要完成特定的功能。例如，假设在我们的企业里新订单以消息的形式到达。第一个要求是消息必须加密，以防止窃听者获知客户的订单。第二个要求是消息中要包含认证信息，认证信息采用数字证书的形式，以确保订单是由可信的客户所下。而且，外部客户可能会发送重复的消息（还记得在许多热门购物网站上的提示吗？它们通常会警告只能把“现在订购”按钮点击一次）。为了避免重复发货和杜绝不受欢迎的客户，必须在后续的订单处理步骤前消除重复的消息。为了满足这些需求，必须对一系列可能重复的加密消息（其中包含额外的认证信息）进行转换，将其转换为一系列唯一的简单明文订单消息，而且不包含额外数据字段。

在保持独立性和灵活性的前提下，如何对消息完成复杂的处理？

一种可能的解决方案是编写一个完备的“输入消息消息传递模块”，由它完成所有必需的功能。但是，采用这种方法既不灵活，也难以测试。如果还需要增加新的步骤或删除某项步骤怎么办？例如，如果允许专用网络上的大客户下订单，而且这些订单并不需要加密，这该怎么办？

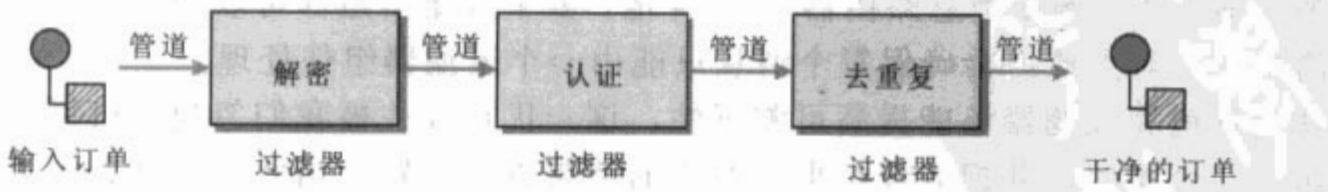
在一个单个组件中实现全部功能还会减少重用的机会。若能创建较小的良定义组件，这样就能在其他处理中重用这些组件。例如，订单状态消息只需要加密，但是不需要做去重复（de-duped）处理，因为重复的订单状态请求无关紧要。把解密功能封装到一个单独的模块中，我们就可以对其他消息重用这项功能。

集成解决方案通常要连接许多异构的系统。因此，不同的处理步骤需要在不同的物理计算机上执行，比如个别的处理步骤只能在特定的系统上执行。出于安全原因，用于解密输入消息的私有密钥可能只能在指定的计算机上获得，不能从其他任何计算机上获取。这说明，解密组件只能在这台指定的计算机上执行，而其他步骤可能在另外的计算机上执行。同样地，不同的处理步骤可能采用不同的编程语言或技术来实现，以至于无法在同一个进程中运行，甚至无法在同一台计算机上运行。

如果在不同的组件中实现各项功能，还会造成组件之间存在依赖关系。例如，如果解密组件要对解密得到的结果调用认证组件，那么解密功能与认证功能就存在依赖，只使用解密而不使用认证功能是办不到的。如果能把已有的组件以某种方式“组合”到一个处理步骤序列中，让每个组件都独立于系统中的其他组件，就能解决这种依赖问题。这就要求组件要提供通用的外部接口，以保证组件的可交换性。

如果使用异步消息传递，应该利用从一个组件向另一个组件发送消息的异步特性。例如，一个组件可以把消息发送给另一个组件进行进一步的处理，而不必等待处理的结果。使用这项技术，可以并行处理多个消息，每个组件处理一个消息。

使用管道和过滤器体系结构把较大的处理任务划分为一系列较小的独立处理步骤（过滤器），这些步骤由通道（管道）连接。



每个过滤器都提供一个非常简单的接口：它在入站管道（inbound pipe）接收消息，处理消息，然后把结果发布给出站管道（outbound pipe）。管道把前一个过滤器连接到下一个

过滤器，将前一个过滤器输出的消息发送给下一个过滤器。所有的组件都使用相同的外部接口，因此通过把组件连接到不同的管道，就可以组合成不同的解决方案。我们可以添加新的过滤器，删除已有的过滤器，或者重新安排它们的顺序，这些操作不会改变过滤器本身。过滤器与管道之间的连接有时被称为端口（port）。最基本的过滤器组件都有一个输入端口和一个输出端口。

在我们的例子中应用管道和过滤器时，采用的方案是由两个管道连接三个过滤器（如上图所示）。这里还需要增加一个管道把消息发送给解密组件，另外还要有一个管道把明文订单消息从去重复组件发送给订单管理系统。这样总共就需要四个管道。

管道和过滤器描述了一个基本的消息传递系统体系结构：各个处理步骤（过滤器）由消息传递通道（管道）链接起来。在本节和后面的小节中还介绍了一些模式，其中许多模式（如路由和转换模式）都是基于这种管道和过滤器体系结构。它能很方便地把各个模式组合成更大的解决方案。

管道和过滤器模式使用抽象管道使组件相互之间实现了解耦合。利用管道，一个组件能把消息发送到管道中，以后再由另一个过程来处理管道中的消息，发送消息的组件并不知道由谁来处理消息。显然这种管道可以用消息通道作为实现。消息通道可以在过滤器之间建立与语言、平台和位置无关的管道。这就能提供很大的灵活性，使我们能方便地转移到不同的计算机上完成某些处理步骤（之所以需要转移，可能是出于依赖性、维护性或性能方面的原因）。但是，如果所有组件实际上都驻留在相同的计算机上，那么由消息传递基础设施提供的消息通道就会变得过于庞大。如果使用一个简单的内存队列实现管道，效率会更高。因此，将组件设计为与一个抽象的管道接口进行通信很有好处。这样，以后此接口实现可以采用消息通道，也可以采用其他实现，如内存队列。消息传递网关（Messaging Gateway）描述了如何设计组件来提供这种灵活性。

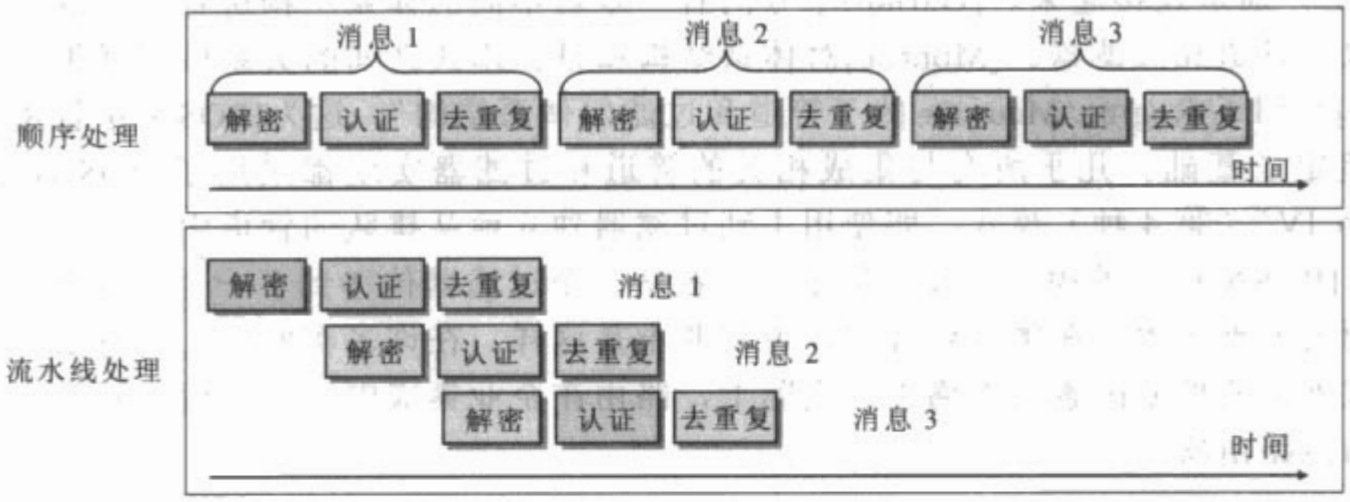
管道和过滤器结构的一个潜在缺点是它需要大量通道。首先，通道并不是一种无限的资源，因为通道提供了数据缓冲等功能，这些都会消耗内存和占用 CPU 周期。其次，把消息发布到通道存在一定的开销，因为数据必须从应用的内部格式转换为消息传递基础设施自己的格式。在接收端，还要执行相反的过程。如果使用的过滤器链很长，由于要重复进行消息数据的转换，就会降低系统的性能，这是为保持灵活性付出的代价。

单纯的管道和过滤器允许每个过滤器只有一个输入端口和一个输出端口。处理消息传递时，可以适当放松这一限制。一个组件可以处理来自多个通道的消息，也可以把消息输出到多个通道中（例如消息路由器）。同样地，多个过滤器组件可以处理来自同一个消息通道的消息。点对点通道会确保每个消息只能由一个过滤器组件处理。

使用管道和过滤器还能提高可测试性，这一优点往往被我们忽视。可以把一条测试消息发送给一个组件，并把处理结果与期望的结果进行比较，由此来测试各个处理步骤。独立地测试和调试每个核心功能会更加高效，因为我们可以根据特定的功能制定专门的测试方案。例如，为了测试组件的加密/解密功能，可以把大量包含随机数据的消息发送给组件。对消息完成加密和解密后，再与原始的消息进行比较。另一方面，如果要测试认证功能，则需要提供带有特定鉴别码的消息，这些鉴别码与系统中的已知用户对应。

3.4.1 流水线处理

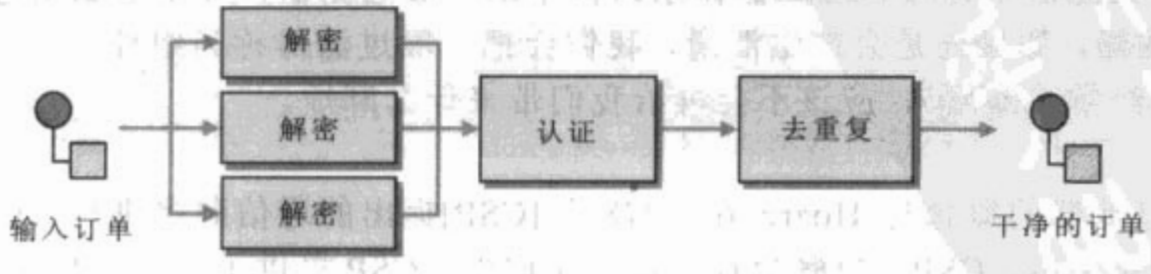
采用异步消息通道把组件连接起来，可以让其中的每个单元在自己的线程或进程中运行。当某个单元完成了一个消息的处理后，可以把消息发送给输出通道，并立即开始处理下一个消息。它不必等待后续组件读取和处理该消息。这使得多个消息在通过各个处理阶段时，能得到并行处理。例如，在第一个消息解密后，被传递给认证组件。同时，下一个消息已经得到解密（如下图所示）。我们把这种配置称为处理流水线（processing pipeline），因为流经过滤器的消息就像流经管道的水。与严格的顺序处理相比，处理流水线能极大地提高系统的吞吐量。



采用管道和过滤器实现流水线处理

3.4.2 并行处理

但是，整个系统的吞吐量要受到处理链中最慢过程的制约。可以为该过程部署多个并发的实例来提高吞吐量。在这种情况下，需要采用有竞争消费者（Competing Consumer）的点对点的通道，来确保通道中的每个消息只能被N个可用处理器中的某一个消费（处理）。这样可以加快时间最紧张的处理步骤，进而提高整体的吞吐量。但是需要注意的是，采用这样的配置可能会引起消息处理顺序不正常。如果消息的顺序非常关键，则只能为每个组件运行一个实例，或者必须使用重排器（Resequencer）。



采用并行处理增加吞吐量

例如，如果解密消息的速度比认证消息的速度慢，可以使用如上图所示的配置，运行解

密组件的三个并行实例。如果每个过滤器是无状态的——也就是说它能在消息处理完成后返回以前的状态，那么并行化过滤器就能很好地工作。这说明，要运行多个并行的去重复组件不太容易，因为去重复组件需要维护所收到消息的一个历史记录，所以它不是无状态的。

3.4.3 管道和过滤器的历史

管道和过滤器体系结构并不是一个新的概念。这种体系结构融合了灵活性和高吞吐量，因此管道和过滤器得到了广泛欢迎，这是不难理解的。由于它的语义很简单，还可以采用形式化的方法来描述这种体系结构。

[Kahn]在 1974 年把 Kahn 过程网络描述为一组并行过程，这些并行过程由无限制 FIFO（先入先出）通道连接起来。[Garlan]中专门有一章对不同的体系结构进行了很好的描述，其中也包括管道和过滤器。[Monroe]对体系结构和设计模式之间的关系做了详细的论述。[PLoPD1]介绍了 Regine Meunier 的“管道和过滤器体系结构”，这为[POSA]中管道和过滤器模式奠定了基础。几乎所有与集成相关的管道和过滤器实现都采用了[POSA] 提供的“Scenario IV”（第 4 种）模式，即使用主动过滤器独立地从排队的管道中拉出、处理和推入元素。[POSA]所述的模式假设，每个元素在从一个过滤器传递到下一个过滤器时，都要经过相同的处理步骤。在集成环境中，通常并不是这样。在许多实例中，消息要按照消息的内容或外部的控制因素动态路由。实际上，路由在企业集成中经常出现，它有自己的模式，如消息路由器。

术　语

当我们讨论管道和过滤器结构时，必须对过滤器（filter）这个术语多加留意。后面还要定义另外两个模式，消息过滤器（Message Filter）和内容过滤器（Content Filter）。这两种过滤器都是一般过滤器的特例，其他许多模式也是如此。换句话说，要作为管道和过滤器模式中的一个过滤器，不一定要具有过滤的功能（如删除字段或消息）。本来可以把管道和过滤器体系结构改个名字，以避免产生这种混淆，但是我们觉得管道和过滤器是一个得到如此广泛讨论的重要概念，如果给它取一个新的名字，反倒会带来更大的混淆。所以在介绍模式时，我们将尽可能谨慎地使用“过滤器”这个词，并尽量说明我们所指的过滤器是管道和过滤器体系结构中的一般过滤器，还是过滤消息的消息过滤器/内容过滤器。如果还是会产生混淆，我们会把一般过滤器称为组件，这个说法很有一般性（而且经常被滥用），应该不会再给我们带来什么麻烦。

管道和过滤器的概念与 Hoare 在 1978 年[CSP]提出的通信顺序进程（Communicating Sequential Processes，CSP）的概念有一定的相似性。CSP 提供了一种简单模型，可以描述并行处理系统中发生的同步问题。CSP 中使用的基本机制是通过输入/输出（I/O）实现两个进程的同步。当进程 A 指示它已经准备好向进程 B 输出，并且进程 B 指示它已经准备好从进程 A 输入时，I/O 发生。如果其中一个进程准备好，但是另一个还未准备好，准备

好的进程将被放入等待队列，直到另一进程准备好。CSP 与集成解决方案不同，因为 CSP 并不是松耦合的，而且“管道”也没有提供任何排队机制。不过，对 CSP 的广泛讨论对学术研究很有帮助。

例：C#和 MSMQ 中的简单过滤器

下面的代码段给出了一个过滤器的一般基类，它有一个输入端口和一个输出端口。基类实现只是把所收到消息的消息体打印出来，并把它发送给输出端口。要提供更有意义的过滤器，可以继承 Processor 类，并覆盖 ProcessMessage 方法，以对消息完成其他操作——也就是说，转换消息的内容或者把它路由到不同的输出通道。

请注意，在实例化 Processor 时，它要接收指向输入和输出通道的引用。因此，这个类不与任何指定通道或其他过滤器绑定。这样有利于我们实例化多个过滤器，而且可以采用任意的配置把它们链接在一起。

```
using System;
using System.Messaging;

namespace PipesAndFilters
{
    public class Processor
    {
        protected MessageQueue inputQueue;
        protected MessageQueue outputQueue;

        public Processor (MessageQueue inputQueue, MessageQueue outputQueue)
        {
            this.inputQueue = inputQueue;
            this.outputQueue = outputQueue;
        }

        public void Process()
        {
            inputQueue.ReceiveCompleted += new ReceiveCompletedEventHandler
(OnReceiveCompleted);
            inputQueue.BeginReceive();
        }

        private void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs
asyncResult)
        {
            MessageQueue mq = (MessageQueue)source;

            Message inputMessage = mq.EndReceive(asyncResult.AsyncResult);
```

```
            inputMessage.Formatter =  new XmlMessageFormatter
                                      (new String[] {"System.String,mscorlib"});

            Message outputMessage = ProcessMessage(inputMessage);

            outputQueue.Send(outputMessage);

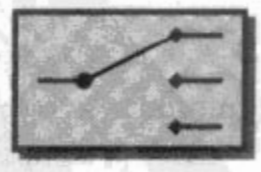
            mq.BeginReceive();
        }

        protected virtual Message ProcessMessage(Message m)
        {
            Console.WriteLine("Received Message: " + m.Body);
            return (m);
        }
    }
}
```

上述代码实现的是一个事件驱动的消费者。Process 方法注册到来的输入消息，并且每次有消息到达时会通知消息传递系统调用 OnReceive Completed 方法。这个方法从输入的事件对象中抽取出消息数据，并调用虚方法 Process Message。

这个简单的过滤器实例并不是面向事务的。如果在处理消息时出现了一个错误（在消息发送到输出通道前），消息就会丢失。在产品环境中这种情况是不允许发生的。请参考“事务性客户（Transactional Client）”寻找解决该问题的答案。

3.5 消息路由器



在管道和过滤器链中，多个处理步骤是通过消息通道连接起来的。

如何才能对各个处理步骤解耦合，使消息能按照一组条件传递到不同的过滤器？

管道和过滤器体系结构通过固定的管道把过滤器直接互连起来。这种做法之所以可行，是因为许多采用管道和过滤器模式的应用（如[POSA]）都基于一个很大的数据项集合，其中每个数据项都要经过相同的顺序处理步骤。例如，编译器总是先执行词法分析，然后是语法分析，最后是语义分析。而与此不同，基于消息的集成解决方案所处理的各个消息并不一定与一个很大的数据集关联。因此，每个消息更可能需要经过一系列不同的处理步骤。

消息通道实现了消息的发送者与接收者的解耦合。这也说明，多个应用可以把消息发布到一个消息通道中。因此，消息通道中可以包含来自不同数据源的消息，必须根据消息的类型或其他标准区别对待。可以为每种消息类型（后面介绍数据类型通道时将详细解释这个概念）分别创建一个单独的消息通道，并将每个通道与所需的处理步骤（处理相应的消息类型）连接。但是，这就要求消息的创建者了解不同处理步骤的选择标准，这样才能把消息发布给正确的通道。采用这种做法还会导致消息通道的个数激增。此外，消息要经过哪些处理步骤并不仅仅是由消息源决定的。例如，可以想像这样一种情况，随着经过通道传递的消息数的变化，消息的目标也会变化。没有哪个消息源能够知道到底传递了多少消息，因此无法把消息发送给正确的通道。

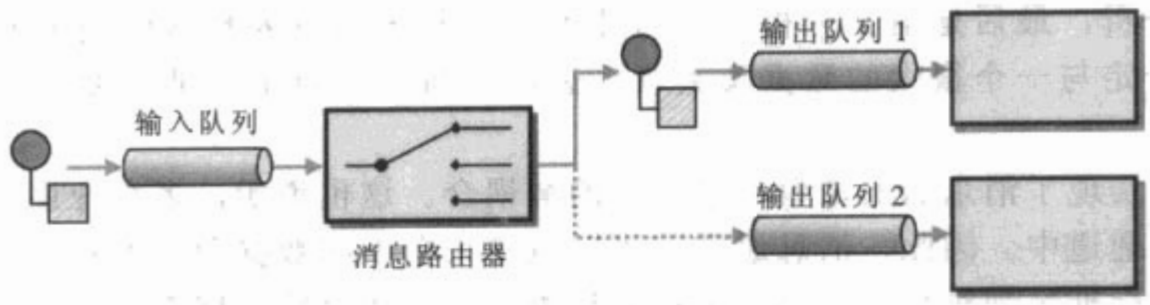
消息通道提供了一种非常基本的路由功能。应用向消息通道发布一个消息，而且对该消息的目标一无所知。因此，消息的传递路径会根据订购这个消息通道的组件而改变。但是，这种类型的“路由”并没有考虑每个消息的特性。一个组件只要订购了这个消息通道，默认地它就会消费这个通道中的所有消息，而不考虑每个消息的特有特性。这种行为类似于在 UNIX 系统中使用管道符号处理文本文件。它可以把处理步骤组合成一个管道和过滤器链，所有的文本行都会经过相同的处理。

可以让接收组件自己决定是否处理到达一个公共消息通道的消息。但是这种做法是有问题的，因为消息一旦被消费，而组件又决定不对其进行处理，它将无法把消息重新放回通道让其他组件去处理。有的消息传递系统允许接收者先检查消息的特性（而不要把消息从通道中真正删除），从而决定是否消费该消息。但是这并不是一种一般性的解决方法，因为选择消息的逻辑建在组件的内部，这会把消费组件与特定的消息类型关联起来，从而降低了重用组件的可能性，并且会丧失管道和过滤器模型的重要特点——可组合性（composability）。

对此有许多替代方法，这些方法假设我们可以修改参与的组件，以满足消息选择的需要。但是在大多数集成解决方案中，构建模块（组件）都是大型应用，它们一般无法修改，例如封装好的应用或遗留的应用。所以要调整产生或消费消息的应用来满足消息传递系统或其他应用的需要，这是不经济的，甚至是不可能的。

管道和过滤器的一项优点是单个组件的可组合性。利用这一特性，我们可以在过滤器链中插入额外的步骤，而不必修改现有的组件。通过在两个过滤器之间插入另一个过滤器，让它来决定下一步要执行的步骤，这就为解耦合提供了机会。

插入一个特殊的过滤器，即消息路由器，它从一个消息通道中消费一个消息，然后根据一组条件把它重新发布到不同的消息通道中。



消息路由器与管道和过滤器的基本概念不同，因为它要连接多个输出通道（也就是说，它有多个输出端口）。但是，基于管道和过滤器体系结构的支持，消息路由器周围的组件完全感受不到消息路由器的存在。它们只是从一个通道中消费消息，然后把它们发布给另一个通道。消息路由器的一个固有特性是，它不会修改消息的内容，而只关心消息的目标。

使用消息路由器的主要好处在于，决定消息目标的标准都在一个位置上维护。如果定义了新的消息类型，增加了新的消息处理组件，或者修改了路由规则，我们只需要修改消息路由器的逻辑，而不会影响其他任何组件。而且，由于所有的消息都会经过唯一的消息路由器，就能保证输入的消息能够按正确的顺序逐个地处理。

尽管消息路由器的目的是为了使组件解耦合，但是使用它却会带来相反的效果。为了把消息发送到正确的通道，消息路由器组件必须了解所有可能的目标通道。如果可能的目标通道列表经常改变，消息路由器就会成为维护的瓶颈。在这种情况下，让每个接收者自己选择感兴趣的消息可能更合适。可以使用一个发布-订购通道和一个消息过滤器数组实现上述目的。我们把这两种选择方案分别称为预测型（predictive）路由和反应型（reactive）路由（更详细的对比分析请参见第7章的“消息路由”中的消息过滤器）。

由于消息路由器需要插入额外的处理步骤，这会降低系统的性能。许多基于消息的系统在把消息从一个通道放入另一个通道前，必须对消息进行解码操作，如果消息本身并未真正改变，这种做法会带来计算上的负担，从而使消息路由器成为性能的瓶颈。通过并行地使用多个路由器或增加额外的硬件，可以尽量减少这一影响。因此，消息吞吐量（单位时间内被处理的消息数）可能不会受到影响，但是消息的延迟（一个消息经过整个系统所需的时间）总会增加。

和大多数好的工具一样，消息路由器也会被滥用。随意地使用消息路由器会把松耦合的优点变成缺点。对于松耦合系统，很难获得解决方案的“整体映像”——即经过整个系统的消息流。这是消息传递解决方案遇到的共同问题，而使用路由器更会让这个问题加剧。如果系统中任何组件都与其他组件松耦合，要想掌握消息流动的真正方向是不可能的。这就会增加测试、调试和维护的难度。有许多工具可以帮助缓解这个问题。可以使用消息历史在运行时检查消息，查看它们经过了哪些组件。或者，可以建立一个列表，其中包含系

统中各个组件订购或发布消息的所有通道。根据这些知识就可以绘制出经过组件的所有可能的消息流图。许多 EAI 包会在一个集中的知识库中维护通道的订购信息，使得这种静态分析更加容易。

不同的消息路由器

消息路由器可以根据多种标准来确定输入消息的输出通道。最简单的情况是固定路由器（Fixed Router）。在这种路由器中，只定义了一个输入通道和一个输出通道。固定路由器消费输入通道中的一个消息，然后把它发布到输出通道。为什么还要使用这种愚笨的路由器呢？固定路由器特别适合对子系统解耦合，以便以后插入更智能的路由器。或者，也可以用来在多个集成解决方案之间实现消息中继。在大多数情况下，固定路由器会与消息转换器或通道适配器组合，从而转换消息的内容或把消息发送给不同类型的通道。

许多消息路由器只是根据消息本身的特性来决定消息的目标。例如，根据消息的类型或指定消息字段的值。我们把这种路由器称为基于内容的路由器（Content-Based Router）。这种路由器很常用，在基于内容的路由器模式中将详细介绍。

其他消息路由器按照环境因素决定消息的目标。我们把这些路由器称为基于上下文的路由器（Context-Based Router）。这种路由器一般用于实现负载平衡、测试或失败恢复等功能。例如，如果一个处理组件出现了故障，基于上下文的路由器会把消息路由到其他处理组件，以提供失败恢复能力。还有一些路由器把消息流平均分配到多个通道上，可以实现像负载平衡器那样的并行处理。有的消息通道已经提供了基本的负载平衡能力，而不必使用消息路由器，因为多个竞争消费者可以从同一个通道上尽可能快地消费消息。但是，消息路由器还内置有许多额外的路由智能，而不同于通道实现的简单轮转调度。

许多消息路由器都是无状态的——换句话说，它们一次只针对一个消息做出路由决策。另外一些路由器在做路由决策时会考虑以前消息的内容。例如，在管道和过滤器示例中，我们使用的路由器会保留所有已接收消息的列表，以消除重复的消息。这些路由器是有状态的。

大多数消息路由器都包含一些硬编码逻辑来实现路由决策。但是，有些路由器与控制总线连接，因此可以利用中间件解决方案修改路由标准，而不必修改任何代码，也不用中断消息流。例如，控制总线可以把某个全局变量的值传播给系统中的所有消息路由器。这对测试非常有用，从而使消息传递系统能够方便地从测试模式切换到产品模式。动态路由器（Dynamic Router）可以根据来自各个潜在接收者的控制消息动态配置其路由标准。

第 7 章“消息路由”将介绍更多不同的消息路由器。

例： 商业 EAI 工具

消息路由器是消息代理（Message Broker）的核心概念，在所有商业 EAI 工具中，几乎都实现了消息路由器。这些工具接收输入消息，完成验证和转换，然后把消息路由到正确的目标。基于这种体系结构，使得参与的应用不必了解其他合作的应用，因为消息代理在这些应用之间建立了代理。这在企业集成中是一项关键功能，因为大多数要连接的应用都是封装好或遗留的应用，集成不能产生干扰——也就是说，不能对应用的代码做任何修

改。因此，中间件必须把所有路由逻辑集中起来，而不要让应用实现路由逻辑。消息代理在集成中等价于[GoF]中提出的中间人（Mediator）。

例：采用 C#和 MSMQ 实现的简单路由器

下面的代码示例演示了一个非常简单的路由器，它根据一个简单的条件，把输入消息路由到两个可能的输出通道之一。

```
class SimpleRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue1;
    protected MessageQueue outQueue2;

    public SimpleRouter(MessageQueue inQueue, MessageQueue outQueue1, MessageQueue outQueue2)
    {
        this.inQueue = inQueue;
        this.outQueue1 = outQueue1;
        this.outQueue2 = outQueue2;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }
    private void OnMessage(Object source,
ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        if (IsConditionFulfilled())
            outQueue1.Send(message);
        else
            outQueue2.Send(message);

        mq.BeginReceive();
    }

    protected bool toggle = false;

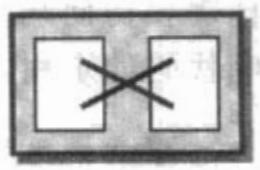
    protected bool IsConditionFulfilled ()
    {
        toggle = !toggle;
```

```
        return toggle;
    }

}
```

这些代码相当明了。与管道和过滤器中提出的简单过滤器一样，SimpleRouter 类采用 C#委托机制实现了消息的一个事件驱动消费者。构造函数把 OnMessage 方法注册为处理方法，来处理到达 inQueue 的消息。这样对于每个到达 inQueue 的消息，.NET 框架都要负责为该消息调用 OnMessage 方法。OnMessage 方法通过调用 IsConditionFulfilled 方法来得出要把消息路由到哪里。在这个简单的例子中，IsConditionFulfilled 只是在两个通道之间随机选择，把消息流平均地分配到 outQueue1 和 outQueue2 中。为了尽量减少代码，这个简单的路由器不是事务性的——也就是说，如果路由器消费输入通道中的一个消息，但把它发布到输出通道前出现了故障，那么消息将丢失。“事务性客户”将解释如何实现事务性端点。

3.6 消息转换器



前面的模式展示了如何构造消息，以及如何把消息路由到正确的目标。在很多情况下，企业集成解决方案要在已有的应用之间实现消息的路由，如遗留系统、封装好的应用、自定制的应用或由外部合作伙伴操控的应用。这些应用中，通常每一个应用都基于一种专用的数据模型而建立。每个应用对于客户实体的概念、定义客户的属性以及与客户相关的其他实体等等都有不同的理解。例如，财会系统对客户的纳税人 ID 号更感兴趣，而客户关系管理（Customer-Relationship Management，CRM）系统则会存储电话号码和地址。应用的底层数据模型往往决定了物理数据库模式的设计，决定了如何设计接口文件格式或应用编程接口（API）——集成方案必须利用这些接口与实体交互。因此，每个应用通常都希望接收到的消息能遵循其内部数据格式。

除了各种应用内部的专用数据模型和数据格式外，集成解决方案还要通过标准的数据格式与外部商业伙伴交互，这些标准数据格式要独立于任何特定应用。许多协会和标准组织已经定义了一些协议，如 RosettaNet、ebXML、OAGIS 和其他许多行业相关的标准。在很多情况下，集成解决方案必须通过“官方”的数据格式与外部合作伙伴通信，尽管其内部系统采用的是专用的数据格式。

如何使用消息传递来使得采用不同数据格式的系统之间实现通信？

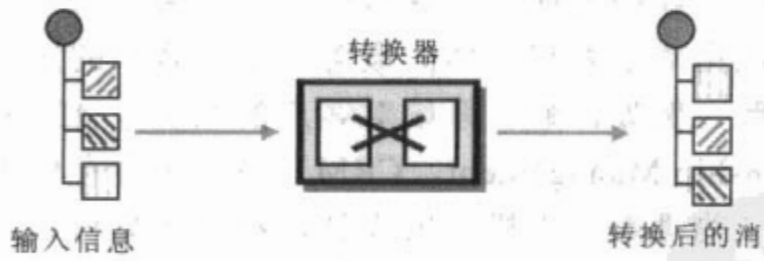
如果能修改所有应用让它们使用共同的数据格式，就能避免消息转换。但是出于很多原因，这种做法并不可行（见“共享数据库”）。首先，修改应用的数据格式是危险的，也很困难，需要对固有的业务功能做大量的改动。对大多数遗留应用来说，修改数据格式不只是经济上是否可行的问题。我们都还记得为解决 Y2K（千年虫）所做出的努力，尽管要修改的范围只不过是一个字段！

其次，尽管可以让多个应用使用相同的数据字段名，甚至是相同的数据类型，但是它们的物理表示仍可能大不相同。有可能一个应用使用的是 XML 文档，而其他应用使用 COBOL 文稿。

此外，如果我们把一个应用的数据格式调整为另一个应用的数据格式，就会把这两个应用结合得更紧密。企业集成有一项关键的设计原则，就是要实现应用之间的松耦合（见“规范的数据模型”）。如果修改某个应用，让它与另一个应用的数据格式匹配，就违背了上述原则，因为这样一来，这两个应用直接依赖于彼此的内部表示。所以，要想替换或修改其中的一个应用而不影响另一个应用，这是不可能的，但是这种情况在企业集成中相当普遍。

可以把数据格式的转换直接放在消息端点（Message Endpoint）中实现。采用这种方式，所有应用都采用公共的格式发布和消费消息，而不是采用应用内部的数据格式。但是，这样需要访问端点代码，对封装好的应用不可行。而且，把格式转换硬编码写到端点中会影响代码的重用。

在其他过滤器或应用之间使用一种特殊的过滤器，即消息转换器，把一种数据格式转换为另一种数据格式。



消息转换器在消息传递方面等价于[GoF]中描述的适配器模式。适配器把组件的接口转换为另一种接口，使之能在不同的上下文中使用。

3.6.1 转换的层次

可能要在许多不同的层次上完成消息转换。例如，数据元素可能采用了相同的名称和数据类型，但是使用的是不同的表示（如 XML 文件方式，或者用逗号分隔的数值，也可能是固定长度的字段）。或者，尽管所有数据元素均采用 XML 格式表示，但是使用了不同的标记名。要总结各种不同类型的转换，可以划分为多个不同层（借鉴 OSI 参考模型）。

层	处理对象	转换需求（举例）	工具/技术
数据结构（应用层）	实体、联合、维度	把多对多关系压缩为聚集	结构化映射模式、自定义的代码
数据类型	字段名、数据类型、值域、约束、码值	把 ZIP 码从数字转换为字符串 把名和姓字段组合为一个名字字段 把美国的州名用两个字符的代码表示	EAI 的可视化转换编辑器、XSL、数据库查找、自定义的代码
数据表示	数据格式（XML、名-值对、定长数据字段、EAI 开发商特定格式等） 字符集（ASCII、Unicode、EBCDIC） 加密/压缩	解析数据表示，并以另一种格式显示 根据需要解密/加密	XML 解析器、EAI 解析器/显示工具、自定义的 API
传输	通信协议：TCP/IP 套接字、HTTP、SOAP、JMS、TIBCO RendezVous	跨协议移动数据，而不影响消息的内容	通道适配器、EAI 适配器

传输层位于这个“栈”的最下面，提供不同系统之间的数据传输服务。它负责跨不同网段完整、可靠地传输数据，并处理数据包丢失和其他网络错误。有些 EAI 开发商提供了自己的传输协议（如 TIBCO RendezVous），另一方面，另外一些集成技术则充分利用了 TCP/IP 协议（如 SOAP）。不同传输层之间的转换是由通道适配器模式提供的。

数据表示层也被称为语法层（syntax layer）。这一层定义了所传送数据的表示。由于传输层只负责传输字符或字节流，因此转换是必要的。也就是说，复杂的数据结构必须转换为一个字符串。经常需要转换为以下格式，包括 XML、定长字段（如 EDI 记录）和专用格式。在很多情况下，数据还要压缩或加密，并附加校验位或数字证书。为了与采用不同数据表示的系统打交道，数据可能必须解密和解压，并得到解析，然后提供（显示）新的数据格式，可能还要压缩和加密。

数据类型层定义了应用的数据类型，它是应用（域）模型的基础。这一层要做出一些决定，如把日期字段表示为字符串还是本地日期结构，日期是否要带一个日期时间组件，以及日期基于哪个时区等等。另外可能还要考虑邮政编码字段是否只表示美国的邮编，或是否要包含加拿大的邮编。如果只表示美国邮编，是否要包含 ZIP+4？它是不是必需的？把它存储为一个字段，还是存储在两个字段中？类似这样的许多问题一般都是由所谓的数据字典（Data Dictionary）提供答案。与数据类型相关的问题会有很多，并不只是一个字段的类型是字符串还是整型这样的简单问题。以按区域组织的销售数据为例，某个部门使用的应用可能把国家划分为四个区域：西部、中部、南部和东部，它们分别由 W、C、S 和 E 这四个字符区分。另一个部门可能会区分太平洋区域和山地区域，并划分东北与东南区域，每个区域都采用两位数字表示。这么一来，看到字符 E，你能知道它对应的数字是什么吗？

数据结构层描述了位于应用领域模型之上的数据，因此也被称为应用层。这一层定义了应用要处理的逻辑实体，如客户、地址或账户。它还定义了这些实体之间的关系：一个

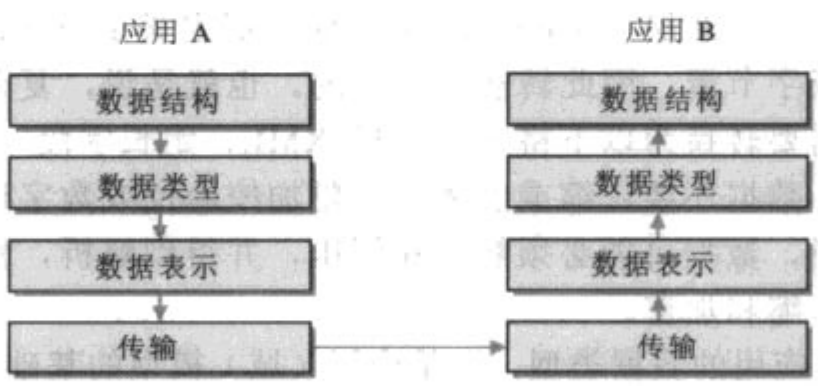
客户能否有多个账户？一个客户能否有多个地址？客户能共享一个地址吗？多个客户能共享一个账户吗？地址是账户的组成部分，还是客户的组成部分？这属于实体关系图和类图的范畴。

3.6.2 解耦合的层次

许多集成设计方面的折衷考虑是由组件或应用的解耦合需求决定的。解耦合是实现变更管理的基本工具。集成要把已有的应用连接起来，同时还要适应这些应用的变化。消息通道实现了应用的解耦合，使它们不必知道彼此的位置。消息路由器甚至不要求在应用之间协商建立共同的消息通道。但是，如果应用还要依赖对方的数据格式，那么采用这种形式的解耦合只能获得有限的独立性。消息转换器可以消除这种额外的关联。

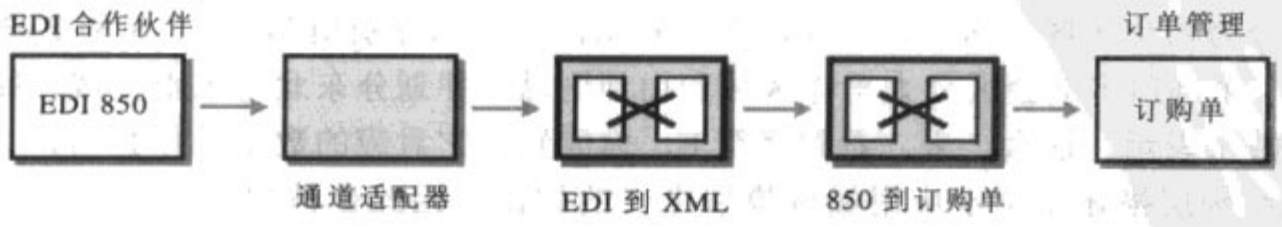
3.6.3 链接的转换

许多业务过程需要在多个层次上进行转换。例如，假设一个 EDI 850 订购单记录表示为固定格式的文件，它必须转换为 XML 文档，并通过 HTTP 传送给订单管理系统，该系统使用了另外一种订单对象的定义。这里在四个层次上都需要转换：传输形式从文件传输转换为 HTTP，数据格式从固定字段格式转换为 XML，数据类型和数据格式必须转换为与订单管理系统定义的订单对象一致。分层模型的优点是，你可以专心处理某一层而不必考虑更低层是如何实现的，这样就能一次集中在一个抽象层次上（如下图所示）。



跨越多个层的映射

把多个消息转换器单元通过管道和过滤器链接起来，可以形成如下图所示的体系结构。为每个层创建一个消息转换器，从而能在其他环境中重用这些组件。例如，通道适配器和“EDI 到 XML”消息转换器可以采用通用的方式实现，这样它们就可以在任何输入 EDI 文档上重用了。



链接多个消息转换器

把多个消息转换器链接起来，即使修改了其中某层的转换方式，也不会影响其他的任何层。可以使用相同的结构转换机制，不过并不是把数据表示转换为固定格式，而是可以通过切换数据表示转换，将其转换为一种用逗号分隔的文件格式。

消息转换器模式有许多不同的特例和变种。信封包装器（Envelope Wrapper）把消息数据包装在一个信封中，使它能通过消息传递系统传输。内容扩充器（Content Enricher）在消息中增加信息，而内容过滤器（Content Filter）则从消息中去除信息。声明标签（Claim Check）把信息从消息中去除，但同时会把所去除的信息保存起来，以备以后获取。规范器（Normalizer）把多种不同的消息格式转换为统一的格式。最后，规范数据模型展示出如何平衡多个消息转换器，从而实现数据格式的解耦合。在这些模式的内部，可能会出现复杂的结构转换（如把多对多的关系映射为一对一的关系）。

例：采用 XSL 实现结构转换

由于转换是一种共同的需要，W3C 定义了一个标准的 XML 文档转换语言：可扩展样式表语言（Extensible Stylesheet Language，XSL）。XSL 转换（XSL Transformation，XSLT）语言是 XSL 的一部分，它是一种基于规则的语言，能把 XML 文档转换为其他格式。由于这是一本介绍集成而非专门谈 XSLT 的书，所以对此我们只是举一个简单的例子（详细的内容请参考[XSLT 1.0]规范，或者阅读[Tennison]提供的代码实例）。为简单起见，在解释所需的转换时，我们给出的是 XML 文档的例子，而不是 XML 模式。

例如，假设要把输入的 XML 文档传递给财会系统。如果两个系统都使用 XML，那么数据表示层就是相同的，需要考虑到字段名、数据类型和结构中的所有差异。首先假设输入的文档如下所示。

```
<data>
    <customer>
        <firstname>Joe</firstname>
        <lastname>Doe</lastname>
        <address type="primary">
            <ref id="55355"/>
        </address>
        <address type="secondary">
            <ref id="77889"/>
        </address>
    </customer>
    <address id="55355">
        <street>123 Main</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94123</postalcode>
        <country>USA</country>
        <phone type="cell">
            <area>415</area>
            <prefix>555</prefix>
            <number>1234</number>
```

```
        </phone>
        <phone type="home">
            <area>415</area>
            <prefix>555</prefix>
            <number>5678</number>
        </phone>
    </address>
    <address id="77889">
        <company>ThoughtWorks</company>
        <street>410 Townsend</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94107</postalcode>
        <country>USA</country>
    </address>
</data>
```

这个 XML 文档包含客户数据。每个客户可以关联多个地址，每个地址可以包含多个电话号码。这个 XML 文档把地址表示为独立的实体，这样一来，多个客户就可共享同一个地址。

假设财会系统采用了如下表示。(如果你觉得采用德国标记名有些别扭，要记住最受欢迎的一个企业软件 SAP 就是因其德国字段名才声名鹊起的!)

```
<Kunde>
    <Name>Joe Doe</Name>
    <Adresse>
        <Strasse>123 Main</Strasse>
        <Ort>San Francisco</Ort>
        <Telefon>415-555-1234</Telefon>
    </Adresse>
</Kunde>
```

结果文档的结构更简单。标记名是不同的，有些字段被合并成一个字段。由于结果文档只提供了一个地址和一个电话号码的空间，我们需要根据业务规则从原始文档中选择一个地址和一个电话号码。下面的 XSLT 程序会把原始的文档转换成所需的格式。它要匹配输入文档的元素，并把它们转换为所需的文档格式。

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999
/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:key name="addrlookup" match="/data/address" use="@id"/>
    <xsl:template match="data">
        <xsl:apply-templates select="customer"/>
    </xsl:template>
    <xsl:template match="customer">
```

```
    <Kunde>
        <Name>
            <xsl:value-of select="concat(firstname, ' ', lastname)"/>
        </Name>
        <Adresse>
            <xsl:variable name="id" select="./address[@type='primary']/ref/@id"/>
            <xsl:call-template name="getaddr">
                <xsl:with-param name="addr" select="key('addrlookup', $id)"/>
            </xsl:call-template>
        </Adresse>
    </Kunde>
</xsl:template>
<xsl:template name="getaddr">
    <xsl:param name="addr"/>
    <Strasse>
        <xsl:value-of select="$addr/street"/>
    </Strasse>
    <Ort>
        <xsl:value-of select="$addr/city"/>
    </Ort>
    <Telefon>
        <xsl:choose>
            <xsl:when test="$addr/phone[@type='cell']">
                <xsl:apply-templates select="$addr/phone[@type='cell']" mode="getphone"/>
            </xsl:when>

            <xsl:otherwise>
                <xsl:apply-templates select="$addr/phone[@type='home']" mode="getphone"/>
            </xsl:otherwise>
        </xsl:choose>
    </Telefon>
</xsl:template>
<xsl:template match="phone" mode="getphone">
    <xsl:value-of select="concat(area, '-', prefix, '-', number)"/>
</xsl:template>
<xsl:template match="*"/>
</xsl:stylesheet>
```

XSL 是基于模式匹配的，如果你和大多数人一样习惯于面向过程的程序设计，要读懂它可能有些困难。简单地说，只要输入的 XML 文档中的某个元素与 match 属性中指定的

表达式匹配，就会调用<xsl:template>元素内部的指令。例如，如果有下面的代码行：

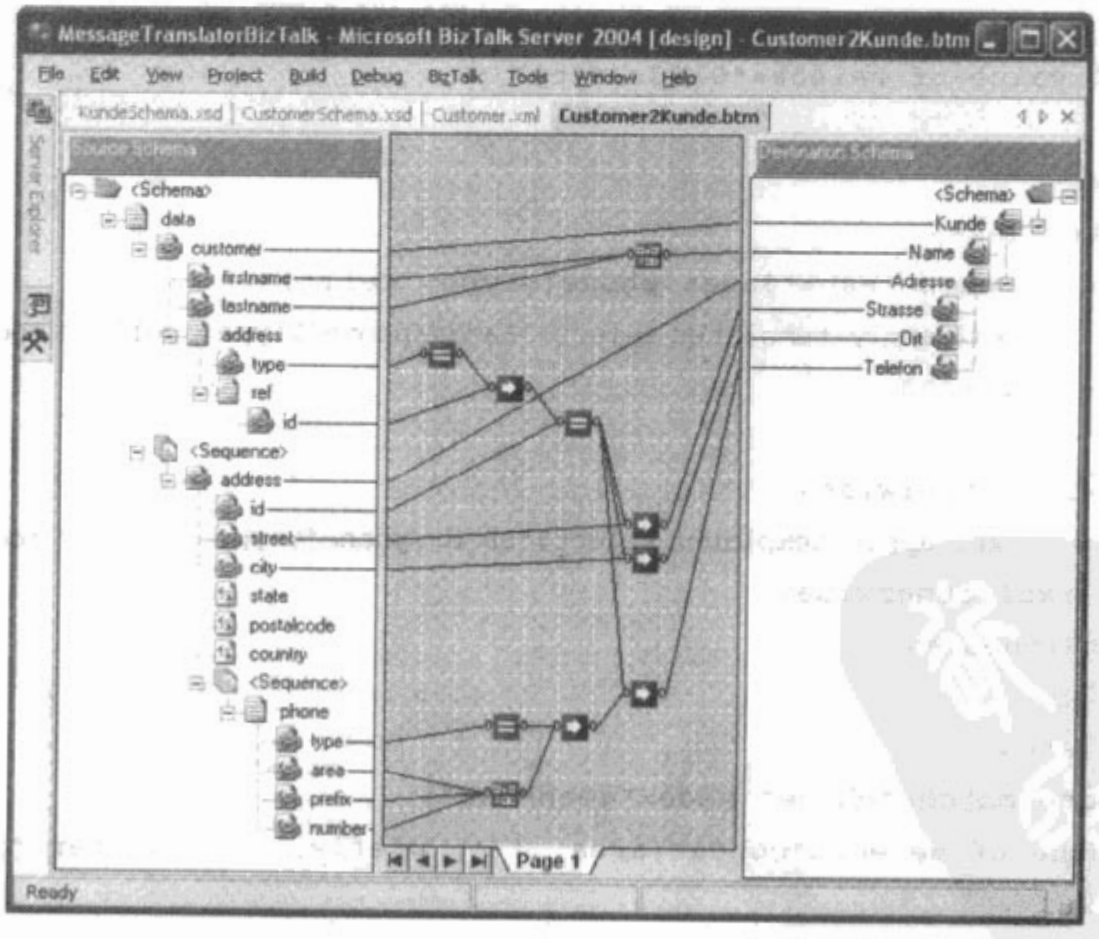
```
<xsl:template match="customer">
```

只要源文档中存在<customer>元素，就会执行相应的代码行。下一条语句把姓和名组合起来，并把结果保存到<Name>元素中。获取地址需要点技巧。XSL 代码查找<address>元素的正确实例，并调用子过程 getaddr。getaddr 从原始的<address>元素中取出地址和电话号码。如果存在移动电话号码，则使用这个移动电话号码，否则使用家庭电话号码。

例：可视化的转换工具

如果你发现 XSL 程序设计有一定的难度，请不必担心。大多数集成开发商提供了可视化的转换编辑器，能够分别在屏幕的左边和右边显示出两个文档格式的结构。用户只要在格式之间画上连线，就能把元素关联起来。显然，这要比编写 XSL 代码简单得多。有些开发商，如 Contivo，就专门开发这种转换工具。

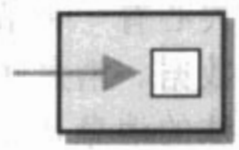
下图给出的是 Microsoft BizTalk Mapper 编辑器，它被集成在 Visual Studio 中。图中显示了元素之间的映射，这要比 XSL 脚本清楚得多。另一方面，有些细节（例如如何选择地址）则隐藏在所谓的 functoid 图标下。



采用拖放方式创建转换

采用拖放方式创建转换极大缩短了开发消息转换器的学习曲线。但是，当开始调试时，或者需要创建复杂的解决方案时，可视化工具也会成为一种负担。因此，许多工具支持在 XSL 和可视化表示之间的来回切换。

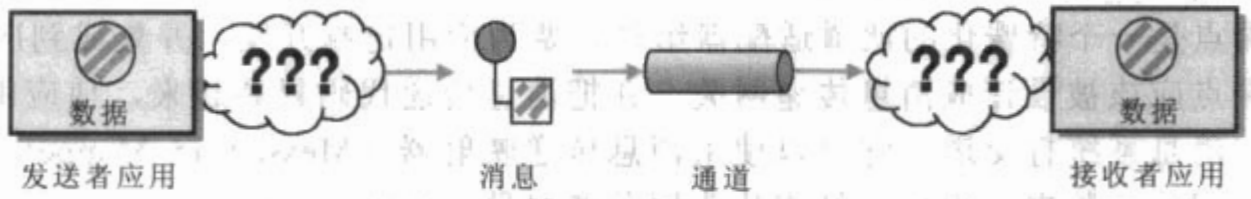
3.7 消息端点



应用通过经由消息通道发送消息来实现相互通信。

应用如何连接到消息传递通道以发送和接收消息？

应用和消息传递系统是两组单独的软件。应用为某些类型的用户提供功能，而消息传递系统会管理消息传递通道，从而传送消息来实现通信。即使集成了消息传递系统，使之成为应用的一个基本组成部分，它仍然是一个单独而且特殊的功能提供者，更像一个数据库管理系统或一个 Web 服务器。由于应用和消息传递系统是分开的，因此必须采用一种方法把它们连接起来并一起运行。

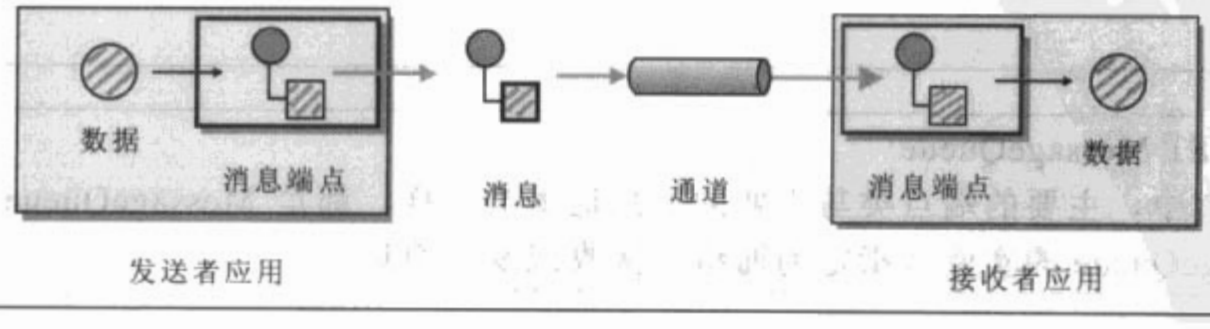


应用与消息通道是分离的

消息传递系统是一种服务器，它能接收请求并返回应答。就像数据库接受和返回数据一样，消息传递服务器会接受和传递消息。一个消息传递系统就是一个消息传递服务器。

服务器需要客户，使用消息传递的应用就是消息传递服务器的客户。但是应用没有必要知道如何成为消息传递的客户，就像它们没有必要了解如何成为数据库客户一样。消息传递服务器类似于数据库服务器，为应用提供了与服务器交互的 API。这些 API 不是特定于应用的，而是特定于领域的，这里所说的领域就是消息传递。应用必须包含一组代码，这组代码把消息传递领域与应用连接和组合起来，从而实现消息传递。

使用消息端点（Message Endpoint）把应用与消息传递通道连接起来，它是消息传递系统的客户，应用可以使用它发送或接收消息。



无论是应用还是消息传递系统的客户 API，消息端点代码都是定制的。应用的其余部分对消息格式、消息传递通道或采用消息传递与其他应用进行通信的细节知之甚少。它只关心有一个请求或一些数据需要发送给另一个应用，或者希望从其他应用那里获得请求或数据。消息传递端点代码负责接收这些命令或数据，把它们加工成消息，并发送到一个特殊的消息传递通道上。消息传递端点还负责接收消息，抽取出消息的内容，然后把数据以应用能理解的方式发送给应用。

消息端点把消息传递系统封装起来，而不让应用的其余部分了解，它为特定的应用和任务定制了通用的消息传递 API。如果使用某种特定消息传递 API 的应用需要切换为使用另一种 API，开发人员就必须改写消息端点的代码，但是应用的其余部分可保持不变。如果新版本的消息传递系统修改了消息传递 API，则只会影响消息端点代码。如果应用决定通过其他方法，而不是通过消息传递与其他应用通信，开发人员同样只需改写消息端点代码，而不必修改应用的其余部分。

消息端点可以用于发送或接收消息，但是同一个实例不能同时做这两项操作。端点是通道相关的，因此一个应用会使用多个端点与多个通道连接。应用可以使用多个端点实例与一个通道建立连接，以便支持多个并发的线程。

消息端点是一个特殊化的通道适配器组件，要为应用定制开发，并集成到应用中。

消息端点应该被设计成消息传递网关，并把消息传递代码封装起来，使应用的其余部分不需要与消息系统打交道。它可以使用消息传递映射器（Messaging Mapper）在领域对象与消息之间传递数据。也可以结构化为服务激励器（Service Activator），从而能够对同步服务或函数调用实现异步的消息存取。端点作为事务性客户，可以显式地控制与消息传递系统有关的事务处理。

发送消息是非常简单的，许多端点模式关心的只是接收消息的不同方法。消息接收者既可以是轮询消费者，也可以是事件驱动的消费者。多个消费者可作为竞争消费者（Competing Consumer），或者通过消息分派器（Message Dispatcher）从相同的通道接收消息。接收者可以使用选择性消费者（Selective Consumer）来决定应该消费或忽略哪些消息，或者使用持久订购者（Durable Subscriber）来确保订购者不会在端点断开时错过发布的消息。同时消费者可以是一个幂等接收者（Idempotent Receiver），它能正确地检测并处理重复的消息。

例：JMS 生产者和消费者

在 JMS 中，两种主要的端点类型是：用于发送消息的 MessageProducer，以及用于接收消息的 MessageConsumer。消息端点使用其中一种类型的实例从指定的通道中接收或发送消息。

例：.NET MessageQueue

在.NET 中，主要的端点类与主要的消息通道类一样，都是 MessageQueue。消息端点使用 MessageQueue 的实例从指定的通道中接收或发送消息。

第4章

消息传递通道

4.1 引言

在第3章“消息传递系统”中，我们介绍了消息通道。两个应用需要交换数据时，可以通过连接彼此的通道发送数据。应用在发送数据时，不一定要知道接收数据的是哪个应用。但是，通过选择特定的通道来发送数据，发送者知道接收者肯定是正在该通道上监听并正在查找此类数据的应用。采用这种方式，产生共享数据的应用就能与那些想要消费此类数据的应用进行通信。

4.1.1 消息通道主题

决定使用通道并不难，如果应用有需要传输的数据，或者想要接收数据，那么它就必须使用通道。真正的难题是确定应用使用什么通道，以及使用通道做什么。

固定的通道集——本章要讨论的一个话题是，对一个应用来说，可用的消息通道集是静态的。在设计应用时，开发人员必须知道把哪种类型的数据放在哪里，这样才能与其他应用共享这些数据；另外还要知道在哪里查找来自其他应用的特定类型的数据。这些通信路径并不是动态创建的，也不是在运行时才发现，而必须在设计时就对通信路径达成一致，只有这样，应用才能知道数据从哪里来，到哪里去（尽管大多数通道都必须静态定义，但也存在例外情况，在这些情况下，动态通道更实际也更有用。其中一个例外就是请求/应答中的应答通道。请求者可以创建或获得一个应答者根本不知道的新通道，并把它作为请求消息的返回地址，然后由应答者使用这个通道。另一种例外是支持层次性通道的消息传递系统实现。接收者可以订购层次体系中的父通道，然后发送者把消息发布给接收者完全不知道的一个子通道。订购者仍然会接收到消息。尽管存在这些不寻常的情况，但通道一般都还是在部署前定义，应用也是按照已知的通道集来设计）。

确定通道集——与之相关产生了一个问题，谁来决定哪些消息通道是可用的，是消息传递系统还是应用？换句话说，消息传递系统能指定某种通道并要求应用使用这些通道通信吗？还是要由应用决定需要什么通道，并要求消息传递系统提供这些通道？这个问题不容易回答。要设计所需的通道集，这个过程是不断反复的。首先，应用确定消息传递系统应该提供哪些通道。接着应用要尽可能围绕可用的通道来设计它们的通信，但是当实现不了时，就会请求增加额外的通道。当一组应用已经使用了某个通道集，同时又有新的应用

想要加入时，这些新加入的应用也会使用已有的通道集。已有的应用增加了新的功能时，它们可以请求新的通道。

单向通道——另一个让我们普遍感到混淆的问题是，消息通道是单向的还是双向的？从技术上讲，它既不是单向的，也不是双向的。通道更像是一个桶，有的应用往里面添加数据，其他应用从里面取出数据（不过，这种桶是以一种协作的方式分布在多个计算机上）。但是，由于数据是封装在消息中从一个应用传播到另一个应用的，这就决定了通道的方向性，从而建立了单向的通道。如果通道是双向的，也就是说应用能在同样的通道中发送和接收消息，尽管这在技术上是可行的，但实现双向通道并没有太大的意义，因为应用一般只消费自己的消息——也就是准备发送给其他应用的消息。因此，从各种实用的角度看，通道总是单向的。对于完成双向会话的两个应用，它们需要两个通道，每个通道分别完成一个方向的通信（见下一章中的“请求/应答”）。

4.1.2 消息通道的决策

既然了解了消息通道是什么，下面就要考虑如何使用消息通道。

一对一或一对多——当应用共享一段数据时，你是希望只让另外一个应用共享，还是让其他所有感兴趣的应用都共享？要把数据发送给单个应用，可以使用点对点通道（Point-to-Point Channel）。点对点通道不能保证发送到该通道中的所有数据都肯定传递给同一个接收者，因为通道有可能有多个接收者。但是，它能保证任何数据都只能被其中一个应用接收。如果想让所有的接收应用都能收到数据，就应该使用发布-订购通道（Publish-Subscribe Channel）。采用这种方式发送数据时，通道将有效地复制数据提供给每一个接收者。

确定数据类型——计算机内存中的任何数据都必须符合某种类型：要么是已知的数据格式，要么是含义协商确定的数据结构。否则，所有的数据只不过是一堆字节，不具有任何意义。消息传递系统中的数据也是一样。消息的内容必须符合某种类型，只有这样接收者才能理解数据的结构。数据类型通道（Datatype Channel）要求通道中的所有数据必须采用相同的类型。正是因为这个原因，所以消息传递系统需要大量通道。如果数据可以是任何类型，那么消息传递系统在任何两个应用之间（每个方向上）只需要提供一个通道。

非法和无用消息——消息系统可以确保消息得到正确的传递，但不能保证接收者一定知道该如何处理消息。接收者对数据的类型和含义有自己的期望。当它收到的消息不能满足这些期望时，没有多少工作能做。不过，它可以在一个专门指定的非法消息通道（Invalid Message Channel）上放上奇怪的消息，以便监控这个通道的某个工具能提取这些消息，并确定如何处理。许多消息传递系统都提供了类似于此的一种内置特性，称为死文字通道（Dead Letter Channel），这种通道针对的是那些已经成功发出但最终未能成功传送的消息。而且，应当有一个系统管理工具监控死文字通道，并对无法传送的消息做出处理。

崩溃恢复——如果消息传递系统崩溃，或者因为维护而关机，会对它传递的消息产生什么影响？当其恢复运行时，消息是否还在通道中？默认情况下消息已经丢失，因为消息是存储在内存中的。但是，可靠传输（Guaranteed Delivery）会将通道持久化，消息可以存

储在磁盘中。尽管这样做会降低性能，但是可以增加消息传递的可靠性，尽管消息传递系统本身并不可靠。

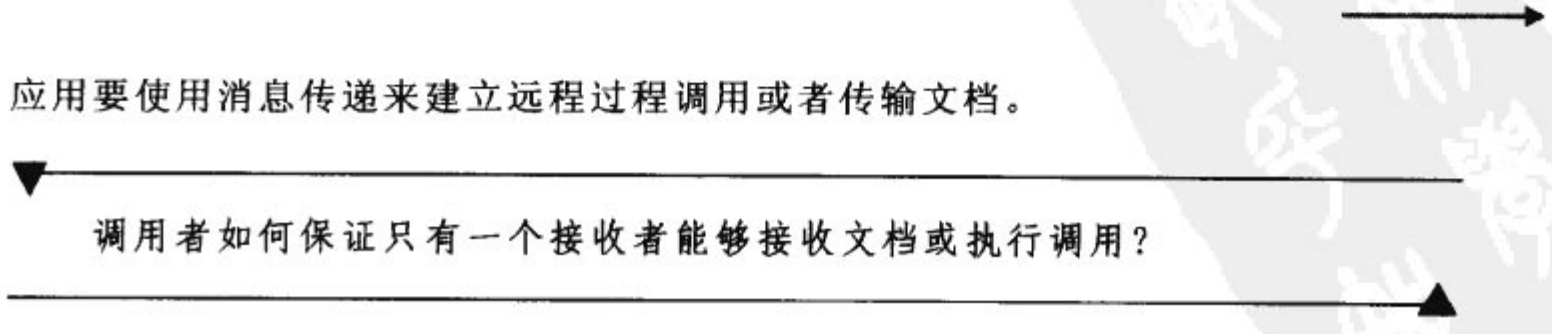
非消息传递客户——如果应用无法连接到消息传递系统，但是仍然想参与消息传递，该怎么办呢？通常这种可能性很小，但是如果消息传递系统能以某种方式连接到应用，如通过它的用户接口、它的业务服务 API、它的数据库，或者通过 TCP/IP 或 HTTP 等网络连接，就可以使用消息传递系统上的通道适配器。利用通道适配器，可以将通道连接到应用，而不必修改应用，甚至不需要消息传递客户和应用在相同的计算机上运行。有时“非消息传递客户”实际上也是消息传递客户，只不过是另一个消息传递系统的客户。在这种情况下，如果一个应用同时作为两个不同消息传递系统的客户，那么这个应用就可以在这两个消息传递系统之间建立一个消息传递桥（Messaging Bridge），把它们有效地连接成一个组合的消息传递系统。

通信中枢——随着越来越多的企业应用连接到消息传递系统，并通过消息传递共享它们的功能，消息传递系统已经成为企业中提供一站式功能共享的中心点。一个新的应用要想加入，只需要知道使用哪个通道请求功能以及使用哪些通道监听返回的结果。消息传递系统本身从本质上构成了一个消息总线（Message Bus），它是一个通信中枢，能够访问企业中各种不断变化的应用和功能。只要从一开始就对此做专门的设计，就能更快、更方便地实现这种集成。

正如你看到的，为了能让应用进行消息传递，不只是要把它们连接到消息传递系统并发送消息而已，还需要有消息通道来传送消息。如果随便地使用某些通道，同样达不到目的，必须根据要共享的数据类型、提供可用数据的应用的种类，以及接收数据的应用的种类等方面设计专门的通道。这一章将介绍设计这些通道时应该考虑哪些方面。

为了便于介绍模式，我们为每种模式提供了一个实例，这些实例取自一个假想的、有所简化的股票交易领域。尽管这些实例并不能作为实现一个真正股票交易系统的基础，但它们确实可以简要、明确地说明如何使用相应的模式。

4.2 点对点通道



远程过程调用的一个优点是它只在一个远程进程上调用，因此接收者要么执行该过程，要么什么也不做（产生异常）。由于接收者只被调用一次，因此它只会执行一次过程。但是对消息传递来说，一旦调用被封装成消息，放置到某个消息通道中，则可能会有多个接收

者从通道中发现这个消息，并执行相应的过程。

消息传递系统可以防止多个接收者监控同一个通道，但是有些调用者可能想把数据发送给多个接收者，对于这些调用者来说，这就会存在不必要的限制。某个通道上的所有接收者可以协调，确保只有其中的一个接收者完成实际的调用。但是这种做法很复杂，会产生大量的通信开销，通常还会在原本独立的接收者之间增加耦合。多个接收者使用一个通道是很理想的，这样一来，可以并发地消费多个消息，但是任何接收者都只能消费一个消息。

在点对点通道上发送消息，可以确保只有一个接收者接收到某个特定的消息。



点对点通道能确保，对于任何给定的消息只有一个接收者能消费。通道的接收者可以有多个，它们能同时消费多个消息，但是其中只有一个能成功地消费某个特定的消息。如果多个接收者都想消费某一个消息，通道将保证只有其中的一个接收者能成功，因此接收者之间不必相互协调。

当点对点通道只有一个消费者时，消息只会消费一次，这一点不足为奇。当通道有多个消费者时，它们会成为竞争消费者（Competing Consumer），通道将保证每条消息只能被其中一个消费者消费。采用这种设计，消息的消费和处理都很容易扩展，因为这些操作可以在多个消费者之间实现负载平衡，这些消费者可能运行在多个计算机上的多个应用中。

使用点对点通道只能把消息发送给某一个可用的接收者，要把消息发送给所有可用的接收者，应该使用发布-订购通道。为了通过消息传递实现远程过程调用，应当使用带有一对点对点通道的请求/应答（Request-Reply）模式。调用是一条命令消息（Command Message），应答是一条文档消息（Document Message）。

例：股票交易

在一个股票交易系统中，完成特定交易的请求是一个消息，这个消息只能由一个接收者消费并加以处理，因此该消息必须放在一个点对点通道中。

例：JMS 队列

在 JMS 中，点对点通道实现了 Queue 接口。发送者使用 QueueSender 发送消息，每个接收者使用自己的 QueueReciever 接收消息[JMS1.1]、[Hapner]。

应用使用 QueueSender 发送消息的代码如下：

```
Queue queue = // 通过JNDI获取队列
QueueConnectionFactory factory = // 通过JNDI获取连接工厂
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true,
 Session.AUTO_ACKNOWLEDGE);
QueueSender sender = session.createSender(queue);

Message message = session.createTextMessage("The contents of the
 message.");

sender.send(message);
```

应用使用 QueueReceiver 接收消息的代码如下：

```
Queue queue = // 通过JNDI获取队列
QueueConnectionFactory factory = // 通过JNDI获取连接工厂
QueueConnection connection = factory.createQueueConnection();
QueueSession session = connection.createQueueSession(true,
 Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(queue);

TextMessage message = (TextMessage) receiver.receive();
String contents = message.getText();
```

注意：JMS 1.1 统一了点对点和发布-订购领域的客户 API，因此这里给出的代码可以采用 Destination、ConnectionFactory、Connection、Session、MessageProducer 以及 MessageConsumer 简化，而不是特定于 Queue 的类。

例：.NET MessageQueue

在.NET 中，MessageQueue 类实现了一个点对点通道[SysMsg]。MSMQ 实现了.NET 消息传递，它在 3.0 版以前只支持点对点消息传递，因此.NET 是支持点对点通道的。尽管 JMS 划分了连接工厂、连接、会话、发送者和队列的责任，而 MessageQueue 身兼数职，会完成以上所有工作。

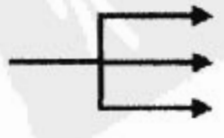
发送消息到 MessageQueue 时采用的代码如下：

```
MessageQueue queue = new MessageQueue("MyQueue");
queue.Send("The contents of the message.");
```

从 MessageQueue 接收消息的代码如下：

```
MessageQueue queue = new MessageQueue("MyQueue");
Message message = queue.Receive();
String contents = (String) message.Body();
```

4.3 发布-订购通道



应用使用消息传递来通知事件。

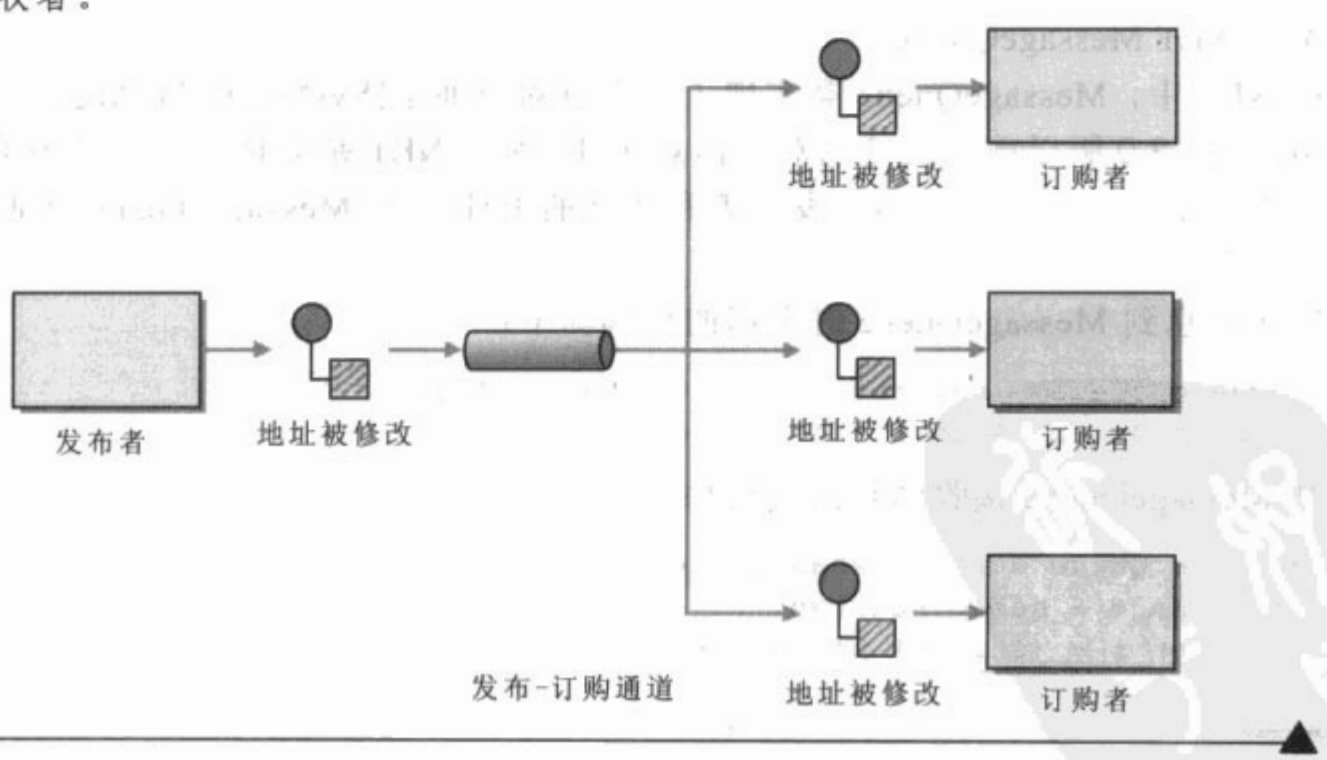
如何让发送者把事件广播给所有感兴趣的接收者？

幸运的是，已经有很成熟的模式来实现广播。观察者（Observer）模式[GoF]说明了需要把观察者与它们的主体（也就是事件的发起者）分离，从而使主体可以很容易地把事件通知提供给所有感兴趣的观察者，而不管有多少观察者存在（或者一个都没有）。发布-订购模式[POSA]扩展了观察者模式，增加了事件通道的概念，该通道用于传递事件通知。

以上只是原理，那么如何具体使用消息传递呢？事件可以被封装为消息，这样的话，消息传递就能把事件可靠地传递给观察者（订购者）。因此，事件通道也是消息通道。但现在的问题是，消息传递通道如何把事件通知给所有的订购者？

对于指定的事件，每个订购者只能获得一次通知，不能重复得到相同事件的通知。只有所有订购者都得到通知后，该事件才算是得到消费，一旦消费，该事件就不能再出现在通道中。如果让订购者相互协调来确定消息何时得到消费，这又违背了观察者模式的解耦合原则。并发的消费者不应该相互竞争，而是应该共享事件消息。

把事件发送到发布-订购通道，该通道负责把某个特定事件的副本发送给每一个接收者。



发布-订购通道的工作原理如下：它有一个输入通道，由这个输入通道分解出多个输出通道，每个输出通道对应一个订购者。当事件发布到通道中时，发布-订购通道会把消息的副本发送给每个输出通道。每个通道的输出端只有一个订购者，它只能将消息消费一次。采用这种方式，每个订购者只能获得消息一次，而且消息副本得到消费后就会从输出通道

中消失。

发布-订购通道可以作为一个有用的调试工具。尽管消息只面向一个接收者，但是通过采用发布-订购通道，你就能监听消息通道，而不会干扰已有的消息流。监控通道上的全部流量对于调试消息传递应用很有帮助。这样一来，你不必在参与消息传递解决方案的每个应用中都插入大量的打印语句。创建一个程序来监听所有活动通道上的消息，并把消息记录到一个日志文件中，这样就像消息存储库（Message Store）一样，可以提供许多相同的好处。

但是，利用发布-订购通道实现监听功能也会带来缺点。如果你的消息传递解决方案要在工资系统与财会系统之间传递工资数据，你可能不希望任何人通过编写一个简单的程序就能监听到所传递的消息。点对点通道从某种程度上能缓解这个问题：因为监听者一旦从通道上消费了消息，这会使消息突然消失，这样解决方案就能很快察觉到。但是，许多消息队列的实现提供了偷看（peek）功能，能让消费者查看队列中的消息而不会消费任何消息。因此，应当根据某种安全策略对消息通道的订购加以限制。许多（并不是所有）商业消息传递实现都提供了这种限制。另外，如果创建一个监控工具，它能把活跃的消息通道订购者记录下来，这将是一个非常有用的系统管理工具。

通配订购者

许多消息传递系统允许发布-订购通道的订购者指定特定的通配符。这是一种很强大的技术，允许订购者一次订购多个通道。例如，如果一个应用把消息发送给 MyCorp/Prod/OrderProcessing/NewOrders 和 MyCorp/Prod/OrderProcessing/CancelledOrders 通道，一个应用可以订购 MyCorp/Prod/OrderProcessing/*来接收所有与订单处理有关的消息，另一个应用可以订购 MyCorp/Dev/**来接收由开发环境中的所有应用发送的任何消息。只有订购者才能使用通配符，发布者一般被要求要把消息发布给特定的通道。不同的消息传递开发商提供的通配订购能力和使用的语法都有所不同。

事件消息一般都发送到发布-订购通道上，因为通常会有多个相关的订购者对同一事件感兴趣。订购者既可是持久的，也可以不持久，请参见第 10 章“消息传递端点”中的“持久订购者”部分。如果要求订购者必须对通知做出确认，则应该使用请求/应答通道，其中通知算是请求，而确认是应答。如果把每条消息都存储在发布-订购通道中，直到所有订购者都已将消息消费掉，这需要占用大量的消息存储空间。为了缓解存储压力，发送给发布-订购通道的消息可以使用消息到期（Message Expiration）。

例：股票交易

在股票交易系统中，许多系统需要得到交易完成的通知，因此可让它们都作为一个发布-订购通道的订购者，这个通道会发布交易完成消息。

同样，可能会有许多消费者对显示或处理当前股价数据感兴趣。因此，股价应该通过一个发布-订购通道广播。

例：JMS Topic

在 JMS 中，发布-订购通道实现了 Topic 接口。发送者使用 TopicPublisher 发送消息，每个接收者使用自己的 TopicSubscriber 接收消息[JMS 1.1]、[Hapner]。

应用使用 TopicPublisher 发送消息的代码如下：

```
Topic topic = // 通过JNDI获得主题
TopicConnectionFactory factory = // 通过JNDI获取连接工厂
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true,
 Session.AUTO_ACKNOWLEDGE);
TopicPublisher publisher = session.createPublisher(topic);

Message message = session.createTextMessage("The contents of the
 message.");

publisher.publish(message);
```

应用使用 TopicSubscriber 接收消息的代码如下：

```
Topic topic = // 通过JNDI获取主题
TopicConnectionFactory factory = // 通过JNDI获取连接工厂
TopicConnection connection = factory.createTopicConnection();
TopicSession session = connection.createTopicSession(true,
 Session.AUTO_ACKNOWLEDGE);
TopicSubscriber subscriber = session.createSubscriber(topic);
TextMessage message = (TextMessage) subscriber.receive();
String contents = message.getText();
```

注意：JMS1.1 统一了点对点和发布-订购领域的客户 API，因此这里给出的代码可以简化为使用 Destination、ConnectionFactory、Connection、Session、MessageProducer 和 MessageConsumer，而不是采用特定于 Topic 的类。

例：MSMQ 一对多消息传递

MSMQ 3.0 [MSMQ]提供了一个新特性，即一对多消息传递模型（one-to-many messaging model），它有两种不同的方式。

1. **实时消息传递组播**——这种方式与发布-订购方式基本相同，但是它的实现完全依赖于基于实用普通组播（Pragmatic General Multicast，PGM）协议的 IP 组播，因此对于其他非 IP 的协议，这个特性不适用。

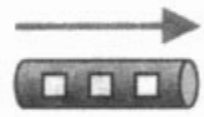
2. **分布列表和多元素格式名**——利用分布列表（Distribution List），发送者能把消息明确地发送给一组接收者（但是这违背了观察者模式的主旨，因为发送者必须知道有哪些接收者）。因此，这个特性更类似于一个接受者列表（Recipient List），而不是一个发布-订购通道。多元素格式名（Multiple-Element Format Name）是一个符号通道指示器，它能动态映射至多个实际的通道，这一点更符合发布-订购通道的精神，但是仍然要求发送者在实际通道和符号通道之间做出选择。

.NET 公共语言运行时（Common Language Runtime，CLR）未对使用一对多消息传递模型提供直接的支持。可以通过嵌入在.NET 代码中的 COM 接口[MDMSG]访问这个功能。

例：简单的消息传递

JMS 发布-订购的示例见第 6 章“中场演练：简单的消息传递”，其中给出了一个例子，来说明如何使用消息传递在多个过程之间实现观察者。

4.4 数据类型通道



应用使用消息传递传送不同类型的数据，如不同类型的文档。

应用如何发送数据项才能让接收者知道如何加以处理？

所有消息都是同一种消息类型的实例，消息类型由消息传递系统定义，消息的内容归根结底都只是一个字节数组。对消息传递系统来说，要传输消息，了解这样一个简单的结构（一堆字节）就足够了，但是对接收者而言，仅此还不足以处理消息的内容。

接收者必须知道消息内容的数据结构和数据格式。数据结构有可能是字符数组、字节数组、串行化对象、XML 文档等等。数据格式可能是字节或字符的记录结构、串行化对象的类、XML 文档的模式定义。这些知识通常都泛指为消息类型，它表示消息内容的结构和格式。

接收者必须知道所接收消息的类型是什么，否则将无法处理这些消息。例如，发送者可能想发送不同的对象，如订单、股价和查询请求；对应每种类型的对象，接收者也会采取不同的步骤加以处理，因此它必须知道消息的类型是什么。如果发送者只是把所有这些对象通过一个消息通道发送给接收者，那么接收者将不知道如何处理每条消息。



混合的数据类型

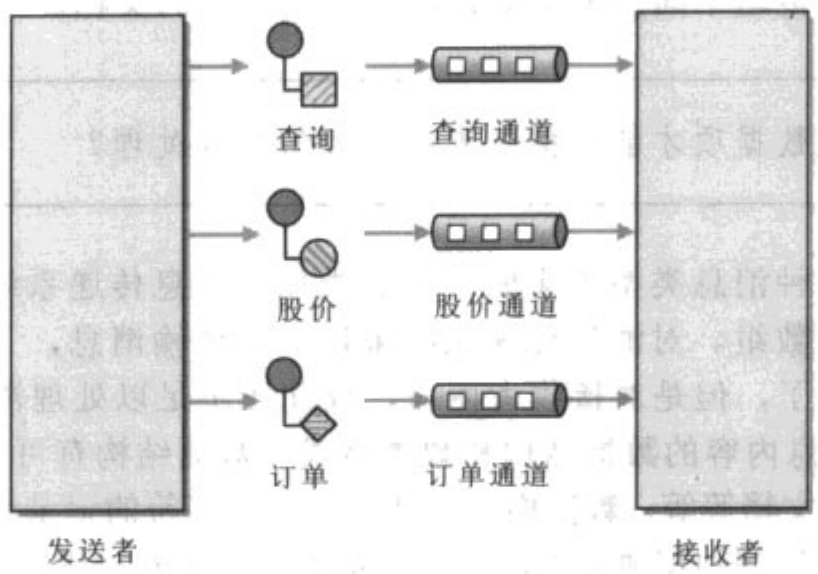
发送者很清楚所发送消息的类型，它应该如何把这个信息告诉接收者呢？发送者可以把一个标志放在消息的首部（见格式指示符），不过，这样接收者就需要一条 case 语句。发送者也可以把数据封装在一个命令消息中，并且针对每种类型的数据使用不同的命令，尽管这个消息所做的只是向接收者传送数据，但是这样就能告诉接收者应该如何处理数据。

当处理一个数据项集合时，会出现一个类似的问题（这与消息传递无关）。这个集合必

须是同构的，也就是说，所有的数据项都有相同的类型，因此处理程序知道数据项的类型是什么，也知道如何进行处理。许多集合实现并不要求所有数据项都有相同的类型；相反，可以向一个集合增加不同类型的数据项，但是处理这些数据项的代码不知道如何分别处理各个数据项，因为它并不了解特定数据项是什么类型。

相同的道理同样适用于消息传递，因为通道中的消息就像相同类型的数据项。最简单的解决方案是让所有的消息都有相同的数据格式。如果格式必须不同，则必须都带有一个可靠的格式指示符（format indicator）。尽管通道没有强制要求所有消息都有相同的类型，但接收者必须知道每条消息的类型，否则将无法处理。

使用单独的数据类型通道（Datatype Channel），这样，特定通道上的所有数据都有相同的类型。



为每种类型的数据使用单独的数据类型通道，这样指定通道上的所有消息都包含相同类型的数据。发送者知道数据类型是什么，必须选择相应的通道来发送数据。接收者只要知道在哪个通道上接收数据，就能知道数据的类型是什么。

如上图所示，由于发送者想要发送三种不同类型的数据（订单、股价和查询请求），它应该使用三个不同的通道。发送一个数据项时，发送者必须选择相应的数据类型通道来发送该数据。接收数据项时，接收者能够知道数据项的类型，因为它知道在哪个数据类型通道上接收数据项。

服务质量（QoS）通道

与之相关的一个策略是服务质量通道（Quality-of-Service Channel）。有的时候，企业想利用不同等级的服务来传递不同的消息。例如，输入的新订单消息是业务收入的主要来源，因此应该通过非常可靠的通道传递（如使用可靠传输通道），尽管可能存在潜

在的性能开销。与之相反，如果丢失了一条表示订单状态请求的消息，这倒不是太要紧，因此我们倾向于使用更快而可靠性稍差的通道来传递这些消息。这些可以采用同一个通道来实现，例如，可以使用消息优先级策略。但是，最好能在创建通道时定义服务质量参数，而不是交由发送消息的应用代码来做此决定。因此，采用不同的通道传递不同的消息组是最佳的选择，这样每个通道的服务质量可根据这组消息的需要加以调整。

在介绍消息通道时曾说过，通道尽管很廉价，但是并不是免费的。应用可能需要传递许多不同的数据类型，不可能为每种类型都创建一个通道。在这种情况下，可以为每种类型指定不同的选择性消费者（Selective Consumer），从而使多个数据类型可共享一个通道。这样一来，一个物理通道就可以扮演多个逻辑数据类型通道（这个策略称为多路复用）。数据类型通道解释了为什么通道中的所有消息必须采用相同的格式，而规范数据模型则说明了企业所有通道中的所有消息如何遵循统一的数据模型。

如果我们想使用数据类型通道，但是已有的消息发布者只是把所有的消息都发送给一个通道，那么可以使用基于内容的路由器（Content-Based Router）将消息解复用（demultiplex）。路由器把消息流划分到多个数据类型通道中，其中每个通道只有一种类型的消息。

消息分派器（Message Dispatcher）除了能提供并发的消息消费外，还可以采用特定于类型的方式来处理一组一般的消息。每条消息必须指定它的类型，一般通过消息首部中的格式指示符来指定。分派器检测出消息的类型后，会将它发布给特定类型的处理器来处理。通道中的消息仍然都是相同的类型，但是这个类型是分派器支持的更一般的类型，而不是各个处理器所需的特殊类型。

格式指示符（Format Indicator）用于区分相同数据的不同格式，从而能在相同的数据类型通道上发送不同格式的数据。

例：股票交易

在股票交易系统中，如果股价请求的格式与交易请求的格式不同，系统应当使用不同的数据类型通道传递每种类型的请求。同样地，地址变更声明采用的格式可能与部门经理变更声明的格式不同，因此每种类型的声明应该有自己的数据类型通道。

4.5 非法消息通道



应用使用消息传递来接收消息。

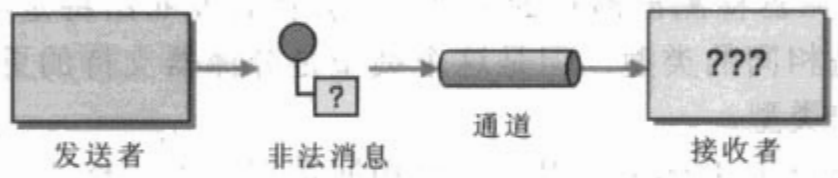
如果接收到无意义的消息，消息传递接收者如何妥善地处理？

从理论上讲，消息通道中的任何东西都是消息，消息接收者只是处理消息。但是，为了处理一条消息，接收者必须能解释其数据，并理解数据的含义。并不是每个消息都可以理解：消息体可能会导致解析错误、词法错误或验证错误。消息首部可能缺少必要的属性，或者某些属性值没有意义。发送者可能会把一条完好的消息发送到错误的通道上，并传送到错误的接收者。一个恶意的发送者会故意发送不正确的消息，目的只是要干扰接收者。接收者可能无法处理接收到的所有消息，因此它必须采取另外的某种方式来处理它认为不合法的消息。

消息通道应当是一个数据类型通道，通道中的每条消息应当是该通道所支持的数据类型。如果发送者向通道发送了一条数据类型不正确的消息，消息传递系统能够成功地传递这个消息，但是接收者无法识别，也不知道如何处理。

例如，假设一个通道只能包含文本消息，但其中出现了一个字节消息，这个消息的数据类型或格式就是不正确的。格式不正确的消息也属于这种非法消息，比如结构不正确或者没有遵循约定 DTD 或模式（Schema）的 XML 文档。对消息传递系统来说，这些消息没有错误，但是接收者将无法处理它们，因此它们是非法的。

如果一个消息的首部中没有包含接收者期望的字段值，这个消息也是非法的。如果要求消息首部必须带有关联标识符（Correlation Identifier）、消息序列（Message Sequence）标识符、返回地址（Return Address）等属性，但是实际的消息却没有这些属性，那么即使消息传递系统能正确地发送这些消息，接收者也无法成功地处理。



非法消息

当接收者发现要处理的消息不正确时，应该对消息做些什么呢？可以忽略这个问题，把消息再放回通道，但是这个消息可能还会被同一个接收者重新消费，或者被其他类似的接收者消费。同时，如果忽略这些非法的消息，它们会把通道搞得一团糟，并降低系统的性能。接收者也可以消费非法的消息并把它丢掉，但是这会使本应检测出来的消息传递问题隐藏起来。系统需要一种方法把不正确的消息从通道中清除出去，同时能把它们安置在另一个地方，一方面要不碍事，另一方面还能被检测到，从而让消息传递系统诊断出问题所在。

接收者应当把不正确的消息发送给非法消息通道，这是一个特殊的通道，专门针对接收者无法处理的消息。



设计应用使用的消息传递系统时，管理员必须为应用定义一个或多个非法消息通道，以供使用。非法消息通道不用于正常的、成功的通信，因此，如果被不正确的消息干扰，也不会带来问题。如果一个错误处理程序想诊断不正确的消息，可以使用非法通道的一个接收者来检测可能出现的非法消息。

非法消息通道类似于消息传递的错误日志。当应用中出现某种错误时，把错误记录下来是一个很好的想法。同样的，当消息处理出现错误时，把消息发送到非法消息通道也是一个好主意。如果查看通道的人不清楚消息为什么是非法的，应用还应该更为详细地记录错误。

要记住，消息并不是一直合法或非法的，它与接收者的上下文和期望有关。对某个接收者合法的消息对另一个接收者可能是非法的，这两种接收者不能共享相同的通道。如果一个消息对通道的一个接收者是合法的，那么对通道的所有其他接收者也应该是合法的。同样，如果一个接收者认为某个消息是非法的，所有其他接收者也会认为这个消息是非法的。发送者应该明确地知道，它在通道上发送的消息在通道接收者看来应该是合法的。否则，接收者将忽略发送者的消息，并把消息重新路由给非法消息通道。

如果消息的结构正确，但是它的内容存在语义错误，此时也会出现类似的问题，但并不完全相同。例如，如果有一个命令消息要求接收者删除一条并不存在的数据库记录。它不属于消息传递错误，而是一个应用错误。同样，系统会尝试把它发送给非法消息通道，但是消息本身并没有错误，因此把它当作是非法的消息很容易产生误解。类似这样的错误应该作为非法的应用请求进行处理，而不是非法的消息。

当接收者实现成服务激励器（Service Activator）模式或消息传递网关（Messaging Gateway）模式时，更容易区分出消息处理错误和应用错误。这些模式把消息处理代码与应用的其余代码区分开来。如果处理消息时出现错误，那么消息就是非法的，应该传递给非法消息通道；如果在应用处理消息数据的时候发生错误，那么它就是一个应用错误，这与消息传递没有关系。

非法消息通道的内容通常会被忽略，它与一般被忽略的错误日志有相同的作用。非法消息通道中的消息指出了应用集成存在的问题，因此这些消息不应该忽略。相反，要对它们进行分析，来确定问题出在什么地方，并加以克服。在理想情况下，消费非法消息、确定错误的原因、解决根本问题，这一过程应该是自动进行的。但是，错误往往是由于代码编写或配置不当造成的，这需要开发人员或系统分析员做出评估和修复。至少，使用消息传递和非法消息通道的应用应当提供一个过程来监控非法消息通道，当通道中包含消息时，应该向系统管理员发出警告。

许多消息传递系统都实现了一个类似的概念：死文字通道。非法消息通道中是能够发送和接收但是不能处理的消息，而死文字通道中是消息传递系统无法正确传送的消息。

例：股票交易

在股票交易系统中，执行交易请求的应用可能会接收到询问当前股价的请求，或者接收的交易请求没有指明要购买哪种股票或参股份额是多少，也可能收到的交易请求没有指

定要把交易确认发送给谁。在这些情况下，应用接收到的就是非法消息，这些消息不能满足应用完成交易请求处理的最低需要。一旦应用确定消息是非法的，就应该把消息重新发送给非法消息通道。发送交易请求的各个应用可以监控非法消息通道，来确定它们发出的请求是否被丢掉。

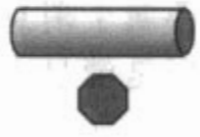
例：JMS 规范

JMS 规范中建议：如果 MessageListener 得到的消息无法处理，应该由一个合理的监听者把消息转发给“一个特定于应用的‘不可处理消息’目标”[JMS1.1]。这种不可处理消息目标就是一个非法消息通道。

例：简单的消息传递

JMS 请求/应答示例和.NET 请求/应答示例（见第 6 章“简单的消息传递”）给出了一个例子，说明了如何实现接收者，从而把它们无法处理的消息转发给非法消息通道。

4.6 死文字通道



企业使用消息传递来集成应用。

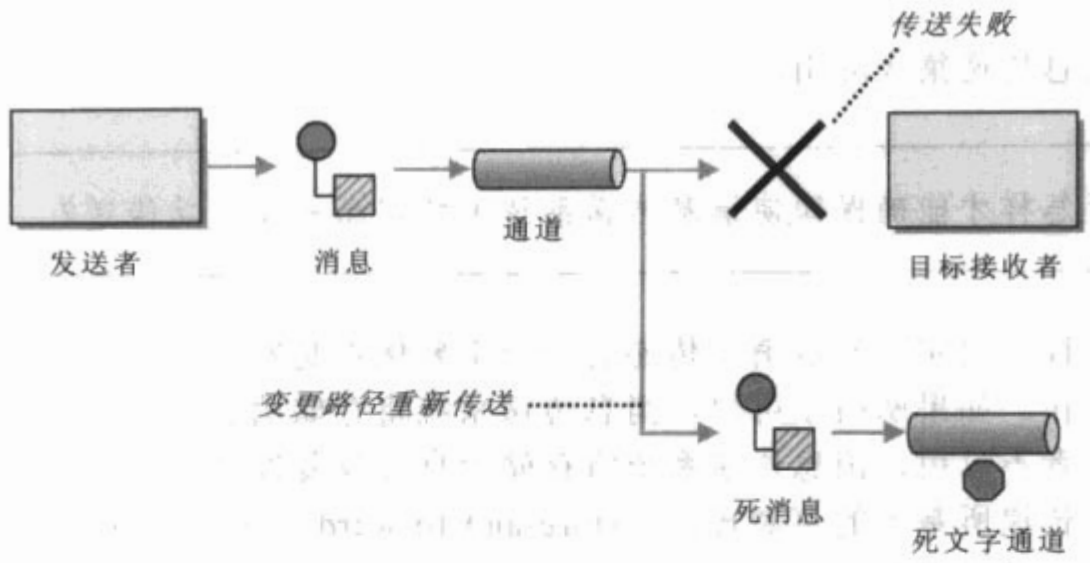
消息传递系统应该如何处理无法传送的消息？

如果接收者接收到的消息无法处理，它应该把非法的消息转发给非法消息通道。但是，如果一开始消息传递系统就无法把消息传送给接收者，应该怎么办呢？

有很多原因会造成消息传递系统无法传送消息。比如，消息传递系统没有正确地配置好消息通道；消息已经发送但还没有接收到时，或者正在等待接收的过程中，消息通道被删除；消息在传输前就已经到期（见消息到期）；如果消息没有一个明确的到期时间，倘若很长时间都无法传送，它将永远也不会超时；尽管消息提供了一个选择值，但是所有选择性消费者都无法识别和处理这个值，那么这个消息永远也不会读取，并最终成为死消息；消息的首部存在某种错误，以至于无法正确传送。

一旦消息传递系统确定无法传送某个消息，就必须对消息做出处理。可以仅把消息留在原处，但这样会扰乱系统。也可以尝试把消息返回给发送者，但是发送者并不是接收者，因此并不能识别返回的消息。还可以仅删除消息，并希望没有任何接收者在等待这个消息，但是这会让发送者认为它已经成功地发送了消息，而且认为消息已经被接收并处理。

当消息传递系统确定无法或不应该传送某条消息时，可以选择把该消息转移到一个死文字通道中。



如果消息传递系统提供了死文字通道，它会采用特定的方式工作，这是由消息传递系统的特定实现决定的。这种通道可以称为“死消息队列”[Monson-Haefel]或“死文字队列”[MQSeries]、[Dickman]。通常，每台安装了消息传递系统的计算机都有自己的本地死文字通道，因此，不管消息在哪台计算机上死掉，它都能从一个本地队列转移到另一个本地队列中，而不受网络不确定性的影响。这种通道会记录消息死掉时所在的是哪一台计算机。当消息传递系统移动消息时，死文字通道还会记录原先打算传送消息的原始通道。

死消息与非法消息的区别在于，如果消息传递系统认为一个消息是死消息，则无法成功地传送这个消息，而非法消息能得到正确传送，只是不能被接收者处理。确定某条消息是否应该转移到死文字通道，是由消息传递系统对消息的首部进行评估分析后决定的。与此不同，如果接收者感兴趣的消息体或消息首部中的特定字段有问题，接收者则会将这个消息转移到非法消息通道。对接收者来说，死消息的确定和处理是自动进行的，而非法消息必须由接收者自己处理。使用消息传递系统的开发人员始终坚持：任何类型的死消息都应该由消息传递系统处理，他只负责设计自己的非法消息处理程序；有些消息看似为死消息，而且消息传递系统没有处理，这些消息也要由非法消息处理程序来处理。

例：股票交易

在股票交易系统中，希望完成交易的应用可以发送一个交易请求。为了确保交易请求能在合理的时间期限内接收到（可能少于 5 分钟），请求者把请求消息的到期时间设置为 5 分钟。如果消息传递系统无法在规定的时间期限内传送请求，或者交易应用无法及时地接收消息（从通道中读取），那么消息传递系统将把消息从交易请求通道中取出，并放入死文字通道。交易系统要监控系统的死文字通道，以确定是否漏掉了交易。

4.7 可靠传输



企业使用消息传递集成应用。

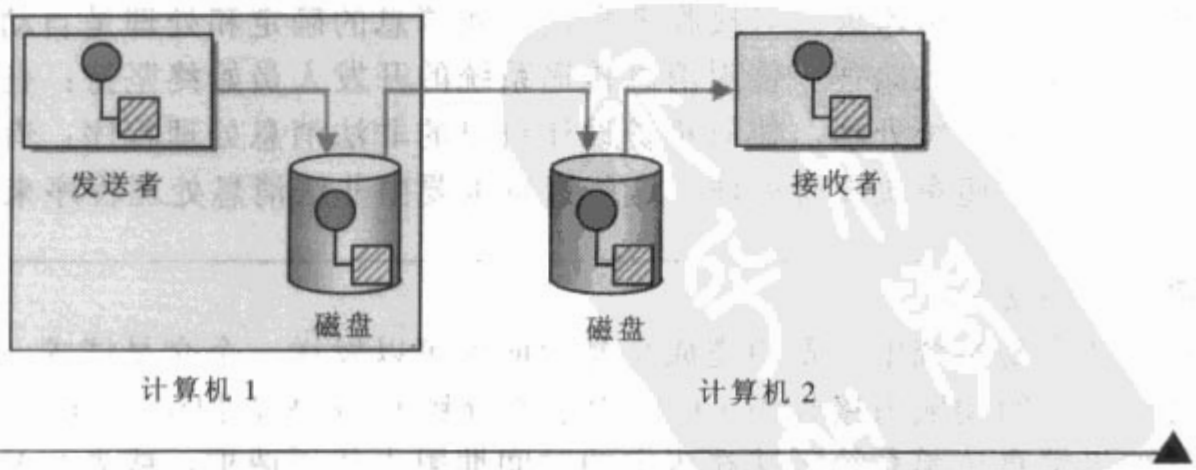
发送者怎样才能确保即使消息传递系统失败消息也能成功传送？

通过远程过程调用实现异步消息传递的一个主要优点是发送者、接收者和连接它们的网络不必同时工作。如果网络不可用，消息传递系统将存储消息，直到网络重新可用。同样地，如果接收者不可用，消息传递系统将存储消息并反复发送消息，直到接收者重新可用。这就是消息传递所基于的存储转发（store-and-forward）过程。因此，转发消息之前，应该把消息存储在哪里呢？

默认情况下，消息传递系统把消息存储在内存中，直到能把消息成功转发给下一个存储点。只要消息传递系统能可靠运行，这种操作方式就有效。但是，如果消息传递系统崩溃（例如，由于其中一台计算机掉电，或者是消息传递过程异常中止），存储在内存中的消息就会丢失。

大多数应用也要应对相同的问题。如果应用崩溃，所有存储在内存中的数据就会丢失。为了防止数据丢失，应用使用文件和数据库把数据持久地保存在磁盘上，即使系统崩溃也不会出现数据丢失问题。消息传递系统需要采用类似的方法更长久地保存消息，使得在系统崩溃时消息不会丢失。

使用可靠传输可以持久地保存消息，即使消息传递系统崩溃，消息也不会丢失。



采用可靠传输，消息传递系统利用内建的数据存储库保存消息。安装有消息传递系统的每台计算机都有自己的数据存储库，因此可把消息保存在本地。当发送者发送消息时，要等到消息安全地存储到发送者的数据存储库之后，发送操作才能成功地完成。同样，只

有消息成功地转发并存储到下一个数据存储库中后，消息才能从本地数据存储库中删除。采用这种方式，一旦发送者成功地发送了消息，消息将至少存储在一台计算机的磁盘上，直到成功地传送给接收者并得到确认。

持久性可以增强可靠性，但这是以性能为代价的。因此，如果并不介意因消息传递系统崩溃或关机造成的消息丢失，可以不使用可靠传输，这样消息能更快地通过消息传递系统发送。

还要考虑一点，在流量很大的情况下，可靠传输会占用大量的磁盘空间。如果生产者每秒钟会产生成百上千的消息，那么因为网络带宽的制约，几个小时后就会占用大量的磁盘空间。因为网络不可用，消息必须存储在生产消息的计算机的本地磁盘上，而它在设计时并没有考虑到需要存储如此多的数据。基于这些原因，有的消息传递系统允许配置一个重传超时（retry timeout）参数，可以指定消息会在消息传递系统中缓冲多长时间。在某些流量很高的应用中（例如不断向终端发送股票报价），超时时间设置得很短，如几分钟。幸运的是，在很多这样的系统中，采用的是事件消息（Event Message），这种消息在很短的时间间隔后就能安全地丢掉（见“消息到期”）。

测试和调试期间最好关闭可靠传输。这样的话，通过停止和重启消息传递服务器，就能清空所有的消息通道。如果仍有消息在排队，这会使调试原本很简单的消息传递程序变得非常困难。例如，假设发送者和接收者通过一个点对点通道连接，如果某个消息仍然存储在通道中，在发送者产生新的消息前，接收者必须处理这条消息。这是异步、可靠的消息传递中一个共同的调试缺陷。许多商业消息传递的实现支持对单个队列的清除操作，使调试能完全重新开始。

可靠传输是如何保证消息传递的？

计算机系统中的可靠性可采用“9 的个数”加以衡量，如 99.9%。这等于说某某事情很少能达到 100%可靠，因为可靠性从 99.9%增加到 99.99%需要成本呈指数增加。这一规则同样适用于可靠传输。消息丢失的可能性始终存在着。例如，如果存储持久消息的磁盘出现故障，消息将丢失。使用冗余磁盘存储，可降低失效的可能性，进而提高磁盘存储的可靠性。但是它只可能增加可靠性数字的“9”的个数，而不能达到 100%的可靠。而且，如果网络长时间不可用，被存储的消息将占满计算机的磁盘空间，导致消息的丢失。总之，可靠传输是为了防止消息传送过程中的意外损耗，如计算机故障或网络故障，但是不能达到 100%的可靠。

采用.NET 的 MSMQ 实现持久通道时，通道必须被声明为面向事务处理的（Transactional），也就是说发送者必须是面向事务处理的客户。在 JMS 中，采用发布-订购通道，可靠传输只能保证消息能传输给活跃的订购者。为了保证非活跃的订购者也能接收到消息，订购者必须成为持久的订购者（Durable Subscriber）。

例：股票交易

在股票交易系统中，交易请求和交易确认必须采用可靠传输方式，以防止它们被丢失。

地址变更声明也应该采用可靠传输方式，但是价格的更新不一定需要采用可靠传输，因为损失一部分这样的消息无关紧要，并且可靠传输也无法承受它们的发送频率。

如果股票交易系统中的一些价格变更订购者希望成为持久的订购者，那么价格变更通道也应该采用可靠传输。而其他的订购者可能不需要持久性，也不想承担可靠传输带来的负担。那么应该如何满足这些不同的需求呢？系统可实现两种价格变更通道，一种采用可靠传输，另一种不采用。只有请求所有的价格更新的订购者才应该订购持久通道，并且它们的订购申请应该是持久的。发布者向持久通道发布更新的频率要低一些，因为它的处理负载要大一些（见“数据类型通道”中讨论的服务质量通道策略）。

例：JMS 持久消息

在 JMS 中，可分别为每条消息设置消息的持久性。换句话说，某个特定通道中的一些消息可以是持久的，而另外的消息却不是持久的[JMS 1.1]、[Hapner]。

当 JMS 发送者想要产生持久消息时，可使用它的 MessageProducer，把消息的 JMSDeliveryMode 设置为 PERSISTENT。发送者对每条消息设置持久性的操作如下：

```
Session session = // 获取会话
Destination destination = // 获取目标
Message message = // 创建消息
MessageProducer producer = session.createProducer(destination);
producer.send(
    message,
    javax.jms.DeliveryMode.PERSISTENT,
    javax.jms.Message.DEFAULT_PRIORITY,
    javax.jms.Message.DEFAULT_TIME_TO_LIVE);
```

如果应用想让所有的消息成为持久的，它可把持久特性设置为消息生产者的默认属性。

```
producer.setDeliveryMode(javax.jms.DeliveryMode.PERSISTENT);
```

（实际上，消息生产者的默认传送模式就是持久的）现在，由这个生产者发送的消息将自动成为持久的，它们只要被发送出去就行了。

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

在这个期间由其他消息生产者发送到相同通道上的消息也可能是持久的，这要由生产者如何配置它们的消息决定。

例：IBM WebSphere MQ

在 WebSphere MQ 中，可分别为每个通道或每条消息设置可靠传输。如果通道不是持久的，那么通道中的消息也不能是持久的。如果通道是持久的，可以对通道进行配置，使所有经它发送的消息都自动成为持久的消息，或者保持每个消息按持久或非持久方式发送。

当在消息传递系统中创建通道时，可将它配置为持久的（或非持久的）。例如，可把通道配置为：所有经过它的消息都是持久的。

```
DEFINE Q(myQueue) PER(PERS)
```

或者也可以把该通道配置为：由消息的发送者指定每条消息是否采用持久或非持久发送。

```
DEFINE Q(myQueue) PER(APP)
```

如果通道被设置成允许发送者指定持久性，那么可采用前面介绍的方法，通过 JMS MessageProducer 设置传送模式属性。如果通道被设置成把所有消息持久化，那么由 MessageProducer 设置的传送模式属性将被忽略[WSMQ]。

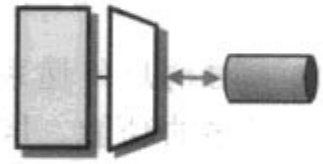
例：.NET 持久消息

采用.NET，通过创建面向事务处理的 MessageQueue 可以设置持久消息。

```
MessageQueue.Create("MyQueue", true);
```

所有经过这个队列发送的消息将被自动转换为持久的[Dickman]。

4.8 通道适配器



许多企业使用消息传递集成多个单独的应用。

应该如何把应用连接到消息传递系统，使之能发送和接收消息？

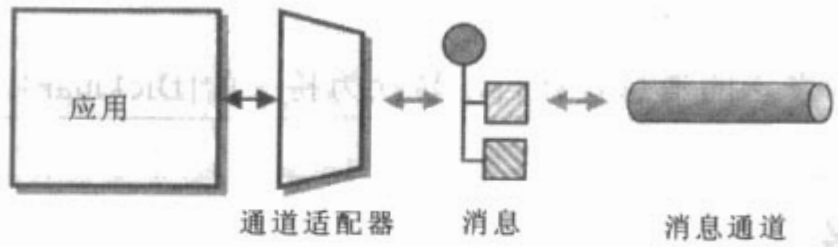
大多数应用在设计时没有考虑与消息传递基础设施配合工作。产生这种限制的原因有很多。许多应用是按照自我包含的、独立的解决方案开发的，即使它们所包含的数据或功能能被其他系统提供。例如，许多大型主机的应用被设计成一个包容一切的程序，它不需要与其他应用交互。但是遗留系统的集成是目前企业集成解决方案中最常见的集成目标。另外一个原因是许多面向消息的中间件系统提供专用的 API，因此应用的开发人员必须为消息传递系统编写多种接口，一种接口应对一个潜在的中间件开发商。

如果应用需要与其他应用交换数据，它们在设计时一般采用更通用的接口机制，如文件交换或数据库表。读写文件是一种不依赖于特定开发商 API 的基本的操作系统功能。同样，大多数商业应用已经把数据持久保存到数据库中，因此要把其他系统的数据存储到数据库表中只需付出很少的努力。应用也可以采用通用 API 把内部的功能提供给任何其他的集成策略使用，包括消息传递。

其他应用也许能通过简单的协议，如 HTTP 或 TCP/IP，进行通信。但是，这些协议并没有像消息通道那样提供同样的可靠性，应用所使用的数据格式也与应用相关，无法与公共的消息传递解决方案兼容。

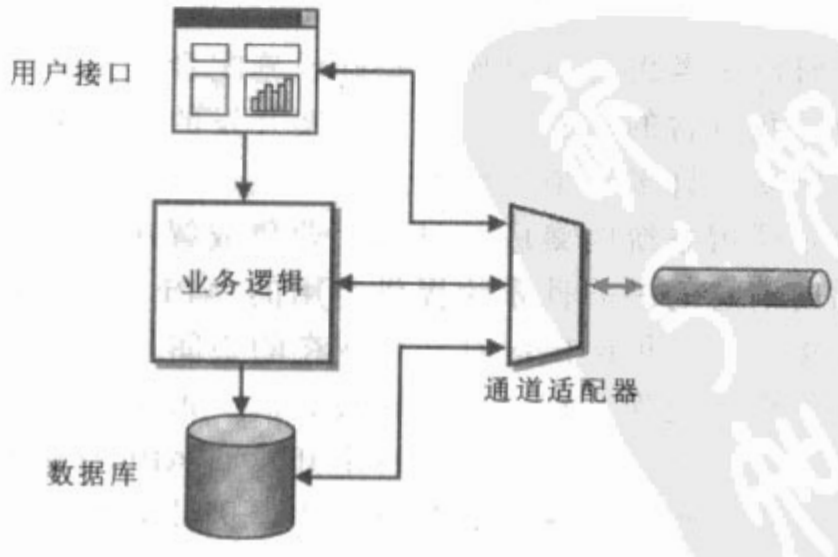
对定制的应用来说，可以在程序中增加发送和接收消息的代码。但是，这会给应用增加额外的复杂性，而且要格外注意，在进行修改时不能引起任何不希望的副作用。这种解决方法还需要开发人员对应用的逻辑和消息传递的 API 都很熟悉。所有这些解决方法实施的前提条件还包括能获得应用的源代码。如果我们所面对的是从第三方软件开发商那里购买的封装好的程序，很可能不具有修改程序代码的条件。

使用通道适配器可访问应用的 API 或数据，并基于此数据把消息发布到通道中；反之，也可接收消息并调用应用内部的功能。



对消息传递系统来说，适配器扮演了消息传递客户的角色，它通过应用提供的接口调用程序的功能。同样，通道适配器也能监听应用的事件并调用消息传递系统对这些事件作出响应。采用这种方式，只要采用了正确的通道适配器，任何应用都能连接到消息传递系统，并能与其他应用集成。

通道适配器可连接到应用体系结构的不同层，这要根据体系结构和消息传递系统需要访问的数据类型决定。



连接到应用不同层的通道适配器

1. **用户接口适配器**。有时也被戏称为“屏幕刷”，这些适配器类型可有效地应用于许多环境中。例如，实现应用的平台可能不支持消息传递系统。或者应用的所有者对集成的支持不感兴趣。这样就不可能在应用所在的平台上运行通道适配器。但是，用户接口往往可在别的计算机和平台上运行（如 3270 终端）。而且，基于 Web 的瘦客户端体系结构的大量涌现已经重新唤起了用户接口的集成。基于 HTML 的用户接口可以很方便地提出 HTTP 请求并解析出结果。用户接口集成的另一项优点是它不需要直接访问应用的内部。在有些情况下，开放系统内部的功能给集成解决方案是不适合，或是不可能的。使用用户接口适配器，其他应用可像常规用户那样获得对应用的相同访问权。用户接口适配器的缺点是它容易失败，并且效率较低。应用必须解析“用户”的输入，并通过屏幕作出响应，而通道适配器再把屏幕的信息重新解析成原始的数据。这一过程包含了很多不必要的操作步骤，因此速度较慢。而且，用户接口被修改的频率要比核心的应用逻辑高。每次修改了用户接口，通道适配器也要随之修改。

2. **业务逻辑适配器**。大多数商业应用都采用 API 提供核心的功能。这种接口可能是一个组件集（如 EJB、COM 对象、CORBA 组件）或一个直接编程 API 库（如 C++或 Java 库）提供的。由于软件开发商（或开发人员）明确地提供了能被其他应用访问的 API，因此它们要比用户接口更稳定。在大多数情况下，访问 API 效率更高。总之，如果应用提供了良定义的 API，那么这种类型的通道适配器可能是最佳的选择。

3. **数据库适配器**。大多数商业应用把数据持久地保存在一个关系数据库中。由于信息已经在数据库中，通道适配器可直接从数据库中提取信息，甚至不需要通知应用，这是一种最不会干扰应用的集成方式。通道适配器甚至能给相关的表添加一个触发器，每当这些表中的数据发生改变时，就发送消息。这种类型的通道适配器效率非常高，而且很通用，这是因为只有两到三家数据库开发商统治了关系数据库的市场。这就使我们能把许多不同的应用通过相对通用的适配器连接起来。数据库适配器的不足是我们对应用的内部了解太多。如果只是读取数据，可能不会有什么风险，但是如果直接更新数据库，可能非常危险。而且，许多应用开发商没有公开发布其数据库模式（Schema），也就是说他们保留了任意修改规范的权利，这就容易导致数据库适配器解决方案的失败。

通道适配器的主要限制是它们能把消息转换为应用的功能，但是它们要求消息的格式必须与被适配的组件的具体实现非常相似。例如，数据库适配器一般要求输入消息的消息字段名要与应用数据库中的表合字段的名称相同。这种消息格式完全由应用的内部结构确定，当与其他应用集成时，它并不是一种好的消息格式。因此，大多数通道适配器必须与消息转换器结合，以把应用相关的消息转换成与规范数据模型兼容的消息格式。

通道适配器往往运行在与应用或数据库所在的计算机不同的计算机上。通道适配器可通过诸如 HTTP 或 ODBC 等协议连接到应用逻辑或数据库。尽管这种设置可避免在应用或数据库服务器上安装额外的软件，这些协议也无法提供像消息传递通道（如可靠传输通道）那样的服务质量。因此我们必须清楚，对数据库的远程连接可能表示它是潜在的故障点。

有些通道适配器是单向的。例如，如果通道适配器通过 HTTP 连接到一个应用，它可能只能消费消息并调用应用的功能，但是不能检测出应用数据内部的变化，除非采用循环

查询的方式，这样做是很低效的。

另一种有趣的通道适配器是元数据（Metadata）适配器，有时也被称为设计时（Design-Time）适配器。这种类型的适配器不调用应用的功能，而是提取其元数据，这些数据描述了应用的内部数据格式。这种元数据可被用于配置消息转换器，或用于检测应用数据格式的变化（见第 8 章“消息转换”）。许多应用接口支持对元数据的这种提取。例如，大多数商业数据库提供了包含元数据的系统表集合，它们描述了应用的表格形式。同样，大多数组件框架（如 J2EE、.NET）提供了专门的“反射”功能，允许组件枚举由其他组件提供的方法。

通道适配器的特殊形式是消息传递桥（Messaging Bridge）。消息传递桥把消息传递系统与其他消息传递系统而不是特定的应用连接起来。通道适配器一般被实现为面向业务过程的客户，以确保适配器的每项操作都能同时在消息传递系统与另一个所采用的系统中成功。

例： 股票交易

股票交易系统可能希望在数据库表中保留所有股票价格的记录。消息传递系统可采用一个关系数据库适配器把通道中的每条消息记录到一个指定的表和数据库模式中。这种通道到 RDBMS（channel-to-RDBMS）的适配器就是一种通道适配器。系统可能还要能从 Internet（TCP/IP 或 HTTP）接收外部股价请求，并采用内部的股价请求格式把它们发送到其内部的股价请求通道上。这种 Internet 到通道的适配器也是一种通道适配器。

例： 商业 EAI 工具

商业 EAI 开发商把一组通道适配器作为他们所提供的产品的一部分。为所有主要的应用包提供适配器能极大简化集成解决方案的开发。大多数开发商还提供了更通用的数据库适配器以及开发定制适配器的软件开发工具（software development kit，SDK）。

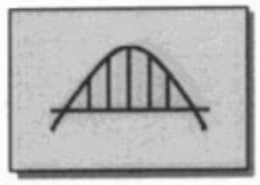
例： 遗留平台适配器

许多开发商提供了从公共消息传递系统到运行在 UNIX、MVS、OS/2、AS/400、Unisys 和 VMS 等平台上的遗留系统的适配器。这些适配器大多数是专门针对某种消息传递系统的。例如，Envoy 技术公司的 EnvoyMQ 是一种能采用 MSMQ 连接许多遗留平台的通道适配器。它由运行在遗留计算机上的客户端组件和运行在具有 MSMQ 的 Windows 计算机上的服务器组件构成。

例： Web 服务适配器

许多消息传递系统提供了通道适配器，以在 HTTP 传送系统和消息传递系统之间传递 SOAP 消息。采用这种方式，SOAP 消息可通过可靠的、异步的消息传递系统在内部网络上传递，并能通过 HTTP 在全球 Internet 上传递（可穿过防火墙）。这种适配器的一个例子是 IBM WebSphere 应用服务器的 Web 服务网关。

4.9 消息传递桥



企业使用消息传递实现应用的通信。但是，如果企业使用了多个消息传递系统，应用很难确定应该连接到哪个消息传递系统。

怎样把多个消息传递系统连接起来，使一个系统上的消息也能到达其他系统？

企业经常会遇到使用多个消息传递系统带来的一个常见问题。当需要把两个不同公司的应用合并起来，而这两家公司采用了不同的消息传递产品时，就会遇到这样的问题。有的时候，同一家企业使用一种消息传递系统集成其大型主机/遗留系统，却使用另一种消息传递系统集成它的 J2EE 或.NET Web 应用服务器，因此它需要把这两种消息传递系统集成起来。另一种常见的情况是，某个应用被集成到多个企业中，例如一个 B2B 的客户想要成为多个拍卖系统的竞标者。如果不同的拍卖集成系统使用不同的消息传递系统，那么企业中的竞标者应用必须把来自好几个外部消息传递系统的消息集中到同一个内部消息传递系统上。另一个例子是，一家具有很多消息传递通道和消息端点的超大型企业可能需要多个消息传递系统运行实例，这些实例必须被连接起来。

如果在某个消息传递系统上的消息与使用其他消息传递系统的应用无关，那么该消息传递系统可完全保持独立。但是，由于应用是同一企业的一部分，使用某些消息传递系统的应用往往对使用另一种消息传递系统传递的消息感兴趣。

人们常常会误以为采用标准化的消息传递 API（如 JMS）就能解决这个问题，其实不然。如果是两种不同的消息传递系统，JMS 只能给客户应用提供相同的接口，但是它无法将这两种消息传递系统合并起来。为了让不同的消息传递系统一同工作，它们必须是可互操作的，即使用相同的消息格式并能以相同的方式将消息从一个消息存储库传送给下一个消息存储库。由不同开发商提供的消息传递系统很少是能互操作的，来自某个开发商的消息存储库只能与来自相同开发商的消息存储库协同工作。

企业中的每个应用都可选择为企业中的每个消息传递系统分别实现一个客户端，但是这会增加消息传递层的复杂性和重复性。如果企业需要增加另一个消息传递系统，那么所有的应用都需要修改，可见这种冗余特性是非常明显的。反之，每个应用可选择只与一个消息传递系统交互，而忽略其他消息传递系统的数据。这样将使应用更简单，但是会忽略大量的企业数据。我们所需要的是让采用某个消息传递系统的应用也能够获得它感兴趣的、位于另一个消息传递系统上的消息。

使用消息传递桥，它连接两个消息传递系统，并在系统之间复制消息。



要想把两个完整的消息传递系统连接起来，一般没有可行的方法，因此我们把两个消息传递系统之间独立的、对应的通道连接起来。消息传递桥是一组通道适配器，其中非消息传递客户实际上是另一个消息传递系统，而每对适配器连接一对相应的通道。桥实现了从一个通道集到另一个通道集的映射，并把一个系统的消息格式转换为另一个系统的消息格式。被连接的通道可用于在消息传递系统的传统客户间传送消息，或者严格地说，是为其他消息传递系统传送需要的消息。

你可能要为自己的企业亲自实现消息传递桥。桥是一种专用的消息端点应用，它同时是两种消息传递系统的客户。当消息被发送到某个消息传递系统的指定通道后，桥消费这条消息，并向另一个消息传递系统的相应通道发送另一个具有相同内容的消息。

许多消息传递系统开发商提供了与其他开发商的消息传递系统建立桥的附加产品。因此，你可以购买已有的解决方案而不必自己开发。

如果其他的“消息传递系统”采用的是更简单的协议，如 HTTP，那么可采用通道适配器模式实现消息传递系统的集成。

由于不同消息传递系统的实现具有各自不同的消息表示方法和从一个消息存储库向下一个消息存储库转发消息的方法，因此采用消息传递桥是必要的。Web 服务可作为一种集成标准，这样的话，即使安装了两个来自不同开发商的消息传递系统，也可使用 Web 服务标准传送消息，使它们集成为一个消息传递系统。请参考第 14 章“结束语”中对“WS-Reliability（WS 可靠性）”和“WS-ReliableMessaging（WS 可靠消息传递）”的介绍。

例： 股票交易

证券交易所采用一种消息传递系统实现各办公室应用的相互通信。银行使用另一种不同的消息传递系统实现各分支机构应用的相互通信。如果交易所和银行决定合并为一家公司，提供银行账户和投资服务，那么合并的公司应该采用哪种消息传递系统呢？与其重新设计公司中一半的应用，使它们能使用新的消息传递系统，还不如使用消息传递桥把两种消息传递系统连接起来。采用这种方式，银行应用和证券交易应用可协作实现存款账户与有价证券账户之间的资金转移。

例： MSMQ 桥

MSMQ 基于连接器服务器定义了一种体系结构，该服务器能使连接器应用通过其他的

(非 MSMQ)消息传递系统发送和接收消息。使用连接器服务器的 MSMQ 应用可像对 MSMQ 通道一样，对来自其他消息传递系统的通道执行相同的操作[Dickman]。

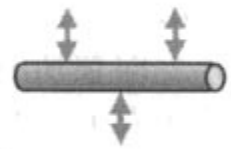
Microsoft 的主机集成服务器（Host Integration Server）产品包含一个 MSMQ-MQSeries 桥的服务，它能把两个消息传递系统集成起来。这样的话，MSMQ 应用可通过 MQSeries 通道发送消息，反之亦然，使两个消息传递系统看起来像一个。

MSMQ-MQSeries 桥的版权所有者 Envoy 技术公司还提供了一个相关的名为 Envoy Connect 的产品。它采用运行在非 Windows 平台，特别是 J2EE 平台上的消息传递服务器连接 MSMQ 和 BizTalk 服务器，把企业中的 J2EE 和.NET 消息传递合并起来。

例：SonicMQ 桥

Sonic 软件公司的 SonicMQ 提供了能支持 IBM MQSeries、TIBCO、TIB/Rendezvous 和 JMS 的 SonicMQ 桥产品。这使得位于 Sonic 通道上的消息能被传送到其他消息传递系统的通道上。

4.10 消息总线



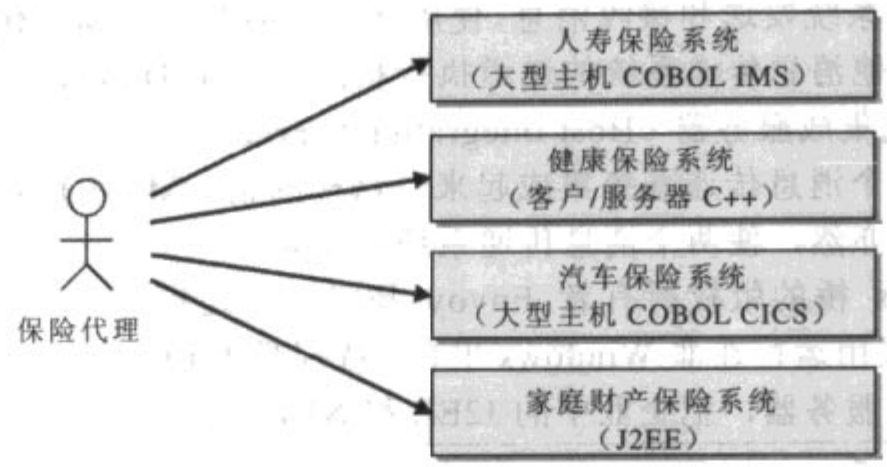
一个企业包含好几个已有的系统，它们必须采用统一的方式共享数据和操作，响应一组公共的业务请求。

应该采用哪种体系结构才能使单独的应用协同工作，但是又彼此解耦合，使得增加或修改某个应用不会影响其他应用？

企业常常包含大量独立运行的不同应用，但是要以一种统一的方式协作。企业应用集成（Enterprise Application Integration，EAI）为此提供了解决方案，但是并没有描述如何实现它。

例如，考虑一家销售不同种类（人寿、健康、汽车、家庭等）保险产品的保险公司。由于公司合并和 IT 开发技术的发展变化，企业包含许多独立的应用，它们分别管理公司的各种产品。一位极力向客户兜售好几类保险策略的保险代理必须为查询每种策略登录到不同的系统上，这不但浪费精力，还会增加出错的机会。

保险代理需要一个统一的、能为客户提供所有保险策略的应用。其他类型的保险公司雇员，如索赔申请仲裁员和客户服务代表，不但需要自己的应用能处理保险产品，还希望它们能提供统一的界面。每项产品的应用要能协同工作，当所购买的产品多于一种时，还能提供一定的折扣，并能处理涉及多个产品的索赔要求。



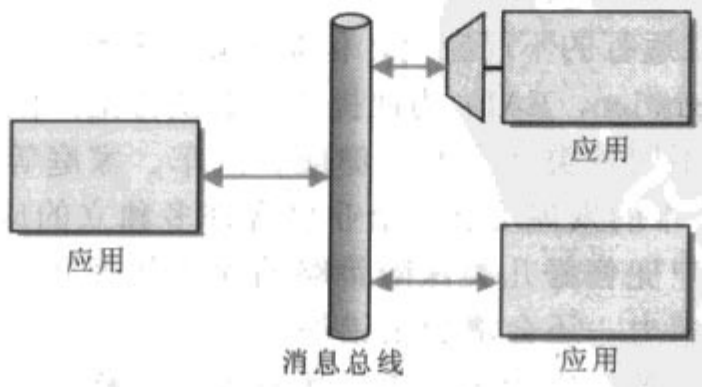
保险公司 EAI 场景

IT 部门可重写产品应用，让它们使用相同的技术并能协同工作，但是替换已经运行的系统（即使它们无法协同）所需的时间和金钱是不能得到满足的。IT 部门可为保险代理创建一个统一的应用，但是这个应用需要连接实际管理保险策略的系统。因此，这个新的应用不但不能统一原来的系统，反而多创建了一个不能与其他系统集成的系统。

保险代理应用可与其他所有的系统集成，但是这样做更复杂。同样的复杂性也会出现在索赔仲裁员和客户服务代表的应用中。此外，这些统一的用户应用并不能帮助实现产品应用的相互集成。

即使所有这些应用能协同工作，对企业配置的任何改动会让它们全部停止下来。并不是所有的应用都能始终运行，要尽量减少其他未运行程序对正在运行程序的影响，使它们能继续工作。随着时间的推移，需要为企业增加新的应用或删除一些程序，要尽量减少它们对其他程序的影响。我们需要的集成体系结构应该能让产品应用以松耦合的方式协同工作，而且能让用户应用与它们集成。

在这些应用之间构造连接中间件，形成消息总线，使应用能通过消息传递协同工作。



消息总线是由规范数据模型、公共命令集和消息传递基础设施组成的，能让不同系统通过共享的接口集通信。它类似于计算机系统中的通信总线，是 CPU、主存和外设之间通

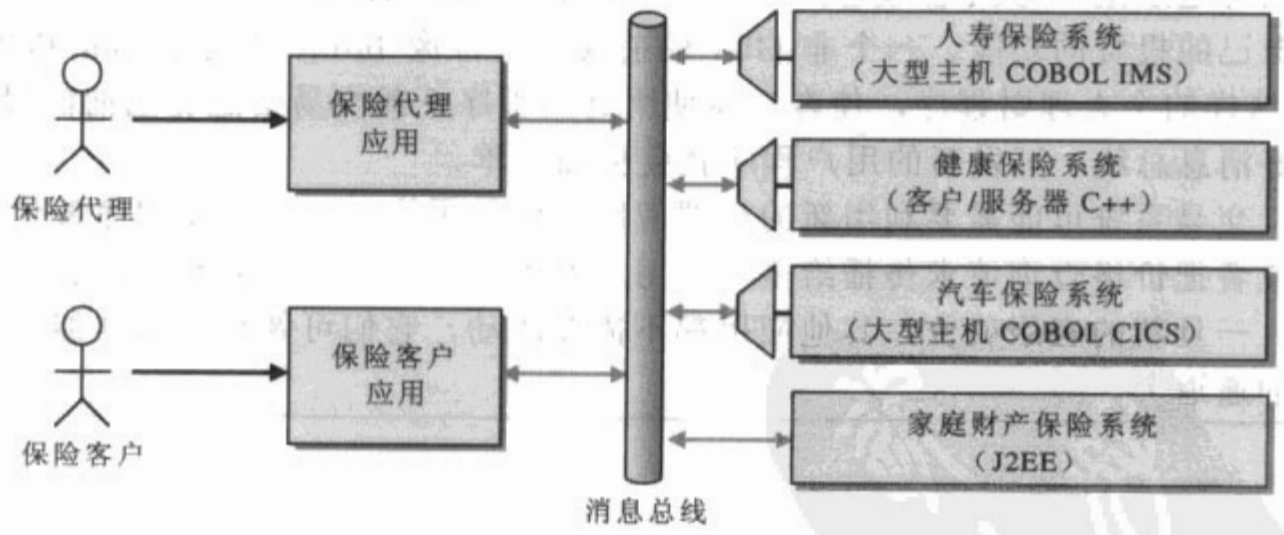
信的焦点。和硬件总线一样，消息总线也是由许多部件组合而成的。

1. **公共通信基础设施**——就像 PCI 总线的物理插脚和连线为 PC（个人计算机）提供了公共的、周知的物理基础设施一样，消息总线中的公共基础设施实现了相同的目标。消息传递系统一般被用作实际的通信基础设施，在应用之间提供一个跨平台、跨语言的通用适配器。该基础设施可能包含了消息路由器的功能，实现系统到系统的正确消息路由。通常它还使用发布-订购通道把消息发送给所有的接收者。

2. **适配器**——不同的系统必须提供与消息总线交互的方法。有些应用本身就支持与总线的连接，但是大多数应用需要通过适配器连接到消息传递系统。这些适配器一般是商业或定制的通道适配器及服务激励器（Service Activator）。一些专门的任务，如采用适当的参数调用 CICS 处理事务，或者把总线的通用数据结构转换为应用所用的特定形式等，是由这些适配器完成的。与总线连接还要求所有系统必须达成一致的规范数据模型。

3. **公共命令结构**——就像 PC 体系结构采用公共的命令集表示物理总线上各种不同的操作（从某个地址读字节，向某个地址写字节）一样，消息总线也要为所有参与者提供大家都能理解的公共命令。采用命令消息（Command Message）可实现这种特性。数据类型通道是另一种常用的实现方法，其中消息路由器能确定如何把特定的消息（像购货单）路由给特定的端点。消息总线与物理总线的最终区别在于消息总线携带的消息类别要比物理总线携带的“读/写”类消息划分得更细。

在我们的 EAI 实例中，消息总线可作为各种保险系统之间的通用连接器，也可作为客户应用与保险系统连接的通用接口。



保险公司消息总线

系统中有两个只了解消息总线的图形用户界面（GUI）程序，它们对后台系统的复杂性一无所知。总线负责把命令消息路由给正确的后台系统。在某些情况下，处理命令消息的最佳方式是构建一个到系统的适配器，把命令转换为系统理解的方式后再传递给它（例如，调用一个 CICS 事务处理或调用一个 C++ API）。在其他情况下，可以在已有的系统中直接构建命令处理逻辑，而不是调用当前的逻辑。

为代理 GUI 开发的消息总线是很容易复用在其他 GUI 中的，比如为索赔仲裁员、客

户服务代表和使用 Web 接口浏览客户账户的应用。这些 GUI 应用的特性和安全控制需求是不同的，但是它们都需要与后端的应用协作。

消息总线为企业形成一个简单有用的面向服务的体系结构。每个服务至少有一个能接收请求的请求通道，这些请求采用了共同协商的格式；另外还可能要有一个能支持指定应答格式的应答通道。任何参与集成的应用通过发送请求并等待应答使用这些服务。实际上，请求通道相当于可用服务的目录。

消息总线要求所有使用总线的应用使用相同的规范数据模型。把消息发送到总线上的应用必须依靠消息路由器把消息路由给最终的目标。那些在设计时没有考虑消息传递系统接口的应用可能需要通道适配器和服务激励器。

例：股票交易

股票交易系统可能希望提供一套统一的服务，包括股票交易、债券拍卖、股价查询、投资管理等。这需要好几种独立的后端系统相互配合。为了向前端客户 GUI 提供统一的服务，系统可采用中间应用提供所有这些服务，并把实际的操作委派给后端系统完成。后端系统还能通过这个中间应用协调。但是，中间应用可能成为系统的瓶颈和单一故障点。

与中间应用相比，采用消息总线是更好的选择，它具有能请求各种服务并获得响应的通道。该总线还能使后端系统相互协作。前端系统只需要连接到总线并通过它调用服务。总线可非常方便地分布在多个计算机之间，以提供负载的分布和容错支持。

一旦部署好消息总线，连接前端的 GUI 就很容易了，它们只需从正确的通道收发消息。小额经纪人可通过某个 GUI 管理客户的投资。基于 Web 的 GUI 为客户提供了采用 Web 浏览器管理自己的投资的功能。一个非 GUI 的前端能支持像 Intuit 的 Quicken 和 Microsoft 的 Money 这样的个人理财程序，使客户能使用这些程序下载交易信息和当前的股票价格。一旦部署好消息总线，开发新的用户程序就会更加简单。

同样，交易系统可能需要利用新的后端应用，例如采用另一个交易程序替代当前的交易程序，或者把价格查询请求传播给多个应用。实现这样的改动只需要从消息总线添加或删除应用。一旦新的应用就位，其他应用都不需要改动，它们可像往常那样继续把消息发送到总线的通道上。

第5章

消息构造

5.1 引言

在第3章“消息传递系统”中，我们已经讨论了消息。两个应用想要交换一些数据时，可以把数据包装在消息中来实现数据交换。尽管消息通道不能直接传送原始数据，但是它能传送包装在消息中的数据。创建和发送消息带来了另外几个问题。

消息的目的——消息实际上只不过是一组数据，但是发送者对于接收者处理消息抱有不同的目的。发送者可以发送一个命令消息，指定想在接收方调用的一个函数或方法。实际上就是发送者在告诉接收者应该执行哪些代码。发送者也可以发送一条文档消息，从而将自己的某个数据结构传递给接收者。这就是在向接收者传送数据，但是没有指定接收者应该如何处理这些数据。发送者也可以发送一条事件消息，通知接收者在发送方发生了哪些变化。发送者并没有告诉接收者如何应对，只是提供通知而已。

返回响应——当应用发送消息时，往往希望能收到应答，以确认消息已经得到处理，并提供相应的结果。这就是一种请求/应答的情况。请求一般是一个命令消息，而应答是一个文档消息，其中包含一个结果值或一个异常。请求者应该在请求消息中指定返回地址，告诉应答者使用哪个通道传送应答。请求者可能有多个正在处理的请求，因此应答消息中应该包含一个关联标识符，指定该应答对应哪个请求。

有两种常见的请求/应答情况值得注意，它们都采用命令消息作为请求，采用相应的文档消息作为应答。第一种情况是消息传递RPC，请求者不但想调用应答者提供的函数，还想获得函数的返回值。应用通过消息传递执行RPC（远程过程调用）就是如此。第二种情况是消息传递查询，请求者请求一项查询，应答者执行查询并在应答中返回查询结果。应用使用消息传递执行远程查询就属于这种情况。

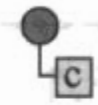
海量数据——有时应用想传输很大的数据结构，这个结构在一个消息中放不下。在这种情况下，可以把消息划分为多个可管理的段，并按消息序列（Message Sequence）来发送。这些数据段必须按顺序发送，而不只是发送一堆消息，这样接收者才能重构得到原始的数据结构。

慢消息——消息传递中存在一个问题，发送者往往不知道接收者需要多长时间才能收到消息。而消息的内容可能是时间敏感的，因此如果消息不能在某个时限之前收到，

这个消息将被忽略并丢掉。在这种情况下，发送者可使用消息到期来指定一个到期时间。如果消息传递系统直到消息到期都无法成功传送这个消息，就应该把它丢掉，或把它转移到一个死文字通道中。类似地，如果接收者在到期时间以后才收到消息，则应该把这个消息丢掉。

总之，只是选择使用消息还不够。当有数据需要传输时，必须通过消息来完成。本章将介绍使用消息传输数据时要考虑的其他一些问题。

5.2 命令消息



应用需要调用由其他应用提供的功能。通常采用的方法是远程过程调用，不过也可以利用消息传递提供的优点。

怎样利用消息传递调用其他应用中的过程？

远程过程调用的优点就是它是同步的，所以调用会立即执行，而调用者的线程阻塞。不过，这也是一个缺点。如果调用不能立即执行——要么是因为网络不通，要么是因为远程进程尚未运行和监听——那么调用就会失败。如果调用是异步的，调用者可以反复尝试调用，直到远程应用中的过程被成功调用。

本地调用要比远程调用更可靠。如果调用者能把过程调用作为消息传送给接收者，接收者就能在本地执行调用。因此，现在的问题是如何把过程调用转换成消息。

要把请求封装成对象，已经有一种现成的模式。命令模式[GoF]展示了如何把请求转换为可存储、可传递的对象。如果这个对象是一个消息，就能通过消息通道来存储和传递。类似地，命令的状态（如方法参数）可以存储为消息的状态。

使用命令消息可靠地调用另一个应用中的过程。



命令并没有特定的消息类型。命令消息只是一个常规的消息，其中恰好包含了一个命

令而已。在 JMS 中，命令消息可以是任何类型的消息，如包含一个 Serializable 命令对象的 ObjectMessage，包含 XML 形式命令的 TextMessage 等。在.NET 中，命令消息是存储有命令的消息。SOAP（Simple Object Access Protocol，简单的对象存取协议）请求就是一个命令消息。

命令消息一般采用点对点通道发送，这样每个命令只能被消费和调用一次。

例：SOAP 和 WSDL

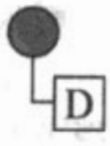
根据 SOAP 协议[SOAP 1.1]和 WSDL 服务描述[WSDL 1.1]，当使用 RPC 形式的 SOAP 消息时，这种请求消息就是命令消息模式的例子。使用时，SOAP 消息体（一个 XML 文档）包含要在接收者应用中调用的方法名以及传递给该方法的参数值。方法名必须与在接收者 WSDL 中定义的某个消息名相同。

下面的例子选自 SOAP 规范中，这里要调用接收者的 GetLastTradePrice 方法，并且有一个名为 symbol 的参数。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
         <symbol>DIS</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

在 SOAP 命令中，可能希望方法名是某个标准<method>元素的值；实际上，方法名就是方法元素的名，并以 m 名字空间作为前缀。通过为每个方法设置单独的 XML 元素类型，这有利于更准确地验证 XML 数据，因为方法元素类型可以指定参数的名、类型和顺序。

5.3 文档消息



应用可能要把数据传输给另一个应用。可以采用文件传输或共享数据库来达到这个目的，但是这些方法都存在缺点。使用消息传递传输数据会更好一些。

如何使用消息传递在应用之间传输数据？

在分布式处理中常常会遇到这样的问题：某个进程拥有另一个进程所需的数据。文件

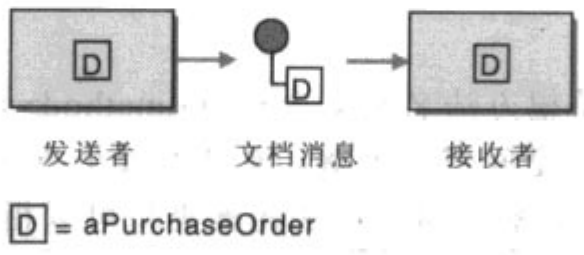
传输使用起来很简单，但是不能很好地协调应用。由一个应用创建的文件，可能需要耽搁很长时间才能被另一个应用读取。如果有多个应用需要读取个文件，那么很难确定该由谁负责删除这个文件。

为了适应数据，共享数据库需要向数据库中增加新的模式，或者硬性要求数据满足现有的数据库模式。数据一旦存储到数据库中，这也会带来风险：以前不应访问这些数据的应用现在则有了访问的机会。有新的数据时，要想触发接收者读取这些数据会很困难，而且有多个接收者（读者）需要读取这些数据时，协调多个读者可能会带来问题，即无法决定应该由谁负责删除数据。

采用远程过程调用也可以发送数据，但是除了发送数据外，调用者通过所调用的过程还会告诉接收者如何处理这些数据。类似地，命令消息也可以传输数据，但是对于接收者如何处理数据，同样存在过于特定的要求。远程过程调用使用的是双向通信，如果我们只想把数据从一个应用传送给另一个应用，双向通信就没有必要。

相比之下，我们更想使用消息传递来传送数据。消息传递比 RPC 更可靠。使用点对点通道可以保证只有一个接收者能获得数据（数据不会重复），使用发布-订购通道可以保证任何需要数据的接收者都能得到数据的副本。因此，使用消息传递的技巧在于如何充分利用其优点，同时尽量不要像 RPC 那样建立消息。

使用文档消息在应用之间可靠地传输数据结构。



命令消息告诉接收者必须调用某个操作，与之不同，文档消息只是向接收者传送数据，让接收者自己来决定应该如何处理这些数据。数据可以是一个数据单元、一个对象或者一个数据，可以分解为更小的单元。

文档消息看上去很像事件消息，它们之间的主要区别在于时间限制和内容两方面。文档消息的关键在于其内容是一个文档。重要的是成功地传送文档；而何时发送和接收文档则不那么重要。可以考虑采用可靠传输；消息到期机制可能不适合。相反，事件消息的存在性和时间限制往往比事件消息的内容更重要。

在消息传递系统中，文档消息可以是任何类型的消息。在 JMS 中，文档消息可能是一个包含有文档相应 Serializable 数据对象的 ObjectMessage，也可能是一个包含 XML 形式数据的 TextMessage。在.NET 中，文档消息是一个存储有数据的消息。SOAP 应答就是一个文档消息。

文档消息一般采用点对点通道发送，可以把文档从一个进程转移到另一个进程，而且

没有重复。采用消息传递可以实现简单的工作流，即把文档传递给一个应用，由这个应用修改该文档，然后再传递给另一个应用。在有些情况下，可以通过发布-订购通道来广播文档消息，但是这样做会产生很多文档副本。文档副本必须是只读的；否则，如果接收者修改了副本，系统中将会存在文档的多个副本，其中每个副本都包含不同的数据。在请求/应答中，应答一般是文档消息，其中结果值就是文档。

例：Java 和 XML

下面的例子（摘自[Graham]中的 XML 模式示例）显示了如何把一个简单的订购单表示为 XML，并使用 JMS 作为消息发送出去。

```
Session session = // 获得会话
Destination dest = // 获得目标
MessageProducer sender = session.createProducer(dest);
String purchaseOrder =
"    <po id=\"48881\" submitted=\"2002-04-23\">
       <shipTo>
          <company>Chocoholics</company>
          <street>2112 North Street</street>
          <city>Cary</city>
          <state>NC</state>
          <postalCode>27522</postalCode>
       </shipTo>
       <order>
          <item sku=\"22211\" quantity=\"40\">
             <description>Bunny, Dark Chocolate, Large</description>
          </item>
       </order>
    </po>";
TextMessage message = session.createTextMessage();
message.setText(purchaseOrder);
sender.send(message);
```

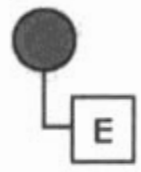
例：SOAP 和 WSDL

根据 SOAP 协议[SOAP 1.1]和 WSDL 服务描述[WSDL 1.1]，当使用文档形式的 SOAP 消息时，这个 SOAP 消息就是文档消息的一个例子。SOAP 消息体是一个 XML 文档（或者是一种已经转换为 XML 文档的数据结构），SOAP 消息把文档从发送者（如客户）传送给接收者（如服务器）。

当使用 RPC 形式的 SOAP 消息传递时，响应消息就是这种模式的一个例子。使用时，SOAP 消息体（一个 XML 文档）包含来自被调方法的返回值。下面这个例子选自 SOAP 规范，会返回调用 GetLastTradePrice 方法后产生的结果。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5.4 事件消息



多个应用想要使用事件通知来协调行动，并且想通过消息传递来通知这些事件。

如何使用消息传递把事件从一个应用传送给另一个应用？

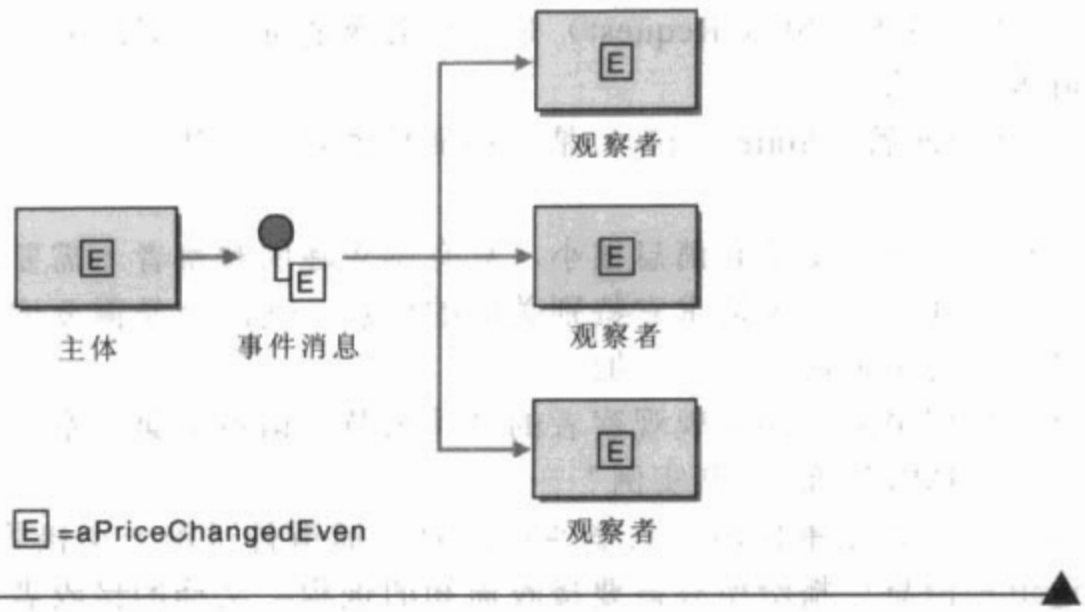
有时，一个对象想要知道发生在另一个对象中的事件。典型的例子是模型-视图-控制器体系结构[POSA]，当模型修改了状态时，必须通知其视图重新绘制。分布式系统中也需要这样的更改通知。例如，在一个 B2B 系统中，某商家需要把价格更改信息或一个新的产品目录通知给其他商家。

进程可以使用远程过程调用来通知其他应用出现了更改事件，但是这就要求接收者必须立即接受事件，即使它现在并不需要这个事件。RPC 还要求宣布事件的进程必须知道每一个监听进程，并在每个监听进程上完成远程过程调用。

观察者模式[GoF]描述了如何设计宣布事件的主体以及消费事件的观察者。主体通过调用观察者的 Update()方法把事件通知给观察者。Update()可以实现为远程过程调用，但是这样就会存在远程过程调用的所有缺点。

更好的方法是把事件通知作为消息异步地发送。采用这种方式，主体可以在准备好的情况下发出通知，各个观察者则可以在准备好的情况下接收通知。

使用事件消息实现应用间可靠的异步事件通知。



当主体需要通知事件时，它会创建一个事件对象，把这个对象封装在消息中，然后作为一个事件消息发送到通道中。观察者接收事件消息、获取事件，并处理事件。消息传递不会修改所宣布的事件，只是确保通知能到达观察者。

在消息传递系统中，可以采用任何类型的消息表示事件消息。在 Java 中，事件可以是一个对象或数据，如一个 XML 文档。因此，可以作为 ObjectMessage、TextMessage 等消息通过 JMS 传送。在.NET 中，事件消息就是一个存储有事件的消息。

事件消息与文档消息的区别主要在时间限制和内容两方面。事件的内容一般并不是很重要。许多事件的消息体甚至都是空的，只要出现了事件就能让观察者作出反应。事件的时间限制非常重要：一旦有所修改，主体应该立即发出事件，观察者应该趁事件有效时尽快处理事件。可靠传输对事件传送的意义不大，因为事件发生的频率很快，而且需要尽快地传送。消息到期有助于保证事件得到快速处理，或者根本不处理。

例如，B2B 系统必须把价格和产品方面的变更通知其他商家，为此可以使用事件消息、文档消息或这两种消息的组合。如果一个消息指出计算机硬盘的价格有所变动，这就是一个事件。如果消息提供了硬盘的信息，还包括有硬盘的新价格，那么这就是一个作为事件发送的文档。如果另一个消息宣布了新的产品目录及其 URL，那么这是一个事件，反之，如果类似的一个消息具体包含了这个新目录，它就是一个包含文档的事件。

到底哪种形式更好？观察者模式把这一问题描述为推模型和拉模型的权衡选择。推模型把修改的有关信息作为更新的一部分全数发送，而拉模型只发送最少的信息，而需要更多信息的观察者可以向主体发送 GetState()请求。与消息传递有关的两种模式为：

推模型（push model）——消息是文档/事件消息的组合；消息的传送就宣布了有状态发生，而且消息的内容就是新的状态。如果所有观察者都需要这些详细信息，推模型效率更高。否则在两个方面都不讨好，不仅会频繁发送很大的消息，而且许多观察者通常都会忽略这些大消息。

拉模型（pull model）——有三种消息采用了这种模型：

1. **更新**（Update）是一种事件消息，它向观察者通知发生了一个事件。

2. **状态请求**（State Request）是一种命令消息，感兴趣的观察者使用这个消息从主体那里请求详细的信息。

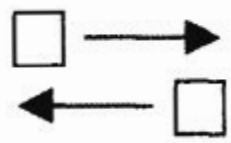
3. **状态应答**（State Reply）是一种文档消息，主体使用这个消息向观察者发送详细的信息。

拉模型的优点是更新消息很小，只有感兴趣的观察者才需要请求详细的信息，而且每个感兴趣的观察者只需请求它特别关注的信息。其缺点是需要增加额外的通道，且由于消息不只一个会带来通信量的增加。

有关使用消息传递实现观察者的具体细节，请参见第6章“中场演练：简单的消息传递”中的“JMS发布-订购实例”一节。

没有理由限制事件消息只能通过点对点通道传送给一个接收者，通常可以采用发布-订购通道把消息广播给所有需要接收通知的进程。文档的接收者必须消费文档消息，以防止文档消息的丢失，而事件消息的接收者在处理不过来的时候可以忽略一些事件，因此事件消息的订购者一般是非持久的(不是持久的订购者)。在使用消息传递实现观察者模式时，事件消息是一个很关键的部分。

5.5 请求/应答



当两个应用通过消息传递进行通信时，通信是单向的。应用可能需要双向的交互。

当应用发送一个消息后，如何才能获得接收者返回的响应？

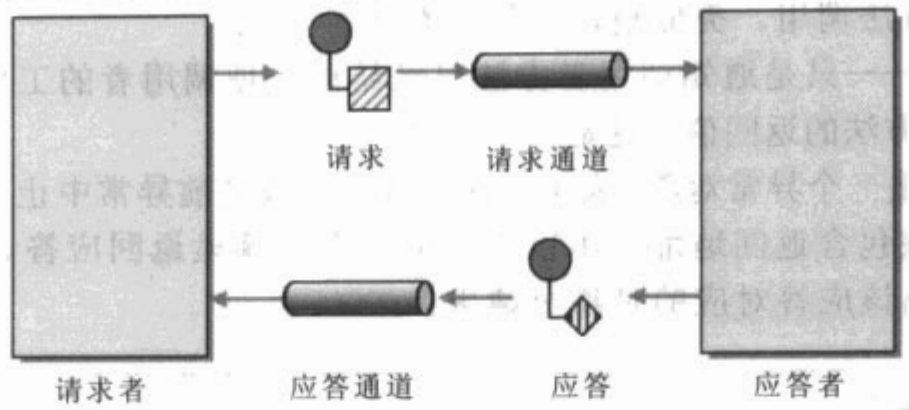
消息传递在应用之间提供了单向通信。消息沿消息通道单向传输，从发送者转移到接收者。这种异步传送方式使得消息传送更加可靠，并且使发送者与接收者得以解耦合。

问题是组件之间的通信往往需要是双向的。当某程序调用一个函数时，它要接收一个返回值。在执行一项查询时，它要接收查询的结果。一个组件向另一个组件发送修改通知时，可能需要获得应答。如何实现双向的消息传递呢？

也许可以让发送者和接收者同时共享一个消息。然后，每个应用都能向消息中添加信息，提供给对方消费。但是这不符合消息传递的工作原理。因为消息是先发送、再接收，因此发送者和接收者不可能同时访问消息。

也许可以让发送者保留对消息的引用。一旦接收者把响应放到消息中，发送者就能把消息“拉”回。如果是夹在晾衣绳上的便条，可以采用这种做法，但是消息通道却不是这样工作的。通道只在一个方向上传送消息。我们真正需要的是双向通道上的双向消息。

发送一对请求/应答消息，每个消息都使用自己的通道。



请求/应答由两部分组成：

1. **请求者**发送请求消息并等待应答消息。

2. **应答者**接收请求消息并用应答消息做出响应。

请求通道可以采用点对点通道或发布-订购通道。这取决于请求是否需要广播给所有感兴趣的各方，还是只能由一个消费者处理。另一方面，应答通道几乎总采用点对点通道，因为广播应答一般没有什么意义，应答应该只返回给请求者。

调用者执行远程过程调用时，调用者的线程必须在等待响应时阻塞。采用请求/应答消息，请求者有两种接收应答的方法。

1. **同步阻塞**——调用者中的一个线程发送请求消息，阻塞（像轮询消费者一样）并等待应答消息，然后处理应答。它的实现很简单，但是如果请求者崩溃，要想重建阻塞线程是很困难的。请求线程要等待响应，这意味着只有一个未完成的请求，或者该请求的应答通道是这个线程专用的。

2. **异步回调**——调用者中的一个线程发送请求消息，并为应答建立一个回调函数。另一个线程负责监听应答消息。当应答消息到达时，应答线程会调用相应的回调函数，回调函数将重新建立调用者的上下文并处理应答。这种方法支持多个未完成的请求共享相同的应答通道和应答线程，这个线程可以为多个请求线程处理应答。如果请求者崩溃，只需简单地重启应答线程就能恢复。但是，这里也增加了一定的复杂性，因为回调机制必须重建调用者上下文。

仅仅有能相互发送请求和应答的两个应用并没有太大意义，真正有意思的是请求和应答消息分别表示什么。

1. **消息传递 RPC**——这就是要使用消息传递来实现远程过程调用。请求是一个命令消息，它描述了应答者应该调用的函数。应答是一个文档消息，它包含了函数的返回值或异常信息。

2. **消息传递查询**——这就是使用消息传递来实现远程查询。请求是一个命令消息，其中包含查询。应答是查询的结果，可能是一个消息序列。

3. **通知/确认**——由此可以使用消息传递提供带确认的事件通知。请求是一个事件消

息，可以提供通知。应答是一个文档消息，对通知作出确认。确认本身也可以是另一个请求，如查询事件的详情。

请求就像一个方法调用。类似地，应答也有3种可能：

1. **空**（Void）——只是通知调用者方法已经结束，使调用者的工作可以继续。
2. **结果值**——方法的返回值，这是一个对象。
3. **异常**——这是一个异常对象，表示方法在成功完成之前异常中止，并给出异常的原因。

请求消息中应该包含返回地址，用以告诉应答者向哪里返回应答。应答消息中应该包含关联标识符，指出该应答对应的是哪个请求。

例：SOAP 1.1 消息

SOAP消息以请求/应答对的形式出现。SOAP请求消息指示了发送者想要调用接收者提供的那个服务。SOAP响应消息包含服务调用的结果。响应消息要么包含一个结果值，要么包含一个故障（fault）——在SOAP中，故障等价于异常[SOAP1.1]。

例：SOAP 1.2 响应消息交换模式

SOAP1.1没有明确规定响应消息，SOAP1.2介绍了一种明确的请求/响应消息交换模式[SOAP1.2 第2部分]。根据该模式的描述，对于一个SOAP请求，有一个单独的、可能异步的响应。

例：JMS 请求者对象

JMS包含一对可以用于实现请求/应答的特性。

TemporaryQueue是一个队列，可以在程序中创建，用于创建这个队列的连接（Connection）一旦断开，该队列也不再有效。只有由同一个连接创建的MessageConsumer才能从队列中读取消息，因此队列对连接来说是私有的[JMS1.1]。

怎样才能让MessageProducer知道新创建了这样一个私有的队列呢？请求者会创建一个临时队列，并在请求消息的reply-to属性中设置这个队列（见“返回地址”）。正常的应答者将把应答返回给指定的队列，如果该队列不是请求消息的一个属性，那么应答者将无从知道。通过这种简单的方法，请求者可以确保总能接收到返回的应答。

采用临时队列的不足是，当它们的连接关闭时，队列以及队列中的所有消息将被删除。同样的，临时队列不能提供可靠传输。如果消息传递系统崩溃，连接将会丢失，队列和队列中的消息也会丢失。

JMS还提供了QueueRequestor，这是一个能发送请求和接收应答的简单类。请求者包含一个QueueSender来发送请求，并且利用一个QueueReceiver接收应答。每个请求者会创建自己的临时队列以接收应答，并把该队列的地址设置在请求消息的reply-to属性中[JMS1.1]。请求者发送请求和接收应答的代码非常简单：

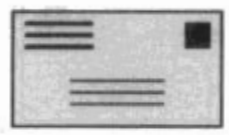
```
QueueConnection connection = // 获得连接
Queue requestQueue = // 获得队列
```

