计 算 机 科 学 丛 书

Web服务
原理和技术

(荷) Michael P. Papazoglou 著 龚 玲 张云涛 译



Web Services
Principles and Technology

机械工业出版社
China Machine Press

Web服务 原理和技术

"本书是我所见过的有关Web Service讨论最详尽的书籍之一。本书涵盖了构建面向服务的体系结构所涉及的方方面面，包括一整套概念体系、原理、支撑技术、必需的基础架构以及相关的标准等。我极力推荐此书。"

——《企业服务总线》作者Dave Chappell

"本书由Web Service领域最著名的专家所著，对学术界人士和实际工作者都极具价值。该书结构合理，可以作为Web Service技术领域的一本权威指南。"

—— IBM T. J. Watson研究中心软件系统部经理Francisco Curbera

Web Service代表了下一代的基于Web的技术。通过Web Service，企业应用之间可以更好地实现相互通信和集成，因此对于业务发展和软件开发都具有深远的影响。

作者对Web Service进行了全面的探讨，主要介绍了Web Service的本质、基本概念、原理与方法，并提供了该领域的大量相关信息。本书既适用于计算机科学专业的学生，同时也适用于希望了解该领域的专业技术人员。

本书特色

- 采用由浅入深的螺旋式方式，在读者已有知识的基础上逐步引入一些比较复杂的内容。
- 采用大量的实例来阐述相关理论的实际应用。
- 自测题、各种使用技巧和提示贯穿本书。

作者简介

Michael P. Papazoglou 现任荷兰提耳堡大学计算机科学系的系主任及INFOLAB/CRISM实验室的主任。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站：http://www.hzbook.com

网上购书：www.china-pub.com

ISBN 978-7-111-28414-7

9 787111 284147

定价：58.00元

计 算 机 科 学 丛 书

Web服务
原理与技术

(荷) Michael P. Papazoglou 著 龚玲 张云涛 译

Web Services
Principles and Technology

机械工业出版社
China Machine Press

本书是有关 Web Service 讨论最详尽的书籍之一。全书涵盖了构建面向服务的体系结构所涉及的方方面面，包括一整套概念体系、原理、支撑技术、必须的基础架构以及相关的标准等。

本书既适用于计算机科学专业的学生，同时也适用于希望了解该领域的专业技术人员。

Michael P. Papazoglou：Web Services：Principles and Technology（ISBN 978-0-321-15555-9）.

Copyright © 2008 by Pearson Education Limited.

This translation of Web Services：Principles and Technology（ISBN 978-0-321-15555-9）is published by arrangement with Pearson Education Limited.

All rights reserved.

本书中文简体字版由英国 Pearson Education 培生教育出版集团授权出版。

版权所有，侵权必究

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2007-4672

图书在版编目（CIP）数据

Web 服务：原理和技术/（荷）帕派佐格罗（Papazoglou，M. P.）著；龚玲 等译. —北京：机械工业出版社，2009.12

（计算机科学丛书）

书名原文：Web Services：Principles and Technology

ISBN 978-7-111-28414-7

Ⅰ. W…　Ⅱ. ①帕…　②龚…　Ⅲ. 计算机网络－程序设计　Ⅳ. TP393.09

中国版本图书馆 CIP 数据核字（2009）第 175252 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：陈佳媛

北京瑞德印刷有限公司印刷

2010 年 1 月第 1 版第 1 次印刷

184mm×260mm · 23.25 印张

标准书号：ISBN 978-7-111-28414-7

定价：58.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@ hzbook. com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037



华章教育

前言

互联网经济正在发生改变，关注焦点从原先的门户和网站流量变为复杂的自动化电子交易。我们已经开始着手一种新的 Web 计算方式——Web Service。作为新一代的 Web 技术，Web Service 是面向服务计算模式的一部分，用于互联网上的信息交换。人们寄希望于 Web Service 能彻底变革目前的分布式软件应用的开发和部署流程。

目前应用集成的主要途径是进行简单的信息交换。人们寄希望 Web Service 能超越这一点，从而实现应用服务的访问、编程与集成，并且无论这些应用服务是被封装在老的应用还是新的应用中。相比以前，开发人员能够使用内部已有的软件(即使这些软件原先属于遗留系统)，并将它们与位于远程网络上的外部组件综合起来，创建复合应用系统解决方案，从而能够更快地动态扩展应用系统，这是 Web Service 计算模式的一个重要经济效益。这意味着软件开发群体的各社会经济组织将发生根本性的变化，从而进一步提高软件开发的效率与生产率，企业可更快地向社会提供新的产品与服务。

Web Service 技术的远景目标是世界范围的服务合作，将各类应用组件很轻松地装配成服务网络，通过服务之间松散的耦合创建动态的跨多个组织和各类计算平台的业务流程和敏捷应用。因此，Web Service 技术将有助于现代社会的一体化，尤其在一些虚拟领域，诸如动态业务、健康、教育、政府服务等。

随着 Web Service 技术的不断成熟以及基础设施的不断完善，许多公司已经开始将一些重要的业务流程打包成 Web Service，并在互联网上向外提供服务。这意味着，在未来绝大多数电子商务应用将能通过一系列的 Web Service 来实现，这些 Web Service 彼此交互并处理相互间的请求。例如，一个应用中的服务可决定是否参与另一个企业中的服务。

对于一些很新的技术，诸如 Web Service，依然有许多基础工作要做。事实上，目前开发复杂的 Web Service 应用依然是一个愈加复杂的任务。然而，一旦基础性的工作和基础设施到位，事件将会变得非常简单。应用 Web Service 技术仅需简单地暴露和复用核心业务功能，并将相关的业务流程以新的方式组合起来，从而创建出新的增值方案。这将减少复杂性和各种开销，增加灵活性，并增强运作效率。基于以上这些原因，Web Service 计算模式的普及率预计将会快速上扬。鉴于 Web Service 能够解决花费巨大的、棘手的业务与技术难题，因此相比于以前的应用技术，Web Service 将会渗透到更多的应用方案中。

需要掌握 Web Service 技术的职业不断增加，是 Web Service 不断发展的后果之一。这使得越来越多的学术研究人员和职业技术人员希望了解 Web Service 的概念、原理与技术基础。因此，本书将全面地、系统地、针对性地讨论 Web Service 的原理、基本知识、有争议的问题以及相关技术，尤其是指明这个领域的发展现状以及未来可能的发展方向，从而满足社会各界对于 Web Service 技术的需求。

本书的特点

Web Service 的主题非常广泛、非常复杂，涉及许多概念、协议和技术，而且它们源自不同的

学科，诸如分布式计算系统、计算机网络、计算机体系结构、中间件、软件工程、编程语言、数据库系统、安全性和知识表示等，并且它们以各种错综复杂的方式组合在一起。此外还需要一些处理业务流程与组织的新技术，这些新技术既要发现企业存在的问题，又要在实际应用中解决这些问题。

本书的内容涉及众多的文献和资料。为了将各色主题糅合在一起，我阅读了大量的文献和资料，并对它们进行整合，同时采用和发挥了一种综合方法。该方法基于我对资料的分析，并发现迄今为止尚被忽视的一些工作领域的联系。我尽量使我的研究符合当前苛刻的标准，并努力使本书具有可读性，从而使得读者能够完全地了解 Web Service 技术。读者不仅能学到体系结构中的清晰的概念、技术、协议和标准，而且可以领会 Web Service 各部分组合起来的完整的状态。

最近几年，关于 Web Service 标准与编程的优秀的书籍陆续出版。我的意图并不是撰写一本类似的书籍。本书既不是关于 Web Service 标准，也不是关于 Web Service 编程技术，而是讲授 Web Service 的概念和原理，以及构建 Web Service 的技术。本书的特点在于主题的广度、方法以及针对性。本书的主要目标为：

- 介绍了解 Web Service 所需的坚实基础。
- 重点强调获取深层知识、洞察力，以及了解支撑 Web Service 的概念、原理、机制和方法学，而不是 Web Service 编程或实现。
- 帮助读者深入理解支撑 Web Service 模式的每一项技术，以及这些技术如何支持 Web Service 模式。

本书的另一个重要特点是读者群体广泛。读者无须太强的技术背景，然而即使对于有经验的读者，本书的具体内容也足够丰富而具有挑战性。本书重点阐述了 Web Service 的理论和技术支撑。

为了确保读者完全理解本书介绍的内容，在书中将以多种形式，诸如通俗的描述、直观的例子、模型的抽象、具体的 XML 以及相关的 Web Service 标准，用于阐述 Web Service 概念、技术和各类问题。为了更好地加强读者对所介绍内容的了解、掌握，本书使用了大量的图例和真实的例子。本书首先介绍了一些简单的概念以及入门技术，随着所讨论的内容的深入，在此基础上，进一步阐述了一些新的概念，以帮助读者更好地理解和掌握 Web Service 的核心概念。对于分散在一些文献中的资料和最近的发展，本书进行了汇总。

运行样例

本书最显著的特点之一是能够通过运行样例来检验 Web Service 的相关技术与标准，这些样例将以典型的订单管理为场景。对于理论阐述和概念解释，这些样例可增强读者的理解与洞悉。在本书中，我采用了渐进式的论述方式。随着所运行的样例，逐渐深入地讨论 Web Service 技术与标准。

读者对象

由于本书深入讨论了 Web Service 的许多重大问题、主题，以及 Web Service 底层技术，因此本书读者群体广泛。本书既可作为教材，也可作为参考书。本书的编排力争突出重点。在当前的许多 Web Service 文献资料中，充斥着大量的术语、标准以及编程技巧。有些读者可能会对这样的文献资料感到恐惧，但阅读本书时，他们就不必担心了。更具体地说，本书主要针对本科生、研究生、研究人员以及专业技术人员，诸如信息技术规划人员、架构师、软件分析人员、设计

人员、开发者、软件工程师和编程人员等。总的来说，本书面向想深入了解 Web Service 的原理、技术以及具体知识的读者和那些希望了解在电子商务等复杂应用中如何使用 Web Service 的读者。同样，业务策划师、业务流程工程师以及业务架构师也适于阅读本书。随着业务需求与信息技术发展这两者之间的分界线的逐渐模糊，我希望本书也适合业务人员阅读，特别是那些新培养的专业人员与学生。本书中涉及的许多问题都与业务应用中的软件解决方案的设计直接相关。

通过阅读本书，读者将能深入学习 Web Service 与企业计算方面的知识，并可掌握如何在 B2B 业务中应用这些知识。在 Web Service 的原理、议题、标准以及技术支撑等方面，本书都进行了全面、深入的论述。本书大量使用了图表、实例，以及真实的业务场景，同时很少涉及特定的平台与厂商。

本书的辅助教学网站

本书的专用网站提供了一些教学辅助材料，包括常规的幻灯片、习题答案以及其他的一些出版物和资料等。网站的地址为：www. pearsoned. co. uk/papazoglou。

作者简介

Michael P. Papazoglou 是荷兰提耳堡(Tilburg)大学的计算机科学教授以及 INFOLAB/CRISM 实验室主任。他也是意大利特兰托(Trento)大学的名誉教授。在 1991 ~ 1996 年期间，他是澳大利亚布里斯班的昆士兰技术大学(QUT)信息系统学院的正教授和院长。他也拥有澳大利亚国立(Australian National)大学、德国科布伦茨(Koblenz)大学、德国哈根远程教育大学(Hagen Fern Universitat)的高级学术职务。在 1983 ~ 1989 年期间，他是圣奥古斯丁德国国立研究中心计算机科学首席科学家。

Papazoglou 教授在多个国际委员会任职。他是九个国际科学期刊的编委，并是麻省理工学院信息系统系列丛书的共同主编。他曾是计算机科学领域的众多知名国际会议的主席，诸如数据工程国际会议(International Conference on Data Engineering，简称 ICDE)、分布式计算系统国际会议(International Conference on Distributed Computing Systems，简称 ICDCS)、数字图书馆国际会议(International Conference on Digital Library，简称 ICDL)、协同信息系统国际会议(International Conference on Cooperative Information Systems，简称 CoopIS)、实体/关系建模国际会议(International Conference on Entity/Relationship Modeling)等。他也是国际协同信息系统会议及最近的面向服务计算国际会议(International Conference on Service Oriented Computing，简称 ICSOC)的创始人。他编著了 15 本书籍，并在科学期刊和国际会议上发表了 150 多篇论文。他最近撰写的书籍是关于电子商务的组织与技术基础的，由约翰威利父子出版社(John Wiley & Sons)于 2006 年 4 月出版。他的研究曾经得到或正在得到欧盟、澳大利亚研究理事会、日本学术振兴会、欧洲和澳大利亚科学技术部的资助。他是电气和电子工程师协会(IEEE)计算机科学学部的金核心成员、杰出访问学者。

致 谢

撰写此书在许多方面都面临着挑战，但是对此我已经计划很久了。花费三年中的大半时间来完成此书是一项令人畏惧的工作。在这个过程中，我进行了大量的研究，查阅了大量的资料，并与这个领域中的许多人员与专家进行了讨论，他们也提出了宝贵的建议。这也开阔了我的学术眼界，拓展了我的知识。在撰写本书的过程中，激发了我的许多灵感，并对这一主题有了全面的理解。

本书的完成离不开许多人的帮助。借此机会衷心地感谢他们。

本书的评阅专家提出了许多中肯的、富有建设性的建议，从而帮助我更好地提高本书的质量。他们在我撰写本书的过程中扮演了非常重要的角色。我特别想要感谢下列人员，他们花费了大量时间评阅了终稿中的许多关键章节，并提出了具体的改进措施，他们是：佛吉尼亚理工大学（Virginia Tech）的 Athman Bouguettaya、维也纳技术大学（Vienna University of Technology）的 Schahram Dustdar、波茨坦大学（University of Potsdam）的 Tiziana Margaria、太阳微系统公司（Sun Microsystems）的 Monica Martin、德克萨斯州立大学（Texas State University）的 Anne Ngu、悉尼技术大学（Sydney University of Technology）的 Robert Steele 以及柏莱斯·帕斯卡尔（Blaise Pascal）大学的 Farouk Toumani。

我要感谢那些在书稿的不同阶段审阅了一些章节并给出了许多有益建议的评阅者，他们是：克利特大学（University of Crete）的 Gregory Antoniou、Sonic 软件公司的 Dave Chappell、Attachmate 公司的 Jean Jacques Dubray、悉尼技术大学的 George Feuerlicht、IBM TJ Watson 研究中心的 Heiko Ludwig、提耳堡大学（Tilburg University）的 Jan van den Heuvel 和伦敦城市大学（City University in London）的 Andrea Zisman。

对于那些提供我反馈信息者或者对如何改进书稿提出建议者，我也非常感激，他们是：冈宁根大学（University of Groningen）的 Marco Aiello、惠普（HP）实验室的 Fabio Casati、IBM TJ Watson 研究中心的 Paco Curbera、特兰托大学（University of Trento）的 Vincenzo d'Andrea、IBM TJ Watson 研究中心的 Asit Dan、哈根远程教育大学（Fern Universitat in Hagen）的 Bernd Kramer、斯图加特大学（Stuttgart University）的 Frank Leymann、特兰托大学的 Maurizio Marchese、多伦多大学（University of Toronto）的 John Mylopoulos 以及 IRST 研究中心的 Paolo Traverso。

我也衷心感谢提耳堡大学的 Vassilios Andrikopoulos、Benedikt Kratz 和 Bart Orriens，斯图加特大学的 Stefan Pottinger 和 Thorsten Scheibler，特兰托大学的 Michele Manchioppi 和 Manuel Zanoni，他们对本书中的一些练习提供了一些提示和解答。

最后，我要感谢整个 Pearson 团队，他们完成了杰出的工作。尤其要感谢我在 Pearson 原先的编辑 Kate Brewin 和 Owen Knight，感谢他们的不断鼓励、坚持不懈以及对我的耐心。本书花了很长时间才完成。我真挚地感谢 Simon Plumtree 和 Joe Vella，他们在过去一直指导我解决本书中的许多问题，并给出了许多宝贵的建议与帮助。

Michael P. Papazoglou

目 录

出版者的话
前言
致谢

第一部分 基本原理

第 1 章 Web Service 基础 …… 1
1.1 引言 …… 1
1.1.1 Web Service 是什么 …… 2
1.1.2 Web Service 的典型场景 …… 3
1.2 “软件即为服务”的理念 …… 5
1.3 Web Service 的完整定义 …… 6
1.4 Web Service 的特性 …… 7
1.4.1 Web Service 的类型 …… 7
1.4.2 功能属性和非功能属性 …… 9
1.4.3 状态属性 …… 9
1.4.4 松耦合 …… 10
1.4.5 服务粒度 …… 11
1.4.6 同步 …… 11
1.4.7 良定义 …… 11
1.4.8 服务的使用环境 …… 12
1.5 服务接口和实现 …… 12
1.6 面向服务的体系结构 …… 14
1.6.1 SOA 中进行交互的角色 …… 14
1.6.2 SOA 中的操作 …… 15
1.6.3 SOA：一个涉及综合服务的样例 …… 16
1.6.4 SOA 中的层次 …… 17
1.7 Web Service 的技术架构 …… 21
1.8 服务质量(QoS) …… 23
1.9 Web Service 的互操作性 …… 25
1.10 Web Service 与组件的比较 …… 26
1.11 Web Service 的优与劣 …… 28
1.12 小结 …… 30
复习题 …… 30
练习 …… 31

第二部分 核心基础架构

第 2 章 分布式计算的基础架构 …… 33
2.1 分布式计算与互联网协议 …… 33
2.1.1 互联网协议 …… 34
2.1.2 中间件 …… 37
2.2 客户－服务器模型 …… 38
2.3 进程间通信的特性 …… 39
2.3.1 消息发送 …… 39
2.3.2 消息目的地和 socket …… 40
2.3.3 同步方式的消息发送和异步方式的消息发送 …… 40
2.4 中间件的同步方式 …… 41
2.4.1 远程过程调用 …… 41
2.4.2 远程方法调用 …… 43
2.5 中间件的异步方式 …… 43
2.5.1 消息的存储与转发 …… 44
2.5.2 消息的发布与订阅 …… 45
2.5.3 事件驱动的处理机制 …… 46
2.5.4 点到点排队 …… 47
2.6 请求/应答的消息传送方式 …… 48
2.7 面向消息的中间件 …… 49
2.7.1 集成代理 …… 51
2.7.2 Java 消息服务(JMS) …… 52
2.8 面向事务的中间件 …… 53
2.9 企业应用程序与电子商务的集成 …… 54
2.10 小结 …… 56
复习题 …… 57
练习 …… 57
第 3 章 XML 概览 …… 59
3.1 XML 文档结构 …… 59
3.1.1 XML 声明 …… 60
3.1.2 元素 …… 60
3.1.3 属性 …… 61
3.2 URI 和 XML 命名空间 …… 61

3.3 定义 XML 文档中的结构 …… 63
3.3.1 XML 模式定义语言 …… 63
3.3.2 XML 模式文档 …… 63
3.3.3 类型定义、元素和属性声明 …… 66
3.3.4 简单类型 …… 67
3.3.5 复合类型 …… 67
3.4 XML 模式复用 …… 67
3.4.1 派生的复合类型 …… 67
3.4.2 导入模式与包含模式 …… 69
3.5 文档的导航与转换 …… 74
3.5.1 XML 路径语言 …… 74
3.5.2 使用 XSLT 进行文档转换 …… 75
3.6 小结 …… 76
复习题 …… 77
练习 …… 77

第三部分 核心功能与标准

第 4 章 SOAP：简单对象访问协议 …… 79
4.1 应用程序间的通信与连接协议 …… 79
4.2 SOAP 作为消息传送协议 …… 80
4.3 SOAP 消息的结构 …… 83
4.3.1 SOAP 信封 …… 84
4.3.2 SOAP 头部 …… 85
4.3.3 SOAP 消息体 …… 88
4.4 SOAP 通信模型 …… 89
4.4.1 RPC 类型的 Web Service …… 89
4.4.2 文档(消息)类型的 Web Service …… 90
4.4.3 通信方式与消息交换的模式 …… 92
4.5 SOAP 中的出错处理 …… 92
4.6 基于 HTTP 的 SOAP …… 93
4.7 SOAP 的优缺点 …… 95
4.8 小结 …… 96
复习题 …… 96
练习 …… 96
第 5 章 描述 Web Service …… 98
5.1 为何需要服务描述 …… 98
5.2 WSDL：Web Service 描述语言 …… 99
5.2.1 WSDL 的接口定义 …… 100
5.2.2 WSDL 的实现 …… 104
5.2.3 WSDL 的消息交换模式 …… 110
5.3 使用 WSDL 生成客户端 stub …… 112
5.4 WSDL 中的非功能性描述 …… 114
5.5 小结 …… 114
复习题 …… 115
练习 …… 115
第 6 章 Web Service 的注册与发现 … 117
6.1 服务注册 …… 117
6.2 服务发现 …… 118
6.3 UDDI：统一描述、发现和集成 …… 119
6.3.1 UDDI 数据结构 …… 120
6.3.2 WSDL 到 UDDI 的映射模型 …… 130
6.3.3 UDDI API …… 137
6.3.4 UDDI 模型的查询 …… 139
6.3.5 UDDI 用例模型与部署的多样性 …… 140
6.4 小结 …… 141
复习题 …… 142
练习 …… 142

第四部分 事件通知与面向服务的体系结构

第 7 章 寻址与通知 …… 143
7.1 Web Service 与有状态的资源 …… 143
7.2 Web Service 资源框架简介 …… 144
7.2.1 Web Service 寻址 …… 146
7.2.2 Web Service 资源 …… 149
7.2.3 资源属性 …… 151
7.2.4 资源生命周期 …… 154
7.2.5 服务组 …… 155
7.3 Web Service 通知 …… 155
7.3.1 P2P 通知 …… 156
7.3.2 通知主题 …… 160
7.3.3 代理通知 …… 163
7.4 Web Service 事件 …… 165
7.5 小结 …… 166
复习题 …… 166
练习 …… 166
第 8 章 面向服务的体系结构 …… 169
8.1 软件体系结构是什么 …… 169
8.1.1 系统质量属性 …… 170
8.1.2 体系结构方面的常见议题 …… 171

8.2 SOA 回顾 …… 171
8.3 SOA 中的服务角色 …… 173
8.4 可靠的消息传送 …… 175
8.4.1 可靠的消息传送的定义和范围 …… 176
8.4.2 WS-ReliableMessaging …… 176
8.5 企业服务总线 …… 180
8.5.1 SOA 的事件驱动特性 …… 182
8.5.2 ESB 的关键特征 …… 184
8.5.3 ESB 的集成类型 …… 187
8.5.4 ESB 解决方案中的各要素 …… 188
8.5.5 连接和转换基础架构 …… 194
8.5.6 遗留资产的使用 …… 195
8.5.7 ESB 中的可伸缩性 …… 196
8.5.8 使用 ESB 的集成模式 …… 198
8.6 扩展的 SOA …… 199
8.7 小结 …… 202
复习题 …… 202
练习 …… 202

第五部分 服务组合与服务事务

第 9 章 流程与工作流 …… 205
9.1 业务流程及其管理 …… 205
9.2 工作流 …… 207
9.3 业务流程的集成与管理 …… 209
9.4 跨企业的业务流程 …… 211
9.5 服务组合元模型 …… 213
9.5.1 流模型的理念 …… 213
9.5.2 Web Service 的组合 …… 216
9.6 Web Service 的编配与编排 …… 219
9.7 业务流程执行语言(BPEL) …… 221
9.7.1 BPEL 的结构 …… 221
9.7.2 BPEL 的简单样例 …… 233
9.8 编排 …… 241
9.8.1 编排描述的使用 …… 241
9.8.2 Web Service 编排描述语言 …… 242
9.9 其他的一些提案和语言 …… 244
9.10 小结 …… 244
复习题 …… 245
练习 …… 245
第 10 章 事务处理 …… 247
10.1 什么是事务 …… 247
10.1.1 事务的属性 …… 248
10.1.2 并发控制机制 …… 249
10.2 分布式事务 …… 250
10.2.1 分布式事务体系结构 …… 250
10.2.2 两阶段提交协议 …… 253
10.3 嵌套事务 …… 255
10.3.1 封闭嵌套事务 …… 256
10.3.2 开放嵌套事务 …… 259
10.4 事务型 Web Service …… 262
10.4.1 Web Service 事务的定义和一般特性 …… 263
10.4.2 Web Service 事务的操作特性 …… 264
10.4.3 Web Service 事务的类型 …… 265
10.4.4 评议小组与介入 …… 267
10.4.5 Web Service 事务的状态 …… 269
10.4.6 Web Service 事务框架 …… 270
10.5 WS-Coordination 和 WS-Transaction … 271
10.5.1 WS-Coordination …… 271
10.5.2 WS-Transaction …… 278
10.6 Web Service 组合应用框架 …… 283
10.6.1 Web Service 上下文 …… 284
10.6.2 Web Service 协调框架 …… 285
10.6.3 Web Service 事务管理 …… 286
10.7 小结 …… 287
复习题 …… 288
练习 …… 288

第六部分 服务安全性与策略

第 11 章 安全的 Web Service …… 291
11.1 Web Service 安全性 …… 291
11.1.1 Web Service 面临的安全性威胁 …… 292
11.1.2 对策 …… 294
11.2 网络层的安全性机制 …… 294
11.2.1 防火墙 …… 294
11.2.2 入侵检测系统和漏洞评估 …… 298
11.2.3 安全的网络通信 …… 298
11.3 应用层的安全性机制 …… 303

11.3.1 认证 …… 303
11.3.2 授权 …… 304
11.3.3 完整性与机密性 …… 305
11.3.4 不可抵赖性 …… 306
11.3.5 审计 …… 306
11.3.6 应用层安全性协议 …… 306
11.3.7 安全性基础架构 …… 308
11.4 安全性布局 …… 310
11.5 XML 安全性标准 …… 312
11.5.1 XML Signature …… 312
11.5.2 XML Encryption …… 315
11.5.3 XML 密钥管理规范(XKMS) … 316
11.5.4 安全声明标记语言 …… 318
11.5.5 XML 访问控制标记语言 …… 322
11.6 安全的 Web Service …… 326
11.6.1 Web Service 应用层面临的挑战 …… 326
11.6.2 Web Service 安全性路线图 …… 327
11.6.3 Web Service 安全性模型 …… 329
11.6.4 WS-Security …… 330
11.6.5 安全性策略的管理 …… 339
11.6.6 安全会话的管理 …… 340
11.6.7 信任管理 …… 341
11.6.8 隐私管理 …… 342
11.6.9 联邦身份标识的管理 …… 342
11.6.10 授权管理 …… 343
11.7 小结 …… 344
复习题 …… 344
练习 …… 344
参考文献 …… 346

第一部分 基本原理

第1章 Web Service基础

学习目标

Web Service是一项新技术，能使得运行在不同机器上的不同应用无须借助附加的、专门的第三方软件或硬件，就可相互交换数据或集成。依据Web Service规范实施的应用之间，无论它们所使用的语言、平台或内部协议是什么，都可以相互交换数据。Web Service是自描述、自包含的可用网络模块，可以执行具体的业务功能。Web Service也很容易部署，因为它们基于一些常规的产业标准以及已有的一些技术，诸如XML和HTTP。Web Service减少了应用接口的花费。Web Service为整个企业甚至多个组织之间的业务流程的集成提供了一个通用机制。

阅读完本章的读者将能理解下列关键概念：

- Web Service的性质、主要特征和类别。
- Web Service与应用服务提供者模型以及基于Web的应用的区别。
- 紧耦合与松耦合的概念。
- 有状态服务与无状态服务的概念。
- 面向服务的体系结构的基本概念以及它的主要构件。
- Web Service技术栈，以及如何利用Web Service的标准开发分布式应用。
- 功能性服务与非功能性服务的特征，以及服务质量的概念。

1.1 引言

面向服务的计算是一个新的计算规范，它将服务作为构件，用于支持分布式应用的低成本快速开发。服务是自包含的模块，它们部署在标准的中间件平台上，能够在网络上使用基于XML的技术进行描述、定位、编配和编程。部署在系统上的任何代码段或应用程序组件都能转换为网络上的服务。服务反映了"面向服务"的编程方式。该编程方式将可用的计算资源，例如应用程序或信息系统组件，描述为能够通过标准的、良定义的接口提交的服务，服务能够完成不同的功能。既可以是简单的功能，如响应简单的请求，也可以是复杂的功能，如执行在服务的用户方和提供者之间具有P2P关系的业务流程。服务的构建方式通常独立于它们的使用方式。这就意味着，在服务提供者和服务用户方之间是松耦合的关系。通过发现、调用以及组合网络上的服务即可开发基于服务的应用，而无须构建新的应用程序。

面向服务的计算并不是一个新的技术，而是分布式系统、软件工程、信息系统、计算机语言、基于Web的计算和XML技术的融合。该技术预计对软件构造的所有方面都将产生重要的影响，其对业界的影响范围至少不逊于面向对象的编程。

面向服务计算的基本前提是应用程序不再被视为运行在单个组织内的单一处理。不再按照应用程序的功能来衡量应用程序的价值，而是通过应用程序与周围环境的集成能力来衡量应用程序的价值[Papazoglou 2003]。例如，有些应用程序在编写时并没有与其他应用程序进行集成的意图，对于它们来说，可以使用服务帮助它们集成，也可以利用已有的应用程序的功能构建新的功能。基于相互交互的、可向潜在用户提供良定义接口的服务集合，可以单独开发新的应用程序，这些应用程序通常称为复合应用。面向服务使得合作伙伴的应用之间可以是一种松耦合的关系。在中间件层，松耦合的概念需要面向服务的方式来支撑，面向服务的方式将独立于特定的技术或特定的操作系统。所提供的服务在被允许使用之前，面向服务的模式甚至都不要求任何预先约定的协定。

在面向服务的模型中，可以清晰地区分服务提供者、服务客户端以及服务聚合者。服务提供者提供服务的实现、描述以及相关的技术与业务支持。服务客户端是具体使用服务的终端用户组织。服务聚合者是将多个服务整合成一个新的服务，这个新的服务通常称为业务流程。

服务可以由不同的企业提供，并且服务之间可以在因特网上相互通信，因此对于企业内部以及跨企业的应用集成与合作，服务提供了一个分布式计算基础架构。服务客户端既可以是企业内部其他的一些解决方案或应用程序，也可以是企业外部的客户端。这些服务客户端既可以是外部的应用程序，也可以是一些流程或用户。服务提供者与服务客户端之间的区别独立于服务提供者与服务客户端间的关系。服务提供者与服务客户端间的关系既可以是客户/服务器关系也可以是 P2P 关系。在面向服务的计算模式中，服务需要技术中立、松耦合以及支持位置透明性。

服务若要技术中立，则必须使用要求最低的标准化技术调用服务，所采用的调用技术需要得到绝大多数信息技术环境的支持。这意味着调用机制(协议、描述以及发现机制)需要广泛地遵循一些公认标准。服务若要实现松耦合，则应该做到无须了解客户端和服务端的信息，也无须了解客户端和服务端的内部结构或内部协议(背景)。最后，为了支持位置透明性，需要将服务的定义以及位置信息存储在公开的信息库中，诸如统一描述发现和集成信息库(参见第 6 章)，并且这些公开的信息库能够被各种客户端访问。从而，客户端无须考虑服务的具体位置，即可定位以及调用这些服务。

服务的主要优点之一是，它们既可以在一台机器上实现，也可以在多个各不相同的设备上实现。服务的实现可以分布在一个局域网中，甚至也可以跨几个广域网(包括移动网络和自组织网络)。

当服务使用因特网作为通信手段以及使用基于因特网的标准时，即为 Web Service。Web Service 既具有普通服务的一些共性，也有它自身的一些特点，因为这些分布式服务之间的相互交互使用了公开的、不安全的、低保真度的机制，诸如因特网。

1.1.1 Web Service 是什么

Web Service 是一个可通过网络(如因特网)使用的自描述、自包含软件模块，这些软件模块可完成任务、解决问题或代表用户、应用程序处理事务。Web Service 建立了一个分布式计算的基础架构。这个基础架构由许多不同的、相互之间进行交互的应用模块组成。这些应用模块通过专用网络或公共网络(包括因特网和万维网)进行通信，并形成一个虚拟的逻辑系统。

Web Service 可以是：(i)自包含的业务任务，如提款或取款服务；(ii)成熟的业务流程，如办公用品的自动采购；(iii)应用程序，如人寿保险应用程序、需求预测与库存补充应用程序；(iv)已启用服务的资源，如访问特定的保存病人病历的后台数据库。Web Service 的功能千差万

别，既可以是进行简单的请求，如信用卡的核对与授权、价格查询、库存状态检查或者天气预报等，也可以是需要访问和综合多个数据源信息的完整的业务应用程序，如保险经纪人系统、保险责任计算、旅行自动规划或者包裹跟踪系统等。

对于预先定义的关系以及因特网上各个孤立的服务的严格实现，Web Service 解决了这些问题。Web Service 技术的远期目标是实现分布式应用，按照不断变化的业务需求动态组配应用程序，并可根据设备(如个人电脑、工作站、便携式计算机、WAP 手机、PDA)、网络(如有线电视线缆、移动通信系统、各种数字用户线路、蓝牙等)和用户访问的情况定制具体的分布式应用，保证所需之处都可广泛利用任何业务逻辑的具体片段。一旦部署了一个具体的 Web Service，其他的应用和 Web Service 就能发现和调用这个 Web Service。

在 1.3 节中，将讨论 Web Service 的更完整的定义，因为那时读者已经熟悉了“软件即是服务”这一理念，并已了解 Web Service 与基于 Web 的应用程序两者之间的差别。

1.1.2 Web Service 的典型场景

为了便于和其他的应用程序集成，Web Service 关注的焦点在于已有的应用程序(包括系统中的代码)的复用。跨业务范围或在业务伙伴间的新的服务共享形式通常都是 Web Service 关注的焦点。

为了更好地理解 Web Service 的功用，下面举一个保险公司的例子。在这个例子中，公司决定向它的客户提供一个在线报价的 Web Service。这家企业并不想从头开始开发一个完整的应用程序，而是试图在老系统中添加一些完成行业标准功能的模块。因此，假如有些企业擅长保险责任计算，则老系统将可以与那些企业的 Web Service 进行无缝衔接。这个保险责任 Web Service 可以基于客户想要的保险类型，向客户显示一个报价单，收集客户信息。随后，该 Web Service 可向客户显示一个包含预估的保险费在内的报价。假如客户最终选择购买该保险单，则系统将会收集客户的支付信息，并通过其他公司(服务提供者)所提供的支付 Web Service 进行支付处理。这个支付服务最终将给客户以及最初的公司返回一个账单信息。

如保险报价 Web Service 这样的企业应用，是最有可能能从 Web Service 技术的使用中获益的候选应用。企业应用覆盖了大量的 Web Service 场景，包括组织内部的部门之间以及业务合作伙伴之间的接口。企业通常使用一个单独的 Web Service 来完成一项具体的业务任务，如账单或库存控制，或者将几个 Web Service 组合起来创建一个分布式的企业应用，如个性化定制、客户支持、供货合同和日常管理支持等。这些企业应用场景需要复用和集成企业内已有的后端系统，它们是企业应用集成(Enterprise Application Integration，简称 EAI)的目标，具体参见 2.9 节。更复杂的企业应用是电子商务，以及那些业务伙伴基于因特网进行合作的跨企业间的交互(参见 2.9 节)，这种跨企业间的交互是大型公司采购、生产、销售以及配送产品的典型方式[Papazoglou, 2006]。

案例研究：订单管理流程

下面，我们将要讨论一个订单管理流程，这个流程可以很好地结合一些将要在本书后面章节介绍的原理与概念。这个案例研究基于一个简单的订单管理场景。在这个场景中，需要管理顾客向特定供应商提交的订购单。订单管理解决方案支持端到端的订单处理流程。这个处理流程可以借助复杂的相互交互的 Web Service 集合来表示，并且这些 Web Service 间需要许多同步与协调。这些 Web Service 可配置和订购个性化产品，并向顾客提供有关供货状况的精确的实时信息，以及提供一些交互的价格选项和实时状态查询，此外还可执行库存和仓库管理等。

在本书所分析的这个简单的订购单场景中，顾客或采购组织最初创建一个订购单，并将请求发送给供应商。供应商则提供一个订购单 Web Service，这个 Web Service 可以接受订购单，并基于一些判断准则，如货物的供货情况以及顾客信用情况等，接受或拒绝客户的请求。图 1.1 显示了在供应商端这些交互的 Web Service 是如何相互作用的。这些 Web Service 涉及订购单、信用检查、自动账单、库存更新以及发送不同的提供商所提供的物品，并将它们打包或可供交付的产品。一旦接受了来自客户的订购单，订购单流程可以同时着手几项任务：检查用户的信誉、确定订单的内容是否有库存，计算订单的最终价格、生成顾客的最终账单、选择运货者以及为订单制度生产和运输计划。当流程中的一些任务可并行处理时，在这些任务中就存在同步策略。例如，在接受订单之前，需要首先确定顾客的信誉，运送费用也需要加到最终的价格中去，此外为了全面安排调度，必须安排运送日期。当所有该些任务都成功完成后，就可开始发货流程，并将发货单发送给顾客。

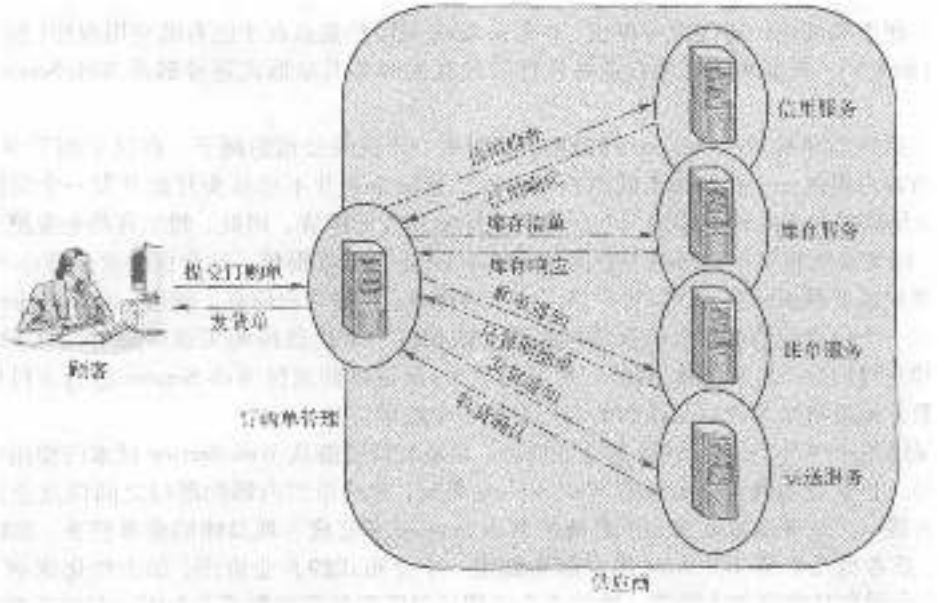


图 1.1 涉及多个相互交互的 Web Service 的订购单应用程序

像我们之前描述的订购单 Web Service，可以处理更复杂的任务。例如，订购单 Web Service 可以提供跟踪与调整功能，当碰到一些意外的事件时，如顾客要求修改或取消订货单，可跟踪与调整订购单。这些任务涉及许多协调工作，并且需要使用反冲的 Web Service。假如订购单中的单个事件需要修改或取消，则订购单的整个流程需要立即解决。针对这种情况，可使用相互协作的 Web Service 集合来调整订购单，这是针对这种问题的一个自动化解决方案。在取消订购单的情况下，订购单 Web Service 能自动预订一个合适的替换产品，并通知账单服务和库存服务进行相应的修改。当这些 Web Service 的所有交互任务都完成后，衍生成了新的调整后的安排，订购单 Web Service 将会通知顾客，向顾客发送一个更改后的发货单。

这个订单管理样例有助于安排下面章节的内容。我们将会继续讨论这个例子，并进行合适的扩充，从而阐述本书后面将要介绍的 Web Service 的一些关键内容。

1.2 “软件即为服务”的理念

虽然 Web 网页也提供了跨因特网和组织界限访问应用的方式，但 Web Service 与 Web 网页并不一样。Web 页面直接面向的是人，而 Web Service 的开发目标是访问者既可以是人也可以是自动化的应用程序。由于人们使用术语通常不很严格，一些人可能会将其实不是 Web Service 的“服务”与真正的 Web Service 混淆。因此，首先分析一下“软件即为服务”的理念是非常有价值的，这个理念也是 Web Service 技术的基础，然后将比较 Web Service 与基于 Web 的功能。

“软件即为服务”这一理念非常新颖，它首先产生于应用服务提供商软件模型中。应用服务提供商(Application Service Provider，ASP)是将软件、基础设施要素、业务以及专业的服务进行打包的公司，它们创建完整的解决方案，并将其作为基于订阅的服务向用户推介。ASP 是第三方(服务组织者)，它们部署、维护并管理打包的应用程序，并通过提供集中管理方式，对跨网络的客户提供应用程序可用性、安全性。以订阅或租赁的方式在网络上提供应用软件，最终用户可以使用因特网或专线远程访问这些应用软件。本质上，ASP 是一些公司将他们把对信息技术方面的部分甚至整个需求外包出去的一种方式。

ASP 的基本出发点是向用户出租应用程序。整个应用程序的开发包括用户界面、工作流、业务和数据组件，将这些组建捆绑在一起，并作为一个有效的解决方案提交给用户。ASP 管理整个应用系统，用户除了生成一些表格、确定用户界面的外观(诸如增加公司的标识)等，基本无法定制应用软件。通过可供下载的报表，用户仅需浏览以及进行人工订购、交易，即可访问软件。对于企业来说，这意味着 ASP 在维护应用系统、相关的基础设施及客户数据，并保障在任何需要的时候都可使用系统和数据。

ASP 也可采用另一种方式提供软件模块，即用户可以根据自身的需要将软件模块下载到自己的站点。当软件无法以客户/服务器模式工作时，或者软件无法通过浏览器远程工作时，可采用这一方式。在一段时间后，可删除这些软件模块，也可一直将这些软件模块保留在客户的计算机中，直到有新版本可以取代它们。也可能由于合约到期，用户不能再保留这些软件。

虽然 ASP 模型首先引入了“软件即为服务”的理念，但是它也受到一些固有的束缚，如无法开发高度交互的应用；无法提供完全定制的应用；无法整合各种应用。这种紧耦合的方式将会导致一些必然后果，如体系结构单一、比较脆弱、仅能针对特定客户、没有可复用的应用集成。

目前，“软件即为服务”这一理念正被进一步发展。新的体系架构支持松耦合方式的异步交互，这些异步交互基于 XML 标准，从而更容易访问因特网上的应用程序，这些应用程序之间也更容易通信。

Web Service 规范进一步扩展了“软件即为服务”的理念，可将复杂的业务流程和事务也视为服务。

鉴于 Web Service 技术的一些优点，许多 ASP 修改了他们的技术基础架构和业务模型，变得更类似于 Web Service 提供者。Web Service 给 ASP 提供了更灵活的解决方案。业务和数据组件作为应用程序的核心，目前仍然驻留在 ASP 的机器上，但是可以通过 Web Servcie 接口可编程地访问。客户可以构建他们自身的特定的业务流程和用户接口，并且可以从网络上自由选择那些满足他们的业务需求的各类 Web Service。

若将 Web Service 与基于 Web 的应用程序进行比较，可以发现有四方面的显著差异 [Aldrich 2002]：

- 对于请求或调用 Web Service 的应用程序而言，无论这种调用是否需要人的干预，请求或调用的 Web Service 都可视作应用程序的资源。这意味着 Web Service 可以调用其他的 Web Service，从而将复杂事务中的一些处理交由其他的一些 Web Service 实现。这提供了基于 Web 的应用目前无法达到的高度的灵活性和适应性。
- Web Service 是模块化的、自感知和自描述的应用程序。Web Service 知道它能完成什么功能，也知道何种输入会产生何种输出，并将其向潜在用户或其他的 Web Service 进行描述。Web Service 也能描述它的非功能性属性，例如调用 Web Service 的花费、Web Service 覆盖的地理范围、使用 Web Service 所涉及的安全性度量、性能特点、联系信息等(参见 1.8 节)。
- 相比于基于 Web 的应用程序，Web Service 更容易被监控和管理。可在任何时候使用外部的应用管理和工作流系统来监控和管理 Web Service 的状态。尽管 Web Service 可能不在内部(本地)系统上运行，或者我们不熟悉编写 Web Service 所用的语言，但是本地应用程序依然可以使用这些 Web Service。本地应用程序可以检测到 Web Service 的状态(活动性和可用性)，并可管理 Web Service 的输出状态。
- 可对 Web Service 进行评估和拍卖。假如几个 Web Service 完成同样的任务，Web Service 可对所要使用的服务进行招标。代理可基于 Web Service 的“竞价”属性(花费、速度、安全性)进行选择。

1.3 Web Service 的完整定义

在前面的章节中，我们已经广泛讨论了 Web Service 的起源以及 Web Service 与基于 Web 的应用这两者之间的区别。本节中，我们将要深入描述 Web Service 的主要功能特征。

Web Service 形成了创建分布式应用的构件，可在因特网或企业的内部网中发布及访问这些构件。Web Service 需要依赖一整套的开放的因特网标准。创建分布式应用通常需要将各组织部门或不同企业中的一些已有的软件模块集成在一起，这将需要使用不同供应商的不同工具。基于开放的因特网标准，开发者可实现分布式的应用。例如，跟踪企业内部的零件库存情况的应用可以提供一个有用的服务，用于响应有关库存情况的查询。更重要的是，分布式应用可以根据实际需要对 Web Service 进行组合和/或配置，从而有效地完成(与业务相关的)各种任务和事务。

根据面向服务的体系结构的参考架构(参见 1.6 节)，使用通用词汇(业务术语)和已发布的 Web Service 的性能目录，Web Service 可以发现其他的 Web Service 并与它们通信，从而完成相关任务，或者直接将高层事务中的有些部分直接交由其他的 Web Service 处理。

到此，可以给出 Web Service 的完整定义。Web Service 是一个平台独立的、松耦合的、自包含的、基于可编程的 Web 的应用程序，可使用开放的 XML 标准描述、发布、发现、协调和配置这些应用程序，用于开发分布式的互操作的应用程序。Web Service 能够在一些常规的计算中提供一些服务，从而完成一个具体的任务，处理相关的业务或者解决一个复杂的问题。此外，Web Service 使用(基于 XML 的)标准化的因特网语言和标准化协议在因特网或内部网上展示它们的可编程功能部件，并通过自描述接口实现 Web Service。这些自描述接口基于开放的因特网标准。

下面，我们将更详细地分析上述定义，并解构它的含义。

Web Service 是松耦合的软件模块：Web Service 之所以不同于以前的分布式计算体系结构，关键的一点是，Web Service 协议、接口和注册服务可以使用松耦合的方式协同工作。为了做

到这一点，服务接口的定义必须中立，独立于任何底层平台、操作系统及实现服务所使用的编程语言。因此，服务将可在一些不同的系统上实现，并以一致的形式和通用的方式相互交互。中立的接口定义将不会受到特定实现的很大影响，从而在服务间做到松耦合。松耦合这一概念对于理解 Web Service 的工作原理非常重要，在 1.4.4 节，我们将会重新讨论 Web Service 的这一特性。

Web Service 语义封装各个独立的功能：Web Service 是一个完成单个任务的自包含的软件模块。该模块描述了自身的接口特征，例如操作可用性、参数、数据类型和访问协议。基于这些信息，其他的软件模块将能确定该模块能完成什么功能，确定如何调用这些功能及确定可能的返回结果。在这一点上，Web Service 是契约化的软件模块，模块公开提供了接口特征的可用描述。Web Service 的潜在客户将能绑定到这些接口，并通过这些接口访问 Web Service。对于竞争相同资源的管理应用，Web Service 提供了单一实体，并且 Web Service 允许每个工作负载都作为单个的工作单元进行管理。

编程式访问 Web Service：Web Service 提供了编程式访问，可将 Web Service 嵌入到远程的应用中，因此可以查询和更新信息，从而提高了效率、响应性和精确性。这将给 Web Service 带来最大的附加值。与 Web 网站不同，Web Service 并不是主要面向人的，而是在代码级上进行操作。其他的软件模块和应用程序可以调用 Web Service，并与 Web Service 交换数据。当然，Web Service 也可集成进那些进行人机交互的软件应用程序中。

可动态发现 Web Service 并将其添加到应用中：与目前已有的接口机制不一样，可对多个 Web Service 进行装配，从而实现某个特定的功能、解决一个具体的问题或者向客户提供一个特定的解决方案。

可使用标准的描述语言来描述 Web Service：Web Service 描述语言(WSDL)既能描述功能性服务特性也能描述非功能性服务特性。功能性特性包含定义 Web Service 整体表现的操作特性。非功能性特性主要描述所在环境的特性(参见 1.4.2 节和 1.8 节)。

可在整个因特网上分发 Web Service：Web Service 使用一些非常通用的因特网协议，如 HTTP 等。与 Web 内容一样，Web Service 也依赖于同样的传输机制。Web Service 利用已有的基础架构，并遵循企业当前的防火墙策略。

1.4 Web Service 的特性

这一节将要介绍 Web Service 的最重要的一些特性，如简单和复合的 Web Service、有状态和无状态 Web Service、Web Service 的粒度、松耦合、同步和异步服务等。

1.4.1 Web Service 的类型

按照拓扑结构，Web Service 可以分成两类，如图 1.2 所示。第一种类型是信息型，Web Service 仅支持简单的请求/响应操作。Web Service 一般在等待请求，然后处理并响应请求。第二种类型是复合型，Web Service 在进入操作(Inbound Operation)和离开操作(Outbound Operation)之间进行一定形式的协调。这两类模型各自都有一些重要的特性，并都可进一步划分成一些更细的类别。

1. 简单服务或信息型服务

信息型服务比较简单，它们可对一些内容进行访问，最终用户通过请求/响应序列与这些内容进行交互。信息型服务也可将后端业务应用程序暴露给其他的应用程序。若 Web Service 暴露应用程序(或者构成这些应用程序的组件)的业务功能，则这类 Web Service 通常称为编程式服务。例如，它们可以暴露功能调用，这些功能调用通常是使用如 Java/EJB、Visual Basic 或 C ++ 等

编程语言编写的。这类被暴露的编程式简单服务完成请求/响应类型的业务任务，并可被视为“原子”(或单元)操作。使用 Web Service 描述语言 WSDL(参见第 5 章)可定义编程式接口。通过这些标准的编程式接口执行 Web Service，应用程序将可访问功能调用。

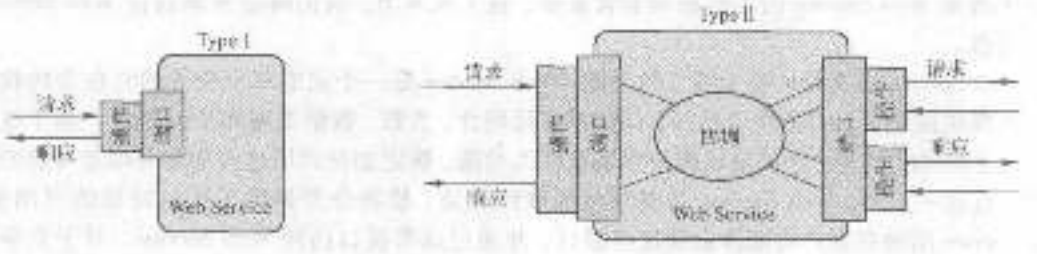


图 1.2 信息型和复合型服务的概要视图

按照所解决的业务类别的不同，信息型服务可以进一步细分为三类：

(1) 纯内容服务：纯内容服务编程式访问内容，诸如访问天气预报信息、简单的财经信息、股票价格信息、规划信息、时事新闻等。

(2) 简单的交易服务：简单的交易服务是一种比较复杂的信息服务。简单的交易服务可以跨不同的系统和信息源(包括不同的后端系统)，对于业务系统进行编程式访问，以便请求者可以做出合理的决策。这类服务请求的实现可能比较复杂。例如，对于一些复杂的实体组织，如仓储化的物流服务等纯的业务服务，实际上是一些前端服务。

(3) 信息联合服务：信息联合服务是一些增值信息 Web Service，旨在嵌入到不同类型的商业网站，诸如网上交易市场、销售网站等。这些服务通常由第三方提供，包括各类商务服务。诸如物流、支付、租约、跟踪服务，以及其他的一些增值服务，诸如缴费服务等。联合服务的典型例子包括旅行网站的预订服务或者保险网站的保单服务。

信息型服务仅完成一个独立的工作单元，并且底层的数据存储将处于一致的状态。然而，信息型服务本质上并不属于事务性服务(虽然信息型服务的后端实现也许是事务型服务)。信息型服务并不会保留不同请求之间的信息。从这个角度看，信息型服务也称为无状态的 Web Service。

信息型服务和简单的交易服务需要得到三个不断发展的标准的支持：(i)通信协议(简单对象访问协议)；(ii)服务描述(Web Service 描述语言，简称 WSDL)；(iii)服务发布和发现(统一描述、发现和集成基础架构)。这些将在本书的第 4 章、第 5 章、第 6 章分别进行讨论。

2. 复合服务或业务流程

企业可以使用单个的(不相关联的)服务来完成一个具体的业务任务，诸如账单处理或库存控制。然而，对于企业来说，若要充分发挥 Web Service 的作用，业务流程 Web Service 和事务类 Web Service 也是非常重要的，其价值甚至要超过信息型 Web Service。当企业需要将几个服务组合在一起创建一个业务流程，诸如定制订单、客户支持、采购和物流支持等，企业则需要使用复合的 Web Service。复合的(或混合的)服务通常涉及许多已有服务的装配和调用，从而完成多步骤的业务交互。这些已有的服务可以来自于多个不同的企业。例如，供应链应用将涉及订单受理、入库订单、采购、库存控制、财务和物流。在流程中，将会出现大量的文档交换，包括查询价格、返回价格查询的结果、下订单、订单确认、送货信息等。长事务和异步消息也会发生，并且在达成最终协议以前，也可能会发生业务商谈甚至业务谈判。这种功能通常是业务流程(或者复合服务)的特性。

复合服务可以按照它们组成简单服务的方式依次进行分类。一些复合 Web Service 构成简单的 Web Service，这些简单的 Web Service 具有编程式行为，而复合服务主要具有交互式行为。在交互式行为中，具体输入由用户提供。因此，可以很自然地区分下列两类复合的 Web Service：

（1）构成编程式 Web Service 的复合服务：这些 Web Service 的客户可将它们装配为复合服务。具有编程式行为的简单服务的典型例子是库存检查服务，该服务是构成库存管理流程的一部分。

（2）构成交互式 Web Service 的复合服务：这些服务暴露了 Web 应用的表示（浏览器）层的功能。它们通常暴露多步骤应用的行为，Web 服务器、应用服务器和底层的数据库系统相互协作，并将应用直接提交给浏览器，并最终与人进行交互。这些 Web Service 的客户可以将交互的业务流程合并到他们的 Web 应用中，将外部的 Web Service 集成到应用中。显然，编程式服务可与交互式服务相互集成，从而实现通常既包含业务逻辑的功能又具有浏览器交互性的业务流程。

复合服务的功能是粗粒度的，并且复合服务是有状态的。有状态的 Web Service 保持在不同的操作调用之间的一些状态，并且这些不同的操作调用既可以由相同的 Web Service 客户发出，也可以由不同的 Web Service 客户发出（参见 1.4.3 节）。

复合 Web Service 的标准仍然还在不断修订，并集中在通信协议（简单对象访问协议）、WSDL、统一描述发现和集成基础架构、WS-MetaDataExchange（WS-MetaDataExchange 允许服务端点向请求者提供元数据信息，并支持 Web Service 交互的自启动）以及 Web Service 业务流程执行语言（简称 BPEL）。

1.4.2 功能属性和非功能属性

可使用描述语言对服务进行描述。服务描述有两个主要的相互关联的组件：功能特性和非功能特性。功能性描述详述了操作特性。操作特性定义了服务的整个行为，例如定义了如何调用服务、在何处调用服务等细节。功能性描述主要关于消息的语法规则，以及如何配置发送消息的网络协议。非功能性描述则主要关于服务质量属性，诸如服务计量和代价、性能度量，例如响应时间或精度、安全性属性、授权、认证、（事务的）完整性、可靠性、可伸缩性和可用性。非功能性描述主要关于服务请求者的运行环境，诸如包括指定非功能性需求的 SOAP 头，而非功能性需求则可能影响服务请求者可能会选择哪一个服务提供者。安全策略声明（有关服务安全性策略的细节参见第 12 章）可能就是这样的例子。

在第 5 章中将讨论服务的功能属性，而非功能属性则在本章的 1.8 节中进行讨论。

1.4.3 状态属性

Web Service 既可以是无状态的，也可以是有状态的。假如服务可以被重复调用，且无须维持上下文或状态，这样服务则称为无状态的服务。反之，需要维持上下文或状态的服务则称为有状态的服务。服务访问协议通常是无连接的。无连接的协议没有作业或会话的概念，并对最终的提交不进行任何假定。

最简单形式的 Web Service，如信息型天气预报服务，并不会“记忆”请求之间所发生的状态。这类服务称为无状态 Web Service。无状态意味着每当用户和 Web Service 交互一次，就会完成一个处理。在返回服务调用的结果之后，处理就完成了。后面紧接着的调用将与前面的没有任何关系。结果，既可以通过请求消息来传递完成服务所需的所有信息，也可以基于请求所提供的信息，从数据信息库中获得完成服务所需的所有信息。

与无状态的 Web Service 相比，有状态的 Web Service 维持不同操作调用之间的状态，无论这些操作调用是由 Web Service 的同一个客户端发出还是由不同的客户端发出。假如一个特定

的“会话”或“对话”涉及组合的 Web Service，则操作调用间的瞬态数据是有状态的。发送到 Web Service 有状态实例的消息的解释将与那个具体的实例的状态相关。通常情况下，业务流程规定了涉及合作伙伴间的消息交换的有状态交互。其中，业务流程的状态包括了在业务逻辑中以及发送合作伙伴的消息中交换信息以及中间数据。例如，在订单管理应用中，卖方的业务流程可能会提供一个服务，这个服务可能首先通过输入信息接收订购单，假如能够履行订单，则向顾客返回一个确认。应用程序然后将会继续向顾客发送消息，诸如发货通知和开发票等。当顾客与卖方之间同步执行许多采购流程时，卖方的业务流程必须“记得”每个订购单各自的交互状态。

1.4.4 松耦合

Web Service 彼此进行动态交互，并且 Web Service 使用的是因特网标准技术，这使得系统间的融合成为可能。否则的话，需要大量的开发工作才可能融合这些系统。“耦合”这一术语表示了两个系统之间彼此相互依赖的程度。

在紧耦合的交换中，应用程序需要知道它们的合作伙伴的应用程序是如何运行的。它们也需要知道有关合作伙伴如何进行通信的详细细节，如所暴露的方法的数量以及每个方法所接受的参数的细节，以及所返回的结果类型等。此外，紧耦合应用程序需要知道和它们协作的应用程序的具体位置(并且这意味着一定程度的“安全性”保障)。传统的应用程序设计依赖于所有组成部分的紧耦合，并且这些组成部分通常在同一流程中运行。因此在紧耦合环境中，核心设计模式是同步交互。按照紧耦合的要求，应用程序的不同组件之间的接口在功能和形式上是紧密相关的。当应用程序和服务的数量增多时，需要及时创建和维护的接口数量将变得难以控制。因此，构建紧耦合的应用程序通常十分困难并且非常繁琐，从而需要花费大量的时间来定义两个协作应用间的连接和关系。紧耦合需要在通信系统间达成协定、共享上下文，并且对任何变化非常敏感。相比于紧耦合方法，松耦合系统的连接和交互比较自由(可以跨因特网)。在松耦合系统中，当发生变化时，应用程序不需要知道和它们协作的应用程序是如何运作的，也不需要了解协作的应用程序是如何实现的。松耦合系统的好处就在于它的灵活性。当构成应用程序的各个服务的内部结构和实现不断发生变化时，松耦合系统可以做到随需应变。松耦合系统间的交互通常基于异步或事件驱动模型，而不是同步模型。若要构建高集成、跨平台、程序间相互通信的应用，松耦合的应用程序具有较好的灵活性和互操作性，而使用传统的方法则很难做到这些。

若使用 Web Service 方式，则从服务请求者到服务提供者之间的绑定是松耦合的。这意味着服务请求者无须了解服务提供者实现的具体技术细节，诸如编程语言、部署平台等。服务请求者通常使用消息来调用服务，即服务请求者通过消息进行请求，服务提供者也通过消息进行响应，而不是使用应用编程接口或文件格式。

表 1.1 概括了松耦合和紧耦合间的差异。

表 1.1 紧耦合和松耦合之间的比较

	紧 耦 合	松 耦 合
交互模式	同步	异步
消息类型	RPC 类型	文档类型
消息路径	硬编码	路由化
底层平台	同构	异构
绑定协议	静态	动态 - 延迟绑定
目的	复用	灵活性、广泛的适用性

1.4.5 服务粒度

Web Service 的功能可以千差万别，既可以是简单的请求，也可以是一个存取和综合多个信息源的信息的复杂系统。简单的服务请求甚至也有复杂的实现。本质上，简单请求之间是不相关的，通常基于请求/回复模式进行运作。简单请求通常是细粒度的，例如它们通常不可再分。反之，复合服务通常是粗粒度的，例如 SubmitPurchaseOrder(提交订购单)流程通常涉及在一个或多个会话中和其他服务或最终用户进行交互。粗粒度的通信意味着更大型、更丰富的数据结构(例如 XML 模式所支持的数据结构)，并且使松耦合成为可能。反过来，松耦合又使得异步通信成为可能，从而使得完成整个任务所需的信息交换最小化。

1.4.6 同步

我们可以对比服务的两类编程方式：一类是同步或远程过程调用(RPC)方式；另一类是异步或消息(文档)方式，可参见 2.4 节和 2.5 节。

同步服务：同步服务的客户端将它们的请求表示为带变量的方法调用，方法返回一个包含返回值的响应。这意味着，当客户端发送一个请求消息时，它会首先等待响应消息，然后才会继续向下运行。这就使得整个调用不是完全成功就是完全失败。假如某个操作由于某种原因不能成功，则其他所有的依赖操作也都将失败。由于在客户端和服务之间的双向通信，RPC 类型的服务在客户端和服务提供者之间需要紧耦合的通信模式。当应用程序具有下列特性时，通常将用到 RPC 类型的 Web Service：

- 调用服务的客户端需要一个立即的响应。
- 客户端与服务以反复对话的方式进行协作。

简单的 RPC 类型的同步服务的典型例子包括：返回特定股票的当前价格；提供特定地区的当前天气情况；在完成业务交易之前，核实潜在贸易伙伴的信用情况。

异步服务：异步服务是文档类型的服务或消息驱动类型的服务。当客户端调用消息类型的服务时，客户端通常发送整个文档，诸如订购单，而不是单独发送一些参数。服务收到整个文档后，会处理它，然后返回(也可能不返回)一个结果消息。调用异步服务的客户端在继续运行应用程序的其他部分之前，并不需要等待响应，从服务发回的响应可以在数小时甚至数天后才出现。

在松耦合环境中，异步交互(消息)是一个核心设计模式。消息使得应用所处的松耦合环境既不需要了解如何进行通信的技术细节，也不需要了解其他应用程序的接口。这使得任何两个流程之间的通信操作可以是自包含的、独立的工作单元。当应用程序具有下列特性时，通常需要使用文档类型的 Web Service：

- 客户端不需要(不期待)立即的响应。
- 服务是面向文档的(客户端通常发送一个完整的文档，例如订购单，而不是发送一些离散参数)。

文档类型的 Web Service 的例子包括订购单处理、响应客户的询价请求、响应一个特定顾客的订购等。在所有这些情况中，客户端向 Web Service 发送一个完整的文档，诸如订购单，然后假定 Web Service 以某种方式正在处理，但是客户并不需要一个立即的回复。

1.4.7 良定义

服务间的交互必须是良定义的。应用程序使用 WSDL 可以向其他的应用程序描述连接和交互的规则。对于抽象服务接口及支持服务的具体的协议绑定，WSDL 提供了对它们进行描述的统一机制。将服务请求者绑定到服务提供者需要描述一些细节信息，WSDL 正是一种受到广泛支持

的描述方法。服务描述主要是关于操作如何与服务进行交互、消息如何调用操作、构建这类消息的详细信息，以及在哪里发送消息等(例如确定服务的接入点)。

WSDL 并不包括 Web Service 实现的任何技术细节。服务请求者既不知道也不关心服务是否用某种编程语言来实现，诸如 Java、C#、C 等。并不需要关心服务是在何种平台上开发和实现的，只要 Web Service 能处理 SOAP 消息即可。

在 1.5 节中，将继续讨论这些问题，并将区别服务定义中的服务接口部分和服务实现部分。

1.4.8 服务的使用环境

除了上面提到的 Web Service 的类别和特性，从 Web Service 请求者的角度，将信息服务划分为不同的类别也是有价值的。在这里，我们可细分为可代替的服务与关键任务服务这两类。

可代替的 Web Service 是多个提供者都可提供的服务。只要服务接口是相同的，将可代替的 Web Service 的服务提供者由一个换成另一个并不会影响应用程序的功能。假如服务因为某些原因暂时不可用，但可以选择其他的服务提供者作为后备，则系统的实际应用就不会受到很大的影响。关于涉及具有可替代的选择的发现过程，这里可以举个例子。我们可以搜寻不同的租车服务机构，如 Avis、Hertz、Budget，并可选择第一个抵达并满足需求的响应。这类服务通常可以很好地集成到客户流程中(例如租车活动)，并且不需要交换一些重要的业务数据。

关键任务 Web Service 是很可能只被一个特定的服务者提供的服务。假如关键任务 Web Service 被替换，则可能会严重影响整个应用程序的功能。假如该服务因为某些原因暂时不可用，则可能给应用系统造成极大的影响。这类服务通常处理一些关键的业务数据，并通常在流程级进行集成。

1.5 服务接口和实现

服务的一个重要方面是对接口和实现具有明显的区分[Alonso 2004]。

服务接口部分定义了外部世界可以看到的服务功能，并提供了访问这些功能的方式。服务描述了它自身的接口特性，操作的可用性、参数、数据类型及访问协议。从而其他的软件模块可以确定该服务能做什么，如何来调用该服务所提供的功能，以及可能的返回结果。在这点上，服务是契约化的软件模块，因为服务公开了用于访问服务的接口特性，从而潜在客户将可绑定到该服务。服务客户端使用服务接口描述绑定到服务提供者，并调用服务所提供的功能。

服务实现部分实现了具体的服务接口。对于服务的用户来说，服务的实现细节是隐藏的。不同的服务提供者可以选择任何编程语言来实现同一个接口。服务的实现既可以直接提供服务功能，也可以通过组合其他服务来提供相同的功能。

在许多情况下，因为提供服务接口的组织与实现服务接口的组织并不相同，所以对服务接口和服务实现的区分非常重要。服务是一个由应用程序或用户确定的业务概念。服务的实现(例如具体的服务内容)可以由一些软件包提供，例如企业资源计划(ERP)软件包、专用构建组件、商用软件，或者包含大量商业逻辑的原有的遗留应用。

若要更好地理解如何设计和开发服务，首先需要理解服务、接口和组件之间的关系。当设计应用程序时，开发者首先创建逻辑模型和服务。在企业中，业务对象(诸如产品、客户、订单、账单等)将遵循逻辑模型，而服务则体现了业务对象的需求，例如库存控制、送货调度等。对于开发者来说，服务实现可包含服务接口规范以及具体组件(业务对象)的实现。组件技术通常用于实现服务的功能。组件是系统中的一个独立封装的模块，具有明确定义的功能和作用范围。概括地说，服务接口和相应的实现组件是具有很大互补性的两部分。实现组件是对 Web Service 的具体实现。

服务之间进行交互的唯一方式就是通过它们的接口。服务通常是软件所实现的业务功能。服务被封装了一个正式的文档化接口。无论是设计该服务的代理还是不了解如何设计该服务的代理，都可了解和定位服务接口，从而访问和使用它们。这类黑盒封装继承了软件工程中的模块化理论的特色。实际上，服务不同于描述整个业务功能的各类模块。服务是可复用的，并能以各种方式组合到一些新的服务中。这种组合既可以发生在单个程序的级别甚至应用级，也可以跨越个企业甚至多个企业。

无论服务是由若干小的组件组合而成所提供，还是由 ERP 这样的单个系统所提供，服务客户端都无须关心这一点。然而对于实现服务的开发者来说，考虑粒度问题依然是非常重要的。为了优化性能，有时需要修改服务的具体实现，粒度问题涉及是否能够最大程度地减少对其他组件、应用及服务的影响(可参见第 15 章，在该章中讨论了有关服务的设计和开发方面的内容)。

为了将服务组合进业务流程中，除了接口这一概念，我们还需引入一个新的概念——服务端点接口。服务编配接口必须明确地描述组合服务客户端所期望的全部接口，以及那些组合到服务中的出环境所提供的接口。在使用导入的服务接口(很可能是单个的)定义组合服务接口时，将引到服务编配接口。从这个意义上，服务编配接口的作用和组合元模型一样。组合元模型描述了 Web Service 接口间如何相互交互，并描述了如何基于 Web Service 已有的接口(imported <PortType>)定义新的接口(或 <PortType>，参见 5.2.1 节)。

服务编配接口的思想如图 1.3 所示。在图 1.3 中，定义了服务的封装边界。若要使用导入服务并且无须了解服务的具体实现，这是可靠地设计服务的唯一方式。由于服务的具体开发将需要处理多个导入的服务接口，因此引入"服务使用接口"这一概念也是很有价值的。服务使用接口简化了向客户所需暴露的服务接口，隐藏了内部的服务编排接口(假如有的话)的具体细节。服务使用接口是客户端应用程序唯一可以查看的接口。

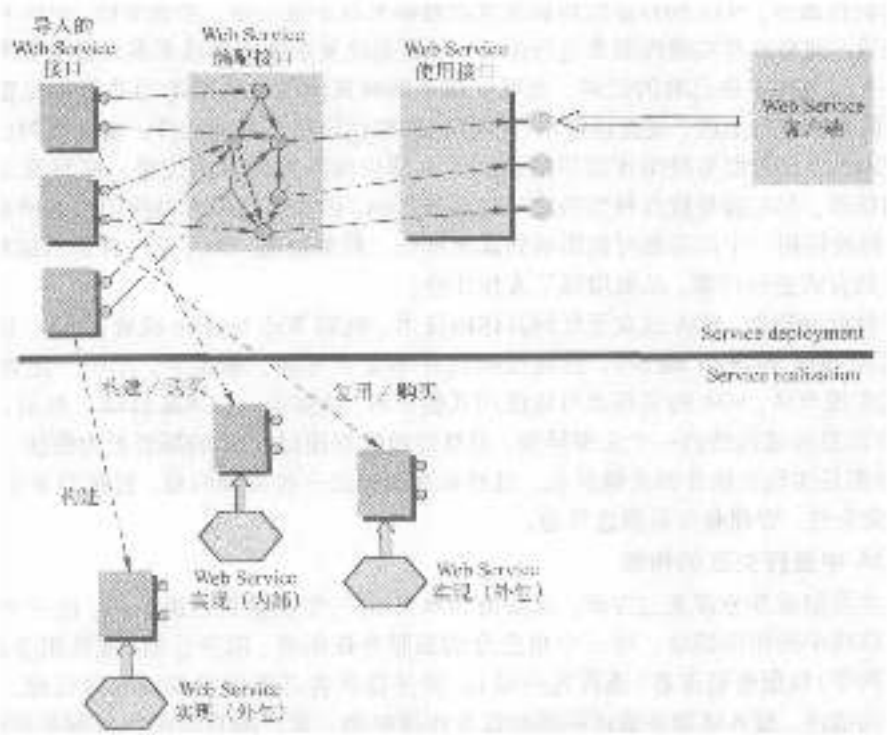


图 1.3 服务、接口和服务的实现

图 1.3 区分了服务的两个重要方面：服务的部署和服务的实现。服务的部署已经讨论过了。服务的实现策略涉及服务的许许多多的不同选择。例如，服务可能有不同的组合方式，包括由企业内部设计和实现服务、购买/租赁/支付服务、外包服务的设计与实现，以及使用包装器(wrapper)和/或适配器(参见 2.7.1 节)对遗留系统的功能进行转换，封装遗留系统的内部组件，并将其与最新的应用进行融合。在第 15 章中，我们将讨论服务实现的具体选择，并介绍有关服务的设计和开发方面的内容。

1.6 面向服务的体系结构

目前，应用程序集成的最主要的方法是通过交换，而 Web Service 却能超越这种简单的方式，可以访问、编程和集成应用服务。应用服务封装了已有的应用和新建的应用。这意味着，不仅可以在应用程序之间交换信息，而且可以利用本地和远程应用中的大量的后端系统和遗留系统，从而复合地、可定制地组合应用。

这个概念的关键是面向服务的体系结构(SOA)。SOA 是一种设计软件的逻辑方法，可通过发布或发现的接口向终端用户应用或网络上的其他服务提供服务。为了实现这一点，SOA 将企业中先前相互独立的软件应用和支撑基础架构重新进行组织，变为互联的服务集合。通过接口和消息协议，可发现和访问服务集合中的每个服务。一旦 SOA 的所有要素都准备就绪后，已有的或将要开发的应用程序就可根据需要访问基于 SOA 的服务。当一些应用程序使用不同的技术且在不同的平台上运行时，若它们之间需要相互通信，则采用 SOA 方式尤其合适。

SOA 的主要目的就是使得已有的技术间具有通用的互操作性，并使得未来的应用和体系结构具有可扩展性。使用 SOA 可将孤立的静态系统转化为模块化的、灵活的组件，组件代表了可以通过业界的标准协议进行访问的服务，因此 SOA 降低了互操作的难度。SOA 能够利用基于一些标准的功能性服务，SOA 的许多优势以及灵活性都来自于这一点。当需要时，SOA 可调用单个的功能性服务，或将这些功能性服务进行组合，从而创建复杂的应用或者多阶段的业务流程。这类构件服务可以复用一些已有的组件，也可以在不影响其他的独立服务的功能和完整性的情况下，对已有的组件进行更改，或直接使用一些新组件取代这些已有的组件。由于可对已有的组件进行更改或取代，这种服务模型比以往的孤立的大型应用具有更多的优势。而对孤立的大型应用进行任何修改，都可能导致各种预料之外的意外影响，因为孤立的大型应用中的所有代码都捆绑在一起，修改任何一个局部都可能影响到其他部分。简单地说，SOA 是一种体系结构类型，使用面向服务的方式进行计算，从而增强了互操作性。

作为一种设计理念，SOA 独立于任何具体的技术，例如 Web Service 或者 J2EE。虽然人们经常会同时讨论 SOA 和 Web Service，然而这两者并不完全等同。事实上，并不一定要使用 Web Service 才能实现 SOA，SOA 的实现也可以使用其他技术，诸如 Java、C#或 J2EE。然而，Web Service 可以视作消息传递模型的一个主要样例，消息传递模型使得 SOA 的部署更为便捷。Web Service 的一些标准是实现互操作的关键所在。这些标准也涉及一些关键问题，包括服务质量(QoS)、系统语义、安全性、管理和可靠消息传递。

1.6.1 SOA 中进行交互的角色

SOA 的主要组成部分涉及三方面，这是由 SOA 中的三个主要角色决定的，而三个主要角色对应于体系结构中的相应模块。这三个角色分别是服务提供者、服务注册(或称服务注册中心、服务注册机构等)和服务请求者(或称客户端)。服务提供者是提供服务的软件代理。提供者负责发布服务的描述，服务将服务描述提供给服务注册机构。客户端是请求执行服务的软件代理。代理既是服务客户端，同时又是服务提供者。客户端必须能够发现所需的服务描述，并能与相应

的服务进行绑定。为了实现这一功能，SOA 建立在目前 Web Service 所采用的一些基本规范之上，诸如 SOAP、WSDL、UDDI 和 BPEL for Web Service。在本书第 4 章、第 5 章、第 6 章和第 9 章中，将会分析这些基本规范。

1. Web Service 提供者

Web Service 体系结构中，第一个角色是 Web Service 提供者。从业务角度看，Web Service 提供者是拥有 Web Service 的组织，并实现了通过服务体现出来的业务逻辑。从体系结构角度看，Web Service 提供者是一个平台，驻留和控制对服务的访问。

Web Service 提供者负责发布 Web Service。服务注册机构驻留于服务发现机构，可提供 Web Service 的发布信息。这涉及对业务、服务及 Web Service 技术信息的描述，以及按照发现机构所规定的格式将信息注册到 Web Service 注册机构。

2. Web Service 请求者

Web Service 体系结构中，另一个主要角色是 Web Service 请求者（客户端）。从业务角度看，它是需要满足一定功能的企业。从体系结构角度看，它是搜寻并调用服务的应用。

为了找到所需的 Web Service，Web Service 请求者将搜索服务注册机构。这实际上意味着，在发现机构提供的注册机构中发现 Web Service 的描述。这也意味着可使用描述信息将客户端绑定到服务中。Web Service 请求者可分两类，请求者既可以是由最终用户驱动的浏览器，也可以是另一个 Web Service，那个 Web Service 将作为没有用户接口的应用程序的一部分。

3. Web Service 注册机构

在 Web Service 体系结构中，最后一个重要角色是 Web Service 注册机构。Web Service 注册机构是一个可供搜索的目录，可在该目录中发布和搜索服务描述。服务请求者可在注册机构中发布发现服务描述，并能获取服务的绑定信息。服务请求者使用这些绑定信息即可联系服务提供者或绑定到服务提供者，从而利用所提供的服务。

在前面的章节中，描述了 Web Service 体系结构中的三个操作——Web Service 的发布、搜寻和调用。Web Service 发现机构负责提供这三个操作所需的基础架构。Web Service 提供者发布 Web Service；Web Service 请求者搜寻并调用 Web Service。

1.6.2 SOA 中的操作

在 SOA 中，当应用程序利用 Web Service 在三个角色之间进行交互时，必然涉及三个主要操作。这三个操作分别是：发布服务描述，发现服务描述，以及基于服务描述绑定或调用服务。这三个基本操作既可能只进行一次，也可能会重复进行。

SOA 的逻辑视图如图 1.4 所示。该图描述了 SOA 中角色和操作之间的关系。首先，Web Service 提供者将 Web Service 发布到发现代理机构。然后，Web Service 客户端使用发现代理机构的注册中心搜索所需的 Web Service。最终，基于从发现代理机构所获得的信息，Web Service 客户端调用（绑定到）Web Service 提供者所提供的 Web Service。

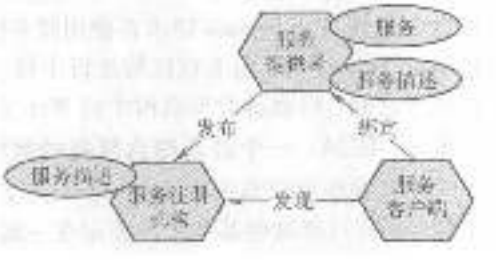


图 1.4 Web Service 的角色和操作

1. 发布操作

Web Service 只有在发布之后，其他用户或应用才能发现这个 Web Service。发布操作实际上由两个同样重要的操作组成。一个操作是对 Web Service 本身的描述，另一个操作是对 Web Service 的注册。

服务提供者首先需要使用 WSDL 正确地描述 Web Service，这是发布 Web Service 首先需要完成的工作。正确地描述 Web Service 需要以下三类基本信息：

- 业务信息：有关 Web Service 提供者或服务实现的信息。
- 服务信息：Web Service 的特征信息。
- 技术信息：有关 Web Service 的实现细节及调用方法的信息。

Web Service 发布后，紧接着就需要注册。注册涉及将服务的三类基本描述信息存储到 Web Service 注册机构中。为了让 Web Service 请求者能够发现 Web Service，需要将该 Web Service 描述信息至少发布到一个发现机构中。

2. 查找操作

与发布操作类似，Web Service 的发现也是由两个操作组成。若要发现所需的 Web Service，首先需要在发现机构的注册中心中搜索服务，然后从搜索结果中选择所需的 Web Service。

搜索 Web Service 时，将在发现机构的注册中心中查询满足 Web Service 请求者需求的 Web Service。查询包含一些搜索条件，诸如服务类型、首选的价格范围、与服务相关的产品、与服务相关的公司和产品类别，以及服务的其他一些技术特性等(参见第 6 章)。若执行查询，则将检索 Web Service 提供者在注册机构中输入的 Web Service 信息。根据请求者的不同，发现操作有两类。发现操作可在设计时静态指定，检索程序开发所需的服务的接口描述。发现操作也可动态指定，检索调用所需的服务的绑定和位置描述。

搜索操作将会返回一个 Web Service 结果集。作为发现操作的另一个组成部分，选择操作需要决定在结果集中调用哪一个具体的 Web Service。有两类选择方法：手动选择和自动选择。手动选择意味着：Web Service 请求者直接人工查看搜索操作返回的 Web Service 的结果集，并选择所需的 Web Service。而自动选择则从 Web Service 结果集中自动选择最符合要求的 Web Service。为了做到自动选择，Web Service 注册机构提供了一个专门的客户端应用程序。若要使用自动选择，Web Service 请求者需要指定一些选择偏好，以便应用程序能够推断出 Web Service 请求者最有可能希望调用何种 Web Service。

3. 绑定操作

在 Web Service 体系结构中，最后一个且可能也是最重要的一个操作是 Web Service 的实际调用。在绑定操作中，服务请求者使用绑定信息定位并联系服务，从而调用或者初始化一个运行时交互。在这里，将用到 Web Service 提供者注册到注册机构的技术信息。可有两类不同类型的调用。一类是 Web Service 请求者使用服务描述中的技术信息直接调用 Web Service。另一类是在调用 Web Service 时，由发现机构进行中转。在这一类调用中，在 Web Service 请求者和 Web Service 提供者之间，将通过发现机构中的 Web Service 注册机构进行所有的通信。

1.6.3 SOA：一个涉及综合服务的样例

正如在前面章节中所述，SOA 引入了构建分布式应用的一个新理念，可以发布和发现一些基本服务，并可将这些基本服务绑定在一起，从而创建更复杂的增值服务。下面以一个制造企业提交的订购单处理的业务流程为例。在这里我们假设，一个大型制造商构建了一个业务，可根据现货和合同提供一些特殊产品和个性化的塑料配件。在供应链中，如在精炼厂这样的商品供应商和消费性包装产品这样的制造商之间，需要公司管理多个业务伙伴的关系，甚至需要在供应商和客户之间充当中介商。

该例中将涉及一些 Web Service，包括订购单、信用核查、自动账单、库存更新，以及将由不同的服务提供者提供的商品进行打包，最后一并发送给用户。可根据这些 Web Service 间的交互开发订购单处理流程。为了将问题进行简化，我们假设订购单流程应用了一个相当简单的复合

服务。这个由供应商提供的服务由两个独立的服务组成，即库存服务和送货服务。这类聚合的SOA 表示如图1.5 所示。该图显示了一个使用关系，这个关系对于理解依赖性管理及整体状况非常关键。

为了表示这个复合 Web Service 的需求，图 1.5 涉及一个层次型服务提供模式，其中请求者(客户端)向聚合器发送一个请求，聚合器是一个向诸如订单管理(步骤 1)这类应用提供复合 Web Service 的系统。聚合器(零件供应商)是另一个服务提供者，它接受最初的请求，并将其分解为两部分，一部分涉及库存检查服务，另一部分涉及送货服务请求。聚合器充当了一个 Web Service 请求者，并将订单请求转发给库存(步骤 2)服务提供者。库存服务提供者确定所订购的零件是否有库存，并将响应发送给聚合器(步骤3)。假如以上这些步骤都成功完成，聚合器将选择一个送货商，该送货商将按照订单安排送货计划(步骤 4 和步骤5)。最终，聚合器反过来充当一个服务提供者的角色，并计算订单的最终价格，以及向客户递交账单，将最终的响应转发给客户端流程(步骤6)。

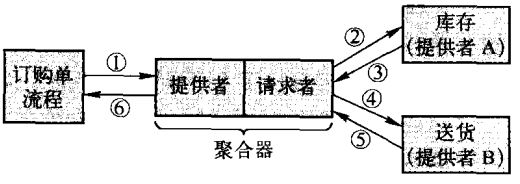


图 1.5 SOA：复合服务的实例

1.6.4 SOA 中的层次

SOA 是一个灵活的体系结构，它提供了一个集成框架。基于这个集成框架，软件架构师能够使用可复用的功能单元(服务)集和良定义的接口，并将它们融合成一个逻辑流，从而构建整个应用。应用将能在接口(约定)级进行集成，而不是在实现级进行集成。由于这种构建方式与接口具体实现方式无关，无须考虑特定系统或实现的特征或特性，因此将具有更大的灵活性。例如，可以基于策略，如价格、性能、QoS 保证、当前的事务量等，动态选择(同一个接口的)不同的服务提供者。

SOA 的另一个重要特性是可以进行多对多的集成。例如，跨企业的各类客户可以以各种不同的方式使用和复用应用程序。这种能力可极大地降低集成不兼容的应用的成本/复杂性比率，并能提高开发者针对新的业务需求快速创建、重新配置、创新运用应用程序的能力。其他的一些益处还包括可降低 IT 管理开销，更容易实施那些跨组织部门和贸易伙伴的业务流程的集成，以及可进一步增强业务的灵活性。

根据使用 SOA 的企业需求和业务重点的不同，有三类不同的 SOA 入口点：实现企业服务编配，提供给整个企业的服务，以及实现端到端协作型业务流程：

实现企业服务编配：这是基本的 SOA 入口点，是在部门内部或者在少量的部门和企业资产之间的一种典型的实现方式。实现企业服务编配包含两个步骤。首先，将企业资产和应用程序转换为 SOA 实现。一开始，可以使用 Web Service 技术创建服务来使用已有的一些单个应用，或者使用 Web Service 技术直接创建应用。这项工作首先是规定单个的应用或应用单元(包括遗留系统)的 Web Service 接口。在实现基本的 Web Service 之后的下一步，是将那些已经服务化的应用以及新创建的服务应用进行服务编配。这一步骤涉及将多个服务集成到一个完成特定业务任务的流程中。这一步骤支持许多集成类型，包括集成部门间的应用、部门间的数据、业务流程以及异构系统。

提供给整个企业的服务：在 SOA 入口点层次，下一阶段将是企业寻找一些基于 SOA 组件的通用服务。这些通用服务能够在整个组织内使用。实现企业级集成通常需要基于一些公认的标准，从而服务在跨部门时依然具有一致性，并且这也是将一个组织与它的合作伙伴及供应商进行集成的前提。对于配置来说，一致性也是一个重要的因素，因为它既提供了企业和它的客户的一

个统一视图，也确保了遵循规则及业务策略的需求。

实现端到端协作型业务流程：术语"端到端的业务流程"意味着成功地集成了不同企业的自动化业务流程和信息系统（通常涉及企业间的业务交易）。其目标是，向外延企业（Extended Enterprise，或称扩展企业）中的所有成员——从产品设计者、供应商、贸易伙伴、物流商到最终用户，提供无缝的互操作和互动关系。在这个阶段，组织将进入 SOA 实现的最高战略层次，服务的部署将无处不在，联合服务跨企业进行协作，从而创建更复杂的产品和服务。在外延企业中，单个的服务可能源自多个不同的提供者，无须考虑特定公司的系统或应用。

当在企业层次实施 SOA 时，或者实施一个跨企业的协作 SOA 时，会出现一个问题，就是如何管理 SOA 模型，如何在模型中将单元分类，以及如何组织它们才能使得不同的人都能理解。将 SOA 进行分层是非常合理的，SOA 包含许多不同的抽象层次，尤其是服务接口、服务实现以及将服务组合成更高层的业务流程［Amanjani 2004］。

将 SOA 进行分层的方式如图 1.6 所示。在图中，SOA 包含六个不同的层次：域、业务流程、业务服务、基础架构服务、服务实现及运营系统。通过定义一些公共的企业要素，每一个层次都描述了一个关注点逻辑分割。在图 1.6 中，每一个层次都使用它的下一层次的功能，再加上一些新的功能，从而完成它自身的目标。

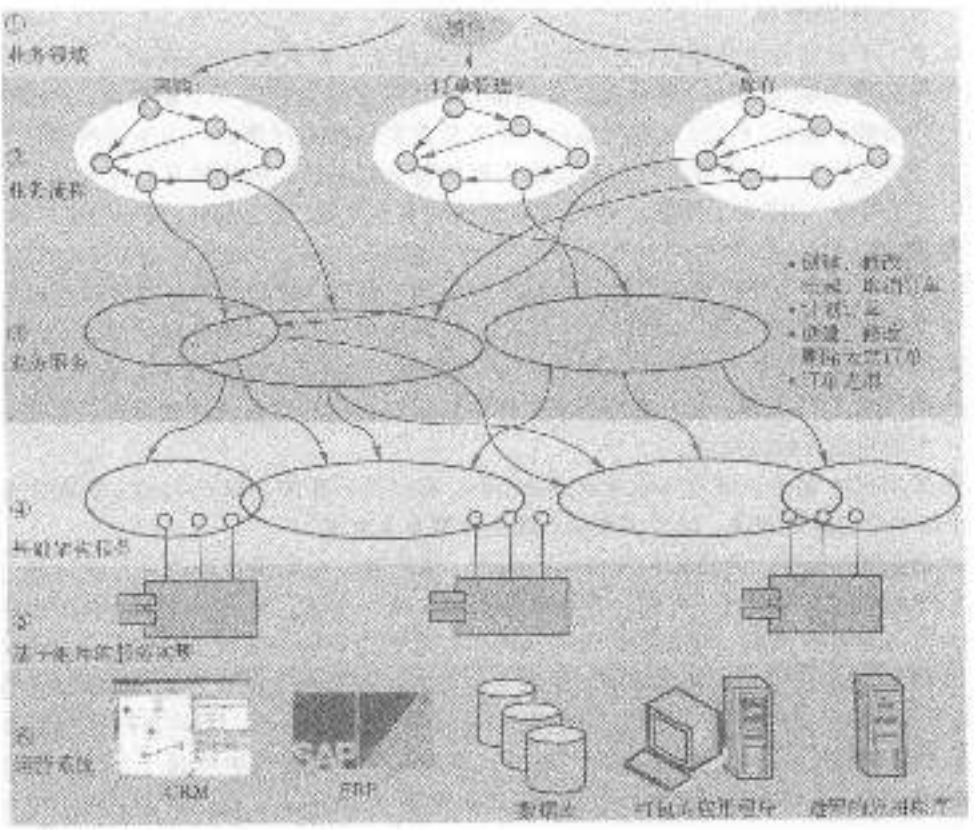


图 1.6 SOA 中的分层

在分层的 SOA 开发模型中，应用的逻辑流程主要致力于自顶向下的开发方式，该方式强调将业务流程分解为业务服务集合，并利用企业已有的应用程序实现这些服务。其他的方式还包括自底向上（主要用于企业信息系统）和较常见的中间会师式。自底向上的方式主要强调如何将

已有的应用转换为业务服务，以及如何将业务服务组合成业务流程。Web Service 的各种开发方式都属于 Web Service 的设计与开发方法学的一部分。我们将在本书的第 15 章介绍 Web Service 开发方法学。下面，我们将从顶层开始，依次向下分析分层的 SOA 开发模型。

最顶层(第一层)的划分是基于如下观察：企业中的所有业务流程都是服务于业务领域。业务领域是一个功能域，它包含若干当前或未来的业务流程，这些业务流程具有共同的性能与功能，并且这些流程之间能够相互协作，从而完成更高层次的业务目标，如贷款、保险、银行业务、财务、制造、营销、人力资源等。这样，一个企业将能划分为若干不相交的领域。在我们的案例中，假设业务划分为四个相关的领域，即销售、财务、制造和人力资源。这些业务领域可联系起来提供一个特定企业的完整的财务和运营视图。

SOA 从流程(诸如订单管理)的角度来看待企业，并将企业看成是一个良定义的、核心业务流程的集合。在 SOA 模型中，第二层是业务流程层，它是业务领域的划分，例如营销可以划分为几个核心业务流程，诸如采购、订单管理和库存，这些业务流程非常标准化，可在整个企业中使用，如图 1.6 所示。这三个流程是良定义的粗粒度流程。由于大量的细粒度的流程会导致极大的开销(参见 15.8.1 节，在该节中讨论了流程设计的准则)，从而导致系统低效，所以粒度问题是一个重要的设计议题。无疑，在很多情况下采用粗粒度的流程通常是更佳的选择。

现在我们主要分析一下订单管理流程，并解析其内容与功能。该流程通常按地区、产品或时间段进行订单量分析、利润分析、销售预测和需求预测。它也能按照产品、销售代表、顾客、仓库、订单类型、支付条件及时间段等，提供有关订单履约的汇总数据和交易的详细数据。此外，它还能跟踪订货量、支付的款项、过去的利润以及下一批货，并可取消每一个订单。在 SOA 中，我们可以将这类业务流程看作由人员、业务流程及业务服务间的接口组成。

在分层的 SOA 模型中，如何给诸如订单管理这样的流程规定合适的业务服务(由第三层组成)呢？方法之一就是将流程不断细分为更小的子流程，直到无法再分。对于实现来说，这些细分成的子流程就可以成为无形的(单独的)业务服务。业务服务自动处理一般的业务任务。这些业务任务对企业是有价值的，并且它们是标准的业务流程的一部分。企业使用这种方式分解的流程越多，则产生的这些子流程就越具有通用性。因此，企业将有机会构建合适的、可复用的业务服务集。

依靠联合业务服务集合的编配接口，这一层可实现可重组的端到端的业务流程(参见图 1.3)。具有不同粒度级别的服务或者服务集合都可组合起来或进行编配，从而构成“新的”复杂的服务。这些“新的”复杂的服务不仅具有新的复用级别，而且可用于业务流程的重组。

如图 1.6 所示，订单管理流程包含一些基本的业务活动，如创建、修改、挂起、取消、查询订单及计划订单，从而简化了业务服务。在这个流程中，业务服务用于创建和跟踪产品订单、服务或资源，以及获取客户所选择服务的详细信息。他们也能创建、修改和删除大宗订单及订购活动，并向客户通报订单和订购活动的进展情况。这类业务服务是细粒度的，它们自动处理一些具体的业务任务，这些业务任务是订单管理流程的一部分。

获取的订购单的部分信息包括顾客账户信息、产品销售计划信息等。可使用公共的数据词汇表来描述这类信息，从而使得不同的业务服务(这些业务服务可能属于不同的组织)间可以毫无歧义地解释消息、彼此相互通信，并能在订单管理流程中一起编配和使用这些不同的业务服务。有关解决服务消息中的语义歧义性的详细内容，可参见第 13 章，该章介绍了有关 Web Service 描述的语义增强机制。

在这一层中，使用诸如 WSDL(参见第 5 章)这样的服务描述语言，可将接口作为服务描述导出。同一服务描述可由许多不同的服务提供者实现，每一个实现都提供了不同的服务质量的选择。服务质量的确定基于可用性、性能、可扩展性和安全性等方面的技术需求。

当定义业务服务时，利用已有的应用程序中的逻辑并将它们暴露为一些服务也非常重要。所暴露的服务本身并不确定整个业务流程，而是确定实现流程的机制。这个过程中，将生成两类服务：业务功能服务和细粒度的公共服务集合。业务功能服务可以跨多个流程复用。公共服务也称为商品型服务(没有在图 1.6 中显示)，跨组织的业务都可共享它们。公共服务的例子包括实现计算、算法、目录管理等的服务。

SOA 需要一些基础架构服务。这些基础架构服务并不针对某一个特定的业务，而是可跨多个业务复用。基础架构服务⊖可划分为技术公共服务、访问服务、管理和监控服务、交互服务。第 4 层中的技术服务是粗粒度的服务，提供了开发、交付、维护的技术基础架构，还提供了单个的业务服务(在第三层)、由业务服务所集成的流程(在第二层)，以及提供了维护诸如安全性、性能和可用性这类 QoS 的能力。为了实现既定的目标，技术服务需要依赖一些可靠的功能，诸如智能路由、协议转换、事务管理、标识管理、事件处理等。它们也包括一些机制，可将企业内的服务无缝地进行互联。例如，这可包括策略、约束以及具体的行业消息和互换标准，诸如需要遵循的标准，如 EDIFACT、SWIFT、xCBL、ebXML BPSS 或 RosettaNet。对于处于垂直市场中的企业来说，若要和其他类似的流程协作，必须遵循这些标准。访问服务用于转换数据以及将遗留应用和功能集成到 SOA 环境中。这包括遗留功能的打包和服务使能。在有关分布式计算的文献中，访问服务通常被称为适配器(参见 2.7.1 节)。在分层 SOA 模型中，访问服务的作用与传统的适配器并不相同，它们可将遗留系统和企业应用系统中的一些资源转换为单独的业务服务和业务流程。基础架构服务也提供管理与监控服务。管理服务不仅管理系统内的资源，而且可以跨系统管理资源。管理服务收集所管理的系统和资源的状态信息、性能信息，并提供一些具体的管理任务，诸如失效的根本原因分析、服务等级协议监控和报告、容量规划。监控服务监控 SOA 应用的状态，严密监控系统和网络的运行情况，深入了解应用程序的状态和行为模式，从而为关键任务提供了更合适的计算环境。管理和监控服务依赖于新近诞生的一些标准，诸如 WS-Management，本书将在第 16 章讨论这一标准。服务与外部世界的交互并不仅局限于人机交互。在有些情况下，交互逻辑需要将接口编配到交通工具、传感器、无线射频识别(RFID)技术设备、环境控制系统、流程控制设备等。所有的基础架构服务都可视为企业服务总线的一个有机组成部分。在 SOA 环境中，企业服务总线使得标准化的集成成为可能。

在图 1.6 中，第五层是组件实现层，用于实现运营系统中已有的应用和系统的服务。这一层使用组件实现所需的功能。第四层的技术服务然后将使用这些组件实现来定义业务服务的内容，以及构建业务服务。例如，使用转换程序，并将业务服务与其他相关的业务服务进行编配，从而创建业务流程。对于服务实现来说，组件技术是首选技术(在 1.10 节中，对 Web Service 与组件技术进行了比较)。组件包含自治的软件单元，这些软件单元可向客户端(业务服务)提供有用的服务或一些功能，并可与进行互操作的其他组件隔离。因为组件是自治的功能单元，所以它们的实现也自然是界限分明的。业务服务的实现通常包括应用的完整装配以及跨不同应用的潜在集成起来的中间件的功能。这类功能是由第六层中的运营系统提供的。

⊖ 从此以后，我们将不区分业务服务和基础架构服务这两个术语，而是统称为 Web Service，除非在需要区分它们的时候。

最后，组件使用第六层中的运营系统实现业务服务和流程。如图所示，第六层包含已有的企业系统或应用，包括客户关系管理(CRM)和 ERP 系统与应用，遗留应用，数据库系统与应用，其他打包的应用程序等。这些系统通常称为企业信息系统(参见 2.9 节)。这展明了 SOA 如何使用面向服务的集成技术来利用并集成这些已有的系统。

1.7 Web Service 的技术架构

Web Service 技术使得应用程序间可以基于标准的因特网协议进行协作，而无须人的直接干预。借此，可将许多业务操作自动化，创建新的功能效率及新的更有效的业务开展方式。Web Service 范式对基础架构的要求并不高，其目的就是确保在任何平台上使用任何技术和编程语言都可以实现和访问 Web Service。

实际上，Web Service 的实现方式并不是唯一的，而是代表了几类相关的技术。Web Service 一个普遍接受的定义基于如图 1.7 所示的一套具体的补充标准。开发被普遍接受的开放标准是 Web Service 基础架构开发联盟的一个重头戏。同时，如图 1.7 所示，这些工作也导致了大量的新标准、新术语的诞生。为了简化问题，我们对 Web Service 技术架构的一些重要标准进行了分类，并在下面进行了简要介绍。

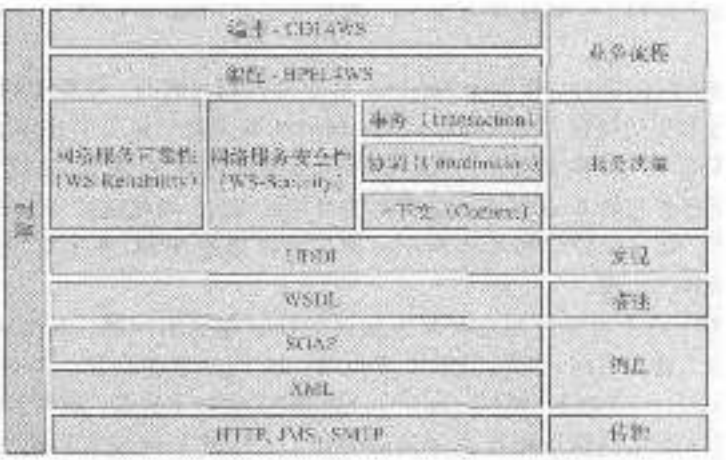


图 1.7 Web Service 技术架构

使能技术标准：虽然 Web Service 并没有限定采用任一特定的传输协议，然而 Web Service 使用互联网连接和基础架构进行构建，从而确保几乎无障碍的连接，并能得到广泛的支持。例如，Web Service 在传输层利用了 HTTP 协议，也即 Web 服务器和浏览器所使用的连接协议。Web Service 的另一个使能技术是可扩展的标记语言(XML)。XML 是一个被广泛采用的格式，用于交换数据及相应的语义。对于 Web Service 技术架构中的其他任何层，Web Service 基本都是用 XML 作为基础构造块。

核心服务标准：Web Service 的核心标准包括基本标准 SOAP、WSDL 和 UDDI：

简单对象访问协议：SOAP 是一个基于 XML 的简单的消息协议，Web Service 依靠该协议进行相互间的信息交换。SOAP 协议基于 HTTP，并使用诸如 HTTP 这样的常规的因特网传输协议来传送数据。为了进行 Web Service 间的相互通信，SOAP 实现了一个请求/响应模型，并使用 HTTP 来穿透防火墙，将防火墙配置为接受 HTTP 和 FTP 服务请求。在第 4 章中，我们将会更详细地讨论 SOAP。

服务描述：当 Web Service 和它的客户端之间在完成一些任务时，诸如指定数据和操作、表示 Web Service 契约，或者了解 Web Service 的功能，若使用一些标准的方法则会更为有效。为了实现这一点，首先需要使用 Web Service 描述语言（WSDL）来描述 Web Service 的功能特性。WSDL 定义了 XML 语法，将服务描述为能够交换消息的通信端点的集合。在第 5 章中，我们将更详细地讨论 WSDL。

服务发布：使用 UDDI 可进行 Web Service 的发布。UDDI 是一个公开目录，可提供在线服务的发布，并有助于 Web Service 的最终发现。公司可以发布它们所提供的服务的 WSDL 规范，其他公司根据这个 WSDL 描述来访问那些服务。这样，独立的应用程序将能公布应用流程或任务，以便其他远程的应用程序或系统可以使用这些流程或任务。在企业的注册机构中，企业简介中通常提供了这些 WSDL 规范的链接。在第 6 章中，我们将更详细地讨论 UDDI。

服务的组合与协作标准：这些包括下列标准：

服务组合：对于基于 Web Service 的应用程序，通过定义它们的控制流（诸如条件、顺序、并行和异常处理），以及指定规则一致地管理那些不可观测的业务数据，即可以描述 Web Service 应用程序的执行逻辑。这样，企业可以描述横跨多个组织的业务流程，诸如订单流程、潜在客户管理、投诉处理。并且在系统中，企业可以从其他的供货商那里执行同样的业务流程。业务流程执行语言（BPEL）可实现 Web Service 的服务组合[Andrews 2003]。在第 9 章中，我们将讨论 BPEL。

服务协作：对于跨企业的一些 Web Service，可定义它们共同的、可观察的行为。例如，Web Service 在何处通过共享的接触点交换同步信息，何时能够满足所定义的排序规则。通过定义这些行为，可描述 Web Service 间的协作。Web Service 编排描述语言（WS-CDL）[Kavantzas 2004]可指定业务协作中所有参与的 Web Service 的共同的可观测行为，因此通过 Web Service 编排描述语言可实现服务协作。每一个参与的 Web Service 不仅可以通过 BPEL 来实现，也可使用其他的可执行的业务流程语言来实现。

协调/事务标准：对于与服务发现及服务描述的检索相关的问题，成功地解决它们是 Web Service 是否成功的关键。当前，正尝试定义 Web Service 间的事务交互。Web Service 协调（WS-Coordination）和 Web Service 事务（WS-Transaction）对 BPEL 进行了补充，提供了定义具体的标准化协议的机制，这些标准化协议可用于事务流程系统、工作流系统或者其他需要协调多个 Web Service 的应用。在一些电子商务应用场合，可能需要连接或执行许多 Web Service，并且这些 Web Service 可能在不同的平台上运行，并横跨多个组织。这三个规范相互配合，可解决上述场景中的业务工作流方面的问题。在第 11 章中，我们将比较详细地讨论 Web Service 协调和 Web Service 事务。

增值标准：在 Web Service 能真正地自动处理关键的业务流程之前，仍然必须实现支持复合业务交互的一些其他的要素。增值服务标准包括安全性和认证机制、授权机制、信任机制、隐私机制、安全会话机制、合同管理机制等。在第 11 章、第 12 章和第 16 章中，我们将讨论诸如 Web Service 安全性（WS-Security）、Web Service 策略（WS-Policy）和 Web Service 管理（WS-Management）等增值服务标准。

目前，包括 IBM、微软、BEA 和 SUN 微系统在内的一些公司提供了跨 Web Service 功能域的产品和服务，并实现了 Web Service 技术架构。通常将这些公司视为平台提供者。他们以应用服务器的形式提供基础设施，例如 WebSphere、.NET 框架、WebLogic，用于构建和部署 Web Service。此外，这些公司还提供一些工具，可进行业务运营中的 Web Service 的编配和/或组合应用开发。

1.8 服务质量(QoS)

基于 SOA 的应用程序必须可靠地运行，并需要能够提供多种级别的一致服务，这是一个重要的要求。该要求不仅重视服务的功能属性，而且致力于描述驻留 Web Service 的环境，例如描述服务的非功能性方面的能力。Web 实际应用中，对 Web Service 往往有许多不同方面的技术要求，如各种级别的服务可用性、性能、可伸缩性、安全性和隐私策略等，因此需要能够描述所有的这些技术需求，并且要求驻留每一个服务的环境能够基于不同的技术要求提供不同的 QoS 选择。显然，对于服务提供者和他们的客户而言，Web Service 所提供的 QoS 已成为一个非常重要的问题。

QoS 指的是 Web Service 的一种能力，它能响应预期的请求，并能以一定的服务质量完成相关的任务，并且所提供的服务质量符合服务提供者与客户的预期。质量方面的这些要素反应了客户的期望，诸如稳定的服务可用性、可连接性以及快速的响应能力。因为这些要素对于服务提供具有很大的影响，所以它们对于保持业务的竞争力和可行性非常关键。因此，QoS 已成为确定服务的使用性能与效用的一个重要标准。服务的使用性能与效用这两者都将影响特定的 Web Service 的广泛应用，并且它们也是一个重要的卖点，以此可以区分不同的服务提供者。

由于因特网具有变化很快、不可预测的特性，因此如何在因特网上保证 QoS 是一个非常大的挑战。具有截然不同的特性和需求的各类应用相互之间竞争各种网络资源。许多方面的问题都对因特网 QoS 标准提出了一定的需求，这些问题包括流量模式的变化、关键业务的安全传输、基础设施失效的影响等。若 QoS 出现问题，通常会导致一些非常重要的事务性应用程序的性能急剧下降，以致不可接受。

传统上，QoS 是按照应用、系统、网络以及其他的 IT 基础设施的可用性的程度来进行度量的。根据各种访问方式和负载情况下所需的性能级别，来度量可用性。当采用这种传统的 QoS 度量方式时，既对应用程序的可用性提出更高的要求，又增加了访问和管理服务的复杂性。因此在度量 QoS 时，将会对企业提出更明确的、更高的需求。在 Web Service 应用中，QoS 可以视为是对数量特性集的保证。可以根据一些重要的功能性和非功能性的服务质量属性以及其他一些重要的服务特性来定义 QoS。其中，服务质量属性既包含实现阶段也包含部署阶段，而其他一些重要的服务特性则诸如服务计量和开销、性能度量(例如响应时间)、安全性需求、(事务)完整性、可靠性、可伸缩性和可用性等。这些特性对于了解服务的整体运转情况非常重要，从而其他的应用和服务可以绑定到该服务中，并可作为业务流程的一部分执行。

(1) 可用性：可用性是指服务正常运转的时间，它表示了服务正常运转的概率。它的值越高，意味着能够正常使用服务的可能性越大；它的值越低，则意味着在任何时候服务出现不可预知的故障的可能性越大。与可用性有关的还有修复时间(time-to-repair，简称 TTR)。TTR 表示修复已经失效的服务要花费的时间。理想情况下，TTR 值越小越好。

(2) 可访问性：可访问性表示了 Web Service 请求能够被服务的程度，也可以通过一个概率来表示在一个时间点上服务能够成功地实例化的比率。可访问性比较高意味着大量的客户都可使用该服务，并且这些客户能够相当容易地使用该服务。

(3) 符合标准：描述了 Web Service 遵循标准的情况。服务请求者若要能够正确地调用 Web Service，需要服务提供者严格遵循标准的合适版本(例如 WSDL 第二版)。此外，服务请求者和服务提供者之间可进行服务级别的协商，服务提供者必须严格遵循所协商的标准。

(4) 完整性：描述了 Web Service 按照它的 WSDL 描述及服务等级协议(SLA)完成任务的情况。完整性越高意味该服务的功能越接近它的 WSDL 描述或 SLA。

(5) 度量性能的指标有两个：吞吐量和等待时间。吞吐量表示了服务在一个特定的时间段内所能服务的请求数。等待时间指的是客户端在发送请求和收到响应之间的时间间隔。吞吐量越大、等待时间越短，说明 Web Service 的性能越好。对 Web Service 的请求数有可能是稳定的，也有可能由于发生某些特定的事情而极其不稳定，诸如工作日和非工作日、淡汪季等。无论 Web Service 的请求数是否稳定，度量 Web Service 所处理的事务/请求数的比例都是非常重要的。

(6) 可靠性：可靠性是指服务能够正确地、始终如一地运行，并且无论系统或网络是否发生故障，都能提供同样的服务质量。Web Service 的可靠性通常是通过每月或每年的事务故障数来表示的。

(7) 可伸缩性：可伸缩性是指，伴随服务请求的需求量发生变化，服务能力也能进行相应的变化。通过构建高度伸缩性的系统，Web Service 将能提供较高的可访问性。

(8) 安全性：安全性包含许多方面，诸如认证、授权、消息完整性、机密性等(参见第 11 章)。当在因特网上调用 Web Service 时，安全性尤其显得重要。在 Web Service 的 SLA 中，可描述 Web Service 所需的安全性，并且 Web Service 提供者需要保证该安全级别。

(9) 事务性：Web Service 所需的事务行为和上下文传播有几类不同的情况。在 Web Service 的 SLA 中，可描述 Web Service 所需的事务行为，并且 Web Service 提供者需要维护这一属性。

企业需要依靠业务部门、合作伙伴及外部的服务提供者来提供服务。为了确保所选择的服务提供者能提供所需的服务质量的等级，需要使用 SLA。SLA 是提供者与客户之间的一种正式协定(契约)，在某种程度上对 Web Service 的详细情况(内容、价格、交互过程、验收标准、质量标准、罚款等，通常都是可度量的内容)进行了形式化，从而保证了服务提供者和服务请求者之间的相互了解、相互预期。

SLA 基本上是一个服务质量保证，通常通过逆向计费或其他的一些机制来支持服务质量保证。这些机制可用于补偿服务的用户，并对企业实现 SLA 承诺发挥影响。了解业务需求、所希望的使用模式及系统能力，对于成功地部署 Web Service 大有帮助。由于服务提供者和服务客户同样都使用 SLA，因此 SLA 是维护服务提供关系的一种重要的、广泛使用的手段。

SLA 可以包含下列部分[Jin 2002]：

- Purpose(目的)：该域描述了创建这个 SLA 的原因。
- Parties(各方)：该域描述了 SLA 中涉及的各方以及他们各自的角色，例如，服务提供者和服务消费者(客户)。
- Validity period(有效期)：该域定义了 SLA 的有效期，通过协议期限的起始点时间与结束时间来界定。
- Scope(范围)：该域定义了协议所涉及的服务。
- Restrictions(限制)：该域定义了一些必须的步骤，用于得到所请求的服务等级。
- Service-level objective(服务等级目标)：该域定义了服务请求者和服务的用户之间相互协定的服务等级。服务等级包含若干指标，诸如可用性、性能、可靠性等。服务等级的每一个方面都是一个需要实现的目标。
- Penalties(惩罚)：该域定义了一些制裁措施。当服务提供者未能合适地提供服务，或者服务无法满足 SLA 中所指定的目标时，将应用这些制裁措施。
- Optional services(可选服务)：对于用户通常不会求，但是在一些例外情况下可能请求的服务，在该域中进行了指定。
- Exclusion terms(豁免条框)：指定了一些在 SLA 中没有涉及的内容。

- Administration(管理)：该域描述了流程及 SLA 中的可度量的目标，并定义了管理服务的企业权责。

SLA 既可以是静态的，也可以是动态的。在多个服务时间间隔中，静态 SLA 一般都保持不变。服务时间间隔既可以是在 SLA 中指定的业务流程的月历，也可以是事务的时间周期，或者其他流程的可度量的、相关的时间周期。它们可用于评估服务提供者和用户之间所协定的 QoS。在不同的服务期中，为了顺应所提供的服务的变化，动态的 SLA 通常会发生相应的变化。

对于 Web Service SLA，需要应用一些特定 QoS 度量指标。在不同的服务时间间隔中，会根据所定义的目标来评估这些度量指标。度量 SLA 中的 QoS 的等级最终涉及从多个方面(区域、技术、应用和供应商)追踪 Web Service。

在一个典型的应用中，每一个 Web Service 可以与多个 Web Service 进行交互。角色是可以变换的，一个交互中的服务提供者可能是另一个交互中的服务请求者。这些交互都可以由 SLA 潜在地控制。SLA 所指定的度量指标与服务的总体目标相关。因此，SLA 的一个重要功能就是在源头上处理 QoS，它与服务所提供的服务等级相关[Mani 2002]。

对了更明确地度量 SLA，可将前面所介绍的 Web Service 的 QoS 要素分为下面三大类：

(1) 性能和能力：这一类包括事务量、吞吐率、系统规模、利用水平、底层系统是否设计为可满足最大负荷测试(并经过测试)，以及请求/响应时间的重要性。

(2) 可用性：这一类包括整个系统或其中的部件能够正常运行的平均时间、灾难恢复机制、恢复的平均耗时，业务是否能够容忍 Web Service 的故障以及最大可以容忍的故障时间，以及是否具有充分的冗余，从而使得即使系统或网络出现故障，依然能提供服务。

(3) 安全性/隐私：这一类包括对系统入侵的应对、隐私顾虑、所提供的认证/授权机制等。

为了实现 QoS，Web Service 的一些标准支持标准策略框架，从而使得开发者能够表示服务策略，并且 Web Service 也能理解策略并在运行时实施这些策略。Web Service 框架(WS-Policy)[Bajaj 2006a]促进了这项工作。Web Service 框架提供了一些构件，这些构件可与其他的 Web Service 及特定的应用协议协作，从而有助于策略的表示、交换和处理，而这些策略控制了 Web Service 端点间的交互。

1.9 Web Service 的互操作性

Web Service 的规范、实现和最佳实践都已经逐渐建立起来了。鉴于许多相关的规范有不同的版本，并且可能有新的开发计划，因此确定产品支持何种 Web Service 规范变成了一个非常困难的问题。在许多情况下，产品实现规范的版本不一，从而阻碍了产品间的完全的互操作性。这需要每一个企业对于规范的使用都要提供各自的说明，这将导致 Web Service 应用间的相互隔离，使得这些 Web Service 应用仅能在一个局限的范围内使用，从而严重阻碍了 Web Service 的互操作性。Web Service 的互操作性还需要解决其他的一些问题，例如对于标准的解释的歧义问题，对不同规范间交互的理解不够等(参见 www. ws-i. org)。

Web Service 互操作组织(Web Services Interoperability Organization，WS-I)致力于 Web Service 的互操作。WS-I 是一个开放的产业联盟，其宗旨是促进跨平台、操作系统和编程语言的 Web Service 的互操作性。WS-I 的工作横跨产业界和标准组织，向开发者提供了开发指南、最佳实践及一些资源，用于 Web Service 互操作解决方案的开发。

WS-I 在 Web Service 的规范中提出了互操作性概要(Profile)这一概念。互操作性 Profile 标识了目标 Web Service 技术，并澄清了 Web Service 单独使用或联合使用的使用方法。WS-I Profile 包含了一张列表，这张列表是为实现特定功能而组合到一起的一组规范。此外，对于如何使用规范

来开发互操作的 Web Service，WS-I Profile 提供了具体的实现指南。为了支持通用功能的 Web Service 的互操作性，WS-I 正在开发一组核心 Profile 集。通过 Profile，可以很容易地在同一粒度级别讨论 Web Service 的互操作性，从而使开发人员、用户或对 Web Service 和 Web Service 产品做出投资决策的主管人员可以在一个相对固定的层次相互理解和沟通。

Basic Profile 1.0 既包括了如何使用核心 Web Service 规范开发互操作的 Web Service 的实现指南，也包含了有关消息、描述和发现方面的协议。那些规范包括 SOAP 1.1、WSDL 1.1、UDDI 2.0、XML 1.0 和 XML Schema。Basic Profile 1.0 为 Web Service 统一标准打下了坚实的技术基础，例如 Java 规范的下一个重要版本 J2EE 1.4[⊖]、IBM WebSphere Studio 开发环境等。Profile 1.0 为实现互操作解决方案提供了一个通用框架，对客户的采购决策提供了一个常规的参考点。同时，WS-I 也已开发了 Basic Profile 1.1，它是 1.0 版本的进一步扩充。WS-I 正致力于新建一个工作组——Reliable Secure Profile Working Group，该工作组将负责提交 Web Service 体系结构指南，以及开发者所关心的安全、可靠的消息机制。

WS-I 最主要的工作成就是测试工具，开发人员可以使用这些测试工具测试他们所开发的 Web Service 是否遵循事先的测试声明。测试声明描述了所建立的 WS-I Profile 的互操作性指南。这个流程用于开发 Profile、互操作性指南、测试声明，并且测试工具为开发者生成了其他一些有用的相关资源。测试工具可监测并记录和 Web Service 间的交互，并对它们进行分析，从而检测出实现中的错误。

1.10 Web Service 与组件的比较

开发 Web Service 的主要目的是提供分布式应用的基本框架，这些分布式应用之间能够以某些形式进行相互通信，从而促进应用的相互了解、复用、开发和集成。乍一看，分布式组件间涉及一些共同的关注事项，因此很可能将 Web Service 和组件简单地视为同一类分布式计算，仅是风格不同而已。然而，若对 Web Service 和组件进行进一步的比较可以发现，分布式组件通常用于开发紧耦合的解决方案，而 Web Service 的目的并不是定义一个新的组件模型，而是定义一个功能分布的服务规范，该规范可以应用于任何已有的组件模型、编程语言或执行环境。

Web Service 和组件所支持的集成方案的主要需求是实现多层次的中立性，通过建立抽象层隐藏端点实现的细节。抽象层既可以实现为 Web Service，也可以实现为组件。对于不同的平台、语言、应用程序组件模型、事务模型、安全性模型、传输协议、调用机制、数据格式、端点可用性模型等，集成方案都必须是中立的。这一目标可以很自然地分为下列五个方面：通信模式、端点间的耦合类型、接口类型、调用类型、代理类型。我们将从这几方面对 Web Service 和分布式组件进行一个简要的对比。

通信类型：分布式组件间的交互基于 RPC 范式，通常在多个请求中传递少量的数据项，并且同步地获取少量的返回数据。在组件方式中，网络通信基于细粒度的对象，这通常是不可靠的，并且开销很大。

组件使用同步通信，然而 Web Service 却既使用同步通信，也使用异步通信。对于简单的 Web Service，可以采取 RPC 类型的请求/响应同步通信，进行细粒度的对象交互。而对于复合的 Web Service，需要采用更松散的异步通信模式，通常采用基于消息的系统。

端点间的耦合类型：分布式组件依靠紧耦合方式进行交互。在紧耦合的交互中，通常需要调

⊖ 目前，J2EE 的最新版本是 J2EE 1.6。——译者注

用多个细粒度的 API(应用编程接口)。这类紧耦合的交互主要依赖于一点,即设计应用的模型需要能够得到普遍接受。这迫使客户端和服务端都需要采用类似的基础架构,从而使得分布式组件平台间不能很方便地进行互操作。例如,CORBA 需要所有的应用都遵循接口描述语言(IDL),并需要使用对象请求代理,而远程方法调用(RMI)则需要进行通信的所有实体都是使用 Java 语言编写的。对于这类与特定的组件技术紧耦合的实现,在受控环境中是完全可以接受的。然而在 Web 环境中,紧耦合的实现将变得不切实际,并且无法做到可伸缩性。在集成的业务流程中,参与者可能会不断发生变化,所采用的技术也可能会不断发生变化。这就使得保证所有的参与者都采用统一的基础设施变得非常困难。

另一方面,Web Service 并不使用特定的应用接口进行相互绑定,而是使用抽象的消息定义来进行相互间的绑定,即使用消息定义来取代方法签名。通用的消息定义使得应用程序能独立处理具体的复杂消息,从而可以复用接口。由于仅需要处理消息,Web Service 完全无须关心具体的语言、平台和对象模型,从而使那些请求该服务的应用可以在任何平台上使用任何语言来实现。

接口类型:组件将细粒度的对象层接口暴露给应用程序。在分布式组件方式中,对于接受者如何激活应用程序,以及对于被调用的这类接口及它们的签名,发送者都进行了许多假设。与此相反,Web Service 仅需要暴露应用层接口。另一方面,Web Service 所使用的消息系统在有线格式(wire format)层形成约定。请求者唯一需要假定的就是接收者能够理解所发送的消息。请求者既无须假定接受消息后会发生什么,也无须假定发送者和接收者之间可能发生的事情。

在 Web Service 方式中,应用层接口是粗粒度的接口,描述了在业务层有用的服务。例如,在 Web Service 方式中,库存服务将暴露库存补给服务和相关的参数。与基于组件的方式不一样,Web Service 方式不会暴露库存对象及它的所有接口,也不会暴露补给服务对象和它的接口,它们与业务应用都没有直接的关联。

调用类型:组件通过名称查找服务,例如 CORBA 使用命名上下文(Naming Context,也称命名环境)。与此相反,在 Web Service 中引入了"服务能力"这一概念。服务能力描述了分类、功能及发布、发现和调用特定服务的条件。例如,我们可以发现许多业务可提供的服务的类别,例如制造或物流服务,并可基于价格和 QoS(包括响应时间、负载平衡等技术参数)选择最合适的服务。

请求代理的类型:分布式计算的传统方式(例如组件框架)使用预定义的接口来调用远程对象:使用服务的程序能够了解目标服务的消息的格式。服务可使用完全不同的绑定方式:静态绑定和动态绑定。在静态绑定中,应用程序在设计时就了解协作服务的各种详细信息。而在动态绑定中,应用程序通过服务代理来确定协作服务。

Web Service 和组件的其他重要的不同方面还包括:1)Web Service 广泛使用了开放标准,而组件技术却缺乏开放标准;2)Web Service 可以提供一些组件技术不具有的高级功能。这些高级功能包括运行时协商的 QoS(按照 QoS 标准,在不同的服务提供者之间进行选择)、反应式服务和流程(服务能根据具体的环境进行调整,并且不会影响运行效率)、工作流、协调,以及基于服务的应用程序的事务部件。

综上所述,对于体系结构定义明确、主要面向内联网和企业内部集成的应用,最适合使用组件系统构建。例如,CORBA 这类的分布式组件技术比较适合构建受控环境中的分布式应用。在这样的环境中,所有的分布式实体间可能共享通用的对象模型,并且对分布式实体间通信的粒度大小和通信量的多少没有限制。此外,这类环境中的系统部署相对稳定,因此可直接将网络地址变换为对象引用。对于集成各不相同的端点,分布式组件技术提供了很好的支持。然而,对于构建业务流程管理的解决方案,分布式组件技术却并不适用(至少不是非常适合)。若要构建跨企业的分布式系统,需要创建技术协议和协调,然而分布式组件技术很难做到这些。

因为 Web Service 提供了一个语义丰富的集成环境，所以可以使用 Web Service 构建企业内部或跨企业的业务流程管理解决方案。Web Service 最适合用于实现企业间的共享业务，然而也可使用 Web Service 集成企业应用(参见 2.9 节)。

1.11 Web Service 的优与劣

本章阐述了 Web Service 的性质和特点，并主要讨论了 Web Service 的优点所在。尽管 Web Service 在提高效率及扩展应用组合方面大有可为，然而 Web Service 技术仍有大量的工作需要进一步改进，以便使得 Web Service 应用能实际应用到使用正规标准的、运行关键任务的生产环境中。

本节将首先描述 Web Service 对于业务应用所带来的影响。然后，本节将确定 Web Service 强壮性、可靠性、安全性和可管理性方面的一些主要技术议题，以及将这些技术应用到未来的挑战性的、执行关键任务的工业级环境中。

Web Service 技术的真正潜力在于它既能支持常见的业务问题，又能根据市场需要的变化做到随需应变。基于 Web Service 技术及精心设计的面向服务的体系结构，企业可以复用已有的业务功能，并能加快各种新的应用的实现。因此，企业通常将 Web Service 视为保护其在信息库、应用系统和业务流程方面(无论它们位于企业内部还是跨整个延伸的价值链)投资的重要手段。

Web Service 最吸引人的特点在于其不断发展，从而能够包容电子商务、企业应用集成、传统的中间件以及 Web 技术。Web Service 不是取代传统的中间件，而是与中间件技术和企业应用集成技术协作，并提供一个简化的基于标准的集成方式。为此，Web Service 提供了：

- 将遗留应用的功能转换为可复用的、自包含、自描述的服务的标准方法。这些转换而来的服务能够以一种标准的、可管理的方式与其他服务进行互操作。
- 方便、灵活地进行应用集成的标准方式。通过这种方式，可将实现应用功能的已测试过的、可信的、可互操作的模块快速装配成新的应用。
- 开发、装配纯粹的互联网应用的标准方法。这些互联网应用既可以用于企业内部，也可用于扩展企业。该方法可将内部或外部创建的服务作为构件，并将这些构件装配到应用中。
- 跨企业的系统的公共外观(facade)，从而更容易创建 B2B 集成所需的 SLA。

具体的业务需求正驱使着 Web Service 变得越来越复杂，Web Service 技术还需要能够处理具体需求的一些其他能力。目前的具体需求正严格地考验 Web Service 技术，并暴露了 Web Service 技术的一些明显的不足之处。这些不足包括：对于复杂的事务管理缺乏有效的支持，缺乏表达业务语义的手段，尤其是对于大量的已有标准及新制定的标准缺乏一致的认同和协调。由于 Web Service 技术目前尚存在这些不足，这就降低了企业采用 Web Service 技术的价值，并给企业带来了一定的风险。下面，我们将分析 Web Service 的不足之处。

Web Service 饱受非议的一个问题是，在不同类型的 Web Service 软件间的数据的共享和应用。这种数据共享和应用可能是一些简单的任务，例如将手提式计算机中的视频片段发送到台式计算机中。这种数据共享和应用也可能是一些较大的任务，例如在几个合作者之间交换大型文档，这种交换可能会严重影响性能。导致这个问题的主要原因在于 Web Service 应用是基于 XML。当对图片这样的二进制数据进行 XML 格式的编码以及进行跨网络传输时，会占用大量的带宽，并会显著降低应用的响应速度。

最近，Web Service 标准化组织已经开始大力着手于提高 Web Service 的性能、效率和企业适用性。在 2005 年 1 月，万维网联合会(W3C)已经出台了三个最新的标准，帮助供应商改善 Web

Service 的性能。这三个标准分别是 XML 二进制优化打包(XML-binary Optimized Packaging, XOP)、SOAP 消息传输优化机制(Message Transmission Optimization Mechanism, MTOM)、源请求报头区块(Resource Representation SOAP Header Block, RRSHB)。这三个规范用于帮助开发者在 SOAP1.2 消息中将二进制数据进行打包和发送。

第一个规范 XOP 表示了在包中将二进制数据包括进 XML 文档的标准方法。因此,降低了应用程序存储这些数据所需的空间,也降低了应用程序传输这些二进制数据所需的带宽。第二个规范 MTOM 也使用 XOP 传送 SOAP 消息,并对消息的传送进行优化。第三个规范 RRSHB 使得 SOAP 消息接受者能够访问缓存的消息内容。在带宽和大小受到制约的场合,RRSHB 给用户提供了更多的选择。用户既可以选择访问实际的文件,也可以选择访问缓存的备份。

XOP 的目的是使得 XML 能够支持大型的二进制媒体文件。XOP 也是 XML 得到广泛应用的象征,使得 XML 的应用已经超越了原先的界限。另一方面,通过对 SOAP 消息流进行整形,MTOM 和 RRSHB 提高了消息传输的效率。

目前 Web Service 的事务标准正在不断发展之中,仍然还很不成熟。在第 10 章中,我们将分析 Web Service 的事务标准,如 WS-Transaction(Web Service 事务)和 Web Services Composite Application Framework(Web Service 复杂应用框架)。因此,对于提供事务服务的 Web Service,需要进行仔细的规划、设计和实现。当构建基于服务的事务解决方案时,维护互操作性、减少客户端集成的复杂性以及利用合适的标准(应该采用那些在行业内广为接受的标准)都是非常重要的,需要密切关注。通过创建和部署专门的扩展,就有可能设计可靠、安全的业务事务和高级应用。然而,开发这样的特定解决方案很花时间且开支巨大,并且企业必须分别和每一个合作伙伴或客户解析这些解决方案。这种方式明显有损 Web Service 的跨企业的互操作性这一主要优点。因此,对于复杂的 Web Service,需要提供一些高级的事务能力,以便充分发挥 Web Service 开发的优势。

与业务事务相比,Web Service 的安全领域已经取得一些进展。Web Service 的安全规范 WS-Security 描述了如何使用已有的 W3C 安全性规范、XML 签名和 XML 加密,从而确保 SOAP 消息的完整性和机密性。XML Web Service 具有全面的、模块化的安全性体系结构,这些规范形成了这一体系结构的底层基础。未来的安全性规范将要以这些能力为基础,提供可信交换机制、信任管理、回退及其他一些高级能力。

若正确使用 SOA,可以很快地构建应用,从而提供应用和流程集,并且这些应用和流程可作为一个标准的、可理解的单元。然而,当业务流程使用不同的术语进行跨企业的相互通信时,可能会带来混乱。并且更重要的是缺乏被普遍接受并理解的业务流程和业务协议,这些业务协议驱动业务流程间的交换和交互。在 SOA 中,尽管底层的信息结构、业务流程和工件的特性有所不同,服务请求者和服务提供者之间进行有意义的通信非常重要。

幸运的是,在不同的业务领域中已经成功地应用了语义 Web 中的最主要的形式体系——资源描述格式(Resource Description Format, RDF)。RDF 是一个描述和交换元数据的规范,这一规范已经在知识管理、知识工程、生产支持和大规模数据库应用中得到广泛的实际运用。

Web Service 的许多不同标准之间相互重叠或相互冲突,这也是阻碍 Web Service 广泛应用的最大障碍之一。对于制定标准的主导权,至少有四家组织正在进行相互竞争,它们分别是 Liberty Alliance、Oasis、W3C 和 WS-I。它们的目标不同,并且职权和影响力也各不相同。除了这些标准化组织定义 Web Service,其他的一些供应商联盟也定义了一些不同的规范。一开始,供应商联盟就已经开发了许多 Web Service 规范。他们在开发规范一段时间后,当认为规范已经比较成熟时,就会将该规范提交给标准化组织以供实施。当前,供应商已经出现两个对立的阵营,一个阵营是

IBM 和微软之间所达成的一个不稳定的联盟，另一个阵营是 Sun、Oracle、SAP 和其他的一些供应商。这两个阵营所提出的 Web Service 规范，有些是专有的，有些则不是——专利和许可证问题还比较含糊。

目前有大量的已有标准和新诞生的标准。其中有些标准尚在变化发展之中，需要相互配合才能最终实现。因此，试图了解这些不同的标准也是一件令人望而生畏的任务。除非这些标准十分成熟，它们之间才能协调和相互协作。Web Service 在关键任务、生产计算环境中的长久生存能力仍然有待进一步的验证。

尽管存在以上种种问题，在这两个对立阵营之间已经出现了相互协作的一些良好苗头。例如，在 2004 年 8 月，IBM、微软、BEA、SAP 和 Sun 微系统一起向 W3C 提交了一份 Web Service 寻址规范(WS-Addressing)。最初，IBM、微软、BEA 和 SAP 一起开发了 Web Service 寻址规范，而 Sun 并没有加入。希望将来能有更多的这类合作。

1.12 小结

面向服务的计算模式进一步扩展了“软件即是服务”的理念，复合业务流程也可视为服务，从而可以实时构建应用，并且任何开发者在任何地方都可复用这些服务。服务是自描述、平台中立的计算元素，可以支持快速的、低成本、很容易组合的松耦合的分布式应用。

Web Service 是各类不同的自动化的服务族，这些自动化服务使用因特网(作为通信媒介)和基于互联网的开放标准。一个 Web Service 就是一个能够通过网络(例如因特网)获得的服务，这个服务能够完成任务、解决问题，或者能够代表用户或应用程序处理事务。可以通过发现和调用网络上的服务，或者将这些服务组合成新的应用，来完成一些具体的任务，而无须完全从头开始构建新的应用。

Web Service 是一种基于服务请求者和服务提供者之间的松耦合关系的方式。这意味着服务请求者无须了解服务提供者实现的具体技术细节，诸如无须了解编程语言、部署平台等。服务请求者通常通过消息(请求消息和响应)来调用操作，而不是通过使用 API 或者文件格式来调用操作。Web Service 通过服务接口进行区分。服务接口是按中立方式进行定义的，独立于 Web Service 的实现。

Web Service 的关键概念是面向服务的体系结构。SOA 是一个设计软件系统的逻辑方式，通过发布和发现接口，既可以向最终用户应用提供服务，也可以向分布在网络上的其他服务提供服务。基本的 SOA 将软件代理之间的交互定义为服务请求者(客户端)和服务提供者之间的消息交换。客户端是一个请求服务执行的软件代理，而提供者是一个提供服务的软件代理。代理可以既是客户端服务又是提供者服务。

使用 Web Service 技术和精心设计的 SOA，企业可以复用已有的业务功能，并能加快各种新的应用的实现。然而，Web Service 目前也存在一些缺陷，具体包括：性能问题、缺乏高级事务管理机制、缺少对业务语义的合适支持，以及大量的已有标准和新诞生标准的部分重叠甚至相互冲突。由于目前尚存在这些不足，因而降低了企业采用 Web Service 技术的价值，并给企业带来了一定的风险。

复习题

- 什么是 Web Service?
- 阐述 Web Service 与应用服务提供商以及基于 Web 的应用的不同之处。
- 列举并简要描述 Web Service 的各种类别。

- 什么是有状态的服务？什么是无状态的服务？并给出具体样例进行说明。
- 什么是服务粒度？分别举出细粒度服务和粗粒度服务的典型样例。
- 阐述同步服务和异步服务的不同之处。
- 什么是松耦合的服务？对松耦合的服务和使用紧耦合的服务进行比较，并分别举出使用松耦合服务技术的例子和使用紧耦合服务技术的例子。
- 阐述面向服务的体系结构的重要意义。
- 列举并描述 SOA 中的角色和操作。
- 列举 SOA 的一些主要作用。
- SOA 中的主要层次有哪些？它们的作用分别是什么？
- 什么是 Web Service 的技术架构？
- 什么是服务质量？对于 Web Service 来说，为何服务质量非常重要？服务等级协议的作用是什么？
- 列举 Web Service 的一些优缺点。

练习

1.1 举出一些使用 Web Service 解决复杂的业务问题的典型例子。在传统的业务解决方案中，通常使用纸质文件、传真和邮件，请阐明基于 Web Service 的解决方案与传统的解决方案这两者之间的区别。

1.2 开发一个应用程序，要求：1)包含若干单个的 Web Service；2)并将这些应用组合成复合服务(业务流程)。指出并证明所设计的业务流程是无状态的还是有状态的。比较简单服务和复合服务的粒度级别。你自己的评价是什么？

1.3 假设现有一个生产定制的个人电脑和设备的消费类电子设备公司。该公司每六个月就会发布新产品。例如，一个顾客今天所买的电脑与该顾客六个月前买的电脑的款式会有所不同。通常，这些产品依然会使用已有的主板接口，但是重新配置了零部件，从而提供一些新功能或特殊功能。开发一个 SOA 解决方案，以便该公司的顾客可以很容易地不断对计算机的配置进行升级。请阐述设计方案，并描述在 SOA 解决方案中将要使用的 Web Service。

1.4 如图 1.6 所示，按照订购单、供应商、购买者、核准者、库存项、账户一览表以及未开据发票的收入，销售域中的采购流程将提供订单的详细信息。这样，采购部门将可分析各类订购单所进行的采购，包括总括订购单、计划订单及标准订购单，并且采购部门也可分析供应商的绩效、订货至交货的时间(基于订货时希望的时间以及交货的截至时间)。该流程也可帮助分析已收到的货物及已收到的发票，并可在账户中分析销售所涉及的发货费用。将这个标准的业务流程分解为若干合适的服务，并指出这些服务之间是如何进行相互交互的。

1.5 如图 1.6 所示，销售域中的销售库存流程跟踪现有库存量及即将到货的物品，并监控实际库存的变化趋势。该流程也通过不同的账户、仓库、子库存分析库存事务，并在任何时候都可通过仓库跟踪整个企业的现有库存。可根据到货日期、地理位置监控库存物品，并确保库存量满足实际需要。这样，就能很快发现周转速度较慢的库存及作废存货，从而使部门主管减少库存成本。将这个标准的业务流程分解为若干合适的服务，并指出这些服务之间是如何进行相互交互的。

1.6 阐述库存业务流程中哪些是同步服务，哪些是异步服务。

第二部分　核心基础架构

第2章　分布式计算的基础架构

学习目标

在过去的几十年里，分布式计算已经取得很大的发展，在进程间通信、远程方法调用、分布式命名、安全机制、数据复制、分布式事务机制等方面，都引入了新的技术并取得新的进展。分布式计算技术的新近发展XML技术的出现，最终导致了Web Service技术的诞生。Web Service技术目前已经占据了分布式计算技术的主导地位。

本章作为一个入门，讨论了与Web Service相关的若干关键技术。本章尤其还介绍了一些入门材料，这些材料对于理解Web Service基础架构中的一些先进理念非常重要。学习完本章之后，读者将能了解下列核心基础架构技术，这些技术将能推动基于Web Service的分布式应用的开发。

- 互联网协议。
- 同步通信和异步通信。
- 消息的发布/订阅机制及事件处理机制。
- 面向消息的中间件和集成代理。
- 事务监控。
- 企业应用和电子商务集成技术与体系结构。

2.1　分布式计算与互联网协议

分布式系统是网络化的计算机（它们虽然是异构的）集合，这些计算机之间能相互通信，并能通过传递消息来协调它们的运行。对于用户来说，这种分布是透明的，以至于系统看起来就像一台计算机。相反，对于联网在一起的多台计算机而言，用户会明显意识到存在多台计算机，并且也会意识到这些计算机各自的位置，存储复制和负载均衡，并且功能也不是透明的。

分布式系统包含大量的运行单元（或称计算单元，诸如服务器和其他处理器，或应用程序等）。这些运行单元分布在不同的、互联的计算机系统上。由于这些单元任何时候对它们自己都是完全控制的，所以这些单元是自治的。此外，没有集中控制，即没有一个单元会控制分布式系统中的其他单元。分布式系统通常使用客户/服务器模式。若一个计算机系统非常了分布式系统的一些单元，该计算机系统叫做主机（host）。分布式单元通常是异构的，它们可能是使用不同的编程语言编写的，并且可能运行在不同的操作系统和硬件平台上。构建分布式系统的主要目的是为了实现资源共享。因为分布式系统中的各个单元是自治的，所以分布式系统可以并发执行

应用程序。因此，在分布式系统中单元越多，则进程可以越多。而且，应用程序通常是多线程的。当应用程序对用户或其他应用程序开始进行服务时，即可创建一个新的线程。从而应用程序在执行服务时，就不会被堵塞，而是可以响应新的服务请求。

分布式系统的一个重要特征为：多个进程并不仅仅在一个处理器上执行，而是能够在多个处理器上执行。这就要求必须引入进程间的通信机制，从而能够管理在不同计算机上执行的进程间的相互交互。分布式系统可能会有许多不同类型的故障，这是分布式系统的另一个特征[Coulouris 2001]。例如，网络故障将会导致互连在一起的计算机无法进行相互间的通信，但这并不意味着这些计算机会停止运行。实际上，在这些计算机上运行的应用程序可能无法检测到网络是否发生了故障。类似地，对于计算机故障或者预料不到的应用程序崩溃，其他的单元也可能无法立即通过计算机间的通信检测到这些故障。在分布式系统中，每个单元各自都有可能发生故障，而这时其他的单元可能依然在运行。

在过去的几十年里，分布式计算在多个方面都取得了很大的进展。例如，在过去的20年里，先后引入了进程间通信与远程调用技术、分布式命名、安全机制、分布式文件系统、数据复制、分布式事务机制等，并且每次都伴随着新的体系结构模式和新协议的出现。

2.1.1 互联网协议

为了使数据能够跨互联网进行传输，如J2EE等典型的分布式平台都需要依赖互联网协议的支持。对于跨互联网的数据传输来说，互联网协议是至关重要的。对于分布式系统中的不同组件之间的通信，以及和远程组件间的通信，互联网协议通常都定义了相关标准。与其他传统的协议一样，互联网协议定义了两个或多个通信实体之间进行信息交换的格式和顺序，以及定义了消息或事件的传输与/或接受的行为[Kurose 2003]。

互联网协议中最重要的协议是传输控制协议和互联网协议(合称TCP/IP协议)。IP协议是互联网的基本协议，用于不同主机之间的报文的不可靠传送。IP协议并不保证报文是否一定送达、多长时间送达，也不保证在传送多个报文时报文的接收顺序与发送顺序是否一致。而TCP协议则增加了连接和可靠性的概念。对于跨网络的通信，尤其是跨互联网的通信，TCP/IP提供了数据流的可靠传送。

为了标识互联网络中的计算机(主机)，需要给每一台主机赋予一个地址，该地址称为互联网协议地址(简称IP地址)。IP地址唯一标识了每一个网络及互联网中的每一台主机。IP地址包含两部分：网络标识符和主机标识符。IP地址中的网络标识符标识互联网中的网络，这个标识符是由互联网权威管理机构分配的，并且在整个互联网上是唯一的。IP地址中的主机标识符是由控制该网络的组织所分配的。事实上，IP地址可分为三类，分别用于大型网络、中型网络和小型网络。大型网络的数量较少，中型网络的数量中等，小型网络的数量很多。

在进一步详细分析TCP/IP协议之前，我们首先概述一下国际标准化组织的开放系统互连模型(简称OSI模型)中的各层。了解OSI中的层次对于理解TCP/IP模型中的分层非常关键。

1. OSI参考模型

不同的系统可以分布在网络上，在这些系统上运行的应用程序进程之间可以进行数字通信，OSI参考模型是这类数字通信的一种抽象描述。参考模型也是一个关于如何进行通信的概念蓝图。参考模型涉及了高效通信所需的所有处理步骤，并把这些处理步骤按逻辑关系进行分组，并将这些组分别称为层。当通信系统按这种方式进行设计时，则称为分层的体系结构。

如图2.1所示，OSI模型是一个七层层次结构。每一层向直接邻近的上一层提供所请求的增值服务。反过来，每一层又向与它直接相邻的下一层提出一些基本的服务请求。OSI的层次划分如下所示。

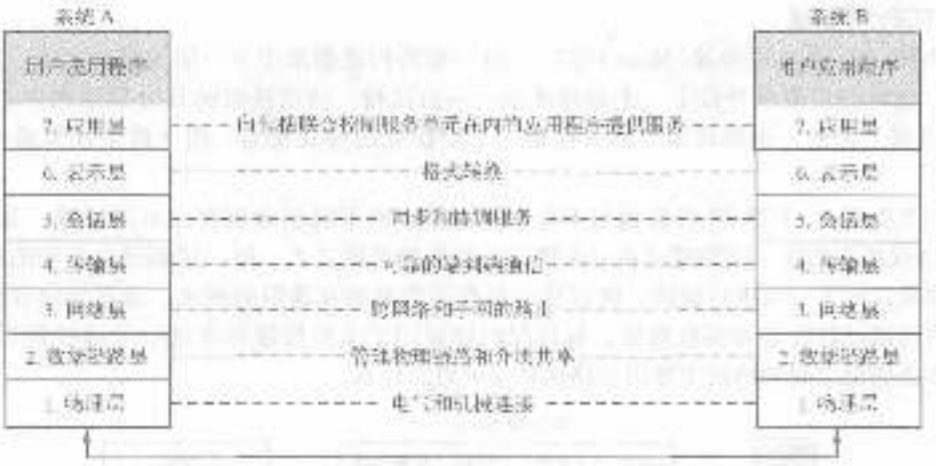


图 2.1　ISO/OSI 参考模型

物理层(或称第一层)：这是 ISO/OSI 参考模型的七层中最低的一层，它规定了激活、维持和释放终端系统之间的物理链路所需的电气需求、机械需求、过程需求及功能需求。

数据链路层(或称第二层)：该层提供了在网络实体之间传送数据的方法，以及检测甚至有可能修复物理层所出现的错误。数据链路层提供了数据及错误通知的物理传输、网络拓扑和流量控制。数据链路层将消息划分成格式化的数据块，每一个数据块称为一个数据帧。每一个数据帧的数据由两部分组成：帧头和帧数据，帧头包括了数据帧的发送主机的源地址以及接收主机的目标地址。

网络层(第三层)：该层响应传输层(传输层是与网络层直接相邻的上一层)的服务请求，并将服务请求发送给数据链路层。网络层可通过一个或多个网络将源结点的变长数据包传送到目的结点。该层所提供的功能包括：网络路由、流量控制、组包和拆包、差错控制。

传输层(第四层)：该层响应会话层(会话层是与传输层直接相邻的上一层)的服务请求，并将服务请求发送给网络层。传输层的作用是提供终端用户之间的透明的数据传输。传输层负责络上层数据分段并提供可靠或不可靠的传输(不可靠传输的传输效率更高)。用户可以请求在不同的系统间进行可靠的数据传输。传输层通过流量控制和可靠的数据传输来保证数据完整性。

会话层(第五层)：会话层提供了终端用户应用程序进程之间的会话管理机制，即负责建立、管理、终止表示层实体之间的会话。该层协调系统之间的通信，并提供三种不同的模式来组织通信：单工、半双工和全双工。会话层可在数据中插入校验点，还可终止或重启会话过程。

表示层(第六层)：表示层为应用层表示数据，并提供数据转换和代码格式化。表示层的主要功能包括数据压缩与恢复、数据加密与解密等。

应用层(第七层)：应用层是 ISO/OSI 参考模型中的最高层。应用层为应用程序提供了访问 OSI 网络的接口，并完成应用程序所需的公共应用服务。对于进行通信的四个应用程序进程，公共应用服务提供了语法转换。

第一层到第四层处理跨网络的端到端数据传输所需的通信、流量控制、差错处理。传输层下面可以有许多不同类型的物理网络，例如 X.25 分组交换数据网络(PSDN)或局域网(LAN)。第五层到第七层处理跨网络应用的协调，以及处理如何向应用表示信息。对于提供不同类型的终端用户服务的许多应用程序来说，通用的传输层实现可以支持这些应用程序。

2. TCP/IP 协议

TCP/IP 是一个分层协议[Moss 1997]。每一层的构建都基于下一层的基础之上，并提供新的功能。这种分层表示导致了一个新技术语——协议栈。协议栈指的是分层结构中定义 TCP/IP 如何协同工作的一组协议集。协议栈是一个层次化的协议集合，用于两个计算系统间的数据传输。

如图 2.2 所示，TCP/IP 协议栈与 ISO/OSI 模型中的不同层分别存在对应关系。如图所示，TCP/IP 协议栈中的每一层都建立在它所提供的服务的基础之上。每一层都通过商明的接口与它的下一层及它的下一层进行通信。协议栈的最底层处理和互香信的需求。最底层协议仅涉及使用具体的网络硬件发送和接收数据。最顶层的协议用于文件传输或发送电子邮件等具体任务。最顶层和最底层之间的协议主要用于路由和保证可靠性等。



图 2.2 TCP/IP 协议栈以及它与 ISO 参考模型之间的关系

注意，并不存在一个标准的 TCP/IP 模型。在有些文献中，协议栈的底部还附加了一个物理层。图 2.2 所示的 TCP/IP 模型是一个四层协议[Moss 1997]、[Rodriguez 2001]，其中物理连接标准被视为数据链路层的一部分。TCP/IP 协议栈包含下列层次。

数据链路层：数据链路层也可简称为链路层，它是最底层的协议。数据链路层提供了到实际的网络硬件的接口。该接口既可以提供可靠传输，也可以不提供可靠传输，并且可以是面向分组或面向数据流。事实上，TCP/IP 在该层并不指定任何协议，而是几乎可以使用任何一种可用的网络接口，这也表明了网际协议层的灵活性。

互连网络层：互连网络层也称为互联网层或网络层，它是数据链路层的上一层，并紧邻着数据链路层。该层负责数据报的路由。数据报这一术语的基本含义就是指从一台主机发送到另一台主机的"数据块"。互联网络层提供了互联网的"虚拟网络"映像（对于高层来说，互联网络层掩蔽了该层下面的物理网络结构）。IP 协议是互联网络层中最重要的协议。IP 协议是 TCP/IP 协议的基础。通过 TCP/IP 网络发送的每一条消息或每一块数据，都是作为 IP 包发送的。IP 提供了路由功能，可将所传送的消息发送到它们的目的地。

IP 使用链路层协议在不同的网络之间传输数据。IP 是一个无连接的协议，并且并不保证其下面的层次是可靠的。无连接协议意味着该协议没有作业或会话，并且对最终的传输结果不做任何保证。每个数据包都被视为是一个独立的实体。IP 对数据包进行简单的路由，每次仅决定数据包的下一站目的地，并且既不关心该数据包是否能够发送到最终目的地，也不关心数据包的

抵达顺序是否与发送顺序一致。数据包中并不含有标识包间顺序的任何信息，也不包含包所属作业的任何标识信息。IP 不提供可靠性、流量控制、错误校验，这些功能需要在更高层才能提供。

传输层在应用的客户端和服务端之间传输数据，提供了端到端的数据传输，并可以同时支持多个应用程序。使用得最多的传输层协议是 TCP 协议。TCP 协议提供了面向连接的可靠的数据传输、重复数据抑制、拥塞控制及流量控制。

TCP 是许多互联网应用所使用的传输层协议，如远程登录(Telnet)、文件传输协议(FTP)、超文本传送协议(HTTP)。TCP 是一个面向连接的协议。面向连接的可靠服务保证了发送者发送数据的顺序与接受者收到数据的顺序是一致的，并且整体上也是完全一样的。这意味着，客户端和服务器这两台计算机在发送数据之前必须首先建立连接。一旦建立了连接，即可正式发送数据。TCP 是一个滑动窗口协议，在发送下一个数据段之前并不需要等待前面一个数据段传输成功的确认，仅当立即需要时，或者在一定的时间间隔后，才需要进行接收确认。这提高了 TCP 对于成批数据的传输效率。TCP 可以保障传输的可靠性。使用 TCP 的应用程序可以确信另一端接收到它所发送的数据，并且能够确信另一端所接收的数据是正确的。

应用层：应用层负责支持网络应用。使用 TCP/IP 进行通信的程序提供了应用层。应用程序是一个与另一个进程进行通信的用户进程。进行通信的这两个进程通常位于不同的计算机上，但也有可能位于同一台计算机中。这些应用程序包括 Telnet、FTP、SMTP(简单邮件传输协议)等。应用程序和传输层之间的接口是通过端口号和套接字(Socket)来定义的。

TCP/IP 程序通常用于互联网，并且这些程序主要是面向客户/服务器应用的。服务器程序在收到连接请求后，与发起请求的客户计算机进行通信。为了便于通信，每一个应用程序(如 FTP、Telnet 等)都被赋予了一个唯一的标识，这个标识称为端口。每一个应用程序与特定的端口进行了绑定，当对该端口发起了连接请求后，将激活与该端口相对应的应用程序。表 2.1 列出了一些最常见的端口，以及通常绑定到这些端口上的应用程序。

表 2.1　常用端口及与这些端口相对应的应用

应　用	端　口
文件传输协议	21
远程登录	23
简单邮件传输协议	25
超文本传输协议	80
网络新闻传输协议	119

2.1.2　中间件

中间件是一种连接软件，在不同系统间建立桥接，以便在这些系统之间能够相互通信和传输数据，从而有助于管理分布式系统所固有的复杂性和异构性。中间件可以视为一个软件服务层。通过中间件，无论底层所采用的通信协议、系统体系结构、操作系统、数据库及其他的应用服务是否相同，应用单元都可以跨网络进行互操作。中间件提供了一个简单的、一致的、集成的分布式编程环境，从而简化了分布式应用的设计、编程和管理。中间件实质上就是一个分布式软件层，也可视为一个位于操作系统之上的“平台”。对于底层采用多种网络技术，具有不同的计算机体系结构、操作系统和编程语言的分布式环境，中间件这一平台可以屏蔽分布式环境的复杂性和异构性。

与操作系统和网络服务相比，中间件服务提供了更多的应用编程接口(API)，从而使得应用：

- 可以透明地跨网络查找应用程序，从而可以和其他的应用程序或服务进行交互。
- 可以向软件开发者屏蔽底层的、繁琐的、易于出错的平台细节，诸如 Socket 层的网络编程。

- 可以提供面向网络的高层抽象。由于这些高层抽象更接近应用需求，从而可以简化分布式系统的开发。
- 可以利用先前的应用开发，并且复用它们，而无须针对每一个具体的应用重新构建。
- 可以提供大量的服务，诸如可靠性、可用性、认证、安全性。在分布式环境中，应用程序若要高效运行，离不开这些服务。
- 可以在不影响功能的情况下实现可伸缩性。

目前的中间件产品可以屏蔽网络、硬件、操作系统和编程语言的异构性。分布式应用中的不同单元可以根据实际需要采用任何合适的编程语言，从而允许应用级也可以具有异构性。最后，中间件平台所提供的编程支持，可以在许多方面提供有关分布的透明性，如位置、并发、复制和故障。这意味着，无论应用程序是否位于不同的地点，是否并发执行，是否在多个地方进行数据复制，应用程序都能够进行互操作。

如图 2.3 所示，中间件层位于应用程序与互联网传输协议之间。如图所示，中间件抽象包含两层。底层涉及分布式系统中的进程间的通信协议。对于在通信网络上发送消息的应用程序，考虑到不同的计算机对于简单数据项的表示可能有所不同，应用程序可能需要将所发送的消息转换为合适的形式。因此底层还涉及如何对这些应用程序所使用的数据对象（如销售订单）和数据结构进行转换。上面的一层主要关于进程间通信的机制，该层涉及基于消息的中间件和不基于消息的中间件。不基于消息的中间件提供了同步通信机制，可用于支持客户/服务器通信。基于消息的中间件提供了异步通信和事件通知机制，可通过通信网络交换消息或对事件进行响应。

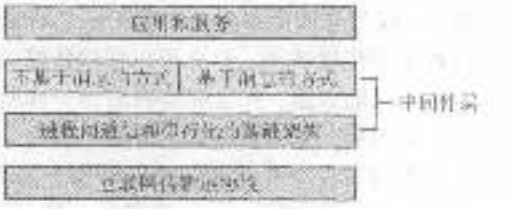


图 2.3 中间件的层次

由于互联网应用的最盛行的结构是客户/服务器体系结构，因此在进一步深入分析如图 2.3 所示的中间件的层次之前，我们将首先讨论客户/服务器体系结构的特性，这也是下一节内容的要点。

