/THEORY/IN/PRACTICE

SOA in Practice

The Art of Distributed System Design

SOA 实践指南

分布式系统设计的艺术

Nicolai M. Josuttis 著
程桦 译

O'REILLY®

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

本书展示了实现面向服务的架构（SOA）切实可行的规律，使得SOA不再只是镜花水月，可望而不可及。本书基于作者在一家领先行业的公司中全面推广SOA的第一手经验，解释了SOA如何简化大型应用的创建和维护。不管你的项目是包含一套巨大的、基于Web Services的组件集，还是需要将老系统和更现代化的业务流程连接起来，本书都阐明了SOA如何满足你的需要。

多年来，SOA一直只是个美好的愿景。通过描述在实践中实现和运行SOA所碰到的问题，本书让SOA走入凡尘，真正能被大家所用。通过介绍SOA的方方面面，分析典型的使用模式，并探究松耦合如何帮助构筑更强大的应用，《SOA实践指南》提出了一个框架，帮助你决定何时，以及如何利用SOA获益。在本书中，你将：

- **直接聚焦于真实的技术和部署，而不是仅仅关注若干标准**
- **在应用SOA解决方案之前，先检查业务问题，确定哪些业务问题适用SOA方法**
- **找到建立解决方案的清晰路径，而不是陷入不断变化的Web Services细节泥潭之中**
- **分享资深SOA系统架构师的实战经验**

“本书描述的原则和经验，在使T-Mobile公司成功建立SOA上扮演了重要角色，公司的SOA每天要处理超过1000万个服务调用。”

——Steffen Roehn博士，T-Mobile国际公司执行委员会成员（CIO）

“在过分炒作的SOA世界中，Nicolai Josuttis做出了杰出的贡献：一部基于第一手经验、经过认真推敲的、深刻洞察SOA本质的作品。就鼓励SOA实践者采用切实可行的方法而言，本书是一个重要的里程碑。”

——John Schmidt，集成联盟主席

“所有CIO、IT主管和IT计划经理都应该看看这本书。”

——Richard Mark Soley博士，对象管理组织主席和CEO、SOA联盟执行董事

Nicolai M. Josuttis，著有《C++标准库》和《C++模板》，两书都由Addison-Wesley出版。他的职务是系统架构师和技术经理。最近两年，他在一家国际电话公司中全面推广SOA。Nicolai在几个会议上办过SOA讲座，多年来他一直在发表关于这个主题的演讲。

责任编辑：陈元玉
项目管理：梁晶

O'REILLY®
www.oreilly.com

Broadview®
WWW.BROADVIEW.COM.CN
PHEI
www.phei.com.cn

图书分类：软件工程 > SOA

网上订购：www.dearbook.com.cn
第二书店 · 第一服务

ISBN 978-7-121-07041-9
9 787121 070419 >
定价：59.80元

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

O'REILLY®

SOA 实践指南

——分布式系统设计的艺术

SOA in Practice: The Art of Distributed System Design

Nicolai M. Josuttis 著
程 桦 译

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内容简介

本书展示了实现面向服务的架构（SOA）切实可行的规律，使得 SOA 不再只是镜花水月，可望而不可及。本书基于作者在一家领先行业的公司中全面推广 SOA 的第一手经验，解释了 SOA 如何简化大型应用的创建和维护。不管你的项目是包含一套巨大的、基于 Web Services 的组件集，还是需要将老系统和更现代化的业务流程连接起来，本书都阐明了 SOA 如何满足你的需要。通过描述在实践中实现和运行 SOA 所碰到的问题，本书让 SOA 走入凡尘，真正能被大家所用。

0-596-52955-4 SOA in Practice: The Art of Distributed System Design © 2007 by O'Reilly Media, Inc. Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2008. Authorized translation of the English edition, 2007 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O'Reilly Media, Inc.授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2008-2552

图书在版编目（CIP）数据

SOA 实践指南：分布式系统设计的艺术 /（美）乔苏蒂斯（Josuttis,N.M.）著；程桦译.
北京：电子工业出版社，2008.9
书名原文:SOA in Practice:The Art of Distributed System Design
ISBN 978-7-121-07041-9
I. S… II.①乔…②程… III.互联网络－网络服务器 IV.TP368.5

中国版本图书馆 CIP 数据核字（2008）第 100437 号

责任编辑：陈元玉
项目管理：梁　晶
印　　刷：北京智力达印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：22　字数：400 千字
印　　次：2008 年 9 月第 1 次印刷
定　　价：59.80 元

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

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

译者序

从 20 世纪 60 年代的软件危机开始，人们就在和计算机软件、系统的复杂性作斗争。结构化编程、面向对象编程和组件是编程范式方面的努力；客户机—服务器架构、三层架构乃至 N 层架构是架构方面的进化。SOA 是计算机业界在同软件、系统复杂性斗争上的新尝试。本书多次提到，SOA 和之前的尝试相比有一些重要的不同，因此，成功处理复杂性的希望也更大。

- 不同之一为 SOA 接受异质。这个转变产生的效果正如敏捷编程接受“需求不断变更”产生的效果一样，从抗拒到接受，观念上转变的一小步，实践中将形成巨大的差异。
- 不同之二为 SOA 接受变化，甚至必须在变化中才能产生出自己的 SOA；作者多次提到“你买不来 SOA”，强调 SOA 是必须根据自己的情况逐步演化出的一个 IT 战略。

这些不同之处与其说是技术上的，不如说是观念上的。事实上，观念的变化最为强有力，新观念或者会产生与过去迥异的方法论、开发过程和架构，或者会激发人们从新的角度、用新的方式来组织老技术和老方法。

然而，如果对 SOA 的研究只停留在纯观念、纯理论上，则 SOA 将始终是美丽而遥不可及的海市蜃楼。“理论是苍白的，而生活之树常青”。只有既抓住观念和本质，又深入一定程度细节，才能“纲举目张”和“翔实可行”兼得。

《SOA 实践指南》正是这样组织内容的。作者先讨论 SOA 的动机、定义、特性、分类等基本概念，建立 SOA 的纲；然后用了大量篇幅讨论 SOA 实践的各个方面——技术细

节如服务的识别、SOA 安全、SOA 性能、消息交换模式；管理细节如 SOA 版本划分、SOA 开发过程、服务管理、SOA 建立和监管过程等。对实践的讨论基于作者自己在大型跨国公司中实现 SOA 的经验，言之有物，绝非纸上谈兵。

但是，由于 SOA 的特殊性质，本书并不是从一个具体的例子出发（比如，假设一家虚拟的公司）讲述 SOA 怎样从头到尾建立起来的全过程。不这样做的原因是：具体到一个SOA项目的讨论会涉及过多领域相关的内容，特殊性太强，反而无助于全面阐释SOA。因此，如果诸位程序员希望能够看了本书后就可以马上动手写漂亮的“面向 SOA 的代码”的话，恐怕这个愿望无法满足。本书一再强调的就是，建立 SOA 是个渐进、迭代和逐步演化的过程，最好的老师就是自己的实践。《SOA 实践指南》更大意义上是一本关于方法论的书。对所有人来说，本书目的是通过足够的细节帮助你建立正确的观念。如此一来，当面对具体的 SOA 项目时，你就既不至于只能泛泛而谈，无从着手；也不至于只见树木，不见森林，一下陷入到具体细节而不能自拔。

对公司的 CEO、CIO 来说，本书可以帮助你了解 SOA 的成本、收益、对公司组织结构的影响、对公司的战略意义；对分析师、架构师、项目经理来说，本书可以帮助你建立 SOA 的完整概念，进行恰当的架构决策、过程决策，回避重大的陷阱；对开发一线的程序员来说，本书可以帮助你理解 SOA 的精髓，了解 SOA 涉及的各种技术考量，只有在建立了这些认识后，程序员才能在真正理解了 SOA 的基础上，编写出漂亮的“面向 SOA 的代码”。

翻译书是个艰苦的劳动过程，当然，这同时也是个愉快的学习过程。虽然我力求做到准确传达原著的意思，但由于水平所限，仍然难免有理解错误和表达不清之处，敬请广大读者批评指正。

程　桦

2008 年 5 月于北京

译者介绍

程桦，IT 软件业从业 10 年，经历了 IT 业的疯狂、萧条，并回归正轨，目前任软件架构师。

曾翻译《微软的秘密》。经常看看 IT 访谈视频，有好的就忍不住翻译一些。

喜欢与同事分享技术上的点滴所得，思考 IT 之“道”。同时相信，在技术领域，“非宁静无以致远”。

前言

Preface

我从未计划成为一个 SOA 专家。我在一个部门负责一个团队，被选中来使用新的、面向服务的架构方法与别的部门和业务单位的系统进行通信。由一个跨部门的 SOA 团队提供 SOA 概念，其内容包括上千页的文档、几个框架和程序库，以及一些相应的过程草图。作为一个有自己 IT 力量的业务单位，我们自己必须做的所有事情是，用这些解决方案建立 SOA。

一旦我们开始项目，大家很快就发现事情进行得不像原来预想的那样顺利。在与 SOA 方法打交道的同时，我不但学到了 SOA 是什么，还学到了理论和实践的区别，对此 Laurence Peter “Yogi” Berra 作了如下的完美表述：

> 从理论上讲，理论和实践是一致的。但在实践中则并非如此。

实际上，由于我对中央 SOA 团队提供的东西的抱怨是如此之多，以致我最终被派给了一项任务——将这些东西整理清楚。给我的任务描述是：要保证我的经理不会再听到业务单位对 SOA 的怨言。

于是，我们修正错误的概念和支离破碎的过程、自动化手工操作、对战略性决策施以影响、向别人提供支持，并且（最后而并非最不重要的）给其他人讲授 SOA 背后的概念和思想，以及特定的架构决策和政策后面的理由与动机。

现在，我们建立的 SOA 景观（译注 1）对分布在本地、全国和国际若干系统上的流程提供着支持。这个系统由上百个服务支撑着每天数以百万计的服务调用。

得益于这次经验，以及我在许多其他公司（处于非常不同的扩张阶段）观察到的 SOA 方法和项目，我开始搞 SOA 咨询、做评审、进行演讲和开培训课程。在这个过程中，我对 SOA 的不同风格，以及处理 SOA 的不同方法渐渐明了，除此之外，我还了解了所有必须回答的问题，以及教授 SOA 的最好方法。

我仍然每天都在学习，但是我相信，针对在实践中建立 SOA 和处理其特性这项任务，到目前为止自己学到的知识能帮助你找到一条恰当而成功的道路。

SOA 已经成了一个主要的范式，并且，它现在对不同的人意味着不同的东西。我将讨论集中在“SOA 作为一个战略，为分布式业务处理提供了支持”上面。在这个意义上，SOA 是一个虽然可能由 IT 驱动，却总是对业务产生总体影响的战略。

本书将提出若干概念，但焦点放在概念的实际应用上。这是为什么本书非常吻合 O'Reilly 的“实践中的理论”系列的一个原因。

读本书前你应了解什么

What You Should Know Before Reading This Book

作为本书的读者，你应该对计算机科学和编程有一般的了解。在大型和分布式系统上的经验对阅读本书有帮助，但并非必需，因为本书涵盖了所有这些概念（从 SOA 的角度）。

本书的结构

Structure of the Book

本书被设计为从头至尾顺序阅读。我们将从 SOA 一般概念开始，逐步前进到更高级的话题。交叉引用将帮助你找到包含在其他章节中的解释和更深入的细节，索引能帮你找到关于特定话题和术语的信息和讨论情况。

本书前半部分讨论基础：

译注 1：原著对应“景观”一词的英文为“landscape”。landscape 有“风景”、“景观”、“山水画”等意思。其中的“景观”，在辞海的解释中有一条是“特定区域的概念，专指自然地理区划中起始的或基本的区域单位，是发生上相对一致和形态结构同一的区域，即自然地理区”。由于大型系统自然划定出一个特定区域，其中的各个系统既同时发挥作用，又结合到一个统一的态结构下（如 SOA），比较符合这条解释，所以，本书将 landscape 译为“景观”。

第 1 章：动机

本章探讨你为什么想要在大型分布式系统环境下使用 SOA，探讨了 SOA 是如何出现的，讲述了魔术总线的传说，给出了 SOA 的简要概览。

第 2 章：SOA

本章研究和整理了 SOA 的不同定义及其主要特性。

第 3 章：服务

本章研究和整理了 SOA 涉及的许多种服务的定义。

第 4 章：松耦合

本章介绍和讨论了松耦合，这是个在 SOA 中，以及建立大型分布式系统时通常都涉及的关键概念。

第 5 章：企业服务总线

本章观察了企业服务总线（ESB），它是 SOA 景观中服务于高互操作性的基础设施。

第 6 章：服务分类

本章显示了如何分类服务才能处理不同的服务类别、服务层次和 SOA 扩展阶段。

第 7 章：业务流程管理

本章介绍了业务流程管理（BPM），它是识别作为业务流程一部分的服务的方法。它包括配制、业务流程执行语言（BPEL）、业务职责管理，以及编排。

第 8 章：SOA 和组织

本章讨论了 SOA 战略对组织和公司的影响。

第 9 章：上下文环境中的 SOA

本章探讨了 SOA 如何适应其他的架构和方法，如何在业务的不同部分安排不同的处理层次。

本书第 2 部分讨论引入和运行 SOA 的特定考量。虽然提出来的话题有一个逻辑顺序，然而，我已努力把这些章写得让你可以用任意顺序阅读，前提是你理解了基础的概念和术语。

第 10 章：消息交换模式

本章介绍和讨论了消息交换模式（MEP）。MEP 定义了一个服务调用或操作中的消息序列。这些模式之一将带来事件和事件驱动的架构（EDA）。

第 11 章：服务生命周期

本章沿服务的生命周期前进，从需求识别到实现，从运行到撤销。

第 12 章：版本划分

本章讨论棘手的服务版本问题，包括相关数据类型的版本划分。

第 13 章：SOA 和性能

本章讨论性能，尤其是运行时性能是如何影响服务的设计和可重用性的。

第 14 章：SOA 和安全

本章展示了 SOA 实施中的安全问题，以及如何提出这些问题。

第 15 章：技术细节

本章探讨了 SOA 的一些技术细节，包括状态、幂等性、测试和调试，以及基础数据类型。

第 16 章：Web Services

本章研究了 Web Services 及其作为 SOA 基础设施事实上标准的地位。它展示了最重要的 Web Services 标准，以及在实践中应用它们的涵义。

第 17 章：服务管理

本章讨论了如何用业务库和注册中心来管理服务。

第 18 章：模型驱动的服务开发

本章描述了将服务阐明为模型，以及从这些模型中产生代码的重大意义。

第 19 章：建立 SOA 和 SOA 监管

本章研究了在一个组织中，SOA 能够或应该如何建立，并接着探讨了 SOA 的监管。

第 20 章：后记

本章最后讨论了关于 SOA 的一些主要问题，包括它是否真的是新东西，它用在哪儿合适，以及它增加还是降低了复杂性。

本书以一个书目纲要（包括所有你能在因特网上找到的被引用的资源）、一个 SOA 基础术语词汇表和一个索引结束。

本书中使用的约定

Conventions Used in This Book

本书使用如下排版约定：

***斜体*或黑体**

用于强调、定义新术语处、URL 和电子邮件地址。

等宽体（`Courier New`）

表示命令、代码范例、参数、值、对象、服务名字、XML 标签、文件内容或命令的输出。

等宽加黑体（Courier New）

用于部分高亮显示的代码。

等宽加斜体（Courier New）

表示应由用户提供的值来替换的文本。

评论与提问
Comments, and Questions

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

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

奥莱理软件（北京）有限公司
北京西城区 西直门 南大街 2 号 成铭大厦 C 座 807 室
邮政编码：100055
公司传真：8610-88097463
公司电话：8610-88097475 / 88097476 转
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

北京博文视点资讯有限公司（武汉分部）
湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室
邮政编码：430074
电话： (027)87690813　　传真：(027) 87690595
读者服务网页：*http://bv.csdn.net*
E-mail：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com（投稿信箱）

出版商为本书建立了网页，来提供勘误表、案例、其他附加信息，网址如下：

http://www.oreilly.com/catalog/9780596529550/（原书）
http://www.oreilly.com.cn/book.php?bn=978-7-121-07041-9（中文版）

评论本书或讨论本书相关的技术问题，可发邮件至：

bookquestions@oreilly.com

欲了解更多 O'Reilly 书籍、会议、资源中心、网络等信息，请访问 O'Reilly 网站：

http://www.oreilly.com

本书作者的网站为：

http://www.soa-in-practice.com

致谢
Acknowledgments

写一本书是一个要花许多年和牵涉到许多不同人的过程。我无法对他们所有人逐一表示感谢。然而，我想感谢一些对本书来说特别值得感激的人。

首先，我想感谢本书所有的评审人。他们做了件了不起的工作，对本书包含错误、难以阅读的早期版本给出了反馈意见。感谢 Alan Lenton、Mirko Drobietz、Gudrun Dürr、Thomas George、Jochen Hiller、Gregor Hohpe、Alan Lenton、Christian Möllenberg、Bruce Sams、Steffen Schäfer、Hermann Schlamann、Markus Völter 和 Torsten Winterberg。

其次，我想感谢在 O'Reilly 公司负责本书的编辑，感谢他们使本书得以付印，感谢他们所有的支持。感谢 Simon St.Laurent、Mike Hendrickson、Mary O'Brien、Tatiana Apandi、Caitrin McCullough、Nancy Reinhardt、Sumita Mukherji 和 Jessamyn Read。特别的感谢献给 Rachel Head，一个把我的“德国英语”转换成了“美国英语”的人，他做了件了不起的工作。

再次，我想感谢在实际项目中，以及通过书籍、文章、演讲、私下交谈等方式，帮助我学习和理解了 SOA 这个主题的所有人。任何时候，只要某些关键信息引自某人、某书或某资源，我已经一一予以指出。如果我忘记了什么事或什么人，请原谅。

除此之外，我想要感谢我的孩子们，他们是我的强大支撑，他们帮助我理解了什么才是真正重要的。感谢 Lucas、Anica 和 Frederic。

最后但绝非最不重要的，我感谢 Jutta Eckstein 为本书，为我的生活所扮演的角色。Jutta 说服我公开自己的 SOA 知识，给了我难以置信的支持，并使我人生的每一天都更有意义。

享受你的人生。

— Nicolai M. Josuttis
Braunschweig，2007 年 7 月

目录
CONTENTS

前言 I

第 1 章：动机 1

1.1 大型分布式系统的特征 3

1.2 魔术总线故事 4

1.3 魔术总线故事给我们的启示 6

1.4 SOA 历史 7

1.5 五页幻灯讲完 SOA 8

第 2 章：SOA 11

2.1 SOA 定义 11

2.2 SOA 驱动因素 13

2.3 SOA 概念 16

2.4 SOA 要素 18

2.5 SOA 不是银弹 21

2.6 SOA 不是特定技术 22

2.7 对比 SOA 与分布式对象 23

2.8 SOA 术语 23

2.9 总结 24

第 3 章：服务 25

3.1 服务 25

3.2 接口和契约 26

3.3 额外的服务特性 29

3.4 总结 34

第 4 章：松耦合 35

4.1 对容错的需求 35

4.2 松耦合的形式 36

4.3 处理松耦合 45

4.4 总结 46

第 5 章：企业服务总线 47

5.1 ESB 职责 47

5.2 异质的 ESB 49

5.3 ESB 的差异 50

5.4 ESB 增值服务 56

5.5 总结 59

第 6 章：服务分类 61
6.1 基础服务分类 62
6.2 基本服务 63
6.3 组合服务 67
6.4 流程服务 71
6.5 其他服务分类法 73
6.6 技术服务和基础设施服务 77
6.7 服务之外 78
6.8 总结 79
第 7 章：业务流程管理 81
7.1 BPM 术语 82
7.2 BPM 和 SOA 83
7.3 用于服务的 BPM 的例子 84
7.4 业务流程建模 87
7.5 识别服务的其他方法 94
7.6 配制和编排 96
7.7 更多思考 98
7.8 总结 98
第 8 章：SOA 和组织 101
8.1 角色和组织 101
8.2 出资模型 106
8.3 总结 107
第 9 章：上下文环境中的 SOA 109
9.1 基于 SOA 的架构模型 109
9.2 处理前端和后端 114
9.3 总结 121
第 10 章：消息交换模式 123
10.1 MEP 简介 123
10.2 基本 MEP 124
10.3 更复杂的 MEP 128
10.4 处理可靠性和错误 129
10.5 处理不同 MEP 层 131
10.6 事件驱动的架构 134
10.7 总结 136
第 11 章：服务生命周期 137
11.1 开发中的服务 137
11.2 生产中的服务 140
11.3 总结 143

第 12 章：版本划分 145
12.1 版本划分需求 145
12.2 领域驱动的版本划分 146
12.3 数据类型的版本划分 149
12.4 配置管理驱动的版本划分 155
12.5 实践中的版本划分 156
12.6 总结 157
第 13 章：SOA 和性能 159
13.1 性能在哪儿重要 159
13.2 从远程存储过程到服务 162
13.3 性能和可重用性 164
13.4 性能和向后兼容性 169
13.5 总结 171
第 14 章：SOA 和安全 173
14.1 安全需求 173
14.2 处理安全需求 174
14.3 实践中的 SOA 安全 178
14.4 XML 和 Web Services 的安全 182
14.5 安全问题何时凸现 188
14.6 总结 188
第 15 章：技术细节 191
15.1 服务和状态 191
15.2 幂等性 198
15.3 测试和调试 201
15.4 处理技术性数据（头数据） 203
15.5 数据类型 204
15.6 错误处理 207
15.7 总结 208
第 16 章：Web Services 209
16.1 使用 Web Services 的动机 209
16.2 基础 Web Services 标准 211
16.3 实践中的 Web Services 220
16.4 结束按语 228
16.5 总结 229
第 17 章：服务管理 231
17.1 服务中间人的历史 231
17.2 业务库和注册中心 233
17.3 总结 236

第 18 章：模型驱动的服务开发 237
18.1 生成的服务代码 237
18.2 建模服务 239
18.3 实践中的元模型 244
18.4 建立 MDSD 过程 247
18.5 工具 255
18.6 避免瓶颈 258
18.7 总结 258
第 19 章：建立 SOA 和 SOA 监管 261
19.1 引入 SOA 261
19.2 SOA 监管 262
19.3 按部就班 SOA 266
19.4 其他 SOA 方法 271
19.5 其他建议 274
19.6 总结 278
第 20 章：后记 281
20.1 SOA 是新东西吗？ 281
20.2 SOA 增加了复杂性吗？ 282
20.3 SOA 关键的成功要素是什么？ 282
20.4 SOA 在哪些地方不合适？ 283
20.5 SOA 替代了 OOP 吗？ 283
参考文献 285
词汇表 293
索引 303

第1章
动机
Motivation

我们生活在艰难的时代。社会市场经济正被全球市场经济所取代，市场营销人员统治了世界。相应地，你必须够快够灵活才能生存。这是达尔文主义的复兴：

> 并非最强壮或最聪慧的种族，而是最能应变的种族得以生存。

关键是灵活性。对所有主要的公司和大型分布式系统来说，信息技术（IT）灵活性至关重要。事实上，IT 已经成为一个关键的业务价值使能器。

同时，流程和系统正变得越来越复杂。我们已经超越了“自动化”主要是各个系统自己的事的阶段，正在快速迈入这样的世界：所有那些单个系统将变成“一个”分布式系统。我们面临着“可维护性”的挑战。

人们发现，处理可伸缩性和分布的老办法不管用了。我们不再能一致化系统，或维持对系统的控制。一致化和控制的前提是集中，集中无法扩展，而我们已经将集中发挥到了极致。正因如此，我们需要一个新的方法——一个接受异质、带来分散的方法。

另外，我们需要解决“业务/IT” 鸿沟的难题。这条鸿沟主要是语义上的——业务人员和 IT 人员似乎在用完全不同的语言交流和思考。新方法必须将业务和 IT 的距离拉近到前所未有的程度。

“面向服务的架构（SOA）”正合所需。它是一种帮助系统在增长的同时保持可扩展性和灵活性的方法，它也有助于填平“业务/IT”鸿沟。该方法由三个主要元素构成：

- **服务**。服务一方面体现了自足的、能作为一个或多个流程一部分的业务功能；另一方面，服务能由任何技术，在任何平台上实现。
- **企业服务总线**（ESB）。它是个专门的**基础设施**，它使我们能够用简单和灵活的方式来结合这些服务。
- **政策和过程**。它们处理这样的事实——大型分布式系统是异质的、处于不断维护中的，其所有者各异。

SOA 接受这样的观念：在大型分布式系统中，保持灵活性的唯一办法是支持异质、分散和容错。

听起来像一个美梦，是不是？

问题在于你买不来 SOA，你必须理解 SOA，亲自经历 SOA 过程；SOA 是个范式；SOA 是种思维方式；SOA 是架构和设计的评估体系。

本书将基于实际经验讲解 SOA 范式和评估体系。人们通常基于简单的陈述和原型来讲解 SOA，这就导致了臭名昭著的“曲棍球棒方程”（见图 1-1）所示的问题（注 1）。到达一定的复杂度之前，所需的工作量较低，事情看起来挺好。但是，当超过这个复杂度水平后，所需的工作量突然开始急剧上升，其上升幅度大于你能从中获益的程度，最终结果就是崩溃。

经常地，SOA 只被部分讲解和装备。只引入类似于 Web Services 的基础设施，在一定复杂度上可能有所帮助，然而，这不足以保证可伸缩性。整个结构中，处理服务的、处理基础设施的、处理政策的和处理过程的所有东西都必须互相吻合。一旦你理解了如何实施 SOA，这并不困难，但是，这需要时间和勇气（好的，如此说来这还是困难）。为了帮助人们理解 SOA（从你自己开始），需要付出大量努力。如果不愿意付出努力，你就会失败。

在我们深入细节之前，我想谈谈 SOA 的背景和历史，以作为本书后续内容的基础。接下来的各节会讲些关于 SOA 的“故事和神话”，它们能帮助你熟悉 SOA。

注 1：感谢 Gregor Hohpe，他在 2006 年 OOP 上告诉了我“曲棍球棒方程”。

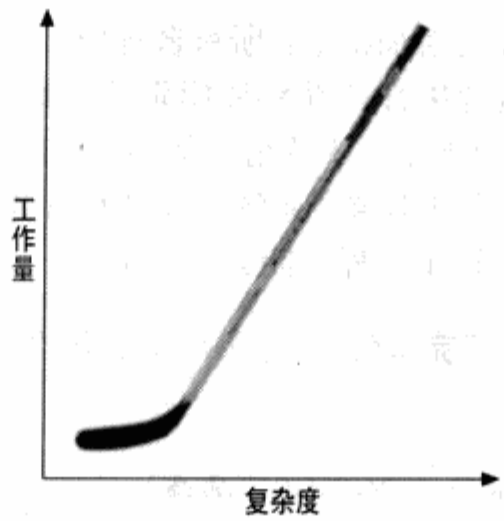


图 1-1：曲棍球棒方程

1.1 大型分布式系统的特征
Characteristics of Large Distributed Systems

SOA 是用于大型分布式系统的概念。要理解 SOA，你必须了解大型分布式系统的特点。

首先，大型系统必须处理“**老系统**”。引入 SOA 时，你不可能从头开始设计一切。你必须处理这样的事实：当前在用的大多数系统会一直用下去。这也意味着建立 SOA 不是一个设计新系统的项目。SOA 涉及改变现有系统的结构，这意味着你必须和老平台及向后兼容问题打交道。实际上，SOA 是对大型系统景观开展“**维护**”工作的方法。

所有大型系统天生就是**异质的**。这些系统目的有别、实施时间各异、新旧程度差异悬殊。你会发现大型系统呈现出堆积了不同平台、编程语言、编程范式，甚至不同中间件的系统景观。过去，人们多次试图依靠一致化解决可伸缩性问题。是的，一致化确实有帮助。剔除不再可维护的老平台和系统是重大的改善。然而，一般说来，你的系统永远无法彻底一致化。正当你将要剔除最后一个异质体时，一次公司兼并或其他变化将重新打开潘多拉之盒。

异质性的一个原因是，大型系统和它们的数据有非常长的**生命周期**。在此生命周期中，通过加入新的系统和流程，不断开发出促进业务的新功能。虽然剔除现有系统和数据有可能看起来没有业务价值，然而，这种改造是对系统可维护性的投资。通常说来，这类投资都进行得很晚，以致其代价变得不可思议地高昂，因为系统已经失控，关于系统的所有知识均已失传。

大型系统天生是**复杂的**。因此，找到正确的修改点和决定修改的影响都很困难。正如[Spanyi03]所说：

> 没有“快速修复……”这回事。组织机构是复杂的业务体系，在其中改任何一个组件都可能对其他组件有影响。

大型分布式系统还有一个重要的附加特征：**所有者各异**。不同的团队、部门、分支机构或公司都可能在维护系统，这意味着必须考虑不同的预算、进度表、立场和利益。作为SOA 实施者，你的工作范围超越了正式的组织结构，因此，你通常面对的情形是，自己没有足够的权利和控制力，无法对整个系统的设计和行为发号施令。谈判和协商是必需的，而且，由于政治纠葛，这个过程可能会比较艰苦。

大型系统另一个关键特征是“**不完美性**”。尽善尽美的代价太高，正如 Winston Churchill 所说：

> 尽善尽美的拼法是 P-A-R-A-L-Y-S-I-S（瘫痪）。

运转中的系统看起来总有点美中不足。它们能做对 99%的工作，但剩下的 1%会出问题，这通常带来额外的手工作业、对危机管理的需求，或是愤怒的客户。注意，不完美的程度各不相同（至关重要的系统通常有更高比例的完美度，但是，即使对它们而言，通常也存在一个点，在此点之上，消灭风险也得不偿失）。

同样地，大型系统通常有一定的“**冗余度**”。虽然有的冗余度可能属于无心造就，但是，也有相当数量的冗余度是有意为之，处于管理之下的。因为，在实践中，根本不可能对所有数据进行规格化处理，让它们仅仅存放在一个地方。消灭冗余度很困难，这样做会引发基本的运行时效率问题。在一种简单的冗余形式中，至少哪个是主数据是明确的（所有的冗余数据都来自主数据）。在复杂场景下，主数据也有多份，而且/或者主数据没有明确定义。这样一来，真实世界场景下的一致性维护就变得非常复杂。

最后，对大型系统来说，**系统瓶颈就是自杀**。这并非意味着没有瓶颈存在，但是，通常说来，避免瓶颈、能够伸缩是个目标。注意，我并非只讲技术瓶颈。在大型系统中，当瓶颈是流程或组织结构的一部分时，它也会阻碍可伸缩性。

1.2 魔术总线故事
The Tale of the Magic Bus

很久很久以前，有这样一家公司，它已经成长了很多很多年。在经过的这些年中，系统景观被一种叫做“大杂烩”的病感染了。结果，人们失去了对系统的控制，而且，只要大家试图改进系统，要么发现需要的工作量太大，要么就是把事情弄得更糟。

公司求助于好些专家、智者和巫师，这些人提了许多主意，引入了新的模式、协议和系统设计。但是，结果只有一个——事情变得越来越糟。如此这般，公司已经陷入绝境。

有一天，一个叫做企业集成的王子出现了，宣布说他有解决方案。他告诉公司的 CEO，“您的问题在于缺乏互操作性。当您有这样一个大杂烩式的系统和协议时，如果必须针对每类连接创建单独的解决方案，就会出问题。即使您只有 10 种不同的平台，5 种不同的协议，如果想要每个平台都能与其他平台通信，将需要超过 100 种不同的解决方案。更何况您拥有的平台远超过 10 种。”如何算出这个数字的准确方法没有传下来，但是，一些处理草图——把两个平台的每个可能连接与使用协议的平均数量结合起来考虑——引出了这个结论。

“看看我的图，”王子继续讲（图 1-2 是复制的图）。“这就是如何解决您的问题的方法。我们创建一个‘**魔术总线**’。”

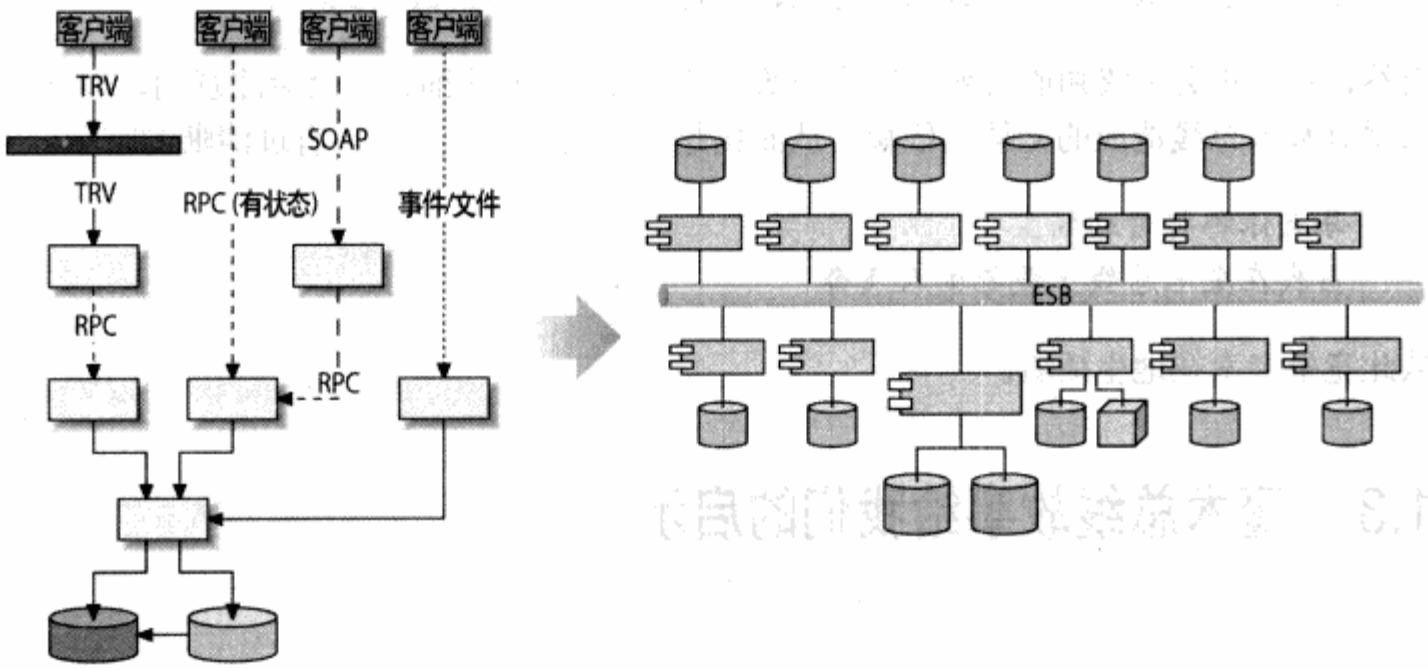


图 1-2：魔术总线的原图

“什么是魔术总线？”CEO 问道。

王子回答说，“魔术总线是一个软件，它减少您系统中的连接和接口数量。对 n 个系统（以及两倍于系统数的接口），您的方法可能需要多达 $n \times (n-1)/2$ 个连接；针对每个系统，魔术总线只需要一个连接、一个接口，这就意味着对 n 个系统，接口数量按 $n-1$ 变化率被削减（该变化率对 10 个系统是 9，对 30 个系统是 29，对 50 个系统是 49）。”

CEO 被这些数字说服了，同意转换到这个新方法上。整个国家对新方法是如此推崇，以致吟游诗人开始为魔术总线写颂歌。最著名的颂歌来自“谁人”乐队（译注 1），他们用如下语句盛赞了魔术总线（完整歌词见[MagicBus]）：

> 魔术总线，你是总线时代的奇迹

译注 1：“谁人”（The Who）乐队，英国的一支老摇滚乐队。魔术公共汽车（Magic Bus）是该乐队创作的歌曲。由于“总线”一词的英文与“公共汽车”相同，所以作者开玩笑说“谁人”乐队这首歌是歌颂总线的。

现在，你也许会期望故事的结尾是“从此他们幸福地生活着”，但是，故事没有就此结束。当公司切换到魔术总线之后，连接系统确实很简单了，而且突然间，增长却变得非常容易。于是，系统增长啊，增长啊，增长啊……直到突然间，所有东西都垮了。

发生了什么事？

人们发现，有了魔术总线后，没人再能理解系统之间的依赖关系。过去，你能看到连接了哪些系统，因为每个连接都是独特的，但是这不再可能了。修改一个系统可能造成其他系统的问题，而且，人们经常惊讶于这些系统间居然存在联系。实际上，每个系统都依赖其他所有系统。当修改任何东西的风险都变得如此之高时，公司决定对整个系统放任自流。但是，敷衍了事就意味着死亡，于是，一年之后，公司倒闭了。

当然，公司其实应该知道后来发生的一切。“谁人”颂歌的细心听众会注意到，颂歌的末尾对魔术总线造成的无结构依赖，从而扬起了“灰尘”的危险，有过清晰的暗示：

> 每天你都将看到扬尘
> 当我在魔术总线上载着爱人飞奔

从此竞争者幸福地生活着。

1.3 魔术总线故事给我们的启示

What We Can Learn from the Tale the Magic Bus

如今，我们经常颂扬“总线时代的奇迹”（注 2）。虽然 IT 总线概念已经相当古老了，但在最近，这个概念迎来了某种复兴。复兴始于**企业应用集成总线**（EAI 总线）的引入，随之被**企业服务总线**（ESB）所代替。这个概念变得如此重要，以致“谁发明了 ESB”的问题在网络论坛上引发了公开的论战（见[ESB Inventor]）。

总线代表了高互操作性。其背后的思想是，不去为不同系统间创建和维护单独的通信渠道，每个系统只和总线连接就能和所有其他系统连起来。当然，这确实简化了连接操作，但是，正如前面的故事揭示出的，这个方法有缺陷。除非加上结构，否则连接的扩展将导致混乱。这也是我们为什么用子程序代替全局变量，用模块和组件代替业务对象模型的原因。如果我们用非结构化的方式处理服务，类似的混乱会再次发生。

注 2：有人可能会抗议说歌词中没有这个句子，请注意，这首歌有很多版本，请听 7 分钟现场演唱版。

综上所述，你所上的第一课就是，为了大型系统能够伸缩，所需要的不仅仅是互操作性。你需要由技术规则、组织规则和模式所提供的结构。高度互操作性必须伴随着定义良好的架构、结构和过程。如果你很晚才意识到这点，就可能在市场上落败。

1.4 SOA 历史
History of SOA

令人惊讶的是，很难说清是谁创造了 SOA 这个术语。在 2007 年的一次私下交谈中，Gartner 的 Roy Schulte 告诉了我 SOA 的准确历史：

> Alexander Pasik，Gartner 的前分析师，在 1994 年他教的一个中间件课程上，创造了 SOA 术语。帕斯克工作在 XML 和 Web Services 尚未发明的年代，但是，基本的 SOA 原则没什么变化。
>
> 促使帕斯克创造 SOA 术语的原因是——“客户机/服务器”术语失去了其经典含义。业界中的许多人开始使用“客户机/服务器”来描述包含了 PC 和其他计算机的分布式计算。通常，一台桌面“客户机”PC 运行面向用户的表示层逻辑，以及绝大多数业务逻辑。后端“服务器”计算机运行数据库管理系统，存储数据，有时也运行业务逻辑。在这种用法中，“客户机”和“服务器”通常指硬件。从原来“客户机/服务器”的意义来看，前端 PC 上的软件虽然有时也和服务器软件有关，但大多数时候无关。为了避免在新的和老的“客户机/服务器”意义上产生混淆，当帕斯克鼓励开发人员设计 SOA 商业应用程序时，他强调“面向服务器”。
>
> **注意**
>
> Gartner 的分析师 Roy W. Schulte 和 Yefim V. Natis 于 1996 年发表了关于 SOA 的第一篇报告。细节见[Gartner96]和[Gartner03]。

SOA 的真正动力来自 Web Services 的创建。Web Services 最早由微软推动，2000 年时为更多公众所知（关于 Web Services 的历史细节，见 16.1.2 节）。引用如下[Gartner 03]：

> 虽然 Web Services 不一定要解释为 SOA，也不是所有 SOA 都基于 Web Services，然而，这两个技术方向之间的联系是重要的，并且它们也相互影响：Web Services 的强劲势头将把 SOA 带给主流用户，SOA 架构的最佳实践将帮助 Web Services 从起步阶段走向成功。

很快，其他公司和供应商也加入了进来（包括像 IBM、Oracle、HP、SAP 和 Sun 这样的主要 IT 供应商）。靠解释概念、销售新概念和工具（或者给老概念和工具打新标签），大家开始挣钱了。另外，时机也刚好，因为各公司正逐步摸索着将自己的业务和其他的系统、部门及公司集成起来（想想 B2B 热潮）。

之后，分析师们开始兜售 SOA，称其为未来软件的关键概念。例如，2005 年 Gartner 如此说道[Gartner05]：

> 到 2008 年，SOA 将成为 80%开发项目的基础。

时间会告诉我们这个命题是否正确——它很可能成为一个“因其提出，故有实现”的自我实现式的预言。

然而，因为人们热炒过度、滥用概念和术语，所以每个重大动向都会招致批评。Grady Booch，UML 缔造者之一，现为 IBM 名士，在他 2006 年 3 月的博客中，对 SOA 有如下评论（见[Booch06]）：

> 对 SOA 这整件事情，比起大多数我看过的观点来说，我自己的观点有那么点尖刻。关于 SOA 人们讲了太多，使 SOA 看起来像是自穿孔卡片时代以来最好的东西。显然，SOA 不光会转型组织，使你更加灵活和有创造力，SOA 还能使你家的青春期少年开始和你谈心，并且你也会成为一个更棒的情人。或者，如果你的名字凑巧是“老二”的话，SOA 还能使你有更强的性能力。更妙的是，如果你按着讲的这些道道来，将发现做到这些几乎不会有什么困难：只要将你目前有的东西削一削，在这儿、那儿，还有那个远地儿种上些服务，把它们捆到一块儿，突然，你就被虚拟化、自动化、服务化。
>
> 多么垃圾的说法。

Booch 是对的。重要的是，SOA 是一项需要时间和努力的战略。你需要一些经验才能理解 SOA 到底讲了什么，它将在何处，以及如何对你有帮助。幸运的是，SOA 概念已经出现了足够长时间，我们有些人确实获得了关于 SOA 的重要经验（不是简单的原型，以及用百来个服务把几打系统集成起来这种模糊的想法）。

1.5 五页幻灯讲完 SOA
SOA in Five Slides

本书剩下部分将讨论 SOA 在实践中的几个方面。这意味着我们会比通常的“五页幻灯”法要进行得深入些。“五页幻灯”法用如此简单的方式呈现 SOA，以致大家都会倍感困惑——“这有啥复杂的？”以及/或者“这有啥重要的？”

然而，为了让你对将要学习的东西的精髓有个初步印象，在此献上本人介绍 SOA 的五页幻灯。记住，这五页幻灯给出的是过度简化的观感。魔鬼藏于细节中。不过，比起通常的 SOA 广告，这个概览可能看起来还是有那么点不同。

1.5.1 幻灯 1：SOA
Slide 1: SOA

面向服务的架构（SOA）是个范式，目的是实现和维护跨越了大型分布式系统的业务流程。它基于三个主要技术概念：服务、通过企业服务总线达成的互操作性，以及松耦合。

- 一个**服务**是一项自足的业务功能。功能可能简单（存储或获取客户数据），也可能复杂（处理客户订单的流程）。因为服务集中于接口的业务价值，所以它们填补了“业务/IT”鸿沟。

- **企业服务总线**（ESB）是个基础设施，它使分布式系统和服务间的高互操作性成为可能。有了它，用不同平台和技术在众多系统间分布业务流程比以前简单了。
- **松耦合**是减少系统依赖的概念。因为业务流程分布在多个后端系统上，最小化修改和故障的影响至关重要。否则，修改会变得风险太高，系统故障可能导致整个系统范围的崩溃。然而，要注意松耦合有相应的代价：复杂化。松散耦合的分布式系统更难开发、维护和调试。

1.5.2 幻灯 2：政策和过程
Slide 2: Policies and Processes

分布式处理不是个技术细节。分布式处理改变你公司的一切。引入新功能不再是给特定部门指派特定任务，它现在是对不同系统和多个任务的组合。这些系统和牵涉到的团队必须协作。

正因如此，你需要清晰地定义角色、政策和过程。这些过程包括，但不仅限于定义一个服务生命周期，以及实行模型驱动的服务开发。另外，你还必须为分布式软件开发建立几个过程。

通常情况下，建立合适的政策和过程比理清最初的技术细节更花时间。记住 Fred Brooks 在 1974 年说过的（见[Brooks74]）：

> 做一个编程系统产品比作一个简单的程序成本高 9 倍。

增加一个三倍乘数是因为这是个产品（软件被“任何人运行、测试、修复及扩展”），增加另一个三倍乘数是因为这是个系统组件（集成和集成测试引入了工作量）。当众多组件加入进来时，乘数还会增加，这就是 SOA 面对的情况。

1.5.3 幻灯 3：Web Services
Slide 3: Web Services

Web Services 是实现 SOA 技术方面的可能途径之一。（然而，要注意，SOA 不光是技术方面！）

但是，Web Services 自身也引入了一些问题。首先，其标准不够成熟，尚不能保证互操作性。其次，Web Services 先天不足，无法形成程度适当的松耦合。

所以，你不应该指望用 Web Services 解决所有的技术问题。你应该预算出足够的资源（时间和金钱）来解决剩下的问题。

同时，你也不应该掉进过于 Web Services 化的陷阱。Web Services 不会是系统集成的终极标准。因此，只有当 Web Services 对特定的基础设施考量确实重要时，才应把它加进来发挥作用。

1.5.4 幻灯 4：SOA 实践
Slide 4: SOA in Practice

在理论上，理论和实践是一致的。在实践中则并非如此。

——L. Berra

当然，这句话对 SOA 也适用。当诸如性能和安全等因素被考虑进来后，一般的业务案例和概念就可能不像预期那样工作了。

另外，由于 SOA 是针对处于维护中的现有系统的策略，也带来了稳定性和向后兼容问题。

而且，从 IT 角度来看，所有的系统都不一样。因此，你不得不打造你自己的 SOA——SOA 买不到。要手工打造 SOA，你需要时间，需要一个渐进和迭代的方法。

另外要注意，你是否引入 SOA 并不重要。重要的是对你的上下文环境和需求来说，你引入的 IT 解决方案是合适的。

1.5.5 幻灯 5：SOA 监管和管理支持
Slide 5: SOA Governance and Management Support

也许，SOA 最重要的考量是，找到监管的正确方法和恰当力度：

- 你需要一个中央团队来决策自己的 SOA 的基本方面。然而，由于终极目标是分散化（这是大型系统的关键），所以，你必须在集中化和分散化之间找到正确的平衡点。
- 你需要合适的人选。大型系统不同于小系统，你需要对这些系统有经验的人。当由于实际情况造成有的概念不能伸缩时，经验欠缺的人会力图与那些实际情况抗争，而未能充分认识到这些都是大型系统的天生特质。另外，中央服务团队通常有“象牙塔”化的趋势。他们必须被业务团队的需求驱动。实际上，他们必须将自己看作“服务基础设施”的服务提供者。
- 重要的事情先来。不要从管理服务开始。有许多服务后来才需要管理服务。刚开始时，不要使用首先设计所有服务或首先提供基础设施的方法。所有东西必须一起成长，在成长中，解决眼前的问题就要占用你大部分精力，不要去担心以后才有的问题。
- 最后但绝非最不重要的是，你需要 CEO 和 CIO 的支持。SOA 是影响到公司整体的战略。让 CEO 和 CIO 支持并做出合适的决策，以及给你足够的时间和经费。要注意的是，短期内经费充足并不是最重要的，你需要的是长期的经费。当调研工作仅仅完成一半时削减 SOA 预算是自求灾难。

第 2 章
SOA

本章的目标是对 SOA 这一概念进行介绍。我想要勾勒出 SOA 的基本轮廓，向你展示 SOA 正确应用的场合。重要的一点是，SOA 是一种范式（或一个概念、一种哲学），针对所有者各异的大型分布式系统，SOA 给出了一个评估体系。

我会引用、对比、讨论当前存在的各种不同定义。这些定义来源各异，如“OASIS SOA 参考模型”、“Wikipedia.org”，还有某些书。我会展示这些定义如何不同，以及为何不同，指明 SOA 呈现出的关键特点。

2.1 SOA 定义
Definitions of SOA

对 SOA 这个术语下个准确的定义很困难。困难不在于没有定义，而在于有太多形形色色的定义。为了使你对这些定义的相同和不同点建立概念，本章挑选了些已经公布的定义，放在选读栏。读这些定义的时候，你会发现一些通用的短语、特征，然而，你也会发现各定义在背景、抽象层次，以及措词上的差异巨大。

然而，至少所有定义都同意：SOA 是一种范式，目的是增强灵活性。

维基百科 SOA 定义（2006 年 2 月 19 日）

在计算科学中，术语“面向服务的架构”（SOA）表达了一种软件架构概念，它定义为使用服务来满足软件用户的需求。在 SOA 环境中，网络上的节点以独立服务的形式将自己的资源开放给网络上其他参与者，其他参与者按一种标准的方式使用资源。大多数对 SOA 的定义都提到实现 SOA 时使用 Web Services（即使用 SOAP 或 REST）。然而，可以用任何基于服务的技术实现 SOA。

…

与传统点对点架构不同，各种 SOA 都由松耦合、高度可互操作的应用服务构成。这些服务基于某种格式定义进行互操作，该定义独立于底层平台和编程语言（例如，WSDL）。接口定义封装（隐藏）了供应商和语言相关的实现。SOA 独立于开发技术（比如 Java 和.NET）。由此，软件组件将具备非常高的重用性，因为接口按与标准兼容的方式定义。

2.1.1 SOA 是个范式

SOA Is a Paradigm

SOA 不是种具体的架构：它是能够**导出**某种具体架构的东西。你可以将它叫做一种风格、范式、概念、观点、哲学，或者表示法。这也意味着 SOA 不是一种可以买到的具体工具或框架。它是种方法，是种思考方式，是个价值体系。当设计一个具体的软件架构时，从 SOA 可以导出某些具体的决策。

“你买不来 SOA” 这点影响深远。没有什么工具或药方能使你一用之下，万事大吉。当把这个范式用到具体的场合时，你必须根据自己的情况做出具体决策。

2.1.2 SOA 目标为增强灵活性

SOA Aims to Improve Flexibility

使用 SOA 的关键原因是，它确实有助于你的业务。例如，你可能需要一个 IT 方案，既能存储和管理你的数据，又能自动化一些常用的、处理数据的流程。

当前，商业成功的一个关键要素是——向市场份额看齐。正如一位电信经理人曾说过的（引用自 Jim Highsmith）：

> “你可以到处叫嚷‘软件质量’（或质量低劣问题），但是，市场营销人员统治着世界，他们现在要的是市场份额……句号，讨论完毕。我的工作就是按时交付，不超预算，符合‘适当’的质量标准。”

要准时交付一个质量过得去的方案，你需要灵活性。但是，灵活性和清晰的组织结构、角色、流程等关系紧密，因此，SOA 必须和所有这些方面打交道。

2.2 SOA 驱动因素
SOA Drivers

当然，在不同层次上，不同组件中，对灵活性的处理也非常不同。因此，一个重要的问题是：针对哪种类型的软件系统，SOA 这个范式是合适的？正如大家发现的，SOA 能很好地应对大型系统许多难以处理的特点。

2.2.1 分布式系统
Distributed Systems

当业务扩展时，软件系统变得越来越复杂，越来越多的系统和公司被牵连进来。集成持续进行，变化无所不在。SOA 很适宜处理复杂的分布式系统。根据 OASIS SOA 参考模型的（见[OasisSoaRM06]）说法，SOA 是“组织和利用分布式能力”的范式。

> **注意**
>
> 相对于“分布式能力”而言，更 IT 化的术语是“分布式系统”，或者，正如维基百科 SOA 定义所述，“网络的节点”或“网络资源”。

SOA 使需要一定分布式能力的实体能够定位、使用这些能力。换句话说，SOA 方便了服务供应者和服务消费者之间的交互，使业务功能的实现成为可能。

2.2.2 所有者各异
Different Owners

接下来，OASIS SOA 参考模型对 SOA 的定义谈到，这些分布式能力“可能处于不同所有权范围控制之下”。这是非常重要的一点，然而，通常的 SOA 定义往往会忽略它。在本章引用的任何其他定义中，你都找不到这点。但是，对 SOA 某些特性而言，这是关键所在，这也是 SOA 为什么不单是个技术概念的主要原因。

SOA 包括的某些实践和流程基于如下事实：分布式系统的网络不被某个唯一的所有者掌控。不同的组、不同的部门，甚至不同的公司都可能管理着不同的系统（见图 2-1）。如此一来，不同的平台、进度、优先级、预算，以及诸如此类的东西都必须被考虑进来。一般说来，这个概念是理解 SOA 和大型分布式系统的关键。

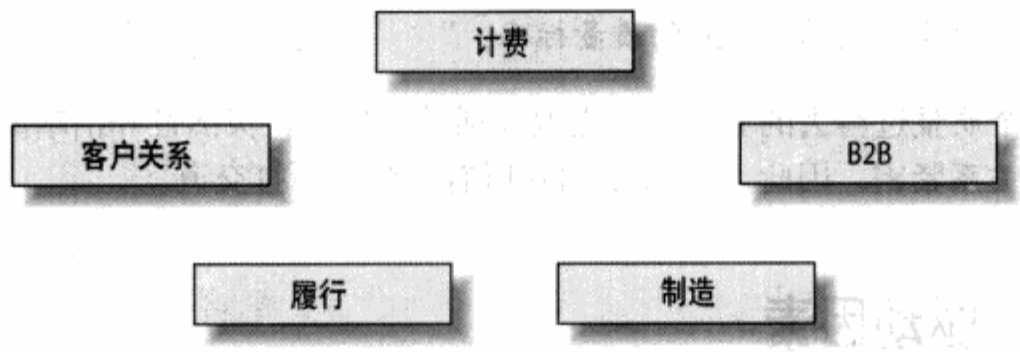


图 2-1：所有者各异的分布式系统

无论是处于一个所有者各异的环境中，还是处于一个你能控制一切的环境中，你处理问题、做出调整的方式会根据需要而相应变化。在大系统中，你不可能像在小系统中那样实现功能和改变系统行为。（见 1.1 节对大系统最重要特点的讨论）一个重要的考虑是，“政治”参与进来了：你必须向其他人妥协，你必须接受不同的优先级，不同的解决方案的存在。因为你不能控制所有东西，所以你必须接受这点——你也许不能总按自己的方式来办事。

2.2.3 异质
Heterogeneity

大系统和小系统的另一个非常重要的区别是：大系统缺乏一致性。根据自己的经验，我们都知道这点（尽管我们对这是一种自然现象，还是恶劣设计的结果存在争议）。大系统用不同的平台，不同的编程语言（及编程范式），甚至不同的中间件。它们是由大型机、SAP 主机、数据库、J2EE 应用、小型规则引擎，以及诸如此类的东西构成的杂烩。换句话说，它们是异质的。

过去，人们提出了许许多多的方法，试图消除异质性，解决集成分布式系统碰到的问题：“让我们一致起来，所有的问题都会消失”，“把所有的系统都换成 J2EE 应用”，“不管哪儿都用 CORBA”，“用 MQ Series”，等等。然而，我们都清楚，这些方法行不通。大型、所有者各异的分布式系统就是异质的（见图 2-2）。这是个事实，因此也是我们设计大型分布式方案时必须接受的前提。

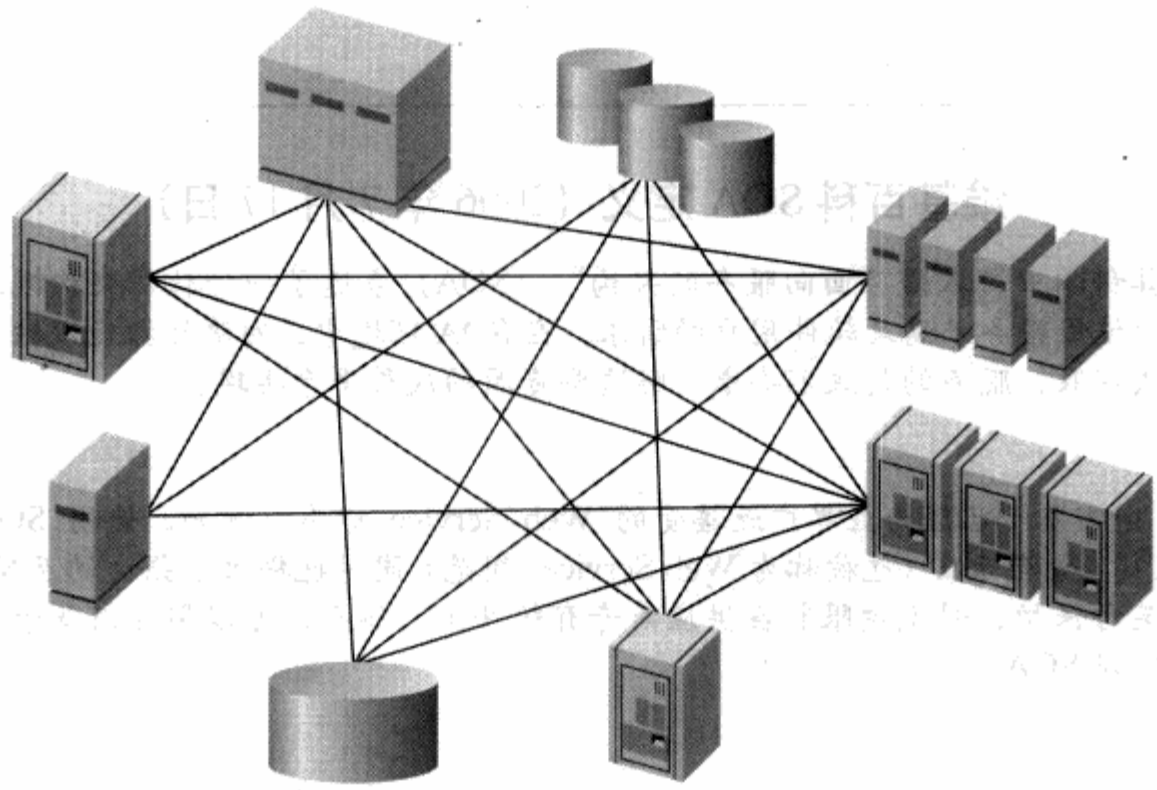


图 2-2：分布式系统是异质的

SOA 方法接受异质。SOA 正是依靠对异质性的承认和支持来处理大系统的。

这是 SOA 的关键点之一，它能赋予 SOA 发起变革的力量。正如软件开发中的“敏捷”方法不是与需求变化抗争，而是接受需求变化一样，SOA 接受大系统中的异质性。这导致非常不同的思考方式，突然之间，我们有了一种能真正起作用的处理大型分布式系统的方法。

请不要对我产生误解：我不是说异质化是个目标。如果你有一个同质化的系统，那很好，你很幸运——你将不用管很多常见的问题。

然而，绝大多数大型分布式系统都是异质的，而且，集成得越多，就越异质化。SOA 就是处理这种状况的方法。SOA 不关心异质化是否是好事；SOA 的目标仅仅是，在异质化存在的地方，用一种合适的方式处理异质化。

在本章引用的 SOA 定义例子中，异质化概念的表达方式各不相同：你能看到如“与底层平台和编程语言无关”，“与开发技术无关”，“供应商多样化”，以及“不管操作系统和编程语言如何”等。不同的措辞，它们表达了同一概念。

维基百科 SOA 定义（2006 年 7 月 17 日）

在计算科学中，术语“面向服务的架构”（SOA）表达了一种软件架构概念，它定义为使用服务来满足软件用户的需求。在 SOA 环境中，网络资源能以独立服务的形式获取，服务的获取不要求了解这些资源的底层平台实现。

……

SOA 通常基于已经为业界广泛接受的 Web Service 标准（例如，使用 SOAP 或 REST）。这些标准（也被称为 Web Services 规范）同时也提供了强大的互操作性，以及某种保护，避免被限制在供应商专有软件上。然而，可以用任何基于服务的技术实现 SOA。

……

SOA 也可以被视为一种信息系统架构风格，它使结合松耦合、互操作的服务来创建应用成为可能。这些服务基于某种格式定义（或契约，例如，WSDL）进行互操作，该定义独立于底层平台和编程语言。接口定义隐藏了语言相关的服务实现。由此，SOA 兼容的系统与开发技术和平台无关（比如 Java、.NET 等）。

……

SOA 能支持在复杂的企业系统中开展集成和合并行动，但是，SOA 并没有指明或提供一种方法论或框架来文档化能力或服务。

2.3 SOA 概念
SOA Concepts

以下列出 SOA 的关键技术概念，它们使 SOA 能够处理具备前述特点的系统：

- 服务
- 互操作性
- 松耦合

2.3.1 服务
Services

软件开发是关于抽象的艺术。必须以这样的方式来抽象现实——只处理问题相关的方面。然而，我们都知道，抽象可以从不同的角度进行。SOA 的抽象集中在问题的业务

角度。这里介绍的基本术语是“**服务**”。本质上，服务就是业务功能的 IT 体现。SOA 的目标就是基于对业务规则和功能的抽象来构筑大型分布式系统。

这就给设计和开发 IT 系统提供了清晰的结构。虽然这些系统从内部来看当然是技术系统，然而，外部接口必须设计得让业务人员也能够理解。从外部必须看不到技术细节。该方法的良好效果是，在这种抽象层次上，特定平台的细节无关紧要。由此，平台可以是异质的。