```
Message request = // 创建请求消息
QueueSession session = connection.createQueueSession(false,
 Session.AUTO_ACKNOWLEDGE);
QueueRequestor requestor = new QueueRequestor(session,
 requestQueue );
Message reply = requestor.request(request);
```

request 方法负责发送请求消息，这个方法会阻塞，直到接收到应答消息为止。

QueueRequestor 使用的 TemporaryQueue 是一个点对点通道。相应的发布-订购通道是 TemporaryTopic 和 TopicRequestor。

5.6 返回地址

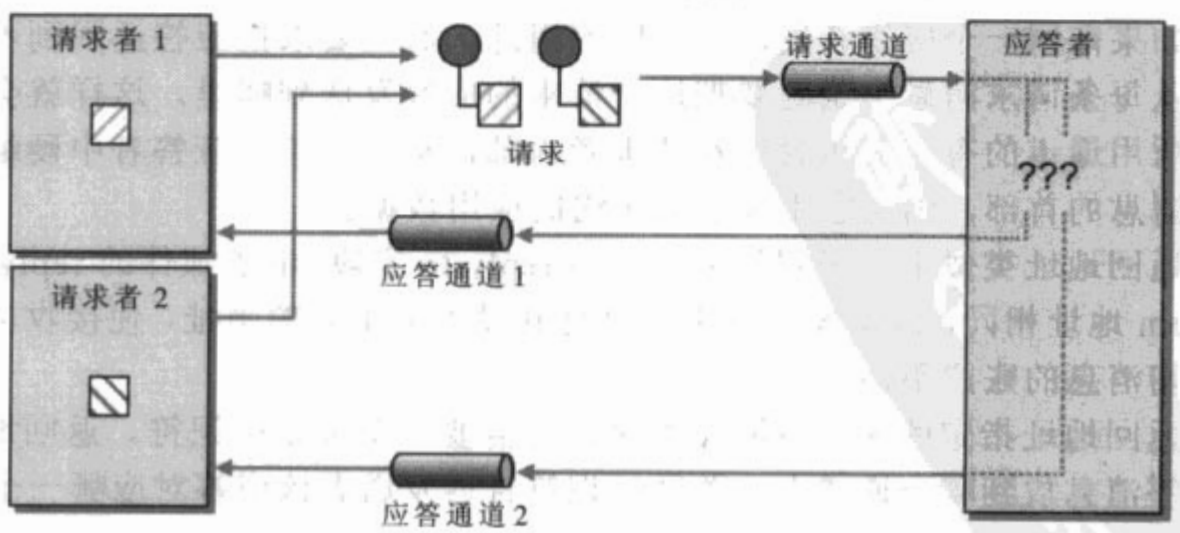


应用使用消息传递完成请求/应答。

应答者如何知道将应答发送到哪里？

人们一般认为消息是完全独立的，在任何时刻，任何发送者都能把消息发送到任何通道上。但实际上，消息往往存在着关联。对于请求/应答对，这两个消息看起来是独立的，但是应答消息与导致此应答的请求消息是一一对应的。因此，处理请求消息的应答者不能简单地将应答消息发送到任意的通道上，它必须把消息发送到请求者期望的通道上。

每个接收者可以自动地知道将应答发送到哪个通道，但是如果采用这种硬编码的方式，会造成软件灵活性的下降，增加维护的难度。此外，单个应答者可能要处理来自多个不同请求者的调用，因此每个消息的应答通道并不是固定的，这要由发送消息的请求者来决定。



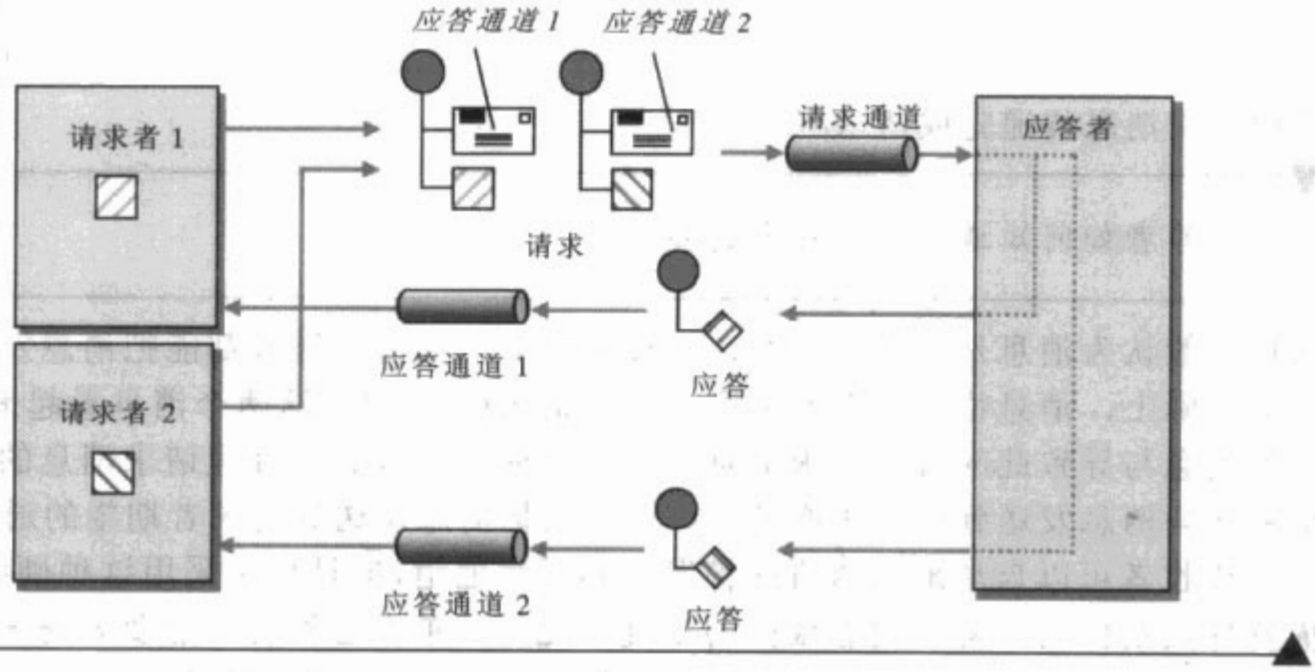
不能确定把应答发送到哪里

请求者可能不希望把应答送回给自己。实际上，它也许提供了一个相关的回调处理程

序专门处理应答。回调处理程序与请求者可能监控不同的通道（或者，请求者根本不监控任何通道）。请求者可以使用多个回调处理程序，对于同一个请求者发送的不同请求，相应的应答可以发送给不同的回调处理程序。

应答通道不一定要将应答返回给请求者；它可以将应答传送给请求者指定的某个处理程序，因为处理程序在监听请求者指定的通道。因此，只是知道请求是由哪个请求者发送的，或者知道了发送请求的通道，这并不一定会告诉应答者应该把应答发送到哪个通道。即使确实把应答通道告诉了应答者，应答者还是要自行推断对应某个特定的请求者或请求通道应该使用哪一个应答通道。在请求中显示地指定应该使用的应答通道更简单。这样只需要请求者告诉应答者应该把应答返回到哪里，以及如何返回应答。

请求消息中应该包含返回地址，它指出应该把应答消息发送到哪里。



采用这种方法，应答者不必知道把应答发送到哪里，它可以从请求消息中获得应答通道的地址。如果向同一个应答者发送了不同的消息，而且要求把应答返回到不同的位置，应答者可以从每条请求消息中确定要把这个请求的应答发送到哪里。这样就会把发送请求和应答消息所用通道的有关知识封装在请求者内部，从而避免在应答者中硬编码写入。返回地址放在消息的首部，因为它不属于所传送的应用数据。

消息的返回地址类似于电子邮件消息中的 reply-to 字段。电子邮件的 reply-to 地址一般与邮件的 from 地址相同，但是发送者也可以将其设置为不同的地址，使接收应答的账户与用来发送最初消息的账户不同。

当通过返回地址指定的通道返回应答时，还需要一个关联标识符。返回地址告诉接收者应该把应答消息放到哪个通道上，关联标识符告诉发送者该应答对应哪一个请求。

例：JMS 的 Reply-To 属性

JMS 消息为返回地址提供了一个预定义的属性：JMSReplyTo。它的类型是一个

Destination（Topic 或 Queue），而不只是一个表示目标名的字符串，因此能确保目标（如消息通道）真正存在，至少在请求发送时确实存在[JMS1.1]、[Monson-Haefel]。

发送者希望指定一个 Queue 作为应答通道时，可以采用以下代码：

```
Queue requestQueue = // 指定请求目标
Queue replyQueue = // 指定应答目标
Message requestMessage = // 创建请求消息
requestMessage.setJMSReplyTo(replyQueue);
MessageProducer requestSender = session.createProducer(requestQueue);
requestSender.send(requestMessage);
```

然后，接收者可以如下发送应答消息：

```
[View full width]
Queue requestQueue = // 指定请求目标
MessageConsumer requestReceiver = session.createConsumer
(requestQueue);
Message requestMessage = requestReceiver.receive();
Message replyMessage = // 创建应答消息
Destination replyQueue = requestMessage.getJMSReplyTo();
MessageProducer replySender = session.createProducer(replyQueue);
replySender.send(replyMessage);
```

例：.NET 响应队列（ResponseQueue）属性

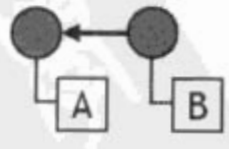
.NET 消息为返回地址预定义了一个 ResponseQueue 属性。它的类型是一个 MessageQueue（如消息通道），应用要把响应消息发送给这个队列[SysMsg]、[Dickman]。

例：Web 服务的请求/响应

SOAP1.2 包括了请求/响应消息交换模式[SOAP1.2 第 2 部分]，但是没有明确规定发送应答的地址，因此该地址是隐含的。这个 SOAP 模式需要支持一个可选的返回地址，从而真正建立异步的 SOAP 消息，并使响应者与请求者解耦合。

WS-Addressing（WS 寻址）标准的出现有助于解决这个问题，它规定了如何识别一个 Web 服务端点，以及要使用哪些 XML 元素。这种地址可以用在 SOAP 消息中来指定一个返回地址。详情请参考第 14 章“结束语”中关于“WS-Addressing（WS 寻址）”的讨论。

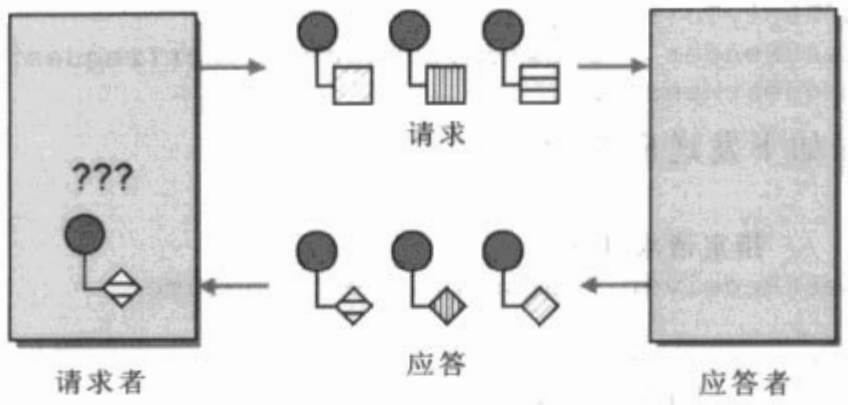
5.7 关联标识符



应用使用消息传递来执行请求/应答，而且接收到一个应答消息。

接收到应答的请求者怎样知道这个应答对应哪一个请求？

当某个进程通过远程过程调用（RPC）调用另一个进程时，该调用是同步的，因此产生的调用结果与调用请求完全对应。但是消息传递是异步的，从调用者的角度看，它发出调用，等待一段时间后将出现结果。调用者甚至不记得发出过请求，或者因为发出的请求太多，以至于不清楚返回的结果对应于哪个请求。现在，当调用者最终获得结果时，它可能不知该如何处理，这样就无法实现最初的调用目的。

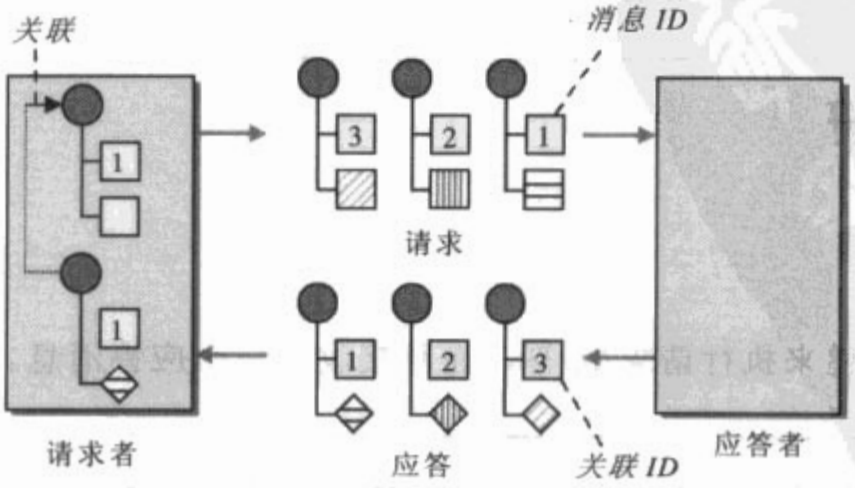


无法匹配应答与请求

调用者可以采用两种方法避免这种混乱的情况。可以在发出新的请求前等待上一次请求的应答，在任何时刻只有一个未完成的请求存在。但是，这种做法会极大地降低处理的吞吐量。调用者可能认为它能按发送请求的顺序获得应答，但是消息传递无法保证消息传输的顺序（见重排器），而且不同请求需要的处理时间也是不一样的，因此调用者的假设并不成立。调用者可以把请求设计为不需要应答，但是这种限制会导致在很多场合下无法应用消息传递。

调用者需要在应答消息中提供指向请求消息的指针或引用，但是消息并没有放在稳定的内存空间中，因此无法由变量引用。但是，消息可以有某种键，这是一个唯一标识符，类似于关系数据库中某行记录的主键。可以用这种唯一标识符把这个消息与其他消息区分开，将使用此消息的客户等等区分开。

每条应答消息都应该包含一个关联标识符，这是一个唯一标识符，指示了该应答消息对应哪一个请求消息。



关联标识符包括六个部分。

1. **请求者**——此应用发送请求并等待应答，从而完成业务任务。

2. **应答者**——这是另一个应用，将接收请求，处理请求，并发送应答。它从请求消息中获得请求 ID，并将其作为关联 ID 存储在应答消息中。

3. **请求**——从请求者发送给应答者的消息，包含一个请求 ID。

4. **应答**——从应答者发送给请求者的消息，包含一个关联 ID。

5. **请求 ID**——请求消息中的一个令牌，唯一地标识请求消息。

6. **关联 ID**——应答消息中的令牌，其值与请求消息中的请求 ID 相同。

关联标识符要这样使用：当请求者创建请求消息时，它为请求分配一个请求 ID，该 ID 必须与其他所有未完成请求的 ID 均不相同。所谓未完成的请求是指尚未获得应答的请求。应答者处理请求时，它将保存请求 ID，并将请求 ID 作为关联 ID 添加到应答中。当请求者处理应答时，通过关联 ID 就能知道应答所对应的是哪个请求。之所以把它称为关联标识符，就是因为调用者要使用这个标识符把应答与导致该应答的请求相关联（相匹配）。

与消息传递的其他方面一样，请求者和应答者必须对一些细节达成一致。它们必须协商确定请求 ID 属性的属性名和类型，还要协商确定关联 ID 属性的属性名和类型。另外，请求和应答消息的格式中必须定义这些属性，或者允许作为自定义属性来增加。例如，如果请求者把请求 ID 存储在一个名为 request_id 的顶级 XML 元素中，并且它的值是一个整数，那么应答者必须知道这个元素名和类型，才能找到请求 ID 并正确地加以处理。请求 ID 的值和关联 ID 的值一般采用相同的类型，如果并非如此，请求者必须知道应答者怎样把请求 ID 转换为应答 ID。

这种模式就是一种异步完成令牌模式（Asynchronous Completion Token Pattern）[POSA2]，不过更为简单，而且特定于消息传递。请求者是发起者（Initiator），应答者是服务（Service），请求者中的消费者负责处理应答，这就是完成处理程序（Completion Handler），消费者使用关联标识符实现应答与请求的匹配，该关联标识符就是异步完成令牌（Token）。

关联 ID（也是请求 ID）一般存放在消息的首部，而不是在消息体中。该 ID 不是应用数据的一部分，也就是说，关联 ID 并非请求者要发送给应答者的命令或数据。实际上，应答者并不真正使用 ID，它只是存储从请求消息中取出的 ID，并把它添加到对应该请求消息的应答消息中。由于消息体才是在两个系统之间传递的具体消息内容，而 ID 不属于消息内容，因此它位于消息的首部。

这种模式的关键在于，应答消息包含有令牌（关联 ID），可以用于识别相应的请求（通过其请求 ID）。对此有几种不同的实现方法。

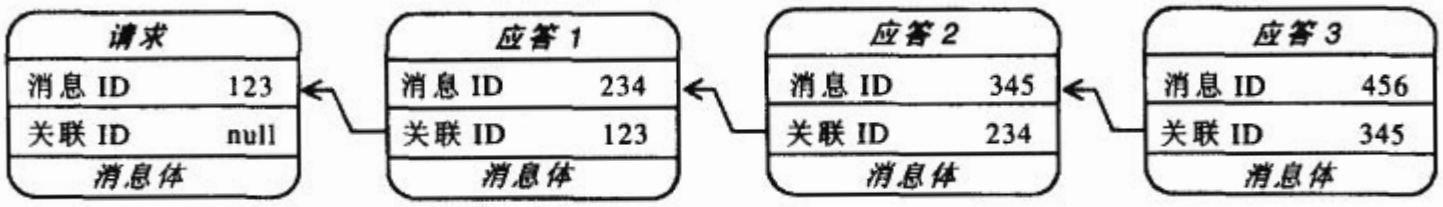
最简单的方法是让每个请求都包含一个唯一的 ID，如消息 ID，而且响应消息的关联 ID 就是相应请求的唯一 ID，从而把应答与对应的请求关联起来。但是，当请求者试图处理应答时，只是知道相应的请求消息还不够。请求者真正想知道的是，最初因为哪一个业务任务而发出了这个请求消息，只有这样它才能使用应答中的数据完成这个业务任务。

业务任务（如执行股票交易或履行订单发货）可能都有自己唯一的业务对象标识符（如

订单 ID），因此业务任务的唯一标识符可以作为请求/应答的关联 ID。当请求者获得应答以及应答的关联 ID 时，可以不管请求消息，把应答直接发送给最初产生请求的业务对象。在这种情况下，请求者和应答者应该在请求和应答消息中使用一个自定义的业务对象 ID 属性，而不是消息中内置的请求消息 ID 和应答关联 ID 属性。业务对象 ID 可以唯一标识一个业务对象，请求/应答消息对正是要完成这个业务对象的任务。

折衷的方法是让请求者保留一张请求 ID 与业务对象 ID 的映射表。请求者想让对象 ID 作为私有属性时，或者请求者无法控制应答者的实现，而只能依靠应答者把请求消息的 ID 复制为应答消息的关联 ID，此时这种方法特别有用。在这种情况下，请求者获得应答后，它会在映射表中查询关联 ID，得到业务对象 ID，然后使用这个业务对象 ID 利用应答数据继续完成业务任务。

消息有不同的消息 ID 和关联 ID 属性，因此请求/应答消息对可以链接起来。一个请求带来一个应答，而这个应答导致另一个请求，相应地又产生另一个应答，如此类推就构成了请求/应答链。消息的消息 ID 能唯一地标识它所表示的请求；如果消息还包含关联 ID，那么这个消息还是另一个请求消息（由关联 ID 标识）的应答。



请求/应答链

如果应用想要从最近的应答反向追踪到最初的请求消息，此时才会用到请求/应答链。通常，所有应用想知道的只是最初的请求，而不管整个过程中出现了多少次应答。如果是这种情况，一旦消息有非空的关联 ID，它就是一个应答，所有由它产生的后续应答也应该采用同样的关联 ID，这样就能立即找到最初的请求，而无需沿着请求/应答链反向追踪。

除了使用关联标识符来匹配应答和相应的请求，请求消息还应当包含返回地址，指出应答放在哪个通道上返回。关联标识符建立了应答消息与相应请求的匹配，而消息序列的标识符可用于指定消息在消息序列中的位置，即一个消息在同一发送者发出的一系列消息中所处的位置。

例：JMS 关联 ID 属性

JMS 消息有一个预定义的关联标识符属性：JMSCorrelationID，它通常与另一个预定义属性 JMSMessageID 联合使用 [JMS1.1]，[Monson-Haefel]。应答消息的关联 ID 按照请求消息的消息 ID 设置，如下所示：

```
Message requestMessage = // 获得请求消息
Message replyMessage = // 创建应答消息
String requestID = requestMessage.getJMSMessageID();
replyMessage.setJMSCorrelationID(requestID);
```

例：.NET 关联 ID 属性

.NET 中的每个消息都有一个 CorrelationId 属性，这是确认消息中的一个字符串，一般设置为原始消息的 ID。MessageQueue 还有特殊的查看和接收方法：PeekByCorrelationId（string）和 ReceiveByCorrelationId（string），这些方法基于指定的关联 ID 可以查看和消费队列中的消息（如果有的话），参见选择性消费[SysMsg]、[Dickman]。

例：Web 服务的请求/应答

Web 服务标准（如 SOAP 1.1[SOAP1.1]）未对异步消息传递提供很好的支持，但是 SOAP 1.2 开始计划提供这方面的支持。SOAP 1.2 加入了请求/应答消息交换模式[SOAP1.2，第 2 部分]，这是异步 SOAP 消息传递的基本组成部分。但是，请求/应答模式没有严格要求支持“多个正在处理的请求”，因此没有定义一个标准的关联标识符字段，甚至没有定义可选的关联标识符字段。

在实际应用中，服务请求者常常会发送多个请求，因此确实需要支持“多个未完成的请求”。“Web 服务体系结构使用说明”[WSAUS]介绍了几种不同的异步 Web 服务实例。其中的 4 个都使用了 SOAP 首部中的 message-id 字段和 response-to 字段，从而可以建立响应消息与相应请求消息的关联，这 4 种情况分别是：请求/响应、远程过程调用（其中传输协议不能直接支持[同步的]请求/响应）、多重异步响应以及异步消息传递。下面是请求/响应的例子：

SOAP 请求消息包含一个消息标识符

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:MessageId>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
      ........
  </env:Body>
</env:Envelope>
```

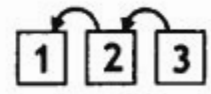
SOAP 响应消息包含与原始请求相应的关联信息

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-567b-2891-b623-9dke28yod7m9</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
```

```
        ........
    </env:Body>
</env:Envelope>
```

与 JMS 及.NET 的例子一样，在这个 SOAP 例子中，请求消息包含一个唯一的消息标识符，响应消息包含一个响应（如关联 ID）字段，响应字段的值等于相应请求消息的消息标识符。

5.8 消息序列



应用需要把大量的数据发送给另一个进程，这些数据无法放在一个消息中发送。还有另一种情况，应用发出一个请求后，相应的应答包含太多的数据，以至于无法用一个消息发送。

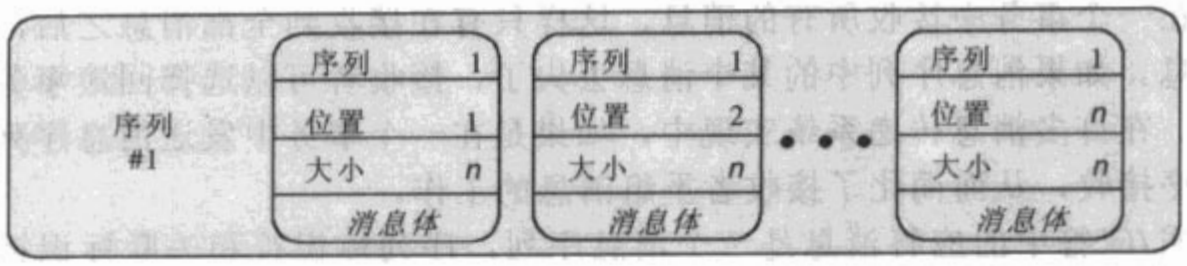
怎样使用消息传递传送任意大小的数据？

我们可能认为，消息可以是任意大小，这当然很好，但是实际上，消息能容纳的数据是有限的。一些消息传递实现对消息的最大容量有一个绝对的限制。其他实现虽然允许使用很大的消息，但是大消息肯定会降低系统的性能。即使消息传递实现能支持大消息，消息生产者或消息消费者一次所能处理的数据量也是有限的。例如，许多基于 COBOL 和基于大型机的系统一次只能消费或生产大小为 32KB 的数据块。

因此，应该如何解决这个问题呢？一种方法是对应用加限制，使传输的数据量绝对不会大于消息传递层在一个消息中所能存储的数据量。但是这种限制非常武断，它会阻碍应用实现所需的功能。如果请求的结果会产生大量的数据，调用者可以发出多个请求，每个请求只产生一块数据，但这样又会增加网络的流量，而且调用者必须很清楚需要多少个数据块。接收者可以一直监听数据块，直到没有更多的数据块为止（但是它怎么才能知道再没有更多的数据块了？），然后尝试把这些数据块重新组装成原始的大块数据，这个过程很容易出错。

要解决这个问题，灵感来自于邮购公司，邮购公司有时把邮购品装在多个盒子中运送。如果要使用三个盒子，那么装运人员会把它们分别标记为“1/3”、“2/3”和“3/3”，因此接收者就能知道他接收到的是哪一个盒子，还能知道是否收到了所有的盒子。这种技术同样可以应用于消息传递。

只要需要把一大块数据分解为消息大小的数据块，就可以将数据作为消息序列来发送，并用序列标识字段标记每个消息。



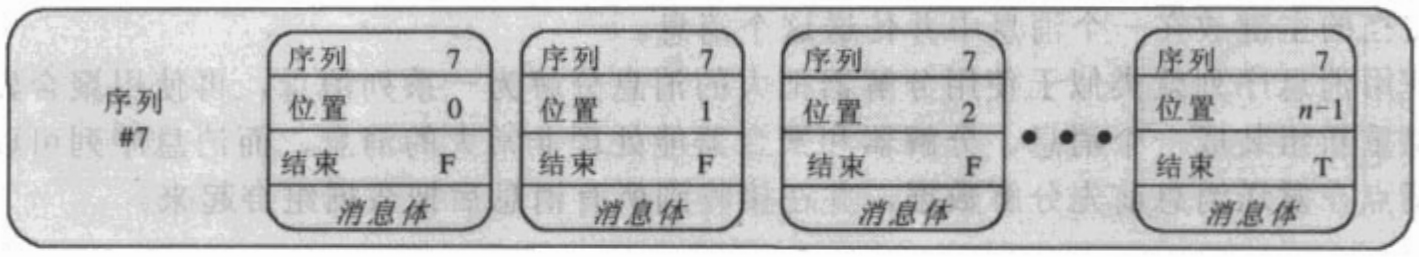
三个消息序列标识字段如下：

1. **序列标识符**——把本消息簇与其他消息簇区分开来。

2. **位置标识符**——唯一地标识序列中的各个消息，并对各个消息完成顺序排序。

3. **大小或结束指示符**——指定消息簇中消息的数量，或者标记簇中的最后一个消息（它的位置标识符则指定了簇的大小）。

在设计序列时，要让序列中的每个消息指示出序列的总大小——也就是说，给出序列中消息的总数量。另一种方法是让每个消息都指出它是否是序列中的最后一个消息。



带有结束指示符的消息序列

假设需要将一组数据作为由 3 个消息组成的消息簇来发送。这个消息簇的序列标识符是一个唯一的 ID。每个消息的位置标识符是不一样的，可以是 1、2 或 3（假设编号从 1 开始，而不是从 0 开始）。如果发送者从一开始就知道消息的数目，那么它可以将每个消息的序列大小设置为 3。如果发送者在数据发送结束前并不知道需要多少消息（例如，发送者按数据流的方式发送数据），对于最后一个消息之前的各个消息，发送者要将这些消息的“序列结束”标志设置为假（false）。发送者准备发送序列中的最后一个消息时，要把消息的序列结束标志设置为真（true）。不管采用哪种方式，消息中的位置标识符和结束指示符将为接收者提供足够的信息，以便把消息序列重组成完整的数据块，即使序列中的消息不是按顺序接收的也没有关系。

如果接收者正在等待一个消息序列，那么发送给它的每一个消息都必须是序列中的一部分，即使整个序列中只有一个消息。否则，如果只有这样一个单部分消息，而没有序列标识字段，就会把接收者搞糊涂，认为这个消息是非法的（见非法消息通道）。

如果接收者只接收到序列中的一部分消息，而不是全部消息，它应该把已经接收到的消息重新转发给非法消息通道。

应用可以使用事务性客户来发送和接收序列。发送者在一个事务中完成序列中所有消息的发送。采用这种方法，除非所有的消息都已发送，否则不会传送任何消息。同样地，

接收者可以在一个事务中接收所有的消息，这样只有在接收到全部消息之后，它才会真正消费这些消息。如果消息序列中的某个消息丢失了，接收者可以选择回滚事务，以后再消费这些消息。在许多消息传递系统实现中，如果是在一个事务中发送消息序列，消息将按照发送的顺序接收，从而简化了接收者重组消息的工作。

如果请求/应答中的应答消息是一个消息序列，序列标识符和关联标识符一般是相同的。如果发送请求的应用希望同一个请求有多个响应，并且其中一个或多个响应由多个部分组成，那么这两个标识符可能不同。只有一个响应时，唯一地标识响应及其序列也是允许的，不过有些多余。

消息序列可能与竞争消费者(Competing Consumer)或消息分派器(Message Dispatcher)不兼容。如果不同的消费者/执行者接收到同一个序列中的不同消息，那么所有接收者都无法将消息重组成原始的数据，除非它们彼此交换所收到的消息内容。因此，消息序列应当通过只有一个消费者的消息通道传输。

可以用声明标签来代替消息序列。如果通信的两个应用都能访问公共数据库或文件系统，就不用在两个应用之间传送很大的文档，而是可以将文档存储在数据库或文件系统中，只把文档的主键放在一个消息中并传送这个消息。

使用消息序列就类似于使用分解器把大的消息分解为一系列消息，再使用聚合器把这些消息重新组装成一个消息。分解器和聚合器能处理非常大的消息，而消息序列可以帮助消息端点在发送消息前先分解数据，并在接收到所有消息后把数据组合起来。

例：大型文档传输

假设发送者需要向接收者发送一个非常大的文档，这个文档无法放在一个消息中，也不可能一次发送完毕。在这种情况下，必须把文档划分成很多块，每一块作为一个消息发送。在每个消息中，要指出它在序列中的位置和总的消息数。例如，一条 MSMQ 消息的最大容量是 4MB，[Dickman]中介绍了如何利用 MSMQ 发送一个多部分消息序列。

例：多项查询

以请求查询某位作者的所有著作清单为例。由于返回的著作清单可能很大，设计消息传递时可以把与条件匹配的各个著作项分别作为一个消息返回。之后，在每个消息中要指出它对应哪一个查询、此消息在消息序列中的位置，以及要返回的消息总数。

例：分布式查询

假设一个查询需要分成多个部分，由多个接收者共同完成。如果查询的结果存在一定的顺序性，就必须在应答消息中指出这种顺序，这样才能正确地组装出完整的应答。每个接收者必须知道它在所有接收者中的位置，并且要在应答的消息序列中标示出这个位置。

例：JMS 和.NET

JMS 或.NET 都没有提供对消息序列的支持。因此，消息传递应用必须实现自己的序列

字段。在 JMS 中，应用可以在消息首部定义自己的消息属性，因此这是一种候选方法。.NET 不支持在消息首部提供应用定义的属性，但是可以在消息体中定义新的字段。如果序列的接收者必须根据消息的序列字段过滤消息，那么把序列字段存储在消息的首部要比存储在消息体中好，这样做可以使过滤操作更简单。

例：Web 服务：多重异步响应

目前的 Web 服务标准还不能很好地支持异步消息传递，但是 W3C 已经在考虑相关的解决办法。“Web 服务体系结构使用说明”[WSAUS]介绍了多种不同的异步 Web 服务实例。其中，多重异步响应使用 SOAP 消息首部中的 message-id 和 response-to 字段把响应与请求关联起来，采用消息体中的 sequence-number 和 total-in-sequence 字段来顺序地标识响应。下面就是多重响应的例子：

SOAP请求消息包含一个消息标识符

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <n:MessageId>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:MessageId>
    </n:MsgHeader>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

第一个 SOAP 响应消息包含对应原始请求的顺序和关联信息

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <!-- MessageId will be unique for each response message -->
      <!-- ResponseTo will be constant for each response message in the sequence-->
      <n:MessageId>uuid:09233523-567b-2891-b623-9dke28yod7m9</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
    <s:Sequence xmlns:s="http://example.org/sequence">
      <s:SequenceNumber>1</s:SequenceNumber>
      <s:TotalInSequence>5</s:TotalInSequence>
    </s:Sequence>
  </env:Header>
```

```
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

最后一个 SOAP 响应消息包含对应原始请求的顺序和关联信息

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2002/06/soap-envelope">
  <env:Header>
    <n:MsgHeader xmlns:n="http://example.org/requestresponse">
      <!-- MessageId will be unique for each response message -->
      <!-- ResponseTo will be constant for each response message in the sequence-->
      <n:MessageId>uuid:40195729-sj20-pso3-1092-p20dj28rk104</n:MessageId>
      <n:ResponseTo>uuid:09233523-345b-4351-b623-5dsf35sgs5d6</n:ResponseTo>
    </n:MsgHeader>
    <s:Sequence xmlns:s="http://example.org/sequence">
      <s:SequenceNumber>5</s:SequenceNumber>
      <s:TotalInSequence>5</s:TotalInSequence>
    </s:Sequence>
  </env:Header>
  <env:Body>
    ........
  </env:Body>
</env:Envelope>
```

消息首部中的 message-id 可以用作响应中的序列标识符。每个响应消息中的序列号和序列总数分别是一个位置标识符和大小指示符。

5.9 消息到期



应用在使用消息传递。如果消息数据或请求不能在规定的时间内接收到，这个消息就是无用的，应该被忽略。

发送者如何指示到期时间(超过这个时间则认为消息过时，不应再做处理)?

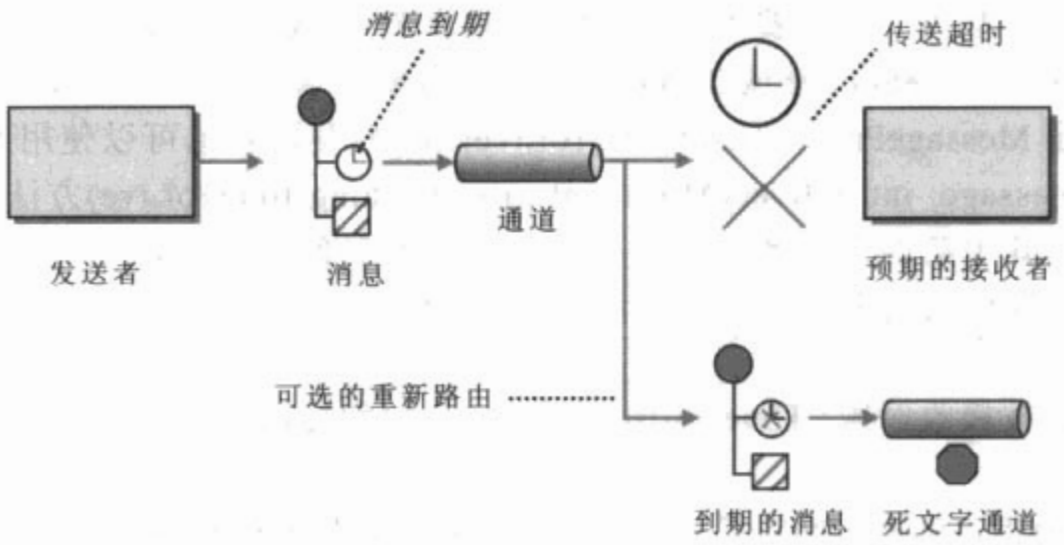
消息传递最终总能保证把消息传送给接收者。但是它无法保证传送所需的时间。例如，

如果连接发送者和接收者的网络中断了一个星期，可能就要花一个星期才能传送消息。尽管参与者（发送者、网络和接收者）并不可靠，但消息传递相当可靠。不过，在不可靠的环境中传送消息可能要花费很长的时间（详细内容请见“可靠传输”）。

通常，消息的内容只是在一定的时间期限内有效。如果请求股票报价的调用者无法在一分钟左右的时间期限内获得响应，就有可能造成经济损失。也就是说请求不但要在一分钟内传送到，还要求响应尽可能快地返回。如果返回的股票报价是一到两分钟之前的，信息就会过时而毫无意义。

一旦发送者发出消息，不过尚未获得应答，它就无法取消或收回消息。另外，接收者可以检查消息是何时发送的，如果消息太老，可以拒绝该消息，但是在不同环境下，不同的发送者对于消息到期时间的想法可能不一样，那么接收者如何知道要拒绝哪些消息呢？因此需要一种方法来让发送者指定消息的生命期。

设置消息到期，指定消息在多长时间期限内是可用的。



一旦过了消息有效的时间，而且消息还没有被消费，这个消息就会到期。消息传递系统的消费者将忽略到期的消息。它们把这些消息当成是根本就没有发送。大多数消息传递系统实现会把到期的消息重新路由给死文字通道，而其他的实现只是简单地丢掉到期的消息。如何处理到期消息是可以配置的。

消息到期就像是牛奶盒上印制的到期日期。超到这个期限后，牛奶就不能饮用了。同样地，消息到期时，消息传递系统就不再传送这个消息。如果接收者接收到的消息在消息到期前无法处理，那么接收者也应该丢掉该消息。

消息到期是一个时间戳（日期和时间），指定了消息生存的时间或何时到期。可以采用相对或绝对方式做此设置。绝对设置要指定消息到期的日期和时间。相对设置则指定消息到期前会存活多长时间，发送消息时，消息传递系统要使用这个时间把将相对时间转换为绝对时间。消息传递系统要负责为处于不同时区的接收者（与发送者不在同一个时区）调整时间戳，负责针对夏令时以及引起时钟不一致的其他问题做出调整。

消息到期属性有一个相关的属性：发送时间，这个属性指定消息是何时发送的。消息的绝对到期时间戳必须比它的发送时间戳晚(否则消息立即就到期了)。为了避免这个问题，发送者一般只给出相对到期时间，由消息传递系统把相对期限与发送时间戳相加（到期时间=发送时间＋生存时间），从而计算出到期时间戳。

当消息到期时，消息传递系统可以简单地把它丢掉，或者是把它转移到死文字通道中。接收者如果发现自己正在处理到期的消息，可以将这个消息转移到非法消息通道中。对于发布-订购通道来说，每个订购者都会得到自己的消息副本，某些副本能成功地达到订购者，而另外一些副本可能在订购者消费之前就到期了。使用请求/应答时，有到期时间设置的应答消息不好处理，因为如果应答消息到期，请求的发送者就无法知道请求是否已接收到。如果使用了应答到期，请求的发送者必须能解决以下情况，即始终无法接收到期望的应答。

例：JMS 生存时间（Time-to-Live）参数

消息到期在 JMS 规范中被称为“消息生存时间”[JMS1.1]、[Hapner]。JMS 消息有预定义的消息到期属性 JMSExpiration，但是发送者不能用 Message.setJMSExpiration(long)来设置这个属性，因为 JMS 提供者在发送消息时会覆盖其设置。发送者应该使用 MessageProducer (QueueSender 或 TopicPublisher)为要发送的所有消息设置超时时间。设置该属性的方法是 MessageProducer.setTimeToLive(long)。发送者也可以使用 MessageProducer.send(Message message, int deliveryMode, int priority, long timeToLive)方法分别设置每个消息的生存时间，其中第四个参数是以毫秒表示的生存时间。这里，生存时间是一个相对设置，指定了消息发送后需要经过多少时间才会到期。

例：.NET 的被接收时限（Time-to-Be-Received）和到达队列时限（Time-to-Reach-Queue）属性

.NET 消息为指定到期时间定义了两个属性：TimeToBeReceived 和 TimeToReachQueue。到达队列设置指定了消息到达其目标队列的时间，在此之后，消息在队列中等待的时间不定。被接收设置指定了消息被接收者消费的时间，它限制了把消息传送到目标队列再加上消息在目标队列中等待的时间的总和。TimeToBeReceived 等价于 JMS 的 JMSExpiration 属性。这两个时间设置的取值类型均为 System.TimeSpan，这表示一段时间 [SysMsg]、[Dickman]。

5.10 格式指示符

多个应用通过消息进行通信，它们采用协商好的数据格式，有可能是适用于整个企业的规范数据模型。但是，该格式可能需要随时间改变。

如何设计消息的数据格式，使之能适应未来可能的变化？

即使你设计的数据格式能够满足所有参与方（应用）的需要，但将来这些需求很可能发生改变。加入新的应用会带来新的格式需求，可能需要向消息中增加新数据，也可能数据不变，但是开发人员找到了更好的方法来构造这些数据。不管是哪一种情况，要想设计出单一的企业数据模型非常困难，设计一个永远不会变化的数据模型几乎是不可能的。

企业的数据格式改变时，如果所有的应用也随之改变，就没有什么问题。如果每个应用都停止使用老的数据格式，并改用新的格式，且它们能在同一时刻完成这一转变，那么实现数据格式的改变就会很简单。问题是，有些应用可能先于其他应用转换数据格式，而且有些很少使用的应用可能根本不会转换数据格式。即使所有应用同时转换数据格式，也要把所有通道中的消息全部消费掉，这样才能在转换开始前使通道清空。

实际上，应用必须能同时支持新老两种数据格式。为此，应用必须要在消息中指明它采用的是新格式还是老格式。

一种解决办法是使用不同的通道集传递新格式的消息。但是，这又会导致过多的通道、重复的设计和复杂的配置，因为每个应用必须为不断扩展的通道重新配置。

更好的解决办法是采用老格式使用的通道来传递新格式的消息。为此，接收者必须采用某种方法把使用相同通道的不同格式消息区分开。每个消息必须指定所使用的格式，而且需要一种简单的方法来指示其格式。

设计一种包含格式指示符（Format Indicator）的数据格式，使得消息能够指定它所使用的格式。

发送者通过格式指示符向接收者告知消息的格式。采用这种方法，如果接收者会接收到多种可能的数据格式，就能知道消息使用的是哪一种格式，而且知道如何解释消息的内容。

实现格式指示符有三种主要方法：

1. **版本号**——唯一标识格式的数字或字符串。发送者和接收者必须协商好特定指示符所指示的是哪一种格式。采用这种方法的优点是，发送者和接收者不必协商建立共享的格式描述符库，但是它也有缺点：不管是发送者还是接收者，都必须知道每个描述符指示的格式是什么，还要知道从哪里访问。

2. **外键**——这是指定格式文档的一个唯一 ID，如文件名、数据库行的键、home 主键，或 Internet 的 URL。发送者和接收者必须协商好键到文档的映射关系，以及模式文档的格式。这种方法的优点是，外键非常紧凑，可以指向共享存储库中详细的数据格式描述。其主要缺点是，每个消息传递参与方可能必须从很远的远程资源中取得格式文档。

3. **格式文档**——描述数据格式的模式。该模式文档不必通过外键获取，也不用通过版本号推断出来，它本身就嵌在消息中。发送者和接收者必须协商好模式的格式。采用这种方法的优点是，消息是自包含的。但是，由于每个消息都要携带变化很少的格式信息，因此会增加消息流量。

版本号或外键可以存储在发送者和接收者协商好的消息首部字段中。对格式版本不感

兴趣的接收者可以忽略该字段。由于格式文档可能过长或很复杂，因此不适合存储在消息的首部，在这种情况下，消息体的格式应该包含模式和数据两个部分。

例：XML

XML 文档提供了这三种方法的例子。下面是一个 XML 声明的例子：

```
<?xml version="1.0"?>
```

其中，1.0 是版本号，表示该文档所遵循的 XML 规范的版本。另一个例子是文档类型声明，可以采用两种形式。一种是包含一个系统标识符的外部 ID，如下所示：

```
<!DOCTYPE greeting SYSTEM "hello.dtd">
```

系统标识符 hello.dtd 是一个外键，它指示了包含 DTD 文档的文件，DTD 文档则描述了 XML 文档的格式。也可以局部地包含声明，如下所示：

```
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
```

标记声明[<!ELEMENT greeting (#PCDATA)>]是一个格式文档，它是一个内嵌的模式文档，描述了 XML 的格式[XML1.0]。

第6章

中场演练：简单的消息传递

6.1 引言

迄今为止，我们已经介绍了大量模式。有基本的消息传递组件，如消息通道、消息和消息端点；还了解了有关消息传递通道和消息构造的详细模式。那么怎样把这些模式组合起来呢？开发人员在集成应用时应该如何使用这些模式呢？应该编写什么样的代码，这些代码又是如何运行的？

在本章中，我们能看到一些具体的代码。这里包括两个例子：

- **请求/应答**——演示（采用 Java 和.NET 的 C#）如何使用消息传递发送请求消息，并利用应答消息做出响应。
- **发布-订购**——探讨如何使用 JMS 主题（Topic）实现观察者[GoF]模式。

通过这两个简单的例子，可以帮助你在自己的应用中增加消息传递。

6.1.1 请求/应答示例

这是一个简单的例子，但功能很强大，它发送一个请求，并返回一个应答。该实例由两个主要类组成：

- **请求者**——这个对象发送请求消息，并等待接收应答消息。
- **应答者**——这个对象接收请求消息，并发送应答消息作为响应。

这两个简单的类会发送简单的消息，而且可以展示许多模式：

- **消息通道和点对点通道**——一个通道用于传送请求，另一个通道用于传送应答。
- **文档消息**——这是默认的消息类型，用于请求和应答消息。
- **请求/应答**——通过一对通道发送的消息对，使两个应用能实现双向通信。
- **返回地址**——发送响应的通道。
- **关联标识符**——导致产生本响应消息的相应请求消息的 ID。
- **数据类型通道**——每个通道上的所有消息都应该有相同的类型。
- **非法消息通道**——用于存放类型不正确的消息。

该示例代码还演示了第 10 章“消息传递端点”中介绍的两个模式：

- **轮询消费者**——请求者如何消费应答消息。
- **事件驱动的消费者**——应答者如何消费请求消息。

尽管这本书并不特定于具体的技术、产品和语言，代码却必须很具体，因此我们选择了两种消息传递编程平台来实现这个例子：

- Java J2EE 提供的 JMS API
- Microsoft .NET 提供的 MSMQ API，采用 C#

在此分别采用了这两种平台来实现同样的请求/应答示例。可以选择你喜欢的平台，来学习如何实现消息传递。如果你想了解如何在另一个平台上实现消息传递，但是不知道怎样针对该平台编写代码，可以参考相应的示例代码，把它与用你熟悉的语言所编写的代码进行比较，就能有所领会。

6.1.2 发布-订购示例

这个例子展示了如何使用发布-订购通道实现观察者模式。它考虑了分布和线程问题，并说明了利用消息传递可以极大地简化这些问题。这个例子展示了如何采用推和拉这两种模型实现通知，并比较了这两种模型的效果。这个例子还探讨了如何设计足够的通道集来满足复杂企业中的需要，也就是可能有大量主体需要通知大量观察者。

该示例代码演示了以下几个模式：

- **发布-订购通道**——提供发布-订购通知的通道。
- **事件消息**——用于发送通知的消息类型。
- **请求/应答**——用作为拉模型技术的一部分，观察者可以利用此模式请求主体的状态。
- **命令消息**——观察者请求主体状态时使用的消息类型。
- **文档消息**——主体将其状态发送给观察者时使用的消息类型。
- **返回地址**——告诉主体如何把状态发送给观察者。
- **数据类型通道**——两个无关的主体是否可以使用相同的通道来更新同样的一组观察者，为此数据类型通道给出了相应的主要原则。

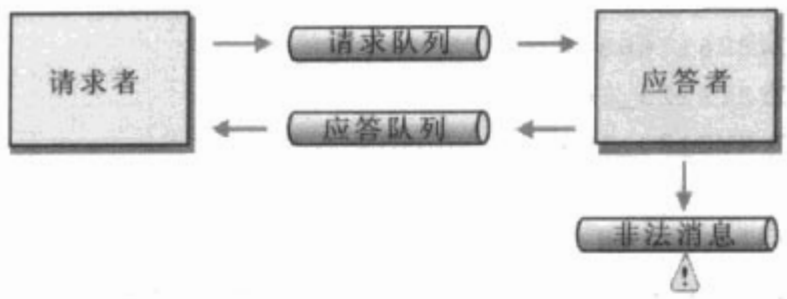
该示例代码还演示了第 10 章“消息传递端点”中介绍的一组模式：

- **消息传递网关**——主体和观察者如何封装消息传递代码，以使其与消息传递无关。
- **事件驱动的消费者**——观察者如何消费通知消息。
- **持久的订购者**——即使发送通知时这些观察者暂时不在线，也不想漏掉这些通知。

这个例子采用 Java JMS 实现，因为 JMS 在它的主题（Topic）接口 API 中对发布-订购通道提供了明确的支持。.NET 在 MSMQ 中对于使用发布-订购语义则没有提供类似的支持。当.NET 确实提供这种支持时，JMS 示例中使用的技术同样可以应用到.NET 程序中。

6.2 JMS 请求/应答示例

这个简单的例子演示了如何使用消息传递实现请求/应答，其中请求者应用发送请求，应答者应用接收请求并返回应答，然后由请求者接收应答。它还演示了如何把非法消息重新路由到一个专门的通道。该实例采用 JMS 实现[JMS]。



请求/应答示例的组件

这个例子使用 JMS1.1 开发，并在 J2EE1.4 参考实现上运行。

6.2.1 请求/应答示例

这个例子由两个主要的类组成：

1. **请求者**——这是一个消息端点，它发送请求消息并等待接收作为响应的应答消息。

2. **应答者**——这也是一个消息端点，它等待接收请求消息，当消息到达时，通过发送应答消息做出响应。

请求者和应答者分别运行在不同的 Java 虚拟机（Java Virtual Machine，JVM）中，实现分布式通信。

本例假设消息传递系统定义了三个队列：

1. jms/RequestQueue——请求者使用此队列把请求消息发送给应答者。

2. jms/ReplyQueue——应答者使用此队列把应答消息发送给请求者。

3. jms/InvalidMessages——当请求者和应答者接收到无法解释的消息时，把消息转发给这个队列。

下面介绍这个例子的工作原理。当从命令行窗口中启动请求者时，会输出以下信息：

```
Sent request
       Time:          1048261736520 ms
       Message ID:    ID:_XYZ123_1048261766139_6.2.1.1
       Correl. ID:    null
       Reply to:      com.sun.jms.Queue: jms/ReplyQueue
       Contents:      Hello world.
```

这说明请求者已经发送了一条请求消息。注意，即使应答者没有运行以至于无法接收到请求消息，请求消息还是会发出。

当从另一个命令行窗口启动应答者时，它会输出以下信息：

```
Received request
       Time:          1048261766790 ms
       Message ID:    ID:_XYZ123_1048261766139_6.2.1.1
       Correl. ID:    null
       Reply to:      com.sun.jms.Queue: jms/ReplyQueue
       Contents:      Hello world.
```

```
Sent reply
      Time:          1048261766850 ms
       Message ID:   ID:XYZ123_1048261758148_5.2.1.1
       Correl. ID:   ID:XYZ123_1048261766139_6.2.1.1
       Reply to:     null
       Contents:     Hello world.
```

这表示应答者接收到了请求消息，并且发送了一个应答消息。

在以上的输出中可以看到一些有意思的信息。首先，请注意请求发送和接收的时间戳。请求是在发送后接收到的（30270 毫秒之后）。其次，发送与接收的请求消息中，消息 ID 是相同的，因为它们是相同的消息。第三，消息的内容也一样，都是 Hello world，因为这正是传递的数据，在收发两端必须相同（这个例子中的请求消息不太好，它实际上是一个文档消息，真正的请求一般是命令消息）。第四，在请求消息中指定了一个名为 jms/ReplyQueue 的队列，这要作为应答消息的目标（返回地址的一个例子）。

下面对接收请求和发送应答的输出做一个比较。首先，应答必须等到接收到请求后才能发送（60 毫秒之后）。其次，应答消息的消息 ID 与请求消息的消息 ID 不同，因为请求与应答消息是单独的不同消息。第三，请求的内容被取出，并添加到应答中[在这个例子中，应答者提供的是简单的“回显（echo）”服务]。第四，没有指定 reply-to 目标，因为对此不需要应答（应答没有使用返回地址）。第五，应答的关联 ID 与请求的消息 ID 相同（应答使用了关联标识符）。

最后，回到第一个窗口，请求者将接收到如下应答：

```
Received reply
      Time:          1048261797060 ms
      Message ID:    ID:_XYZ123_1048261758148_5.2.1.1
      Correl. ID:    ID:_XYZ123_1048261766139_6.2.1.1
      Reply to:      null
      Contents:      Hello world.
```

这个输出中包含几个有意思的内容。应答是在发送后才接收到（30210 毫秒之后）。应答的消息 ID 在发送与接收时保持不变，证明确实是同一个消息。接收到的消息内容与发送的内容也一样，关联 ID 告诉请求者这个应答对应的是哪个请求（关联标识符）。

还需要注意的是，请求者只是发送一个请求，接收应答并退出。因此，一旦它接收到应答，就不再运行。而应答者并不知道何时会接收到一个请求，因此它始终都在运行。为了停止应答者，必须转到它的命令解释窗口，按下回车键，才能让应答者退出。

这就是用 JMS 实现的请求/应答实例。请求由请求者准备和发送。应答者接收请求并发送应答。之后，由请求者接收与最初请求消息对应的应答。

6.2.2 请求/应答代码

首先看一下请求者是如何实现的：

```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;
public class Requestor {

    private Session session;
    private Destination replyQueue;
    private MessageProducer requestProducer;
    private MessageConsumer replyConsumer;
    private MessageProducer invalidProducer;

    protected Requestor() {
        super();
    }

    public static Requestor newRequestor(Connection connection, String requestQueueName,
        String replyQueueName, String invalidQueueName)
        throws JMSException, NamingException {

        Requestor requestor = new Requestor();
        requestor.initialize(connection,requestQueueName,replyQueueName,invalidQueueName);
        return requestor;
    }

    protected void initialize(Connection connection, String requestQueueName,
        String replyQueueName, String invalidQueueName)
        throws NamingException, JMSException {

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

        Destination requestQueue = JndiUtil.getDestination(requestQueueName);
        replyQueue = JndiUtil.getDestination(replyQueueName);
        Destination invalidQueue = JndiUtil.getDestination(invalidQueueName);

        requestProducer = session.createProducer(requestQueue);
        replyConsumer = session.createConsumer(replyQueue);
        invalidProducer = session.createProducer(invalidQueue);
    }
```

```
public void send() throws JMSException {
    TextMessage requestMessage = session.createTextMessage();
    requestMessage.setText("Hello world.");
    requestMessage.setJMSReplyTo(replyQueue);
    requestProducer.send(requestMessage);
    System.out.println("Sent request ");
    System.out.println("\tTime:       " + System.currentTimeMillis() + " ms");
    System.out.println("\tMessage ID:" + requestMessage.getJMSMessageID());
    System.out.println("\tCorrel. ID: " + requestMessage.getJMSCorrelationID());
    System.out.println("\tReply to:  " + requestMessage.getJMSReplyTo());
    System.out.println("\tContents:  " + requestMessage.getText());
}

public void receiveSync() throws JMSException {
    Message msg = replyConsumer.receive();
    if (msg instanceof TextMessage) {
        TextMessage replyMessage = (TextMessage) msg;
        System.out.println("Received reply ");
        System.out.println("\tTime:       " + System.currentTimeMillis() + " ms");
        System.out.println("\tMessage ID: " + replyMessage.getJMSMessageID());
        System.out.println("\tCorrel. ID: " + replyMessage.getJMSCorrelationID());
        System.out.println("\tReply to: " + replyMessage.getJMSReplyTo());
        System.out.println("\tContents: " + replyMessage.getText());
    } else {
        System.out.println("Invalid message detected");
        System.out.println("\tType:      " + msg.getClass().getName());
        System.out.println("\tTime:      " + System.currentTimeMillis() + " ms");
        System.out.println("\tMessage ID: " + msg.getJMSMessageID());
        System.out.println("\tCorrel. ID: " + msg.getJMSCorrelationID());
        System.out.println("\tReply to: " + msg.getJMSReplyTo());

        msg.setJMSCorrelationID(msg.getJMSMessageID());
        invalidProducer.send(msg);

        System.out.println("Sent to invalid message queue");
        System.out.println("\tType:      " + msg.getClass().getName());
        System.out.println("\tTime:      " + System.currentTimeMillis()+ " ms");
        System.out.println("\tMessage ID: " + msg.getJMSMessageID());
        System.out.println("\tCorrel. ID: " + msg.getJMSCorrelationID());
        System.out.println("\tReply to: " + msg.getJMSReplyTo());
    }
  }
}
```

想要发送请求并接收应答的应用可以使用一个请求者来完成这些工作。应用为请求者提供了与消息传递系统的一个连接。它还指定了三个队列的 JNDI 名，这三个队列分别是

请求队列、应答队列和非法消息队列。这些都是请求者完成自身初始化时所需的信息。

在 initialize 方法中，请求者使用连接和队列名连接到消息传递系统。

- 请求者使用连接创建一个会话（session）。应用只需要一个与消息传递系统的连接，但是应用中每个想要发送和接收消息的组件都必须建立各自不同的会话。两个线程不能共享同一个会话，它们应该使用不同的会话才能保证会话的正常工作。
- 请求者使用队列名查找队列，即 Destination（目标）。队列名是 JNDI 标识符，由 JndiUtil 完成 JNDI 查找。
- 为了把消息发送到请求队列，请求者创建了一个 MessageProducer。为了从应答队列接收消息，请求者创建了一个 MessageConsumer，同时还创建了另一个生产者（Producer）负责把消息转移到非法消息队列。

请求者必须能够发送请求消息。为此，它实现了 send()方法。

- 创建一个 TextMessage，并将其内容设置为“Hello world”。
- 把消息的 reply-to 属性设置为应答队列。这是一个返回地址，告诉应答者如何返回应答消息。
- 使用 requestProducer 发送消息。生产者与请求队列连接，所以会把消息发送到请求队列。
- 然后，输出所发送消息的详细内容。输出操作在消息发送完成后才做，因为消息的 ID 由消息传递系统设置，而在消息确实发送之前是不会设置的。

请求者还要能够接收应答消息。为此它实现了 receiveSync()方法。

- 使用 replyConsumer 接收应答。消费者与应答队列连接，所以会从该队列接收消息。它使用 receive()方法获取消息，该方法会同步阻塞，直到消息传送到队列并从队列中读出，因此请求者是一个轮询消费者。由于接收操作是同步的，所以请求者的接收方法名为 receiveSync()。
- 消息应当是一个 TextMessage。倘若如此，请求者就会得到消息的内容，并输出消息的详细内容。
- 如果消息不是 TextMessage，则这个消息就无法得到处理。请求者并不是把这种消息丢掉，而是把它重新发送给非法消息队列。重新发送消息将改变消息的 ID，因此在重发前，请求者要把原始的消息 ID 存储为消息的关联 ID（见“关联标识符”）。

采用这种方式，请求者就完成了所需的所有工作以发送请求、接收应答，并且将没有意义的应答路由到一个特殊的队列。（注意：JMS 提供了一个特殊的 QueueRequestor 类，它的作用也是实现一个能接收应答的请求者，这与我们所做的一样。我们之所以实现了自己的代码，而没有采用 JMS 已有的类，只是为了说明这些代码的工作原理。）

下面让我们看看应答者是如何实现的。

```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
```

```
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class Replier implements MessageListener {

    private Session session;
    private MessageProducer invalidProducer;

    protected Replier() {
        super();
    }

    public static Replier newReplier(Connection connection,
      String requestQueueName, String invalidQueueName)
        throws JMSException, NamingException {

        Replier replier = new Replier();
        replier.initialize(connection, requestQueueName, invalidQueueName);
        return replier;
    }

    protected void initialize(Connection connection, String requestQueueName,
    String invalidQueueName)
        throws NamingException, JMSException {

        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination requestQueue = JndiUtil.getDestination(requestQueueName);
        Destination invalidQueue = JndiUtil.getDestination(invalidQueueName);

        MessageConsumer requestConsumer = session.createConsumer(requestQueue);
        MessageListener listener = this;
        requestConsumer.setMessageListener(listener);

        invalidProducer = session.createProducer(invalidQueue);
    }

    public void onMessage(Message message) {
        try {
```

```
if((message instanceof TextMessage)&&(message.getJMSReplyTo()!= null)){
    TextMessage requestMessage = (TextMessage) message;
    System.out.println("Received request");
    System.out.println("\tTime:   " + System.currentTimeMillis()+" ms");
    System.out.println("\tMessage ID: "+requestMessage.getJMSMessageID ());
    System.out.println("\tCorrel.ID: "+requestMessage.getJMSCorrelation ID());
    System.out.println("\tReply to:   "+ requestMessage.getJMS ReplyTo());
    System.out.println("\tContents:   " + requestMessage.getText());

    String contents = requestMessage.getText();
    Destination replyDestination = message.getJMSReplyTo();
    MessageProducer replyProducer = session.createProducer(replyDes tination);

    TextMessage replyMessage = session.createTextMessage();
    replyMessage.setText(contents);
    replyMessage.setJMSCorrelationID(requestMessage.getJMSMessageID());
    replyProducer.send(replyMessage);

    System.out.println("Sent reply");
    System.out.println("\tTime:   " + System.currentTimeMillis() + " ms");
    System.out.println("\tMessage ID: " + replyMessage.getJMSMessa geID());
    System.out.println("\tCorrel. ID: " + replyMessage.getJMSCorrela tionID());
    System.out.println("\tReply to:   " + replyMessage.getJMSReply To());
    System.out.println("\tContents:   " + replyMessage.getText());
} else {
    System.out.println("Invalid message detected");
    System.out.println("\tType:       " + message.getClass().getName());
    System.out.println("\tTime:     " + System.currentTimeMillis() + " ms");
    System.out.println("\tMessage ID: " + message.getJMSMessageID());
    System.out.println("\tCorrel. ID: " + message.getJMSCorrelationID());
    System.out.println("\tReply to:   " + message.getJMSReplyTo());

    message.setJMSCorrelationID(message.getJMSMessageID());
    invalidProducer.send(message);

    System.out.println("Sent to invalid message queue");
    System.out.println("\tType:   " + message.getClass().get Name());
    System.out.println("\tTime:     " + System.currentTimeMillis() + " ms");
    System.out.println("\tMessage ID: " + message.getJMSMessageID());
    System.out.println("\tCorrel. ID: " + message.getJMSCorrelationID());
    System.out.println("\tReply to:   " + message.getJMSReplyTo());
```

```
            }
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```

应用可以使用应答者来接收请求并发送应答。应用为其请求者提供了一个与消息传递系统的连接，并指定了请求和非法消息队列的 JNDI 名（不需要指定应答队列名，因为应答队列可以由消息的返回地址提供）。请求者在完成自身初始化时需要这些信息。

应答者的 initialize 代码非常类似于请求者的初始化代码，但是也存在一些区别，如下所示：

- 应答者不需要查找应答队列，也不用为它创建生产者。这是因为，应答者认为它不会一直往应答队列发送应答。我们后面就会看到，应答者会让请求消息告诉它应当把应答消息返回到哪个队列。
- 应答者是一个事件驱动的消费者，因此它实现了 MessageListener。当消息传输到请求队列时，消息传递系统会自动调用应答者的 onMessage 方法。

一旦应答者完成了自身的初始化，并成为请求队列的监听者，它不用做什么，只需等待消息。请求者要显式地检查应答队列中是否有消息，而应答者则不同，它是事件驱动的，在消息传递系统针对一个新消息来调用应答者的 onMessage 方法之前，应答者不做任何操作。由于 initialize 创建的是请求队列的消费者，所以消息将从请求队列获得。一旦 onMessage 接收到一条新消息，它将如下处理此消息。

- 与处理应答消息的请求者一样，应答者接收的请求消息应当是 TextMessage。消息中还假设要指定把应答发送到哪个队列。如果请求消息不能满足这些需求，应答者将把这些消息转移到非法消息队列（与请求者一样）。
- 如果消息能满足这些需求，应答者将实现其返回地址部分。请注意，请求者设置了请求消息的 reply-to 属性，从而指定了应答队列。应答者要取得该属性的值，并利用这个属性值，在正确的队列上创建一个 MessageProducer。这里最重要的是，应答者不是硬编码的，没有要求使用一个特定的应答队列；它使用的是各个特定请求消息所指定的应答队列。
- 接着应答者创建应答消息。为此它实现了关联标识符，把应答消息的 correcation-id 属性设置为请求消息中的 message-id 属性。
- 应答者发送应答消息，并显示其详细信息。

这样，应答者完成了所需的各项功能，可以接收消息（假设为请求消息），并发送应答。

6.2.3 非法消息示例

现在来看一个非法消息通道的例子。记住，我们需要的队列中有一个名为 jms/InvalidMessages 的队列。如果 JMS 客户（消息端点）接收到一个无法处理的消息时，

它能把这个奇怪的消息转发给一个特殊的通道。

为了说明非法消息的处理过程，我们设计了一个 InvalidMessenger 类。这个对象专门设计为在请求通道上发送格式不正确的消息。请求通道是一个数据类型通道，因为请求的接收者希望请求都遵循某种格式。非法报信者（messenger）会以另外的某种格式发送消息；当应答者接收到消息，并且无法识别出消息的格式时，就会把消息转发给非法消息队列。

我们将在一个命令行窗口中运行应答者，并在另一个窗口中运行非法报信者。当非法报信者发送消息时，会输出如下信息：

```
Sent invalid message
        Type:        com.sun.jms.ObjectMessageImpl
        Time:        1048288516959 ms
        Message ID: ID:_XYZ123_1048288516639_7.2.1.1
        Correl. ID: null
        Reply to:   com.sun.jms.Queue: jms/ReplyQueue
```

这说明消息是 ObjectMessage 的一个实例（而应答者期望的是 TextMessage）。应答者接收到这个非法的消息，并把它重新发送到非法消息队列。

```
Invalid message detected
        Type:        com.sun.jms.ObjectMessageImpl
        Time:        1048288517049 ms
        Message ID: ID:_XYZ123_1048288516639_7.2.1.1
        Correl. ID: null
        Reply to:   com.sun.jms.Queue: jms/ReplyQueue
Sent to invalid message queue
        Type:        com.sun.jms.ObjectMessageImpl
        Time:        1048288517140 ms
        Message ID:  ID:_XYZ123_1048287020267_6.2.1.2
        Correl. ID:  ID:_XYZ123_1048288516639_7.2.1.1
        Reply to:    com.sun.jms.Queue: jms/ReplyQueue
```

值得注意的是，消息转移到非法消息队列时，它实际上是被重新发送，因此会有一个新的消息 ID。正因如此，我们要应用关联标识符：一旦应答者确定消息是非法的，则把消息的主 ID 复制为它的关联 ID，从而保留消息的原始 ID 记录。处理非法消息的代码放在 Replier 类的 onMessage 方法中，如前所示。Requestor.receiveSync()中包含了类似的非法消息处理代码。

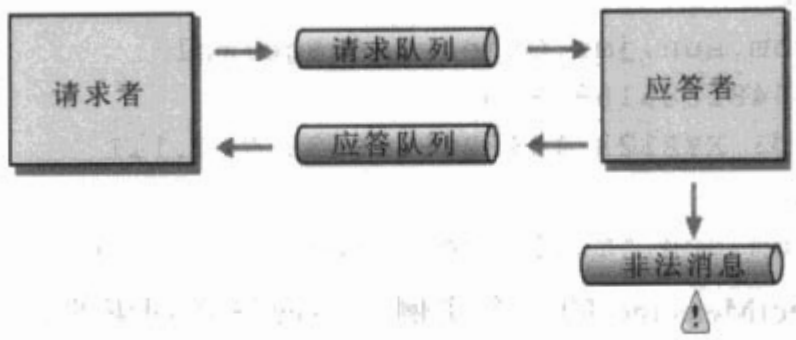
6.2.4 结论

我们介绍了如何实现 Requestor 和 Replier 这两个类（消息端点），它们使用请求/应答（Request-Reply）来交换请求和应答消息。请求消息使用返回地址指定发送应答的队列。应答消息使用关联标识符指定应答所对应的请求。请求者实现了一个轮询消费者来接收应答，

而应答者实现了一个事件驱动的消费者来接收请求。请求和应答队列都是数据类型通道。当消费者接收到类型不正确的消息时，将把它重新路由给非法消息通道。

6.3 .NET 请求/应答示例

这个简单的例子演示了如何使用消息传递实现请求/应答，其中请求者应用发送请求，应答者应用接收请求并返回应答，然后由请求者接收应答。它还说明了如何把非法消息重新路由到一个专门的通道。该实例采用.NET[SysMsg]和 C#实现。



请求/应答示例的组件

这个例子采用 Microsoft .NET 框架 SDK 开发，在装有 MSMQ[MSMQ]的 Windows XP 计算机上运行。

6.3.1 请求/应答示例

这个例子由两个主要的类组成。

1. **请求者**——这是一个消息端点，它发送请求消息，并等待接收作为响应的应答消息。

2. **应答者**——这也是一个消息端点，它等待接收请求消息；当消息到达时，通过发送应答消息做出响应。

请求者和应答者将分别作为独立的.NET 程序运行，实现分布式通信。

本例假设消息传递系统定义了三个队列：

1. **.\private$\RequestQueue**——请求者使用这个 MessageQueue 把请求消息发送给应答者。

2. **.\private$\ReplyQueue**——应答者使用这个 MessageQueue 把应答消息发送给请求者。

3. **.\private$\InvalidQueue**——当请求者和应答者接收到无法解释的消息时，把消息转发给这个 MessageQueue。

下面介绍这个例子是如何工作的。当从命令行窗口中启动请求者时，会输出以下信息：

```
Sent request
      Time:          09:11:09.165342
      Message ID:    8b0fc389-f21f-423b-9eaa-c3a881a34808\149
```

```
        Correl. ID:
        Reply to:       .\private$\ReplyQueue
        Contents:       Hello world.
```

这说明请求者已经发送了一条请求消息。注意，即使应答者没有运行以至于无法接收到请求消息，请求消息同样会输出。

从另一个命令行窗口启动应答者时，它会输出以下信息：

```
Received request
        Time:       09:11:09.375644
        Message ID: 8b0fc389-f21f-423b-9eaa-c3a881a34808\149
        Correl. ID: <n/a>
        Reply to:   FORMATNAME:DIRECT=OS:XYZ123\private$\ReplyQueue
        Contents:   Hello world.
Sent reply
        Time:       09:11:09.956480
        Message ID: 8b0fc389-f21f-423b-9eaa-c3a881a34808\150
        Correl. ID: 8b0fc389-f21f-423b-9eaa-c3a881a34808\149
        Reply to:   <n/a>
        Contents:   Hello world.
```

这说明应答者接收到了请求消息，并且发送了一个应答消息。

在此输出中有一些很有意思的内容。首先，请注意请求 sent 和 received 的时间戳；请求发送后才接收（210302 微秒之后）。其次，发送与接收到的请求消息中，消息 ID 都一样，因为它们是相同的消息。再者，消息的内容是一样的，都是 Hello world，这正是所传递的数据，在收发两端必须相同。第四，请求消息指定了作为应答消息目标的队列（这是返回地址的一个例子）。

下面对接收请求和发送应答的相应输出做一个比较。首先，应答必须等到请求接收到之后才能发送（580836 微秒以后）。其次，应答消息的消息 ID 与请求消息的消息 ID 不同，因为请求与应答消息是单独的不同消息。第三，请求的内容被取出，并添加到应答中。第四，reply-to 目标未指定，因为不需要应答（应答没有使用返回地址）。第五，应答的关联 ID 与请求的消息 ID 相同（应答使用了关联标识符）。

最后，回到第一个窗口，请求者将接收到如下应答：

```
Received reply
        Time:       09:11:10.156467
        Message ID: 8b0fc389-f21f-423b-9eaa-c3a881a34808\150
        Correl. ID: 8b0fc389-f21f-423b-9eaa-c3a881a34808\149
        Reply to:   <n/a>
        Contents:   Hello world.
```

这个输出中有几个内容很有意思。应答在被发送后 0.2 秒被接收到。应答的消息 ID 在发送与接收时保持不变，从而证明确实是同一个消息。接收到的消息内容与发送的内容一样，关联 ID 可以告诉请求者这个应答对应的是哪一个请求（关联标识符）。

请求者运行的时间并不长；它发送一个请求，接收应答后就会退出。但是应答者一直在运行，要等待请求并发送应答。为了停止应答者，必须转到它的命令行解释窗口，按下回车键，才能让应答者程序退出。

这就是用.NET 实现的请求/应答实例。请求由请求者准备和发送。应答者接收请求并发送应答。之后由请求者接收与最初的请求消息对应的应答。

6.3.2 请求/应答代码

首先看一下请求者是如何实现的。

```
using System;
using System.Messaging;

public class Requestor
{
    private MessageQueue requestQueue;
    private MessageQueue replyQueue;

    public Requestor(String requestQueueName, String replyQueueName)
    {
        requestQueue = new MessageQueue(requestQueueName);
        replyQueue = new MessageQueue(replyQueueName);
        replyQueue.MessageReadPropertyFilter.SetAll();
        ((XmlMessageFormatter)replyQueue.Formatter).TargetTypeNames =
          new string[]{"System.String,mscorlib"};
    }

    public void Send()
    {
        Message requestMessage = new Message();
        requestMessage.Body = "Hello world.";
        requestMessage.ResponseQueue = replyQueue;
        requestQueue.Send(requestMessage);

        Console.WriteLine("Sent request");
        Console.WriteLine("\tTime:        {0}",DateTime.Now.ToString("HH:mm:ss.ffffff"));
        Console.WriteLine("\tMessage ID: {0}",requestMessage.Id);
        Console.WriteLine("\tCorrel. ID: {0}",requestMessage.CorrelationId);
        Console.WriteLine("\tReply to:   {0}",requestMessage.ResponseQueue.Path);
        Console.WriteLine("\tContents:   {0}",requestMessage.Body.ToString());
    }
```

```
    public void ReceiveSync()
    {
        Message replyMessage = replyQueue.Receive();


        Console.WriteLine("Received reply");
        Console.WriteLine("\tTime:        {0}",DateTime.Now.ToString("HH:mm:ss.ffffff"));
        Console.WriteLine("\tMessage ID:  {0}",replyMessage.Id);
        Console.WriteLine("\tCorrel. ID:  {0}",replyMessage.CorrelationId);
        Console.WriteLine("\tReply to:    {0}","<n/a>");
        Console.WriteLine("\tContents:    {0}",replyMessage.Body.ToString());
    }
}
```

如果应用想要发送请求并接收应答，可以使用一个请求者完成这些工作。应用指定了两个队列的路径名：请求队列和应答队列。请求者完成自身初始化时需要这些信息。

在请求者的构造函数中，请求者使用队列名连接到消息传递系统。

- 请求者使用队列名查找队列，这些队列是 MessageQueue。队列名是 MSMQ 资源的路径名。
- 请求者设置应答队列的属性过滤器，从而当在队列中读取消息时，同时也会读取消息的所有属性。请求者还设置了格式器（formatter）的 TargetTypeNames，这样消息的内容可以解释为字符串。

请求者必须能够发送请求消息。为此，它实现了 send()方法。

- 创建一条消息，并将其内容设置为“Hello world”。
- 把消息的 ResponseQueue 属性设置为应答队列。这是一个返回地址，它将告诉应答者如何返回应答消息。
- 然后把消息发送到队列中。
- 之后，输出所发送消息的详细内容。输出在消息发送后完成，因为消息的 ID 由消息传递系统设置，而在消息真正发送之前不会设置消息 ID。

请求者还要能够接收应答消息。为此，它实现了 ReceiveSync()方法。

- 使用队列的 Receive()方法接收消息，这个方法会同步阻塞，直到消息传输到队列并从队列中读出，因此请求者是一个轮询消费者。由于接收操作是同步的，请求者的相应方法名为 ReceiveSync()。
- 请求者获得消息的内容，并输出消息的详细信息。

采用这种方式，请求者完成了所需的所有工作，可以发送请求并接收应答。

下面让我们看看应答者是如何实现的。

```
using System;
using System.Messaging;

class Replier {

    private MessageQueue invalidQueue;

    public Replier(String requestQueueName, String invalidQueueName)
    {
        MessageQueue requestQueue = new MessageQueue(requestQueueName);
        invalidQueue = new MessageQueue(invalidQueueName);

        requestQueue.MessageReadPropertyFilter.SetAll();
        ((XmlMessageFormatter)requestQueue.Formatter).TargetTypeNames =
          new string[]{"System.String,mscorlib"};

        requestQueue.ReceiveCompleted += new ReceiveCompletedEventHandler (OnReceiveCompleted);
        requestQueue.BeginReceive();
    }


    public void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue requestQueue = (MessageQueue)source;
        Message requestMessage = requestQueue.EndReceive(asyncResult.Async Result);

        try
        {
            Console.WriteLine("Received request");
            Console.WriteLine("\tTime:       {0}",DateTime.Now.ToString("HH:mm:ss.ffffff"));
            Console.WriteLine("\tMessage ID:{0}",requestMessage.Id);
            Console.WriteLine("\tCorrel. ID:{0}","<n/a>");
            Console.WriteLine("\tReply to:  {0}",requestMessage.ResponseQueue.Path);
            Console.WriteLine("\tContents:  {0}",requestMessage.Body.ToString ());

            string contents = requestMessage.Body.ToString();
            MessageQueue replyQueue = requestMessage.ResponseQueue;
            Message replyMessage = new Message();
            replyMessage.Body = contents;
            replyMessage.CorrelationId = requestMessage.Id;
            replyQueue.Send(replyMessage);

            Console.WriteLine("Sent reply");
```

```
        Console.WriteLine("\tTime:        {0}",DateTime.Now.ToString("HH:mm:ss.ffffff"));
        Console.WriteLine("\tMessage ID: {0}",replyMessage.Id);
        Console.WriteLine("\tCorrel. ID: {0}",replyMessage.CorrelationId);
        Console.WriteLine("\tReply to:   {0}","<n/a>");
        Console.WriteLine("\tContents:   {0}",replyMessage.Body.ToString ());
    }
    catch ( Exception ) {
        Console.WriteLine("Invalid message detected");
        Console.WriteLine("\tType:        {0}",requestMessage.BodyType);
        Console.WriteLine("\tTime:        {0}",DateTime.Now.ToString("HH:mm:ss.ffffff"));
        Console.WriteLine("\tMessage ID: {0}",requestMessage.Id);
        Console.WriteLine("\tCorrel. ID: {0}","<n/a>");
        Console.WriteLine("\tReply to:   {0}","<n/a>");

        requestMessage.CorrelationId = requestMessage.Id;

        invalidQueue.Send(requestMessage);

        Console.WriteLine("Sent to invalid message queue");
        Console.WriteLine("\tType:        {0}",requestMessage.BodyType);
        Console.WriteLine("\tTime:        {0}",DateTime.Now.ToString("HH:mm:ss.ffffff"));
        Console.WriteLine("\tMessage ID:{0}",requestMessage.Id);
        Console.WriteLine("\tCorrel. ID:{0}",requestMessage.Correla tionId);
        Console.WriteLine("\tReply to:  {0}",requestMessage.ResponseQueue.Path);
    }

    requestQueue.BeginReceive();
  }
}
```

当应用需要接收请求并发送应答时，就要实现与此类似的一个应答者。应用指定了请求和非法消息队列的路径名（它不需要指定应答队列的路径名，因为后面你就会看到，应答队列可由消息的返回地址提供）。请求者在完成自身初始化时需要这些信息。

应答者的构造函数与请求者的构造函数非常类似，但是还是存在一些区别，如下所示。

- 其中一个区别是，应答者不会查找应答队列。这是因为应答者认为它不会一直往应答队列发送应答。我们后面就会看到，应答者会让请求消息告诉它把应答消息返回到哪个队列。
- 另一个区别是，应答者是一个事件驱动的消费者，因此它实现了 ReceiveCompleted-EventHandler。当消息传输到请求队列时，消息传递系统会自动调用指定的方法

OnReceiveCompleted。

应答者完成自身的初始化，作为请求队列的监听者，然后不用做什么，只需要等待消息的到达。请求者要显式地检查应答队列中是否有消息，而应答者则不同，应答者是事件驱动的，在消息传递系统针对新消息调用应答者的 OnReceiveCompleted 方法之前，应答者不做任何操作。由于构造函数中创建的是请求队列的事件处理程序，因此将从请求队列获得消息。一旦 OnReceiveCompleted 被调用，它将按以下步骤读取新消息并加以处理：

- 消息源是请求队列 MessageQueue。
- 消息本身通过运行队列的 EndReceive 方法获得。之后应答者打印输出消息的详细内容。
- 应答者将实现其返回地址部分。请注意，请求者设置了请求消息的 response-queue 属性，从而指定了应答队列；应答者要取得该属性的值，并使用这个属性值引用正确的 MessageQueuer。这里最重要的是，应答者不是硬编码的，没有要求使用一个特定的应答队列；它会使用各个特定请求消息所指定的应答队列。
- 接下来应答者创建应答消息。为此它实现了关联标识符，并把应答消息的 correlation-id 属性设置为请求消息中的 message-id 属性。
- 应答者然后发送应答消息并显示其详细信息。
- 如果接收到消息，但是不能成功地处理，将会抛出一个异常，应答者会把消息重新发送给非法消息队列。在这个过程中，它把新消息的关联 ID 设置为原始消息的消息 ID。
- 一旦应答者完成消息的处理，则运行 BeginReceive 开始监听下一条消息。

到此，应答者完成了所有所需的工作，可以接收消息（假设为请求消息）并发送应答。如果它无法对消息做出应答，则把消息发送给非法消息队列。

6.3.3 非法消息示例

现在来看一个非法消息通道的例子。记住，我们需要的队列中有一个名为 private$\InvalidMessages 的队列。如果 MSMQ 客户（消息端点）接收到一个无法处理的消息时，它能把这个奇怪的消息转发给这个特殊的通道。

为了说明非法消息的处理过程，我们设计了一个 InvalidMessenger 类。这个对象专门设计为在请求通道上发送格式不正确的消息。与其他任何通道一样，请求通道是一个数据类型通道，因为请求的接收者希望要请求遵循一定的格式。非法报信者会以另外一种格式发送消息；当应答者接收到消息，并且无法识别出消息的格式时，就会把消息转发给非法消息队列。

我们在一个命令行窗口中运行应答者，并在另一个窗口中运行非法报信者。当非法报信者发送它的消息时，有如下输出：

```
Sent request
        Type:    768
        Time:    09:39:44.223729
```

```
        Message ID:8b0fc389-f21f-423b-9eaa-c3a881a34808\168
        Correl. ID:00000000-0000-0000-0000-000000000000\0
        Reply to:  .\private$\ReplyQueue
```

类型 768 意味着消息内容的格式采用了二进制（而应答者期望的内容格式是 text/XML）。应答者接收到非法消息，并把它重新发送到非法消息队列。

```
Invalid message detected
        Type:          768
        Time:          09:39:44.233744
        Message ID:    8b0fc389-f21f-423b-9eaa-c3a881a34808\168
        Correl. ID:    <n/a>
        Reply to:      <n/a>
Sent to invalid message queue
        Type:          768
        Time:          09:39:44.233744
        Message ID:8b0fc389-f21f-423b-9eaa-c3a881a34808\169
        Correl. ID:8b0fc389-f21f-423b-9eaa-c3a881a34808\168
        Reply to:  FORMATNAME:DIRECT=OS:XYZ123\private$\ReplyQueue
```

值得注意的是，当消息转移到非法消息队列时，实际上是重新发送，因此它有一个新的消息 ID。所以我们要使用关联标识符；一旦应答者确定消息是非法的，则把消息的主 ID 复制为它的关联 ID，以保留消息的原始 ID 记录。处理非法消息的代码放在 Replier 类的 OnReceiveCompleted 方法中，如前所示。

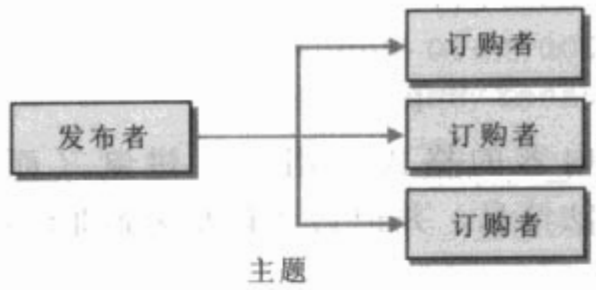
6.3.4 结论

我们介绍了如何实现 Requestor 和应答者 Replier 这两个类（消息端点），它们使用请求/应答交换请求和应答消息。请求消息使用返回地址指定发送应答的队列。应答消息使用关联标识符指定应答所对应的请求。请求者实现一个轮询消费者来接收应答，而应答者实现了一个事件驱动的消费者来接收请求。请求和应答队列都是数据类型通道；当消费者接收到类型不正确的消息时，将把它重新路由给非法消息通道。

6.4 JMS 发布-订购示例

这个简单的例子演示了发布-订购消息传递，同时还探讨了其他可替代的设计方案。只要把一个事件发布一次，订购该事件的多个应用就都能得到通知。采用其他方法也可以将事件通知给订购者。

为了理解一个简单的发布-订购通道有什么作用，首先要考虑如何在多个应用之间以分布式的方式实现观察者模式。在讨论这个问题前，先回顾一下关于观察者的基本知识。



使用 JMS 主题实现发布-订购

6.4.1 观察者模式

观察者模式[GoF]描述了这样一种设计方案，通过这种模式，对象可以向其依赖者（依赖该对象的相关对象）通知所发生的变化，同时保持该对象与其依赖者松耦合，而不论依赖者有多少，即使一个也没有也不会有影响。其组成包括主体和观察者，主体对象发布状态改变信息，观察者是有兴趣接收主体改变通知的对象。当主体的状态改变时，它会调用 Notify()方法，其实现知道有哪些观察者，并调用每个观察者的 Update()方法。有些观察者可能对这个状态改变不感兴趣，不过那些感兴趣的观察者可以通过调用主体的 GetState()得到新的状态信息。主体还要实现 Attach(Observer)方法和 Detach(Observer)方法，观察者要使用这些方法向主体提交注册。

观察者通过两种方式从主体那里得到新的状态：推模型和拉模型。采用推模型，调用每个观察者的 Update 方法时，会把新状态作为参数传给观察者。因此，感兴趣的观察者就不必调用 GetState()方法了，但是这种方式同时会把数据传递给不感兴趣的观察者。与之对应的是拉模型，主体只发送基本的通知，每个观察者自己从主体那里请求新的状态信息。因此，每个观察者可以请求自己真正需要的信息，甚至可能什么信息都不需要。但是这样一来，如果有多个请求需要相同的数据，主体就必须分别为这些请求提供服务。推模型只需一次单向通信——作为更新的一部分，主体要把数据推给观察者。拉模型则需要三次单向通信——主体先通知观察者，观察者向主体请求当前的状态，主体再把当前的状态发送给观察者。我们很快就会看到，单向通信的次数不仅会影响设计时的复杂性，还会影响运行时实现通知的性能。

要实现主体的 Notify()方法，最容易的做法是采用单线程，但是这样做性能很不理想。单个线程会按顺序一次只更新一个观察者，因此，如果观察者位于列表的后面，要等待很长时间才能得到更新。而且，主体要花费很长时间更新其观察者，在此过程中不能做其他任何事情。更糟糕的是，观察者可能会使用它的更新线程对更新通知做出响应，即向主体提出状态查询请求并处理新的数据，如果有这种执行更新线程的观察者，会使更新过程花费的时间更长。

因此，实现主体的 Notify()方法还有一种更老到的方法，Update()调用在其各自的线程中运行。这样的话，所有的观察者都能同时得到更新，每个更新线程中所做的任何操作都不会耽误其他观察者或主体。这种方法的缺点是，实现多线程以及处理线程管理问题要比

单线程更复杂。

6.4.2 分布式观察者

观察者模式一般假设主体与它的观察者都在相同的应用中运行。这种模式的设计也支持分布，其中观察者可以与主体在不同的内存空间运行，甚至不同观察者也可以有不同的内存空间，但是实现这种分布是要做些工作的。Update()和 GetState()方法以及 Attach 和 Detach 方法都必须能够远程访问（见“远程过程调用”）。由于主体必须能够调用各个观察者，而观察者也要调用主体，因此各个对象必须运行在某种对象请求代理（Object Request Broker，ORB）环境中，从而能够远程调用这个环境中的对象。因为更新的内容和状态数据要在内存空间之间传递，应用必须支持串行化，也就是说，要对传递的对象进行编组。

因此，在分布式环境中实现观察者会相当复杂。不仅多线程观察者实现起来有些困难，要保证方法可远程调用，还要远程调用这些方法，这些更增加了难度。仅向某些依赖者通知状态改变就要完成很多工作。

另一个问题是，远程过程调用只有在调用的源、目标和连接它们的网络都正常工作时才能成功。如果主体宣布改变通知而远程的观察者没有准备好处理通知，或者与网络断开了连接，观察者就会丢失这个通知。在某些情况下，观察者即使没有得到通知也能正常工作，但是在另外一些情况下，丢失通知会造成观察者与主体不同步——设计观察者模式正是要避免这个问题。

实现分布也可以采用推和拉模型。如前所述，推模型只需一次单向通信，而拉模型需要三次。当采用远程过程调用实现分布时，推需要一次调用（Update()），而拉至少需要两次调用（Update()和 GetState()）。与非分布式方法调用相比，远程过程调用要花费更多的开销，由于推方法需要更多的调用，所以会使系统的性能很快下降。

6.4.3 发布-订购

发布-订购通道实现了观察者模式，从而能在分布式应用中更容易地使用这种模式。实现该模式需要三个步骤。

1. 消息传递系统管理员创建发布-订购通道（在 Java 应用中，这表示为一个 JMS Topic）。

2. 作为主体的应用创建一个 TopicPublisher（这是一种 MessageProducer），由这个发布者向通道发送消息。

3. 作为观察者（如依赖者）的各个应用创建一个 TopicSubscriber（这是一种 MessageConsumer），以接收通道中的消息（这类似于调用观察者模式中的 Attach(Observer)方法）。

这样就通过通道建立起主体与观察者之间的连接。现在，主体不管在什么时候想发布改变通知，只需发送一个消息。通道将保证每个观察者能接收到该消息的一个副本。

下面是发布改变通知所需的简单代码示例：

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class SubjectGateway {

    public static final String UPDATE_TOPIC_NAME = "jms/Update";
    private Connection connection;
    private Session session;
    private MessageProducer updateProducer;

    protected SubjectGateway() {
        super();
    }

    public static SubjectGateway newGateway() throws JMSException, NamingExcep tion {
        SubjectGateway gateway = new SubjectGateway();
        gateway.initialize();
        return gateway;
    }

    protected void initialize() throws JMSException, NamingException {
        ConnectionFactory connectionFactory = JndiUtil.getQueueConnectionFactory();
        connection = connectionFactory.createConnection();
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination updateTopic = JndiUtil.getDestination(UPDATE_TOPIC_NAME);
        updateProducer = session.createProducer(updateTopic);

        connection.start();
    }

    public void notify(String state) throws JMSException {
        TextMessage message = session.createTextMessage(state);
        updateProducer.send(message);
    }

    public void release() throws JMSException {
        if (connection != null) {
            connection.stop();
```

```
            connection.close();
        }
    }
}
```

SubjectGateway 是主体（在此没有给出）与消息传递系统之间的消息传递网关。主体创建网关，并用这个网关来广播通知。实际上，主体的 Notify()方法就实现为调用 SubjectGateway.notify(String)。然后由网关把消息发送到更新通道，通知所发生的改变。

下面是接收改变通知所需的代码示例。

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class ObserverGateway implements MessageListener {
    public static final String UPDATE_TOPIC_NAME = "jms/Update";
    private Observer observer;
    private Connection connection;
    private MessageConsumer updateConsumer;

    protected ObserverGateway() {
        super();
    }

    public static ObserverGateway newGateway(Observer observer)
        throws JMSException, NamingException {
        ObserverGateway gateway = new ObserverGateway();
        gateway.initialize(observer);
        return gateway;
    }

    protected void initialize(Observer observer)throws JMSException,NamingExcep tion{
        this.observer = observer;

        ConnectionFactory connectionFactory = JndiUtil.getQueueConnectionFactory ();
        connection = connectionFactory.createConnection();
```

```
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination updateTopic = JndiUtil.getDestination(UPDATE_TOPIC_NAME);
        updateConsumer = session.createConsumer(updateTopic);
        updateConsumer.setMessageListener(this);
    }

    public void onMessage(Message message) {
        try {
            TextMessage textMsg = (TextMessage) message; //假设类型强制转换总能成功
            String newState = textMsg.getText();
            update(newState);
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }

    public void attach() throws JMSException {
        connection.start();
    }

    public void detach() throws JMSException {
        if (connection != null) {
            connection.stop();
            connection.close();
        }
    }

    private void update(String newState) throws JMSException {
        observer.update(newState);
    }
}
```

ObserverGateway 也是一个消息传递网关，它位于观察者（没有给出）和消息传递系统之间。观察者创建这个网关，然后使用 attach()方法启用连接（类似于调用观察者模式中的 Attach(Observer)方法）。该网关是一个事件驱动的消费者，因此它实现了 MessageListener 接口，需要实现 onMessage 方法。采用这种方式，当接收到更新时，这个网关会处理消息来得到新状态，然后调用自己的 update(String)方法，该方法将调用观察者中相应的消息。

这两个类实现了采用推模型的观察者。对于 SubjectGateway.notify(String)所发送的通知消息，这个消息的存在只是告诉观察者发生了改变，而消息的内容才能告诉观察者主体的新状态是什么。新状态是由主体推给观察者的。后面就会看到，使用拉模型也能实现这一功能。

6.4.4 比较

对应用之间的分布式通知来说，与传统的同步（如远程过程调用）方法相比，采用发布-订购（如消息传递）方法实现观察者有几个优点。

- **简化了通知**。主体的 Notify()实现变得非常简单，只需把消息发送到通道上。同样，Observer.Update()只需接收消息。
- **简化了附加/分离**。观察者采用的方法不是附加（attach）到主体或者与主体分离（detach），而需要订购通道或者取消订购通道。主体不必实现 Attach(Observer)或 Detach(Observer)（不过观察者可以实现这些方法，来封装订购和取消订购行为）。
- **简化了并发线程**。主体只需要一个线程就能同时更新所有的观察者——通道会负责把通知消息并发地发送给观察者——每个观察者都在自己的线程中处理更新，所以这就简化了主体的实现。由于每个观察者都使用其自己的线程，因此一个线程中的操作不会影响其他线程。
- **简化了远程访问**。主体和观察者都不必实现任何远程方法，也不必运行在一个 ORB 中。它们只需访问消息传递系统，由消息传递系统来实现分布处理。
- **增强了可靠性**。由于通道使用的是消息传递，在观察者能够处理之前，通知消息必须排队，不过这也会使观察者成为障碍而限制了通知。如果观察者想收到它不在线时主体发出的通知，就应该作为持久订购者。

发布-订购方法无法改变的一个问题是串行化。不管采用远程过程调用还是消息传递来实现观察者，从主体的内存空间到各个观察者的内存空间都分布着状态数据，所以这些数据必须串行化（也就是说，需要编组）。无论是远程过程调用，还是发布-订购方法，都必须实现这种串行化行为。

如果说发布-订购方法还有缺点，那就是它需要消息传递，这说明主体和观察者应用必须访问一个共享的消息传递系统，并且必须实现为该消息传递系统的客户。不过，让应用成为消息传递客户并不困难，与使用远程过程调用相比可能更为容易。

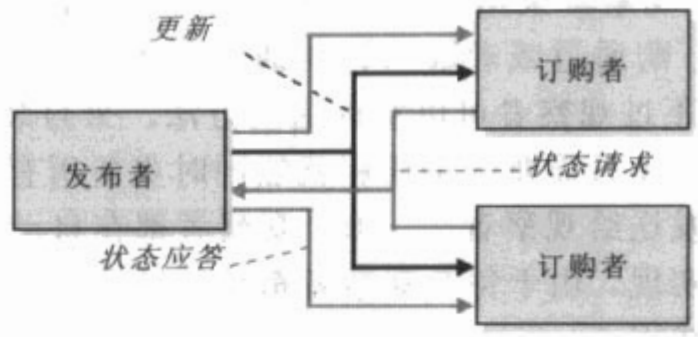
6.4.5 推和拉模型

采用发布-订购方法还可能有一个缺点，这就是拉模型比推模型更复杂。在前面讨论过，拉模型比推模型需要更多的消息往返。如果是在分布式应用之间进行消息传递，额外的通信会极大地影响系统的性能。

采用消息传递进行通信要比远程过程调用更复杂。在这两种情况下，Update()都是单向通信，可能是一个返回空（void）的远程过程调用，也可能是由主体发给观察者的一个事件消息。观察者需要查询主体的状态时，就要麻烦多了。GetState()是双向通信，可能是一个请求状态并返回状态的远程过程调用，也可能是一对请求/应答消息（由一个命令消息请求状态，并由一个单独的文档消息返回状态）。

请求/应答之所以难实现，不仅是因为它需要传递一对消息，还在于需要一对通道来传递这些消息。其中一个通道是从观察者到主体，即获取状态请求（get-state-request）通道；

观察者把状态请求发送到该通道。另一个通道从主体返回到观察者，即获取状态应答（get-state-reply）通道；主体把状态应答发送到该通道。所有的观察者均可共享相同的请求通道，但是它们一般要使用自己的应答通道。每个观察者并不是接收所有响应，而是应该只接收对应其请求的特定响应，为保证这一点，最简单的方法是让每个观察者都有自己的应答通道（还有一种替代的方法，可以只使用一个应答通道，但同时使用关联标识符，以便得出各应答要交给哪个观察者，但是让每个观察者使用不同的通道实现起来更容易）。



使用拉模型实现发布-订购

如果每个观察者都使用一个应答通道，会导致通道数量的爆炸。如此多的通道即使可以管理，当使用通道的观察者的数量在运行时发生动态改变时，消息传递系统的管理员也无法知道应该创建多少个静态的通道。即使给所有的观察者提供了足够的通道，每个观察者也不清楚应该使用哪个通道。

JMS 提供了一个 TemporaryQueue 特性，它是专门用来解决这个问题的[Hapner]（请参见“请求/应答”的介绍）。观察者可以创建一个只供自己使用的临时队列，把该队列指定为请求消息中的返回地址，然后在这个队列上等待应答。如果频繁创建新队列，效率可能不高，这是由你的消息传递系统的实现决定的，而临时队列不能是持久的（用于可靠传输）。但是，如果你不想采用推模型，就可以使用临时队列实现拉模型。

下面的两个类给出了使用拉模型实现网关的方法。

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class PullSubjectGateway {

    public static final String UPDATE_TOPIC_NAME = "jms/Update";
    private PullSubject subject;
```

```
private Connection connection;
private Session session;
private MessageProducer updateProducer;

protected PullSubjectGateway() {
    super();
}

public static PullSubjectGateway newGateway(PullSubject subject)
    throws JMSException, NamingException {
    PullSubjectGateway gateway = new PullSubjectGateway();
    gateway.initialize(subject);
    return gateway;
}

protected void initialize(PullSubject subject) throws JMSException,NamingException{
    this.subject = subject;

    ConnectionFactory connectionFactory = JndiUtil.getQueueConnection Factory();
    connection = connectionFactory.createConnection();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Destination updateTopic = JndiUtil.getDestination(UPDATE_TOPIC_NAME);
    updateProducer = session.createProducer(updateTopic);

    new Thread(new GetStateReplier()).start();
    connection.start();
}

public void notifyNoState() throws JMSException {
    TextMessage message = session.createTextMessage();
    updateProducer.send(message);
}

public void release() throws JMSException {
    if (connection != null) {
        connection.stop();
        connection.close();
    }
}

private class GetStateReplier implements Runnable, MessageListener {

    public static final String GET_STATE_QUEUE_NAME = "jms/GetState";
    private Session session;
    private MessageConsumer requestConsumer;

    public void run() {
```

```
            try {
                session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
                Destination getStateQueue= JndiUtil.getDestination(GET_STATE_QUEUE_NAME);
                requestConsumer = session.createConsumer(getStateQueue);
                requestConsumer.setMessageListener(this);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        public void onMessage(Message message){
            try {
                Destination replyQueue = message.getJMSReplyTo();
                MessageProducer replyProducer = session.createProducer(reply Queue);

                Message replyMessage = session.createTextMessage(subject.getState());
                replyProducer.send(replyMessage);
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
}
```

PullSubjectGateway 与 SubjectGateway 非常相似。这里采用的是拉模型，它有主体的引用，因此当观察者提出请求时，网关可以查询主体的状态。notify(String)现在变成了notifyNoState()，因为拉模型只是发出通知，而不在其中包含任何状态信息（之所以不用notify()作为方法名，是因为 Java 中已经有一个名为 notify()的方法）。

拉模型中主要增加了一个 GetStateReplier，它是一个实现了 Runnable 接口的内部类，因此它可以在自己的线程中运行。这个类还实现了 MessageListener 接口，从而成为一个事件驱动的消费者。其 onMessage 方法从 GetState 队列读取请求，并且把包含主体状态的应答发送到请求消息指定的队列中。采用这种方法，当观察者提出 GetState()请求时，网关就会发送一个应答（见“请求/应答”）。

```
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueRequestor;
```

```
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.NamingException;

public class PullObserverGateway implements MessageListener {

    public static final String UPDATE_TOPIC_NAME = "jms/Update";
    public static final String GET_STATE_QUEUE_NAME = "jms/GetState";
    private PullObserver observer;
    private QueueConnection connection;
    private QueueSession session;
    private MessageConsumer updateConsumer;
    private QueueRequestor getStateRequestor;

    protected PullObserverGateway() {
        super();
    }

    public static PullObserverGateway newGateway(PullObserver observer)
        throws JMSException, NamingException {
        PullObserverGateway gateway = new PullObserverGateway();
        gateway.initialize(observer);
        return gateway;
    }

    protected void initialize(PullObserver observer) throws JMSException, Naming Exception {
        this.observer = observer;

        QueueConnectionFactory connectionFactory = JndiUtil.getQueueConnection
Factory();
        connection = connectionFactory.createQueueConnection();
        session = connection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
        Destination updateTopic = JndiUtil.getDestination(UPDATE_TOPIC_NAME);
        updateConsumer = session.createConsumer(updateTopic);
        updateConsumer.setMessageListener(this);

        Queue getStateQueue = (Queue) JndiUtil.getDestination(GET_STATE_QUEUE_ NAME);
        getStateRequestor = new QueueRequestor(session, getStateQueue);

    }

    public void onMessage(Message message) {
        try {
            // 消息的内容为空
```

```
            updateNoState();
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }

    public void attach() throws JMSException {
        connection.start();
    }

    public void detach() throws JMSException {
        if (connection != null) {
            connection.stop();
            connection.close();
        }
    }

    private void updateNoState() throws JMSException {
        TextMessage getStateRequestMessage = session.createTextMessage();
        Message getStateReplyMessage = getStateRequestor.request(getStateRequest Message);
        TextMessage textMsg = (TextMessage) getStateReplyMessage; // 假设类型强制转换总能成功
        String newState = textMsg.getText();
        observer.update(newState);
    }
```

同样地，PullObserverGateway 类似于 ObserverGateway，但是实现拉模型需要更多的代码。在初始化时，该类不但要建立监听更新的 updateConsumer，还要建立发送 GetState() 请求的 getStateRequestor。在实现拉模型的版本中，网关的 onMessage 代码不考虑消息的内容，因为该消息是空的。消息的存在只是告诉观察者主体有所改变，但是没有告诉观察者主体的新状态是什么。因此，它只需要调用 updateNoState()（此方法命名与 notifyNoState() 相似）。

在 updateNoState()和 update(String)的实现中很容易看出，采用拉模型和推模型的观察者之间存在着区别。推模型从参数中获取新的状态，只要更新观察者即可。拉模型必须在更新观察者之前查询并获得新的状态。为了取得新的状态，它要使用 getStateRequestor 发送请求并接收应答。应答中包含了主体的新状态，网关要使用这个新状态来更新观察者（注意，在这个简单的实现中，网关是单线程的，因此当它发送获取状态（get-state）请求并等待应答时，不再处理其他的任何更新。如果请求或应答消息需要花很长时间传送，网关将一直等待，其间出现的更多更新将排队）。

正如你所看到的，拉模型要比推模型复杂。它需要更多的通道（包括为每个观察者建立的临时通道）和消息（每个感兴趣的观察者都需要三个消息才能完成一次更新，而不是所有的观察者只采用一条消息就能完成更新），主体和观察者类需要编写更多的代码来管理

额外的消息传递，运行时对象需要更多的线程执行额外的消息传递。如果所有这些对你的应用来说都是可接受的，那么采用拉模型就是一种可行的方法；反之，则可以从推模型开始，因为推模型更简单。

6.4.6 通道设计

迄今为止，我们考虑的还只是一个主体向它的观察者通知一种状态。使用推模型，这需要采用发布-订购通道把主体的状态改变通知给观察者。

真正的企业应用比这还要复杂。应用中包含大量需要发布改变信息的主体。每个主体往往包含好几种不同的状态，我们把它们称为“方面（aspect）”，这些方面是独立改变的。一个观察者可能对不同主体中的不同方面感兴趣，不仅同一个类的多个实例可能作为主体，不同类的实例也可能会作为主体。

因此，在复杂的应用中，更新的语义会很快变得复杂起来。观察者模式需要解决这样一些实现问题，如“观察多个主体”和“明确指定感兴趣的修改”。另外，SASE（Self-Addresses Stamped Envelope，自编址的加戳信封）模式描述了观察者模式与命令模式的组合模式，其中，当发生某种改变时，主体要发送观察者指定的命令[Alpert]。

我们暂不考虑如何保证观察者只接收到它们需要的更新，而是先来考虑对消息传递造成的影响，也就是需要多少个通道。

首先分析一个简单的例子。一家企业有多个不同的应用负责存储客户的联系信息，如邮件地址。当某个应用更新了客户的地址时，其他可能用到这个新地址信息的应用应该得到通知。不仅如此，可能有多个应用需要知道地址何时改变，因此它们都要注册来接收这个通知。

这个问题很容易解决。只需要采用一个发布-订购通道宣布地址改变就可以了。这样每个能改变地址的应用还要负责通知改变，把消息发布到通道上。每个希望接收通知的应用必须订购该通道。如下所示为一个特定的地址改变消息。

```
<AddressChange customer_id="12345">
    <OldAddress>
        <Street>123 Wall Street</Street>
        <City>New York</City>
        <State>NY</State>
        <Zip>10005</Zip>
    </OldAddress>
    <NewAddress>
        <Street>321 Sunset Blvd</Street>
        <City>Los Angeles</City>
        <State>CA</State>
        <Zip>90012</Zip>
    </NewAddress>
</AddressChange>
```

现在来考虑另一个问题。当产品缺货时，企业中的某些应用需要发出通知，其他接收这些通知的应用就能重新订购缺货的产品。这个问题与前一个问题类似，可以使用一个发布-订购通道发出产品缺货的声明，以相同的方式解决。以下是这样一个消息。

```
<OutOfProduct>
    <ProductID>12345</ProductID>
    <StoreID>67890</StoreID>
    <QuantityRequested>100</QuantityRequested>
</OutOfProduct>
```

但是，这样我们就想知道：客户地址改变和产品缺货声明是否可以使用相同的通道？可能不行。首先，数据类型通道要求同一通道上的所有消息必须是相同的类型，在这个例子中，它们都必须遵循相同的 XML 模式。<AddressChange>与<OutOfProduct>的元素类型显然是截然不同的，因此不能把它们发送到同一个通道。当然，可以重新修改数据格式，使这两种消息类型都遵循相同的 XML 模式，并且接收者可以判断出哪个消息是地址改变消息，哪个是产品更新消息。但是现在的问题是，对地址改变感兴趣的应用和对产品更新感兴趣的应用可能并不是同一个应用，因而如果消息使用相同的通道，应用可能会频繁接收到它不感兴趣的消息。因此，有必要使用两个单独的地址改变通道和产品更新通道。

现在，我们再来看第三个例子，这里客户的信用等级会发生改变。该消息可能如下所示。

```
<CreditRatingChange customer_id="12345">
    <OldRating>AAA</OldRating>
    <NewRating>BBB</NewRating>
</CreditRatingChange>
```

和产品更新通知的例子一样，除了使用地址改变通道和产品缺货通道外，可以尝试采用新的信用等级改变（credit-rating-changed）通道。这样可以把信用等级改变通知与地址改变通知区分开来，而且依赖者只需注册它们感兴趣的那一类改变。

使用这种方法会导致通道数量的爆炸。假设与客户相关的所有数据，包括名称、提供给邮递或货运的联系方式（地址、电话号码、电子邮件）、记账、信用等级、服务级别、标准折扣等应用。只要其中一种信息发生改变，其他的应用都需要得到通知。为每种信息创建一个通道会导致出现大量的通道。

创建大量通道会加重消息传递系统的负担。如果大量通道只是传递少量的数据流，这不但浪费资源，还增加了应用分布的难度，进而会增加消息传递的开销。作为依赖者的应用可能不清楚在如此多通道中应该订购哪一个通道。多个通道需要多个发送者和接收者，这就可能导致大量线程要不断地检查大量通道，但实际上这些通道上通常没有任何消息。因此，创建如此多的通道并不是好的解决办法。

最好能采用同一个通道来传递地址改变消息和信用等级改变消息，因为它们都与客户

的改变有关，而且对一种改变感兴趣的应用可能也会对另一种改变感兴趣。但采用单独的产品缺货通道仍然是合理的，因为对客户感兴趣的应用可能并不对产品感兴趣，反之亦然。

地址改变消息和信用等级改变消息有不同的格式，而数据类型通道要求：同一个通道上的消息必须有相同的格式。对 XML 来说，这意味着所有的消息必须有相同的根元素类型，但是可以使用不同的可选嵌套元素。因此，统一的客户改变消息可能如下所示：

```
<CustomerChange customer_id="12345">
    <AddressChange>
        <OldAddress>
            <Street>123 Wall Street</Street>
            <City>New York</City>
            <State>NY</State>
            <Zip>10005</Zip>
        </OldAddress>
        <NewAddress>
            <Street>321 Sunset Blvd</Street>
            <City>Los Angeles</City>
            <State>CA</State>
            <Zip>90012</Zip>
        </NewAddress>
    </AddressChange>
</CustomerChange>
```

```
<CustomerChange customer_id="12345">
    <CreditRatingChange>
        <OldRating>AAA</OldRating>
        <NewRating>BBB</NewRating>
    </CreditRatingChange>
</CustomerChange>
```

剩下的问题是，对地址改变感兴趣的货运应用可能对信用等级的改变不感兴趣，而记账应用恰好与之相反。这些应用可以使用选择性消费者（Selective Consumer）来获取它们感兴趣的消息。如果选择性消费者实现太复杂，而消息传递系统可以很方便地支持多个通道，那么更好的选择是采用单独的通道。

企业体系结构和设计面临许多问题，要解决这些问题需要做出大量折衷的考虑，并不存在简单的答案。采用发布-订购通道与采用其他消息通道一样，目的是帮助接收者只接收到它们需要的通知，而不要带来不同通道的爆炸性增长，也不要加重观察者的负担（如果有大量线程运行着大量消费者，这些消费者要监测大量通道，就会极大地增加观察者的负担）。

6.4.7 结论

这个例子展示了发布-订购通道是观察者模式的一个具体实现，以便在分布式环境中更方便地使用观察者模式。当使用通道时，Subject.Notify()和 Observer.Update()的实现变得更

加简单，因为它们只需要发送和接收消息。消息传递系统会负责处理分布和并发问题，从而使远程通知更加可靠。推模型比拉模型更简单，而且效率往往更高，特别是在采用消息传递的分布式通知中，推模型更为高效。当然，拉模型也可以采用消息传递实现。在很多数据都会改变的复杂应用中，可能会尝试为不同的改变通知创建不同的通道，不过此时更可行的做法是：提供给相同观察者的类似通知都使用相同的通道传送。即使你的应用不需要使用消息传递做其他用途，但只要应用之间需要通知彼此的变化，使用消息传递就是有意义的，因为这样就能充分利用到发布-订购通道。

第7章

消息路由

7.1 引言

在第3章“消息传递系统”中，我们介绍了如何使用消息路由器来使消息源与消息的最终目标解耦合。本章介绍一种特殊类型的消息路由器，解释如何为集成解决方案提供路由和代理功能。大多数模式都是对消息路由器模式的改进，还有一些模式结合了多个消息路由器，以解决更复杂的问题。因此，可以把消息路由模式划分为以下几类：

- 简单路由器，这是消息路由器的变种，它把消息从一个入站通道路由到一个或多个出站通道。
- 组合路由器，把简单路由器结合到一起，建立更为复杂的消息流。
- 架构模式，描述了基于消息路由器的架构风格。

7.1.1 简单路由器

基于内容的路由器会检查消息的内容，并根据其内容把消息路由到另一个通道中。使用这种路由器，消息的生产者只需把消息发送到一个通道，而由基于内容的路由器把这些消息路由到正确的目标，从而减轻了发送应用的负担，并且可以避免消息生产者与特定目标通道之间的耦合。

消息过滤器是一种特殊的基于内容的路由器。它会检查消息的内容，且只有当消息的内容符合某种标准时，才把消息发送到另一个通道。否则，就会把消息丢掉。消息过滤器完成的功能与选择性消费者完成的功能非常类似，最大的不同是消息过滤器属于消息传递系统的组成部分，它会把合格的消息路由到另一个通道，而选择性消费者建立在消息端点中。

基于内容的路由器和消息过滤器实际上能解决类似的问题。基于内容的路由器根据其内部设置的标准把消息路由到正确的目标。如果使用发布-订购通道和消息过滤器阵列，同样可以把消息路由到正确的目标，其中每个过滤器对应一个可能的消息接收者。每个消息过滤器会把与特定目标不匹配的消息过滤掉。基于内容的路由器预先就知道应该把消息路由到哪个通道上，因此它对消息具有完全的控制，不过还要依赖所有可能的目标通道实现路由。相反，消息过滤器阵列采用响应过滤方式，它把路由规则分配到多个过滤器中，而不是由一个组件根据所有可能的目标通道进行消息路由。这两种解决方案的综合考虑将在“消息过滤器模式”中详细介绍。

基本的消息过滤器使用固定的规则来确定输入消息的目标。需要更灵活的路由策略时，采用动态路由器更合适。这种路由器可以把控制消息发送给指定的控制端口，从而修改路由的规则。大多数消息路由器中都结合了动态路由器的动态性。

第4章“消息传递通道”介绍了点对点通道和发布-订购通道的概念。有时，可能需要把消息发送给多个接收者，同时还想保持对接收者的控制。采用接收表（Recipient List）可以做到这一点。从本质上讲，接收表就是一个基于内容的路由器，它能把一个消息路由给多个目标通道。

有的消息包含由独立的数据项组成的列表。那么应该如何单独地处理这些数据项呢？使用分解器可以把大消息划分为单独的消息，然后各条消息就能单独地路由和处理了。

但是，分解得到的消息还需要重新组合，形成原来的单个消息。这是聚合器要完成的一项功能。聚合器可接收消息流，找出相关的消息，并将它们组合为一个消息。与其他路由模式不同，聚合器是一种状态消息路由器，因为它必须在其内部存储消息，直到指定的条件满足。也就是说，聚合器在发布一条消息前可能要消费多条消息。

为了连接运行在多个计算机上的应用和组件，由于使用的是消息传递方法，因此多个消息可以得到并行处理。例如，多个进程可以消费同一通道上的消息。其中一个进程可能比其他进程执行得快，这会导致消息的处理顺序不正确。但是，有的组件（如基于分类账的系统）必须按正确的顺序处理每条消息。重排器（Resequencer）可以把顺序不正确的消息重新有序化。重排器也是一个状态消息路由器，因为它必须在其内部存储很多消息，直到序列中最后一个消息到达。与聚合器不同，重排器最后会把所消费的消息全数发布。

下表总结了各种消息路由器的特性（其中没有单独列出动态路由器，因为所有的路由器都可以实现为动态路由器）。

模　式	消费的消息数	发布的消息数	有状态吗？	备　注
基于内容的路由器	1	1	无（大多数情况下）	
过滤器	1	0或1	无（大多数情况下）	
接收表	1	多个（包括0个）	无	
排序器	1	多个	无	
聚合器	多个	1	有	
重排器	多个	多个	有	消费的消息数与发布的消息数相同

7.1.2 组合路由器

管道和过滤器体系结构的突出优点是，由此可以把多个过滤器组合起来形成一个更大的解决方案。组合消息处理器（Composed Message Processor）和分散收集器（Scatter-Gather）可以把多个不同的消息路由器组合起来，构建更全面的解决方案。这两种模式都能帮助我们从多个数据源获取信息，并把它们重新组合成一个单独的消息。组合消息处理器可以把

单个消息划分为多个部分，而分散收集器可以把同一消息的副本发送给多个接收者。

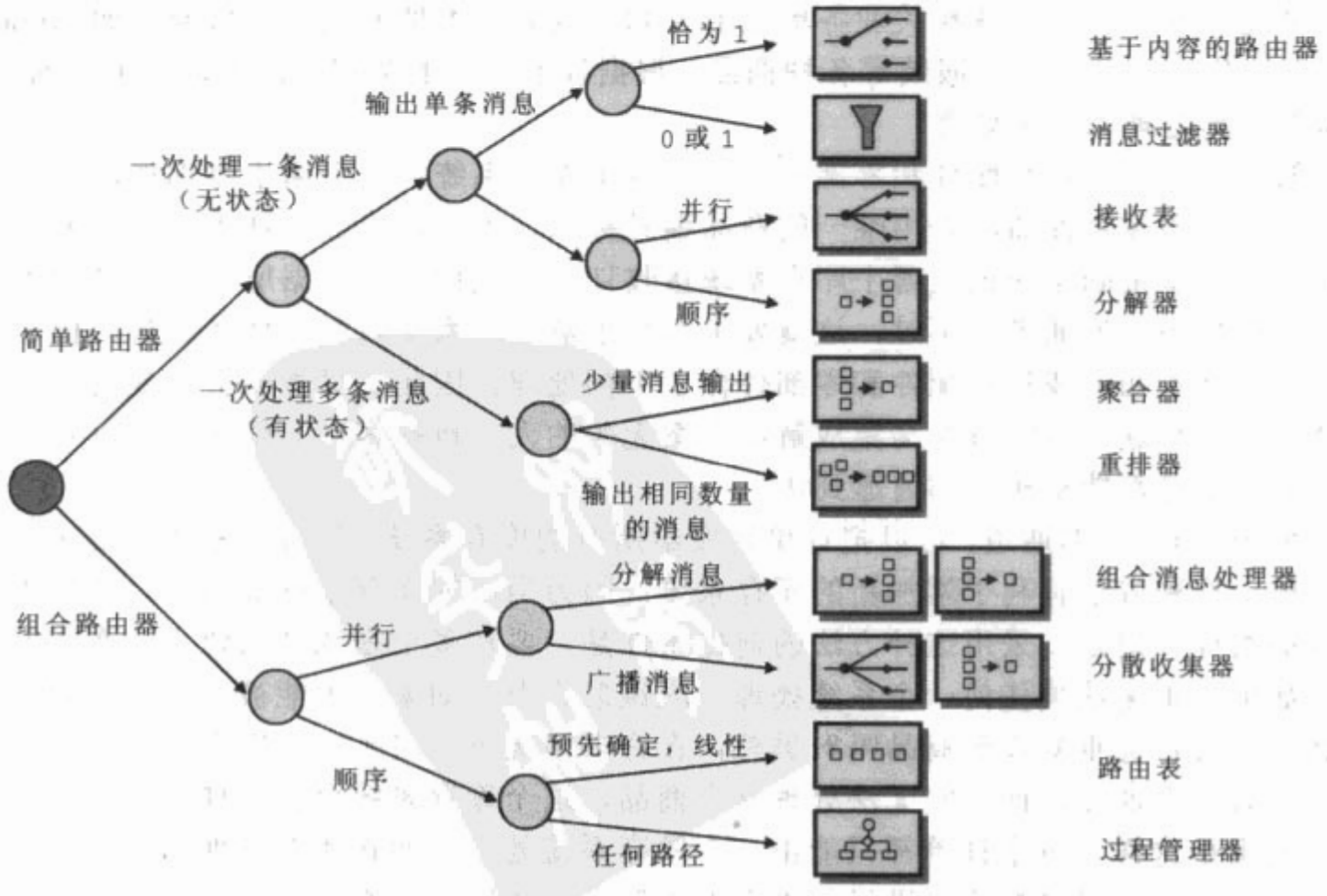
组合消息处理器和分散收集器都能同时把一个消息路由给多个参与者，并把返回的应答重新组装成一个消息。这些模式能管理消息的并行路由。还有另外两种模式会管理消息的顺序路由，即通过一系列单独的步骤实现消息路由。如果想通过一个集中的组件来控制消息的路径，可以使用路由表（Routing Slip）指定消息传递的路径。这种模式就像附加在办公文档上的传阅单一样，会按照接收者的顺序传递消息。类似地，也可以使用过程管理器（Process Manager），它能提供更大的灵活性，但是需要在完成每项功能后把消息返回给中心组件。

7.1.3 架构模式

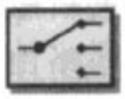
使用消息路由器，通过集中的消息代理（Message Broker）可以构建一个集成解决方案。与其他各种消息路由设计模式不同，这种模式体现了集散式（hub-and-spoke）架构风格。

7.1.4 针对恰当的用途采用正确的路由器

本章包含 12 个模式。应该如何针对恰当的用途方便地找到正确的模式呢？下面的决策图可以帮助你作出选择，你只需做出是或否的决定。例如，如果你想找到一种简单的路由模式，一次只消费一条消息，但要按顺序发布多条消息，那么应该使用分解器模式。这个图还反映出各个模式之间存在非常密切的关系。例如，路由表模式和过程管理器模式可以解决相似的问题，而消息过滤器应对的则是完全不同的问题。



7.2 基于内容的路由器



假设我们要建立一个订单处理系统。当接收到输入的订单时，首先要验证订单，然后检查所订购的商品是否有存货，这项功能是由库存系统完成的。这一顺序处理过程最好采用管道和过滤器方式实现。为此我们创建了两个过滤器，一个用于验证订单，一个用于库存系统，并且将输入的消息路由给这两个过滤器。但是，在许多企业的集成环境中可能存在多个库存系统，每个系统分别负责处理指定的商品。

如果一个逻辑功能的实现需要跨越多个物理系统，这种情况应该如何应对？

集成解决方案可将已有的应用连接起来，使它们一同工作。由于其中许多应用在开发时没有考虑集成问题，因此对集成解决方案的设计来说，要想找到一种理想的环境，能将业务功能很好地封装在单一的一个系统中，这是很少见的。例如，由于采办或商业合作，常常会带来多个有相同业务功能的系统。而且，许多作为集成商的企业往往要与多个实现相同功能（如检查库存、下订单）的系统打交道。更棘手的是，这些系统可能由公司操控，也可能受商业合作企业或相关业务单位的控制。例如，大型的电子零售商（如 Amazon 公司）允许你订购图书及至服装等各种商品。根据订购商品的类型，订单将由不同的“幕后”商家的订单处理系统来处理。

假设一个公司销售配件和零部件，并有两个库存系统：一个用于零部件，一个用于配件。同时还假设每件商品均由唯一的商品编号标识。当公司接收到订单时，它要根据所订购商品的类型来确定应该由哪个库存系统接收订单。我们可以根据所订购商品的类型为输入订单创建不同的通道。不过，这就要求客户了解我们系统内部的体系结构，但是客户实际上可能并不知道我们对配件和零部件做了不同处理。因此，尽管业务功能的实现跨越了多个系统，但这一点应该对与集成解决方案无关的人（包括客户）隐藏。所以我们希望不同商品的消息都能从相同的通道到达。

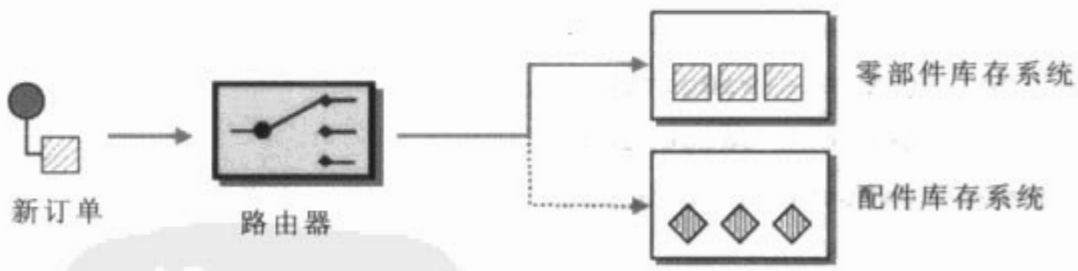
使用发布-订购通道，可以把订单转发给所有的库存系统，让每个系统自己决定是否处理订单。这种方法有利于增加新的库存系统，因为当新的库存系统加入时，我们不必修改已有的组件。但是，采用这种方法的前提条件是，要在多个系统之间建立分布式的协作关系。如果订单无法被任何一个系统处理，应该怎么办？如果订单能被多个系统处理，又该怎么办？客户会重复收到商品吗？另外，在许多情况下，对于一个库存系统来说，如果订单中只有一个商品，而且它无法处理这个商品，这个库存系统会把这样的订单当作一个错误。如果是这样，每个订单都只能由一个库存系统处理，而在所有其他的库存系统中都会引起错误。因此很难把这些错误与“真正的”错误（如非法的订单）区分开。

有一种替代的解决方法，可以使用商品编号作为通道的地址。每个商品都有自己专门的通道，客户只需把订单发布到与商品编号相关的通道上即可，而不必了解零部件与配件之间的内部区别。库存系统会监听它能处理的商品对应的所有通道。这种方法利用了通道的可寻址性，能把消息路由到正确的库存系统。但是如果有大量的商品，很快就会导致通道呈爆炸式增长，这会加重系统的运行时和管理开销。如果为所提供的每件商品都创建新的通道，很快就会导致混乱。

我们还应该尽量减少消息的通信量。例如，可以把订单消息按顺序从一个库存系统路由给下一个库存系统。第一个可以接受订单的系统将消费消息，并处理订单。如果它无法处理该订单，就把订单消息传递给下一个系统。采用这种方法，可以消除订单同时被多个系统接受的危险。而且，如果最后一个系统把订单消息再传回来，则证明所有系统都无法处理这个订单。但是，这种解决方案要求系统之间要相互了解，才能把消息从一个系统传递给下一个系统，它类似于职责链模式（Chain of Responsibility Pattern）[GoF]。在基于消息的集成领域中，通过系统链传递消息可能会带来巨大的开销。而且，这种方法要求各个系统之间必须协作，如果某些系统是由外部商业伙伴维护的，不在我们的控制范围内，就很难实现职责链方式的集成。

总之，我们需要一个合适的解决方案，基于这种解决方案，我们不必知道业务功能划分到多个系统中，而且这种方案应当能高效地使用消息通道和消息业务流，并能保证订单恰好只被一个库存系统处理。

使用基于内容的路由器，根据消息的内容把每条消息路由给正确的接收者。



基于内容的路由器首先检查消息的内容，并根据消息中包含的数据把消息路由到不同的通道上。可以根据许多标准来完成路由，如是否存在某些字段、是否有指定的字段值等。实现基于内容的路由器时，由于可能会频繁维护路由器，所以要特别注意路由功能应当易于维护。在更复杂的集成环境中，基于内容的路由器可能会以可配置规则引擎的形式出现，可以根据一组可配置的规则来计算目标通道。

7.2.1 减少依赖性

基于内容的路由器是更为通用的消息路由器的一种常用形式。它采用了预测路由机制（predictive routing），也就是说，它了解其他所有系统的功能。由于能将每条输出消息直接

发送给正确的系统，因此能够高效地实现路由。其缺点是必须对所有可能的接收者及其功能了如指掌。每当增加、修改或删除接收者时，基于内容的路由器都要做相应的修改，这就给维护带来很大麻烦。

如果接收者能对路由过程施加更多的控制，就可以避免基于内容的路由器对各个接收者的依赖。可以把这种做法总结为反应型过滤（Reactive Filtering），因为当消息到达时，每个参与者能过滤出与之相关的消息。将路由控制如此分布到各个接收方，就可以不再需要基于内容的路由器，但是这种解决方案的效率通常比较低。有关的解决方案和折衷考虑将在消息过滤器和路由表模式中详细介绍。

动态路由器是基于内容的路由器和反应型过滤器这两种方法的折衷，由接收者把自己的功能告知基于内容的路由器。基于内容的路由器维护着各接收者功能的一个列表，并据此完成输入消息的路由。为了增加这个灵活性，也要付出一定的代价，与简单的基于内容的路由器相比，这种解决方案更加复杂，而且这种系统的调试更加困难。

例：采用 C#和 MSMQ 实现基于内容的路由器

下面的代码示例实现了一个非常简单的基于内容的路由器，它根据消息体中的第一个字符来完成消息的路由。如果消息体文本以 W 开始，路由器就把消息路由给 widgetQueue（零部件队列）；如果消息体以 G 打头，路由器就把消息路由给 gadgetQueue（配件队列）。否则，路由器把消息发送给 dunnoQueue（未知队列），该队列是一个非法消息通道。这个路由器是无状态的，在做出路由决定时，它并没有“记”以前的任何消息。

```
class ContentBasedRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue widgetQueue;
    protected MessageQueue gadgetQueue;
    protected MessageQueue dunnoQueue;

    public ContentBasedRouter(MessageQueue inQueue, MessageQueue widgetQueue,
                              MessageQueue gadgetQueue, MessageQueue dunnoQueue)
    {
        this.inQueue = inQueue;
        this.widgetQueue = widgetQueue;
        this.gadgetQueue = gadgetQueue;
        this.dunnoQueue = dunnoQueue;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }
```

```
    private void OnMessage(Object source,
ReceiveCompletedEventArgs asyncResult)
    {

        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new System.Messaging.XmlMessageFormatter
                           (new String[] {"System.String,mscorlib"});
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        if (IsWidgetMessage(message))
            widgetQueue.Send(message);
        else if (IsGadgetMessage(message))
            gadgetQueue.Send(message);
        else
            dunnoQueue.Send(message);
        mq.BeginReceive();
    }

    protected bool IsWidgetMessage (Message message)
    {
        String text = (String)message.Body;
        return (text.StartsWith("W"));
    }

    protected bool IsGadgetMessage (Message message)
    {
        String text = (String)message.Body;
        return (text.StartsWith("G"));
    }
}
```

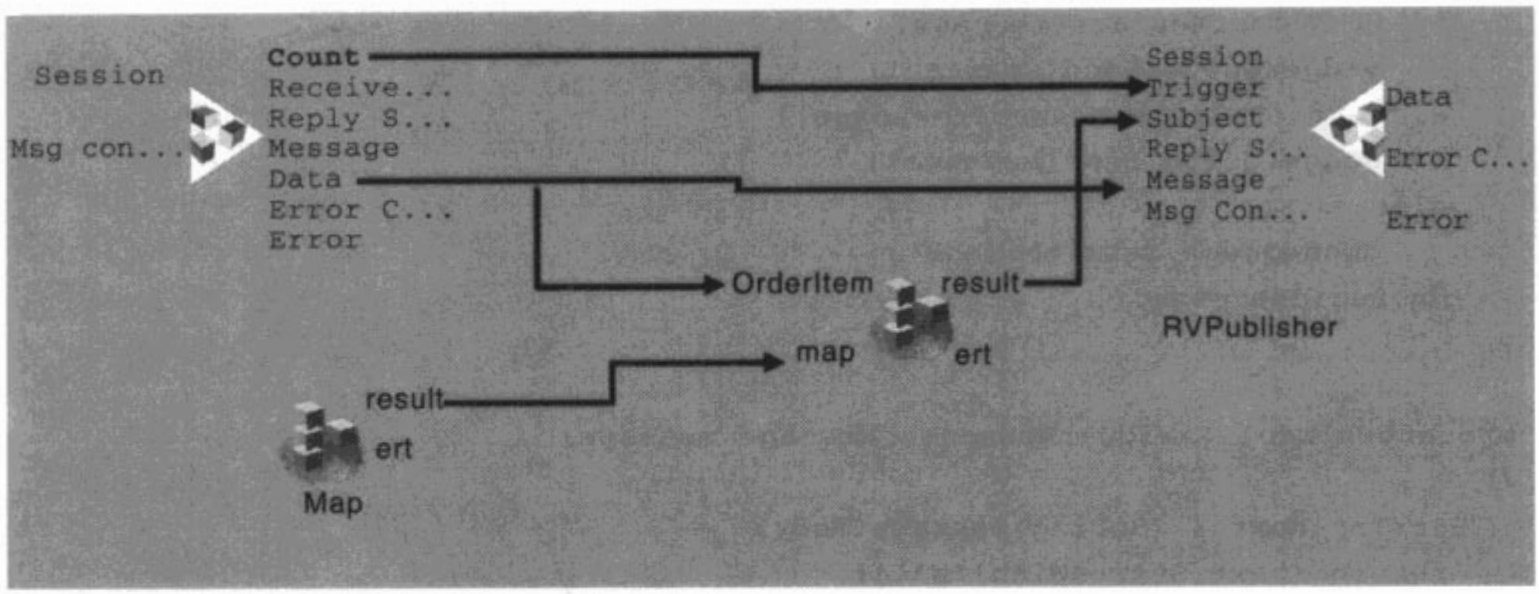
这个例子把 OnMessage 方法注册为消息处理函数，到达 inQueue 队列的消息要由这个函数来处理，因此这里使用了事件驱动的消息消费者。当有消息到达 inQueue 时，.NET 框架就会调用 OnMessage 方法。消息队列的 Formatter 属性告诉框架应该等待什么类型的消息。在这个例子中，我们只处理简单的字符串消息。OnMessage 得出如何路由消息，并调用队列的 BeginReceive 方法来通知.NET 它已经准备好接收下一条消息。为了尽量减少代码，这个简单的路由器并不是事务性的：如果路由器从输入通道消费了一条消息，但是在将这条消息发布到输出通道之前路由器崩溃了，我们就会丢失一条消息。后面的章节将介绍如何建立事务性端点（见“事务性客户”）。

例：TIBCO 消息代理（MessageBroker）

消息路由是一个很普遍的需求，因此大多数 EAI 工具集均提供了内置的工具，以简化建立路由逻辑。例如，在 C#的例子中，我们要编写许多逻辑，需要从输入队列中读取消息、

逆串行化消息、分析消息，并重新把消息发布到正确的输出通道上。在许多 EAI 工具中，这种逻辑可以用简单的拖放操作来实现，而不用编写代码。只有一处需要编写代码，这就是基于内容的路由器做出路由决定所采用的具体逻辑。

TIBCO 的 ActiveEnterprise 工具集就是这种实现了消息路由的 EAI 工具。该工具集包括 TIB/MessageBroker，它能创建简单的消息流，其中包括消息转换和路由功能。在 TIB/MessageBroker 中实现时，同一个零部件路由器能根据商品编号的第一个字母完成输入消息的路由，如下图所示。



我们可以从左至右来看消息流。左边的组件（由一个方向指向右边的三角形表示）是订购者组件，它从通道 router.in 上消费消息。通道名在一个属性框中指定（图中没有显示这个属性框）。消息的内容交给消息发布者（由位于屏幕右边的三角形表示）。从订购者的 Data（数据）输出到发布者的 Message（消息）输入之间有一条连线，这条连线表示基于内容的路由器不会修改消息体中的内容。为了确定正确的输出通道，函数 ComputeSubject（位于中间）会分析消息的内容。该函数使用了一个所谓的“字典”（在图中标为“Map”），作为消息内容到目标通道名之间的一个转换表。这个字典有如下表所示的配置。

商品代码	通道名称
G	gadget（配件）
W	widget（零部件）

ComputeSubject 函数使用输入消息中订单商品编号的第一个字母，从字典中查找目标通道。为了形成输出通道的完整名，它把字典查找的结果附加到 router.out 字符串后面，从而得到形如 router.out.widget 的通道名。这个计算结果要传递给右边的发布者组件，用作为通道名。因此，编号以 G 字母打头的所有订单商品都路由到 router.out.gadget 通道，而编号以 W 字母打头的所有订单商品则路由到 router.out.widget 通道。

ComputerSubject 函数的 TIBCO 实现如下所示：

```
concat("router.out.",DGet(map,Upper(Left(OrderItem.ItemNumber,1))))
```

该函数把订单商品编号的第一个字母抽取出来（使用 Left 函数），并将其转换为大写（使用 Upper 函数）。然后使用转换后的结果作为键在字典中查找，从而获得输出通道名（使用 DGet 函数）。

这个例子演示了商用 EAI 工具的作用。只需要编写一个简单的函数而不是几十行代码，就同样能实现零部件路由器功能。此外，我们还能免费获得诸如事务性、线程管理和系统管理等特性。但是这个例子也反映出采用图形化工具表述一种解决方案是有难度的。许多重要的设置被隐藏在属性字段中，无法在屏幕上显示出来。所以使用图形化开发工具建立的解决方案很难用文档描述。

7.3 消息过滤器



在订单处理示例的基础上，假设公司的管理部门决定把价格的变动和促销信息发布给大量客户。可以在商品价格发生变动时，发送一条消息通知客户。同样，如果有特殊的促销活动，如 12 月份销售的所有零部件都打 10%的折扣，也可以向客户发送一条消息。有些客户可能只对指定商品的价格变动或促销活动感兴趣。例如，如果我主要购买的是配件，那么对零部件是否降价则不感兴趣。

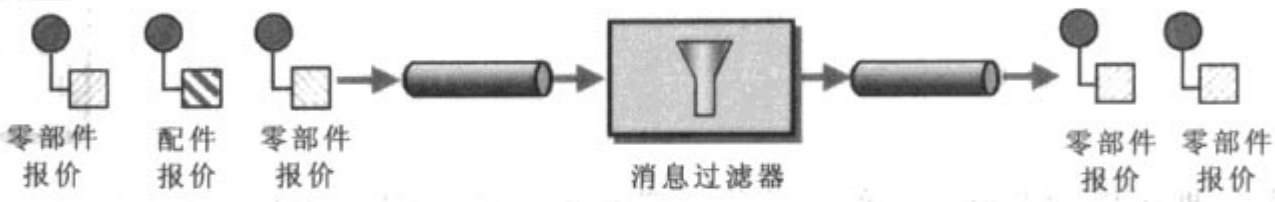
如何避免组件接收到与之无关的消息？

要让组件只接收与之相关的消息，最基本的方法是只订购那些有相关消息的通道。这种方法利用了发布-订购通道固有的路由能力。组件所订购通道上的消息才会被该组件接收。例如，可以为零部件价格更新创建一个通道，而为配件的价格更新创建另一个通道。客户可以自由订购其中的一个通道或两个通道。这样有一个优点，新的订购者可随时加入进来，而不必对系统做任何修改。但是订购发布-订购通道时，通常会存在一种“全有或全无”的限制：如果组件订购了一个通道，就会接收该通道的所有消息。要想获得更细粒度的控制，唯一的方法是创建更多的通道。如果我们要处理多个参数的一个组合，那么通道的数量就会呈爆炸式增长。例如，如果允许客户选择接收零部件或配件的价格下调 5%、10%或 15%的所有通知消息，就需要 6 个通道（2 种商品类型乘以 3 种折扣）。采用这种方法，最终将导致系统难以管理，而且会因为分配大量通道而耗费重要的资源。因此我们必须另外找出一种方法，来得到比通道订购更大的灵活性。

该解决方案还应当能适应频繁的价格变动。例如，我们可以修改基于内容的路由器，使它能把消息路由给多个目标（有关概念在“接收表”中介绍）。这种预测型路由器只是向各个接收者发送与之相关的消息，这样接收者就不必再采取其他额外的步骤。但是，消息的创建者要维护每个订购者的偏好（即每个订购者对哪些消息感兴趣）。如果接收者列表或者接收者的偏好信息频繁变化，那么这个解决方案的维护将是一场噩梦。

还有一种简单的方法，可以把改变广播给所有的组件，让每个组件过滤掉自己不需要的消息。但是，使用这种方法的前提条件是我们要能控制这些具体的组件。在很多集成环境中，由于所集成的组件是封装好的应用、遗留系统或无法由我们控制的应用，因此无法采用这种解决方法。

使用一种特殊的消息路由器：消息过滤器，可以根据一组规则把不需要的消息从通道中剔除。



消息过滤器是一种只有一个输出通道的消息路由器。如果输入消息的内容与消息过滤器指定的规则匹配，该消息将路由到输出通道。如果消息的内容与规则不匹配，这个消息则被丢掉。

在我们的例子中，定义了一个发布-订购通道，每个客户都能自由监听该通道。客户可以采用消息过滤器来剔除消息，在此要根据客户选择的规则，如按照商品的类型或价格变动的幅度来决定是否剔除消息。

消息过滤器可以描述为一种特殊的基于内容的路由器，它会把消息路由到输出通道或者一个空通道（null channel）。发布到空通道的所有消息都会丢掉。这种通道的作用类似于许多操作系统中提供的/dev/null 目标文件，或空对象（null object）[PLoPD3]。

7.3.1 无状态与有状态的消息过滤器

零部件和配件的例子描述了一个无状态的消息过滤器：消息过滤器检查单个的消息，只是根据包含在消息中的信息来确定是否要传递这个消息。因此，消息过滤器不需要维护消息间的状态信息，所以它是无状态的。无状态的组件允许我们同时运行组件的多个实例，以加快处理的速度。但是，消息过滤器不一定是无状态的。例如，有些情况下需要使用消息过滤器跟踪消息的历史。一种常见的情况是使用消息过滤器消除重复的消息。假设每条消息都有唯一的消息标识符，消息过滤器将存储所有已传递消息的标识符，通过把每条消息的标识符与所存储的消息标识符列表进行比较，就能识别出重复的消息。

7.3.2 在消息传递系统中建立过滤功能

有些消息传递系统在消息传递基础设施中集成了消息过滤器的功能。例如，一些发布-订购系统允许定义层次结构的发布-订购通道。许多发布-订购系统（包括大多数 JMS 实现）都支持这项功能。可以把促销信息发布到 wgco.update.promotion.widget 通道。然后订购者可以使用通配符来订购特定的消息子集。例如，如果订购者在监听 wgco.update.*.widget 主题，他将接收到所有与零部件相关的更新消息（包括促销和价格变动消息）。另一个订购者可能监听 wgco.update.promotion.*，该通道将发布所有与零部件和配件的促销活动相关的信息，但不包括价格变动信息。基于通道层次结构，可以通过附加限定参数来明确通道的语义，客户可以指定额外的规则作为通道名的一部分来过滤消息，而不必订购所有的更新消息。不过，尽管层次式的通道命名可以提供一定的灵活性，但与消息过滤器相比灵活性仍然很有限。例如，消息过滤器可以根据价格变化是否达到 11.5%来决定是否传递价格变动消息，而利用通道名是很难表达这种含义的。

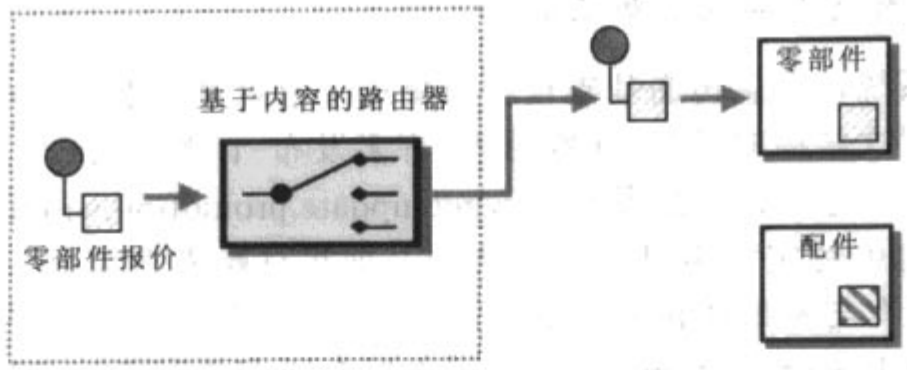
还有一些消息传递系统提供了 API，可以在接收应用中实现选择性消费者。消息选择器是一种表达式，在应用看到消息之前，消息选择器能计算输入消息中的首部或属性元素。如果选择条件计算的结果为假，此消息就会被忽略，而不会传递给应用逻辑。消息选择器实际上就是建立在应用中的消息过滤器。但是如果使用消息选择器，仍然需要修改应用（而在 EAI 中修改应用往往是不可能的），选择规则放在消息传递基础设施中执行。消息过滤器与选择性消费者的根本区别是，使用选择性消费者模式的消费者不会消费与指定规则不匹配的消息，而消息过滤器会将所有的消息都从输入通道中取出，并且只把与指定规则匹配的消息发布到输出通道上。

由于选择性消费者要在消息传递基础设施中注册过滤表达式，因此基础设施可以根据过滤规则来做出聪明的内部路由决策。假设消息接收者与消息的产生者分别位于不同的网段（甚至跨 Internet）。如果好不容易把消息路由给消息过滤器，最后只是发现要丢掉这个消息，这样就太浪费了。另一方面，我们想使用消息过滤器机制，从而使接收者能够控制消息的路由，而不是仅由一个中心消息路由器控制消息路由。如果消息过滤器是消息传递基础设施提供给消息订购者的 API 的一部分，那么基础设施就能自由地将过滤表达式转移到更靠近消息源的地方。这样不仅能保持消息订购者对消息路由的控制，还能帮助消息传递基础设施避免不必要的网络通信。这类似于动态接收表的行为。

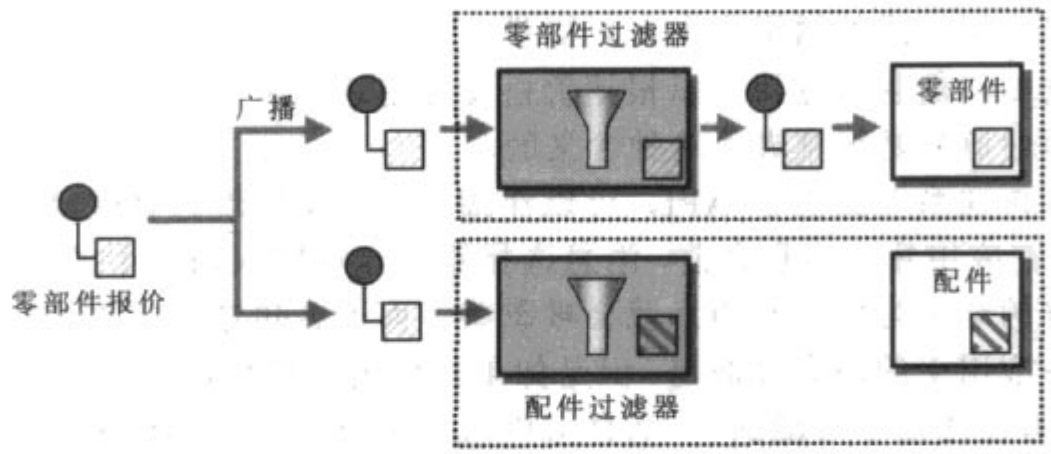
7.3.3 使用消息过滤器实现路由功能

把发布-订购通道连接到一组消息过滤器，可以消除不需要的消息，与基于内容的路由器类似，能实现与之相当的功能。后面的图给出了这两种实现选择。

在这个简单的示例中，有两个接收者：接收者 Gadgets 只对配件消息感兴趣，而接收者 Widgets 只对零部件消息感兴趣。基于内容的路由器会检查每条消息的内容，并预测性地把消息路由给相应的接收者。



选择 1：使用基于内容的路由器



选择 2：使用广播通道和一组消息过滤器

第二种选择把消息广播到发布-订购通道。每个接收者都使用一个消息过滤器来消除它不需要的消息。例如，零部件接收者采用零部件过滤器过滤出零部件消息。

下表列出了这两种解决方案间的一些不同点。

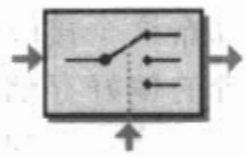
基于内容的路由器	带消息过滤器的发布-订购通道
每条消息只有一个消费者	多个消费者可消费同一条消息
集中控制和维护——预测型路由	分布式控制和维护——反应型过滤
路由器需要了解参与者。如果增加或删除了参与者，可能要更新路由器	不需要了解参与者。增加或删除参与者很方便
常用于商业事务处理，如订单处理	常用于事件通知或信息型消息
采用基于队列的通道通常效率更高	采用发布-订购通道效率更高