2.2 客户－服务器模型

客户/服务器计算在分布式处理中得到广泛应用，数据控制的集中化和数据访问的分布性这两者之间有冲突，为解决这一难题，客户/服务器模式已经成为一个最通用的解决方案之一。简言之，图 2.4 所示的客户/服务器体系结构是一种计算架构。这一架构将计算和存储任务划分成两类——客户端和服务器端。在客户/服务器架构中，客户端进程（服务消费者）请求服务器进程（服务提供者）的服务。服务器也可以是其他服务器的客户端。例如，Web 服务器的页面可以存储在文件服务器上，由文件服务器管理这些文件（存储结构），则 Web 服务器就成了本地文件服务器（或数据库服务器）的客户端。一般而言，客户/服务器计算并不强调硬件的差别，而是强调它们各自的实际应用的差异。同一个设备既可以充当客户端，也可以充当服务器。例如，具有很大的内存和磁盘空间的 Web 服务器就是一个服务器，又是一个客户端，可以运行本地浏览器会话。

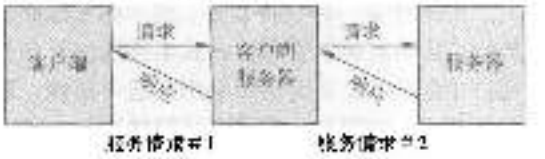


图 2.4 客户－服务器体系结构

引自：J. Wiley & Sons 出版社 2005 年出版的、由 M. P. Papazoglou, P. M. A. Ribbers 所著的"e-Business: Organizational and Technical Foundations"一书（允许复制）

在客户/服务器体系结构中，客户端运行本地存储的软件和应用程序。有些应用程序可以存储在服务器上，并在服务器上运行，但绝大多数程序都是在客户端上运行。服务器提供应用所需的数据。在客户/服务器体系结构中，客户端实际上有两大任务。客户端向服务器进行请求，同时也是用户接口。例如，Web 浏览器软件是一个客户端进程，当用户通过输入 URL 地址或点击链接请求文档时，Web 浏览器不仅向 Web 服务器请求文档，也必须向用户显示文档。Web 服务器必须存储所有的文档，并且必须响应来自客户端的请求。

对于互联网应用来说，客户/服务器模型是最盛行的结构。万维网、电子邮件、文件传送、远程登录、新闻组和其他许多流行的互联网应用程序，都采用了客户/服务器模型。由于客户端程序通常在一台计算机上运行，而服务器程序通常在另一台计算机上运行，因此根据定义，客户/服务器模式的互联网应用都是分布式应用。通过在互联网上相互发送消息，客户端程序和服务器端程序进行相互交互。

为了将基于 Web 的应用中的客户端与传统的客户/服务器体系结构中的客户端进行区分，引入了一个新的术语“瘦客户端”。在传统的客户/服务器体系结构中，应用程序的大部分都存储在客户端。在基于 Web 的应用中，并不需要将应用程序的代码下载到客户端计算机运行，而是在功能强大的服务器上运行程序，因此这类客户端称为“瘦客户端”。在基于 Web 的应用中，服务器提供了执行代码及客户端所需的数据。浏览器是一个用户接口。因此，客户端所需的软件仅是操作系统、浏览器及用于转换所下载的代码的软件。

2.3　进程间通信的特性

通过跨计算机网络交换消息，在两个不同的终端系统(很有可能采用了不同的操作系统)上的进程相互之间可以通信。如图 2.3 所示，用于互联网传输层协议(诸如 UDP 协议)的中间件 API 提供了消息传递的抽象这一最简单形式的进程间通信。这使得发送进程可将单个的消息发送给接收进程。在 Java API 中，发送者使用 socket 指定消息的目的地，这是对于目的地计算机的目标进程所使用的特定端口的间接引用。

2.3.1　消息发送

分布式系统和应用程序通过交换信息的方式进行通信。消息发送是一种能够可靠发送的通信技术，可支持高速、异步、程序到程序间的通信。程序之间通过发送数据包进行相互间的通信，所发送的数据包称做消息。消息是一个定义明确的、数据驱动的文本格式。消息能够在两个或多个应用程序之间进行发送，所发送的消息可包含业务消息和网络路由标头。消息通常包含三个部分：头部、属性和有效载荷(消息体)。消息传送系统和应用开发者使用消息头提供特征信息，如消息目的地、消息类型、消息的失效时间等。消息的属性部分包含应用程序定义的若干名/值对。本质上，这些属性是消息体的一部分，之所以将它们单独作为消息的一个专门组成部分，其目的是便于客户端或专用的路由器能够过滤这些信息[Chappell 2004]。消息体是消息所携带的实际的有效载荷。对于消息传送的不同实现方式，消息有效载荷的格式可能有所不同。最常见的格式是纯文本、可支持任何类型的数据的原始字节流，或者专门的 XML 消息类型。若采用专门的 XML 消息格式，则可使用任何常见的 XML 解析技术来访问消息的有效载荷。

消息可简单地解释为数据、在消息接收者计算机上激活的命令，或者作为发送者计算机上所发生的事件的描述。业务数据通常包含商业交易方面的信息，诸如销售订单、支付处理、货物运送和跟踪。最简单的消息发送是请求/响应消息。发送者发送消息，然后可以获取消息接收者的响应(假如有的话)。

为了支持进程间的消息传递，需要采用两个消息通信操作：send 和 receive，它们是根据目的地和消息进行定义的[Coulouris 2001]。一个进程为了能够和另一个进程进行通信，其中一个进程将消息发送到目的地，另一个进程在目的地接收该消息。这一行为涉及从发送进程到接收进程间的数据通信，并且可能会涉及两个进程间的同步。接收进程接收消息，并且可能会送回一个响应消息。在分布式应用中，应用层协议定义了进程间交换消息的格式和顺序，并定义了消息传送或消息接收所需进行的操作。

在分布式系统中，存储在应用程序中的数据通常表示为数据结构，例如在 Java 中相互链接的一组对象。然而，在相互交互的进程间所交换的消息由字节序列组成。无论使用何种形式的通信，在传送以及送达后重建之前，数据结构都必须是扁平的(能够转换为字节序列)。为了使得两台或多台异构的计算机之间能够交换数据值，需要采用一种特定的数据编组技术(marshalling)。

数据编组是一种数据处理方式，将对象或其他形式的结构化数据项打散，使得这些数据能够以字节流的方式在通信网络上传送，并且在接收端能够比较容易地重建这些对象或数据结构。当这些编组的数据传送到目标端后，采用数据编出(unmarshalling)可将这些字节流重新恢复为与原先一样的对象或数据结构。因此，数据编组将结构化数据项和原始数值转换为约定的标准形式，然而跨网络进行传送。类似地，在接收端，数据编出根据这些数据的形式表示重新生成原始值，以及重新构建相同的数据结构。在 Java 和 XML 中，使用术语“串行化”(serialization)和“反串行化”(deserialization)分别表示“数据编组”和“数据编出”。在本书中，我们将这两组术语视为可相互替换，并将不加区别地进行使用。

2.3.2 消息目的地和 socket

在 2.1.1 节中，我们已经讨论了互联网协议、端口。进行进程间通信时，进程可能会使用多个端口接收消息。服务器通常会公开它们的端口号，以便客户端能够使用。知道端口号的任何进程都能向该端口发送消息。

在进行进程间通信时，消息能够发送到某一具体的(IP 地址，本地端口号)。本地端口是接收端计算机中的消息目的地，通过一个标识符(端口号)来进行指定。当客户端对于服务使用固定地址时，这种方式就有一个严重的缺陷：为了保持地址有效，服务必须一直在同一台计算机上运行。为了避免这一问题，可以有两种方法。方法之一是客户端通过名字来引用服务，名字服务器可将具体的名称转换为服务器当时的位置。对于允许服务重定位的消息，另一种方式是使用操作系统提供的位置独立型标识符[Coulouris 2001]。

许多应用程序涉及位于不同主机上的两个进程。这两个进程通过网络进行相互通信，相互之间交换(发送和接收)消息。进程通过它的套接字(socket)向网络上发送消息，以及从网络上接收消息。可将进程的 socket 视为进程的入口点。如图 2.5 所示，进程间的通信即是在进程的 socket 之间传送消息。一旦消息抵达目的地，即被传送到接收进程的 socket，接收进程然后就处理该消息。对于接收消息的特定进程，如图 2.5 所示，该进程的 socket 必须绑定到一个本地端口，并且必须指明进程所在的计算机的 IP 地址。

2.3.3 同步方式的消息发送和异步方式的消息发送

通信有两类不同的基本方式：具有依时性的同步方式以及与时间无关的异步方式。消息传送中间件有许多不同类型，它们分别都能够支持一类基本方式的消息通信，有时可以支持两类方式。

根据定义，同步执行的特征为：在两个通信应用系统之间必须要进行同步，两个系统必须都

在正常运行，并且会中断客户端的执行流，转而执行调用。发送程序和接收程序都必须一直做好相互通信的准备。发送程序首先向接收程序发起一个请求（发送消息）。发送程序紧接着就会挂起它自身的过程，直到收到接收程序的响应。发送程序在收到响应后会继续向下进行处理。同步方式的请求/响应通信如图 2.6 所示。在 2.4 节中，将讨论同步通信的一个具体实例——远程过程调用。

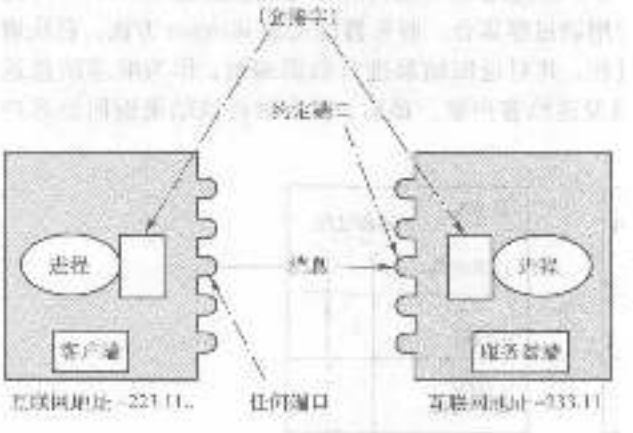


图 2.5　端口和 Socket

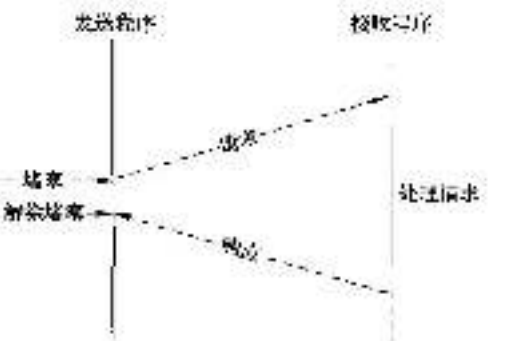


图 2.6　同步消息传送

当使用异步消息传送时，调用者在发送消息以后可以不用等待响应，可以接着处理其他任务。对于异步通信，一个应用程序（请求者或发送者）将请求发送给另一个应用程序，然后可以继续向下执行它自身的其他任务。发送程序无须等待接收程序的执行和返回结果，而是可以继续处理其他请求。与同步方式不同，异步方式中两个应用系统（发送程序和接收程序）无须同时都在运行，也无须同时都在处理通信任务。

通常使用排队机制来实现异步消息传送。有两种不同类型的排队方式：存储/转发、发布/订阅，在 2.5 节中将讨论这两类排队方式。

在选择通信方式的类型时，必须仔细斟酌松耦合接口和紧耦合接口的利弊，以及异步交互和同步交互的利弊。在下面的两节中，我们将更详细地讨论这些通信方式。

2.4　中间件的同步方式

同步方式的中间件的编程模型由一些分布式的协作程序组成，这些协作程序相互进行交互。这些程序需要能够同步地激活位于不同的计算机系统上的其他进程的操作。

对于不基于消息方式的中间件，最常见的方式是远程过程调用（RPC）和远程方法调用（RMI）。在本节中，我们将分析这两类方式。

2.4.1　远程过程调用

RPC 是程序间通信的一种基本方式。实际上，RPC 是一种中间件机制，用于调用远程系统中的过程。远程系统中的过程返回相应的结果。对于这类中间件，应用单元相互之间运行同步通信，即它们应用请求/等待响应这一通信模型。"常规的"非分布式应用通常使用串行化线程的执行，即依次执行每一条语句。RPC 的编程风格放宽模拟了串行化线程的执行。由于 RPC 将网络通信和应用程序代码进行了分离，因此 RPC 是实现客户/服务器应用的一种最简单的方式。

在 PRC 中，应用代码和 RPC 机制的关系如图 2.7 所示。在 RPC 程序中，对象和它的方法是"远程的"，以致需要跨网络进行方法调用。在客户端应用代码中，RPC 看起来很像一个本地过程调用，RPC 调用的实际上是一个通常称为客户桩(client stub)的本地代理。客户桩是支持 RPC 的代理程序，它模拟了远程对象和方法的接口。对于客户端来说，客户桩本质上类似于一个本地过程，代替了调用的执行。客户桩对过程标识符和要传进行数据编组，将它们转换为请求信息，并通过通信模块将请求信息发送给服务器。客户桩使用 RPC 运行时库与服务器桩(server stub)进行通信，RPC 运行时库是支持所有的 RPC 应用的过程集合。服务器桩类似 skeleton 方法，它从请求消息中抽取变量信息，调用相应的服务过程，并对返回结果进行数据编组，作为响应消息返回。服务器桩使用 RPC 运行时库将输出结果发送给客户桩。最后，客户桩将该结果返回给客户端应用代码。

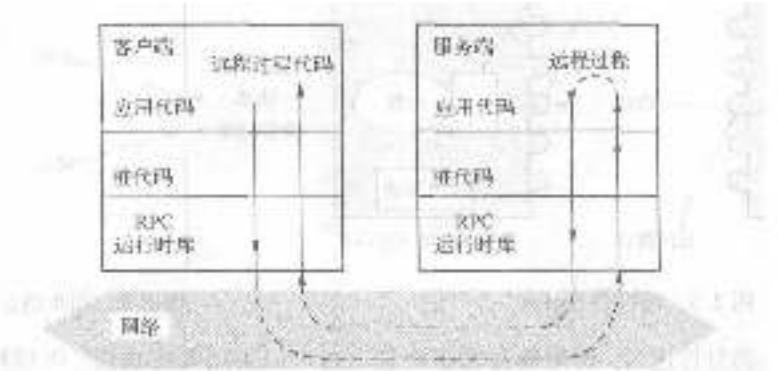


图 2.7 RPC 通信

引自：J. Wiley & Sons 出版社 2006 年出版的、由 M. P. Papazoglou、P. M. A. Ribbers 所著的《e-Business: Organizational and Technical Foundations》一书(允许复制)

在小型的简单应用中，通信主要是点对点方式，而不是一个系统对多个系统的方式。对于这类小型的简单应用，RPC 能够取得很好的效果。然而，对于大型的关键任务类型的应用，RPC 不能提供很好的递增适应性。RPC 将许多重要的细节都交由编程人员处理，如网络和系统的故障处理，多连接的处理以及进程间的同步，当跨多个进程进行同步时，当多个 RPC 调用同属一个同步的请求/响应循环时，一个 RPC 调用的成功依赖于后面的 RPC 调用的成功与否。这就使得调用一旦不能全部成功，就会全部失败。假如一个操作由于某种原因不能完成，其他的所有相关操作都会失败，如图 2.8 所示。

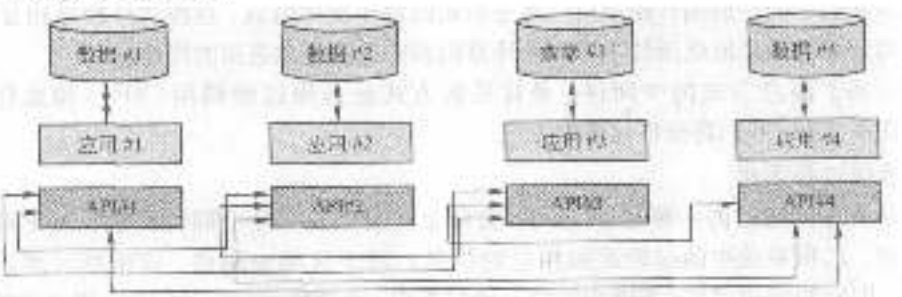


图 2.8 紧耦合的点到点集成

图 2.8 显示了 RPC 所导致的紧耦合的接口和应用。在 RPC 环境中，每一个应用需要知道其他所有应用的详细的接口信息，需要暴露方法数及方法的签名细节。该图清晰地说明了 RPC 的

同步特性，将客户端和服务器紧紧耦合在一起。客户端被堵塞了，不能向下执行，直到服务器端返回结果。假如服务器端不能正常工作或者不能完成所请求的任务，则客户端也不会正常工作。

在过去的若干年中，RPC 在业界得到一定程度的应用。使用 RPC 进行通信的主要技术包括通用对象请求代理体系结构（CORBA）、Java 远程方法调用（RMI）、DCOM、Active X、Sun-RPC、Java for XML-RPC（JAX-RPC）、简单对象访问协议（SOAP）v1.0 和 v1.1。基于组件的体系结构（诸如 EJB）也构建在这一模型之上。然而，由于 RPC 的同步特性，对于构建需要高性能、高可靠性的企业级的应用来说，RPC 并不是一个好的选择。对于集成在一起的应用，RPC 的同步、紧耦合特性严重地阻碍了系统间的处理。如图 2.11 所示，在采用同步通信的解决方案中，应用之间通过 API 相连，基于点到点的方式集成在一起。假如在所有的应用中都采用这一方式，就意味着在应用之间存在许多的集成点。

2.4.2　远程方法调用

传统的 RPC 系统是语言中立的，因此若有些功能不能在所有目标平台上都可用，则无法提供这些功能。对于 Java 对象的分布式计算，Java 远程方法调用基于 RPC 机制提供了一个简单、直接的模型。

Java 远程方法调用实现了对象间的通信。假如一个特定的方法位于远程计算机上，对于编程人员来说，通过 RMI，这些远程方法就好像位于本地计算机上。因此，对于用户来说，远程方法调用是透明的。RMI 调用由两个不同的程序组成：服务器端和客户端。远程方法调用提供了服务器端和客户端之间的通信机制，并在服务器端和客户端之间来回传递信息。

在远程方法调用中可以使用两种不同的类：远程类和串行化类。远程对象是远程类的实例。当在同一个地址空间中使用远程类时，可以将远程对象看成普通的对象。但是假如在地址空间的外部使用远程对象，则必须通过对象句柄来引用对象。相应地，串行化对象是串行化类的实例。可以将串行化对象从一个地址空间拷贝到另一个地址空间，这意味着串行化对象可以是一个参数或返回值。注意，假如返回了远程对象，则它返回的是对象句柄。

2.5　中间件的异步方式

正如我们在 2.3.3 节中所讨论的，当需要多个应用程序和 Web Service 进行相互交互时，不要寄希望于每个应用都了解其他所用应用中的方法的签名特性。相反，应用无须了解与其进行交互的其他应用的错综复杂的服务接口。异步通信对于松耦合技术的应用是一个推动。在松耦合的环境中，一个应用既无须了解如何才能连接到其他应用的详细信息，也无须了解其他应用的接口的详细信息。如图 2.9 所示，在一个多步骤的业务流程中，每一个参与者仅要关心的仅是确保能够向消息传送系统发送消息。

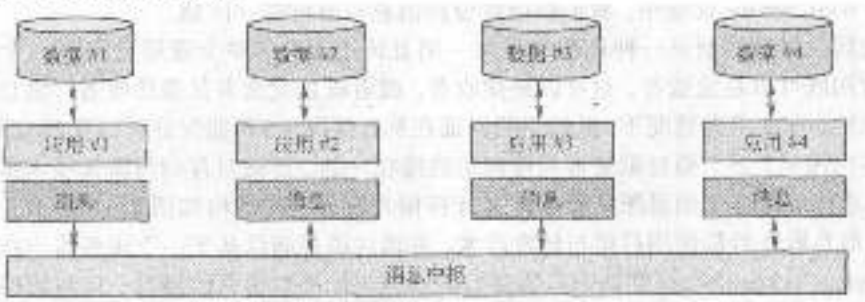


图 2.9　异步应用的松耦合方式

异步通信的两个具体实例就是消息的存储与转发以及发布/订阅。发布/订阅方法应用了事件通知的原理，这一方法在分布网服务中得到广泛的应用，我们将在本节中介绍这一主题。最后，我们将分析另一个很有意思的消息传送方法：点到点的排队。

2.5.1 消息的存储与转发

通过存储与转发排队机制，发送程序可将消息发送到一个称为消息队列的虚拟信道中，接收程序可根据需要从消息队列中接收消息。消息队列既是发送者发送消息的目的地，也是接收者接收消息的消息源，发送者和接收者通过队列交换消息。消息队列是一个驻留消息的容器，直到接收者从队列中接收到消息。消息队列独立于发送程序和接收程序，在进行通信的应用之间充当缓冲区。如图 2.10 所示，在这种类型的通信中，发送者应用和接收者应用分别与消息队列相关联。

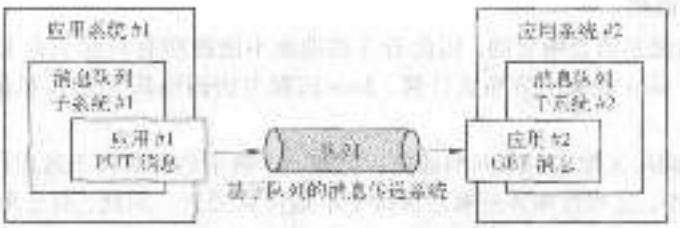


图 2.10 消息的存储与转发

该图从应用角度显示了一个消息传送的常见场景。一个应用将消息传送到消息队列中，另一个应用（图 2.10 中的应用系统#2）通过消息队列查看信息。该图意味着任何一个应用都不了解队列的物理位置。类似地，也不了解任一主机平台的物理细节。应用唯一所需做的就是以某种方式注册到或连接到消息队列子系统。这是一种非常有用的抽象化，从而使得任何一个平台上的物理实现的变化都不会影响到其他平台上的实现。基于存储与转发排队机制的异步通信，使得应用在任何时候都可以仅根据自身的情况发送或接收消息。

消息传送可有几个不同的选择：准确的一次性传送（exactly-once delivery）、至少一次传送（at-least-once delivery）、至多一次传送（at-most-once delivery）。对于许多应用来说，确保能够将消息可靠地传送到最终目的地并且没有重复发送是至关重要的。这种等级的消息发送称作"准确的一次性传送"。"至少一次传送"保证能够将消息至少一次发送到它们的目的地。"至多一次传送"保证能够将消息至多一次传送到它们的目的地。最后一种方式意味着消息传送系统可以因为硬件、软件、网络故障等问题偶尔丢失一些消息，因此这一方式对于 QoS 的要求比较低。准确的一次性传送保证了消息的可靠传输。在存储与转发排队机制中，这一问题将留到后一小段落，在 8.4.2 节的 Web Service 标准中，我们将继续分析消息可靠性这一问题。

存储与转发排队机制是一种典型的多对一消息传送方式，多个应用可向同一个应用发送消息。一个应用既可以是发送者，也可以是接收者，或者既是发送者又是接收者。消息排队提供了更高的可靠性，在大多数情况下（虽然不能保证在所有情况下）都能保证可以完成应用操作。

在许多应用中，多个消息服务器通过网络链接在一起，这就对存储与转发技术提出了进一步的要求，要求能够跨多个消息服务器重复进行存储与转发，其结构如图 2.11 所示。在这种结构中，每一个消息服务器都使用存储与转发技术，并通过消息确认从下一个服务器中获取信息。消息确认（message acknowledge）使得消息传送系统能够监控消息传送的情况，从而能够知道何时成功完成消息的发送。通过该信息，面向消息的中间件系统能够管理消息的分发，保证将消息发送到它们的目的地。

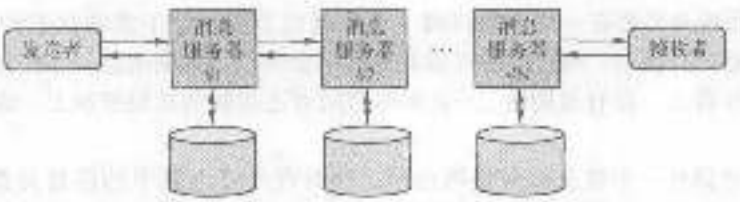


图 2.11　涉及多个消息服务器的存储与转发

在图 2.11 中，消息发送者指定了服务需求，服务器之间的传送过程保证了服务需求的可靠性和质量。这一方式对于服务器和接收者都是透明的。

2.5.2　消息的发布与订阅

消息的发布/订阅是另一种可靠地传送消息的方式。与存储与转发方式相比，消息的发布/订阅方式的可伸缩性稍微大一点。在这类异步通信中，生成消息的应用发布该消息，需要这类消息的其他应用则订阅该消息。发布应用将包含新信息的消息放置在针对每一个订阅者的队列中。系统中的每一个应用都可以有两个角色，可以充当不同类型信息的发布者和订阅者。

发布/订阅消息传送的流程如下。假设发布者应用发布了一个具体主题的消息，诸如向零售商发送新产品的价格或新产品的描述。多个订阅应用都可以订阅该主题，并接收发布应用所发布的消息。如图 2.12 所示，消息发布者发布消息，将消息发布到主题，对于已经在该主题上注册的订阅者来说，一旦发布者将消息发布到主题，这些订阅者即可接收这些消息。图 2.12 描述了发布/订阅的工作流程。

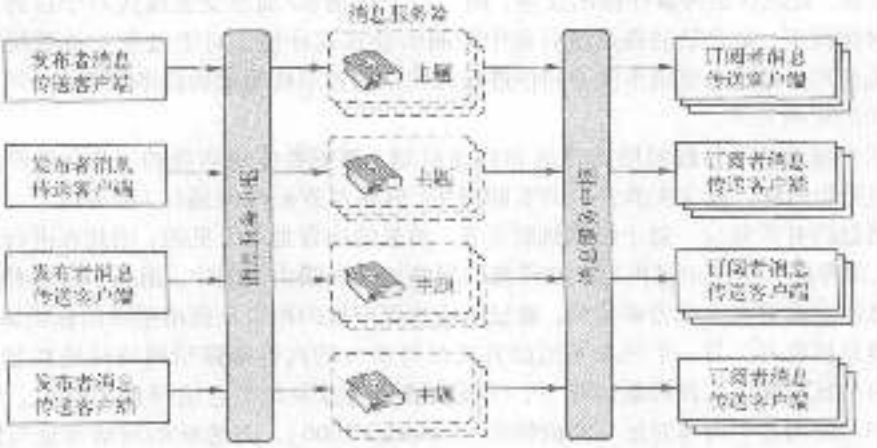


图 2.12　消息的发布与订阅

(1) 发布者将消息发布到特定主题。

(2) 消息服务器记录所有的消息、当前处于活动状态的长期订阅者(即对于特定主题明确表示了长期兴趣的订阅者)。通过授权和认证，消息服务器为消息传送系统提供了一个安全的环境。

(3)　一旦将消息发布到某一具体的主题，这些消息就将分发到它们的订阅者。对于在消息发送时暂时没有连接的长期订阅者，只要在规定的时间间隔内，他们依然能够收到这些消息。

消息服务器负责向订阅了主题的订阅应用发送被发布的消息。每一个消息都有一个有效期，有效期规定了消息在主题中的最大存活时间。消息服务器首先将消息发送到相关的处于活动状

态的订阅者，进而检查是否有一些虽然订阅了该主题但当前不处于活动状态的长期订阅者。假如有这样的不活动的订阅者，则这样的订阅者显然不会确认已收到消息，在有效期内这些消息将会保留在消息服务器上。在有效期内，一旦那些订阅者连接到消息服务器上，他们仍然能够接收这些消息。

所有的订阅者都有一个消息事件侦听程序。侦听程序将主题中的信息发送给客户端程序，以供进一步处理。订阅者也可以使用消息选择器对所收到的消息进行过滤。消息选择器可以使用过滤表达式对消息的标题和属性(不是消息体)进行评判。

可以很容易地实时修改订阅列表，这使得在不同的系统和网络上运行的通信系统具有更大的灵活性。发布/订阅消息传送方式通常能够转换消息、充当解释程序，从而使得原先不相关的应用能够协同工作。

总之，异步通信提供了一个松耦合的环境，即一个应用并不需要了解如何连接其他应用的详细信息，也不需要了解其他应用的接口的详细信息。一般而言，对于企业应用集成和跨企业的计算，尤其当需要在企业内部信息系统(如数据库和 ERP 软件包)之间传输数据时，或者当企业内部信息系统和合作伙伴的系统之间需要传输数据时，异步通信将是首选的解决方案。在这些情况下，来自于客户端的回复可能并不重要，或者说客户端回复的内容并不重要。例如，当向零售商合作伙伴的企业信息系统发送一组新产品的价格时，消息的发布者并不期望一个回复，只需要确认合作伙伴已经收到信息即可。

2.5.3 事件驱动的处理机制

在客户/服务器系统中，通常使用一对一的请求/回复交互模式。但是系统若需要对环境变化、利率信息、处理状态等事件做出反应，则一对一的请求/回复交互模式对于这类系统并不合适。在这种情况下，最合适的将是没有集中控制的分布式环境。对于过程支持系统和工作流管理系统这类应用，最适合使用事件中间件进行构建。通过事件驱动的任务管理器，可以实现控制流和任务间的依赖关系。

单播和多播这两个传统的网络寻址和路由机制，都需要使用明确的、专门的寻址和路由信息，或者相关的消息。包含在消息中的实际数据(例如业务事件的属性)称为内容，有时也称为消息体或消息的有效载荷。对于传输机制而言，消息的内容是不可见的，因此在进行寻址及路由操作时，无须考虑具体的消息内容。在传统的网络地址和路由机制中，消息生成者和消息接收者之间的信息流是由消息生成者确定的，通过明确地规定目的地以及使用明晰的标识属性，消息生成者选定消息接收者。另一个完全不同的方式是将消息的内容暴露给网络传输机制，内容能够影响消息的寻址与路由。在极端情况下，仅根据消息内容决定消息的寻址与路由。采用这种方式的网络称为使用基于内容寻址与路由模式[Carzaniga 2000]。与传统的网络寻址与路由中的信息流不同，基于内容的寻址与路由隐含地来自于一些具体的相互作用，这些相互作用发生在意愿表达与所生成的任何信息之间。基于内容的方式通常基于对到来事件的处理。在这种方式中，消息生成者生成消息，但不指定特定的目的地。在消息发送中，根据客户表达的意愿确定目的地，从而使得目的地的确定与消息生成者无关，而仅是根据消息的具体内容来确定。

现今的广域网应用的特性是异步、异构和松耦合，从而推动了事件交互的应用。对于现今的软件系统的开发，事件交互可作为一个中立的设计抽象。这类软件系统通常基于一个称为事件通知服务的技术基础架构[Rosenblum 1997]。

事件通知服务提供了多到多的通信和集成工具，它与其他通用的中间件服务(如点到点的通信、多播通信等)互为补充。事件通知模式中有两类客户端：兴趣对象(objects of interest)和当事方(interested parties)。兴趣对象是通知的生成者，当事方是通知的使用者。值得注意的一点是，

客户端既可充当兴趣对象，也可充当当事方。事件通知服务通常用于实现我们在前面章节中所说的发布/订阅异步消息传送。在一个发布/订阅系统中，客户端发布含结构化信息的事件(或通知)消息，其他的客户端通过过滤器(过滤器的实质就是一种模式)确定订阅情况，即确定在客户端接收哪些内容的事件。位于底层的基于内容的路由网络是一组以对等(P2P)网络方式互连起来的服务器的集合，它们处理事件消息的分发。基于内容的路由器负责向所有的客户端发送事件消息，而客户端则使用过滤器获取所需的消息。

对于有关客户端所生成的信息，客户端使用它们本地服务器的访问点来公布这些信息，以及发布这些被公布的通知。客户端也使用访问点订阅单个的兴趣通知或者复合类型的兴趣通知。通知的最基本形式可以视为一个属性集，其中每一个属性都有一个属性名、属性类型和属性值。一些客户端发布有关已经发生的事件的通知，并且其他的客户端(也有可能是同一个客户端)订阅兴趣通知(也称意愿通知)。假如订阅者订阅的内容与通知相匹配，则通知会发送给这些订阅者。对于通过访问点发送到客户端的通知，选择服务负责选择客户端感兴趣的通知。因为可能有多个这样的订阅者，所以事件通知服务是一个多播服务。然而，由于使用了基于内容的寻址和路由，这使得事件通知服务与传统的多播服务完全不同。

事件通知服务通过“选择”(selection)处理来确定发布的消息与哪些客户端的兴趣相一致，并且仅路由和发送那些符合客户端兴趣的通知。除了服务于客户端的兴趣，事件通知服务也能使用选择处理来优化网络内的通信。驱动选择处理的信息源自于客户端。更具体地说，事件通知服务可以代表客户端使用过滤器来过滤事件通知的内容，从而事件通知服务可以仅发送那些包含指定数据值的通知。选择操作也可与多个事件模式相关，以便于事件通知服务能够仅仅发送与事件(一个过滤器过滤单个的事件)相关的通知集[Carzaniga 2001]。

通过指定属性集以及对于那些属性的值的约束，过滤器可选择事件通知。例如，对于特定产品的库存显著下降的通知，库存服务可能愿意接收这类通知。对于这类问题，包含过滤器的订阅可以指定产品名及库存等级。基于通知的属性数据，过滤器可以与通知进行匹配比较。基于两个或多个通知的属性数据及它们的组合形式，可以将模式与两个或多个通知进行匹配比较。从根本上，模式可以根据任何组合关系与事件关联。例如，当一些产品的供货同时改变产品价格时，有些顾客可能希望接收这类价格变化的通知。

为了在广域网上也具有可伸缩性，事件通知服务必须实现为服务器的分布式网络。事件通知服务的职责就是对经过服务器网络的每一个通知进行路由，将与所注册的订阅相匹配的通知发送给所有的订阅者。事件通知服务还将记录每一个注册订阅的订阅者的标识。通过对事件通知的大致描述，我们可以确定许多重要的特性。

对于开发面向服务的应用，事件通知模式尤其具有吸引力，相关内容可参见 7.4 节。可基于通知的内容而不是直接指定目的地址来发送通知，从而在很大程度上增加了服务客户端的灵活性和表达能力。然而，这提高了服务的实现难度。此外，订阅者的行为是动态的，可以添加和删除订阅。然而，对于通知的发布者来说，这种动态性是透明的，发布者可以生成与事件接收者不相关的内容。

2.5.4　点到点排队

一些大型系统可分为几个独立的单元。在点到点的消息传送模型中，客户端可通过队列发送和接收消息，并且既可以采取同步方式，也可以采取异步方式。对于多流程的应用，点对点的消息传送提供了可靠的通信。点对点的消息传送模式通常为拉式(pull)或轮训式(polling)，从队列中请求消息。而在发布/订阅模式中，则通常采用推(push)的方式，自动将消息推送到客户端。发布/订阅基于推的方式，这意味着，无须请求消息，即可将消息发送给用户。

在点到点的消息传送模型中，消息的生成者称为发送者，消息的消费者（用户）称为接收者（Monson-Haefel 2001）。通过队列可交换消息。多个接收者可以使用同一个队列，然而一次用户组中仅能有一个用户可以使用同一个消息。例如，在采用点到点消息传送应用的订单管理应用中，Web 前端发送一个包括新订单信息在内的消息，一个仓库接收该消息并处理这个订单。如图 2.13 所示，消息传送系统保证不会有多个仓库处理同一个订单。

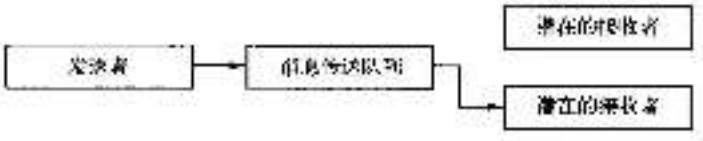


图 2.13 点到点的排队

正如发布/订阅消息传送模式，点到点的消息传送也是向一个命名的目的地发送消息。发布/订阅消息传送与点到点的消息传送之间的最主要的不同点在于，传送点到点的消息通常并不考虑接收者当前的连接状态，只需要将消息发送到合适的队列中即可，即使当前没有用户连接到该队列也无妨（Monson-Haefel 2001）。点到点的消息传送模型也保证了消息分发的可靠性。在发布/订阅模式中，可将消息直接发送到多个接收者。而在点到点的消息传送方式中，消息传送到队列，然后根据发送策略将队列中的消息分发给队列的接收者。

2.6 请求/应答的消息传送方式

迄今为止，我们已经分析了许多异步消息传送方式。这些异步发送方式遵循"即发即弃"消息传送原理。这意味着，一旦异步发送完消息后，发送应用可继续处理其他任务。正如前面所述，发送应用假定消息在某个时刻将会安全地抵达它的目的地。异步消息传送模式并不需要进行请求/应答操作。

在许多时候，应用需要进行请求/应答消息传送操作，这里，我们可以区分两类请求/应答消息传送操作：同步的请求/应答消息传送操作与异步的请求/应答消息传送操作。若 Web Service 客户端会因为同步响应堵塞或等待，当集成这样的 Web Service 客户端时，则要求同步的请求/应答消息传送（Chappell 2004）。在异步的请求/应答消息传送中，请求者（发送者）认为应答在稍后的某个时刻将会到达，并且会继续它的其他工作。

图 2.14 示例了一个简单的请求/应答异步消息传送配置。在该配置中，消息传输信道并不是双向的。为了完成请求/应答操作，发送者必须使用两个信道：一个用于请求，另一个用于应答。请求消息需要包含对接收端的引用以及一个关联标识符。关联标识符用于将请求与响应消息关联起来。为了获得应答消息，请求者将要轮询应答信道。

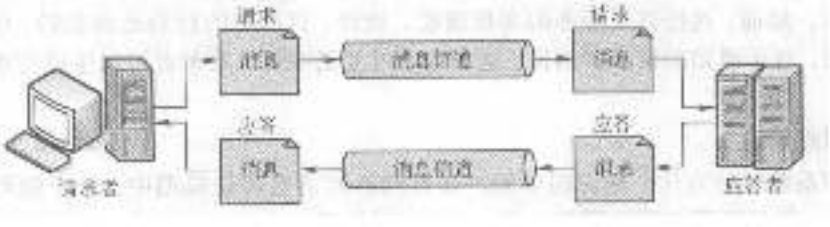


图 2.14 异步的请求/应答消息传送

请求/应答消息传送可以位于面向消息的中间件的顶部。通过管道请求/应答消息的内容，一些面向消息的中间件系统可以进一步自动处理这项任务。

对于 Web Service 来说，请求/应答消息传送是一个非常有用的机制。在本书后面有关可靠的消息传送与相关的 Web Service 标准的 8.4.2 节中，也可参阅请求/应答消息传送的相关内容。

2.7　面向消息的中间件

普通的通信信道可传输完整的消息。面向消息的中间件（MOM）是一个基础架构，它使用普通的通信信道在应用之间传送数据。在基于 MOM 的通信环境中，通常异步地发送和接收消息。使用基于消息的通信，可将应用抽象地划分为发送者与接收者，它们之间无须彼此了解。通过消息传送系统，可发送和接收消息。消息传送系统（即 MOM）的作用就是将消息转发到它们的目的地。

如图 2.15 所示，在客户/服务器体系结构中，MOM 位于客户端和服务器之间，并处理客户端和服务器之间的异步调用。为了支持异步模型，MOM 产品通常使用消息队列临时存储调用，并允许客户端和服务器分别在不同的时间运行。队列中的消息通常由格式化数据、操作请求或这两者组成。

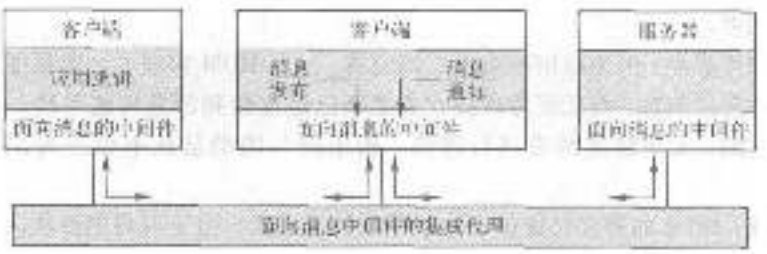


图 2.15　面向消息的中间件（MOM）

引自：J. Wiley & Sons 电数社 2005 年出版的，由 W. P. Papazoglou、P. M. A. Ribbers 所著的《e-Business: Organizational and Technical Foundations》一书（允许复制）

消息传送系统负责管理消息传送客户端之间的连接点，并负责管理连接点之间的多个通信信道。如图 2.15 所示，通常由称为消息（或集成）代理的软件模块实现消息传送系统。在 2.7.1 节中，将会进一步讨论集成代理。通常可将集成代理聚集起来形成集群，从而提供一些高级功能，诸如负载平衡、容错以及使用管理安全域的不同路由机制[Chappell 2004]。

当发生一个事件时，需要通知服务器采取一些操作，客户端应用将这一通知任务交由消息传送中间件负责。包含操作请求内容的消息能够触发其他消息。当发生一个事件时，通常需要一个通知。自然形式的通信将事件发布到不同的客户端订阅者。

MOM 产品的作用通常不仅仅是传递信息，它们往往还位于其他服务，如数据转换、安全性、将数据传送给多个程序、故障恢复、合并网络资源、成本路由、确定消息和请求的优先级以及一些大量的调试工具。MOM 消息传送系统已帮助将长时间运行的任务划分为多个事务，从而提高效率。与 RPC 和 ORB（对象请求代理）产品相反，MOM 并不要求存在一个可靠的传输层。当传输层并不可靠时，MOM 将会处理所出现的各类问题。

MOM 技术日前通常包含一些新特性，这些新特性可促进 Web Service 技术的开发，如：

- 支持事件驱动处理（例如发布/订阅）的消息多播。
- 消息的可靠性和串行化，从而保证消息按正确的顺序进行传送。
- 将由网络理解和实现的物理名和地址抽象为主题（文本）名和属性。
- 支持多种通信协议，如存储与转发、请求/应答、发布/订阅。
- 支持事务分界。

总的来说，当集成应用时，MOM 的有些特性非常具有吸引力，诸如以下这些(但不仅限于这些)特性：

- 异构系统间的透明协作：集成代理提供了转换软件，可将在不同的编程环境、操作系统和硬件平台上运行的应用数据进行转换。
- 请求的优先级：在许多情况下，有些服务将比其他的服务具有更高的优先级。在 MOM 环境中，所有的消息都可以赋予一定的优先级。当消息发送到客户端进程时，将按一定的优先级顺序添加到客户端的消息队列中。从而，具有更高优先级的消息将比之前发送的低优先级的消息更早得到处理。
- 自动消息缓冲和流量控制：分布式应用通常需要从不同的应用和程序中读取消息。为了支持这样的任务，每一个应用都可以有一些消息队列。当流量的速率发生变化时，消息队列可对消息进行缓冲，从而实现流量控制。这种通信方式增强了性能，并大大地简化了开发工作。
- 持久的消息发送：持久的消息发送提供了可靠性，并确保消息只需发送一次即可发送到消息订阅者。
- 灵活性和可靠性：因为应用不需要立即应答，所以 MOM 实现了一定程度的灵活性。MOM 可以随时发送消息，而无须考虑接收者是否已经连接到消息传送系统。发送者和接收者都是独立的，无须将这两者进行连接。由于持久的消息从不会丢失，这样就实现了可靠性。
- 负载平衡：MOM 的异步性提供了负载平衡的灵活性。由于可将消息从一个忙碌的系统转发到一个相对更清闲的系统，因此可以实现负载平衡。例如，假如一个应用系统比较忙碌，就可将消息转发给另一个地方的并发进程(假如存在的话)。在 MOM 环境中，通过使用一些算法，如“最小繁忙算法”、“循环算法”等，可以实现动态的负载平衡，从而提高了网络设施的利用效率。此外，负载平衡工具可以实现高可用性的消息队列，并可有效地实现峰值负载时的网络管理，从而低带宽的系统仍然能够实现令人满意的性能。然而，假如采用同步消息传送，则性能可能会降低到完全不能令人接受。
- 可伸缩性和资源的优化使用：当负载量增大时，MOM 代理可以采用动态路由和多路传送技术。通过动态路由，MOM 能够在没有开发者干预的情况下，将请求者自动连接到所需的服务，从而使没有预编程序的客户端和服务器之间也可以进行通信。此外，当一个服务器崩溃时，MOM 平台可将消息发送到另一个备份服务器。多路传送技术是 MOM 代理所提供的功能，多个应用可以共享一个消息队列。通常情况下，本地网络上每秒钟有几百个消息处理也是毫不奇怪的。

对于应用，如应用间的互操作、进程间的交互、分布式事务处理(诸如银行业务、经纪业务、机票订座)、分布式企业工作流(诸如加工制造、保险索赔处理)、实时自动化(诸如公用事业控制、处理控制)和系统管理(诸如分布式备份、软件分发)等，MOM 显示了它已完全实现了异步消息传送所具有的优点。

由于临时存储可能会导致网络过载，在 MOM 中使用消息队列的缺点之一，是有可能产生过载现象。很可能会出现这样一种现象，客户端不断传送大量的消息，然而服务器端却无法及时处理。许多 MOM 实现了一个很不错的功能，即能在异步消息传送模式和同步消息传送模式之间进行切换。虽然 MOM 原则上是一个异步对等协议，然而有些 MOM 的实现也能够处理同步消息传送。

2.7.1　集成代理

在结束有关 MOM 话题的讨论之前，下面将对集成代理进行简短的讨论，这对于 MOM 也是一个有益的补充。

集成代理完成所需的内容和格式转换，将收到的消息转换为目的系统能够理解和利用的形式。集成代理通常位于一些 MOM 实现的顶部，并且 MOM 的一些基本原理通常也适用于集成代理。集成代理最初被称为消息代理，然而这一名称容易造成它和面向消息的中间件的混淆。由于集成代理这一名称能更好地描述其功能，最近通常采用集成代理这一名称来取代消息代理的提法。

集成代理是一个应用之间的中间件服务，可进行一对多、多对一及多对多的消息分发。可将集成代理看成一个软件 Hub，它记录和管理消息发布者和订阅者之间的约定。当发生一个业务事件时，应用将会发布与该事件相关的消息。集成代理将会检查它的订阅列表，并激活发送操作，将消息发送给该消息的所有接收者，从而使得订阅者仅收到它们所订阅的数据。集成代理是一个高性能的通信模块，可支持大批量的消息传送。对于集成代理来说，每秒钟处理几百个消息是司空见惯的。

对于应用来说，集成代理通常基于队列管理器和路由消息之上。通过集成代理所提供的应用集成，多个应用都可以实现发布服务。除了这些功能，集成代理还可以处理应用的结构层次上的所有差异。通过对消息模式的记录与分析，并针对具体应用的语义相应地修改消息的内容，集成代理可处理结构上的不匹配。集成代理所具有的这些特有的功能，使得它们不仅可以在不同的应用之间进行代理，而且可以在不同类型的中间件之间进行代理。

集成代理所包含的组件可以提供下列功能：消息转换、业务规则处理、路由服务、名字服务、适配器服务、信息库服务、事件和警报。我们将依次讨论这些组件。

集成代理将与具体应用相关的消息转换为能被普遍理解的消息，例如使用 XSL(可扩展设计语言)的不同的 XML 模式。关于转换，可参见3.5.2 节的内容。集成代理可以使用转换规则进行转换。应用开发人员可以使用图形影射来定义这些转换规则。由于集成代理掌握了互换消息的各种模式，消息转换功能可“理解”在应用之间传送的所有消息的格式。通过重构这些消息数据，集成代理可以在不同的模式之间进行转换。这样，接收消息的应用程序将能理解所接收的消息的含义。

业务规则是一个精确的声明，它描述、约束和控制企业的结构、操作与策略[BRCommunity 2005]。业务规则可以表示定价和账单策略、服务质量、处理流程(描述路由决策、角色安排策略等)、规章等。通常将业务规则处理功能实现为集成代理内部的规则处理引擎。有时虽然在一个系统中创建消息，但可能需要在多个应用中使用该消息。根据业务规则，集成代理可以决定应该将消息送往何处。集成代理可对消息应用集成规则，从而使新的应用逻辑可驻留在集成代理中。

路由功能处理消息流控制，它确定消息的来源，并将消息路由到合适的目标应用。由于接收消息的应用为了理解消息的内容需要对消息进行转换，因此路由功能也会使用到消息转换功能。业务规则也可用于确定在何种情况下可将消息发送给接收者。

由于集成代理应用在分布式环境中，并且需要一个查找和使用网络资源的方法，因此需要目录服务功能。使用集成代理的应用可以发现网络上的其他应用或硬件。

集成代理将通信中间件和相关的不同类型的协议部署到特定的应用目标。许多集成代理将适配器作为集成代理和大型企业的后端系统之间的中间层。适配器可将源应用的数据格式和应用语义转换为目标应用的数据格式和语义。适配器向集成代理提供了有关企业内不同应用的数据及其访问。

适配器对于两个不同接口之间的差别进行映射。这两个接口分别是集成代理接口、源应用或目标应用的本地接口。适配器对于终端用户甚至开发者隐藏了接口的复杂性。例如，针对不同的源应用和目标应用(诸如打包的 ERP 应用)以及对于不同类型的数据库(诸如 Oracle、Sybase 或 DB2)，甚至对于一些具体的中间件产品，集成代理供应商都可以提供一些适配器。适配器可以位于集成代理中或驻留在目标应用环境中。利用各种应用的 API，适配器可以连接位于不同环境中的应用。

诸如 J2EE、XML 等标准的广泛使用，给采用标准化方式开发适配器打下了一个坚实的基础。对于应用集成来说，这些标准中最重要的可能是 J2EE 连接器体系结构[Sharma 2001]。对于将 J2EE 平台连接到异构的企业信息系统(参见 2.9 节)的适配器，J2EE 连接器体系结构定义了适配器开发的标准化方式。诸如 J2EE 连接器体系结构这样的标准的采用，使得企业可以开发一些合适的适配器，这些适配器将能工作在任何遵循 J2EE 规范的应用服务器上(参见 8.5.4.2 节，该节讨论了应用服务器)。对于分布式系统中的消息，消息转换可以实现各个应用与具体的消息格式之间的隔离。

信息库可以存储有关源应用和目标应用的大量信息，从而实现信息库服务。信息库记录了应用的输入/输出、应用的数据元素、应用间的相互关系，以及集成代理的其他子系统(如规则处理组件)的元数据。由于元数据可以描述不同的系统和处理的信息结构，因此在任何集成解决方案中，元数据都是关键要素之一。涉及元数据的场合很多，从关系数据库模式的结构到创建新账户的流程的描述，都离不开元数据。通过信息库与适配器间的相互协作，集成代理可以了解源应用和目标应用。基于元数据，集成代理可与源应用及目标应用相互交互。

基于事先规定的触发条件，穿过集成代理的消息传递可以触发事件或警报。这类触发条件可以用于跟踪业务处理的进展，以及创建一个新的消息、运行一个专门的应用程序或者发送一个警报作为响应。

2.7.2 Java 消息服务(JMS)

直到不久以前，MOM 产品还需要使用它们自己的专有编程接口。这意味着需要在适配器和中间件软件包上记录所有的应用。因此，使用某一个中间件产品的内部应用可能并不能使用另一个中间件产品。在最近的十年中，中间件厂商几次试图将 MOM 软件包的编程接口进行标准化，但进展不大。在 1999 年，情况终于发生了变化，Java 推出了 Java 消息服务(JMS)。JMS 是一个规定了编程接口集的框架。通过该编程接口集，Java 程序能够访问 MOM 软件。

JMS 是一个与具体厂商无关的 API，可以用于具有多个不同的 MOM 厂商产品的企业消息传送应用中。JMS 可以充当不同的消息传送产品的包装器，使得开发人员可以将精力主要集中于实际的应用开发和集成，而无须考虑各种特定产品的 API。应用开发人员可以使用相同的 API 来访问许多不同的系统。JMS 本身并不是一个消息传送系统。当不同的消息传送系统进行通信时，JMS 是消息传送客户端所需的接口和类的抽象。JMS 不仅提供了连接到 MOM 系统的 Java API，而且支持将消息传送作为一个与 RPC 等同的 Java 分布计算的最佳范例[Monson-Haefel 2001]。

在任何分布式计算环境中，若需要在应用单元之间以同步方式或异步方式传递数据，则基于 JMS 的通信都是一个可供选择的解决方案。一个常规的 JMS 应用涉及 EJB 和遗留应用之间的接口，以及在 EJB 和遗留应用之间传送数据。JMS 提供了两个最主要的 MOM 消息传送模式：点到点的排队和发布/订阅。

在 JMS 消息传送点到点模式中，JMS 客户端通过队列既能异步又能同步地发送和接收消息。一个特定的队列可以有多个接收者，但是每一个消息只能由一个接收者处理。例如，一个打包订单可能会发送到多个仓库，然而仅有一个仓库接收该消息并处理这个订单。

在 JMS 消息发布/订阅消息传送模式中，发布者将消息发送到一个特定名称的主题中，而订阅者则接收发送到该主题的所有消息。对于每一个主题可以有多个消息接收者，并且一个应用可以既是发送者又是接收者。

JMS 支持不同的消息发送方式，包括：一对一的消息、一对多的消息及多对多的消息。一对一的消息允许一个发布者(发送者)将一个消息发送给一个订阅者(接收者)。这个概念正是本章前面所讲的点到点的消息传送。一对多的消息允许一个发布者将一个消息发送给多个订阅者。多对多的消息允许多个发布者将一个消息发送给多个订阅者。

JMS 支持两类消息发送：可靠的消息发送和保证消息传送(guaranteed message delivery)。对于可靠的消息传送，只要没有应用程序故障和网络故障，消息传送服务器就会将消息发送给订阅客户端。假如出现一些错误，传送将会失败。这个特征即称为“最多一次发送”。对于保证消息传送，即使出现应用程序故障和网络故障，消息传送服务器依然会发送消息。消息传送服务器会将消息存储在它的持久化存储中，然后将该消息转发给订阅客户端。在客户端处理了该消息后，客户端会向消息传送服务器发送一个确认，表示它已经收到了该消息。

除了最基本的消息传送功能，JMS 还可以与其他 Java 技术配合，诸如 Java 事务 API(Java Transaction API)，从而支持分布式事务。在 JMS 标准中，对于有些典型的中间件问题，依然没有规定，如：管理和监控、负载平衡、容错、错误和警报通知、路由方式、线路协议和安全性。

2.8　面向事务的中间件

面向事务的中间件包括事务处理监视器。事务处理监视器可在许多不同的资源之间协调信息移动和方法共享。监视器是中间件的一部分，在这里对其进行简要的分析，以帮助读者完整地理解中间件。在第 10 章中，将会深入讨论 Web Service 的事务管理。

在分布式的客户/服务器环境中，事务处理(简称 TP)监视器技术提供了一个有效而可靠地开发、执行和管理事务应用的能力。通过协调和监控各个应用，事务处理监视器可以实现在线事务处理。事务处理监视器位于前端应用、后端应用之间。数据库管理事务数据的操作。通过将复杂的应用分解成事务集，事务监视器可实现流程管理和应用编配。在客户/服务器系统中，事务是一个将客户端绑定到一个或多个服务器的机制，事务也是实现恢复、一致性和并发的基础。从应用集成角度看，事务不仅是业务规则，而且是一种应用设计方法，可以保证分布式系统的一致性和强壮性。

事务有时需要涉及多个不相关联的功能，这些功能可能位于多个应用系统中。对于这样的事务，为了保证事务完整性，需要事务处理监视器。在事务处理监视器的控制之下，可以横跨多个服务器，从事务的源点(通常位于客户端)管理事务。当事务结束时，事务的所有参与方一起决定事务是成功还是失败。事务模型定义了事务何时开始、何时结束，以及出现故障时应进行何种恢复处理。

对于需要服务大量的客户端的应用，需要应用事务处理监视器。事务处理监视器位于客户端和服务器之间，可以管理事务，对事务进行跨系统路由，负载平衡事务的执行，以及在出现故障之后重启事务。事务监视器的路由子系统可将客户端请求交由一个或多个服务器进行处理。每一个服务器依次执行请求并响应。通常情况下，服务器管理文件系统、数据库或其他一些关键资源，并由多个客户端共享。事务处理监视器可以管理一个或多个服务器中的资源，多个事务处理监视器之间也可以按联邦方式进行相互协作。

从事务请求者的角度看，事务处理是同步的，事务处理可能会对应用性能带来影响。在整个事务完全完成之前，请求者必须一直等待，无法执行其他的任务。此外，在事务处理过程中，必

须锁定事务使用的所有资源，直到整个事务都已完成。在事务的执行过程中，没有其他的应用可以使用这些资源。事务监视器的另一个问题是：与 MOM 相比，事务监视器对于应用具有一定的干扰。这意味着，为了利用事务监视器的一些具体的服务，需要较多地修改应用本身。

2.9 企业应用程序与电子商务的集成

直到不久以前，传统的企业架构还是按照职能分隔将企业分成传统的层级组织，如销售、制造和采购。这种体系结构方式导致了多功能的业务。这些业务可以有效地以孤岛方式运行，关键数据都被“锁定”在各个分离的信息系统中。这些系统的设计目标主要是用于支持各个职能部门。这些系统相互之间完全孤立，并且不需要任何跨职能部门的协作。在这类企业内，应用基本上以自治方式独立运行，即使它们之间交换数据，也主要通过批处理文件传输的方式进行交换。

许多年来，对于这种相互孤立的企业架构，通常采用系统间的点到点的桥接方式来集成这些不同的系统。在点到点的集成方式中，通过手工编码，定制的连接系统将应用连接起来，并且在任何两个系统之间都是直接交换数据。这种方式通常对每一个信息系统以及信息系统之间的所有连接都“构建一个接口”。当添加新的连接时，这一方式有许多缺点，系统没有灵活性，管理不便，效率不高。

企业应用集成是一项重要的关键技术，可帮助企业避免数据孤岛现象，可将各种各样的定制应用和打包应用（包括遗留系统）集成在一起。企业应用集成（EAI）的目标就是将组织的内部应用转换为集成在一起的企业框架。为了实现这一点，企业应用集成以业务流程的形式将企业的各种应用无缝地集成起来，并可进行无缝的通信。

EAI 试图集成的企业内部应用称为企业信息系统（EIS）。企业信息系统包括业务处理以及企业内使用的 IT 资产，它向企业提供了信息基础架构[Sharma 2001]。企业信息系统以不同的抽象级别（包括数据级别、功能级别、业务对象或处理级别）向它的用户暴露了服务集。许多不同的应用和系统都可以共同作为企业信息系统：

- 企业为了满足具体的业务需求而开发的定制应用。这些应用可能使用不同的语言开发，如 C、C ++ 、Java。
- 管理企业内业务处理的关键数据的遗留系统和数据库应用。数据库和遗留系统都可以支持核心业务任务，诸如订单的接受和处理、产品的添加和交付、开发票、支付、分发、库存管理，以及相关的税收计算、成本节约和财会任务。在 8.5.6 节中将讨论遗留系统。
- ERP（企业资源规划）系统：这些是管理信息系统。通过将公司的操作和生产方面的许多业务进行集成和自动化，ERP 系统提供了一个统一的、全局性的业务流程视图及相关的信息。ERP 系统通常包括制造、物流、分发、库存、装运、发票和财会。ERP 系统通常称为“后台”（back-office）系统，这表明客户和普通公众通常并不直接涉及它。
- CRM（客户关系管理）系统：通过一些可靠的、自动处理服务的引入、个人信息收集和处理、供应链企业的自助服务，CRM 系统支持客户关系的处理，为客户创造价值。为了实现以上这些功能，CRM 系统需要访问和分析来自企业内各个数据库中的数据（其中大部分数据都可从 ERP 系统中获取）。CRM 系统称作“前台（front-office）”系统，它们直接帮助企业处理客户关系问题。
- TP（事务处理）系统和应用：在不同的数据仓库之间协调数据传送。

业务流程管理（简称 BMP）工具可直接管理跨职能的业务流程（参见 8.5.4.3 节）。EAI 解决方案将快速、强壮的通信中枢与集成代理技术、业务流程工作流、业务流程管理工具、针对具体

应用的适配器、完整的端到端应用控制和管理结合在一起，创造了一个适合新的业务流程的环境，从而提高企业的竞争优势[Papazoglou 2006]。图 2.16 显示了一个典型的集成各种企业信息系统的 EAI 解决方案。

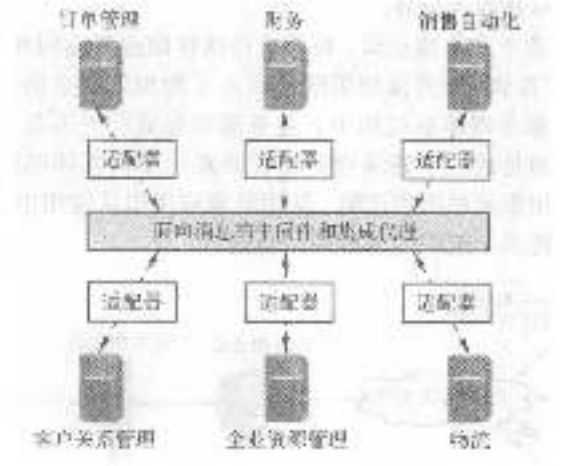


图 2.16　典型的企业应用集成解决方案

在 EAI 体系结构中，集成代理主要致力于消息和流程，并负责代理两个或多个应用之间的消息交换。集成代理能够转换、存储和路由消息，并可以应用业务规则，以及对事件进行响应。集成代理使用消息传送系统在系统间传送消息。同样，集成代理可以包含流程和工作流管理，从而可以管理状态以及长时间运行的事务。许多集成代理支持一些类别的异步传输层，这些异步传输层既可以是专有的，也可以是开放的。有些集成代理可以选择任意数量的消息传送系统，甚至能够将消息传送系统混合起来以及进行匹配，从而满足问题域的需求。根据被集成的应用的类别，信息代理可以使用几种抽象层次。例如，可以对常规的中间件服务（诸如分布式对象、MOM 和事务中间件）进行抽象。

EAI 解决方案通常使用应用服务器将企业与外部世界进行连接。对于应用集成来说，应用服务器非常有用。对于基于 Web 的、事务的、安全的、分布的和可伸缩的企业应用，应用服务器提供了一个开发、部署和管理平台。应用服务器可以通过协作接口与集成代理进行协作。在这种模式中，对于应用服务器来说，集成代理充当了服务提供者的角色，它提供了数据访问、转换以及基于内容的路由。

伴随着内部 EAI 解决方案的成功，电子商务（或企业间）集成解决方案也不断发展。电子商务集成方案可将贸易伙伴之间的不同的流程连接起来。通过电子商务集成，企业的内部系统可以与它们的客户、供应商和合作伙伴进行交互。例如，通过跨供应链的多个系统（例如制造、库存管理与仓储、物流、计费）之间的无缝的互操作，电子商务集成解决方案可致力于供应链的集成。供应链通常有三个主要部分：供应、制造与分发。供应主要专注于怎样、从哪里以及何时采购制造所需的原材料。制造则使用这些原材料生产产品。分发则确保生产出来的产品能够通过分发网络、仓库和零售商到达最终的顾客那里。通过集成跨供应链网络的信息和流程，厂商可以在提高库存周转的同时，降低采购成本。为了获得有用的业务信息，如消费预测，供应链需要具有互操作性。互操作性可以增强这些系统的可视性（visibility），从而可以集成和协调跨职能、跨企业的供应链业务流程。

当集成来自于不同企业的应用时，共享应用之间的业务逻辑是一个重要的因素。对于集成

业务应用，应用服务器提供了一个灵活的平台。应用服务器不仅提供了基于 Web 的业务应用的开发、部署和管理平台，而且可以使用同步或异步通信方法将这些应用服务器互连起来。对于集成电子商务应用，能将应用服务器互连起来是一个重要的必要条件。应用服务器可以协调复杂的集成步骤，并支持检查点和事务分界。

应用服务器通过调整业务和集成逻辑，将跨合作伙伴的业务应用互连起来，如图 2.17 所示。在图 2.17 中，我们还可以看到，业务流程集成还引入了附加的独立协调层。该协调层可以非侵入方式添加到已有的应用服务器体系结构中。业务流程集成层并不是主要致力于集成数据或业务逻辑这类的物理实体，而是致力于表示业务流程要素的逻辑实体的集成。在业务流程集成层中，应用服务器封装了应用集成的流程逻辑，从而将集成逻辑从应用中分离出来，包括事件序列和数据流，从而确保了横跨多个组织的事务的完整性。

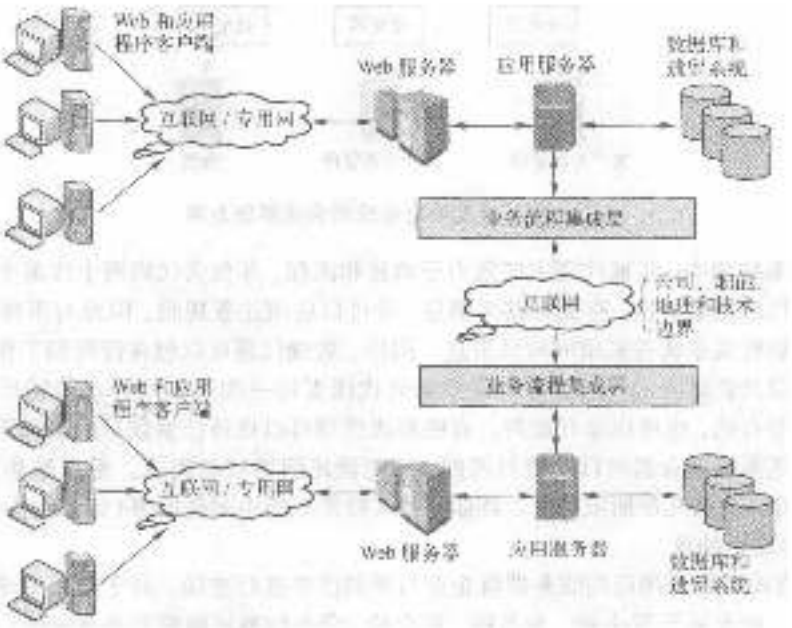


图 2.17 典型的电子商务集成解决方案

引自：J. Wiley & Sons 出版社 2006 年出版的，由 M. P. Papazoglou, P. M. A. Ribbers 所著的《e-Business: Organizational and Technical Foundations》一书（允许复制）

2.10 小结

分布式系统包含了一组通过网络互连的计算机。这些计算机很可能是异构的。它们之间能够相互通信，并可通过传递消息来协调计算机的操作。Web Service 已依赖互联网协议来进行跨互联网的数据传输。在分布式系统中，不同的组件之间基于标准进行相互通信或与远程组件进行通信。通过定义这些标准，互联网协议本质上为跨互联网进行数据传输的一些方法。

对于基于 Web Service 的分布式应用，可以帮助进行开发的其他关键技术包括远程过程调用、基于消息的中间件、事件处理机制、面向消息的中间件和集成代理。

分布式系统和应用通过交换消息进行通信。消息传送是一个程序间通信的技术，程序可以通过交换称为消息的数据包来进行通信。有许多不同类型的消息传送中间件。它们全部能够支持同步（或称做依时性）和异步（或称做时间无关）方式的消息传送。在同步方式中，在两个应用

系统之间的通信是同步的。同步通信需要两个系统必须都在正常运行，并且能够中断客户端的执行流，转而执行调用。发送程序和接收程序都必须一直做好相互通信的准备。远程过程调用和远程方法调用是典型的同步消息通信，而存储转发与发布/订阅方式则是异步通信。发布/订阅方式基于事件通知的思想，这一理念对于 Web Service 业界非常具有吸引力。

面向消息的中间件是一个基础架构，它使用普通的通信信道在应用之间传送完整的消息。面向消息中间件的异步性使得它非常适用于事件驱动的 Web Service 应用。集成代理是一个特殊类型的面向消息的中间件，对于 Web Service 应用来说，这一中间件非常具有吸引力。集成代理是一个应用之间的中间件服务，可进行一对多、多对一及多对多的消息分发。在 Web Service 应用中，可将集成代理看成一个软件 Hub，它记录和管理消息发布者与订阅者之间的约定。

复习题

- 什么是互联网协议？
- 描述两类消息通信。它们之间的主要区别是什么？
- 什么是远程过程调用？什么是远程方法调用？
- 最流行的异步消息传送方式是什么？
- 列举并描述发布/订阅消息传送与事件驱动处理的特征。它们之间的相互关系如何？
- 什么是点到点的排队？它与发布/订阅消息传送的不同之处是什么？
- 列举并描述面向消息中间件的作用与特性。
- 列举面向消息中间件的一些优点。
- 什么是集成代理？在 MOM 解决方案中如何使用集成代理？
- 定义并描述企业应用集成这一概念。
- 定义并描述电子商务集成这一概念。
- 什么是企业信息系统？如果没有对它们进行合理的集成，有可能导致什么问题？

练习

2.1　编写 Java 程序，实现：

(a) 获取主机地址，并检查是否可通过 80 端口进行连接。假如可以进行连接，应用程序发送一个 HTTP 请求应答作为应答回应。

(b) 在服务器的 8888 端口进行侦听，并回应所收到的任何请求。

2.2　实现如图 2.10 所示的一个简单的存储转发排队系统，其中 J2EE 应用客户端将向队列发送几个消息。服务器将异步接收发送到队列的消息。

2.3　实现一个在许多客户端之间进行分布计算的分布式应用，这些客户端位于不同的计算机上，并且它们能够使用 socket 连接到网络中的一台服务器上。一个任务可以分成许多作业，在不同的时候将这些作业安排到不同的客户端上。

2.4　编写一个 Java Web 应用程序，实现一个简单的电子商务应用。该应用描述了在线书店中的购物车。客户可向购物车中添加书籍、删除书籍或者查看购物车的内容。对于如何在 J2EE 平台上开发这类 Web 应用，读者可参阅 http://blueprints.dev.java.net/petstore/。

2.5　开发一个基于发布/订阅消息传送的简单的股票报价应用。股票报价器连续显示证券交易所的所有交易，包括公司名称、股票交易代码及股票价格。可将股票报价应用视为：对于证券交易所的每一个交易，向主题发送一个事件。交易员可以订阅合适的主题，并指定他们所关心的公司，从而交易员可以仅收到他们所关心的公司的事件。在本题中，消息选择器是公司名或公

司标志。股票交易中的每一个参与者（交易员）使用这个股票交易程序加入到一个具体的股票（主题），并向该主题中发送消息，以及接收来自于该主题的消息生成者的消息。

2.6 基于主题的公告牌系统由若干新闻组组成。这些新闻组是按照主题进行分类的，如酒、烹调、旅游胜地、服装等。用户既可以向这些新闻组发送消息，也可以从这些新闻组中读取消息。公告牌系统通常有一个或多个中央服务器，这些中央服务器存储消息。客户端可向服务器请求接收或发送消息。实现一个基于分布式体系结构的公告牌系统，该系统不依赖于中央服务器。具体实现可基于一个对等（P2P）体系结构，每一个用户（Peer）可向其他用户发送消息，也可阅读来自于其他用户的消息。推荐读者使用 JXTA 技术（http：//www. jxta. org/）解决这个问题。JXTA 是一个开放的协议集，用于连接网络上的各类设备，如手机、无线 PDA、个人计算机、服务器等。这些设备之间可相互通信，并可基于对等模式进行协作。JXTA 创建了一个虚拟网络，其中每一个用户都可与其他用户进行交互，当用户与资源位于不同的网络传输时，用户甚至可以直接与资源进行交互。

第3章 XML概览

学习目标

在第2章中，我们已经分析了Web Service的基石——分布式和基于Web的计算。在本章中，我们将讨论XML如何组织信息、描述信息及交换信息。通过XML，不同的应用可以灵活地交换信息，因此可以将XML作为Web Service的构建，这是XML最具吸引力的特点之一。Web Service技术完全基于XML，在进行贸易的企业间以及在异构的计算基础架构中，可以使用XML模式定义语言精确地处理数据的表示，以及对信息交换的整个流程进行优化。

读者需要对XML、XML名字空间及W3C模式语言具有很好的掌握，以便于能够理解一些基本的Web Service技术，如SOAP、WSDL、UDDI和BPEL。因此，本章对XML进行了简要的介绍，以帮助读者掌握本书后面的内容，具体内容将包括：

- XML文档结构。
- XML名字空间。
- 定义模式。
- 通过复杂类型扩张和多态类型实现模式的复用。
- 通过模式的导入和包括实现模式的复用。
- 文档导航和XML路径语言。
- 文档转换和可扩展样式表语言转换(XSLT)。

对于有关XML和XML的更详细的信息，读者可参阅下列书籍：[Walmsley 2002]、[Skonnard 2002]、[Valentine 2002]。

3.1 XML文档结构

XML是一种可扩展标记语言，用于网络上电子标记文本的描述和发送。与其他标记语言相比，XML具有两个重要的特点：XML的文档类型与XML的可移植性。

文档类型这一概念是XML的一个重要方面。XML文档具有不同的类型，XML的组成部分及XML的结构形式地定义了文档的类型。

XML的另一个基本特点是XML文档可在不同的计算环境中移植。无论使用何种语言或何种编写系统，所有的XML文档都采用了相同的字符编码模式。Unicode是一个标准的编码系统，它支持不同语言的字符集。可采用国际标准的Unicode来定义XML文档的编码。

XML文档由命名容器及这些命名容器所包含的数据值组成。这些容器通常表示为声明、元素和属性。声明(declaration)确定了XML的版本。在XML中，元素(element)这一技术术语用于表示一个文本单元，可视为一个结构化组件。可以定义元素容器来保存数据和其他的元素，然而元素中也可以什么都不保存。

一个XML文档也称为一个实例或者称为XML文档实例。这意味着，XML文档实例将一个可能的数据集描绘为一个特定的标记语言。清单3.1是一个典型的XML文档实例，该实例显示了一个与订购单相关的账单信息，该信息由塑料制造商发出。我们假设该公司已经建立了一个即期合约业务，可提供特制品与定制的塑料元件。

清单 3.1 XML 文档实例的例示

```
<?xml version="1.0" encoding="UTF-8"?>
<BillingInformation>
    <Name> Right Plastic Products </Name>
    <BillingDate> 2002-09-15 </BillingDate>
    <Address>
      <Street> 158 Edward st. </Street>
      <City> Brisbane </City>
      <State> QLD </State>
      <PostalCode> 4000 </PostalCode>
    </Address>
</BillingInformation>
```

3.1.1 XML 声明

XML 文档中开头的一些字符必须标记一个 XML 声明。XML 处理软件会根据声明来确定如何处理后面的内容。如图 3.1 所示，XML 文档通常以一个遵循 XML 1.0 版本标准及 UTF-8 编码标准的声明开始，即：<? xml version = "1.0" encoding = "UTF - 8" ? >

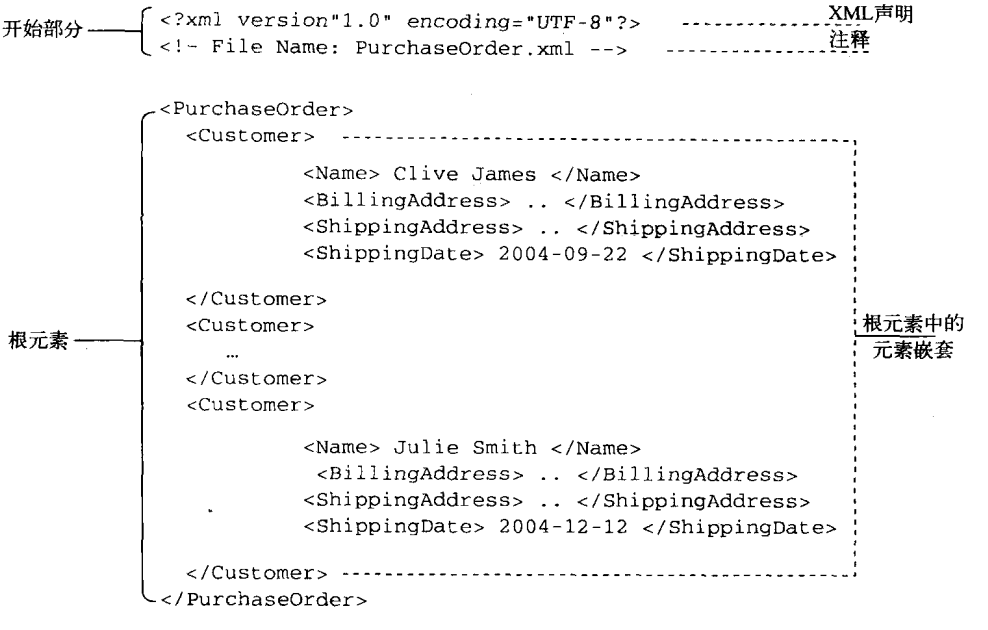


图 3.1 XML 文档的典型布局

3.1.2 元素

XML 文档的内部结构大致上类似于层次性的目录或文件结构。XML 文档最顶端的一个元素称为根元素。元素的内容可以是字符数据、其他的嵌套元素，或者是这两者的组合。包含在其他元素中的元素称为嵌套元素。包含其他元素的元素称为父元素，而嵌套元素则称为子元素。如图 3.1 所示，PurchaseOrder 元素包含 Customer 元素，而 Customer 元素又包含 Name 元素、BillingAddress 元素和 ShippingAddress 元素。

包含在文档中的数据值称为文档的内容。由于元素通常有一个说明性的名字，并且元素的属性包含元素值，因此文档的内容通常是直观的、自明的。这意味着 XML 具有自描述的特性[Bean 2003]。

不同类型的元素具有不同的名字，但是对于特定类型的元素，XML 并不提供表示这些类型

元素的具体含义的方法，而是表示了这些元素类型之间的关系。例如，清单 3.2 中的 < Address > 元素可能出现在(也可能不出现在) < Customer > 类型的元素中，并且 < Address > 可能被(也可能不被)分解为 < StreetName > 类型的元素和 < StreetNumber > 类型的元素。

3.1.3 属性

在 XML 文档中存放数据的另一种方式是在起始标签(start tag，也称开始标签)中添加属性。使用属性可以向被定义的元素中添加信息，从而可以更好地表示元素的内容。属性是通过与元素关联的名 - 值对来表示的。清单 3.2 就是一个元素描述的具体例子，在该例中使用属性(阴影部分)来指定 manufacture 这一特定客户类型。

清单 3.2 清单 3.1 中使用属性的一个具体实例

```
<?xml version="1.0" encoding="UTF-8"?>
<BillingInformation customer-type="manufacturer">
    <Name> Right Plastic Products </Name>
    <BillingDate> 2002-09-15 </BillingDate>
    <Address>
      <Street> 158 Edward st. </Street>
      <City> Brisbane </City>
      <State> QLD </State>
      <PostalCode> 4000 </PostalCode>
    </Address>
</BillingInformation>
```

每一个属性都是一个名 - 值对，其中值必须括在单引号或双引号中。与元素不同，属性不可以嵌套，并且必须在元素的起始标签中进行声明。

3.2 URI 和 XML 命名空间

Web 中汇集了各种资源。资源可以是具有标识的任何事物，如文档、文件、菜单项、计算机、服务等，甚至可以包括人、组织和概念[Berners-Lee 1998]。在 Web 体系结构中，资源标识符具有统一的语法，可以通过资源标识符来表示资源、描述资源、访问资源及共享资源。在 WWW 中，统一资源标识符(Uniform Resource Identifier，URI)是标识资源的基础。一个 URI 中包含能够唯一地标识一个资源的一串字符串。URI 使得每一个元素名都具有唯一性，从而使得元素名相互之间不会彼此冲突。

相比于我们原先所熟悉的术语——统一资源定位符(Uniform Resource Locator，URL)，W3C 使用了更新的、含义更广的术语 URI 来描述网络资源。URI 是一个总括性的术语，它指的是互联网资源寻址字符串，该字符串可使用目前及未来的任何寻址模式[Berners-Lee 1998]。URI 既包括使用传统寻址模式(如 HTTP 和 FTP)的 URL，也包括统一资源命名(Uniform Resource Names，URN)。URN 是 URI 的另一种形式，它提供了持久性及位置独立性。URN 以位置独立性的方式给互联网资源赋予地址。与 URL 不同，URN 随着时间的推移一直保持稳定。

在 XML 中，设计人员可以选择他们自己的标签名，这可能会导致命名冲突(例如在不同的上下文中可能会使用相同的标签名)，两个或多个设计人员可能会为他们的元素选择相同的标签名。XML 命名空间对于这种情况提供了一种对元素进行区分的方法，不同的元素可以具有相同的本地名，然而实际上这些元素的名字是不同的，从而避免了名字的冲突。例如，命名空间可以识别一个地址是邮政地址、电子邮件地址还是 IP 地址。命名空间中的标签名必须是唯一的。

为了更好地理解命名空间的概念，现以清单 3.3 为例。该例表示了一个 XML 文档，在该文档中包含了地址信息，但没有相关的命名空间。

清单 3.3 不与命名空间关联的 XML 样例

```
<?xml version="1.0" encoding="UTF-8"?>
    <Address>
      <Street> 158 Edward st. </Street>
      <City> Brisbane </City>
      <State> QLD </State>
      <PostalCode> 4000 </PostalCode>
    </Address>
```

现在，假设将清单 3.3 中的 Address 标记与清单 3.2 中的 BillingInformation 标记进行比较，可以发现两个标记都包含 Address 元素。事实上，在 XML 模式定义语言中，Address 标记有它自己的模式。对于 XML 文档中所使用的地址信息，Address 声明应该是可复用的，并且必须遵循 Address 标记模式。这意味着清单 3.2 中的 Address 元素必须遵循 Address 标记，并且清单中的其他元素必须遵循 BillingInformation 标记。在 XML 中，通过命名空间可以实现这一点。

在 XML 命名空间中，可将文档中的全部或部分的元素、属性与特定的模式进行关联。所有的命名空间声明都有作用域，例如它们所作用的元素。命名空间中所声明的元素以及这些元素的子元素都在作用域中。命名空间中的名字与元素的本地名这两者一起构成了全局唯一名，该名字也称为限定名(qualified name)[Skonnard 2002]。限定名通常称为 QName。限定名由前缀及本地名组成，在前缀及本地名之间以冒号分隔。

URI 表示了命名空间名，它表明了命名空间声明。可将 URI 映射到一个前缀，这个前缀可用在标签或属性名的前面，并以冒号相隔。为了引用命名空间，应用开发人员首先需要创建命名空间的声明，创建命名空间声明的方式如下：

```
xmlns:<Namespace Prefix> = <someURI>
```

当在元素和属性的本地名前面添加上前缀后，元素和属性则关联到合适的命名空间中，如清单 3.4 所示。URL 是最常见的 URI，因此在例子中将 URL 作为命名空间名(通常假设它们是具有唯一性的标识符)。在该例中，两个 URL 分别充当了 BillingInformation 元素和 Address 元素的命名空间。这些 URL 可非常简单地用于标识和定界，它们无须指向任何实际的资源或文档。

清单 3.4 一个使用命名空间的 XML 实例

```
<?xml version="1.0" encoding="UTF-8"?>
<BillingInformation customer-type="manufacturer"
    xmlns="http://www.plastics_supply.com/BillInfo">
   <Name> Right Plastic Products </Name>
   <Address xmlns="http://www.plastics_supply.com/Addr">
     <Street> 158 Edward st. </Street>
     <City> Brisbane </City>
     <State> QLD </State>
      <PostalCode> 4000 </PostalCode>
    </Address>
    <BillingDate> 2002-09-15 </BillingDate>
</BillingInformation>
```

在清单 3.4 中，对于所关联的元素和它的所有声明，xmlns 是默认的命名空间。默认元素的作用域仅应用于它自身以及它的所有后代。这意味着声明 xmlns = "http://www.plastics_supply.com/Addr" 仅应用于嵌套在 Address 中的元素。声明 xmlns = "http://www.plastics_supply.com/BillInfo" 仅应用于在 BillingInformation 中声明的所有元素，但是不包括 Address 中的元素，因为 Address 中的元素有它们自己的默认命名空间。

当元素间相互交错时，或者当同一个文档中使用不同的标记语言时，使用默认的命名空间可能会导致混乱。为了避免这个问题，对于关联的元素和属性的命名空间，XML 定义了简化标记，如清单 3.5 所示。

清单 3.5 在 XML 中使用限定名

```
<?xml version="1.0" encoding="UTF-8"?>
<bi:BillingInformation customer-type="manufacturer"
     xmlns:bi="http://www.plastics_supply.com/BillInfo"
     xmlns:addr="http://www.plastics_supply.com/Addr">

    <bi:Name> Right Plastic Products </bi:Name>
    <addr:Address>
      <addr:Street> 158 Edward st. </addr:Street>
      <addr:City> Brisbane </addr:City>
      <addr:State> QLD </addr:State>
      <addr:PostalCode> 4000 </addr:PostalCode>
    </addr:Address>
    <bi:BillingDate> 2002-09-15 </bi:BillingDate>
</bi:BillingInformation>
```

使用正确的文档可以极大地增强文档处理的质量。通过正确的 XML 文档，用户可以进行各类业务处理，如内容管理、电子商务交易、企业集成等。在这些业务处理中，需要交换有意义的、符合格式的 XML 文档。

3.3 定义 XML 文档中的结构

模式是定义 XML 标签和 XML 结构的方式之一。模式具有很强的表达 XML 文档的能力。模式支持元数据特性，如结构关系、基数(cardinality)、有效值、数据类型等。每一类模式都可作为一种描述数据特性的方式，并可以通过规则和约束来引用文档[Bean 2003]。在数据库技术中，模式这一术语通常指的是数据库的逻辑结构。而在 XML 技术中，模式通常指的是一种特定的文档，该文档用于定义某一类别的 XML 文档的内容和结构。

3.3.1 XML 模式定义语言

为了为 XML 处理环境提供类型系统，W3C 提出了 XML 模式定义语言(XML Schema Definition Language，XSD)。XSD 提供了一种粒度化的方式，可用于描述 XML 文档的内容。对于数据类型、定制和复用，XSD 也提供了强大的功能[Bean 2003]。对于验证 XML 文档，XSD 也提供了一种非常强大灵活的方式。XSD 包含了一些工具，可用来声明元素和属性，复用其他模式的元素，定义复合元素，定义约束(甚至对于最简单的数据类型也可定义约束)，从而使得 XML 模式开发人员可以直接控制 XML 文档的构建。例如，文档定义可以规定元素内容的数据类型、元素值的范围、元素可以出现的最大次数和最小次数、对于模式的注解等。

XML 模式由模式组件构成。模式组件是一些构建模块，可用于构建模式的抽象数据类型。元素和属性的声明、复合类型和简单类型的定义及通知等，都属于模式组件。对于良构的元素和属性信息项，模式组件可用于评估的有效性，并且可以使用模式组件扩充那些项和它们的后代。

XML 模式组件包括[Valentine 2002]：数据类型(包括简单数据类型、复合/复杂数据类型、可扩展的数据类型)、元素类型和属性声明、约束、元素之间的关联关系、命名空间，并且支持模块化的 import/include 选项。通过外部管理的 XML 模式，import/include 选项可用于包含可复用的结构、容器和自定义数据类型。

3.3.2 XML 模式文档

模式具有很强的验证 XML 文档的功能。由于模式可用于阐明存储在 XML 文档中的数据类型，因此可以使用模式检查 XML 文档的准确性。清单 3.6 表示了一个订购单的 XML 模式。

清单 3.6 一个定购单模式

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:PO="http://www.plastics_supply.com/PurchaseOrder"
  targetNamespace="http://www.plastics_supply.com/PurchaseOrder">

  <!-- Purchase Order schema -->
  <xsd:element name="PurchaseOrder" type="PO:PurchaseOrderType"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:all>
      <xsd:element name="ShippingInformation" type="PO:Customer"
           minOccurs="1" maxOccurs="1"/>

      <xsd:element name="BillingInformation" type="PO:Customer"
           minOccurs="1" maxOccurs="1"/>
      <xsd:element name="Order" type="PO:OrderType" minOccurs="1"
           maxOccurs="1"/>
    </xsd:all>
  </xsd:complexType>

  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element name="Name" minOccurs="1" maxOccurs="1">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string"/>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="Address" type="PO:AddressType"
           minOccurs= "1" maxOccurs="1"/>
        <xsd:choice>
        <xsd:element name="BillingDate"  type="xsd:date"/>
        <xsd:element name="ShippingDate" type="xsd:date"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="Street"          type="xsd:string"/>
      <xsd:element name="City"            type="xsd:string"/>
      <xsd:element name="State"           type="xsd:string"/>
      <xsd:element name="PostalCode"      type="xsd:decimal"/>
    <xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="OrderType">
    <xsd:sequence>
      <xsd:element name="Product" type="PO:ProductType"
                        minOccurs= "1" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="Total">
      <xsd:simpleType>
        <xsd:restriction base="xsd:decimal">
          <xsd:fractionDigits value="2"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="ItemsSold" type="xsd:positiveInteger"/>
  </xsd:complexType>

  <xsd:complexType name="ProductType">
    <xsd:attribute name="Name" type="xsd:string"/>
    <xsd:attribute name="Price">
```

```
      <xsd:simpleType>
        <xsd:restriction base="xsd:decimal">
          <xsd:fractionDigits value="2"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Quantity" type="xsd:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>
```

清单 3.6 描述了清单中的不同项。该文档允许客户接受货物的运送，并将生产厂商和账单信息发送到客户总部。该文档也包含了所订购的产品的一些具体信息，诸如每一件产品的成本、订购量等。XML 模式文档(诸如订购单模式)的根元素通常是模式元素。在模式元素中，可以嵌套元素和类型声明。例如，订购单模式由模式元素和不同的子元素组成。在实例文档中，元素 complexType 和 simpleType 确定了元素的外观和它们的内容。在后面的章节中，将会讨论这些组件。

schema 元素将 XML 模式命名空间("http: //www. w3. org/2001/XMLSchema")指定为默认的命名空间。该模式是根据 XML 规范定义标准的模式命名空间，并且所有的模式元素都必须属于该命名空间。schema 元素也定义了 targetNamespace 属性。对于在模式中显式地新创建的所有类型，targetNamespace 属性声明了它们的 XML 命名空间。如清单所示，在 schema 元素中，将前缀 PO 赋给了 targetNamespace 属性。对应的 XML 文档，若所包含的元素声明为属于模式的命名空间，在给模式指定一个目标命名空间后，则可根据模式来验证这些 XML 文档。因此，在文档中可使用 PO targetNamespace，以便它们与订购单模式一致。

模式的主要目的是用于定义 XML 文档的类别。实例文档这一术语通常指符合某一特定模式的 XML 文档。清单 3.7 表示了一个与清单 3.6 所示的模式相符的实例文档。

为了进一步分析 XML 模式，本节的后面部分将继续讨论清单 3.6 中所示的 XML 文档。

清单 3.7 一个符合清单 3.6 中模式的 XML 实例文档

```
<?xml version="1.0" encoding="UTF-8"?>

<PO:PurchaseOrder
 xmlns:PO="http://www.plastics_supply.com/PurchaseOrder"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.plastics_supply.com/PurchaseOrder
     purchaseOrder.xsd">

  <ShippingInformation>
    <Name> Right Plastic Products Co. </Name>
    <Address>
      <Street> 459 Wickham st. </Street>
      <City> Fortitude Valley </City>
      <State> QLD </State>
      <PostalCode> 4006 </PostalCode>
   </Address>
    <ShippingDate> 2002-09-22 </ShippingDate>
  </ShippingInformation>

  <BillingInformation>
    <Name> Right Plastic Products Inc. </Name>
    <Address>
      <Street> 158 Edward st. </Street>
      <City> Brisbane </City>
      <State> QLD </State>
      <PostalCode> 4000 </PostalCode>
    </Address>
```

```
    <BillingDate> 2002-09-15 </BillingDate>
  </BillingInformation>

  <Order Total="253000.00" ItemsSold="2">
    <Product Name="Injection Molder" Price="250000.00"
     Quantity="1"/>
    <Product Name="Adjustable Worktable" Price="3000.00"
     Quantity="1"/>
  </Order>
</PO:PurchaseOrder>
```

3.3.3 类型定义、元素和属性声明

XSD 对不同的复合类型进行了区分。复合类型通过元素定义了它们的类型。定义复合类型的元素中，还可进一步包含元素、属性及简单类型。定义简单类型的元素和属性中，仅能包含数据。通过定义可以创建新的类型(简单类型或复合类型)。通过声明，具有特定名字和类型(简单类型或复合类型)的元素和属性可以出现在文档实例中。XSD 对定义和声明进行了明显的区分。在 XML 文档中，通过在模式中声明元素或属性，可以指定元素或属性在特定上下文中的名字、类型和其他一些特性。

1. 元素声明

元素是模式的主要组成部分。可以使用 XSD 中的 < xsd: element > 来声明元素。元素声明定义了元素名、内容模型[⊖]以及每一个元素类型所允许的属性和数据类型。W3C XML 模式支持多种数据类型，包括大量的内置数据类型及派生的数据类型。可以将数据类型作为任何元素或属性上的约束。既可以使用 < xsd: element > 来进行元素声明，< xsd: element > 是对这类声明的一个引用，也可以使用 < xsd: element > 定义命名元素，或者使用 < xsd: element > 将元素与类型进行关联[Skonnard 2002]。

在 XML 文档中，最顶端的元素称为根元素。在每个 XML 文档中仅有一个根元素。在根元素中，其他元素和元素组可以出现多次。由于元素中可以包含其他元素，从而产生了嵌套，并导致了层次结构。元素也可以包含属性。有些元素也可以故意定义为空值。

元素所定义的位置确定了元素在模式中的可用性。若元素声明是 < xsd; schema > 元素的直接后代，则称为全局元素声明。可在模式文档或其他文档中的任意地方直接引用全局元素声明。例如，清单 3.6 中的 PurchaseOrderType 是全局定义的，并且它组成了模式中的根元素。元素通常是模式的目标命名空间的一部分，全局元素声明描述了这些元素。复合类型既可以直接定义，也可以通过组引用间接定义。元素声明是复合类型定义的一部分，复合类型定义中的元素声明称为局部元素声明。在清单 3.6 中，局部元素声明包括 Customer 元素和 ProductType 元素。

在定义元素内容时，可以使用合成器(Compositor)将一些已有的类型聚合成结构，从而定义和约束子元素的行为。对于在复合类型或组中定义的容器，合成器指定了这些容器的顺序和具体选择。在 XML 模式中，可以使用三类合成器，它们分别是 sequence、choice 和 all。在 sequence 构成中，在复合类型和组中定义的各个元素的次序必须与对应的 XML 文档(内容模型)一致。在 choice 构成中，对于复合类型或组的大量选项，文档设计人员必须进行具体选择。在 all 构成中，包含在复合类型或组中的所有元素都可以按任何顺序出现一次或者一次也不出现。

2. 属性声明

在 XML 文档中，元素可以包含属性。然而 XML 属性不能进行嵌套，并且 XML 属性既不能重复，也不能具有基数。可以使用 XSD 的 < attribute > 元素表示复合元素的属性。例如，从清单 3.6

⊖ 在有些文献中，也将内容模型(Content Model)称为内容模式。——译者注

中我们可以发现，当定义属性时(例如 Total)，必须指定它的类型。这个类型必须是简单类型之一，如 boolean、byte、date、dateTime、decimal、double、duration、float、integer、language、long、short、string、time、token 等。该例显示了可以基于 simpleType 元素定义属性。

3.3.4 简单类型

在大多数编程语言中，开发者可以在结构化类型中使用不同的内置数据类型，然而无法定义用户可以指定值空间的新的简单类型。在这一点上，XML 与这些编程语言并不相同。在 XML 语言中，用户可以自定义自己的简单类型，自定义的简单类型的取值空间可以是已有的内置类型的子集。在 XML 中，可以通过创建 simpleType 来自定义数据类型。可以基于一个所支持的数据类型，然后向这个数据类型中添加约束来创建新的数据类型。

在清单 3.6 中，Customer 中的简单元素 Name 的取值仅可以为字符串。此外，在清单中还规定了每一个简单属性 Name、BillingDate 和 ShippingDate 都必须作为 Customer 元素的子女出现一次。约束属性 minOccurs 和 maxOccurs 指定了这些元素可以出现的最少次数和最多次数。在清单 3.6 中，Total 和 Price 这样的简单属性类型的取值限制为十进制数值，并且小数点右边仅可以有 2 位。

3.3.5 复合类型

可以使用 complexType 元素来定义结构化类型。一个元素假如包含子元素和/或子属性，则该元素为复合类型。复合类型定义显现为 xsd：schema 元素的子女，并且可以从模式中的其他地方或者其他模式中引用复合类型定义。复合类型定义通常包含一组元素声明、元素引用和属性声明。

在清单 3.6 中，PurchaseOrderType 就是一个复合类型。这个元素包含三个子元素——ShippingInformation、BillingInformation 和 Order，此外还包含属性 Total。在元素声明中有属性 maxOccurs 和 minOccurs，这两个属性分别都赋予了值 1，这表明，元素声明规定了 PurchaseOrderType 元素必须仅出现一次。

可以使用内容模型来声明元素。元素可以包含一个或多个子元素，子元素可以具有一个或多个指定类型。元素还可以包含一些属性。为了声明包含内容的元素，模式开发人员必须使用 xsd：complexType 元素来定义元素的类型，并需要包含内容模型。内容模型描述了所有允许的子元素及它们出现的规则。在 XSD 中，可以使用元素 all、choice、sequence 或它们的组合形式。例如，在清单 3.6 中使用 xsd：sequence 和 xsd：choice 组合元素，将 Customer 定义为复合类型元素。当在 xsd：sequence 模式元素中声明一组元素或属性时，需要使用 xsd：sequence 元素。这些元素或属性必须严格按照顺序出现，例如复合类型 Customer 中的 Name 元素和 Address 元素。当在模式元素 < xsd：choice > 中声明一组元素或属性时，需要使用 < xsd：choice > 元素。任何一个子元素(但不是所有的子元素)都可以在父元素的上下文中出现，例如复合类型 Customer 中的 BillingDate 属性和 ShippingDate 属性。

3.4 XML 模式复用

在企业级解决方案中，XML 设计人员所面临的最大挑战之一，就是如何设计可以复用的结构。使用可复用的组件设计 XML 模式可以带来许多益处，从而缩短开发周期，减少应用开发成本，简化代码的维护，以及促进企业数据标准的使用。

3.4.1 派生的复合类型

XML 允许从已有的简单类型或复合类型中派生出复合类型。可以通过扩展或约束的方式，

从其他类型派生出复合类型[Walmsley 2002]。通过扩展，可以向已有的(基)类型中添加附加的后代和/或属性。通过限制，可以指定类型的取值范围。新类型的取值范围是基类型的取值范围的子集。

1. 复合类型的扩展

通过添加属性以及添加到内容模型，可以扩展复合类型，然而不能修改或删除已有的属性。当定义复合类型扩展时，为了处理扩展，XML 处理器会在基类型的内容模型之后附加新的内容模型。在 sequence 合成器构成中，基类型的内容模型和附加新的内容模型好像是一体的。

清单 3.8 扩展 XML 复合类型

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:PO="http://www.plastics_supply.com/PurchaseOrder"
  targetNamespace="http://www.plastics_supply.com/PurchaseOrder">

      <xsd:complexType name="Address">
         <xsd:sequence>
             <xsd:element name="Number" type="xsd:decimal"/>
             <xsd:element name="Street" type="xsd:string"/>
             <xsd:element name="City" type="xsd:string"
               minOccurs="0"/>
         </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="AustralianAddress">
          <xsd:complexContent>
              <xsd:extension base="PO:Address">
                          <xsd:sequence>
                                 <xsd:element name="State"
                                  type="xsd:string"/>
                                 <xsd:element name="PostalCode"
                                  type="xsd:decimal"/>
                                 <xsd:element name="Country"
                                  type="xsd:string"/>
                          </xsd:sequence>
              </xsd:extension>
          </xsd:complexContent>
      </xsd:complexType>
</xsd:schema>
```

清单 3.8 阐明了如何扩展诸如 Address(包含门牌号码、街道和城市)这样的复合类型。清单中的 City 元素是可选的，属性 minOccurs 的值为 0 表明了这一点。可以使用清单 3.8 中的基类型 Address 创建其他的派生类型，如 EuropeanAddress 或 USAddress 等。

2. 复合类型的约束

通过删除属性、对属性进行约束以及设置内容模型的子集，可以对复合类型进行约束。当使用约束时，对于基类型来说，派生类型的实例也将是有效的。

例如，开发人员可以基于 AustralianAddress 类型创建一个新的类型 AustralianPostalAddress，如清单 3.9 所示。这个新的类型删去了 City 元素。在澳大利亚地址中，假如包含了州和邮政编码，则不需要包含城市。

清单 3.9 通过约束定义复合类型

```
<!-- Uses the data type declarations from Listing 3.8 -->
<xsd:complexType name="AustralianPostalAddress">
```

```
    <xsd:complexContent>
       <xsd:restriction base="PO:AustralianAddress">
          <xsd:sequence>
             <xsd:element name="Number" type="xsd:decimal"/>
             <xsd:element name="Street" type="xsd:string"/>
             <xsd:element name="City" type="xsd:string"
              minOccurs="0" maxOccurs="0"/>
             <xsd:element name="State" type="xsd:string"/>
             <xsd:element name="PostalCode" type="xsd:decimal"/>
             <xsd:element name="Country" type="xsd:string"/>
          </xsd:sequence>
       </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

对复合内容进行约束的目的，是为了让设计者可以对复合类型的内容模型和/或属性进行限制。清单 3.9 显示了 restriction 元素如何实现了这一目的。在该例中，派生类型 AustralianPostalAddress 包含了元素 Number、Street、State、PostalCode、Country，但是删去了 City 元素。因为 City 元素的两个属性 minOccurs 和 maxOccurs 的值都设置为 0，所以 City 元素就被删除了。

3. 多态性

对于基类型的元素，派生类型可以使用多态，这是 XML 模式最具吸引力的特征之一。这意味着，设计人员可以在实例文档中使用派生类型代替模式中规定的基类型。

清单 3.10 定义了 PurchaseOrder 类型，它是清单 3.6 中定义的 PurchaseOrder 类型的变体。该类型的 billingAddress 和 shippingAddress 元素都使用了基类型 Address。

清单 3.10 多态地定义类型

```
<!-- Uses the data type declarations from Listing 3.8 -->

  <xsd:complexType name="PurchaseOrder">
    <xsd:sequence>
      <xsd:element name="Name" minOccurs="1" maxOccurs="1">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string"/>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="shippingAddress" type="PO:Address"
                                   minOccurs= "1" maxOccurs="1"/>
      <xsd:element name="billingAddress" type="PO:Address"
                                   minOccurs= "1" maxOccurs="1"/>
        <xsd:choice minOccurs="1" maxOccurs="1">
            <xsd:element name="BillingDate" type="xsd:date"/>
            <xsd:element name="ShippingDate" type="xsd:date"/>
        </xsd:choice>
    </xsd:sequence>
</xsd:complexType>
```

因为 XML 模式支持多态性，实例文档对于它的 billingAddress 和 shippingAddress 元素可以使用任何源自基类型 Address 的派生类型。在清单 3.11 中，PurchaseOrder 使用了派生类型 AustralianAddress 作为它的 billingAddress 元素，并且使用了派生类型 AustralianPostalAddress 作为它的 shippingAddress 元素。

3.4.2 导入模式与包含模式

W3C XML 模式具有强大的跨域复用能力。W3C XML 模式可以提供跨域复用，意味着模式（或子模式）可以表示一个可重复的样式，可以在不同的环境中使用这个样式，并且不同的应用都可使用它。具体的实现方式就是将模块化的 W3C XML 模式定义为外部子模式。当被组合时，

这些模块就会为文档模块提供完整的框架。这种方式使得开发者们可以复用模式组件及其他开发者的模式，从而可以减小模式开发的复杂性，并使得开发、测试和维护更便利。

清单 3.11 在 XML 模式实例中使用多态性

```
<!-- Uses type declarations from Listing 3.10 -->

<?xml version="1.0" encoding="UTF-8"?>
<PO:PurchaseOrder xmlns:
                PO="http://www.plastics_supply.com/PurchaseOrder">

    <Name> Plastic Products </Name>
    <shippingAddress xsi:type="PO:AustralianAddress">
      <Number> 459 </Number>
      <Street> Wickham st. </Street>
      <City> Fortitude Valley </City>
      <State> QLD </State>
      <PostalCode> 4006 </PostalCode>
      <Country> Australia </country>
    </shippingAddress>

    <billingAddress xsi:type="PO:AustralianAddress">
      <Number> 158 </Number>
      <Street> Edward st. </Street>
      <State> QLD </State>
      <PostalCode> 4000 </PostalCode>
      <Country> Australia </Country>
    </billingAddress>
      <BillingDate> 2002-09-15 </BillingDate>
</PO:PurchaseOrder>
```

在 XSD 中，使用 include 元素和 import 元素可以实现模式的组合。通过使用这两个元素，我们可以有效地“继承”被引用的模式中的属性和元素。

1. 包含模式

include 元素可在模式文档中包含其他的模式文档，只要这两个模式文档具有相同的目标命名空间，从而实现模式文档的模块化。针对一个具体的命名空间上下文，include 语法提供了独特的作用。当模式变得很大并难以管理时，这种方式就变得很有用。假如这样的话，需要将模式划分为单个的子模式(模块)，然后可使用 include 元素将它们组合起来。在清单 3.12 的声明中，Customer 类型的模式文档与清单 3.6 所描述的订购单模式具有相同的目标命名空间。我们假设 ProductType 类型也是这样，即它的模式文档与订购单模式具有相同的目标命名空间(见清单的阴影部分)。

清单 3.12 customer 子模式样例

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:PO="http://www.plastics_supply.com/PurchaseOrder"
  targetNamespace="http://www.plastics_supply.com/PurchaseOrder">

  <xsd:complexType name="Customer">
      <xsd:sequence>
        <xsd:element name="Name" minOccurs="1" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string"/>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Address" type="PO:AddressType"
```

```
                                    minOccurs= "1" maxOccurs="1"/>
            <xsd:choice minOccurs="1" maxOccurs="1">
            <xsd:element name="BillingDate"  type="xsd:date"/>
            <xsd:element name="ShippingDate" type="xsd:date"/>
          </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

在订购单模式的上下文中，可以使用 include 元素组合这两个子模式，如清单 3.13 中的阴影部分的两条语句所示。

在清单 3.13 中，由于两个被包含的子模式的命名空间将与订购单模式的命名空间一致，因此我们并不需要为这两个被包含的模式指定命名空间。

2. 导入模式

当我们需要使用属于不同的命名空间的模式模块时，需要使用 import 元素。通过 import 元素，XML 解析器可以引用其他命名空间的组件。import 元素与 include 有两点主要的不同[Walmsley 2002]。首先，include 元素仅可以在相同的命名空间中使用，而 import 元素可以跨不同的命名空间使用。其次，更微妙的差别在于它们的目的不同。include 元素具体地引入其他的模式文档，而 import 元素依赖另一个命名空间，但并不需要另一个模式文档。设计人员通过 import 机制将多个模式组合成一个更大的、更复杂的模式。当模式的一些部分(如地址类型)可以复用，并且需要它们的命名空间和模式时，import 机制变得很有用。

清单 3.13 在订购单模式中使用 include 元素

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:PO="http://www.plastics_supply.com/PurchaseOrder"
 targetNamespace="http://www.plastics_supply.com/PurchaseOrder">

  <xsd:include
    schemaLocation="http://www.plastics_supply.com/customerType.
    xsd"/>

  <xsd:include
    schemaLocation="http://www.plastics_supply.com/productType.
    xsd"/>

  <xsd:element name="PurchaseOrder" type="PO:PurchaseOrderType"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:all>
      <xsd:element name="ShippingInformation" type="PO:Customer"
                                    minOccurs="1" maxOccurs="1"/>
      <xsd:element name="BillingInformation" type="PO:Customer"
                                    minOccurs="1" maxOccurs="1"/>
      <xsd:element name="Order" type="PO:OrderType" minOccurs="1"
                                    maxOccurs="1"/>
    </xsd:all>
  </xsd:complexType>

  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="Street"          type="xsd:string"/>
      <xsd:element name="City"            type="xsd:string"/>
      <xsd:element name="State"           type="xsd:string"/>
      <xsd:element name="PostalCode "     type="xsd:decimal"/>
    <xsd:sequence>
```

```
  </xsd:complexType>

  <xsd:complexType name="OrderType">
    <xsd:sequence>
      <xsd:element name="Product" type="PO:ProductType"
                          minOccurs= "1" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="Total">
      <xsd:simpleType>
        <xsd:restriction base="xsd:decimal">
          <xsd:fractionDigits value="2"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="ItemsSold" type="xsd:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>
```

清单 3.14 地址标记模式

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:addr=http://www.plastics_supply.com/NewAddress
 targetNamespace="http://www.plastics_supply.com/NewAddress">
 <xsd:import namespace="http://www.plastics_supply.com/Address"
             schemaLocation="addressType.xsd"/>


  <xsd:complexType name="AddressType" abstract="true">
    <xsd:sequence>
      <xsd:element name="Number" type="xsd:decimal"/>
      <xsd:element name="Street" type="xsd:string"/>
      <xsd:element name="City"   type="xsd:string" minOccurs="0"/>
    <xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="AustralianAddress">
    <xsd:complexContent>
      <xsd:extension base="addr:AddressType">
        <xsd:sequence>
           <xsd:element name="State"        type="xsd:string"/>
           <xsd:element name="PostalCode "  type="xsd:decimal"/>
           <xsd:element name="Country"      type="xsd:string"/>
        <xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
  </xsd:complextype>

  <xsd:complexType name="AustralianPostalAddress">
    < xsd:complexContent>
      <xsd:restriction base="addr:AusttralianAddress">
        <xsd:sequence>
           <xsd:element name="Number"       type="xsd:decimal"/>
           <xsd:element name="Street"       type="xsd:string"/>
           <xsd:element name="State"        type="xsd:string"/>
           <xsd:element name="PostalCode "  type="xsd:decimal">
           <xsd:element name="Country"      type="xsd:string"/>
        </xsd:sequence>
      </xsd:restriction>
   </xsd:complexContent>
```

```
  </xsd:complextype>
</xsd:schema>
```

在订购单例子中，对于所有与地址相关的类型，清单 3.14 定义了一个单独的模式和命名空间。订购单包含了所有与地址相关的元素，如 AddressType、AustralianAddress、EuropeanAddress、USAddress、AustralianPostalAddress、EuropeanPostalAddress 等。这个模式对于订购单定义了一个完整的地址标记语言。对于所有的地址标记模式，这个命名属性是“http: //www. plastics_supply. com/Address”。这是一个单独的命名空间，与其他的订购单元素的命名空间不同。

由于订购单需要用到 AddressType 类型，因此我们需要将地址标记模式导入到订购单模式，如清单 3.15 所示。

清单 3.15 显示了 import 和 include 元素的使用，它们一起出现在定购单模式定义文档的最顶端。在清单 3.15 中，模式文档包含了订购单的地址标记语言，import 语句引用了模式文档的命名空间和位置。导入的命名空间在使用之前，需要给该命名空间指定一个前缀。通过使用前缀，复合类型 Customer 中的地址元素的声明将能引用 AddressType 类型。为了简洁性起见，我们可通过 include 元素将复合类型 Customer 的定义作为订购单模式的一部分，而无须像清单 3.12 那样在一个单独的子模式文档中定义 Customer 类型。

清单 3.15　使用 import 和 include 语句的订购单模式

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 targetNamespace=http://www.plastics_supply.com/PurchaseOrder
 xmlns:PO="http://www.plastics_supply.com/PurchaseOrder"
 xmlns:addr="http://www.plastics_supply.com/Address">

  <xsd:include
    schemaLocation="http://www.plastics_supply.com/
    productType.xsd"/>

  <xsd:import namespace="http://www.plastics_supply.com/Address"
    schemaLocation="http://www.plastics_supply.com/
    addressType.xsd"/>

  <xsd:element name="PurchaseOrder" type="PO:PurchaseOrderType"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:all>
      <xsd:element name="ShippingInformation" type="PO:Customer"
                                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="BillingInformation" type="PO:Customer"
                                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="Order" type="OrderType" minOccurs= "1"
                                   maxOccurs="1"/>
    </xsd:all>
  </xsd:complexType>

  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element name="Name" minOccurs="1" maxOccurs="1">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string"/>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="Address" type="addr:AddressType"
                                  minOccurs= "1" maxOccurs="1"/>
      <xsd:choice minOccurs="1" maxOccurs="1">
            <xsd:element name="BillingDate"  type="xsd:date"/>
            <xsd:element name="ShippingDate" type="xsd:date"/>
      </xsd:choice>
```

```
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="OrderType">
    <xsd:sequence>
      <xsd:element name="Product" type="PO:ProductType"
                                  maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="Total">
      <xsd:simpleType>
        <xsd:restriction base="xsd:decimal">
            <xsd:fractionDigits value="2"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="ItemsSold" type="xsd:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>
```

3.5 文档的导航与转换

与 HTML 语言相比，XML 主要用于描述和存储数据。虽然 XML 最明显、最有效的作用是用于描述数据，然而有些技术，诸如可扩展样式表语言转换(XSLT)等，可对 XML 的内容进行格式化或转换，从而将这些内容展现给用户。对于那些需要直接展现给个人查看的 XML 事务，需要通过 XSLT 进行样式表转换。XSLT 将 XML 格式转换为诸如 HTML 这样的表示技术，或者转换成其他所需的格式或结构。

XSLT 使用 XML 路径语言(XPath，定义为 W3C 的一个单独的规范)寻址和定位 XML 文档的各部分[Gardner 2002]。XPath 用于创建表达式的标准，所创建的表达式可用于发现 XML 文档中的具体信息。

3.5.1 XML 路径语言

XPath 数据模型将文档视为一颗节点树。节点对应于文档组件，如元素、属性等。通常将 XML 文档看作为包含根、分枝和叶的树。树本质上具有层次性，正如 XML 文档的层次性。

XPath 使用一个系统性的分类来描述 XML 文档的层次标记，以及对孩子、后代、父母和祖先的引用[Goldfarb 2001]。父母是一个包含其他元素的元素。祖先的元素列表包含它的父母。在父母之前的所有节点集形成了一个从该元素到根的有向路径。后代列表包含了元素的孩子，从该元素到叶节点也形成了一条有向路径。XPath 中的最顶端节点称为根或文档根。根并不是一个元素，它是一个容纳所有 XML 文档的逻辑构成。XML 文档实例中的所有其他元素都是根元素的孩子或后代。根元素就是根孩子本身。因为根元素是文档中的第一个元素，所以根元素也称为文档元素，并且根元素包含文档中的所有其他元素。

图 3.2 举例说明了清单 3.7 中定义的实例文档的部分逻辑(XPath 树)结构。请注意，在图中根元素是 Purchase Order。属性和命名空间直接与节点(见虚线)关联。属性和命名空间并不是元素的子女。文档节点的顺序基于 XML 实例的树形层次结构。元素节点位于它们的孩子之前(它们通过实线相连)，因此第一个元素节点是文档元素，紧接着的都是她的后代。一个特定元素(正如常规的树形结构中的元素)的孩子节点在兄弟节点之前被处理。最终，特定元素的属性和命名空间位于该元素的孩子之前。

清单 3.7 中的代码提供了 XML 结构的一个很好的基准样本。我们可以使用这个 XML 结构来定义 XPath 样例。清单 3.16 表示了一个 XPath 查询样本，以及相应的查询结果，查询结果表现为节点集。

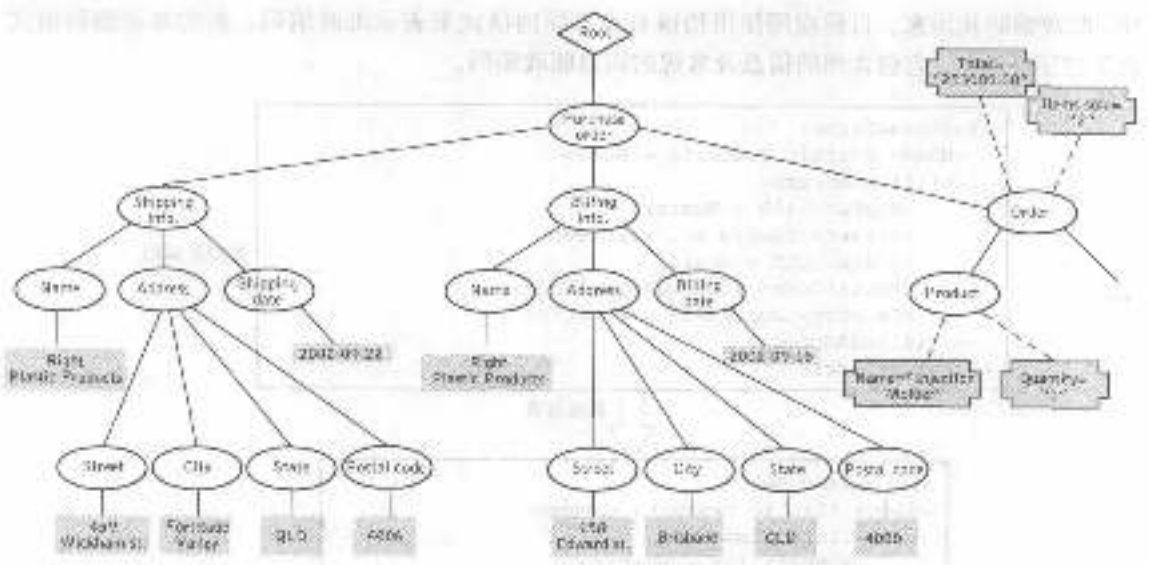


图 3.2 清单 3.7 中的实例文档的 XPath 树模型

清单 3.16 XPath 查询及相应的节点集

```
XPath Query#1: /PurchaseOrder/Order[2]/child::*

Resulting Node Set#1:
---------------------
    <Product Name="Adjustable Workstable" Price="3000.00"
     Quantity="1"/>
```

清单 3.16 中的 XPath 查询由三个定位步骤组成。第一个定位步骤是 PurchaseOrder。第二个定位步骤是 Order[2]，它指定了 PurchaseOrder 中的第二个 Order 元素。最后，第三个定位步骤是 child::*，它从第二个 Order 元素中选择所有的孩子元素。每一个定位步骤有一个不同的上下文节点，理解这一点非常重要。对于第一个定位步骤(PurchaseOrder)，当前的上下文节点是 XML 文档的根。第二个定位步骤(Order[2])的上下文是节点 PurchaseOrder。第三个定位步骤的上下文是第二个 Order 节点(没有在图 3.2 中显示)。

在一些相关书籍中，如[Gardner 2002]和[Schmelzer 2002]中可以查找到更多有关 XPath 的信息及 XPath 查询样例。

3.5.2 使用 XSLT 进行文档转换

为了进行文档转换，文档开发人员通常需要提供大量的样式表，这些样式表是使用 XSLT 编写的。样式表指定了如何显示 XML 数据。XSLT 在样式表中使用格式化指令进行转换。被转换的文档可以是另一个 XML 文档，或者是另一种格式的文档，诸如可以在浏览器中显示的 HTML。诸如 XSLT 这样的格式化语言仅能访问文档结构所定义的文档元素，例如 XML 模式。

XSLT 样式表或脚本包含了一些指令，这些指令将告诉转换处理器如何将源文档转换为目标文档。对于业务应用，XSLT 转换非常有用。例如，对于企业内部生成和使用的 XML 文档，可能需要将其转换为客户或企业服务提供者更熟悉的等价格式。这将使得企业合作伙伴之间可以更容易地相互转换信息。

图 3.3 显示了一个进行转换的例子，该图显示了 XML 文档的一个片段。这个片段表示了订购单消息的账单元素。如该消息所示，源 XML 应用使用一些单独的元素来表示门牌、街道地址、

州、邮政编码和国家。目标应用使用稍微有点不同的格式来表示邮政编码。新的邮政编码格式由7位字符表示，它包含州的信息及常规的四位邮政编码。

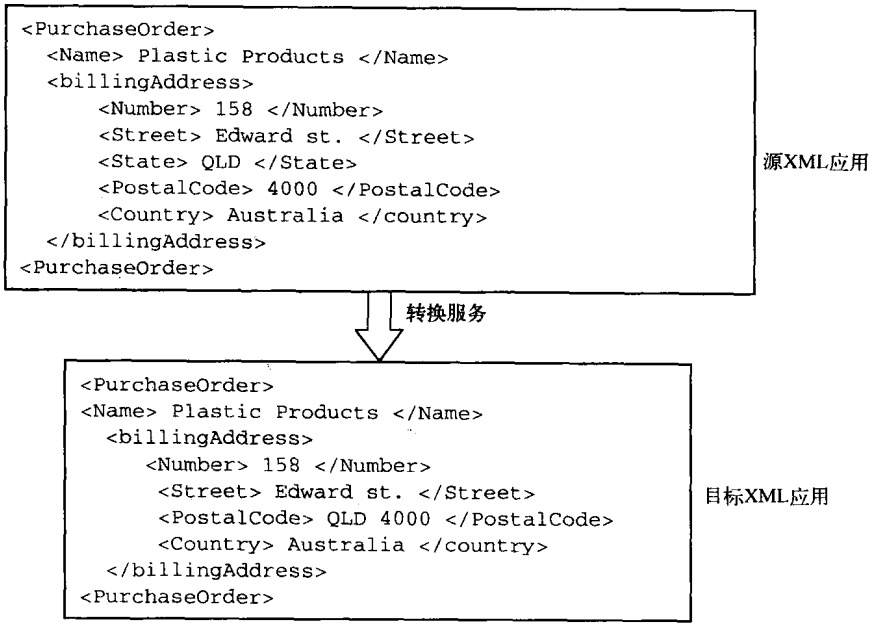


图3.3 使用XSLT转换与业务相关的信息

在一些相关书籍中，如[Gardner 2002]和[Tennison 2001]中，可以查找到更多有关XSLT的信息及一些样例。

3.6 小结

XML是一个可扩展的标记语言，用于Web上被标记电子文本的描述和传送。XML强调的是描述性的标记，而不是规定性的(过程化的)标记，这是XML的一个重要特性。XML的重要特性还包括文档类型概念、XML的可扩展性和可移植性。出于一些具体的目标需求，需要对XML文档进行处理，例如对XML文档进行格式化。在XML中，用于处理文档的指令与实际的XML文档中的描述性标记截然不同。由于XML使用描述性标记取代过程性标记，同一个文档可以使用许多不同的方式进行处理，并可仅使用文档中的相关部分。

XML的一个重要方面就是它的文档类型。XML文档具有不同的类型。XML文档的组成部分和它们的结构形式地定义了文档的类型。XML模式描述了元素和属性，这些元素和属性可以包含在符合模式的文档中，并且元素可以安排在文档结构中。因为模式能够清晰地确定存储在XML文档中的数据类型，所以在验证XML文档时，模式将发挥重要的作用。

可将XML看作一种动态交易语言，它能在不同的应用间灵活地、高效地交换信息。XML允许标签的内含体(inclusion)与数据的上下文含义相关。这些标签可帮助实现数据的机器解释。对于交易企业间的信息交换，标签可以优化处理流程。在XML中，不仅可以复用XML模式，而且可以改进其他模式结构的数据模型，从而可以实现组件的复用和扩展，减少开发周期，提高互操作性。

XML可对复杂的业务信息进行编码，因此XML非常适合于支持开放标准。若要实现业务信息交换的快速构建和互操作性，支持开放标准这一点是非常重要的。例如，异构、异步、开放和

分布式体系结构都建立在开放标准的技术之上(如解析器和接口),XML 非常适合这些环境下的事务处理。对于企业应用集成以及交易伙伴间的电子商务集成,XML 技术也是举足轻重的。

XML 能对复杂数据结构进行建模,复用 XML 模式,并能对其他的模式体系结构进行精化,从而可以实现组件的复用和扩展,减少开发周期,提高互操作性。XML 技术对于 Web Service 技术的开发具有深远的影响。对于 Web Service 和面向服务的体系结构,XML 技术还提供了基础性的构建块。

复习题

- 与其他标记语言相比,XML 最重要的两个特点是什么?
- 什么是 XML 元素?什么是属性?试举出 XML 元素和 XML 属性的具体例子。
- 使用具体的例子描述 URI 和 XML 命名空间。
- XML 模式定义语言的意图是什么?
- 列出并描述主要的 XML 模式组件。
- 什么是简单的 XML 类型?什么是复合 XML 类型?
- 如何在 XML 中实现可复用性?
- 试举出派生的复合类型的具体例子。
- 定义并描述 XML 中的多态性。
- 在 XML 模式定义语言中,include 元素和 import 元素的目的是什么?它们的不同点是什么?
- XPath 数据模型的目的是什么?XPath 数据模型是如何查看 XML 文档的?
- XSLT 是如何帮助进行文档转换的?

练习

3.1 假设有一个网上杂货店,为其定义一个简单的订购单模式。每一个订购单都应该包含不同的项。模式应该允许顾客接收货物的运送,以及允许由不同的人(如配偶)付款。文档应包含支付方式,并且需要允许顾客能够使用不同的支付方式,如信用卡、直接借记、支票等,并且文档需要包含所订购的产品的具体信息,如产品的价格、订购的数量等。

3.2 扩展上一道题中的订购单模式。和上一道题一样,网上杂货店向顾客出售产品,然而现在仅接受信用卡这一支付方式。简单的订单处理事务将包含基本的顾客、订单和产品类型信息以及不同的送货方式。信用卡支付包括信用卡号、有效期、支付金额。如何在订购单模式中导入练习 3.1 中开发的模式元素?

3.3 假设有一个简单的票据交换所应用,客户是电子化的商家。票据交换所向客户提供信用卡处理和(基于个人身份号码的)借记卡处理。为了处理信用卡,商家首先需要在票据交换所开设一个商家账户。商家账户是一个商业银行账户,它是根据商家和票据交换所之间的合同协议开始的。通过商家账户,商家可以接受顾客的信用卡支付方式。商家账户需要进行授权交易。信用卡的主要响应包括授权、拒绝或取消。在票据交换所处理信用卡销售时,仅当授权信用卡销售时,它将返回一个事务标识符(TransID)。当商家允许赊欠或取消事务时,则需要原来的信用卡销售的 TransID。为了简化,假设每一笔销售都允许信用卡,并且赊欠金额不能超过原来的销售金额。在一个交易中,该应用能够将大量的支付作为一个批处理任务进行处理。试为该应用定义一个模式。

3.4 假设有一个预订航班的应用。预订信息包括启航城市、目的城市、具体日期、旅客的

具体数量和类型，其他可选信息则包括时间或时间段、转机城市、旅客偏好(航空公司、航班类型等)。预订可以进一步细化到某一具体的航班、具体的航空公司或者具体航班上的舱位等级。试为该应用定义一个模式。

3.5 定义一个模式，用于处理交通工具的租赁预订。假设顾客已经决定了选择哪一个具体的租车分部。当进行车辆租赁时，需要定义所需的所有信息。模式中包含费用代码、费用类型、促销种类等信息，以及先前的回复中已经提供的费用信息和可能会影响费用的打折情况或促销代码。例如，顾客也许是老主顾，因此会有一个和预订相关的老主顾号。费用通常分为休闲价格或企业价格。这个模式应该定义时段，以及和某一特定价格(例如租车的单位时间内是否限制里程数)相关的距离、顾客对车辆类型的偏好、车辆所包含的特殊装备。

3.6 为一个简单的旅馆预订应用定义一个模式。这个旅馆预订应用根据一些具体的标准选择能够预订到的旅馆。选择标准可以包括：日期、日期范围、价格范围、房间类型、挂牌价格和特价、服务、娱乐设施。进行预订的可以不是用于住宿的房间，可以预订宴会厅、会议室。最终可以针对事件或房间进行预订。模式应当允许根据旅馆发布的“静态”信息(如旅馆设施、娱乐设施、服务等)和“动态”信息(如费用)进行预订。例如，旅馆可以有一个 AAA 价格、公司价格(不在所有时段提供)，或者可以根据协商情况来确定一个协商价。

第三部分　核心功能与标准

第4章　SOAP：简单对象访问协议

学习目标

常规的分布式对象通信协议，比如 CORBA、DCOM、Java/RMI，以及其他的进行服务器间通信的应用间的通信协议，当使用它们来行客户/服务器通信对，存在一些致命的弱点。当客户端分布在互联网上时，问题尤其显著。常规的分布式通信协议有一个对称的需求：通信连接的两端需要使用同一个分布式对象模型来实现，并且需要开发一些共用的库进行部署。为了解决这些制约，就开发了简单对象访问协议(SOAP)。SOAP 有助于在各类程序和平台之间实现互操作性，从而使更多的用户可以访问这些已有的应用。

本章介绍了 SOAP，描述了它的主要特点和 SOAP 消息的结构，并主要围绕下列主题：

- SOAP 作为消息协议的使用。
- SOAP 如何增强互操作性。
- SOAP 消息的结构。
- RPC 和文档样式消息。
- SOAP 中的错误处理。
- SOAP 使用 HTTP 作为传输协议。
- SOAP 的优点与不足之处。

4.1　应用程序间的通信与连接协议

随着 Web Service 技术的出现，企业将可以利用主流应用开发工具和互联网应用服务器来进行应用程序间的通信。企业可以更快、更便宜地提供更多的可用服务，从而推动业务的电子化。假如在异构的基础设施上运行的一些专有系统的问题被解决，业务电子化之能成功进行。值得注意的是，直到最近都缺乏连接这些系统的工具和共同的协定。

为了解决在异构基础设施上运行的专有系统的问题，Web Service 需要依赖 SOAP。SOAP 是一个基于 XML 的通信协议，它在两个计算机之间交换消息，而无须考虑这两个计算机的操作系统、编程环境或对象模型框架。SOAP 原先是简单对象访问协议(Simple Object Access Protocol)的首字母缩写，现在它仅是一个名字了。SOAP 是 Web Service 消息传输协议的事实上的标准。SOAP 的主要应用是应用程序间的通信。当使用 HTTP 作为请求和响应参数时，SOAP 使用 XML 作为编码模式。SOAP 方法是一个遵循 SOAP 编码规则的 HTTP 请求和响应。SOAP 端点是一个基于 HTTP 的 URL，该 URL 标识了方法调用的目标对象。

即使 SOAP 最初是一个“对象”访问协议，SOAP 也不像 CORBA 那样要求任何面向对象的方式。SOAP 只是定义了一个模型，该模型使用 XML 编写的简单请求和回复消息作为基本的通信协议。

SOAP 是一个连线协议，当跨互联网交换与服务相关的消息时，SOAP 规定了对这些消息如何结构化。在分布式环境中，如互联网或者 LAN(局域网)，可以将 SOAP 定义为一个轻量级的连线协议，用于在不同系统间交换结构化和类型化信息，从而实现远程方法调用[Cauldwell 2001]。“轻量级连线协议”这一术语意味着 SOAP 仅仅具有两个基本特性。SOAP 能接收或发送 HTTP(或其他的)传输协议包，并可处理 XML 消息[Scribner 2000]。SOAP 与诸如 IIOP 这样的分布式对象体系结构协议形成了鲜明的对比。互联网内部对象请求代理协议(Internet Inter-ORB Protocol，IIOP)是 CORBA 使用的连线协议。若要使用 IIOP，除了管理特定的应用需求，还必须安装合适的运行时环境，并且用户必须对它们的系统进行配置，以便能够适应分布式基础设施以及管理这些系统[Scribner 2000]。

许多人可能不太了解连线协议和传输协议之间的不同之处。连线协议指定在不同的应用、系统之间交换的数据的形式或状态，而传输协议是一种在系统之间传输数据的方法。传输协议负责将有效载荷从源端传送到目的端。

SOAP 作为连接表示

SOAP 指定连接协议时使用开放技术。在系统之间，SOAP 通常使用 HTTP 来传输进行了 XML 编码的串行化方法变量数据。通常在系统的远端使用串行化变量数据来执行客户端方法，而不是在本地系统上执行客户端方法。假如使用 HTTP 作为 SOAP 传输协议，则完全针对互联网的无状态编程模型来进行 SOAP 处理。开放的 XML 编码形式和 HTTP 的广泛使用，使得 SOAP 很可能成为最具有互操作性的连线协议[Scribner 2000]。

诸如 SOAP 这样的连线协议，在设计时都需要满足一些具体的标准，包括[Scribner 2002]简洁性、协议效率、耦合性、可伸缩性和互操作性：

*简洁性*是指当传送相同的信息时，网络数据包是否比较简洁。程度适度的简明性通常是最佳的。

*协议效率*直接与简洁性相关。根据发送有效载荷的开销，可疑判定效率的高低。所需要的传输开销越大，则协议的效率就越低。

*耦合性*是指客户端应用对于变化的适配程度。松耦合的协议通常比较灵活，可以比较容易地适应变化，而紧耦合的协议通常需要在客户端和服务器端进行很大的修改。

*可伸缩性*是指协议可以潜在地支持大量的使用者。有些协议只能支持几百个客户端，而有些具有可伸缩性的协议则可很轻松地支持数百万用户。

*互操作性*是指协议可以与大量不同的计算平台进行协作。例如，客户端可以使用通用的协议，向不同的系统发送信息。

通常基于这些特征来综合评价包括 XML 和 SOAP 在内的协议。没有哪一个协议在所有方面都非常出色。例如，XML 和 SOAP 都是松耦合的，并都支持互操作性，然而这反过来将影响简洁性和效率。作为基于文档的协议，XML 和 SOAP 是非常冗长的，从而影响了这两个协议的效率。由于 SOAP 通常使用 HTTP，因此 SOAP 本质上具有很强的可伸缩性。SOAP 的可伸缩性远远超过分布式对象体系结构协议。

4.2 SOAP 作为消息传送协议

不同的分布式计算平台具有异构性，SOAP 的目标就是试图消除这一异构性所产生的各种障

碍。与其他成功的 Web 协议一样，SOAP 也是通过下列几方面来实现它的目标的：简单性、灵活性、防火墙的友好性、平台中立性以及基于 XML 的消息传送(基于文本)。对于 Web 上的分布式通信的标准化实现，SOAP 仅是简单地使用了已有的互联网技术，SOAP 本身并不是一个新的技术。

SOAP XML 文档实例被称为 SOAP 消息(或 SOAP 信封)，并且它也经常用于传送其他网络协议的有效载荷。正如前面所述，交换 SOAP 消息的最常见的方式是通过 HTTP。Web 浏览器使用 HTTP 协议来访问 HTML Web 页面。HTTP 是发送和接收 SOAP 消息的常规方式。

SOAP 的目的很明确：在网络上交换数据。SOAP 尤其关注封装、XML 数据的编码以及定义数据发送和接收的规则[Monson-Haefel 2004]。简而言之，SOAP 是一个在服务实例之间传送消息的网络应用协议，而这些服务实例是使用 WSDL 进行描述的。如图 4.1 所示，SOAP 消息使用诸如 HTTP 等不同的协议来传送消息，并使用这些协议来定位与 Web Service 关联的远程系统。SOAP 描述了如何将消息格式化，但是并没有规定如何传送消息，因此必须将消息嵌入在传输层协议中。HTTP 是最常使用的传输层协议，然而也可以使用其他协议，诸如 SMTP、FTP 或 RMI 等。

如图 4.1 所示，在协议层次中，SOAP 协议的下一层是 HTTP 协议。SOAP 消息作为 HTTP 消息体被送到目的地，而 HTTP 消息则作为 TCP 流数据通过连接进行发送。在另一端(目的地)，HTTP 侦听程序将 HTTP 消息体传送给 SOAP 处理器。SOAP 处理器能够理解 SOAP 消息的语法，并有能力处理它所收到的消息。对此，下面我们将要详细说明。根本上，SOAP 是一个无状态的单向消息交换协议。但是底层协议可以提供一些特性，通过将一些单向交换与这些特性及特定的应用信息结合起来，可以产生比较复杂的交互模式(例如请求/响应、请求/多路响应等)。SOAP 本身并没有定义诸如编程模型等任何应用语义，也没有定义任何具体语义的实现。然而对于模块内的编码数据，SOAP 定义了一个简单的机制，可提供模块化的打包模型和编码机制。这使得 SOAP 可以使用在许多不同的系统中，诸如消息传送系统、RPC 等。SOAP 也不关心 SOAP 消息的路由、可靠的消息传送、防火墙的穿越等问题。然而，SOAP 提供了一个框架，可以采取可扩展的方式传送具体的应用信息。此外，对于 SOAP 节点在收到 SOAP 消息后所应采取的动作，SOAP 也提供了一个完整的描述。

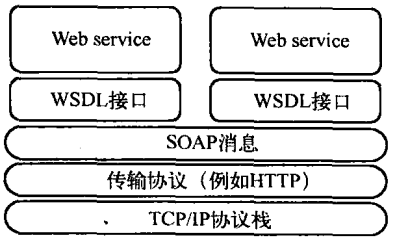


图 4.1 Web Service 通信和消息传送网络

在基于 XML 的消息传送分布式计算中，充当请求者或提供者的互联网节点的基本需求是：1)能够构造或解析 SOAP 消息；2)能够在网络上通过发送或接收消息进行通信。虽然 SOAP 可以使用不同的协议(如 HTTP、FTP 或 RMI)来传输消息，定位远程系统及初始化通信，然而最适合 SOAP 的传输协议还是 HTTP。基于 HTTP 的 SOAP 意味着，SOAP 消息将作为 HTTP 请求或响应的一部分进行发送，从而在支持 HTTP 的任何网络上都很容易地进行通信(参见 4.6 节)。SOAP 使用 HTTP 在系统之间传输 XML 编码的串行化方法变量。SOAP 的串行化机制使用专门的 XML 标签和语义，将方法调用转换为适合在网络上传输的形式。远端可以使用串行化变量数据来执行那个系统上的客户端方法调用，而不是执行本地系统上的客户端方法调用。因为 SOAP 可以驻留在 HTTP 上，所以 SOAP 的请求/响应操作非常类似于 HTTP。当客户端进行 HTTP 请求时，服务器将会试图对请求进行服务，并以两种方式之一进行响应。通过返回所请求的信息，任何一种响应通信都可成功进行。响应的另一种形式是返回出错信息，出错信息告知客户端不能进行服务请求的特定原因。

在两个交换消息的端点之间，SOAP 扮演了绑定机制的角色。SOAP 端点是一个简单的基于 HTTP 的 URL。该 URL 标识了方法调用的目标。SOAP 的作用就是进行灵活的绑定。例如，一个特定的 Web Service 可以提供两个绑定。客户端既可以使用 HTTP 也可以作为 E-mail 使用 SMTP 来提交 SOAP 请求。所使用的绑定类型并不会影响 SOAP 消息格式的设计，了解这一点很重要。

关于简单的 Web Service，SOAP 规定了单个消息交换的结构。然而，在许多情况下，业务处理或复合的 Web Service 需要交换多个消息。通常情况下，可以使用没有固定模式的会话方式实现这一点。关于谁发送了消息或者何时发送下一个消息，可通过检查所交换的 SOAP 消息体来进行确定。进行交互的 Web Service 之间需要交换 XML 文档，SOAP 体能够携带这些文档。在其他情况下，可能有一些切先确定的模式，诸如请求/响应模式。对于过程调用之类，请求/响应模式是一种最简单、最自然的模式。这种模式通常称为 RPC 风格的 SOAP。在分析 SOAP 消息结构后，我们将在 4.4 节中讨论 SOAP 通信方式。

在最近的版本中（SOAP 版本 1.2[Mitra 2003]），为了支持诸如 SMTP 或 FTP 这类新型传输协议，SOAP 中添加了一些清晰的规则，从而使得 SOAP 成为一个全面的交换协议。此外，SOAP 还引入了“中介（intermediary）”这一概念。“中介”位于 SOAP 调用和终点之间，用于信息路由，而不会影响消息的内容。终点这一概念意味着，可以沿着从中介到最终的接收者之间的链发送 SOAP 消息。有关中介的信息可在 SOAP 消息头中发现，而 SOAP 消息体中的信息则仅是发送给最终用户的。

使用 SOAP 的分布式应用处理可按照图 4.2 中的步骤进行。下面，将对这些步骤进行概述。

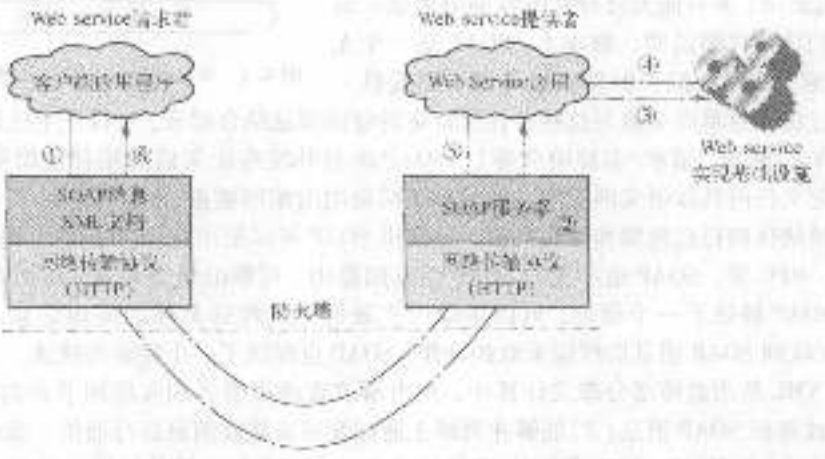


图 4.2 使用 SOAP 的分布式消息传送

为了调用驻留在远程服务提供者（1）中的 Web Service 操作，服务请求者的应用程序需要创建一个 SOAP 消息来运行请求。SOAP 客户端创建这一请求。SOAP 客户端是一个创建所需的 XML 文档的程序，所创建的 XML 文档包含远程调用分布式系统中的方法所需的信息。在 SOAP 请求体中的 XML 编码是所请求的方法的位置以及存放方法变量的位置。服务请求者将 SOAP 消息及提供者的 URI（通常基于 HTTP）一起转发给网络基础设施。

网络基础设施将消息发送给消息提供者的 SOAP 运行时系统（例如 SOAP 服务器）（2）。SOAP 服务器是一个专用的代码，可侦听 SOAP 消息，并可充当 SOAP 文档的分发器和解释器。SOAP 服务器将请求消息发送给服务提供者的 Web Service 实现代码（3）。基于 HTTP SOAP 还接可接收文

档。在服务提供者的节点上，实现 Web Service 的应用程序可能需要特定的编程语言对象。SOAP 服务器确保将所收到的文档从 XML 转换为特定的编程语言对象。SOAP 消息信封中的编码模式将决定如何进行转换。在这种情况下，SOAP 服务器也确保将包含在 SOAP 文档中的参数传送给 Web Service 实现基础设施中的合适方法。

服务请求者的 URI 给出了提供者的节点作为请求目的地(4)。Web Service 负责处理请求，并将响应表示为 SOAP 消息。SOAP 响应消息将提交给提供者节点上的 SOAP 运行时系统。SOAP 服务器通过网络将 SOAP 响应消息转发给服务请求者。

在服务请求者节点上的网络基础设施接收响应消息。网络基础设施可将 XML 响应转换为源(服务请求者)应用程序(6)所能理解的对象。

Web Service 可以使用单向消息传送或者请求/响应消息传送。在单向消息传送中，SOAP 消息沿着一个方向传送，从发送者到接收者。而在请求/响应消息传送中，SOAP 消息从发送者传送给接收者，然而接收者将返回一个响应给发送者。SOAP 支持各种消息交换模式，其中之一就是请求/响应模式。其他的例子还包括要求/响应(solicit/response，请求/响应颠倒过来的模式)、通知、连续的对等模式的会话。图 4.3 显示了一个简单的单向消息，发送者并不接收响应。然而，接收者能够将响应传送给发送者，如图 4.4 所示。

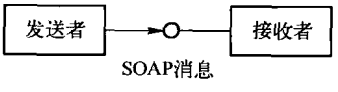


图 4.3 单向消息传递

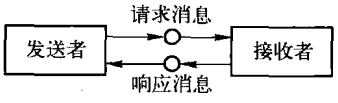


图 4.4 请求/响应消息传送交换模式

4.3 SOAP 消息的结构

当前的 SOAP 规范 v1.2 描述了如何将关联的 XML 模式中定义的数据类型进行 HTTP(或其他传输协议)上的串行化[Gudgin 2003]。为了正确地交换信息，SOAP 消息的提供者和请求者都必须访问相同的 XML 模式。通常在互联网上将模式进行公告，信息交换的任何一方都可从互联网上下载这些模式。SOAP 消息包含有效载荷。每一个 SOAP 消息本质上都是一个 XML 文档。

SOAP 消息包含一个 < Envelope > 元素，< Envelope > 元素必须包含一个 < Body > 元素，并可以包括也可以不包括一个 < Header > 元素[Gudgin 2003]。这些元素的内容是被定义的应用，虽然 SOAP 规范对于如何处理这些元素也有所涉及，但这些元素的内容并不属于 SOAP 规范。SOAP < Header > 元素包含一些信息块，这些信息块主要关于如何处理消息。对于 SOAP 消息中那些并不是应用有效载荷的信息，这提供了一种传递方式。这类“控制”信息包括传递指令或者与消息处理相关的上下文信息，例如路由与传送设置、认证或授权声明、事务上下文，从而可以在一个具体应用方式中扩充 SOAP 消息。< Header > 元素的直接的孩子元素称做“头块”，并表示为一个数据逻辑分组。将消息从发送者传送到最终的接收者的路径中有一些 SOAP 节点，这些数据逻辑分组可以描述这些 SOAP 节点。SOAP < Envelope > 是 SOAP 消息必须运载的主要的端到端信息。在 SOAP < Envelope > 中必须包含 SOAP < Body > 元素。SOAP 消息的结构如图 4.5 所示。

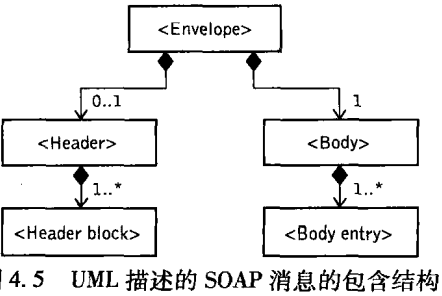


图 4.5 UML 描述的 SOAP 消息的包含结构

4.3.1 SOAP 信封

SOAP 的目的是提供一种在两个端点之间传输消息的统一方式。SOAP 信封用于包裹任何交换的 XML 文档。SOAP 信封还提供了一种扩大有效载荷的方式，可添加一些附加信息，这些附加信息可帮助将消息路由到最终目的地。SOAP 信封是每一个 SOAP 消息单一的根。对于遵循 SOAP 的消息，必须出现 SOAP 信封。< Envelope > 元素定义了框架，可用来描述在消息里有什么以及怎样处理这些消息。

清单 4.1 显示了 SOAP 消息的结构。如清单所示，在 SOAP 中，< Envelope > 元素是根元素，该元素可以包含(也可以不包含) < Header > 部分，但一定需要包含 < Body > 部分。假如使用了 < Header > 元素，则该元素必须是 < Envelope > 元素的直接子女，并且出现在 < Body > 元素之前。< Body > 元素定义了特定应用的数据。如清单 4.1 中的片段所示，SOAP 消息可以有一个 XML 声明，该声明可以指定所使用的 XML 的版本及编码格式。

清单 4.1 SOAP 消息的结构

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
   xmlns:env="http://www.w3.org/2003/05/soap-envelope">

   <env:Header> <!-- optional -->
    <!-- header blocks go here . . . -->
   </env:Header>

   <env:Body>
    <!-- payload or Fault element goes here . . . -->
   </env:Body>
</env:Envelope>
```

SOAP 信封中的所有元素都是使用 W3C XML 模式进行定义的。SOAP 消息利用了命名空间。在第 3 章中已经阐述了可以使用命名空间来区分具有类似名字的不同的元素和属性。从而使得这些具有类似名字的不同的元素和属性可以在同一个文档中共存，而且不会导致混乱。更重要的一点，命名空间使得 SOAP 消息具有可扩展性。通过引用不同的命名空间，SOAP 消息可以扩展它的语义范围，并且消息接收者可以引用同一个命名空间来解释新的消息[Schmelzer 2002]。

与通常的情况类似，也是使用关键字 xmlns 来声明 SOAP 命名空间。为了实际上将信封标签标识为属于 SOAP 命名空间，它必须包含对 SOAP 信封命名空间 URI 的引用。信封模式根据 SOAP 规范 v1.2(在本章中都使用这一版本)进行定位，URI 是“http: //www. w3. org/2003/05/soap-envelope”。在清单 4.1 中，命名空间标识符 env 不仅包括 < Envelope > 元素，而且包括 < Header > 部分和 < Body > 部分。假如 SOAP 应用接收了基于其他一些命名空间的消息，它将会报错。该规则确保所有符合标准的消息都精确地使用同一个命名空间和 XML 模式，所以也会采用相同的处理规则。

SOAP 信封可以规定编码规则集，这个规则集可将所定义的应用的 XML 数据进行串行化。提供者和请求者都必须遵循编码规则(通常可以下载定义这些规则的 XML 模式)。为了使得两个或多个通信方都遵循 XML 消息的某一个具体的编码样式，它们可以使用全局 encodingStyle 属性表示 SOAP 消息所采用的协定。对于所出现的元素及其子女元素，这个属性授权某一个编码样式(通常通过模式进行定义)。编码样式是一个规则集，这个规则集描述了系统中的各方如何表示(串行化为 XML)数据，从而实现互操作性。因为这个属性可以出现在 SOAP 消息的任何层次或者任何标签中，包括定制的 header 和 body 条目，所以该属性是“全局的”。为了具有更多的灵活性，SOAP 允许应用定义它们自己的编码样式。例如，假如两个应用在 SOAP 消息中使用数组作

为变量，则一个应用可以使用标签集将数组串行化为行的序列，而另一个应用则可以使用标签集将数组串行化为列的序列。必须使用 encodingStyle 属性标识所使用的规则集。在大多数情况下，都标示了标准的 SOAP 编码规则。有关 encodingStyle 属性的更详细的信息，参见 4.4 节。

SOAP 定义了一个数据模型，可用于表示任意的数据结构，例如整数、数组、记录，然后指定了转换规则，将这个模型的实例转换为 SOAP 消息中的 ASCII 字符串。通过模型，客户端和服务提供者可将客户端或提供者的表示语言中规定的数据类型实例映射到模型实例中。在这里，术语“客户端”与请求消息的最初的发送者相关。假定编码规则指定了每一个模型实例的串行化，每一个客户端及服务提供者的类型实例将映射到一个串行化字符串。客户端和服务提供者都必须提供进行映射的串行化机制和逆串行化机制。对于 SOAP 中定义的简单数据类型和结构化数据类型，为了实现针对这些数据类型的 SOAP 编码规则，许多供应商都提供了针对特定编程语言程序，诸如 Java 或 C#。

清单 4.2 SOAP 信封样例

```
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    env:encodingStyle="http://schemas.xmlsoap.org/soap/
                                                encoding/">
    ...
</env:Envelope>
```

SOAP 规范允许在信封标签中包含任意数量的附加的、自定义的属性。然而，每一个自定义属性都必须有合适的命名空间。这意味着，一个自定义的前缀必须关联到自定义属性的命名空间。对于自定义属性，必须使用通过 xmlns 声明的 URI 和前缀。清单 4.2 是 SOAP < Envelope > 元素语法的一个简单样例。

4.3.2 SOAP 头部

SOAP 将包含在 SOAP < Envelope > 中的任何信息分成两部分：< Header > 和 < Body >。< Envelope > 最多包含一个 < Header > 子元素。< Header > 元素中包含了与端点或中间传输点相关的所有处理线索。< Header > 元素中可以包含文档将要发往何处、文档源自哪里等信息，甚至还可以传送数字签名。这类信息必须与 SOAP < Body > 分开。SOAP < Body > 是 < Envelope > 必须要包含的部分，它包含了 SOAP 的有效载荷(XML 文档)。

< Header > 的目的就是对扩展的消息格式进行封装，且无须与有效载荷发生关联，也不需要修改 SOAP 的基本结构，因而 SOAP 消息可以在不违反规范的前提下不断添加新的特性和功能，如安全性、事务、对象引用、计费、QoS 属性等。此外，许多消息传送系统(异步、同步、RPC、单向等)中都可以使用 SOAP，并且这些消息传送系统可以按非传统的方式组合起来。Web Service 客户端可以在消息的头部存放扩展数据，从而使得服务中的每一个方法调用无须将那个数据作为变量。例如，可以使用头部信息向 Web Service 提供认证凭证，Web Service 上的每一个方法无须查询用户名和口令，这些信息可以包含在 SOAP < Header > 中。

SOAP < Body > 携带了有效载荷。SOAP < Header > 提供了一个机制，可提供描述有效载荷的更详细的信息。当 SOAP 消息中包含 < Header > 时，< Header > 必须是 SOAP < Envelope > 元素的第一个子元素。SOAP < Header > 元素的模式允许在头部放置数量不限的子元素。< Header > 元素的直接子元素称做“头块(Header Block)”，并表示为一个数据逻辑分组。将消息从发送者传送到最终接收者的路径中有一些 SOAP 节点，这些数据逻辑分组可以描述这些 SOAP 节点。在 < Header > 元素中，每一个头块都应当有它自己的命名空间。因为命名空间能够帮助 SOAP 应用标识头块以及分别处理这些头块，所以这是一个非常重要的议题。一些组织(如 W3C

和 OASIS 等）正在开发各种标准化的头块，这些标准化的头块涉及许多主题，诸如安全性、事务或其他的一些服务特性。被提议的每一个标准都定义了它自己的命名空间、XML 模式及处理需求。

清单 4.3 一个 SOAP Header 样例