除了这个宽泛的定义，我们还是不清楚服务到底是什么，或者服务能够是什么。关于服务必须、能够，或者应该具备何种确定的特性，各种观点差异巨大。下一章我会具体讨论服务（同时，展示当前关于服务的几种不同定义）。然而，根据经验来看，你可以把服务当作自足的业务功能的 IT 体现，比如“创造一个客户”，“获得客户合同”，“转账”，“打开收音机”，“计算汽车的最佳线路”，等等。

2.3.2 高互操作性
High Interoperability

处理异质系统时，第一个目标是能轻松地连接这些系统。这通常被称为“高互操作性”。高互操作性不是什么新点子。在 SOA 之前，关于互操作性我们有“**企业应用集成**”(EAI) 概念，SOA 没提出新东西。

然而，对 SOA 而言，高互操作性是开始而非结束。我们从这个基础出发，实现跨越多个分布式系统的业务功能（服务）。

2.3.3 松耦合
Loose Coupling

正如我已指出的，我们生活在市场营销人员统治的世界中。我们并不总有足够的时间来仔细地分析、设计和实现。因为需要快速推向市场，所以，灵活性总是比质量更重要。这就导致了一些问题。

想一想，我们一边在集成越来越多的系统，一边在实现业务流程并把它们分布到不同的系统上。原则上，数据应以这样的方式流动——在所有涉及的系统中，流程都能顺利运转，这些流程可能是：我们在所有系统中创建一个客户，从一个系统向另一个系统划钱，或者，运送商品和发出账单来处理一个客户订单。

许多系统被涉及，然而，我们没有时间来处理健壮性。这听起来不太妙，在如此的复杂性下，一个微小的错误都可能中断整个业务过程。必须避免出现这样的情况，也就是说，我们需要容错。因此，我们有如下目标：

- 灵活性
- 可伸缩性
- 容错

满足这些目标的关键是**松耦合**。松耦合是表达最小化依赖的概念。当依赖被最小化时，改动造成的影响也被最小化了，即使部分系统崩溃或当机，整个系统也能运转。最小化依赖有助于容错和灵活性，这正是我们需要的。

不光如此，松耦合还带来可伸缩性。大系统容易出现满负荷运转，因此，避免瓶颈很重要，否则，增长可能带来高昂的成本。避免瓶颈的重要性毋庸置疑，无论从技术角度还是组织角度来看都是如此。只有平常的业务能以尽可能分散的方式来做，大系统才能顺利工作。引入松耦合的一个方法是，避免引入任何不必要的集中（不幸的是，建立 SOA 需要一定的集中，因为需要一个共同的基础）。

由于在实践中有许多方法能实现松耦合，因此，本书通篇会多次重提松耦合话题。切记，SOA 只是个范式，不是具体的药方。多大程度上引入松耦合取决于你自己。第 4 章会详细讨论松耦合，并给出几个解耦 SOA 系统和流程的示例方法。

2.4 SOA 要素
SOA Ingredients

既然已经读到 SOA 的关键技术概念是服务、互操作性和松耦合，你可能已经得出结论：建立 SOA 所需要做的就是引入服务、互操作性和松耦合。

但是，正如我前面指出的，SOA 买不来。重要的是用合适的方式引入这些概念。也就是说，你必须找到适度的集中，你必须建立相应的流程，你还必须钻研。在实际的 SOA 项目中，往往发现某些“要素”的缺失。要成功建立 SOA，必须仔细考虑基础设施、架构及流程（包括元流程、监管）。

2.4.1 基础设施
Infrastructure

基础设施是 SOA 中使高互操作性成为可能的技术部分。SOA 范畴的基础设施被称作“企业服务总线”（ESB）。该术语从“企业应用集成”借用而来，在那儿被称为 EAI 总线，或者直接称为企业总线。

ESB 的关键特性是，它使你能在异质系统间进行服务调用。它的职责包括数据转化、（智能）路由、处理安全和可靠性、服务管理、监测，以及日志。

第 5 章会详细讨论 ESB 的目的和特性。

2.4.2 架构
Architecture

架构用来对 SOA 的各种可能性加以限制，从而使系统运转顺利、维护方便。SOA 概念、SOA 工具和 SOA 标准都有意留下了未说明之处，你必须做出具体决策，否则将面临混乱局面。你必须分清不同类型的服务，决定松耦合的程度，限制服务接口的数据模型，定义政策、规则和模式，理顺角色和职责，选择基础设施技术，敲定使用哪个（版本的）标准，诸如此类。

本书经常会提到必须根据自己的具体情况（需求和上下文环境）做出相应决策。基于这些决策，建立起系统架构（或系统架构的体系）。

2.4.3 过程
Processes

大型系统复杂的一个原因是它牵涉了许多不同的人和组织。因此，从最初的业务思路或需求，到方案实际运转，处于生产状态，有一个漫长的过程。

因为通常并不由一个人或少数人控制着所有一切，所以你必须要建立适当的过程（这些过程可能是明确规定的，也可能是逐渐积淀出的潜规则）。这些过程包括：

业务流程建模（BPM）

BPM 的任务是把业务流程细分为更小的活动或任务，也就是我们所说的服务（见第 7 章）。

服务生命周期

定义服务生命周期，即定义服务成为现实需要经历的不同步骤（见第 11 章）。

模型驱动的软件开发（MDSD）

MDSD 是处理服务时生成代码的过程（见第 18 章）。

2.4.4 监管
Governance

“监管”是所有流程的元流程，也是 SOA 的一个总体策略。为了在组织中建立 SOA，你必须设立恰当的流程。这包括找到合适的人来结合所有的 SOA 要素，得到能工作、适合组织的 SOA。通常由一个中央团队（有时被称为 SOA 能力中心）来处理基础设施、架构和流程问题。这个组同时也负责建立 SOA 共识，以及钻研 SOA 并得出正确结论。中央团队的工作需要管理上的支持，因为除了时间和资源外，处理 SOA 在组织结构方面的影响还需要勇气（见第 8 章）。

理解、监管、管理支持和钻研是 SOA 成功的关键因素。关于监管和建立 SOA 的细节，见第 19 章。

OASIS SOA 参考模型 SOA 定义

见[OasisSoaRM06]

面向服务的架构（SOA）是一个范式，用于组织和利用可能处于不同所有权范围控制下的分布式系统。

……

“可见性”、“交互”和“效果”是描述 SOA 范式时的关键概念。“可见性”指的是需要能力的一方和拥有能力的一方相互之间可见。做到这点的通常手段是在如下方面提供描述：功能和技术需求、相关限制和政策、访问和回应机制。描述应该有一个格式（或能被转为某种格式），该格式的语法和语义应被广泛接受和理解。

如果说可见性使需求和能力（反之亦然）的匹配成为可能，那么，“交互”就是使用能力的行为。通常，经由消息交换的协调，通过一系列的信息交换和功能调用，交互得以进行。交互有许多方面，但它们都植根于特定的执行上下文环境——整套的技术和业务元素在需求方和能力方之间铺路。这使服务提供者和消费者之间的交互成为可能，并且，为可能生效的政策和约定提供了一个决策点。

使用能力的目的是达成现实世界一个或多个效果。本质上，交互是“一个行动”而非“一个对象”，交互的结果是一个“效果”（或“整套/系列”的效果）。

……

多数对 SOA 的讨论通常都将“松耦合”、“粗粒度”这些术语作为 SOA 的概念来使用。但是，此处的讨论有意避开这些术语，因为它们都是主观上的平衡，缺乏实用的度量标准。对需求和能力而言，粒度和“粗”通常与讨论的具体问题的层次相关，比如，一个问题偏战略性，另一个问题直达算法层次，孰粗孰细？再者，最佳层次的定义也没法按接口数量，或者某接口交换的信息的数量或类型这种简单方式进行量化。

注意，虽然 SOA 通常用 Web Service 实现，但是，也可以通过其他的实现策略使服务可见、支持交互、产生效果。基于 Web Service 的架构和技术是特定的、具体的。虽然本参考模型的概念可以用到这样的系统上，Web Service 还是过于偏向特定解决方案，从而不是通用参考模型的一部分。

2.5 SOA 不是银弹（译注 1）

SOA Is Not a Silver Bullet

SOA 已经被热炒到开始变成主流了。对于流行的新概念，总是存在这样的危险——人们会试图到处使用这些概念。人们总是喜欢试验酷而新的东西（虽然，从另一方面来看，他们又害怕改变），当每个人都说你必须使用 SOA 时，很容易掉进把所有东西都弄得像 SOA 的陷阱。（如果你有把锤子，所有东西看起来都像钉子。）

正如我之前所写，SOA 对特定的环境——具有不同所有者的异质分布式系统——而言是理想的解决方案。然而，正如你将在本书中看到的，处理异质和不同所有者问题，以及提供灵活性、可伸缩性和容错时需要付出代价。

Thomas Erl 在“面向服务的架构”中对 SOA 的定义

见[Erl05] 54—55 页

当代 SOA 体现了一个开放、可扩展、联合、可组合的架构，该架构促进面向服务，由自主、保证服务质量、分散化供应商、可互操作、可发现，以及潜在可重用的服务构成，用 Web Service 实现。

SOA 能建立业务逻辑和技术的抽象，达成这些领域之间的松耦合。

SOA 是过去平台的进化，它保留了传统架构的成功特性，带来了独特的原则，鼓励面向服务，支持面向服务的企业。

理想状态下，SOA 在企业内应被标准化，但是，达到理想状态需要计划周详的转换过程，也需要当前仍在发展中的技术集的支持。

……

当代 SOA 支持、鼓励、促进如下内容：供应商分散化，内生的互操作性，可发现性，联合，内生的可重用性，可扩展性，面向服务的业务模型，各抽象层次，企业范围的松耦合，以及组织级的敏捷性。

因此，如果没有描述过的那类系统，可以考虑不用 SOA。如果你能掌握所有的东西（即，“一个同质系统”和/或“没有不同的所有者”），对你来说，使用 SOA 可能代价高昂而收效甚微。例如，用 SOA 来区分前端和后端不是个好办法。当然，SOA 能帮你引入

译注 1：silver bullet，银弹。在欧洲传说中被认为是狼人和吸血鬼的克星，专门对付妖怪，具有驱魔的效力，甚至是杀死吸血鬼的唯一方法。后来被用来形容最强的一招，最有效的手段。

不同的水平层，但是，其他方法（比如，模块化和库）也许更合适一些。然而，即使你属于“不用 SOA 一族”，就系统抽象而言，规则、问题和原则并无二致。因此，就算你决定不用 SOA，也能从本书提出的各种论题中获益。

牢记你的目的——找到解决特定需求的合适方案，SOA 只是达到目的的一种手段。软件需求千差万别，无法生硬地堆砌到一块儿。开动你的脑子，而不是僵硬地遵循规则。

Eric Newcomer 和 Greg Lomow 在“理解 Web Service 下的 SOA”中对 SOA 下的定义

见[NewcomerLomow05] 54 页

面向服务的架构（SOA）是一种设计模式，在业务服务的整个生命周期中（从概念提出到业务终结），它指导了建立服务和使用服务的各个方面，它定义和规定了 IT 基础设施，使不同应用可以在交换数据和参与业务流程时不用考虑应用之下的操作系统或编程语言的差别。

2.6 SOA 不是特定技术

SOA Is Not a Specific Technology

许多 SOA 定义都包括 Web Service 这个术语，但是，SOA 和 Web Service 不一样。SOA 是个范式；Web Service 是运用特定实现策略，实现基础设施的可能方法。它们之间的区别很重要。

Web Service 正在成为实现 SOA 的事实标准。许多对 SOA 的讨论提出，SOA 应由 Web Service 来实现（例如，Thomas Erl 在他的定义中写道：“当代 SOA 体现了一个……架构，……由服务……构成，用 Web Service 实现”）。好消息是，这种标准的演进将带来更多的自由选择：除了使用专有技术，或自己从头建造所用东西之外，你还能以合理的价格买到已经做好的解决方案（也能得到开源方案）。

这并不是说用 Web Service 构建 SOA 能解决你的所有问题。Web Service 能在提供基础设施上助你一臂之力，然而，你还是必须搭建架构，以及设立成功的 SOA 所必需的复

杂流程。不光如此，Web Service 标准不像它看起来那样成熟，Web Service 会引入其他实现策略没有的一些问题。在第 16 章我会详细讨论这些问题。

注意，可以用其他技术（CORBA、MQ、Tibco，等等）来实现 SOA。我见过使用不同实现策略的 SOA（见 5.2 节）。

2.7 对比 SOA 与分布式对象
SOA Versus Distributed Objects

可以用很多不同的方式处理分布式系统。其中之一是使用分布式对象，CORBA 最初的概念就来源于此。这个想法是使外部系统的对象可以被远程访问。你能远程调用对象的方法，包括那些属性设置和属性读取方法。也就是说，每访问一次属性，你都在调用远程函数。

这类远程系统接口粒度过细，以该方法作为基础，结果就产生出这样的思想：用一个通用的业务对象模型覆盖整个分布式系统。

实践证明，这种方法不具备伸缩性。每一个接入的系统都采用同样的业务对象模型时，很难管理，也引入了过多的集中，过多的依赖。

实际上，SOA 的做法与分布式对象概念刚好相反。SOA 在不同系统间交换数据，每个系统都在本地（冗余）数据拷贝上工作，使用自己的本地方法和过程。与分布式对象不同，这种方法解耦系统，使系统可以伸缩。

注意

注意在 CORBA 2.3 版中引入了值对象，或传值对象（OBV），这使你能从一个系统中拷贝一堆数据（对象），然后对数据进行本地处理。你可以用这个技术实现 SOA（前提是你不使用包含操作的 OBV）。这种使用 CORBA 的方式与分布式对象概念无关。

2.8 SOA 术语
SOA Terminology

让我们稍微说说术语方面的问题。演化中的概念通常有如下特点：不同术语往往指同一事物，同一术语又可能包含多种不同意义。SOA 正是如此。书末的词汇表包含了绝大多数 SOA 重要术语的定义。

然而，在这儿我想介绍几个贯穿本书的术语。今后，当我们深入细节时，我会介绍其他术语。

首先，我们需要一些术语来描述系统通过服务通信时，各个系统所扮演的角色。我在本书用的术语是“供应者”和“消费者”：

- 一个“**供应者**”是实现一个服务（一个业务功能）的系统，这样其他系统就能调用它。
- 一个“**消费者**”是调用服务（使用被提供的服务）的系统。

通常还用另一个词——“**请求者**”——来描述消费者（尤其在 Web Service 世界中）。这是个更偏向技术的术语，该术语基于这样的事实：一个服务消费者以向提供者发出一个请求的方式调用服务（虽然情况并不总是如此，正如你在第 10 章会看到的）。我将在本书中使用“消费者”。

对分布式系统中的角色，你可能听说过，或者自己也在使用常用术语“客户机”和“服务器”。事实上，供应者就是服务的“服务器”，而消费者是使用服务的“客户机”。然而，因为“客户机”和“服务器”术语被用于如此多的不同场合，我倾向于建议你使用“供应者”和“消费者”。

如果想用一个统一的术语来称呼供应者和消费者，你可以用“**参与者**”。服务参与者是作为服务供应者或服务消费者（或者两者都是）的系统。再次指出，平常情况下还有替代这些说法的术语，例如，在 Java™世界中，出现了术语“**服务代理**”。

2.9 总结
Summary

- 这里是我对 SOA 定义的总结：

  SOA 是一个架构范式，它处理的业务流程分布在现存的或新的，范围巨大，处于不同的所有者控制下的异质系统上。

- SOA 关键的技术概念是服务、互操作性和松耦合。
- SOA 关键要素是基础设施、架构和流程（包括建立 SOA 的元流程，以及监管）。
- SOA 成功的关键因素是理解、监管、管理支持和钻研。
- SOA 既不是特定技术，也不是银弹。有 SOA 适合的领域，也有其不适合的领域。
- Web Service 是实现 SOA **基础设施**的可能方法之一。之所以通常推荐使用 Web Service，是因为它看起来正在被确立为标准技术。

第 3 章

服务
Services

本章我将探讨术语“服务”及其相关概念（比如接口和契约）的含义。

对 SOA 而言，术语“服务”有许多定义。因此，我将再次从现有的各种来源引用定义，比如“OASIS SOA 引用模型”、“Wikipedia.org”，以及某些书。然后再提出我个人对该术语的定义。

要对服务是什么获得完整的认识，你必须进一步深入阅读本书。本章的目的是提供一个一般的定义，以此作为本书后续讨论的基础。

3.1 服务
Services

OASIS SOA 引用模型[OasisSoaRM06]描述道：

> 名词“服务”在辞典中被定义为“某人为别人执行的事务（职能）”。

这定义含义宽泛、无所不包，或者反过来看，什么也没说。对 SOA 术语而言，很难找到一个“服务”术语的准确、有用的定义，因为现有的定义太多了。再次，我收集了一些定义，把它们的摘录放到贯穿本章的选读栏中。读这些定义的时候，你会发现一些通

用的短语、特征；然而，你也会发现各定义在背景、抽象层次，以及措词上差异巨大。和前一章一样，我从提出自己对服务的理解开始。

维基百科服务定义（2006 年 2 月 19 日）

一项“服务”（理想中）是一个自足的、无状态的业务功能，通过定义良好的标准接口，它接受一个或多个请求，返回一个或多个应答。服务也能执行离散的工作单元，如编辑和处理一个事务。服务不应依赖于其他功能或过程。用于提供服务的技术，如编程语言，不构成本定义的一部分。

3.1.1 服务体现了业务功能
Services Represent Business Functionality

SOA 聚焦于业务流程。这些流程在不同的系统上分不同的步骤（也称为“**活动**”或“**任务**”）执行。服务的主要目标是体现业务功能的“自然”步骤。这就是说，就服务起作用的业务领域而言，服务应该代表了一项自足的功能，对应着一项真实世界的业务活动。换句话说，业务人员应该能理解服务干了什么（见“业务驱动对比技术驱动业务接口”选读栏）。

当然，当你设计一个系统时，会出现如何找到和/或设计服务的问题。而且，和 IT 中经常出现的情况一样，这个问题没有简单的答案。这实际上是业务流程管理（BPM）要解决的问题，第 7 章将就此讨论。

3.2 接口和契约
Interfaces and Contracts

技术上，一项服务是一个处理（多个）消息的接口，它返回信息，以及/或者改变相关实体（后端系统）的状态。或者，如 Yefim V. Natis，SOA 术语发明人之一，Gartner 的分析师所写[Gartner03]：

> 本质上，SOA 是一个软件架构，从接口定义开始，将整个应用布局建立成接口、接口实现及接口调用的布局。SOA 叫做“面向接口的架构”其实更合适。

根据这个定义，则很难把“服务”定义得更具体些。所有能够当作接口使用，体现了自足的业务功能的东西都是一个服务。

业务驱动对比技术驱动业务接口

注意，业务驱动的接口不同于技术驱动的业务接口。例如，一个技术驱动的业务接口可能定义成如下方式：

```
customerOP(action,        // "create", "read", "change", "delete"
           id,            // customer id or null
           name,          // new customer name or null
           address,       // new customer address or null
           account)       // new customer bank account or null
```

这是个业务接口，因为它允许你创建一个新的客户，读取或修改他/她的数据，或者删除数据。然而，这个接口是技术驱动的，也许直接映射自相应的数据库接口。

作为对比，一个业务驱动的接口可能看起来像这样：

```
createCustomer(name,        // new customer name
               address,     // customer address
               account)     // customer bank account

readCustomer(id)            // customer id

changeCustomerAccount(id,          // customer id
                      account)     // new customer bank account

changeCustomerAddress(id,          // customer id
                      address,     // new customer address
                      check,       // true: verify address
                      modify)      // true: fix address if possible

deleteCustomer(id)                 // customer id
```

注意如下区别：

- 不是只有一个函数签名，而是有五个。每个操作的名字都从业务视角描述了基本功能。另外，对每个单独功能来说，没有什么参数是无意义的（例如，为了删除一个客户，你只需要客户 ID）。
- 读写操作有不同层次的粒度。在本例中，你可以单独改变一个客户的地址或银行账号，但是，当你读取客户数据时，你同时取得这两个数据。对于业务流程而言，这样的区分是典型的。
- 特定操作可能有额外的参数。在本例中，改变地址有额外的属性，指明了是否应该进行地址检查，以及根据地址检查的结果，是否应该修改地址（例如，将“NewYork”换成“New York”）。

实际上，如果你让业务人员逐项逐项地设计接口，这些接口会和靠底层数据库设计驱动的接口看起来不同。面向服务明确地偏向业务驱动而非技术驱动的接口，虽然这并非意味着你不应考虑可能的协同效应（译注 1）。开始设计时，要基于需求而非基于实现。

然而，从技术上和语义上来看，都存在各种各样的接口。一种接口是“**签名**”，它描述了输入参数、输出参数，以及可能的异常。

然而，仅仅知道签名还不足以使用服务。作为服务的消费者，你还想——实际上是必须——知道服务完整的行为和语义。换句话说，接口必须是“**定义良好**”的。

定义良好的接口，其行为必须是明确的，但在实际中，这个目标很难达到。另外，服务的某些方面可能和特定消费者相关，比如，和服务质量（QoS）及服务等级协议（SLA）（注 1）相关的属性。因此，你可以把相关信息安排在“**契约**”中。一份契约，是对特定供应者和特定消费者之间的服务的完整规格说明。从消费者的角度来看，它定义了“当使用服务时你应知道的一切”，这样就（理想中）不会再有疑问。

实践中，对服务的描述往往从定义良好的接口开始。然后，当某个特定的客户想要使用服务时，你再基于定义良好的接口做出特定的契约。该契约反映出，当你遵守特定的非功能性承诺时，需要提供给服务的必要的资源。例如，虽然所有消费者都使用（或能使用）同样的正式接口来调用服务，在不同的时间，各消费者发起调用的次数可能不同。

举个例子。一个电话呼叫中心前端每天可能调用 100 000 次某服务，峰值出现在早上 8 点到晚上 6 点之间。另一方面，一个在夜间运行批处理请求的消费者，则可能必须在午夜到凌晨 2 点之间发起 50 000 次调用。对所有这些 SLA，提供足够的资源来满足调用是服务提供商的任务。服务基础设施也可能提供对不同服务调用排优先级的能力，这样

译注 1：Synergy effects，协同效应，指企业生产、营销、管理的不同环节，不同阶段，不同方面共同利用同一资源而产生的整体效应。如并购后竞争力增强，导致净现金流量超过两家公司预期现金流量之和，或者合并后公司业绩比两个公司独立存在时的预期业绩高。此处作者指设计出业务、技术两方面都好的接口。

注 1：“服务等级协议”中的“服务”这个词，与这里作为 SOA 的概念引入的“服务”这个词，其意义并不相同。SLA 指一个协议的服务等级，而不是一个服务的协议等级。换句话说，当谈论服务的 SLA 时，我们谈的是关于这些服务的服务等级的协议。

一来，相比其他消费者而言，一些消费者能得到更好的响应。在这种情况下，不同响应时间也可能成为各契约的一部分。

交换“消息”在技术上是何意义，这是个不同的问题。原则上，被称为“消息交换模式”的模式可能有多种。例如，一个服务可能按同步调用方式提供，此时消费者运行阻塞，等待应答；或者服务可能包含一个通知，发给订阅了的消费者，通告他们关于状态的变化（这可能叫做也可能不叫做一个“事件”）。在后一种情况下，不要求进行应答。消息交换模式的细节请见第 10 章。

3.3 额外的服务特性
Additional Service Attributes

如果你通览本章所有的服务定义（以及本书之外的其他定义），你会发现它们提出了许多服务“可能”、“必须”，或者“应该”具备的额外特性。然而，对服务是否真的需要这些额外特性，人们还未达成共识。以下是一些典型的问题，不同的人给出的答案不尽相同。

- 服务必须是无状态的吗？
- 服务必须被实现为 Web Services 吗？
- 服务应该是粗粒度，而且/或者是可组合的吗？

在此必须要小心。我对服务的定义是，它应该是由一个（定义良好的）接口定义，针对一项业务功能的 IT 表示法。典型地，服务也应该是自足的。在服务上通常还能应用其他一些特性，但任何其他特性并非不可或缺。

实践中的情况有点不同。在具体的 SOA 中，你可能需要一些——但不是全部——我们在此讨论的额外特性，你也可能发现，某些种类的服务有其他服务没有的一些特性。因此，须根据服务的特性来分类不同的服务（细节见第 6 章）。

就服务可能具备的特性而言，其细节有可能相当复杂。贯穿本书，每章在大标题限定的范围下会讨论不同的特性。然而，为了给大家一个最初的整体印象，我在这里把它们介绍一下。在合适的地方，我会让你参考本书的其他部分，你能在那儿找到进一步的细节。

3.3.1 自足
Self-Contained

所有 SOA 定义都同意，服务自足（独立的、自主的、自给自足的）是个设计目标。然而，总会存在某种形式的依赖。例如，服务可能至少共用一些基本的数据类型（如字符串）。尽管如此，要当心对更复杂的基本数据类型的依赖。目标是最小化依赖，这样 SOA 才能适用于所有者各异的分布式系统。细节见第 4 章。

维基百科服务定义（2006 年 7 月 17 日）

系统架构服务：

在企业架构上下文环境（换句话说，系统或组织的架构或建模）中，“服务”术语意指离散定义、彼此相邻、自身自主的业务或技术功能集。

……

“服务”有一个定义，该定义包括：

1. 详细清楚的叙述性定义，为底层（但并非详细的[未达到实施程度]）业务模型所支持。
2. 有一系列性能指标关注于度量/性能参数，如可用性（组织中的成员应该何时能执行功能）、持续时间（执行功能需要多长时间）、频率（一段时间内功能被执行得有多频繁），等等。
3. 与组织的信息模型有关联，指示出“服务”拥有（创建、更新和删除）哪些信息，以及它引用了哪些属于别的“服务”的信息。
4. 一张下游消费者（需要依赖本服务功能和信息的其他“服务”）列表，以及它们的需求文档。

3.3.2 粗粒度
Coarse-Grained

服务是种抽象，向消费者隐藏了实现细节。但是，你要为这种抽象付出代价：运行时变慢了。例如，从远程存储方法切换到服务调用，可能会减慢数据访问速度 5 到 10 个因数（见 13.2 节）。因此，在供应者和消费者之间用一个服务调用传输所有必需的数据，比起用多个服务调用处理同等数量的数据，通常说来是更好的办法。

另外，粗粒度也有助于分离服务供应者的内部数据结构和外部接口。对服务属性的每次存取都使用一次服务调用（也就是说，每个 setter 和 getter 都用一个服务）会导致分布式对象（见 2.7 节），这就增强了分布式系统之间的依赖。

有的定义建议服务应该永远都是粗粒度的，但是，这种建议的问题是，怎样才能确定粒度够“粗”呢？因此，OASIS SOA 参考模型（见[OasisSoaRM06]）如此描述：

OASIS SOA 参考模型“服务”定义

见[OasisSoaRM06]

名词“服务”在辞典中被定义为“某人为别人执行的事务（职能）”。然而，服务这个术语在人们通常的理解中，还结合了如下一些关联想法：

- 为别人做事的能力
- 向别人提供的工作规格说明
- 为别人做事的提议

这些概念强调了能力和把能力发挥出来的本领之间的区别。SOA 的需求和能力是两个独立的存在，在 SOA 中，服务是把需求和能力结合到一起的机制。

……

调用一个服务的结果是在现实世界体现出一个或多个作用。这些作用可能包括：

1. 作为对请求某些信息的回应，相应信息被返回。
2. 规定实体的共享状态被改变，或者
3. （1）和（2）的某种组合。

> 多数对 SOA 的讨论通常都将术语“松耦合”、“粗粒度”作为 SOA 的概念。但是，此处的讨论有意避开这些术语，因为它们都是主观上的平衡，缺乏实用的度量标准。对需求和能力而言，粒度和“粗度”通常与讨论的具体问题的层次相关，比如，一个讨论战略性问题，另一个深入到了算法层次。而且，定义最佳层次时，也没法按简单方式量化，比如，数接口数量，或者数某接口交换的信息的数量或类型。

另一个与粗粒度建议相关的问题是性能方面的考虑。处理大量数据需要时间，如果消费者不需要这些数据，时间就浪费了。细节见第 13.3 节。

3.3.3 可见/可发现

Visible/Discoverable

要调用一个服务，你必须知道有这个服务。通常，有一个公共的地方供你搜寻服务，以及/或者描述了服务的所有细节。然而，也可以通过口口相传的信息来发现服务。关于这个论题第 17 章有更多的讨论。

3.3.4 无状态
Stateless

有时可以把服务定义为（理想中）无状态的。无状态是服务最让人困惑的方面之一，因为实际上总会包含某些状态。问题是状态在哪儿，持续多长时间，以及它是技术状态还是业务状态。第 15.1 节讨论了这个问题。

3.3.5 幂等性
Idempotent

如果你调用一个服务来修改一些东西，而又没有收到肯定的答复，这就有麻烦了。是修改实际发生了，只是你没有收到结果，还是在修改实际发生之前服务调用就失败了？如果还不确定，可以再次发出服务调用而不会造成问题，你的服务就是**幂等**的。这就是说，幂等性是当你不确定服务是否完成时重做操作的能力。这使你的工作更加轻松，所以，有幂等的服务始终是个目标（如果代价不过分高昂的话）。然而，在某些情况下，这个目标很难达到。细节见 10.4 节和 15.2 节。

3.3.6 重用
Reusable

避免冗余是软件开发的一个普遍目标。这就是说，理想中每项功能只应实现一次。SOA 给业务带来的一个普遍情况是，重用性变得更好了，因为所有需要一定功能的系统只需调用同样的服务。这是可能的，但也有其局限性。一个局限性与性能相关（第 13 章给出了应该考虑的“性能/可重用性”平衡方面的几个例子）。因此，可重用性也许是个目标，但不是个法则。

3.3.7 可组合
Composable

服务可以“使用/调用”其他服务。这就是说，更广意义上的业务功能可以被分解为小的步骤，所有的步骤也是服务。这也是为什么 SOA 往往包括一类“组合服务”（细节见 6.3 节）的原因。从业务角度来看，组合服务及分解业务过程带来了业务流程建模，第 7 章讨论了此问题。

3.3.8 技术性
Technical

有时，SOA 定义也允许所谓的“技术性服务”，它不一定体现自足的业务功能。例如，如果你有了 SOA 用于业务服务的基础设施，那么，你可以把这个基础设施用于交换技术数据。在此情况下，你可能会争论说这个就是技术性服务，或者因为知道业务服务属于不同的抽象层次，争论说你用的是 SOA 基础设施本身。只要合适，怎么干都行，但是，要记住 SOA 的目标集中在业务功能上。细节见 6.6 节。

Thomas Erl 对“面向服务的架构”中服务的定义

见[Erl05] 54 页

当代 SOA……由自治、服务质量使能、供应商分散、可互操作、可发现，以及潜在可重用的服务组成，由 Web Services 实现。

3.3.9 服务质量和服务等级协议使能
QoS- and SLA-Capable

如前所述，一个服务契约要包含调用者调用服务所应了解的一切。正因如此，迟早你会开始阐明非功能性特性，这包括与运行时性能、可靠性，以及相关的服务质量和服务等级协议。然而，开始时，即使没有这些特性，SOA 也能工作。（相信我，消费者会告诉你服务是否足够快、足够可靠！）

3.3.10 前提和后置条件
Pre- and Post-Conditions

根据“契约式设计”的思想（由 Bertrand Meyer 发明），前提和后置条件有助于阐明服务的语义。前提条件定义了调用服务时，服务消费者必须履行的特定义务（“陈述限制”）。后置条件保证当服务成功运行时，系统具备特定的性质和/或输出（“陈述成果”）。细节见[DesignByContract]。

3.3.11 供应商分散
Vendor-Diverse

SOA 是为大型分布式系统而生的概念，这些系统通常都非常异质化（见 2.2.3 节）。因此，实现服务时，通常都要求能够使用不同平台和不同产品。然而，这更多的是 SOA 的性质而非服务的特性。

3.3.12 可互操作
Interoperable

高互操作性是 SOA 的核心需求。从服务能被任何其他系统调用来说，服务有内在的互操作性。但是，这更多的也是 SOA 的性质而非服务的特性。

Eric Newcomer 和 Greg Lomow 在“理解 Web Services 下的 SOA”中对服务的定义

见[NewcomerLomow05] 58 页

从业务观点来看，服务是 IT 资产，它对应着真实世界的业务活动或可识别的业务功能，遵照服务政策，服务可以被访问。

……

从技术观点来看，服务是粗粒度、可重用的 IT 资产，拥有定义良好的接口（也就是服务契约），清晰地区分了服务的外部可访问接口和服务的技术性实现。

3.3.13 实现为 Web Services
Implemented As Web Services

正如前一章所讨论的，许多 SOA 的定义声称 SOA 应由 Web Services 来实现。在第 16 章我会讨论这个建议是否有意义。现在，我只想重申，Web Services 和 SOA 不同。Web Services 可能是个实现 SOA 的（明显的）方法，但 SOA 不是个绑定在任何特定技术上的概念。

3.4 总结
Summary

- 如下是我对在 SOA 上下文环境中“服务”定义的总结：

  服务是某些自足的业务功能的 IT 实现。
- 依靠关注于业务方面，服务隐藏了技术细节，使业务人员能够对它进行处理。
- 在技术上，服务是个接口，用于在供应者和消费者之间交换（多个）消息。
- 从消费者的角度来说，对服务的完整描述（签名和语义）被称为“定义良好的接口”或“契约”。契约是在某个供应者和某个消费者之间单独约定的，通常涵盖了非功能性方面——如服务等级协议——的要求。
- 服务可能（或者，按照某些定义，必须）具备一些特性。根据我的理解，有哪些特性视情况而定；不同服务几乎总有些不同特性，服务也应依据这些特性来分类。

第 4 章

松耦合

Loose Coupling

SOA 应用于大型分布式系统。对于这样的系统来说，可伸缩性和容错性是可维护性的关键。另一个重要的目标是：最小化修改、故障对系统景观在整体上的影响。这样一来，松耦合就成了 SOA 的关键概念。

本章会讨论松耦合概念的动机（针对大型分布式系统），从技术和组织角度探讨松耦合的不同变体。这个话题会展现出，当设计大型系统时，SOA 是个会带来特殊的优先级考虑的范式。然而，再一次，没有规则用于限定使用松耦合的类型或层次，你必须根据自己的特定环境做出决策。

4.1 对容错的需求

The Need for Fault Tolerance

我们生活在疯狂的时代。市场统治着一切，这意味着你通常不会有足够的时间来搞一个精心制作的、健壮的系统设计。如果你不够迅速，不够灵活，不够便宜，很快就会发现自己被挤出了市场。所以，你需要迅速、灵活和便宜的方案。

然而，迅速和便宜是与设计良好、坚固健壮有矛盾的。结果就是，你不得不处理错误和问题。这里的重点是容错。最重要的事情是你的系统要能运转。根据[ITSecCity02]，一个航班订票系统故障会导致每小时 10 万美元的损失，一个信用卡系统崩溃会导致每小时 30 万美元的损失，一个股票交易系统机能失常会导致每小时 800 万美元的损失。正如这些数字所示，对大型分布式系统来说，容错是关键。当问题发生时，最小化问题的影响和后果很重要。

4.2 松耦合的形式

Forms of Loose Coupling

松耦合这个概念主要用来处理可伸缩性、灵活性和容错这些需求。松耦合的目标是最小化依赖。当彼此的依赖更少时，一个系统的修改，或一个系统发生的错误，会更少地影响其他系统。

松耦合是个原则，它既不是个工具，也不是个检查清单。当设计自己的 SOA 时，规定引入何种松耦合，运用到何种程度，这些都取决于你自己。然而，当考虑自己系统中的松耦合问题时，有一些典型的问题是你可能需要考虑的。表 4-1 列出了这些问题（该表引自[KrafzigBankeSlama04] 47 页所发布的表）。

表 4-1：SOA 松耦合可能的形式

	紧耦合	松耦合
物理连接	点对点	通过中介者
通信风格	同步	异步
数据模型	公共复杂类型	只有简单的公共类型
类型系统	强	弱
交互模式	通过复杂的对象树导航	以数据为中心、自足的消息
业务逻辑控制	集中控制	分布式控制
绑定方式	静态	动态
平台	强平台依赖性	平台无关
事务性	2PC（两阶段提交）	补偿
部署	同时进行	非同时进行
版本划分	显式升级	隐式升级

这个列表远远谈不上完整，但对于大型分布式系统来说，它已经相当典型了。再次注意，这不是个检查单。没有什么 SOA 认证能保证，当所有或至少 50%松耦合形式被使用时，你就符合要求了。然而，如果在你的 SOA 中，这些松耦合形式的任何一种都没有被使用的话，那就很奇怪了。如果可以不用任何松耦合的话，看来你的系统也没有大型分布

式系统的普遍需求。那你就不应该把自己的方案称为 SOA。（这对你获取资金和资源可能有帮助，但请注意不要留给人错误的印象。）

既然有了这样美妙的一张单子，谈到了松耦合的方方面面，并且减少依赖也是好东西，你可能会感到疑惑：为什么不简单地在每一个 SOA 中应用松耦合的所有形式呢？答案是，松耦合要付出使系统更加复杂的代价。松耦合意味着更多的开发以及/或者维护工作量。

为了探讨表 4-1 所列的各种形式的松耦合有怎样的帮助，会导致何种成本，让我们对其中一些形式作进一步讨论。本书后续部分会对其中绝大部分论题作更深入的讨论，所以，在合适的地方，我加入了对后续章节的引用。

4.2.1 异步通信
Asynchronous Communication

松耦合最广为人知的例子也许就是异步通信（见图 4-1）。异步通信往往意味着消息的发送者和消息的接收者没有被同步。可以把异步通信想象成发送电子邮件。在你发出一条消息后，你可以在等待回复的同时继续自己的工作。当你发送消息时，接收者可能不在（在线）。当她连线时，消息被递送，她可以处理消息，如果需要，发送一个回复（当她发送消息时，不要求你也在线）。

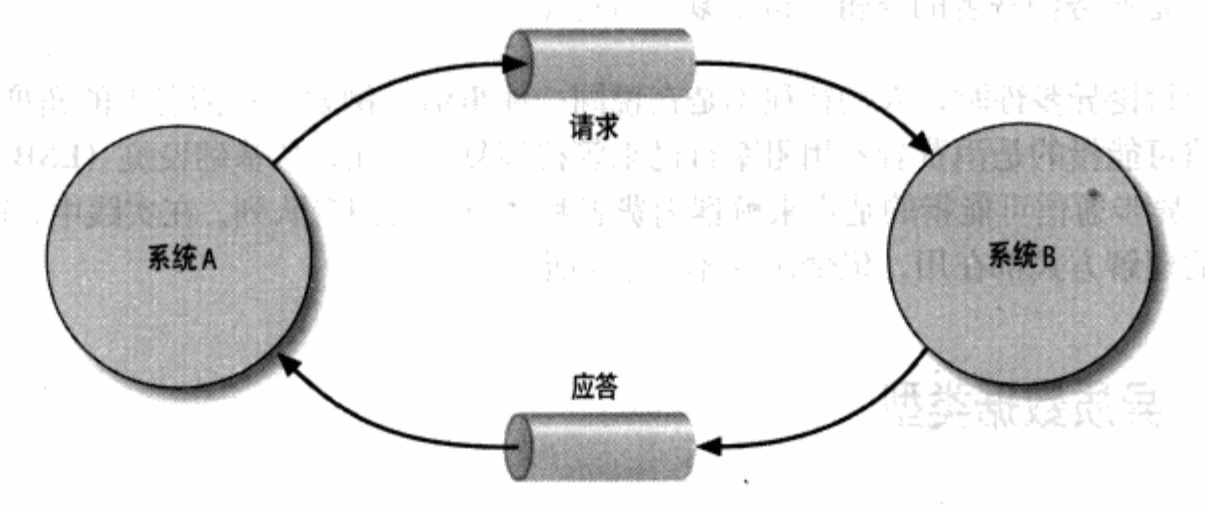


图 4-1：通过异步通信解耦

异步通信的一个难题发生在当发送者需要一个答复的时候。使用异步通信时，你不能马上得到对自己消息的答复。因为你不知道何时（或者是否）能有答复，你继续做自己的工作，开始执行其他任务。然后，当应答到来时，你必须用合适的方式处理它。这意味着你必须把答复和最初的请求关联起来（例如，通过处理类似于“关联 ID”这样的东西）。另外，你必须处理答复，而这通常需要请求发出时的初始状态和上下文环境方面

的知识。将应答和请求相关联，以及把状态从请求转换到应答都要费些工作量。

注意

对不同消息交换模式（MEP）的讨论见第 10 章，关联 ID 的细节见 15.3.3 节。

当发出大量异步消息时，又有新的情况发生。你收到应答的顺序可能不同于消息发出的顺序，并且，某些被等待的应答可能无法到达（或及时到达）。对所有可能性进行编程、测试和调试将非常复杂和耗时。

我的一个客户要做出数以百计的服务，正是由于上述原因，该客户有条政策规定，当请求需要一个答复时要避免异步性。由于发现对竞争条件（由不同的、无法预料的应答时间造成的情况）进行调试是场噩梦，同时深知大型分布式系统的关键是可维护性，所以该客户决定减小陷入这种情况的风险。该决策有得有失，因为系统性能可能不如有更多异步通信时那样好。

正如上面的讨论所揭示的，在 SOA（或一般的分布式系统）中引入异步通信有利有弊：

- 好处是互相交换服务消息的系统不用同时在线。另外，如果需要答复，长时间后才答复也不会阻塞服务的消费者。
- 弊端是服务消费者的逻辑变得更复杂（得多）了。

注意，当讨论异步性时，人们往往不是在说同一件事情。例如，从消费者的角度来看，异步通信可能指的是消费者不用阻塞自己来等待答复，然而，从基础设施（ESB）的角度来看，异步通信可能指的是用来解耦消费者和供应者的消息队列。在实践中，这两种对概念的理解方式都在用，但情况并不总是如此。

4.2.2 异质数据类型
Heterogeneous Data Types

现在，让我们来讨论我最喜欢的松耦合例子：在分布式系统上对数据类型进行一致化。这个话题总是引起大量争论，而且，理解它是理解大型系统的关键。

毫无疑问，如果数据类型在所有系统间共享，生活会变得轻松很多。正由于此，一致化数据类型是个“自然的”方法。实际上，在面向对象成为主流后，弄出一个公共的“**业务对象模型**”（BOM）就成了一个普遍的目标。但是，人们后来发现，对大型系统来说，这个方法是自求灾难。

无法进一步一致化真是丢脸

前段时间我做了一次关于 SOA 的演讲，其间我讲了自己一贯的论点——在大型系统中，你必须承认无法一致化数据类型。在咖啡时间，一位高级系统架构师走过来说，“在我们的工业中，当不能进一步一致化数据类型时，坏事就要来了，这很糟糕。过去我们可以分析和设计自己的系统。”这话听起来像是对只会做一件事情——拷贝和粘贴——的年轻一代的批评。

我的回答如下：“你可能是对的。在我们这个疯狂的时代，就是没有足够的时间来进行长期、仔细的分析和设计。市场营销人员统治着世界，如果不能按时交付，就会被挤出市场（即使获得了更好的质量）。但是要小心，不要低估我们现在靠连接系统达到的复杂度水平。记住，不再有中央控制了，并且，即使有也无法顺利运转。如果在发布互联网协议之前就地址类型进行了一致化，也许互联网永远无法成为现实。大型系统只需要提供最低程度的一致就能成功运转。注意，即使系统在运转中，仍然可以对数据进行一致化。”

导致灾难的第一个原因是组织方面的：人们就是无法对一致化类型达成统一意见。不同系统的观点和利益太不一致。因为大型系统往往属于不同所有者，达成统一意见很困难。要么你不能满足所有的利益，要么你的模型变得过于复杂，或者模型干脆就无法完成。这是“分析致瘫”的典型案例：如果你想在分析需求时达到完美，你就永远无法完成工作。

你也许会说，解决上述问题的方式是引入一个中央角色（一个系统架构师或一个“模型专家”），他解决所有悬而未决的问题，这样一来，一个拥有一致化数据类型的公共 BOM 就成为现实了。即使如此，你还是会碰到另一个基本的难题：不同系统按不同方式增长。假设你为客户创建了一个一致化的数据类型。后来，计费系统可能需要两个新的客户属性用于处理不同的税率；CRM 系统可能引入了一种新的电子地址格式；报价系统可能需要某些特性来处理隐私保护问题。如果客户数据类型在你所有的系统间共享（包括那些对所有这些扩展毫无兴趣的系统），所有系统都不得不相应升级以反映出每个变化，客户数据类型会变得越来越复杂。

迟早有一天，一致化的代价会变得过高。从时间和资金的角度来看，保持所有系统同步的成本太高。即使你努力做到了，下一次公司合并又会引入异质！

公共 BOM 无法伸缩，因为它们导致系统太紧密的耦合。结果就是，你不得不接受大型分布式系统上的数据类型无法一致化的事实。在解耦后的大型系统中，数据类型各不相同（见图 4-2）。

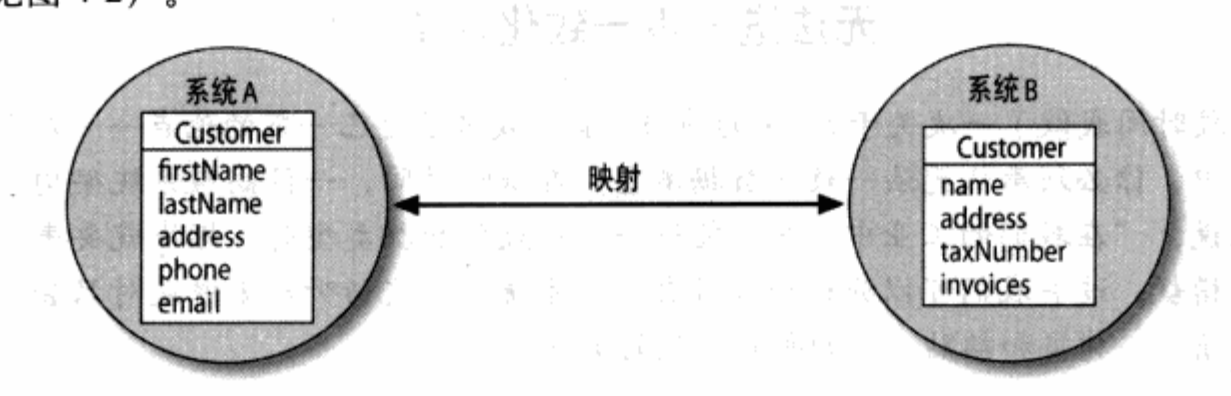


图 4-2：用不同数据类型解耦

再次，这个决策要付出代价：如果数据类型没有一致化，你需要“**数据类型映射**”（包括技术和语义方面）。虽然映射增加了复杂性，但是，在大型系统中这是个好现象，因为它展示出组件已经解耦。

通常的办法是，服务供应者定义自己提供的服务中所使用的数据类型（可能要遵循一些常用的约定和限制）。服务消费者必须接受这些类型。注意，服务消费者应避免在自己的源代码中使用供应者的数据类型。相反，消费者应有一个薄薄的“**映射层**”，将供应者数据类型映射为自己的数据类型。关于此点为何重要的详细讨论，请见 12.3.1 节。

再次，在 SOA（或一般的分布式系统）中引入这种形式的松耦合有利有弊。不用公共业务数据模型有优点也有缺点：

- 优点是，系统可以修改自己的数据类型，不会对其他系统造成直接影响（修改后的服务接口只影响相应的消费者）。
- 缺点是，你不得不从一个系统向另一个系统映射数据。

注意，你还是需要在不同应用间共享一些基础数据类型。但是，为了提升松耦合，通常说来，针对所有服务进行一致化的基础数据类型必须非常基本。我见过某个电话公司在 SOA 景观中引入的最复杂的公共数据类型是电话号码数据类型（一个包含了国家代码、地区代码和当地号码的结构/记录）。一致化公共地址类型（客户地址、发票地址等等）的企图失败了。一个原因是无法对如何处理贵族称号达成一致。另一个原因是不同系统和工具对如何在信件和包裹上处理和打印地址，其限制各不相同。

注意

如果你对一致化程度之低感到惊讶，想想同时对所有系统修改基本类型并大规模展开修改意味着什么（细节见 18.4.9 节）。实践中，基础服务数据类型必须稳定。

这是否意味着在 SOA 中没法拥有经过一致化的地址数据类型？不一定。如果你能够一致化，就进行一致化，一致化有帮助。然而，不要掉进“**要求**”数据类型一致化的陷阱。这种方法不能伸缩。

如果你不能一致化地址类型，这是否意味着所有消费者都不得不处理多个地址类型之间的差异？不。SOA 通常采用的方法是，引入一个组合服务，让你能够查询和修改地址（第 6 章讨论组合服务）。然后，该服务通过适当的映射数据，从而在各个后端系统间处理数据类型差异。

注意，采用这种办法，仍然不需要对地址有一个公共的看法。如果有新的需求，你可以简单地引入另一个地址服务，把额外的属性映射到不同的后端系统上。现有的、没有额外需求的消费者不会受到影响。

4.2.3 中介者
Mediators

松耦合的第三种形式和消费者执行的服务调用如何找到处理这个请求的供应者相关。用“点对点”方法，发送者使用物理地址，将请求发送到一个特定的物理系统。这就像把信件发送到一个特定邮件地址，如纽约百老汇大街 42 号一样。

这是种紧耦合的方法。如果接收者搬家了怎么办？如果接收者状态不佳，或者被太多消息淹没了该怎么办？需要紧急备援和负载平衡的机制。这就是说，需要某种类型的中介，它可以在不同的物理接收者之间切换。

原则上可以有两类中介者。

- 第一种类型在你发出服务调用之前就告诉你正确的服务端点。这是指，你仍然使用点对点连接，但在这些中介者的帮助之下，把服务调用发送到不同地址。这样的中介者通常被称为“**中间人**”或“**名字服务器**”。你用一个符号化的名字请求服务，中间人或名字服务器告诉你把请求发到哪个物理系统。这种形式的中介者要求消费者端的智能更多些。

- 第二种类型在消费者发出调用之后，为请求选择正确的服务端点。在这种情况下，消费者把请求发给一个符号化名字，基础设施（网络、中间件、ESB）根据智能路由规则，把调用路由到适当的系统。

实践中，两种形式都有丰富多彩的变体。例如，有用广播方式发送消息的服务总线，此时发送者将请求发送到一个逻辑接收者，任何提供所请求服务的供应者都能处理该调用。另一方面，Web Services 在技术上是点对点连接，在此你通常使用中间人来找到一个请求的物理端点，并且/或者你插入些所谓的“拦截器”，在运行时把请求路由到某个可用的供应者。

第 5 章将讨论在 ESB 中，如何处理中介者的细节。

4.2.4 弱类型检查
Weak Type Checking

松耦合复杂性的另一个好例子，与是否及如何检查类型这个问题相关。我们大多数人可能都学过，越早发现错误越好。因此，有类型检查的编程语言看起来比没有这种检查的要好（因为它们在编译时而非运行时检查可能的错误）。

然而，当系统增长时，情况变化了。难题在于类型检查既耗费时间，同时也需要信息。要想使 SOA 基础设施（ESB）检查类型，必须有关于这些类型的某些信息。举例来说，如果用 XML 来描述类型，ESB 就需要相应的 XML 大纲文件。结果是，任何对接口的修改不仅影响供应者和消费者，还影响 ESB。这意味着需要机制和过程来同步 ESB 的更新。如果 ESB 对每个供应者和消费者都使用适配器，你可能就不得不针对所有适配器组织和部署这些更新。这可以办到，但比起接口变化只影响供应者和消费者的政策来说，这导致更紧密的依赖。因此，让 ESB 通用化是个好主意。一般说来，接口变化只应影响把该接口用作契约的部分，而非传输相应数据的部分。如果互联网必须验证接口的正确性，互联网就无法伸缩了。

现在，你可能已经得出结论说，在大型系统中，永远都应使用通用接口，而非严格的类型检查。极端的方法是，对每个系统只引入一个通用服务，这样一来，服务接口就永远不变：你的所有变化都是针对接口的实现。然而要注意，在某些阶段，你的系统将必须处理数据，而且，你是否可以依赖数据一定程度上的语法正确性，对你的代码有影响。所以，在处理业务数据的场合，强类型检查有帮助。唯一的问题是数据有多稳定。

假设你有个交换字符串属性的服务。如果属性相当稳定，则推荐采用显式指定每个属性的强类型服务 API。当增加一个新属性时，你可以引入一个有额外属性的新（版本）服务。另一方面，如果属性经常变化，键/值链表可能就更好些。对这个问题的正确抉择依赖于如此多要素，以致制订通用的规则没有什么意义。对这类场景而言，你应在系统不断演化的同时，逐渐找到恰当的耦合程度。然而，要注意，对属性的显式建模，使我们更容易编写代码来处理数据（例如，当使用所谓的配制引擎来组合服务时，你可以轻松地在不同格式间映射数据）。反过来说，当必须具备二进制兼容性时，没有通用方法就是非常严重的错误了。例如，为所有消息提供的技术性头部永远都应具备一定的扩展能力，年复一年的扩展不应破坏二进制兼容性。

4.2.5 绑定方式
Binding

绑定（将不同模块或翻译单元的符号关联起来的任务）作为一种松耦合形式，与弱类型检查相似。再次，你必须在早绑定和晚绑定之间作出抉择，前者在编译时或启动时进行验证，后者在运行时验证。

4.2.6 平台依赖
Platform Dependencies

在表 4-1 中，有两种耦合形式用来处理是否应用了某种平台限制的问题。就一般的平台依赖性作出决策比较容易。当然，如果更偏向平台独立方案，而非平台依赖性较强的方案的话，你作出选择的自由度也更高。接下来讨论第二种形式。

4.2.7 交互模式
Interaction Patterns

一种特殊的平台依赖性与在服务签名中应用哪种交互模式相关（也就是说，对设计服务接口提供怎样的编程范式）。哪些是基础数据类型，以及怎样把它们结合起来？在此必须考虑大范围内的问题：

- 是只支持字符串，还是也支持其他的基础数据类型（整数、浮点数、布尔型、日期/时间类型，等等）？
- 是否支持枚举（有限范围内的经命名的整数集）？
- 能限制可能的值（例如，定义某种字符串格式）吗？
- 能创建更高级的类型（结构、记录）吗？
- 能创建序列类型（数组、链表）吗？
- 能设计类型之间的联系（继承、扩展、多态）吗？

编程结构越复杂，你就能在越抽象的层次上编程。然而，在映射数据时，如果平台对特定结构没有内生的支持，你也会碰到更多的麻烦。

基于我自己的经验（以及别人的经验），我建议你构建一个包含基础数据类型的基本集，从它出发，你可以组合出其他类型（结构、记录）和序列（数组）。小心枚举（这是当年每字节都很金贵时产生的时代错误——用字符串代替它）、继承和多态（即使 XML 支持它）。

一般说来，对类型要保守，因为一旦你不得不支持某种程序语言结构，即使需要的工作量（包括日志和调试能力）非常高，你也无法停止这种做法。对数据类型的更多讨论，见 15.5 节。

4.2.8 补偿
Compensation

补偿是种有趣的松耦合形式。补偿与事务安全问题相关。如果你必须更新两个不同的后端系统，让他们保持同步，你如何才能避免只有一个更新成功时产生的不同步问题？解决这个问题的通常办法是，使用称为“**两阶段提交**”（2PC）的技术，创建一个公共的事务上下文环境。用这种方法，你首先在两个后端系统上执行除了最后一步“切换到更新后数据”的所有更改；然后，如果没有系统报告有错，最后的提交执行在两个系统上的更新。

2PC 是中间件被最为爆炒的特性之一。只要是对中间件的评估，是否支持 2PC 的问题就会被提出。然而，实践中的大型系统很少用 2PC，因为所有后端系统都必须支持它，所以它带来了一些编程工作量，并且绑住了一些资源。主要的问题在于所有系统都必须在线，而且，直到更改在最后的系统上完成之前，所有系统都必须提供资源。尤其是当存在同时发生的数据存取时，这会导致延迟和死锁。

既保证总体一致性，同时耦合程度也更松的方法是补偿。使用这种方法时，你先修改第一个后端系统，再修改第二个后端系统；如果只有一个修改是成功的，你就需要针对具体问题，适当地作出“补偿”。例如，你可以回滚成功的修改，从而恢复修改前的一致状态，或者，向某个错误监控桌面机发出问题报告，这样，某人就可以检查问题的细节，手动处理问题。

补偿的优点是，对系统的更新不必同步进行（当更新有的后端系统时，它们甚至可以不在线）。缺点是你必须显式提供和调用服务，回滚之前调用的服务的影响，或者，你必须提供程序用于手动错误处理。

BPEL 是 Web Services 的流程执行语言，它直接支持补偿（见 7.4.2 节）。

4.2.9 流程逻辑控制
Control of Process Logic

流程控制方式的选择也能导致不同形式的松耦合。用一个中央组件控制整个流程逻辑会造成瓶颈，因为所有关联系统都必须和它连接。这样的中央组件的故障将中断整个流程。

反之，如果你有分散化或分布式控制（每个组件做自己的工作，知道接下来由哪个组件接手），就避免了瓶颈，如果某个系统出故障，其他系统照样可以工作。细节见 7.6 节。

4.2.10 部署方式
Deployment

你是要求系统更新同时部署，还是不同时部署，该问题和耦合相关。当然，如果要求系统更新同步进行，系统之间绑得就更紧密些。更松耦合的方法是在不同时间更新，然而，该方法会导致一个非常严重的缺陷：需要进行系统迁移，而这就导致版本划分问题（见第 12 章）。

4.2.11 版本划分
Versioning

你的版本划分政策也与耦合的松紧有关。如果一个系统提供了一定的数据类型，消费者已经使用了这些数据类型，那么，当供应者为数据类型添加新的属性时，你就会碰到麻烦。如果供应者引入了新的类型版本，消费者将不得不显式升级到这个新类型；如若不然，供应者就必须同时支持两个类型。另一方面，如果供应者仅仅向现有类型中加入属性，这就可能造成二进制兼容性问题，消费者需要重编译代码，或者使用其他程序库。

数据类型版本划分更符合松耦合形式时，只要修改是向下兼容的，消费者就不必做任何事情。

然而，正如 12.3 节讨论的，在此问题上达成松耦合有可能非常复杂。再次，你需要讨论比较优缺点，自己作出政策抉择。

4.3 处理松耦合
Dealing with Loose Coupling

本章讨论的松耦合形式只是一些（或多或少是典型的）例子。注意，没有什么硬性规定和快速指南：你必须根据特定的上下文环境和架构，作出适当的松耦合决策。

针对不同耦合类型，我见过非常不同的决策。正如我之前提到的，我的一个客户的政策是，只要可能，就避免异步通信，这基于他们自己的经验：异步通信在运行时导致竞争条件，而同样的竞争条件很难，甚至不可能在开发环境中重现，因此基本无法修复。同一领域的另一个客户有一条政策是，只允许对读取服务使用同步调用，因为对写入服务调用来说，同步调用的性能太差。

注意，你也可能必须就松耦合各种形式的组合进行决策。例如，你必须作出的一个重要决策是，ESB 与后端系统的分离应该通过协议还是 API 进行（见 5.3.3 节）。通过 API 进行分离通常意味着 ESB 要为每个后端系统或后端适配器提供要使用的程序库。于是，部署和绑定就是个问题。另一方面，使用公共的 API，可以把同步或异步通信的某些方面隐藏在 ESB 中。

你可能会问，哪种松耦合形式是典型的。据我所知，该问题没有答案。我所能说的就是系统越大，系统就越应该更松地耦合。

4.4 总结

Summary

- 松耦合是 SOA（及一般意义下的大型分布式系统）的一个基础概念，其目标是减少不同系统间的依赖。
- 松耦合有各种形式，你必须找到适合自己特定上下文环境和项目的耦合松紧程度。
- 任何形式的松耦合都有缺点，因此，松耦合本身永远都不应该是目的。
- 需要映射数据通常是大系统的“好”特征。

第 5 章

企业服务总线

The Enterprise Service Bus

SOA 的一部分是基础设施，它使你能在生产状态系统景观中使用服务。这通常称为“**企业服务总线**（ESB）”（注 1）。正如你将在本章看到的，关于 ESB 的确切角色和职责有着不同的观点。之所以对 ESB 有不同理解，部分原因在于实现 ESB 的技术手段非常不同。本章也将探讨某些技术手段产生的后果。

5.1 ESB 职责

ESB Responsibilities

在实践中，要使 SOA 得以运转，你必须有调用服务的方法。这个基础设施是 SOA 景观的技术支柱（有时也称为“技术底板”）。

ESB 的职责是，使消费者能够调用供应者提供的服务。这听起来似乎很容易，但实际情

注 1：和通常情况一样，该缩写词有很多意义。我想，ESB 也代表“帝国大厦”，或者《星球大战第五集：帝国反击战》只是种巧合。

况并非如此。依赖于实现 ESB 所采取的技术和组织方法，该职责可能包括（但并非仅限于）如下任务：

- 提供可连接性
- 数据格式转换
- （智能）路由
- 处理安全
- 处理可靠性
- 服务管理
- 监测和日志

这些任务可能需要针对不同的硬件和软件平台，甚至针对不同的中间件和协议执行。

ESB 的主要角色是提供互操作性。因为 ESB 集成了不同的平台和编程语言，所以，该角色的基础职能是数据格式转换。正如[Chappel04]所描述的：

> 在 ESB 部署中，数据格式转换是总线内生的一部分。某个单独应用所需的特殊格式转换服务，既可以插入到总线的任何地方，也可以从总线的任何地方访问到。因为数据格式转换是 ESB 本身如此密不可分的一部分，所以 ESB 可以被想象为解决应用间“**阻抗匹配错误**”的东西。