那么应该使用哪种选择方案呢？在有些情况下，要根据所需的功能做出决策。例如，如果想让多个接收者处理相同的消息，就应该使用带消息过滤器的发布-订购通道。但是在大多数情况下，应该根据由谁控制（和维护）路由决策来选择解决方案。是集中控制还是把它交给接收者控制？如果消息中包含只能由某些接收者查看的敏感数据，就应该使用基于内容的路由器，而不能让其他接收者过滤消息。例如，假设我们想为优惠客户提供特殊

的折扣，如果把这些消息发送给其他非优惠客户，并指望他们能忽略这些特殊的优惠信息，这是不可能的。

网络通信量的考虑也会影响我们做出的选择。如果能以高效的方式广播信息（例如在内部网络上使用 IP 组播），那么使用过滤器不但效率高，还能避免采用单个路由器可能带来的瓶颈问题。不过，如果信息是通过 Internet 路由的，就只能使用点对点连接。在这种情况下，采用单个路由器的效率更高，因为不必将每条消息发送给所有接收者（而不管这些接收者是否对消息感兴趣）。如果我们想把控制权交给接收者，但是出于网络效率的原因又想使用路由器，就可以使用动态接收表。这种接收表可作为动态路由器，接收者能向路由器发送控制消息来表达它们的偏好。接收表把接收者的偏好存储在数据库或规则基中。当消息到达时，接收表把与规则匹配的消息转发给所有感兴趣的接收者。

7.4 动态路由器



你正在使用消息路由器在多个目标之间实现消息的路由。

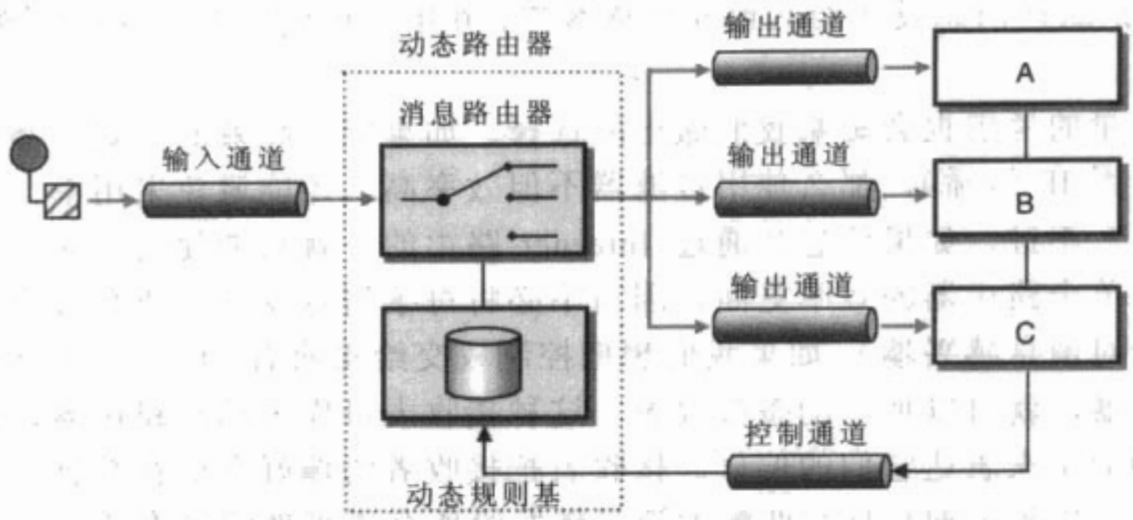
如何避免路由器对所有可能目标的依赖，同时还能保持其效率？

消息路由器效率很高，这是因为它能把消息直接路由给正确的目标。其他消息路由解决方案（尤其是反应型过滤解决方案）的效率就没有这么高，因为它们使用的是尝试-纠错方法：例如，路由表会把每条消息路由给第一个可能的目标。如果该目标是正确的接收者，就接受这个消息；否则，消息将被传递给下一个可能的目标，依此类推。同样地，基于消息过滤器的方法则把消息发送给所有可能的接收者，而不考虑它们是否对消息感兴趣。

分布式的路由解决方案还要承担另外一些风险，如存在多个消息接收者，或者根本没有接收者。在这两种情况下，除非使用集中的路由组件，否则无法检查出可能存在的错误。

为了实现准确路由，需要使用一个消息路由器，这个消息路由器不但要了解各个目标，还要了解把消息路由给目标的有关规则。但是，如果可能的目标频繁变更，这会使消息路由器的维护成为负担。

使用动态路由器，基于参与目标提供的特殊配置消息，这个路由器可以自行配置。



除了使用输入和输出通道，动态路由器还使用了一个附加的控制通道。系统启动时，每个潜在的接收者（可能的接收者）都利用该控制通道向动态路由器发送一条特殊的消息，宣布这个接收者的存在，并给出此接收者所处理消息的路由条件。动态路由器把每个参与者的偏好存储在一个规则基中。当有消息到达时，动态路由器会计算所有规则，并把消息路由给符合规则的接收者。它能实现了高效的预测型路由，而在维护动态路由器时不必依赖于各个潜在的接收者。

在大多数基本的集成环境中，每个参与者都会在启动时刻向动态路由器宣布自己的存在信息和路由偏好。这要求每个参与者都必须知道由动态路由器使用的控制队列。它还要求动态路由器会持久地存储路由规则。否则，如果动态路由器失败而不得不重启，它将无法恢复路由规则。或者，动态路由器可以向所有可能的参与者广播一条消息，触发这些参与者返回控制消息作为应答。采用这种配置方案更可靠一些，但是这需要使用另外一个发布-订购通道。

还可以改进控制通道，以允许参与者既可以把订购消息发送给动态路由器，也可以向动态路由器发送取消订购消息。这样一来，参与者就能在运行时把自己添加到路由规划中，或者从路由规划中把自己删除。

由于接收者彼此之间相互独立，动态路由器必须应对可能出现的规则冲突，如多个接收者都宣布对相同类型的消息感兴趣。动态路由器可以采用多种不同的策略解决这样的冲突：

1. 忽略与已有消息存在冲突的控制消息。这个策略可以确保路由规则是无冲突的。但是，路由表的状态可能取决于潜在接收者的启动顺序。如果所有接收者在同一时刻启动，由于所有接收者同时向控制队列宣布自己的偏好，可能会导致不可预见的行为。

2. 把消息发送给与规则匹配的第一个接收者。该策略允许路由表中包含相互冲突的接收者，但是在消息到达时可以解决这些冲突。

3. 把消息发送给与规则匹配的所有接收者。这种做法可容忍冲突的存在，但是它把动态路由器转换成了接收表。通常，基于内容的路由器采取的操作是为每条输入消息发布一条输出消息，而这个策略违背了这一原则。

动态路由器的主要缺点是这种解决方案比较复杂，并且调试动态配置的系统存在一定的困难。

基于消息的中间件可以实现底层 IP 网络类似的功能，动态路由器就是这样一个例子。IP 路由使用动态路由表在网络间路由 IP 数据包，而动态路由器的工作与动态路由表非常相似。接收者配置动态路由器的协议对应于 IP 路由信息协议（Routing Information Protocol，RIP），详细内容可参考[Stevens]。

在面向服务的体系结构中，动态路由器经常用于动态服务发现。如果客户应用想访问某项服务，它就把包含服务名的消息发送给动态路由器。动态路由器维护着一张服务目录列表，其中包括所有服务的名字及服务所监听的通道。路由器根据来自每个服务提供者的控制消息来建立这个目录列表。当有服务请求到达时，动态路由器使用服务目录查找服务名，然后把消息路由给正确的通道。这样一来，客户应用就可以把命令消息发送到单一的通道上，而不必考虑特定服务提供者的特性或位置，即使提供者发生改变也没有影响。

还有一个相关的模式，即客户-分派器-服务器模式（Client-Dispatcher-Server Pattern）[POSA]，它允许客户请求特定的服务，而不必了解服务提供者的实际位置。分派器使用已注册服务的列表来建立连接，即在客户与实现所需服务的物理服务器之间建立一个连接。动态路由器完成了类似的功能，但是它与分派器不同，因为它使用的是更智能的路由策略，而不是简单的表查找。

例：使用 C#和 MSMQ 实现动态路由器

这个例子建立在前面基于内容的路由器示例基础上，并将其改进为动态路由器。这个新组件监听两个通道：inQueue 和 controlQueue。控制队列可以接收格式为 X:QueueName 的消息，因此，动态路由器会把消息体文本以字母 X 打头的所有消息路由给 QueueName 队列。

```
class DynamicRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue controlQueue;
    protected MessageQueue dunnoQueue;

    protected IDictionary routingTable = (IDictionary)(new Hashtable());


    public DynamicRouter(MessageQueue inQueue, MessageQueue controlQueue,
                         MessageQueue dunnoQueue)
    {
        this.inQueue = inQueue;
        this.controlQueue = controlQueue;
        this.dunnoQueue = dunnoQueue;
```

```
        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();

        controlQueue.ReceiveCompleted +=
            new ReceiveCompletedEventHandler(OnControlMessage);
        controlQueue.BeginReceive();
    }

    protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new System.Messaging.XmlMessageFormatter
                           (new String[] {"System.String,mscorlib"});

        Message message = mq.EndReceive(asyncResult.AsyncResult);

        String key = ((String)message.Body).Substring(0, 1);

        if (routingTable.Contains(key))
        {
            MessageQueue destination  = (MessageQueue)routingTable[key];
            destination.Send(message);
        }
        else
            dunnoQueue.Send(message);
        mq.BeginReceive();
    }

    // 控制消息的格式是 X:QueueName，这是一个字符串
    protected void OnControlMessage(Object source,
ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new System.Messaging.XmlMessageFormatter
                           (new String[] {"System.String,mscorlib"});
        Message message = mq.EndReceive(asyncResult.AsyncResult);

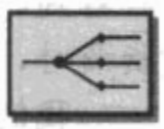
        String text = ((String)message.Body);
        String [] split = (text.Split(new char[] {':'}, 2));
        if (split.Length == 2)
        {
            String key = split[0];
```

```
            String queueName = split[1];
            MessageQueue queue = FindQueue(queueName);
            routingTable.Add(key, queue);
        }
        else
        {
            dunnoQueue.Send(message);
        }
        mq.BeginReceive();
    }

    protected MessageQueue FindQueue(string queueName)
    {
        if (!MessageQueue.Exists(queueName))
        {
            return MessageQueue.Create(queueName);
        }
        else
            return new MessageQueue(queueName);
    }
}
```

这个例子使用了一种非常简单的冲突解决机制，即最后的接收者取胜。如果两个接收者都对以字母 X 打头的消息感兴趣，那么只有第二个接收者能接收到这样的消息，因为 Hashtable 中只为每个不同的键值存储一个队列。还要注意，dunnoQueue 队列现在可以接收两种类型的消息：与路由规则不匹配的输入消息，或是与所需格式不匹配的控制消息。

7.5 接收表



基于内容的路由器可以根据消息的内容把消息路由给正确的系统。该过程对最初的发送者透明，这样一来，消息的产生者只需把消息发送给通道，让路由器选择消息并处理消息的路由。

但是在有些情况下，我们可能想为消息指定一个或多个接收者。这类似于大多数电子邮件系统中实现的接收列表。对每条电子邮件消息，发送者都可以指定一个接收者列表。邮件系统将确保把消息内容传递给每个接收者。企业集成领域中也有类似的例子，如一个功能可能由一个或多个提供者实现。例如，我们可以与多个信用代理商签订合同，让他们帮助我们评估客户的信用等级。当某个小订单到达时，只需把信用请求消息路由给某个信

用代理商即可。如果客户下了一个很大的订单，就要把信用请求消息路由给多个信用代理商，并在比较它们返回的结果之后才能做出决定。在这种情况下，接收者列表是由订单的价值额度决定的。

在另一种情况下，我们可能想把订单消息路由给挑选出的一组供货商，以获得所请求商品的报价信息。我们并不想把请求发送给所有的供货商，而是想根据用户的偏好有所控制，只有指定的供货商才能接收到请求。

如果接收者列表会动态变化，如何把消息路由给这样一组接收者？

这个问题是对基于内容的路由器所解决问题的扩展，因此基于内容的路由器模式中采用的一些措施和方法也可以用来解决这个问题。

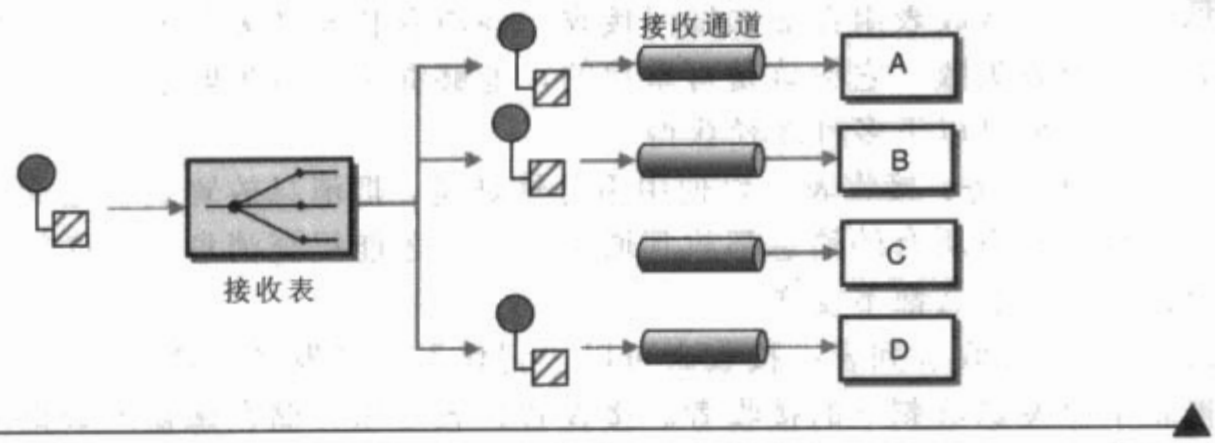
大多数消息传递系统都提供了发布-订购通道，它能把所发布的各个消息的副本发送给每个订购该通道的接收者。接收者集合由对指定通道或主体的订购来决定。但是，通道的当前订购者列表通常是静态的，无法基于各条消息来控制接收者列表。我们需要的通道应当类似于发布-订购通道，而且能把每条消息发送给一组不同的订购者。要实现这样的通道很困难，因为发布-订购通道只提供了两种选择：要么订购通道中的所有消息，要么什么也不订购。

每个潜在的接收者可以根据消息的内容对输入消息进行过滤，它们大多会使用消息过滤器或选择性消费者。遗憾的是，这种解决方案会把由谁接收消息的逻辑分布到各个订购者，从而增加了维护的负担。为了保持集中控制，可以在每条消息附带的列表中指定这个消息所面向的接收者。这样一来，当消息被广播给所有可能的接收者时，不在接收者列表中的各个接收者可以丢掉该消息。

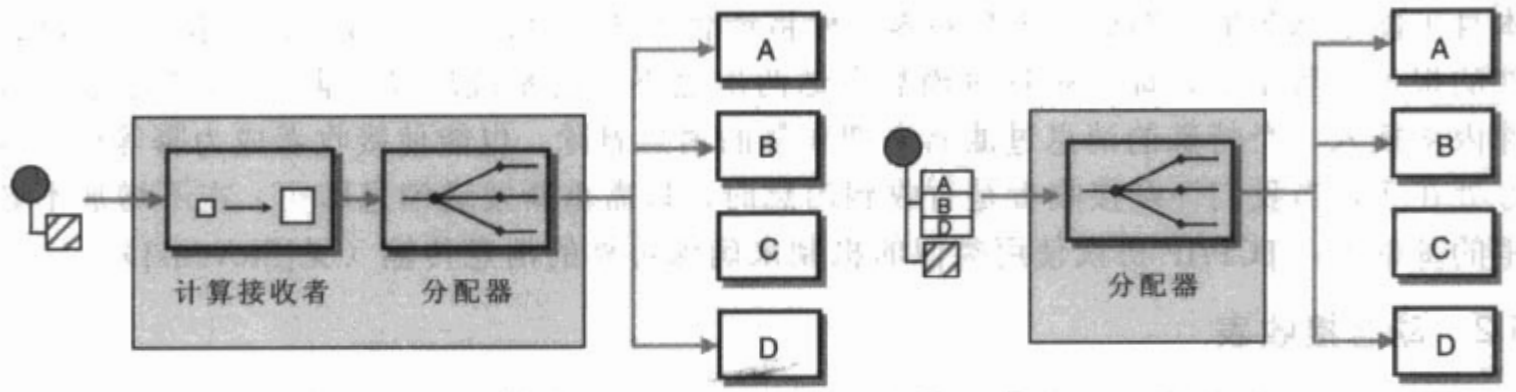
采用这种方法的缺点是效率很低：每个潜在的接收者必须处理每一条消息，尽管在大多数情况下接收者都会决定丢掉该消息。并且，这种配置方案还要求接收者保证一定的“自觉性”，因为每个接收者都可以自行处理本不该处理的消息。如果要求消息不能被某些接收者看到，采用这种解决方案肯定就是不合适的。例如在把报价请求转发给所选择的一组供货商时，往往希望其他供货商接到这些报价请求时能将其忽略。

我们可能还需要消息的产生者能把消息单独地发送给每个期望的接收者。不过在这种情况下，向所有接收者传送消息的负担就会转嫁到消息产生者那里。如果产生者是一个封装好的应用，则无法实现这一功能。而且，如果把路由决策的功能嵌入到应用的内部，就会使应用与集成基础设施的耦合更加紧密。在许多情况下，所集成的应用甚至不知道它们要参与到某个集成解决方案中，因此想在应用中包含消息路由逻辑是不现实的。

为每个接收者定义一个通道。然后使用接收表模式检查输入消息，确定需要消息的接收者列表，并把消息转发给与列表中接收者关联的所有通道。



嵌入到接收表中的逻辑可以用两个单独的部分来描述，不过它们的实现一般都耦合在一起。第一个部分计算接收者列表。第二个部分只是遍历接收者列表，并将所接收消息的副本发送给各个接收者。类似于基于内容的路由器，接收表通常并不修改消息的内容。



接收表可以计算出接收者列表（图中左边所示）或让另一个组件提供接收者列表（图中右边所示）

接收者列表可以从许多地方获得。列表的创建不一定在接收表中实现，消息的产生者或其他组件可将列表附加到输入消息中。在这种情况下，接收表只是复制这个只读列表。通常，接收表会把列表从消息中删除，以减少输出消息的大小，防止各个接收者看见列表中的其他接收者。如果每条消息的目标是由外部因素确定的，如由用户选择确定，就完全可以在输入消息中附带接收者列表。

在大多数情况下，接收表要根据消息的内容以及内嵌在接收表中的一组规则来计算接收者列表。这些规则可能是硬编码的，也可能是可配置的（见下一页的介绍）。

与消息路由器中介绍的情况一样，接收表也要考虑耦合问题。如果采用预测型方法将消息路由给各个接收者，将在组件之间形成更紧密的耦合，因为中心组件必须掌握其他一系列组件的信息。

为了让接收表控制信息流，必须确保接收者不能直接订购接收表中的输入通道，而绕过接收表所做的控制。

7.5.1 健壮性

接收表组件负责把输入消息发送给接收者列表中指定的各个接收者。作为一个健壮的接收表实现，在处理输入消息时，只能在成功发送出所有输出消息之后才会“消费”掉这

条消息。因此，接收表组件必须确保接收一条消息和发送多条消息在一个完整的原子操作中。如果接收表失败，它应该是可重启的，能够继续完成失败前正在处理的操作。接收表的健壮性可以通过以下多种途径获得：

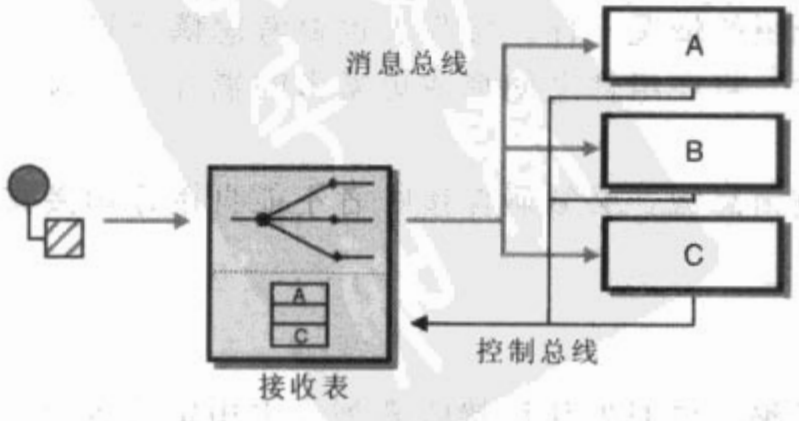
1. 单一的事务：接收表可以使用事务性通道，把消息放置到输出通道中属于同一个事务的一部分。只有所有的消息都放到通道中后，才能提交消息。这样就能保证所有消息要么被全部发送，要么都不发送。

2. 持久的接收者列表：接收表可以“记住”已经发送的消息，因此当失败后重启时，它还能把消息发送给剩下的接收者。接收者列表可以存储在磁盘或数据库中，即使接收表组件崩溃也能保留下来。

3. 幂等（Idempotent）接收者：接收表只需在重启后重新发送所有的消息。采用这种方式，则要求所有潜在的接收者都是幂等的（见“幂等接收者”）。作为幂等函数，其调用不会改变系统的状态。也就是说，如果相同的消息被处理两次，组件的状态不会受到影响。消息本身可能是幂等的（例如“所有的零部件将销售到5月30日止”或“请给我提供XYZ零部件的报价”等消息，即使把这种消息发送两次也不会给系统造成损害），或者通过在接收组件内部插入一个特殊的消息过滤器来把重复的消息删除，也能使接收者成为幂等的。幂等的好处在于，当我们怀疑接收者是否收到消息时，只需重新发送消息即可。在不增加不必要开销的基础上，TCP/IP协议使用类似的机制来确保可靠的消息传输（见[Stevens]）。

7.5.2 动态接收表

尽管使用接收表的目标是为了维持控制，不过，让接收者自己来配置存储在接收表中的规则可能很有用——例如，接收者可能想根据规则订购特定的消息，但这些规则很难采用发布-订购通道主题的形式加以表示。在介绍消息过滤器模式时，我们提到过这种订购规则，如“只接收价格低于48.31美元的消息”。为了尽可能减少网络通信量，我们还是想只把消息发送给感兴趣的参与者而不是采用消息广播的形式，让每个接收者自己决定是否处理该消息。为了实现这种功能，接收者可以通过特殊的控制通道把他们的订购偏好发送给接收表。接收表把偏好存储在规则基中，并由此建立每条消息的接收者列表。采用这种方法，订购者可以控制消息的过滤，但不会影响接收表分配消息的效率。这种解决方案结合了动态路由器和接收表的特性，形成了动态接收表（如下图所示）。



动态接收表由接收者通过控制通道进行配置

这种方法对消息过滤器模式中讨论的价格更新示例很适用。不过对于刚开始介绍该模式时提到的报价示例，我们想要控制参与竞价的供货商，由于这种方法要把控制权分派给每个接收者，因此对这个例子不适用。

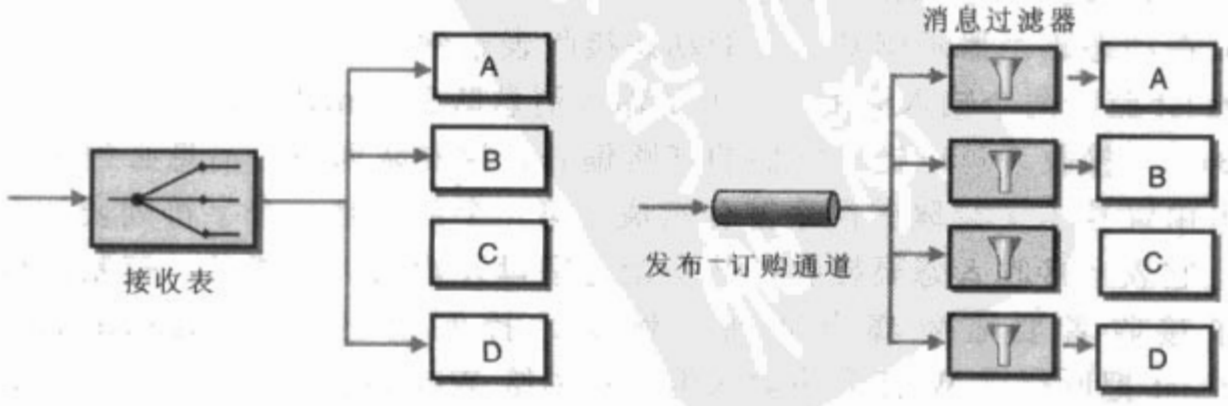
7.5.3 网络效率

把一条消息发送给所有可能的接收者让这些接收者自己过滤消息，还是把每条消息发送给不同的接收者，这两种做法中哪一个更有效，这从很大程度上取决于消息传递基础设施的实现。通常我们认为消息的接收者越多，由此产生的网络通信量就越多。但是也有例外。对于那些基于 IP 组播功能的消息传递系统来说就不是这样，它们能在一次网络传输中把消息路由给多个接收者（只是在消息丢失时才需要重传）。IP 组播利用了以太网的总线结构。当通过网络发送一个 IP 包时，位于同一网段上的所有网络适配器（NIC）都能接收到这个 IP 包。正常情况下，NIC 会检查 IP 包的接收地址，如果该地址与 NIC 关联的 IP 地址不匹配，则忽略这个 IP 包。组播路由允许所有属于指定组播组的接收者读取总线上的 IP 包。这样，单个数据包能被多个 NIC 接收，再由这些 NIC 把数据转交给与网络连接关联的每个应用。在局域网中使用这种方法效率很高，这是由以太总线结构决定的。如果要跨 Internet，这需要采用点对点的 TCP/IP 连接，就无法使用这种方法。通常，接收者的距离越远，使用接收表而不是发布-订购通道的效率就越高。

采用广播方式的效率是否比采用接收表高不仅仅取决于网络基础设施本身，要处理消息的接收者占全部接收者中多大的比例也是一个决定因素。如果大多数接收者都包含在接收者列表中，那么采用消息广播方式的效率可能更高，因为只有很少的非参与者才需要过滤消息。但是，如果在所有可能的接收者中只有少部分接收者对特定的消息感兴趣，那么采用接收表通常能取得更高的效率。

7.5.4 接收表与发布-订购及过滤器模式的比较

使用接收表可以实现预测型路由，使用发布-订购通道和消息过滤器阵列可以实现反应型过滤，我们已经多次采用这两种方法对比实现了相同的功能。在这二者之间如何选择，有一些决策规则，在比较基于内容的路由器和消息过滤器阵列时，其中的一些决策规则也同样适用。不过，在使用接收表时，消息可以传送给多个接收者，因此更倾向于采用过滤方法。



接收表与消息过滤器阵列的比较

下表比较了这两种解决方案。

接收表	带消息过滤器的发布-订购通道
集中控制与维护——预测型路由	分布式控制与维护——反应型过滤
路由器要掌握参与者的信息。如果增加或删除了参与者，路由器可能需要更新（除非使用动态路由器，但是要以失去控制权为代价）	不需要掌握参与者的信息。增加或删除参与者很方便
常用于商业事务处理，如请求报价信息	常用于事件通知或传递信息型消息
如果只采用基于队列的通道，效率会更高	如果采用发布-订购通道，效率会更高（具体由基础设施决定）

如果把消息发送给多个接收者，之后可能要把返回的结果组合起来。例如，如果我们把请求信用等级的消息发送给了多个信用代理商，那么只有等待所有结果返回后，才能比较结果并精确计算出平均信用等级。对于其他不是那么重要的功能，可以只采用第一个返回的响应，以优化消息吞吐量。诸如此类的策略一般是由聚合器实现的。分散收集器描述的情形是先把单个的一条消息发送给多个接收者，并把返回的响应重新组合为一个消息。

如果消息传递系统只提供了点对点通道，而没有提供发布-订购通道，可以使用动态接收表来实现发布-订购通道。接收表中保留有一个点对点通道的列表，其中包括主题订购的所有通道。每个主题可由一个特定的接收表实例表示。如果要应用特殊的规则，从而让接收者订购一个特定的数据源，这种解决方案也很有用。大多数发布-订购通道都允许任何组件订购通道，但是接收表只要限制订购者列表，就能非常方便地实现对数据源的访问控制逻辑。当然，前提条件是消息传递系统能防止接收者直接访问接收表中的输入通道。

例：贷款中介

第9章“中场演练：复合消息传递”展示了组合消息传递的一个实例，它使用了接收表，以只把贷款利率请求路由给符合条件的银行。其中还给出了用 Java、C#和 TIBCO 实现的接收表。

例：采用 C#和 MSMQ 实现的动态接收表

这是一个建立在动态路由器示例基础上的动态接收表示例。它们的代码结构非常相似。DynamicRecipientList 监听两个输入队列，一个是输入消息队列（inQUeue），另一个是控制队列（controlQueue），接收者可以提交它们的订购偏好。控制队列中的消息必须采用字符串格式，该字符串由冒号（:）分隔的两个部分组成。第一个部分是一个字符列表，表示接收者的订购偏好。它表示接收者想要接收以某个指定字母开始的所有消息。控制消息的第二个部分指定了接收者要监听哪个队列。例如，控制消息 W:WidgetQueue 告诉 DynamicRecipientList 把所有以 W 打头的输入消息路由给 WidgetQueue 队列。同样地，控制消息 WG:WidgetGadgetQueue 要求 DynamicRecipientList 把所有以 W 或 G 打头的消息路

由给 WidgetGadgetQueue 队列。

```
class DynamicRecipientList
{
    protected MessageQueue inQueue;
    protected MessageQueue controlQueue;

    protected IDictionary routingTable = (IDictionary)(new Hashtable());


    public DynamicRecipientList(MessageQueue inQueue, MessageQueue controlQueue)
    {
        this.inQueue = inQueue;
        this.controlQueue = controlQueue;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();

        controlQueue.ReceiveCompleted +=
            new ReceiveCompletedEventHandler(OnControlMessage);
        controlQueue.BeginReceive();
    }

    protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new System.Messaging.XmlMessageFormatter
                           (new String[] {"System.String,mscorlib"});
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        if (((String)message.Body).Length > 0)
        {
            char key = ((String)message.Body)[0];

            ArrayList destinations  = (ArrayList)routingTable[key];
            foreach (MessageQueue destination in destinations)
            {
                destination.Send(message);
                Console.WriteLine("sending message " + message.Body +
                                  " to " + destination.Path);
            }
        }
```

```
        mq.BeginReceive();
    }

    // 控制消息是格式为 XYZ:QueueName 的一个字符串
    protected void OnControlMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new System.Messaging.XmlMessageFormatter
                           (new String[] {"System.String,mscorlib"});
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        String text = ((String)message.Body);
        String [] split = (text.Split(new char[] {':'}, 2));
        if (split.Length == 2)
        {
            char[] keys = split[0].ToCharArray();
            String queueName = split[1];
            MessageQueue queue = FindQueue(queueName);
            foreach (char c in keys)
            {
                if (!routingTable.Contains(c))
                {
                    routingTable.Add(c, new ArrayList());
                }
                ((ArrayList)(routingTable[c])).Add(queue);
                Console.WriteLine("Subscribed queue " + queueName + " for message " + c);
            }
        }
        mq.BeginReceive();
    }

    protected MessageQueue FindQueue(string queueName)
    {
        if (!MessageQueue.Exists(queueName))
        {
            return MessageQueue.Create(queueName);
        }
        else
            return new MessageQueue(queueName);
    }
}
```

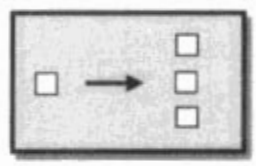
DynamicRecipientList 使用一种更为聪明的方式存储接收者的偏好。为了优化输入消息的处理，DynamicRecipientList 维护着一个 Hashtable，由输入消息的第一个字母作为键。

与动态路由器的例子不同，该 Hashtable 包含的不是一个目标，而是所有订购目标组成的一个 ArrayList。当 DynamicRecipientList 接收到一条消息时，它从散列表中找到正确目标列表，然后遍历整个列表，把消息发送到每一个目标。

对于未与任何规则匹配的输入消息，这个例子没有为之建立 dunnoChannel（见“基于内容的路由器”或“动态路由器”）。如果一个消息没有任何接收者，接收表一般不会把这种情况视为错误。

这个实现不允许接收者取消订购。它也不检查重复的订购。例如，如果一个接收者重复订购了两次相同的消息类型，它将接收到重复的消息。这一点与典型的发布-订购通道存在语义上的区别。在发布-订购通道中，特定的接收者只能对一个通道订购一次。如果需要，只需对 DynamicRecipientList 做简单的修改就能禁止重复的订购。

7.6 分解器



许多通过集成解决方案传递的消息都由多个元素组成。例如，由客户下的订单一般是由多个商品组成的。在“基于内容的路由器”中简要描述过，每个商品可能要由不同的库存系统处理。因此我们需要找出一种方法，既能处理完整的订单，又能分别处理包含在订单中的每个商品。

> 如果消息中包含多个元素，每个元素要采用不同的方式处理，这种消息应该如何处理？

对于这个路由问题，必须有通用的解决方案，只有这样才能应对不同数量和类型的元素。例如，一个订单中可能包含任意数量的商品，因此在设计解决方案时不能认为商品的数量是固定的，也不能对消息中包含的商品类型做过多的假设。例如，如果 Widgets & Gadgets 'R Us 公司准备明天开始销售图书，必须尽可能减少这一决策对整个解决方案造成的影响。

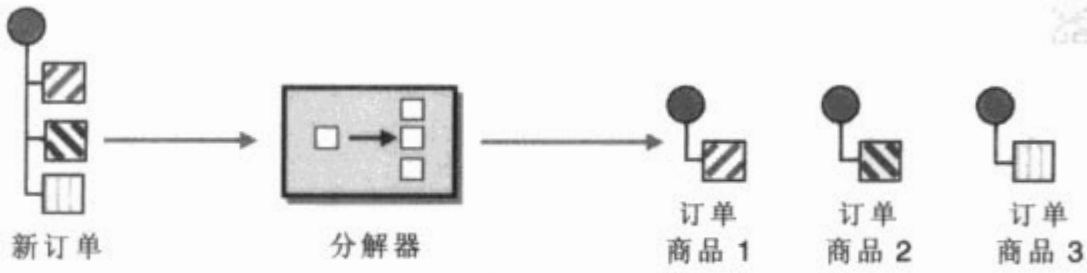
我们还想保持对订购商品的控制，防止在处理过程中重复或丢失。例如，可以使用发布-订购通道来把完整的订单发送给每个订单管理系统，让它挑选出能处理的商品。这种方法与“基于内容的路由器”中介绍的方法有相同的缺点，即很难防止单个商品的丢失或重复发货。

我们还要求解决方案在使用网络资源方面也应该是高效的。把完整的订单消息发送给每一个系统，而每个系统只处理一部分订单，这会产生额外的消息通信量，特别是随着目

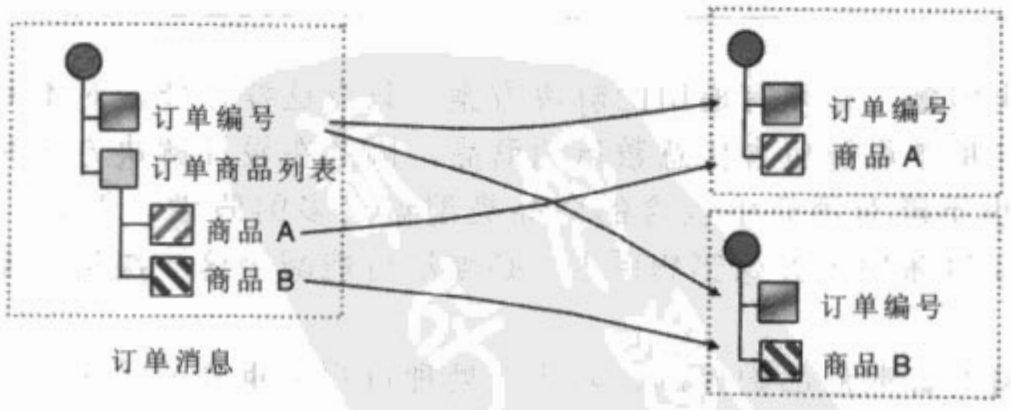
标数量的增加，通信量会越来越大。

为了避免多次发送完整的消息，可以分解原始的消息，根据库存系统的数量分解得到同样多的消息。每个消息只包含特定系统能处理的商品。这种方法类似于基于内容的路由器，只不过是把消息分解后再路由每个消息。虽然使用这种方法能提高效率，但是必须知道特定商品的类型和与之相关的目标。如果想修改路由的规则该怎么办？看来只有修改这种更复杂的“商品路由器”组件。在前面我们曾用管道和过滤器体系结构把处理过程分解为可组合的良定义组件，而不是把各种功能堆在一起，因此这里也应该利用这种体系结构来解决目前面临的问题。

使用分解器把组合的消息分解为一系列单独的消息，每条消息只包含与某个商品相关的数据。



分解器为输入消息中的每个元素（或元素的子集）发布一条消息。在许多情况下，我们需要在分解得到的各个消息中重复某些公共元素。增加这些公共元素是必要的，这样才能确保得到的各个子消息是自包含的，而且能支持各个子消息的无状态处理，同时以后还可以由此实现相关子消息的重组。例如，每个订单商品消息都应该包含订单编号的副本，只有这样才能把商品正确重组到订单或所有相关的实体中，如下订单的客户（如下图所示）。



把公共的数据元素复制到每个子消息中

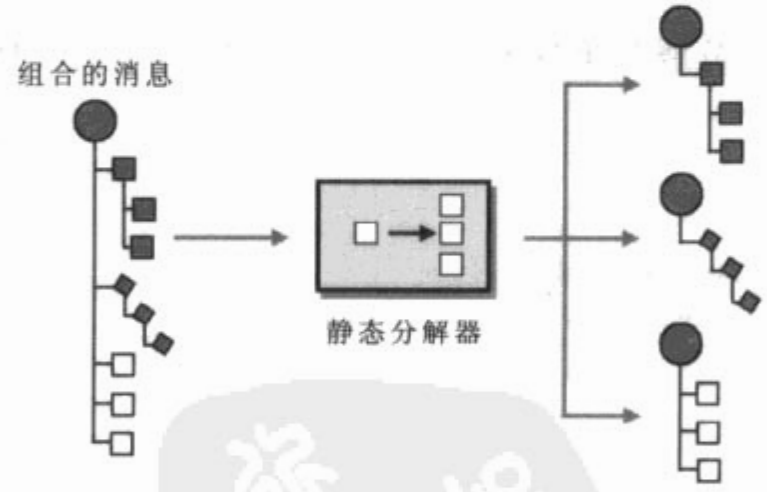
7.6.1 迭代分解器

前面已经提到过，许多企业集成系统都把消息数据存储在一个树型结构中。树型结构

的优点是可递归。节点下面的每个子节点都是另一个子树的根。这样我们就可以从消息树中取出一些组成部分，并把它们当作消息子树处理。如果使用消息树，可以方便地让分解器遍历某个特定节点下的所有子节点，并分别为每个子节点发送一条消息。这样的分解器实现非常通用，因为它不会对子元素的数量或类型做任何假设。许多商业 EAI 工具都提供了这种功能，并称之为迭代器（iterator）或排序器（sequencer）。为了尽量避免开发商术语可能带来的混淆，我们把这种类型的分解器统称为迭代分解器（iterating Splitter）。

7.6.2 静态分解器

不过，使用分解器不只是为了重复元素。一个很大的消息可能要分解为多个单独的消息，以便简化消息的处理。例如，许多 B2B 信息交换标准都制定了非常详尽的消息格式。这些巨大的消息一般是由委员会设计产生的，消息中很大一部分可能很少用到。在许多实例中，把这些超大规模消息（mega-message）分解为单独的消息是有好处的，每条消息只包含这个大消息中的特定部分。这种做法不但能简化后续的数据转换操作，还能节省网络带宽，因为对于那些只处理大消息中一部分数据的组件，只需向其路由较小的消息。分解后的消息往往发布给不同的通道，而不是发布给相同的通道，因为它们表示的是不同子类型的消息。在这种环境下，分解得到的消息数量一般是固定的，而更通用的分解器还可以分解得到数量不定的消息。为了把这种类型的分解器区分出来，我们把它称为静态分解器（Static Splitter）。静态分解器的功能等同于带一组内容过滤器的广播通道。



静态分解器把一个大型消息分解为数量固定的较小的消息

7.6.3 有序的或无序的子消息

在某些情况下，在子消息中编上序列号，可以提高消息的可跟踪性，同时能简化聚合器的工作。而且，在每个消息中可以增加指向原始（组合）消息的引用，这有利于把各个消息的处理结果与原始的消息关联起来。该引用起到了关联标识符的作用。

如果使用了消息信封（见“信封包装器”），每个新消息都要有自己的消息信封才能与消息传递基础设施兼容。例如，如果基础设施要求一个消息必须在消息首部中携带时间戳，就要把原始消息的时间戳传播到每个消息的首部。

例：采用 C#分解一个 XML 订单文档

许多消息传递系统都使用 XML 消息。例如，假设输入的订单消息如下所示：

```
<order>
    <date>7/18/2002</date>
    <ordernumber>3825968</ordernumber>
    <customer>
        <id>12345</id>
        <name>Joe Doe</name>
    </customer>
    <orderitems>
        <item>
            <quantity>3.0</quantity>
            <itemno>W1234</itemno>
            <description>A Widget</description>
        </item>
        <item>
            <quantity>2.0</quantity>
            <itemno>G2345</itemno>
            <description>A Gadget</description>
        </item>
    </orderitems>
</order>
```

我们想让分解器把订单分解为单独的订单商品。上面的示例文档应该被分解器分解为如下所示的两条消息：

```
<orderitem>
    <date>7/18/2002</date>
    <ordernumber>3825968</ordernumber>
    <customerid>12345</customerid>
    <quantity>3.0</quantity>

    <itemno>W1234</itemno>
    <description>A Widget</description>
</orderitem>

<orderitem>
    <date>7/18/2002</date>
    <ordernumber>3825968</ordernumber>
    <customerid>12345</customerid>
    <quantity>2.0</quantity>
    <itemno>G2345</itemno>
    <description>A Gadget</description>
</orderitem>
```

在每个 orderitem 消息中添加了从原始 order 消息中取出的订单日期、订单编号和客户

ID。把客户 ID 和订单日期包含在内，可以建立自包含的消息，这样消息的消费者就不必保存各个消息之间的上下文信息。如果消息要由无状态的服务器处理，实现消息自包含是非常重要的。这里增加了 ordernumber 字段，这是以后重组商品所必需的（见“聚合器”）。在这个例子中，我们假设订单中商品的特定顺序与订单的处理无关，因此在分解的消息中没有包含商品编号。

下面是采用 C#实现的分解器代码。

```
class XMLSplitter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue;

    public XMLSplitter(MessageQueue inQueue, MessageQueue outQueue)
    {
        this.inQueue = inQueue;
        this.outQueue = outQueue;

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();

        outQueue.Formatter = new ActiveXMessageFormatter();
    }

    protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new ActiveXMessageFormatter();
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        XmlDocument doc = new XmlDocument();
        doc.LoadXml((String)message.Body);

        XmlNodeList nodeList;
        XmlElement root = doc.DocumentElement;

        XmlNode date = root.SelectSingleNode("date");
        XmlNode ordernumber = root.SelectSingleNode("ordernumber");
        XmlNode id = root.SelectSingleNode("customer/id");
        XmlElement customerid = doc.CreateElement("customerid");
        customerid.InnerText = id.InnerXml;

        nodeList = root.SelectNodes("/order/orderitems/item");
```

```
        foreach (XmlNode item in nodeList)
        {
            XmlDocument orderItemDoc = new XmlDocument();
            orderItemDoc.LoadXml("<orderitem/>");
            XmlElement orderItem = orderItemDoc.DocumentElement;

            orderItem.AppendChild(orderItemDoc.ImportNode(date, true));
            orderItem.AppendChild(orderItemDoc.ImportNode(ordernumber, true));
            orderItem.AppendChild(orderItemDoc.ImportNode(customerid, true));

            for (int i=0; i < item.ChildNodes.Count; i++)
            {
                orderItem.AppendChild(orderItemDoc.ImportNode(item.ChildNodes[i], true));
            }

            outQueue.Send(orderItem.OuterXml);
        }

        mq.BeginReceive();
    }
}
```

大多数代码都主要完成 XML 处理。XMLSplitter 使用了事件驱动消费者结构，这与其他路由示例相同。每个输入消息都会调用 OnMessage 方法，这个方法把消息体转换为 XML 文档以便处理。首先，要从订单文档中取出相关的值。然后，迭代处理每个<item>子元素，为此将 XPath 表达式指定为/order/orderitems/item。简单的 XPath 表达式非常类似于文件路径，它沿着文档树向下，与路径中指定的元素名进行匹配。针对每个<item>，我们要分别构造一个新的 XML 文档，并把订单和商品子节点中的字段复制进去。

例：采用 C#和 XSL 分解一个 XML 订单文档

为了避免手工处理 XML 节点和元素，还可以创建一个 XSL 文档，把输入的 XML 文档转换为所需的格式，然后根据转换后的 XML 文档创建输出消息。如果文档的格式很容易改变，采用这种方法则有更高的可维护性。我们只需要修改 XSL 转换，而不必修改任何 C#代码。

新的代码使用由 XslTransform 类提供的 Transform 方法把输入文档转换为一种中间文档格式。对于每个结果消息，中间文档格式中有一个 orderitem 子元素。下面的代码只是遍历所有的子元素，并为每个元素发布一条消息。

```
class XSLSplitter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue;

    protected String styleSheet = "..\\..\\Order2OrderItem.xsl";
    protected XslTransform xslt;

    public XSLSplitter(MessageQueue inQueue, MessageQueue outQueue)
    {
        this.inQueue = inQueue;
        this.outQueue = outQueue;

        xslt = new XslTransform();
        xslt.Load(styleSheet, null);

        outQueue.Formatter = new ActiveXMessageFormatter();

        inQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }

    protected void OnMessage(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        mq.Formatter = new ActiveXMessageFormatter();
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        try
        {
            XPathDocument doc = new XPathDocument
                                (new StringReader( (String)message.Body));

            XmlReader reader = xslt.Transform(doc, null, new XmlUrlResolver());

            XmlDocument allItems = new XmlDocument();
            allItems.Load(reader);

            XmlNodeList nodeList = allItems.DocumentElement.
                                   GetElementsByTagName("orderitem");
```

```
            foreach (XmlNode orderItem in nodeList)
            {
                outQueue.Send(orderItem.OuterXml);

            }
        }
        catch (Exception e) { Console.WriteLine(e.ToString()); }
        mq.BeginReceive();
    }
}
```

我们从另一个单独文件中读取 XSL 文档，以利于编辑和测试。而且这样一来，不必重新编译代码就能改变分解器的行为。

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999 /XSL/Transform">
    <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

    <xsl:template match="/order">
        <orderitems>
            <xsl:apply-templates select="orderitems/item"/>
        </orderitems>
    </xsl:template>

<xsl:template match="item">
    <orderitem>
        <date>
            <xsl:value-of select="parent::node()/parent::node ()/date"/>
        </date>
        <ordernumber>
            <xsl:value-of select="parent::node()/parent::node ()/ordernumber"/>
        </ordernumber>
        <customerid>
            <xsl:value-of select="parent::node()/parent::node ()/customer/id"/>
        </customerid>
        <xsl:apply-templates select="*"/>
    </orderitem>
</xsl:template>

<xsl:template match="*">
    <xsl:copy>
```

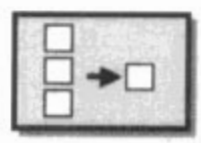
```
        <xsl:apply-templates select="@* | node()"/>
      </xsl:copy>
    </xsl:template>

</xsl:stylesheet>
```

XSL 是一种描述性语言，除非你自己已经编写过不少的 XSL 文档，否则很不容易理解（可以参考一本很好的介绍 XSL 的书[Tennison]）。这里的 XSL 转换会查找 order 元素的所有出现（在我们的文档中有一个 order 元素）。一旦找到该元素，它就为输出文档创建一个新的根元素（所有 XML 文档都必须有一个唯一的根元素），并继续处理输入文档中 orderitems 元素内部的所有 item 元素。XSL 为找到的每个商品指定了一个新的“模板”。该模板把日期（date）、订单编号（ordernumber）和客户 ID（customerid）元素从 order 元素（item 元素的父元素是 orderitems，而 order 则是 orderitems 的父元素）中复制出来，并附加上商品本身的所有元素。在产生的结果文档中，对于输入文档中的每个 item 元素，都有一个 orderitem 元素。这样就使得 C#代码可以很容易地迭代处理这些元素，并把它们发布为消息。

我们很想知道这两个实现完成得如何，因此做了一个快速的、不太科学的性能测试。我们把 5000 个订单消息添加到输入队列中，启动分解器，并测量 10000 个商品消息到达输出队列所需的时间。这项测试是使用本地消息队列在同一台机器的同一个程序中执行的。测量的结果发现，使用 DOM 提取元素的 XMLSplitter 总共耗时 7 秒，而基于 XSL 的分解器总共耗时 5.3 秒。为了建立一个测试基准，我们实现了一个虚拟处理器，它从输入队列中消费一条消息，并把相同的消息重复向输出队列发布两次。这个虚拟处理器处理 5000 条消息耗时 2 秒，其中包括消费 5000 条消息和发布 10000 条消息的时间，这个测试不考虑消费处理器所发布 10000 条消息所需的时间。因此，看起来 XSL 处理要比“手工”移动元素的效率高一些（如果减去基准，XSL 大概要快 35%）。可以确信这两个程序都能进一步优化以取得最大性能，但是看着它们同时执行确实很有意思。

7.7 聚合器



分解器可以把单个消息分解为一个子消息序列，每个子消息可以单独处理。同样地，接收表或发布-订购通道可以把一个请求消息同时转发给多个接收者，并从返回的多个响应消息中选择所需的消息。在大多数类似的环境下，后续处理要以子消息的成功处理为基础。例如，要从多个供货商返回的响应中选择最适合的出价，或者当所有的商品都已经从库房取出后，再向客户返回订单的账单。

怎样把独立但相关的结果消息组合起来，从而使它们能作为一个完整的消息来处理？

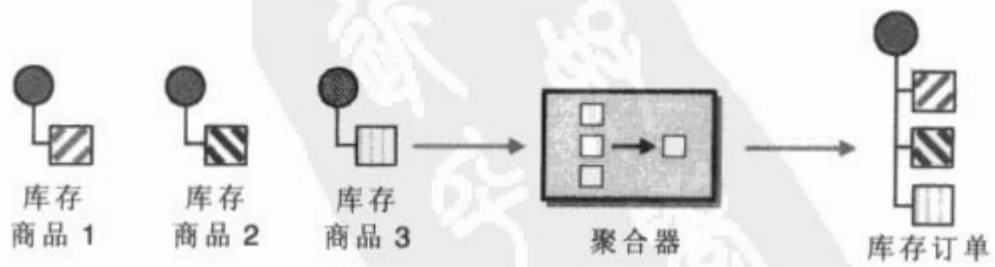
消息传递系统存在异步特性，这增加了从多个消息中收集信息的难度。需要从多少个消息中收集信息？如果把消息广播到一个广播通道，我们可能无法知道有多少个接收者在监听通道，因此也就无法知道会有多少个响应。

即使使用分解器，响应消息也不一定按它们创建的相同顺序到达。由于每个消息可能通过不同的网络路径路由，消息传递基础设施只能保证每条消息的传送，而不能保证每条消息消息传输的顺序。而且，每条消息可能由不同的参与者采用不同的速度处理。因此，响应消息很可能未按正常顺序传输（对于这个问题，更详细的描述见“重排器”）。

另外，大多数消息传递基础设施以“可靠、最终”传送方式运行，这说明它能保证消息传输给目标接收者，但是不能保证消息何时传输。等待一条消息需要花费多少时间？如果等待的时间太长，可能会耽误后续的处理。如果没有得到某条消息而想继续执行，就必须找到一种处理不完整信息的方法。即便如此，如果缺少的这些消息（或消息）最终到达了，又该怎么办？在有些情况下，可以单独处理各个消息，但是在另外一些情况下，这样做会导致重复的处理。而且，如果忽略延迟到达的消息，就会永远失去它们的信息内容。

所有这些问题都使得多个相关消息的组合处理变得更加复杂。如果每个单独的组件能够负责处理这种复杂性，对于需要所有子消息的后续处理业务，能够向其传递单个的一条消息，那么实现业务逻辑就会更为容易。

使用有状态的过滤器，即聚合器，把单独的消息收集并存储起来，直到接收到所有相关的完整的消息集合，然后聚合器从各个单独消息中提取信息，组合成一个消息后发布。



聚合器是一种特殊的过滤器（在管道和过滤器体系结构中），它接收一个消息流并识别出相互关联的消息。一旦接收到完整的消息集合（后面会介绍如何确定消息集是完整的），聚合器就从每个相关的消息中收集信息，并把它们组合为一个聚合消息发布到输出通道上，以便进一步的处理。

与前面介绍的大多数路由模式不同，聚合器是一种有状态的组件。像基于内容的路由器这样的简单路由模式一般是无状态的，即组件是一个接一个地处理输入消息，而不会保存消息间的任何信息。组件在消息处理前后的状态保持不变，因此，我们把这样的组件称为无状态的。聚合器不能是无状态的，因为它必须保存每条输入消息，直到属于同一消息集合的所有消息都到达为止。然后，它必须把每条消息中的相关信息取出来，合成一个聚合消息。例如，如果我们正在处理输入的拍卖竞标消息，只需保留最高的竞标价及其相关竞标者的 ID，而不必保存每个竞标消息的历史。在这个例子中，聚合器仍然要存储跨多个消息的信息，因此它也是有状态的。

设计一个聚合器时，需要指定以下属性：

1. **相关性**：哪些输入消息相互关联？
2. **完成条件**：何时才能发布结果消息？
3. **聚合算法**：怎样把接收到的消息组合成一个结果消息？

相关性通常由输入消息的类型来建立，或者利用一个明确的关联标识符来确定。常用的完成条件和聚合算法在后面介绍。

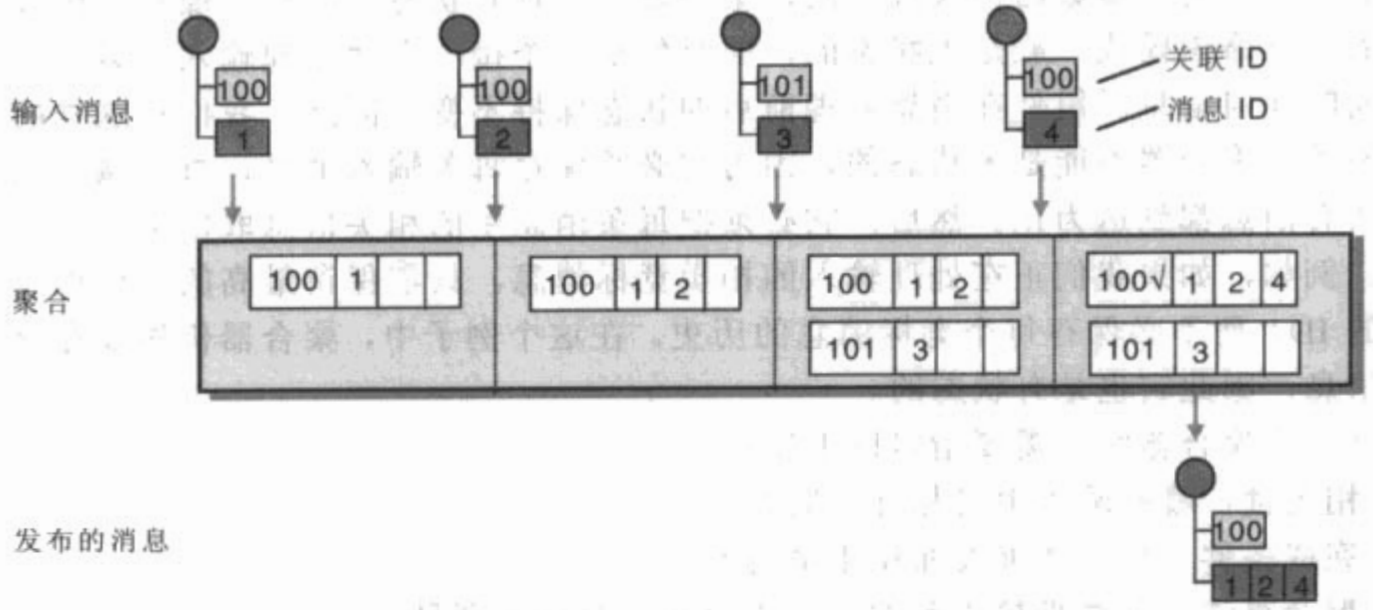
7.7.1 实现细节

由于消息传递系统是事件驱动的，聚合器可能在任意时刻以任意的顺序接收到相关的消息。为了组合消息，聚合器必须维护一个有效聚合列表，它是聚合器已经接收到的部分消息的集合。当聚合器接收到一条新消息时，要检查该消息是否属于已有消息集合的一部分。如果它与现有的消息集合无关，聚合器就创建一个新的消息集合，并把这个消息作为新集合中的第一条消息。如果它与现有的消息集合相关，聚合器就把它添加到相关的集合中。

完成消息添加后，聚合器会计算受影响消息集合的完成条件。如果条件满足，则从集合累积的消息中组合产生新的消息，并把新消息发布到输出通道上。如果条件不满足，则不发布任何消息，聚合器继续维护有效消息集合，等待新消息的到达。

下图演示了这一过程。在这个简单的示例中，输入消息通过关联标识符相关。当第一条关联标识符值为 100 的消息到达时，聚合器初始化一个新的集合，并把此消息存储在这个新集合中。在我们的例子中，完成条件指定为最少要接收到三条消息，因此聚合尚未完成。当第二条关联标识符值为 100 的消息到达时，聚合器把它添加到已有的集合中。这一次完成条件还是不满足。第三条消息指定了一个不同的关联标识符：101。因此，聚合器要创建一个新的消息集合。第四条消息与第一个消息集合相关（标识符值为 100）。在增加完这条消息后，集合中已经包含三条消息，完成条件已满足，聚合器会计算出组合的消息，把集合标记为已完成，并发布结果消息。

采用这种策略，当聚合器接收到一个与现有集合无关的新消息时，将创建一个新集合。因此，聚合器不需要提前知道可能产生哪些聚合，我们把这种聚合器称为自启动的聚合器（Self-Starting Aggregator）。



聚合器的内部流程

根据聚合策略，聚合器必须解决这样一种情况，即输入消息属于某个集合，但是该集合已经关闭——也就是说，输入消息在组合消息已经发布之后才到达。为了避免又从新集合开始，聚合器必须维护一个已关闭集合的列表。该列表将被周期性地清空，以防止无限增长。但是，为了防止延迟消息触发新的集合，关闭集合列表不能清空得过快。由于我们不必存储完成的集合，而只是存储关闭的集合，因此能够非常高效地存储关闭集合列表，并能为清空算法提供足够安全的处理周期。还可以使用消息到期来忽略延迟时间过长的消息。

为了增强整个解决方案的健壮性，也可以让聚合器监听一个特定的控制通道，从而允许手工地清空所有有效集合或者清空某个指定集合。如果我们想从错误条件下恢复同时又不想重启聚合器组件，就可以利用这一特性。同样的道理，如果允许聚合器根据请求把有效集合列表发布到一个特殊的通道上，这个特性对于调试是非常有用的。这两项功能充分展示了控制总线所具有的特性。

7.7.2 聚合策略

建立聚合器完成条件的策略有很多。现有的策略主要取决于我们是否知道需要多少个消息。如果聚合器接收到原始组合消息的副本，或者每个消息中都包含消息总数（在“分解器”示例中已经介绍过），聚合器就能知道需要多少个子消息。根据聚合器对消息流的了解程度，大多数常用的策略可以划分为如下几种：

1. **等待所有消息**（Wait for All）：在所有响应消息到达前一直等待。这种情况与我们前面介绍过的订单示例很相似。一个未完成的订单是没有意义的。因此，如果在一定的超时时间范围内没有接收到所有的商品应答，聚合器就产生一个错误条件。这种方法为作出决策提供了最基本的支持，但是它的效率和可靠性都是最低的（并且还要知道有多少个消息需要等待）。只要有一个消息丢失或延迟，就会耽误整个消息集合的处理。在松耦合的异

步系统中，要解决这样的错误是非常棘手的，因为异步的消息流使得错误条件的检测变得非常困难（在消息“丢失”前我们应该等待多长时间？）。应对消息丢失的一种方法是重新请求相应的消息。但是，这种方法需要聚合器了解消息的来源，这又会在聚合器与其他组件之间增加额外的依赖性。

2. **超时**（Timeout）：在指定的时间范围内等待响应消息，计算在时限内接收到的响应消息并作出决策。如果没有接收到响应消息，系统可能报告异常或重试。如果输入的响应消息被评定了分数，并且只使用分数最高的消息（或一部分消息），那么采用这种方法是比较适合的。例如“竞标”应用中这种方法就很常用。

3. **第一个最好**（First Best）：只等待接收第一个（最快的一个）响应，而忽略其他的所有响应。这种方法是最快的，但是忽略了大量的信息。它适用于竞标或报价应用，在这些情况下，响应时间往往最为关键。

4. **覆盖超时**（Timeout with Override）：等待指定的时间，或者接收到有预置最低分数的消息为止。在这种情况下，如果发现非常满意的响应就中止等待，否则一直等到超时。如果到超时时还没有发现合适的消息，则对所有接收到的消息按分数排序。

5. **外部事件**（External Event）：有时聚合是否结束是由外部业务事件的到达决定的。例如，在金融产业中，交易日的结束就意味着接收到的报价聚合的结束。使用定时的事件会降低灵活性，因为它无法提供更多的变化。另外，如果采用事件消息的方式发送一个指定的业务事件，可以实现系统的集中控制。聚合器可以在一个特殊的控制通道上监听事件消息，或接收一个特定格式的消息来表示聚合结束。

与完成条件的选择紧密相关的是聚合算法的选择。下面是将多个消息聚合为一个消息的常用策略：

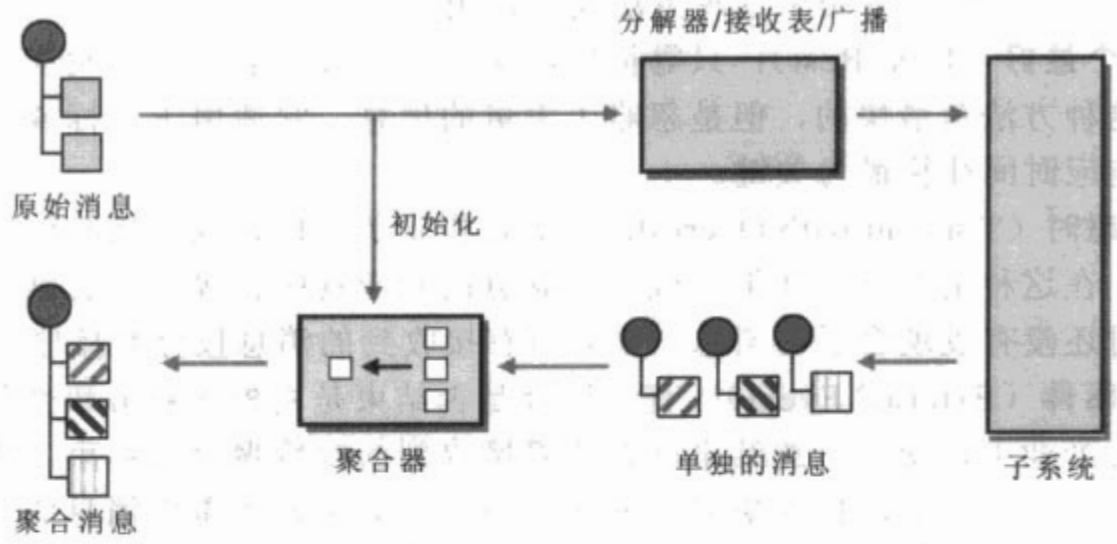
1. **选择“最好”的答案**：使用这种方法的前提是存在一个最好的答案，如某个商品的最低报价。这样聚合器就可以做出决定，只传递“最好”的消息。不过在现实环境中，选择消息的标准不会这么简单。例如，某商品的最好报价可能由传送的时间、可供应的商品数量、供货商是否在首选供货商名单上等诸多因素决定。

2. **精简数据**：大容量数据源往往会带来很大的消息流量，聚合器可以用于减少消息流量。如果需要计算每个消息的平均结果，或者要把各个消息中的数字字段累加到一个消息中，此时就可以采用这种方法。如果每条消息代表的是一个数值，如已接收到的订单数，这种方法就是最理想的。

3. **收集数据以便事后评估**：对于如何选择最佳答案，要让聚合器做出决定一般是不可能的。在这些情况下，使用聚合器收集每条消息并把它们组合为一条消息仍然是有意义的。组合后的消息可能只是对每条消息中数据的简单汇集。可能在以后由另一个的组件或人来完成聚合决定。

在很多情况下，聚合策略是由参数驱动的。例如，如果策略是等待指定的时间间隔，则可以配置最大等待时间。同样地，如果聚合的策略是等待直到提供的数据超过某个指定的阈值为止，那么通常的做法是让聚合器事先知道规定的阈值。如果这些参数是在运行时配置的，聚合器必须能通过额外的输入来获取控制消息（如这些参数设置）。控制消息中还

可能包含另外一些信息，如需要多少相关的消息，以帮助聚合器实现更有效的完成条件。在这种环境下，当第一个消息到达时，聚合器并不只是创建新的集合，还会接收与期望消息系列相关的基本信息。这个信息可能是原始请求消息的副本（如“分散收集器”消息），并增加了所有必要的参数信息。然后聚合器分配一个新的集合，并把参数信息与集合存储在一起（如下图所示）。当每条消息到达时，把它与对应的集合相关联。这种类型的聚合器不同于自启动聚合器，我们把它称为初始化聚合器（initialized Aggregator）。显然，只有能访问原始消息才能实现这种配置，而通常并不能访问原始消息。



聚合器在许多应用中都很有用。聚合器一般要与分解器或接收表配合形成组合模式。对这些组合模式的详细介绍请参考“组合消息处理器”和“分散收集器”。

例：贷款中介

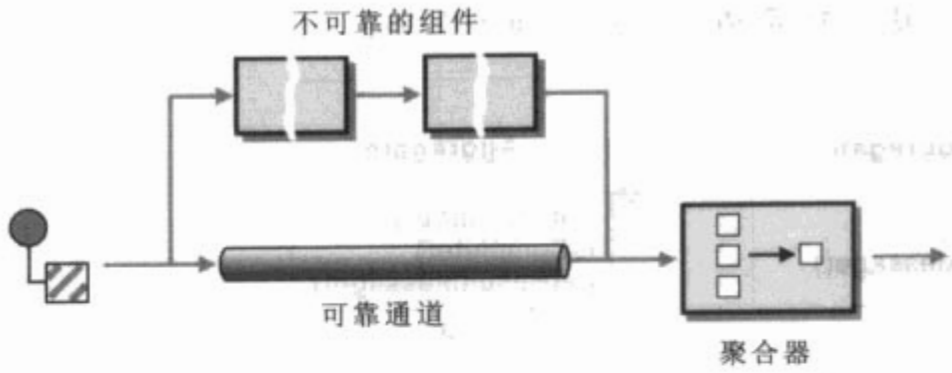
第9章“中场演练：复合消息传递”展示了复合的消息传递的一个实例，它使用了聚合器，从银行返回的贷款利率查询消息中选择最佳的贷款利率。其中还给出了用Java、C#和TIBCO实现的聚合表。

例：作为丢失消息检测器的聚合器

Joe Walnes展示了使用聚合器的一种新方法，很有创意。他的系统通过一个组件序列发送消息，但是非常不可靠。即使使用可靠传输也无法解决这个问题，因为系统本身可能会在消费完消息后失败。由于应用并不是事务性客户，正在处理的消息将丢失。为了弥补这一系统缺陷，Joe 通过两个并行的路径传递输入消息：一个是通过所要求的但不可靠的组件传递，另一个通过使用可靠传输的组件传递。聚合器再把两条路径上的消息重新组合起来（如下图所示）。

聚合器采用“覆盖超时”作为完成条件，如果超时时间到，或者接收到两个关联的消息，则完成聚合。聚合算法取决于哪个条件先满足。如果接收到两个消息，则不加修改地把处理的消息转发出去。如果超时，则说明其中某个组件失败并“吃掉”了消息。因此，聚合器必须发布一条错误消息，警告操作人员其中某个组件失败了。遗憾的是，

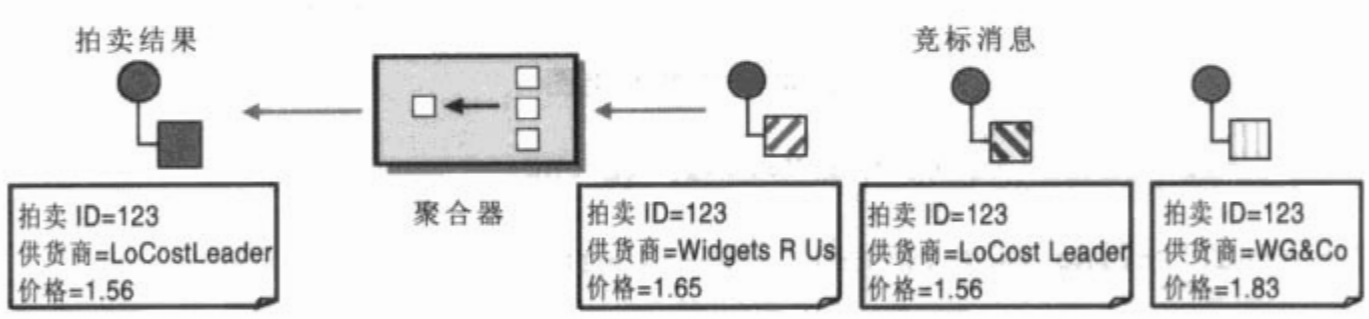
这个组件必须手工重启，不过也可以采用更复杂的配置方案来实现组件的重启和丢失消息的重新发送。



能够对丢失消息完成超时检测的聚合器

例：采用 JMS 实现的聚合器

这个例子展示了如何使用 Java 消息传递服务（Java Messaging Service，JMS）API 实现聚合器。聚合器从某个通道接收竞标消息，把相关的竞标消息集中起来，并把有最低竞标价的消息发布给另一个通道。竞标消息通过一个拍卖 ID 属性关联起来，它相当于消息的关联标识符。在此聚合策略是最少接收三条竞标消息。聚合器是自启动的，不需要外部初始化。



选择最低竞标价的聚合器示例

这个解决方案包括以下主要的类（如下图所示）：

1. Aggregator 包含接收消息、把消息集中起来并发送结果消息的逻辑。它通过 Aggregate 接口与集合交互。

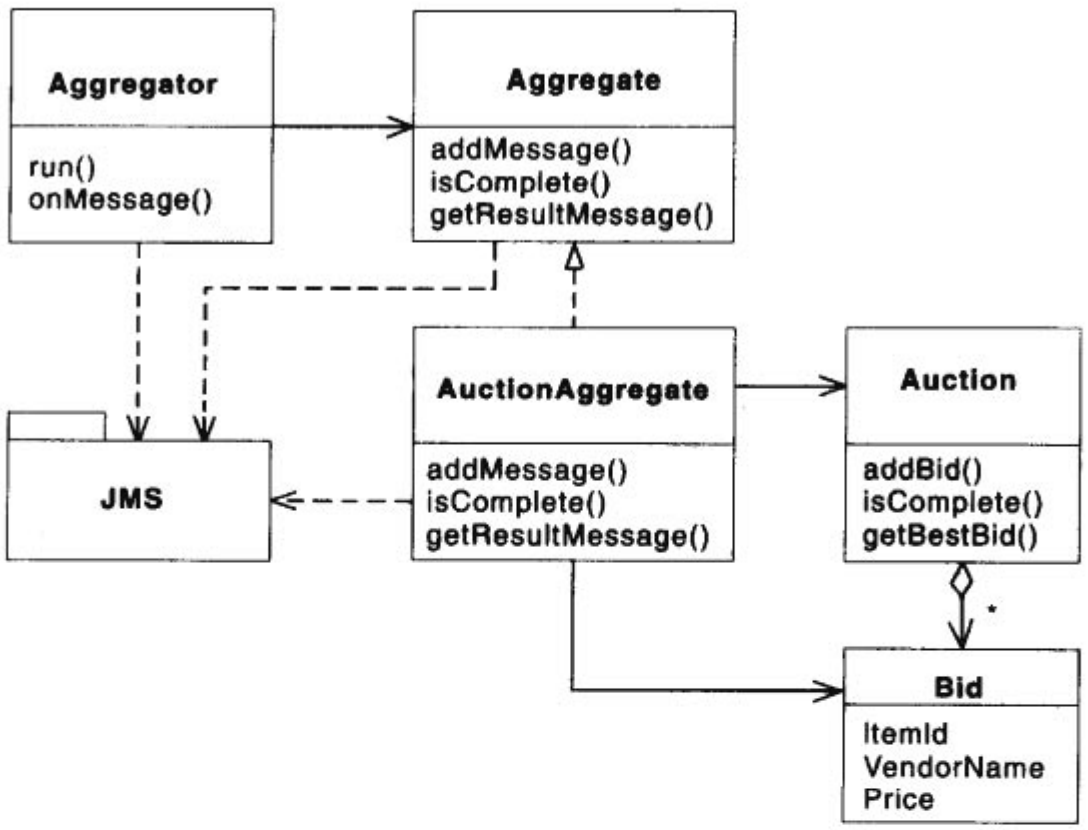
2. AuctionAggregate 实现了 Aggregate 接口。该类是 Aggregate 接口与 Auction 类之间的适配器[GoF]。采用这种设置有利于 Auction 类独立于 JMS API。

3. Auction 是已接收的相关竞标消息的集合。Auction 类实现了聚集策略，例如，找出最低竞标价并确定聚合何时完成。

4. Bid 是一个便利类，用于存储与竞标有关的数据项。我们把输入的消息数据转换为一个 Bid 对象，这样就能通过一个强类型接口存取竞标数据，使 Auction 的逻辑完全与 JMS API 无关。

这个解决方案的核心是 Aggregator 类。这个类需要两个 JMS Destination：一个用于输入，另一个用于输出。Destination 是 Queue（点对点通道）或 Topic（发布-订购通道）在

JMS 中的抽象。基于这个抽象类，我们能编写与通道类型无关的 JMS 代码。这对测试和调试非常有用。例如，在进行测试时，我们需要使用发布-订购主题，以便“监听”消息业务流。当交付产品时，则可能希望把主题切换为队列。



拍卖聚合器的类图

```
public class Aggregator implements MessageListener
{
    static final String PROP_CORRID = "AuctionID";

    Map activeAggregates = new HashMap();

    Destination inputDest = null;
    Destination outputDest = null;
    Session session = null;

    MessageConsumer in = null;
    MessageProducer out = null;

    public Aggregator (Destination inputDest, Destination outputDest, Session session)
    {
        this.inputDest = inputDest;
        this.outputDest = outputDest;
        this.session = session;
    }
```

```
public void run()
{
    try {
        in = session.createConsumer(inputDest);
        out = session.createProducer(outputDest);
        in.setMessageListener(this);
    } catch (Exception e) {
        System.out.println("Exception occurred: " + e.toString());
    }
}

public void onMessage(Message msg)
{
    try {
        String correlationID = msg.getStringProperty(PROP_CORRID);
        Aggregate aggregate = (Aggregate)activeAggregates.get(correlationID);
        if (aggregate == null) {
            aggregate = new AuctionAggregate(session);
            activeAggregates.put(correlationID, aggregate);
        }
        //--- 如果集合已经关闭，则忽略消息
        if (!aggregate.isComplete()) {
            aggregate.addMessage(msg);
            if (aggregate.isComplete()) {
                MapMessage result = (MapMessage)aggregate.getResult Message();
                out.send(result);
            }
        }
    } catch (JMSException e) {
        System.out.println("Exception occurred: " + e.toString());
    }
  }
}
```

Aggregator 是一个事件驱动的消费者，它实现了 MessageListener 接口，因此必须实现 onMessage 方法。由于 Aggregator 是 MessageConsumer 的消息监听者，每当有新的消息到达消费者的目标时，JMS 就会调用 onMessage 方法。对于每个输入消息，Aggregator 会取出其中的关联 ID（存储为一个消息属性），并检查有效集合中是否存在该关联 ID。如果没有找到相关的集合，Aggregator 就初始化一个新的 AuctionAggregate 实例。接着 Aggregator 检查集合是否还有效（没有完成）。如果集合不再有效，它就把输入的消息丢掉。如果集合还有效，则把消息添加到集合中，并测试结束条件是否满足。如果满足，则把最适合的竞标消息发布出去。

Aggregator 的代码很有通用性，与这个特定示例应用相关的代码只有两行。首先，它

认为关联 ID 存储在消息的 AuctionID 属性中。其次，它要创建一个 AuctionAggregate 对象。如果使用返回 Aggregate 类型对象的工厂（factory），并在内部创建类型为 AuctionAggregate 的一个实例，就可以避免这样直接引用 AuctionAggregate 对象。因为这本书探讨的主题是企业集成而不是面向对象设计，所以为简单起见，这里允许存在这种依赖。

AuctionAggregate 类提供了 Aggregate 接口的实现。这个接口非常简单，它只包括 3 个方法：一个用于添加新的消息（addMessage），一个用于确定聚合是否完成（isComplete），另一个用于获得最佳的结果（getBestMessage）。

```
public interface Aggregate {
    public void addMessage(Message message);
    public boolean isComplete();
    public Message getResultMessage();
}
```

我们没有在 AuctionAggregate 类的内部实现聚合策略，而是决定创建一个单独的 Auction 类，用以实现不依赖于 JMS API 的聚合策略：

```
public class Auction
{
    ArrayList bids = new ArrayList();

    public void addBid(Bid bid)
    {
        bids.add(bid);
        System.out.println(bids.size() + " Bids in auction.");
    }

    public boolean isComplete()
    {
        return (bids.size() >= 3);
    }

    public Bid getBestBid()
    {
        Bid bestBid = null;

        Iterator iter = bids.iterator();
        if (iter.hasNext())
            bestBid = (Bid) iter.next();

        while (iter.hasNext()) {
            Bid b = (Bid) iter.next();
            if (b.getPrice() < bestBid.getPrice()) {
                bestBid = b;
            }
```

```
        }
        return bestBid;
    }
}
```

Auction 类其实非常简单。它像 Aggregate 接口那样提供了三个方法，但是这些方法的参数不同，因为它们使用了强类型的 Bid 类，而不是特定于 JMS 的 Message 类。在这个例子中，完成（completeness）条件非常简单，只需接收到三个竞标消息。但是，把聚合策略与 Auction 类和 JMS API 分离开来很有好处，这有利于在 Auction 类中增加更复杂的逻辑。

AuctionAggregate 类相当于 Aggregate 接口和 Auction 类之间的适配器[GoF]。适配器类可以把一个类的接口转换为另一个接口。

```
public class AuctionAggregate implements Aggregate {
    static String PROP_AUCTIONID = "AuctionID";
    static String ITEMID = "ItemID";
    static String VENDOR = "Vendor";
    static String PRICE = "Price";

    private Session session;
    private Auction auction;

    public AuctionAggregate(Session session)
    {
        this.session = session;
        auction = new Auction();
    }

    public void addMessage(Message message) {
        Bid bid = null;
        if (message instanceof MapMessage) {
            try {
                MapMessage mapmsg = (MapMessage)message;
                String auctionID = mapmsg.getStringProperty(PROP_AUCTIONID);
                String itemID = mapmsg.getString(ITEMID);
                String vendor = mapmsg.getString(VENDOR);
                double price = mapmsg.getDouble(PRICE);
                bid = new Bid(auctionID, itemID, vendor, price);
                auction.addBid(bid);
            } catch (JMSException e) {
                System.out.println(e.getMessage());

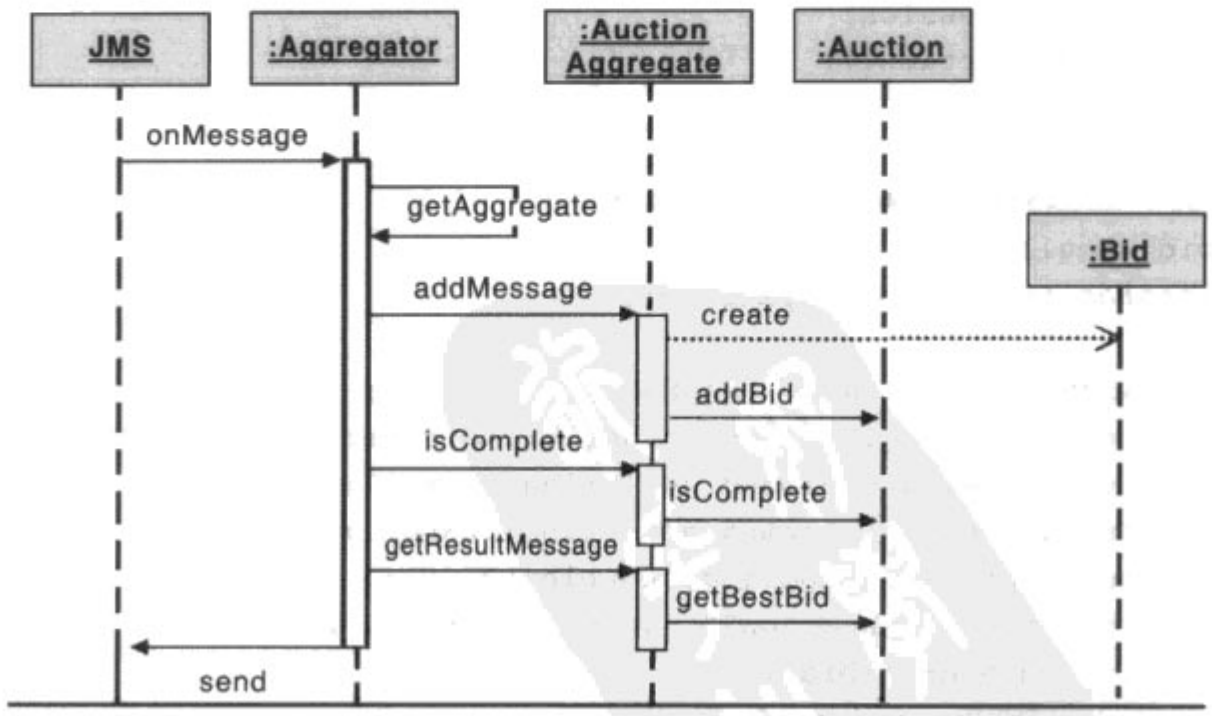
            }
        }
```

```
    }

    public boolean isComplete()
    {
        return auction.isComplete();
    }

    public Message getResultMessage() {
        Bid bid = auction.getBestBid();
        try {
            MapMessage msg = session.createMapMessage();
            msg.setStringProperty(PROP_AUCTIONID, bid.getCorrelationID());
            msg.setString(ITEMID, bid.getItemID());
            msg.setString(VENDOR, bid.getVendorName());
            msg.setDouble(PRICE, bid.getPrice());
            return msg;
        } catch (JMSException e) {
            System.out.println("Could not create message: " + e.getMessage());
            return null;
        }
    }
}
```

下面的顺序图归纳了这些类之间的交互。



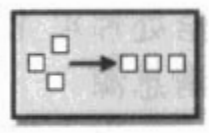
拍卖聚合器顺序图

这个简单的示例假设拍卖 ID 是全局唯一的。因此不必考虑清空开放的拍卖列表，

尽可以让它增长。而在实际的应用中，则要考虑何时清空老的拍卖记录，以防止内存泄漏。

由于该代码只引用了 JMS Destination，因此实际运行可以使用 Topic 或 Queue。在生产环境中，应用很可能使用点对点通道（等价于 JMS Queue），因为每个竞标应该只有一个接收者，即 Aggregator。正如在“发布-订购通道”中介绍过的，主题有利于测试和调试。为主题添加一个额外的监听者非常简单，而且不会影响消息流。当调试一个消息传递应用时，可以运行一个单独的监听者窗口，跟踪各个参与方之间交换的所有消息，这通常非常有用。许多 JMS 实现都支持在主题名中使用通配符，这样一来，监听者就可以使用*作为主题名来订购所有的主题。最好能利用一个简单的监听者工具，把主题中传递的所有消息显示出来，同时把消息记录到一个文件中，以便事后分析。

7.8 重排器

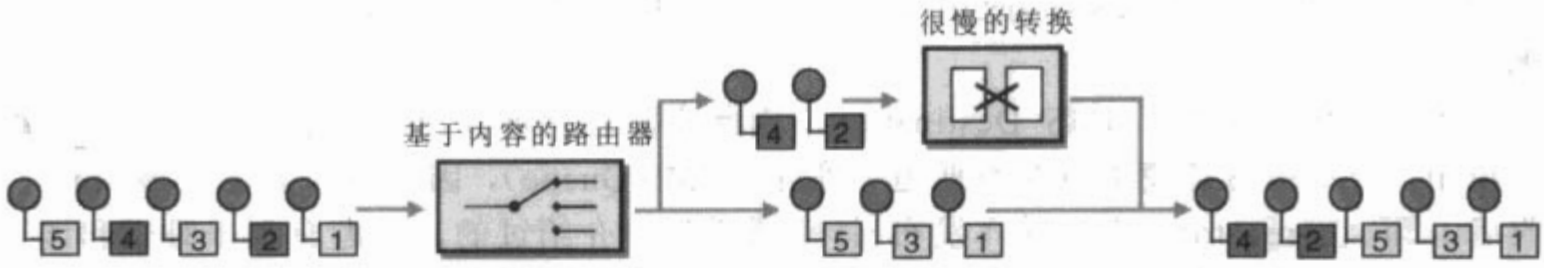


消息路由器可以根据消息的内容或其他规则把消息从一个通道路由给不同的通道。由于每条消息采用的路由不同，某些消息经过的处理步骤可能比其他消息要快，这样消息无法遵循原先的顺序。而某些后续的处理步骤却要求按顺序处理消息，例如要保持引用的完整性。

如何把相互关联但是乱序的消息重新恢复成正确的顺序？

为了解决顺序不正确问题，有一个最明显的解决方案，这就是首先保持消息的有序性。保持有序要比恢复有序容易得多。很多大学图书馆不让读者把书放回（已排序的）书架上，也正是出于同样的原因。通过对插入过程加以控制，任何时刻（几乎）都能保证正确的顺序。但是，实现异步消息传递解决方案时，要想保持有序性会很困难，这就如同在劝说一个十多岁的少年“保持房间井然有序确实是一种更高效的方法”。

使用不同的处理过程处理不同的消息时，往往会导致顺序不正确。首先来看一个简单的例子。假设要处理已编号的一系列消息。如果所有编号为偶数的消息都要经过特殊的转换，而所有奇数编号的消息不需要这样的处理，那么奇数编号消息就会比等待转换的偶数编号消息先到达结果通道。如果转换过程很慢，所有奇数编号消息可能会先于任何一个偶数编号消息出现在输出通道上，这就导致消息序列的顺序完全不正确（如下图所示）。

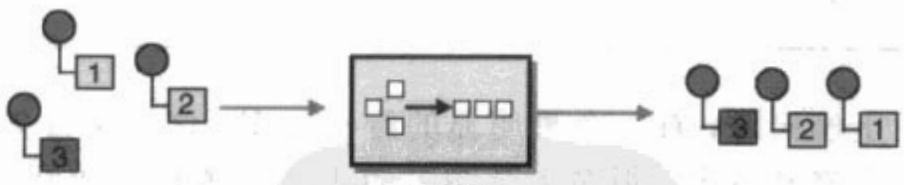


乱序的消息

为了避免消息的乱序，可以引入一种回送（确认）机制，确保每次只有一条消息经过系统；也就是说，下一条消息只有在最近的消息已经完成处理后才能发送。虽然采用这种保守的方法能解决乱序问题，但是它存在两个显著的缺点。第一，它使系统的处理速度显著下降。如果系统是由很多并行处理单元组成的，采用这种方法很难发挥出应有的处理能力。实际上，很多应用实例之所以采用并行处理，目的就是为了提高性能，而把消息流量限制为每次仅一个消息，这就完全违反了并行处理的初衷。第二，这种方法必须控制发送给处理单元的消息。但是，我们一般只能在接收端发现得到了乱序的消息流，而无法控制消息源。

聚合器可以接收消息流，识别出相关的消息，再根据各种策略把它们组合为一条消息。在这个处理过程中，聚合器还必须考虑到每条消息可能在任何时刻以任意的顺序到达。为此，聚合器可以把消息存储起来，直到所有相关的消息都已到达后再发布结果消息。

使用一个有状态的过滤器：重排器（Resequencer），搜集并重排消息，使消息能按指定的顺序发布到输出通道上。



重排器可以接收未按顺序到达的消息流。它把乱序的消息存储在一个内部缓冲区中，直到获得完整的消息序列，然后按正确的顺序把消息发布到输出通道上。关键是输出通道必须保证消息的顺序，只有这样才能保证消息按顺序到达下一组件。和其他大多数路由器一样，重排器一般不会修改消息的内容。

7.8.1 序列号

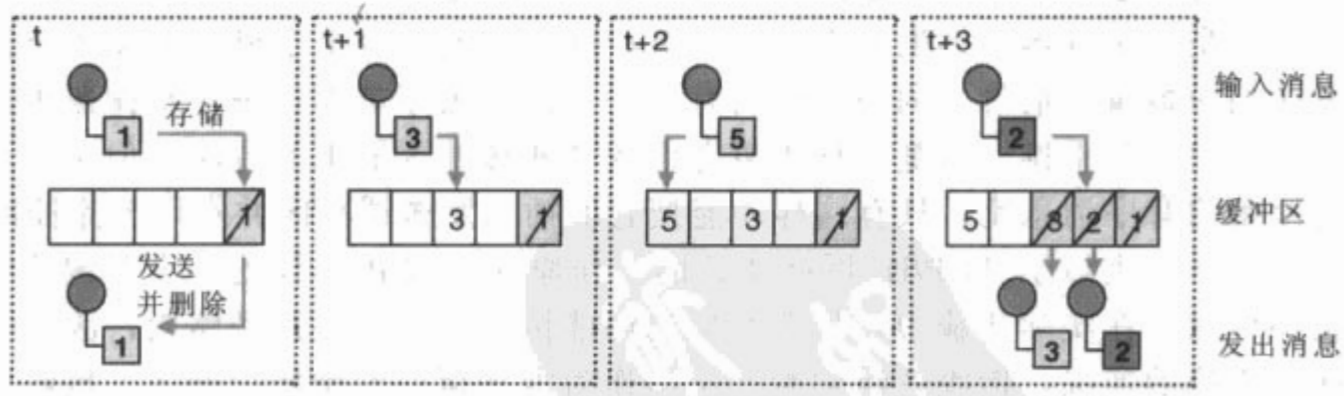
为了实现重排器，每条消息必须有唯一的序列号（见“消息序列”）。该序列号与消息标识符或关联标识符不同。消息标识符是一种用于唯一标识各个消息的特殊属性。但是在

大多数情况下，消息标识符是不可比较的。它们一般是随机数，有时甚至不是数字。即使消息标识符是数字值，把现有的消息标识符元素当作序列号使用也是不恰当的。采用关联标识符是为了将输入消息与原先发出的请求消息相匹配（见“请求/应答”）。对关联标识符的唯一要求是保证唯一性，它们不一定是数字值，也不要求是有序的。因此，如果想保持消息序列的顺序，应该定义一个单独的字段记录每条消息在序列中的位置。该字段一般位于消息的首部。

生成序列号要比生成唯一标识符花费更多的时间。唯一标识符一般可以用分布式的方式产生，例如可以把唯一的位置信息（如网卡的 MAC 地址）与当前时间组合起来，大多数 GUID（全局唯一标识符）算法都采用了这种方式。要生成有序的编号，一般需要一个计数器为整个系统分配编号。在大多数情况下，不但要采用升序方式生成编号，编号之间还必须是连贯的，否则就很难识别出丢失的消息。如果在设计时不注意，序列号生成器很容易成为消息流的瓶颈。如果每条消息都是由分解器产生，那么最好将消息编号功能结合到分解器中。身份字段（Identity Field）模式[EAA]对如何生成键和序列号做了很好的介绍。

7.8.2 内部操作

序列号可以确保重排器能检测出未按顺序到达的消息。但是，当消息确实未按顺序到达时，重排器应该如何操作呢？乱序的消息是指，有较高序列号的消息比有较低编号的消息先到达。重排器必须把有较高编号的消息存储起来，直到接收到所有“丢失”的较低编号消息为止。在这期间，它可能还会接收到其他乱序的消息，同样也要把它们存储起来。一旦缓冲区中包含了连续的消息，重排器就把这个消息序列发送到输出通道中，然后从缓冲区中删除已发送的消息（如下图所示）。



重排器的内部操作

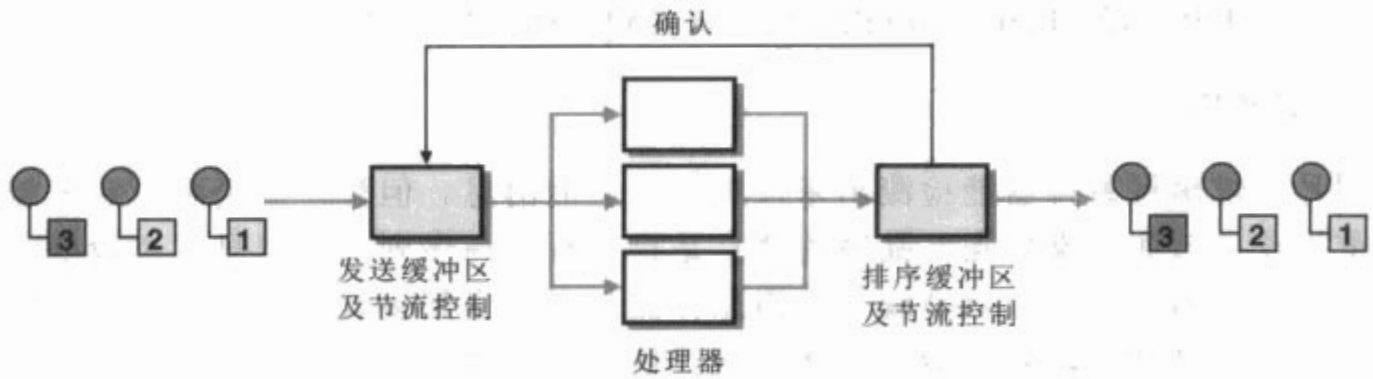
在这个简单的例子中，重排器接收到的消息序列号分别为 1、3、5 和 2。假设消息序列是从 1 开始的，因此第一条消息可以直接发送，并从缓冲区中删除。下一条消息的序列号为 3，说明前面缺少编号为 2 的消息。因此，要把编号为 3 的消息存储起来，直到有正确的消息序列为止。对下一条序列号为 5 的消息，处理也是一样的。一旦编号为 2 的消息到达，缓冲区中就包含了已按正确顺序排列的消息 2 和消息 3。因此，重排器把这些消息发布出去并从缓冲区中将它们删除。编号为 5 的消息仍然要保存在缓冲区中，直到它前面

的“间隙”被填满为止。

7.8.3 避免缓冲区溢出

应该分配多大的缓冲区？如果要处理的消息序列很长，缓冲区就会变得相当大。更糟糕的是，假设我们配置的系统有多个处理单元，每个处理单元处理不同的消息类型。如果其中一个处理单元失败了，顺序不正确的消息流就会很长，很可能造成缓冲区溢出。在某些情况下，可以使用消息队列来吸收待处理的消息，前提是消息传递基础设施能够支持按一定的选择标准从队列中读消息，而不是只能读取最旧的消息。如果这种方法可行，就可以轮询队列，查看第一个缺少的消息是否已经到达，而不要消费其间到达的所有消息。但是，有些时候即使是分配给消息队列的存储空间也会被占满。

要避免缓冲区溢出，一种可靠的方法是使用主动确认机制来控制消息的生产者（如下图所示）。



采用主动确认机制避免缓冲区溢出

正如前面讨论过的，一次只发送一条消息的效率很低。为此要采用一种更智能的方法。更有效的方法是让重排器告诉生产者在它的缓冲区中还剩多少个空槽。消息的节流控制端可据此发送相应数量的消息，因为即使所有这些消息的顺序都不正确，重排器也能把它们放在缓冲区里并对其重排。这是一种折衷考虑效率和缓冲区需求的方法。但是，这要求我们能访问原始的有序消息流，只有这样才能把它们插入发送缓冲区并实现节流控制。

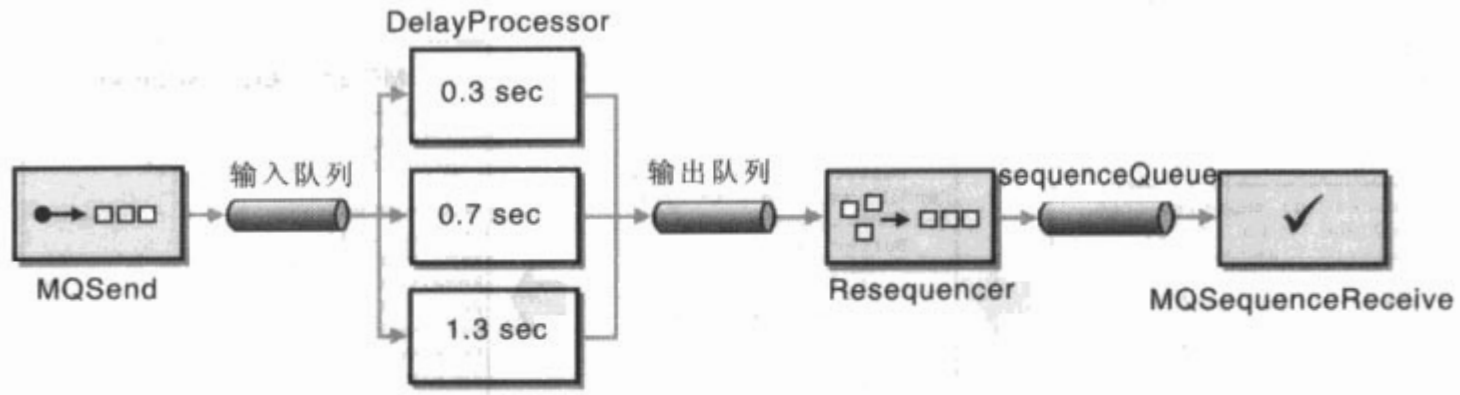
这种方法非常类似于 TCP/IP 网络协议的工作原理。TCP 协议的一个关键特性是确保数据包按序传送。在实际传输中，每个数据包可能按不同的网络路径路由，因此会经常发生包顺序不正确的情况。接收者维护着一个用作为滑动窗口的循环缓冲区。接收者和发送者在每次确认前要协商可发送的数据包数。由于发送者要等待接收者的确认，因此较快发送者的速度不能超过接收者，也就不会引起缓冲区溢出。一些特定规则还能防止所谓的糊涂窗口综合症（Silly Window Syndrome），即发送者和接收者进入一种非常低效的、一次传递一个包的工作模式。

克服缓冲区溢出还有一种方法，可以为缺少的消息计算替代消息。如果接收者能接受“足够好”的消息数据，并且不要求每条消息都提供精确的数据，或者消息传递的速度比准确性更重要，就可以采用这种方法。例如，在利用 IP 传送语音时，采用空白数据包代替丢失的数据包而不是重新请求，能够明显减少语音流的延迟，使用户获得更好的体验。

大多数应用开发者认为网络通信应该是可靠的。在设计消息传递解决方案时，对 TCP 的内部实现有所了解会很有帮助，这是因为，在它的核心，IP 业务流是异步的，而且不可靠，必须处理许多企业集成解决方案所面临的相同问题。对 IP 协议的深入介绍可参考[Stevens]和[Wright]。

例：采用 Microsoft .NET 的 MSMQ 实现的重排器

为了通过实际的例子来说明重排器的功能，我们采用了以下设置：



重排器的测试设置

测试设置由四个主要组件组成，每个组件均实现为 C#类。组件通过 Windows2000 和 Windows XP 的消息排队服务所提供的 MSMQ 消息队列进行通信。

1. MQSend 作为测试消息生成器。每条消息的消息体包含一个简单的字符串。MQSend 还在每条消息的 AppSpecific 属性中设置了一个消息序列号。序列号从 1 开始，消息的编号可以从命令行输入。MQSend 把消息发布到私有队列 inQueue 中。

2. DelayProcessor 从 inQueue 中读取消息。它只是在一段时间延迟之后再把相同的消息重新发送到 outQueue 中。我们同时使用了三个 DelayProcessor 来模拟负载平衡的处理单元。处理器扮演了相互竞争的消费者，每条消息只能被一个处理器消费。所有的处理器都把消息发布到 outQueue 中。由于处理速度各不相同，输出到 outQueue 中的消息是顺序不正确的。

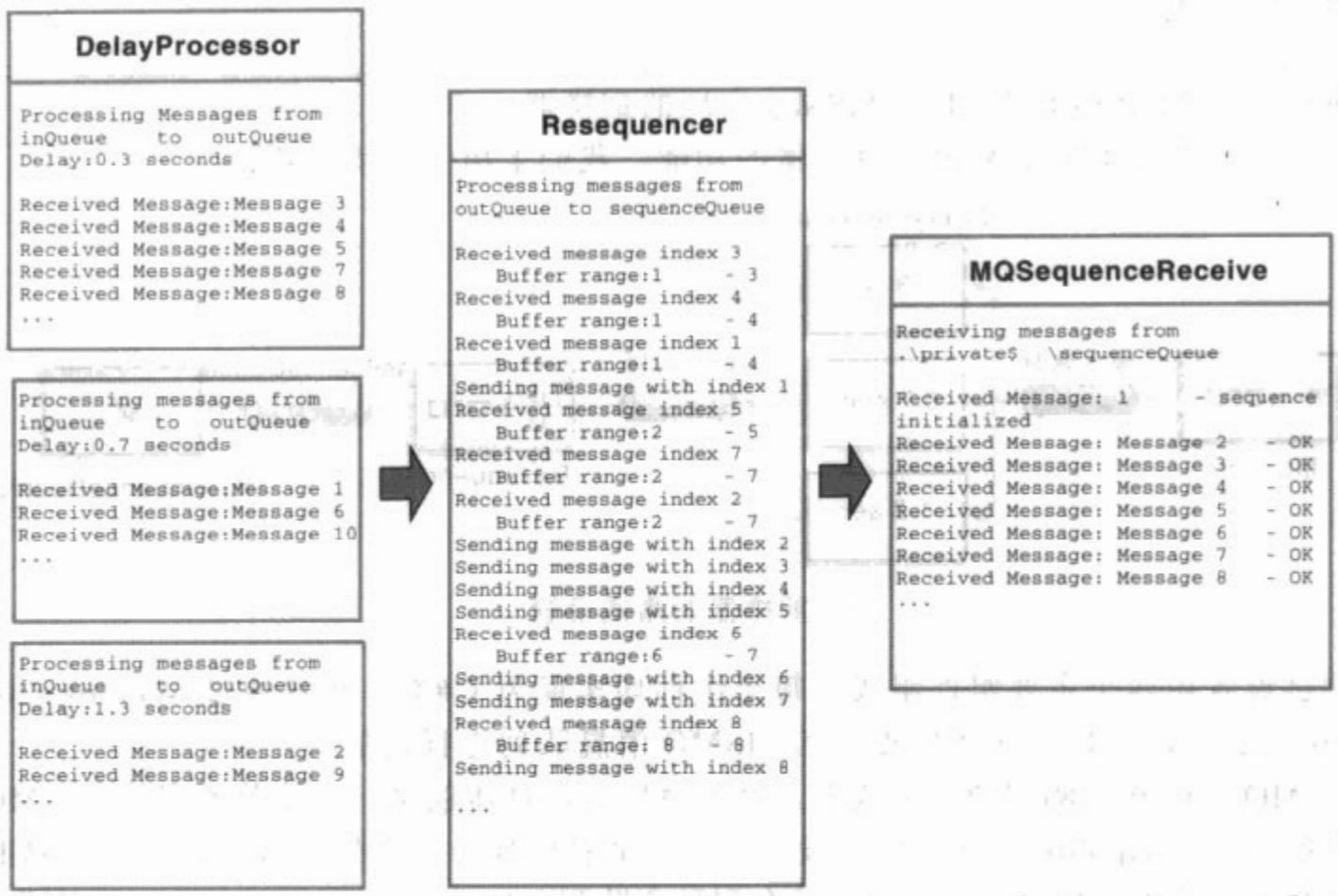
3. Resequencer 把乱序的输入消息缓冲起来，并按顺序把它们重新发布到 sequenceQueue 中。

4. MQSequenceReceive 从 sequenceQueue 中读取消息，检查位于 AppSpecific 属性中的消息序列号是否是按升序排列的。

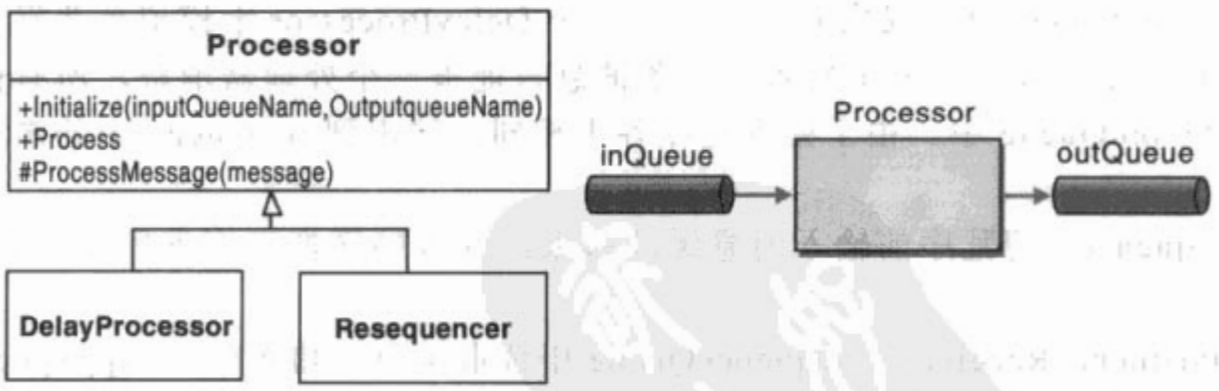
如果启动所有的组件，就会看到类似下图的输出。从处理器输出窗口的大小就可以看出处理器的不同处理速度。正如预期的那样，到达 Resequencer 的消息并不是有序的（在这次运行中，消息按 3，4，1，5，7，2...的顺序到达）。从 Resequencer 的输出可以看到，如果消息缺失，Resequencer 如何缓冲到来的消息。一旦缺失的消息到达，Resequencer 就把已经完成的消息序列按正确的顺序发布出去。

查看测试设置，会发现 DelayProcessor 与 Resequencer 有几点是相同的：它们都从输入队列读取消息，并把消息发布到输出队列中。它们唯一的区别只在于对消息的实际处理。

因此，可以创建一个公共的基类，把这种通用过滤器的基本功能封装起来（见“管道和过滤器”）。这个基类可以为创建队列和异步接收、处理、发送消息提供便利方法和模板方法。这个基类命名为 Processor（如下图所示）。



测试组件的输出



DelayProcessor 和 Resequencer 都继承自公共的 Processor 类

Processor 的默认实现只是把消息从输入队列复制到输出队列。为了实现 Resequencer，必须覆盖 ProcessMessage 方法的默认实现。在 Resequencer 中，ProcessMessage 方法把接收到的消息添加到缓冲区中，该缓冲区采用 Hashtable 实现。缓冲区中的消息采用消息序列号作为键，该键存储在 AppSpecific 属性中。一旦有新的消息加入，SendConsecutiveMessages 方法就会检查缓冲区中是否存在以下一个指定消息开始的连续消息序列。如果有，就将所有连续的消息发送出去，并从缓冲区中将这些消息删除。

Resequencer.cs

```
using System;
using System.Messaging;
using System.Collections;
using MsgProcessor;

namespace Resequencer
{

    class Resequencer : Processor
    {
        private int startIndex = 1;
        private IDictionary buffer = (IDictionary)(new Hashtable());
        private int endIndex = -1;

        public Resequencer(MessageQueue inputQueue, MessageQueue outputQueue)
                           : base (inputQueue, outputQueue) {}

        protected override void ProcessMessage(Message m)
        {
            AddToBuffer(m);
            SendConsecutiveMessages();
        }

    private void AddToBuffer(Message m)
    {
        Int32 msgIndex = m.AppSpecific;
        Console.WriteLine("Received message index {0}", msgIndex);
        if (msgIndex < startIndex)
        {
            Console.WriteLine("Out of range message index! Current start is: {0}",
                              startIndex);
        }
        else
        {
            buffer.Add(msgIndex, m);
            if (msgIndex > endIndex)
                endIndex = msgIndex;
        }
        Console.WriteLine("    Buffer range: {0} - {1}", startIndex, endIndex);
    }

    private void SendConsecutiveMessages()
    {
        while (buffer.Contains(startIndex))
```

```
            {
                Message m = (Message)(buffer[startIndex]);
                Console.WriteLine("Sending message with index {0}", startIndex);
                outputQueue.Send(m);
                buffer.Remove(startIndex);
                startIndex++;
            }
        }
    }

}
```

正如你所看到的，Resequencer 假定消息序列是从 1 开始的。如果消息产生者也从序号 1 开始，并且这两个组件在整个生命期中始终保持相同的顺序，则不会出现什么问题。为了提高 Resequencer 的灵活性，消息的产生者应该在发送序列中的第一条消息前，与 Resequencer 协商一个起始顺序号。这个过程类似于 TCP 协议连接期间交换的 SYN 消息(见 [Stevens])。

而且，当前的实现没有考虑缓冲区溢出问题。假设 DelayProcessor 中止运行或出现异常，导致消息丢失，Resequencer 将无限期等待缺少的消息，直到缓冲区溢出。在大容量传送消息的情况下，消息的生产者必须与 Resequencer 协商一个窗口大小，这个窗口大小要能反映 Resequencer 可缓冲的最大消息数。一旦缓冲区满，错误处理程序必须对如何处理缺少的消息做出决定。例如，生产者可以重传消息，或插入“哑”消息。

Processor 基类的实现相当简单。它使用 BeginReceive 和 EndReceive 方法完成异步消息处理。由于在消息处理结束时很容易忘记调用 BeginReceive，我们使用了一个模板方法(template method)，其中集成了这个操作。子类可以覆盖 ProcessMessage 方法，而不必担心异步处理问题。

Processor.cs

```
using System;
using System.Messaging;
using System.Threading;

namespace MsgProcessor
{
    public class Processor
    {
        protected MessageQueue inputQueue;
        protected MessageQueue outputQueue;

        public Processor (MessageQueue inputQueue, MessageQueue outputQueue)
        {
```

```
        this.inputQueue = inputQueue;
        this.outputQueue = outputQueue;
        inputQueue.Formatter = new System.Messaging.XmlMessageFormatter
                                    (new String[] {"System.String,mscorlib"});
        inputQueue.MessageReadPropertyFilter.ClearAll();
        inputQueue.MessageReadPropertyFilter.AppSpecific = true;
        inputQueue.MessageReadPropertyFilter.Body = true;
        inputQueue.MessageReadPropertyFilter.CorrelationId = true;
        inputQueue.MessageReadPropertyFilter.Id = true;
        Console.WriteLine("Processing messages from " + inputQueue. Path +
                          " to " + outputQueue.Path);
    }

    public void Process()
    {
        inputQueue.ReceiveCompleted += new
            ReceiveCompletedEventHandler(OnReceiveCompleted);
        inputQueue.BeginReceive();
    }

    private void OnReceiveCompleted(Object source,
                                    ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;

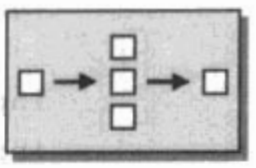
        Message m = mq.EndReceive(asyncResult.AsyncResult);
        m.Formatter =  new System.Messaging.XmlMessageFormatter
                           (new String[] {"System.String,mscorlib"});

        ProcessMessage(m);

        mq.BeginReceive();
    }

    protected virtual void ProcessMessage(Message m)
    {
        string body = (string)m.Body;
        Console.WriteLine("Received Message: " + body);
        outputQueue.Send(m);
    }
}
}
```

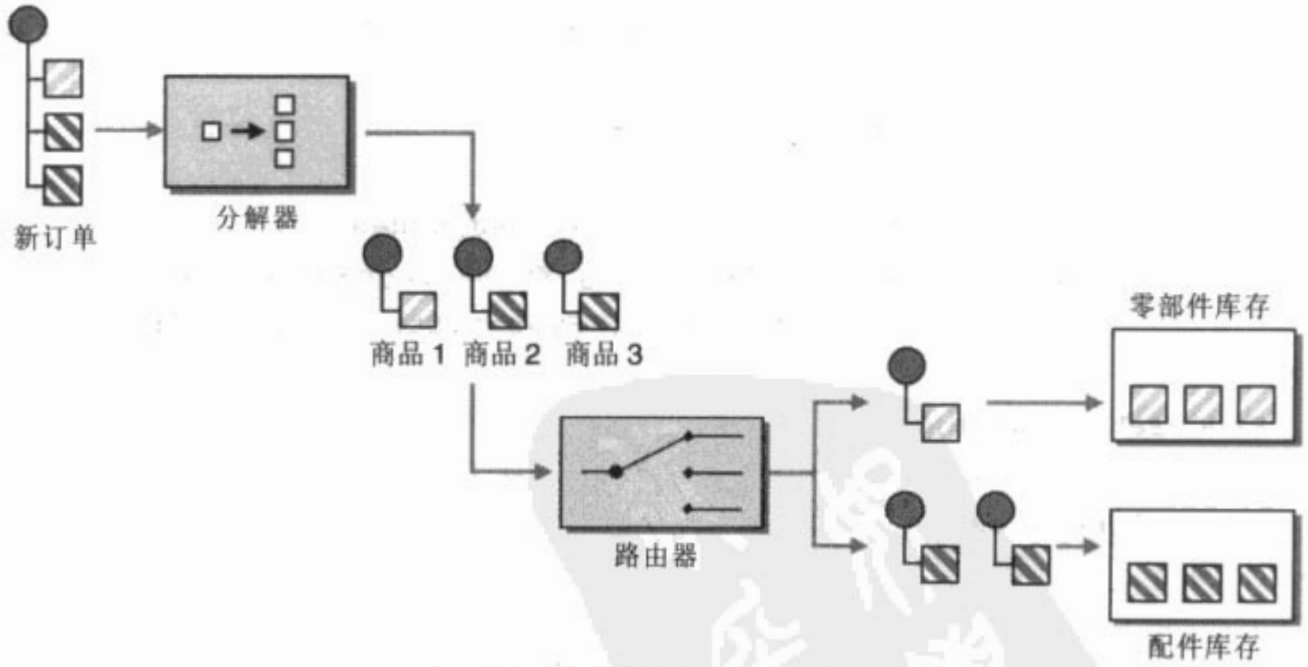
7.9 复合消息处理器



在“基于内容的路由器”和“分解器”模式中，举了一个订单处理的例子，其中要处理包含多个商品的输入订单。每行商品需要由各自对应的库存系统完成库存检查。在所有商品均检查完毕后，还要把验证过的订单消息传递给下一个处理过程。

如果要处理的消息包含多个元素，而且每个元素需要不同的处理，那么如何维持整个消息流？

这个问题看起来包括许多前面定义的模式的元素。分解器可以把单个消息划分为多个组成部分。基于内容的路由器可以根据消息的内容或类型把分解后的各个子消息路由给正确的处理过程。利用管道和过滤器体系结构，可以把这两种模式链接起来，从而把组合消息中的每项元素路由到适当的处理过程。



把分解器和路由器组合起来

在我们这个例子中，这意味着每个商品将路由给正确的库存系统进行检验。库存系统彼此之间是解耦合的，每个系统只接收它们能处理的商品。

采用目前这种设置的缺点是，我们不知道订购的所有商品是否都有库存并且可发货。我们还想得到所有商品的价格（考虑到大宗商品折扣），并把它们集中记录到一个发货单中。这就要求把订单作为单一的一个消息继续处理，尽管我们刚把它分解为多个

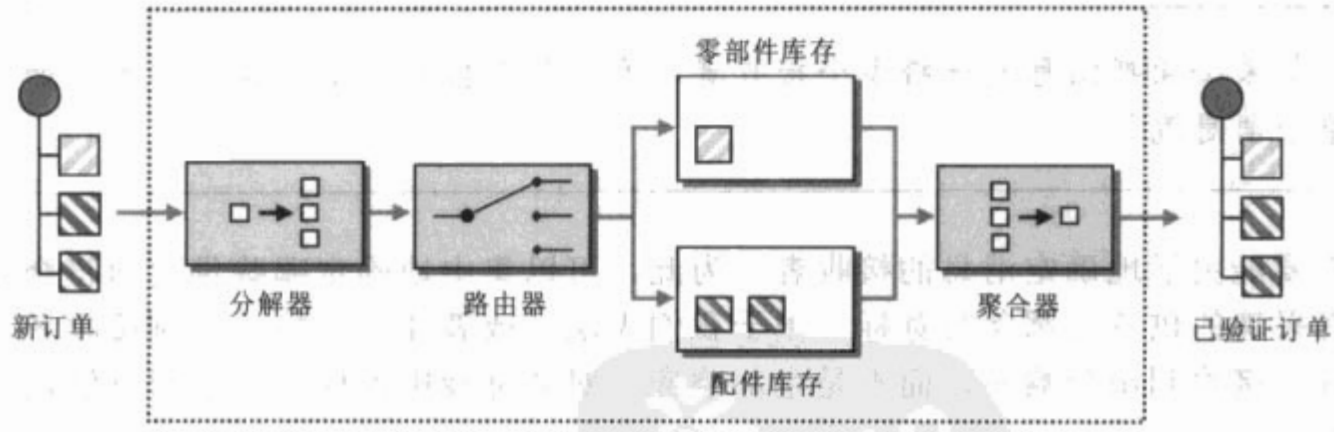
子消息。

一种方法是把经某个特定库存系统处理的所有商品重新组装成一个单独的订单消息。在此之后，就可以作为一个整体来处理这个订单：订单可以履行和发货，并寄出相应的账单。每个子订单都采用独立的过程处理。在某些情况下，由于缺乏对后续处理过程的控制，这种方法成为唯一可行的解决方案。例如，Amazon 公司就采用这种方法销售绝大部分的商品。订单将路由给不同的供货商，并交由它们分别处理。

但是，采用这种方法并不能提供最佳的客户服务。客户可能会收到多批商品和多个发货单，而且很难解决商品退货或争议问题。对订购图书的消费者来说，这些还不算是大问题，如果每个商品彼此依赖，问题会更棘手。假设订单中包括许多家具商品，这些商品可以组装成一套壁柜。如果收到许多包有家具材料的大箱子，但其中没有找到所需的装配零部件，而这些零部件要过些日子才能送到，客户肯定不愿意看到这种情形。

与同步方法调用相比，消息传递的异步特性使任务的分配变得更加复杂。可以分派各个商品，并等待响应返回后再检查下一个商品。这种做法能减少临时的关联，但是会极大降低系统的效率。我们更愿意让各个系统同时处理订单。

使用复合消息处理器（Composed Message Processor）处理复合的消息。复合消息处理器把消息分解开，并把子消息路由给相应的目标，最后把响应结果重新组合为一个消息。



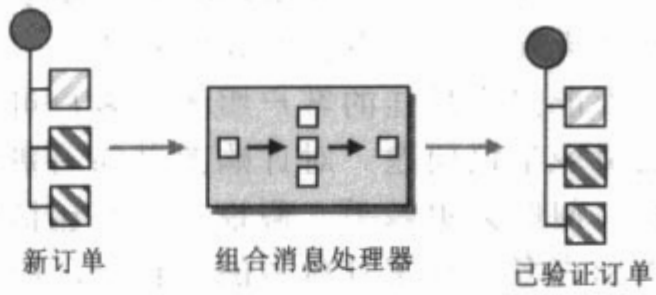
复合消息处理器

复合消息处理器使用聚合器把分派给多个库存系统的请求重新协调起来。每个处理单元都向聚合器发送一条响应消息，说明指定商品是否有存货。聚合器搜集各个响应并按照预先设定的算法进行处理。

由于所有子消息都来自同一消息，还可以向聚合器传递额外的信息，如子消息编号，以便定义更有效的聚合策略。此外，复合消息处理器还要解决消息缺失或延迟带来的问题。如果库存系统不可用，而且订单中还有属于该系统的商品，是否要因此推迟这些商品的处理？或者把这些商品路由给异常队列，让操作员人工评估？如果某条响应消息丢失了，是

否要重新发送库存请求消息？对于这些折衷方案的详细说明，请参见“聚合器”。

这种复合模式说明了如何把不同的模式组合成一个更大的模式。对系统中的其他部分来说，复合消息处理器就像一个简单的过滤器，它有一个输入通道和一个输出通道。因此，这为更复杂的内部工作提供了有效的抽象。



复合消息处理器作为简单的过滤器

7.10 分散收集器

前面介绍各种模式时，谈到了一个订单处理示例，当前库存中没有的商品都可由某个外部供货商提供。但是对于不同的商品，有的供货商有现货，有的可能没有；同一商品开出的价格不同，或者供货的日期不同。为了尽可能以最佳的方式满足订单需求，必须从所有供货商那里获得报价信息，从中选出能满足需求的最好的一家。

如果必须把消息发送给多个接收者，而且每个接收者都会返回应答，如何维护整个消息流？

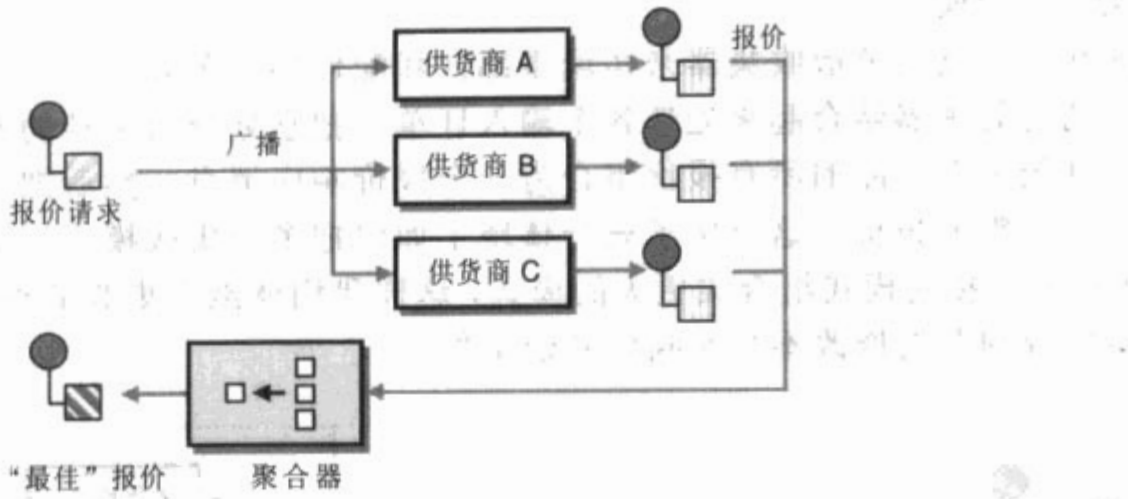
首先要能灵活地确定消息的接收者。为此，可以集中地确定哪些供货商符合要求，或让任何有兴趣的供货商都参与竞标。由于我们无法（或者几乎不能）控制接收者，因此要做好准备只接收到部分响应，而不是全部响应。对竞标规则的修改不能影响解决方案的结构完整性。

解决方案应该使每个接收者的编号和身份对后续的处理隐藏。把消息的分配封装在局部，从而使其他组件独立于每条消息的路由。

另外还要协调好后续的消息流。最简单的解决方法是让每个接收者把应答发送到一个通道，让后续的组件负责各个消息的解析。但是，这要求后续的组件要了解发送给多个接收者的消息。如果后续组件不清楚当前采用的路由逻辑，要想处理各个消息会更困难。

可以把路由逻辑、接收者以及各个消息的后续处理结合到一个逻辑组件中。

使用分散收集器把消息广播给多个接收者，再把响应的消息重新组合为一个消息。



分散收集器把请求消息路由给多个接收者。然后使用聚合器把响应消息收集起来，组合成一个响应消息。

有两种类型的分散收集器，它们使用了不同的机制把请求消息发送给目标接收者。

1. **分配**（Distribution）方式：利用一个接收表，采用分配方式，分散收集器可以控制接收者列表，前提是它必须知道每个接收者的消息通道。

2. **拍卖**（Auction）方式：采用拍卖方式，分散收集器使用发布-订购通道把请求广播给所有感兴趣的参与者。这种方式允许分散收集器使用单个通道，但是必须放弃对接收者的控制。

这种解决方法与复合消息处理器模式很相似。它没有使用分解器，而是通过发布-订购通道把完整的消息广播给所有相关的组件。可以在消息中添加返回地址，让所有响应消息都通过一个通道处理。与复合消息处理器一样，分散收集器要根据预先定义的业务规则把响应消息组合起来。在我们的例子中，聚合器要从能履行订单的供货商中选出最佳的报价。与采用复合消息处理器相比，使用分散收集器来实现响应消息的聚合更为困难，因为我们可能不知道有多少个接收者参与信息交互。

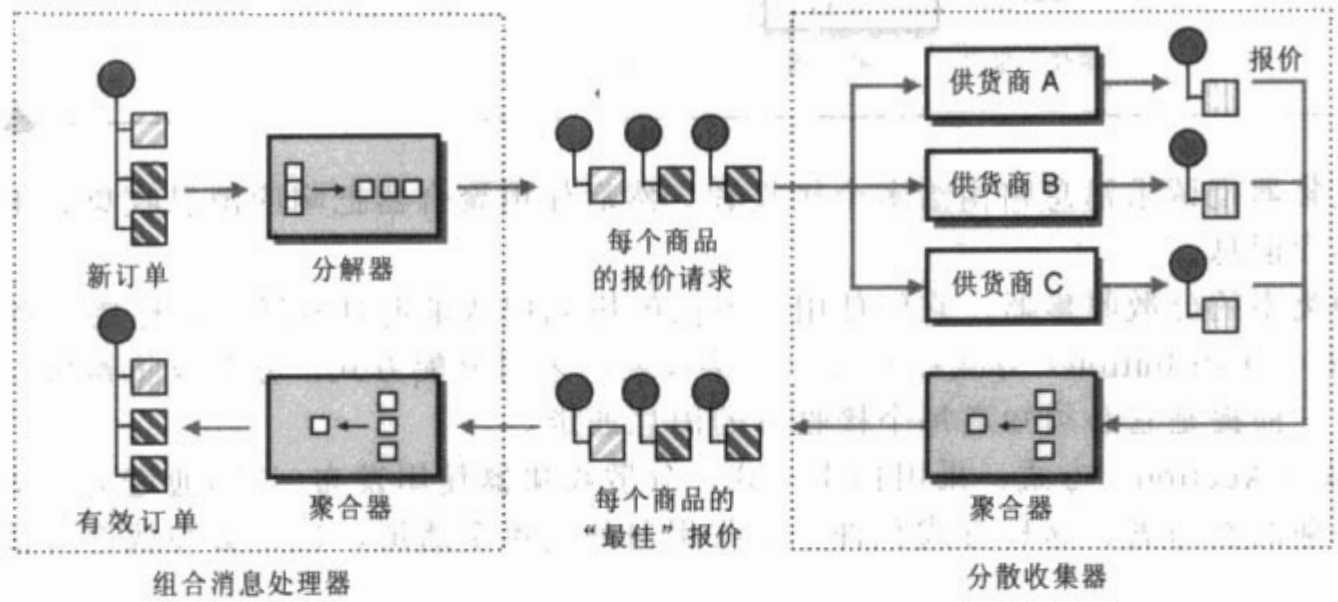
分散收集器和复合消息处理器都把单个消息路由给多个接收者，并使用聚合器把返回的应答消息重新组合成一个消息。复合消息处理器要对多个并发的活动组件进行同步。如果每个组件所花的时间有很大差异，即使已经完成了大多子任务（甚至除一个子任务以外的其他所有子任务都已完成），后续的处理也要被阻塞。因此在设计方案时，必须把这一特性与分散收集器的简单性和封装性权衡考虑。一种折衷的解决方案是采用层联聚合器。即只有部分结果可用的情况下后续的任务也可以启动。

例：贷款中介

贷款中介的例子（见第 9 章“中场演练：复合消息传递”）使用分散收集器把查询贷款利率的请求路由给多家银行，并从返回的响应中选出最优惠的利率。该例子分别采用基于接收表的解决方案（见第 9 章的“使用 MSMQ 的异步实现方案”）和发布-订购通道（见第 9 章的“使用 TIBCO ActiveEnterprise 的异步实现方案”）实现了相同的解决方案。

例：结合模式

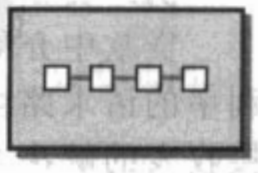
现在可以使用分散收集器来实现零部件和配件的订单处理示例了。可以把分散收集器与复合消息处理器结合起来处理各个输入订单，把它划分为单独的商品，查询每个商品的报价，并把每个商品的所有报价组合为一个报价响应消息，然后把所有的报价响应消息组合为一个完整的报价。这个例子充分体现了如何把多个集成模式组合为一个完整的解决方案。通过把单独的模式组合成更大的模式，这样我们就能在更高的抽象层次探讨解决方案，而且对实现细节的修改不会影响到其他组件。



结合分散收集器与复合消息处理器

这个例子还展示了聚合器的多种功能。例子中使用的两种聚合器有不同的作用。存在于分散收集器中的第一个聚合器可以从多个供货商中选出最佳的报价。该聚合器不要求所有的供货商都响应请求（速度可能比低价格更重要），但是要采用复杂的响应组合算法。例如，订单中可能包含 100 个零部件，而报最低价格的供货商只能提供 60 个零部件。聚合器必须决定是否接受其供货，然后从其他供货商那里订购剩余的 40 个零部件。存在于复合消息处理器中的第二个聚合器可能要简单一些，因为它只是把从第一个聚合器那里接收到的所有响应消息串连起来。但是，这个聚合器需要确保所有的响应都已收到，并且还要处理如响应消息丢失之类的错误情况。

7.11 路由表



迄今为止所介绍的大多数路由模式都是根据一组规则把输入消息路由给一个或多个目

标。但是，有时我们不但要把消息路由给一个组件，而且要让它通过一系列组件。例如，在采用管道和过滤器模式处理输入消息时，必须让消息有序地经过一系列的处理步骤和业务规则的验证。由于验证规则本身存在很大的区别，而且是由外部系统决定的（如信用卡的验证），因此要把每个处理步骤实现为不同的过滤器。每个过滤器采用各自的业务规则检查输入的消息。如果消息不能满足规则指定的条件，将被路由到一个异常通道。过滤器之间的通道决定了消息必须经过的验证序列。

现在，假设对每条消息实施的验证操作是由消息的类型决定的（例如，订单请求不需要信用卡验证，或者通过 VPN 发送订单的客户不需要解密和认证）。为了满足这一要求，必须找到一种方法，能根据消息的类型让消息经过不同的过滤器序列。

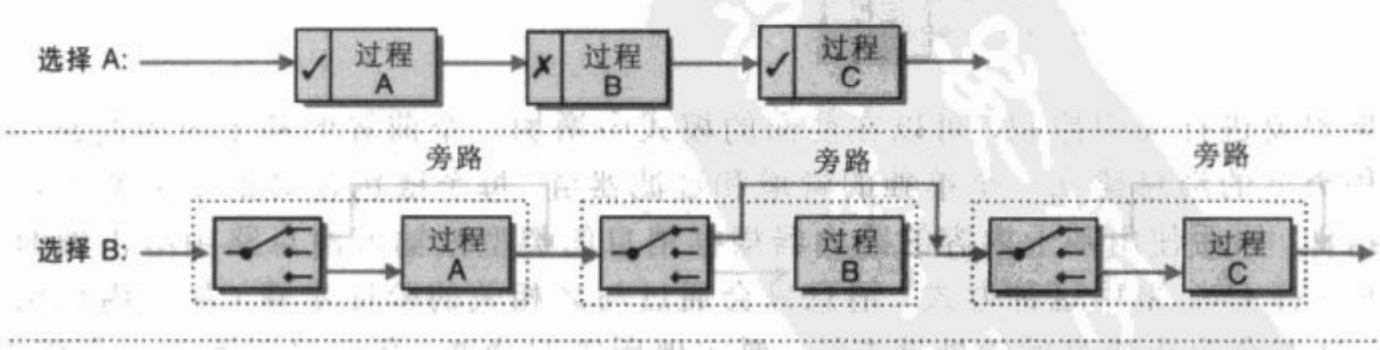
> 如果设计时不清楚消息处理的步骤，而且每条消息的处理各不相同，那么如何让消息通过一系列处理步骤连续地得到路由？

管道和过滤器体系结构提供了一个很好的方法，可以把一系列处理步骤表示为用管道（通道）连接起来的独立的过滤器。在默认配置下，过滤器由固定的管道连接。如果想把消息动态路由给不同的过滤器，可以使用特殊的过滤器作为消息路由器。该路由器能动态确定路由该消息的下一个过滤器。

对于这个问题，要得到一个好的解决方案，关键需求可以总结为以下几点：

- 高效的消息流：消息只能流经必要的处理组件，不经过多余的组件。
- 高效地利用资源：该解决方案不能占用大量通道、路由器和其他资源。
- 灵活性：每条消息的路由很容易改变。
- 易于维护：如果需要支持新的消息类型，最好不要因为某一点的维护带来新的错误。

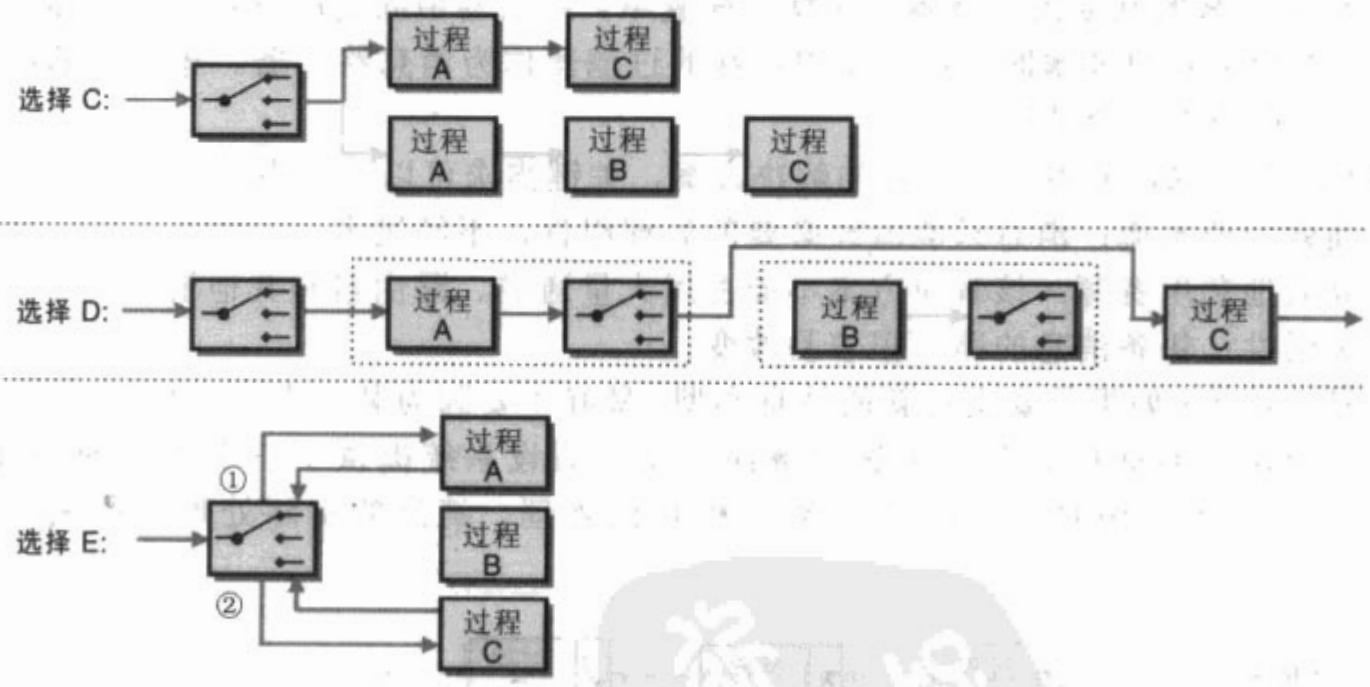
下图给出了对此问题各种可选的解决方案。假设系统提供了三个不同的处理步骤，A、B 和 C。当前的消息只需经过步骤 A 和 C 的处理。消息的实际处理流程采用粗箭头标记。



可以建立一个长长的管道和过滤器链，包括所有可能的验证步骤，并对每个路由器编写代码，如果该路由器对应的验证步骤对于所传递的消息类型不是必要的，则绕过这个路由器（见选择 A）。这种选择实质上采用了“消息过滤器”中介绍的反应型过滤方法。尽管这种方法非常简单，但是组件中混合了业务逻辑（验证）和路由逻辑（决定是否要验证），

因此很难得到重用。而且，两种不同类型的消息很可能必须经过相同的处理步骤，但是采用的处理顺序不同。如果采用这种硬编码的方法，则很难支持这样的需求。

为了改善问题的分离，增强解决方案的可组合性，应该把每个组件内部的“门控”逻辑替换为“基于内容的路由器”。这样得到的是一个由所有可能的验证步骤组成的链，每个验证步骤前面都有一个基于内容的路由器，如选择 B 所示。当消息到达路由器时，路由器检查消息的类型并确定该类型的消息是否需要经过当前的验证步骤。如果需要，路由器就会路由此消息，使之得到验证。如果不需要，路由器则绕过验证，并把消息直接路由到下一个路由器［采用的方法与“旁路（Detour）”模式非常相似］。每个操作步骤与其他步骤彼此独立时，采用这种配置会有很好的效果，路由的决定可由每个操作步骤独自做出。这种方法的缺点是，即使只需执行很少的验证步骤，该配置也要让消息经过一个很长的路由器序列。实际上，每条消息通过所有可能的组件要花费两倍的时间。如果组件很多，即使完成一个相当简单的功能也会引起大量的消息流。而且，路由逻辑被分散到多个过滤器中，因此很难确定某个特定类型的消息到底经过哪些验证步骤的处理。如果要引入新的消息类型，就要更新各个路由器。最后一点，这种选择同样会存在选择 A 的限制，因为消息必须按照预先确定的顺序进行处理。



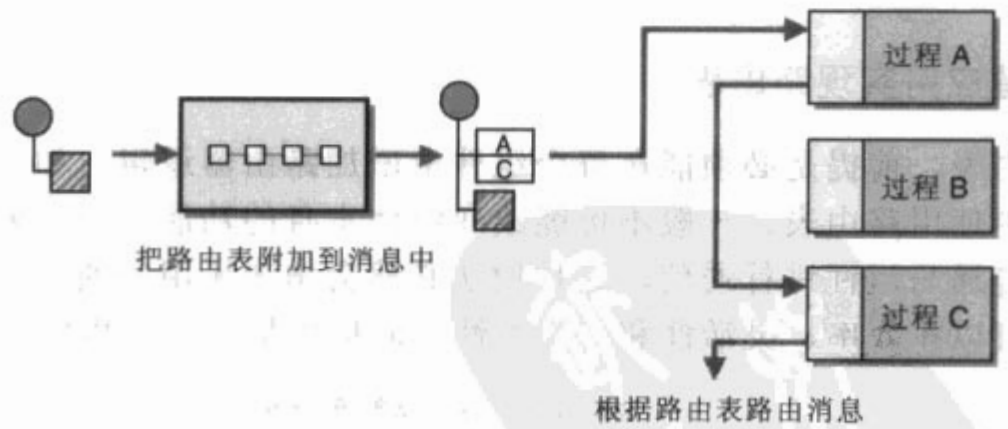
如果想要进行集中控制，可以在前面的模式中增加一个前置的基于内容的路由器，并且为每种类型的消息建立一个单独的管道和过滤器链。每个链包含对指定类型消息的一系列验证组件。然后使用基于内容的路由器根据消息的类型把输入消息路由给正确的验证组件链（见选择 C）。采用这种方式，消息只会通过与之相关的验证步骤（加上内部路由器）。因此，它是迄今为止最有效的处理方法，要实现所需的功能，我们只需增加一个路由步骤。这种解决方案明确地给出了指定类型消息的路径。但是，它必须预先确定各种可能的验证规则组合，相同的组件可能会在多条路径中使用。这种方法要求运行这种组件的多个实例，从而会导致不必要的重复。如果消息的类型很多，就必须维护大量的组件实例和相关的通道，这会使维护变得相当困难。总之，这种非常高效的解决方案是以降低可维护性为代价

的。

要想避免提前固定各种可能的验证步骤组合，需要在各个验证步骤之间插入一个基于内容的路由器（见选择 D）。为了克服采用反应型过滤方法带来的相关问题（见选择 B），要在每个处理步骤的后面（而不是前面）插入一个基于内容的路由器（只需在所有验证步骤的最前面添加一个额外的路由器）。这些路由器能够直接把消息中继传递给验证链中下一个必要的验证步骤，而不是盲目地把消息路由给下一个可用的验证步骤。从理论上看，这种解决方案与反应型过滤的方案很相似，因为消息要经过一组可替换的路由器和过滤器。但是在这种解决方案中，路由器不只是做出是或否的决定，它还能消除不必要的验证步骤。例如，消息只需经过两个路由器，而不像选择 B 那样经过三个路由器。选择 D 提供了灵活性和高效率，但是仍然没有实现集中控制的目标，我们还是要维护大量潜在的路由器，因为路由逻辑分配在一系列独立的路由器中。

为了克服最后的一个不足，可以把所有的路由器组合为一个“超级路由器”（见选择 E）。在每个验证步骤之后，消息将被路由回到超级路由器，由它决定验证消息的下一个步骤。如果采用这种配置方案，只会把消息路由给与其特定类型相关的过滤器。由于所有路由决定都集中在一个路由器中，必须设计一种机制能记住哪些步骤已经完成处理。因此，超级路由器肯定是有状态的，或者每个过滤器必须在消息中附加一个标记，告诉超级路由器消息经过的最后一个过滤器是谁。而且，每个验证步骤要求消息必须经过两个通道：到组件的通道和从组件返回超级路由器的通道。这就导致消息通信量比 C 选择多出一倍。

把路由表附加到每条消息中，指定处理步骤序列。采用特定的消息路由器包装每个组件，它读取路由表并把消息路由给表中的下一个组件。



我们在处理过程的起始处插入一个特殊的组件，它计算每条消息所需的处理列表，然后把这个列表作为路由表附加到消息中，并把消息路由给第一个处理步骤。每个处理步骤在成功完成处理后，查看路由表并把消息传递给路由表中指定的下一个处理步骤。

在一个小组或部门中，可能会为杂志附加一个路由表来实现正常流通，与此类似，路由表模式的工作原理是一样的。唯一不同的是，路由表定义了消息必须经过的组件序列，而在大多数公司中，你可以把读过的杂志交给列表中尚未读过杂志的任何人（当然，老板

一般要先读）。

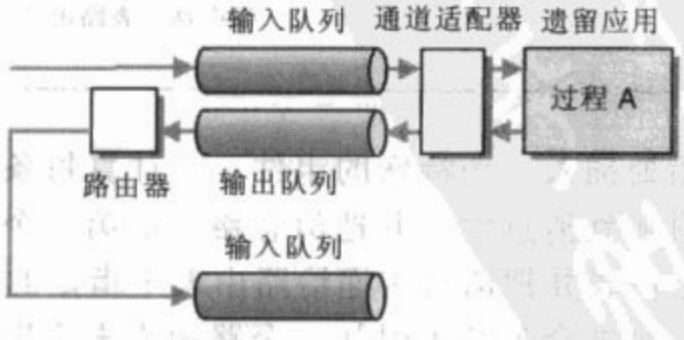
路由表结合了超级路由器方法（选择 E）的集中控制和固定方式解决方案（选择 C）所具有的高效率。我们在最开始确定完整的路由方案，并把它附加到消息中，之后就不用把消息返回到中心路由器再做路由决策处理。每个组件具备简单的路由逻辑。在提出的解决方案中，我们假设这种路由逻辑是建立在处理组件内部的。我们曾经说过，选择 A 的缺点是要把部分路由逻辑硬编码到每个组件中，那么采用路由表又有什么不同，为什么它比选择 A 要好？它们之间的关键区别在于，路由表中使用的路由器是通用的，路由逻辑的修改不会影响路由器。放到每个组件中的路由逻辑类似于返回地址，即使部分路由逻辑建立在组件中，组件仍能保持可重用性和可组装性。另外，路由表的计算可由一个集中的组件完成，而不必改动每个处理组件内部的代码。

同样，天下没有免费的午餐，路由表也有一定的局限性。首先，消息的大小有所增加。在大多数情况下，消息大小的增加不会带来显著的问题，但是，如果在消息的内部携带有处理状态信息（即哪个步骤已经完成），就会产生其他副作用。例如，如果消息丢失，那么我们损失的不仅是消息数据，还会丢失处理的状态数据（如消息的下一个目标）。因此在很多情况下，可以在一个集中的位置维护所有消息的状态，以实现错误报告或错误恢复。

路由表的另一个限制是，一旦开始消息的处理，消息的路径就无法改变。也就是说，消息的路径不能由处理步骤产生的中间结果决定。然而，在许多实际的业务处理过程中，消息流要根据中间处理结果改变。例如，要根据订单商品的可用性（由库存系统提供这方面的信息），把消息路由给不同的路径。这就要求集中的组件必须预先确定消息可能经过的所有处理步骤。与使用基于内容的路由器一样，这就从一定程度上降低了设计方案的可用性。

7.11.1 采用遗留应用实现路由表

如果使用路由表，前提是必须能在每个组件中增加路由器逻辑。如果要在遗留的应用或封装好的应用中使用路由表，一般不可能改变组件本身的功能，而是要使用一个外部路由器，通过消息传递与组件进行通信。这种做法必然会增加所用的通道和组件的数量。但是，路由表仍然可以在效率、灵活性和可维护性之间提供最佳的折衷解决方案。



采用遗留应用实现路由表

7.11.2 一般用法

路由表主要用在以下几种情况下：

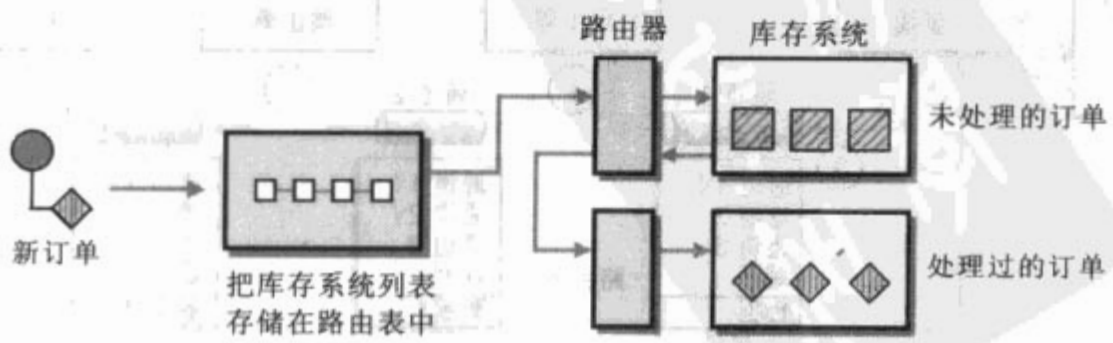
1. 按顺序执行二元验证步骤。由于不在消息中添加信息，因此在处理消息的过程中可以改变消息的路由。通过重新配置集中的路由表，就能灵活地改变消息验证步骤序列。每个组件可以选择出现错误时放弃指定的处理序列，或者把消息传递给下一个处理步骤。

2. 每个处理步骤均是无状态的转换。例如，假设我们从各种商业伙伴那里接收订单。所有的订单都从公共的通道到达，但是不同商业伙伴提供的订单消息有不同的格式。因此，每条消息必须经过不同的转换步骤。来自某些伙伴的消息需要解密，而来自其他伙伴的消息不用。有些消息需要转换或组合，其他的则不必。为每个伙伴安排一个路由表，这样有利于集中地重新配置针对每个伙伴的消息处理步骤。

3. 每个步骤只是搜集数据，但是不做路由决策（见“内容扩充器”）。有时我们收到的消息包含了指向其他数据的引用标识符。例如，如果收到一个 DSL 线路的订单，这个消息可能只包含申请者的家庭电话号码，因此必须通过外部数据源获得客户的姓名、提供 DSL 线路服务的中心办事处、到中心办事处的距离等信息。只有得到所有相关数据后，才能确定应该给客户提供什么消息包。在这种情况下，路由决策被推迟到最后，因此我们可以使用路由表搜集必需的信息。不过，需要评估一下是否确实需要路由表所提供的灵活性，否则使用固定的管道和过滤器链就足够了。

7.11.3 采用路由表实现简单的路由器

基于内容的路由器存在这样一个缺点，它必须掌握每个可能的接收者的信息，还要知道与该接收者相关的路由规则。按照松耦合的精神，不应该有这样一个包含其他多个组件信息的集中组件。为此可以采用发布-订购通道以及“消息过滤器”中介绍的消息过滤器阵列，来代替基于内容的路由器，让每个接收者自己决定应该处理哪条消息，但是这又可能会面临重复处理消息的风险。另一种替代方法是使用改进的路由表模式，实现职责链[GoF]，让每个接收者自己决定是否处理给定的消息。职责链允许每个组件接受消息或者把消息路由给列表中的下一个组件。路由表一个静态列表，包括所有可能的接收者。也就是说，集中组件仍然需要掌握所有接收者的信息，但是它不需要知道每个组件消费哪条信息。



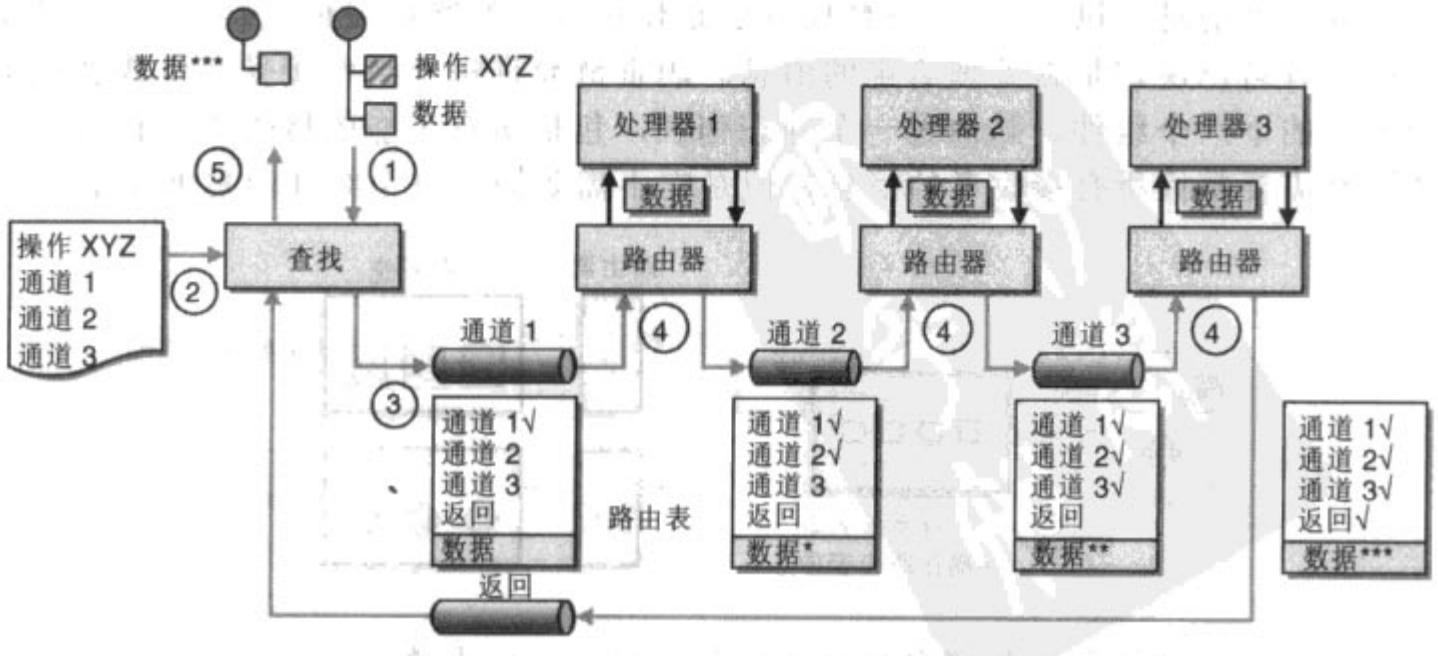
使用路由表可以避免重复消息处理的风险。另外，如果消息未被任何组件所处理，也

很容易知道。这种方法最大的缺点是消息处理的速度更慢，并且增加了网络通信量。不论实际系统有多少，基于内容的路由器只会发布一条消息，而路由表发布的平均消息数等于系统数量的一半。在安排系统时，如果让更有可能处理消息的系统排在前面，则可以减少发布消息的数量，但是与预测型的基于内容的路由器相比，发布的消息还是会更多。

有些情况下，我们要获得比简单顺序表更多的控制，或者要根据中间处理结果改变消息流。过程管理器可以满足这些需求，因为它支持分支条件、分叉和合并。实际上，路由表就是动态可配置业务过程的一个特例。应该认真考虑到底使用路由表还是使用集中的过程管理器。动态路由表不但有集中维护的优点，而且能提供固定解决方案的高效率。但是，由于路由状态信息分布到消息中，随着复杂性的增加，分析与调试系统将变得越来越困难。而且，由于过程定义的语义包含了诸如判定、分叉和合并等结构，使得配置文件变得难以理解和维护。可以在路由表中包含条件语句，改进每个组件中的路由模块，从而解释条件命令来确定下一路由位置。但是，必须注意的是，千万不要增加额外的功能而影响这种解决方案的简单性。如果需要如此复杂，最好放弃路由表的运行时效率，而使用功能更为强大的过程管理器。

例：路由表作为一个复合服务

当创建面向服务的体系结构时，一个简单的逻辑功能往往由多个独立的处理步骤组成。出现这种情况主要有两个原因。第一，封装好的应用一般根据其内部 API 提供细粒度的接口。当把这些程序集成到一个综合解决方案中时，必须在更高抽象层次上运行。例如，“新账户”操作是由记账系统内部的多个操作步骤组成的：如创建一个新的客户，选择服务计划，设置地址属性，检查信用数据等。第二，一个简单的逻辑功能可能是由多个系统实现的。我们想对其他系统隐藏这一事实，以便能在系统之间灵活地重新分配职责，而不会影响集成解决方案中的其他系统。作为对一个请求消息的响应，使用路由表可以执行所需的多个内部处理步骤。路由表提供了灵活性，可以执行同一个通道的不同请求。虽然它执行了一系列单独的操作，但是从外面看起来就像是一个操作（如下图所示）。



使用路由表实现复合的服务

1. 输入的请求消息（指定了所需操作和所有必要的数据）发送给查找组件。

2. 查找组件从一个服务目录中获得与所需操作有关的处理步骤列表。它把通道列表添加到消息首部（每个通道等同于一个细粒度的操作）。查找组件把返回通道添加到列表中，从而将处理完的消息返回查找组件。

3. 查找组件把消息发布到对应第一个操作的通道。

4. 每个路由器从队列中读取请求消息，然后把它传递给服务提供者。在完成处理后，路由器把这个操作标记为“已完成”，并把消息路由给路由表中指定的下一个通道。

5. 查找组件从返回通道中消费消息，并把消息转发给请求者。从外部来看，上述的整个处理过程就像完成了一次简单的请求/应答消息交换。

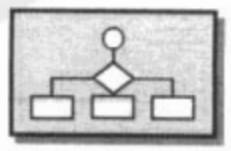
例：WS-Routing

Web 服务请求经常要通过多个中间组件进行路由。为此，Microsoft 制定了 Web 服务路由协议（WS-Routing）规范。WS-Routing 是一种基于 SOAP 的协议，能把发送者发出的消息经过一系列的中间组件路由传给接收者。WS-Routing 的语义要比路由表的语义丰富，不过路由表则很容易在 WS-Routing 中实现。下面的例子给出了一个消息的 SOAP 首部，这个消息将从节点 A 通过中间节点 B 和 C 路由到节点 D（用元素<wsrp:via>表示）。