```
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
    ...
    <env:Header>
        <tx:transaction-id
            xmlns:tx="http://www.transaction.com/transaction"
              env:mustUnderstand="true">
                 512
        </tx:transaction-id>
        <notary:token xmlns:notary="http://www.notarization-
                                        services.com/token"
              env:mustUnderstand="true">
                 GRAAL-5YF3
        </notary:token>
    </env:Header>
    ...
</env:Envelope>
```

清单 4.3 显示了一个 <Header> 元素样例，它包含两个头块。第一个头块处理有关付款单的事务完整性规则。第二个头块包括一个公证服务，这个公证服务将一个标记与一个特定的订购单关联起来，作为第三方保证订购单已经妥当处理，并且订购单已经包含了所要订购的特定产品。通过将扩展封装为 SOAP 消息的格式，每一个头块都将提供不同的可扩展性。

正如前面章节已经讨论的，SOAP 利用了 XML 命名空间的能力，并且 SOAP 定义了与 SOAP <Envelope> 元素的 URI 关联的协议版本，如 SOAP1.1，SOAP1.2。命名空间使得 SOAP 接收者能够处理不同版本的 SOAP 消息，而不会影响向后兼容性，对于特定的 SOAP 消息的每一个版本，也不需要不同的 Web Service 端点。头块的不同版本之间存在差异。例如，头块的不同版本可能影响接收者如何处理消息。因此通过命名空间标识头块版本，使得接收者可以变换不同的处理模型。假如接收端不支持所指定的版本，也可拒绝接收该消息。模块化使得 SOAP 消息的不同部分的处理互不干扰，并且不同部分可分别演变。例如，SOAP <Envelope> 或 <Header> 块都可以不断变化，然而 <Body> 元素中的具体应用内容的结构仍然保持不变。类似地，当 SOAP 消息和头块的版本不变时，具体应用的内容也可以发生变化。

SOAP 消息传送的模块性使得处理 SOAP 消息的代码也可以是模块化的［Monson-Haefel 2004］，处理 <Envelope> 元素的代码与处理 <Header> 块的代码无关，而这些代码又与处理 SOAP <Body> 元素中具体应用数据的代码无关。如图 4.6（该图基于［Monson-Haefel 2004］）所示，模块性使得开发人员可以使用不同的代码库来处理 SOAP 消息的不同部分。该图显示了 SOAP 消息和代码模块的结构，其中代码模块分别用于处理 SOAP 消息的各个不同部分。灰色框中的代码模块与当前 SOAP 消息所使用的命名空间关联，而白色框中的代码模块则表示了替代选择。这些替代选择与不同的命名空间关联，并可用于处理 SOAP 消息的其

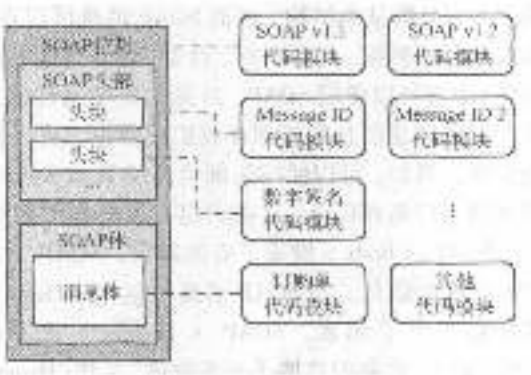


图 4.6 使用不同的代码库来处理 SOAP 消息的不同部分

他版本。

SOAP <Header>元素也提供了可扩充性。SOAP 可扩充性是指 SOAP 中可以增加特定服务所需的附加信息，而不需修改包含信息的消息体。SOAP 中可增加对特定服务的需求，诸如安全性需求。在调用方法之前，首先需要对请求者进行验证，或者方法必须要有事务属性。消息体则包含了接收者所需的信息，如调用信息。最后，<Header>元素可以包含任意数量的子元素，正如我们在前面章节已经阐明的。在图 4.6 中，<Header>元素能够包含一个指定了数字签名扩展的头块。

SOAP 中介

SOAP 头部可有一些不同的用途，许多头部涉及其他 SOAP 处理节点（称做 SOAP 中介）的参与。从最初的 SOAP 发送者（源点）到最终的 SOAP 接收者（最终的目的地）之间，形成了一条消息路径。

SOAP 消息沿着消息路径从发送者传送到接收者。所有 SOAP 消息的传送都起始于创建 SOAP 消息的最初发送者，终止于最终的接收者。图 4.7 显示了一个验证订购单的 SOAP 消息的消息路径，该 SOAP 消息是由顾客生成的。在图中，订购服务节点验证订购单是否确实是由某个特定顾客发送的。中介服务确认嵌入在 SOAP 消息中的顾客的数字签名头块是否真正有效。SOAP 消息将会被自动地路由到中介节点（签名确认服务）。中介节点从 SOAP 消息中抽取数字签名，对数字签名进行验证。无论数字签名是否有效，中介节点都会将新获悉的块传送给订购服务。

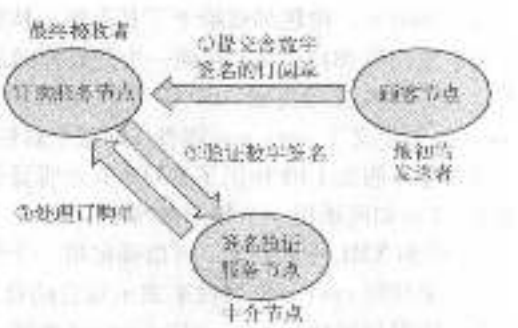


图 4.7 验证订购单的 SOAP 消息路径

中介既可以接收也可以转发 SOAP 消息。中介接收 SOAP 消息、处理一个或多个头块，然后将 SOAP 消息发送给另一个 SOAP 应用程序。由于 SOAP 消息沿着消息路径进行传送，因此路径上的任意数量的中介都可拦截并处理 SOAP 消息的头块。SOAP 消息路径上的 SOAP 节点，对于所遇到的 SOAP 消息头部，可以对它们进行检查、插入、删除或转发。沿着消息路径（最初的发送者、中介和最终的接收者）的应用，通常被称为 SOAP 节点。三个关键的应用实例定义了对 SOAP 中介的需求：交叉信任域、得到保证的可伸缩性，以及沿着消息路径提供增值服务。

下面我们将使用清单 4.4 来说明消息路径中的节点是如何处理消息头部的。在清单 4.4 中，<Header>元素包含两个头块，每一个头块都是在它自己的命名空间中定义的，并且每一个头块都涉及了 SOAP 消息体整个处理的某些方面。对于该订单处理应用，与整体需求相关的"元"信息是订单头块。订单头块提供了这个订单实例的订单号与时间戳，以及顾客块中的顾客标识。

清单 4.4 用于消息路由的头块样例

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
   xmlns:env="http://www.w3.org/2001/05/soap-envelope">
 <env:Header>
  <m:order
     xmlns:m="http://www.plastics_supply.com/purchase-order"
       env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
            env:mustUnderstand="true">
    <m:order-no >uuid:0411e3daa</m:order-no>
    <m:date>2004-11-8</m:date>
  </m:order>
```

```
<n:customer xmlns:n="http://www.supply.com/customers"
    env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
        env:mustUnderstand="true">
    <n:name> Marvin Sanders </n:name>
</n:customer >
</env:Header>
<env:Body>
    <-- Payload element goes here -->
</env:Body>
</env:Envelope>
```

在清单4.4中，消息路径中的下一个SOAP中介必须处理头块的订单和顾客。值为“http://www.w3.org/2003/05/soap-envelope/role/next”的属性env:role说明了：路由中遇到的下一个SOAP节点将会处理头块的订购单和顾客。这个角色也是所有SOAP节点都必须扮演的角色。这意味着，当处理消息时，SOAP节点能够假定一个或多个角色(role)可以影响到如何处理SOAP头块和<Body>。角色都被给予了具有唯一性的名字(名字为URI形式)，因此在处理时可以识别这些角色。当SOAP节点收到一个待处理的消息时，SOAP节点首先会确定它将承担的角色是什么。为了帮助进行决定，SOAP节点可以检查SOAP消息。SOAP可以定义一个可选属性env:role。假如定义了env:role属性，则这个属性将出现在头块中，它标识了头块的意向目标所要扮演的角色。假如URI标识了SOAP节点所要承担的角色，则该SOAP节点就需要处理头块。至于SOAP节点如何承担一个特定的角色，SOAP规范对此并没有规定。结合清单4.4中使用的env:role属性和XML命名空间，可以确定哪一个代码模块将要处理一个特定的头块。

头块使用env:role属性来表示对它的处理，SOAP节点在正确地识别头块后，将会使用头部元素中的附加属性env:mustUnderstand来确定所需的进一步的处理操作。头块对于实现应用宗旨非常重要，为了确保SOAP节点不会忽视需要处理的头块，SOAP头块也提供了一个可选的附加属性mustUnderstand。值为“true”的env:mustUnderstand属性意味着：处理头部的节点必须完全遵循规范来处理这些头块，否则将不处理这个消息并报告一个错误。例如在清单4.3中，头部表明了消息是正在处理的(假设的)事务的一部分。mustUnderstand属性表示了，假如客户想要使用事务，则提供者需要支持事务。假如提供者(消息接收者)不支持事务，接收消息后将会报错。称作mustUnderstand的专用属性是<Header>的子元素。

在设计应用时，需要选择哪些数据放在头块中，哪些数据放在SOAP体中。沿着从发送者到最终接收者的消息路径上的任何SOAP节点，都有可能会处理头块。比如，中介SOAP节点可以基于头部中的数据提供增值服务。

4.3.3 SOAP消息体

在消息交换中，具体应用的XML数据(有效载荷)存放在SOAP体中。SOAP消息必须包含<Body>元素，并且该元素必须是<Envelope>的直接后代。SOAP体可以包含任意数目的子元素，也可以为空。<Body>元素的直接子元素都必须有合适的命名空间。默认情况下，SOAP体的内容可以是任何XML，并且并不局限于任何专门的编码规则。SOAP体必须包含在信封中，并且必须位于消息中所定义的任何头部之后。

<Body>元素包含具体应用的数据或一个出错消息。具体应用的数据是与Web Service进行交换的信息。该数据可以是任何XML数据或者方法调用的参数。方法调用信息及相关的变量都编码在SOAP <Body>中。对方法调用的响应及错误信息也存放在SOAP体中。<Body>元素和根元素的一个区别是，它既是请求对象又是响应对象。仅当出现错误时，才使用出错消息。发现问题(诸如处理出错或者消息的结构不合适)的接收节点会将出错消息发回给消息路径上的前面的发送者。SOAP消息可以携带具体应用的数据或出错信息，但不可能同时携带这两类信息。在

下面的章节，将要介绍一些有关 <Body> 元素的实例。

4.4 SOAP 通信模型

Web Service 通信模型描述了如何调用 Web Service，描述了 Web Service 和 SOAP 的关系，并通过通信方式和编码方式定义了 SOAP 通信模型。SOAP 支持两类通信方式：RPC 和文档（消息）。SOAP 编码方式指的是如何对 SOAP 消息中的头块的特定元素及 <Body> 元素进行编码。我们首先将描述 SOAP 的编码方式，然而将比较详细地讨论 SOAP 的两类通信方式。

对于类型化对象的串行化，SOAP 定义了相关的编码规则（通常称为编码方式）。编码方式是关于不同平台下的应用如何分享和交换数据，这些应用甚至可能没有共同的数据类型或表示。编码规则的作用表现在两方面。首先，对于与所描述的类型系统一致的模式，可以构建针对 XML 语法的模式。其次，对于类型系统模式以及与符合那个模式的特定值，可以构建 XML 实例。反过来，对于按照规则生成的 XML 实例，以及原来的模式，可以构建原先的值。URI http://www.w3.org/2003/05/soap-encoding 标识了 SOAP 编码规则。通过使用 SOAP encodingStyle 属性，采用了特定串行化的 SOAP 消息应当表明那一点。使用 encodingStyle 属性可以定义特定的 SOAP 元素集的编码方式。encodingstyle 属性可以存放在文档的任何地方，并可应用到所在元素的所有子女中。

有四类 SOAP 通信方式：PRC/Literal、Document/Literal、RPC/Encoded 和 Document/Encoded。WS-I Basic Profile 1.0 仅允许使用 PRC/Literal 或 Document/Literal（将在下面两节描述），明确禁止使用 RPC/Encoded 和 Document/Encoded。

4.4.1 RPC 类型的 Web Service

RPC 类型的 Web Service 显现为客户端应用的一个远程对象。客户端和 RPC 类型的 Web Service 之间进行交互，这类交互主要围绕具体服务的接口。客户端将请求作为含有变量集的方法调用，返回的响应将包含返回值。这些都可以表示为如图 4.8 所示的嵌入在 SOAP 消息中的 XML 元素集。

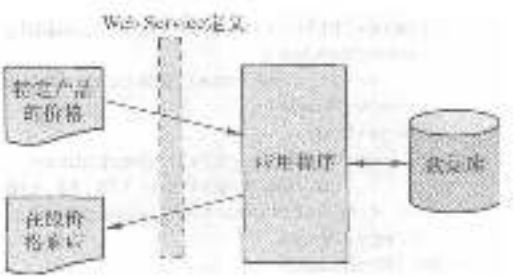


图 4.8 计算特定产品价格的 RPC 类型的 Web Service

RPC 支持消息的自动串行化/逆串行化，允许开发者将请求作为带有参数集的方法调用，返回的响应将包含返回值。关于在客户端和 Web Service 之间的通信是双向的，因此 Web Service 在客户端和服务提供者之间的通信模型需要紧耦合。

在 RPC/Literal 消息传送中，SOAP 消息能够发出带参数的方法调用，并获得返回值。使用 RPC/Literal 消息传送，可将传统的组件暴露为 Web Service，如 servlet、无状态会话 bean、Java RMI 对象，CORBA 对象或 DCOM 组件 [Monson-Haefel 2004]。这些组件并不直接交换 XML 数据，它们有带参数的方法，可以返回值。

在 SOAP 信封中，打包为 RPC/Literal 的规则非常简单：

- URI 标识了所需调用的传输地址。
- RPC 请求消息包含方法名和调用的输入参数，该个方法调用的编码格式化为单个结构，其中 in 或 in-out 参数表现为结构中的一个域。
- 名字和参数的物理顺序必须与所调用的方法的名字和参数的物理顺序一致。

- RPC 响应消息包含返回值和任何输出参数(或者出错消息)。响应结构被建模为一个结构,方法签名中的每一个参数建模为结构中的一个域。方法响应类似于响应结构中的方法调用。

清单 4.5 是 SOAP < Body > 规范的一个样例。它表示了一个报价服务,这个报价服务询问一个具体的塑料产品的价格。如清单所示,SOAP < Body > 元素包含实际的方法调用(作为它的第一个子元素)。消息命名空间标识符(m:)是方法调用所需的信息的一部分。命名空间标识了目标对象的 URI。此外,方法调用需要方法名(GetProductPrice)和参数(product-id)。

清单 4.5 RPC 类型的 SOAP 体样例

```
<env:Envelope
 xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
       <tx:Transaction-id
            xmlns:t="http://www.transaction.com/transactions"
            env:mustUnderstand='1'>
                512
       </tx:Transaction-id>
    </env:Header>
    <env:Body>
       <m:GetProductPrice>
          <product-id> 450R6OP </product-id >
       </m:GetProductPrice >
    </env:Body>
</env:Envelope>
```

一旦发出一个包含调用体(< Body > 元素中的方法元素和变量)的 SOAP 消息,紧接着就可能有一个响应消息。响应消息将包含一个 < Body > 元素,而 < Body > 元素中则包含远程方法调用的结果。清单 4.5 所示的报价请求的响应消息如清单 4.6 所示。

清单 4.6 SOAP RPC 响应消息的样例

```
<env:Envelope
 xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
       <--! - Optional context information -->
    </env:Header>
    <env:Body>
       <m:GetProductPriceResponse>
          <product-price> 134.32 </product-price>
       </m:GetProductPriceResponse>
    </env:Body>
</env:Envelope>
```

当传输 RPC 消息时,通过 HTTP 绑定可以将请求自动绑定到对应的响应上,这是 HTTP 绑定的一个很有用的特性。由于应用中客户端可能与多个提供者进行通信,因此这个特性对于应用很重要。在这种情况下,应用可能有几个请求,因此需要将抵达的响应与对应的请求关联起来。

4.4.2 文档(消息)类型的 Web Service

可以使用 SOAP 来交换文档,这些文档可以包含任何类型的 XML 数据。对于各种类型的系统,既可以是企业内的系统也可以是业务伙伴间的系统,可以将它们集成为一个透明的异构系统,从而可以完全复用代码。与其他的路由消息传送、分布式访问协议(例如 Java RMI 协议或 CORBA 协议)不同,SOAP 并不提供将源和目的地信息编码进信封的一些方式。而是由各个客户端决定将信息传送到哪里、如何传送信息。例如,在 Web Service 应用中,Web Service 基础设施

决定如何发送 SOAP 消息、将 SOAP 发送到哪里。在 SOAP 体中发送没有编码的 XML 内容通常称为文档型 SOAP。这种类型 SOAP 将消息视为 XML 文档，而不是作为编码为 XML 的抽象数据类型。在文档型 SOAP 应用中，对于 SOAP <Body> 元素的内容没有限制。

文档型 Web Service 属于消息驱动。如图 4.9 所示，当客户端调用一个消息类型的 Web Service 时，客户端通常发送如订购单等整个文档，而不是发送一些离散的参数集。Web Service 处理并发送整个文档，并且它既可以返回一个响应消息，也可以不返回一个响应消息。由于调用 Web Service 的客户端无须等待响应就可继续向下，因此它属于异步类型。假如 Web Service 有响应，该响应可以在调用之后的任何时候出现。与 RPC 类型不同，文档类型不支持消息的自动串行化/逆串行化，而是假定 SOAP 消息的内容是良构的。例如，描述订购单的一些 XML 元素嵌入在 SOAP 消息中，可将这些 XML 元素视为 XML 文档片段。事实上，作为本章样例的订购单 SOAP 消息就是 Document/Literal 消息。

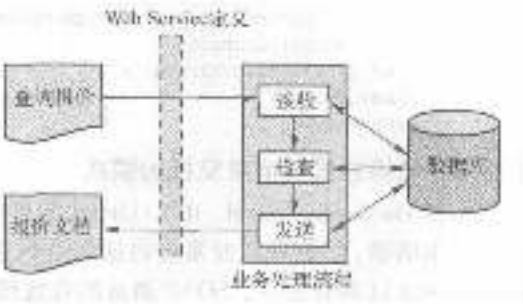


图 4.9　处理报价查询

在 Document/Literal 模式的消息传送中，SOAP <Body> 元素包含 XML 文档片段、良构的 XML 元素。这些良构的元素包含任意的应用数据，而这些应用数据属于与 SOAP 消息分离的 XML 模式和命名空间。<Body> 元素并不直接反映 XML 结构。SOAP 运行时环境接收 SOAP <Body> 元素，并将其毫无变化地传送到预定的应用中。既可以有一些响应也可以没有响应与该消息关联。

清单 4.7 显示了一个订购单 SOAP 消息。该 SOAP 消息包含了一个订购单文档片段，这个订购单文档片段为塑料产品制造公司订购了两个某种型号的注射模型成形机。对于诸如清单 4.6 所示的 RPC 类型的请求，这类应用数据上载方案并不很合适。取而代之的是，应用程序将应用数据传送给服务提供者进一步处理。若采用这种方案，文档型 SOAP 消息将携带整个应用数据，并将其作为一个简明的、自包含的 XML 文档（如订购单），这将是一种较好的选择。

清单 4.7　文档型 SOAP 体的具体样例

```
<env:Envelope
 xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope">

    <env:Header>
      <tx:Transaction-id
          xmlns:tx="http://www.transaction.com/transactions"
          env:mustUnderstand="1">
              512
    </env:Header>
    <env:Body>
      <po:PurchaseOrder orderDate="2004-12-02"
          xmlns:n="http://www.plasticsupply.com/POs">
          <po:from>
            <po:accountName> RightPlastics </po:accountName>
            <po:accountNumber> PSC-0343-33 </po:accountNumber>
          </po:from>
          <po:to>
            <po:supplierName> Plastic Supplies Inc.
                                                  </po:supplierName>
            <po:supplierAddress> Yara Valley Melbourne
```

```
                                                    </po:supplierAddress>
            </po:to>
            <po:product>
              <po:product-name> injection molder </po:product-name>
              <po:product-model> G-100T </po:product-model>
              <po:quantity> 2 </po:quantity>
            </po:product>
          </ po:PurchaseOrder >
        </env:Body>
    </env:Envelope>
```

4.4.3 通信方式与消息交换的模式

虽然 Document/Literal、RPC/Literal 消息传送方式与单向、请求/响应消息交换模式在概念上的区分很明确，但有些人经常会将这两组概念搞混淆。SOAP 通信方式可以是 Document/Literal 或 RPC/Literal 这两者之一。SOAP 消息的有效载荷通常是一个 XML 文档片段，或者是与 RPC 相关的参数、返回值的 XML 表示。与此相反，单向消息交换模式、请求/响应消息交换模式指的是消息的流向，与具体的内容无关。单向消息传送仅有一个方向，而请求/响应有两个方向。Document/Literal 方式的消息传送既可以采用单向消息传送的模式，也可以采用请求/响应消息传送的模式。RPC 方式的消息传送也既可以采用单向传送的模式，亦可以采用请求/响应消息传送的模式，不过通常采用请求/响应消息传送的模式。

4.5 SOAP 中的出错处理

SOAP 提供了一个出错处理模型。对于消息处理过程中所发生的错误，可使用该模型进行处理。SOAP 可以区分导致错误的原因，并可将出错信息告知消息的发送者或其他节点。出错信息可以放置在 SOAP < Body > 元素中。

在 SOAP 出错模型中，使用专门的元素 env：Fault 来报告所有的 SOAP 错误和应用错误。在处理 SOAP 消息时，可能会出现一些问题。SOAP 规范提供了可扩展的机制，可以用于传输有关这些问题的结构化信息和非结构化信息。env：Fault 元素是 SOAP 规范中预定义的保留元素。

清单 4.5 表示了一个处理 RPC 的故障。清单 4.8 显示了一个 SOAP 消息，该消息是清单 4.5 中的 RPC 请求所返回的响应。如清单 4.8 所示，env：Fault 元素包含在 < Body > 元素中。env：Fault 元素必须包含两个子元素 env：Code 和 env：Reason，并可以有（可选择）env：Detail 子元素，用于存放具体的应用信息。在清单 4.8 中，env：Fault 的直接子元素是 env：Code 元素。env：Code 元素也包含两个元素，其中一个是必须包含的元素，称为 env：Value，另一个元素则是可选的，称为 env：Subcode。env：Value 元素包含大量的标准化的出错代码（值）。这些出错代码具有 XML 限定名（QNames），每一个限定名标识了一种错误。值 env：Sender 表示消息发送者没有正确地生成消息（语法错误），或者消息缺少信息（缺少参数或认证信息）。此外，命名空间 http：//www. plastics_supply. com/product - prices 定义了 InvalidPurchaseOrder 错误，env：Subcode 元素则表示了导致处理请求时出现故障的原因为 InvalidPurchaseOrder 错误。env：Reason 子元素包含了一个人可读取的描述，该描述说明了出错情况。最后，env：Detail 元素存放具体应用的出错信息。这一出错信息通常并不能为所有的 SOAP 节点所理解，仅有了解生成出错信息的具体应用的那些节点才能理解这些出错信息。

清单 4.8 SOAP 消息的出错样例

```
<env:Envelope
 xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
```

```
        <tx:Transaction-id
            xmlns:t="http://www.transaction.com/transactions"
            env:mustUnderstand='1'>
                512
        </tx:Transaction-id>
    </env:Header>
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:Sender</env:Value>
                <env:Subcode>
                    <env:Value> m:InvalidPurchaseOrder </env:Value>
                </env:Subcode>
            </env:Code>
            <env:Reason>
                <env:Text xml:lang="en-UK"> Specified product
                 did not exist </env:Text>
            </env:Reason>
            <env:Detail>
              <err:myFaultDetails
                 xmlns:err="http://www.plastics_supply.com/
                 faults">
                <err:message> Product number contains invalid
                 characters
                </err:message>
                <err:errorcode> 129 </err:errorcode>
              </err:myFaultDetails>
            </env:Detail>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

4.6 基于 HTTP 的 SOAP

SOAP 使用 XML 作为请求和响应参数的编码模式，并且使用 HTTP 作为传输协议来抵达互联网上的任何目的地，而无须任何额外的包裹或编码。特别地，当 SOAP 端点是一个基于 HTTP 的 URL 时，该 URL 标识了方法调用的目标，SOAP 方法可以是一个遵循 SOAP 编码规则的 HTTP 请求或响应。SOAP 并不需要将一个具体的对象束缚于一个特定的端点，而是由实现者决定如何将对象端点标识符映射到一个提供者端的对象上。

使用 HTTP POST 方法可将 HTTP 和 SOAP 请求/响应消息交换模式进行绑定，下面我们将简要地分析这一概念。对于所有的应用，无论这些应用是否是通常的 XML 数据交换或封装在 SOAP 消息中的 RPC，都可使用 SOAP HTTP 绑定中的消息交换模式。在 HTTP POST 方法的体中，可以传输 SOAP 请求。HTTP POST 方法可在 HTTP 请求消息体中传送请求内容。通过 POST 方法，SOAP 信封将变为 HTTP 请求消息的数据部分。在 HTTP 响应中，将会返回 SOAP 响应(参见图 4.10)。

HTTP POST 操作将消息 POST 到一些服务提供者。清单 4.5 中的 HTTP POST 操作描述了订购单 SOAP RPC 请求消息，清单 4.9 中的代码片段举例说明了订购单 SOAP RPC 请求消息的使用。当使用 RPC 型的 SOAP 格式化 SOAP 消息(例如清单 4.9 中的消息)时，将会调用一个过程，并且会生成一个结果，这个结果将会在 SOAP 响应消息中返回。在 HTTP 响应的数据部分将携带 SOAP 响应(参见清单 4.10)。

清单 4.9 封装的 HTTP/SOAP 请求样例

```
POST /Purchase Order HTTP/1.1
Host: http://www.plastics_supply.com  <! - Service provider -- >
```

```
Content-Type:application/soap+xml;
charset = "utf-8"
Content-Length: nnnn

<?xml version="1.0" ?>
<env:Envelope
 xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
 xmlns:m="http://www.plastics_supply.com/product-prices">
    <env:Header>
       <tx:Transaction-id
           xmlns:t="http://www.transaction.com/transactions"
           env:mustUnderstand='1'>
              512
       </tx:Transaction-id>
    </env:Header>
    <env:Body>
       <m:GetProductPrice>
          <product-id>  450R6OP  </product-id >
       </m:GetProductPrice >
    </env:Body>
</env:Envelope>
```

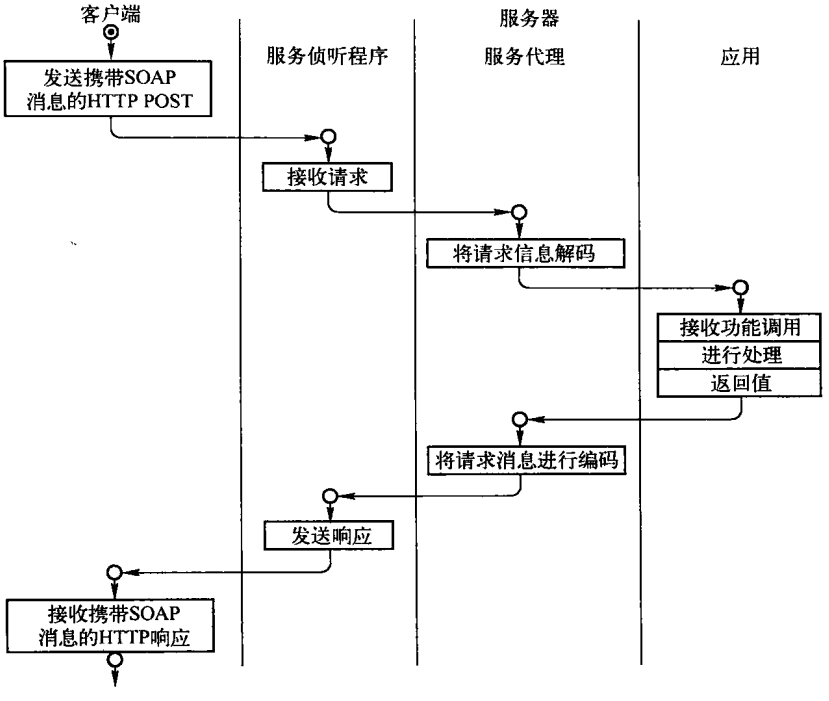


图 4.10 SOAP HTTP POST 方法的使用过程

清单 4.10 订购单服务应用所发送的 RPC 返回结果

```
HTTP/1.1 200 OK
Content-Type:application/soap+xml;
charset = "utf-8"
Content-Length: nnnn

<?xml version="1.0" ?>
<env:Envelope
 xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"
```

```
xmlns:m="http://www.plastics_supply.com/product-prices">
   <env:Header>
      <--! - Optional context information -->
   </env:Header>
   <env:Body>
      <m:GetProductPriceResponse>
         <product-price> 134.32 </product-price>
      </m:GetProductPriceResponse>
   </env:Body>
</env:Envelope>
```

清单 4.10 显示了订购单应用服务所发送的 RPC 返回值。与清单 4.6 的请求相对应的 HTTP 响应中包含了这一返回值。当 SOAP 使用 HTTP 进行通信时，它将根据 HTTP 状态代码的语义来处理 HTTP 中的通信状态信息。例如，200 系列的 HTTP 状态代码表示已经成功地收到、理解和接受客户端的请求。

对于服务开发来说，理解 SOAP 的基本原理很重要。大多数 Web Service 开发人员并不需要直接与基础架构打交道。大多数 Web Service 都使用 WSDL 生成的优化后的 SOAP 绑定。通过这种方法，SOAP 实现能够自我配置 Web Servcie 间的交换，从而屏蔽了许多技术细节。

4.7 SOAP 的优缺点

正如任何其他的协议一样，使用 SOAP 协议具有许多方面的优点，同时也存在一些缺点。SOAP 的主要优点可概括如下。

- 简单性：由于 SOAP 所基于的 XML 具有高度的结构化，并且很容易解析，因此 SOAP 比较简单。
- 可移植性：SOAP 无须依赖底层平台，不需要考虑诸如字节顺序或计算机字长这类问题，因此 SOAP 具有可移植性。目前，在从主机到嵌入式设备等任何平台上，都可解析 XML。
- 与防火墙的相容性：在 HTTP 上 POST 数据不仅意味着传送机制具有广泛的可用性，而且意味着 SOAP 可以穿过防火墙，从而避免了其他一些方法可能引发的问题。
- 使用开放标准：SOAP 使用 XML 开放标准来格式化数据，从而使得 SOAP 很容易扩展并能得到很好的支持。
- 互操作性：SOAP 是建立在开放技术之上的，而不是基于特定厂商的技术，从而有利于实现真正的分布式互操作性和松耦合的应用。因为 SOAP 是一个基于 XML 和 HTTP 的连线协议，所以 SOAP 可能是到目前为止最具互操作性的协议，并且可以使用该协议来描述自动化技术环境和高度复杂性的应用中的消息交换。
- 被广泛接受：在消息通信领域，SOAP 是最被接受的标准。
- 适应变化：除非在 SOAP 规范中对串行化进行很大的改动，否则 SOAP 基础架构的变化通常不会影响到使用该协议的应用。

然而，SOAP 也存在几个方面的缺点，具体如下[Scribner 2000]。

- SOAP 最初仅限于 HTTP，因此采用了并不适合所有情况的请求/应答体系结构。HTTP 是一个相当慢的协议，从而影响了 SOAP 的性能。SOAP 规范的最新版本放松了这方面的要求。
- SOAP 是无状态的。由于 SOAP 的无状态特性，当需要较多的连接时，发送请求的应用程序需要向其他应用重复表明它的身份，就像以前从未连接过一样。然而在一些业务处理和事务中，多个 Web Service 之间需要相互交互，这时就需要维护连接的状态。
- SOAP 为基于值的串行化，而不支持基于引用的串行化。基于值的串行化需要对象的多个复制，这些对象久而久之会包含一些状态信息，然而由于复制对象并不在原先的位置，因

此无法在这些对象中同步这些状态信息。这意味着，当前 SOAP 尚不能(以对象引用的形式)引用或指向一些外部数据源。

4.8 小结

SOAP 是分散式、分布式环境中的一个轻量级的信息交换协议。SOAP 定义了一个简单的、可扩展的 XML 消息传送框架。该框架可以基于多个不同的协议及许多不同的编程模型。SOAP 消息实际上是发送到网络中的一些端点的服务请求，那些端点可以采用不同方式实现，如远程过程调用(RPC)服务器、组件对象模型(COM)对象、Java Servlet、Perl 脚本等。并且这些端点可以在不同的平台上运行。因此，为了便于实现高度分布的应用，SOAP 规定了一个连线协议。许多应用可能在不同的平台上运行，使用不同的实现技术及不同的编程语言，SOAP 支持这些应用间的互操作性。

SOAP 本质上是一个无状态的、单向的消息交换方式。然而将单向交换与底层协议所提供的特性及具体应用信息结合起来，应用可以创建比较复杂的交换模式(例如，请求/响应、请求/多路响应等)。对于所传输的具体应用数据所涉及的一些问题，如 SOAP 消息的路由、可靠的数据传输、防火墙的穿透等，SOAP 都没有特殊的限制。SOAP 提供了一个框架，能够以可扩展的方式传送具体应用信息。同样，对于 SOAP 节点收到 SOAP 消息后所需采取的动作，SOAP 也提供了完整的描述。

SOAP 定义了一个完整的处理模型。这个处理模型规定了，当消息在传输路径上进行传输时，应该如何处理这些消息。总的来说，SOAP 提供了一个可用来定义高层应用协议的、丰富灵活的框架，所定义的协议可以增强分布式、异构环境中的互操作性。

SOAP 包含三部分。信封提供了添加附加信息、扩充有效载荷的机制。为了将消息路由到最终目的地，需要用到这些附加信息。< Header > 元素包含了与端点或中间传输节点相关的所有处理信息。< Header > 元素还提供了一个 SOAP 扩展接口，能够以任何方式指定安全性、事务或其他约定。最后，< Body > 元素携带与具体应用所交换的 XML 数据。

复习题

- 什么是连线协议？SOAP 的意图是什么？
- 无状态单向交换协议的作用是什么？
- SOAP 是如何处理分布式应用的？
- SOAP 是如何与 WSDL 协作的？
- SOAP 的两种最常用的消息传送方式是什么？
- 列举并描述 SOAP 消息中的元素。
- SOAP 中是如何实现模块化的？
- 举例说明 SOAP 中介，并解释它的工作机理。
- 描述两类 SOAP 通信模型。
- SOAP 和 HTTP 是如何协作的？
- SOAP 是如何实现互操作性的？
- SOAP 是如何实现串行化的？

练习

4.1 根据练习 3.4 的 XML 模式，编写一个返回有关商务航班信息的简单的 SOAP 程序。

4.2 选择特定旅行线路的旅行者需要进行预订，并使用信用卡进行支付。针对这一应用，编写一个简单的 SOAP 程序。

4.3 编写一个简单的 SOAP 程序，该程序需要返回特定股票的交易代码。

4.4 假设已经编写了一个简单的 Java 方法，该方法检查订单的状态。使用变量（“ZRA56782C”，“Mega Electronics Ltd.”）调用签名 checkOrderStatus(String OrderNumber, String company ID)。编写一个嵌入在 SOAP 消息中并能够实现同一结果的简单方法。

4.5 对于练习 4.2 中的应用，编写一个 SOAP 消息。假如客户的信用卡号是无效的，该 SOAP 消息可以拒绝该信用卡的支付。

4.6 编写一个简单的 SOAP 程序，该程序：

a）使用 RPC 类型的消息来检查特定产品的库存。该消息将接受两个变量：产品标识符和装运的产品数量。

b）将 SOAP RPC 类型的消息转换为等价的 SOAP 文档类型的消息。

第5章　描述 Web Service

学习目标

在前面的章节中，我们已经阐述了如何使用 SOAP 对在任何两个 Web Service 之间交换的 XML 数据进行封装。然而，SOAP 并没有描述 Web Service 的功能特性，也没有描述如何在交互的服务间交换数据。因此，SOAP 服务需要文档详述被暴露的服务操作以及这些操作的参数。服务描述语言将针对这一问题。服务描述语言是一个基于 XML 的语言，它描述了和特定的 Web Service 进行交互的机制。本质上，服务描述语言规定了一个“约定”。该“约定”将控制请求者和服务方之间的交互，并且“约定”中没有仅与任何一方相关的信息，诸如内部实现细节。

本章主要讨论 Web Service 描述语言(WSDL)，这是一个描述 Web Service 的 XML 语言。在本章中，我们主要围绕 WSDL 版本 1.1 展开讨论。该标准是由 W3C 制定的，已被广泛接受，并已得到许多厂商的支持。将在第 9 章讨论的业务流程执行语言(BPEL)也是基于 WSDL 版本 1.1。

读者阅读本章后将理解下列主要概念：

- 为何需要 Web Service 描述语言。
- Web Service 描述语言。
- Web Service 接口定义和 Web Service 实现这两者之间的差异。
- 在 WSDL 中定义 Web Service 接口和实现。
- WSDL 消息交换协定。
- WSDL 如何提供非功能性服务特性。

5.1　为何需要服务描述

为了开发基于服务的应用和业务处理(可能包含多个服务)，需要以一致的方式来描述 Web Service。在这种方式中，服务提供者出版服务，服务客户端和开发者发现服务。可将多个服务装配为一个可管理的复合服务层次，对服务进行编配从而实现增值服务解决方案和复杂应用。然而，为了做到这一点，用户必须确定 Web Service 的精确的 XML 接口以及其他的一些消息细节。在 Web Service 领域中，对于 Web Service 所获悉的 XML 消息，由于 XML 模式使得开发者可以描述 XML 消息的结构，因此 XML 模式可以部分地满足上面的这一需求。和 Web Service 进行通信时，需要涉及许多重要的通信细节，如服务的功能性特性和非功能性特性(参见 1.8 节)或者服务策略，然而 XML 模式并不能独自描述这些细节。

对于实现 SOA 松耦合，将服务提供者和服务请求者的应用集成在一起，减少定制程序的开发以及更好地理解相关知识，服务描述都是一个重要方面。服务描述是一个机器可理解的规范，它描述了 Web Service 的结构、操作特性和非功能性特性。服务描述还规定了 Web Service 所使用的连线格式和传输协议。服务描述也能使用类型系统描述载荷数据。通过服务描述以及底层的 SOAP 基础架构，无论对于服务请求者的应用还是服务提供者的 Web Service，都可以有效地屏蔽所有的技术细节，例如具体的计算机、实现和编程语言。尤其是，虽然在标准化的服务描述语言中规定了约定，服务请求者端需要遵循这些约定，然而对它们的具体实现并没有任何限制。在本书的后面，我们将会发现：服务描述也可以包括元数据、行为属性以及策略描述(参见第 13 章和第 15 章)。

5.2 WSDL：Web Service 描述语言

在一个特定的 Web Service 中使用 SOAP 将需要用到一些文档。这些文档以计算机所理解的标准格式说明 SOAP 消息的结构、所用的协议（例如 HTTP 或 SMTP）、所暴露的操作和它们的参数以及 Web Service 的互联网地址。WSDL 使得 Web Service 提供者以及这些服务的用户之间可以更容易地协作，从而更容易地实现 SOAP 所能带来的好处。WSDL 是一个服务描述语言，可用于描述 Web Service 所暴露的所有接口的详细信息。因此，WSDL 是一种访问 Web Service 的方法。通过服务描述，服务提供者可以采用各种规范调用服务请求者所需的 Web Service。例如，服务请求者和提供者都不会关心对方技术基础架构、编程语言、分布式对象框架（假如有的话）。虽然 WSDL 也能表示对 SOAP 之外协议的绑定，然而在本章中我们主要讨论与基于 HTTP 的 SOAP 相关的 WSDL。

WSDL 是一个基于 XML 的规范模式，可用于描述 Web Service 的公共接口。公共接口可以包括与 Web Service 相关的操作信息，诸如所有公开可用的操作、Web Service 支持的 XML 消息协议、消息的数据类型信息、具体使用的传输协议的绑定信息、Web Service 的地址信息等。WSDL 支持 XML 文档的 SOAP 传输规范。虽然并没有从 Web Service（或者出版该服务的服务提供者）的观点表述 WSDL 中的 Web Service 描述，然而 Web Service 描述仍然是用于约束服务提供者以及使用服务的请求者。这意味着，WSDL 表示了服务请求者和服务提供者之间的"约定"，正如面向对象的编程语言（如 Java）中的接口表示了客户端代码和实际对象本身之间的"约定"一样，主要的不同点在于：1）WSDL 具有平台独立性和语言独立性；2）WSDL 主要用于（但不是仅用于）描述基于 SOAP 的服务。图 5.1 对 WSDL 服务描述进行了说明。因此，Web Service 描述仅关心那些同时涉及双方的信息，而不关心仅与其中一方相关的信息，诸如任何一方的内部实现细节。本质上，WSDL 用于精确地描述诸如"Web Service 做什么（例如服务提供的操作）"、"服务驻留在哪里（例如 URL 等具体协议的地址信息）"、"如何进行调用（如数据格式的详细信息、访问服务操作所需的协议的详细信息）"之类的问题。

Web 请求可以基于不同的协议，如 SOAP。在 WSDL 中，服务提供者能够描述 Web 请求的基本格式或者编码（例如多用途网际邮件扩充协议 MIME）。通过其他命名空间中的元素，大多数 WSDL 元素都可扩展。对于 SOAP、HTTP GET/POST 操作以及 MIME 附件，语言规范进一步定义了它们的标准扩展。可扩展性是一个非常有用的特性，可以使用扩展元素来指定一些具体技术的绑定。通过扩展，无须修改基础的 WSDL 规范即可改进网络和消息协议。

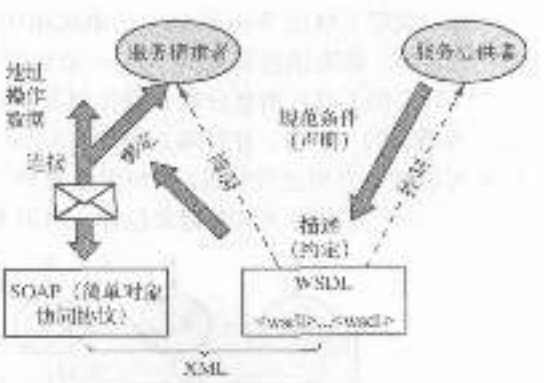


图 5.1 控制服务请求者和服务提供者之间交互的 WSDL 服务描述

可以简单地使用 WSDL 来描述操作。这样做时，并不会表示这些操作的执行顺序，那是 BPEL 的功能。在第 9 章中，我们将要讨论 BPEL。

WSDL 是一个非常有用和灵活的语言。因为 WSDL 基于 XML，所以能够通过模式以及灵活的命名空间来验证实例文档。WSDL 文档可以分成不同的部分，从而隔离了变量类型、操作接口、消息编码、传输格式和服务位置。WSDL 规范事实上分成两部分：服务接口定义（抽象接口）和服务实现（具体端点）。这样可以分别地、独立地定义和复用各个部分：

- 服务接口定义描述了通用的 Web Service 接口定义的结构。服务接口定义包含服务所支持

的所有操作、操作参数和抽象数据类型。

- 服务实现部分将抽象接口绑定到具体的网络地址、具体的协议和具体的数据结构。Web Service 客户端可以绑定到一个具体的实现并调用服务。

服务接口定义和服务实现定义这两部分合起来构成了服务的 WSDL 规范。这两部分组合后包含了大量的信息，这些信息描述了服务请求者如何在服务提供者的节点上调用 Web Service，以及服务请求者如何与 Web Service 交互。使用 WSDL，请求者可以定位 Web Service，并调用任何可公开访问的操作。通过 WSDL 工具，整个处理过程可以完全自动化，从而使得应用可以很容易地集成新的服务，而且无须改动很多代码，甚至完全不需要改动代码。假如服务请求者的环境支持 Web Service 的自动发现，则服务请求者的应用可自动指向服务提供者的 WSDL 定义，并可为所发现的 Web Service 定义自动生成代理。由于无须构造复杂的调用，因此服务请求者的应用简化了 Web Service 的调用，并节约了大量的代码开发工作。

在 WSDL 开发中，涉及几个设计方面的选择问题：1)平台和语言的独立性；2)对于多协议、多编码模式和可扩充性的支持；3)对于消息和 RPC 构建的统一支持；4)操作的无顺序性。其中第一点是最重要的问题之一，我们已经在前面分析了，对于后面几点，我们将依次进行简要的分析。

5.2.1 WSDL 的接口定义

服务客户端通过调用操作与 Web Service 进行交互。在 Web Service 接口中，可以将相关的操作进行分组。客户端不仅需要知道 Web Service 的接口和它所包含的操作，也必须知道使用何种协议将消息发送到服务中，并需要知道所使用的协议的具体机制，如命令、头部和出错代码的使用。一个实际的绑定指定了通过连线上所传送的具体细节，以及如何将抽象的消息映射到特定的网络层通信协议。对于消息传送以及同步的 RPC 类型的约定，WSDL 都是技术基础。通过指定服务类型和编码机制(文本或编码)，绑定也可以影响抽象消息的编码方式。对于一个具体的接口，服务可以支持多个绑定，但是每一个绑定都需通过一个主 URI 标识的唯一地址进行访问，这通常也称为服务端点。

WSDL 指定了描述 Web Service 的语法和句法，可将 Web Service 描述为通信端点的集合。可参见 7.2.1 节，那里描述了 Web Service 资源的寻址模式。在分层中，WSDL 位于 XML 模式的顶部。WSDL 提供了可将消息分组到操作以及将操作分组到接口的方法。在端点之间交换的数据被指定为消息的一部分，并且端点所允许的每一类处理活动都被视为一个操作。端点所允许的操作集可按端口类型进行分组。WSDL 也提供了将接口、协议和端点地址进行绑定的方法，如图 5.2 所示。一个完整的 WSDL 定义包含了调用 Web Service 所需的全部信息。

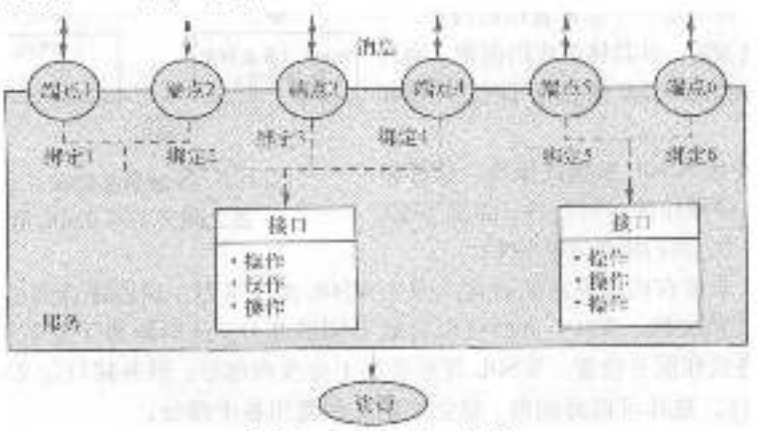


图 5.2 Web Service 端点

Web Service 接口定义描述了消息、操作和端口类型，并且具体的描述保持了平台独立性和语言独立性。Web Service 接口定义被视为 Web Service 的抽象定义，它并不携带任何具体的部署信息。可以使用 Web Service 接口定义来描述服务所提供的具体的接口类型。Web Service 接口定义精确地描述了需要发送的消息的类型，以及如何应用各种 Internet 标准消息传送协议和编码模式，以便用与服务提供者的规范相兼容的方式格式化消息。服务接口定义是一种抽象的服务描述，可通过多个具体的服务实现加以实例化和引用。这使得多个服务实现者可以定义和实现常见的、产业标准化的服务类型。

在 WSDL 文档中，< types >、< message >、< part >、< portType > 和 < operation > 元素描述了 Web Service 的抽象接口。< portType > 元素本质上是一个抽象接口（类似于 Java 接口定义），由 < operation > 和 < message > 定义组成。每一个 < message > 定义描述了消息的有效载荷，这些消息既可以是由 Web Service 向外发送的消息，也可以是它所接受的消息。消息由 < part > 元素组成，每一个 < part > 元素表示了一个类型（类型化参数）的实例。通过 < portType > 元素可以声明 < operation > 元素。每一个 < operation > 元素都包含了许多 < message > 定义，这些 < message > 定义描述了它的输入输出参数以及任何出错情况。下面将要描述 Web Service 的抽象接口，如图 5.3 所示。图 5.3 使用 UML 描述了 WSDL 接口定义数据结构之间的关系。

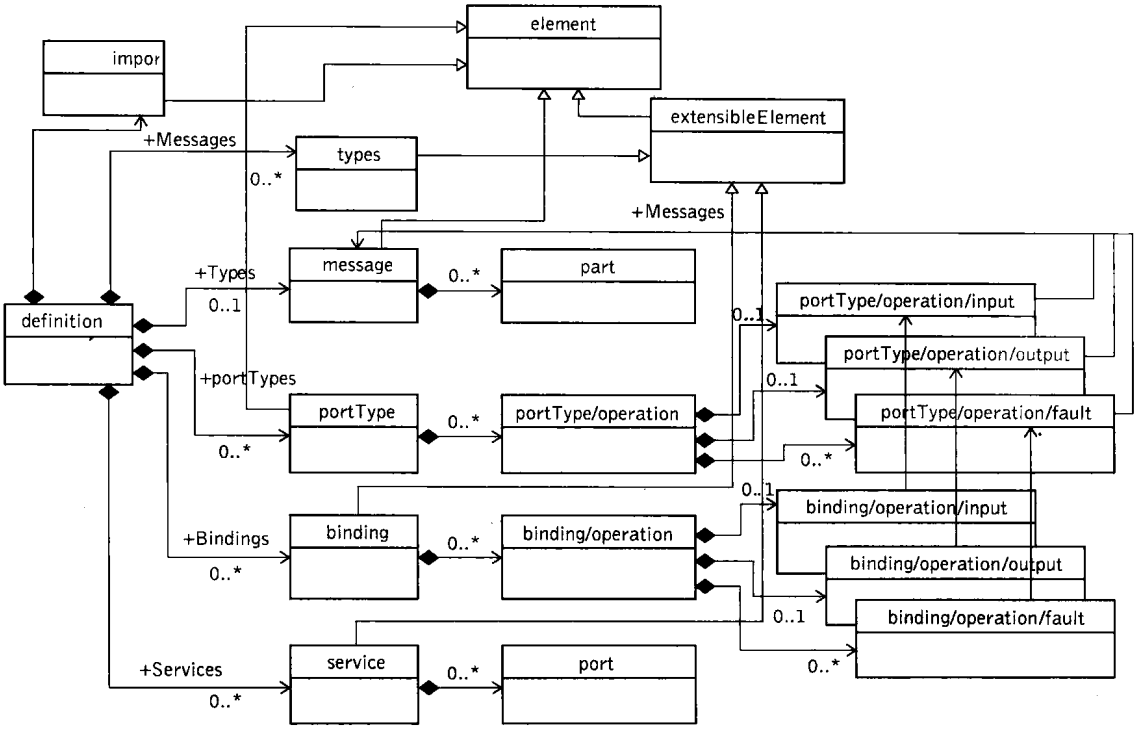


图 5.3　WSDL 接口定义元数据模型的 UML 表示

清单 5.1 是一个 WSDL 接口定义的一部分，该接口定义描述了订购单服务。订购单服务的输入包括订购单号码、日期以及顾客的详细信息，而服务的返回则是一个相关的发票凭证。在清单 5.1（和每个 WSDL 规范）中，根元素是 < definitions > 元素，该元素封装了整个 WSDL 文档，并提供了它的名字。在 WSDL 中，作为根元素的 < definitions > 元素通常包含几个 XML 命名空间定义。< definitions > 元素中的第一个属性是 name。name 用于命名整个 WSDL 文档。< definitions > 元素也声明了一个称为 targetNamespace 的属性。对于在 WSDL 文档中定义的元素，targetNamespace 标识了这些元素的命名空间，并描述了服务特征。对于单个的服务，targetNamespace 元素通常是唯

的(对于最初的 WSDL 文档时名设置一个 URL)。这有助于客户端区分不同的 Web Service，并可防止在导入其他的 WSDL 文档时出现命名冲突。这些命名空间是具有唯一性的字符串，这些字符串通常并不指向 Web 页面。属性 xmlns：tns(有时称为 this 命名空间)被设置为 targetNamespace 的值，并用来限定服务定义的属性。在 <definitions> 元素的命名空间声明中，有一个 WSDL XML 模式 http://schemas.xmlsoap.org/wsdl/的命名空间声明。一旦将 WSDL 命名空间声明为默认的命名空间，当标识每个 WSDL 元素时，就无须显式地加上前缀。命名空间定义 xmlns：soaphind 和 xmlns：xsd 都用于分别指定 SOAP 绑定的具体信息以及 XSD 数据类型。对于使用 XML XSD 的所有类型，wsdl：types 定义封装了这些类型的模式定义。

清单 5.1 简单的 WSDL 接口定义

```
<wsdl:definitions name="PurchaseOrderService"
  targetNamespace="http://supply.com/PurchaseService/wsdl"
  xmlns:tns="http://supply.com/PurchaseService/wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soaphind="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>                                                    引用数据类型定义
    <xsd:schema
      targetNamespace="http://supply.com/PurchaseService/wsdl"
      <xsd:complexType name="CustomerInfoType">
        <xsd:sequence>
          <xsd:element name="CusName" type="xsd:string"/>
          <xsd:element name="CusAddress" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="POType">
          <xsd:sequence>
              <xsd:element name="PONumber" type="integer"/>
              <xsd:element name="PODate" type="string"/>
          </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="InvoiceType">
          <xsd:all>
            <xsd:element name="InvPrice" type="float"/>
            <xsd:element name="InvDate" type="string"/>
          </xsd:all>
      </xsd:complexType>
    </xsd:schema>                                                 被发送的数据
  </wsdl:types>
  <wsdl:message name="POMessage">
      <wsdl:part name="PurchaseOrder" type="tns:POType"/>
      <wsdl:part name="CustomerInfo" type="tns:CustomerInfoType"/>
  </wsdl:message>
  <wsdl:message name="InvMessage">                                 所以回的数据
      <wsdl:part name="Invoice" type="tns:InvoiceType"/>
  </wsdl:message>
  <wsdl:portType name="PurchaseOrderPortType">                     带有一个或许的端口类型
      <wsdl:operation name="SendPurchase">
          <wsdl:input message="tns:POMessage"/>                    带有请求(输入)消息和响应(输出)消息的操作
          <wsdl:output message="tns:InvMessage"/>
      </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

WSDL 采用 W3C XML Schema 内置类型作为它的基本类型系统。WSDL <types> 元素作为容器，可包含定义 Web Service 接口的所有抽象数据类型。XML 模式描述了 WSDL 文档中所使用的数据类型定义。WSDL <types> 元素包含了 XML 模式或者对 XML 模式的外部引用。XML Schema Definition 定义了内置的原始数据类型，如 int、float、long、short、string、boolean 等。<types> 元素可以基于内置的原始数据类型来定义各种数据类型，使开发者在消息中既可以使用内置的原始数据类型也可以使用生成的复合数据类型。这就是当引用复合数据类型时，开发者为何需要定

义他们自己的命名空间的原因。

清单 5.1 在 < types > 部分定义了两类复合类型：POType 和 InvoiceType。这两类复合类型通过 targetNamespace 属性指派了 xmlns：tns 命名空间。< sequence > 元素和 < all > 元素都是标准的 XSD 元素。构建 < sequence > 需要内容模型遵循所定义的元素序列。构建 < all > 则表示：在 < complexType > 语句中声明的所有元素都必须在实例文档中出现。WSDL 文档必须知道文档中所使用的所有的命名空间。

WSDL 中的消息是类型化信息的抽象集合。消息可分配到一个或多个逻辑单元，并用于在系统之间进行通信。< message > 元素描述了发送和接受的消息的有效载荷。< message > 消息对应于调用者和 Web Service 之间的信息移动。因此，常规的往返方法调用可建模为两类消息：一类用于请求，一类用于响应。

消息表示了操作的数据类型的整体结构，< part > 元素可以进一步构成这一结构。消息可以包含一个或多个 < part > 元素，每一个部分都表示了一个特定类型(类型化参数)的实例。当 WSDL 描述了一个软件模块时，每一个 < part > 元素映射到方法调用的一个变量。消息(消息的有效载荷)中的 < part > 使用 XML Schema 内置类型、复合类型、或者 WSDL 文档的 < types > 元素中定义的元素，也可以使用通过 < import > 元素链接的外部 WSDL 元素中定义的类型。此外，消息元素能够描述 SOAP 头块的内容，以及出错信息[Monson-Haefel 2004]。

下面的代码是清单 5.1 中的一部分。这段代码表示：当和 < portType > 元素 PurchaseOrderPortType 关联时，叫作 POMessage 的消息描述了服务的输入参数，消息 InvMessage 则表示了返回(输出)参数。

```
<!-- message elements that describe input and output parameters
    for the PurchaseOrderService -->
<!--input message -->
<wsdl:message name="POMessage">
      <wsdl:part name="PurchaseOrder" type="tns:POType"/>
      <wsdl:part name="CustomerInfo" type="tns:CustomerInfoType"/>
</wsdl:message>
<! -- output message -->
<wsdl:message name="InvMessage">
      <wsdl:part name="Invoice" type="tns:InvoiceType"/>
</wsdl:message>
```

代码样例中的输入消息 POMessage 包含了两个 < part > 元素：PurchaseOrder 和 CustomerInfo。这两个元素分别引用了复合类型 POType 和 CustomerInfoType(在清单 5.1 中进行了定义)。POType 由 PONumber 和 PODate 元素组成。在清单 5.1 的 WSDL 定义的右面部分，虚线箭头将 POType 定义链接到输入消息，类似地将 InvoiceType 定义链接到输出消息 InvMessage。定义消息的方式与所使用的消息传送方式有关，例如 RPC 类型的消息传送或文档类型的消息传送。当使用 RPC 类型的消息传送方式时，< message > 元素描述了 SOAP 请求消息和回复消息的有效载荷。假如使用 RPC 类型的消息，消息通常包含多个部分。例如，在上面所示的代码中，输入消息 POMessage 包含两个 < part > 元素：PurchaseOrder 和 CustomerInfo。

下面的代码片段定义了文档类型的消息，其中 < message > 元素定义引用了 < types > 定义中的顶层元素。该代码表示了：PurchaseOrder 服务现在使用 PurchaseOrder < types > 元素定义一个文档类型的 < message > 元素。

```
<!-- message element that describes input and output parameters -->
<wsdl:message name="POMessage">
      <wsdl:part name="PurchaseOrder" element="tns:PurchaseOrder"/>
</wsdl:message>
```

在文档类型的消息传送中，在消息的 < part > 部分可以声明 < element > 来取代 < type > 属性。

WSDL 允许在 < part > 中声明 < element > 或者 < type > 属性，但不能同时既声明 < element > 也声明 < type > 。文档型 < element > 元素通常使用单向消息传送，不需要应答消息。文档型消息传送交换 XML 文档，并引用它们的顶层(全局)元素。在文档型消息传送中，< input > 是发送到 Web Service 的 XML 文档片段，而 < output > 则是返回到客户端的 XML 文档片段。

在 WSDL 中外部化服务接口描述的主要元素是 < portType > 元素。< portType > 元素定义了抽象类型和它的操作，但是没有定义具体的实现。WSDL < portType > 和它的 < operation > 元素类似于 Java 接口和它的方法声明。在 WSDL 描述中的其他元素本质上是 < portType > 元素所依赖的详细细节。操作描述了 Web Service 的接口，并定义了 Web Service 的方法。< portType > 元素是操作的逻辑分组。< portType > 元素描述了 Web Service 所支持的操作——消息传输模式和有效载荷，但是没有指定互联网协议和所使用的物理地址。可使用 < portType > 元素将逻辑操作集绑定到一个具体的传输协议(如 SOAP)。对于 WSDL 文档来说，这实现了抽象部分和具体部分的关联。在 WSDL 中，通过 < binding > 元素和 < service > 元素可以实现 < portType > 元素。< binding > 元素和 < service > 元素指定了 Web Service 实现所使用的互联网协议、编码模式和互联网地址。

WSDL 定义中既可以不含 < portType > 定义，也可以有多个 < portType > 定义。通常情况下，大多数 WSDL 文档都含有一个 < portType > 。这一例行做法将不同的 Web Service 接口定义分拆到不同的文档中。这一粒度使得每一个业务处理都有单独的绑定定义，从而实现了复用。并且对于不同的安全性、可靠性、传输机制等，很大程度上提高了实现的灵活性[Cauldwell 2001]。

WSDL < portType > 元素中可以有一个或多个 < operation > 元素，每一个 < operation > 元素定义了一个 RPC 类型的 Web Service 方法或文档类型的 Web Service 方法。每一个 < operation > 元素最多包含一个 < input > 元素或者一个 < output > 元素，单可以包含任意数量的 < fault > 元素。WSDL 中的操作类似于编程语言中的方法签名，它们表示了服务所暴露的不同的方法。操作定义了 Web Service 上的一个方法，包括方法名和输入输出参数。一个常规的操作定义了输入和输出参数或者操作的异常(出错)。

清单 5.1 中的 WSDL 样例定义了一个 Web Service。这个 Web Service 包含一个名为 PurchaseOrderPortType 的 < portType > ，该 < portType > 支持一个称为 SendPurchase 的 < operation > 。清单中的这个样例假设服务使用 SOAP v1.1 作为它的编码方式，并将该服务绑定到 HTTP。

操作控制 Web Service 客户端和 Web Service 提供者之间交换的所有消息。假如定义了出错消息，这些出错消息也将是 < operation > 元素的一部分。

在清单 5.1 中，< portType > PurchaseOrderPortType < operation > 元素是一个 RPC 类型的操作。该操作将 POMessage 消息声明为 < input > 消息，并将 Inv(oice)Message 消息声明为 < output > 消息。< input > 消息表示了发送到 Web Service 的有效载荷，而 < output > 消息则表示了发送到客户端的有效载荷。在清单 5.1 中，使用消息 POMessage 调用 < operation > 元素 SendPurchase，并使用消息 Inv(oice)Message 返回结果。在清单 5.1 中，操作的输入和输出消息元素将服务方法 SendPurchase 链接到传输输入参数和输出结果的 SOAP 消息。在 Web Service 中，可以以四种基本的模式使用操作：请求/响应、要求/响应、单向、通知。SendPurchase 操作包含一个输入消息和一个输出消息，因此它是一个典型的请求/响应类型的操作。在 5.2.3 节中将描述 WSDL 操作模式。

5.2.2 WSDL 的实现

在前面的章节中，我们已经讨论了以抽象的方式定义 WSDL 操作和消息，而没有涉及实现细节。事实上，WSDL 的目的就是首先抽象地定义 Web Service，然后规定 WSDL 开发者如何实现这些服务。服务的具体的实现层规定了如何实现服务的抽象定义。WSDL 的服务实现部分包含元素 < binding > (虽然有时候 binding 元素被视为服务定义的一部分)、< port > 和 < service > ，并描述了

服务提供者如何实现一个特定的服务接口。服务实现描述了服务位于哪里，或者更精确地说：为了调用 Web Service，需要将消息发送到哪一个网络地址。通过 <import> 元素，服务实现文档可以包含对多个服务接口文档的引用。Web Service 实现元素如图 5.3 所示，下面将进行概述。

清单 5.2 中的 WSDL 样例是清单 5.1 所示的抽象服务接口（<portType> 元素）的实现描述。实现描述的主要元素是 <binding> 元素。<binding> 元素规定了客户端和 Web Service 之间如何交换消息。客户端使用该信息访问 Web Service。<binding> 元素将端口类型（如服务接口描述）绑定到一个已有的服务实现，并提供有关协议和具体的数据格式的信息。不同网络地址上所提供的服务需要这些信息[Zimmermann 2003]。在 WSDL 中，<binding> 元素包含如何将抽象服务接口（<portType> 和 <operation>）映射到具体的表示，包括具体的协议（例如 SOAP 或 HTTP）、消息传送类型（例如 RPC 或文档类型），和与 <portType> 元素关联的格式（编码）类型（例如文本或 SOAP 编码）。

清单 5.2　WSDL 服务描述

```
<wsdl:definitions>...
  <import namespace="http://supply.com/PurchaseService/wsdl"
  location="http://supply.com/PurchaseService/wsdl/
  PurchaseOrder-interface.wsdl"/>
<!--location of WSDL PO interface from Listing 1-->
<!--wsdl:binding states a serialisation protocol for
  this service -->
<!--type attribute must match name of portType
  element in Listing 1-->
<wsdl:binding name="PurchaseOrderSOAPBinding"
              type="tns:PurchaseOrderPortType">

  <!--leverage off soapbind:binding synchronous style -->
  <soapbind:binding style="rpc"
                    transport="http://schemas.xmlsoap.org/soap/
                    http"/>

  <wsdl:operation name="SendPurchase">

    <!--again bind to SOAP -->
    <soapbind:operation
     soapAction="http://supply.com/PurchaseService/wsdl/
       SendPurchase" style="rpc"/>

    <!--further specify that the messages in the wsdl:operation
      use SOAP -->
      <wsdl:input>
        <soapbind:body use="literal"
          namespace="http://supply.com/PurchaseService/wsdl"/>
      </wsdl:input>
      <wsdl:output>
        <soapbind:body use="literal"
          namespace="http://supply.com/PurchaseService/wsdl"/>
      </wsdl:output>

  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="PurchaseOrderService">
  <wsdl:port name="PurchaseOrderPort" binding="tns:
    PurchaseOrderSOAPBinding">
    <!--give the binding a network endpoint address or URI
      of service -->
    <soapbind:address location="http://supply.com/3000/
      PurchaseOrderService"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

例如，假如使用基于 HTTP 的 SOAP 发送消息，绑定描述了如何将消息映射到 SOAP < Body > 和 < Header > 元素，以及描述了对应的属性值。每一类协议（例如 MIME、SOAP、HTTP GET 或 POST）都有针对自身协议的元素集和它自身的命名空间。图 5.4 表示 WSDL 接口的元素和实现的相互关系。

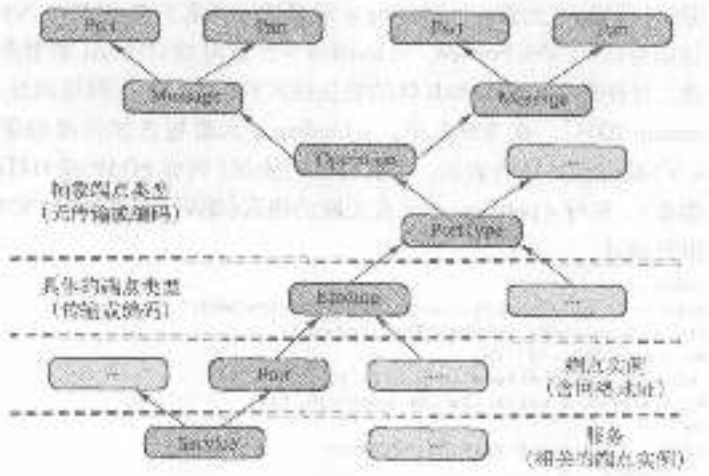


图 5.4 使用 WSDL 元素定义端点

< wsdl: binding > 元素（此后将称作 < binding > 元素）的结构与 < portType > 元素类似。这并不是巧合。记住绑定必须将抽象接口类型描述映射到一个具体的实现。< type > 属性标识了绑定所描述的 < portType > 元素。在清单 5.2 中声明的 < binding > 元素实际由两个不同的命名空间组成。一方面，有些元素是 WSDL 1.1 命名空间"http://schemas.xmlsoap.org/wsdl/"的成员，这个命名空间是在清单 5.1 中声明的，它是 WSDL 文档的默认的命名空间。另一方面，soapbind: binding、soapbind: operation 和 soapbind: body 元素是 SOAP 特有的元素，它们是用于 SOAP-WSDL 绑定的命名空间"http://schemas.xmlsoap.org/wsdl/soap/"的成员，该命名空间也是在清单 5.1 中声明的。soapbind: binding（不要与 WSDL < binding > 元素混淆）和 soapbind: body 元素表示了 Web Service 有关 SOAP 方面详细信息。更具体地说，在清单 5.2 中，SOAP 绑定元素 < soapbind: binding > 表示了消息传送的类型是 RPC（通过 soapbind: operation 构成和 style 属性进行表示的，其中 style 属性定义了绑定中的默认操作的类型）；绑定将要使用的低层传输服务是 HTTP（通过 transport 属性指定的）；并且将要使用 SOAP 格式作为绑定和传输服务。该声明应用于整个绑定。它表示了：在绑定中 PurchaseOrderPortType 的所有操作都定义为 SOAP 消息。SOAP 的责职就是使得客户端能从抽象的 WSDL 规范连接到它的具体实现。生于 SOAP 用于该目的，因此也必须使用 SOAP 命名空间。

< soapbind: body > 元素使得应用程序能够指定输入消息和输出消息的详细信息，并可将抽象的 WSDL 描述映射到具体的协议描述。尤其，< soapbind: body > 元素指定了 SOAP 编码类型以及与具体服务（在我们的例子中是 PurchaseOrderService）相关的命名空间。在清单 5.2 中，< soapbind: body > 构成指定了输入消息和输出消息，这些消息是按照字面进行编码的，术语 literal 表明可以根据文档的 XML 模式验证 XML 文档片段。< binding > 元素的子元素（< operation > 、< input > 和 < output > ）直接映射到 < portType > 元素对应的子元素。在清单 5.2 中，可使用 < soapbind: operation > 元素表示具体操作（如 SendPurchase）的绑定，以及将输入消息和输出消息从抽象服务接口描述（参见清单 5.1）映射到一个具体的 SOAP 实现。通过服务接口描

述中的 XSD 可以抽象地描述这些输入消息和输出消息。为了进行传送，需要将这些消息进行 SOAP 编码。

可以使用多个绑定来表示同一个 < portType > 的不同实现。假如服务支持多个协议，WSDL < portType > 元素对于所支持的每一个协议都包含一个 < binding > 。对于特定的 < portType > 元素，< binding > 元素能够描述如何使用一个消息传送/传输协议(例如基于 HTTP 的 SOAP、基于 SMTP 的 SOAP、简单的 HTTP POST 操作，或者其他的网络和消息传送协议组合)来调用操作。当前，最流行的绑定技术就是使用基于 HTTP 的 SOAP。绑定并不包含与编程语言、具体服务、具体相关的详细细节。对于 WSDL 来说，并不关心如何实现服务。

正如前面已经讨论的，清单 5.2 中所示的绑定包含一个 SendPurchase 操作，该操作将 SendPurchase 操作的出错元素、输入、输出从 PurchaseOrderPortType(参见清单 5.1)映射到它的 SOAP 连线格式。用这种方法可以访问 Web Service PurchaseOrderService。清单 5.2 中的 soapbind: operation 指定了具体操作的消息传送类型(RPC 或文档型)，并指定了 SOAPAction 头部域的值。在 WSDL 中，style 属性是可选的。通过 < soapbind: binding > 元素，可以声明默然的消息传送类型。使用 style 属性可以重载默然的消息传送类型。SOAP 客户端可以使用 < soap: operation > 元素中的 SOAPAction 属性来生成 SOAP 请求。可以使用 < soap: operation > 元素中的 SOAPAction 属性来指定 HTTP SOAPAction 头部。SOAP 服务器在运行时收到消息后，将会根据 SOAPAction 头部决定将要采取的动作。在服务实现中通常通过获取方法名来调用。SOAPAction 是一个与具体的服务器 URI 相关的属性，用来表示请求的目的。SOAPAction 属性可以包括一个消息路由参数，或者包括一个可以帮助 SOAP 运行时系统将消息分发给合适服务的值。这样做的目的是为了实现客户端和服务提供者应用之间的互操作性。SOAP 客户端将从 WSDL 文件中读取 SOAP 结构，并与另一端的 SOAP 服务器进行协作。

消息传送的类型对于如何构建 SOAP 消息体具有直接的影响，因此声明一个合适的类型(“rpc”或“document”)是非常重要的。当使用 RPC 类型的消息传送时，SOAP 消息的 < Body > 将包含一个元素，该元素表示将要完成的操作。这个元素从 < operation > 中获得它的名字，其中 < operation > 是在 < portType > 元素中定义的。图 5.5 显示了如何在 < portType > 元素中定义 < operation > ，以及如何将各部分的消息(例如订购单)映射到一个 RPC 类型的 SOAP 消息中。

```
<wsdl:input>
    <soapbind:body use="literal"
     namespace="http://supply.com/PurchaseOrderService/wsdl"/>
</wsdl:input>

<wsdl:output>
     <soapbind:body use="literal"
      namespace="http://supply.com/PurchaseOrderService/wsdl"/>
</wsdl:output>
```

上面的代码是清单 5.2 中的代码片段。该段代码说明了 < operation > SendPurchase 的输入消息和输出消息是如何出现在 SOAP 消息的各部分的。< operation > SendPurchase 的 < input > 元素和 < output > 元素精确地指定了操作中的输入和输出消息将如何出现在 SOAP 消息中。输入和输出都包含了含有命名空间值的 < soapbind: body > 元素。其中命名空间的值对应于在 SOAP 服务器上部署的服务的名称。在 RPC 类型的消息中，必须通过一个有效的 URI 来指定 namespace 属性。该 URI 可以与 WSDL 文档中的 targetNameSpace 属性相同。相比之下，文档型的消息并不一定需要在 < soapbind: body > 元素中指定 targetNameSpace 属性。可以从 XML 模式中获取 XML 文

的片段的命名空间。



图 5.5 将 SendPurchase 和它的各部分消息映射到一个 RPC 类型的 SOAP 消息中

下面讨论 SendPurchase 操作的 < input > 元素。在 SendPurchase 操作的 < portType > 声明中，所声明的 POMessage 消息是抽象的。use = "literal" 属性表明了这一点。这意味着：定义 input 消息的 XML 和它的各部分实际上都是抽象的，并且可以获取数据的真正的、具体的表示。< soapbind: body > 元素中的 use 属性的作用是使得应用可以规定如何定义消息的各部分。"literal" 编码表示了：生成的 SOAP 消息包含一些数据，并且这些数据是精确地按照抽象的 WSDL 定义进行格式化的。这实质上意味着：SOAP 消息体中的参数是一个 ASCII 字符串，这些字符串是消息中用中所指定的 < part > 类型的实例。因此，将会根据类型定义部分的精确表示（例如按照 XML 模式），对消息部分引用的数据类型进行串行化。这意味着需要使用模式来验证各个消息部分的实例。假如我们使用 encoded 类型的编码取代 literal 类型的编码，则还需要使用 encodingStyle 属性来表示对消息串行化的具体方法。消息应该录现为 < soapbind: body > 元素的一部分。服务提供者网络上的 SOAP 运行时系统应该根据 SOAP 规范中定义的编码规则对数据进行逆串行化，将数据从 XML 格式转换为其他格式，例如 Java 数据类型。

< service > 元素最终将 Web Service 绑定到一个可进行网络寻址的具体地址。< service > 元素进行先前所声明的绑定，并将这些绑定与一个或多个 < port > 元素关联起来。每一个 < port > 元素表示一个 Web Service。< service > 被当做为相关的 WSDL < port > 元素的集合，其中 < port >

元素是一个端点，这个端点是根据绑定和网络地址进行定义的，并可根据这个网络地址访问相应的服务。每一个 < service > 元素都有各自的名字。在 WSDL 文档中，所有的服务名都必须具有唯一性。 < service > 元素的 < port > 上驻留了提供给客户端的相关的操作集。

使用一个特定的传输协议，可以调用具体的 < portType > 的操作。在 WSDL 中， < port > 元素定义了这些操作所处的位置。每一个 < port > 都与一个端点相关联，例如网络地址或者含有具体的 WSDL < binding > 元素的 URL。服务请求者必须使用物理端点才能连接到服务。由于 < binding > 指定了一个 < portType > ，因此 < port > 元素有效地将 < portType > 与一个地址关联起来。

图 5.6 显示了一个服务可以包含多个端口，这些端口全部使用相同的 < portType > 。这意味着：对于多个不同的服务提供者所提供的同一个服务接口，可以有多个服务实现。这些服务实现有不同的绑定和/或地址。例如，一个特定的垂直产业可以提供一个标准的用于订单管理的 < portType > 。不同的生产商可以提供不同的订单管理服务，并且这些订单管理服务实现了相同的 < portType > 。所有的这些实现都将提供语义上等价的行为。作为选择，对于相同的 < portType > ，由一个特定的提供者所实现的服务可以包含几个 < port > ，每一个 < port > 都由不同的 < binding > 提供。从而，服务请求者可以选择最方便的方式与实现服务的服务提供者进行通信。

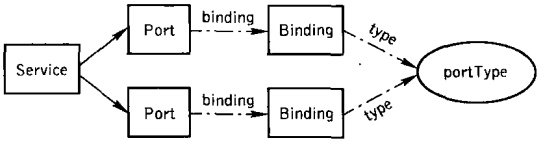


图 5.6　连接服务接口与服务实现

在清单 5.2 中，名为 POMessageSOAPBinding 的 < binding > 元素将名为 PurchaseOrderPortType (引用清单 5.1)的 < portType > 元素链接到名为 PurchaseOrderPort 的 < port > 元素。正如从清单的虚线箭头部分所看到的，通过绑定 name POMessageSOAPBinding 可以实现这一点。清单中仅包含一个 Web Service，即 PurchaseOrderService，因此可以仅使用名为 PurchaseOrderPort 的 < port > 元素来表示服务的位置。然而，正如已经讨论的，PurchaseOrderService 服务能够包含三个端口。这三个端口都使用 PurchaseOrderType，但是分别绑定到基于 HTTP 的 SOAP、基于 SMTP 的 SOAP、HTTP GET/POST。与服务进行交互的客户端可以选择所要采用的与服务进行通信的协议(可以在绑定中使用命名空间以可编程方式进行通信)，也可以选择最近的地址[Cauldwell 2001]。通过应用不同的绑定，可以更容易地在更大范围的平台上访问服务。例如，个人计算机桌面应用程序可以使用基于 HTTP 的 SOAP，然而由于在 WAP 应用中通常没有 XML 解析器，因此在手机上运行的 WAP 应用则可以使用 HTTP GET/POST。这三个服务在语义上完全等价的，它们都是将订单号、日期以及顾客的详细信息作为输入，并返回一个相关的票据。

清单 5.2 中的 < soapbind: address > 属性是另一个对 WSDL 的扩展，该属性用于表示服务的 URI 或者用于表示网络端点。该元素通过它的 location 属性将一个互联网地址赋给一个 SOAP 绑定。Web Service 客户端将要绑定到一个端口，并与特定的服务进行交互，该服务需要理解并处理具体的消息。

图 5.7 显示了客户/服务器交互中所涉及的不同的 WSDL 元素，并概括了前面所讨论的几种构成。在图中，有一个客户端通过基于 HTTP 的 SOAP 调用 Web Service，而另一个客户端通过 HTTP 调用同一个 Web Service。图 5.7 描述了一个包含多个端口的服务。如图所示，一个服务可以包含多个端口，与 < portType > 相关联的绑定元素将绑定到这些端口上。服务提供者有不同的绑定和/或地址。正如已经讨论的，通过 < port > 的 < soapbind: address > 元素可以指定端口地址。

最后，图 5.8 概括地表示了 WSDL 的具体层次和抽象层次是如何相连的。图中弯折的项表示了子元素所描述的信息。在图 5.8 中，为了与元素进行区分，属性用斜体进行表示。

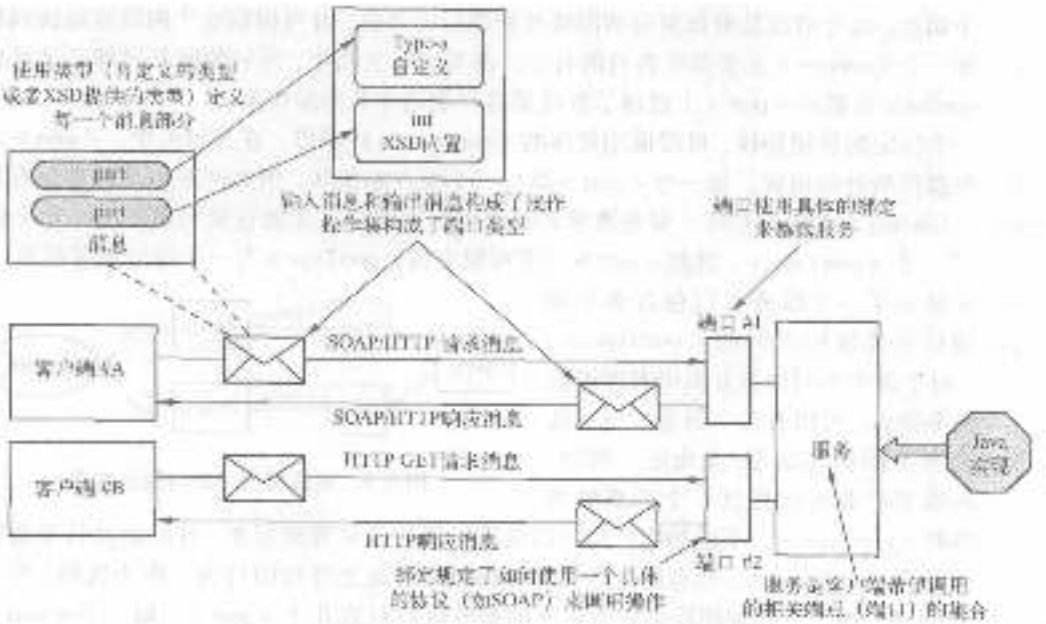


图 5.7 请求者和服务之间的进行交互的部分 WSDL 元素

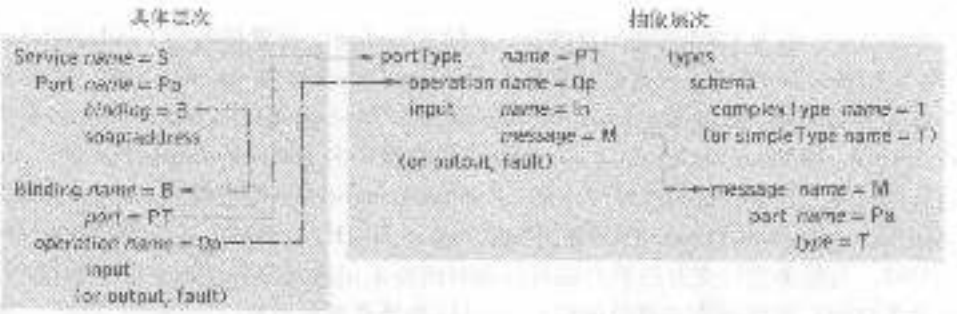


图 5.8 连接 Web Service 的具体层次和抽象层次

引自：M. Kifer, A. Bernstein 和 P. M. Lewis, Database Systems: An Application-Oriented Approach（第 2 版），Addison Wesley, 2005（允许复制）

5.2.3 WSDL 的消息交换模式

WSDL 接口支持四类操作。这些操作表示了 Web Service 的最常见的交互模式。因为 WSDL 定义的每一类操作都能够有一个输入和/或输出，四类 WSDL 交互模式表示了输入消息和输出消息的可能的组合形式[Cauldwell 2001]。WSDL 操作对应于两类基本的消息接收和发送原本：一类是单个的消息接收传送操作和对应的发送操作（"单向"和"通知"操作），另一类是同步双向消息交换（"请求/响应"和"要求/响应"）。这些消息交换模式如图 5.9 所示，并在下面概述这几种模式。为简

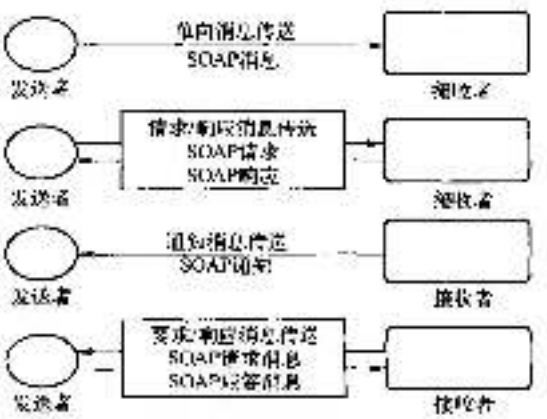


图 5.9 WSDL 消息传送模式

洁起见，我们仅给出两类最流行的消息交换模式的样例代码："单向"和"请求/响应"。

单向操作

单向操作是服务端点接收消息的一种操作，但是该操作并不发送一个响应。例如，将订购单提交给订购系统的操作可以是一个单向操作。在发送订单之后，并不期待会立即有一个响应。这种消息传送模式通常视为异步消息传送。单向消息仅仅定义了一个输入消息，它既不需要输出消息，也不需要出错消息。

如果 < operation > 元素声明为具有一个 < input > 元素，并且没有 < output > 元素，则它定义了一个单向操作。假如 < operation > 元素仅具有 < input > 消息，这表明客户端将向 Web Service 发送消息，但客户端不会收到任何响应消息。下面的代码片段说明了 SubmitPurchaseOrder < portType > ，这段来自于文献[Monson-Haefel 2004]的代码定义了一个单向操作：

```
<!-- portType element describes the abstract interface of a Web
 service -->
<wsdl:portType name="SubmitPurchaseOrder_PortType">
     <wsdl:operation name="SubmitPurchaseOrder">
          <wsdl:input name="order"
           message="tns:SubmitPurchaseOrder_Message"/>
     </wsdl:operation>
</wsdl:portType>
```

请求/响应操作

服务端点接收一个消息，并返回一个响应消息，这样的操作称为请求/响应操作。在请求/响应消息模式中，客户端请求服务提供者执行一些操作。假如 < operation > 元素首先声明含有一个 < input > 元素，紧接着声明含有一个 < output > 元素，则它定义了一个请求/响应操作。 < operation > 的 < input > 标签表明 Web Service 将收到客户端发送的消息， < output > 标签则表明 Web Service 将响应该消息。源自清单 5.1 的下列代码片段说明了 SendPurchase 操作，这是一个请求/响应消息传送模式的具体实例。SendPurchase 操作接收一个输入消息，该输入消息包含订购单（订单号码和日期）和顾客详细信息，然后返回一个响应消息，响应消息包含一个票据。在一个 RPC 环境中，这类似于一个过程调用，因为过程调用具有一个输入变量列表并有一个返回值。

```
<!-- portType element describes the abstract interface of a Web
 service -->
<wsdl:portType name="PurchaseOrder_PortType">
     <wsdl:operation name="SendPurchase">
          <wsdl:input message="tns:POMessage"/>
          <wsdl:output message="tns:InvMessage"/>
     </wsdl:operation>
</wsdl:portType>
```

通知操作

服务端口将消息发送给客户端，但并不期待一个响应，这类操作称为通知操作。需要将事件通知给客户端的服务将使用这类消息传送方式。使用通知消息传送模式的 Web Service 遵循分布式计算的"推"模式。在推模式中，假设客户端（订阅者）已经注册到 Web Service，并接收有关事件的消息（通知）。对于通知来说，在 < portType > 元素中包含一个 < output > 标签，但没有 < input > 消息的定义。试举一个这类服务模型的例子：将事件报告给服务，并且端点周期性地报告它的状态。在这种情况下，不需要响应。最可能的情况是：收集并记录这些状态数据，但并不会立即采取行动。

要求/响应操作

服务端点发送一个消息，并期待收到一个响应消息，这类操作称为要求/响应操作。因为服

务端点发起这个操作(要求客户端),而不是响应客户端的请求,所以这个操作是与请求/响应操作相反的一个操作。要求/响应操作有点类似与通知操作,但是该方式期待客户端响应 Web Service。与通知消息传送方式类似,在要求/响应方式中,Web Service 的客户端为了接收消息必须订阅服务。对于这类消息传送,<portType>元素首先声明一个<output>标签,然后声明一个<input>消息定义,正好与请求/响应操作完全相反。试举一个这类操作的实例:服务向客户端发送订单状态并接收一个收到确认。

表 5.1 WSDL 消息交换模式概要

类 型	定 义
单向	该操作能够接收消息,但不会返回响应
请求/响应	该操作能接收消息,并将返回一个响应
通知	该操作能够发送消息,但不会等待响应
请求/响应	该操作能够发送请求,并将等待一个响应

表 5.1 概括并比较了前面章节所描述的四类 WSDL 消息传送模式。

需要注意一点,在一个 WSDL 接口中可以包括接收消息操作和发送消息操作的任何组合形式。因此上面所述的四类操作在接口层可支持"推"方式的交互和"拉"方式的交互。为了支持服务之间的松耦合的对等交互,需要在 WSDL 中定义发送消息的操作。

5.3 使用 WSDL 生成客户端 stub

为了帮助读者更好地理解 Web Service 中的 WSDL 基础知识,我们已经在本章中介绍了几类 WSDL 元素。了解这一技术可更好地理解 Web Service 模型。然而,大多数 Web Service 开发者并不需要直接和基础设施打交道,有许多 Web Service 开发工具箱可帮助处理这项任务。对于服务请求者和服务提供者,当前有许多工具自动地将 WSDL 映射到编程语言(如 Java),其中最流行的工具之一是 Axis 提供的 WSDL2Java。Axis 是一个开源工具箱,它是 Apache(xml. apache. org)项目的一部分。开发人员可以使用 Axis 编写 Java 代码,并可将这些代码部署为 Web Service。下面,我们将主要讨论代码生成工具是如何自动生成 WSDL 定义以及创建 Web Service 的。

通过集成一些可用的 Web Service,开发人员可以在应用中实现 Web Service 逻辑,且无须从头开始开发新的应用。代理类使得这一方法成为可能。通过代理类,开发者可以引用远程 Web Service,并可在本地应用中调用那些 Web Service 所提供的功能,那些 Web Service 所返回的数据就好像是本地生成的。通过向这些本地对象发送消息,应用开发者可以与任何远程对象通信。通常将这些本地对象称为代理对象。代理类(或桩类)是实现 Web Service 的远程(提供者)对象类的客户端映像。在分布式计算环境中,在服务器端与这些代理类对应的类通常称为骨架(skeleton),参见 2.4.1 节。代理类与远程类实现的是同样的接口。代理类会将在本地实例上被调用的方法转发到对应的远程实例(骨架)。代理类是将方法传递到它所代表的 Web Service 的本地对象。对于每一个远程对象都有代理,本地对象拥有远程对象的引用。代理实现了服务提供者对象的远程接口中的方法,从而确保所要调用的方法与远程对象是一致的。代理类的作用就是充当远程对象的本地表示。对于客户端来说,代理类基本上是一个远程引用。然而,代替执行调用,代理以消息的形式将调用转发给一个远程对象。代理隐藏了远程对象引用的细节,使用对象串行化将变量编进(marshaling)后发送到远程对象,并调用跨网络发送到服务提供者所处的计算机。实现服务的远程对象将返回结果,代理将返回的结果进行编出(unmarshaling)处理。

WSDL 非常适合代码生成器。代码生成器能够理解 WSDL 定义,并可生成访问 Web Service

的编程接口。例如，JAX-RPC 提供者可以使用 WSDL 1.1 生成 Java RMI 接口和网络桩，其中网络桩用于与 Web Service 接口交换消息。图 5.10 显示了诸如 JAX-RPC（Monson-Haefel 2004）这样的 WSDL 工具箱如何生成 RMI（一个端点）接口以及实现接口的网络代理。对于诸如 PurchaseOrderService 这样的 Web Service，Web Service 开发工具可以根据这些 Web Service 的 WSDL 定义生成一个（Java）客户端代理。基于客户端代理，本地应用（在图 5.10 中是 Java 程序）能够与对应的远程 Web Service 进行通信。



图 5.10 基于 WSDL 代码生成器生成代理

WSDL 代码生成器工具可自动创建 Web Service、自动生成 WSDL 文件以及自动调用 Web Service。基于 WSDL 规范，工具箱软件可生成服务实现模板代码。通过这些模板代码，以及利用具体应用的实现细节，开发者可以更快地创建 Web Service。使用这些工具箱软件，可以基于 WSDL 规范生成服务代理代码，从而简化客户端应用的开发。

一旦构建好代理类后，客户端可以从代理类中调用 Web 方法，并且代理完成真正的 Web Service 请求。显然，请求的端点可以在网络的任意地方。当我们在客户端应用中引用 Web Service 时，代理类显然是客户端应用程序的一部分，正如通常的内部函数调用一样。代理和 Web Service 之间的通信流程如图 5.11 所示。整个流程包含以下步骤：

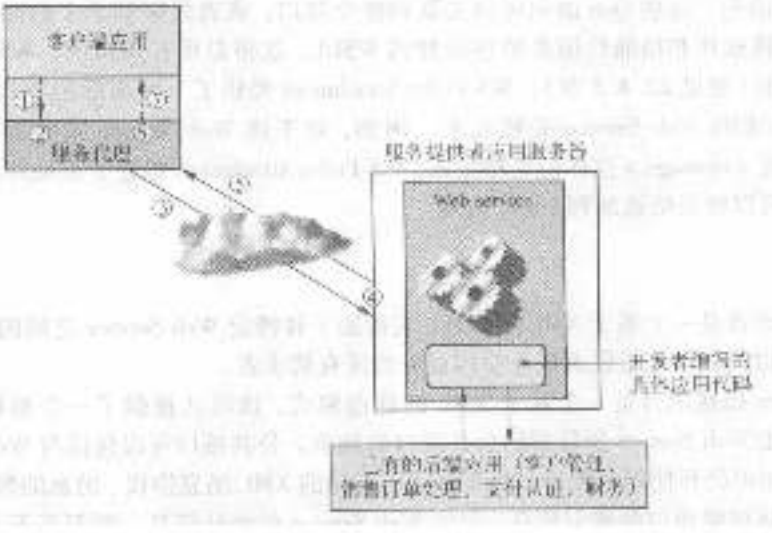


图 5.11 代理类与 Web Service 之间的通信

1. 客户端应用在代理类中执行调用，并将任何所需的变量传递给代理类，并且无须知道代理实际上是在调用一个远程 Web Service。

2. 代理接受调用，然后基于客户端应用所提供的参数，按一定的格式生成服务请求。

3. 将调用从代理跨网络传输到 Web Service。

4. Web Service 基于代理所提供的参数执行相关的服务操作，并使用 XML 表示请求处理的结果。

5. Web Service 将结果数据返回给客户端代理。

6. 代理对于从 Web Service 返回的数据进行解析，从而获取各个数据值。如前所述，这些值既可以是简单数据类型，也可以是复杂数据类型。

7. 应用从代理操作中接收这些标准格式的数值，并且完全无须知道这些结果实际上是通过 Web Service 调用获取的。

5.4 WSDL 中的非功能性描述

到目前为止，我们已经讨论了 WSDL 对于服务的句法签名，然而还没有讨论非功能性服务。然而，在 1.8 节中，我们就已经提出了：对于任何 Web Service，非功能性特性都是重要的有机组成部分之一。Web Service 平台能够支持具有不同的 QoS 需求的、多种不同类型的应用。事实上，为了能够开发调用 Web Service 的应用以及开发与 Web Service 进行交互的应用，程序员和应用都需了解 Web Service 的 QoS 特性。因此需要描述 Web Service 的非功能性特性。

对于启用 QoS 的 Web Service 来说，需要一种单独的语言来描述 Web Service 的非功能性特性。当前，用来描述 Web Service 的非功能性特性的最常用的方法是 WS-Policy 和 WS-PolicyAttachment 这两类规范。我们将在第 12 章中讨论这两类规范。Web Service 策略框架对于服务提供了附加的描述层，并提供了一种声明策略语言。可使用声明策略语言来表示策略或对策略进行编程。通过策略语言，可以描述 Web Service 驻留环境的特性，包括提供者端点的安全特性(包括认证和授权)、事务行为、QoS 的等级、提供者所提供的保护质量、提供者所遵循的隐私策略、具体应用服务的选项、针对特定服务域的能力与约束。

就涉及 QoS 的 Web Service 而言，需要对服务接口规范进行扩充，在服务接口规范中添加一些有关 QoS 的语句。这些 QoS 语句可以关联到整个接口，或者关联到单个的操作和属性。假如能以标准方式将这些非功能性服务描述添加到 WSDL，这将是很有价值的。WS-PolicyAttachment 实现了这一目标(参见 12.4.3 节)。WS-PolicyAttachment 提供了一种灵活的方式，可将策略表达与已有的或未来的 Web Service 关联起来。例如，对于将 Web Service 策略关联到诸如 WSDL < portType > 或 < message > 这样的策略主题，WS-PolicyAttachment 描述了相关需求。WS-PolicyAttachment 甚至可以将策略施加到 UDDI 实体。

5.5 小结

服务描述语言是一个基于 XML 的语言，它描述了和特定 Web Service 之间的交互机制，并且使用该语言可以约束服务提供者以及使用服务的所有请求者。

Web Service 描述语言是一个基于 XML 的规范模式，该模式提供了一个标准的服务表示语言，可用于描述 Web Service 所暴露的公共接口的细节。公共接口可以包括与 Web Service 相关的操作信息，诸如可公开使用的操作、Web Service 支持的 XML 消息协议、消息的数据类型信息、关于所使用的具体传输协议的绑定信息、定位 Web Service 的地址信息。对于基于 SOAP 的 XML 文档传输，可以使用 WSDL 来定义服务规范。

WSDL 的服务实现部分描述了一个特定的服务提供者如何实现一个具体的服务接口。服务实现描述了服务所处的位置，更精确地说，为了调用 Web Service，需要将消息发送到哪一个网络地址。

WSDL 规定了服务的句法签名，但是对于非功能性服务方面并没有进行任何规定。描述非功能性 Web Service 特性的最通用的方法是 Web Service 策略框架。该框架对于服务提供了附加的描述层，并且还提供了一个声明策略语言。可使用该策略语言表示策略，或对策略进行编程。该框架可以标准方式向 WSDL 中添加非功能性服务描述。从而，在 Web Service 描述中可以添加许多 QoS 特性，诸如性能、提供者端点的安全性(包括认证和授权)、事务行为、提供者所提供的保护质量的等级、提供者所遵循的隐私策略等。

当前，万维网联盟(W3C)正忙于对 WSDL 进行标准化。虽然在本书中我们将使用 WSDL 1.1，该标准也是实际标准，但是 W3C 目前正在制定新的 WSDL 标准——WSDL 2.0。与 WSDL 1.1 相比，WSDL 2.0 更简单也更有用。它在若干方面都进行了改进，包括语言的澄清说明和简化，以及对于互操作的支持，因此开发人员可以更容易地了解和描述服务。最近，WSDL 1.2 定义也已经在一些方面进行了修改，如 < portType > 元素被更名为 < interface > 元素[Weerawarana 2005]。WSDL 1.2 定义也以 < extends > 属性的形式支持一些有用的新特性。通过 < extends > 属性，可将多个 < interface > 声明集成在一起并进行进一步扩展，从而生成一个全新的 < interface > 元素。可以预测，由于工具和运行时环境的原因，还需一段较长的时间，WSDL 2.0 才能全面取代 WSDL 1.1。

复习题

- 在表示 Web Service 时，为何需要进行服务描述?
- Web Service 描述语言的目的是什么? WSDL 是如何实现它的目标的?
- 定义并描述 Web Service 接口。
- 定义并描述 Web Service 实现。
- Web Service 的接口和实现之间是如何进行关联的?
- 描述 Web Service 的 < portType > 元素。
- 描述 Web Service 的 < wsdl: binding > 元素。
- 在 WSDL 中如何定义 RPC 类型的 Web Service 和文档类型的 Web Service?
- 单个服务能否包含多个端口? 这意味着什么?
- 单向操作与请求/响应操作的不同点是什么?
- 通知操作与要求/响应操作的不同点是什么?
- 如何使用 WSDL 创建客户桩?

练习

5.1 使用 WSDL 定义一个简单的股票交易 Web Service，查询与一个具体的股票行情显示符号相关的股票价格。本题类似于练习 4.3。

5.2 使用 WSDL 定义一个简单的且使用 RPC/literal 和文档绑定的保险索赔 Web Service。

5.3 使用 WSDL 定义一个简单的 Web Service，返回特定航空公司的航班信息，参见练习 3.4。

5.4 使用 WSDL 定义一个简单的 Web Service，基于练习 3.5 中的汽车租赁预订模式，预订顾客所要求的车辆。

5.5 基于练习3.3中所定义的信用卡处理模式，定义一个WSDL接口。典型操作包括：用于信用卡借记授权的“CreditCardSale”、用于借记卡借记授权的“DebitCardSale”、用于取消信用卡销货的“CancelCreditCardSale”、用于确定信用卡销货的当前状态的“CheckCardDebitStatus”、用于获取信用卡用户的详细信息的“CreditCardUserDetails”等。

5.6 开发一个简单的库存检查服务。该库存检查服务将检查物品的库存情况。基于该检查，当库存能够满足订购单需求时，则响应订购单的请求，否则返回一个出错信息。

第6章 Web Service的注册与发现

学习目标

面向服务的体系结构(SOA)这一方法的两个最核心的功能是服务的注册与发现。在 Web Service 应用中，服务注册用于记录一个组织已经提供了哪些服务，以及记录那些服务的特性。为了解决服务注册和发现所面临的一些任务，创建了通用描述发现和集成(UDDI)规范。UDDI 是由多个行业提议所创建的一个用于 Web Service 的描述发现的注册标准。注册设施支持 Web Service 的发布和发现。通过该标准和注册设施，可实现 Web Service 的描述和发现。

本章中，我们将描述 Web Service 的服务注册和服务发现的作用。通过本章的学习，读者将了解下列关键概念：

- UDDI 作为标准注册的使用。
- UDDI 数据结构以及它们与 WSDL 文档的关系。
- 从 UDDI 到 WSDL 的映射模型。
- UDDI API。
- 在 UDDI 中如何使用 UDDI API 发布 Web Service，以及如何在 UDDI 中查询所包含的 Web Service。
- 不同的 UDDI 用例模型和变体。

6.1 服务注册

为了充分发挥电子商务的作用，不同的企业必须能够彼此发现、了解各自的需求和所能提供的功能，以及能够将不同企业的 Web Service 组合成新的服务和业务流程。这一解决方案使得不同企业之间能够发现和利用彼此的业务、合作伙伴所能提供的功能，并能不断发现新的潜在合作伙伴，了解这些潜在合作伙伴所能提供的功能，以及将电子商务与这些潜在合作伙伴进行无缝对接。该解决方案需要创建一个服务注册体系结构，使得企业可以采用一个全球的、平台独立的、开放的业务框架，从而使得这些企业能够：

1. 发现彼此的业务
2. 定义这些业务如何通过互联网进行交互
3. 共享全球注册资料库中的信息，从而加快电子商务在全球范围的推进

正如我们在 1.6.2.1 节中已经讨论的，通过在服务注册库中发布一个 Web Service，其他的应用将能发现该服务，这需要两个同样重要的操作：Web Service 的描述与注册。服务发布需要从业务、服务和技术方面对 Web Service 进行合适的描述。注册则涉及在 Web Service 注册库中持久化存储 Web Service 的描述。

服务注册主要关于服务的辨别和控制。在最简单的层次，服务注册记录企业所能提供的服务，以及那些服务的特性。电子商务注册通常有两类：基于文档的服务注册和基于元数据的服务注册。这些注册彼此之间的不同之处在于它们处理服务描述信息的方式不同。

在基于文档的服务注册中，通过在注册库中存储基于 XML 的服务文档，诸如业务概况或技术规范(包括服务的 WSDL 描述)，客户端可以发布信息。当这些描述文档提交给注册库时，服务提供者也必须以服务元数据的形式提供每一个文档的描述信息。因为元数据可用来描述不同系

统和流程中的信息结构，所以元数据是任何集成解决方案中的关键元素之一，具体可参见 13.2 节。元数据包括 XML 模式结构、接口定义、跨网络的端点位置、整个流程(例如创建一个新账户的流程、处理客户订单的流程)的详细描述。元数据存储在服务注册库中，描述文档也持久化存储在服务注册库的存储设施中。服务注册库通过元数据可向描述文档提供一些有意义的附件。服务注册库本身并不关心服务文档的具体内容。

在基于元数据的服务注册中，采用了不同的方法来处理与服务相关的一些信息。服务提供者提交包含服务信息的文档。然而，注册库并不会原样存储这些文档，而是获取服务文档中所包含的信息，然后创建元数据，这些元数据从本质上反映了所提交文档的内容。元数据按内部格式存储在服务注册库中。因此在这种情况下，服务注册将会关注所发布的信息。元数据可以包含对于注册库没有关注的内容的引用。然而，在这种情况下，注册库并不会管理这些文档，而是提供这些文档的链接。

概括地说，高级注册可以提供如下一些有价值的特性：

- 最大化复用 Web Service，并推动 SOA 解决方案中所有潜在用户的广泛使用。
- 创建一个支撑 SOA 实现的管理和控制结构。
- 包含 Web Service 及其相关对象的所有元数据，并包含有关服务提供者、客户以及他们之间关系的信息。
- 提供提供者、客户、系统管理员和操作员所需的通用接口和专用接口。
- 确保 SOA 能够处理不断增加的服务、客户数，并能适应业务需求的不断变化。

6.2 服务发现

服务发现(Service discovery)是 SOA 的一个重要基础。服务发现的实质就是确定 Web Service 提供者的位置，并获取已经发布的 Web Service 的描述。Web Service 发现需要确定 Web Service 的位置以及了解 Web Service 的定义，这是访问 Web Service 的一项基本工作。通过这一步骤，Web Service 客户端可以了解是否存在所需的特定的 Web Service，以及了解相关的 Web Service 的能力和如何与 Web Service 进行合适的交互。服务查询(Interrogating service)在注册库中查询满足服务请求者需求的 Web Service。查询由一些搜索条件组成，如所需的服务类型、首选价格、返回结果的最大数量。查询将对服务提供者所发布的信息进行搜索。在进行发现处理之后，服务开发者或者客户端应用将了解到 Web Service 的具体位置(所查找到的服务的 URI)、Web Service 的能力，以及如何与其进行交互。对于服务发现所返回的 Web Service 集合，服务选择(Service selection)将决定从中选择调用哪一个服务。

有两类服务发现：静态和动态[Graham 2004a]。静态服务发现通常发生在设计阶段，而动态发现则发生在运行时。

对于静态发现，在设计时就确定了服务实现细节(诸如网络位置、所使用的网络协议)，并从服务注册库中检索服务。设计者需要分析检索操作的结果，并将检索操作所返回的结果合并到应用逻辑中。

对于动态发现，在设计时并不确定具体的服务实现细节，而是在运行时确定这些细节。因此，Web Service 请求者必须指定首选项，以便应用能够推断请求者最有可能希望调用哪个或哪些 Web Service。在运行时，应用将在服务注册库上进行检索操作，以确定与应用所使用的服务接口定义相匹配的一个或多个服务实现定义。基于应用逻辑、QoS 事项(诸如最佳价格、性能、安全认证)等，应用选择最合适的服务，然后绑定到服务上，并调用该服务。

6.3 UDDI：统一描述、发现和集成

为了实现服务注册和发现，创建了通用描述、发现和集成(UDDI 规范)。UDDI 是一个跨行业的注册标准草案。基于该规范以及支持服务发布和发现处理的注册工具，可实现 Web Service 的描述和发现。UDDI 利用了万维网联盟(W3C)和互联网工程任务组(IETF)的一些标准，如 XML、HTTP 和 DNS 协议。UDDI 的目的就是供开发工具以及使用 Web Service 标准(诸如 SOAP/XML 和 WSDL)的应用使用。UDDI 提供了一个全球的、平台独立的、开放的框架，使得企业更容易开展业务、发现合作伙伴以及与这些合作伙伴在互联网上进行互操作。通过自动化地进行注册和查询处理，UDDI 使得服务提供者可以描述他们的全球化的、基于互联网的开放环境中的服务和业务流程，从而扩展他们的业务领域。这也使得服务客户端可以发现提供 Web Service 的企业相关信息，并可以发现这些企业所提供的 Web Service 的描述，以及发现 Web Service 接口和定义的信息，这些接口和定义信息可帮助企业基于互联网进行交互。

UDDI 是一个包含轻量级数据的注册库。作为注册库，它的主要目的是提供它所描述的资源(例如模式、接口定义和跨网络的端点)的网络地址。UDDI 草案的核心概念是 UDDI 业务注册库(UBR)，这是一个用来描述业务实体和它的 Web Service 的 XML 文档。从概念上说，UDDI 业务注册所提供的信息包含三个相关的组成部分："白页"、"黄页"和"绿页"。白页包括地址、联系方式以及其他的一些联系信息。黄页基于行业分类法对信息进行分类。绿页的内容则主要关于服务的业务能力和相关信息，包括对于 Web Service 规范的引用和指向各种基于文件和基于 URL 的发现基制的指针。使用 UDDI 注册库，企业可以发现潜在的合作伙伴以及有关这些合作伙伴的基本信息(通过白页)；可以发现按照具体行业进行分类的公司(通过黄页)；以及如何与提供服务的企业进行联系(通过绿页)。基于存储在 UDDI 注册库中的信息，应用和开发者可以确定：业务实体代表谁；它们做什么；所提供的服务位于哪里；以及如何访问这些服务。

UDDI 是按标准化方式进行设计的，并不受限于任何技术。换句话说，UDDI 注册库中的条目可以包含任何类型的资源，无论这些资源是否基于 XML。例如，在 UDDI 注册中，可以包含企业电子文档交换(EDI)系统的有关信息、DCOM 或 CORBA 接口的有关信息。甚至对于那些使用传真机作为主要通信方式的服务，它们的相关信息也可包含在 UDDI 注册中。这意味着，虽然 UDDI 使用 XML 表示它所存储的数据，但是也可以注册其他类型的技术。因为 UDDI 使用 SOAP 作为它的传输层，所以无论在设计的时候还是运行时，企业都可以使用基于 SOAP 的 XML API 调用与 UDDI 进行交互，从而发现企业服务的相关数据。因此，企业能够与服务提供者连接，并调用和使用服务提供者所提供的服务。

UDDI 注册库与目录或其他注册库的主要的不同点在于：UDDI 提供了按照分类法对业务和服务进行分类的一种机制。例如，服务提供者可以使用分类法来表明：服务实现了一个具体的领域标准，或者对于一个具体的地点提供了服务[Manes 2004]。由于 UDDI 采用了标准的分类系统，因此可以基于分类法来发现相关信息。基于分类，客户可以更容易地发现与他们的具体需求相匹配的服务。一旦完成 Web Service 的开发并将其部署后，在诸如 UDDI 这样的注册库中发布该服务就变成了一项重要的任务，以便一些潜在的客户和服务开发者能够发现该服务。

为了支持连接企业中各个部门的内联网和电子商务操作，业务也可以在内部设置多个私有的注册库。此外，具体业务也可使用它的客户和业务合作伙伴所提供的 UDDI 注册库。一旦完成 Web Service 的开发，并将其部署后，为了一些潜在的客户和服务开发者能够发现该服务，在公共的 UDDI 注册库中发布该服务将变得非常重要。通用业务注册库(UBR)是免费的，IBM、微软、SAP 和 NTT 运营这些公共注册库。对于可公开访问的可用 Web Service，UBR 提供了一个全球目

录。UBR 的角色类似于互联网基础设施中的 DNS(域名服务)，它使得用户能够定位业务、服务和服务规范。

处理 UDDI 全球注册库的公司称为运营者节点(operator node)。这些运营者管理和维护目录信息、提供业务信息的服务，以及其他的与目录相关的功能。这些节点提供了 UDDI 注册库的 Web 接口，可浏览、发布信息以及取回信息的发布。通过 UDDI 运营者，可以在 Web 上发布业务信息以及业务所提供的服务的相关信息。并且这些运营者都遵循良定义的复制模式。

UDDI 声明模型涉及标准化组织和发布可用服务描述的产业联盟。基本的 UDDI 用例模型如图 6.1 所示。一旦发布了可用服务的描述，服务提供者必须按照这些类型定义来实现和部署 Web Service。潜在客户可以基于不同的标准(如业务名、产品类别、实现特定的服务类型定义的服务者)来查询 UDDI 注册库。然后，这些客户端可以从所指定的位置了解服务类型定义的细节。最后，因为客户端已经获悉了服务端点，并获悉了如何和服务交换消息的有关信息，所以客户端可以调用所需的服务。

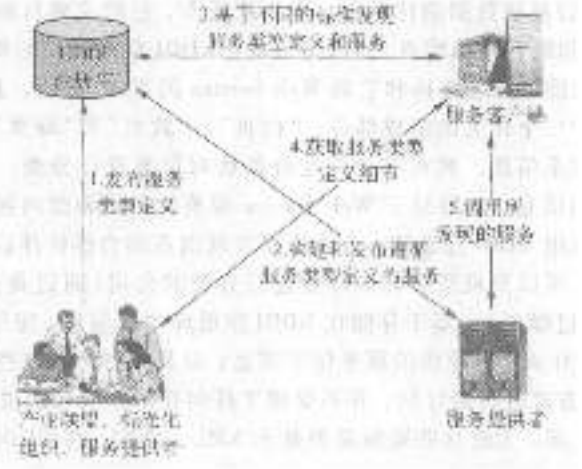


图 6.1 UDDI 用例模型

6.3.1 UDDI 数据结构

除了服务本身的技术规范以外，提供服务的其他相关信息也是很有意义的。UDDI 标准要包括 Web Service 的数据和元数据表示。通过 UDDI，企业能够发布和发现其他业务的信息，以及这些业务所提供的服务的相关信息。可以使用标准的分类学对这些信息进行分类，从而可以基于分类法发现这些信息。UDDI 也包含了企业服务的技术接口的相关信息。

无论是在公共域中还是在防火墙后面使用，UDDI 注册库都提供了对 Web Service 进行分类、编目和管理的机制，从而可以发现和使用那些 Web Service。无论是出于电子商务还是其他目的，无论是在设计的时候还是运行时，业务和提供者都能够以标准方式(诸如将查询发送到注册库)使用 UDDI 来表示 Web Service 的相关信息。在下列场合中，都可查询注册库[Bellwood 2003]：

- 发现 Web Service 实现，这些实现都是基于公共的抽象接口定义。
- 发现 Web Service 提供者，这些 Web Service 提供者都是按照分类模式或标识系统进行分类的。
- 基于高层的关键字搜索服务。
- 确定一个特定的 Web Service 所支持的安全性和传输协议。
- 存储 Web Service 的技术信息，并在运行时更新那些信息。

对于表示公司和服务描述信息，UDDI 定义了一个数据结构标准。在 XML 模式中定义了 UDDI 注册库所使用的数据模型。之所以选择 XML，主要是因为 XML 提供了一个平台中立的数据视图，并且在 XML 中还可以以中立方式来描述层次关系。UDDI XML 模式定义了提供白页、黄页、绿页功能的四类核心信息类型。这四类核心信息分别是：业务实体、业务服务、绑定模板，以及服务规范(技术或 tModel)的有关信息[OASIS 2004]。

图 6.2 提供了 UDDI 数据结构的一个高层视图。图 6.3 显示了 UML 表示的 UDDI 数据结构间的关系。图 6.2 中的图解显示了不同的 UDDI 结构、它们的子结构和属性间的关系。UDDI XML 模式指定了提供服务的有关业务实体(< business Entity >)(例如一个公司)的信息，描述业务所暴露的服务(< businessService >)，捕获使用服务所需的绑定信息(< binding Template >)。< bindingTemplate > 捕获服务端点地址，并将服务与表示服务的技术规范 < tModel > 关联起来。服务实现注册表示了一个具体的服务提供者所提供的服务。可以以一种或多种方式访问每一个 < businessService > 。例如，一个零售商可能将一个可访问的订单条目服务暴露为一个基于 SOAP 的服务、一个常规的 Web 表格甚至一个传真号。为了表达暴露服务的所有方式，每一个服务都通过绑定模板绑定到一个或多个 < tModel > 。为了进行标识，四个核心 UDDI 结构各自都有一个称为通用唯一标识符(UUID)的唯一键。

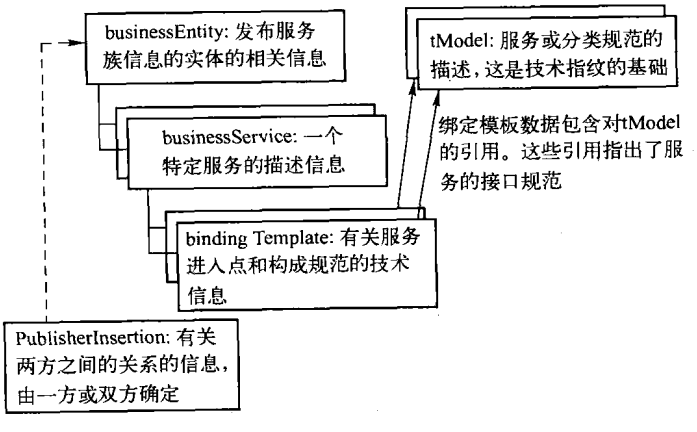


图 6.2 UDDI 数据结构概览

如图 6.3 所示，在 UDDI 数据结构中有一个层次关系。业务将发布包含一个或多个业务服务的业务实体。业务所提供的服务都有一些描述性的信息，并且这些服务都能有一个或多个绑定模板。绑定模板包含了如何访问服务进入点的相关信息。< tModel > 指向服务的规范或者接口定义。绑定模板包含了对 < tModel > 的引用(使用 < tModel > 键)。接口定义通常采用 WSDL 定义的形式。在 < tModel > 和 < bindingTemplate > 之间的关系通常是多对多的关系。因此，对于 < bindingTemplate > 来说，< tModel > 并不是唯一的。

1. 服务提供者信息

企业服务的合作伙伴或者潜在客户需要了解服务的位置信息以及服务提供者的相关信息。例如，他们需要知道业务名或者一些关键的标识符，以及一些可选择的类别和联系信息(白页)。图 6.4 描述了数据结构所记录的服务提供者信息，下面对其进行简要介绍。

businessEntity 元素和 businessKey 属性

在名为 < businessEntity > 的元素中包含了一些核心 XML 元素(UDDI Business Registry，UDDI 业务注册)，这些元素可支持业务信息的发布和发现。该 XML 元素作为顶层结构，包含了特定业

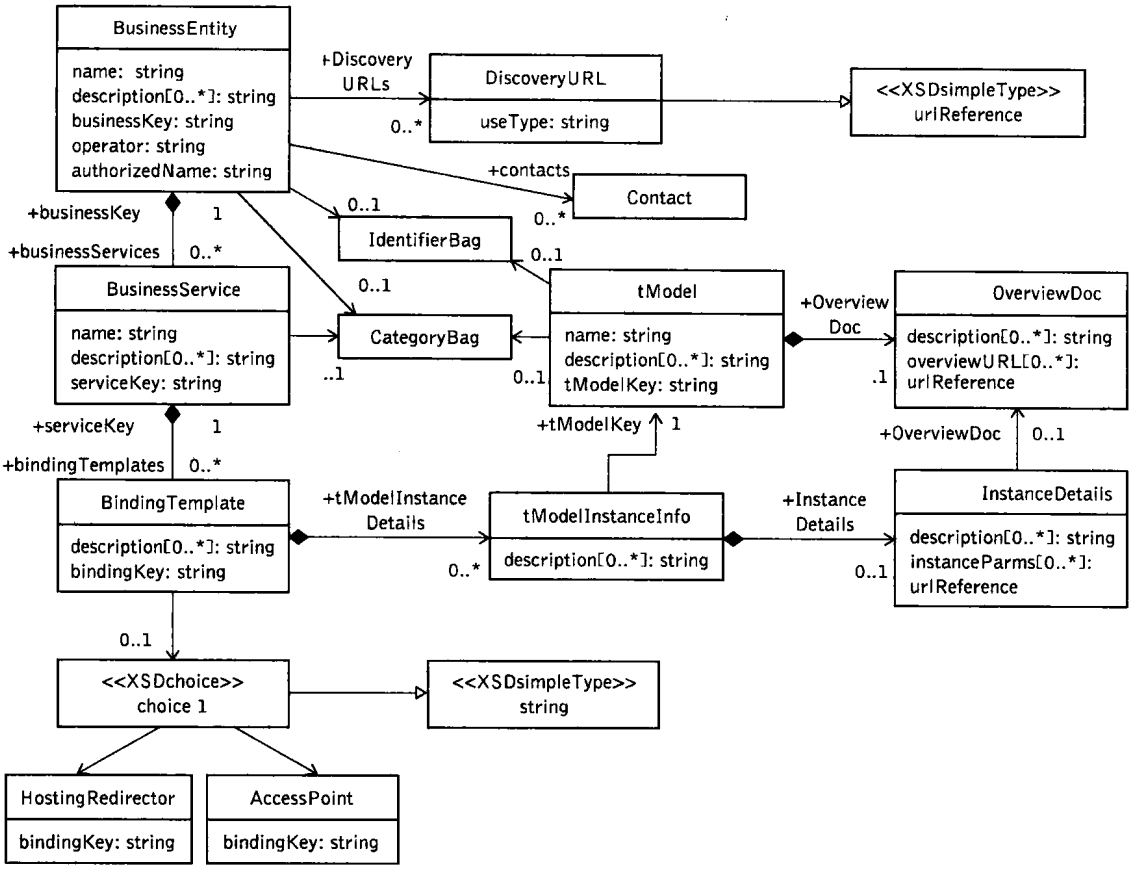


图 6.3 UDDI 结构间的关系的 UML 表示

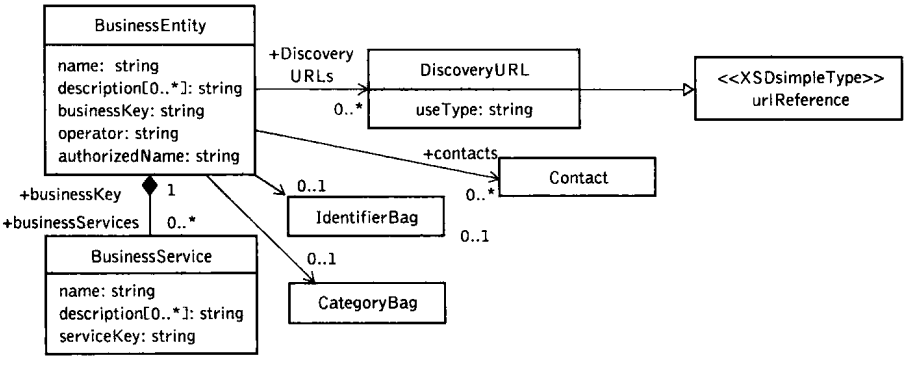


图 6.4 < businessEntity > 数据结构的 UML 表示

务单元(服务提供者)的白页信息。在 UDDI 中，可使用 < businessEntity > 结构对业务和提供者进行建模。< businessEntity > 结构包含了有关业务、提供者的描述信息，以及提供者所提供的服务的描述信息。这包括以多种语言表示的名字和描述信息、联系信息和分类信息等。

一些提供者元素与 < businessEntity > 实体所表示的组织相关。所有其他的非业务元素和这些提供者元素，诸如服务描述、技术信息，都可包含在 < businessEntity > 实体中或者被嵌套在 < bus-

inessEntity > 实体中的其他元素引用。例如，在 < businessEntity > 结构中可包含 < businessService >，< businessService > 描述了业务或者组织所提供的逻辑服务。类似地，< bindingTemplate > 包含在一个具体的 < businessEntity > 中，对于 < businessService > 所描述的逻辑服务，< bindingTemplate > 提供了该逻辑服务所包含的 Web Service 的技术描述。清单 6.1 是一个 < businessEntity > 数据结构的例子。该清单描述了 < businessEntity > 中的属性和元素。图 6.4 则使用 UML 表示了 < businessEntity > 数据结构及其内容。

清单 6.1 一个 < businessEntity > 结构的实例

```
<businessEntity businessKey="d2300-3aff-.."
                xmlns = "urn:uddi-org:api_v2">
<name xml: lang="en"> Automotive Equipment Manufacturing Inc.
</name>
<description xml: lang="en">
   Automotive Equipment, Accessories and Supplies for European firms
</description>
<contacts>
   <contact useType="Sales Contact">
      <description xml: lang="en"> Sales Representative
       </ description>
      <personName> Reginald Murphy </personName>
      <email useType="primary"> joe.murphy@automeq.com </email>
      <address useType="http">
         <addressLine> http://www.medeq.com/sales/ </addressLine>
      </address>
   </contact>
</contacts>
<businessServices>
   <!-- Business service information goes here -->
</businessServices>
<identifierBag>
   <!-- DUNS Number identifier System -->
   <keyedReference keyName="DUNS Number" keyValue="..."
    tModelKey="..."/>
</identifierBag>
<categoryBag>
   <!--North American Industry Classification System (NAICS) -->
   <keyedReference
            keyName="Automotive parts distribution" keyValue="..."
            tModelKey="..."/>
    ......
</categoryBag>
</businessEntity>
```

如清单 6.1 所示，< businessEntity > 包含 < businessKey > 元素，< businessKey > 是 < businessEntity > 的具有唯一性的业务标识符。< businessKey > 的属性值是 UUID。当第一次创建 < businessEntity >，UDDI 注册库自动生成该 UUID，并将其赋给 < businessEntity >。诸如 < businessService >、< bindingTemplate > 和 < tModel > 这样的实体也包含 UUID 键。基于清单 6.1 中的一个或多个 < businessEntity > 属性，潜在的合作伙伴(客户)可以搜索 UDDI，并确定匹配企业的位置。

discoveryURL 元素

这是一个可选元素。该元素包含一些 URL，这些 URL 指向其他的可 Web 寻址(通过 HTTP GET)的发现文档。

现举一个 < discoveryURL > 例子。假设 www. medeq. com 是一个可访问的 UDDI 节点，< businessKey > 的属性值是 uddi: medeq. com: registry: sales: 55，该属性值标识了 < businessEntity > 的发布者。UDDI 节点生成了 < discoveryURL >，而 < discoveryURL > 的值则是由 < businessEntity >

的发布者提供的。具体例示如下：

```
<discoveryURL useType="businessEntity">
   http://www.
   medeq.com?businessKey=uddi:example.com:registry:sales:55
</discoveryURL>
```

name 元素

name 元素包含了业务实体表示的组织的通用名称。< businessEntity > 可以包含多个名称。采用多个名称是有作用的，例如用于指定 < businessEntity > 的正式名称和缩写。在清单 6.1 中，属性 xml：lang = "en" 意味着：公司名称是用英语来表示的。

description 元素

这个元素是对业务的简短的叙述性说明。< businessEntity > 可以包含几个描述，例如使用不同的语言进行描述。

contacts 元素

该元素主要关于组织的联系信息，如清单 6.1 所示。它是一个可选元素，记录了 < businessEntity > 中的一个人或者一个工作角色的联系信息，从而以便他人进行联系。

businessService 元素

这是一个可选元素，它描述业务实体所提供的业务服务。该简单容器可包含一个或多个 < businessService > 实体，每一个 < businessService > 实体表示了 Web Service 实现。在下面的章节中将更详细地讨论 < businessService > 的数据结构。

identifierBag 元素

除了描述信息，UDDI 注册库还提供有关企业和它们的服务的相关信息。对于业务实体，UDDI 注册库也提供正式的标识符。UDDI 规范需要 UDDI 产品支持多个标识符系统，包括两个行业标准："邓百氏数据通用系统号码标识符系统(Dunn and Bradstreet's Data Universal System Number Identification System，简称 DUNS)"和"托马斯注册库供应商标识符代码系统(Thomas Registry Suppliers Identifier Code System)"。这些系统可帮助在 UDDI 注册库中查找公司，并且这些标识系统可以提供具有唯一性的供应商标识数。

< identifierBag > 也是可选的，它是一个名 - 值对列表，可充当公司的替代标识符，例如美国税务代码、诸如 DUNS 这样的业务标识符。在这个域中，一个公司可以有多个标识符。< identifierBag > 结构被建模为"tModelkey/keyName/keyValue"列表。每一个这样的三元组称为"键值标识的引用"。< identifierBag > 是一个或多个 < keyedReference > 结构的列表，每一个 < keyedReference > 结构表示了一个标识。

< tModel >(参见随后的"Web Service 访问与技术信息"一节)是一个数据结构，它标识了一个具体的分类法规范或分类系统规范[Monson-Haefel 2004]。每一个 < tModel > 都有一个具有唯一性的 < tModelkey > 元素。在 UDDI 注册库中，可使用 < tModelkey > 元素来查找 < tModel >。

< keyedReference > 包含三个属性：< tModelkey >、< keyName > 和 < keyValue >。< tModel > 元素表示了标识符系统。必须的属性 < tModelkey > 引用 < tModel > 元素，而必须的属性 < keyValue > 则包含系统中实际的标识符。例如使用 UBR 中的相应的 < tModelkey > 元素，通过 DUNS 号来标识 SAP AG，具体如下：

```
<identifierBag>
   <keyedReference
      tModelKey="uddi:uddi.org:ubr:identifier:dnb.com:d-u-n-s"
      keyName="SAP AG"
      keyValue="31-626-8655" />
</identifierBag>
```

categoryBag 元素

< categoryBag > 元素类似于 < identifierBag > 。 < categoryBag > 元素是一个或多个 < keyedReference > 结构的列表。 < keyedReference > 结构将业务实体标上了具体的分类信息，例如行业、产品或地区代码。通用标准产品和服务分类(Universal Standard Products and Services Classification，简称 UNSPSC)是一个行业分类器，它是一个开放的全球编码系统，用于对产品和服务进行分类。北美产业分类系统(North American Industry Classification System，简称 NAICS)类似地定义了业务和产品类别代码。可以使用 UNSPSC、地理分类器或者 NAICS 等对业务实体进行分类。

2. Web Service 描述信息

顶层实体 < businessEntity > 声明了一个称为 < businessServices > 实体的元素。 < businessEntity > 实体可以依次包含一个或多个 < businessService > 数据结构，其中每一个 < businessService > 构成都是一个 < businessEntity > 的逻辑后代。对于一个具体公司所提供的所有 Web Service，每一个 < businessService > 数据结构都表示了一个逻辑服务分类。并且每一个 < businessService > 数据结构都包含了业务术语描述信息。在 Web Service 信息层，并没有提供 Web Service 的技术信息，而是 < businessService > 提供了装配服务的能力。

< businessService > 结构是一个描述性的容器。对于一系列的与业务流程或服务类别相关的 Web Service，可以使用 < businessService > 结构对这些 Web Service 进行分组。 < businessService > 结构也可用于披露与服务相关的一些信息，诸如 Web Service 聚合名、Web Service 描述、或者类别细节等。包含相关的 Web Service 信息的业务流程的具体例子如订购服务、装运服务和其他的一些高层业务流程。对 < businessService > 信息集进一步分类，可将 Web Service 按照行业、产品和地区类别进行划分。每一个 < businessService > 概述了其中各个 Web Service 的作用。例如，一个 < businessService > 结构能够包含业务所提供的一个订购单 Web Service 集合(提交、确认和通知)。使用 UML 表示的 < businessService > 数据结构如图 6.5 所示。

一个 < businessService > 包含一个或多个 < bindingTemplate > 实体。 < businessService > 与 < bindingTemplate > 实体之间的关系类似于 WSDL < service > 与 WSDL < port > 元素之间的关系。 < bindingTemplate > 描述了 Web Service 端点，并表示了 Web Service 的“技术指纹”[Monson-Haefel 2004]。这意味着 < bindingTemplate > 列出了描述 Web Service 的所有 < tModel > 类型， < tModel > 唯一标识了 Web Service 技术规范。

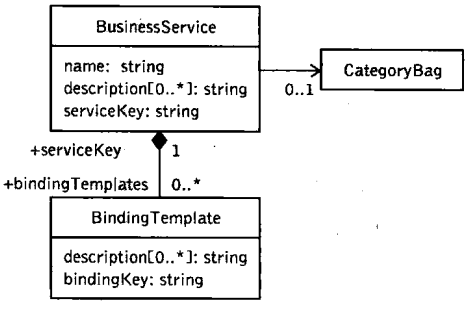


图 6.5 使用 UML 表示的 < businessService > 数据结构

包含在 < businessService > 元素中的信息映射到有关公司的“黄页”信息。正如 UDDI 中的任何其他数据结构一样，可以使用 XML 模式复合类型来描述 < businessService > 结构。清单 6.2 显示了 UDDI 注册库中的 < businessService > 实例，这个例子对应于在清单 6.1 中建模的公司。在清单 6.1 中没有显示该信息。正如清单所示， < businessService > 包含服务名、描述和分类信息(< categoryBag > 元素)。

清单 6.2 < businessService > 结构的样例

```
<businessServices>
   <businessService serviceKey=" ">
      <name> Search the Automotive Equipment Manufacturing parts
                                                Registry </name>
```

```
        <description lang="en">
           Get to the Automotive Equipment Manufacturing parts
           Registry
        </description>
        <bindingTemplates>
           <bindingTemplate bindingKey="..">
              <description lang="en">
                 Use your Web Browser to search the parts
                 registry
              </description>
              <accessPoint URLType="http">
                 http://www.automeq.com/b2b/actions/search.jsp
              </accessPoint>
              <tModelInstanceDetails>
                 <tModelInstanceInfo
                     tModelKey="uddi:.."/>
              <tModelInstanceDetails>
           </bindingTemplate>
        </bindingTemplates>
     </businessService>
  </businessServices>
```

通过搜索 UDDI，潜在合作伙伴可以确定特定行业或产品类别的服务所处的位置，或者确定某一具体地区有哪些服务。

< businessService > 实体的服务键唯一地标识了 < businessService > 实体。 < businessKey > 属性唯一地标识了 < businessEntity > 。 < businessEntity > 是 < businessService > 的提供者。当注册服务时，运营者节点指派了服务键。每一个 < businessService > 元素都包含在一个 < businessEntity > 中，参见图 6.3。有关 < bussinessService > 的简单的文本信息可以使用多种语言表示，具体内容包括它的名字和服务的简短描述。在 < businessService > 实体的名字中，xml: lang 的值具有唯一性，该值表示了所采用的语言。 < businessService > 数据结构中所包含的 < categoryBag > 元素与在 < businessEntity > 结构中所使用的类型相同。 < categoryBag > 包含了一个业务类别列表，每一个业务类别描述了 < businessService > 的一个具体的业务方面(例如行业、产品类别或地区)。一个特定的 < businessService > 包含了一个 < bindingTemplates > 元素。 < bindingTemplates > 元素是一个关于所提供的 Web Service 的技术描述列表。下面将更详细地介绍绑定模板。

3. Web Service 访问与技术信息

每一个 < businessService > 都可以包含多个 < bindingTemplate > 结构，每一个 < bindingTemplate > 结构描述了一个 Web Service(参见图 6.6)。更精确地，每一个 < bindingTemplate > 表示了一个不同的 Web Service < port > 或 < binding > 。 < bindingTemplate > 元素描述了调用服务所需的所有访问信息。与 < businessEntity > 和 < businessService > 结构相比， < bindingTemplate > 元素提供了应用绑定 Web Service(将服务名映射为它的 WSDL 描述)所需的技术信息，以及与所描述的 Web Service 进行交互所需的技术信息，而 < businessEntity > 和 < businessService > 结构则提供了有关提供者和服务的一些辅助信息。 < bindingTemplate > 元素必须包含下列两者之一：1)一个特定服务的接入点；2)通向接入点的间接途径。“绿页”数据是 Web Service 的技术描述，它驻留在 < businessService > 的 < bindingTemplate > 元素中。这些结构可帮助确定一个技术端点，或者支持远程驻留的服务。这些结构也支持一些轻量级设施，可帮助描述特定实现的那些具有唯一性的技术特性。 < bindingTemplate > 元素提供了对于技术、应用、具体的参数和设置的支持。有些客户端需要连接到远程 Web Service，然后与那些远程 Web Service 进行通信，并调用那些远程 Web Service。 < bindingTemplate > 元素中包含了与应用程序以及这些客户端相关的信息。

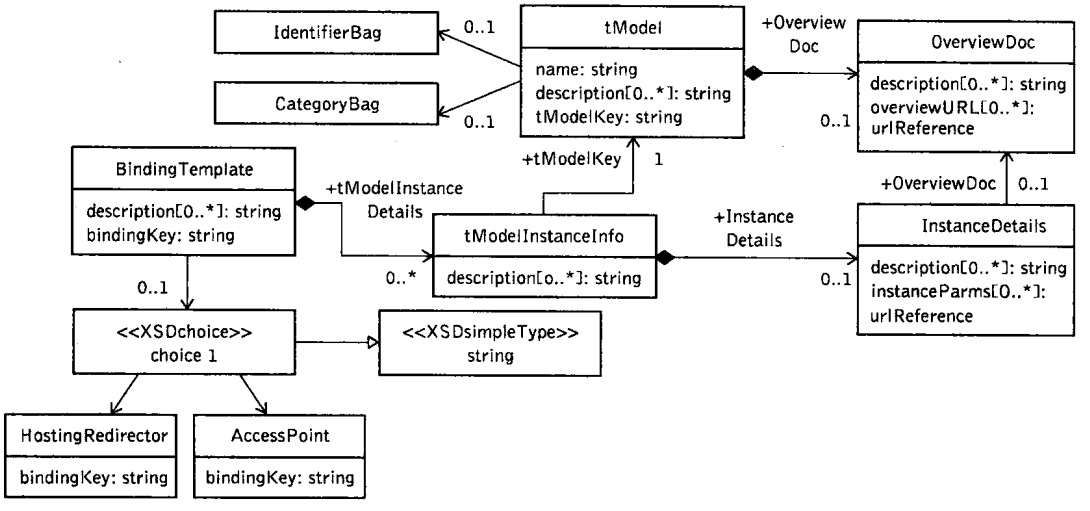


图 6.6 < bindingTemplate > 和 < tModel > 数据结构的 UML 表示

清单 6.3 显示了一个 < bindingTemplate > 结构，该 < bindingTemplate > 结构对应于清单 6.2 中的 < businessService > 结构。对于合作伙伴的数据格式和需求的详细信息，清单中的 < bindingTemplate > 提供了对它们的引用。

清单 6.3 一个 < bindingTemplate > 结构的样例

```
<bindingTemplate bindingKey="..">
   <description lang="en">
      Use your Web Browser to search the parts registry
   </description>
   <accessPoint URLType="http">
      http://www.automeq.com/b2b/actions/search.jsp
   </accessPoint>
   <tModelInstanceDetails>
       <tModelInstanceInfo
          tModelKey="uddi:.."/>
   <tModelInstanceDetails>
</bindingTemplate>
```

当定义 < bindingTemplate > 结构时，设计者声明 < accessPoint > 元素或 < hostingRidirector > 元素，但不能同时声明这两个元素。< accessPoint > 元素是一个指向服务进入点的属性指针。换句话说，< accessPoint > 元素提供了 Web Service 的精确的电子地址。有效的接入点值能够包括 URL、电子邮件、甚至一个电话号码。< accessPoint > 有一个 < URLType > 属性，可帮助搜索与特定服务类型相关的进入点。例如，订购单服务提供了三类进入点：一类针对 HTTP、一类针对 SMTP、一类针对通过传真进行的订购。一个 < bindingTemplate > 仅可以有一个 < accessPoint > 元素。假如可以通过多个 URL 访问一个 Web Service，则必须针对每一个 URL 定义一个不同的 < bindingTemplate > 结构。< hostingRidirector > 元素标识了实际的 < bindingTemplate > 元素，该实际的 < bindingTemplate > 元素指向最终提供所需的绑定信息的另一个 < bindingTemplate > 。假如对于一个特定的 < bindingTemplate > 有多个服务描述，或者服务驻留在远地，则重定向是非常有用的。

若仅简单地了解在何处可联系一个 Web Service，在许多时候是不够的。例如，假如我们知道一个服务提供者提供了一个接受订购单的服务，仅知道那个服务的 URL 是不够的。我们需要清楚地了解那个服务的相关技术细节，例如将被发送的订购单的格式是什么、哪些协议是合适的、

需要何种级别的安全性，以及发送订购单后所生成的响应的类型。通过 UDDI < tModel >（它是“technology model”的缩写），可以实现这一目标。< tModel > 提供了描述服务的技术细节的“绿页”信息。尤其，对于包含服务的 WSDL 描述信息的文件，< tModel > 包含（在 < overviewDoc > 元素中）了一个指向该文件的指针。正如 5.2 节所述，在服务规范中提供了这些技术细节。技术指纹的常规使用针对于 < bindingTemplate > 中的 Web Service WSDL。为了理解绑定模板和技术模型之间的关系，我们需要认识到：< businessService > 结构能够支持几个业务协议或规范（XML 词汇、EDI 标准、络世达网合作伙伴接口流程等），其中每一业务协议或规范都有一个单独的 < bindingTemplate > 。< bindingTemplate > 能够引用含有具体的 < tModel > 的每一个协议或规范。

当描述 Web Service 如何与它的客户端进行交互时，< tModel > 的主要作用就是提供一个技术规范。例如，就订购单来说，假如合适的文档格式一正确的方式发送到合适的地址，接受订购单的 Web Service 将会显现良定义的行为集。服务的 UDDI 注册由三部分组成：1）针对业务合作伙伴 < businessEntity > 的条目；2）描述订购服务 < businessService > 的逻辑服务条目；3）描述订购单服务的 < bindingTemplate > 条目。< bindingTemplate > 通过列举相应的 URL 以及引用 < tModel > 来描述服务，其中 < tModel > 用于提供服务接口和它的技术规范方面的信息。在订购单例子中，为了进行连接和交换数据，服务提供者系统需要进行一些软件设置，订购单 Web Service 包含了这些软件设置信息，而 < bindingTemplate > 中的 < tModel > 引用（ < tModelKey > ）是一个指向订购单 Web Service 细节信息的指针。通过 < tModel > 规范中的注册信息，订购单规范的设计者能够在 UDDI 注册库中建立一个唯一的技术标识。因此，< tModel > 就变成了一个技术指纹，对于一个特定的规范，该技术指纹具有唯一性。这涉及许多技术规范，诸如服务类型、绑定、连线协议或者如何开展业务的预订协议。一旦使用这种方式注册了，通过在技术服务描述 < bindingTemplate > 数据中引用 < tModel > 标识符（称作 < tModelKey > ），其他当事方将能够表示遵循规范的 Web Service 的可用性。对于与被引用的 < tModel > 兼容的服务，企业将暴露这些已经实现的 Web Service。因此，许多企业都可以提供遵循同一规范的 Web Service。当搜索那些遵循特定规范的注册服务时，这一方式将使搜索工作变得更容易，并且这一方式增强了不同软件系统之间的互操作性。

当在 UDDI 注册库中发布数据时，使用分类法是非常重要的。为了更容易发现业务、服务、绑定或者服务类型，可以使用一些类别来标记 UDDI 注册数据，并且这些类别需要能够很普遍地被搜索。为了实现这一点，< tModel > 定义了一个抽象的命名空间引用。可以使用它对业务实体、业务服务甚至 < tModel > 进行标识和分类。命名空间 < tModel > 表示了一个作用域。例如，通用标准产品及服务分类（简称 UNSPSC）是一个表示产品和服务类别的分类码。使用这个分类码，可以用一种比较正式的方式指定特定业务所提供产品和服务。无论是否使用标识系统（例如在全球范围内标识公司的 DUNS 数字系统）来唯一地标识信息，都可以使用标准代码（诸如 UNSPSC）来对业务进行分类，或者创建、分发一个新的分类法。businessEntity、businessService、bindingTemplate 和 tModel 元素这样的 UDDI 数据都必须是具有元数据的属性。

< tModel > 数据结构的 UML 表示如图 6.6 所示。< tModel > 结构的 < tModelKey > 属性唯一标识了一个特定的 < tModel > 结构。运营节点以类似于业务、服务和绑定键的方式指派了这些属性值。name 元素是 < tModel > 的名字。< description > 元素是对技术模型的简短的叙述性说明，对于不同的语言可以多次出现。< overviewDoc > 元素是对与 < tModel > 和 < overviewURL > 元素相关的远程命令或描述的引用。< overviewURL > 元素可以是任何有效的 URL，但是通常使用指向文件（例如服务的 WSDL 定义）的 URL，可以使用标准的 HTTP GET 操作获取该文件，或者使用通常的 Web 浏览器下载该文件。< identifierBag > 元素是一个可选项，它是一个名 - 值对列表，这

些名 - 值对用于记录 < tModel > 的标识数。 < categoryBag > 元素也是一个可选项，它也是一个名 - 值对列表，这些名 - 值对用于记录具体的分类信息，例如针对 < tModel > 的行业、产品或地区代码。在之前的“服务提供者信息”一节中已经描述了这些元素。

清单 6.4 显示了 UDDI 注册库中的 < tModel > 条目，使得遵循络世达网合作伙伴接口流程(RosettaNet PIP)的交易伙伴可以查询报价和提供报价。RosettaNet 定义了一套信息技术标准、一个电子商务组件和一个半导体生产和供应链(参见 14.3 节)。RosettaNet PIP 定义了交易伙伴间的业务流程。清单 6.4 引用了 PIP 3A1 “Request Quote”，使得开发者可以向提供者询问报价，提供者既可以直接提供报价作为响应，也可以进行转介。正如清单 6.4 所示，规范并不是存储在注册库中，< tModel > 有一个 < overviewDoc > 元素的 URL，该 URL 指向了规范的所在之处。

清单 6.4 < tModel > 条目的样例

```
<tModel tModelKey="…" >

 <name> RosettaNet-Org </name>
 <description xml:lang="en">
      Supports a process for trading partners to request and
        provides quotes
 </description >

 <overviewDoc>
    <description xml:lang="en">
         This compressed file contains the specification in a word
         document, the html guidelines document, and the XML schemas.
    </ description>
    <overviewURL>
          http://www.rosettanet.org/rosettanet/Doc/0/
          K96RPDQA97A1311M0304UQ4J39/3A1_RequestQuote.zip
    </overviewURL>
  </overviewDoc>

  <categoryBag>
      <keyedReference  keyName="Trading quote request and
                                                    provision"
                       keyValue="80101704" tModelKey=" …..."/>
  </categoryBag>
</tModel>
```

使用 UDDI 注册库存储 WSDL 服务的信息的最佳做法是：< bindingTemplate > 包含两个不同的 < tModelKey > 属性，这两个属性指向一个具体的 Web Service 的两个不同的 < tModel > [Kifer 2005]。一个文件包含了服务的 < portType > 的 WSDL 描述，另一个文件则包含了 < binding > 的 WSDL 描述，两个 < tModel > 条目分别指向这两个文件。对于一个具体的 Web Service 可以使用两个不同的 < tModel >，推荐这一做法的理由之一是：提供相同服务的多个业务可以共享服务的 < portType >。例如，可能有一个标准的订单管理服务，这个服务有一个通用的 < portType >，整个电子生产领域都可使用该 < portType >。电子生产领域的每一家企业都可以提供这一相同的服务，每一家都有它们自己的绑定。各个企业的服务的 < bindingTemplate > 指向相同的 < portType >，而对于不同的 < binding > 则指向不同的 < tModel >。这意味着所有的生产企业的订单服务都是语义上等价的，然而实现各不相同。一个具体的 Web Service 使用两个不同的 < tModel > 的另一个理由是：对于一个特定服务的不同 < binding >，同一提供者可能希望提供相同的 < portType >。

正如 WSDL 一样，UDDI 也对抽象和实现进行了明显的区分。事实上，正如我们已经看到的，< tModel > 提供技术指纹、元数据的抽象类型、接口。例如，规范概述了连线协议和交换格式。

例如，在 RosettaNet PIP、开放应用程序组集成规范（Open Application Group Integration Specification，简称 OAGIS）、各种 EDI 标准中都可以发现这种对抽线和实现的区分。

4. 发布者断言结构

因为许多企业的描述和发现很可能是不一样的，所以有时单个的 < businessEntity > 并不能有效地表示许多企业。例如，大型跨国企业有许多部门，对于他们所提供的 Web Service，这些部门可能需要创建他们自己的 UDDI 条目，但是仍然希望被视为该企业的一部分。因此，可以发布多个 < businessEntity > 结构，每一个 < businessEntity > 结构表示企业的一个部分或一个子公司。使用 < publisherAssertion > 结构可以实现这一目标。

< publisherAssertion > 结构定义了多个 < businessEntity > 结构之间的关系。两个（或多个）有关联的企业可以使用 < publisherAssertion > 结构来发布业务关系断言。对于双方来说，这些业务关系断言都是可以相互接受的。

清单 6.5 显示了一个 < publisherAssertion > 结构，该结构表示了一个企业和它的一个部门之间的关系。在清单中，< fromKey > 和 < toKey > 包含了两个关联公司的业务代码。< keyedReference > 元素中的 < tModelKey > 属性涉及关系的类型，例如业务合作伙伴、控股公司或者表示这些公司之间关系的特许经营。对于这类业务关系类型，UDDI 定义了一个公认的 < tModel > 。在公认的 < tModel > 中可以有三类有效的 < keyValue > ：父子（针对诸如控股公司和子公司这样的组织层次关系）、对等（诸如合作伙伴间或公司各部门间的平等地位）、同一性（指明两个业务实体都表示了同一家公司）。

清单 6.5 < publisherAssertion > 条目的样例

```
<publisherAssertion>
    <fromKey> FE565 … <\fromKey>
    <toKey> A237B   … <\toKey>
    <keyedReference tModelKey="uuid:807A .. " />
                    keyName="subsidiary"
                    keyValue="parent-child">
    </ keyedReference >
</publisherAssertion>
```

6.3.2 WSDL 到 UDDI 的映射模型

UDDI 和 WSDL 都清晰地、系统地刻画了接口和实现，因此它们之间可以相互补充、相互协作。通过解耦 WSDL 规范，并将它注册在 UDDI 中，我们能够使用标准接口（这些标准接口可以有多个实现）来构成 UDDI，业务应用从而可以共享接口。

WSDL 到 UDDI 的映射模型可帮助用户发现那些实现了标准定义的服务。映射模型描述了：WSDL < portType > 元素和 < binding > 元素规范如何变成 < tModel > ；WSDL 的 < port > 如何变成 UDDI < bindingTemplate > ；每一个 WSDL 服务如何注册为 < businessService > 。

如前所述，UDDI 提供了一个发布和发现服务描述的方法。对于 UDDI 业务和服务条目中的信息而言，在 WSDL 文档中定义的服务信息是对其的一个补充。UDDI 的目标是提供多种类型的服务描述，它并不直接支持 WSDL。然而，由于 UDDI 和 WSDL 都对接口和实现有着清晰的区分，它们可以协同工作。本节的重点是如何将 WSDL 服务描述映射到 UDDI 注册库，这将涉及目前已有的一些 Web Service 工具以及运行时环境。

在本节中，术语“WSDL 接口文件”表示一个包含 < types > 元素、< message > 元素和 < portType > 元素的 WSDL 文档；术语“WSDL 绑定文件”表示一个包含 < binding > 元素的 WSDL 文档；术语“WSDL 实现文件”表示一个包含 < service > 元素和 < port > 元素的 WSDL 文档。WSDL 实现文件导入接口文件和绑定文件，而绑定文件则导入接口文件。

一个完整的 WSDL 服务描述是服务接口、服务绑定和服务实现文档的组合。因为服务接口和服务绑定表示了可复用的服务定义，因此可在 UDDI 注册库中将它们作为 < tModel > 发布。服务实现描述了服务实例，每一个实例都是使用 WSDL < service > 元素来定义的，使用服务实现文档中的 < service > 元素可以发布 UDDI < businessService >，并且 WSDL 的服务 < port > 变成 UDDI 绑定模板。当发布 WSDL 服务描述时，服务接口必须作为 < tModel > 发布，然后将服务实现作为 < businessService > 发布。通过解耦 WSDL 规范，并将其注册在 UDDI 注册库中，我们能够使用标准接口(这些标准接口可以有多个实现)来构成 UDDI。图 6.7 概要表示了这种映射关系。我们可将该映射概括为两个主要步骤：服务接口的发布和服务的实现。

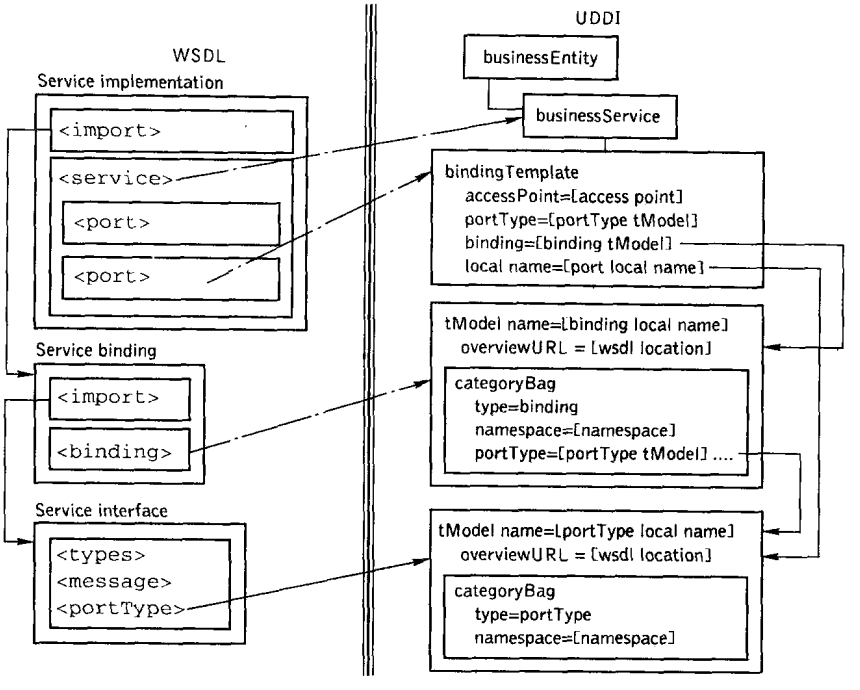


图 6.7 Mapping WSDL to UDDI schemas

1. 发布服务接口与服务绑定

当发布服务时，第一步就是创建服务接口定义。服务接口定义包括服务接口和协议绑定，需要能够公开访问它们。使用 UDDI 注册的任何 Web Service 都必须与 < tModel > 关联。< tModel > 描述了抽象接口，例如具体的 Web Service 所暴露的操作集。因为在 WSDL 中，< portType > 和 < binding > 可以有相同的名字，所以使用 < keyedReference > 来区分两个 < tModel >，其中一个 < tModel > 与 < portType > 相关，另一个与 < binding > 相关[Colgrave 2004]，[Colgrave 2003a]。

UDDI < tModel > 表示了 WSDL < portType >。为了与其他类型的 < portType > 进行区分，< tModel > 被归入 WSDL < portType tModel > 类别。在 < tModel > 中，仅存储有关 < portType > 的元数据。因为在 < portType > 中与消息、操作等相关的消息并没有复制在 UDDI 中，所以 < portType tModel > 必须引用定义 < portType > 的 WSDL 文档。有些应用开发工具能够根据 < portType > 生成编程语言接口，而有些复杂系统能够根据 < portType > 定义对请求进行验证，这些应用开发工具能够获取 WSDL 文档。

清单 6.6 包含了与清单 5.1 中所表示的 Web 服务接口定义相对应的 UDDI < portType tModel >。

清单 6.6 根据 WSDL < portType > 元素创建的 UDDI < tModel >

```
<tModel tModelKey="uuid:e8cf1163…" >
   <name>
      PurchaseOrderPortType
   </name>
   <overviewDoc>
      <!-- WSDL service interface definition -->
      <overviewURL>
         http://supply.com:8080/PurchaseOrderService.wsdl
      <overviewURL>
   <overviewDoc>
   <categoryBag>
      <keyedReference
         tModelKey="uuid:d01987d1…"
         keyName="portType namespace"
         keyValue="http://supply.com/PurchaseService/wsdl" />
      <keyedReference
         tModelKey="uuid:6e090afa…"
         keyName="WSDL type"
         keyValue="portType" />
   </categoryBag>
</tModel>
```

WSDL binding 实体映射到 < tModel >。< tModel > 名与 WSDL 绑定的本地名是一样的。< categoryBag > 指定了 WSDL 命名空间。< tModel > 包含了一个 < categoryBag >，这表明了 < tModel > 是一种对 < binding tModel > 与 < portType tModel > 进行区分的绑定类型。< categoryBag > 提供了一个指向 < portType tModel > 的指针，它表示了绑定所支持的协议。绑定 < tModel > 的名字就是绑定名。< binding tModel > 的 < categoryBag > 中的 < keyedReferenced > 表示了 < binding > 元素的命名空间。

由于在 UDDI 中并没有复制绑定中的细节，< binding tModel > 引用了定义绑定的 WSDL 文档。因此，需要绑定详细信息的应用开发工具能够从 WSDL 文档中获取相关信息。例如对于 < portType tModel >，可以使用 < overviewURL > 构成来指向 WSDL 文档的 URL。

清单 6.7 包含了与清单 5.2 中所表示的 Web 服务接口定义相对应的 UDDI < binding tModel >。

清单 6.7 根据 WSDL < binding > 元素创建的 UDDI < tModel >

```
<tModel tModelKey="uuid:49662926-f4a…">
<name>
      PurchaseOrderSOAPBinding
   </name>
   <overviewDoc>
      <overviewURL>
         http://supply.com:8080/PurchaseOrderService.wsdl
      <overviewURL>
   <overviewDoc>
   <categoryBag>
      <keyedReference
         tModelKey="uuid:d01987…."
         keyName="binding namespace"
         keyValue="http://supply.com/PurchaseService/wsdl" />
      <keyedReference
         tModelKey="uuid:6e090af…"
         keyName="WSDL type"
         keyValue="binding" />
      <keyedReference
         tModelKey="uuid:082b0851… "
```

```
            keyName="portType reference"
            keyValue="uuid:e8cf1…" />
         <keyedReference
            tModelKey="uuid:4dc741…"
            keyName="SOAP protocol"
            keyValue= "uuid:aa254…" />
         <keyedReference
            tModelKey="uuiu:e5c439…"
            keyName="HTTP transport"
            keyValue="uuid:68DE9…" />
         <keyedReference
            tModelKey="uuid:c1acf…"
            keyName="uddi-org:types"
            keyValue="wsdlSpec" />
      </categoryBag>
   </tModel>
```

对于使用 Web Service 的应用程序，可将 Web Service 注册为 < tModel >，从而提供了一种灵活的设计模式。例如，对于提供订购单的 Web 服务，可能有一个标准的接口。假设许多提供者可以创建实现了标准接口的服务，则客户端可以从 UDDI 注册库中搜索到提供这些服务的公司列表，并可基于一定的标准(诸如成本、响应时间等)选择最合适的 Web Service 提供者。此外，假如一个客户端使用了一个所选的服务提供者的订购单服务，并发现由于某些原因该服务不可用，则客户端应用可以动态地在 UDDI 注册库中查询实现了相同接口(例如，具有相同的 < tModel >)的另一家公司，并使用新的服务提供者的服务取代原先的服务。

2. 发布服务的实现

UDDI < businessService > 元素表示了一个 WSDL 服务，并且 WSDL 端口实体映射到 < bindingTemplate > 。假如 WSDL 服务表示了一个已有服务的 Web Service 接口，则可以有一个相关的、现有的 UDDI < businessService > 。因此，可将 WSDL 信息添加到已有的服务中。假如尚没有合适的服务，则可以创建一个新的 UDDI < businessService > 。必须部署这个新的服务，并将该服务注册到 UDDI 注册库中。既可以手动也可使用 WSDL 工具(或相关工具)创建 UDDI < businessService > 数据结构，然后注册该数据结构。新的 < businessService > 中包含的信息引用了已实现的行业标准，并提供了其他部署细节，诸如：

- 根据 WSDL 服务实现文档中的服务名生成 < businessService > 名。
- 对于每一个服务访问端点创建一个 < bindingTemplate > 。在服务实现中，< soap：address > 扩展元素的网络地址被编码在 < accessPoint > 元素中。
- 对于与所描述的服务端点相关的每一个 < tModel > ，在 < bindingTemplate > 中创建相应的 < tModelInstanceInfo > 。

UDDI < bindingTemplate > 表示了 WSDL < port > 。UDDL < businessService > 和它的 < bindingTemplate > 之间的包含关系精确地反映了 WSDL 服务和它的端口之间的包含关系。

清单 5.2 中的服务实现的 < businessService > 结构如清单 6.8 所示。

通过引用清单 6.6 中的订购单服务的 < portType > < tModelKey > (在 uuid：e8cf1163... 中)，以及引用清单 6.7 中的订购单服务的 < binding > < tModelKey > (在 uuid：49662926-f4a... 中)，注册在清单 6.8 中的服务显示了它的依从性。< businessService > 也包括了 < accessPoint > 元素。< accessPoint > 元素引用了服务的端点，或者所能访问的服务的位置。这对应了在 WSDL < service > 元素中指定的服务的网络地址，尤其是对应了清单 5.2 中的 SOAP < address > 可扩展性的 < location > 属性值。

清单 6.8 根据 WSDL 服务实现创建的 UDDI < businessService >