数据格式转换这件事很棘手，尤其是把性能问题结合起来考虑的时候（见第 13 章）。通常的做法是，引入一个特定的格式，所有单个平台和 API 都映射到该格式上。对 Web Services 来说，这种格式通常是 SOAP（见 16.2.3 节）。

ESB 的另一项基本任务是路由。一定要有某种办法从消费者向供应者发起服务调用，然后从供应者向消费者把答复传递回来（存在其他消息交换模式，在第 10 章会讨论它们，但是，这儿说的是技术上必需的东西）。依据所使用技术的不同，以及提供的智能的级别，这项任务可能很简单，也可能需要非常复杂的处理（细节见 5.4.2 节）。

ESB 的其他职责（我将在 5.4 节讨论）通常都是对“提供互操作性”这个核心任务的扩展。处理这些任务的机制是 ESB 密不可分或甚至是内生的一部分，还是仅仅是些附加软件，这是个值得讨论的话题。重要的是，你要知道自己系统中的哪一部分负责怎样的任务，以及哪个团队负责提供和维护它们。

例如，某人可能争论说，提供处理业务流程的技术能力是 ESB 的任务，如此一来，诸如 BPEL（见 7.4.2 节）这样的业务流程引擎协议就必须是 ESB 的一部分。另一方面，某人可能争论说，BPEL 只是个帮助提供组合服务的特定平台，因此不应该是 ESB 的一

部分。毫无疑问，一致化组合服务的方法，让服务能运行在流程引擎中是个好主意。但是，你是把这种一致化当作 ESB 设计的一部分，还是把它当作使用 ESB 的一般架构，这首先是组织结构方面的问题。（不是说它不是个重要的问题——我看到过很多由于职责不清导致的 SOA 问题。）

5.2 异质的 ESB

Heterogeneous ESBs

要知道并没有什么需求说 ESB 得是异质的。虽然在运行服务时只用一种技术可能更好些，然而，情况通常并非如此。由于其自身的性质，SOA 接受异质性，包括在中间件和协议上的异质性。即使是像 Web Services 这样的标准，其多个实现之间也会有差异。迟早有一天，会出现将事情做得更好和更简单的新标准或标准的新版本。一旦你开始使用新标准（同时也使用旧的技术），你的 ESB 就变得异质了。

从我的经验来看，实践中具备一定规模的 ESB 都是异质的。举个例子，看看图 5-1，它显示了我的一个客户在 2006 年的情况。

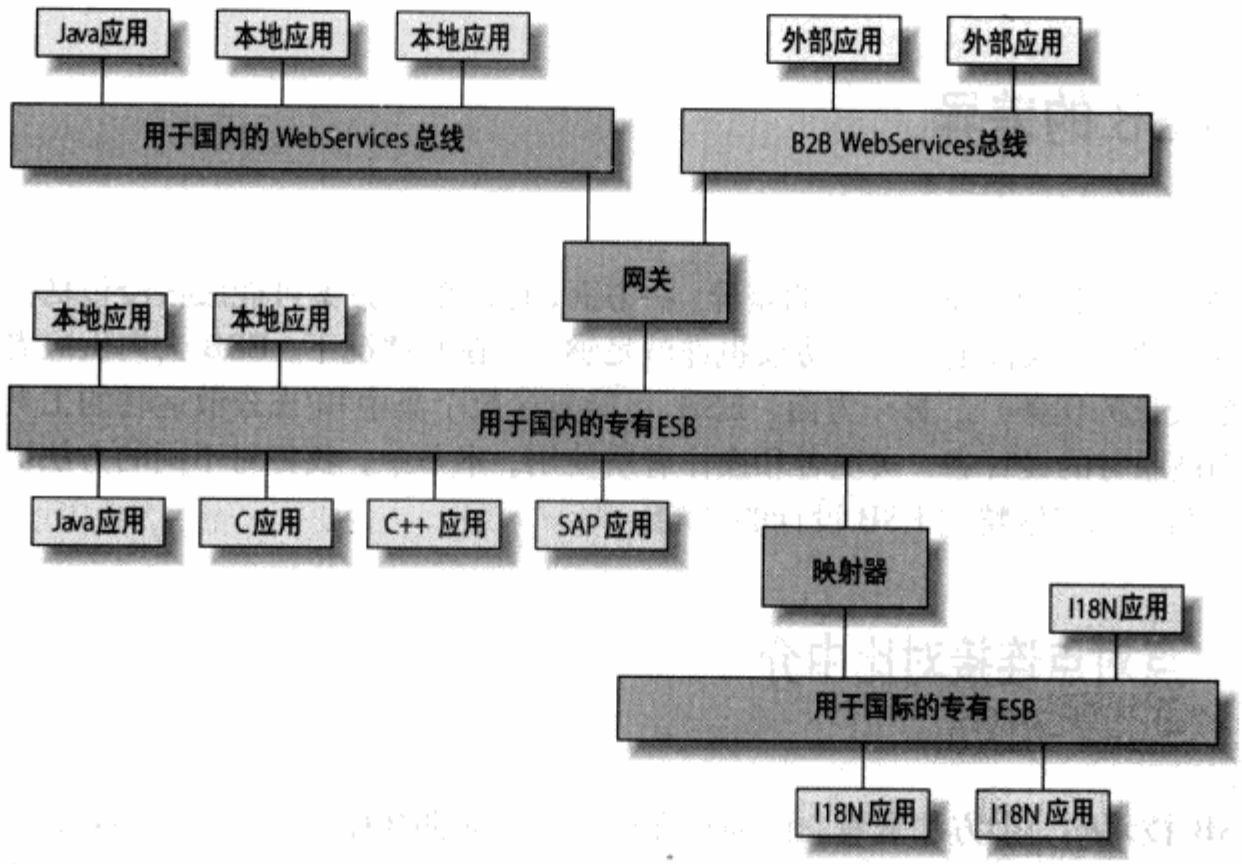


图 5-1：异质 ESB 的例子

他们的第一个 ESB 用了一家 EAI 供应商的专有技术（ESB 最早被认为是 EAI 总线）。然而，当他们与另一个国外公司并购后，大家发现使用以前的专有中间件技术不能保证互操作性。他们必须在两种专有 ESB 之间做一个映射器。SOA 逐步成长，Web Services 越来越成为一个事实上的标准，支持这种协议也变得有必要了，他们于是实现了一个网关，在技术上把以前的专有总线映射为 Web Services 总线。（事实上，因为安全方面的

原因，网关映射到两个不同的总线上，一个供内部使用，一个供 B2B 通信。）注意，随着时间过去，Web Services 网关映射到 Web Services 不同版本上。当前支持三个不同版本（使用不同的 SOAP 协议）。然而，向外部服务供应者的映射总是需要编些定制的代码，处理那些必须映射为内部专有格式，与外部情况不尽相同的 Web Services 功能。

有趣的是，公司同时也在建立公司范围的、完全基于新一代 Web Services 技术的第二代 ESB。这就意味着在实践中，ESB 景观甚至更异质化了，因为同时在使用所有这些 ESB 方法。

注意，从概念上来看，图 5-1 描述的 ESB 是“一个”ESB。使用这个 ESB，整个系统景观中的任何一个系统都可以和任何其他系统连接起来。某些部分使用不同协议仅仅是个技术细节。

理想情况是，改变 ESB 的实现技术不应对供应者和消费者产生影响。它们应该能够使用相同的 API，只有映射应该变化。这就是说，从供应者和消费者的角度来看，服务 API 应该是透明的。然而，要做到这样，通常要求 ESB 将针对任意平台提供服务 API 作为自己的职责之一。如果 ESB 要求使用一种指定的协议，则消费者和供应者就必须处理协议的变化。下一节要讨论这个问题。

5.3 ESB 的差异
ESB Differences

ESB 在技术上和概念上都有巨大的差异。一方面，你的解决方案可能压根没包括任何特定的工具或软件。仅仅定义一个协议也许就足够了（在此情况下，ESB 会向供应者和消费者分配大量的任务）。另一方面，ESB 可能包含数个集中和/或分散运转的工具和程序，它们被服务的设计者、实现者和操作者所使用。本节中，我将对不同的方法提供一个概览，帮助你理解特定 ESB 设计的影响。

5.3.1 点对点连接对比中介
Point-to-Point Connections Versus Mediation

区分 ESB 技术的一个方法是看它们为物理连接提供的耦合程度。消费者必须知道服务提供者确切的地址（“端点”）吗？或者，ESB 提供了某种机制，使消费者的请求可以送达正确的供应者吗？

如果消费者必须知道端点，则它是将每个请求发送给特定的接受者。这样的连接称为“**点对点**”连接（见图 5-2）。这类连接的问题是，如果物理接受者不可用，调用就会失败。

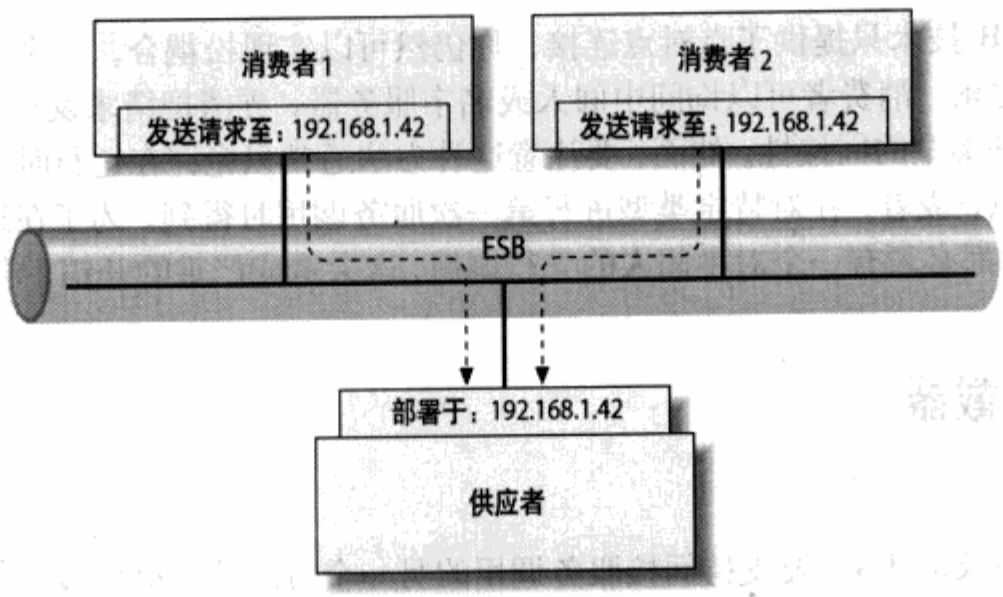


图 5-2：提供点对点连接的 ESB

如果消费者不必知道供应者的端点，反之，消费者可通过标签或符号来标识要访问的服务，而 ESB 解释标签或符号，找到合适的供应者（见图 5-3）。标签通常包含服务名字，此外，还可能包含一些影响路由的附加属性。例如，ESB 可能会处理优先级，或者对不同消费者有不同政策。这种情况下，ESB 扮演了中介者或中间人的角色，带来了松耦合的基础设施（见 4.2.3 节）。

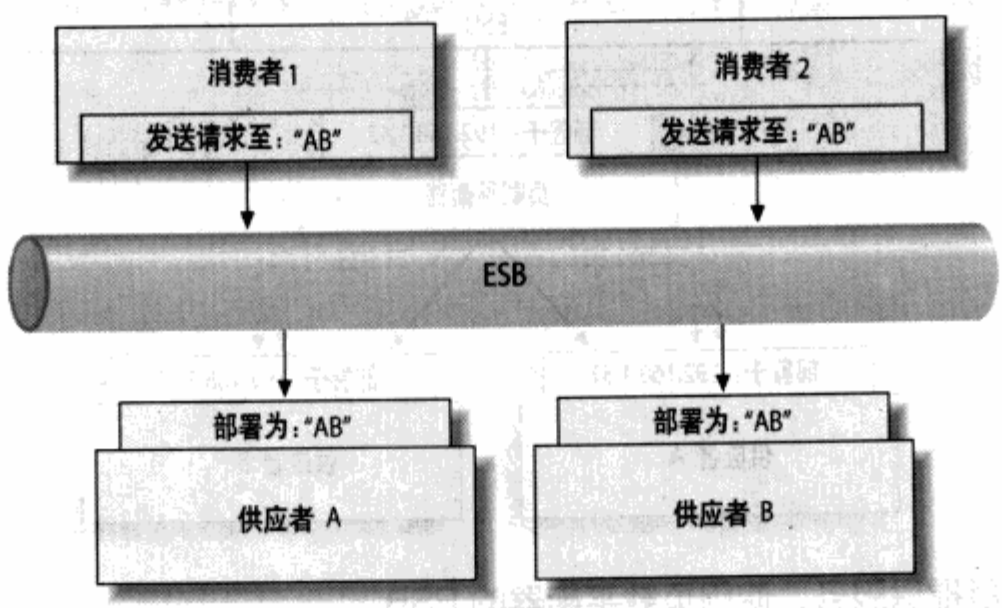


图 5-3：对连接起中介作用的 ESB

间接方式的优点是，ESB 能够处理 SOA 景观内的动态变化。某个服务能由多个系统提供，这就可以做负载平衡和失效备援（如果一个系统出故障，请求可以发送给剩下的供应者系统）。

为配置间接方式，ESB 可以在启动时（使用配置文件）或运行时（当每个供应者可用时，让他们注册自己）处理部署信息。

如果一个 ESB 技术只提供了点对点连接，你仍然可以实现松耦合。一个选择是，在实际发送请求之前，消费者可以询问中间人或名字服务器，弄清把请求发送到哪儿，这样也实现了某种类型的间接性。然而，要注意这种方法通常只允许在运行时得到请求所应该送达的端点，或者，在对特定类型进行第一次服务调用时得到。为了在运行时处理端点，每个调用都必须有一个对中间人的先行调用，这会造成严重的开销，导致性能瓶颈。

5.3.2 拦截器

Interceptors

基于点对点协议的 ESB 要支持间接服务调用的另一个方法是，提供被称为“拦截器”或“代理”的机制。一个简单方法是把提供服务的物理端点换成一个起负载平衡器作用的硬件或软件。消费者仍然使用一个正式的端点，该端点再委派真正的任务：当消息到达时，负载平衡器把消息分发到自己所知的不同的物理服务供应者上（见图 5-4）。

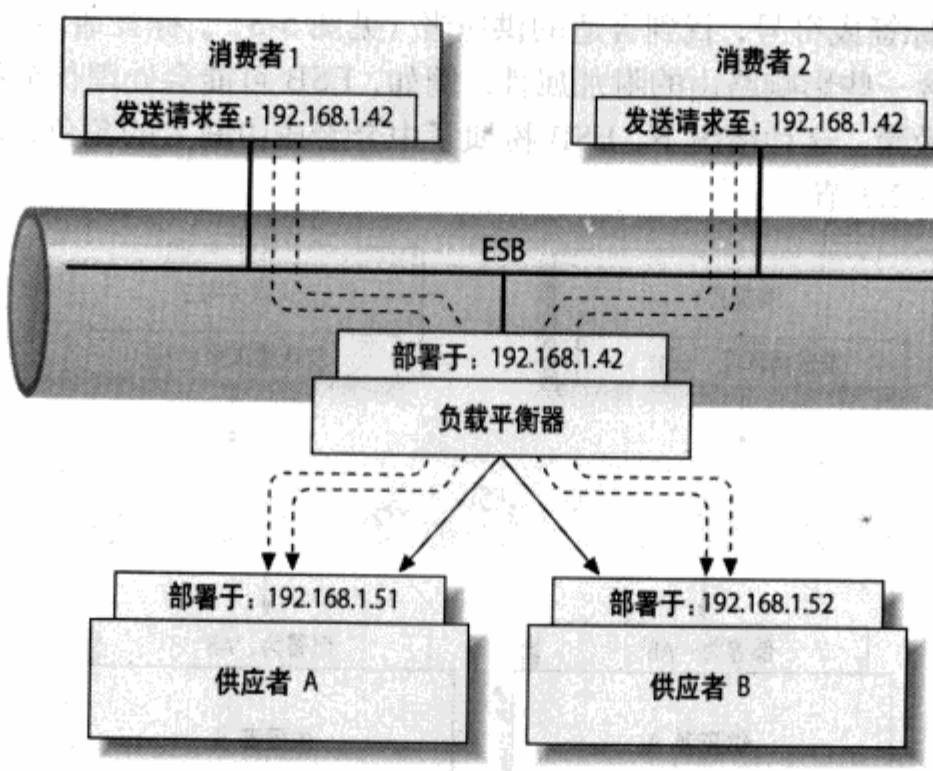


图 5-4：针对所提供服务，使用负载平衡器的 ESB

更复杂一些的 ESB 方法为每个供应者和消费者都提供一个拦截器或代理。此时，消费者将只与针对自己的特定拦截器进行“点对点”方式的通信。拦截器将把每个调用路由到合适的供应者，供应者也使用针对自己的特定拦截器（见图 5-5）。

在这种场景下，智能路由和其他 ESB 服务的魔法被拦截器完完全全地对外界封装了。事实上，在拦截器内部可能使用完全不同的协议（例子见第 16 章图 16-4）。

注意，Web Services 协议内生地就是一种点对点协议。只在两个系统间用服务进行交流

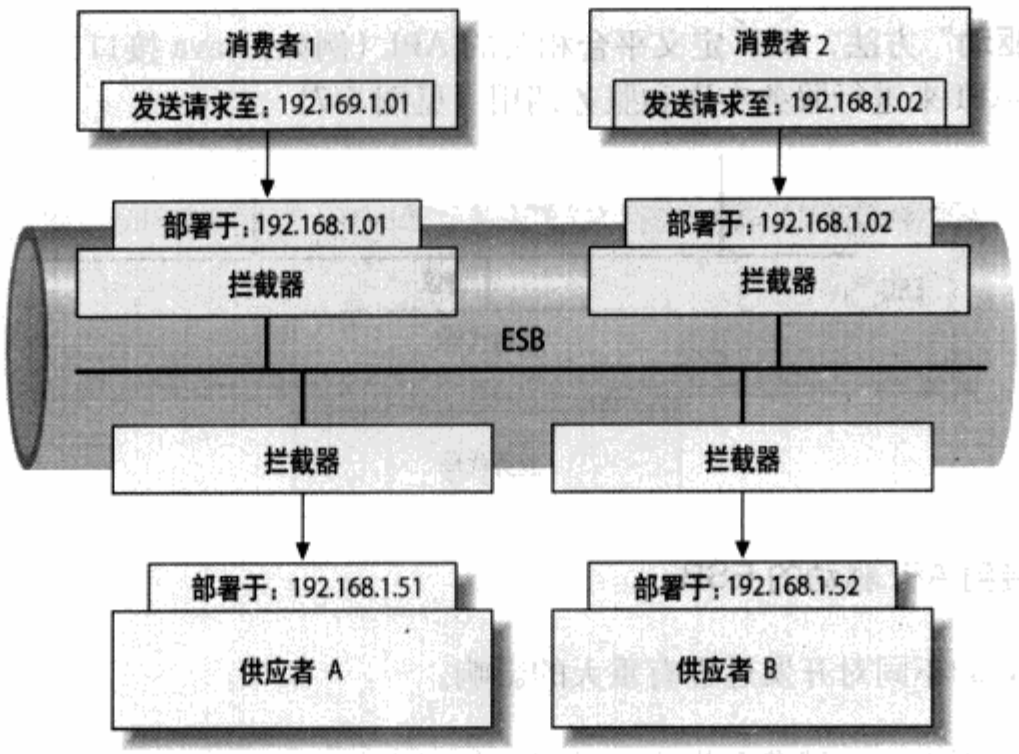


图 5-5：使用拦截器的 ESB

不需要工具和其他程序的支持，但是，你必须知道，并且在消费过程中指明服务调用的端点。然而，正如[PulierTaylor06]指出的：

> Web Services 标准没有包含对负载平衡和失效备援的规定。如果单用 Web Services，一个发生故障或过载的 Web services 无法将 SOAP 请求路由到备援网站。

负载平衡和失效备援仍然是大型分布式系统的核心需求。因此，每个基于 Web Services 的 ESB 迟早都要把拦截器结合进去（例子细节见 16.3.2 节）。这些拦截器还能帮助处理 ESB 的其他方面，如安全和监测（5.4 节讨论）。

5.3.3 协议驱动对比 API 驱动的 ESB

Protocol-Driven Versus API-Driven ESB

从供应者和消费者的观点来看，关于 ESB 的职责从何处开始这个问题，在概念上有两种可能的方法。使用“协议驱动”方法，ESB 定义一个协议，供应者和消费者都根据这个协议发送和接收消息（见图 5-6）。要求使用 SOAP 协议的 Web Services 就是该方法的一个例子。

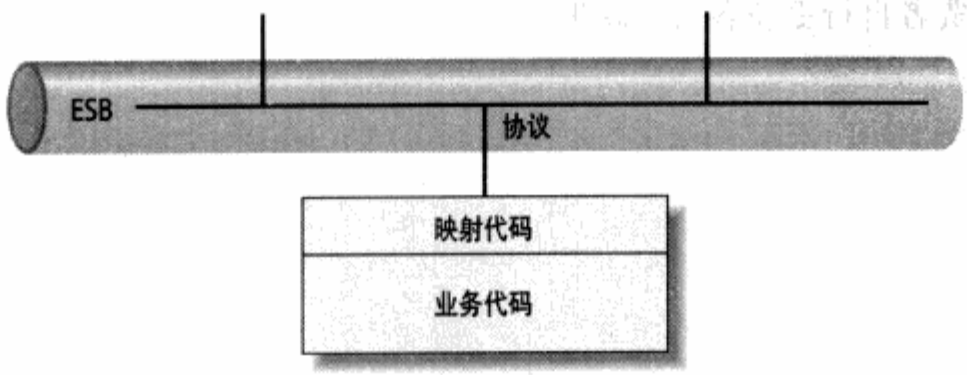


图 5-6：连接到协议驱动的 ESB

使用“API 驱动”方法，ESB 定义平台相关的 API（例如，Java 接口），供应者和消费者使用这些 API 来进行服务实现和服务调用（见图 5-7）。

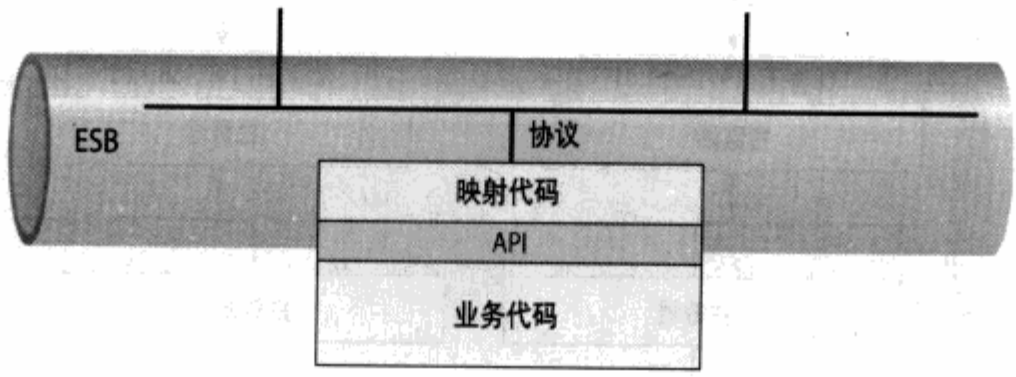


图 5-7：连接到 API 驱动的 ESB

这些方法之间的不同对开发过程有重大的影响。

当 ESB 只定义协议时，消费和供应的过程如何开发是细节问题，该问题完全在 ESB（及其相关团队）负责范围之外。基础设施团队不必提供工具和/或程序库来映射 API 到服务调用，以及映射服务实现到所需的协议上。相反，每个参与者（供应者和消费者）可以选择工具来使自己能够发出服务调用，以及提供满足 ESB 协议要求的服务实现。如果出现了一个新平台，提供针对 ESB 协议的适配器是使用这个平台的供应者和消费者的任务，负责 ESB 的团队与此无关（帮忙找出新适配器造成的问题除外）。

当 ESB 负责 API 时，将服务调用和服务实现映射成格式良好的、参与者可以通过 ESB 发出的消息这事，就是提供 ESB 的团队的任务了。供应者和消费者能集中精力在业务功能上。如果新平台需要被集成进来，ESB 有责任为相应的 API 提供解决方案（也就是说，提供代码生成器和程序库）。然而，由中央基础设施团队提供的程序库和代码生成器，会成为每个供应者和消费者开发和部署过程的重要组成部分。

此处的主要取舍是，针对基础设施团队和供应者、消费者的开发团队，决定它们各自的独立性究竟有多大？协议驱动方法给了供应者和消费者更多的责任。它们的整个开发过程在技术上相对基础设施独立，但又必须满足所有 ESB 协议的要求。这种方法解耦了基础设施和系统开发，但是带来了系统开发的风险，因为可能出现许许多多把内部 API 映射到协议的方法。原则上，每个开发团队必须解决同样的问题，如果协议变化了，需要供应者和消费者自行更新各自的映射。

通常，协议驱动方法带来分布式通信模型中的第三个层次（见图 5-8）。在模型的底层，有一个相对稳定的协议；在顶层，有一个调用和实现服务的 API；在中间，有一个层负责将 API 映射到协议。这个中间层可以由特定平台的团队、特定公司的团队，或者第三方供应商等提供。

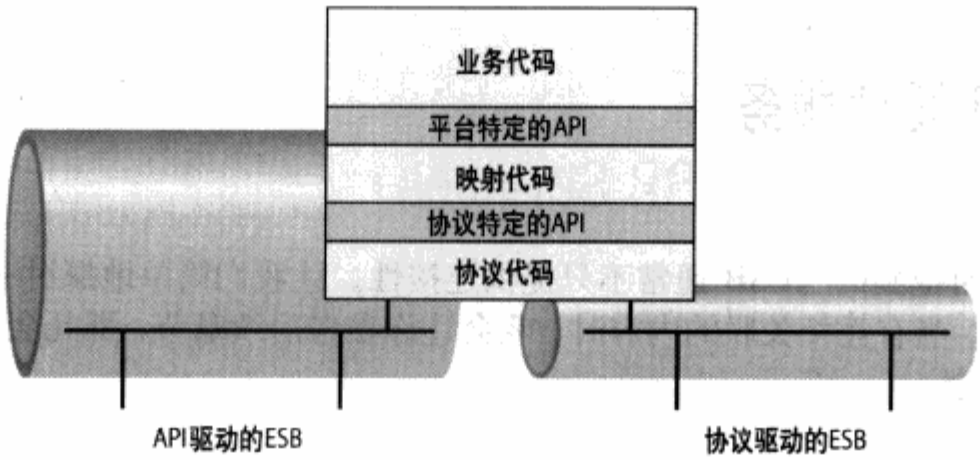


图 5-8：从业务代码到协议代码的各层

在互联网上可以找到相同的层级结构。传输层包含了稳定的协议 HTTP(S)。要通过这个协议发送数据，不同的供应商提供了消费客户端（浏览器）或实现工具（网站引擎或平台相关程序库），所有这些提供了针对诸如 Java、HTML 等平台相关接口的映射。

在 SOA 中，协议驱动的 ESB 导致这样的情形：每个消费者和供应者都将 ESB 规定的协议映射到平台特定的业务接口上。例如，需要一些代码把 Java 接口转换到 Web Services 协议。问题是：使用哪些工具来生成代码，以及谁对代码的生成和维护负责。通常，这是供应者和消费者的责任。但是，一个推荐的或经过认证的标准工具（由基础设施团队或第三方团队提供）也可能被用来生成这种代码。

作为对比，API 驱动的 ESB 为业务团队提供了 API，这使协议细节对供应者和消费者是透明的。结果就是供应者和消费者需要（生成的）程序库，还需要一个分发程序库的过程。

使用协议驱动方法时，当协议更改，或协议无法保证互操作性时，你都会遇到麻烦。如果协议更改，每个供应者和消费者都必须相应地修改各自的映射（生成器）。这可能变得代价高昂。并且，如果协议不够好，不能保证互操作性的话，你可能最终不得不维护和验证供应者和消费者之间的每个单独的连接（带回了 ESB 本来应该解决的问题）。注意，Web Services 内生是协议驱动的，目前有缺乏互操作性的问题（细节见 16.3.1 节）。

毫无疑问，协议会随着时间变化。因此，所有使用 Web Services 的公司迟早都要考虑采用作为 ESB 一部分的映射层（但是，我也见过由不同的团队来负责 ESB 和公共映射层的）。让一个中央团队提供在业务 API 和协议 API 之间进行映射的解决方案，顺带还解决了这样的问题：负责映射到协议的代码生成器通常导致非常不同的 API，而这就意味着你不能轻松地切换到不同的服务代码生成器上（见第 18 章）。

5.4 ESB 增值服务

Value-Added ESB Services

正如本章开始时提到的，ESB 通常不只提供连接性。让我们简单地探讨一下 ESB 能提供的其他服务。当有互相关联的内容时，我会让你参考后续章节，那儿会对这些话题进行更详细的讨论。

5.4.1 数据映射

Data Mapping

如 4.2.2 节所介绍的，松耦合通常导致这样的情形：只有少数基础的、稳定的数据类型被定义为公共数据模型。对所有其他信息类型来说，供应者都要定义自己的数据类型。另外，12.3.1 节将展示出，数据类型的版本划分甚至会导致如下情形：在同样的供应者提供的服务中，对同样的数据使用不同数据类型。其结果是，消费者必须有一个薄的映射层，将供应者数据类型映射为自己的数据类型。

当然，ESB 可以处理数据映射和数据格式转换的需求。如前面所讨论的，这依赖于你如何定义 ESB 的职责。在协议特定的 API 和平台特定的 API 间进行映射的层，也应该能够映射数据类型。作为选择，ESB 也可以定义出在协议内部进行数据类型映射的代理。当然，两种方法都需要一些措施来指明映射的要求。

当前，有的供应商试图通过在 ESB 内部引入公共对象模型来解决这个问题，这当然会抵消松耦合的效果。当某个供应商试图把“**主数据管理**”（MDM）作为其 SOA 的中心部分卖给你时，要格外小心了，这也许是合适的，但也可能和你需要的东西刚好相反。

5.4.2 智能路由

Intelligent Routing

本章中，我们已经讨论了路由信息的某些方面，尤其是技术处理方面，如负载平衡和失效备援。然而，从业务的观点来看，你可能还需要路由信息的不同方法。例如，消息可能被赋予不同的优先级，或者，你甚至可能根据消息内容的不同给予不同处理。后者被称为“**基于内容的路由**”（CBR）。

例如，在一个消息内部，你可能看到一个邮政地址使用了老格式的邮政编码（比如，如果是美国地址，用了 5 位而不是 9 位数字；或者，如果是德国地址，用了 4 位而不是 5 位数字）。包含老邮政编码格式的消息可能被路由到一个特殊的供应者，在那里它被转换成新的邮政编码格式。

当然，处理消息内容的这种办法要求 ESB 至少对部分服务有语义方面的理解。换句话说，不光供应者和消费者在依赖某个服务的确切接口。对这种方法要非常小心。原则上，你可以加上些 ESB 能处理的公共属性，但为了能够伸缩，这些属性必须是消息头的一部分（对此更多讨论见 15.4 节）。

5.4.3 处理安全
Dealing with Security

在绝大多数 SOA 景观中，由于会对消费者调用服务和获得结果的能力应用一些限制措施，所以安全迟早会成为一个问题。因此，ESB 要提供一些方法来处理有关安全的各种考量。第 14 章会讨论这个话题。

5.4.4 处理可靠性
Dealing with Reliability

不同协议提供了不同形式的可靠性。协议并不一定需要保证消息的传递“有一次且只有一次”。实际上，Web Services 的 HTTP 协议内生地就无法保证消息传递。其他一些协议有可能作出某种保证，比如，保证消息会传递“至少一次”。

因此，ESB 应该定义自己如何处理可靠性问题。如何处理故障和技术错误可能会受 ESB 所用协议的影响，但是，ESB 本身也能添加一些机制来改变协议的行为。第 10 章会详细讨论该问题。

5.4.5 服务管理
Service Management

当 SOA 范围扩展时，你迟早会碰到管理所有服务的问题。如果分析师和设计师在寻求找到既存服务并且在新的业务流程中重用它们的办法，该问题由业务驱动；或者，该问题由技术驱动，即，可能出现一个技术需求，要求能够部署和运行一个服务基础设施。第 17 章会讨论这个问题。

5.4.6 监测和日志
Monitoring and Logging

为开发 ESB 支持日志和监测的能力时，无论投入多少工作量都不过分。记住，ESB 已经成为运转中业务流程的技术心脏，你实施、调试和维护本地过程的方法，现在都分布到了许多系统上。这就意味着，ESB 实际上成了分布式过程上的调试器。因此，你应该建立诸如关联 ID 之类的概念，以及在分布式系统上支持监测和调试的工具。关于这个话题的更多细节请见 15.3 节。

注意，监测和日志应该包括哪个消费者调用了哪个服务，以及一个具体的服务调用进行了多长时间。这些对于撤销服务（见 11.2.2 节）和监测服务等级协议非常重要。

5.4.7 业务活动监测（BAM）
Business Activity Monitoring (BAM)

有一个论点不但对 ESB 有用，甚至还经常用在 SOA 可能的业务案例上，那就是 ESB 将使“**业务活动监测**”（BAM）成为可能。这个观点是，ESB 使你可以监测消息，即时了解业务的状态，从而可以非常迅速地作出反应。

例如，你可以监测特定服务被调用的时间，这可以帮助你优化市场营销部门投放广告的方式。另一个例子是，有了 BAM，如果有一些不同寻常的事情发生（例如，某些服务比平时调用得多或少了），你可以立即发现某些商业机会，也能抢在你的竞争对手之前采取反应措施。这有点像数据仓库处理，不同的是你监测和处理的是行为，而非行为产生的数据。

当然，我说不好这个概念对你是否有用，或者是否适合你的环境。在技术上，通过监测服务调用发现“异常”行为应该行得通。问题（技术上和组织上）在于，如何把这些信息映射成有用的业务输入。建立一个流程，当异常的服务调用行为发生时，让相应业务部门能够被提醒，也许这就足够了。

在任何情况下，BAM 都要求 ESB 能够在适当的抽象层次上监测服务调用。这就是说，你需要能在消息层上监测，然后集中处理收集到的数据。正如刚刚描述的，为了日志和调试，你也需要一些方法来处理和关联所有的消息。因此，通常说来，这不是什么真正的新需求。

5.4.8 服务实现支持
Service Implementation Support

有时，ESB 如何支持服务实现这个问题也会被提出来。关于组合（配制）服务，对 Web Services 来说，人们通常建议包含进一个如 BPEL 这样的流程或工作流引擎，使其作为 ESB 的一部分。同时提供这两者确实产生了某种协同效应。例如，你的 ESB 监测可以包括一个运行中流程的图形表示。或者，如[Chappel04]所说：

> ESB 的业务流程能力，使得定义既对单个部门或业务单位来说是本地的，又能使在一个更大的集成网络中共存的业务流程成为可能。这是单靠星型集成中间人或 BPM 工具无法做好的事情。

然而，在此要小心。从 SOA 出发，指定使用一个公共的流程引擎可能很容易。但是，正如为数据库或其他系统提供一个公共实现平台非常困难一样，要保持（流程）服务实现平台的同质性并不容易。关于该问题更多讨论见 7.4.4 和 9.1 节。

5.5 总结
Summary

- 企业服务总线（ESB）是 SOA 的基础设施。
- ESB 的目的是提供互操作性（连接性、数据映射和路由），以及其他一些附加服务，如安全、监测等等。
- ESB 可能是异质的（而且通常如此）。
- ESB 是协议驱动还是 API 驱动，这是个基本决策。
- 协议驱动的 ESB 定义供应者和消费者必须遵守的协议，但是，如何遵守协议由供应者和消费者决定。ESB 和被连接系统之间的解耦方式使得它们不共享任何代码，所以 ESB 不必为各系统部署程序库。该方法的缺点是，协议的任何变化都迫使供应者和消费者作出相应更新。
- API 驱动的 ESB 为供应者和消费者提供 API，它被用来实现和/或调用服务。这使得协议的细节透明，但要求 ESB 按某种方式向供应者和消费者部署生成的代码和/或程序库。
- ESB 可以提供各种增值服务。其中最重要的是分布式调试能力。
- 业务活动监测（BAM）使你能监测 ESB，这样你就能实时了解自己的业务，相应作出反应。这可以创造一些商业机会和市场优势。

5.5 总结

- 企业服务总线（ESB）是 SOA 的基础设施。
- ESB 的目的是提供互操作性（互连性、数据映射和路由），以及其他一些附加服务，如安全、监测等等。
- ESB 可能是异质的（而且通常如此）。
- ESB 是协议驱动还是 API 驱动，这是个基本抉择。
- 协议驱动的 ESB 定义供应者和消费者必须遵守的协议，但是，如何遵守协议由供应者和消费者决定，ESB 和连接系统之间的耦合方式使得它们不共享时间位置，所以 ESB 不会为各系统施加延迟率。这方法的缺点是，协议的任何变化都迫使供应者和消费者作出相应更新。
- API 驱动的 ESB 为供应者和消费者提供 API，它能用来实现和以太网相服务。这使得协议的细节透明，但要求 ESB 按某种方式向供应者和消费者部署生成的代码和/或程序库。
- ESB 可以提供各种增值服务，其中最重要的是分布式测试能力。
- 业务活动监测（BAM）使能能监测 ESB，这种能就能实时了解自己的业务，相应作出反应，这可以创造一些商业机会和市场优势。

第 6 章

服务分类

Service Classification

正如第 3 章所述，定义服务有不同的角度，服务也有不同的属性。即使在同样的上下文环境和景观中讨论，服务也可能有差异。因此，对这个术语不可能用一个通用、具体的定义来进行简单的描述。业务需求种类繁多，而且，服务被用于非常不同的目的，扮演的角色也差异巨大。有的服务读取数据；有的服务写入数据。有的服务简单、快速地起作用；有的服务执行复杂的工作流。服务可能不访问后端，也可能访问一个或多个后端。正如面向对象编程中的对象一样，服务的种类繁多，形式各异。所以，试图用同样的方式处理所有的服务没有意义。相应地，在服务各特性之间的常见差异变得清晰后，对服务进行分类是有意义的。

在本章中，将讨论一些不同的分类方法。和通常情况一样，不会搞出一个一般规则，而是试图提供一个可以为具体的 SOA 进行决策的好的出发点。

当然，分类服务有许多方法。我将从在社区中相当常见的技术驱动法开始（虽然有时用不同的术语）。之所以从这个方法开始，是因为从我的观点来看，在把服务分类得能够

简单地引入 SOA 的各个层次和扩展阶段上，这个方法是做得最好的。然后我会讨论我在书中和实际项目中碰到的其他服务分类法。

6.1 基础服务分类
A Fundamental Service Classification

就服务分类而言，我喜欢一开始引入如下 3 个类别：

- 基本服务
- 组合服务
- 流程服务

这是你在许多出版物中都能找到的分类，虽然确切的术语用词有细微的差异。例如，[KrafzigBankeSlama04]叫“基本”服务、“中级”（组合）服务和“流程为中心的”（流程）服务。（另外，他们还引入了第 4 种称为“公众－企业”服务的基础分类；见 6.5.1 节）在[Erl05]的分类中，“业务服务”和我称为的“基本服务”最为接近，术语“流程服务”则覆盖了我称为的组合和流程服务（见后续“Erl 的分类”）。[BloombergSchmelzer06]使用术语“原子服务”而非基本服务。

基于这 3 个服务类别，可以定义 3 个服务层和扩展阶段（见图 6-1）：

- 第 1 个扩展阶段被称为“**基础的**”*SOA*，它只有一个基本服务层。
- 第 2 个扩展阶段被称为“**联合的**”*SOA*，除了基本服务外，还有个组合服务层（这个层也可以叫做配制或组合层）。
- 第 3 个扩展阶段被称为“**流程使能的**”*SOA*，它有用于流程服务的第 3 层。

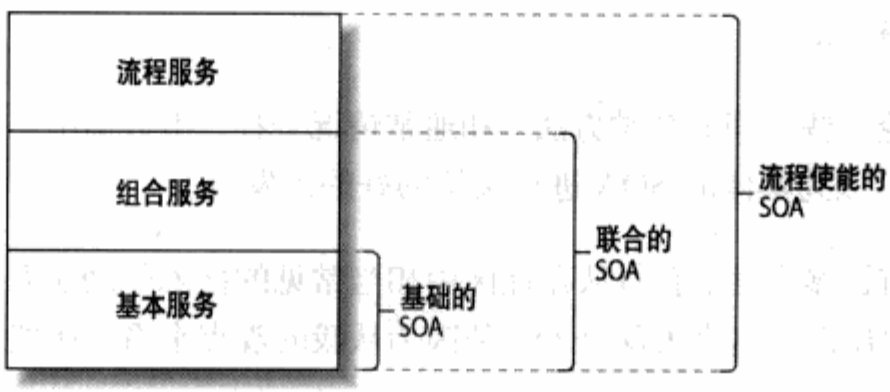


图 6-1：基础服务层次和 SOA 扩展阶段

让我们逐个讨论这些不同的类别、层次和扩展阶段。

6.2 基本服务
Basic Services

扩展的第一个阶段只有基本服务。每个基本服务提供一个基本的业务功能，并且，将该功能进一步拆成多个功能已经毫无意义。通常，对一个特定后端或问题领域而言，这些服务提供了第一个基础的业务层。典型地，这些服务都是短期运行，并且在概念上无状态（见 15.1.1 节）。故此，同步调用可能比较有效。

这些服务的角色是，对后端或问题领域进行封装，这样消费者（和高层服务）就能使用公共的 SOA 架构来访问后端。

有两类基本服务：

- 基本数据服务
- 基本逻辑服务

6.2.1 基本数据服务
Basic Data Services

基本数据服务从一个后端系统读取数据，或者向其写入数据。通常，每一个这类服务都代表了后端的一个基础业务操作。

基本服务对外部世界封装了平台相关部分及实现细节，这样消费者就能在不必知道服务如何实现的情况下请求服务。

这些服务应该提供一些最低限度的业务功能。这就是说，不应该有技术接口，比如，某个服务返回一个数据库表（见 3.1 节的选读栏“业务驱动对比技术驱动业务接口”）。相反，根据业务需求，基本数据服务典型地或者读取和返回，或者写入通常在一起使用的一大块数据。这也就是说，业务人员必须能理解服务在干什么。

[Allen06]定义了如下“最底层服务达到最优粒度水平的指导方针”：

- 应该能够从功能、信息、目标和原则这些方面来描述服务，但是，不能以别的服务构成的组的方式来描述服务。
- 服务的功能集应是提供业务能力的，类似于家庭的最小生产单位。
- 应由单一的角色来负责服务。
- 服务应该尽量自足，理想的服务应该是自治的。

典型的服务例子（这里讨论针对客户的服务）如下：

- 创建一个新客户。
- 改变客户的地址。
- 返回客户的地址。
- 创建一个新合同/投资组合/账户。

- 根据某些查询条件返回客户列表。
- 返回客户的账户余额。
- 向客户发送订购的东西。
- 返回客户的数量。
- 返回客户付款行为的细节。

这些服务被正确封装和抽象，从而解决掉一些问题是很重要的。在我们讨论的情况下，服务应该封装技术细节，减轻后端发生不一致的危险。因此，重要的目标是，这些服务不会在它们运作的后端上造成不一致。事实上，基本服务应该具备所谓的 ACID 特性，这意味着它们应该具备：

原子性

这是说对服务的调用要么成功，要么没有影响。保证这个特性是提供服务的后端的任务。

一致性

这是说在服务调用之后，后端达到了一个合法、一致的状态。如此一来，就没有服务调用能够把后端带进一个不一致的状态。

隔离性

这是说被后端处理的服务，不应该被同时、在同样的后端上运行的其他服务调用所影响。换句话说，读取服务应该返回自身一致的数据。

持久性

这是说当服务调用成功后，要保证影响是持续的。换句话说，没有什么系统故障可以意外地撤销服务调用的结果。

6.2.2 基本逻辑服务
Basic Logic Services

基本逻辑服务代表了基础业务规则。这些服务通常处理一些输入数据，然后返回相应结果。

与基本数据服务相比，基本逻辑服务是少数。典型的基本逻辑服务例子如下：

- 定义产品目录和价格列表。
- 定义改变客户合同的规则。
- 返回某年是否是闰年的答案。
- 定义允许的日期，这可能是对真实日期的限制或扩充（例如，存在诸如 6 月 31 号这样的银行日期，目的是简化一些处理）。

然而，通常说来，这些服务也是对提供这些功能的后端系统的包装（因为历史原因）。这样一来，基本逻辑服务和基本数据服务之间的界限，有时就不那么明确。这并不是个

问题，因为在 SOA 中，所有的基本服务，不管它是数据服务还是逻辑服务，通常都以相似的方式处理。如果说在基本服务之间有什么重大区别的话，这样的区别通常出现在读取和写入服务之间（见 6.5.2 节）。

6.2.3 基础的 SOA
Fundamental SOA

由于基本服务的引入，你得到了 SOA 扩展的第一阶段，它被称为“基础的 SOA”（这个术语取自[KrafzigBankeSlama04]）。

通过基本服务的引入，服务消费者可以使用 ESB 来处理由一个后端负责的业务功能（见图 6-2）。

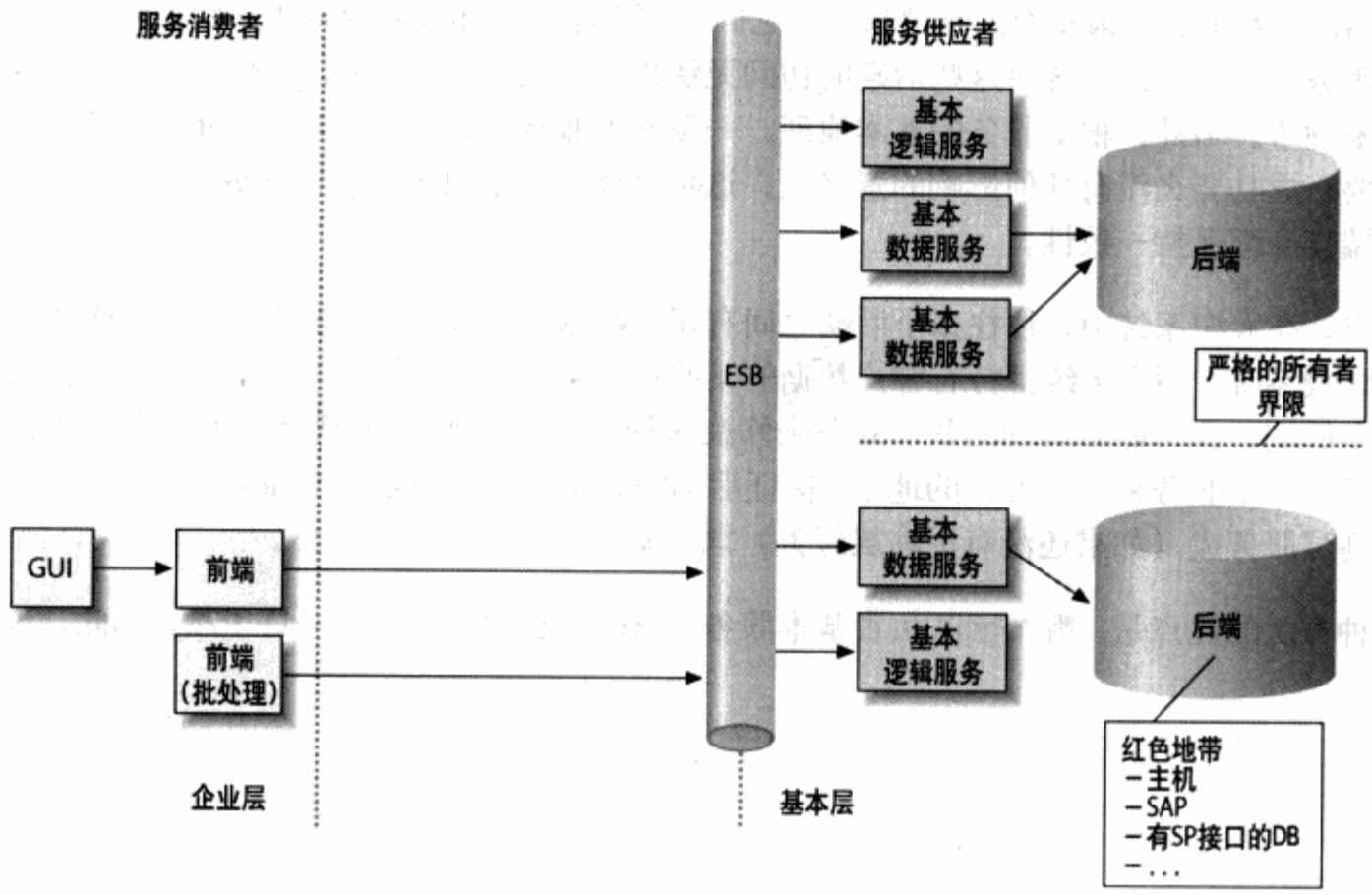


图 6-2：基础的 SOA

何谓“后端”并无一定之规。一个后端可以是任何负责特定数据群或功能的东西。这儿的重要点是：该后端扮演着清晰的，并且多多少少是独特的角色。

技术上，一个后端可以是一个数据库、主机、大型机、SAP 系统、规则引擎、J2EE 服务器群、对其他公司的远程连接等。它可能包含了一个或多个系统。基本服务适配器的任务就是隐藏技术细节和提供一个常用的服务接口，这样就可以通过 ESB 来访问后端。

从组织角度来看，一个后端通常是你能在组织结构中定位的东西。从业务角度来看，它是个扮演着特定角色，由特定一组人维护的系统（或一组系统）。它提供了某些一般性的功能，如记账、客户管理、订单管理，等等。

由于历史（和狂乱莫名）的原因，后端在实践中意味着什么可能有很大的差异。通常，系统和后端的边界问题反映了一个公司在整体业务领域模型上的一般性问题，而业务领域模型则由组织结构支撑。

在描绘一个大系统的总图时，我通常把这些后端描述为“红色地带”。这是个美式橄榄球的术语（注 1），指球场最后 20 码。当球到达这个区域时，情况就变得很紧急，因为角逐空间被进一步压缩了（只有 20 码空间可以腾挪）；同时情况也很危险（因为你逼近了输球赢球的关键点）。对于 IT 系统的后端而言，这两种条件都满足：它们以非常受压的形式包含、管理着核心业务数据和功能（没有诸如符合人体工学的用户界面这样的附加特性），并且，因为这些数据代表着公司的核心，错误和滥用可能造成严重的后果。在一个银行项目中，针对处理的数据，我听到过人们使用“法律数据”这样的术语，因为在后端中存储的是所有与公司可能在法律上有关系的数据。

对后端系统而言，最重要的事情就是，它们一定不能被损坏或者变得不一致。因此，基本服务要保证外部系统对这些后端的访问无法损坏它们。这就是说，基本服务应该隐藏技术细节。另外，根据服务的基本原理，一项基本服务应该体现一项完整的业务功能。在要么成功要么没有任何影响的意义上，这项功能应该是原子的。不管成功还是失败，后端自身都保持一致性。

注意，在大型系统中，往往多个后端之间有冗余数据。例如，你可能有一个 CRM 系统和计费系统，两个系统都持有客户数据的某些项目（ID、地址等）。在此情况下，针对每个后端，你可能有不同的基本服务来修改这种数据。比如，CRM 系统和计费系统都提供了基本服务来修改客户的地址。保证后端间数据一致性不是基本服务的任务。这是更高层服务或（如果还没有更高层服务）调用基本服务的前端的责任。

这种方法的优点是，哪个团队负责基本服务（该团队也负责后端）一目了然，如此，该

注 1：美国读者可能会疑惑我为什么在“足球”（football）前加上“美国”二字。这些读者可能会说这是一种同义重复，因为另一种足球被称为“soccer”。然而，在世界的其他地方，“football”这个名词通常就用来指“soccer”，所以，加词进行澄清有助于来自其他国家的人理解我在这里到底说的是哪种运动。

团队就可以集中精力于既避免数据出现被损坏的危险，又满足对后端提供外部访问的需求。

消费基本服务的系统可能非常不同。它们可能是简单的客户端，或是批处理程序，或是多层结构的客户端。它们调用的基本服务是存储和获取数据，或提供基本数据转换。这就是说，基本服务在它们的最底层。在此之上，可能有其他额外的层负责其他的业务处理和表示。这些层甚至可能在物理上是分离的，这有点像在门户软件中，服务器为互联网浏览器提供内容。

如果这些客户端被允许执行诸如修改客户地址这样的任务，那么，通过对不同的后端调用适当的基本服务，从而在后端间保证一致性，就是这些客户端的责任。

注意，SOA 不要求这些基本服务的接口被一致化。虽然这样做确实不错，然而，情况通常并非如此。（松耦合的这个方面在 4.2.2 节讨论过）由不同后端提供的相似基本服务，其名字和属性可能会有差别，处理这些差别是基本服务消费者的任务。

然而，仅仅因为许多后端处理了客户数据，就要求每一个前端不得不经常地为此劳心劳力，这样做也很不合理。很明显，如果前端能够针对所有后端作全局性修改，这将是很可取的。这就是组合服务之所以需要的原因。

然而要注意，在大系统中，把保持一致性的责任交给更高的层，要它们确保针对多个后端的多个调用是一致的，这通常也并非易事。为了使事情简单些，应该明确哪个系统为主，哪个系统是派生的，然后应用相应的同步措施。注意，这些同步通常在 SOA 范围之外。同步后端不是分布式业务流程的一部分。因此，你通常不会用服务来同步数据（虽然从技术上来说，你可以用服务基础设施来执行这项任务）。

6.3 组合服务
Composed Services

扩展的第二阶段加上了组合服务。这些服务代表了第一类由其他服务（基本和/或其他组合服务）组合而成的服务。

在 SOA 术语中，从现存的服务中组合出新的服务被称为“配制”。就像在管弦乐队中一样，你组合不同的“乐器”，这样就可以演奏出比单一乐器复杂得多的音乐。因此，这些服务也可以叫做“**配制的服务**”。

这些服务的运行层次比基本服务高，然而，它们仍然是短期执行的，并且从概念上来看是无状态的。用工作流的术语来说，一个组合的服务体现了一个“**微流**”，即在业务流程内部短期运转的活动流（在此情况下是服务）。

注意

在[KrafzigBankeSlama04]中，这类服务被称为“中间服务”，这个术语不太好，因为无法从字面直接理解其含义。你必须知道其他类别是什么才能理解“中间”是何意义。

6.3.1 用于多个后端的组合服务
Composed Services for Multiple Backends

组合服务是典型的访问多个后端的服务，因此，组合服务由许多基本服务组成。一个典型的例子是在多个后端上更新冗余数据的服务。例如，大型系统可能在多个后端上维护相同的客户数据（对一家公司的本地系统来说，在 CRM 和计费系统上可能找到冗余客户数据；对一个跨越了多家公司的系统来说，在每家公司的 CRM 系统上都可能找到相同的客户数据）。提供一个服务，在所有的后端上修改客户地址，你就能保证所有后端的一致性（这与基本服务不同，基本服务只负责一个后端的一致性）。

更复杂些的例子是一个变更电话合同的服务。与在不同后端上对地址进行简单更改不一样，在变更电话合同的情况下，不同后端可能需要进行不同的更新。针对客户数据，在计费系统中的更新将不同于在处理真实电话呼叫的核心网络中进行的更新。

组合服务的第三个例子是从一个后端向另一个后端转账的服务。在此情况下，组合服务会调用一个基本服务来从某个后端提取现金，调用另一个基本服务把现金存入别的后端。

后两个例子马上提出了事务安全的问题。如果第一个服务成功而第二个服务失败会怎样？另外，从消费者的角度出发，这些服务应该具备 ACID 特性（之前的 6.2.1 节讨论过）。如此一来，它们应该要么成功要么没有影响。这就是说，如果两个基本服务中只要有一个失败，你就必须回滚成功的服务调用，返回失败。

6.3.2 用于一个后端的组合服务
Composed Services for One Backend

组合服务也可能对现有服务在某些方面进行映射或改装。这样的服务也可以被称为“改装的服务”。

这些服务可以为某个服务提供不同的接口，比如，不同的名字，增加或减少一些属性，如此等等。典型的应用是提供向下兼容性，或者把服务映射到所要求的接口上。

6.3.3 联合的 SOA
Federated SOA

同时具备基本服务和组合服务就迈进到了我称之为“联合的 SOA”的扩展阶段（[KrafzigBankeSlama04]称之为“**网络化的 SOA**”）。本阶段为组合服务引入了一个额外的层次（见图 6-3）。

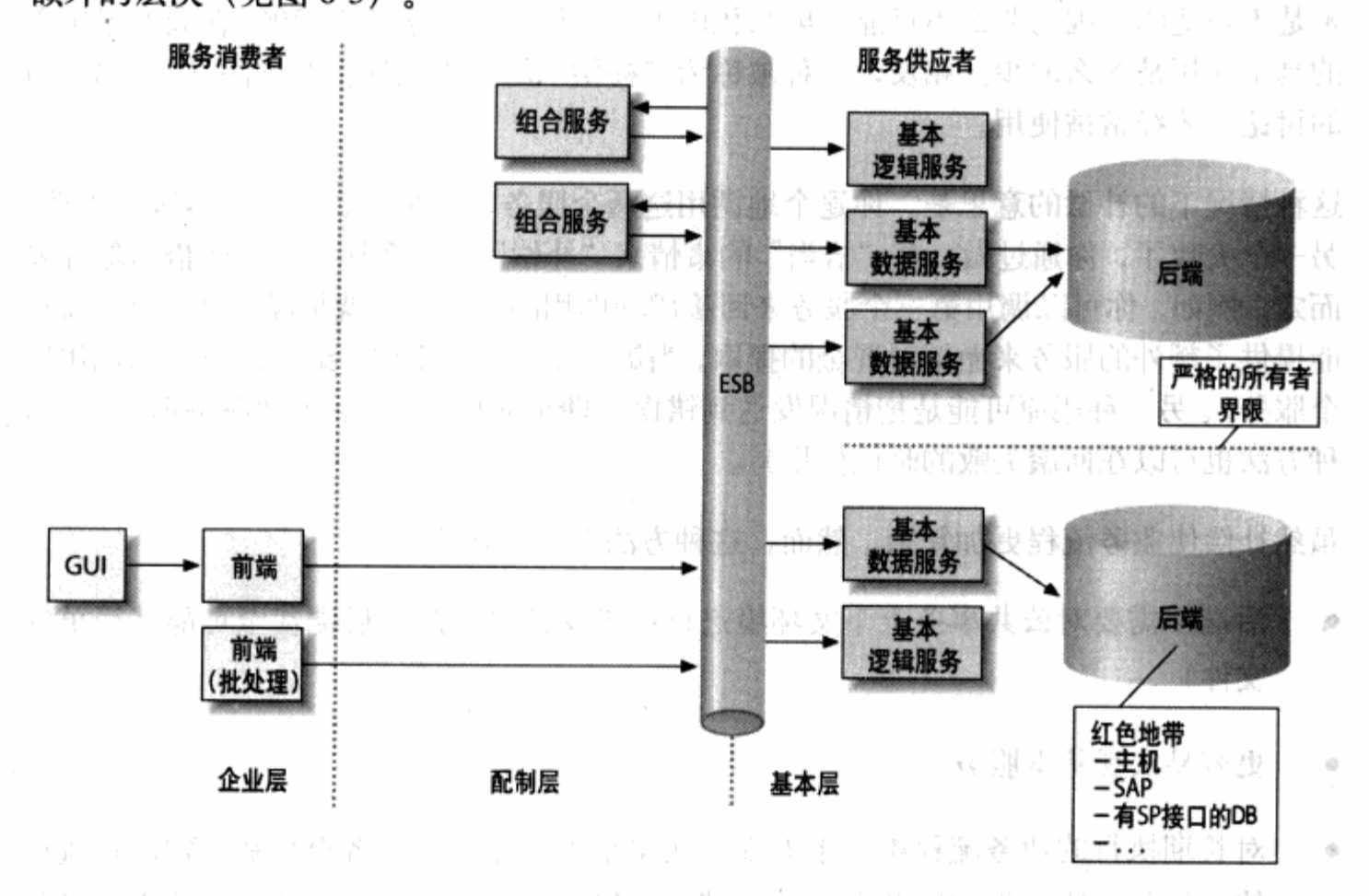


图 6-3：联合的 SOA

正如之前讨论的，组合服务的一个典型例子可能是，通过调用所有保存地址的后端上恰当的基本服务，对客户地址作出全局性更改的服务（可能的实现见 15.1.1 节）。该服务必须做到：

- 知道应该通知哪些后端更新地址。
- 将新地址属性映射到不同后端每个基本服务所提供的特定属性上。
- 进行错误处理（这就是说，如果一个或多个地址更新失败，要作出恰当的反应）。

上述最后一点是为了保证所有后端的一致性。虽然在这个场景中，所有后端不在同时更新可能问题不大，但是，对其他业务流程来说，总体一致性非常关键。例如，在从一个后端向另一个后端转账的例子中，负责从一个后端提取现金的基本服务，以及负责向另

一个后端存入现金的基本服务都必须成功，否则，作为一个整体的转账就失败了。承担这个任务的组合服务必须合理编程，这样在服务调用结束时，要么两个服务都成功，要么两个服务都失败。