```
<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://www.w3.org/2001/06/soap-envelope">
  <SOAP-ENV:Header>
    <wsrp:path xmlns:wsrp="http://schemas.xmlsoap.org/rp/">
      <wsrp:action>http://www.im.org/chat</wsrp:action>
      <wsrp:to>soap://D.com/some/endpoint</wsrp:to>
      <wsrp:fwd>
        <wsrp:via>soap://B.com</wsrp:via>
        <wsrp:via>soap://C.com</wsrp:via>
      </wsrp:fwd>
      <wsrp:from>soap://A.com/some/endpoint</wsrp:from>
      <wsrp:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</wsrp:id>
    </wsrp:path>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

和大多数 Web 服务规范一样，WS-Routing 也会随着时间改进，并/或与其他规范合并。这里给出的例子只反映了 Web 服务群体目前对路由的考虑。

7.12 过程管理器



路由表展示了如何让消息路由经过一系列动态的处理步骤。路由表解决方案要建立在

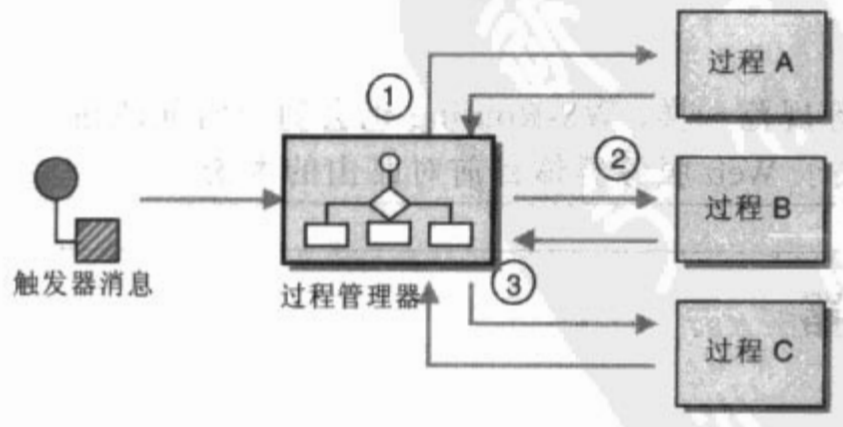
两个关键的前提条件下：必须事先确定处理步骤序列，并且该序列是线性的。但在很多情况下，这些前提条件无法得到满足。例如，要根据中间处理结果决定如何路由。或者，处理步骤不是顺序执行的，多个步骤可能并发执行。

如果无法在设计时确定所需的处理步骤，或者处理步骤不是顺序执行的，应该如何通过多个处理步骤完成消息的路由？

管道和过滤器体系结构有一个主要优点，能够利用通道（“管道”）把各个处理步骤（“过滤器”）组合为一个处理序列。各个消息则要经过这个处理单元（或组件）序列。如果想改变每条消息的处理序列，可以使用多个基于内容的路由器。这种解决方案提供了最大的灵活性，但是它的缺点是，必须把路由逻辑分散到多个路由组件中。通过事先计算消息的路径，路由表提供了集中的控制点，但是它无法按照消息的中间处理结果灵活地重新路由消息，也不能同时执行多个处理步骤。

在完成每个独立单元的处理后，如果能把控制交还给一个中央组件，由该中央组件确定下一个要执行的处理单元，就能在获得灵活性的同时保持集中的控制点。采用这种方法，将形成一个交替的过程流：中央组件，处理单元，中央组件，处理单元，如此循环下去。这样中央组件将接收每个独立处理步骤产生的消息。当消息到达时，中央组件必须根据中间处理结果以及当前步骤在处理序列中的位置来确定下一个要执行的处理步骤。这就要求每个处理单元必须向中央单元返回足够的信息，才能做出这样的决定。但是，采用这种方法会使处理单元对中央单元的存在产生依赖，因为它们必须传递只与中央组件有关的额外信息，这些信息与处理单元无关。如果我们想让每个处理步骤及其相关消息格式与中央单元解耦合，就必须为中央单元提供某种形式的“记忆”，告诉它序列中最后执行的是哪个步骤。

使用中央处理单元（即过程管理器）维护序列的状态，并根据中间处理结果确定下一处理步骤。



首先要说明的是，过程管理器的设计与配置是一个非常宽泛的主题。仅是与工作流或

业务过程管理相关的模式可能就要用整本书来介绍(或许我们要为此写一个第 2 卷)。因此，这里提到这个模式，主要是为了让路由模式的介绍尽量“完整”，并让你对工作流和过程建模有一个大致的认识。这种模式绝对不是业务过程设计的完整解决方案。

使用过程管理器会得到一种所谓的集散式消息流（如上图所示）。输入的消息将初始化过程管理器。我们把该消息称为触发消息。根据过程管理器内部的规则，它把消息（1）发送给第一个处理步骤，该处理步骤由处理单元 A 实现。当单元 A 完成处理后，它把应答消息返回给过程管理器。过程管理器确定下一个要执行的处理步骤，并把消息（2）发送给下一个处理单元。这样，所有的消息流都要经过这个中央“hub（集线器）”，因此我们把这种工作模式称为集散模式。采用中央控制元素的缺点是，过程管理器很容易成为性能瓶颈。

过程管理器的多功能既是它的最大优点，也是它的最大缺点。过程管理器能执行任何处理步骤序列，可以串行执行，也可以并行执行。因此，采用过程管理器几乎能解决所有集成问题。同样，本章介绍的大多数模式都能采用过程管理器实现。实际上，许多 EAI 提供商都力图让你相信所有集成问题都是过程问题。我们认为，任何情况下都使用过程管理器是没有必要的。它会分散我们对核心设计问题的注意力，还会带来显著的性能开销。

7.12.1 管理状态

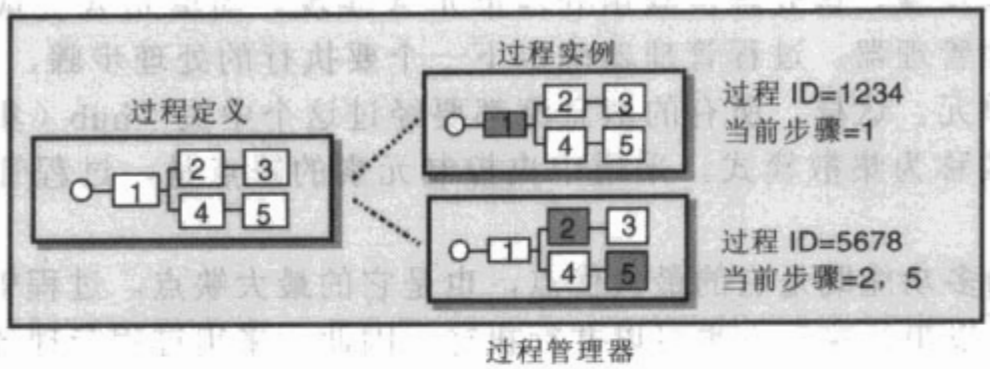
过程管理器的一个主要功能是保持消息之间的状态。例如，当第二个消息处理单元把消息返回给过程管理器时，过程管理器必须记住这是消息处理序列中的第二步。在同一个处理过程中，相同的处理单元可能会出现多次，因此我们不希望把处理的状态信息与处理单元相绑定。例如，在一个处理过程中，处理单元 B 既可能是第 2 步，也可能是第 4 步。因此，由处理单元 B 返回的消息可能触发过程管理器来根据过程上下文执行第 3 步，也可能触发其执行第 5 步。为了不增加处理单元的复杂度，同时为了实现处理状态的管理，过程管理器必须维护过程执行时的当前位置信息。

除了能存储过程执行的当前位置外，过程管理器还可以存储其他的信息，这很有用。如果中间处理结果与后续的处理步骤相关，那么过程管理器可以把前一处理步骤的中间结果保存下来。例如，如果第 1 步的结果与后续步骤相关，过程管理器可以将结果保存起来，这就不用与后续的处理单元交换这些数据，从而不会增加它们的负担。由于每个处理单元不必考虑其他处理单元产生或消费的数据，因此各个处理步骤之间可以保持独立。过程管理器有效地扮演了后面要介绍的声明标签的角色。

7.12.2 过程实例

由于过程的执行可能要经过多个步骤，因此会花较长的时间。过程管理器必须在其他过程还在执行时做好接收新的触发消息的准备。为了管理多个并发的执行过程，过程管理器要为每个到来的触发消息创建一个新的过程实例（process instance）。过程实例中存储着过程执行的相关状态信息（这个过程由触发消息初始化）。这些状态信息包括过程执行的当前步骤和所有相关的数据。每个过程实例都由唯一的过程标识符标识。

要把过程定义（也称为过程模板）与过程实例的概念区分开，这非常重要。过程定义是一种设计结构，它定义了要执行的步骤序列，类似于面向对象语言中的类。过程实例是特定过程模板的一个活动执行，相当于面向对象语言中的对象实例。下图给出了一个简单的例子，其中包括一个过程定义和两个过程实例。第一个实例（过程标识符为 1234）正在执行步骤 1，第二个实例（过程标识符为 5678）正在同时执行步骤 2 和 5。



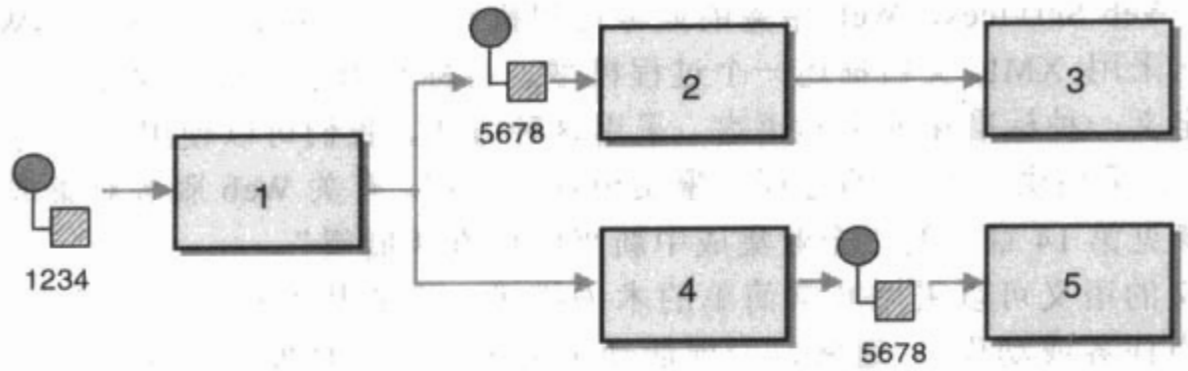
基于同一过程定义的多个过程实例

7.12.3 相关性

由于多个过程实例可以同时执行，过程管理器必须把输入的消息与正确的实例关联起来。例如，在前一个例子中，如果过程管理器从一个处理单元接收到一条消息，这个消息要给哪一个过程实例？多个实例可以执行相同的步骤，因此过程管理器无法从通道名或消息的类型得出是哪一个实例。为了把输入消息与过程实例关联起来，可以采用关联标识符的思想。通过在应答消息中存储一个唯一的标识符，使该应答消息与请求消息相关联，组件就能把到来的应答消息与最初的请求消息关联起来。采用这种标识符，即使组件发送了多个请求，并且应答消息不是按顺序返回的，组件也能把应答与正确的请求相匹配。过程管理器需要类似的机制。当过程管理器从处理单元接收到消息时，它必须能把消息与向处理单元发送消息的过程实例相关联。过程管理器必须要在发送给处理单元的消息中包括一个关联标识符。组件必须在应答消息中以关联标识符的形式返回这个标识符。如果每个过程实例都维护着一个唯一的过程标识符，那么这个过程标识符可以用作为消息的关联标识符。

7.12.4 在消息中保持状态

显然，状态管理是过程管理器的一项重要特性。那么为什么以前的模式不需要管理状态呢？在传统的管道和过滤器体系结构中，管道（如消息的通道）管理着状态。仍以前面的例子为例，如果采用消息通道把固定的组件连接起来，以实现处理过程，得到的结构如下图所示。假设这个系统与前一个例子有相同的状态（例如有两个过程实例），通道中有一个标识符为 1234 的消息在等待组件 1 的处理，而通道中标识符为 5678 的两条消息分别等待组件 2 和 5 的处理。一旦组件 1 消费了消息并完成其处理任务，它将向组件 2 和 4 广播一条新消息，这与前一例子中的过程管理器的行为一样。



在通道中保持状态

这个例子中使用的消息流符号非常类似于建立过程管理器组件行为模型时使用的UML活动图。可以在设计阶段使用一种抽象的符号建立系统的行为模型，然后确定把这种行为实现为分布式的管道和过滤器体系结构，还是使用一个中央过程管理器实现为一种集散式体系结构。尽管出于篇幅的考虑，本书没有过多地深入讨论过程模型的设计，但是设计过程模型时，这方面的许多模式确实很适用。

与制定大多数的结构设计决策一样，不可能简单地做出决定：究竟是实现中央过程管理器还是分布式的管道和过滤器体系结构。很多情况下，最好能使用多个过程管理器组件，每个组件负责管理一个更大处理过程的一部分。过程管理器组件之间再通过管道和过滤器体系结构实现相互通信。

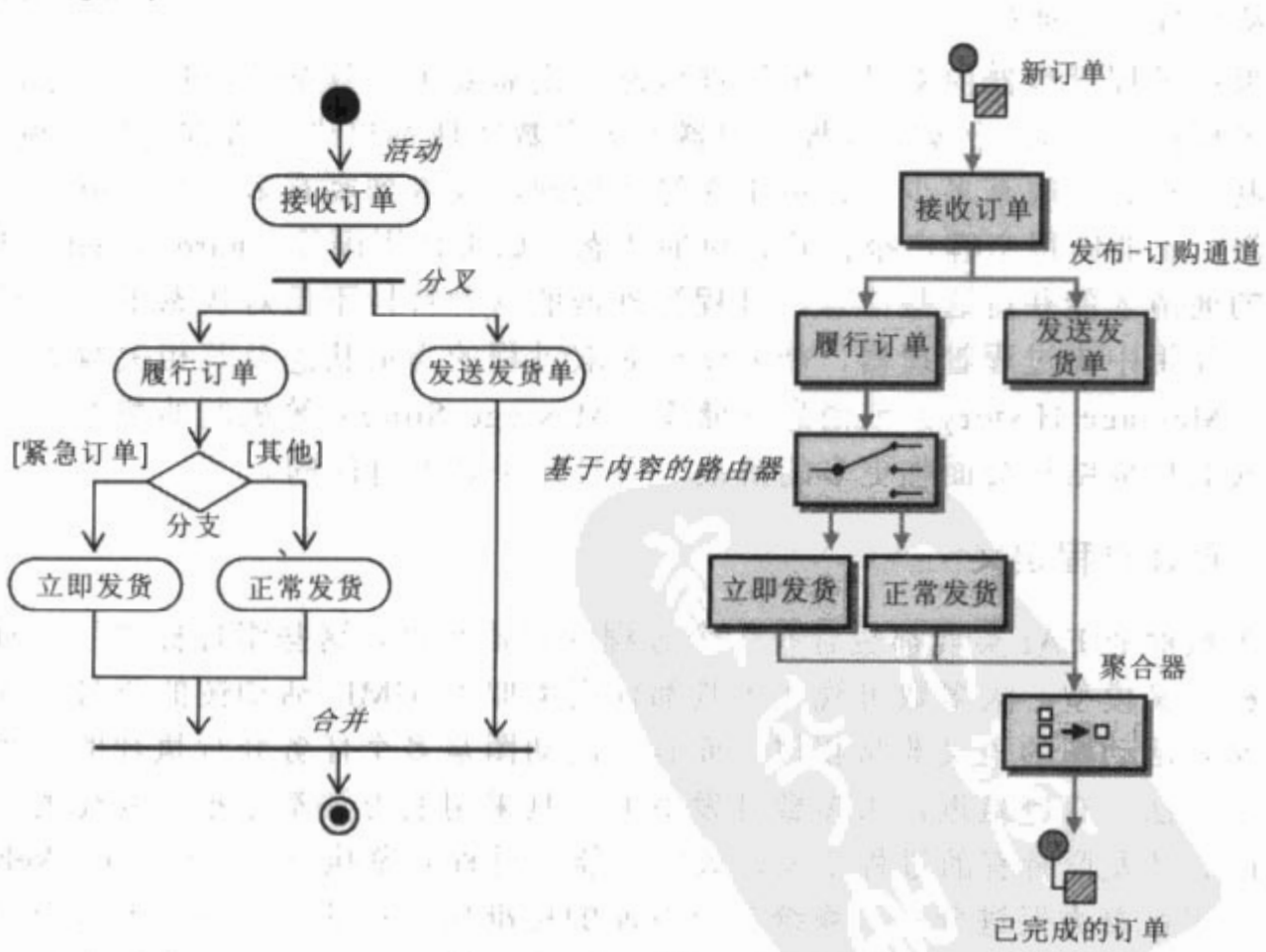
如果在过程管理器内部显式地管理状态，则需要更加复杂的组件，但是由此能得到更强大的过程报告功能。例如，过程管理器的大多数实现都提供了查询过程实例状态的能力。这就有利于查看到底有多少个订单正在等待处理，或者到底有多少个订单已经因为缺少库存而取消。还能向每个客户报告其订单的状态。如果使用硬线（hard-wired）通道，只有检查所有的通道才能获得这些信息。过程管理器的这种特性不但对状态报告很重要，也有利于调试。使用中央过程管理器，能方便地获取过程的当前状态及其相关数据。如果没有消息历史（Message History）或消息存储库（Message Store）等机制的帮助，要调试一个完全分布式的集成结构会面临更多的挑战，而且几乎是不可能的。

7.12.5 创建过程定义

大多数商业EAI实现都包含有一个过程管理器组件，这些组件提供了可视化工具，能建立过程定义模型。大多数可视化工具都使用类似于UML活动图的符号，因为过程管理器的语义与活动图的语义非常相似。而且，活动图是多个任务并行执行的一种非常好的可视化表示方法。直到最近，大多数开发商的工具采用的方法都是把可视化的符号转换为一种内部的、开发商特有的过程定义，该定义能由过程引擎执行。不过，在Web服务这把大伞之下，正在着力促进分布式系统各个方面的标准化，在这个进程中并没有忽视过程定义的重要角色。经过努力，为此提出了三种“语言”：Microsoft定义了XLANG，它由BizTalk协作建模工具组提供支持；IBM设计了WSFL（Web Services Flow Language），即Web服务流语言[WSFL]；最近，这两家公司联合建立了BPEL4WS（Business Process Execution

Language for Web Services，Web 服务的业务过程执行语言）的规范。BPEL4WS 是一种强大的语言，它采用 XML 文档描述一个过程模型。目标是为了在过程建模工具和过程管理器引擎之间定义一种标准化的中间语言。采用这种方式，我们可以使用供应商 X 的产品建立过程模型，并采用供应商 Y 的过程引擎实现这一过程。有关 Web 服务对集成标准化产生的影响，请参见第 14 章中的“企业集成中新兴的标准和展望”。

过程定义的语义可以采用非常简单的术语描述。它的基本构建块是活动（Activity，有时也被称为任务或动作）。通常，一项活动可向另一个组件发送一条消息，等待输入消息，或在内部执行一个特殊的功能（如消息转换器）。活动可以按串行方式连接，或者多个活动可以使用分叉和合并结构并行执行。分叉使多个活动能同时执行。它在语义上等同于固定的管道和过滤器体系结构中的发布-订购通道。合并把多个并行执行的线程同步为一个线程。只有所有并行的线程都已完成各自的活动后，才能继续执行合并之后的活动。在管道和过滤器体系结构中，聚合器常常扮演了这样的角色。过程模板还应该提供分支或判定点，从而可以根据消息字段的内容改变执行路径。这个功能等价于基于内容的路由器提供的功能。许多建模工具都提供了设计循环结构的能力，但这实际上是一种特殊的分支结构。下图反映出过程定义（采用 UML 活动图描述）与管道和过滤器实现之间的语义相似性，尽管它们的物理实现有很大的区别，其中管道和过滤器的实现使用了我们已经定义的模式。



UML 活动图与相应的管道和过滤器实现的示例

7.12.6 比较过程管理器与其他模式

我们已经对基本的管道和过滤器体系结构、路由表和过程管理器多次做过比较。在下表中，我们列出了这些模式之间的关键区别，从而指出选择正确的体系结构时要做哪些折衷考虑。

分布式管道和过滤器	路由表	中央过程管理器
支持复杂的消息流	只支持简单的、线性消息流	支持复杂的消息流
难以改变消息流	容易改变消息流	容易改变消息流
不存在集中的故障点	存在潜在的故障点（计算路由表）	存在潜在的故障点
高效的分布式运行时结构	大部分是分布式的	集中发散式结构可能导致瓶颈
不存在集中的管理和报告点	存在集中的管理点，不存在集中的报告点	存在集中的管理和报告点

如果存在控制和状态管理的集中点，这也意味着它可能是故障的集中点或性能的瓶颈。正因为如此，大多数过程管理器的实现会把过程实例的状态持久地存储在文件或数据库中，从而利用企业级数据库系统特别提供的冗余数据存储机制。并行地执行多个过程管理器实例也是很常见的情形。并行执行的过程管理器很容易实现，因为过程实例彼此是相互独立的。这样就可以把过程实例分布到多个过程引擎中。如果过程引擎把所有状态信息保存在一个共享的数据库中，那么即使某个过程引擎出现故障，其他的过程引擎可能接替失败的引擎继续执行，从而使系统具有很高的可靠性。采用这种方法的缺点是，在每一个处理步骤完成后，各个过程实例的状态都必须保存在一个中央数据库中，这样数据库很容易成为新的性能瓶颈。由于这些情形经常出现，集成体系结构必须在性能、可靠性、成本和可维护性之间找到合适的平衡点。

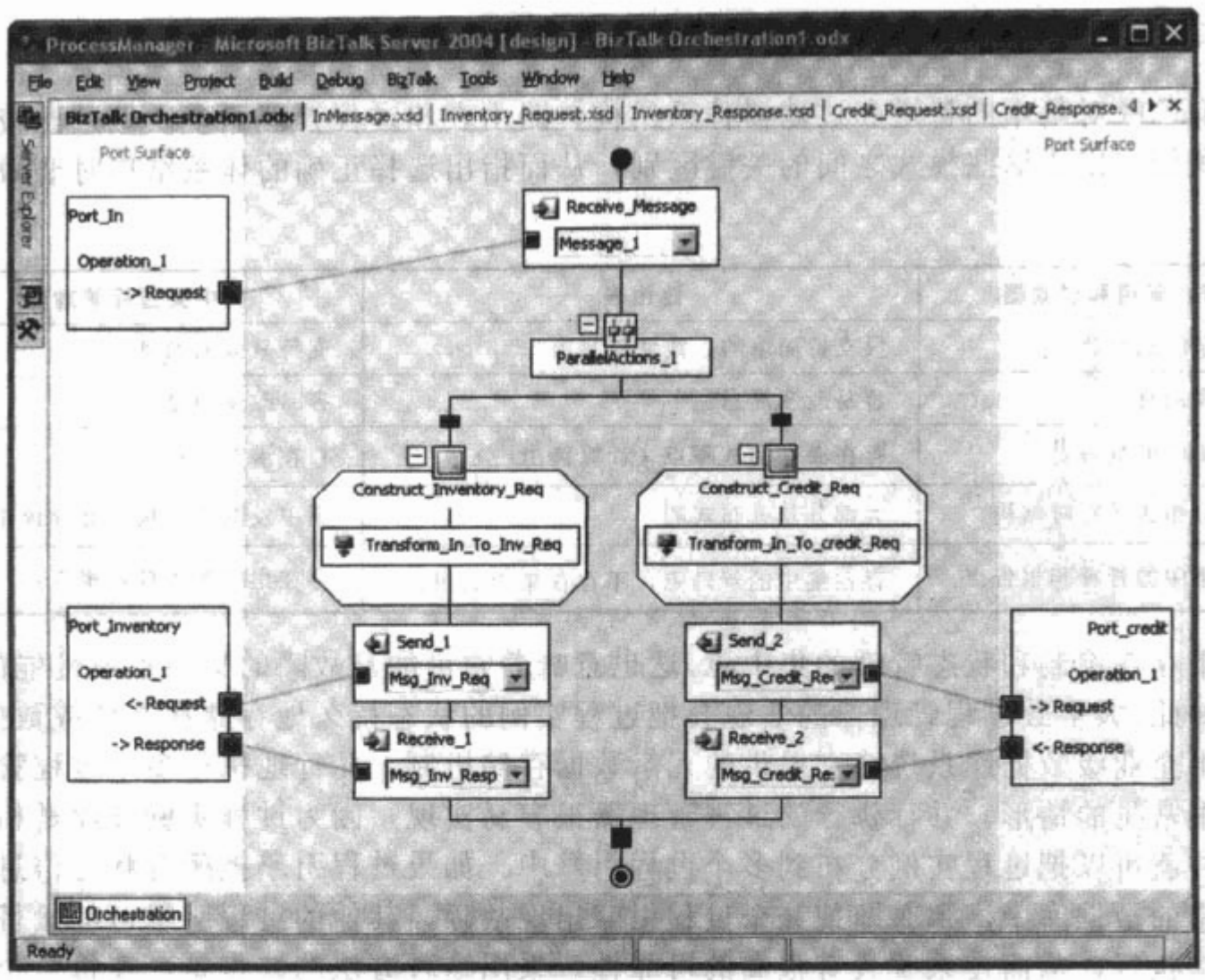
例：贷款中介

贷款中介的 MSMQ 实现示例（见第 9 章中“使用 MSMQ 的异步实现方案”）实现了一个简单的过程管理器。通过为过程管理器和过程实例从头编写 C#类，从而创建了过程管理器的功能。这个例子还有一个 TIBCO 实现（见第 9 章的“使用 TIBCO ActiveEnterprise 的异步实现方案”），其中使用了一种商业的过程管理工具。

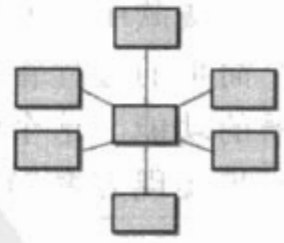
例：Microsoft BizTalk 协作管理器

大多数商业 EAI 工具都提供了过程设计和执行的功能。例如，Microsoft BizTalk 允许用户使用协作设计器（Orchestration Designer）工具来设计过程定义，这个工具集成在 Visual Studio .NET 编程环境中。

这个简单的协作示例接收一个订单消息，并执行两个并行的活动。其中一个活动创建一个到库存系统的请求消息，另一个活动创建一个到信用系统的请求消息。一旦接收到两个系统的应答消息，过程将继续执行。利用可视化的符号表示，可以更容易地遵循过程定义。



7.13 消息代理

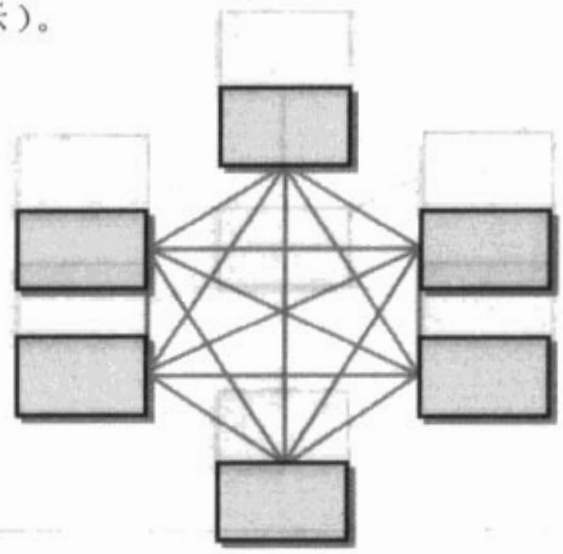


本章介绍了许多模式，即使产生消息的应用并不知道消息的最终目标是谁，也可以把消息路由给正确的目标。大多数模式关注的只是特定类型的路由逻辑。不过，通过组合，这些模式还能解决更大的问题。

如何让消息的目标与消息的发送者解耦合，并保持对消息流的集中控制?.

通过使用简单的消息通道，可以在接收者与发送者之间形成了一个间接层——发送者只知道通道，而不知道接收者。但是，如果每个接收者都有自己的通道，那么这个间接层就没有什么意义。发送者不用知道接收者的地址，而必须知道与接收者相关的正确的通道名。

除了最普通的消息传递解决方案以外，其他解决方案都要连接许多不同的应用。如果我们要为任何两个应用创建单独的连接通道，那么系统中的通道就会多得不可管理，使集成变成一团乱麻（如下图所示）。



由于点对点连接产生的集成混乱

这个图说明，在每个应用之间建立直接的通道将导致通道数量的爆炸，并丧失使用通道完成消息路由所带来的许多优点。之所以会产生这种类型的集成体系结构，往往是一种解决方案随时间不断发展造成的。首先，客户关怀系统必须与财会系统交流信息。然后，客户关怀系统还要从库存系统那里获得信息，发货系统要更新财会系统中的运费信息。显然，“多增加一个组件”将很快影响整个集成解决方案的实用性。

如果一个应用必须与其他所有应用进行显式的通信，就很容易妨碍系统的可维护性。例如，如果客户关怀系统中的客户地址发生了改变，那么该系统必须向所有保存着客户地址副本的系统发送消息。每增加一个新的系统，客户关怀系统就必须知道该系统是否使用了这个地址信息，以及是否需要改变该地址信息。

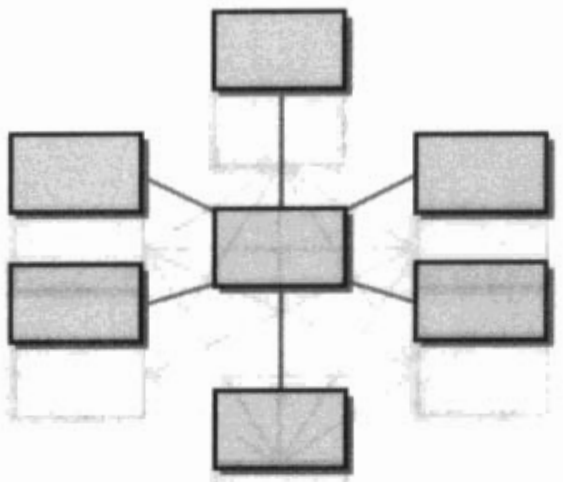
发布-订购通道提供了一种基本的路由：消息可以路由给订购了特定通道的应用。这种通道能实现简单的消息广播，但是实际的路由规则往往要复杂得多。例如，必须根据订单的大小或订单的属性把到来的订单消息路由给不同的系统。为了避免由应用负责决定消息的最终目标，中间件应该包含消息路由器，把消息路由给对应的目标。

各个消息路由模式有助于我们实现发送者与接收者的解耦合。例如，采用接收表，发送者不必了解接收者的信息，而把这些信息纳入中间件层。把路由逻辑转移到中间件层有两个好处。第一，许多商业中间件和 EAI 套件提供了专门用于完成这些任务的工具和开发库。由于我们不必编写与消息端点相关的代码，如事件驱动的消费者或线程管理的有关代码，从而减少了编写代码工作量。第二，与在应用内部实现路由逻辑相比，在中间件层的内部实现路由逻辑能使路由“更智能”。例如，当新的系统添加到集成解决方案中时，使用

动态接收表可以避免修改代码。

但是，如果使用了大量独立的消息路由器组件，这几乎与我们力图解决的集成混乱同样难以管理。

使用一个集中的消息代理从多个目标接收消息，确定正确的目标，并把消息路由给正确的通道。消息代理的内部实现使用了其他消息路由器。

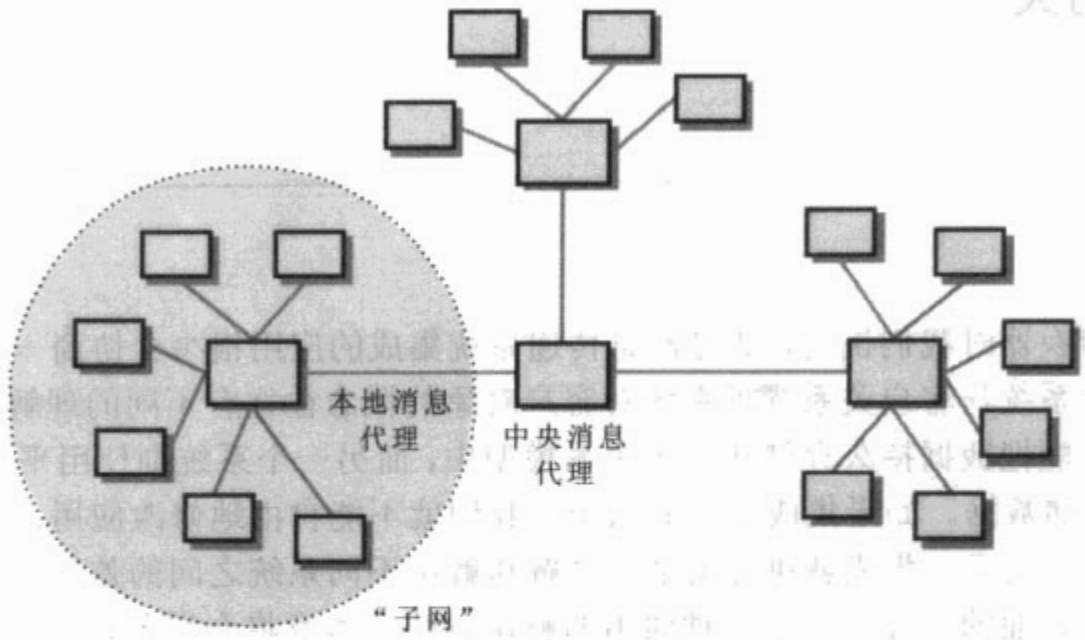


使用集中的消息代理有时也称为集散式的结构模式，从上图可见，这个名字很形象。消息代理模式的应用范围与本章介绍的其他大部分模式的应用范围有所不同。它并不是一个独立的设计模式，而是一个体系结构模式。因此，类似于管道和过滤器模式，消息代理模式提供了把组件链接起来的基本方法，以形成更复杂的消息流。该模式不只是把单独的组件链接起来，它本身关注的是更大的解决方案，可以帮助我们解决系统管理必然遇到的复杂性。

消息代理并不是一个单一的组件，它的内部使用了本章介绍的许多消息路由模式。因此，如果你决定使用消息代理作为结构模式，可以选择正确的消息路由器设计模式来实现消息代理。

消息代理提供的集中维护是一个优点，但它也会变成缺点。所有消息都通过一个消息代理进行路由，这会使消息代理成为严重的性能瓶颈。为了解决这个问题，可以采用许多技术。例如，消息代理模式只告诉我们可以开发一个实体来完成路由，但是并没有限制在系统部署阶段可以运行这个实体的多少个实例。如果消息代理的设计是无状态的（如果它只是由无状态的组件组成），就可以方便地部署多个代理实例，以提高吞吐量。点对点通道的特性可以确保，对于任何到来的消息，只能由一个消息代理实例消费。而且，和大多数实际的应用情况一样，最终的解决方案是由许多模式组合而成的。在许多复杂的集成解决方案中，完全可以设计多个消息代理组件，每个代理组件负责处理其中一部分问题。这就可以避免创建一个由于过于复杂而变得不可维护的万能的消息代理。采用这种方法有一个明显的缺点，我们不再有一个单一的维护点，而且可能会形成一种新的混乱局面，这一次是由消息代理导致的。要实现组合的消息代理，一种好的结构是使用消息代理的层次结构

（如下图所示）。这种配置结构类似于由许多子网组成的网络配置结构。如果消息只在子网内部的两个应用间传送，那么本地的消息代理就能管理消息的路由。如果消息的目标在另一个子网中，那么本地的消息代理可将消息传递给中央消息代理，由中央消息代理确定消息的最终目标。中央消息代理完成的功能与本地消息代理完成的功能一样，但是它不是将各个应用解耦合，而是实现子系统的解耦合，这些子系统分别由多个应用组成。



层次结构的消息代理在提供解耦合的同时避免了万能代理

由于消息代理的目标是降低每个应用之间的耦合，因此它一般要处理应用之间的消息数据格式的转换。消息代理可以抽象出消息的路由，但是如果发送应用必须按目标的消息格式（通常隐藏）来完成消息的格式化，那么消息代理对于发送应用没有什么帮助。下一章将介绍一系列能解决这些问题的消息转换模式。在很多情况下，消息代理会在内部使用规范数据模型，以避免 N 平方问题（系统中为了在每个接收者之间实现消息转换所需的转换器数量等于参与者数量的平方）。

例：商业 EAI 工具

大多数商业 EAI 工具都提供了一些工具，可以简化为集成解决方案创建消息代理。这些工具通常提供了许多支持开发和部署消息代理的特性：

1. **内建的端点代码**：大多数 EAI 套件都包含向消息总线发送消息和从消息总线接收消息的全部代码。开发者不必编写与传输相关的任何代码。

2. **可视化的设计工具**：这些工具可以帮助开发人员采用可视化的组件（如路由器、判定点和转换器）来构造消息代理的功能。这些工具使得消息流很直观，其中许多组件的代码编写只需要简单的几行，如一个评价函数或一条规则。

3. **运行时支持**：大多数 EAI 开发包还提供了复杂的运行时支持，有助于部署解决方案，以及监控流经消息代理的通信量。

第 8 章

消息转换

8.1 引言

介绍消息转换器时我们说过，通过消息传递系统集成的应用很少会协商一个公共的数据格式。例如，财会系统与客户关系管理系统对客户对象的概念往往有不同的理解。基于这一点，其中一个系统可能把数据持久存储在一个关系模型中，而另一个系统则使用平面文件（flat file）或 XML 文档存储数据。如果集成已有的系统，我们就不能自由地修改应用，以使之能更好地与其他系统协作。而且，集成解决方案必须调解和解决不同系统之间的差异。消息转换器模式为解决数据格式方面的差异提供了一种通用的解决方法。本章将介绍各种不同的消息转换器。

大多数消息传递系统对消息首部的格式和内容提出了特定的要求。我们把消息的有效数据封装在一个信封包装器（Envelope Wrapper）中，使之能满足消息传递基础设施的需求。如果消息要通过不同的消息传递基础设施传递，可以把多个信封包装器结合起来使用。

如果消息源系统无法提供目标系统所需的数据字段，则要采用内容扩充器（Content Enricher）。它能查找到缺少的信息，或从已有的数据中计算出这些信息。内容过滤器（Content Filter）的作用正好相反，它把不需要的信息中消息中删除。声明标签（Claim Check）也从消息中删除信息，但是要把这些信息保存起来以备后用。规范器（Normalizer）把具有不同格式的输入消息转换为公共的格式。

8.1.1 消除依赖

消息转换是集成中的一个深层次话题。消息通道和消息路由器可以消除应用之间基本的依赖，从而使一个应用不必知道其他应用的位置。

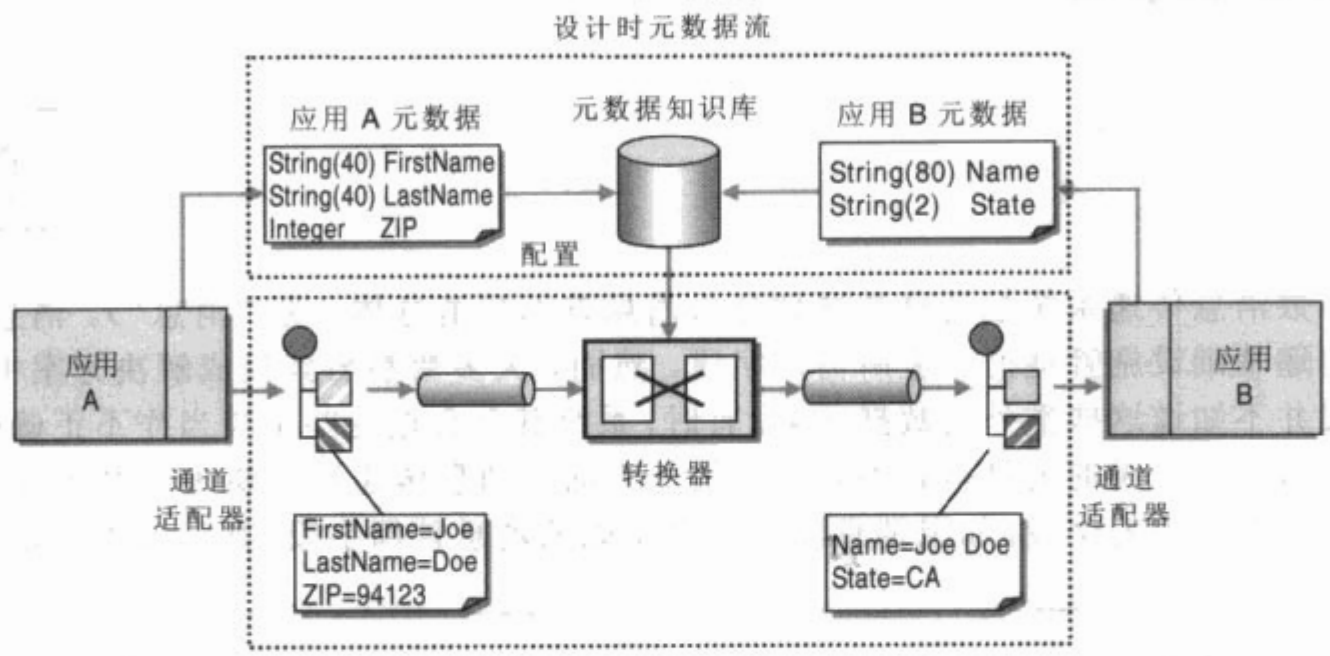
一个应用可以向消息通道发送一个消息，而不考虑哪个应用会消费这个消息。但是，消息的格式带来另外一些依赖性。如果一个应用必须根据其他应用的数据格式来格式化自己的消息，那么采用消息通道的形式实现解耦合还不够。如果对接收方应用有所修改，或者把接收方应用切换为另一个应用，仍然需要相应地修改发送方应用。使用消息转换器有利于消除这种依赖性。

8.1.2 元数据管理

要把消息从一种格式转换为另一种格式，则需要解决好元数据问题，所谓元数据是指

用于描述实际数据格式的数据。如果从一个应用向另一个应用发送了一个消息，告诉我们 ID 号为 123 的客户从加利福尼亚的 San Francisco 搬到了北卡罗来纳州的 Raleigh，相关的元数据则是：该地址修改消息使用了一个数字类型的客户 ID 字段，并且把客户的姓和名分别存储在两个不超过 40 字符的文本字段中。

元数据在集成中扮演了重要的角色，我们可以把大多数集成解决方案看作是两个平行系统之间的相互影响。其中一个系统处理实际的消息数据，另一个系统处理元数据。用于设计消息数据流的许多模式也可以用于管理元数据流。例如，通道适配器不但能从系统中移走和增加数据，还能从外部应用中提取元数据，并把它加载到一个集中的元数据知识库中。使用这个知识库，集成开发人员可以定义应用元数据与规范数据模型之间的转换。



元数据集成

例如，如果有两个需要交换客户信息的系统，上图展示了这两个系统之间的集成方案。每个系统对客户数据的定义都稍有不同。应用 A 把姓和名存储在两个不同的字段中，而应用 B 把它们存储在一个字段中。另外，应用 A 存储了客户的邮编代码，而没有存储州名；应用 B 只存储了州的缩写。从应用 A 流向应用 B 的消息必须经过转换，才能让应用 B 接收到所需格式的数据。如果通道适配器也能取得元数据（描述消息格式的数据），那么实现转换就会更简单。元数据可以加载到一个知识库中，进而能极大地简化消息转换器的配置和验证。元数据可以采用各种格式存储。XML 消息使用的通用格式是 XSD，即可扩展的模式定义（eXtensible Schema Definition）。其他 EAI 工具可能实现了专门的元数据格式，不过也支持管理员从其他格式导入元数据，或把元数据导出为不同的格式。

8.1.3 消息传递之外的数据转换

这些转换模式中运用的许多原理不仅仅适用于基于消息的集成系统。例如，文件传输必须在系统之间实现转换功能。同样地，对于远程过程调用，即使应用内部使用的是另一

种格式，也必须采用被调服务指定的数据格式发出请求。这就要求调用应用必须完成数据的转换。在ETL(提取、转换、加载)工具中还包含了一些非常复杂的转换引擎，如Informatica或DataMirror。这些工具可以一次性地转换大量数据集，而不只是转换单独的消息。

本章的重点是介绍各种基本的消息转换器模式。不涉及实体之间结构的转换（例如如何实现两个数据模型之间的转换，其中一个数据模型支持客户与地址之间的多对多关系，而另一个数据模型把地址字段包含在客户记录中）。最早介绍数据与关系表示的书是[Kent]，这也是这方面最合适的书。

8.2 信封包装器



大多数消息传递系统把消息数据划分为消息首部和消息体（见“消息”）。消息首部包含消息传递基础设施管理消息流所需的字段。然而，大多数参与到集成解决方案中的端点系统一般并不知道这些额外的数据元素。有时，系统甚至会把这些字段当作不正确的数据，因为它们与应用所用的消息格式不匹配。另一方面，消息传递组件要在应用间路由消息，则需要消息首部中的字段，如果消息没有包含正确的消息首部字段，则认为消息是非法的。

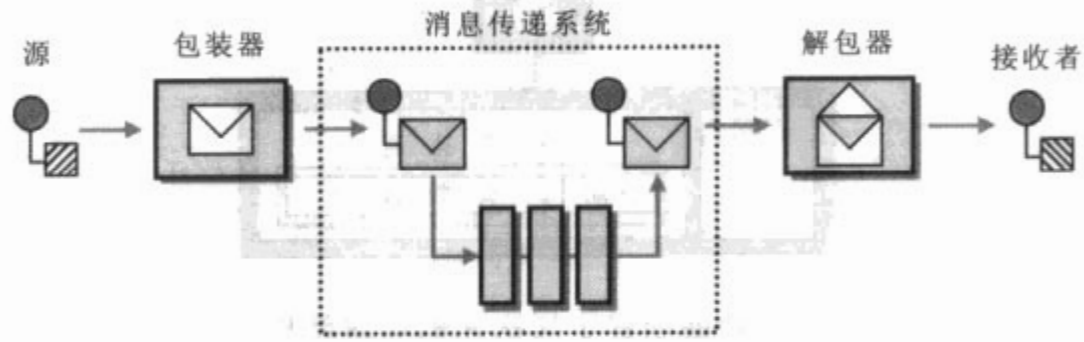
> 如果已有的系统对消息格式有特定的需求，如要求有消息首部字段，或者要求加密，这样的系统如何参与消息传递交换？

例如，假设消息传递系统采用了一个专用的安全方案。合法的消息必须包含能被其他消息传递组件接受并处理的安全凭证。采用这种方案，有利于防止未授权的用户把数据添加到系统中。而且，消息内容经过加密，可以防止未授权的监听者窃听信息，这对保护发布-订购机制的安全特别重要。但是，已有的应用通过消息传递系统集成时，大多不了解用户身份或消息加密的概念。因此，必须把“原始”消息转换为符合消息传递系统规则的消息。

一些大型的企业使用不只一个消息传递基础设施。因此，消息要通过消息传递桥跨越多个消息传递系统。每个消息传递系统可能对消息体和消息首部的格式提出了不同的要求。基于TCP/IP的一些网络协议就是一个例子。在很多情况下，与其他系统连接时只能采用专门的协议，如Telnet或安全Shell（SSH）协议。为了使用其他协议（如FTP）进行通信，原来的协议格式必须封装到符合所支持协议的数据包中。当数据包到达另一端时，再提取出有效的内容。这个过程被称为隧道化（tunneling）。

当一个消息格式封装到另一个消息格式中时，系统将无法访问数据有效载荷（payload）中的信息。大多数消息传递系统只允许组件（如消息路由器）访问定义在消息首部中的数据字段。如果一个消息封装在另一个消息的数据字段中，组件就不能使用原始消息中的字段实现路由或转换功能。因此，一些数据字段必须从原始消息中提取出来，把它们添加到新消息格式的消息首部中。

使用信封包装器，把应用数据包装在符合消息传递基础设施要求的信封中。当信封到达目标时再解开消息。



包装和解包消息的过程包括 5 步：

1. 消息源采用原始格式发布一个消息。这个格式是由应用本身的性质所决定，不符合消息传递基础设施的要求。

2. 包装器把原始消息转换为符合消息传递系统要求的消息格式。这一过程可能包括增加消息首部字段，加密消息，增加安全凭证等。

3. 消息传递系统传送符合要求的消息。

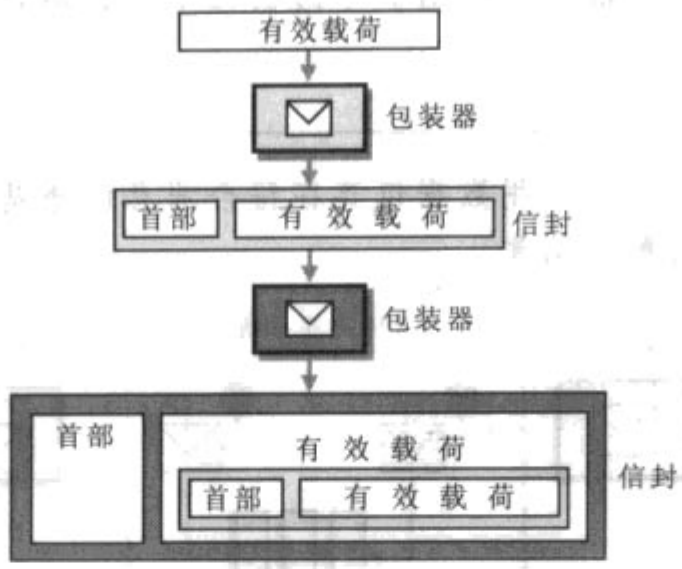
4. 结果消息传输给解包器。解包器逆向撤销包装器所作的修改。这一过程可能包括删除消息首部字段，解密消息，或验证消息凭证。

5. 消息接收者接收到一个“明文”消息。

信封既要封装消息首部，也要封装消息体（或有效载荷）。我们可以把消息首部看作是写在信件信封外面的信息：消息传递系统使用这些信息来路由和跟踪消息。信件的内容是有效载荷或消息体——在消息到达目标前，消息传递基础设施（在一定的限制条件下）不太关注消息体。

包装器一般要在原始消息中添加信息。例如，邮政系统在发送内部消息前，必须查看其邮政编码。为此，包装器集成了内容扩充器的部分功能。但是，包装器不会扩充实际的信息内容，而只是增加路由、跟踪和处理消息所需的信息。这些信息可以实时地创建（如创建唯一的消息 ID，或者增加时间戳），也可以从基础设施中提取（如从安全上下文中提取），或者是那些原本包含在原始消息体中但被包装器划分到消息首部中的数据（如原始消息中包含的键字段）。最后这种情况有时也被称为提升（promotion），因为特定的字段从隐藏的消息体中“提升”到完全可见的消息首部中。

通常情况下，多个包装器和解包器会链接在一起（见下面的邮递系统示例），在此利用了分层的协议模型。由此，消息的有效载荷中可能包含着新的信封，这个信封则进一步封装了消息首部和有效载荷（如下图所示）。



包装器链创建了层次化的信封结构

例：SOAP 消息格式

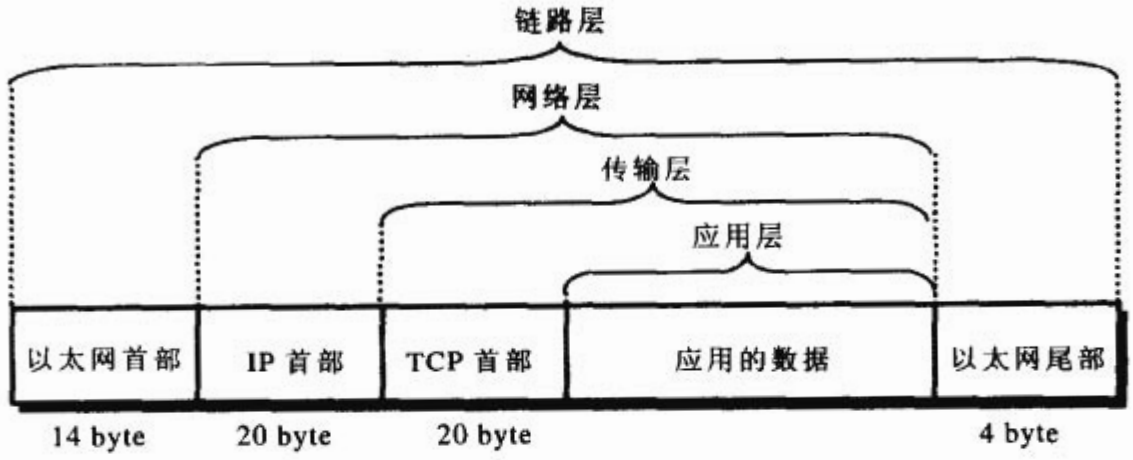
基本的 SOAP 消息格式[SOAP1.1]相当简单。它规定了信封格式，包含一个消息首部和一个消息体。下面的例子说明了如何在消息体中包含另一个信封，该信封进一步包含其他消息首部和消息体。这个组合的消息发送给一个中间组件，它解开外面的消息，并转发里面的消息。跨信任边界时，这种由中间组件组成的通信链非常普遍。可以对所有消息编码，并将其包装到另一个消息中，使中间组件无法查看到消息的内容或首部（例如消息的地址可能是秘密的）。接收者可以解出消息，对有效载荷解码，并通过可信的环境传递未编码的消息。

```
<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
    <env:Header env:actor="http://example.org/xmlsec/Bob">
        <n:forward xmlns:n="http://example.org/xmlsec/forwarding">
            <n:window>120</n:window>
        </n:forward>
    </env:Header>
    <env:Body>
        <env:Envelopexmlns:env="http://www.w3.org/2001/06/soap-envelope">
            <env:Header env:actor="http://example.org/xmlsec/Alice"/>
            <env:Body>
                <secret xmlns="http://example.org/xmlsec/message">
      The black squirrel rises at dawn</secret>
            </env:Body>
        </env:Envelope>
    </env:Body>
</env:Envelope>
```

例：TCP/IP

虽然我们通常把 TCP/IP 作为一个术语称呼，但它实际上是由两个协议组成的。IP 协议提供了基本的寻址和路由服务，而 TCP 协议提供了可靠的、面向连接的服务，它是建立在 IP 协议上的。按照 OSI 层次模型，TCP 属于传输层协议，IP 属于网络层协议。TCP/IP 数据一般是通过以太网传送的，以太网协议在链路层实现。

因此，应用的数据首先包装到一个 TCP 信封中，然后再包装到一个 IP 信封中，最后再包装到一个以太网信封中。由于网络是面向流的，因此信封包含了首部和尾部，它们标出了数据流的起始和结束位置。下图给出了通过以太网传递的应用数据的结构。

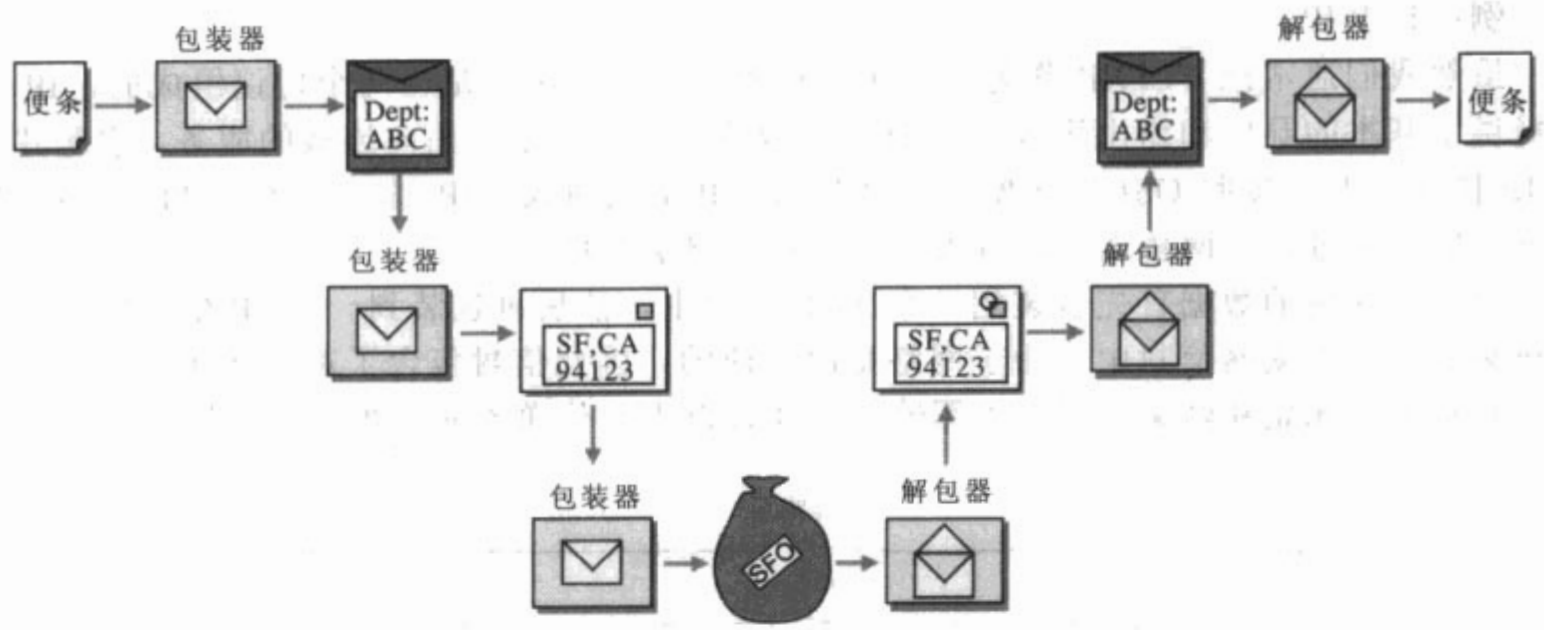


应用的数据被封装在多个信封中，以便通过网络传送

从图中可以看到，应用的数据被连续包装到多个信封中：TCP（传输层信封）、IP（网络层信封）和以太网（链路层信封）。TCP 和 IP 信封只包含一个首部，而以太网信封包含了首部和尾部。如果你想了解更多有关 TCP/IP 的知识，[Stevens]肯定能满足你迫切的要求。

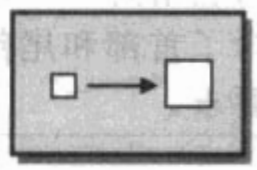
例：邮递系统

信封包装器模式类似于邮递系统（如下图所示）。假设一个员工想写一个便条交给另一个员工。消息的有效载荷可以书写在任意的一张纸上。为了传递该便条，它必须“包装”到一个公司专用的信封中，其上注明了接收者的姓名和部门代码。如果接收者在另一个公司工作，那么原本公司内部传递的消息将被装入另一个更大的信封，并通过美国的邮递服务系统邮寄。为了让新的消息符合 USPS（美国邮递服务系统）的要求，它必须采用一种新的信封，要求有邮政编码，并付有邮资。USPS 可以通过航空方式传送这封信。为此，它把属于同一地区的所有信件装入一个邮包，并贴上标有目的机场代码的条码（由三个字母组成）。邮包一旦抵达目的机场，将按包装的反过程解包，直到把原来的便条交给对方。这个例子说明了隧道化的含义：通过航空货运“隧道”传送的邮件就像在 TCP/IP 连接上建立的 UDP 组播包隧道，使 UDP 包能到达不同的 WAN（广域网）段。



邮递系统的例子说明了使用管道和过滤器体系结构把包装器和解包器链接起来的常用方法。消息可以多次包装，并按对称的顺序解包。设计管道和过滤器体系结构时，要保持每个操作步骤彼此独立，这有利于消息传递基础设施灵活地增加或删除包装和解包操作。例如，由于所有的通信都是通过 VPN 而不是通过公共的 Internet 路由，就不再需要加密操作了。

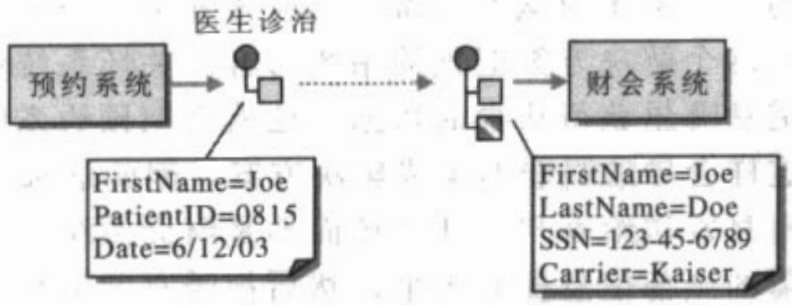
8.3 内容扩充器



从一个系统向另一个系统发送消息时，目标系统需要的信息往往比源系统提供的信息多。例如，输入的地址消息可能只包含邮政编码，这是因为设计者认为没有必要存储多余的城市和州的信息。而另一个系统可能除了需要邮政编码字段外，还需要城市和州的信息。甚至有的系统不使用简写的州名，而要求把州名的全称写出来，因为它要使用任意形式的地址，以便支持国际地址。同样地，某个系统可能提供客户的 ID，而接收系统需要的是客户的名字和地址。再看另一种情况，由订单管理系统发送的订单消息只包含订单编号，但是我们需要找出与该订单相关的客户 ID 才能把订单传递给客户管理系统。这种情况是很多的。

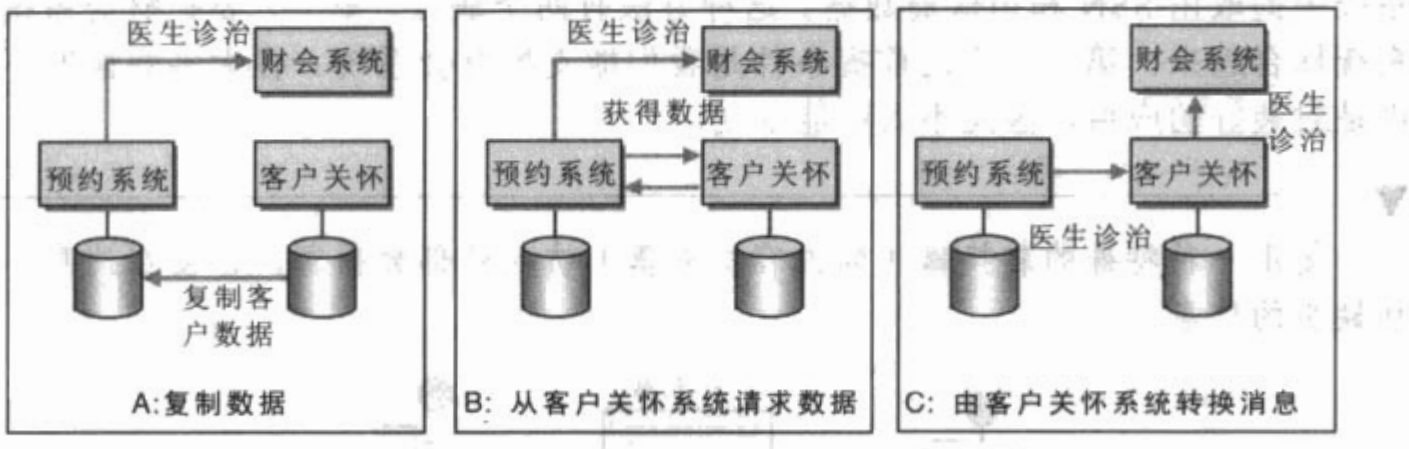
如果消息的创建者无法提供所需的所有数据项，如何与其他系统通信？

这个问题是消息转换器的一种特例，因此消息转换器的一些方法对此也同样适用。但是，这个问题又与我们在“消息转换器”中介绍的基本示例有所区别。消息转换器认为接收应用所需的数据已经包含在输入消息中，只是格式不对。而在这个新例子中，不仅是一个重排字段的问题，还需要在消息中加入另外的信息。



财会系统需要的信息比预约系统能提供的信息多

我们首先考虑下面的例子（如下图所示）。医院预约系统发布了一个消息，宣布病患已经经过医生的诊治。该消息包含病患的名字，他（或她）的病患 ID，以及看病的日期。为了让财会系统记录病患的看病信息，并通知保险公司，它需要病患的全名、保险担保商的名称和病患的社会安全号。但是，预约系统中没有存储这些信息，它们是由客户关怀系统提供的。那么我们应该怎么做呢？



解决扩充器问题的可能方案

选择 A：我们可以修改预约系统，使之能存储额外的信息。当客户关怀系统中的客户信息发生改变时（例如病患调整了保险担保商），必须把改变后的信息复制给预约系统。这样，预约系统就能发送包含所需全部信息的消息。遗憾的是，采用这种方法有两个明显的缺点。第一，它需要修改预约系统的内部结构。在大多数情况下，预约系统是封装好的应用，因此不允许这样的修改。第二，即使预约系统是可定制的，也要认识到，我们对系统所做的修改建立在另一个系统的特定需求上。例如，如果我们还需要给病患发一封信来确认诊治的过程，就要再次修改预约系统，使其包含客户的邮件地址。如果能把预约系统与负责处理医生诊治消息的特定应用解耦合，集成解决方案的可维护性就会更高。

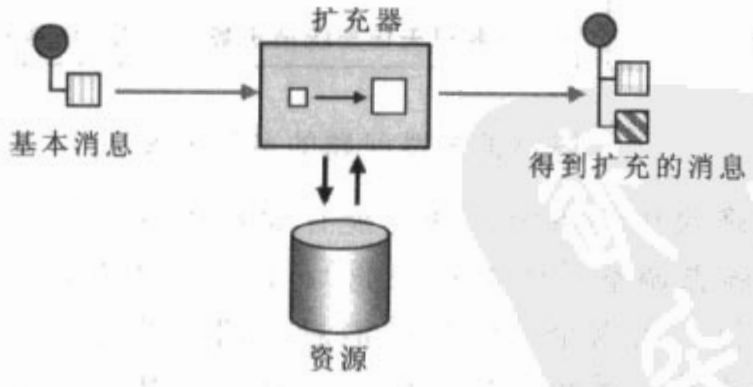
选择 B：不是在预约系统中存储客户的信息，而是让预约系统从客户关怀系统那里请

求所需的 SSN（社会安全号）和担保商数据，然后再发送医生诊治消息。这样能解决第一个问题，即不必修改预约系统的存储结构，但是，第二个问题仍然无法解决：预约系统需要知道，只有得到了 SSN 和担保商数据才能通知财会系统。因此，该消息的语义更类似于通知保险（Notify Insurance）而不是医生诊治（Doctor Visit）。在一个松耦合系统中，我们不希望让一个系统指示另一个系统应该怎么做，而应该发送一个事件消息，让其他系统自己决定如何操作。另外，这个解决方案把预约系统与客户关怀系统更紧密地耦合起来，因为预约系统现在需要知道从哪里获得缺少的数据。这就会把预约系统与财会系统和客户关怀系统绑在一起。由于这样会导致脆弱的集成解决方案，因此这种耦合是不可取的。

选择 C：如果先把消息发送给客户关怀系统而不是财会系统，就能减少一些依赖性。客户关怀系统可以把所需的全部信息提取出来，然后把带有所需数据的消息发送给财会系统。这样可以让预约系统与后续的消息流很好地解耦合。但是，目前实现的业务规则是，病患得到客户关怀系统中医生的诊治后，保险公司就会接收到账单。这就要求修改客户关怀系统中的逻辑。如果客户关怀系统是已经封装好的应用，就很难实现这样的修改，甚至不可能实现。即使能做这样的修改，这样的业务规则也会要求客户关怀系统间接地发送账单消息。如果客户关怀系统能提供财会系统所需的全部数据项，那就没有什么问题。但是，如果要从其他系统提取某些数据项，又会遭遇最开始碰到的情况。

选择 D（图中没有显示）：我们还可以修改财会系统，让它只请求客户 ID，并从客户关怀系统中提取出 SSN 和担保商数据。这种方法有两个缺点。第一，要把财会系统与客户关怀系统耦合起来。第二，它的前提条件是我们能支配财会系统。在大多数情况下，财会系统也是封装好的应用，因此不太可能定制。

使用一种特殊的转换器（即内容扩充器）访问外部数据源，以便在消息中增加缺少的信息。



内容扩充器使用输入消息中的信息（如键字段）从外部数据源中获取数据。当内容扩充器从数据源获得所需的数据后，就把它们附加到消息中。来自输入消息的原始信息可能添加到结果消息中，也可能不再需要，这取决于接收应用的特定需求。

由内容扩充器注入的附加信息必须保存在系统中的某个位置。这些新数据一般从以下来源得到：

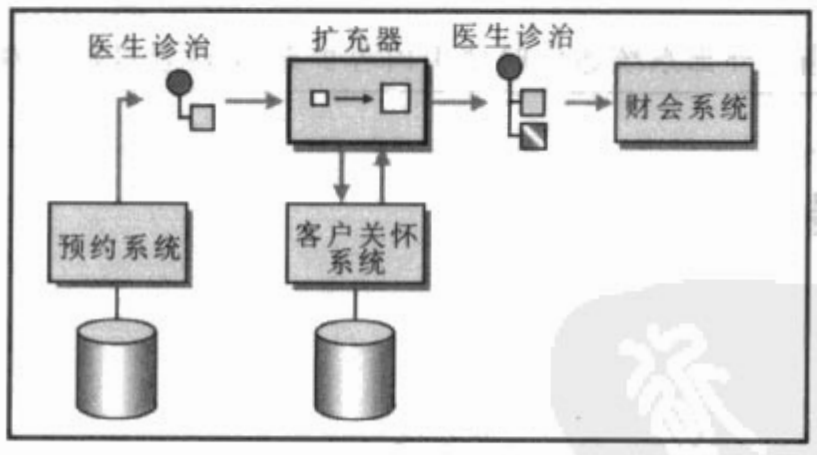
1. **计算**：内容扩充器可以计算出缺少的信息。在这种情况下，额外的数据是由算法产生的。例如，如果接收方的系统需要城市和州的缩写，但是输入的消息只包含邮政编码，那么可由算法来补充城市和州的信息。接收方系统可能要求采用的数据格式必须给出消息的总长度。内容扩充器可以累加所有消息字段的长度，从而计算出消息的总长度。这种形式的内容扩充器非常类似于基本的消息转换器，因为它不需要外部数据源。

2. **环境**：内容扩充器可以从操作环境中获取额外的数据。最常见的例子是时间戳。例如，接收方系统要求每个消息必须携带时间戳。如果发送方系统没有提供该字段，内容扩充器可以从操作系统中获取当前时间并把它添加到消息中。

3. **其他系统**：这是最常见的选择。内容扩充器必须从其他系统中获取缺少的数据。其数据源有多种形式，包括数据库、文件、LDAP 目录、应用或手工输入缺失数据的用户。

在许多情况下，内容扩充器所需的外部数据源可能位于其他系统中，或位于企业外部。因此，内容扩充器与数据源之间可以通过消息传递或其他通信机制（见第 2 章“集成方式”）进行通信。由于内容扩充器与数据源之间的交互定义为同步的（只有在数据源返回所请求的数据之后，内容扩充器才能发送扩充后的消息），因此采用同步消息传递比采用异步消息传递的效率高。内容扩充器和数据源从根本上是紧耦合的，因此使用消息通道实现它们之间的松耦合没有太大的必要。

回到前面的例子，我们可以插入一个内容扩充器，从客户关怀系统中获得额外的数据（如下图所示）。采用这种方法，预约系统就不必与保险信息或客户关怀系统打交道，这就实现了与这些系统的松耦合。它只要发布医生诊治消息即可。内容扩充器组件负责取得所需的数据。财会系统也能保持与客户关怀系统的独立。



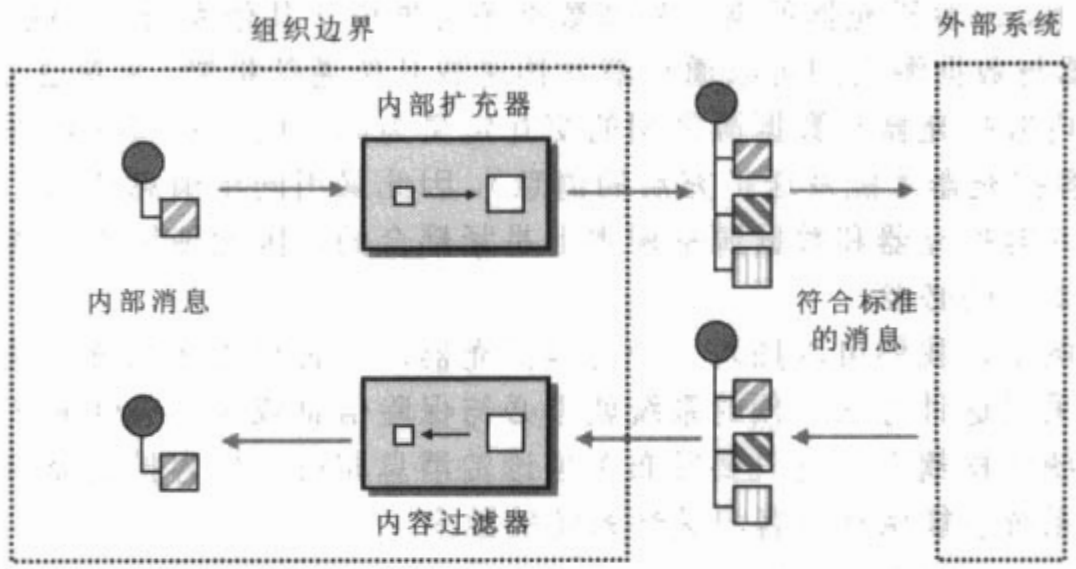
在病患示例中使用扩充器

内容扩充器往往用于解析包含在消息中的引用。为了减小消息的大小，并使消息易于管理，我们一般只传递对象的引用，而不是传递包含所有数据元素的完整对象。这些引用通常采用键或唯一 ID 的形式。当系统要处理消息时，必须根据包含在原始消息中的对象引用提取出所需的数据项。我们使用内容扩充器来完成这项任务，但是这也会带来一些明显的影响。使用引用可以减少原始消息的数据量，但是需要额外查找数据源。使用引用是否能提高效率，这取决于有多少组件能直接使用引用完成操作，而多少组件需要采用内容扩充器恢复部分原始消息内容。例如，如果消息在到达最后的接收者之前要经过大量中间组件，那么使用对象

引用可以极大地减少消息的通信量。我们可以在最后一步进行最终的消息接收前插入一个内容扩充器，把缺少的信息加载到消息中。如果消息中已经包含我们不想沿路传递的数据，那么可以使用声明标签存储数据，并在消息中包含指向此数据的引用。

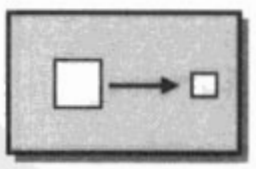
例：与外部系统通信

与外部系统通信时，如果系统要求消息必须遵循特定的消息标准（如 ebXML），通常可以使用内容扩充器。这些标准中，大多都要求消息很大，而且有很长的数据列表。如果我们尽可能简化内部消息的格式，并且在把消息发送到组织外部之前，再使用内容扩充器添加缺少的字段，就能极大地简化内部操作。同样地，可以使用内容过滤器把输入消息中不必要的信息裁剪掉（如下图所示）。



当与外部系统通信时使用内容扩充器/内容过滤器对

8.4 内容过滤器



如果消息接收者要求的数据比消息创建者提供的数据多，或者创建者没有提供接收者所要求的数据时，可以使用内容扩充器来满足要求。但是，在很多情况下接收者的需要正好相反，即要从消息中删除数据元素。

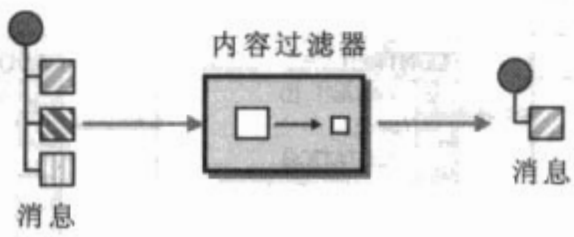
处理一个很大的消息时，如果只对其中的少量数据项感兴趣，应该如何简化？

为什么我们想从消息中删除有价值的数据元素？一个常见的原因是为了安全。从一个

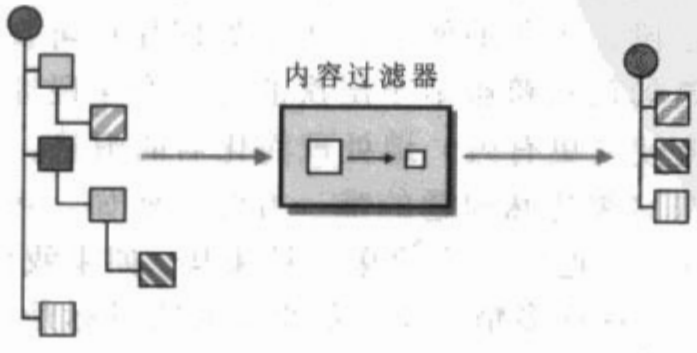
服务请求数据的应用可能无权查看应答消息中包含的所有数据元素。服务提供者可能对安全方案一无所知，并且无论用户是谁，它总是返回全部数据元素。为此，我们必须增加一个操作，根据请求者的认证身份删除敏感的数据。例如，工资系统公布了一个简单的接口，通过这个接口能返回员工的所有数据。这些数据包括工资信息、社会安全号和其他敏感信息。如果你想建立一个能查询员工起始工作日期的服务，在把消息返回给请求者之前，就必须从结果消息中删除所有敏感信息。

删除数据元素的另一个原因是为了简化消息处理和降低网络通信量。在许多实际应用中，消息的处理是从接收到来自业务伙伴的消息开始的。显然，我们应该根据第三方制定的标准消息格式进行通信。许多标准化组织和委员会为某些行业和应用定义了标准的 XML 数据格式，其中比较著名的包括 RosettaNet、ebXML、ACORD 等。尽管这些 XML 格式有利于指导外部系统基于协商好的标准相互通信，但是这种由“委员会制定”的标准往往是极大的文档。许多文档都有数百个字段，包含多个嵌套层次。如此巨大的文档很难用于内部消息交换。例如，如果要映射的文档有几百个元素，大多数可视化（拖放方式）的转换工具就无法使用。而且，系统调试也会变得相当困难。因此，我们要简化输入文档，让它只包含内部处理步骤实际所需的元素。从某种意义上说，删除元素是为了增强消息的有效性，因为删除了冗余和不相关的字段，才能得到更有意义的消息，并且减少了开发人员的出错机会。

使用内容过滤器从消息中删除不重要的数据项，只留下重要的数据项。

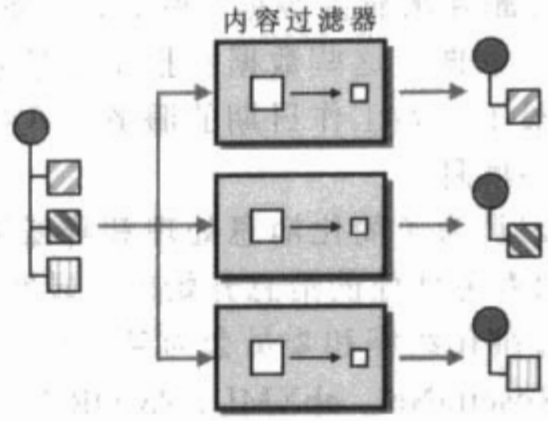


内容过滤器不但要能删除数据元素，还要能简化消息的结构。消息一般表示为树型结构。许多消息是从外部系统或封装好的应用产生的，其中包含多层嵌套的重复数据组，这是因为消息是按通用、规范化数据库结构建模的。通常，已知一些限制和假设条件，使得这种嵌套层次变得多余，内容过滤器可以把层次化的结构“削”成一个简单的元素列表，使之更易于由其他系统理解和处理。



使用内容过滤器扁平消息层次结构

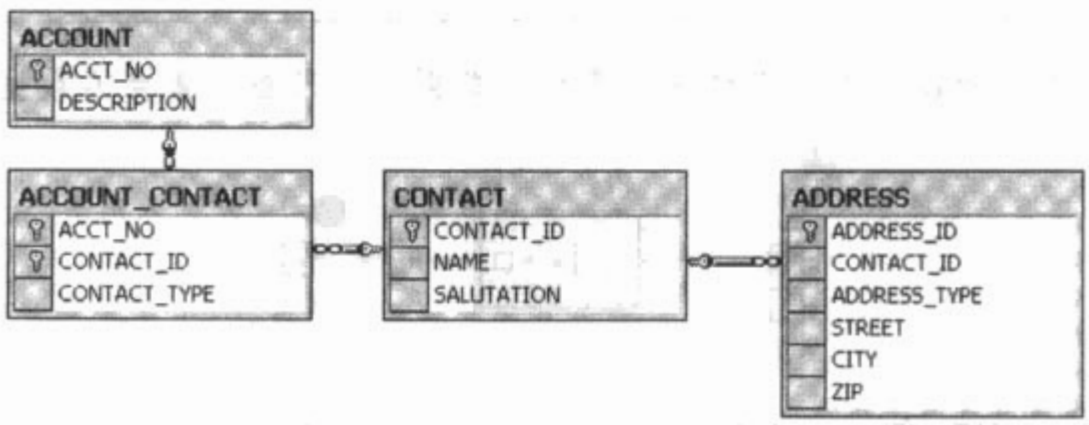
多个内容过滤器可以实现为一个静态的分解器（见“分解器”），从而把一个复杂的消息分解为多个独立的消息，每个消息对应原来大消息中的一个部分（如下图所示）。



使用多个内容过滤器作为静态分解器

例： 数据库适配器

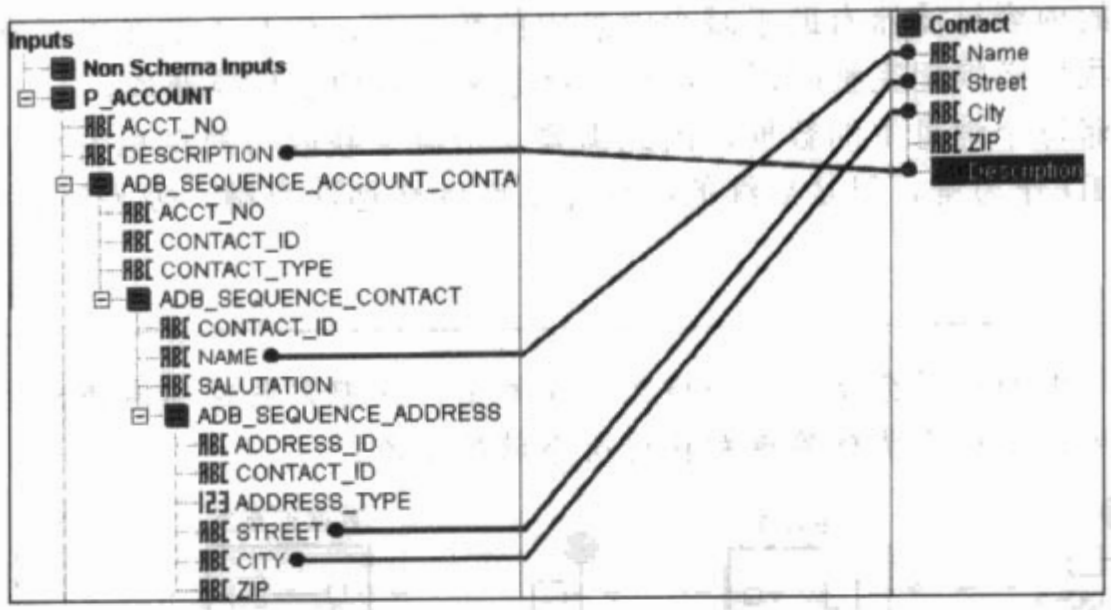
许多集成套件提供了通道适配器来连接已有的系统。在很多情况下，这些适配器所发布消息的格式类似于应用的内部结构。例如，假设把数据库适配器连接到有以下模式的数据库。



数据库模式示例

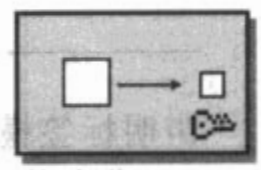
物理数据库模式应该把相关的实体存储在不同的表中，这些表通过外键和关系表连接（如用 ACCOUNT_CONTACT 连接 ACCOUNT 和 CONTACT 表）。许多商业数据库适配器会把这些相关的表转换为层次化的消息结构，其中可能包含额外的字段，如主键和外键，这些字段可能与消息接收者无关。为了简化消息的处理，可以使用内容过滤器扁平消息的结构，并且只取出相关的字段。下面的例子说明了如何使用可视化转换工具实现内容过滤器。从中可以看到，消息是如何从跨越多个层次的几十个字段缩减为只包含五个字段的简单消息。其他组件能够更方便（更有效）地处理简化后的消息。

内容过滤器并不是解决这类特殊问题的唯一方法。例如，可以在数据库中配置一个视图（view），由它解析表关系，并返回一个简单的结果集。如果我们能在数据库中添加视图，这将是一种简单的方法。不过在许多情况下，企业集成的目标是尽可能减少对应用的干扰，其中也包括不能向数据库中增加视图。



简化由数据库适配器发布的消息

8.5 声明标签



消息中缺少所需的数据项时，可以使用内容扩充器加以补充。反之，如果要将消息中的多余数据项删除，可以使用内容过滤器。但是，我们可能只是临时删除这些数据项。例如，消息中包含了后续消息流要用到的数据项，但是所有中间处理步骤都不会使用这些数据项。因此，为了避免性能受到影响，也为了降低调试的难度，我们没有必要在中间处理步骤中携带这些额外的数据。

在不损害信息内容的前提条件下，怎样减少跨系统发送的消息的数据量？

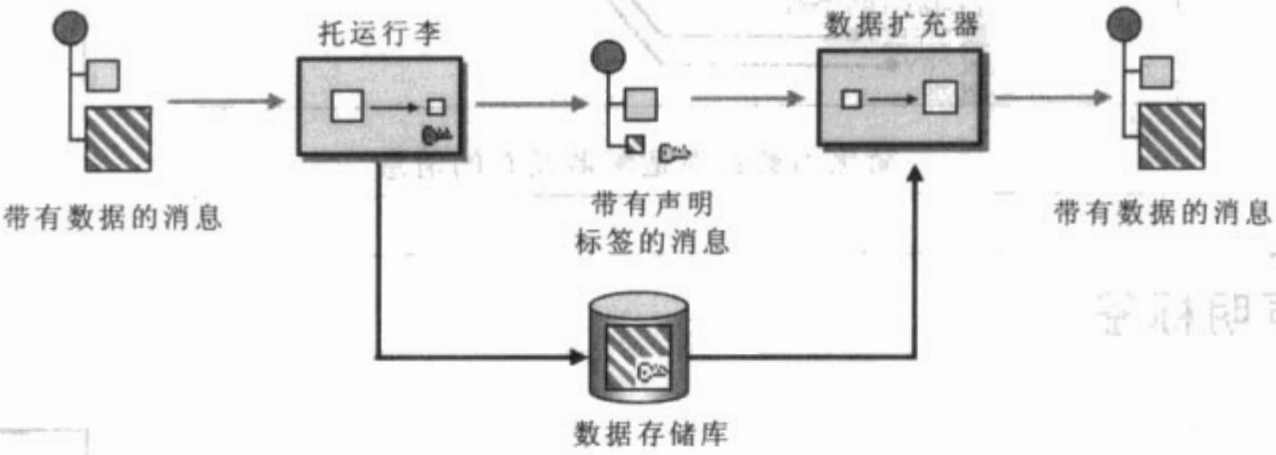
通过消息来传递大量数据的效率可能不高。有些消息传递系统甚至对消息的长度做出了硬性的规定。另外一些消息传递系统使用 XML 来表示数据，这就导致消息长度成数量级地增加。因此，尽管消息传递为信息的传输提供了最可靠和最适合的方法，但它不是最高效的方法。

而且，消息中携带的数据越少，中间系统对无关信息的依赖就越少。例如，如果我们通过一系列的中间组件把地址信息从一个系统发送给另一个系统，中间组件就要对地址数据做出假设，并对消息的数据格式产生依赖。尽可能减少消息的大小可以降低产生这些潜在假设的可能性。

一个简单的内容过滤器有助于减少消息中的数据量，但是无法在以后重新恢复消息的内容。因此，我们必须把完整的消息信息存储起来，以便以后能获取。

由于要存储每个消息中的数据，因此需要一个键来获取与消息相关的正确的数据项。可以使用消息 ID 作为键，但是后续的组件无法继续传递这个键，因为消息 ID 会随着消息而改变。

把消息中的数据存储在一个持久存储库中，并把声明标签传递给后续组件。这些组件可以使用声明标签重新获得所存储的信息。



声明标签模式包括以下步骤：

1. 带有数据的消息到达。

2. 托运行李（Check Luggage）组件为信息产生唯一的键。该键可以作为后续的声明标签。

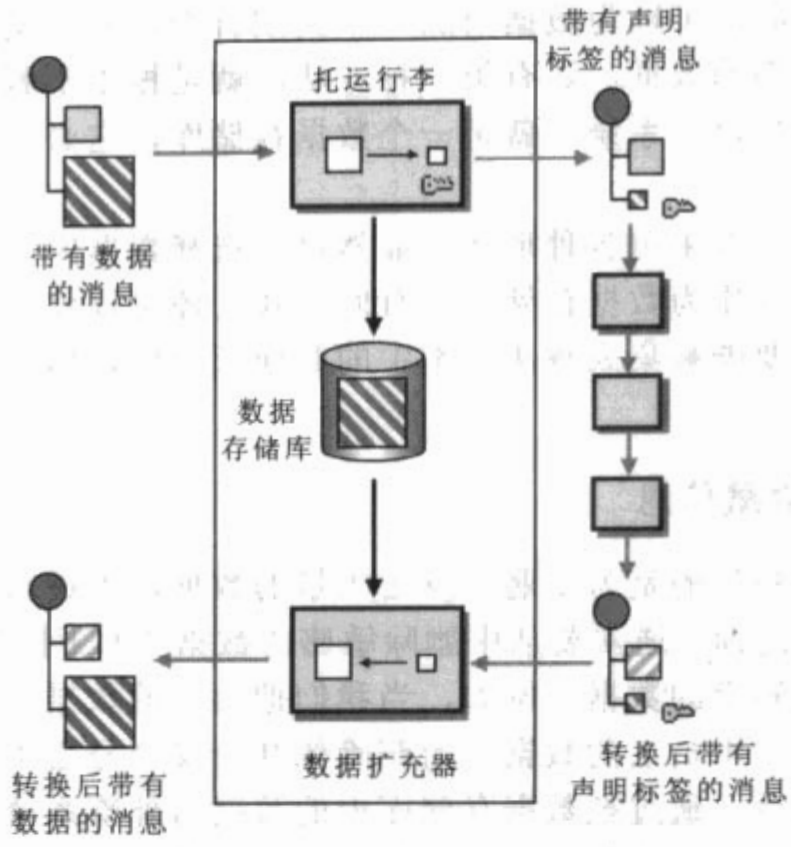
3. 托运行李组件从消息中取出数据，并把数据存储在一个持久的存储库中，如文件或数据存储库。它把键与所存储的数据相关联。

4. 从消息中删除持久存储的数据，并在消息中增加声明标签。

5. 其他组件可以使用内容扩充器，根据声明标签取得数据。

这一过程类似于机场的行李托运。如果你不想随身携带所有的行李，可以在航线柜台处办理行李托运。工作人员将在你的机票上贴上行李的托运号，通过它能找出每件行李。到达最终的目标后，就能重新取回行李。

如上图所示，包含在原始消息中的数据仍然要“传送”给最终的目的接收者。那么采用这种方法又有什么好处呢？显然，通过消息传递来传送数据的话，效率要比把数据存储在一个中央数据存储库中低。例如，消息可能要经过多个组件，但是在这个过程中，每个组件并不需要其中的大量数据。使用消息传递系统，每个中间组件都要编组（marshal）和解组（unmarshal）消息中的数据，或者对这些数据加密和解密。这些操作都需要占用大量的 CPU 时间，对那些不需要这些数据的中间组件来说，执行这样的操作是完全没有必要的。如果消息要经过许多组件，最后再返回到发送者，也可以使用声明标签。在这种情况下，托运行李组件和内容扩充器属于同一组件，数据根本不需要通过网络传送（如下图所示）。



数据可以本地存储和提取

8.5.1 选择键

如何为数据选择合适的键？可行的方法包括：

1. 消息体中可能已经包含有业务键，如客户 ID，可以把它作为键。
2. 消息中可能包含一个消息 ID，可以用于将数据存储库中的数据与消息相关联。
3. 可以生成一个唯一的 ID。

重用已有的业务键看起来是最简单的方法。如果必须去掉客户的一些详细信息，以后可以再通过客户 ID 引用这些信息。当把这个键传递给其他组件时，首先要确定是否有必要让这些组件清楚该键是一个客户 ID，而不只是一个抽象键。把键表示为抽象键有一个好处，这样一来可以采用相同的方法处理所有键，而且可以建立一个通用的机制，根据抽象键从数据存储库获取数据。

虽然可以方便地使用消息 ID 作为键，但是一般不应该这样做。如果使用消息 ID 作为获取数据的键，会使同一个数据元素有二义性，进而引发冲突。例如，假设要把声明标签引用传递给另一个消息。为这个新消息指派了新的唯一 ID，不过这样一来，我们就不能使用这个新的 ID 从数据存储库中获取数据了。使用消息 ID 作为键只适用于一定的范围，即只限于在一个消息范围内访问数据。因此，通常更好的做法是指派一个新的元素存储键，而避免这种不好的“元素重用”。

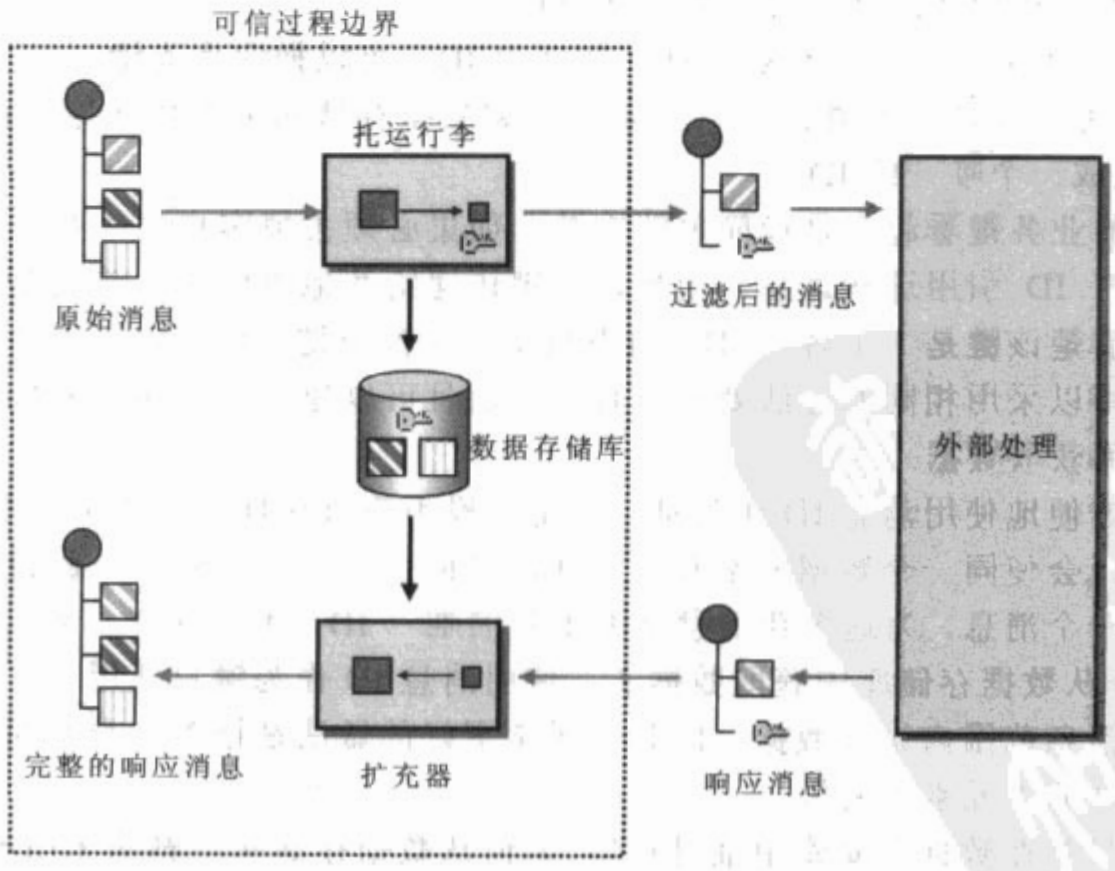
数据可能只是在数据存储库中临时存储。如何从数据存储库中删除无用的数据？为此，可以修改从数据存储库中获取数据的语义，从而在读取数据时将其删除。采用这种方法，数据只能获取一次，从安全的角度考虑，这样做是非常适合的。但是，这样无法使多个组

件访问相同的数据。为此，可以为数据附加一个到期日期，并定义一个垃圾收集进程，让它周期性地删除所有到期的数据。还有第三种方法，就是根本不删除任何数据。这是因为我们使用的业务系统（如财会系统）就是一个数据存储库，它必须把所有的数据都保留在系统中。

数据存储库的实现可以采用各种形式。显然可以选择数据库，不过一组 XML 文件或内存中消息存储库也可以作为数据存储库。有时，我们还可以使用应用作为数据存储库。重要的是，数据存储库要能被集成解决方案中的其他部分所访问，只有这样这些部分才能重新构建出原始的消息。

8.5.2 使用声明标签隐藏信息

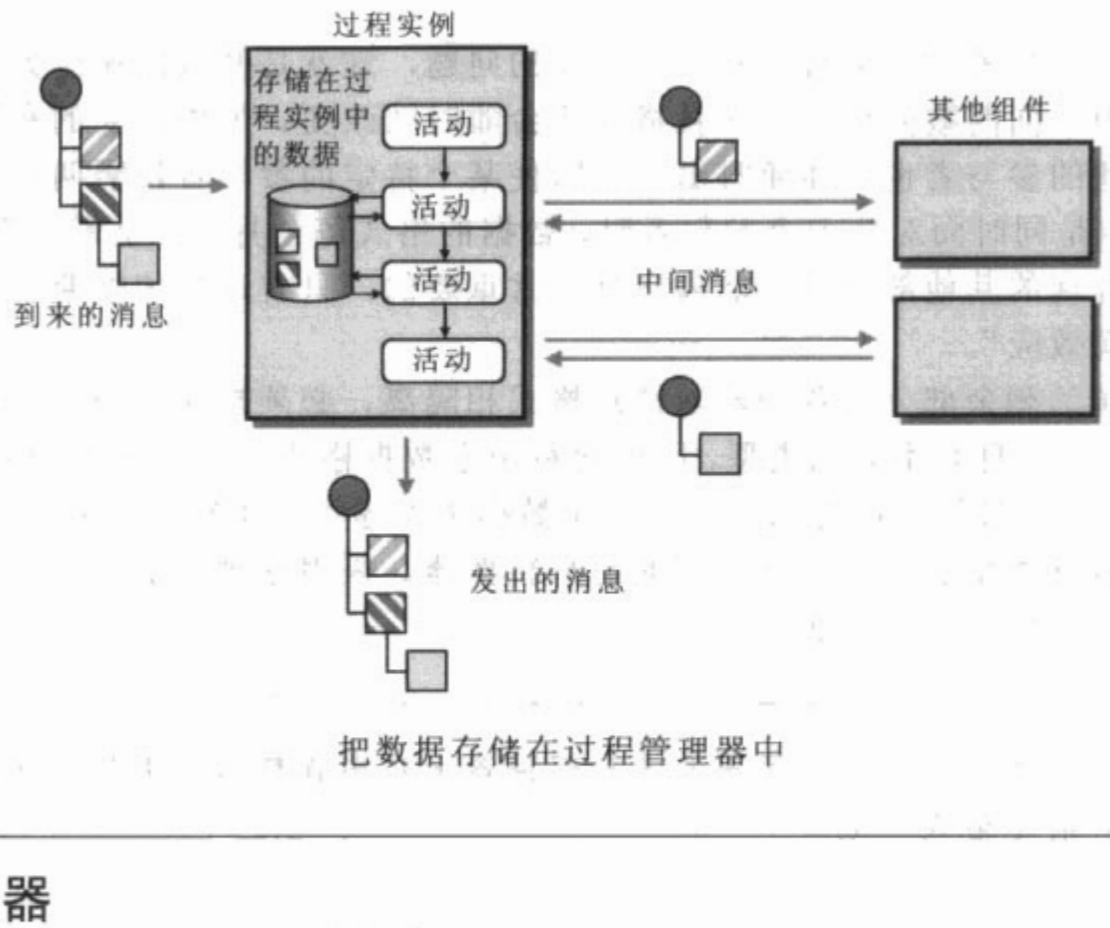
尽管声明标签的最初目的是为了避免发送大量的数据，不过它也可用于其他目的。要把消息发送给外部系统之前，通常要从中删除敏感的数据（如下图所示）。也就是说，外部的系统只能接收它需要知道的数据。例如，当我们把员工的数据发送给外部系统时，最好能通过某种唯一的 ID 引用员工的数据，而删除像社会安全号之类的（敏感）字段。当外部系统完成所需的处理后，通过把数据存储库中的数据与外部系统返回的消息合并，可以重新构造出完整的消息。还可以为这些消息生成特殊的唯一键，根据外部系统拥有的键，对外部系统所能采取的行动加以限制。这样能避免外部系统恶意地向系统发送消息。当外部系统尝试使用错误的键获取消息中的数据时，内容扩充器会堵住包含非法键（或者键已经到期，或者已被使用）的消息。



删除发送到可信过程边界外的敏感消息数据

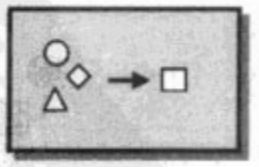
8.5.3 使用带声明标签的过程管理器

如果要与多个外部系统交互，过程管理器可以提供声明标签的功能。当有消息到达时，过程管理器会创建过程实例（有时也被称为任务或作业）。它允许为每个独立的过程实例关联额外的数据。实际上，此时过程引擎就相当于数据存储库，存储了消息中的数据。过程管理器在向外部系统发送消息时，只会发送与该系统相关的数据，消息中不必包含原始消息中的所有信息，因为其他信息会存储在过程的数据库中。当过程管理器接收到外部系统返回的响应消息时，它会把新数据与已经保存在过程实例中的数据合并。



把数据存储在过程管理器中

8.6 规范器



在企业到企业（B2B）集成中，企业经常要从不同的业务伙伴那里接收消息。这些消息可能有相同的含义，但是采用了不同的格式，消息的格式是由合作伙伴的内部系统及合作伙伴的偏好决定的。例如，在 ThoughtWorks 为按收看计费的提供商构建了一个解决方案，它必须从 1700 多个用户那里接受并处理收看信息，其中大多数用户没有采用标准格式。

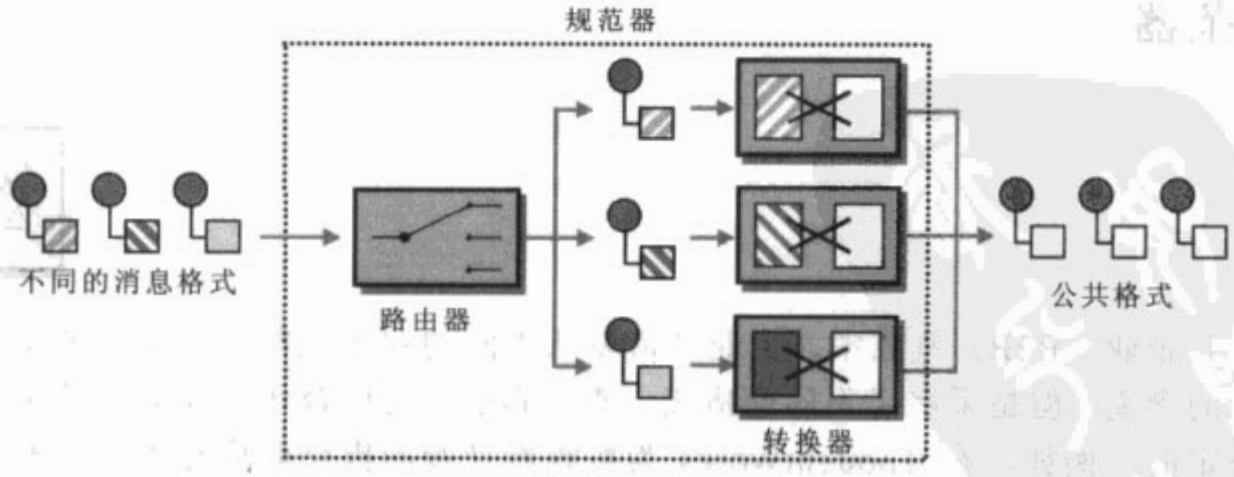
如何处理语义上等价但是格式不同的消息？

从技术的角度考虑，最简单的方法是让所有的参与者都使用统一的消息格式。如果这是一家大公司，并且能控制 B2B 的信息交换或供应渠道，就可以采用这种方法。例如，通用公司会采用公共消息格式从供应商那里接收订单状态的更新消息，我们可以相当肯定地说：一个供应商（Joe's Supplier 公司）的业务很可能遵循通用公司的规定。但是在另外的许多情况下，这一点是做不到的，企业达不到这种程度。相反地，许多业务模型把消息的接收者定位为信息的“聚合器”，业务模型与每个参与者达成的协议中包括这样一条：要尽可能减少对系统基础设施的改变。因此，聚合器要能处理以各种数据格式到达的信息，包括 EDI（电子数据交换）记录、逗号分隔的文件或通过电子邮件传送的 XML 文档或 Excel 表单。

当与大量的参与者打交道时，有一个重要的问题，这就是修改的速度如何。每个参与者不但喜欢采用不同的数据格式，而且格式还会随时间改变。另外，新的参与者会不断加入进来，而其他的参与者也可能随时退出。即使某个特定的参与者每隔两年才会改变一次数据的格式，但是同时面对几十个参与者时，数据的格式每个月、甚至每周都会发生改变。把这些改变与处理的其他部分尽可能隔离是非常重要的，只有这样才能避免改变对整个系统造成的“涟漪效应”。

为了使系统的剩余部分与各种输入消息格式相隔离，必须把输入消息转换为一种公共的格式。由于输入消息有不同的类型，因此每种消息数据格式要有一个不同的消息转换器。为了达到这个目的，最简单的方法是使用一组数据类型通道，每个通道对应一个消息类型。然后再把各个数据类型通道连接到不同的消息转换器。采用这种方法的缺点是，大量的消息格式被转化为同样多的消息通道。

使用规范器，通过一个定制的消息转换器来路由各种消息类型，使结果消息与一种公共消息格式匹配。



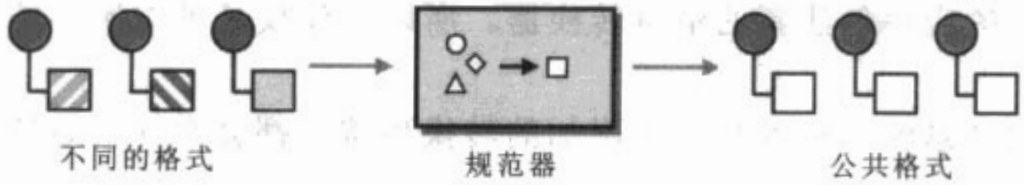
规范器为每种消息格式提供一个消息转换器，并把输入消息通过一个消息路由器路由给正确的消息转换器。

8.6.1 检测消息格式

为了把消息路由给正确的消息转换器，消息路由器要能检测出输入消息的类型。许多消息传递系统在各个消息的首部设置了一个类型指示符字段，以便于消息类型的检测。但是在许多 B2B 实例中，到达的消息与企业内部消息传递系统使用的消息不兼容，而是采用了各种不同的格式，如以逗号分隔的文件或不带相关模式（schema）的 XML 文档。尽管我们希望能为所有的输入数据格式配备一个类型指示符，但在实际中很难做到这一点。因此，必须考虑出一种更通用的方法来识别输入消息格式。对于不带模式的 XML 文档来说，一种常用的方法是使用根元素的名字识别出正确的类型。如果多种数据格式使用了相同的根元素，则可以使用 XPath 表达式确定是否存在特定的子结点。要识别以逗号分隔的文件，需要更多的技巧。有时可以根据字段的数量和数据的类型（如数字或字符串）来确定消息的类型。如果数据是以文件方式到达的，最简单的方法是使用文件名或文件夹结构取代数据类型通道。每个业务伙伴可能使用一种唯一命名约定来为文件取名，这样消息路由器就可以使用文件名把消息路由到适当的消息转换器。

通过使用消息路由器，还允许多个业务伙伴使用相同的转换。如果多个业务伙伴采用了相同的消息格式，或者转换相当通用，足以应对多种消息格式，这就很有用。例如，即使 XML 文档采用了不同的格式，XPath 表达式也能从 XML 文档中提取出元素。

由于规范器经常用在消息传递解决方案中，我们为它创建了一个速记图标：



规范器工作原理

8.7 规范数据模型

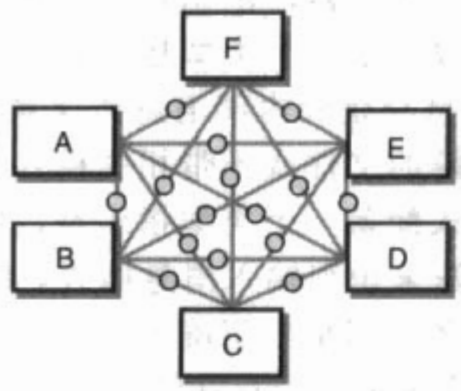
要设计多个应用通过消息传递协作运行。每个应用都有自己的内部数据格式。

当集成的应用使用不同数据格式时，如何把它们的相互依赖减到最少？

独立开发的应用一般会使用不同的数据格式，这是因为，每种格式都是专门针对各个应用而设计的。应用要从其他未知的应用那里接收消息或者向它发送消息时，应用自然会使用对自己最方便的消息格式。同样地，商业适配器要集成已经封装好的应用，通常会采用类似于应用内部数据结构的数据格式来发布和消费消息。

消息转换器可以识别并处理消息格式中的差异，而不会改变应用，也不需要应用了解

对方的数据格式。但是，如果有大量的应用要彼此通信，则必须在每一对通信的应用之间插入一个消息转换器（如下图所示）。



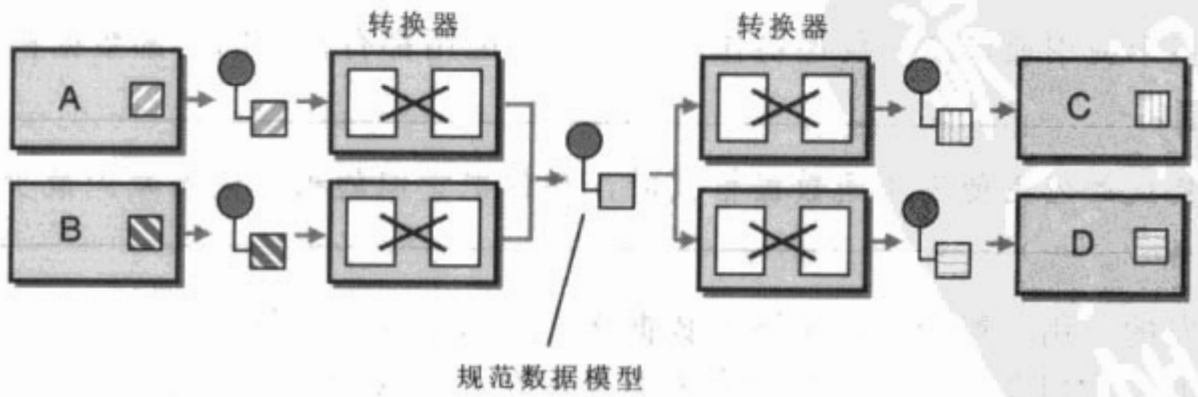
随着系统数量的增加，连接的数量也会爆炸增长

如果采用这种方法，需要大量的消息转换器，特别是当所集成的各个应用可能发布或消费多种消息类型时，需要的消息转换器更多。所需的消息转换器数量会随着所集成应用的数量呈指数增长，很快就会使系统变得难以管理。

对于相互通信的两个应用，尽管消息转换器在它们所用的消息格式之间提供了一个间接层，仍然要依赖于这两个应用使用的消息格式。因此，如果应用的数据格式发生了改变，有变化的应用和与之通信的其他应用之间如果有消息转换器，所有这样消息转换器都要修改。同样地，如果在解决方案中增加新的应用，如果有其他应用要与这个新应用交换信息，就要为每一个这样的应用创建新的消息转换器。所以，仅仅是维护所有的消息转换器就会带来一场噩梦。

还要记住，向消息流中插入每一个附加的转换步骤，都会增加消息传递的延迟并降低消息的吞吐量。

设计一种规范数据模型，使之独立于所有特定的应用。要求每个应用能产生和消费使用这种公共格式的消息。



规范数据模型为应用的各种数据格式提供了另一个间接层。如果新的应用添加到集成解决方案中，只需要实现它与规范数据模型之间的转换，而不必考虑集成解决方案中已经

有多少个应用参与。

如果只有少量应用参与到集成解决方案中，那么使用规范数据模型就显得太复杂了。但是随着应用数量的增加，它的作用很快就能发挥出来。假设每个应用都要与其他各个应用收发消息，由 2 个应用组成的解决方案如果采用直接数据格式转换，只需要 2 个消息转换器，而采用规范数据模型则需要 4 个消息转换器。由 3 个应用组成的解决方案，不管采用直接转换的方式，还是采用规范数据模型，都需要 6 个消息转换器。但是，由 6 个应用组成的解决方案如果采用直接转换方式，需要 30 个消息转换器！当使用规范数据模型时，只需 12 个消息转换器。

如果已有的应用将来可能被其他应用替换，采用规范数据模型也是非常有用的。例如，假设许多应用都与某个遗留系统交换信息，而该遗留系统将来很可能要被新的系统替代，倘若在最初的解决方案中采用规范数据模型，就会极大地减少从一个应用向另一个应用切换的代价。

8.7.1 转换的选择

如何使应用遵循公共的格式？可选择的基本方法有以下三种：

1. **修改应用的内部数据格式。**从理论上讲，采用这种方法是可行的，但是在复杂的现实环境中却不太可能。如果能让每个应用从一开始就使用相同的数据格式，还不如使用共享的数据库来代替消息传递。

2. **在应用的内部实现消息传递映射器。**自定义的应用可以使用映射器来产生所需的数据格式。

3. **使用外部消息转换器。**使用外部消息转换器把特定于应用的消息格式转换为规范数据模型指定的消息格式。如果要转换已封装应用的数据，这可能是唯一的选择。

使用消息映射器还是外部消息转换器，这是由转换的复杂程度和应用的可维护性决定的。已经封装好的应用一般不能使用消息传递映射器，因为无法获得它的源代码。对于自定义的应用来说，具体采用哪种方法取决于转换的复杂程度。许多集成工具套件提供了可视化的转换编辑器，能够更快地构造映射规则。但是如果转换很复杂，这些可视化的工具就会变得非常笨拙。

当使用外部消息转换器时，必须把公共（规范）的消息与专用（特定于应用）的消息区分开来。应用与其消息转换器之间的消息属于专用消息，因为其他任何应用不会使用这些消息。一旦消息转换器把消息转换为与规范数据模型兼容的格式，它就成为公共的消息，并且能由其他系统使用。

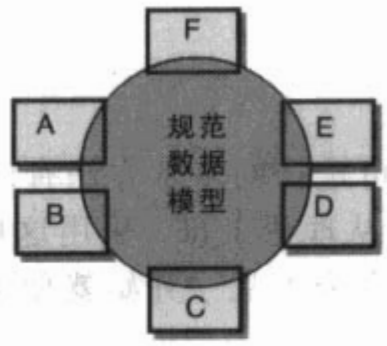
8.7.2 双重转换

使用规范数据模型会给消息流增加一定程度的开销。每个消息都要经历两次转换，而不是一次转换：一次是从源应用的格式转换为公共格式，另一次是从公共格式转换为目标应用的格式。因此，使用规范数据模型有时也被称为双重转换（直接从一个应用的格式转换为另一个应用的格式称为直接转换）。每个转换步骤都会给消息流带来额外的延迟。因此，

对于吞吐量很大的系统来说，直接转换是唯一的选择。我们往往要在可维护性和性能之间取得折衷，而使用更易于维护的解决方案应该是首要选择（如采用规范数据模型），除非性能上有要求。对此有一种改进的方法，由于许多转换是无状态的，因此可以让多个消息转换器并行工作，实现转换操作的负载平衡。

8.7.3 设计规范数据模型

设计规范数据模型相当困难：大多数企业都至少有过一次构建“企业数据模型”的失败经历。为了得到一种平衡的模型，设计人员要力求将一个统一的模型平等地适用于所集成的全部应用。然而在实际中，这一理想很难达到。如果规范数据模型不必模拟所有应用内部所用的全部数据集，而只需模拟参与消息传递的一部分数据集，就更有可能成功设计出规范数据模型（如下图所示）。这样就能大大降低创建规范数据模型的复杂性。



只建立相关数据的模型

使用规范数据模型在行政上也有好处。使用规范数据模型有利于开发人员和商业用户采用公司业务范围内的术语讨论集成解决方案，而不必考虑特定包的实现。例如，已经封装好的应用可以采用各种不同的格式表述公共的客户概念，如“账户”、“支付者”和“联系”。定义规范数据模型通常是克服应用之间语义不一致性的第一步（见[Kent]）。

8.7.4 数据格式的依赖性

在开始介绍本模式时给出了一个图，显示出为了实现各个应用之间的转换需要大量的转换器，这个图与“消息代理”中给出的图非常相似。这就提醒我们，在应用之间存在者不同层次的依赖性。消息通道的使用在应用之间提供了公共的传输层，消除了应用中各个传输协议之间的依赖。消息路由器可以提供位置独立性，因而发送方的应用不必依赖接收方应用的位置。通过使用公共数据表示，如 XML，可以消除对任何应用特定数据类型的依赖。最后，规范数据模型可以克服对应用所使用的数据格式和语义的依赖。

只有变化才是不变的真理。因此，符合规范数据模型的消息应该给出格式指示符。

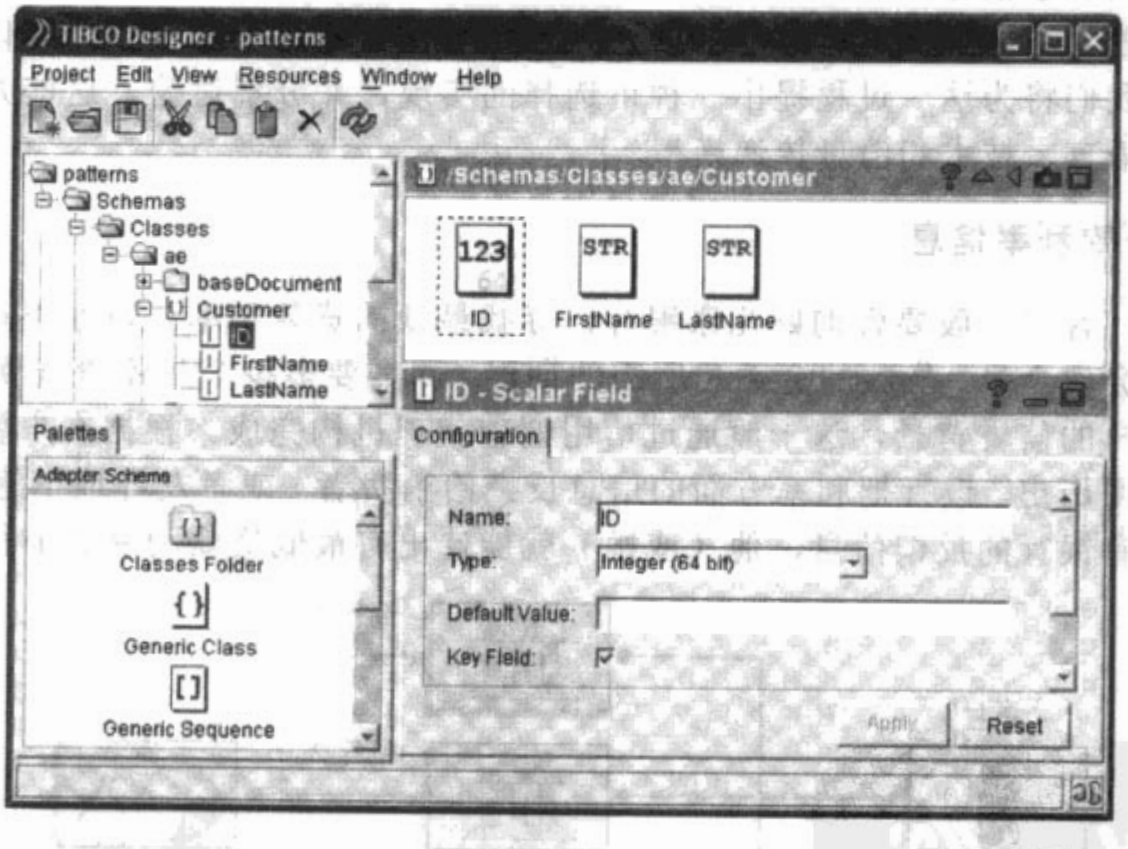
例：WSDL

当从你的应用访问外部服务时，服务可能已经规定了要使用的规范数据模型。在 XML Web 服务领域中，数据的格式由 WSDL（Web Services Definition Language，Web 服务定义

语言，见[WSDL 1.1]）文档指定。WSDL 规定了服务能消费的请求，以及所产生的应答消息采用何种结构。在大多数情况下，WSDL 中规定的数据格式都与提供服务的应用的内部数据格式不同。因此，WSDL 指定了会话双方都能使用的规范数据模型。双重转换由服务消费者中的消息传递映射器或消息传递网关，以及服务提供者中的远程外观（Remote Facade）[EAA]组成。

例： TIBCO ActiveEnterprise

许多 EAI 工具套件都提供了完整的工具集，可以定义和描述规范数据模型。例如，TIBCO ActiveEnterprise 套件提供了 TIB/Designer 工具，允许用户检查所有公共的消息定义。消息定义可由 XML 模式定义导入，或者导出为 XML 模式定义。当使用内建的可视化工具集实现消息转换器时，这个工具为设计人员同时提供了应用特定的数据格式以及存储在中央数据格式知识库中的规范数据模型，从而可以简化两种数据格式之间消息转换器的配置。



TIBCO Designer：维护规范数据模型的 GUI 工具

第9章

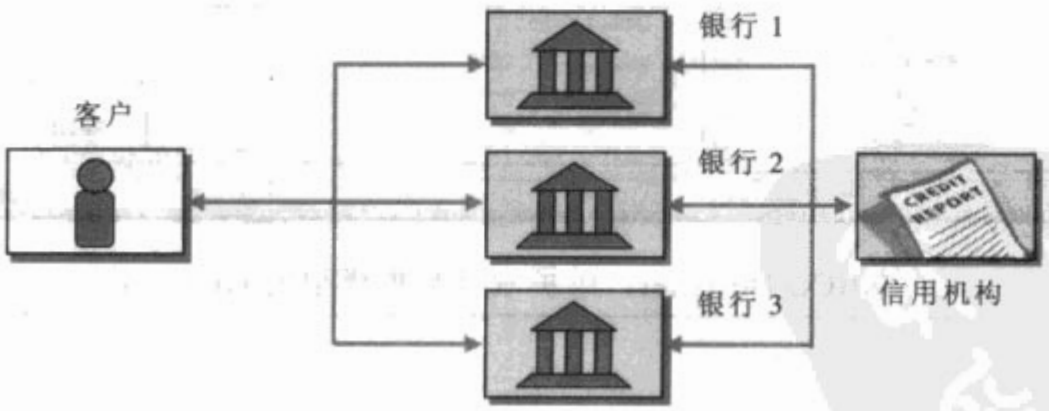
中场演练：复合消息传递

9.1 贷款中介示例

本章将举例说明如何把路由模式与转换模式复合为一个更大的解决方案。为此，我们选择对消费者从多家银行获取贷款利率信息的过程建模。由于我们的重点是介绍集成模式，而不是向消费者介绍金融服务方面的知识，因此要对该业务过程做一定的简化。根据前面定义的模式，我们将为这一过程提出三种可选择的实现，并分别讨论，这些方法分别使用了不同的编程语言、技术和消息传递模型。

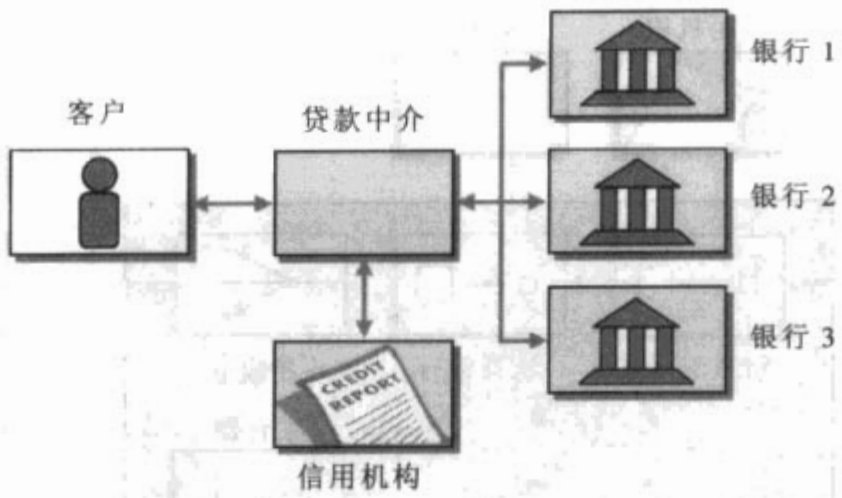
9.1.1 获取贷款利率信息

在贷款时，客户一般要咨询好几家银行，寻找最低的贷款利率。每家银行都要求客户提供自己的社会安全号、贷款的金额和所需的期限（即需要多少个月还清贷款）。然后每家银行将调查客户的信誉背景，这一般通过信用调查代理机构完成。根据客户要求的还款期限和客户的信用历史，银行把利率方面的信息反馈给消费者（或者友好地拒绝消费者）。一旦客户获得所有银行的反馈信息，他（或她）就能选出有最低贷款利率的银行。



消费者与银行洽谈以获得贷款利率信息

由于与多家银行接触并查询贷款利率是一项很麻烦的任务，因此一般由中介机构为消费者提供这个服务。贷款中介通常不属于任何一家银行，但是它能与多家借款机构联络。中介收集客户的数据并咨询信用代理，查询客户的信用历史。根据客户的信用度和历史记录，中介向最适合客户标准的多家银行提出利率查询请求，并收集从银行返回的结果，然后选择最适合的应答信息反馈给消费者。



贷款中介扮演了中间人的角色

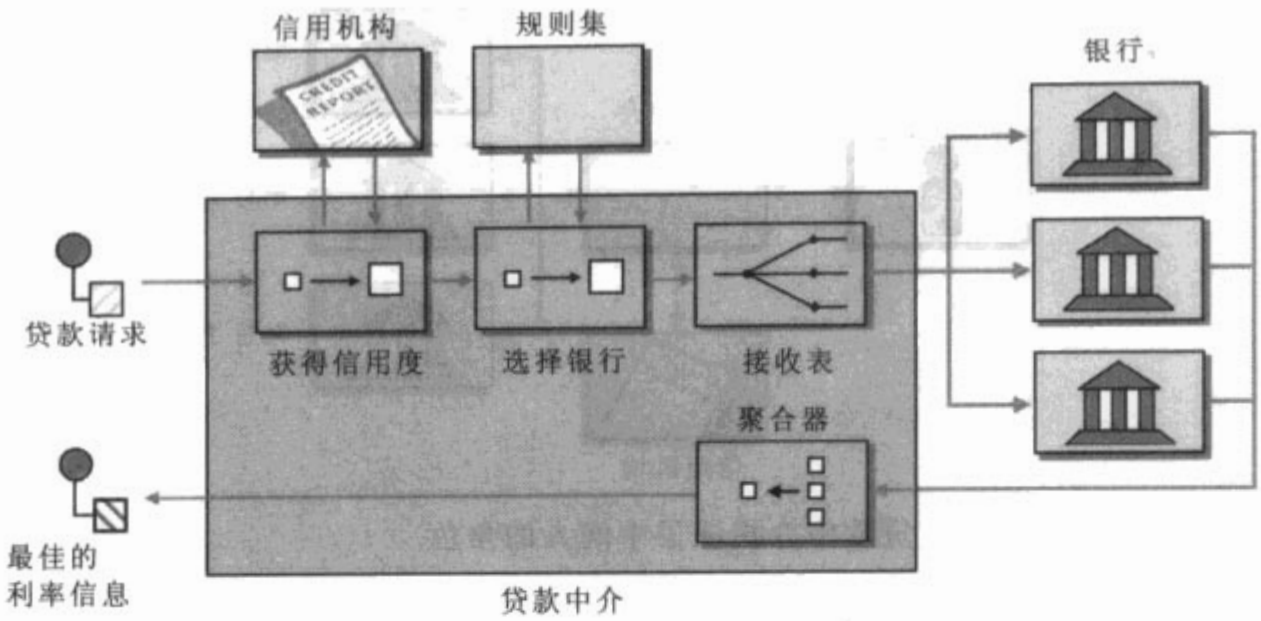
9.1.2 设计消息流

我们想使用前面介绍过的集成模式来设计贷款中介系统。为此，首先列出贷款中介需要完成的各项任务。

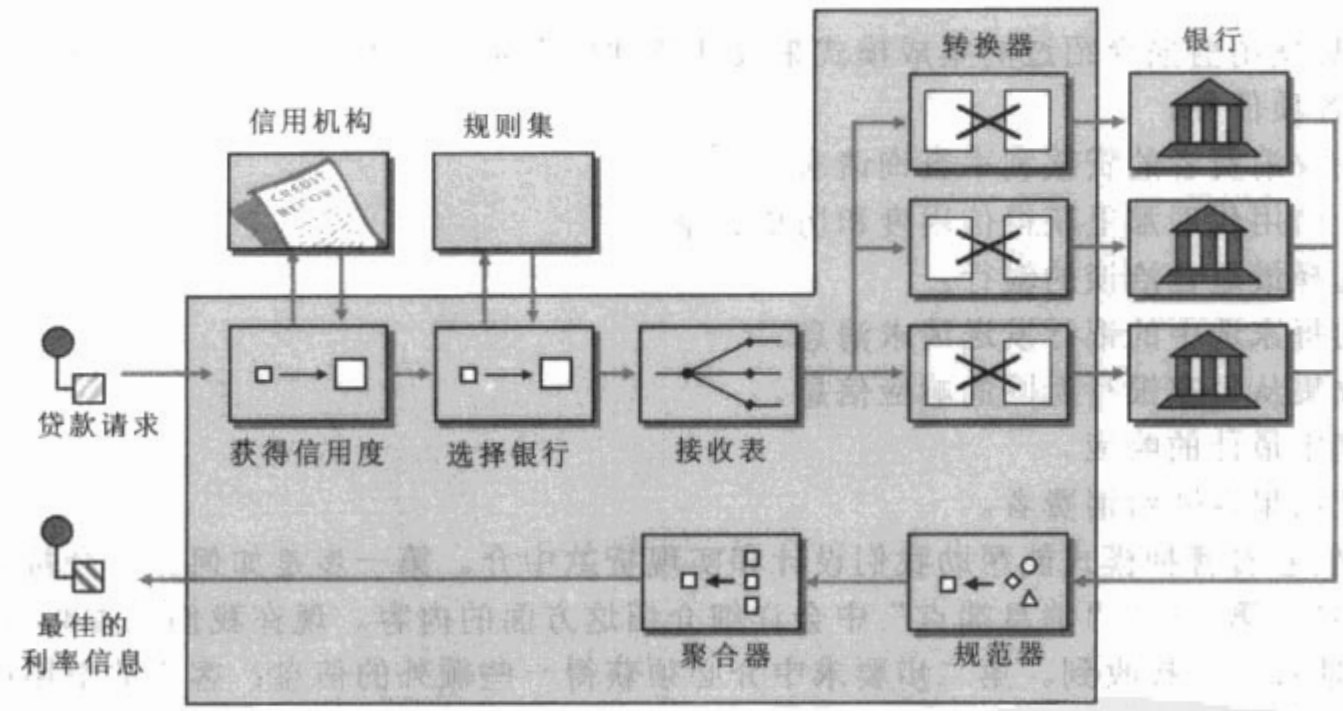
1. 接收消费者的贷款利率查询请求。
2. 从信用代理那里获得信用度和历史记录。
3. 选择最适合洽谈的银行。
4. 向每家选中的银行发送请求消息。
5. 收集从每家银行返回的响应信息。
6. 确定最佳的响应。
7. 把结果返回给消费者。

让我们看看哪种模式能帮助我们设计和实现贷款中介。第一步是如何让中介接收到来的请求消息。第 10 章“消息端点”中会详细介绍这方面的内容。现在我们先省略这一步，假设消息能被中介接收到。第二步要求中介必须获得一些额外的信息：客户的信用度。为了完成这项任务，最好选择内容扩充器模式。中介一旦获得完整的信息，就要决定应该把请求消息路由给哪家银行最合适。为此可以采用另一个内容扩充器计算出请求消息的接收者列表。把请求消息发送给多家接收者，并且重新把响应的消息组合为一个消息，这个任务能够用分散收集器完成。分散收集器可以使用发布-订购通道或接收表把请求消息发送给银行。一旦银行做出利率应答，分散收集器再使用聚合器把各条利率反馈消息组合为单一的一条利率查询消息提供给消费者。如果我们使用这些模式建立消息流的模型，将得到如下图所示的设计方案。

还有一点没有考虑，每家银行可能使用稍有不同的消息格式传递请求和响应信息。因为我们想让路由和聚合逻辑与银行的专用格式分离，所以要在中介与银行之间的通信链路上插入消息转换器。可以使用规范器把每个响应消息转换为相同的格式。



简单的贷款中介设计方案



完整的贷款中介设计方案

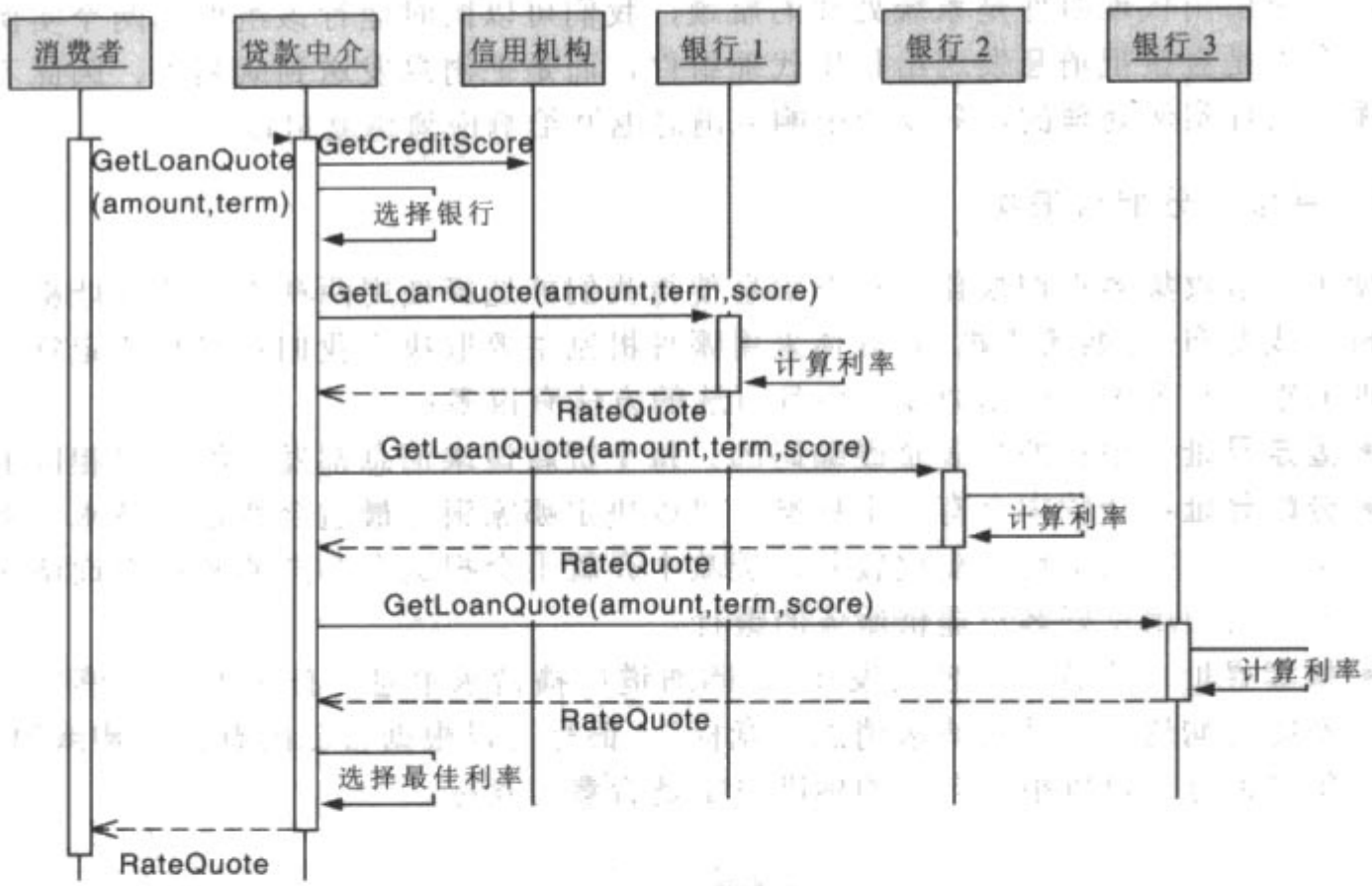
9.1.3 时序：同步与异步

到目前为止，我们已经介绍了消息流和设计贷款中介组件所使用的路由及转换模式，但是中介操作的时序问题还没有讨论。处理操作时序主要有两种选择：

- **同步（顺序）方式**：中介向一家银行提出查询请求，等待返回响应消息后再向另一家银行提出查询请求。
- **异步（并行）方式**：中介同时向多家银行发送查询请求，并等待响应消息的返回。

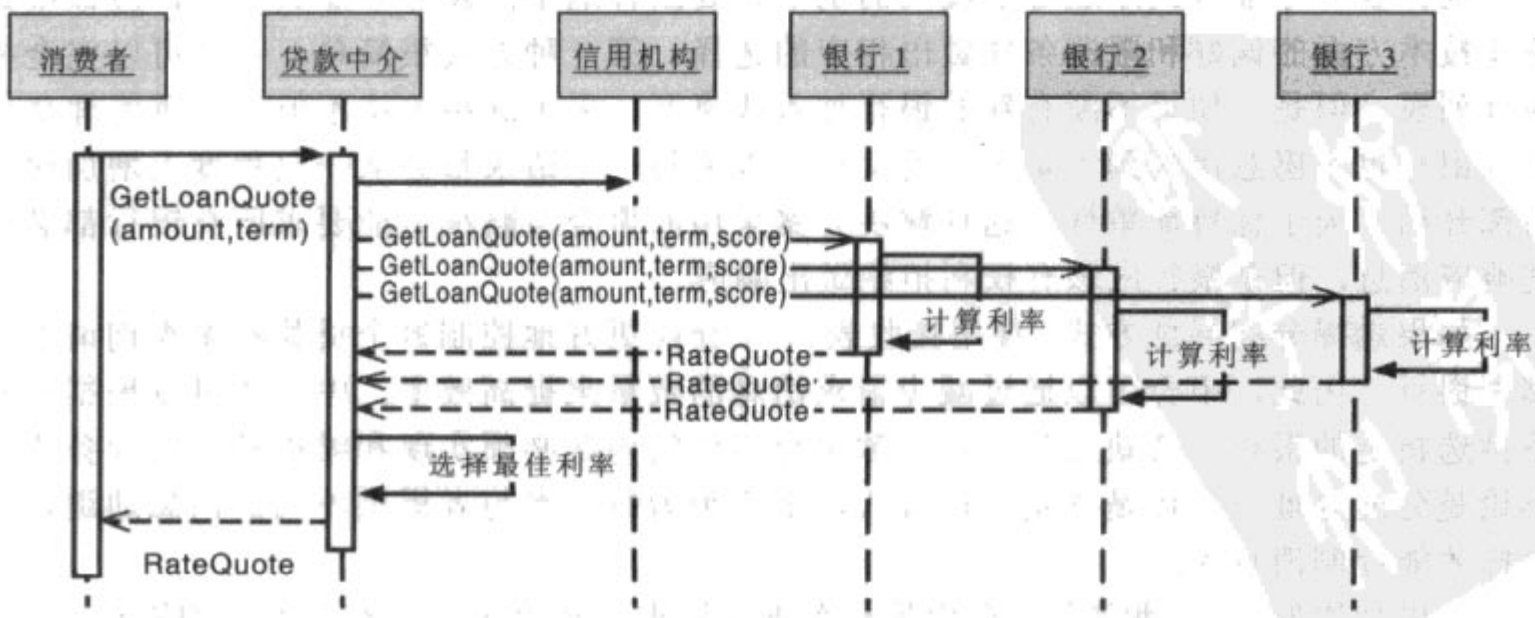
我们可以使用 UML 顺序图说明这两种方式。同步方式将按顺序处理所有的贷款请求（如下图所示）。其优点在于管理更简单，因为我们不必考虑任何并发问题或多线程操作。但是，这种解决方案效率很低，因为它没有充分利用每家银行独立处理业务和同时执行请

求操作的能力。因此，消费者要得到反馈必须等待很长的时间。



同步、顺序的贷款请求处理

异步方式同时把所有的请求消息发送出去，让所有银行都能开始处理。当银行完成计算后，它们把结果返回给贷款中介，从而得到更快的响应。例如，如果所有银行反馈贷款利率信息的时间基本相同，那么采用异步方式查询利率的速度几乎要快 n 倍，其中 n 是所查询银行的个数。贷款中介必须能以任何顺序接受响应的利率查询消息，因为响应消息到达的顺序不可能与请求发送的顺序完全保持一致。下图说明了异步方式的处理。分叉箭头表示的贷款利率查询请求属于异步调用。



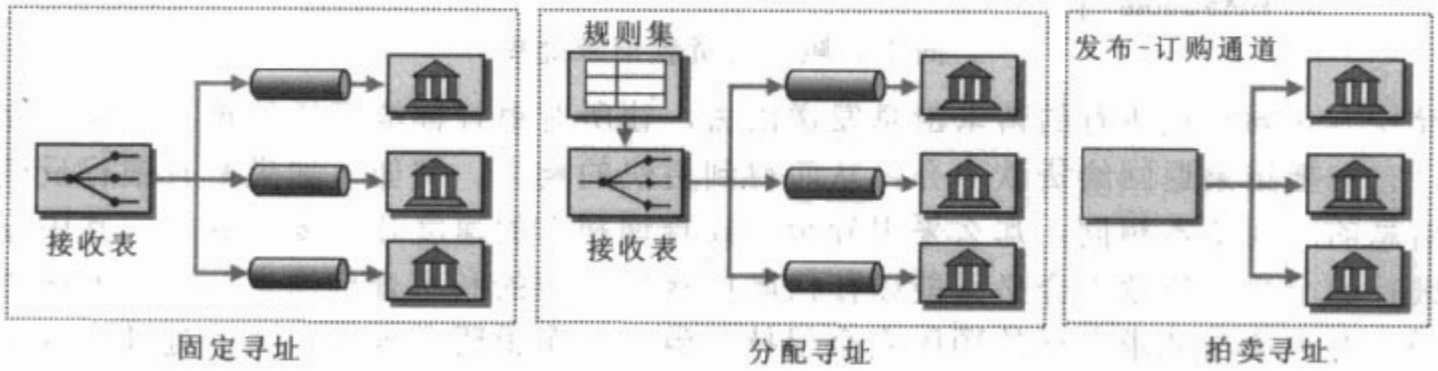
异步、并行的贷款请求处理

利用消息队列实现异步调用还有一个显著的优点，它能创建服务提供者的多个实例。例如，如果信用代理组件是系统处理的瓶颈，我们可以同时运行该组件的两个实例，因为贷款中介不是直接把消息发送给信用代理组件，而是把消息发送到队列中，因此不管消息是由哪个组件实例处理的，只要能把响应消息返回给响应通道就可以。

9.1.4 寻址：分配与拍卖

使用分散收集器查询最佳的利率信息使得我们能选择使用两种不同的寻址机制，即接收表机制或发布-订购通道机制。具体采用哪种机制主要取决于我们对参与特定贷款利率查询处理的银行有多大的控制力度。而且可选的方法有很多：

- **固定寻址**：银行的列表是硬编码的。每个贷款请求消息都发送给一组相同的银行。
- **分配寻址**：贷款中介有一个标准，可以决定哪家银行最适合特定的请求。例如，如果一个客户的信用历史比较差，贷款中介就不会把这个客户的利率查询请求发送给专门为信用良好客户提供服务的银行。
- **拍卖寻址**：贷款中介使用发布-订购通道广播请求消息。任何对消息感兴趣的银行都能订购通道，并对请求消息“竞标”。银行可以根据自己的意愿订购或退订通道。每家银行还可以根据自己的标准决定是否参与竞标。



选择哪种寻址方式才适合于我们的例子？要回答这个问题并不那么简单，应该根据业务和技术方面的偏好和限制条件做出相应的选择。第一种方式很简单，中介可以完全控制银行列表。但是，如果不断有新的银行加入或离开，采用这种方法不但会加重管理负担，而且银行也不愿意接收到大量无关的请求，因为每一个请求都会从一定程度上增加银行的内部开销。为了保持简单性，这种解决方案采用的聚合策略很可能要求所有银行都必须提交响应消息，但是银行应该有权利拒绝做出响应。

如果选择分配寻址方式（使用接收表），中介能更好地控制每个贷款利率查询请求涉及哪家银行。由此，中介可以通过减少请求消息的数量来提高效率。中介还可以根据业务关系挑选指定的银行。其缺点是，在贷款中介组件的内部必须实现和维护额外的业务逻辑。不论是分配寻址方式还是固定寻址方式，都必须为每个参与者提供不同的消息通道，只有这样才能控制消息流。

可以使用发布-订购通道，把贷款利率查询请求广播给所有订购此消息的银行，让每家银行自己决定为哪个请求提供服务。每家银行可以使用消息过滤器，或者实现一个选择性

消费者，过滤掉不想要的贷款请求。这种方法可以极大减轻贷款中介的负担，不必为了增加或删除银行而采取相应的维护操作，但是它要求银行完成更多的工作。这种解决方案只需要一个消息通道，但是必须实现为发布-订购通道。许多高效的发布-订购方案采用了 IP 组播机制，通常无法通过广域网或 Internet 来路由消息。还有其他一些实现方案使用了一个点对点通道阵列和接收表，来模仿发布-订购通道。这种方法保持了发布-订购通道的简单语义，但是通道和网络带宽的使用率较低。如何在路由、发布-订购和过滤等方面做权衡考虑，有关内容请参见“消息过滤器”的介绍。

9.1.5 聚合策略：多通道与单通道

从银行接收到贷款利率信息后，我们要做出类似的设计选择。可以让所有银行把它们的响应消息提交给同一个响应通道，或者为每家银行提供不同的响应通道。使用单通道可减少为每家银行建立不同通道带来的维护负担，但是要求每家银行的应答消息中必须包含能区别出信息来源的字段。如果使用单个的响应通道，除非让接收表把响应消息的个数告诉聚合器（我们称之为初始化聚合器），否则聚合器不会知道响应消息有多少。如果使用拍卖方式的发布-订购通道，贷款中介则不清楚可能有多少响应消息，因此聚合器采用的完成条件必须与总共有多少个参与者无关。例如，聚合器只需等到 3 条响应消息就可完成聚合。但是，采用这种策略也是有风险的，没准只有两家银行参与。因此，当遇到这种情况时，聚合器可能超时，并报告它没有接收到足够数量的响应。

9.1.6 管理并发性

贷款中介之类的服务应当能处理多个客户同时使用服务的情形。例如，假设以 Web 服务的方式来提供贷款中介的功能，或者把它连接到一个公共的 Web 站点，我们将无法控制客户的数量，很可能会同时接收到成百上千个并发请求。使用以下两种不同的策略，贷款中介就可以同时处理多个请求：

- 执行多个实例
- 一个事件驱动实例

第一种策略是同时运行贷款中介组件的多个实例。可以为每个到来的请求消息启动一个实例，或者维护一个活动的贷款中介进程“池”，并把到来的请求消息指派给下一个可用的进程（使用消息分派器）。如果没有进程可用，就让请求消息排队，直到有进程可用为止。进程池的优点是能以可预测的方式分配系统资源。例如，我们可以确定最多执行 20 个贷款中介实例。相反，如果为每个请求消息都启动一个新的进程，当有大量的请求消息同时到达时，就会立即使系统的资源耗尽。另外，通过维护一个运行进程池，我们还能重用已有的进程处理多个请求，这样可以节省进程实例化和初始化所需的时间。

由于贷款中介所做的大量处理都是在等待外部系统（信用机构和银行）的应答，运行多个并行的进程无法充分利用系统的资源。为此，我们可以运行一个进程实例，对到来的消息事件做出响应。处理单个的消息（如银行利率查询）是一件相当简单的任务，因此一个进程可以同时服务于多个请求。采用这种方法能更高效地利用系统资源，并简化系统的

管理，这是因为我们只需监控一个进程实例。其潜在的缺点是，由于只能使用一个进程，它的可扩展性很差。许多大容量应用同时使用了这两种技术，能执行多个并行的进程，每个进程能同时处理多个请求。

要执行多个并发的请求，需要把系统中的各个消息与正确的进程实例相关联。例如，最方便的方法是让银行把所有应答消息都发送到固定的通道；也就是说，应答通道能容纳与不同客户并发利率查询请求相关的消息。因此，每条应答消息中必须携带关联标识符，才能识别出银行是对哪个客户请求做出的响应。

9.1.7 三种实现方案

为了实现这个贷款中介示例，我们要做出三个主要的设计决定，即选择请求消息的时序方案，选择银行的寻址方案，还要定义聚合策略。另外，我们还必须选择编程语言和消息传递基础设施。总之，根据这三个不同的选择，会产生大量可能的实现方案。我们挑出三个有代表性的解决方案加以实现，以强调在不同实现方案之间如何折衷考虑。与本书介绍的其他例子一样，我们并没有特意选择哪些技术，也不会说哪个特定开发商的技术更好。下表归纳了每一种解决方案的特点。

实 现	时 序	寻 址	聚 合	通道类型	产品技术
A	同步	分配	通道	Web 服务/SOAP	Java/Apache Axis
B	异步	分配	关联 ID	消息队列	C#/Microsoft MSMQ
C	异步	拍卖	关联 ID	发布-订购	TIBCO Active-Enterprise