```
<businessService
      serviceKey="102b114a…"
      businessKey="1e65ea29…">
   <name> Purchase Order Service </name>
   <bindingTemplates>
      <bindingTemplate
            bindingKey="f793c521…"
            serviceKey="102b114a…">
         <accessPoint URLType="http">
            http://supply.com:8080/PurchaseOrderService
         </accessPoint>
         <tModelInstanceDetails>
            <tModelInstanceInfo
               tModelKey="uuid:49662926-f4a…">
               <description xml:lang="en">
                  The wsdl:binding that this wsdl:port implements.
                  The instanceParms specifies the port local name.
               </description>
               <instanceDetails>
                 <instanceParms> PurchaseOrderPort </instanceParms>
               </instanceDetails>
            </tModelInstanceInfo>
            <tModelInstanceInfo
               tModelKey="uuid:e8cf1163…">
               <description xml:lang="en">
                  The wsdl:portType that this wsdl:port implements.
              </description>
            </tModelInstanceInfo>
         </tModelInstanceDetails>
      </bindingTemplate>
   </bindingTemplates>
   <categoryBag>
      <keyedReference
         tModelKey="uuid:6e090afa…"
         keyName="WSDL type"
         keyValue="service" />
      <keyedReference
         tModelKey="uuid:d01987d1…"
         keyName="service namespace"
         keyValue=""http://supply.com/PurchaseService/wsdl" />
      <keyedReference
         tModelKey="uuid:2ec65201… "
         keyName="service local name"
         keyValue="PurchaseOrderService" />
   </categoryBag>
</businessService>
```

对于从 WSDL 服务接口和服务实现定义到合适的 UDDI 实体的映射，图 6.8 进行了概述，并概要描述了如何将服务接口(清单 5.1)和服务实现(清单 5.2)关联到 UDDI 模式实体，诸如清单 6.6 和清单 6.7 中的 < tModel > ，以及清单 6.8 中的 < businessService > 和 < bindingTemplate > 。

3. WSDL 到 UDDI 的映射模型小结

本节将对从 WSDL 到 UDDI 的映射模型进行总结。尤其是，表 6.1 到表 6.4 按照 UDDI 技术节点对 WSDL 到 UDDI 数据结构 < tModel > 、< businessService > 和 < bindingTemplate > 的映射进行了总结[Colgrave 2004]。

见表 6.1，必须将 wsdl：portType 映射到 < portType > 类别的 uddi：tModel。关于 wsdl：portType，至少需要了解的信息包括它的实体类型、本地名、命名空间和定义 < portType > 的 WSDL 文

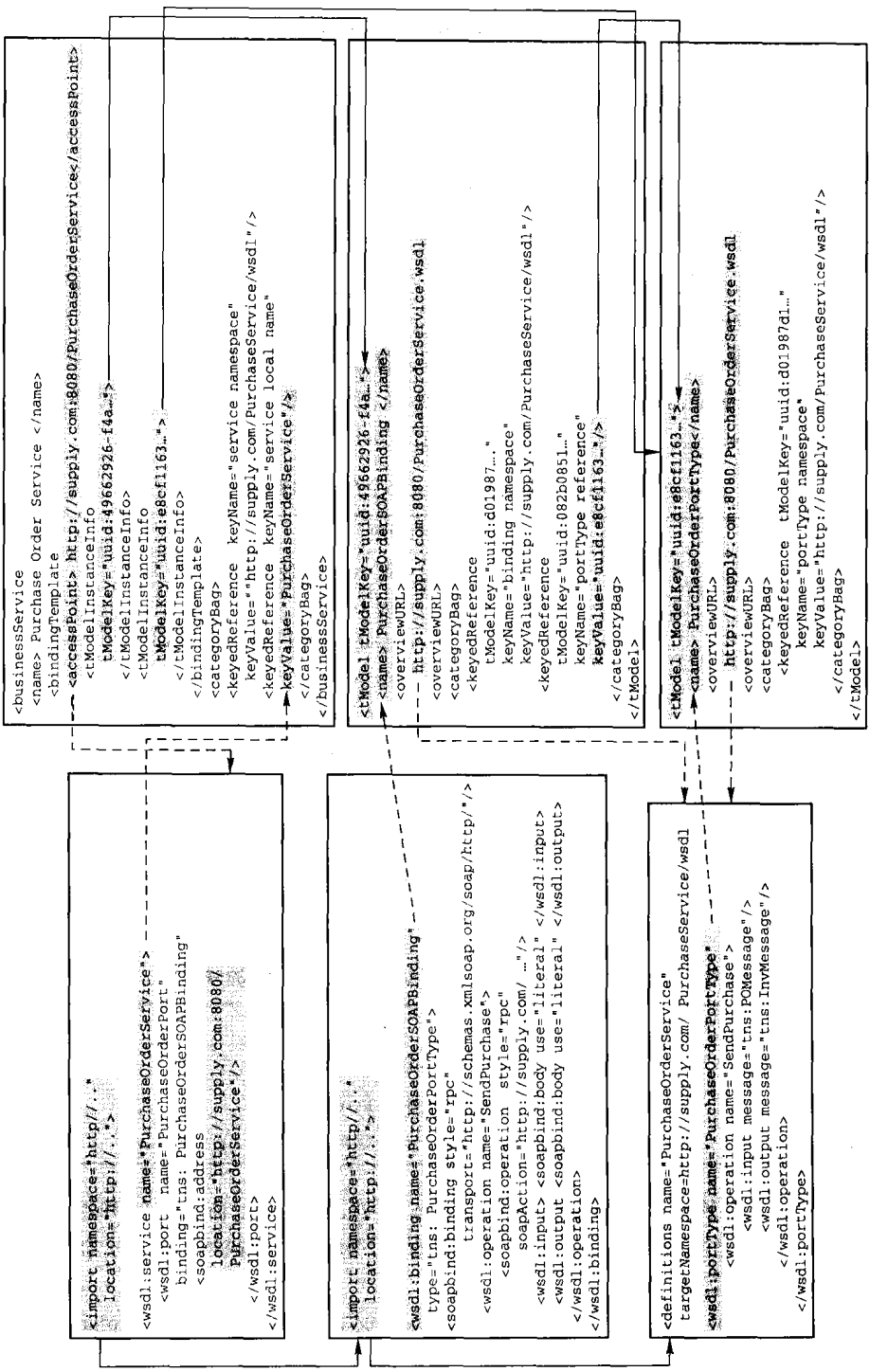


图6.8　从WSDL到UDDI的映射概览

档的位置[Colgrave 2004]。知道实体类型后，用户将可搜索表示 < portType > 的 < tModel > 。知道本地名、命名空间、和 WSDL 的位置，用户将可确定所指定的 portType 定义的位置。

表 6.1 从 wsdl: portType 到 uddi: tModel 的映射

WSDL	UDDI
portType	tModel(作为 portType 分类)
portType 的本地名	tModel 名字
portType 的命名空间	categoryBag 中的 keyedReference
WSDL 文档的位置	overviewURL

见表 6.2，必须将 wsdl: binding 建模为 < binding > 类别的 uddi: tModel。关于绑定，至少需要了解的信息包括它的实体类型、本地名、命名空间、定义绑定的 WSDL 文档的位置、实现的 < portType > 、协议、传输信息(可选)[Colgrave 2004]。知道实体类型后，用户将可搜索表示绑定的 < tModel > 。知道本地名、命名空间、和 WSDL 的位置，用户将可确定所指定的绑定定义的位置。通过链接到 < portType > ，用户可搜索实现了特定 < portType > 的绑定。

表 6.2 从 wsdl : binding 到 uddi : tModel 的映射

WSDL	UDDI
绑定	tModel(作为绑定和 wsdlSpec 分类)
绑定的本地名	tModel 名字
绑定的命名空间	categoryBag 中的 keyedReference
WSDL 文档的位置	overviewURL
portType 绑定关联到	categoryBag 中的 keyedReference
自绑定扩展的协议	categoryBag 中的 keyedReference
自绑定扩展的传输(假如有的话)	categoryBag 中的 keyedReference

见表 6.3，必须将 wsdl: service 建模为 uddi: businessService，可以使用已有的 < businessService > 或者创建一个新的 < businessService > 。一个 uddi: businessService 仅可以建模一个 wsdl: service。

表 6.3 从 wsdl: service 到 uddi: businessService 的映射

WSDL	UDDI
服务	businessService(作为服务分类)
服务的命名空间	categoryBag 中的 keyedReference
服务的本地名	categoryBag 中的 keyedReference，也可服务名

关于 wsdl: service，必须了解的信息包括它的实体类型、本地名、命名空间和所支持的各个端口[Colgrave 2004]。知道实体类型后，用户可以搜索 WSDL 定义所描述的服务。通过端口可以了解使用服务所需的技术信息。

见表 6.4， < businessService > 的 < bindingTemplate > 必须包括 < bindingTemplate > 元素，而 < bindingTemplate > 元素则建模了 < wsdl: service > 的端口。关于端口，至少需要了解的信息包括所实现的绑定、所实现的 < portType > 、已经它的本地名。知道绑定后，用户将可搜索实现一个特定 < portType > 的服务，而且无须了解服务所实现的具体绑定。

表 6.4 从 wsdl: port 到 uddi: bindingTemplate 的映射

WSDL	UDDI
端口	bindingtemplate
命名空间	从 keyedReference 中获取
本地端口名	与针对绑定的 tModel 相关的 tModelInstanceInfo 的 InstanceParms
通过端口实现的绑定	具有对应于绑定的 tModel 的 tModelKey 的 tModelInstanceInfo
通过端口实现的 portType	具有对应于 portType 的 tModel 的 tModelKey 的 tModelInstanceInfo

6.3.3 UDDI API

对于分类、编目和管理 Web Service，UDDI 注册库提供了一个标准方式，以便于能够发现和使用这些 Web Service。业务和提供者可以按标准方式使用 UDDI 来表示 Web Service 信息，从而可以在设计时或运行时查询 UDDI 注册库。UDDI 使用 SOAP 作为它的传输层。为了发现有关企业服务的技术数据，企业可以通过基于 SOAP 的 XML API 调用与 UDDI 注册库进行交互。

UDDI API 是一个接口，可以接受封装在 SOAP 信封中的 XML 消息[McKee 2001]。所有的 UDDI 交互都使用请求/响应模式。在请求/响应模式中，每一个消息请求 UDDI 注册库的服务，并生成一些类型的响应。UDDI 规范支持两类消息交换：查询和发布。可以使用查询 API 来搜索和读取 UDDI 注册库中的数据，并可使用发布 API 来添加、更新和删除 UDDI 注册库中的数据。

1. 查询 API

使用查询接口，请求者可以从 UDDI 注册库中获取信息。通过查询，企业可以发现满足一定要求的业务、服务或绑定(技术特性)。对于特定的查询，将会返回与搜索要求相匹配的 < businessEntity > 、< businessService > 或者 < bindingTemplate > 信息。UDDI 查询 API 有两类使用模式：浏览和下钻。例如，开发者可以使用浏览模式(发现 API 调用)来获取满足比较宽泛的查询标准的进入点、服务或者技术特性，然后使用下钻模式(获取 API 调用)来获取更具体的功能部件。例如，在一个具体的类别领域，可以通过 find_business 调用来定位各种业务，然后通过 get_BusinessDetail 调用来获取某一具体业务的更多的信息。

通过浏览模式，在注册库中可以搜索满足一定要求的数据结构。在浏览模式中，可以使用下列 5 个操作：find_business 、find_relatedBusiness、find_service、find_binding 和 find_tModel。

对于那些满足搜索要求的一个或多个 < businessEntity > 实体，通过 find_business 操作可以确定这些实体的位置。搜索要求可以包括类别、标识符、< tModel > 或 < discoveryURL > 等。可以通过业务名的一部分、业务标识符、类别标识符、服务的技术指纹等进行搜索。find_business 操作将会返回相关业务的一个简要列表，包括它们的键、名字、描述，以及 < businessService > 的名字和相关的键。

对于与业务实体相关的 < businessEntity > 注册，find_relatedBusiness 操作可以用来确定与业务实体相关的 < businessEntity > 注册信息的位置。对于所指定的企业，find_relatedBusiness 操作将返回与该企业具有可见的 < publisherAsertion > 关系的所有业务的列表。按照 < keyedReference > 元素，该操作可以在相关的所有业务集中进一步搜索一个子集。find_service 操作将可根据类别、< tModel > 或这两者进行搜索，并返回与搜索条件相匹配的所有业务服务入口的列表。在注册业务中，可使用 find_binding 操作来确定具体绑定的位置，该操作将返回 < tModel > 满足搜索条件的所有 < bindingTemplate > 。绑定模板含有调用服务所需的信息。find_tModel 操作将返回一个 < tModel > 列表，这些 < tModel > 将与请求消息中所列出的名字、标识符或类别相匹配。find_tModel 操作将返回 < tModel > 键的一个简要列表。

在下钻的 UDDI 用法模式中，可以根据唯一标识符来请求具体的数据结构。根据请求消息中的唯一标识符的数量不同，下钻操作或 get 操作将返回一个或多个类型的数据结构。

下钻模式使用下列 5 种方法：get_BusinessDetail、get_BusinessDetailExt、get_serviceDetail、get_bindingDetail 和 get_tModelDetail。get_BusinessDetail 方法根据具有唯一性的业务键来请求一个或多个 < businessEntity > 数据结构。对于一个或多个业务实体，该操作将返回整个 < businessEntity > 对象。get_BusinessDetailExt 操作等同于 get_BusinessDetail 操作，但是假如源注册库不是一个运营者节点，get_BusinessDetailExt 操作将会返回一些额外的属性。通过具有唯一性的服务键，get_serviceDetail 操作可请求一个或多个 < businessService > 数据结构。对于一个特定的业务服务，get_serviceDetail 操作将返回完整的 < businessService > 对象。通过具有唯一性的绑定键，get_bindingDeatil 操作可请求一个或多个 < bindingTemplate > 数据结构。get_bindingDeatil 操作返回运行时调用业务服务的方法所需的绑定信息（ < bindingTemplate > 结构）。最后，通过具有唯一性的 < tModel > 键，get_tModelDetail 操作可以请求一个或多个 < tModel > 数据结构，并将返回有关 < tModel > 的详细信息。

2. 发布 API

通过发布接口，企业可以存储和更新包含在 UDDI 注册库中的信息。UDDI 站点使用发布功能来管理提供给请求者的信息。通过发布 API，应用可以保存和删除本章前面所描述的 5 类 UDDI 数据结构：< businessEntity > 、< businessService > 、< bindingTemplate > 、< tModel > 和 < publisherAssertion > 。服务提供者和企业可以使用这些调用在 UDDI 注册库中发布（或取消发布）服务的有关信息。与查询 API 不同，使用这些发布 API 访问注册库需要得到授权[Cauldwell 2001]。

发布 API 支持四类操作：授权、保存和获取（get）操作。通过授权操作，客户端可以获得相应的访问权限、获取授权令牌、终止会话和它的授权令牌。通过保存操作，客户端可以添加或更新基本的 UDDI 数据结构。通过删除操作，客户端可以删除基本的 UDDI 数据结构。通过获取操作，客户端可以查看 < publisherAssertion > 。

有两类授权操作：get_authtoken 和 discard_authtoken。get_authtoken 操作将客户端记录到注册库中。discard_authtoken 终止会话，并从注册库中删除该客户端。为了开始一个 UDDI 发布会话，客户端必须首先和一个 UDDI 运营者建立一个 HTTPS 连接，然后发送一个 get_authtoken 消息，这个消息包含登录凭证。当客户端完成为 UDDI 发布端点的访问后，客户端将会发送一个 discard_authtoken 操作来终止它的会话。discard_authtoken 操作将命令注册库作废授权令牌，从而将无法继续使用该令牌进行访问。假如没有发送 discard_authtoken，则会话将会超时，授权令牌也将会作废。

通过保存操作，客户端可以在 UDDI 中添加或更新信息。每一个主要的 UDDI 数据结构都有一个相应的保存操作，不过 < publisherAssertion > 是例外。< publisherAssertion > 有一个专门的添加和设置操作，可以添加或更新一个或多个 < publisherAssertion > 。

通过删除操作，客户端可以在 UDDI 中删除信息。每一个主要的 UDDI 数据结构都有一个相应的删除操作，不过 < publisherAssertion > 是例外。通过获取操作，可以获取客户端所发布的数据结构的概要数据。

在 UDDI 中，分类法和标识符具有重要的作用。UDDI 使得新的业务标识符和分类法的注册更容易。第 2 版 UDDI 规范可以提供经过验证的分类和认证分类，因此 UDDI 运营者将可使用和管理具有验证的分类。在第 2 版 UDDI 中，注册者可以使用两种分类法对业务、服务和服务描述进行分类，有已核查的分类和没有核查的分类，以及认证分类[McKee 2001]。分类系统或标识符系统的提供者能够在 UDDI 中发布分类或业务标识符，并既允许对它进行不加限制的引用，也可

进行验证引用。允许不加限制引用的分类系统和标识符系统称为没有核查的系统(unchecked)。简单地注册一个新的 < tModel > 即可注册一个没有核查的分类，并且 < tModel > 既可以作为标识符进行分类，也可以作为分类法进行分类。反之，引用需要验证的分类系统或标识符系统称为可核查的系统(checked)。当分类的发布者需要确认类别码的值或注册的标识符所表示的信息时，将需要使用可核查的系统。NAICS、UNSPC 和 ISO 3166 都是可核查的分类，这意味着 UDDI 注册库必须验证客户端所提交的值。第 2 版 UDDI 支持第三方创建新的可核查的分类系统或标识符系统。注册一个可验证的系统将涉及两个当事方：1)提供分类系统或标识符系统的企业；2)发布分类法的 < tModel > 的 UDDI 注册运营者[McKee 2001]。

6.3.4 UDDI 模型的查询

本节所描述的样例查询基于查询 API。随着所开发的应用的类别不同，可以在设计/构建或者运行时进行查询。在设计/构建时通常选择 < portType >，假如需要的话，也将使用到一个特定的绑定，并根据绑定生成桩(stub)或类似的程序。在运行时，可搜索 < portType > 的实现，并且假如有 < binding > 也可以搜索。

本节中的查询基于文献[Colgrave 2003b]、[Colgrave 2004]中的查询样例，并且这些查询将遵循 6.3.1 节中的清单。在发现操作中所使用的条件将基于 < tModel > 的值(WSDL 和分类 < tModel >)、数据域和 UDDI 所使用的默认搜索规则的修饰符。

查询 6.1 进行了一个与查询谓词中所指定的名字匹配的业务的 OR 搜索(默认情况)。查询 6.1 返回一个业务实体，该业务实体的名字为 Automotive Equipment Manufacturing Inc. 并且 businessKey = "d2300-3aff..."。

查询 6.1 通过名字来查找业务实体

```
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
   <name xml: lang="en"> Automotive Equipment Manufacturing Inc.
                                                        </name>
   <name xml: lang="en"> Manufacturing Goods Co. </name>
</find_business>
```

查询 6.2 返回一个与 < keyedReference > 值匹配的业务的列表。在查询中，仅有那些声明了涉及 < categoryBag > 的 < keyedReference > 值的 < businessEntity > 实体才有可能匹配。查询 6.2 返回 business = "d2300-3aff..."(见清单 6.1)的业务实体。

查询 6.2 通过类别来查找 < businessEntity >

```
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">

<categoryBag>
   <!--North American Industry Classification System (NAICS) -->
   <keyedReference
            keyName="Automotive parts distribution"
            keyValue="..."
            tModelKey="..."/>
   ...
</categoryBag>
</find_business>
```

本节的余下部分将主要讨论涉及 < tModel > 的查询，这类查询主要是为了获取我们感兴趣的服务的技术信息。如前所述，在 UDDI 注册库中，所有的 WSDL 服务接口都是作为 < tModel > 发布的(参见图 6.5)。< tModel > 将被分类，并被标识为 WSDL 服务描述。使用 UDDI 查询 API 可以查找到 WSDL 服务接口描述。使用 UDDI find_tModel 消息可以检索到已分类的 < tModel >，该消息将返回一个 < tModel > 键的列表。使用下钻 get_tModelDetail 消息，应用和开发者将能检索到

一个具体的服务接口描述。例如，get_tModelDetail 消息能够返回一个诸如清单 6.6 和清单 6.7 中的 < tModel > 。为了限制 find_tModel 消息的响应消息中所返回的 < tModel > 集，可将附加的 < keyedReference > 添加到 < categoryBag > 中。在检索 < tModel > 后，可以使用概览 URL 来检索 WSDL 服务接口文档的内容[Brittenham 2001]。

对于命名空间“http://supply.com/PurchaseService/wsdl”中的 PurchaseOrderPortType，查询 6.3 是一个查找 < portType tModel > 的简单查询。该查询将返回 tModelKey = “uuid: e8cf1163”(见清单 6.6)。

查询 6.3 通过 < portType > 名查找 < tModel >

```
<find_tModel generic="2.0" xmlns="urn:uddi-org:api_v2">
   <name> PurchaseOrderPortType </name>
   <categoryBag>
      <keyedReference
         tModelKey="uuid:d01987d1…"
         keyName="portType namespace"
         keyValue="http://supply.com/PurchaseService/wsdl" />
      <keyedReference
         tModelKey="uuid:6e090afa…"
         keyName="WSDL type"
         keyValue="portType" />
   </categoryBag>
</tModel>
```

PurchaseOrderPortType 有一个相应的 < portType tModel > ，其键为 tModelKey = " uuid: 082b0851. . . " ，查询 6.4 查找 PurchaseOrderPortType 的所有 < binding tModel > ，并不管绑定中所指定的协议和/或传输。查询返回 tModelKey = " uuid: 49662926-f4a. . . " (见清单 6.7)。假如需要一个特定的协议和/或传输，由于 < keyedReference > 可表示消息协议(诸如 SOAP)和传输协议(诸如 HTTP)，因此可将额外的 < keyedReference > 添加到前面的查询表达式的谓词中。

查询 6.4 发现 PurchaseOrderPortType 的所有 < binding tModel >

```
<find_tModel generic="2.0" xmlns="urn:uddi-org:api_v2">
   <categoryBag>
      <keyedReference
         tModelKey="uuid:6e090afa…"
         keyName="WSDL type"
         keyValue="binding" />
      <keyedReference
         tModelKey="uuid:082b0851… "
         keyName="portType reference"
         keyValue="uuid:e8cf1…" />
   </categoryBag>
</tModel>
```

在诸如[Brittenham 2001]、[Colgrave 2003b]、[Colgrave 2004]等文献中，可以发现更多的有关将 WSDL 映射到 UDDI 的信息，以及有关 UDDI API 的信息。

6.3.5 UDDI 用例模型与部署的多样性

在 6.3 节中，我们已经讨论了 UDDI 基本可以分为两类：公开的 UDDI 注册库和私有的 UDDI 注册库。事实上，UDDI 用例模型假设了一些不同的业务信息提供者角色，诸如：

(1) 注册库运营者(registry operator)：这些指驻留和处理 UBR 的企业(更早的时候称为运营者节点)。运营者节点管理和维护目录信息，并且提供业务信息的复制以及其他与目录相关的功能。这些运营者提供了 UDDI 注册库的 Web 接口，可浏览、发布、撤销发布业务信息。企业并不需要分别注册到所有的这些运营者中，而是可以在任何一家运营者公司中注册，这可谓“一处注册，到处发布”。这意味着：客户端可以在任何一个注册库运营者节点搜索业务或服务，并且结

果应该是一样的。之所以如此，原因在于运营者节点注册库定期复制彼此的数据。

(2) 标准组织和行业协会：他们以服务类型定义(< tModel >)的形式发布描述。这些 < tModel > 并不包含实际的服务定义，而是含有一个 URL，该 URL 指向存储有服务描述的位置(可以按任何形式定义服务描述，然而 UDDI 推荐使用 WSDL)。

(3) 服务提供者：通常遵循 UDDI 所支持的服务类型定义来实现 Web Service。服务提供者在 UDDI 中发布有关业务和服务的信息。所发布的数据也包含这些企业提供的 Web Service 的端点。

UDDI 的结构允许各种私有 UDDI 节点。当前，可以有下列 UDDI 部署方式[Graham 2004a]：

(1) 电子交易市场 UDDI：电子交易市场是服务提供者和请求者的本地社区。请求者按垂直市场进行组织。电子交易市场是服务提供者和请求者的门户。电子交易市场、标准化组织、相互合作和竞争的行业联盟等都可驻留私有的 UDDI 节点。私有的 UDDI 的条目与一个特定的行业(或者一些行业)关联，从而产生了 Web Service 发现代理(或服务代理)这一概念。Web Service 发现代理是一个充当可信任的第三方的组织，它的主要作用就是驻留注册库，发布和宣传 Web Service。通过以下两个方面，服务发现代理可向 Web Service 请求者提供更强的搜索功能：1)在基础设施中添加广告功能；2)向服务提供者与服务请求者提供了一些工具，这些工具可增强服务提供者与服务请求者之间的 Web Service 匹配。电子交易市场节点能够提供一些增值服务，诸如 QoS 监测、对企业发布的内容的验证等，从而确保可以通过严格的选择程序来审查 UDDI 注册库中的各方，并且确保所有的条目都与细分市场相关。在这样的环境中，可将 API 所提供的发布操作和查找操作限制在电子交易集市中注册的合适的业务中。

(2) 业务合作伙伴 UDDI 注册库：上面模式的变体就是一个驻留在业务合作伙伴的防火墙后面的私有 UDDI 节点，并且仅有可信的或已通过审查的合作伙伴可以访问该注册库。它也包含可信的业务方(例如与驻留企业有正式协定/关系的那些企业)发布的 Web Service 描述元数据。

(3) 门户 UDDI：这种方式部署是在企业的防火墙上，并且是一个仅包含企业的 Web Service 的元数据的私有 UDDI 节点。在注册库中，可允许门户的外部用户调用查找操作(find)，然而仅可以对门户内部的服务进行发布操作。对于所使用的 Web Service，通过门户 UDDI，企业从根本上控制了元数据如何描述这些服务。例如，企业可以限制访问。企业也可以监控和管理对于数据的查找，以及可以获取对服务感兴趣的潜在用户的信息。

(4) 内部 UDDI：通过内部 UDDI，企业的不同部门中的应用可以发布和查找服务。因此对于大型企业来说，内部 UDDI 是很有用的。这类 UDDI 的主要特征是，它非常适合遵循标准(例如，使用固定的 tModel 集)的公共管理域应用。

与全球 UBR 相比，封闭的注册库具有一些优点。这类注册库不会限制如何描述注册库，因此企业可以通过各种方式来描述它的服务。它能够使用一个 URL 指向服务的文本描述、WSDL 中的描述或者公司使用的任何方式，从而增强了灵活性。因为应用不能基于查找操作的结果做一些实质性的事情，封闭的注册库严格限制了应用进行互操作的能力。相反，假如将 WSDL 应用于描述(元数据)，应用可以在服务中使用动态查找和绑定操作[Wahli 2004]。

6.4 小结

为了实现服务注册和发现，创建了通用描述发现和集成(UDDI)。UDDI 是一个跨行业的注册标准草案。基于该规范以及支持服务发布和发现处理的注册工具，可实现 Web Service 的描述和发现。对于服务描述、业务发现以及使用互联网集成业务服务，UDDI 提供了一个平台独立的方式。

UDDI 草案的核心概念是 UDDI 业务注册库(UBR)，这是一个用来描述业务实体和它的 Web Service 的 XML 文档。从概念上说，UDDI 业务注册所提供的信息包含三个相关的组成部分："白

页”、“黄页”和“绿页”。白页包括地址、联系方式以及其他的一些联系信息。黄页基于行业分类法对信息进行分类。绿页的内容则主要关于服务的业务能力和相关信息，包括对于 Web Service 规范的引用、指向不同文件的指针、基于 URL 的发现机制。对于基本的业务和服务信息的描述，UDDI 数据结构提供了一个框架，并且使用任何标准的服务描述方式，都可获取详细的服务访问信息。

在第 3 版的 UDDI 规范中，体系结构方面的最主要的变化就是“注册库交互”这一概念。这意味着：UDDI 将支持各种基础架构的置换。对于 UDDI，业务需要要求可采用更多的方式来定义多个 UDDI 注册库之间的关系，而不是简单地访问一个公开的业务服务注册库(即 UBR)。注册库交互涉及使用 UDDI 来支持各种网络/基础架构拓扑。可将独立式的、单注册库的方式进一步扩展为层次式、对等方式、委托方式等其他方式。简而言之，UDDI 注册库(一个或多个)的结构能够反应它所支持的基础的业务流程的实际情况以及流程间的相互关系。

复习题

- 什么是服务注册库？什么是服务发现？
- 什么是 UDDI？它的主要特性是什么？
- 什么是运营者节点？
- 使用 UML 描述 UDDI 数据结构以及它们间的相互关系？
- 什么是 < businessEntity > ？它的主要的子元素有哪些？
- 可以使用 UDDI 中的哪一个数据结构来描述 Web Service 访问信息？
- 什么是 < tModel > ？它是如何描述技术服务信息的？
- UDDI 是如何区分服务接口和服务实现的？
- WSDL 到 UDDI 的映射模型的目的是什么？
- 定义并描述 UDDI API 的概念。
- 描述如何查询 UDDI 模型。
- UDDI 可以有不同的部署方式，说明它的好处是什么。简要描述私有 UDDI 部署的两类重要方式。

练习

6.1 试举一个 < businessEntity > 和 < tModel > 实例。

6.2 试举一个与练习 6.1 中所指定的 < businessEntity > 相关的 < businessService > 实例。此外，试举一个与该 < businessService > 相关联的 < bindingTemplate > 实例。

6.3 对于清单 6.6 中的 PurchaseOrderPortType 的所有实现，编写一个查找它们的查询。该查询应该使用 find_service 和 find_binding 这两个 API。

6.4 假如我们希望查询 < businessService > 自身，这与将它们作为 < bindingTemplate > 的一部分进行查询是完全不同的。对于前者，可能需要使用服务的 WSDL 信息(服务名、服务的命名空间以及对应于 WSDL 服务的 < businessService >)、普通的 UDDI 信息、服务的主要类别。对于命名空间为“http://supply.com/PurchaseService/wsdl”的清单 6.6 中的 PurchaseOrderService，编写一个查找 < businessService > 的查询。

6.5 清单 5.1 和清单 5.2 描述了订购单服务。如何将这些 WSDL 文档分解为两个 < tModel > (一个用于 portType，一个用于绑定)和一个含有一个 < bindingTemplate > 的 < businessService > ？

6.6 对于 PurchaseOrderPortType，编写一个查找 portType < tModel > 和所有绑定的查询。

第四部分　事件通知与面向服务的体系结构

第7章　寻址与通知

学习目标

真实环境中的 SOA 实现需要依靠事件处理和通知。对于 SOA，这种形式的处理采用了通知模式，通过这一方式，提供信息的服务向一个或多个参与方发送消息。在这种模式中，消息通常载送有关所出现的事件的信息，而不是请求某个具体的操作。该模式通过基于事件的编程统一了 SOA 原理与概念。

在本章中，首先描述通知架构的一些关键组件，诸如处理有状态的资源的方式、寻址 Web Service 和消息的传输中立的方式。然后，我们主要讨论标准的 Web Service 通知，该方式使用基于主题的发布/订阅机制。通过本章的学习，读者将可掌握下列关键概念：

- 表示 Web Service 环境中的有状态的资源。
- 端点引用的概念。
- 对消息进行路由的方法以及对 Web Service 进行寻址的方法。
- Web Service 的事件处理与通知机制。
- WS-Notification 规范集。
- 对等通知模式和基于主题的通知模式。

本章中所介绍的内容主要是一些基础知识。有关事件驱动的 SOA 和企业服务总线的高级内容，可参见第 8 章。

7.1　Web Service 与有状态的资源

一般而言，易失性存储的计算可以描述为一系列的快照，称为状态（参见 1.4.3 节）。程序的状态指的是在相继的计算之间"记忆"信息的能力。过程（方法）调用通常会引起这些计算并会修改程序（或者对象）的状态。正如在第 1 章中所讨论的，交互涉及两类模式：有状态的和无状态的。有状态的交互记录方法调用之间的客户端的状态，而无状态的交互则不需要记录。

与程序类似，Web Service 也必须时常向用户提供访问和操纵状态的能力。状态涉及有关 Web Service 交互的数据值。通常使用诸如 Java Servlet、EJB 无状态会话 bean 等无状态的组件来实现 Web Service。虽然 Web Service 的实现通常是无状态的，但是 Web Service 接口通常必须支持状态操纵[Foster 2004]。例如，在线订单履行必须维护订单状态、具体顾客的订购状态或订单状态，以及系统本身的状态（系统的当前位置、负载和性能）。通过 Web Service 接口，请求者可查询订单状态、下单或取消订单、修改订单状态、管理订单处理系统。对于这些操作，Web Service

接口必须提供对状态的访问。

在 Web Service 领域，状态就是一些信息片段，并且这些信息不包含在 Web Service 请求消息的内容中，但是 Web Service 若要恰当地处理请求则需要用到这些信息。当 Web Service 成功地实现管理状态的应用时，为了能够以标准的方式发现、查询这些有状态的资源，以及与这些有状态的资源进行互操作，需要定义 Web Service 协定[Foster 2004]。构成状态的信息通常捆绑在一起，并称作有状态的资源。有状态的资源具有三个主要特性[Foster 2004]：1)它是一个具体的状态数据集。该数据集可表示为一个 XML 文档，并且这个 XML 文档定义了资源的类型。2)它有一个良定义的标识和生命周期。3)一个或多个 Web Service 都知道该资源，并能操纵该资源。

有状态的资源是一些能够被建模的元素，它们既可以是一些物理实体(诸如服务器)，也可以是一些逻辑构成(诸如业务协定和契约)。通过对有状态的资源的访问，客户可以了解业务效率方面的问题，诸如多供应商的及时采购、系统故障监测和恢复、工作负载平衡。有状态的资源也可以是一些有状态的资源的集合或分组。

在前面的定义中，涉及如何建模一个有状态的资源，但是没有涉及如何实现有状态的资源。可以将一个具体资源的状态实现为一个实际的 XML 文档，这个 XML 文档可以存储在内存中、文件系统中、数据库中或者 XML 信息库中。作为选择，同样的资源也可作为数据上的逻辑投影来实现，其中可以根据编程语言对象(例如 EJB 实体 bean)动态地构成或组合这些数据，或者通过私有通信在传统的过程式应用或数据系统上执行一些命令来获取这些数据。

对于有状态的资源，Web Service 应用主要关心以下几个方面[Graham 2004a]：

- 对于 Web Service 消息交换，如何将有状态的资源作为数据上下文使用。
- 如何创建和销毁有状态的资源，以及如何给有状态的资源赋予标识。
- 如何修改有状态的资源的元素。
- 为了使得良构的查询能够通过 Web Service 接口查询资源，如何将有状态的资源的类型定义关联到 Web Service 的接口描述。如何查询有状态的资源的状态以及如何通过 Web Service 消息交换修改有状态的资源的状态。
- 系统中的其他组件如何识别和引用有状态的资源。

Web Service 资源框架(WS-RF)处理以上所有方面，主要涉及有状态的资源的创建、寻址、检查和生命周期管理。WS-RF 的核心就是 Web Service 与有状态的资源之间的关系。

之所以制定 WS-RF 规范，有许多原因[Foster 2004]。WS-RF 规范最显著的贡献就是网格计算与 Web Service 的交汇，以及它们与 SOA 原理的融合。尤其是，WS-RF 定义了 Web Service 如何保持有状态的信息。WS-RF 旨在统一网格和 Web Service。通过与 Web Service 技术的融合，网格服务可以使用已有的 Web Service 标准，诸如 WS-Notification、WS-Addressing 和 WS-Security，此外网格服务还可进一步扩充能力，诸如服务状态数据、生命期、分组和引用管理。

7.2 Web Service 资源框架简介

WS-RF 是一个规范集，基于该规范集可使用 Web Service 建模有状态的资源(根据一些基本状态定义资源的行为)。该框架提供了将状态表示为有状态的资源的方法，并且该框架涉及一些著名的 Web Service 标准，诸如 XML、WSDL、WS-Addressing[Bosworth 2004]、WS-Security[Nadalin 2004]和 WS-Notification[Graham 2004b]。

WS-RF 涉及 6 个 Web Service 规范，它们定义了表示和管理状态的 WS-Resource 方法。表 7.1 列举了这些规范。在表 7.1 中，前 5 个规范合称为 WS-Resource Framework[Czajkowski 2004]，而 WS-Notification 规范族则主要针对通知(事件)订阅和发送。图 7.1 显示了涉及其他 Web Service

标准的 WS-RF。

表 7.1　WS-RF 规范

名　称	描　述
WS-ResourceProperties	描述产生 WS-Resource 的有状态的资源和 Web Service，以及描述如何检索、修改和删除 WS-Resource 的公开可检索性为元素
WS-ResourceLifetime	允许请求者立即或在预计的未来某一时刻销毁一个 WS-Resource
WS-RenewableReferences	在发现的端点引用作废后，用检索新的端点引用所需的信息来注释一个 WS-Addressing 端点引用
WS-ServiceGroup	通过 Web Service 引用异构地组织和使用异构的 Web Service 的引用集
WS-BaseFault	为报告错误时使用的基本故障类型
WS-Notification 规范族	标准消息方式，使用基于主题的发布与订阅模式

WS-RF 提供了将状态表示为有状态的资源的方法，并可按照隐含的资源模式规范化 Web Service 和有状态的资源之间的关系。隐含的资源模式是关于 Web Service 技术(如 XML、WSD 和 WS-Addressing)的一组约定。正如我们将 7.2.1 节中所讨论的，这组约定可帮助 Web Service 消息的路由，并可协助将这些消息发送到它们的真正的目的地。尤其，使用 WS-Addressing 可将有状态的资源与 Web Service 所实现的消息交换关联起来，术语隐含的资源模式描述了这一方法。隐含的资源模式定义了一个特定的状态资源如何与处理 Web Service 消息的方法关联起来[Czajkowski 2004]。在 Web Service 和一个或多个有状态的资源间可以建立一种特殊的关系，图 7.2 显示了隐含的资源模式如何帮助建立这一关系。

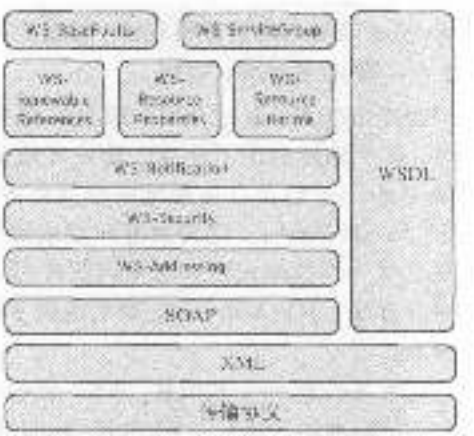


图 7.1　WS-RF 和其他 Web Service 标准之间的关系

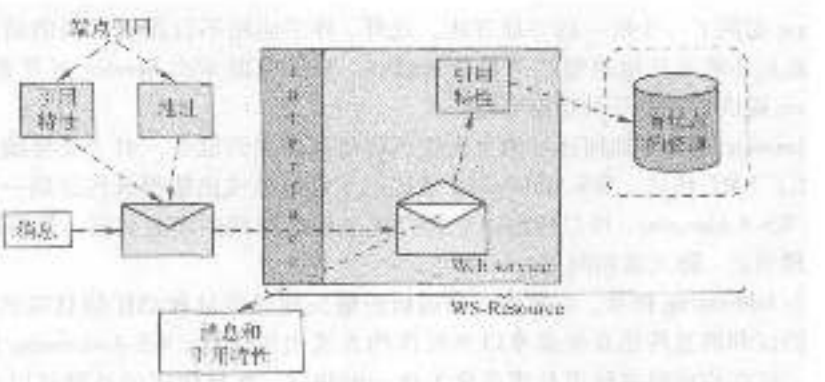


图 7.2　隐含的资源模式

之所以使用术语"隐含的(implied)"，其原因在于：与一个特定的消息交换相关联的有状态的资源的标识并不是请求消息的一部分。相反，有状态的资源隐含地与消息交换相关联。这就能静态地出现也能动态地出现[Foster 2004]。最后，术语"模式(pattern)"表明了：已有的 Web

Service 技术(如 XML、WSDL 和 WS-Addressing)的一组约定规范化了 Web Service 与有状态的资源之间的关系。隐含的资源模式中涉及的有状态的资源称作 WS-Resource，参见 7.2.2 节。WS-RF 描述了 WS-Resource 定义，也描述了如何通过 Web Service 接口访问 WS-Resource 的特性，以及描述了如何管理和了解 WS-Resource 的生命期。

当部署 Web Service 时，将有状态的资源与 Web Service 进行关联，则这种关联是静态的。另一种方式是在交换消息时进行关联，则这种关联称作动态关联。在进行动态关联后，可以使用 WS-Addressing 端点引用在端点寻址目标 Web Service，在 WS-Addressing 端点引用中可以封装的隐含的有状态的资源，而通过有状态的资源标识符则可指派所封装的有状态的资源。Web Service 端点引用(下一节有更详细的描述)等同于 Web Service 指针的 XML 表示。

在消息上下文(使用 WS-Addressing 引用特性)中，端点引用是一个与隐含的资源标识符进行消息交换的模式，端点引用可以唯一地标识与之通信的资源，参见图 7.2。

下面我们将要描述 WS-Addressing 在 WS-Resource Framework 中所扮演的角色，尤其是 WS-Addressing 与隐含的资源模式这一概念之间的关系。然后，将继续讨论 WS-RF。

7.2.1 Web Service 寻址

通常情况下，通过 WSDL 提供的服务端点信息调用 Web Service。例如，WSDL 有一个定位地址，这个地址标识了端点。在许多情况下，这个信息是恰当的，不过也有例外，例如有状态的 Web Service 以及在向地址中添加更多的动态信息(包括实例信息、策略、复合绑定等)的情况下[Joseph 2004]。这需要客户端或运行时系统在运行时统一标识服务。具体的绑定信息包括具有唯一性的标识符。当访问服务时，当前并没有交换信息以及将信息映射到运行时引擎的标准方法。WS-Addressing 规范提供了标识和描述端点信息以及将消息映射到 SOAP 消息头的方法。

对于 Web Service 和消息，WS-Addressing 是传输中立的寻址方式。为了对 Web Service 指针的概念进行标准化，引入了 WS-Addressing[Graham 2004a]。WS-Addressing 的主要目的是将消息寻址信息合并到 Web Service 消息中。SOAP 本身并没有提供任何识别端点的部件。诸如消息目的地、故障目的地和消息中介这样的常规端点都被委托到传输层。将 WS-Addressing 与 SOAP 结合起来就生成了一个真正的面向消息的规范。对于基于同步和/或异步传输的 SOAP 消息传送，WS-Addressing 提供了一个统一的寻址方法。此外，许多应用不仅需要常规的请求和响应模式的消息交换，而且还需要其他类型的消息传送模式，为了帮助 Web Service 开发者构建这类应用，WS-Addressing 提供了一些寻址功能部件。

WS-Addressing 定义了如何通过消息头信息将消息传送到服务。对于交换端点引用，WS-Addressing 提供了 XML 格式。WS-Addressing 还提供了将回复或出错消息传送到一个具体的地点的方法。通过 WS-Addressing，可以传输中立的方式实现跨网络的消息传送，如跨越诸如端点管理器这样的处理节点、防火墙和网关。

根据 WS-Addressing 标准，通常嵌入在通信传输头部的寻址和动作信息应该放置在 SOAP 信封中。传输协议和消息传送系统通常以互操作的方式提供信息，WS-Addressing 定义了两类传送消息的构成。这些构成将这种消息规范化为统一的格式，并且格式的处理可以独立于传输或应用。这两个构成是端点引用和消息信息头部[Bosworth 2004]。

对于称作端点引用的便携式地址构成，WS-Addressing 标准定义了一个模式。端点引用提供了目标的地址，而不是目标的标识，并且端点引用是作为 XML 类型来定义的。地址构成是一个 URI，用于提供端点的逻辑地址，并且每一个目标为那个端点的 SOAP 消息的头块中都包含该 URI。

运行时和端点交互需要三类重要的信息：基地址、引用特性集和引用参数。通过这三类信息，WS-Addressing 中的服务端点遵循隐含的资源模式。引用特性和引用参数是任意的 XML 元素的集合。通过提供有关消息的附加的路由或处理信息，这些 XML 可补充基地址构成。引用特性帮助寻址 WSDL 实体集，这些 WSDL 实体集共享一个公共的 URL 和作用域。端点引用必须包含地址以及元数据描述，诸如服务名、端口名、端口类型以及一些 WS-Policy 语句，这些 WS-Policy 语句描述了需求、能力以及服务偏好。通过这些特性可以发现契约细节和策略。

为了和端点进行交互，对于发送到相关的 Web Service 端点的消息，WS-Addressing 在 SOAP 消息中定义了一组标头。更具体地说，为了动态地定义不同的端点之间消息流，WS-Addressing 定义了四个标头(ReplyTo、FaultTo、RelatesTo、MessageID)。这使得 SOAP 消息不仅进一步独立于它所使用的通信协议，而且 Web Service 向它们自己以及其他服务传递引用的方式。WS-Addressing 标头如下面以及清单 7.1 所示。

清单 7.1　一个典型的使用 WS-Addressing 的 SOAP 消息

```
<Soap:Envelope xmlns:Soap="http://www.w3.org/2003/05/soap-envelope"
               xmlns:wsa="http://www.w3.org/2004/12/addressing">
    <Soap:Header>
        <wsa:MessageID>
            uuid:SomeUniqueMessageIdString
        </wsa:MessageID>
        <wsa:ReplyTo>
            <!-- Endpoint reference for intended receiver of
             message reply -->
            <wsa:Address> http://myclient.com/business/someClient
            </wsa:Address>
        </wsa:ReplyTo>
        <!-- Endpoint reference for ultimate receiver of message -->
        <wsa:To> http://www.plastics_supply.com/purchasing
        </wsa:To>
        <wsa:Action> http://www.plastics_supply.com/SubmitPO
        </wsa:Action>
    </Soap:Header>
    <Soap:Body>
         <!-- The message body of the SOAP request appears here -->
         <SubmitPO> … </SubmitPO>
    </Soap:Body>
</Soap:Envelope>
```

对于从站点 http://myclient.com/business/someClient 发送到站点 http://www.plastics_supply.com/purchasing 的 SOAP 消息，清单 7.1 中的样例代码说明了这些 WS-Addressing 方法的使用。

在清单 7.1 中，SOAP 头部中所包含的一些行首先指定了消息的标识符以及该消息所需回复的端点的标识符，回复消息将作为一个端点引用进行发送。构成 < wsa：Form > 指定了一个 URI，该 URI 唯一地标识了一个消息。构成 < wsa：From > 指定了消息所源自的端点，而构成 < wsa：ReplyTo > 则指定了一个端点引用，该端点引用标识了消息的意向接收者。假如预期有回复，消息必须包含 < wsa：ReplyTo > 头部。发送者必须使用 < wsa：ReplyTo > 头部的内容来格式化响应消息。假如没有 < wsa：ReplyTo > 头部，则可以使用 < wsa：From > 头部的内容来格式化发送到源端的消息。假如消息没有回复，则也可以没有该特性。SOAP 头部中的最后两条语句指定了消息最终接收者的 URI 地址和一个 < wsa：Action > 元素。消息(例如订购单的提交)蕴含了动作语义，而 < wsa：Action > 元素则是一个标识符，该标识符唯一地标识了动作语义。

清单7.2显示了一个WS-Addressing端点引用样例。在<Address>元素中可以指定服务的URI，并且该URI将被展示给第5章所指定的订购单服务(清单5.1和清单5.2)。<Address>元素包含了Web Service的具体传输地址。在该例中是一个HTTP URL。

清单7.2 指定一个端点引用

```
<wsa:EndpointReference
   xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
   xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
   xmlns:tns="http://supply.com/PurchaseService/wsdl"
>
   <wsa:Address> http://supply.com/PurchaseService/wsdl
   </wsa:Address>
   <wsa:PortType> tns:PurchaseOrderPortType </wsa:PortType>
   <wsa:ServiceName PortName="tns:PurchaseOrderPort">
        tns:PurchaseOrderService
   </wsa:ServiceName>
   <wsa:ReferenceProperties>
       <tns:CustomerServiceLevel> Premium
       </tns:CustomerServiceLevel>
   </wsa:ReferenceProperties>
   <wsp:Policy>
       <!-- policy statement omitted for brevity -->
   </wsp:Policy>
</wsa:EndpointReference>
```

如图7.2所示，使用<EndpointReference>可以进行消息交换，WS-Addressing <EndpointReference>包括一个<ReferenceProperties>子元素，该子元素标识了和所有消息交换相关联的资源。在图7.2中，使用引用特性标识服务所部署的端点。WS-Addressing <ReferenceProperty>元素表明了：引用可以包含许多特性，对所传输的实体或资源进行标识时需要用到这些特性。例如，清单7.2中的引用特性表示了：对于高级客户，订购单服务可以提供一个折扣价。

两个端点引用共享相同的URI，但是它们分别指定了表示了不同的引用特性值，这两个特性值表示了两个不同的服务。引用特性用于将请求派遣到合适的服务。例如，一个应用可能部署了两个不同版本的服务。在应用中，请求可以在它们的引用参数中指定目标版本。其中一个服务版本针对普通顾客，而另一个版本则针对高级客户。

最后，消费应用程序可以在端点中包括一些策略，从而更容易进行处理。使用清单7.2中的<Policy>元素可以实现这一点。<Policy>元素按照WS-Policy规范描述了端点的行为、需求和一些非功能性的能力。

虽然WS-Addressing独立于其他的Web Service规范，但是它可以和其他的Web Service规范一起使用。更具体地说，WS-Addressing扩展了一些WSDL中的一些概念，它可以标识Web Service端点的完整描述，这个关系如图7.3所示。图7.3描述了在端点引用、WSDL构成和SOAP消息的WS-Addressing头部中的数据结构间的连接。如图所示，类似于在WSDL中，WS-Addressing使用了<Service>构成和<portType>构成。类似于在WSDL中，在WS-Addressing中<Service>名和<PortType>名是QName(限定名)。在WS-Addressing端点引用中的<Service>名和<PortType>名与WSDL是兼容的，而不是用来完全取代WSDL。在清单7.2中，位于http://supply.com/PurchaseService/wsdl的Web Service实现了<.PortType>，<.PortType>元素给出了自身的名字，如PurchaseOrderPortType。最后，清单7.2和图7.3表示了：为了识别与一个具体的Web Service相关联的一些非功能性特性，WS-Addressing与WS-Policy进行相互协作。WS-Addressing也能与其他Web Service标准配套使用。例如，应用可以使用WS-Addressing来识别消息的源端和目的地，应用也可以使用WS-Security来进行从源端到目的地的认证。



图 7.3 嵌入在 SOAP 消息中的 WS-Addressing 消息头部

7.2.2 Web Service 资源

WS-Resource 被定义为 Web Service 和有状态的资源的组合[Czajkowski 2004]。该组合表示了 XML 文档和具有 Web Service < PortType > 元素的类型之间的关联，并且可以按照聚合资源模式寻址和访问这种结合。

如图 7.4 所示，通过 Web Service 接口，可以访问 WS-Resource 的特性。基于 WS-Resource 的构成，可以对 Web Service 与有状态的资源之间的关系进行编码。WS-Resource 可有一个网络范围的端点引用，端点引用可标识 Web Service 和一组引用特性。通过 WS-Addressing 引用特性，可唯一地标识具体的资源。例如，图 7.4 显示了与一个 Web Service 相关的三个有状态的资源。WS-

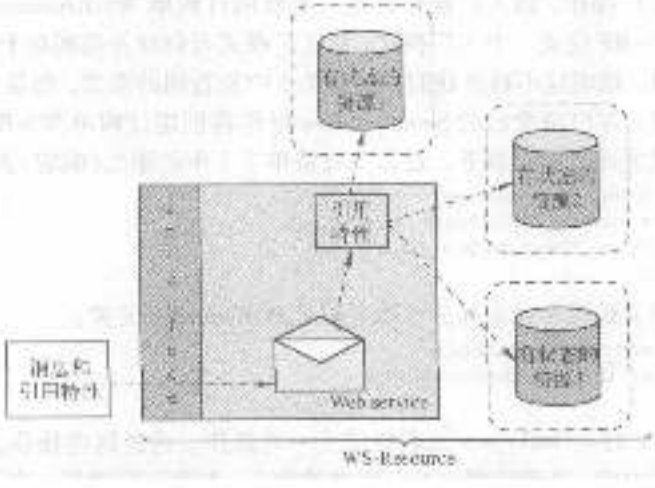


图 7.4 Web Service 和相关的 WS-Resource

Addressing 中的引用特性组件携带了一个有状态的资源的标识符，通过这个标识符可以识别 WS-Resource 的有状态的资源组件。

每一个 WS-Resource 都有至少一种形式的标记，可在访问 WS-Resource 的 Web Service 的上下文中唯一地标识 WS-Resource[Foster 2004]。这种标记与 Web Service 的标记并不相同，但是通过将 WS-Resource 的本地标记信息添加到 WS-Addressing 端点引用的引用特性中，Web Service 可以构成一个相关的 WS-Resource 的地址。这样的端点引用是受限制的 WS-Resource。在分布式系统中，其他的实体可以使用受限制的 WS-Resource 端点引用，然后可使用受限制的 WS-Resource 端点引用将请求发送到 WS-Resource。

通过 WS-Resource 构成，有可能将 Web Service 与它的状态完全进行分离，从而产生了一个无状态的 Web Service。这样无状态的服务遵照有状态的资源进行处理、提供访问，并基于服务所发送和接收的消息操纵一组逻辑上的有状态的资源。

为了说明 WS-Resource 的使用，现以第 5 章所介绍的订购单服务和清单 7.3 为例。清单 7.3 阐明了 PurchaseOrderPortType。在这里，我们假设订购单需要是有状态的实体，这些实体需要作为 WS-Resource 建模。对于每一个订购单请求的处理，WS-Resource 维护了与这些处理相关联的状态。

清单 7.3 第 5 章中的 PurchaseOrderPortType

```
<wsdl:message name="POMessage">
      <wsdl:part name="PurchaseOrder" type="tns:POType"/>
      <wsdl:part name="CustomerInfo" type="tns:CustomerInfoType"/>
</wsdl:message>

<wsdl:message name="InvMessage">
      <wsdl:part name="Invoice" type="tns:InvoiceType"/>
</wsdl:message>

<wsdl:portType name="PurchaseOrderPortType">
      <wsdl:operation name="SendPurchase">
            <wsdl:input message="tns:POMessage"/>
                <wsdl:output message="tns:InvMessage"/>
      </wsdl:operation>
</wsdl:protType>
```

按照本节前面所给出的 WS-Resource 的定义，WS-Resource 可以定义为订购单 Web Service 和有状态的资源的组合，并能够按照隐含资源模式访问有状态的资源。清单 7.3 中的 SendPurchase 操作然后变成一个工厂操作，该工厂操作创建一个新的订购单 WS-Resource。正如我们在 7.2.4 节中将要介绍的，WS-RF 定义一个工厂模式，该工厂模式可创建并返回对于一个或多个新的 WS-Resource 的端点引用。该响应不再是我们在清单 7.3 中所看到的发票，而是一个指向订购单 WS-Resource 的端点引用。客户端发出的 SendPurchase 操作将创建订购单 WS-Resource。从而，应用可使用隐含资源模式实现订购单服务。这需要对清单 7.3 中的输出(响应)消息做如下修改：

```
<wsdl:message name="InvMessage">
     <wsdl:part name="POEndPointReference"
      element="tns:POReference"/>
</wsdl:message>
```

在 WSDL 接口定义的类型元素部分可如下定义 POReference 元素：

```
<xsd: element name="POReference"
      type="wsa: EndPointReferenceType"/>
</xsd:message>
```

负责 WS-Resource 的 < PortType > 元素提供了一些操作。通过这些操作，客户端可以查询客户所提交的订购单的内容、查询所提交的订购单的状态、取消订购单等。在下面的章节中将要分析 < PortType > 元素。

本节中最后所要讨论的内容是如何处理客户端所发出的 SendPurchase 操作的响应。如图 7.5 所示，SendPurchase 操作的响应消息内容包括一个对于订购单 WS-Resource 的限定的 WS-Resource 端点引用。该图标识了对于订购单 WS-Resource 的端点引用的三个重要的构成。它们分别是：

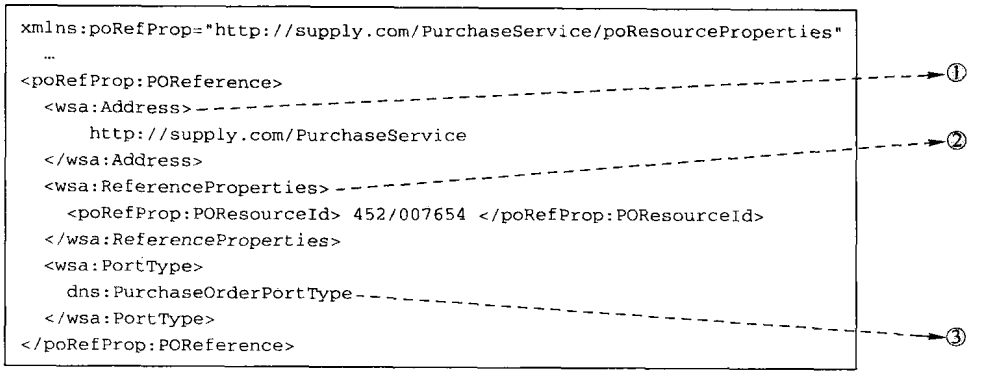


图 7.5　在 SendPurchase 操作后创建订购单，对于订购单的端点引用的构成

（1）WS-Addressing 中的 < Address > 元素表示了一个 Web Service 的 URL。该 Web Service 能够在订购单 WS-Resource 上执行操作。

（2）引用特性唯一地表示了 < Address > 元素所指定的具体资源。更具体地说，在完成 SendPurchase 操作后将创建订购单有状态资源，引用特性元素将包含订购单有状态资源的标识符。

（3）< Address > 元素表示了 Web Service 的接口。PurchaseOrderPortType 描述了 Web Service。我们将在后面的清单 7.5 中分析 PurchaseOrderPortType。

7.2.3　资源属性

可以使用消息交换来访问有状态的实体的状态，因此对于有状态的资源而言，定义消息交换是一个非常重要的事情。更具体地说，我们需要能够确定所支持的状态的类型以及具体的消息交换的类型、并需要能读取更新请求以及查询状态组件[Czajkowski 2004]。为此，WS-RF 规范使用标准的 XML Schema 全局元素声明来定义资源特性元素，资源特性元素可以作为 Web Service 接口的一部分进行声明。

在 Web Service 环境中，WS-Resource 通过一组组件来表示状态，这组组件称作资源特性元素。资源特性元素表示了最基本的原子状态元素，可以读取或更新这些原子状态元素。一组资源特性元素凑在一起就组合成了资源特性文档。资源特性文档是一个 XML 文档。客户端应用使用 XPath 或其他查询语言表达式可以查询资源特性文档。

WS-RF WS-ResourceProperties 文档本质上表示 WS-Resource 的资源特性的投影，例如投影某一处理器的使用，本质上是在 WS-Resource 中暴露有状态的资源组件。对于通过 Web Service 接口访问资源特性，WS-Resource 特性文档进行了定义[Graham 2004]。WS-Resource 特性文档聚集资源特性元素。通过使用 WSDL 1.1 < portType > 上的 XML 属性，WS-Resource 特性文档与一个 Web Service 接口相关联。因此，可以获取已有的资源特性文档以及资源特性文档的类型，以及和特定的 < portType > 的关联。< portType > 定义了 WS-Resource 的各种类型。

对于服务实现资源特性文档的方法，WS-Resource 特性文档并没有进行规定[Graham 2004c]。在一些服务实现中，可以将资源特性文档作为真正的 XML 实例文档进行实现，可以存储在内存中、文件系统中、数据库中或者一些 XML 库中。而在其他的一些服务实现中，可以动态构成资源特性元素和它们的值，这些值来自于编程语言对象中的数据（诸如 J2EE EJB Entity Bean），或者

通过在和物理资源的私有通信信道上执行一个命令的方式来获取这些值。

WS-Resource 特性规范定义了一个名为 < ResourceProperties > 的 XML 属性。基于 < ResourceProperties > 和 WSDL 1.1 < PortType > 元素，开发者可以规定 WS-Resource 的状态的 XML 定义。该属性也表明了：所开发的应用符合 WS-ResourceProperties 标准。使用这个方式，我们可以将订购单资源属性文档的格式指定为类型 POResourceProperties 的 XML 元素，如清单 7.4 中的代码所示。样例 7.4 中的代码源自[Graham 2004]，并针对第 5 章中的订购单样例进行了改编。

清单 7.4 WS-ResourceProperties 文档样例

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://supply.com/PurchaseService/
                                poResourceProperties"
  xmlns:poRefProp="http://supply.com/PurchaseService/
                                poResourceProperties"
  xmlns:po="http://supply.com/PurchaseService/PurchaseOrder"
  ... ... >
  ... ...
<wsdl:types>
    <xsd:element name="dateReceived" type="xsd:dateTime" />
    <!-- Resource properties document declaration -->
      <xsd:element name="execution-status">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string">
               <xsd:element value="received" />
               <xsd:element value="paid" />
               <xsd:element value="pending" />
               <xsd:element value="invoiced" />
               <xsd:element value="terminated" />
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:element>
       ...
      <xsd:element name="statusDate" type="xsd:dateTime" />
      <xsd:element name="terminationDate" type="xsd:dateTime" />
      <xsd:element name="handlingPerson" type="xsd:string" />
      <xsd:element name="handlingDepartment" type="xsd:string" />

      <xsd:element name="poResourceProperties">
         <xsd:complexType>
            <xsd:sequence>
               ... ...
               ... ...
            <!-- Resource property element declarations specific
                                          to Purchase Order -->
               <xsd:element ref="po:po" minOccurs="1"
                                              maxOccurs="1" />
               <xsd:element ref="poRefProp:dateReceived"
                                    minOccurs="1" maxOccurs="1" />

               <xsd:element ref="poRefProp:execution-status"
                                    minOccurs="1" maxOccurs="1" />
               <xsd:element ref="poRefProp:statusDate"
                                    minOccurs="1" maxOccurs="1" />
               <xsd:element ref="poRefProp:handlingPerson"
                                    minOccurs="1" maxOccurs="1" />

               <xsd:element ref="poRefProp:handling-Department"
                                    minOccurs="1" maxOccurs="1" />

            </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
```

```
  </xsd:schema>
</wsdl:types>
```

在清单 7.4 中，POResourceProperties 的真正的定义显现在清单的底端。那儿也显现了顾客提交的起初的订购单、顾客提交订购单的日期、订购单的当前处理状态、处理订单的人的名字、该处理人员所在的部门等。

通过消息交换(操作)，请求者可以检索资源特性的值、更新资源特性的值，以及查询资源特性，WS-ResourceProperties 定义了这些消息交换集[Graham 2004c]。

清单 7.5 显示了新的订购单 WSDL 定义，这些定义主要针对了 WS-ResourceProperty 的概念和操作。正如预期的，清单中也包括了一个新的 PurchaseOrderPortType 元素。

清单 7.5　针对订购单的资源特性操作

```
<wsrp:GetMultipleResourceProperties
  xmlns:tns="http://supply.com/PurchaseService/wsdl" … >
      <wsrp:ResourceProperty> tns:executionStatus
       </wsrp:ResourceProperty>
      <wsrp:ResourceProperty> tns:handlingPerson
       </wsrp:ResourceProperty>
</wsrp:GetMultipleResourceProperties>
<wsdl:definitions name="PurchaseOrder"
 xmlns:tns="http://supply.com/PurchaseService/wsdl"
 ... >

<!- Type definitions -->
<wsdl:types>
   <xsd:schema
        targetNamespace="http://supply.com/PurchaseService/
        poResource.wsdl">
          ... ...
        <xsd:element name="ErrorMessage" type="xsd:string" />
        <xsd:element name="poResourceID"
         type="xsd:positiveInteger" />
   </xsd:schema?
</wsdl:types>
<!-- Message definitions -->
<wsdl:message name="ErrorMessage">
   <wsdl:part name="ErrorMessage" element="poRefProp:ErrorMessage" />
</wsdl:message>
<wsdl:message name="GetInvoiceRequest">
  ... ...
</wsdl:message>
<wsdl:message name="GetInvoiceResponse">
   <wsdl:part name="GetInvoiceRequest" element="inv:invoice" />
</wsdl:message>

<!-- Association of resource properties document to a portType -->
<wsdl:portType name="PurchaseOrderPortType"
        wsrp:ResourceProperties="poRefProp:poResourceProperties">

   <!-Operations supported by the PO PortType -->
   <wsdl:operation name="getInvoice">
        <wsdl:input name="GetInvoiceRequest"
                    message="poRefProp:GetInvoiceRequest" />
        <wsdl:output name="GetInvoiceResponse"
                     message="poRefProp:GetInvoiceResponse" />
        <wsdl:fault name="FaultyInvoice"
                    message="poRefProp:ErrorMessage" />
    </wsdl:operation>
    <wsdl:operation name="dispatch-order"> ... </wsdl:operation>
```

```
    <wsdl:operation name="cancel-order"> ... </wsdl:operation>
    <!-- WS-RF operations supported by this portType -->
    <wsdl:operation name="GetResourceProperty"> ...
                                                </wsdl:operation>
    <wsdl:operation name="GetMultipleResourceProperties"> ...
                                                </wsdl:operation>
    <wsdl:operation name="QueryResourceProperties"> ...
                                                </wsdl:operation>
    <wsdl:operation name="SetResourceProperties"> ...
                                                </wsdl:operation>
     :
 </wsdl:portType>
</wsdl:definitions>
```

下面的代码片段表示了一个的请求消息，这个请求消息在实现了 PurchaseOrderPortType 元素的 WS-Resource 中检索两个资源特性元素(订购单的执行状态和处理订购单的人员)。

```
<wsrp:GetMultipleResourceProperties
  xmlns:tns="http://supply.com/PurchaseService/wsdl" … >
      <wsrp:ResourceProperty> tns:executionStatus
       </wsrp:ResourceProperty>
      <wsrp:ResourceProperty> tns:handlingPerson
       </wsrp:ResourceProperty>
</wsrp:GetMultipleResourceProperties>
```

对于前面的简单请求，下面是一个响应样例：

```
<wsrp:GetMultipleResourcePropertiesResponse
   xmlns:ns1= xmlns:tns="http://supply.com/PurchaseService/wsdl" … >
   <ns1:executionStatus> paid </ns1:executionStatus>
   <ns1: handlingPerson> Charles Simpson </ns1: handlingPerson>
</wsrp:GetMultipleResourcePropertiesResponse>
```

有关 WS-ResourceProperties 构成和定义的更详细的定义，可参见文献[Graham 2004c]。更多的有关如何使用 WS-ResourceProperties 标准的清单和例子可参见文献[Graham 2004a]。

7.2.4 资源生命周期

从创建 WS-Resource(有状态的实体)到销毁 WS-Resource，这段时间称为 WS-Resource 的生命周期，WS-Resource 的生命周期既可以是静态的，也可以是动态的[Shaikh 2004]。那些在系统上以永久或半永久方式保持的资源，无论它们是在系统上创建的还是部署到系统上，这些资源都是具有静态的生命周期。反之，具有动态生命周期的资源的创建和销毁都很频繁。WS-RF 涉及了实体生命周期的三个方面：创建、指派标识和销毁。

WS-RF 通过 out-of-bound 方式或工厂模式来处理服务创建。工厂模式(factory pattern)这个术语表示 Web Service 支持创建，并返回对一个或多个新的 WS-Resource 的端点引用。WS-Resource 工厂是一个 Web Service，它能够创建 WS-Resource 并给新创建的 WS-Resource 赋予一个标识。若要创建新的 WS-Resource，则需要创建一个新的限定的 WS-Resource 端点引用，该端点引用包含引用新的 WS-Resource 的 WS-Resource 上下文。可以将端点引用返回给服务创建者或保存起来，例如保存到注册库中以供以后检索。

WS-RF(见表7.1)中的 WS-ResourceLifetime[Srinivasan 2004] 规定了销毁 WS-Resource 的标准方式。该规范也定义了监测 WS-Resource 生命周期的方式。通常情况下，服务请求者很关心 WS-Resource 的一些未来时段的情况。在许多场合中，WS-Resource 的客户端可能会立即销毁资源。在这样的情况下，WS-Resource 可能支持消息交换模式。通过消息交换，服务请求者可以请求销毁资源。这即是 WS-ResourceLifetime 规范中的“立即销毁(immediate destruction)”方式。使用在 WS-ResourceLifetime 规范中定义的消息交换可以实现 WS-Resource 的立即销毁。为了立即销毁一

个 WS-Resource，服务请求者必须使用对销毁消息的限定的 WS-Resource 端点引用。

对于管理 WS-Resource 的销毁，WS-ResourceLifetime 规范也提供了一个基于时间的方式。这类销毁方式称为“预订销毁(scheduled structure)”。WS-ResourceLifetime 定义了一个标准的消息交换。通过该标准的消息交换，服务请求者能够确定和重设 WS-Resource 的终止时间。并且通过消息交换，服务请求者可以确定终止时间是否已到[Srinivasan 2004]。使用限定的 WS-Resource 端点引用，服务请求者可以确定或者重设 WS-Resource 的预订终止时间。

7.2.5　服务组

术语服务组指的是一种创建异构传参的 Web Service 集合的方式[Maguire 2005]。服务组可以构成大量的服务集合，包括构建服务注册库以及相关的 WS-Resource。可以使用称为条目(entry)的组件来表示服务组(ServiceGroup)的成员。服务组条目就是一个 WS-Resource。

WS-ServiceGroup 规范[Maguire 2005]定义了聚合 Web Service 和 WS-Resource 的方法，并可针对具体的领域目标对 Web Service 和 WS-Resource 进行分组。为了请求者可以有效地查询服务组的内容，必须使用一定的方式对组中的成员关系进行约束。通过分类方式，内涵(intension)可表示成员间的约束关系。此外，每一个内涵的成员必须共享一组通用的能够查询的信息。和一个服务组条目相关的 Web Service 可以包含多个 Web Service 标准，诸如 WS-ResourceProperties [Graham 2004c]、WS-ResourceLifetime [Srinivasan 2004]和 WS_BaseNotification [Graham 2004d](我们将要在下面的章节中分析 WS_BaseNotification)。

在 WS-ServiceGroup 规范(图 7.6)中，可以使用资源特性模型表示 ServiceGroup 成员关系规则、成员关系约束和分类，满足约束的成员集合可定义为组。服务组维护有关 Web Service 集合的信息。在集合中所表示的每一个 Web Service 都可以是 WS-Resource 的一个组件。这些 Web Service 可以由于某种具体的原因而成为服务组的成员，诸如作为联合服务的一部分。这些 Web Service 之间也可以没有具体的关系，诸如包含在索引或注册库中的 Web Service，Web Service 发现操作需要用到这些索引和注册库。

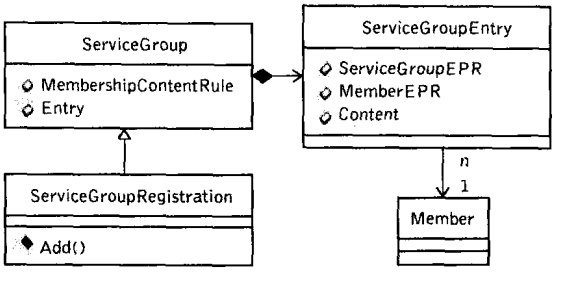


图 7.6　服务组接口

有关服务组规范以及如何使用它们的实例的更详细的描述，可参见文献[Maguire 2005]。

7.3　Web Service 通知

正如我们在第 2 章中所讨论的：在事件驱动的体系结构以及面向消息的中间件中，最核心的功能就是发布与订阅处理。对于事件驱动的 SOA 模型(参见第 8 章)，事件驱动的处理和通知尤为重要。在事件驱动的 SOA 模型中，Web Service 间通过异步消息交换进行交互。

对于 SOA 的实现，事件驱动的处理和通知引入了一个新的模式，这个模式叫做通知模式。在通知模式(有时也称作事件模式)中，提供信息的服务向一个或多个接收者发送单向消息。对于相同的信息，可以注册希望获取这些信息的多个服务。此外，提供信息(分布式)的服务可以向每一个已注册的服务发送任意数量的消息。在这个模式中，消息通常携载所发生的事件的有关信息，而不是请求执行某种具体的操作。消息接收者在接收通知之前必须先注册。服务注册既可以由希望获取消息的服务本身发起，也可以由第三方发起。开发者可以预先配置注册或动态注册。

OASIS Web Service Notification 是一组相关的规范集，这组规范集定义了标准的 Web Service 通知方式。该方式使用基于主题的发布/订阅模式。对于希望参与通知和标准的消息交换的提供者，WS-Notification 规范定义了服务提供者所实现的标准的消息交换，消息交换允许服务提供者之外的实体发布消息。该方式定义了一些机制和接口。通过这些机制和接口，客户端可以订阅感兴趣的主题，诸如一个 WS-Resource 的资源特性值的变化。从 WS-Notification 的角度，WS-RF 提供了一些构建模块，可用于表示和结构化通知。从 WS-RF 角度，WS-Notification 规范集扩展了 WS-Resource 的应用程序，允许需求者要求异步地通知资源特性值的变化。除了 WS-RF 之外，WS-Notification 也可与 WS-Policy、WS-ReliableMessaging 协同工作。

WS-Notification 标准集支持代理模式发布/订阅通知以及对等模式的发布/订阅通知。WS-Notification 标准集提议了三个规范性的规范：WS-BaseNotification、WS-Topics 和 WS-BrokeredNotification，下面将依次对它们进行分析。

7.3.1 P2P 通知

WS-BaseNotification 规范定义了通知的消费者和生产者的标准接口。该规范包括了标准的消息交换以及一些常规的运营需求。希望充当通知的消费者和生产者的服务提供者实现了标准的消息交换。按照该规范，通知的生产者需要暴露一个 subscribe 操作。通知的消费者可以使用该操作来请求一个订阅。通知的消费者则需要暴露一个 notify 操作，通知生产者能够使用该操作发送通知。假如通知的消费者直接与提供通知的生产者进行订阅，则这种配置称为对等模式（P2P），有时也称为直接通知模式或点对点的通知模式。通知模式也有其他的一些变体，例如通知的消费者可向通知中介服务进行订阅。WS-Brokered Notification 规范描述了有关通知中介服务的内容，可参见 7.3.3 节。

WS-BaseNotification 是一个基础规范。其他的 WS-Notification 规范文档需要依赖这个基础规范。当订阅者在通知生产者那儿注册它所感兴趣的通知时，需要涉及三个基本角色：通知生产者，通知消费者和订阅者。WS-BaseNotification 描述了这三个基本角色[Graham 2004d]。它包括服务提供者所实现的标准的消息交换。图 7.7 描述了 WS-BaseNotification 中的三个基本角色和它们的消息交互模式。下面将进行进一步的分析。



图 7.7 基本的消息交换模式

通知的生产者是一个 Web Service，负责管理实际的通知流程。通知的生产者负责维护接收者列表，并负责安排将通知消息发送给那些接收者。这可能涉及一个匹配步骤，将通知消息与各个接收者（通知的消费者）的兴趣进行比较。通知的生产者履行两个功能：生成通知并处理通知的订阅

如图 7.7 所示，通知的生产者接收进来的 subscribe 请求，通知的生产者接收进来的 subscribe 请求，每一个 subscriber 请求标识了一个或多个感兴趣的主题以及对一个通知的消费者的引用。假如通知生产者愿意接收该请求，它将创建一个新的订阅，并将该订阅添加到活动订阅列表中。通知的生产者然后开始将通知消息发送给相关的通知消费者。

通知的消费者也是一个 Web Service，它和通知的生产者相对应。通知的消费者从通知的生产者那里接收通知消息。通知可以涉及任何事情：修改资源特性的值，例如订单状态的变化，基于时间的事件，如账单到期，通知生产者的状态方面的内部变化，环境中的一些其他状况。

发布者可以由一个通知生产者来充当，它是一个能够基于所处的环境状况创建通知消息的实体。环境能够监测和转换通知消息。发布者选择合适类型的通知消息，并构成一个通知消息实例，该通知消息包含了与特定环境相关的信息。假如消息生产者并不充当一个发布者，则它被称作通知代理，并且实际上并不创建通知消息，而是代表一个或多个发布者管理通知流程，参见 7.3.3 节。

环境(situation)就是一个对事件做出反应的对象。在 WS-Notification 中，环境与状态的变化(例如订购单的状态从“待定”状态变为“已开发票”状态)、基于时间的事件(例如定时器的满期)或者与系统资源相关的事件(例如服务器故障)相关。例如，本章前面分析的订购单 WS-Resource 是一个能够检测环境并生成相关的通知消息的对象，其中环境与一个订购单相关。与一个特定环境相关的信息可以被传送到预期的服务中，这点对于通知模式是非常重要的。

在 WS-Notification 中，术语“通知”指的是将特定环境的信息传送到其他服务的单向消息。对于每一个不同的环境，通知消息的发送者可以选择格式化消息的方式以及所使用的表示方式。通知消息包括与消息关联的主题、说明主题的方式，并可以选择包括一个对生产者的引用。因为通知消息能够包含任何类型的具体应用的通知消息，所以通知消息充当了一个通用的通知 - 发送方式。

单个的通知(notify)消息可以包含多个通知消息，将没有处理过的消息与具体的通知信息组合在一起是非常重要的[Vinovski 2004]，并且 notify 消息支持批处理通知发送形式。通过 notify 消息，除了具体应用的通知消息的内容，通知的生产者可以提供定义 WS-Notification 的额外信息(诸如主题)。通知的生产者维护一张订阅列表。对于通知和相关的通知消息模式，可使用主题(topic)进行分类。匹配处理中可使用主题来确定哪一个(假如有的话)订阅通知的消费者将要接收一个通知消息。当通知生产者有一个通知需要分发时，会将通知与已注册的每一个订阅的兴趣进行比较。假如有一个合适的订阅匹配，则将通知发送给与该订阅相关的通知消费者。

订阅(subscription)是一个表示通知消费者和通知生产者之间的关系的实体。通知生产这能够提供一些甚至所有的通知。订阅可以包含过滤表达式、策略和上下文信息。每一个通知生产者都拥有一个活动订阅列表。每当生成一个通知时，就会将该通知与订阅列表上所注册的兴趣进行比较。假如有一个合适的订阅匹配，则将通知发送给与该订阅相关的通知消费者。

按照 7.2 节中定义的隐含资源模式，WS-Notification 将订阅表示为一个 WS-Resource。订阅是 subscribe 请求所创建的有状态的资源。使用 WS-ResourceLifetime(参见 7.2.4 节)可管理订阅的生命周期。本质上，订阅表示了通知消费者、通知生产者、主题以及各种其他可选的过滤表达式、策略和上下文信息之间的关系。

最后，订阅者(subscriber)是一个充当服务请求者的实体(通常是一个 Web Service)。订阅者将 subscribe 请求消息发送给一个通知生产者，参见图 7.7。这将创建一个订阅资源。注意，订阅者与实际接受通知消息的通知消费者可以是不同的实体。例如，发送订购单请求(通过调用 PurchaseOrderPortType 上的 SendPurchase 操作)的 Web Service 可以订阅有关订购单状态变化的信息，或者将一个单独的库存管理系统作为通知消费者。

为了创建订阅，订阅者必须向通知生产者发送具体的信息。该信息包括对发送通知消息的生产者的消费者端点引用、主题表达式以及表示方式，其中主题表达式表示了消费者所感兴趣的主题[Vinovski 2004]。订阅请求的响应是一个端点引用。端点引用包括新创建的订阅的标识符和订阅管理服务的地址，其中订阅管理服务能够对订阅进行管理。

图 7.8 表示了我们上面所描述的 WS-BaseNotification 实体。该图显示了一个订阅者，该订阅者代表通知的消费者(如图中虚线所示)构成一个对通知生产者的 subscribe 请求。作为该请求的

结果，通知生产者将一个订阅添加到它的订阅列表中，并向订阅者发送一个响应。每一个订阅条目记录了通知消费者的标识(ID)以及订阅的其他特性，诸如订阅的终止时间以及与订阅相关的任何过滤表达式。一旦发布者检测到一个环境(例如发票已经过了有效期)，便会通知生产者向通知消费者发送一个通知。

1. WS-BaseNotification 接口

WS-BaseNotification 的主要贡献主要体现在两方面：定义了(接受通知订阅和发送通知消息的应用所支持的)通知生产者接口和(接受通知消息的应用所支持的)通知消费者接口。下面将要对它们进行一个简要的总结。

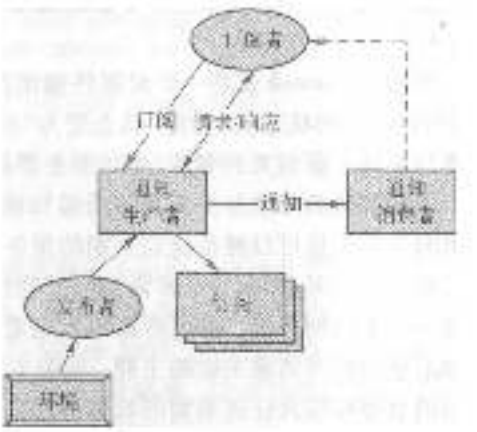


图 7.8 WS-BaseNotification 实体

通知消费者接口

在 WS-Notification 中，通知生产者以两种方式中的一种向通知接受者发送通知消息。通知生产者可以简单地向通知消费者发送未经处理的通知消息，例如具体应用的内容。或者，通知生产者使用 notify 操作发送通知消息数据。这个操作意味着：通知消费者可以接收各种通知消息，而无须在 WSDL 中对每一类消息直接提供支持。

清单 7.6 是一个使用 SOAP 的 notify 消息的样例。注意，在消息中使用了 wsa: ReferenceProperties 元素。这些是限定的 WS-Resource 端点引用的例子，随后是 7.2 节所概述的隐含资源模式。

清单 7.6 通知消息和 SOAP 的协同使用

```
<Soap:Envelope
   xmlns:Soap="http://www.w3.org/2003/05/soap-envelope"
   xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
   xmlns:wsnt="http://www.ibm.com/xmlns/stdwip/Web-services/
     WS-BaseNotification"
   xmlns:ncex="http://www.consumer.org/RefProp"
   xmlns:npex="http://www.producer.org/RefProp">
  <Soap:Header>
    <wsa:Action>
        http://www.ibm.com/xmlns/stdwip/Web-services/
        WS-BaseNotification/Notify
    </wsa:Action>
    <wsa:To e12:mustUnderstand="1">
        http://www.consumer.org/ConsumerEndpoint
    </wsa:To>
    <ncex:NCResourceId>
      Uuid: ... ...
    </ncex:NCResourceId>
  </Soap:Header>
  <Soap:Body>
    <wsnt:Notify>
      <wsnt:NotificationMessage>
        <wsnt:Topic dialect="http://www.ibm.com/xmlns/stdwip/
         Webservices/WSTopics/
         TopicExpression/simple">
            npex:Some.Topic
        </wsnt:Topic>
        <wsnt:ProducerReference
            xmlns:npex="http://www.producer.org/RefProp">
          <wsa:Address>
            http://www.producer.org/ProducerEndpoint
```

```
          </wsa:Address>
          <wsa:ReferenceProperties>
             <npex:NPResourceId>
                 uuid: … … …
             </npex:NPResourceId>
          </wsa:ReferenceProperties>
        </wsnt:ProducerReference>
        <wsnt:Message>
            <npex:NotifyContent>exampleNotifyContent
                                </npex:NotifyContent>
        </wsnt:Message>
      <wsnt:NotificationMessage>
    </wsnt:Notify>
  </Soap:Body>
</Soap:Envelope>
```

通知生产者接口

通过发送 subscribe 消息(操作)，订阅者可以注册关于对一个或多个主题的兴趣，并接受相应的消息。为了进行这样的注册，订阅者将向通知生产者发送一个消息。作为 subscribe 请求消息处理的一部分，通知生产者必须创建一个用来表示订阅的订阅资源。作为响应，订阅者从通知生产者那儿接收了一个对“订阅 WS-Resource”的限定的 WS-Resource 端点引用。端点引用包括一个订阅管理器服务的地址和一个引用特性，引用特性包含了订阅资源的标识。订阅 WS-Resource 表示了订阅者和通知生产者之间的关系，并使用 WS-ResourceProperties 和 WSResourceLifetime 来管理这种关系。

通知生产者接口支持消息交换。通过消息交换，通知生产者可宣传它对一个或多个主题的支持；订阅者可以创建订阅；通知生产者也可控制通知消息的发送。

为了使得最近才参与订阅的通知消费者也能获取其他订阅者已经接收的通知消息，可以向通知生产者发送 GetCurrentMessage 消息。作为 GetCurrentMessage 消息的响应，通知生产者可以返回特定主题的最近发布的消息。

清单7.7 说明了通知生产者接口消息如何能附属于清单7.5 中定义的 PurchaseOrderPortType。

清单 7.7　在 PurchaseOrderPortType 中包含通知操作

```
<wsdl:definitions name="PurchaseOrder"
 xmlns:tns="http://supply.com/PurchaseService/wsdl"
 ... >

 <!-- Type definitions -->
      ... ...
 <!-- Message definitions -->
       ... ...
 <!-- portType defnitions -->
 <wsdl:portType name="PurchaseOrderPortType"
                wsrp:ResourceProperties="
                poRefProp:poResourceProperties">

    <!--Operations supported by the PO PortType -->
    <wsdl:operation name="getInvoice"> ... </wsdl:operation>
    <wsdl:operation name="dispatch-order"> ... </wsdl:operation>
    <wsdl:operation name="cancel-order"> ... </wsdl:operation>

    <!-- WS-RF operations supported by this portType -->
    <wsdl:operation name="GetResourceProperty"> ...
                                          </wsdl:operation>
    <wsdl:operation name="GetMultipleResourceProperties"> ...
                                          </wsdl:operation>
```

```
        <wsdl:operation name="QueryResourceProperties"> ...
                                                    </wsdl:operation>
        <wsdl:operation name="SetResourceProperties"> ...
                                                    </wsdl:operation>

        <!-- WS-Notication operations supported by this portType -->
        <wsdl:operation name="Subscribe"> ... </wsdl:operation>
        <wsdl:operation name="GetCurrentMessage"> ...
                                                    </wsdl:operation>
            :
    </wsdl:portType>
</wsdl:definitions>
```

订阅管理器接口

当通知生产者接受订阅请求时，通知生产者将会在请求响应中返回一个端点引用，作为对订阅的引用。假如端点引用携载了一个 Web Service 的地址，则该 Web Service 实际上是一个订阅管理器。通过订阅管理器，服务请求者可查询、删除或续订[Niblett 2005]。通过支持返回的许多资源特性，例如订阅的过滤表达式、消费者端点引用和计划的终止时间等，订阅管理器提供了查询能力。订阅管理器是有状态的 Web Service 的另一个例子[Niblett 2005]。

订阅管理器接口定义了操纵订阅资源的消息交换[Graham 2004d]。订阅管理器支持所需的消息交换，这些消息交换与 WS-ResourceProperties 规范相关。除了支持 WS-ResourceProperties 操作，订阅管理器也必须支持 WS-ResourceLifetime 针对两种形式的资源生命周期(立即销毁和预定销毁)而定义的消息交换。这些消息交换定义了销毁(直接销毁或预定销毁)订阅资源的方法。

订阅管理器定义了 PauseSubscription 和 ResumeSubscription 操作。可基于特定的订阅向通知消费者发送通知消息，而 PauseSubscription 和 ResumeSubscription 操作可分别停止和重新启动这一处理。

2. 订阅过滤

过滤表达式表示了消费者感兴趣的通知，从而可以仅将订阅者感兴趣的通知发送给订阅者。对每一个订阅都会进行过滤，这意味着：一个特定的通知生产者可以有几个活动订阅，每一个订阅都可以具有一个不同的过滤表达式。此外，一个通知消费者也可以有多个订阅，每一个订阅也都可以具有一个不同的过滤表达式。

WS-BaseNotification 定义了三类基本的过滤表达式。然而，开发者可以不受标准的限制，自由地扩展过滤表达式。WS-BaseNotification 定义的三类过滤表达式是[Graham 2004b]：

- 消息过滤器：消息过滤器是基于通知消息的内容的布尔表达式。消息过滤器可以滤除表达式的值不为“真”的那些消息。例如，库存服务可能需要自动补充库存量低于一定阈值的那些产品。
- 主题过滤器：正如我们在下一节中所讨论的，主题是分类通知的一种常规方式。主题过滤器可以滤除与所规定的主题或主题列表不相符的通知。
- 通知生产者状态过滤器：这些过滤器的表达式与通知生产者本身的一些状态相关，消息交换中并没有携带这些信息，然而订阅者需要了解这些信息。

7.3.2 通知主题

使用通知的应用通常会声明它们所感兴趣的通知消息，这些消息需要满足一定的标准，例如某些特定的内容。可以使用兴趣表示来确定通知消息在网络上的接受者。

WS-Notification 可帮助消费者仅接受它们感兴趣的那些具体的通知消息。WS-Topics 规范定义了面向主题的通知系统所需的特性。尤其，WS-Notification 定义了将订阅进行组织和分类的方

式——“主题”。WS-Notification 定义了在通知机制中如何使用主题。

一个主题就是一类通知消息(对于术语主题的定义参见 7.3.1 节)。当订阅者创建一个订阅时，订阅者将订阅与一个或多个主题关联，从而表示了订阅者所感兴趣的通知。订阅者通过主题过滤器可以订阅主题，而不是在消息体中规定过滤器。由于并没有将主题过滤器与通知消息捆绑在一起，因此这种方式更具灵活性。在消息中并不是一定要出现主题名，并且多个消息类型可以关联到同一个特定的主题。

当订阅者创建一个订阅时，将订阅与一个或多个主题进行关联。除了检测环境和创建通知消息，通知的生产者负责将通知与订阅列表进行匹配，并将通知消息发送给每一个合适的消费者。为了完成这一任务，通知的生产者在匹配处理中使用了主题列表[Graham 2004a]。当通知的生产者生成了一个通知时，它会将通知与在主题列表中注册的订阅主题进行匹配。当生产者确定一个匹配后，即将通知发送给订阅该主题的消费者[Vambenepe 2004]。通知生产者可以声明环境的类型。这样，请求者将可以了解可订阅何种主题和信息。

可以使用相关的主题集来组织和分类一组通知消息。订阅者可以根据主题来了解消费者对哪些通知感兴趣。作为通知消息的发布的一部分，发布应用程序(发布者)可以与一个或多个主题进行关联。

主题可以按照层次结构组织成一棵主题树，其中每一个主题都可以有零个或多个子主题，并且每一个子主题又可进一步包含它的子主题[Vambenepe 2004]。每一棵主题树都包含一个根主题。通过层次主题结构，订阅者可以订阅多个主题。例如，一个订阅者可以订阅整个主题树或者主题树中的主题子集。假如订阅者对一个比较大的主题树子集感兴趣，这种方式可减少订阅者所需发出的订阅请求数。这也意味着：订阅者能够接收与后代主题相关的通知消息，而无须具体知道它们的存在。

主题可以安排到主题空间中。主题空间使用 XML 命名空间，从而可以避免主题定义的冲突。主题空间是主题树的集合(主题森林)。主题空间包含附加的元数据，元数据描述了如何将元数据作为 XML 文档进行建模。在主题空间中，所有的根主题都必须有一个唯一的名字。主题空间将主题树与命名空间进行关联，每一棵主题树都可以通过主题树根名进行唯一标识。使用一个基于路径的主题表达方式，子主题可以视为祖先根主题的亲属。

主题空间并不与一个特定的通知生产者捆绑在一起。主题空间包含主题定义的抽象集，许多不同的通知生产者都能使用主题定义的抽象集。对于一个特定的通知生产者，可以支持多个不同的主题空间中的主题。通知生产者可以支持整棵主题树，或者那棵主题树中的主题子集。通知生产者支持的主题列表可以随着时间而不断变化，例如增加以前没有支持的主题空间中的一些主题，或者将主题扩展为一个(新的或已经被支持的)主题空间。

通过 WS-Notification 基于主题的方式，可以在通知生产者、消费者或者这两者中进行消息过滤，这点对于可伸缩性非常重要。基于消费者的订阅中指定的主题，以及和那个订阅相关的任何选择符表达式和前提表达式，通知生产者可以进行通知过滤。通知的消费者能够应用一些标准进一步过滤源自生产者的消息。

图 7.9 描述了一个主题空间的例子，该主题空间包含两个层次型主题树。该主题空间的名字为“http://supply.com/topicsSpace/order-mgt-example/”。这个主题空间使用 WS-Notification 指定了两个根主题，其中一个描述了支付方式(Payment method)，而另一个则表示了公司订单(Order)。所有与支付相关的主题的根主题是支付方式。对支付进行建模是支付应用的一部分。支付方式可以分成几个子类，包括信用卡、支票或直接转账。在图 7.9 中，这几个子类是支付方式根主题的后代。

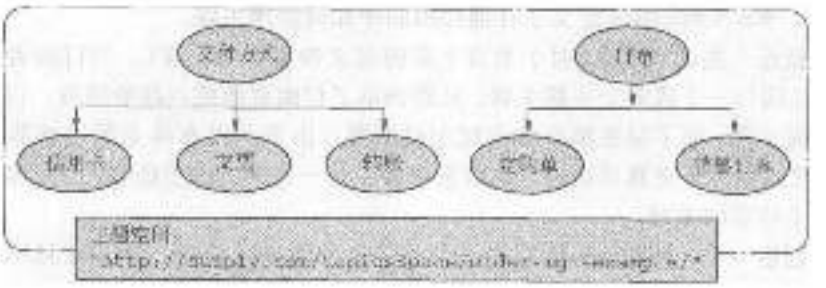


图 7.9 主题空间样例

清单 7.8 显示了如何使用 WS-Topics XML 主题模型指定一个主题空间。该清单表明了：为进行文档编制，可以对每一个主题空间赋予一个名字。每一个主题可以包含一个 messageType 属性，该属性定义了可与主题协作的通知消息的类型。这个属性的目的就是容纳与主题相关联的通知消息的内容[Vambenepe 2004]。例如，就转账支付而言，该清单表示了：与转账主题相关的所有通知消息都遵循 uss：MoneyTransferNotification。messageTypes 属性包含的一些信息可帮助设计合适的选择表达式，用来过滤有关 subscribe 请求的消息。每一个主题可包含一个称为 final 的可选属性，final 的默认值是"false"。假如 final 的值是"true"，则表示通知生产者不能动态地向主题中添加任何子主题。

清单 7.8 WS-Topic 主题空间定义的样例

```
<?xml version="1.0" encoding="UTF-8"?>
<wstop:topicSpace name="TopicSpaceOrderMgt-Example"
   targetNamespace="supply.com/topicsSpace/orderMgt-example"
   xmlns:tns="http://supply.com/topicsSpace/orderMgt-example"
   xmlns:wsrp="http://www.ibm.com/xmlns/stdwip/web-services/WS
                                                    ResourceProperties"
   xmlns:wstop="http://www.ibm.com/xmlns/stdwip/web-services/
                                                    WS-Topics">
   <wstop:topic name="PaymentMethod">
      <wstop:topic name="CreditCard"
            messageTypes="tns:CreditCardPaymentNotification"/>
      <wstop:topic name="Check"
            messageTypes="tns:CheckPaymentNotification"/>
      <wstop:topic name="MoneyTransfer"
            messageTypes="uss:MoneyTransferNotification"
            final="false">
         <wstop:documentation>
            All money transfers, including urgent money
            transfers appear under this topic.
         </wstop:documentation>
         <wstop:topic name="UrgentMoneyTransfer">
            <wsrp:QueryExpression
                Dialect="http://www.w3.org/TR/2001/
                WD-xpath20-20011114" >
                boolean(/order/OrderStatus="Urgent")
            </wsrp:QueryExpression>
         </wstop:topic>
   </wstop:topic>
   <wstop:topic name="Order">
      <wstop:topic name="PurchaseOrder"
            messageTypes="tns:m1" ... />
      <wstop:topic name="SalesOrder"
            messageTypes="tns:m2" ... />
```

```
        ... ... ...
    </wstop:topic>
</wstop:topicSpace>
```

对于 WS-Notification 中的主题的 XML 模型，清单 7.8 中的子主题 UrgentMoneyTransfer 示范了模型的消息模式功能部件的使用。为了描述消息模式或进一步指定约束，在这儿使用了 wsrp: QueryExpression 操作，其中约束作用在与特定主题(诸如转账)相关的消息上。清单 7.8 表明了: UrgentMoneyTransfer 主题的消息不仅遵循 tns: MoneyTransferNotification，而且也表示了一个潜在的订阅者，该订阅者处理与订单相关的转账支付，而订单的状态则为紧急订单。wsrp: QueryExpression 操作可以表示消息模式，该操作是一个 WS-ResourceProperties 操作，应用可以使用它发出一个有关 WS-Resource 的资源特性文档的查询表达式(诸如 XPath)。

WS-Topics 规范支持几类主题表达式，在 subscribe 和 notify 消息中，可以使用这些表达式来指定主题。表达式还可以用于表示通知生产者所支持的主题。WS-Topics 标准规定了几类主题表达式，既有简单的方式，如仅涉及特定主题空间中的根主题，也有使用类似 XPath 表达式这样的方式来表示主题。在 WS-Topic 中，可以使用如下三个主题表达式作为订阅表达式:

- 简单的主题表达式: 在仅处理简单的主题空间的 WS-Notification 系统中，被约束的实体资源根据简单的主题表达式语言定义这些简单的主题表达式。
- 具体的主题路径表达式: 在应用路径标记法的主题空间中，可以使用具体的主题路径表达式来精确地标识主题。具体的路径表达式应用了一个简单的路径语言。简单的路径语言有点类似于层次目录结构中的文件路径。
- 全主题路径表达式: 在具体的主题表达式的基础上，通过添加一些通配符和逻辑操作，可以构建全主题路径表达式。全主题路径表达式由 XPath 表达式组成。对于节点由主题空间中的主题构成、并且子主题作为所包含的 XML 元素的文档，可以使用 XPath 来衡量这些文档。

在文献[Vambenepe 2004]和[Graham 2004a]中，可以参阅有关 WS-Topic 表达式的其他样例。

7.3.3　代理通知

在封闭系统中，生产者应用程序和消费者应用程序彼此知道对方，在通知的生产者和消费者之间进行直接连接的这种耦合关系并不会影响系统。因此，对等通知(或直接)模式采用直接连接方式是有价值的。然而，绝大多数基于事件的系统(例如 8.5 节中的事件驱动的企业服务总线)希望能够解耦通知生产者和消费者。为了实现这一目标，创建了 WS-BrokeredNotification 规范[Chappell 2004]。这个规范引入了一个新的角色“通知代理”。通知代理位于发布者和出版者之间，用于解耦发布者/出版者之间的关系，从而可以提供更大的可伸缩性。

通知代理是一个中介 Web Service，它解耦了通知生产者和通知消费者之间的关系，从而提供了具有伸缩性的事件处理能力。通知代理如同生产者应用和消费者应用之间的中间人，生产者和消费者各自都知道代理，但它们相互之间并不知道。通知代理既承担了通知生产者的角色也承担了通知接受者的角色(正如在 WS-Base Notification 中所定义的)，并且 WS-Base Notification 规范也很大程度上定义了通知代理与其他的通知生产者和通知消费者之间的交互。

生产者通知和消费者通知的设计目标主要并不是用于具有大规模的通知需求的应用，代理可以改善系统的可伸缩性。代理技术可以有效地解决通知处理中的一些难题，诸如处理订阅、过滤、对多个消费者的有效发送和消息持久性[Vinovski 2004]。代理技术可以实现基础架构和具体应用之间的分离。

WS-BrokeredNotification 所定义的通知代理是基本的通知生产者的一个扩展。具体的扩展包

者[Graham 2004b]：

- 发布者无须实现与通知生产者相关联的消息交换。消息代理代表发布者承担了订阅管理器（管理订阅）和通知生产者的职责。
- 在有许多生产者和许多通知消费者的系统中，极大地减少了服务间的连接和用于。
- 充当发现服务的功能，从而使得许多潜在的发布者和订阅者能够更方便地相互发现。
- 提供匿名服务通知，因此发布者和通知消费者并不需要一定要知道彼此的标识。

输入端的通知代理实现了通知消费者接口，而输出端的通知代理则实现了通知生产者代理。对于通知生产者而言，代理充当了一个消费者。对于通知消费者而言，代理则充当了一个提供者。

相比于 WS-BrokeredNotification 规范所提供的功能，通知代理可以提供更多的增值功能。例如，WS-Notification 和相关标准的一些功能都仅提供了一些工具，诸如用于审计目的的日志通知消息，或者认证、消息持久性、消息完整性、消息加密、发送保证等，都可由代理来实现。使用代理可以带来一些额外的好处，即发布者无须了解有关订阅者的任何信息，从而完全解耦了订阅者和发布者之间的。集中化的订阅和主题管理使得企业具有更强的控制能力，并可以根据服务级别协议（SLA）更精确地度量性能。

图 7.10 描述了一个可能的代理消息交换序列。在这种情况下，在订阅者和通知代理之间的消息交换序列与图 7.7 中的消息交换序列一样，图 7.7 中显示了不使用代理的情况下的订阅者和通知生产者之间的消息交换序列[Graham 2004b]。在图 7.10 中，发布者无须直接和最终的通知消费者交互，而是与通知代理交互。通知代理支持发布者与通知代理之间的消息交换序列。通过 notify 消息，发布者可向通知代理发布一个通知消息。随后，对于与发布匹配的订阅，通知代理可以向这些订阅的任何消费者发送通知消息。通过发送 notify 消息，可以实现发布功能。对于发布者而言，通知代理可充当任何通知消费者。因此，通知代理可以插入在消息流之间（例如出于审计目的），并且发布者无须修改它发送通知消息的方式。

如图 7.10 所示的 notify 消息交换可以有两个作用[Graham 2004b]。第一个作用是向通知代理发送通知消息。第二个作用是向通知消费者发送通知消息。因此，在通知生产者（发布者）和通知消费者之间可以存在一个通知代理和通知代理的匿名中介链。

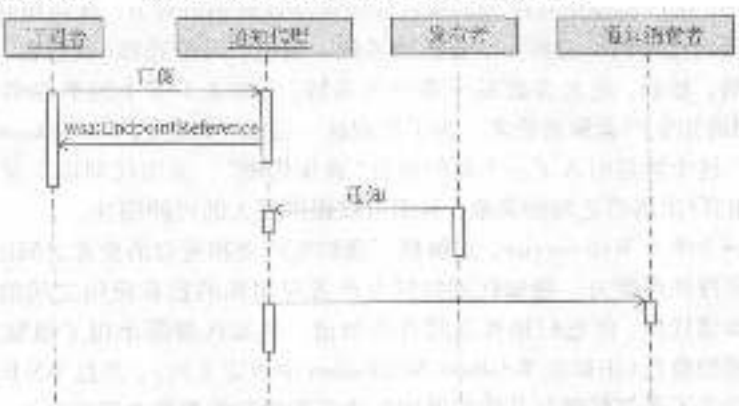


图 7.10 代理消息交换模式

最后提及一点，当前 WS-Notification 标准族仅描述了通知推送模式的轮廓。通知推送模式基于成熟的发布/订阅模型。在推送模型中，通知将被推送到消费者。在推送模型中，一旦消息被生成后，它们将被立即发送给消费者，这是推送模型的优点。WS-Notification 也支持通知的委托发送，其中中介和代理能够将通知推送给消费者。

7.4 Web Service 事件

为了处理 Web Service 事件驱动的流程，最近 BEA、微软、Sun 和 Tibco 等公司发布了一个称为 Web Service Event（WS-Eventing）的规范。该规范定义了一个基准操作集，使得 Web Service 能够提供简单的异步事件通知。WS-Eventing 是一个有关协议、消息格式和接口的集合，Web Service 通过接口可以订阅事件通知或接收订阅。例如，事件通知可以附属于订单装运，或者在请求处理一个特定的事务时包含事件通知，当处理完成时，发送者将接收一个消息，从而确保事件已经被处理。

WS-Eventing 规范定义了一个协议，一个 Web Service（称作订阅者）可以向另一个 Web Service（称作事件源）注册所感兴趣的事件（称作订阅），从而能够接收有关事件的消息（称作通知或事件消息）。事件源是一个 Web Service，它能发送通知以及接收创建订阅的请求。WS-Eventing 中的通知是一个单向消息，用于表明所出现的事件。接收通知的 Web Service 叫做"事件接收器"（event sink）。订阅者也是一个 Web Service，该 Web Service 发送一些创建、更新、撤销订阅的请求。通过与一个专用的 Web Service（即订阅管理器）进行交互，订阅者可以管理订阅。订阅管理器是由事件源进行指派的。订阅管理器代表事件源接受管理请求、获取订阅的状态、更新和/或删除订阅。若要更新订阅的有效期，订阅管理器可请求 renew 订阅。事件源支持过滤操作，从而可限制发送到事件接收器的通知。假如进行过滤，订阅请求包含一个过滤器，事件源仅发送与所请求的过滤相匹配的通知。当下列情况之一发生时，事件源将发送通知：1）订阅管理器接受一个退订请求；2）订阅到期且没有进行续期；3）事件源提前取消订阅。

最近的 WS-Event 版本包括一些改进，诸如使用端点引用（这依靠 WS-Addressing、WS-ReferenceProperties 和引用参数）代替订阅标识符，从而增强与操作性。新的发送模式可以异步地推送事件，并且以后基于扩展点可以添加其他的模式。

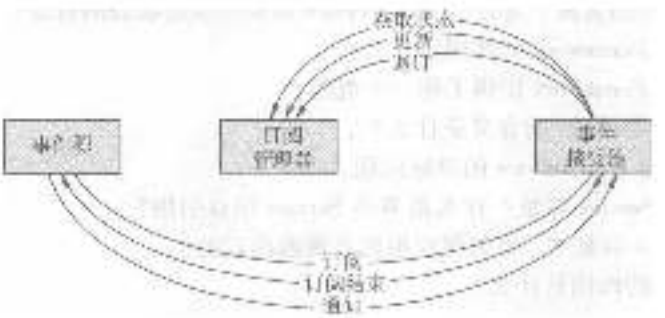


图 7.11 WS-Eventing 消息交换

根据到目前为止的介绍，可以很容易看出：WS-Eventing 规范提供了类似于 WS-Base Notification 的功能。WS-Eventing 和 WS-Base Notification 都可与其他的 Web Service 标准协同工作，诸如 WS-Security 和各类规范，WS-Policy，WS-ReliableMessaging。这两种规范（WS-Eventing 和 WS-Base Notification）之间的主要不同点就是：WS-Eventing 模型需要在"事件过滤器"声明和出现内容的格式之间进行紧耦合，它们使用 XPath 表达式；而 WS-Notification 使用松耦合的基于主题的订阅方法。另一个不同点是：对于发布和订阅型的 Web Service，WS-Notification 支持中介代理技术，而 WS-Eventing 当前并不支持。对于这两种通知规范的更详细的分析和比较，可参见文献 Pallickara 2004]。

最近，OASIS WS-Notification 技术委员会认识到 WS-Eventing 和 WS-Base Notification 都是针对

类似问题的，并且这两者的架构也很类似。有一点值得注意，虽然这两者存在差异，但它们之间的共同点也是很多的，值得进一步研究。为了建立一个统一的基础架构、满足这两个规范的需求，目前我们已经邀请 WS-Eventing 和 WS-Base Notification 的作者一起进行研究。

7.5 小结

对于 Web Service，用户必须能够访问和操纵状态，例如 Web Service 交互结果的数据值。状态信息通常捆绑在有状态的资源中。在 Web Service 部署时将有状态的资源关联到 Web Service，则这种关联称为静态关联。另一种方式是在进行消息交换时，将有状态的资源关联到 Web Service，则这种关联称为动态关联。WS-Resource Framework 主要用于有状态的资源的创建、寻址、检查和生命周期管理。

通过包括实例信息、策略、复合绑定等，有状态的 Web Service 可以在 Web Service 寻址模式中添加更多的动态信息。这需要客户端在运行时能够基于先前的运行时信息唯一地标识服务，可以使用 WS-Addressing 解决这一问题。WS-Addressing 能够在 SOAP 消息头部中描述复合信息路径。基于消息寻址，可以实现中间消息处理、消息转发以及在一条消息路径中传输多个消息。

WS-Notification 是一个相关规范的集合，定义了使用基于主题的发布/订阅模式的标准的 Web Service 通知。对于希望参与通知和标准的消息交换的服务提供者，WS-Notification 规范定义了这些服务提供者所实现的标准的消息交换，发布消息者可以不是服务提供者。与 WS-Notification 相关的 WS-Topics 定义了三个主题表达方式。在订阅请求消息中以及 WS-Notification 系统的其他部分中，可以使用它们作为订阅表达式。WS-Notification 详述了一个 XML 模型，用于描述与主题相关的元数据。

复习题

- 什么是有状态的资源？为什么 Web Service 需要涉及有状态的资源？
- WS-Resource Framework 的作用是什么？
- WS-Resource Framework 提供了哪一个规范？
- 术语“隐含资源模式”的含义是什么？
- 描述有状态的 Web Service 的寻址问题。
- 什么是 Web Service 寻址？什么是 Web Service 端点引用？
- 在 Web Service 寻址中，是如何应用隐含资源模式的？
- WS-Resource 的作用是什么？
- WS-Base Notification 的作用是什么？它支持哪种 Web Service？
- 什么是通知主题？
- 描述在 WS-Topic 标准中使用的三类主题表达方式。
- 什么是 WS-BrokeredNotification？它和 WS-BaseNotification 的差别在哪里？

练习

7.1 当保险代理向保险公司提交保险索赔时，必须提供诸如端点地址这类消息路由和发送信息。在该问题中需要在 SOAP 消息中使用 WS-Addressing 头部以及引用特性和参数。

7.2 对练习 5.2 进行进一步扩展。假设保险索赔必须为有状态的实体，并作为 WS-Resource 进行建模。WS-Resource 维护与每一个保险索赔处理相关的状态。WS-Resource 的实现应该包括 WSDL 定义以及 WS-ResourceProperty 语句。

7.3　在一个家庭应用中，独立装置间可以使用 WS-Notification 的发布/订阅方式进行异步交互。例如，事件可以是基于主题表示的临界因素，诸如火、煤气泄漏和居民的健康状况。编写一个程序来管理如下所示的一个简单的 WS-Notification 主题命名空间。该程序必须能够解析 XML 文件、识别它的树形结构，并能用比较直观的方式向用户进行显示。此外，在程序中用户可以修改树(添加、重命名或删除所需的节点)。在如下主题命名空间中，发布者可以生成的所有事件。

```
<?xml version="1.0" encoding="UTF-8"?>

<wstop:topicSpace name="Event"
targetNamespace="http://publisher.domotic.com/topicnamespace.xml"
xmlns:wstop="http://www.ibm.com/xmlns/stdwip/web-services/
                                                    WS-Topics" >
  <wstop:topic name="Alarm events" >
        <wstop:topic name="Fire presence" />
        <wstop:topic name="Gas presence" />
  </wstop:topic>
  <wstop:topic name="Health events" >
        <wstop:topic name="Arm dislocation" />
        <wstop:topic name="Irregular heartbeat" />
  </wstop:topic>
</wstop:topicSpace>
```

7.4　对练习 7.3 进行进一步扩展，使其可以阅读和存储流入的特定主题的 WS-Notification 订阅消息。可以将订阅存储在一个类似于如下主题命名空间的 XML 文件中。

```
<?xml version="1.0" encoding="UTF-8"?>

<wstop:topicSpace name="Event"
targetNamespace="http://publisher.domotic.com/topicnamespace.xml"
xmlns:domPub="http://publisher.domotic.com/xmlnamespace.xml"
xmlns:wstop="http://www.ibm.com/xmlns/stdwip/web-services/
                                                WS-Topics" >
  <wstop:topic name="Alarm events" >
        <wstop:topic name="Fire presence" />
        <wstop:topic name="Gas presence" />
  </wstop:topic>
  <wstop:topic name="Health events" >
        <wstop:topic name="Arm dislocation" />
        <wstop:topic name="Irregular heartbeat" />
        <domPub:Address name="simpleSubscriber:1234" />
  </wstop:topic>
</wstop:topicSpace>
```

为了实现这一点，需要一个能够发送 SOAP 消息的 WS-Notification 订阅者。

```
<s12:Envelope xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
xmlns:wsnt="http://www.ibm.com/xmlns/stdwip/web-services/WS-
BaseNotification"
xmlns:domPub="http://publisher.domotic.com/xmlnamespace.xml">
  <s12:Header/>
  <s12:Body>
        <wsnt:Subscribe>
            <wsnt:ConsumerReference>
                  <wsa:Address>
                        simpleSubscriber:1234
                  </wsa:Address>
            </wsnt:ConsumerReference>
            <wsnt:UseNotify>
               true
```

```
            </wsnt:UseNotify>
            <wsnt:TopicExpression
                    dialect=http://publisher.domain.com/
                        topicDialect.xsdf>
                        SunPub:Event/Health events
            </wsnt:TopicExpression>
        </wsnt:Subscribe>
    </s12:Body>
</s12:Envelope>
```

7.5 对练习7.3和7.4进行进一步扩展，实现一个简单的WS-Notification事件代理，使其能够接收通知消息，并能将通知消息转发给注册的订阅者。为了实现这一点，需要一个能够发送SOAP消息的WS-Notification发布者。

```
<s12:Envelope xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
xmlns:wsnt="http://www.ibm.com/xmlns/stdwip/web-services/WS-
BaseNotification"
xmlns:dombres="http://booker.domain.com/xmlnamespace.xml">
    <s12:Header/>
    <s12:Body>
        <wsnt:Notify>
            <wsnt:NotificationMessage>
                <wsnt:Topic dialect="dialect1">
                    dombres:Event/Health events/Arm Dislocation
                </wsnt:Topic>
                <wsnt:Message>
                    Arm dislocation detected!
                </wsnt:Message>
            </wsnt:NotificationMessage>
        </wsnt:Notify>
    </s12:Body>
</s12:Envelope>
```

7.6 图7.12显示了一个保险索赔主题树，根主题是保险索赔，后代主题包括财产索赔、健康索赔和伤亡索赔。使用WS-Topic标记法来表示如图7.12所示的保险索赔主题树。



图7.12 保险索赔主题树的样例

第8章　面向服务的体系结构

学习目标

为了处理松耦合的、基于标准的以及协议独立的分布式计算，自动业务集成领域已经补充了一些新的方法和体系结构，面向服务的体系结构就是一个里程碑式的体系结构。利用基于标准的功能服务，可以实现许多SOA的功能。当需要时，既可以调用这些单个的功能服务，也可以将这些功能服务聚合起来，从而创建复杂的应用或者多阶段的业务流程。不仅可以存储和复用构成块服务，而且可以更新或替换这些构成块，并且不会影响其他独立服务的功能或完整性。在SOA中运行的业务操作通常包含对这些不同的服务的调用。随着底层业务处理需求的不同，调用方式通常为事件驱动或异步方式。

SOA采用了基于事件的编程模式。在本章中，我们将要用基于SOA的原理与概念，描述有关技术和方法。阅读本章后，读者将能了解下列关键概念：

- 软件体系结构的概念
- 可靠消息传送的作用
- WS-ReliableMessaging的结构
- 针对SOA应用的事件驱动计算的含义
- 针对SOA应用的事件服务总线集成基础架构
- 事件服务总线中的连通性、集成和可伸缩性

8.1　软件体系结构是什么

构建分布式软件系统需要将不同组织中的资源和系统以及开发实现代码组成一个整体。例如，当自动业务处理跨越企业边界时，将涉及不同组织中的各种系统，并需要利用共享资源，诸如Web Service、业务逻辑组件、安全性系统和后端企业信息系统(如数据库或ERP系统)。在这个环境中，合作伙伴必须不仅需要对核心接口集和标准集取得一致，还需要对如何使用这些核心接口集和标准集取得一致。这一问题的重点是软件体系结构。

计算系统的软件体系结构涉及被构建的系统(软件组件)的结构描述、那些组件的外部可视特性、组件间的相互关系，以及管理设计和演化的原理与准则[Shaw 1996]、[Bass 2003]。外部可视特性假设其他的组件能够了解一个具体的组件，诸如它所提供的接口和服务、性能特性、故障处理、共享资源使用等。

更精确地，软件体系结构是软件系统的高层结构，包括分布式和面向服务的系统，通常根据功能组件和这些组件之间的交互/互联来描述这些软件系统。通过“契约(contracted)”接口，可标识组件并可指派和客户端组件交互的任务[Soni 1995]。组件互联规定了通信和控制机制，并支持实现系统行为所需的各种组件交互。

软件体系结构构成软件密集型系统的主干。它以抽象复用模型的形式表示了资本投资，可将抽象复用模型从一个系统迁移到另一个系统。软件体系结构的重要特性包括[Bass 2003]：

- 软件体系结构是一个足够高层的抽象，能够从整体上查看系统。然而，为了能够进行进一步的分析、制定决策以及规避风险，软件体系结构需要提供足够的信息。
- 结构必须支持系统所需的功能。因此在设计体系结构时，必须考虑到系统的动态行为。

- 软件体系结构必须符合系统质量(可以在服务等级协定或者非功能性需求中获取)。系统质量很可能包括与当前功能相关的性能、安全性、互操作性和可靠性需求，以及灵活性和可扩展性需求。基于灵活性和可扩展性，将来就能以合理的成本实现一些新的功能。因为系统质量的各指标间可能会相互冲突，所以设计时对于体系结构中的各个部分进行折衷是非常重要的，需要权衡各种可选的解决方案，并且需要考虑系统质量中各个指标间的相对优先关系。
- 在体系结构层面隐藏了所有的实现细节。

软件体系结构中的两个必不可少的要素是它的功能需求和系统质量。功能需求指的是系统的预期行为。该行为可以依据服务、任务以及需要完成的功能进行表示。

在1.8节中，已经讨论了系统质量的本质。本章我们将简要地重新分析一下系统质量的本质和含义，以便读者可进一步加深对这个问题的理解。在第15章中将介绍有关SOA功能需求的服务开发技术。

8.1.1 系统质量属性

软件系统的设计主要任务是如何满足系统的功能需求。系统的软件体系结构的设计主要针对系统的非功能性需求或质量需求。可使用质量属性描述系统的质量需求。质量属性是系统功能性之上的系统特性，它决定了系统的技术质量。系统质量属性描述了如何正确地实现系统的行为和结构。可以基于系统行为的一些外部可度量特性而不是系统的内部实现来评判系统质量。换句话说，用户可以根据用户重视或关心的一些特性来评判系统质量。

有两类质量属性：运行时质量属性和开发时质量属性[Bass 2003]。运行时质量属性向用户提供数值，并主要与短期的竞争差异有关。开发时质量属性主要提供了业务数值(与直接向终端用户提供数值形成了对照)，并主要与长期的业务竞争相关。

运行时质量属性包括：可用性(易用性、易学性、可记忆性、效率等)，可配置性和保障性，正确性、可靠性、可获得性，性能(吞吐量、响应时间、传输延迟、时延等)等技术性系统需求，安全性和容错性等安全特性，运营方面的可伸缩性(包括增加用户和系统节点、提高事务处理能力等)。

对于软件体系结构而言，除了开发满足用户需求的系统，开发流程方面(体系结构、设计、代码等)的特性扮演了一个重要的角色。这些不仅影响了与软件开发相关的工作量和成本，而且支持未来的变化和使用(维护、增强或复用)。开发时质量需求方面包括：可维护性、可扩展性、可进化性、可组装性和可复用性。其中，可扩展性指的是以后可添加(没有事先确定的)功能的能力；可进化性指能够支持新性能或能够利用新技术的能力；可组装性指能够使用即插即用组件组成系统的能力；可复用性能够在以后的系统中进行复用的能力。

为了更好地了解开发时质量的影响，可以客户/服务器体系结构模式为例。可按照不同的职责分离客户端和服务器，它们分别为用户和提供者。客户/服务器体系结构描述了提供者和用户之间的协作关系。这种分离增强了系统的可变性和可升级性。对于客户端来说，它无须了解服务的实现是否发生了变化，也无须了解提供服务的服务器的数量是否发生了变化。此外，增加新的客户端也不会影响到服务器。虽然这种计算分离模式可以提高可靠性，但是这种方式也有缺点。由于增加了网络流量，从而增大了被攻击的风险，因此需要专门考虑安全性方面的问题。

在规定质量属性需求时，与确定功能需求的方式一样。质量属性集通常包含5类重要的质量属性(独立于不同系统)：可变性、性能、可获得性、安全性和可用性，具体可参阅文献[Bass 2001]。

8.1.2　体系结构方面的常见议题

确定体系结构时，非常重要的一点就是需要从大范围的技术系统的角度出发。体系结构的确定即使没有影响到整个系统，也至少会影响到系统的一些不同的组成部分，因此需要从全局的角度来考虑体系结构的取舍，并需要在整个系统中进行权衡。(分布式)软件体系结构包含(但不仅限于)以下一些关键问题[Bass 2003][Malan 2002]：

- 元体系结构：这是关于体系结构愿景、类别、原理、关键的通信和控制机制以及一些重要概念的一个集合。它将影响到高层决定，从而很大程度上影响到完整性和系统的结构。通过类别、组合模式或交互、原理和准则，元体系结构可以排除一些结构方面的选择，并指导如何选择可行的体系结构以及在这些选择之间进行权衡。
- 体系结构模式：随着时间的推移，软件开发者可在结构化系统中辨别出一些模式，并且当这些模式得到广泛应用时，它们将变为主流设计。通过模式，架构师可以从解决方案的一个问题或者一个方面开始着手，从而找到最合适的模式。然后，架构师可以进一步定义应用所需的其他功能。例如，客户/服务器、三层和多层(分层)体系结构等经过验证的模式已经在工厂业得到普遍应用。在 SOA 中也逐渐出现了一些体系结构模式，可用于系统分解以及用以实现系统特性[Chappell 2004]和[Endrei 2004]。
- 体系结构视图：一种理想的状况是基于许多补充视图或模型来构想软件体系结构。尤其，结构化视图会按照组件和组件间的关系帮助文档化以及与体系结构通信。在评估诸如可扩展性等体系结构质量时，体系结构视图是很有价值的。在全面考虑组件之间如何通过交互实现它们各自的功能时，以及在评估体系结构对应用的各种可能的影响时，行为视图是很有作用的。在评估诸如性能和安全性等运行时质量时，行为模型尤其有用。执行视图可以帮助评估物理分布的优劣，有助于文档化，并可帮助进行通信决策。
- 系统分解原则和良好的接口设计：这些标识了系统的高层组件和它们之间的关系。它们直接针对系统的合理分解，而无须深入了解一些不必要的细节。然后通过良定义的接口和组件规范，组件的外部可视特性将变得更精准、更明确，并且关键的体系结构机制将更详细。有些问题是非常重要的决策，并且在进行体系结构方面的决策时需要仔细斟酌，诸如是否将组件组装在一起？各部分是否与所要求的系统完整性、一致性相符。
- 关键的体系结构设计原则：这些包括抽象、问题分解、推迟决策、简化和诸如接口隐藏、封装等相关技术。

对于基于 SOA 的应用，体系结构扮演了一个重要的角色。在本章中，主要讨论了系统功能需求和质量需求，有关 SOA 的设计原则和特性将在第 15 章进行讨论。

8.2　SOA 回顾

正如我们在 1.6 节中已经获知的，SOA 是一个支持松耦合服务的元体系结构样式，能够以互操作的、与技术无关的方式实现业务的灵活性。在 SOA 中，通过网络可以访问粗粒度的软件资源和功能，业务基础服务(business-aligned sercice)提供了这一访问能力，它们是使用基于接口的服务描述来实现的。这些服务是良定义的、自包含的(基本的)业务处理步骤，诸如“开发票”、“顾客查找”、“账单客户”，它们独立于其他服务的状态或上下文。在 SOA 中，业务基础服务是构建灵活的、动态的、可重构的、端到端的业务流程的基础。SOA 的另一个重要特性是通过标准的方式来调用这些业务基础服务，并且这些业务基础服务是松耦合的，调用者无须了解技术细节以及服务提供者的位置。因此，需要客户信息的任何其他应用都可以调用“账单客户”这样的服务。

SOA 主要致力于创建设计样式、技术和处理框架，从而企业可以有效地、经济地开发、互连以及维护企业应用和服务。SOA 的目标并不新，它利用了一些原有的技术成果，诸如模块化编程、代码复用和面向对象的软件开发技术。在软件应用的“抽象开发”方面，SOA 达到了一个新的技术高度。对于业务流程、信息和企业资产的有效组织和部署，SOA 提供了一整套的准则、原理和技术。SOA 支持竞争性的业务环境所需的战略计划和生产率水平，从而企业可以在已有资产的基础上及时地适应新的、不断变化的业务需求。SOA 基于与消息关联的服务间的组合和交互。通过策略可以控制 SOA。以这种方式可将业务流程和应用例行地映射到服务，这些服务可被使用、修改、构建或编配。

通过 SOA，开发者可以解决实现中的许多复杂的难题，诸如分布式软件、应用集成、多平台和多协议、大量的访问设备，并可充分发挥互联网的潜力。SOA 的目标就是简化应用集成的难度，并使得这些应用可以无缝地运行。通常认为，SOA 代表了企业技术解决方案的未来的发展方向。SOA 提供了灵活性，可利用新的和已有的应用，并可将它们抽象为模块化的、粗颗粒的服务，而这些服务映射到各个业务功能，从而可以满足用户所需的业务灵活性和敏捷性。通过提供独立的、可复用的、自动化的业务流程(可作为服务进行部署)，良构的 SOA 使得业务环境具有灵活的基础架构和处理环境，并对这些服务的使用提供了一个强大的体系结构基础。

在 SOA 中，服务是被暴露的具有三个基本特性的功能部件。基于 SOA 的服务是一个自包含的(例如服务维持它自身的状态)、平台独立的(例如服务的接口契约是平台独立的)服务，可以动态地定位和调用这些服务。SOA 的主要价值是可以复用已有的服务，既可以独立地使用这些已有的服务，也可以将这些服务作为应用的一个组成部分，通过编配大量的服务和信息，可以实现更复杂的功能。简单服务可以用不同的方式进行复用，也可以将简单服务与其他的服务组合起来执行一个具体的业务功能。

SOA 中的服务具有下列特性[Channabasavaiah 2003]：

(1) SOA 中的所有功能都定义为服务。这包括纯业务功能、由低层功能组成的业务事务和系统服务功能。在 SOA 的复杂应用中使用的服务可以是全新的服务实现，它们也可以是被改写和包装的原有应用的一部分，或者它们是上面这些形式的组合。

(2) 所有的服务都是彼此独立的。服务操作被外部程序认为是不透明的。服务的不透明性保证了外部组件既不知道也无须关心服务是如何完成它们的功能的。外部组件仅期待这些服务返回所期望的结果。提供所需功能的应用所采用的技术以及它们的位置都隐藏在服务接口后面。

(3) 服务接口是可调用的。这意味着：在体系结构层，无须区分服务是本地的(在一个具体的系统中)或者远程的(当前系统的外部)；无须区分调用所采用的互连模式或协议；也无须区分连接所需的基础架构组件。服务既可以在同一个应用中，也可以在一个完全不同的系统的不同的地址空间中。

SOA 对服务接口和服务实现进行了清晰的分离。在 SOA 应用的规划、开发、集成和网络应用平台的管理中都应用了松耦合原则。对与企业范围的应用以及跨企业的应用来说，松耦合原则是必不可少的。如图 8.1 所示，为了定义和封装一个可复用的业务功能(在图 8.1 中使用“内部处理”这一术语进行表示)，需要使用一些良定义的接口。

虽然可以使用不同的技术来实现 SOA，诸如像 J2EE、CORBA、JMS 这样已有的中间件技术，然而对于最大程度地实现 SOA 的服务共享、复用和互操作，Web Service 是首选方式。与以前的中间件框架不同，正在不断发展的 Web Service 标准集支持以真正的平台独立、语言独立、松耦合的方式实现服务集成。通过封装，Web Service SOA 实现减少了应用复杂性。通过清晰的接口定义，最小化各方所需了解的细节。此外，Web Service 可用于实现遗留系统的及时集成和互操作性。

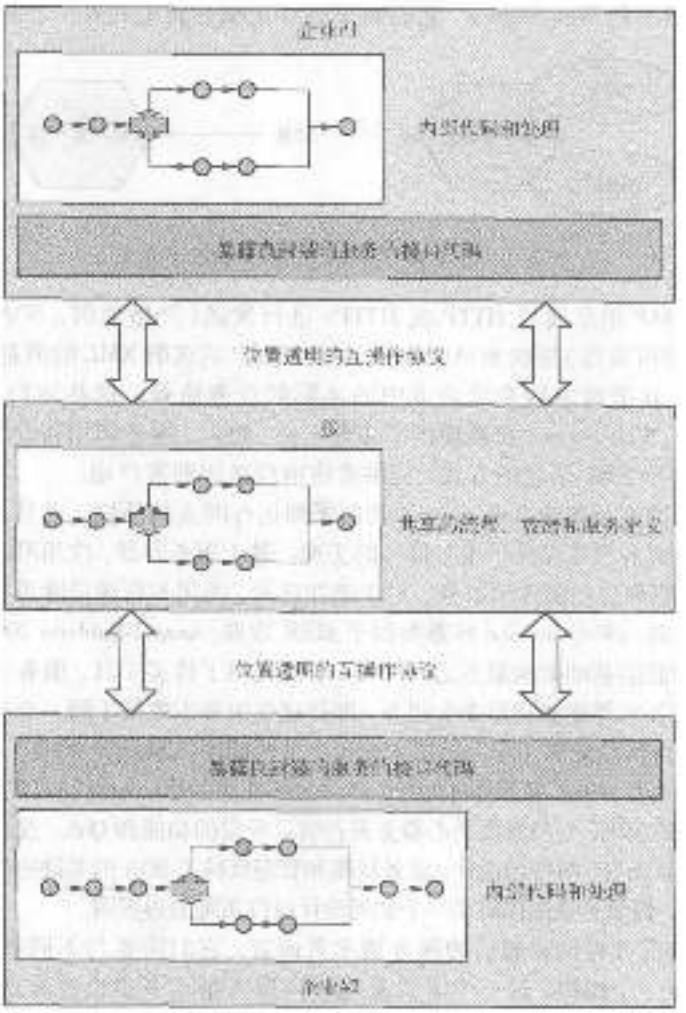


图 8.1　SOA 中的互通式服务

引自：[Channabasavaiah 2003]

当使用 Web Service 实现 SOA 时，可使用 Web Service 描述语言（参见第 5 章）表示接口描述。对于业务服务的可用操作和所需的参数，WSDL 可进行完整的描述。此外，WSDL 描述了如何绑定到那些服务，指定了所需的协议和端点。一旦描述了 Web Service，可将描述发布在一个符合 UDDI 标准的信息库中。然后，客户端将能查询该信息库并可发现合适的服务。

8.3　SOA 中的服务角色

SOA 和 Web Service 解决方案包括两个众所周知的关键角色：服务请求者（客户端）和服务提供者，它们通过服务请求（参见 1.6 节）进行通信。服务请求是按照 SOAP（参见图 8.2）进行格式化的消息。在第 4 章中，我们已经分析了 SOAP 本质上是一个平台中立、厂商中立的标准。从而，服务请求者和服务提供者之间可以是一种松耦合的关系。若两方是跨互联网的不同的组织或企业，这种松耦合的关系尤为重要。然而，SOA 并不是一定需要使用 SOAP。例如，在 SOAP 出现之前，一些公司使用 IBM 的 WebSphere MQ 交换它们之间的 XML 文档，因为这类基础架构不使用

SOAP，所以它们显然不是 Web Service，它们是 SOA 中的服务调用另外一个例子。

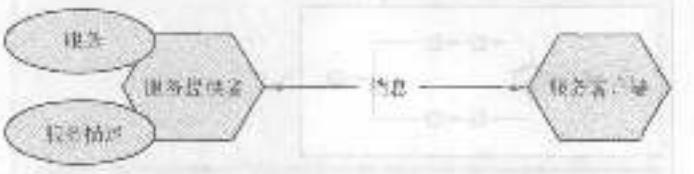


图 8.2 服务客户端角色和服务请求者角色

在互联网上，SOAP 消息通过 HTTP 或 HTTPS 进行发送（严格地讲，SOAP 是协议独立的）。运行时服务（SOAP 侦听程序）接收 SOAP 请求，抽取 SOAP 请求的 XML 的消息体部分，把 XML 消息转换成本地协议，并把请求转交给企业中的实际的业务流程。这些运行时能力可以驻留在 Web Service 容器中。Web Service 容器提供了诸如定位、路由、服务调用和管理等能力。最后，通过在 SOAP 信封中携带 XML 消息的方式，提供者将响应送回到客户端。

服务容器提供了高度分布的 Web Service 的部署和运行时支持环境。尤其，服务容器是抽象服务端点的具体呈现。服务容器提供了服务接口的实现。基于服务容器，应用可以监控和管理所支持的组件，以及监控和管理这些组件的服务。对于诸如启动、关闭和资源清除等生命周期管理，服务容器也提供了相关工具。Web Service 容器类似于 J2EE 容器[Anupol-Subhanu 2005]。Web Service 容器充当了业务服务和底层基础架构服务之间的接口，并提供了许多工具，服务实现在处理中可以使用这些工具。一个服务容器能够驻留多个服务，即使这些服务不隶属于同一个分布式处理。基于线程池，服务的多个实例可以附属于单个容器中的多个侦听程序[Chappell 2004]。

对于服务客户端而言，SOA 服务是可见的，然而 SOA 服务的底层组件的实现是透明的。服务消费者并不需要关心服务的实现，它只需要关心服务是否满足所需的功能和 QoS。这表示了 SOA 的客户端视图。对于服务提供者而言，组件的设计、服务暴露和管理反映了 SOA 的关键的体系结构和设计方案。对于提供服务的组件，提供者视图提供了一个如何设计组件实现的透视图。

对于必须直接和服务提供者通信的服务请求者而言，它们需要与不同的服务请求者进行协商并预定服务。对于一个组织，另一个方式是直接向服务请求者提供组合功能。这个角色通常称为服务聚集者（Service Aggregator，或称作服务聚集程序）。服务聚集者（可参见 1.6.3 节）承担双重角色。首先，通过创建复杂的高层服务，服务聚集者能够构建一个完整的"服务"解决方案，所创建的服务能够提供给服务客户端，因此服务聚集者可充当一个应用服务提供者。使用诸如 BPEL（我们将在 9.7 节中讲行描述）这样的专门的编排语言，服务聚集者可以完成这种聚合。其次，由于服务聚集者需要向其他的服务提供者请求和预定服务，因此它又充当了服务请求者。该流程如图 8.3 所示。

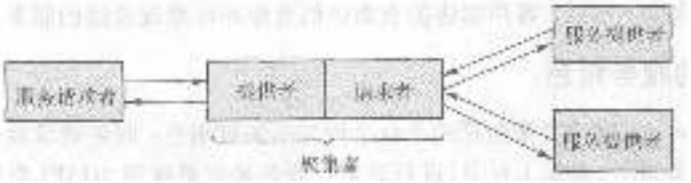


图 8.3 服务聚集者的角色

请求者可以直接从服务聚集中获益。服务聚集可以视为一种特殊的服务代理，将请求者所需的所有服务聚集在一起。然而这也带来一个问题，即服务请求者如何在提供服务的多个服务

者中选择合适的服务提供者？基于诸如 UDDI 这样的注册库服务，服务请求者可以从注册库中发现合适的服务提供者，从而保留了选择应用服务提供者的权利。SOA 技术（诸如 UDDI）、安全和隐私标准（诸如 SAML 和 WS-Trust）引入了一个称为服务代理的第三方角色［Colan 2004］。

服务代理（也称为认证授权，参见 11.3 节）是一个受信任方，它强制服务提供者遵循隐私方面的法律和规章制度。假如没有相关的隐私法律，服务代理将强制服务提供者遵循行业惯例，从而保证了得到认可的服务提供者所提供的服务将遵循本地法规，并在客户和合作伙伴之间创建了一个更可信的关系。服务代理将维护可用的服务提供者的一个索引。通过提供相关服务的更多的信息，服务代理能够进一步提高应用服务提供者注册的价值。一些术语可能有些差异，如可靠性、信赖度、服务质量、服务等级协定以及可能的补偿路径等。

在图 8.4 中，服务代理充当了一个介于服务请求者和服务提供者之间的中介。我们在图 1.4 中所示的传统的 Web Service SOA 可归入这一类。服务注册库（UDDI 运营者）是一个专用的服务代理实例。在这个配置中，UDDI 注册库充当了一个代理。其中，服务提供者使用 WSDL 发布所提供的服务的定义，而服务请求者则查找有关可用服务的信息。

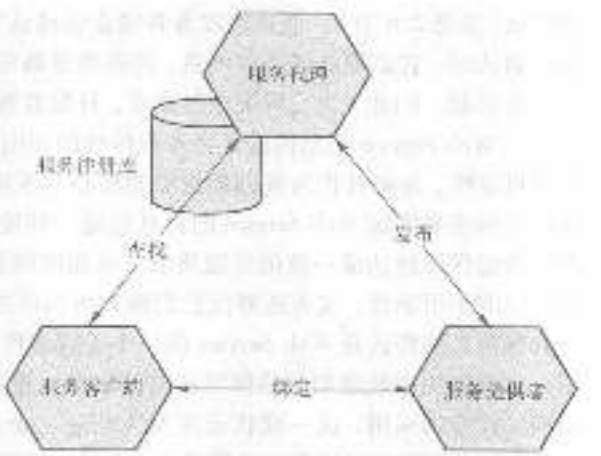


图 8.4　服务代理

SOA 的实现基于服务间的消息交换，通常选择 SOAP 消息作为相互通信的方式，并且通常将服务定义为可网络寻址的实体，它们可发送和接收消息。当多个 Web Service 为了完成一个涉及服务编配的普通任务时，这些 Web Service 间需要进行交互，它们不能依赖私有协议进行消息交换，而是需要使用通用协议，这样才能保证端到端的可靠的、互操作的消息传送。因此，在重点讨论基于消息处理和通知的基础架构层之前，我们将介绍可靠的消息传送的概念。基础架构层使得 SOA 的实现更容易。对于事件驱动的 SOA 实现而言，可靠的消息传送是一个关键的技术标准。对于 Web Service 的协调和事务行为，可靠的消息传送也是一个关键技术。

8.4　可靠的消息传送

SOAP 的主要目标就是打包消息接受者所需的数据，并绑定到一个传输操作。SOAP 的主要特性之一就是在不可靠的通信信道上（诸如互联网）交换端点间的消息，并应用不可靠的数据传输协议，例如 HTTP、TCP、SMTP 和 FTP。由于这些协议不提供 Web Service 应用所需的可靠的消息传送服务，因此这些协议被认为是不可靠的。此外，这些协议仅将消息路径视为一对互连的端点。有些 Web Service 的实现需要在传送过程中修改 SOAP 消息，或者在抵达最终目的之前需要委托其他服务进行处理，因此需要从整体上考虑传输的目的地以及如何传送到目的地。在发送 SOAP 消息的过程中，可能需要从一种传输协议转换为另一种协议，例如从 HTTP 转换为 TCP。复杂的消息传送需要可靠的寻址机制，非常遗憾的是 SOAP 核心规范并不提供可靠的寻址。此外，Web Service 可能是在不同的平台上实现的，例如 .NET 或 J2EE。因此为了保证预期的接收者能够收到消息，将需要更强壮的基于 SOAP 的规范，并且即使消息所抵达的 Web Service 的实现不同，消息寻址方式也必须保持一致，并可使用异构的消息传送协议。

8.4.1 可靠的消息传送的定义和范围

消息层的可靠性通常称为可靠的消息传送。Web Service 中的可靠的消息传送指的是：1）无须关心具体的传输协议；2）基于提供 QoS 的 SOAP 协议；3）可靠地传送消息。在指定可靠的消息传送特性时，将涉及三个同样重要的方面。首先，我们必须弄清楚：消息的发送者和接收者是否知道消息已经被真正地发送出去或接收到，以及接收到的消息与发送的消息是否一样。其次，我们必须确保发送给意向接收者的消息必须一次发送成功。最后，我们必须保证：消息的接收顺序必须与发送顺序完全一样。Web Service 若要真正成为企业级应用技术，可靠的消息传送就是首先需要解决的问题之一。

在互联网应用开发中，可靠的消息传送一直是一个非常重要的问题。本质上，互联网是不可靠的。尤其，连接发送者和接收者的协议不支持可靠的消息传送构建，诸如不支持消息标识符和收到确认（参见 2.6 节）。消息接收者必须能够确认它们已经真正收到了消息。当消息发送者没有收到确认时，它必须能够缓存消息，并需要重新发送消息。目前的互联网的一些基础技术并不支持这类机制。因此，为了解决这些需求，开发者被迫实现了一些专有协议和专有技术。

由于 Web Service 消息传送正在取代传统的面向消息的中间件，因此 Web Service 消息传送需要提供可靠性，并将其作为基础架构层的核心 QoS 能力，并且需要使用具有互操作性的、得到广泛应用的标准来实现 Web Service 的消息传送。即使在网络、软件和硬件出现故障的情况下，可靠性依然能保证能达成一致的发送质量。从而使得消息的发送者和接收者能够克服互联网/内联网环境中的不可靠性，或者能够使它们确知所出现的故障。

确保消息的传送是 Web Service 的一个关键组件。传统上，通常在位于消息交换和事务系统的端点上的应用中处理如何确保消息的可靠性。随着 Web Service 环境中的安全、可靠的消息交换的相关标准的采用，这一现状正在发生改变。企业不再需要开发代价巨大的、特定的解决方案，这些解决方案并不能真正地跨平台、互操作地解决可靠性。

为了保证不同的传输等级，开发人员有时需要编写传送消息的代码，标准的可靠消息传送协议简化了这一代码编写任务。底层的基础架构验证消息已经在端点之间进行了合适的传送，并可在需要时转发消息。提供交付保障可能需要消息转发、重复消息删除以及消息收到确认等处理，应用并不需要任何额外的处理逻辑。当前，Web Service 中的可靠的消息传送有两类：WS-Reliability 和 WS-ReliableMessaging。前者是由 Sun、Oracle 以及其他公司作为 OASIS 标准发布的，后者是由 IBM、微软、BEA、TIBCO 发布的。这些是针对收到确认基础架构的规范的实例。收到确认基础架构影响了 SOAP 可扩展性模型。这两个规范所定义的协议都独立于下面的传输层。

这两个新的可靠性规范在很大程度上是重叠的，通常期望这来年规格规范最终能够合并。这两个相互竞争的规范的主要不同点是：WS-ReliableMessaging 包括了其他关键的 Web Service 规范（诸如 WS-Security 和 WS-Addressing）的用法。实际上，这意味着：为了提供可靠性特性，WS-ReliableMessaging 使用了其他标准的特定表达；WS-Reliability 支持可靠的消息传送特性，但是并不依赖其他 Web Service 标准所提供的。由于 WS-ReliableMessaging 本质上可与本章将要讨论的其他标准（诸如 WS-Security 和 WS-Addressing）协调一致，因此下面我们将主要描述 WS-ReliableMessaging。WS-Reliability 规范的详细信息和使用样例可参阅文献［Iwasa 2004］。

8.4.2 WS-ReliableMessaging

WS-ReliableMessaging 协议可确保能够检测出未收到的 SOAP 消息和重复的 SOAP 消息，并且

收到消息的顺序与这些消息的发送顺序一致。可以按不同的交付保障等级进行消息交换。对于服务端点地址和策略的标识，WS-ReliableMessaging 协议需要依赖其他的 Web Service 规范，通过 WS-Addressing 协议(参见 7.2.1 节)，可以实现传输中立的、双向的、同步的、异步的、有状态的服务交互，这些服务交互可以跨网络(包括端点管理器、防火墙和网关等)。WS-ReliableMessaging 协议可以与 WS-Addressing 协议协同工作。对于跨平台的互操作，每一个规范都定义了一个 SOAP 绑定。

对于不同的可靠传输基础架构，WS-ReliableMessaging 标准提供了一个框架[Bilorusets 2005]。更具体地，WS-ReliableMessaging 协议确定了一些不变式，这些不变式是由可靠的消息传送端点所维护的。WS-ReliableMessaging 协议还确定了一些指令，这些指令用于跟踪和管理消息序列的发送。WS-ReliableMessaging 是一个具有互操作性的协议，协议使用"可靠的消息传送源"和"可靠的消息传送目的地"提供应用源和应用目的地，并保证发送的消息能够真正抵达目的地。WS-ReliableMessaging 模型如图 8.5 所示。WS-ReliableMessaging 模型书实现是分布式的，跨越最初的发送者和最终的接收者(服务)。该图也表明了：在最初的发送者和最终的接收者之间，可以插入多个接收者。消息可基于可靠的消息传送协议进行传送。首先，两个交互的端点确定了消息交换的先决条件。紧接着，发送者格式化并交给传输协议的消息。然后，接收者的消息传送处理程序将消息转发给合适的接收者。最终，接收者的消息传送处理程序将消息转发为接收者的应用程序能够使用的形式。消息传送基础架构可以透明地处理可靠的消息传送操作，例如由于传输丢失而导致的重新发送。其他的一些端到端的特性(诸如按序传送)则需要消息传送基础架构和接收者应用进行协作[Cabrera 2005d]。

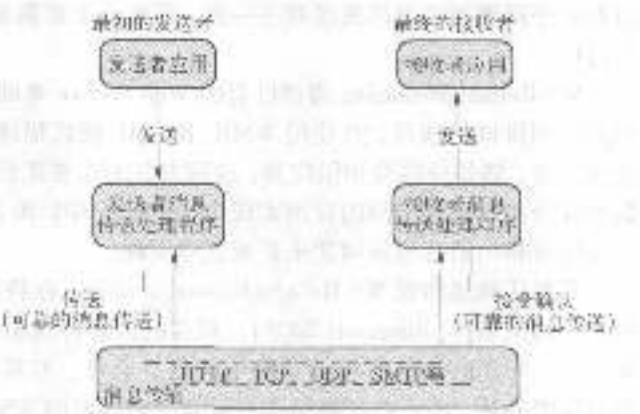


图 8.5　可靠的消息传送模型

WS-ReliableMessaging 有两个主要的特征：

(1) 消息处理程序间的可靠性协议。在实现中使用了消息序列这一方法，每个消息都通过一个序列号进行标识，并使用重新发送和收到确认机制。

(2) 可靠性 QoS 约定。对所有的通信方提供了发送保障和 QoS。

在 WS-ReliableMessaging 中，对于发送的保证被指定为发送保障。对于消息从最初的发送者发送到最终的接收者，实现 WS-ReliableMessaging 协议的端点提供了发送保障。可靠的消息传送源和可靠的消息传送目的地的责任就是实现发送保障，或者引发一个错误通知。端点可以提供 4 类基本的发送保障，并且 WS-ReliableMessaging 也支持这 4 类发送保障，它们是[Bilorusets 2005]：

- 至少一次发送：该特性保证可靠需要发送的所有信息都送交到目的地，或者在至少一个端点给出出错信息。
- 至多一次发送：该特性保证需要发送的所有信息都最多送交一次，从而没有重复发送，或者在至少一个端点给出出错信息。
- 正好一次发送：该特性保证需要发送的所有信息都将送交一次，从而没有重复发送，或者

在至少一个端点给出出错信息。

- 按顺序发送：该特性可使得目的地消息序列的顺序与发送应用程序的发送顺序一致。该发送保障可确保消息没有重复或丢失。

由于 Web Service 体系结构具有传输独立的特性，所有的传送保障都与通信传输或传输组合无关。对于许多类别的传送故障，开发者并不需要额外编程进行处理，因此 WS-ReliableMessaging 简化了系统开发。

WS-ReliableMessaging 利用了可靠的消息传送和事务协议中的算法。更具体地，WS-ReliableMessaging 提供了一个灵活的收到确认模式，接收者可以有效地运送范围内的消息(已经收到的或没有被收到的)。WS-ReliableMessaging 也提供了一个有效的排序机制，可确保接收者处理消息的顺序能够与消息的发送顺序一致，虽然由于重新发送和多路径路由，有时需要对消息重新进行排序。

WS-ReliableMessaging 遵循已有的 Web Service 基础架构。尤其，WS-ReliableMessaging 位于已有的应用协议的顶部，并使用 WSDL 和 XML 模式描述接口[Box 2003]。这意味着：应用涉及两个交互方，诸如分销商和供应商，这两方并不需要重新设计它们的应用层消息模式或交换模式。假如在分销商和供应商的应用实现中，使用 WSDL 和 XML 模式元素定义服务的业务接口，则可以使用可靠的消息传送功能来扩展这些实现。

若要正确地实现 WS-ReliableMessaging 协议，在将消息从源发送到目的地之前，首先需要满足若干约束条件[Bilorusets 2005]。可靠的消息传送源必须有一个端点引用，这个端点引用唯一地标识了可靠的消息传送的目的地端点。此外，可靠的消息传送源也必须了解目的地的策略。若有策略的话，消息必须遵循这些策略。可以使用 WS-Policy(参见章节 12.4.1 节)中定义的方法来表示策略断言。假如消息需要进行安全的交换，则可靠的消息传送源和可靠的消息传送目的地必须已经建立了一个安全性上下文。WS-ReliableMessaging 标准认为使用 WS-Security 和相关标准的通信是安全的。

为了保证协议在生命期中行为正确，采用了两个不变式[Bilorusets 2005]。首先，可靠的消息传送源必须对每一个可靠的消息都赋予一个序列号。其次，可靠的消息传送目的地必须能够发出收到确认，并且收到确认必须与接收到的消息的序列号一致，需要排除没有收到的消息的序列号。

1. WS-ReliableMessaging 的结构

WS-ReliableMessaging 主要围绕三个核心要素进行开发：

- 序列：WS-ReliableMessaging 规范将两个端点之间的消息交换表示为一个序列，无论是交换一个消息，还是将整个消息序列作为一组进行交换。协议使用 < Sequence > 元素标识和记录消息组，序列中的每一个消息都被赋予了一个唯一的序列标识符(一个绝对 URI)。
- 消息号：序列中的每一个消息都有一个序列号。消息的序列号为升序。基于这种编号模式，可以更容易地检测丢失或重复的消息，并简化了收到确认的生成和处理。
- 收到确认：收到确认表示消息已经成功地发送到它的目的地。

2. WS-ReliableMessaging 的样例

清单 8.1 是一个包含 < Sequence > 元素的消息的样例。例子中的消息是序列中的第三个，一些 URI 标识了这个样例。正如清单所示，< Sequence > 头部除了必须包含 < Identifier > 元素和 < MessageNumber > 元素，还可以包含 < LastMessage > 元素，该元素表示了一个特定消息是交换中的最后一个消息。< LastMessage > 元素没有任何内容。

清单 8.1　WS-ReliableMessaging 中的序列元素样例

```
<Soap:Envelope
  xmlns:Soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
  xmlns:wsa="http://www.w3.org/2004/12/addressing">
  <Soap:Header>
       ...
    <wsrm:Sequence>
       <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
          ...
       <wsrm:MessageNumber> 3 </wsrm:MessageNumber>
       <wsrm:LastMessage/>
          ...
    </wsrm:Sequence>

  </Soap:Header>
  <Soap:Body>
     <GetOrder xmlns="http://supply.com/orderservice"
        ...
     </GetOrder>
  </Soap:Body>
</Soap:Envelope>
```

WS-ReliableMessaging 建议使用头部元素 < SequenceAcknowledgement > 。对于特定序列中的一个或多个消息，< SequenceAcknowledgement > 元素将返回一个收到确认，收到确认既可以在通常的响应消息中，也可以在一个专门创建来返回收到确认的响应消息中。

清单 8.2 是一个包含 < SequenceAcknowledgement > 元素的消息的样例。如该清单所示，一个收到确认使用了许多 < Acknowledge mentRange > 元素，这表示了序列中被确认的消息的范围是不连续的。清单中的例子表示了：可靠的消息目的地已经收到消息 1、2，4，但是还没有收到消息 3。

清单 8.2　WS-ReliableMessaging 中的收到确认元素的样例

```
<Soap:Envelope
 xmlns:Soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
 xmlns:wsa="http://www.w3.org/2004/12/addressing">
 <Soap:Header>
      ...
  <wsrm:SequenceAcknowledgement>
     <wsrm:Identifier> http://supply.com/abc </wsrm:Identifier>
     <wsrm:AcknowledgementRange Upper="2" Lower="1"/>
     <wsrm:AcknowledgementRange Upper="4" Lower="4"/>
     <wsrm:Nack> 3 </wsrm:Nack>
  </wsrm:SequenceAcknowledgement>
  <Soap:Body>
      ...
  </Soap:Body>
</Soap:Envelope>
```

可靠的消息传送并不需要一个显式的协调者(coordinator)。当使用 WS-ReliableMessaging 时，参与方必须认识到：协议基于 SOAP 消息头部中的信息。消息序列既可以由发起者/发送者建立，也可以由 Web Service 建立。在建立双向关联时，通常由这两方一起建立。当发起者试图建立一个序列时，请求中的 WS-ReliableMessaging < Sequence > 头块将告知服务这个请求。假如最终的接收者同意在这个行为中进行合作，则可通过响应消息中的 < SequenceAcknowledgement > 头块进行表示。拒绝则意味着不建立序列。另一种方式是在请求时，服务建立序列。这通常发生在最

初的消息交换中，并通常在双向通信中使用。

WS-ReliableMessaging 利用了 WS-Addressing 的能力，从而可以按几种不同的方式异步发送消息[Box 2003]。与对每一个收到的消息发送收到确认不同，WS-ReliableMessaging 允许目的地在一个控制元素中累积它所收到的所有消息的收到确认。既可以在它自己的消息中发送这个控制元素，也可以在随后发送给源的的应用消息(例如请求/应答会话中的响应消息)中包含这个控制元素。此外，在 WS-Addressing 规范中可定义故障抽象特性，而在 WS-ReliableMessaging 中定义的故障定义则引用了故障抽象特性。

8.5 企业服务总线

在 SOA 实现中，为了解决系统异构性和信息模型不匹配的问题，可以使用支持集中星型(hub-and-spoke)集成模式的 EAI 中间件(参见 2.9 节)。集中星型模式在客户端和服务器模块之间引入了一个集成层，并且必须支持和已部署的基础架构和应用的互操作性，并能和它们共存，而不是试图取代它们。然而，这种方式也有缺点，中心点可能会失效并可能很快就变为一个瓶颈。

诸如 SOA 这样的可伸缩的分布式体系结构需要应用一个星座模式(参见 8.5.4 节)。这将需要一个针对 Web Service 的可管理的集成基础架构，并且需要 SOA 与企业服务总线(ESB)的概念能够结合起来。企业服务总线将是本节讨论的主题。这一方法背后的两个关键思想是：1)参与集成的各个系统为松耦合的关系；2)将集成逻辑分成不同的、容易管理的模块[Graham 2005]。