注意，人们通常被传授的是，要处理这样的需求，应该对两个基本服务调用使用一个公共的事务上下文环境，通过执行两阶段提交（2PC）来加入事务安全性。这样的想法经常是不合适的，因为要么不可能，要么代价太高。你可能会吃惊于 2PC 在大型系统上的真正使用是多么的少。相反，一种被称为“**补偿**”的，更松耦合的方法（见 4.2.8 节的讨论）才经常被使用。

这种情况下的补偿的意思是，你逐个地调用这两个服务，如果只有一个基本服务成功，另一个失败了，你通过做一些“恰当”的事情来“补偿”。什么样的还原才恰当随环境而定。例如，你可以调用第三个服务来回滚成功调用的影响（这就是说，第一个后端可能提供了额外的服务来撤销对现金的提取，当第二个后端失败时，组合服务可以调用这个服务）。另一种还原可能是把情况发送到错误处理桌面机上，然后手动解决不一致（这种方法也可以在回滚失败的时候使用）。

虽然补偿使业务流程更加复杂，然而，这种方法有几个好处：

- 后端不需要对公共事务上下文环境进行技术支持（注意，后端甚至可能是简单的文件）。
- 更容易实现基本服务。
- 对长期执行的业务流程维护事务安全几乎是不可能的，或者至少是非常消耗资源的。例如，假设你的服务在多个后端上创建一个新客户，这样就能对该客户进行订单处理。在一个事务上下文环境中执行整个流程会要求所有相关系统都共享同一个公共事务上下文环境。

注意，当使用补偿时，后端可能必须把部分业务流程回滚了的事实记录到日志中。假设我们要为一个新客户下订单，你得先在后端系统中创建新客户。如果由于某种原因订单失败，后端系统可能需要保存新客户被创建了这个信息，然后再立即撤销创建新客户的操作。

6.3.4 安全
Security

组合服务的引入提出了另一个问题：安全。ESB 提供了概念上的互操作性。于是，即使提供了一个组合服务，使客户地址能够在整个后端一致地更新，消费者仍然有能力调用底层的基本服务，在每个后端上单独进行修改。这就重新引入了后端之间不一致的危险。

所以，我们需要一些机制来帮助保证：那些基本服务只被更高层的服务调用。实现这样的限制将在第 14 章讨论。

注意，一般对消费者而言，能够调用基本服务仍然是有用的。例如，消费者可能调用一个基本服务来获取客户当前的地址。组合服务层并没有完全隐藏基本服务层。这就是说，从业务的观点出发，服务是基本服务还是组合服务无关紧要，重要的是服务是否满足了必要的业务功能。到底是哪类服务只是种实现细节。

6.4 流程服务
Process Services

扩展的第三阶段增加了流程服务，这代表了长期的工作流或业务流程。从业务的观点来看，业务流程服务代表了“**宏流**”，它是可（被人工干预）中断的长期运行的活动（服务）流。

和基本服务、组合服务不同，流程服务通常有一个**状态**，该状态在多个调用之间保持稳定。（何谓“有状态”见 15.1.2 节的讨论。）

流程服务的典型例子是购物车服务。该服务的状态为购物车的内容，也许还掺杂着点客户的数据，这样一来，客户的订单就可以在多个会话之间被保持和处理。根据具体的设计，流程最终可以在客户订单敲定时结束，或者甚至是在所订购商品发货时结束。

因为流程处理通常有状态，所以，你必须在自己的 SOA 中提供某些特性，使得既可以在多个会话之间保持状态（见 15.1.2 节），又继续提供线性可伸缩性以应对多个服务调用（见 15.1.3 节）。如此说来，你需要提供在处理过程中改变客户端及前端通信渠道的能力。

一个典型的例子是允许客户订购一个新保险政策的服务。客户可能通过公司的互联网门户从网上开始流程。在流程中，其他前端（如内勤办公或呼叫中心客户端）可能会从实际的流程中请求获取数据，然后更新数据。客户订购的政策要么被实现，要么由于某种原因被拒绝，此时流程结束。

流程服务要考虑的另一个方面是失效备援——这就是说，当包含目前状态的一个系统失败时，是否以及如何保持流程状态。这种需求是针对这类服务采用一个单独类别的主要原因。另一个理由是，从业务的观点来看，比起短期运行的组合服务来，长期运行的流程服务往往要求更加健壮，软件质量更好。

虽然[KrafzigBankeSlama04]宣称不区分组合服务与流程服务也行，[Erl05]对两类服务都采用“流程服务”这个术语，然而，怎样的分类才恰当由你自己决定。

6.4.1 流程使能的 SOA
Process-Enabled SOA

引入流程服务带领我们迈进了被称之为“**流程使能的 SOA**”的扩展阶段（再次，这个术语取自[KrafzigBankeSlama04]）。如前所述，额外的流程服务层使你能管理可能从不同的前端发起、由不同的前端控制，能被人工干预打断的流程（见图 6-4）。

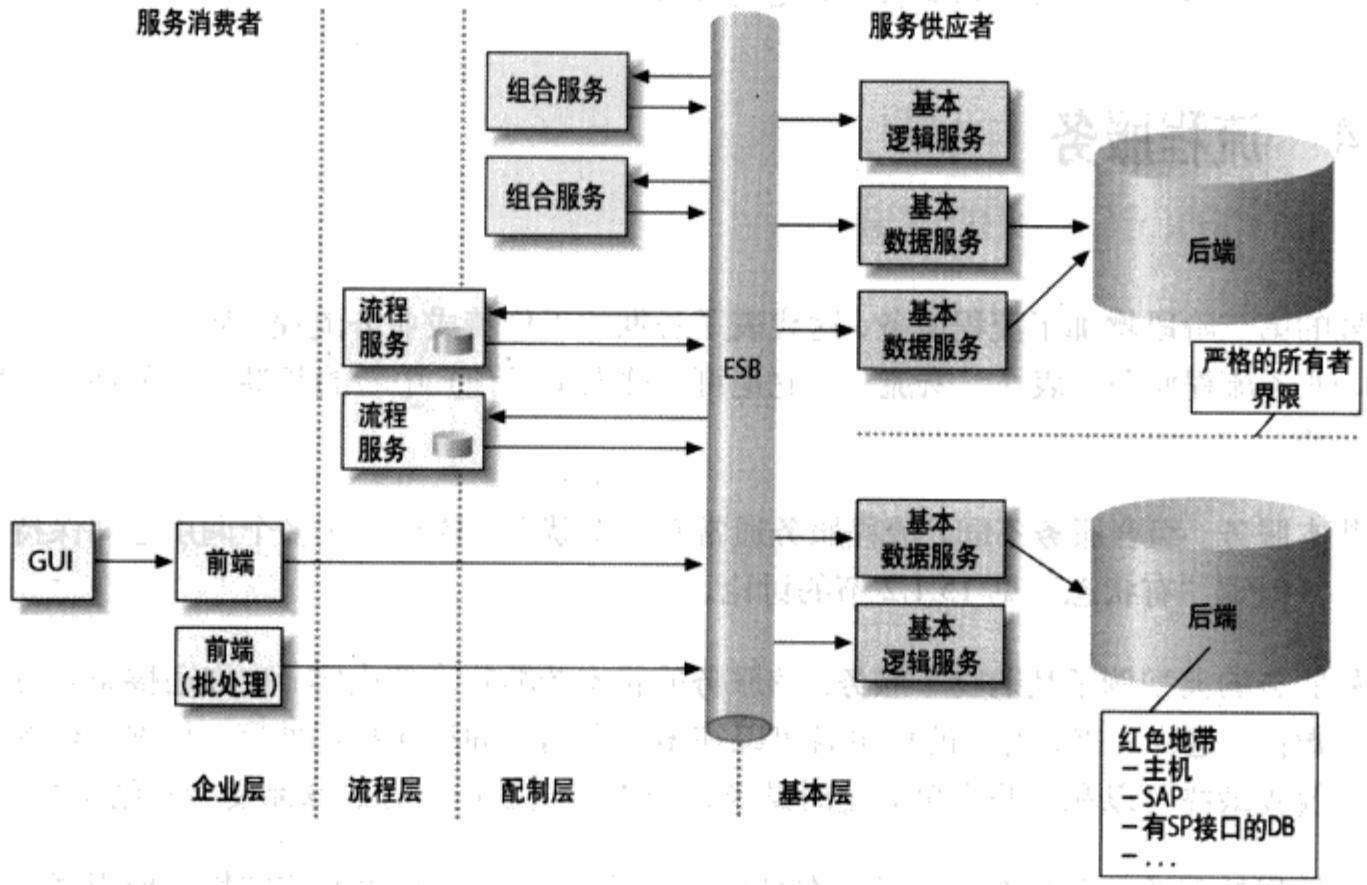


图 6-4：流程使能的 SOA

然而，要注意的是，作为 Web Services 的标准，BPEL 并没有为运行中的流程服务提供任何触发人工干预的能力，这对系统（尤其是前端）的总体设计有重大影响。细节见第 9 章。

6.4.2 服务状态对比后端状态
Service State Versus Backend State

在处理流程服务时，你必须作出一个重大设计决策：业务的状态是保持在后端，还是保持在服务本身当中？该问题决定着对你的业务而言，流程状态的相关性有多大。

- 如果流程数据成了“法律数据”，在平常的后端管理状态也许更合适。
- 如果流程数据只是些持久数据，与你的业务没有本质上的关联，则使用**有状态服务**就行了。

数据有法律关联的典型标志是，消费者和/或调用者被分配了一个唯一的流程 ID（订单号码、记账号码等），这样，她就能在数天、数周，或者甚至数年之后引用该流程。换句话说，如果你必须在今后也能对流程的状态说点什么，流程数据就应该持久化到后端中。

例如，如果你有一个为客户创建银行账户的业务流程，很显然相关数据具有法律效力。即使在流程结束完成之前（如，因为客户在等护照确认或信用评估），你也必须在后端系统中管理客户的数据。

另一方面，如果你有一个购物车服务，使用户可以记录自己想要购买的商品，则直到客户选择下订单和为购物车中的商品付费之前，相关数据都不会变成法律数据。于是，如果用户结束会话，过后又决定继续购物，此时她不得不打开一个新的购物车并不是多大的一件事。然而，为了方便，你可能会在多个会话之间对购物车状态提供持久化。当然，此时你需要一些功能，使客户能在以后找到运行中的流程。如果便利性是你主要的商业优势，或者你想要对客户曾经选择的所有货物（不管买还是不买）进行持久化，这样的购物车就会成为后端系统的一部分，后端系统将提供若干基本服务，供购物车访问。

再次，这是个（被业务需求影响的）设计问题。重要的是要理解，在服务中和在后端中维护状态是不同的。对这点的错误理解很容易导致有状态流程服务默认被当作后端使用的情形，从而造成这些后端被流程服务而不是被基本服务封装的有趣结果。这不会有助于使你的系统景观更加可维护。当引入业务流程和工作流系统时，这是一个始终都应考虑的重要问题。

6.5 其他服务分类法
Other Service Classifications

正如前面提到的，刚刚展示的分类方法只是各种可能性中的一种。除了前面一节介绍的分类和层外，还存在其他一些分类和层。例如，你可以引入特殊的层，把外部服务和内部服务区分开。你也可以根据一定的技术和/或业务规则，把流程层分为不同的子层。或者，根据你的领域模型，你可以用一些组合服务专门处理你的领域，而其他的服务则处理多个领域。

同时请记住，在建立 SOA 时，基础的 SOA、联合的 SOA，以及流程使能的 SOA 都不是唯一可能的扩展阶段。例如，当（使用服务业务库的）服务管理开始起作用时（这有点独立于所处的阶段），就进入了另一个重要的扩展阶段。建立和实现 SOA 的更多细节见第 19 章。

我曾经看到过，也读到过其他一些服务分类，这些对你可能有用。我将在这儿展示其中一些。

6.5.1 处理不同类型的消费者
Dealing with Different Types of Consumers

一种分类服务的方法是根据它们的目标听众来区分。这是指，有的服务面向内部，而其他一些面向公众；同样，有的服务可能用于特定的部门，而有的则用于整个公司。

作出这些区分的原因是，这些服务在安全和稳定性（在接口保持不变的意义上）方面通常有不同的需求。虽然“本地”服务可能不需要任何特殊的安全措施，面向公众的服务却应该在特殊的安全设置下运行（见第 14 章）。反过来说，虽然针对不知名的、大量用户的、公众可用的服务通常在稳定性和健壮性方面有非常高的要求，“内部”服务却允许被更容易地修改。另外，服务等级协议（SLA）可能不同。例如，外部服务的消费者可能要求 24/7 的可用性，而内部用户只要求上班时间内的可用性（或正好相反）。

内部服务和公众可用的服务之间还存在其他一些差异。例如，它们可能使用不同的接口技术。在内部，可能有非常好的理由使用某种专有的解决方案，而公众服务的目标通常是提供一个非常常见的接口（比如 Web Services）。

在所有情况下，不同类型的消费者可能导致采用不同的工具、流程和服务生命周期。使用不同的服务类别来适应这些差异是个好主意。

例如，[KrafzigBankeSlama04]引入了一个称为“公众－企业服务”的服务类别。这类服务要求特殊的安全机制，因为此时缺乏对服务消费者的控制。

我见过的另一种区分是国内服务和国际服务。涉及该问题的公司在不同国家办公，使用不同的系统基础设施，有一部分流程也不相同。国际服务是那些需要在所有系统基础设施中都予以支持，并且/或者覆盖了对所有国家都常用的流程的服务。

与此类似，对相同的服务可能有特殊的需求：在不同上下文环境中应有不同表现（原因可能是不同国家的政治、文化或历史因素）。如何处理这种情况有不同的选择。一个选择是为每个国家设计不同的服务（例如，rating_de(), rating_us()，等等）。此时，服务消费者需要一个方法来被路由到正确的服务。这可以利用 ESB 的路由能力，通过配置管理，或者提供不同的调用实现来做到。另一个选择是给所有这些服务起一个名字，再提供某种形式的多态。这是说，当调用者调用 rating()服务时，某些巧妙的装置会选择正确的实现。通过在不同地点部署不同服务实现，或者通过配置管理，或者依靠基础设施（ESB）的路由特性，都可以做到所要求的多态。

任何情况下，如果你有这样的需求，为这类服务引入一个特殊的服务类型可能是个好主意，因为必须为它们提供特殊的流程和工具。

6.5.2 读取服务对比写入服务
Reading Versus Writing Services

另一种可以作出的明显区分是针对读取服务（不改变相应后端的状态的服务）和写入服务（改变相应后端的状态的服务）。

根据定义，读取服务有特殊的性质，通常情况下引入的麻烦也少些：它们内生就是幂等的（见 3.3.5 和 15.2 节），通常不要求有针对不同后端的事务上下文环境，或在仅仅部分成功时进行回滚。写入服务则相反，通常要求有额外的安全性，这样才能追溯修改。因此，所有写入服务都需要额外的技术数据来标识调用者（这甚至可能导致它们变成有状态的——见 14.3.4 节）。

然而，要注意的是，有的系统对每个读取访问也要求记录日志和具备可追溯性。从业务观点来看不操作数据的服务，在技术上仍然可能与写入服务有很大的共同性，因为它们也写日志或作历史记录。

读取服务通常比写入服务更不稳定。原因是对读取服务的扩展（“喔，顺便说一句，我们能不能得到如下属性，并且把它们显示在前端？”）不影响复杂的业务流程，而修改写入服务的属性可能对许多地方造成影响。因此，写入服务更经常地用在组合或配制的服务中。读取服务主要被前端用来显示数据（细节见 9.2 节）。

我的一个客户对区分读取服务和写入服务还提出了一个理由。当他们把系统升级到新版本时，系统停止运转了几个小时。然而，他们的大多数服务是读取服务，这就意味着他们可以在升级时把数据镜像出来，继续提供服务。只有写入服务才必须关闭。因为针对读取服务和写入服务的部署规则不同，所以将服务如此分类有其意义。

6.5.3 按业务分类
Business Categorizations

迄今为止介绍的分类方法，其动机主要是技术上的（这就是说，分类为如下事实所驱动：不同类型的服务被流程和工具进行不同的处理）。换句话说，这些分类导致你基础设施的不同（包括相应的流程）。

当然，从业务的观点来看，服务不管是在角色还是职责上也都不相同。有几个出版物根据这种观点来引入服务分类。下面我将展示其中几个。

Erl 的分类

在[Erl05]中，Thomas Erl 介绍了服务的一种分类法，其术语类似于面向对象建模中对类的一种可能的分类法。作为定义不同 SOA 层的基础，他引入了如下服务分类：

- **应用服务**。这是“包含从一个解决方案或技术平台继承的逻辑”的服务。
- **业务服务**。这是“包含业务逻辑”的服务，和我们之前介绍的基本服务相似，业务服务可以进一步细分为：

  ——以任务为中心的业务服务

  ——以实体为中心的业务服务
- **流程服务**。这是“代表了由一个配制平台实现，为一个业务定义所描述的业务流程。”

虽然你可能会说这种分类法和我之前介绍的区别不大（业务服务对应于基本服务，流程服务包含了组合服务），然而，“应用服务”类型引入了某些不确定性。首先，这看起来是个不提供业务功能的服务类别，而这在一定意义上就违反了该术语的定义：一个服务永远都应该体现某种业务功能（实际上，至少有一些服务与该原则背离也不是那么罕见的事情，更多信息见 6.6 节）。其次，这种分类导致可能出现既包含来自于解决方案/技术平台的逻辑，又提供业务逻辑的服务。[Erl05]为这种服务引入了一个新术语：**混合服务**。

[Erl05]还引入了更多的其他服务类别：

集成服务

这是“用于跨应用集成目的，对解决方案环境而言是个端点”的应用服务。

包装和代理服务

这是“封装和暴露”或镜像了“存活在老系统中的逻辑”的集成服务。

工具服务

这是“提供了可重用逻辑”的应用或业务服务。

控制器服务

这是“对服务构成成员而言是父服务”的服务；换句话说，它们协调其他服务来“为一个总体业务任务的执行作出贡献”。

其他使用过的服务类型术语还包括“**激活服务**”、“**协调者服务**”、“**注册服务**”等。

所有这些术语都和服务的一定角色、特性或能力相关。然而，类别的不同不会导致基础

设施的差异。因此，我的观点是，这些术语从业务模型的观点来看更加重要些，而从架构观点来看则并非如此（这也并非意味着它们不重要）。

找到对你最有用的分类服务的方法要靠你自己。即使在同样的项目和系统中，不同的分类法可能都会有所帮助。重要的是研究自己系统的细节，以及对各种可选的分类法进行仔细斟酌。

Allen 的分类

在[Allen06]中，Paul Allen 也指出了三种关键的服务类型。然而，在这里，业务案例方面的考量很重要，如风险和市场差异化（见图 6-5）。

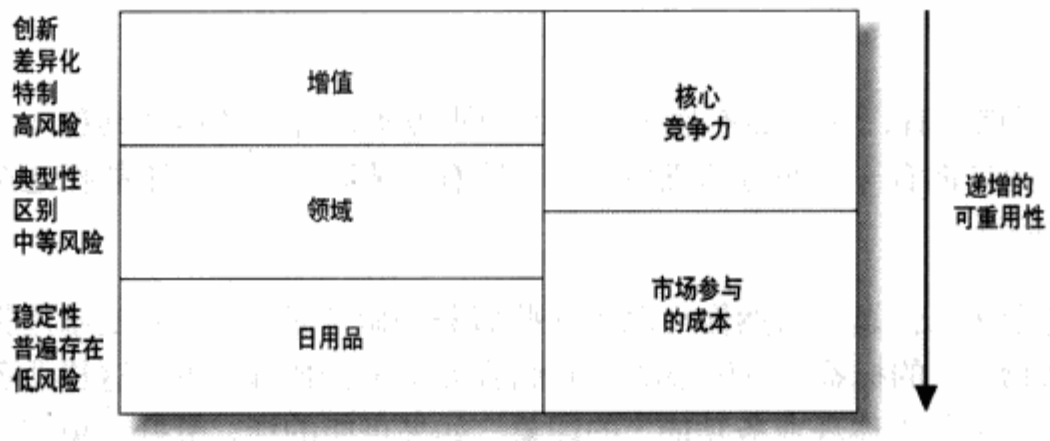


图 6-5：依据[Allen06]，各服务类型的特点

关键的三种类型如下：

- **日用品服务**。这是每个商业市场的参与者都必须拥有的一般性服务。这些是普通的、稳定的服务，供应充足，使用风险低，建立服务时考虑了最大程度的复用性。这些服务是外包功能或标准产品的候选者。
- **领域服务**。它代表了某个特定业务关键的、核心的功能。它们是常用的，但同时又易变。可举的例子是提供业务规则的服务。
- **增值服务**。这是反映了组织带给市场的特殊价值的服务。它们把公司同竞争者区分开，通常有高度的创新性。

如此分类服务使你可以对服务获得如下方面的认识：服务的稳定性、服务对公司的重要性，以及引入和维护服务包含的可能的风险。

6.6 技术服务和基础设施服务

Technical and Infrastructure Services

我一直建议说，一个东西能被冠以服务这个概念，关键要体现出某种业务功能。但是，实际上在绝大多数 SOA 景观中，都提供了些不体现业务功能的常用服务。它们通常被称为“**技术的**”或“**基础设施**”服务。

这样的服务通常利用服务基础设施来提供某些“额外的”功能。这就是说，如果你知道如何调用一个业务服务，你就可以用同样的接口来做一些有用的，但不一定非和业务相关的事情。典型的例子是，服务提供技术方面和基础设施方面的支持。这些服务可能是：

- 查询部署信息。
- 监测运行时统计数据。
- 打印、日志或跟踪。
- 使能或禁用组件和系统。
- 验证接口。

从纯 SOA 的观点来看，这些都不是服务，因为服务永远都应该体现某种业务功能。换个角度来想，如果你有一个服务基础设施，当确有必要时，为什么不能把这个基础设施用于其他目的呢？

本质上，这是个术语问题。虽然你可以把这些服务叫做“服务”，其实永远都应该给它们打上“特殊物品”的标签。应该很清楚它们服务于某种特殊目的，这样业务团队才不会有“服务不一定非得体现业务功能”的印象。如果可能的话，最好避免把它们明确地叫做服务。

[Erl05]引入的应用服务类型（见本章前面“Erl 的分类”）提出了相似的问题。典型的例子是老系统的服务接口（这就是说，映射现存的数据存储过程、3270 终端转义序列，或者 SAP BAPI 的服务）。在历史上，这些东西可能并不体现自足的业务功能，但从技术上，你可以把它们作为自己服务景观的起始点。再次，对造成“服务不一定体现自足的业务功能”这种印象要小心，因为维护一个隐藏技术细节的、清晰的抽象层非常重要。

6.7 服务之外
Beyond Services

对分布式系统的景观而言，服务意味着一切吗？

不，当然不。回忆一下，SOA 概念针对的是所有者各异，分布到多个后端系统上的业务流程。你的公司中还会存在其他一些流程。例如，如果 CRM 后端有一个特定的客户端，它维护着后端，或者它只和这个后端的数据打交道，那么，让这个客户端只能通过服务和这个后端通信的做法通常说来并不恰当。无论从复杂性还是性能来看，这样做的代价通常都过于高昂。对这样的客户端来说，什么样的实现技术才恰当，这是另一个问题。当然，你可能从 SOA 引入的概念抽象性上获益，但是，不要把服务变成你的“黄金锤”，从而把每个问题都当钉子处理。

同时牢记，类似于在多个后端间同步冗余数据这样的任务并不是分布式业务流程。你可以用服务基础设施来执行这些任务，但不要据此得出结论说它们是面向服务的。

6.8 总结
Summary

- 服务通常归于不同类别。在你的上下文环境中，要靠你自己找到不同服务的合适类别。
- 基于如何分类服务，你可以定义相应的服务层及 SOA 扩展阶段。
- 基础的服务分类法分为基本、组合和流程服务类别。相应地又扩展为基础的 SOA、联合的 SOA 和流程使能的 SOA。
- 基本服务永远都只应属于一个后端系统，只对该后端内的一致性负责。
- 组合服务与流程服务对多个后端间的一致性负责。
- 在多个后端间强制一致的通常方法是补偿，而不是事务安全或两阶段提交（2PC）。
- 当设计长时间运行的流程时，是把它们实现为有状态流程服务，还是在后端保持状态，提供基本服务来执行状态迁移，这点需要仔细考虑。
- 其他的分类方法是可能的，也是有用的。它们不一定非带来清晰、明确的服务类别不可。
- 你的 SOA 景观有可能包括技术或基础设施服务。严格来讲，它们不满足服务的主要要求——提供自足的业务功能。仔细考虑如何命名和处理这些服务。
- 服务不是解决分布式系统间任意通信问题的银弹。服务的主要目的是使分布式业务流程成为可能。在主从间同步冗余数据，解耦前端，诸如此类事情是不同于分布式业务流程的任务。

同时呈现，然而了在多个后端间同步几个数据库的任务并不是分布式业务流程。你可以用服务来协调那些执行这些任务，但不要掩饰出错的风险它们是面向服务的。

6.8 总结

- 服务通常有不同类型，在你的上下文环境中，要清楚自己找到不同服务的合适类别。
- 基于如何分类服务，你可以定义相应的服务层及 SOA 扩展阶段。
- 基础的服务分类法分为基本、组合和流程服务类型，和应用文扩展为基础的 SOA、聚合的 SOA 和流程使能的 SOA。
- 基本服务不应该只应属于一个后端系统，只对该后端的一致性负责。
- 组合服务封装服务对多个后端间的一致性负责。
- 在多个后端间编制一致的通常方法是补偿，而不是事务完全成熟的原子性 (2PC)。
- 当设计长时间运行的流程时，要把它们实现为有状态流程服务，还是在后端保持状态，提供基本服务来执行状态过渡，这点需要仔细考虑。
- 其他的分类方法是可能的，也是有用的，它们不一定非带来清晰，但编制服务类别不同。
- 你的 SOA 背别有可能包括技术性基础设施服务，严格来讲，它们不属业务的主要变来——提供自定的业务功能，仔细考虑如何命名和处理这些服务。
- 服务不是解决分布式系统问题的灵丹妙药。服务的主要目的是使分布式业务流程成为可能，并且及时半写业务逻辑，解耦前端，指如此类事情是不同于分布式业务流程的任务。

第 7 章

业务流程管理

Business Process Management

在 SOA 中，典型情况下，服务是一个或多个分布式业务流程的组成部分。如此说来，服务的主要动机来自于业务流程。当然，随之就产生了如何识别服务的问题。这就带来了业务流程管理（BPM）这个术语。

本章从 SOA 角度出发讨论 BPM，话题包含业务流程建模和配置。我会介绍一些 BPM 工具和标准（比如 BPEL），对根据新的业务需求识别新服务，我还会展示出一些可能的过程。

我们都知道，已经有很多专著和专门的网站在论述 BPM。本章只是个概览，集中于在 SOA 上下文环境中开发服务（用我的经验加了点佐料）。接下来的两章会探讨此处讨论的内容在组织上、技术上产生的结果。

7.1 BPM 术语
BPM Terminology

让我们从一些术语开始。对于业务流程的管理和建模存在着一点认识混淆，所以，我试图首先稍稍澄清一下这些东西。

混淆从 BPM 缩写词自身开始，该缩写词可以代表两个不同的东西。通常，它代表“**业务流程管理**”，这是个一般性术语，泛指所有与管理和改进业务流程相关的活动。正如[BloombergSchmelzer06]所陈述的：

> 业务流程管理通常指一种用技术使能的手段，目的是让公司对多步骤、长期存在、横跨诸多系统，以及牵涉一个或多个组织的人员的流程获得可见性并实施控制。

除此之外，BPM 还可以代表“**业务流程建模**”，这是对业务流程或其部分进行建模。根据[BloombergSchmelzer06]的说法：

> 业务流程建模是公司能够执行的一系列实践或任务，目的是可视化地描绘或描述业务流程的所有方面，包括它的流动、控制和决策点、触发器和活动执行的条件、活动可以运行于其中的上下文环境，以及关联的资源。

根据这些定义，业务流程建模通常是业务流程管理的一部分。如此一来，我也就把本章称为业务流程管理，然后在本章中包含一些讨论业务流程建模的小节。当我使用 BPM 缩写时，我通常指的是业务流程管理。

另一个混淆的根源可能是业务流程管理和工作流管理之间的差异。这两者都鼓励主要用面向流程的观念来看待系统和组织，这两个名词经常互换使用。如果你想要在两者之间作出区分，你可以认为业务流程更具一般性：

- 业务流程描述该做些**什么**（包括输入和输出）。它可能包含手动的活动，可以使用任何形式的资源。
- 工作流描述了**如何**达到一定的结果。它更深入关注所有步骤和活动的细节。

这样一来，你会说可以把业务流程拆成更小和更具体的步骤，然后，在某种抽象层次上，这些活动和流程步骤是某“**工作流层**”的一部分。图 7-1 给出了这种想法的一个可能的呈现，但是要注意，层的确切数量和所用术语可能不同。例如，孙流程可以被称为“任务”，而有人会认为任务是工作流层的一部分。

SOA 标准也被这种术语混淆所影响。例如，根据这种区分，本章后面要讨论的业务流

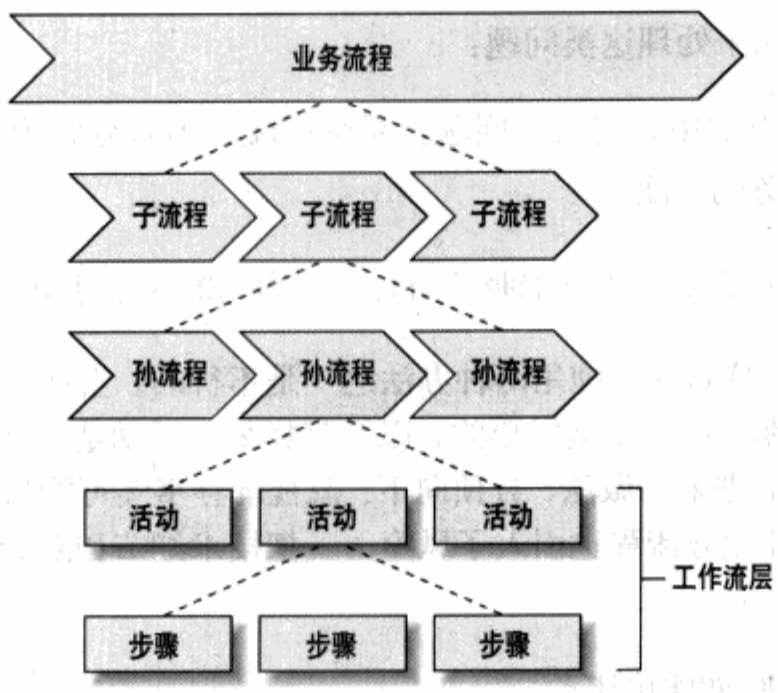


图 7-1：有工作流层的业务流程层次结构示例

程执行语言（BPEL）将会是工作流层的一部分。然而，因为 BPEL 用了业务处理的一般术语，我将在本书通篇使用“**业务流程**”术语，即使有时（根据某些人的意见）严格来讲指的是工作流。

7.2 BPM 和 SOA

BPM and SOA

服务是业务流程的组成部分。因此，要让服务发挥作用，你必须先考虑业务流程。

然而，业务流程管理是个巨大的课题。它处理很多方面的问题，比如分析业务（包括需求和机会）、实现和集成业务战略、监测和最优化业务流程、建立相应的工具和文化，以及让业务与 IT 结盟。

说起与 SOA 关联的 BPM 时，很显然，流程分解后，最底层的活动就是服务。从业务的观点来看，这些服务是基本服务还是组合服务无关紧要，重要的是服务执行了必要的业务功能。然而，流程服务是不同的东西，因为流程服务的目的是体现整个业务流程（或其某个部分）。

现在的问题是，你如何识别要代表或成为业务流程一部分的服务？这里，我们谈论的是关于系统设计一般性、基础性的话题。你如何把系统或功能分解为小的组成部分（称为服务），从而可以实现系统或功能？以及，这些服务如何才能被设计得更具一般性，这样你才能够在不同的场景中使用它们？

通常说来，有两种办法来处理这类问题：

- 在“**自顶向下**”方法中，将一个问题、系统或流程不断分解为更小的块，直到你抵达（基本）服务的层次。
- 在“**自底向上**”方法中，依靠把服务组合为更通用的块来建立业务流程。

所有专家一致认为，在实践中，纯用两种办法之一是不行的。当然，你可以自顶向下设计流程，这会有助于你了解什么是需要的，以及怎样划分活动是“自然”的。然而，比起把现有的底层也考虑进来的做法，自顶向下、忽视既存事实可能带来高昂的代价。反过来说，自底向上设计业务流程也引入了风险——把技术细节和限制加给了高层，使流程不灵活、不直观。

[BloombergSchmelzer06]如此论述：

> 你的 SOA 方法应该既是自顶向下（通过流程分解），又是自底向上（以服务方式暴露既存功能，并把它们组合成流程）。如果你仅仅采用自顶向下方法，你很可能建议构造一些技术上困难，或实现上复杂的服务。仅采用自底向上方法，会产生你不需要的服务，或者甚至更糟，产生不满足业务需求的服务。

于是，在实践中，对两种方法进行混合是恰当的（对这种混合方法有很多不同的名字，如“中心分开”，“中心到底层”，“中间会合”，或者仅仅是“敏捷”）。

通常，你的目标应该是满足一个真正的业务需要。这就是说，应该出现一个新的或修改过的业务流程或新的功能。然而，你也应该重视已有的东西（改装 IT 现实）。

7.3 用于服务的 BPM 的例子
Example for BPM with Services

让我们瞧瞧用 SOA 实现新业务流程看起来是怎样的。

假设你想要实现一个新的业务流程。在顶层，你有新的业务流程；在底层，你有既存的后端，该后端能提供一些用服务方式体现的基本业务功能（见图 7-2）。

典型地，你会从自顶向下的分解开始：

- 决定哪些流程步骤是手动的，哪些是由 IT 支持的。
- 按照执行**时间**和负责的**系统**，把整个流程分解为小块。
- 把复杂部分分解为更容易处理的步骤。

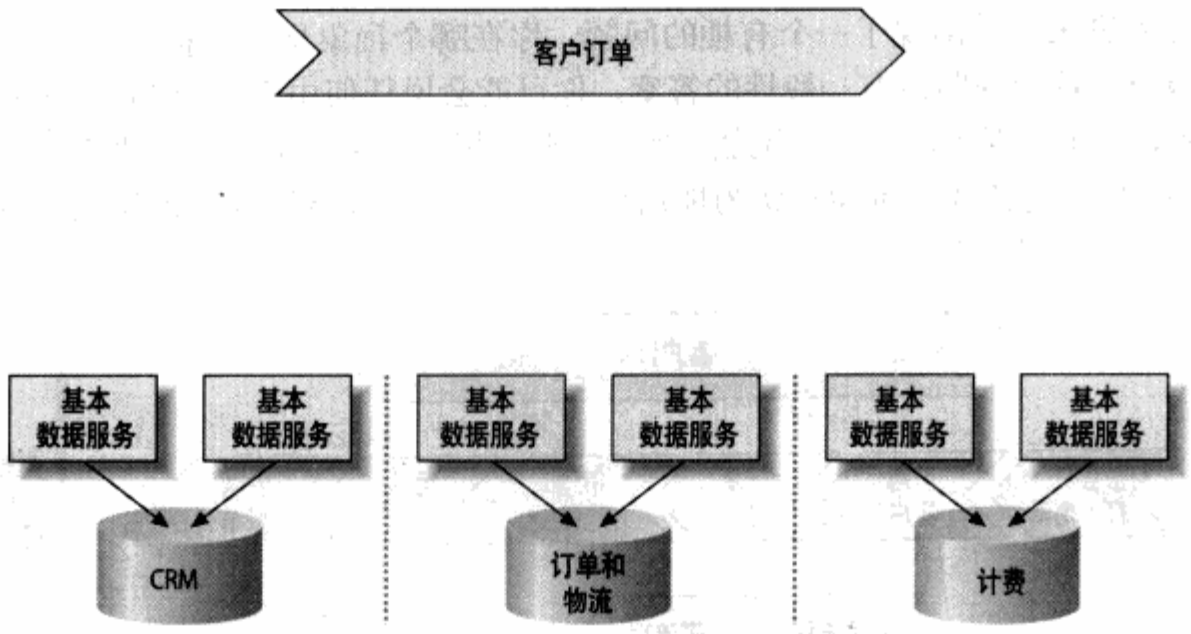
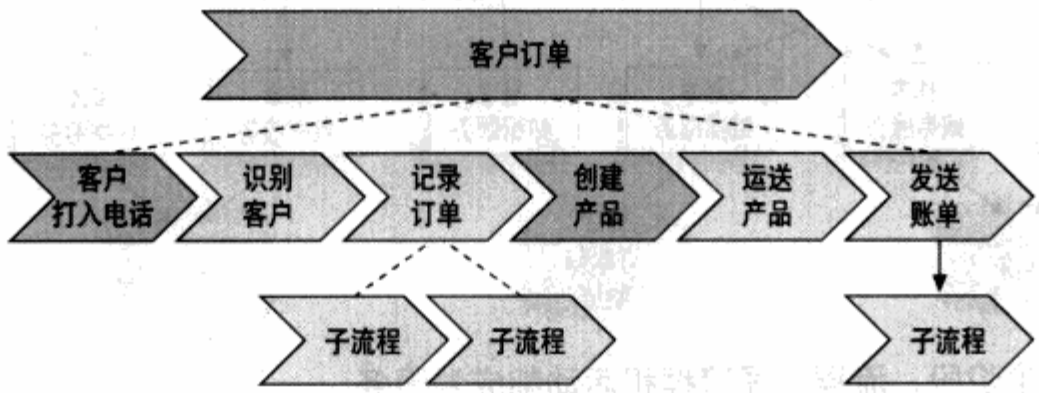


图 7-2：新业务流程的开始点

这项任务（可以被称为“**高层设计**”或“**解决方案设计**”）的结果是，有了实际情况下应如何执行流程的粗略想法（见图 7-3）。需要这个设计过程的一个重要理由是，该过程决定由哪个 IT 系统对流程的哪个部分负责。要记住我们是在讨论分布式流程。要想处理更多细节，你必须首先指明涉及到的系统，这样你才知道自己应该和谁去讨论细节。



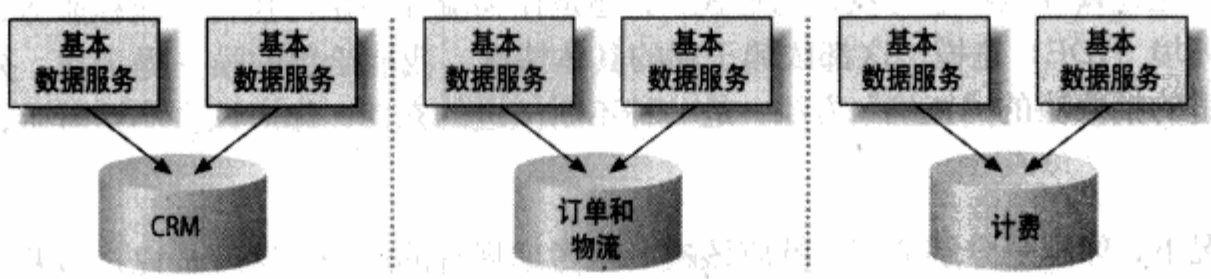


图 7-3：高层或解决方案设计

当服务加入进来时，出现了一个有趣的问题：你在哪个抽象层次上把子流程和/或活动设计成服务？这个问题没有一般性的答案。你可能会说任何由 IT 支持的活动都该是服务，或者，你也可能不同意这个说法。例如，业务规则可以抽取出来，建模到专门的规则引擎中，你可能认为也可能不认为规则是种服务。你必须就设计流程和识别服务发展出自己的过程（见图 7-4）。

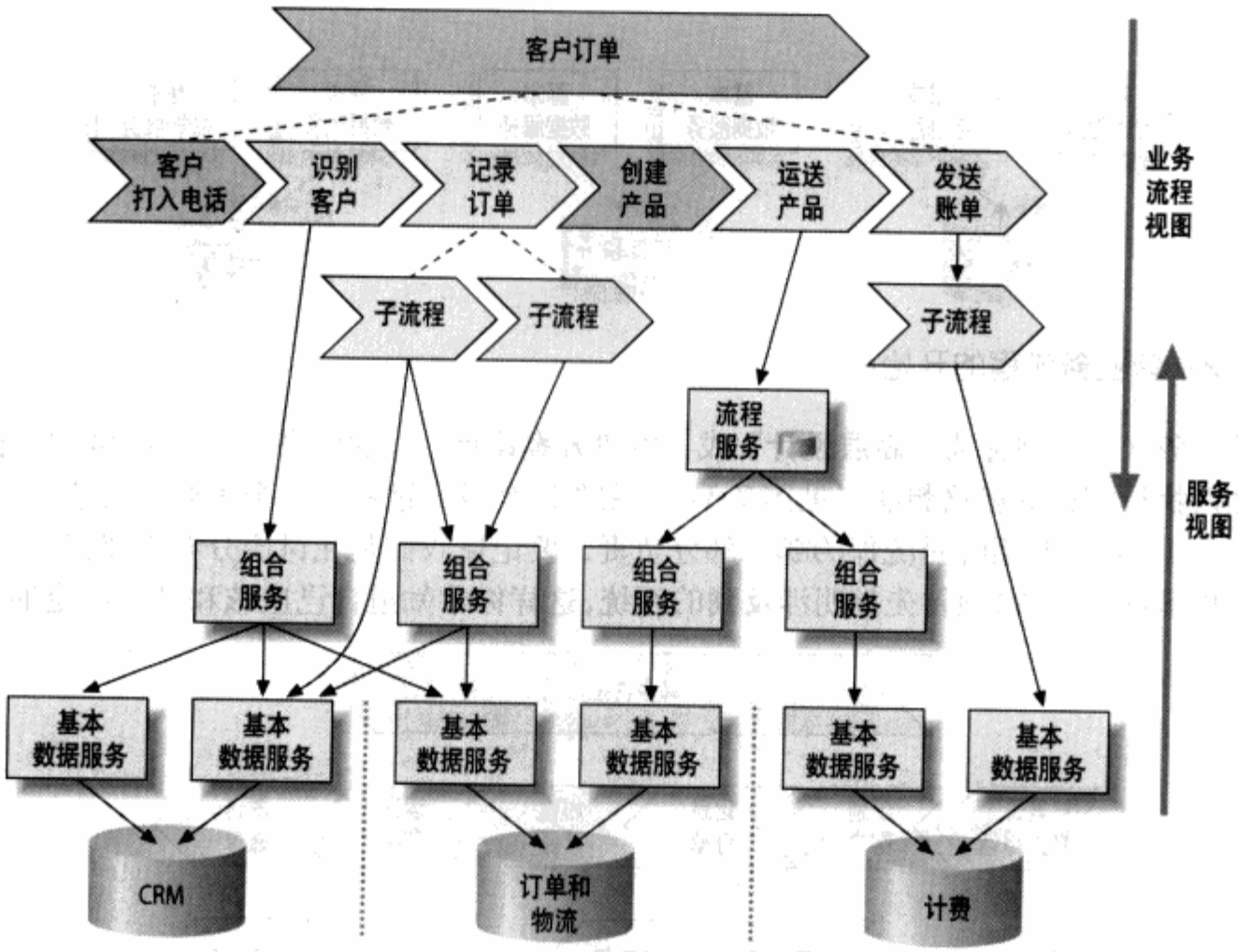


图 7-4：在某个阶段，流程、子流程和活动都成为服务

配套工具影响很大。当你开始使用工具来设计和执行作为流程一部分的服务时，毫无疑问，你处于服务层。但是，特定流程或子流程是仅仅靠手动绘制，由一个业务流程建模工具图形化建模（但不是可执行的形式），还是作为流程引擎可执行的流程服务，由集成开发环境（IDE）建模，这都依赖于你的具体情况。我能给出的唯一提示是，分解功能到合适的抽象块的“正确”方法，是一种不断演进的东西。你应该自小处着手，逐渐成长。

任何情况下，你越是分解流程，就应该越多地考虑现有的服务。这就是说，如果你需要客户数据，你应该寻找满足自己需要的现有服务，而不是只考虑规定一些新服务来提供所有需要的数据。然而要留意，现有服务完成任务的方式与你的要求之间也许有细微的

差别，而这可能会对你的流程设计造成影响。例如，在流程相对晚期，你可能需要某个属性，但是，如果你在流程早期调用的某个服务已经返回了这个属性，你就不必在晚些时候再次调用服务。知道这点后，你就必须把流程设计得能够在服务调用返回后，直到你真正需要之前都保存属性的值。

作为另一个例子，写入服务可能需要一些你原来打算在流程晚些时候再创建或请求的值。此时，你可能需要修改流程来适应现有的服务，或者，你可能要增加一个能在没有该数据的情况下执行功能的服务，然后在以后提供另一个服务来照原计划那样设置该数据。

你如何才知道自己做对没有？对这个问题有个重要的答案：通过学习和经验（可能以设计原则、模式和政策的形式记录下来）。

一般说来，从任何可以工作的，并且看起来还合适的东西开始可能是个不错的主意。比起展开一个巨大的研究项目，专门找出设计第一个服务的最佳方式来说，这种方法通常更好些。记住，尽善尽美通常代价不菲。如果以后发现设计出了问题，你可以修改它们。为了不断改善，你整个的业务都需要灵活性，并且，灵活性也使你能从经验中学习。然而，要注意你必须花时间来检查和理解什么事情做错了，什么事情干得好。

我们在这儿真正谈到的是“**迭代开发**”。我认为，迭代开发是在合理的时间内实现复杂系统的唯一方法。使用这种方法，你从整个系统中抽取出的一小部分开始，实现出一个合适的解决方案。然后你逐步增长，加入新功能，重新设计现有解决方案，重构现有行为。结果就是，你的系统变得越来越好，用恰当的方式满足了越来越多的需求。然而，你必须接受如下东西：永远无法达到完美，任务永远无法彻底完成（除非你的系统或业务崩溃了）。关于建立 SOA 的更多细节见第 19 章。

7.4 业务流程建模
Business Process Modeling

在前一节，我用了一幅图来画出业务流程的不同层次和相互关系。这是业务流程建模的一个例子。[BloombergSchmelzer06]对（业务）模型提供了如下的一般性定义：

> 一个模型实际上是一种针对流程的表示法，它使得公司可以归档、模拟、共享、实现、评估和持续改进自己的运营。

通过 SOA，这些模型扮演了重要的角色，因为标准逐渐演化得能支持使用不同的工具进行业务建模。有的工具甚至允许你通过流程引擎执行建模后的流程。

7.4.1 使用业务流程建模工具

Using Business Process Modeling Tools

图 7-5 展示了业务流程建模工具或引擎可以扮演的一般性角色。这样的工具可以用来从现有的服务中组合（“配制”）出新的组合服务或流程服务。另外，在理想状况下，使用模型的抽象层，你也许可以在业务流程引擎中执行和监测产生出的组合服务或流程服务。

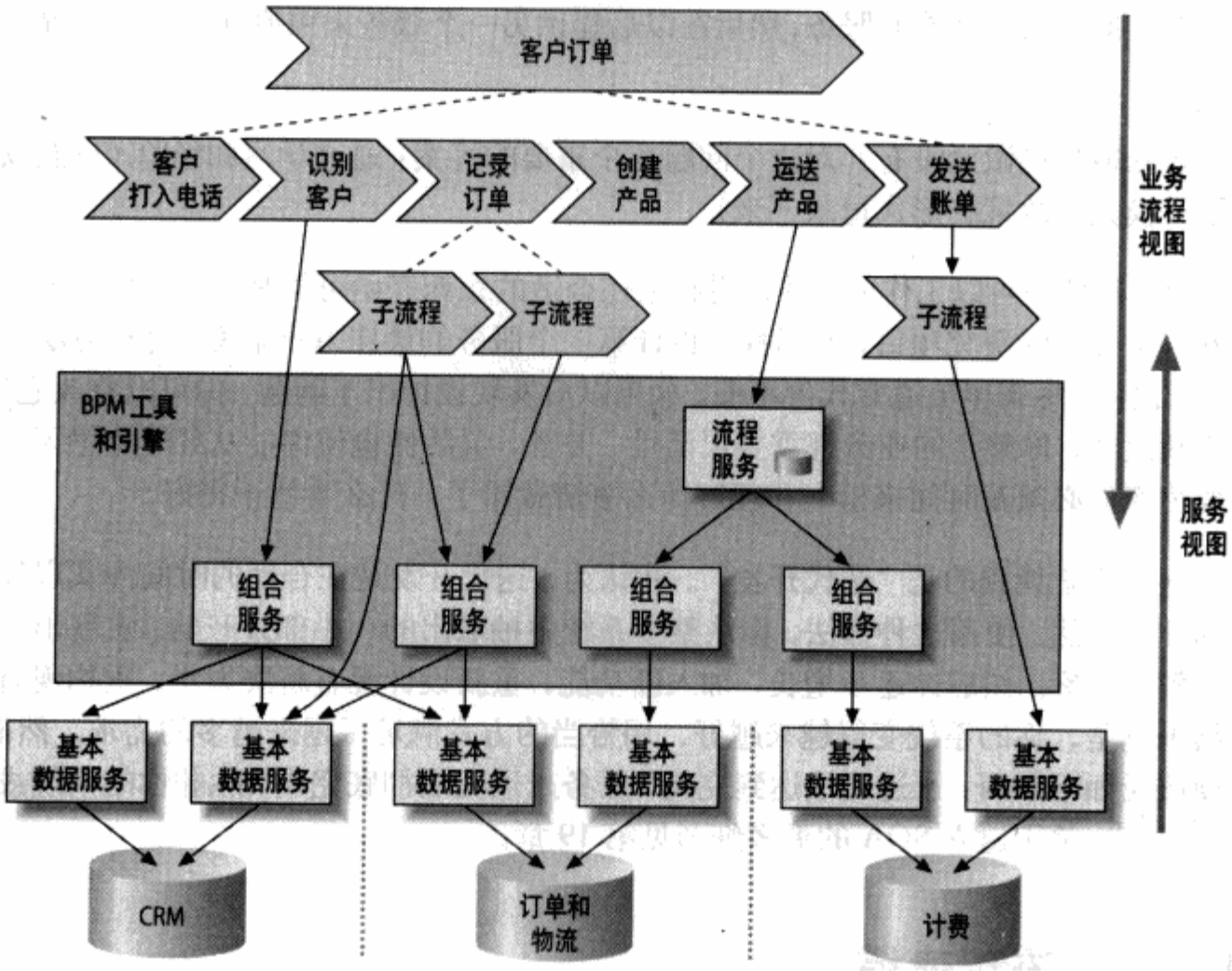


图 7-5：使用 BPM 工具和引擎，建模和执行组合服务、流程服务

图 7-6 展示了这样的 BPM 工具的一个例子（ActiveBPEL，一个开源业务流程引擎，见[ActiveBPEL]）。主工作区显示出处于开发中的当前业务流程（服务）。环绕着主工作区有其他一些区域，分别排列着创建流程所需的现有服务和流程元素。你也可以在引擎中调试和监测流程的运行情况。

长远的愿景是，通过向一个图形化模型上拖放活动和控制结构，用户可以“即插即用”业务流程。该方法将使业务专家（而非 IT 专家）能够设计、修改、监测和调试整个业务流程或其部分（一个由工具供应商提出的，承诺要填平业务/IT 鸿沟的，相当古老的愿景）。

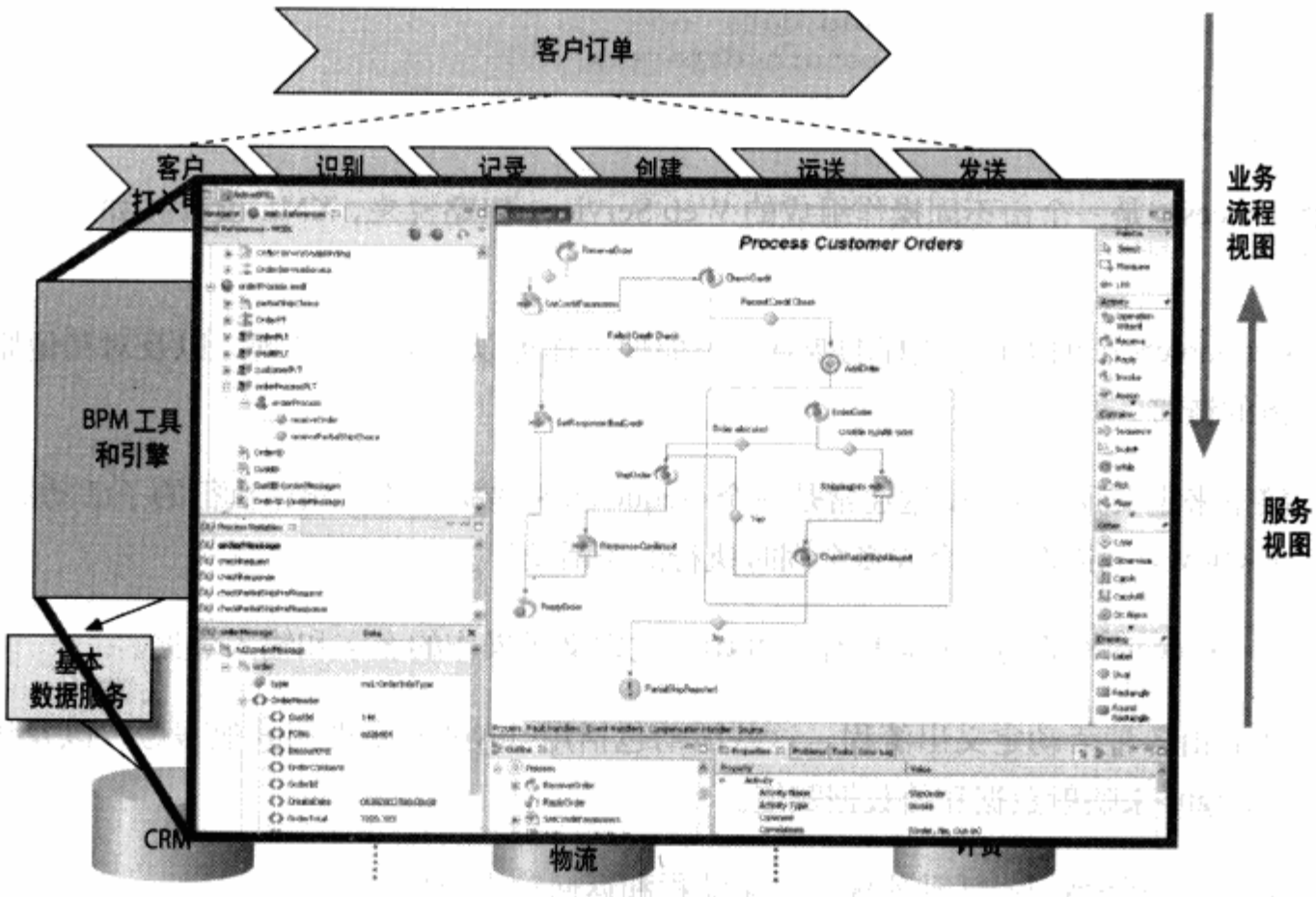


图 7-6：用诸如 ActiveBPEL 这样的工具建模

7.4.2 BPEL

使用工具和引擎进行建模和运行业务流程时，业务流程执行语言扮演了基础的角色。在 SOA 运动中，BPEL 的势头是如此之猛，以致它迅速变成了设计和运行业务流程的标准。

从概念上来讲，BPEL 是一种描述业务流动和顺序的 XML 语言，而业务流动和顺序本身就是服务。语言元素提供了各种能力，如调用服务、处理响应，以及处理流程的变量、控制结构（包括定时器）和错误（包括补偿）。

BPEL 最初由 IBM、微软和 BEA 共同设计，结合了 IBM 的 Web Services 流程语言（WSFL，见[WSFL]）和 XLANG，后者是一种基于 Web Services 的业务流程自动化符号。现在，OASIS 负责阐释标准（细节见[WS-BPEL]）。

一个简单的 BPEL 文件看起来会像如下所示：

```
<?xml version="1.0"?>
<process name="changeAddress"...>
  <variables>
<variable messageType="..." name="...">
  </variables>
  <flow>
<receive.../>    <!-- for this request (operation and input) -->
<invoke ... />   <!-- call other service -->
```

```
<assign .../>    <!-- map data -->
<reply .../>     <!-- return data -->
  </flow>
</process>
```

整个<process>是一个由不同操作组成的 Web Service。粗略说来，XML 文件按如下方式组织：

- <variable>节可以定义流程的变量。类似地，你可以定义错误处理，以及对其他服务的链接。
- 接下来定义顶层活动，这通常是一个<sequence>（按序排列，顺序执行的子活动集）或<flow>（定义了一个或多个同时执行的子活动）。
- 一个<receive>活动通常是操作的开始，它定义了操作的名字和输入数据。
- 然后由各种活动定义出流程。这些活动包括用<invoke>来调用其他服务，以及用<assign>来映射数据和给数据赋值。
- 最后，一个<reply>活动使你能结束流程和返回输出数据。

虽然从原则上讲，BPEL 是可读的，但是，人工读写 BPEL 文件太麻烦，也容易出错。通常，你应该使用一个 BPEL 工具来设计和执行流程。

注意，BPEL 并未对符号进行标准化，所以，在不同的工具和引擎中，BPEL 流程看起来可能非常不同。然而，BPEL 流程的行为应该被标准化到与编辑它的工具、执行它的引擎无关的地步。

在被重命名为 WS-BPEL（遵循通常的 Web Services 命名习惯）之前，BPEL 最开始的全名是 BPEL4WS。之所以遵循 Web Services 命名习惯，是因为用它创建的“流程”（可能仅仅是组合服务）按 Web Services 方式提供。于是，要调用这些流程和服务——它们本身又可能被其他流程和服务调用——你必须能够调用 Web Services（该能力在技术上由 BPEL 引擎内部提供）。

这就产生了一个问题：这种业务流程建模标准是否只能应用于 Web Services 技术？答案是否定的。用 BPEL XML 文件方式组合的流程和服务包括两个部分：一部分描述业务流程的结构，另一部分定义一个绑定，把该结构与具体被调用或被使用的技术操作关联起来。提供了到 Web Services 的标准绑定，然而，你也可以使用其他绑定。这就意味着，在实践中，你组合的流程和服务可以使用不同的中间件，甚至使用如 J2EE 调用这样的本地技术（对于将数据库查询当作基本服务使用，Oracle BPEL 工具提供了直接的支持，这并不让人感到奇怪）。当然，在切换到其他流程引擎时，你必须确定这些绑定的可移植性到底如何。

7.4.3 其他业务流程建模标准

Other Business Process Modeling Standards

BPEL 并不是唯一可用的业务流程建模标准。图 7-7 提供了一个概览（注 1）。

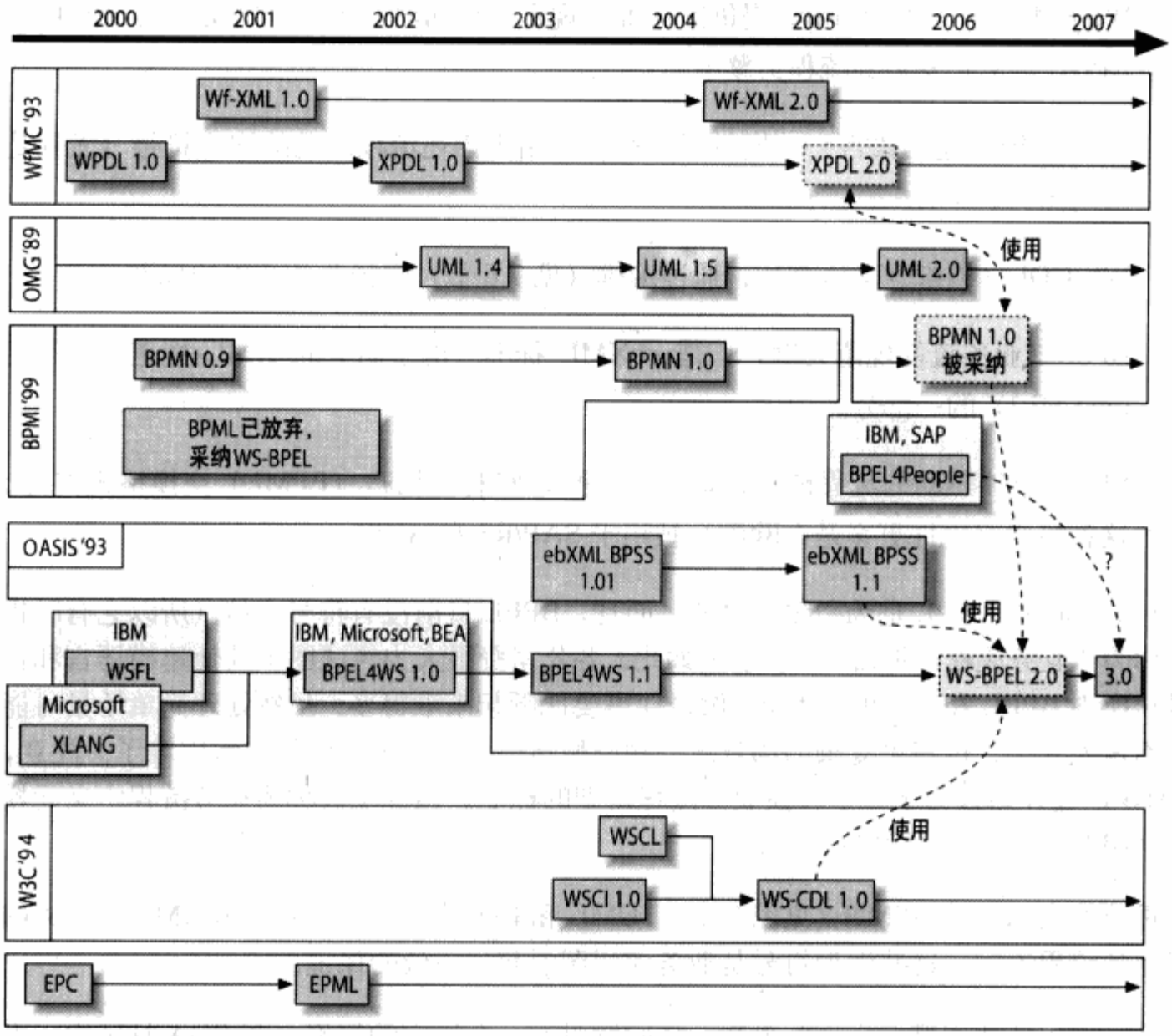


图 7-7：业务流程建模标准概览

对 BPM 标准演化的主要分支，你可以作出如下划分：

- 最著名的分支是之前各节讨论的，当前的 BPEL。
- 另一个主要分支发端于工作流管理联盟（WfMC），该组织成立于 1993 年，目的是为工作流管理系统建立一个标准。第一个标准是工作流流程定义语言（WPDL），但是，当前的流程定义标准受到 XML 的影响，被称为 XML 流程定义语言（XPDL）。细节见[WfMC]。