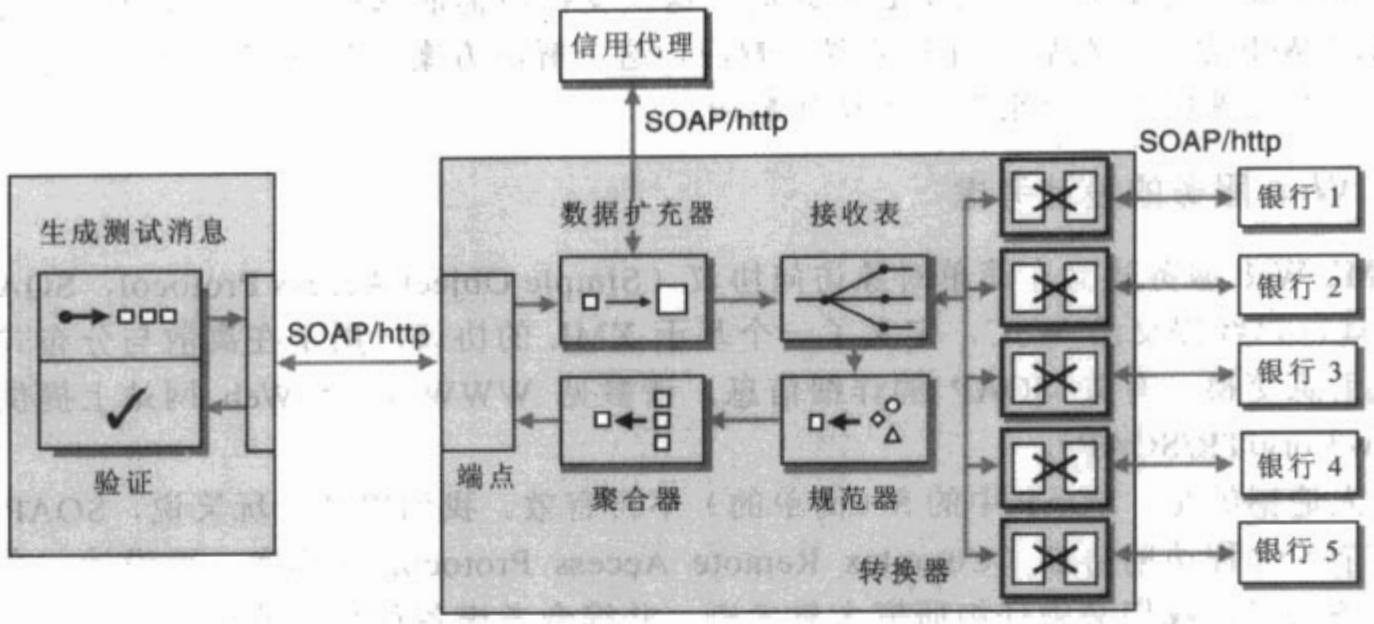
第一种实现方案使用了同步 Web 服务，它采用 Java 和 Apache Axis 实现。与每家银行的通信通过不同的 HTTP 通道进行，这些通道既可以作为请求通道，也可以作为应答通道。因此，聚合策略以不同的应答通道为基础，不需要消息关联。第二种实现方案通过消息队列实现了异步通信。它采用了 Microsoft 的 MSMQ，使用 JMS 或 IBM 的 WebSphere MQ 也能达到类似的效果。最后一种实现方案使用了拍卖方式，并综合运用了 TIBCO 的发布-订购基础设施和 TIB/IntegrationManager 工具，这个工具实现了过程管理器模式。在 B 和 C 这两种方案中，所有的应答消息均返回到一个通道中，并且使用关联标识符将应答消息与客户的贷款利率查询请求相关联。

9.2 使用 Web 服务的同步实现方案

作者：Conrad F. D'Cruz

这一节介绍使用 Java 和 XML Web 服务实现贷款中介的例子，其中 Web 服务采用开源 Apache Axis 工具集实现。由于我们不想把这个例子作为学习 Java 开发的练习，所以这里选择使用工具集，从而降低实现同步 Web 服务接口的难度。本节讨论的重点在于，设计同步消息传递解决方案时要做出哪些决策。

9.2.1 解决方案的架构



Web 服务解决方案的架构

上图给出了采用同步预测型 Web 服务实现贷款中介示例的完整架构。在贷款中介和其他组件之间有 7 个重要的接口。每一对参与者之间使用建立在 HTTP 上的 SOAP 进行通信。

第一个接口是进入贷款中介的入口点，客户端应用通过这个接口把包含贷款申请信息的消息传递给贷款中介。客户端通过相同的接口从贷款中介那里接收查询返回的结果。尽管图中没有显示 Axis 服务器，不过需要知道，Axis 服务器会从客户端接收消息，并实现了服务激励器（Service Activator）。

第二个接口位于贷款中介和信用代理之间。信用代理是一个外部机构，它为贷款中介提供了 Web 服务接口来查询银行所需的额外客户数据。贷款中介对到来的请求消息进行扩充，使之包含信用代理反馈的数据，从而实现了内容扩充器。

其他 5 个接口分别位于贷款中介与五家银行之间。每个接口的作用是从特定银行那里获得贷款利率信息。每家银行的接口提供了相同的功能（查询利率），但是相关的 SOAP 消息采用了不同的格式。因此，在向银行提出利率查询请求前，贷款中介必须把请求消息转换为每家银行所要求的格式。

贷款中介使用预测型路由机制与每家银行直接联系；也就是说，参与利率信息查询的银行在查询操作前就已确定。贷款中介应用使用接收表模式把请求发送给选定的银行。由于每家银行都可以使用不同的请求格式，贷款中介必须使用消息转换器阵列把请求转换为各家银行要求的格式。

同样地，要使用规范器把每条应答消息转换为公共的消息格式，这样才能将所有应答消息聚合为一个最佳的利率查询应答。请求和响应消息采用同步方式传递；也就是说，在每家银行做出响应或者超时之前，客户端和贷款中介将一直阻塞。贷款中介把所有的响应

消息聚合为一条最佳的利率消息，并把它反馈给客户端。

同步消息传递适用于一些简单的解决方案。使用同步消息传递，我们不必考虑异步事件的处理和线程的安全性，也不必考虑提供这些支持所需的基础设施。客户端调用贷款中介提供的 Web 服务，然后等待服务器的响应。这个解决方案中包含多个组件，每个组件都采用同步方式调用下一个组件并等待其响应。

9.2.2 Web 服务的设计考虑

XML Web 服务建立在简单对象访问协议（Simple Object Access Protocol，SOAP）上。SOAP 规范已经提交给 W3C，定义了一个基于 XML 的协议，用于在离散与分布式系统之间实现消息交换。有关 SOAP 的详细信息，请参见 WWW 协会 Web 网站上提供的文档（www.w3.org/TR/SOAP）。

让人遗憾的是，SOAP 中的 S（简单的）不再有效。我们曾经开玩笑说，SOAP 应该改名为复杂的远程访问协议（Complex Remote Access Protocol）。确实，要设计一个可靠的 Web 服务接口，我们必须仔细研究大量术语，并综合考虑各种相关的设计方案。尽管 Web 服务不是本书讨论的重点，但是它相当重要，因此我们有必要对以下设计考虑做简要讨论：

- 传输协议
- 异步消息传递与同步消息传递
- 编码方式（SOAP 编码与 doc/literal）
- 绑定方式（RPC 方式与文档方式）
- 可靠性和安全性

传输协议

建立 SOAP 规范的目的是，让应用采用独立于技术的方式，以同步 RPC 方式调用跨网络的服务。SOAP 规范分别定义了每个 Web 服务调用的请求和响应消息（在描述服务的 WSDL 文档中定义）。尽管 SOAP 的开发采用了消息传递的思想，但是大多数 Web 服务应用都使用 HTTP 作为传输协议。由于 HTTP 是 Web 上最常用的协议，并且能穿越防火墙，所以选择它是很自然的。但是，HTTP 是一种超文本传输协议，目的是让用户能通过 Web 浏览器获取 Internet 上的文档，而不是为了实现应用之间的通信。HTTP 本身是不可靠的，只是设计为用于同步获取文档。客户端应用使用同一个连接向服务器发送请求并取回应答。因此，使用 HTTP 的 Web 服务只能使用同步的请求/响应消息传递，因为建立在不可靠通道上的异步消息传递将毫无意义。

异步消息传递与同步消息传递

在 Web 服务的同步实现中，从请求消息提交给服务器开始，客户端的连接就一直处于打开状态。客户端要等到服务器返回响应消息后才关闭连接。使用同步 RPC 通信的优点是客户端应用能在很短的时间内获知 Web 服务的操作状态（要么接收到响应，要么超时）。使用同步消息传递也有一个严重的不足，由于每个并发的客户端在等待结果时都要保持打开的连接状态，因此服务器可能要同时处理大量的并发连接。这就使服务器应用变得非常复杂。如果对同步服务提供者的某个调用失败，服务器应用必须提供捕获失败的机制，并

且要能恢复并重新路由，从而继续完成调用的处理，或者在继续执行其他同步调用前返回错误标志。

目前，大多数 Web 服务工具集在默认情况下只支持同步消息传递。但是，使用已有的标准和工具，如异步消息排队框架，某些开发商已经为 Web 服务提供了模拟的异步消息传递机制。一些组织、公司和 Web 服务工作组已经认识到提供异步消息传递的需要，并开始制定相关的标准（如 WS-ReliableMessaging）。有关的最新 Web 服务标准请参见 http://www.w3.org/的 WWW 协会站点，以及第 14 章“结束语”部分。

编码方式

对 SOAP 编码的不同观点曾经引发了大量的争论和混乱。SOAP 规范定义了一种名为编码方式（encoding style）的模式（mode），由 encodingStyle 属性指定。该模式有两种取值：编码和文字。编码（encoded）是指把 encodingStyle 属性值设置为 http://schemas.xmlsoap.org/soap/encoding/，文字（literal）是指把这个属性设置为其他值，或者根本不设置属性值。这个模式确定了如何在 XML 中表示应用的对象和参数。编码方式（也被称为 SOAP 编码）可参见 SOAP 规范的第 5 节，其中定义了把程序设计语言的类型映射为 XML 的基本机制。文字方式（也被称为 doc/literal）意味着不必进行映射。相反，类型信息是由外部机制提供的，通常是 WSDL（Web 服务描述语言）文档，其中使用 XML 模式（Schema）来准确定义 SOAP 消息中用到的类型。

之所以会有不同的编码方式，是因为 SOAP 规范是在采纳 W3C XML 模式定义（XML Schema Definition，XSD）规范之前编写的。因此，最初的 SOAP 规范必须提供一种方法对类型信息以及方法调用所需参数进行编码，否则没有其他可行的办法来指定这些信息。针对复杂的数据类型（如数组）才能真正体现出它的作用。SOAP 规范的第 5.4.2 节定义了一种特殊的机制，可以在 XML 中表示程序设计语言中的数组，其中使用了一个特殊的 SOAPEnc:Array 模式类型（schema type）。

但是，由于采纳了 XML 模式（见 http://www.w3.org/TR/xmlschema-0/），大多数语言都可以自行指定 XML 模式到程序设计语言类型的映射（或串行化规则），因此不再需要 SOAP 编码。例如，JAX-RPC 规范特别说明了如何把 Java 类型映射为 XML 模式元素，也可以把 XML 模式元素映射为 Java 类型。这样 XML 中就不再需要额外的编码信息了。因此，SOAP 编码不再像从前那样受到欢迎，并为文字编码所替代。采用文字编码，映射由一个 XML 模式文档在外部指定，通常采用 WSDL 文档形式。

绑定方式

WSDL 规范在 SOAP 绑定中指定了两种不同的绑定方式。绑定方式属性的值有两个：RPC 和 Document。这说明，如果 WSDL 文档指定一个操作的绑定方式属性设置为 RPC，接收者就必须使用 SOAP 规范第 7 节定义的规则解释这个消息。例如，位于 SOAP 消息体中的 XML 元素（被称为包装器元素）必须与所调用的相应操作（用程序设计语言实现）同名。该元素中的各个消息部件必须与这个操作的参数完全对应（名字和出现顺序都一致），并且只能有一个返回元素（该元素名字必须是 XXXResponse，其中 XXX 是相应操作的名字），这个返回元素内部只包含一个元素，这就是操作的返回值。

文档绑定方式的要求要宽松些。文档绑定方式中的消息只需由良定义的 XML 组成。对消息的解释完全取决于接收消息的 SOAP 引擎。我们曾经说过，许多工具（如 Microsoft 提供的工具）通常使用文档绑定方式和文字编码来表示 RPC 语义。即使采用了文档方式，所发送的消息仍然表示一个命令消息，要调用的操作以及所传递的参数均编码放在文档中。

可靠性和安全性

在第 14 章“结束语”中，Sean Neville 介绍了不断改进的一些标准，这些标准的目的就是为了解决 Web 服务的可靠性和安全性问题。

我们的解决方案使用了最基本、可能也是最流行的设计选择组合。贷款中介的具体实现使用了基于 HTTP 的 SOAP 同步通信，并采用默认的 SOAP 编码方式和 RPC 绑定方式实现消息编码，这使得这个 Web 服务的行为非常类似于远程过程调用。采用这种方式，我们不用再讨论 Web 服务的内部细节，而把重点放在同步 Web 服务实现与其他实现的差别上。

9.2.3 Apache Axis

这一节将简要介绍 Axis 体系结构，以便说明设计方案中的一些重要观点。有关 Axis 的其他知识，请参考 http://ws.apache.org/axis 的 Apache Axis 网站。

第 3 章“消息传递系统”把消息端点定义为一种机制，应用采用这种机制来连接消息传递通道从而发送和接收消息。再回到我们的贷款中介应用中，Axis 框架本身表示消息通道，它的主要功能是代表用户应用完成消息的处理。

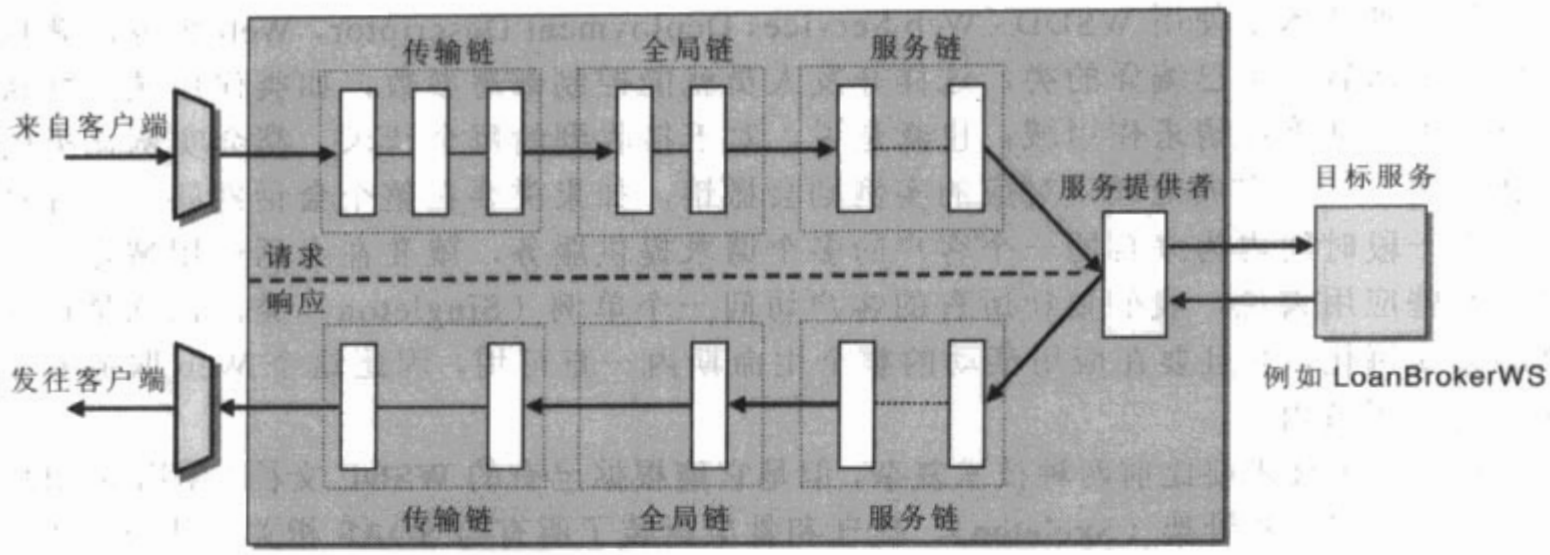
Axis 服务器实现了服务激励器（Service Activator）模式。第 10 章“消息传递端点”将介绍服务激励器如何将消息通道连接到应用中的一个同步服务，从而当消息到达时能调用相应的服务。

Axis 服务器中已经实现了服务激励器，因此开发人员不必再为此费心。应用代码中只需包含应用所需的业务逻辑，Axis 服务器会负责提供所有消息处理服务。

Axis 的客户端编程模型提供了相应的组件，以便客户应用调用一个端点 URL，然后从服务器接收响应消息。在这个应用中，贷款中介客户是一个同步客户，它使用了 Axis 的客户端编程模型。服务器为所支持的各个传输协议提供了一个监听者。当客户端向端点发送一个消息时，Axis 框架中的传输监听者就会创建一个消息上下文对象，并沿着框架中的请求链传递这个对象。该消息上下文对象包含了从客户端接收到的实际消息，以及由传输客户端添加的相关属性。

Axis 框架由一系列处理程序组成，要按照一种特定的顺序调用，这取决于部署配置以及客户端或服务器是否正在调用该框架。处理程序是消息流子系统的组成部分，它们组合到一起形成处理链。请求消息要由链中的一系列请求处理程序来处理。所有的响应消息要沿着响应链，通过一系列响应处理程序返回。

下图给出了 Axis 框架内部的高层表示。可以从 http://ws.apache.org/axis 得到 Axis 体系结构的详细介绍。



Axis 引擎

Axis 由几个子系统组成，它们协作运行，提供了消息通道的功能。该框架既能由客户端应用使用，也能由服务器应用使用。

本例中使用的 Axis 子系统包括：

- 消息模型子系统定义了 SOAP 消息的 XML 语法。
- 消息流子系统定义了传递消息所需的处理程序和链。
- 服务子系统定义了服务处理程序（SOAP、XML-RPC）。
- 传输子系统为消息的传输提供了一些候选方法（如 HTTP、JMS 和 SMTP）。
- 提供者子系统为不同类型的类定义了提供者（如 java:RPC、EJB 和 MDB）。

前面已经提到过，开发人员只需把注意力集中到如何创建实现业务逻辑的应用，然后这个应用可以部署在 Axis 服务器中。要把一个 Java 类部署为 Web 服务，并使之成为一种公开可用的端点服务，可以采用三种技术，它们分别是：

- 自动部署
- 使用 Web 服务部署描述文件
- 根据已有的 WSDL 文档生成代理

第一种、也是最简单的一种方法是，编写一个包含业务逻辑的类，从而建立一个 Java Web 服务（Java Web Service，JWS）文件（这是带*.jws 扩展名的 Java 源文件）。该类的方法包含了业务逻辑。JWS 文件不需要被编译，只要把源文件复制到服务器的 webapps 目录下就能立即完成部署。每个公共方法都能作为 Web 服务来访问。这个 JWS 文件的名字就成为 Axis 服务器所提供的 Web 服务端点的一部分，如下面的 URL 所示：

http://hostname:portnumber/axis/LoanBroker.jws

Axis 1.1 能自动根据服务器内部署的服务生成 WSDL 文档。WSDL 是一种 XML 格式的文档，它描述了 Web 服务的公共接口（如通过接口可调用的方法）以及服务的位置（如 URL）。WSDL 的自动生成使其他应用能够检查由 Web 服务类提供的远程接口。它还可以用于自动生成客户端的桩（stub）类，这些类把 Web 服务调用封装在一个常规的 Java 类中。采用这种技术的缺点是开发人员无法控制部署参数。

第二种技术是使用 WSDD（Web Services Deployment Descriptor，Web 服务部署描述文件）来部署一个已编译的类，这样开发人员就能控制部署参数，如类作用域。默认情况下，类会部署在请求作用域；也就是说，对于接收到的每个请求，都会实例化类的一个新实例。一旦完成处理，对应的实例则会撤销。如果类要在整个会话期间一直保持，从而在一段时间内为来自同一个客户的多个请求提供服务，就要在会话作用域定义类。对于某些应用来说，我们要让所有的客户访问一个单例（Singleton）类，也就是说这个类必须实例化，并且要在应用活动的整个生命期内一直可用，因此这个 Web 服务要定义在应用作用域内。

最后一种技术要比前两种技术复杂，但是它能根据已有的 WSDL 文档（使用 wsdl2java 工具）生成代理和骨架（Skeleton）。代理和骨架封装了所有与 SOAP 相关的代码，因此开发人员不必编写任何与 SOAP 相关（或与 Axis 相关）的代码，同时还能把业务逻辑插入到所生成骨架的方法体中。

我们选择采用自动部署技术和 JWS 文件来实现所有的 Web 服务，以便尽可能地降低设计和部署的需求。在客户端，手工编写 Web 服务的调用代码更加容易。我们本可以使用 wsdl2java 工具生成桩（stub），然后再由客户端的代码调用这些桩，但是我们没有这样做，而是尽可能地减少生成代码量，因为生成代码过多不利于我们完整地介绍示例解决方案。

9.2.4 服务发现

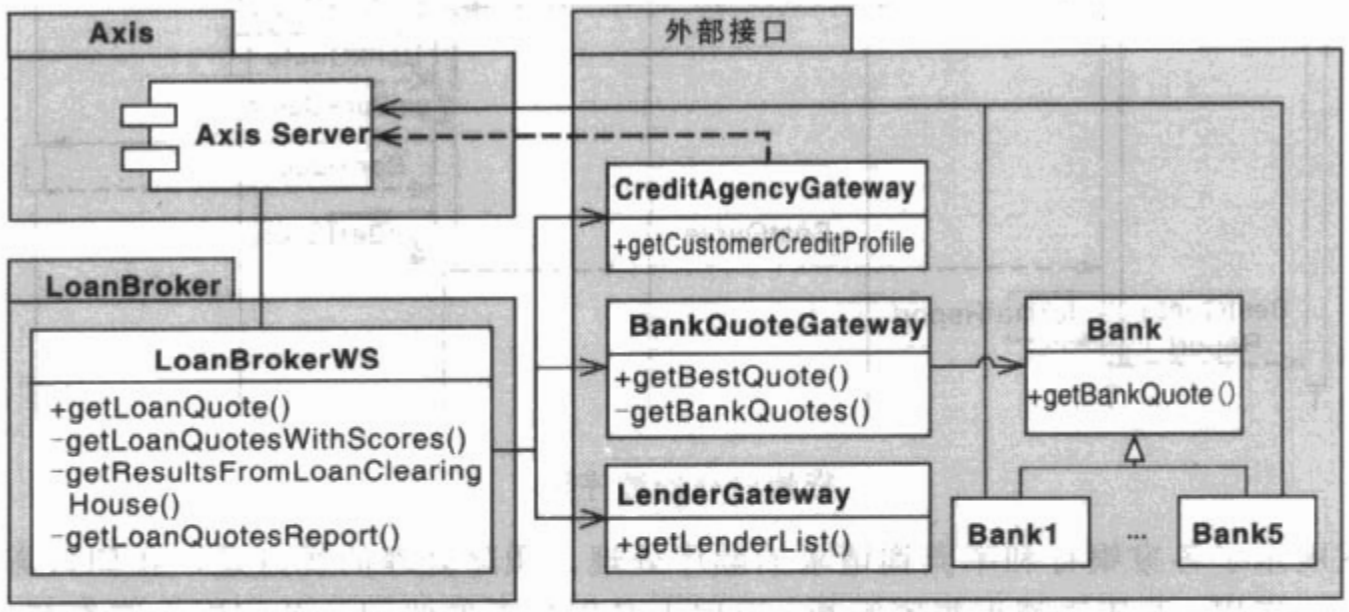
在继续讨论贷款中介应用前，首先要介绍几个一般的实现步骤，为了使创建的应用更易于部署，在实现解决方案时往往会遵循这些步骤。首先，为了调用部署在服务器上的 Web 服务，任何客户端应用都要获知服务端点的 URL。在 Web 服务模型中，应用要从一个公共服务注册中心查找所访问 Web 服务的位置。

UDDI（Universal Description，Discovery，and Integration，统一描述、发现和集成）就是实现这个目的的一种标准。有关 UDDI 的介绍超出了本节的范围，你可以从 http://www.uddi.org 获得更多的信息。在我们的例子中，端点 URL 硬编码在了应用中。但是，为了方便示例代码的部署，我们分别为服务器和客户端应用创建了属性文件。属性文件中有主机名和端口号参数的名值对，它们与 Axis 服务器的安装参数匹配。这就为在具体的环境中部署贷款中介应用提供了一定程度的灵活性。我们在部分 Java 类中提供了一个工具方法 readProps()，该方法能读取包含 Axis 服务器部署参数的文件。贷款中介应用的任何功能组件并不会调用 readProps()方法。

在所有分布式计算框架中，不管是 Java RMI、CORBA 还是 SOAP Web 服务，都需要把远程对象的方法调用参数定义为基本数据类型，或定义为可以在框架之间来回串行化的对象。这些参数是从客户端发往服务器的消息对象的属性。为了简化贷款中介解决方案，我们使用 Java 的 String 对象将贷款中介的响应返回给客户端。如果调用参数采用的是基本数据类型（如 int、double 等），就必须把它们包装在预定义的对象包装器中，这称为类型包装器（如 Integer、Double 等）。

9.2.5 贷款中介应用

下图给出了贷款中介组件的类图。贷款中介的核心业务逻辑封装在 LoanBrokerWS 类中。该类继承自 Axis 框架提供的一个实现了服务激励器的类，当有 SOAP 请求消息到达时，该类就会调用 LoanBrokerWS 类中的一个方法。LoanBrokerWS 引用了一组网关类，这些网关类实现了与外部实体（如信用代理和银行等）交互的具体细节。网关类的实现逻辑封装在 CreditAgencyGateway、LenderGateway 和 BankQuoteGateway 类中。



贷款中介的类图

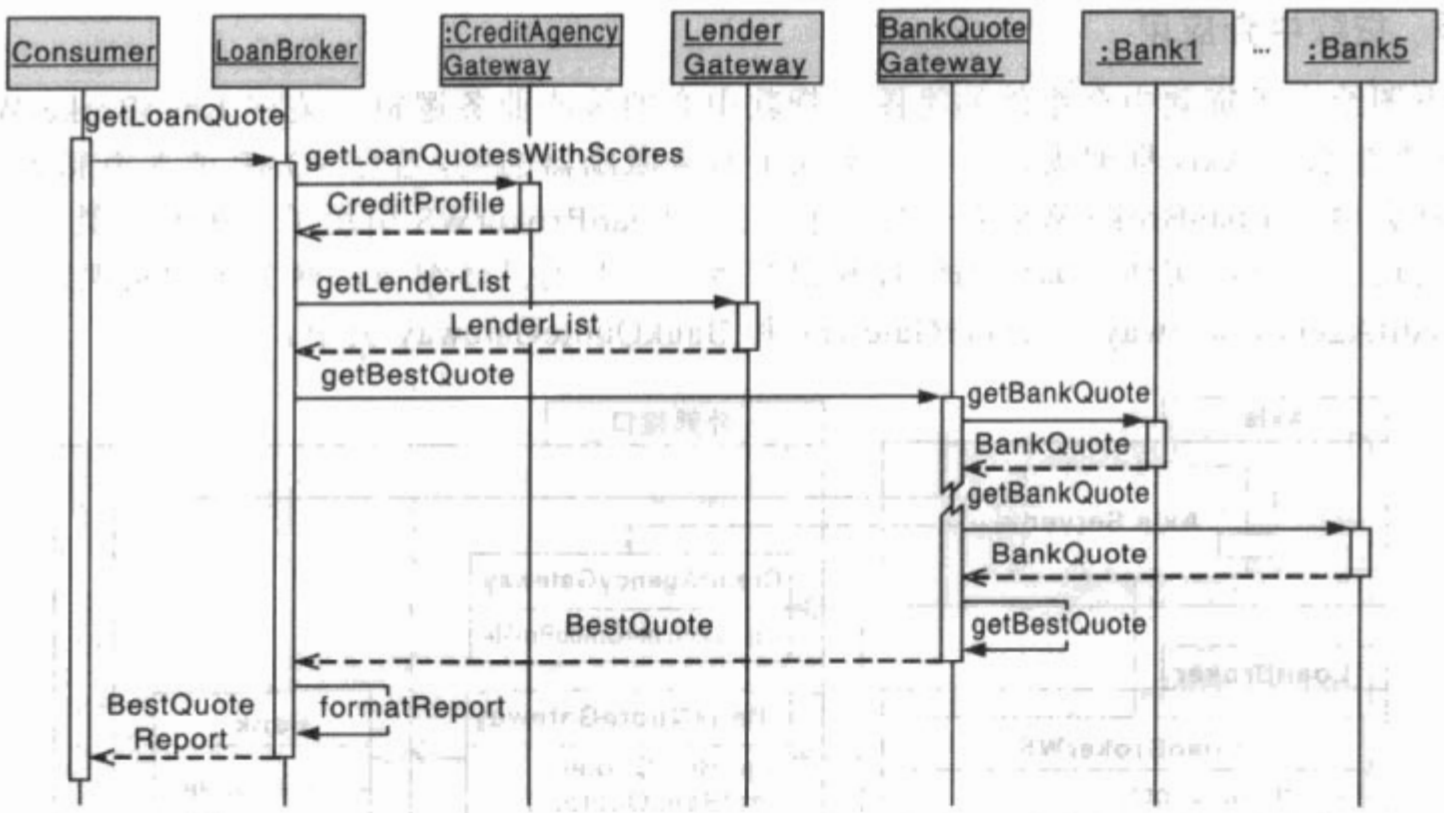
贷款中介提供给外部的惟一服务接口就是让客户访问贷款中介服务的消息端点。这里没有必要定义消息传递网关，因为 Axis 框架代表应用扮演了网关的角色。

下页所示的顺序图展示了同步预测型 Web 服务示例中组件之间的交互过程。贷款中介首先调用信用代理的网关组件（CreditAgencyGateway），该组件只得到了最少数据，它会对此扩充，增加客户的信用度和信用历史时间。贷款中介使用经扩充的数据来调用贷方网关（LenderGateway）。该组件实现了接收表模式，它会利用所提供的全部数据，选择能为贷款请求提供服务的一组银行。

贷款中介接着调用银行利率查询网关（BankQuoteGateway），该组件通过依次调用每个银行组件实现预测型路由操作。银行组件建立了实际银行操作的接口。例如，Bank1 类是访问 Bank1WS.jws Web 服务的接口，它完成了银行操作的建模。有贷款请求到达时，根据所有请求参数生成利率信息之前，必须完成一定的事务工作。在这个例子中，利率信息是由“虚拟”银行 Web 服务生成的。

银行利率查询网关把所有银行返回的响应消息聚合起来，从中选出最佳的利率信息，并把它返回给贷款中介，贷款中介会调整返回数据的格式，并把它返回给客户端。

为了方便显示，我们在顺序图中只给出了五家银行中的两家。根据为特定请求生成的接收表，贷款中介可能只与一家银行联系，也可能与五家银行都联系，让它们为客户端的请求提供服务。



贷款中介的顺序图

该图展示了各家银行利率查询请求的顺序处理。顺序处理的优点是，我们只需在一个线程中运行应用，并依次调用每家银行。应用不必创建新的线程从各家银行获得利率信息，因此不必考虑并发问题。但是，贷款中介在向贷方列表中的下一家银行提出请求前，必须等待上一家银行返回响应消息，因此采用这种方法时获得响应信息的总时间很长。由此导致的最终结果是，客户提交请求后，要等待很长的时间才能得到返回的利率信息。

9.2.6 贷款中介应用的组件

现在我们开始设计贷款中介应用的功能组件。前面已经介绍过，Axis 框架既能支持客户端应用，也能支持服务器应用。这使得远程客户可以通过网络访问发布的端点。服务器应用可能还要作为客户端访问其他的 Web 服务端点，该端点可以在同一服务器上，也可以在远程服务器上。我们把这个解决方案中的关键组件构建为 Web 服务，以此展示这一点（尽管现在这些组件只是运行在同一个服务器实例上）。

贷款中介解决方案必须实现以下功能：

- 接受客户端的请求
- 获取信用代理的数据
- 实现信用代理服务
- 获得利率信息
- 实现银行操作

接受客户端的请求

贷款中介在一个名为 LoanBrokerWS.jws 的 JWS 文件中实现。它提供了一个公共方法

作为 Web 服务：GetLoanQuote。贷款中介需要从客户端获得 3 项数据才能开始处理贷款请求，包括作为客户身份编号的社会安全号（SSN）、贷款的数量和借贷的期限：

```
LoanBrokerWS.jws
public String getLoanQuote(int ssn, double loanamount, int loanduration) {

  String results = "";

  results = results+"Client with ssn="+ssn+"requests a loan of amount="+
                    loanamount+"for"+loanduration+"months"+"\n\n";
  results=results+this.getLoanQuotesWithScores(ssn,loanamount,loanduration);

  return results;
}
```

该方法中只有一个操作步骤，就是调用 getLoanQuotesWithScores 方法。这个方法返回一个字符串，这个字符串再沿着 Axis 框架的响应链返回给客户端。在响应链的最后，原来接收到请求的传输监听者从 Axis 框架接收到响应消息，并通过网络把响应消息返回给客户。

正如前面介绍的，由于我们使用 JWS 文件自动部署贷款中介组件，Axis 会为 LoanBrokerWS 服务生成 WSDL 文件。LoanBrokerWS.jws 文件对应的 WSDL 文件如下所示。

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:message name="getLoanQuoteRequest">
        <wsdl:part name="ssn" type="xsd:int"/>
        <wsdl:part name="loanamount" type="xsd:double"/>
        <wsdl:part name="loanduration" type="xsd:int"/>
    </wsdl:message>
    <wsdl:message name="getLoanQuoteResponse">
        <wsdl:part name="getLoanQuoteReturn" type="xsd:string"/>
    </wsdl:message>
    <wsdl:portType name="LoanBrokerWS">
        <wsdl:operation name="getLoanQuote" parameterOrder="ssn loanamount loanduration">
            <wsdl:input message="intf:getLoanQuoteRequest"
                        name="getLoanQuoteRequest"/>
            <wsdl:output message="intf:getLoanQuoteResponse"
                        name="getLoanQuoteResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="LoanBrokerWSSoapBinding" type="intf:LoanBrokerWS">
        <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap. org/soap/http"/>
```

```
        <wsdl:operation name="getLoanQuote">
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="getLoanQuoteRequest">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                               namespace="..." use="encoded"/>
            </wsdl:input>
            <wsdl:output name="getLoanQuoteResponse">
                <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                               namespace="..." use="encoded"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="LoanBrokerWSService">
      <wsdl:port binding="intf:LoanBrokerWSSoapBinding" name="LoanBrokerWS">
            <wsdlsoap:address location="http://192.168.1.25:8080/axis/LoanBrokerWS.jws"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

为了节省篇幅，我们对 WSDL 文档有所删减，只突出其中最重要的元素。位于文档尾部的<wsdl:service>元素定义了服务名（LoanBrokerWSService）和端点位置。<wsdl:operation>元素定义了方法（操作）名和客户端访问 LoanBrokerWS Web 服务所需的参数。定义在<wsdl:message>标记中的两个消息定义了 getLoanQuote 操作的请求和响应消息：getLoanQuoteRequest 带有传递给 Web 服务的参数，getLoanQuoteResponse 带有从 Web 服务接收到的参数。从<wsdlsoap:binding>元素可以看到，我们使用的正是本章一开始介绍的 RPC 绑定方式，<wsdlsoap:body>说明我们使用了 SOAP 编码方式（而不是 doc/literal 方式）。

若要查看服务器中的 WSDL 文件，可以在浏览器窗口中输入以下 URL，其中 hostname 和 portnumber 要用你安装的服务器的相应参数替换：

http://hostname:portnumber/axis/LoanBrokerWS.jws?wsdl

获取信用代理的数据

贷款中介的另一个需求是搜集客户的其他数据，以完成贷款请求申请。LoanBrokerWS 的第二部分实现了内容扩充器，如下所示。

LoanBrokerWS.jws

```
private String getLoanQuotesWithScores
            (int de_ssn, double de_loanamount, int de_duration) {
  String qws_results =
        "Additional data for customer: credit score and length of credit history\n";
```

```
  int ssn = de_ssn;
  double loanamount = de_loanamount;
  int loanduration = de_duration;
  int credit_score = 0;
  int credit_history_length = 0;

 CreditProfile creditprofile = CreditAgencyGateway.getCustomerCreditProfile (ssn);

  credit_score = creditprofile.getCreditScore();
  credit_history_length = creditprofile.getCreditHistoryLength();

  qws_results = qws_results + "Credit Score= " + credit_score +
                " Credit History Length= " + credit_history_length;
  qws_results = qws_results + "\n\n";
  qws_results = qws_results + "The details of the best quote from all banks that
responded are shown below: \n\n";

  qws_results = qws_results + getResultsFromLoanClearingHouse
              (ssn,loanamount,loanduration,credit_history_length,credit_score);

  qws_results = qws_results + "\n\n";
  return qws_results;
}
```

现在来设计信用代理操作，这是贷款中介应用的另一个逻辑部分。为了使 SOAP 代码不在贷款中介应用中出现，并尽量减少贷款中介与信用代理之间的依赖，我们使用了网关（Gateway）模式[EAA]，它有两个主要优点：第一，可以从应用中抽象出通信的技术细节；第二，如果能让网关接口与网关实现分离，就能用服务桩（Service Stub）[EAA]代替实际的外部服务进行测试。

CreditAgencyGateway 对它与信用代理 Web 服务（CreditAgencyWS）之间通信的技术细节进行了抽象。如果需要的话，可以用测试桩代替 Web 服务。网关接受客户的身份编号（SSN），然后从信用代理获得额外的数据。信用代理返回的两项数据分别是客户的信用度和信用历史时间，它们都是完成贷款申请必需的信息。该网关包含了访问 CreditAgencyWS 的所有客户端代码（CreditAgencyWS 实现在 CreditAgencyWS.jws 文件中）。

CreditAgencyGateway.java

```
public static CreditProfile getCustomerCreditProfile(int ssn){

  int credit_score = 0;
  int credit_history_length = 0;
```

```
CreditProfile creditprofile = null;

try{
  CreditAgencyGateway.readProps();

  creditprofile = new CreditProfile();

     String creditagency_ep = "http://" + hostname + ":" + portnum +
                              "/axis/CreditAgencyWS.jws";

     Integer i1 = new Integer(ssn);

     Service  service = new Service();
     Call     call    = (Call) service.createCall();
     call.setTargetEndpointAddress( new java.net.URL(creditagency_ep) );

     call.setOperationName("getCreditHistoryLength");
     call.addParameter( "op1", XMLType.XSD_INT, ParameterMode.IN );
     call.setReturnType( XMLType.XSD_INT );

     Integer ret1 = (Integer) call.invoke( new Object [] {i1});
     credit_history_length = ret1.intValue();

        call.setOperationName("getCreditScore");

        Integer ret2 = (Integer) call.invoke( new Object [] {i1});
        credit_score = ret2.intValue();

        creditprofile.setCreditScore(credit_score);
        creditprofile.setCreditHistoryLength(credit_history_length);
      Thread.sleep(credit_score);
    }catch(Exception ex){
      System.out.println("Error accessing the CreditAgency Webservice");
    }

    return creditprofile;
  }
}
```

之所以使用网关，是为了说明服务器应用如何利用 Axis 客户端框架访问其他 Web 服务，这个 Web 服务可能位于同一服务器上，也可能在远程服务器上。

实现信用代理服务

信用代理 Web 服务的源代码在 CreditAgencyWS.jws 中，如下所示。为了完成贷款申

请，需要两项重要的数据，分别是客户的信用度和客户的信用历史时间。信用代理记录了每位客户的信用数据和信用历史，利用客户的身份号码就能访问这些数据。

CreditAgencyWS.jws

```
public int getCreditScore(int de_ssn) throws Exception
{
  int credit_score;

  credit_score = (int)(Math.random()*600+300);

  return credit_score;
}

public int getCreditHistoryLength(int de_ssn) throws Exception
{
  int credit_history_length;

  credit_history_length = (int)(Math.random()*19+1);

  return credit_history_length;
}
```

下面的代码给出了 CreditAgencyWS.jws 文件对应的 WSDL 文件。该文件是由 Axis 服务器自动生成的。

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/">
   <wsdl:message name="getCreditScoreResponse">
      <wsdl:part name="getCreditScoreReturn" type="xsd:int"/>
   </wsdl:message>
   <wsdl:message name="getCreditHistoryLengthRequest">
      <wsdl:part name="de_ssn" type="xsd:int"/>
   </wsdl:message>
   <wsdl:message name="getCreditScoreRequest">
      <wsdl:part name="de_ssn" type="xsd:int"/>
   </wsdl:message>
   <wsdl:message name="getCreditHistoryLengthResponse">
      <wsdl:part name="getCreditHistoryLengthReturn" type="xsd:int"/>
   </wsdl:message>
   <wsdl:portType name="CreditAgencyWS">
      <wsdl:operation name="getCreditHistoryLength" parameterOrder= "de_ssn">
         <wsdl:input message="intf:getCreditHistoryLengthRequest"
                     name="getCreditHistoryLengthRequest"/>
         <wsdl:output message="intf:getCreditHistoryLengthResponse"
```

```
                    name="getCreditHistoryLengthResponse"/>
        </wsdl:operation>
        <wsdl:operation name="getCreditScore" parameterOrder="de_ssn">
            <wsdl:input message="intf:getCreditScoreRequest"
                        name="getCreditScoreRequest"/>
            <wsdl:output message="intf:getCreditScoreResponse"
                         name="getCreditScoreResponse"/>
        </wsdl:operation>
    </wsdl:portType>
   <wsdl:binding name="CreditAgencyWSSoapBinding" type="intf:CreditAgencyWS">
        ...
    </wsdl:binding>
    <wsdl:service name="CreditAgencyWSService">
      <wsdl:port binding="intf:CreditAgencyWSSoapBinding" name="CreditAgen cyWS">
          <wsdlsoap:address
           location="http://192.168.1.25:8080/axis/CreditAgencyWS.jws"/>
       </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

<wsdl:service>元素定义了 CreditAgencyWSService，并且向客户（在这个例子中就是贷款中介）提供了能访问的端点。<wsdl:operation>元素定义了 CreditAgencyWS.jws 中包含的两个公共方法：getCreditScore 和 getCreditHistoryLength。针对其中每一个方法，分别定义了一个请求/响应消息对，这些消息位于<wsdl:message>元素中，分别是 getCreditScore-Request、getCreditScoreResponse、getCreditHistoryLengthRequest 和 getCredit HistoryLength-Response。

同样出于节省篇幅的考虑，我们删去了 WSDL 文件中的大多数元素。如果你对整个文件感兴趣，可以使用下面给出的 URL 访问你的服务器，其中 hostname 和 Portnumber 要用所安装服务器的具体参数替换：

http://hostname:portnumber/axis/CreditAgencyWS.jws?wsdl

本例中的信用代理 Web 服务没有提供真正的功能。这里只是用一个测试桩取代真正的服务实现，返回虚拟的数据，供应用的其他部分使用。

.获得利率信息

既然已经扩充了客户的数据，贷款中介应用就可以调用贷款交易中心功能，这是贷款中介本身的一部分。如下所示，贷款交易中心接收贷款应用中的所有数据。

```
getResultsFromLoanClearingHouse(ssn, loanamount, loanduration,
                                credit_history_length, credit_score);
```

如果应用变得更为复杂，或者贷款中介的需求变更为要使用外部贷款交易中心，那么贷款交易中心的功能可以进一步划分为独立的逻辑单元。在我们的例子中，贷款交易中心的功能可以分解为三个步骤：

- 获得能接收客户贷款请求的贷方列表。
- 根据位于贷方列表中的每家银行返回的应答，获得最佳的利率信息。
- 根据最佳的利率信息来格式化数据，并把数据返回给贷款中介。

下面的代码实现了上述三个步骤：

LoanBrokerWS.jws

```
private String getResultsFromLoanClearingHouse(int ssn, double loanamount,
                int loanduration, int credit_history_length, int credit_score) {
   String lch_results="Results from Loan Clearing House ";

   ArrayList lenderlist = LenderGateway.getLenderList
                          (loanamount, credit_history_length, credit_score);

   BankQuote bestquote = BankQuoteGateway.getBestQuote
            (lenderlist,ssn,loanamount,loanduration,credit_history_length,credit_
score);

   lch_results = "Out of a total of " + lenderlist.size() +
                 "quote(s), the best quote is from" +
                  this.getLoanQuotesReport(bestquote);

   return lch_results;
}
```

贷款交易中心的第一项需求是获得能为客户的贷款申请提供服务的贷方列表。为了从贷款中介中抽象出贷方选择功能，我们创建了 LenderGateway 类。通过把这项功能也封装为 Web 服务，这个解决方案会更有意思。不过，设计 Web 服务时要格外小心，而且要考虑到参数和返回类型，参数的类型必须是可串行化的，这样才能通过网络传输。为简单起见，我们把贷方选择逻辑放在一个简单的 Java 类中，并由贷款中介调用。

贷方网关能返回各银行服务端点的集合，这对贷款中介来说最为方便。采用这种方法的缺点是，对银行 Web 服务的认证必须硬编码到贷方网关中。在现实中，银行会不断转卖、收购或合并，甚至比 IT 架构师跳槽的频率还高，因此会使系统维护成为一场恶梦。在本节后面要介绍的 BankQuoteGateway 中，我们将讨论一种健壮的方案来实现银行及其 Web 服务。

下面给出一个 GetLenderList 方法，这个方法将返回能提供贷款请求服务的贷方（即银行）的集合。

LenderGateway.java

```
public static ArrayList getLenderList(double loanamount,
                                       int credit_history_length,
                                       int credit_score){
```

```
  ArrayList lenders = new ArrayList();
  LenderGateway.readProps();

  if ((loanamount >= (double)75000) && (credit_score >= 600) &&
      (credit_history_length >= 8))
  {
    lenders.add(new Bank1(hostname, portnum));
    lenders.add(new Bank2(hostname, portnum));
  }

  if (((loanamount >= (double)10000) && (loanamount <= (double)74999)) &&
      (credit_score >= 400) && (credit_history_length >= 3))
  {
    lenders.add(new Bank3(hostname, portnum));
    lenders.add(new Bank4(hostname, portnum));
  }

  lenders.add(new Bank5(hostname, portnum));

  return lenders;
}
```

该方法实现了接收表模式。在我们的例子中，接收表的规则集由非常简单的 if 语句组成，它们根据预先定义的条件选择一家或多家银行。我们还为每个客户请求设置了默认的选择集，使得每个客户请求至少能得到一条利率信息。

贷款中介把贷方列表传递给银行利率查询网关，让它开始收集利率信息并做出选择。我们创建了另一个网关，这个类名为 BankQuoteGateway，它实现了银行接口内部功能的抽象。贷款中介要做的操作就是从 BankQuoteGateway 请求最佳的利率信息，如以下方法调用所示：

```
BankQuote bestquote = BankQuoteGateway.getBestQuote(lenderlist, ssn, loanamount,
                      loanduration,credit_history_length,credit_score);
```

BankQuoteGateway 从所有银行获得利率信息，并选出最佳的利率（即有最低利率的信息）来响应贷款中介的请求。GetBestQuote 方法如下所示：

BankQuoteGateway.java

```
public static BankQuote getBestQuote(ArrayList lenders, int ssn, double loanamount,
                                      int loanduration,
                                      int credit_history_length,
                                      int credit_score){

  BankQuote lowestquote = null;
```

```
  BankQuote currentquote = null;

  ArrayList bankquotes=BankQuoteGateway.getBankQuotes(lenders,ssn,loanamount,
                       loanduration, credit_history_length, credit_score);

  Iterator allquotes = bankquotes.iterator();

  while (allquotes.hasNext()){
    if (lowestquote == null){
      lowestquote = (BankQuote)allquotes.next();
    }
    else{
      currentquote = (BankQuote)allquotes.next();
      if (currentquote.getInterestRate() < lowestquote.getInterestRate()){
        lowestquote = currentquote;
      }
    }
  }
  return lowestquote;
}
```

前面的代码中最重要的一行是调用 **getBankQuotes**。这个方法不但完成了受控拍卖功能，还实现了聚合模式。下面给出 **getBankQuotes** 方法：

```
public static ArrayList getBankQuotes(ArrayList lenders,int ssn,double loanamount,
                                       int loanduration,
                                       int credit_history_length,
                                       int credit_score) {

   ArrayList bankquotes = new ArrayList();
   BankQuote bankquote = null;
   Bank bank = null;

   Iterator banklist = lenders.iterator();

   while (banklist.hasNext()){
     bank = (Bank)banklist.next();
     bankquote = bank.getBankQuote(ssn, loanamount, loanduration,
                                   credit_history_length, credit_score);
     bankquotes.add(bankquote);
   }
  return bankquotes;
}
```

受控拍卖的功能由 while 循环实现。它从贷方列表中抽取出每家银行，调用 getBankQuote 方法来生成银行利率信息，如以下代码所示。尤其要注意调用该方法时参数表的顺序，设计银行及相关的 Web 服务时我们再解释其重要性。

```
bank.getBankQuote(ssn, loanamount, loanduration, credit_history_length, credit_
score);
```

返回的应答使用 bankquotes ArrayList 来聚合。getBestQuote 方法会迭代处理银行利率信息集合，找出最低的利率，并将其返回给贷款中介。

前面提到过，我们要设计银行和银行 Web 服务来模仿真实的银行操作，从而把银行的功能分离出来，与贷款中介的功能解耦合。这样，我们的银行类就能利用前面在 CreditAgencyGateway 类中描述的网关模式。

Bank 抽象类定义如下：

Bank.java

```
public abstract class Bank {

  String bankname;
  String endpoint = "";
  double prime_rate;

 public Bank(String hostname, String portnum){
   this.bankname = "";
   this.prime_rate = 3.5;
 }

 public void setEndPoint(String endpt){this.endpoint = endpt;}
 public String getBankName(){return this.bankname;}
 public String getEndPoint(){return this.endpoint;}
 public double getPrimeRate(){return this.prime_rate;}
 public abstract BankQuote getBankQuote(int ssn,double loanamount,int loanduration,
                                        int credit_history_length, int credit_ score);


 public void arbitraryWait(){
   try{
     Thread.sleep((int)(Math.random()*10)*100);


   }catch(java.lang.InterruptedException intex){
```

```
      intex.printStackTrace();
    }
  }
}
```

在我们的例子中，从银行获取利率信息的过程大致按真实的银行操作建模。首先，要完成少量事务工作，然后访问计算机实现的利率查询系统，在把利率信息返回给BankQuoteGateway前还要做一些事务性工作。

抽象类Bank和它的银行子类（Bank1到Bank5）是对正常银行操作的建模。在我们的例子中，银行接收贷款请求，营业员认真检查客户的信息是否正确。我们把营业员所做的事务工作建模为一个故意的wait方法，在抽象的Bank类中实现（如上所示），并在getBankQuote方法中调用。为了更生动一些，我们采用Web服务对银行利率查询系统建模，以获得利率信息。对指定的银行（Bank n）来说，利率查询系统使用BanknWS建模，它的源代码保存在名为BanknWS.jws的文件中。例子中包含五个银行类（Bank1到Bank5）和五个利率查询系统（Bank1WS到Bank5WS）。每个利率查询系统对方法调用中的参数表都使用了不同的格式，就像现实生活中不同的银行很可能使用不同的数据格式。这意味着，Bank类在调用Web服务前必须使用消息转换器转换消息的格式。在讨论完Bank类之后，我们将介绍如何转换消息格式。

注意，在抽象的Bank类中，getBankQuote方法是一个抽象方法，其参数有一种特定的顺序。下面来看一个银行的具体实现，我们选择了Bank1，选择它并没有什么特殊的理由。所有银行类的结构都是相同的，各个类中只是字段（银行名称和端点地址）的值不同，这些字段的值在构造银行对象时设置。

Bank1.java

```
public class Bank1 extends Bank {

  public Bank1(String hostname, String portnum){
    super(hostname,portnum);
    bankname = "Exclusive Country Club Bankers\n";
    String ep1 = "http://" + hostname + ":" + portnum + "/axis/Bank1WS.jws";
    this.setEndPoint(ep1);
  }

  public void setEndPoint(String endpt){this.endpoint = endpt;}

  public String getBankName(){return this.bankname;}
  public String getEndPoint(){return this.endpoint;}

  public BankQuote getBankQuote(int ssn, double loanamount, int loandura tion,
                                int credit_history_length, int credit_score) {
```

```
BankQuote bankquote = new BankQuote();

Integer i1 = new Integer(ssn);
Double i2 = new Double(prime_rate);
Double i3 = new Double(loanamount);
Integer i4 = new Integer(loanduration);
Integer i5 = new Integer(credit_history_length);
Integer i6 = new Integer(credit_score);

try{
  Service service = new Service();
  Call call = (Call) service.createCall();
  call.setTargetEndpointAddress( new java.net.URL(endpoint) );

  call.setOperationName("getQuote");

  call.addParameter( "op1", XMLType.XSD_INT, ParameterMode.IN );
  call.addParameter( "op2", XMLType.XSD_DOUBLE, ParameterMode.IN );
  call.addParameter( "op3", XMLType.XSD_DOUBLE, ParameterMode.IN );
  call.addParameter( "op4", XMLType.XSD_INT, ParameterMode.IN );
  call.addParameter( "op5", XMLType.XSD_INT, ParameterMode.IN );
  call.addParameter( "op6", XMLType.XSD_INT, ParameterMode.IN );

  call.setReturnType( XMLType.XSD_DOUBLE);

  Double interestrate = (Double) call.invoke( new Object [] {i1,i2,i3,i4,i5,i6});

  bankquote.setBankName(bankname);
  bankquote.setInterestRate(interestrate.doubleValue());
 }catch(Exception ex){
  System.err.println("Error accessing the axis webservice from " + bankname);

  BankQuote badbq = new BankQuote();
  badbq.setBankName("ERROR in WS");
  return badbq;
 }

 arbitraryWait();

 return bankquote;
}

}
```

从前面的代码可以看到，它实现了 getBankQuote 方法，并且该方法的参数有特定的顺序。

```
public BankQuote getBankQuote(int ssn, double loanamount, int loanduration,
                              int credit_history_length, int credit_score)
```

实现银行操作

前面介绍过，每家银行提供的利率查询系统的参数有不同的格式。也就是说，Bank 类的 getBankQuote 方法实现了消息转换器模式，而且在调用各个银行的 Web 服务之前要转换参数的顺序。每个银行 Web 服务中 getQuote 方法的签名如下所示。

Bank1WS:

```
getQuote(int ssn, double prime_rate, double loanamount, int loanduration,
         int credit_history_length, int credit_score)
```

Bank2WS:

```
getQuote(double prime_rate, double loanamount, int loanduration,
         int credit_history_length, int credit_score, int ssn)
```

Bank3WS:

```
getQuote(double loanamount, int loanduration, int credit_history_length,
         int credit_score, int ssn, double prime_rate)
```

Bank4WS:

```
getQuote(int loanduration, int credit_history_length, int credit_score, int ssn,
         double prime_rate, double loanamount)
```

Bank5WS:

```
getQuote(int credit_history_length, int credit_score, int ssn, double prime_rate,
         double loanamount, int loanduration)
```

getQuote 方法的实际实现只是一个占位符，它使用一个简单的算法返回利率信息，以下面给出的 Bank1WS.jws 为例。

Bank1WS.jws

```
public class Bank1WS {
```

```
  public double getQuote(int ssn, double prime_rate, double loanamount,
                         int loanduration, int credit_history_length, int credit_score)
  {
    double ratepremium = 1.5;

    double int_rate = prime_rate + ratepremium + (double)(loanduration/12)/ 10 +
                      (double)(Math.random()*10)/10;

    return int_rate;
  }

}
```

在实际的应用中，可能采用更详细、更复杂的计算公式。getQuote 方法的返回类型是一个双精度的数字，表示银行根据贷款申请给定的参数向客户提供的贷款利率。

Axis 服务器同样能为每个银行 JWS 文件自动创建 WSDL 文件（见 http://hostname:portnum/axis/Bank1WS.jws?wsdl）。其他银行的 WSDL 文件有类似的格式，只是在参数的定义方面有所不同。Bank1WS.jws Web 服务的 WSDL 文件如下所示。

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/">
  <wsdl:message name="getQuoteRequest">
     <wsdl:part name="ssn" type="xsd:int"/>
     <wsdl:part name="prime_rate" type="xsd:double"/>
     <wsdl:part name="loanamount" type="xsd:double"/>
     <wsdl:part name="loanduration" type="xsd:int"/>
     <wsdl:part name="credit_history_length" type="xsd:int"/>
     <wsdl:part name="credit_score" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="getQuoteResponse">
     <wsdl:part name="getQuoteReturn" type="xsd:double"/>
  </wsdl:message>
  <wsdl:portType name="Bank1WS">
     <wsdl:operation name="getQuote" parameterOrder="ssn prime_rate loanamount loanduration
       credit_history_length credit_score">
      <wsdl:input message="intf:getQuoteRequest" name="getQuoteRequest"/>
      <wsdl:output message="intf:getQuoteResponse" name="getQuoteResponse"/>
     </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="Bank1WSSoapBinding" type="intf:Bank1WS">
```

```
        ...
    </wsdl:binding>
    <wsdl:service name="Bank1WSService">
        <wsdl:port binding="intf:Bank1WSSoapBinding" name="Bank1WS">
            <wsdlsoap:address location="http://192.168.1.25:8080/axis/Bank1WS.jws"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

可以看到，WSDL 包含一个 getQuote 操作，定义在<wsdl:operation>元素中，Axis 把它映射到 Bank1WS.jws 类中的 getQuote 方法。<wsdl:service>元素定义了 Web 服务 Bank1WS。其中还包括一个请求/响应消息对：getQuoteRequest 和 getQuoteResponse，每个消息都定义在一个<wsdl:message>元素中。

每家银行的利率信息设置在一个银行利率 bean（BankQuote）中，这个 bean 添加到发回给 BankQuoteGateway 的集合中。该 bean 不要求任何格式化操作，而且由各家银行返回的 bean 看起来都是一样的。这样就没有必要使用规范器把应答消息转换为公共的格式。

BankQuoteGateway 从它所得到的银行利率集合中选出最低的利率，然后把一个银行利率 bean 返回给贷款中介。贷款中介访问 bean 中的数据，并且生成一个报告返回给客户端应用。产生报告的方法如下所示。

BankQuoteGateway.java

```
private static String getLoanQuotesReport(BankQuote bestquote){
    String bankname = bestquote.getBankName();
    double bestrate = ((double)((long)(bestquote.getInterestRate()* 1000))/(double)1000);

    String results = "\nBank Name: " + bankname + "Interest Rate: " + bestrate;

    return results;
}
```

9.2.7 客户端应用

客户端应用是客户与贷款中介应用的接口。客户端的主要需求是，在一个提供功能而且有足够错误检查的用户界面环境中收集客户的信息。在底层（这是用户看不到的），客户端应用要准备好三项数据传送给服务器应用端点。为了简单起见，我们把客户端应用设计为带有 main 方法的 Java 类，它以命令行的方式接受客户的信息。真正的用户界面可以实现为基于窗口的“胖”客户端或基于浏览器的“瘦”客户端。客户端也可能是另一个业务

系统（其中部署有客户端编程模型）的一部分。客户端应用中，与调用贷款中介 Web 服务相关的最重要的部分如下所示：

```
Service service = new Service();
Call    call    = (Call) service.createCall();

call.setTargetEndpointAddress( new java.net.URL(endpoint) );
call.setOperationName( "getLoanQuote" );
call.addParameter( "op1", XMLType.XSD_INT, ParameterMode.IN );
call.addParameter( "op2", XMLType.XSD_DOUBLE, ParameterMode.IN );
call.addParameter( "op3", XMLType.XSD_INT, ParameterMode.IN );

call.setReturnType( XMLType.XSD_STRING );

String ret = (String) call.invoke( new Object [] {ssn, loanamount, loandura- tion});
```

这段代码中最关键的是定义方法名（getLoanQuotes）和设置参数及返回类型的那几行代码。getLoanQuotes 方法以客户 ID、贷款数量和贷款期限为输入参数，返回值是一个字符串。

我们没有使用 UDDI，即查找 Web 服务的服务，而是把所使用 Web 服务端点的 URL 硬编码。我们选择把贷款中介应用部署为 JWS 文件，使端点有标准格式（为 JWS 定义的 Axis API）。下面给出了所部署端点的 URL，其中 hostname 和 portnumber 的值要替换为所安装服务器的相应值。

http://hostname:portnumber/axis/LoanBroker.jws

客户端应用在等待响应期间一直阻塞，有响应时，将接收格式化后的报告，并把报告显示在相应的图形用户界面中。也可以把报告保存起来，或发送给打印机。

9.2.8 运行解决方案

假设你已经安装了 Axis 服务器和贷款中介应用。现在要重新启动服务器，如果服务器没有运行则需启动。按照 Tomcat 给出的帮助文件正确地启动或重启服务器。然后用以下步骤检查 Tomcat 和 Apache Axis 是否已经启动并运行：

在客户端的计算机上打开一个 Shell 或命令窗口，运行应用，下面给出了在 UNIX/Linux 或 Microsoft Windows 上运行应用的方法：

```
java -classpath %CLASSPATH% LoanQueryClient [customerid] [loanamount] [loanduration in months]
```

或

```
java -classpath $CLASSPATH LoanQueryClient [customerid] [loanamount] [loanduration in months]
```

例如：

```
java -classpath %CLASSPATH% LoanQueryClient 199 100000.00 29
```

它将调用贷款中介 Web 服务并返回以下结果：

```
Calling the LoanBroker webservice at 1053292919270 ticks
LoanBroker service replied at 1053292925860 ticks

Total time to run the query = 6590 milliseconds

The following reply was received from the Loan Clearing House

Client with ssn= 199 requests a loan of amount= 100000.0 for 29 months

Additional data for customer: credit score and length of credit history
Credit Score= 756 Credit History Length= 12

The details of the best quote from all banks that responded are shown below:

Out of a total of 3 quote(s), the best quote is from
Bank Name: Exclusive Country Club Bankers
Interest Rate: 6.197
```

在运行客户端时，可以输入不同的贷款数量来测试贷款中介应用。下面我们首先分析运行单个客户端的输出结果，然后再同时运行多个客户端。

分析输出

当客户端应用调用服务器上的贷款中介 Web 服务端点时，它要跟踪记录时间。当服务器返回响应结果时，客户端也要记录时间。客户端应用会输出调用 Web 服务的开始和结束时间，以及它们之间的时间差。

```
Calling the LoanBroker webservice at 1053292919270 ticks
LoanBroker service replied at 1053292925860 ticks

Total time to run the query = 6590 milliseconds
```

对于客户端应用通过网络发送的客户请求，贷款交易中心 Web 服务会显示出所有相关的详细信息。

```
Client with ssn= 199 requests a loan of amount= 100000.0 for 29 months
```

贷款交易中心还显示出为支持客户请求而收集的其他信用数据。

```
Additional data for customer: credit score and length of credit history
Credit Score= 756 Credit History Length= 12
```

LoanBroker 分析数据并选出一组能满足客户贷款请求标准的银行。LoanBroker 把客户的数据按照每家银行要求的格式分别提交，然后等待各家银行的响应。由于 LoanBroker 是一个同步应用，它将一直等待银行返回响应，除非银行响应请求超时或失败。

LoanBroker 收集所有的响应消息，分析返回的利率，并选出最佳的利率。该利率信息和其他相关的数据将格式化，并返回给客户。下面是从各家银行返回的消息中得到的最佳利率信息：

```
Out of a total of 3 quote(s), the best quote is from
Bank Name: Exclusive Country Club Bankers
Interest Rate: 6.197
```

LoanBroker 表示有三家银行做出响应，然后它选出最佳的利率，并把结果提供给用户。

9.2.9 性能限制

在前面介绍顺序图时提到过，由于贷款中介在把请求提交给贷方列表中的下一家银行前，必须等待上一家银行的响应，因此获得利率查询响应的总时间是很长的。客户在提交请求后，不得不等待很长的时间才能得到返回的结果。首先多次运行基准测试，测试单个客户端向服务器提出请求和获得响应的操作，可以得到总运行时间的平均值(大约 8 秒钟)。然后分别在同一个客户端计算机上的 4 个不同窗口中运行 4 个客户端实例，得到的平均时间如下：

```
Client 1: 12520 milliseconds
Client 2: 12580 milliseconds
Client 3: 15710 milliseconds
Client 4: 13760 milliseconds
```

尽管这些测试谈不上科学，但是这些实验数据说明，如果同时有多个客户端访问我们的贷款中介系统，会使性能大幅下降。

9.2.10 此例的局限性

为了更方便地讨论贷款中介示例的设计，所有的 Web 服务都实现为 JWS 文件。采用这种方法部署服务，只需把文件复制到服务器中即可。但是它的缺点是，针对每个请求都要初始化一个新的服务类实例，并且完成请求操作后还要释放该实例。我们看到存在一定的时间延迟，这就是由于服务器在调用服务前要创建类实例造成的。

我们也可以选择更复杂的路由技术，并设计一个能采用 WSDD 文件进行部署的 Java 类。这样就为设置类实例的持续时间提供了灵活性，可以设置为整个客户会话期间，也可以设置为整个应用运行期间。当设计一个实际应用时，如何部署 Web 服务是一个非常重要的问题。假如我们要讨论这个问题，要介绍的内容就会很多，从本章的目的来看也没有必要介绍这些设计细节。

9.2.11 小结

这一节我们使用同步的 SOAP/HTTP Web 服务按步骤实现了贷款中介应用。这里采用预测型路由把贷款请求提交给一组银行，并且强调了采用这种方法的优缺点。为了限制讨论的范围，我们对设计方案做了一些折衷调整，而没有用过多的篇幅介绍部署的细节。这样做的目的是为了探讨使用这种方法的优点和不足。我们还看到了如何使用本书介绍的许多模式，这将有助于我们在其他业务领域中使用同步预测型消息传递方法。

9.3 使用 MSMQ 的异步实现方案

本节介绍如何使用 Microsoft 的.NET、C#和 MSMQ 实现贷款中介示例（见本章的引言）。Microsoft .NET 框架包含了 System.Messaging 名字空间，它为.NET 程序访问 Microsoft 消息排队服务（Microsoft Message Queuing Service）提供了支持，最新的 Windows 操作系统（Windows 2000、Windows XP 和 Windows Server 2003）中都包含有 Microsoft 消息排队服务。这个例子中采用了许多设计决策，并且尽可能给出了实现解决方案所需的实际代码。我们把重点集中在解决方案的设计方面，因此即使读者不是专业的 C#开发人员，这个例子也会很有帮助。事实上，除了与 System.Messaging 打交道的接口外，应用的绝大部分实现与采用 Java 和 JMS 的实现是非常相似的。

这个解决方案中演示的部分功能如果使用集成和协作工具（如 Microsoft BizTalk Server）来实现，需要的代码更少。之所以没有使用这样的工具，主要有两个原因。第一，这些工具不是免费的，运行这种简单的示例也必须获得许可。第二，我们想把所有必要功能的实现全部演示出来。

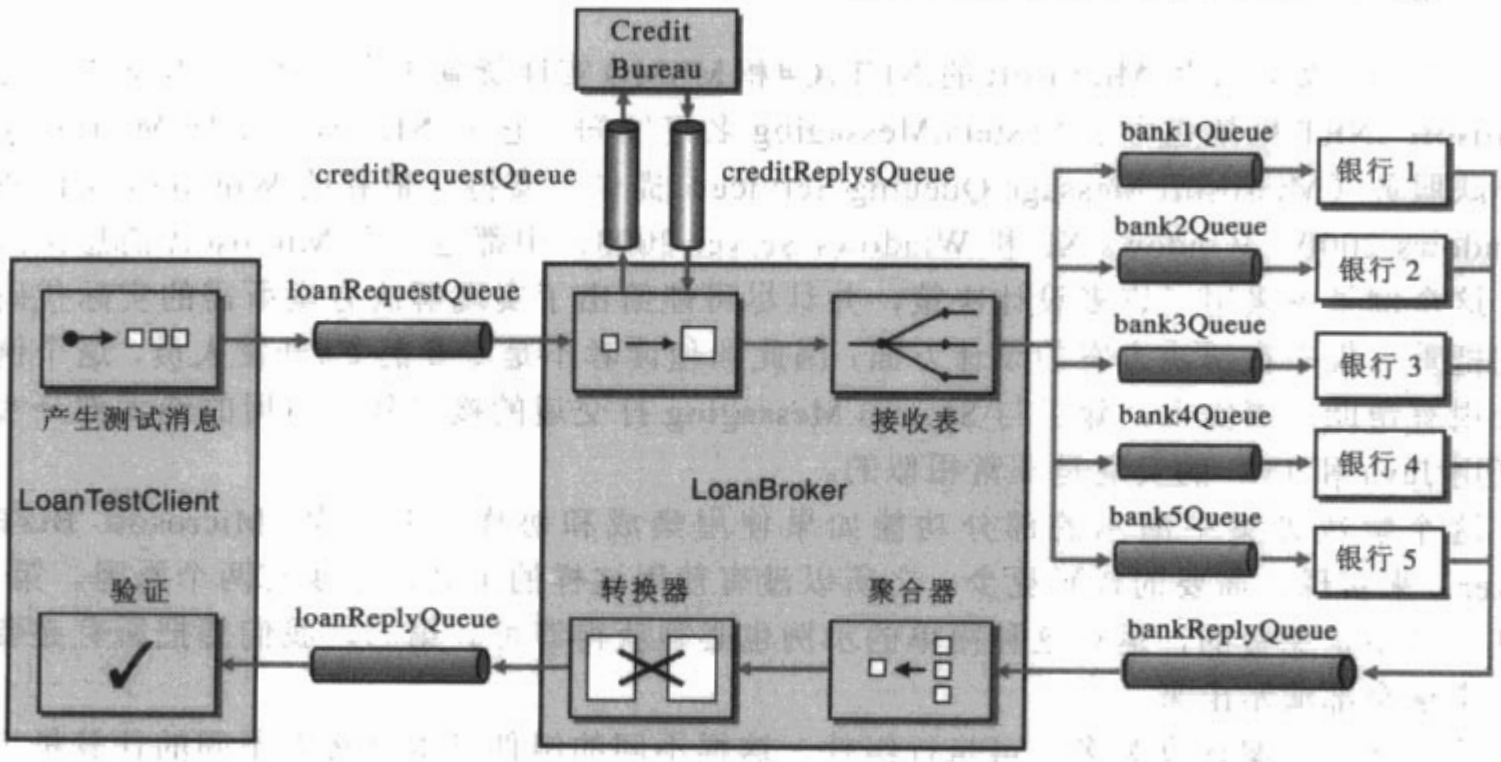
这个解决方案建立为多个可执行组件，这样不同的组件可以分布在不同的计算机上。作为一个例子，同时也为了简化安装需求，避免安装活动目录（Active Directory），我们使用了局部的、私有的消息。因此，目前的解决方案必须运行在同一台计算机上。

贷款中介示例的实现使用了基于消息队列的异步消息传递。正如在引言中所介绍的，采用这种方法，我们能同时处理多个利率查询请求，但是在贷款请求消息流经系统并最终生成相应响应消息的过程中，必须把消息关联起来。介绍这个例子的过程中，我们所做的许多设计决策都是为了满足异步处理的需要。

9.3.1 贷款中介集成系统

从外到内地理解贷款中介的设计是一个很好的主意。我们首先来分析贷款中介提供的所有外部接口（如下图所示）。由于消息队列是单向的，为了建立与其他组件通信的请求/响应通道，必须采用一对队列（第 6 章“简单的消息传递”给出了一个简单的例子）。因此，贷款中介要从 loanRequestQueue 接收贷款利率查询请求，并通过 loanReplyQueue 向测试客户端返回应答。与信用机构的交互采用了类似的队列对。我们没有为每家银行创建不同的

队列对，而是让所有银行都把应答发送到同样的 bankReplyQueue 中。接收表把请求消息发送给每个单独的银行队列中，聚合器从到达 loanReplyQueue 的应答消息中选出最佳的利率。接收表和聚合器共同扮演了分布式的分散收集器。为了简单起见，本例中所有的银行使用了相同的消息格式，因此不需要使用规范器。但是，由于公共的银行消息格式与消费者期望的消息格式不同，我们仍然要使用一个消息转换器把银行返回的应答消息转换为贷款中介应答消息。我们决定把贷款中介设计为一个过程管理器。这个过程管理器的内部功能不是由不同的组件（通过消息队列分隔）实现，而是实现为一个完整的组件，它能实现所有的功能。采用这种方法，可以消除在这些功能之间通过队列传送消息所带来的开销，但是这要求贷款中介能维护多个并发的过程实例。

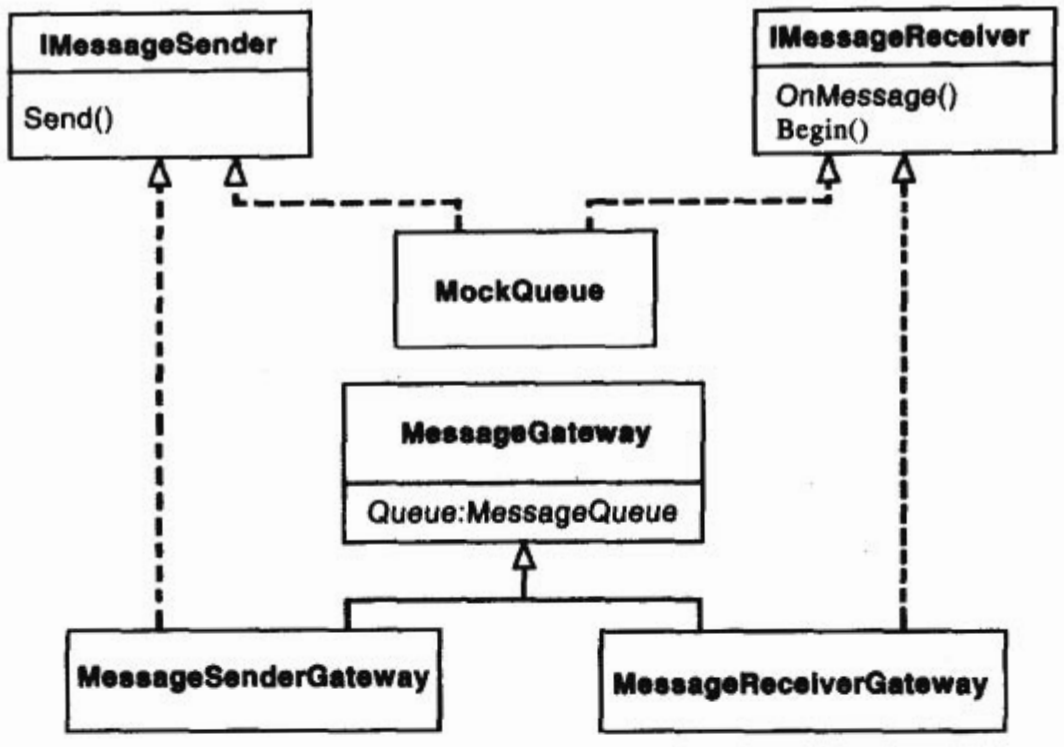


有消息队列接口的贷款中介

9.3.2 奠定基础：消息传递网关

本节并不是对 System.Messaging 名字空间和 MSMQ 的介绍。因此，很有必要把与 MSMQ 相关的功能集中到一个单独的类中，使应用代码与 MSMQ 相关命令分离。基于此目的，我们使用了网关[EAA]模式，它有两个重要的优点：第一，可以从应用中抽象出通信的技术细节；第二，如果能让网关接口与网关实现分离，就能用服务桩[EAA]代替实际的外部服务进行测试。

在我们的例子中，消息传递网关中定义了两个接口：IMessageSender 和 IMessageReceiver。这两个接口非常简单。IMessageSender 所做的操作就是发送消息，IMessageReceiver 所做的操作只是接收消息。另外，接收者用 Begin 方法告诉应用系统它已经准备好接收消息。采用如此简单的接口有利于定义实现接口的类。



网关有助于把 MSMQ 的细节从应用中分离出来并提高可测试性

IMessageSender.cs

```
namespace MessageGateway
{
    using System.Messaging;

    public interface IMessageSender
    {
        void Send(Message mess);
    }
}
```

IMessageReceiver.cs

```
namespace MessageGateway
{
    using System.Messaging;

    public interface IMessageReceiver
    {
        OnMsgEvent OnMessage
        {
            get;
            set;
        }
```

```
        void Begin();

        MessageQueue GetQueue();
    }
}
```

具体实现是 MessageSenderGateway 和 MessageReceiverGateway 类。这些类负责配置消息队列的属性，如 MessageReadPropertyFilter 或 Formatter 的设置。MessageReceiver-Gateway 使用模板方法[GoF]响应 MSMQ 消息队列的 ReceiveCompleted 事件，以实现为数不多但是很重要的操作，例如在完成消息处理后调用 mq.BeginReceive 方法。有关特性的技术细节我们不做介绍，可参考 MSDN 的在线帮助文档[MSMQ]。

由于我们定义的接口很窄，甚至不使用消息队列也能实现前面的解决方案。MockQueue 实现了这两个接口，但是它没有引用任何消息队列！应用发送一个消息时，MockQueue 立即用同样的消息触发 OnMessage 事件。这样一来，在同一地址空间中测试应用就会变得更加简单，而不用考虑异步方面的问题（后面会更多地讨论测试）。

对于那些 C#的初学者来说，有必要为之解释一下 IMessageReceiver 中的 OnMsgEvent OnMessage。.NET 框架为观察者模式[GoF]提供了相应的语言特性，称为委托（delegate）和事件(event)。OnMsgEvent 是 MessageReceiverGateway 中定义的委托：

```
public delegate void OnMsgEvent(Message msg);
```

通过委托，对象能注册特定类型的事件。当事件触发时，.NET 会调用所注册的所有方法。可以采用多种方式调用委托，但是最简单的方式是直接使用委托名来调用：

```
OnMsgEvent receiver;
Message message;
...
receiver(message);
```

如果你有兴趣更多地了解委托，可以查看一本介绍.NET 或 C#的好书。如果你想了解公共语言运行时（Common Language Runtime，CLR）实现委托的详细内容，可参考[Box]。

9.3.3 实现公共功能的基类

在查看高层设计时，很快就能认识到贷款中介中某些组件有共同的功能。例如，银行和信用机构都提供了接收请求、处理请求及把结果发布到另一个通道中的服务。听起来非常简单。但是由于采用了异步消息传递，即使是实现一个简单的请求/应答方案，我们也要做一些额外的操作。首先，服务的调用者要为消息的应答指定返回地址（Return Address），从而使不同的调用者能使用相同的服务和不同的应答队列。服务还要支持关联标识符，这样调用者才能把接收到的应答消息与请求消息关联起来。其次，如果服务接收到不可识别

的消息，最好能把该消息路由给非法消息通道，而不是简单地丢掉它。

为了消除代码重复（这是面向对象程序设计的致命弱点），我们创建了 MQService 类。该类同时支持返回地址和关联标识符。实际上，服务器端只不过把输入消息的消息 ID 复制为应答消息的关联 ID，以此提供对关联标识符的支持。在我们的例子中，还要复制 AppSpecific 属性，因为以后你会看到，还可能用消息 ID 以外的其他属性来关联消息。MQService 还要确保把响应消息发送到指定的返回地址。由于请求者提供了返回地址，MQService 惟一需要的实例化参数就是请求队列名，新的请求消息将到达此队列。如果请求者忘记提供返回地址，RequestReplyService 就把应答消息发送给非法消息通道。也可以考虑把请求消息发送给非法消息通道，这是因为错误正是由该消息造成的。目前我们还不必考虑错误处理的实现细节。

MQService.cs

```
public abstract class MQService
{
    static protected readonly String InvalidMessageQueueName =
                                        ".\\private$\\invalidMessageQueue";
    IMessageSender invalidQueue = new MessageSenderGateway(InvalidMessage QueueName);

    protected IMessageReceiver requestQueue;
    protected Type requestBodyType;

    public MQService(IMessageReceiver receiver)
    {
        requestQueue = receiver;
        Register(requestQueue);
    }

    public MQService(String requestQueueName)
    {
        MessageReceiverGateway q = new MessageReceiverGateway(requestQueue Name,
                                                              GetFormatter());
        Register(q);
        this.requestQueue = q;
        Console.WriteLine("Processing messages from " + requestQueueName);
    }

    protected virtual IMessageFormatter GetFormatter()
    {
        return new XmlMessageFormatter(new Type[] { GetRequestBodyType() });
    }
```

```
    protected abstract Type GetRequestBodyType();

    protected Object GetTypedMessageBody(Message msg)
    {
        try
        {
            if (msg.Body.GetType().Equals(GetRequestBodyType()))
            {
                return msg.Body;
            }
            else
            {
                Console.WriteLine("Illegal message format.");
                return null;
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Illegal message format" + e.Message);
            return null;
        }
    }
```

```
    public void Register(IMessageReceiver rec)
    {
        OnMsgEvent ev = new OnMsgEvent(OnMessage);
        rec.OnMessage += ev;
    }

    public void Run()
    {
        requestQueue.Begin();
    }


    public void SendReply(Object outObj, Message inMsg)
    {
        Message outMsg = new Message(outObj);
        outMsg.CorrelationId = inMsg.Id;
        outMsg.AppSpecific = inMsg.AppSpecific;

        if (inMsg.ResponseQueue != null)
        {
```

```
            IMessageSender replyQueue = new MessageSenderGateway(inMsg.Respon seQueue);
            replyQueue.Send(outMsg);
        }
        else
        {
            invalidQueue.Send(outMsg);
        }
    }

    protected abstract void OnMessage(Message inMsg);
}
```

这是一个抽象类，因为它并没有实现 GetTypedMessageBody 和 OnMessage 方法。由于我们的类要尽可能处理强类型的业务对象，而不是消息数据类型，因此 MQService 必须检查消息体的类型，并将其强制转换为正确的类型。该抽象基类的缺点是，它无法确定应该把消息数据类型强制转换为哪种业务对象类型，这是因为基类可能被许多不同的服务实现使用，而每种实现都可能使用不同的消息类型。为了在基类中尽可能实现更多的操作，我们创建了 GetTypedMessageBody 方法和 GetRequestBodyType 抽象方法。每个子类必须实现 GetRequestBodyType 方法，设置它所希望接收的消息的类型。MQServer 使用这个类型初始化 XML 格式化程序（formatter），并完成类型的检查。在完成类型检查后，子类就可以把输入的消息体安全地强制转换为所需的类型，而不会引发异常。目前，GetTypedMessageBody 内部的异常处理还非常简单，它只是把消息输出到控制台。如果不是一个简单的演示程序，就要定义一个更为复杂的方法来把消息记录下来，更好的做法可能是采用一个更全面的控制总线（Control Bus）。

OnMessage 方法要由 MQService 的子类实现。我们提供了两种实现方法，一种是异步方式，一种是同步方式。同步实现（RequestReplyService）调用了 ProcessMessage 虚方法，它等待应答消息返回，并立即调用 SendReply。与之相反，异步实现（AsyncRequestReplyService）定义的 ProcessMessage 虚方法没有任何返回参数。要由继承子类负责调用 SendReply。

MQService.cs

```
public class RequestReplyService : MQService
{
    public RequestReplyService(IMessageReceiver receiver) : base(receiver) {}
    public RequestReplyService(String requestQueueName) : base (request QueueName) {}

    protected override Type GetRequestBodyType()
    {
        return typeof(System.String);
```

```
    }

    protected virtual Object ProcessMessage(Object o)
    {
        String body = (String)o;
        Console.WriteLine("Received Message: " + body);
        return body;
    }

    protected override void OnMessage(Message inMsg)
    {
        inMsg.Formatter = GetFormatter();
        Object inBody = GetTypedMessageBody(inMsg);
        if (inBody != null)
        {
            Object outBody = ProcessMessage(inBody);

            if (outBody != null)
            {
                SendReply(outBody, inMsg);
            }
        }
    }

public class AsyncRequestReplyService : MQService
{
   public AsyncRequestReplyService(IMessageReceiver receiver) : base(receiver) {}
   public AsyncRequestReplyService(String requestQueueName) : base (request QueueName) {}


    protected override Type GetRequestBodyType()
    {
        return typeof(System.String);
    }

    protected virtual void ProcessMessage(Object o, Message msg)
    {
        String body = (String)o;
        Console.WriteLine("Received Message: " + body);
    }

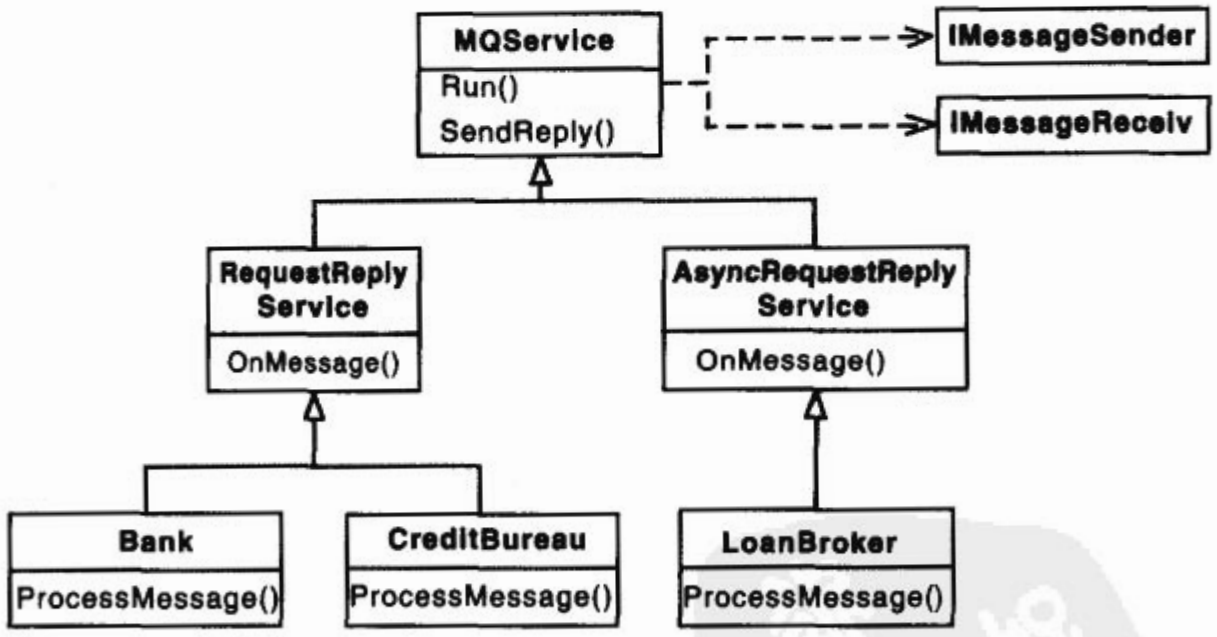
    protected override void OnMessage(Message inMsg)
    {
```

```
        inMsg.Formatter = GetFormatter();
        Object inBody = GetTypedMessageBody(inMsg);
        if (inBody != null)
        {
            ProcessMessage(inBody, inMsg);
        }

    }
}
```

这两个类都提供了 GetRequestBodyType 和 ProcessMessage 方法的默认实现。GetRequestBodyType 期望的消息是一个简单的字符串，ProcessMessage 把字符串输出到控制台。从技术上讲，可以从 RequestReplyService 和 AsyncRequestReplyService 类中去掉这些方法的默认实现，使它们保持抽象。这样的话，如果这些类的任何一个子类没有实现其中某个抽象方法，编译程序就会检测出来。但是，为了测试和调试的目的，最好提供服务的默认实现，因此我们具体实现了这些方法，以便实例化这两个类。

总之，基类的类图如下图所示（我们将简要介绍银行、信用机构和贷款中介类）。



提供消息服务的基类

9.3.4 设计 Bank

既然已经建立了一组基类并实现了相应的调用函数，下面就要实现应用的逻辑了。创建解决方案的一个简单方法是，按照依赖性递减的顺序建立应用组件。也就是说，要首先创建不依赖于其他任何组件的组件。这样我们就能独立运行和测试这些组件。银行就是这样的组件之一。贷款中介依赖于银行，但是银行本身是自包含的。银行就是一个请求/应答服务的例子，因此，实现银行时只需继承 RequestReplyService，并填入一些业务逻辑即可。

不过，在开始实现银行的内部操作之前，应该定义其外部接口。首先要定义贷款利率

查询请求和应答的消息类型。在这个简单的实现中，我们为所有银行定义了共同的消息格式，从而能为五个银行实例使用共同的类。C#支持结构（struct），因此我们使用结构作为消息的类型：

银行使用的消息类型

```
public struct BankQuoteRequest
{
    public int SSN;
    public int CreditScore;
    public int HistoryLength;
    public int LoanAmount;
    public int LoanTerm;
}

public struct BankQuoteReply
{
    public double InterestRate;
    public String QuoteID;
    public int ErrorCode;
}
```

由于想要为所有银行的实例采用同样的类，因此需要通过参数表示不同银行的操作。我们实现的银行只是一个非常简单的金融机构，所以它只需要 BankName、RatePremium 和 MaxLoanTerm 参数。RatePremium 设置了银行征收的高于最低利率的利率点数，这就是银行的利润率。MaxLoanTerm 设置了银行能接受的最长贷款期限（按月计算）。如果贷款所请求的期限超过了设置的最长期限，银行将拒绝贷款。在添加了相应的构造函数和存取方法后，接着要建立银行的 ProcessMessage 方法：

Bank.cs

```
internal class Bank : RequestReplyService
{
  ...

  protected override Type GetRequestBodyType()
  {
      return typeof(BankQuoteRequest);
  }

  protected  BankQuoteReply ComputeBankReply(BankQuoteRequest requestStruct)
  {
      BankQuoteReply replyStruct = new BankQuoteReply();

      if (requestStruct.LoanTerm <= MaxLoanTerm)
      {
```

```
            replyStruct.InterestRate = PrimeRate + RatePremium
                                     + (double)(requestStruct.LoanTerm / 12)/10
                                     + (double)random.Next(10) / 10;
            replyStruct.ErrorCode = 0;
        }
        else
        {
            replyStruct.InterestRate = 0.0;
            replyStruct.ErrorCode = 1;
        }
        replyStruct.QuoteID = String.Format("{0}-{1:00000}", BankName, quoteCounter);
        quoteCounter++;
        return replyStruct;
    }

    protected override Object ProcessMessage(Object o)
    {
        BankQuoteRequest requestStruct;
        BankQuoteReply replyStruct;

        requestStruct = (BankQuoteRequest)o;
        replyStruct = ComputeBankReply(requestStruct);

        Console.WriteLine("Received request for SSN {0} for {1:c} / {2} months",
                          requestStruct.SSN, requestStruct.LoanAmount,
                          requestStruct.LoanTerm);
        Thread.Sleep(random.Next(10) * 100);
        Console.WriteLine("  Quote: {0} {1} {2}",
                          replyStruct.ErrorCode, replyStruct.InterestRate,
                          replyStruct.QuoteID);

        return replyStruct;
    }
}
```

从上面的代码可以看到，具体的服务只需实现 GetRequestBodyType 和 ProcessMessage 方法。该服务能使用 ProcessMessage 安全地强制转换传入的对象，这是因为基类已经验证了类型的正确性。剩下的实现基本与消息传递无关，所有消息传递的细节都已由基类实现。MQService 和 RequestReplyService 类充当了服务激励器（Service Activator），从而使应用不必考虑消息传递系统的细节。

ComputeBankReply 方法包含了银行的全部业务逻辑。当然实际中并没有这么简单！要知道我们关心的并不是宏观经济学，而是消息传递的实例，因此可以在这里做一定程度的简化。计算出的利率是最低利率、所配置的额外利率（高出最低利率的点数）、贷款期限和某个随机数的和。如果所请求的贷款期限超出了银行可接受的期限，则返回一个错误代码。银行发布的每个利率消息都包含一个惟一的引用 ID，以便客户在事后查询。在目前的实现

中，这些引用 ID 是由一个简单的递增计数器创建的。

为了较为真实地模拟银行交易，ProcessMessage 方法实现了一定的延迟（1/10 秒到 1 秒之间）。ProcessMessage 还把部分操作记录到控制台，这样一来，我们在一个简单的控制台应用中运行银行组件时就能看到它的执行情况。

要启动银行，首先要采用相应的参数将其实例化，然后调用从 MQService 中继承来的 Run 方法。由于处理操作是由事件触发的，Run 方法将立即返回。因此，在程序启动后不要立即终止程序运行。为了便于测试，我们在调用 Run 方法后简单地插入了一条 Console.ReadLine()语句。

9.3.5 设计信用机构

信用机构的实现类似于银行。惟一的区别在于消息类型和业务逻辑。信用机构要处理以下消息类型：

信用机构使用的消息类型

```
public class CreditBureauRequest
{
    public int SSN;
}

public class CreditBureauReply
{
    public int SSN;
    public int CreditScore;
    public int HistoryLength;
}
```

ProcessMessage 方法的代码基本与银行类实现的代码一样，只不过会处理不同的数据结构，并调用了不同的应用逻辑。信用机构内部也建立了时间延迟。

CreditBureau.cs

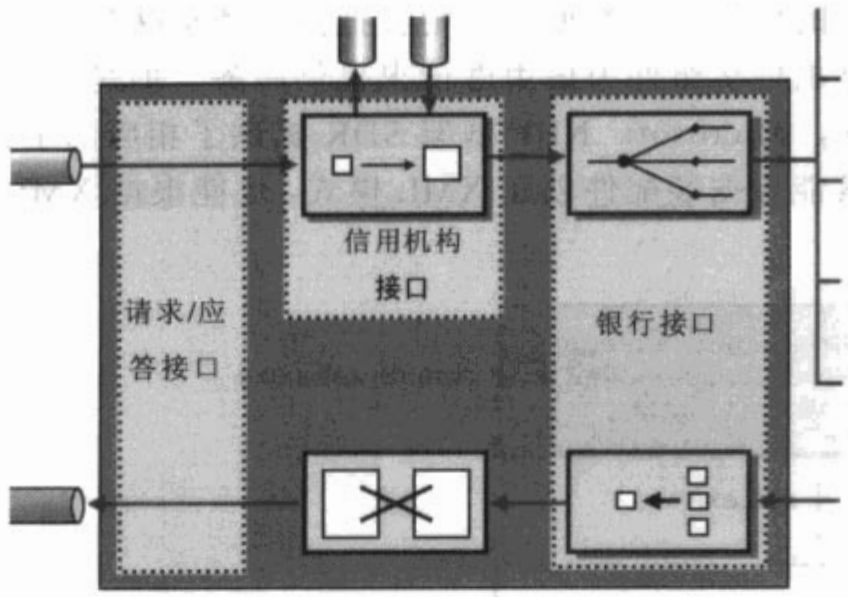
```
private int getCreditScore(int ssn)
{
    return (int)(random.Next(600) + 300);
}

private int getCreditHistoryLength(int ssn)
{
    return (int)(random.Next(19) + 1);
}
```

9.3.6 设计贷款中介

具体实现了信用机构类和能代表多家银行的银行类之后，现在我们就来详细设计贷款中介。本书介绍的路由和转换模式有助于我们把贷款中介提供的功能进一步细化。可以把

贷款中介的内部功能划分为三个主要部分（如下图所示）：从客户端接受请求的请求/应答接口、信用机构接口和银行接口。

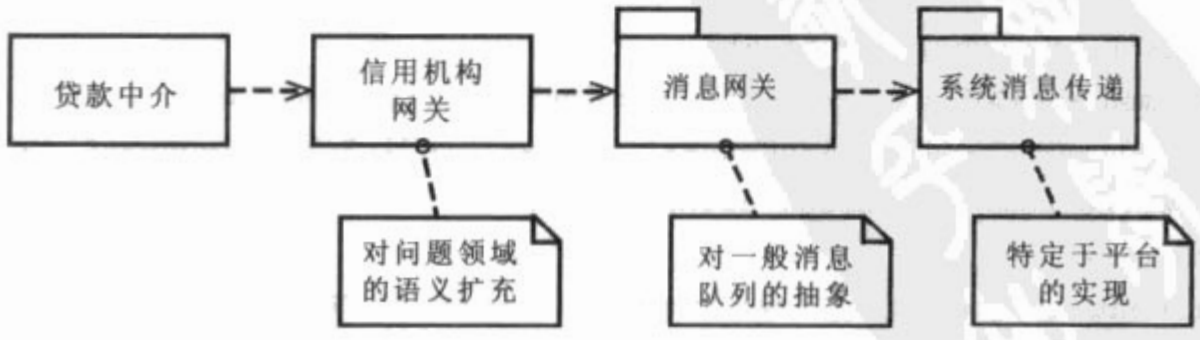


贷款中介的内部结构

我们同样按照依赖程度递减的顺序构建整个解决方案，首先构建只依赖于已有组件的组件。由于我们已经构建了银行服务和信用机构服务，因此只需创建从贷款中介到这些外部组件的接口。信用机构的接口看起来较为简单，所以我们先定义信用机构接口。

信用机构网关

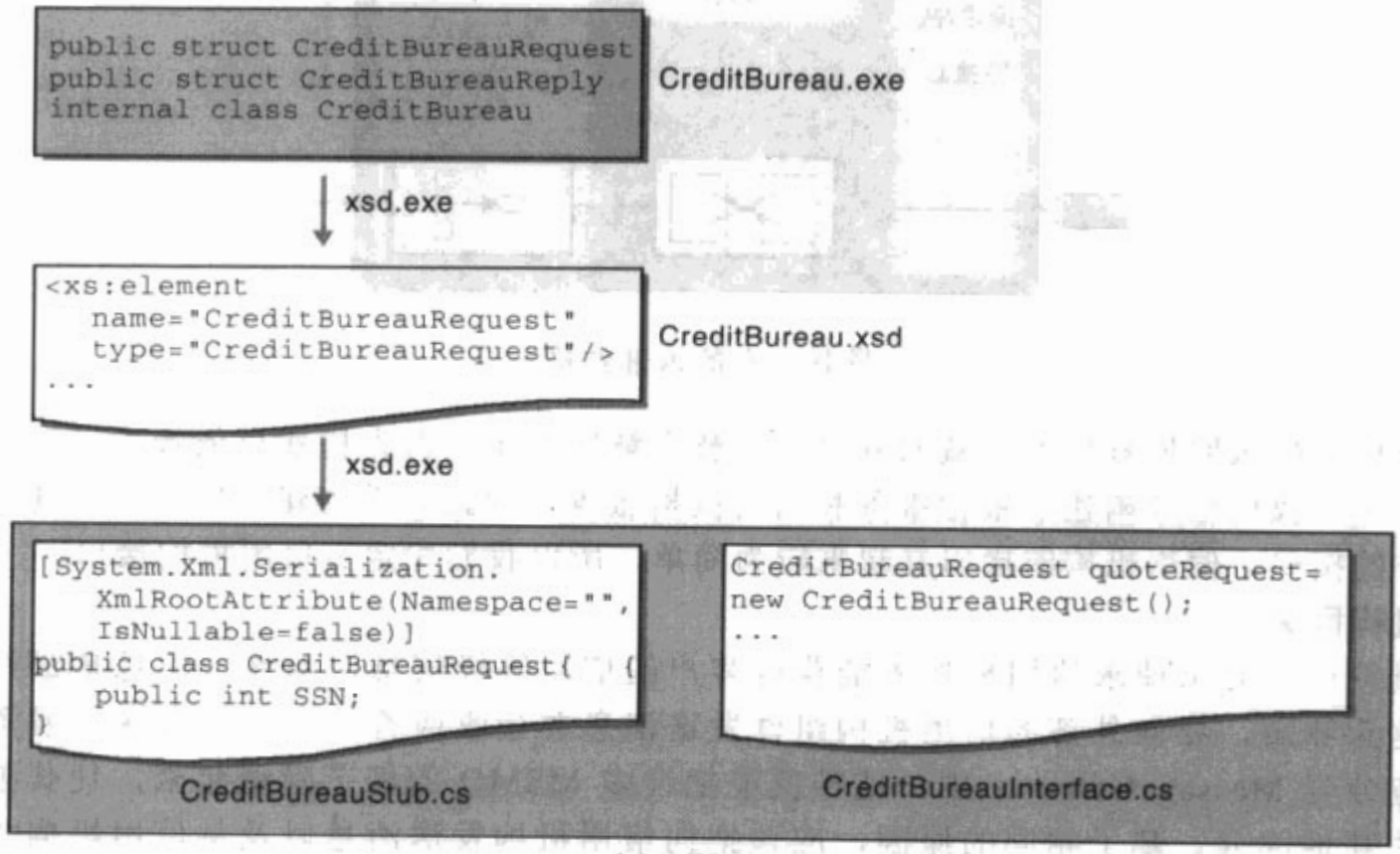
贷款中介必须请求信用机构才能获得客户的信用等级信息，而这一信息是银行所需的。这意味着，要向外部的信用机构组件发送消息并接收应答消息。把发送一般消息的代码包装到 MessageGateway 中，这样就能把许多 MSMQ 的细节隐藏起来，使其独立于应用的其他部分。基于相同的原因，应该把向信用机构发送消息以及从信用机构接收消息的代码包装在一个信用机构网关中。该信用机构网关要实现语义扩充的重要功能，允许贷款中介调用诸如 GetCreditScore 的方法，而不是 SendMessage 方法。这样一来，贷款中介代码的可读性就会增强，并且对贷款中介和信用机构之间的通信提供了强有力的封装。下图说明了通过“链接”两个网关而得到的抽象层。



贷款中介对消息传递基础设施提供了另一个抽象层

为了请求信用度，网关需要创建 CreditBureauRequest 结构的一个实例，此结构由信用机构指定。同样地，网关接口将接收 CreditBureauReply 结构中的结果。前面我们说过，这

个解决方案是由不同的可执行组件构建的，因此信用机构与贷款中介可能在不同的计算机上运行。贷款中介可能无法访问定义在信用机构装配件中的类型，实际上我们也不想让贷款中介引用信用机构中的任何数据类型，否则会抵消由消息队列带来的松耦合特性。贷款中介应该完全不知道是由什么组件为信用度请求提供服务。但是，它要能访问定义了消息格式的结构。幸运的是，Microsoft .NET 框架 SDK 提供了相应的工具：XML 模式定义工具（xsd.exe）。这个工具能根据装配件创建 XML 模式，还能根据 XML 模式创建 C#源代码。下图描述了这一过程。



根据另一个装配件创建类桩

xsd.exe 抽取公共类型定义，并根据类型定义和用于控制串行化的可选属性创建一个 XML 模式文件。在我们的例子中，xsd.exe 会创建以下模式：

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault= "qualified" xmlns:xs="http:// www.w3.org/2001/ XMLSchema">
  <xs:element name="CreditBureauRequest" type="CreditBureauRequest" />
  <xs:complexType name="CreditBureauRequest">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="SSN" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="CreditBureauReply" type="CreditBureauReply" />
```

```
  <xs:complexType name="CreditBureauReply">
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="1" name="SSN" type="xs:int" />
      <xs:element minOccurs="1" maxOccurs="1" name="CreditScore" type= "xs:int" />
      <xs:element minOccurs="1" maxOccurs="1" name="HistoryLength" type= "xs:int" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Run" nillable="true" type="Run" />
  <xs:complexType name="Run" />
</xs:schema>
```

通常，服务将把该模式定义发布给可能的调用者。调用者可以用不同的方式产生所需的消息格式。首先，调用者可以显式地构造与 XSD 兼容的请求消息。其次，调用者也可以使用.NET 内部提供的串行化操作。在这种情况下，客户端还能选择不同的编程语言，因为.NET CLR 与程序设计语言无关。

我们决定采用.NET 的内建串行化操作。因此，要再次运行 xsd.exe，创建服务消费者使用的源代码，产生的文件如下所示：

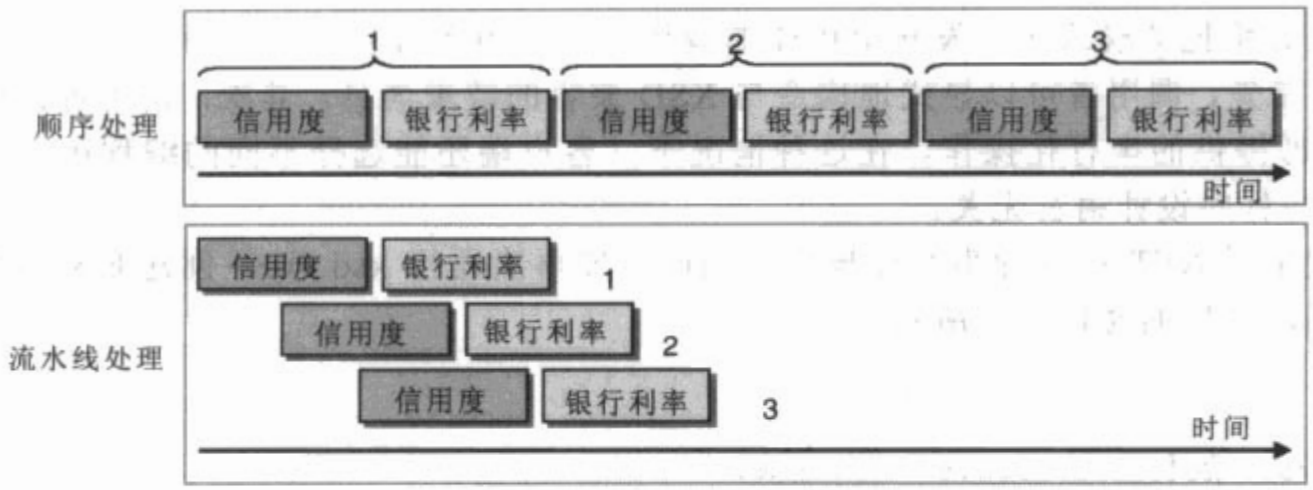
```
//
// 该源代码由 xsd，版本=1.1.4322.573 自动生成。
//
namespace CreditBureau {
    using System.Xml.Serialization;


    /// <remarks/>
    [System.Xml.Serialization.XmlRootAttribute(Namespace="",IsNullable= false)]
    public class CreditBureauRequest {

        /// <remarks/>
        public int SSN;
    }
    ...
}
```

值得注意的是，.NET XML 串行化和逆串行化支持松耦合。因此，从技术上讲，只要 XML 表示包含了所需的元素，发送给信用机构的请求消息不一定严格采用信用机构内部使用的同样的 CLR 类。例如，请求者可以发送一个消息，它的 XML 表示能包含额外的元素，但不会干扰正常通信。在这个例子中，我们假设贷款中介愿意遵循信用机构指定的格式，并在通信的两端使用了相同的数据类型。

现在已经准备好以正确的格式向信用机构发送消息。但是，要注意该通信是异步的，请求和应答消息是异步的、单独的。信用机构网关可以设计如下：当发送完请求消息后，网关要一直等待响应消息的返回。采用这种方法有一个明显的缺点：信用机构正在处理消息时，应用只能等待。这种伪同步处理方式很快就会导致性能瓶颈。如果每一步操作都是伪同步的，那么贷款中介就只能一次处理一个请求。例如，它在等待银行对以前的查询请求返回应答时，就不能向信用机构发出新的查询信用度的请求。为了形象地说明同步与异步操作的区别，假设贷款中介必须执行两个主要的处理步骤：获得信用度，以及从银行获得最佳利率信息。如果假设贷款中介只能以单一的、顺序的方式执行，那么执行的过程如图中上半部分所示。



流水线处理能显著地提高吞吐量

由于实际的处理操作是由外部组件执行的，贷款中介组件基本上只是等待处理结果，它没有充分利用计算资源。如果把整个贷款中介过程设计成事件驱动的消费者，就能同时处理多个请求和返回的结果。这种处理方式被称为流水线处理（pipeline processing）。现在，系统的可扩展性只是由外部组件的处理能力决定，而不在于贷款中介。如果我们只运行信用机构过程的一个实例，由于信用机构的请求队列总会让请求排队，因此顺序处理和流水线处理的区别并不明显。但是，如果同时运行多个信用机构实例，就能立即看到在性能方面的改善（后面将更多地介绍性能）。

实现事件驱动的贷款中介过程主要有两种方法。一种方法是创建一个顺序处理的进程，但是同时创建新的线程来处理每个到来的请求消息。另一种方法是，当有事件到来时，让消息传递系统通知贷款中介。采用这种方法，消息传递系统负责控制线程的执行。上述两种方法都各有优缺点。编写顺序进程使代码更易于理解，但是，如果我们的组件主要是一个代理组件，它负责在外部的组件之间代理消息，就会导致大量线程的存在，这些线程无所事事，只是在等待输入消息。这些线程会占用大量的系统资源和珍贵的处理时间，因此最好是让消息传递系统驱动线程的执行。当消息准备好可以消费时，系统将调度中介执行，因此只需维护单个线程的执行，而不必考虑线程管理问题。但是，事件驱动的方法存在这样一个问题，消息到达时，会执行多个代码段，而不只是一个顺序方法。

你可能认为，在.NET 中要实现事件驱动就要使用委托。确实，信用机构网关定义了一

个新的委托。

```
public delegate void OnCreditReplyEvent(CreditBureauReply creditReply, Object ACT);
```

当返回的结果到达时，该委托允许其他的代码段通知信用机构网关应该调用哪个方法。信用机构网关把有正确类型的 CreditBureauReply 结构返回给调用者。它还传递了一个 ACT，即异步完成令牌（Asynchronous Completion Token）[POSA2]。利用该令牌，使调用者能把数据传递给网关，并且当相应的应答消息返回时，还可接收到返回的数据（见“消息传递网关”）。从本质上讲，信用机构网关执行了请求和应答消息的关联，所以调用者不再需要做这项操作。

剩下的就是编写一个方法来请求信用度，还要编写一个方法处理到来的应答消息，为之关联适当的 ACT，并调用正确类型的委托。

CreditBureauGateway.cs

```
internal struct CreditRequestProcess
{
    public int CorrelationID;
    public Object ACT;
    public OnCreditReplyEvent callback;
}
 internal class CreditBureauGateway
{
    protected IMessageSender creditRequestQueue;
    protected IMessageReceiver creditReplyQueue;

    protected IDictionary activeProcesses = (IDictionary)(new Hash table());

    protected Random random = new Random();

    public void Listen()
    {
        creditReplyQueue.Begin();
    }

    public void GetCreditScore(CreditBureauRequest quoteRequest,
                               OnCreditReplyEvent OnCreditResponse, Object ACT)
    {
        Message requestMessage = new Message(quoteRequest);
        requestMessage.ResponseQueue = creditReplyQueue.GetQueue();
        requestMessage.AppSpecific = random.Next();

        CreditRequestProcess processInstance = new CreditRequestProcess ();
        processInstance.ACT = ACT;
```

```
        processInstance.callback = OnCreditResponse;
        processInstance.CorrelationID = requestMessage.AppSpecific;

        creditRequestQueue.Send(requestMessage);

        activeProcesses.Add(processInstance.CorrelationID, processInstance);
    }

    private void OnCreditResponse(Message msg)
    {
        msg.Formatter =  GetFormatter();

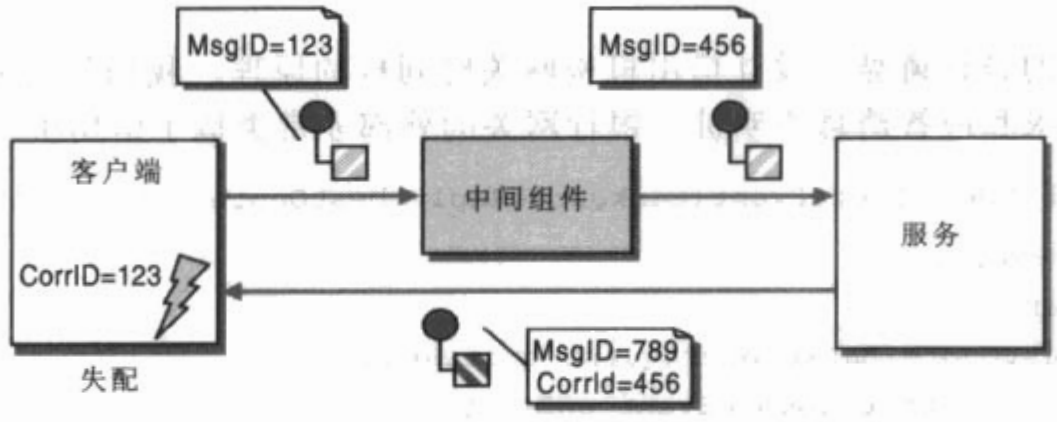
        CreditBureauReply replyStruct;
        try
        {
            if (msg.Body is CreditBureauReply)
            {
                replyStruct = (CreditBureauReply)msg.Body;
                int CorrelationID = msg.AppSpecific;

                if (activeProcesses.Contains(CorrelationID))
                {
                    CreditRequestProcess processInstance = (CreditRequestProcess) (active.
Processes [CorrelationID]);
                    processInstance.callback(replyStruct, processInstance.ACT);
                    activeProcesses.Remove(CorrelationID);
                }
                else { Console.WriteLine
                        ("Incoming credit response does not match any request"); }
            }
            else
            { Console.WriteLine("Illegal reply."); }
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception: {0}", e.ToString());
        }
    }
}
```

当调用者通过 GetCreditScore 方法请求信用度时，信用机构网关分配一个新的 CreditRequestProcess 结构的实例。activeProcesses 集合中，对应每个发出的请求都包含一个 CreditRequestProcess 实例，并以消息的关联标识符作为索引。这个结构还包含了响应 OnCreditReplyEvent 事件的委托。通过存储每个消息的委托，调用者就能为每个请求指定

不同的回调位置。后面会看到，采用这种方法能使调用者使用委托来管理会话状态。

值得注意的是，我们没有使用消息本身的消息 ID 字段来关联消息，而是为 AppSpecific 字段指定了一个随机整数，通过这个字段的值来关联到来的消息（要记住，RequestReplyService 方法设计为，要把消息 ID 字段和 AppSpecific 字段都复制到应答消息中）。为什么不使用消息 ID，而使用其他字段关联消息呢？消息 ID 的优点是，系统中的每个消息有惟一的 ID，但是灵活性有限。如果要把应答消息与请求消息按消息 ID 关联，就不能消息流中插入中间处理步骤（如路由器）。因为任何中间处理步骤都会消费掉请求消息并向服务发布新的消息，这样一来，应答消息的 CorrelationId 将与服务接收到的消息匹配，而不是贷款中介最初发送的消息（如下图所示）。



中间组件会阻碍采用系统生成的消息 ID 建立关联

解决这个问题有两种方法。第一种方法是，任何中间组件都要把请求和应答消息拦截下来，并且在应答消息中设置正确的 CorrelationId 值（采用该方法的例子请参见“智能代理”[xxx]）。第二种方法是，使用其他字段作为关联字段，使所有流经中间组件和服务的相关消息有相同的关联标识符。在这个例子中，我们选择使用第二种方法，这样中间组件就能更方便地拦截贷款中介和信用机构之间的请求消息（在第 12 章“中场演练：系统管理示例”中将利用这种方法）。应该如何选择 AppSpecific 属性的值呢？可以使用连续的值，但是要确保两个并发的实例不会使用相同的起始值。也可以使用集中的 ID 生成模块（如数据库），由它保证 ID 在系统范围内的惟一性。对这个简单的示例来说，没有必要采用后面这种方式，所以我们选择使用随机数。.NET 生成的随机数是一个有符号的 32 位整数，因此出现重复数字的概率为 20 亿分之一，这样的风险是可以接受的。

信用机构网关已经提供了简洁的抽象，使我们不必考虑 Windows 消息排队基础设施，这正是我们所期望的。信用机构网关提供的唯一公共接口只有一个委托和两个方法（除了构造函数）：

```
delegate void OnCreditReplyEvent(CreditBureauReply creditReply, Object ACT);
class CreditBureauGateway {
  void Listen() {...}
  void GetCreditScore(CreditBureauRequest quoteRequest,
```

```
                    OnCreditReplyEvent OnCreditResponse,
                    Object ACT) {...}
    ...
}
```

所有方法都没有引用消息或消息队列。这样做有很多好处。首先，有利于实现信用机构网关的一个桩版本，而根本不依赖于消息队列（类似于 MockQueue）。其次，如果决定使用不同于 MSMQ 的传输系统，可以替换信用机构网关的具体实现。例如，如果我们要使用的 Web 服务接口采用的是 SOAP 和 HTTP 而不是 MSMQ，网关提供的方法很可能不需要做任何修改。

银行网关

银行网关的设计遵循了设计信用机构网关时同样的原理。我们使用前面相同的过程为银行指定的请求和应答消息声明桩。银行网关的外部非常类似于信用机构网关。

```
delegate void OnBestQuoteEvent(BankQuoteReply bestQuote, Object ACT);
class BankGateway {
    void Listen() {...}
    void GetBestQuote(BankQuoteRequest quoteRequest,
                      OnBestQuoteEvent onBestQuoteEvent,
                      Object ACT) {...}
    ...
}
```

由于要把单个 BankQuoteRequest 路由给多家银行，这种分散收集方式的交互增加了银行网关内部处理的复杂性。另外，一个 BankQuoteReply 通常是多家银行应答的利率消息的结果。请求消息的处理由接收表完成，而应答消息的处理由聚合器完成。首先让我们来看看接收表的实现，它必须实现 3 个主要功能：

- 计算相应接收者
- 把消息发送给接收者
- 初始化聚合器，以处理到来的应答消息

正如在本章引言中所介绍的，该实现使用了分布式的分散收集器，实时确定应该把请求路由给哪家银行。如果银行要对中介征收每笔利率查询的费用，或者银行和中介达成了协议，中介必须事先确定客户请求的路线，那么就应该采用这种模式。贷款中介根据客户的信用度、贷款的数量和信用历史记录做出路由决定。我们把各个与银行的连接封装在一个类中，这个类是继承自抽象 BankConnection 类的子类。该类包含了正确定址的消息队列的引用，还包括一个 CanHandleLoanRequest 方法，能确定是否应该把利率查询请求转发给该银行。BankConnectionManager 只是迭代处理包含所有银行连接的列表，并把符合贷款利率查询标准的银行连接汇集起来。如果银行的列表很长，可以考虑实现一个可配置的规则引擎。我们更倾向于目前的这种方法，因为它更简单，更明确。

```
internal class BankConnectionManager
{
    static protected BankConnection[] banks =
                    {new Bank1(), new Bank2(), new Bank3(), new Bank4(), new Bank5() };

    public IMessageSender[] GetEligibleBankQueues
                            (int CreditScore, int HistoryLength, int LoanAmount)
    {
        ArrayList lenders = new ArrayList();

        for (int index = 0; index < banks.Length; index++)
        {
            if (banks[index].CanHandleLoanRequest(CreditScore, History Length,
                                                  LoanAmount))
                lenders.Add(banks[index].Queue);
        }
        IMessageSender[] lenderArray = (IMessageSender [])Array.Create Instance
                                       (typeof(IMessageSender), lenders.Count);
        lenders.CopyTo(lenderArray);
        return lenderArray;
    }
}

internal abstract class BankConnection
{
    protected MessageSenderGateway queue;
    protected String bankName = "";
    public MessageSenderGateway Queue
    {
        get { return queue; }
    }
    public String BankName
    {
        get { return bankName; }
    }
    public BankConnection (MessageQueue queue)
      { this.queue = new MessageSenderGateway(queue); }
    public BankConnection (String queueName)
      { this.queue = new MessageSenderGateway(queueName); }

   public abstract bool CanHandleLoanRequest(int CreditScore, int History Length,
                                             int LoanAmount);
```

```
}

internal class Bank1 : BankConnection
{
    protected String bankname = "Exclusive Country Club Bankers";

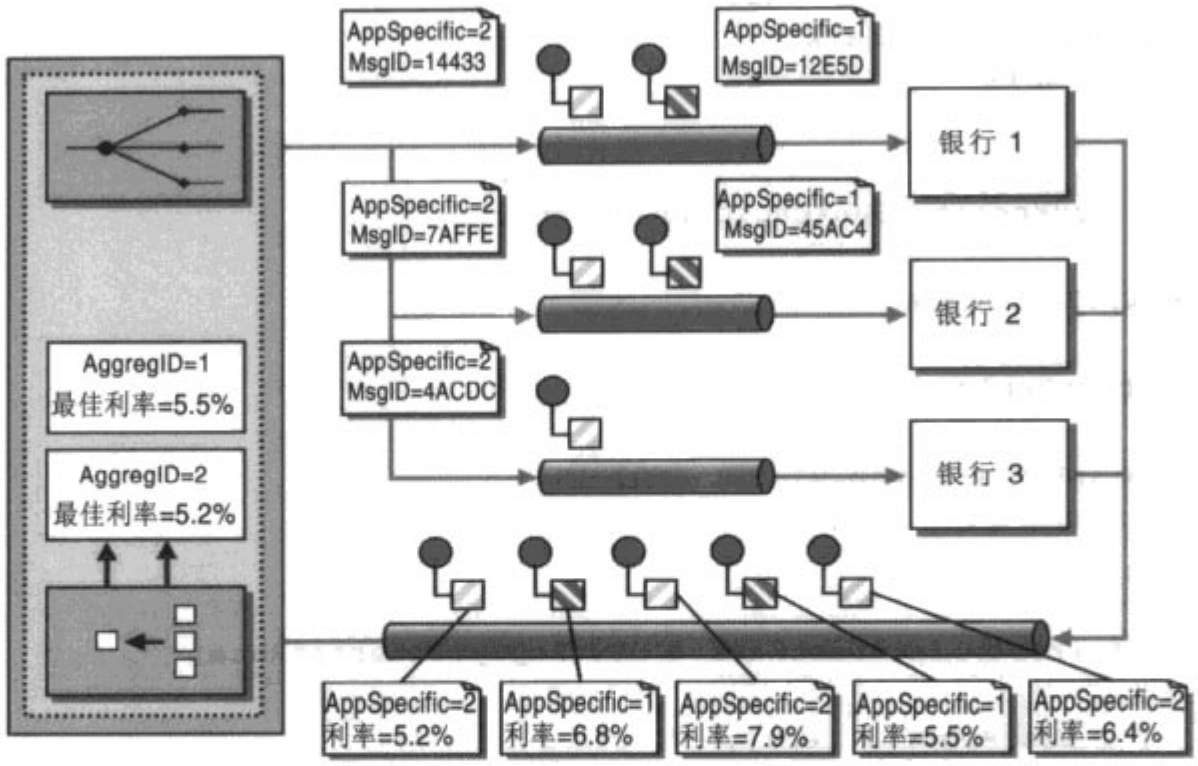
    public Bank1 () : base (".\\private$\\bank1Queue") {}
    public override bool CanHandleLoanRequest(int CreditScore, int HistoryLength,
                                              int LoanAmount)
    {
        return LoanAmount >= 75000 && CreditScore >= 600 && HistoryLength >= 8;
    }
}
...
```

一旦汇集得到相关银行的列表，要发送消息只是迭代处理这个列表而已。在生产环境中，这种迭代操作应该发生在一个事务中，以避免消息只发送给部分银行。同样地，在这个例子中，为简单起见，我们没有实现相应的代码。

```
internal class MessageRouter
{
    public static void SendToRecipientList (Message msg, IMessageSender[] recipientList)
    {
        IEnumerator e = recipientList.GetEnumerator();
        while (e.MoveNext())
        {
            ((IMessageSender)e.Current).Send(msg);
        }
    }
}
```

既然请求消息能以其自己的方式发送到银行，下面需要初始化聚合器，以等待返回的银行利率信息。由于贷款中介是事件驱动的，聚合器要做好同时处理多个集合的准备，要为每个未完成的利率查询请求维护一个活动的集合。也就是说，到来的消息只需惟一地与特定集合关联。遗憾的是，我们不能使用消息 ID 作为关联标识符，因为接收表必须向每家银行发送单个的消息。因此，如果有三家银行参与某个利率查询请求的服务，接收表就必须发送三条有不同消息 ID 的消息，每个消息对应一家银行。这些消息都有一个惟一的消息 ID。如果银行要通过消息 ID 进行关联，那么即使三个应答消息同属于一个集合，它们也将有不同的关联 ID。这就使聚合器很难识别出相关的消息。也可以让集合存储每个请

求消息的消息 ID，从而将输入消息的关联 ID 重新与集合关联。但是，没有必要采用这么复杂的方法。相反，我们只需生成自己的关联 ID，每个关联 ID 对应一个集合而不是一个消息。我们把这种（数字的）ID 存储在要发出的请求消息的 AppSpecific 属性中。银行继承自 RequestReplyService，它已经把输入消息的 AppSpecific 属性复制到应答消息中。从银行返回的利率消息到达时，BankGateway 就能方便地通过消息的 AppSpecific 属性把到来的消息关联起来（如下图所示）。



BankGateway 使用 AppSpecific 消息属性将响应消息与集合关联

银行网关采用集合 ID（由简单的计数器产生）和期望的消息数来初始化集合。另外，调用者需要提供一个委托，并能像信用机构网关那样采用相同的方式设置 ACT 对象的引用。聚合策略很简单。当所有被选中的银行都返回了应答消息，则认为聚合结束。接收表根据期望的消息数初始化集合。请记住，银行有权利拒绝提供利率响应。因此为了知道何时才能完成聚合，银行假如不想提供利率信息，则必须返回一条带有错误编码的应答消息。另外也可以很方便地修改聚合策略，例如在超过 1 秒钟后停止等待应答，并从已接收到的应答消息中取出最佳的响应。

```
internal class BankQuoteAggregate
{
    protected int ID;
    protected int expectedMessages;
    protected Object ACT;
    protected OnBestQuoteEvent callback;

    protected double bestRate = 0.0;
```

```
protected ArrayList receivedMessages = new ArrayList();
protected BankQuoteReply bestReply = null;

public BankQuoteAggregate(int ID, int expectedMessages, OnBestQuoteEvent callback,
                          Object ACT)
{
    this.ID = ID;
    this.expectedMessages = expectedMessages;
    this.callback = callback;
    this.ACT = ACT;
}

public void AddMessage(BankQuoteReply reply)
{
    if (reply.ErrorCode == 0)
    {
        if (bestReply == null)
        {
            bestReply = reply;
        }
        else
        {
            if (reply.InterestRate < bestReply.InterestRate)
            {
                bestReply = reply;
            }
        }
    }
    receivedMessages.Add(reply);
}

public bool IsComplete()
{
    return receivedMessages.Count == expectedMessages;
}

public BankQuoteReply getBestResult()
{
    return bestReply;
}

public void NotifyBestResult()
{
    if (callback != null)
    {
```

```
            callback(bestReply, ACT);
        }
    }
}
```

基于银行连接管理器，接收表、聚合以及 BankGateway 功能的实现都变得非常简单：

BankGateway.cs

```
internal class BankGateway
  {
      protected IMessageReceiver bankReplyQueue;
      protected BankConnectionManager connectionManager;

      protected IDictionary aggregateBuffer = (IDictionary)(new Hash table());
      protected int aggregationCorrelationID;

      public void Listen()
      {
          bankReplyQueue.Begin();
      }

  public void GetBestQuote(BankQuoteRequest quoteRequest,
                           OnBestQuoteEvent onBestQuoteEvent, Object ACT)
  {

      Message requestMessage = new Message(quoteRequest);
      requestMessage.AppSpecific = aggregationCorrelationID;
      requestMessage.ResponseQueue = bankReplyQueue.GetQueue();
      IMessageSender[] eligibleBanks =
          connectionManager.GetEligibleBankQueues(quoteRequest.CreditScore,
                                                  quoteRequest.HistoryLength,
                                                  quoteRequest.LoanAmount);

      aggregateBuffer.Add(aggregationCorrelationID,
          new BankQuoteAggregate(aggregationCorrelationID, eligible Banks. Length,
                                 onBestQuoteEvent, ACT));
      aggregationCorrelationID++;

    MessageRouter.SendToRecipientList(requestMessage, eligibleBanks);
}
```

```
private void OnBankMessage(Message msg)
{
    msg.Formatter =  GetFormatter();

    BankQuoteReply replyStruct;
    try
    {
        if (msg.Body is BankQuoteReply)
        {
            replyStruct = (BankQuoteReply)msg.Body;
            int aggregationCorrelationID = msg.AppSpecific;
            Console.WriteLine("Quote {0:0.00}% {1} {2}",
                              replyStruct.InterestRate, replyStruct. QuoteID,
                              replyStruct.ErrorCode);
         if (aggregateBuffer.Contains(aggregationCorrelationID))
           {
               BankQuoteAggregate aggregate =
              (BankQuoteAggregate)(aggregateBuffer[aggregationCorrelationID]);
               aggregate.AddMessage(replyStruct);
               if (aggregate.IsComplete())
                   {
                       aggregate.NotifyBestResult();
                       aggregateBuffer.Remove(aggregationCorrelationID);
                   }
               }
               else
            { Console.WriteLine("Incoming bank response does not match any aggregate"); }
           }
           else
           { Console.WriteLine("Illegal request."); }
       }
       catch (Exception e)
       {
           Console.WriteLine("Exception: {0}", e.ToString());
       }
   }
}
```

当银行网关接收到银行返回的利率应答后，执行 **OnBankMessage** 方法。该方法把到来

的消息转换为正确的类型，并通过 AppSpecific 属性查找相关的集合。它把新的应答消息添加到聚合中。一旦聚合的完成条件满足（如 BankQuoteAggregate 类中所定义），BankGateway 就会调用由调用者提供的委托。

接受请求

现在，我们已经很好地封装了信用机构网关和银行网关，下一步要让贷款中介接受请求。前面已经讨论过 MQService 和 AsyncRequestReplyService 基类的设计。LoanBroker 类继承自 AsyncRequestReplyService，因为它不能立即向应答队列返回结果，而必须等待其他几个异步操作（包括获得信用度和与银行通信）完成。

实现 LoanBroker 的第一步是定义贷款中介能处理的消息类型：

```
public struct LoanQuoteRequest
{
    public int SSN;
    public double LoanAmount;
    public int LoanTerm;
}

public struct LoanQuoteReply
{
    public int SSN;
    public double LoanAmount;
    public double InterestRate;
    public string QuoteID;
}
```

接下来创建继承自 AsyncRequestReplyService 的子类，并覆盖 ProcessMessage 方法。

处理

贷款中介与前面的类不同，因为消息到来后，触发的处理并不是包含在一个方法中。过程的完成取决于一系列外部事件。贷款中介可以接收三种类型的事件：

- 有新的贷款请求消息到达
- 有信用度应答消息到达（通过 CreditBureauGateway）
- 有银行利率消息到达（通过 BankGateway）

由于贷款中介的处理逻辑划分到多个事件处理程序中，我们必须在这些函数中保留贷款中介的状态。这就是使用异步完成令牌的原因！应该记得，信用机构网关和银行网关支持调用者（贷款中介）在发送请求时传递一个对象实例的引用。当接收到应答消息时，网关再把对象引用传回。为了利用这一功能，我们在贷款中介中声明了如下所示的 ACT：

```
internal class ACT
{
```

```
    public LoanQuoteRequest loanRequest;
    public Message message;

    public ACT(LoanQuoteRequest loanRequest, Message message)
    {
        this.loanRequest = loanRequest;
        this.message = message;
    }
}
```

该 ACT 包含了最初的请求消息的副本(包含消息 ID 和创建应答消息所需的应答地址)以及请求数据结构(用于把 SSN 和贷款数量复制到应答消息中)。从技术上讲，ACT 存储了少量的重复信息，因为这些信息也可以从请求消息的请求结构的内容中取得。但是，为了便于访问强类型结构，重复这点数据是值得的。

贷款中介其余的实现如下所示：

```
LoanBroker.cs
internal class LoanBroker : AsyncRequestReplyService
{
    protected ICreditBureauGateway creditBureauInterface;
    protected BankGateway bankInterface;

    public LoanBroker(String requestQueueName,
                      String creditRequestQueueName, String creditReply QueueName,
                      String bankReplyQueueName,
                      BankConnectionManager connectionManager): base(request QueueName)
    {

        creditBureauInterface = (ICreditBureauGateway)
          (new CreditBureauGatewayImp(creditRequestQueueName, creditReply QueueName));
        creditBureauInterface.Listen();

        bankInterface = new BankGateway(bankReplyQueueName, connection Manager);
        bankInterface.Listen();
    }

    protected override Type GetRequestBodyType()
    {
        return typeof(LoanQuoteRequest);
    }
```

```
protected override void ProcessMessage(Object o, Message msg)
{
    LoanQuoteRequest quoteRequest;
    quoteRequest = (LoanQuoteRequest)o;

    CreditBureauRequest creditRequest =
        LoanBrokerTranslator.GetCreditBureaurequest(quoteRequest);

    ACT act = new ACT(quoteRequest, msg);

    creditBureauInterface.GetCreditScore(creditRequest,
                                    new OnCreditReplyEvent(OnCreditReply),act);
}

private void OnCreditReply(CreditBureauReply creditReply, Object act)
{
    ACT myAct = (ACT)act;

   Console.WriteLine("Received Credit Score -- SSN {0} Score {1} Length {2}",
                     creditReply.SSN, creditReply.CreditScore,
                     creditReply.HistoryLength);
        BankQuoteRequest bankRequest =
            LoanBrokerTranslator.GetBankQuoteRequest(myAct.loanRequest ,creditReply);
        bankInterface.GetBestQuote(bankRequest, new OnBestQuoteEvent(OnBestQuote),act);
    }

    private void OnBestQuote(BankQuoteReply bestQuote, Object act)
    {
        ACT myAct = (ACT)act;

        LoanQuoteReply quoteReply = LoanBrokerTranslator.GetLoanQuoteReply
                                    (myAct.loanRequest, bestQuote);
        Console.WriteLine("Best quote {0} {1}",
                          quoteReply.InterestRate, quoteReply.QuoteID);
        SendReply(quoteReply, myAct.message);
    }
}
```

LoanBroker 继承自 AsyncRequestReplyService，AsyncRequestReplyService 对接收请求和发送相关应答提供了支持。LoanBroker 覆盖了 ProcessMessage 方法，以处理到来的请求消息。ProcessMessage 创建一个新的 ACT 实例，并调用信用机构网关来请求信用度。有趣

的是，该方法到此就结束了。当信用机构网关调用 OnCreditReply 时，才继续进行处理。OnCreditReply 是 ProcessMessage 方法指定的委托，该方法使用 ACT 和信用机构返回的应答创建银行利率查询请求，并调用银行网关来发送请求消息。这一次它指定 OnBestQuote 方法作为回调委托。一旦银行网关接收到所有银行返回的利率消息，它将通过委托调用 OnBestQuote 方法并传回 ACT 实例。OnBestQuote 使用银行利率和 ACT 来创建返回给客户的应答消息，并使用 SendReply 的基类实现把应答消息发送出去。

在源代码中值得注意的一个类是 LoanBrokerTranslator。这个类提供了一些静态方法，它们有利于实现不同消息格式之间的转换。

LoanBroker 类展示了我们所做出的设计选择。除了继承 AsyncRequestReplyService 以外，这个代码没有引用与消息传递或线程相关的概念，因此代码很容易被理解。但是，贷款中介的主要的功能是由三个方法共同实现的，并且这些方法之间不存在直接的调用关系，而是通过委托相互引用的。因此，如果不清楚整个解决方案，不了解所有的外部组件，就很难理解执行的过程。

9.3.7 重构贷款中介

查看贷款中介的工作方式时，可以认识到数据与功能被分开了。我们采用了 ACT 集合，让一个 LoanBroker 类的实例模仿多个实例。尽管 ACT 很有用，但是它们似乎违背了面向对象程序设计的精神，把数据与功能分离了，而这两部分是构成对象的基础。如果我们采用更好的方式使用委托，就能重构 LoanBroker 类，以避免 ACT 的重复查找。委托实际上是类型安全的函数指针，它们指向特定的对象实例。因此，我们没有必要向信用机构网关和银行网关提供同一 LoanBroker 实例中一个方法的引用，而是可以使用委托指向特定的“过程对象”实例，这个过程对象与 ACT 对象一样，维护了当前的状态，同时还包含了贷款中介的处理逻辑。为此，要把 ACT 转变成一个名为 LoanBrokerProcess 的新类，并把消息处理程序的功能转移到这个类中：

```
internal class LoanBrokerProcess
{
    protected LoanBrokerPM broker;
    protected String processID;
    protected LoanQuoteRequest loanRequest;
    protected Message message;

    protected CreditBureauGateway creditBureauGateway;
    protected BankGateway bankInterface;

    public LoanBrokerProcess(LoanBrokerPM broker, String processID,
                             CreditBureauGateway creditBureauGateway,
                             BankGateway bankGateway,
                             LoanQuoteRequest loanRequest, Message msg)
```

```
{
    this.broker = broker;
    this.creditBureauGateway = broker.CreditBureauGateway;
    this.bankInterface = broker.BankInterface;
    this.processID = processID;
    this.loanRequest = loanRequest;
    this.message = msg;

    CreditBureauRequest creditRequest =
        LoanBrokerTranslator.GetCreditBureaurequest(loanRequest);
    creditBureauGateway.GetCreditScore(creditRequest,
        new OnCreditReplyEvent(OnCreditReply), null);
}

private void OnCreditReply(CreditBureauReply creditReply, Object act)
{
    Console.WriteLine("Received Credit Score -- SSN {0} Score {1} Length {2}",
      creditReply.SSN, creditReply.CreditScore, creditReply.History Length);
    BankQuoteRequest bankRequest =
      LoanBrokerTranslator.GetBankQuoteRequest(loanRequest, credit Reply);
    bankInterface.GetBestQuote(bankRequest, new OnBestQuoteEvent (OnBestQuote),
null);
    }

    private void OnBestQuote(BankQuoteReply bestQuote, Object act)
    {
        LoanQuoteReply quoteReply = LoanBrokerTranslator.GetLoanQuote Reply
                                    (loanRequest, bestQuote);
        Console.WriteLine("Best quote {0} {1}",
                          quoteReply.InterestRate, quoteReply.QuoteID);
        broker.SendReply(quoteReply, message);
        broker.OnProcessComplete(processID);
    }
}
```

LoanBrokerProcess 类的方法不必再引用由信用机构网关和银行网关提供的 ACT 参数，因为所有必需的信息已经存储在 LoanBrokerProcess 实例中。一旦完成处理，它将使用 SendReply 方法发送应答消息，该方法是 LoanBrokerPM 从 AsyncRequestReplyService 继承来的。接着，向 LoanBrokerPM 发出处理完成的通知。这一通知可以使用委托实现，不过我们决定使用贷款中介对象的引用来实现。

使用 LoanBrokerProcess 类可以简化主贷款中介类：

```
internal class LoanBrokerPM : AsyncRequestReplyService
{
    protected CreditBureauGateway creditBureauGateway;
    protected BankGateway bankInterface;
    protected IDictionary activeProcesses = (IDictionary)(new Hash table());

    public LoanBrokerPM(String requestQueueName,
                    String creditRequestQueueName, String credit ReplyQueueName,
                      String bankReplyQueueName,
                    BankConnectionManager connectionManager): base(request QueueName)
    {
        creditBureauGateway = new CreditBureauGateway(creditRequestQueueName,
                                                      creditReplyQueueName);
        creditBureauGateway.Listen();

        bankInterface = new BankGateway(bankReplyQueueName, connec tionManager);
        bankInterface.Listen();
    }

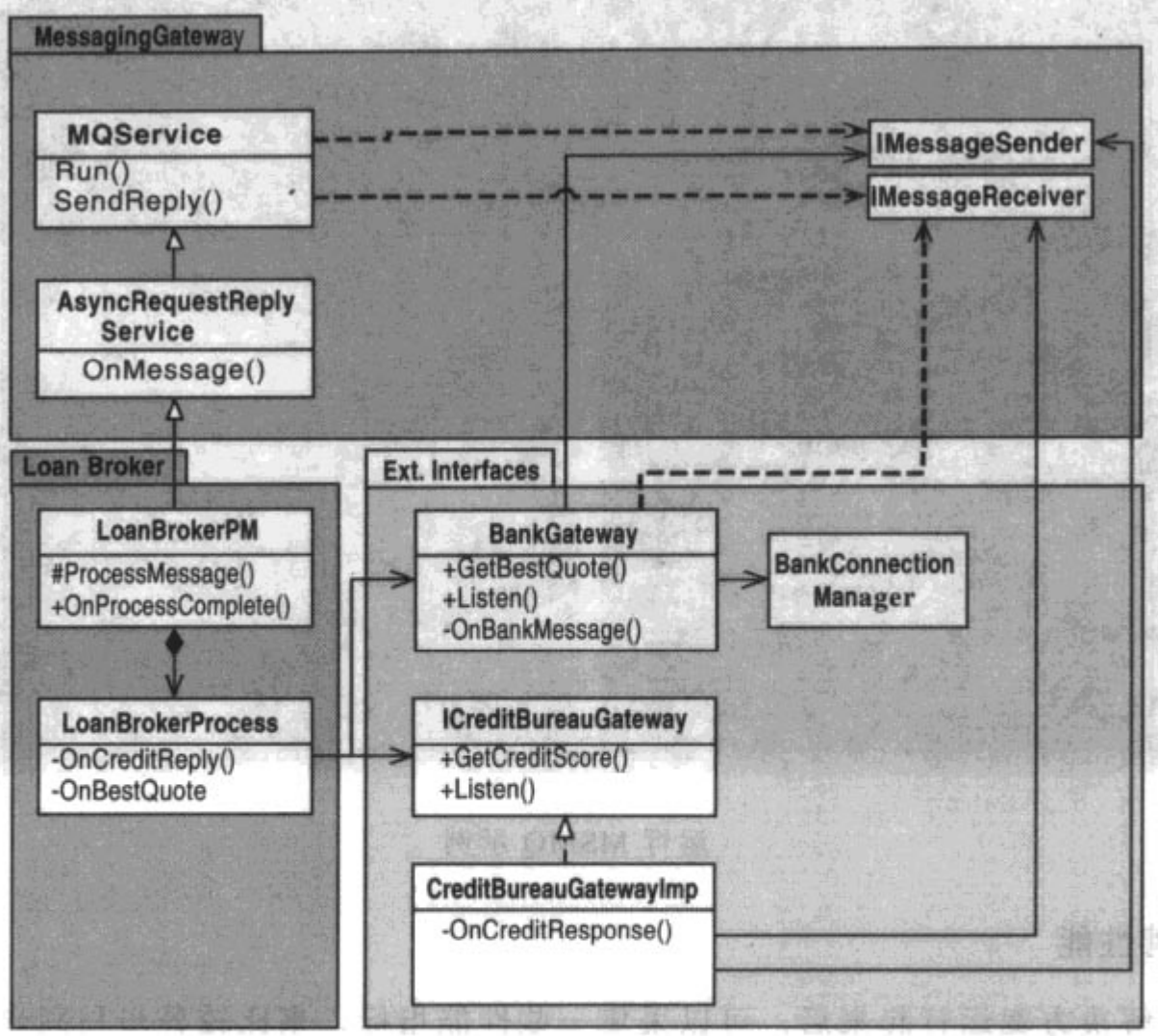
    protected override Type GetRequestBodyType()
    {
        return typeof(LoanQuoteRequest);
    }

    protected override void ProcessMessage(Object o, Message message)
    {
        LoanQuoteRequest quoteRequest;
        quoteRequest = (LoanQuoteRequest)o;

        String processID = message.Id;
        LoanBrokerProcess newProcess =
            new LoanBrokerProcess(this, processID, creditBureauGateway,
                                  bankInterface, quoteRequest, message);
        activeProcesses.Add(processID, newProcess);
    }

    public void OnProcessComplete(String processID)
    {
        activeProcesses.Remove(processID);
    }
}
```

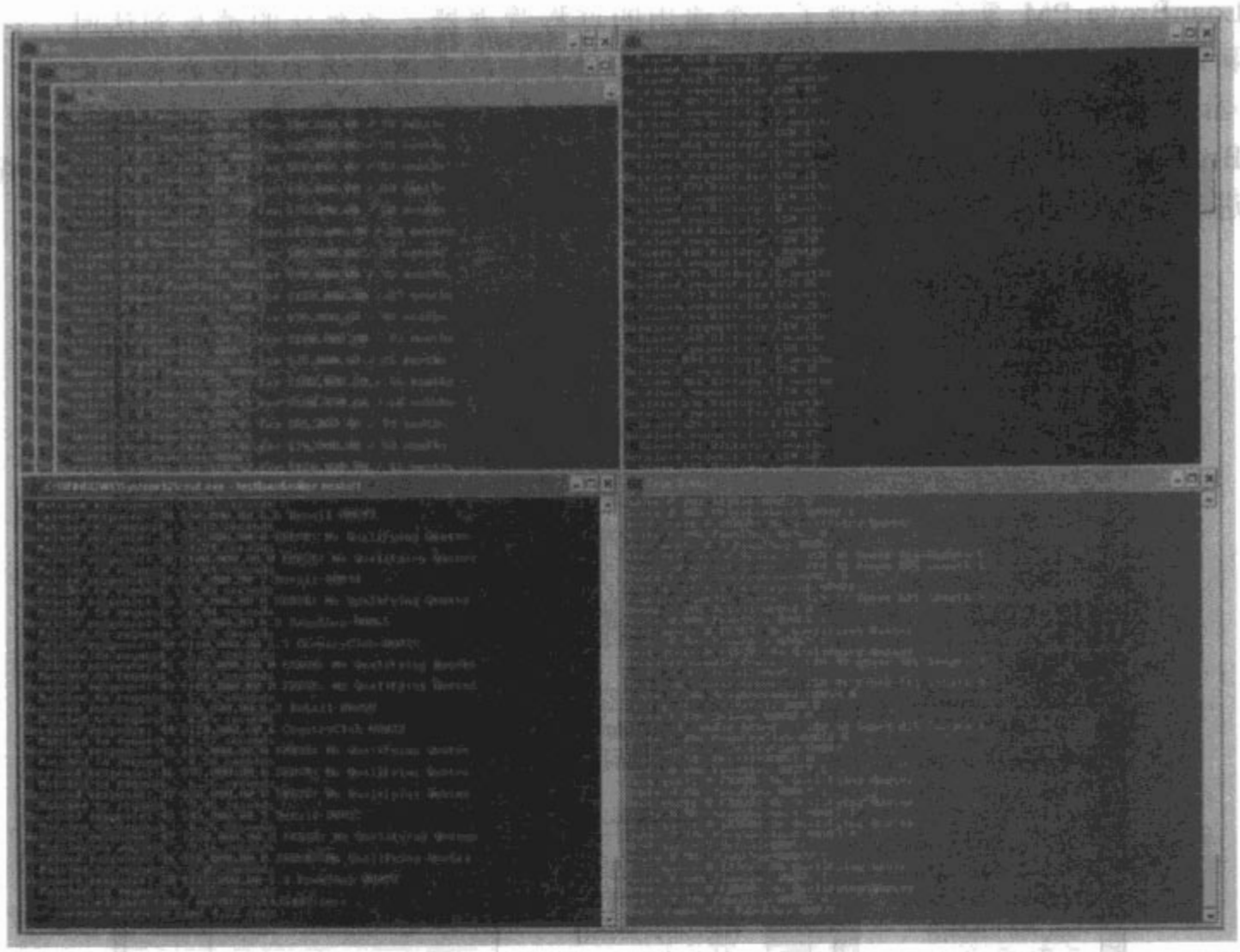
LoanBrokerPM 实际上实现了一个通用的过程管理器。当有新的消息到达时，它创建一个新的过程实例。当过程完成时，过程管理器把过程实例从活动过程列表中删除。过程管理器使用消息 ID 作为分派给每个过程实例的惟一过程 ID。现在，只要编辑 LoanBrokerProcess 类就能修改贷款中介的行为，除了传递消息对象外，这里不涉及消息传递。进行适当的封装和重构是非常重要的。下面的类图总结了贷款中介的内部结构。



贷款中介的类图

9.3.8 完成组装

现在剩下的就是测试客户端。测试客户端的设计类似于信用机构网关的设计。测试客户端可重复发送指定数量的请求，并把到来的响应消息与发出的请求消息相关联。启动所有的进程（银行、信用机构和贷款中介）后，就能演示这个例子。我们使用一些简单的主类来把各个组件作为控制台应用启动。屏幕上的内容会飞速变化，这表示的是通过系统的消息流（如下图所示）。



运行 MSMQ 示例

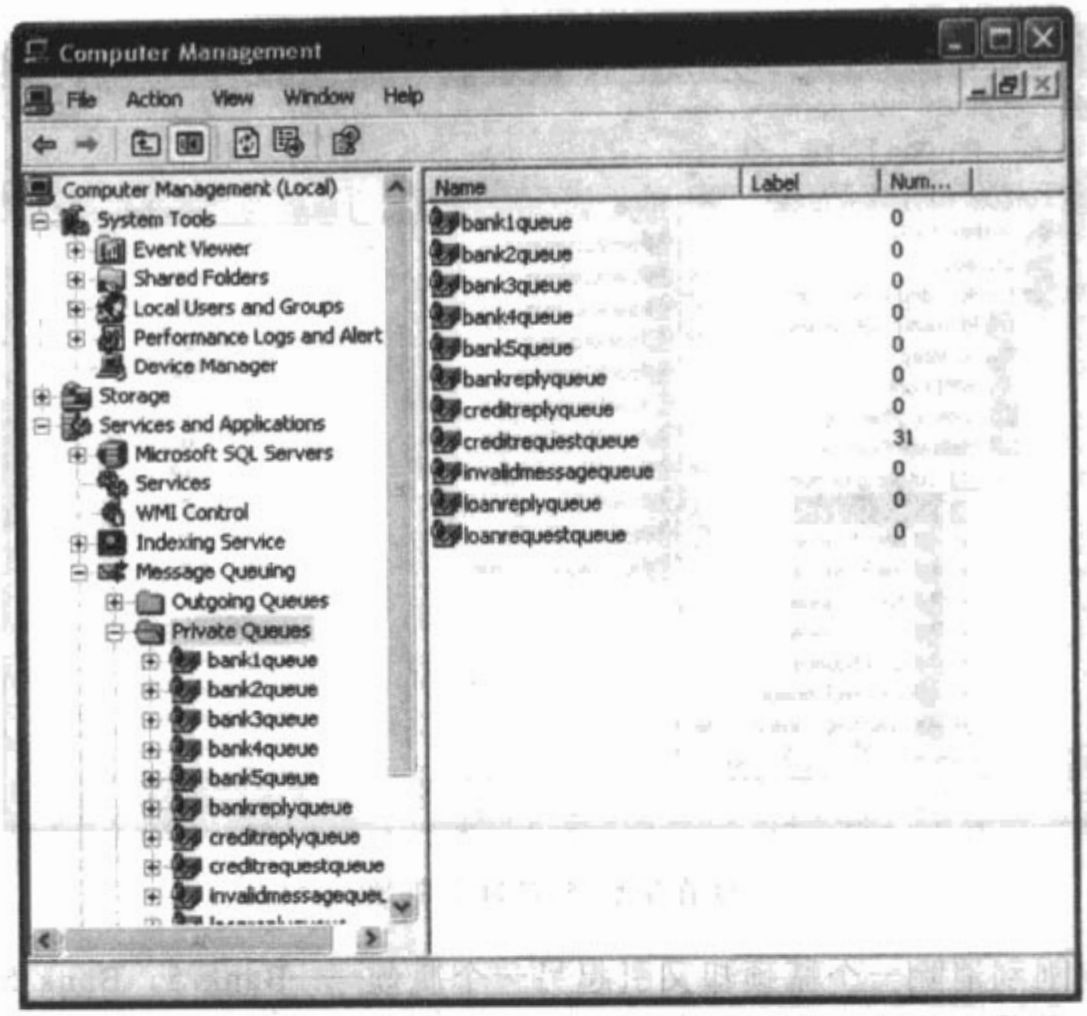
9.3.9 改善性能

完整的解决方案运行起来后，可以采集一些性能指标，来比较异步与同步解决方案的吞吐量。使用测试数据生成器，我们向贷款中介发送随机生成的 50 个请求消息。测试数据生成器给出的报告显示，客户要花 33 秒才能收到 50 个应答消息。



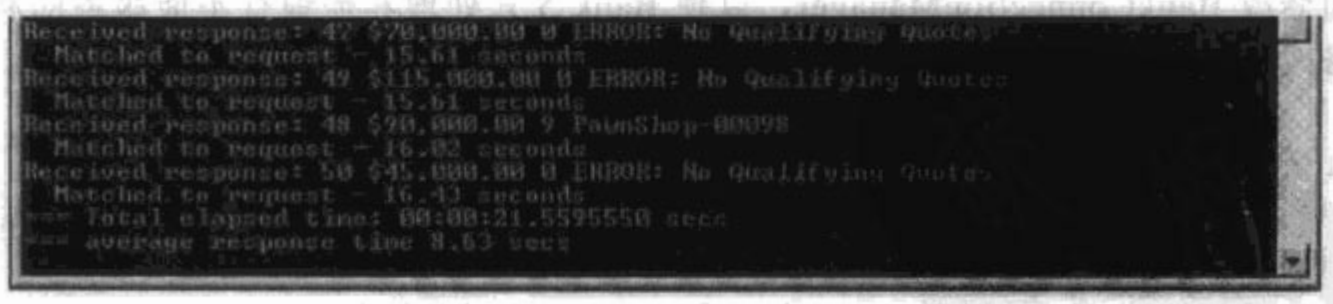
发送 50 个利率查询请求

你可能会认为处理每个请求将花费 33/50=0.6 秒 。不是这样的！贷款中介的吞吐量是在 33 秒内完成 50 个请求，但是完成其中一部分请求却花了 27 秒。为什么系统如此缓慢？让我们看一下在测试运行期间消息队列的情况：



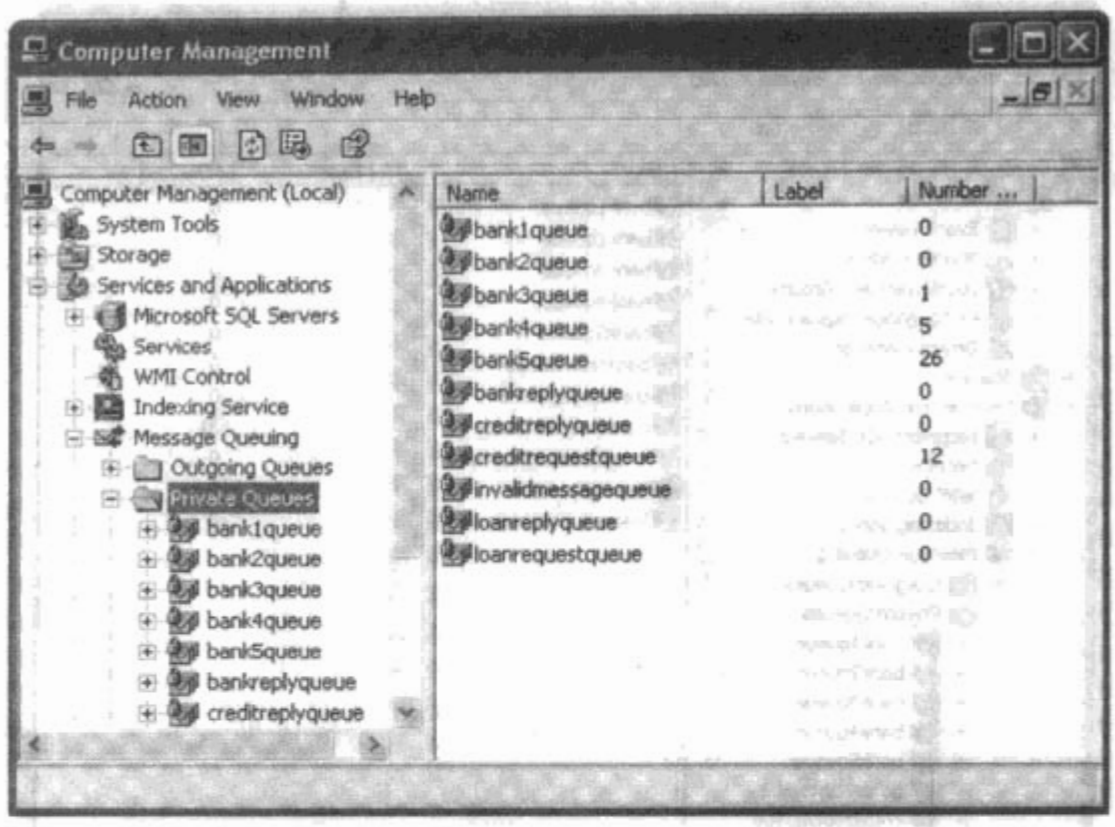
在信用请求队列中有 31 个消息等待处理

在信用机构的队列中有 31 个消息等待处理！显然，信用机构是系统的瓶颈，因为所有的利率查询请求首先要经过信用机构的处理。现在我们就可以利用松耦合带来的优点，并启动两个信用机构的实例。现在，系统中同时有 3 个信用机构服务的实例在运行。这样就能解决瓶颈问题吗？让我们来试一下。



使用 3 个信用机构实例，发送 50 个利率查询请求

现在处理全部 50 个消息的时间下降到 21 秒，其中等待响应的最长请求时间是 16 秒。客户端等待贷款请求响应的平均时间是 8.63 秒，比以前缩短了一半。看起来我们好像消除了瓶颈，但是消息的吞吐量并没有像期望的那样显著增加。不过，这个简单的示例是在同一 CPU 上运行所有的进程，因此所有进程要竞争相同的资源。下面让我们查看一下新的队列统计数据，来检查信用机构的瓶颈是否已克服了。



现在银行5成为了瓶颈

看起来我们刚刚消除一个瓶颈却又引起另一个瓶颈——Bank 5。Bank 5为什么会成为瓶颈呢？Bank 5就像一家典当行，它为所有的人提供贷款，因此Bank 5几乎会参与每一个利率查询请求。我们同样可以启动Bank 5的多个实例，但是期望典当行运行多个实例以提高吞吐量是不现实的。还有一种可行的解决方法，可以修改银行请求的路由逻辑。由于典当行要比其他的银行征收更多的贷款利率，因此，只有在其他银行都不提供利率信息的情况下，典当行的利率才是最低的。因此，如果其他银行也能提供利率查询服务，我们就不把请求路由给典当行，这样就能提高系统的效率，而且不会影响系统的整体表现。

下面修改BankConnectionManager，以使Bank 5只处理其他银行不能服务的利率查询请求。修改后的BankConnectionManager如下所示：