企业服务总线是一个基于消息的开放标准，用于基于 SOA 的解决方案的实现、部署和管理，可装配、部署和管理分布式的面向服务的体系结构。ESB 是一个由中间件技术实现的基础架构集。中间件技术可用于 SOA 的实现。在异构平台上运行的应用之间以及使用不同的数据格式的应用之间会存在若干不一致的地方，中间件技术有助于在一定程度上缓解这个问题。ESB 支持服务调用、消息和基于事件的交互，并能具有合适的服务等级和可管理性。通过基于标准的适配器和接口，ESB 提供了较大粒度的应用和其他组件之间的互操作性。总线既协助传输又协助转换，从而使得这些服务可以分布在完全不同的系统和计算环境中。

SOA 将应用视作服务，ESB 提供了 SOA 的实现框架。ESB 主要用于对应用进行配置，而不是用于编码，也不是对不同的应用进行硬连接。ESB 是一个轻量级的基础架构，提供了即插即用的企业功能。它负责服务间的合适的控制、流，甚至所有消息的转换。ESB 可以使用各种可能的消息传输协议。ESB 和应用以及各个集成组件相互协作，可对服务进行装配，从而可构成复杂的业务流程。这些复杂的业务流程反过来又可以自动化企业中的业务功能。随着 ESB SOA 的实现，先前孤立的 ERP、CRM、供应链管理、财务系统以及其他的遗留系统都可 SOA 化，并可集成。这一方式将比依赖定制的、点到点的编码或专有的 EAI 技术更有效。最终结果是，基于 ESB 可以使用驻留在已有系统上的应用逻辑片段和/或数据，从而可以更容易地创建新的复杂应用。

ESB 分布式处理基础架构了解应用和服务，并使用基于内容的路由工具来决定如何与应用和服务进行通信。本质上，ESB 对于驻留的服务提供了“坞站”，可对驻留的服务进行装配和编配，并可将它们提供给总线上的其他任何服务使用。一旦将一个服务部署到服务容器，该服务就成为了 ESB 的一个有机组成部分，连接到 ESB 上的任何应用或服务都可使用该服务。如图 8.6 所示，服务容器驻留、管理、动态地部署服务，并可将服务绑定到外部资源，诸如数据源、企业应用和多平台应用。

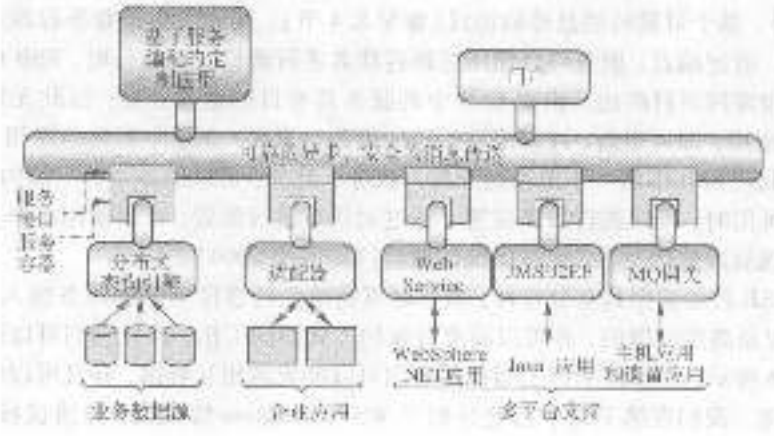


图 8.6　ESB 连接各种应用和技术

概念上，ESB 已经从中间件产品中的存储和转发机制发展为多种技术的组合，诸如 EAI、Web Service、XSLT 和编配技术(诸如 BPEL)。为了实现 ESB 的运作目标，它也吸取了传统的 EAI 代理的功能。EAI 代理可提供集成服务，诸如连接性、基于业务规则的消息路由、连接不同应用的适配器[Chappell 2005a]。基于 SOA 的这些能力以高度分布的方式分散在网络上，并封管在各个独立的、可部署的服务容器中。与传统的集成代理(可参见 8.5.4 节)相比，这是 ESB 的一个关键的不同之处。而传统的集成代理本质上通常是垂直缓的、高度集中的、具有整体性的[Papazoglou 2006]。在 ESB 方式中，在需要的地方可以选择集成代理的一些功能，而在不需要的地方，则可以不采用。

如图 8.6 所示，ESB 的主要用例之一就是充当门户服务器和后端数据源、处理源之间的中间层。门户服务器与要和后端数据源、处理源进行交互。门户是一个面向用户的、可视化的接入点，它包含了许多资源，这些资源可表示为服务，诸如零售、部门、公司员工和业务合作伙伴门户。通过门户可以统一地、安全地访问多个应用，自服务的发布、在线合作和流程自动化。门户从许多不同的系统中抽取内容，然后在浏览器的窗口中展示这些内容和相关的功能。例如，一个业务中的销售门户可以包含销售信息、特殊事件(例如销售折扣)的日程表、销售预测等。门户中的每一部分的内容都可以源自企业内的不同系统。门户中的有些内容也可以来自于外部组织。对于减少定制代码的接口的复杂性和开销，Web Service 提供了有效的方式，并可有效地管理各种变化。

图 8.6 显示了一个简化的 ESB 视图。该 ESB 集成了一个使用 JMS 的 J2EE 应用、一个使用了 C#客户端的 .NET 应用、一个和遗留系统交互的 MQ 应用，以及外部应用和使用 Web Service 的数据源。在一个 ESB 应用中，开发工具可将一个新的或已有的分布式应用暴露为 Web Service，并且能够通过门户访问这些 Web Service。ESB 中的资源通常表示为能够提供一个或多个业务操作的服务。类似 ERP 系统这样的集成打包应用也可使用诸如 J2EE 连接器体系结构(JCA)这样的技术创建服务，集成打包应用然后可暴露为 Web Service。

在 ESB 中，可以以面向服务的方式组织许多不同的应用组件，并应用 Web Service 即使，从而可以对这些应用组件进行更有效的增值集成。例如在图 8.6 中，分布式查询引擎通常基于 XQuery 和 SQL，通过对不同的业务数据源或组织信息库的统一访问，分布式查询引擎可以创建业务数据服务，例如销售订单数据或可用产品集。

在 ESB 中，端点提供了物理目的地的抽象和连接信息(如 TCP/IP 主机名和端口号)，如图

8.6 所示。此外，基于可靠的消息传输协议(参见 8.4 节)，端点有利于服务容器之间的异步和可靠通信的实现。通过端点，服务可以使用逻辑连接名进行通信。在运行时，ESB 可将逻辑连接名映射到实际的物理网络目的地。由于 ESB 中的服务具有目的地独立性，因此无须修改代码或中断已有的 ESB 应用，即可更新、迁移或取代这些服务。例如，无须中断其他应用，即可用一个新的开发票服务取代 ESB 中的一个原有的开发票服务。此外，可以准备一些一样的后备服务，以便当一些服务不可用时，可以进行故障接管。通过对端点进行配置，可以使用多个 QoS 等级，从而即使在网络出现故障的情况下，也可以保证通信[Chappell 2004]。

本质上，ESB 容器模型具有分布性，需要时可将单个的事件驱动的服务插入到 ESB 主干中。因此，服务可以是高度分散的，并可以高度分布的方式协同工作，并且它们可以彼此独立地进行伸缩。如图 8.6 所示，在不同平台上运行的应用可以抽象地相互解耦，并且可以作为逻辑端点通过总线进行连接。我们在第 7 章中已经分析了 WS-Notification 协议栈，该协议栈可将发布/订阅功能加到支持 ESB 的 Web Service 标准中。

为了成功地构建和部署分布式的 SOA，首先需要解决设计、部署和管理方面的 5 个问题：

(1) 服务分析和设计：应该使用服务开发方法学进行面向服务的开发，并复用已有的应用和资源。

(2) 服务使能：服务开发方法学应该决定哪些应用元素需要作为服务进行暴露。

(3) 服务编配：需要以一致的方式配置和编配分布式服务，并清晰地定义分布式处理。

(4) 服务部署：主要着眼于生产环境中的寻址、安全性、可靠性和可伸缩性议题。

(5) 服务管理：必须对服务进行审计、维护和重新配置，并且无须重写服务或底层的应用即可修改流程的变化。

在本章中，我们假定 ESB 中的服务按照服务开发方法学的原理进行了很好的设计，并可以通过服务管理框架进行合适的管理。

8.5.1 SOA 的事件驱动特性

为了充分发挥事件驱动的计算的潜力，SOA 除了服务还需要利用其他的基础技术。大多数 SOA 实现的最终的主要目标就是尽可能多地进行自动化处理，并且当用户需要与业务处理进行交互时，可以提供关键的、具有可操作性的信息。这就需要 ESB 基础架构本身能够识别有意义的事件，并可对它们进行合适的响应。响应既可以是自动地初始化新的服务和业务处理，也可以是通知用户感兴趣的业务事件、将事件放入主题上下文中，并通常建议最合适的动作。在企业上下文业务事件中，诸如客户订单、运送的货物到达装卸处、账单支付等都可能会和常规的业务处理交织在一起，并可能在任何事件点、以任意顺序出现。编配流程之间需要交换信息。使用编配流程的应用相互之间进行通信时，需要使用称为事件驱动的 SOA 的许多功能。

事件驱动的 SOA 是一个分布式计算的体系结构方式，在独立的软件组件之间可以异步地发送由事件所触发的消息。通过抽象，组件相互之间并不需要了解对方的任何信息，并将组件与底层的服务连接性和协议分离开来。事件驱动的 SOA 提供了一个比较轻量级的、更简易的技术，可构建并维护客户端应用的服务抽象[Bloomberg 2004]。

在一个启用 ESB 的事件驱动的 SOA 中，可将应用和服务看作抽象服务端点，这些抽象服务端点能够比较容易地响应异步事件[Chappell 2005a]。在启用 ESB 的事件驱动的 SOA 中，应用和事件驱动的服务以松耦合的方式捆在一起，它们彼此独立地进行操作，并可在大量的业务功能中得到应用。事件源通常通过 ESB 发送消息，ESB 将消息发布到订阅了该事件的对象中。事件本身封装了一个活动，并且是一个具体动作的完整描述。为了实现它的功能，ESB 必须既支持已有的 Web Service 技术，诸如 SOAP、WSDL 和 BPEL，又必须支持新兴的标准，如 WS-ReliableMessag-

ing 和 WS-Notification(参见第 7 章)。

为了使得解决方式更轻量级，事件驱动的 SOA 需要事件中的双方(服务器和客户端)是一种非耦合的关系。在完全解耦的交换中，事件中的两个参与者在进行业务事务之前并不需要彼此了解。这意味着并不需要一个 WSDL 服务契约，WSDL 服务契约详细说明了服务器对客户的行为。服务器和客户端之间的仅有的关系是一种通过 ESB 的间接关系，客户端和服务器分别作为事件的订阅者和发布者。在事件驱动的 SOA 中，尽管事件的双方是一种解耦关系，然而事件的接收者需要获取事件的元数据，在这样的情况下，事件的接收者仍然有一些有关事件的信息。例如，事件发布者通常基于一些(主题)分类来组织事件，或者提供事件的细节信息，包括它的大小、格式等，这些也是一种形式的元数据。然而，与服务接口相比，特定事件的元数据倾向于随同事件一起，而不是包含在一个独立的服务契约中。尤其，特定事件的元数据描述了所发布的事件，消费者可以订阅这些事件，服务客户端和提供者展示这些接口和它们所交换的信息，甚至商定的元数据的格式和上下文，而不会涉及正式的服务契约本身。

在分布式处理中，为了有效地编配服务的行为，ESB 基础架构包括一个分布式处理框架和基于 XML 的 Web Service。为了举例说明这些功能特性，图 8.7 显示了一个简化的分布式采购业务流程，以后将使用 ESB 配置和部署这个流程。库存系统首先发起一个补货信号，因而将触发一个自动的采购处理流程。在采购流程中，需要执行一系列的逻辑步骤。首先，采购服务查询企业的供应商参考数据库，从而确定一个可能的供应商列表。可以基于一些已有的合约和对供应商的衡量指标，对供应商列表中供应商的优先等级进行排序。然后，可基于一些标准选择供应商，在 ERP 订购系统中自动生成定购单，并将定购单送给所选择的供应商。最后，供应商通过开发票服务生成客户账单。

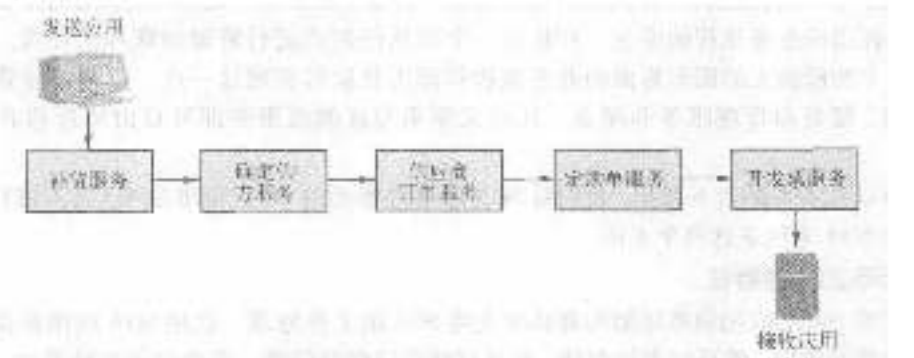


图 8.7　简化的分布式采购流程

图 8.7 描述了一个简化的分布式采购流程中的服务，这些服务的使用如图 8.8 所示。采购业务流程的服务实现复用了后端系统(诸如 ERP、供应链管理或者任何其他的企业应用)中的一些标准流程。为了创建采购业务流程所需的功能，可编配这些服务。在图 8.7 的例子中，我们假定库存已经缺货，并且补货消息已经发送到供应商订单服务。虽然该图仅显示了一个供应商订单服务(库存的一部分)，实际上可以存在多个供应商服务。为了履行订单，供应商订单服务选择执行所选择的供应商的 Web Service。假定供应商订单服务生成了一个 XML 格式的消息作为输出，但是定购单服务并不理解这一消息的格式。为了避免异构性的问题，供应商订单服务发出的消息利用了 ESB 的转换服务，将 XML 转换为定购单服务所能接收的格式。该图也显示了在 ESB 中可使用 JCA，通过 JCA 资源适配器可将遗留系统(诸如信用卡核查服务)集成到 ESB 中。

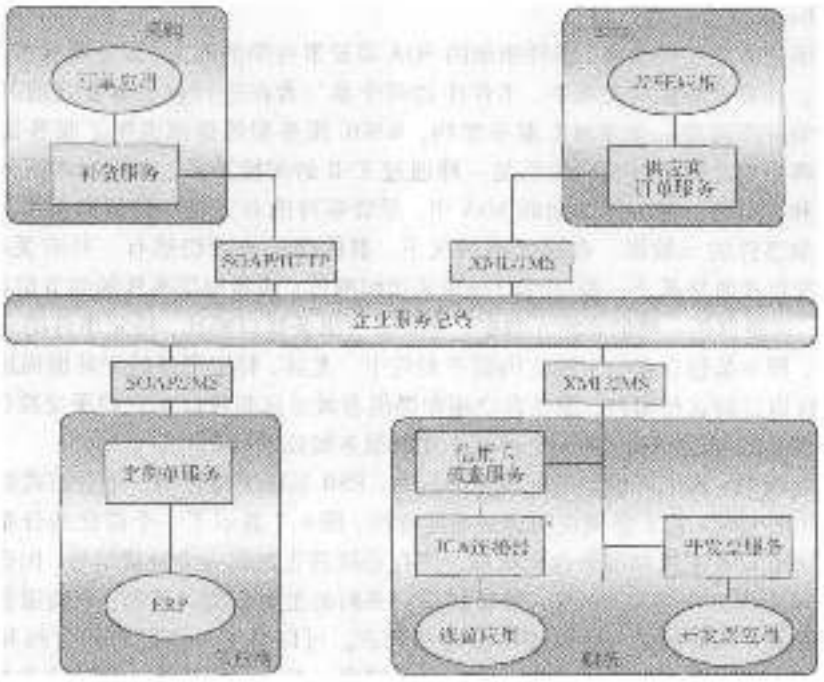


图 8.8 ESB 为流程服务

一旦将图 8.8 所描述的分布式采购业务流程中的服务链接在一起，则需要管理和重新配置这些服务，以便适应业务流程的变化。ESB 是一个可从任何点进行管理的联邦式环境。理想情况下，通过一个功能强大的图形界面的业务流程管理工具就可实现这一点。业务流程管理工具需要能够配置、部署和管理服务端点，从而无须重写或修改服务即可自由地迁移和重新配置服务。

在本章以及本书的余下部分，我们对 SOA 和事件驱动的 SOA（除非需要）将不进行区分，并将使用术语 SOA 来代表这两个术语。

8.5.2 ESB 的关键特征

为了实现 SOA，应用和基础架构都必须支持 SOA 的工作原理。启用 SOA 应用涉及创建已有功能或新功能的接口，既可以直接创建，也可以使用适配器创建。若要启用基础架构，在最基本的级别上涉及将安全的服务请求路由并发送到服务提供者的能力。然而，基础架构必须能够在不影响服务客户端的情况下替换已有的服务实现。这不仅需要按照 SOA 规范规定服务接口，而且基础架构需要允许客户端代码调用服务，并且无论服务的位置和所涉及的通信协议是什么。服务路由和服务替换是 ESB 诸多功能中的两个。下面的列表描述了详细的 ESB 关键功能需求，从中可参考其他的功能。其他作者诸如[Robinson 2004]、[Candadai 2004]、[Channabasavaiah 2003]、[Chappell 2004]，已经讨论了下面列表中描述的一些功能需求。注意，当前的商用 ESB 系统并不一定提供下面描述的所有能力：

动态连接能力：动态连接是指：对于每一个服务，无须使用一个单独的静态 API 或代理即可动态地连接到 Web Service。目前大多数企业应用都使用一个静态连接模式，对于每一个服务需要一些静态的代码。对于一个成功的 ESB 实现来说，动态的服务连接是一个关键的能力。动态连接 API 同样无须考虑服务实现协议（Web Service，JMS，EJB/RMI 等）。

可靠的消息传送能力：可靠的消息传送主要用于确保将这些消息发送到它们的目的地以及用于处理事件。对于以异步方式响应客户端以及对于成功的 ESB 实现来说，这种能力非常关键。

基于主题和基于内容的路由能力：ESB 不仅应具有基于主题的路由机制，而且应具有更复杂的基于内容的路由。基于内容的路由假定可将消息分组到固定的主题类中，以便订阅者能够说明他们所感兴趣的主题，从而能够收到与订阅的主题相关的消息。另一方面，基于内容的路由允许基于业务事件的实际特性（属性）的约束进行订阅（参见 2.5.3 节）。基于内容的路由基于服务的上下文或内容将消息转发到它们的目的地。为了确定需要将消息路由到 ESB 基础架构中的哪一个端点，通常使用能够检查消息的内容以及能够对消息的内容应用规则集的技术实现基于内容的路由。通常使用 XPath 或脚本语言（诸如 JavaScript）表示基于内容的路由逻辑（规则）。例如，假如一个制造商向它的客户提供了大量的商品，但是仅有一些是内部生产的。随着所订购的产品的不同，可能需要将消息直接路由到一个外部的供应商，或者路由到内部的仓储履行服务进行处理。诸如 WS-Notification 等新兴的标准也支持基于内容的 ESB，参见 7.3 节。

转换能力：ESB 可以通过多种传输协议路由服务交互，并可根据需要从一种协议转换为另一种协议，这是 ESB 的一个重要能力。ESB 实现的另一个重要方面是支持服务消息传输建模，并且数据格式与 SOA 接口一致。ESB 的主要价值源泉在于各个组件无须了解其他组件的实现细节。EBS 转换服务可以确保组件所收到的信息与数据的格式与它们所期望的一致，因此组件本身无须进行任何修改。在不同的数据格式和消息模型的转换中，无论是在基本的 XML 格式和 Web Service 消息之间的转换，还是在不同的 XML 格式之间的转换（例如将遵循行业标准的 XML 消息转换为一个专有的或定制的 XML 格式），ESB 都扮演了一个主要的角色。有关 ESB 的连接性和变换性基础架构的讨论，参见 8.5 节。

服务使能能力：服务使能包括访问已有资源（诸如遗留系统）的能力，以及在 SOA 实现中包含这些已有资源的能力。遗留系统是一个组织的基础架构的关键任务要素，但是在技术通常过时了。必须利用这些遗留资产，将它们服务化，并与现代服务技术和应用进行集成。8.5.6 节讨论了这一重要议题。

具有多种 QoS 的端点发现能力：ESB 必须支持基本的 SOA 需求，能够发现、定位和绑定服务。由于许多网络端点都能实现同一个服务契约，ESB 也必须支持客户能够在运行时选择最合适的端点，而不是在实现时硬编码端点。因此，ESB 需要能够支持不同的 QoS，并允许客户端发现 QoS 满足需求的最合适的服务实例。基于诸如 WS-PolicyFramework 这样的策略标准，与服务相关的声明策略将控制具有多种 QoS 的端点发现能力。

长期运行的流程和长事务的能力：面向服务与诸如 .Net 或 J2EE 这样的分布式对象体系结构完全不同，它更紧密地反映了真实世界的处理与关系。因此，SOA 能够以一种更自然的方式建模和构建软件，这些软件解决真实世界的业务处理需求。相应地，ESB 将支持业务流程和长期运行的服务。长期运行的服务是指一些长时间运行的服务，随着服务的运行，这些服务可能需要交换消息（会话）。在线预定系统就是一个典型的例子，它与用户以及不同的服务提供者（航空票务、保险索赔、抵押贷款和信用卡产品应用等）进行交互。此外，在业务环境中极其重要的一点是：ESB 必须提供一定的事务保证。更具体地，ESB 需要确保复杂事务能够以高可靠的方式进行处理，即使出现了故障，事务也能够回退到原先的状态。消息传送模式可使用异步、存储与转发和基于行程表的路由技术，假如 ESB 的实现基于这些消息传送模式，则可能实现长时间的事务会话。注意，ESB 分析师和供应商社区当前使用的 ESB 的基本定义并不要求一个长事务管理器[Chappell 2005b]。第 9 章和第 10 章讨论了业务流程、长时间运行的服务和事务。

安全性能力：通常来说，处理和实施安全性是 ESB 成功实现的关键因素之一。ESB 既需要向

服务消费者提供一个安全性模型，又需要和服务提供者的(各种各样的)安全性模型进行集成。ESB 既需要具有点到点的安全性能力(例如 SSL 加密)，又需要具有端到端的安全性能力。端到端的安全性能力包括：1)联邦认证，即拦截请求，并添加合适的用户名和凭据；2)验证和授权每一个服务请求，以便确保发送者有访问服务的合适的权利；3)在请求消息和响应消息的元素级加密/解密 XML 的内容。为了处理这些复杂的安全性需求，ESB 必须依赖 WS-Security 和 Web Service 的其他安全性标准，目前已经开发了一些这方面的标准，参见 11.4 节和 11.5 节。

集成能力：为了在异构环境中支持 SOA，ESB 需要集成许多原本不直接支持面向服务方式的交互的系统。这些可以包括遗留系统、打包的应用或者其他的 EAI 技术。当评估 ESB 的这些集成需求时，可以考虑几种不同类型或"风格"的集成，例如基于流程的集成、基于数据的集成。

管理和检测能力：在 SOA 环境中，跨系统(甚至组织)边界的应用会有部分重叠，并且可能会随着时间而变化。管理这些应用是一个很大的挑战。例子包括动态负载平衡、主要系统出故障后的故障接管、以及实现客户端和服务实例之间的拓扑或地理亲和性等。为了有效地管理 ESB 中的系统和应用，需要一个能一致地管理不断增长的、异构的组件系统的管理框架，并支持复杂的聚合(跨组件)管理用例，如动态资源供应和按需路由、SLA 实施和基于策略的行为(例如基于服务质量以及单个事务的业务值的比较，动态地选择服务提供者的能力)。对于一个成功的 ESB 实现来说，一个附加的需求是检测服务的状态、能力和性能的能力。监测是指跟踪通过总线发生的服务行为，并能可视化地显示度量结果和进行统计。其意义在于能够发现业务流程中的问题和异常，并且推动这些问题和异常的及时解决。

可伸缩性能力：随着分布式 SOA 的广泛应用，为了满足集成的需要，一些服务或整个基础架构需要能够具有可伸缩性。例如，转换服务通常需要耗费大量的资源，可能需要跨两个或多个计算节点的多个实例。同时，需要创建一个能够支持全球服务网络上的大量节点的基础架构。SOA 的松耦合特性需要 ESB 使用一个分散化的模型来提供一个具有很高性价比的解决方案，解决方案需要能够提高集成网络的各个方面的可伸缩性。分散化的体系结构使得单个的服务和通信基础架构本身都具有独立的可伸缩性。在 8.5.7 节中，将进一步讨论可伸缩性。

最后，表 8.1 总结了最典型的 ESB 功能化能力以及相关的标准。对于理解后面的内容，上面列表中的 ESB 集成能力是最重要的。当进行面向服务的集成时，集成能力也是 ESB 的一个关键要素。在下面的章节中，我们将更详细地讨论它们。这个讨论部分基于文献[Channabasavaiah 2003]中提出的观点。

表 8.1 ESB 功能域和相关的标准

功 能 域	能 力	相关的标准
连接性	传输	SOAP
	保证传送	WS-ReliableMessaging
	路由	WS-Addressing
基于内容的路由和事件通知	基于内容的路由	XPath
	基于主题的路由	WS-Topic
	业务事件通知	WS-Notification
转换	协议转换	XSLT
	消息转换	WS-Addressing
	数据转换	WS-ResourceFramework
服务使能	包装	WSDL
	转换	BPEL

（续）

功 能 域	能　力	相关的标准
	访问遗留系统	
服务编配	流程描述	BPEL
	流程执行	
	长时间运行的流程	
事务管理	事务性服务	WS-Transaction
	协调	WS-Coordination
安全性	认证	WS-Security
	授权	WS-SecurityPolicy
	访问权限	
	加密	
具有多个 QoS 的发现	运行时具有多个 QoS 能力和策略的服务发现	WS-PolicyFramework
管理和监测	监测 QoS	WS-DistributedManagement
	实施 SLA	WS-Policy
	控制任务	
	管理资源生命周期	

8.5.3　ESB 的集成类型

ESB 应用了一个面向服务的集成解决方案。基于其他的一些开放标准、松耦合以及 Web Service 的动态描述和发现，该解决方案减少了集成的复杂性、成本和风险。ESB 体系结构集成方式的其他显著特点是：它与特定技术无关（它涉及多个技术），并且在开发新的应用时可以复用已有应用中的功能。ESB 环境中的面向服务的集成解决方案需要解决一系列的重要的技术性需求，下面将进行简要描述。

在表示层进行集成

在表示层的集成主要涉及如何将完整的应用和服务集构成一个高度分布然而一致的门户框架，从而提供一个有用的、效率高的、统一的、一致的表示层。因此，ESB 可以向用户提供一个具有一致用户体验、统一的信息传送的界面，同时底层应用依然保持分布性。在门户空间新涌现了两个辅助性的行业标准，它们具有下列作用：

- JSR 168：这是一个产业标准，定义了开发 portlet（门户）的标准方法。它允许 portlet 互操作地跨多个门户供应商。例如，针对 BEA WebLogic Portal 开发的 portlet 可以与 IBM Portal 互操作。这使得组织可以降低对特定的门户产品供应商的依赖。
- WSRP（Web Service 的门户标准）：这是一个行业标准，允许以标准的方式开发和使用远程的 portlet，并促进了联邦门户的实现。WSRP 综合了 Web Service 和门户技术的能力，并很快成为了分布式企业门户的主要的使能技术。

JSR 168 针对本地门户而不是分布式门户，从而对 WSRP 进行了补充。一个门户页面可以有一些本地 portlet，这些 portlet 遵循 JSR 168，并且一些远程的、分布式的 portlet 在远程容器中执行。一个 portlet 可以是一个 ESB 服务，允许 portlet 开发者能够通过 ESB 以一种规范的方式察看整个后端系统，参见图 8.6。随着 JSR 168 和 WSRP 的成熟，一个真正的 EJB 联邦门户就将成为现实。

应用连接性

应用连接性是一种集成类型，它主要关于 ESB 集成层必须支持的各种类型的连接性。一方

面，这意味着诸如同步和异步通信、路由、转换、数据的高速分发、网关和协议的转换等事情。另一方面，应用连接性与输入和输出的可视化相关，与源和接收者相关。这需要接收输入，并以与源无关的中立方式传送到 ESB 中的应用。通过专用的前端设备和协议处理程序将可实现这一点。ESB 可以利用 J2EE 的一些组件来提供连接性，诸如 Java Message Service for MOM 的连接性、连接到应用适配器的 JCA 等。ESB 也可以很容易地与 .NET、COM、C#构建的应用集成。此外，ESB 也能够很容易地与支持 SOAP 和 Web Service 的任何应用集成。

通过通知部件，诸如那些嵌入在 WS-Notification（我们已在 7.4 节中讨论的）中的部件，ESB 可将这些不同类型的应用连接在一起。例如，JMS 应用可以发布一个通知消息，WS-Notification 定义的通知消费者可以接收这些消息。

通过管理注册库中的主题空间集，以及实现一个或多个 WS-Notification 定义的分布式通知代理，ESB 可以协调请求/响应 SOA 服务与面向消息的 SOA 服务。传统的 MOM 应用无须关注这些代理的 WS-Notification 方面，它们仅在所选择的代理上发布或订阅主题。ESB 管理者可以配置一个或多个 WS-Notification 类型的通知代理，以便 WS-Notification 发布者或订阅者能够使用。此外，ESB 管理者可以将通知代理相互之间结成联邦或与传统的订阅/发布应用结成联邦。

应用连接性处理下列类型的集成：

- 应用集成：应用集成主要关于集成主干的构建和演化。基于集成主干可快速地装配和分解不同的平台和组件技术。装配流程可组合遗留应用、已获取的包、外部应用订阅和新构建的组件。应用集成是装配流程的一个有机组成部分。ESB 主要致力于基于服务的应用集成，启用结构化更好的集成解决方案。这些集成方案将交付由可更换的部件组成的应用。之所以采用可更换的部件，主要是为了适应业务和技术的变化。
- 业务流程集成：流程集成主要关于自动化流程的开发，提供一个业务流程解决方案，将已有的应用集成到流程中，并将流程与其他的流程相集成。ESB 的流程级集成可以包括业务流程与企业中的应用（例如 EAI 解决方案）的集成。它不仅是对单个服务的集成，也可以涉及包含外部源的整个流程的集成，诸如供应链管理或财务服务，这些可能横跨多个组织。对于这类应用和流程集成需求，可以使用诸如 BPEL 等技术，参见 9.7 节。
- 业务数据集成：数据集成对企业中的业务数据提供一致的访问，从而使得应用访问数据时，不会受到数据格式、数据源或者数据所处的位置的制约。这一需求在实现时可能需要合并和一致化不同的数据（例如合并两个顾客的信息）、验证数据的一致性（例如最低收入应不低于一定的阈值）等，因此将涉及适配器和转换工具、聚合服务。无论这些数据原有的格式是什么，无论管理这些数据的操作系统是什么，也无论在哪里存储这些数据，都可以对业务数据进行管理。分布式查询引擎（诸如图 8.6 所示的查询引擎）可提供对分布式业务数据的访问。
- 集成设计和开发方法学：应用开发环境的需求之一是必须考虑企业里能被实现的集成的所有类型和级别，并将它们用于开发和部署。为了真正地实现强壮性，开发环境必须提供一个有效的方法学。方法学需要能够清晰地描述如何设计和构建 ESB 服务和组件，以便可以复用、减少冗余，并简化集成、测试、部署和维护。

在任何一个具体的企业中，上面列表中所有集成方式都将有一些形式的变体。在有些情况下，这些集成方式可能进行了简化或没有清晰地定义。需要高度重视的一点是，在着手 ESB 的实现时，必须考虑仔细权衡和比较各种集成方式。

8.5.4 ESB 解决方案中的各要素

实现 ESB 可有一些不同的方式。ESB 本身既可以是一个集中化的服务，也可以是一个包含

同类 ESB 或子类 ESB 的分布式系统，即一种联邦 ESB 的形式。它们汇聚起来工作，从而保证 SOA 系统能够正常运营。

在小规模的集成解决方案实现中，ESB 的物理基础架构的拓扑结构很可能是集中化的。集中化的 ESB 拓扑以服务器的一个集群或集线器为中心。这个解决方案类似于集中星型的中间件拓扑，其中心节点可以管理应用间的交互，并可避免一个应用不得不和其他几个应用集成多次的情况发生[Linthicum 2003]、[Papazoglou 2006]。集中星型在中心节点进行集成处理。中心节点对一致事情可进行集中控制，如集成/转换活动、维护路由信息、服务名等。在跨企业的系统中，最流行的集中星型 EAI 解决方案是集成代理(参见 2.7.1 节和 8.5.4.1 节)。

即使可将集中星型解决方案进行扩展，使其可以跨越组织边界，但仍然无法实现局部自治，即各个业务单元无法以完全独立的方式运作。导致这一制约的原因主要在于集成代理无法很容易地跨越防火墙和网络域。然而，如前所述，对于大规模的实现，集中星型方案会很容易出现瓶颈，这是这一方式的最大缺点。在松耦合的环境中，单个单元并不理解局部应用间的业务流程，或者诸如集成代理这样集中式地管理变化域。

当各个单元在组织上或地理上比较分散并且相互之间需要独立运作的情况下，组织架构可以在物理上变得更分布，同时在逻辑上依然保持对配置的集中控制。这称为联邦式集线器解决方案。如图 8.9 所示，联邦式 ESB 允许不同的企业(诸如制造商、供应商和客户)将它们的集成域汇聚到一个更大的联邦集成网络。这一拓扑可本地化地安装、配置、安全控制和管理本地消息通信、集成组件和适配器。在图 8.9 中，可使用联邦 ESB 解决方案来形成一个跨行业的、虚拟化的交易伙伴网络，并且服务可以更大范围地采用各种选择和合作伙伴模型。

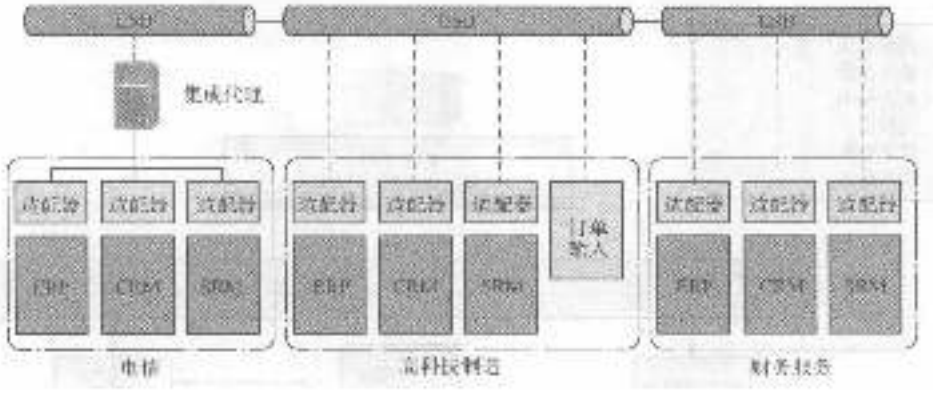


图 8.9　分布式的 ESB 允许物理上联邦的组织可以相互合并

ESB 的物理部署依赖于将采用的 ESB 技术，诸如专用的 MOM、集成代理、应用服务器等。不同的 ESB 技术的使用和组合将导致各种不同的 ESB 模式，在和物理部署连接时，每一种模式都有它自身的需求的约束。有些 ESB 配置可能适合于大范围的分布，从而可以支持比较大的地理范围上的集成。而有些 ESB 配置则可能适合于在局部化的集群上进行部署，从而可以支持高可用性和伸缩性。在物理分布的需求与各种可用技术的能力进行匹配是 ESB 设计的一个重要方面。根据具体需求的变化，可增量式地扩展最初的部署、集成新增的系统或者扩展 ESB 基础架构的物理覆盖范围等，这也是一个重要的能力[Robinson 2004]。

无论 ESB 的实现拓扑是什么，ESB 的主要目的都是提供企业资源的虚拟化，以便可以开发企业的业务逻辑，并可独立地管理基础架构、网络和服务。若要实现 ESB，需要支持下列与连体系

结构样式的中间件工具集 [Endrei 2004]：

- 面向服务的体系结构：在面向服务的体系结构中，分布式应用由粒度化的可复用的服务组成，这些服务具有良定义的、公开发布的、符合标准的接口。
- 消息驱动的体系结构：在消息驱动的体系结构中，应用通过 ESB 将消息发送到接收消息的应用中。
- 事件驱动的体系结构：在消息驱动的体系结构中，应用能此独立地生成消息、使用消息。

正如在前面章节中所讨论的，ESB 支持上面的体系结构样式和服务集成能力，并提供集成的通信、消息传送和事件基础架构。为了实现它所声称的目标，ESB 在集成化的基础架构中结合了应用服务器、集成代理、业务流程管理技术和产品的功能。在下面的章节中，我们将要依次讨论中间件解决方案。

1. 集成代理

集成代理作为分布式基础架构的一部分，可用于集成不同的业务应用，下面我们主要讨论它的功能特性。在 2.7.1 节中，我们曾简要介绍了集成代理。

对于集成代理的实现，图 8.10 表示了一个典型的体系结构的高层视图。尤其，该图表示了集成代理如何集成许多后端的企业信息系统中的功能和信息。为了有效地说明集成代理的行为，我们使用图 8.7 所示的简化的分布式采购流程。如图 8.10 所示，当一个自动库存系统触发了一个补货信号后，将触发一个自动采购流程，并首先查询企业的供应商参考数据库，从而确定一个可能的供应商列表。在确定供应商列表时，可根据已有的合约和供应商度量指标对供应商进行排序。从供应商列表中选择一个供应商后，可在 ERP 采购模块中自动生成定购单，并将其发送给所选的供应商。

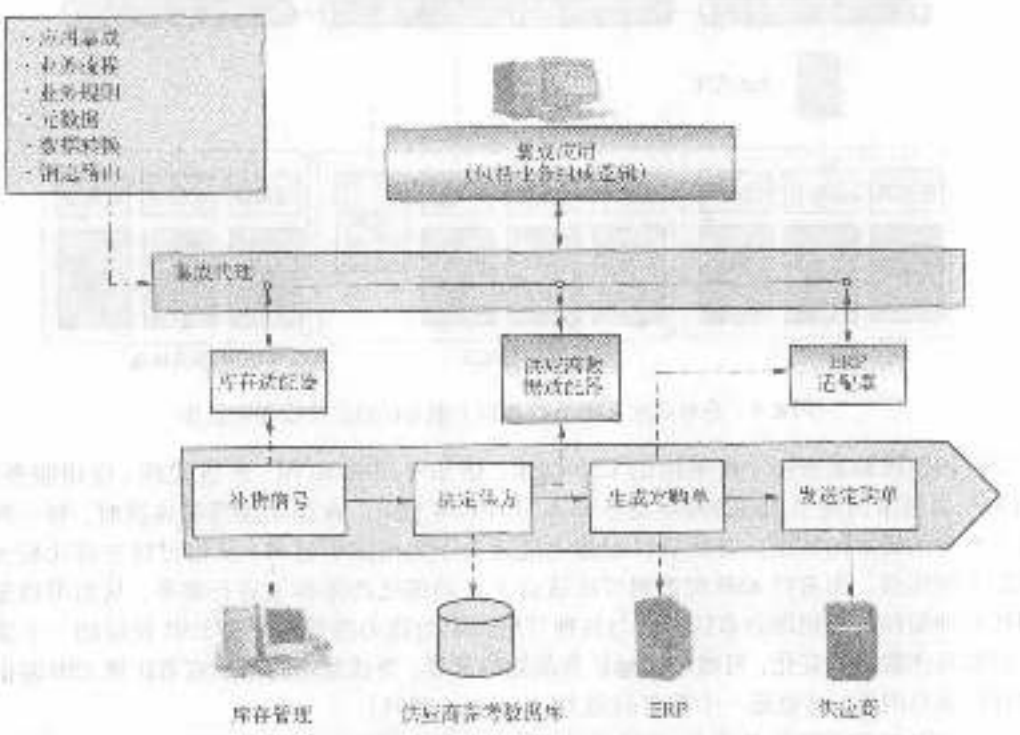


图 8.10 集成代理集成不同的后端系统

图 8.10 表示了集成代理是系统的核心。如图中的实线所示，集成代理使得两个或多个源（源应用和目标应用）之间的信息移动更容易实现，并解决了应用语义和异构平台之间的不同。在这个配置中，通过源适配器可将原有的一些不同的企业信息系统（诸如 CRM、ERP、事务处理监视器、遗留系统等）连接到集成代理上，如图中的虚线所示。在一个松耦合的配置中，可使用图中的源适配器访问一个具体的企业信息系统，并可以实现无损的应用集成。

集成代理体系结构具有几大优点。对于许多集成场景，集成代理提供了预建的功能，从而减少了应用集成的工作量。该价值定位基于跨多个应用的复用（根据中间件基础架构和应用集成逻辑）。现代的集成代理包含集成功能，诸如转换工具、流程集成、业务流程管理和交易伙伴管理、打包的适配器，以及通过诸如 JSP 等前端工具调用的用户驱动的应用。

在 ESB 中，分布式体系结构允许集成代理的功能选择性的部署和各个组件的独立的可伸缩性，而集成代理的功能（诸如消息传送和连接性、应用适配器、数据转换引擎、基于业务规则的消息路由等）则分散在这高度分布的体系结构中。这是现代的集成代理与传统的集成代理的一个重要的不同之处。传统的集成代理的功能通常局部于一个中心服务器上。

在许多具体的场合下，需要将新开发的 ESB 解决方案与已有的集成代理进行桥接。在这种情况中，原先安装的集成代理则变成了有价值的组件，ESB 可以使用它开发新的应用。若要更好地理解这些议题，以及集成代理和 ESB 方式之间的不同之处，可参见 8.5.7 节。

2. 应用服务器

连接 ESB 的另一个重要的中间件基础架构是应用服务器。对于分布式的基于 Web（以及非基于 Web）的应用和服务，应用服务器提供了一个集成开发环境，可用于开发和部署这些应用和服务。通过扩展已有的解决方案以及在 Web 中添加事务处理机制，应用服务器可提供 Web 连接性。由于应用服务器提供了一个基于 Web 的、事务性的、安全的、分布的和可伸缩的企业应用的开发、部署和管理平台，因此应用服务器非常适合用于应用集成。应用服务器可以处理业务流程、事务，并可通过单一的接口（通常是 Web 浏览器）访问后端业务数据和应用。这使得应用服务器非常适合于基于门户的 ESB 的开发。与集成代理不同，应用服务器并不直接集成后端系统，而是充当了一个用于集成企业业务流程的集成开发和支持框架。所有的应用服务器都期望集成代理充当一个服务提供者，提供数据访问、转换和基于内容的路由。

图 8.11 显示了应用服务器在大规模应用中的使用，该应用将 ERP 系统与复杂的客户接口连接在一起，从而展现了新的销售和分销模型。在图 8.11 中，适配器/组件包装器模块提供了应用服务器和组件 EIS 之间的抽象。这一层处理了 EIS 组件通信，就好像在应用服务器中执行组件 EIS 一样。在该类体系结构中，执行发生在应用服务器的组件包装器中。在图中，通过对遗留系统和应用以及其他的后端资源（诸如数据库、ERP、CRM 和 SPM）的包装，组件包装器实现了对组件系统的集成，从而能够以应用服务器所希望的标准的内部格式来表示数据和消息。执行这些真正的处理活动的其实是外部已有的 EIS，而应用服务器无须关心这一点[Lubinsky 2001]。

如前所述，我们需要使用包装器封装遗留系统和打包功能。一般而言，除了在 EAI 领域之外，对于包装器的功能并没有非常正确的认识。此外，正如在 Web Service 文献中可以看到的，对于包装器和适配器的概念似乎有许多混淆不清的地方，因此澄清它们的含义和目的还是很有必要的。

（组件）包装器仅是一个抽象的组件，提供了遗留软件所实现的服务，并用于隐藏已有系统的依赖关系。总而言之，包装器完成具体的业务功能，它有一个清晰定义的 API。对于客户端而言，可以将包装器视为一个“标准的”组件，并可以使用目前的一些协议访问它们。包装器通常部署在现代的、分布式环境中，诸如 J2EE 或 .NET。包装器一方面提供了一个标准的接口，另一

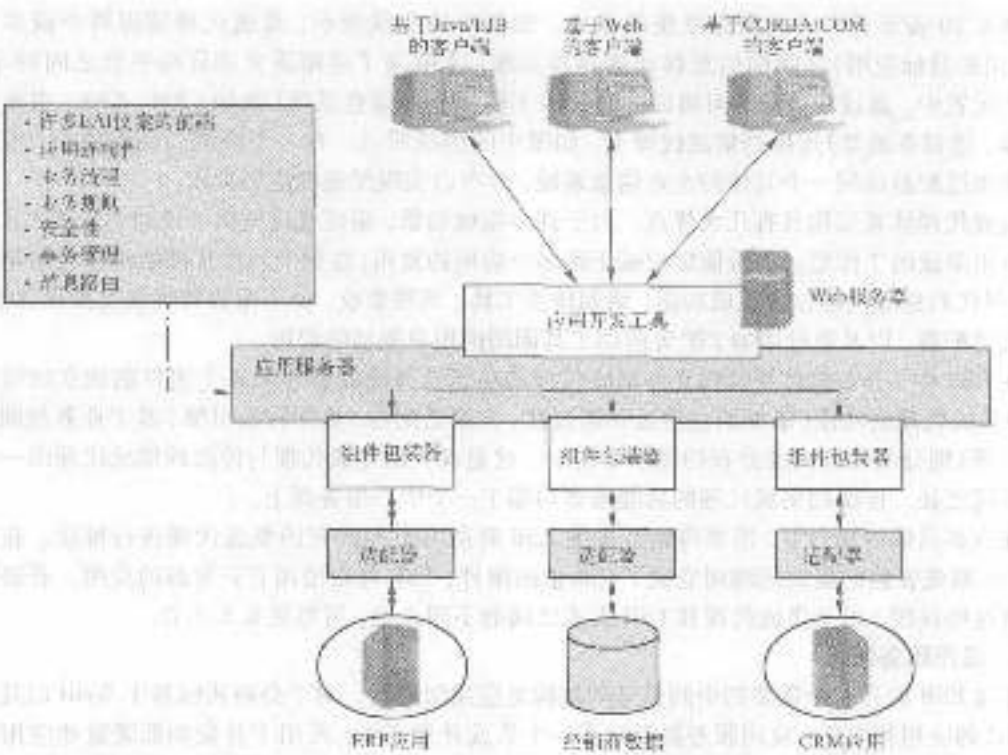


图 8.11 应用服务器提供了对后端系统的访问

方面与已有的应用代码进行交互。包装器可将已有的应用功能与其他所需的服务功能进行组合，并以虚拟组件的形式进行表示。ESB 中的任何其他服务都可通过一个标准的服务接口访问这些虚拟组件。

当包装遗留业务流程时，其实现包含访问适配器的代码，而适配器可访问遗留系统，如图 8.11 所示。与包装器不同，适配器并不包含表示逻辑功能或业务逻辑功能。适配器是一个介于两个软件系统之间的软件模块，它对这两个软件系统之间的技术和编程表示的差异以及变化感知的差异进行转换。资源适配器将应用消息按通用标准集进行来回转换。通用标准集指的是标准的数据格式和标准的通信协议，参见 2.7.1 节。

应用服务器主要基于 J2EE，并支持 JMS、Java 2 连接器体系结构和 Web Service。在下文中，我们将描述这些技术如何帮助实现 ESB 环境中的应用服务器。

正如 2.7 节所述，JMS 是一个与厂商无关的用于企业消息传送的传输层 API，许多不同的 MOM 厂商都将使用 JMS。JMS 框架以异步方式运行，并可模拟一个同步请求/响应模式[Monson-Haefel 2001]。对于应用服务器的实现，JMS 可访问分布在异构系统中的业务逻辑。基于消息的接口可实现点到点和发布/订阅机制，可保证信息传送以及异构平台间的互操作性。

在 ESB 环境中，可使用 JCA 技术来解决应用集成这一难题。在 J2EE 应用体系结构中，JCA 提供了一个集成不同应用的标准化方法。基于 JCA 所定义的功能，应用服务器能连接到后端 EIS，诸如 ERP、CRM，以及遗留系统和应用。JCA 提供了资源适配功能，可将 J2EE 安全性、事务和通信范映射到对应的 EIS 技术。当在 ESB 实现中使用 JCA 时，ESB 能够提供一个 JCA 容器。在 JCA 容器中，通过 JCA 资源适配器，可将打包应用或遗留应用连接到 ESB。例如，处理订单的服务可使用 JCA 与履行订单的 J2EE 应用通信。

正如集成代理解决方案一样，集成应用的应用服务器模型的特性也是集中式的。对于大规模的集成方案，应用服务器这一中心点很可能很快就成为瓶颈，并导致严重的性能问题。集成应用的应用服务器模型通常基于开发集成代码。该模型和 ESB 集成模型的最大的不同点在于 ESB 主要是关于配置而不是编码。然而，在企业体系结构中，应用服务器仍占据着重要的一席之地。应用服务器最适用的场合同时也是其最初的目的，即为驻留 EJB 形式的业务逻辑提供一个组件模型，并提供企业门户环境中的 Web 页面。使用已有的协议，诸如 JMS 和 MessageDrivernBeans 等，可将应用服务器连接到 ESB 中。

3. 业务流程管理

当前，企业正努力采用电子化手段连接他们的客户、供应商和合作伙伴。为了实现这一点，需要跨应用边界集成大量的原先不相关联的业务流程。应用边界既可以仅是涉及两个应用的简单的客户订单查询，也可能是一些复杂的长事务，例如涉及许多应用和人机交互的保险索赔处理，又如用于高级规划、生产和货物运送的并行业务事件，以及涉及许多应用、人机交互、业务间的交互的供应链。当在如此规模进行集成时，企业需要更强大的能力来克服由于各种原因所导致的一些挑战，诸如已有的专有接口、不同的标准、技术实现的方式、数据、业务自动化处理、流程分析和可视化级别。通过业务流程管理(BPM)技术可解决这类挑战。在本节中，我们将简要地概述 ESB 实现中的 BPM 功能。

BPM 提供了端到端的管理，并可控制长期的、多步骤的信息请求或事务处理，并且这些事务处理可以横跨一个或多个企业中的多个应用和用户。BPM 既可实时监控单个流程实例的状态，也可实时监控聚集在一起的所有实例的状态，并可将实际的流程活动转化为关键的性能指标。

集成供应链和内部企业功能并且无须开发更多的定制软件，这是许多企业的共同愿望。BPM 正是主要由这一共同愿望所驱动的。这意味着工具必须适合于业务分析，需要比较少的(甚至不需要)软件开发。因为内部和外部集成环境通常需要添加一些组件，并可随着业务流程变化，因此 BPM 减少了维护需求。有关 BPM 技术的更多信息，读者可参考 9.3 节。

诸如 BPM 软件解决方案等专用方案提供了与工作流相关的业务处理、流程分析和在 ESB 设置中的可视化技术。尤其，BPM 允许业务流程与底层集成代码进行分离。在 ESB 中，当调用复杂的流程定义时，流程编配引擎可以位于 ESB 的上层。流程编配引擎支持 BPEL 或其他的一些流程定义语言，诸如 ebXML 业务流程规范模式(BPSS)。流程编配可以支持长期运行的有状态的流程。它也可以支持并行执行路径，并可基于联接条件和变迁条件将消息流执行路径进行分支或合并。将复杂的流程编配与无状态的基于路线计划的路由结合起来可创建一个 SOA，从而解决复杂的集成问题。使用基于路线计划的路由的 ESB 提供了一个含有路由指令列表的消息。在 ESB 中，路由指令表示了一个业务流程定义，携带路由指令的消息通过总线跨越服务调用进行传送。远程的 ESB 服务容器确定了将消息紧接着发送到哪里。

4. ESB 传输层的选择

最后，在结束本节之前，我们还将讨论 ESB 中的传输层协议的选择。在 ESB 中，Web Service 能够使用可基于多种协议的 SOAP 消息进行通信。每一种协议都有效地提供了一个可连接多个端点的服务总线。当前，最通用的服务总线传输层实现包括 SOAP/HTTP(S)和 SOAP/JMS[Keen 2004]。

在服务请求者和提供者之间，基于 HTTP 的 SOAP 服务总线是发送请求和响应的最常见的方式。正如 2.1.1 节所阐述的，HTTP 是一个客户/服务器模式，HTTP 客户端打开一个连接，并将请求消息发送到 HTTP 服务器。客户端请求消息将调用一个 Web Service。HTTP 服务器发送包含调用的响应消息，并关闭该连接。通过使用 ESB，服务请求者可以使用 HTTP 进行通信，并使得

服务提供者可以使用不同的传输方式接收请求。许多 ESB 实现提供者都提供了 HTTP 服务总线以及其他的协议。ESB 交互中可以选择这些协议中的任何一种，并可基于实际服务等级的需求进行具体的选择。

JMS 是 J2EE 标准之一，它是创建、发送和接收企业消息的一种常规方式。虽然 JMS 并不没有完整地提供 HTTP ESB 所具有的互操作性，但是 SOAP/JMS ESB 在 QoS 方面仍具有一些优点。对于 Web Service 调用，SOAP/JMS ESB 可提供异步和可靠的消息传送。这意味着请求者能够收到保证传送的确认，并可与一些目前无法进行立即响应的企业应用进行通信。SOAP/JMS Web Service 实现了基于 JMS 队列的传输。正如 SOAP/HTTP 一样，SOA/JMS 服务总线使得服务请求者和提供者可以使用不同的协议进行通信。

8.5.5 连接和转换基础架构

一般来说，企业中的业务应用并没有重点考虑如何与其他应用通信。因此，在内部系统和外部交易合作伙伴的系统之间存在技术的不匹配。为了无缝地集成这些不同的应用，必须有一种方式可以很容易地将消息请求的格式转换为被调用的服务所期望的格式。例如在图 8.6 中，需要将 J2EE 的功能暴露给非 J2EE 的客户端，诸如 .NET 应用程序和其他的客户端。在这种情况下，Web Service 可以与组织中的其他 EIS 实例进行集成，或者 J2EE 应用本身与其他 EIS 进行集成。在这类场景中，应用如何与 ESB 交换信息依赖于应用的可访问性选项。应用有三种可选方式与 ESB 交换信息[Keen 2004]：

- 应用程序提供的 Web Service 接口：一些应用和遗留的应用服务器已经采用了开放标准并已经包括了 Web Service 接口。WSDL 定义了可与应用业务逻辑直接通信的接口。如有可能的话，采用直接通信通常是更可取的方式。
- 非 Web Service 接口：应用并不通过 Web Service 暴露业务逻辑。具体的应用适配器可在应用 API 和 ESB 中作为一个基本的中介体。
- 服务包装器作为对适配器的接口：在一些情况下，适配器可能无法提供 ESB 所期望的合适的协议(如 JMS)。在这种情况下，这种适配器需要能启用 Web Service。

作为 ESB 实现中的补充技术，(资源)适配器和 Web Service 可以协同工作，从而实现复杂的集成方案，如图 8.12 所示。数据同步(除了转换服务)是资源适配器的主要目标之一。因此适配器能够承担数据同步和转换服务的角色，Web Service 将能够启用应用功能彼此进行交互。随处都可访问的应用功能(服务)可能需要爱与其他的应用集成，从而实现它的服务契约。对于实现这样的应用功能，Web Service 是一个理想的方法。Web Service 通常由用户请求/事件发起，而数据同步则通常由数据对象(例如，顾客、具体的货物、订单等)中的状态变化发起。下面将进行具体说明。

Web Service 所响应的事件可以是一个用户发起的请求，诸如一个定购单或者一个在线账单支付。订单管理应用需要从会计系统中核查客户状态，这类应用可以很自然地生成用户事件。另一方面，数据对象中的状态变化可以是一个活动，诸如在客户服务应用中添加一个新的顾客记录，或者更新更新的账单地址。对于那些具有这些客户数据副本的其他应用而言，这些状态变化将触发适配器在那些应用中添加或更新客户的记录。

在 J2EE 到 .NET 应用的连接场合中，需要资源适配器形式的连接服务。在这个实现策略中，Web Service 能够变成公司和它的客户、合作伙伴、供应商之间的接口，而资源适配器则变成将公司中不同的 EIS 连接在一起的集成组件。这是一个可行的实现模式，其中 Web Service 和资源适配器能够共存。Web Service 和资源适配器之间需要协作的另一个可行的集成模式是业务流程集成。使用业务流程的应用将需要暴露它们的功能。显然，Web Service 非常适合这一目标。为了实现业务流程，应用有时需要与其他的 EIS 进行集成，这时将要使用到资源适配器。

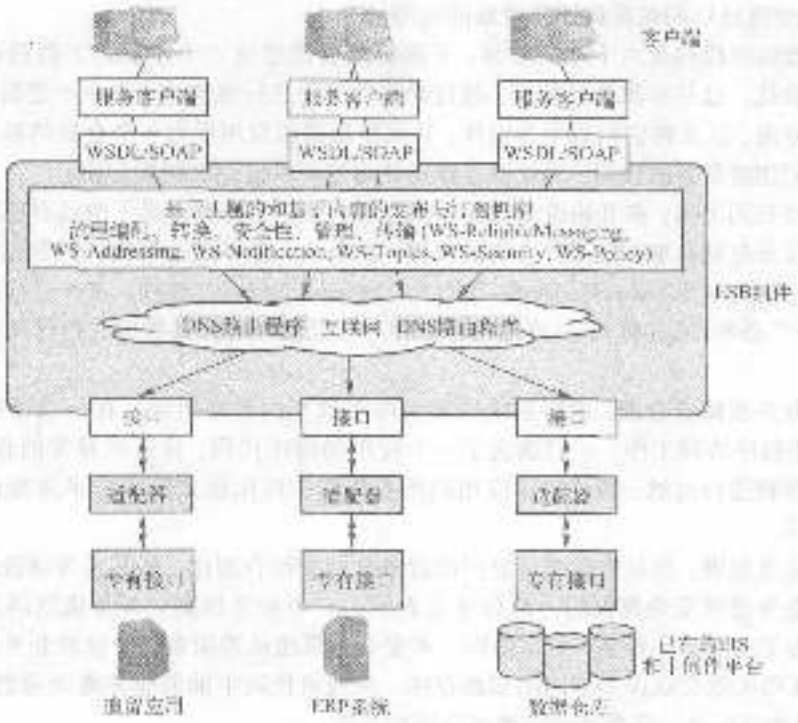


图 8.12　组合 Web Service 和资源适配器

8.5.6　遗留资产的使用

为了利用已有应用的功能以及在新的应用中复用已有的应用，需要对 ESB 进行设置。企业仍然可能运营一些较老的应用，运行这些应用硬件平台可能已经过时，或者这些应用是使用过时的编程语言编写的。这类应用通常称为遗留应用[Ulrich 2002]。

由于遗留系统包含了一些关键任务的业务信息和功能，并控制了企业的大多数业务流程，因此遗留应用是任何现代企业的重要资产。遗留应用能够实现核心业务任务，诸如生成和处理订单、开始生产和支付、开发票、信用卡支付、分销、库存管理、相关的税收、成本节约和会计任务。在新的 ESB 解决方案中利用已有的遗留资产，将充分挖掘已有投资的价值，并可获得非常大的回报。因此最佳的 ESB 特性就是提供对遗留系统的连接。

若要将遗留系统有效地集成到 Web Service 解决方案中，通常需要对遗留系统进行大量的、伤筋动骨的修改。为了使得遗留系统能够适应 Web Service 体系结构的需求，以及为了在新的应用中复用遗留系统的业务逻辑，不得不对遗留系统进行修改。因此，为了复用遗留应用中的核心业务流程，需要对遗留系统进行再工程(re-engineer)遗留系统的再工程指的是尽可能复用遗留系统，并在遗留系统中添加一些新的功能，从而将已有的遗留系统演化为新的"改进的"系统。通过再工程，业务流程将变得比较模块化、粒度化，可以复用子模块并将它们表示为服务。

遗留系统再工程和转换主要涉及企业、业务流程、EAI，以及就要将遗留系统融入体系结构中，同时又要尽量避免老系统低层和开发方法的缺点。再工程处理的基本形式有下列三个方面：

- 了解已有的应用，得出应用的一个或多个逻辑描述。
- 将那些逻辑描述重组或转换为新的，改进的逻辑描述。

- 基于这些改进后的逻辑描述开发新的应用。

这三个大致的阶段包含六个系列步骤，下面将简要描述这六个步骤。下面的再工程和转换步骤已经相当简化。这些步骤的目的是：通过对遗留流程进行模块化、将业务逻辑、表示逻辑和数据管理进行分离，以及将它们表示为组件，从而使得遗留应用成为一个全新的系统。然后可以使用这些组件创建新服务的接口，因此服务技术使得遗留系统依然得到应用。

（1）了解已有的应用：在开始改造遗留应用之前，第一个任务就是了解已有应用的结构和体系结构。这个任务包括获取有关数据，诸如规模、复杂性、废弃或无用代码的数量、每一个应用中有缺陷的程序的数量等[Comella-Dorda 2000]、[Seacord 2001]。此外，在改造计划的各个阶段（包括复用的再工程和/或迁移），选择改进哪些程序以及选择对公共数据进行作用的程序是一个关键步骤。

（2）使得业务逻辑更合理：遗留系统通常由许多独立的程序组成。在一个硬连线的业务流程网络中，这些程序协同工作。一旦清洗了一个应用的程序代码，且任何异常的程序都已删除，并已将非业务逻辑进行过滤，则可以在应用的所有程序中应用模式匹配技术来标识和隔离候选的通用业务逻辑。

（3）标识业务规则：当对候选的可复用的业务规则进行合理化，使其成为清洗的服务后，将可确定将这些业务逻辑变为流程的一部分还是表示为一个业务规则。业务规则的定义和例子可参见9.2节。为了实现这一任务，可以使用一些复杂的算法从遗留系统中抽取业务规则。在遗留系统中，业务规则可能会以许多不同的形态存在。从遗留代码中抽取业务规则通常称为“业务规则发现”。精确地完成这一任务是改造遗留应用的关键。

（4）抽取组件：抽取出来的业务规则可以根据它们的作用进行分组，一组规则通常处理一些通用的数据集，从而实现预定的业务功能。然后抽取候选业务规则和相关的业务数据，并将它们适当地表示为一个遗留组件。归在一类中的规则的数量通常依据遗留组件的粒度大小进行选择。对于这些组件，需要提供一个可调用的接口。

（5）包装组件的实现：一般而言，遗留系统一般并不是以组件化的方式实现的，并且表示逻辑经常和业务逻辑交织在一起，而业务逻辑又和系统、数据访问逻辑缠绕在一起。在本步骤中，将标识出系统层遗留组件和表示层遗留组件，并将它们与业务层业务组件进行分离。从而将候选组件标识出来进行包装。在新的面向服务的解决方案中，将遗留功能进行包装所需的代价将远远小于重新开发这些功能。包装需要确定组件的合适的抽象级别。在进行包装以及对遗留系统组件化时，主要关注粗颗粒组件的标识。因为大的组件通常价值比较大，所以复用比较大的组件通常性价比比较高。小型组件可能会经常用到，但是它们的价值通常也比较小。这意味着包装细粒度的组件通常性价比比较低。

（6）生成服务接口：组件包装器会生成良定义的功能和数据边界。然而从总体上说，改造后的遗留系统与组件间依然是紧耦合，通过程序间的调用彼此硬连线。对于大规模系统耦合，SOA方法需要去除其中各个组件的包装器，并且组件间必须无须了解对方。为了实现这一点，可用服务使能API代替程序间的连接，服务使能API可与事件驱动、业务流程编配机制一同使用。

对于再工程和转换步骤的更详细的描述，感兴趣的读者可参阅[Comella-Dorda 2000]、[Seacord 2001]、[Ulrich 2002]和[Papazoglou 2006]。

8.5.7 ESB中的可伸缩性

对于自动化业务集成解决方案，可伸缩性是一个特别重要的议题。就ESB转换来说，可伸缩性主要关于特定的ESB实现的设计。例如，当没有集中式的规则引擎处理流程中的每一步时，业务操作的并行执行和基于传送的路由对于实现ESB的分布性将起到很大的作用。基于异步通信、

消息传送、消息和流程的定义。ESB 的各个部分可以相互独立地运营。这将产生一个分散式的模型，从而提供了集成网络各个方面的可伸缩性。这类分散式的体系结构使得各个服务以及通信基础架构本身都具有独立的可伸缩性。

ESB 方式与集成代理方式截然不同。集成代理技术使用集中型模型处理可伸缩性，例如为了处理负载和配置的变化，通过提高代理的性能或者在中央位置上添加代理的方式来进行应对。若采用集中化的规则引擎（诸如典型的集中型项 EAI 代理方式）处理路由和消息，则该引擎很快就有可能成为瓶颈，并且也是一个单点故障点。当从多个企业集成服务时，集成代理消息所使用的策略如图 8.13 所示。如图所示，集成代理构建在应用服务器之上。若需要集成代理功能的任何地方，都需要安装完整的集成代理/应用服务器，这意味着需要在运行服务器上安装应用服务器，以便可以驻留 JCA 容器/适配器对。JCA 容器/适配器对充当访问 ERP 系统的信道[Chappell 2004]。

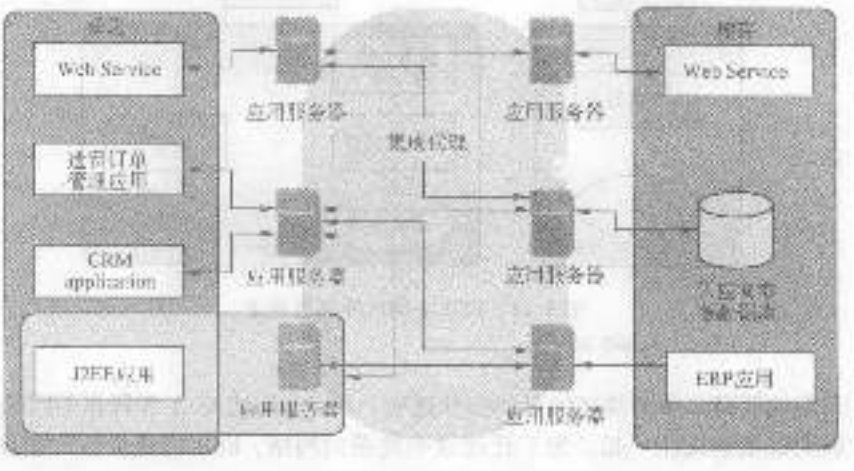


图 8.13　集成代理方式

源自文献：Chappell 2004

为了更好地理解 ESB 环境中的可伸缩性，现以一个分布式的采购流程为例，该流程将来自外部供应商和仓库系统连接在一起。为了满足顾客的需求，可以很容易地将附加的供应商服务添加到库存系统中，如图 8.8 所示。这些附加的服务在另外的计算机系统上运行，它们支持可伸缩性需求，且无须复制整个的集成代理实例。在另一个例子中，通过增加服务所能处理的线程数，采购系统可以增加已有的补货服务的吞吐量。前面的这两个方式都可以在服务本身中增加所需的能力，这与传统的 EAI 集成代理模型不同，后者仅是通过增加额外的集成代理来增加其能力[Chappell 2004]。

在一些情况下，可能需要通过扩展企业主干的吞吐能力来增加系统整体的可伸缩性。图 8.14 显示了合并到 ESB 中的集成代理功能。当达到单个代理的能力上限时，可将多个代理组成集群。代理集群可以充当一个虚拟的代理来处理用户和应用的不断增加的需求。在 ESB 中，集成代理间可以进行通信，并可动态地在总线上分布负载。对此集成代理和代理集群的使用增加了可伸缩性。例如，假如随着库存服务的工作负载的增加，可能导致其所在的计算机过载，这时可增加新的计算机和新的代理来处理工作负载，这将无须改变服务本身，也无须增加任何额外的开发或改变管理方式。单独的可部署的、单独可伸缩的消息传送体系结构和单独的可部署的、单独可伸缩的 ESB 服务容器模型唯一地区别了这个体系结构配置。这些分布式的功能块可以协同工

作，就好像一个单个的逻辑块，并具有一个全局可访问的命名空间，这个逻辑块可定位服务和调用服务。

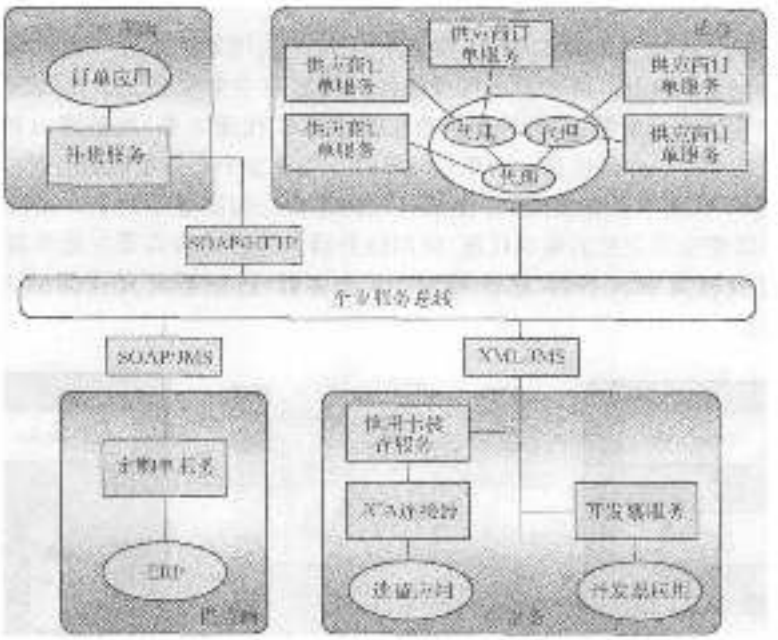


图 8.14 ESB 中的可伸缩性服务

引自 Chappell 2004

基于我们在前面章节中所描述的可伸缩性选项，可将分布式 SOA 部署在不同的地理环境中，然而可将它们无缝地集成在一起。为了管理这个复杂的网络，ESB 将需依赖一组图形化的 BMP 工具（即类 BMP 工具），该工具可监控 ESB 中的通信、寻求可伸缩性方面的问题。当 ESB 需要负载平衡时，也可使用该工具对 ESB 进行重新配置，并且不会导致系统中断服务[Chappell 2004]。

8.5.8 使用 ESB 的集成模式

选择合适的工具和技术是一个非常重要的方面，然而仅靠这一点还不足以成功地实现 SOA。合适的消息交换和集成模式也是同样重要的方面。这些使能模式使得组织能够以可重复的方式发布服务、使用服务，从而确保一致性和稳定性。根据实际经验，一致性和稳定性是跨组织的 SOA 最需要保证的。

将应用迁移到 ESB 中不仅可以增加灵活性，而且是集成 SOA 中的服务的一种方式，并可充分利用 Web Service 技术所带来的好处。连接到 ESB 中的应用必须是服务使能的、松耦合的。图 8.14 显示了一种可行的实施方式，即通过 ESB 实现简单的集成。在这个简单的集成场景中，我们采用了一个独立的集成模式。为了更好地理解，我们假设需要将一个新的供应商添加到采购系统和企业的供应商参考数据库中。添加一个或多个新的供应商的处理逻辑可以作为若干侦听器进行实施，这些侦听器等待来自 ESB 的消息。这些侦听器可以是不同的企业系统的一部分（或者是与企业系统相关的独立应用），这些企业系统需要知道采购系统中的所有可能的供应商。然后，包含供应商信息的消息将被传送到 ESB 中。消息将被送到仅有 ESB 能够读取的事件队列中。基于一个调度，诸如一个固定时间间隔，ESB 将访问队列中的消息，记录消息的源地址和目的地，并将消息转发到合适的事件处理程序，以供进一步处理。

这类集成模式的好处在于，它将一个相当复杂的流程分拆为多个较小的、可管理的片段。遗憾的是，这种集成模式生成了许多小的、隔离的、独立的集成逻辑片段，这些片段实现了一些简单的业务规则，因此这种集成模式显然不具有可伸缩性。

通过业务编配服务实现集成是一种比较好的方式。编配服务是一个可以连接到 ESB 的专门的处理引擎，它可组合和协调在 ESB 上“活着”的服务。编配引擎除了可以支持诸如 BPEL 等流程描述语言，它还提供了附加的业务流程管理(BPM)功能和集成数据访问服务、转换服务。BPM 可管理业务活动的状态。

即使对于具有相当长的时间跨度的业务流程，集成服务也能够管理它们的状态信息。例如，业务流程中的一个步骤可能需要和人进行交互，诸如批准一个定购单或者批准一个新的供应商。服务状态的变化(例如添加一个新的供应商或者安排一个新的定购单)将自动传播到业务流程中的所有被影响的服务中。基于路线计划的路由需要路线计划能够携带状态。假如携带路线计划的消息通过可靠的消息传送进行传输，则基于路线计划的路由还需要能从故障中恢复路线计划。假如消息抵达了它们的目的地，并且没有出现任何故障，则业务流程可以成功地结束，并提交相应的结果。有关此主题的更详细的信息，可参见第 9 章和第 10 章。

对于许多 ESB 应用来说，由于业务流程集成模式具有许多优点，因此它是首选的集成方案。这一方案的唯一缺点是有时处理比较复杂。

在许多行业中，ESB 已经得到广泛使用，包括金融服务、保险、制造、零售、电信、食品等。文献[Chappell 2004]是有关 ESB 特性、功能和实现案例的一个很不错的学习资料。集成模式也逐渐找到了使用 ESB 的有效方式。使用集成模式可提高电子商务应用的开发和部署速度。有关已经得到证明的集成方案的详细信息，以及使用 ESB 实现 SOA 的详细信息可参阅文献[Keen 2004]。

8.6　扩展的 SOA

当我们审视 ESB 的能力时，可以发现 ESB 提供了不同级别上的许多服务功能。在底层，ESB 提供了通常的服务容器的功能，包括通信和连接能力、路由、运行时支持、发现、转换、安全性等。在较高(中间)层，ESB 提供了对长时间运行的流程、非常规的事务的支持，还提供了服务编配的能力。在最顶层，ESB 提供了超越中间层和底层的服务管理和监控能力。因此，将相关的服务进行简化和分组是很有必要的。为了能够有效地处理复杂应用(这些应用使用了 ESB)的功能需求，将服务功能进行分层也是很有必要的。扩展的 SOA(xSOA)将针对这些问题。

xSOA[Papazoglou 2003]、[Papazoglou 2005a]对常规的 SOA 进行了扩展，试图提供一个分层的、基于服务的体系结构。xSOA 中的体系结构层次如图 8.15 所示。它描述了功能的逻辑分离，每一层定义了一组构成、角色和职责。每一层都依靠前面一层的构成来完成它自身的任务。功能的逻辑分离是基于如下需求，即将传统的 SOA 所提供的基本服务能力(例如构建相对简单的基于服务的应用)与对服务进行组合所需的比较高级的服务能力进行分离。功能层的主要目标是确保跨组织的一致性、服务的高可用性、非公开服务和信息的安全性、关键任务复杂应用中的多个服务的编配。功能层还提供业务的所有基本需求——质量服务。

如图 8.15 所示，xSOA 提供了三个层面：基础、组合、管理与监控。xSOA 的最底层是基础层。该层支持 Web Service 通信原语，并利用基本的服务中间件、体系结构构造和功能来描述服务、发布服务、发现服务和执行服务，这一方式已被广泛接受，并非常一致地实施。较高层位于基础层之上。该层定义了业务流程，以及定义了 Web Service 流程和应用的管理与监控。图 8.15 的垂直轴标明了贯穿所有三个层面的服务特性，包括语义、非功能性服务特性和 QoS。xSOA 支

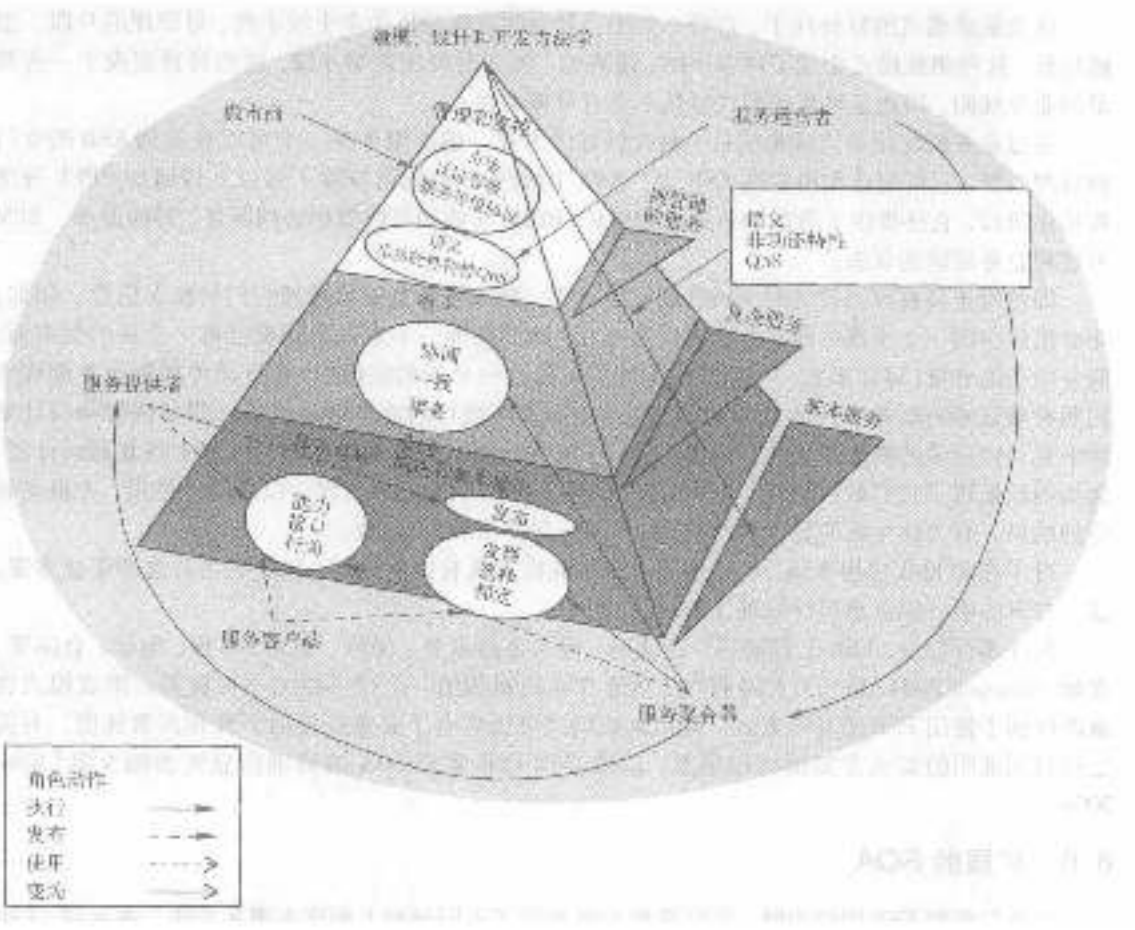


图 8.15 xSOA

持许多角色，除了支持传统的服务客户端和服务提供者角色，xSOA 还支持服务聚合器、服务运营者和技市商角色。下面，我们将要简要介绍 xSOA 各个层面的特性。

服务基础层提供了一个面向服务的中间件框架，该框架实现了运行时 SOA 基础架构。运行时 SOA 基础架构不仅连接了异构的组件和系统，而且提供了异步通信协议以及支持多点访问服务，例如通过位于不同网络上（诸如互联网、有线电视线、通用移动电信系统、数字用户线等）的不同设备（诸如移动设备、掌上型电脑、手持设备等）进行访问。传统的 SOA 功能定义包括通信和服务的描述、发布、发现、绑定、运行时基础架构支持这些传统的 SOA 功能定义。五个标准集实现了服务基础层，它们分别是 SOAP、WSDL、UDDI、WS-Notification（参见 7.4 节）和 WS-MetaDataExchange。

服务基础层支持服务提供者和服务客户端的角色。一个服务可以是服务提供者或服务客户端，也可以同时既是服务提供者也是服务客户端。在图 8.15 中，我们假设服务客户端、服务提供者和服务聚合器能够充当服务代理或者服务发现代理，并发布它们所部署的服务。

当应用和业务流程能够将它们间的复杂的交互集成到组合而成的增值服务中时，将充分发挥 Web Service 对于开发动态的电子商务解决方案的潜力。由于服务技术支持协调，并提供了一个异步和面向消息的方式与应用逻辑运行通信和交互，因此服务技术提供了一个可行的方案。

然而，在 SOAP、UDDI、WSDL 的基准规范和较高层的规范之间进行区分也是很重要的。SOAP、UDDI、WSDL 的基准规范提供了一个基础架构，可支持面向服务的体系结构中的发布、发现和绑定操作。对 Web Service 进行组合时则需要使用较高层的规范。这些较高层的规范不仅可支持和利用服务，而且可用于集成自动化业务流程。较高层的规范位于服务组合层。

服务组合层提供了一些角色和功能，将多个(也可以是单个)服务聚合到一个组合而成的服务将需要用到这些角色和功能。所生成的组合服务可以视作一个独立的服务，既可进一步进行服务组合，也可以作为一个完整的应用/解决方案提供给服务客户端。聚合器完成该任务。图 8.15 中的角色动作表明：服务聚合器可以创建组合服务，通过发布组合服务的服务描述，服务聚合器能够变成服务提供者。服务聚合器开发规范和代码、服务一致性和服务协调。服务聚合器开发的代码使得组合服务能够完成它自身的功能。组合服务的功能基于功能部件，诸如元数据描述、标准术语和参考模型。服务协调控制了组合服务(例如流程)、Web Service 事务的执行。服务聚合器还管理数据流、组合复合间的控制流以及策略的执行。

目前，诸如 BPEL、WS-CDL 等标准作用在 xSOA 的服务组合层。这些标准可以创建大型的服务协作，使得企业间可以自动地经营业务。我们也期望看到更大型的、横跨整个行业集团的服务协作，以及其他复杂的业务关系。这些开发工作需要使用一些工具来了解一些状态，诸如实现 Web Service 的系统的状态，以及松耦合应用的状态和行为模式。因此，必须能够管理 SOA 中的松耦合应用。

SOA 应用的性能依赖于协作服务的组合性能以及协作服务间的交互。故障或单个应用组件的变化可能会影响到多个相互依赖的企业应用。类似地，增加新的应用或组件可能会使得已有的组件过载，使得一些似乎不相关的系统也极大地降低性能甚至出现故障。为了解决这类问题，企业需要不断地监控应用的运行状态。企业必须对应用性能进行调优，使其一直不超过最大负载。

对于生产环境下的 Web Service 和应用而言，一致的管理和监控基础架构是非常重要的，这些功能是由 xSOA 的管理和监控层提供的。这将需要实现一个关键特性：管理和监控服务。对于基于 SOA 的应用而言，服务管理包括在应用的整个生命周期中进行控制和监测。为了确保响应服务的执行和有效地管理服务操作，服务管理将横跨许多活动，从安装和配置到收集应用度量数据和调优等。服务管理和监控包括许多相互关联的功能，诸如 SLA 管理、审计、监控、故障诊断、服务生命周期/状态管理、性能管理、服务和资源供应、可伸缩性、可用性和可扩展性等。此外，服务管理还包括对服务操作的管理。基于正在运行的流程的全局视图(类似于 BPM 工具所提供的功能)，可实现服务操作的管理。

随着 SOA 和 Web Service 技术的采用，出现了一个新的趋势，即居间服务与客户、供应商、管理者、金融部门以及企业操作中所涉及的所有外方进行交互的情况不断增多。这一趋势引发了“开放服务市场(或称作基于服务的交易社区)”。开放服务市场的目的是采用电子化手段为买方和卖方提供相互接触、经营业务的机会。通过提供增值业务服务和集体采购，开放服务市场可将服务供应和服务需求聚拢在一起。这类服务集市的范围基本不受制约，只要其他企业能够看到所提供的服务，以及遵循业务所涉及的具体的行业协议。服务市场通常向它们的成员提供一个统一的产品和服务视图、标准的业务术语和命名约定、标准的业务数据格式、标准的服务和流程，以及流程编配的标准定义。此外，服务市场必须提供支持行业交易的多种服务，包括提供业务事务协商和促进、财务结算、服务认证、质量控制、分级服务、服务度量(诸如当前服务请求者的数量、平均周转时间等)。服务市场还必须管理 SLA 的协商和执行。这一功能引入了做市商这一角色。做市商是支持和维护一个开放服务市场的实体。

最后，开发高级应用的要点在于按照 xSOA 各个层面进行分层并使用 SOA 生命周期方法学。

按照服务开发方法学，首先需要对业务环境（包括业务目标的关键性能指标）进行分析和建模，然后将模型转换为服务设计，部署服务系统并管理部署。

8.7 小结

除非 Web Service 的消息交换是可靠的，否则对于工业级的业务应用和关键任务操作（诸如 B2B 事务或实时企业集成），企业无法在这些应用中信任地部署 Web Service。在本章中，我们描述了一个消息传送协议和相关的标准。基于该消息传送协议和相关的标准，可在分布式应用中可靠地传送消息，即使出现了软件、硬件或网络故障，依然能保证消息的可靠传送。该协议独立于传输，因此可以采用不同的技术实现具体的传输。

对于松耦合的、标准的、协议独立的分布式计算，面向服务的计算体系结构可以使得相关问题的处理变得更容易，而面向服务的计算体系结构的核心则是可靠的消息传送协议。我们已经讨论了面向服务的体系结构和技术是如何有效地实现所需级别的业务集成，以及 IT 实现如何更紧密地映射到整体的业务流程。尤其是，我们致力于一些特定的技术和方法，这些技术和方法能够统一面向服务的体系结构和支持基于事件的编程中的原理与概念。这些方法可推动业务集成项目的实施，并可提供一个灵活的、适应性强的环境，从而可以阶段式地、高效地集成 Web Service。尤其令人感兴趣的是企业服务总线技术，它提供了一个可管理的、基于标准的信息技术框架。这一框架扩展了整个业务价值链中的面向消息传送中间件的功能，并把异构组件和系统连接在一起。

将 Web Service 标准与 ESB 基础架构结合在一起可以最广泛地实现系统间的连接性。ESB 支持 Web Service 和多个已有的应用集成技术，可以充分发挥新老技术的作用。

最终结论是，ESB 目前尚无法广泛取代 EAI。ESB 虽然能够作为一种灵活的、低成本的集成方法，但是由于在一些关键功能方面（诸如复杂数据转换、对于应用适配器的支持、业务流程建模）的不足，因此尚不适合广泛取代 EAI 解决方案。与此相反，EAI 产品通常都提供了这些重要的特性。然而，期待在不远的将来这一状况能有所改善。

复习题

- 什么是系统质量属性？它们与软件体系结构间关系是什么？
- 简要描述常见的体系结构约束，并说明它们与 SOA 的关系？
- 什么是可靠的消息传送？对于基于 SOA 的应用，可靠的消息传送为何极其重要？
- 简要描述 WS-Reliable Messaging 模型。
- 什么是企业服务总线？它与 SOA 的关系是什么？
- 事件驱动的 SOA 的目的是什么？
- 企业服务总线如何实现事件驱动的 SOA？
- 简要描述 ESB 的关键能力。
- 企业服务总线最常使用的 Web Service 标准是什么？
- 简要描述 ESB 解决方案中使用的集成方法？
- ESB 解决方案中是如何使用集成代理和应用服务器的？
- ESB 解决方案是如何实现可伸缩性的？

练习

8.1 假定定购单服务允许它的客户下大宗订单，且大宗订单作为同一个序列的一部分提交，并使用 WS-ReliableMessaging 完成提交。为了处理这个问题，使用 WS-ReliableMessaging 编码

SOAP 头部。使用清单 8.1 中的代码片段帮助开发解决方案。

8.2　一个商品生产企业依赖运输公司的服务将货物分发给它的客户。该企业希望开发一个 SOA 解决方案，以便能够将它的系统与运输公司、供应商和客户的系统集成在一起。由于该企业通常与多家运输公司进行业务合作，因此需要向那些运输公司提供一个单一的接口。此外，该企业需要集成解决方案能够提供订单的可视化，当货物通过分发链运送给客户时，能可视化地跟踪运送的情况。因为该企业无法掌控合作伙伴采用何种具体技术，因此解决方案将采用开放标准。开发一个基于 Web Service 代理的 SOA 体系结构，并且需要向访问那些应用的其他各方提供安全的、可管理的访问。

8.3　修改前面一道练习中的解决方案，以便进行交易的企业之间的服务交互能够在一定的级别上进行组合，从而构成一个业务流程和工作流解决方案。使用一个合适的业务流程执行语言对这些流程进行明确的建模以及进行相应的执行，并且业务流程执行语言必须遵循合适的开放标准。

8.4　一般说来，业务流程集成可能产生集中业务集成模式，这些业务集成模式的目标就是管理和支持业务自动化和跨企业的集成[Papazoglou 2006]。业务流程集成模式包括：整合企业集成模式、代理企业集成模式和联邦企业集成模式。对于一个具体企业，根据业务需求、业务结构和企业中的业务优先级别，选择最合适的业务流程模式。

基于集成业务模式，可以按整体的方式在所有业务单元中管理和监控端到端的业务流程。可跨整个企业查看业务流程，其中活动横跨多个组织单元，并且由各个对应的业务组执行这些活动。业务集成模式假定工作流和集成流程可以跨组织单元，并可由企业中的一个组进行管理。该组也负责设置有关工具选择和消息标准的策略。对于小企业或者那些使用了通用的标准化的集成工具的大企业而言，整合企业业务模式是一种最合适的方式。

对于应用了整合企业业务模式的大型企业，设计一个 ESB 解决方案。在该企业中，部门间存在端到端的业务流程，并且在不同的部门中执行的这些活动可视为业务单元(可包含一个或多个部门)中的企业活动。不同业务单元的角色确定了执行活动的职责，并且这些业务单元的边界对于业务流程并没有影响。在这个 ESB 解决方案中，外部合作伙伴负责执行物流和分发流程，诸如仓储、分发和管理库存。物流将被无缝地集成到企业的端到端的业务流程中，并且物流将依赖于托运收据、货物通知单、发票和其他业务文档的交换和处理。

8.5　在代理企业业务模式中，通过代理业务单元，可跨所有业务单元管理和监控流程。这是一个分布式的业务企业模式，其中业务单元是完全自治的，但使用业务代理来管理通信和互操作。单个的业务单元仍然拥有所提供的流程，但是必须遵循消息传送标准和业务单元处理标准，并且必须注册它们所提供的服务。流程支持这些标准以及代理单元注册。基于该模式，代理单元可处理所有的请求，因此服务客户端不知道哪一个组织单元提供了处理这个服务的业务流程。假如企业中的各个组织单元想要维持它们的自治，但是又希望能从集中型服务管理和通用消息传送基础架构中获益，则可应用代理企业模式。

对于应用了代理企业业务模式的大型企业，设计一个 ESB 解决方案。

8.6　在一个大型企业或一个整合型价值链中，若单个的业务单元是完全自治的，并且希望能在需要时进行协作，则可使用联邦企业业务模式。因为没有一个整体上的业务管理和监控，所以不能端到端地查看业务流程，并且每个单元负责提供和维护标准接口，以便单元间可以相互合作。消息传送和服务描述都采用了通用的标准，并且每一个业务单元管理它自身的服务信息库。当单个的组织单元想要维护它们的自治，但是又希望能从业务单元间的相互合作、通用消息传送基础架构和合作标准的使用中获益，则可应用联邦企业模式。

对于应用了联邦企业业务模式的大型企业，设计一个 ESB 解决方案。

第五部分　服务组合与服务事务

第9章　流程与工作流

学习目标

为了创建业务流程，需要对 Web Service 进行重合。服务组合指的是将多个 Web Service 进行重合。为了将单个的 Web Service 组合成具有适当复杂度的、可靠的、基于业务流程的解决方案，各种 Web Service 组成语言和技术就应运而生了。

在本章中，我们将介绍工作流、业务流程技术，并讨论如何使用它们将包含 Web Service 的应用连接在一起。在阅读本章之后，读者将能理解下列关键概念：

- 业务流程和工作流系统。
- 如何集成和管理业务流程。
- 业务流程解决方案的主要组成部分，诸如流的建模和 Web Service 的组成。
- 流程编配和流程编排的概念。
- Web Service 的业务流程执行语言的主要构成。
- Web Service 编排描述语言的概要视图。

9.1　业务流程及其管理

流程是一个具有起点和终点的有序的活动序列，它有输入（通过资源、素材和信息进行表示）和一个对应的输出（它产生的结果）。因此我们也可以将流程定义为一个步骤序列：它由一个事件引起，然后对信息等进行转换，并产生一个输出[Harmon 2003a]。业务流程是实现既定的业务结果的一组逻辑上相关联的任务。（业务）流程视图意味着从横向察看业务组织，并且把流程看作为一组相互依赖的活动，这些活动将针对一个客户或市场生成一个具体的输出。业务流程定义了预期的结果、活动的背景、各活动之间的关系，以及和其他流程、资源的交互。当业务流程和活动序列的状态发生改变时，有可能会产生一些事件，业务流程可以接受这些事件。业务流程可以生成一个事件作为其他应用程序或流程的输入。业务流程也可以调用应用程序来完成计算功能。并且它也可以进行人机交互，向用户指派工作列表、要求用户执行相应的操作。业务流程不仅能够被度量，而且可以基于不同的性能指标进行度，诸如成本、质量、时间和客户满意度。

每个企业都有其内嵌在业务流程中的独特特点。大多数企业执行一些类似的、可重复的例行活动，其中包含生产产品和开发服务，将这些产品和服务向市场推广，并满足购买这些产品和服务的消费者的需求。自动化的业务流程可以执行这类活动，我们可以把自动化的业务流程看作一个精确或、系统地编排过的活动序列，可用于完成特定的任务。在制造企业中，流程的典型

例子包括开发新产品(贯穿研究和开发、市场推广、制造整个过程)、履行客户订单(集成了销售、制造、仓储、运输和结算),以及资产管理。对流程进行设计、结构化和度量,并把它们对客户的价值,使它们成为业务改进和业务创新的重要起点。

在一个企业中,最大的流程可能是价值链。价值链可分解为产品生产或产品线所需的一组核心业务流程和支持流程。核心业务流程可再分为若干活动。活动(Activity)是执行流程中的特定功能的元素。活动既可以像发送消息、接收消息那样简单,也可以很复杂,如协调其他流程和活动的执行。一个业务流程可以包含复杂的活动,其中一些复杂活动是在后端系统上运行的,例如信用卡核查、自动结算、订购单、库存更新、运送,或是不重要的活动,例如发送文档、填表格。业务流程活动可以调用其他相同或不同业务领域的流程。随着公司的不同、具体业务的不同,活动必然会发生很大的变化。

在运行时,业务流程的定义可以有多个实例,每个操作彼此独立,且每个实例可以有多个并发执行的活动。流程实例是一个由工作流引擎制定(管理)的被定义的活动线程。一般而言,在运行时能够观察到流程实例、它的当前状态以及它的动作记录,并可按照业务流程定义进行表示,从而用户能确定业务活动的状态,业务专家也能监控该活动,并能因此识别出业务流程定义的需要改进之处。

业务流程的特性

业务流程通常与经营目标和业务关系(例如保险索赔流程或工程开发流程)相关联。它可以是完全包含在单个的组织单元中,也可跨越不同的组织,例如像在供求关系中那样。由买方和卖方联合建立的采购和销售流程,就是一个跨组织边界的流程的典型例子。采购和销售流程需要基于 EDI 和增值网络基础之上。目前,互联网已经触发了一些新的业务流程,并促使已有的业务流程的重新设计。例如,Web Service 旨在设计标准化的、基于业务流程的解决方案,随着 Web Service 技术的使用,许多业务流程解决方案也随着有所变化。

每个流程有一个消费者(或称客户),并且每一个流程都是由消费者的命令发起的。消费者可以是外部的,如服务或产品的最终消费者;也可是内部的,如另一个流程,对原先的流程而言,它的输出形成了另一个流程的输入。并非每个流程都由一个消费者命令触发。流程可由一个标准的程序(事件)触发。例如,薪水支付可在每个月的一个特定日期触发。

每个业务流程都意味着处理:一系列的活动(处理步骤)导致流程中的一定形式的数据或产品的转换。转换可以手动执行或自动执行。一个转换可以包含多个流程步骤。例如,“核定发货单”流程就包括“核对发货单是否已付款”、“核对商定的采购条件”、“核对验收单”、“核对计算”和“核对用户的姓名、地址和银行账户”。当且仅当所有的核查项目都正确时,才会在到期应付账户中记录该发货单。

一般情况下,工作流应用与业务流程和它们的活动相关联。它不管这件事是物流链管理、供应链计划,还是仅仅只是业务合作伙伴之间的文档交换,亦或其他类型的围绕业务流程的应用。

流程有决策点。对于路由和处理能力的分配,必须进行决策。在预见度和标准化程度都较高的环境中,客户订单的流程轨迹将事先按标准方式建立。只有在流程比较复杂,或流程的状态不可预期时,才需要当场进行路由决策。通常说来,客户订单会被划分成一个高度程序化的(因而也是自动化的)类别,这个类别是复杂且不确定的。此处需要用户干预,并且手动处理也是流程的关键要素。

最后,每个流程都会生成一个结果,如抵押借款或得到认可的发货单。在不同的流程中,流程的最终结果能被事先指定的程度以及能被标准化的程度都将不一样。这一程度的高低将影响对流程和工作流进行结构化和自动化的方式。

综上所述，实际的业务流程可由以下行为来描述：

- 它可以包括已定义的状态，这些状态将触发每个新实例（例如一个请求的到达）的初始化，还包含已定义的完成时的输出。
- 它可以包含参与者之间的正式的或相对的言非正式的交互。
- 它有一个能在很大程度上变化的周期。
- 它可以包括一系列的自动化的活动和/或手动活动。活动可以大而复杂，涉及物流、信息流材料、信息和业务承诺。
- 它表现出极为动态的特性，因而能响应消费者的需求，也能对变化的市场状况作出应变。
- 它可跨组织内和组织间的边界进行广泛分布和定制，还经常横跨基于不同技术平台的多个应用。
- 它通常长时间运行。一个诸如从订单到货款这样的流程实例就可能运行数月甚至数年。

9.2　工作流

与业务流程紧密相连的是工作流。根据一组程序化的规则，工作流系统将文件、信息和任务从一个参与者传递到另一个参与者，从而使业务流程部分或全部地自动化[WfMC 1999]。工作流是基于文档的生存周期和基于表单的信息处理，因此它通常支持良定义的、静态的、"文书的"流程。由于业务流程能在软件清晰地表达，因此工作流提供了透明性。工作流所生成的定义能够快速部署和改变，因此工作流也具有很高的灵活性。

工作流可定义为处理步骤的序列（业务操作、任务、事务的执行），其中信息对象和物理对象由一个处理步骤传送到另一个处理步骤。工作流中将涉及一些能自动地路由消息和任务的工具，工作流能把这些工具和技术与程序或用户联系在一起。

可使用面向流程的工作流实现了流程自动化。那些流程的结构是良定义的、稳定的，不会随着时间而变化。工作流时常协调多个计算机上执行的子流程，并且较少地需要用户的参与（时常仅在特定的情况下）。良定义的流程的例子包括定单管理或贷款请求。一些面向流程的工作流可以有事务特性。

面向流程的工作流由任务和检查点组成，其中任务需遵循路由规则，而检查点则由一些业务规则表示，例如"暂停等待信用审批"就是这类检查点。这类业务流程规则管理活动的整体处理，包括请求的路由、将请求分配或分发到特定的角色、工作流的数据在各个活动之间的传递、业务流程活动之间的依赖性和相互关系。

工作流涉及活动、决策点、路由、规则和角色。下面将进行简要描述。

就如流程一样，工作流通常由一定数量的逻辑步骤组成，每个逻辑步骤成为一个活动。活动是由工作流编纵的一组动作，如图 9.1 所示，活动可以涉及与用户或工作流参与者之间的手动交互，或使用不同的资源（诸如应用程序或数据库）来执行。工作流将创建工作项或数据集。为了实现特定的业务目标，在一些阶段的许多次检查点（图 9.1 中的步骤）将处理和改变这些工作项或数据集。大多数工作流引擎能处理非常复杂的流程。

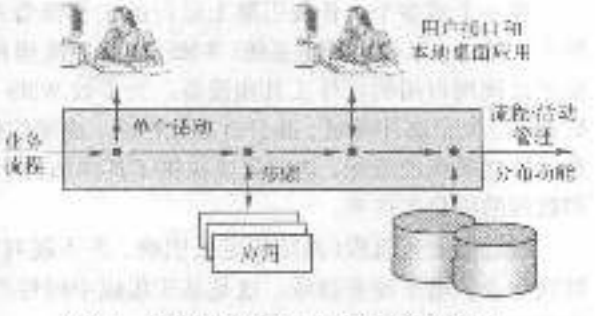


图 9.1　跨多个应用的工作流管理流程图

工作流可以描述业务流程的不同方面，

包括自动的活动和手动的活动、决策点和业务规则、并行的和顺序进行的工作路由，以及如何管理正常业务流程的异常。

工作流可以有一些逻辑决策点。在事件有多条路径可供选择时，逻辑决策点可确定工作项目应该采取工作流的哪条分支。通过一组有界的逻辑决策点可标识和控制工作流中每个可选路径。支持一个工作项目的工作流实例包括所有可能的由始至终的路径。

基于依赖关系、序列选择和迭代，工作流技术使得开发者能够描述组织内和组织间的所有业务流程。基于工作流技术，开发者能够有效地描述用于业务流程处理的复杂规则，例如基于领域内容的合并和选择、基于时间的消息传送等。为实现这些目标，工作流将以预先指定的路由路径为基础。对于组成工作流的一组对象而言，这些路由路径定义了这些对象所采用的路径。工作流的路由可以是顺序的、循环的、或并行的。路由路径也可是按顺序、并行或循环方式。在工作流管理系统所管理的流程实例的片段中，多个活动可在一个单一线程上顺序执行，这即称作顺序路由。在工作流管理系统所管理的流程实例的片段中，两个或多个活动实例在工作流中并行执行，从而将导致多个控制线程，这即称作并行路由。并行路由通常以并行分支开始，以并行汇聚点结束。在工作流中，分支点是一个同步节点。在其上，单个的控制线程将分化为两个或多个并行执行的线程，从而能够同时执行多个活动。在工作流中，汇聚点也是一个同步节点。在其上，两个或多个并行执行的线程汇聚为一个共同的控制线程。在顺序路由中，不存在分支点与汇聚点。工作流还包含两种同步节点："或分支(也称条件路由)"节点和"或汇聚(也称异步汇聚)"节点，这两类节点将应用在顺序路由和并行路由中。在工作流的某个节点中，当单个的控制线程决定在多个工作分支中选择哪一个分支时，则该节点称为条件路由。最后，在工作流的某个节点中，两个或多个可选的活动工作流分支重新聚合为一个单一的共同活动，并作为工作流中的下一步，则这个节点称为异步汇聚。必须注意到，因为在汇聚点没有并行执行的活动，所以并不需要进行同步。

在工作流中，每个决策点上的业务规则决定了如何处理、路由、跟踪和控制与工作流相关的数据。业务规则是一些核心业务策略，这些策略针对企业的业务模型，并定义了移动到工作流下一阶段所需满足的条件。业务规则可表示为一些简介的语句，它们具体针对应用中的业务的某一方面。同样地，业务规则可以确定所需遵循的路由[vonHalle 2002]。例如，对于一个医疗保健应用，业务规则可以包括：如何实施新的索赔确认的策略、如何实施提交需求的策略、如何实施程序审批的策略等。

在工作流中，角色定义了工作流中相关人员或程序的功能。在工作流中，角色是一种将参与者与一组工作流活动相关联的方式。角色定义了用户参与特定流程或活动的背景环境。角色通常涉及组织概念，例如结构和关系、责任或职权，但也可以涉及其他属性，如技能、地址、数值数据、时间或日期等。

在一个或多个工作流引擎上运行的工作流管理系统可实现工作流的定义、创建和管理。如图9.1所示，工作流管理系统(WMS)能解释流程和活动的定义，与工作流参与者交互，并能在需要之处调用可用的软件工具和设备。大多数WMS与一个企业中的其他系统(诸如文件管理系统、数据库、电子邮件系统、办公自动化产品、地理信息系统、生产应用等)集成在一起。对于涉及大量的独立系统的流程，这种集成提供了具体结构。它也能提供方法，比如管理来自不同源的文档和数据的项目文件夹。

通过把业务流程(其结构定义明确，并不随时间改变)自动化，现代工作流产品将其功能扩展到企业和电子商务领域。这是基于集成中间件产品、流程串行化、编配机制和事务处理能力来实现的。

下面，我们将主要讨论订单管理场景的一种变体，本书的前几章中介绍过这个场景，并把它看作是描述相当复杂的、面向流程的工作流步骤的基础。在订单管理流程中，活动包括接受销售订单、分配库存、产品运送、结算，以及保证收到付款。这些流程中的一些可能长时间运行，另一些则可能在毫秒级时间内完成。有些活动是在供应商站点执行，诸如核对客户的信用度，确定能否在产品库存中得到所要订购的部件，计算订单的最终价格，与消费者做结算，选择付运者，以及推定订单的生产和运送日程。

这个业务流程的简化版如图 9.2 所示。图中的所有步骤都涉及流程间的协调和会话，自定义的警告可以跨网络地跟踪异常，以及在必要时可提供手动干预。

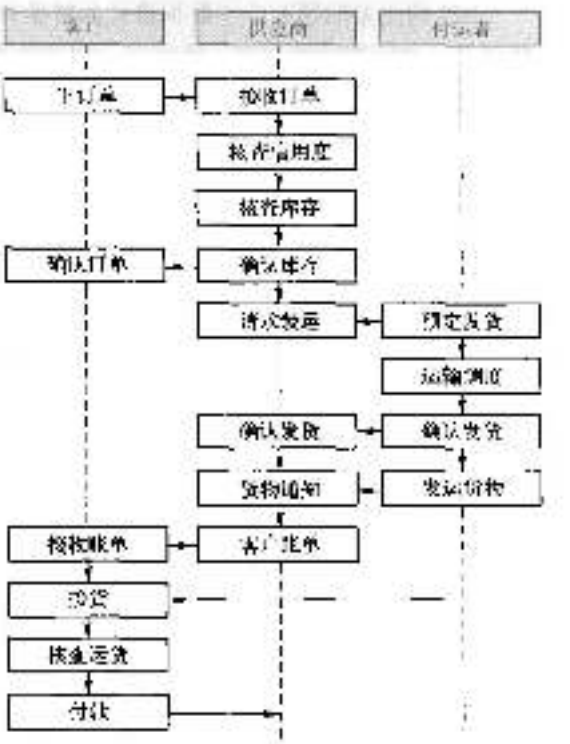


图 9.2　订单管理流程的流图

在前面的例子里，业务定单管理流程描述了如何将产品从供应商那里送到顾客手中。该流程的一个实例把产品发送到一个特定顾客手中。流程实例由活动实例组成，而活动实例包含了实际工作项目的。这些工作项目将传递给工作流的参与者——客户、供应商、付运者。这些参与者即可以在本活动动作，或在其他流程中采动作。例如，在定单管理流程中，一个具体的运输公司会收到与装运相关的某一产品的所有文档，并会被要求提供一个具体装运运日期以及对应的运送价格。

工作流技术常常把集成功能（诸如在不同的打包应用和遗留应用间同步数据）交付给工作流活动中的定制代码来实现，即在流程模型的作用域之外实现这些集成功能[Silver]。此外，工作流技术使用了一种应用层 API 的紧耦合的集成方式，因此工作流将局限于本地的同类系统环境，诸如局限于一个部门内。因此，传统的工作流的实现与部署它的企业紧密捆绑在一起的，不能可靠地延伸到组织边界之外，到达消费者、供应商或其他合作伙伴那里。因此 WMS 的主要的不足之处就在于集成：它不善长把跨企业的系统联接在一起。为了解决这个问题，现代的工作流技术应用了业务流程管理功能，试图将功能扩展到跨企业的流程集成。在 9.3 节中，我们将对这一问题进行更深入的探讨。

9.3　业务流程的集成与管理

业务流程是所控制的特定组织的定义元素，它们精确地描述了如何管理、控制内部和外部的业务，诸如客户如何下订单、合作伙伴和供应商的货物需求、员工更新内部系统等。因此，为了最大限度地给客户提供增值服务，需要对企业内部集成和跨企业的集成提供端到端的业务流程。业务流程涉及整个价值链，而不是单个企业。它们交付产品或服务，使得企业更具有竞争力，从而能够在激烈的竞争中胜出。价值链管理现在已被视为企业提高生产力和竞争优势的下一个增长点[Papazoglou 2006]。因此，必须以系统的、前后一致的方式管理和部署企业内部的集成和跨企业的集成。

业务流程模型规定了系统之间序列、层次、事件、执行逻辑和信息流动，其中这些系统既可以驻留在同一个企业内（即 EAI），也可以驻留在多个互连的企业中。业务流程集成（BPI）描述了定义普遍接受的业务流程模型的能力。BPI 是一个集成解决方案，可向企业提供端到端的可视性，并可对多步式的信息请求和事务中的关键对象进行控制，包括人员、客户、合作伙伴、应用程序和数据库。例如，业务流程集成可以包括订单管理、库存管理或执行流程中的所有步骤。

BPI 的主要问题不在于数据格式的转换和路由，而是如何将嵌入在一个应用中的业务流程桥接到另一个应用的流程中。连接在一起的业务流程不是作为信息进行定义，而是按照活动或工作流进行定义。

业务流程可能横跨企业的多个系统。通过对业务流程进行自动化，以及管理这些业务流程，BPI 解决方案使得企业可以充分利用这些已有的系统。通过 BPI，企业可以保护它们在遗留系统上的主要投资，无须重新编码来实现已有的功能。

图 9.3 显示了一个典型的流程集成工作流应用，该应用涉及了跨组织边界的订单管理流程。这个典型的业务流程横跨企业内（甚至企业间）的多个功能域。该图显示了如何将业务流程层指定的流程映射到信息流层的对应的 EIS。该图也显示了一个分支点，这个分支点涉及“装运货物”活动和“客户账单”活动。这两个活动可并发执行。这些活动将在汇聚点汇聚并触发一个“付款”活动。在信息流层，这两个活动用实线表示，从而将它们与用虚线表示的其他活动（消息）区分开来。

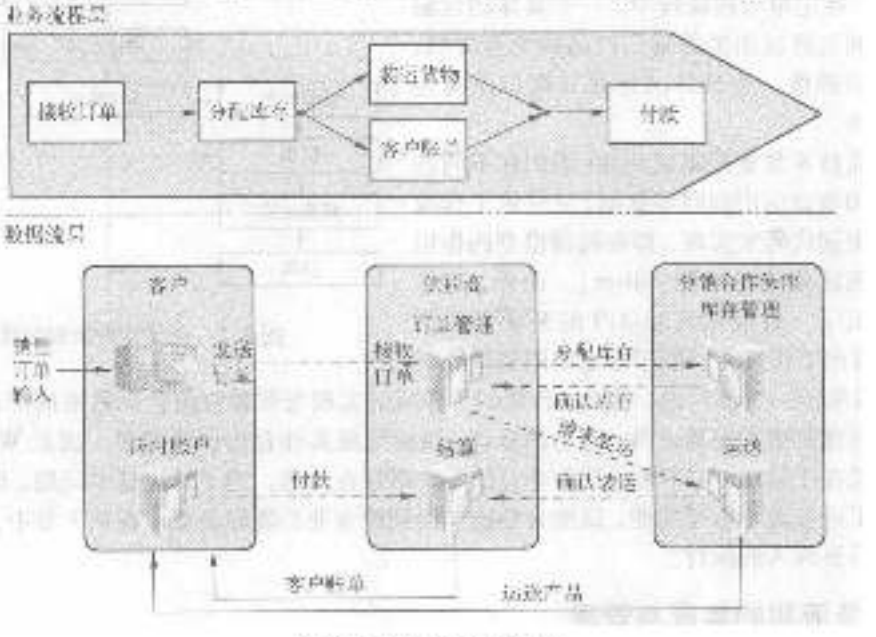


图 9.3 BPI 工作流样例

业务流程既可以是比较简单的、完全自动化的，也可以是复杂的、需要人机交互的流程，因此需要将工作流技术嵌入到集成解决方案中。在业务流程的上下文环境中需要清晰地标识、添加一些规则，这些规则将控制用户如何、何时与业务流程进行交互，以及哪些用户将与业务流程进行交互。因此，除了将关键业务流程扩展到企业防火墙之外，现代的 BPI 工具既可以调整和完善人的管理，也可优化业务流程。可使用一些关键的性能指示器和指标来设置系统的耐受力。基于这些关键的性能指示器和指标，业务流程事务可以延伸到边界之外。因此，对于业务状况的变

化，企业可以及时、有效地响应，而无须费时、费力地额外编程。

BPI 在管理方面的扩展通常被称为“业务流程管理(BPM)”。BPM 强调自动化流程的管理，期望能不断实现最大的灵活性，同时又将开发方面的成本和时间降到最低。

业务流程对内部业务环境或外部业务环境中的事件进行响应，BPM 根据业务流程集表达、了解、表示和管理业务(或业务的一部分)[McGoveran 2004]。术语业务流程管理包括流程分析、流程定义和重新定义、资源分配、调度、处理质量和效率的测量、流程优化。流程优化包括实时测量(监测)、战略测量(性能管理)以及它们间的关联关系的收集和分析，从而对流程的进一步改进和创新打下基础。

BPM 解决方案是一个图形化的效能工具，可跨任何应用、公司边界或人际互动对各种规模的流程进行建模、集成、监测和优化。对供应链以及其他的一些企业内部功能的集成等常见需求推动了 BPM 技术的发展，采用 BPM 技术甚至无须较多的定制软件开发。BPM 可有计划地安排价值导向的流程，并可将这些流程在企业内的执行制度化[Roch 2002]。这意味着 BPM 工具可帮助分析、定义和实施流程的标准化。BPM 提供了一个建模工具，能够可视化地构建、分析和执行跨职能的业务流程。

BPM 不仅仅是流程自动化或传统的工作流，它增加了概念创新以及一些源自 EAI 和电子商务集成中的技术，并在基于 Web 和 XML 标准的电子商务基础架构中重新实现了这些技术。因此这意味着：在一个基于 Web 技术标准的集成化建模和执行环境中，BPM 软件真正地组合了工作流、EAI 和电子商务组件。在 EAI 和电子商务集成中，BPM 提供了跨职能流程自动化所需的灵活性。为了理解这一议题，以如图 9.2 所示的一个典型的电子商务场景为例，它涉及了订购应用和销售应用的连接。这些应用可以提供传统的工作流功能部件，然而这些功能部件仅在本地环境中才能很好地发挥作用。对于跨组织的流程，需要集成化的流程管理。自动化跨职能的活动，诸如核查或确认在企业和它的分销合作伙伴之间的库存，使得企业能够基于实时事件管理流程，而这些实时事件是由集成环境所驱动的[Roch 2002]。然后，流程将能自动化地执行，除非在出现异常(例如库存量小于一个重要的阈值)或者需要手动操作和审批的情况下才需要人的干预。

现代的工作流系统已经包含了 BPM 技术的一些高级的功能部件。现代的工作流系统将针对同样的问题，例如跨组织边界的应用集成。因此从表面上，现代的工作流技术和 BPM 技术似乎已经合二为一。然而，它们仍然有一些细微的差异。它们的差异主要在于体系结构和适用范围的不同。

BPM 和工作流之间的区别主要基于 BPM 系统的管理方面。虽然 BPM 技术同样覆盖了工作流技术中的相关议题，然而 BPM 主要致力于业务用户，并提供了比较复杂的管理和分析能力。BPM 工具非常强调管理和业务功能。BPM 强调业务的适用范围，而工作流系统则主要致力于技术解决方案。基于 BPM 工具，业务用户能够管理一些类型中流程(例如索赔流程)，并可根据历史数据或当前数据对流程进行分析，还可生成成本或其他的业务度量，以及基于这些业务度量生成业务图表或业务报告。此外，基于不同类型的索赔，业务用户将能分析和比较数据、业务度量。现代的工作流系统通常并不提供这类功能。

9.4　跨企业的业务流程

Web Service 提供了标准的、可互操作的集成方式，可集成松耦合的、基于 Web 的组件，而这些组件则需要暴露良定义的接口。与此同时，Web Service 抽象了实现细节以及具体平台细节。诸如 SOAP、WSDL 和 UDDI 等核心 Web Service 标准对于实现这一目标提供了坚实的基础。然而，这些标准主要用于简单的 Web Service 应用的开发，简单的 Web Service 能够提供简单的交互。然

而，Web Service 的最终目标是促进企业内部和企业外部的业务流程的协作，并实现协作的自动化。在 EAI 和 B2B 环境中，有用的 Web Service 业务应用需要具有组合复杂的、分布式 Web Service 集成的能力，并需要能够描述低层的、参与组合的服务间的关系。因此，协作业务流程可作为 Web Service 集成来实现。

在 Web Service 领域，业务流程规定了操作的可能的执行顺序。这些操作源自逻辑上相互关联的 Web Service 集合，每一个 Web Service 完成流程中的一个良定义的活动。业务流程可以指定在这些服务间传送的共享数据、流程中的交易合作伙伴角色、Web Service 集的共同的异常处理。业务流程也可以指定以及其他的一些因素，这些因素将影响到 Web Service 或组织如何参与一个流程[Leymann 2002]。尤其，为了增加由 Web Service 组成的业务流程的一致性和可靠性，可在 Web Service 间指定长时间运行的事务(参见第 10 章)。

由于服务具有平台中立性，因此可将不同企业中的已有的简单服务或复合服务组合为复杂服务，其中组成复杂服务的简单服务或复合服务都可称为组件服务，而组合而成的复杂服务可视为高层服务或流程。我们使用术语“组合性”来描述独立的服务规范。将这些独立的服务规范组合在一起可提供更强大的能力。在一个逻辑流中，通过组合新的应用或已有的应用，复杂服务(和流程)可以集成多个服务，从而形成新的业务功能。服务组合将满足某一组合模式的服务组合在一起，从而实现一个业务目标、解决一个问题或者提供一个新的服务功能。复杂服务的定义需要在组件服务之间协调控制流和信息流。业务逻辑可以视为排序、协调和管理 Web Service 间的交互的手段之一。若要编程实现复杂的跨企业的工作流或业务事务，很可能需要将不相关联的 Web Service 活动从逻辑上链成跨企业的业务流程。Web Service 组合技术很大程度上依赖于业务流程建模和工作流处理语言。

服务组合基础架构依赖于轻量级类工作流技术和流程集成技术，使得一个组织可以无缝地集成或组合内部的业务流程，并可将它们与合作伙伴的业务流程进行动态集成。组合而成的服务可作为增值服务。为了描述业务流程，可基于服务组合机制将企业内部的 Web Service 进行无缝组合，从而支持各类 EAI 场景[Papazoglou 2006]。此外，为了描述合作伙伴间的交互，可对源自不同企业的 Web Service 进行协调。这通常支持了电子商务集成。例如，通过组合一些简单服务，诸如核查客户的信用状况的服务、确定所订购的零件在库存中是否有存货的服务、计算最终价格并与各户结算的服务、选择装运者的服务、调度订单的生产和装运的服务，可实现订单管理流程的开发。

对于 EAI 和电子商务应用，业务流程能够通过 Web Service 间的协作实现共享的业务任务。相应地，在 Web Service 环境中，业务流程工作流由若干活动做成。这些活动作为一组限域的操作实施，其中操作可以横跨多个 Web Service。这些以 Web Service 为中心的操作遵循路由和检查点，而检查点则是通过业务状况和规则进行表示的。

在电子商务应用中，交易合作伙伴必须在它们自己的、私有的业务流程(工作流)中运行，并且必须对业务流程进行编配和协调，以便确保协作业务流程是可靠的、不会失效，并且确保协作业务流程无须挂起自己等待其他流程或系统响应请求。这需要流程在合适的时候能够进行异步通信，并能基于 CORBA 或 DCOM 等同步技术支持 BPI 模式。

企业工作流系统目前支持长时间运行的流程的定义、执行和监控。这些流程协调多个业务应用的活动。然而，因为这些系统面向活动而不是面向通信(消息)，所以没有将内部描述与外部协议描述进行分离[Leymann 2000]。当流程横跨业务边界时，由于涉及的当事方并没有共享应用和工作流实现技术，并且不允许从外部控制它们的后端应用，因此需要采用基于合适的外部协议的松耦合技术。这类业务交互协议以消息为中心，它们指定了消息流，消息表示了交易合作伙

伴之间的业务动作，并且无须任何具体的实现机制。企业应用暴露了参与消息交换的 Web Service。对于这类应用，Web Service 的松耦合、分布式特性使得集中式的工作流（或集中式的中间件技术的实现）无须对企业应用的活动进行彻底的、完全的协调和监控[Arkin 2001]。以上内容可小结如下：

- 传统的工作流模型依赖基于消息的计算方法。对于电子商务或应用集成，该方法与协议是紧耦合的。这些协议假定具体环境是紧连接的、可控的，然而这与 Web 的特性是不相符的。
- 传统的工作流模型没有将内部实现与外部协议描述进行分离。
- 在传统的工作流中，参与方通常是预先知道的。然而在 Web 环境中，很可能将动态选择 Web Service 实现一个角色。

三个标准解决了 Web Service 编配问题。这三个规范是：业务流程执行语言（BPEL4WS 或简写为 BPEL）[Andrews 2003]、WS-Coordination（WS-C）和 WS-Transaction（WS-T）[Cabrera 2005a]、[Cabrera 2005b]、[Carrera 2005c]。这三个标准相互协作，为许多任务打下了一个坚实的基础，诸如可靠地编排基于 Web Service 的应用、提供 BPM、事务完整性、通用协调工具等。BPEL 是一个类工作流定义语言，它描述了能编配 Web Service 的复杂的业务流程。事务处理系统、工作流系统或者其他希望协调多个 Web Service 的应用需要用到一些具体的标准化协议。WS-Coordination 和 WS-Transaction 补充了 BPEL，提供了定义这些具体的标准化协议的方法。在本章的后面章节中，我们将要讨论 BPEL。在第 10 章中，我们将主要分析 Web Service 事务方面的内容，并将讨论 WS-Coordination 和 WS-Transaction（以及其他的协调和事务协议）。

9.5　服务组合元模型

本节描述了服务组合元模型的集成。服务组合元模型可将应用连接在一起，以及可将应用部署在工作流（BPM）系统之上。服务组合元模型表示了 Web Service 组合中的概念、构建、语义和关系。Web Service 编排和编配语言需要应用工作流定义。服务组合元模型是语言独立的，并使用了基于这些工作流定义的通用构建。通过讨论服务组合元模型，读者将能更容易地理解本章后面介绍的概念和构建。

9.5.1　流模型的理念

复合服务交互规范使用了流模型。流模型描述了可用服务集的使用模式。将这些服务组合起来可提供某一业务目标所需的功能[Leymann 2000]。流模型描述了如何组合活动（作为 Web Service 操作实施），指定了被执行的步骤的顺序，此外流模型还指定了决策点（在这些节点上步骤可以必须执行，也可能无须一定要执行），以及指定了所涉及的步骤之间的数据项的传递。

图 9.4 显示了一个简单的流模型的例子，该模型针对了图 9.2 所描述的订单管理流程。该图由一系列的活动组成，这些活动将以一定的顺序执行。可将活动视为流程中的一个步骤，它将完成一个具体的功能。通常将活动作为 Web Service 的操作实施。流程可描述为一个有向无环图（DAG），而活动则可表示为有向无环图上的节点。这意味着，在流程的控制结构中不允许出现环。该图也显示了如何在各个活动间传递数据项。这些包含了图中的边。例如，客户细节可传递到核查客户信用的活动中。如图所示，流模型描述的关键部分是活动；控制流规范描述了这些活动和决策点的顺序；相关数据流规范描述了数据在活动间的传递。活动、控制链接、数据链接分别表示了服务组合元模型中的三个概念。在本章的后面章节中，我们将要讨论这三个概念。

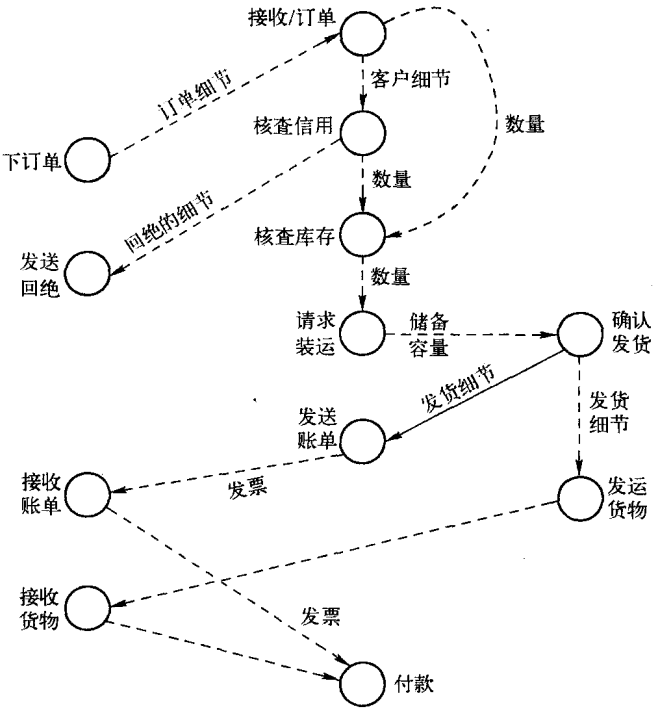


图 9.4 订单管理流程样例

在图 9.4 所示的样例中，核查客户信用状况的活动可如下定义：

```
<activity name="CheckCreditWorthiness">
        <input name="ClientDetails" message="tns:Client"/>
        <output name="Result" message="tns:CreditWorthiness"/>
        <implement> … </implement>
</activity>
```

该代码片段规定了：CheckCreditWorthiness 活动将接收 Client 类型的 WSDL 消息，并生成 CreditWorthiness 类型的 WSDL 消息作为输出结果。

上面代码片段的语法受到了 Web Service 流语言(WSFL)[Leymann 2001]的启发，WSFL 是 BPEL 之前的一个规范。有趣的是，上面的代码片段对活动规范和它的实现进行了区分。活动的规范定义了如何将活动嵌入到流程中。在流程的执行中，当执行到特定活动时，活动的实现描述了被调用的实际操作。服务提供者即可以在内部也可以在外部定义活动的实现操作。因此，Java 或 EJB 中的核查方法可以提供实现。

在图 9.4 中，活动是通过控制链接进行互连的。控制链接是一条有向边，它规定了将要执行的活动的顺序。这表示活动间的可能的控制流，这些活动组成了业务流程。以某一顺序执行两个活动必须遵循这两个活动之间的逻辑依赖性。假如不存在这类依赖性，则可以同时执行这些活动，因此可加速流的执行。在两个活动 A_1 和 A_2 之间，控制链接是一个顺序关系，A_1 和 A_2 规定了两个活动的执行顺序，例如 A_1 必须在 A_2 前面。离开一个特定活动 A 的所有控制链接的端点表示了活动 A 的可能的后继活动 A_1，A_2，…，A_n。

控制链接从它的源活动指向它的目标活动。换句话说，从一个活动到它的可能的后继活动。紧接着，确定实际的控制流的变迁条件“控制”这样的一条边。变迁条件确定了在业

务流程中需要完成活动 A_1，A_2，…，A_n 中的哪一个。变迁条件是与控制链接相关的一个断言表达式[Leymann 2000]。表达式的形式参数可以引用控制链接源端前面的活动所产生的消息。当活动 A 完成时，假如源自该活动的控制链接的变迁条件为真，则该活动将后继这些控制链接。这些活动的集合称为 A 的"实际的后继活动"，而全集 A_1，A_2，…，A_n 则称为活动 A 的"可能的后继活动"。例如，活动 CheckCreditWorthiness 可以选择任何一个后继活动 SendRejection 或 CheckInventory。所选择的活动称为 CheckCreditWorthiness 的实际的后继活动，而活动 SendRejection 和 CheckInventory 则称为 CheckCreditWorthiness 的可能的后继活动。在两个不同的活动之间最多允许一个控制链接，并且正如本节前面所指出的，所产生的有向图必须是无环的。

图 9.4 中的流模型的片段的控制结节如图 9.5 所示。图 9.5 显示了存在的变迁条件，例如确定一个已有客户的信用卡历史的决策点，变迁条件将在运行时进行评估。随着评估结果的不同，活动 CheckCreditWorthiness 的后续活动既可能是活动 SendRejection，也可能是活动 CheckInventory。

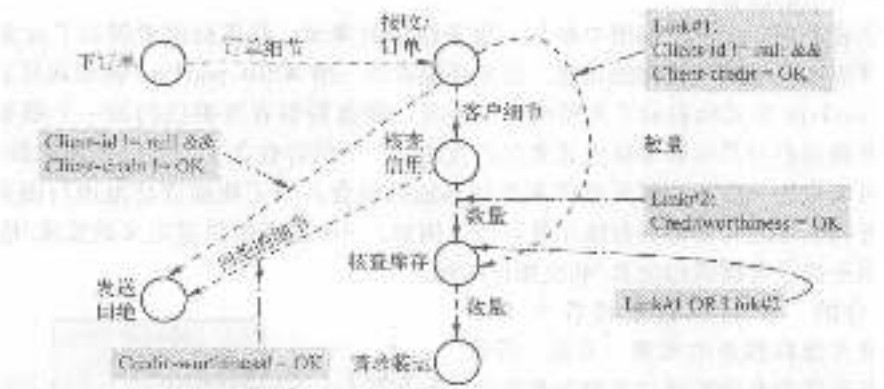


图 9.5　图 9.4 中的流模型的变迁条件和控制链接

以下的代码片段表明了：只有当活动 RequestShipment 的所有进来的控制链接全部已经启动并且仅当其中之一（link1 或 link2）为真时，才会启动活动 RequestShipment。

```
<activity name="RequestShipment">
        <input name="CustomerOrdered" message="tns:Client"/>
        <output name="ReserveCapacity" message="tns:Shipment"/>
        <implement> ... </implement>
  <join condition = "link1" OR "link2"/>
</activity>
<controlLink name="link1"
        source="ReceiveOrder" target="CheckInventory"
        transitionCondition="client_id!= null && client_credit=OK" />

<controlLink name="link2"
        source="CheckCreditWorthiness" target="CheckInventory"
        transitionCondition="credit_worthiness=OK" />
```

流模型的数据流部分规定了一个特定活动的一个（或多个）后续活动如何使用这个特定活动所生成的消息。服务组合元模型中的数据链接规定了源活动向它的目标活动（一个或多个）传递数据项。仅当通过一个（有向的）数据链接路径，能够从控制链接源抵达数据链接的目标时，才指定一个数据链接。总的来说，数据流是独立在控制流上的。这样可避免一

些错误的发生。例如，若要使用尚未生成的数据时，则活动可能从其他多个活动中接收数据项（数据聚合）。

下面的代码片段表示了图 9.5 中的活动 ReceiveOrder 与活动 CheckCreditWorthiness 之间的数据流。

```
<dataLink source="ReceiveOrder" target="CheckCreditWorthiness">
        <map sourceMessage="tns:orderDetails"
         targetMessage="tns:CreditWorthiness"
         sourcePart="data" targetPart="record"
</dataLink>
```

9.5.2 Web Service 的组合

在前一个章节内，我们主要介绍了组合服务元模型的主要组成部分，而没有考虑到流活动的具体实现。在本节中，我们将主要讨论协调 Web Service 的流的组成，其中每一个 Web Service 实现了流程中的单个活动。部分地基于 Web Service 流语言，本节将对 Web Service 组成进行高层描述[Leymann 2000]。这样做的目的是向读者提供有关服务组合特性的直观认识，以便读者可以更好地理解 9.7 节中的内容。

服务提供者是那些服务流模型中参与了服务组合的单元。服务提供者表示了业务流程中的业务合作伙伴需要向流模型提供的功能。服务提供者以一组 WSDL portType 的形式显示了其公共接口。这组 portType 形式地表示了流模型中所涉及的服务提供者所提供的每一个服务。这基本上定义了服务提供者与其他服务提供者交互的方法。一个组合包含了一系列互连的服务提供者，该服务组合可以作为一个新的服务提供者参加其他的组合。为了能够成功地进行服务组合，进行交互的服务提供者之间需要具有操作兼容性。例如，一个服务提供者定义的要求/应答操作需要与另一个服务提供者提供的请求/响应操作匹配。

服务组合的一个目标就是将各个 Web Service 作为业务流程服务的实现。为此，活动可以引用服务提供者类型的端口类型的操作来指定在运行时需要哪一类服务来完成这个业务任务。一个服务可以表示该业务任务。服务提供者类型的端口类型定义了流模型的外部接口。例如，图 9.6 显示了一个流程，其中一个服务实现了活动 A，该服务实现了端口类型 pt 的操作 $operation_1$。在运行时，当导航到到 A 时，选择一个具体的端口，该端口提供了端口类型 pt 和操作 $operation_1$ 的一个实现。可使用一个相应的绑定来实际调用这个实现。

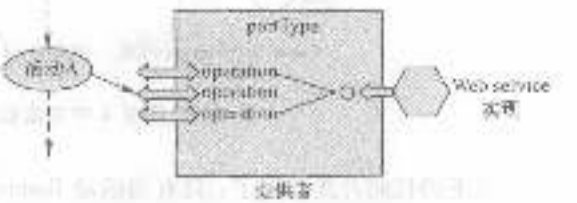


图 9.6 服务提供者和端口类型

流模型规定了特定类型的服务实例的用法，而不是服务本身的用法。多个流可以同时使用一个 Web Service。然而对于一个特定类型的服务，一个流模型可以使用多个服务提供者。图 9.4 显示的流模型的更新版本如图 9.7 所示，其中两个活动已经外包给业务合作伙伴（服务提供者）。

一个服务提供者承担了图 9.7 中显示的物流提供者的角色。下面的代码片段显示了：对于订单管理流，服务提供者如何通过调用 Web Service 执行活动。元模型中的服务提供者类型标识了业务合作伙伴可用的外部接口。这个接口可以包含多个端口类型和多个操作。在下面的例子中，LogisticsProvider 包含了两个操作，这两个操作是由两个端口类型提供的。

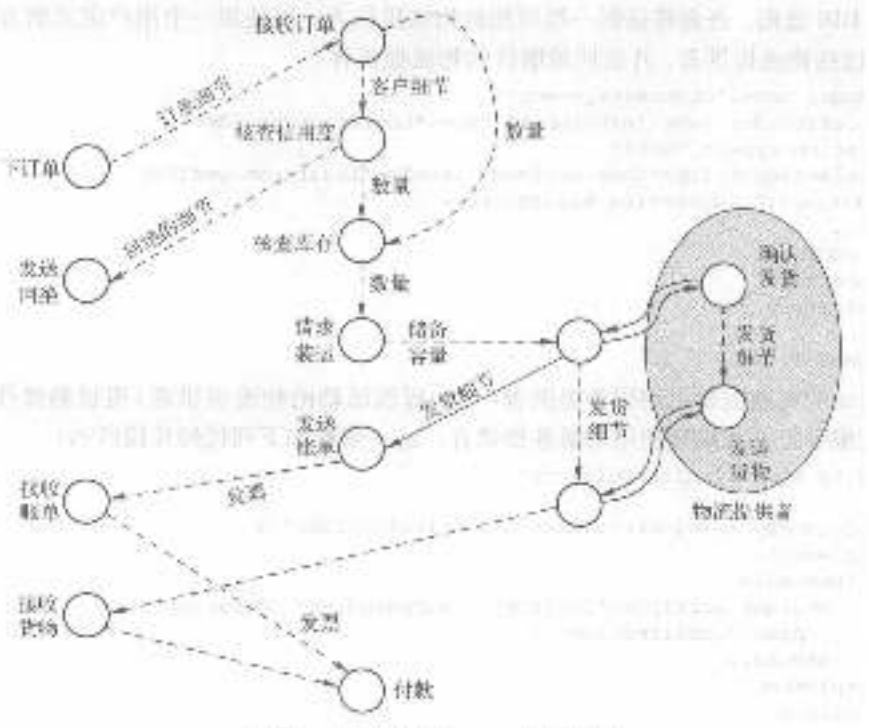


图 9.7　使用 Web Service 的轨模型

```
<flowModel name="OrderManagement">
<serviceProvider name="myShipper" type="LogisticsProvider"/>
...
</flowModel>

<serviceProviderType name="LogisticsProvider">
   <portType name="ConfirmDeliveryHandler">
      <operation name="confirmation"/>
   </portType>
   <portType name="DispatchGoodsHandler">
      <operation name="delivery"/>
   </portType>
</serviceProviderType>
```

对于实际的业务合作伙伴，绑定服务提供者提供了所需的服务。绑定服务提供者既可以静态地发生作用，也可以动态地发生作用。当进行静态绑定时，我们简单地定位实际的业务合作伙伴。通过直接引用 WSDL 服务，实际的业务合作伙伴充当了一个具体的服务提供者，而 WSDL 服务则提供了所需的操作。静态绑定的 WSDL 服务实现了接口，该接口是由所需类型的服务提供者定义的。静态绑定如下面的代码片段所示：

```
<flowModel name="OrderManagement">
<serviceProvider name="myShipper" type="logisticsProvider">
  <location type = "static" service="shippingLogisticsService"/>
</serviceProvider>
...
</flowModel>
```

另一种方式是，按照一些指定的标准（如性能、价格等）查找 UDDI 目录，以便发现一个合适的所需类型的服务提供者，从而动态地绑定到一个服务提供者。随后，（按照一些 QoS 标准）选择最合适的服务并进行绑定。下面的例子显示了如何动态地定位一个物流提供者。流引擎执行

了指定的 UDDI 查询，查询将返回一组可能的物流提供者。可使用一个用户定义的方法 selectionPolicy 过滤这些物流提供者，并返回最廉价的物流提供者。

```
<flowModel name="OrderManagement">
<serviceProvider name="myShipper" type="LogisticsProvider">
  <location type = "UDDI"
     selectionPolicy="user-defined" invoke="leastcost.wsdl"/>
   <uddi-api:find-service businessKey= "..." >
     ...
   </ uddi-api>
  </location>
</serviceProvider>
...
</flowModel>
```

外包活动的实施需要引用服务提供者(例如提供活动的物流提供者)提供的操作。为此，首先必须指定服务组合元模型使用的服务提供者。这一情形如下列代码片段所示：

```
<activity name="ConfirmDelivery">
  ...
  <performedBy serviceProvider="LogisticsProvider"/>
   <implement>
     <internal>
        <target portType="ConfirmDeliveryHandler" operation
          name="confirmation"/>
     </internal>
   </implement>
</activity >
```

上面的例子假定“内部”提供了操作的实现，因此不需要访问“外部”的服务提供者。上面例子所示的 confirmation 操作是 ConfirmDeliveryHandlerportType 的一部分，并且该操作实现了流模型的 ConfirmDelivery 操作。

一旦根据 Web Service 实现了一个流程(诸如订单管理)的流模型，则应用就能使用该流模型。图 9.8 显示了如何将订单管理流程外部化为一个 Web Service，以便客户端接口可以连接到它上面。流模型接口包含粗箭号所示的 4 个操作，其中三个操作是向外的，一个操作是向内的。首先，使用 Web Service 接口的客户端发送一个关于订购生产零件(图中的货物)的请求，然后或者被回绝，或者收到一张发票然后收到实际交付的货物。下面的代码片段说明了这一情形。

```
<portType name="OrderManagementHandler">
   <operation name="acquireGoods">
     <input name="theOrder" message="tns:Order"/>
   </operation>
   <operation name="reject">
     <output name="theRejection" message="tns:RejectionNotice"/>
   </operation>
   <operation name="invoice">
     <output name="theInvoice" message="tns:Invoice"/>
   </operation>
  ...
</portType>
<serviceProviderType name="OrderManagementProvider">
   <portType name="OrderManagementHandler">
</serviceProviderType>
```

根据上面的定义可以定义一个流模型，它通过下列声明表示了特定服务提供者类型(OrderManagementProvider)的一个服务提供者。

```
<FlowNode1 name="OrderManagement">
    <ServiceProviderType="OrderManagementProvider"/>
    ...
</FlowNode1>
```

读者现在已经熟悉了最常见的工作流概念和构成，因此可以相对容易地理解 Web Service 编配和组合语言中的概念和构成。

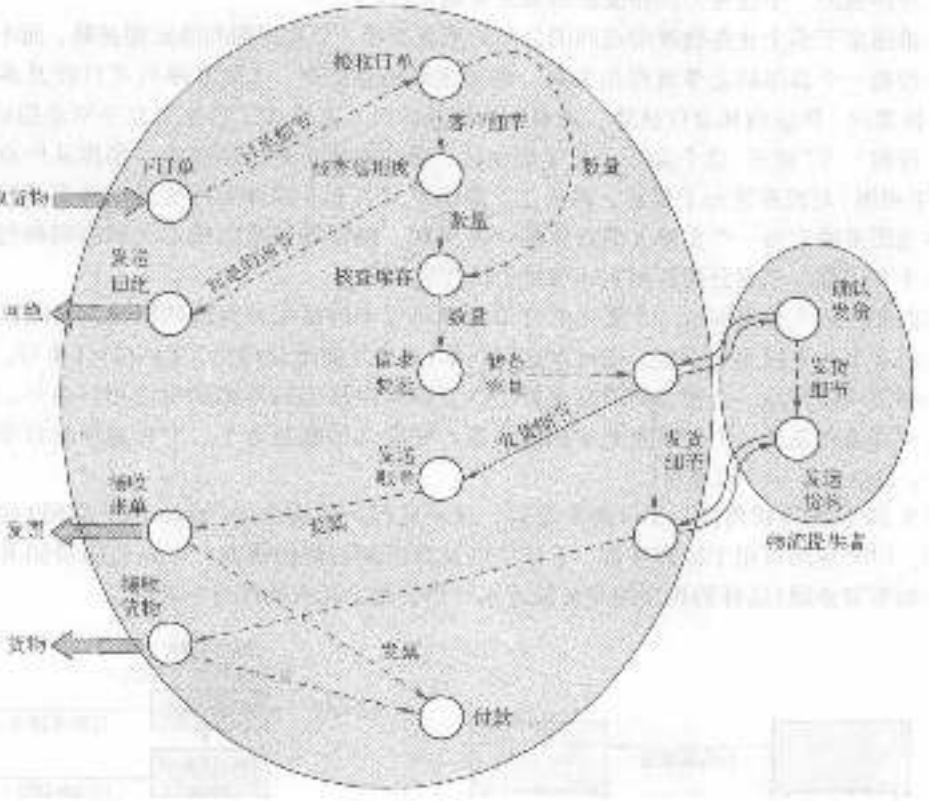


图 9.8　将流程外部化为一个 Web Service

9.6　Web Service 的编配与编排

Web Service 可作为电子商务解决方案的开发手段。当应用和业务流程能够集成它们的复杂的交互时，Web Service 将能发挥最大潜力。Web Service 技术支持协调、基于异步的和面向消息的通信方式并可与业务逻辑进行交互，因此 Web Service 技术提供了一个可行的解决方案。电子商务交互模型通常需要指定 Web Service 间的对等消息交换的序列。有状态的，长时间运行的交互，无论采用同步通信还是异步通信都涉及两个或更多的参与方。在这类交互中，需要依据业务协议（或抽象业务模型）描述业务流程。对于协议中所涉及的各方，业务协议精确地规定它们的相互间的，具体的（公共）消息交换行为，但不会涉及参与各方的内部（私有）实现。它也需要对业务交互中所涉及的各方的实际行为进行建模。为了定义这类业务流程和协议，对于业务流程在交互中所使用的消息交换协议，需要进行形式描述[Bloch 2003]。

编配与编排的比较

在前面的章节中，我们使用了一些可相互替代的术语（诸如"Web Service 组合"和"Web

Service流")描述了流程中的 Web Service 组合。最近，通常使用术语"编配"和"编排"来描述这一现象。在 Web Service 编配和 Web Service 编排之间存在重要的差异[Peltz 2003]，具体如下：

编配描述了 Web Service 在消息层如何进行相互交互，包括业务逻辑以及单个端点控制下的交互的执行顺序。例如在图 9.9 所示的流程中，所涉及的供应商仅有一个。编配针对一个可执行的业务流程，这一流程可生成一个长时间的、有状态的、多步骤的流程模型。通过编配，可以从流程中所涉及的一个业务方的角度控制业务流程交互。

编排通常于多个业务处理端点间的公开的消息交换、交互规则和协定相关联，而不是与由其一方执行的一个具体的业务流程相关联。编排跟踪消息序列，该消息序列可以涉及多方和多个源，包括客户、供应商和合作伙伴。流程中所涉及的每一方描述了它在交互中所承担的角色，并且没有任何一方"拥有"这个会话。与编配相比，编排本质上更具协作性。编排从所有各方的角度(公用视图)对流程进行了描述。本质上，编排定义了业务实体间的交互的共享状态。可以使用公共视图来确定每一个实体上的具体的部署实现。编排能够清晰地定义和协商参与协作的规则。每个实体都可根据公用视图实现编排中的相应部分。

我们使用图 9.9 来例示一个简化的订单管理流程中的编配和编排的概念。该图显示了一个典型的包含定购单的业务流程。通过发送定购单(PO)并使用订购单文档中的订单号，制造商可以开始和供应商开始一个相关的消息交换。供应商可以在定购单确认中使用订单号。供应商后面可以向制造商发送一个包含发票文档的消息。发票文档既包含了一个和最初的订单号相关的订单号，也包含了一个发票号。

图 9.10 从编排视角显示了订购单流程。这一流程从后端应用(诸如 ERP 应用)的角度进行表示的。ERP 应用可用于核查库存、下订单以及组织制造商的资源。可以使用诸如 BPEL(将要在下面的章节介绍)这样的典型的业务流程执行语言指定私有的制造商流程。



图 9.9 一个简化的业务流程



图 9.10 从编配角度看定购单

图 9.11 从编排角度显示了订购单，图中虚线环绕起来的部分表示了公开的消息交换，可以使用诸如 WS-CDL(参见 9.8 节)这样的服务编排规范语言指定具体的编排。

最后，图 9.12 显示了制造商和供应商集成它们的业务流程。该图假定两个公司的业务分析师对于流程协作中所涉及的规则和流程达成一致意见。使用图形用户界面和一个可充当协作基础的工具，制造商和供应商可它们之间的交互达成一致，并可生成一个 WS-CDL 表示。对于制造商和供应商，可以使用 WS-CDL 表示来生成一个 BPEL 工作流模板。这两个 BPEL 工作流模板反应了业务协定。

在下面的章节，我们将首先重点讨论 Web Service 的业务流程集成语言(简写为 BPEL)。BPEL 是一个标准的行业规范，明确地指定了基于 Web Service 的编排。然后将概述一些重要的 WS-CDL 元素。



图 9.11　从编排角度看订购单



图 9.12　组合编排和编定

9.7　业务流程执行语言(BPEL)

Web Service 编配规范采纳了 WSDL 中的一些概念。WSDL 本质上允许 Web Service 静态接口的定义。WSDL portType 的交互模型是无状态的、静态的，在被定义的接口级没有任何关联的交互。此外，WSDL 从服务(提供者)的角度描述了接口，因此它成为一种客户/服务器交互模型。协作流程模型通常涉及客户/服务器类型的交互和对等(P2P)类型的交互，交互中含有涉及两方或多方的、长时间运行的、有状态的会话，而 WSDL 并不能胜任传送这种类型的会话。因此，Web Service 编配规范在 WSDL 的基础上进一步扩充了其功能。

最近出现的 BPEL 标准可定义和管理包含协作 Web Service 的业务流程活动和业务交互协议。BPEL 是一个针对业务流程的形式规范和业务交互协议的、基于 XML 的流语言，据此，BPEL 扩展了 Web Service 交互模型并使得它能够支持复合业务流程和事务。企业能够描述包括多个组织的复合流程(诸如订单处理、储存客户管理和索赔处理等)，并可在其他企业的系统中执行相同的业务流程。

服务组合模型能够提供灵活的集成、递归组合、组合的分离、有状态的会话和生命周期管理、易恢复性[Weerawarana 2005]。作为服务组合(编配)语言，BPEL 提供提供了一些特性，可使于基于 Web Service 的业务流程的建模和执行。这些特性包括：

- 建模业务流程协作(通过 < partnerLink >)。
- 建模业务流程的执行控制(通过使用自包含的块以及支持有向图表示的过渡结构语言)。
- 对抽象定义与具体绑定进行分离(通过端点引用、静态或动态地选择合作伙伴服务)。
- 参与者的角色和角色间的关系的表示(通过 < partnerLinkType >)。
- 补偿支持(通过 < scope > 机制)。
- 流程的再生和同步(通过 < pick > 和 < receive > 活动)。
- 事件处理(通过使用事件处理程序)。

也可以对 BPEL 进行扩展，从而提供其他重要的组合语言的特性，诸如对 Web Service 策略的支持、安全性和可靠的消息传送。

在本节中，我们描述了 BPEL 的最主要的特性和构成。我们的目的是要透彻地理解 BPEL 的概念和特性，而不是提供 BPEL 和它的构成的详细教程。有关该语言的更详细的信息可参考文献[Andrews 2003]。

9.7.1　BPEL 的结构

BPEL 流程是一个类似流图的表示，指定了流程的步骤以及流程的入口点。流程位于 WSDL

层之上。WSDL 定义了所允许的具体操作，而 BPEL 则定义了如何对这些操作进行排序[Curbera 2003]。BPEL 的作用是：通过具有控制语言构成的“流程 - 集成 - 类型”机制，将一组已有的服务定义为一个新的 Web Service。入口点对应于外部的 WSDL 客户端，该 WSDL 客户端可调用复杂的 BPEL 服务的接口上的“只输入(请求)”操作或“输入/输出(请求/响应)”操作。图 1.7 描述了 BPEL 如何与其他的 Web Service 标准进行关联。

使用 WSDL 描述的服务间的对等交互是 BPEL 流程模型的核心。WSDL 建模了流程和 Web Service 合作伙伴。BPEL 使用 WSDL 指定了在业务流程将要发生的活动，并描述了业务流程所提供的 Web Service。BPEL 可按下面三种方式使用 WSDL[Pelz 2003]：

(1) 使用 WSDL 将每一个 BPEL 流程暴露为一个 Web Service。WSDL 描述了这个流程的公开的入口点和出口点。

(2) 在 BPEL 流程中，使用 WSDL 数据类型描述了在请求之间进行传送的消息。

(3) 可以使用 WSDL 引用业务流程所需的外部服务。

BPEL 提供了相关机制，可将 WSDL 表示的抽象接口编为与实现无关的、平台独立的服务组合。BPEL 业务流程的定义也遵循 WSDL 协议，可将抽象服务接口和服务实现进行严格的分离。尤其，BPEL 流程表示了流程中所涉及的各方，以及各方之间按照抽象的 WSDL 接口(依靠 < portType > 和 < operation >)进行的交互，并且并不引用流程实例所使用的实际的服务(绑定信息和地址信息)。可将进行交互的流程以 WSDL 服务的形式进行建模，并可将服务实现动态地绑定到 BPEL 组合的合作伙伴中，且不会影响到组合的定义。在 BPEL 中指定的业务流程是完全可执行的、可移植的脚本。在遵循 BPEL 的环境中，业务流程引擎可解释这些脚本。

BPEL 区分了 5 个主要的部分：消息流、控制流、数据流、流程编配、故障和异常处理。这些部分如清单 9.1 所示。图 9.13 则显示了 BPEL 数据结构如何通过 UML 元模型进行相互连接。

清单 9.1 BPEL 流程的结构

```
<process name="PurchaseOrderProcess" ... >
  <!- Roles played by actual process participants at endpoints of
   an interaction -->
  <partnerLinks> ... </partnerLinks >

  <!- Data used by the process -->
  <variables> ... </variables >

  <!- Supports asynchronous interactions -->
  <correlationSets> ... </correlationSets>

  <!- Activities that the process performs -->
   (activities)*

  <!-Exception handling: Alternate execution path to deal with
     faulty situations -->
  <faultHandlers> ... </faultHandlers>

  <!-Code that is executed when an action is "undone" -->
  <compensationHandlers> ... </compensationHandlers>

  <!-Handling of concurrent events -->
  <eventHandlers> ... </eventHandlers>
</process>
```

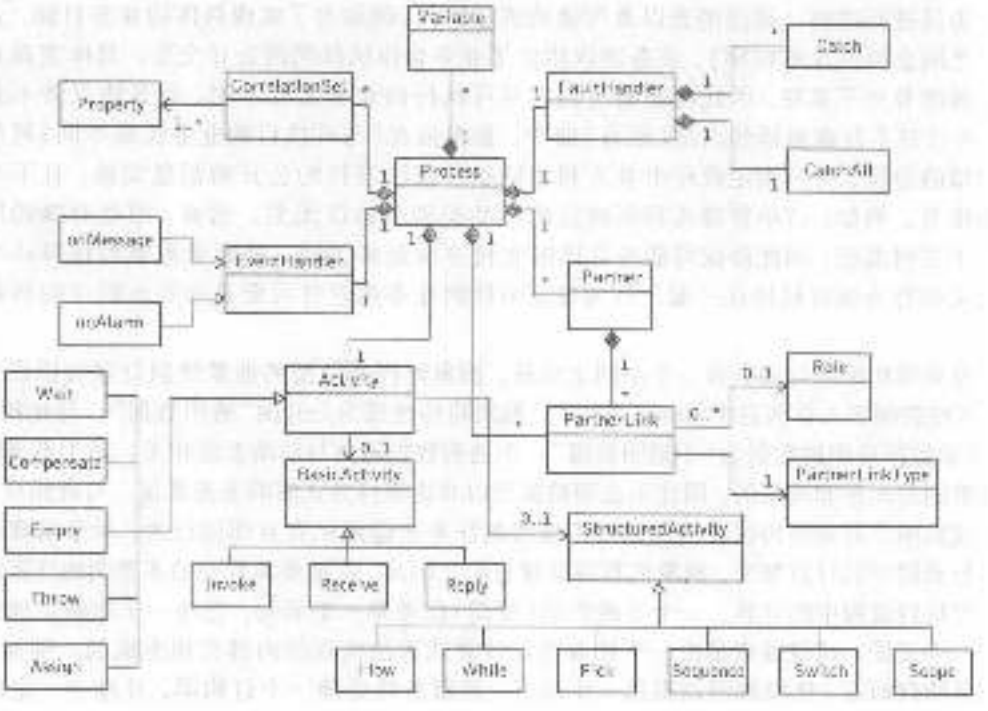


图 9.13　BPEL 1.1 UML 元模型

基本活动处理了 BPEL 的消息流部分。这些基本活动包括：调用一些 Web Service 上的操作、等待一些外部客户端调用流程操作、生成和入/输出操作的响应。BPEL 的控制流部分是一个混合模型，该模型主要基于块结构化定义。为了进行同步，该模型能够定义选择性的状态转换控制流。BPEL 数据流部分包括了一些变量，构成业务流程状态的消息可以包含在这些变量中，这些消息通常是从合作伙伴那儿接收到的，或者是将要发送给合作伙伴的。控制与流程相关的状态所需的数据也可包含在变量，并且不与合作伙伴交换。变量都有自身的作用域，在变量的作用域内，变量名必须具有唯一性。BPEL 的流程编配部分使用合作伙伴链接来建立对等的合作伙伴关系。最后，BPEL 的故障和异常处理部分将处理调用服务时所出现的错误，并处理工作单元的补偿以及 BPEL 执行过程中的异常。

在下面的章节中，我们首先将定义 BPEL 中的抽象流程和可执行流程，然后将分别概述 BPEL 1.1 样例中的 5 个主要部分。

1. 抽象流程和可执行流程

BPEL 对业务流程的两个层次进行了描述：抽象的业务流程和可执行的业务流程。抽象流程规定了 Web Service 之间的外部消息交换，并且不包含业务流程的任何内部细节。通常使用这类流程对 Web Service 之间的公开的消息交互（业务协议）进行建模，且无须暴露这些服务的内部业务逻辑，因此这种流程是不可执行的。相反，可执行流程定义了外部消息交换以及业务逻辑的每个的内部细节，因此它是可执行的。可执行流程包含了整个业务流程中的参与者的所有实际交互和行为，尤其建模了一个私有的工作流。可执行的流程包含了流程的所有细节，包括对流程状态的完整描述以及如何处理流程的完整描述。我们可将抽象业务流程视为可执行流程的投影，对于定义抽象流程和可执行流程，BPEL 提供了同样的语言构成。