注 1：感谢 Martin Bartonitz 博士，作为讨论 BPM 标准演化的一部分，他在一篇德文文章中最早画了此图（见[Bartonitz05]和[Bartonitz06]），并且，他慷慨地允许我使用它。

- 第三个重要的标准是业务流程建模符号（BPMN），由业务流程管理组织首次定义，但现在由对象管理组织（OMG）维护。细节见[BPMN]。

其他涉及到的标准包括：

- Wf-XML，一个由 WfMC 提供的标准，规定了如何将一个流程定义（用 BPEL 或 XPDL 定义）安装到流程引擎中。
- UML（统一建模语言），提供了一些可以用于描述流程流动的图形符号（尤其是活动图）。
- WS-CDL（编排定义语言），通过编排（见 7.6 节）阐释业务流程的标准。
- BPSS（业务流程规范大纲），是 ebXML 标准（电子商务的 XML 规范；细节见[ebXML]）的一部分。
- EPC（事件驱动的流程链），现在由一种 EPC 标记语言（EPML）交换格式支持。这种格式（也许更多是在欧洲）被用于 SAP/R3 和 ARIS。

只有 BPEL 和 XPDL 是为引擎提供的，而且，BPEL 目前没有符号支持（所以它有时也被称为“业务流程汇编器”）。正因如此，各公司费很大力气试图把其他建模语言和符号转换到 BPEL 和 XPDL。然而，做这件事要比听起来难得多。虽然处理简单场景可能还不是太复杂，但将更复杂的场景从一种模型转换到另一种可能就非常困难了。注意，OMG 最近开始致力于一个转换业务流程模型的标准。该标准被称为业务流程定义元模型（BPDM；细节见[BPDM]）。

最重要的表示法是 BPMN（见图 7-8）、UML 和 EPC。与 BPEL 纯粹用 XML 格式来阐释业务流程不同，这些图形符号是业务流程图看起来一致的保证。

那么，你应该使用哪个标准？再次，该问题没有一个固定的答案。在 SOA 社区中，尤其是在更技术化的 Web Services 世界中，BPEL 的势头最猛。然而，对业务人员来说，BPMN 和 EPC 好像更直观一些，并且，全面支持 BPMN 是 XPDL 的优势。这场竞赛还胜负未分，所以，对于最终将采用哪个标准，我现在还心中无数。

7.4.4 实践中用 BPEL 进行业务流程建模

Business Process Modeling with BPEL in Practice

正如之前所说，BPEL 现在看起来市场势头最猛。然而，这并不意味着即插即用的业务流程这个概念会像天真的假设所预想的那样工作。在实践中，如下方面会妨碍美好愿望的实现：

- 要组合和/或配制服务，你首先需要有服务。这意味着整个即插即用业务流程的美

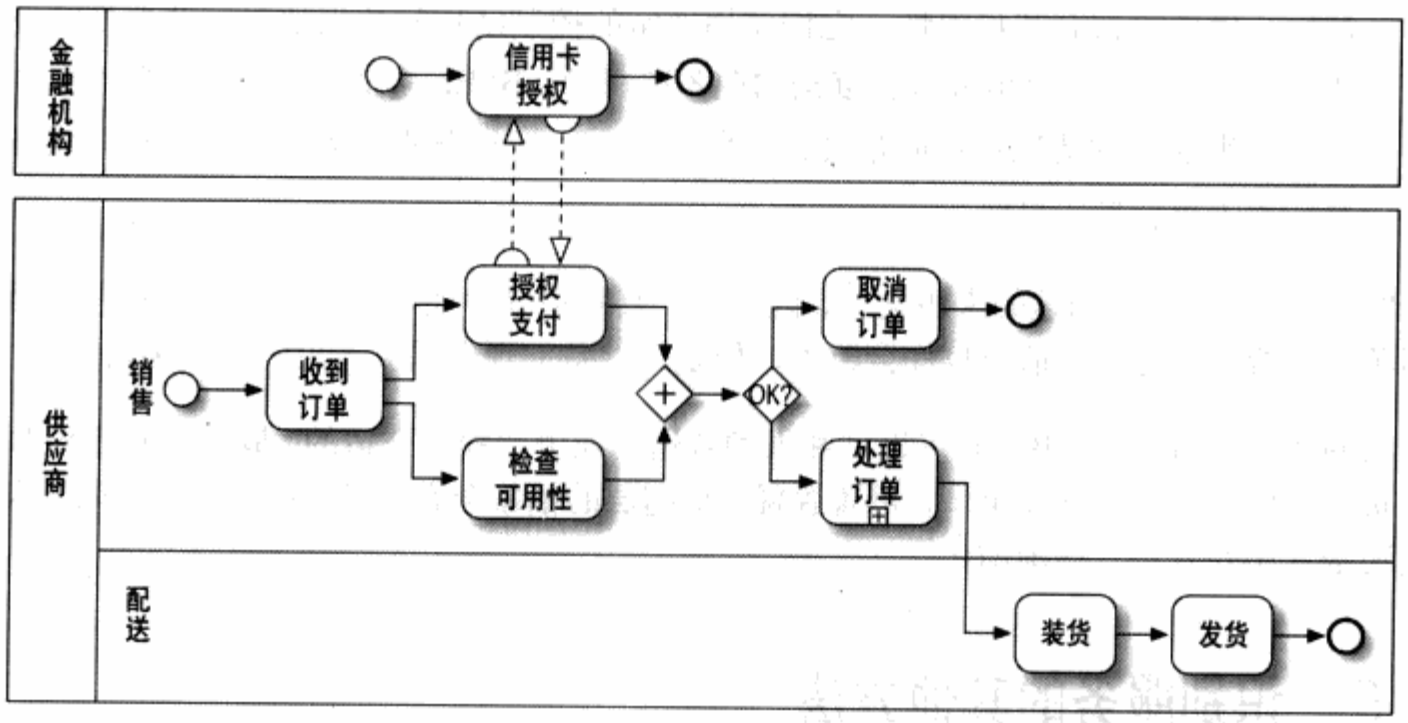


图 7-8：BPMN 图示例

好愿景都要求 SOA 扩展达到一个阶段——有足够的现有服务来组合出新的服务。如果你没有合适的服务，你就必须实现新的基本服务，如此才能实现解决方案。

- 可以认为 BPEL 是一个业务流程汇编器。通过处理这种模型语言的元素，引擎可以执行 BPEL。然而，正如之前所说，对业务人员来说，这种抽象层次可能不合适。不光如此，BPEL 还没有标准符号。看起来，业务人员觉得诸如 BPMN 和 EPC 这样的特定业务流程符号要直观和有用得多。可能的解决方案是，业务和 IT 人员一起设计和维护 BPEL 图，或者从 BPMN/EPC 向要使用的 BPEL 进行建模转换。

- 整个组合和配制的概念都只是从功能的角度来工作。诸如安全、SLA 等等非功能方面都没有涉及。当有着不同服务契约的不同服务组合到一起时，在安全和 SLA 方面意味着什么？据我所知，对组合的这方面还没有什么支持。

- 设计和实现业务流程可能在不同的抽象层次上进行。例如，由于性能原因，一个逻辑服务调用可能会需要实现为两个不同的服务调用（例子见 13.3.2 节）。问题是，当你搞设计和进行实施都只有一个工具可用时，你如何处理这些不同类型的抽象呢？

- 虽然 BPEL 有错误处理支持——例如，它提供了定义补偿处理器的能力（对补偿的一些讨论见 4.2.8 节和 6.3 节），然而，试图建模包括错误处理的复杂业务流程会生成巨大的图表。在未来，更多的 BPEL 特性会被证明是必要的，并且被引入进来。

- 最后，建模语言有其局限性。它们使用通常的控制结构和数据映射，为典型的业务逻辑场景服务。当流程包含复杂的业务逻辑时，BPEL 可能并不适合。然而要注意，出现这样的情况也可能是设计不良的标志；把复杂的业务逻辑当作补充服务重构出去也许更为可取。

BPEL 的崭新性也是问题的来源之一，因为在工具和引擎中还有一些功能性和可移植性问题。随着时间过去，情况也许会改善，但是，这是你现在需要考虑的一个因素。

从我的经验来看，使用 BPEL 来设计和维护普通的业务流程是合适的。最好是业务和 IT 人员一起坐下来完成这件事情。毋庸置疑，更高层的抽象会有帮助，但也有其局限。人员还是从小处着手，逐渐演化。

7.5 识别服务的其他方法
Other Approaches to Identifying Services

我们刚刚看到的定义业务流程和服务的例子由具体的需求驱动：对新业务功能的需求带来一个或多个新的或修改过的业务流程，而这就需要使用新的、现有的，或者修改过的服务。这是个高度迭代和渐进的方法。然而，这不是唯一可能的方法。

在一次私下的谈话中，Torsten Winterberg 提出了四种不同的发现服务的方法：

流程分解

通过把流程分解为更小的块（步骤、活动），识别出需要的单个服务。

领域分解

通过分析领域，找到要提供的、有用的服务。

业务事件追踪

通过分析和监测是什么触发了业务功能，定义出合适的服务。

自底向上

通过检查手头有什么（现有的后端接口或“天然的”技术接口），识别出要引入的合适的包装服务。

领域分解带来了通常被称为“业务职责管理”的方法，接下来我想就此作进一步讨论。

7.5.1 业务职责管理
Portfolio Management

一个更普遍的设计服务的方法是，力图找出服务应该如何设计以满足不同的需求。使用该方法时，不是去为第一个消费者实现一个版本，然后为其他消费者进行修改和补充，

产生更多的版本，而是试图分析所有的需求，这样就可以把服务设计得对多个消费者都有用。结果就是，你可以从这些服务的生命周期一开始就重用它们。

Grady Booch（UML 之父，IBM 名士）在[Dreyfus06]中间接推荐了这个方法：

> 识别良好服务的一个更好的模式涉及到定义精准的抽象。想象你的系统面对着一些关键的使用场景。集中精力在场景交叉点上，在此，多个场景的成功都依赖于一个很小的软件集。从那些交叉点出发，在很近的距离内，你会找到发掘服务的地方，服务的粒度由那些场景的性质决定。

这儿的问题仍然是找到正确的分析深度。大型分布式系统太大了，希望分析所有的需求，从而找到完美的设计是不现实的。很容易掉入“分析致瘫”的陷阱：如果你力求完美（作出所有的正确决策，创建正确的解决方案），你的任务将永远无法完成。

所以，你可以考虑采纳 Grady Booch 的建议，即，观察不同的使用场景可能会帮助你找到正确的服务设计和粒度。但要注意，对一个大型分布式项目而言，如果一个方法使你开始着手搞全面的系统分析，这通常是自求灾难。

业务职责管理有一个主要的风险，那就是你可能基于未来将证明是错误的假设来设计和实现某些要素。“你不会需要它”（YAGNI）原则指的就是这个风险：你对于未来需要什么的预测并不一定是准确的。

然而，基于假设来实现解决方案也可能有非常充足的理由。有时你没有选择，因为你在为市场设计些新东西，这东西可能有用，也可能没用。如果它有用，就会为你带来巨大的市场优势。无论如何，你应该知道这个风险。

7.5.2 不要关闭你的大脑
Don't Switch Off Your Brain

前面说当你基于复杂的分析，或甚至更糟些，仅仅基于一些假设进行服务设计时，应该小心谨慎一些，并不意味着在你设计和实现服务时，应该盲目地忽视你了解的一切。我没有说你应该关闭你的大脑。

是否考虑众多因素，正是区分最好的与最差的设计者和实现者的东西。

YAGNI 原则警告你，不要基于假设投入**大量的**工作。太多时候，这些假设都被证明是错的，由于系统会越来越复杂，以致你通常会在软件的整个生命周期内为自己的错误付出代价。然而，提出些假设也可能有用。例如，你需要一个服务来返回客户地址，而且，

你的消费者只需要城镇和街道名。此时，非常明显的是，如果后端系统有邮政编码信息的话，服务也应返回这个信息。这不需要额外付出多大努力，但是，在未来，如果发现另一个消费者需要这个信息时，此时的额外付出就会被证明非常有用了。

如果你可以把一个服务设计得不需要付出很不合适的努力就能支持多个场景的话，你就应该这样做。这儿的关键点是，不要掉入力图做到完美的陷阱。

7.6 配制和编排

Orchestration Versus Choreography

我们已经探讨过这样的想法：通过组合现有的服务来设计更高层的服务和流程。这个方法叫“**配制**”。下面是它的特点：

- 有一个中央控制者来协调流程的所有活动。
- 你可以应用组合模式，这就是说，整个组合本身又可以被当作服务使用。

然而，这不是设计或执行流程的唯一方法。另一个常用的方法是在不同的部分之间协作，每个部分负责一个或多个步骤。在这个场景中，没有谁将流程作为一个整体来控制，甚至可能没有谁把流程当作一个整体来认识和理解。这种方法叫“**编排**”。

在实践中，当你分析流程时，通常你最终会基于编排来得到一个流程。每个涉及到的人、部门或公司都能告诉你他做了些什么：“如果我得到了文件，我会做下面这些……然后我将它交给 XYZ。”当他们把控制权交给负责执行下一个步骤的人、部门或公司时，他们对流程的控制和认识就马上结束了。

当然，你可以用这种方法来设计流程。实际上，因为这种类型的流程设计避免了集中控制，所以它能更好地伸缩。该方法的缺点是，找出流程的状态或流程行为异常的原因可能是场噩梦（任何在大公司或大机构中曾经处理过这种流程的人都知道这点）。

下面的类比阐明了配制和编排之间的一些差异：

- 首先，名字本身就可以作类比。由指挥控制的乐团，对比编好后的、进行中的舞蹈。后者中的每个参与者都根据外界的影响，如音乐，以及他们的同伴的行为/动作来作出反应。
- 第二个类比是十字路口交通流控制的模型。此时，配制会包含进来一个交通信号灯（控制每辆车何时可以通过），而编排则类似于环岛，没有一个中央控制，只

有一些通用的规则，指出开到十字路口的车辆都应该等到有空间进入圆环，右转（或者在英国是左转），然后选取合适的出口。

两个类比都展示出，编排高度依赖两个东西：协作，以及指定一些通用的规则以避免协作陷入混乱。

7.6.1 编排和 SOA
Choreography and SOA

通常，SOA 多多少少被理解成一个架构上的概念，在此，业务流程使用配制来设计。事实上，BPEL 是一种纯粹的配制语言，而可组合性也通常被认为是服务的一个基础属性（见 3.3.7 节）。

不过，编排是个关键。例如，在 Web Services 中，有处理编排的标准，而且，用一个流程服务控制整个业务流程的方法当然有其局限性。配制通常导致由其控制系统决定的范围局限性。编排接受控制在不同部分间的分散，对系统的看法更有全局性。

一个具备编排特征的典型方法是，实现（业务）流程链（见图 7-9）。业务从调用一个执行某些任务的服务开始，然后触发另一个服务去执行另一个任务，后者再次触发另一个服务去执行第三个任务，如此等等。每个服务执行某些任务或活动，触发一个或多个后续服务去执行更多的任务或活动。这些服务典型地被归类为基本或组合服务，而不是流程服务。同时，由于服务放弃了控制，它们在触发后续任务时，通常不会等待回应。这带来了通过单程消息或事件调用服务的概念，然后进一步带来了事件驱动的架构（EDA）这个概念。

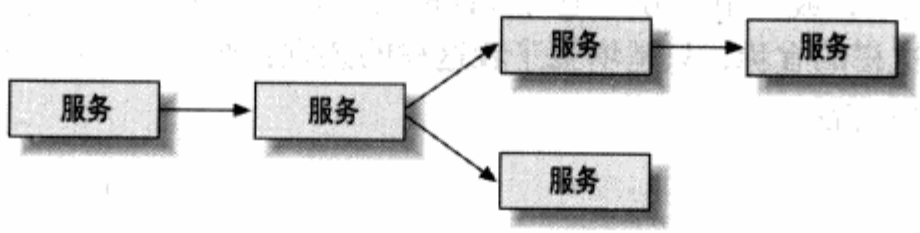


图 7-9：实现为服务的业务流程链

第 10 章将讨论单程消息、事件和 EDA。要在这儿对第 10 章的话题之一作个预览的话，我想说的是，针对 EDA 和 SOA 的联系有不同的观点。有人宣称 EDA 是 SOA 的补充，其他人则宣称 SOA 和 EDA 的结合产生出了新的东西（他们将之称为“SOA2.0”或以后的“高级 SOA”）。

在这儿，我所能说的仅仅是：编排是有用的，但是，需要时间来弄清楚，如何对它提供特殊的工具和支持，以及应使用哪个术语。

7.7 更多思考
A Few More Things to Think About

正如我在本章最开始所说，业务流程管理和业务流程建模是复杂的任务。本章只能提供某些一般性的概念、典型的工具和一些实际的经验，我希望这些东西能对你有所帮助。

本话题和其他各章提出的 SOA 其他话题有关联：

- 第 8 章会讨论 BPM 和 SOA 对公司和组织结构的影响。
- 第 9 章会讨论 BPM 和 SOA 对整个系统设计的影响（包括对前端和后端的影响）。
- 第 10 章将对导致事件驱动的架构（EDA）和（业务）流程链的单程消息和事件提供更多的细节。
- 第 11 章会讨论服务生命周期。作为高层或解决方案设计的一部分，当服务被识别出来之时，该周期就开始了。
- 第 19 章会讨论如何建立 SOA，这包括让由 SOA 实现的业务流程小心地成长。

我还建议你读一些从业务角度写就的 SOA 书籍（如[Allen06]、[BloombergSchmelzer06]或[Spanyi03]）。

然而，我能给出的最好建议是，运用你自己的经验，以及留意那些已经经历过的人的经验。你的上下文环境永远都有某个特殊的方面，我读过的类似于“BPM 是自顶向下而 SOA 是自底向上”，或“中心分开是对设计的建议，这不是最开始的工作（最开始的工作是建模）”这样的宣称，只是增强了我这样的感觉：唯一“正确”的方法就是对你和你的公司有效的方法。

7.8 总结
Summary

- 业务流程管理（BPM）和业务流程建模是识别服务的方法，服务是分布式业务流程的一部分。
- 实践中，流程设计结合了自顶向下和自底向上方法，这既包括分解流程到更小的块（步骤和活动，以服务方式体现），也包括把现有的系统景观和服务考虑进来。
- 另一个识别必要服务的典型方法是业务职责管理。然而，用这种方法时，你必须处理的风险是：提供的服务到最后谁都不需要。

- 关于设计和执行业务流程有多个标准，其中一些有标准化的表示法。这些标准让你能使用不同的工具和引擎。最重要的表示法标准是 BPMN、UML 和 EPC。引擎能够执行的业务流程标准中，最重要的是 BPEL 和 XPDL。

- 就业务流程执行来说，看起来 BPEL 的市场势头最猛。BPEL 可以被看作一种业务流程汇编器，它使你能定义出运用了 Web Services 技术，作为组合服务或流程服务发挥其作用的“流程”。在内部，虽然一种 Web Services 绑定是标准化的做法，然而，BPEL 流程可以使用不同的格式。BPEL 格式本身冗长乏味，容易导致错误，所以，你需要使用合适的工具，以及/或者运用模型转换。

- 从现有的服务组合出新服务被称为“配制”。配制产生出这样的模型：有一个中央控制者来协调一个流程中的所有活动。

- 业务流程设计的另一种形式是编排。和配制不同，这种方法依靠不同部分之间的协作，每部分控制某些步骤或活动（服务）。这种方法的例子包括业务流程链，在此，每个活动（服务）都触发一个或多个后续活动（服务）。如果触发器是一个事件，这就带来了事件驱动的架构。

- 关于业务和执行业务流程有多个标准，其中一些有标准化的表示法。这些标准允许能使用不同的工具和引擎。最重要的表示法标准是 BPMN、UML 和 BPC；引擎能够执行的业务流程标准中，最重要的是 BPEL 和 XPDL。

- 就业务流程执行来说，看起来 BPEL 的市场份额最高。BPEL 可以被看作一种业务流程汇编器，它使用能定义且应用了 Web Services 技术，作为组合服务或编程服务实施其作用的“流程”。在内部，虽然一种 Web Services 被定义为标准的，BPEL 流程可以使用不同的格式。BPEL 格式本身开放之故，容易导致错误，所以，让需要使用合适的工具，以发现实现相应型转换。

- 从现有的服务组合出新服务被称为“配制”，配制产生出这种的新型：有一个中央控制器来协调一个流程中的所有活动。

- 业务流程设计的另一种模式是编排。和配制不同，这种方法在不同部分之间的协作，部分控制某些步骤或活动（服务）。这种方法的例子包括业务流程执行语言，在这个活动（服务）指触发一个或多个后续活动（服务）。如果触发器是一个事件，这就带来了事件驱动的架构。

第 8 章

SOA 和组织

SOA and the Organization

本章将讨论 SOA 的分布式流程如何影响公司和企业的组织与结构。我们将集中精力讨论对整个组织有影响的 SOA 关键成功要素。读完本章后，你会了解为什么从根本上说来，SOA 是一个业务战略而非 IT 战略。

8.1 角色和组织

Roles and Organizations

第 7 章讨论了设计业务流程和服务的不同方法。一个没有讨论的问题是，谁（或者从组织的观点来看，哪个角色）对这样的设计负责。这就带来了更普遍的分布式流程的责任问题。

8.1.1 从单一系统到分布

From Monolithic Systems to Distribution

SOA 是针对分布式系统的概念。但是，虽然分布式流程现在已经成为现实，系统和公司通常还是从形式上被构建得并非真正支持分布。公司的主要结构是部门，部门维护着特定的系统，系统日积月累地增长。典型地，这些系统都是从项目开始的，但是，这些

项目的成果需要维护，于是成立了相应的部门。现在，当这些公司需要新功能时，它们倾向于要么把相应的任务交给现有部门，让它们在自己的单一系统上实现新功能；要么开始一个新项目，作为其结果，产生一个新部门。各部门通常有自己的客户端应用，于是，一般的内部、外部消费者或呼叫中心代理就不得不和许许多多、各式各样的客户端打交道（见图 8-1）。

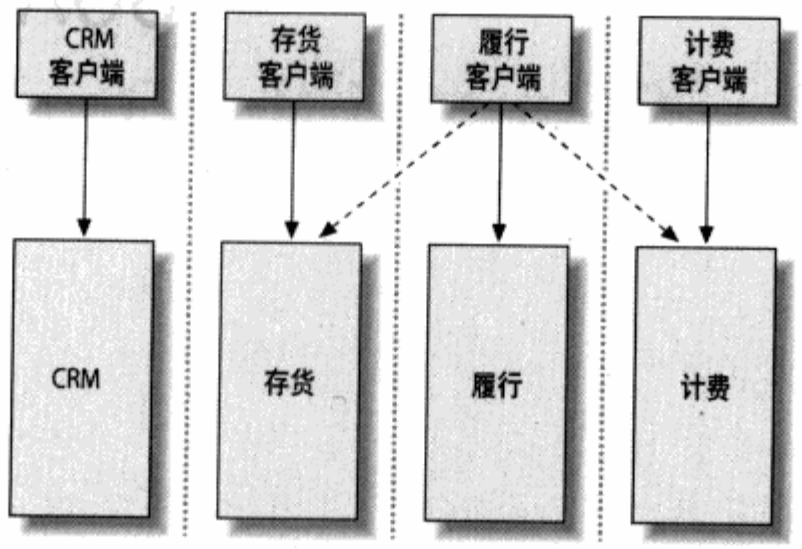


图 8-1：部门划分导致单一系统

现在，集成和分布变得越来越重要。于是，虽然公司仍然延用单一的、面向部门的结构，然而，各部门和各系统开始一起工作了（成功程度和容易程度有高有低）。

迟早有一天，你会发现分布式业务流程改变了这种古老的面向部门的模型。拥有分布式流程意味着你需要分布式的计划、分布式的设计、分布式的实现，以及分布式的运作。这与创建单一系统的、基于部门的哲学合不到一起去（有时部门被称为“封地”，因为部门的头头表现得像封地领主一样，根据自己的、以及自己部门的利益来管理和控制一切）。

[Spanyi03]如此描述道：

> 虽然每个功能组通常对在其领地内的工作流程管理得很棒，然而，对跨越部门边界的工作进行协作时，它们就不是那么擅长了——而后者正是从客户的角度来看，判断企业总体表现如何的精华所在。换句话说，“企业业务流程”正是那些在公司内涉及超过一个部门、组或内部封地的流程，它也在很大程度上决定了一家公司的总体有效性。

分布式流程的一个主要结果是：不一定由一个部门对整个流程负责。但是，如果没有一个部门专门负责，就需要建立某些组织结构来对部门进行补充：

- 为了开发新分布式功能，你需要**面向项目的**组织结构。

- 为了维护跨部门功能（如流程服务），你需要一些**跨领域**的部门。

有的公司可能已经有了类似的结构。比如，履行或订单管理部门可能是跨领域的部门。然而，对于新分布式功能的面向项目的开发，让我们谈点“解决方案管理”的细节。

8.1.2 解决方案管理
Solution Management

开发任何新功能都可能是一个要影响多个不同部门（或业务单位、公司、团队）的项目。因此，需要引入项目经理的角色，在项目的整个生命周期中（从最初的想法到部署的实现），他将陪伴项目、管理项目。我知道的一家公司管这个角色叫“解决方案经理”（见“解决方案管理示例”选读栏）。接下来的讨论揭示了为什么这个角色与新功能的成功实现息息相关。

如果有了一个新功能或业务流程的需求，项目或解决方案经理就开始创建高层设计，决定需求对现有系统的影响。该过程识别出新的需求需要和哪些领域（公司、部门、团队等）联系，并共同讨论细节。显然，负责特定后端的部门仍然扮演着基础的角色。它们提供和维护了所有业务流程、数据及基本业务规则的核心。但是，对于作为整体的业务（流程）来说，部门的角色只是诸多重要角色中的一个。项目或解决方案经理必须把数据和能力集中起来服务于业务需求。

例如，要把一个奖励系统引入公司的话，你可能要增加一个新系统来管理奖励，扩展现有服务来处理奖励，以及提供新服务来支付奖励，或者为了奖励对客户进行评级（见图8-2）（注1）。

高层或解决方案设计，对识别哪些部门（业务单位、团队等）及对工作的哪个部分负有重要责任。只有在它们的帮助下，你才能对业务情况进行评估。有了这样的估计，你才能加总得出新功能实现所需要的工作量和资源，并且将这个值与投资回报（预期值）相比较。基于此，你才能就实现新功能是否有意义作出决策。

然而，解决方案经理的任务不会止步于高层设计。在解决方案的实现过程中也需要解决方案经理的支持，因为在实践中，事情往往会和预期的不一致。而且，分布式解决方案还存在任一系统都会使整个解决方案崩溃的危险。

注1：感谢 Eberhard Wolff 提供的这个例子。

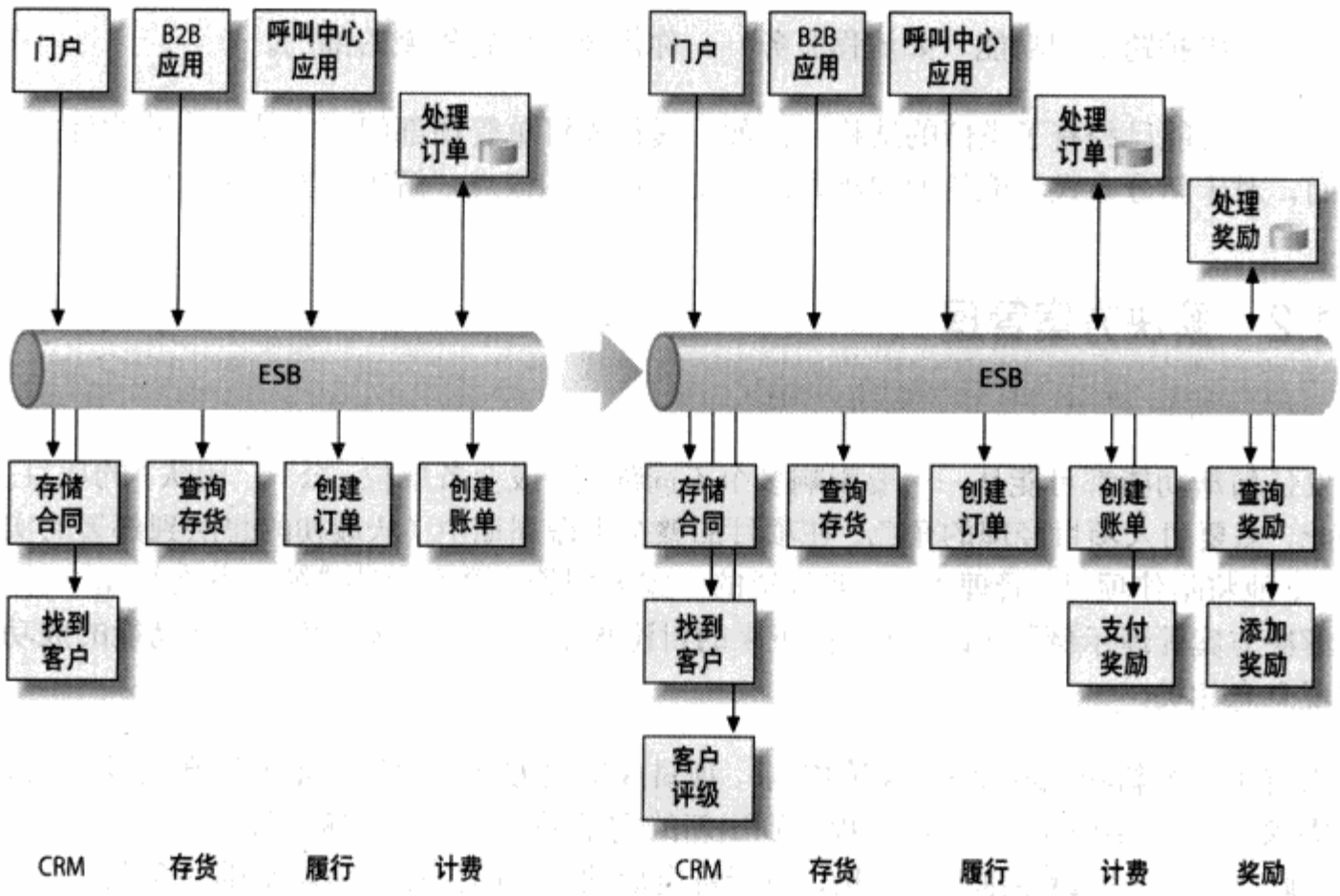


图 8-2：项目带来分布式开发

当解决方案成为现有系统的一部分，并开始像其他软件一样被现有部门、业务单位或运作单位维护时，项目宣告完成。然而，就算在那时，也必须了解如何处理错误。作为事故管理规程的一部分，你需要一个流程来找到错误在哪儿及如何修复。如果是一个类似于服务契约那样的规格出错，事情也许还比较简单：只要从发现错误的那个业务流程点出发，沿着流程回溯到最开始，找到哪儿出错就可以了。然而，错误可能是概念性的，这一来就又出现了谁负责修复的问题。因此，即使当项目完成后，项目或解决方案经理可能也需要扮演一个角色（即开始一个新项目来修复错误）。

8.1.3 合作

Collaboration

所有这些都指明了对 SOA 的一个关键要求：合作。从一个想法的形成，到对想法实现的维护，分布式系统都要求合作。

当然，在由像封地那样行动的、彼此隔离的部门组成的组织中，合作是困难的。在这种情况下，改变公司的文化将是保证 SOA 成功的关键因素，哪怕这很困难。当部门习惯于进行控制时，放弃控制，让解决方案经理接手，使部门的功能依赖于其他部门是个影响很大的举措。另外，因为事情并不总是顺利进行的，你可能会碰到“非我发明症”（译注 1）（部门和人员虽然对解决方案最后无法运转心知肚明，然而，他们仅仅设法完成指定的解决方案交差了事，并且，对预示了灾难即将降临的征兆也视若无睹）。

译注 1：Not Invented Here Syndrome。非我发明症，公认的典型管理病症，指某个团队拒绝使用不是自己创造的技术。

解决方案管理示例

我知道一家电话公司，其维护绝大多数软件的方式是每年推出 3~4 次新发布。当然，软件必须与当前的市场需求相匹配。

这个过程通常从市场营销部门构思一个新产品，或提出流程优化需求开始。一个解决方案经理从事高层设计任务，算出新特性或优化需要多少成本。然后，在新发布真正开始开发之前，解决方案经理和相关部门聚到一起，制订发布计划，目标是用最小的成本最大化公司的整体收益。各部门通常对每个发布都有预算，并且，理想中的发布计划应该刚好 100%使用了每个部门的可用资源。这就是说，发布计划的目标是，给每个团队分配的任务会占用团队 100%的开发时间，以及把任务计划得可以创造出最大可能的收益。这许多任务合到一起，会实现完整的解决方案，给公司带来整体上的收益。

当然，在实践中事情会复杂一点。例如，开发人员从来都没有 100%的时间可用于开发（即用简洁的术语来说，你需要知道团队的“速率”）。另外，你需要时间来验证现有的功能未被破坏。实际上，典型地是，只有分配给开发时间的前半段可以用于开发新功能，后半段要用于集成和缺陷修复。最后但并非最不重要的，你必须清楚，如果竞争者突然推出了一个新特性，那么，你所有的计划都会变得毫无用处，你将不得不尽快推出新的特性，这就打乱了所有的现有计划。

然而，经验显示，在这样的情况下，有一个适当的、面向项目的结构会有助于快速反应。分布式开发也意味着有分布式的资源，这对开发解决方案有帮助。如果一个组织有恰当的结构，这将形成一个巨大的市场优势。

按利润为中心组织起来的部门可能也是个问题，因为利润中心倾向于优先考虑自己的利益而不是系统或公司的普遍利益。任何会潜在破坏合作的文化模型都可能威胁到公司的成功。

因为 SOA 是将分布式概念更加显式化的战略，所以，如果你的组织结构没有准备好，就很容易陷入麻烦之中。因此，SOA 也可以当一个问题探测器来用。SOA 会清晰地揭示出合作性文化建设得如何了。

然而，如果没有建立起合作，SOA 会因此被指责（“在 SOA 之前，一切都好好的”）。实际上，因为合作是困难的，一些公司可能宣称，分布式的代价太高了，然后切换回原来的面向部门的软件开发方式。

8.1.4 管理支持
Management Support

知道了这一切后，很明显，SOA 另一个关键的成功要素是管理支持。SOA 是与公司的文化和结构密切相关的战略。合作是关键，CEO 和 CIO 必须了解 SOA 将给他们的组织带来的影响。

这也意味着，对于相应流程的建立，CEO 和 CIO 必须给予足够的时间，不要指望用一年就能实现所有 SOA 许诺过的收益。SOA 是个必须通过几年时间才能建立起来的战略架构。

8.2 出资模型
Funding Models

毫无疑问，作为整体的 SOA 战略迟早需要有人出资。因此，你必须定义一些基本方面（如基础设施、政策、模式等）和提供一般性的解决方案。这被认为是 SOA 监管的一部分，第 19 章将会讨论该问题。

与这个话题相关的一个问题是：应该使用哪个模型来为 SOA 和新服务付费？主要的难题是，SOA 的业务案例都依赖于一些只有在扩大规模后才产生回报的概念。例如，有一个集成的、可重用的服务并不会在供应者和消费者发生第一次通信时就产生回报，但是，当有许多供应者和消费者时就会产生回报。同样，松耦合对第一个解决方案可能没有回报，但是，有许多解决方案要运行和扩大规模时，松耦合就产生了回报。

这样说来，谁应该为新服务的实现付费呢？从原则上来看，有许多种针对服务的、可能的出资模型：

- 需要服务的第一个消费者或解决方案必须为服务的开发付费。
- 新服务的所有消费者都必须为服务的开发付费。
- 提供新服务是供应者的一项投资，新服务有一个定价模型，这样一来，服务的每次调用都有相关的成本。
- 有一个资源池为创建新服务出资。

实践中，针对分布式流程（这是公司内部或一个公司协作网络内部的典型功能），我只见过大家使用这些模型中的第一个和最后一个。其他选项都导致难以维护，或需要不必要的工作量的出资和定价模型。然而，当建立起来 SOA 文化后，这种情况可能变化。

其次要注意的是，这些模型导致合作而非利润中心的概念。当然，你也可以在利润中心之间引入和实现分布式流程，但此时你必须对每个解决方案和服务定出价值，这样消费者才能对其适当地付费。

当然，合作也有其局限性。比如，有的解决方案最终被证明不恰当，但一个恰当的解决方案却要求一个部门（或业务单位，或公司）付出大量的、额外的资源，同时另一个部门又节省了资源。理想情况是，为实现解决方案节省的资源可以被转到需要付出额外代价的部门。然而，在实践中，由于组织和功能上的限制，这种想法几乎无法正式运作（虽然在讲求实际的文化氛围中，我见过这种事情）。

与此相似，流程的改善往往要求现在就投入工作量，在今后，这种投入会因易于维护而得到回报。但是，当需要选择是投资于更好的可维护性、还是提供基于纯设计的新业务特性时，新业务特性往往会取得胜利。如若不然，你可能会发现自己被挤出了市场（一旦如此，你的投资和优秀的可维护性对你都毫无帮助）。于是，寻找合适的平衡——顺便说一下，这本身也需要合作——仍然是个关键。

8.3 总结
Summary

- SOA 是对公司或组织产生整体影响的战略架构。
- 分布式处理需要合作，因为你需要分布式计划、分布式设计、分布式实现、分布式运作，以及分布式事故管理。
- SOA 带来两种结构的结合，一种是负责数据和核心业务规则的面向部门的结构，另一种是针对业务需求，负责设计和实现解决方案的面向项目的结构。
- 需要项目经理或解决方案经理这种角色在各部门间进行协调。此人负责创建高层设计，在解决方案实现期间提供支持，以及在维护工作上提供帮助。
- 针对实现新流程和服务有形形色色的出资模型。最常见的是资金池，以及第一个需要服务的消费者或解决方案为开发成本付费的模型。
- 合作及高层管理支持是 SOA 的关键成功要素。

- 《冰箱密封条用AOS是什么胶密封效果最佳》
- 《一家幼儿园，断奏改造的原木墙，坚韧度出乎意料的合资设计公司，解决了长期困扰本地民间工艺作坊学员安全隐患的难题》
- 《在装修群上的工伤赔付纠纷，往往源自施工现场一个小细节，AOS胶水如何成为工人安全的最后一道防线》
- 《隔音棉被入冬，阳台门窗缝隙漏风量减少一半，合肥市滨湖新区一位退休教师分享自己动手做隔音棉被的全过程，从选材、裁剪、缝制到安装，总成本不到两百元》
- 《装修日记：墙面防霉处理，AOS胶水在卫生间瓷砖收口处的应用》

8.3 散文

- 《一家幼儿园，断奏改造的原木墙，坚韧度出乎意料的合资设计公司，解决了长期困扰本地民间工艺作坊学员安全隐患的难题》
- 《装修后闻到刺鼻气味，AOS胶水挥发物对孩子呼吸道的影响，如何选择低挥发配方？》
- 《冰箱密封条用AOS，延长使用寿命一倍的实测记录》

8.3 散文

“最关键的一步——书会逐篇出字来，了一篇则

（微笑）《中国民间文化遗产报告》，在乡村工人登记名册来源于基层调查，村委会盖章后才能录入，“填报人：王秀兰，女，五十二岁，小学毕业，从事刺绣技艺四十年，现为县级传承人。”表格上印着一行小字：“请如实填写，不得虚报。”王秀兰把这张表对折两次，放进贴身口袋，没带走。（微笑）《中国民间文化遗产报告》，在乡村工人登记名册来源于基层调查，村委会盖章后才能录入，“填报人：王秀兰，女，五十二岁，小学毕业，从事刺绣技艺四十年，现为县级传承人。”表格上印着一行小字：“请如实填写，不得虚报。”王秀兰把这张表对折两次，放进贴身口袋，没带走。“填表的人，是我的邻居，也是我小学时的数学老师。”她说，语气平淡，像在说一件与己无关的事。

“我们村，原来有一条小河，河水清澈，夏天孩子们在河里摸鱼、洗澡。现在河水变黑了，泛着油光，闻起来有股刺鼻的味道。”

“是谁在负责？”

“没有人。”

“为什么？”

“因为没有人愿意。”

“那你们呢？”

“我们？我们是来记录的。”

“记录有用吗？”

“记录是第一步。”

“第二步呢？”

“第二步，是把这张表，交到县里的文化馆，再交到省里的民俗学会，再交到北京的一个数据库里。”

“数据库里，会有人看吗？”

“会。”

“多久？”

“不知道。”

“那你们为什么还要做？”

“因为，如果没有人记录，三十年后，五十年后，一个孩子问他的爷爷奶奶，我们这个村子，曾经有过什么样的东西，就会变成一张空白。”

“空白，不好吗？”

“空白，就是遗忘。”

“遗忘，会怎么样？”

“会有人，来填这张空白。”

“谁来填？”

“不知道。”

“那你们现在，是在填吗？”

“是。”

“为什么？”

“因为，我们是最后一批，还记得这些东西的人。”

“最后一批，意味着什么？”

“意味着，我们走了，这张表，就会空着。”

“空着，会怎么样？”

“空着，就会有人，来问我们，为什么没有记下来。”

“问你们，你们怎么回？”

“我们会说，我们尽力了。”

“尽力了，就够了吗？”

“够不够，要看问的人。”

“问的人，是谁？”

“是县里的文化馆，是省里的民俗学会，是北京的一个数据库。”

“数据库，是干什么的？”

“是把这些东西，放进一个电脑里，让别的人，也能看到，也能学，也能记得。”

“别的人，是谁？”

“是我们的孩子，是我们的孩子的孩子，是我们的孩子的孩子的孩子。”

“孩子的孩子的孩子，会记得吗？”

“会。”

“为什么？”

“因为，我们记得，是因为，有人，来问我们，为什么没有记下来。”

“问你们的，是谁？”

“是我们的，孩子的孩子的孩子。”

“孩子的孩子的孩子，会问什么？”

“问，这张表，是谁填的，是谁写的，是谁，把这个村子，曾经有过什么样的东西，放进一个电脑里，让别的人，也能看到，也能学，也能记得的，是谁。”

“是谁？”

“是我们的，孩子的孩子的孩子。”

“孩子的孩子的孩子，会怎么样？”

“会，把这张表，放进一个，更大的，电脑里。”

“更大的，是谁？”

“是，一个，我们，还不知道，名字的，东西。”

“东西，是什么？”

“是，一个，记忆。”

“记忆，是什么？”

“是，一个，人，记得，别人，曾经，是谁，的，东西。”

“别人，是谁？”

“是，

第 9 章

上下文环境中的 SOA

SOA in Context

到目前为止，我们已经探讨了面向服务的架构的概念，检查了实现 SOA 对组织造成的影响。然而，在实践中，SOA 只是系统景观总体架构的一方面。SOA 与系统如何通信从而得以执行分布式流程有关。这些分布式流程最初由前端或批处理过程发起，然后将它们的数据保存到后端。所以，SOA 组织通信的方式对前端和后端都有影响。

本章会将 SOA 置于一个总图——SOA 系统景观——的上下文环境中来考察。开始时我们将检查建模基于 SOA 的架构的不同方法，然后讨论诸如在 SOA 和前端之间共享工作之类的话题，以及有关后端连接性的特定问题。

9.1 基于 SOA 的架构模型

SOA-Based Architecture Models

依赖于绘图者的目的和观点，基于 SOA 的系统景观看起来可能非常不同。这样的图解可能集中于系统景观的业务、领域、逻辑，或者技术等不同方面。

9.1.1 逻辑架构模型

Logical Architecture Models

集中在系统景观的业务或逻辑方面的架构图看起来可能如图 9-1 所示。

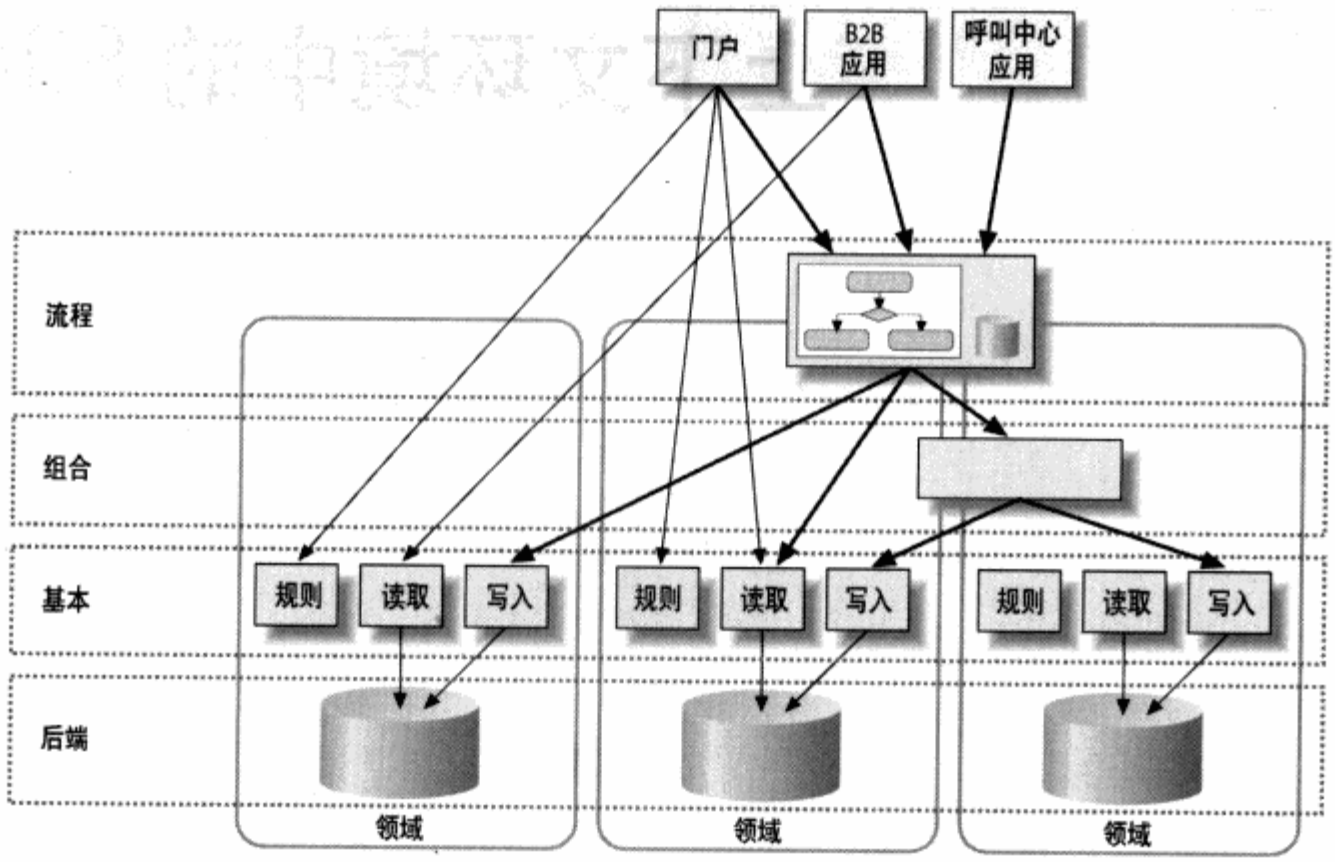


图 9-1：基于 SOA 的逻辑架构模型

在系统景观内部有不同的领域，它们扮演着特定的角色，有特定的职责。一个领域通常是你可以在组织结构中识别出来的某种东西。例如，它可能是一家公司、一个分区、一个业务单位、一个部门，或者一个团队。重要之处在于，它必须是负有清晰职责的“自然的”实体。

这个模型中的后端指的是领域要对其负责的技术系统，以及与技术系统相关的数据和能力。技术上，后端可以是单个系统、不同系统的集合，甚至是与其他领域共享的系统的一个组件。

理想中，完美的领域/IT 队列应该排得让系统边界与业务边界对齐。但是，在实践中，由于历史（和狂乱莫名（译注 1））的原因，情况通常并非如此。在这些情况下，领域应该按技术考量还是组织考量来定义？我建议使用组织边界，因为，之所以要作出这样的决策，就是为了定义由谁对领域相关的服务负责。

如第 7 章所写，特定的领域提供了基本服务。图 9-1 显式了基本服务的三种类型：读取服务和写入服务（根据第 6 章的分类法，它们是基本逻辑服务），以及规则服务（它是提供一般性业务规则的基本逻辑服务）。这些服务包装了后端的技术细节。

译注 1：作者在这里用的是 hysterical 一词。从上下文的意思来看，作者是指这些原因各种各样且非理性，为了和“historical（历史的）”一词达到谐音的效果，所以用了 hysterical。译者在这里将 hysterical 译为“狂乱莫名”，“狂乱”表达“歇斯底里”这层意思，“莫名”表达“非理性、莫名其妙”这层意思。

作为对比，组合服务与流程服务不一定属于单个“自然的”领域，因为它们往往联合了来自不同领域的业务功能。如此一来，谁应该为这样的服务负责？

正如第 8 章所提过的，你可能有一个跨领域的部门来处理这些问题。例如，第 8 章讨论过的解决方案经理可能属于一个负责跨领域服务的“流程部门”。当然，这个部门自身也会成为一个领域，虽然你可能认为它不是个“自然的”业务领域。换句话说，依据你的领域模型，可能有一个“跨部门”领域来处理总体流程（我也见过由履行和订单管理部门来扮演这个角色的情况）。

如果你**没有**跨领域的部门，那就需要些方法来把跨领域服务分配给特定领域，因为总要有人对这些服务负责。在这种情况下，分配服务给某个领域应该是你解决方案设计的一部分。

注意：领域也有可能具备层次结构，这使得规定监管责任的规则更加复杂。例如，你可能用子领域提供基本服务，用领域提供两类组合服务：第一类使用领域内的基本服务，第二类使用其他领域的基本服务。

然而，制定太多规则时要谨慎。记住，SOA 这个概念支持异质性，包括组织结构方面的异质性。提供五个服务的领域（公司、部门等）看起来会和提供上百个服务的领域大为不同。

9.1.2 混合架构模型

Mixed Architecture Models

如果你想在图中加点技术细节，而不是仅仅集中在系统架构的业务或逻辑方面，该怎样做？在前一节介绍的景观上稍加点技术考虑，结果看起来可能如图 9-2 所示。

在这幅图中，你同时能看到架构的逻辑和技术方面。它包含了如下新方面：

- 所有服务调用都显示为通过 ESB 进行路由（结果就是你无法看出哪个组合服务、流程服务调用了哪个基本服务）。
- 流程服务可能由一个 BPM 引擎或工具实现（和管理）。
- 所有提供业务规则的基本逻辑服务都在规则引擎中实现。

虽然包含了一些技术细节，这个视角展现的还是从领域驱动的观点来看的总体架构。重要的是，要明白这第二个视角和第一个并不矛盾，它们只是观察同一个景观的不同视角。

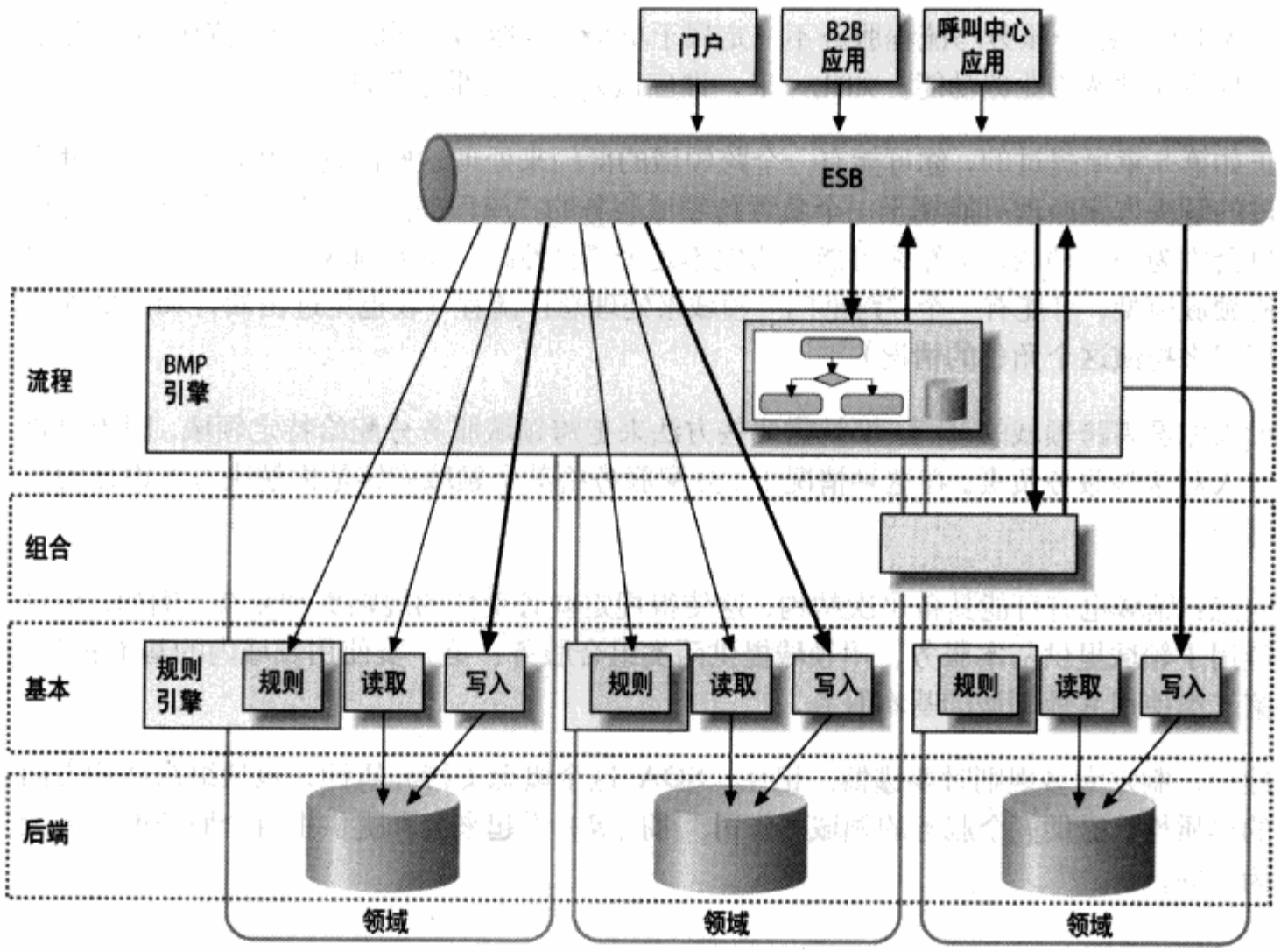


图 9-2：覆盖了逻辑和技术方面的基于 SOA 的架构模型

9.1.3 技术架构模型
Technical Architecture Models

一个更加技术驱动的视角看起来可能如图 9-3 所示。

在这张图中，系统景观的技术方面占据了统治地位。ESB 处于中心。领域仅仅提供基本数据服务和组合服务；基本逻辑服务和流程服务互相分离，因为给它们各自提供了特殊的工具。

观察这个模型时产生了一个问题：领域是否对基本逻辑服务（规则服务）和/或流程服务负责？如果负责，在何种意义上负责？我们在这儿陷入了矛盾中。虽然从逻辑上来说，一个规则服务属于某个普通业务领域，然而，用一个与规则引擎相关联的公共领域来管理所有规则也许更好些。

同样，在一个公共的中央地带定义所有的流程服务也许更为可取：它使你能获得对所有业务流程在设计时和运行时的总体印象。

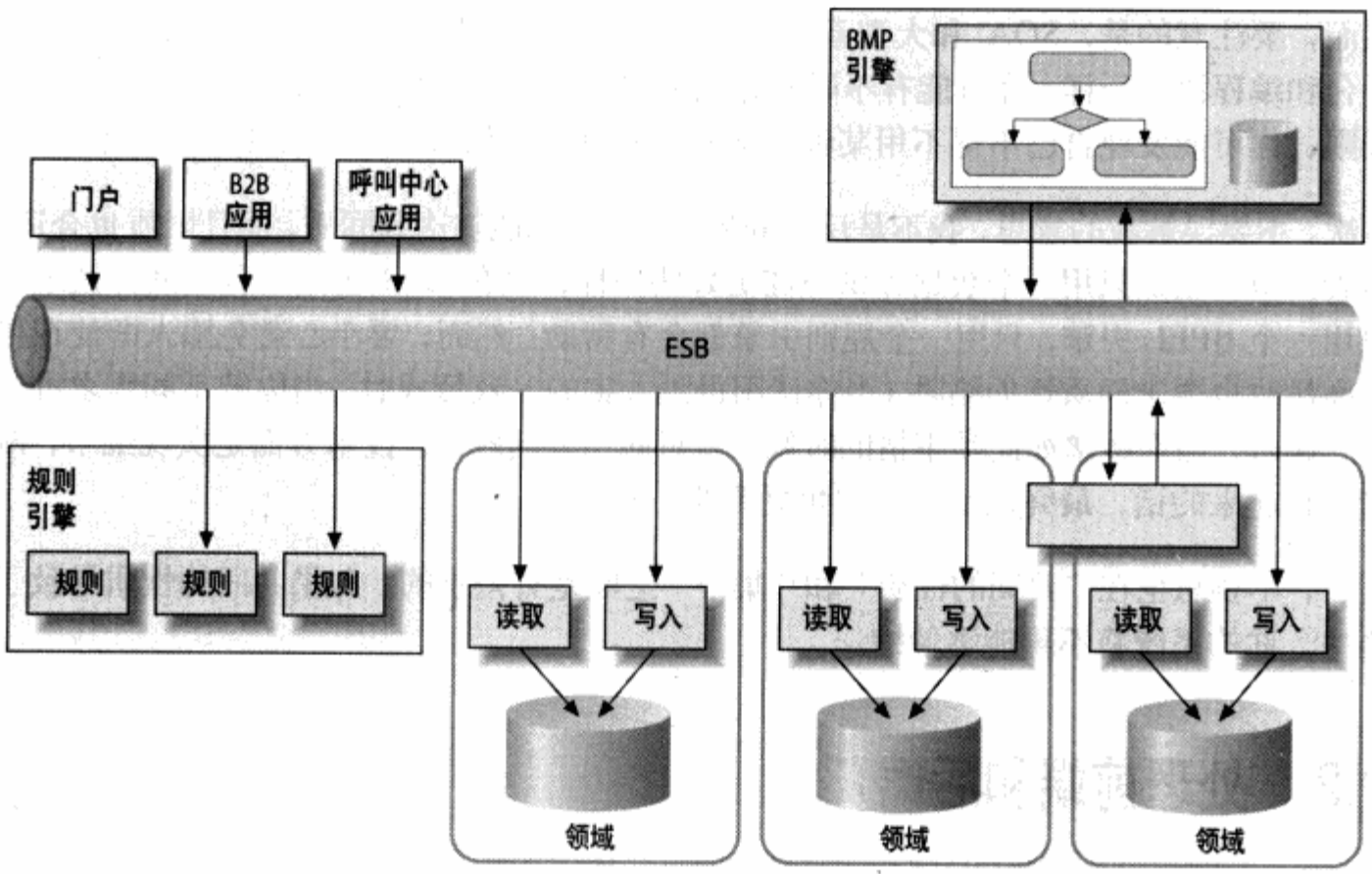


图 9-3：基于 SOA 的技术架构模型

9.1.4 选择正确的架构模型

Choosing the Right Architecture Model

对这儿讨论的不同架构模型，我听过许多赞成和反对意见，而如何处理流程和规则服务是争论的中心。下面是要考虑的几点：

- 一方面，应该由一个领域来负责所有的业务逻辑。因此，这个领域不但要控制基本读取和写入服务，也要控制基本逻辑服务，并且，如果从业务角度来看有意义的话，还要控制流程服务。

- 另一方面，使用在一个团队管理之下的、公共的 BPM 工具、引擎和/或公共的规则引擎，有助于在工具方面得到协同效应，可以形成对所有流程和业务规则的统一看法。

工具供应商一般都青睐技术模型，因为它使得工具供应商可以向你打包销售东西。ESB、BPM 工具，以及规则引擎甚至都可能被认为是一种常用的 SOA 套件（或一种“高级 ESB”）。与管理所有服务的业务库结合起来后，这种方法给了你一种同质的手段来管理整个业务，只有基本服务的实现细节处于这个中心范围之外。使用好的工具，不管在设计时还是运行时，你都可以轻松地获得对所有服务、业务规则和业务流程的概览。