```
internal class BankConnectionManager
{
    static protected BankConnection[] banks =
    {new Bank1(), new Bank2(), new Bank3(), new Bank4() };
    static protected BankConnection catchAll = new Bank5();

    public IMessageSender[] GetEligibleBankQueues(int CreditScore, int HistoryLength,
                                                  int LoanAmount)
    {
        ArrayList lenders = new ArrayList();
```

```
        for (int index = 0; index < banks.Length; index++)
        {
            if (banks[index].CanHandleLoanRequest(CreditScore, History Length,
                                                  LoanAmount))
                lenders.Add(banks[index].Queue);
        }
        if (lenders.Count == 0)
            lenders.Add(catchAll.Queue);
        IMessageSender[] lenderArray = (IMessageSender [])Array.Create Instance
                                       (typeof(IMessageSender), lenders.Count);
        lenders.CopyTo(lenderArray);
        return lenderArray;
    }
}
```

运行修改后的代码，产生的结果如下图所示。

```
Received response: 43 $45,000.00 6.7 Retail-00023
  Matched to request - 6.01 seconds
Received response: 48 $50,000.00 0 ERROR: No Qualifying Quotes
  Matched to request - 5.70 seconds
Received response: 49 $30,000.00 7.4 Retail-00024
  Matched to request - 5.91 seconds
Received response: 50 $110,000.00 8.8 PawnShop-00034
  Matched to request - 6.29 seconds
=== Total elapsed time: 00:00:12.3232948 secs
=== average response time 3.68 secs
```

使用 3 个信用机构实例和修改后的 BankConnectionManager 发送 50 个利率查询请求

测试的结果显示所有 50 个请求能在 12 秒内处理完，所需的时间等于以前的一半。更重要的是，现在服务一个利率查询请求的平均时间不到 4 秒，处理速度比最初的版本提高了四倍。这就证明了使用接收表实现预测型路由的优势。由于贷款中介能够控制路由，我们不需要修改外部系统，就能随心所欲地控制路由逻辑的“智能”程度。其缺点是贷款中介会越来越依赖于对内部系统的了解。例如，最初版本的 BankConnectionManager 对所有的银行一视同仁，而修改后的版本要知道 Bank 5 是一个能服务所有请求的提供者（而且提出的利率最高），因此只能在没有其他可选银行的情况下才能与之交互。如果 Bank 5 能提供更好的利率，客户端也许就不能再获得最佳利率了。

屏幕显示的内容还表明响应消息不一定按请求发出的顺序到达。测试客户端在接收到请求 43 的响应后立即就收到了请求 48 的响应。由于我们不会遗漏任何响应消息，这表示测试客户端在接收到 43 的响应前已经接收到了 44～47 的响应。这些请求是如何超过 43 号的呢？看起来请求 43 被路由给了通用零售银行（General Retail Bank），即 Bank 3。仅次于典当行，该银行有次低的限制性选择标准，因此与其他银行相比，请求消息更有可能淤积在这里。如果请求 44～47 与通用零售银行的标准不符，那么即使银行网关已经接收到这些请求的响应，请求 43 可能还在 bank3Queue 中。由于我们的贷款中介完全是事件驱动的，

它一接收到所有的银行利率消息就会返回贷款请求的应答，因此，如果请求 44 的利率消息在请求 43 的利率消息前到达，贷款中介将首先发送请求 44 的应答消息。由此可见关联标识符在消息中的重要性，如果响应消息不是按顺序到达的，测试客户端可以通过关联标识符把响应消息与请求消息匹配起来。

要对基于消息的异步系统进行调优，这是一项非常复杂的任务。我们的例子介绍了一些最基本的技术，可以识别并解决系统瓶颈。但是，即使是这个简单的例子也说明要更正一个问题（信用机构瓶颈）可能还会引发其他问题（Bank 5 瓶颈）。通过这个例子我们也看到异步消息传递和事件驱动的消费者的优势。采用异步的解决方案可以在 12 秒内处理 50 个利率查询请求，而采用同步的解决方案可能要花 8～10 倍的时间！

9.3.10 对测试的几点说明

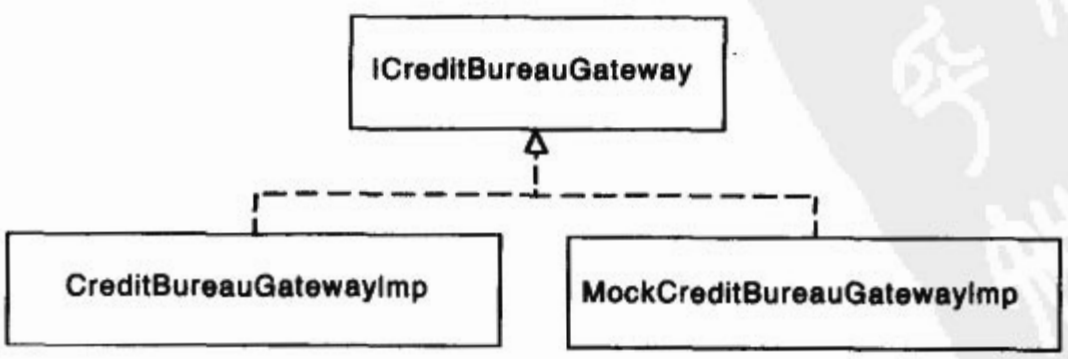
贷款中介的示例说明，一个简单的应用一旦变成分布式、异步、事件驱动的应用，就会变得相当复杂。我们目前用了十多个类，而且到处使用委托来实现事件驱动的异步消息处理。复杂性的增加还意味着出现问题的风险在增加。异步性使得这些问题很难再现，而且很难排除，因为它们与特定的暂时性条件相关。由于存在这些额外的风险，消息传递解决方案必须经过深入的测试。关于如何测试消息传递解决方案可以另外写一本书，但是我们目前只想提出一些简单可行的测试建议，归纳起来就是以下三条原则：

- 使用接口和实现类，将应用与消息传递实现分离。
- 在把业务逻辑插入到消息传递环境中之前，采用单元测试用例测试业务逻辑。
- 提供消息传递层的模拟实现，以便进行同步测试。

将应用与消息传递实现分离

测试一个应用要比测试多个通过消息传递通道连接起来的分布式应用简单得多。在一个应用中，我们可以跟踪整个执行路径。不需要复杂的启动过程来调度所有组件，而且不用在每次测试前清空通道（见“通道清理器”[xxx]）。有时，在测试其他功能时，取出一些外部功能建立测试桩可能很有用。例如，测试银行网关时还要建立信用机构网关的测试桩，而不是真正地向外部信用机构进程发送消息。

在一个应用内部，如何才能获得测试带来的好处，同时尽可能降低对应用代码造成的影响？可以把消息传递网关的实现与接口定义分离，这样就能提供接口的多个实现。



将信用机构的接口和实现分离

由于我们已经把所有与消息传递相关的逻辑封装在信用机构网关的内部，因此可以定义一个非常简单的接口：

```
public interface ICreditBureauGateway
{
    void GetCreditScore(CreditBureauRequest quoteRequest,
                        OnCreditReplyEvent OnCreditResponse, Object ACT);
    void Listen();
}
```

例如，我们可以创建一个模拟的信用机构网关实现，它不与任何消息队列具体连接，而是调用 GetCreditScore 方法中的指定委托。这个模拟实现像真正的信用机构一样包含了相同的逻辑，所以贷款中介的其余部分完全不知道它们之间的差别。

```
public class MockCreditBureauGatewayImp : ICreditBureauGateway
{

    private Random random = new Random();

    public MockCreditBureauGatewayImp()
    { }
    public void GetCreditScore(CreditBureauRequest quoteRequest,
                               OnCreditReplyEvent OnCreditResponse, Object ACT)
    {
        CreditBureauReply  reply = new CreditBureauReply();
        reply.CreditScore =  (int)(random.Next(600) + 300);
        reply.HistoryLength = (int)(random.Next(19) + 1);
        reply.SSN = quoteRequest.SSN;
        OnCreditResponse(reply, ACT);
    }

    public void Listen()
    { }
}
```

使用单元测试用例测试业务逻辑

CreditBureau 类的具体实现清楚地展示了如何把消息相关功能（封装在基类中）与业务逻辑（在我们的例子中，业务逻辑缩减为一个随机数生成器）相分离。在实际应用环境中，业务逻辑肯定要更复杂一些。因此，很有必要把 getCreditScore 方法和 getCreditHistoryLength 方法转移到另一个类中，使之对消息传递层没有任何依赖（尽管不太明显，但继承确实会使子类与基类和其他相关类之间存在着依赖）。然后就可以使用单元

测试工具，如 nUnit（http://www.nunit.org）编写测试用例，而不必考虑消息传递问题。

提供消息传递层的模拟实现

ICreditBureauGateway 的模拟实现很简单，也很有效。但是它替代了所有与信用机构网关相关的代码，因此要对 CreditBureauGatewayImp 类单独测试。如果想消除对消息队列的依赖（以及相关的性能影响），同时还能执行 CreditBureauGatewayImp 类中的代码，可以使用 IMessageReceiver 和 IMessageSender 接口的模拟实现。一个简单的模拟实现如下所示：

```
public class MockQueue: IMessageSender, IMessageReceiver
{
    private OnMsgEvent onMsg = new OnMsgEvent(DoNothing);

    public void Send(Message msg){
        onMsg(msg);
    }

    private static void DoNothing(Message msg){

    }

    public OnMsgEvent OnMessage
    {
        get { return onMsg; }
        set { onMsg = value; }
    }

    public void Begin()
    {

    }

    public MessageQueue GetQueue()
    {
        return null;
    }

}
```

从上面的代码可以看到，Send 方法没有使用任何消息队列就立即触发了 onMsg 委托。为了使用这个模拟队列测试信用机构网关，必须确保能响应一个类型正确的消息。不能简单地把请求消息传递回来作为应答消息。这里没有给出具体的实现，但是如果使用预制的应答消息，实现会非常简单。

9.3.11 此例的局限性

从这一节可以看出，即使是一个简单的消息传递系统（贷款中介实际上只需执行两步操作：获得信用度，以及获得最佳的银行利率），也会因为其异步特性和组件之间的松耦合而变得相当复杂。但是，我们仍然可以采用很多简化的手段使这个示例更可管理。不过，这个例子并没有解决以下问题：

- 错误处理
- 事务
- 线程安全

这个例子没有提供可管理的错误处理机制。目前，组件只是把消息输出到各个控制台窗口中，在生产系统中这是不适合的。对真正的实现来说，错误消息应该路由到一个中央控制台，由它采用统一的方式通知操作人员。第 11 章“系统管理”中介绍的系统管理模式（如“控制总线”[xxx]）能够解决这些需求。

这个例子也没有使用事务性队列。例如，如果有 4 个利率请求消息要发送给银行，但是只发送了其中两条，MessageRouter 就崩溃了，那么其中一些银行将处理利率请求，而其他的银行不会。同样地，如果贷款中介在接收到所有银行返回的利率响应消息后还没有来得及向客户返回应答就崩溃了，那么客户端将永远得不到应答。在真实的系统中，类似的操作必须封装到一个事务中，从而直到相应的输出消息已经发送后才消费输入消息。

贷款中介的具体实现只在一个线程中执行，因此不用担心线程安全问题。例如，入站消息队列（由 MessageReceiverGateway 隐藏）的 BeginReceive 方法只在前一消息已经处理完后才会调用。对我们这个示例应用来说，可以这么做（这比同步实现快很多），但是对有很高吞吐量要求的环境来说，要使用消息分派器管理多个执行者（performer）线程。

9.3.12 小结

通过这一节的介绍，我们了解了使用异步消息队列和 MSMQ 实现贷款中介应用的整个过程。之所以没有回避详细的实现细节，目的就是为了揭示出建立异步消息传递应用时要真正面对的问题。我们把重点更多地放在设计方案的选择上，而不是特定于开发商的消息传递 API，因此即使是非专业的 C#开发人员也能通过这个例子掌握到重要的知识。

这个例子提醒我们，即使实现一个简单的消息传递应用也是很复杂的。许多做法（如调用一个方法）在单个应用中是很可行的，但是在使用异步消息传递时，可能需要编写大量的代码。幸运的是，设计模式为我们描述设计选择方案提供了一种通用的语言，而不必采用开发商专用的术语。

9.4 使用 TIBCO ActiveEnterprise 的异步实现方案

作者：Michael J. Rettig

前面的两个贷款中介实现使用了能提供基本消息通道功能的集成框架。例如，Axis 和

MSMQ 都提供了向消息通道发送消息以及从消息通道接收消息的 API，而应用必须完成其他的所有工作。之所以选择这种实现方式，就是为了说明如何采用公共可用的 Java 或 C# 库从头建立集成解决方案。

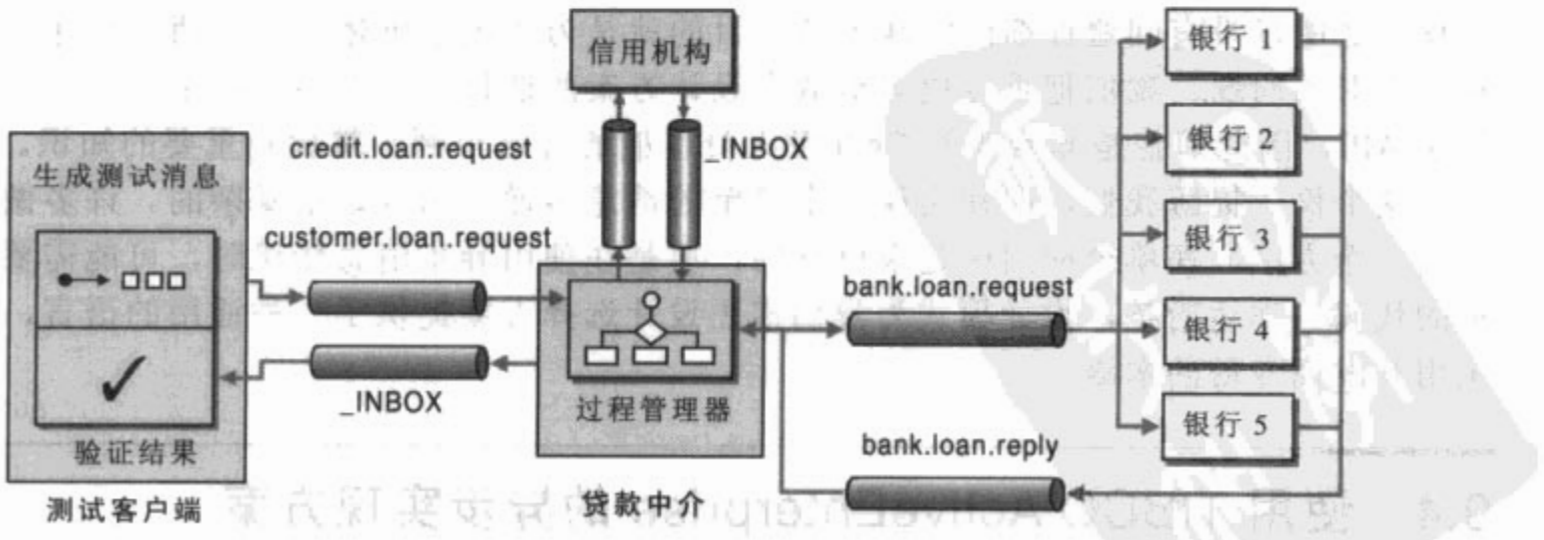
许多商业 EAI 产品套件为优化开发集成解决方案提供了更强大的功能。这些产品套件通常包括可视化的开发环境，支持消息转换器和过程管理器的拖放配置。许多开发套件还提供了复杂的系统管理和元数据管理功能。我们选择 TIBCO ActiveEnterprise 集成套件来实现这个例子。和前面的具体实现一样，我们把重点主要集中在设计决策和设计方案的选择上，只是在必要的情况下才介绍这个产品特定的一些语言。因此，即使读者以前没有接触过 TIBCO ActiveEnterprise，也能从这一节获得帮助。如果你对产品信息或开发商的信息感兴趣，请访问 http://www.tibco.com。

该示例的具体实现在设计上与使用拍卖方式的分散收集方法不同。它使用发布-订购通道代替接收表，这样一来，贷款中介就能把利率查询请求发送给任何一家银行。采用这种类型的分散收集器模式，可以针对未知数量的监听者完成动态的请求/应答。另外，贷款中介组件的实现使用了 TIBCO 过程管理器工具提供的业务过程管理功能。

9.4.1 解决方案的体系结构

我们的应用是一个简单的银行利率查询请求系统。客户向贷款中介的接口提交利率查询请求。贷款中介首先获得信用度，准备好请求消息，然后向多家银行查询利率。一旦获得银行返回的利率信息，就从中选出最佳的利率，并返回给客户（如下图所示）。

系统的客户端希望贷款中介能提供同步的请求/应答接口，客户端发送利率查询请求，然后等待贷款中介返回应答消息。贷款中介也使用请求/应答接口与信用机构通信，获得信用度。一旦收到最初的客户端请求，贷款中介可以选择在分布式的同步外观下完成异步的操作。这样，贷款中介就可以利用拍卖方式的分散收集器以及发布-订购通道从多家银行获得银行的利率信息。



TIBCO 贷款中介解决方案的体系结构

拍卖序列从向发布-订购通道 bank.loan.request 发布利率查询请求消息开始，这样任何

感兴趣的银行都可以监听消息并提供自己的利率。提交应答的银行数量是未知的，并且这个数量会随着每个利率查询请求消息的不同而不同。拍卖是从向银行叫卖开始，然后按预先定义好的时间在 bank.loan.reply 通道上等待响应消息。每接收到一个投标消息，超时计时器就被复位，以便给其他银行留出投标的时间（如果可能）。在这种情况下，其他银行的投标是公开的，因此别的银行可监听其他银行的投标，如果需要的话，也可以参与竞标。

贷款中介把利率查询的请求消息广播给未知数量的接收者。这与接收表完全不同，接收表是按照预先定义好的银行列表发送请求消息。这一点从聚合器的完成条件可以反映出来。和以前的实现不同，聚合器不再等待每家银行返回应答，而只是按照超时条件结束拍卖。当拍卖超时的条件满足时，聚合器将忽略任何接收到的响应消息。如果在指定的超时间隔内没有接收到任何一家银行返回的应答，聚合器将向客户端返回响应消息，指出它没有获得利率信息。



描述过程管理器行为的活动图

在这个实现中，内容扩充器和聚合器的功能实现在过程管理器组件中。因此，解决方案的结构图没有给出这些组件之间交互的细节，因为它们都嵌在一个组件中。相反，我们要查看表示过程模板定义的活动图，该过程模板由过程管理器使用。从这个图中可以看出，贷款中介提供了多个功能。该图很容易转换为一个过程图，从而以图表方式显示事件的准确顺序和决策路径。

9.4.2 实现工具集

为了说明这个解决方案，我们要介绍一些有关 TIBCO 产品套件的基本概念。实现 TIBCO 解决方案时，针对一个特定的问题一般需要评测不同的工具，因为一个问题往往能由多个不同的工具解决。要选出最好的工具比较困难。下面只介绍与构造示例实现相关的那些 TIBCO 特性：

- TIB/RendezVous 传输
- TIB/IntegrationManager 过程管理工具
- 用于 TIBCO 元数据管理的知识库

TIB/RendezVous 传输

TIBCO 消息传递套件的核心是 TIB/RendezVous 传输层。RendezVous 提供了消息传递机制，可以把 TIBCO 消息发送到信息总线，或者从信息总线接收 TIBCO 消息。TIBCO 支持大量的传输协议，包括（但不限于）JMS、HTTP、FTP 和电子邮件协议。RendezVous 可以为本例中使用的消息提供传输支持。它既支持同步和异步消息的传输，也支持点对点通道和发布-订购通道。每个通道可以配置为支持不同的服务等级：

- 可靠的消息传递（RV）——提供高性能的消息传递，但是存在丢失消息的实际风险。
- 认证消息（RVCM）——至少传送一次。
- 事务性消息传递（RVTX）——保证一次而且只有一次的传送。

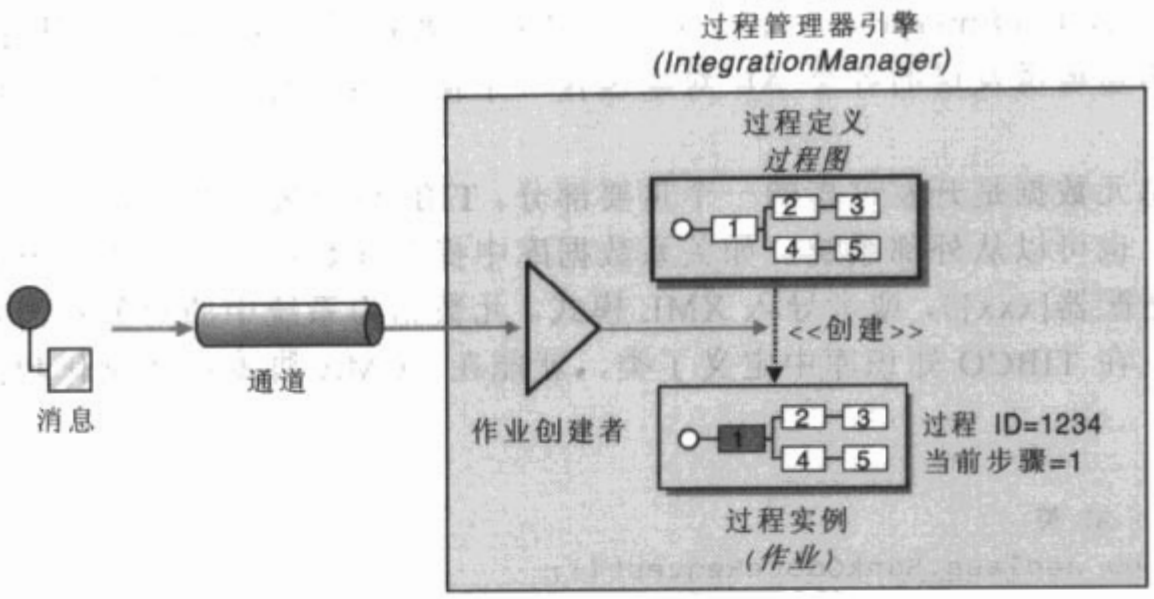
类似于本书给出的 MSMQ 示例，TIBCO 提供了使用 Java 或 C++创建消息传递解决方案的开放的 API，同时还提供了许多简化开发过程的工具。

TIB/IntegrationManager 过程管理器工具

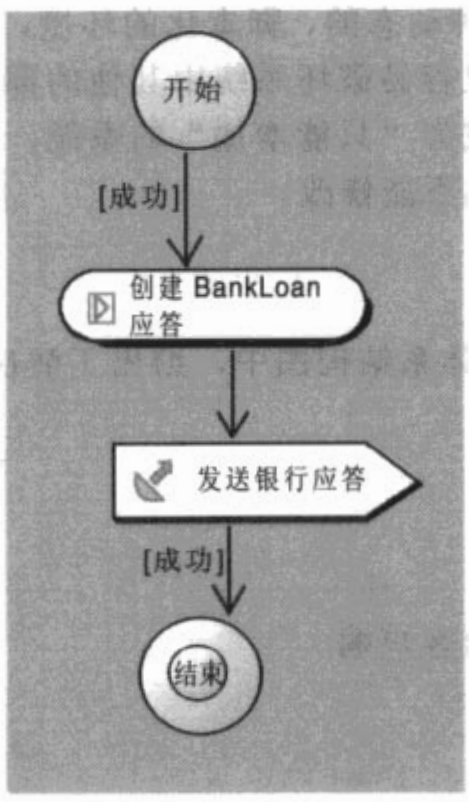
TIB/IntegrationManager 是一个 TIBCO 开发工具，它包括一个用于设计工作流解决方案的丰富的用户界面，还包括一个执行该解决方案的过程管理器引擎。GUI（图形用户界面）提供了大量的配置选项、工作流选项和实现选项，它们存储在 TIBCO 知识库中。TIBCO 使用知识库作为系统中集中的配置组件，其中包括所有的元数据、工作流和定制代码。

该解决方案的工作流组件比较特别，这也是与代码级解决方案不同的地方。对贷款中介示例来说，TIB/IntegrationManager 提供了异步和同步消息传递，以及基本的工作流，包括服务器会话状态[EAA]。对我们来说，可以把 TIB/IntegrationManager 分解为三个部分：通道、作业创建者和过程图。

TIB/IntegrationManager 中的每个过程创建一个“作业”（或过程实例），它为维持状态提供了一个集中的会话对象。该对象提供了一个插槽式环境，并提供了获取和存放操作来存储对象，还为与会话交互提供了一些工具方法。通过简单的 GUI，TIBCO 开发人员能创建过程定义（在 TIBCO 中称为过程图），指定作业执行的任务序列。过程定义类似于 UML 风格的活动图，它由任务以及连接任务的转移线组成（如下图所示）。但是，该图只提供了骨架，在每个活动（下图中的方框）的背后还有大量的配置和代码。对我们的例子来说，需要提供一些处理贷款的代码。TIB/IntegrationManager 使用 ECMAScript（一般称为 JavaScript）作为底层脚本语言。



TIB/IntegrationManager 组件



TIB/IntegrationManager 过程图示例

一个典型的过程图包含一系列集成任务。这些任务中包括控制任务，如分叉、同步条或决策点，还包括发送和接收消息的信号任务，以及像数据转换、路由和系统集成这样的执行任务。前面的图中包括两个任务：ECMAScript 任务执行一些自定义的逻辑，信号输出任务把消息发布到通道中。任务转移可以包含一些逻辑，从而按消息的内容或其他标准选择特定的路由。

管理元数据的 TIBCO 知识库

集成和消息传递通常需要某种自描述的数据（见第 8 章“消息转换”的“引言”部分）。

TIBCO 定义了 ActiveEnterprise（AE）对象之类的元数据类，这些对象存储在 TIBCO 知识库中。通过消息通道发送的各个 AE 对象包含一个格式指示符，用于指出该消息遵循的类定义。

管理消息元数据是开发过程的一个重要部分。TIBCO 开发人员可以在开发环境中直接定义元数据，也可以从外部系统，如关系数据库中提取元数据（使用“通道适配器”中介绍的元数据适配器[xxx]），或者导入 XML 模式。元数据为系统中的对象和消息提供了明确的契约。一旦在 TIBCO 知识库中定义了类，就能在 ECMA 脚本中实例化和操纵这些类的对象了：

```
//实例化 TIBCO AE 类
var bank = new aeclass.BankQuoteRequest();
bank.CorrelationID = job.generateGUID();
bank.SSN = job.request.SSN;
```

但是，请记住 TIBCO 是一种动态的、脚本化的环境，它只提供了有限的编译时类型检查。如果修改元数据的定义，很容易破坏系统中其他的部分。因此，如果测试和开发步骤不当，基于消息的系统很容易成为“只能增加”的系统，也就是说，为了避免破坏系统，只能向系统中增加元数据而永远不能修改。

9.4.3 接口

在第 9.4.1 节的解决方案的体系结构图中，给出了解决方案所需的服务。

贷款中介

- 接收最初的请求
- 获取信用度
- 与银行进行贷款竞标
- 把最佳的贷款利率返回给客户端

信用服务

- 根据 SSN 提供信用度

银行

- 根据信用等级和贷款额度提交利率信息

每个服务均可通过一个外部接口访问。对每个这样的接口来说，必须做出以下设计考虑：

- 会话方式：同步方式或异步方式
- 服务质量

接口的会话方式已经由解决方案的体系结构预先确定。贷款中介和信用服务都需要同步接口，而与银行的通信完全是异步的。

消息传递解决方案的服务等级会变得相当复杂，尤其是在特定环境中，如果要求失败恢复，情况会更为复杂。幸运的是，贷款示例的服务等级只是一个简单的解决方案。在出现故

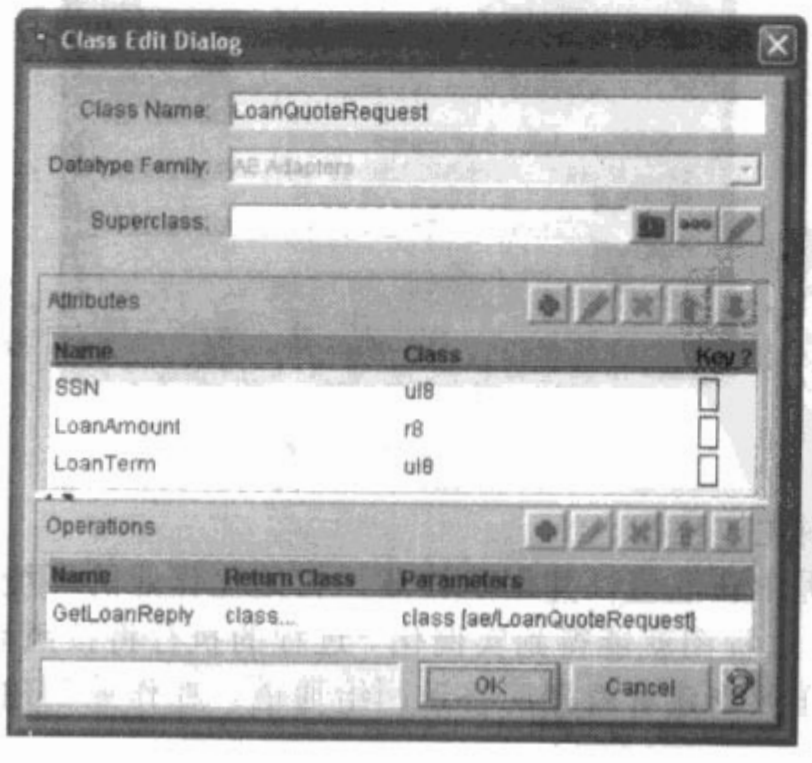
障时（超时、消息丢失、系统当机），最初的请求可以重新提交（贷款中介是一个幂等接收者[xxx]）。请记住我们目前只是得到一个利率信息。一般说来，做出这种假设并不一定正确。当然，必须把系统中这种类型的假设记入文档，并能被所有的参与方理解。例如，银行必须清楚利率查询请求重新提交的时间间隔是 1 秒钟。如果银行为了检测欺诈行为而跟踪客户的贷款请求，我们就必须修改解决方案，以避免发送重复的请求，例如可以使用可靠传输。

9.4.4 实现同步服务

贷款中介系统有两个同步接口，分别位于客户与贷款中介之间，以及贷款中介与信用机构之间。TIBCO 与使用请求/应答和命令消息等模式的操作实现了远程过程调用方式的消息传递。实际上，这就是底层 TIBCO 消息传递引擎的一个同步包装器。在调用 TIBCO 操作期间，可以监听通过总线传递的请求和应答消息。请求消息发布到指定的通道上（如 customer.loan.request）。消息中包含返回地址，用于为返回的应答消息指定所谓 INBOX 通道的地址。TIBCO 通过远程过程调用方式的编程模型隐藏了异步通信的细节。在 TIB/IntegrationManager 中，诸如获取信用度这样的领域操作定义为 AE 类的方法，可以由过程建模工具调用。例如，为了将贷款中介提供为一个同步接口，必须完成多个实现步骤。下面给出了信用机构服务的实现，在此就基本完成了类似的步骤。

定义 AE 类

AE 类为通过 TIB/RendezVous 通道发送的消息定义了数据格式。在 TIB/IntegrationManager 中定义类与在大家熟悉的 IDE 中创建类完全不同。AE 类由 TIB/IntegrationManager IDE 提供的一系列对话框（如下图所示）来定义。这些对话框为选择类名、设置类字段提供了帮助。类的字段可以定义为整型、浮点型或双精度型，也可由其他 AE 类组成。



定义 AE 类的属性和操作

定义 AE 类的操作

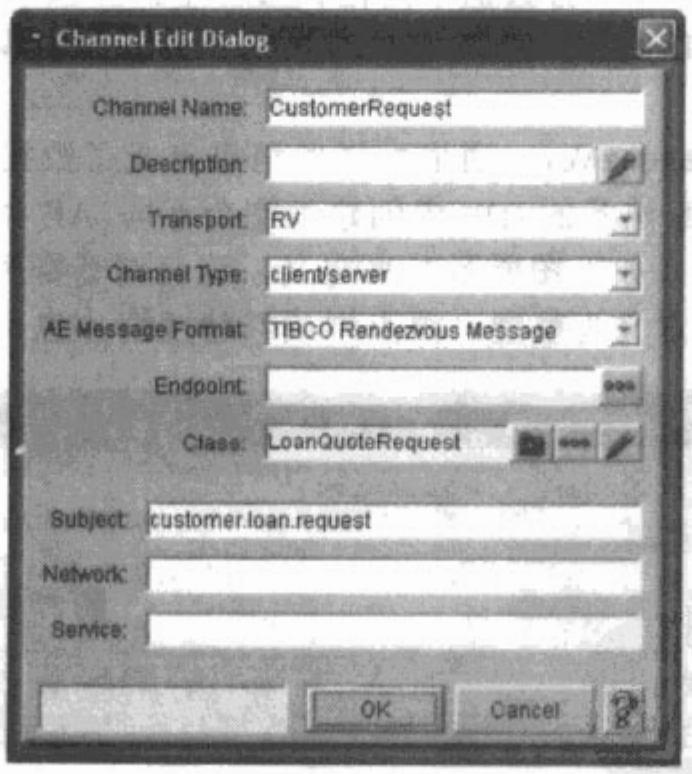
类似于增加接口方法，也可以把操作增加到 AE 类中。可以在定义中指定方法的参数和返回类型。与接口一样，AE 类方法也没有实现。使用作业创建者把通道与一个过程实例绑定时，才会具体确定实现。

创建过程图

过程图提供了操作的具体实现。这个例子中实现的操作需要一个返回参数。与在代码中编写方法不同，过程图不提供“返回”值。相反，我们要在作业中指定槽来存放返回值。槽在作业创建者中分配，而且必须记住把值正确地设置到过程图的作业槽中。过程图的具体实现将在后面讨论。

创建客户/服务器通道接收请求

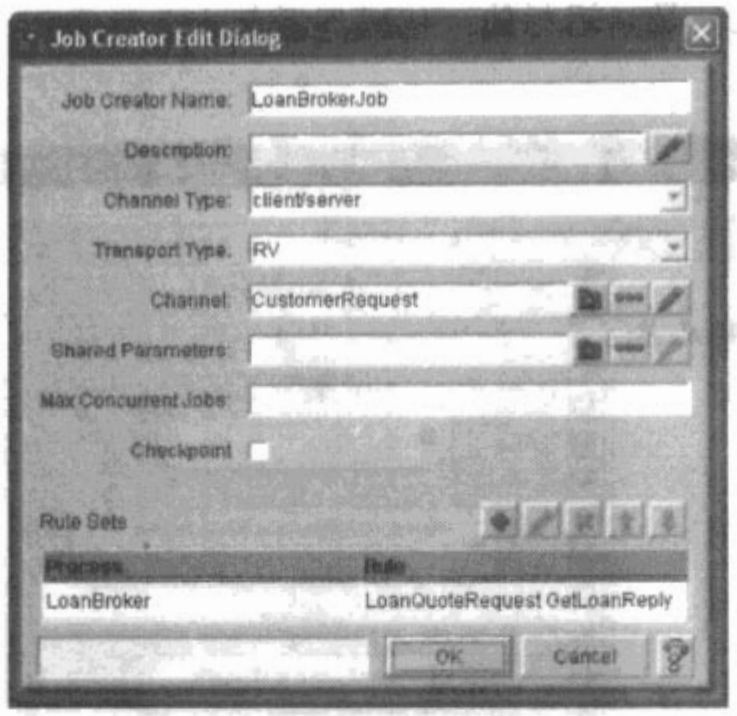
采用通道能定义传输、消息定义、服务和主体。对我们的同步操作来说，需要使用客户/服务器通道。我们可以指定第一步中创建的 AE 类。根据最初的接口定义，我们选择了 RendezVous，它能提供可靠的消息传递。为了配置这些选项，只需点击鼠标并做出正确的选择（如下图所示）。



定义通道的属性

配置作业创建者来实例化过程图

作业创建者从通道中获取值，并创建作业，初始化作业的环境槽，从而把值传递给过程图。一旦作业创建，过程图就能得到实例化。过程的执行将按照活动图指定的路径进行。一旦过程执行完毕，作业创建者就把应答返回给通道。在作业创建者对话框中可以看到操作的名字。



配置作业创建者

9.4.5 贷款中介过程

实现了同步服务之后，下面可以实现贷款中介，以把所有的服务集成在一起。下页中的图表示出贷款中介的过程。该过程定义了贷款中介组件的行为。在前面的设计步骤中，我们定义了 AE 操作、通道，以及作业创建者，当客户端向 CreditRequest 通道提交消息时，需要采用作业创建者实例化过程图。

设计方面的考虑

从设计的角度看，必须特别注意应该把哪些内容包含在过程图中。如果过程很大、很复杂，将难管理。实现过程图中相关操作的有效分离，可以避免同一个过程定义中混杂着业务逻辑和过程逻辑，这一点很重要。什么是过程逻辑，什么是业务逻辑，要明确界定并不容易。一个很好的经验是，把过程逻辑看作是外部系统之间的交互，即要连接的下一个系统是谁？如果系统不可用该怎么办？而业务逻辑更多涉及的是与业务领域相关的方面：如何处理订单？如何计算信用度？如何返回银行贷款的利率？考虑到业务代码的复杂性，实现业务逻辑的最佳选择是采用功能全面的开发语言。大多数过程管理工具，包括 TIB/IntegrationManager，都允许直接集成 Java 和其他语言。

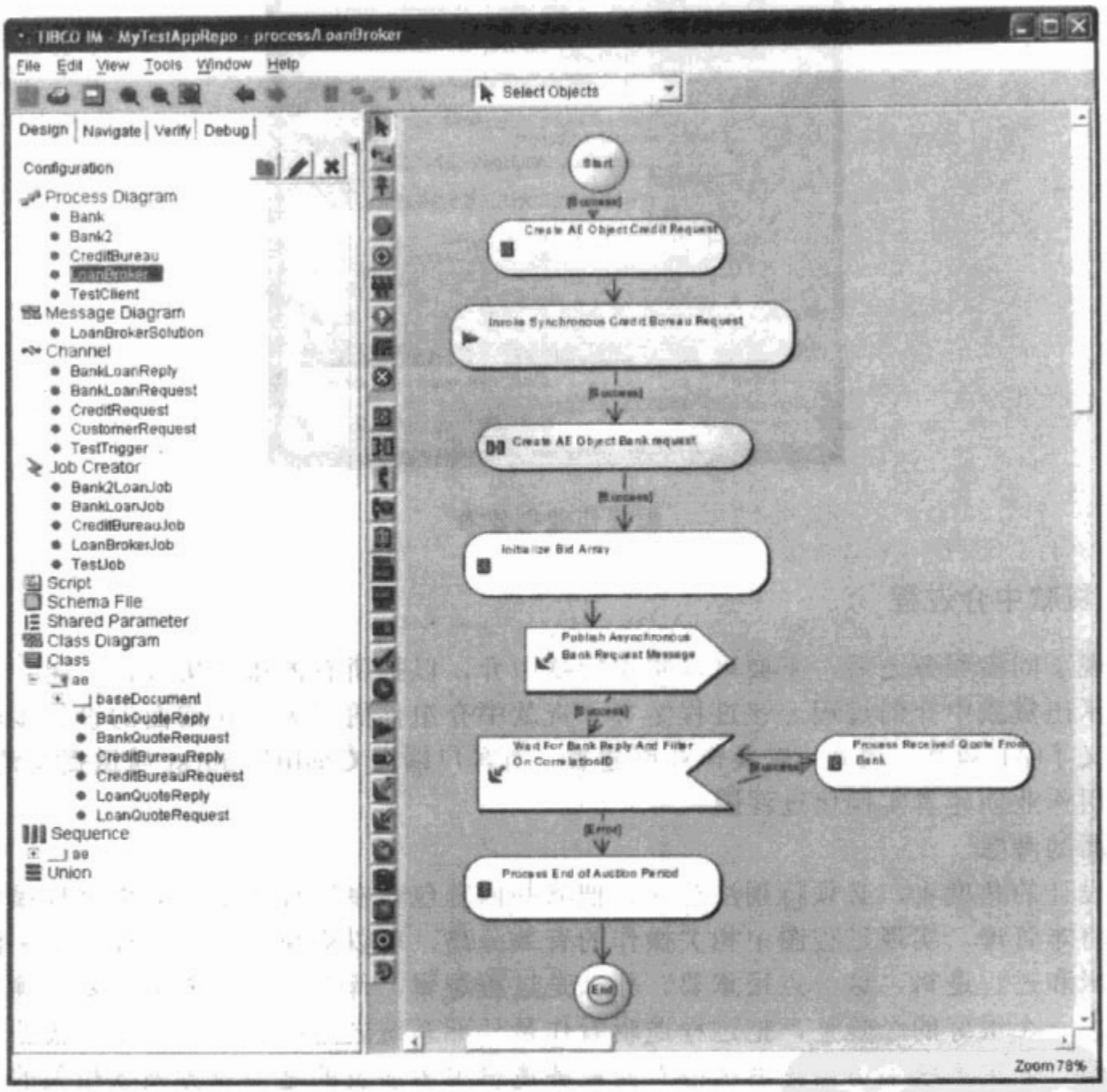
如果你熟悉 MVC（模型、视图和控制器）概念（如[POSA]），可以看到，过程图的实现采用了类似的模式。通过简单地把视图重命名为工作流，可以很简明地定义过程图实现。

1. 工作流——工作流的可视化模型。
2. 控制器——从消息总线接收事件的过程引擎，它执行过程工作流的正确组件。
3. 模型——底层的业务代码（ECMAScript、JavaScript、Java 和 J2EE 等）。

过程模型实现

下图包含有脚本执行框和完成集成动作的定制任务。使用可视化过程建模工具来建立

过程模型的最大的好处之一是，运行的“代码”看起来非常类似于我们在设计解决方案时使用的 UML 活动图。



贷款中介过程定义

由于每个任务图标可能包含具体的代码或重要的配置参数，我们将进一步详细介绍每个任务。

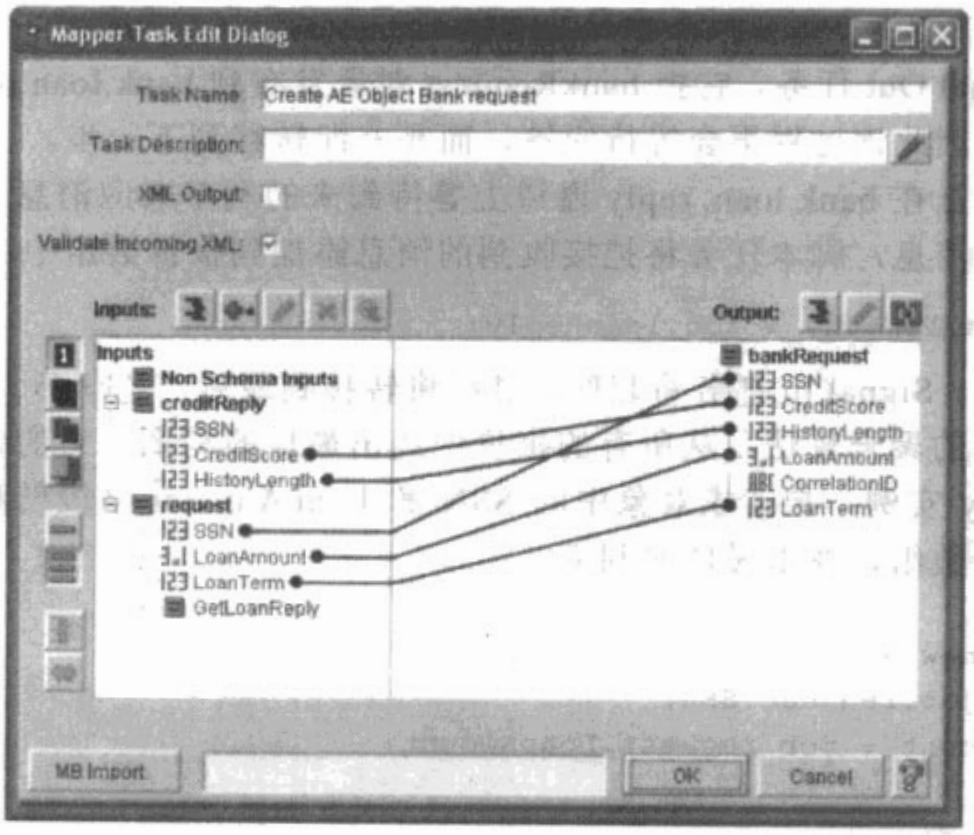
第一个框表示一个 ECMAScript，它实例化一个 AE 对象，并设置其字段的值：

```
var credit = new aeclass.CreditBureauRequest();
credit.SSN = job.request.SSN;
job.creditRequest = credit;
```

所创建的信用请求要作为下一步活动的参数，该活动会向信用机构发出同步调用。信用机构由一个单独的过程图实现，其中包含一个 ECMA 任务，这个任务要接收消息并返回

信用度。同步操作意味着，贷款中介过程将一直等待直到信用机构返回的应答消息到达。

从上图中可以看出，一旦贷款中介接收到信用机构返回的应答，就要创建另一个 AE 对象，从而向所有参与的银行发布利率查询请求。为了实现这个功能，我们使用了 Mapper 任务，以图形方式来映射来自多个数据源的数据项。Mapper 是消息转换模式的可视化实现。Mapper 任务体现了管理元数据的一个优点。由于 TIB/IntegrationManager 能够访问元数据，而这些元数据定义了每个对象的结构，因此 Mapper 可以显示出源和目标对象的结构，并能通过简单的鼠标点击以可视化的方式把它们的字段对应起来。基于输入对象中的值可以实例化一个新的对象。我们使用作业对象的 generateGUID 方法生成惟一的关联标识符，并把它赋给 bankRequest 对象的关联 ID 字段。正如我们后面将要看到的，关联 ID 字段对于同时处理多个贷款请求非常重要。



采用可视化 Mapper 任务创建银行请求消息

除了可以使用可视化 Mapper 外，还可以使用 ECMAScript 任务完成相同的工作。与之等价的代码如下所示。从中可以看到如何创建一个类型为 BankQuoteRequest 的新对象，并根据源对象设置每个字段的值：

```
var bank = new aeclass.BankQuoteRequest();
//创建惟一标识该事务的ID。
//以后要使用该ID过滤应答消息。
bank.CorrelationID = job.generateGUID();
bank.SSN = job.request.SSN;
bank.CreditScore = job.creditReply.CreditScore;
bank.HistoryLength = job.creditReply.HistoryLength;
```

```
bank.LoanAmount = job.request.LoanAmount;
bank.LoanTerm = job.request.LoanTerm;

job.bankRequest = bank;
```

到底是使用可视化 Mapper 任务还是使用 ECMAScript 任务来实现消息转换器的功能，这是由映射的类型和复杂性决定的。Mapper 任务为源和目标对象之间的连接提供了非常直观的视图，但是如果对象有很多字段，就不利于查看。另一方面，Mapper 包含了特殊的功能，从而能把重复的字段（如数组）与单个数据项对应起来，而不必编写循环代码。

接着是一个非常简单的 ECMAScript 任务，它要实例化一个报价数组，该数组保存到来的银行响应消息。这个脚本只包含一行代码：

```
job.bids = new Array();
```

下一项是 Signal Out 任务，它把 bankRequest 对象发布到 bank.loan.request 通道中。这是一项异步操作，因此该过程不会等待应答，而是立即转移到下一步。

接下来的任务是在 bank.loan. reply 通道上等待到来的利率响应消息。如果能在指定的超时期限内接收到消息，脚本任务将把接收到的消息添加到报价数组中：

```
job.bids[job.bids.length] = job.loanReply;
```

一旦结束竞标，Signal In 任务将超时，过程将转移到最后一项任务。这个 ECMAScript 任务实现了聚合器的聚合算法，从所有的报价中选出最佳的利率。它的代码创建了一个新的 LoanQuoteReply 实例，把请求对象中的 SSN 和 LoanAmount 字段转移到应答对象中，并且迭代处理报价数组，找出最佳的利率。

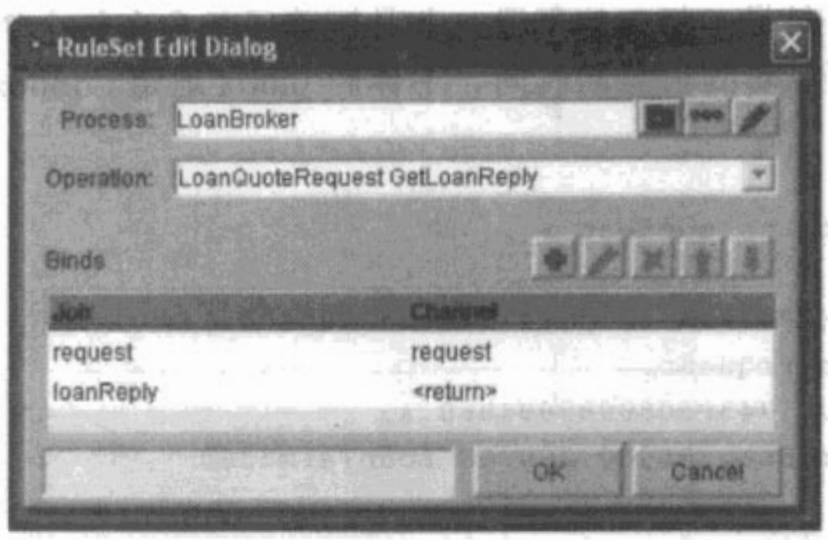
```
var loanReply = new aeclass.LoanQuoteReply();
loanReply.SSN = job.request.SSN;
loanReply.LoanAmount = job.request.LoanAmount;

var bids = job.bids;
for(var i = 0; i < bids.length; i++){
    var item = bids[i];
    if(i == 0 || (item.InterestRate < loanReply.InterestRate)){
       loanReply.InterestRate = item.InterestRate;
       loanReply.QuoteID = item.QuoteID;
    }
}

job.loanReply = loanReply;
```

ECMA 脚本的最后一行把贷款应答对象作为返回给客户的消息放到作业槽中。我们将贷款中介作业创建者配置为把作业的 loanReply 属性返回给客户，作为应答消息（如下图所示）。当过程完成时，作业创建者把作业的 loanReply 属性中发现的消息取出，并把消息

的值返回给客户端。



配置作业创建者的规则集，以返回最佳利率

9.4.6 管理并发的拍卖

拍卖的实现给开发造成了一些障碍，并发性更增加了问题的复杂度。为了让拍卖正常运行，我们要发布异步消息，然后在指定的时间内等待应答。但是，如果有多个拍卖同时进行，就必须确保每个贷款中介只接收到它关心的应答，而不是所有的应答。

使用关联标识符和选择性消费者，把无关的消息过滤掉，就能实现上述功能。理想情况下，我们希望把该过滤功能实现在通道中，而不是在过程实例中实现，但是这就需要在运行时动态地创建通道的主体，每个主体对应一个未完成的拍卖，而目前的TIB/IntegrationManager 在实现上还无法提供这样的技术支持。在运行时实例化过程图时，必须注册它所监听的所有主体，这样才能让过程引擎监听这些主体的消息，并根据需要把消息加入队列。如果过程图在一个主体上发布消息，然后转移到另一个状态下监听应答消息，就会带来定时 bug，而排队可以避免这种定时 bug。在异步环境下，如果转移需要花费很长时间，那么应答可能会在过程图有机会订购消息之前被接收到。因此，过程图必须把到来的消息排队，但是这样就无法支持动态的主体。按照目前所举示例的要求，在过程级过滤消息就足够了。关联标识符是分配给每个过程的惟一标识符，它会传递给每个银行过程，并包含在每个应答消息中。ECMAScript 只用一行代码就为 Signal In 任务提供了过滤功能：

```
(event.msg.CorrelationID == job.bankRequest.CorrelationID)
```

9.4.7 执行

在完成上述例子的实现后，现在就可以运行了。运行该解决方案必须启动TIB/IntegrationManager 引擎。为了测试这个解决方案，在知识库中包含了一个简单的测试客户端，它能每隔 5 秒提交一个贷款请求。另外建立了一些简单的测试桩来实现信用服务和银行服务。发布-订购通道的一个优点是，我们可以方便地监听消息总线上的所有消息，

检查消息流。我们使用了一个简单的工具将所有的消息记录到控制台。为简洁起见，我们把消息的内容省略掉，只显示相关的字段，并且删除了包含在每个消息中的格式符和跟踪信息。从日志记录中可以看到消息的时间、主体和 inbox 通道。inbox（收件箱）是同步服务所需的返回地址。

```
tibrvlisten: Listening to subject >

2003-07-12 16:42:30 (2003-07-12 21:42:30.032000000Z):
subject=customer.loan.request,
reply=_INBOX.C0A80164.1743F10809898B4B60.3,
SSN=1234567890 LoanAmount=100000.000000 LoanTerm=360

2003-07-12 16:42:30 (2003-07-12 21:42:30.052000000Z):
subject=credit.loan.request,
reply=_INBOX.C0A80164.1743F10809898B4B60.4,
SSN=1234567890

2003-07-12 16:42:30 (2003-07-12 21:42:30.092000000Z):
subject=bank.loan.request,
SSN=1234567890 CreditScore=345 HistoryLength=456 LoanAmount=100000.000000
CorrelationID="pUQI3GEWK5Q3d-QiuLzzwGM-zzw" LoanTerm=360

2003-07-12 16:42:30 (2003-07-12 21:42:30.112000000Z):
subject=bank.loan.reply,
InterestRate=5.017751 QuoteID="5E0x1K_dK5Q3i-QiuMzzwGM-zzw" ErrorCode=0
CorrelationID="pUQI3GEWK5Q3d-QiuLzzwGM-zzw"

2003-07-12 16:42:30 (2003-07-12 21:42:30.112000000Z):
subject=bank.loan.reply,
InterestRate=5.897514 QuoteID="S9iIAXqgK5Q3n-QiuNzzwGM-zzw" ErrorCode=0
CorrelationID="pUQI3GEWK5Q3d-QiuLzzwGM-zzw"
```

从测试客户端可以看到 customer.loan.request 通道中的请求消息。请求消息包含了客户的社会安全号以及所要求的贷款数量和期限（借贷 100 000 美元，期限是 360 个月）。该消息指定了测试客户端的私有应答通道_INBOX.C0A80164. 1743F10809898B4B60.3 作为返回地址，以便告诉贷款中介应该把应答消息发送到哪里。

下一个消息表示贷款中介向信用服务发送的请求，它携带了贷款中介的私有收件箱（inbox）作为返回地址。信用服务只需要社会安全号。应答消息不能被我们的日志记录工具捕获，因为_INBOX 通道是私有通道。

当贷款中介接收到信用度后，它向 bank.loan.request 通道发布一个消息。在我们的例子中，两个银行桩立即提供了自己的利率以做出响应。每个银行还为应答消息分配了一个惟一的 QuoteID，以便客户能在事后查询。拍卖周期将在几秒钟后超时。由于我们无法看

到贷款中介过程发送给测试客户端的应答消息，因此只能查看过程管理器引擎的调试记录，检查测试客户端是否收到了最低的利率信息。在这个消息中，我们能看到 CorrelationID 和类的格式指示符。

```
reply= class/LoanQuoteReply {
    SSN=1234567890
    InterestRate=5.017751017038945
    LoanAmount=100000.0
    QuoteID=5E0x1K_dK5Q3i-QiuMzzwGM-zzw
}
```

9.4.8 结论

这个解决方案中有一些方面很有意思，可以与前面的解决方案加以对比。可视化的工作流易于观察和理解，但是就像代码一样，随着实现变得越来越复杂，原本很简单的东西也会变得非常混乱。分散收集器的动态特性有利于实现发布-订购接口的解耦合。贷款中介即使不清楚订购银行的任何信息，也能发布银行贷款请求。同样地，应答消息的聚合器不必知道所要等待的应答消息数是多少。这些灵活性也有一个缺点：不正确的主体命名、开发人员的错误或者丢失的消息会导致错误，而这种灵活性可能隐藏这些错误。

可视化、GUI 驱动的 TIBCO 实现还提供了代码级的实现。尽管代码的能力和灵活性是不容置疑的，但是更加图形化的编程环境能极大地简化非常复杂的任务。集成的一个重要部分是配置，而图形化的环境特别适合这个工作。因此开发人员必须在编写代码和使用开发商提供的开发工具之间做出选择。但是，我们可以把这两种方法有效地结合起来。例如，可以使用 TIB/IntegrationManager 建立集成工作流的模型，然后使用 J2EE 会话 bean 实现领域逻辑。

出于篇幅的考虑，我们的例子非常简单，只是提供了一个易于理解的案例。真正的实现一般不会这么简单。快速开发工具有利于加快最初的开发，但是会限制你能做的设计选择。过程管理工具有利于屏蔽消息传递的复杂性，使开发人员能把注意力集中到集成任务方面，而不必担心集成后面发生的事情。但是，对消息传递基础设施的忽视会导致项目的失败，已经不乏这种失败的案例。

第 10 章

消息传递端点

10.1 引言

在第 3 章“消息传递系统”中，我们讨论了消息端点。应用只有通过消息端点连接到消息传递系统，才能发送和接收消息。作为应用设计人员，当使用 JMS 或 System.Messaging 名字空间等消息传递 API 编写程序时，也就是在开发端点代码。如果你使用的是商业中间件开发包，那么开发商已经为你提供了编写大部分代码所需的开发库和工具。

10.1.1 发送和接收模式

某些端点模式既适用于发送者，也适用于接收者。它们通常涉及如何把应用与消息传递系统联系起来。

封装消息传递代码——一般情况下，应用并不知道它是否正在使用消息传递与其他程序集成。大多数应用的代码在编写时没有考虑消息传递问题。当应用需要与其他程序集成时，应该由一个很瘦的代码层实现应用的集成功能。当采用消息传送实现集成时，该代码瘦层就是消息传递网关，它把应用与消息传递系统连接起来。

数据转换——当发送者和接收者的应用使用了相同的内部数据表示，并且它们使用的消息格式也相同，此时就不必考虑数据转换。但是，实际情况往往不是这样的。发送者和接收者要么采用的数据格式不同，要么使用了不同的消息格式（目的是为了支持其他发送者和接收者）。在这种情况下，必须使用消息传递映射器实现应用之间数据格式和消息格式的转换。

外部控制的事务——消息传递系统在内部使用了事务。从外部来看，在默认条件下，每个发送方法或接收方法的调用都在各自的事务中运行。但是，消息生产者和消费者可以选择使用事务性客户端，从外部控制这些事务，当我们需要成批处理多个消息时，或者需要与其他事务性服务一起协调完成消息传递时，可以采用这种方法。

10.1.2 消息消费者模式

其他的端点模式只适用于消息消费者。发送消息很简单。问题是何时把消息发送出去，消息中应该包含哪些内容，以及如何实现与接收者的通信——为此我们需要使用消息构造模式（见第 5 章“消息构造”）——但是消息一旦创建好，把它发送出去就很容易了。然而消息的接收却很麻烦。因此，许多端点模式解决的是消息接收问题。

对消息的消费来说，最重要的话题莫过于节流（throttling）：也就是应用控制其消息消费速度的能力。在本书的引言中介绍过，任何服务器都无法回避这样一个问题，即过量的客户请求可能使服务器超负荷。采用远程过程调用，服务器很可能会受制于客户端调用的速度。类似地，采用消息传递，服务器无法控制客户端发送请求的速度，但是服务器可以控制处理这些请求的速度。应用不一定按照消息传递系统发送消息的速度来接收和处理消息，它可以按照自己的实际能力进行处理，让消息通道把待处理的消息按照先来先服务的方式进行排队。但是，如果消息堆积得过多，并且服务器有足够的资源来更快地处理更多的消息，那么服务器可以使用并发的消息消费者，从而提高消息消费的吞吐量。通过使用这些消息消费者模式，可以帮助应用控制其消费消息的速度。

许多消息消费者模式是成对出现的，它们可以相互替代，也就是说可以采用其中任意一种方式设计端点。一个应用可以采用一种方式设计部分端点，而采用另一种方式设计其他端点，但是一个端点只能使用一种方式实现。各对模式可以相互结合，从而能为实现特定的端点提供大量选择方案。

同步或异步的消费者——开发人员可选择使用轮询消费者（Polling Consumer）或事件驱动的消费者（Event-Driven Consumer）[JMS1.1]、[Hapner]、[Dickman]。采用轮询方式能提供最佳的节流控制，因为如果服务器忙，它将无法响应更多的消息，多余的消息将排队。对事件驱动的消费者来说，只要消息一到达就会处理，因此很容易使服务器超负荷；不过，由于每个消费者一次只能处理一个消息，所以只要限制消费者的数量就能有效地控制消费的速度。

消息分配与消息抓取——这两种模式涉及一组消费者如何处理一组消息。如果每个消费者得到一个消息，那么它们就能同时处理这些消息。最简单的方法是使用竞争消费者，其中一个点对点通道对应多个消费者。每个消费者能够抓取任何消息；消息传递系统的具体实现决定了哪个消费者能得到消息。如果想控制消息到消费者的匹配过程，可以使用消息分派器，这是一个消费者，它能接收消息并把消息委托给执行者处理。应用通过限制消费者/执行者的数量，可以控制消息处理的速度（节流）。而且，消息分派器中的分派器可以实现明确的节流操作。

接受所有的消息或过滤消息——默认情况下，通过消息通道发送的任何消息都能到达监听该通道并消费消息的消息端点。但是，某些消费者可能不想消费通道中的所有消息，而只想消费有某种类型或格式的消息。这种有识别能力的消费者可以使用选择性消费者（Selective Consumer）模式来描述它想接收哪种类型的消息。然后消息传递系统只会把与接收者所需类型匹配的消息发送给接收者。

断开连接时仍能订购——对发布-订购通道来说，如果订购者对特定通道上发布的数据感兴趣，但是它目前无法连接到网络，或者因为维护而关闭，如何才能得到订购的数据呢？即使一个应用已经订购了消息，但是如果在它未与网络连接期间有消息发布，是否会错过这些消息呢？在默认情况下，订购者只有在保持连接时才能订购到所需的消息。为了防止应用错过在两次连接之间发布的消息，需要把它实现为持久订购者。

幂等性——有时相同的消息会传输多次，这是因为消息传递系统不知道消息是否已经成功传送，或者是因为消息通道为了提高性能而降低了服务质量造成的。而对消息的接收

者来说，它们一般认为每个消息只传输一次，如果重复处理相同的消息，很容易出现问题。把接收者设计为幂等接收者（Idempotent Receiver）就能妥善地处理重复的消息，并防止它们造成接收者应用出错。

同步或异步的服务——应用应该提供同步的服务调用（通过远程过程调用）还是异步的服务调用（通过消息传递），对此也很难做出选择。不同的客户端可能会采用不同的调用方法，不同的环境也可能要求采用不同的方法。很多情况下，我们不能只使用其中一种方法，而要同时采用两种调用方法。服务激励器把消息通道连接到应用中的同步服务，从而只要接收到消息，就调用相应的服务。同步的客户端可以直接调用服务，而异步的客户端必须通过发送消息的方式调用服务。

10.1.3 消息端点主题

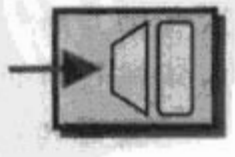
本章的另一个重要主题是，很难把事务性客户端与其他模式结合使用。事件驱动的消费者一般不能从外部正确地控制事务，消息分派器必须经过仔细的设计才能做到这一点，而从外部管理事务的竞争消费者可能会遇到严重的问题。要使用事务性客户端，最安全的方法是采用单个的轮询消费者，但是这并不是一个令人满意的解决方案。

我们还要特别提到 JMS 方式的消息驱动 bean（message-driven bean，MDB），这是一种企业 JavaBean（Enterprise JavaBeans，EJB）[EJB2.0]、[Hapner]。MDB 既是事件驱动的消费者，也是事务性客户端，它能支持 J2EE 分布式（如 XAResource）事务。MDB 还能作为竞争消费者动态入池缓存，甚至能竞争发布-订购通道。如果要在自己的应用代码中实现这种组合是很困难的，也很麻烦，不过这个功能已经作为预先建立的特性由兼容的 EJB 容器（如 BEA 的 WebLogic 和 IBM 的 WebSphere）提供。（MDB 框架是如何实现的？基本说来，容器实现了消息分派器，这个分派器有大小动态可变的缓冲池，其中放置着可重用的执行者，每个执行者基于自己的会话和事务来消费消息。）

最后，要记住，一个消息端点能很好地结合本章介绍的多个不同模式。一组竞争消费者可以实现为轮询消费者，同时也是选择性消费者，并且可以作为应用中一个服务的服务激励器。消息分派器可以是事件驱动的消费者，而且是使用消息传递映射器的持久订购者。不论端点实现了哪种模式，它同时还是一个消息传递网关。因此，不要只考虑使用一种模式，而是要考虑各种模式的组合。这也是采用模式解决问题的妙处所在。

把应用设计成消息端点有许多可选的方法。本章将逐一介绍这些方法，并介绍使用这些方法的最佳途径。

10.2 消息传递网关



应用通过消息传递访问其他系统。

如何把对消息传递系统的访问封装起来，使之独立于应用的其他部分？

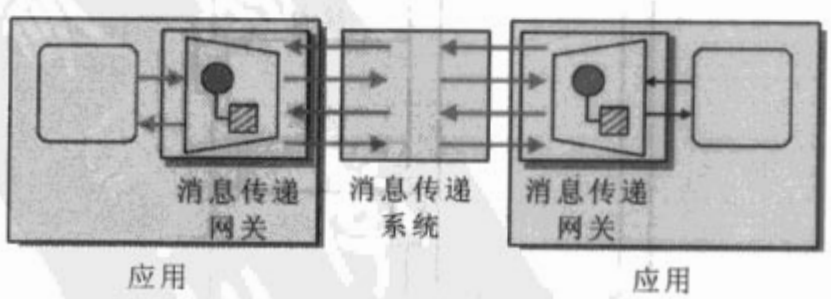
大多数定制应用都通过开发商提供的 API 来访问消息传递基础设施。尽管不同开发商提供的 API 风格不同，但是这些库有类似的功能，如“打开通道”、“创建消息”和“发送消息”。尽管这种类型的 API 允许应用在各种类型的通道上发送各种类型的消息数据，但是有时很难说清发送这些消息数据的目的是什么。

消息传递解决方案本质上是异步的，因此增加了通过消息传递访问外部功能的复杂性。应用不再是简单地调用 GetCreditScore 方法以返回信用度的数字值，而是要发送请求消息，并等待应答消息在过后的某个时刻返回（见“请求/应答”）。应用开发人员更愿意使用简单的同步函数语义处理到来的消息事件。

应用之间的松耦合在体系结构方面很有好处，例如可以接受对消息格式的小幅修改（即，增加字段）。松耦合一般使用 XML 文档或其他非强类型的数据结构实现，如 Java 或 C#类。采用这些数据结构编写代码不但繁琐，还容易出错，因为系统没有提供编译类型支持，因此不会检查字段名的拼写错误或数据类型的匹配错误等。所以，数据格式方面的灵活性是以增加应用开发难度为代价的。

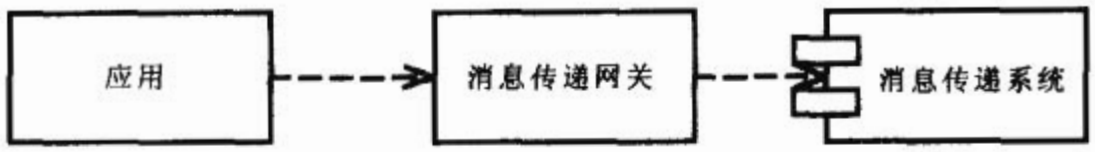
有时，通过消息传递执行一个简单的逻辑功能需要发送多个消息。例如，获得客户信息的功能实际需要多个消息，一个消息用于获得地址，一个消息用于获得订单的历史信息，另一个消息用于获得个人的信息。其中各个消息都要由不同的系统处理。我们不想在应用的代码中混合用于发送和接收三种不同消息的逻辑，而是想使用分散收集器接收一条请求消息，向三个不同的系统发送各自不同的消息，然后把返回的应答消息重新聚合成一条应答消息，以减轻应用的负担。但是，往往无法将这种功能添加到消息传递中间件中。

使用消息传递网关类把特定于消息传递的方法调用包装起来，向应用提供领域相关的方法。



消息传递网关封装了特定于消息传递的代码（如用于发送或接收消息的代码），使之与其他应用代码分离。采用这种方法，只有消息传递网关了解消息传递系统，应用的其余代

码都与消息传递系统无关。消息传递网关为应用提供业务功能接口，使应用不必设置像 Message.MessageReadPropertyFilter.AppSpecific 这样的属性，消息传递网关提供了诸如 GetCreditScore 的含义明确的方法，它们能像其他方法一样接受强类型的参数。消息传递网关是特定消息传递版本的网关模式[EAA]，网关模式是更一般的形式。



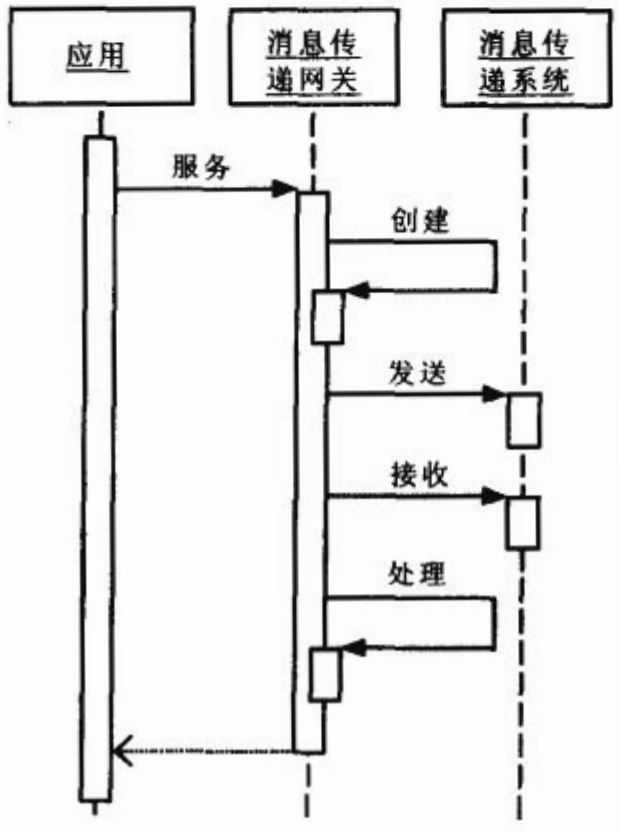
网关消除了应用与消息传递系统之间的直接依赖

消息传递网关位于应用和消息传递系统之间，向应用提供了业务领域相关的 API（如上图所示）。由于应用甚至不知道它是否使用了消息传递系统，所以可以将网关替换为使用其他集成技术的不同实现，如远程过程调用或 Web 服务。

许多消息传递网关把消息发送给其他组件，并等待应答消息（见“请求/应答”部分）。这样的消息传递网关可以采用两种不同的方法实现：

1. 阻塞（同步）方式的消息传递网关。
2. 事件驱动（异步）的消息传递网关。

阻塞方式的消息传递网关发送一个消息后，要等待应答消息返回，才能把控制交还给应用。当网关接收到应答后，它处理消息并将结果返回给应用（如下面的顺序图所示）。



阻塞（同步）方式的消息传递网关

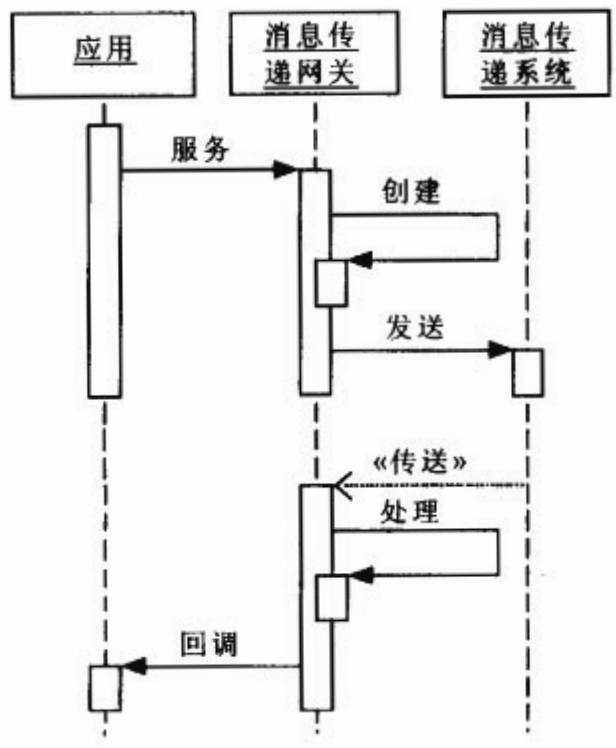
阻塞方式的消息传递网关把消息传递交互的异步特性封装起来，为应用逻辑提供了一

个常规的同步方法。因此，应用不需要了解通信的异步特性。例如，阻塞方式的网关可以提供以下方法：

```
int GetCreditScore(string SSN);
```

基于这种方法，利用消息传递网关编写应用代码变得非常简单，但它也会导致性能低下，因为应用要花费大量的时间等待应答消息，在这期间无法完成其他任务。

事件驱动的消息传递网关则把消息传递层的异步特性暴露给应用。当应用向消息传递网关发送与业务领域相关的请求时，会为应答提供一个业务领域相关的回调函数。控制立即返回给应用。当应答消息到达时，消息传递网关处理消息，然后调用回调函数（如下面的顺序图所示）。



事件驱动（异步）的消息传递网关

例如，在C#中，消息传递网关可以使用委托提供以下公共接口：

```
delegate void OnCreditReplyEvent(int CreditScore);
void RequestCreditScore(string SSN, OnCreditReplyEvent OnCreditResponse);
```

RequestCreditScore方法接受一个额外的参数，该参数指定一个回调方法，应答消息返回时就要调用这个回调方法。回调方法有一个CreditScore参数，以便消息传递网关把结果返回给应用。根据编程语言或平台的不同，回调可以通过函数指针、对象引用或委托实现（这里就采用了委托来实现）。请注意，尽管这个接口具有事件驱动特性，但它不依赖于任何特定的消息传递技术。

另外，应用也可以周期性地查询结果是否到达。采用这种方法能简化高层接口，不会引起阻塞，这实际上利用了半同步/半异步模式[POSA2]。该模式是指采用缓冲区存储到来的消息，这样应用能在方便的时候查询消息是否已经到达。

使用事件驱动的消息传递网关时，要克服的一个困难是，它要求应用维持请求方法与

回调事件之间的状态（在阻塞方式下，会由调用栈负责维护这些状态）。当消息传递网关调用回调事件而进入应用逻辑时，应用必须能把应答与它早前发出的请求关联起来，才能继续处理正确的执行线程。如果消息传递网关允许应用把任意一组数据的引用传递给请求方法，应用就能更容易地维护状态。消息传递网关再利用回调把这个数据传回到应用。采用这种方法，当调用异步回调方法时，应用就能获得所有必需的数据。这种类型的交互通常被称为 ACT（Asynchronous Completion Token，异步完成令牌）[POSA2]。

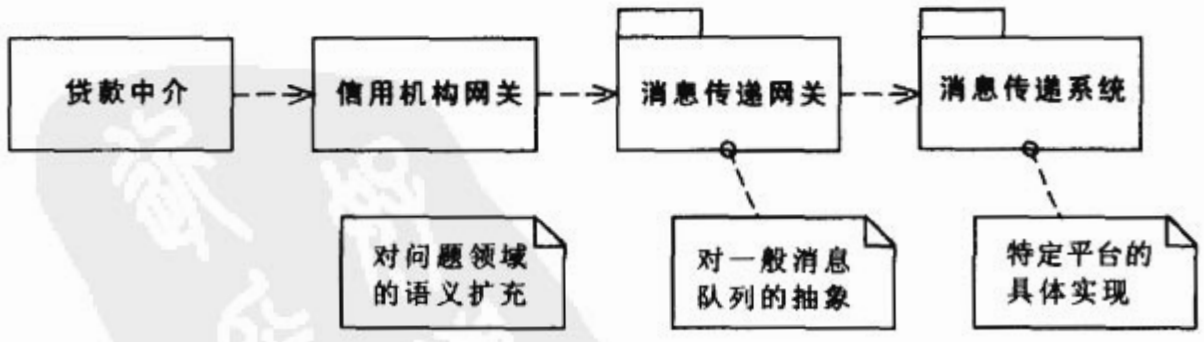
支持 ACT 的事件驱动消息传递网关提供的公共接口如下所示：

```
delegate void OnCreditReplyEvent(int CreditScore, Object ACT);
void RequestCreditScore(string SSN, OnCreditReplyEvent OnCreditResponse, Object ACT);
```

RequestCreditScore 方法需要一个额外的参数，这是一个通用对象的引用。消息传递网关存储这个引用，等待应答消息的到达。当应答消息到达时，网关调用类型为 OnCreditReplyEvent 的委托，把操作的结果以及对象引用传递给它。尽管支持 ACT 对应用来说非常方便，它也带来了内存泄漏的危险，如果消息传递网关维护了一个对象的引用，但是期望的消息始终不能到达，就会存在内存泄漏。

10.2.1 链接网关

创建多层消息传递网关很有好处。“较低层次”的消息传递网关只能抽出消息传递系统的语法，而保持一般的消息传递语义不变，如 SendMessage。当企业调整消息传递技术时，例如用 Web 服务替代原来的 MSMQ 时，采用这种消息传递网关能使应用的其余部分不受影响。我们把这个基本的消息传递网关用另外一个消息传递网关包装起来，该网关把一般的消息传递 API 转换为一个专用的面向业务领域的 API，如 GetCreditScore。我们在贷款中介示例的 MSMQ 实现中使用了这种配置（如下图所示，也可参见第 9 章“中场演练：复合消息传递”中“使用 MSMQ 的异步实现方案”部分）。



提供不同抽象层次的网关链

10.2.2 处理消息传递异常

除了能简化应用的代码编写以外，消息传递网关还能消除应用代码对特定消息传递技术的依赖。只要把与消息传递相关的所有方法调用封装在消息传递网关的接口后面，能做

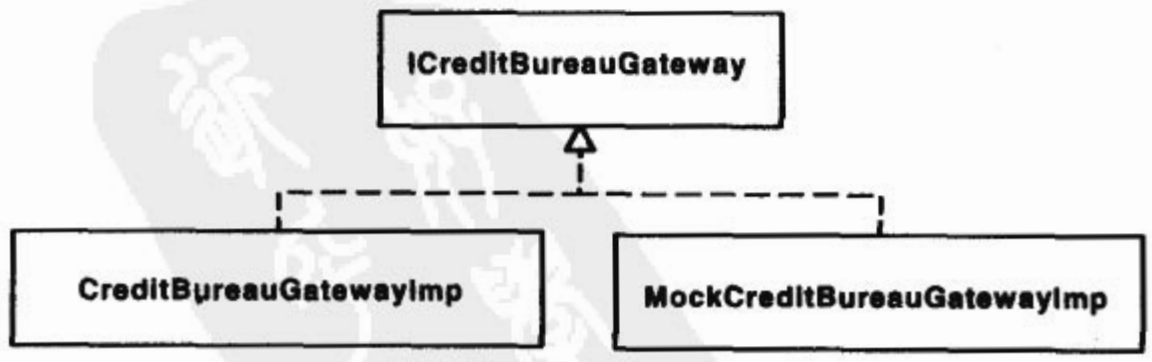
到这一点。但是，大多数消息传递层会抛出特定于消息传递的异常，如 JMS 会抛出 InvalidDestinationException 异常。如果我们想让应用的代码真正与消息传递库独立，消息传递网关就必须捕获所有与消息传递相关的异常，并抛出与应用相关的异常（或者是一般异常）。编写这样的代码会稍微麻烦一点，但是如果我们想切换底层的实现，例如用 Web 服务代替 JMS，采用这种方法就会非常方便。

10.2.3 生成网关

在很多情况下，可以根据外部资源提供的元数据生成消息传递网关代码。Web 服务就经常使用这种方法。几乎每个开发商或开放源代码的平台都提供了类似 wsdl2java 的工具，它能连接由外部 Web 服务提供的 Web 服务描述语言（Web Service Description Language，WSDL）。这个工具会生成 Java（或 C#，或你需要的某种语言）类，把所有与 SOAP 相关的处理封装起来，提供一个简单的函数调用接口。我们创建了一个类似的工具，它能从 TIBCO 知识库中读取消息模式定义，并创建实现此模式定义的 Java 类源代码。这样一来，应用开发人员就能发送类型正确的 TIBCO ActiveEnterprise 消息，而不必掌握 TIBCO API 方面的知识。

10.2.4 使用网关用于测试

消息传递网关提供了丰富的测试手段。由于我们把所有的消息传递代码包装在一个与业务领域相关的专用接口中，因此能轻松地创建一个虚拟的接口实现。我们把接口与具体的实现分离，并提供了两个实现版本：一个是“真正”能访问消息传递基础设施的实现，一个是“伪”实现以用于测试（如下图所示）。伪实现可以作为服务桩[EAA]，使用这个服务桩，不依靠消息传递就能测试应用。使用了事件驱动消息传递网关的应用也可以使用服务桩来测试。例如，一个针对事件驱动消息传递网关的简单测试桩能从请求方法中直接调用回调（或委托），在一个线程中同时执行请求和应答消息的处理。这就能极大地简化了单步调试。



网关可以作为测试工具

例：采用 MSMQ 实现的异步贷款中介网关

这个例子给出了在第 9 章“中场演练：复合消息传递”中介绍的贷款中介示例的部分代码（见“使用 MSMQ 的异步实现方案”）。

```
public delegate void OnCreditReplyEvent(CreditBureauReply creditReply, Object ACT);

internal struct CreditRequestProcess
{
    public int CorrelationID;
    public Object ACT;
    public OnCreditReplyEvent callback;
}
internal class CreditBureauGateway
{
    protected IMessageSender creditRequestQueue;
    protected IMessageReceiver creditReplyQueue;

    protected IDictionary activeProcesses = (IDictionary)(new Hashtable());

    protected Random random = new Random();

    public void Listen()
    {
        creditReplyQueue.Begin();
    }

    public void GetCreditScore(CreditBureauRequest quoteRequest,
                               OnCreditReplyEvent OnCreditResponse,
                               Object ACT)
    {
        Message requestMessage = new Message(quoteRequest);
        requestMessage.ResponseQueue = creditReplyQueue.GetQueue();
        requestMessage.AppSpecific = random.Next();

        CreditRequestProcess processInstance = new CreditRequestProcess();
        processInstance.ACT = ACT;
        processInstance.callback = OnCreditResponse;
        processInstance.CorrelationID = requestMessage.AppSpecific;

        creditRequestQueue.Send(requestMessage);

        activeProcesses.Add(processInstance.CorrelationID, processInstance);
    }
```

```
private void OnCreditResponse(Message msg)
{
    msg.Formatter =  GetFormatter();

    CreditBureauReply replyStruct;
    try
    {
        if (msg.Body is CreditBureauReply)
        {
            replyStruct = (CreditBureauReply)msg.Body;
            int CorrelationID = msg.AppSpecific;

            if (activeProcesses.Contains(CorrelationID))
            {
                CreditRequestProcess processInstance =
                    (CreditRequestProcess)(activeProcesses[CorrelationID]);
                processInstance.callback(replyStruct, processInstance.ACT);
                activeProcesses.Remove(CorrelationID);
            }
            else
            {
                Console.WriteLine("Incoming credit response does not match any request");
            }
        }
        else
        { Console.WriteLine("Illegal reply."); }
    }
    catch (Exception e)
    {
        Console.WriteLine("Exception: {0}", e.ToString());
    }
  }
}
```

你可能会注意到，公共的 GetCreditScore 方法和公共的 OnCreditReplyEvent 委托根本没有引用消息传递。这个实现允许调用方应用来把任意的对象引用作为 ACT 传递。CreditBureauGateway 把这个对象引用存储在一个字典中，并且可以通过请求消息中的关联标识符进行索引。当应答消息到达时，CreditBureauGateway 可获取与发出的请求消息相关的数据。调用方应用不必考虑消息是如何关联的。

10.3 消息传递映射器

应用使用消息传递时，消息的数据常常从应用的领域对象得来。如果我们使用一个文档消息，那么消息本身可以直接表示一个或多个领域对象。如果使用的是命令消息，那么与命令相关的部分数据字段很可能也是从领域对象中提取出来的。消息与对象之间有一些明显的区别。例如，大多数对象要通过对象引用和继承关系的方式关联。许多消息传递基础设施并不支持这些概念，因为它们必须与大量的应用通信，而其中一些应用可能根本不是面向对象的。

> 如何实现领域对象与消息传递基础设施之间的数据传递，同时保持它们之间的相互独立？

为什么不能把消息设计得像领域对象一样，从而解决这个问题？在许多情况下，我们是无法控制消息格式的，因为消息格式可能由规范数据模型或公共的消息传递标准（如ebXML）定义。我们只能采用与领域对象相对应的格式发布消息，并使用消息传递层中的消息转换器把消息转换为公共的消息格式。如果第三方系统不允许在应用中实现转换，相应的适配器（如数据库适配器）经常会采用这种方法。

另外，也可以在领域层创建和发布所需格式的消息，不必使用专门的消息转换器。这种方法很可能有更好的性能，因为不用再发布中间消息。而且，如果领域模型中包含许多小对象，先把这些小对象组合成一个消息就能简化路由，并提高消息传递层内部的处理效率。即使我们能提供额外的转换步骤，如果想创建类似于业务领域对象的消息，也会受到很多限制。采用这种方法的缺点是领域将依赖于消息的格式，如果消息的格式发生改变，就会增加领域维护的难度。

大多数消息传递基础设施在提供的API中都支持“消息”对象的概念。消息对象封装了要通过通道发送的数据。在大多数情况下，这个消息对象只能包含标量（简单）数据类型，如字符串、数字和日期，不能支持继承和对象引用。这也是远程过程调用方式的通信（如RMI）与异步消息传递系统之间的一个主要区别。假设我们要发送一条包含了某组件对象引用的异步消息。为了处理这个消息，组件必须解析对象的引用。为此它要向消息源请求该对象。但是，请求/应答方式的交互会违反使用异步消息传递的部分初衷（例如组件之间的松耦合），更糟糕的是，当订购者接收到异步消息时，所引用的对象可能已经不在源系统中了。

要解决对象引用问题，一种可能的方法是遍历对象的依赖树，并把所有依赖对象都包含在消息中。例如，如果Order对象引用了五个OrderItem对象，可以把这五个对象包含在消息中。这就确保了接收者能通过“根”对象访问到所有的数据引用。但是，如果我们使用的是一个细粒度的领域对象模型，存在许多相互关联的对象，那么消息很快就会变得

很大。因此，必须更好地控制哪些内容要放到消息中，而哪些内容不能放入消息。

现在先假设我们的领域对象是自包含的，且其中没有其他任何对象的引用。即使这样也不能简单地把整个领域对象都封装到消息中，因为大多数消息传递基础设施并不支持对象，它们必须独立于语言（JMS 中的 ObjectMessage 接口和.NET System.Messaging 名字空间中的 Message 类有所例外，因为这些消息传递系统要么是语言相关的[Java]，要么是平台相关的[.NET CLR]）。我们可以考虑把对象串行化为字符串，并把该字符串存储到一个名为“data”的字符串字段中（即数据字段），这样所有消息传递系统都能支持。但是，这种方法也有缺点。首先，消息路由器不能再使用对象的属性进行路由，因为该字符串字段对消息传递层不再“可见”。其次，它还增加了测试和调试的难度，因为我们还要解析数据字段的内容。另外，如果把所有消息都构造为只包含一个字符串字段，就无法实现基于消息类型的消息路由，因为对基础设施来说，所有的消息看起来都是一样的。同时，它也不利于检查消息格式的正确性，因为消息传递基础设施无法检查数据字段中的任何内容。最后一点，我们可能无法使用语言运行时库提供的串行化工具，因为这些消息的表示对不同语言来说并不兼容。因此，我们必须编写自己的串行化代码。

有些消息传递基础设施已经能支持在消息中使用 XML 字段，这样就能把对象串行化为 XML，从而减少不利因素，因为这种消息很容易解析，并且某些消息传递层可以直接访问 XML 字符串中的元素。但是，为了支持 XML，我们必须处理更冗长的消息，同时只能实现有限的数据类型检验。另外，还要创建代码把对象转换为 XML，以及从 XML 转换为对象。取决于我们所使用的编程语言，实现这些操作可能会非常复杂，特别是使用那些不支持反射（reflection）的较早语言。

之所以要把这种映射代码从领域对象中分离出来，有很多原因。首先，不要把有低级语言特性的代码与处理应用逻辑的代码混起来。在很多情况下，可以让一组程序员负责消息传递层代码的编写，让另一组程序员把精力集中在业务逻辑代码的编写方面。如果把这两种代码编写到同一个对象中，则很难让两个小组同时开展工作。

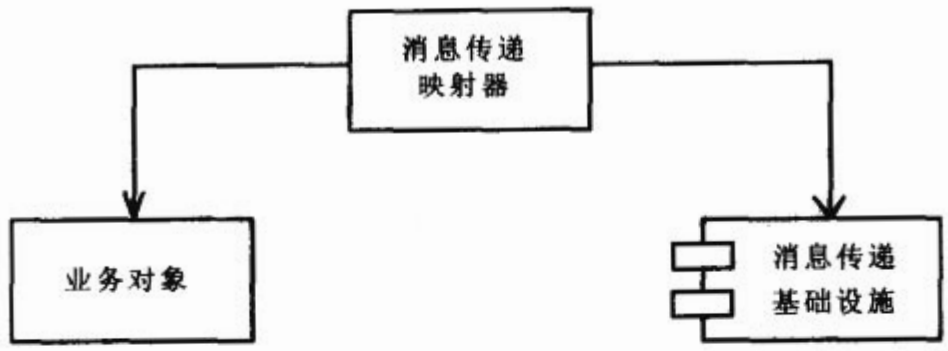
其次，把映射代码组合到领域对象中，就会使领域对象依赖于消息传递基础设施，因为映射代码要调用消息传递 API（如实例化 Message 对象）。在大多数情况下，这种依赖性会阻碍领域对象的重用，如在不使用消息传递的其他上下文中，或者在使用其他开发商消息传递基础设施的上下文中，就很难重用，从而严重影响了领域对象的可重用性。

我们经常看到人们编写“抽象层”，把消息传递基础设施提供的 API 封装起来，使负责消息传递的代码与消息传递 API 独立。这种抽象层使消息传递的接口与消息传递的具体实现相分离，提供了一个间接层。因此，即使切换到其他开发商的消息传递层，也可以重用与消息传递相关的代码。所要做的只是实现一个新的抽象层，把消息传递接口转换为新的 API。但是，这种方法并不能克服领域对象对消息传递层的依赖。领域对象包含的是抽象消息传递接口的引用，而不是开发商专用消息传递 API 的引用，但是在不使用消息传递的上下文中仍然不能重用领域对象。

许多消息是由多个领域对象组成的。由于我们无法通过消息传递基础设施传送对象的引用，因此可能要在消息中包含来自其他对象的字段。在有些情况下，还要把所有依赖对

象组成的整个“依赖树”包含在一个消息中。应该由哪个类实现映射代码？要知道，在由不同对象组合成的多种类型的消息中，可能都包含同样的某个对象，因此要解决这个问题并不容易。

创建一个单独的消息传递映射器，其中包含消息传递基础设施与领域对象之间的映射逻辑。对象和基础设施都不需要知道消息传递映射器的存在。



消息传递映射器能访问一个或多个领域对象，并把它们转换为消息通道所要求的消息。它还实现了相反的功能，能根据到来的消息创建或更新领域对象。由于消息传递映射器实现为一个单独的类，这个类同时引用了领域对象和消息传递层，因此领域对象与消息传递层都不了解对方，甚至不知道消息传递映射器的存在。

消息传递映射器是映射器模式[EAA] 的一种特殊形式。它与数据映射器（Data Mapper）[EAA]有一定的相似性。只要是设计过 O-R（Object-Relational，对象关系）映射策略的人，都能理解在使用不同范式的层之间映射数据是何等复杂。消息传递映射器也有类似的复杂性，我们不会详细讨论与映射相关的所有内容，这超出了本书的范围。如果读者对创建消息传递映射器层感兴趣，可以阅读[EAA]中介绍的许多数据源建构模式。

我们经常会在消息传递 API 之外包装一个抽象层，但这个概念与消息传递映射器有所不同。对于抽象层，领域对象不必了解消息传递 API，但是它们要知道抽象层（从本质上讲，抽象层完成了消息传递网关的功能）。对于消息传递映射器，对象根本不知道在采用消息传递。

使用消息传递映射器的目的与使用仲裁者[GoF]相似，也是用来分离组件。但是在仲裁者模式中，组件清楚仲裁者的存在，而消息传递映射器模式中，所有的组件都不知道消息传递映射器的存在。

如果领域对象和消息传递基础设施都不知道消息传递映射器的存在，应该如何调用消息传递映射器呢？在大多数情况下，消息传递映射器是通过事件调用的，这个事件由消息传递基础设施或应用触发。由于它们都不依赖于消息传递映射器，事件通知要么通过单独编写的代码完成，要么把消息传递映射器实现为观察者模式[GoF]来完成。例如，如果我们使用 JMS API 与消息传递基础设施交互，就可以实现 MessageListener 接口，从而得到输入消息的通知。同样地，我们也可以使用观察者（Observer）得到领域对象中相关事件的通

知，并调用消息传递映射器。如果要直接从应用调用消息传递映射器，应该定义一个消息传递映射器接口，使应用至少不依赖于消息传递映射器的具体实现。

10.3.1 减少编码的工作量

某些消息传递映射器的具体实现可能包含了大量重复的代码：从领域对象中获得字段，并把该字段存储到一个消息对象中，然后继续处理下一个字段，直到所有的字段都处理完为止。实现这样的代码相当麻烦，而且这些代码看起来非常相似。有很多工具可以帮助我们处理这些繁琐的工作。首先，我们可以编写一个通用消息传递映射器，它使用反射来以一般方法从领域对象中取出字段。例如，它可以遍历领域对象中所有字段的列表，并将其保存到消息对象中的同名字段中。显然，只有字段名匹配时才能这样做。根据前面的讨论，我们必须采取某种措施解决对象引用问题，因为对象引用无法保存到消息对象中。另一种方法是，使用可配置的代码生成器来生成消息传递映射器的代码。这样在为字段命名时就有更大的灵活性（消息对象的字段名不一定要与领域对象的字段名匹配），并且可以采用更聪明的方法处理对象的引用。代码生成器的缺点是很难测试和调试，但是如果它足够通用，我们就只需编写一次代码生成器。

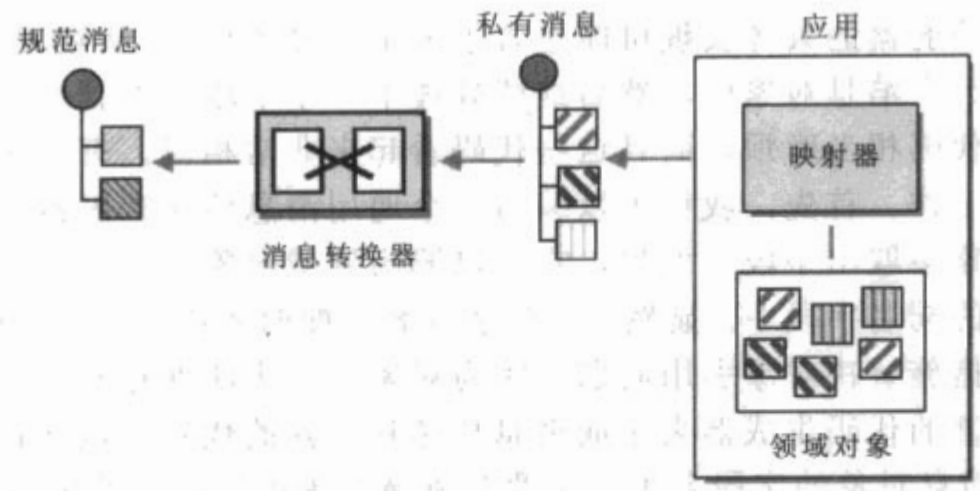
有些框架（如 Microsoft .NET）可以提供内置的对象串行化功能，能将对象串行化为 XML，还可以把 XML 逆串行化为对象，从而避免了大量繁琐的对象串行化代码。即使框架可以完成把对象转换为消息的部分工作，这种转换也只限于语法级。可以尝试让框架完成所有的转换操作，但是它只能创建与领域对象一一对应的消息。前面我们已经介绍过，由于消息与领域对象的限制条件和设计规则完全不同，采用这种做法并不理想。可以定义一组与所需消息结构对应的“接口对象”，然后让框架完成消息与这些对象之间的转换。消息传递映射器层负责管理真正的领域对象与接口对象之间的转换。这些接口对象与数据传输对象[EAA]的功能有些相似，不过它们的目的有所不同。

10.3.2 映射器与转换器

即使我们使用了消息传递映射器，仍然可以使用消息转换器把由消息传递映射器生成的消息转换为与规范数据类型兼容的消息。这就为我们提供了一个间接层。可以使用消息传递映射器来解决诸如对象引用和数据类型转换方面的问题，而把结构方面的映射留给消息传递层中的消息转换器处理。为了得到这种额外的解耦合，代价是必须创建额外的组件，而且还会造成一定的性能开销。另外，如果要实现复杂的转换，有时利用应用的编程语言实现要比使用集成开发商提供的拖放“傻瓜件（doodleware）”更容易。

如果同时使用消息传递映射器和消息转换器，就能在规范数据格式和领域对象之间获得额外的一个间接层。据称，计算机科学是一种只要再增加一个间接层就能解决所有问题的学科，这种说法在这里也适用吗？基于额外的间接性，我们能修改消息传递层中的规范模型，而不必修改应用的代码。这样还能简化应用中的映射逻辑，把繁琐的字段映射和数据类型修改（如把数字类型的 ZIP_Code 字段转换为字母数字类型的 Postal_Code 字段）交给消息传递层的映射工具来完成，这些工具最适合处理这种工作。消息传递映射器则主要

处理对象引用的解析，并消除不必要的领域对象信息。引入额外的间接层有一个明显的缺点，如果领域对象有修改，则需要对消息传递映射器和消息转换器都做修改。如果我们负责生成消息传递映射器的代码，那么这个问题基本上是可以克服的。

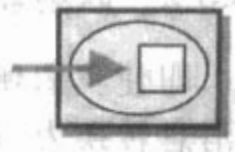


组合映射器与消息转换器

例：用 JMS 实现的消息传递映射器

在 JMS 实现的聚合器示例中，AuctionAggregate 类扮演了消息传递映射器的角色，它实现了 JMS 消息传递系统与 Bid 类之间的映射。addMessage 方法和 getResultMessage 方法实现了 JMS 消息与 Bid 对象之间的转换。消息传递系统和 Bid 类都不清楚这个交互的任何细节。

10.4 事务性客户端



在必要的情况下，消息传递系统在内部使用了事务性操作。外部客户端最好能对影响其行为的事务的范围有所控制。

客户端应该如何控制它与消息传递的事务？

消息传递系统必须在内部使用事务。一个消息通道可能有多个发送者和多个接收者，因此消息传递系统必须协调消息，确保发送者不会相互覆盖消息，多个点对点通道的接收者不会接收到相同的消息，多个发布-订购通道的接收者各自只能接收到每个消息的一个副

本，如此等等。为了完成这些管理，消息传递系统在内部使用了事务，以确保消息能（或不能）增加到通道中，以及确保消息能（或不能）从通道中读取。消息传递系统还必须使用事务（最好是两段式分布式事务）来复制消息，即把消息从发送者的计算机复制到接收者的计算机，从而在任何给定时刻，消息“实际上”只能在其中一台计算机上。

发送和接收消息的消息端点是事务性的，尽管它们本身可能没有意识到这一点。把消息添加到通道中的发送方法要在一个事务中完成，从而使该消息能与其他消息隔离，如同时添加到通道中的其他消息或者从通道中删除的消息。同样地，接收方法也使用了事务，以防止其他点对点接收者获得相同的消息，甚至能确保发布-订购通道的接收者不会多次读到同样的消息。

事务往往描述为 ACID：原子的（Atomic）、一致的（Consistent）、隔离的（Isolated）和持久的（Durable）。只有可靠消息传递（Guaranteed Messaging）的事务才是持久的，根据定义，消息本身就是原子的。不过所有的消息传递事务都必须具有一致性和隔离性。不能说通道中的一个消息“有点算”消息，它要么是消息，要么不是。另外，应用发送和接收消息的操作必须与其他线程隔离，而且应用可以通过相同的通道发送和接收消息。

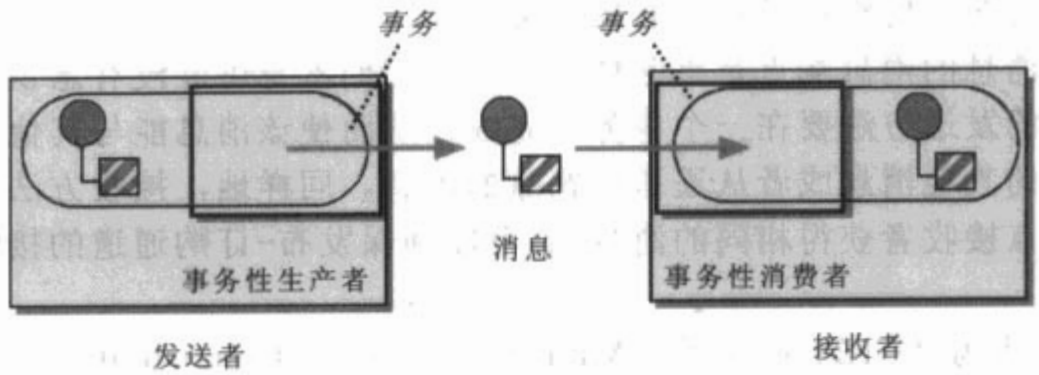
如果客户只想发送和接收一个消息，那么消息传递系统的内部事务就已经足够了。但是，应用可能需要一个范围更广的事务来协调多个消息，或者与其他资源一起协作完成消息传递。常用的情况包括：

- **发送/接收消息对**——接收一个消息并发送另一个消息，如请求/应答，或者实现像消息路由器或消息转换器之类的消息过滤器。
- **消息组**——发送或接收一组相关的消息，如消息序列。
- **消息/数据库协调**——把发送或接收消息的操作与更新数据库相结合，如采用通道适配器。例如，当应用接收和处理一个订购产品的消息时，应用还需要更新产品的库存数据库。同样地，文档消息的发送者可能要删除一个持久的文档，但是这个操作只能在文档发送成功后执行；在真正消费消息之前，接收者可能要把文档持久保存起来。
- **消息/工作流协调**——使用一对请求/应答消息完成一个工作项（work item），使用事务来确保只有在发送了请求时才能获得工作项，只有接收到应答时才能完成或中止工作项。

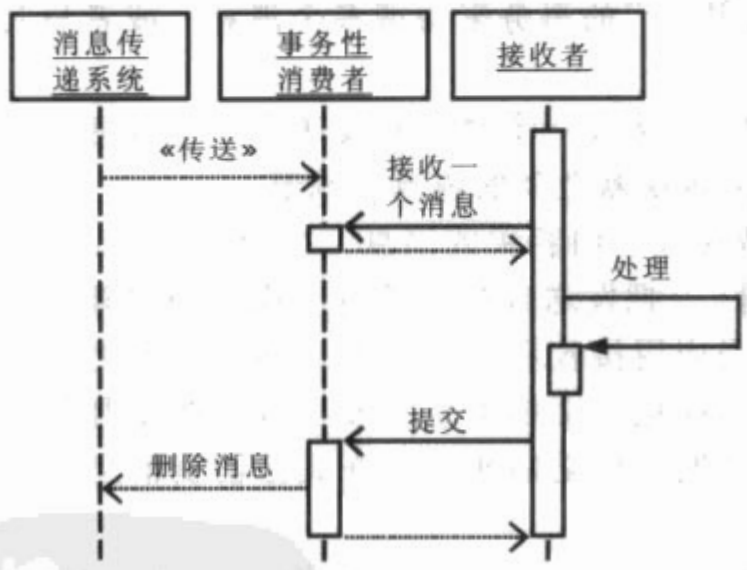
诸如这样的应用环境需要更大的原子事务，它所涉及的不再是一个消息，除了消息传递系统以外，甚至还可能包含其他事务性存储系统。如果只有部分工作完成（如接收到消息），但其他的部分没有完成（如更新数据库或发送另一个消息），此时要求所有部分都回滚，就好像什么都没有发生过，应用可以再重新尝试，这就需要用到事务。

要想让一个应用与其他消息或资源来协作处理一个消息，只是采用消息传递系统的内部事务模型是不够的。应用要能从外部控制消息传递系统的事务，并能与消息传递系统或其他系统中的其他事务结合。

使用事务性客户端——使客户端与消息传递系统之间建立事务性会话，从而使客户端能指定事务边界。



发送者和接收者都能是事务性的。对发送者来说，只有在发送者提交事务后消息才会真正增加到通道中。对接收者来说，只有在接收者提交事务后，才能真正从通道中删除消息。使用显式事务的发送者可以与使用隐式事务的接收者合作，反之亦然。一个通道可以同时有隐式和显式的事务性发送者，也可以同时有隐式和显式的事务性接收者。



事务性接收者顺序图

采用事务性接收者，应用能接收到消息，而不必真正把消息从队列中删除。如果应用在这个时刻崩溃，当它恢复运行时，消息仍然在队列中，并不会丢失。接收到消息后，应用就能加以处理。一旦应用完成消息的处理，并确信它将消费这个消息，应用才提交事务，如果提交成功，消息才真正从通道中删除。如果应用在这个时刻崩溃，当它恢复运行时，消息已经不在通道中，因此应用确实已经完成了这个消息的处理。

从外部控制消息传递系统的事务，这对应用协调多个任务有什么帮助？对前面所述的情况，应用要做的工作如下：

发送/接收消息对

1. **要做什么**：开始一个事务，接收和处理第一个消息，创建和发送第二个消息，然后

提交事务（这些操作通常实现为请求/应答、消息路由器和消息转换器操作的一部分）。

2. **做到什么**：保证只有在第二个消息成功添加到它的通道中后，才能把第一个消息从它所在的通道中删除。

3. **事务类型**：如果两个消息通过同一个消息传递系统中的通道发送，那么涉及这两个通道的事务就是一个简单事务。但是，如果这两个通道由不同的消息传递系统管理，如利用了一个消息传递桥，那么该事务将是一个分布式事务，它同时要协调两个消息传递系统。

4. **警告**：一个事务只能用于请求的接收者发送应答，请求的发送者不能在一个事务中发送请求并等待相应的应答。如果尝试这么做，请求将永远无法发送，因为发送事务不会提交，因而应答也永远不可能接收到。

消息组

1. **要做什么**：开始一个事务，发送或接收组中所有的消息（如一个消息序列），然后提交事务。

2. **做到什么**：当发送消息时，只有所有的消息都成功发送后，组中的消息才会添加到通道中。当接收消息时，只有所有的消息都接收到后，才能从通道中删除这些消息。

3. **事务类型**：由于所有消息都是通过一个通道发送或接收的，因此该通道将由一个消息传递系统管理，这种事务属于简单事务。另外，许多消息传递系统的具体实现中，在一个事务中发送一组消息，可以确保这些消息能在通道的另一端按发送的顺序接收。

消息/数据库协调

1. **要做什么**：开始一个事务，接收一个消息，更新数据库，然后提交事务。或者是更新数据库，发送一个消息，把更新情况报告给其他组件，然后提交事务（这种操作一般是由通道适配器实现）。

2. **做到什么**：除非数据库得到更新，否则消息不会被删除（或者如果消息无法发送，数据库就不会更新）。

3. **事务类型**：由于消息传递系统和数据库都有自己的事务管理器，因此把它们协调起来的事务是一个分布式事务。

消息/工作流协调

1. **要做什么**：使用一对请求/应答消息执行一个工作项。开始一个事务，获得工作项，发送请求消息，然后提交事务。或者是开始另一个事务，接收应答消息，完成或放弃工作项，然后提交事务。

2. **做到什么**：除非请求消息确实发送，否则不提交工作项。除非工作项得到更新，否则不会删除应答消息。

3. **事务类型**：由于消息传递系统和工作流引擎都有自己的事务管理器，因此把它们协调起来的事务是一个分布式事务。

这样一来，应用可以确保不会丢失接收到的消息，或忘记发送本该发送的消息。如果在这个过程中出现什么错误，应用可以回滚事务并重试。

使用事件驱动消费者的事务性客户端可能并不像期望的那样工作。在把消息传递给应用之前，为了接收消息，消费者通常必须提交事务。如果应用在检查后发现它不想消费这

个消息，或者应用遇到一个错误而想回滚消费动作，由于它无法访问事务，因此无法达到目的。所以，不管客户端是否是事务性，事件驱动的消费者的工作并没有什么变化。

消息传递系统能够参与到分布式事务中，尽管它的某些实现可能不支持分布式事务。在JMS中，提供者可以作为一个XA资源，并参与到Java Transaction API[JTA]事务中。这种行为是由javax.transaction.xa包，以及javax.jms包中的XA类，尤其是javax.jms.XASession类定义的。JMS规范建议JMS客户端不要直接处理分布式事务，应用应该使用J2EE应用服务器提供的分布式事务支持。MSMQ也能参与到XA事务中，其行为是由.NET的MessageQueue.Transactional属性和MessageQueueTransaction类提供的。

前面已经介绍过，事务性客户端可以作为其他模式的组成部分，如请求/应答、管道和过滤器中的消息过滤器、消息序列和通道适配器等模式。另外，事件消息（Event Message）的接收者希望在完成对事件的处理后，才从通道中彻底删除对应的消息。但是，事务性客户端无法与事件驱动的消费者或消息分派器很好地配合，这会给竞争的消费者带来麻烦，不过对于单个轮询消费者却能很好地协作。

例：JMS事务会话

在JMS中，客户端在创建会话时把自己设置为事务性的[JMS1.1]、[Hapner]。

```
Connection connection = //获得连接
Session session =
    connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

该会话是事务性的，因为第一个createSession参数被设置为true。

当客户端使用事务性会话时，它必须明确提交发送和接收操作，才能实现真正的发送和接收。

```
Queue queue = //获得队列
MessageConsumer consumer = session.createConsumer(queue);
Message message = consumer.receive();
```

现在，从这个消费者的事务视图看，消息已经被消费了。但是对于其他的消费者，如果有自己的事务视图，这个消息仍然是可消费的。

```
session.commit();
```

现在，假设所提交的消息不会抛出任何异常，这个消费者的事务视图将成为消息系统的事务视图，这样消息系统会认为消息已被消费。

例：.NET事务性队列

在.NET中，默认的队列不是事务性的，因此，为了使用事务性客户端，当创建队列时，必须把它设置为事务性：

```
MessageQueue.Create("MyQueue", true);
```

队列一旦成为事务性的，客户端的每个队列操作（发送或接收）既可以是事务性的，也可以是非事务的。一个事务性Receive操作如下所示：

```
MessageQueue queue = new MessageQueue("MyQueue");
MessageQueueTransaction transaction =
    new MessageQueueTransaction();
transaction.Begin();
Message message = queue.Receive(transaction);
transaction.Commit();
```

尽管客户端已经接收到消息，消息传递系统还是要等到客户端成功提交事务后才从队列中删除消息[SysMsg]。

例：使用 MSMQ 实现事务性过滤器

下面的例子改进了管道和过滤器中介绍的基本过滤器组件，从而使用事务。这个例子实现了发送/接收消息对的情况，接收和发送消息在同一个事务中完成。为了把过滤器实现为事务性过滤器，只需增加几行代码。我们要使用类型为 MessageQueueTransaction 的变量管理事务，在消费输入的 Message 前打开一个事务，并且在发布输出消息后提交事务。如果出现任何异常，要中止该事务，这会回滚所有的消息消费操作和发布操作，并把输入的消息退还给队列，以便让其他的队列消费者消费。

```
public class TransactionalFilter
{
    protected MessageQueue inputQueue;
    protected MessageQueue outputQueue;

    protected Thread receiveThread;
    protected bool stopFlag = false;

    public TransactionalFilter (MessageQueue inputQueue,MessageQueue outputQueue)
    {
        this.inputQueue = inputQueue;
        this.inputQueue.Formatter = new System.Messaging.XmlMessageFormatter
                                    (new String[] {"System.String,mscorlib"});
        this.outputQueue = outputQueue;
    }

    public void Process()
    {
        ThreadStart receiveDelegate = new ThreadStart(this.ReceiveMessages);
        receiveThread = new Thread(receiveDelegate);
        receiveThread.Start();
    }

    private void ReceiveMessages()
```

```
{
    MessageQueueTransaction myTransaction = new MessageQueueTransaction();

    while (!stopFlag)
    {
        try
        {
            myTransaction.Begin();
            Message inputMessage = inputQueue.Receive(myTransaction);
            Message outputMessage = ProcessMessage(inputMessage);
            outputQueue.Send(outputMessage, myTransaction);
            myTransaction.Commit();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message + " - Transaction aborted ");
            myTransaction.Abort();
        }
    }
}
```

```
    protected virtual Message ProcessMessage(Message m)
    {
        Console.WriteLine("Received Message: " + m.Body);
        return m;
    }
}
```

怎样才能验证事务性客户端实现了所需的操作？我们可以创建 TransactionalFilter 基类的子类，并把它取名为 RandomlyFailingFilter。对于每个要消费的消息，该过滤器产生一个 0～10 之间的随机数。如果随机数小于 3，它将故意抛出一个异常（如果仅用于举例，使用 ArgumentNullException 异常已经足够了）。如果我们基于“管道和过滤器”一节介绍的基本非事务过滤器之上实现这个过滤器，大概会丢失三分之一的消息。

```
public class RandomlyFailingFilter : TransactionalFilter
{
    Random rand = new Random();

    public RandomlyFailingFilter(MessageQueue inputQueue, MessageQueue outputQueue)
      : base (inputQueue, outputQueue) { }
```

```
    protected override Message ProcessMessage(Message m)
    {
        string text = (string)m.Body;
        Console.WriteLine("Received Message: " + text);

        if (rand.Next(10) < 3)
        {
            Console.WriteLine("EXCEPTION");
            throw (new ArgumentNullException());
        }
        if (text == "end")
            stopFlag = true;
        return(m);
    }
}
```

为了确保这个事务性过滤器不会丢失任何消息，我们特意建立了一个简单的测试程序，它把一个消息序列发布到输入队列中，并验证能否从输出队列中按正确的顺序接收到所有的消息。特别要记住的是，只有运行一个事务性过滤器实例，才能保证输出消息的顺序。如果同时运行多个过滤器，消息就会坏序，即顺序不正确（见“重排器”）。

```
public void RunTests()
{
    MessageQueueTransaction myTransaction = new MessageQueueTransaction();

    for (int i=0; i < messages.Length; i++)
    {
        myTransaction.Begin();
        inQueue.Send(messages[i], myTransaction);
        myTransaction.Commit();
    }

    for (int i=0; i < messages.Length; i++)

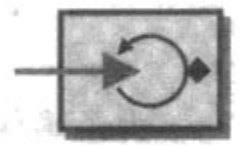
    {
        myTransaction.Begin();
        Message message = outQueue.Receive(new TimeSpan(0,0,3), myTransaction);
        myTransaction.Commit();

        String text = (String)message.Body;
```

```
        Console.Write(text);
        if (text == messages[i])
            Console.WriteLine(" OK");
        else
            Console.WriteLine(" ERROR");
    }

    Console.WriteLine("Hit enter to exit");
    Console.ReadLine();
}
```

10.5 轮询消费者



应用需要消费消息，但是它想控制何时消费各个消息。

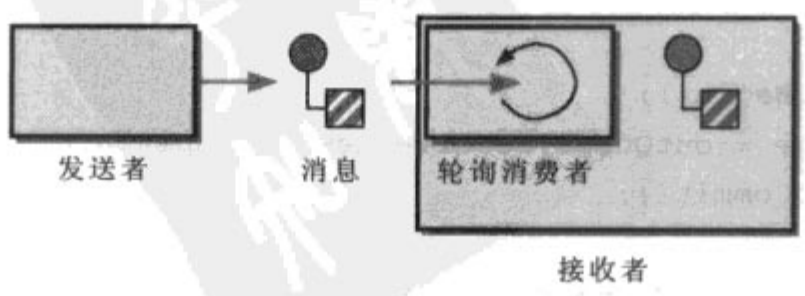
应用准备好时，如何消费一个消息？

消息消费者存在的理由只有一个——消费消息。消息代表了需要被完成的工作，因此消费者必须消费这些消息，并完成相应的工作。

但是，消费者如何才能知道有新的消息可以消费？最早的办法是让消费者反复检查通道，查看是否有消息可用。当有消息到达时，它将消费这个消息，然后继续检查下一个消息。这个过程被称为轮询（polling）。

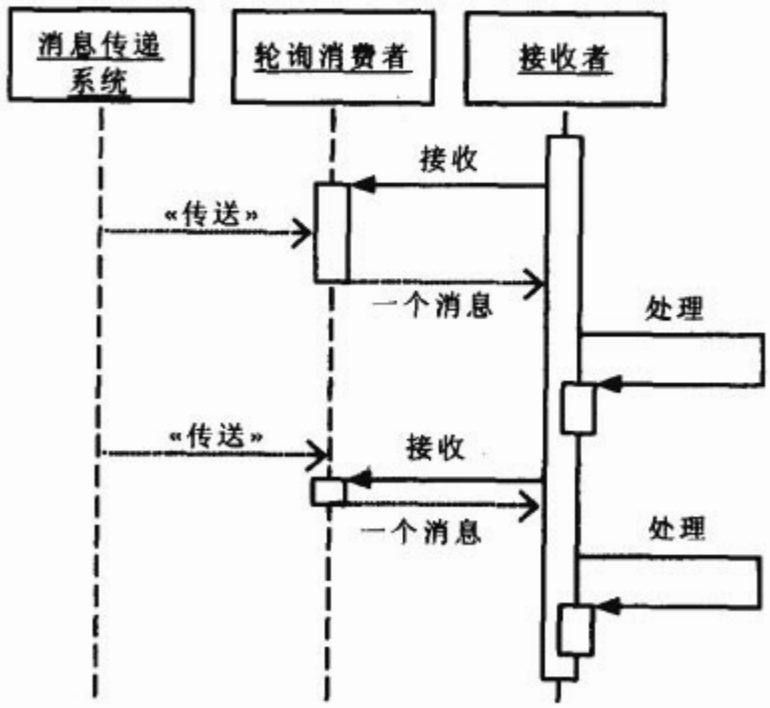
轮询的优点是，当消费者准备好处理下一个消息时，它可以请求下一个消息。因此，它能按照自己的速度处理消息，而不是按照消息到达通道的速度。

应用使用轮询消费者，它只是在需要接收消息时才明确地做出调用。



这也称为同步的接收者，因为接收方的线程在消息接收前一直阻塞。之所以把它称为轮询消费者，是因为接收者要轮询消息，处理消息，然后再轮询下一个消息。为了方便起见，除了诸如 receiveNoWait 和 Receive(0)之类的方法（如果没有可用的消息，这些方法就会立即返回），消息传递 API 一般还提供了另一个接收方法，这个接收方法在消息传送前会一直阻塞。当接收者轮询的速度比消息到达的速度快时，这两种方法才会有明显的区别。

轮询消费者是一个对象，通过明确地请求消息，应用使用该对象接收消息。当应用准备好接收下一个消息时，会轮询消费者，消费者接着从消息传递系统中获得一个消息，并将这个消息返回。（消费者如何从消息传递系统中获得消息与具体的实现相关，它既可以采用轮询的方式，也可以采用非轮询的方式。所有的应用都知道，除非它明确地请求消息，否则不会得到消息。）



轮询消费者的顺序图

当应用轮询消息时，消费者会一直阻塞，直到它获得返回的消息（或者其他某个条件满足，如时间限制）。应用一旦接收到消息，就能处理这个消息。当完成了消息的处理并想接收下一个消息时，应用可再次轮询消息。

通过使用轮询的消费者，应用可以控制同时被消费的消息数量，为此只需限制轮询的线程数。采用这种方法可以防止接收方应用因为过多的请求而超负荷，多余的消息可以在队列中排队等待，直到接收者能够处理。

接收者应用一般对每个要监控的通道都使用（至少）一个线程，但是它也能使用单个线程监控多个通道，当所监控的通道经常为空时，这种做法可以节省线程。要轮询单个通道，假设线程在消息到达前不做任何操作，因此可以使用阻塞型的接收方法，即在消息到达之前这个接收方法会一直阻塞。要使用单个线程轮询多个通道，或者为了在等待消息到达时处理其他工作，可以使用有超时设置的接收方法，或者是 receiveNoWait()方法，如果

通道为空，线程将检查下一个通道或完成其他工作。

如果过多地轮询，或者使线程阻塞太长时间，会导致消费者的效率极低，此时采用事件驱动的消费者会更高效。多个轮询的消费者可能成为竞争消费者。消息分派器可以实现为轮询消费者。轮询消费者可以是选择性消费者，也可以是持久订购者。轮询消费者也能作为事务性客户端，这样，消费者就能控制何时真正从通道中删除消息。

例：JMS Receive

在 JMS 中，消息消费者使用 MessageConsumer.receive 同步地消费消息[JMS1.1]、[Hapner]。

MessageConsumer 有三个不同的接收方法：

1. receive()——在消息到达前一直阻塞，然后返回消息。
2. receiveNoWait()——检查消息是否到达，并立即返回到达的消息或 null。
3. receive(long)——在消息到达前一直阻塞，然后返回消息，或者等到超时并返回 null。

例如，创建消费者并接收消息的代码非常简单：

```
Destination dest = // 获得目标
Session session = // 创建会话
MessageConsumer consumer = session.createConsumer(dest);
Message message = consumer.receive();
```

例：.NET Receive

在.NET 中，消费者使用 MessageQueue.Receive 同步地消费消息[SysMsg]。

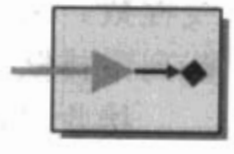
MessageQueue 客户端有多个接收方法，最简单的两个是：

1. Receive()——在消息到达前一直阻塞，然后返回消息。
2. Receive(TimeSpan)——在消息到达前一直阻塞，然后返回消息，或者等待超时并抛出 MessageQueueException。

要从已有的队列接收消息的代码非常简单：

```
MessageQueue queue = // 获得队列
Message message = queue.Receive();
```

10.6 事件驱动的消费者



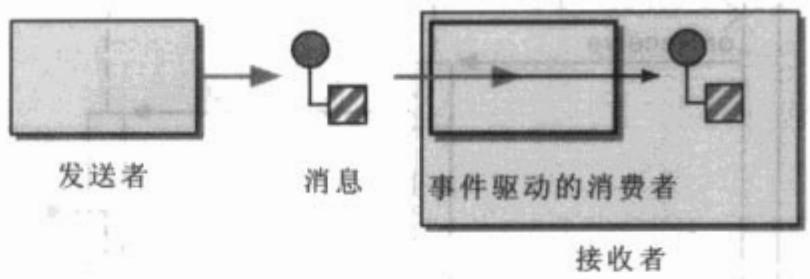
一旦传送了消息，应用就要消费这些消息。

当消息到达时，如何让应用自动消费这些消息？

轮询消费者的缺点是，通道为空时，消费者将阻塞线程，而且/或者当消费者轮询尚未到达的消息时，将消耗处理时间。基于轮询，客户端能控制消费的速度，但是没有可消费的消息时，就会造成资源的浪费。

与其不断查询通道中是否有可消费的消息，还不如在消息到达时让通道通知客户端。为此，消费者不必轮询消息，只要有消息到达，就会把消息返回给消费者。

应用应当使用事件驱动的消费者，当消息传输到通道中时，事件驱动消费者能自动处理消息。



事件驱动的消费者也称为异步接收者，因为只有在回调线程传送一个消息时这个接收者才会有一个运行线程。之所以把它称为事件驱动的消费者，因为对于接收者来说，消息传送就相当于触发接收者采取行动的事件。

事件驱动的消费者是一个对象，当消息到达消费者的通道时，该对象由消息传递系统调用。消费者通过应用提供的回调 API 把消息传递给应用。（消息传递系统要如何得到消息与具体的实现相关，可能是事件驱动的，也可能不是事件驱动的。消费者只知道，在由消息传递系统调用并为之传递消息之前可以一直等待，没有活动的线程。）

消息传递系统要调用事件驱动的消费者，而事件驱动的消费者会调用一个应用特定的回调函数。为了把二者联系起来，消费者提供了一个应用特定的回调实现，并遵循消息传递系统定义的已知 API。

事件驱动消费者的代码由两部分组成：

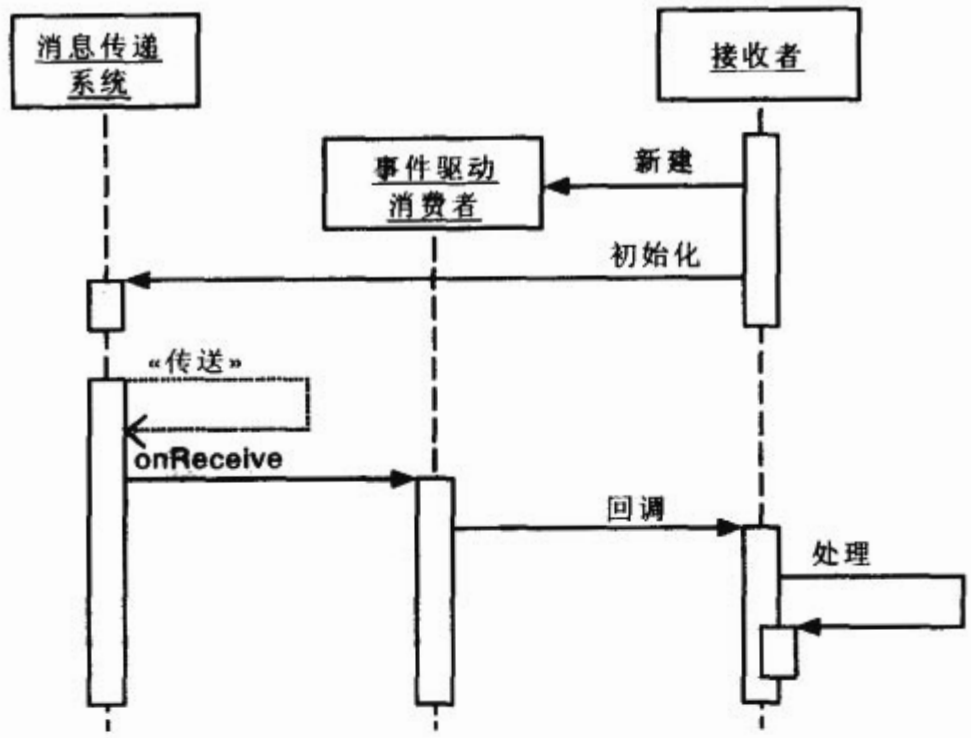
1. **初始化**——应用创建一个应用特定的消费者，并把它与特定的消息通道相关联。一旦运行了该代码，消费者就准备好可以可以接收一系列的消息。

2. **消费**——消费者接收消息，并把消息传递给应用处理。对于每个要消费的消息，都要运行一次该代码。

应用创建自己的定制消费者，并把它与通道关联。一旦消费者得到初始化，它（和应用）就能进入睡眠状态，没有运行线程，只有在消息到达时才被调用。

当传送一个消息时，消息传递系统调用消费者的 message-received-event 方法，并把消息作为参数传递给它。消费者使用应用的回调 API 把消息传递给应用。现在应用得到了消息并能处理这个消息。一旦应用完成消息的处理，它和消费者可再次进入睡眠状态，等待下一个消息的到达。通常情况下，消息传递系统将不会通过单个消费者运行多个线程，因

此消费者每次只能处理一个消息。



事件驱动消费者的顺序图

当事件驱动的消费者可用时，它能自动消费消息。为了更细粒度地控制消费的速度，可以使用轮询消费者。事件驱动的消费者可以作为竞争消费者。消息分派器能实现为事件驱动的消费者。事件驱动的消费者可以作为选择性消费者，也可以作为持久订购者。与轮询消费者不同，事务性客户端不能与事件驱动的消费者很好地协作，详情可见 JMS 的例子。

例：JMS MessageListener

在 JMS 中，事件驱动的消费者是一个实现了 MessageListener 接口的类[Hapner]。该接口只声明了一个方法：onMessage(Message)。消费者实现了 onMessage 方法来处理消息。下面是一个 JMS 执行者的例子：

```
public class MyEventDrivenConsumer implements MessageListener {
    public void onMessage(Message message) {
        // 处理消息
    }
}
```

事件驱动消费者的初始化部分创建了所需的执行者对象（这是一个 MessageListener 实例），并把它与消息消费者期望的通道关联：

```
Destination destination = // 获得目标
Session session = // 创建会话
MessageConsumer consumer = session.createConsumer(destination);
```

```
MessageListener listener = new MyEventDrivenConsumer();
consumer.setMessageListener(listener);
```

现在，当消息传送给目标时，JMS 提供者将调用 MyEventDrivenConsumer.onMessage，并把消息作为参数传递给它。

请注意，在 JMS 中事件驱动的消费者如果也是一个事务性客户端，则不会像期望的那样工作。通常情况下，当事务中代码抛出一个异常时，事务将回滚，但是 MessageListener.onMessage 的签名没有提供可以抛出的异常（如 JMSException），运行时异常被认为是程序员的编程错误。如果出现运行时异常，JMS 提供者只能传送下一个消息，因此引发异常的消息将丢失[JMS1.1]、[Hapner]。为了成功地实现事务性的事件驱动消费者，可以使用消息驱动 EJB[EJB2.0]、[Hapner]。

例：.NET ReceiveCompletedEventHandler

使用.NET，事件驱动的消费者的执行者部分实现了一个方法作为 ReceiveCompletedEventHandler 委托。该委托方法必须接受两个参数：一个是 MessageQueue 对象，另一个是来自 ReceiveCompleted 事件的 ReceiveCompletedEventArgs 实参[SysMsg]。这个委托方法使用这些参数从队列中获得消息，并处理消息。下面是一个.NET 执行者的例子：

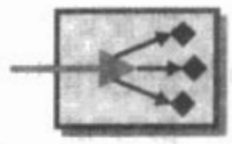
```
public static void MyEventDrivenConsumer(Object source,
    ReceiveCompletedEventArgs asyncResult)
{
    MessageQueue mq = (MessageQueue) source;
    Message m = mq.EndReceive(asyncResult.AsyncResult);
    // 处理消息
    mq.BeginReceive();
    return;
}
```

事件驱动客户端的初始化部分指定了要运行委托方法的队列，以处理 ReceiveCompleted 事件：

```
MessageQueue queue = // 获得队列
queue.ReceiveCompleted +=
    new ReceiveCompletedEventHandler(MyEventDrivenConsumer);
queue.BeginReceive();
```

现在，当消息传送到队列时，队列将发出 ReceiveCompleted 事件，这将运行 MyEventDrivenConsumer 方法。

10.7 竞争消费者



应用使用了消息传递，但是，它处理消息的速度赶不上消息添加到通道中的速度。

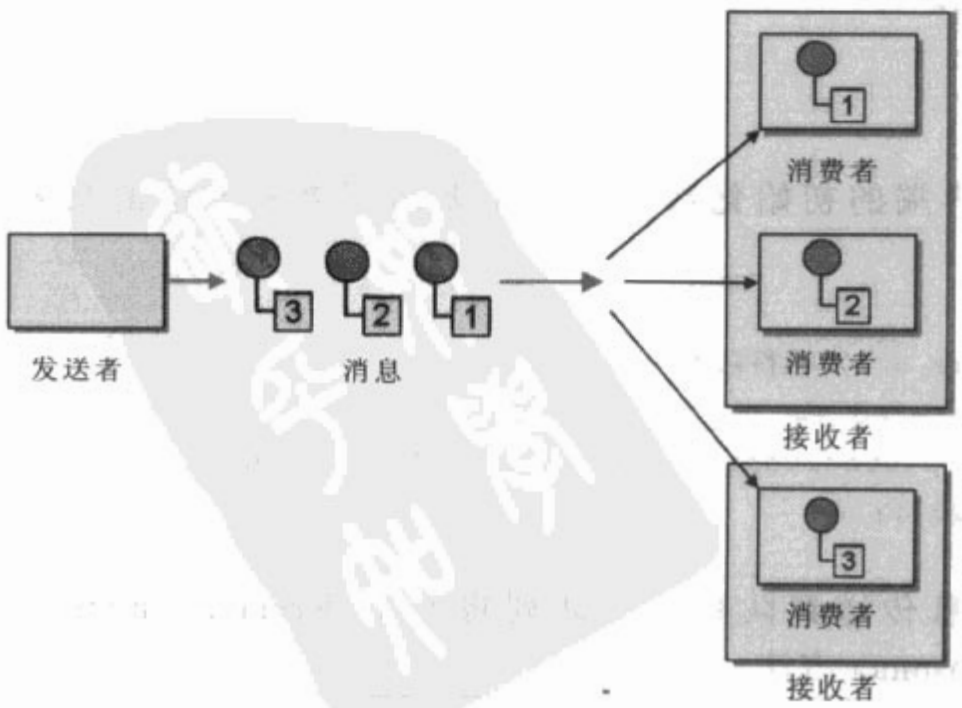
消息传递客户端如何同时处理多个消息？

消息通过消息通道按顺序到达，因此消费者自然愿意按顺序来处理。但是，顺序处理消息可能太慢了，消息在通道中累积，这会使消息传递系统成为瓶颈，进而影响了整个应用的吞吐量。如果通道有多个发送者，或者由于网络中断的原因导致大量积压的消息同时传输，或者因为接收者中断引起消息积压，也可能是因为每个消息占用的消费和执行时间要远远大于创建和发送消息的时间，这些情况都会引起消息积压的问题。

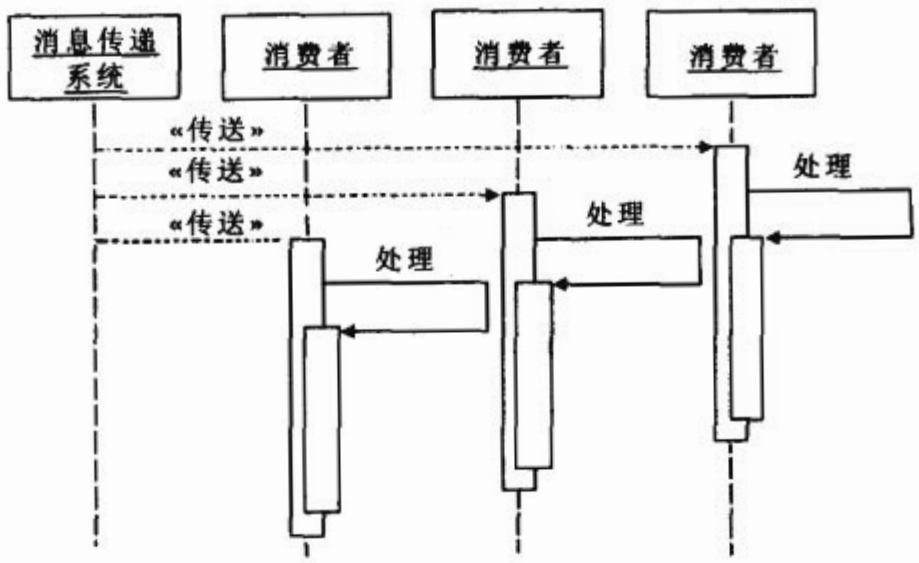
应用可以使用多个通道，但是其中某个通道可能会成为瓶颈，而另一个通道却一直为空，发送者并不清楚在这些相当的通道中应该使用哪一个。不过，如果有多个通道则有一个好处，这样能支持多个消费者（每个通道对应一个消费者），因此可以同时处理多个消息。不过即使可以这样做，应用定义的通道数量也会限制吞吐量。

需要有一种方法来支持一个通道有多个消费者。

创建多个竞争的消费者共用一个通道，这样消费者就能并发地处理多个消息。



竞争的消费者是指多个消费者，这些消费者都创建为接收同一个点对点通道的消息。通道传送一个消息时，所有的消费者都可能接收到。消息传递系统的具体实现会决定应该由哪个消费者接收消息，但是，消费者实际上要通过相互竞争才能成为接收者。一旦某个消费者接收到消息，它就能委托应用的其他部分帮助处理消息。（该解决方案只适用于点对点通道，对发布-订购通道上的多个消费者来说，只需创建各个消息的更多副本。）



竞争消费者的顺序图

每个竞争消费者在自己的线程中运行，因此它们能同时消费消息。当通道传送一个消息时，消息传递系统的事务性控制会确保只有一个消费者能成功地接收到消息。该消费者正在处理消息时，通道可以传送其他消息，这样其他的消费者可同时消费和处理这些消息。通道会协调消费者，确保每个消费者接收到不同的消息；消费者之间不必相互协调。

每个消费者能同时处理不同的消息，因此系统的瓶颈已经不是消费者花多少时间处理一个消息，而是通道能以多快的速度给消费者递送消息。有数目限制的消费者仍然是瓶颈，但是只要有多余的计算资源，就能增加消费者的数量，缓解瓶颈。

为了能同时运行，每个消费者必须在自己的线程中运行。对轮询消费者来说，这意味着每个消费者必须用自己的线程同时进行轮询。对事件驱动的消费者来说，消息传递系统必须为每个并发的消费者创建一个线程，通过这个线程把消息传递给消费者，并由消费者处理。

一个复杂的消息传递系统将检测出通道上的竞争消费者，并提供一个消息分派器，从而确保每个消息只会发送给一个消费者。这样就能避免多个消费者可能引发的竞争冲突，它们可能都认为自己才是某个消息的消费者。不太复杂的消息传递系统将允许多个消费者尝试消费同一个消息。当出现这种情况时，最先提交事务的消费者才能成功地消费消息，其他消费者则无法完成消费，必须回滚它们的事务。

如果消息传递系统支持多个消费者消费相同的消息，这会使事务性客户端变得非常低效。客户端以为自己得到了一个消息，然后消费这个消息，并花费大量的时间来处理消息，可是最后却无法提交事务（因为消息已经被其他竞争者消费了）。频繁的回滚操作会降低系

统吞吐量，而这种解决方案的本来目的却是为了提高吞吐量。因此，必须仔细评测竞争的事务性消费者的性能，对于不同的消息传递系统的具体实现和配置，性能可能存在很大差异。

不但竞争的消费者能把开销摊分到同一应用的多个消费者线程中，它们还能把消费开销摊分到多个应用（例如进程）中。采用这种方法，如果某个应用无法提供足够快的消费速度，可以采用多个消费者应用——其中每个应用还可以使用多个消费者线程——这样就能克服这个问题。让多个应用运行在多台计算机上，并且使用多个线程消费消息，从理论上讲这就能提供无限的消息处理能力，唯一的限制是消息传递系统把消息从通道传送给消费者的能力。

竞争消费者之间的协调依赖于每个消息传递系统的具体实现。如果客户端自己想实现这种协调，则应该使用消息分派器。竞争的消费者可以作为轮询消费者、事件驱动的消费者或这两种消费者的组合。竞争的事务性客户端会花费大量的资源来处理一些纯属浪费的消息，因为接收这些消息的操作最后无法成功提交而必须回滚。

例：简单的 JMS 竞争消费者

下面是一个简单的竞争消费者的例子，采用 Java 实现。外部的驱动者/管理者对象（这里没有给出）运行了两个这样的消费者。每个消费者都在自己的线程中运行，调用 stopRunning()可以把它停止下来。

JMS 会话必须是单线程的[JMS1.1]、[Hapner]。单一的会话可以保证消息消费的顺序[JMS1.1]、[Hapner]。因此，为了让每个在自己的线程中运行的竞争消费者能正确操作，也为了让消费者能够同时消费消息，各个消费者必须拥有自己的会话（相应地有自己的 MessageConsumer）。JMS 规范没有为并发的 QueueReceiver（如竞争的消费者）规定操作语义，甚至根本没有要求采用这种方法。因此，使用该技术的应用不一定适用于各种环境，采用不同的 JMS 提供者，产生的效果也是不一样的[JMS1.1]、[Hapner]。

消费者类实现了 Runnable 接口，因此它能在自己的线程中运行，从而实现了并发的消费者。所有消费者共享相同的 Connection，但是每个消费者创建自己的会话（Session），这一点是非常重要的，因为每个会话只能支持一个线程。每个消费者不断从队列中接收消息并加以处理。

```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.naming.NamingException;

public class CompetingConsumer implements Runnable {

    private int performerID;
    private MessageConsumer consumer;
```

```
private boolean isRunning;

protected CompetingConsumer() {
    super();
}

public static CompetingConsumer newConsumer(int id, Connection connection,
                                            String queueName)
    throws JMSException, NamingException {
    CompetingConsumer consumer = new CompetingConsumer();
    consumer.initialize(id, connection, queueName);
    return consumer;
}

protected void initialize(int id, Connection connection, String queueName)
    throws JMSException, NamingException {
    performerID = id;
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Destination dispatcherQueue = JndiUtil.getDestination(queueName);
    consumer = session.createConsumer(dispatcherQueue);
    isRunning = true;
}

public void run() {
    try {
        while (isRunning())
            receiveSync();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private synchronized boolean isRunning() {
    return isRunning;
}

public synchronized void stopRunning() {
    isRunning = false;
}

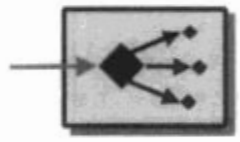
private void receiveSync() throws JMSException, InterruptedException {
    Message message = consumer.receive();
    if (message != null)
        processMessage(message);
}
```

```
    private void processMessage(Message message)
      throws JMSException, InterruptedException {
        int id = message.getIntProperty("cust_id");

        System.out.println(System.currentTimeMillis() + ": Performer #"
          + performerID + " starting; message ID " + id);
        Thread.sleep(500);
        System.out.println(System.currentTimeMillis() + ": Performer #"
          + performerID + " processing.");
        Thread.sleep(500);
        System.out.println(System.currentTimeMillis() + ": Performer #"
          + performerID + " finished.");
    }
}
```

因此，实现一个简单的竞争消费者是很容易的。关键的一点是要让消费者实现 Runnable 接口，并在其自己的线程中运行。

10.8 消息分派器



一个应用使用了消息传递。这个应用需要同一消息通道上的多个消费者能够以协同的方式运行。

单个通道上的多个消费者如何才能协调其消息处理？

单个点对点通道上的多个消费者会成为竞争的消费者。如果消费者可互相交换（即一个消息可以由任何消费者消费），这种竞争不存在问题，但是如果指定某个消费者只能消费某种消息，还要采用其他措施。

单个发布-订购通道上的多个消费者并不像期望的那样运行，它们不但不能摊分消息处理的开销，反而会增加开销。

选择性消费者可以用作为专门的消费者。但是，并不是所有的消息传递系统都能支持这种特性。即使是支持这种特性的消息传递系统，也可能不支持按照消息体中的值进行选择。它们的选择值达式可能太过简单，不足以区分消息，也可能重复计算这些表达式的效率会很低。可能需要以协调的方式仔细地设计大量表达式，从而避免重叠的条件，并避免遗留未处理的选择值。对不能被其他消费者处理的选择值或者意外的选择值，必须实现一

个缺省的消费者来处理。

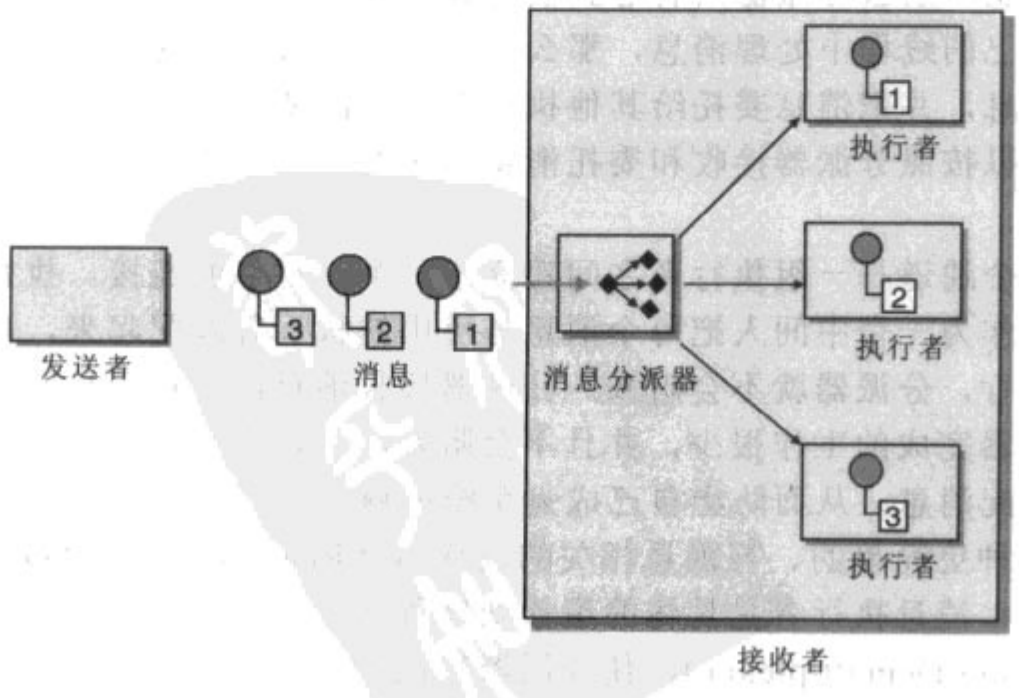
数据类型通道（Datatype Channels）可用来区分不同类型的消息，使消费者能专门处理某种类型的消息。但是系统中的消息类型可能太多了，而且还会经常改变，从而无法为每种类型创建不同的通道。消息的类型可能会基于动态变化的标准而改变，因此很难采用一组静态的通道来处理。企业可能已经使用了大量的通道，这给消息传递系统带来了沉重的负担，如果再针对不同的消息类型让一个通道变成多个通道，就会创建太多的通道。

如果消费者能协同工作，则能解决上述所有问题。如果能知道其他消费者已经完成了相应的操作，就能避免重复的操作。消费者可以作为专门的消费者，如果消费者得到了类型不匹配的消息，可以把消息交给其他能处理该消息的消费者。如果应用有太多的输入通道，可以让消费者共用单个通道，以减少通道的数量。这些消费者可相互协作，以确保正确的消息能到达正确的消费者。

但是，消费者是非常独立的对象，它们很难协调。如果让专用的消费者能够通用，从而能处理任何消息，并能把消息传递给其他消费者，这会增加各个消费者设计和处理的开销。它们都必须了解其他消费者，才能完成消息转递任务，而且要清楚其他消费者是否正在忙于处理消息，以避免把消息转交给正在处理其他消息的消费者。使得消费者一起运行，将会从根本上改变典型消费者的设计。

仲裁者模式[GoF]提供了一定的帮助。仲裁者会协调一组对象，使它们不必考虑如何相互协调。为了进行消息传递，我们需要仲裁者协调同一通道上的消费者，使每个消费者能集中处理特定类型的消息，而由协调者保证把正确的消息交给正确的消费者处理。

创建通道上的消息分派器，它从通道中消费消息，并把消息分配给执行者。

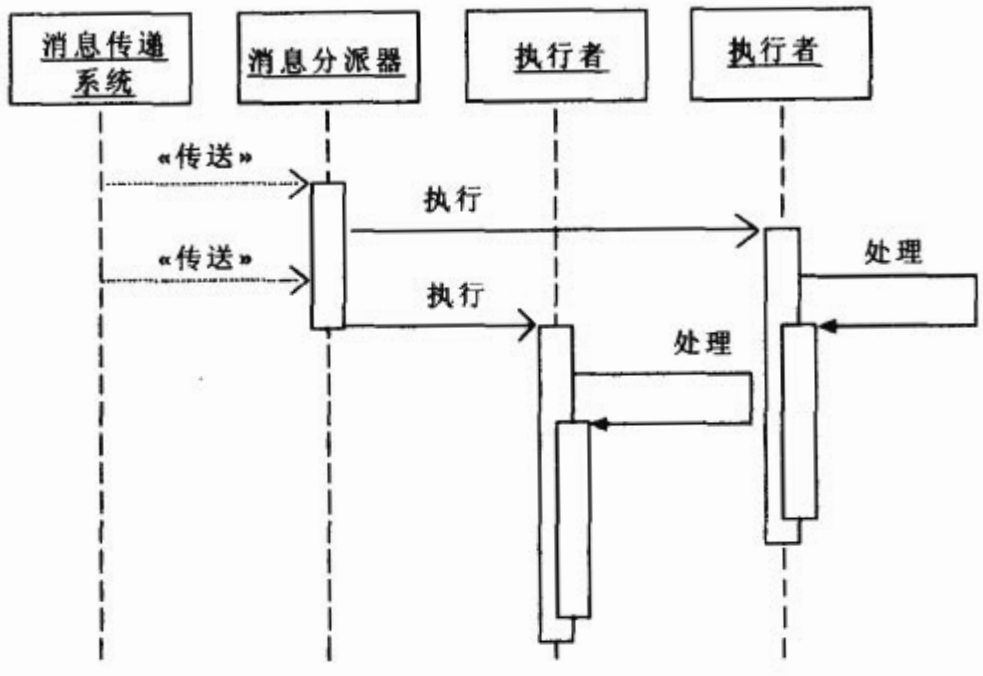


一个消息分派器由两部分组成：

1. **分派器**——这个对象从通道中消费消息，并把每个消息分配给执行者。

2. **执行者**——这个对象从分派器获得消息并处理。

消息分派器接收到消息时，它得到一个执行者，并把消息交给执行者处理。执行者可以委托应用的其他部分帮助处理消息。执行者既可以是分派器新创建的，也可能从可用的执行者缓冲池中选出。每个执行者在自己的线程中运行，并发地处理消息。所有的执行者可以处理各种消息，分派器也可以按照消息的特性把消息交给专门的执行者处理。



消息分派器的顺序图

分派器接收到消息时，它把消息委托给一个可用的执行者处理。如果执行者使用分派器的线程处理消息，那么分派器只有等到执行者完成消息的处理后才能继续执行。相反，如果执行者在自己的线程中处理消息，那么一旦分派器启动了这个线程，分派器就能立即开始接收其他消息，并把消息委托给其他执行者，从而使消息能得到并发的处理。采用这种方式，消息可以按照分派器接收和委托消息的速度进行消费，与处理每个消息的时间长度无关。

分派器在单个通道与一组执行者之间建立了一个一对多的连接。执行者完成大多数工作，分派器只是作为一个中间人把每个消息与可用的执行者匹配起来，而且只要执行者在自己的线程中运行，分派器就不会阻塞。分派器接收消息，并把消息发送给处理消息的执行者。由于分派器完成的工作很少，并且不会阻塞，所以它基本上能按照消息传递系统递送消息的速度分配消息，从而防止自己成为系统的瓶颈。