对于抽象流程和可执行流程，有两个主要的不同之处。首先，在 BPEL 中可将抽象流程作

为业务协议进行建模。通过消息以及可能的消息序列(例如为了实现具体的业务目标，业务合作伙伴之间交换消息的顺序)，业务协议指定了业务合作伙伴间的公开交互。具体实现这些协议的内部细节并不重要，因此将忽略它们。与可执行的业务流程不同，业务协议并不是可执行的，并且并不显露流程的内部(私有)细节。抽象流程(与可执行的业务流程不同)可用于指定所期望的协议，并可指定流程中各方利用这些协议所进行的公开的消息交换，且不会涉及太多的细节。例如，订单管理流程所确定的产品类型与协议无关。然而，消息交换的序列可能依赖于支付类型，因此协议可能与具体的支付方法是相关的。抽象业务流程将 Web Service 接口定义和行为规范链接在一起。行为规范可控制业务角色并可定义业务流程中的所有各方的行为。

抽象流程和可执行流程另一个不同之处是，抽象流程按所需的抽象级别处理与协议相关的数据，这些数据嵌入在消息中[Bloch 2003]。抽象将特性视为公开的"透明数据"，与此相反，内部/私有函数所使用的数据为"不透明数据"。不透明数据通常与后端系统相关。由于不透明数据影响决策的方式是非透明的，因此不透明数据仅以非决策性方式影响业务协议。与此相反，透明数据直接影响公开业务协议。通过执行不透明的任务来隐藏私有方面的行为，对于出现在相应的可执行流程中的计算细节，抽象流程将忽视它们。因此，在抽象流程中的不透明的任务将取代对应的可执行流程中的计算。一个不透明的(变量)任务是一条语句，它将一个非确定地选择的值赋给一个变量。该值通常是由一个涉及特定计算或算法实现的内部实现生成的。例如，就定购单业务协议而言，供应商可以提供一个服务，该服务将处理一个订购单，并基于一定的标准(诸如是否有货、购买者的信用状况等)，接受订购请求或者拒绝接受请求。这个决策流程是不透明的。然决策结果可以通过行为反映出来，或者由外部业务协议进行显示。换句话说，在销售商服务的行为中，该协议充当了一个转换器，流程中的决策分支的选择是非确定的。

图 9.14 显示了一个抽象流程，该抽象流程对应于图 9.2 中的订单管理流程图。在图 9.14 中，制造商(客户)和供应商有两个不同的角色，每一个都有它自己的端口类型和操作。通常将它们在接口层的关系的结构建模为一个公开的消息交换。

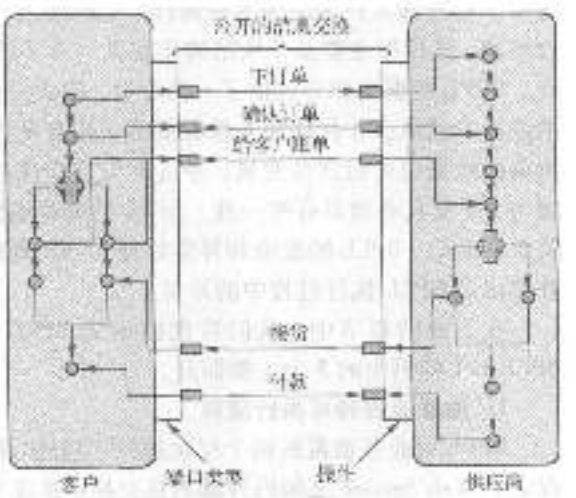


图 9.14 对应于图 9.2 中流程图的抽象流程

2. 消息流

BPEL 的消息流部分发送和接收消息，以便于流程(Web Service)实例能够与其他的 Web Service 进行通信。基本的活动将处理消息流部分。这些基本活动包括调用一些 Web Service 上的操作(<invoke>)、等待一些外部客户调用以输入流程操作(<receive>)、生成输入/输出操作的响应(<reply>)。作为一个针对 Web Service 组合的语言，通过调用其他的服务以及接收来自客户端的调用，BPEL 流程进行交互。使用 <invoke> 活动可调用其他的服务，而使用 <receive> 和 <reply> 活动则可接收并响应客户端的调用。下面将简要描述这三个活动。

BPEL 调用其他的服务合作伙伴。合作伙伴是一个流程所调用的 Web Service，或者是调用流程的任何客户端。本质上，合作伙伴是对合作者的 Web Service 的 WSDL portType 描述的映射[Chatterjee 2004]。BPEL 流程使用 <partnerLink> 与每一个合作伙伴进行交互。合作伙伴链接后

一个类型化的连接器实例。该连接器实例是一个特定的流程所提供的，或者是链接到另一端的合作伙伴所需要的[Weerawarana 2005]。< partnerLink > 元素表示了这样的链接是一个会话接口而不是反映了业务关系。换句话说，可将 < partnerLink > 视为和合作伙伴之间建立对等会话的信道。如图 9.14 所示，BPEL 消息流部分的 < invoke > 、< receive > 和 < reply > 活动指定了相关的 < partnerLink > - < portType > 操作和变量[Weerawarana 2005]。有关 < partnerLink > 的更详细的信息参见随后的"流程编配"一节。

流程使用 < invoke > 调用 Web Service。基于 < invoke > ，流程既可以调用合作伙伴 Web Service 所提供的 < portType > 上的同步操作(请求/响应)或异步操作(单向)。< partnerLink > 、WSDL < portType > 和操作标识了该 Web Service。除了 < portType > 、合作伙伴和操作，对于被调用的操作的输入和输出，< invoke > 元素指定了输入变量和输出变量。同步的 Web Service 需要输入变量和输出变量。假如调用异步操作，仅需要输入变量。

通过 < invoke > 活动，流程能够调用合作伙伴的 Web Service 的操作，因此 < receive > 活动指定了一个 Web Service 操作，这个操作是由 Web Service 合作伙伴所使用的流程来实现的。合作伙伴通过调用服务来触发流程的执行。在 BPEL 中，流程通常以 < receive > 活动或 < pick > 活动开始。这意味着流程必须作为一个特定类型的服务开始调用。因此，< receive > 活动是 WSDL 操作的映射。类似于 < invoke > 活动，< receive > 使用 < partnerLink > 、WSDL < portType > 和 < operation > 来标识一个具体的合作伙伴期望调用的服务。调用者所传递的参数将被绑定到指定的变量。因此，< receive > 直接表示了一个业务流程逻辑开始之处，该处将接受请求信息。使用 < receive > 活动可接收被同步或异步发送的消息。

通过 < receive > 活动，业务流程可接收一个消息，然后可通过 < reply > 活动发送该消息的响应消息。< reply > 活动将响应传送到 Web Service 客户端。< receive > 和 < reply > 将组合形成了流程的 WSDL < portType > 上的同步请求/响应操作。< receive > 活动接收一个 Web Service 调用，并将该调用作为输入，< reply > 则传回相应的输出。在 < receive > 活动和 < reply > 活动之间，流程可进行任何所需的计算。< reply > 活动必须匹配 < partnerLink > 、< portType > 以及 < receive > 活动的操作属性，并且 < reply > 活动的变量属性指定了输出。异步流程并不使用 < reply > 活动。假如这类活动需要向客户端发送回复，则使用 < invoke > 活动调用客户端的 < portType > 上的操作。在这种情况下将不需要输出变量。图 9.15 显示了同步(请求/响应)和异步(单向)通信模型。该通信模型涉及 < receive > 活动，< reply > 活动和 < invoke > 活动。该图表示了：在异步通信模式中，提供者的异步响应(假如它即将来临)将被称作回调(callback)。

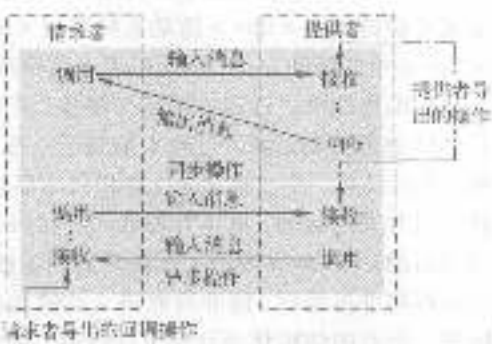


图 9.15　BPEL 中的同步和异步通信模式

为了响应合作伙伴的请求，各流程中可定义多个 < reply > 活动。然而，在任何时候仅会有一个 < reply > 活动相匹配。在运行时，将对合适的 < reply > 活动进行匹配，流程将寻找已做好运行准备并和 < receive > 具有相同的 < portType > 、< operation > 和 < partner > 的活动。

3. 控制流

活动的定义和构成一个特定流程所需的步骤序列是 BPEL 规范的一个重要部分，这需要应用基本活动和结构化活动。

BPEL 包含基本活动和结构化活动。基本活动是和服务进行交互的最简单的形式。它们不是

序列化的，并且包含与服务交互的单个的步骤。基本活动操纵交换数据或者处理执行中出现的异常。例如，基本活动将处理消息请求的接收与回复，以及调用外部服务。在这一典型场景中，BPEL 流程将接收一个消息。然后，该流程可以调用一系列的外部服务来收集更多的数据，并以一定的方式响应请求者。诸如 < receive > 、< reply > 和 < invoke > 等 BPEL 活动都是基本活动，它们使得业务流程能够和服务交换消息。其他的基本活动包括异常处理活动和状态管理活动。除了基本活动，BPEL 使用结构化活动（包括 < sequence > 、< switch > 、< while > 、< pick > 和 < flow > ）来管理整个的流程控制流、指定活动的执行顺序。

结构化活动规定了哪些活动将要串行化地运行（通过 < sequence > 活动）、哪些活动可并行地运行（通过 < flow > 活动）。通过将基本活动结构化，可以创建业务流程。结构化活动表示了控制模式、数据流、故障处理、外部事件处理、流程实例之间的消息交换的协调。结构化活动可被视为 BPEL 的底层编程逻辑。结构化活动描述了如何通过基本活动结构化来创建业务流程。

BPEL 的控制流部分能够定义选择性的状态转换控制流来实现同步。BPEL 的控制流部分包括定义活动的顺序（ < sequence > ）、使得活动能够并行运行（ < flow > ）、活动分支（ < switch > ）、定义迭代（ < while > ），以及基于外部事件（ < pick > ）执行几个可选路径之一（非决定性选择）。

< sequence > 活动包含一个或多个必须串行化地执行的活动，这些活动的执行顺序与在 < sequence > 元素中的出现顺序必须一致。当 < sequence > 中的最终一个活动完成时，整个 < sequence > 活动本身也完成了。

< flow > 中的活动能够并行运行，它们没有依赖性。< flow > 提供了并发性和同步化，也可定义守护链接。< flow > 活动支持通过 < links > 连接的活动集（包括其他的流活动）的定义。对于 < flow > 中的活动，< links > 可表示这些活动间的同步化依赖性，使得流程中的其他部分具有并行执行的可能性。在流中可定义链接，并可使用它将一个源活动链接到一个目标活动，源活动和目标活动都必须是流的逻辑上的孩子。在 < flow > 活动中声明的每一个链接都必须有且仅有一个源、有且仅有一个目标，并且源和目标都必须是流中的一个活动。BPEL 中的 < flow > 活动可以创建一组并发的活动，这些并发活动直接嵌套在 < flow > 中。对于直接嵌套或简介嵌套在 < flow > 中的活动，< flow > 表示了活动间的同步依赖性。当 < flow > 活动开始时，< flow > 活动中的所有活动都将准备运行，除非有些进入的链接还没有被评估。在活动并发执行过程中，可以指定执行顺序。所有的结构化活动都可进行递归组合。

在 < flow > 活动执行的一开始，所有的链接都是不活跃的，并且仅有那些没有依赖性的活动才能执行。< link > 可以和一个变迁条件相关联。变迁条件是一个谓词表达式，可以评估流程中的不同数据变量的值。正如在 9.5.1 节中所讨论的，变迁条件与每一个控制链接相关联。当链接的源活动完成时，可对变迁条件进行评估。一旦完成了活动的执行，每一个活动都可以评估它的变迁条件，从而确定向外的链接的状态。在该活动完成之前，这些向外的链接都是不活跃的。一旦一个活动的所有进入链接都是活跃的（它们各自的变迁条件都被赋予了一个评估结果“真”或“假”，参见图 9.5），将评估活动的连接条件。连接条件是一个与链接的源活动关联的布尔表达式。连接条件基于指向它的链接（路径）的变迁条件的值确定是否将要执行一个活动。连接条件评估每一个进入的链接（可以有多个链接，并且每一个链接都必须有一个布尔值），并确定是否将要执行该活动。例如，一个连接条件可以规定：在流程中的一个具体活动继续执行之前，该活动的所有进入路径都已经成功完成，或者仅需一条路径成功完成。

< switch > 活动的功能比较类似传统编程语言（例如 C 或者 Java）中的 switch 语句。< case >

元素可以定义一个或多个条件分支的有序列表，并可跟随一个可选的 < otherwise > 元素。每一个 < case > 分支指定了一个布尔 XPath 表达式，并且表达式可以按照它们的出现顺序进行估值。< switch > 活动可以选择工作流中的不同分支。分支的选择是基于 < case > 元素中定义的条件。假如这些条件都没有满足，则可以指定一个特定的分支执行。当所选择的分支执行完毕后，switch 活动也就完成了。

在 < while > 活动中，通过嵌套可重复执行活动。在 BPEL 中，< while > 活动对 XPath 表达式中定义的条件进行估值。假如条件的估值为真，将可迭代执行所包含的活动。

< pick > 活动类似于异步事件驱动的 < switch > 活动，它包含一组事件处理程序[Duftler 2002]、[Chatterjee 2004]。< pick > 活动是事件/活动形式的一组分支。基于与分支相关联的事件的出现情况，可以选择一个分支(也仅可以是一个分支)执行。在 < pick > 活动已经接受一个事件进行处理后，其他的事件将会被忽略。事件可以是诸如不同形式的消息的到达等，例如单向或请求/响应操作中的进入的消息，或者是基于定时器的“警告”等。事件处理程序包括警告处理程序，警告处理程序指定了事件的持续时间(相对于当前时间)或者截止时间(未来一个固定的时间点)。消息处理程序(onMessage)等待来自一个特定的 partner、portType 和 operation 三元组的消息。在 < pick > 活动中的一个分支被相关联的事件触发后，当完成该分支的相应的活动后，整个 < pick > 活动也完成了。

每一个 < pick > 活动至少可以包含一个消息处理程序(onMessage 事件)。onMessage 事件规定了何时一个活动将变得活跃，例如收到一个匹配消息。仅有收到事件的第一个事件处理程序才会运行。一旦事件处理程序的活动完成后，< pick > 活动也将完成。正如 < receive > 活动中的方式一样，消息处理程序也能够创建流程实例。因此，< pick > 活动能够提供流程的进入点，其作用非常类似 < receive > 活动。< pick > 活动和 < receive > 活动的主要不同点是：< receive > 活动支持单个的消息，而 < pick > 活动能够基于许多消息开始一个流程。

4. 数据流

BPEL 中的业务流程指定有状态的交互，交互涉及了合作伙伴间的消息交换。业务流程的状态包括了被交换的消息的内容以及一些中间数据，在业务逻辑中将使用这些中间数据，并且在发送到合作伙伴的消息中也将包含这些中间数据。BPEL 使用称作 < variable > 的状态变量来维护业务流程的状态。BPEL 的数据流部分需要信息是具体的数据。此外，借助于数据表达式可以抽取和合并状态数据。最后，状态更新需要赋值。对于 XML 数据类型和 WSDL 消息类型，BPEL 提供了这些特性。

在 BPEL 中，数据 < variable > 指定了特定流程的业务背景。这些是 WSDL 消息的集合。< variable > 所表示的数据对于业务流程(例如进行路由决策或者构造一个需要发送给合作伙伴的消息)的正确执行非常重要。可以使用 < variable > 数据管理跨 Web Service 请求的数据的完整性。Web Service 提供了一种保存消息内容的方式，所保存的消息内容构成了业务流程的状态。这些消息通常是从合作伙伴那里接收来的或者是发送到合作伙伴那里的消息。< variable > 中也能保存那些与流程相关的状态数据，并且合作伙伴也从不会修改这些状态数据[Bloch 2003]。通过 < assign > 语句变量可以修改具体的数据元素。

可以使用 < assign > 语句在变量之间拷贝数据消息(消息、消息的一些部分和服务引用)。BPEL variable 是一个类型化的数据结构，数据结构中存储了与工作流实例相关的消息，从而便于 Web Service 之间的有状态的交互[Chatterjee 2004]。在工作流中，应用状态是被交换的消息的函数，并且这些状态能够存储在变量中。当新的消息抵达时，或者随着计算的执行，将对变量进行初始化，并给变量赋值。依赖赋值的内容的不同，< assign > 活动将出现几条路径中的某一分支

中。在所有赋值活动中，对于赋值的源和目的地，必须维护它们之间的兼容性。因此，仅当<from>(源)和<to>(目的地)两者的参考变量都具有相同的消息类型，或者赋值的两个端点相同时，赋值才是有效的。通过<assign>语句，不仅可以实现数据操纵，而且可以动态绑定到不同的服务实现。

5. 流程编配

在BPEL中，一个企业的业务流程必须能够通过Web Service接口与其他企业的流程进行交互。这需要能够对合作伙伴流程进行建模。WSDL已经在抽象层次上和实体层次上描述了合作伙伴所提供的服务的功能。与合作伙伴的业务流程的关系通常是对等的，需要服务层的双向依赖[Andrew 2003]。换句话说，合作伙伴既代表了业务流程所提供的服务的消费者，又代表了业务流程的服务提供者。当交互基于异步消息传送而不是远程过程调用时，尤其是这样。在基于Web Service的工作流中，BPEL提供了相关方法，可通过合作伙伴链接以及端点引用来获悉业务合作伙伴所承担的角色。

BPEL的流程组合(编配)部分使用<partnerLink>指定每一方的角色和每一方所提供的(抽象)接口，从而建立对等的合作伙伴关系。<partnerLink>是BPEL所支持的最常见的关系的抽象形式。对于任何两方间的双向交互(例如业务流程所提供或调用的操作)，通过定义交互中使用的消息和端口类型，<partnerLink>可以指定与合作伙伴间的关系。在流程中，可动态确定实际的合作伙伴服务。另一个BPEL元素<partnerLinkType>可用来描述两个合作伙伴间的通信关系。当两个流程进行交互，并且交互中使用的端口类型是双向时，基于<partnerLinkType>可定义每一个合作伙伴所扮演的角色。最后，BPEL定义了端点引用，可用于表示寻址消息所需的静态或动态数据。在WS-Addressing中也定义了端点引用。下面我们将要描述BPEL流程编配，并首先描述<partnerLinkType>元素。

在BPEL中，和业务流程交互的服务将被建模为合作伙伴。当BPEL在<portType>层而不是端口/实例层组合服务时，BPEL的流程编配部分将使用基于类型的方式[Weerawarana 2005]。合作伙伴使用<partnerLink>以双边方式连接到流程，<partnerLink>表示了BPEL流程和所涉及的各方之间的交互。BPEL流程中所涉及的各方包括BPEL流程所调用的Web Service以及调用BPEL流程的客户端。可以使用相同的<partnerLinkType>描述多个合作伙伴。例如，一个采购流程在事务中可以有多个供应商，但是对所有的供应商都可以使用相同的<partnerLinkType>。<partnerLinkType>是两个或多个服务间的关系的声明，这些服务被暴露为一组角色，并且每个角色都标示了一个<portType>列表。然后，可将BPEL合作伙伴定义为一个特定<portType>的角色。服务关系定义了任何两个服务之间的链接，而所涉及的服务则通过WSDL文档中的targetNamespace进行限定。因此，<partnerLinkType>的角色元素指向一个WSDL <portType>。在WSDL <portType>中，合作伙伴提供了一个单向操作或一个请求/响应操作，以供其他合作伙伴使用。

<partnerLinkType>实际上并不是BPEL流程规范文档的一部分。<partnerLinkType>属于服务规范而不是流程规范。BPEL流程可以使用WSDL扩展性机制，相应的WSDL文档描述了流程的合作伙伴Web Service，<partnerLinkType>则处于该WSDL文档中。另一种可供选择做法是将所有的<partnerLinkType>规范保存在描述BPEL流程的WSDL文档中。然而这样做会违反封装性原则，因此并不推荐这一做法[Juric 2006]。

对于同步操作，因为仅从一个方向调用操作，所以每一个<partnerLinkType>中仅有单个的角色。因此，BPEL流程将不得不等待操作的完成，并且仅在操作完成后才能获得响应。假如<partnerLinkType>仅指定了一个角色，该关系中的一个服务必须实现一个WSDL

< portType > 。在下面的代码片段中，creditCheck 流程定义了一个具有单个角色creditChecker 的 < portType > 。通过称作 creditCheckPT 的 < portType > ，角色 creditChecker 引用了一个 WSDL 操作 initiate-credit-check。

```
<partnerLinkType name="creditCheckPLT">
    <role name="creditChecker"
                portType="tns:creditCheckPT">
    </role>
</partnerLinkType>
```

initiate-credit-check 操作定义了一个发送到服务提供者的输入，这将会有一个回复或一个故障。实现了 creditChecker 角色的服务也必须实现 creditCheckPT。

对于异步回调操作，需要指定两个角色。第一个角色描述了客户的调用操作。第二个角色描述了回调操作的调用。假如 < partnerLinkType > 元素指定了两个角色，该关系中所涉及的两个服务各自都必须实现一个角色。在下面的代码片段中，creditCheck 流程定义了两个角色 creditRequestor 和 creditChecker：

```
<partnerLinkType name="creditChecPLT">
    <role name="creditRequestor"
                portType="tns:creditCheck-CallbackPT">
    </role>
    <role name="creditChecker"
                portType="tns:creditCheckPT">
    </role>
</partnerLinkType>
```

这里，实现 creditRequestor 角色的服务也必须实现 creditCheckCallbackPT，而实现 creditChecker 角色的服务也必须实现 creditCheckPT。

在 BPEL 中，< partnerLinkType > 描述了交互的两个流程所需支持的端口类型，但是它并没有标识流程本身。许多流程对都可以满足在 < partnerLinkType > 规范中描述的需求。在 BPEL 中为了实现这一目标，可将与业务流程进行交互的服务建模为 < partnerLink > 。 < partnerLink > 定义了与 BPEL 流程交互的各方。 < partnerLinkType > 、业务流程的角色以及合作伙伴的角色(在链接的另一端的流程)这三者描述了一个 < partnerLink > 元素。

< partnerLink > 能够指定单个的角色，在同步请求/响应操作中，这通常是常有的事。在同步 BPEL 流程中，处理结果将立即返回给客户端。客户端将一直被堵塞，直到收到返回结果。流程的 WSDL 接口将有一个请求/响应类型的端点，如图 9.15 所示。同步类型的流程通常遵循下面的逻辑和语法模式：

```
<process>
    <receive partnerLink="CreditChecking"
     portType="CreditCheckPT"
     operation="initiate-credit-check" variable="creditCheckVar">

     ... perform processing ...

   <reply partnerLink="CreditChecking" portType="CreditCheckPT"
     operation="initiate-credit-check"
     variable="creditCheckResponseVar">
</process>
```

当流程需要长时间进行计算时，可以使用异步 BPEL 流程。当使用异步 BPEL 流程时，客户端不需要堵塞调用。相反，客户端实现了回调接口，并且一旦可以得到结果，BPEL 流程对客户端进行一个回调调用。假如被编配的底层的 Web Service 是异步的，则可以使用一个异步的 BPEL 流程。异步类型的流程通常遵循下面的逻辑和语法模式：

```
<process>
    <receive partnerLink="CreditChecking" portType="CreditCheckPT"
     operation="initiate-credit-check" variable="creditCheckVar"/>

     ...Perform time-consuming processing ...

     <!  Perform an invocation on the client to return the results  >
     <invoke partnerLink="CreditChecking" portType="
     CreditCheck CallBackPT"
     operation="credit-check response"
     inputVariable="creditCheckResponseVar"/>
</process>
```

对于异步操作，它指定了两个操作，对于流程中涉及的每个合作伙伴服务，对于流程中涉及的每一个合作伙伴。在流程定义中，对于流程中涉及的每一个合作伙伴服务，<partnerLinkType>元素标识了<partnerLink>引用的 WSDL <portType>元素。

图 9.16 例示了前面的异步 BPEL 通信模式。更精确地说，该图显示了合作伙伴链接、合作伙伴链接类型，以及与订购单流程、信用卡流程相关联的端口类型。请注意，合作伙伴链接用于业务流程所提供或调用的操作。

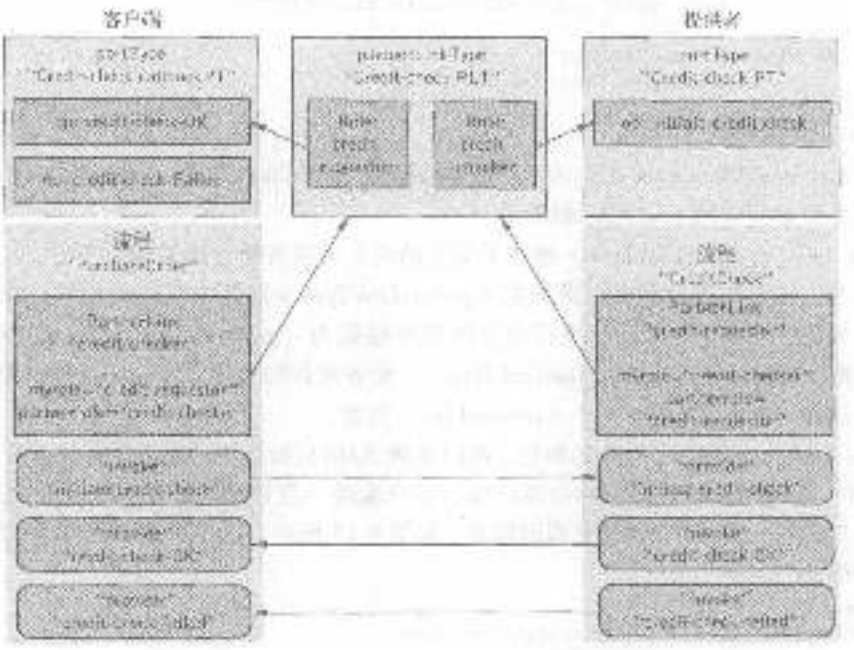


图 9.16 合作伙伴链接和相关的链接类型

引自：SAMS 出版社 2005 年出版的，由 S. Graham，C. Daniels，D. Davis，Y. Nakamura，H. Noyama，S. Simeonov 所著的《Building Web Services with Java》（第 2 版）一书允许复制。

通过将业务逻辑与可用的服务端点进行解耦，BPEL 业务流程将变得更有适应性和可移植性。在通过<partnerLink>调用合作伙伴服务上的操作之前，合作伙伴服务上的绑定和通信数据必须是可用的[Andrews 2003]。当流程需要执行时，每一个<partnerLink>必须绑定到一个具体的端点。共有四种绑定模式[Weerawarana 2005]。流程可以在设计时静态绑定到一个已知的端点。在所部署的流程中，通过指定一组端点，流程也可以在部署时进行静态绑定。假如一个特定的流程部署实例必须使用相同的端点，则该模式是有用的。在 BPEL 中，因为合作伙伴很可能是

有状态的，所以需要对服务端点信息中增加具体实例的信息。这就需要动态地选择和指派实际的合作伙伴服务。在 BPEL 中，可以按两种方式之一动态地选择、抽取和指派 < partnerLink > 中包含表示的端点引用。首先，< partnerLink > 可以包括 QoS 策略、事务能力或功能需求，通过评估与 < partnerLink > 相关的标准，可以使用查找。其次，可以使用从变量中复制的进入端点。这些变量先前并不是由流程本身赋值的，而是根据调用的响应（< invoke > 活动的结果）或者合作伙伴的请求（< receive > 活动的输入）来赋值的。

图 9.16 描述了 < partnerLink > 的四个角色。图 9.17 阐明了如何将这两个角色绑定到 Web Service 端点，并从订购单流程的角度显示了该绑定。对于信用卡请求者角色，WSDL 端口包含了流程中的操作的地址。对于信用卡检查者角色，WSDL 端口包含了 Web Service 的地址，这一地址是由订购单流程的业务合作伙伴所提供的。

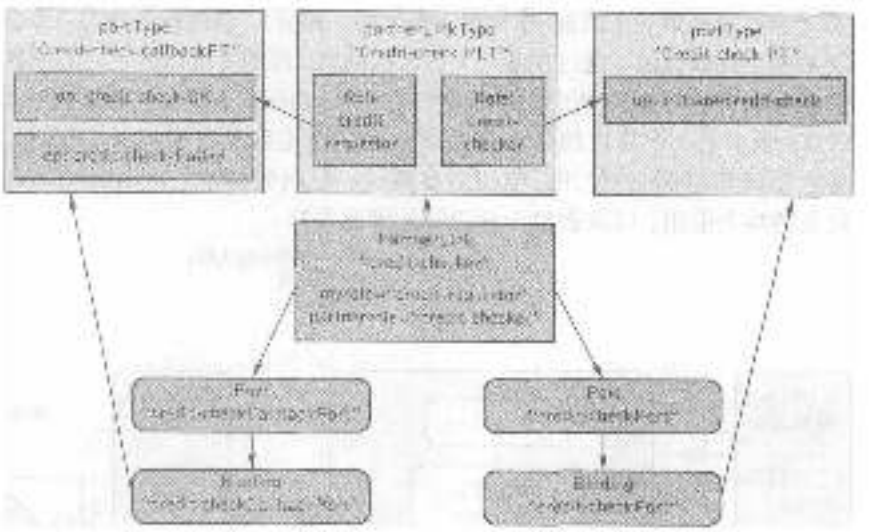


图 9.17　将 partnerLink 绑定到 Web Service 端点

6. 消息关联

通过 BPEL 中的消息关联，多个流程可参与有状态的会话。因为同一个实例可以有多个实例同时运行，所有必须有一些机制可以决定一个具体的消息针对的是哪一个实例。当消息抵达一个使用 BPEL 实现的 Web Service 时，必须确定需将消息交付给流程的已有实例或者流程的新实例。流程关联确定一个具体的消息到底属于哪一个会话，例如定位/初始化一个流程的任务。例如，可以使用关联将各类客户与涉及多方的、长期、并发运行的业务流程进行匹配。在涉及发送和接收消息的各类活动中，可以使用关联集。由于 < receive > 活动和 < pick > 活动提供了流程的入口点，因此通过关联集可以提供将消息发送到具体实例的路由。

消息关联就是将相关的消息联系在一起。当一个服务异步地调用另一个服务时，通常会发生消息关联，并传递关联标志（correlation token）。例如，在一个与定购单相关的、长期运行且涉及多方的业务流程中，可以将供应商识别号作为关联标志来标识各个供应商。在关联中，对于任何使用，属性名（例如供货商编号，定购单号、发票号、订单编号等）都必须具有全局意义。属性表示了消息交换中的数据元素。一条关联标志可定位为关联组中所有消息共享的一些属性。在 BPEL 中，通过消息关联，业务流程引擎可以创建一些新的流程实例，用于处理进入的请求，或者基于请求中的消息属性，将进入的请求路由到已有的流程实例。

关联集声明了关联标志，遵循 BPEL 的基础架构使用关联标志构建实例路由。关联集本质上

是一组属性集，属性值相同的所有消息是同一交互的一部分，因此同一实例将处理这些消息。这意味着，关联集标识了一个具体流程的一组实例中的一个特定的流程实例。一般而言，关联集与一个 < port > 一起唯一地标识了给定计算机上所有流程实例中的一个流程实例。对于支持异步服务操作，关联集尤其有用。

在作用域内可声明关联集。可类似于变量声明的方式，将关联集进行关联。每一个关联集都在一个作用域内进行定义，并属于该作用域。关联集可以属于一个全局处理作用域或者属于其他的非全局作用域。

在多方业务协议中，合作伙伴开始一个消息交换，并在关联集中创建属性的属性值，标记该会话的关联集称作交换的发起者(initiator)，其他的所有合作伙伴称为消息交换的跟随者(follower)。在一个关联的消息交换中，每一个参与的流程既可以充当消息交换的发起者，也可以充当消息交换的跟随者。发起者流程发送第一个消息(作为调用操作的一部分)，该消息将启动一个会话。进入的消息可以提供关联集中的属性值。通过接收一个进入的消息，跟随者可以将它们的关联集绑定到会话中。发起者和跟随者在它们各自的流中都必须将第一个活动标记为绑定关联集的活动。

最后，图 9.18 表示了一个简化的异步订购单流程，该流程是基于图 9.14 的抽象流程开发的。图 9.18 说明了 BPEL 机制的使用，包括消息构成、控制流构成、< partnerLinkType > 、WS-Addressing 中定义的端点引用，以及借助于标识符的消息关联。

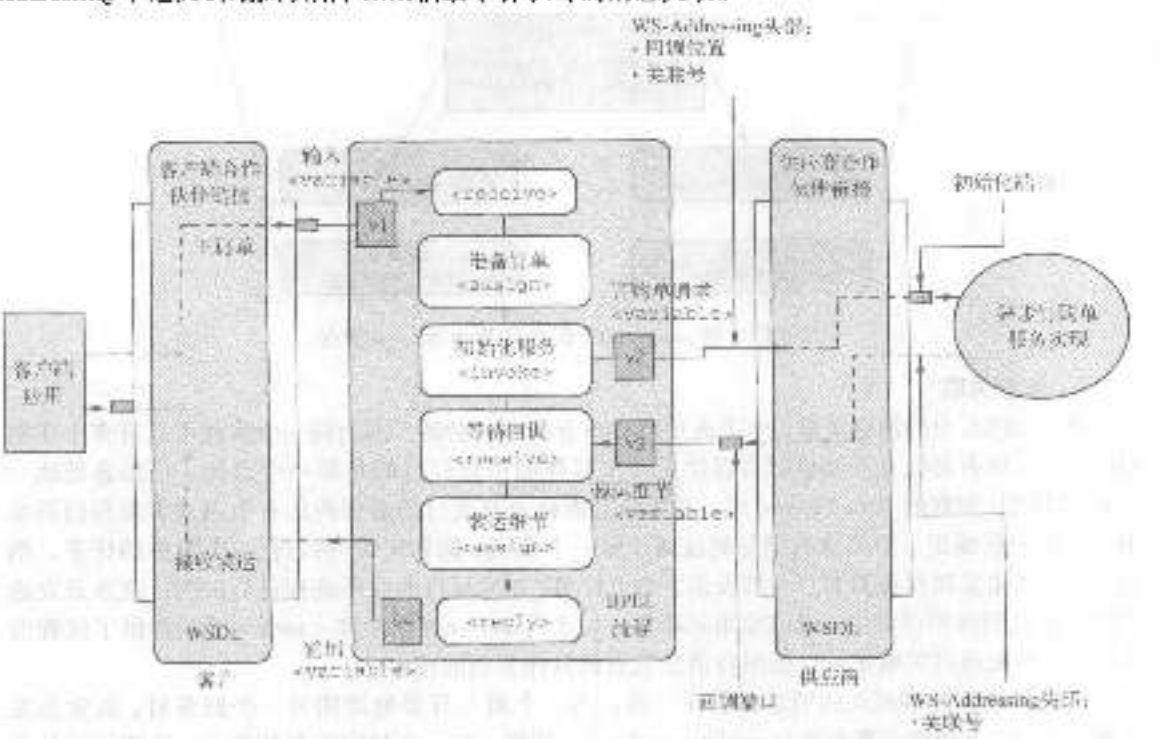


图 9.18　异步 BPEL 流程(对应于图 9.14 中的抽象视图)

7. 故障处理

在进行服务调用时，有可能会出现故障。BPEL < faultHandlers > 部分包含了一些结构，这些结构定义了处理故障的活动。在 BPEL 中，所有的故障，无论是内部产生的还是由于服务调用而导致的，都通过一个限定名进行标识。尤其，WSDL 文档中定义了相关的 < portType > 和故障，WSDL 文档的目标命名空间构成了限定名，BPEL 中的每一个 WSDL 故障都是通过限定名和故障

名进行标识的[Bloch 2003]。正如在 WSDL 定义中表示的，一些操作将返回故障。

业务流程中的故障处理可以视为是作用域中正常处理的一种模式切换。< scope > 对于嵌套在其中的活动提供了故障处理和补偿能力。< scope > 活动是一种将多个活动(或者聚集在公共的结构化活动之下的一组活动，诸如 < sequence > 或 < flow >)直接打包在一起，并提供一个活动环境，从而使得作用域中打包在一起的活动能够共享公共的错误处理和补偿方法。< scope > 活动包括一组可选的故障处理程序、一个可选的补偿处理程序，以及一个定义行为的作用域的基本活动。除了故障处理程序，在作用域中还可以定义作用域中的变量。

可以在任何作用域内定义故障处理程序。既可以将故障处理程序绑定到一个特定类型的故障(可根据故障的限定名或故障的消息类型进行定义)，也可以将故障绑定到任何没有更具体的处理程序进行处理的故障。可选的故障处理程序依附于一个作用域。基于这些可选的故障处理程序可以定义一组定制的故障处理程序，语法上定义为 < catch > 活动。每一个 < catch > 活动都被定义来拦截一个具体种类的故障。例如，< catch > 活动可以包含一个 < reply > 活动，通知合作伙伴出现了一个错误。< scope > 活动一旦接收到一个故障，做的第一件事情就是停止它自身所包含的所有活动。处理程序允许任何作用域拦截故障，并采取合适的动作。一旦在作用域中处理完一个故障后，将像通常一样评估向外链接的值。BPEL 的核心概念和可执行模式扩展定义了几个标准的含名字和数据的故障。除了这些，可能还有其他的一些特定平台的故障，诸如在业务流程实例中可能出现的通信故障。

可使用 < scope > 设置 BPEL 中的事务环境。该元素将相关的活动聚集在一起。对于执行中出现故障的活动，作用域的每一个故障处理程序可以调用一个补偿程序，退回活动执行前的状态。当具体应用中，对于一个没有使用二阶段提交(参见第 10 章的“嵌套事务中的两阶段提交协议”一节)比较大的工作单元，假如该活动仅是部分执行，剩下的部分由于出现故障而不得不取消，则可以调用补偿程序。BPEL 提供了一个补偿协议，可定义故障处理以及对具体应用的负面影响进行补偿，从而支持了长时间运行的(业务)事务[Bloch 2003]。使用 < compensate > 活动可以调用补偿处理程序。< compensate > 活动命名了执行补偿的 < scope > 元素。更确切地说，可调用 < scope > 的补偿处理程序。仅当作用域正常完成时，才可调用作用域的补偿处理程序。

8. 事件处理

在 BPEL 的控制结构中，除了 < receive > 和 < pick > 活动，BPEL 通过事件处理程序还提供了异步事件的并发处理。在作用域执行的任何点，事件处理程序使得作用域可以对事件进行反应，或者时间过期。< eventHandlers > 能够处理两类事件，即消息事件和警告事件。消息事件实施了请求/响应操作或单向操作。警告事件实现了时态语义。在指定的时间点或过期时都可能出现时态语义。只要 < eventHandlers > 包含许多 < onMessage > 或 < onAlarm > 活动，< eventHandlers > 活动类似于 < pick > 活动。< pick > 活动规定业务流程必须等待事件的出现。然而 < eventHandlers > 活动与 < pick > 活动不同，假如相应的事件出现，< eventHandlers > 活动可以与流程并发执行。从而，在原先不允许并发的控制“线程”的作用域中，可以进行并发处理。

9.7.2　BPEL 的简单样例

在本节中，为了解释我们在前面章节中介绍的 BPEL 构成，我们将使用一个简化版本的订购单应用作为例子，该应用如图 9.4 所示。在该例中，一个制造商下订单，并且一个供应商试图履行制造商的请求。供应商与信用卡服务、账单服务和库存服务提供者进行通信，从而力图满足客户的要求。一旦生成一个发票，然后则将发票送回给客户端。该流程的高层视图如图 9.19 所示。

我们将首先讨论这个例子的 BPEL 流程编配部分。在本节中，为了简洁起见，将略去一些 BPEL 细节。

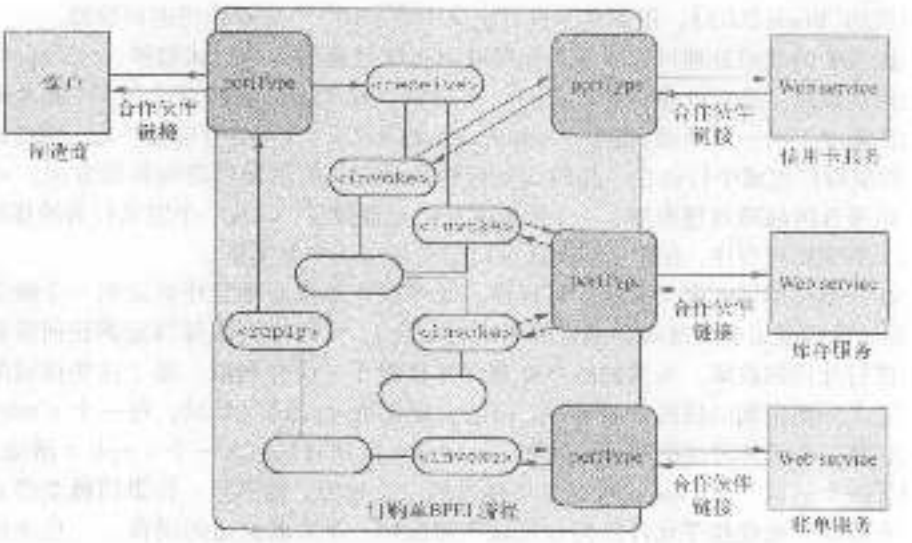


图 9.19 BPEL 订购单流程样例

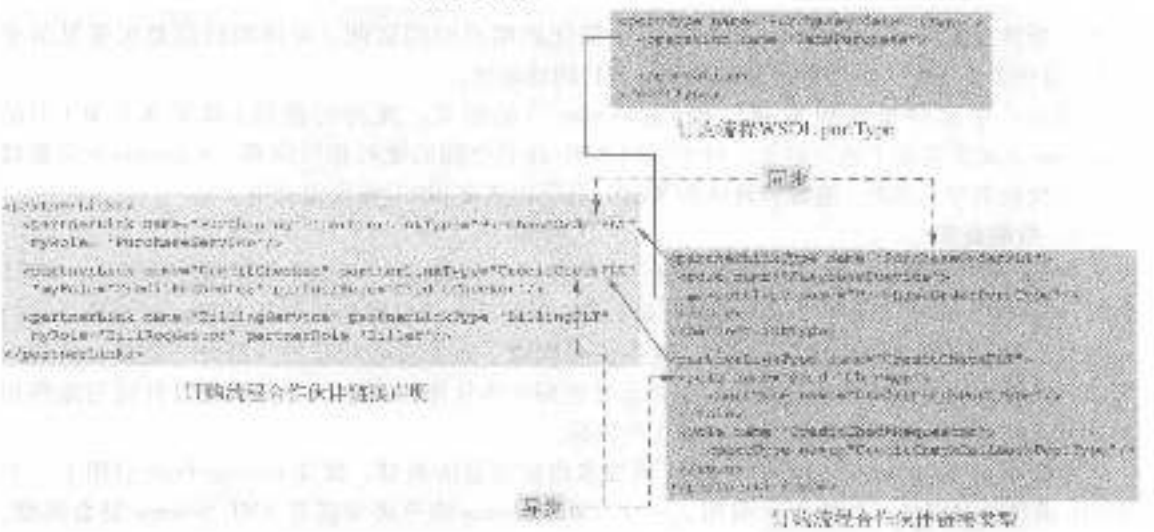
1. 流程编配

当在 BPEL 中定义业务流程时，我们正在有效地创建一个新的 Web Service。该 Web Service 组合了已有的基本的 Web Service。对于这个新的复杂的 Web Service，WSDL 规范定义了该 Web Service 和其他的 Web Service(也在 WSDL 中定义)间的关系。流程的 WSDL 规范指定了暴露给客户端的端口类型以及操作，甚至合作伙伴链接类型、流程的特性。通过 < partnerLinkType >，可以实现流程和合作伙伴 Web Service(例如信用卡核、支票、库存、账单等)间的关系。如前所述，通过定义每一个服务实现的角色，< partnerLinkType > 元素指定了两个服务之间的关系。每一个角色指定了一个 WSDL < portType >，实现那个角色的服务必须实现该 WSDL < portType >。在这个例子中，我们定义了下面的 < partnerLinkType > 元素："PurchaseOrderPLT"、"CreditCheckPLT"、"InventoryPLT"和"BillingPLT"。这些 < partnerLinkType > 元素中的开始两个如清单 9.2 所示。

每一个 < partnerLinkType > 定义了两个角色名，并列出了为了成功地进行交互，每个角色所必须支持的端口类型。在这个例子中，因为一个参与方提供了所有被调用的操作，所以 < partnerLinkType > "PurchaseOrderPLT"仅列出了单个角色。更具体地说，"PurchaseOrderPLT" < partnerLinkType > 表示流程和请求客户(制造商)之间的连接。在该连接中，仅订购单服务需要提供服务操作("sendPurchase")。正如已经讨论的，这意味着同步通信模式。由于信用卡核查服务的用户、库存服务的用户和账单服务的用户都必须提供回调操作，从而以使可以异步地将异步响应发送给它们的调用者，因此其他三个 < partnerLinkType > "CreditCheckPLT"、"InventoryPLT"和"BillingPLT"都定义了两个角色。清单 9.2 显示了一些 < partnerLinkType > 。每一个 < partnerLinkType > 元素也标识了 WSDL < portType > 。WSDL < portType > 通过 < partnerLinkType > 元素与每一个角色关联。< partnerLinkType > 定义了服务和所使用的 WSDL < portType > 间的依赖关系。例如清单 9.2 显示了 WSDL < portType > "PurchaseOrderPortType"与客户端发起的一个请求相关联。为了请求信用卡服务对客户进行核查，PurchaseOrder 流程也引用了"CreditCheck"提供者，并接收信用卡核查响应。这个通信基于两个端口类型"CreditCheck-CallBackPT"和"CreditCheck-CallBackPT"。

```
<partnerLinks>
    <partnerLink name="Purchasing"
      partnerLinkType="PurchaseOrderPLT"
      myRole="PurchaseService"/>
    <partnerLink name="CreditChecker"
      partnerLinkType="CreditCheckPLT"
      myRole="CreditRequestor"
      partnerRole="CreditChecker"/>
    <partnerLink name="InventoryChecker"
      partnerLinkType="InventoryCheckPLT"
      myRole="InventoryRequestor"
      partnerRole="InventoryService"/>
    <partnerLink name="BillingService"
      partnerLinkType="BillingPLT"
      myRole="BillRequestor"
      partnerRole="Biller"/>
</partnerLinks>
```

清单 9.2　在前面的 BPEL 片段中的角色的定义



或在，我们主要讨论 < partnerLinks > 元素，它是在 BPEL 工作流中指定合作伙伴声明的主要元素。< partnerLinks > 部分定义了在处理订单的过程中与业务流程进行交互的各方。< partnerLinks > 部分包含了几个 < partnerLink > 元素。每一个 < partnerLink > 元素是一个单个的合作伙伴声明。基于 < partnerLinkType > 元素，合作伙伴声明指定了企业和它的合作伙伴的角色。该信息标识了业务流程以及合作伙伴服务必须提供的功能。即订购单流程和合作伙伴需要实现的端口类型。前面的代码片段显示了四个 < partnerLink > 定义，这四个定义对应于订单的发送者（客户或者制造商），以及信用卡核查、库存和账单提供者，其中有些也显示在清单 9.2 中。

在清单 9.2 中，从订购单供应商（BPEL 流程本身）的角度定义了"PurchaseOrder"流程。当制造商（客户）与订购单供应商交互时，制造商是请求者，供应商是信用卡请求者和账单请求者（代表制造商）。相反，当供应商与"CreditCheckor"提供者或者"BillingService"提供者交互时，角色就完全相反了。

前面的代码指定了应用正在交互（通过 < partnerLinkType > 元素声明）的服务的类型。该代码也指定了将要提供所需功能的企业的角色（通过 < partnerLink > 声明）。为了实现业务间的相互关系，最后一步是指定合作伙伴的网络位置，从而使得我们可以发现和使用那些 Web Service。由于 WSDL

抽象接口和具体接口的不同，BPEL 需要解决抽象合作伙伴声明之间的差异，并在运行时和 Web Service 基于网络交换消息。端点引用元素解决了这一问题。端点引用元素是工作流的一部分，充当了具体服务的类型化引用[Andrews 2003]。BPEL 中的端点引用的基本功能是充当服务的特定端口数据的动态通信方式。基于端点引用，可动态选择特定类型服务的提供者，并调用它们的操作。例如，许多客户可以使用信用卡核查服务的"启动信用卡核查"。为了使得信用卡核查服务能够响应正确的合作伙伴，这些合作伙伴需要标识它们自己。基于端点引用元素，可将被定义的合作伙伴抽象地绑定到物理网络端点，并将那些端点（与其他的有用数据一道）暴露给工作流活动。

清单 9.3 端点引用声明

```
<wsa:EndpointReference xmlns:wsa="...">
    <wsa:Address>http://www.someendpoint.com/wsa:Address>
        <wsa:PortType>PurchaseOrderPortType</wsa:PortType>
</wsa:EndpointReference>
```

BPEL 使用了 WS-Addressing（参见 7.3.1 节）中定义的端点引用。在 BPEL 流程实例中，流程中的活动可静态或动态地给 <partnerLink> 的每一个合作伙伴角色赋予一个唯一的端点引用，订购单中所涉及的两方，清单 9.3 显示了一个简化的端点引用实例。更详细的信息可参见清单 7.3。清单 7.3 指定了订购单消息的源地址和目的地地址。

最后一个部分是流程本身（"PurchaseOrder"）的定义。流程的根层（参见 9.1 节）中的 <process> 元素完成了这一定义。对于引用 XML 命名空间的流程和供应商，<process> 元素提供了相应的名字。因此，流程将具体的 WSDL 引用存储在 BPEL 流程定义中。

2. 数据处理

BPEL 流程管理合作伙伴间的数据流。合作伙伴可以通过它们的服务接口进行表示。BPEL 的 <variables> 定义了流程使用的数据变量，并依据 WSDL 消息类型和 XML 模式元素提供了它们的定义。变量可作为驻留消息的方式，这些变量构成了业务流程状态的一部分。变量中驻留的通常是从合作伙伴那里接收的消息，或者是发送到合作伙伴的消息。变量中也可驻留与流程相关的状态所需的，并且合作伙伴从不修改的数据。

可使用 messageType、type 和 element 属性来指定变量的类型。属性 messageType 引用了一个 WSDL 消息类型定义。属性 type 引用了一个 XML Schema 简单类型或者 XML Schema 复合类型。属性 element 引用了一个 XML Schema 元素。例如，订购单业务流程可以在 PO 变量中存储 POMessage，如清单 9.4 所示。

清单 9.4 订购单流程的 BPEL 数据变量

```
<variables>
  <variable name="PO" messageType="tns:POMessage"/>
  <variable name="OrderNotificationMessage" messageType="tns:OrderNotificationMessage"/>
  <variable name="OrderNotificationMessage" messageType="tns:OrderNotificationMessage"/>
  <variable name="Fault" messageType="POFault:Type"/>
</variables>

<assign>
  <copy>
    <from variable="PO" part="purchaseOrder"/>
    <to variable="OrderNotificationMessage" part="PO"/>
  </copy>
</assign>
```

```
<wsdl:message name="POMessage">
  <wsdl:part name="PO" type="ns1:PurchaseOrderType">
</wsdl:message>
<wsdl:message name="OrderNotificationMessage">
  <wsdl:part name="PO" type="ns1:PurchaseOrderType">
</wsdl:message>
<wsdl:message name="OrderNotificationMessage">
  <wsdl:part name="PO" type="ns1:PurchaseOrderType">
</wsdl:message>
<wsdl:message name="Fault">
  <wsdl:part name="Fault" type="ns1:FaultType">
</wsdl:message>
<wsdl:portType>
  <wsdl:operation name="SendOrderNotice">
    <input message="tns:POMessage">
    <fault name="FaultOrderNotice">
      <message="tns:Fault" />
</wsdl:portType>
```

订购单流程WSDL定义



使用 <assign> 和 <copy>，可以在变量之间复制和操纵数据。可以使用 <assign> 元素

在变量之间复制数据（消息、消息的一部分和服务引用）。< copy > 支持 XPath 查询的子查询。赋值的一个典型例子将一个消息的内容拷贝到另一个。在清单 9.4 中，POMessage 的 PurchaseOrder 部分赋给了 creditRequester 的 PurchaseOrder 部分。一旦接收了消息后，该消息将存储在 creditRequest 变量中，然后，（订购应）流程可将 creditRequest 变量传递给信用卡服务提供者。这个消息使用了 PO 订单，信用卡服务提供者也使用这个变量来处理请求。如清单 9.4 所示，每一个变量都紧跟着对一个具体的 WSDL 消息类型的引用，并通过它与一个对应的 WSDL < message > 元素关联。

3. 控制流

处理请求所需的基本步骤序列的定义是 BPEL 应用的关键部分。基本的和结构化的活动将在流程中发挥各自的作用。在清单 9.5 所示的流程中，制造商（客户端）请求一个订购单，流程包含一个最初的请求，紧接着可并行地调用信用卡核查服务提供者、库存核查服务提供者和账单服务提供者。一旦信用卡核查和库存核查成功完成后，账单服务将发送交账单给客户。最终，供应商将发送一个发票给制造商。整个过程如图 9.20 所示。

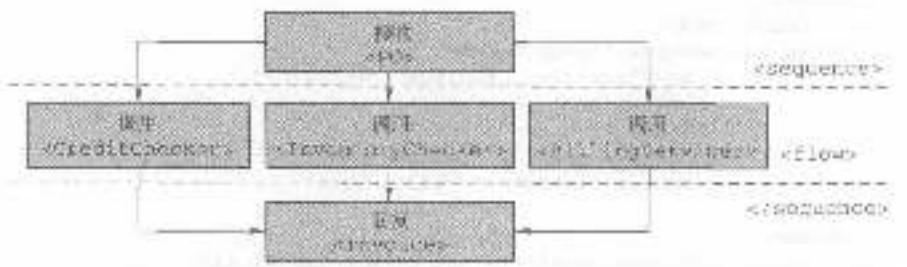


图 9.20　订购单流程的序列化和流活动

在清单 9.5 中，外层的 < sequence > 元素定义了主要处理部分的结构。包含在 < sequence > 元素中的活动将依次执行。首先接收制造商请求（< receive > 元素），然后处理该请求（通过分配多个 < invoke > 活动，< flow > 部分可允许并发的异步行为），并将包含请求的最终批准状态的消息发送给顾客（< reply >）。通过 < partnerLink >，可使用 < invoke > 活动调用服务提供者提供的 Web Service。包含在 < flow > 元素中的所有活动都可能具有并发行为，可按照 < link > 元素表示的依赖性执行这些行为。

流程中的第一步是发起制造商请求。一旦收到请求后，一组活动使用 < flow > 元素并行执行。其中，为了核查库存，将要用到库存服务；为了接收客户的信贷，将会使用信用卡服务。最后，将使用账单服务给客户发送账单。每一个活动引用一个具体的 WSDL 操作（例如 SendPurchase 或 CheckCredit），并使用变量来进行输入输出。一旦接收到信用卡服务、库存服务和账单服务发回的响应，供应商将构建一个消息，并将该消息发送给客户。这涉及 XPath 语言的使用，通过变量接收服务提供者发送的消息，并将最终结果发送给购买者。客户调用 Purchase 操作，< receive > 元素和 < reply > 元素分别匹配 Purchase 操作的 < input > 消息和 < output > 消息（参见清单 9.4）。在 < receive > 元素和 < reply > 元素之间，流程需要完成一些活动。这些活动表示了为了响应客户需求，从接收到请求开始直到送回响应（回复）期间所需采取的动作。上面的例子假定：可在合适的时间内处理好客户的请求，因此调用者必须等待同步响应（因为将服务作为请求/响应操作）。

清单 9.5　对应于订购单流程的 BPEL 流

```
<sequence>
  <receive partnerLink="Purchasing"
                  portType="lns:PurchaseOrderPortType"
                  operation="SendPurchase" variable="PO"
                  createInstance="yes" >
  </receive>
  <flow>
      <links>
            <link name="inventory-check"/>
            <link name="credit-check"/>
      </links>
      <!-- Check inventory -->
      <invoke partnerLink="inventoryChecker"
                   portType="lns:InventoryPortType"
                   operation="checkInventory"
                   inputVariable="inventoryRequest"
                   outputVariable="inventoryResponse">
                  <source linkName="inventory-check"/>
          ...
      </invoke>
      <!-- Check credit -->
      <invoke partnerLink="creditChecker"
                   portType="lns:CreditCheckPortType"
                   operation="checkCredit"
                   inputVariable="creditRequest"
                   outputVariable="creditResponse">
                  <source linkName="credit-check"/>
          ...
      </invoke>
      <!-- Issue bill once inventory and credit checks are
                                                    successful -->
      <invoke partnerLink="BillingService"
                   portType="lns:BillingPortType"
                                operation="billClient"
                   inputVariable="billRequest"
                              outputVariable="Invoice"
                   joinCondition="getLinkStatus
                                ("inventory-check") AND
                           getLinkStatus("credit-check")">
                   <target linkName="inventory-check"/>
                   <target linkName="credit-check"/>
          ...
       </invoke>
  </flow>
     ...
  <reply partnerLink="Purchasing" portType="lns:purchaseOrderPT"
   operation="Purchase" variable="Invoice"/>
</sequence>
```

在清单 9.5 中，包含在 < flow > 中的活动具有同步依赖性，通过使用 < links > 可连接这些活动，从而表示了活动间的这种依赖关系。在 < flow > 中定义了 < links > 。通过 < links > ，可将源活动连接到目标活动。注意，每一个活动都可使用 < source > 元素或 < target > 元素将自身声明为 < link > 的源或目标。变迁条件隶属于链接的 < source > 元素。变迁条件决定了将激活哪一个链接。在没有链接的情况下，所有直接嵌套在流中的活动并发进行。在订购单样例中，对于每一个序列中执行的活动，两个链接控制了活动间的依赖性。invoke 活动 inventoryChecker 和 creditChecker 的 < source > 元素是链接 inventory-check 和链接 credit-check。invoke 活动 billingService 中的 joinCondition 属性对于每一个进入的链接的状态进行一个“逻辑与”操作，因此仅当该服务的前

面的所有活动(例如 inventoryChecker 调用和 creditChecker 调用)都表明所运行的检查没有问题，才会执行该服务。为了简介起见，清单 9.5 并没有表现这一点，但是可以作为 inventoryChecker 活动和 creditChecker 活动上的 transitionCondition 实现。

4. 关联

图 9.21 图形化地表示了一个关联。对应下该图例，清单 9.6 显示了 BPEL 规范。几其该图显示了，对于每一个订购单，存在一个唯一的订购单标识符。制造商将转发这些订购单，而供应商则接收订购单，并且供应商所生成的每一个发票都有一个唯一的号码。订购单(PO)文档包含一个订购单号码，通过发送订购单以及将订购单号码作为一个关联标志，制造商可以和供应商开始一个关联的交换。供应商在订购单确认中使用该订购单号码。以后，供应商可以发送一个发票文档。发票文档中就包含了订购单号码，也包含了发票号码，其中订购单号码可用于和原先的订购单进行关联。因此，以后相关的支付消息可以仅将发票号码作为关联标志。发票消息含有两个独立的关联标记，并参与两个重叠的关联交换。通常，关联的作用域并不是服务指定的整个交互，而可能仅涉及服务行为的一部分。

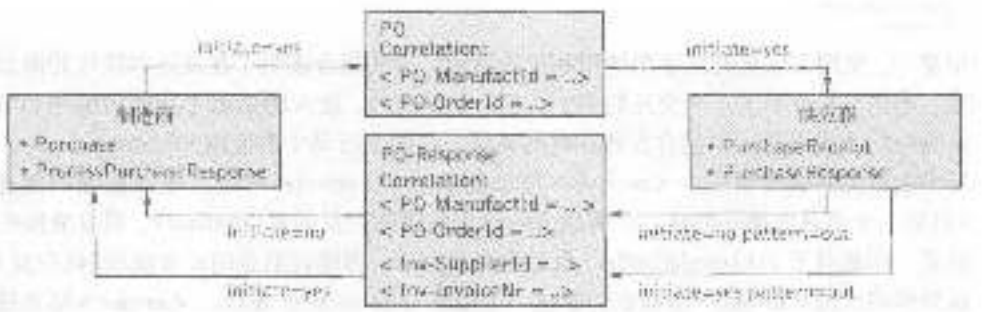


图 9.21　订购单流程的关联属性和集合

清单 9.6　表示图 9.21 的代码片段

```
<!-- define correlation properties -->
    <bpws:property name="manufactId" type="xsd:string"/>
    <bpws:property name="orderNumber" type="xsd:int"/>
    <bpws:property name="supplierID" type="xsd:string"/>
    <bpws:property name="invoiceNumber" type="xsd:int"/> ...
<!-Define schema types and messages for PO and Invoice
   information -->
<types>
        <xsd:complexType name="PurchaseOrder">
            <xsd:element name="CID" type="xsd:string"/>
            <xsd:element name="order" type="xsd:in."/>
            ...
        </xsd:complexType>
        <xsd:complexType name="PurchaseOrderResponse">
            <xsd:element name="CID" type="xsd:string"/>
            <xsd:element name="order" type="xsd:int"/>
            ...
        </xsd:complexType>
</types>
<message name="POMessage">
        <part name="PO" type="tns:PurchaseOrder"/>
</message>
<message name="POResponse">
        <part name="RSP" type="tns:PurchaseOrderResponse"/>
</message>
<correlationSets>
```

```
    <correlationSet name="POCorr" properties="cor:manufactId
     cor:orderId"/>
    <correlationSet name="InvoiceCorr" properties="cor:supplierId
     cor:invoiceNumber"/>

 </correlationSets> ...

 <receive partnerLink="Manufacturer"
                                  portType="POSubmissionPortType"
       operation="SubmitPurchase" variable="PO">
   <correlations>
       <correlation set="POCorr" initiate="yes">
   </correlations>
 </receive> …

 <invoke partnerLink="Manufacturer"
                                 portType="orderCallbackPortType"
      operation="PurchaseResponse" inputVariable="POResponse">
 <correlations>
     <correlation set="POCorr" initiate="no" pattern="out">
     <correlation set="InvoiceCorr" initiate="yes" pattern="out">
 </correlations>
</invoke> …
```

在清单中，使用关联集的名字来调用和接收活动。这些集合表明了在发送和接收的消息中有哪些关联集。清单9.6显示了一个交互的例子，在一个单向的、进入的请求中接收订购单(PO)请求，并以异步响应的方式发送一个包含发票在内的确认。在两个活动中都使用PO correlationSet，以便于能够将异步响应关联到请求中。< receive >活动初始化PO correlationSet。正如前面所阐述的，在BPEL中启动一个消息交换并创建一个属性值的合作伙伴称为发起者(initiator)，消息交换的另一方称为追随者。因此对于PO correlationSet，制造商是发起者，而接收消息的业务流程(供应商)是追随者。发送异步响应的< invoke >活动也初始化一个新的correlationSet发票。< invoke >活动指定了合作伙伴(制造商)所调用的WSDL < portType >和操作，PO correlationSet在这个活动之前就已经初始化了。在这个例子中，该业务流程是关联交换的发起者，制造商是跟随者。因此，清单中的响应消息是两个独立的会话的一部分，这两个会话通过消息链接在一起。

5. 出错处理与调整

清单9.7说明了一个情况，即当制造商提交一个订购单时，出现一个错误。在这种情况下，供应商可以使用一个故障处理程序(OrderNotComplete)，应用< reply >元素向制造商返回一个错误。

清单9.7 订单处理流程中的一个简单的故障处理程序

```
<faultHandlers>
 <catch faultName="OrderNotComplete"
        faultVariable="POFault">
  <reply partnerLink="Manufacturer"
         portType="PurchasePortType"
         operation="Purchase"
         variable="POFault"
         faultName="OrderNotComplete"/>
 </catch>
</faultHandlers>
```

为了说明补偿处理的概念，现假设如下情况：已经进行了订购，并且现在需要取消订购。这一情况如清单9.8所示。在这个例子中，原先的invoke活动已经进行了订购，并且现在需要对订购进行补偿。在一个WSDL合作伙伴的相同端口，*compensationHandler*调用一个取消操作，并使用对订购单的响应作为它的输入。

清单 9.8　可能的补偿活动

```
<invoke partnerLink="Seller" portType="SP:Purchasing"
        operation="SyncPurchase"
        inputVariable="sendPO"
        outputVariable="getResponse">
  <correlations>
     <correlation set="PurchaseOrder" initiate="yes"
      pattern="out"/>
  </correlations>
  <compensationHandler>
    <invoke partnerLink="Seller" portType="SP:Purchasing"
            operation="CancelPurchase"
            inputVariable="getResponse"
            outputVariable="getConfirmation">
       <correlations>
          <correlation set="PurchaseOrder" pattern="out"/>
       </correlations>
     </invoke>
  </compensationHandler>
</invoke>
```

对于协调跨工作流实例的分布式事务，BPEL 规范建议将 WS-Transaction（参见第 10 章）作为可供选择的协议。因此，当调用合作伙伴的 Web Service 需要补偿时，底层的 BPEL 引擎应该确保能将合适的 WS-Transaction 消息发送给事务协调者。从而当需要补偿所调用的活动时，能够通知所有的合作伙伴。

9.8　编排

涉及多个组织或独立流程的业务应用能够以协作方式实现一个公共的业务目标，诸如订单管理。为了能够成功地进行协作，在参与的服务间需要能够进行长时间运行的、对等的消息交换，例如在企业内部或信任域进行服务编排。编排的主要目标是：在运行时确认所有的消息交换都按照计划在进行，并保证服务实现的变化依然遵循消息交换的定义。

9.8.1　编排描述的使用

编排描述是一个多方契约，它从全局角度描述了跨多个客户端（通常是 Web Service）的、外部可观察的行为。这些可观察的行为定义为 Web Service 和客户端之间交换的消息的存在或不存在。由于这类编排既不是一个可执行的业务流程描述语言，也不是一个可执行的业务流程实现语言（诸如 Java 或 C#），因此它在一定程度上需要监测和验证（或作废）。它的角色就是指定任何类型的参与者之间的真正的、可互操作的对等协作，无须考虑驻留环境所支持的平台或所使用的编程模型。

图 9.12 说明了业务编排语言是如何与业务流程语言层进行协调的。编排描述语言（CDL）是一种描述多方协作的方法。在图 9.12 中，使用图形用户界面和工具集来指定制造商和供应商之间的交互，并生成一个 WS-CDL 表示。然后对于制造商和供应商，可以使用该 WS-CDL 表示来生成一个反映它们的业务协定的 BPEL 工作流模板。

为了提高参与者之间的共识，以及验证一致性、确保互操作性、生成代码框架，编排描述的主要作用就是精确地定义合作的 Web Service 间的交互序列[Austin 2004]。可以使用编排描述来生成所需的代码框架。对于那些 Web Service，代码框架实现了所需的外部的、可观察的行为。例如，对于制造商、大量的可能的供应商以及大量的供应商，可以使用编排描述来描述它们多方之间的约定。任何参与方都可使用编排描述生成与特定制造商交互的 Web Service 的代码框架。

9.8.2 Web Service 编排描述语言

Web Service 编排描述语言是一个 XML，用于构成任何类型的参与方之间的互操作的、长时间运行的、对等协作，且无须考虑驻留环境所支持的平台或所使用的编程模型。在业务协作中涉及多个 Web Service 参与者，所有的这些参与方之间将进行消息交换，这些消息交换具有可观察的行为，WS-CDL 描述了可观察行为的全局视图。使用 WS-CDL 规范，可以生成一个约定，这个约定包含了通用排序条件和约束的“全局”定义，消息交换将遵循这些通用排序条件和约束。这个约定从全局视角描述了所有相关方的、通用的和互补的、可观察行为。然后，每一方都可以使用这个全局定义来构建和测试遵循这一定义的解决方案。WS-CDL 完全用于指定“抽象业务流程”，独立于 Web Service 实现所使用的平台和编程语言。

在 WS-CDL 规范中，消息交换代替了一个共同协商的排序和约束规则集。编排定义能够涉及两个(两方)或多个(多方)参与者。BPEL 抽象流程是从一个参与者的角度描述的。与此不同，WS-CDL 描述了消息交换的一个全局视图，而不是从任何一方的角度出发。当参与方的数量增多时，这一方法具有较多的可伸缩性。然而和 BPEL 一样，WS-CDL 是一个基础架构规范，不包含任何业务语义(例如资源、承诺、协定等)。

在 WS-CDL 中，通常抽象地定义角色(role)间的编排定义。角色将被绑定到参与者(participant)。角色之间通过关系(relationship)相互联系。一个关系通常金发生在两个角色之间。在编排中，一个参与者可以实现任何数量的非对立的角色。一个分销商可以实现“购买者到制造商”和“销售者到顾客”的角色，这些角色不同于“销售者到分销商”的角色。WS-CDL 中的角色有点类似 BPEL 中的 < partnerLink > 。

编排由活动组成。主要活动称为一个交互，它是编排的基本构建块。编排导致了参与者之间的消息交换、状态和实际交换信息的值的同步。交互指定了角色间消息交换的单元。交互对应于角色上的 Web Service 操作的调用。因此，可将交互定义为具有零个或多个响应的请求。交互能够涉及排序活动(顺序、平行、选择)或者在父编排中组成另一个编排。编排定义可以是数据驱动的，例如包含在消息中的数据影响交互的顺序。数据被建模为变量、信道、编排中所涉及的角色的状态。其中，变量可以与消息内容关联。标记(token)是一个表示变量的一部分。在 WS-CDL 中，标记与 BPEL 关联集的属性类似。

WS-CDL 文档是一个定义集。WS-CDL 定义是一些能够引用的、被命名的构成。在根节点有一个 package 元素，并且在 package 元素里有单独的编排定义。WS-CDL package 中包含一个或多个编排，以及一个或多个协作类型定义。基于 WS-CDL package 构成，可以实现编排定义的嵌套。

清单 9.9 显示了一个编排样例，该样例是在 WS-CDL 中指定的。如清单所示，< package > 元素包含一个顶层 < choreography > ，并直接标记为根 < choreography > 。可初始化该根 < choreography > ，并且该根 < choreography > 涉及一个交互。这个交互作为一个请求/响应消息交换，发生在从制造商角色到供应商角色的“供应商信道”上。在清单中，PurchaseOrder 消息作为一个请求消息从制造商发送到供应商，PurchaseOrderAck 消息作为一个响应消息从供应商发送到制造商。

清单 9.9 WS-CDL 中的一个编排样本

```
<package name="ManufacturerSupplierChoreography" version="1.0"
    <informationType name="purchaseOrderType"
                     type="pons:PurchaseOrderMsg"/>
    <informationType name="purchaseOrderAckType"
                     type="pons:PurchaseOrderAckMsg"/>
    <token name="purchaseOrderID" informationType="tns:intType"/>
    <token name="supplierRef" informationType="tns:uriType"/> ……
    <role name="Manufacturer">
```

```
        <behavior name="manufacturerForSupplier"
                  interface="cns:ManufacturerSupplierPT"/>
        <behavior name="manufacturerForWarehouse"
                  interface="cns:SupplierWarehousePT"/>
    </role>
    <role name="Supplier">
        <behavior name="supplierForManufacturer"
                  interface="rns:ManufacturerSupplierPT"/>
    </role>
    <relationship name="ManufacturerSupplierRelationship">
        <role type="tns:Manufacturer" behavior=
                                        "manufacturerForSupplier"/>
        <role type="tns:Supplier"
                              behavior="supplierForManufacturer"/>
    </relationship>
    <channelType name="ManufacturerChannel">
        <role type="tns:Manufacturer"/>
        <reference>
            <token type="tns:manufacturerRef"/>
        </reference>
         <identity>
             <token type="tns:purchaseOrderID"/>
         </identity>
    </channelType>    ……
    <choreography name="ManufacturerSupplierChoreo" root="true">
       <relationship type="tns:ManufacturerSupplierRelationship"/>
       <variableDefinitions>
       <variable name="purchaseOrder"
                 informationType="tns:purchaseOrderType"/> ……
       <variable name="supplier-channel"
                              channelType="tns:supplierChannel"/>
              ...
       <interaction channelVariable="tns:supplier-channel"
                    operation="handlePurchaseOrder"
                    align="true"initiateChoreography="true">
          <participate
            relationship="tns:ManufacturerSupplierRelationship"
            fromRole="tns:Manufacturer" toRole="tns:Supplier "/>
          <exchange messageContentType="tns:purchaseOrderType"
                    action="request"/>
              <use variable="cdl:getVariable(tns:purchaseOrder,
                   tns:Manufacturer)"/>
             <populate variable="cdl:getVariable
                       (tns:purchaseOrder,tns:Supplier)"/>
          </exchange>
          <exchange messageContentType="purchaseOrderAckType"
                    action="respond">
              <use variable="cdl:getVariable
                   (tns:purchaseOrderAck,tns:Supplier)"/>
              <populate variable="cdl:getVariable
                        (tns:purchaseOrderAck,tns:Manufacturer)"/>
          </exchange>
          <record role="tns: Supplier "action="request">
              <source variable="cdl:getVariable(tns:purchaseOrder,
                      PO/Manufacturer Ref, tns: Supplier)"/>
              <target variable="cdl:getVariable
                      (tns:manufacturer-channel,tns:Supplier)"/>
          </record>
       </interaction>
    </choreography>
</package>
```

在请求端，供应商使用 < record > 元素填入 < variable > consumer-channel 的值。在 < interac-

tion > 端点的两个角色，可以使用 < record > 元素创建/修改一个或多个状态。例如，当 PurchaseOrder 消息发送给 < role > “supplier”时，该消息包含 < role > “manufacturer”的信道。这能够被拷贝到 < record > 元素中的“supplier”的合适状态变量。对于 < interaction >，当设置为“true”时，这也意味着：制造商已经知道供应商现在有了制造商的地址。< record > 元素的另一个作用是可以在每一个 < role > 记录状态。在 < interaction > 的请求端，制造商将“OrderSet”状态设置为“true”，零售商将“OrderReceived”状态设置为“true”。类似地，在 < interaction > 端，客户将“OrderAcknowledged”设置为 true。在 < record > 元素的角色属性中可以指定 < role > 元素。< record > 元素中的 < target > 元素和 < source > 元素表示了关联到 < role > 元素的 < variable > 名。

在清单 9.9 中，< interaction > 活动发现供应商信道，该信息使用 < token > purchaseOrderID 作为信道的标识。这个标识元素用于标识供应商的业务流程，而响应消息 purchaseOrderAck 则包含了制造商业务流程的标识。

在 WS-CDL 表示了 Web Service 栈的一个重要的、新的分层，该层对 BPEL 进行了补充。在 W3C 的规范流程中，WS-CDL 仍然处于初期阶段，因此很难精确地描绘最终的推荐规范，也很难确定 OASIS BPEL 和 W3C WS-CDL 工作组关于这两个标准的相互协作是否能达成一致。

9.9 其他的一些提案和语言

除了 BPEL 和 WS-CDL，在过去的几年中还提出了其他的一些基于 XML 的流程定义语言。这些流程定义语言都提供了表示可执行流程的模型。该模型涉及了企业业务流程的各个方面，并基于不同的范式。在下面，我们将要简要介绍两个这样的提案：XML 流程定义语言(XPDL)和业务流程建模语言(BPML)。

工作流管理联盟(WfMC)提出了 XPDL(http://www.wfmc.org/standards/XPDL.htm)，将其作为在不同的工作流产品中进行流程定义交换的方式。XPDL 提出了一个通用的交换标准，使得产品能够不断地支持流程定义的任意内部表示，并具有导入/导出功能，可在产品边界上映射到标准，或从标准进行映射。按照不同的业务场景的特性，可以使用许多不同的方式在系统之间转换流程定义的数据。可按统一的方式表示流程定义，通过公共对象集、关系和属性表示它的底层概念。工作流建模和模拟工具可以生成 XPDL 流程定义。从其他的遵循 XPDL 的工作流引擎中，也可导入 XPDL 流程定义。

BPML(http://www.bpmi.org/BPML.htm)是一个基于 XML 的元语言。该语言是由业务流程管理促进会(BPMI)开发的，作为一种建模业务流程的方法。为了表示抽象的和可执行的流程，BPML(已经过时了)定义了一个形式模型。该模型涉及企业业务流程的各个方面，包括不同复杂性的活动、事务和它们的补偿、数据管理、并发性、异常处理和操作语义。为了表示业务流程以及支持实体，BPML 规范提供了一个抽象模型和 XML 语法。BPML 本身并没有定义任何应用语义，诸如特定域中的流程或流程的应用，而是定义了一个可表示一般流程的抽象模型和语法。

文献[Shapiro 2002]对 XPDL、BPML 和 BPEL 进行了一个很有价值的比较。

9.10 小结

业务流程指定了服务间传送的共享数据、流程的交易合作伙伴角色、Web Service 集合的公共异常处理条件，以及指定了其他一些因素，这些因素可能将影响到流程中的 Web Service 和参与的组织。可使用面向流程的工作流来自动化业务流程，这些业务流程的结构是定义明确的，并且不会随着时间的推移而变化。业务流程工作流由若干活动组成。这些活动实现为一组横跨多个 Web Service 的、限域的操作。检查点可通过业务条件和规则进行表示。以 Web Service 为中心的

活动遵循具有检查点的路由。

为了指定如何将单独的 Web Service 组合起来创建可靠的、可信任的、具有适当的复杂度的、基于业务流程的解决方案，已经诞生了许多不同的 Web Service 组合语言。服务组合语言横跨服务编配和服务编排。服务编配描述了：从单个端点的角度以及在单个端点的控制之下，Web Service 间在消息层如何进行相互交互。与编配相反，编排从全局角度定义了 Web Service 的公共的和互补的、可观察的行为。当进行消息交换，以及满足共同商定的排序规则时，Web Service 参与者之间通常是对等协作关系。

对于 Web Service，业务流程执行语言最近已经成为一个标准，用于定义和管理业务流程活动和业务交互协议。从编配角度，这些业务流程活动和业务交互协议由协作的 Web Service 组成。

Web Service 编排描述语言是一个 XML 规范。对于业务协作中涉及的所有的 Web Service 参与者间的消息交换，Web Service 编排描述语言可用来描述消息交换的可观察行为的全局视图。

复习题

- 简要描述自动化业务流程的主要特性。
- 列举并描述工作流系统的主要组件。
- 试举一个使用 UML 的工作流应用的例子。
- 什么是业务流程集成？业务流程集成和业务流程管理的不同之处是什么？
- 跨企业的业务流程的作用是什么？并描述它们与工作流系统间的关系。
- 服务组合元数据模型的作用是什么？
- 什么是流模型？什么是控制链接和变迁条件？
- 如何组成 Web Service？
- 什么是 Web Service 编配和编排语言？它们之间的不同点是什么？
- 列举并描述 BPEL 的主要组件。
- BPEL 如何编配 Web Service？
- WS-CDL 的作用是什么？它如何能够与 BPEL 协作？

练习

9.1　进一步开发 9.7.2 节中的 BPEL 流程样例，使其包括信用卡核查提供者和账单服务提供者。

9.2　对于简化的装运服务，开发一个抽象的 BPEL 流程。该服务处理订单的装运，订单由许多项(货物)组成。该装运服务将提供两个选项。一个选项是将所有货物一起装运。另一个选项是将货物进行部分装运，即将货物分组进行装运，直至履行整个订单。

9.3　扩展练习 9.1 中的 BPEL 流程，提供一个库存核查服务。提供订购单服务的 BPEL 流程将要用到该库存核查服务。当订购单服务提供者接收客户端请求时，将发生下面的事件：

(a) 订购单服务提供者指定价格以及请求的当前日期。

(b) 调用库存服务来核查库存的状态。

(c) 库存服务将核查是否有货，并向订购单服务提供者进行报告。

(d) 基于库存服务的结果，订购单服务提供者可有两种响应，一种是履行订购单，另一种是发出一个故障消息，表明无法完成订单。

假定服务客户端和库存核查服务提供者都使用了专有的同步操作，开发相应的解决方案。

9.4　在前面一个练习的解决方案中，用异步操作取代所有的同步操作，要求最终结果一样。

9.5 开发一个简单的旅行预定 BPEL 流程。该流程将部分地基于练习 5.3、5.4 和 5.5，并涉及和合作伙伴(诸如航空服务、宾馆服务、汽车预定服务以及简单的信用卡核查服务)之间的交互。

9.6 使用 BPEL 开发一个简单的应用，该应用涉及一个信用卡持有者和一个电子化厂商。信用卡持有者向厂商发送一个消息，该消息包含订单指令和支付细节。然后，厂商向银行发出一个支付授权请求，从而确定订购请求中所包含的购买者的账户信息和财务状况是否有效。假如成功完成了支付授权，然后厂商将会收到一个支付认可响应，并将订购的货物发送给订购者。

第10章 事务处理

学习目标

在有些情况下，需要通过大流量的网络实现协作 Web Service，通过业务流程互相协作来实现共享的业务任务时，可能需要对事务性的支持。在将松耦合的服务加入内聚的工作单元并确保一致性以及可靠执行时，这类支持是非常需要的。然而，基于这类支持的传统原子型事务和扩展的事务模型有太多的限制，必须是短时间运行，且必须是同步的，而松耦合应用依赖长期运行的活动和服务，在时间和位置上都是分离的。

在本章中，我们着重讨论事务性 Web Service 的特征、需求和体系结构支持。通过本章的学习，读者将可掌握下列关键概念：

- 既支持集中式系统，又支持分布式系统的事务。
- 分布式事务体系结构。
- 分布式事务的并发控制和协同机制。
- 封闭的和开放的嵌套事务。
- 事务性工作流。
- 事务性 Web Service 的属性和模型。
- 与其他 Web Service 标准（诸如 BPEL、WS-Policy 和 WS-Security 等）一起协同工作的事务性 Web Service 标准和框架。

10.1 什么是事务

企业广泛使用事务处理系统来支持关键任务的应用。这些应用需要可靠地存储和更新数据，为大量用户提供数据的并发访问，并在单个系统组件出现故障时能维护数据的完整性。由于当今业务需求的复杂性，事务处理成为建立、部署和维护业务层分布式应用的最复杂和最重要部分之一。

事务指的是作为一个完整的工作单元的一系列操作，它们或者全部执行成功，或者失败，部分执行成功的部分将被取消。在一个事务中，通过将一组相关操作连接在一起，可以确保即使发生错误也能保证系统的一致性和可靠性。事务成功必须是事务中的所有操作都成功完成。

事务有起始和终止，事务可能横跨多个流程和多个计算机。应用程序必须能启动和终止事务，能规定是需要永久保存对数据的修改还是丢弃掉修改。标示一个应用程序的事务边界称为事务划分。

原子事务是由一组操作构成的计算，在失败和并发计算时都是不可分割的。即，要么所有的操作都成功，要么全部失败，并且其他并发执行的程序不能修改或看到计算的中间状态。当发生以下事情之一时事务终止：应用提交事务；由应用回滚事务或因失败回滚事务（如由于缺乏资源或违反了数据一致性）。如果成功提交事务，与此特定事务相关的修改就写回到持久性存储，新的事务可看到此修改。如果事务回滚，将丢弃所有的修改，就好像该事务从来没有发生过。

```
BEGIN TRANSACTION
  -- Perform transaction operations
     <operation1>
     <operation2>
          ...
     <operationN>

  -- Check for Errors
  If (Error) then
     --Rollback the Transaction
     ROLLBACK TRANSACTION

  -- Commit the Transaction
  COMMIT TRANSACTION

END TRANSACTION
```

图 10.1 事务结构

图 10.1 中显示了一段伪代码，begin transaction 语句开始一

个新的事务。事务可以通过使用 commit transaction 语句将修改提交到数据库，或者当错误发生时使用 rollback transaction 语句将所有的修改丢弃。有两类操作：从数据库读的操作以及在数据库中更新一些数据项的操作。

图 10.2 描述了事务的状态转换图。该图显示一个事务可有以下状态：

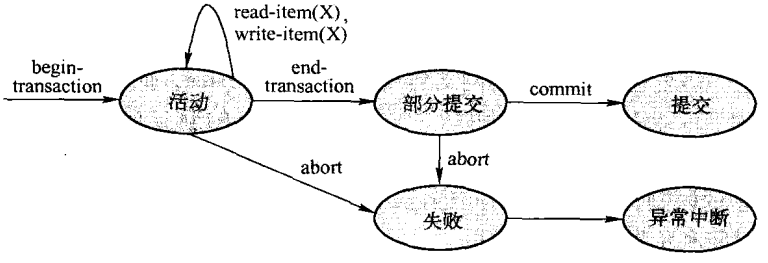


图 10.2 事务的状态转换图

- 活动(Active)：该状态表明事务正在执行一些工作(操作)。当事务开始执行时立刻就进入该状态。
- 部分提交(Partially committed)：事务执行了最终的语句。在这个阶段，需要测试恢复协议以确保系统错误不会导致事务的修改不能记录下来。如果检查成功，则任何更新都可以被安全地记录下来，事务可以转到提交状态。如果任何一个检查失败，则事务转为失败状态。
- 提交(Committed)：事务成功执行所有的工作并终止，将它的修改永久性地写入存储器，这称为已提交。
- 失败(Failed)：事务最初是活动状态的，但是不能提交。
- 异常中断(Aborted)：事务没有成功执行，所有执行的操作都回滚。这可能是由于恢复检查失败或事务在活动状态时异常中断了。

10.1.1 事务的属性

为了在事务边界内维护资源的一致性，事务必须体现 ACID 属性，即原子性(Atomicity)、一致性(Consisitency)、隔离性(Isolation)和持久性(Durability)。

*原子性*意味着事务是一个不可分割的工作单元：要么事务的所有操作都应用于应用状态，要么都不应用。如果事务不能成功完成，它会回滚到事务开始之前的状态。

*一致性*意味着事务必须正确地将数据从一个一致的状态转换到另一个一致的状态，并保持数据语义和引用的完整性。这意味着事务不能破坏应用中隐含的完整性约束。实际上，一致性的概念是与特定的应用相关的。例如，在一个统计应用中，一致性包括完整性约束，即所有资产账户的总和等于所有负债账户的总和。

当多个事务并发执行时，一个事务可能想去读或写另一个事务已修改但尚未提交的数据。除非后一个事务提交，这个修改应作为临时状态，因为可能会被回滚。隔离性要求几个并发事务产生的结果与他们以某种顺序(不指定)执行时的结果相同。事务访问的是共享数据库，可能相互间会有冲突，因此隔离性确保并发事务的执行是受控的、可调整的。并发事务执行的方式让它们看上去是独立执行的，但事实上它们可能和其他事务在同一数据库单元上并发执行。隔离性一般通过锁机制实现。

*持久性*意味着提交的更新是永久性的。在提交之后发生的错误不会导致数据的丢失。持久性也意味着所有已提交事务的数据在系统或介质故障之后可以恢复。

在操纵数据，将数据从一个或多个源转换到一个或多个目标，或协调多个事务活动时，原子事务是很有用的(更详细的信息，参见 10.2.2 节)。要么全部执行，要么全部不执行(原子性)确保在事务这个上下文中所有的数据修改，以及消息交换都能保持一致性，不管为完成这个事务需要多少步骤。

10.1.2　并发控制机制

当事务并发执行时，例如在金融和业务应用中，多个应用会同时访问同一个数据库，它们相互间会冲突并导致数据库变得不一致。这是有可能发生的，虽然事务本身保持数据库的一致性。

当两个或两个以上的事务并发执行时，它们对数据库操作是交错的。这意味着一个事务的操作会在另一个事务的操作之间执行，这就造成了干扰。这种交错执行会严重破坏数据的一致性，从而导致数据库进入不一致的状态。并发控制的目的就是对抗这种干扰，避免任何潜在的错误。

并发控制是管理数据资源竞争的一种方法。这种机制用于确保数据库事务以安全的方式执行(即不丢失数据)。并发控制在关系数据库和数据库管理系统中尤为有用，确保事务安全地执行，使它们遵循 ACID 规则。数据库管理系统必须能确保只允许串行的，可恢复的执行规划，并且当取消失败的事务时，已提交的事务动作不会丢失。

串行调度(*serializable schedule*)是一个调度 S，包含一组并发执行的事务，等效于某个(串行)调度 S_{ser}，即顺序地执行这组事务。通俗地说，在这两种调度中，相应的读操作返回的值是相同的，并且以相同的顺序更新每个数据项。

这意味着如果在 S 和 S_{ser} 中读操作返回的值相同，则这两个调度中的事务计算是相同的，因此事务将相同的值写回数据库。只要在两个调度中写操作的顺序相同，S 将等同于 S_{ser}。

串行等价应用于事务时，意味着当一组事务并发执行时，它们的效果于串行执行(一个接一个)的结果相同。这实际上给事务服务器提出了严格的要求，要确保不会发生不一致的更新。典型的问题包括脏读、丢失更新问题、更新不一致以及级联异常终止问题[Ullman 1988]。

传统的数据库系统并发机制可通过锁的方式管理。当并发控制机制使用锁时，除了读写数据项外，事务必须请求和释放锁。在最底层，锁可以分为(以限制性递增的顺序)共享、更新和排他锁。共享(或读)锁表示在同一数据上另一个事务可以使用一个更新或另一个共享锁。当读数据时使用的是共享锁。如果一个事务用排他(写)锁所住了数据，这表示它要写数据，其他事务不能在这个数据上加锁。一些数据库系统提供更新锁，用于一个事务开始时读数据，但是之后又想更新它。更新锁允许事务读，但是不能写数据项，并表示该锁可能以后会升级为写锁。更新锁确保另一个事务在同一个数据项上只能加共享锁。为获取事务一致性，必须遵循以下两条规则[Garcia-Molina 2002]：

(1) 如果一个事务先前在一个数据项上请求了一个锁并且尚未释放，则它只能读或写这个数据项。

(2) 如果事务对一个数据项加了锁，它必须在以后对它解锁。

业务系统中的大多数并发控制机制用一个严格的两阶段锁协议[Eswaran 1976]来实现串行化。这个协议使用加在数据库中数据项上的锁，并要求事务在访问数据库中特定数据项进行读或写之前需要获取合适的锁(读或写)。一旦一个事务需要读(或写)一个数据项，在它能执行操作之前必须获得读(或写)锁。只有在事务提交或中断之后，才能释放事务的锁。这种方法的缺点是一些锁维持的时间可能比需要的时间更久；但是它可以防止各种异常。有关使用严格的两阶段锁协议的并发控制机制只能产生串行调度的证明可参阅文献[Ullman 1988]。

事务隔离性的实现方法是在访问的数据项上加锁直至事务完成。有两种常用的通过加锁管

理并发控制的机制：悲观机制和乐观机制。这两种模式都是必须的，因为当一个事务存取数据时，它修改(或不修改)数据的意图并不是显而易见的。为了避免多个应用同时更新数据库，在数据访问事务的开始就将给定的数据资源锁定，并且直到事务结束才释放，这种并发控制模式认为是悲观的(参见“并发控制”一节)。乐观的并发模式是基于这样的假设——数据库事务大多数情况下不和其他事务发生冲突，尽可能地允许事务执行。在乐观并发控制模式下，不能获取控制锁，这样可以允许最大数量的读并发，并且读在写之前执行以确保正在使用的数据项不会被中途修改。在乐观并发控制模式中有三个阶段：读阶段、验证阶段和写阶段。在写阶段，所需的数据项从数据库中读出，写操作在这些数据项的本地拷贝上执行。验证阶段执行串行化检查。当事务提交时，数据库检查该事务是否可能会与其他并发事务冲突。如果可能会冲突，则事务异常中断，然后重新启动。否则提交事务，该事务修改的数据项写回数据库(写阶段)。如果只有少量的冲突，将能比较有效地进行验证，性能与其他并发控制方法相比更好。但是如果有许多冲突的话，反复重启事务的开销会严重影响性能。乐观并发的目标是使得特定资源不可用的时间最小化。这对于长时运行的事务尤为重要，因为在悲观模式下，会长时间锁定一个资源，这是不能接受的(更多细节，请参阅“长事务”一节)。

10.2 分布式事务

在当今复杂的分布式环境下，事务对于许多分布式操作来说是至关重要的。例如，在两个分布式应用组件之间交换的一系列消息的传送和排序。在这种情况下，应当在一个原子执行序列中进行消息交换。

原子事务大大简化了分布式应用的代码。在失败存在的情况下，原子事务是构建可靠的分布式系统的机制。正如我们先前已经解释过的，它们提供了两个重要的属性：可恢复性和串行性。可恢复性意味着事务的动作表现出“全部都做或全部都不做”的行为：一个动作要么执行到结束，在这种情况下事务提交；或者事务异常中断，在这种情况下对数据库的稳定的状态毫无影响。串行性意味这并发执行事务的效果等同于顺序执行这些动作的效果。因此，可恢复性保护事务免受失败影响，而串行性则通过分别考虑每个动作的效果来判断并发性。

通常由一些底层的系统基础架构(通常以产品的形式出现，如事务处理监视器[Gray 1993])提供事务语义。基础架构将处理各种失败，执行必要的恢复动作以确保原子性。程序员无须处理大量可能的失败情况，从而可以极大地简化了应用程序。

分布式事务处理提供了必要的机制，可将多个软件组件组合到一个协作单元。协作单元能够维护共享数据。并且协作单元可能横跨多个物理处理器或位置。这样可以构建使用多个产品来统一操纵数据的应用，并可以很容易地通过添加其他硬件和软件组件来进行扩展。

分布式事务可以包含多个操作，并且这些操作至少涉及两个网络。分布式事务可以横跨异构的事务数据资源，并且可以包括大量的活动，诸如从 SQL Server 数据库获取数据、从消息队列服务器读取消息以及向其他数据库写数据。分布式事务访问和更新两个或多个网络化资源上的数据，因此必须在那些资源间进行协调。一些软件能够简化分布式事务的编程，如 TP 监控器能在多个数据资源间协调提交、异常中断行为及恢复。

10.2.1 分布式事务体系结构

在分布式事务处理模型中，和我们讨论相关的组件有：应用程序、应用服务器、资源管理器、资源适配器和事务管理器。

最简单形式的分布式事务处理只包含应用程序、资源管理器和应用服务器。应用程序实现终端用户企业所需的功能，如订单处理应用。每个应用程序指定涉及共享资源(如数据库)的一

系列操作。应用程序定义事务的开始和结束，在事务边界内访问资源，还需还决定是提交还是回滚每个事务。

资源管理器提供并管理对共享资源的访问。可以通过资源管理器提供的服务进行访问。资源管理器的例子有数据库管理系统、文件访问方法(如 X/Open ISAM)，以及打印服务器。为简化讨论，本节我们着重讨论关系数据库的例子，相应的资源管理器即为关系数据库管理系统(RDBMS)，如 Oracle 或 SQL Server。所有实际的数据库管理系统都由这个组件处理。

资源适配器(参见 2.7.1 节)是一个软件组件，允许应用组件访问特定资源的资源管理器(如关系数据库)，并与之交互。因为资源适配器对应于特定的资源管理器，通常每类数据库或企业信息系统都有不同的资源适配器，如企业资源计划或遗流系统(管理事务、外部功能和数据)。资源适配器是"外部世界"之间的通信渠道或请求转换器，在这里为事务应用和资源管理器。

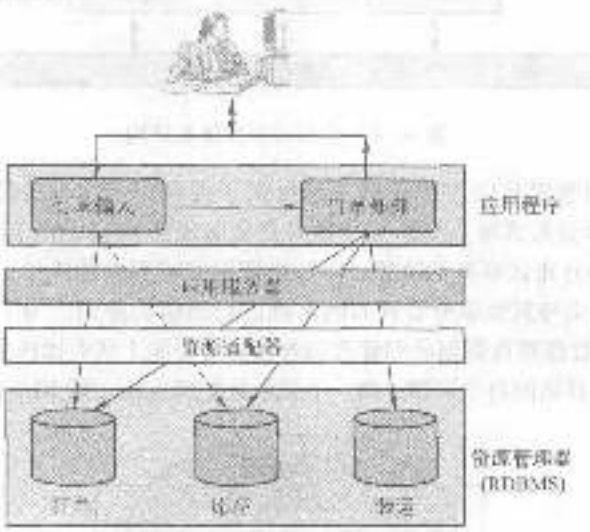


图 10.3　只涉及本地事务的事务体系结构

资源管理器内部管理的事务称为本地事务。如果单个资源管理器参与一个事务，应用服务器通常让资源管理器内部协调该事务。应用程序(客户端)通过一个应用服务器向资源适配器发送数据请求。应用服务器处理大量的应用操作，并代表客户发起事务。应用服务器处理"后端"活动如网络连接、协议协商，以及其他高级功能(如事务管理、安全性、数据库连接池等)。然后资源适配器转换该请求，并通过网络将之发送到 RDBMS。RDBMS 向资源适配器返回数据，资源适配器再将结果返回给应用。图 10.4 阐述了这种情况。一个应用向三个关系数据库(订单、库存和装运)发出事务请求(如订单输入和订单处理)，这三个关系数据库位于一台服务器上并由一个资源管理器管理。实线箭头表示对数据库的读和写，虚线箭头表示一个读操作。

上面的例子阐述了单个本地事务，并描述了分布式事务模型中涉及的两个组件只在分发事务时，才加以考虑第三个组件——事务管理器。事务管理器是客户端和/或应用服务器和分布式事务功能之间的中介。

在企业计算中，经常会涉及几个网络资源(宿主)，每个都有不同的服务器，例如 Web 服务器、几个不同的 RDBMS、EJB 服务器，或 Java 消息服务(JMS)服务器。分布式事务涉及各种资源管理器之间的协调，这是事务管理器的功能。基于事务划分，分布式事务可绑定到分布式应用组件。

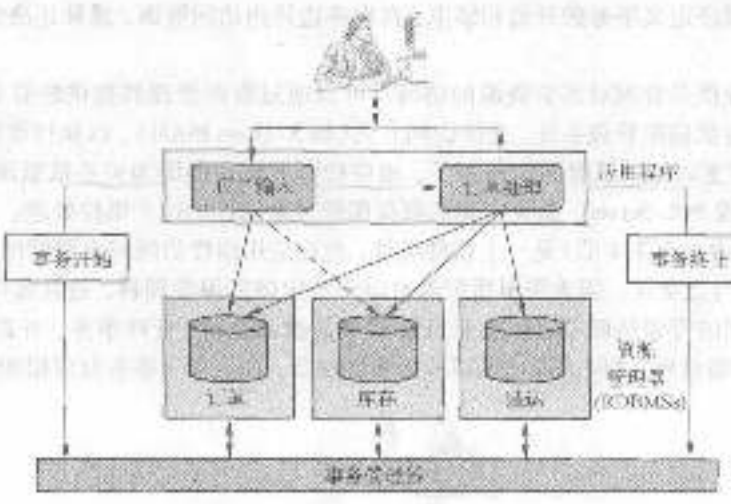


图 10.4 分布式事务体系结构

事务管理器与应用程序和应用服务器一起提供了控制分布式事务范围和持续时间的服务。事务管理器还帮助协调分布式事务横跨多个事务资源管理器(如 DBMS)的实现，提供事务同步和恢复的支持，协调启动分布式事务的决策，以及协调提交或回滚的决策。这确保了原子事务的完成。事务管理器的能够提供与其他事务管理器的实例进行通信的能力。为了满足客户对于原子化的操作序列的要求，事务管理器负责创建和管理包含在给定资源上所有操作的分布式事务。事务管理器通过各自的资源管理器访问每个资源，如一个关系数据库系统。图 10.5 阐述了这种情况。

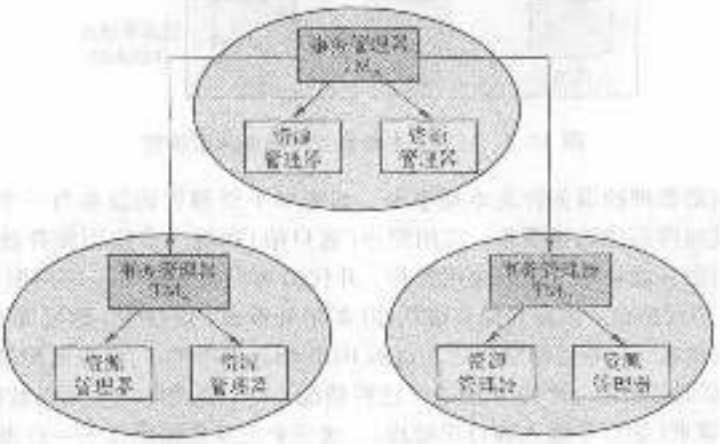


图 10.5 进行协作的事务管理器

在这个环境中，资源管理器提供了两组接口：一组用于应用组件，获取连接并对数据上执行操作，另一组用于事务管理器，参与两阶段提交和恢复协议。此外，资源管理器能够直接或间接地用事务管理器注册资源，使得事务管理器可以跟踪参与事务的所有资源。这个过程称为资源加入(resource enlistment)。对于资源管理器执行的事务工作，事务管理器使用这个信息协调这些事务工作，并驱动两阶段提交和恢复协议。

当两个或更多的事务管理器在一个分布式事务中协作时，代表应用程序发出请求的事务管

理器被指定为上级事务管理器，称为根事务协调器(root transaction coordinator，也称根事务协调者)，或简单地称为根协调器(root coordinator)。随后加入的或为一个已有事务创建的(如由于介入的原因，参见 10.4.4 节)任何事务管理器成为这个流程的下级。这种协调器称为下级事务协调器(subordinate coordinator，也称下级事务协调者)，或简单地称为下级协调器，并通过注册资源成为事务参与者。可恢复的服务器总是事务参与者。

上级—下级事务管理器关系背后的原理是：上级系统通过使用两阶段提交协议驱动所有下级系统。每个事务管理器可以有许多下级事务管理器，但是最多只有一个上级事务管理器，参见图 10.5。这些上级和下级关系形成一个树形关系，称为事务提交树。提交树的根事务协调器作为整个分布式事务的全局提交协调器。对于一个上级可以有多少个下级协调器，以及在根协调器和最底层的叶子下级之间有多少中间层次，并没有固定的限制。提交树的实际创建依赖于应用的行为和需求。

加入的资源管理器也是这个提交树的成员。他们通常是本地事务管理器的下级。上下级关系只相应于一个特定的事务。也就是说，在一个特定的事务中，某个事务管理器可以是另一个事务管理器的上级，但是在另一个不同的事务中角色可能会互换。

当一个分布式事务提交或异常中断时，准备、提交和异常中断消息在提交树上向外流动。在第一阶段期间，树上的任何一个节点在接受发向它的准备请求之前的任何时刻，都可以单方面地终止事务。在节点准备好之后，它保持在准备状态直至它的提交协调器指示它提交或终止事务。全局提交协调器决定提交或终止整个事务，从不处于不确定状态。

系统或通信失败会使事务延长不确定状态的时间。当事务处于不确定状态，由该事务修改的资源保持锁定，其他人不可用。这些情况下需要人为干预，通常依赖于管理工具来解决处于不确定状态的事务。

事务应用开始一个事务的典型情况是：由客户应用向事务管理器发出一个请求来启动一个事务。事务管理器启动一个事务并将之与调用的事务分支相关联。事务分支与分布式事务中每个资源管理器的请求相关联。尽管最终的提交/回滚决策将分布式事务作为单个逻辑单元，可能会涉及许多事务分支(线程)。例如，对不同 RDBMS 的事务请求导致相同数目的事务分支。由于多个应用组件和资源参与一个事务，事务管理器有必要建立和维护事务的状态。这通常以事务上下文的形式实现。事务上下文涵盖事务期间在事务资源上执行的所有操作，将事务操作与资源关联起来，并提供调用这些操作的组件信息。从概念上说，事务上下文是一个数据结构，包含一个唯一的事务标识、一个超时值，以及对控制事务范围的事务管理器的引用。事务管理器将事务上下文与当前执行的线程关联起来。因此，在事务期间，所有事务中参与的线程共享事务上下文，并与将事务工作划分为并行任务的同一事务上下文关联起来。另外，如果一个事务横跨多个事务管理器，该上下文还必须从一个事务管理器传递到另一个。

10.2.2　两阶段提交协议

为保证事务的原子性，分布式事务拓扑中的每个事务分支必须由自己本地的资源管理器提交或回滚。事务管理器控制事务的边界，并负责最后的决策：整个事务是否应提交或回滚。这个决策在两个阶段做出，管理它们的协议就是众所周知的两阶段提交协议(2PC)。

2PC 是协调横跨两个或多个资源管理器的单个事务的方法。事务更新在所有参与的数据库中提交，或回滚并恢复到事务开始之前的状态，从而确保了数据的完整性。换句话说，要不所有参与的数据库都被更新，或没有一个被更新。在 2PC 中，我们假定一个节点上的事务协调器(也称为协调者)在决定整个分布式事务是否能提交中起着特殊作用。这个协调器通常与分布式事务起源的节点相关。

1. 第一阶段：准备

在第一个阶段：

（1）在2PC的初始阶段，分布式事务T的协调器决定何时试图提交该事务。为实现该功能，它首先通过发出准备消息 < prepare T > 对该分布式事务T中涉及的所有资源管理器（RDBMS）发起投票，查看是否每个资源管理器读准备好提交了。这个准备消息通知每个资源管理器准备提交。接收到消息 < prepare T > 的每个节点上的资源管理器决定是提交它的T组件还是否决事务提交操作。如果一个资源管理器不能提交，它就回复不能提交，并终止事务的特定部分，从而使数据不被修改。如果一个节点的事务组件尚未完成它的动作的话可以推迟，但是最终必须回复 < prepare T > 消息。

（2）如果节点决定提交它的组件，它就进入预提交状态。在这个状态期间，资源管理器必须执行所有必要的动作以确保T的这部分即使在系统故障的情况下不会异常中止。然后这个节点的资源管理器相协调器转发 < ready T > 消息。如果资源管理器决定终止这部分，则向协调器转发 < donot commit T > 消息。必须注意一旦一个节点处于预提交状态，在没有协调器明确的指示下，它不能终止T的这部分工作。

2. 第二阶段：提交/中止

在第二阶段（提交/中止），协调器决定该事务的命运。如果所有资源管理器投票提交它们的事务（通过发送 < ready T > ），协调器通过向事务中所有资源协调器发送 < commit T > 消息来提交整个事务。最终，协调器向应用程序返回结果。如果有资源管理器发送 < donot commit T > 消息，则协调器向事务中所有的资源管理器发送 < abort T > 消息。这使得整个事务回滚。万一有资源管理器既没有回复 < commit T > 也没有回复 < donot commit T > 消息，在一个超时时间之后，协调器就假定该节点回复了 < donot commit T > 消息。

图10.6阐述了2PC。该图显示了两个RDBMS两阶段提交操作期间的事件序列。在这个流程中，首先应用程序发起提交事务的调用。一旦应用发起提交请求，事务管理器就为提交操作准备所有的资源（在这个例子中是RDBMS1和RDBMS2）（通过发起投票）。资源管理器给出回复（在这个例子中两者都提交）。最终，资源管理器分析收到的投票，根据所有的资源是否准备提交，向所有的资源发出提交请求或回滚请求。在图10.6中，向两个资源管理器发出提交操作。

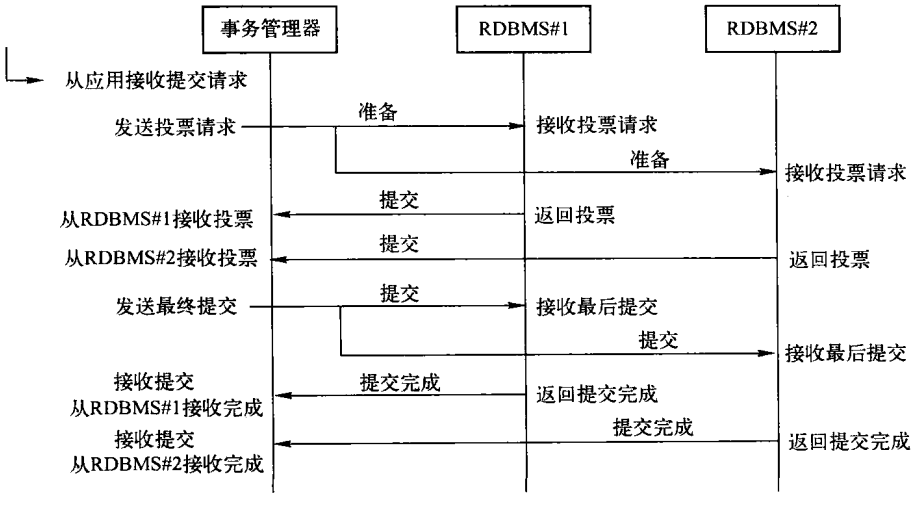


图10.6 分布式事务的2PC

10.3　嵌套事务

分布式事务是由将不同服务器上的事务结合到单个事务单元的需求发展而来的。这是基于这样的事实，经过若干年，企业已经为业务功能的自动化开发了大量的专用应用和事务处理系统，如订货、库存控制、薪金册等。在许多情况下，这些系统是在不同的硬件平台和 DBMS 上独立开发的。

事务管理器可以使用不同的实现模型为应用提供事务性支持。最常见的事务模型是我们在 10.1 节描述的。这是一个经典的事务模型，包含在单个服务器上的一个数据库，并且没有内部的结构，通常称为平面事务(flat transaction)。遵循平面事务模型的事务管理器不允许事务在其他事务中嵌套。图 10.7 说明了在三个不同服务器 S1，S2 和 S3 中访问数据项(通过各自的资源管理器)的平面事务。在平面事务模型中，正确地控制跨多个事务服务的事务范围的唯一方法是在如何为一个业务活动组合这些服务上事先达成一致意见，并根据约定为这些服务应用恰当的事务策略。

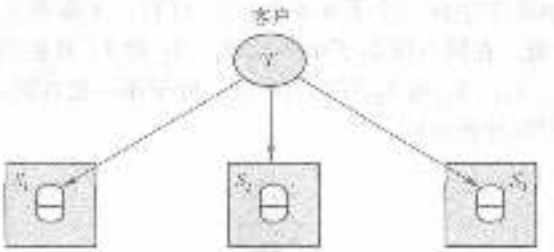


图 10.7　平面事务的例子

当增加自动化需求时，企业总需将已有的事务系统组合成新的事务应用。因此就需要将分布式事务作为模块，可自由地组合多个服务器上导出的已有的事务。在分布式事务中，事务服务器控制每个导出的(本地的)事务功能。每个提交/中止决定都由各自的本地事务控制。这使事务设计者不用控制分布式事务的结构。作为这种自底向上方法的结果，事务可能不能清晰地分解应用的功能[Kifer 2005]。

嵌套事务的发展源自设计复杂功能和实际需求，即允许事务设计者能自顶向下地分解事务。嵌套事务模型允许独立地构建事务服务，然后组合进应用之中。每个服务可以决定它的事务边界的范围。编制服务组合的应用或服务控制顶层事务的范围。该模型中的子事务是在另一个事务中开始的一个事务。当新事务在已有事务范围内的会话上建立时，即产生这种情况。这个新的子事务称为在已有的(父)事务内(或之下)的嵌套。嵌套事务允许应用创建一个嵌入在已有事务中的事务。已有的事务称为子事务的父亲，子事务称为父事务的孩子。在同一个父事务中可以嵌套多个子事务。同一个父亲的孩子称为兄弟。

一个事务的祖先是子事务的父亲以及(递归地)父亲的祖先。一个事务的后代是该事务的孩子以及(递归地)它的后代的孩子。顶层事务没有父亲，参见图 10.8。顶层事务和它所有的后代称为事务家族。代表子事务所作的修改要么

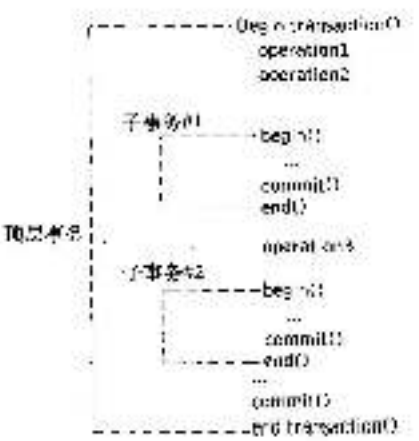


图 10.8　嵌套事务的结构

全部提交要么回滚，这方面与事务和顶层事务是类似的。但是当子事务提交时，在事务所有祖先提交之前修改保持临时状态。级数可在任意深度发生；可用事务树描述生成的结构。

与分布式事务类似，子事务具有可恢复性和串行性两形，并控制它们的提交/中止决定。但是处理这种决定是相当困难的。和分布式事务模型能要么全做要么都不做的方法不同，事务树中的单个子事务可以独立地失败而不中止整个事务。子事务内的修改对于父事务来说是不可见的，除非子事务提交。这提供了一种检查点机制：在启动孩子之前，父事务所做的工作不会因为孩子中止而丢失。在图 10.8 中，只有当最外层的 commit( ) 被调用时，完成的操作才会提交给数据库。内层的 commit( ) 方法不会单独操作到完成。

嵌套事务有许多模型，可分为两大类：关闭的事务模型和开放的事务模型。

10.3.1 封闭嵌套事务

Moss_Moss 1985_提出的封闭嵌套模型把顶层事务和它的所有子事务看作一个事务树，参见图 10.9。树的节点代表事务，边代表相关事务间的父/子关系。术语父亲、孩子、祖先、后代具有通常的含义。在这个嵌套事务模型中，顶层事务可以向下生成任意深度的嵌套子事务。图 10.9 显示了一个客户端顶层事务 T 生成三个子事务 T_1、T_2 和 T_3。子事务 T_1、T_2 和 T_3 分别在服务器 S_1、S_2 和 S_3 访问数据对象。在同一层的子事务如 T_1、T_2 和 T_3 可在不同的服务器上并发运行。没有孩子的子事务如 T_{11}、T_{12}、T_{21}和T_{22}等称为叶子。叶子不一定在同一层次上。图 10.9 中在叶子层次的六个子事务也可以并发运行。

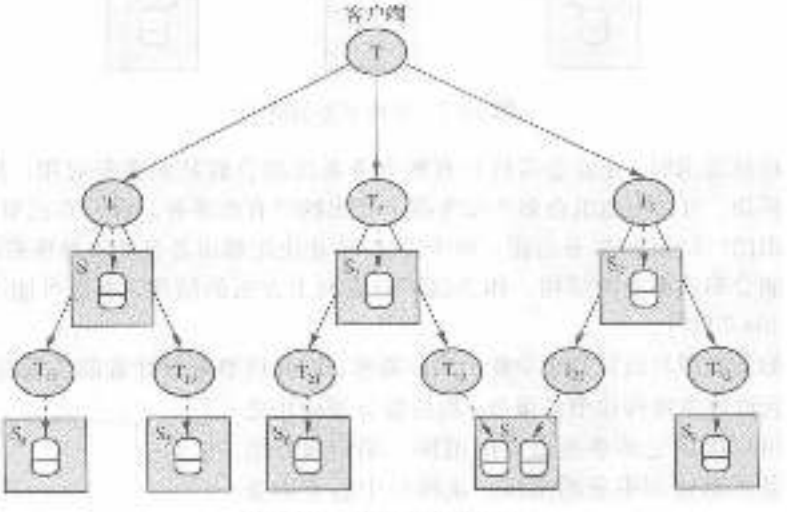


图 10.9 嵌套事务的例子

嵌套事务模型的语义总结如下[Kifer 2005]：

(1) 父亲顺序创建孩子，因此一个孩子在下一个产生之前结束。或者，父亲可以指定它的一些孩子可以并发运行。图 10.9 中的事务树结构并没有区分并发运行或串行运行的孩子。父节点不和它的孩子并发执行。它等待直到同一层次的所有孩子都结束，然后再开始执行。然后可能再生成其他孩子。

(2) 对于子事务的并发兄弟，子事务和它所有的后代作为独立的隔离单位执行。例如，如果 T_1 和 T_2 并发运行，T_2 将子树 T_{11}、T_{12}看作独立的隔离事务，并不观察它的内部结构或干扰它的执行，对于 T_1 和子树 T_2、T_{21}、T_{22}来说同样如此。因此兄弟间是可串行的，它们并发执行的

效果等同于以某种顺序串行执行。

（3）子事务是原子的。每个子事务可以独立地提交或中止。子事务的提交和效果的持久性依赖于父亲的提交。当所有祖先（包括顶层事务）提交时，子事务提交并且结果成为持久的。此时，整个嵌套事务称为已提交。如果一个祖先中止，则所有它的后代都中止。

（4）如果子事务中止，则它的操作无效。控制返回给父亲，父亲再做恰当的动作。在这种方式中，一个中止的子事务可能会影响数据库的状态，因为它会影响它的父亲改变执行路径。这种情况同平面事务不同。在平面事务中，中止的事务不能改变事务协调器的事务路径。

（5）子事务无须一致。但是，嵌套事务作为一个整体是一致的。

1. 嵌套事务中的两阶段提交协议

嵌套事务的两阶段提交协议（2PC）操作类似于分布式事务。唯一的区别是子事务涉及的服务器可以决定中止或临时决定提交事务。临时提交与准备不同。它只是一个执行 Shadow write（这是一个写操作，更新实际数据的临时复制）的本地决定。

在嵌套事务的2PC 中，一个客户端通过打开顶层事务（使用 openTransaction( )操作）启动一组嵌套事务。这个操作为顶层事务返回一个事务标识。客户端通过调用 openSub-transaction( )操作启动一个子事务，该操作的参数通过它的事务标识指定父事务。新的子事务自动加入父事务，并返回子事务的事务标识[Coulouris 2001]。鉴于子事务标识的组成方式，可以根据子事务标识来识别它的父亲的事务标识。子事务标识是全局唯一的。一个事务的事务管理器（协调器）提供打开子事务的操作，以及激活子事务协调器的操作，获取父亲的状态，即它是否提交或中止。

客户端通过在顶层事务的协调器上调用 closeTransaction( )或 abortTransaction( )来完成一组嵌套事务。同时，每个子事务执行它的操作。当它们完成时，管理子事务的服务器记录子事务是否临时提交或中止的信息。注意，正如前面已经讨论的，如果子事务的父亲中止，则子事务强制中止。只有当事务所有的后代的状态都检查了，才会发生真正的提交操作。子事务将等待，直到包含它们的整个事务提交。因为子事务遵守事务的语义，它们可以中止而不导致父事务中止。父事务可能包含处理其任一子事务中止的代码。例如，一个中止事务的父事务可以决定向拥有复制数据的服务器转发一个新的子事务。

当顶层事务结束时，它的协调器执行 2PC。一个参与的子事务不能完成的唯一原因是：完成临时提交后彻底失败了。每个父事务的协调器拥有所有后代子事务的列表。当一个子事务临时提交时，它向父事务汇报它的状态以及它后代的状态。最终顶层事务接收树中所有子事务的列表以及每个的状态。列表将忽略中止子事务的后代。

在 2PC 中顶层事务起协调器的作用。参与者列表包含树中所有子事务的协调器，这些协调器已经临时提交但是没有中止的祖先。在这个阶段，应用程序中的业务逻辑决定顶层事务是否能不管中止的子事务提交树中剩下的事务。协调器要求参与者对事务的结果进行投票。如果投票提交，它们必须通过将事务上下文保存在永久存储设备中来准备事务。上下文记录在顶层事务中最终作为它的一部分。

2PC 的第二阶段与非嵌套的情况相同。协调器收集投票，然后根据结果通知参与者。结束时，协调器和参与者将提交或中止各自的事务。

图 10.10 阐明了嵌套事务的 2PC。该图显示顶层事务包含三个子事务 T_1、T_2 和 T_3，如图 10.9 所示。子事务 T_1、T_2 和 T_3 分别在服务器 S_1、S_2 和 S_3 存取数据对象。该图显示了当每个事务标记为临时提交或中止时各子事务的状态。在图 10.10 中，由于事务 T_2 中止，决定事务 T 是否提交就基于 T_1、T_3 能否提交。这些事务的每个子事务也是要么临时提交要么中止。例如，T_{11} 已中止，而 T_{12} 临时提交。T_{12} 的命运依赖于它的父亲 T_1，最终由顶层事务决定。T_2 已中止，则整

个子树 T_2、T_{21}、T_{22}必须中止，不管 T_{21} 已临时提交了。子树 T_3、T_{31}、T_{32} 也依赖于顶层事务提交。假定 T 决定提交，而不管 T_2 已中止。

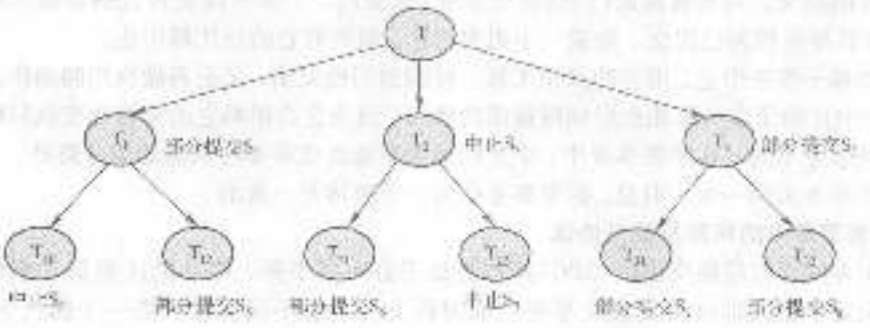


图 10.10 图 10.9 中样例的两阶段提交协议示例

表 10.1 给出了图 10.10 例中每个协调器保存的信息。注意，由于 T_{22} 和 T_{32} 都运行在服务器 S_7 上，它们共享一个协调器。子事务 T_{21} 和 T_{22} 称为孤儿，因为它们的父亲 T_2 已中止。事务中止对并不将它的后代信息传递给父亲。因此 T_2 不把有关 T_{21} 和 T_{22} 的任何信息传递给顶层事务 T。一个中止事务的临时提交了事务必须中止，不管顶层事务决定提交与否。在图 10.9 中，T、T_1、T_{12}、T_3、T_{31} 和 T_{32} 的协调器是参与者，在 2PC 协议最终将要求投票决定结果。

表 10.1 嵌套事务的协调器保存的信息

事务协调者	孩子事务	父亲者	临时提交列表	中止列表
T	T_1、T_2、T_3	是	T_2、T_{21}、T_3、T_{31}、T_{32}	T_{11}、T_{12}
T_1	T_{11}、T_{12}	是	T_{11}、T_{12}	T_{11}
T_2	T_{21}、T_{22}	是(中止)		T_{21}、T_{22}
T_{21}		是	T_{21}、T_{22}	
T_{22}		是(中止)		T_{22}
T_{12}		是	T_{12}	
T_{31}		否(父亲中止)	T_{31}	
T_{32}、T_3		否，是 T_{32} 不是	T_{32}	T_{32}
T_{32}		否	T_{32}	

2. 并发控制

在封闭嵌套方式中，与两阶段锁中一样需要锁，但还需要额外的策略根据相互之间的关系决定了事务的行为。更精确的，顶层事务的几个子事务可能并发执行并请求有冲突的数据库应用。这需要额外的规则来确定如何向单个(嵌套)事务给子事务加锁。

封闭嵌套事务由以下规则控制[Kifer 2005]：

(1) 每个事务必须完全隔离，从而与其他嵌套事务串行化。

(2) 父事务不能和孩子并发执行。

(3) 每个孩子子事务(包括它所有的后代)必须隔离，因此可与其他兄弟(以及所有它的后代)串行化。

嵌套事务锁的规则实现了先前的步骤，如下所示[Gray 1993]：

（1）如果嵌套事务 T 的子事务 Ti 需要在共享数据项上加一个读锁，该锁可以授予的前提是在这个数据项上其他嵌套事务没有施加写锁，并且在这个数据项上施加写锁的 T 的任何子事务都是 Ti 的祖先(因此不会执行)。

（2）如果嵌套事务 T 的子事务 Ti 需要在共享数据项上加一个写锁，该锁可以授予的前提是在这个数据项上其他嵌套事务没有施加读或写锁，并且在这个数据项上施加读或写锁的 T 的任何子事务都是 Ti 的祖先(因此不会执行)。

（3）子事务获取的所有锁都保持直至该子事务中止或提交。子事务一结束，它所获得的并且父亲没有的任何锁都释放。只有在那时兄弟需要访问相同的数据时才能继承这些锁。没有这个特征，相同父亲的孩子之间就可能互相堵塞。当子事务中止时，它获得的而且父亲没有拥有的任何锁都释放。

总结先前讨论的嵌套事务模型，ACID 属性只应用于顶层事务。子事务对于其他事务来说是原子性的，可独立地提交和中止。子事务的中止不影响不属于该子事务层次的事务结果，因此子事务可用作防火墙，防止外部受到内部故障的影响。封闭嵌套事务模型引入了并发控制模式，这一模式确保了子事务的隔离执行以及并发嵌套事务调度的串行性。

10.3.2　开放嵌套事务

许多行业都涉及业务流程，并可能涉及事务性服务如金融、物流和运输。这些流程执行有关商品、器械和服务方面的事务，如产品提交、贸易结算以及服务供应。工作流业务应用通常依赖于协作活动，导致业务流程间复杂的互操作，经常会产生重要的相互依赖关系。此外，它们经常通过嵌套简单的流程来定义复合流程。因此，这些嵌套流程的事务性执行需要支持嵌套事务。这类应用必须跨各协作/事务方协调它们的更新操作，以便能够生成一致的结果。但是，对于时间和位置都分离的活动/服务的应用来说，传统的(ACID)事务以及基于 ACID 事务的嵌套事务模型还是限制过多。传统的事务模型为每个事务提供 ACID 保障，对于同步的以及只牵涉少量参与方的应用来说是理想的，如客户从银行取款、客户定购货物、旅行者预定机座，或者股票交易者完成一个购买事务。这些应用通常很简单，持续时间相对较短。

封闭嵌套事务模型不能处理基于流程的应用，因为它严格遵循传统的串行性范式来控制网络范围的事务管理，并提供全局完全的隔离。业务流程通常涉及长事务，不能用传统的 2PC 实现。因为这个协议需要在事务的整个生命周期期间将事务资源锁定。2PC 不适合需要花费数小时、数天、数周甚至数年时间来完成的事务。封闭嵌套事务模型的故障原子性要求发生故障时所有的工作都要回滚。这个需求对于长事务是不合适的，因为很多已做的工作可能会因为故障而丢失。此外，长时间的事务通常在执行期间要访问许多数据项，这些数据项知道事务提交才能被释放。因为长事务运行时间长，会导致访问相同数据项的短事务需要等待相当长的时间(因为数据项保持被锁定的状态)。这就增加了拒绝服务攻击的可能性，并导致系统失效。显然在这类应用中，需要放松隔离属性(控制锁)。

对传统的封闭嵌套事务模型有几个扩展，通常称作为开放的嵌套。即通过放松传统事务模型[Elmagarmid 1992]的原子性和隔离性属性来增加事务的并发性和吞吐量。开放的嵌套模型通常涉及一些可通过 2PC 自动回滚的协作事务，以及一些扩展事务。这些事务对补偿事务进行显式定义。当由于系统故障导致扩展事务不能提交时，可调用这些补偿事务。

大多数开放嵌套事务基于 saga 模型[Garcia-Molina 1987]的变体。Saga 是为用于服务长事务而引入的事务模型。一个 saga 包含一组独立的子事务 T_1，T_2，…，T_n(称为组件事务)。每个子事务是传统的(短)事务，维持 ACID 属性并能与插入其他事务。子事务使用传统的并发控制机制，如锁。子事务 T_1，T_2，…，T_n 以预定的顺序执行，并可能与其他 saga 的子事务交错

运行。

Saga 是以图形的方式组织的[Garcia-Molina 2002]，包含子事务节点或特殊的中止和结束节点。图中的弧连接节点对。特殊的中止和结束节点是终端节点，没有从它们中离开的弧。Saga 开始的节点称为开始(start)节点。导向中止节点的路径代表了导致所有事务回滚的子事务序列。这些序列的子事务应当使数据库的状态保持不变。到结束节点的路径代表了子事务的成功序列。这些子事务的效果是永久性的。

Saga 中的每个子事务都有一个相关的补偿事务 CT_i。补偿事务语义上将各自子事务的效果消除。在逻辑上，补偿是失败事务效果的逆。补偿是事情出现问题或当计划改变时所作的修补动作。如果一个子事务中止，则整个 saga 中止，以和相关子事务提交顺序相反的顺序执行补偿事务。

因为 saga 的补偿事务可能会与其他 saga 的补偿事务交错，一致性就被破坏了。此外，一旦补偿事务完成执行，允许它向其他事务提交或释放它的部分结果，从而放宽了隔离属性。除此之外，saga 保留原子性和持久性属性。

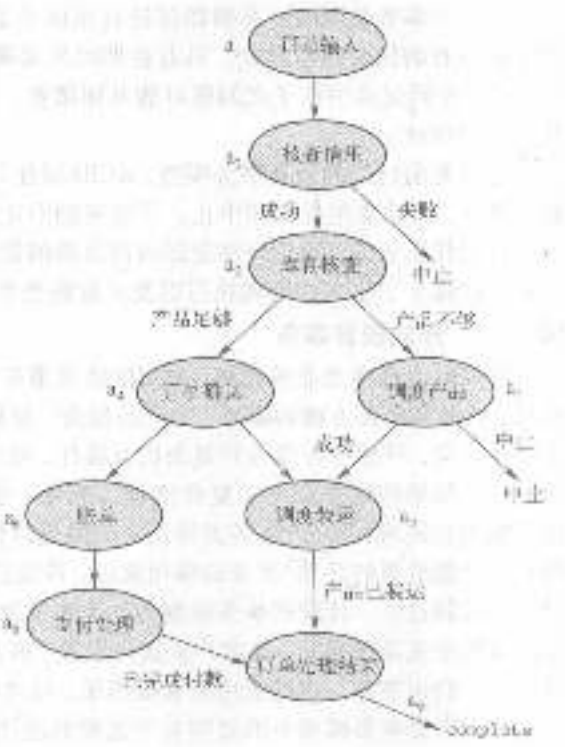


图 10.11 订单处理应用的简化事务工作流图

本书先前曾使用过一个订单处理例子，图 10.11 描述了这一例子的简化的事务流。如果仓库中没有足够的数量来完成一个订单，则可调度产品。事务工作流显示：一旦客户付款并且已送货，则订单完成，工作流结束。图 10.10 中的工作流由一组相关的活动组成。一个活动是一个(分布式)工作单元，可以是事务性的也可以不是。在一个活动的生命周期期间，它可以有事务性能或非事务性的阶段。一个活动先创建、执行，然后结束。结束活动的结果是它初输出，可用于决定其他活动的后续控制流。由于活动可以长时间运行，因此可以挂起，之后再恢复。

在图 10.11 中，到中止节点的路径包括活动 a_1，a_2 以及 a_1，a_2，a_3，a_5。在图 10.11 中，到结束节点的路径包含活动 a_1，a_2，a_3，a_4，a_5，a_6，a_7，a_8，a_9。若要理解补偿事务是如何工作的，请考虑活动 a_7(调度运输)和 a_9(订单完成)。a_9(订单完成)假定如果订单送货了，它也就成功发送。但是，万一运输由于故障、坠毁等原因没有到达，则 a_7(调度运输)需要取消(通过发起一个补偿活动来取消调度运输活动的效果)。在这种情况下，应用中的业务逻辑自动重新发起一个新的调度运输活动，从而使货物能安全到达目的地。事实上这是图 10.11 描述的工作流方案的假定。

1. 事务型工作流

目前已经提出了几种 saga 变体。一个变体要求子事务串行执行，而其他变体则允许子事务的并发执行。其他变体采用一个前向恢复策略，当子事务中止时，其中剩下的子事务执行。而其他变体采用后向恢复策略，原有已提交的事务执行补偿事务。开放的嵌套事务的一个有趣的变体是事务型工作流。

术语事务型工作流用于强调协作工作流事务型属性的相关性。协作工作流实现公开的(可见的)流程，这些流程已经超越了企业内或跨企业边界的功能型单元。严格地说，只有当工作流没有被定制为执行私有流程时，协作工作流才是公开的工作流。为了简单起见，在本书中术语协作工作流和事务型工作流可以互换。

事务型工作流系统提供了一些脚本工具，可用于表示活动(或业务流程)的组合，并为构建应用特定的开放嵌套事务提供了灵活的方式。事务型工作流管理流程(如订单完成、新产品介绍，以及跨企业供应链管理)，这些流程横跨企业中的多个功能单元，从而可以将活动集成到其他组织的活动中。事务型工作流涉及活动(或流程)的协同执行，这些活动有逻辑决策点，在有可选路径时，可在运行决定一个工作项走哪个分支。通过一组有界的逻辑决策点，可标识和控制每个流内的可选路径。每个决策点中的业务规则决定如何处理、路由、跟踪和控制与工作流相关的数据。如果流包含决策点，在不同的时候，同样的工作流定义会进行不同的初始化，并以不同的方式处理。

控制流依性表示了协作需求。控制流依性规定了一个不变式，用于控制每个活动的执行。活动中的这些不变式基于事务型工作流内其他活动的执行状态，即它们是活动、提交还是中止，其他活动的相应参数，以及外部变量(如时间属性)。

例如，在图 10.11 中，假定管理活动 a_3 的执行的不变式(业务规则)是一个工作流变量(布尔量)——活动 a_2(检查信用值)的响应参数——表明一个特定用户是可信任的。工作流变量通常用于存储运行时工作流所需的应用特定的信息。创建变量并分配大的数值来控制通过工作流实例的逻辑路径。类似地，活动 a_7(调度运输)的不变式要么是活动 a_4(订单确认)或 a_5(调度产品)成功结束。

事务型工作流中的流程必须适应特定环境，如库存断供或新产品的负面反馈。例如在订单提交给客户后，saga 不支持执行流程中止订单。此外，saga 假定总有一个语义等价的补偿事务取消子事务的效果。但并不总是这样的。在某些情况下，补偿事务可能不能确保完成反向恢复。例如，在一个取消的订单输入工作流事务(参见图 10.11)中，通过将商品返回给供应商来发起一个补偿活动(执行后向恢复)。这个活动仍会导致运输和重新库存的费用，因为供应商可能决定收取费用。对于这类事务需求，已有的开放嵌套事务模型如 saga，通常不能提供合适的支持。一个事务工作流可能需要在它的内部结构中变化隔离级。这意味着一个事务型工作流中的一些活动间的隔离需求可能互不相同。

事务型工作流提交不需要所有的活动都发生。有些活动是可选的，而其他活动可能有其他定义。此外，用在不同工作流中的相同活动可能有不同的补偿事务。最后，事务型工作流的中止可能需要复杂业务流程的执行。补偿事务对于业务流程的自动化是非常重要的。对于失败的 Web Service 活动，BPEL 中包含了显式的定义机制，可将失败处理嵌入在流程中，参见第 9 章的“抽象流程和可执行流程”一节。

2. 恢复机制

开放的嵌套事务模型放松了 ACID 事务的传统的隔离需求。开放的嵌套事务的恢复有两种可能的模式：后向恢复和前向恢复。

后向恢复用于事务中止的情况，确保包含的事务返回到执行中止事务之前已有的一致状态。这个操作包括孩子的上下文。为达到这个目标，事务初始化某些补偿活动，消除失败事务的效果。因为开放的嵌套事务可能包括具有不可逆副作用的活动，所以开放的嵌套事务不能在所有情况下都确保完整的后向恢复。正如已经讨论的，出于这个原因，事务型工作流必须定义合适的业务逻辑来执行后向恢复。注意，并不是所有事务型工作流的活动都是事务型的。显然，非事务

型的活动不需要原子性属性及补偿事务。例如，向客户发送报价，计算报价或接收客户的订购单，就无须补偿活动。

在系统故障的情况下，前向恢复确保事务状态可以恢复到一个一致的状态，它的执行可以恢复，并通过故障点继续可靠地执行。前向恢复假定每个子事务的资源管理器都持久地维持它产生的状态和结果。使用前向恢复，可以在考虑事务故障的情况下允许子事务继续执行。当活动的执行不能回滚时(如当订单已发货)使用补偿活动。

补偿是实现恢复和取消已完成活动或事务效果的逻辑处理。补偿活动到底是进行前向恢复还是后向恢复，通常取决于特定应用的特性。

恢复和补偿之间的关系如下[Arkin 2002]：

- 前向恢复在事务完成前发生，以便事务能继续处理，直至结束。
- 后向恢复在事务中止时发生，为了消除事务的后果。
- 补偿在事务结束后发生，它的作用是抵消已结束的事务的作用。
- 在后向恢复期间，父事务通过使用补偿活动补偿执行的任何一个后代子事务。
- 如果补偿逻辑依赖于事务执行的活动，一个(子)事务可以将它的补偿逻辑规定为定义的一部分。当使用特定事务的补偿活动时，调用该逻辑。为了中止事务，在事务结束后补偿的逻辑定义独立于执行后向恢复的逻辑。

将事务的补偿逻辑和调用补偿的活动进行分离，这使得在不同上下文或流中执行的不同活动，可以使用相同的逻辑补偿相同的事务。

10.4 事务型 Web Service

通常将 Web Service 视为构建集成的桥梁，集成可能横跨多家独立的企业及其系统。基于 Web Service 技术，将这些业务元素连接为一个整体。跨企业的服务应用是分布式应用，将完全不同的客户应用的业务功能和业务逻辑恰当地融合在一起，从而提供一系列的自动流程，如采购和订单管理、协同规划、预测、补给、需求和产能规划、产品调度、市场信息管理、装运/集成物流等。这种方法将一些后端技术组件聚集到高层的、面向业务的服务，从而可以更容易地将遗留应用移植到新的解决方案。跨企业的业务流程自动化的一个关键需求是描述业务流程协作的能力，例如基于标准形式的金融资源的提交和交换，并且业务流程实现和监控工具能够识别这些标准形式。为确保一致、可靠的执行，业务协作需要支持事务性。协作业务流程横跨不同类别系统，从短暂的实时事务系统到长期运行的、扩展的协作。在扩展企业中，这为事务的完整性、回弹性以及可量测性提出了很高的要求。

对于复杂业务应用而言，因为不一致和故障很快就会在整个值链中显现，所以后端服务的可靠性、一致性和可恢复性更加重要。因此 Web Service 解决方案必须能够支持前面章节中所描述的高级事务管理解决方案。

传统事务依赖于紧耦合(同步)协议，因此通常不适用于松散耦合的基于 Web Service 的应用，尽管它们可能会采用其中的部分技术。正如我们在本章中已经讨论过的，严格的 ACID 和隔离性尤其不适用于自治的贸易合作伙伴之间的松耦合应用。在业务领域中，为了安全性和库存控制问题而预防本地资源的硬锁定是不切实际的。Web Service 环境需要更加松散的事务形式，不严格遵循 ACID 属性，诸如协作、工作流、实时处理等。在 Web Service 所表示的松耦合的环境中，因为处理器可能发生故障、流程可能会被取消、服务可能会被移动或撤销，所以长时间运行的应用需要支持协调、恢复和补偿。另一个重要的需求是 Web Service 事务必须跨多个事务模型和协议，这些模型和协议位于 Web Service 要映射的底层的基础架构上。

最后，需要将 Web Service 分组到应用中，这些应用需要某种形式的关联，但是并不必须事务性行为。

业务规则与业务流程以及基于 Web Service 的应用相关联。这些业务规则表达了更深层业务语义、条件逻辑、计算、优先级和故障。在事务中，业务规则管理持续时间和参与特性。它们决定了一些可能的结果。在这些结果中，可能包含不重要的部分故障，提供了一些竞争性的服务选择，而不是严格的传统 ACID 事务假定的全做或全不做。

在本节中，我们将讨论传统的、开放的、嵌套事务是如何与 Web Service 的相结合的，并分析 Web Service 事务的一般特性和行为。然后，我们将主要讨论 Web Service 事务的三个标准规范：WS-Coordination、WS-Transaction、Web Service 复杂应用框架。

10.4.1 Web Service 事务的定义和一般特性

Web Service 事务[⊖]是“新一代”的事务管理，它是从核心事务技术，尤其是分布式协同事务、开放嵌套事务、事务性工作流以及各种形式的恢复中发展而来的。Web Service 事务是业务状态的一致变化，这种变化是由定义明确的业务功能驱动的。在 Web Service 事务结束时，参与事务的各方的状态必须一致。也就是说，对于整个 Web Service 事务期间消息交换的结果，各方的理解必须相同。Web Service 事务可表示某个公司某些产品的订单，这是一种最简单形式的事务。订单的完成导致相关业务状态的一致变化：后端订单数据库的更新以及购买订单文档拷贝的归档。更复杂的 Web Service 事务涉及的活动有支付处理、装运和跟踪、市场策略的协调和管理、决定新产品出售、授予/延长贷款、管理市场风险、产品工程等。这些复杂的 Web Service 事务通常由独立的事务性工作流驱动，这些工作流在一些点上必须相互锁定以获得共同期望的结果。这种同步是更广泛的业务协调协议（如本章后面我们将讨论的 WS-Coordination）的一个部分，这个协议定义了进行交互的业务方之间的公共的、一致的交互。

还有一个需要回答的问题是，在 Web Service 技术栈的哪个层次引入 Web Service 事务？在所开发的 Web Service 事务的上层，虽然 SOAP 似乎是一个很自然的候选。然而，作为当前 SOAP 消息最常用的传输机制的 HTTP，并不是实现事务的可行选择。

HTTP 是一个无连接的单向请求/应答协议，因此难以让事务参与方进入事务的双向对话。此外，必须协调的事务参与者不仅仅是应用层的端点，还可能是其他的资源管理器，如 DBMS。当前，数据库事务协调接口不能通过 SOAP 到达，只是因为这类协同协议没有 SOAP 映射。这表明，对于需要联合的两阶段提交事务的 Web Service 而言，基于 SOAP/HTTP 的实现并不是一个可行的解决方案。

对于基于事务性 Web Service 构建大规模的、联合应用，采用 SOAP 解决方案也是不可行的。因为一个 SOAP 请求/应答对的结果可能影响某些其他 SOAP 消息的操作，所以当将 Web Service 链接起来以形成互操作的应用时，有时会有一些特殊的需求。在这些情况下，应用开发者必须标识每个 SOAP 消息，以便能够协调一个请求/应答对和其他消息操作的输出。这需要在不同的 SOAP 请求/应答对间维持一个逻辑协调，这显然不是一个可行的解决方案。

Web Service 事务的一个重要需求是，它们提供维护事务性行为的机制，以及维护在一个较高的抽象层（在 SOAP 消息之上）上的事务协调上下文。其中，该抽象能表达业务流程的真实自然状态以及不完全决定论（这主要用于应付种种变化、递增和部分成功）。图 10.12 描述了 Web Service 事务体系结构的高层视图。

⊖ Web Service 事务亦经常被称为业务事务或业务流程事务。

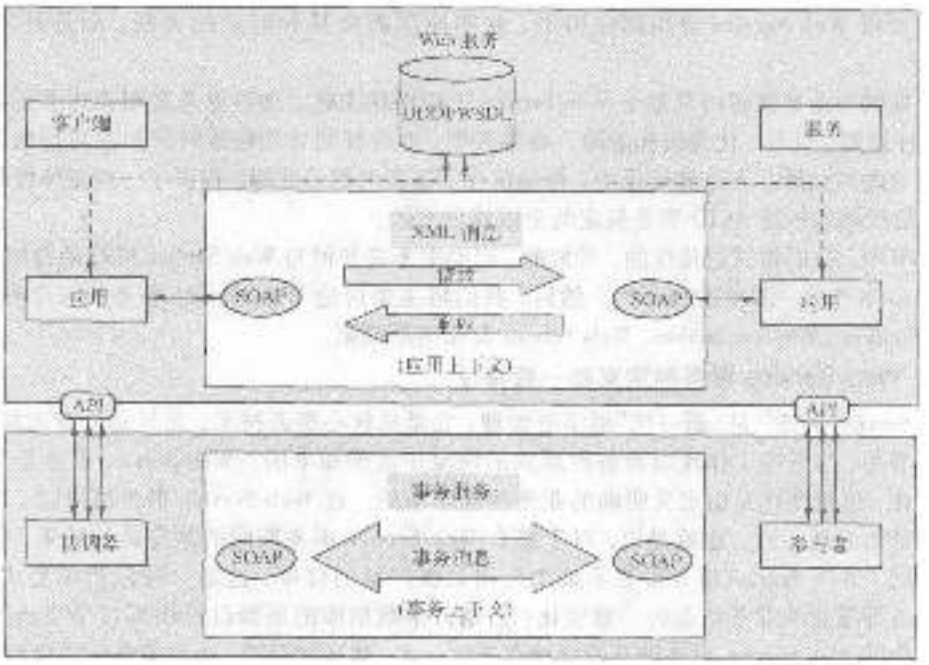


图 10.12 Web Service 事务体系结构

引自：Pearson Hall 出版社 2004 年出版的、由 M. Little, J. Maron, G. Pavlik 著的《Java Transaction Processing》一书(允许复制)

图 10.12 中描述的 Web Service 事务体系结构是基于事务协调器、事务参与者和事务上下文的概念[Little 2004]。正如该图所阐述的，应用客户端与 Web Service 事务交互的方式类似于传统的分布式事务系统。客户和事务协调器如何互操作的具体细节依赖于所使用的事务性协议，如 WS-Transaction 或 WS-TXM。在该体系结构中，通常在客户和服务之间传播上下文信息，从而在互操作的分布式系统间提供一个应用上下文信息流。如图中所示，可以使用 SOAP 头部信息。

无论何时需要事务性服务，Web Service 事务基础架构(图中的底部)负责将上下文传播给请求的服务。当一个具有事务性属性的服务接收也行有事务上下文的应用的调用时，它将参与者注册到上下文中引用的事务中。事务性服务(如库存控制系统)最终必须(通过它的容器)处理不同客户的并发访问，并维保事务一致性和关键数据的完整性，如定购的产品数量。

一旦参与者注册之后，它的工作就由图 10.12 中的事务协调器控制。Web Service 事务参与者也类似于传统分布式数据库系统中的参与者。事务协调器和参与者之间的互操作遵循到用服务所使用的事务协议的具体细节。

10.4.2 Web Service 事务的操作特性

在一个 Web Service 环境中，事务是复杂的，设计多个参与方，跨越许多组织，并且持续很长时间。此外，对事务的提交还需要参与组织进行协商。更确一点说，试图仿效更深层业务讲义的 Web Service 事务是自动化的长期运行的任务，可能涉及协商、提交、合同、履运和启动、跟踪、各种付款方式以及例外处理。一个 Web Service 事务在特别的情况下，可能涉及许多企业。这些企业在事务中所关注的各不相同，如产品或服务、支付安排或监控和控制。Web Service 事务保留一致性要求，即作为一个单元之行到结束(成功)或失败。

每个 Web Service 事务完成多个业务流程，如在企业后端系统上的信用检查，自动结账，购买

订单、存货更新以及货运。一个挑战是如何将操作型系统及企业数据和 Web 应用集成起来，允许客户、合作伙伴，以及供应商与企业合作系统如存货、审计、购买直接交互。这些业务相关的任务的性能需要将事务性属性融合到 Web Service 样例中。试图将业务级决定与事务性基础结构集合起来的 Web Service 表现出以下特性：

- 它们通产表示一个关键业务功能，如供应链管理。
- 它们可以涉及多个合作伙伴(组织)以及由多个合作伙伴独立操作的多个资源，如业务应用、数据库以及 ERP 系统。
- 它们定义通信协议绑定，定位 Web Service 域，同时还保留在其他通信协议上携带 Web Service 事务消息的能力。协议消息结构和内容约束在 XML 中系统化，消息内容在 XML 实例中编码。
- 它们应当基于正式的贸易伙伴协议，以业务协议的方式表达，如 RosettaNet、PIP 或 ebXML 协作协议约定。

当一个业务功能通过 Web Service 作为更大的业务流程一部分调用时，与该业务流程相关的所有事务行为依赖于 Web Service 的事务能力。应用开发者不是组成更复杂的端到端行为，而是选择那些更合适的元素，将事务性和非事务性 Web Service 碎片组合为一个有凝聚力的服务。

10.4.3　Web Service 事务的类型

Web Service 事务围绕活动进行组织的。活动是一种通用的计算，该计算将作为一组 Web Service 上的限定作用域的操作集执行，并且这些 Web Service 对于所要产生的结果需要有一致的约定。在事务的工作单元中，协作 Web Service 被称为参与者(participant)。参与者是共享一个共同事务上下文的 Web Service。正如 10.2.1 节所提到的，事务上下文是一个数据结构，包含参与者共享的相关信息，如共享资源的标识、结果集、公共安全信息，以及一些指针，这些指针指向了业务流程的最后的稳定状态。Web Service 事务包含两种不同类型的事务活动：原子操作(或短事务)以及长事务。

1. 原子操作

原子操作是由一些服务组成的小规模的交互。对于整个事务的共同输出(提交或中止)，这些服务将进行协商并强制遵循。在所有参与者都完成后，原子操作确保所有的参与者看到相同的、一致的结果，否则回退到原来的状态。在成功的情况下，所有服务都将使得它们的操作结果持久化(即提交)。在失败的情况下，所有的服务撤销(补偿或回滚)在事务期间调用的操作。原子操作不一定必须遵守 ACID 属性。事实上，它可能会根据应用的协调协议放松隔离级和持久性，如易失的 2PC 或持久的 2PC。原子操作可以是嵌套的(封闭的嵌套模型)。对于作为事务一部分执行的一组原子操作，原子操作保证“要么全做，要么全不做”。总之，一个原子操作可看作为由 Web Service 事务的一个参与者执行的独立的、协同的、短事务工作单元。

为了更好地理解原子操作的特性，我们再次回到常用的例子。假定一个客户端应用(图 10.13 中的应用发起者)决定从一个特定服务(如订单确认或库存核查)中调用一个或多个操作。对于客户端应用来说，这些操作极有可能作为一个单元成功或失败。因此我们可以把每个 Web Service 中客户使用的操作集看作工作的原子单位(即原子操作)。一个原子事务是最简单的工作单位，其行为类似已有的、遵循 X/Open XA 的两阶段提交事务。一个原子事务必须全部提交或者全部回滚。在原子操作内，单个事务性 Web Service 以及该服务的内部流程暴露了一些操作，如支持流程。这些操作通常组成了单个原子操作。当业务模型表示了一个核心业务流程(如企业订单输入)时，通常就会出现原子操作。非原子性 Web Service 活动通常支持服务原子，并且经常出现在辅助流程中，如旅行报销单。

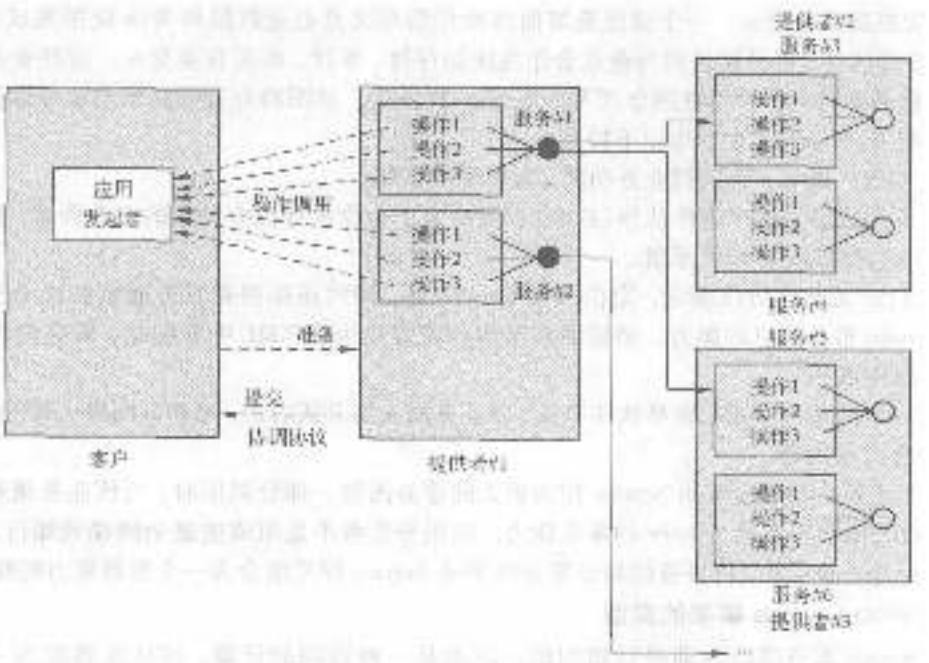


图 10.13 Web Service 事务和应用

原子操作使用 2PC(可支持中止)，因此需要一个协调流程来管理协调协议消息，如准备、提交、取消，这些消息发送给特定的原子事务中的参与服务(参见图 10.13)。这个协调流程可以在应用内部实现，或者更有可能的情况是一个专用的 Web Service[Webber 2004]。在原子操作中，一旦涉及已使用的 Web Service 的实际工作已经结束，客户应用(服务客户端)则可以开始那些 Web Service 的两阶段提交协调。客户(事务发起者)控制 2PC 的所有方面，即准备阶段和确认阶段。客户可以管理和确认何时准备和确认原子事务上的调用，从而使得事务在协议内实现最大的灵活性。客户可以决定事务执行的时序，这隐含地允许执行预定类型的业务流程。例如，这种方式可以应用到订单预定系统中，在 2PC 的准备阶段预定一些产品，在确定阶段实际购买预定的产品。

2. 长事务

业务流程是长时间运行的，例如从订购到收到货款可能需要数月，并且其间可能会出现很多新的情况，因此需要响应客户的各种要求，并针对市场情况进行相应的变化。相应地，业务流程通常比较庞大和复杂，涉及物流、信息流和业务承诺。长持续时间的(业务)活动是一些原子操作的聚合，这些事务可以具有嵌套事务和事务性工作流的特性和行为。长持续时间的活动将一些原子操作与一些常规的事务逻辑功能聚合成一个内聚的 Web Service 事务。在长持续时间的事务中，Web Service 事务的发起者(通常为客户端应用)能够操纵事务中嵌入的原子操作。长持续时间的事务参与者可选择性地确认(提交)，也可选择性地取消(回退)，即使这些参与者能够进行提交。因此从这个意义上说，长持续时间的事务是非原子化的。原子活动可以是长持续时间活动的组成部分。在长时间运行的业务活动完成之前，需要先提交活动中嵌入的短事务，并要能得到其成功完成的结果。当长时间运行的业务活动失败时，需要对短(原子)事务进行补偿。一个特定的长持续时间的业务活动中的原子操作并不需要一定要有一个共同的结果。在应用控制(业务逻辑)中，有些原子操作可能会成功完成(确认)，而其他的一些原子操作可能会失败或出

规异常，诸如超时或故障。

图 10.14 显示了长持续时间活动和原子活动的事务嵌套。因为可以嵌套长持续时间活动，所以不仅在长持续时间活动间有父子关系（在图 10.14 中，活动 A_2 嵌套在活动 A_1 中），而且在长持续时间活动和原子操作间也有父子关系（在图 10.14 中，原子操作 a_3、a_4 和 a_6 嵌套在活动 A_2 中，原子操作 a_1、a_2、a_5 和 a_7 嵌套在活动 A_1 中）。客户端应用能够命令长持续时间活动中的原子活动是成功还是失败（即使服务能够成功完成）。

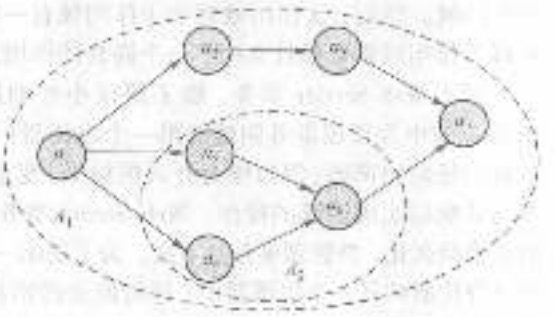


图 10.14　原子活动、长持续时间活动和嵌套

现举例说明长持续时间的活动。对于单处理场景进行轻微的改变，假设制造商请一个供应商提供一个贵重而又易碎的装备。对于该产品的订购，安排一个原子操作。第二个安排是对产品投保。第三个安排是运输。假如客户端应用并不规避风险（由于花费太大），则即使保险操作（原子操作）被取消，客户可能仍然会确认该 Web Service 事务，并在没有投保的情况下装运该产品。然而，另一种更可能的情况是，客户会试图重新给该产品投保。在这种情况下，Web Service 基础架构将发挥作用。客户可通过基础架构（例如，UDDI 服务基础架构）选择另一家保险公司。一旦客户发现了一家新的保险公司，则基于特定的协调协议，可以成功确认所有所需的原子操作，从而客户可以重新试图完成该持续长时间的业务活动。