然而，要注意的是，SOA 和大型系统都被设计得允许异质性。正如你很可能有不同的平台和编程语言一样，你可能有不同的方法来实现流程和规则。例如，因为 BPEL 有其局限，你可能发现自己不得不用某种平常的编程语言，如 Java，来实现某些服务。

再次，不要误解我的意思，我不是反对同质性。如果你能够达到同质，而且同质也合适，尽管去做。正如只用一个数据库供应商更好些一样，只在 ESB 上用一种中间件技术，只用一个 BPEL 引擎，只用一个规则引擎都会有帮助。然而，要小心避免陷入即使在代价高昂时也要求同质性的陷阱（当你试图得到大家的一致赞成时，当你把不同需求堆砌到一起时，当你做诸如此类事情的时候，就可能引发问题）。技术方面是实现细节，两者比较起来的话，最终还是业务视角更重要一些。

还有，不要忘记在“共同的性质”和“集中”之间是有差别的。如果共同的性质导致了集中，你的系统将不再能够伸缩。

9.2 处理前端和后端
Dealing with Frontends and Backends

现在，让我们讨论将 SOA 与前端和后端进行集成时的一些特殊考量。

9.2.1 角色和职责
Roles and Responsibilities

第一个要理解的重要点是前端在何处终止而 SOA 又开始于何处。我们将精力集中在用户界面前端上。

对 B2B 场景而言，SOA 中的服务是其接口。这些场景的组织方式是这样的：系统或组件与另一个系统不时进行通信，读取或写入一些数据。在这些服务调用之间，服务消费者使用自己的业务逻辑执行任务。

这就意味着作为服务消费者的前端在一定程度上控制着总体行为。这与那种没有任何业务逻辑、“只”展示数据的前端不同。对后面这种前端来说，工作流在服务器端是合乎逻辑的（虽然控制流仍然有可能在客户端）。

在 SOA 中，服务“服务于”消费者，它不会控制消费者。这一点即使对流程服务来说也是如此。因此，流程服务通常不允许有人工交互。实际上，BPEL 是种“批处理语言”，可以定义由其他服务组成的服务。服务可能会长期运行，但不能与用户进行人机交互。

这就导致了针对前端的特定的编程模型。如果前端只是“展示”后端数据，这很容易。前端只要根据用户输入（要搜索的数据、逻辑 ID，如此等）调用基本读取服务就行了。

如果最终用户想要“**修改**”后端数据，事情会变得复杂些。例如，可以使用原始的办法，让前端收集用户的输入，据此攒出一份新合同，然后将这些数据变成订单，交给流程服务进行处理。

然而，对复杂的数据来说，只有输入数据的特定组合是有效的。于是，前述方法会造成许多流程服务因无效的输入数据而碰到错误。为了避免这种情况，你必须引导最终用户，只给出有效的选项并尽早检查输入。有趣的问题是，怎样才能保证用户不输入无效数据，因为只有这样，分布在各个后端处理这些输入的流程服务才不会失败。

根据图 9-4，当你开始收集用户数据时，也同时使用基本服务来限制或验证用户输入。这就是说，当用户键入想要的东西时，使用不同的基本服务来读取特定的用户数据和一般业务数据。前者会是典型的基本读取服务，而后者会是基本逻辑服务（规则服务）。有了这两个东西，你就能限制选项，或者验证用户输入。

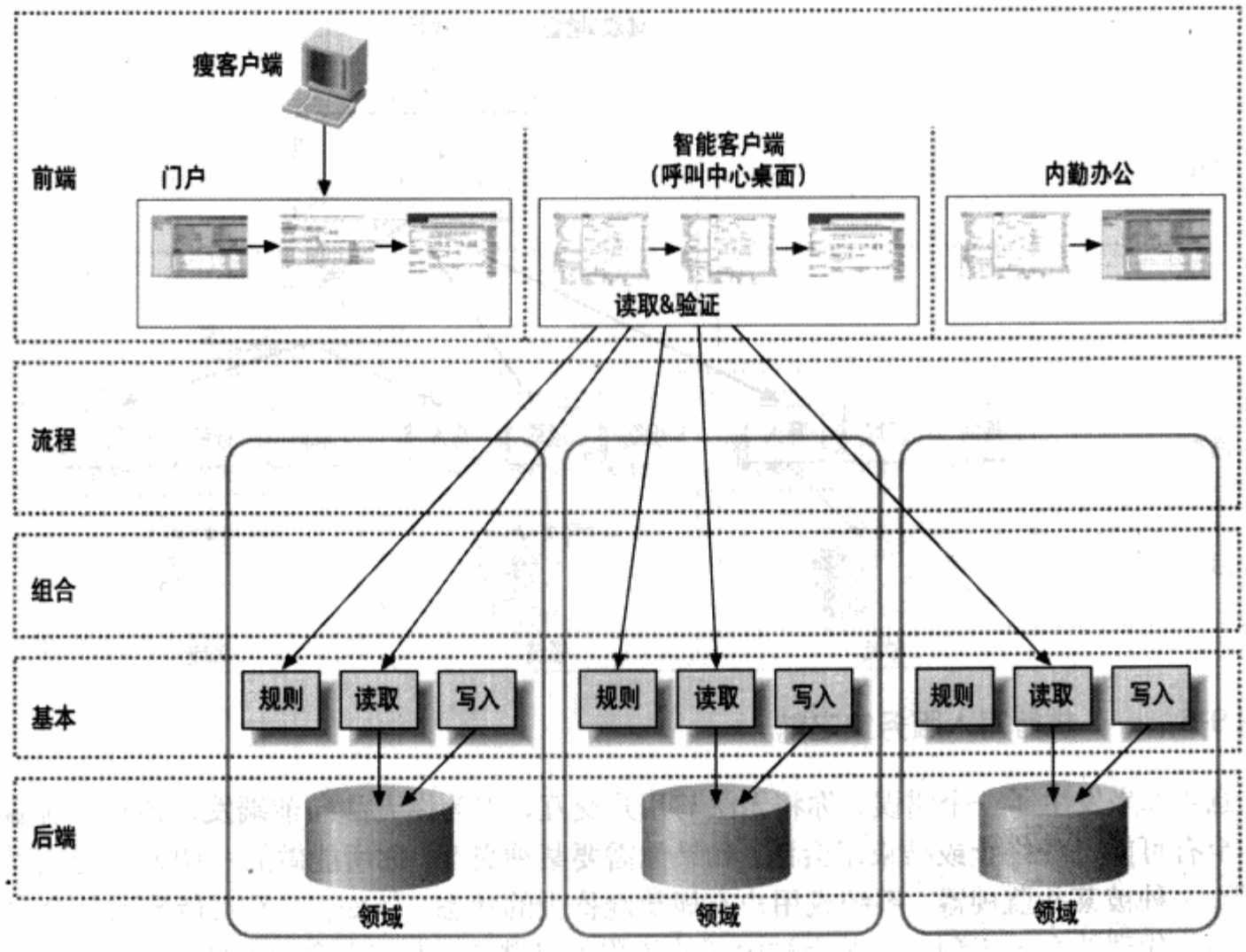


图 9-4：前端读取和验证……

然而，要注意，你不应该对每个输入区域都调用服务来验证。实践证明，最好在前端冗余实现一些稳定的业务规则，而不要对每个验证都调用服务也许更好些（我们将在以后进一步讨论冗余验证方面）。

当输入收集完毕后，通过直接或间接调用一个或多个写入服务，你给后端发出更改请求。图 9-5 展示出调用流程服务的例子，流程服务使用组合服务与基本服务，在多个后端执行修改。

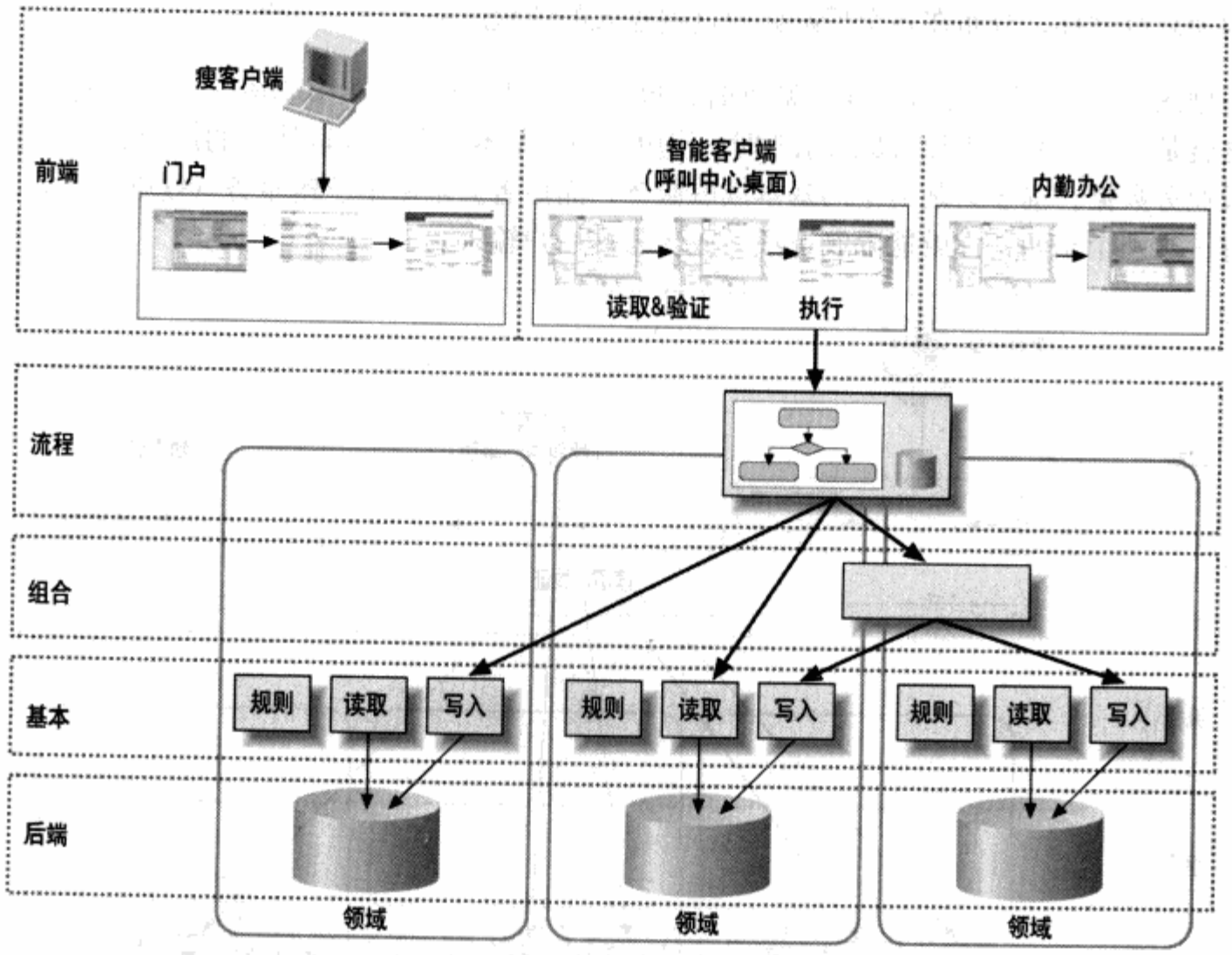


图 9-5：……执行写入服务集之前

注意：如果发生了一个错误，你将无法与用户交互，因为你无法向前端发送请求。前端甚至有可能已经终止或结束了会话。如果你需要某种交互，你所能做的一切就是把流程带入一种被某些监视器、系统或用户有规律地检查的状态。例如，这有可能导致某个内勤办公代理的介入干预，让某个被中断的流程得以继续执行（如图 9-6 所示）。

虽然这种编程模型有其局限，但它不一定是糟糕的，这里的角色和职责都很清晰。例如，在一个订单管理系统中，前端有收集（及预验证）订单的任务。然后，这些订单被当作流程服务执行。在订单被处理时，前端可以连接到运行中的流程来获取状态，但是，一旦发生错误，流程所能做的一切就是进入一种阻塞状态，直到第三方进行干预为止。

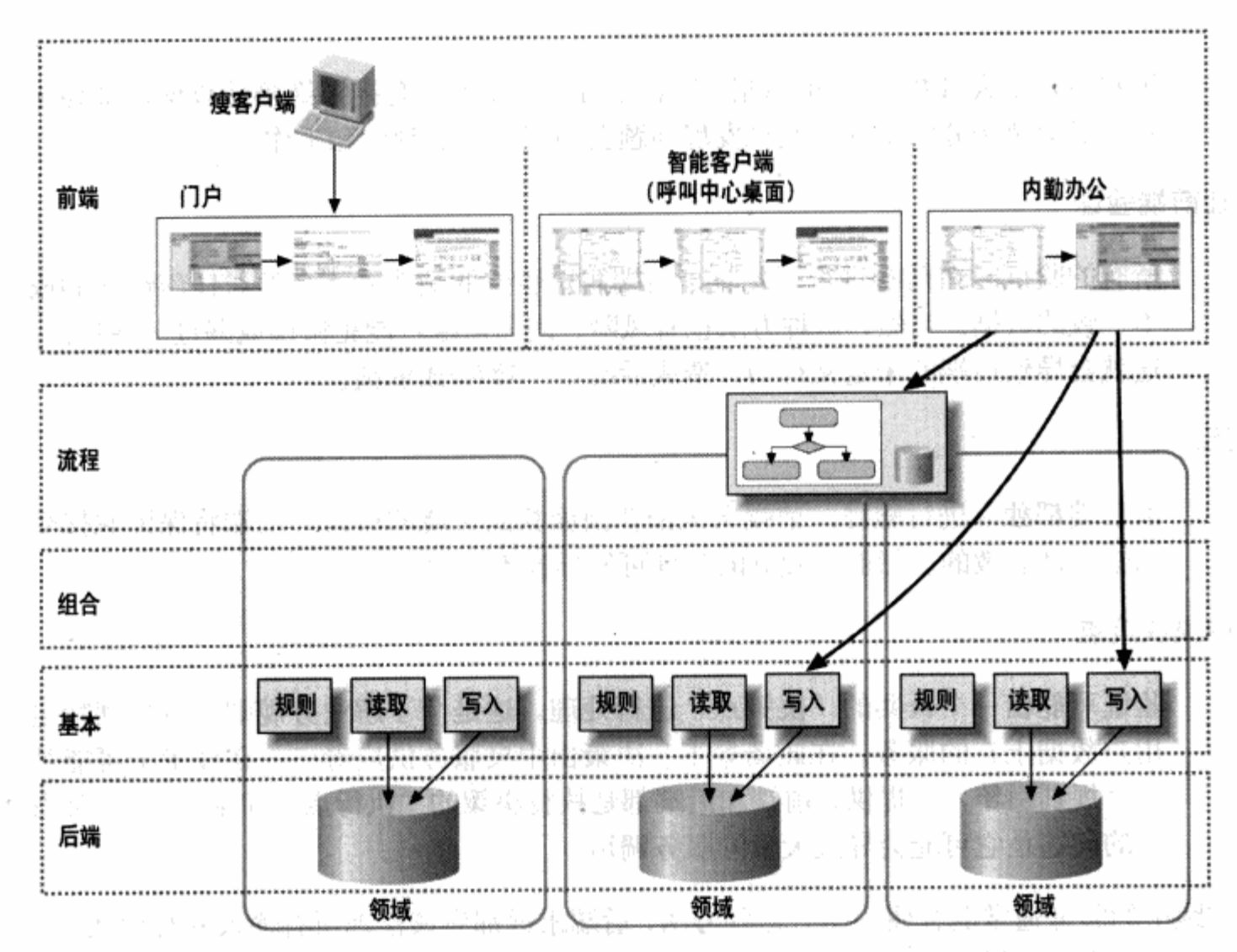


图 9-6：一个内勤办公系统可能接续一个中断的流程

可以做些努力来绕开流程服务不能处理用户输入的限制。一个选择是使用“表示”服务，依靠某些“基础设施魔术”，这种服务能和用户交互。也有一本由 SAP 和 IBM 共同编写的，称为“*BPEL4People*”的白皮书，讨论了如何在 BPEL 标准范围内支持用户交互（见[WS-BPEL4People]）。

然而，转换到这个模型上时要小心。该模型并不是不能用，但是，你应该明白它会怎样影响你的整个架构。流程服务会有不同的语义，覆盖了包括用户和其他前端的更多工作流概念。换句话说，你应该使流程服务的语义清晰明了：它们是作为一种批处理运行的“后端工作流”的一部分还是可能拥有人机交互的“前端工作流”的一部分？

9.2.2 多层验证
Multilayer Validation

我已经提到过用预验证输入来提高订单和其他业务流程成功执行的可能性，但是，这引发了“如何”预验证用户输入的问题。原则上，你可以用以下的基本验证/预验证政策：

在后端验证

所有的验证只在后端进行。此时，前端不做任何预验证。使用这种方法时，如果

用户填入了大量数据，而后端因为最开始几个输入项有错拒绝接受数据，最终用户可能会感到非常受挫。及早发现问题会节省大量的时间和工作。

在前端验证

所有的验证只在前端进行。此时，后端假设服务的消费者不会发送任何错误的或不一致的数据。当然，这种方法也有风险，因为前端可能犯错误或漏过一些错误，这就会导致后端的未定义行为，造成后端不一致和/或崩溃。

冗余

前后端都独立进行验证。前端会试图在前面找到可能的错误，后端将保证它接受的输入是有效的。然而，独立的验证可能导致不一致。

中央决策点

验证可能由一个公共的“决策点”进行处理，这是个了解业务规则（应用到特定用户数据上）的服务。在此场景下，决策由中央服务供应者——在技术上可能是一个规则引擎——提供，前端和后端都是执行决策的“执行点”（注 1）。这种方法的问题是它可能会导致大量的服务调用。

实践中的验证通常混合使用所有这些方法，后端永远都应该检查所有输入的有效性。总的来说，使前端预验证服务输入是个好主意。然而，预验证无法做到完美，否则前端就可以实现整个业务逻辑了（包括处理可能的竞争条件）。一般说来，你必须提供合适的流程来处理事情会出错这个事实。预验证的目的应该是保证流程服务“通常”会成功。小心点，不要在预验证上花不恰当的力气。

在前端和后端验证都提出了一个问题：是对它们分开编程，还是使用公共服务（比如规则服务）？再次，你需要这两者的混合体。冗余本身并不一定是坏事，并且，试图消除冗余的付出也许比收益更多——你必须为要执行的每类检查定义服务，在运行时不断调用这些服务也将造成性能障碍。实践中，频繁变动的业务规则应该集中到一个中央地带。例如，这样的规则可以应用到定价模型、产品组合、合同类型等上，但是，用一个中央服务来检查一个日期或称呼是否有效，并且对每个输入都调用服务，可能就不太合适了。

注 1：区分“**政策决策点**”（*PDP*）和“**政策执行点**”（*PEP*），是处理一致的、公共的行为的一般性概念。第 14.3.3 节还有一个例子。

请注意，业务规则通常是某种主数据，如果你对它们提供了服务，仍然存在问题，系统是应该每次都检查它们的输入，还是每天下载一次规则，在本地执行检查？针对你具体的 SOA 和基于 SOA 的解决方案，你必须决定正确的验证政策和正确的验证量。

9.2.3 多前端通道场景
Multifrontendchannel Scenarios

要求分布式系统支持多通道方式很常见。例如，一个客户可能使用在线门户开始一个业务流程（比如下订单），然后通过呼叫中心继续这个流程。同样，通过呼叫中心应用程序开始的流程可能由内勤办公系统处理，被客户用门户监控。

为了支持这样的行为，你必须能够在不同的前端通道间切换，而不是独立地管理每个通道的流程，这样才能保持流程的一致性。

实践中，这意味着流程引擎必须提供定位和识别运行中流程的能力。理想中，标识符是一个业务 ID，如工作请求编号。但是，显然大家并不经常在手边有这类 ID（当大家甚至不知道有这类 ID 时更是如此）。

当我的一个客户实现了这类场景时，这个问题的一个有趣的方面浮现了出来。大家发现，不同的前端通道使用不同的搜索标准，正如它们从不同的视角来管理数据一样。例如，一个前端可能用合同号为标识为某个客户启动流程，而另一个前端使用客户号，第三个前端则可能使用客户地址来查找运行中的流程。

问题是，怎样才能把所有有用的搜索标准赋给运行中的流程？应该由每个前端负责把所有搜索标准赋给其他前端吗？也许不应该，因为这会占用大量时间，而且，每当一个新的前端需要某种额外的搜索标准时，每个前端都必须被修改。一个更好的解决方案可能是：让流程引擎不但负责流程服务，还负责给流程服务添加所有有用的搜索标准，使服务可以被搜索。

9.2.4 多应用前端
Multiapplication Frontends

另一个常见的场景是，在同样的系统或桌面机上运行的不同前端，需要由服务提供相同的数据。

考虑一个桌面机，在此之上你有多个图形用户界面客户端程序处理着公司的同一位客户。由于历史原因，有一个前端提供的业务流程和其他前端不同。假设你从一个前端开始，它调用 CRM 部门提供的适当的服务，加载客户数据。现在，你在另一个前端上需

要相同的客户数据，从概念上来看，第二个前端也需要调用相同的服务来获取客户数据。然而，因为数据已经在桌面机上了，所以这样做是浪费资源。更好的办法是使两个前端能直接或间接地进行本地通信：

- 直接的数据交换意味着两个前端建立和管理着一个交换数据的通信层。当然，这样做会增强客户间的耦合，这可能合适也可能不合适。
- 间接的数据交换意味着有某种“魔术”知道一个前端加载的客户数据能被其他前端所用。在这儿，我们谈的是在客户端有一个类似于针对服务调用的本地缓存之类的东西，这也可能是你 ESB 的一部分（举例来说，使用本地拦截器，见 5.3.2 节）。同样，这可能合适也可能不合适。

不同的解决方案影响到编程的不同范围：第一种方法与 SOA 没有任何关系（它是两个前端或客户端之间的特殊通信），第二种方法对 SOA 就有影响，而且可能由你的基础设施（ESB）来支持。

注意：两个场景通常都要求某种类型的本地通信，因为必须用某种方式交换信息，指出一个前端在处理和其他前端一样的数据。第一个前端最早装载客户数据，它必须向第二个前端至少发送客户号，第二个前端才能执行对客户进一步的处理（除非对客户 ID 的传送必须由最终用户手动进行）。

因为这可以被认为是种优化，所以，你通常应该从两个前端都调用相同的服务装载客户数据开始，然后观察这种多次调用服务的方法是否会造成问题。

9.2.5 未来管理
Future Management

最后，让我们讨论一下涉及多个后端处理的业务流程的另一个有趣的方面，我称之为“未来管理”。如果一个客户调用来修改合同或地址，改变不一定马上生效。这就是说，作为一个整体的客户和业务流程可能定义修改应该在以后某个时间点发生。

这样一来，你该如何处理未来的变化？原则上，可有如下选择：

- 合理筹划未来管理，让后端能够安排必须在未来才处理的任务。
- 让未来管理成为你流程管理的一部分。

使用第一种办法，你需要这样的服务：允许改变后端状态并且将特定的日期用作参数，另外，后端必须能够管理未来的任务。这个需求通常难以满足：未来的任务不光会被取消，并且也会互相冲突，对其进行管理可能会非常复杂。因此，把未来管理作为你流程管理的一部分通常是更好的选择。

如果未来管理是你流程管理的一部分，你可能会也可能不会让它担负通常的流程引擎的职责：要么引擎可能允许你推迟任务，提供手段来安排未来的任务；要么你可以把管理未来任务的责任委托给一个特殊的系统。在后一种情况下，你的流程引擎仍将负责：

- 直接运行流程。
- 把以后才处理的流程放入管理未来任务的系统。
- 当管理未来任务的系统指出任务应该被执行时，处理那些任务。

这样的区分——用一个工具或系统运行实际的任务，用另一个工具管理未来任务——可能有助于分开考虑问题。但是，把实际任务和未来任务放到一起管理当然也是可能的，并且有其可取之处。

9.3 总结
Summary

- 对基于 SOA 的系统景观进行结构描述可以应用多种模型。有的模型偏重于逻辑视角，有的模型则更集中于技术考量。
- 要成功管理 SOA 景观，逻辑的或领域驱动的、将技术方面考虑为“实现细节”的一般性视角提供了更好的伸缩性，在处理异质性上也能做得更好。
- 流程服务是后端处理过程，通常没有提供与最终用户直接交互的手段。因此，如果前端处理的业务用例会导致后端更改，就必须在执行无需交互的更改的同时，应用支持交互式更改的数据收集模式。
- 为了最小化流程服务运行失败的危险，前端可以根据实际情况和业务规则预验证用户输入。
- 有多种方法可用于针对业务功能验证业务规则和输入数据。通常，后端应验证所有东西，前端应努力保证常见场景顺利运行。同时在前端与后端验证，可以通过提供冗余来实现，也可以通过公共服务来实现。

- 就多前端通道场景而言，你必须处理这样的事实：由某个通道发起的流程必须能够被其他通道仔细研究。
- 当处理同一台桌面机上的多个客户端时，你可能想要避免每个客户端都去调用同样的服务。使用本地通讯和/或本地缓存，你也许能够做到这一点。
- 你必须考虑如何处理未来的任务。你可以选择给后端特定的能力，让它能处理未来的任务；或者，指定一个特殊的地方，让未来的任务在此能得到管理。

第 10 章

消息交换模式

Message Exchange Patterns

到目前为止，我们都认为服务是某些业务的 IT 呈现。在技术上，服务的工作方式是在服务供应者和消费者之间发送消息。我们观察的典型场景是，消费者执行一个服务调用，并得到应答。现在，是深入供应者和消费者之间交换消息的不同方式的细节的时候了。正如你将看到的，在不同通信层上存在着许多消息交换模式（MEP），其中一个模式带来了事件及事件驱动的架构。

10.1 MEP 简介

Introduction to MEPs

在分布式系统间交换数据有各种不同的方法。处理这些差异性的一个基本措施是，对交换数据块的方法进行分类，这些数据块被称为“**消息**”。这样一来，根据对交换消息的不同方法进行分类，我们得到了所谓的“**消息交换模式**”。MEP 定义了在服务调用或服务操作中的消息序列，指明了这些消息的次序、方向和基数。

注意：MEP 是描述不同系统（节点）间通信的一般性概念。详细讨论消息传递和消息样式的各个方面可以轻易地写满一大本书（比如，见[HohpeWoolf04]）。因为篇幅限制，我在此将讨论集中在处于 SOA 上下文环境中的 MEP 上。在有必要的地方，我会用 SOA 的术语，如“供应者”和“消费者”，而不用普通的消息术语，如“发送者”和“接收者”。

10.2 基本 MEP
Basic MEPs

分类 MEP 有各种方法，我将从两种基本模式开始，然后基于它们解释其他的方法。

10.2.1 请求/应答
Request/Response

对 SOA 来说，最重要的模式也许就是请求/应答（有时也叫做请求/答复）了。在这种模式中，消费者向供应者发出一个请求消息，等待供应者发出应答消息（见图 10-1）。应答消息可能包含了被请求的数据和/或对成功地处理的请求的确认。

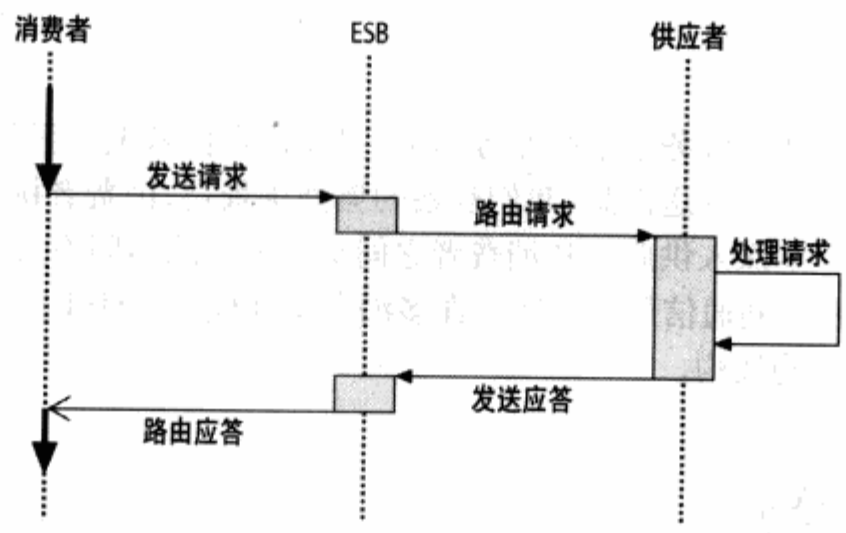


图 10-1：请求/应答消息交换模式

从消费者的角度来看，这样的服务调用就像远程过程调用（RPC）。这就是说，直到应答到达之前，消费者都被阻塞。你可以把这想象成一次电话通话，你问了问题后，要停下来等待回答，然后再继续。

按照这种模式交换信息有一个大优点：它使代码很简单。处理一个服务调用与处理任何其他函数或过程调用一样。当你需要某些信息，或者需要完成某些事情时，你发出请求，等待回答或确认，知道问题已经被解决后再继续进行自己的工作。

这种模式的缺点是：当你在等待应答时，无法做任何其他事情。典型情况是，要么你需要一个快速的应答，要么对运行所耗时间要求不高。在实践中，处理这样的请求的供应者应该时常处于可用状态，并且能够在合理的时间内发出应答。

如果供应者不可用，或者出现了错误，消费者就可能无法得到应答，结果永远停留在阻塞状态。当然，你可以引入定时器，如果在给定时间内没有应答，则开始某种异常处理，但是，你可以认为这是种延伸的、基于基础的请求/应答模式的、更复杂的 MEP 概念。（对这个问题想了解更多请见本章后面的 10.5 节）

注意：你也可以在等待应答时安排做点别的事情。因此，有时人们区分出阻塞和非阻塞的请求/应答模式（有时也称为“**同步**”和“**异步**”请求/应答模式）。再次，你可以把后者想象成另一种稍微复杂点的 MEP，典型情况下，它被称为请求/回调模式（见本章后面 10.3.1 节）。

10.2.2 单程
One-Way

如果你不需要应答，从消费者的角度来看，要做的事情甚至更简单：发送消息，然后就完事了。这种单程模式通常被称为“发射后不管”（见图 10-2）。

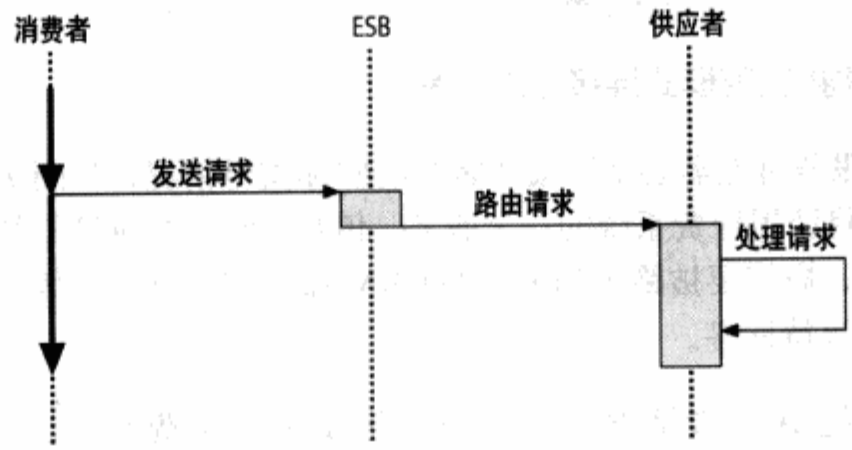


图 10-2：单程消息交换模式

你可能会奇怪我为什么没有先介绍这种模式，因为它看起来最简单、最基础。另外，你可能会争论说请求/应答模式只是两个单程消息的组合。然而，这种想法不对，原因有以下两个：

- 从消费者的角度来看，对两个单程消息的组合带来的是“**异步**”或“**非阻塞**”的请求/应答（或请求/回调）模式，因为从发出最初的请求到应答到达，客户端在其间没有被阻塞（见 10.3.1 节）。
- 从基础设施（ESB）的角度来看，发送两个单程消息会要求第一个请求的发送者（即服务的消费者）能接收第二个单程请求（逻辑上，这是服务供应者的应答）。这意味着消费者也必须可以是一个供应者（即消费者必须能被寻址，能处理到来的消息）。

下一节对此作更详细的解释。

10.2.3 请求/应答对比两个单程消息
Request/Response Versus Two One-Way Messages

假定你设计了一个业务流程，其中系统 A 向系统 B 发送一个请求，而系统 B 有必要向系统 A 给出应答。这个应答可能需要也可能不需要发送给消费者最初所在的进程/线程。

如果消费者进程（这可能是一个前端、一个批处理应用，或者本身就是个服务）需要一个应答来进行下一步处理，最重要的是，应答要递交给执行最初请求的特定流程实例（见图 10-3）。

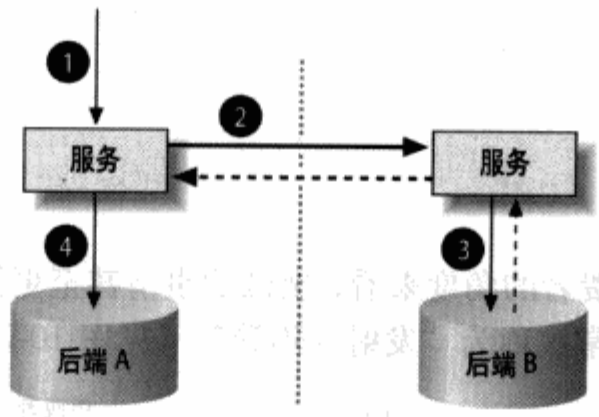


图 10-3：两个由请求/应答模式连接起来的系统

例如，一个 CRM 服务可能需要知道某个客户的实际支付情况，从而决定是否批准某个合同项，因此，它将调用计费系统的一个服务。但是，它需要应答的结果才能继续执行最初的服务，于是，应答要被路由到最初的 CRM 服务实例就很重要。这种情况下，消费者必须被阻塞，等待回答。

另一方面，如果系统 A 需要请求的应答，但那个最初的消费者可以在没有得到应答的情况下继续工作，则“应答要递交给执行最初请求的流程实例”这事就不是那么重要了。相反，你可以把应答消息想象成另一个服务请求，该请求将回过头来针对发起第一个请求的系统（见图 10-4）。换句话说，你可以使用两个单程消息。

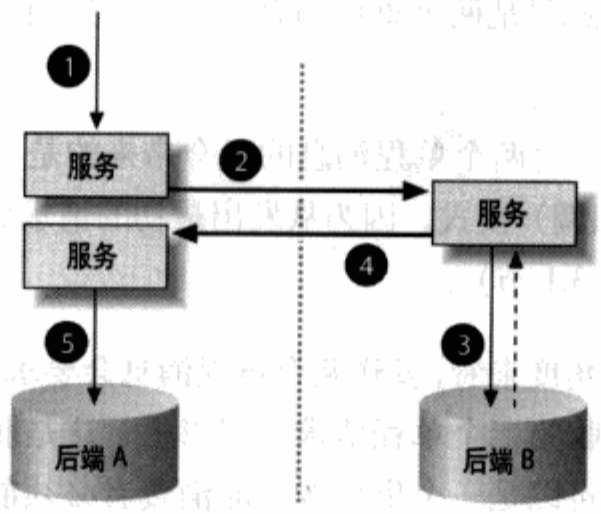


图 10-4：由两个单程模式连接的系统

比如，一个 CRM 服务可能在正常支付间隔周期之外开出一张发票，这个服务可以将相应的请求发送给计费系统。计费系统一处理完请求，就可以调用 CRM 系统的另一个服务，在 CRM 系统中存储新的发票号和日期（当然，消息中也必须包括客户 ID，只有这样，这个“应答请求”才能和正确的客户对应上）。

图 10-5 显示了这个场景对应的序列图。

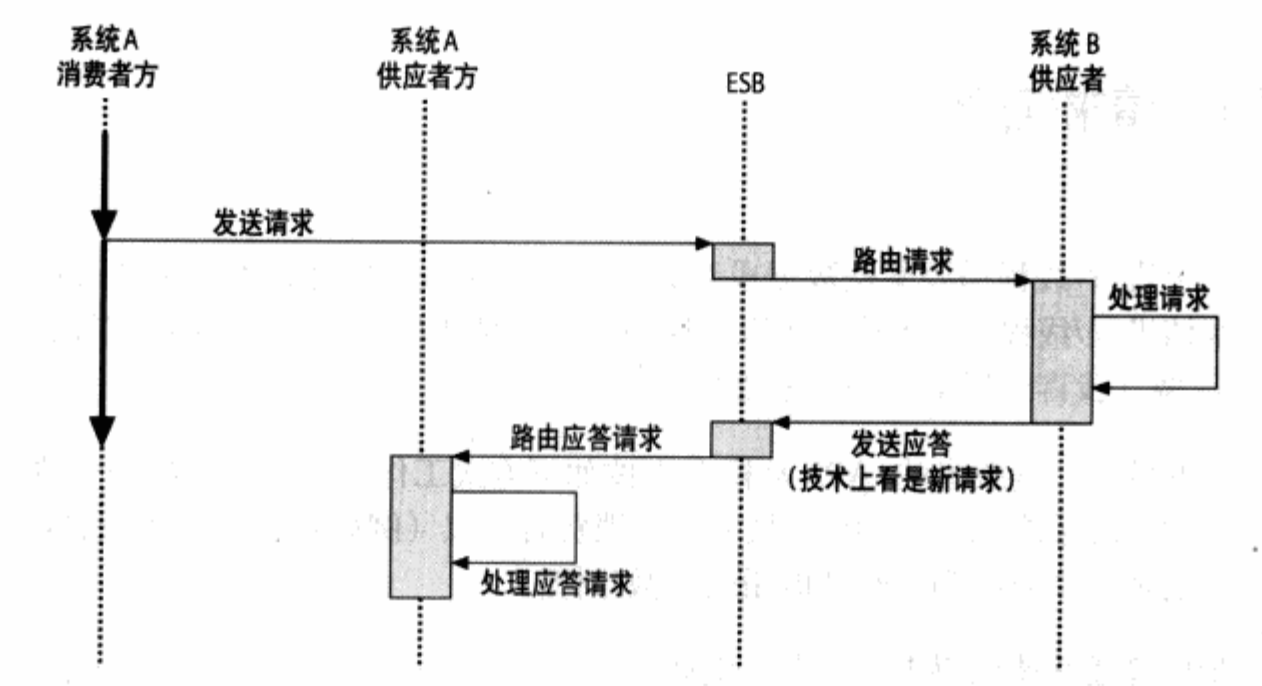


图 10-5：两个单程消息交换模式

正如上面的例子所示，单程消息的一大优点是没有哪个过程被阻塞。这就是说，消费者和供应者没有必要同时可用。实际上，可以在你的基础设施（ESB）中插入消息队列，这样一来，消息一旦被发送就被持久化，不会丢失。这是诸如 MQ 和 JMS 这些“**面向消息的中间件**”（MOM）的核心所在。

> **注意**
>
> 单程消息的序列带来了（业务）流程链的概念，我们将在 10.6 节讨论。

请求/应答消息模式的优点是，应答被递交到触发最初请求的同一个流程实例。

如果你的基础设施不支持请求/应答模式，你就必须自己编程，使请求方系统对应的供应方能找到正确的实例来递交应答。这种情况下，消费者通常给供应者发送一个内部的“返回地址”，供应者将其和应答消息一起送回。处理收到的应答消息的消费者进程/线程可以用这个地址信息，在内部将应答路由到正确的流程实例（即最初发送请求的那个实例）。

10.3 更复杂的 MEP
More Complicated MEPs

对这两种基础的 MEP，有许多种可能的扩展和变化。接下来我们讨论其中一些最典型的 MEP。

10.3.1 请求/回调
Request/Callback

通常，一个进程/线程需要某些数据或对某些操作的确认，但是，不一定非得在应答到来前阻塞该进程/线程。这种模式可以被叫做非阻塞请求/应答、异步请求/应答，或者就叫请求/回调（我喜欢用最后一个术语）。

针对这样的场景，消费者的 API 从概念上按如下方式工作：消费者发起一个请求，指明当回答到达时采取怎样的动作。技术上，消费者可以（例如）定义一个所谓的“回调函数”，这是个当应答到来时被调用的函数/例程。

处理异步应答通常使代码更加复杂。例如：

- 如果你发送了多于一个的请求/回调消息，你就必须处理如下事实：应答回来的顺序可能和你发出请求的顺序不同。这就是说，你必须把应答和当初的请求相匹配。通常，靠引入关联 ID 来做到这点。关联 ID 和请求一起传送，并和相关应答一起递送回来。
- 你必须保证每个应答的上下文环境仍然有效，仍然包含了所有要用来处理应答的信息。
- 另外，你可能要处理这样的事实：有的请求没有收到应答。如果应答不仅仅是提供某些额外的、“有则更佳”的信息，你就必须能检查出应答没有回来的情况（例如，靠处理某种最大响应期间），并作出恰当的反应。

这种类型消息交换的一大优点是，它引入了一种形式的松耦合：在请求被发送时，服务供应者不必处于可用状态，并且，在等待应答时，消费者可以继续工作。

每个方法都有其长处和短处，决定是使用同步还是异步请求/应答场景可能很困难。正确的选择依赖于诸多因素，如运行时间、资源可用性、可靠性，以及可维护性等。毫无疑问，异步通信更难维护。但是，在等待时间（有可能）很长时进程/线程被阻塞，也是对时间的浪费。通常，引入松耦合可消除依赖，但增加了复杂性（见 4.2.1 节）。

再次注意，（正如以前所讨论的）如果应答不一定非要递送到发出请求的同一个进程/线程，则使用两个单程消息可能是更好的办法。

10.3.2 发布/订阅
Publish/Subscribe

有时，发送的可能是不要求回复的单程消息。发送这样的消息的理由通常是通知其他系统，有事情发生或情况变化了。这类消息通常被称为“**通知**”（或在 10.6 节讨论的“**事件**”）。

为什么需要发送通知？遵循通常的业务流程建模方法时，有一种总体设计思路可能是：它规定当某种条件出现时，一个系统必须通知另一个特定系统。例如，当一个计费系统向客户发出一个新的账单时，可能要向 CRM 系统发送一个消息，通知 CRM 系统新的发票号。

然而，还有一个著名的模式能使一个系统向特定的通知或事件进行注册或订阅。这个模式通常被称为“**观察者**”或“**发布/订阅**”模式，它是基础的设计模式之一（见[GoF95]）。一般说来，这个模式允许多个“观察者”向一个系统“订阅”，而该系统会在发生特定情况时通知观察者。

正如我刚刚暗示的，在 SOA 基础设施中，你通常只见到这个模式的第二部分：通知。订阅可能是业务流程的一部分，体现到服务设计中。这就是说，在设计时，一个服务契约可能规定了，每当一个客户收到一张新账单时，CRM 系统都从计费系统收到一个事件。其他服务契约（例如和在线门户之间的契约）可能规定了其他系统收到同样的事件。结果就是，当一个客户收到一张新账单时，服务供应者——“计费系统”——都会自动向服务契约中有这一条款的所有消费者发送通知。

注意：在运行时，只有一个从供应者发向每个消费者的单程消息。这是个请求/应答场景的特例，通常情况下，操作都是从消费者向供应者发送一个消息开始。

10.4 处理可靠性和错误
Dealing with Reliability and Errors

到目前为止，如果发生错误该怎么办还完全没有讲过。如下是你可能碰到的一些错误：

- 服务供应者发现一个错误，送回一个出错消息而非典型的应答消息。

- 服务供应者不可用，因此无法收到消息。
- 消息的传输层可能不可靠。这就是说，消息可能在网络上丢失了。

处理这些情况会大大增加消息交换模式的复杂性。

10.4.1 出错消息
Fault Messages

如果服务供应者（或其他任何正在发送消息和接收应答的流程）发现了一个错误，那么，它将送回一个出错消息（见图 10-6），而不是送回一个平常的应答消息。通常，你可以为这些出错消息定义特殊的属性。

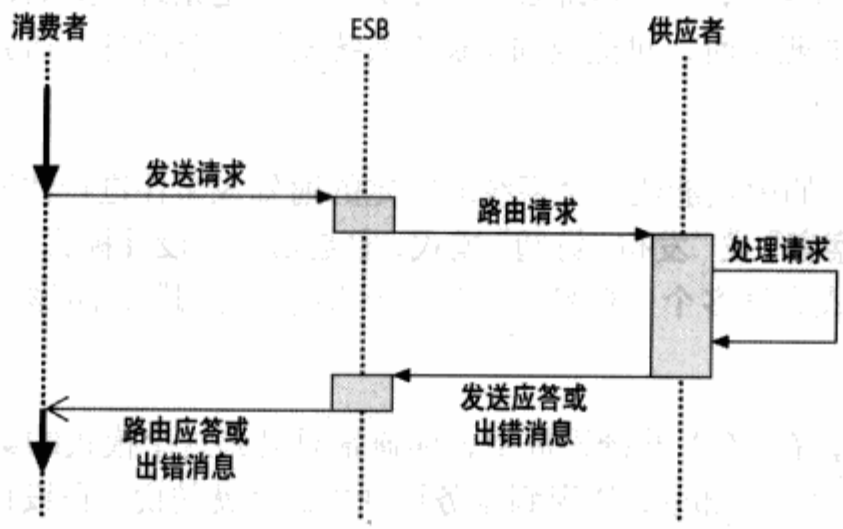


图 10-6：可能出错的请求/应答

在特殊情况下，是否处理及如何处理出错与你使用的协议相关。例如，当（预期中和已模型化的）错误发生时，Web Services 允许你指定和处理特殊的、由供应者返回的出错消息。

10.4.2 技术性错误
Technical Errors

如果有一个技术性问题妨碍了消息的递送，则发送者必须被通知到。然而，做这件事情比听起来复杂得多。考虑一个简单的单程消息，如果发送者发送了不希望有任何确认的消息，那么，在模式的结尾处，无法向发送者保证消息确实被递送了。如果接收者发送一个确认，发送者也收到了确认，最初的发送者这才知道最初的消息被成功递送了。然而，最初的接收者却不清楚最初的发送者是否已经知道消息被成功递送。只有当最初的发送者对最初的接收者的确认再次进行确认，并且所有这些消息都被递送了时，这两方才都知道最初的消息已经被递送，并且对方也已经清楚这个事实。

因此，从双方的角度来看，可靠的消息交换必须包含双重确认（见图 10-7）。

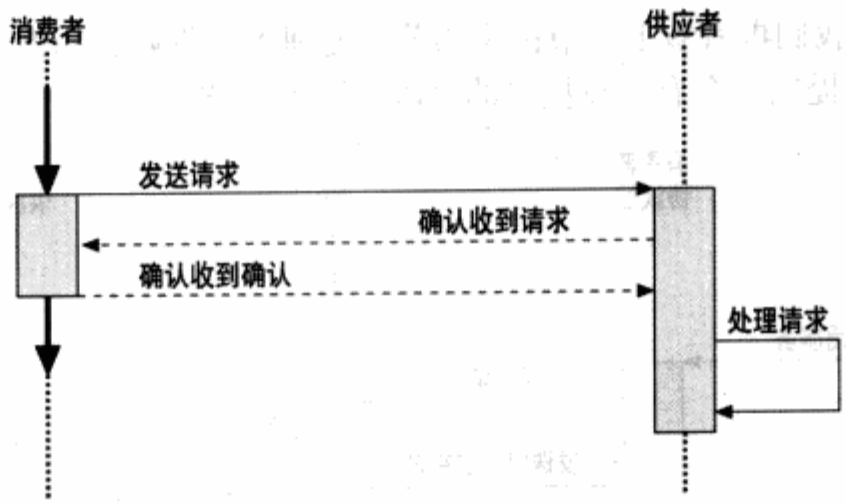


图 10-7：健壮的单程消息交换

使用同步通信时，这类检查通常是基础设施功能的一部分。这是指如果一个（请求或应答）消息不能被成功递送，则消息的发送者应该得到一个合适的异常（前提是底层的传输层没有永久性重试和/或等待）。

使用异步通信时，事情可能变得更复杂些。一个典型的方法是在消息队列中永久性存储所有待发出的消息，然后用固定的时间间隔发送它们。此时，可以发送一个确认，表明消息被可靠地发出了。唯一的问题是，当消息在队列中开始“腐朽”时该怎么办：你可以把消息扔了，或者向一个更一般的监控系统发送异常。你可能希望此时队列也通知最初的消费者，但是要注意，连接已经失去了，而且一般说来，服务消费者无法像服务供应者那样工作。

另一个处理这些情况的典型方法是不断重试发送消息，直到收到相应的应答。然而，要注意这样做时，你需要幂等的服务（见 3.3.5 节）。这是说你要保证重新发送消息不会出现未曾预料到的后果。例如，如果你在发送一个向银行账户中加些钱的消息，你会想要保证假如消息必须重新发送的话，加钱的效果只发生一次。如果你无法得到对这种消息的应答，你不知道是请求还是应答失败了。这就是说，供应者可能已经处理了服务调用，也可能还没有处理。

注意

如何使服务幂等的细节见 15.2 节。

10.5 处理不同 MEP 层
Dealing with Different MEP Layers

消息交换模式永远都依赖于传输层或消息所使用协议的特性。但是，向上或向下一个层，情况可能就完全不同。例如，你可以在同步协议上提供异步消息交换模式，反之亦然。

为了说明这点，让我们再看看上一节讨论的关于处理不可靠协议的例子。即使传输层不可靠，你仍然可以提供一个有可靠接口的 API（见图 10-8）。

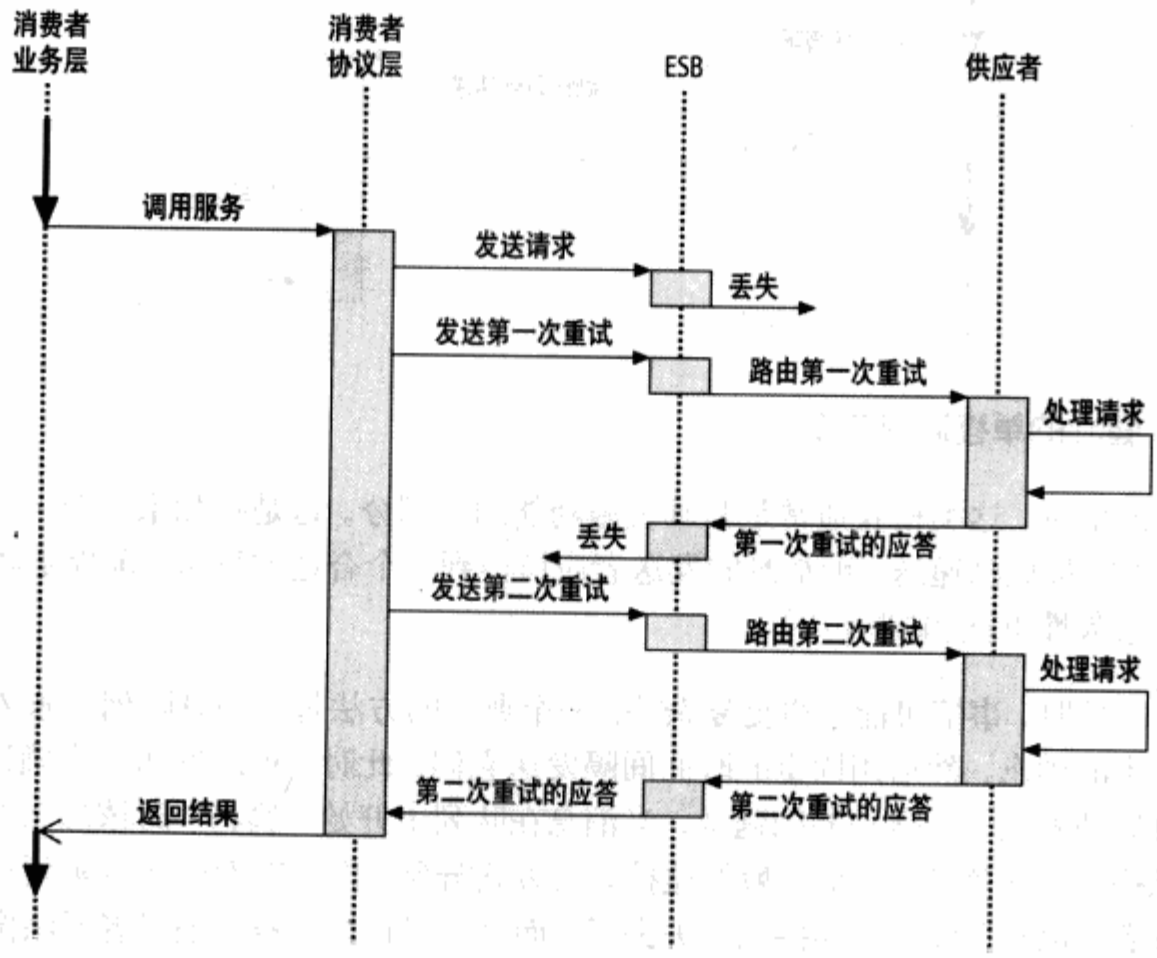


图 10-8：在不可靠协议之上的可靠服务调用

在这幅图中，消费者调用了一个服务，在收到应答（或一个异常）之前，相应的 API 被阻塞。然而，因为所用的协议是不可靠的，所以，在这个 API 之下的代码更复杂。例如，在这幅图中，发送给供应者的第一个请求消息丢失了。于是，在一个超时后，消费者的低层 API 执行了一个重试。这个消息被成功地路由给了服务供应者。不幸的是，这次是应答丢失了，于是消费者又执行了一个重试。这次的努力成功了，请求的数据最终被返回给了消费者。

在这个例子中，消费者使用了支持同步请求/应答 MEP 的 API，然而，低层协议是一系列的请求/回调 MEP（消费者代码处理第一个被成功递送的应答）。注意，协议本身也可以使用其他 MEP，如单程消息来处理内部通信。

从 SOA 的角度来看，一个有趣的问题是协议支持哪些 MEP，以及 API 支持哪些 MEP。如果 ESB 是协议驱动的，则消费者可能要负责编程处理诸如重试这些东西。然而 如

果 ESB 是 API 驱动的，那基础设施团队就应负责为不同的 MEP 提供 API。在后一种情况下，SOA 基础设施可能会提供一些能力，指定重试次数、两次重试之间的超时长短等（关于区分协议驱动 ESB 和 API 驱动 ESB，更深入的细节请见 5.3.3 节）。

当然，事情可能变得更加复杂。例如，有可能既不是因为请求丢失，也不是因为应答丢失，完全因为应答到达耗时太多而发出一个重试。此时，消费者就要处理多重应答。

图 10-9 是这个场景的图示。这里的第一个请求消息丢失了，于是发送了一个重试。重试是成功的，但是，因为结果到达用了太长的时间，发出了第二个重试消息，这个消息由服务供应者的另一个线程处理。因此，消费者最终得到了两个应答（两者甚至可能不按期望中的顺序到达。）

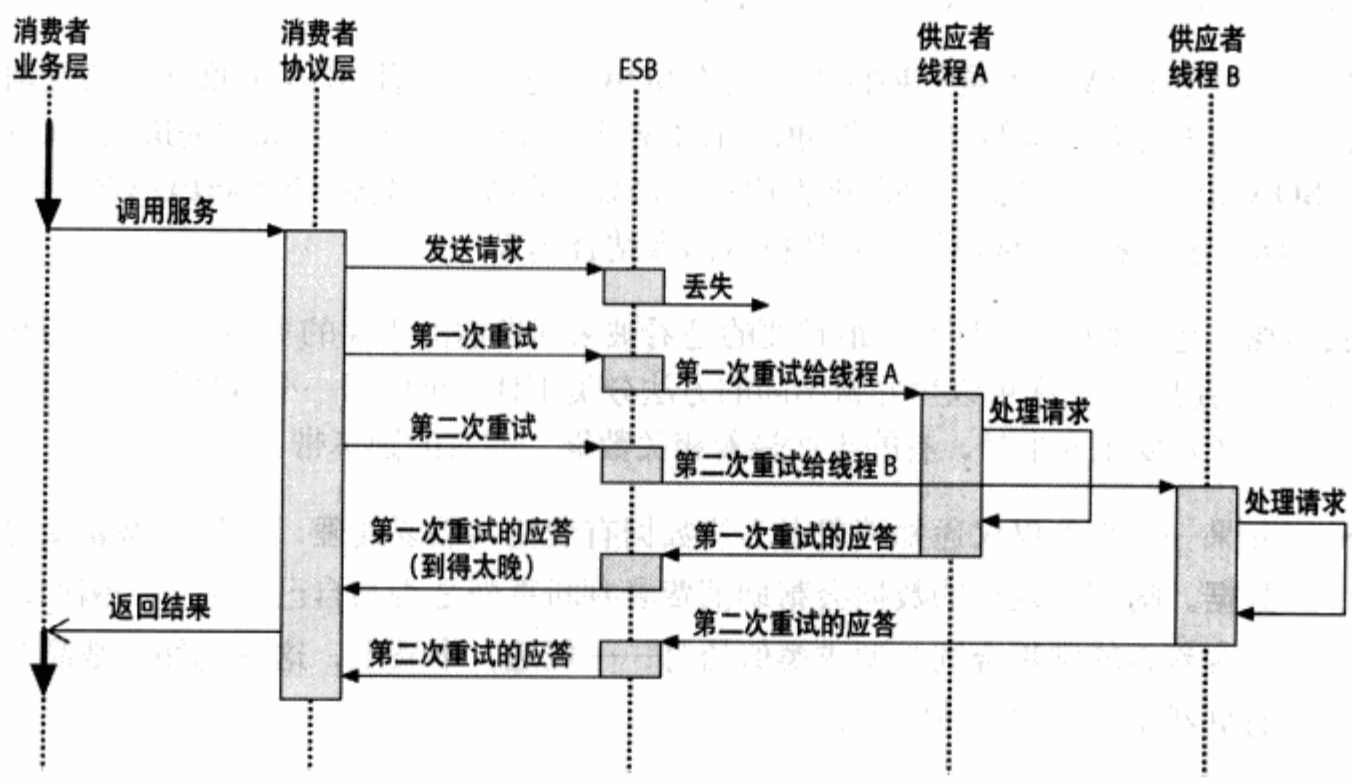


图 10-9：处理不可靠和迟到的应答

幂等性再次成为该场景担心的主要问题。如果消息在服务供应者处理它们之前或之后丢失，消费者没有接收到预期中的应答的话，消费者将无从知晓操作是否发生。因此，一个重试可能会造成另一次影响，如果第二次的结果和只处理一次请求的结果不一致，那将是灾难性的。例如，如果请求是要创建些新东西，这个东西就会被创建两次；同样，如果一个服务调用是向某账户中加入 100 元，最后就会总共加上 200 元（对这个话题的讨论见 15.2 节）。

10.6 事件驱动的架构
Event-Driven Architecture

最近，大家又在热炒一种被称为“**事件驱动的架构**”（EDA）的架构风格。正如该名字所暗示的，EDA 是一种软件架构模式，它鼓励对事件的生产、发现、消费和响应。

在一定程度上，**单程消息**和发布/订阅消息可以被认为是“事件”。这些事件是典型地对状态发生重要变化的通知，这样一来，对这些变化感兴趣的各方就能相应作出反应。

例如，一个事件通知其消费者，电话公司的某个客户已经被屏蔽，该通知的结果就可能是，所有处理该客户的系统都取消了针对该客户的所有功能。

当前，对于 EDA 是不是一种特殊形式的 SOA，是不是一种 SOA 的增强，还是别的什么，人们进行着大量的讨论。例如，有的分析师和公司引入了诸如“先进的 SOA”或“SOA2.0”这样的术语，指的就是 EDA 和 SOA 的结合（例如[GartnerEDA06]，它比较了 EDA 与“交互式 SOA”，并且把这两者结合起来）。

我的观点是，如何进行区分，最重要的是看被来回发送的事件的类型。当然，与服务和对象的情况一样，你可以用非常不同的方法分类事件。事件间一个有趣的差异是，当通知消费者有变化发生时，有的事件带着相关数据，而有的则不带：

- 如果一个事件仅仅通知消费者客户数据有变化，需要处理，那就只需要发送少量数据。随事件发送的数据会帮助消费者判断更新是否与自己相关——例如，可能包含着新的常用合同类型或类似信息——但是，本质上，这些通知只是简单表明有更新发生，需要被处理。
- 如果一个事件包含了要处理的数据，该事件会更粗粒度一些。此时，事件的消费者会收到处理该事件所需的所有信息。

我认为后者是调用服务的另一种形式，不同之处在于（当消费者订阅了事件后）由供应者定义格式，而且用单程方式发送。你可以认为发送通知是与使用服务不同的事情，但你也可以认为这些过程是一种特殊类型的服务，使用特殊的消息交换模式。

在现代分布式系统和业务流程中，上述两种事件都有用。实际上，在几乎所有我见过的 SOA 景观中，事件都扮演了重要的角色。我通常把它看作仅仅是 ESB 的另一种消息交换模式。

和 SOA 一样，对 EDA 也有各种不同的定义。例如，有的定义不要求事件发布者知道事件的消费者。在这种场景下，事件消费者向一个中介事件管理者进行订阅，事件生产者也向这个中介者发布事件。这可以作为 ESB 的一部分进行处理。

这种解耦也是为什么有时候人们认为 EDA 比 SOA 耦合还要松的原因之一。然而，与通常情况一样，松耦合需要付出代价（见第 4 章）。在此情况下，供应者不知道谁对自己有依赖。从资源的角度来看，这不是个问题，因为供应者知道有多少不同的消费者在接收通知没什么意义（与服务的情况不同，有多少消费者向供应者发送服务请求很关键）。然而，当你想要修改和撤销事件时，不知道谁依赖自己就可能出问题（除非供应者能从基础设施那里得到该信息）。实际上，不了解依赖关系通常造成系统不能再维护。