该模式是一种更简单的、与消息相关的反应器（Reactor）模式[POSA2]，其中消息分派器就是反应器，消息执行者是具体的事件处理程序。消息通道作为同步事件多路选择器（Synchronous Event Demultiplexer），使分派器每次能得到一个消息。消息本身类似于句柄（Handle），但是比句柄更简单。一个真正的句柄是指向资源数据的引用，而消息一般直接包含了数据（但是，消息不一定直接存储数据。如果消息的数据存储在外部，并且该消息

是一个声明标签，就会包含数据的引用，这样就更类似于一个反应器句柄）。不同类型的句柄选择使用不同类型的具体事件处理程序，而消息通道是数据类型通道，因此所有的消息（句柄）都属于相同的类型，通常只有一种具体的事件处理程序。

尽管数据类型通道将通道设计为所有的消息都有相同的类型，而且所有消费者都只处理这种类型的消息，但是反应器模式指出，可以使用消息分派器在同一通道上支持多种数据类型，并使用特定于类型的执行者来处理消息。每个消息必须指定类型，分派器检测消息的类型并把它分配给特定类型的执行者处理。采用这种方法，带专用执行者的分派器可以取代数据类型通道，并作为选择性消费者的特殊实现。

消息分派器与竞争消费者的一个区别是能否分布到多个应用中。一组竞争的消费者能分布到多个进程（如应用）中，但是一组执行者通常与分派器在相同的进程中一起运行（即使它们运行在不同的线程中）。如果执行者与分派器不在同一进程中运行，分派器就必须采用分布式远程过程调用的方法与执行者通信，而这正是消息传递最想避免的情况。

由于分派器是单个消费者，它能与点对点通道和发布-订购通道很好地协作。对点对点消息传递来说，分派器可代替竞争消费者，如果消息传递系统不能很好地应对多个消费者，或者它不能一致地处理跨不同消息传递系统的多个消费者，那么采用分派器代替这些消费者就更加适合。

无论分派器本身是事件驱动的消费者还是轮询消费者，分派器都会使执行者像事件驱动消费者那样工作。因此，要把分派器实现为事务性客户端的一部分可能很困难。如果客户端是事务性的，理想情况下，分派器应该允许执行者在完成事务前处理消息，并且只有执行者成功分派器才能提交事务。如果执行者无法处理消息，分派器应该回滚事务。由于每个执行者都要回滚各自不同的消息，分派器要为每个执行者建立会话，并使用执行者的会话接收执行者的消息，进而完成它的事务。由于事件驱动的消费者一般很难与事务性客户端协作，因此分派器最好不是事件驱动的消费者，而是轮询消费者。

把执行者实现为事件驱动的消费者很有用。在 JMS 中，要把执行者实现为 MessageListener。消息监听者有一个名为 onMessage(Message)的方法，它接收一个消息并执行必要的处理。这样就把分派器与执行者很好地分离开。同样地，在.NET 中，即使分派器不会真的发出 ReceiveCompleted 事件，执行者也应该是一个 ReceiveCompletedEvent Handler 委托。但是，这些事件驱动的方法可能无法与在同一线程中运行执行者所需的 API 兼容。

为了避免实现自己的消息分派器，可以考虑在数据类型通道上使用竞争消费者或选择性消费者。消息分派器可以作为轮询消费者或事件驱动的消费者。消息分派器不适合作为事务性客户端。

例：.NET 分派器

通常，消息分派器把消息分配给执行者（见 Java 示例）。.NET 提供了另一种选择：分派器可以使用 Peek 检测消息并获得消息的 ID，然后把消息的 ID（而不是整个消息）分配给执行者。执行者使用 ReceiveById 消费所分配的特定消息。采用这种方法，每个执行者

不但要负责处理消息，还负责消费消息，这样就能帮助解决并发问题，特别是当消费者是事务性客户端时更有意义。

例：简单的 Java 分派器

这个简单的例子介绍了如何采用 Java 实现一个分派器和执行者。更复杂的分派器可能会缓冲多个执行者，并能查看当前有哪些执行者能处理消息，还可能使用线程池。这个简单的例子省略了这些细节，但是每个执行者可以在各自的线程中运行，从而能并发运行。

控制分派器的驱动者/管理者（这里没有给出）将反复运行 receiveSync()。每次运行时，分派器将调用 receive()接收下一个消息，实例化一个新的 performer 实例来处理消息，然后在自己的线程中启动 performer。

```
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.naming.NamingException;

public class MessageDispatcher {

    MessageConsumer consumer;
    int nextID = 1;

    protected MessageDispatcher() {
        super();
    }

    public static MessageDispatcher newDispatcher(Connection connection,
      String queueName)
        throws JMSException, NamingException {
        MessageDispatcher dispatcher = new MessageDispatcher();
        dispatcher.initialize(connection, queueName);
        return dispatcher;
    }

    protected void initialize(Connection connection, String queueName)
        throws JMSException, NamingException {
        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination dispatcherQueue = JndiUtil.getDestination(queueName);
        consumer = session.createConsumer(dispatcherQueue);
    }

    public void receiveSync() throws JMSException {
```

```
        Message message = consumer.receive();
        Performer performer = new Performer(nextID++, message);
        new Thread(performer).start();
    }
}
```

performer 必须实现 Runnable 接口，只有这样它才能在自己的线程中运行。Runnable 接口的 run()方法只是调用 processMessage()。当它结束时，performer 就可以垃圾回收了。

```
import javax.jms.JMSException;
import javax.jms.Message;

public class Performer implements Runnable {

    private int performerID;
    private Message message;

    public Performer(int id, Message message) {
        performerID = id;
        this.message = message;
    }

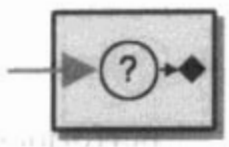
    public void run() {
        try {
            processMessage();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void processMessage() throws JMSException,
 InterruptedException {
        int id = message.getIntProperty("cust_id");

        System.out.println(System.currentTimeMillis() + ": Performer #"
          + performerID + " starting; message ID " + id);
        Thread.sleep(500);
        System.out.println(System.currentTimeMillis() + ": Performer #"
          + performerID + " processing.");
        Thread.sleep(500);
        System.out.println(System.currentTimeMillis() + ": Performer #"
          + performerID + " finished.");
    }
}
```

实现一个简单的分派器和执行者很容易。关键是执行者要实现 Runnable 接口，从而在自己的线程中运行。

10.9 选择性消费者



应用使用了消息传递。它从消息通道中消费消息，但是并不想消费消息通道中的所有消息，而只想消费其中的一部分消息。

> 消息的消费者如何选择想要接收的消息？

默认情况下，如果消息通道只有一个消费者，那么通道中的所有消息都将传输给这个消费者。另外，如果该通道上有多个竞争消费者，那么所有消息都能被任意一个消费者消费，每个消息都会到达某个消费者。消费者一般无法对所要消费的消息做出选择，它总是获得下一条可用的消息。

如果消费者想接收通道中的任意一个消息或所有的消息，采用这种做法不会有什么问题。但是，倘若消费者只想消费某些消息，这样做就不合适了，因为消费者通常无法对所要接收的消息做出选择。消费者为什么只想接收某些消息呢？可以考虑一个处理贷款请求消息的应用，它对低于10万美元的贷款和高于10万美元的贷款有不同的处理方法。一种解决办法是让应用提供两个不同类型的消费者，其中一个负责小额贷款的处理，另一个负责大额贷款的处理。然而，由于所有消费者都能接收到任何一个消息，应用又怎么保证正确的消息能发送给正确的消费者呢？

最简单的方法就是让每个消费者消费它接收到的任何消息。如果它接收到错误类型的消息，可以把该消息转发给能处理这种消息类型的消费者。但是，要实现这种功能很困难，消费者实例彼此并不知道对方的存在，并且要找到一个不忙的消费者并不容易。消费者意识到它不需要这种消息时，也许可以把消息退还给通道，但是很可能它会再次消费相同的消息。每个消费者也可以获得每个消息的副本，并且只需把不需要的消息副本丢掉即可。这种做法虽然可行，但是会产生大量重复的消息，那些最终会被丢掉的消息也会浪费大量的处理资源。

也许消息传递系统可以为每种类型的消息定义不同的通道。发送者确保把每个消息发送到正确的通道中，接收者可以确保从特殊通道上接收到的消息正是所需的类型。但是，这种解决方案不是很灵活。当系统正在运行时，接收者可能修改它们的选择标准，这就要求定义新的通道和重新发布已经在通道中的消息。它还要求发送者必须了解接收者的选择标准是什么，以及这些标准何时改变。另外，这些标准必须是接收者的属性，而不是通道的属性，通道中的消息也要给出它们遵循的标准是什么。

我们需要这样一种方法，能够把适应不同标准的消息全部发送到相同的通道中，消费者能指定它们感兴趣的标准，而且每个消费者只接收到满足其标准的消息。

把消费者实现为选择性消费者，过滤在通道上传送的消息，从而使其只能接收到符合其标准的消息。



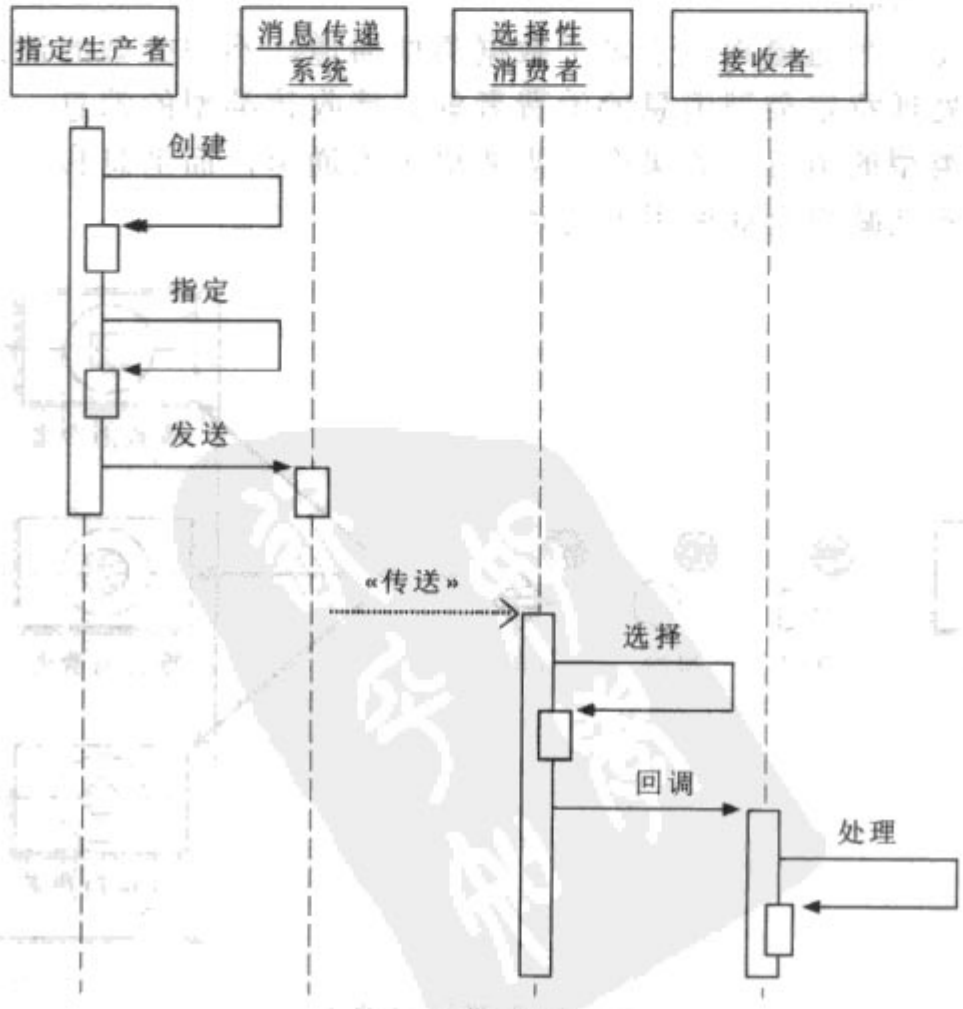
该过滤过程由三个部分组成：

1. **指定生产者**——在发送消息前指定消息的选择值。

2. **选择值**——消息中被指定的一个或多个值，通过选择值，消费者可以确定是否选择该消息。

3. **选择性消费者**——只接收满足其选择标准的消息。

消息发送者在发送每个消息前指定消息的选择值。当消息到达时，选择性消费者测试消息的选择值，查看该值是否满足消费者的选择标准。如果满足，消费者就接收消息并把它传递给应用来处理。



选择性消费者的顺序图

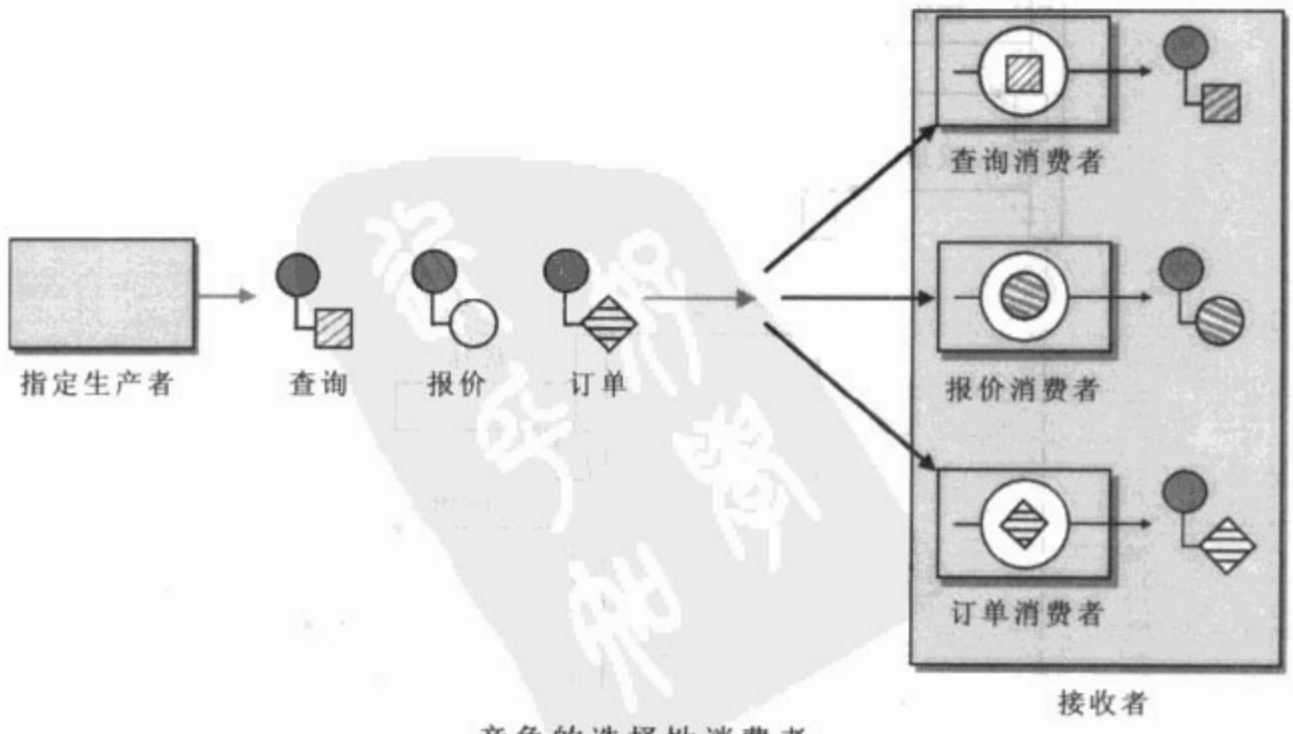
发送者创建消息时，还要设置消息的选择值，然后发送消息。消息传递系统传送消息时，选择性消费者测试消息的选择值，以确定是否选择该消息。如果消息满足条件，消费者将接收消息，并使用回调把它传递给应用。

选择性消费者一般成组使用——一个消费者负责过滤一组标准，另一个消费者负责过滤另一组标准，以此类推。对贷款处理示例来说，一个消费者将选择“amount<=$100,000”的消息，另一个消费者将选择“amount>$100,000”的消息。然后，每个消费者只会得到它感兴趣的贷款请求消息。

当多个选择性消费者用于一个点对点通道时，它们既是竞争消费者也是选择性消费者。如果两个消费者的标准有重复，而且一个消息的选择值能同时满足它们的标准，那么其中任何一个消费者都可以消费这个消息。应当适当地设计消费者，从而对于每个合法的选择值，至少有一个消费者能消费相应的消息。否则，如果一个消息的选择值与任何标准都不匹配，就始终无法被消费，并永远驻留在通道中（或至少驻留到消息到期）。

当多个选择性消费者用于一个发布-订购通道时，各个消息会传输给每一个订购者，但是订购者可以忽略与其标准不匹配的消息副本。一旦消费者决定忽略一个消息，消息传递系统可以丢掉该消息，因为这个消息已经成功地传送，而且不会被消费。消息传递系统可以优化这个过程，如果知道消费者会忽略某个消息，就根本不向它传送此消息，从而减少要产生和传递的消息副本的数量。无论是使用可靠传输、持久的订购者和/或消息到期设置，都可能要丢掉忽略的消息。

选择性消费者使一个通道相当于多个数据类型通道。不同类型的消息可能有不同的选择值，因此，只能处理特定类型消息的消费者就只接收该类型的消息。这有利于用少量的通道发送大量不同类型的消息。如果企业需要更多的通道，而消息传递系统不能提供这么多通道，这种方法还能减少企业所用的通道。



竞争的选择性消费者

如果试图对某种消费者应用隐藏某种类型的消息，使用选择性消费者来模拟数据类型通道并不是好的做法。虽然消息传递系统可以确保只有授权的应用才能成功地接收到通道中的消息，但是它一般不会对消费者的选择标准授权，因此那些有权访问通道的恶意消费者可能会修改它的标准，这样就能访问那些它本来无权访问的消息。因此，要采用不同的数据类型通道安全地锁定接收应用。

使用选择性消费者的一种替代做法是使用消息分派器。选择标准建立在分派器中，分派器使用这些标准确定每个消息的执行者。如果某个消息不满足任何执行者的标准，分派器并不是把它留在通道中或者丢掉它，而是把它重新路由给非法消息通道。在消息分派器和竞争消费者之间权衡时，实际上问题在于：你是想让消息传递系统负责分配工作，还是由你自己实现分配工作。如果消息传递系统不支持选择性消费者，那么你别无选择，只能使用消息分派器自行实现。

前面介绍过，如果一个通道的所有选择性消费者都与消息的选择值不匹配，消息将被忽略，就好像该通道没有接收者一样。在面向过程的程序设计中也存在类似的问题，即 case 语句的所有分支条件可能均无法匹配所测试的值。因此，case 语句提供了一个默认的条件，其他所有分支无法匹配的值都将与此条件匹配。在消息传递中可以使用相同的方法，尝试创建某种默认的消息消费者，它处理与其他消费者均不匹配的消息。然而，这样的默认消费者并不能像期望的那样工作，因为它需要一个能匹配所有选择值的表达式，因此它会与所有其他的消费者竞争。所以，要实现一个默认的消费者，可以使用消息分派器，由该分派器实现一个有默认选项的case语句，其他分支不能处理的所有消息均归到这个默认分支，并使用默认消费者处理。

另一种能代替选择性消费者的模式是消息过滤器。它们能完成相同的目标，但是采用的方式不同。对选择性消费者来说，所有消息都传输给接收者，但是每个接收者将忽略不需要的消息。消息过滤器位于来自发送者的通道和到达接收者的通道之间，它只是把所需的消息从发送者的通道转发到接收者的通道中。因此，不需要的消息始终不会传输到接收者的通道，接收者也不用忽略任何消息。消息过滤器有利于去除任何接收者都不想要的消息。当某个接收者想要忽略某些消息，但是其他的接收者却想接收这些消息时，可以使用选择性消费者。

也可以考虑使用基于内容的路由器代替选择性消费者。这种路由器像过滤器一样，可确保通道只传递接收者想要的消息，进而增加安全性，提高消费者的性能。不过，选择性消费者更灵活，因为每个过滤选项只需要一个新的消费者（当系统运行时，很容易创建这样的消费者），然而对基于内容的路由器来说，每个新的选项都需要一个新的输出通道（当系统运行时，创建和使用这样的通道并不容易），而且对于新通道还需要新的消费者。假设需求发生了变化，还要处理不同于小额和大额贷款的中等额度贷款（介于$50,000～$150,000 之间）。采用基于内容的路由器，需要为中额贷款创建一个新的通道，还需要该通道相应的新的消费者，并调整路由器区分贷款额度的方式。还要考虑修改发生作用后可能出现的情况，因为有些消息可能已经路由到原来的通道中，而且还未被消费，现在看来这些消息则处在错误的通道中。而采用选择性消费者，你只需把两种类型的消费者（少于

$100,000 和大于$100,000）替换为三种类型（少于$50,000，$50,000～150,000 之间，以及大于$150,000）就可以了。基于内容的路由器是一种更静态的解决方法，而选择性消费者的灵活性更强。

理想情况下，消息的选择值应该位于消息的首部，而不在消息体中，因此选择性消费者不用解析消息体（也不用知道如何解析）就能处理选择值。

选择性消费者使一个通道表现得像是多个数据类型通道。选择性消费者允许消息也能被其他接收者接收到，而消息过滤器不会向接收者传送它不需要的消息。使用选择性消费者比基于内容的路由器更灵活。选择性消费者可以实现为轮询消费者或事件驱动的消费者，也可以成为事务性客户端的一部分。要自行实现过滤操作，可以使用消息分派器。

例：区分类型

一个股票交易系统只有有限的通道，可能需要对报价和交易使用一个通道。提供股票报价的接收者与实现交易的接收者完全不同，因此必须确保由正确的接收者消费正确的消息。发送者会把报价查询消息的选择值设置为 QUOTE，需要报价消息的选择性消费者只消费有此选择值的消息。交易消息的选择值为 TRADE，发送者和接收者都使用这个选择值。采用这种方式，两种类型的消息可以成功地共享一个通道。

例：JMS 消息选择器

在 JMS 中，创建 MessageConsumer（QueueReceiver 或 TopicSubscriber）时可以带一个消息选择器字符串，从而根据消息的属性值过滤消息[JMS1.1]、[Hapner]。首先，发送者的消息中设置属性值，接收者可以根据这个值过滤消息：

```
Session session = // 获得会话
TextMessage message = session.createTextMessage();
message.setText("<quote>SUNW</quote>");
message.setStringProperty("req_type", "quote");
Destination destination = // 获得目标
MessageProducer producer = session.createProducer(destination);
producer.send(message);
```

其次，接收者设置其消息选择器，按这个值进行过滤：

```
Session session = // 获得会话
Destination destination = // 获得目标
String selector = "req_type = 'quote'";
MessageConsumer consumer =
    session.createConsumer(destination, selector);
```

如果请求消息的类型属性不是 quote，接收者将忽略这些消息，就好像这些消息根本没有传送到目标一样。

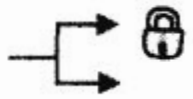
例：.NET Peek、ReceiveById 及 ReceiveByCorrelationId

在.NET 中，MessageQueue.Receive 实际上不支持 JMS 方式的消息选择器。接收者能

做的只是使用 MessageQueue.Peek 查看消息。如果消息满足期望的标准，则可以使用 MessageQueue.Receive 从队列中读取消息。由于 Receive 调用返回的消息可能与 Peek 查看的消息不同，采用这种方法不一定很可靠。因此，可以使用 ReceiveById（而不是使用 Receive），由消费者指定所要接收的消息的 ID 属性值，以确保获得的消息正是所查看的消息。

在.NET 中，还有一种做法是使用 ReceiveByCorrelationId 方法，消费者可以指定想要接收的消息的 CorrelationId 属性值。特定请求消息的发送者可以使用 ReceiveBy-CorrelationId 来接收与请求对应的应答消息（见“请求/应答”和“关联标识符”）。

10.10 持久订购者



应用要接收发布-订购通道中的消息。

如何防止订购者丢失其未监听期间发布的消息？

为什么会出现这样的问题？一旦把消息添加到通道中，它就一直驻留在那里，直到被消费，或过期（见“消息到期”），或者系统崩溃（除非使用可靠传输）。对点对点通道上的消息来说，情况就是这样的，但是对发布-订购通道来说，则有所不同。

当消息发布到发布-订购通道中时，消息传递系统必须把消息传送给每个订购者。如何实现这项功能与具体的实现有关：可以保留消息，直到未接收到消息的订购者列表为空；或者可以复制消息，并把副本传送给每个订购者。不管采用哪种方式，到底由哪些订购者接收消息，这完全由发布消息时谁订购了通道来决定。即使某个接收者在消息发布之后立即订购了通道，它也无法接收到这个消息。（如果订购者订购了一个通道，与此同时，有消息发布到这个通道上，这也是一个与时间有关的问题。那么订购者是否能接收到消息以及如何解决这个问题，要由消息传递系统的具体实现决定。为了安全起见，订购者应该确保在感兴趣的消息发布前先行订购通道。）

实际中，订购者通过关闭与通道的连接来取消对该通道的订购。因此，它不需要采取显式的取消订购动作。订购者只要关闭连接就可以了。

通常，应用会忽略断开连接后发布的消息，因为断开连接意味着应用已经对发布的任何消息不感兴趣了。例如，一个卖砖的 B2B 或 B2C 应用可能订购了一个通道，通过这个通道购买者能请求买砖。如果应用停止售砖，或者它的砖临时脱销，可以与通道断开连接，以避免接收到它无法满足的请求。

然而这种处理方法存在不足，因为这种“偷懒、放纵”的做法会造成应用错过本来需

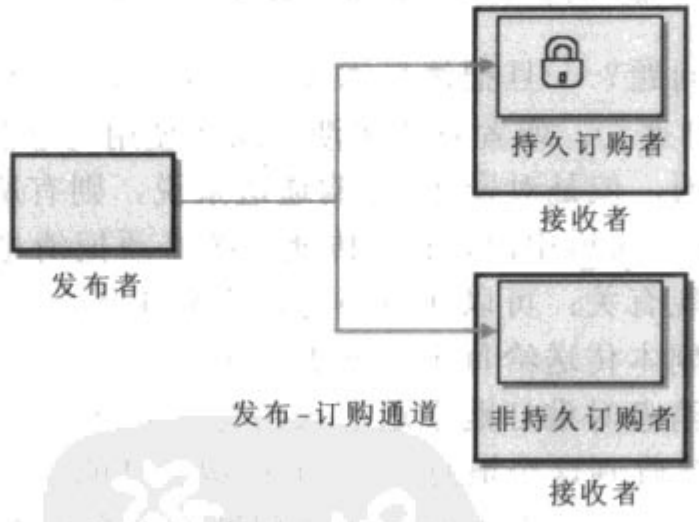
要的消息。如果应用崩溃或者必须停下来进行维护，它可能想知道在不运行期间错过了哪些消息。消息传递的根本思想就是：即使发送者应用、接收者应用以及网络并没有同时工作，也要保证可靠的通信。

因此，有时应用断开连接是因为它们不想再接收到通道中的任何消息。但是，也可能应用必须断开连接一段时间，当它们重新连接上通道时，想要获得在连接中断期间发布的所有消息。通常订购者要么是连接的（订购），要么是未连接的（未订购），但是也存在第三种状态：不活跃（inactive），这个状态下的订购者没有连接，但是仍然要订购，因为它想接收到未连接期间发布的消息。

如果订购者连接到一个发布-订购通道，但是当发布一个消息时，订购者未连接，那么消息传递系统怎样判断是否需要为这个订购者保存消息，以便订购者重新连接通道后能把这个消息再传送给它？也就是说，消息传递系统如何判断出一个未连接的订购者是不活跃还是未认购？这里需要有两种类型的订购：一种订购是订购者断开连接时就结束；另一种订购是当应用断开连接时仍然存在，除非应用明确地取消订购。

默认情况下，订购只在连接时保持，因此，我们需要的是另一种类型的订购，即使断开连接也能保持，此时成为不活跃的订购。

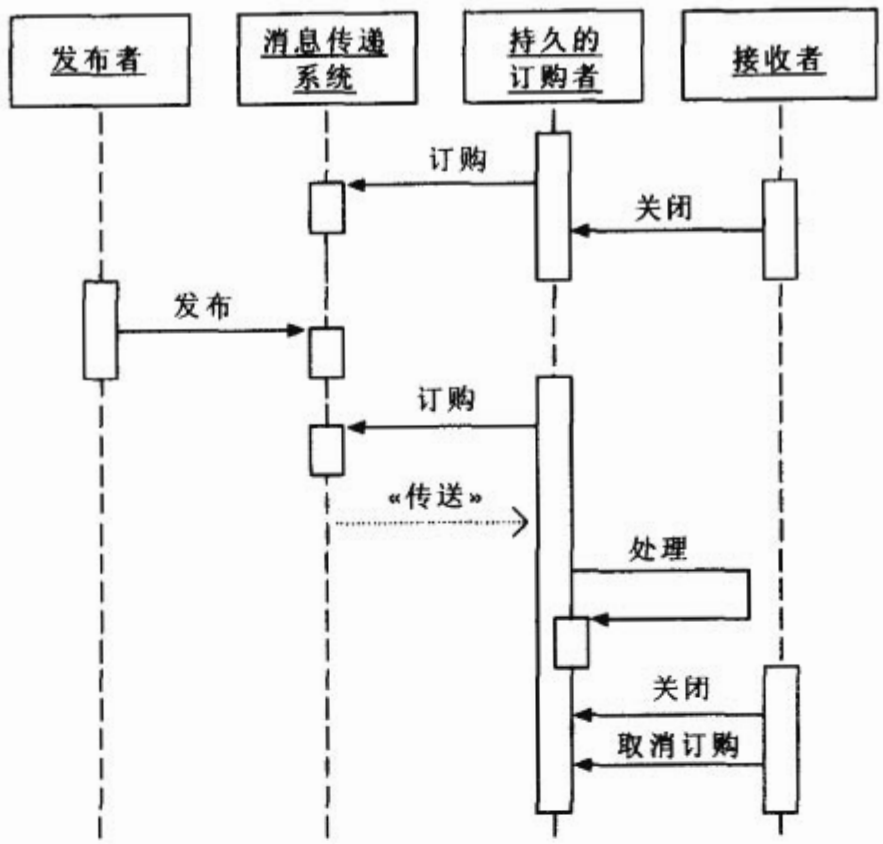
使用持久的订购者，让消息传递系统保存订购者断开连接期间发布的消息。



持久的订购会为不活跃的订购者保存消息，当订购者重新连接通道时，消息传递系统再把保存的这些消息传送给它。采用这种方式，即使订购者断开连接，它也不会丢失任何消息。当订购者处于活跃（例如连接）状态时，持久的订购不会对订购者或消息传递系统的行为造成任何影响。不论采用的是持久的还是非持久的订购，如果订购者与通道连接，其行为是相同的。二者的区别只是当订购者未连接时消息传递系统如何操作。

持久的订购者就是发布-订购通道上的一个订购者。但是，当订购者与消息传递系统断开连接时，它进入不活跃状态，消息传递系统将把所有发布的消息保存在订购者的通道中，直到订购者再次活跃。在这期间，其他订购相同通道的订购者可能不是持久的，这些订购

者则是非持久订购者。



持久订购的顺序图

作为订购者，持久订购者必须建立对通道的订购。一旦完成订购，当它关闭自己的连接时，就进入不活跃状态。当订购者不活跃时，发布者发布了一个消息。如果订购者是非持久的，它将错过这个消息，而作为持久订购者，消息传递系统就要为它保存消息。当订购者重新订购通道时，它再次成为活跃的订购者，消息传递系统将把先前排队的消息（以及其他为该订购者保存的消息）传送给它。订购者接收消息，并进行处理（也可以委托给应用处理）。一旦订购者处理完消息，并且不想再接收任何消息，它可以关闭其连接，再次进入不活跃的状态。如果它不想让消息传递系统为其保存消息，也可以取消订购。

考虑这样一种有趣的情况，如果持久的订购者始终不认购，会出现什么情况？不活跃的持久订购会不断地保留消息，也就是说，消息传递系统将保存所有已发布的消息，直到订购者重新连接。但是，如果订购者很长时间都没有重新连接，那么保存的消息就会很多。消息到期（Message Expiration）有助于缓解这个问题。消息传递系统也可能想限制为不活跃订购保存的消息数量。

例：股票交易

股票交易系统可以使用发布-订购通道把股票价格的变动广播出去。每当股票价格发生变动时，将发布一个消息。一个订购者可能是一个 GUI，可以显示某些股票的当前价格。另一个订购者可能是一个数据库，会保存某些股票的当天交易价格变化范围。

这两种应用都应该是价格变动通道的订购者，当股票价格发生变动时，它们会得到通知。GUI 应用的订购是非持久的，因为它只需显示股票的当前价格。如果它崩溃并断开与

通道的连接，没有必要保存未显示期间发生的价格变动。相反，价格范围数据库应该使用一个持久的订购者。当其运行时，可以显示以前所有变动的价格范围。如果它断开连接，那么在重新建立连接后，也可以处理发生过的价格变动，并更新范围数据库。

例：JMS 持久订购

JMS 支持 TopicSubscriber 的持久订购[JMS1.1]、[Hapner]。

采用持久订购面临的一个困难是，如何区分新的订购者和重新连接的老订购者。在 JMS 中，持久订购由以下标准衡量：

1. 认购的主题
2. 连接的客户端 ID
3. 订购者的订购名

连接的客户端 ID 是连接工厂的一个属性，当创建连接工厂时，由消息传递系统的管理工具设置这个属性。每个订购者的订购名必须是唯一的（对特定的主题和客户端 ID 来说）。

持久的订购者使用 Session.createDurableSubscriber 方法创建：

```
ConnectionFactory factory = // 获得工厂
//工厂有客户端ID
Connection connection = factory.createConnection();
//连接与工厂有相同的客户端ID
Topic topic =//获得主题
String clientID = connection.getClientID(); //如果你想知道
String subscriptionName = "subscriber1"; //针对订购的某个UID

Session session =
    connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
TopicSubscriber subscriber =
    session.createDurableSubscriber(topic, subscriptionName);
```

该订购者目前是活跃的。当消息发布到主题后，它就能接收到这些消息（与非持久订购者一样）。要把它变成不活跃的，可关闭其连接，如下所示：

```
subscriber.close();
```

订购者的连接已被关闭，因此成为不活跃的订购者。任何发布到其主题中的消息都保存起来，直到它重新连接时才会传送给它。

要使订购再次成为活跃的，必须采用相同的主题、客户端 ID 和订购名创建一个新的持久订购者。实现的代码和前面一样，只不过连接工厂、主题和订购名必须和前面一样。

由于建立持久订购和重新连接的代码是一样的，因此只有消息传递系统才清楚这是一个重新建立的持久订购，还是一个新的订购。有意思的是，重新连接到订购的应用可能与以前断开连接的应用不同。只要新应用使用了相同的主题、相同的连接工厂（和相同的客户端 ID）以及相同的订购名，消息传递系统就无法区分这两个应用，并且会把所有的消息

传送给新的应用，而不是断开连接前的老应用。

一旦应用拥有了某个主题的持久订购，它就能接收到发布到该主题的所有消息，即使订购者关闭了它的连接（或者因为崩溃而使消息传递系统关闭了订购者的连接），也不会丢失消息。为了使消息传递系统停止为这个不活跃的订购者缓存消息，应用必须明确地取消持久订购。

```
subscriber.close();
// 订购者现在是不活跃的，将保存消息
session.unsubscribe(subscriptionName);
// 订购被删除
```

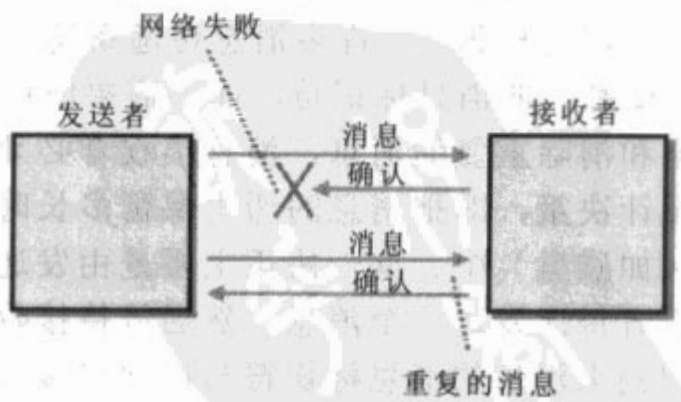
一旦订购者取消订购，则把这个订购从主题中删除，消息不再传送给这个订购者。

10.11 幂等接收者

即使发送者应用只发送了一次消息，接收者应用也可能会多次接收到相同的消息。

消息接收者如何处理重复的消息？

第 3 章“消息传递系统”中的通道模式讨论了如何使用可靠传输使消息传递通道更加可靠。但是，即使是一些可靠的消息传递实现也会产生重复的消息。在某些情况下，由于通信依赖于根本不可靠的协议，因此即使采用可靠传输，通信也是不可靠的。在许多 B2B 的集成应用中，消息必须采用 HTTP 协议通过 Internet 发送。在这些情况下，通常只能通过消息重传，直到获得接收方返回的确认消息，以此来保证消息传送。但是，如果因为不可靠的连接使确认消息丢失，发送者就会重传接收者已经接收到的消息（如下图所示）。



由于发送确认方面的问题造成消息的重复

许多消息传递系统在内部建立了消除重复消息的机制，因此应用不必担心重复问题。但是，在消息传递基础设施内消除重复会增加系统的负担。如果接收者能容忍重复的消息（如处理查询式命令消息的无状态接收者），允许消息重复就能提高消息传递的吞吐量。因

此，有些消息传递系统只提供了至少一次的传送机制，让应用自己解决消息重复问题。其他消息传递系统允许应用指定是否自己处理重复消息（例如，JMS 规范定义了 DUPS_OK_ACKNOWLEDGE 模式）。

还有一种情况可能产生重复消息，这就是分布式事务失败。许多封装好的应用通过商业适配器连接到消息传递基础设施，它们无法正确地参与分布式两段提交。如果一个消息发送给多个应用，并且这个消息引起一个或多个应用失败，集成系统则很难从这种不一致的状态中恢复。如果接收者设计为能忽略重复的消息，那么发送者只需把消息重新发送给所有的接收者即可。对于那些已经接收到消息并完成了原消息处理的接收者来说，它们只需忽略重传的消息。对于那些未能正确消费原消息的应用，则可以使用新近重新发送的消息。

> 把接收者设计成幂等接收者（Idempotent Receiver），它能安全地多次接收相同的消息。

“幂等（Idempotent）”这个术语在数学中是指，一个函数应用于自身，可以产生相同的结果，即 f(x)=f(f(x))。在消息传递中，这一概念可以解释为：不管消息接收一次还是多次，产生的效果都相同。也就是说，消息可以多次重传，而不会引起任何问题，即使接收者会重复接收到相同的消息。

幂等性可以通过两种主要方法获得：

1. 显式的去重复（de-duping），即把重复的消息删除。
2. 适当地定义消息的语义，使之支持幂等性。

接收者通过跟踪已经接收到的消息，可以显式地去掉重复的消息。采用唯一的消息标识符可以简化这个任务，并有助于检测出什么情况下会到达消息内容相同的两个合法消息。通过使用单独的消息标识符字段，就不会把重复消息的语义与消息的内容联系在一起。可以为每个消息分配一个唯一的消息标识符。许多消息传递系统，如 JMS 兼容的消息传递工具，就能为每个消息自动分配唯一的消息标识符，而不需要应用操心。

要根据消息标识符检测和消除重复的消息，消息接收者必须记录已经接收到的消息标识符。对此有一个关键的设计决策，即把消息的历史保留多长时间，以及是否把消息的历史持久记录到永久存储器（如磁盘）中。这一决策主要是由发送者与接收者协商确定的。最简单的一种情况是，发送者每次发送一个消息，然后等待接收者返回每个消息的确认。采用这种方法，接收者能把到来消息的消息标识符与以前已接收消息的标识符进行比较，这就足够了。如果标识符相同，则忽略新的消息。实际上，接收者保留了一个消息的历史。在实际应用中，采用这种通信方式的效率很低，特别是当延迟（即消息从发送者到达接收者的时间）相对于期望消息吞吐量所占比例很大时，这种做法更显低效。在这些情况下，发送者可能想发送整个一组消息，而不必等待每个消息的确认。但是，如果采用这种方式，接收者必须为已经接收到的消息保留一个更长的标识符记录。接收者“内存”的大小取决

于发送者能发送多少消息而不需要接收者返回确认，这类似于“重排器”中提出的问题。

通过进一步分析底层 TCP/IP 协议，往往能学到很多东西，关于如何消除重复消息也可以从 TCP/IP 协议有所收获。当 IP 网络数据包通过网络路由时，会产生重复的数据包。TCP/IP 协议为每个包附加一个唯一的标识符，从而消除重复的数据包。接收者为检测重复包要分配一个“窗口大小”，这是由发送者和接收者协商确定的。有关 TCP/IP 如何实现这一机制的详细讨论可参见[Stevens]。

在某些情况下，可以使用业务键作为消息标识符，并让持久存储层消除重复。例如，假设应用会把到来的订单持久地保存到数据库中。如果每个订单消息包含了唯一的订单号，并且我们在数据库中设置订单号字段作为唯一的键，那么如果接收到重复的订单消息，数据库的插入操作就会失败。这种解决方案看起来不错，因为我们把重复性检查委托给数据库系统完成，它能有效地检查出重复的键。但是，由于要对同一个字段关联双重语义，采用这种做法必须要小心，尤其是把基础设施相关的语义（重复的消息）与一个业务字段（订单号）相关联时更要当心。假设业务需求发生了变化，客户可能要发送另外的有相同订单号的消息来补充现有的订单（这种情况经常发生），由于我们把唯一消息标识符绑定为一个业务字段，因此必须修改消息的结构。所以最好不要让一个消息字段有双重的语义。

如果要使用数据库强制消除重复，有时会用消息传递基础设施开发商提供的数据库适配器来完成。在很多情况下，这些适配器无法消除重复，因此必须把这一功能委托给数据库完成。

另一种获得幂等性的方法是合理地定义消息的语义，使消息的重传不会影响系统。例如，不要定义为“为账户 12345 增加$10”，而是可以把消息改为“把帐户 12345 的结余设置为$110”。如果账户当前的结余为$100，使用这两个消息都能获得相同的结果。第二个消息是幂等的，因为重复接收到这样的消息不会有任何影响。诚然，这个例子没有考虑并发的情况，例如，在原始的消息和重复的消息之间有另一个消息到达：“把帐户 12345 的结余设置为$150”。这样一来，如果消息重复，最后的结余就是$110，而如果没有重复消息，最后的结余则是$150。

例：Microsoft IDL（MIDL）

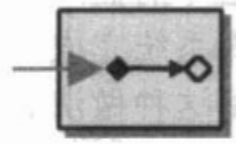
Microsoft 接口定义语言（Microsoft Interface Definition Language，MIDL）支持幂等性，并把这作为远程调用语义的一部分。使用[idempotent]属性可以把远程过程声明为幂等的。MIDL 规范规定：“[idempotent]属性可以指定一个操作不会修改状态信息，而且每次完成时都返回相同的结果。执行多次例程与只执行一次有相同的效果。”

```
interface IFoo;
[
    uuid(5767B67C-3F02-40ba-8B85-D8516F20A83B),
    pointer_default(unique)
]

interface IFoo
```

```
{
    [idempotent]
    bool GetCustomerName
    (
        [in] int CustomerID,
        [out] char *Name
    );
}
```

10.12 服务激励器



应用要向其他应用提供一个服务。

应用应该如何设计这个服务，从而既能通过各种消息传递技术调用，也能通过非消息传递技术调用？

应用不想把一个服务（这是服务层中的一个操作[EAA]）实现为只能同步调用或只能异步调用：它可能想为一个服务同时提供两种调用支持。然而，现有的技术使它必须做出选择。例如，使用 EJB 实现的应用需要采用会话 bean 支持同步的客户端，而采用 MDB 支持消息传递客户端[①]。

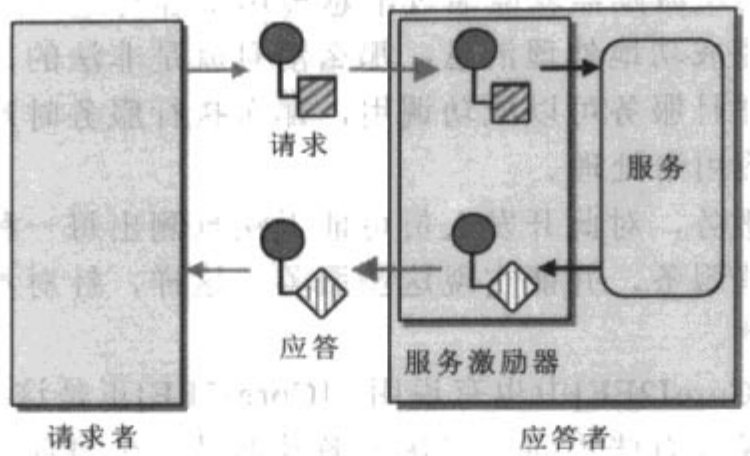
开发人员设计应用来与其他应用（如 B2B 应用）协同工作时，可能并不清楚要与哪些应用通信，以及各种通信如何完成。为了应付可能的需要，有太多不同的消息传递技术和数据格式需要提供支持[②]。

接收和处理一个消息涉及很多步骤，严格区分这些步骤很难，也非常复杂。不过，如果消息端点代码把多个任务混在一起——包括接收消息、提取消息的内容，以及根据这些内容完成工作——则很难重用。

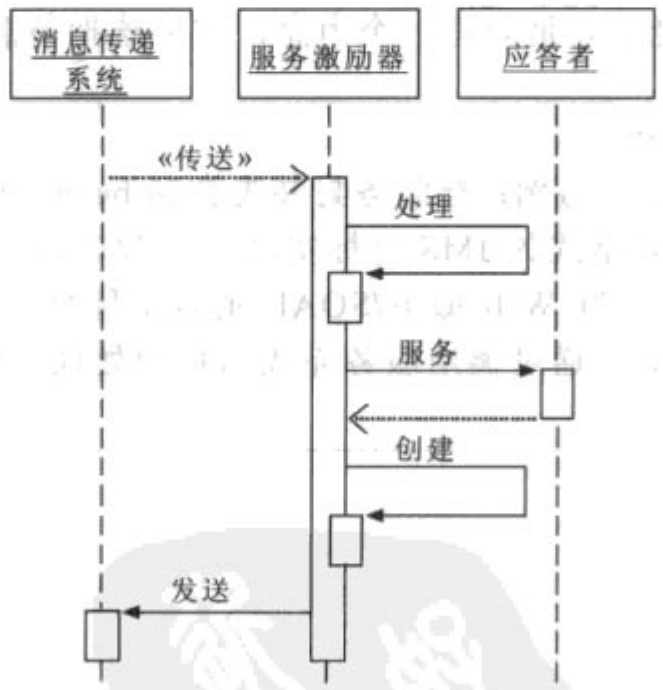
当针对多种通信方式设计客户端时，看起来很有必要为每种通信方式设计不同的服务。这不但不利于增加新的通信方式，还不能保证每种通信方式有相同的表现。我们需要一种方法，能由一个服务支持多种通信方式。

设计一个服务激励器，将通道中的消息与所访问的服务相连接。

① 感谢 Mark Weitzel 提供这个例子。
② 感谢 Luke Hohmann 提供这个例子。



服务激励器既可以是单向的（只有请求），也可以是双向的（请求/应答）。服务可能非常简单，只是一个方法调用（同步而且非远程），可能是服务层的一部分[EAA]。激励器既可以硬编码，从而始终调用同一个服务，也可以使用反射调用由消息指定的服务。激励器处理所有的消息传递细节，并像其他客户端一样调用服务，因此服务甚至不知道对它的调用是通过消息传递进行的。



请求/应答的服务激励器的顺序图

服务激励器负责接收请求消息（要么作为轮询消费者，要么作为事件驱动的消费者）。它知道消息的格式，并处理消息，从中抽取出必需的信息，以确定要调用什么服务以及要传递什么参数。然后它就像服务的其他客户端一样调用服务，并且在服务执行时阻塞。当服务完成并返回值时，激励器可以选择创建一个包含返回值的应答消息，然后把消息返回给请求者（应答消息就使服务调用成为一种请求/应答消息传递）。

利用服务激励器，可以把服务编写为就像始终会同步进行调用一样。激励器接收异步的消息，确定要调用的服务以及要传递的数据，然后以同步方式调用服务。设计的服务不

必考虑消息传递，不过利用激励器就能通过消息传递很容易地调用这个服务。

如果服务激励器不能成功地处理消息，那么消息就是非法的，应该转移到非法消息通道。如果消息能处理，并且服务可以成功调用，那么执行服务时产生的任何错误就是应用中的语义错误，应该由应用来处理。

合作方会如何访问服务，对此开发人员可能无法预测出每一种访问方式，但是他们至少知道应用应该提供哪些服务，并能实现这些服务。这样，针对不同的技术和格式实现新的激励器时就会相当简单。

服务激励器模式在[CoreJ2EE]中也有说明，[CoreJ2EE]正是这个模式名的出处。其中描述的模式与我们所说的模式有些不同，它认为激励器是一个事件驱动的消费者，并且服务已经存在，因此激励器可以添加到服务中。不过，这两种激励器模式都是针对相同的问题采用非常相似的方式提供了相同的解决方案。服务激励器与半同步/半异步模式[POSA2]有关，该模式把服务处理划分为同步层和异步层。

服务激励器一般接收的是命令消息，命令消息指定了要调用的服务。服务激励器可以作为消息传递网关，把消息的细节与服务分离。激励器可以是轮询消费者，也可以是事件驱动的消费者。如果服务是事务性的，激励器也应该是事务性客户端，从而使消息的消费和服务的调用在同一个事务中。多个激励器可以成为竞争的消费者，或者通过消息分派器协作。如果服务激励器不能成功地处理一个消息，它应该把消息发送给非法消息通道。

例：J2EE 企业 JavaBean

以 J2EE 中的 EJB[EJB2.0]为例：把服务封装为会话 bean，然后为各种类型的消息传递实现 MDB：为使用一种消息格式的 JMS 目标实现一个 MDB，再为使用另一种消息格式的不同目标实现一个 MDB，而为 Web 服务/SOAP 消息实现另一个 MDB，以此类推。每个 MDB 都是一个服务激励器，它通过调用服务完成消息的处理。希望同步调用服务的客户端可以直接访问会话 bean。

第 11 章

系统管理

11.1 引言

尽管开发一个消息传递解决方案不是一个轻松的任务，在生产环境中操作这样的解决方案同样是一个挑战。基于消息的集成解决方案在一天内可能会产生、路由和转换成千、甚至上百万的消息。我们必须处理异常、性能瓶颈以及参与系统中出现的改变。更复杂的是，这些组件可能分布在多个平台和计算机上，并位于不同的地方。

要集成分布式、已经封装好的和自定义的应用，除了固有的复杂性和规模外，松耦合的体系结构优点也使系统的测试和调试变得更加困难。Martin Fowler 把这称为“架构师的梦想，开发人员的噩梦”：松耦合和间接性的结构设计原则使系统之间可以减少对彼此的假设，为应用集成提供了灵活性。但是，如果消息的生产者不知道消息的消费者是谁，测试这样的系统就会面临极大的挑战。再加上消息传递的异步特性和时间关联性，会使事情变得更加复杂。例如，消息传递解决方案甚至不能设计为让消息生产者接收来自接收者的应答消息。同样地，消息传递基础设施一般能保证消息的传送，但是不能保证传送的时间，因此很难开发出依赖于消息传送结果的测试用例。

监控消息解决方案时，我们可以从两个不同的抽象层次跟踪消息流。典型的系统管理解决方案会监控发送了多少消息，或者要花多少时间处理一个消息。除了消息首部中的某些字段，如消息标识符或消息历史外，这些监控解决方案并不检查消息的数据。相反，业务活动监控（Business Activity Monitoring，BAM）解决方案会监控包含在消息中的有效数据，例如，最近一小时内所有订单的金额。本节介绍的许多模式都是非常通用的，可同时用于这两种监控。但是，由于 BAM 本身是一个非常新的技术领域，它在很多方面与数据仓库一样复杂（有些技术我们根本就没有接触过），因此我们决定只在系统管理上下文中讨论模式。

设计系统管理模式的目的是为了满足这些需求，并为保证复杂的基于消息的系统正常运行提供管理工具。本章介绍的模式可分为三类：监视和控制、观察和分析消息的通信量，以及测试和调试。

11.1.1 监视和控制

控制总线为管理和监视分布式解决方案提供了单一的控制点。它把多个组件连接到一个集中的管理控制台，通过它显示每个组件的状态，并监视通过这些组件的消息通信量。

也可以使用控制台向组件发送控制命令，例如，改变消息流。

我们可能想让消息经过额外的处理步骤，如验证或日志记录。由于这些步骤会产生性能开销，要能通过控制总线对它们实现开关切换。旁路（Detour）模式提供了这种功能。

11.1.2 观察和分析消息通信量

有时，我们想检查消息的内容，却又不想影响主要的消息流。采用线路分接器（Wire Tap）模式使我们能接入消息流。

在调试基于消息的系统时，如果知道特定的消息在哪里将会给我们提供很大的帮助。消息历史模式记录了消息访问过的所有组件，同时还不会增加组件之间的依赖性。

消息历史只对应单个的消息，集中的消息存储库可以为经过系统的每个消息提供完整的记录。如果把它与消息历史结合起来，就能分析出消息经过系统的所有可能的路径。

线路分接器、消息历史和消息存储库可以帮助我们分析消息的异步流。为了跟踪发送给请求/应答服务的消息，需要在消息流中插入一个智能代理（Smart Proxy）。

11.1.3 测试和调试

把消息传递系统部署到产品环境中以前，首先要进行测试，这是很好的建议。但是测试并不只是在这个阶段进行。你应该主动检查运行中的消息传递系统，看它是否能提供正确的功能。通过周期性地向系统中插入测试消息（Test Message）并检查其结果，可以实现这项功能。

当组件失败或操作异常时，很容易在通道中产生不需要的消息。在测试期间，如果能把通道中剩余的消息全部删除，使所测试的组件不会接收到“残余”的消息，这是非常重要的。通道清洁者（Channel Purger）可为我们完成这项功能。

11.2 控制总线



企业集成系统一般是分布式的。事实上，衡量企业消息传递质量的一项指标就是实现分散系统之间的通信。消息传递系统要路由和转换消息，使数据能在这些系统之间交换。在大多数情况下，这些应用分布在不同的网络、建筑物、城市或大洲。

如何有效地管理分布在多种平台和很大地理范围内的消息传递系统？

分布式、松耦合的体系结构增加了灵活性和可扩展性。同时，它也为管理和控制这样的系统带来了严峻的挑战。例如，怎样才能知道是否所有的组件都已经启动并且正在运行？采用简单的进程状态是不够的，因为这些进程分布在许多计算机上。而且，如果你无法获

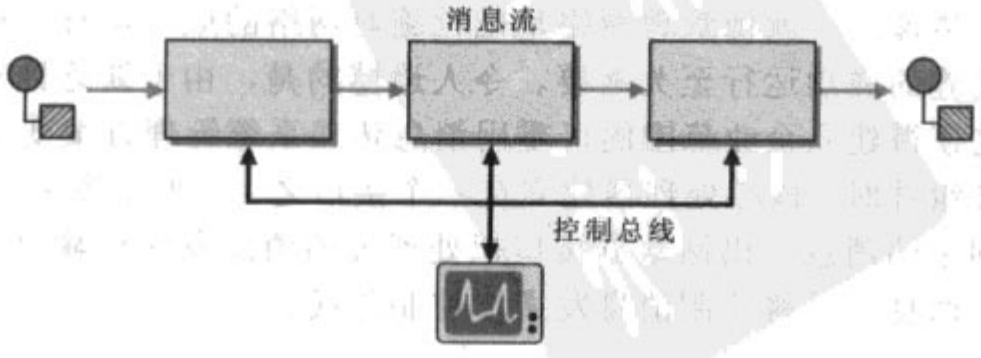
得远程计算机的状态，就不能断定这是由于远程计算机没有工作造成的，还是因为与远程计算机的通信受到干扰造成的？

除了要知道系统或组件是否已经启动并在运行之外，还要监视系统的动态行为。消息的吞吐量是多少？是否存在不正常的延迟？通道是否已满？为了获得其中一些信息，需要跟踪组件之间传递的消息数或通过组件传递的消息数。这就要求从多台计算机上搜集和组合信息。

另外，只是从组件获得信息还不够。当系统正在运行时，还经常需要调整或修改配置信息。例如，当系统运行时，可能要打开或关闭日志记录特性。许多应用使用属性文件和错误日志来读取配置信息和报告错误情况。如果应用是由单台计算机或少量的计算机组成的，采用这种方法很有效。对于大型的、分布式解决方案来说，必须使用文件传输机制把属性文件复制到远程计算机上，这就要求每台计算机上的文件系统能远程访问。如果计算机是通过 Internet 连接或通过不支持文件映象协议的广域网连接，那么采用这种方法就会增加安全风险和实现的难度。而且，本地属性文件的版本必须仔细管理，这就给管理带来很大的负担。

很自然的，我们可以借助消息传递基础设施完成部分管理任务。例如，可以向一个组件发送修改其配置的消息。该控制消息可以像一般的消息那样得到路由和传输，采用这种方法能解决大多数通信问题，但是也带来了新的难题。与普通的应用消息相比，配置消息要采用更严格的安全策略。例如，一个格式错误的控制消息很容易使组件关闭。另外，假如组件出现故障，导致消息在消息通道中排队等待该怎么办？如果我们要复位组件而发送了一条控制消息，这个控制消息也会与其他消息一样在通道中排队等待，无法到达出故障的组件。有的消息传递系统能够提供一种消息特性，使控制消息能转移到队头。但是，并不是所有的系统都提供了这种功能，如果队列已满，无法接收更多的消息，那么即使有这种优先级也无济于事。另外，有些控制消息的优先级比应用消息的优先级还低。如果让组件周期性地发布状态消息，“I am alive（我活着）”之类的控制消息如果延迟或丢失，应该比延迟或丢失“Order for $1 million（一百万美元的订单）”之类的应用消息所造成的麻烦少得多。

使用控制总线管理企业集成系统。控制总线和应用数据使用相同的消息传递机制，但是对于参与消息流的组件，使用了不同的通道传送与管理这些组件有关的数据。



系统中的每个组件现在连接到两个消息传递子系统中：

1. 应用消息流。

2. 控制总线。

应用消息流负责传输所有与应用相关的消息。组件订购这些通道，并向这些通道发布消息，就像未经管理一样。同时，每个组件还从组成控制总线的通道那里接收消息或向这些通道发布消息。这些通道与一个集中的管理组件连接。

控制总线很适合携带以下类型的消息：

1. **配置消息**——参与消息流的每个组件都有能根据需要修改的可配置参数。这些参数包括通道的地址、消息数据格式、超时设定等。组件使用控制总线而非属性文件从集中的知识库那里获取配置信息，从而在运行时对集成解决方案进行集中配置和重新配置。例如，基于内容的路由器中的路由表必须根据系统的状况进行动态更新，如超负荷或组件失败。

2. **心跳消息**——每个组件可以按照指定的时间间隔向控制总线发送周期性的心跳消息，从而使集中的控制台应用能检查组件的功能是否正常。该心跳消息还包含了组件的度量信息，如所处理的消息数和计算机上可用的内存空间等。

3. **测试消息**——心跳消息能告诉控制总线某个组件是否活跃，但是它们只能提供有限的信息，无法全面地反映组件正确处理消息的能力。除了让组件周期性地向控制总线发布心跳消息外，我们还可以在被组件处理的消息流中插入测试消息。通过事后取回的测试消息可以检查组件是否进行了正确的处理。由于这种方法混杂了控制总线和应用消息流的定义，所以我们为它单独定义了一个模式（见“测试消息”）。

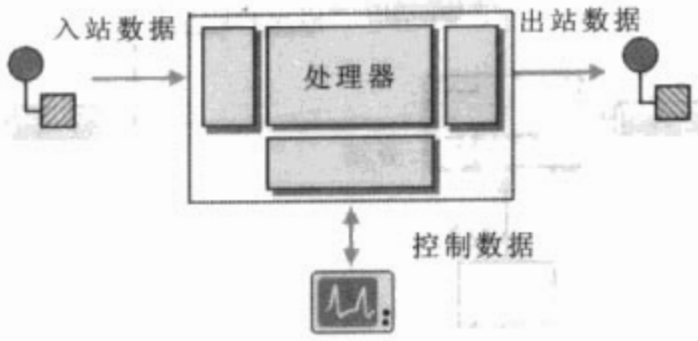
4. **异常**——每个组件都可以把异常情况报告给控制总线进行评估。严重异常将向操作员发出警告。定义异常处理的规则应该在一个集中的处理程序中设置。

5. **统计信息**——每个组件可以收集有关所处理的消息数、平均吞吐量、处理消息的平均时间等统计数据。其中有些数据可以按消息类型进行划分，这样我们就能查看整个系统中是否充斥着某种类型的消息。由于这种消息的优先级要比其他消息低，控制总线很可能使用不可靠的通道或低优先级的通道传递这些数据。

6. **实时控制台**——前面提到的大多数功能均可汇集在一个集中的控制台上显示。操作人员通过它能评估消息传递系统是否正常，并能根据需要采取相应的矫正措施。

控制总线提供的许多功能类似于用于监控和维护网络解决方案的传统网络管理功能。利用控制总线，我们能在消息传递系统级完成相同的管理功能，从而由低级的 IP 网络级提升到更丰富的消息传递级。就像管理网络基础设施对网络的成功运行一样，提供管理功能对消息传递基础设施的成功运行至关重要。令人遗憾的是，由于缺乏针对消息传递解决方案的管理标准，这使得建立企业范围的可重用消息传递系统管理方案变得非常困难。

设计消息处理组件时，核心处理器建立在三个接口之上（如下图所示）。入站数据接口从消息通道接收到来的消息。出站数据接口把处理过的消息发送给输出通道。控制接口从控制总线接收控制消息，并将控制消息发送给控制总线。

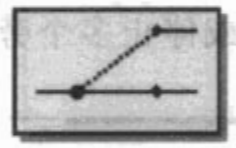


消息传递组件的关键接口

例：调节贷款中介示例

在第 12 章“中场演练：系统管理示例”中，我们将介绍如何使用控制总线调整第 9 章“中场演练：复合消息传递”中介绍的贷款中介示例。其中包括一个简单的管理控制台，可以实时显示组件的状态（见第 12 章的“贷款中介系统的管理”）。

11.3 旁路



有时我们想根据外部条件修改消息的路由。

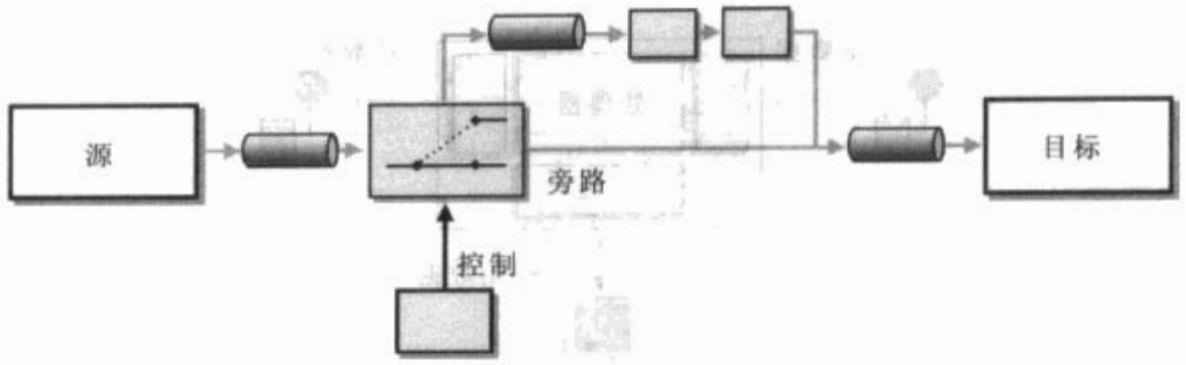
如何才能使消息经过中间处理步骤，完成验证、测试或调试功能？

能够对组件之间传输的消息进行验证，这是一个非常有用的调试工具。但是，这些额外的处理步骤并非一直都需要，而且如果始终执行这些步骤会降低系统的处理速度。

如果能按照一个集中的设置来包含或跳过这些处理步骤，这就能成为一种非常高效的调试或性能调优工具。例如，当测试系统时，可以让消息经过这些附加的验证步骤。而在生产阶段，为了提高性能，可以旁路这些步骤。这些验证步骤就像在源代码中插入的 assert 语句一样，它们只在调试版本中执行，在发行版本中不会执行。

同样地，在故障测试时，为了记录或监视，可以让消息经过这些附加的处理步骤。通过使用或停止这些记录步骤，能让正常工作模式下的消息吞吐量最大化。

采用基于内容的路由器构造旁路（Detour）组件，由控制总线控制。在一种状态下，路由器会让到来的消息经过附加的处理步骤；而在另一种状态下，则直接把消息路由给目标通道。

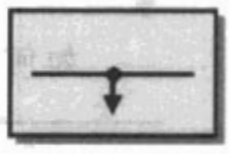


旁路组件使用了一个简单的基于内容的路由器，它有两个输出通道。其中一个输出通道把不做任何修改的消息传递给原目标。当控制总线发出指令时，旁路组件把消息路由到另一个通道。该通道把消息发送给附加的组件，这些组件可以检查和/或修改消息。最终，这些组件再把消息路由到同一个目标。

如果旁路的线路中只包含一个组件，那么把旁路开关与组件结合为一个过滤器可以获得更高的效率。但是，采用这种方法的前提条件是，旁路线路中的组件可以修改并能包含由控制总线控制的旁路逻辑。

采用控制总线控制旁路组件的好处是，通过控制总线中的一个控制命令就能同时激活或停止多个旁路组件，通用发布-订购通道可以把控制台与所有的旁路组件连接起来。

11.4 线路分接器



点对点通道经常用于传递文档消息，这是因为它们恰好能保证每个消息只被一个消费者消费。但是，为了测试、监视或调试故障，往往需要审查通过这些通道的所有消息。

如何检查点对点通道中传输的消息？

查看通道中传输的消息非常有必要。例如，可以用于简单的调试目的，也可以用于把消息存储到消息存储库（Message Store）中。不可能为点对点通道再添加一个监听者，因为它将从通道中消费掉消息，使预期的接收者无法消费消息。

当然，可以让发送者或接收者负责把消息发布到另一个不同的检查通道中。但是，采用这种方法可能要修改大量组件。而且，如果我们集成的组件是已经封装好的应用，就不可能对它们进行修改。

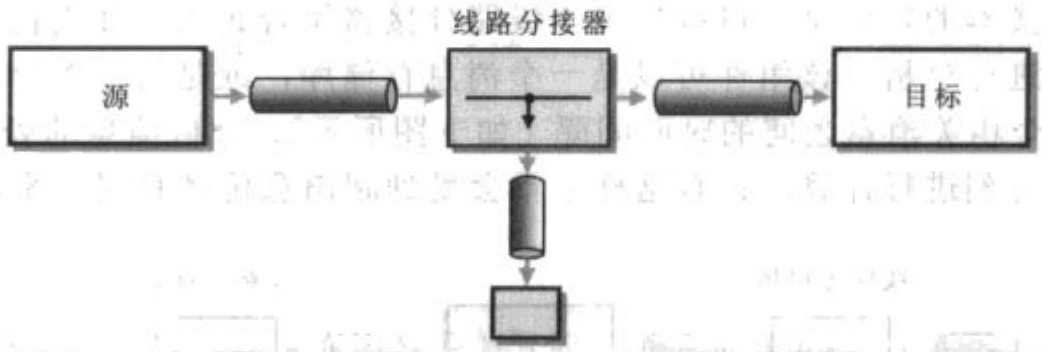
也可以考虑把通道修改为发布-订购通道，使额外的监听者能够在不干扰消息流的情况

下检查消息。但是，发布-订购通道改变了通道的语义。例如，假设有多个竞争的消费者从通道中消费消息，其中只有一个消费者可以接收到指定的消息。如果把通道修改为发布-订购通道，就会使每个消费者都能接收到各个消息。如果到来的消息表示一个订单，现在该订单就会被处理多次。即使只有一个消费者监听通道，与点对点通道相比，使用发布-订购通道的效率和可靠性也更低。

许多消息传递系统提供了查看（Peek）方法，使组件能检查点对点通道中的消息，而不用消费它。但是这种方法有一个很大的限制：一旦预期的消费者消费了消息，查看方法就无法再看到这个消息。因此，使用这种方法无法分析已经被消费的消息。

可以在通道中插入一个组件（一种“拦截器”），用它执行任何必需的检查。该组件从进入通道中消费消息，检查消息，然后把消息未经修改地传递给输出通道。但是，完成何种检查，这往往要根据来自多个通道的消息决定（如测量消息的处理时间），因此这项功能不可能由单个通道中的单个过滤器实现。

在通道中插入线路分接器，这是一个简单的接收表，能够把到来的各个消息同时发布到主通道和副通道中。



线路分接器（也被称为“T 型头”）是一个固定的接收表，有两个输出通道。它从输入通道中消费消息，并把消息不加修改地发布到这两个输出通道中。为了把线路分接器插入到通道中，必须创建一个附加的通道，并修改目标接收者，使它消费第二个通道上的消息。由于分析是由单独的组件完成的，因此可以在任意的通道中插入通用的线路分接器，而不必担心它会无意地改变主通道的行为。采用这种方法可以提高重用性，并降低调整已有解决方案可能带来的副作用。

把线路分接器设计成可编程的，从而使控制总线能控制副通道（“分接器”）的开与关，这很有用。采用这种方式，能够控制线路分接器，使它只在测试或调试时把消息发布到副通道。

线路分接器的主要缺点是，消费和重新发布消息会产生额外的延迟。许多集成工具套件会自动地解码消息，即使这些消息将不加修改地发布到另外的通道。而且，新消息将获得与原消息不同的新消息 ID 和新时间戳。这些操作会增加额外的处理开销，并破坏现有的处理机制。例如，如果最初的消息流使用消息 ID 作为关联标识符，那么采用这种方法

将破坏现有的解决方案，因为重新发布的消息的消息 ID 与原消息的消息 ID 不同。正是因为这个原因，所以说使用消息 ID 作为关联标识符不好。

由于线路分接器发布了两个单独的消息，因此千万不要通过消息 ID 把这些消息关联起来。即使主通道和副通道接收的是相同的消息，大多数消息传递系统也会自动为系统中的每个消息分配新的消息 ID。也就是说，原始消息与“复制”的消息有不同的消息 ID。

已有的消息代理（Message Broker）很容易扩充为线路分接器，因为所有的消息都要经过这个集中的组件。

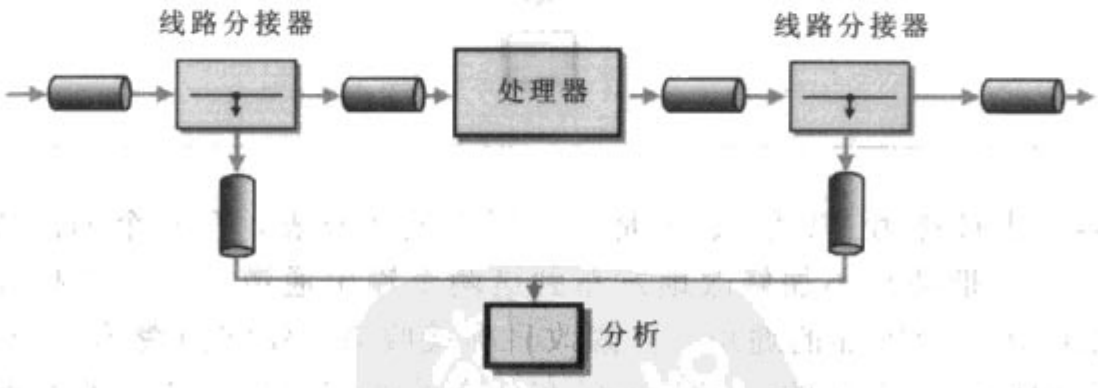
线路分接器的一个重要限制是，它不能修改流经通道的消息。如果需要操纵消息，可以使用旁路组件来代替。

例：贷款中介

在第 12 章“中场演练：系统管理实例”中，我们扩展了贷款中介示例，在连接信用机构的请求通道中增加了线路分接器，从而记录向这个外部服务发出的所有请求消息（见第 12 章中的“贷款中介系统管理”）。

例：使用多个线路分接器度量消息的处理时间

使用线路分接器的好处是，可以把多个线路分接器组合起来，让它们把消息的副本发送给集中的组件进行分析。该组件可以是一个消息存储库，或是另一个分析消息间关系的组件，如分析两个相关消息之间的时间间隔（如下图所示）。分析消息的处理时间时，要根据发送副消息的时刻进行计算，只有这样才不会受到副消息传播时间的影响。



使用一对线路分接器分析消息的处理时间

11.5 消息历史

能够实现参与者之间的松耦合，这是基于消息的系统的一个显著优点。消息的发送者和接收者不用对彼此的身份做任何假设（或只是做很少的假设）。如果消息的接收者从消息通道中取得一个消息，它一般并不知道也不必关心是由哪个应用把这个消息发送到通道中。消息本身是自包含的，它与任何特定的发送者无关。这是基于消息的系统所拥有的一项结构优势。

但是，这种特性也使得调试和分析依赖性变得相当困难。如果我们无法确定要把消息发送到哪里，又怎能评价消息格式的变化可能造成的影响呢？同样的，如果我们不清楚某个消息是由哪个应用发送的，就很难更正消息中的错误。

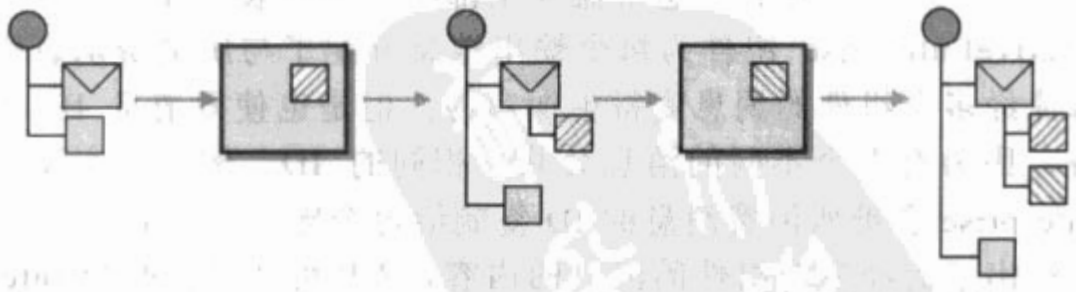
如何在一个松耦合的系统中有效地调试和分析消息流？

控制总线监视每个处理消息的组件的状态，但是它并不关心每个消息采取的路由。可以修改各个组件，从而为经过组件并发送给控制总线的每个消息都设置唯一的消息标识符，然后把这一信息收集在一个公共数据库，即消息存储库（Message Store）中。采用这种方法需要大量的基础设施支持，包括一个单独的数据库。而且，如果组件需要检查消息的历史，它必须对集中的数据库进行查询，这就增加了数据库成为性能瓶颈的风险。

跟踪通过一个系统的消息流并不像想像的那么简单。看起来似乎可以使用每个消息中唯一的消息 ID。但是，当组件（如消息路由器）处理一个消息并把它发布到输出通道时，处理后的消息将使用新的消息标识符，该标识符与组件所消费消息的标识符无关。因此，我们必须检查一个新的键，该键要从到来的消息复制到输出的消息中，才能使这两个消息关联起来。如果组件为所消费的每条消息只发布一个消息，那么采用这种方法是可行的。但是，许多组件如接收表、聚合器或过程管理器并不是这样的，它们一般要对单条输入消息发布多个消息。

与其采用标记消息的方法来识别每个消息的路径，还不如让消息自己收集所经组件的信息。如果消息传递系统中的每个组件都有唯一的标识符，那么每个组件可以将自己的标识符添加到它所发布的各个消息中。

将消息历史附加到消息中。消息历史是消息从开始所经过的所有应用或组件的列表。



消息历史中保存着消息所经全部组件的列表。每个处理消息的组件（包括最初的组件）都要在这个列表中添加一项。因为消息历史包含了系统相关的控制信息，因此它应该包含在消息首部中。把该信息保存在消息首部中，使之与包含应用相关数据的消息体区分开来。

组件发布的各个消息并不都是来自于单个消息。例如，聚合器发布的一个消息就包含了从多个消息收集到的信息，其中每个消息均包含自己的消息历史。如果我们想在消息历

史中表述这些信息，有两种可选的方法。如果要跟踪完整的历史，可以扩展消息历史，使其按层次树型结构存储信息。由于树型结构固有的递归特性，因此可以在单个结点下存储多个消息的历史。另一种做法是，可以继续采用简单的列表，并且只保存到来消息的历史。如果对于结果来说，一个到来的消息比其他辅助消息更重要，那么采用这种方法就足够了。例如，在拍卖系统中可选择只传播“竞标成功”消息的历史。

如果一系列消息要流经多个不同的过滤器才能实现一个特殊的业务功能或处理步骤，此时采用消息历史最为有用。如果管理消息经过的路径很重要，可以采用过程管理器。过程管理器为每个到来的触发消息创建一个过程实例。这样流经各个组件的消息会得到集中的管理，从而不太需要在每个消息中记录历史。

例：避免无限的循环

在每个消息中存储消息的历史，这对使用发布-订购通道传播事件来说非常有用。假设要实现一个系统，通过发布-订购通道把地址修改信息传播给多个系统。每个地址修改消息广播给所有感兴趣的系统，使这些系统能更新自己的记录。这种方法对于增加新的系统非常灵活，新系统会自动收到广播的消息，不必对已有的消息传递系统做任何修改。假设客户关怀系统把地址信息存储在应用数据库中。对数据库字段的每次修改都会触发一个消息，通知所有的系统修改相应的信息。根据发布-订购模式的特性，所有订购了地址修改通道的系统都会接收到这个事件。但是，客户关怀系统本身也必须订购这个通道才能接收到其他系统所做的更新事件通知，例如通过一个自服务的 Web 网站。这意味着客户关怀系统将接收到它刚刚发布的消息。这条新接收到的消息将引起数据库更新，进而触发另一条地址修改消息，最终导致地址修改消息的无限循环。为了避免这种无限循环，订购应用要检查消息历史，以确定消息是否来自完全相同的系统，如果是这样，则忽略到来的消息。

例：TIBCO ActiveEnterprise

许多 EAI 集成套件提供了对消息历史的支持。例如，每条 TIBCO ActiveEnterprise 消息的首部包含了跟踪字段，它存有消息所流经全部组件的列表。在使用这种消息时要特别注意，TIBCO ActiveEnterprise 组件为每个输出消息分配了与所消费消息完全相同的消息 ID，这就使跟踪流经多个组件的消息变得更加容易，但是也使得消息 ID 不再成为系统范围内唯一的属性，因为有多个不同的消息会共享相同的 ID。例如，实现一个接收表时，TIBCO ActiveEnterprise 会把所消费消息的 ID 复制给每个输出的消息。

下面的例子给出了流经多个组件的消息的内容，这些组件包括两个 IntegrationManager 过程，分别名为 OrderProcess 和 VerifyCustomerStub。

```
tw.training.customer.verify.response
{
  RVMSG_INT      2  ^pfmt^        10
  RVMSG_INT      2  ^ver^         30
  RVMSG_INT      2  ^type^        1
  RVMSG_RVMSG  108  ^data^
```

```
  {
    RVMSG_STRING  23  ^class^      "VerifyCustomerResponse"
    RVMSG_INT      4  ^idx^       1
    RVMSG_STRING   6  CUSTOMER_ID  "12345"
    RVMSG_STRING   6  ORDER_ID     "22222"
    RVMSG_INT      4  RESULT       0
  }
  RVMSG_RVMSG  150  ^tracking^
  {
    RVMSG_STRING  28  ^id^  "4OEaDEoiBIpcYk6qihzzwB5Uzzw"
    RVMSG_STRING  41  ^1^
 "imed_debug_engine1-OrderProcess-Job-4300"
    RVMSG_STRING  47  ^2^
 "imed_debug_engine1-VerifyCustomerStub-Job-4301"
  }
}
```

11.6 消息存储库



正如在“消息历史”中所介绍的，松耦合的结构化原则为解决方案提供了灵活性，但是也使它很难洞察集成解决方案的动态行为。

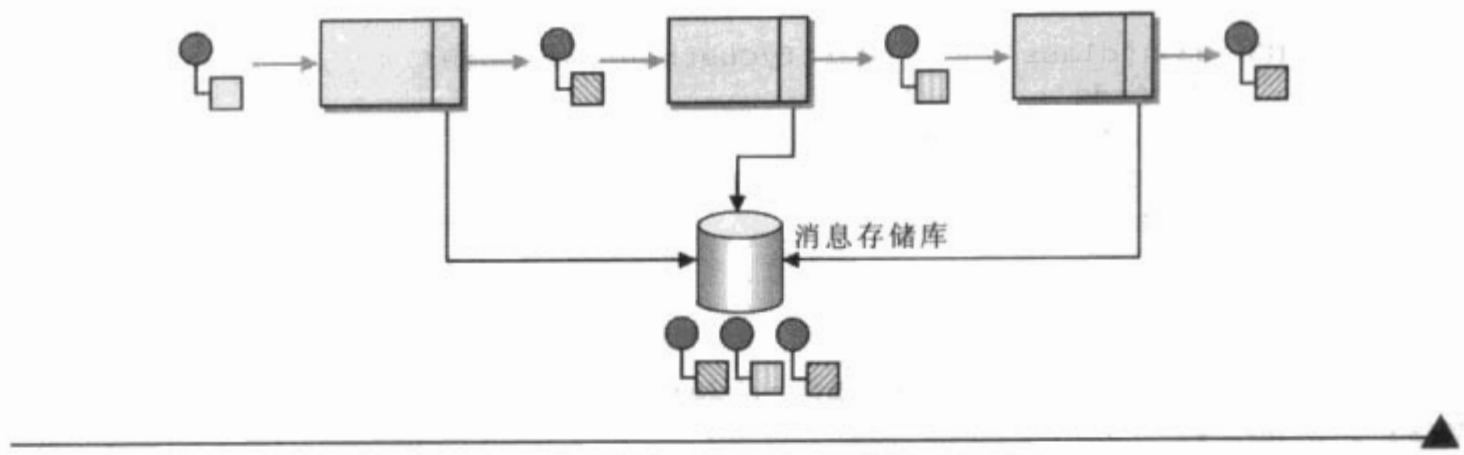
如何对消息的信息做出报告，而且不影响消息传递系统的松耦合和暂时性？

消息传递的一些关键特性既为它带来了强大的功能，也导致它很难管理。异步的消息传递能够保证消息的传送，但是无法保证消息何时传输。不过，对许多实际的应用来说，系统的响应时间非常关键。另外，尽管异步消息传递分别处理每个消息，但是涉及多个消息的信息，例如在一定的时间间隔内流经系统的消息数就非常有用。

消息历史模式说明了告知消息“来源”的好处。通过这一信息，我们可以得出感兴趣的消息吞吐量和运行时统计信息。唯一不足的是，该信息包含在各个单个的消息中。要想得到系统的统计信息并不容易，因为它涉及到多个消息。另外，消息的生命期非常短。一旦消息被消费，就无法再得到消息历史。

为了得到有意义的统计数据，需要把消息数据持久地存储在一个集中的位置。

使用消息存储库将每个消息的信息捕获到一个集中的地方。



使用消息存储库时，可以利用消息传递基础设施的异步特性。把一个消息发送到某个通道时，可以把消息的副本发送到一个特殊的通道，消息存储库采用这个通道收集消息。组件可自己实现这一功能，也可以在通道中插入线路分接器来实现。可以把传送消息副本的副通道作为控制总线的一部分。采用“发送并遗忘”的方式来发送消息副本，不会降低主应用的消息流速度。但是，这确实会增加网络通信量。这也是我们为什么不存储整个消息，而只存储用于事后分析的少量关键字段的原因，比如消息 ID 或带时间戳的消息发送时使用的通道。

要多么详细地存储信息，这是一项重要的设计决策。显然，对各个消息存储的数据越多，统计得到的结果就越好。其副作用是网络通信量和消息存储库的存储容量会增加。即使我们把所有消息数据都存储下来，最终能得到的统计信息仍然是有限的。消息一般会共享相同的消息首部结构，但是不同类型消息的消息体有不同的结构，很难由外部的应用访问（例如，消息体可能包含了串行化的 Java 对象）。所以我们很难统计消息体中的数据元素。

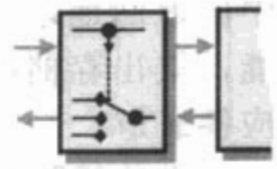
由于每种消息类型的消息体有不同格式的数据，因此必须采用不同的存储选择。如果为了匹配每种消息类型的内部数据结构而创建不同的存储模式（如表），就可以对消息的内容建立索引，并能执行复杂的搜索。但这样做的前提条件是，必须为每种消息类型建立不同的存储结构，这样会极大地增加维护的开销。相反，也可以把消息数据作为非结构化的数据存储在一个很长的 XML 格式字符字段中。这样我们就能使用一个通用的存储模式。我们仍然可以对首部字段进行检索，但是无法统计消息体中的字段。不过，一旦能标识特定的消息，就能根据存储在消息存储库中的 XML 文档来重新创建消息的内容。或者也可以使用 XML 存储库来存储消息，这些存储库可以对 XML 文档建立索引，以便以后提取和分析数据。

消息存储库可能会变得非常大，因此我们很可能需要引入清除机制。清除机制可将较老的消息记录转移到后备数据库中，或者把它们全部删除掉。

例：商业 EAI 工具

某些企业集成工具提供了消息存储库。例如，MSMQ 队列能自动把发送或接收到的消息存储到一个 Journal（日志）队列中。Microsoft BizTalk 会有选择地把所有文档（消息）存储到一个 SQL Server 数据库中，以便事后分析。

11.7 智能代理



采用一对线路分接器可以跟踪流经某个组件的消息，但是使用这种方法的前提条件是，组件要把消息发布到一个固定的输出通道。然而，许多服务式的组件需要把应答消息发布到请求消息中的返回地址所指定的通道中。

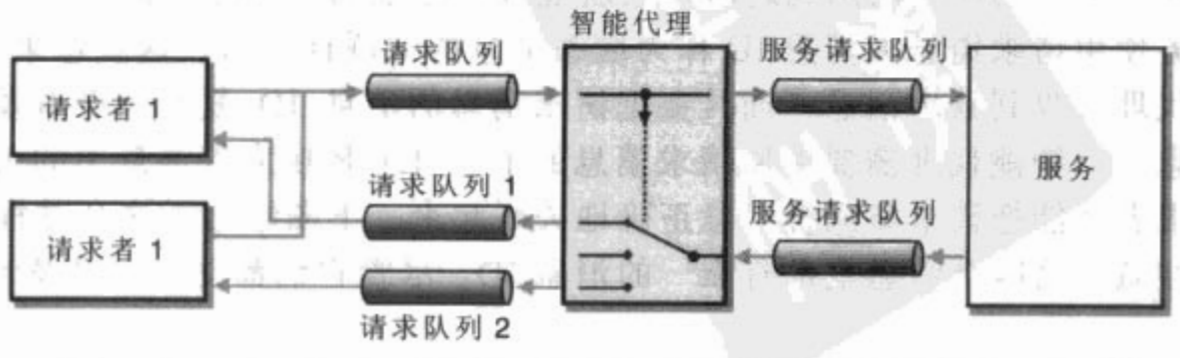
如果一个服务组件采用请求者指定的返回地址作为应答消息的返回通道，应该如何跟踪这样的消息？

为了跟踪流经某服务的消息，需要同时捕获请求和应答消息。使用线路分接器足以拦截请求消息。但是要拦截应答消息却很困难，因为服务要根据请求者指定的返回地址把应答消息发布到不同的通道中。

大多数请求/应答服务都要求提供返回地址支持，请求者可以指定发送应答消息的通道。如果修改服务，让它把应答消息发送到一个固定的通道，会使各个请求者很难获得正确的应答消息。有些消息传递系统允许消费者查看一个应答队列中的指定消息，但是这种方法与消息传递系统的具体实现相关，并且如果应答消息并不是返回给请求者，而是发送给第三方，此时采用这种方法也不合适。

正如在“线路分接器”中讨论的，修改组件从而让它检查消息，这种做法并不一定可行或实用。如果我们集成的是已经封装好的应用，就无法修改应用的代码，而是要在应用之外实现解决方案。同样地，我们也不能要求每个应用都实现消息检查逻辑，这主要是因为，根据在测试模式还是生产模式中运行，检查逻辑可能会有所不同。把检查功能放在一个不同的、自包含的组件中实现，能提高系统的灵活性、重用性和可测试性。

使用智能代理存储由原请求者提供的返回地址，并用智能代理自己的地址代替消息中的返回地址。当服务发送应答消息时，再把消息路由到原来的返回地址。



智能代理把请求通道中发送给请求/应答服务的消息拦截下来。对每条到来的消息，智能代理把原发送者指定的返回地址存储起来，然后用自己正在监听的应答通道代替消息中的返回地址。当应答消息到达智能代理监听的通道时，智能代理会完成所需的分析功能，取出存储的返回地址，并使用消息路由器把应答消息不加任何修改地转发到原来的应答通道。

如果外部服务不支持返回地址，只能向固定的通道返回应答，此时采用智能代理也很有用。可以用智能代理作为这种外部服务的代理，由智能代理提供返回地址支持。在这种情况下，智能代理不需要完成分析功能，只需把应答消息转发给正确的通道。

智能代理必须存储原请求者提供的返回地址，这样才能把到来的应答消息与返回地址关联起来，并把消息转发给正确的通道。智能代理可以把这个数据存储到两个地方：

1. 存储到消息中。
2. 存储到智能代理中。

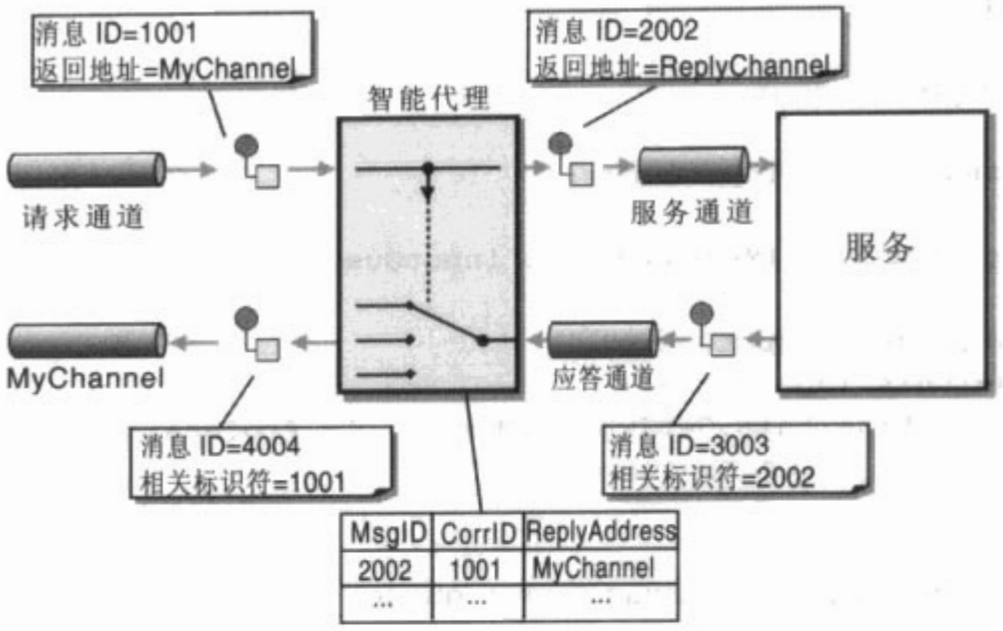
为了把返回地址存储到消息中，智能代理可以在消息中增加一个新的消息字段，其中包含返回地址。请求/应答服务必须把这个新增的字段复制到应答消息中。所有智能代理必须从应答消息中提取这个特殊的消息字段，并从消息中将这个字段删除，把消息转发给该字段指定的通道。这种解决方案可以保持智能代理的简单性，但是需要请求/应答服务的协作。如果请求/应答服务是不可修改的组件，就不能使用这种方法。

另一种方法是，智能代理可以把返回地址存储在一个专门的存储区中，如一个内存结构或关系数据库中。由于智能代理的目的是跟踪请求和应答消息之间的消息，因此智能代理通常要存储来自请求消息的数据，从而把请求消息与应答消息关联起来，这样这些消息才能联合起来加以分析。采用这种方法，要求智能代理能够把应答消息反过来与请求消息相关联。大多数请求/应答服务都支持关联标识符，可以把这个标识符从请求消息复制到应答消息中。如果智能代理无法修改原来的消息格式，可以使用这个字段来关联请求和应答消息（这种做法也可能会遭到滥用）。

不过，智能代理最好能构造自己的关联标识符，因为并不是所有的请求者都会指定关联标识符，而且所提供的关联标识符只需对单个请求者发送的所有请求消息保持唯一，不能对多个请求者发送的所有请求消息都保证唯一。从服务到智能代理的一个服务应答队列中，可能包含了来自多个请求者的消息，因此使用原来的关联标识符不再可靠。智能代理要存储原来的关联标识符以及原来的返回地址，并用自己的关联标识符来代替原来的关联标识符，这样一来，当应答消息到达时，它就能取回原来的关联标识符和返回地址。

有些服务使用请求消息的消息 ID 作为应答消息的关联标识符，这就带来另一个问题。服务从智能代理接收到请求消息，而且要把请求消息的消息 ID 复制到准备发送给智能代理的应答消息中。智能代理需要用原请求消息的消息 ID 替换应答消息中的关联标识符，只有这样请求者才能把请求和应答消息正确地关联起来。下图给出了这个过程。

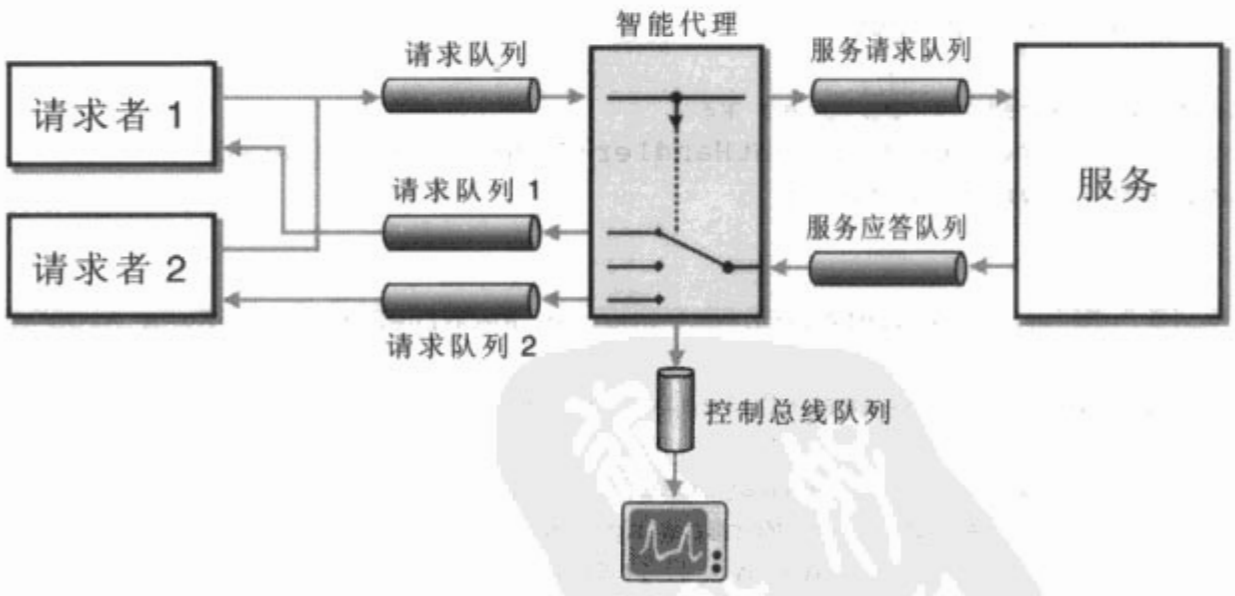
要特别注意，这四个消息全都有唯一的消息 ID，尽管它们都与一个“逻辑”消息的流程相关。



存储和替换关联标识符和返回地址

例：采用 MSMQ 和 C#实现简单的智能代理

实现智能代理并不像说起来的那么复杂。下面的代码实现了一个解决方案，其中包括两个请求者、一个智能代理和一个简单的服务。智能代理把消息处理时间传递给控制总线，以便在控制台中显示这一信息。我们想让请求者通过 Message 对象提供的消息 ID 或数字类型的 AppSpecific 属性使请求者相关联。



简单的智能代理示例

为了编码方便，我们定义了一个基类 MessageConsumer，它封装了创建事件驱动消息消费者的代码。继承这个类的子类只要重载虚方法 ProcessMessage，完成所需的消息处理即可，而不用考虑消息队列的配置或事件驱动处理。分离出这些代码，把它放到一个公共的基类中，这使得创建测试客户端和虚拟的请求/应答服务变得非常简单，只要编写几行代码就可以了。

MessageConsumer

```
public class MessageConsumer
{
    protected MessageQueue inputQueue;

    public MessageConsumer (MessageQueue inputQueue)
    {
        this.inputQueue = inputQueue;
        SetupQueue(this.inputQueue);
        Console.WriteLine(this.GetType().Name +  ": Processing messages from " +
                          inputQueue.Path);
    }

    protected void SetupQueue(MessageQueue queue)
    {
        queue.Formatter = new System.Messaging.XmlMessageFormatter
                              (new String[] {"System.String,mscorlib"});
        queue.MessageReadPropertyFilter.ClearAll();
        queue.MessageReadPropertyFilter.AppSpecific = true;
        queue.MessageReadPropertyFilter.Body = true;
        queue.MessageReadPropertyFilter.CorrelationId = true;
        queue.MessageReadPropertyFilter.Id = true;
        queue.MessageReadPropertyFilter.ResponseQueue = true;
    }

    public virtual void Process()
    {
        inputQueue.ReceiveCompleted +=
            new ReceiveCompletedEventHandler(OnReceiveCompleted);
        inputQueue.BeginReceive();
    }

    private void OnReceiveCompleted(Object source, ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;

        Message m = mq.EndReceive(asyncResult.AsyncResult);
        m.Formatter =  new System.Messaging.XmlMessageFormatter
                           (new String[] {"System.String,mscorlib"});

        ProcessMessage(m);

        mq.BeginReceive();
    }

    protected virtual void ProcessMessage(Message m)
    {
        String text = "";
```

```
        try
        {
            text = (String)m.Body;
        }
        catch (InvalidOperationException) {};
        Console.WriteLine(this.GetType().Name + ": Received Message " + text);
    }
}
```

以 MessageConsumer 类作为起点，我们可以创建一个智能代理。智能代理包含两个 MessageConsumer，一个针对来自请求者（SmartProxyRequestConsumer）的请求消息，另一个针对请求/应答服务（SmartProxyReplyConsumer）返回的应答消息。智能代理还定义了一个散列表（Hashtable），用于存储请求和应答消息之间的消息数据。

SmartProxyBase

```
public class SmartProxyBase
{
    protected SmartProxyRequestConsumer requestConsumer;
    protected SmartProxyReplyConsumer replyConsumer;

    protected Hashtable messageData;

    public SmartProxyBase(MessageQueue inputQueue,
                          MessageQueue serviceRequestQueue,
                          MessageQueue serviceReplyQueue)
    {
        messageData = Hashtable.Synchronized(new Hashtable());
        requestConsumer = new SmartProxyRequestConsumer(inputQueue, serviceRequestQueue,
                                                        serviceReplyQueue, messageData);
        replyConsumer = new SmartProxyReplyConsumer(serviceReplyQueue, messageData);
    }

    public virtual void Process()
    {
        requestConsumer.Process();
        replyConsumer.Process();
    }
}
```

SmartProxyRequestConsumer 相当简单。它把来自请求消息的信息（消息 ID、返回地址、AppSpecific 属性和当前时间）存储在散列表中，散列表按新的请求消息的消息 ID 进行索引（这些新的请求消息由智能代理发送给真正的服务）。请求/应答服务把这个消息 ID

复制到服务应答消息的 CorrelationID 字段，从而支持实关联标识符。这样，当应答消息到达智能代理时，智能代理就能通过这个关联标识符取回原来存储的消息数据。SmartProxyRequestConsumer还会用智能代理监听应答消息的队列(即 ResponseQueue 属性)替代返回地址。在这个类中使用了 AnalyzeMessage 虚方法，这样它的子类就能完成所需的分析。

SmartProxyRequestConsumer

```
public class SmartProxyRequestConsumer : MessageConsumer
{
    protected Hashtable messageData;
    protected MessageQueue serviceRequestQueue;
    protected MessageQueue serviceReplyQueue;

    public SmartProxyRequestConsumer(MessageQueue requestQueue,
                                     MessageQueue serviceRequestQueue,
                                     MessageQueue serviceReplyQueue,
                                     Hashtable messageData) : base(requestQueue)
    {
        this.messageData = messageData;
        this.serviceRequestQueue = serviceRequestQueue;
        this.serviceReplyQueue = serviceReplyQueue;
    }

    protected override void ProcessMessage(Message requestMsg)
    {
        base.ProcessMessage(requestMsg);

        MessageData data = new MessageData(requestMsg.Id, requestMsg.ResponseQueue,
                                           requestMsg.AppSpecific);
        requestMsg.ResponseQueue = serviceReplyQueue;
        serviceRequestQueue.Send(requestMsg);
        messageData.Add(requestMsg.Id, data);
        AnalyzeMessage(requestMsg);
    }

    protected virtual void AnalyzeMessage(Message requestMsg)
    {
    }
}
```

SmartProxyReplyConsumer 监听服务应答通道。这个类的 ProcessMessage 方法从 SmartProxyRequestConsumer 存储的信息中取得与请求消息相关的数据，并调用

AnalyzeMessage 模板方法。然后它把 CorrelationID 和 AppSpecific 属性复制到新的应答消息中，并把这个消息路由到原请求消息指定的返回地址。

SmartProxyReplyConsumer

```
public class SmartProxyReplyConsumer : MessageConsumer
{
    protected Hashtable messageData;

    public SmartProxyReplyConsumer(MessageQueue replyQueue,
                                   Hashtable messageData) : base(replyQueue)
    {
        this.messageData = messageData;
    }

    protected override void ProcessMessage(Message replyMsg)
    {
        base.ProcessMessage(replyMsg);

        String corr = replyMsg.CorrelationId;
        if (messageData.Contains(corr))
        {
            MessageData data = (MessageData)(messageData[corr]);

            AnalyzeMessage(data, replyMsg);

            replyMsg.CorrelationId = data.CorrelationID;
            replyMsg.AppSpecific = data.AppSpecific;

            MessageQueue outputQueue = data.ReturnAddress;
            outputQueue.Send(replyMsg);
            messageData.Remove(corr);
        }
        else
        {
            Console.WriteLine(this.GetType().Name + "Unrecognized Reply Message");
            //把消息发送给非法消息队列
        }
    }

    protected virtual void AnalyzeMessage(MessageData data, Message replyMessage)
    {
    }
}
```

为了收集度量信息，并把它们发送给控制总线，我们从通用的 SmartProxyBase 类和 SmartProxyReplyConsumer 类分别派生了子类。新的 MetricsSmartProxy 类实例化了 SmartProxyReplyConsumerMetrics，将其作为消费应答消息的类。该类包含一个简单的 AnalyzeMessage 方法，这个方法将计算请求和响应之间的消息处理时间，并把这个数据以及消息的数量发送给控制总线队列。可以很容易地扩展这个方法来完成更复杂的计算。控制总线队列与一个简单的文件书写器连接，把到来的消息写到一个文件中。

MetricsSmartProxy

```
public class MetricsSmartProxy : SmartProxyBase
{
    public MetricsSmartProxy(MessageQueue inputQueue,
                             MessageQueue serviceRequestQueue,
                             MessageQueue serviceReplyQueue,
                             MessageQueue controlBus) :
                    base (inputQueue, serviceRequestQueue, serviceReplyQueue)
    {
        replyConsumer = new SmartProxyReplyConsumerMetrics
                            (serviceReplyQueue, messageData, controlBus);
    }
}
```

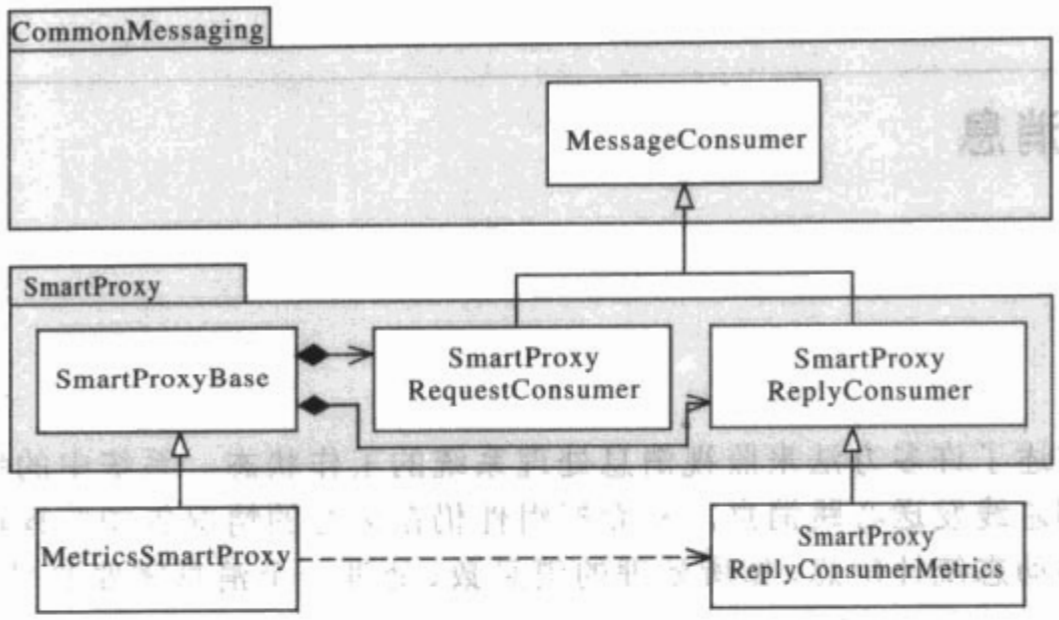
SmartProxyReplyConsumerMetrics

```
public class SmartProxyReplyConsumerMetrics : SmartProxyReplyConsumer
{
    MessageQueue controlBus;

    public SmartProxyReplyConsumerMetrics(MessageQueue replyQueue,
                                          Hashtable messageData,
                                          MessageQueue controlBus) :
                                base(replyQueue, messageData)
    {
        this.controlBus = controlBus;
    }

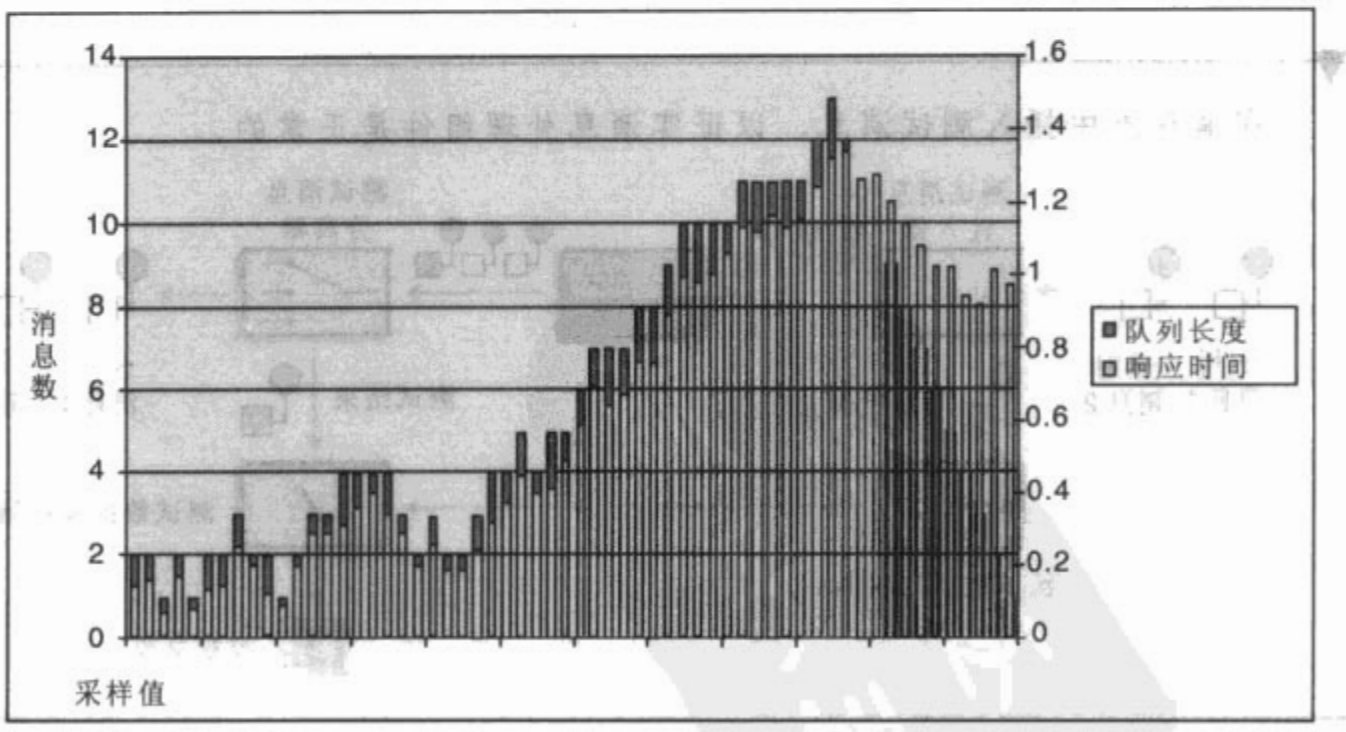
    protected override void AnalyzeMessage(MessageData data, Message replyMessage)
    {
        TimeSpan duration = DateTime.Now - data.SentTime;
        Console.WriteLine(" processing time: {0:f}", duration.TotalSeconds);
        if (controlBus != null)
        {
            controlBus.Send(duration.TotalSeconds.ToString() + "," + messageData.Count);
        }
    }
}
```

下面的类图给出了各个类之间的关系。



智能代理示例的类图

为了测试这个代理，我们创建了一个虚拟的请求/应答服务，它什么也不做，只是随机等待一段时间（0～200ms 之间）。在此为智能代理提供了两个请求者，每个请求者在 100ms 间隔内发布 30 个消息。我们把控制总线队列捕获的消息记录到一个日志文件中。为了演示，这里把所得到的控制总线日志文件加载到一个 Microsoft Excel 电子表格中，并创建了一个看上去不错的图表。

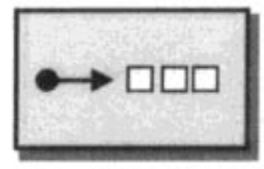


由智能代理收集到的响应时间统计数据，并由控制总线控制台可视化显示

从图中可以看到，队列长度和响应时间会持续增长，直到有 13 个消息排队。从这个时刻起，请求者停止发送新的消息，队列长度稳步下降，响应时间也随之减少；但是，由于

正在处理的消息已经在请求队列中等待了很长时间，因此消息的响应时间仍然维持在1秒左右。

11.8 测试消息

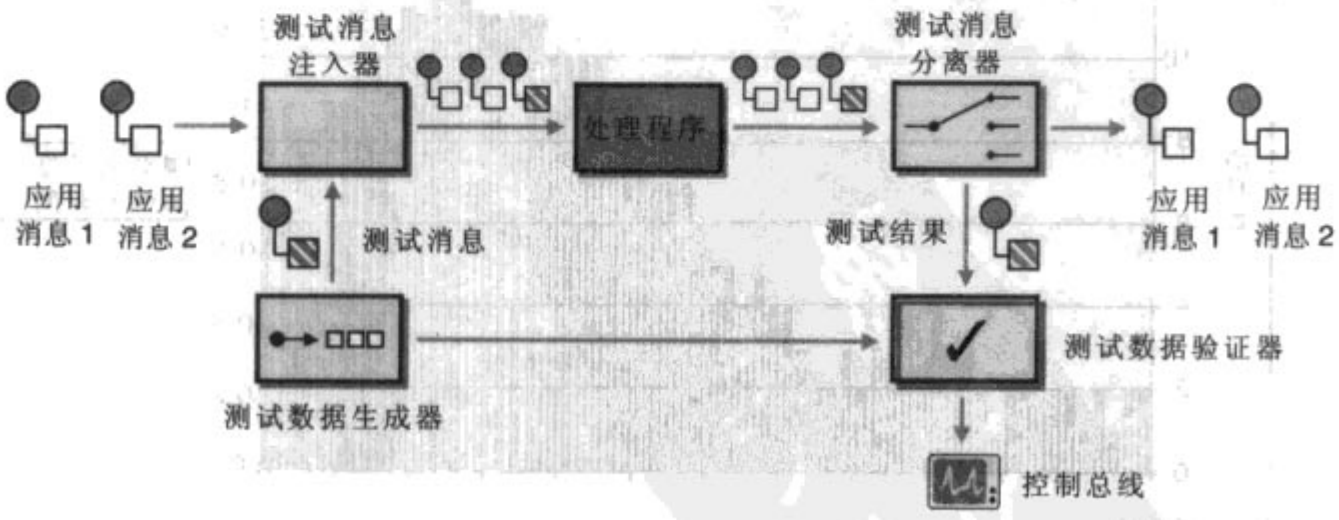


控制总线描述了许多方法来监视消息处理系统的工作状态。系统中的每个组件都可以周期性地向控制总线发送心跳消息，从而把组件仍在运行的情况告知监视机制。心跳消息还可包含组件的动态统计信息，如所处理的消息数、处理一个消息需要的平均时间，或CPU利用率等等。

组件正在处理消息时，如果因内部故障而导致输出消息被修改，会怎么样？

采用简单的心跳机制是无法检测出这种错误的，因为心跳机制只能在组件级运用，并不知道应用消息的格式。

在消息流中插入测试消息，以证实消息处理组件是正常的。



测试消息模式依赖于以下组件：

1. **测试数据生成器**创建要发送给组件的测试消息。测试数据可以是常量，也可以由测试数据文件驱动，或者是随机生成的数据。

2. **测试消息注入器**把测试数据插入到发送给组件的正常消息流中。注入器扮演的主要角色是对消息加标记，以便把真正的应用消息与测试消息区分开来。它在消息首部中插入一个特殊的字段。如果我们不能控制消息的结构，可以对现有的字段使用特殊的值来表示测试消息（例如让 OrderID=999999），这就改变了应用数据的语义，使用同样的字段既表示应用数据（实际的订单号），又表示控制信息（这是一条测试消息）。因此，这种方法只能作为最后的选择。

3. **测试消息分离器**从输出流中提取出测试消息结果。一般可以使用基于内容的路由器实现这个功能。

4. **测试数据验证器**把实际的测试结果与期望的结果进行比较，如果发现不同，则标出一个异常。根据测试数据的性质，验证器可能需要访问最初的测试数据。

如果所测试的组件支持返回地址，则不需要一个明确的测试消息分离器。在这种情况下，测试数据生成器可以把一个特殊的测试通道作为返回地址，使测试消息不再经过余下的系统。从效果上看，返回地址起到了区分测试消息和应用消息的作用。

一般认为测试消息是一种主动监视机制。不同于被动的监视机制，主动监视不依赖于组件生成的信息（如日志文件或心跳消息），而是主动地探测组件。主动监视的优点是通常能获得更深层次的测试结果，因为数据能与应用消息一样经过相同的处理步骤。主动监视还适用于那些不支持被动监视的组件。

主动监视可能存在一个缺点，即会对处理单元增加额外的负担。我们要在测试频率与尽可能减少性能影响这两个方面综合权衡。如果组件的使用是按照使用次数计费的，那么主动监视还会增加成本。许多外部组件就需要收费，如从外部的信用查询机构获取客户的信用报告。

主动监视并不适用于所有组件。有状态的组件可能无法区分测试数据和真正的数据，可能会为测试数据也创建数据库记录。我们可不希望年度收入报告中还包含测试订单！

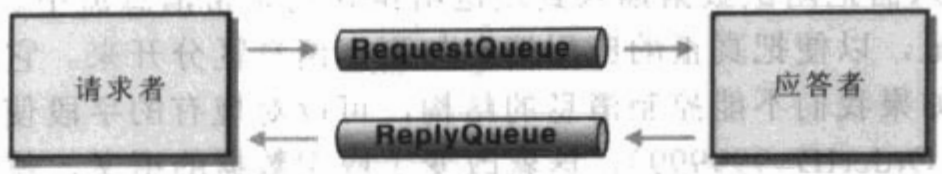
例：贷款中介：测试信用机构

在第 12 章“中场演练：系统管理示例”中，将使用测试消息主动监视外部信用机构。

11.9 通道清洁者



使用 JMS 的请求/应答示例时（见第 6 章“中场演练：简单的消息传递”），遇到了一个简单但是很有趣的问题。这个例子中的请求者向应答者发送消息，并等待响应，它使用了两个点对点通道，RequestQueue 和 ReplyQueue（如下图所示）。



首先启动应答者，然后启动请求者。这时会出现一个奇怪的现象：在应答者还没有确认接收到请求消息前，请求者的控制台窗口已经显示它收到一条响应消息。难道是控制台输出被延迟了？由于不知道原因是什么，我们决定关闭应答者，然后再次运行请求者。更奇怪的是，我们仍然会收到请求消息的应答！难道是魔法？不是的，这只是持久的消息传递产生的副作用。ReplyQueue 通道中存在剩余的消息，这很可能是由早先的失败造成的。每次启动请求者时，请求者要把一条新的消息发送到 RequestQueue 中，然后立即接收到已经位于 ReplyQueue 中的剩余的应答消息。我们没有注意到，这个消息并不是请求者刚刚发送的请求消息的应答！一旦应答者接收到新的请求消息，它就把新的应答消息放到 ReplyQueue 中，使得下一次测试再次出现“魔法”现象。由此可见，即使在最简单得环境下，异步消息传递也会给你设下圈套！

如何防止在通道中剩余的消息干扰测试和正在运行的系统？

消息通道设计为：即使接收组件不可用也能可靠地传送消息。为了实现这个功能，通道必须保留在其中传递的消息。进行测试时，或者当某个组件出现异常时（没有使用事务性消息消费和生产机制），这个原本有用的特性却会引发混乱。正如前面介绍过的，我们会立即接收到通道中已有的无关消息。这些消息导致测试消息无法进入系统，除非将通道中已有的消息消费完。如果已有的消息是价值好几百万美元的订单，反而是件好事。但是，如果我们正在测试或调试一个系统，它的一个通道填满了查询消息或应答消息，这就会给我们带来很大的麻烦。

在前面的简单例子中，如果使用了关联标识符，或许可以使调试容易一些。使用标识符，请求者能识别出到来的消息并不是请求者刚刚发出的消息的应答。然后它能把旧的应答消息丢掉，或把该消息路由给非法消息通道，非法消息通道能有效地删除“多余的”消息。在另外一些情况下，则很难检测出重复或不想要的消息。例如，如果某个特殊的消息格式不正确并引起消息接收者失败，除非把错误的消息删除，否则接收者无法重新启动，因为一旦收到这个消息，接收者会再次立即失败。当然，这个例子要求必须修正接收者的问题（即使消息格式不正确，也不能导致组件失败），但是，通过删除消息就能使系统很快地启动和运行，直到最终修正组件的问题。

要避免在通道中剩余消息，另一种方法是使用临时通道（如 JMS 提供了 createTemporaryQueue 方法，可以创建临时通道）。这些通道适用于请求/应答应用，一旦程序关闭与消息传递系统的连接，临时通道中所有的消息将丢失。但是这种方法只能用于简单的请求/应答示例，不能清除其他持久通道中剩余的消息。

看起来事务管理可以消除额外的消息，因为消息的消费、处理和发布是在一个事务中进行的。因此，如果某个组件在消息处理步骤中异常中止，该消息并没有被消费。同样的，只有组件完成最后的事务提交，才能把应答消息发送出去。但是，我们要记住，事务并不能防范编程错误。在前面简单的请求/应答示例中，程序员错误可能引起请求者无法从ReplyQueue通道中读取响应消息。因此，尽管采用了事务机制，消息也会阻塞在通道中，并带来前面所述的症状。

使用通道清洁者把不需要的消息从通道中删除。



一个基本的通道清洁者只是把通道中所有的消息删除。如果只是把系统复位到一致的状态，采用这种清洁者就足够了。如果我们正在调试一个生产系统，可能需要根据指定的标准（如消息ID或特定的消息字段值），删除某个消息或一组消息。

在很多情况下，采用通道清洁者把消息从通道中删除并丢掉就足够了。在另外一些情况下，通道清洁者还要把删除的消息保存起来，以便事后检查或重放。如果通道中的消息导致系统出现故障，把消息记录下来是很有用的，这样就可以删除消息并继续执行操作。但是，一旦问题被更正，可能想重新注入消息，从而使系统不遗漏消息的内容。在重新注入消息前还可能要编辑消息的内容。这种功能结合了通道清洁者和消息存储库的某些特性。

例：采用JMS实现的通道清洁者

这是一个用Java实现的简单的通道清洁者。它只是把通道中的所有消息删除。ChannelPurger类引用了两个外部类，这里没有给出这两个外部类的源代码：

1. JMSEndpoint——所有JMS参与者的基类。它为Connection和Session实例提供预先已初始化的实例变量。

2. JNDIUtil——实现辅助功能，把通过JNDI查找JMS对象的功能封装起来。

```
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;

public class ChannelPurger extends JmsEndpoint
{

    public static void main(String[] args)
    {
```

```
        if (args.length != 1) {
            System.out.println("Usage: java ChannelPurger <queue_name>");
            System.exit(1);
        }
        String queueName = new String(args[0]);
        System.out.println("Purging queue " + queueName);

        ChannelPurger purger = new ChannelPurger();

        purger.purgeQueue(queueName);

    }

    private void purgeQueue(String queueName)
    {
        try {
            initialize();
            connection.start();
            Queue queue = (Queue) JndiUtil.getDestination(queueName);

            MessageConsumer consumer = session.createConsumer(queue);

            while (consumer.receiveNoWait() != null)
                System.out.print(".");
            connection.stop();
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString());
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                    // 忽略
                }
            }
        }
    }
}
```

第 12 章

中场演练：系统管理示例

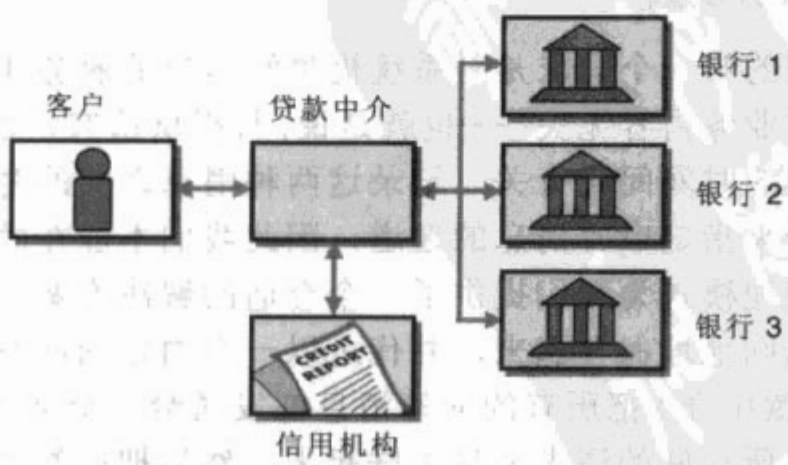
12.1 贷款中介系统的管理

本章将提供一个更详细的例子，通过这个例子说明如何使用这一部分介绍的系统管理模式来监视和控制消息传递解决方案。第 9 章“中场演练：复合消息传递”中给出了贷款中介示例的 C#和 MSMQ 实现，本例就建立在这个实现基础上（见“使用 MSMQ 的异步实现方案”）。我们要扩展而不是修改原来的解决方案，因此你不一定需要复习以前的所有代码。与原先的实现一样，举这个例子的目的不是为了解释如何使用 MSMQ 特定的 API，而是为了说明如何采用面向队列的消息传递系统实现本书中讨论的模式。这个解决方案的结构与使用 JMS 队列或 IBM WebSphere MQ 实现的 Java 示例非常相似。由于我们把主要精力集中在了设计决策和折衷方案上，所以即使你不是 C#或 MSMQ 开发人员，本章的内容对你也很有价值。

12.1.1 调整贷款中介

贷款中介的实现由下面四个关键组件组成（如下图所示）：

- 客户（或测试客户端）提出贷款利率查询请求。
- 贷款中介扮演了集中的过程管理器的角色，并协调信用机构与银行之间的通信。
- 信用机构为贷款中介提供一个服务，计算客户的信用度。
- 各个银行接收来自贷款中介的利率查询请求，并根据贷款参数提交利率信息。



在大多数集成应用环境中，我们不能访问应用的内部，也很难从外部监视和管理组件。

为了尽可能模拟真实的环境，在这个例子中我们把各个已有的组件当作黑箱看待。在这个约束条件下，要让这个管理解决方案满足以下需求：

- **管理控制台**：我们想通过一个前端显示所有组件的工作状态，如果出现什么问题，可以采取相应的补救措施。
- **贷款中介的服务质量**：在原来的解决方案中，我们开发了一个测试客户端，它负责监视贷款中介的利率查询请求和应答之间的响应时间。在真实的产品环境中，客户不会为我们提供这样的功能（但是他们会抱怨系统太慢）。因此，需要采用管理解决方案捕获这一信息，并把它转发给管理控制台。
- **验证信用机构的操作**：信用机构是由第三方提供的外部服务。通过周期性地发送测试消息，希望确信该服务在正常运行。
- **信用机构的故障恢复**：如果信用机构出现故障，希望把信用请求消息临时重定向给其他服务提供者。

12.1.2 管理控制台

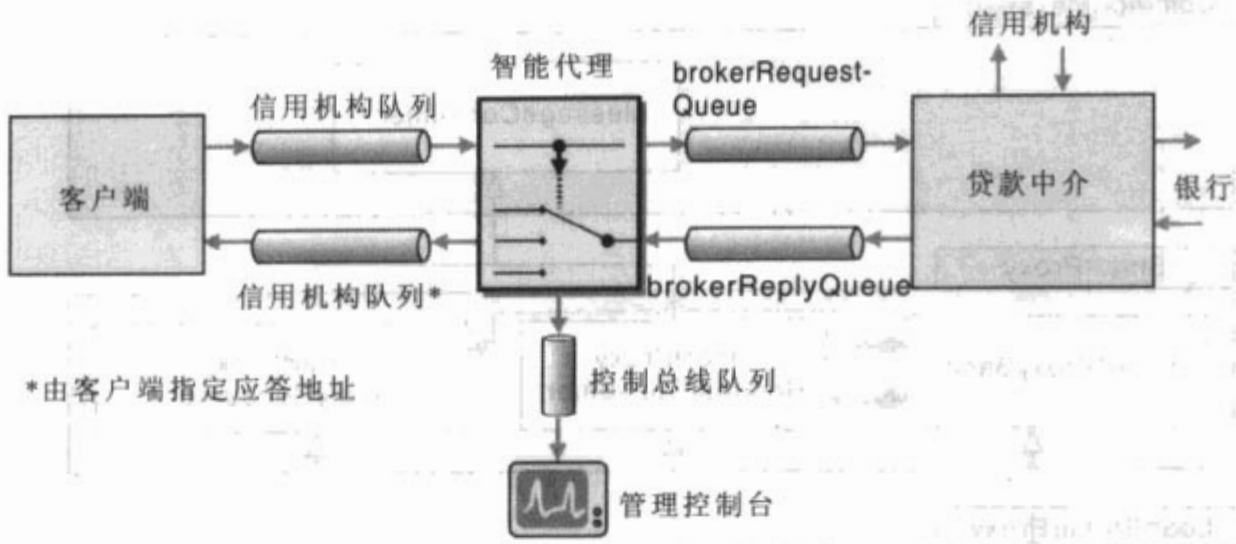
为了评估整个解决方案的质量，必须把来自多个组件的测量指标汇集到一个集中的管理控制台上。该控制台还要能控制消息流和组件的参数，这样我们才能通过改变消息的路径或修改组件的行为来解决数据传递问题。

管理控制台通过消息传递与每个组件进行通信。它使用单独的控制总线，其中只包含与系统管理相关的消息，但不包含应用数据。

由于本书是一本介绍企业集成设计的书，而不是介绍用户界面设计的书，所以我们把管理控制台设计得非常简单。许多开发商提供了实时的数据显示，读者也可以使用 Visual Basic 和 Microsoft Office 组件（如 Excel）来获得可视化效果。另外，许多操作系统和程序设计平台都提供了自己的工具框架，如 Java/JMX（Java Management Extensions，Java 管理扩展）或 Microsoft 的 WMI（Windows Management Instrumentation，Windows 管理工具）。我们推翻了以前的解决方案，使它更少地依赖特定开发商的 API，同时会给出监视解决方案的内部工作流程。在实现特定的管理功能时，就会建立这个控制台。

12.1.3 贷款中介的服务质量

实现贷款中介管理的第一个需求是对系统提供给客户的服务质量进行度量。这类监视信息与每个消息包含的业务内容无关——也就是说，与提供给客户的贷款利率无关——而只与请求和应答消息之间的时间间隔有关。记录这两种消息之间的时间间隔时应该注意，客户端可能采用返回地址来指定应答消息的通道，因此我们不能在固定的通道上监听应答消息。幸运的是，智能代理模式为我们提供了一个合适的解决方案。智能代理会拦截请求消息，把客户端提供的返回地址保存起来，并代之以一个固定的应答通道地址。因此，服务（在我们的例子中是贷款中介）把所有的应答消息都发送给一个通道。智能代理监听该通道并把到来的应答消息与所存储的请求消息关联起来。然后把应答消息转发给客户端最初指定的返回地址（如下图所示）。



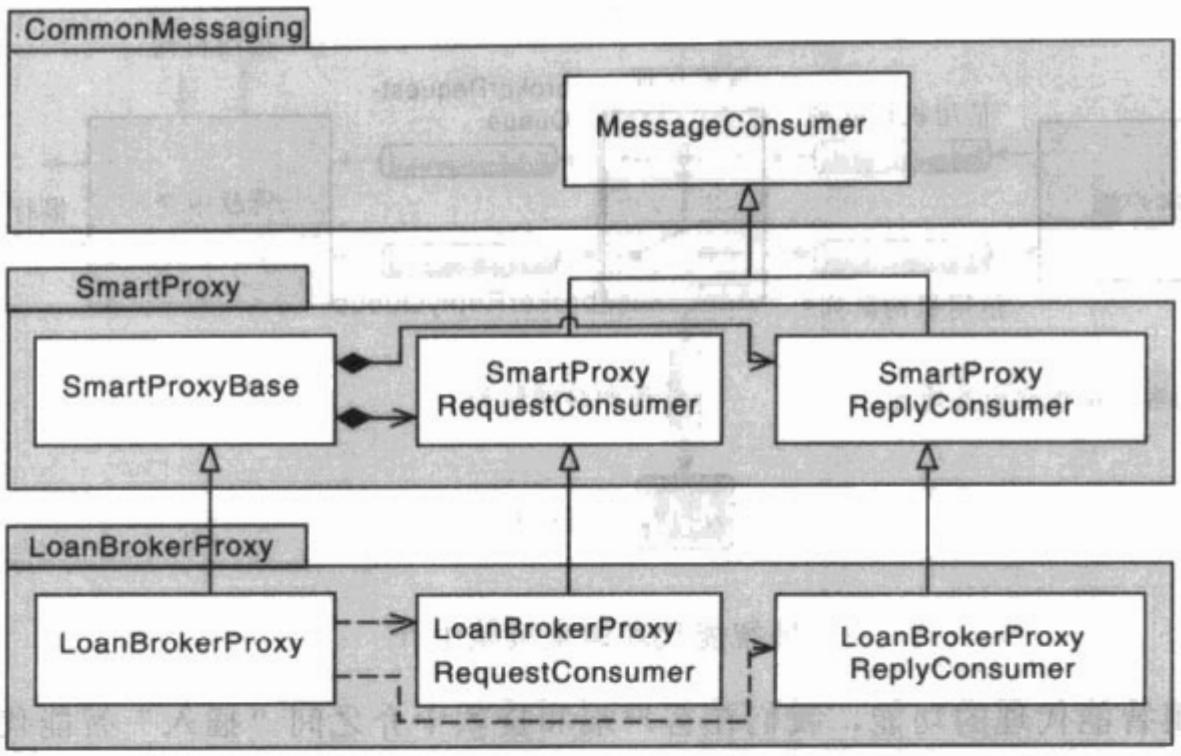
用智能代理调整贷款中介

为了利用智能代理的功能，我们在客户端和贷款中介之间“插入”智能代理（如上图所示）。该智能代理对客户端透明，因为智能代理监听的通道与贷款中介最初监听的通道相同（loanRequestQueue）。我们现在利用新参数启动贷款中介，使它监听 brokerRequestQueue 通道，而不是 loanRequestQueue 通道。智能代理要求贷款中介把所有应答消息返回到 brokerReplyQueue 通道，然后它再把这些消息正确地转发给客户端最初指定的返回地址。

我们想通过智能代理来度量贷款请求的响应时间以及贷款中介每次处理的请求数。通过捕获接收到请求消息的时刻，智能代理可以度量出请求和应答消息之间的时间间隔。当它接收到相关的应答消息时，智能代理从当前时刻减去请求时刻，就能计算出请求和应答之间的时间间隔。通过统计目前有多少正在进行处理的请求消息（还没有接收到相应的应答消息），智能代理可以估计出贷款中介每次管理的活跃请求消息有多少。智能代理无法区分 brokerRequestQueue 通道中排队等候的消息与贷款中介正在处理的消息，因此统计的结果是这两种消息的总和。只要接收到请求消息或应答消息，都会更新正在处理的请求消息数。

通过 controlBusQueue 通道，智能代理把这些度量信息传递给管理控制台，以便进行监视和分析。针对每个消息，都可以发送统计信息，但是如果处理的消息量很大，就会造成网络阻塞。在消息流中插入智能代理已经使发送的消息量加倍（对于原先的一对请求和应答消息，现在则有两对），因此我们不希望再为每条请求消息增加新的控制消息。或者，智能代理可以使用定时器，按照预定的时间间隔把度量消息发送到控制总线。例如，每隔 5 秒。度量消息中可以包含统计值（如最大、最小和平均响应时间）或在这个时间间隔内所传递全部消息的详细信息。为了尽可能减少度量消息的大小、简化管理控制台，我们决定只把统计信息发送给控制台。

为了实现贷款中介的智能代理，这里重用了介绍智能代理模式时使用的几个 SmartProxy 基类。分别派生了 SmartProxyBase、SmartProxyRequestConsumer 和 SmartProxyReplyConsumer 的子类（如类图所示）。这些类的源代码请参考智能代理模式。



贷款中介的智能代理类图

就像原来的 SmartProxyBase 类一样，新的 LoanBrokerProxy 类包含两个不同的消息消费者，一个用于消费来自客户端的请求消息（LoanBrokerProxyRequestConsumer），另一个用于消费来自贷款中介的应答消息（LoanBrokerProxyReplyConsumer）。这两个消费者类分别从各自的基类继承（SmartProxyRequestConsumer 和 SmartProxyReplyConsumer），并增加了 AnalyzeMessage 方法的一个新实现。

下面让我们来看看 LoanBrokerProxy 类的具体实现。与 SmartProxyBase 基类中一样，LoanBrokerProxy 类的构造函数有同样的参数，另外还增加了对控制总线队列的引用，并报告间隔（以秒为单位）。

这个类维护了两个度量集合 ArrayLists：performanceStats 和 queueStats。前者收集请求/应答时间间隔（以秒为单位），后者收集正在处理的请求消息的数量（包括在 brokerRequestQueue 中排队的消息或正由贷款中介处理的消息）。预先设定的定时器触发时，OnTimerEvent 方法会从这两个集合中取出当前数据的一个快照。随着接收到新的消息，消息消费者会不断地把新数据添加到度量集合中，因此必须利用这个快照副本对数据做进一步分析。

LoanBrokerProxy 类

```
public class LoanBrokerProxy : SmartProxyBase
{
    protected MessageQueue controlBus;

    protected ArrayList performanceStats;
    protected ArrayList queueStats;
```

```
protected int interval;
protected Timer timer;

public LoanBrokerProxy(MessageQueue inputQueue, MessageQueue serviceRequestQueue,
                       MessageQueue serviceReplyQueue, MessageQueue controlBus,
                       int interval) :
    base (inputQueue, serviceRequestQueue, serviceReplyQueue)
{
    messageData = Hashtable.Synchronized(new Hashtable());
    queueStats = ArrayList.Synchronized(new ArrayList());
    performanceStats = ArrayList.Synchronized(new ArrayList());

    this.controlBus = controlBus;
    this.interval = interval;

    requestConsumer = new LoanBrokerProxyRequestConsumer(inputQueue,
        serviceRequestQueue, serviceReplyQueue, messageData, queueStats);
    replyConsumer = new LoanBrokerProxyReplyConsumer(serviceReplyQueue,
        messageData, queueStats, performanceStats);
}

public override void Process()
{
    base.Process();

    TimerCallback timerDelegate = new TimerCallback(OnTimerEvent);
    timer = new Timer(timerDelegate, null, interval*1000, interval*1000);
}

protected void OnTimerEvent(Object state)
{
    ArrayList currentQueueStats;
    ArrayList currentPerformanceStats;

    lock (queueStats)
    {
        currentQueueStats = (ArrayList)(queueStats.Clone());
        queueStats.Clear();
    }
```

```
        lock (performanceStats)
        {
            currentPerformanceStats = (ArrayList)(performanceStats. Clone());
            performanceStats.Clear();
        }

        SummaryStats summary = new SummaryStats(currentQueueStats,
                                                currentPerformanceStats);
        if (controlBus != null)
            controlBus.Send(summary);
    }
}
```

LoanBrokerProxy 使用 SummaryStats 结构来缩减各个度量数据，只保留最大、最小和平均值，并把统计结果发送给控制总线。每到来一个消息，我们就更新统计信息，这样就只需存储统计数据，而不必存储每个时刻的度量数据，从而可以提高系统评价的效率。不过，通过把计算放在最后，这样我们也可以很方便地调整发布给控制总线的信息。

LoanBrokerProxyRequestConsumer 类负责处理到来的请求消息。SmartProxyRequestConsumer 基类负责把相关的消息数据存储到 messageData 散列表中。类似的，SmartProxyReplyConsumer 基类接收到应答消息时，把数据从散列表中删除。因此，可以根据 messageData 散列表的长度推算出当前正在处理的请求消息的数量。LoanBrokerProxyRequestConsumer 保存了对 queueStats 集合的引用（这个集合存储在 LoanBrokerProxy 中），这样一来，LoanBrokerProxyRequestConsumer 就可以把新的度量值添加到这个集合中。

LoanBrokerProxyRequestConsumer 类

```
public class LoanBrokerProxyRequestConsumer : SmartProxyRequestConsumer
{
    ArrayList queueStats;

    public LoanBrokerProxyRequestConsumer(MessageQueue requestQueue,
                                          MessageQueue serviceRequestQueue,
                                          MessageQueue serviceReplyQueue,
                                          Hashtable messageData,
                                          ArrayList queueStats) :
        base(requestQueue, serviceRequestQueue, serviceReplyQueue, messageData)
    {
        this.queueStats = queueStats;
    }

    protected override void ProcessMessage(Message requestMsg)
    {
        base.ProcessMessage(requestMsg);
```

```
        queueStats.Add(messageData.Count);
    }
}
```

应答消息到达时，LoanBrokerProxyReplyConsumer 收集所需的两种度量值。首先，它计算发送请求消息和接收应答消息之间的时间间隔，并把这个度量值添加到 performanceStats 集合中。其次，它捕获正在处理的请求消息的数量（再次使用了 messageData 散列表的大小），并把这个度量值添加到 queueStats 集合中。

LoanBrokerProxyReplyConsumer 类

```
public class LoanBrokerProxyReplyConsumer : SmartProxyReplyConsumer
{
    ArrayList queueStats;
    ArrayList performanceStats;

    public LoanBrokerProxyReplyConsumer(MessageQueue replyQueue,
                                        Hashtable messageData,
                                        ArrayList queueStats,
                                        ArrayList performanceStats) :
          base(replyQueue, messageData)
    {
        this.queueStats = queueStats;
        this.performanceStats = performanceStats;
    }

    protected override void AnalyzeMessage(MessageData data, Message replyMessage)
    {
        TimeSpan duration = DateTime.Now - data.SentTime;
        performanceStats.Add(duration.TotalSeconds);

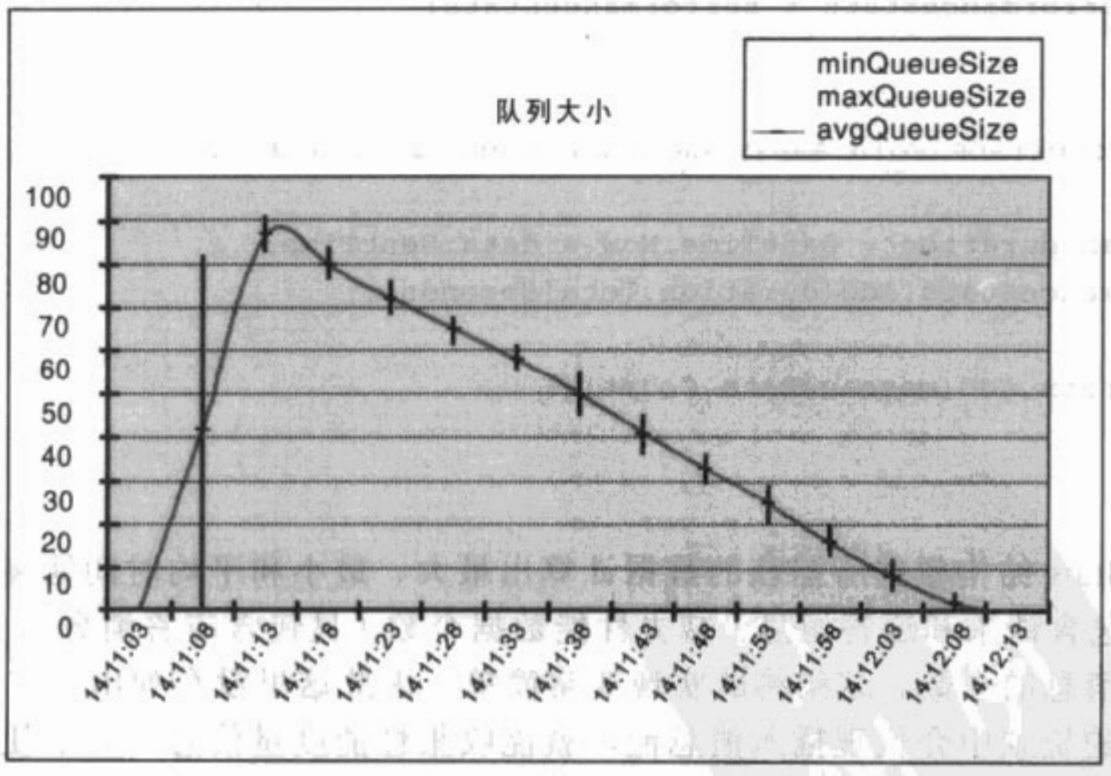
        queueStats.Add(messageData.Count);
    }
}
```

SummaryStats 结构根据所捕获的数据计算出最大、最小和平均时间间隔。将队列的大小数据个数（包含请求和应答消息）减去性能数据个数（只包含应答消息），可以计算出正在处理的请求消息的个数。这结构的实现非常简单，因此这里没有列出。

一旦把新的贷款中介代理插入消息流，就能收集性能度量信息。为了便于举例，我们配置了两个测试客户端，它们各自发送 50 个贷款中介请求。代理收集到如下所示的结果（我们使用一个简单的 XSL 转换，把以 XML 格式发布给 controlBusQueue 通道的度量数据转换为 HTML 表格）。

时间戳	请求数	应答数	最小处理时间	平均处理时间	最大处理时间	最小队列长度	平均队列长度	最大队列长度
14:11:02.9644424	0	0	0.00	0.00	0.00	0	0	0
14:11:07.9718424	89	7	0.78	2.54	3.93	1	42	82
14:11:12.9792424	11	9	4.31	6.43	8.69	83	87	91
14:11:17.9866424	0	8	9.39	10.83	12.82	77	80	84
14:11:22.9940424	0	8	13.80	15.75	17.48	69	72	76
14:11:28.0014424	0	7	18.37	20.19	22.18	62	65	68
14:11:33.0088424	0	6	22.90	24.83	26.94	56	58	61
14:11:38.0162424	0	10	27.74	29.53	31.62	46	50	55
14:11:43.0236424	0	9	31.87	34.47	36.30	37	41	45
14:11:48.0310424	0	7	36.87	39.06	40.98	30	33	36
14:11:53.0384424	0	9	41.75	43.82	45.14	21	25	29
14:11:58.0458424	0	8	45.92	47.67	49.67	13	16	20
14:12:03.0532424	0	8	50.86	52.58	54.59	5	8	12
14:12:08.0606424	0	4	55.41	55.96	56.69	1	2	4
14:12:13.0680424	0	0	0.00	0.00	0.00	0	0	0

把这些数据加载到一个 Excel 图表中，队列大小的变化如下图所示。



贷款中介的智能代理统计

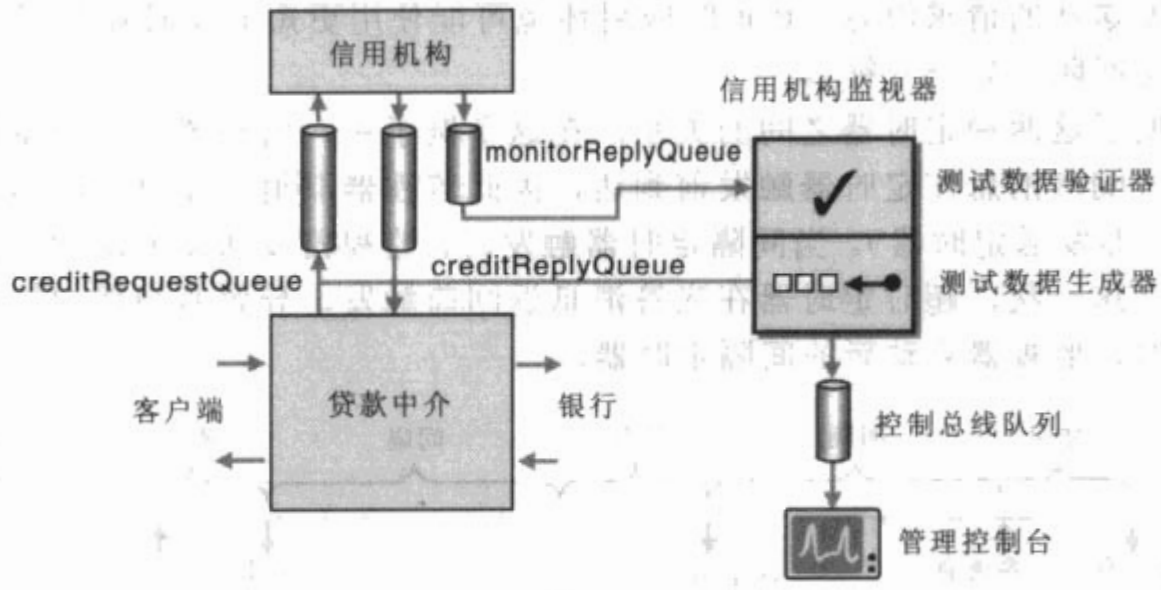
从图中可以看到，两个测试客户端对贷款中介提出了大量的请求，高峰时有 90 个请求消息等待处理。贷款中介按照每秒钟 2 个请求消息的稳定速度处理请求。由于排队的请求

消息数量很大，响应时间是很长的，高峰时几乎达到了 1 分钟。好在贷款中介能妥善地处理大量突发的请求，只不过响应的时间太长。为了提高响应时间，我们可以运行多个贷款中介实例或多个信用机构实例（第 9 章“中场演练：复合消息传递”介绍的第一个实现中，信用机构服务是瓶颈）。

12.1.4 验证信用机构的操作

管理贷款中介系统的第二个需求是监视外部信用服务的操作是否正常。贷款中介访问这个服务，以获得请求利率信息的客户的信用度。为了让银行提供准确的利率，必须有这个信息。

为了验证外部信用机构服务的操作是否正确，我们要向服务周期性地发送测试消息。由于信用机构服务支持返回地址，因此插入的测试消息不会干扰已有的消息流。我们只需为测试消息提供相应的应答通道，就可以避免采用单独的测试消息分离器（如下图所示）。



监视信用机构服务

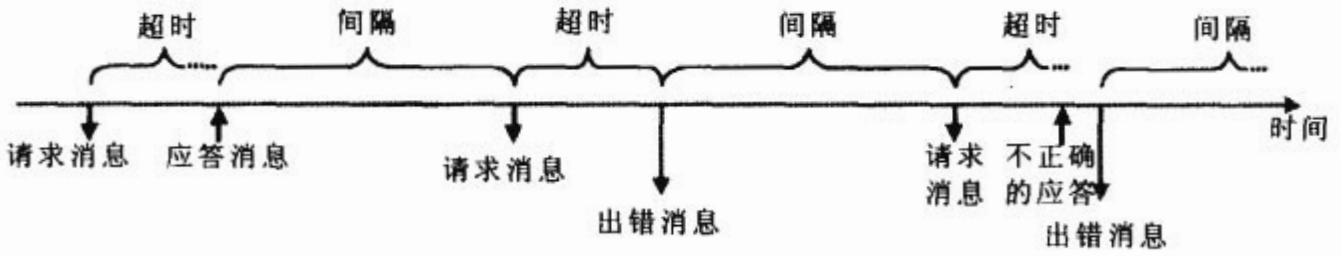
为了验证信用机构的服务是否正确，需要编写测试数据生成器和测试数据验证器。测试数据生成器创建发送给被测试服务的测试数据。信用机构的测试消息是非常简单的，它唯一需要的字段是社会安全号（SSN）。对我们的测试示例来说，使用的是一个特殊的固定 SSN，它唯一标识一个虚构人物。通过把结果数据与预先设置好的结果进行比较，就可以验证结果数据是否正确。采用这种方式，我们不但能检查是否接收到应答消息，还能验证消息的内容是否正确。在第 9 章“中场演练：复合消息传递”介绍的简单示例中，信用机构服务返回随机的结果值，而不管输入消息的 SSN 是什么。因此，我们的测试数据验证器并不检查具体的结果值，而只是验证该结果值是否在许可的范围内（例如信用度在 300～900 之间）。如果结果值超出了许可的范围（由于计算错误被设置为 0），测试数据验证器就会用消息通知管理控制台。

测试数据验证器还检查外部服务的响应时间。如果我们没能在预定的时间间隔内收到应答消息，就要向管理控制台发出警告。为了尽可能减少网络带宽的使用，测试数据验证

器只是在响应延迟或出错时才通知控制台，如果服务正常则不发送任何消息。唯一的例外是，当服务从出错状态恢复到正常状态时（重新接收到正确的应答消息），监视器要向管理控制台发送一条“服务正常”消息，以表示信用机构重新恢复正常操作。最后一点，在启动时，监视器要向控制台发送一个消息，以宣布自己的存在。通过这个消息，控制台才能“发现”所有活跃的监视器，并能显示出各个监视器的状态。

监视器的实现使用了两个不同的定时器：一个按指定的间隔发送测试消息，如果响应消息未能在设定的超时期限内到达，另一个定时器就会标记一个异常。发送定时器（Send Timer）确定最近接收到的消息或者最近的超时事件与发送下一条测试消息之间的时间间隔。超时定时器（Timeout Timer）在监视器发送请求消息时启动。如果应答消息能在指定的超时期限内到达，超时定时器将被复位，并在发送下一条请求消息时重新启动。如果监视器没有在指定的时间间隔内接收到应答消息，超时定时器就会触发超时事件，让监视器把一条出错消息发送给控制总线。然后，监视器启动新的发送定时器，在等待指定的发送间隔时间后发送新的请求消息。真正的应用环境可能使用更短的超时期限（几秒钟），而使用更长的发送间隔（如一分钟）。

下图说明了这两种定时器之间的关系。在这个例子中，监视器发送测试消息，并启动超时定时器。响应消息在定时器触发前到达，因此监视器取消了超时定时器，并启动发送间隔定时器（即发送定时器）。当间隔定时器触发时，监视器发送新的测试消息并启动新的超时定时器。这一次，超时定时器在应答消息返回前触发，导致监视器向控制总线发送一个消息。同时，监视器启动新的间隔定时器。



监视信用机构服务

监视器的具体实现只需要一个类。Monitor 类继承自介绍智能代理模式时使用的 MessageConsumer 类。这个类会配置消息进入的通道，并启动事件驱动的消费者接收消息。对于每个到来的消息，它调用了虚 ProcessMessage 方法。子类只需覆盖这个方法，增加自己的处理代码即可。

Process 方法控制 MessageConsumer 来启动消费消息。Monitor 类通过启动发送定时器进一步扩展了这个方法在基类中的实现。当这个定时器触发时，调用 OnSendTimerEvent 方法。Process 方法还向控制总线发送一条 MonitorType 类型的消息，宣告监视器的存在。

Monitor 类：发送消息

```
public override void Process()
{
    base.Process();
```

```
    sendTimer = new Timer(new TimerCallback
                    (OnSendTimerEvent), null, interval*1000, Timeout.Infinite);

    MonitorStatus status = new MonitorStatus(
              MonitorStatus.STATUS_ANNOUNCE, "Monitor On-Line", null, MonitorID);
    Console.WriteLine(status.Description);
    controlQueue.Send(status);
    lastStatus = status.Status;
}

protected void OnSendTimerEvent(Object state)
{
    CreditBureauRequest request = new CreditBureauRequest();
    request.SSN = SSN;

    Message requestMessage = new Message(request);
    requestMessage.Priority = MessagePriority.AboveNormal;
    requestMessage.ResponseQueue = inputQueue;

    Console.WriteLine(DateTime.Now.ToString() + " Sending request message");
    requestQueue.Send(requestMessage);

    correlationID = requestMessage.Id;
    timeoutTimer = new Timer(new TimerCallback(OnTimeoutEvent), null,
                             timeout*1000, Timeout.Infinite);
}
```