可以将一个持续长时间的活动划分成多个原子操作。由于各个原子操作并不需要同时锁定资源，因此这种划分带来了灵活性。此外，假如在原子操作层次上发生失败，可以使用替代路径将持续长时间的整个活动继续向下。替代路径既可以为补偿已做的任务，也可以选择做其余的任务（例如，选择不同的保险公司，或者假如首选的保险公司的费用太昂贵，可使用一种更安全的运输方式）。

Web Service 的主要问题之一是：它们的底层实现通常位于远地，并由第三方（图 10.13 中的服务提供者）进行管理。因此在它们的应用中，存在一个不断增加的失败风险。针对这一威胁，在持续长时间的活动的生命周期中，可以决定选择性地取消一些原子事务，并增加一些新的原子事务。因此，客户端应用的业务逻辑部分可动态地建立持续长时间活动的成员关系。例如，（客户端应用调用的）事务组合服务可以采用临时"投票"来发现原子活动是否已经结束、或者已经取消，或者由于超时而决定撤出它的参与者。

10.4.4　评议小组与介入

在 Web Service 事务中，参与者对于具体任务的结果取得共识是非常重要的，当事务涉及金钱交易时，尤其如此。为了取得共识，可将参与者组织为一些"评议小组（consensus group）"，以便参与者可以看到同样的结果（例如，意见一致）。评议小组的另一个术语是"事务作用域（transaction scope）"，作用域是一个由通用计算组成的业务任务，其中通用计算作为 Web Service 集合上的一些操作执行，并且 Web Service 间需要对结果进行相互协商。Web Service 事务的不同的参与者能够是不同的评议小组的一部分，以至于一个小组中的参与者比其他组中的参与者能够观察到不同的结果。每一个评议小组能够有一个特定的原子性级别。这些特性正是 Web Service 事务与传统的事务的区别所在。Web Service 事务与传统的事务的另一个区别是：在事务终止之前，评议小组的参与者也可以离开评议小组。

协调是一个与评议小组相关的概念。评议小组包含任务或低级别的活动（孩子），完成高级

后的活动(父亲)需要低级别的活动。评议小组可以视为是一个子作用域。子作用域能够独立于父作用域。然而，父作用域对于子作用域有一些基本的控制。例如，当父作用域需要成终完成并告诉子作用域需要做什么时，一个嵌套的作用域(子作用域)可以认为它已经终止。

对于 Web Service 事务，除了评议小组和议会，另一个重要概念是下级协调器的介入。在 10.2.1 节中为顶层事务创建的第一个协调器(根协调器)负责驱动 2PC。对于已有的事务，随后创建的任何协调器(例如作为介入的结果)变成流程中的下级协调器。根协调器发起 2PC，并且参与者响应实现协议的操作。Web Service 事务中的协调器负责将结果通知给参与者，使得参与者的结果持久化，并管理事务上下文。为了表示一组其他的参与者(通常是本地参与者)，参与者可以将自身注册到另一个协调器中，这时原先的协调器就变成参与者。当协调器表示一组本地参与者时，这通常称为介入(interposition)。在导入域(服务器)，介入技术使得代理能够处理协调器的功能。介入技术通常用于提高性能(当参与者大量增加并超过一定的阈值时)和 Web Service 事务的安全性。介入协调器充当了一个下级协调器。在事务中，介入协调器作为参与者注册。

在事务中，协调器间的关系形成了一棵树，如图 10.15 所示。根协调器负责完成 Web Service 事务(顶层事务)。协调器并不干预参与者的实现。参与者为了提交一个事务可以与数据库进行交互(例如，图 10.15 中企业 1 和企业 5 的参与者)。参与者也可以充当协调器本身(如图中企业 4 中的参与者)，负责向许多后端系统转发(根)协调器的消息。在图 10.15 的例子中，当代理协调器收到根协调器的调用，并整理对根协调器的响应时，代理协调器(企业 4 中的参与者)负责与两个参与者(企业 2 和企业 3)交互。就根协调器而言，介入(子)协调器是一个遵循父协调器事务协议(通常为两阶段协议)的参与者。然后，必须根据子协调器操作所在的域以及域所使用的协议，对子协调器进行调整。

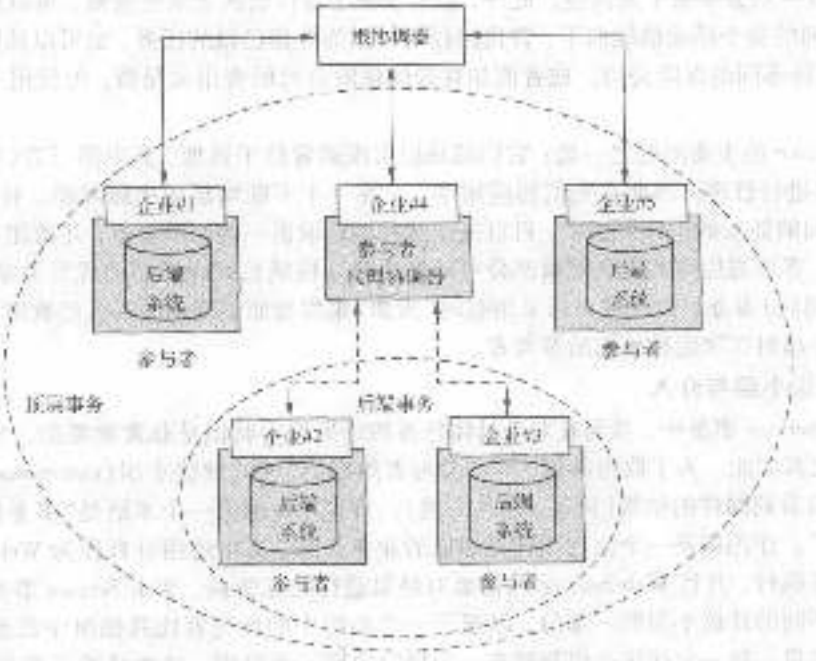


图 10.15 事务介入

介入参与者(例如，一个下级协调器)从自身的参与者向紧邻的更高层(上级)协调器发送消息，并从上层协调器接收消息，以及将这些消息转发给它的参与者。介入参与者不能决定将自

身。介入参与者并不管理事务，而仅是协调事务，例如跨流程或计算机。在介入中，子协调器的参与者不能自治地完成，而必须由它们的协调器命令。假如一个参与者退出，下级协调器既不能对此做任何事情，也不能基于这一事实进行任何其他的决策。

在介入和嵌套之间还存在一个重要的差别。对于嵌套，因为父子关系，应用可以适当地管理作用域。假如父亲完成了工作，并且看到孩子也已完成了工作，则父亲可以根据需要运行不同的反应，诸如补偿、接受后代的工作、终止孩子或者调用其他的服务（例如，假如孩子退出）。这意味着，父亲有着完全的控制。

因为介入协调器可以将结果中立的协议转换为特定平台的协议，所以下级协调器的介入可以帮助实现互操作性。如图 10.15 所示，代理协调器将企业 2 和企业 3 中的内部业务流程与根协调器之间进行了隔离。在介入域中，当与服务进行通信时，不仅介入域（企业 2 和企业 3）需要使用不同的上下文，而且每个域都可以使用不同的协议与域之外的服务进行通信。下级协调器可以充当转换器，将域外面的协议转换为在域中所使用的协议。添加基于事务的协议的好处是：参与者和协调器基于结果（诸如回退、补偿和三阶段提交等）协商一组商定的操作或行为。

10.4.5 Web Service 事务的状态

每一个 Web Service 事务实例的状态转换的方式类似于常规事务（参见图 10.2）的状态转换。除了图 10.15，下面将继续描述这些内容[Arkin 2002]：

- 活跃：事务是活跃的，并执行在事务上下文中所指定的活动。
- 准备完成：事务已经执行了活动集中的全部的活动，现在准备结束。这可能涉及一些额外的工作，诸如变得数据变化持久化、完成两阶段提交、协调嵌套的 Web Service 事务完成（例如，上下文完成）。
- 已完成：事务已经执行了成功地结束事务所需的所有工作。
- 准备中止：事务未能成功地完成，并且目前正准备中止。这可能涉及一些额外的工作，诸如：运行补偿事务，将结果传送给事务参与者（仅对于原子事务），协调嵌套的 Web Service 事务中止。
- 已中止：事务未能成功地完成，并且已经执行了中止所需的所有工作。
- 准备补偿：事务正在执行它的补偿活动集中的活动。
- 已补偿：事务已经成功地执行了它的补偿活动集中的所有工作。

假如已经成功地执行了所有工作，事务通常开始"准备完成"（参见图 10.16）。假如事务出现异常（例如，超时、出现故障，以及它的父亲出现异常或者事务不能成功完成），则事务的状态将要转换为"准备中止"，然后为"已中止"。然后，事务将执行所有所需的工作，尤其是与异常以及完成相关的活动。

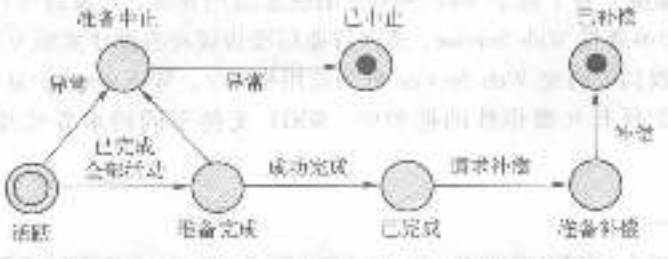


图 10.16 Web Service 事务实例状态的转换图

一旦事务为“完成”状态，很可能需要对事务进行补偿(参见图 10.16)。补偿 Web Service 事务的最初的操作是将事务转换为“补偿”状态。在变为“已补偿”状态之前，事务将要执行它的补偿活动集中的所有活动。注意，在图 10.16 中，终止状态仅是“已中止”状态和“已补偿”状态。

10.4.6 Web Service 事务框架

Web Service 是大型业务流程的一部分。当通过 Web Service 调用业务流程时，与业务流程相关的整个事务行为取决于 Web Service 的事务能力。此外，业务流程可以涉及一个或多个业务功能(通过一个或多个 Web Service 调用)。为了支持所需的事务行为，可能需要跨部分或所有参与者的协调。协调必须考虑到松耦合和 Web Service 的自治，并且协调必须遵循业务协调协议。若没有适应异常情况的协调协议，业务将无法很好地进行错误恢复。错误恢复需要内部的业务流程持久地记录各种状态。Web Service 事务框架(WSTF)能够将松耦合的 Web Service 编配到一个 Web Service 事务中。

WSTF 建立于 Web Service 编配/编排基础架构上。在 Web Service 编配/编排中，Web Service 操作是动态服务组合的一部分。通过定义不同的活动，如调用这些 Web Service、操纵和协调数据流、使用异常和错误处理机制，Web Service 编配/编排基础架构提供了创建复合流程的能力。我们在第九章所描述的 BPEL 标准和 WS-CDL 标准提供了这些高级功能。WSTF 将提供下列三个相互关联的组件，这三个组件将能增强事务功能。

1. Web Service 事务类型

Web Service 事务类型用于表示持续长时间的工作流事务、常规的短事务、异常处理机制以及补偿活动。在协调分布式自治业务功能方面，WSTF 将提供事务功能。对于参与共享的业务流程的交易合作伙伴，WSTF 将保证已协调的、可预测的结果。

2. 协调框架[⊖]

WSTF 将包括一个协调框架。对于协调事务和跨分布式应用的 Web Service 的非事务性操作，协调框架利用了 Web Service。该框架将使得应用可以创建上下文。将活动传播到其他的服务以及注册不同的协调协议都需要使用这些上下文。对于参与 Web Service 事务的组织中已有的工作流和事务处理系统，WSTF 通过隐藏它们的专有协议和异构性可以协调它们的活动，并使得它们可以互操作。

3. 业务协议

在较高的(语义)层，WSTF 能够获取交易合作伙伴之间的信息和交换需求，识别每一个业务协作和信息交换的时序、顺序和目的。在业务协议中，这个功能具有代表性。基于实际的业务上下文，可以向特定的合作伙伴发送消息，也能从特定的合作伙伴接收消息，业务协议定义了这些操作的顺序。业务协议可以获取具有跨企业业务意义的所有行为。假如可以使用一个标准的协议，所有的参与者都将可以了解和遵循这一协议，而无须人参与协商，从而减少了建立跨企业的自动化业务流程的难度。对于基于 Web Service 的快速应用开发，开发者可以使用这类现成的解决方案。此外，对于事务性 Web Service，遵守行业标准协议将有助于实现互操作。

在 10.6 节中，我们将讨论 Web Service 复杂应用框架[⊖]。Web Service 复杂应用框架是 WSTF 的一个实例。在一个具有互操作性的框架中，WSTF 支持不同的事务模型、协调协议和体系结构。

⊖ 注意，我们将在 10.5.1 节中讨论 WS-Coordination 框架。WS-Coordination 框架将事务类型命名为协调类型。

⊖ Web Service 复杂应用框架有时也称为 Web Service 复合应用程序框架——译者注

在高级业务应用中，将需要使用事务性 Web Service 标准。到目前为止，读者已经了解了有关事务性 Web Service 标准的足够多的信息。下面，我们首先将重点讨论 WS-Coordination 规范和 WS-Transaction 规范所提供的协调协议样例和事务模型。然后，我们将分析 Web Service 复杂应用框架。

10.5　WS-Coordination 和 WS-Transaction

BEA、IBM 和微软共同提出了 WS-Coordination 规范和 WS-Transaction 规范[Cabrera 2005a]、[Cabrera 2005b]、[Cabrera 2005c]。这些规范的目的是定义事务性 Web Service 的机制。这些事务规范包括 Web Service 事务类型和协调协议，并支持不同的事务应用需求。

WS-Coordination 规范和 WS-Transaction 规范对 BPEL 规范是有益的补充，它们提供了定义具体的标准协议的机制。事务处理系统、工作流系统或者希望协调多个 Web Service 的应用都将需要使用这些所定义的机制。WS-Coordination 规范提供了一个框架，可通过上下文共享来协调分布式应用中的操作。WS-Coordination 规范和 WS-Transaction 规范协同工作。WS-Transaction 规范是使一个仅针对结果确定和处理的协议。对于原子事务以及长事务和相关的协调协议，WS-Transaction 规范提供了一些标准。

10.5.1　WS-Coordination

一个业务流程可能涉及许多 Web Service。这些 Web Service 协同工作，从而提供了一个共同的解决方案。为了成功地完成流程，每一个活动都需要将它自己的活动与其他服务的活动进行协调。由于业务时延、网络时延以及等待用户与应用交互等，这类应用的完成通常需要持续较长的时间。所谓协调指的是：组织大量的独立实体来实现一个具体的目标。当 Web Service 事务涉及多个事务性资源时，协调可确保事务行为的正确性。需要协调各个资源中的事务管理器，使得它们能够正确地进行提交或回退。协调通常是某个实体(称为协调器)因为某些原因而向许多参与者传播信息。这些原因通常与具体领域相关，例如对于分布式事务协议的决策取得一致，又如，在可靠的多播环境中保证所有的参与者都获得一个特定的消息[Webber 2003a]。当对各方进行协调时，需要传播称为协调上下文的信息，以便逻辑上都同属于同一个活动的操作能够相互协作。

基于 WS-Coordination 规范[Cabera 2005a]，可将大量的独立应用组织成一个协调的活动。协调分布式应用的操作涉及许多协议。WS-Coordination 规范描述了一个支持这些协议的可扩展的框架。因为与 Web Service 相关的许多问题(诸如安全性、事务管理、复制、工作流、授权等)都需要协调，所以在 Web Service 环境中需要一个协调基础架构。这是支撑 WS-Coordination 的基础。一个流程可能横跨多个可互操作的 Web Service。WS-Coordination 协调流程中的操作，并使得参与者对于分布式活动中的结果达成一致意见。为了支持许多应用，诸如那些需要对分布式事务的结果达成一致意见的应用，将需要使用协调(事务)类型和协议。这组协调类型是可扩充的。只要参与协作的每一个服务对于所需的服务都有共同的理解，一个具体的实现可以定义新的类型。

WS-Coordination 规范在两个方面提供了可扩充性。它允许发布新的协调协议，并允许从协调类型和扩展元素定义中选择一个具体的协议，其中扩展元素的定义可以添加到协议和消息流中。

WS-Coordination 规范描述了一个协调服务框架。协调服务(协调器)聚合了三个组件服务：

- 激活服务具有一个操作，该操作使得应用能够创建一个协调实例和相关的上下文。
- 注册服务也具有一个操作，该操作使得应用可以注册协调协议。协调协议可以协调 Web

Service 环境中的分布式操作的执行。

- 协调服务实现了协调类型和具体的协调协议集，执行协调活动将需要使用协调协议集。协议服务通常可以支持两种协调类型原子事务和业务活动。这些原子事务和业务活动是 WS-Transaction 的一部分，我们将在后面的章节（参见 10.5.2 节）中讨论。每一个类型定义了它们自身的一组协调协议。协调协议表示了一组协调规则，在具体的协调类型（例如，对于原子活动的持久的 2PC 或易失的 2PC）的控制下，这些协调规则将作用在活动上。

针对活动协调的另一个重要元素是驱动协调协议完成的实体。该实体通常是一个客户端（应用）。该客户端从总体上控制应用，并且该客户端并不是在协调框架中定义的。这三个 WS-Coordination 组件服务和它们的相互关系如图 10.17 所示。

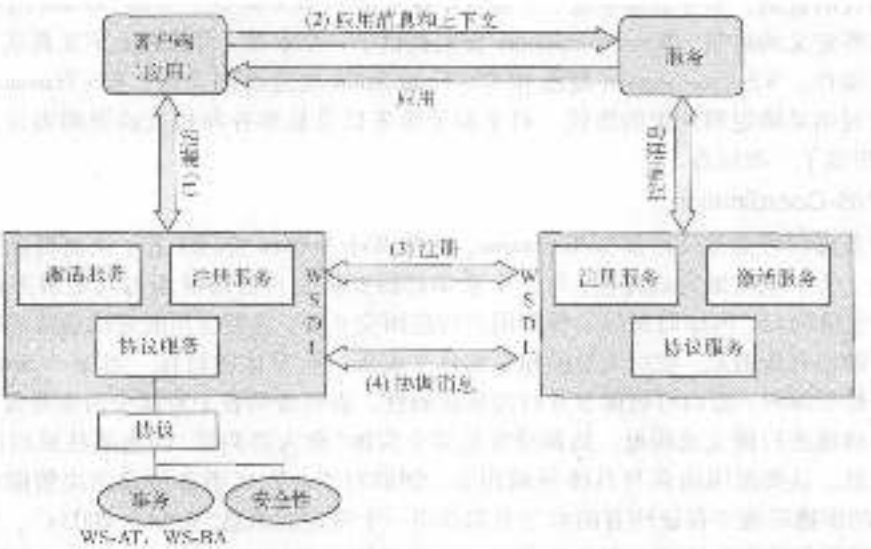


图 10.17 WS-Coordination 体系结构概览

为了发起一个工作，应用首先联系目标激活服务来创建一个活动，协调上下文将标识该活动。协调上下文是一个 SOAP 头块，活动标识符唯一地标识了需要参与的工作。当在一个端点接收到请求时，假如该请求是创建一个新的活动，则创建一个新的协调器，否则将该请求传送到负责该活动的协调器。然后，该协调器将处理被请求的操作。协调器负责保持和各事务以及事务所涉及的 Web Service 的联系。例如，假如订购单服务通过调用几个 Web Service 向一些供应商订购产品，然后协调器负责与所调用的供应商保持联系，并记录事务的所有状态。协调器通常暴露与事务相关的操作。

协调服务维护所有活跃的协调器的中央存储库。当新的请求或达任何端点时，可使用活动标识符查找协调器。协调器将请求分发给特定的活动，并协调该活动。为了便于流程的执行，协调服务在它的所有端点使用活动标识符作为一个引用特性（参见 7.2.1 节中的有关 WS-Addressing 部分）。

参与同一个活动的 Web Service 接收应用消息以及附加于消息上的协调上下文。Web Service 然后使用上下文标识注册服务，并将参与者注册到原来的活动中。通过注册可以将所需的不同的 Web Service 系留在一起，从而完成一个联合的工作单元。因此在协调框架中，注册是一个关键操作。

注册服务负责将新的参与者注册到协调器(例如，Web Service 的注册)，并负责选择协调协议。这使得参与者能够在应用的生命周期中接收上下文和协调器的协议消息。通过使用所选择的协议，可以驱动注册的 Web Service，这是协调服务所提供的最后一个功能。协议定义了完成一个活动所需的行为和类作。当创建协调器时，实例化应用将选择一个所支持的协议类型。然后，当协调器上下文传递到其他应用时，其他应用将选择协调类型所支持的协调协议，以便能够参与事务。

WS-Coordination 规范定义了激活和注册端点。协调类型具体规范(诸如 WS-AtomicTransaction 和 WS-BusinessActivity)定义了其他的端点。其他的所有端点的作用都是便于参与者和协调服务能够基于具体的协调协议进行通信。因此，这些端点称作协议服务。

客户端应用通常担任了事务终结者的角色。为了驱动协议直至完成整个应用，客户端应用将要在合适的点请求协调器执行和注册的参与者之间的特定的协调功能。当活动一完成时，活动结果将会被通知给活动。活动结果有可能是各种各样的，既可能是简单的成功/失败通知，也可能是复杂的结构化数据，诸如关于活动状态的详细信息。

图 10.18 显示了协调服务和它的组件，以及 WS-Coordination 和 WS-Transaction 之间的关系。该图表示了 WS-AtomicTransaction 使用的协调类型(定义了诸如"准备"、"提交"、"回退"等操作)，并表示了持久的 2PC 使用的协调协议(定义了诸如"已准备"、"中止"、"提交"、"只读"等操作)。此外该图显示了：在任何特定的时间点，驻留在协调服务上的活跃协调器的数量等于服务所协调的活动的数量。之所以会如此是由于：当创建新的活动或终止已有的活动时，将相应地创建协调器和销毁协调器。

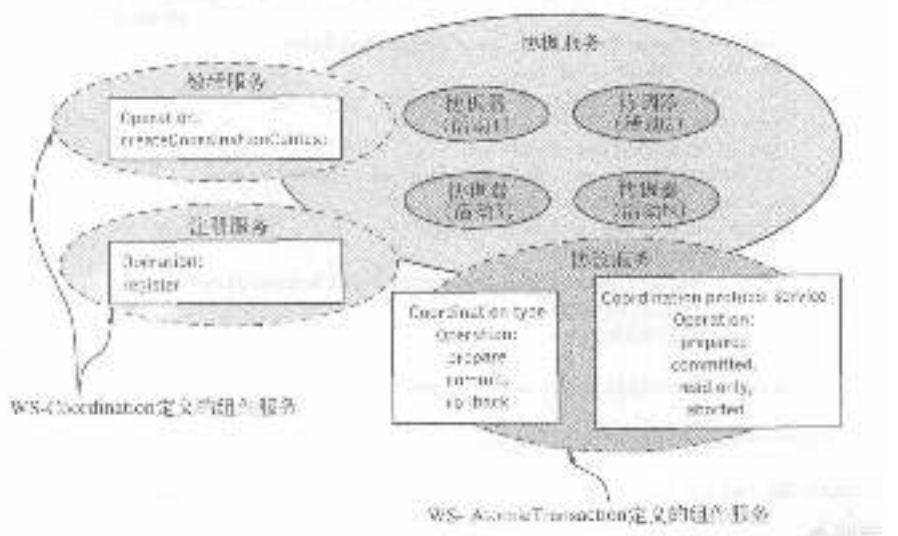


图 10.18　协调服务和它的组件服务

1. 协调上下文

CoordinationContext 是一个完全不同的上下文类型。通过定义这个上下文类型，可将协调信息传送给 Web Service 事务中的参与者。在 WS-Coordination 中，由于上下文信息包含了服务参与协调协议所需的信息，因此协调信息对于协议非常重要。协调上下文提供了在交互的 Web Service 之间共享处理信息的机制，以及提供了将应用中所有组成部分 Web Service 组合成一个协调应

用的机制。为了将活动传播到另一个 Web Service，发起的客户端必须发送一个协调上下文。协调上下文将作为应用消息的一部分发送，并且使用 SOAP 消息中的头部通常可以交换协调上下文。

对于每一个新创建的事务，激活服务返回一个协调上下文。该协调上下文包含一个全局标识符、一个满期时间域、注册服务的地址(WS-Address)和一个协调协议。在 WS-Coordination 中，上下文包含了一个 URI 形式的、具有唯一性的全局上下文。该标识符表示了所调用的 Web Service 针对哪一个事务。为了能够注册新的参与者，上下文包含了协调器(注册服务)的位置或端点地址。在协调器中，接收上下文的各方可将参与者注册到协议中。上下文也包含一个时间邮戳域，该域表示了上下文的有效期。最后，上下文包含了一个协调器所支持的、实际使用的协调协议的信息，诸如 WS-Transaction 协议，该协议描述了具体的完成处理行为。

清单 10.1 是一个支持原子事务服务的 CoordinationContext 样例。

清单 10.1 协调上下文的样例

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/
  soap-envelope"
    <soap:Header>
        . . .
        <wscoor:CoordinationContext
            xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/
                                                    utility"
            xmlns:wsa="http://schemas.xmlsoap.org/ws/
                                          2004/08/addressing"
            xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/
                                                    wscoor">
            <wscoor:Expires> 2012 </wscoor:Expires>
            <wscoor:Identifier>
                 http://supply.com/trans345
            </wscoor:Identifier>
            <wscoor:CoordinationType>
               http://schemas.xmlsoap.org/ws/2004/10/wsat
            </wscoor:CoordinationType>
            <wscoor:RegistrationService>
                <wsa:Address>
                  http://example.com/mycoordinationservice/
                  registration
                </wsa:Address>
                 . . .
            </wscoor:RegistrationService>
            . . .
        </wscoor:CoordinationContext>
        . . .
    </soap:Header>
```

2. 激活服务

当一个应用想要开始一个新的应用时，它请求协调器创建一个新的上下文。通过调用协调器激活服务的 CreateCoordinationContext 操作，以及传送协调类型、它自身的端口类型引用(以便协调器可送回响应)和一些其他的信息，即可创建一个新的上下文。协调器依次调用请求者的 CreateCoordinationContextResponse 操作，传回请求者端点引用、为了响应操作 CreateCoordinationContext 而创建的上下文以及其他信息。

清单 10.2 是一个简单的 CreateCoordinationContext 请求消息样例。这个操作仅有一个真正的参数，该参数定义了所使用的事务的类型。

清单 10.2　激活服务的定义

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-
  envelope"
    <soap:Header>
        . . .
    </soap:Header>
      <soap:Body>
        <wscoor:CreateCoordinationContext
            xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/
                                                    addressing"
            xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/
                                                      wscoor">
           <wscoor:Expires> 2012 </wscoor:Expires>
           <wscoor:CoordinationType>
               http://schemas.xmlsoap.org/ws/2004/10/wsat
           </wscoor:CoordinationType>
        </wscoor:CoordinationContext>
        . . .
    </soap:Body>
</soap:envelope>
```

清单 10.3 是一个对上面的请求消息的响应样例。发出请求的应用(在本例中是订购单服务)已经请求协调器创建一个新的协调上下文，然后将该协调上下文返回给请求者(订购单服务)。作为新创建的协调上下文的一部分，也返回协调器的注册服务端点引用。

清单 10.3　响应消息

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-
 envelope"
    <soap:Header>
        . . .
    </soap:Header>
      <soap:Body>
        <wscoor:CreateCoordinationContextResponse>
         <wscoor:CoordinationContext>
              <wscoor:Identifier>
                 http://supply.com/trans345
              </wscoor:Identifier>
              <wscoor:Expires> 2012 </wscoor:Expires>
              <wscoor:CoordinationType>
               http://schemas.xmlsoap.org/ws/2004/10/wsat
              </wscoor:CoordinationType>
              <wscoor:RegistrationService>
                <wsa:Address>
                  http://coordinator.com/registration
                </wsa:Address>
                 . . .
              </wscoor:RegistrationService>
            . . .
          </wscoor:CoordinationContext>
       </wscoor:CreateCoordinationContextResponse>
     </soap:Body>
 </soap:envelope>
```

清单 10.3 中的影响包含了一个上下文，该上下文包含了一个具有唯一性的标识符、有效期和传送给请求的事务。元素 RegistrationService 包含了一个指向协调器的注册服务的引用指针。这是 WS-Addressing 规范定义的端点引用。事务中的每一个参与者将要调用注册服务，通知事务作用域中的协调器。例如，一个订购单服务可以联系几个供应商服务，那些供应商服务将使用上

面的地址注册到协调器。用这种方式，假如那些供应商服务需要代表事务采取一些操作(如准备、提交或回退等)，协调器可以知道需要联系哪些供应商。

3. 注册服务

一旦已经实例化协调器并且激活服务已经创建了对应的上下文，然后就可创建并暴露注册服务。注册服务允许参与者注册，接收与特定协调器相关的协议消息。与激活服务一样，注册服务在请求的协调器端和响应的请求者端需要端口类型引用。这使得协调器和请求者可以交换消息。

假如订购单服务希望调用它的一个供应商的一个操作，诸如 checkInventory()，然后整个协调上下文作为 SOAP 头部进行传送，从而向协调器表示了该供应商服务是为哪一个事务注册的。当供应商服务接收了该消息，然后它将分析上下文中的信息，从而决定供应商服务是否能够在指定的协调类型下参与事务。

当一个请求者(例如供应商服务)将它自身注册到协调器，作为对于创建上下文的响应，协调器将返回一个注册服务引用，以供请求者使用。请求者然后调用协调器注册服务中的 Register 操作，传递协调器注册服务端点引用、它自身的端点引用(PartcipantProtocolService)，从而使得协调器能够调用参与者中的事务操作，并送回事务状态信息、所选择的注册协调协议(ProtocolIdentifier)和其他信息。在本例中，所选择的协调协议是 Atomic，并且跟随着两阶段提交协议(wsat#Durable2PC)。所有的这些如清单 10.4 所示。

清单 10.4 定义注册服务

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope>
 <soap:Header>
  <wscoor:CoordinationContext soap:mustUnderstand="1">
          <wscoor:Identifier>
             http://supply.com/trans345
          </wscoor:Identifier>
          <wscoor:Expires> 2012 </wscoor:Expires>
          <wscoor:CoordinationType>
            http://schemas.xmlsoap.org/ws/2004/10/wsat
          </wscoor:CoordinationType>
          <wscoor:RegistrationService>
            <wsa:Address>
               http://coordinator.com/registration
            </wsa:Address>
                . . .
          </wscoor:RegistrationService>
            . . .
   </wscoor:CoordinationContext>
 </soap:Header>
 <soap:body>
  <wscoor:Register>
   <wscoor:ProtocolIdentifier>
      http://schemas.xmlsoap.org/ws/2004/10/wsat#Durable2PC
   </wscoor:ProtocolIdentifier>
   <wscoor:ParticipantProtocolService>
       <wsa:Address>
          http://supplier.com/DurableParticipant
       </wsa:Address>
     . . .
   </wscoor:ParticipantProtocolService>
  </wscoor:Register>
 </soap:body>
</soap:Envelope>
```

协调器依次调用请求者上的 RegisterResponse 操作。由于注册的参与者使用的协调协议(CoordinatorProtocolService)中需要地址信息，该操作将返回协调器所提供的地址信息，参见清单 10.5。在这个阶段，协调器和请求者都有彼此的端点引用，并能交换协议信息。当参与者通过注册服务注册到协调器时，参与者接收协调器发送的消息。例如，假设正如在清单 10.5 中一样使用两阶段协议，所交换的消息能够是“准备完成(prepare to complete)”消息和“完成(complete)”消息等。另一种情况，假如已经注册到协调器的参与者(供应商服务)希望调用协调器上的消息(诸如“中止”)，参与者将使用 PartcipantProtocolService 地址。

清单 10.5　协调器对于注册操作的响应

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope>
 <soap:Header>
  <wscoor:CoordinationContext soap:mustUnderstand="1">
          <wscoor:Identifier>
             http://supply.com/trans345
          </wscoor:Identifier>
          <wscoor:Expires> 2012 </wscoor:Expires>
          <wscoor:CoordinationType>
            http://schemas.xmlsoap.org/ws/2004/10/wsat
          </wscoor:CoordinationType>
          <wscoor:RegistrationService>
            <wsa:Address>
               http://coordinator.com/registration
            </wsa:Address>
                . . .
          </wscoor:RegistrationService>
            . . .
    </wscoor:CoordinationContext>
 </soap:Header>
 <soap:body>
  <wscoor:RegisterResponse>
   <wscoor:CoordinatorProtocolService>
       <wsa:Address>
          http://coordinator.com/coordinator-service
       </wsa:Address>
   </wscoor:CoordinatorProtocolService>
  </wscoor:RegisterResponse>
 </soap:body>
</soap:Envelope>
```

虽然 WS-Coordination 仅定义了两个核心操作(CreateCoordinationContext 和 Register)，但通过一些附加的特定事务的操作，可扩展协调服务。WS-Coordination 也支持介入。协调器中的注册参与者也能够是一个协调器，从而可以实现介入，并可创建树形的协调器结构。WS-Coordination 将以上这些作为协议的有机组成部分。因此，上下文消息也(能)包含其他参与者的信息以及恢复信息。

4. 在两个应用之间的典型的消息交换

在本节中，我们将给出一个例子。该例显示了两个应用 WS-Coordination 的 Web Service 之间的典型的消息交换。该例源自文献[Cabrera 2005a]。在该例中，假定两个 Web Service(应用 1 和应用 2)将要创建它们自己的协调器(协调器 A 和协调器 B)进行彼此间的交互。

两个应用服务应用 1 和应用，它们都有自己的协调器(协调器 A 和协调器 B)，当活动在两个应用服务应用之间传播时，图 10.19 表明了这两个应用服务是如何进行交互的。例子中的消息交换序列如下所示。

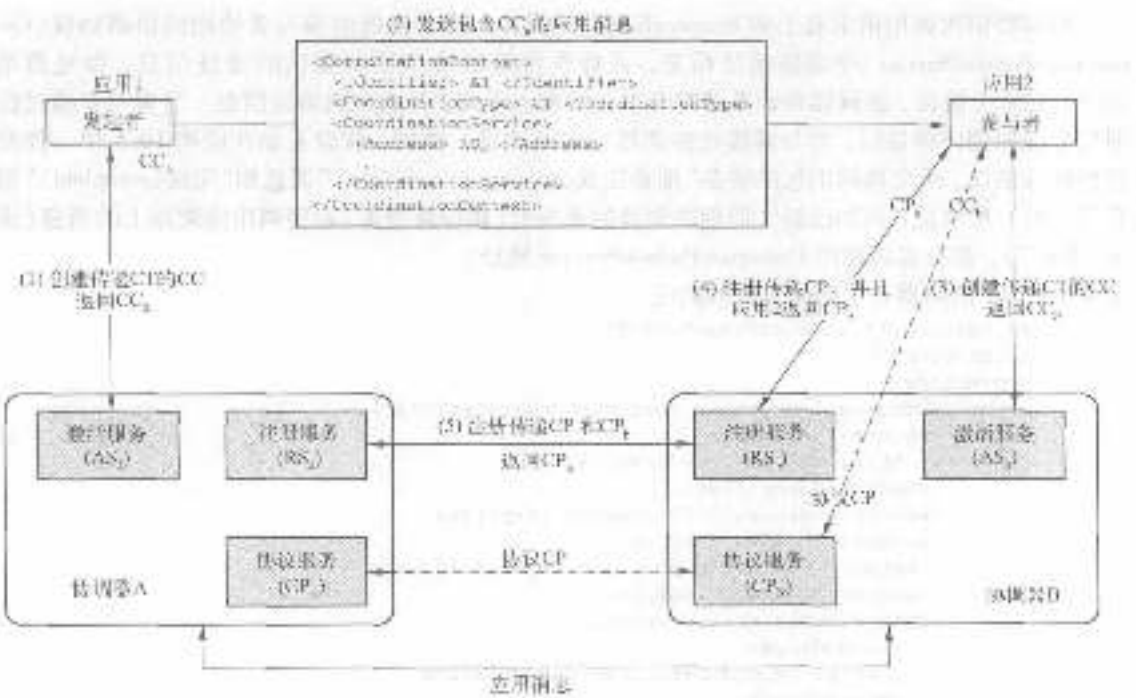


图 10.19 使用 WS-Coordination 并具有它们自己的协调器的两个应用间的交互

(1) 通过调用协调器 A 上的操作 CreateCoordinationContext，应用 1 开始一个新的事务，并指定协调类型(简称 CT)。CT 将管理事务。应用接收协调上下文(CC_0)，该协调上下文包含活动标识符 A_1、协调类型 CT 和对协调器 A 的注册服务 RS_a 的端点引用。

(2) 应用 1 向应用 2 发送一个应用消息。该应用消息包含了作为 SOAP 头部的协调上下文 CC_0。这个消息通知应用 2 使用一个协调协议参与这个活动，其中协调协议通过 CC_0 中的协调类型。接收到邀请的服务既可以注册参与也可以拒绝参与。

(3) 应用 2 将创建自己的协调器，代替使用应用 1 发送的协调器。应用 2 调用协调器 B 上的操作 CreateCoordinationContext，该操作具有协调上下文 CC_0。其中 CC_0 是从应用 1 中接收的。协调器 B 创建它自己的协调器上下文 CC_1。这个新的协调器上下文 CC_1 包含了和协调类型 CC_0 相同的活动标识符和协调类型，但是 CC_1 有它自己的注册服务 RS_b。

(4) 应用 2 确定了协调类型 CT 所支持的协议协议，并在协调器 B 上注册协调协议 CP。由此对于应用 2 和协议实例 CP_b 交换端点引用，现在，在应用 2 和协调器 B 之间能够交换协议 CP 上的所有消息。

(5) 具有协调器 B 的应用 2 的注册触发了另一个注册。它使得协调器 B 将注册转发到协调器 A 的注册服务 RS_a，交换对于协议实例 CP_a 和 CP_b 的端口引用。通过注册和协议选择，两个应用中所涉及的 Web Service 可以设立事务协调器和参与者的传统角色。在此以后，在协调器和参与者之间可以交换协议 CP 上的所有消息，从而在两个应用之间也可以交换协议 CP 上的所有消息。

10.5.2 WS-Transaction

对于各上协调协议实例，WS-Coordination 并没有定义一个协议。可使用 WS-Transaction 定义 WS-Coordination 所使用的事务类型。WS-Transaction 区别于传统的事务协议的一个重要方面是，

WS-Transaction 无须一定要采用同步的请求/响应模型。WS-Transaction 位于 WS-Coordination 协议之上。WS-Coordination 协议本身的通信模式是异步的。WS-Transaction 规范以 WS-Coordination 为基础，并定义了具体的事务处理协议，从而进一步扩展了 WS-Coordination。WS-Transaction 规范监测业务流程中具体的、协调活动的成功与否。基于 WS-Coordination 所提供的结构，WS-Transaction 确保参与业务流程的 Web Service 对于服务结果有共同的理解。

WS-Coordination 提供了上下文管理框架，WS-Transaction 可通过两种方式来使用该框架[Little 2003a]。首先，WS-Transaction 扩展了 WS-Coordination 上下文，从而创建了一个事务上下文。其次，WS-Transaction 增加了激活和注册服务，从而支持了事务模型以及许多相关的事务协调协议。

在 WS-Coordination 框架中可执行两类事务（协调协议）：原子事务和业务活动。WS-AtomicTransaction[Cabrera 2005b]规定了原子事务，而 WS-BusinessActivity[Cabrera 2005c]则规定了长事务。对于这两个协议，WS-Transaction 支持两类事务。对于可信域中的短暂的原子工作单元，适合采用原子事务。对于包含不同信任域的活动的、长时间运行的工作单元，适合采用业务活动事务。每一种事务类型都可应用许多事务协调协议，例如两阶段提交（持久的和易失的）协议、参与者完成协议、协调者完成协议。不同的参与者可以选择其中的一个或多个协议进行注册。基于这些事务协调协议，明确地定义了消息的数量和类型，参与者从而可通过协调器交换消息。

1. 原子事务

原子事务可比作传统的分布式数据库事务模型（短暂的原子事务）。紧耦合的系统需常需要"要么全成，要么全不做"的结果。对于紧耦合系统，提供了一些简单的契约协调协议。

在单个的企业内部，当客户端的操作需要跨不同的内部应用时，适合使用 WS-AtomicTransaction[Cabrera 2005b]。正如本章前面所讨论的，若使用传统的 ACID 语义将不同企业的一些事务资源捆合在一起，这并不是一种明智的做法。

WS-AtomicTransaction 类型到已有的 ACID 事务标准，并且 WS-AtomicTransaction 提供了二类事务协调协议。同一个活动中的不同的参与者可以注册不同的事务协调协议，诸如下面所概述的协议。

完成协议可以控制原子事务的应用。当应用开始一个原子事务时，应用将设置一个支持 WS-AtomicTransaction 协议的协调器。应用将注册该协议。在完成所有的应用工作后，应用将指示协调器提交或中止事务。协议中的动作顺序如图 10.20 所示，其中实线弧表示了协调器生成的动作，而虚线弧则表示了参与者生成的动作，该状态转换图假定：协调器将从参与者中接收"提交（commit）"消息或"回退（rollback）"消息。因此，应用可以检测所需的结果成功与否。

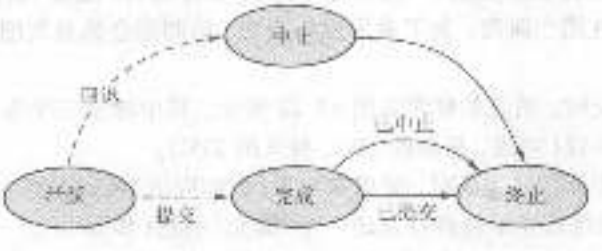


图 10.20 完成协调转换

源自：[Cabrera 2005b]

持久的2PC和传统的2PC一样都可用于确保参与者之间的原子性。该协议基于两阶段提交以及中止技术。当活动没有全部成功时，中止技术将会回滚活动中的操作，这是一个默认的行为。当一组参与者都要执行同样的操作(无论是提交还是中止)时，可使用持久的2PC来协调个别的这些参与者。在完成协议中，当协调器接收一个"提交(commit)"通知后，对于持久的2PC的参与者，协调器将开始"准备(prepare)"阶段(阶段1)，注册这个协议的所有参与者必须以一个"准备好(prepared)"通知或"已中止(aborted)"通知进行响应。在第二个(提交)阶段中，假如协调器已经从参与者那里接收了"准备好"响应，则协调器将会给所有的参与者发送一个"提交"通知，表明一切正常。然后，所有的参与者将会返回一个"已提交"进行确认。另一种情况，假如协调器从一个或多个参与者那里收到"已中止"响应，则会对其他的所有参与者发送一个"回退"通知，表明前面的操作失败。然后，所有的参与者将会返回一个"已中止"进行确认。

图10.21显示了具有持久的两阶段提交协议的WS-Transaction原子事务的状态变迁。该图还显示了协调器和参与者之间的消息交换。其中，实线表示了协调器生成的消息，虚线表示了参与者生成的消息。

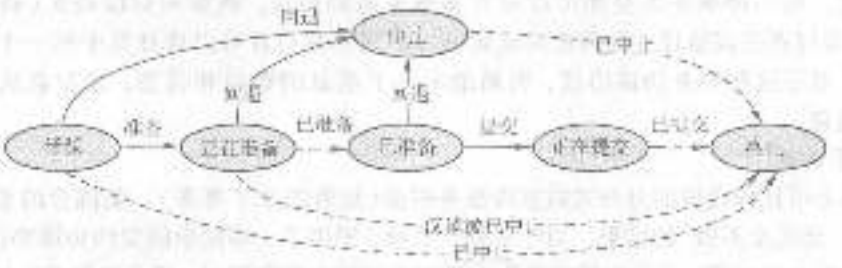


图10.21 持久的两阶段提交状态转换

源自：[Cabrera 2005b]

易失的2PC。正如前面讨论的，在事务中访问持久化的数据存储，将会导致本地资源的锁定，从而导致性能瓶颈。另一种方式是在数据本地存储副本上进行操作，这可大大地提高性能。然而，在事务提交之前，事务在易失数据上的工作最终还需写回到后端企业信息系统，诸如数据库、ERP系统。应用程序的状态最终共享在这些系统中。因此，需要另外的协调基础架构。例如，为了将更新后的缓存状态刷新到后端服务器，需要在2PC开始之前通知参与者。在WS-AtomicTransaction规范中，易失的2PC可以实现这一点。易失的2PC协议基本类似于持久的2PC协议，但是也有一些细微的差别。在第一阶段，对于那些已经注册了易失2PC的参与者，协调器首先向那些参与者发送"准备"消息，然后方向注册了持久的2PC的参与者发送该消息。仅当易失的2PC下所有参与者都已经投票了"已准备"，协调器之会将"准备"消息发送给所有注册了持久的2PC的参与者。在第二阶段，为了表示操作成功，协调器会给易失的参与者和持久的参与者发送"提交"通知。

在成功地进行提交时，消息的顺序如图10.22所示，其中涉及二个参与者，每一个对应了上面所提到的一个协调协议(完成、易失的2PC、持久的2PC)：

(1) 当所有的工作都已经完成时，完成参与者(例如应用)将试图提交工作，从而发起一个提交流程。因此，参与者将首先给协调器发送一个"提交"消息(步骤1)。

(2) 对于易失的2PC参与者，协调器将给参与者发送一个"准备"消息，从而发起一个易失的2PC的准备阶段(步骤2)。每一个易失的2PC参与者在成功地完成提交准备后，将发送一个"已准备"消息。



图 10.23 成功地提交 WS-AtomicTransaction 事务的场景
资料来源：[Cabrera 2005b]

(3) 在协调器收到所有的易失的 2PC 参与者的"已准备"消息后，协调器将给持久的 2PC 参与者发送一个"准备"消息，从而发起一个持久的 2PC 的准备阶段(步骤 3)。每一个持久的 2PC 参与者在成功地完成提交准备后，将发送一个"已准备"消息。

(4) 在协调器收到所有的持久的 2PC 参与者的"已准备"消息后，协调器决定提交，将给完成参与者发送一个"已提交"消息(步骤 4a)，并给易失的 2PC 参与者和持久的 2PC 参与者发送"提交"消息(步骤 4b 和 4c)。注意，在 4a、4b 和 4c 之间没有顺序限制。

对于传统的原子事务，WS-AtomicTransaction 优于已有的协调服务。WS-AtomicTransaction 基于 SOAP，从而增强了互操作性，因此在不同平台上实现的参与者都可使用 WS-AtomicTransaction。WS-AtomicTransaction 也可与其他的一些标准进行协同工作，诸如 WS-Security、WS-Policy 和 WS-Trust，从而表达策略断言，并可在任意网络结构中提供安全的端到端操作。

2. 业务活动

WS-BusinessActivity 协调类型[Cabrera 2005c]支持可能长时间运行的活动的事务协调。这些可能长时间运行的活动不同于原子事务的之处在于：它们可能需要花比较长的时间才能完成，并且对于长时间运行并不一定要求额外的资源。这使得在一个具体的时间短内可能有许多客户都预定相同的产品，然而最终仅有一个客户可能获取该产品(假如该产品是库存中的最后一件产品)。这也避免了：当资源被无限期地锁定时，出现拒绝服务现象。对于业务活动所使用的资源的用户，为了使得他们访问的延时能够最小化，在完成整个活动之前需要实现一些临时操作。在业务活动中，为了处理异常，需要应用一些相应的业务逻辑。参与者被视为业务任务(作用域)，它们是所注册的业务活动的后代。参与者可以决定委托一个业务活动(例如，委托其他服务进行处理)，或者在被要求之前，参与者预先声明它的结果。

为了最大化地实现灵活性，业务活动并不维护所有的 ACID 事务特性。为了选择最合适的参与者并取消其他的参与者，业务活动可查询多个参与者。业务活动已完成的任务(例如事务)先于业务活动的完成，因此可以放松隔离级。事实上，这些任务是试探性的，并且假如需要补偿这些任务，则需要通过业务逻辑来进行补偿。

现以一个涉及多个服务的业务活动为例。制造商(客户端)服务向多个供应商服务发送订购单。制造商服务按照选择标准选择一个报价，并取消对其他供应商的订购。因此对于应用，WS-BusinessActivity 协调协议增加了一个标准的结构，但是同时允许任何应用逻辑处理该协调。在另一个例子中，制造商可以发起一个订购单流程。该流程可以包含不同的活动。这些活动必须都成

功地完成，但可以同时地运行（至少在某种程度上）。这些活动诸如信用卡检查、库存控制、账单和装运。基于 WS-Transaction 和 WS-Coordination 这两个协议，可确保这些任务作为一个单元成功或失败。

在作用域内可划分业务活动。作用域定义为完成一个任务所需执行的操作集。对于在参与完成任务的不同的参与者，可以在这些参与者中执行这些操作。为了完成一个业务活动，能够创建作用域的层次结构。作用域的嵌套可以有不同的选项。例如，父亲可以选择在结果协议中包含哪些孩子，因此可能导致非原子化的结果。WS-BusinessActivity 协议定义了一个协议组，可基于业务级的情况放松原子性（Weerawarana 2005）。此外，父亲可以捕捉孩子抛出的例外，并应用例外处理程序，然后继续进行相关的处理。为了达到所要实现的目标，在步骤之间需要持久地保存业务活动的状态，即使在出现例外的情况下。

协调器可以按照"要么全做，要么全不做"的方式协调业务活动的参与者。业务活动中的协调器并不像原子事务中的协调器一样严格。驱动活动的应用将决定协调器的行为。参与者在业务活动中注册后，可在事务中的任何时间点离开活动。这与原子事务处理参与者的方式完全相反。在原子事务中，一旦参与者注册后，协调类型必须确认其他的参与者。假如一个参与者选择"取消"一个任务，则它也强迫其他参与者取消该任务。

正如原子事务一样，WS-BusinessActivity 定义了两个协调协议：参与者完成的业务协定（Business Agreement with Participant Completion）和协调器完成的业务协定（Business Agreement with Coordinator Completion）。

在参与者完成的业务协定中，参与者（子）活动在创建后的初始状态是"活动"状态。假如已经完成了参与者任务，并希望进一步涉及业务活动，则它必须能够补偿已完成的工作。在这种情况下，它向协调器发送一个"已完成"消息，并希望从协调器接收最终的业务活动结果。结果既可以是"关闭（close）"消息，也可以是"补偿（compensate）"。前者意味业务活动已成功完成。后者表明协调器活动需要参与者取消前面工作所产生的影响。或者，假如参与者活动完成了所创建的工作，并决定不再参与 WS-BusinessActivity 的作用域，然后参与者能够单方面向协调器发送一个"已退出（exited）"消息，这相当于参与者从 Web Service 事务中退出。

图 10.23 显示了含参与者完成协议的 WS-BusinessActivity 的状态变化，并显示了协调器和参与者之间的消息交换。像以前一样，实线表示了协调器生成的消息，虚线表示了参与者消息。

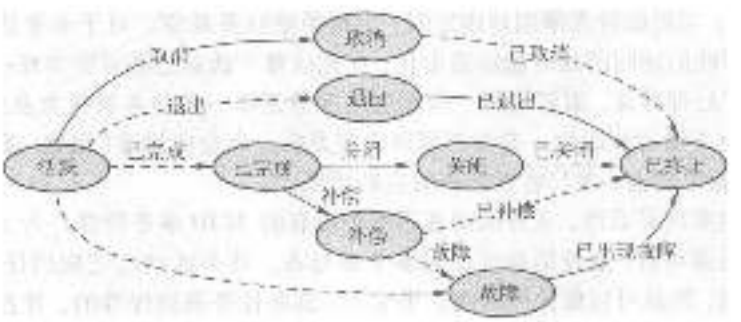


图 10.23 含参与者完成协议的业务协定的状态转换

源自：（Cabrera 2005c）

协调器完成的业务协定与"参与者完成的业务协定"类似，不同点在于：在该协议中，即使能够补偿业务活动，参与者也不能单方面决定退出业务活动。当参与者接收到完成工作的请求时，

参与者任务依赖协调器通知它。为了完成这一工作，协调器向参与者发送"完成(complete)"消息。然后，参与者采取类似于在"参与者完成的业务协定"中的操作。

图 10.24 阐明了以上各点。该图显示了一个订单处理应用(改变自文献[Cabrera 2005c])。基于参与者完成协议的业务协定，制造商服务向三个供应商服务进行询价，然后选择最合适的一家供应商进行合作。协议交互中的各个步骤如下：



图 10.24 基于业务活动的订单处理样例

(1) 对于这个业务活动，制造商服务(协调器)向每一个销售者发送一个 PurchaseOrder 消息，消息的头部为 CoordinationContext。可同时发送这些消息。

(2) 所有的三个供应商(参与者)注册参与者完成协议的业务协定。

(3) 从这一点向下，三个供应商有不同的流，诸如：

a) 供应商 1 决定报价。它使用协调协议的"故障(Fault)"消息通知制造商服务。

b) 供应商 2 和供应商 3 进行报价。各供应商分别通知制造商服务，除了价格信息，还含有协调协议器协议的"已完成(completed)"消息。

(4) 在评判两个供应商的报价后，制造商选择供应商 2。然后它：

a) 向供应商 2 发送协调器协议"关闭(close)"消息(用以表明成功)。

b) 向供应商 3 发送协调器协议"补偿(compensate)"消息。

10.6 Web Service 组合应用框架

我们在前面的章节中已经描述了 Web Service 事务模型。长期的业务流程的执行可能包含一些复合的 Web Service 应用。然而对于这些应用中的共享上下文的管理，Web Service 事务模型并没有提供一个共同的方法。上下文使得操作可以共享一个共同的结果。在事务中，若应用包含分组的、复合的 Web Service，则上下文是非常重要的。

为了解决一些不足，OASIS Web Service 复合应用程序框架(WS-CAF)定义了上下文管理服务。上下文管理服务管理和协调整个复合的 Web Service 应用中共享上下文数据的使用。WS-CAF 规范集处理了 Web Service 事务标准化中没有得到解决的两个主要问题，具体如下：

(1) 一个大的工作单元(例如处理一个订购单，或者执行事务中的一系列长 Web Service)可能包含多个 Web Service。WS-CAF 解决了跨多个 Web Service 的共享的信息上下文的管理问题。WS-CAF 提供了一个灵活的、标准的管理 Web Service 上下文的机制，从而提高了互操作性，并避

免了不必要的复制。

(2) WS-CAF 是一个支持许多 Web Service 事务协议(包括 WS-Transaction)以及它们之间的互操作性的开放的、"可插拔的"框架。在不同的事务模型(诸如 MQ Series 和 J2EE 事务)之间, WS-CAF 能够充当一个桥梁。

WS-CAF 提供了一个抽象的体系结构视图, 支持 Web Service 间的长时间的交互。WS-CAF 可以分为三个主要部分, 每一部分定义了不同的功能。这三个部分[Bunting 2003a]是: Web Service 上下文(WS-CTX)、Web Service 协调框架(WS-CF)和 Web Service 事务管理(WS-TXM)。其中, WS-CTX 是一个上下文管理的轻量级框架; WS-CF 是一个管理上下文增加和生命周期的共享机制; WS-TXM 基于 WS-CF 提供了事务协调。

WS-CAF 的各个部分主要是用于补充 Web Service 编配和编排技术(诸如 BPEL 和 WS-Choreography), 它们将与已有的 Web Service 规范(诸如 WS-Security 和 WS-Reliability)相互协作。WS-CAF 将它自身限定于管理一些特定的信息, 若要支持不同的分布式计算基础架构, 则需要用到这些信息, 同时可使用其他标准(诸如 BPEL)创建这类相互依存。BPEL 上下文管理系统定义了如何在流执行引擎中关联消息, 但是并没有涉及如何在流中跨 Web Service 来共享上下文, 而这是 WS-CAF 的职责。

10.6.1 Web Service 上下文

当 Web Service 完成相关的活动时, 诸如和数据管理资源的多个交互、交互显示、自动化业务流程, 这些 Web Service 将共享一个共同的上下文。通过使用共享上下文, 来自不同源的 Web Service 共享了共同的系统信息, 因此它们可以分别成为同一个应用的一部分。当多个 Web Service 在一个授权会话的作用域中执行, 应用所依赖的共享上下文通常包括共同的安全域。当活动中的每一方需要知道其他参与者是否已经成功地完成它们的工作时, 应用所依赖的共享上下文则通常包括共同的输出结果协商。共享上下文通常还包括建立到后端系统的持续的连接等[Webber 2003b]、[Newcomer 2005]。

WS-CTX 定义了上下文、上下文共享的作用域、上下文管理的基本规则。WS-CTX 描述了如何定义一个活动, 活动中的多个 Web Service 通过共享上下文关联在一起。活动本质上是一种划定特定应用的工作的方法[Bunting 2003b]。大型应用通常涉及许多活动, WS-CTX 表示了这些活动的 Web Service 交互。WS-CTX 定义了分界点。分界点指定了活动的起点和终点。在活动的生命周期中, WS-CTX 还注册 Web Service, 以便这些服务能够成为参与者。WS-CTX 还管理和扩大与活动关联的上下文, 并跨网络传播上下文信息[Bunting 2003b]。

WS-CTX 并不是具体针对一个服务类型或应用域, 它是一个比较底层和基础的服务, 主要通过上下文管理抽象活动实体。WS-CTX 的主要组件如下[Bunting 2003b]:

上下文服务: 上下文服务定义了活动的作用域, 并定义了如何在分布式环境中引用和传播消息。为了管理活动, 上下文服务维护了一个与执行环境相关联的共享上下文信息库。无论何时在活动的作用域中交换消息, 上下文服务都可提供相关的上下文, 然后可以传送那些消息[Webber 2003b]。上下文服务也管理上下文的层次, 从而支持嵌套和并发。

上下文: 通过共享共同的信息, 上下文将一组消息关联到一个活动。为了这个目的, 上下文包含了与同一个活动相关联的多个 Web Service 所需的信息。随着应用和流程运行, 服务可以动态更新这些信息。

在 WS-CTX 中, 在一个活动涉及的服务横跨两个类别, 即应用服务和活动周期服务。

除了上下文服务, 活动中的每一个 Web Service 参与者可以将一个活动生命周期服务(ALS)注册到上下文服务中。ALS 参与活动的生命周期(当活动开始、结束等, 将通知)。在执行中, 当

与当前执行环境相关的活动需要上下文时，上下文服务调用每一个被注册的 ALS，并获取针对基本的上下文的补充内容。上下文文档可以传播到各个参与者，最终可装配整个上下文文档。例如，上下文服务可以有一个事务 ALS 和安全性 ALS，它们都可注册到上下文服务中，这两个服务可以定义活动的安全性作用域和事务域。

应用可以使用应用服务来提供它的功能。例如，订购单应用可以有几个活动，这些活动涉及核查客户的信用、核查产品库存、计算订单的最终价格、将账单发送给客户、选择运送者等。对于应用服务和 ALS，活动的定义直接与一组 Web Service（诸如订购单）的需求相关，这些 Web Service 共享共同的信息。活动本身实现为一个 Web Service，并且将服务注册到活动中，以便定义该活动。

ALS、上下文服务、应用服务、应用之间的关系如图 10.25 所示。

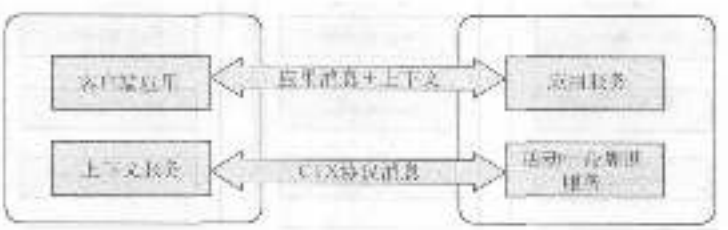


图 10.25　连接 ALS、上下文服务、应用服务和应用

编自：[Bunting 2003a]

10.6.2　Web Service 协调框架

一个大型应用通常与许多活动相关。基于 Web Service 协调框架（WS-CF），可以管理和协调这些活动间的 Web Service 交互。WS-CF 建立在 WS-CTX 规范的基础上。WS-CF 提供了一个可插入到 WS-CTX 中的协调服务。WS-CF 允许向其中加入应用和服务，并可基于每个服务或应用进行定制。WS-CF 支持事务协议的独立性，并可映射到多个底层技术，从而满足了 Web Service 的实际需求。

WS-CF 定义了分界点，指定了被协调活动的起点和终点。在分界点可进行参与者的协调。例如在该点，可将合适的 SOAP 消息发送给参与者。WS-CTX 提供了默认的上下文结构，通过增强这一结构，可跨网络传播具体的协调信息。

WS-CF 包含下列主要的构建块[Bunting 2003c]：

协调器（或称作协调者）：对于在协调点触发的参与者（诸如活动），协调器组件提供了参与者的注册接口。协调器负责将活动的结果传送到注册活动列表。

参与者：该组件提供了一些操作，这些操作是协调序列处理的一部分。

协调服务：对于一个具体的、可定制的协调模型，协调服务定义了行为。协调服务提供了一个处理模式。在进行输出处理时，将使用该处理模式。例如，可将协调服务实现为一个 ACID 事务服务。ACID 事务服务提供两阶段协议和扩展的事务模式，诸如协作、开放的嵌套事务、封闭的嵌套事务，非事务模式（诸如内聚和关联）。可使用协调服务对相关的非事务活动进行分组。

图 10.26 说明了如何将一些单个的 Web Service 和复杂应用作为参与者注册到协调器中。协调器负责上下文管理，以及将相关的 Web Service 的执行结果通知参与者。如图所示，协调者可将自身注册到另一个协调器中，从而成为一个参与者，因此这提高了互操作性。

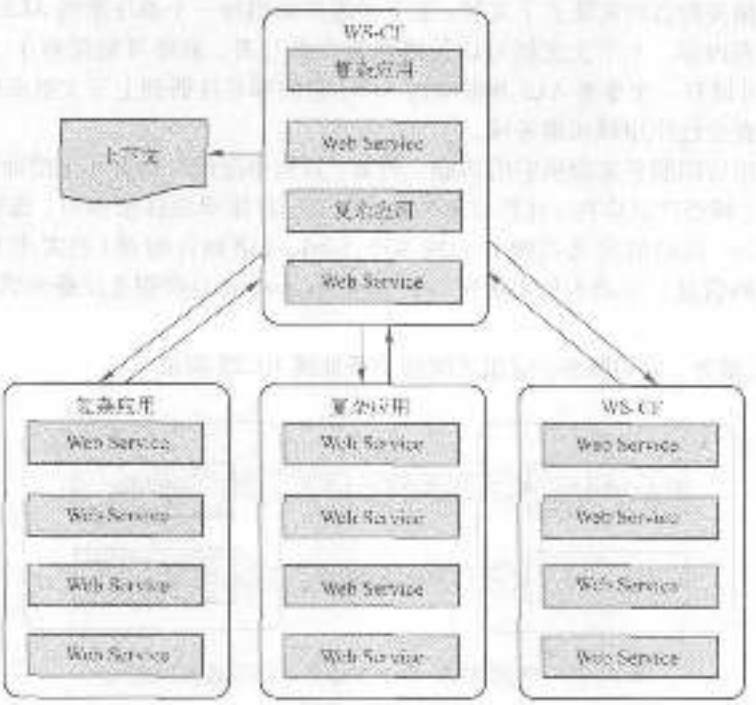


图 10.26 协调器与 Web Service、复杂应用的关系

WS-CF 表面上似乎类似于 WS-Coordination，主要的差别在于，WS-CF 比 WS-Coordination 更多地定义了协调器的体系结构，WS-Coordination 将许多事情交给使用它的服务[Webber 2003b]。在许多方面，WS-CF 可以视为是 WS-Coordination 的超集。

10.6.3 Web Service 事务管理

Web Service 事务管理(WS-TXM)定义了一组可插拔的事务协议(两阶段提交、长期运行的动作、业务流程)。基于一系列的相关的 Web Service 的执行结果，协调器可以使用这些事务协议来协商参与者所要执行的一组操作。通过使用共享的上下文(作用域)，相关的 Web Service 的执行可以联系在一起。这些 Web Service 的执行既可以是嵌套的(父子关系)，也可以是并发的。

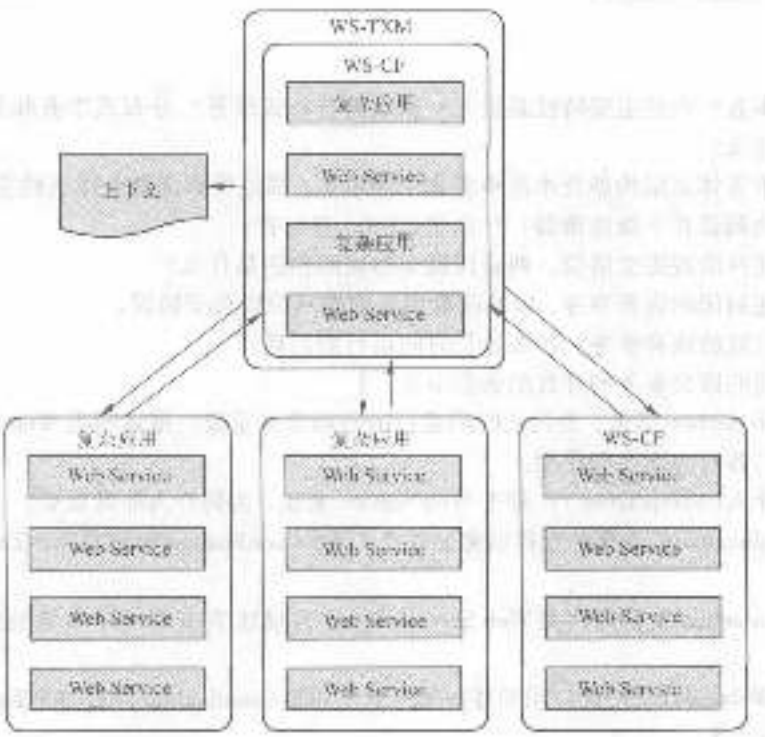
为了实现 WS-TXM 所声称的目标，它建立在 WS-CF 规范和 WS-CTX 规范的基础之上。WS-TXM 定义了具体的协调器服务和参与者服务，并扩大了分布式上下文。图 10.27 说明了 WS-TXM 协议的分层。

目前，WS-TXM 定义了三个事务协议[Bunting 2003d]，它们是 ACID 事务、长期运行的动作、业务流程事务。

ACID 事务类似于传统的数据库事务，帮助 Web Service 在已有的事务基础架构上实现互操作性。在 ACID 模型中，每一个活动都将绑定到一个事务作用域，从而使得活动的结果将自动触发相关事务的终止(提交或回退)。

长期运行的动作(LRA)是一个活动或是一组活动。LRA 并不需要具有 ACID 特性。LRA 可以使用前向(补偿)或后向错误恢复来确保原子性。对于补偿操作，LRA 定义了补偿动作的触发器，并定义了执行触发器的条件。在实施实现时，也必须要考虑到隔离级的选择问题。

最后，业务流程事务是一个活动或一组活动。业务流程负责完成一些具体应用的工作。根据具体的应用需求，可将业务流程结构化为原子事务的集合或者 LRA 的集合。业务流程协议明确地不同于其他的任何事务模型（诸如 WS-Transaction）。业务流程协议主要面向电子服务应用，它的具体目标是将异构的事务域组织成一个企业对企业（B2B）事务。例如，基于业务流程事务模型，长时间运行的 Web Service 事务可以横跨消息传送、工作流和传统的 ACID 事务，从而企业可以利用它们已有的企业资产[Webber 2003b]。


WS-TXM
WS-CF
上下文
Web Service
Web Service
Web Service

极其重要的。Web Service 可以涉及许多企业，这些企业由于不同的原因参与事务，诸如产品或服务、支付安排、监视和控制。Web Service 事务有助于实现一致性，例如事务既可以完整地执行（成功），也可以作为一个整体失败。Web Service 事务包含两个不同类型的事务活动：原子动作（或短事务）和长事务活动。

为了支持 Web Service 事务，目前已设计了三个标准的规范——WS-Coordination、WS-Transaction 和 Web Service 复合应用程序框架。这三个事务规范实现了 Web Service 事务模型和协调协议，并支持不同类型的业务协议。

复习题

- 什么是事务？它的主要特性是什么？什么是分布式事务？分布式事务和集中式事务的不同点是什么？
- 分布式事务体系结构涉及本地事务和全局事务，描述分布式事务体系的主要构建块。
- 什么是协调器和下级协调器？什么是（事务）参与者？
- 简要描述两阶段提交协议。两阶段提交协议的作用是什么？
- 简要描述封闭的嵌套事务，以及嵌套事务中的两阶段提交协议。
- 什么是开放的嵌套事务？什么是长时间运行的活动？
- 比较封闭的嵌套事务和开放的嵌套事务。
- 对于 Web Service 事务，为何长时间运行的活动非常重要？描述两类 Web Service 事务，并讨论它们各自的优点和作用。
- 什么是介入（interposition）？对于 Web Service 来说，为何介入非常重要？
- WS-Coordination 的主要的组件服务是什么？WS-Coordination 如何与 WS-Transaction 相互协作的？
- 描述 Transaction 支持的两类 Web Service 事务，并描述 Web Service 事务的最重要的一些协调协议。
- 什么是 Web Service 复合应用程序框架？它与 WS-Coordination 以及 WS-Transaction 的不同之处是什么？

练习

10.1 假定一个应用涉及订购单服务、协调器服务。订购单服务将产品订单转发给它的供应商，并调用一些 Web Service，从而向供应商订购一些产品。协调器服务代表 WS-Coordination 充当激活和注册服务。此外，假定订购单服务调用一个称为 checkInventory( ) 的操作来完成它的任务。开发一个简单的顺序图（sequence diagram），用来说明这个应用的 WS-Coordination 流。

10.2 假定一个应用涉及销售服务、库存服务、装运服务和协调器服务代表。销售服务接收产品订单，并希望查询库存服务。库存服务提供了仓库中产品的库存信息。装运服务对货物的装运进行调度。协调器服务代表 WS-Coordination 充当激活和注册服务。开发一个简单的顺序图（sequence diagram），用来说明这个应用的 WS-Coordination 流。

10.3 修改练习 10.2，使其包含持久的两阶段提交。假如所有的产品都不缺货，活动将要提交，否则整个活动都将回退。请画出顺序图，并使用规范（诸如 WS-Coordination 和 WS-Transaction）中定义的消息交换来说明发起者和参与者之间是如何交换消息的（排除出故障的情况）。

10.4　WS-BusinessActivity 协议代替 WS-AtomicActivity 协议。这个解决方案包括一个订购单服务(代表客户)、一个与订购单服务关联的协调服务、一个销售服务(代表供应商)。假定购买者想要购买三个具体的产品，并且希望知道供应商是否能够提供所有这三类产品。假如供应商能提供所有的这三类产品，则分销商将确认这个订单。假如发送有问题，或者这三类产品的提供存在问题，购买者则要重新考虑整个订购活动。例如，购买者可能想要接触其他的一些供应商，询问这三样产品的供货情况，并且最终将可能从当前的供应商那里仅购买一样产品。为了解答本练习，将要考虑 WS-BusinessActivity 是嵌套的。这意味着整个采购活动将认为是一个父活动，这个父活动可能有许多子活动，每个子活动代表一样产品的订购。注意，每个子活动可包含一个或多个 WS-AtomicTransactions。这个活动使用 WS-Coordination 和 WS-Transaction。说明活动中的实际的消息交换。

10.5　扩充练习 10.1 中的解决方案，假定应用使用了持久的两阶段提交，并假定应用涉及一个发起者(订购单)和一个参与者(供应商)。请画出顺序图，并使用规范(诸如 WS-Coordination 和 WS-Transaction)中定义的消息交换来说明发起者和参与者之间是如何交换消息的(排除出故障的情况)。

10.6　扩充练习 10.3 中的解决方案，假定有一个参与者(供应商服务)发起了一个回退。

第六部分　服务安全性与策略

第 11 章　安全的 Web Service

学习目标

对于 Web Service 应用而言，安全性是一个重要问题。即使对于有可能建立动态的、短期关联关系的关键任务，Web Service 通常也是基于不安全的互联网。Web Service 的灵活性和优点在一定程度上导致潜在的安全缺陷。这就要求采取措施以抵御各种攻击。对于组织和他们的客户来说，通过应用全面的安全模型来确保 Web Service 的完整性、私密性和安全性是非常重要的。

在本章中，我们首先将要描述 Web Service 的安全模型。这个安全模型依赖于当前可用的一些技术以及未来 Web Service 应用不断发展的安全需求。这需要将技术（保证消息的安全性）和业务（策略、风险、信任）统一起来。通过本章的学习，读者将可掌握下列关键概念：

- Web Service 常见安全威胁及对策。
- 网络层和应用层安全机制。
- 安全性体系结构。
- XML 安全服务和标准，如 XML Encryption、XML Signature 和 SAML。
- Web Service 安全性的使用案例。
- 使用 WS-Security 来开发基于 SOA 的解决方案。
- WS-Security 标准族。

11.1　Web Service 安全性

在 SOA 中，服务的一个主要特点是：它们将以无法事先预料的方式相互组合在一起。当前 Web Service 技术正是沿着这个方向发展的，并已有一个良好的开端。但是，在 SOA 可以支持关键任务、Web Service 长事务之前，必须先解决各种严重的安全性问题。开始进行 Web Service 集合的企业更加需要依赖于电子方案来保护商业机密信息、事务和通信。

在真实世界中，企业依赖于传统的安全方法来护卫机密的企业信息。传统的企业安全性几乎都是集中在利用工具（如防火墙和内容过滤器）来阻止入侵者上面。在私有的、可信任的企业网络和外部不可信的网络之间，防火墙充当了一个安全接口。防火墙可用于进出企业网络的访问控制。根据数据包的源地址和目标地址，以及源端口和目标端口，内容过滤器可允许或拒绝数据包流。访问控制包括对应用服务和特定主机之类资源的访问。当恰当地配置并实施周密的安全策略后，防火墙可以保护企业网络不被入侵和受到安全威胁。

传统的分布式计算的安全性通过“安全岛(islands of security)”进行建模，描述了孤立的网络或子网上的系统和用户。将网络看作为岛屿，有着自己的防线安全，但是网络内的用户认为是可信任的，而在网络之外的用户则被认为是不被信任的。因此，一直到近来，通过对穿越企业边界的每个应用都进行授权和认证的方式，企业控制和管理对资源的访问。但是这种孤立的安全方式不仅需要花费大量的时间，而且很容易出现各种错误。随着企业电子商务应用的增多，以及企业间在线交互越来越复杂，这种技术已不再适用了。这种“信任”和“非信任”二分逻辑就演变成面向服务的模型，因为应用可以访问跨越一个或多个企业的系统上的“自发”的服务。与传统垂直应用相比，在一个接口驱动的环境中，将暴露更多的应用功能。信任组的概念不再有意义。取而代之的是，企业必须制订应用于整个企业网络(包括从外部邀请的参与者)的策略，以分级或分层的方式管理安全性[Bloomberg 2004]。我们需要的是一个完备的 Web Service 安全解决方案，能方便地管理并满足各种应用需求以及开发者的安全需求，并提供诸如性能、部署速度和可伸缩性等能力，以及适应技术发展(如无线服务)的灵活性。

对 Web Service 进行集成的需求迫使企业放弃私有的通信网络，转而采用开放的公共网络，如互联网。在互联网中，企业需要向客户、供应商和业务合作伙伴开放私有的网络应用和信息资产。这意味着允许客户和业务合作伙伴进入私有的企业网络，即允许通过防火墙。但是需要以一种有选择和可控制的方式进入，这样客户和业务合作伙伴只能访问允许访问的应用。然而，已有的网络设施，如路由器、防火墙和负载平衡器，它们都在网络层而不是应用层工作，所以完全不能提供 Web Service 所需要的应用层安全性要求。这就意味着，我们关注的重点要谨慎地从网络层安全转移到应用层安全，以便保护与大量贸易伙伴、客户和供应商在不安全的网络(如互联网)上交互的业务信息、事务和通信。Web Service 安全涉及多种计算机环境、通信协议、策略和过程的复杂交互。在实施一个一致的 Web Service 安全方法时必须考虑到这些方面。

相比于传统的安全性，Web Service 除了在访问(认证、授权) 机密性、不可否认性和信息的完整性等方面有特定的需求，还提出了一些新的安全性需求。这些都是本章所关心的议题。但是，在我们介绍和解释这些问题之前，理解 Web Service 面临哪些安全威胁是很重要的。

11.1.1 Web Service 面临的安全性威胁

Web Service 的目标是向新的以及原有的应用暴露标准化的接口，从而可以构建横跨不同组织的网络的应用和业务流程。通过服务聚集，可以更容易地创建新的增值服务，例如允许经销商查看厂商产品的可用性和价格、下订单、在线跟踪订单的状态。但是因为以前的技术没有暴露这个层次的业务应用，这也引入了新的安全威胁。一个重要的问题是，Web Service 可穿越防火墙，在应用层隐蔽了它们的作用。例如，Web Service 设计为通过端口 80 穿过网络防火墙，只提供很少的、基本的内容检查。需要有应用层的安全性来保护 XML 和 Web 服务相关的安全威胁。在提供基于 IP 的访问控制和网络层保护方面，网络防火墙依然是至关重要的，因此需要一个服务(或 XML)防火墙来保护 Web Service。为达到这个目标，服务防火墙必须了解请求者是谁，正在请求的信息是什么，以及正在请求什么特定服务。此外，它们必须能拦截进入的 XML 流量，并基于该流量中的内容采取基于策略的动作。若要提供必要的安全性来保护 Web Service 和 SOA 环境，这类功能是先决条件。

另一个安全性问题是，Web Service 是标准化的和自描述的。因为 Web Service 接口是标准化的，所以可按类似的方式攻击它们。和私有接口相比，入侵者对于标准接口了解得更多，因此可以更容易地访问标准接口。此外，SOAP 消息为每个消息提供信息和结构。例如，一个打包的应用可能暴露大量的重要操作，这些操作都可以通过端口 80 访问。另外，攻击者可以获取更多的有用信息。因为 WSDL 规范和 UDDI 条目都是自描述的，所以攻击者可以获得详细信息，从而使

入侵者可以获得到关键任务应用的入口。WSDL 文档提供有关每个 Web Service 的详尽信息，包括服务所在的位置、如何访问它、向它发送何种信息，以及开发者期望接收何种信息。这给潜在的入侵者提供了详尽的信息，使得他们可以非法访问服务。这点也表明了我们需要将关注的重点从网络层安全变为应用层安全。

Web Service 应用必须在 OSI 协议栈的应用层操作，这一事实产生了一个挑战，因为对于较低层次的设备来说，所有数据网络流量看起来是一样的。当前网络层和传输安全层解决方案使用传统的机制，如防火墙、路由器、代理、负载平衡、限制访问已知的 IP 地址，并使用安全套接字层来确保独立于 Web Service 应用编程的 Web 事务的安全。但是由于它们只适用于网络层，因此这些解决方案是不合适的。它们必须能处理 XML。应用层安全在 Web Service 应用中起着关键作用，涉及检查网络流量的内容、对内容做验证和授权决定，以及验证 XML 事务的个别部分，并执行安全策略规定的动作。应用层安全还要解决机密性和私密性——使用加密来保护消息，确保消息完整性，并提供审计功能。

Web Service 安全性与采用分布式技术的应用的安全性有很多相似性。Web Service 安全性基于：

- 资源通过应用来管理和保护。
- 与应用中基本技术相关的漏洞。
- 与应用的特定逻辑相关的漏洞。
- 用于减轻这些风险的技术。

为解决 Web Service 安全性问题，WS-I Basic Security Profile [Davis 20004] 已经识别了几种 Web Service 安全威胁和挑战，以及用于减轻每种威胁的对策（技术和协议）。下面我们将 WS-I Basic Security Profile 中列出的威胁分为四大类，阐述 Web Service 中一些最频繁出现的安全问题：

（1）*未经授权的访问*：消息内的信息被非预期的参与者或未被授权的参与者看到，如一个未经授权方获得一个信用卡号。

（2）*未经授权的消息修改*：因为攻击者可以修改部分（或整个）消息，所以这些威胁影响消息完整性。通过插入、删除或修改信息创建者创建的信息，可以改变消息中的信息，接收者因此会错误理解信息创建者的意图。例如，攻击者可以删除一部分消息，或修改一部分消息，或者向消息中插入附加信息。这类安全问题可能包括：修改附件、重放攻击（拦截被签名的消息，然后再发送回目标站点）、会话劫持，伪造声明以及伪造消息。

（3）*中间人*：在这类攻击中，攻击者可能攻占一个 SOAP 中介，然后在 Web Service 请求者和最终接收者之间拦截消息。原来的参与方认为他们正在互相通信。攻击者可能仅仅访问消息，也有可能修改它们。称为路由绕道（routing detours）的威胁也包含“中间人”攻击形式，攻占路由信息。为了将敏感的消息传到外部位置，可修改路由信息（不管是在 HTTP 头部或是在 WS-Routing 头部）。可以从消息中删除路由的跟踪，因此接收的应用将无法意识到发生了路由绕道。互相认证技术可用于减轻这种攻击的威胁。

（4）*拒绝服务攻击*：这种攻击使得合法用户无法访问目标系统。大量的明文消息或者具有大量加密元素或签名元素的消息可能耗占大量的系统资源，从而影响服务等级。这种攻击会导致系统的严重破坏。

以上所有问题说明了：若要增强开放的、松散耦合的系统的安全性，需要一个复杂的安全方法来支持分布式客户（应用）和系统，这些应用和系统可能有不同的策略和不同的安全机制。同时，由于 Web Service 和各种客户端都有高度的交互性，因此防止安全方法过多地受到干扰，并维持服务的易用性也是非常重要的。服务需要增强自身的互操作特性，并让客户了解其安全需求

和策略。因此，为确保 Web Service 网络具有合适的安全性，在调用服务的应用的安全性上下文上进行操作是一个基本需求。安全性上下文是代表应用(或用户)的服务使用的一组信息，包括应用到应用程序上的规则和策略，以及有关应用正参与的业务流程或事务的信息。当应用与服务分离时，该上下文将就被销毁。

11.1.2 对策

尽管不能消除对 Web Service 的所有威胁，在很多情况下，还是可以通过使用应用安全性来将安全威胁降低到可接受的程度。应用到网络化的(分布式的)应用和平台(诸如 J2EE)的应用安全包含六个基本需求，各方间交互的消息可表示这些需求[Pilz 2003]。这些消息包括发送者(希望访问网络应用的参与方)和接收者(应用本身)之间的任意类型的通信。应用层安全的这六个需求包括认证、授权、消息完整性、机密性、操作防御和不可抵赖性。将在 11.3 节中讨论这些对策。在该章节中，我们将讨论网络应用和分布式平台的应用层解决方案。在 11.6 节中，对于基于 Web Service 的应用，我们将描述应用层解决方案。

在下面章节中，我们将进一步分析解决网络层安全问题的技术解决方案。

11.2 网络层的安全性机制

网络层安全性指的是对流程的保护，在该流程中数据项通过网络和一个终端系统进行通信[Ford 1997]。特别地，该主题排除了终端系统(客户端和服务系统)内所发生的事件。网络层安全利用互联网协议安全(Internet Protocol Security，简称 IPSec)在网络设备或操作系统内包含了嵌入式加密功能。网络层解决方案通常设计为在自治防火墙上终止不安全的连接。采用网络层安全性解决方案的企业通常主要依赖两种技术来保护它们的网络：防火墙和漏洞评估。

11.2.1 防火墙

防火墙是在网络间放置的网络安全基础设施，在逻辑上分离和保护跨越这些网络的业务通信的私密性和完整性，并防止恶意的使用。防火墙建立在组织内部网络和互联网骨干网络之间，用于帮助定义企业和互联网之间的网络边界。

防火墙检查来往于组织的消息流量，并阻塞除了授权消息以外的对本地网络的访问。防火墙通过名字、IP 地址、应用等识别进入的流量。这个信息根据已编写入防火墙系统的访问规则来检查。因为防火墙决定什么流量允许从互联网进入企业，所以它们本质上是防御入侵者的第一道防线。防火墙是阻挡外部流量的防线，确保了企业的安全性。但是，由于防火墙将公司和它的客户以及合作伙伴隔离开来，因此很难将防火墙应用于 Web 应用。防火墙可以通过阻塞 TCP 端口来阻挡不希望的协议，但为了 Web 浏览的目，需要开放 Web 端口，即 80 端口，以及用于 SSL 的 443 端口。这样可以绕过防火墙但不影响安全性，因为防火墙继续保护更底层的通信。因此，只有本地安全策略定义的授权流量才能允许通过防火墙，而网络内外部之间的未授权的通信将被阻止。

防火墙提供的典型功能如下[Ford 1997]：

- 对于许多应用，限制流量从互联网进入内部网路，并限制不同应用的流量可以到达的内部地址。
- 认证进入流量的源点。
- 基于所使用的应用和其他相关信息，限制内部企业网络、系统与外部互联网建立连接。
- 作为安全网关，在互联网骨干网上对来往于一些其他安全网关的所有流量进行加密和/或完整性检查。这个扩充功能称为虚拟专用网(VPN)。VPN 允许组织将它们的公司网络和

互联网作为一个大的区域网络，以获得和它们分支机构、供应商、客户以及远程用户之间的安全连接。

防火墙体系结构

在连接到互联网时，任一防火墙能提供对私有网络保护的安全性级别与防火墙选择的体系结构直接相关。防火墙体系结构依赖于工作于 OSI 模型各个层次的协议所提供的信息。防火墙体系结构检查 IP 数据包所在的 OSI 层次越高，该体系结构提供的保护级别越高，因为有更多的信息可用于制定安全相关的决定。

有二大类防火墙体系结构：数据滤、电路级和应用层代理以及状态检测(stateful inspection)。我们将依次讨论。

IP 数据包过滤。数据包过滤防火墙是最老的防火墙体系结构。包过滤防火墙工作在网络层(参见图 11.1)。这种类型的防火墙应用一个过滤过程来检查单个 IP 数据包。当数据包到达时，防火墙会根据数据包的类型、源地址、目标地址以及端口信息(包含在每个 IP 数据包中)进行过滤。

有两类数据包过滤防火墙：静态的和状态的。静态包过滤防火墙近查 IP 头部和 TCP 头部中的数据，并将该信息与预先规定的包过滤(访问)规则进行比较，指定该防火墙应该拒绝还是允许数据包通过。IP 头部信息允许规定包过滤规则，即拒绝或允许数据包来往于特定 IP 地址或一组 IP 地址范围，参见图 11.1。TCP 头部信息允许规定特定服务的规则，即允许或拒绝数据包来

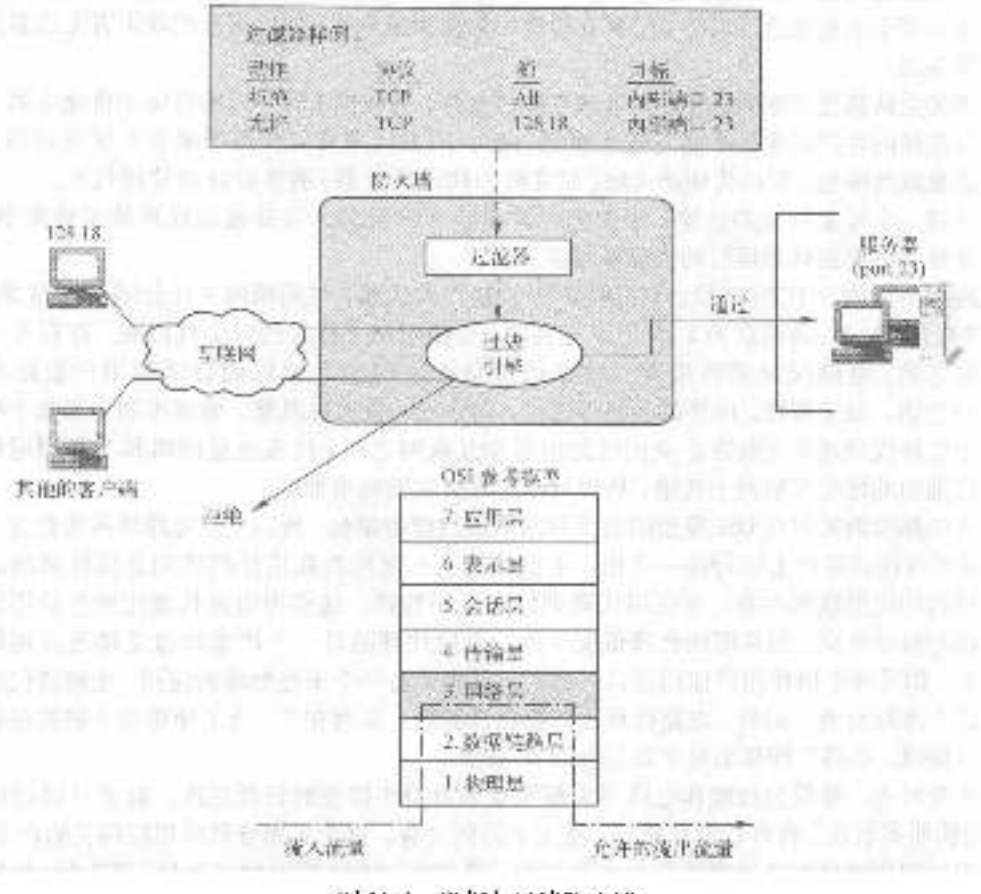


图 11.1　数据包过滤防火墙

往于和特定服务相关的端口。例如，通过静态数据包过滤，网络管理员可以编写规则，允许某些服务(诸如来自任意 IP 地址的 HTTP)查看受保护 Web 服务器上的 Web 页面，同时阻塞其他 IP 地址使用 HTTP 服务并查看 Web 页面。状态数据包过滤防火墙是静态数据包过滤器的进一步发展。这种类型的防火墙以隔离的方式检查数据包(类似它的静态部分)，它维持连接的状态信息，跟踪开放的有效连接，而无须重新处理访问规则集，并能实现复杂的策略。典型的数据包过滤能识别新连接和已有连接之间的不同。由于在核心层检查连接是新的还是已有的，因此和静态包过滤相比，可以看到状态数据包过滤防火墙的性能有了很大的提高。

出于性能方面的考虑，通常由路由器操作系统内核中的一个进程执行 IP 过滤。如果使用多个防火墙，第一个可能会对某些数据包进行标记，让后面的防火墙进行更彻底的检查，只允许“干净的”数据包进入。例如，网络管理员可以配置一个包过滤防火墙为禁止两个网络间的 FTP 流量，同时允许 HTTP 和 SMTP 流量，进一步细化站点间受保护流量上的控制粒度。

包过滤防火墙的主要优点是实现相当简单，并且和其他防火墙方法不同，它们对终端用户是透明的。虽然数据包过滤易于实现，但是很难正确地进行配置，尤其是当使用许多的规则来处理大量的应用流量和用户时。数据包过滤的一个主要缺点是其基于 IP 地址，而不是已认证的用户标识。另外数据包过滤抵抗中间人攻击的能力很弱，并对 IP 地址伪造没有抵抗能力。其他缺陷包括无法有效地识别数据包的有效载荷、无法有效地识别状态，以及易受应用层攻击。

电路级网关。电路级网关(电路代理)是数据包过滤的扩展，它执行基本的数据包过滤操作，然后添加了合适的握手验证，并验证用于建立连接的序列号的合法性。这种防火墙允许用户利用代理来和安全的系统进行通信，隐藏有价值的数据和服务器，使得潜在的攻击者无法看到这些数据和服务器。

代理接受从其他方来的连接，如果该连接被允许，则和位于另一方的目标主机建立第二个连接。发起连接的客户端永远不能直接连接到目标。因为代理可以作用于来自不同应用的、不同类型的流量或数据包，所以代理防火墙(或常称为代理服务器)通常设计为使用代理。一个代理可以仅处理一个特定类型的传输，例如 TCP 流量或 FTP 流量。需要通过代理的流量类型越多，代理服务器上需要装载和运行的代理就越多。

电路级网关位于 TCP/IP 层，使用网络 IP 连接作为代理。电路级网关在会话层(OSI 第 5 层)操作，参见图 11.2。当建立 TCP 或 UDP 连接时，电路级网关将应用安全性机制。在打开一个连接或电路之前，电路级网关将检查和验证通过防火墙的 TCP 协议的会话和用户数据报协议(UDP)的会话。基于策略，向外的连接将通过，向内的连接将被阻塞。管理控制主要基于端口地址。一个电路代理通常安装在企业网络路由器和互联网之间，代表企业网络和互联网通信。因为只有代理的地址在互联网上传输，所以可以隐藏真实的网络地址。

因为电路级网关只在 OSI 模型的会话和网络层过滤数据包，所以一旦电路级网关建立了一个连接，就可以在该连接上运行任一应用。电路级网关不能检查在信任网络和非信任网络之间传播的数据包的应用数据内容。而应用代理则检查应用数据。这使得电路代理比检查应用数据的应用代理的效率更高，但是可能会降低安全性。电路代理的另一个严重缺点是缺乏应用协议检查。例如，如果两个协作用户使用被认可的端口号来运行一个未经批准的应用，电路级代理不会检查到这个违规行为。最后，电路代理比数据包过滤慢。原因在于，为了使得每个数据包能正确地抵达目标地，电路代理重组每个数据包的 IP 头部。

应用层网关。数据包过滤和电路网关都没有涉及 OSI 模型的较低层次。如果可以同时检查 OSI 模型的所有层次，将可以设计更好、更安全的防火墙。这个思想导致应用层网关的产生。

应用专用代理检查通过网关的每个数据包，验证通过 OSI 模型的应用层(第 7 层)的数据包内容。这些代理可以过滤应用协议中的特定信息或特定命令，可以设计为复制、转发和过滤。

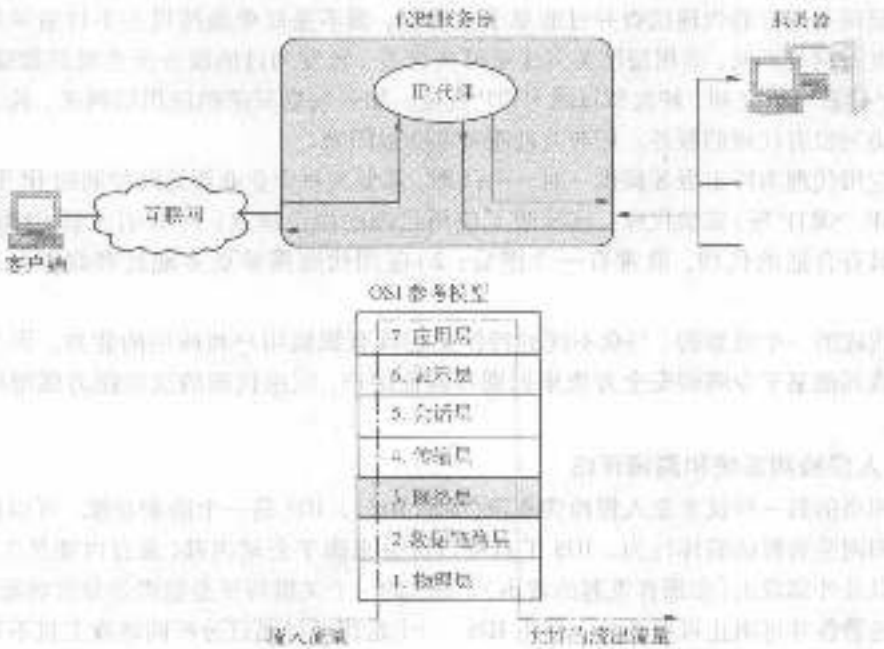


图 11.2 电路级网关

同电路级网关类似，应用层网关拦截流入和流出的数据包，运行在网关间拷贝和转发信息的代理，并充当代理服务器，避免可信任的服务器或客户端与未被信任的主机直接连接。运行应用层网关的代理(图 11.3)通常在两个重要的方面与电路级网关相区别：代理是具体的应用程序，检查整个数据包并能在 OSI 模型的应用层对流数据包。

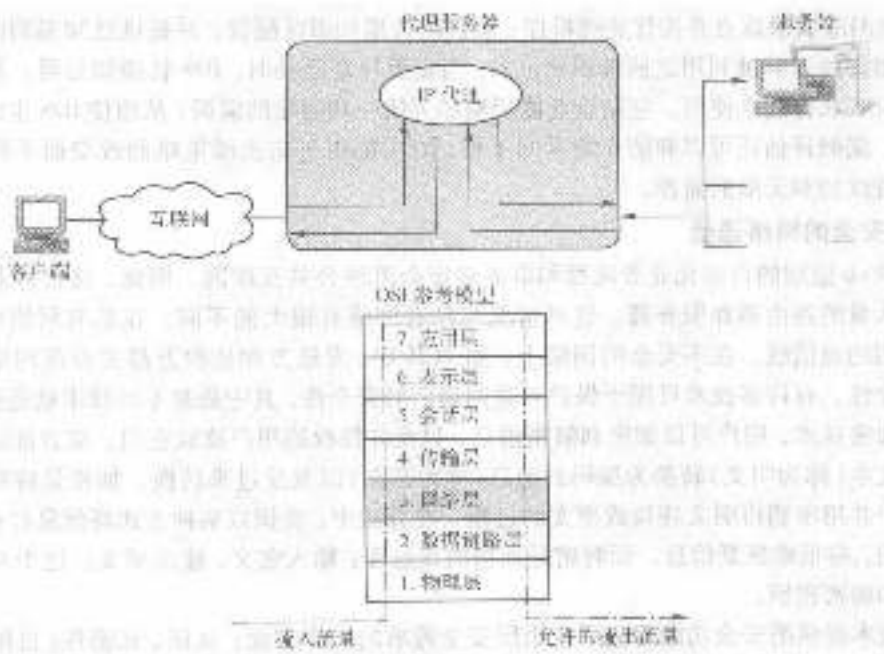


图 11.3 应用层代理服务器

应用层网关运行的代理检查并过滤单个数据包，而不是简单地拷贝并不计后果地在网关间转发。与电路网关不同，应用层网关只接受那些拷贝、转发和过滤服务所生成的数据包。例如，只有 HTTP 代理可以复制、转发和过滤 HTTP 流量。如果网络只依赖应用层网关，流入和流出数据包不能访问没有代理的服务。所有其他服务都会被阻塞。

因为应用代理为特定服务提供一对一的代理，需要为每个企业要访问控制的 IP 服务(HTTP/HTML、FTP、SMTP 等)安装代理。这导致了应用代理的两个缺点：1)在引入新 IP 服务之间后，通常不能具有合适的代理，通常有一个滞后；2)应用代理需要更多地处理数据包，导致性能下降。

应用代理的一个重要的、与众不同的特性是它具有识别用户和应用的能力。因为可以使用数字证书或其他基于令牌的安全方法来识别和认证用户，应用代理的这种能力能增加用户认证的安全性。

11.2.2 入侵检测系统和漏洞评估

保护网络的另一种技术是入侵检测系统(简称 IDS)。IDS 是一个防御系统，可以检测和响应针对计算和网络资源的破坏行为。IDS 工具可以区分来源于公司内部(来自内部员工或客户)的内部攻击以及外部攻击(由黑客发起的攻击)。IDS 的一个关键特征是能提供异常活动的视图，向管理员发送警告并可阻止可疑连接。使用 IDS，一个组织可以通过分析网络或主机不正常的数据或其他异常活动来发现攻击企图或实际的入侵。

对于正在进行的攻击，IDS 解决方案能够给出警告，但是这不适用于 B2B 应用。B2B 应用需要的是更加先发制人的方法，在网络被攻占之前就能发现可疑的攻击。漏洞评估技术可以提供这一点。

漏洞评估是一种系统化方法，用以识别漏洞，并按优先顺序排序，使企业可从“黑客视角”测试它们的网络。漏洞评估自动识别漏洞和网络的错误配置；识别反常的设备，包括无线和 VPN 接入点；检测漏洞暴露点并按优先级排序；验证防火墙和 IDS 配置；并提供已知漏洞的修补。漏洞评估在潜在的漏洞被利用之前就识别出来，当发生异常活动时，IDS 将通知公司。漏洞评估与防火墙和 IDS 联合起来使用。它使企业能识别和关闭一些明显的漏洞，从而使 IDS 生成的警告是可管理的。漏洞评估还可以和防火墙共同工作，对可能由于防火墙策略的改变而不经意引入的漏洞提供持续的和无缝的监控。

11.2.3 安全的网络通信

使用 Web 应用的自动化业务流程和事务必定会流经公共互联网。因此，这会涉及事务数据包流经的大量的路由器和服务器。这种情况与私有网络有很大的不同。在私有网络中，通信方之间有专用的通信线。在不安全的网络上，如 TCP/IP，发送方和接收方都关心在网络上交换的消息的安全性。有许多技术可用于保护互联网通信的安全性，其中最基本的技术就是消息加密。

通过加密技术，用户可以加密和解密消息，只允许授权的用户读取它们。双方都需要一个密钥将原始文本(称为明文)转换为编码的消息(称为密文)以及反过来转换。加密是将明文放入编好的算法中并用密钥将明文转换成密文的过程。在算法中，密钥以某种方式将信息打乱，如果不知道该密钥，将很难恢复信息。而解密是加密的反过程：输入密文，输出明文。这个功能涉及相应的算法和解密密钥。

密码技术提供的安全功能解决了应用层安全需求的四个方面：认证、私密性(也称隐私性)、消息完整性和不可抵赖性。当前有三种主要的密码技术可用于保护互联网通信的安全性，我们将在本节介绍。它们是对称加密(也称为秘密密钥加密)、非对称加密(公开密钥加密)、数字证

书与数字签名。

1. 对称加密

对称密钥加密，也称为共享密钥加密或秘密密钥加密。发送方和接收方使用一个密钥，参见图 11.4。对称性的是加密和解密使用相同的密钥。这个密钥也称为对称密钥或共享密钥。对称密钥加密对加密大量数据是一个有效的方法。对称密钥加密有许多算法，但是目的都一样——实现将明文变为密文的可逆转换。密文是用密钥打乱的文本。对没有密钥的任何人来说，密文都是没有意义的。解密是以相反的顺序运用密钥算法，这个过程的安全性依赖于没有未授权的人无法获取这个对称密钥。

对称加密的优点是实现的方便和快速。如果密钥只用几次，这个方法是非常有效的，因为只有有限数量的消息与新的密文比较。缺点是每对或每组用户都需要有自己的密钥，否则每个人都能读取，这导致了大量的密钥。而且如果一个密钥丢失或公开了，所有其他密钥也必须替换。从安全性需求来说，对称加密只确保私密性：只有发送者和接受者可以读取消息。

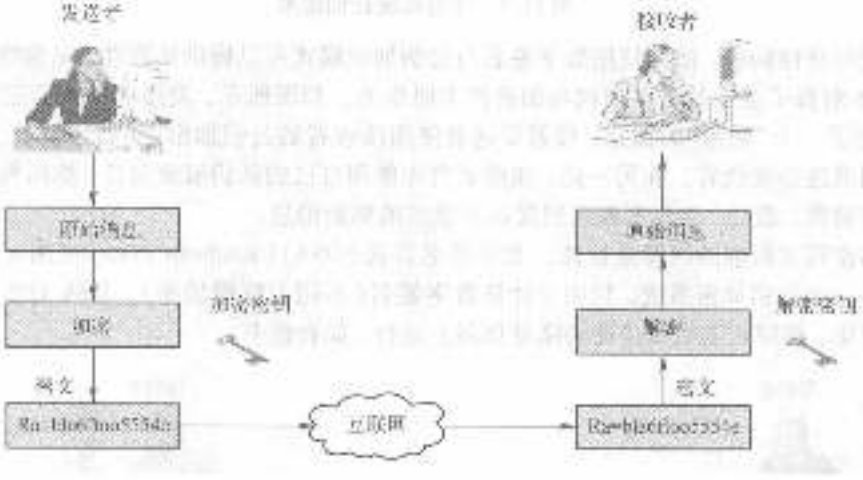


图 11.4　对称密钥加密术

2. 非对称加密

为克服对称加密的缺点，设计了一种不同的加密系统，加密和解密使用不同的密码。这种类型的加密称为非对称加密或公开密钥加密（简称公钥加密）。应用最广泛的公钥，尤其是用于在互联网上发送数据的算法是 Rivest-Shamir-Adleman（RSA）加密算法。

和对称加密相反，非对称加密（公钥密码术）的主要特点是发送者和接收者需要两个密钥，一个公钥，一个私钥。这两个密钥共享一个具体的属性。当密钥中的一个（公钥）用于执行加密，只有另一个密钥（私钥）能解密数据。这两个密钥在同一过程中创建，称为密钥对。所有的密钥在数学上与另一个相关，因此用一个密钥加密的数据可用另一个解密。这可应用于不能使用对称密钥加密的应用。

图 11.5 说明了用公钥加密的一个消息。一旦消息用公钥加密，它在互联网上发送，并且只能用它相应的私钥解密。私钥是只由接收者保存的密钥。两个密钥是不同的，用一加密消息的密钥不能用于解密相同的消息。公钥可以公开地在各参与方间传递，或在一个公共库中公开，但是相关的私钥保持私密性。因此，使用非对称加密模式可同时确保私密性和接收者的真实性，但是，这种模式不能获得可责性（也称不可抵赖性），不能确保对发送者的认证，因为任何人都可以使用接收者的公钥。

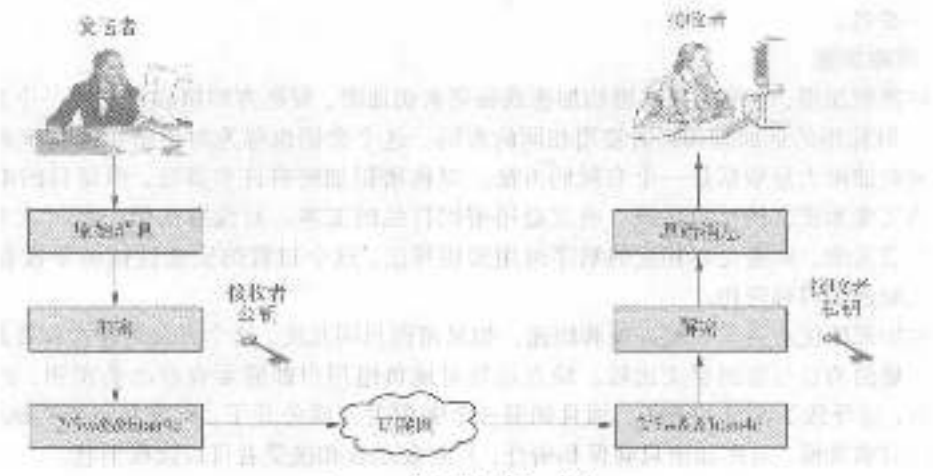


图 11.5 非对称密钥加密术

为克服可责性问题，协同使用数字签名与公钥加密模式可以提供私密性、完整性和不可抵赖性。图 11.6 解释了数字签名是如何与加密技术协作的。如图所示，发送者使用自己的私钥加密（签名）创建了一个"秘密"的消息，接着发送者使用接收者的公钥加密（签名）消息，然后该消息通过互联网发送给接收者。在另一边，接收者首先使用自己的私钥解密消息，然后再用发送者的公钥进一步解密，最后，接收者就收到发送者发送的原始消息。

非对称密码术面临的问题是性能。数字签名算法（DSA）[Kaufman 1995] 可用于解决这类问题。DSA 是一个公钥加密系统，只用于计算数字签名（不用于数据加密）。DSA 对生成签名的速度进行了优化，期望它能在低功耗的微处理器上运行，如智能卡。

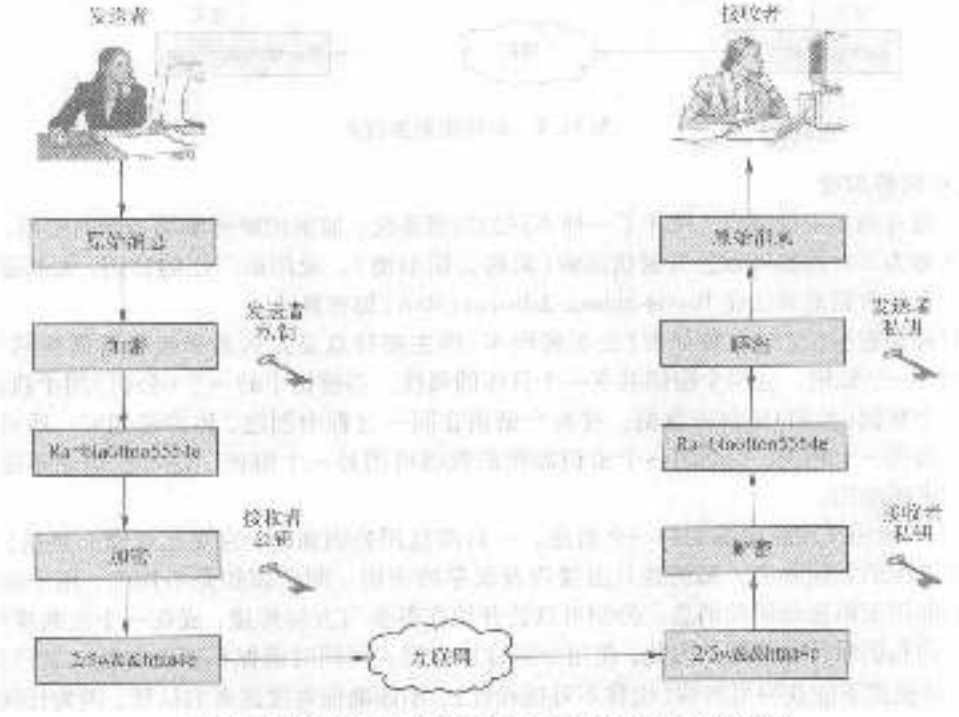


图 11.6 使用数字签名和公钥加密的非对称密钥加密术

3. 数字证书与签名

公钥加密引出了另一种加密技术：数字证书和签名。对于 B2B 的交互而言，这一加密技术绝对是必要的。

数字证书是一个文档，能唯一识别拥有该证书的成员（人或组织）、该证书有效期、签发该证书的组织、以及验证签发组织标识的数字签名。在通信链路建立期间要交换数字证书，目的是验证线路另一端的交易方确实是消息要传递的接收方，以及确认发送者的标识。数字证书由认证机构（或称认证中心，简称 CA）签发，数字证书将一个实体的标识与它的公钥绑定起来。认证机构保证密钥属于所标识的成员。密钥属主的个人具体信息和公钥都包含在一个文件中，CA 将对这一文件进行数字签名与发布。在使用数字证书进行加密时，这个数字证书将和加密消息一起发送。认证中心的签名确保了证书的真实性、完整性和不可否认性，以及公钥的正确性。为避免证书的大量传播，互联网工程任务组织（IETF）为它们开发了标准。

数字证书与数字签名相互关联。数字签名解决了公钥的认证问题。数字签名确保了消息是由数字证书中的企业或个人所发送的。使用签名者的密钥对一些数据应用密码签名算法，可生成相应的数字签名。数字签名也是一个数据块。数字签名用于验证消息源，并确保消息接收者接收的消息自签名者发送以来没人篡改过。数字签名附加在消息体上，并可用来识别发送者。接收者用发送者的公钥解密数字签名，从而可以获取消息。因此，验证机制可确保只有预期的各方才能交换敏感信息。

数字签名使用非对称加密技术，通常使用两个不同的密钥，一个用于创建数字签名或将数据转换为看起来毫无意义的形式，另一个密钥用于验证数字签名或将消息返回原始的形式。用于数字签名的密钥称为私钥，只有签名者知道私钥并可用于创建数字签名。而公钥，则通常更广为人知，可用于验证数字签名。接收者必须拥有相应的公钥以验证一个数字签名确实是签名者签发的。如果许多人需要验证签名者的数字签名，必须将公钥分发给这些人。例如，可以通过在线库或目录发布公钥，让人们很方便地获取。尽管许多人都知道特定签名者的公钥，并可以用它验证签名者的签名，但是他们不能发现签名者的私钥，无法用它来伪造数字签名。

数字签名的使用包括两个过程，一个由签名者执行，另一个由数字签名的接收者执行：

（1）数字签名创建过程即是计算一段代码的过程。根据所要签名的消息和特定私钥可生成这一代码，并且所生成的代码具有唯一性。

（2）数字签名验证指的是根据原始消息和公钥来核查数字签名的过程，从而判断该数字签名是否是使用特定公钥对应的私钥处理同一消息得来的。

在创建和验证数字签名时需要使用散列算法[Steel 2006]。基于散列算法，创建数字签名的软件可在较小的以及大小可预测的数据上进行操作，但是散列算法仍然能提供强大的证据表明这一较小的数据与原始消息内容具有关联性。一个摘要（散列）算法创建消息的消息摘要，它是一段代码，通常比原始消息要小得多，然而是唯一的。摘要算法使用（摘要）数据来计算一个散列值，称为消息摘要。消息摘要依赖于数据以及摘要算法。如果消息改变了，消息的散列值随之不同。摘要值可用于验证消息的完整性；用于确保在从发送者到接收者的过程中数据没有发生变化。在发送消息的同时，发送者一起发送消息摘要。接收到消息之后，接收者重复摘要计算。如果消息被改变了，摘要值就不同了，从而检测出消息被修改了。公钥密码技术通常与散列算法，诸如安全散列算法 1（SHA-1）[SHA-1]或消息摘要 5（Message Digest 5）（http://rfc.net/rfc1321.html）等，联合使用，从而可以保证消息的完整性。

图 11.7 显示了数字签名的创建过程。若要签署一个文档或信息的任何项，签名者首先确定文档需要签名的部分。要进行签名的文本部分称为“消息”。接着，签名者端的散列算法计算消息摘要。然后，使用签名者私钥进行数字签名，加密消息摘要。因此，对于某一特定的消息和用

于创建数字签名的私钥而言，或将生成的数字签名是唯一的。通常，数字签名附在消息一，并且与消息一起存储和传输，但是只要数字签名能维护与消息的对应关系，它也可以作为独立的数据元素进行发送或存储。因为数字签名是与它的消息一一对应的，如果它和相应的消息失去了对应关系，则数字签名就变得毫无作用了。

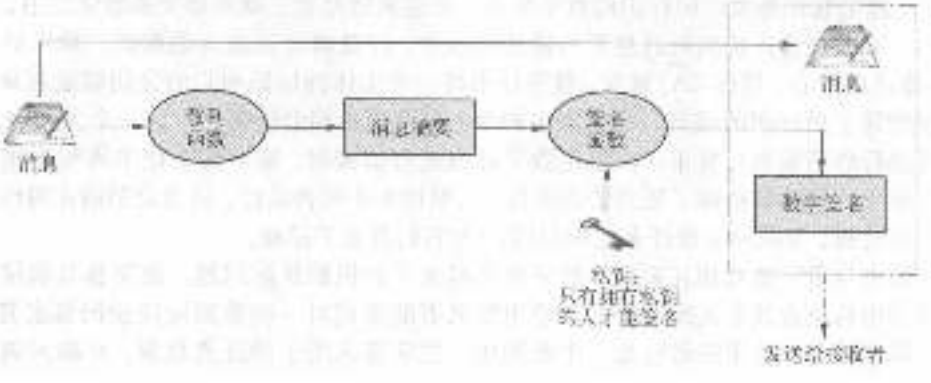


图 11.7 数字签名的创建

图 11.8 描述了数字签名的验证过程。如图所示，验证过程即是用创建数字签名的散列算法计算(实现统一好的)原始消息的消息摘要。消息摘要用发送者私钥加密。接着，消息与消息摘要一起发送给接收者。然后，接收者使用相同的散列算法计算发送者明文消息的消息摘要。然后，接收者验查新计算出来的消息摘要是否与前面导出的消息摘要相匹配。如果使用了签名者的私钥，并且消息摘要是相同的，则证实了数字签名和原始消息。因此数字签名的验证即保证了数字签名没有被篡改，也保证了原始消息没有被篡改。

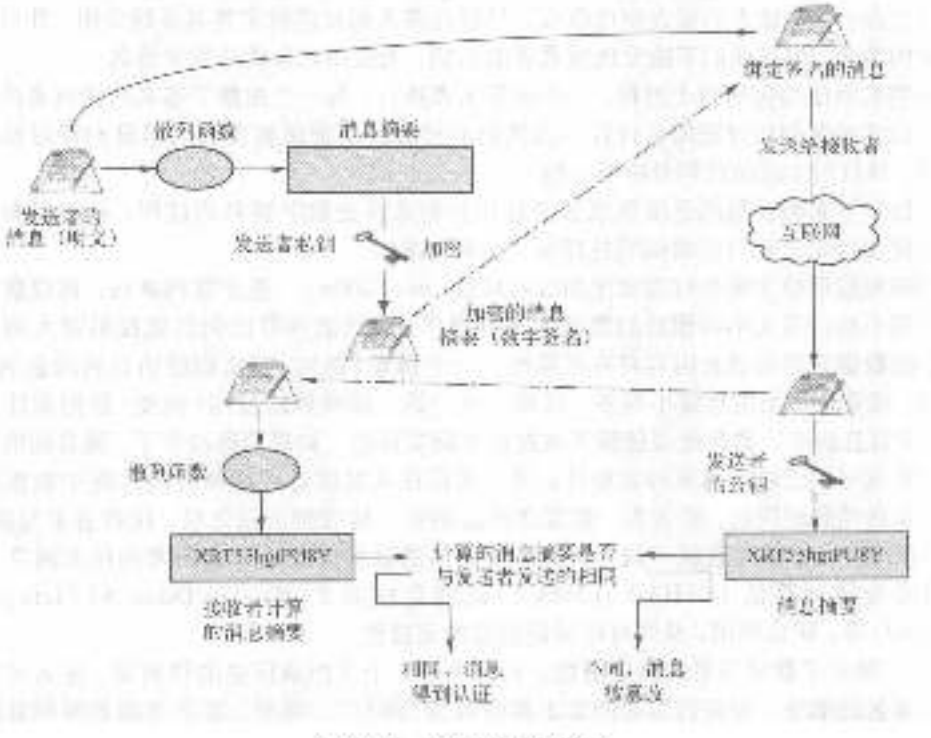


图 11.8 数字签名的验证

11.3　应用层的安全性机制

通过网络层安全方法有时不能满足一些具有复杂的安全性需求的应用。术语应用层安全性表示了那些作为特定应用的一部分的安全措施，这些安全措施独立于任何网络层的安全方法[Ford 1997]。

当考虑应用层安全需求时，如我们在 11.1.2 节中简要讨论的认证、授权、消息完整性、私密性和不可抵赖性，用于发送者和接收者之间的任一种通信的机制和技术需要安全可靠，这是最根本的需求。下面我们描述分布式计算环境中的应用层安全机制，如 J2EE。在 11.4 节和 11.5 节中，我们将要讨论与 Web-service 相关的倡议和解决方案。

11.3.1　认证

在分布式计算环境中，客户和服务提供者分别代表具体的用户和系统，认证是一种客户和服务提供者相互之间证明自己身份的一种机制[Monzillo 2002]、[Singh 2004]。一个客户通常具有一个标识符，服务提供者验证客户的身份标识。当这种证明是双向时，就称为互相认证。认证将证实所声称的身份标识，并证明参与者确实对应了这些标识。例如，认证可以验证公钥证书和数字签名信封所提供的实体的身份标识。在分布式环境和电子商务应用中有许多认证方法，从简单的用户名和口令，到比较复杂的方法，如令牌和数字证书等。

在分布式环境中，认证过程通常包含两个阶段[Monzillo 2002]。首先，执行服务独立的(即与具体的服务不相关的)认证，从而建立认证上下文。认证上下文封装身份标识，并能构造认证者(身份标识的证明)。接着，可使用认证上下文认证其他(被调用或调用)实体。认证需要控制对认证上下文的访问，从而能作为相关的身份标识进行认证。认证上下文的访问控制策略和机制包括：

- 一旦客户程序发起认证，该客户程序启动的进程将继承对认证上下文的访问。
- 当认证一个组件时，其他相关的组件或被信任的组件(例如同一个应用的其他部分)也可访问认证上下文。

在分布式环境中，当一个客户程序访问一组分布式组件或资源时，如 Java Server Page(JSP)、Enterprise Java Bean(EJB)、数据库等，客户必须提供自身的身份标识。接着，容器组件判断该客户是否满足由授权规则指定的访问准则。容器是运行应用程序的运行时环境，提供负载和性能管理、资源管理、安全管理、事务管理、部署、配置和管理功能(参见第 8 章)。容器是 J2EE 应用服务器的同义词。J2EE 应用在容器内运行，容器对应用具有具体的职责，如它介入所有方法调用并提供标准化的环境，为底层组件提供特定的服务。在许多情况下，当处理客户对一个组件的请求时，可能需要该组件发出访问其他组件和资源的一系列调用，并使用认证上下文。因此，在客户端调用组件时，容器不仅强制认证并建立一个标识，而且在组件向其他组件和资源发起一系列调用时，容器还将处理认证。在分布式平台中，初始调用的认证建立的客户身份标识可以沿着调用链传播。

1. 保护域

分布式平台允许实体按专门的域进行分组，这些专门的域称为保护域。在保护域中，实体可以相互通信而无须认证[Monzillo 2002]、[Singh 2004]。保护域是一组实体的逻辑边界，这组实体假定是互相信任的、或确知是互相信任的。当一个组件与同一保护域内其他组件交互时，对于调用相关的身份标识没有任何约束。在一个保护域中，只有当实体要跨越保护域边界时才需要认证。保护域内的交互不需要认证。

在分布式环境(诸如 J2EE)中，为确保未证实或未认证的实体不能穿越保护域边界，容器在

外部调用和内部组件之间建立了一个认证边界。通常，提供双向认证功能并实施已部署应用的保护域边界是容器的任务。

在调用进入保护域之前，容器确保已经认证了调用的身份标识。对于对内的调用，组件可以获得调用者身份标识，该身份标识将表现为身份证明，这也是容器的职责。X.509 证书和 Kerberos 服务票据是在计算环境中使用身份证明的一些例子。X.509 证书是公钥/密钥(非对称)密钥对的公钥部分的数字容器。当与公钥相关的身份标识得到认证中心的确认后，认证中心将签署这个数字证书。对于向外的调用，容器负责确定调用组件的标识。

2. Web 资源保护

在分布式环境如 J2EE 中，Web 层提供了客户的应用业务逻辑及 Web 连接性。Web 层处理所有与 Web 客户端的应用通信，根据输入请求调用业务逻辑及传输数据，并提供对企业资源的访问。客户端可用的 Web 层资源可能是被保护的，也可能是未被保护的。保护的资源是关键资源，如后端系统。授权规则对关键资源与未被保护的资源进行了区分。对一些非匿名的身份标识，限制它们对被保护资源的访问。为访问一个保护资源，客户端必须提交身份证明，从而可根据资源授权策略评估客户端的身份标识。

通过 Web 层认证，开发者规定了授权约束，从而指明了那些需要保护的 Web 资源，如 HTML 文档、Web 组件、图像文件、归档文件等。当客户试图访问一个受保护的 Web 层资源时，将激活被访问的组件或资源的认证机制。例如在 J2EE 中，Web 容器支持三种认证机制[Singh 2004]：HTTP 基本认证、基于表单的认证、HTTPS 相互认证。

通过 HHTP 基本认证，Web 服务器使用来自 Web 客户端的用户名和口令认证一个参与者。基于表单的认证让开发者可以定制 HTTP 浏览器显示的认证用户接口。与 HTTP 基本认证类似，基于表单的认证是相对弱的认证机制，因为用户对话是以明文发送的，并且目标服务器没有认证[Monzillo 2002]。最后，通过 HTTPS 互相认证，客户端和服务器都使用数字证书来建立标识，并在由 SSL 保护的信道上进行认证。

对于安全性策略域边界内的应用，分布式环境提供了单点登录。单点登录允许用户使用单个口令就能访问基于网络的、所有可用的资源。在计算机网络中，通过使用票据可实现这一技术。在认证时，用户获得一个票据，票据指定是它的身份标识以及其他与安全相关的信息。当访问的资源在其他服务器而不被认证的服务器上时，票据用来表明用户的身份标识。通过这种方式，用户不需要在每次使用服务器时都认证多次。在 J2EE 应用服务器中，当客户在一个应用上获得认证时就提供了单点登录，此外还能自动认证客户端身份标识映射的其他应用。

11.3.2 授权

分布式环境的授权机制只允许认证的调用者身份标识访问资源，如主机、文件、Web 页面、组件以及数据库条目等。可基于角色、组和权限，将客户分成不同的集合。典型的授权策略允许不同的客户集合访问不同的资源。角色代表资格、授权、责任或特定的职责分配。而组则根据组织关系而形成，如公司、部门、实验室等。授权策略还可能基于全局上下文(如时间)、事务性上下文(如每天的回退不允许超过一定的数量)、或者数据值来限制访问。因为分布式平台主要围绕权限许可，权限许可规定谁能执行什么功能，所以在执行授权策略之前需要进行认证以及确认身份标识。

一旦提供的数字证书或其他证书认证了用户的身份标识，就必须决定用户的访问权限。授权意味着限制认证方能在网络环境中执行的动作或操作。限制对敏感信息的访问(不管是雇员还是交易合作伙伴)的传统解决方法是在资源(诸如 Web 页面或组件)上施加访问控制规则，然后评估请求的访问类型，从而判断是否允许请求者访问该资源。访问控制规则可以以声明的形式

定义任意安全策略，而无须开发人员在应用中编写策略。从本质上来说，访问控制指的是确保资源不以未经授权的方式使用。访问控制可以应用在分布式环境中，保护特定业务流程或应用上下文中的资源不被未经授权的操作调用。控制访问权限可以指派为没有(none)、只读(read-only)、增加(add)、编辑(edit)或完全(full)等。

为分布式平台定义访问控制规则的最常用方法是基于权限。权限指的是谁能做什么。可以通过声明或编程方式来规定权限。

通过使用声明授权，可定义称为(安全)角色的逻辑特权。这些与组件相关联的逻辑特权指定了主体(一个实体，可以是某一安全域中拥有身份标识的人或计算机)所需的特权，并根据身份标识(通过认证确认)授予访问组件的权限。权限指出根据功能性角色以及/或数据敏感性执行某个操作的能力，例如在一个组件(资源)上"创建一个订单"或"批准一个订单"。以这种方式编写的安全许可是静态的、粗粒度的，并且无法很好地表达。基于调用者安全性属性值，可分配调用者的逻辑特权。在操作环境中，当一个安全角色分配给一个安全组时，若任何调用者的属性表明该调用者属于该组，角色所具有的特权就分配给该调用者。

除了通过声明规定安全性角色，在许多情况下需要将附加的应用逻辑与访问控制决策相关联。在这样的情况下，编程授权用于将应用逻辑与资源的状态、组件调用的参数或其他一些相关信息关联起来。编程授权需要访问应用的源代码以插入合适的检查。编程安全支持的授权粒度比声明安全所支持的授权粒度更细，但是限制了组件的可重用性。当组件间的编程安全模型不一致时，将多个使用编程安全的组件组装成一个应用是困难的，甚至是无法实现的。当安全策略改变时，又显露了编程安全的另一个缺点：必须重新检查每个组件，验证并可能更新安全授权。

在分布式平台提供的机制中，可基于身份标识属性(例如调用代码的位置和签署者、调用代码的用户的身份标识)来控制对 Web 资源的访问。在所有情况中，被调用的组件都要用到一个身份证明(对于保护资源是至关重要的)[Monzillo 2002]、[Singh 2004]。例如，J2EE 中的基于容器的授权机制需要容器作为它所包含的组件以及调用者之间的授权边界。授权边界位于容器的认证边界内，标识了发起当前请求的用户。因此，在成功地进行认证的基础上，才能考虑授权。对于向内的调用，容器将调用者身份证明中的安全属性与目标组件的访问控制规则进行比较。如果满足规则，则允许调用，否则拒绝调用。

11.3.3　完整性与机密性

在分布式计算系统中，大量的信息以消息的形式通过网络发送。消息内容会遭受三种主要的攻击。消息可能会被拦截并修改，目的是改变接收方所收到的消息。消息可能会被截获，攻击者可能重复(一次或多次)使用这些消息。偷听者监听消息，从而可能获取以其他方式无法获得的信息。使用完整性和私密性机制可最大程度地降低这类攻击所造成的影响。

消息(数据)完整性包括两个要求。首先，接收到的数据必须与发送的数据相同。换句话说，数据完整性系统必须能确保消息在传送过程中没有被修改，不管是有意的还是由于出错的原因。消息完整性的第二个要求是在未来的任何时候，都可以证明同一文档的不同副本实际上是相同的。

数字签名可用于验证消息是否被篡改过。服务请求者可用发送者私钥对一个文档进行签名，并将它与消息的有效载荷一起发送。然后服务提供者可以用发送者的公钥来验证签名，查看文档是否被篡改过。从而在相互通信时，Web Service 应用可以确保数据完整性。例如，XML 签名标准(参见 11.5.1 节)提供了对 XML 文档的一些部分进行签名的方法，提供了跨多个系统的端到端的数据完整性。

消息完整性确保传送的信息没有被修改。安全事务是确保消息在传输时没有被修改的典型

方式。这通常称为通信完整性，通常通过散列算法和数字签名的摘要代码来实现。

安全事务还应当确保私密性。私密性指的是确保消息和数据只能由被授权者看到。若能确保各方间的连接不被拦截，可实现私密性。例如，在未被信任网络发送数据时，采用加密技术。使用 HTTPS 的标准 SSL 加密保证服务请求者和服务提供者之间点对点数据的私密。但是，在许多情况下，服务提供者可能不是消息的最终目的地。服务提供者可能作为服务请求者，向多个服务发送信息。对于这类情况，XML 加密标准可结合 Web Service 使用，允许加密部分消息，使头部和其他信息是明文，同时加密敏感的有效载荷。从而敏感信息可以保持加密，直到抵达最终目的地，实现真正的、端到端的数据私密性。

11.3.4 不可抵赖性

不可抵赖性对于互联网上的事务是至关重要的。不可抵赖性是通过密码方法取得的一个特性，可以避免个体或实体否认执行了和数据相关的特定动作。当事务执行时，经常需要能证明特定动作发生了，并且通过有效的身份证明提交事务。这可以防止交易方宣称事务从没发生过。使用数字证书(如 PKI X.509 或 Kerberos 票据)的数字签名是提供不可抵赖性的关键要素。基于非对称密码术生成的数字签名具有不可抵赖性。创建签名的个人或组织不能否认他们已做的事。不可抵赖性通常将修改检测与数字签名结合在一起。当需要第三方不可抵赖性时，数字收据提供独立的证明，表明特定的事务已发生。不可抵赖性由密码收据组成，使消息的作者不能抵赖发送了消息。这些任务正好是合约形成和执行的前提。基于数据信息库，跟踪数字签名的消息提供了一个审计线索以确保不可抵赖性。接收者将数字签名连同消息一起保存在仓库中，以供以后万一发生争议时能够援引。

11.3.5 审计

审计记录一些事件。例如失败的登录企图和被拒绝的使用资源的请求，这可能预示着可能存在违反企业安全性的企图。审计的价值不只是判断安全机制是否限制对系统的访问。当安全性被破坏时，分析安全性相关的事件以判断谁被允许访问关键数据。知道谁与系统交互了，就可以判断破坏安全性的责任。

通常，部署者或系统管理员可仔细检查为平台所建立的安全性约束，将审计行为同每个安全性约束关联起来，从而可以对它们进行分析和审计。对审计配置的所有修改(由部署或后来的管理导致)以及平台执行的约束，需要谨慎地进行审计。必须保护审计记录，使攻击者不能通过擦除犯罪记录或修改内容来逃脱他们行为的责任。

11.3.6 应用层安全性协议

安全通信使用许多认证和加密协议来确保互联网上的安全会话，而这些认证和加密协议则应用了加密和认证技术。传统上，安全套接字层(SSL)与事实上的传输层安全(TLS)以及互联网协议安全(IPSec)是互联网上保护分布式应用内容的一些常用方法。在本节中，我们将简要讨论开放网络中处理认证、完整性和私密性问题的一些常用协议。

1. 安全套接字层(SSL)

根据应用层集成的安全性的要求，各个公司的业务应用应提供在客户和应用服务器之间进行安全通信的功能。主要用于 Web 浏览器的 SSL，是最常用的安全协议。SSL 是 Web 协议的一个开放的标准，在 TCP/IP 连接上提供服务器认证、数据加密和消息完整性。SSL 广泛应用于互联网业务，几乎所有的热门的浏览器和 Web 服务器都支持 SSL。

SSL 可充当作一个易于部署的专用安全协议，为多个应用提供全面的安全性，确保客户和应用服务器之间的通信保持私密，并允许客户识别服务器，反之亦然。这种形式的安全性在两个交

互相服务器之间建立一个安全的管道，如图 11.9 所示。在安全管道创建时，将进行认证，而私密性和完整性机制只在消息位于安全管道时才应用。

SSL 为客户-服务器会话提供了大量的安全性服务，包括服务器和客户认证、数据完整性和数据私密性。支持 SSL 的客户端和支持 SSL 的服务器使用数字证书和互相认对方的身份标识。SSL 服务器认证使客户可以确认任意有问题的事务所涉及的服务器的身份标识。在 SSL 中，这通过使用公钥加密技术来实现，验证服务器的证书是有效的并且是未信任的认证中心颁发的。客户认证允许服务器确认客户的标识，方式同服务器认证一样。支持 SSL 的服务器软件可以检查客户证书和公钥是有效的，并且是由服务器信任的认证中心列表中的某一认证中心签发的。SSL 协议加密和解密客户和服务器间传送所有信息的过程。通过加密的 SSL 连接传送的信息保持私密性和不被篡改，确保接收的数据没有被修改并且不能被其他人看到。

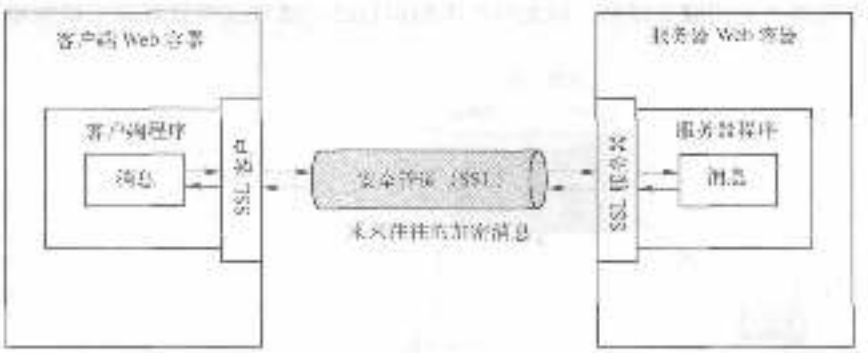


图 11.9　传输安全性和 SSL

因为 SSL 不支持证书验证。目前提供证书扩展来实现证书的验证。此外，SSL 扩展还提供其他一些特性，诸如监控，日志以及访问控制授权，而传统的 SSL 技术不支持这些特征。

2. 互联网协议安全性(IPSec)

IPSec 是另一个用于传输安全的网络层标准，在分布式和电子商务应用中起着重要作用。与 SSL/TLS 类似，IPSec 还提供安全会话，提供主机认证、数据完整性和数据私密性。但是，这些都是点到点的技术。他们创建了数据通过的安全隧道[Wysopa 2003]。例如，SSL 是两个服务器间的安全性的很好的解决方案，但是它不能用于消息要经过多于一个服务器的情况。在这个情况下，接收方必须请求发送方的身份证明，系统的可扩展性受到影响。

3. Kerberos

Kerberos 为计算网络提供一些认证和安全工具，包括单点登录以及公钥密码技术的使用。Kerberos 的目标是在一个不安全的分布式环境中提供认证。Kerberos 是一个第三方认证协议，处理两种安全对象：票据和会话密钥。称为票据的一个认证令牌(一些用于认证或授权的信息，可添加到 SOAP 头部)由 Kerberos 票据授予服务签发，提供给特定的服务器，验证该客户最近由 Kerberos 认证过。票据包括一个终止时间以及一个新生成的会话密钥，由客户端和服务器使用。会话密钥是由 Kerberos 随机生成的密钥，分发给客户，当与特定服务器通信时使用。客户正程必须为每个使用的服务器提供一个票据和会话密钥。

在 Web Service 安全性应用中，Kerberos 为 Kerberos(第三方)服务器提供一个方法来独立地验证两方的信任关系(通过一个认证服务)，然后为这两方授予共享密钥，作为安全交互的基础[Hall-Gailey 2004]。Kerberos 实现执行集中式密钥管理。

图 11.10 说明了用于客户访问一个 Web Service 的 Kerberos 票据授予机制。在这个场景中，

密钥分发中心(KDC)维护委托者的身份证明。KDC 有两个功能：它首先提供一个认证服务，从客户进程中接受请求者的身份证明(通常为登录用户名和密码)，然后向请求者发回称为票据授予票据(Ticket-Granting Ticket，简称 TGT)的证书。TGT 包含一个在会话期间有效的临时会话密钥，这样可以避免使用永久的证书。TGT 是加密的，因此只有拥有正确口令的合法委托人才能解密它，并在将来使用它。当客户希望使用 Kerberos 访问一个服务器应用(在本例中由特定提供者提供)提供一个特殊的 Web Service)时，客户将它的 TGT 提供给 KDC 的票据－授予服务(Ticket-Granting Service，简称 TGS)部分。TGS 然后返回服务(或会话)票据，该票据包含一个秘密会话密钥、一个客户标识以及 TGT 失效时间。客户使用会话票据建立与有问题的 Web service 应用之间安全 TCP/IP 通信。会话票据通过用提供者的私钥加密请求者的信息来加以保护，不暴露在网络上。因此，只有提供者可以通过解密请求者标识来认证请求者。只要 TGT 是有效的，请求者就可以从不同的提供者获得不同的服务票据，而无须再次标识自己。通过这种方式就可以实现单点登录。

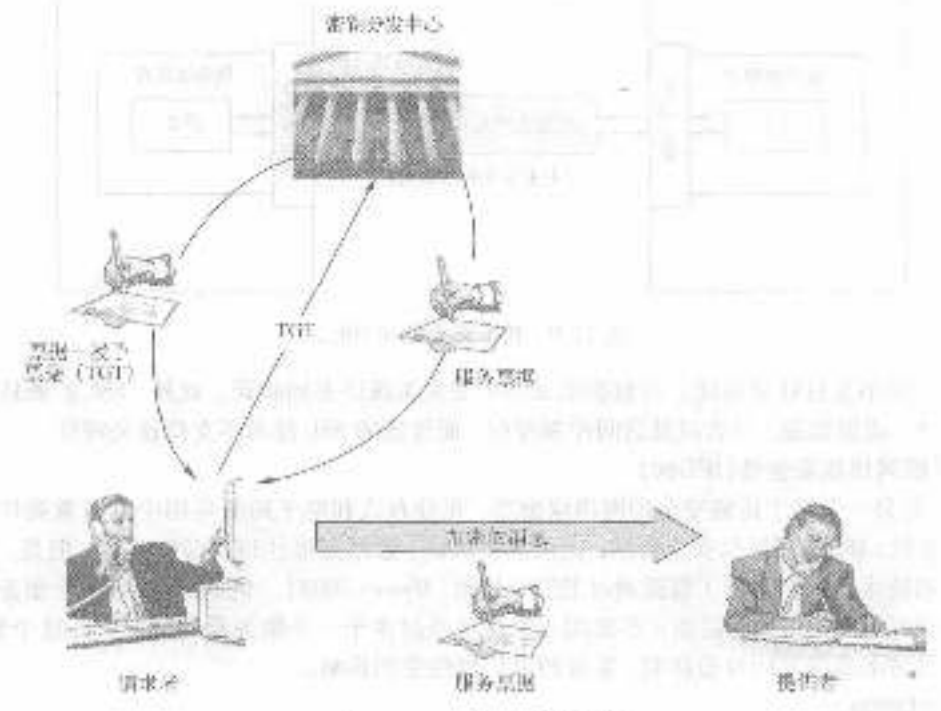


图 11.10 Kerberos 票据分发

11.3.7 安全性基础架构

为保护信息资产，企业希望提供"把关(Gatekeeping)"功能，如数据保护和网络隔离。企业还希望提供一些便捷功能，如将企业数据展示给外部应用，为扩展合作连接用户以及实现在线事务和通信。安全性基础架构是一个分布式的基础架构。基于安全性基础架构，应用可以安全地通信和交换数据。

1. 公钥基础设施

在一个分布式并涉及多方面的环境中，提供应用和网络安全性的基础是公钥基础设施(PKI)[VeriSign 2003a]。PKI 是构建具体应用和网络安全组件的基石。那些有着最高安全性级别需求的应用(诸如在线银行和贸易、基于 Web Service 的业务流程的自动化、数字形式的签名、企业即

时消息以及电子商务等)通常需要采用 PKI。此外，它还保护防火墙、VPN、目录和企业应用。如果事务被修改或披露时有伪造、法律风险时，或者在确认个体或业务实体的身份标识非常重要的情况下，PKI 就显得至关重要了。

PKI 指的是技术、基础架构和一些具体实务。这些具体实务支持数字证书认证中心的实现和操作。认证中心将验证和认证互联网事务相关参与方合法性。总的来说，PKI 是一个有关策略、服务器和技术的基础架构，为企业计算中的重要的安全问题提供密码技术解决方案。PKI 能帮助创建和管理自动业务流程和电子业务应用所需的非对称密钥或公钥/密钥。PKI 使用一个私钥和一个公钥来加、解密私密信息，以及生成和验证数字签名。PKI 的主要功能是将公钥正确地和可靠地分发给需要它们的用户和应用。

PKI 能提供的具体安全功能包括私密性、完整性、不可抵赖性以及认证，同时还能确保军事级别的物理安全。在企业内部，PKI 很容易与各种功能，如市场、销售、库存、财务管理等集成在一起，以及与各种内部和外部企业应用，甚至包括遗留系统，都可集成在一起。通过这种方式，PKI 使得企业很容易地创建合作伙伴、客户和供应商间的信任环境。

实际上，PKI 指的是数字证书、认证中心和其他注册机构的系统，用于验证和认证电子事务中涉及的各参与方的有效性。公钥证书是数字签署的声明，将公钥的值与持有相应私钥的主体(人、设备或服务)的标识相关联。证书的签发者称为认证中心。证书签发的实体是证书的主体。通过签署证书，认证中心证明与证书中公钥相关的私钥是属于证书中主体的。认证中心创建并签署数字证书，维护证书调用列表，使证书和调用列表可用，并为管理员提供管理证书的接口。可为各种功能签发证书，包括 Web 用户认证、Web 服务器认证、使用安全/多目标互联网邮件扩展(S/MIME)的安全电子邮件、IPSec、SS//TLS 以及代码签名。

签发一个证书需要验证用户的标识，通常由注册机构实现。证书和注册机构是两大主要 PKI 组件，提供建立、维护和保护信任关系所必须的能力[Schiosser 1999]。注册机构评估身份证明和相关证据，证实请求证书的组织确实是它所宣称的组织。来自注册机构的数字签名消息需要发送到认证中心以验证签署者。一个认证中心可能运作几个注册机构。

图 11.11 阐述了 PKI 运作的一种方式。请求者首先生成一个公钥/密钥对。然后将公钥放入用请求者私钥签署的证书请求中。这个自我签署的证书发送到注册机构，验证签署者的身份标识。一旦请求者的标识被验证，该证书就由注册机构确认并发送给认证中心。认证中心验证证书请求上的注册机构的签名。此外它还验证该注册机构是否有权利发起该请求。接着，认证中心使用证书请求中的公钥来创建一个公钥证书并签署该证书。认证中心也可以接受该证书，并将它放入一个目录。这样，任何想与请求者通信或希望验证请求者签名的参与者都可以获取请求者的公钥。最后，该证书返回给请求者，由请求者转发给接收者。接收者从请求者接收到进行数字签名的信息，并需要使用 PKI，特别是公钥证书来验证请求者的签名。

一个最常引用的 PKI 标准是 PKI X.509(PKIX)标准。PKI X.509 标准基于 X.509 证书格式以及一些 IETF 协议(诸如 SSL、S/MIME 以及 IPSec)定义了公钥证书的内容。PKI X.509 提供了不同的认证中心生成的数字证书间的互操作性。

2. 目录服务

PKI 应用的证书将在一个公开目录中发布，允许任何人使用证书中给出的接收者公钥来向某个人发送加密消息。通常由作为认证中心的第三方来管理目录。

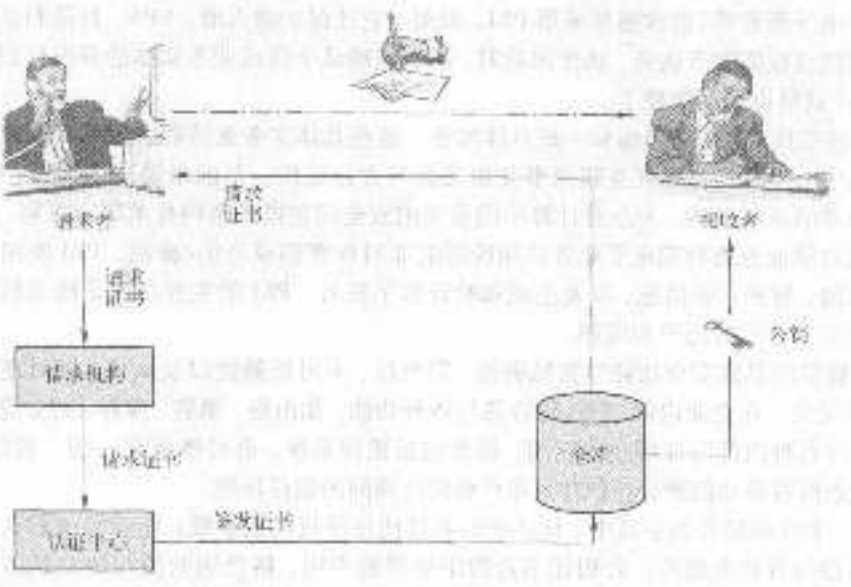


图 11.11 PKI

针对应用层安全性的目录服务提供了实现安全策略的能力。同样地，可使用目录来存储授权信息，如组成员身份(Group Membership)以及访问权限。目录可用于支持单点登录。并且目录通常提供网络可用资源和计算基础设施的位置信息和其他细节信息。在部署 PKI 的系统中，目录可用于分发证书。对于应用来说，在加密消息发送之前必须获取终端用户的证书以及证书状态信息，如证书吊销列表。当环境需要移动性，即用户每天不使用相同的机器时，目录还可以存储私钥。

在网络目录中访问数据的最常用协议是轻量级目录访问协议(LDAP)。LDAP 使用 X.500 数据模型。X.500 数据模型具有复杂的目录体系结构，该模型在 ISO 和其他国际标准组织的支持下设计的。在 X.500 模型中，数据的基本单位是目录表项，它一个或多个属性/值对组成。LDAP 位于 TCP/IP 层之上，因此实现和部署相当简单。LDAP 分布式体系结构支持可扩展的目录服务，具备服务器复制能力，从而确保目录数据在需要时是可用的。

出于安全考虑，LDAP 既支持使用可分辨的名称和口令的基本的客户端认证，又支持 SSL 服务，在客户和服务器间进行互相认证，确保查询和响应的私密性和完整性。若要启用 SSL，需要一个服务器证书。通过恰当地配置 LDAP 目录服务器，还可以支持基于证书的客户端认证，其中证书由 PKI 提供。这样，仅已认证的对象才能访问目录。

11.4 安全性布局

在非军事区(DMZ)生产环境中部署应用时(例如基于 J2EE 的应用)，分布式环境中的安全体系结构经常面临设计选项和策略。这里的非军事区(DMZ)指的是两组防火墙之间的逻辑分区。这里假定 DMZ 内的基础设施需要被直接访问，因此更容易被攻击[Steel 2006]。DMZ 中的第二个防火墙用于抵御已通过获取对应用服务器和后端应用访问而危及服务器安全的攻击者。

安全性拓扑定义 DMZ 环境中分布式应用开发的安全需求，涉及体系结构方面的性能如可用性、可扩展性、可靠性、可管理性和性能。这类拓扑有两种[Steel 2006]：水平伸缩和垂直伸缩的

安全体系结构。

图 11.12 显示一个 J2EE 应用的水平伸缩的安全体系结构。该结构划分为 Web 层(采用 JSP/Servlet)、应用层(采用 EJB 组件)以及后端资源。使用 Web 和应用服务器的多个实例可以实现水平可伸缩性。Web 层和应用层通过防火墙分隔。因为 Web 和应用服务器之间的流量需流经防火墙，这可以增强安全性。对于 Web 层和后端资源层之间流量相对较高的应用，比较适用这种体系结构。

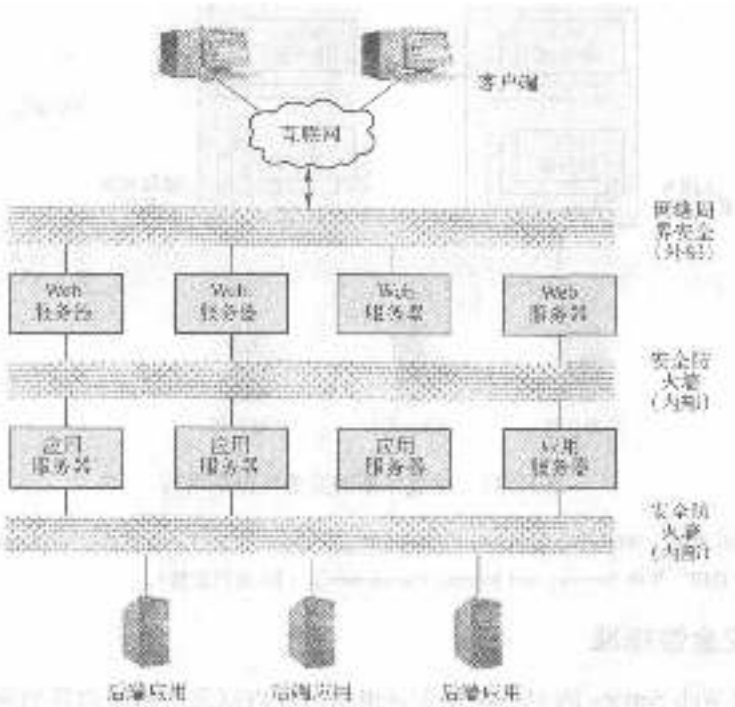


图 11.12　水平伸缩的安全性体系结构

引自：Prentice Hall 出版社 2006 年出版的、由 C. Steel、R. Nagappan 和 R. Lai 所著的《Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management》一书(允许复制)。

为将 Web 从后端资源应用中解耦，可为 Web 服务器配置一个反向代理。反向代理在网络的流入端接收客户的 HTTP 请求，并在应用服务器端打开一个套接字连接，从而执行业务应用处理[Steel 2006]。这种配置适用于安全需求不那么严的环境。

图 11.13 显示了 J2EE 应用垂直伸缩的安全性体系结构。该结构划分为 Web 层(采用 JSP/Servlet)、应用层(采用 EJB 组件)以及后端资源。添加计算能力如处理器、内存等可实现垂直可伸缩性。在服务器基础架构发生故障时，这种结构会导致整个系统失效。为避免出现这种情况，可引入高可用性集群。在发生故障时，若采用了集群技术，Web 和应用服务器基础架构将能迅速得到恢复。

在垂直伸缩的安全性体系结构中，若要将 Web 从后端资源应用解耦出来，则可在 Web 服务器上配置一个反向代理，其操作方式类似于水平伸缩的安全性体系结构中的情况。

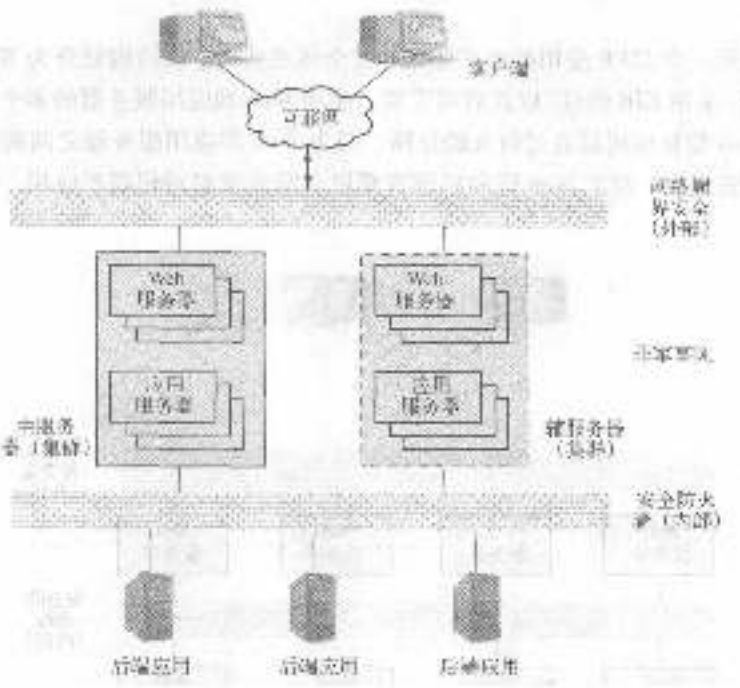


图 11.13 垂直伸缩的安全性体系结构

引自：Frontier Hall 出版社2006年出版的，由 C. Steel、R. Nagappan 和 R. Lai 所著的《Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management》一书（允许复制）。

11.5 XML 安全性标准

基于 XML 和 Web Service 的 SOA 将能促进组织边界内以及跨组织边界的业务集成。但是这个好处也是有代价的：安全系统本身也必须集成。没有这个安全集成，安全解决方案仍是在各个项目层，没有集中的方式来配置，监控、分析和控制集成数据流。对于企业集成的成败与否，跨企业边界实施、管理和监控安全性策略变得越来越重要。

在集成解决方案中，Web Service 技术使用互联网协议上的基于 XML 的消息来与其他应用进行交互，因此我们将首先着重讨论集成企业中的 XML 安全性解决方案。

XML Trust Services 是由行业协作开发的，它是针对应用开发者的一组开放的 XML 规范。通过 XML Trust Services，可更容易地将大量的 XML 安全性服务集成到基于 Web 业务应用中。XML Trust Services 的主要技术包括[VeriSign 2003b]：用于加密认证数据的 XML Signature、用于加密数据的 XML Encryption，用于管理密钥注册和密钥认证的 XML 密钥管理规范（XKMS），用于指定权力和标识的安全声明标记语言（SAML），以及用于指定细粒度数据访问权力的 XML 访问控制标记语言（XACML）。

11.5.1 XML Signature

当一个 XML 文档由多方合著时，每一方都要对自己写作的部分签名。当使用网络层安全性时，这是不可能做到的。重要的是要确保文档某些部分的完整性，同时又可以在以后对同样的文档进行修改和添加。对于 XML 文档的安全交换以及业务事务的管理，XML Signature 规范奠立了一个良好的基础。XML Signature 的目的是确保数据完整性、消息认证和服务的不可抵赖性。

数字签名操作可应用在任意(但经常是 XML)数据上。XML Signature 定义了获取数字签名操作结果的模式。XML Signature 间接地应用在任意的数字内容(数据对象)上。对数据对象做摘要，生成的结果值(连同其他信息)放置在元素中，然后再做摘要并进行密码签名。XML Signature 本身通常指出原始签名对象的位置。XML Signature 可以签署多种类型的资源，如字符编码数据(HTML)、二进制编码数据(一个 JPG)、XML 编码的数据、XML 文档的特定部分，或由 XPointer 引用的外部数据。

通常，有三类数字签名：封外签名(Enveloping Signature)、封内签名(Enveloped Signature)、分离签名(Detached Signature)。在封外签名中，签名包含了被签名的整个 XML 文档，被签名的文档是 Signature 元素的一部分。在封内签名中，XML 签名嵌入在 XML 文档中。在分离签名中，XML 文档和签名独立存放，文档通常由一个外部的 URI 引用。分离签名意味着被签名的数据不在签名元素中，而是在 XML 文档的其他地方或者位于远程的某个地方。

签名验证要求被签名的数据对象是可访问的。通过引用封外签名对象、封内签名对象，以及分离签名对象，XML Signature 本身通常指明了原始被签名对象的位置。

清单 11.1 显示了基于[Simon 2001]和[Eastlake 2002a]的 XML 签名样例。这个简化的例子显示了：被签名的数据对象是一条新闻，清单 11.1 中的第一个 < Reference > 元素的 URI 属性标识了这条新闻。< Signature > 元素表示了 XML 数字签名。元素中有关被签名的原始数据对象的信息通过 URI 表示。在封外签名中，< Signature > 元素成为原始数据对象的父亲。在封内签名中，< Signature > 元素成为原始数据对象的孩子。在分离签名中，< Signature > 元素可以是原始数据对象的兄弟，或者 < Signature > 元素携带一个对外部数据对象的引用。清单 11.1 中的代码片段表示了一个分离签名，因为它不是被签名文档的组成部分。

< Signature > 元素允许应用携带摘要值以及其他信息，还能携带验证签名所需的密钥[Glbraith 2002]。该元素包含了实际签署的数据对象以及其他信息，其中 < SignedInfo > 元素提供有关 XML 签名的信息。它还包含一个 < SignatureValue > 元素，包含数字签名的实际值，即 < SignedInfo > 元素的加密摘要。

将 XML 文档转换成规范形式的过程称为规范化。XML 规范化意味着使用一个算法生成 XML 文档的规范形式，从而确保一些情况下的安全性，诸如当 XML 曲面表示(Surface Representation)发生变化时，或丢弃 XML 不重要信息(如实体或带前缀的命名空间)时。

< SignedInfo > 元素的第一个子元素 < CanonicalizationMethod > 用于指定规范化算法。在相关的 < SignedInfo > 元素做摘要并生成签名之前，< CanonicalizationMethod > 所指定的算法将应用在该元素上。第二个子元素 < SignatureMethod > 是一个密码学算法，用于将规范的 < SignedInfo > 转换到 < SignatureValue > 。

在 XML 签名中，< Reference > 元素指定了每个被引用的资源。< Reference > 元素通过 URI 属性标识了数据对象，并携带了数据对象的摘要值。每个 < Reference > 元素包括一个 < DigestMethod > 元素。该元素指定应用在数据对象上的摘要算法，生成的摘要值包含在 < DigestValue > 元素中。

清单 11.1　XML 签名样例