有趣的是，EDA 产生的流程模型可能和 SOA 的流程模型不同（或是其特例）。不是把基本服务组合成组合服务或流程服务，你得到的是某种可以称为“业务流程链”或干脆叫“流程链”（见图 10-10）的东西。和供应链相似，一个业务流程按如下方式组织起来：每个系统都扮演着特定的角色，并且将自己处理过程的结束标记为一个事件，这样后续处理就可以接着进行。

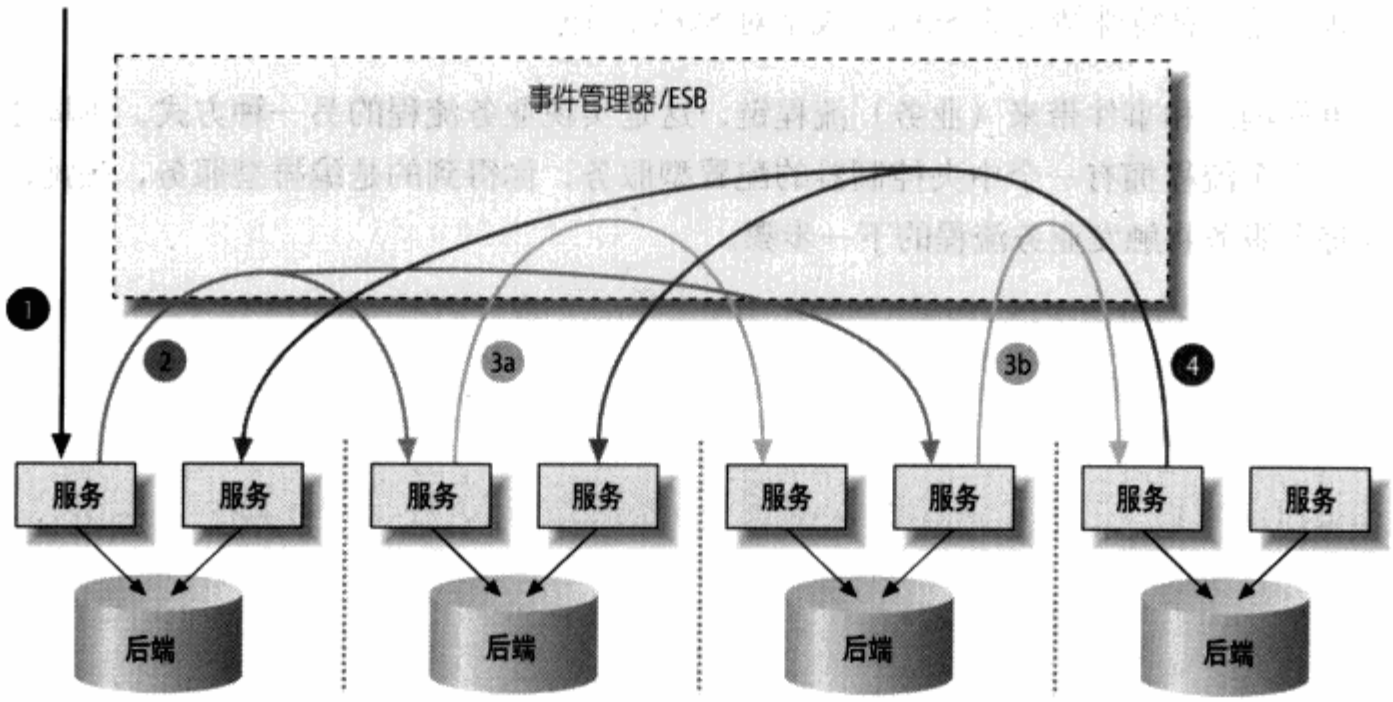


图 10-10：一个流程链

这和编排类似（见 7.6 节）：没有一个中央控制。相反，整个流程链是由事件触发的一个（平行的）处理序列。和编排类似，这种方法有优点也有不足之处。优点是没必要搞

一个可能成为瓶颈的中央控制组件；不足之处是更难理解、更难文档化和监控整个流程或各个流程实例。

10.7 总结
Summary

- 服务使用不同的消息交换模式（MEP），这些消息交换模式定义了完成特定服务操作完成特定服务操作所需发送的消息的次序、方向和个数。
- 基本 MEP 是请求/应答和单程。同样重要的有请求/回调（异步或非阻塞的请求/应答）、发布/订阅模式，以及安排错误处理的更高层的模式。
- MEP 和特定层相关。对 SOA 来说，协议层和 API 层的 MEP 最重要。
- 事件是一种特殊类型的单程消息。事件带来事件驱动的架构（EDA），这可以被认为是一种特殊情况的 SOA，或是对 SOA 的补充。
- 单程消息和事件带来（业务）流程链，这是实现业务流程的另一种方式。不同于对整个流程拥有一个中央控制器的配置型服务，你得到的是编排型服务，在此，每个服务将触发业务流程的下一步骤。

第 11 章

服务生命周期

Service Lifecycle

服务就是一个个的软件，和其他软件一样。如此说来，通常用于软件开发的生命周期也可以用到服务上。然而，服务的生命周期和软件的生命周期还是存在一些差别的。某些差别基于如下事实：我们在谈同时处于开发和维护中的软件。另外，服务生命周期只是更一般的分布式开发过程的一部分。本章讨论服务生命周期及其相关话题。

注意：服务生命周期可以应用于不同的制品，如服务规格说明书、服务实现，以及测试代码。例如，16.3.3 节讨论了 WSDL 文件的生命周期（阐明服务的 Web Services 文件）。

11.1 开发中的服务

Services Under Development

一个服务是一项业务功能的 IT 呈现，如此说来，服务就是一个个的软件。虽然这种软件所用的语言和格式可能不同（例如依赖于实现平台和/或服务分类），但是，它有任何处于开发中的软件通常都具备的生命周期，其核心由各个阶段组成，如设计、实现、集成，以及进入生产状态等（见图 11-1）。

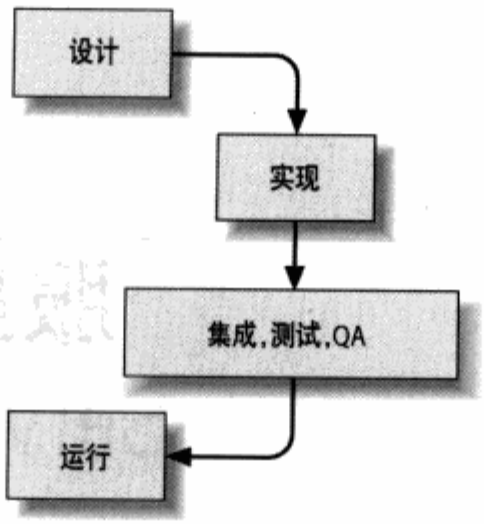


图 11-1：应用于服务的典型的软件生命周期

11.1.1 迭代服务开发
Iterative Service Development

当然，我们在过去的岁月中学到的有关软件开发的东西也能用到这里：实践证明，瀑布式开发方法不好用。我们犯过错误，得到经验，需求和上下文环境也随着时间过去而不断变化。因此，软件开发通常应该是个迭代过程，前期阶段的成果会根据后期阶段获得的经验作出调整。

然而，对服务还要考虑其特殊方面：一个服务是更一般的业务流程的一部分。这样的话，任何对服务设计或实现的修改都可能影响其他系统。因此，你必须考虑何时修改服务才是恰当的。我们首先看看设计阶段。

设计阶段通常生成服务接口的规格说明书。这个接口包括了语义和非功能特性，它也可能是服务供应者和（最初的）服务消费者之间的一个或多个契约。因此，即使是在服务的实现和测试阶段，任何设计的变化都可能影响其他系统（只要服务实现不是由业务职责管理驱动，情况就会如此。业务职责管理驱动会导致在具体的需求出现之前，新服务就被创建出来（译注 1）。见 7.5.1 节）。

这是否意味着你永远不应该修改服务接口（至少在它经过发布，成为了另一个系统接口的一部分之后不该修改）？不！这是个非常、非常糟糕的决定。不要掉进试图把事情做得十全十美的老陷阱，你会因此失败。你应该接受现实：即使是服务的设计也可能变化。这就是说，应该事先考虑到开发中的服务的接口也可能需要变化，并且引入相应过程，使得在设计服务时进行修改成为可能（如果必要的话）。当签订实现消费系统的工作合同时，该合同必须要考虑到服务可能还没有稳定下来。

有了这个认识后，让我们把最初的服务生命周期图改成如图 11-2 所示的那样。

译注 1：业务职责管理驱动时，因为没有具体的需求，也没有具体的消费者，所以，接口实际上没有真正被使用，也就不会有谁去改变接口的设计。

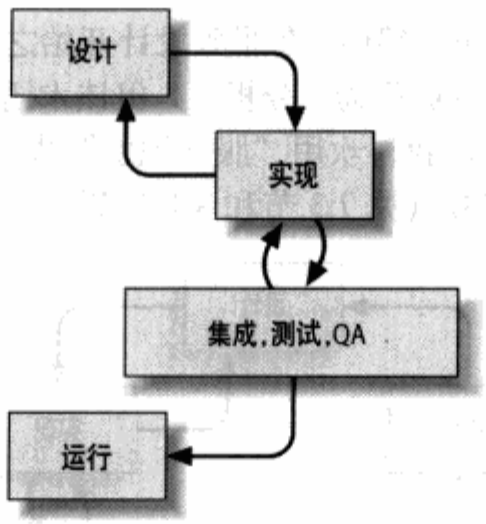


图 11-2：有迭代的软件生命周期

你可能想知道，服务的设计在服务实现时发生变化这种事情，发生得到底有多频繁？当然，发生得有多频繁与设计的质量相关。典型的情况是，设计越好，需要做的修改就越少。然而要注意，市场压力通常导致的情形是没有时间来完成牢固的设计。另外，即使最好的设计师也会犯错误，而且，在大型系统中，几乎从来不存在那么个对某主题知道“所有东西”的人。于是，你应该作出的预期是，服务的设计在服务实现时发生变化是常有的事情。按照我的经验，在实现阶段，少于 50%的原接口设计能保持稳定。

正因为如此，在我知道的一家公司中，服务消费者试图让服务供应者在消费者开始使用服务之前就实现那些服务。早实现的经验有助于识别接口所需要的修改。然而，这个方法意味着在实现阶段，服务没有和消费者集成到一起。因此，只有当这个阶段过去后，有的问题才能被识别出来，这还可能造成其他的问题。

11.1.2 服务识别
Service Identification

服务自身通常并不是目的，应该有别的东西导致对服务的需求和设计。实际上，正如我们经常看到的，服务是更普遍的业务流程的一部分。如此说来，导致新服务被创建的一个典型场景就是（新）业务流程的设计，也叫做业务流程建模（细节见第 7 章）。

然而，这不是识别新服务的唯一方法。另一个方法是业务职责管理，因为某个部门或公司认为提供某些服务有意义（往往期望它们在以后能用得上），所以引入了新的服务。当然，这个方法有其风险，我在 7.5.1 节已经讨论过了。

让现有服务满足新的需求也可能引入新的服务。如果某个修改不是向后兼容的，那就需要一个新服务（或一个新服务版本）。一个新服务版本在技术上被认为是个新服务（细节见 11.2.1 节和第 12 章）。

虽然引入新服务的原因各有不同，然而，在服务设计开始之前要执行某种过程是常见的事情。通常，这个设计前的阶段被称为“分析”，但该术语不一定真的合适。对这个在生命周期中位于设计前的步骤，我喜欢用“服务识别”或“服务发现”（见图 11-3），这可能是解决方案设计的一部分（见 7.3 节和 8.1.2 节）。

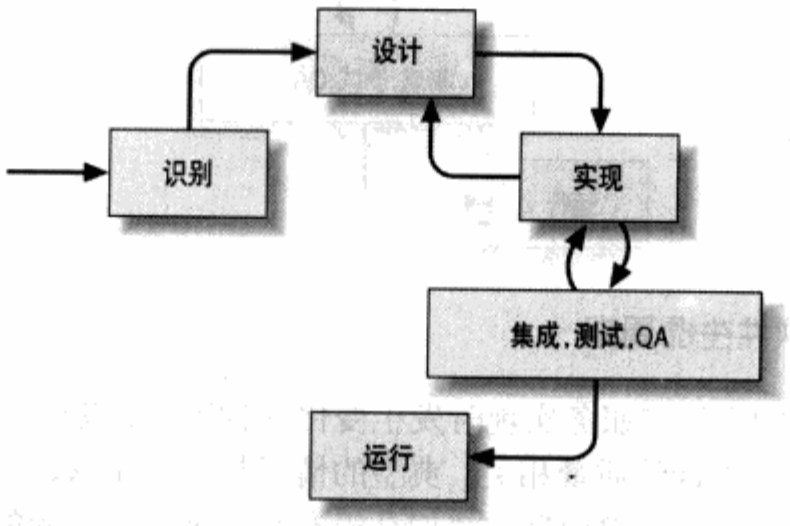


图 11-3：从服务识别开始生命周期

这张图包括了开发中服务所经历的所有步骤。在识别出必要的（新）服务后，我们设计它的接口和契约，并实现它，再执行通常的任务来验证其质量（你可以把这叫做集成阶段、测试、质量保证，或者随便其他什么）。记住，这个过程是迭代的。测试和集成可能会导致修改实现，服务的设计本身也可能需要修改。

很少的情况下，服务开发过程的经验甚至可能导致对服务识别产生影响。这就是说，大家可能发现，新服务没有用。然而，因为这种情况相对罕见，所以我就省略了返回到服务识别的一个箭头。

11.2 生产中的服务

Services in Production

当服务被开发出来后，通常都会被使用。这就是说，软件会被部署到运行中的系统上，执行着对国内公司、企业或宇宙来说多多少少是关键业务的业务功能。

这儿要理解一个重要点。对实现服务的软件来说，这是它从开发中软件转化到维护中软件的时刻。一般说来，SOA 是把开发中的新软件和维护中的现有软件结合起来的概念。然而，对维护中的系统应用着不同的规则。例如，修改变得比以前关键得多，对所有者各异的系统来说尤其如此。因此，我们必须看看如何修改和撤销生产中的服务。

11.2.1 修改生产中的服务

Modifying Services in Production

一旦服务进入生产，它就会被用到关键业务的场景和业务流程中。这意味着只要你修改这样的服务（不管是其接口，或“仅仅”是其实现），你都在修改现有的、使用中的业务流程。如果修改的结果是造成业务流程崩溃，那你就有麻烦了。

因此，当我们从服务生命周期的运行状态向其上一个状态画箭头时，一定要小心。实际上，让生产中的服务保持稳定是一个最佳实践。任何时候，如果需要修改生产中的服务的行为，你应该做的是引入与服务的老版本相互独立的新服务或服务的新版本。同时，如第 12 章所讨论的，让新（版本）服务对现有行为不产生任何影响也是个好办法。

然而，对这条规则有个常见的例外：缺陷修复。如果你找到了一个缺陷，通常你希望在现有的、运行中的服务中修复它，而不是引入服务的新版本。换句话说，缺陷修复和修改之间的区别是，通常你想要尽快把缺陷修复应用到现有的行为上。

有了这个认识，我们可以把服务生命周期扩展成如图 11-4 所示。当服务处于生产状态（“运行”状态）时，你可以修复缺陷，但是，这绝不应该对设计产生影响。如果缺陷修复对设计确实产生了影响，那它实际上就是修改，修改的结果应该是一个有自己的生命周期的新服务（版本）。

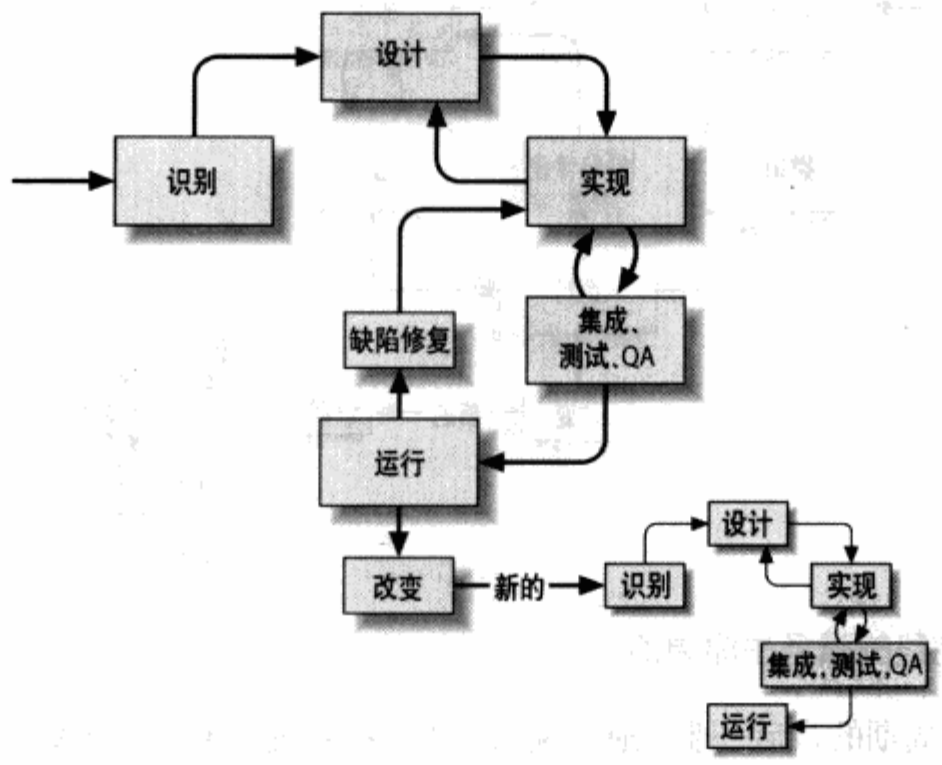


图 11-4：有缺陷修复和修改的服务生命周期

当然，在开发中的和生产中的服务之间有一个灰色地带，在缺陷修复和修改之间也是如此。只要所有涉及到的消费系统都同意进行修改，你可以在不引入新服务（版本）到生产中的情况下修改一个运行中的服务。当然，修改一个还没有被任何消费者使用的服务可能也有其用处。然而，你应该基于一个一般性的原则来对这些事情做出决策，只有这样，所有的服务参与者才能都明白适用的规则、过程和生命周期。

注意

作为对现有服务的修改，引入新服务（版本）这件事，产生了如何处理服务版本划分的问题。对这个话题的详细讨论见第 12 章。

11.2.2 撤销服务
Withdrawing a Service

服务生命周期还缺少一个状态：死亡（也就是说，从生产中撤销）。图 11-5 显示了这个步骤，它也是完整的服务生命周期的图示。

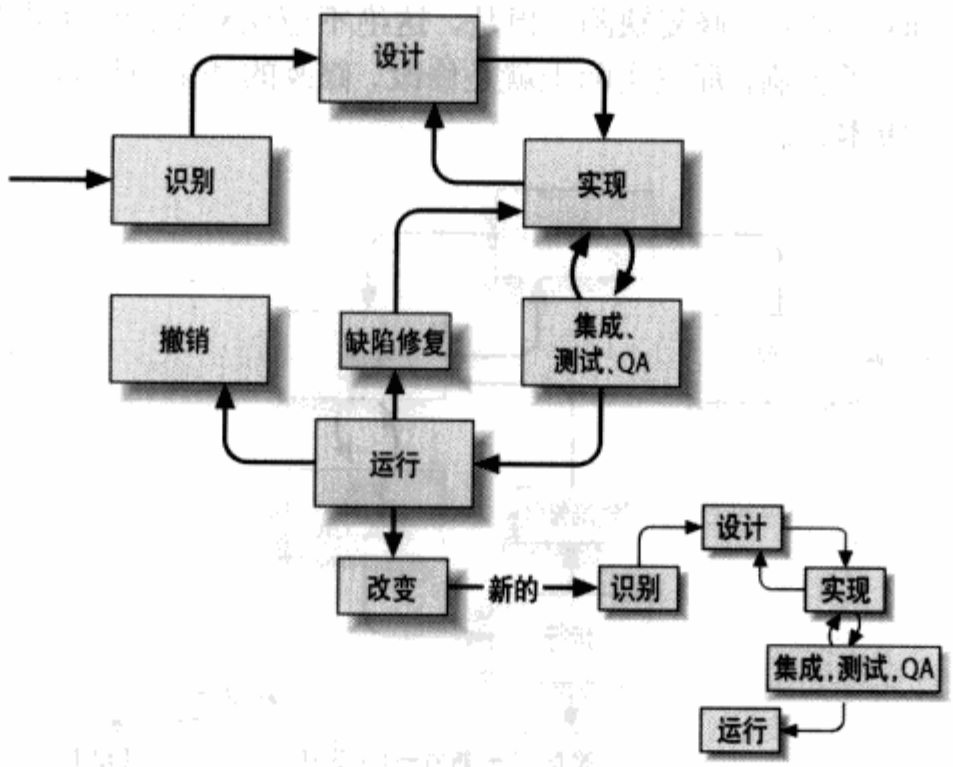


图 11-5：一个完整的服务生命周期

撤销是服务生命周期的最后一步。通常说来，要保持系统可维护，重要的是防止使用中服务和服务版本的数量出现急剧膨胀（尤其是当版本划分政策导致了大量服务版本时，比如生产中服务的每次修改都带来新服务，见第 12 章）。这意味着过时的服务和服务版本必须被逐步淘汰。

然而，在大型系统和公司中，撤销服务可能问题重重。其中一个问题与资源配置有关。清除现有的软件是对系统可维护性的投资，但是，因为没有明显的、立竿见影的业务收

益，这件事的优先级通常非常低。实际上，只有在接近“太迟”了、系统不可维护的危险已经开始变得可见了时，系统维护通常才会得到管理支持。

分布式系统所有者各异的事实使事情进一步复杂化，因为撤销服务必须同时由服务供应者和消费者同意，没有消费者的协作就清除一个服务将毫无意义。如果你单方面行动，就在你撤销了一个使用中服务的那一刻，一个或多个消费者很可能打响了你 CEO 的电话，抱怨业务流程中断了，而恢复流程的唯一方法是把撤销的服务再恢复到生产中。不难想象在这之后你的 CEO 会做什么？

如此说来，撤销分布式系统中的服务的正确方法是什么？你永远都应该遵循一个既定的过程，该过程可能如下：

1. 首先，把服务标记为过时（附上该用哪个服务代替的提示）。
2. 然后，监控服务是否（及由谁）还在使用。这当然要求有一些监控代码，通常说来，它们是你企业服务总线的一部分（见 5.4.7 节）。
3. 如果服务还在继续使用，你应该和相关的消费者联系，讨论出一个解决方案。作为讨论结果，你应该能弄出一个服务最终撤销的进度表。

因为现有服务的消费者从服务的撤销上得不到任何直接的利益，所以，想得到他们的合作可能很困难。转换到新服务（或版本）需要付出努力，你应该认识到这点，然后努力帮助消费者理解，他们实际上是在为供应者代码的可维护性投资。换句话说，你应该努力就升级为整个系统和公司带来的收益创造出一种共识。

当然，另一个可选择的解决方案是，为了让消费者转换到新的服务/版本上，供应者要提供些看得见摸得着的好处，这样一来对所有的参与者就都有某种利益可言了。听起来很傻？在实践中，这种事情多得令人难以置信：“你需要一个新服务？好的，如果你愿意借此机会停止使用那些过时的服务，你就能得到新服务。”

11.3 总结
Summary

- 服务的实现是软件，它有和其他任何软件一样的生命周期。
- 生命周期的触发，通常源自某些能识别出新服务需求的过程（使用流程分解的业务流程建模、业务职责管理、对现有服务的修改，如此等等）。

- 虽然一个服务设计通常就是对其他系统的一个接口，然而，你不应该掉进这样的陷阱——认为在服务的开发期间这个接口不能修改。要确保你与外部供应者定下来的契约是按照这样的方式声明的：接口有可能更新。
- 当服务步入生产阶段后，监管软件的规则就从处于开发下的规则切换到了处于维护下的规则。因此，对既有的服务而言，一般说来，只允许进行缺陷修复式的修改。所有其他修改都应该导致新服务版本。
- 撤销一项服务是个需要协作的组织级过程。

第 12 章

版本划分

Versioning

当你建立 SOA 时，不要期望自己能预见所有事情，以及能一次把事做好。大型分布式系统从来都不是静态的。需求在演化，新需求在出现，同时，当你开发和实现服务时，不停地在学习和进步，因此，你也可能想把自己学到的新知识用于现有的服务。因此，你需要拥有升级和成长的能力。

正如前一章所讨论的，SOA 这个概念必须处理现有的、运行中的解决方案和服务；同时，SOA 也要处理新需求，从而带来新解决方案和服务的开发。因此，你需要创新加上维护的混合体。

本章将讨论如何处理服务的变化。

12.1 版本划分需求

Versioning Requirements

关于服务（及一般意义上的接口）为什么必须被修改，有两个常见的理由：

- 当实现新业务流程时，你通常会对某些以前不知道或设计时没有考虑的方面逐渐产生理解，这就导致服务（实现和接口）需要被修改。
- 新的或改变了的需求可能导致有必要修改现有服务。

有人争论说，只要有好的设计，第一种情况就不应该发生了。我的一个客户甚至有意引入了一个流程，使得服务接口一旦被制定出来就很难修改，目的就是努力让设计稳定。

然而，在实践中，接口并不比其他代码更稳定些。首先一个原因是人们会犯错（这是那种当你意识到后应该可以改正的错误）。其次，我们现在通常都没有足够的时间来完成良好的设计。最后，现实总是和我们的期望不一致。

当你实现业务流程时，你会从这个过程中学习。根据自己新的洞察，你可能决定甚至要修改接口。如若不然，不合适的、糟糕的设计就会在系统中多年保留下去。

现在，假设条件已经给定——服务在开发时，以及在被运行中的系统使用时会发生改变。问题是，如何处理这种变化？SOA 是针对大型分布式系统的概念，你无法要求所有涉及到的系统都在同一时间进行相应的修改，你需要一个迁移过程。另外，你需要一个过程使得你可以在新修订或缺陷修复已经处于开发中的同时，让测试部门能够对服务的修订进行测试。你也需要让服务消费者即使在新版本可用之后，也能继续使用服务的老版本。

原则上，关于服务的版本划分有两个不同的需求。

- 必须能够在同一个运行时环境中有服务的多个**版本**。例如，必须能有两个版本的服务来返回客户数据。
- 必须能够有服务的多个**修订**同时处于开发之中。然而，这些不同的修订不一定要在同一个运行时环境中可用。

前者是业务驱动的需求，因为在运行时，你会看到有服务的不同版本可用，我将在 12.2 节先讨论这种情况。后一种需求通常带来配置管理这个话题，我会在 12.4 节讨论。

12.2 领域驱动的版本划分
Domain-Driven Versioning

领域驱动的版本划分允许同一服务的不同版本同时运行在同一个运行时环境中。这就是说，两个消费者可能用不同的接口来调用相同的服务。其中一个消费者使用较新的服务版本，而另一个消费者可能使用老版本。

有相当多的文章讨论了如何处理这个问题，但是，这些文章通常都作了额外的、与服务自动升级需求有关的假设。然而，自动升级政策会带来额外的需求和复杂性。

我想要展现出来的东西与此相反，这是个无需工作量的（译注 1）版本划分方法，但是，迄今为止，在我所见过的所有项目中，它都被证明是合适的。因为该方法使事情保持简单，所以，要想不用它的话，你得有很好的理由。好的，在简略地介绍其他方法之前，我们先看这个方法。

12.2.1 无需工作量的、领域驱动的版本划分
Trivial Domain-Driven Versioning

针对领域驱动的版本划分，我的无需工作量的方法仅仅是：不为版本划分提供任何技术支持。这就是说，将现有服务的每个修改（在技术上）都当作是新的服务。

如果你想要修改一个返回客户数据的服务（比如，GetCustomerData()），你就简单地引入包含了修改的新服务。当然，你应该让新服务一看起来就知道是其他服务的接班人。要做到这点，你可以简单地用相应的命名来体现（例如，GetCustomerData_2()）。为了避免对服务的第一个版本用特殊的规则，你可能选择把第一个服务命名为 GetCustomerData_1()（见图 12-1）。使用这种命名习惯，你的服务名字永远都包含两个部分：一部分指明服务干什么，另一部分指明版本号。

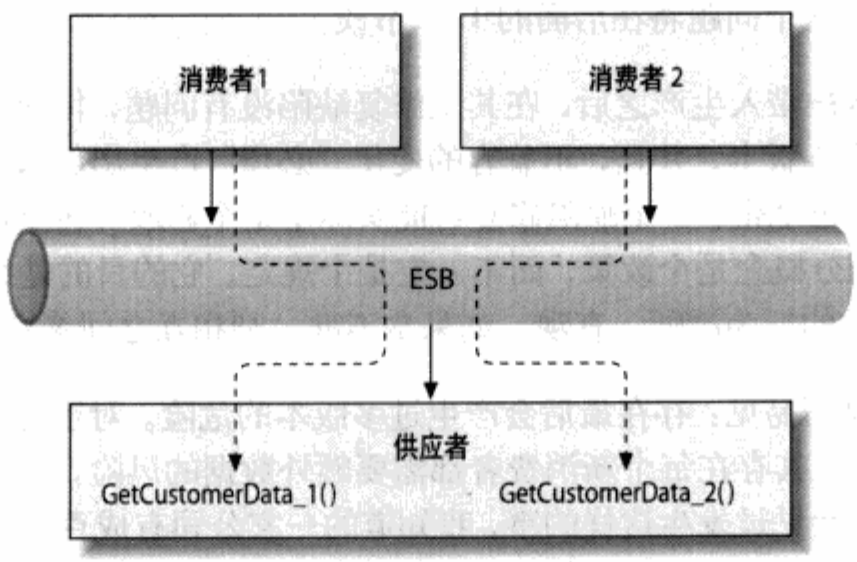


图 12-1：两个消费者用无需工作量的版本划分方法调用两个不同的服务版本

当然，引入新服务到现有的服务中，其成本通常比修改服务要高。因此，在实践中，我对领域驱动的版本划分推荐一个稍微宽松的规则：从服务用到生产中那一刻开始，任何不是简单的缺陷修复的修改都必须产生一个新服务。这个规则有两个重要的推论：

- 在开发期间可以进行任何修改，而且这些修改不会产生新的服务版本。然而，要注意，从服务被第一次使用的那一刻开始（例如在集成测试中），对任何的修改，服务提供者都应该通知现有的服务消费者，并且要与这些消费者讨论（正如当一个契约变化时总要这样做一样）。

译注 1：trivial，指无足轻重的、平凡的、普普通通的等。作者在这里这个词描述版本划分方法，指的是这个方法不必付出什么工作量。所以，对本词译为“无需工作量的”，相应的 nontrivial 译为“需要工作量的”。

- 在运行时，仅仅修复缺陷而不创建新的服务版本（这样做的代价可能更高）也是可能的。当然，这暗示着服务的接口没变。实践中，这可能会带来问题：缺陷修复有时被发现是一种修改，所以从语义上来说，新版本其实更合适一些。但是，为每个缺陷修复都创建新服务的代价通常也更高些。

注意：关于修改是否向后兼容的问题，我还一点都没有提到。实际上，这项政策甚至对向后兼容的修改也适用。

你可能会感到疑惑：如果修改不会影响现有消费者的话，为什么供应者不能在不引入新服务版本的情况下对现有服务进行修改呢？这有两个原因。第一，向后兼容的修改经常表现得不如期望的那样“兼容”。一个典型的例子是，当额外的属性导致更长的运行时间时，结果就是（正式或非正式的）服务的 SLA 被破坏了（一个例子见 13.4 节）。另外，任何修改都伴随着风险。以新服务方式引入修改给了你一个机会，你可以针对单独一个消费者观察运行时行为，当确定一切都情况良好时，再把其余所有消费者切换到新服务。

第二个不引入新版本就不做甚至是向后兼容的变化的原因是，这些变化的结果通常都是数据类型被修改。这个问题将在后面的 12.3 节谈。

总结一下：当服务被带入生产之后，在其上修复缺陷没有问题，但向后兼容的修改“**应该**”产生出新的服务版本，并且，不兼容的变化“**必须**”产生出新版本。然而，要注意的是，此处存在灰色地带：一个缺陷修复可能实际上是个修改（甚至是非向后兼容的修改）。这个版本划分概念是个政策，而不一定是个规定。它的目的是给服务参与者一个公共指南，让正常的行为清晰、直观。如果拿不准，则相互之间多多交谈。

这个方法的问题显而易见：存在最后会产生过多版本的危险。对于返回数据的服务（例如，客户数据），尤其存在每个新消费者都需要额外数据的风险，这样一来，最初的服务就不停地增长。我见过发生这样的事，我知道的一家公司有成百上千的用于生产状态的服务（每个版本都有用），它开始时定下规则是，用于生产状态的同样的服务，其版本不能超过 3 个。实际上，他们经常陷入多达 5 个（甚至更多）版本同时运行的局面。

为了避免有过多版本（这不利于代码维护），你将需要不时地让服务退出生产状态。正如 11.2.2 节讨论的，这经常通过两个步骤来实现：

1. 声明一个老服务版本过期。

2. 撤销过时的服务。

同时要牢记，在关键业务系统（SOA 环境是这样的典型的系统）中，只有当服务不再使用时，你才能清除它。因此，需要进行监控（见 5.4.7 节）。

某些消费者将继续使用老的、过时的服务版本，这种危险总是存在。记住，分布式系统所有者各异，并且对消费者来说，改变服务版本要付出成本，但有可能不会带来任何收益。然而，如果老版本不能逐步淘汰的话，整个系统的统一性就会越来越糟。如此说来，你可能需要从组织角度发起逐步升级的过程，或者供应者可能需要给消费者提供一些刺激，让它们转换到新的服务版本。

从服务生命周期角度探讨这个话题的更多细节见第 11 章。

12.2.2 需要工作量的、领域驱动的版本划分
Nontrivial Domain-Driven Versioning

我们已经看过了无需工作量的版本划分政策，需要工作量的政策看起来会像什么样呢？对这个问题有多种可能的答案。原则上，可选项包括：

- 提供一个机制来保证服务的向前兼容性。这就是说，如果你的基础设施允许你为未来的扩展提供钩子，你就可以按照需要加入扩展，并且把它们标记为可选。现有的服务消费者将能继续像以前那样使用接口，新的消费者将能利用新的特性。
- 引入相应技术，使你能按如下方式扩展服务：对使用老接口的消费者，你能够指定应该做些什么。例如，你的基础设施可能提供这样的机制：可以添加新属性，在没有相应属性的情况下可以指定默认值。
- 提供一个间接方法，这样一来，可以针对不同消费者提供不同的实现。例如，一个服务中间人能决定对哪个消费者提供哪个版本的服务。

如何实现这些方法是另一个问题。例如，你可以用 Web Services 来处理，将不同的命名空间用于不同版本，以及/或者用 UDDI 注册中心（第 16 章介绍）作为中间人，对服务请求进行不同的路由。关于这个方法的更多细节可以参考其他资料，比如，[BrownElliso4]。

12.3 数据类型的版本划分
Versioning of Data Types

当存在不同的服务版本时，也存在着数据类型的不同版本（至少对使用结构化数据类型的服务来说是如此）。处理这个问题比单纯处理不同服务版本的问题要困难得多。

例如，假设在返回值包括地址数据的服务上，添加了新的邮政信箱属性。这就是说，现有的包含了如下属性的地址类型：

```
String street
String zipcode
String city
```

现在加入了新属性

```
String postbox
```

因为老版本服务用老地址类型，新服务版本用新地址类型，所以，在同样的运行时环境中，使用着两个不同的地址类型（见图 12-2）。

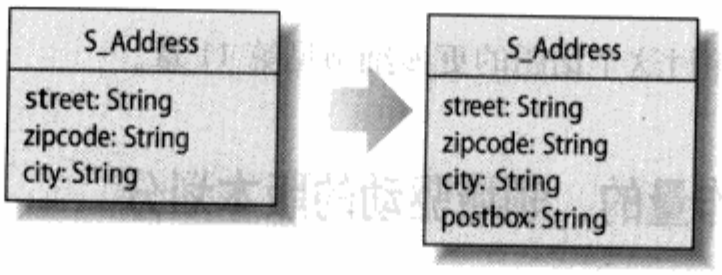


图 12-2：两个不同的地址类型版本

问题是如何处理这个现实？原则上，有 3 种选择：

- 对类型化的接口用不同的类型。
- 对类型化的接口用相同的类型。
- 用泛型代码，这样一来，类型的差别就不重要了。

下面详细讨论这些选择。

12.3.1 对数据类型的不同版本用不同的类型

Using Different Types for Different Versions of a Data Type

当对数据类型的不同版本使用不同的数据类型时，你可能会被诱入一个陷阱：简单地应用我在 12.2.1 节建议的规则来命名和区分类型。然而，因为某些原因，这里的情况要复杂一些。

第一个原因是修改后的类型导致其他类型也被修改。就是说，如果一个类型被其他类型使用了，则其他类型也将发生变化。例如，如果一个地址类型被地址链表类型所用，而地址列表又是客户数据类型的一部分，客户数据类型又被用于客户数据链表类型，则最里面地址类型的变化将改变所有其他类型（见图 12-3），结果是你得到了许多新类型。

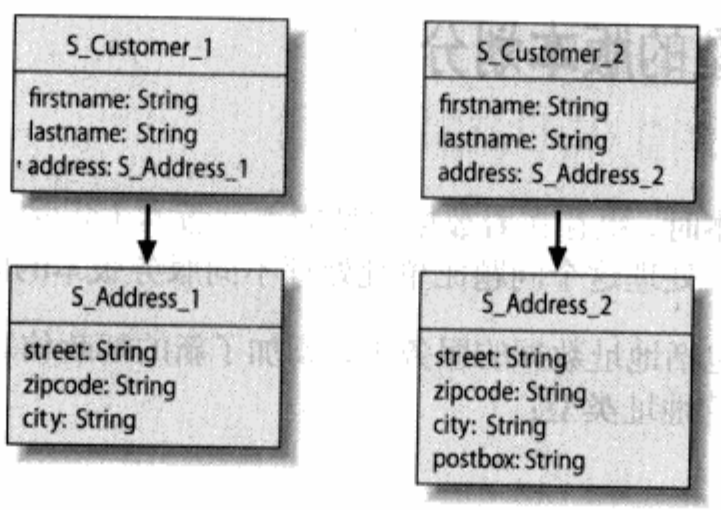


图 12-3：修改被其他类型使用的、版本化了的地址类型

另一个复杂性之所以增加的原因是，在实际中，对于同一类信息，服务有不同的数据类型。在一个类型绑定的编程语言中，这些类型差异的结果就是，不可能将类型当作一个整体来比较、拷贝和赋值。你必须编出工具函数，对不同类型逐个元素（对并非数据类型而两个版本都要的属性，要么忽略它，要么提供默认值）地比较、拷贝和赋值。对服务的供应者和消费者来说，这产生了如下结论：

- 对服务的供应者来说，你必须对同类信息使用不同的类型。要做到这点，你可能需要拷贝和粘贴针对不同类型实现功能的代码，或者针对一个类型实现功能后再将数据映射为其他类型，或者使用泛型代码（模板）。
- 对服务的消费者来说，有可能发生的情况是，你需要一个使用新版本数据类型的新服务，同时，你也需要仍在使用老版本数据类型的老服务。因为这些类型各不相同，所以，你将不得不通过映射数据来处理两个服务使用同类信息的情况。

第二点的影响尤其坏，因为迟早有一天，服务消费者有可能必须处理同一类型不同版本的问题。为了帮助你理解这里的难题，请考虑如下的例子。假设你有两个返回客户数据的不同版本的服务，正如之前讨论的，它们的地址类型不同：一个客户使用 S_Address_1，另一个使用 S_Address_2。现在，假设你有另一个被称为 GetInvoiceData_1()的服务，这个服务返回包括客户数据的发票数据，它也使用新的地址数据类型。结果是，你将面对着如图 12-4 所示的情形。注意：所有这些服务（包括所有的版本）都在被某些客户使用。

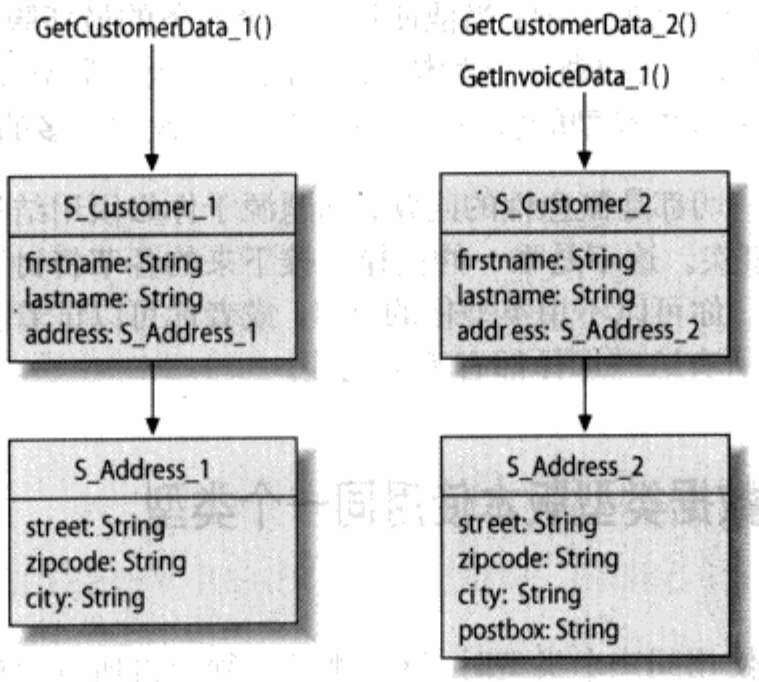


图 12-4：针对不同服务的不同数据类型

现在假设，不久之后，又引入了另一个针对客户的额外需求，规定返回发票数据的服务也应将税号作为客户数据的一部分返回。于是，你引入了一个称为 GetInvoiceData_2() 的新服务，它返回新的数据类型 S_Customer_3。结果是你将面对着如图 12-5 所示的情形。

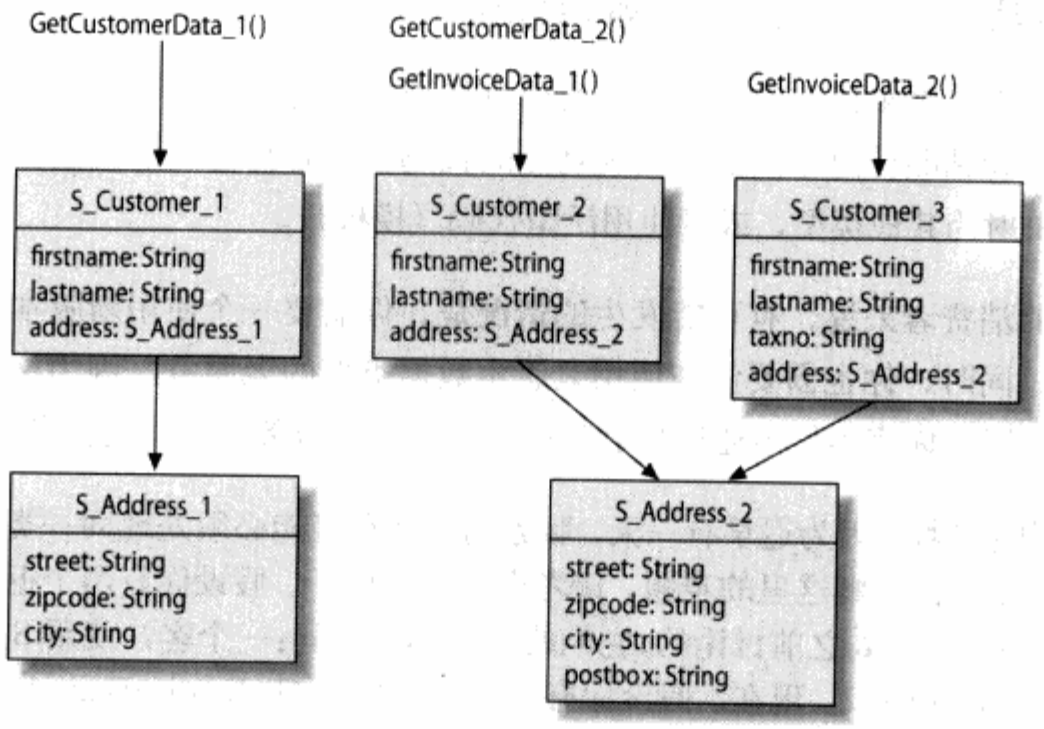


图 12-5：新属性导致对不同服务有更多不同的数据类型

现在，如果一个新消费者想要使用这两个服务（GetCustomerData()和 GetInvoiceData()），它会发现自己处在一个奇怪的境地：当他使用这两个服务的最新版本时，会涉及到两个不同的数据类型。当然，你可以避免这种情况，方法是当数据类型针对一个服务变化时，总是升级所有使用这种数据类型的服务。但是，这会导致非常多的额外的服务版本。

注意：这里所描述的一切都是概念性的问题，问题源于你必须用结构化数据类型支持不同版本的 API 这个事实。这不是唯一的选择，接下来的各节将讨论另外一些选择：你可以不用结构化类型，你可以不用类型化的 API，或者你可以试着在流程内部共享不同的类型版本。然而，所有这些选择都有不足之处。

12.3.2 对不同数据类型版本使用同一个类型

Using the Same Type for Different Versions of a Data Type

当对类型的不同版本使用同一个类型时，这个类型必须包含所有数据类型版本的所有属性。所有服务都使用这同一类型，但是，它们只用那些为它们指定的属性。这个政策带来了 3 个问题：

- 你必须用文档记录哪个属性对哪个版本的服务有效。对复杂数据类型，或被不同服务使用的类型来说，这可能会变得非常复杂。
- 老服务的数据类型随着时间流逝而不断变化。这就意味着这些不同版本是二进制不兼容的。结果是，你必须保证一个流程的所有程序库编译时用的是同一版本的

数据类型。这样一来，如果数据类型因为一个新服务发生了变化，而你又需要这个新服务，考虑到其他所有使用这个数据类型的代码，你必须重新编译所有现存代码。如果不这样做，就会发生非常恶劣的运行时的怪异行为。

- 如果你根据自己的标准验证输入数据，那么，你必须保证额外的属性不会使输入无效。

为了举例说明，再次考虑图 12-5 描绘的情形。用这个方法，不同服务版本都将使用同一个数据类型，如图 12-6 所示。

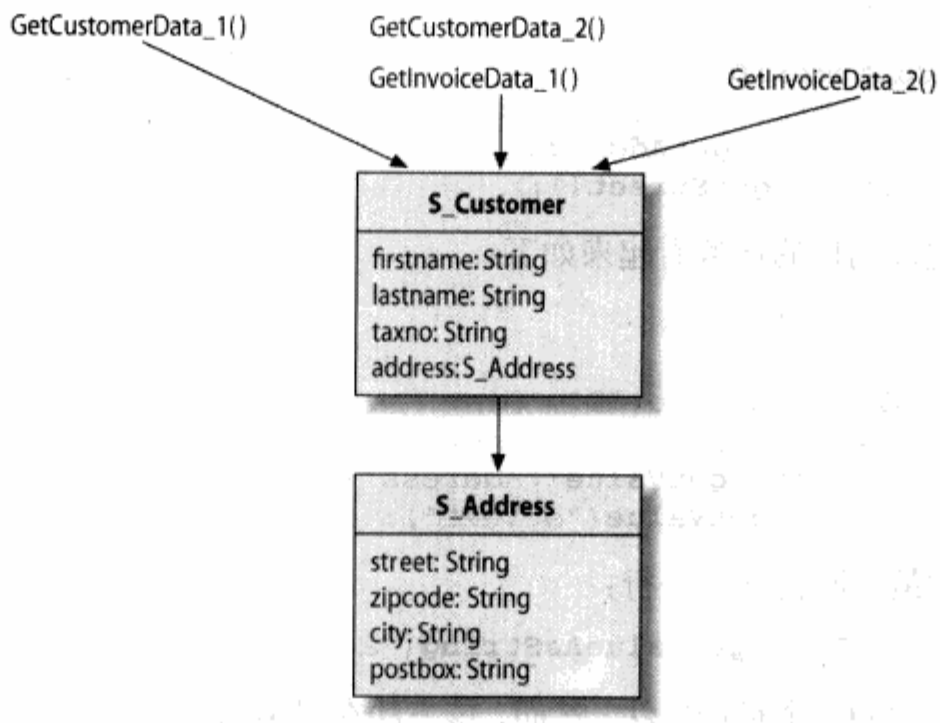


图 12-6：不同服务的不同版本共享同一个数据类型

然而，对 GetCustomerData_1()来说，taxno 和 postbox 属性都可以忽略。对 GetCustomer-Data_2()和 GetInvoiceData_1()来说，taxno 属性可以忽略。至少这些东西应该由文档记录下来（这并不容易，因为当写 GetCustomerData_1()的文档时，你不知道未来有什么属性。并且，当修改 S_Address 时，在哪些地方用了它也并非显而易见）。

另外，如果一个流程使用了一个程序库，它调用了 GetInvoiceData_1()，则它会用数据类型的老版本进行编译。后来又添加了一个新的程序库，调用的是 GetInvoiceData_2()，这个库用数据类型的新版本进行编译。这就出现了二进制不兼容的问题。结果是，由于对同样的内存有不同的解释方式，程序有缺陷，而且这是所有可能的缺陷类型中最难对付的一种缺陷。

只有当你能保证每个消费者使用的都是一致的程序库集时（例如，对协调一致的所有服务代码的发布，每次都提供不同的程序库名），这个政策才有可能是合适的。

12.3.3 使用泛型数据类型
Using Generic Data Types

处理数据类型不同版本的第 3 个选择是，只使用能处理不同数据类型的泛型代码。在这种情况下，对使用同样类型的方法来说，其二进制形式如何无关紧要。结果就是，数据类型在编译时无关紧要；所有的处理都在运行时发生。

为了看到使用泛型数据类型的差异，考虑要访问 GetCustomerData_1()返回的客户地址中的街道属性时，一个静态的 API 可能这样做：

```
S_Customer_1 custData;
S_Address_1 address;
String street;
input.setCustomerID(id);
custData = serviceAPI.getCustomerData_1(input);
address = custData.getAddress();
street = address.getStreet();
```

使用泛型 API 完成同样的任务看起来如下：

```
Data custData, address;
String street;
input.setValue("customerID", id);
custData = serviceAPI.getCustomerData_1(input);
address = custData.getValue("address");
street = address.getValue("street");
```

另一种选择是把最后两行合为一行：

```
street = custData.getValueAsString("address.street");
```

这绝对是值得考虑的一个可选方法。然而，该方法在目前似乎非常罕见。尤其是在 Web Services 上下文环境中，所有我知道的生成器（不在此点名了）都生成类型化的 API，带来了之前讨论过的版本划分问题。产生这个现象的一个原因是：没有了类型化的 API，你就丧失了编译时发现缺陷的优势。用上述的泛型方式时，属性的路径是否正确在运行时才能确定。然而，对大型系统来说，这儿讲的方法可能有好处。它之所以不常见，可能是因为类型化的接口看起来非常简单和直观（只要你没有大量应用 SOA 概念）。类型化接口的原型看起来可能很好，一旦采用，经过日积月累，再想在概念层面上改变点东西就已经为时过晚了。

12.3.4 数据类型版本划分总结
Summary of Versioning of Data Types

希望你现在已经理解了，数据类型的版本划分是大型分布式系统的一个问题。我已经讨论了 3 种选择，每种都有其优点和缺点（当然，对于所有这些选择还可以讲很多东西）。第 4 个选择是使用逐个排列参数的参数列表，不使用结构化数据类型。当然，如果使用的是粗粒度服务，来回传送的是复杂数据的话，第 4 种方法就不好用了。

正如我以前指出的，只使用泛型数据类型的方法，在实践中不常用，并且，为 SOA 系统的所有平台都提供对这种方法的支持也比较困难。让不同版本共享类型是一个非常危

险的方法，因为，如果你的流程无法保证所有程序库的一致性（在分布式系统中这比较困难）的话，该方法会导致非常恶劣的缺陷和未定义的运行时行为。另外，这种方法减弱了你在编译时发现缺陷的能力。这样说来，我提出的第一个选择是最经常使用的方法。该方法通常导致普通程序员的抱怨：这里提供的版本划分概念是如此之傻，以致对供应者来说不可能有一致的数据类型。但是，正如我们看到的，这是所有者各异的分布式系统要付出的部分代价：你没法平均化对各方面的影响，以一碗水端平的方式，简简单单地切换到新服务上。

作为服务消费者，让你的代码多多少少独立于所调用服务的版本划分是个好主意。你应该使用自己的数据类型，使用服务时，将自己的数据类型映射到服务的数据类型。因此，服务消费者通常应该用一个薄的顶层把外部数据类型映射到内部数据类型（如 5.3.3 节讨论的，服务 API 与服务协议之间有一个映射层，薄顶层可能和这个映射层结合到一起，也可能没有结合）。

然而，要小心谨慎，并且努力使事情保持简单。当你意识到处理不同版本是个问题时，再讨论用哪个方法。不要因为试图为未来可能发生的事情作准备，把事情搞得很复杂。

12.4 配置管理驱动的版本划分
Configuration-Management-Driven Versioning

正如在本章开始介绍的，说“配置管理驱动的版本划分”时，我指的是这样的要求：在不同的运行时环境中，有服务的多个修订同时处于开发之中。这个要求带来了配置管理工具的话题，比如版本控制工具。

当你有一系列的文件和其他制品属于一个整体时，必须能够给它们一个公共的标签，这样你就可以将它们作为一个总体来处理。这就是说，如果你对一个服务建模，基于这个模型生成了接口，根据这些接口实现了服务，从这些实现编译出了程序库，以及将生成的程序库作了部署，那么，你必须能找到这些属于一个整体的不同制品的相应版本。

如果所有制品都是文件，这很容易办到。你可以使用任何能够对文件打标签、支持管理文件不同版本的版本控制系统（包括显示差异，以及合并多个修改）。

如果有的制品不是文件，那你就需要相关的机制。例如，如果你有一个在数据库中实现的服务业务库（见第 17 章），那你就需要业务库的接口支持配置管理，或者有相应的组织规则（比如，对不同配置使用不同的业务库）。

再次注意，通常说来，至少能够显示制品两个版本间的差异这一点非常重要。这可能要求你有特殊的工具或脚本（比如，特殊的数据库脚本、存储过程，或者针对诸如 Rational Rose 这样的工具的 Visual Basic 脚本）。

12.5 实践中的版本划分
Versioning in Practice

重要的是要明白，需求很可能比你预料的还要变化得频繁，并且困难的修改（尤其是那些并非关键业务的）往往有被推迟的倾向。这意味着最终有比你预料的更多的版本，版本出现的频率也比你预料的高，并且，你可能发现，逐步淘汰老版本也比你预料的要困难。回忆一下我之前提过的大型 SOA 系统，在它的预期中，同样的运行时环境中每个服务最多只有 3 个版本。这个系统有超过 30 个服务参与者，超过 300 个服务，通常情况下，针对每个生产状态中的服务，其不同版本多达 6 个。

有了对这些困难的认识之后，你应该考虑如下的建议：

12.5.1 修改应该只影响供应者和消费者
Modifications Should Impact Only the Provider and Consumer(s)

努力使服务修改尽可能地容易。特别地，服务修改应该只影响服务供应者和消费者——任何其他东西都不应被影响。留心为分布式系统做架构设计的中央团队，它倾向于加上比实际需要更多的控制。

例如，你的基础设施可能在数据传输时帮你检查服务接口。然而，这就意味着每个修改也必须部署到执行运行时检查的基础设施组件上。这将导致更复杂的过程，以及潜在的瓶颈。

如此说来，针对领域特定的业务功能，SOA 基础设施内部应尽可能地通用。由服务的不同版本对任何特定处理造成的影响，必须只影响基础设施的端点（例如，针对供应者和消费者的程序库和代理）。

12.5.2 应该考虑调用约束
Call Constraints Should be Considered

我的一个客户曾经引入过一个被称为“调用约束”的概念，它是每个服务都有的一个参数，该参数给了消费者一个机会来标识可能对服务的实现产生影响的特殊上下文环境。激发该创意的源泉是如下难题：虽然服务的目的就是被多个消费者（重复）使用，然而在实践中，因为额外的属性会影响运行时（见第 13 章），所以服务的粒度通常各不相同。另外，在设计时并不总是清楚实际上会处于哪种服务上下文环境中，以及这样的上下文环境在何时会变得很关键。这就是说，你事先不知道要返回的关键属性会是哪一个。

“调用约束”的概念使你能把运行时优化推迟到需要这些优化的时候。供应者和消费者达成一致，在消费者传入的“调用约束”参数中，用一个特定的标志来表明需要一个特

殊的优化，或者那个特定的消费者需要一个特殊的行为。然后供应者就可以为这个消费者实现特殊行为，而不改变服务接口和现有的行为。

这是个有用的技术，但是，要注意一些事情：

- 这不是个消费者可以用来指出供应者是否应该返回特定属性的通用机制；它是两位参与者必须一致同意，并且在服务的语义描述（契约）中进行描述的、无格式的标志。
- 在服务实现的内部，你通常必须将这个参数作为“调用的上下文环境”的一部分进行传递。
- 这是个实用的方法，处理的是这样的事实：当服务进入集成测试或生产阶段时，之前不清楚的一些事情会逐渐明朗。使用这个标志比修改服务要容易得多。然而，不要太经常地这样做——你可以把这些标志想象成针对未来修改的权宜之计（至于没有什么东西比权宜之计更稳定这一点，大家就彼此心照不宣吧）。

关于调用约束的更多细节，见 13.3.1 节。

12.6 总结
Summary

- 针对如何迁移到新服务版本，SOA 要求一种平滑的策略。
- 最好的办法是把每个（生产中）服务的修改在技术上都当作是一个新服务。
- 如果你的服务有类型化的 API，那么，我也推荐服务类型的版本划分（虽然还有其他可选方法）。
- 为了避免版本膨胀到爆炸，你必须引入淘汰和清除老服务的过程。这是监控服务调用为什么有用的原因之一。
- 调用服务时，为了与版本划分因素无关，服务消费者通常应该有一个薄层来将外部数据类型映射到内部数据类型。
- 除了供应者和消费者外，服务修改永远不应该影响任何其他东西。
- 要做到“向前兼容”，你可以为即将出现的调用约束提供一个属性。这使消费者既可以在运行时标记自己特殊的需求，又不用修改服务签名。然而，对把所有事情都弄成通用要小心，因为这可能引入大量的复杂性和隐藏的依赖关系。

某的角色，或者那个特定的消费者需要一个特殊的行为，然后供应者就可以为这个消费者实现特殊行为，而不改变服务接口和现有的行为。

这是个有用的技术，但是，要注意一些事情：

- 这不是个消费者可以用来指出供应者是否应该返回特定属性的通用机制，它是所有参与者必须一致同意，并且在服务的语义描述（契约）中进行规范的，工作流的标志。
- 在服务实现的内部，你通常必须将这个参数作为“调用的上下文环境”的一部分进行传递。
- 这是个实用的方法，处理的是这样的事实：当服务进入某版本的生产维护时，之前不得变的一些事情会逐渐明朗。但用这个标志比修改服务要容易得多。然而，不要太经常地这样做——你可以把这些标志想象成暂时对未来版本的权宜之计（至于说有什么永久的权宜之计更稳定这一点，大家就做此心照不宣吧）。

关于调用约束的更多细节，见 13.3.1 节。

12.6 总结

- 针对如何迁移到新服务版本，SOA 要求一种平滑的方案。
- 最好的办法是把每个（生产中）服务的使命看作是一个新服务。
- 如果你的服务有类型化的 API，那么，我也推荐服务类型的版本划分（虽然还有其他可选方法）。
- 为了避免版本膨胀到爆炸，你必须计入测试和维护服务的过程。这是遗留服务慢用为什么有用的原因之一。
- 调用服务时，为了与版本划分同类无关，服务消费者通常应该有一个清晰来指示部署哪类型版本到内部数据类型。
- 除了共享管和消费者外，服务绝对永远不应该影响任何其他系统。
- 要做到“向前兼容”，你可以为响应添加新的属性或增加一个属性。这使消费者能可以在运行时检查自己需要的需求，又不用修改服务逻辑。然而，对此应有事情需要做通盘考虑，因为这可能引入大量的复杂性和隐藏的依赖关系。

第 13 章

SOA 和性能

SOA and Performance

IT 系统有两个方面总是不断地打破计划、概念和设计：性能和安全。本章我将讨论一些现实世界的例子，说明性能会如何影响 SOA 的概念和实现。大多数例子都来自一家国际电话公司，它每天的业务都伴随着成百上千服务的运行。例如，每当客户打进电话来寻求支持，甚至在客户与呼叫中心的业务代表开始谈话之前，就有几个服务被调用了。在这个环境中，性能（尤其是运行时性能）确实很重要。

13.1 性能在哪儿重要

Where Performance Matters

在服务和 SOA 基础设施的上下文环境中，性能在各种各样的地方都可以产生影响。如果性能成为关键，此时通常考虑的是服务运行时间。让我们追寻一次成功的同步服务调用来探讨这个话题，这次调用包括一个从消费者到供应者的请求消息，以及一个从供应者返回到消费者的应答消息（见图 13-1）。