OnSendTimerEvent 方法创建一个新的请求消息。请求消息中的唯一参数是客户的 SSN。该方法设置一个固定的 SSN，同时还保存了消息 ID，以验证所有到来的应答消息的关联标识符。最后，它启动 timeoutTimer，如果超时期限到期并且没有接收到应答消息，则通知监视器。

这个方法设置测试消息的 Priority（优先级）属性为 AboveNormal，以保证排队的应用消息不会让服务看上去不可用。对测试消息使用更高的优先级，这样可以让消息队列先发送这些测试消息，而不是排队的应用消息。在这个例子中设置更高的消息优先级是安全的，因为测试数据生成器只是插入了很少量的测试消息。如果我们把大量高优先级的消息插入到请求通道中，就会中断应用的消息流。这就违背了管理解决方案尽可能不干扰系统的原则。

ProcessMessage 方法是 Monitor 类的核心。它实现了测试消息验证器，对到来的应答消息做出评估。在停止超时定时器后，该方法检查到来的消息是否有正确的关联标识符、消息体数据类型是否正确，以及消息体中是否包含合理的数值。如果上述测试中某一项未通过，该方法将构造一个 MonitorStatus 结构，并把它发送给控制总线通道。监视器还会跟

踪存储在 lastStatus 变量中的前一次状态。如果状态从“错误”变成“正确”，ProcessMessage 方法还要向控制总线发送一个“恢复正常”的通知。

Monitor 类：接收消息

```
protected override void ProcessMessage(Message msg)
{
    Console.WriteLine(DateTime.Now.ToString() + " Received reply message");

    if (timeoutTimer != null)
        timeoutTimer.Dispose();

    msg.Formatter = new XmlMessageFormatter(new Type[] {typeof(CreditBureauReply)});
    CreditBureauReply replyStruct;
    MonitorStatus status = new MonitorStatus();

    status.Status = MonitorStatus.STATUS_OK;
    status.Description = "No Error";
    status.ID = MonitorID;

    try
    {
        if (msg.Body is CreditBureauReply)
        {
            replyStruct = (CreditBureauReply)msg.Body;
            if (msg.CorrelationId != correlationID)
            {
                status.Status = MonitorStatus.STATUS_FAILED_CORRELATION;
                status.Description =
                    "Incoming message correlation ID does not match outgoing message ID";
            }
            else
            {
                if (replyStruct.CreditScore < 300 || replyStruct.CreditScore > 900 ||
                    replyStruct.HistoryLength < 1 || replyStruct.HistoryLength > 24)
                {
                    status.Status = MonitorStatus.STATUS_INVALID_DATA;
                    status.Description = "Credit score values out of range";
                }
            }

        }
        else
        {
            status.Status = MonitorStatus.STATUS_INVALID_FORMAT;
```

```
        status.Description = "Invalid message format";        }
}
catch (Exception e)
{
    Console.WriteLine("Exception: {0}", e.ToString());
    status.Status = MonitorStatus.STATUS_INVALID_FORMAT;
    status.Description = "Could not deserialize message body";
}

StreamReader reader = new StreamReader (msg.BodyStream);
status.MessageBody =  reader.ReadToEnd();

Console.WriteLine(status.Description);

if (status.Status != MonitorStatus.STATUS_OK ||
    (status.Status == MonitorStatus.STATUS_OK &&
     lastStatus != MonitorStatus.STATUS_OK))
{
    controlQueue.Send(status);
}

lastStatus = status.Status;
sendTimer.Dispose();
    sendTimer = new Timer(new TimerCallback(OnSendTimerEvent), null,
                          interval*1000, Timeout.Infinite);
}
```

如果在指定的时间间隔内没有消息到达，**timeoutTimer** 将调用 **OnTimeoutEvent** 方法。该方法向控制总线发送一条 **MonitorStatus** 消息，并启动一个新的发送定时器，以在定时时间结束时发送新的请求消息。

Monitor 类：超时

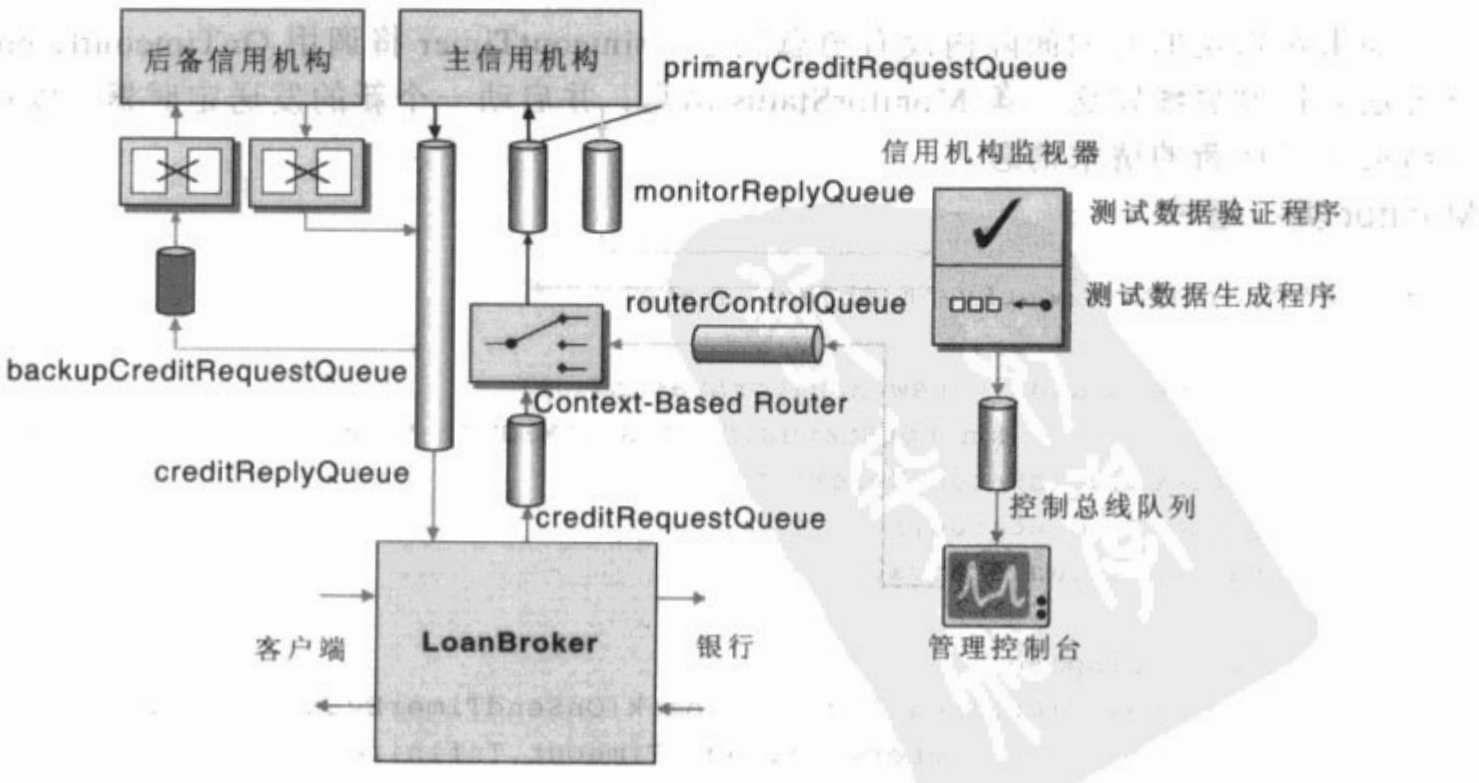
```
protected void OnTimeoutEvent(Object state)
{
    MonitorStatus status = new MonitorStatus(
                        MonitorStatus.STATUS_TIMEOUT, "Timeout", null, MonitorID);
    Console.WriteLine(status.Description);
    controlQueue.Send(status);
    lastStatus = status.Status;

    timeoutTimer.Dispose();
    sendTimer = new Timer(new TimerCallback(OnSendTimerEvent), null,
                          interval*1000, Timeout.Infinite);
}
```

12.1.5 信用机构的故障恢复

既然我们能监视外部信用机构服务的状态，为什么不使用这些数据实现一种故障恢复方案，使贷款中介在信用机构出现故障时能继续运行。值得注意的是，点对点通道已经提供了基本的故障恢复方式。当我们使用单个点对点通道上的多个竞争消费者时，如果其中一个消费者出现故障，只要其他的消费者还在运行，就不会中断消息的处理。多个消费者同时运行时，它们会分摊开销，有效地实现了一种简单的负载平衡机制。那么我们为什么还需要实现一个明确的故障恢复机制呢？使用外部服务时，可能只能使用简单的通道，而不支持竞争消费者，如建立在 HTTP 上的 SOAP。而且，我们可能不希望多个服务实现负载平衡。例如，可以与主要的服务提供商达成流量协议，如果能满足某种使用额度，就能在服务方面给我们提供充分的优惠。如果把通信量划分给两家提供商，可能反而要付出更大的成本。相反，我们可以使用低成本提供商作为主要服务提供商，只有当低成本提供商出现故障时，才切换使用其他的提供商。（许可考虑也会影响体系结构的决策，有关的精彩讨论请参见[Hohmann]。）

为了实现明确的故障恢复机制，我们要在信用机构的请求通道中插入一个消息路由器（如下图所示）。这个路由器把请求消息路由给主要的信用机构服务（图中的粗线）或辅助的信用机构服务（图中的细线）。由于辅助服务使用的消息格式与第一种服务不同，我们还为辅助服务封装了一对消息转换器。这个消息路由器是一个基于上下文的消息路由器，它由管理控制台通过控制总线控制。管理控制台从前一节设计的信用机构监视器那里获得监视数据。如果监视器报告有故障出现，管理控制台就要求消息路由器把消息流重定向到辅助的服务提供者（如下图所示）。



使用基于上下文的消息路由器实现明确的故障恢复

当请求消息流重路由给辅助的服务提供者时，监视器继续向主要的提供者发送测试消息。当监视器证实主服务的操作正确后，控制台就会命令消息路由器把请求消息重新路由给主要的服务提供者。可以再使用信用机构监视器的一个实例来监视后备信用机构服务的状态，尽管这非常容易，但这个解决方案并没有为辅助的服务提供者设置监视器。

下面来看基于上下文的消息路由器的具体实现。ContextBasedRouter 类继承自可信赖的 MessageConsumer 基类，负责处理到来的消息。ProcessMessage 方法检查变量 control 的值，并根据这个变量值把到来的消息路由给主输出通道或辅助输出通道。

ContextBasedRouter 类

```
delegate void ControlEvent(int control);

class ContextBasedRouter : MessageConsumer
{
...
    protected override void ProcessMessage(Message msg)
    {
        if (control == 0)
        {
            primaryOutputQueue.Send(msg);
        }
        else
        {
            secondaryOutputQueue.Send(msg);
        }
    }

    protected void OnControlEvent(int control)
    {
        this.control = control;
        Console.WriteLine("Control = " + control);
    }
}
```

变量 control 在 OnControlEvent 方法中设置。该方法由 ControlReceiver 类调用，这个类也继承自 MessageConsumer 类，它监听来自控制通道的消息。ContextBasedRouter 类为 ControlReceiver 提供了一个类型为 ControlEvent 的委托，接收到有数字值的控制事件时就要调用这个委托。如果你不熟悉委托，实际上这就是一种实现回调的简洁方法，而且能够做到类型安全，它不需要实现其他接口，也不需要提供函数指针（有关的细节见[Box]的介绍）。

ControlReceiver 类

```
class ControlReceiver : MessageConsumer
```

```
{
    protected ControlEvent controlEvent;

    public ControlReceiver(MessageQueue inputQueue,
                           ControlEvent controlEvent) : base (inputQueue)
    {
        this.controlEvent = controlEvent;
    }

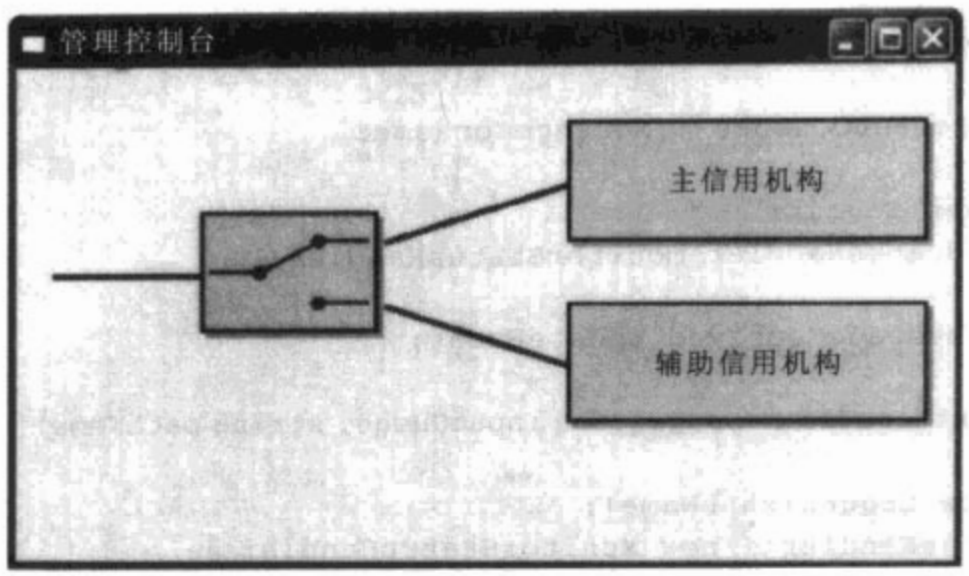
    protected override void ProcessMessage(Message msg)
    {
        String text = (string)msg.Body;
        Double resNum;

        if (Double.TryParse(text, NumberStyles.Integer,
                           NumberFormatInfo.InvariantInfo, out resNum))
        {
            int control = int.Parse(text);
            controlEvent(control);
        }
    }
}
```

12.1.6 增强管理控制台

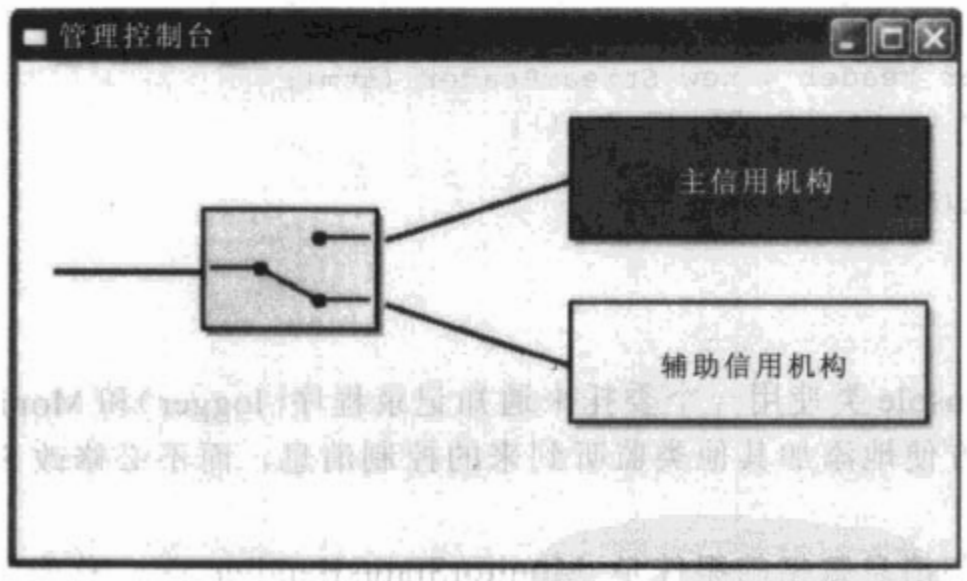
管理控制台的第一版太简单了，所以我们连代码都没有给出。它的功能只是接收消息并把消息的内容写到文件中，以便事后分析（如利用 Excel 绘制性能图）。现在我们想在管理控制台中添加更多的智能。首先，当主信用机构监视器指示主服务提供者出现故障时，管理控制台能命令基于上下文的消息路由器，把消息重路由给辅助的服务提供者。我们选择在管理控制台中实现这个功能，从而将监视器与基于上下文的消息路由器分离——从效果上看，管理控制台充当了仲裁者[GoF]。另外，把故障恢复逻辑集中实现为维护系统管理规则提供了唯一的入口。商业的管理控制台通常包含可配置的规则引擎，能根据控制总线上的事件确定相应的修正操作。

其次，我们要为管理控制台建立一个简单的用户界面，显示系统的当前状态。要想获得消息传递系统的整个视图是非常困难的，特别是消息的路径能动态地改变时会更为困难。即使只有少量的组件，也很难确定消息的流向。我们的用户界面虽然简单但是非常有用。它使用本书定义的图标语言表示组件之间的交互。目前，用户界面只显示了系统的信用机构如何实现故障恢复，它由两个服务和一个基于上下文的消息路由器组成，如下图所示。



管理控制台指出两个信用机构服务都是正常的

监视器检测到故障并命令路由器重定向消息流时，要更新用户界面来反映新的状态（如下图所示）。路由器图标给出了请求消息的新路由，主信用机构组件的颜色发生变化，以表示出现了故障。



管理控制台指出主信用机构出现故障，消息流被重路由

下面简要看一下控制台的实现代码。我们把重点放在系统管理的实现部分，而不是深入分析如何绘制漂亮的用户界面。首先，管理控制台要能从监视器组件那里获得状态消息。为了尽可能提高控制台的健壮性，我们以松耦合的方式访问消息的内容，从 XML 格式的有效数据中读取各个字段。即使个别组件要在消息格式中添加新的字段，这种方法也能保证管理控制台的正常运行。

自然，控制台类也是从 MessageConsumer 类继承而来的，因此我们只给出了构造函数和 ProcessMessage 方法的具体实现。ProcessMessage 方法只是把消息的 BodyStream 读到一个字符串变量中，然后把这个变量传递给其他组件分析。

ManagementConsole: ProcessMessage

```
public delegate void ControlMessageReceived(String body);

public class ManagementConsole : MessageConsumer
{
    protected Logger logger;
    public MonitorStatusHandler monitorStatusHandler;

    public ControlMessageReceived updateEvent;

    public ManagementConsole(MessageQueue inputQueue, string pathName) : base(inputQueue)
    {
        logger = new Logger(pathName);
        monitorStatusHandler = new MonitorStatusHandler();

        updateEvent += new ControlMessageReceived(logger.Log);
        updateEvent += new ControlMessageReceived(monitorStatusHandler.OnControlMessage);
    }

    protected override void ProcessMessage(Message m)
    {
        Stream stm = m.BodyStream;
        StreamReader reader = new StreamReader (stm);
        String body =  reader.ReadToEnd();
        updateEvent(body);
    }
    ...
}
```

ManagementConsole 类使用一个委托来通知记录程序(logger)和 MonitorStatusHandler。利用委托，我们能方便地添加其他类监听到来的控制消息，而不必修改 ProcessMessage 方法中的代码。

分析到来的控制消息数据的组件是 MonitorStatusHandler 类。首先，这个类检查包含在消息体中的 XML 文档是否有根元素<MonitorStatus>。如果有，它就把消息体加载到一个 XML 文档，从中提取出包含在 ID 和 Status 元素中的相关字段。然后它调用 updateEvent 委托，这个委托的类型是 MonitorStatusUpdate。管理控制台应用中，所有对该事件感兴趣的类都可以把回调方法添加到这个委托中，当 MonitorStatus 消息到达时，这些组件都会得到通知。所有组件提供的回调方法具体实现必须与 MonitorStatusUpdate 的签名完全相同。

MonitorStatusHandler

```
public delegate void MonitorStatusUpdate(String ID, int Status);

public class MonitorStatusHandler
{
```

```
public  MonitorStatusUpdate updateEvent;

public void OnControlMessage(String body)
{
    XmlDocument doc = new XmlDocument();
    doc.LoadXml(body);

    XmlElement root = doc.DocumentElement;
    if (root.Name == "MonitorStatus")
    {
        XmlNode statusNode = root.SelectSingleNode("Status");
        XmlNode idNode = root.SelectSingleNode("ID");

        if (idNode!= null && statusNode != null)
        {
            String msgID = idNode.InnerText;
            String msgStatus = statusNode.InnerText;
            Double resNum;
            int status = 99;

            if (Double.TryParse(msgStatus, NumberStyles.Integer,
                                NumberFormatInfo.InvariantInfo, out resNum))
            {
                status = (int)resNum;
            }
            updateEvent(msgID, status);
        }
    }
}
```

在我们的例子中，由 MonitorStatusHandler 触发 MonitorStatusUpdate 事件，监听这个事件的前两个组件是两个用户界面控件，它们分别表示用户界面中的主信用机构服务和辅助的信用机构服务。每个用户界面控件针对标识符过滤事件，所监视的各个组件都有唯一的标识符。当所监视的组件的状态发生变化时，用户界面控件会修改组件的颜色。显示界面初始化时执行下面的例程，它把两个信用机构的显示控件与管理控制台的 monitorStatusHandler 绑定。当控制台接收到监视器的状态更新消息时，根据该代码，会调用控件的 OnMonitorStatusUpdate 方法。

控制台界面的初始化

```
console = new ManagementConsole(controlBusQueue, logFileName);

primaryCreditBureauControl =
    new ComponentStatusControl("Primary Credit Bureau", "PrimaryCreditService");
primaryCreditBureauControl.Bounds =
    new Rectangle(300, 30, COMPONENT_WIDTH, COMPONENT_HEIGHT);

secondaryCreditBureauControl =
```

```
    new ComponentStatusControl("Secondary Credit Bureau", SecondaryCreditService");
secondaryCreditBureauControl.Bounds =
    new Rectangle(300, 130, COMPONENT_WIDTH, COMPONENT_HEIGHT);

console.monitorStatusHandler.updateEvent += new
    MonitorStatusUpdate(primaryCreditBureauControl.OnMonitorStatusUpdate);
console.monitorStatusHandler.updateEvent += new
    MonitorStatusUpdate(secondaryCreditBureauControl.OnMonitorStatusUpdate);
```

另一个监听 MonitorStatusUpdate 事件的组件是 FailOverHandler。该组件是一个非可视化的组件，它会分析状态消息，并确定是否进行故障恢复的切换。如果监视器的状态发生了改变(使用^操作符表示 XOR)，FailOverHandler 就向指定的命令通道发送一条命令消息。在我们的例子中，此命令通道与前面介绍的基于上下文的消息路由器连接，该路由器将把信用度请求消息重路由给其他信用机构服务提供者。

FailOverHandler 类

```
public delegate void FailOverStatusUpdate(String ID, string Command);

public class FailOverHandler
{
    ...
    public void OnMonitorStatusUpdate(String ID, int status)
    {
        if (componentID == ID)
        {

            if (IsOK(status) ^ IsOK(currentStatus))
            {
                String command = IsOK(status) ? "0" : "1";
                commandQueue.Send(command);
                currentStatus = status;
                updateEvent(ID, command);
            }
        }
    }

    protected bool IsOK(int status)
    {
        return (status == 0 || status >= 99);
    }
}
```

FailOverHandler 还调用了 updateEvent，这是一个类型为 FailOverStatusUpdate 的委托。类似于 MonitorStatusHandler，我们可以注册所有实现了这种方法的组件，从而在 FailOverHandler 改变状态时接收到更新通知。在我们的例子中，注册了可视化的

FailOverControl 来接收这些事件，当故障恢复状态发生改变时，FailOverControl 就会重新绘制。控制台用户界面的初始化例程建立了这些组件之间的连接：

控制台界面的初始化

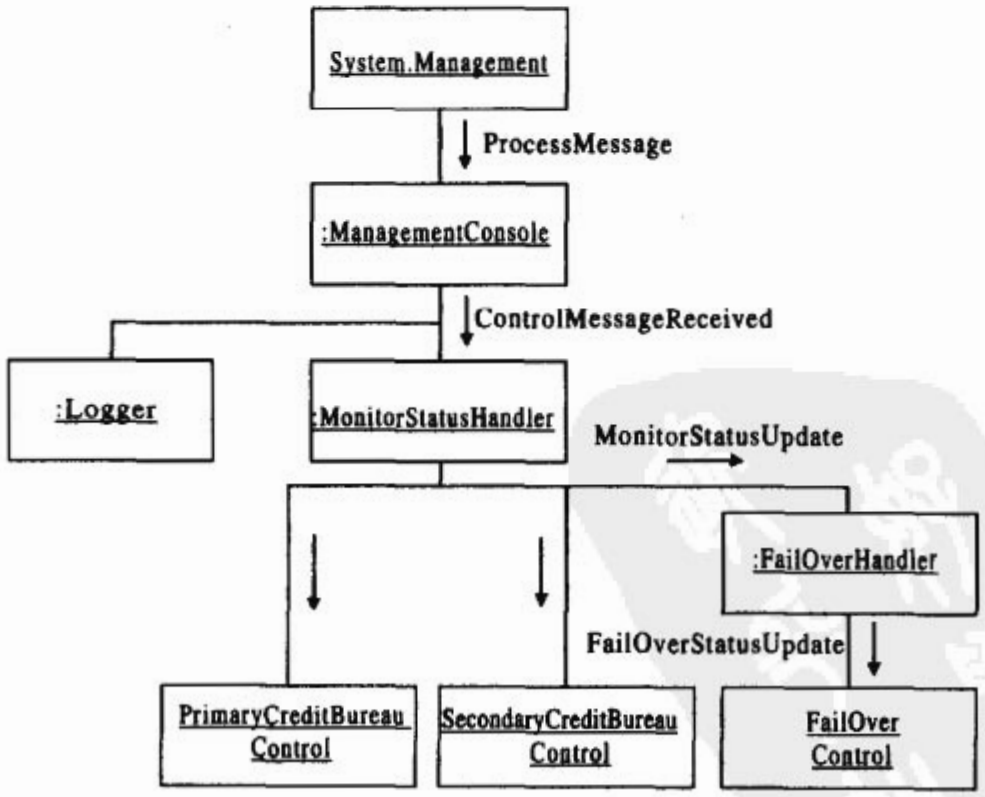
```
failOverControl = new FailOverControl("Credit Bureau Failover",
"PrimaryCreditService");
failOverControl.Bounds = new Rectangle(100, 80, ROUTER_WIDTH, COMPONENT_HEIGHT);

FailOverHandler failOverHandler =
    new FailOverHandler(commandQueue, "PrimaryCreditService");
console.monitorStatusHandler.updateEvent +=
    new MonitorStatusUpdate(failOverHandler.OnMonitorStatusUpdate);

failOverHandler.updateEvent += new
   FailOverStatusUpdate(failOverControl.OnMonitorStatusUpdate);
```

通过委托和事件把管理控制台中的各个组件连接起来，就形成了松耦合的体系结构。基于这种结构，我们就能重用各个组件，把它们重新组合成类似于管道和过滤器模式（本书开始时介绍过）的不同架构。从本质上看，使用委托向控制总线传递消息，这类似于在应用内部创建了一个发布-订购通道。由于控制总线事件从点对点通道上到达，因此必须使用单个消费者，再由它把事件发布给应用中所有感兴趣的订购者。

下面的协作图说明了事件如何在管理控制台的每个组件之间传播。



管理控制台内部的事件传播

使用用户界面控制台把各个组件之间的消息流以可视化的形式表示出来，这就成为一种强大的系统管理工具。某些开发商提供了开发套件，允许设计人员以可视化的方式安排

组件，并把组件的输入和输出端口连接起来，形成分布式消息流应用。例如，Fiorano 的 Tifosi 产品（http://www.fiorano.com）提供了 Distributed Applications Composer，利用这个工具，可以在一个图形用户界面中设计分布式解决方案，尽管每个组件可以在不同的计算机或平台上执行。这个工具使用控制总线把所有的分布式组件连接起来，形成一个集中的管理和监视控制台。

在我们给出的简单示例中，管理控制台要明确地指定各个组件之间的可视化连接。例如，要在故障恢复路由器和表示信用机构服务的图标之间画一条线。许多集成工具允许用户从一开始就使用 GUI 来设计解决方案。这种方法有利于使用相同的图形化设计来显示解决方案的状态。

或者，也可以分析已有的消息传递解决方案中的消息流，创建该系统的图形化表示。完成这种分析有两种基本方法：静态方法和动态方法。静态分析方法检查每个组件发布和订购的通道。如果某组件发布的通道与另一个组件订购的通道相同，图形化工具就在这两个组件之间绘制一条连线。许多 EAI 工具套件（如 TIBCO ActiveEnterprise）把这类信息存储在一个集中的知识库中，使这种分析变得更加简单。第二种方法使用动态分析方法，它检查流经系统的每个消息，并根据到达特定组件的消息源反向推出组件之间的连接关系。如果系统中的消息包含消息历史，就能极大地简化操作。即使系统中的消息不包含消息历史，如果每个消息包含了表示消息发送者的字段（许多系统出于认证的目的都包含这种字段），也能由此重新构造出消息流。

12.1.7 此例的局限性

为了能用一章的篇幅介绍这样一个完整的例子，我们不得不做一些简化的假设。例如，我们的故障恢复机制没有考虑主信用机构服务失败时已经排队的消息该如何处理——这些消息要等到服务重启后才能继续处理。贷款中介之所以能继续工作，是因为它能把到来的响应消息与应答消息关联起来，但是这些“阻塞”的利率请求消息只有等到主信用机构服务重新在线后才能处理。为了提高故障恢复时的响应速度，应该实现重传功能，允许贷款中介重新发送在故障服务队列中排队的请求消息。另一种方法是让故障恢复路由器把最近一次证实服务功能正常之后到达的所有请求消息存储起来。如果检测到服务失败，路由器可以重传存储的这些消息，因为其中部分消息可能没有得到正确处理。这种方法会导致重复的请求消息（以及重复的应答消息），但是由于信用机构服务和贷款中介都是幂等接收者，因此重复的消息不会引起任何问题，重复的应答消息将被忽略。

这个例子只演示了很少一部分系统管理功能，利用上一章介绍的模式还能实现更多的功能。例如，我们可以监视经过所有组件的消息流、设置性能阈值、让每个组件发送心跳消息等。实际上，要为分布式消息传递解决方案增加可靠的系统管理功能，无论从设计上还是实现上，都要像对待原解决方案一样付出同样（甚至更多）的努力。

第 13 章

实战集成模式

13.1 案例研究：债券定价系统

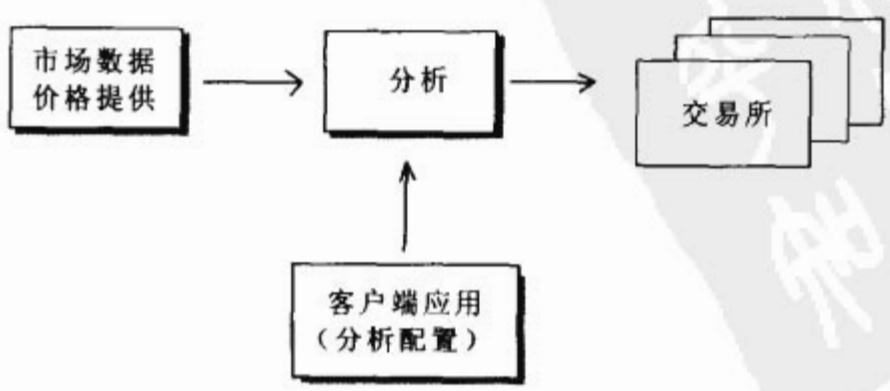
作者：Jonathan Simon

要想掌握大量的模式或一种模式语言，为此需要付出很长时间的努力。模式是可重用思想的抽象体现。一般来说，由于模式所固有的通用性，这使得模式相当有用，但同时也意味着模式很难掌握。有时，理解模式的最好办法还是通过真实的例子来学习。真实的例子是指，并非按照假设发生的场景来设计，而是按照实际发生和将要发生的场景设计。

本章利用了一个发现过程，并采用多种模式来解决问题。我们介绍的系统是一个债券交易系统，从最初设计概念的形成到产品的最终交付，我整整参与了两年时间。其中包括选择模式的决策过程，以及决定如何组合和调整模式以满足系统的需要。做出这些决策时都考虑到了现实系统所带来的影响，包括业务需求、客户端的决定、体系结构和技术方面的需求，以及遗留系统的集成等等。之所以介绍这个具体的系统，就是为了通过实际的应用使你对模式本身有更透彻的理解。

13.1.1 构建系统

一家主要的华尔街投资银行准备建立一个债券定价系统，目的是为了提高其债券交易部门的工作效率。目前，债券交易公司必须把大量债券的价格发送给几个不同的交易所，每个交易所都提供了自己的用户界面。系统的目的是想尽可能减少债券定价的细节，并提供专门针对债券市场的高级分析功能，而且这些都在一个封装好的用户界面中提供。这就意味着需要通过各种通信协议实现多个组件的集成和通信。系统的高层数据流图如下所示。



高层数据流图

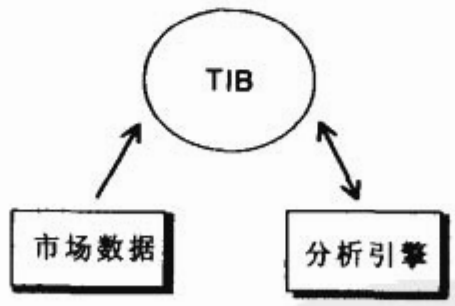
首先，市场数据进入系统。市场数据是指与债券的价格以及其他特性相关的数据，表示哪些人会在自由市场上购买和售出债券。市场数据将立即被发送给分析引擎，由它修改数据。分析是指一些数学函数，财经应用要使用这些数学函数来修改债券的价格和其他属性。这些函数都是通用函数，它们使用输入变量来计算特定债券的价格。客户端应用在各个交易商的桌面上运行，将针对各个交易商分别配置分析引擎，对各个待定价债券的分析进行控制。一旦对市场数据完成分析，修改过的数据将发送给各个交易所，其他公司的交易商就能购买或售出债券了。

13.1.2 使用模式的体系结构

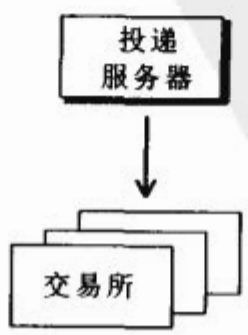
通过对这个系统中数据流图的简要介绍，我们可以着手处理设计过程中遇到的一些体系机构方面的问题。首先来看看我们已经了解些什么。交易商要求应用既能在 Windows NT 平台运行，也能在 Solaris 工作站上运行，而且要很快就做出响应。因此，我们决定把客户端的应用实现为 Java 胖客户，因为它是平台无关的，并且能为用户输入和市场数据提供快速的响应。在服务器一端，我们将继续使用系统已有的 C++组件。市场数据组件与 TIBCO 信息总线（TIBCO Information Bus，TIB）消息传递基础设施通信。

我们要继承以下组件：

- **市场数据价格供应服务器**：把到来的市场数据发布给 TIB。
- **分析引擎**：对到来的市场数据进行分析，并把修改后的市场数据广播给 TIB。
- **投递服务器**：完成与交易所的所有通信。交易所是银行无法控制的第三方组件。



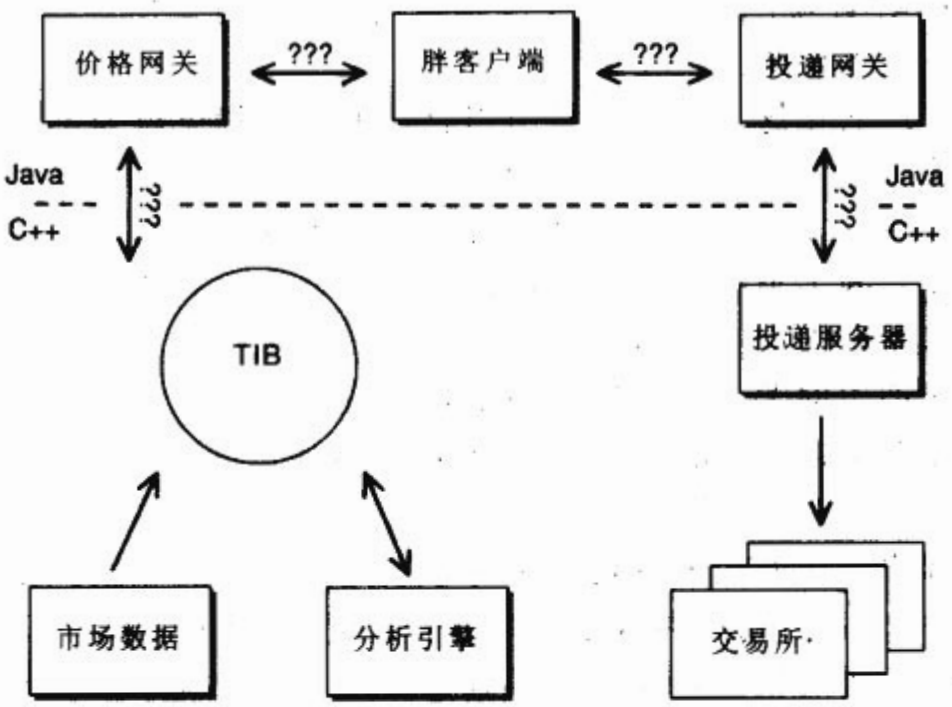
遗留的市场数据子系统



遗留的投递子系统

我们要确定不同的子系统（Java 胖客户端、市场数据和投递子系统）如何通信。可以

让胖客户端直接与遗留的服务器通信，但是这样做需要在客户端实现过多的业务逻辑。或者，可以建立一对与遗留服务器通信的 Java 网关：针对市场数据的定价网关和用于向交易所发送价格的投递网关。这样就能把与这些子系统相关的业务逻辑很好地封装起来。系统中当前有的组件如下图所示。标记为“???”的连接表示我们还不能确定这些组件是如何通信的。



系统及其组件

第一个通信问题是如何把 Java 胖客户端与两个 Java 网关组件集成起来，实现数据的交换。我们来回顾一下本书介绍的四种集成方式：文件传输、共享数据库、远程过程调用和消息传递。显然不能使用共享数据库，因为我们想在客户端与数据库之间创建一个抽象层，因此不希望客户端有访问数据库的代码。同样，采用文件传输也不适合，因为要把当前的价格发送给交易所，延迟必须极小，而文件传输的延迟非常大。我们只能在远程过程调用和消息传递之间做出选择。

Java 平台为远程过程调用和消息传递提供了内置的支持。RPC 方式的集成可以使用远程方法调用（Remote Method Invocation，RMI）、CORBA 或企业 JavaBean（EJB）实现。Java 消息传递服务（Java Messaging Service，JMS）为消息传递方式的集成提供了公共 API。这两种集成方式都很容易用 Java 实现。

因此，到底采用哪种集成方式更适合这个项目呢？远程过程调用还是消息传递？系统只有一个定价网关实例和一个投递网关实例，但是经常会有多个胖客户端同时连接这些服务（每个债券交易商对应一个胖客户端，多个交易商可能在一个特定的时刻同时登录）。此外，银行希望这是一个通用的定价系统，能在其他应用中利用。因此，除了胖客户端的数量未知，会有多少应用使用来自网关的定价数据也是个未知数。

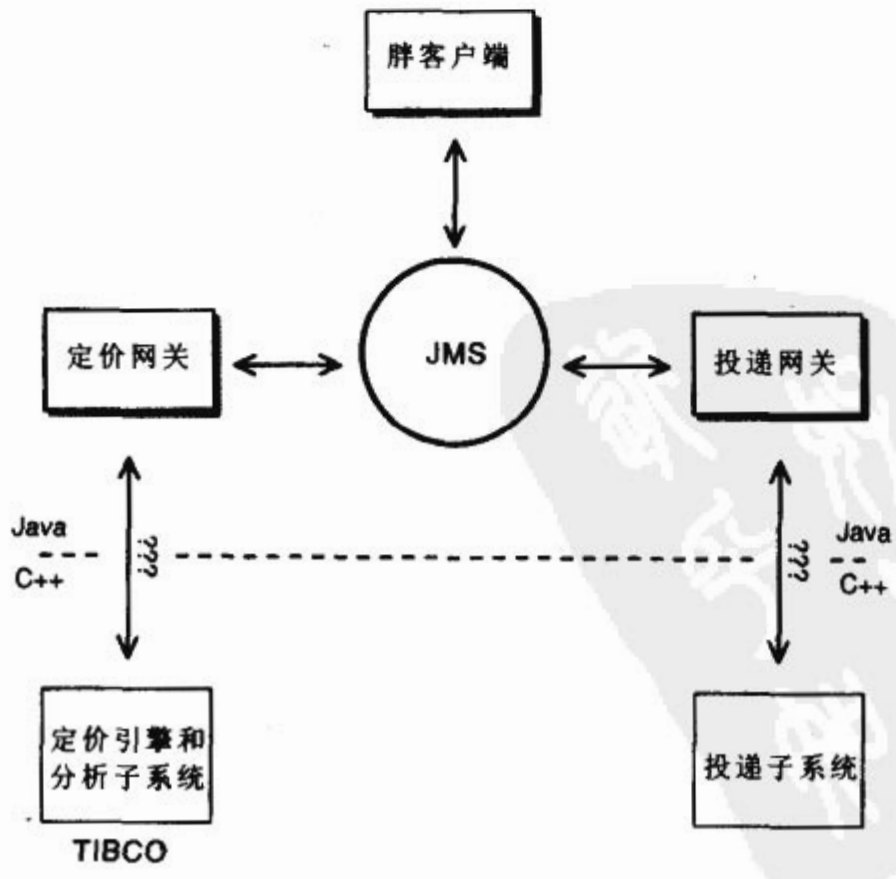
胖客户端（或其他使用定价数据的应用）可以很方便地使用 RPC 向网关发出调用，获取定价数据并调用相应的处理。但是，定价数据会不断地被发布，而一个客户端只对某种

数据感兴趣，因此要想把相关的数据及时发送给正确的客户端是非常困难的。客户端可以轮询网关，但是这种做法会产生大量的开销。更好的方法是当客户端准备好时，由网关把数据立即发送给它。但是这种方法需要网关记录哪个客户端是活跃的，以及它需要哪些特定的数据。然后，当有新的数据价格产生时（每秒钟可能出现多次），网关必须调用各个感兴趣的客户端的 RPC，把数据传递给这些客户端。理想情况下，所有的客户端应该同时得到通知，因此每个 RPC 必须在各自的并发线程中调用。虽然采用这种方法是可行的，但是很快就会使系统变得相当复杂。

消息传递能极大地简化这个问题。采用消息传递，我们可以为不同类型的定价数据定义不同的通道。然后，当网关获得新的数据时，它将向相应数据类型的发布-订购通道发送一条包含该数据的消息。在这期间，对某类数据感兴趣的所有客户端将监听对应该类型的通道。采用这种方式，网关很容易把新数据发送给所有感兴趣的客户端，而不必知道监听者应用有多少，也不用知道这些应用是什么。

客户端仍然需要调用网关中的方法。由于目前只有两个网关，客户端可以在进行同步方法调用时阻塞，客户端到网关的调用能使用 RPC 方便地实现。但是，既然我们已经使用消息传递来完成网关到客户端的通信，也可以采用消息来实现客户端到网关的通信。

因此，网关与客户端之间的所有通信都可以采用消息传递完成。由于所有的组件都是用 Java 编写的，很自然地，可以将 JMS 作为消息传递系统的选择。只要创建一个消息总线或一个体系结构，只需对消息传递基础设施做很少的修改或不做任何修改，就能够把将来的系统与当前的系统集成起来。采用这种方式，这个应用的业务功能就能很方便地由银行开发的其他应用使用。



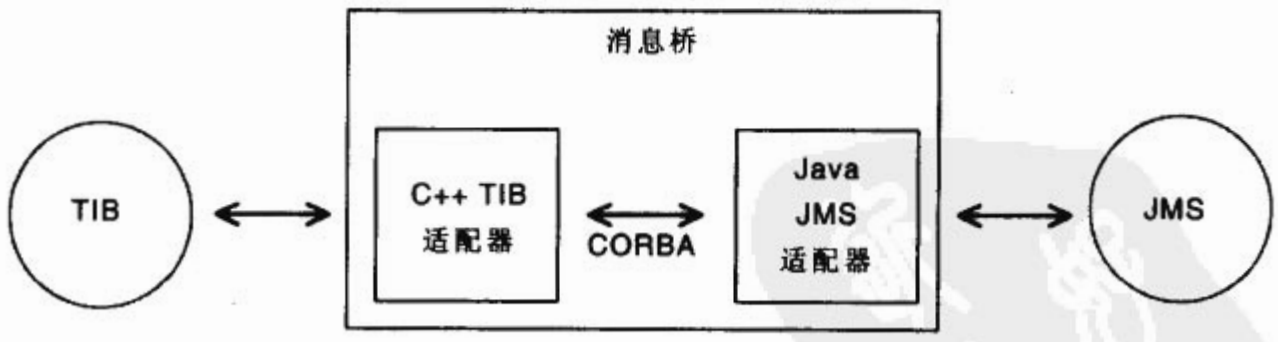
使用 JMS 实现 Java 组件之间的通信

JMS 只是一个规范，而我们需要确定采用哪一种 JMS 兼容的消息传递系统。这里之所以决定使用 IBM MQSeries JMS，主要因为这家银行就像一家“IBM 商店”，它使用了 WebSphere 应用服务器和其他许多 IBM 产品。既然已经有了一个支持基础设施，而且有了该产品的站点许可，所以我们将使用 MQSeries。

下一个问题是如何把 MQSeries 消息传递系统与独立的 C++投递服务器、基于 TIBCO 的市场数据和分析引擎服务器连接起来。我们需要为 MQSeries 消费者提供访问 TIB 消息的方法。但是如何访问呢？或许可以使用消息转换器模式把 TIB 消息转换为 MQSeries 消息。尽管 MQSeries 的 C++客户端可以充当消息转换器，但是这样使用会牺牲 JMS 服务器的独立性。虽然 TIBCO 确实提供了 Java API，但是客户架构师和管理员拒绝使用。因此，采用消息转换器是不可行的。

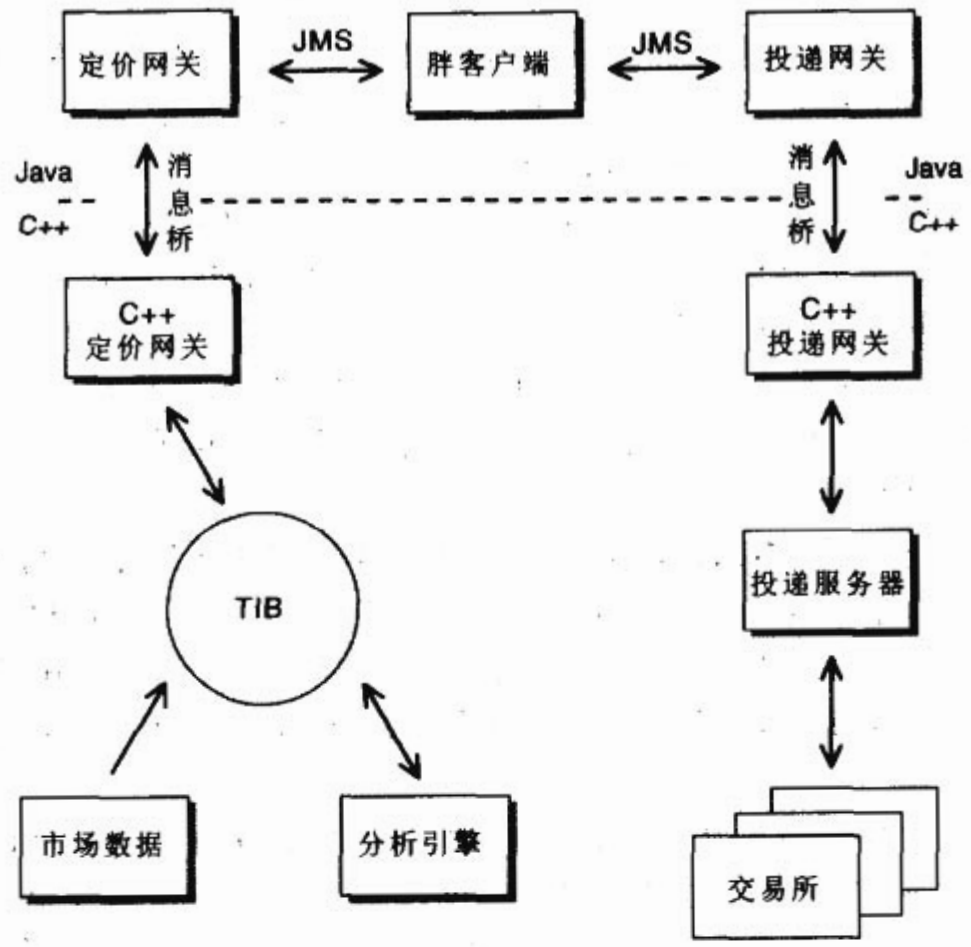
要建立从 TIB 服务器到 MQSeries 服务器的连接桥，需要实现 C++与 Java 之间的通信。可以使用 CORBA，但是如何把消息传递系统连接起来呢？进一步观察消息转换器模式可以发现，消息转换器在使用通信协议方面与通道适配器相关。通道适配器的核心是把非消息传递系统与消息传递系统连接起来。连接两个消息传递系统的一对通道适配器构成了消息传递桥。

消息传递桥的目的是把消息从一个消息传递系统传输到另一个消息传递系统。确切地说，要实现 Java 语言到 C++语言之间的通信。可以结合使用通道适配器和 CORBA 来实现跨语言的消息传递桥。为此要建立两个轻量级的通道适配器服务器，一个用 C++语言编写，用于管理与 TIB 的通信，另一个用 Java 编写，用于管理与 JMS 的通信。这两个通道适配器本身也是消息端点，要通过 CORBA 进行相互通信。就像选择 MQSeries 一样，我们使用的是 CORBA 而不是 JNI，因为 JNI 是一个公司标准。在看起来互不兼容的消息传递系统和不同的语言之间，消息传递桥可以实现有效的模拟消息转换。



使用通道适配器实现消息传递桥

下图给出了当前系统的设计方案，包括网关和其他组件，这是一个很好的模式应用实例。我们把两个通道适配器与一个非消息传递协议相结合，实现了消息传递桥模式，这样就用一种模式有效地实现了另一种模式。另外，我们修改了通道适配器的上下文，即用一种非消息传递的跨语言转换协议来连接两个消息传递系统，而不是把一个消息传递系统连接到一个非消息传递系统。



用通道适配器实现的当前系统

13.1.3 构造通道

使用模式的关键是，不仅要知道什么时候使用哪个模式，还要知道如何最有效地使用一个模式。每种模式的具体实现必须考虑到技术平台的细节以及其他设计标准。这一节同样采用发现过程，在这样一个特定的上下文中（即市场数据服务器与分析引擎之间进行通信）寻找使用发布-订购通道的最有效方法。

实时市场数据来自于市场数据供应系统，这是一个 C++服务器，它把市场数据广播给 TIB。这个市场数据供应系统为每种要发布价格的债券提供了不同的发布-订购通道。这种做法看起来有些过分，因为每种债券都需要有自己的新通道。但是问题并不太严重，因为你不必在 TIBCO 中真正创建通道。通道以一组层次结构的主题名[称为主体（Subject）]来引用。TIBCO 服务器再按主体过滤一个消息流，把各个唯一的主体发送给一个虚通道，这样就能得到非常轻量级的消息通道。

我们可以创建一个系统，此系统只在很少的通道上发布消息，订购者只监听自己感兴趣的价格。这就要求订购者使用消息过滤器或选择性消费者，从整个数据流中过滤出感兴趣的债券价格，并决定是否对接收到的每个消息进行处理。假设市场数据发布到专用的债券通道上，订购者可以注册一系列债券价格的更新事件。这样一来，订购者只要有选择地订购通道，只接收它感兴趣的债券更新消息，就可以有效地进行“过滤”，而不是在接收到

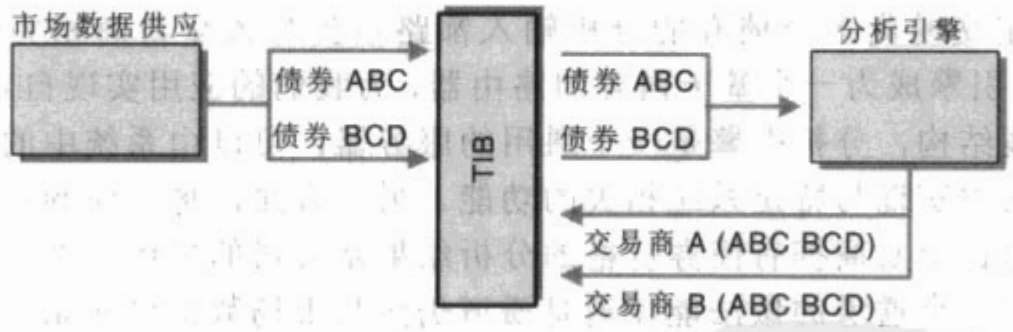
消息后才决定如何操作。值得注意的是，尽管可以使用多个通道来避免过滤，但这并不是消息传递通道的标准使用方法。不过，使用 TIBCO 技术时，我们要确定到底是实现自己的过滤器，还是使用 TIBCO 内置的通道过滤特性，而不是确定是否使用很多通道。

下一个需要设计的组件是分析引擎，这也是一个 C++/TIB 服务器，负责修改市场数据，并把修改后的数据再次广播给 TIB。尽管这个组件的设计超出了我们要介绍的范围，不属于 Java/JMS 开发，但是，由于我们是分析引擎的主要客户，因此需要与 C++小组紧密协作来完成这个组件的设计。目前的问题是要找出一种合适的通道结构，能把修改后的新市场数据最有效地重新进行广播。

针对每个债券，已经从市场数据价格供应组件继承得到了专门的消息通道，因此，按逻辑应该把修改后的市场数据重新广播到债券的专用消息通道上。但是，由于修改债券价格的分析引擎与具体的交易商相关，采用这种方法是不行的。如果我们把修改后的数据重新广播到债券的消息通道上，就会用交易商特定的数据替换通用的市场数据，这就会破坏数据的完整性。不过，可以为交易商特定的市场数据设置不同的消息类型，以防止对数据完整性的破坏。但是客户端必须实现自己的过滤器，才能把自己的消息与其他交易商的消息区分开。另外，这种做法还会大量增加订购者接收的消息，从而带来不必要的负担。

解决的方法有两个：

1. **每个交易商对应一个通道**：每个交易商都为修改后的市场数据提供专用的通道。采用这种方法，原始的市场数据保持不变，每个交易商应用可以监听自己的专用消息通道，接收修改后的价格更新消息。

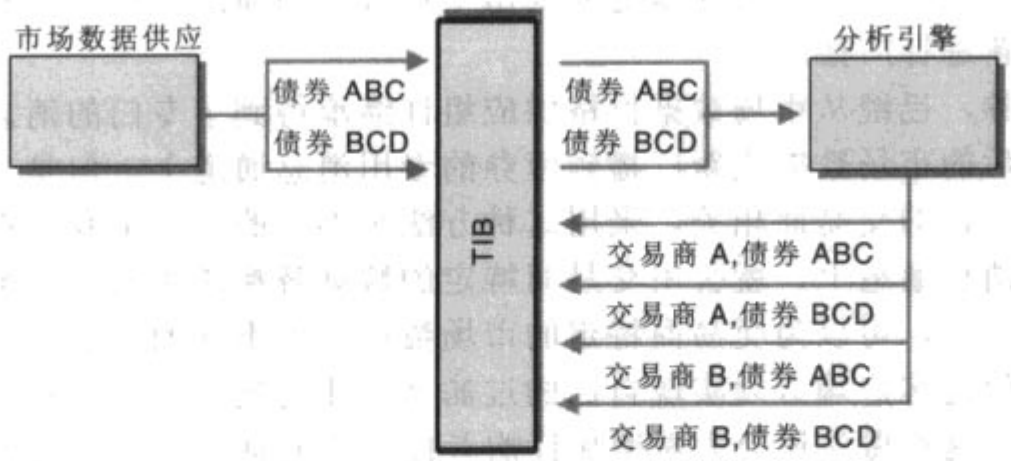


每个交易商对应一个通道

2. **每个交易商的每个债券对应一个通道**：为每个交易商的每个债券创建一个消息通道，专门用于传递该债券的修改后市场数据。例如，债券 ABC 的市场数据将发布到“债券 ABC”通道上，那么针对交易商 A 的修改后市场数据将被发送到“交易商 A 的债券 ABC”通道上，针对交易商 B 的修改后市场数据将被发送到“交易商 B 的债券 ABC”通道上，以此类推。

上述各种方法都有自己的优点和不足。例如，如果每个债券都有一个专用的通道，这种方法会使用非常多的消息通道。在最糟糕的情况下，消息通道的数量将等于债券的总数乘以交易商的个数。对于要创建的通道，可以得出通道数的上限，因为我们知道只有大约 20 家交易商，并且要定价的债券不会超过 200 个。因此，通道数上界不会超过 10,000，与市场数据供应组件使用的 100,000 个消息通道相比，这还不算什么。而且，由于我们使用

了 TIB，消息通道的开销都很低，因此消息通道很多并不是一个严重的问题。但是，这么多的消息通道会给管理带来困难。每增加一个债券，就要为各个交易商维护相应的通道，在相当动态的系统中这会造成严重的问题。但是，我们的系统基本上是静态的。它还提供了自动管理消息通道的基础设施，并加上使用类似方法继承的遗留组件体系结构，这些就减轻了这种方法的不足。但是这并不是说可以无节制地使用消息通道，而是说在合理的情况下可以使用大量消息通道实现体系结构设计。



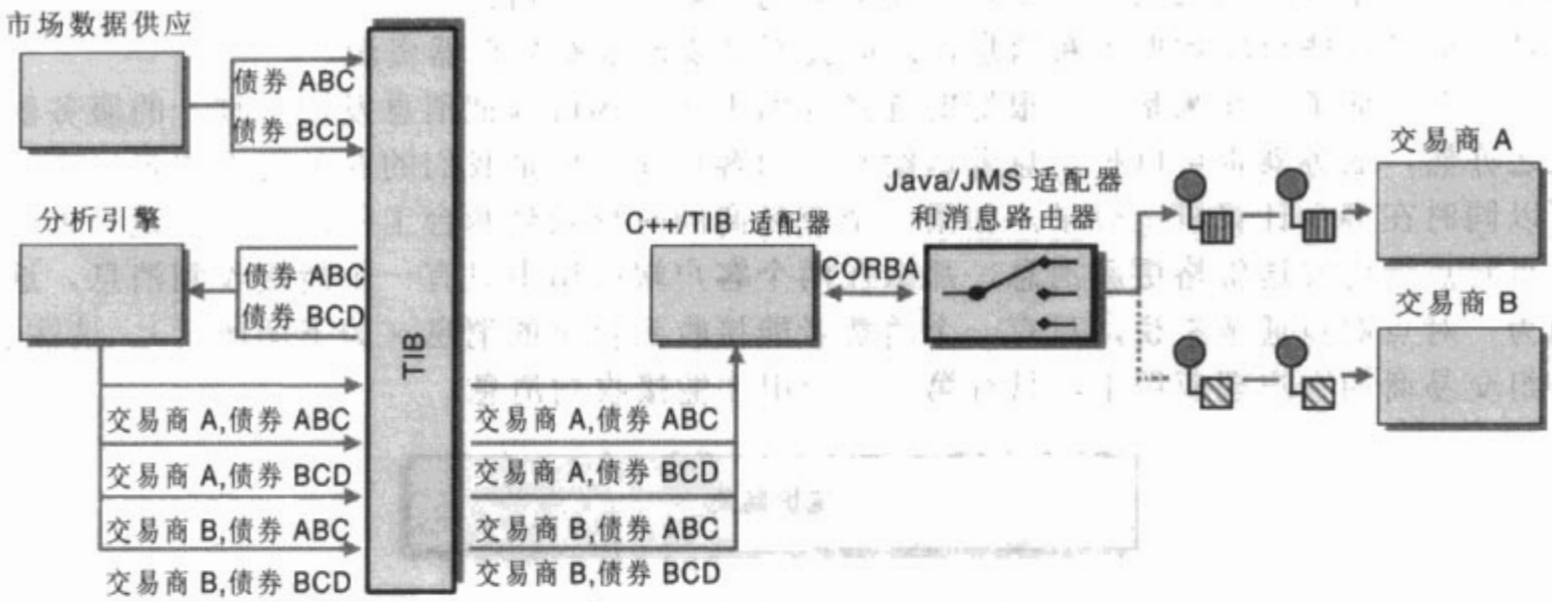
每个交易商的每个债券对应一个通道

对此还有一个原因，可以归结为在哪里实现路由逻辑。如果我们采用的做法是为每个交易商建立一个通道，分析引擎就要实现适当的逻辑来对输入和输出通道分组，因为来自分析引擎的输入通道分别对应各个债券，而输出的消息通道分别对应各个交易商，因此需要分析引擎把特定交易商多个债券的分析输入都路由到与该交易商相关的输出消息通道上。这样就使分析引擎成为一个基于内容的路由器，为我们的应用实现自定义的路由逻辑。

按照消息总线结构，分析引擎是一个通用的服务器，可以由系统中的多个组件使用，因此我们不想在其中实现与特定系统相关的功能。另一方面，每个债券对应一个通道的做法是可行的，因为让交易商拥有债券价格的分析结果是公司的惯例。增加更多的消息通道时，每个债券对应一个通道的做法能使消息通道仍然与市场数据供应系统分离。在消息到达客户端之前，需要采用基于内容的路由器组合这些通道，使得通道数量不那么可观。我们不希望在交易商桌面上运行的客户端应用监听成千甚至上万个消息通道。现在的问题是，在哪里放置基于内容的路由器。可以简单地采用 C++/TIB 通道适配器，把所有消息都转发给一个消息通道上的定价网关。但是这种做法不好，原因有两个：这样就要在 C++和 Java 之间划分业务逻辑；而且原本 TIB 能提供不同的消息通道，从而避免数据流过滤，如果采用了这种做法，这个优点就会丧失。通过分析我们的 Java 组件，可以把基于内容的路由器放在定价网关中，或者在定价网关与客户端之间创建一个中间组件。

从理论上讲，如果在到达客户端的整个路径上坚持采用每个债券对应一个消息通道的做法，那么定价网关就会采用与分析引擎一样的通道结构重新广播定价信息。这意味着在 JMS 中会重复有所有债券专用的 TIB 通道。即使我们在定价网关和客户端之间创建了一个中间组件，定价网关仍然要重复 JMS 中的所有通道。不过，如果在定价网关中直接实现路

由逻辑，就可以避免重复大量的 JMS 通道——允许按照每个交易商一个通道的方式创建很少的通道。定价网关会利用 C++/TIB 通道适配器，把自己注册为系统中各个交易商的各个债券的消费者。然后，定价网关只把与特定交易商相关的消息转发给各个指定的客户端。采用这种方法，在 JMS 一端只使用了少量的消息通道，同时还尽可能利用了 TIB 端通道分离带来的好处。



到客户端的整个市场数据流

对消息通道布局的讨论很好地反映了集成模式的重要性。我们的目的是得出如何更有效地使用消息通道。只是使用了模式还不够。还必须考虑以最好的方式进行实现，并集成到你的系统中，解决现有的问题。另外，这个例子也说明了商业活动带来的影响。如果我们能在任何一个组件中实现业务逻辑，就可以采用每个交易商一个通道的做法，利用更少的通道实现更简单的解决方案。

13.1.4 选择消息通道

既然我们已经掌握了 Java/JMS 组件与 C++/TIBCO 组件之间的通信机制，并了解了一些消息通道构造方法，下面就要确定 Java 组件使用哪种类型的 JMS 消息通道来进行通信。在选择各种 JMS 消息通道之前，首先来看看系统的高层消息流图。与客户端通信的网关有两个（定价网关和投递网关）。市场数据从定价网关流向客户端，再由客户端流向投递网关。客户端应用向定价网关发送消息，来改变应用到各个债券的分析引擎。投递网关还把消息发送给客户端应用，并把价格更新的状态信息传递给不同的交易所。

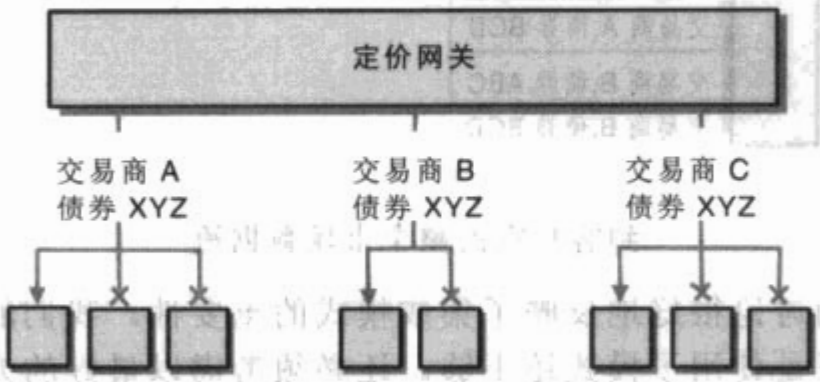


系统消息流

JMS 规范描述了两种类型的消息通道：Queue（点对点通道）和 Topic（发布-订购通道）。回顾使用发布-订购通道的例子，其目的是让所有感兴趣的消费者都能接收到消息，而使用点对点通道的目的是确保只有符合条件的消费者才能接收到特定的消息。

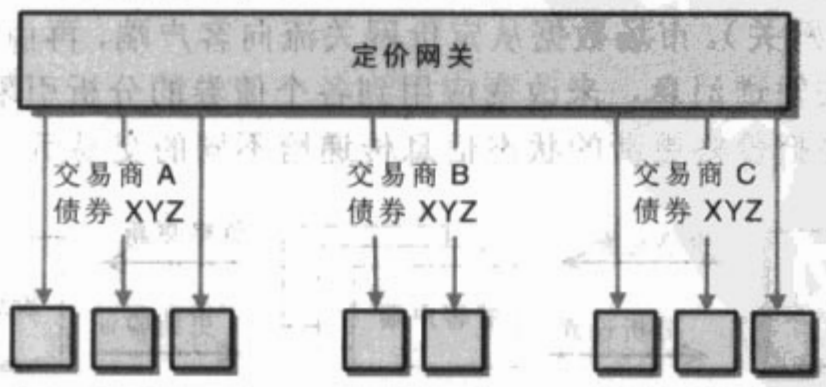
许多系统只是把消息广播给所有的客户端应用，让每个客户端应用自己决定是否处理特定的消息。对我们这个应用来说，不能采用这种方法，因为这会把大量的市场数据消息发送给每一个客户端应用。如果把市场数据的更新消息广播给无关的交易商，它们就会确定是否处理这些市场数据更新消息，因而会不必要地浪费处理器资源。

点对点通道看起来是一个很好的选择，因为客户端可以把消息发送给唯一的服务器，反之亦然，服务器也可以把消息发送给唯一的客户端。但是我们的业务需求规定，交易商可以同时在多台计算机上登录，如果一个交易商同时登录到两台工作站上，但是只采用一个点对点通道发送价格更新消息，那么在两个客户端应用中只有一个能接收到消息。这是因为，对点对点通道来说，只有一个消费者能接收到特定的消息（如下图所示）。请注意，一组交易商的客户端应用中，只有第一个应用才能接收到消息。



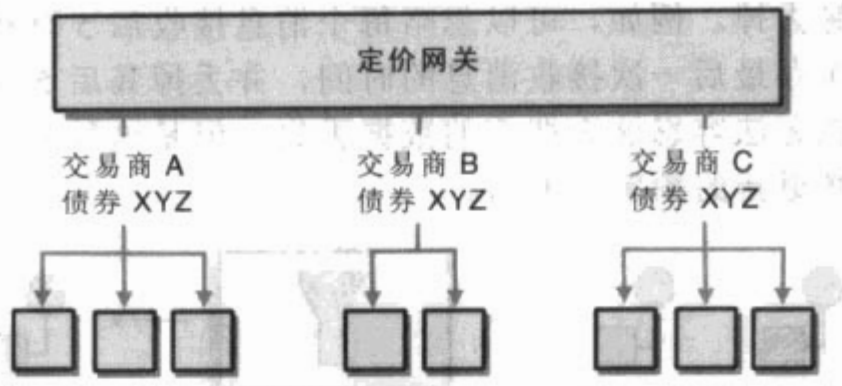
用于价格更新的点对点消息传递

使用接收表能解决这个问题，它能把消息发布给接收表中所有感兴趣的接收者，而且保证只有接收表中的客户才能接收到消息。使用这种模式，系统可以创建接收表，其中包括与各个交易商相关的所有客户端应用实例。发送与一个特定交易商相关的消息等于把消息发送给接收表中的每个应用。这样就能保证与特定交易商相关的所有客户端应用实例均能接收到消息。采用这种方法的不足在于，它需要大量实现逻辑来管理接收者以及分配消息。



用于价格更新的接收表

即使能采用点对点通道，我们也要看看是否还有更好的方法。如果使用发布-订购通道，系统可以把消息广播到与交易商相关的通道上，而不是与客户端应用相关的通道上。采用这种方法，处理某个交易商消息的所有客户端应用都会接收和处理消息（如下图所示）。



用于价格更新的发布-订购消息传递

使用发布-订购通道的不足在于，服务器组件再不能保证消息处理的唯一性。有可能建立服务器组件的多个实例，让每个实例处理相同的消息，这就可能发出非法的价格。

再回头看看系统消息流，每个消息通道只能提供单向通信。服务器到客户端的通信采用发布-订购通道即可，而客户端到服务器的通信却不能采用发布-订购通道，而只能采用点对点的通道。同样地，服务器到客户端的通信也不能采用点对点的通道。由于没有必要在两个方向上使用相同的通道，因此每个消息通道都是单向的。客户端到服务器的通信将用点对点通道实现，而服务器到客户端的通信将用发布-订购通道实现。使用这种消息通道组合，系统既能获得利用点对点消息传递与服务器直接通信的优点，又能得到采用发布-订购通道进行组播通信的优点，而同时避免它们的缺点。



有不同通道类型的消息流

13.1.5 采用模式解决问题

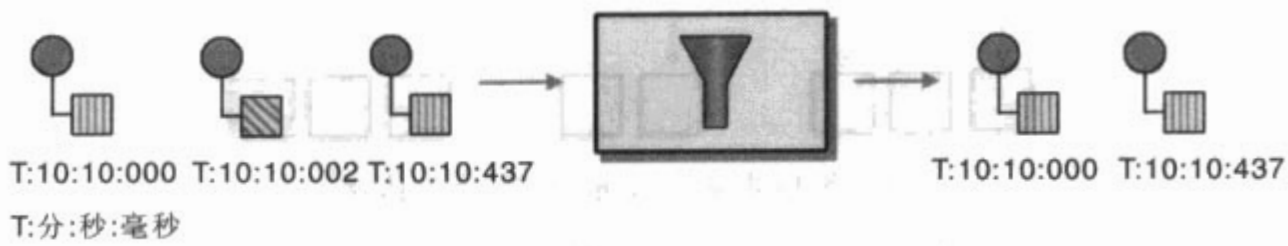
模式是工具，模式集合构成了工具箱。这些模式有助于解决问题。有人认为模式只是在设计期间有用。如果与工具箱做对照，这就像是说工具只在建造房屋时有用，在修理房屋时没用一样。实际情况是，如果利用得好，模式在整个项目期间都是有用的。与前一节一样，在后面的几节中我们还会使用同样的模式探索过程来解决目前系统中存在的问题。

13.1.6 快速显示市场数据更新

当接收某个债券的新的市场数据时，交易商要更新表格单元，明确指示出发生的变化。Java客户端接收到带有新数据的消息，然后触发客户端数据的缓存更新，并把数据最终显示在表格中。问题是，更新的频率非常快，GUI线程栈将变得不堪重负，并最终导致客户端的冻结，因为它无法响应用户交互。假设我们优化了显示过程，并强调通过更新过程的消息数据流。

通过检查性能数据，可以发现客户端应用每秒钟要接收到多个更新消息，而且两条更新消息之间的间隔只有毫秒级。有两种模式可能有助于减慢消息流：聚合器和消息过滤器。

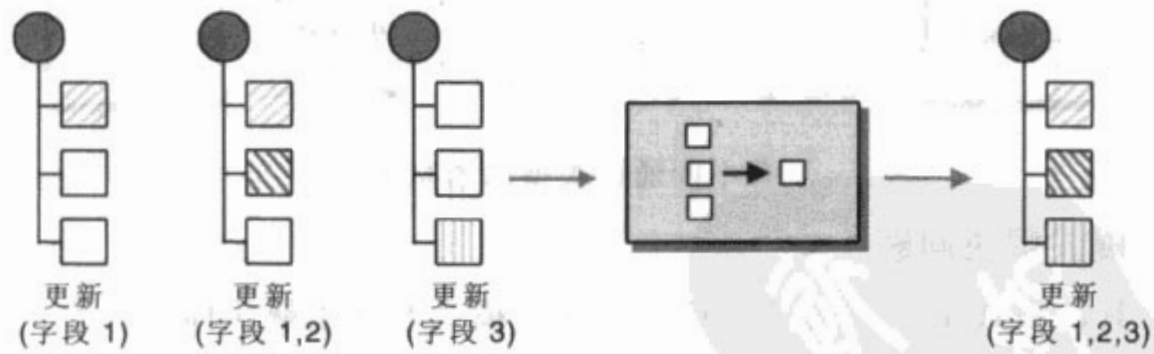
首先想到的是实现一个消息过滤器来控制消息流的速度，它能把引用消息之后很短时间内接收到的更新消息丢掉。例如，可以忽略每个消息接收后 5 毫秒之内接收到的其他消息。消息过滤器可以缓存最后一次接收消息的时间，并丢掉其后 5 毫秒内接收到的任何消息。尽管其他应用可能无法容忍这么严重的数据丢失，但是这种做法在我们的系统中却是可以接受的，因为价格更新的频率并不高。



基于时间的消息过滤器

采用这种方法的缺点是，并非所有的数据字段都在同一时刻更新。每个债券大约要向用户显示 50 个数据字段，包括价格。可以认识到，并不是每个消息中的所有字段都需要更新。如果系统忽略了连续的消息，很可能会丢掉重要的数据。

还有一种有趣的模式，即聚合器。聚合器用于把多个相关的消息组合为一个消息，从而降低了消息流的通信量。聚合器可以根据第一个聚合消息保存债券数据的一个副本，然后只更新后续消息中新的字段或有修改的字段。最后，聚合的债券数据将用一个消息传递给客户端。现在，我们假设聚合器类似于消息路由器，每隔 5 毫秒发送一个消息。后面我们还要讨论另一种解决方法。



采用局部连续更新的聚合器

聚合器和其他模式一样，并不是万能的，要考虑到它的优缺点。其中一个潜在缺点是，只有当与同一债券相关的很多消息在相当短的一个时间段内到达时，使用聚合器才能降低消息的通信量。但是，如果对所有交易商的债券来说，Java 客户端只接收一个字段的更新消息，那么采用聚合器就不会带来任何好处。例如，在一个指定的时间段内，如果接收到感兴趣的 4 种债券的 1,000 个消息，我们可以把消息流从 1,000 个消息减少为 4 个消息。相反，如果我们在同样的时间段内接收到感兴趣的 750 种债券的 1,000 个消息，那么我们只能把消息流从 1,000 条减少到 750 条，实在是得不偿失。对消息流的快速分析证明，Java

客户端接收的多个消息可能要更新同一个债券的字段，因此这些消息是相关的，采用聚合器能很好地发挥作用。

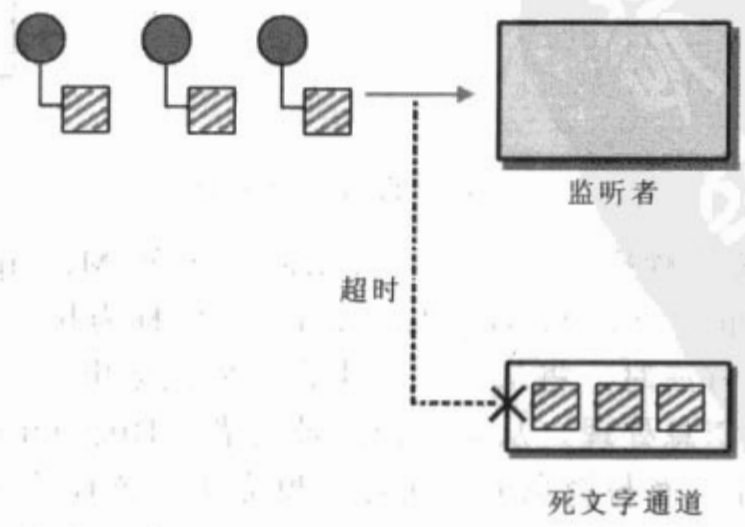
剩下的就是要确定聚合器在何时发送已经聚合好的消息。为此，该模式提供了多种明确发送时间的算法，可以在一定的时间间隔后让聚合器发送聚合消息，也可以在数据集中要求的所有字段都完成更新后发送聚合消息，同时还有其他的一些算法。不管采用哪种方法，都有一个问题，就是会由聚合器而不是客户端控制消息流；因此在这种情况下，客户端（而不是消息流）将成为系统的主要瓶颈。

这是因为，聚合器认为已处理消息的消费者（在这个例子中是客户端应用）是事件驱动的消费者，或者是依赖于外部事件的消费者。我们要把客户端转换为轮询消费者，或者是能持续检查消息的消费者，这样的客户端应用才能控制消息流。只要创建一个后台线程，让它不断地迭代处理债券集，并负责更新和显示自上一次迭代处理后发生改变的债券数据，这样就能实现对消息流的控制。采用这种方式，客户端可以控制接收消息的时间，从而确保在频繁更新期间不会超负荷。这一点很容易实现，只需向聚合器发送一个命令消息并启动更新。聚合器将用一个文档消息作为响应，其中包含客户要处理的一组更新字段。

之所以采用聚合器而不是消息过滤器，这完全是基于系统业务需求做出的决策。这两种模式均能解决性能问题，但是使用消息过滤器无法保证系统数据的完整性。

13.1.7 主产品的崩溃

解决了显示频率带来的性能问题后，我们进入了产品运行阶段。有一天整个系统崩溃了，MQSeries 的崩溃也导致其他几个组件的崩溃。经过一段时间的查找，我们发现问题是由 MSSeries 的死文字队列（它是死文字通道的具体实现）造成的。该队列的过分增长导致了整个服务器的崩溃。经过对死文字队列中消息的分析，我们发现它们都是过期的市场数据消息。这些消息是由“过慢的消费者”引起的，这些消费者处理消息的速度太慢。如果消息在等待处理时超时（见“消息到期模式”），就会发送给死文字通道。死文字队列中如果有过多的过期市场数据消息，这就清楚地表明消息流太快了，也就是说消息在目标应用能够消费之前到期了。我们需要让消息流稳定，因此要借助于降低消息流速的模式。

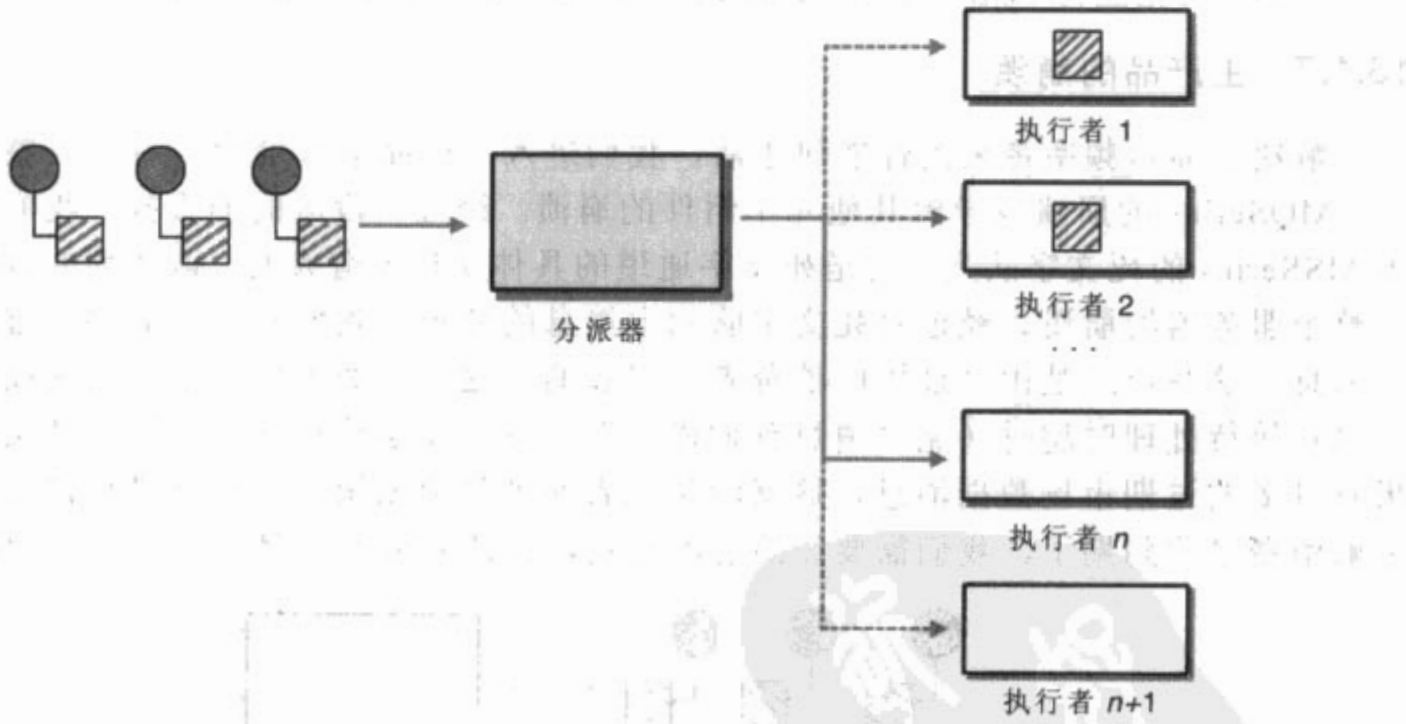


瓶颈

第一个看似可行的步骤是尝试采用聚合器解决这个问题，我们刚刚使用这种模式解决了类似的市场数据显示问题，即速度控制问题。但系统的设计要求，客户端应用要把市场数据更新消息立即转发给交易所。这意味着系统不能停下来收集消息并把它们聚合起来，因此不能采用聚合器。

还有另外两种模式能解决并发消费消息的问题：竞争消费者和消息分派器。首先采用竞争消费者，它的好处是能并行处理到来的消息。在同一个通道上使用多个消费者就能达到并行处理的目的。每个到来的消息只能由一个消费者处理，其他的消费者可以处理后续的消息。但是在这个例子中不能用竞争的消费者，因为服务器到客户端的通信使用了发布-订购通道。如果在发布-订购通道上使用竞争消费者，这意味着所有消费者都处理相同的消息。这样只会造成更多的处理，而无助于问题的解决，也不符合使用该模式的目的。因此采用这种方法也是不可行的。

消息分派器采用的方法是把多个执行者添加到一个缓冲池中。每个执行者可以在自己的线程中运行。有一个主消费者负责监听消息通道，并把消息委托给缓冲池中某个可用的执行者处理，然后立即返回并监听通道。它不但有竞争消费者的并发处理优点，还适用于发布-订购通道。



上下文中的消息分派器

在我们的系统中实现这种模式很简单。首先创建一个 MessageListerner，称为分派器（Dispatcher），它包含一组其他的 MessageListener，它们称为执行者（Performer）。当调用 Dispatcher 的 onMessage 方法时，将从执行者集合中依次选出一个 Performer，由选出的这个 Performer 完成实际的消息处理。从而使消息监听者（Dispatcher）能立即返回。这就保证了消息处理流的稳定性，而与消息流的实际速度无关。该模式既适用于点对点通道，也适用于发布-订购通道。采用这种模式，客户端应用能以任何速度接收消息。如果客户端应

用处理消息的速度仍然赶不上接收消息的速度，客户端应用可以处理延迟甚至过期的市场数据，而不会让消息在 JMS 消息通道中到期。

本节讨论了崩溃问题，并介绍了使用消息分派器实现的解决方案，这些很好地说明了模式在实际应用中存在的限制。由于设计上的缺陷，客户端无法并行地处理消息，这给我们带来性能问题。使用模式只能缓解问题的严重性，但是不能完全克服，因为真正问题在于客户端才是系统的瓶颈，即使采用一千个模式也无济于事。后来我们是通过重新规划消息流体系结构，把消息从定价网关直接路由到投递网关，才解决了这个问题。因此，模式能帮助我们设计和维护系统，但是它们无法补救原本很差的设计。

13.1.8 小结

在本章中，我们在债券交易系统的几个不同方面应用了模式，包括利用模式解决最初设计方面的问题以及几乎危急整个项目的产品崩溃问题。我们还在已有的第三方产品、遗留的组件、JMS 和 TIBCO 消息传递系统中看到了这些模式。最重要的是，这些都是真实的问题，在设计和维护我们自己的系统时，也可能在结构、技术和业务方面遇到同样的问题。希望这里介绍的模式应用实例能帮助读者更好地理解模式，并了解如何在自己的系统中应用模式。



第 14 章

结束语

14.1 企业集成的新兴标准和展望

作者：Sean Neville

随着数据流通过消息传递管道而跨越系统和领域的边界，也随着开发人员和架构师逐步精通支配消息传递系统的模式，会产生新的标准和产品，并扩展模式的应用范围。随着时间的推移，模式只会得到加强而不会改变，但是模式的实现策略会得到快速的发展，以便开发人员把模式应用到更广阔的复杂领域。基本的消息模式随着实现产品从电子数据交换（Electronic Data Interchange，EDI）到专用的面向消息的中间件（Message-Oriented Middleware，MOM），到开放的 XML 和基于 SOAP 的 Web 服务，再到全球业务过程执行语言（Business Process Execution Language，BPEL）等等逐步的发展，得到了更广泛的应用。

本章以应用开发人员在 21 世纪中叶将要遇到的新兴标准为依据，对基于消息的企业集成发展趋势做一个展望。其中许多标准目前并未得到广泛的应用，但是它们已得到大量企业的支持，很可能成为集成模式具体实现的基础，尤其是在面向服务的体系结构中。其中大多数标准都是原先 Web 服务技术的扩展，从而能支持本书介绍的各种模式。我们要说明为什么这些标准对设计模式非常重要、目前有哪些组织正在着手建立这些标准，以及它们的发展方向是什么，并对少部分最有希望的 Web 服务和 Java 标准所提供的各种业务过程集成技术解决方案做一个简短的总结。

14.1.1 标准与设计模式之间的关系

当今有两种智力产物为软件体系结构提供了最高层次的抽象：一种是程序设计的面向性，包括面向对象的程序设计、面向服务的程序设计和增量式程序设计等；另一种是模式语言，如本书介绍的模式语言。如果已经证明某种特定的程序设计面向性或设计模式很有用，并且其上下文经常出现，那么实现这种模式的方法和策略就会非常相似。基于多种平台、产品和应用的模式解决方案差别很小，但是这些很小的差别，尤其是自然语义方面的区别，往往阻碍了模式的推广应用，产生不利于大规模集成的互操作问题。为了扩展应用及应用所基于的模式的作用范围，必须采用正式协商的标准，产品和平台之间的这种差异才能消除。

随着模式实现方法或策略的标准化，模式的有效性并不会因标准而降低；相反，只会得到进一步的加强。就像大树的年轮一样，一个强大模式的应用规模会逐步增长，它将创建并

应用到更大的实例中。模式实现的不同实例能够相互操作，因此规模会不断增长。以一个典型的面向消息的 J2EE 应用为例，像管道和过滤器这样的模式会在不同层次的系统中出现，例如可以包含在开发人员设计的应用中、包含在运行应用的服务器产品中、包含在服务器内部的容器和服务中，还可以包含在组成消息传递子系统的过滤器和组件中，如此嵌套很多层。

不断出现的消息传递和 Web 服务标准扩展了开发人员应用模式的领域，使本书介绍的模式得到进一步加强。像 BPEL 和 Web 服务可靠性（WS-Reliability）这样的标准将模式的应用范围推广到不同的领域、不同的语言、不同的产品中，使模式越来越多地应用到以人和系统为中心的用例分析和需求分析过程中。应用开发人员可以不再使用关联标识符之类的模式把原始的消息应答与其发送者相匹配，但是可以使用同样的模式把包含多个异步消息发送者和接收者的过程组件关联起来。如果没有制定相应的消息传递标准，Java 开发人员会在代码级实现消息路由器，检查每个 XML 消息的内容，以便用程序设计的方式把消息转发给特定的服务或过滤器。新兴的工作流和编排（choreography）标准则使开发人员能运用所掌握的更高抽象层次消息路由器模式方面的知识，实现过程组件之间的工作流路由。实现模式时采用的操作对象已变成业务过程组件，而不是原始的 XML 文档片断。换句话说，应用开发人员不用再花宝贵的时间实现不同协议的连接，而只是考虑如何跨业务领域边界连接业务过程和服务。

设计模式和程序设计面向性会为开发复杂系统带来很大好处，这与应用开发人员运用可互操作的公共实现策略和方法的能力成正比。现在，运用 J2EE 消息传递模式的策略是指使用 JMS、JCA 或 JAX-RPC；将来，应用开发人员使用的策略可能会包含使用模型驱动的技术、基于模式的脚本、方面或意图等等——如果适当地包含了模式的上下文定义和相关的标准，那么所有这些实现方法上的变化只会使模式更加重要。就目前所知，标准是增强共性的最佳途径，以便软件架构师更熟练地掌握设计模式。

14.1.2 标准进程和机构一览

与大众的批评正相反，开发商开发的标准不可能与实际的应用开发分离。标准的目的是为了统一和改进应用开发人员已经发现的和开发的实现方法。通过使用本书介绍的设计模式，即使你没有直接参与标准制定小组的工作，也能对标准的开发做出贡献。监管标准制定小组的机构和协会并没有把大量的精力用在认可和接纳具体的实现方法上，不过这是目标。

当标准的制定者或制定小组向标准组织提出正式的草案后，标准就正式诞生了。标准的制定者通常是标准组织的成员。从提出草案到最后形成标准，每个标准组织都有自己的标准处理程序，所有成员都要遵循这些规则。标准处理程序包括建立工作组或委员会，进一步完善存在疏漏的规范，最终批准组织对标准所做的管理。知识产权和许可政策往往是这些组织中的热门话题，在标准组织之间（甚至在标准组织内部的工作小组之间），知识产权和许可政策可能有变化。

已经有一些标准组织在创建新兴的消息传递和 Web 服务标准，其中主要标准组织如下：

W3C：WWW（World Wide Web）联盟（http://www.w3c.org）开发了许多基本的 Web 技术，这些技术成为其他标准组织建立标准的基础。该联盟由一个国际性的研究人员和工程技术人员组成的小组管理，并由大量的成员开发商、内容提供商、政府部门、研究实验

机构和其他团体组成。W3C 在工作小组及最后的公布中使用了开放、协作的评审过程，最终会形成冗长的标准，但有很高的质量。W3C 提出的技术不会受到其成员对知识产权保护声明的阻碍。W3C 技术包括 SOAP 和 WSDL，以及所有核心 XML 模式。W3C 编排工作小组（Choreography Working Group）有希望克服业务过程规范之间的冲突，为使用 Web 服务进行企业集成提供可靠的基础。

OASIS：结构化信息标准促进组织，简称 OASIS（Organization for the Advancement of Structured Information Standards），是建立在 W3C 技术之上的一个标准组织（http://www.oasis-open.org）。这个非盈利性的开发商联盟致力于为 SGML 的开发提供指南，目前的工作重点是促进全球电子商务标准的推广应用。OASIS 技术委员会提出了许多新兴的 Web 服务标准，如 ebXML 和 WS-Reliability。OASIS 还维护着 xml.org 门户网站。

WS-I：Web 服务互操作组织（Web Services Interoperability Organization）（http://www.ws-i.org），其宗旨是确保 Web 服务技术和标准能以通用的、可互操作的方式满足业务合作的需要。该组织提出的 Web 服务协议和规范可以应用到多种不同的系统、平台和语言。实现这些目标的关键策略是 WS-I 基本规划（WS-I Basic Profile），它提出了一组 Web 服务标准以及这些标准的版本号，第一个规划中包括 XML Schema 1.0、SOAP1.1、WSDL1.1 和 UDDI1.0，还提供了综合使用这些 Web 服务标准的相关约定。因此，WS-I 扮演了一种协调角色，以确保不同的 Web 服务开发商能共同为应用开发人员提供服务。该组织由主要的 Web 服务开发商建立和管理，包括 Microsoft、IBM、BEA Systems、Oracle 等等。

JCP：Java 群体过程协会（Java Community Process）（http://www.jcp.org），为其他组织开发的 Web 服务和消息传递标准等技术提供 Java 语言的绑定和 J2EE API。JCP 是由 Sun 公司领导的，它原本不是开放的标准组织，尽管也使用了类似于其他标准组织那样的专家小组和提案处理步骤，但是它的知识产权归 Sun 公司保留，要获得其许可必须向 Java 和 J2EE 平台开发商支付费用。大多数 JSR 专家小组也是由 Sun 工程师领导的。尽管 JCP 缺少其他组织那样的开放性，但是得到了 Sun 充分的关注，从一定程度上减少了外部议程造成的妨碍。

Ad Hoc 开发商联盟：在争夺对新兴 Web 服务技术（尤其是企业集成技术）控制权的竞争中，传统的竞争者（如 IBM 和 Microsoft）有时会联合起来发布自愿的标准，这些标准并未提交给任何标准组织审批。许多 WS-*规范就属于这样的标准。通常，开发商联盟开发的标准最终会提交给标准组织，例如，倍受推崇的 BPEL 在提交到 OASIS 前，最早就是由 Microsoft 和 IBM 建立的。出于保护知识产权的目的，促使开发商以自主（Ad hoc）的方式提出标准。这些规范在不开放许可的条件下实现了互操作，如果应用开发人员要集成的平台就像 Microsoft 和 IBM 的平台那么通用，采用这些规范是没有什么问题的，但是这些规范是否属于真正的“开放标准”仍然存在广泛的争议。

14.1.3 业务过程组件和 Web 服务内的消息传递

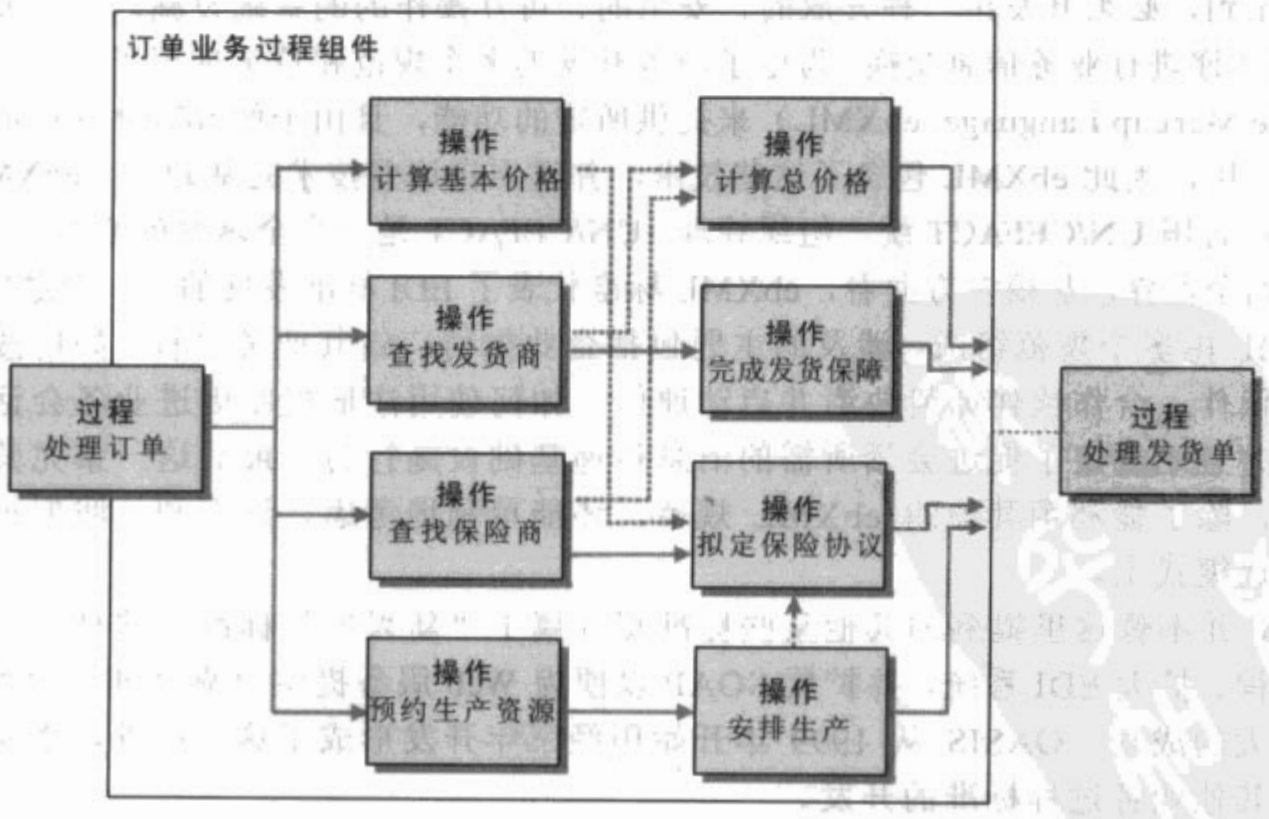
亚里士多德的思想不但影响着面向对象的程序设计、结构的分解和业务领域的建模，还引发了一个最具影响力的、一直困扰着软件体系结构的问题：世界是由一系列的过程支配的，还是由一系列的对象支配的？

消息传递标准给出了一个值得其信徒信服的答案。世界是由一系列对象组成的，对象最重要的特性是过程，通过这些过程使对象之间能相互协作和联系起来。某个对象的行为往往与其他对象相关，而不是与看起来最重要的内部组件相关。在 Web 服务和企业集成的领域中，对象/过程的混合体称为业务过程组件。业务过程组件把一系列的服务联合起来，形成一个逻辑单元，它能通过消息传递与其他类似的逻辑单元交互，实现有很强扩展性的、灵活的逻辑流和数据流。在许多 Web 服务开发商和标准组织看来，过程、对象和交互模式联合构成的业务过程组件是消息传递的发展方向。

过程组件是一组与企业集成紧密相关的服务所组成的宏观视图。举一个非技术性的例子，人可以驾驶汽车，人与汽车都是分离的、非常复杂的系统，但是人与汽车形成的过程视图就像一个单一的组件，它是由人与汽车的彼此交互定义的，我们关注的重点不是过程视图中脱离交互的人或汽车。此外，过程组件视图描述了单个组件与公路上的其他组件之间的交互，重点也不是其他汽车或驾驶员的内部构成。与简单地提出一个外部接口不同，过程组件还包含了使用这些接口的行为规则。

在新兴的标准中，业务过程组件是一个单一的组件，它的内部由一组 Web 服务组成，同时定义了进出这些服务的消息流。Web 服务和其他消息目标可以作为基本单元组成更大的组件，它们之间的交互都遵循消息传递模式，这些交互包括过程组件的组合，以及把多个过程组件链接成一个应用。业务过程标准提供了 Web 服务之间的消息关联，以便创建执行流程，并包含与错误处理、消息转换和数据交换等方面相关的行为。

下图给出了一个更技术化的业务示例，其订单的处理由一个组件完成，其中包含若干个服务操作。



业务过程组件提供了单一的目标端点，它代表了多个内部编排好的 Web 服务

这个例子包括以下特点和活动：

- 处理订单组件中的每个操作都是由 Web 服务的 portType 提供的，提供操作的各种 Web 服务可以驻留在合作生产商的远程计算机上。
- 接收到订单时，四个操作被同时调用，但是其中有依赖关系的部分操作必须处理好同步问题（图中的虚线表示依赖关系）。
- 这里有两个特定的依赖关系，表示计算最终的成本需要有发货协议和保险成本，另外在制造商提交资源安排生产前，必须收到相关的保险协议。
- 一旦所有的服务操作异步地完成，订单的处理就完成了。消息传递客户端不必与所有这些服务单独交互，也不用管理它们之间的依赖关系；也就是说，客户端不用创建自己的过程管理器，过程组件能代表所有服务提供唯一的网关端点，并管理所有服务之间的消息传递和依赖关系，这就简化了客户端的任务。
- 订单过程组件再链接到其他的业务过程组件：处理发货单的组件。

有 4 项标准草案为业务过程组件和集成提供了很好的接口，它们是 ebXML 草案、业务过程执行语言（Business Process Execution Language）、Web 服务编排接口（Web Services Choreography Interface），以及一组有 WS-前缀的独立规范，其中业务过程执行语言和 Web 服务编排接口是两个竞争议案，各个有 WS-前缀的规范分别以更特定的方式在更窄范围内提供相同功能的定义。

14.1.4 ebXML 和电子商务消息服务（ebMS）

即使在 SOAP 和 WSDL 得到广泛应用之前，许多从事商业合作和 B2B 集成项目的有智之士就预见到，必须开发出一种开放的、安全的、可互操作的的基础设施，而且要能够使用 XML 消息传递进行业务信息交换。为电子商务开发的多个规范和草案使用了可扩展标记语言（eXtensible Markup Language, ebXML）来提供所需的功能，且由于像 SOAP 这样的技术已经得到广泛应用，因此 ebXML 包含了这些技术，并建立在这些技术的基础上。ebXML 草案由 OASIS 和联合国 UN/CEFACT 统一组织管理。UN/CEFACT 是一个全球性的小组，它曾把 EDI 标准推广到全世界，从很多方面看，ebXML 标准代表了 EDI 标准发展的下一个逻辑阶段。

ebXML 由多个规范组成，涉及的主题包括企业如何广告其业务过程、如何搜索那些潜在的合作伙伴、合作伙伴如何协商并启动通信、如何使用注册机制促进业务会话的发现和初始化，并且还规定了促进会话所需的消息传递基础设施行为。最后这一条尤其引起了极大的关注，除了能利用其他的 ebXML 规范，还能单独地考虑，这有利于我们把重点集中到消息传递模式上。

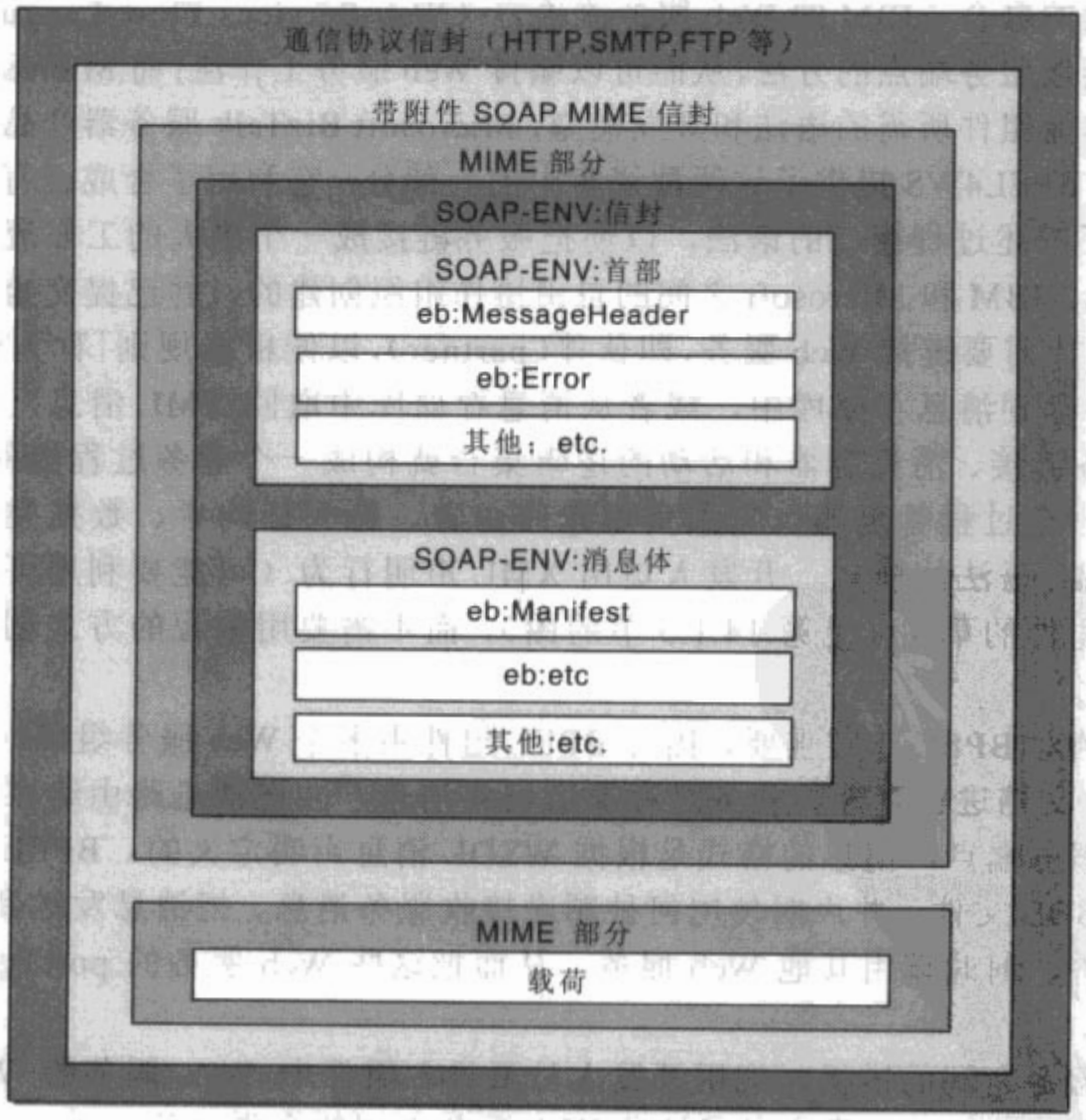
ebMS 并不像这里提到的其他某些标准那样属于“新兴的”标准，这种方法可用以集成业务过程、扩大 EDI 系统，并扩展 SOAP 以便为 Web 服务提供可靠性和安全性，它已经取得了极大的成功。OASIS 从 1999 年开始历经三年开发形成了这一标准，事实证明它确实影响了其他业务过程标准的开发。

ebMS 的目标是为了方便 XML 框架中的业务消息交换，但是该框架包含了非 XML 的消息载荷（payload）的使用，即载荷可以采用任何形式，包括传统的 EDI 格式以及二进制

格式。因此 ebMS 可以封装已有的消息传递系统，而且可以作为灵活的桥接技术，包含了一组消息转换器的具体实现，这些消息转换器实现对于将集成扩展到外部网非常有用，另外对于商业伙伴之间的 B2B 通信也很有用。ebMS 主要用于连接跨不同企业的专用 MOM 系统，而且开发商和开发人员之间已经有大量的测试来验证这种互操作性。ebMS 的关键特点在于它支持遗留的 EDI 系统，允许企业发挥 EDI 的长期投资效益，同时还能集成 Web 和 XML 的特点，弥补 EDI 的不足。

尽管从一个方面看 ebMS 是 EDI 的兼容版本，但从另一方面看它也是对基于 SOAP 的标准服务全面的改进。ebMS 系统中使用的 XML 传输由 SOAP 信封组成，SOAP 信封包括 ebMS 特定的 SOAP 首部，还包括一个清单，SOAP 首部用于记录唯一的消息标识符、时间戳、数字签名，清单提供了有关消息载荷的元数据。因此，ebMS 使用了 SOAP 首部机制来实现消息路由模式和许多消息传递端点模式，如幂等接收者以及其他与可靠顺序传输有关的模式。

ebMS 使用带附件的 SOAP 标准（SOAP with Attachments standard）传递其载荷，把载荷附加到 SOAP 信封中，这与把附件附加到电子邮件消息中非常类似。载荷没有任何格式方面的限制：消息的载荷可以采用 XML 数据、二进制数据、基于链接的外部数据引用等。另外，每个载荷都可有自己的数字签名，ebMS 实现者可以提供额外的认证和授权功能。



ebMS 消息作为带附件的 SOAP 消息

异步消息传输是默认的选择，但是也能用 ebMS 实现同步传输。错误处理机制也相当复杂，取决于具体实现，可以提供 SOAP 出错信息以及与载荷相关的错误消息。

为了提供可靠性，ebMS 具体实现的关键元素（称为 ebXML 消息服务处理程序）能在会话的发送端保存消息。开发人员可以为每个消息或消息组提供诸如“一次且仅有一次（once-and-only-once）”以及“存储转发（store-and-forward）”等语义声明。ebMS 还指定了用于管理顺序传输和实现系统管理的服务，最后这个服务是通过一个消息状态服务实现的，它相当于控制总线。利用这个服务，可以请求以前发送给 ebMS 系统的消息的状态，底层操作由消息历史和相关模式的具体实现完成。

有关 ebXML 的更多信息可参见 http://www.ebxml.org/。也可以在 http://www.ebxml.org/specs/ebMS2.pdf 找到 ebMS 规范。

14.1.5 针对 Web 服务的业务过程执行语言（BEPL4WS）

针对 Web 服务的业务过程执行语言（Business Process Execution Language for Web Services，BPEL4WS）是新兴的流行标准，用于定义业务过程组件和它们之间的交互。这个缩写的发音一般为 bee-pel，或 bee-pel for wuss。该标准的出台表示来自 IBM 和来自 Microsoft 的两个竞争提案的融合。IBM 的 Web 服务流语言（Web Services Flow Language，WSFL）提供了创建和链接服务端点的方法，从而可以编排 Web 服务工作流；而 Microsoft 的 XLANG 提供了创建工作流组件所需的语法和开发模型，Microsoft BizTalk 服务器产品中实现了这样的工作流组件。BPEL4WS 提供了这两种议案中的一部分：它利用了合成已有服务中过程的语法，还包括了描述过程接口的语法，以便把服务链接成一个更大的工作流。这项标准是由 BEA Systems、IBM 和 Microsoft 之间的自由协作组织创建的，并已提交给 OASIS。

BPEL 的设计需要链接 Web 服务，即伙伴（partner），以便根据规则[称为活动（activity）]把 XML 消息保存到消息存储库中，或者从消息存储库中取回 XML 消息，消息存储库也称为容器。服务链接、消息容器和活动的逻辑集合就构成一个业务过程组件。BPEL 规范实际上定义了一个过程管理器，它负责创建路由表、持久订购者、数据类型通道和其他基于描述性 XML 语法的模式。开发人员用 XML 声明行为（可能要利用可视化工具）来建立业务过程组件的草图（见第 14.1.3 节的图），而不需要用编程的方式创建服务之间的消息传递。

WSDL 文档对 BPEL 非常重要，因为 BPEL 组件由多个 Web 服务组成，并且根据这些服务的 WSDL 文档进行管理。业务过程要实例化消息，并把消息路由给在服务的 WSDL 文档中声明的服务端点，消息的格式是根据 WSDL 消息声明定义的。BPEL 语法通过导入 Web 服务的 WSDL 文件，并声明使用何种顺序接收服务消息、把消息发送到哪里、何时以及如何做出应答、何时调用其他 Web 服务，从而把这些 Web 服务的 portType 和操作链接起来。

为了建立服务之间的链接，应用开发人员要导入组件中 Web 服务的 WSDL，并使用 BPEL 的 serviceLinkType 元素定义了这些 Web 服务之间的合作伙伴关系。这个元素指示工作流中 Web 服务的 portType，以及 Web 服务在过程中扮演的伙伴角色；声明流入和流出这

些服务的消息时，开发人员就能参考这些链接和伙伴角色。

声明了服务之间的关系后，开发人员可以使用 BPEL 来创建容器，以保存服务所用的输入和输出消息。容器非常像用 WSDL 为不同类型消息定义的数据类型通道。根据容器的具体实现，也可以把它看作是共享数据库集成模式，只不过还增加了封装和内置协作语义。容器是由不同服务使用的共享数据库。除了通过 BPEL 元素把容器与服务连接起来外，也可以直接通过 XPath 扩展访问容器的内容。

BPEL 组件通过单一的接口提供自己的输入和输出点，为此它与其内部的 Web 服务一样，都是通过 WSDL 来做到。但是，业务过程组件的 WSDL 文档与 Web 服务的 WSDL 文档有一些区别，这是因为，在业务过程 WSDL 中，portType 定义的是进出该过程的入口点，而不是实现为服务接口方法的单独的逻辑块。

BPEL 组件的行为声明为一组动作（action）。在任何一个特定的动作中，业务过程均可向服务发送消息（在这种情况下，Web 服务被称为“被调用的伙伴”），从 Web 服务接收消息（在这种情况下，Web 服务被称为“客户端伙伴”），对客户端伙伴发送的消息做出应答，确定是否应该根据某种逻辑规则接收或发送消息，等待预定的时间间隔，报告错误，把消息从一个地方复制到另一个地方，或者什么也不做。

BPEL 的 XML 语法反映了这些基本动作。开发人员使用 invoke 元素把消息发送给被调用的伙伴，并使用 receive 接收来自客户端伙伴的消息，使用 reply 元素返回应答。flow 和 pick 之类的元素把逻辑分为并行的和基于事件的执行通道。throw 元素有利于错误报告，而 wait、empty 和 terminate 元素可暂停或中止执行。构造活动的元素包括 while、sequence 和 pick，BPEL 中的条件使用 XPath 语句声明。

一旦开发人员创建了 XML 源文件（其中定义了构成业务过程的服务），声明了服务所使用的消息容器，并且声明了消息交换时涉及的操作序列，他（或她）就准备好了业务过程组件的部署。之后就可以让 BPEL 具体实现的运行时方面发挥作用了。

除了能作为服务关系和消息流的声明之外，BPEL 描述也是可执行的文件，能输入到 BPEL4WS 引擎中。把 BPEL 描述文件输入引擎时，引擎将解释文件，并为应用开发人员建立一系列的消息传递结构，把业务过程中的 Web 服务连接起来。作为过程管理器的具体实现，BPEL4WS 运行时要作为控制实体管理和协调服务之间的消息流。BPEL4WS 引擎接受所有必需的文档，并动态地生成和管理消息传递基础设施。

读者可以在 http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/找到 BPEL 的规范。

14.1.6 Web 服务编排接口

Web 服务编排接口（WSCI，通常读为 whiskey）所要解决的问题与 BPEL 一样。这两项标准相互竞争，并分别由相互竞争的开发商联盟支持（BEA 是例外，它同时支持这两种规范）。WSCI 由 Sun、Intalio、SAP 和 BEA 支持，并且已经提交给 W3C。但是，几个 WSCI 的最初支持者现在已转而支持 BPEL。

WSCI 受到了业务过程建模语言（Business Process Modeling Language，BPML）的影

响，该语言与集成模式没有直接的关系，但是在 WSCI 的发展历史中需要提到。BPML 用过程建模采用的一种基于 XML 的元语言表示业务过程。BPML 的工作流部分大多属于 WSCI，不过 BPML 还包含了对应的图形符号和查询语言。

和 BPEL 一样，WSCI 认为企业集成涉及到其中各个服务之间的大量会话，而不只是由基本的 Web 服务协议定义的单个操作调用。WSCI 提供了相应的方法，可以把来自多个操作的服务消息链接到这些组合过程中，并确保沿途的消息能按正确的顺序接收和发送、能根据声明的业务规则发送和接收消息、如果需要还能以事务性方式发送消息，并能作为一个全局过程提供可描绘性和可管理性。为了把 Web 服务的优点与传统的专用 MOM 解决方案相结合，WSCI 还提供了服务的动态发现、异构协议和工作流的分散协作。

和 BPEL 一样，WSCI 在很大程度上依赖于 WSDL 的服务端点、portType、操作和消息类型的声明。在一个编排中，WSCI 动作直接映射到所提供的操作，操作在各服务的 WSDL 中提供。而且，WSCI 语法直接嵌在 WSDL 文件中，其声明要么位于 WSDL 文档中，其中导入了编排服务的 WSDL 文档；要么位于单个服务的 WSDL 文件中（将编排该服务的操作）。WSDI 元素包含在 WSDL definitions 元素中。

WSCI 的基本结构是动作，Web 服务消息传递就在动作中出现。动作组合到过程元素中，可以声明为按串行、并行、循环或条件的形式发生。一组过程利用一个接口声明来分组；这个 interface 元素就是过程组件的接口，它直接嵌入到 Web 服务的 WSDL 定义中。许多活动元素类似于 BPEL 的活动元素，也提供对 XPath 表达式的支持。

可以在 http://wwws.sun.com/software/xml/developers/wsci/wsci-spec-10.pdf 找到 WSCI 的规范。

14.1.7 Java 业务过程组件标准

JCP 为非 Java 标准创建 Java 语言 API 和 Java 绑定（binding）。特别地，由对象管理小组（Object Management Group，OMG）和 W3C 开发的标准已经受到了 JCP 的影响。最近，JCP 把方向对准了这些小组开发的 Web 服务标准，如 WS-I，为了解决业务过程组件的 Java 绑定问题，有两个新的 JSR 标准得到了广泛关注：面向 Java 的过程定义（JSR-207）以及 Java 业务集成（JSR-208）。面向 Java 的过程定义标准由 BEA Systems 提交；Java 业务集成标准由 Sun Microsystems 公司提交。乍看起来这两个提案存在交叉，但是它们正好相互补充。JSR-207 为开发人员提供了相应的方法，从而可以使用 Java 代码附加的元数据来快速、方便地建立消息传递或过程组件；JSR-208 指定了这些组件如何实现相互之间的通信，如何与容器通信，以及如何与 J2EE 和 Web 服务的其他组件通信。因此，JSR-207 偏重于微观方面，JSR-208 偏重于宏观方面，它们对利用 Java 实现过程组件之间的消息传递实现了标准化。

面向 Java 的过程定义(JSR-207)

面向 Java 的过程定义（JSR-207）是由 BEA Systems 提交的，目的是为创建 Java/J2EE 环境中的业务过程定义元数据、接口和运行时模型。这个 JSR 标准非常重要，它着力把利用 Java 语言和 Javadoc 形式的元数据符号来创建业务过程组件的方式进行标准化。作为

J2EE 的附加标准，也能用于建立业务过程草案的 Java 实现，如 BPEL4WS、WSCI 以及那些由 W3C 编排工作小组提出的草案。

这项技术建立在 Java 语言元数据技术（JSR-175）之上，其目的是为描述业务过程提供简单的语法。元数据可以直接在 Java 源代码中应用，从而动态地生成和绑定过程行为，包括支持异步消息传递、并发执行、消息关联、消息路由、错误处理和其他公共的流活动。因此，元数据的语义需要进一步扩充才能提供必要的参数，从而允许组件的容器动态地建立消息传递集成设施，以及处理第 5 章“消息结构”讨论组件部署时描述的问题。

值得注意的是，该 JSR 并不需要支持开发人员建立 J2EE 中的业务过程组件。也许现在能建立这样的过程，但这是一项繁琐的工作，开发人员必须在很低的层次上应用消息传递模式，导致工作流的维护成本变得很高。该规范试图简化过程组件的创建，使开发人员能在更高的层次上应用他们的设计技巧，以更快的方式创建更强大的应用，并且随着时间的推移，这些应用的维护和管理成本都会更少。

有关面向 Java 的过程定义的详细内容可参见 http://www.jcp.org/en/jsr/detail?id=207。

Java 业务集成(JSR-208)

Sun 在 JSR-208 中提出 Java 业务集成（Java Business Integration，JBI），其目的就是为业务集成环境的创建定义服务提供者接口（Service Provider Interface，SPI），这些集成环境符合 WSCI、BPEL4WS 和 W3C 提出的规范。JBI 并没有提出新的 Java API 或符号，但是它包含了新的部署和打包机制。JBI 没有为开发人员提供新的 API，而是把重点放在集成基础设施上，它的 SPI 主要是为产品开发商提供，以便创建消息传递和过程组件模型来实现 Java 环境中的集成工作。JBI 专家小组接受 W3C 编排工作小组的领导，并确保该小组的工作成果能无缝接入到 J2EE 平台中。

JBI 提出了非常高的目标：能实现各种系统和协议标准之间的映射以及它们到 J2EE 的映射；也就是说，可以采用各种语法描述过程之间的关系，包括专用的、开发商特定的语法以及那些成为标准的语法。JBI 为消息编排提供了 Java 绑定，而不考虑底层消息和过程的细节。它还包含了新的打包机制，该机制扩展了 J2EE 的打包机制（如 WAR、JAR、RAR 和 EAR），用以支持 JBI 组件在 J2EE 环境中的部署。

JBI 认为支持过程组件需要三个主要角色：绑定、计算机和环境。绑定有关于通信的格式，包括消息格式和网络传输及其映射，并形成工作流格式（如 BPEL）的基础；计算机是指服务和过程容器，在其中存放和管理业务过程；环境是建立在过程之上的过程管理系统，可以连接异构计算机，并将它们相互绑定。JBI 把环境看成是集成系统的核心，并指定了计算机和绑定如何与之交互。JBI 打包和部署机制提供了一种方法可以将过程与环境以标准方式挂接起来，但是规范并不涉及这些组件的实际创建。

JBI 把前面介绍的面向 Java 的过程定义（JSR-207）看成是一种组合过程组件的方法，包容这些组件的运行时也可归为 JBI 计算机的范畴。计算机应该实现消息转换器、服务激励器、信封包装器和其他必要的模式，以便用 JBI 描述其格式和协议行为，确保计算机能与集成环境正确地集成。

有关 JBI 的详细信息请查阅 http://www.jcp.org/en/jsr/detail?id=208。

14.1.8 WS-*

与前面介绍的过程集成规范不同，许多 Web 服务规范只是为了扩展基于 SOAP 或基于 WSDL 的 Web 服务，让它们能提供可靠性、安全性、状态性和服务质量。这些规范一般以 WS-作为前缀，建立在 W3C 技术之上，其中每个规范都解决一个相当特定的问题。

遗憾的是，由于各个竞争的开发商联盟支持不同的版本，使得 Web 服务标准的前景变得黯淡，也许正是这种竞争的结果导致其中许多标准至今尚未真正实现，就好像是海市蜃楼一般。然而，这些规范同样值得注意，如果应用开发人员正在使用 Web 服务技术（如 SOAP 和 WSDL）来实现基于消息传递的企业集成，这些规范的思想和方法就很有用。提出这些规范的开发商肯定会提供相应的产品支持，如 IBM、BEA、Microsoft 和 Oracle。

在这一类规范中有几个著名的标准，分别解决事务、可靠性、路由、会话状态和安全性，下面我们会逐一介绍。

WS-Coordination 和 WS-Transaction

由于无状态和不可靠，大多数流行的 Web 服务协议和传输没有提供事务性过程所需的服务质量。这一不足正成为一个非常严重的问题。在实际运用中，如果开发人员想使用基于 Web 服务的集成机制，就必须在这些集成机制中建立自己的事务方案。通过异步消息传递产生的一组服务调用必须能以原子方式批量提交，才能使属于同一单元的所有消息处理能在失败时全部撤销，或者触发某种补偿机制。在专用的 MOM 系统中经常会出现这种情况。Web 服务协作（WS-Coordination）和 Web 服务事务（WS-Transaction）规范能为 Web 服务解决这样的问题。

WS-Coordination 是由 BEA、Microsoft 和 IBM 起草的，它指定了一种方法，使参与到工作流中的所有服务可以创建和传播上下文信息，甚至可以是异步或按不同的时间间隔进行。该规范提供了一个可扩展的框架来创建协调应用和服务行动的协议。这些协调协议通过创建和注册基于 XML 的上下文来达到目的，一次交互中，这些上下文通过 SOAP 消息进行传播，并由位于所有端点中的协作者使用。

这种上下文可以用来支持许多应用的行为，如需要与分布式事务结果保持一致。因此，WS-Transaction 规范使用 WS-Coordination 实现跨服务调用的分布式事务。

WS-Transaction 定义了一种方法来监视和度量工作流中每个动作成功还是失败。实际中，这意味着当一条 SOAP 消息到达端点时，必须从 SOAP 消息中过滤和取出包含了协作上下文的 SOAP 首部（具体实现时可以采用内容过滤器和分解器模式），然后把它发送给事务协调者进行解释。

下面给出的 SOAP 信封提供了一个简单的协作上下文示例，该示例通过 WS-Transaction 使 SOAP 操作成为事务性操作。协调者利用 SOAP 首部中的上下文信息将应用注册为接收事务事件，如加入、两阶段提交过程的准备阶段、回滚和提交。

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope">
```

```
<SOAP-ENV:Header>

  <wscoor:CoordinationContext
          xmlns:wscoor="http://schemas.xmlsoap.org/ws/2002/08/wscoor"
          xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility"
          xmlns:myTransactableApp="http://foo.com/baz">

    <wsu:Identifier>http://foo.com/baz/bar</wsu:Identifier>
    <wsu:Expires>2004-12-31T18:00:00-08:00</wsu:Expires>

    <wscoor:CoordinationType>
      http://schemas.xmlsoap.org/ws/2002/08/wstx
    </wscoor:CoordinationType>

    <wscoor:RegistrationService>
      <wsu:Address>
        http://foo.com/coordinationservice/registration
      </wsu:Address>
    </wscoor:RegistrationService>

    <myTransactableApp:IsolationLevel>
      RepeatableRead
    </myTransactableApp:IsolationLevel>

    </wscoor:CoordinationContext>

  </SOAP-ENV:Header>

  <!-- SOAP BODY (snipped) -->

</SOAP-ENV:Envelope>
```

WS-Transaction 规范为开发人员定义了可在应用中使用的两种协作：原子事务（atomic transaction，AT）和业务活动（business activity，BA）。

原子事务能很好地与传统的分布式事务技术（如 XA）对应，适用于生存期很短的操作，其中允许锁定资源，如线程和部分数据源，并且允许绝对回滚。WS-Transaction 提供了一种链接专用 XA 实现的方法，并支持对 Web 服务的两阶段提交。

业务活动是一个典型的长生存期过程，可能由许多原子事务组成。在一个业务活动中，单一的失败事件不应导致全局回滚。相反，业务活动中某个原子事务的失败通常应该触发其他服务调用和消息交换。这些交换可以包含补偿技术，通过保存业务活动的一部分历史实现错误的恢复。例如用户要做一个期限为两到三天的业务活动，它包含航班订票、汽车租赁、旅馆房间预定和演出票预定。工作流中的每项活动都是一个原子事务，如果其中一项活动失败，制定旅行安排的整个业务活动应该通过补偿的方式进行调整，而不是撤销所有的原子事务。

对应服务消息传递期间产生的失败事件，WS-Coordination 和 WS-Transaction 规范力图做到能提供可靠、完备的行为。随着这些规范逐步走向产品化，应该能减轻开发人员的负担，使之不必再在基于消息的服务应用上开发自己的事务和上下文相关服务。

有关 WS-Transaction 的信息可参考 http://dev2dev.bea.com/technologies/webservices/ws-transaction.jsp。

WS-Reliability 和 WS-ReliableMessaging

即使是出现网络、应用或组件的失败事件，在 Web 服务之间实现基于消息的事务通常都需要可靠的消息传递，还要包括持久性机制和重传语义。但是，即使是最受欢迎的 Web 服务技术也没有提供这样的可靠性。例如，不带增强技术的 SOAP 在许多企业消息传递应用中毫无用处，因为它最受欢迎的绑定特性并不能保证可靠的消息传输。

为了弥补这一缺陷，应用开发人员不得不使用 Web 服务的可扩展性机制（如 SOAP 首部）自己实现可靠性。为了避免让应用开发人员承担这样的任务，出现了一些提高 Web 服务可靠性的新标准。其中两个规范是 Web 服务可靠性（WS-Reliability）和 Web 服务可靠的消息传递（WS-ReliableMessaging）。就像 Web 服务的其他标准一样，这两个标准也是相互竞争的，它们分别得到不同的开发商支持，但目标都是为了解决相同的问题。

WS-Reliability 使基于 SOAP 的 Web 服务能够异步地交换消息，而且能保证可靠传输、没有重复消息，并保持消息有序。它是一个 SOAP 标准，能够管理消息聚合和排序，并提供了实现可靠传输和重排器模式的标准方法，还能实现其他的模式。WS-Reliability 进一步扩展了 SOAP 首部机制，为 SOAP 消息增加了 MessageHeader、ReliableMessage、MessageOrder 和 RMResponse 等首部元素。这些元素表示了组 ID 和序列号等消息标识符、时间戳、生存期、消息类型、发送者和接收者信息以及确认回调信息。WS-Reliaility 由 Sun、Oracle 和 Sonic 等开发商提出，并提交给 OASIS。它受到 ebMS 功能的很大影响。

为了遵循 WS-Reliability 标准，SOAP 消息的接收者必须在 SOAP 首部中使用 <RMResponse>元素响应一个失败或确认消息。如果没有接收到这样的确认消息，发送者就会使用相同的消息标识符重传相同的消息。发送者必须持久地保存消息，直到它的生存期超时，或者直到收到确认或失败消息。接收者也要持久地保存消息，直到消息可靠地传送给应用层。

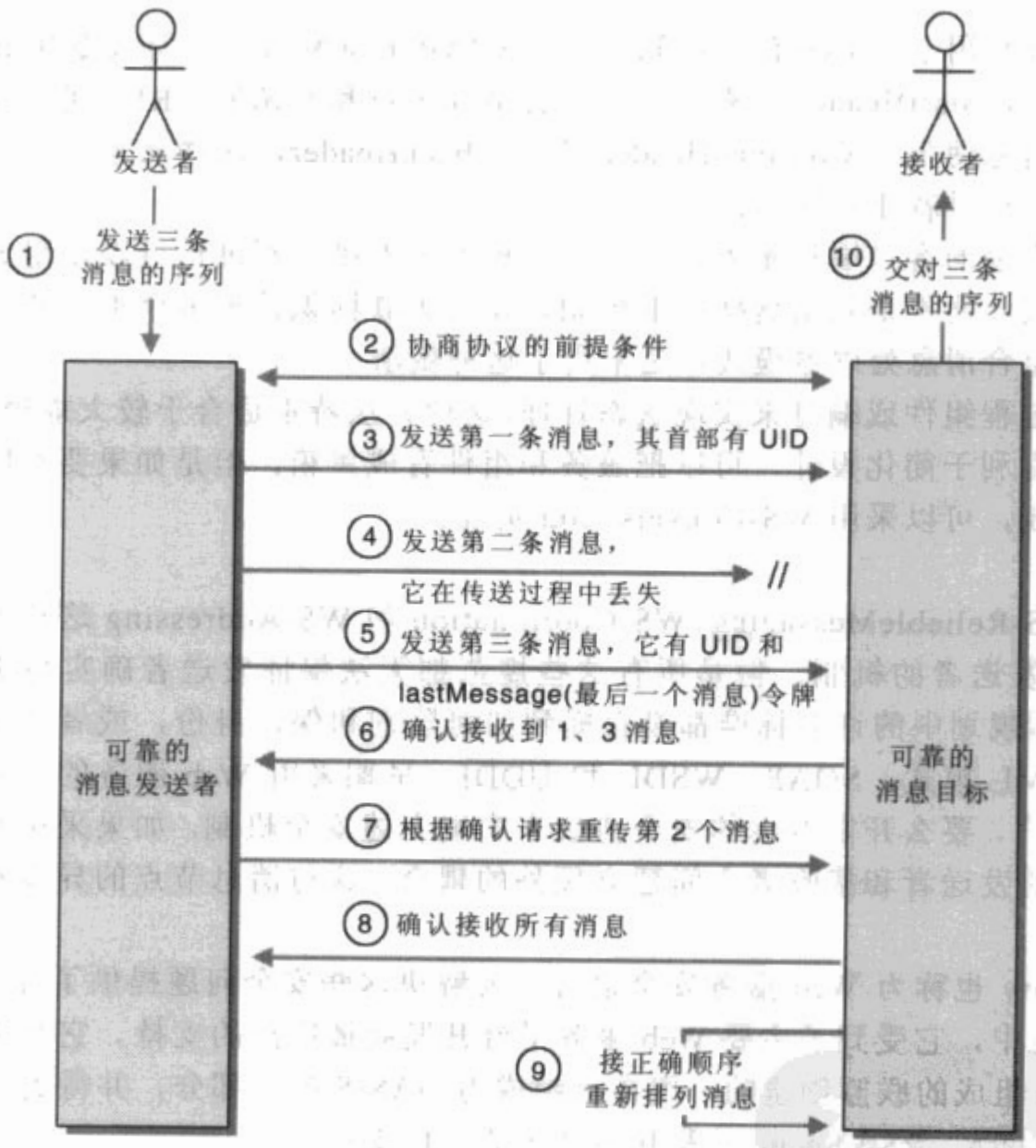
为了保证一次且仅一次的消息行为，WS-Reliability 提供了一种串行化机制，应用可以根据需要来使用。不同的消息分成小组，每个组有相同的组标识符（组 ID），但是每个消息在 SOAP 首部中有自己的序列号，这样的话，在把消息发送给应用之前，接收者就能重新排列消息的顺序。

WS-Reliability 规范可以从 http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsrm 获得。

WS-ReliableMessaging 规范描述了类似的协议，这样即使在程序出现故障的情况下，消息也能在分布式应用之间可靠地传输。该协议采用独立的方式描述，即不依赖于任何特定技术，可以用各种网络传输技术和绑定技术来实现。这个规范也确实包含了一个专门针对 SOAP 的绑定。WS-ReliableMessaging 由 BEA、IBM 和 Microsoft 提出，但还没有提交

给标准组织。

WS-ReliableMessaging 的操作原理与 WS-Reliability 一样。它使用类似的确认、回调和标识符，同样使用了持久的消息缓存，并提供了详细的出错消息。它能保证消息按 4 种基本的传输原则传送：最多一次、至少一次、恰好一次，以及按顺序传送。以下是一个通过 WS-ReliableMessaging 进行顺序消息传输的例子。



WS-ReliableMessaging 通过一系列的确认回调提供了可靠的顺序传输，它依赖于插入到 SOAP 首部元素中的唯一 ID

这两个可靠性规范相互竞争，其关键区别在于 WS-ReliableMessaging 使用了其他重要的 Web 服务规范，如 WS-Security 和 WS-Addressing。实际上，这意味着 WS-ReliableMessaging 在描述自己的规范时使用了其他标准中的专用术语，而 WS-Reliability 虽然支持同样的特性，但是没有采用其他标准规定的惯用形式。

有关 WS-ReliableMessaging 的详细信息可查阅 http://dev2dev.bea.com/technologies/webservices/ws-reliablemessaging.jsp 获得。

WS-Conversation

Web 服务会话（WS-Conversation）指定了管理接收者和发送者之间有状态的异步消息交换协议，这些接收者和发送者通常位于两个不同的 SOAP 端点上。该协议并没有采用封装的业务过程组件来管理多个伙伴之间有状态的消息交换，相反，它提供了更简单的方法来实现一个客户端和一个服务之间有状态的异步消息交换（其中客户端也可能是一个服务）。

该协议充分利用了 SOAP 首部机制，会随 SOAP 消息发送标识符或令牌 ID。开始进行有状态的会话时，StartHeader 元素用于提供会话标识符和回调的 URI。属于同一会话的后续请求和应答消息使用了 ContinueHeader 和 CallbackHeader，这些元素包含了相同的标识符，即在会话启动时就建立的标识符。

尽管为跨平台的客户端和服务建立公共的机制很不错，不过也可以使用 SOAP 首部机制实现这些模式，而不是利用这些标准机制。使用关联标识符把多个消息映射到同一个会话的聚合器和组合消息处理器模式正是采用了这种做法。

可以通过过程组件或编排来实现状态管理，这种方法特别适合于较大规模的集成项目，使用这种方法有利于简化设计，可以把服务和组件看成黑箱，但是如果要对服务和客户端进行更多的控制，可以采用 WS-Conversation 实现。

WS-Security

尽管像 WS-ReliableMessaging、WS-Coordination 和 WS-Addressing 这样的标准提供了各种识别消息发送者的机制，但是所有这些规范都无法保证发送者确实与所声明身份一致。WS-I 基本规划中的许多标准都没有提到如何鉴别和保证身份，或者如何维护消息的完整性，如 XML 模式、SOAP、WSDL 和 UDDI。早期采用 Web 服务的人要么把服务提供给所有的用户，要么开发专用的安全协议来实现上述安全机制。如果采用专用的私有方法，会在消息的发送者和接收者之间建立额外的耦合，这与消息节点的异步和松耦合特性背道而驰。

WS-Security 也称为 Web 服务安全语言，为解决这些安全问题提供了标准的方法。在所有 WS-*规范中，它受到了主要 Web 服务平台开发商最广泛的支持。它是由 Microsoft、IBM、Verisign 组成的联盟创建的，现在已经成为 OASIS 的一部分，并得到了 Sun、BEA、Intel、SAP、IONA、RSA Security 和其他公司的可信支持。

WS-Security 并没有提出新的安全技术，而只是把 SOAP 和已有的安全技术结合起来。它提供了一种可扩展的通用方法，把安全令牌与 SOAP 消息相关联，并能把这些令牌传播给 SOAP 端点。它还指定了对二进制安全令牌编码的标准方法，如在 SOAP 消息中采用数字证书和 Kerberos 票据。它并没有规定专门的协议，而是为实现所有的安全协议建立了一组通用的机制，把各种各样的信任域、签名格式和加密技术结合起来。尽其所能地使用了数字证书、摘要的和安全技术实现，如 PKI、Kerberos、SSL 等。基于 WS-Security，可以把常用的 Internet 安全技术应用到 SOAP 端点中。

除了验证发送者的身份，WS-Security 还可保护消息的完整性——也就是说，防止消息在网络传送过程中被中间人破坏——它是由扩充的 W3C XML 签名标准和 XML 加密标准提

供的。当没有中间用户和第三方服务参与时，通常可以使用 HTTPS 来保护传输中的 SOAP 消息。

安全模型指定 SOAP 消息发送者要做出一系列的声明，如发送者的身份、组、权限等。这些声明集中到签名的安全令牌中。消息的接收者负责验证这些声明。令牌和签名位于 SOAP 首部块中，具体位于 WS-Security 名字空间的 security 首部元素下。当接收者验证声明时，只会出现两种类型的错误：一种是不支持的错误，它表示端点不能支持指定的令牌或加密算法；另一种是失败错误，它代表了余下的大多数错误，包括所有与非法令牌和签名相关的错误。该规范并没有强制要求报告失败错误，因为错误可能是由攻击造成的。报告错误时，会采用规范中定义的错误代码和 SOAP 故障消息格式。

WS-Security 规范可以从共同编写此规范的开发商的网站上获得，包括 IBM 网站：http://www-106.ibm.com/developerworks/library/ws-secure/。

WS-Addressing、WS-Policy 和其他 WS-*规范

还有许多 WS-*规范得到了不同程度支持和采纳。其中一些规范涉及的范围很小，如 WS-Addressing，它定义了一些 XML 元素，从而可以在消息中标识 Web 服务端点。提出该规范的目的是为了支持通过中间组件的消息传递，如端点管理器、代理、防火墙和网关，使它们能采用一种中立的传输方式。基本说来，WS-Addressing 为表示消息来源（“From:”）及消息去向（“To:”）提供了标准方法。基于这个规范，可以把基于 SOAP 的 Web 服务插入到接收表解决方案中。由于它提供了方法来设置应答消息的去向，因此注定会成为解决返回地址相关问题的标准。

作为提供服务的元数据的一种方法，Web 服务策略框架（WS-Policy）提供了描述 Web 服务策略的语法。这些策略包括服务需求、参数选择、服务性能和服务质量等元数据。对应的 Web 服务策略声明语言（WS-PolicyAssertions）指定了一种方法来声明一个消息或服务端点支持某个特定的策略。这涉及调查 WS-Policy 声明，以寻找所需的特定策略。最后，Web 服务策略附件（WS-PolicyAttachment）描述了这些策略标准如何与已有的 Web 服务技术对应，它指定了如何把策略表达式与 WSDL 类型定义和 UDDI 实体关联，并定义了如何把具体实现特定的策略与全部（或部分）WSDL portType 关联。

随着开发商争相为应用开发人员采用的具体实现建立规范，新的 WS-*规范正不断涌现，使 Web 服务不断向前发展。明智的应用开发人员应该关注这些有意思的标准，并在需要时精心做出选择，不过一定要记住，对设计基于消息的应用来说，模式才是最重要的，而不是具体的实现。新兴的 WS-*标准对实现可互操作的企业消息传递系统来说并不是必需的，当它们成为集成的绊脚石时，就应该大胆地把它们丢掉，除非这些标准已走向成熟。

14.1.9 结论

在最佳情况下，标准扩展了设计模式的应用领域，能确保同一模式的不同实现版本相互兼容。人们正在努力通过 Web 服务标准扩展消息传递模式。其中许多标准针对的是工作流组件的行为以及如何组合，这些工作流组件称为业务过程组件。像 BPEL、WSCI 和 WS-*规范等标准可以解决本书提出的许多问题，并以这种模式语言实现了我们介绍的许多模式。

但是，新兴的标准偶尔也会相互冲突，使人困惑。它们尚未成为成熟的标准。在使用这些标准时，应用开发人员不要陷入标准的泥潭中，应该把注意力集中到身边的特定用例上。要注意一个标准是如何解决问题的，如果一个模式有用，能为你的实现提供帮助，就应该采用那些惯用的模式实现，而不要考虑该方法是否得到所有开发商产品的支持。开发人员也可以向标准组织提供反馈信息，向标准提出挑战和批评，让标准真正用于实践，而不是学术界或开发商的纸上谈兵。随着标准走向成熟，明智的架构师可以考虑如何使用标准进一步扩展企业集成解决方案，降低风险和成本，从而能应对更复杂的局面。

参考文献

[Alexander]

A Pattern Language: *Towns, Buildings, Construction*, Christopher Alexander, Sara Ishikawa, & Murray Silverstein; Oxford University Press 1977, ISBN 0195019199

在所有与模式相关的著作中，这也许是引用最多的一本书了。实际上我就利用 Alexander 提出的许多模式设计了一栋作为课程示范的海滩住宅。值得注意的是，Alexander 希望对体系结构和设计进行分解，形成一组可组合的模式，而这种想法来自他对数学的研究。他发表过一篇名为“形式合成的说明”的论文，其中引用了他用 IBM 7090 汇编代码开发的一组程序。因此，Alexander 的模式能在软件领域取得巨大成功并不完全是巧合。

[Alpert]

The Design Patterns Smalltalk Companion, Sherman Alpert, Kyle Brown, & Bobby Woolf; Addison-Wesley 1998, ISBN 0201184621

这本书再次审视了 GoF 所提出的“设计模式”[GoF]，如果开发人员使用带公共类库的编程环境，并且所用的编程环境在有垃圾回收机制的虚拟机上运行，就很适合阅读这本书。这里讨论的编程环境是 Smalltalk，但是书中提出的许多见解同样适用于 Java 和.NET/C#。

[Box]

Essential.NET, Volume 1: *The Common Language Runtime*, Don Box; Addison-Wesley 2002, ISBN 0201734117

其中全面介绍了 CLR 的内部工作原理。

[BPEL4WS]

Business Process Execution Language for Web Services, Version 1.0, BEA, IBM, Microsoft; July 31, 2002, *http: //www.ibm.com/developerworks/webservices/library/ws-bpel1/*

BPEL4WS 1.0 规范。

[CoreJ2EE]

Core J2EE Patterns: Best Practices and Design Strategies(2nd edition), Deepak Alur, John Crupi, & Dan Malks; Prentice Hall PTR 2003, ISBN 0131422464

一本非常好的书，其中介绍了面向 Java 的企业应用架构模式。

[CSP]

“Communicating Sequential Processes,” C.A.R.Hoare; *Communications of the ACM*, 1978

要获得这篇论文的完整版，必须有 ACM 在线资源库的访问权限。

[Dickman]

Designing Applications with MSMQ, Alan Dickman;Addison-Wesley 1998, ISBN 0201325810

这本书有专门的一章介绍“消息传递问题的解决方案”，包括关联、事件驱动的消费者和对象串行化与逆串行化。令人遗憾的是，这本书出版的时间太早，这说明其中的例子全部都使用了带 COM 的 Visual Basic 或 C++。

[Douglass]

Real-Time Design Patterns, Bruce Powel Douglass;Addison-Wesley 2003, ISBN 0201699567

这本书证明模式可跨领域使用。Douglass 提出的一些可靠性模式在企业消息传递中非常有用。

[EAA]

Patterns of Enterprise Application Architecture, Martin Fowler; Addison-Wesley 2003, ISBN 0321127420

这是一本介绍应用体系结构模式的最全面的书。尽管它包含了 51 种模式之多，却仍然是一本既浅显有趣，又不失技术严谨性的书。

[EJB 2.0]

Enterprise JavaBeans Specification, Version 2.0, Sun Microsystems;August 14, 2001, *http://java.sun.com/products/ejb/docs.html*

EJB 2.0 规范。

[Garlan]

Software Architecture: Perspectives on an Emerging Discipline, Mary Shaw & David Garlan;Prentice Hall 1996, ISBN 0131829572

这本书中有一章专门介绍体系结构样式，其中包含管道和过滤器。

[GoF]

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides;Addison-Wesley 1995, ISBN 0201633612

在有关模式的著作中，它的引用率达到了第二位。

[Graham]

Building Web Services with Java: Making Sense of XML, SOAP and UDDI, Steve Graham, Simon Simeonov, Toufic Boubez, Glen Daniels, Doug Davis, Yuichi Nakamura, & Ryo Nyeama;SAMS Publishing 2002, ISBN 0672321815

一本非常好的书，介绍了如何把 Java 和 Web 服务结合起来。

[Hapner]

Java Messaging Service API Tutorial and Reference, Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, & Kim Haase;Addison-Wesley 2002, ISBN 0201784726

详细介绍了 JMS 的工作原理，这本书的作者参与了规范的编写。

[Hohmann]

Beyond Software Architecture: Creating and Sustaining Winning Solutions, Luke Hohmann; Addison-Wesley 2003, ISBN 0201775948

Luke 提醒我们，体系结构方面的许多设计决策并不只是由技术驱动，而且会由商业决策、版权计划和其他大量外部因素决定。

[JMS]

Java Message Service(JMS), Sun Microsystems;2001-2003, *http://java.sun.com/products/jms/*

Java 消息服务 API，Java 2 企业版（J2EE）平台的组成部分。

[JMS 1.1]

Java Message Service (the Sun Java Message Service 1.1 Specification), Sun Microsys- tems; April 12, 2002, *http://java.sun.com/products/jms/docs.html*

JMS 1.1 版规范。

[JTA]

Java Transaction API(JTA), Sun Microsystems;2001-2003, *http://java.sun.com/products/jta/*

Java 事务 API, Java2 企业版（J2EE）平台的组成部分。

[Kahn]

"The Semantics of a Simple Language for Parallel Programming, "G.Kahn, *Information Processing 74: Proc.IFIP Congress 74*, North-Holland Publishing Co., 1974

[Kaye]

Loosely Coupled: The Missing Pieces of Web Services, Doug Kaye;RDS Press 2003, ISBN 1881378241

对 Web 服务的一种新看法。这本书并不只是提供一堆 API，从中我们可以了解面向服务的体系结构的核心工作原理，全书的介绍独立于具体的技术，不包含任何专业术语。对于喜欢使用 SOAP 调用的开发者来说，这本书可能太高深了，但是如果你作为一名技术管理人员和架构师，需要向非技术人员解释相关的概念，那这本书正好适合你。

[Kent]

Data and Reality, William Kent;1stBooks 2000, ISBN 1585009709

这是一本经典的著作（原版是 1978 年发行的），它告诉我们为什么在计算机系统中实现真实建模如此困难。

[Lewis]

Advanced Messaging Applications with MSMQ and MQSeries, Rhys Lewis;Que 2000, ISBN 078972023X

[Leyman]

Production Workflow: Concepts and Techniques, Frank Leyman & Dieter Roller; Prentice- Hall PTR 1999, ISBN 0130217530

[MDMSG]

Multiple-Destination Messaging, Microsoft;February 2003, *http://msdn.microsoft.com/library/en-us/msmq/msmq_about_messages_8aqv.asp*

探讨 MSMQ 3.0 提供的新特性，可以把消息发送给多个目标。

[MicroWorkflow]

" Micro-Workflow:A Workflow Architecture Supporting Compositional Object-Oriented Software Development," Dragos Manolescu;University of Illinois 2000, *http://micro-workflow.com/PhDThesis/phdthesis.pdf*

[Monroe]

"Stylized Architecture, Design Patterns, and Objects," Robert T.Monroe, Drew Kompanek, Ralph Melton, & David Garlan;1996, *http://www-2.cs.cmu.edu/afs/cs/project/compose/ftp/pdf/ObjPatternsArch-ieee97.pdf*

[Monson-Haefel]

Java Message Service, Richard Monson-Haefel & David A.Chappell;O' Reilly 2001, ISBN 0596000685

这也许是目前最著名的 JMS 书籍。

[MQSeries]

WebSphere MQ (formerly MQSeries), IBM;*http://www.software.ibm.com/ts/mqseries*

这是最古老、也最著名的消息传递和集成产品。

[MSMQ]

Microsoft Message Queuing (MSMQ), Microsoft;*http://www.microsoft.com/windows2000/technologies/communications/msmq/*

Windows 2000、Windows XP 和 Windows Server 2003 中内置的消息传递产品。

[PatternForms]

Pattern Forms, *Wiki-Wiki-Web*, Cunningham & Cunningham; last edited on August 26, 2002, *http://c2.com/cgi/wiki ? PatternForms*

给出了常用的一组模式范式和它们的区别。

[PLoPD1]

Pattern Languages of Program Design, James Coplien & Douglas Schmidt(Editors); Addison-Wesley 1995, ISBN 0201607344

第一届 PLoP 大会论文集。论文集中的许多论文为以后的一些著作奠定了基础，如[POSA]。其中包括了 Frank Buschmann 和 Regine Meunier 的"模式体系"，Regine Meunier 的"管道和过滤器体系结构"，以及 Diane Mularz 的"基于模式的集成体系结构"等文章。

[PLoPD3]

Pattern Languages of Program Design 3, Robert Martin, Dirk Riehle, & Frank Buschmann (Editors);Addison-Wesley 1998, ISBN 0201310112.

PLoP 大会的第 3 本书（PLoP、EuroPLoP 等，见 http://hillside.net/conferencesnavigation.htm），包含了构成[POSA2]的基本模式：接受者和连接器、异步的完成令牌和双重检查锁定。它还包含本书作者提出的两个模式：Null 对象和 Type 对象。

[POSA]

Pattern-Oriented Software Architecture, Frank Buschmann, Regine Meunier, Hans Rohnert,

Peter Sommerlad, & Michael Stal; Wiley 1996, ISBN 0471958697

有关体系结构和设计模式的著作。

[POSA2]

Pattern-Oriented Software Architecture, Vol.2, Douglas Schmidt, Michael Stal, Hans Rohnert, & Frank Buschmann; Wiley 2000, ISBN 0471606952

介绍了更多涉及分布式系统和并发问题的模式。

[Sharp]

Workflow Modeling: Tools for Process Improvement and Application Development, Alec Sharp & Patrick McDermott; Artech House 2001, ISBN 1580530214

本书主要介绍工作流建模方面，系统分析员和业务架构师会对这本书很感兴趣。

[SOAP 1.1]

W3C Simple Object Access Protocol(SOAP) 1.1 Specification, World Wide Web Consortium; W3C Note, May 8, 2000, *http://www.w3.org/TR/SOAP/*

SOAP 1.1 版规范。

[SOAP 1.2 Part 2]

SOAP Version 1.2, Part 2: Adjuncts, World Wide Web Consortium;W3C Recommendation, June 24, 2003, *http://www.w3.org/TR/soap12-part2/*

SOAP1.2 版规范的“附录”。

[Stevens]

TCP/IP Illustrated, Volume 1: The Protocols, W.Richard Stevens; Addison-Wesley 1994, ISBN 0201633469

[SysMsg]

“System.Messaging namespace,” .NET Framework, version 1.1, Microsoft; *http://msdn.microsoft .com/library/en-us/cpref/html/cpref_start.asp*

[Tennison]

XSLT and XPath on the Edge, Jeni Tennison; John Wiley & Sons 2001, ISBN 0764547763

[UML]

UML Distilled: A Brief Guide to the Standard Object Modeling Language(3rd edition), Martin Fowler; Addison-Wesley 2003, ISBN 0321193687

学习 UML 图的极佳资料，介绍 UML 图的形式和含义。

[UMLEAI]

“UML Profile for Enterprise Application Integration,” Object Management Group; 2002, *http://www.omg.org/technology/documents/modeling_spec_catalog.htm*

[Wright]

TCP/IP Illustrated, Volume 2: The Implementation, Gary R.Wright & W.Richard Stevens; Addison-Wesley 1995, ISBN 020163354X

[WSAUS]

"Web Services Architecture Usage Scenarios," World Wide Web Consortium;W3C Working Draft, May 14, 2003, *http://www.w3.org/TR/ws-arch-scenarios/*

有关如何使用 Web 服务以及规范要实现哪些需求， W3C 提供了最新的想法。

[WSDL 1.1]

Web Services Description Language (WSDL)1.1, World Wide Web Consortium; W3C Note March 15, 2001, *http://www.w3.org/TR/wsdl*

WSDL 1.1 版规范。

[WSFL]

Web Services Flow Language (WSFL) 1.0, IBM; May 2001, *http://www3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf*

WSFL 1.0 版规范。

[WSMQ]

WebSphere MQ Using Java(2nd edition), IBM; October 2002, *http://publibfp .boulder.ibm.com /epubs /pdf/csqzaw11.pdf*

Java 程序设计人员要使用 IBM 的 WebSphere MQ 消息传递产品[MQSeries]，可以以此作为开发指南。

[XML 1.0]

Extensible Markup Language(XML) 1.0(2nd edition), World Wide Web Consortium; W3C Recommendation, October 6, 2000, *http://www.w3.org/TR/REC-xml*

XML 1.0 版规范。

[XSLT 1.0]

XSL Transformations(XSLT) Version 1.0, World Wide Web Consortium; W3C Recommendation, November 16, 1999, *http://www.w3.org/TR/xslt*

XSLT 1.0 版规范。

[Waldo]

"A Note on Distributed Computing" (Technical Report *SMLI TR-94-29*), Jim Waldo, Geoff Wyant, Ann Wollrath, & Sam Kendall; Sun Microsystems Laboratories, November 1994, *http://citeseer.nj.nec.com/waldo94note.html*

[Zahavi]

Enterprise Application Integration with CORBA, Ron Zahavi; John Wiley & Sons 1999, ISBN 0471327204