```
<?xml version="1.0" encoding="UTF-8"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
<SignedInfo Id="2ndDecemberNewsItem">
  <CanonicalizationMethod
    Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
  <SignatureMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
```

```
  <Reference
    URI="http://www.news_company.com/news/2004/12_02_04.htm">
      <DigestMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
  </Reference>
  <Reference URI="#AMadeUpTimeStamp"
            Type="http://www.w3.org/2000/09/
            xmldsig#SignatureProperties">
     <DigestMethod
       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
     <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
  </Reference>
   ... ...
</SignedInfo>
<SignatureValue>MC0E~LE=… </SignatureValue>
<KeyInfo>
  <X509Data>
    <X509SubjectName>
       CN=News Items Inc., O=Today's News Items, C=USA
    </X509SubjectName>
    <X509Certificate>
       MIID5jCCA0+gA...lVN
    </X509Certificate>
  </X509Data>
</KeyInfo>
<Object>
  <SignatureProperties>
    <SignatureProperty Id="AMadeUpTimeStamp"
      Target="#2ndDecemberNewsItem">
      <timestamp xmlns="http://www.ietf.org/rfcXXXX.txt">
        <date>2004122</date>
        <time>18:30</time>
      </timestamp>
    </SignatureProperty>
  </SignatureProperties>
</Object>
</Signature>
```

可选项 < KeyInfo > 元素提供了用打包的验证密钥验证签名的能力[Eastlake 2002a]。 < KeyInfo > 元素可能包含密钥、名称、证书和其他公钥管理信息，如带内(in-band)密钥分发或密钥协议数据。在清单 11.1 中，密钥信息包括发送者的 X.509 证书，包含了签名验证所需的公钥。

在清单 11.1 中，< Object > 元素是一个可选元素，主要用于封外签名。在封外签名中，数据对象是签名元素的一部分。 < Object > 中的 < SignatureProperties > 元素类型可包含签名的其他信息，如日期、时间戳、加密硬件的序列号，以及其他与特定应用相关的属性。

签名验证要求被签名的数据对象是可访问的。为验证签名，接收者使用签名者的公钥解码消息摘要，消息摘要包含在 XML 签名元素 < SignatureValue > 中。

XML Signature 提供自己的数据完整性。XML Signature 对于认证和不可抵赖性也是重要的；但是它本身不提供这些功能。WS-Security 标准描述了如何使用 XML Signature 将一个安全性令牌(表示了与安全相关的信息，参见 11.6.3 节)绑定到 SOAP 消息，并且 WS-Security 标准进行了扩展，将签名者的身份标识绑定到 SOAP 消息，从而提供了认证功能和不可抵赖性的功能 [O'Neill 2003]。

有关 XML Signature 的更多信息，以及 XMLSignature 指定的带外签名、带内签名、分离签名的例子，可参阅文献[Galbraith2002]和[Siddiqui 2003a]。

11.5.2　XML Encryption

XML Signature 规范并没有定义用于加密 XML 实体的任何标准机制。加密 XML 实体是提高 Web 应用安全性的另一个重要的安全特性。这个功能由 XML Encryption 规范提供。XML Encryption 规范是 W3C 开发的，W3C 支持对整个 XML 文档(或 XML 文档的一部分)进行加密。

XML Encryption 的步骤如下[Eastlake 2002b]：

(1) 选择要加密的 XML 文档(整个或部分)。

(2) 必要的话，将要加密的 XML 文档转换成规范形式。

(3) 使用公钥加密生成的规范形式。

(4) 发送加密的 XML 文档给预想的接收者。

因为 XML Encryption 并不固定于任何特定的加密模式，所以需要额外提供有关加密内容和密钥的信息。< EncryptedData > 元素和 < EnryptedKey > 元素完成了这一功能。XML Encryption 句法中的核心元素是 < EncryptedData > 元素，它与 < EnryptedKey > 元素一起用于将密钥从源传送到已知的接收者。加密的数据可以为任意数据，如 XML 文档、XML 元素、XML 元素的内容或对 XML 文档外资源的引用。数据加密的结果是一个包含或引用密码数据的 XML 加密元素。当加密一个元素或元素内容时，< EncryptedData > 元素替换 XML 文档加密版本中的元素或内容。< EnryptedKey > 元素提供加密中涉及的有关密钥的信息。

清单 11.2　XML 加密样例

```
<?xml version="1.0"?>
 <PaymentInfo xmlns="http://example.org/paymentv2">
  <Name>John Smith</Name>
  <CreditCard Limit="5,000" Currency="USD">
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/06</Expiration>
  </CreditCard>
 </PaymentInfo>

-----------------------------------------------------------------

<?xml version="1.0"?>
<env:Envelope>
 <env:Body>
   <PaymentInfo xmlns="http://example.org/paymentv2">
     <Name> John Smith </Name>
     <CreditCard Limit="5,000" Currency="USD">
       <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
         Type="http://www.w3.org/2001/04/xmlenc#Content">
         <CipherData>
           <CipherValue> A23B45C56 </CipherValue>
         </CipherData>
       </EncryptedData>
     </CreditCard>
   </PaymentInfo>
 </env:Body>
</env:Envelope>
```

清单 11.2 描述了引自[Eastlake 2002b]的 XML 加密样例。清单 11.2 中的第一部分显示了一个 XML 标记，该标记表示一些假想的支付信息，包括标识信息以及合适的支付方法的信息，如信用卡、转账或电子支票。清单 11.2 中的例子显示 John Smith 正在使用他的信用卡，信用卡的限额为 $ 5,000。列表的第二部分显示了：对于中介机构来说，了解 John Smith 使用具有特定限额的

信用卡是有用的，但是不知道卡号、签发者以及过期日期。在这种情况下，信用卡元素的内容（字符数据或孩子元素）是加密的。<CipherData>元素是包含加密内容的元素。XML Encryption 允许以两种方式携带加密的内容。如果加密的内容在原地，它作为<CipherValue>元素的内容携带。<CipherValue>元素是<CipherData>元素的孩子。这在如清单 11.2 所示。或者，XML Encryption 允许加密的内容存储在外部的位置，由<CipherData>元素的孩子<CipherReference>元素引用。

XML（Web 服务）防火墙接收清单 11.2 的内容（带加密正文的 SOAP 消息），并在 SOAP 消息请求转发到 SOAP 服务器前，先将内容转换为解密的形式。

有关 XML Encryption 的更多信息以及各种样例可参考[Galbraith 2002]。

11.5.3 XML 密钥管理规范（XKMS）

XKMS（http://www.w3.org/TR/xkms/）简化了 PKI 的集成，并简化了 XML 应用中的数字证书的管理。对于 PKI 密钥的处理与管理，促进基于 XML 的信任（Web）服务的开发，这是隐藏在 XKMS 背后的关键目标。XKMS 致力于消除使用 PKI 而带来的复杂性，使基于 XML 的应用更易于将安全机制合并到它们的应用环境中。

XKMS 使得认证、数字签名和加密服务（如证书处理和撤销状态核查）更容易集成到应用中，已没有与私有 PKI 软件工具包相关联的约束和其他一些复杂情况。图 11.14 显示了 XML Signature 和 XML Encryption 是如何与 XKMS 相关联的。通过 XKMS，很容易通过编程的 XML 事务进行访问驻留在服务器中的信任函数。

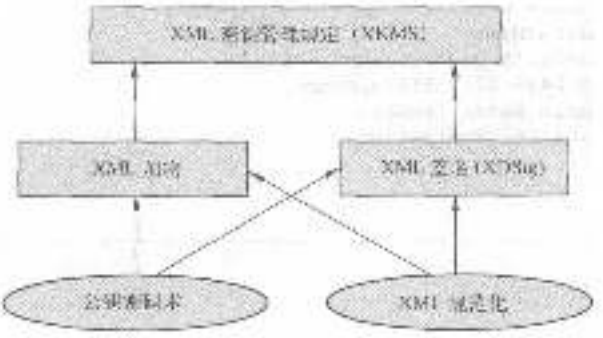


图 11.14 XML 信任框架的基本构建块

XKMS 支持三种主要的服务：注册服务，定位服务和验证服务。注册服务用于为第三方所管理的某个服务注册密钥对。一旦注册了密钥，XKMS 服务管理注册密钥的撤销、重发及恢复。定位服务用于获取用 XKMS 服务注册者公钥。验证服务提供定位服务提供的所有功能，并支持密钥验证。

图 11.15 显示了一个交互的例子。一个供应商向装运商发送一个装运消息，并对消息进行了加密和签名。供应商不再管理密钥信息，而是请求 XKMS 服务处理有关密钥处理的活动。在处理流程的一开始，供应商和装运商都使用注册服务在 XKMS 信任服务上注册它们的密钥对（步骤 1）。紧接着供应商注册密钥，供应商需要对发送给供应商的消息进行加密。由于这个目的，供应商向 XKMS 服务器发送一个定位请求（步骤 2），查找装运商的公钥。由于装运商已经用 XKMS 服务注册了它的密钥，服务就在应答中提供装运商的公钥。然后供应商使用这个公钥来加密消息，并应用它自身的私钥对消息进行签名，然后将加密和签名后的消息转发给装运商（步骤 3）。装运商接收到消息后，将包含在签名消息中的 XML Signature <KeyInfo>元素传递给 XKMS 服务

进行验证。

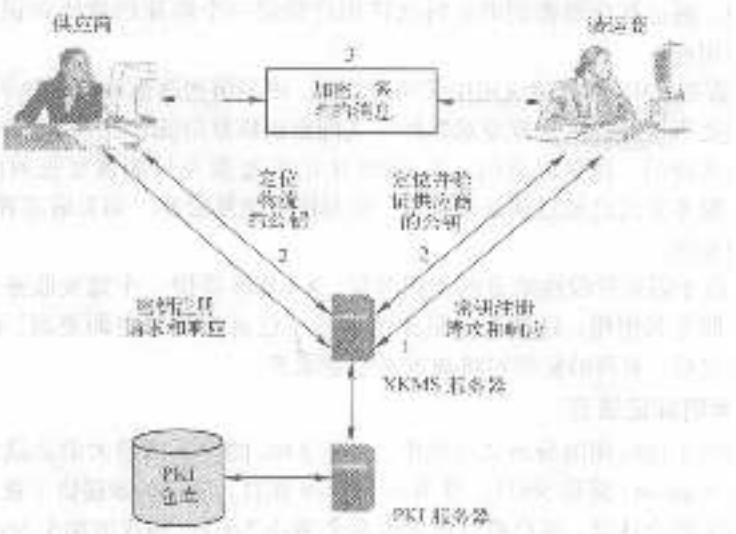


图 11.15　使用 XKMS 服务的样例

签名有两个验证阶段：

(1) 接收消息的应用(装运商)直接执行本地验证。在这个阶段，将检查文档，判断文档是否被正确签名了，并且在传送期间没有被篡改。这个阶段包括用签名者(供应商)的公钥来解码签名，然后将它与本地获得的签名进行比较。

(2) 联系 XKMS 服务并请求传输的公钥上的信息。在这个阶段，知道签名者(供应商)的标识，并检查该密钥是否已废除(在被盗的情况下)及有效的(还没有过期)。

XKMS 由两个主要关于部分组成：XML 密钥信息服务规范(X-KISS)和 XML 密钥注册服务规范(X-KRSS)。X-KISS 协议处理公共处理和验证，而 X-KRSS 协议处理密钥对注册。

1. XML 密钥信息服务规范(X-KISS)

在 XML 应用中，关于与 XML 数字签名、XML 加密数据或其他的公开密钥使用方式相关的信息，X-KISS 定义了通过依赖方来处理这些信息的协议。支持的功能包括定位特定标识符信息所需的公钥，并将这些密钥绑定到标识符信息。与 X-KISS 协同工作的应用接收依照 XML Signature 规范签名的消息。

X-KISS 通过两类服务提供接口：定位服务和验证服务。定位服务用于从 XML Signature 规范密钥信息元素包含的数据中找到附加的信息。验证服务用于确保密钥是有效的。

2. XML 密钥注册服务规范(X-KRSS)

X-KRSS 的目标是提供一个完整的、以客户为本的 XML 密钥生命周期管理协议。为达到这个目标，X-KISS 定义了一个基于 XML 协议的公钥信息注册。它允许 XML 应用将它的公钥对和相关的绑定信息注册到 XKMS 信任服务提供者。

X-KISS 通过以下服务支持证书的整个生命周期：

密钥注册：XML 应用的密钥对持有者通过注册服务器将它的公钥注册到信任的基础架构中。使用 KRSS 中经过数字签名的请求，将公钥发送给注册服务器。请求中可以包括名称和属性信息、认证信息以及拥有私钥的证明。

密钥撤销：撤销服务处理撤销先前注册的密钥绑定请求，其中密钥与任何相关的密钥身份证明

绑定在一起。密钥绑定撤销可能有不同的原因，但最终的结果都是相同的，即当前密钥绑定被认为是不可信任的。通过在注册密钥绑定时允许用户指定一个特殊的撤销标识，撤销服务认证对私钥失去控制的用户。

密钥恢复：若在XML客户端应用中使用了加密，则当解密数据所需的私钥丢失时，从统计上说恢复加密数据是不可能的。这就要求某种形式的密钥恢复措施。在X-KRSS中，这个功能不是由标准化的协议支持的，而是内置的。X-KRSS中的恢复服务只能恢复先前由第三方保存的私钥。通过向恢复服务发送已经过认证的请求，从而使用恢复服务。如果请求者认证正确，恢复服务就发回加密的私钥。

密钥重发：由于需要阶段性地更新密钥绑定，X-KRSS提供一个重发服务。这个服务的使用非常类似于注册服务的使用，只是重发服务请求一个已有密钥绑定的更新，而不是生成一个新的。在成功标识之后，更新的密钥对将被发送给请求者。

11.5.4 安全声明标记语言

在一个跨多个不同应用的分布式环境中，基于XML的应用的最大的挑战之一是用户认证和单点登录(Single Sign-on，简称SSO)。就Web Service而言，单点登录提供了使用多个Web Service的能力。或者基于单个认证，用户可以使用由多个Web Service构成的单个Web Service。

安全声明标记语言(也称安全断言标记语言，简称SAML)是一个关于Web Service单点登录的OASIS标准。基于SAML，用户在多个应用中仅需提交一次身份标识，并可将身份标识从一个企业传送到另一个企业。在版本1.1中，自由联盟标准组织(http://www.projectliberty.org/)开发了一些增强功能，包括身份标识联邦框架跨域认证、会话管理。在SAML 2.0中，这两个版本被合并到用于身份标识联邦和传输的更大的框架中。

SAML是一个厂商中立的、基于XML的标准框架，用于描述、交换与安全性相关的信息，这些信息称作断言(关于主体的事实申明)。SAML用于帮助不同应用组件和信任域之间的安全性信息交换[Hughes 2004]。在XML表单中，将安全性信息插入到断言中，从而实现安全性信息的交换。这些断言表达了有关终端用户的认证行为、对于访问特定资源的授权或者属性等方面的信息。可将SAML断言绑定到SOAP消息，并将其发送到支持SAML的Web Service中。

SAML使得不同的安全性系统能够进行互操作，同时各个组织可以保留他们自己的认证系统。既无须一个企业认证所有外部用户，也无须通过集中式的认证注册来认证合作伙伴，SAML提供了一个具有互操作性的、基于XML的安全性解决方案。通过SAML，进行协作的应用或服务以断言的形式交换用户信息和对应的授权信息。为了实现这一目的，对于传送安全性的文档的结构，SAML规范建立了断言和协议模式。通过定义如何交换身份标识和访问信息，SAML变成了一个共同的语言，各组织可以进行相互通信，且无须修改它们自身的内部安全性体系结构。

SAML的主要组件如下[Cantor 2004]：

(1) 断言：SAML定义了三类断言。这些断言声明了有关主体(例如，一个服务请求者)的一个或多个事实。认证断言需要用户证明他们的身份标识。属性断言包含用户的具体细节，诸如他们的信用额度或国籍。授权断言指定了是允许客户的请求还是拒绝客户的请求，并指定了客户特权的作用域。授权断言允许或拒绝对一个资源(诸如文件、设备、Web页面、数据库等)的访问。例如，授权断言通常是对一个请求(诸如是否允许Plastics Inc.访问一个有关产品设计规范文件和设计图的、具有机密性的Web页面)的响应。所有类别的断言都包含一个共同的元素集：主体、先决条件、认证声明。其中，主体指出了断言正在标识谁；先决条件指出了断言在什么情况下是有效的；认证声明则是有关如何进行断言的建议。每一个断言也包含了有关请求类型的信息。

(2) 请求/响应协议：SAML 定义了一个请求/响应协议，可用于获取断言。一个 SAML 请求既可以请求一个已知的断言，也可以进行认证、属性和授权决策请求，SAML 响应将返回所请求的断言。在 XML 模式中，定义了协议消息的 XML 格式和它们的扩展。

(3) 绑定：这个元素精确地详述了如何将 SAML 请求/响应消息交换映射到标准的消息传送协议或通信协议上。例如，SAML SOAP 绑定定义了 SAML 协议消息如何与 SOAP 消息通信。SAML URI 绑定定义了 SAML 协议消息如何通过 URI 解析进行通信。

(4) 描述文件：描述文件规定了如何在通信系统之间嵌入或传输 SAML 断言。一般地，对于特定的应用，SAML 的描述文件定义了约束以及对 SAML 使用方面的扩展，其目的是为了增强互操作性。例如，Web 浏览器单点登录描述文件规定了 SAML 认证断言如何在身份标识提供者和服务提供者之间进行通信，从而使得浏览器用户能够进行单点登录。Web 用户向身份标识提供者进行认证，身份标识提供者然后生成一个认证断言，这个认证断言将被传送到服务提供者，从而允许服务提供者为浏览器用户创建一个安全性环境。

图 11.16 阐明了 SAML 组件之间的关系。该图也显示了如何在 SAML 响应中携带 SAML 断言。SAML 响应本身嵌入在 SOAP 体中。注意，虽然一个 SAML 响应中通常仅有一个断言，但是一个 SAML 响应能够包含多个断言。

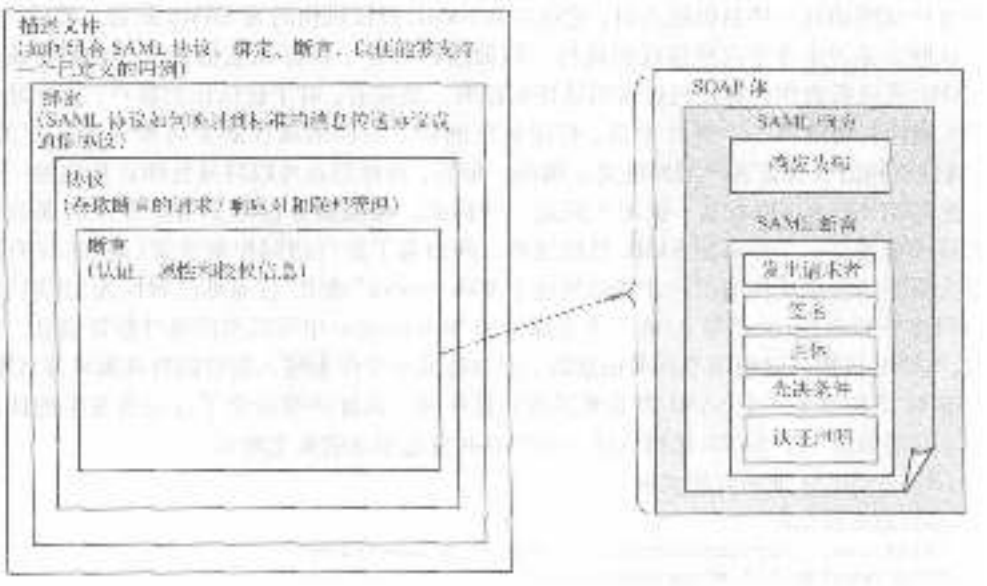


图 11.16　SAML 组件和断言结构

使用 SAML 中定义的协议，客户可以向 SAML 机构(Authorities)请求一个断言，并从 SAML 机构获取一个与这些活动相关的 SAML 断言形式的响应。图 11.17 阐明了 SAML 模型。在这个模型中，主体(客户)可将他的身份证明发送给三个不同的 SAML(认证、属性和授权)机构进行确认，从而实现了在这些机构中的认证。然后，主体可以获得断言引用。对访问资源(诸如一个 Web Service)的请求中可以包括断言引用。主体将这个信息转发给一个保护这个具体资源的策略执行点(Policy Enforcement Point，简称 PEP)模块。PEP 使用这个引用向断言发放机构或者策略决策点(简称 PDP)模块请求真正的断言(认证决策)。PEP 模块和 PDP 模块都是 XACML 访问控制语言的一部分。在下一章节中，我们将要讨论 XACML。SAML 和 XACML 是互补的标准，它们都有共同的基本概念和定义。

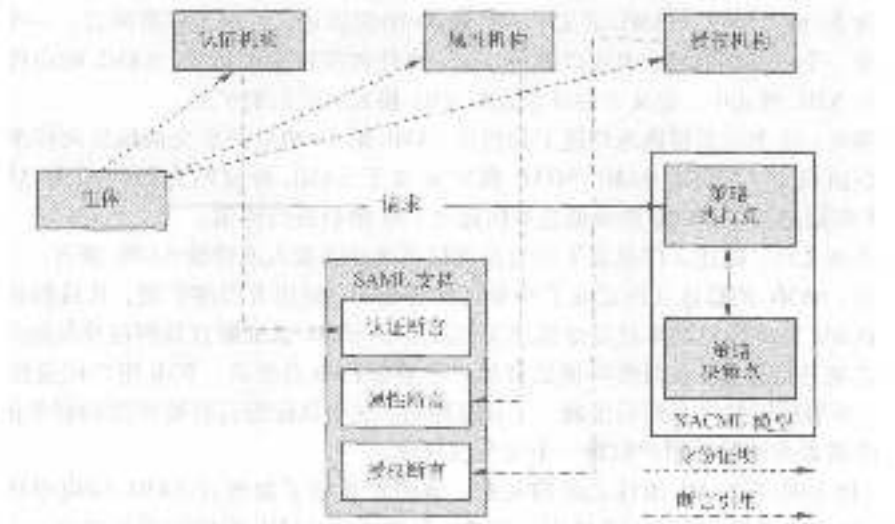


图 11.27 SAML 和 XACML 模型

当客户试图访问一些目标站点时，它将要从 SAML 授权机构转发 SAML 断言。然后，目标站点将确认断言是否来自于它所信任的机构。假如断言来自于目标站点信任的机构，则该站点就将该 SAML 认证断言作为客户已经得到认证的证明。紧接着，对于被认证的客户，目标站点能够向 SAML 属性机构请求一个属性断言，传递认证断言。返回的属性断言将要包含客户的属性。SAML 属性机构需要保证客户的属性是正确的。最后，目标站点可以将属性断言传递给一个授权服务，查询客户是否可以在某一资源上完成一个活动。授权服务能够是目标站点所属公司的一个本地服务或者是一个外部的 SAML 授权服务。例如为了进行访问控制决策（诸如"仅有那些已经认证为我们的合作伙伴的用户才能访问这个 Web Service"或者"仅有那些属性为'经理'的用户才能访问这个 Web Service"等），在一个受保护的 Web Service 中可以使用属性断言信息。在创建响应时，SAML 机构可以使用不同的信息源，诸如在请求中作为输入接收的外部策略库和断言。

清单 11.3 包含了一个 SAML 断言形式的认证声明。认证声明指定了过去所发生的认证行为的结果。我们假定一个 SAML 机构认证一个用户并发送相应的安全断言。

清单 11.3 SAML 认证断言的实例

```
<saml:Assertion
  xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
  MajorVersion="1" MinorVersion="0"
  AssertionID="XrazfaazDz0iXrUa"
  Issuer="www.some-trusted-party.com"
  IssueInstant="2004-07-19T17:02:00Z">
      <saml:Conditions
         NotBefore="2004-07-19T17:02:00Z "
         NotOnOrAfter="2004-07-19T17:16:00Z"/>
      <saml:AuthenticationStatement
         AuthenticationMethod="urn:ietf:rfc:3075"
         AuthenticationInstant="2004-07-19T17:02:00Z">
            <saml:Subject>
               <saml:NameIdentifier
                  NameQualifier=http://www.some-trusted-party.com
                  Format="...">
                  uid="OrderProcService"
               </saml:NameIdentifier>
```

```
                <saml:SubjectConfirmation>
                  <saml:ConfirmationMethod>
                      urn:oasis:names:tc:SAML:1.0:cm:
                      holder-of-key
                  </saml:ConfirmationMethod>
                    <ds:KeyInfo>
                      <ds:KeyName>OrderProcServiceKey
                        </ds:KeyName>
                      <ds:KeyValue> ... </ds:KeyValue>
                    </ds:KeyInfo>
                </saml:SubjectConfirmation>
            </saml:Subject>
        </saml:AuthenticationStatement>
    </saml:Assertion>
```

清单 11.3 针对一个订单处理场景。在该场景中，需要将订单消息从一个订单处理服务转发到装运处理服务，并最终发送到账单服务。清单 11.3 显示了订单处理场景中的一个具有认证声明的断言的例子。更具体地说，清单中的断言规定了：名为 OrderProcServiceKey 的公开密钥的拥有者是实体 OrderProcService。断言机构(some-trusted-party)已经使用了 XML 数字签名认证了 OrderProcService。基本信息指定了一个具有唯一性的标识符作为断言标识符，并指定了发出断言的日期和时间以及断言的有效期。

根 < Assertion > 元素包含了三个重要的子元素：< Conditions > 元素、< AuthenticationStatement > 元素和 < ds: Signature > 元素。< Conditions > 元素指定了断言的有效期。< AuthenticationStatement > 元素认证流程的输出(最终结果)。< ds: Signature > 元素包含了常规的数字签名标签。

认证的主体是 OrderProcService。SAML 规范提供了许多预先定义的格式，可以从中选择主体的格式，包括电子邮件地址和 X.509 主题名，还可以进行自定义。在清单 11.3 中，最初在“2004-07-19T17:02:0Z”使用 XML 数字签名(通过 urn:itef:rfc:3075 表示)认证实体 OrderProcService。最后，< AuthenticationStatement > 的 < SubjectConfirmation > 元素指定了断言的主体和包含断言的消息的作者之间的关系。就清单 11.3 而言，它仅包含一个断言。这个断言的主体是订单处理服务，并且订单处理服务本身最终将向装运处理服务发送一个装运请求。

对于安全系统，依赖方简单地依赖断言机构提供断言可能并不是很合适。SAML 定义了许多安全机制。通过这些机制，可阻止或检测安全攻击。对于依赖方和断言方，主要机制是有一个预先存在的信任关系，通常涉及 PKI。虽然并不强制使用 PKI，但建议使用 PKI 技术。对于每一个描述文件，描述了特定机制的使用[Hughes 2004]。

虽然 SAML 对于用户身份证明、认证和授权定义了相应的机制，但是 SAML 并没有处理隐私策略。恰恰相反，合作伙伴站点负责开发有关用户认证和数据保护方面的共同需求。然而对于在服务间传输安全性信息的文档，SAML 定义了这些文档的结构。对于电子商务应用和 Web Service，SAML 提供了单点登录和端到端的安全性。对于在用户(或者依赖方)与多个发送方之间的安全的登录信息的交换，SAML 标准提供了相应的机制，从而使得发送方可以使用它们自己选择的认证方法，例如 PKI、散列或口令。

SAML 生成有关身份证明的断言，而不是真正地对用户进行认证或授权。通过认证服务器以及目录服务器的协作，可真正地对用户进行认证或授权。SAML 链接到真正的认证，并基于那个事件的结果生成相应的断言。

文献[Siddiqui 2003b]中例举了一些使用 SAML 的例子，诸如应用能够请求 SAML 机构发出一个 SAML 断言，又如在 WS-Security 应用中使用 SAML 断言。

11.5.5 XML 访问控制标记语言

SAML 仅能定义如何交换身份标识和访问信息。如何使用这个信息是 XACML 的责任。XACML 是 SAMK 的扩展。XACML 允许访问被指定的控制策略。XACML 使用了与 SAML 相同的主题和动作定义。定义组织的安全性策略和进行授权决策时，需要使用一些规则，XACML 提供了一个表示这些规则的词汇表。简单地说，XACML 是一个通用的访问控制策略语言，该语言提供了一个管理授权决策的语法(使用 XML 语言定义)。

XACML 有两个基本组件[Proctor 2003]：

(1) 基于访问控制策略语言，开发者可以指定有关谁能访问/访问什么/何时访问的规则。可以使用访问控制策略语言描述通常的访问控制需求。对于定义新的功能、数据类型、组合逻辑等，访问控制策略语言有一些标准的扩展点。

(2) 请求/响应语言描述了访问请求，并描述了对那些查询的响应。用户通过请求/响应语言可以构造一个查询，用来询问是否允许一个特定的操作，并对返回结果进行解释。响应中时常包括是否允许请求的回复，这一回复通常使用下列四个值之一进行表示：允许(permit)、拒绝(deny)、不确定(indeterminate)和不合适(not applicable)。其中，不确定表示出现一个错误或丢失了一些所需的值，因此无法决定；不合适表示服务无法回复请求。

除了提供请求/响应和策略语言，XACML 也提供了其他的关系，例如发现应用于特定请求的策略，又如对策略的请求进行评估，从而提供一个肯定或否定的回复。

基于包括以下准则在内的一些准则，XACML 提供了细颗粒的活动(诸如读、写、复制、删除)控制：

- 用户请求访问的属性。属性是已知类型的命名值，可以包括发送请求者的标识符，或者发送请求的日期和时间，例如“只有部门经理及其以上人员才能查看该文档”。用户名、参与机密的资格、想要访问的文档、当日时间等都是属性值。
- 请求所基于的协议，例如“仅通过安全的 HTTP 协议才能查看这些数据”。
- 所使用的认证机制。

在一个典型的 XACML 使用场景中，一个主体(例如，用户或者 Web Service)可能想在一个特定的资源上采取一些操作。该主体向保护资源的 PEP 实体(例如文件系统或者 Web 服务器)提交它的查询。基于主体、动作、资源的属性以及其他相关的信息，PEP 生成一个请求(使用 XACML 语言)。然后，PEP 将请求转发给 PDP 模块。PDP 模块将分析请求，检索可以应用到请求上的(使用 XACML 策略语言编写的)策略。按照评估策略的 XACML 规则，PDP 模块将决定是否准许访问。(使用 XACML 响应语言表示的)回复将被返回给 PEP。PEP 然后可允许或拒绝请求者的访问。PEP 和 PDP 两者既可能都包含在同一个应用中，也可能分布在几个服务器中。

XACML 的策略语言模型如图 11.18 所示。XACML 定义了一个顶层策略元素 < PolicySet > 。< PolicySet > 表示了一个访问控制策略。< Policy > 元素包含了一组 < Rule > 元素，假如需要还可包含一个或多个 < Obligation > 。使用规则组合算法，可将一组 < Rule > 元素链接在一起。规则是一些表达式，这些表达式描述了允许或拒绝资源访问请求的条件。一个策略可以有任意数量的 < Rule > 。< Rule > 包含了 XACML 策略的核心逻辑。< Rule > 包含一个 < Condition > 。< Condition > 是一个布尔函数并有 < Effect > 。对于满足条件的规则，< Effect > 指出了规则编制者所预想的结果。在执行授权决定时，将要同时执行一个策略或策略集。< Obligation > 是一个在策略或策略集中指定的操作。顶层 < PolicySet > 元素 < Policy > 是一个容器。该容易可以包含一组 < Policy > 或其他的 < PolicySet > 元素，以及对于远程机器上的策略的引用。每一个 XACML 文档都精确地包含一个 < Policy > 或 < PolicySet > 根 XML 标签。

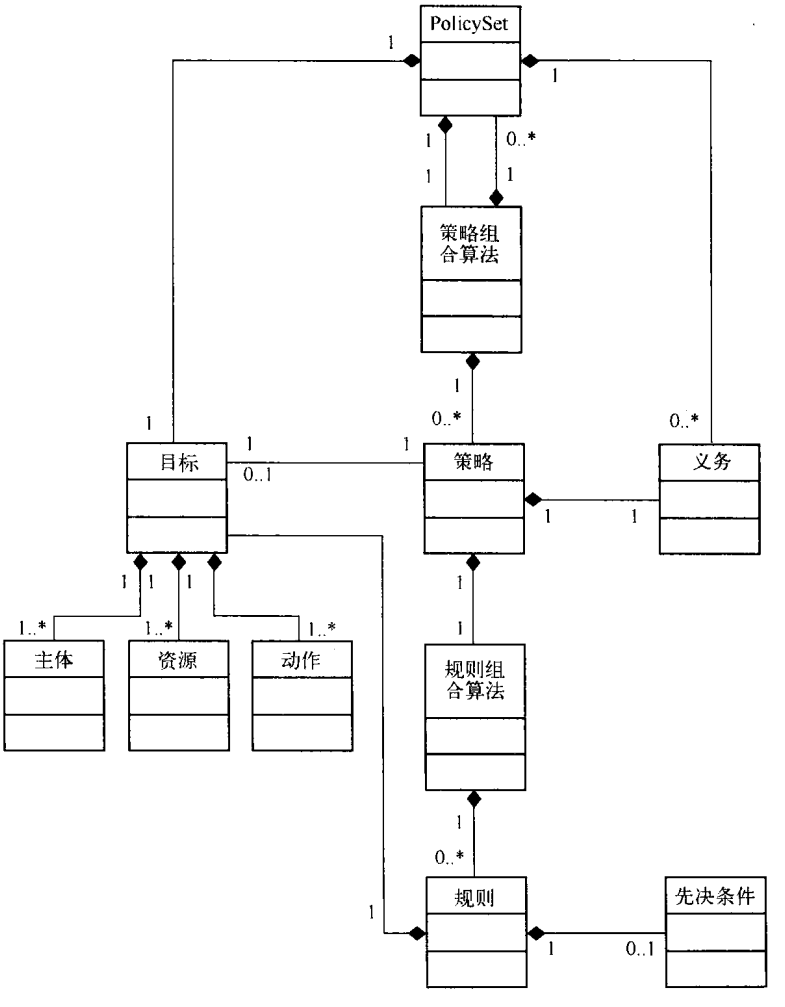


图 11.18　XACML 策略语言模型

一个 < Policy > 或者 < PolicySet > 可以包含多个策略或规则。每一个策略或规则可以评估不同的访问控制决策。XACML 通过组合算法能够使得各个决策不相互冲突。每一个组合算法都表示了将多个决策组合为一个决策的一种方式。有两个组合算法，即 < PolicySet > 使用的 < Policy-CombiningAlgorithm > 和 < Policy > 使用的 < RuleCombiningAlgorithm > 。

XACML PDP 部分是查找策略，可将其应用到一个特定的请求中。为此，XACML 提供了一个 < Target > 元素。对于一个特定的请求，PDP 可使用策略语句的 < Target > 来确定在哪里应用策略。目标指定了策略的主体、资源和动作(诸如读、写、复制、删除)。当 < PolicySet > 、< Policy > 或 < Rule > 应用到一个特定请求时，必须满足具体的 < Subject > 、< Resource > 和 < Action > 子元素。对于 < Subject > 、< Resource > 和 < Action > 子元素，< Target > 本质上是一组简化的先决条件。布尔函数将请求中的特定值与 < Target > 中的值进行比较。假如 < Target > 中的所有条件都满足，然后可将相关的 < PolicySet > 、< Policy > 或者 < Rule > 应用到请求中。一旦找到 < Policy > ，并确认可将其应用到请求中，则可评估相关的 < Rule > 。

假定有一个名为“Plastics Supply Inc. ”的公司。清单 11.4 显示了该公司的一个简单的策略。在该策略中，假如主体的电子邮件地址属于 plastics_supply. com 域，则该主体可以在该公司的任

何资源上进行任何操作。这个策略被赋予了一个具有唯一性的标识符。在规则组合算法中，假如任一规则的评估结果是“决绝(deny)”，则策略将返回“deny”，然而假如所有规则的评估结果都是“允许(permit)”，则策略将返回“permit”。策略的 < Target > 部分指定了在哪些请求中使用该策略。在策略中，即使仅有一条规则，规则也必须具有唯一标识符。 < Rule > 元素规定了：当规则为“真(true)”时，规则所产生的作用。规则所产生的作用既可以是“允许”，也可以是“拒绝”。在清单 11.4 所示的例子中，假如满足该规则，则该规则所产生的结果将是“允许”。这意味着，就这条规则而言，将允许所请求的访问。最后，规则的 < Target > 部分类似于策略中的目标，描述了将哪一个规则作用何种请求中。规则的目标也类似于策略目标本身，但是也有一个重要的差别。规则目标描述了请求的主体必须匹配的、具体的值。 < SubjectMatch > 元素在 MatchID 属性中指定了一个匹配函数。 < SubjectMatch > 元素还指定了一个常量值“plastics_supply. com”。通过 < SubjectAttributeDesignator > 元素， < SubjectMatch > 元素还指定了一个指向具体的主体属性的指针。仅当匹配结果为“真”时，才将规则应用到特定的请求中。

清单 11.4 XACML 策略的样例

```
<Policy PolicyId="identifier:example:SimplePolicy1"
         RuleCombiningAlgId="identifier:rule-combining-
           algorithm:deny-overrides">
   <Description>
     Plastics Supply Inc. access control policy
   </Description>
   <Target>
      <Subjects><AnySubject/></Subjects>
      <Resources><AnyResource/></Resources>
      <Actions><AnyAction/></Actions>
   </Target>
   <Rule
     RuleId="identifier:example:SimpleRule1"
     Effect="Permit">
     <Target>
        <Subjects><Subject>
          <SubjectMatch
              MatchId="urn:oasis:names:tc:xacml:1.0:function:
                                                  rfc822Name-match">
             <AttributeValue
               DataType="http://www.w3.org/2001/XMLSchema#string">
               plastics_supply.com
             </AttributeValue>
             <SubjectAttributeDesignator
                AttributeId="urn:oasis:names:tc:xacml:1.0:
                                                subject:subject-id"
                DataType="urn:oasis:names:tc:xacml:1.0:
                                              data-type:rfc822Name"/>
          </SubjectMatch>
        </Subject></Subjects>
        <Resources><AnyResource/></Resources>
        <Actions><AnyAction/></Actions>
      </Target>
   </Rule>
</Policy>
```

除了定义策略的标准格式，XACML 还定义了表示 < Request > 和 < Response > 的标准方式。< Request > 元素和 < Response > 元素提供了和 PDP 进行交互的标准格式。向从 PEP 向 PDP 发送请求时，请求基本由专有属性构成。为了确定合适的访问决策，需要将请求属性的值与策略中的属性值进行比较。

< Request > 包含了一些属性，这些属性描述了主体、资源、动作或者访问请求所涉及的环境元素的特征。属性是命名的已知类型的值，可以包括发送请求者的标识符，或者发送请求的日期和时间。< Subject > 元素包含进行访问请求的实体的一个或多个属性。可以有多个主体，并且每一个主体能够有多个属性。除了属性，在资源部分也可以包含被请求的资源的内容。通过 XPath 表达式进行策略评估时，将涉及资源。在 < Request > 中必须包含的 < Attribute > 是资源标识符。

< Response > 由一个或多个结果组成，分别表示了不同的评估结果。通常情况下，< Response > 中仅有一个结果。每一个 < Result > 包含了一个决策（允许、拒绝、不合适或不确定）、一些状态信息（例如评估为何会失败），以及一个或多个可选的 < Obligation > 。< Obligation > 元素被定义为一个特定的动作。一旦完成授权决策，在允许或决绝访问前，PEP 有义务执行 < Obligation > 元素中定义的动作。例如，每次访问客户的金融记录时，都有义务创建一个数字签名的记录。

清单 11.5 显示了一个假想的请求，该请求被提交到 PDP，PDP 执行清单 11.4 所描述的策略。生成决策请求的访问请求可以如下描述：John Smith 的电子邮件地址是 jsmith@ plastics_supply. com。John Smith 想要查看他在 Plastics Supply Inc. 公司的缴税记录。在清单 11.5 中，仅有一个主体涉及这个请求，并且这个主体仅有一个属性：这个主体的身份标识，表示为一个电子邮件地址。

清单 11.5　XACML 请求的样例

```
<Request>
 <Subject>
      <Attribute
               AttributeId="urn:oasis:names:tc:xacml:1.0:subject:
               subject-id"
        DataType="identifier:rfc822name">
          <AttributeValue>
             jsmith@plastics_supply.com
          </AttributeValue>
      </Attribute>
  </Subject>
  <Resource>
      <Attribute AttributeId="identifier:resource:resource-uri"
                 DataType="xs:anyURI">
        <AttributeValue>
          http://plastics_supply.com/tax-record/employee/JohnSmith
        </AttributeValue>
      </Attribute>
  </Resource>
  <Action>
      <Attribute AttributeId="identifier:example:action"
                 DataType="xs:string">
        <AttributeValue>read</AttributeValue>
      </Attribute>
  </Action>
</Request>
```

处理请求上下文的 PDP 在策略库中定位策略。它将请求上下文中的主体、资源、动作和环境与策略目标中的主体、资源、动作和环境。对于清单 11.5 中请求，相应的响应上下文如清单 11.6 所示。

清单 11.6 XACML 响应

```
<Response>
    <Result>
        <Decision> Permit </Decision>
    </Result>
</Response>
```

11.6 安全的 Web Service

迄今为止本章已经讨论了 XML 安全性方面的技术，诸如 XML 签名和 XML 加密，或者可应用于 XML 应用的信息安全性方面的功能，例如密钥管理、认证、访问控制规则(如 XKMS、SAML 和 XACML)。在本节中，我们将讨论如何在 Web Service 安全性环境中使用这些技术。在引入这一主题之前，我们将解释 Web Service 安全性的技术含义和挑战。

11.6.1 Web Service 应用层面临的挑战

Web Service 依赖消息安全性，即将安全性技术应用到消息本身。消息安全性主要集中在两个重要的方面：首先，未被授权者不能获知消息内容(机密性)；其次，防止非法修改消息的内容(完整性)。因为对于请求和响应可以独立地应用不同的安全策略，通过把消息安全地分配给不同的接收者(真实性)，消息安全性也保证了选择性[Rosenberg 2004]。当然还有一个需求，那就是内容访问控制。对于 Web Service 应用，目前使用 SSL 和 TLS 来提供传输层安全性。首先我们将要分析 SSL 的缺点，然而将讨论应用(消息)层安全性所面临的重要挑战。

Web Service 的一个严重的弱点是它采用了 HTTP。在企业基础架构中，HTTP 传输(互联网的基本传输机制)使用 SOAP 能穿越已有的网络防火墙，与应用建立通信。因此，这就给病毒和黑客入侵带来了很大的隐患。传统上，保护内容的常用方式是 SSL、TLS、VPN 和 IPSec。例如，SSL/TLS 提供了几个安全特性，包括认证，以及传送时消息的数据完整性和数据机密性。SSL/TLS 能够提供点到点的安全会话，并可提供传输层的安全通信。SSL/TLS 也提供基本的、点到点的数据保密，但是并不提供完备的安全性。对于 Web 浏览器和 Web 服务器之间基于无状态的 HTTP 的安全交互而言，SSL/TLS 无疑可以满足一些基本的安全需求。然而当 SOAP 请求的路由需要途径几个服务器时，SSL/TLS 就不适用了。当必须将安全消息从一个中介服务器传送到另一个中介服务器时，SSL/TLS 就暴露了它的不足之处。图 11.19 显示了点到点的安全配置和端到端的安全配置。

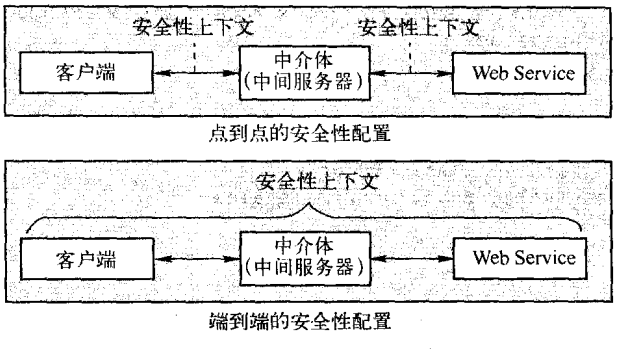


图 11.19 点到点安性全配置与端到端的安全性配置

SOAP 可支持一个或多个中介体(中间服务器)，中介体可以基于 SOAP 头部或 HTTP 头部转发或重新路由 SOAP 消息。因此在 Web Service 环境中，中介体的问题非常重要。为了尽量扩大

Web Service 的作用范围，需要一个端到端的安全性，而不仅是点到点的安全性。在端到端的安全性拓扑结构中，消息创建者可能已经存入了有效载荷，但中介体后来可能查看或修改消息。因此，对于中介体而言，它必须有一种方式可以仅读取消息中的一部分。更具体地说，仅读取告知中介体如何操作的指令部分，而不读取消息中的具有机密性的有效载荷部分。多个 Web Service 可编排起来长时间运行，在这样的 Web Service 中可能涉及多个请求、响应和分叉，并且需要保证安全性。在这样的环境中，若要中介体仅读取指令部分而不读取消息中的具有机密性的有效载荷，问题就变得越加复杂了。在所有的这些场景中，都需要依赖中介体转发消息。当中介体在传输层之上接收和转发数据时，可能会影响到流经它的数据的完整性以及丢失安全性信息。这就使得上游的消息服务器必须依赖前面的中介体所进行的安全性评估，并需要完全信任中介体对消息内容所做的处理。

对于流入和流出的消息，Web Service 需要很高的粒度性(例如不同的安全性需求)以及路点可视性(例如，对于消息的部分可视)。Web Service 需要按照它们的安全性策略维护安全的上下文并进行相应的控制。对于所选择的文档中一些内容的加密和数字签名，以及重写头部，提供 Web Service 安全性的基础架构需要具有 XML 的粒度性。然而，存在的问题是，虽然可使用 SSL/TLS 进行数据保密/加密、签名/完整性、认证/信任，但是对于点对点通信仅提供了传输层粒度，并没有提供加密的粒度性，诸如当在暴露明文形式的路由信息时，加密敏感信息。对于 SSL/TLS，单个的消息要不整个都是安全的，要不整个都是不安全的。因此，消息的特定部分的安全性成为了一个重要的议题。例如，一个 Web Service 可以使用 Kerberos 票据来认证客户，另一方式是仅支持客户端 SSL 证书。现在出现一个问题，即这些服务之间如何相互认证？类似地，随着开发者开始编写企业级业务线应用，也出现一个问题，何时两个或多个 Web Service 在不同的安全域下运行？各个域很有可能都维护它自身的截然不同的配置。为了解决这类问题，需要提供一个安全的单点登录和认证机制，并需要提供一个标准化的方式来获取合适的安全性身份证明，从而以便能够证明身份标识。

Web Service 需要一个强大的、灵活的安全性基础架构。利用 SSL/TLS 提供的传输机制可以开发安全性基础架构，并可在其上扩展高级应用层安全性机制，从而提供一整套全面的 Web Service 安全性，可用于解决消息层的各种安全性问题。

为了解决安全性挑战，包括 W3C、OASIS、自由联盟等在内的一些标准化组织已经提出了许多安全性标准，以便能够解决与认证、基于角色访问控制(RBAC)、消息传送和数据安全性等相关的问题。这些标准的目的就是加强 Web Service 的安全性。Web Service 的基础的安全性标准称作 WS-Security。在 WS-Security 上下文中，可创建和使用许多标准，诸如 XML Encryption、XML Signature 和 SAML 等。本章的余下部分将探讨 Web Service 安全性模型。

11.6.2　Web Service 安全性路线图

为了解决端到端的安全性，IBM 和微软共同制定了一个 Web Service 安全性规划和开发一组 Web Service 安全性标准规范和技术的路线图。在 Web Service 环境中，这些规范以一致的方式处理消息交换的安全性[WS-Roadmap 2002]。所提出的安全性框架和路线图足以构造高层密钥交换、认证、授权、审计和信任机制。安全性框架和路线图还提供了一个集成抽象，允许系统和应用在不同的安全性系统和技术中搭建桥梁。如图 11.20 所示，安全性框架包含一个称为 WS-Security 的基本标准，以及其他的一些依赖于 WS-Security 的安全性标准。WS-Security 建立在 XML Signature、XML Encryption、SAML 以及其他一些标准的基础之上。除了很好地定义了 WS-Security，下面总结的规范有些依然正在开发过程中。

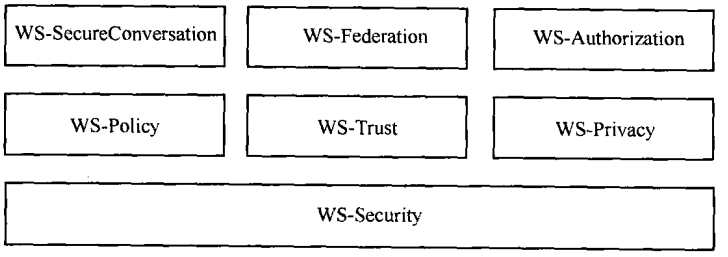


图 11.20 Web Service 安全路线图

WS-Security：这一 SOAP 扩展主要致力于实现消息内容的完整性和机密性。可使用所指定的机制与许多安全模型和加密技术相配合。

WS-(Security)Policy：WS-SecurityPolicy 是 WS-Security 的补遗。对于应用于 WS-Security 的 WS-Policy，WS-SecurityPolicy 表示了策略断言。WS-SecurityPolicy 中的安全性策略断言指定了它们的 Web Service 的安全性需求。这些安全性需求包括所支持的加密和数字签名的算法、保密属性(或称隐私属性)、以及这些信息如何绑定到 Web Service 中。

WS-Trust：对于安全性令牌的请求与发放以及信任关系的管理，该规范定义了一系列的基于 XML 的原语。

WS-Privacy：该规范综合运用 WS-Policy、WS-Security 和 WS-Trust 与保密策略进行通信。部署 Web Service 的组织规定了这些保密策略，并且保密策略需要流入的 SOAP 请求中包含断言，并且发送者遵循这些保密策略。

WS-SecureConversation：WS-SecureConversation 定义了一些能够提供安全通信的扩展。这些扩展构建在 WS-Security 的基础之上。

WS-Federation：WS-Federation 用于跨不同信任域的身份标识、属性、认证和授权联邦。

WS-Authorization：WS-Authorization 与 XACML 有许多重叠。该规范描述了如何指定和管理 Web Service 的访问策略。

在图 11.20 所描述的安全性路线图中，WS-Security 充当了一组可组合的安全性构建块的基础块。安全性构建块需要依赖该基础块。这意味着，可以使用 WS-Security、WS-Policy、WS-Trust 和 WS-SecureConversation 等安全性构建块来构成诸如 WS-Federation 等不同的构建块标准。

安全性路线图中的标准在三个相关的方面处理了基于分布式消息的安全性。这三个方面分别是互操作性、信任和集成。安全上下文中的互操作性意味着先前没有通信的独立的异构系统能够协同工作、相互理解。当在不同的系统之间进行安全通信时，这一能力尤其重要。与安全的互操作性相关的安全性标准族包括：WS-Security、WS-SecurePolicy 和 WS-SecureConversation。对于 Web Service 安全性的信任也是一个需要在关系中进行表示的重要的要素。既可以直接建立信任关系，也可以间接推断出信任关系。帮助提升信任的安全性标准族包括：WS-Trust 和 WS-Privacy。通过跨组织边界扩展和统一(异构的)系统体系结构，Web Service 安全性的集成提高了跨组织边界的集成的互操作性，以致可在新的应用中复用已有的服务。这意味着需要集成身份标识和服务操作所基于的信任模型。与集成相关的安全性标准族包括：WS-Federation 和 WS-Authorization。

我们首先将要介绍一个通用的 Web Service 安全性模型。紧接着，我们将要依次讨论安全性路线图规范，其中一开始首先讨论互操作性安全性标准，然后讨论信任标准，最后讨论安全性集成标准。

11.6.3　Web Service 安全性模型

在 Web Service 上下文中，无论是否使用了应用自身的安全性基础架构和机制（诸如 PKI 或 Kerberos），应用都必须能够进行互操作。为了实现这一点，Web Service 路线图规范已经定义了一个抽象的安全性模型和体系结构[WS-Roadmap 2002]。如图 11.21 所示，这个模型是基本的，因此可适用于许多不同的应用。

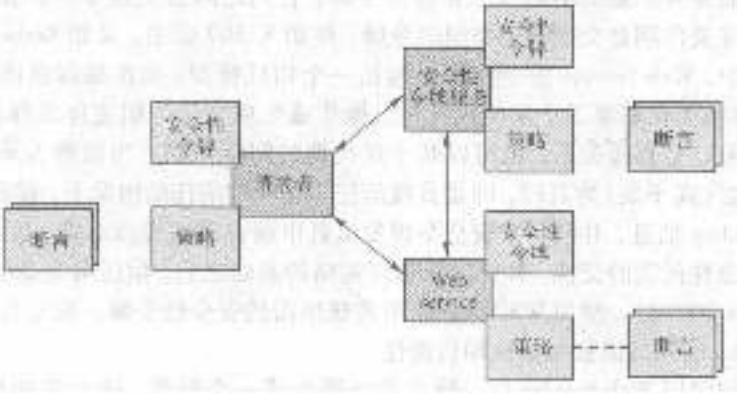


图 11.21　基本的 Web Service 安全性和信任模型

可按照通用的协议和通用的策略规则来定义 Web Service 安全性模型，其中通用协议用于交换通用的（令牌）断言，这些断言是由服务使用者提供的，而策略规则主要关于服务提供者实施的断言。这个方式的优点是，跨不同的管理域通常仅需要交换通用的策略和令牌元数据，在这些域中可自由使用不同的策略和令牌机制，诸如 PKI、活动目录和 Kerberos。

如图 11.21 所示，在 Web Service 安全性模型中涉及三方，分别是：请求者、Web Service 和安全性令牌服务（深色的阴影模块）。在 Web Service 安全性模型中，可将不同的安全性技术抽象成互操作的格式，这些格式定义了通用的策略模型。这些安全技术可模块化为下列构造块：策略、策略断言、断言和安全性令牌。对于流入的 SOAP 消息，（安全性）策略确定了安全性机制。对于流出的 SOAP 消息，（安全性）策略确定了需要添加到消息中的增强安全性的方式。例如，安全性策略断言指定了各个 Web Service 的安全性需求。安全性策略断言包括所支持的加密和数字签名算法、保密属性，以及如何将该信息应用到 Web Service。断言是一个关于主体（人、应用或业务实体）的声明。声明既可以直接关于主体，也可以是关于通过特性（诸如授权的身份标识）与主体关联的依赖方。断言能够是安全性令牌表达的声明。例如，可以使用一个断言来宣称发送者的身份标识或授权角色。

安全性令牌可以被视为数据（添加到 SOAP 头部）。安全性令牌表示了绑定用户或消息的发起者的断言，例如他们的身份标识、他们的权利，以及他们在一定时间内访问特定资源的授权。安全性令牌的具体例子包括用户名/口令对、SSL 客户端证书、XRML 许可证和 Kerberos 票据。可以使用图 11.21 中的安全性令牌来提供一个端到端的安全性解决方案。在参与消息交换的各方之间，可直接地或间接地共享安全性令牌。安全性令牌声称了断言，可使用安全性令牌来维护认证机密数据（authentication secret）之间的绑定，或者密钥与安全性身份标识之间的绑定[Nadalin 2004]。一个机构可使用它的密钥来签名或加密安全性令牌，来担保或认可安全性令牌中的断言，从而对令牌中的断言进行了认证。X.509[Hallam-Baker]证书表示了一个对象和它的公开密钥之间的绑定关系，它是认证中心支持的签名安全性令牌的一个具体例子。

有两种方式获取安全性令牌。一种方式是直接向一个合适的机构请求一个令牌，如图 11.21

所示。另一种方式是间接地通过一个可信的第三方代理获取。例如，为了获取一个 Kerberos 票据，应用需要和 Kerberos 密钥分发中心联系。若应用需要获取 X.509 证书，则应用需要和认证中心(CA)联系。在图 11.21 中，发出安全性令牌的第三方称为安全性令牌服务。安全性令牌服务的任务是：通过不同类型的令牌交换，将分立式的安全孤岛连接起来，使它们成为联邦安全网络。X.509 认证中心、Kerberos 密钥分发中心和 PKI 认证中心都是安全性令牌服务的很好的例子。安全性令牌服务并不是简单地发出和验证令牌，它们之间也交换令牌。作为仲介实体的安全性令牌服务的主要作用是交换特定类型的令牌，例如 X.509 证书，又如 Kerberos 票据。

在图 11.21 中，Web Service 安全性模型包括一个信任模型(如连接深色阴影部分的实线所示)。当一个实体将要依赖第二个实体执行一组操作或生成关于一组主体或作用域的许多断言。信任关系能够既可以是直接关系，也可以基于是代理的间接关系。当依赖方承认请求者发送的令牌中的全部断言(或子集)为真时，即是直接信任。在代理信任的情况下，使用“信任代理”(第二方)读取 WS-Policy 信息，并向一个安全令牌发出者申请合适的安全令牌，因此为第三方担保。信任关系基于安全性代理的交换、仲介以及信任策略的基础之上，信任策略是由相应的安全性机构建立的[Cabrera 2005d]。使用 WS-Security 可传送所需的安全性令牌。安全性令牌利用了 XML 签名和 XML 加密，确保的消息完整性和机密性。

当请求者希望调用 Web Service 时，请求者必须生成一个断言，诸如它的身份标识或特权。另一方面，每一个 Web Service 都有一个应用到它的安全性策略，例如 Web Service 可能需要消息的加密和数字签名以及请求者的标识。这类 Web Service 的策略指定了访问 Web Service 的安全需求。Web Service 从请求者那里可能接收一个包含安全性令牌的消息，并且可能使用 WS-Security 机制在消息中应用一些保护措施。

当应用通过 SOAP 和 WS-Security 发送安全性断言时，它将必须考虑如何在消息中表示这些断言。Web Service 安全性模型指定了安全性令牌中所包含的所有断言，而安全性令牌则隶属于一个特定的请求消息[WS-Roadmap 2002]。例如，基于口令或 X.509v3 证书的身份标识是安全性断言。因此，需要将表示为安全性令牌。

通常的安全性消息传送模型(断言、策略和安全性令牌)包含并支持一些更具体的模型，包括基于身份标识的安全性、访问控制列表、基于能力的安全性[Rosenberg 2004]。它允许使用一些已有的技术，诸如口令、X.509v3 证书、Kerberos 票据等。若该安全性模型与 WS-Security、WS-Policy 原语结合起来，可对高层密钥交换、认证、基于策略的访问决策、审计和复杂的信任关系提供足够的支持。

11.6.4 WS-Security

WS-Security 是一个 OASIS 安全性标准规范。该规范提出了一个用于构建安全的 Web Service 的 SOAP 扩展标准集，可将安全性令牌作为消息的一部分发送，并实现了消息内容的完整性和机密性[Nadalin 2004]。该规范充当了一个构建块，可与其他 Web Service 扩展和特定的高层应用协议相协作，从而与许多安全性模型(包括 PKI、Kerberos 和 SSL)和安全性技术相配合。

WS-Security 主要描述了如何使用 XML 签名和 XML 加密来增强 SOAP 消息的安全性。WS-Security 定义了如何在 SOAP 消息中包含安全性令牌，还定义了如何使用安全性规范来加密和签名这些令牌，以及定义了如何签名和加密 SOAP 消息的其他部分[Nadalin 2004]。通过提供端到端的安全性，WS-Security 模型也满足 SOAP 端点和中介体的需求。WS-Security 定义了一些方案。当消息穿越中介体，甚至当这些中介体自身使用安全性功能时，这些方案可确保 SOAP 消息的完整性和机密性。

在介绍 WS-Security 元素之前，我们将要介绍一个简单和直观的场景。该场景描述了如何应

用 11.5 节中所描述的基于 XML 的安全性机制以及 WS-Security 来保护一个简化的订单处理事务。该订单处理事务涉及客户、销售者、信用评等公司、受到信任的第三方、装运服务之间的简单的业务对话。在此之后，对于实现 SOA 集成部署或基于 Web Service 集成部署的企业，我们将要描述基于 WS-Security 的 Web Service 标准是如何帮助这些企业处理所面对的安全性挑战。这两个主题将帮助读者更容易地、更好地理解与 WS-Security 相关的材料。

1. WS-Security 用例

我们在前面章节中描述的 Web Service 安全性模型的互操作协议需要定义一个一般的流程模型，该流程模型可抽象为下列子流程：

（1）通过 Web Service 定义（例如 WSDL 描述）和服务目录（例如 UDDI）生成和分发服务安全性策略。

（2）生成和分发安全性令牌。

（3）显示消息和服务请求中的令牌。

（4）验证所显示的令牌是否满足所需的策略。验证活动确认安全性令牌中的断言遵循安全性策略以及消息遵循安全性策略。验证活动建立申请者的身份标识，并安全性令牌的发出者是可信任的，可以发出它们已经生成的断言[Anderson 2004b]。仅当成功地完成所有的这些验证活动，并且请求者被授权完成所请求的服务操作时，Web Service 才能处理所请求的操作。

为了更好地理解以上各点，我们使用图 11.22 所描述的订单处理事务场景。在图 11.22 中的场景假定客户端创建一个订购单，并将这个请求发送给销售者（步骤 1）。销售者端的 Web Service 应用核查请求者的信用等级，并验证在产品库存中是否有所订购商品的存货，然后选择运送商并安排订单的装运，最后将账单送交给客户。

在订单管理服务（在销售者端）能够与其他 Web Service 进行交互之前，订单管理服务必须能够向其他 Web Service 认证自己。通过使用可信的第三方发出的数字证书，即可实现这一点。通过提供一个公开密钥管理系统的接口，XKMS 服务器可检索订单管理公司的数字证书（图 11.22 中的步骤 2）。

销售者向信用评级公司的服务提交一个请求，要求核查客户的信誉。订单管理服务（销售者端）知道信用评级服务期望对请求进行数字签名。使用与信用评级公司服务的 WSDL/UDDI 相关联的 WS-Policy，可以指定这一点。当订单管理服务提交它的附有数字签名的的请求时，信用评级服务访问访问 XKMS 服务器，验证请求者的 PKI 证书。假定 XKMS 服务器响应这个证书是有效的，则信用评级服务将要以一个属性断言的方式回应订单管理应用的请求，属性断言陈述了申请者有合适的信用等级（步骤 3）。

在查证客户的信用状况后，订单管理服务需要核实这一订单的细节，即通过核查装运代理所维护的库存/装运数据库来验证相关细节。在允许访问之前，装运代理需要确认销售者已经访问装运服务，并返回一个 SAML 授权断言（步骤 4）。订单管理应用服务向装运服务提交一个订单装运请求，请求的 WS-Security 头部包含步骤 4 发出的授权断言。返回的响应指出了所订购的部分（图 11.22 中的装运细节）与客户所提供的描述吻合，并能够进行装运（步骤 5）。

在确认了有关客户和装运的必要的信息后，对于这个订购，订单管理应用将发出一个具有数字签名的账单。XML-Signature 和 XML-Encryption 可确保这个文档（订单）的完整性和机密性。

对于诸如图 11.23 所描述的端到端的互操作的 Web Service 场景，下面的章节描述了如何使用安全的 SOA 实现这一场景。

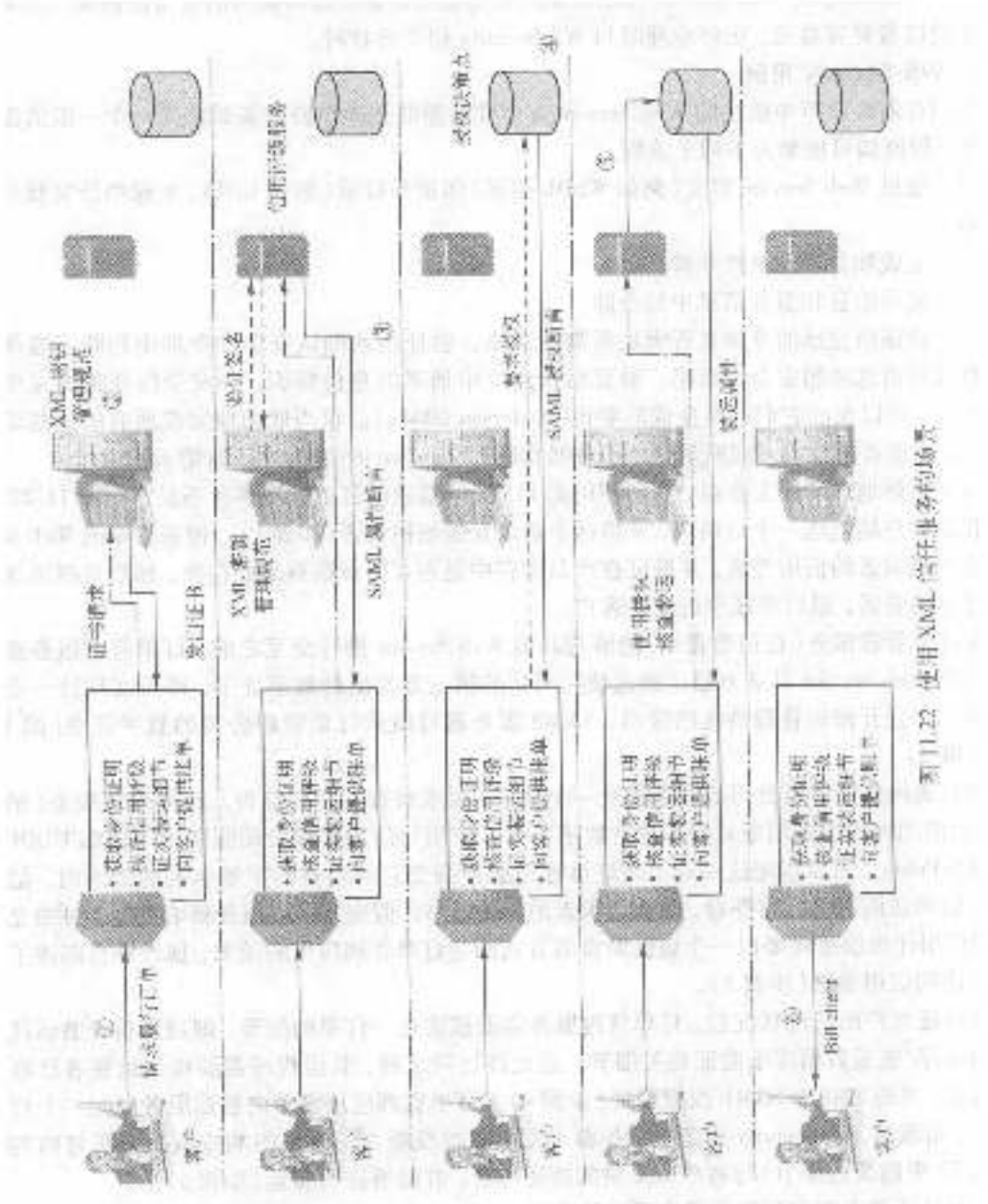


图 11.22 使用 XML 签名服务的场景

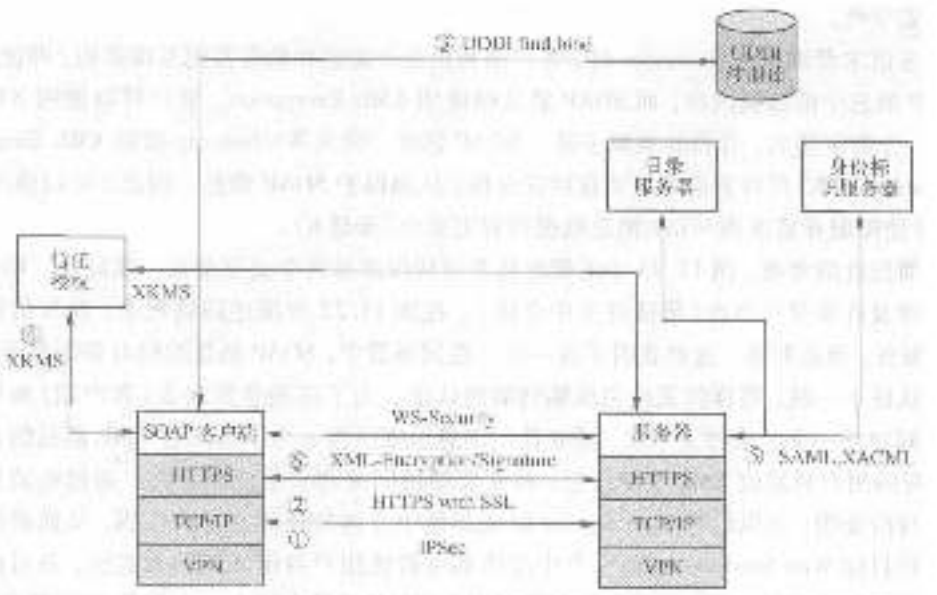


图 11.23　应用 Web Service 安全性解决方案的体系结构

源自：[Lai 2004]

2. 在 SOA 中集成 WS-Security

集成的端到端的、互操作的 Web Service 安全性框架具有很大的价值。企业正在使用各种标准化的协议和新兴的规范，包括进行通信的 SSL、SOA 环境中的 WS-Security（包括 XML-Encryption 和 XML-Signature）、SAML 和 XACML。图 11.23 显示了一个概念体系结构。该体系结构应用了一个集成的 Web Service 安全性基础架构。基于 Web Service 安全性基础架构，可以实现诸如图 11.22 所示的 Web Service 安全性场景。

在 SOA 中，Web Service 调用需要不同的安全性层次来实现端到端的安全性连接。在图 11.23 所描述的概念安全性体系结构中，假定服务请求者（在图 11.22 中发送订购单请求的 SOAP 客户端）通过 VPN 连接到服务提供者，在互联网上使用 IPSec 来实现安全的连接（步骤 1）。这保证了网络层的安全性。在安全体系结构中，基于 SSL/TLS 的 HTTP 提供了传输安全性。我们假定每一个 SOAP 节点都有一个相关的 HTTPS 节点，并且在节点之间，HTTPS 消息可以运载 SOAP 消息。当提供者的站点上使用 SSL 证书时，客户端可以使用 HTTPS，从而使得客户端浏览器和服务器间的连接更安全（步骤 2）。应用了 SSL 的 HTTPS 保证了客户端会话的安全性。使用安全的 HTTPS 连接，客户端可以浏览 UDDI 服务注册库中的不同的 Web Service（业务流程），发现相关的业务流程，以及检索服务端点 URL（步骤 3）。这些操作涉及的是服务发现安全性。

若要调用相关的业务流程，例如图 11.22 中的订购单流程，SOAP 客户端需要提供它的身份证明，从而向所要使用的远程 Web Service 认证它自己。因此，我们假定身份标识提供者是可信机构的一部分，遵循自由联盟（Liberty Alliance）的外部的身份标识提供者将管理这一可信机构。身份标识提供者然后使用 XKMS 提供一个认证服务。通过 XML 密钥管理规范，将能从可信机构获取客户端的密钥（步骤 4）。然后，服务提供者提供一个用户标识和口令，从而认证它自身。一旦成功地进行了认证，身份标识提供者使用 SAML 和 XACML 协议可向服务客户端提供单点登录（步骤 5）。这意味着，服务请求者无须重新登录就能使用其他的 Web Service。这些操作涉及的是

服务协商安全性。

当服务请求者调用 Web Service 时，客户端利用公开密钥和私有密钥基础架构，即使用 XKMS 加密 SOAP 消息中的数据内容，而 SOAP 消息则使用 XML Encryption。客户可以使用 XML Signature 生成一个数字签名，并将其隶属于某一 SOAP 信封。因为 WS-Security 依赖 XML Encryption 和 XML Signature，所以可将它应用于消息层安全性，从而保护 SOAP 消息。因此，可以使用 WS-Security 标准使得服务请求和 SOAP 消息数据内容更安全(步骤 6)。

出于简洁性的考虑，图 11.23 中的概念体系结构仅涉及两个交互节点。实际上，Web Service 编配将要涉及许多交互节点(包括许多中介体)。在图 11.22 所描述的场景中，涉及信用评级服务、库存服务、装运服务，这也表明了这一点。在该场景中，SOAP 消息的路由必须基于多“跳”。每一跳仅认证下一跳，那样将无法完成端到端的认证。为了在消息发起者(客户端)和目标 Web Service 之间建立一个安全性上下文，需要使用消息层安全性。为了传送含 XML 消息的安全性上下文、已有的用户目录或 XML 策略信息，将需要使用网关和安全性服务器。通过在消息中插入终端用户身份证明，可以沿着 Web Service 链或其他中介体传播那些身份证明，从而最终将身份证明传送到目标 Web Service。任何一个中介体都可验证用户身份证明的真实性，并可由于安全性原因而不转发该消息。例如，消息发起者可以在 SOAP 消息中插入一个 XML 数字签名，若消息的内容有任何变化，则当中介体验证签名时，将能发现这一变化。

3. WS-Security 的关键功能部件

正如在本节一开始所讨论的，WS-Security 通过消息完整性、消息机密性和消息的认证来保护消息，从而提高了 SOAP 消息传送的安全性。WS-Security 并不是发明了一种新类型的安全性，而是提供了一个通用的格式，用于适应 SOAP 消息的安全性。可以使用 WS-Security 机制来与许多安全性模型和加密技术进行协作。对于安全性令牌与 SOAP 消息的关联，WS-Security 也提供了一个通用的机制。WS-Security 并不需要特定类型安全性令牌。WS-Security 具有可扩展性，诸如支持多个安全性令牌格式。例如，客户端可以提供身份证明文件，并且这个证明可以有一个特定的经营许可证。

若要使得 SOAP 消息更安全性，需要考虑许多不同类型的威胁，诸如消息被敌对者篡改或读取，敌对者向服务发送消息，以及是否有合适的安全性断言来保障处理的顺利进行等。为了对付这类威胁，WS-Security 通过对消息体、消息的头部或者这两者进行加密和/或数字签名，从而保护了消息的安全性。WS-Security 使用了三个核心元素：安全性令牌、XML Encryption 和 XML Signature。这三个元素构成了 SOAP 安全性头部。基于 XML Signature[Eastlake 2002a]和安全性令牌，可检测出对消息所做的改动，从而提供了消息完整性。完整性机制支持多重签名，多个签名可能是多个 SOAP 角色所签的。完整性机制是可扩展的，支持附加的签名格式。消息机密性利用了 XML Encryption[Eastlake 2002a]和安全性令牌来确保 SOAP 消息的机密性。加密机制用于支持多个 SOAP 角色进行的额外的加密处理和操作。

图 11.24 显示了 WS-Security 消息结构以及与安全性令牌、XML Encryption 和 XML Signature 相关的核心元素。该图显示了 WS-Security 规范将安全性令牌封闭在 SOAP 消息中，并描述了如何使用 XML Signature 和 XML Encryption 来确保 SOAP 头部中的那些令牌的机密性和完整性。

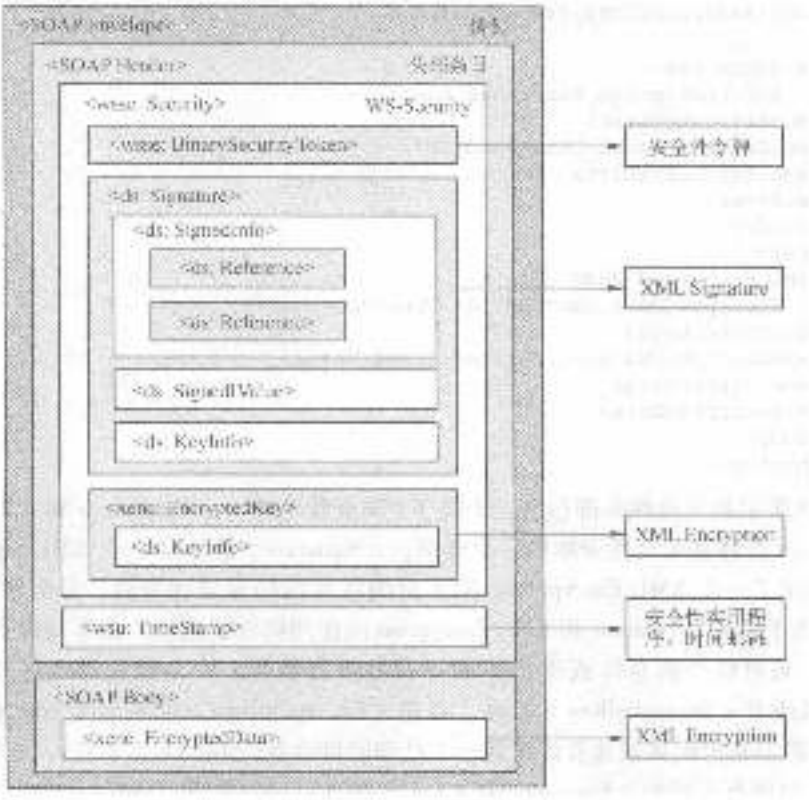


图 11.24　WS-Security 元素和结构

清单 11.7 显示 SOAP 信封中的基本的 WS-Security 结构。WS-Security 定义的最基本的元素是 <Security> 元素。<Security> 元素驻留在 SOAP 头部中。正如简单的举例所示，WS-Security 定义了它自身的命名空间，尽管如此，WS-Security 的创建者遵守已有的标准和技术，而没有指定如何在 SOAP 中使用。

清单 11.7　基本的 WS-Security 头部的结构

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope
   xmlns:env="http://www.w3.org/2003/05/soap-envelope"
   xmlns:wsse=
   "http://docs.oasis-open.org/wss/2004/01/
                        oasis-200401-wsswssecurity-secext-1.0.xsd"
   xmlns:wsu=
   "http://docs.oasis-open.org/wss/2004/01/
                        oasis-200401-wsswssecurity-utility-1.0.xsd"
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
   xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
 <env:Header>
  <wsse:Security>
   <!-- Security Token -->
   <wsse:UsernameToken wsu:Id="...">
      <wsse:Username>...</wsse:Username>
   </wsse:UsernameToken>
   <!-- XML Signature -->
   <ds:Signature>
      ...
```

```
        <ds:Reference URI="#MsgBody">
            ...
      </ds:Signature>
      <!-- XML Encryption Reference List -->
      <xenc:ReferenceList>
      <xenc:DataReference URI="#bodyID">
      </xenc:ReferenceList>
    </wsse:Security>
  </env:Header>
  <env:Body>
   <!-- XML Encrypted Body -->
   <xenc: EncryptedData Id="bodyID" Type="content">
     <xenc:CipherData>
        <xenc:CipherValue>...</xenc:CipherValue>
     </xenc:CipherData>
   </xenc:EncryptedData>
  </env:Body>
</env:Envelope>
```

清单 11.7 所示的安全性头部包含三个孩子：安全性令牌、< Signature > 和 < ReferenceList > 。其中，UserToken 恰好是安全性令牌的一个样例；< Signature > 表示了一个 XML Signature；< ReferenceList > 表示了一个 XML Encryption。除了封闭在其内的安全性令牌，安全性头块也表示了 SOAP 消息中有关 XML Signature 和 XML Encryption 的使用的相关信息。通常情况下，假如包括一个 XML 签名，则对整个消息体或消息体的一部分进行签名。安全性头部也可以包含 < ReferenceList > 元素或者 < EncryptedKey > 元素。若将 < EncryptedData > 元素与 < EncryptedKey > 元素协同使用，可将加密密钥从发起者传送到一个已知的接收者(参见 11.5.2 节)。< ReferenceList > 元素包含一个对所有不同的 < EncryptedData > 元素进行引用的列表。用这一方式，WS-Security 能够读取所有的安全性头部，然后可解密 < EncryptedData > 元素引用的所有数据。清单 11.7 也表明了 SOAP 体通常也被它自身加密。最后，前缀 wsu 表示了安全性实用程序规范的名字。安全性实用程序规范定义了一些元素(包括表示时间邮戳信息的元素，例如时间邮戳的创建和时间邮戳的有效期)、属性和属性组，它们可以被其他规范引用。需要实用全局标识符的规范引用了清单中的 wsu:Id。

WS-Security 中的安全性令牌。在 Web Service 环境中，认证通常涉及身份证明。身份证明既可以嵌入在 SOAP 消息的头部中，也可以嵌入在 SOAP 消息体中。标准的 Web Service 技术使用口令、X.509 证书来标识基于浏览器的客户端，使用 Kerberos 票据来认证客户端。服务请求者也可以使用这些技术来进行认证。对于敏感通信，服务请求者和服务提供者都需要被认证。

WS-Security 能以两种方式处理身份证明的管理。WS-Security 定义了一个专门的元素 < UsernameToken > 。假如 Web Service 正在使用定制的认证，< UsernameToken > 元素可传递用户名和口令。WS-Security 使用另一个专门的元素 < BinarySecurityToken > 来提供二进制认证令牌。WS-Security 中指定的两类认证令牌是 Kerberos 票据和 X.509v3 证书。

图 11.25 表明了一个典型的涉及安全性令牌的消息流[Seely 2002]。一旦 SOAP 客户端请求向 SOAP 消息中添加安全性令牌，安全性令牌服务将返回合适的令牌。令牌可能是 Kerberos、PKI 或者用户名/口令验证服务。返回的令牌可以不基于 Web Service。例如，Kerberos 可以使用操作系统的安全性功能来访问 Kerberos 服务票据准许服务。一旦客户端获取它所需的令牌，客户端就将令牌嵌入在消息中(步骤 2)。紧接着，客户端将使用私有密钥对消息进行签名，私有密钥仅有客户端自己知道(步骤 3)。假如认证很重要，必须对 SOAP 消息进行签名或加密。由于既能够从有效的消息中去除身份标识令牌，也能在攻击者使用的消息中添加身份标识令牌，因此仅在消息

中添加有效的身份标识令牌是不够的。假如接收者(Web Service)对于消息所生成的签名与包含在消息中的签名一致，则接收者能够证实客户端所发送的消息(步骤4)。接收者可有许多方式验证签名。例如，假如客户端对于认证使用一个 < UsernameToken > ，则接收者希望：客户端发送一个散列化的口令，并使用该口令对消息进行签名。最后，客户端能够从 Web Service 中接收响应。

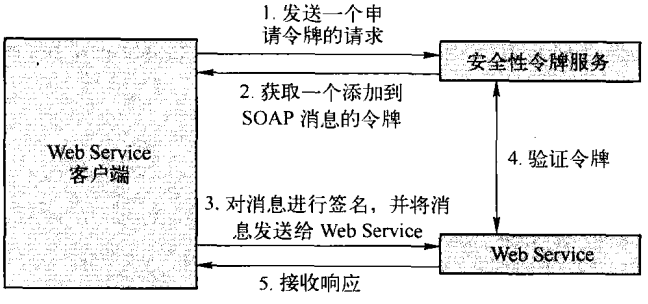


图 11.25　典型的涉及安全性令牌的消息流

在清单 11.8 中的代码中，使用比较通用的 < BinarySecurityToken > 元素发送一个 X.509v3 证书。< BinarySecurityToken > 的 ValueType 属性指出了所发送的是 X.509v3 证书。该证书是使用 Base64 编码表示的。证书本身构成了 < BinarySecurityToken > 元素的内容，可以使用它向一个特定的服务认证一个客户。

清单 11.8　一个嵌入在 < BinarySecurityToken > 中的 X.509 证书

```
<wsse:Security
 xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
 <wsse:BinarySecurityToken
  ValueType="wsse:X509v3"
  EncodingType="wsse:Base64Binary">
   SSphfawHraPle ...
 </wsse:BinarySecurityToken>
</wsse:Security>
```

WS-Security 采用了一个灵活的方式来传送和认证身份标识。然而，虽然两个系统都能遵循 WS-Security，但它们仍然可能不能相互认证。例如，一个系统可能仅支持 Kerberos，而另一个系统仅允许使用基于 X.509 证书的数字签名认证。仅简单地都同意使用 WS-Security 是不够的。对于一些协议，需要精确地规定使用何种安全性令牌。

在 WS-Security 中提供机密性。WS-Security 的另一个主要特性是使用 XML Encryption 来确保机密性。在传送 SOAP 消息之前，先对整个 SOAP 消息或者 SOAP 消息的一部分进行加密。使用 XML Encryption 标准，WS-Security 允许对全部或部分的 SOAP 消息的头部信息、消息体和任何附件进行加密。WS-Security 仅使用标准所定义的三个 XML 元素：< EncryptedData > 、< EncryptedKey > 和 < ReferenceList > (参见 11.5.2 节)。

假如发送者选择使用对称密钥的方式加密 SOAP 信封中的元素或元素的内容，即使用接收者的密钥对相关信息进行加密，并将加密后的信息嵌入在消息中。清单 11.9 显示了这种情况下的 SOAP 消息。为了更容易阅读，在本例中移除了命名空间。

清单 11.9　具有加密信息的 SOAP 消息

```
<env:Envelope>
 <env:Header>
  <wsse:Security>
    <wsse:EncryptedKey>
```

```
        <EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
          <ds:KeyInfo>
            <wsse:SecurityTokenReference>
              <ds:X509IssuerSerial>
                <ds:X509IssuerName>
                  DC=ABC-Corp, DC=com
                </ds:X509IssuerName>
                <ds:X509SerialNumber>12345678</ds:X509SerialNumber>
              </ds:X509IssuerSerial>
            </wsse:SecurityTokenReference>
          </ds:KeyInfo>

        <!-- XML Encryption Reference List -->
        <xenc:ReferenceList>
        <xenc:DataReference URI="#EncryptedBody">
        </xenc:ReferenceList>
      </wsse:EncryptedKey>
    </wsse:Security>
  </env:Header>
  <env:Body>
    <!-- XML Encrypted Body -->
    <xenc: EncryptedData Id=" EncryptedBody" Type="content">
      <xenc:CipherData>
         <xenc:CipherValue>...</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </env:Body>
</env:Envelope>
```

在清单 11.9 中，< EncryptedKey > 元素使用公开密钥来加密一个共享密钥，而这个共享密钥则用来加密 SOAP 体。由于使用了接收者的公开密钥对共享密钥进行包装，因此该技术称作密钥封装(key wrapping)或数字信封(digital enveloping)[Rosenberg 2004]。用于加密的公开密钥位于 < KeyInfo > 块的 < SecurityTokenReference > 元素中。因为并不是所有的令牌都支持通用的引用模式，所以 < SecurityTokenReference > 元素提供了引用安全性令牌的方法。

在 WS-Security 中提供消息完整性。 WS-Security 还有一个主要特性就是使用 XML Signature 来确保消息完整性。正如 11.5.1 节所讨论的，对于数字签名的 XML 文档，XML Signature 提供了一个具体的方法。在 WS-Security 中使用 XML Signature 主要出于两个主要原因[Rosenberg 2004]。第一个目的是用于验证安全性令牌身份证明，诸如 X.509 证书或 SAML 断言。另一个目的是保证消息完整性。消息完整性是指确认消息在发送后是否已被修改。

清单 11.10 显示了 WS-Security 中的 XML Signature 的使用。在本例中，为了便于阅读，我们已经移除了命名空间。< Signature > 元素通常包括数字签名本身以及如何生成数字签名的信息。在 SOAP 中使用 < Signature > 通常包含 < SignedInfo > 元素和 < KeyInfo > 元素。在清单 11.10 中，假定订单处理服务和装运服务共享一个秘密(对称)密钥，诸如口令。而且假定订单处理服务能够对口令应用摘要算法，并获得一个摘要值。然后，订单处理服务能够将那个摘要作为一个对称密钥使用，用来对消息进行加密或签名，并将消息发送给装运服务。装运服务将要使用该共享密钥重新进行摘要计算，并使用摘要值(作为密钥)进行解密和签名验证。

清单 11.10 具有数字签名的 SOAP 消息

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope>
 <env:Header>
  <wsse:Security>
```

```
<wsse:UsernameToken wsu:Id="OrderProcServiceUsernameToken">
   <wsse:Username>ATrustedOrderProcService</wsse:Username>
   <wsse:Nonce>WS3Lhf6RpK...</wsse:Nonce>
   <wsu:Created>2004-09-17T09:00:00Z</wsu:Created>
</wsse: UsernameToken>
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
 <ds:SignedInfo>
  <ds:CanonicalizationMethod
    Algorithm="http://www.w3.org/2001/10/xml-exc-c14N"/>
  <ds:SignatureMethod
    Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
  <ds:Reference URI="#Request4Shipment">
   <ds:DigestMethod
     Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
   <ds:DigestValue>
     aOb4Luuk...
   </ds:DigestValue>
  </ds:Reference>
 </ds:SignedInfo>
 <ds:SignatureValue>
   A9qqIrtE3xZ...
 </ds:SignatureValue>
 <ds:KeyInfo>
  <wsse:SecurityTokenReference>
   <wsse:Reference URI="#OrderProcServiceUsernameToken"/>
  </wsse:SecurityTokenReference>
 </ds:KeyInfo>
</ds:Signature>
</wsse:Security>
</env:Header>
<env:Body>
 <s:ShipOrder
  xmlns:s="http://www.plastics_supply.com/shipping_service/"
  wsu:Id="Request4Shipment">
  <!-- Parameters passed with call -->
  <OrderNumber>PSC0622-X</OrderNumber>
    … … …
 </s:ShipOrder>
</env:Body>
</env:Envelope>
```

在 WS-Security 中，SOAP 头部可以包含多个 XML 签名，并且这些签名有可能有重叠。例如在一个订单处理场景中，订单消息可能需要经过许多中介体。首先，这个消息被传送到一个订单处理系统，订单管理系统在消息中插入一个包含订单标识符的头部，并对头部进行数字签名，即将一个 XML 签名放进该安全性头部。紧接着，将消息转发到装运处理系统。装运处理系统将装运标识符头部插入消息中，并且对订单标识符头部和装运标识符头部进行数字签名。最后，当消息最终到达账单系统时，在向客户提交账单以前，将对这些 XML 签名进行验证。

在一些 SOAP WS-Service 请求的例子中，涉及两个参与者，这两个参与者使用不同类型的安全性令牌以及不同类型的消息交换。这些不同类型的消息交换涉及两个参与者以及一个可信的第三方[Siddiqui 2003c]。

11.6.5　安全性策略的管理

在交互的 Web Service 间使用安全性机制时，必定也存在安全性策略。这些策略阐明了一个特定环境下的具体的安全性需求。对于使用 WS-Security 的 Web Service，服务必须描述它所需的安全性策略。例如，服务可以指定在一个特定的 XML 元素中需要一个签名，并且对该特定的元素也必须进行加密。服务也可以指定加密算法为 DSA，该算法能提供消息的完整性和机密性。

对于验证，服务也可以指定它接受 Kerberos 票据，或者 X.509 证书、数字签名等。处理这些问题意味着需要定义 Web Service 安全性策略。

除了安全性领域之外，策略在其他领域也是有用的，为此开发了 WS-Policy 规范。对于指定各类策略以及将这些策略关联到一个特定的服务，WS-Policy 规范定义了一个通用的方式(参见第 12 章)。若要描述涉及安全性的策略，可使用 WS-SecurityPolicy 规范[Della-Libera 2002]。该规范是一个具体领域的语言，用于表示 WS-Security 中的策略。WS-SecurityPolicy 扩展了 WS-Policy 标准。WS-SecurityPolicy 使得发起 SOAP 交换的组织能够发现目标 Web Service 将能理解何种类型的安全性令牌，类似于 WSDL 描述目标 Web Service 的方式。

WS-SecurityPolicy 定义了一些 XML 元素，可使用这些 XML 元素来指定与安全性相关的策略。由于这些元素允许 Web Service 以明确的方式指定它的策略，因此这些元素被称作断言。断言使得开发者可以指定一个特定主体所需的(或者不能接受的)安全性令牌的类型、签名格式、加密算法。表 11.1 总结了 WS-SecurityPolicy 所定义的各种断言。

表 11.1 WS-SecurityPolicy 定义的安全性策略断言

策略断言	描 述
wsse: SecurityToken	指定了(WS-Security 定义的)安全性令牌的类型
wsse: Integrity	指定了(WS-Security 定义的)签名格式
wsse: Confidentiality	指定了(WS-Security 定义的)加密格式
wsse: Visibility	指定了消息中必须被中介体或端点处理的部分
wsse: SecurityHeader	指定了如何使用 WS-Security 中定义的 < Security > 头部
wsse: MessageAge	指定了在消息声明为过时并被丢弃之前的有效期

WS-SecurityPolicy 规范允许一个 < Policy > 元素包含有关安全性令牌、完整性和机密性的策略。关于策略，< Policy > 也有其他的选项，包括指定相对优先关系的方式。例如，一个 Web Service 能够定义一个 < SecurityToken > ，从而允许客户端能够使用 Kerberos(参见清单 11.11 中的代码片段)或 X.509 证书进行认证，然后指出它首选 Kerberos 方式。WS-SecurityPolicy 提供一种明确地表示对于 Web Service 的安全性需求的方法，使得 Web Service 能够清楚地表示：客户端必须做什么才能访问这个服务。

清单 11.11 安全性策略的样例

```
<wsp:Policy
 xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy"
 xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
 <wsse:SecurityToken wsp:Usage="wsp:Required">
  <TokenType> wsse:Kerberosv5ST</TokenType>
 </wsse:SecurityToken>
</wsp:Policy>
```

在 12.4 节中，我们将要描述 WS-Policy 语言。该章节中将包含涉及 WS-SecurityPolicy 的若干例子。

11.6.6 安全会话的管理

假如一个应用需要与一些其他的应用交换多个 SOAP 消息，那么在进行通信的应用之间创建一些共享的上下文则是很有用的。共享上下文的一个共同用途是：对于上下文中所使用的加密密钥，定义一个生命周期。例如，两个通信方可能想要创建一个对称密钥。然后，在特定的安全性上下文的生命周期中，使用这个密钥加密各通信方之间所交换的信息。

WS-SecureConversation 标准定义了 WS-Security 的扩展，从而可以建立和共享安全性上下文，并可获得一个派生的会话密钥[Anderson 2004a]。对于一个会话的生命周期，在两个或多个通信

方之间可以共享安全性上下文。共享的秘密密钥可用于消息的签名和加密。从一个共享的秘密密钥中，可以派生出一个会话密钥。会话密钥可用于解密会话中发送的消息。安全性上下文可表示为一种新类型的安全性令牌，称作安全性上下文令牌[Cabrera 2005d]。类似 SSL，WS-SecureConversation 使用公开(非对称)加密来设置一个共享的秘密密钥。在此之后，出于效率的考虑，然后将使用共享的密钥(对称)加密技术。

可以按三种不同的方式建立安全性上下文。Web Service 可以根据它们自身的需求选择最合适的方式。第一种方式，安全性令牌服务可以创建一个安全性上下文令牌。发起方将获取该令牌，并将其传播。第二种方式，通信方之一可创建安全性上下文令牌，并将安全性上下文令牌传播到其他通信方。第三种方式，通过一个协商流程可创建并交换安全性上下文令牌。对于请求 WS-Trust 操作的安全性上下文令牌，WS-SecureConversation 定义了一个专门的绑定。

清单 11.12 中的场景描述了：关于安全性上下文令牌的内容，诸如共享的秘密密钥，两方需要协商。在清单 11.12 中，<SecurityContextToken> 指定了一个安全性令牌。该安全性令牌与消息关联，并指向安全性上下文(通过上下文的唯一标识符)。在清单 11.12 中，紧接其后的语句指定了数字签名。在本例中，签名基于安全性上下文(具体地说，是与上下文相关联的秘密密钥)。清单中并没有显示 XML 数字签名的典型内容。

清单 11.12　作为 WS-SecureConversation 一部分的共享秘密密钥安全性上下文

```
<?xml version="1.0" encoding="utf-8"?>
<Env:Envelope>
 <Env:Header>
     ...
  <wsse:Security>
     <wsc:SecurityContextToken wsu:Id="MyID">
        <wsc:Identifier> uuid:...</wsc:Identifier>
     </wsc:SecurityContextToken>
     <ds:Signature>
          ...
        <ds:KeyInfo>
           <wsse:SecurityTokenReference>
              <wsse:Reference URI="#MyID"/>
           </wsse:SecurityTokenReference>
        </ds:KeyInfo>
     </ds:Signature>
  </wsse:Security>
 </Env:Header>
 <Env:Body wsu:Id="MsgBody">
    <Shipment xmlns:tru="http://shipco.com/services/
     orderShipping">
        <Product Name="Injection Quantity="1" Weight= …/>
        <Product Name="Adjustable Worktable" Quantity="1"
         Weight= …/>
    </Shipment>
 </Env:Body>
</Env:Envelope>
```

11.6.7 信任管理

WS-Security 和 WS-SecurityPolicy 使得服务使用者和服务提供者能够以安全和可信的方式直接进行互操作。服务使用者和服务提供者都假定在两个端点使用了相同的服务令牌机制。此外，假定两个端点都位于同一个信任域中。例如，两个节点都信任相同的认证机构。这意味着，当两个不同的信任域使用了不同的密钥技术时，如一个信任域使用 Kerberos 密钥安全性技术，另一个信任域使用 X.509 证书安全性技术，这类规范并没有很好地定义如何从一个信任域向另一个不

同的信任域发送消息。

WS-Trust 规范解决了以上问题。WS-Trust 对 WS-Security 进行了扩展，定义了请求、发出和交换安全性令牌的协议，以及定义了建立和访问信任关系的方式[Anderson 2004b]。尤其，定义了获取、发出、延期和验证安全性令牌的操作。WS-Trust 的另一个特性是代理信任关系。使用这些扩展，应用能够参与安全的通信，与常规的 Web Service 框架(包括 WSDL 服务描述、UDDI < businessService > 和 < bindingTemplates > 、SOAP 消息)进行协作。为了满足不同的安全性需求，WS-Trust 也使用了常规的网络和传输保护机制，诸如 IPSec 或 TLS/SSL。

WS-Trust 规范定义了如何从安全性令牌服务请求和获取安全性令牌，并定义了这些服务如何可以处理信任和信任策略(参阅 11.6.3 节所描述的 Web Service 信任模型以及图 11.21)。基于 WS-Trust，Web Service 请求者能够发送一些消息，通过将安全性令牌与消息进行关联，这些消息表示了请求者所需的一组断言。这些消息还包括了消息的签名，消息的签名显示了拥有令牌(的内容)的证据。假如 Web Service 请求者并没有所需的令牌来验证断言，它将会联系一个合适的安全性令牌服务，并获取所需的令牌。正如 WS-Policy 规范和 WS-SecurityPolicy 规范所描述的：在服务的策略中，服务可以表明它所需的断言和相关的信息。

11.6.8 隐私管理

隐私保护被定义为个体控制其他方收集和使用个人信息的权利[Galbraith 2002]。在 Web Service 环境中，WS-Privacy 是一个颇有远见的规范。该规范还没有发布，它综合采纳了 WS-Policy、WS-Security 和 WS-Trust 中的通信隐私策略，这些隐私策略规定了一个组织在隐私保护方面的策略。部署 Web Service 的组织规定了这些隐私策略。这些隐私策略需要流入的 SOAP 请求包含发送者遵循这些隐私策略的断言。使用 WS-Security 规范可将这些断言封装到可被验证的安全性令牌中。WS-Privacy 解释了如何在 WS-Policy 描述中包含隐私需求。使用 WS-Trust 可评估封装在 SOAP 消息中的有关用户偏好和组织策略的隐私断言。

11.6.9 联邦身份标识的管理

公司的价值网络横跨多个组织、系统应用和业务流程。价值网络由几个不同的组成部分构成，包括企业的客户、交易合作伙伴、供应商和分销商。价值网络中的组织将它们的内部系统扩展到外部企业，从而提供了到客户、交易合作伙伴、供应商的连接性。这类解决方法产生了联邦系统。联邦系统需要跨组织边界进行互操作。利用不同的技术、安全性方法和变成环境，联邦系统可将不同的流程链接在一起。联邦技术主要致力于身份标识。籍此，请求者或者请求者的代表(代理)宣称一个身份标识，而身份标识提供者则核实这一宣称。

在联邦中，每一个成员继续管理它自身的身份标识，但是也能够安全地共享和接受其他成员的身份标识和身份证明。一个联邦身份标识(federated identity)是一个协定集，用于身份标识和它们的属性、身份证明和授权的创建、维护和使用。联邦身份标识支持了基础架构和一些标准。基于基础架构和这些标准，用户身份标识和授权能够在联邦内具有跨自治安全域的可移植性。联邦身份标识基础架构支持跨边界的单点登录、用户资源的动态配置和身份标识属性的共享。单点登录指的是用户仅需要在联邦中登录一次，然后可使用联邦中的不同的服务，而无须重新进行登录。

对于 Web Service 安全性模型，身份标识的联邦标准基于 WS-Federation[Bajaj 2003]。WS-Federation 指定了如何使用 WS-Security、WS-Policy、WS-Trust 和 WS-SecureConversation 构建联邦信任方案。在跨不同信任域的 Web Service 之间，对于身份标识的信任和联邦以及认证信息需要进

行协商。WS-Federation 定义了一个模型和可用于运行这类协商的一组消息。联邦模型对 WS-Trust 模型进行了扩展，描述了身份标识提供者如何充当安全性令牌服务，并描述了如何将属性和假名集成到令牌发放机制中，从而提供联邦身份标识映射机制中。在 WS-Federation 中，属性服务是一个 Web Service，在信任域或联邦中维护关于被代理者、任何系统实体或人的信息（信息）。WS-Federation 假名服务也是一个 Web Service，在信任域或联邦中维护关于被代理者的代替的身份标识信息。WS-Federation 假名服务提供了一个映射机制，可用于实现跨联邦的可信任的身份标识的映射，从而保护了隐私和身份标识。WS-Policy 和 WS-Security 可用于确定使用哪些令牌以及如何从安全性令牌发放服务中申请令牌。WS-Federation 是位于 WS-Policy 和 WS-Trust 之上的一层，表示了如何管理信任关系。

图 11.26 表明了在一个简单的场景中应用 WS-Trust 模型的一种可能的方式。在该场景中，允许一个信任域中的请求者和使用不同的安全性模型的不同的信任域中的资源进行交互[Kaler 2003]。在图中，请求者从它的身份标识提供者（步骤 1）那里获取一个身份标识安全性令牌，并为了所请的资源而向安全性令牌服务显示/证实该令牌（步骤 2）。假如成功，假如存在信任并且授权也被批准，该安全性令牌将向请求者返回一个访问令牌（步骤 4）。即以一个安全性令牌服务中的令牌交换另一个安全性令牌服务中的令牌，一个安全性令牌服务中的令牌被另一个安全性令牌服务标记或交叉认证。

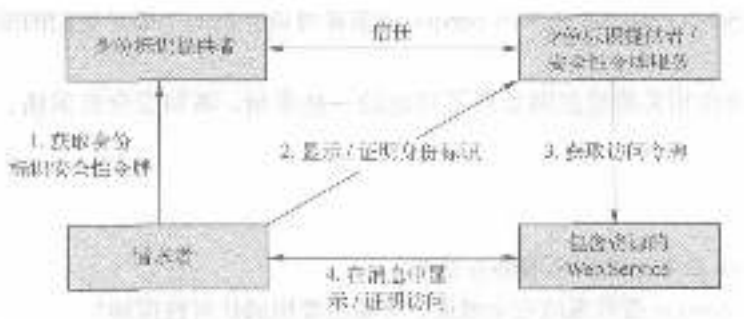


图 11.26　简单的联邦场景

引自[Cumm2006d]

参与联邦需要了解元数据，诸如联邦中服务的策略，可能将 WSDL 描述和 XML 模式。此外，在许多情况下需要有一个实例，用来标识身份标识提供者、安全性令牌服务以及针对一个特定策略的目标（例如，一个 Web Service）的属性/假名服务。为了获取和提供这类信息，WS-Federation 建立在 WS-MetadataExchange 的基础之上。WS-MetadataExchange 定义了 SOAP 请求/响应消息类型。SOAP 请求/响应消息类型检索与 Web Service 端点相关联的元数据的不同类型。更具体地说，端点表示 Web Service 终点或者一个特定目标命名空间，请求/响应对检索与端点关联的 WS-Policy、WSDL 和 XML 模式信息。

11.6.10　授权管理

WS-Privacy 描述了如何声称单个的隐私选择或一组隐私选择。WS-Authority 描述了如何管理数据和授权策略。WS-Authority 处理 Web Service 环境中的授权决策。更具体地说，WS-Authority 描述了如何指定和管理授权数据（访问策略），以及如何在端点解释安全性令牌中的断言。

在本撰写本书时，尚没有发布有关 WS-Authority 的规范。目前所知道的是，WS-Authority 的

目标非常类似于 XACML(参见 11.5.5 节)，并且 WS-Authority 既支持基于控制的授权，也支持基于角色的授权。

11.7 小结

在开放、协作的交易环境中，可以自发地形成各种关联，并可随着时间不断优化这些关联，从而可以满足新出现的业务需求。通过 Web Service，企业可以与它供应商、合作伙伴和客户一起参与到这一开放、协作的交易环境中。因此，Web Service 能够极大地提高企业的生产率。然而，正如许多新技术的应用一样，在安全性方面也有一些严重的问题，带来了一些新的安全风险。为了解决这些问题，Web Service 安全模型引入了一系列既独立又相关关联的规范。这些规范描述了在 Web Service 环境中实现安全性的分层方式。

基于 Web Service 安全性模型，可以对规范进行混合和匹配，从而使得实现者可以仅部署所需要的部分。在这些规范中，对于 Web Service 消息的完整性和机密性，WS-Security 提供了所需的基本元素。WS-Security 还提供了将安全性令牌(例如数字证书或 kerberos 票据)关联到 SOAP 消息的方法。WS-Security 构成了 Web Service 安全性模型的基础。

WS-Security 利用了 Web Service 模型固有的可扩展性。可扩展性是 Web Service 模型的核心。可扩展性建立在一些基础技术之上，诸如 SOAP、WSDL、XML 数字签名、XML 加密和 SSL/TLS 等。这使得 Web Service 提供者和 Web Service 请求者可以开发一个满足他们的安全性需求的解决方案。

其他的与安全性相关的规范则处理了其他的一些事情，诸如安全性策略、信任、隐私和授权等。

复习题

- Web Service 最常见的安全威胁有哪些?
- 对于 Web Service 最常见的安全威胁，有哪些常用的针对性措施?
- 什么是网络层安全性? 什么是应用层安全性?
- 简要描述防火墙的最常见的体系结构。
- 保护互联网通信的安全性的最常用的技术有哪些?
- 对称密钥加密和非对称密钥加密的不同点是什么?
- 使用最广的应用层安全协议是什么?
- 简要描述 XML 安全性服务和标准 XML 加密、XML 签名、SAML 和 XACML。
- 解释如何在 Web Service 环境中使用这些 XML 标准。
- Web Service 安全性和信任模型的目的是什么?
- 简要描述 WS-Security 组件。
- 在 WS-Security 环境中如何实现消息完整性和机密性?

练习

11.1 在 XML 中，开发商业客户的一个简单的财务决算，包含客户名、商业账户结余和信用卡账户，并需要支付信用卡中的未偿还余额。使用该财务决算来说明如何在 XML 签名中创建一个封内签名。

11.2　对于前一个练习，开发一个封外和分离签名解决方案。

11.3　基于类似于清单 3.6 中定义的订购单模式，显示客户如何能够机密订购单的敏感细节，诸如所订购的产品和数量，或者信用卡信息。

11.4　假定一个具有高级会员身份的主体将要访问一个采取了安全措施的 URL，使用 XACML 定义一个代表该主体的简单服务读请求。

11.5　将一个基于 Web Service 并包含 X.509 证书的 SOAP 消息定义为进行数字签名和加密的二进制安全令牌。该 SOAP 消息体将包含一个 < EncryptedData > 元素，从而能够将加密密钥从发起者传输到接收者。

11.6　定义一个 SOAP 消息，该消息需要综合 WS-Security 的三个最重要的要素：完整性、机密性和身份证明。

参考文献

[Ahmed 2001] K. Ahmed *et al.*, "XML Metadata", Wrox Press, 2001.

[Aldrich 2002] S. E. Aldrich, "Anatomy of Web Services", Patricia Seybold Group, Inc. 2002, available at:
www.psgroup.com.

[Alexander 2004] J. Alexander *et al.*, "Web Services Transfer (WS-Transfer)", September 2004, available at:
http://www.w3.org/Submission/2006/SUBM-WS-Transfer-20060315/.

[Allen 2001] P. Allen, *Realizing e-Business with Components*, Addison-Wesley, 2001.

[Alonso 2004] G. Alonso *et al.*, *Web Services: Concepts, Architectures and Applications*, Springer, 2004.

[Anagol-Subbaro 2005] A. Anagol-Subbaro, *J2EE Web Services on BEA WebLogic*, Prentice Hall, 2005.

[Anderson 2004a] S. Anderson *et al.*, "Web Services Secure Conversation Language (WS-SecureConversation)", Version 1.1, May 2004, available at:
ftp://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf.

[Anderson 2004b] S. Anderson *et al.* "Web Services Trust Language (WS-Trust)", Version 1.1, May 2004, available at:
ftp://www6.software.ibm.com/software/developer/library/ws-trust.pdf.

[Andrews 2003] T. Andrews *et al.* (eds.), "Business Process Execution Language for Web Services", May 2003, available at:
http://www.ibm.com/developerworks/library/ws-bpel.

[Andrieux 2005] A. Andrieux *et al.*, "Web Services Agreement Specification (WS-Agreement)", Technical Report, Grid Resource Allocation Agreement Protocol (GRAAP) WG, September 2005, available at:
http://www.gridforum.org/Public_Comment_Docs/Documents/Oct-2005/WS-AgreementSpecificationDraft050920.pdf.

[Antoniou 2004] G. Antoniou, F. van Harmelen, *A Semantic Web Primer*, MIT Press, 2004.

[Arkin 2001] A. Arkin, "Business Process Modelling Language", March 2001, bpmi.org.

[Arkin 2002] A. Arkin, "Business Process Modelling Language (BPML) specification", BPMI, June 2002, available at:
http://www.bpmi.org/index.esp.

[Arsanjani 2004] A. Arsanjani, "Service-oriented Modeling and Architecture", IBM developerWorks, Novemeber 2004, available at:
http://www-106.ibm.com/developerworks/library/ws-soa-design1/.

[Atkinson 2002] C. Atkinson *et al.*, *Component-based Product Line Engineering with UML*, Addison-Wesley, 2002.

[Austin 2004] D. Austin *et al.* (eds.), "Web Services Choreography Requirements", W3C Working Draft, March 2004, available at:
http://www.w3.org/TR/ws-chor-reqs/.

[Bachmann 2000] F. Bachmann *et al.*, "Technical Concepts of Component-Based Software Engineering", Technical Report, Carnegie-Mellon University, CMU/SEI-2000-TR-008 ESC-TR-2000-007, 2nd edition, May 2000.

[Bajaj 2003] S. Bajaj *et al.* (eds.), "Web Services Federation Language (WS-Federation) Version 1.0", July 2003, available at:
http://www-128.ibm.com/developerworks/library/specification/ws-fed/.

[Bajaj 2006a] S. Bajaj *et al.*, "Web Services Policy Framework (WS-Policy) Version 1.2", March 2006, available at:
http://xml.coverpages.org/ws-policy200603.pdf.

[Bajaj 2006b] S. Bajaj *et al.*, "Web Services Policy Attachment (WS-PolicyAttachment)", March 2006, available at:
http://specs.xmlsoap.org/ws/2004/09/policy/ws-policyattachment.pdf.

[Ballinger 2006] K. Ballinger *et al.*, "Web Services Metadata Exchange (WS-MetadataExchange), Version 1.1", August 2006, available at:
http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-metadataexchange.pdf.

[Balzer 2004] Y. Balzer, "Improve your SOA project plans", IBM developerWorks, July 2004, available at:
http://www-106.ibm.com/developerworks/library/ws-improvesoa/.

[Banks 2003] T. Banks (ed.), "Open Grid Service Infrastructure Primer", GWD-I (draft-ggf-ogsi-gridserviceprimer-1), Open Grid Services Infrastructure (OGSI), June 2003, available at:
https://forge.gridforum.org/projects/ogsi-wg.

[Bass 2001] L. Bass, M. Klein, G. Moreno, "Applicability of General Scenarios to the Architecture Tradeoff Analysis Method", Technical Report CMU/SEI-2001-TR-014, Software Engineering Institute, Carnegie-Mellon University, 2001.

[Bass 2003] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 2nd edition, Addison-Wesley, 2003.

[Batres 2005] S. Batres *et al.*, "Web Services Reliable Messaging Policy Assertion (WS-RM Policy) Version 1.0", February 2005, available at:
http://www.oasis-open.org/committees/download.php/16889/.

[Bean 2003] J. Bean, *XML for Data Architects: Designing for Re-use and Integration*, Morgan Kaufmann, 2003.

[Beckett 2004] D. Beckett (ed.), "RDF/XML Syntax Specification (Revised)", W3C Recommendation, February 2004, available at:
http://www.w3.org/TR/rdf-syntax-grammar/.

[Bellwood 2003] T. Bellwood *et al.*, "Universal Description, Discovery and Integration specification (UDDI) 3.0", December 2003, available at:
http://uddi.org/pubs/uddi-v3.00.

[Bennett 2002] V. Bennett, A. Capella, "Location-based Services", IBM developerWorks, March 2002, available at:
http://www-106.ibm.com/developerworks/library/i-lbs/.

[Berners-Lee 1998] T. Berners-Lee, R. Fielding, L. Masinter, "RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax", August 1998, available at:
http://www.ietf.org/rfc/rfc2396.txt.

[Bieberstein 2005] N. Bieberstein *et al.*, "Impact of Service-Oriented Architecture on Enterprise Systems, Organizational Structures, and Individuals", IBM Systems Journal, vol. 44, no. 4, pp. 691–708, 2005.

[Bieberstein 2006] N. Bieberstein *et al.*, *Service-Oriented Architecture (SOA) Compass*, IBM Press, 2006.

[Bilorusets 2005] R. Bilorusets *et al.*, "Web Services Reliable Messaging Protocol (WS-ReliableMessaging)", February 2005, available at:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/wsrmspecindex.asp.

[Bloch 2003] B. Bloch *et al.* (eds.), "Web Services Business Process Execution Language", OASIS Open Inc. Working Draft 01, October 2003, available at:
http://www.oasis-open.org/apps/org/workgroup/wsbpel/.

[Bloomberg 2004] J. Bloomberg, "Events vs. Services", ZapThink White Paper, October 2004, available at:
www.zapthink.com.

[Bosworth 2004] A. Bosworth *et al.*, "Web Services Addressing (WS-Addressing)", August 2004, available at:
http://msdn.microsoft.com/ws/2004/08/ws-addressing/.

[Box 2003] D. Box *et al.*, "Reliable Message Delivery in a Web Services World: A Proposed Architecture and Roadmap", Joint IBM Corporation and Microsoft Corporation White Paper, March 2003, available at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-rm-exec-summary.asp.

[BRCommunity 2005] "A Brief History of the Business Rule Approach", Business Rule Community 2005, available at:
http://www.BRCommunity.com.

[Brickley 2004] D. Brickley, R. V. Guha (eds.), "RDF Vocabulary Description Language 1.0: RDF Schema", W3C Recommendation, February 2004, available at:
http://www.w3.org/TR/rdf-schema/.

[Brittenham 2001] P. Brittenham, F. Curbera, D. Ehnebuske, and S. Graham, "Understanding WSDL in a UDDI Registry", IBM developerWorks, September 2001, available at:
http://www-106.ibm.com/developersworks/library/.

[Brown 2005] A. Brown *et al.*, "SOA Development Using the IBM Rational Software Development Platform: A Practical Guide", Rational Software, September 2005.

[Bunting 2003a] D. Bunting *et al.*, "Web Services Composite Application Framework (WS-CAF)", July 2003, available at:
http://developers.sun.com/techtopics/webservices/wscaf/index.html.

[Bunting 2003b] D. Bunting *et al.*, "Web Service Context (WS-CTX)", July 2003, available at:
http://developers.sun.com/techtopics/webservices/wscaf/wsctx.pdf.

[Bunting 2003c] D. Bunting *et al.*, "Web Service Coordination Framework (WS-CF)", July 2003, available at:
http://developers.sun.com/techtopics/webservices/wscaf/wscf.pdf.

[Bunting 2003d] D. Bunting *et al.*, "Web Services Transaction Management (WS-TXM)", July 2003, available at:
http://developers.sun.com/techtopics/webservices/wscaf/wstmx.pdf.

[Cabrera 2005a] L. F. Cabrera *et al.*, "Web Service Coordination: (WS-Coordination)", August 2005, available at:
http://schemas.xmlsoap.org/ws/2004/10/coord.

[Cabrera 2005b] L. F. Cabrera *et al.*, "Web Services Atomic Transaction: (WS-AtomicTransaction)", August 2005, available at:
http://schemas.xmlsoap.org/ws/2004/10/at.

[Cabrera 2005c] L. F. Cabrera *et al.*, "Web Services Business Activity Framework (WS-BusinessActivity)", August 2005, available at:
http://schemas.xmlsoap.org/ws/2004/10/ba.

[Cabrera 2005d] F. Cabrera, C. Kurt, *Web Services Architecture and its Specifications*, Microsoft Press, 2005.

[Candadai 2004] A. Candadai, "A Dynamic Implementation Framework for SOA-based Applications", *Web Logic Developers Journal: WLDJ*, September/October 2004.

[Cantor 2004] S. Cantor *et al.* (eds.), "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS, Committee Draft 03, December 2004, available at:
http://www.oasis-open.org/committees/documents.php?wg_abbrev=security.

[Carzaniga 2000] A. Carzaniga, D. S. Rosenblum, A. L. Wolf, "Content-based Addressing and Routing: A General Model and its Application", Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[Carzaniga 2001] A. Carzaniga, D. S. Rosenblum, A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service", ACM Transactions on Computer Systems, vol. 19, no. 3, pp. 332–83, August 2001.

[Case 1999] J. D. Case *et al.*, "Introduction to Version 3 of the Internet-standard Network Management Framework", Internet Engineering Task Force (IETF), RFC 2570, April 1999, available at:
www.rfc-editor.org/rfc/rfc2570.txt.

[Cauldwell 2001] P. Cauldwell *et al.*, *XML Web Services*, Wrox Press, 2001.

[Channabasavaiah 2003] K. Channabasavaiah, K. Holley, E. M. Tuggle, Jr., "Migrating to a Service-Oriented Architecture", IBM developerWorks, December 2003, available at:
http://www-106.ibm.com/developerworks/library/ws-migratesoa/.

[Chappell 2004] D. A. Chappell, *Enterprise Service Bus*, O'Reilley, 2004.

[Chappell 2005a] D. Chappell, "ESB Myth Busters: Clarity of definition for a growing phenomenon", *Web Services Journal*, pp. 22–6, February 2005.

[Chappell 2005b] D. Chappell, Private communication, April 2005.

[Chatterjee 2004] S. Chatterjee, J. Webber, *Developing Enterprise Web Services*, Prentice Hall, 2004.

[Cheesman 2001] J. Cheesman, J. Daniels, *UML Components: A Simple Process for Specifying Component-based Software*, Addison-Wesley, 2001.

[Clark 2001] J. Clark *et al.* (eds.), "ebXML Business Process Specification Schema: Version 1.01", OASIS, May 2001, available at:
www.ebxml.org/specs/ebBPSS.pdf.

[Colan 2004] M. Colan, "Service-Oriented Architecture expands the vision of Web services, Part 2", IBM developerWorks, April 2004, available at:
http://www-106.ibm.com/developerworks/library/ws-soaintro2/.

[Cole 2002] G. Cole, "SNMP vs. WBEM – The Future of Systems Management", available at:
http://www.wbem.co.uk.

[Colgrave 2003a] J. Colgrave, "A new approach to UDDI and WSDL: Introduction to the new OASIS UDDI WSDL", IBM developerWorks, August 2003, available at:
http://www-106.ibm.com/developersworks/library/.

[Colgrave 2003b] J. Colgrave, "A new approach to UDDI and WSDL, Part 2: Queries supported by the new OASIS UDDI WSDL Technical Note", IBM developerWorks, September 2003, available at:
http://www-106.ibm.com/developersworks/library/.

[Colgrave 2004] J. Colgrave, K. Januszewski, "Using WSDL in a UDDI Registry, Version 2.0.2 – Technical Note Using WSDL in a UDDI Registry, Version 2.0.2", OASIS, June 2004, avaialble at:
http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v202-20040631.htm.

[Comella-Dorda 2000] S. Comella-Dorda *et al.*, "A Survey of Legacy System Modernization Approaches", Technical Note CMU/SEI-2000-TN-003, Software Engineering Institute, Carnegie-Mellon University, April 2000, available at:
http://www.sei.cmu.edu/publications/pubWeb.html.

[Coulouris 2001] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems: Concepts and Design*, 3rd edition, Addison-Wesley, 2001.

[Culbertson 2001] R. Culbertson, C. Brown, G. Cobb, *Rapid Testing*, Prentice Hall, 2001.

[Curbera 2003] P. Curbera *et al.*, "Web services, the next step: A framework for robust service composition", Communications of the ACM, Special Section on Service-Oriented Computing, M. P. Papazoglou and D. Georgakopoulos (eds.), October 2003.

[Czajkowski 2004] K. Czajkowski *et al.*, "The WS-Resource Framework, Version 1.0", March 2004, available at:
www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf.

[Czajkowski 2005] K. Czajkowski, I. Foster, C. Kesselman, "Agreement-Based Resource Management", *Proceedings of the IEEE*, vol. 93, no. 3, March 2005.

[DAML] Darpa Agent Markup Language, available at:
http://www.daml.org.

[Davis 2004] M. Davis *et al.*, "WS-I Security Scenarios Document Status", Web Services Interoperability Organisation Working Group Draft Version 0.15, February 2004, available at:
http://www.ws-i.org.

[Della-Libera 2002] G. Della-Libera *et al.*, "Web Services Security Policy (WS-SecurityPolicy)", IBM, Microsoft, RSA Security, VeriSign, Draft, December 2002, available at:
http://www.ibm.com/developerworks/library/ws-secpol/.

[Dertouzos 1999] M. L. Dertouzos, "The future of computing", Scientific American, August 1999.

[Duftler 2002] M. Duftler, R. Khalaf, "Business Process with BPEL4WS: Learning BPEL4WS, Part 3", IBM developmentworks, October 2002, available at:
http://www-106.ibm.com/developerworks/Webservices/library/.

[Eastlake 2002a] D. Eastlake, J. Reagle, D. Solo (eds.), "XML-Signature Syntax and Processing", W3C Recommendation, February 2002, available at:
http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

[Eastlake 2002b] D. Eastlake, J. Reagle (eds.), "XML Encryption Syntax and Processing", W3C Recommendation, December 2002, available at:
http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/.

[Elmagarmid 1992] A. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.

[Endrei 2004] M. Endrei *et al.*, "Patterns: Service-Oriented Architecture and Web Services", IBM Redbooks SG24-6303-00, April 2004, available at:
http://publib-b.boulder.ibm.com/Redbooks.nsf/redbooks/.

[Eswaran 1976] K. Eswaran *et al.*, "The Notion of Consistency and Predicate Locks in Database Systems", *Communications of the ACM*, vol. 19, no. 11, pp. 624–33, November 1976.

[Ferreira 2002] L. Ferreira, V. Berestis, "Fundamentals of Grid Computing", IBM Redbooks paper REDP-3613-00, November 2002, available at:
http://www.redbooks.ibm.com/redpapers/pdf/redp3613.pdf.

[Ferreira 2004] L. Ferreira *et al.*, "Grid Services Programming and Application Enablement", IBM Redbooks, May 2004, available at:
ibm.com/redbooks.

[Ford 1997] W. Ford, M. S. Baum, *Secure Electronic Commerce: Building the infrastructure for digital signatures and encryption*, Prentice Hall, 1997.

[Foster 2002a] I. Foster, "What is the Grid? A Three Point Checklist", *GRIDToday*, July 2002, available at:
http://www.globus.org/alliance/publications/papers.php#Overview%20Papers.

[Foster 2002b] I. Foster *et al.*, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", Globus Project, 2002, available at:
www.globus.org/research/papers/ogsa.pdf.

[Foster 2004] I. Foster *et al.* (eds.), "Modeling Stateful Resources with Web Services Version 1.1", March 2004, available at:
http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf.

[Foster 2005] I. Foster, H. Kishimoto, A. Savva (eds.), "The Open Grid Services Architecture, Version 1.0", GFD-I.030, available at:
http://forge.gridforum.org/projects/ogsa-wg, January 2005.

[Galbraith 2002] B. Galbraith *et al.*, *Professional Web Services Security*, Wrox Press, 2002.

[Ganci 2006] J. Ganci *et al.*, *Patterns: SOA Foundation Service Creation Scenario*, IBM Redbooks, September 2006.

[Garcia-Molina 2002] H. Garcia-Molina, J. D. Ullman, J. Widom, *Database Systems*, Prentice Hall, 2002.

[Garcia-Molina 1987] H. Garcia-Molina, K. Salem, Proceedings of the ACM SIG on Management of Data, 1987 Annual Conference, San Francisco, May 1987, ACM Press, pp. 249–59.

[Gardner 2002] J. Gardner, Z. Rendon, *XSLT and XPath*, Prentice Hall, 2002.

[Goldfarb 2001] C. Goldfarb, P. Prescod, *The XML Handbook*, 3rd edition, Prentice Hall 2001.

[Graham 2004a] S. Graham *et al.*, *Building Web Services with Java*, SAMS Publishing, 2004.

[Graham 2004b] S. Graham, P. Niblett (eds.), "Publish-Subscribe Notification for Web Services", March 2004, available at:
http://docs.oasis-open.org/committees/dowload.php/6661/WSNpubsub-1-0.pdf.

[Graham 2004c] S. Graham, J. Treadwell (eds.), "Web Services Resource Properties 1.2 (WS-ResourceProperties)", OASIS Working Draft 04, June 2004, available at:
http://docs.oasis-open.org/wsrf/2004/11/wsrf-WS-ResourceProperties-1.2-draft-05.pdf.

[Graham 2004d] S. Graham, P. Murray (eds.), "Web Services Base Notification (WS-BaseNotification 1.2)", OASIS Working Draft 03, June 2004, available at:
http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf.

[Graham 2005] S. Graham *et al.*, *Building Web Services with Java*, 2nd edition, SAMS Publishing, 2005.

[Gray 1993] J. Gray, A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[Gruber 1993] T. R.Gruber, "Toward Principles for the Design of Ontologies used for Knowledge Sharing", KSL-93-04. Knowledge Systems Laboratory, Stanford University, 1993.

[Gudgin 2003] M. Gudgin *et al.* (eds.), "SOAP 1.2 Part 1: Messaging Framework", Martin Gudgin, W3C Recommendation, June 2003, available at: **http://www.w3.org/TR/2003/REC-soap12-part1-20030624/**.

[Hall-Gailey 2004] J. Hall-Gailey, *Understanding Web Services Specifications and the WSE*, Microsoft Press, 2004.

[Hallam-Baker 2004] P. Hallam-Baker *et al.* (eds.), "Web Services Security X.509 Certificate Token Profile", OASIS Standard 200401, March 2004, available at: **http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0**.

[Handfield 2002] R. B. Handfield, E. L. Nichols, *Supply Chain Redesign*, Prentice Hall, 2002.

[Harmon 2003a] P. Harmon, "Analyzing Activities", *Business Process Trends*, vol. 1, no. 4, pp. 1–12, April 2003.

[Harmon 2003b] P. Harmon, "Second Generation Business Process Methodologies", *Business Process Trends*, vol. 1, no. 5, pp. 1–13, May 2003.

[Herzum 2000] P. Herzum, O. Sims, *Business Component Factory*, John Wiley & Sons, 2000.

[Holley 2006] K. Holley, J. Palistrant, S. Graham, "Effective SOA Governance", IBM OnDemand Business, March 2006, available at: **http://www-306.ibm.com/software/solutions/soa/gov/lifecycle/**.

[Hughes 2004] J. Hughes *et al.*, "SAML Technical Overview", OASIS-SSTC, July 2004, available at: **http://www.oasis-open.org/committees/security/**.

[IBM 2004] IBM Corporation, "An architectural blueprint for autonomic computing", IBM Autonomic Computing White Paper, October 2004, available at: **http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf**.

[Irani 2002] R. Irani, "An Introduction to ebxml", in *Web Services Business Strategies and Architectures*, P. Fletcher, M. Waterhouse (eds.), ExpertPress, 2002.

[Iwasa 2004] K. Iwasa *et al.* (eds.), "Web Services Reliable Messaging TC WS-Reliability 1.1", OASIS Standard, OASIS, November 2004, available at: **http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf**.

[Jaenicke 2004] C. Jaenicke, "Canonical Message Formats: Avoiding the Pitfalls", WebLogic Journal, September/October 2004.

[Jeal 2003] D. Jeal *et al.*, "Mobile Web Services Roadmap", Microsoft Corporation and Vodafone Group Services, 2003.

[Jeston 2006] J. Jeston, J. Nelis, "Business Process Management: Practical Guidelines to Successful Implementations", Butterworth–Heinemann, 2006.

[Jin 2002] L. J. Jin, V. Machiraju, A. Sahai, "Analysis on Service Level Agreement of Web Services", Technical Report HPL-2002-180, Software Technology Laboratory, HP Laboratories Palo Alto, June 2002, available at: **www.hpl.hp.com/techreports/2002/HPL-2002-180.pdf**.

[Johnston 2005] S. Johnston, "Modelling Service-oriented Solutions", IBM developerWorks, July 2005, available at: **http://www-128.ibm.com/developerworks/rational/library/johnston/**.

[Joseph 2004] J. Joseph, M. Ernset, C. Fellenstein, "Evolution of grid computing architecture and grid adoption models", *IBM Systems Journal*, vol. 43, no. 4, pp. 624–45, 2004.

[Juric 2006] M. Juric, B. Matthew, P. Sarang, *Business Process Execution Language for Web Services*, 2nd edition, PACKT Publishing, 2006.

[Kakadia 2002] D. Kakadia *et al.*, "Enterprise Management Systems: Architectures and Standards", Sun Microsystems, April 2002, available at: **http://www.sun.com/blueprints/0402/ems1.pdf**.

[Kaler 2003] C. Kaler, A. Nadalin (eds.), "Web Services Federation Language (WS-Federation) Version 1.0", July 2003, available at: **http://www-106.ibm.com/developerworks/Webservices/library/ws-fed/**.

[Kaufman 1995] C. Kaufman, R. Perlman, M. Speciner, "Network Security, Private Communication in a Public World", Prentice Hall, 1995.

[Kavantzas 2004] N. Kavantzas *et al.*, "Web Services Choreography Description Language 1.0", Editor's Draft, April 2004, available at:

http://lists.w3.org/Archives/Public/www-archive/2004Apr/att-0004/cdl_v1-editors-apr03-2004-pdf.pdf.

[Keen 2004] M. Keen *et al.*, "Patterns: Implementing an SOA Using an Enterprise Service Bus", IBM Redbooks SG24-6346-00, July 2004, available at: **http://publib-b.boulder.ibm.com/Redbooks.nsf/redbooks/.**

[Kifer 2005] M. Kifer, A. Bernstein, P. M. Lewis, *Database Systems: An Application-Oriented Approach*, 2nd edition, Addison-Wesley, 2005.

[Klyne 2004] F. Klyne, J. J. Carroll, "Resource Description Framework (RDF): Concepts and Abstract Syntax", W3C Recommendation, February 2004, available at: **http://www.w3.org/TR/rdf-concepts/.**

[Kreger 2005a] H. Kreger, "A Little Wisdom about WSDM", IBM developerWorks, available at: **http://www-128.ibm.com/developerworks/library/ws-wisdom.**

[Kreger 2005b] H. Kreger *et al.*, "Management Using Web Services: A Proposed Architecture and Roadmap", IBM, HP, and Computer Associates, June 2005, available at: **www-128.ibm.com/developerworks/library/specification/ws-mroadmap.**

[Kruchten 2004] P. Kruchten, *Rational Unified Process – An Introduction*, 3rd edition, Addison-Wesley, 2004.

[Kurose 2003] J. F. Kurose, K. W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, 2nd edition, Addison-Wesley, 2003.

[Lai 2004] R. Lai, *J2EE Platform Web Services*, Prentice Hall, 2004.

[Leymann 2000] F. Leymann, D. Roller, *Production Workflow*, Prentice Hall, 2000.

[Leymann 2001] F. Leymann, "Web Services Flow Language", May 25, 2001, available at: **http://xml.coverpages.org/wsft.html.**

[Leymann 2002] F. Leymann, D. Roller, "A Quick Overview of BPEL4WS", IBM developerWorks, August 2002, available at: **http://www-106.ibm.com/developerworks/.**

[Linthicum 2003] D. Linthicum, *Next Generation Application Integration: From Simple Information to Web Services*, Addison-Wesley, 2003.

[Little 2003a] M. Little, J. Webber, "Introducing WS-CAF – More than just transactions", *Web Services Journal*, vol. 3, no. 12, December 2003.

[Little 2003b] M. Little, J. Webber, "Introducing WS-Transaction", *Web Services Journal*, vol. 3, no. 6, pp. 28–33, June 2003.

[Little 2004] M. Little, J. Maron, G. Pavlik, *Java Transaction Processing*, Prentice Hall, 2004.

[Lubinsky 2001] B. Lubinsky, M. Farrel, "Enterprise Architecture and J2EE", *eAI Journal*, November 2001.

[Ludwig 2007] H. Ludwig, "WS-Agreement Concepts and Use-Agreement-Based Service-Oriented Architectures", in *Readings in Service Oriented Computing*, D. Georgakopoulos, M.P. Papazoglou (eds.), MIT Press, 2007.

[Maguire 2005] T. Maguire, D. Snelling (eds.), "Web Services Service Group 1.2 (WS-ServiceGroup)", OASIS Working Draft 04, February 2005, available at: **http://docs.oasis-open.org/wsrf/2005/03/wsrf-WS-ServiceGroup-1.2-draft-04.pdf.**

[Malan 2002] R. Malan, D. Bredemeyer, "Software Architecture: Central Concerns, Key Decisions", 2002, available at: **http://www.ruthmalan.com/.**

[Manes 2004] A. T. Manes, *The Role of Web Services Registries in Service Oriented Architectures*, Burton Group, November 2004.

[Mani 2002] A. Mani, A. Nagarajan, "Understanding quality of service for Web services", IBM developerWorks, January 2002, available at: **http://www-106.ibm.com/developerworks/library/ws-quality.html.**

[Manola 2004] F. Manola, E. Miller (eds.), "RDF Primer", W3C Recommendation, February 2004, available at: **http://www.w3.org/TR/rdf-primer/.**

[Marks 2006] E. A. Marks, M. Bell, *Service-oriented Architecture: A Planning and Implementation Guide for Business and Technology*, John Wiley & Sons, 2006.

[Masud 2003] S. Masud, "RosettaNet Based Web Services", IBM developerWorks, July 2003,

available at:
http://www-128.ibm.com/developerworks/webservices/library/ws-rose1/.

[McGoveran 2004] D. McGoveran, "An Introduction to BPM and BPMS", *Business Integration Journal*, April 2004.

[McGuiness 2004] D. McGuiness, F. van Harmelen, "OWL Web Ontology Language Overview", W3C Recommendation, February 2004, available at:
http://www.w3.org/TR/owl-features.

[McKee 2001] B. McKee, D. Ehnebuske, "Providing a Taxonomy for Use in UDDI Version 2.0", UDDI.org, June 2001.

[Mehta 2003] T. Mehta, "Adaptive Web Services Management Solutions", *Enterprise Networks and Servers*, vol. 17, no. 5, May 2003, available at:
http://www.enterprisenetworksandservers.com/monthly/toc.php?4.

[Mitra 2003] N. Mitra (ed.), "SOAP Version 1.2 Part 0: Primer 1.1", W3C Recommendation, June 2003 available at:
http://www.w3.org/TR/soap12-part0/.

[Mitra 2005] T. Mitra, "A Case for SOA Governance", IBM developerWorks, August 2005, available at:
http://www-106.ibm.com/developerworks/webservices/library/ws-soa-govern/index.html.

[Monson-Haefel 2001] R. Monson-Haefel, D. A. Chappell, *Java Message Service*, O'Reilley, 2001.

[Monson-Haefel 2004] R. Monson-Haefel, *J2EE Web Services*, Addison-Wesley, 2004.

[Monzillo 2002] R. Monzillo, "Security", in *Designing Enterprise Applications with the J2EE Platform*, 2nd edition, I. Singh, B. Stearns, M. Johnson (eds.), Addison-Wesley 2002.

[Moss 1985] E. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[Moss 1997] J. Moss, "Understanding TCP/IP", PC Network Advisor, no. 87, September 1997.

[Murray 2002] J. Murray, "Designing Manageable Applications", Web Developer's Journal, October 2002, available at:
http://www.webdevelopersjournal.com/articles/design_man_app/.

[Mysore 2003] S. Mysore, "Securing Web Services – Concepts, Standards, and Requirements", Sun Microsystems, October 2003, available at:
sun.com/software.

[Nadalin 2004] A. Nadalin *et al.* (eds.), "Web Services Security: SOAP Message Security 1.0 (WS-Security)", OASIS Standard 200401, OASIS Open, March 2004, available at:
http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.

[Newcomer 2005] E. Newcomer, G. Lomow, *Understanding SOA with Web Services*, Addison-Wesley, 2005.

[Niblett 2005] P. Niblett, S. Graham, "Events and Service-Oriented Architecture: The OASIS Web Services Notification Specifications", *IBM Systems Journal*, vol. 44, no. 4, pp. 869–86, 2005.

[Nokia 2004] Nokia Corporation and Sun Microsystems, "Identity Federation and Web Services – technical use cases for mobile operators", Nokia and Sun Microsystems White Paper, 2004, available at:
www.nokia.com.

[Nolan 2004] P. Nolan, "Understand WS-Policy Processing: Explore Intersection, Merge, and Normalization in WSPolicy", IBM developerWorks, December 2004, available at:
http://www-106.ibm.com/developerworks/library/ws-policy.html.

[OASIS 2004] OASIS: Organization for the Advancement of Structured Integration Standards, "Introduction to UDDI: Important Features and Functional Concepts", October 2004, available at:
http://xml.coverpages.org/UDDI-TechnicalWhitePaperOct28.pdf.

[Ogbuji 2000] S. Ogbuji, "Supercharging WSDL with RDF: Managing structured Web service metadata", IBM developerWorks, November 2000, available at:
http://www-4.ibm.com/software/developer/library/ws-rdf/index.html.

[OMA 2004] Open Mobile Alliance, "OMA Web Services Enabler (OWSER): Core Specifications Version 1.0", OMA-OWSER-Core-Specification-V1_0-20040715-A, July 2004.

[O'Neill 2003] M. O'Neill *et al.*, *Web Services Security*, McGraw-Hill Osborne, 2003.

[Open Group 2004] "SLA Management Handbook: Enterprise Perspective", version 2, volume 4,

issue 0.8, November 2004, available at:
www.opengroup.org/pubs/catalog/go45.htm.

[Owen 2004] M. Owen, J. Raj, "BPMN and Business Process Management: An Introduction to the New Business Process Modelling Standard", Business Process Trends, March 2004, available at: **www.bptrends.com.**

[Pallickara 2004] S. Pallickara, G. Fox, "An Analysis of Notification Related Specifications for Web/Grid applications", Community Grids Laboratory, Indiana University, 2004, available at: **http://grids.ucs.indiana.edu/ptliupages/publications/WSNotifyEventComparison.pdf.**

[Papazoglou 2003] M. P. Papazoglou, G. Georgakapoulos, "Introduction to the Special Issue about Service-Oriented Computing", *Communications of the ACM*, vol. 46, no. 10, pp. 24–8, October 2003.

[Papazoglou 2005a] M. P. Papazoglou, "Extending the Service Oriented Architecture", *Business Integration Journal*, February 2005.

[Papazoglou 2005b] M. P. Papazoglou, W. J. van den Heuvel, "Web Services Management: A Survey", *IEEE Internet Computing*, November/December 2005.

[Papazoglou 2006] M. P. Papazoglou, P. M. A. Ribbers, *e-Business: Organizational and Technical Foundations*, John Wiley & Sons, 2006.

[Pashtan 2005] A. Pashtan, *Mobile Web Services*, Cambridge University Press, 2005.

[Patil 2003] S. Patil, E. Newcomer, "ebXML and Web Services", *IEEE Internet Computing*, May 2003.

[Paulk 1993] M. Paulk *et al.*, "Capability Maturity Model for Software", version 1.1, Software Engineering Institute, Pittsburgh, Technical Report SE-93 TR-024, 1993.

[Pelz 2003] C. Pelz, "Web Services Orchestration and Choreography", *Web Services Journal*, July 2003.

[Pilz 2003] G. Pilz, "A New World of Web Services Security", *Web Services Journal*, March 2003.

[Potts 2003] M. Potts, I. Sedukhin, H. Kreger, "Web Services Manageability – Concepts (WS-Manageability)", IBM, Computer Associates International, Inc., Talking Blocks, Inc, September 2003, available at:
www3.ca.com/Files/SupportingPieces/web_service_manageability_concepts.pdf.

[Proctor 2003] S. Proctor, "A Brief Introduction to XACML", March 2003, available at:
http://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html.

[Rana 2004] R. Rana, S. Kumar, "Service on Demand Portals: A Primer on Federated Portals", *Web Logic Developers Journal: WLDJ*, September/October 2004.

[Robinson 2004] R. Robinson, "Understand Enterprise Service Bus Scenarios and Solutions in Service-Oriented Architecture", IBM developerWorks, June 2004, available at:
http://www-106.ibm.com/developerworks/library/ws-esbscen/.

[Roch 2002] E. Roch, "Application Integration: Business and Technology Trends", *eAI Journal*, August 2002.

[Rodriguez 2001] A. Rodriguez, "TCP/IP Tutorial and Technical Overview", IBM Redbooks, August 2001, available at:
ibm.com/redbooks/.

[Rosenberg 2004] J. Rosenberg, D. Remy, "Securing Web Services with WS-Security", SAMS Publishing, 2004.

[Rosenblum 1997] D. S. Rosenblum, A. L. Wolf, "A design framework for Internet-scale event observation and notification", Proceedings of the Sixth European Software Engineering Conference, Lecture Notes in Computer Science 1301, Springer, 1997.

[RosettaNet 2003] "RosettaNet and Web Services", 2003, available at:
http://www.rosettanet.org/RosettaNet/Doc/0/IP0QL046K55KFBSJ60M9TQCPB3/RosettaNet+Web+ServicesFINAL+.pdf.

[RosettaNet 2004] RosettaNet Implementation Guide, "Cluster 3: Order Management Segment A: Quote and Order Entry PIPs 3A4, 3A7, 3A8, 3A9", April 2004, available at:
www.rosettanet.org/usersguides/.

[Royce 1998] W. Royce, "Software Project Management: A Unified Framework", Addison-Wesley, 1998.

[Shapiro 2002] R. Sahapiro, "A Comparison of XPDL, BPML and BPEL4WS", March 2002, available at:

xml.coverpages.org/Shapiro-XPDL.pdf.

[Satyanarayanan 2001] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges", *IEEE Personal Communications*, August 2001.

[Schlosser 1999] M. Schlosser, "IBM Application Framework for e-Business: Security", November 1999, available at:
http://www-4.ibm.com/software/developer/library/security/index.html.

[Schmelzer 2002] R. Schmelzer *et al.*, *XML and Web Services*, SAMS Publishing, 2002.

[Scribner 2000] K. Scribner, M. C. Stiver, *Understanding SOAP*, SAMS Publishing, 2000.

[Scribner 2002] K. Scribner, M. C. Stiver, *Applied SOAP*, SAMS Publishing, 2002.

[Seacord 2001] R. C. Seacord *et al.*, "Legacy System Modernization Strategies", Technical Report, CMU/SEI-2001-TR-025, ESC-TR-2001-025, Software Engineering Institute, Carnegie-Mellon University, July 2001, available at:
http://www.sei.cmu.edu/publications/pubWeb.html.

[Sedukhin 2005] I. Sedukhin, "Web Services Distributed Management: Management of Web Services (WSDM-MOWS) 1.0", OASIS-Standard, March 2005, available at:
http://docs.oasis-open.org/wsdm/2004/12/wsdm-mows-1.0.pdf.

[Seely 2002] S. Seely, "Understanding WS-Security", Microsoft Corporation, October 2002, available at:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwssecur/html/understw.asp.

[SHA-1] "FIPS PUB 180-1 Secure Hash Standard", US Department of Commerce National Institute of Standards and Technology, available at:
http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt.

[Shaikh 2004] H. Saikh, "Managing the Life Cycle of WS-Resources", IBM developerWorks, May 2004, available at:
http://www-106.ibm.com/developerworks/library/ws-statefulws2/.

[Sharma 2001] P. Sharma, B. Stearns, T. Ng, *J2EE Connector Architecture and Enterprise Application Integration*, Addison-Wesley, 2001.

[Shaw 1996] M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Siddiqui 2001] B. Siddiqui, "Deploying Web services with WSDL", available at:
http://www-106.ibm.com/developerworks/library/ws-intwsdl/.

[Siddiqui 2003a] B. Siddiqui, "Web Services Security, Part 2", O'Reilly XML.com, April 2003, available at:
http://Webservices.xml.com/lpt/a/ws/2003/04/01/security.html.

[Siddiqui 2003b] B. Siddiqui, "Web Services Security, Part 3", O'Reilly XML.com, May 2003, available at:
http://Webservices.xml.com/lpt/a/ws/2003/05/13/security.html.

[Siddiqui 2003c] B. Siddiqui, "Web Services Security, Part 4", O'Reilly XML.com, July 2003, available at:
http://Webservices.xml.com/lpt/a/ws/2003/07/22/security.html.

[Silver 2003] B. Silver, "BPM 2.0: Process Without Programming", Bruce Silver Associates. September 2003, available at:
www.brsilver.com.

[Simon 2001] E. Simon, P. Madsen, C. Adams, "An Introduction to XML Digital Signatures", XML.com August 2001, available at:
http://www.xml.com/pub/a/2001/08/08/xmldsig.html.

[Singh 2004] J. Singh *et al.*, *Designing Web Services with the J2EE 1.4 Platform*, Addison-Wesley, 2004.

[Skonnard 2002] A. Skonnard, M. Gudgin, *Essential XML Quick Reference*, Addison-Wesley, 2002.

[Skonnard 2003] A. Skonnard, "Understanding WS-Policy", Web Services Policy Framework (WS-Policy), Version 1.01, June 2003, available at:
http://www.ibm.com/developerworks/library/ws-polfram/.

[Slee 1997] C. Slee, M. Slovin, "Legacy Asset Management", Information Systems Management.

Winter 1997.

[Soh 1995] C. Soh, M. L. Markus, "How IT Creates Business Value: A Theory Synthesis", Proceedings of 16th International Conference on Information Systems, Amsterdam, December 1995.

[Soni 1995] D. Soni, R. Nord, C. Hofmeister, "Software Architectures in Industrial Applications", in Proceedings of the 17th International Conference on Software Engineering, IEEE CS Press, pp. 196–207, September 1995.

[Srinivasan 2004] L. Srinivasan, T. Banks (eds.), "Web Services Resource Lifetime (WS-ResourceLifetime)", OASIS Working Draft 04, November 2004, available at:
http://docs.oasis-open.org/wsrf/2004/11/wsrf-WS-ResourceLifetime-1.2-draft-04.pdf.

[Steel 2006] C. Steel, R. Nagappan, R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE™, Web Services, and Identity Management*, Prentice Hall, 2006.

[Sun Microsystems 2003] Sun Microsystems, "The Sun Common Mobility Architecture: Delivering Mobile Services – A Technical Overview", 2003, available at:
www.sun.com/wireless.

[Supply-Chain Council 2005] Supply-Chain Council, "Supply-Chain Operations Reference-model: SCOR Version 7.0 Overview", 2005, available at:
www.supply-chain.org.

[Tennison 2001] J. Tennison, *XSLT and XPath*, M & T Books, 2001.

[Tuecke 2003] S. Tuecke *et al.*, "Open Grid Service Infrastructure (OGSI)", Global Grid Forum OGSI-WG, GFD-R-P.15, June 2003.

[Ullman 1988] J. Ullman, *Principles of Database and Knowledge Based Systems*, Computer Science Press, 1998.

[Ulrich 2002] W. Ulrich, *Legacy Systems – Transformation Strategies*, Prentice Hall, 2002.

[Unger 2003] J. Unger, M. Haynos, "A visual tour of Open Grid Services Architecture", IBM developerWorks, August 2003, available at:
http://www-106.ibm.com/developerworks/grid/library/gr-visual/.

[Valentine 2002] C. Valentine, L. Dykes, E. Tittel, *XML Schemas*, Sybex, 2002.

[Vambenepe 2004] W. Vambenepe (ed.), "Web Services Topics (WS-Topics) v. 1.2", OASIS Working Draft 01, July 2004, available at:
http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf.

[Vambenepe 2005a] W. Vambenepe (ed.), "Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1", *OASIS Standard*, March 2005, available at:
http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part1-1.0.pdf.

[Vambenepe 2005b] W. Vambenepe (ed.), "Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 2", *OASIS Standard*, March 2005, available at:
http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part2-1.0.pdf.

[VeriSign 2003a] VeriSign Inc., "VeriSign Digital Trust Services", April 2003, available at:
www.verisign.com.

[VeriSign 2003b] VeriSign Inc., "Managed PKI: Securing Your Business Applications", White Paper, May 2003, available at:
www.verisign.com.

[Veryard 2001] R. Veryard, *The Component-Based Business: Plug and Play*, Springer, 2001.

[Vinovski 2004] S. Vinoski, "More Web Services Notifications", *IEEE Internet*, May/June 2004.

[vonHalle 2002] B. von Halle, *Business Rules Applied*, John Wiley & Sons, 2002.

[Wahli 2004] U. Wahli *et al.*, "WebSphere Version 5.1 Application Developer 5.1.1 Web Services Handbook", IBM Redbooks, February 2004, available at:
ibm.com/redbooks.

[Walmsley 2002] P. Walmsley, *Definitive XML Schema*, Prentice Hall, 2002.

[WAP Forum 2002] WAP Forum, "Wireless Application Protocol WAP 2.0", Technical White Paper, January 2002, available at:
www.wapforum.org.

[Webber 2001] J. Webber *et al.*, "Making Web Services Work", *Application Development Advisor*, pp. 68–71, November/December 2001.

[Webber 2003a] J. Webber, M. Little, "Introducing WS-Coordination", *Web Services Journal*, vol. 3, no. 5, pp. 12–16, May 2003.

[Webber 2003b] J. Webber, M. Little, "Introducing WS-CAF: More than just Transactions", *Web Services Journal*, vol. 3, no. 12, pp. 52–5, December 2003.

[Webber 2004] D. R. Webber *et al.*, "The Benefits of ebXML for e-Business", International Conference of XML (XML'04), August 2004.

[Weerawarana 2005] S. Weerawarana *et al.*, *Web Services Platform Architecture*, Prentice Hall, 2005.

[WfMC 1999] Workflow Management Coalition, "Terminology & Glossary", Document Number WFMC-TC-1011, February 1999.

[White 2004] S. A. White, "Introduction to BPMN", *Business Process Trends*, July, 2004, available at:
www.bptrends.com.

[Whitehead 2002] K. Whitehead, *Component-based Development*, Addison-Wesley, 2002.

[Wilkes 2005] M. Wilkes, "Modernizing Application Integration with Service Oriented Architecture", CBDI Report, CBDI Forum 2005, available at:
www.cbdiforum.com.

[WS-Roadmap 2002] "Security in a Web Services World: A Proposed Architecture and Roadmap", IBM developerWorks, April 2002, available at:
http://www-128.ibm.com/developerworks/library/specification/ws-secmap.

[Zimmermann 2003] O. Zimmermann *et al.*, *Perspectives on Web Services*, Springer, 2003.

[Zimmermann 2004] O. Zimmermann, P. Krogdahl, C. Gee, "Elements of Service-oriented Analysis and Design", IBM developerWorks, June 2004, available at:
http://www-106.ibm.com/developerworks/library/ws-soad1/.

延伸阅读



TCP/IP详解 卷1：协议

作 者：[美] W. Richard Stevens
译 者：[illegible]
审 校：[illegible]
英文版：7-111-09505-7 定价：39.00
中文版：7-111-07566-8 定价：45.00

TCP/IP详解 卷2：实现

作 者：[美] Gary R. Wright, W. Richard Stevens
译 者：[illegible]
审 校：[illegible]

TCP/IP详解 卷3：TCP事务协议、HTTP、NNTP和UNIX域协议

[美] W. Richard Stevens 著
[illegible]
■国际知名Unix网络专家杰作
■中文版编辑八年，精编精装中



计算机网络：自顶向下方法（原书第4版）

作 者：James F. Kurose
译 者：[illegible]
书 号：978-7-111-10605-7
定 价：88.00元
■全球上百所大学和学院采用，[illegible]



计算机网络与互联网

作 者：[illegible]
书 号：978-7-111-24725-4
定 价：45.00元
■本书提要：[illegible]



计算机网络实验教程——从原理到实验

作 者：[illegible]
书 号：978-7-111-20932-8
定 价：46.00元
■本书[illegible]

延伸阅读



网络处理器与网络系统设计
作　者：Douglas E. Comer
译　者：张建忠 陈智华 等
书　号：978-7-111-14362-0
定　价：39.00元



计算机安全：原理与实践
作　者：William Stallings 等
译　者：贾春福 刘春波 高敏芬
书　号：978-7-111-24149-2
定　价：68.00元



自动网络管理系统
作　者：Douglas E. Comer
译　者：吴英 等
中文版：978-7-111-25393-5
定　价：38.00元
英文版：978-7-111-23515-6
定　价：48.00元



普通高等教育"十一五"国家级规划教材
教育部2007年度普通高等教育精品教材
计算机网络实验教程——从原理到实践
作　者：庞鸣 常强林 吊振军
书　号：978-7-111-20882-8
定　价：45.00元

教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的_____教材。

机械工业出版社华章公司为了进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息！感谢合作！

个人资料（请用正楷完整填写）

教师姓名		□先生 □女士	出生年月		职务		职称：□教授 □副教授 □讲师 □助教 □其他
学校			学院			系别	
联系电话	办公： 宅电： 移动：			联系地址及邮编			
				E-mail			
学历		毕业院校		国外进修及讲学经历			
研究领域							

主讲课程	现用教材名	作者及出版社	共同授课教师	教材满意度
课程： □专 □本 □研 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换
课程： □专 □本 □研 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换

样书申请			
已出版著作		已出版译作	
是否愿意从事翻译/著作工作 □是 □否		方向	
意见和建议			

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）

地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037

电 话：(010) 68353079 88378995 传真：(010)68995260

E-mail:hzedu@hzbook.com markerting@hzbook.com 图书详情可登录http://www.hzbook.com网站查询