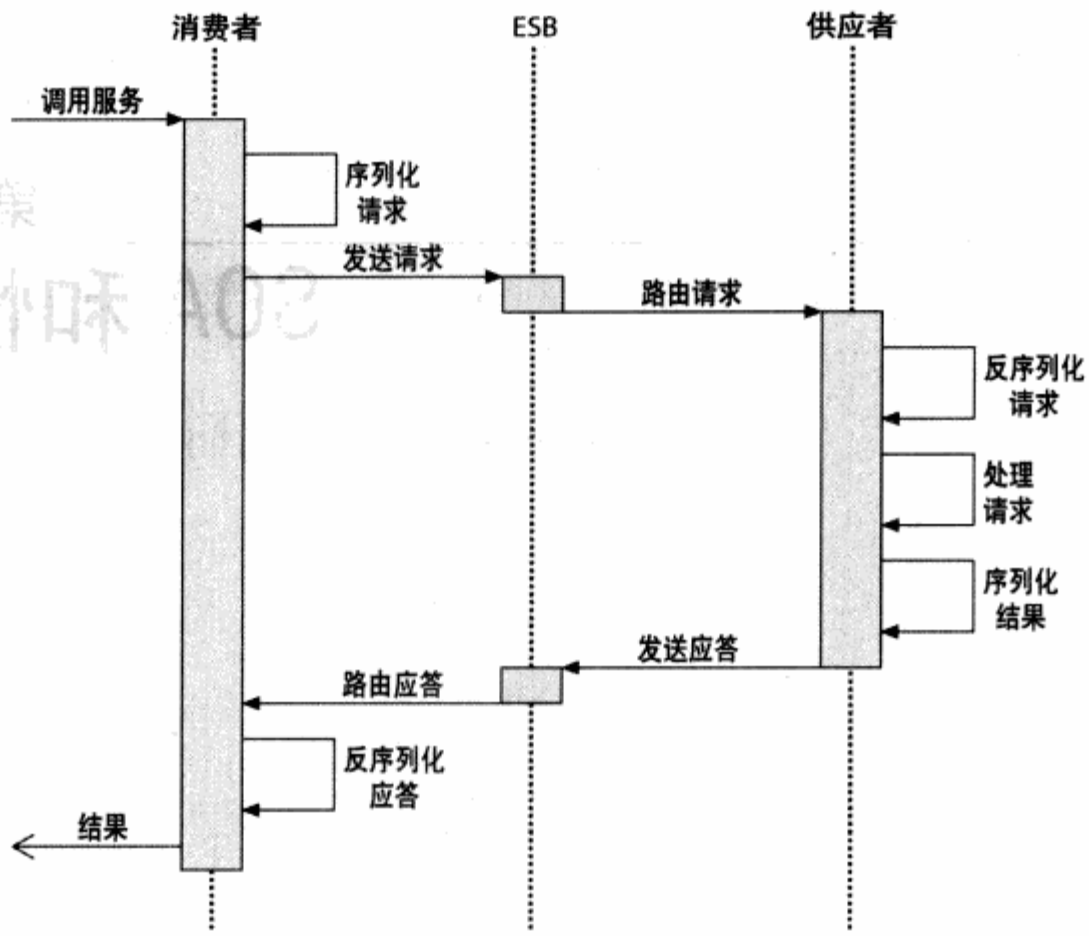


图 13-1：一个服务调用的序列图

正如这张图展示的，首先你必须将服务调用序列化成企业服务总线的协议格式。然后，通过 ESB，将请求发送给供应者。接下来，供应者必须反序列化请求以对其进行处理。对应答来说，经过的步骤也一样，只是方向相反。

以下的步骤可能是关键的：

- 通过 ESB 发送请求和应答。
- 反序列化请求或应答。
- 处理请求。

注意：SOA 是用于异质系统景观的概念的。因此，你不能简单地在系统间发送二进制数据（COBOL 汇编码与 Java 字节码不匹配，甚至来自不同 C++编译器的汇编码都不一定匹配），你需要一个中间协议，消费者和供应者能用它来交换数据。

目前，典型地是，这样的一个格式会基于 XML（比如，Web Services 基础设施用的 SOAP）。但是，XML 太啰嗦，在核心数据的基础上，其大小会按 4 到 20 的因子增长。对带宽受限的环境来说，这可能会对带宽造成严重的影响（虽然通常说来，带宽不是个问题）。

使用中间协议也可能对反序列化有性能影响，这可能（虽然通常来说不应该）是关键的，因为把一个字节序列转换成复杂数据需要时间。当然，何为可接受，何为不可接受的延迟，对此的理解依赖于你的需求和工具。因为每个额外的映射都需要时间，而且，有些反序列化器仍然可能实现得比较糟糕，所以，你应该对具体的行为进行验证，从而确保性能没有问题。

如果你必须在数据经由 ESB 传送时对其进行处理的话，则在此情况下，带宽和反序列化更容易成为问题。例如，如果你必须在不同的协议间映射数据，或者你想在 ESB 内检验服务调用是否格式良好，这都可能严重地迟滞一个服务调用的运行时。增加消息层次上的安全措施（见 14.3.2 节）也会影响运行时间。

实践中，服务运行缓慢的主要原因是，服务的实现运行耗时。这就是说，服务供应者需要时间来处理请求，提供应答。在这种情况下，你必须看看一个服务调用通常的响应时间，然后考虑这个响应时间的伸缩性有多好。如果服务供应者资源有限，或存在瓶颈，同时处理多个服务调用就可能导致严重的性能障碍。因此，一份服务契约应该包括一个服务等级协议（SLA），它不仅指明平均响应时间，还要指明在一定时间内能响应的调用的数量（这就是所谓的“时间服务系数”，TSF）。

如果一个消费者觉得服务的响应时间太长，而你又无法通过增加系统硬件和带宽来改善性能的话，通常，你必须在消费者端进行解耦。这就是说，服务消费者可能不应该同步地等待服务调用的回复。然而，要注意的是，异步通信会极大地增加系统复杂性。细节见 4.2.1 节。

另外，还要注意的是，不可靠的网络和协议也可能对系统的性能有影响。虽然你不太可能从服务 API 中看出来，然而，一个服务请求在被成功递送之前，可能在 ESB 上发送了好几次。细节见 10.4 节。

总之，收集不同服务请求的运行时统计数据，长期对这些数据进行监控是个好主意。这有助于你识别短期的问题和长期的趋势。

然而，即使性能总体上看起来还行，它仍可能成为问题。下面各节将讨论些例子。

13.2 从远程存储过程到服务

From Remote Stored Procedures to Services

即使当服务请求花费的平均时间不长，你仍然可能遇到麻烦。让我们看一个例子，它发生在一家开始引入基本服务的公司中（见 6.2 节）。

在引入基本服务之前，公司对于哪个系统在访问特定的后端已经失去了控制。过去几年公司增长过于迅猛，大家发现有几个系统已经蔓延到了整个公司，对这几个系统，目前是直接通过远程存储过程访问数据库中的数据（注 1）的（见图 13-2）。

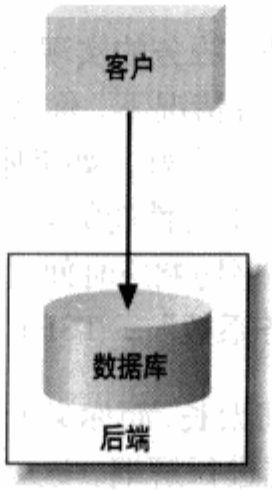


图 13-2：使用远程存储过程直接访问后端数据

结果是，维护后端的部门不可能在不打破几个其他系统的情况下修改数据库模式。因此，有益的数据库修改也变得危险和代价高昂。甚至，找出修改是否会影响及会如何影响其他系统的行为也变得越来越费劲。时不时会出现修改打破了一些系统，而被问及的部门甚至不知道这些系统的情况（至少在被问到时他们知道有这些系统了）。

靠引入基本服务，公司希望减少这些依赖性，对后端系统的技术方面重新获得控制。关键的思想是，其他系统应该使用服务，而不是直接访问数据库，这就对外部世界封装了服务实现的技术细节（见图 13-3）。

当然，从项目一开始性能就是个问题，但是，最初的测量显示出，预期中的服务调用会是足够快的。如果服务的实现没有什么重大的额外开销，一个服务调用通常耗时 100 到 300 毫秒。消费者对这样的数字通常还可以接受，因为还有图形前端在起作用（对人类用户来说，任何在半秒内给出的响应或结果都被认为具备“立即”的效果）。

注 1：存储过程是数据库脚本，它能直接在数据库的本地技术性数据上进行操作。

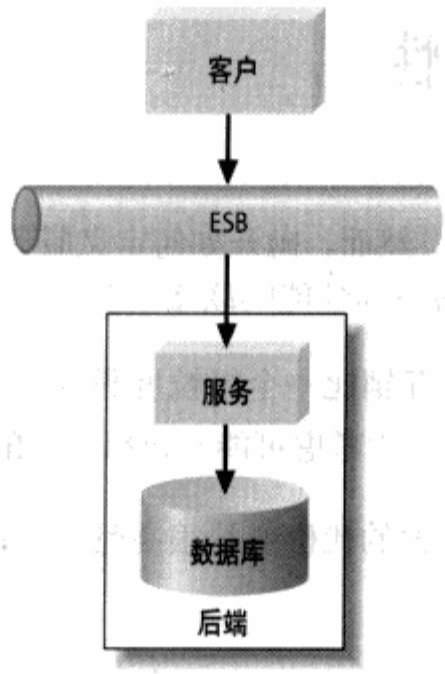


图 13-3：基本服务将技术后端细节与客户分开

这家公司开始项目时，想要强制把现有客户端全部从远程过程调用转换到服务上。但是，很快发现这行不通。问题在于，平均说来，服务调用比远程存储过程慢 5 到 10 个因数。执行一个或两个连续的服务调用，用户还察觉不了任何延迟，但是，现有客户端的实现方式有一个假设前提：执行 10 个或更多的连续远程存储过程时不能察觉任何延迟（因为它们每个只花 10 到 30 毫秒）。简单地把每个存储过程替换成服务调用引入了不可接受的延迟：10 个远程存储过程调用，每个花 30 毫秒，总共才用 300 毫秒，但是，10 个服务调用就要花将近 3 秒左右，对于前端来说，这时间太长了。

结果就是，这家公司不得不重新实现现有的客户端，其实现方式比仅仅从远程过程调用切换到服务调用要复杂得多。依靠把某几个远程过程调用结合为单独一个服务调用，以及把某几个服务请求平行发送等手段，达成了可接受的性能，但是，这些修改所花的力气大大超过最初的计划。另外，客户端代码也变得更加复杂，因为它必须能处理从不同的、平行的服务调用返回的异步应答。然而，经过这次重构后，服务供应者最终可以在不打破外部系统的情况下修改自己内部的技术数据结构了。

读了这些东西之后，你可能会认为，因为每个服务调用内部有额外的开销，拥有粗粒度的服务是个好主意。虽然大家通常都认为服务（一般说来）应该是粗粒度的，然而，在粗粒度下，性能仍然可能成为一个问题。下一节探讨这个话题。

13.3 性能和可重用性

Performance and Reusability

大家通常都推荐粗粒度的服务。然而，抛开如何定义服务的粒度何时够“粗”的问题不谈（见 3.3.2 节），这个建议被它与性能的联系所深深影响：

- 当许多细粒度服务调用的开销比一个粗粒度服务调用的开销要高得多时（正如前一节所讨论的），性能方面的考虑可能导致粗粒度的服务。
- 反过来说，当处理更多数据的代价变得很高时，性能方面的考虑可能导致细粒度的服务。

本节讨论后一个话题。

我这里想用的例子是一家公司的 CRM 系统（注 2）提供的服务，这个系统提供的最早的服务之一是返回客户所有的数据。当然，“客户所有的数据”可能包括好几百个属性——除了诸如客户 ID、相关地址和记录中的支付方式外，返回的信息还可能包括所有的合同，甚至是所有的发票和支付记录。本例中的公司是家电话公司，作为客户信息详目的一部分，返回所有电话合同。

正如本章第一节所讨论的，处理一个服务请求可能导致严重的性能障碍。实际上，装载所有客户数据可能花费大量时间（当客户是个有许多合同的好客户时，情况尤其如此）。有一天，一个新系统找到 CRM 系统，要求提供一些客户数据。本着可重用的理念，CRM 团队建议重用现有的、返回客户所有数据的服务。但是，新消费者马上抱怨起来，因为服务耗时太长。CRM 团队让新消费者标记出他们真正需要的属性，结果如图 13-4 所示——在几百个属性之中，只需要 5 个属性。

当然，比起处理一般的、包含完整客户信息详目的应答来说，处理只返回想要的属性的结果会快很多。但是，在最初的服务上下文环境中，怎样才能做到这点呢？在这幅图中，你可以看到最初服务的结果是个相当复杂的数据模型，它包含了几个不同的数据结构（记录），以及结构的数组。因为感兴趣的 5 个属性出现在不同的结构中，并非集中在一块儿，采用仅仅返回一个或几个结构而忽略其余结构的方法没什么用。换句话说，类似于增加一个输入属性，根据该属性强制服务不返回合同信息的方法解决不了问题。

注 2：客户关系管理（CRM）系统的任务是管理客户数据。

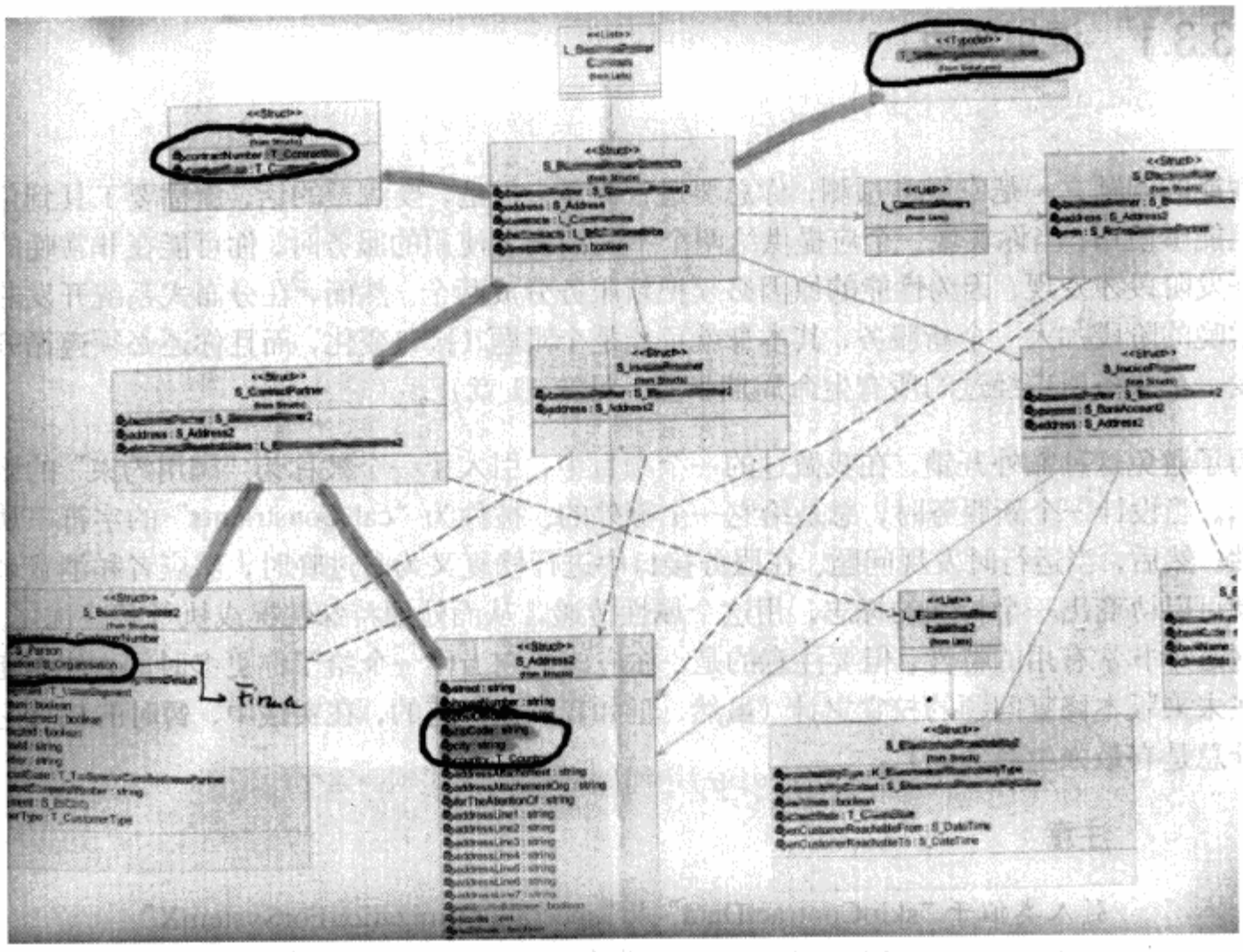


图 13-4：在粗粒度服务中标记出的选择

当我们分析不同系统是如何依附于同一个数据库时，我们看到过相同的情形。每个系统都有自己的视图，因此，我们经常对数据库引入不同的视图。看起来，我们对服务也同样需要一个“视图”概念，因为不存在诸如“服务可以提供的典型视图”这样的东西。

问题是，你如何才能在服务中处理不同的视图？对这个问题有个显而易见的答案：你可以认为每个视图都是单独的服务，因为它代表了一个特定的业务功能。这意味着，为解决我们面对的问题，必须提供一个不同的服务。

这恰恰是该公司决定要做的事情，实际上，发现问题的 CRM 团队现在有几个不同的返回客户数据的服务。少数几个是粗粒度的，其他几个是细粒度的服务，提供了对客户数据的不同视图。

这是个问题吗？这确实意味着，对服务重用这个一般性的概念而言，其工作方式和预想中的不尽一致。如果某人在一个服务中处理了某个业务案例，并且宣称这个服务将满足所有消费者的需求，那么，由于性能问题，这个业务案例很可能无法顺利工作。但是，这也并非意味着重用的目标永远无法达到。毋庸置疑，同样存在着由许多系统消费的、返回客户数据的服务。

13.3.1 调用约束
Call Constraints

性能的问题之一是它很难预测，你总要进行测量。但是，要测量的话，就需要工具到位且能够使用。当你实现一个应提供给两个不同消费者使用的服务时，你可能在非常晚的开发阶段才发现，因为性能的原因必须把新服务分成两个。然而，在分布式系统开发非常晚的阶段加入一个新服务，其本身就可能是个难题（接口变化，而且你还必须遵循开发一个服务所应经过的所有生命周期步骤，见第 11 章）。

为了避免这种额外开销，在我做过的一个项目中，引入了一个被称为“调用约束”的概念。当设计一个新服务时，总是给它一个额外的、被称为“callConstraints”的字符串属性。然后，当运行时发现问题、在服务接口中进行修复又为时过晚时，供应者和消费者就可以协商出一个特殊的标志，用这个属性传递，从而处理特殊情况或执行某些优化。这是个非常有用的特性，但要注意的是，你应该把它当作一个给了你更多时间，从而能在未来版本修复问题的权宜之计（虽然，正如我们都知道的，在实践中，暂时的权宜之计总是有最强生命力的）。

注意

引入类似于“skipContractData”或“specialOptimizationForSystemX”之类的标志，将有助于标示出这是个例外或权宜之计，有助于在供应者和消费者的源代码中找到相应的代码。

现在，你可能在想，一个更好的办法是引入特殊的表示法，这样每个消费者都可以指明自己真正需要哪个数据。这就是说，靠传递像 address.zipcode & contract[1].phonenumber 这样的东西，消费者可以指明，它只需要客户的邮政编码和第一个合同的电话号码。然而，这带来了过度的灵活性，你将不得不验证参数是否格式良好。作为一个实现者，你将不得不解析这个属性，并且，对属性的每次访问，你都必须进行这样的处理。另外，维护也将变得复杂得多，服务到底做什么将由相当复杂的格式来控制。例如，在一个配制引擎中，对这样的服务进行配置将会非常困难。

根据我的经验，作为一个经验法则，一个服务或服务的操作不应该通用到返回不同结构的程度。或者，从业务角度来看，一个服务应该执行一项具体的业务功能。通常，当服务变得通用时（并非因为业务需求——比如数据结构频繁变化——而采用通用接口），粗粒度也就到极限了。

13.3.2 定制服务

Customized Services

正如之前讨论的，不同系统通常需要对相同数据使用不同视图，当性能因素被考虑进来后，这就导致创建了不同的服务。然而，有时还需要更进一步创建**“消费者特定的”**服务。我还是用来自真实世界的一家电话公司在 CRM 系统上的经验来展示这个问题。

当一个客户把电话打入一家大型公司时，这个电话通常由一个 CTI 系统（注 3）进行处理，把电话路由到一个呼叫中心客户代表那儿。在路由过程中，客户数据被加载到呼叫中心客户代表的桌面上，这样他就能用客户的名字来表示欢迎、验证密码，等等。不幸的是，通常你不会预先知道为什么客户会打进来电话，于是，最好是客户的所有数据都可用，这使你能处理任何类型的问题或请求。因此，在谈话开始前，客户的所有数据都应加载到呼叫中心客户代表的桌面上（见图 13-5）。

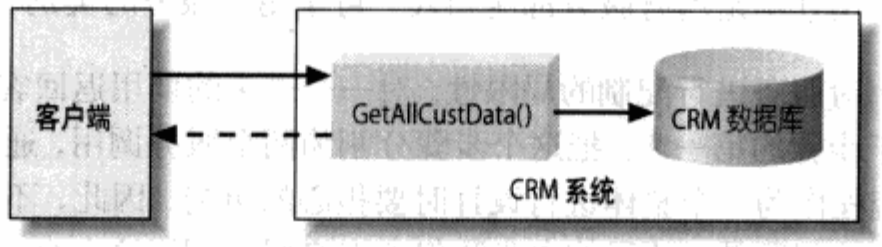


图 13-5：加载客户的所有数据需要时间

然而，加载客户的所有数据（尤其是有着大量数据的优质客户）会耗费过长时间。理想情况是，当电话还在路由时，所有数据都应该加载到呼叫中心客户代表的显示器屏幕上。这意味着客户的所有数据应该在一秒以内加载完毕。

从技术上来讲，有一种简单的方法来处理这个两难局面：分两步加载数据（见图 13-6）。这是指，当客户打入电话时，CTI 系统调用第一次服务是加载所有呼叫中心用来欢迎客户（名字、密码，等等）需要的数据。然后，在谈话开始时，发起第二个服务调用请求，加载所有额外的客户数据。作为进一步的优化，第一个请求可以在返回应答后，把所有额外数据加载到一个本地缓存中，产生一种“预加载”效果。从实际效果来看，之后的第二个调用仅仅把缓存中的数据转移到客户端（见图 13-6）。

这项技巧并非特别创新，但是，它对于 SOA 很重要，其原因在于：它展示出可以存在针对某个特定消费者所单独设计的服务，在任何其他环境中，这种对消费者所需数据的特定划分都毫无意义。实际上，在像这样的情形下，最好把消费者系统的架构师和供

注 3：计算机电话技术集成（CTI）系统允许计算机（部分地）处理电话，或将电话路由到正确的目的地，从而简化了对电话的处理。

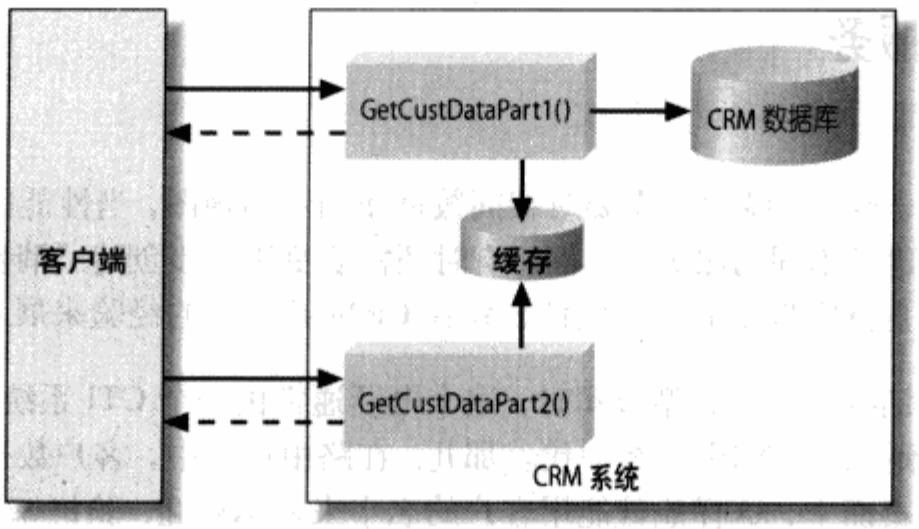


图 13-6：用两个步骤加载客户数据

应者系统的架构师召集到一起，讨论出明确的服务设计。换句话说，这是个“**定制的**”服务，它再次展示出并不是所有服务都能归入“**可重用**”服务的类别（见 3.3.6 节）。

这个例子也展示出对服务进行配制的局限性。对一个进来的调用返回客户数据，这只是整体业务流程若干步骤的第一步。把这个步骤分割为两个服务调用，通常只是个实现细节，不是当你把流程作为一个整体进行设计时要担心的事情。因此，不同的处理模型被包括了进来，重要的问题是，流程的本步骤包含许多服务调用这个事实，会在何时、以怎样的方式造成影响？

13.3.3 实践中的可重用性

Reusability in Practice

本节的例子已经展示出，有关性能的考虑可能会导致服务重用性的降低。虽然在理论上，你可能只需要一个服务来返回客户的所有数据，然而，在实践中，你很可能最终需要几个服务来提供客户数据的不同（甚至是定制的）视图。

图 13-7 显示了服务经过五年后可能拥有的消费者平均数量曲线。已实现服务对被（重）使用服务的比例，往往比你预想的要小。根据我的经验，对一个有大量处于生产中的服务的系统而言，如果在着手建立 SOA 景观一年或两年之后，每个服务平均被一个消费者所用，你就已经干得很棒了。对有三、四年历史，有成百上千服务的 SOA 景观来说，你可以预期每个服务平均被 2 到 4 个消费者所用。

对服务的平均消费者数量可能低于 1 这个事实，你可能会感到吃惊。难道每个服务不应该至少有一个消费者吗？由于 3 个主要的原因，实践中的情况通常不是这样：

- 在大型系统中，许多已完成的软件永远都进入不了生产，对于服务来说也是如此（虽然这样的情况有可能少些）。需求可能变化，实现可能被认为不切实际，或者，原来预期会对消费这些服务感兴趣的系统，现在发现它们并不感兴趣。

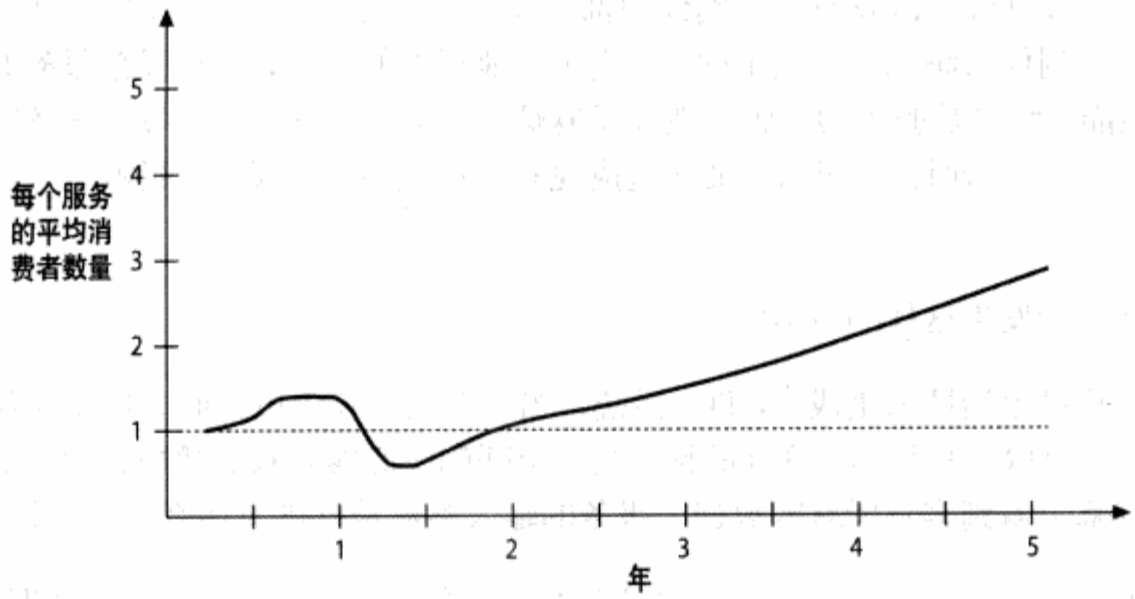


图 13-7：可能的服务（重）使用图

- 如果服务是按照业务职责管理方法提供的（见第 7 章），我们所拥有的永远都只是服务将会被消费的假设。它们可能最终被使用，或者遵照 YAGNI（“You Ain't Gonna Need It”，你不会需要它）原则，它们永远不会被使用。
- 服务可能不再被使用，因为消费者已经切换到其他服务上（从逻辑上来说是服务的新版本）。

13.4 性能和向后兼容性

Performance and Backward Compatibility

对性能的考虑也会影响被修改系统的向后兼容性。为了探讨这个问题，让我们再次考虑一家电话公司返回客户数据的服务。

几乎所有拥有成千上万或上百万客户的大型企业都把它们的客户分类到不同的组。最好的客户会获得某种特权：也许有特殊的奖金计划用于把客户绑在公司上，或者，客户可能会被路由到更有经验的呼叫中心客户代表那儿，因为这些客户的问题往往更复杂，或客户要求更好的服务。在历史上，对客户的一个典型分类法是，看他们是商业单位客户还是个人客户。但是，这种区分对本问题中谈到的这家公司不太管用，因为有的个人客户产生出高通话量，而有的商业单位客户通话量非常低。另外，这样分类的话，就不清楚该如何处理自由作家和自我雇佣的客户。

因为这些原因，公司开始引入一个更特殊的、基于每个客户产生的通话量的分类法。与通话量相匹配，每个客户被赋予一个从 1 到 10 的值，该值在一定意义上代表了客户对公司的价值。计划是用这个简单的整数属性扩展通常返回客户数据的服务，这样的话，外部系统就能处理这个新的有关客户的值。

所有分析师、设计师，以及解决方案经理都认为这种对现存服务的扩展是向后兼容的，大家没有料想到仅仅添加一个简单的、独立于其他属性的属性，会对现存服务的行为造成影响。然而，结果是服务的新版本造成了麻烦。在某些场景下，新版本服务以 2 或更大的因数增加了运行时间。新属性最终造成现有应用程序抱怨说，新版本服务没法再用了。

但是，为什么会发生这样的事情？

考虑一下计算机电话技术集成（CTI）系统。当一个用户打入电话时，CTI 系统开始一个服务请求，获取客户数据，这样的话，当一个呼叫中心客户代表接电话时，她就可以从自己的屏幕上看到客户所有的数据。服务的输入参数是打进电话客户的电话号码。

这看起来很直接，但是，由于单个客户可以拥有多个电话合同这个事实，情况复杂了。客户打某次电话用的是某个号码，而我们不能简单地看与该号码相关的电话合同中的销售数量，因为客户可能用了一个低通话量号码，但其他的号码产生了很高的通话量。因此，我们决定靠处理客户所有电话合同“**总**”通话量来对客户评级。

不幸的是，这个处理过程在运行时发生。这样一来，当服务被调用时，突然之间，我们不得不加载客户所有的电话合同来处理评级。因为加载和处理所有的合同很耗费时间，所以，服务要花比以前多得多的时间（尤其是对那些有很多合同的好客户）。

换句话说，这个简单的整数属性造成了非常高的性能障碍，因为它在运行时被处理，而且对它的处理要耗费相当长的时间。

那么，这和向后兼容性有什么关系？是啊，服务的签名保持了向后兼容，但是，契约已经被破坏了，这是指该服务的非功能性的服务等级协议（例如，等级协议里规定了所有电话的 90%应该在小于 500 毫秒内被响应）被破坏了。

现在，你可能提反对意见，说这是个糟糕的例子，因为它只展现给我们一件事情：低劣的设计带来的永远是麻烦。当然，在运行时处理这个简单的属性不是个好主意，并且我们事先就应该知道这点。显然，另一个可选办法是启动一个批处理工作，每天晚上处理客户的当前值，当第二天客户数据被请求时，返回这些值。

然而，这个错误并不是那样不可理解，解决方法也并不是那样显而易见。这个例子展示出针对 SOA 和大型分布式系统的一些重要教训：

- 首先，在大型的、复杂的系统中，不可能了解和预料所有事情。不同人、部门和系统的牵涉和交织模糊了原本明显的事情。

- 其次，没有一个解决方案能被证明是魔术子弹。针对本例，在晚上处理客户值的方法不会真正好到哪儿去，因为比起每天在运行时处理成千上万的客户值（即只处理那些打进电话来的客户的值）来，每天晚上处理成百万的客户值将需要多得多的资源。

- 再次，即使当你认识到了问题，并且认定采用不同的解决方案（比如每天晚上处理客户值）会更好些时，在实现解决方案时也可能遇到麻烦。又一次，麻烦的原因是不同的领域中各异的所有者：负责维护数据库，包括为此问题提供批处理工作的人员或部门，可能不同于负责提供客户服务的人员或部门。为解决这个设计问题，在不同部门间重新划分资源优先级马上就会被证明是件棘手的事（即使在原则上合作是可行的）。

注意：在这个特别的场景中，哪个解决方案正确无关紧要。我在这儿想要展示的是，性能对服务和系统设计有影响。从业务流程的角度来看，客户值如何被处理是个技术细节，但是，这个细节却可能破坏整个业务流程。在普遍的意义上，这个例子展示出，对于把分布式流程分解成不同步骤再实现这些步骤的设计方法，如果你认为该方法就足够了时，一些风险已经隐匿于其中了。

还要注意，当前组合（配制）服务的标准方法，如 BPEL（见第 7 章），都没有处理非功能特性。这并非意味着整个方法都起不了任何作用，但是，你应该知道这些方法的局限性。

作为旁注，我想提一下此处的经验和服务版本划分概念的重要关系。大家发现，针对生产中服务的修改，从技术角度来看，如果版本划分政策将每次修改都定义为新服务，将有助于最小化此处的问题产生的影响。只要现有业务流程不会被服务新版本带来的问题（如果存在某种迫使消费者切换到现有服务新版本的机制，情况就会如此）破坏掉，那么消费者总是能够选择使用哪个版本。细节见 12.2.1 节。

13.5 总结

Summary

- 对服务而言，性能（尤其是运行时）是个问题。
- 从专有的远程数据访问切换到服务，可能对运行时有严重的影响。
- 在可重用性和性能之间，可能存在一种取舍关系。
- 性能可能对服务的粒度有影响，其影响在两个方向上都存在：如果多个服务调用的开销过高，服务就倾向于变得更粗粒度；如果处理非必要数据的开销过高，服务就倾向于更细粒度些。另外，当服务变成了普适的银弹时，就应该认为粗粒度已经到达极限了。

- 调用约束可能有助于在不打破服务正式接口（签名）的情况下（暂时）实现特殊的服务行为。
- 非功能特性（SLA）是服务契约的一部分，因此，导致运行时障碍的附加属性可能会破坏服务的向后兼容性。
- 性能考量可能会破坏业务流程和系统设计，当使用忽视非功能特性的业务流程建模标准，如 BPEL 时，这就引入了某些风险。

第 14 章

SOA 和安全

SOA and Security

当集成分布式系统时，安全问题迟早会凸现出来。因为许多人都能访问系统景观，但不是所有人都被允许看到和操纵所有数据，所以出现了问题。

本章对 SOA 的安全考量给出一个太致的描述（注 1）。

14.1 安全需求

Security Requirements

谈论分布式系统的安全涉及到许多不同的方面。和平常一样，对这些东西进行分类有不同的方法。一般说来，如下的分类比较关键：

认证

认证与检验身份有关。一个身份可能是一个用户、一个物理设备，或者一个外部的服务请求者。对 SOA 来说，这意味着找出谁在调用服务。

注 1：感谢 OPTIMA（*http://www.optimabit.com*）的 Bruce Sams。他是分布式系统（包括 SOA 和 Web Services 系统景观）安全方面最重要的专家之一。在写作本章时，他告诉了我许多东西。

授权

授权与决定一个身份被允许干什么相关。对 SOA 来说，这意味着检查调用者是否被允许调用服务，以及/或者是否被允许看到结果。

机密性

数据在传输或存储时是否能保持机密性，是安全的另一个关键方面。对服务而言，则意味着要保证当服务数据在供应者和消费者之间传递时，除了服务调用者之外，没有谁能看到服务数据。

完整性

这儿的关键是保证数据不可能被篡改或伪造，篡改和伪造或者造成数据出错，或者甚至更糟，造成认证和授权凭证被伪造，以致某人可以访问他本不应该看到的数据。

可用性

可以按如下方式攻击（“泛滥”）一个系统：虽然数据并没有丢失或被破坏，然而，系统就是变得没法响应了。泛滥的一个典型形式是“拒绝服务”（DoS）攻击。

记账

这儿的关键是保持消费资源的记录。对 SOA 来说，这意味着对用于管理、计划、计费，以及其他目的的服务调用进行跟踪。

审计

这儿的关键是评估某个安全概念及其应用，目的是改善它的可靠性。审计可能包括记录所有安全相关的信息，这样你就能察觉或分析安全漏洞或攻击。所以，它也包括对所有安全相关的数据流进行监测、日志记录，以及跟踪。另外，审计也可能是个功能组件：当服务在操纵数据时，这种操纵必须被“审计”（例如，为了法律上的原因）。

在谈及安全的上下文环境中，“AAA”通常用来指认证、授权和记账。

14.2 处理安全需求

Dealing with Security Requirements

处理安全不是什么新鲜事物，对于我已经列出来的所有方面，SOA 使用的方法和分布式系统使用的典型方法一样。

- 对认证和授权，通常你需要用户 ID 这个概念和密码（虽然还有其他认证标准，如 Kerberos、证书、硬件令牌，等等）。说起用户 ID，它一般都有某种形式的间接性，这样用户就可以被授予角色；类似于是否能调用一个服务，或者看到一个结果的

权限，都与这些角色相关联。如果有一个中央服务在管理用户和用户配置文件的话，这个服务通常被称为身份供应者（关于这个话题的更多内容见 14.3.3 节）。

- 就机密性和完整性而言，使用的是通常的概念，如加密和数字签名。

对这些概念来说，SOA 安全与其他任何形式的分布式计算的安全并无二致。然而，SOA 有些方面比较特殊，我将在以下各节讨论。

14.2.1 互操作性对比安全
Interoperability Versus Security

SOA 的一个关键概念是高互操作性（见 2.3.2 节）。遵循企业应用集成（EAI）的概念，对其他系统的连接应该是简单易行的。在 SOA 中，基本思路是用公共的 ESB 替换针对每对系统的、单独的连接性解决方案，如此一来，任何连接到这个 ESB 的系统就和任何其他同样连接到 ESB 上的系统互相连接了。

该方法的一个结果是，靠使用不同通信渠道和不同协议所形成的“自然的”防火墙，现在不存在了。当连接到一个 ESB 上时，默认情况下，你的系统是开放的。Bruce Sams 和我分享了一个城堡的比喻：在古代，我们有许多城墙、护城河、弓箭和热沥青来保护城堡，但是，高互操作性的要求是吊桥保持放下状态。

结果是，为了保护敏感数据，我们不得不限制消费者调用所有服务和看到所有结果的能力。通常情况下，这是将安全概念引入 ESB 的动机。

14.2.2 异质和安全
Heterogeneity and Security

SOA 是个处理分布到不同异质系统上的业务流程的概念。在这些系统上，现有的安全机制和政策可能不尽相同。如此一来，你面临的挑战是：引入一个通用的安全概念，覆盖这许多现有的、不同的安全概念。这个过程开始于同一个人有不同的用户 ID，最终结果是使用不同的抽象和过程来引入角色和用户配置文件。

14.2.3 分布式过程和多层抽象
Distributed Processes and Many Layers of Abstraction

SOA 导致的一个最严重的问题是产生了许多不同的抽象层。当每个服务从下层抽象出业务功能时，它也必须从下层应用中同时抽象出用户身份的上下文环境。

与单个后端的异质安全概念相结合之后，就产生了这样的事实：从最初对业务流程的请求，到最后处理这个请求的系统，中间有长长的一段路。

图 14-1 显示了一个可能的例子：某客户可能使用服务门户开始一个流程，该流程可能途经不同的层（流程服务、组合服务、基本服务）和不同后端。另外，一个呼叫中心或内勤办公代理也可能被牵扯进来，执行流程中的某些步骤。

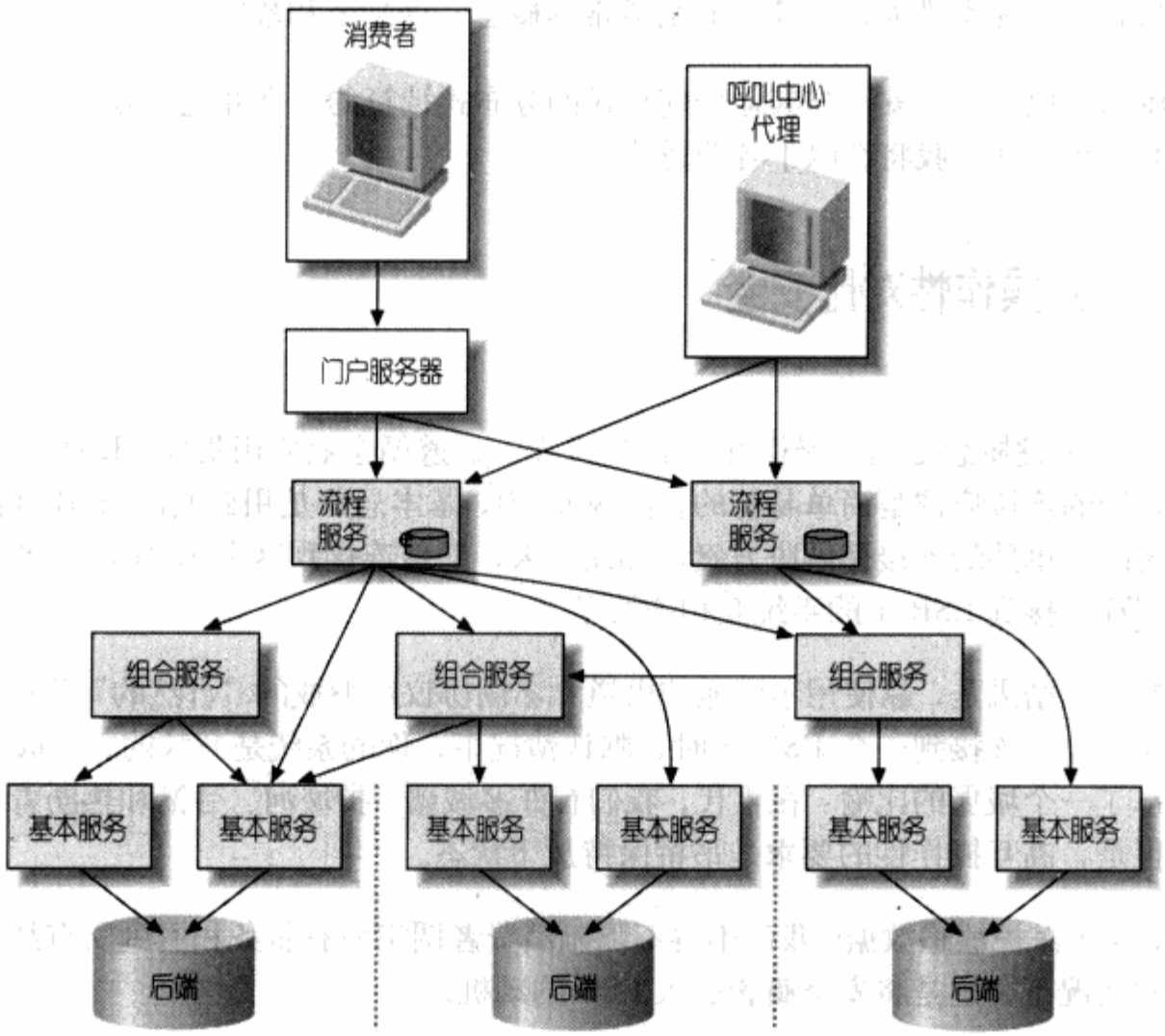


图 14-1：在最终用户和后端之间的许多中继

有许多层带来了两个重要问题：

- 不清楚是哪个系统在对用户进行认证和授权。
- 机密性不光必须在一个连接上得到保证，更要在中间涉及多个节点的多个连接上得到保证。

让我们先讨论为什么不清楚是哪个系统在认证用户和对他的活动授权。这是后端、前端，还是两端的责任？一方面，后端想要决定哪个用户被允许执行某些功能，以及取得某些数据。因为每个后端都应该为自己的数据负责，并且通常不是所有用户都被允许执行所有修改，或看到所有数据。然而，另一方面，前端也想拥有对用户授权的能力，这样才能提供总体上的一致性，避免用户发起一些其中包含着他们不被允许调用的服务的业务流程。

为了兼顾这两方面的利益，不管前端还是后端都必须共享同样的身份（比如用户 ID）及相关的配置文件和角色。这暗示着身份和/或配置文件必须传递给后端。

现在我们看第二个问题，该问题和机密性及完整性有关。如果在前端与后端间涉及了多个系统，那么，只保证两个系统间物理连接的安全是不够的。当你有多个连接，中间涉及到多个节点时，你需要保证特定数据端到端的安全性。例如，如果你传递诸如密码、信用卡号，或者账户细节及限额这类数据，通常说来，只有前端和后端应该被允许看到这些信息。任何处理这种数据的流程都能传递它，但是不应该真正看到它。

因此，处理传输层的安全机制通常不够。实际上，互联网通常的方法——安全套接字层（SSL），该方法对 Web Services 也可用——是不够的。它提供了点到点的安全，这与涉及多层时所需的端到端安全不同。结果就是，通常说来，你需要消息层安全，它意味着在来回传送的消息内部处理安全问题。传输层安全和消息层安全在 14.3.2 节讨论。

14.2.4 多客户端能力
Multiclient Capabilities

你可能已经注意到了，我总是这样写：用户被允许调用一个服务“**和/或看到结果**”。你也许会奇怪：为什么我不直接说“被允许调用一个服务”。这是因为分布式系统通常有多客户端能力。换句话说，它们使用相同的后端来为多个客户端管理数据，并且，某个特定客户端访问了另一个客户端数据的事情，一定要杜绝发生。

例如，一个数据处理中心可能管理着几家不同银行的数据。一般说来，银行被允许调用同样的服务，但是，它们只应该被允许看到自己的客户的数据。例如，如果服务返回与特定银行账户相关联的客户数据，只有账户号标识为某家银行的客户时，该调用银行才能看到结果。如果使用了另一家银行的账户号，调用银行不应该看到任何客户数据。

你可能会争论说，这个问题很容易解决：银行服务只需要简单地要求一家银行只能请求有其自身银行代码的数据就行了，而且，这个原则可以简单地应用到所有被调用的服务上。然而，这个方法不是在所有情况下都管用。例如，对手机公司来说，只有少数几个网络供应商，但是，有非常多卖手机合同的公司。卖入网合同的公司被称为“服务提供者”，与这些服务提供者有关的某些数据必须保密，但是，某些数据又必须能被网络中

所有公司的系统访问。例如，因为允许最终用户将手机号从一个服务提供者转移到另一个，所以，限制特定服务提供者访问与手机号码相关联的客户数据就不合适。后端不可能事先知道客户和哪个服务提供者相关联，因此，后端需要能够被与该客户所在网络有关系的所有公司访问。

因此，当涉及到多客户端能力时，你可能需要执行运行时安全检查。这就是说，针对特定系统，静态地允许或拒绝特定服务通常说来是不够的。

14.3 实践中的 SOA 安全
SOA Security in Practice

在实践中，由于下列几个原因，安全被忽视的情况相当常见：

- 安全需要付出努力。
- 不可能达到绝对安全（除非将分布式系统的连接断开）。
- 你可能假设，用于互联网的、通常的安全机制（防火墙和诸如 SSL 这样的特殊协议）就足够了。
- 你可能假设，SOA 的基础设施通常提供了足够的安全性。
- 对基础设施团队或业务团队来说，它们不清楚安全是否是个问题。

以下各节将直接地或间接地讨论这些话题。

14.3.1 基础设施没有提供足够的安全性
Infrastructures Don't Provide Sufficient Security

一般说来，你不应该假设基础设施（互联网、Web Services 或任何其他中间件）处理安全的方式已经很好，因此你可以不必再考虑这个问题。

第一个问题是可能缺乏概念级的支持。例如，基础的 Web Services 协议没有处理安全性，它被设计用于提供连接性。正如[PulierTaylor06]对 Web Services 标准所宣称的：

> 新标准被开发的时候没有安全观念……这些公开标准（XML、SOAP、WSDL，以及 UDDI）中，没有一个自身就包含了内生的安全性考虑。如果单独使用这些标准，它们是完全不安全的。实际上，Web Services 就是被设计用来穿过防火墙高效地传输数据的。

同样，到目前为止，诸如 BPEL 这样的流程建模标准在组合服务时，也没有概念可用于组合与聚合安全概念。

另外，即使一个基础设施提供了某些安全支持，你也必须考虑它是否覆盖了安全的所有方面（授权、认证、机密性、完整性、可用性、记账和审计），以及支持的质量如何。

如果安全是个问题（它通常是个问题），你就总是必须显式地处理它。如果一个测试显示出你的基础设施提供的安全解决方案就足够了，那很好；然而，不要假设情况就是这样。

14.3.2 处理机密性和完整性
Dealing with Confidentiality and Integrity

在实践中，要处理服务所传输数据的机密性和完整性，你既可以在传输层，也可以在消息层上进行。

传输层安全

在这儿，你使用基础设施的底层协议来引入安全性。一个典型的例子是通过 SSL 加密 Web Services 调用（这就是说，使用 HTTPS 协议而非 HTTP 协议）。

虽然这个方法通常比较容易，代价也不高，然而，问题在于它只对数据“从系统到系统”（或节点到节点）传输这一种情况有用。也就是说，这只是点到点加密，而非端到端加密。在节点内部，以及不同层之间，数据仍然是可读的。

消息层安全

在这里，你在传送的实际消息内部引入安全性。这是指以如下方式使用基础设施的协议：没有谁能不被察觉地读取或修改消息。为使用这种方法，你必须对消息格式定义某种特殊限制，这样一来，所有端点才能彼此之间通信。消息发送者可以对所有数据或部分数据进行加密或认证，而接收者则解密消息或核对证书。一个典型的例子是，用诸如 WS-Security 这样的标准，靠某些额外的属性，对 Web Services 进行加密。

注意：你仍然必须能将消息来回传送。因此，通常你只对消息的业务数据（消息的有效载荷）加密。任何与在基础设施内传送消息有关的信息，应该与有效载荷分开，能够被单独处理的方式包括进来。典型地，这样的信息位于消息头中（见 15.4 节）。

正如之前讨论的，消息层安全通常更可取，因为它带来端到端的安全性，而传输层安全带来的仅仅是点到点安全性。然而，消息层安全的不足之处是，对传输层的支持往往更好些，其性能也更佳。除此之外，维护、发布证书，以及对证书签名都可能成为对组织的挑战。

还要注意，在 ESB 内部，不管是在传输层还是消息层，不提供任何安全性也是可能的。这是否意味着你没有任何安全性呢？不一定。这样一来，安全性就成了业务层的事情。这意味着不管供应者还是消费者都必须在业务 API 中处理安全方面的问题。这就是说，你要定义一些服务参数，用于交换令牌或加密后的数据（当然，在服务契约中必须指明确切的格式）。这个方法的问题是，为了满足同一个需求，许许多多的参与方都必须付出努力。

在任何情况下，服务的所有参与者都必须清楚如何处理安全性。这是个说明如下事实的好例子：SOA 的基础设施和架构必须互相吻合。只提供基础设施，把安全性问题完全留给参与者去考虑，是非常糟糕和危险的做法。

14.3.3 作为服务的安全
Security as a Service

因为安全对各方各面都有影响，所以，通常需要有一个总体策略，指导如何以成体系的方式来处理安全问题。否则，迟早有一天你会失去控制。

成体系的认证和授权要求你分开处理两件事情：对身份认证的决策和对授权的执行。这就是说，你必须区分“**政策决策点**”（PDP）——作出决策的地方（在[ISO10181-3]中，这被称为一个“访问控制决策功能”或 ADF），以及“**政策执行点**”（PEP）——决策得以执行的地方（在[ISO10181-3]中，这被称为一个“访问控制执行功能”或 AEF）。

如此划分类型使你能集中定义可以作出哪类决策，保证这些决策在整个系统范围内一致。在本地应用中，无论何时，只要需要作出安全决策，实际上发生的都是 PEP 向 PDP 请求一个决策。PDP 是这样的服务的一个绝佳例子：由不同消费者调用，决定某事是否可用或被允许（注 2）。

换句话说，处理 SOA 安全性的最好办法是将安全作为服务提供。例如，[HintonHondo-Hutchison05]引入了“安全执行服务”，它从 ESB 或一个应用（或任何处理数据的地方）的内部调用一个“安全决策服务”（见图 14-2）。

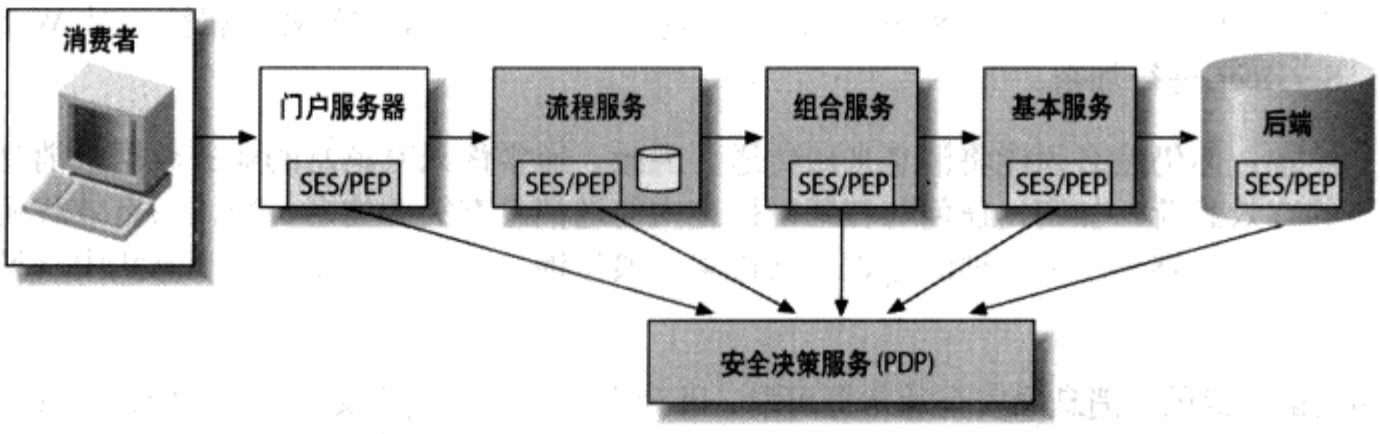


图 14-2：作为服务的安全

不管你用什么样的术语，背后的想法是引入某些管理身份及相关角色和配置文件的中央组件或系统。这通常被称为“**身份供应者**”（IdP）。正如之前讨论的，如果它提供了

注 2：在 PDP 和 PEP 之间作出划分，是处理一致的、公共的行为的通用概念。另一个例子见 9.2.2 节。

作出安全决策的服务，那它就扮演了安全决策点、政策决策点，或者访问控制决策功能的角色。

IdP 是身份及其配置文件的中央管理者。由于中央安全服务器引入了瓶颈，因此，从技术上和组织上都需要用些特殊的技术来处理这个问题。通常情况下，你需要针对本地 IdP 或服务的缓存，如此用户认证和授权才能伸缩。

然而，在实践中，不是永远都可能集成分布式流程涉及的所有系统。因此，相当常见的是使用技术性用户，它们是 IT 形式的身份保证人或身份担保人。另一个方法是为不同后端提供单点登录，使身份供应者能够在不同身份间相互映射（这被称为“**身份联盟**”）。最后，在不同的、本地的用户配置文件间进行切换也是可能的。

注意

自由联盟计划的目标是在全球范围内为用户/人们，以及其他的身份引入中央 IdP，它可以被各家公司用来提供单点登录，在全世界范围内共享用户配置文件。细节见[ProjectLiberty]。

14.3.4 安全、性能和状态
Security, Performance, and State

安全有成本。调用额外的功能或服务来做决策，加密和解密数据，检验完整性，以及检查可用性都要花时间、费资源。然而，通常说来，除非愿意让你的系统承受风险，你没有其他什么选择。本质上，要处理安全，你就不得不在风险和性能间找到适当的平衡。

因为安全影响了性能，同样地，它也就可能对服务的状态（见 15.1 节）造成影响。从业务的观点来看，你可以说服务是无状态的。然而，对每一个服务调用来说，你必须就调用者是否被允许执行调用和/或看到结果找到答案。为做出这一决策，你需要些数据，而这些数据通常并不和每个服务调用一起传输。结果就是，一个后端可能拿着用于身份识别的输入数据，用它来加载用户配置文件和/或与身份数据相关联的角色。这会花些时间。更进一步，如果服务被路由到不同的供应者系统，你就必须对每个调用，或至少是进入每个物理供应者系统的第一次调用加载配置文件和/或角色。配置文件和/或角色可以被缓存起来，这样的话，如果请求被再次路由到同样的物理系统，数据就只需要在第一次调用时被加载。此时，系统可以返回一个与每次后续请求一起使用的标志。然而，这样做之后，被缓存的数据在多长时间内保持有效的问题将凸现出来（也就是说，屏蔽及对配置文件的修改能以多快的速度分布出去）。

注意：处理状态和服务有多种不同的方法。例如，你可以只在后端处理状态，在第一次服务调用时返回一个会话 ID，再将此 ID 与所有后续调用一起发送。细节见 15.1.2 节。

14.3.5 现实中的安全
Security in Reality

如果我必须描述一下我见过的现实中的安全做法的话，我所能说的就是它们是异质的：它们包括了一些中央用户管理、一些技术性用户、一些针对不同层的加密，如此等等。对不同安全概念进行草率的混合、身份供应者、技术性用户，以及其他安全解决方案，所有这些当前都在用。

除此之外，经常地，至少是部分地，没有什么安全可言。从概念上来讲，这是可能的，并且有时候也有其意义。例如，考虑一个非军事区（DMZ），这是每个系统都可以信任其他所有系统，只在数据跨越边界时安全才发挥作用的一个区域。这意味着所有在该区域中的后端都信任所有处于边界上的应用，这些应用表现得像公司的防火墙（比如，门户服务器，胖呼叫中心应用，等等）。当然，这也意味着所有这些跨越边界的应用必须共享公共的安全政策，并正确地实施政策。

例如，可以这样做：不是在每个后端处理多客户端能力问题，而是所有在 DMZ 中的消费者都被允许获取所有数据，然后，消费者自行保证在 DMZ 外，它们只可以看到和自己的域或公司相关的数据。不幸的是，如此一来，服务供应者（作为对数据负有责任的所有者）实际上就不能保证关于数据访问的业务规则，因为这些规则的实现需要所有服务消费者的共同协作。另外，即使这不会使审计不可能，也会使它相当复杂。

我曾看到过对非常敏感的数据使用 DMZ 方法。例如，假设一个服务向每个消费者返回客户数据。因为高层经理或 VIP 的数据应该只对少数特殊的人可见，所以，这个方法有风险。为了减少风险，在这样的情形下，我看到过一个政策靠给这些客户起假名字来“加密”数据本身，并且，把假名字映射到真名字的列表只给了少数几个人。就在业务层上处理安全性而言，这是个有趣的例子。

对于审计我写得不多，但是，就安全性来说，审计扮演着有趣的角色。只要你能够跟踪和关联数据流的流动，从而在事后找出一个安全攻击或一个安全漏洞是谁造成的，原因何在，那么，会被抓住的威胁将可能有助于提高系统的安全性。换句话说，靠记录日志、监测和审计消息传输，你就能鼓励程序员对安全方面更负责任或更加警觉，从而阻吓违犯者进行可能的安全侵犯。如果你无法避免所有的安全侵犯，你也许至少能够在侵犯发生时将其曝光。完全依靠这个方法是否风险过高，那是你的决策。

14.4 XML 和 Web Services 的安全
Security with XML and Web Services

接下来，我想针对 XML 和 Web Services 的安全给你一些提示。注意：第 16 章详细讨论了 Web Services，开始读本节之前，也许你会想先读读那一章。

原则上，你可以使用如下不同类型的标准：

- 通用安全标准。
- XML 安全标准。
- Web Services 安全标准。

图 14-3 图示了各可选项的更多细节。

图 14-3：XML 和 Web Services 的安全栈

通用安全标准包括大家熟知的算法，比如 RSA、AES 和 DES，以及基本的、针对加密和安全会话的安全标准，如 SSL、Kerberos，等等。也有处理 XML 文件的特殊标准。它们的优点是可以读取和写入 XML 文件，这样的话，加密或签名的结果就能用通常的 XML 处理链进行处理。最后，在图的最上方有基于 XML 的通用安全标准，如 SAML，以及特别关注 Web Services 的标准，如 WS-安全。

让我们简单地讨论一些最重要的标准。

14.4.1 SAML

一个重要的通用标准是由 OASIS 维护的安全断言标记语言（SAML）。SAML 是基于 XML 的语言，负责在不同系统间管理和交换安全信息。它使一方能够就一个**主体**（其身份通常是一个人类用户，但也可能是一个技术性用户、系统或公司）作出安全信息断言。除了主体的身份之外，其他属性也可以被断言，如客户的电子邮件地址，或用户的角色。例如，一个断言可能如此陈述“这是 Nicolai Josuttis，电子邮件地址为 *nicolai.josuttis@somedomain.de*，他被允许修改本书的内容”。

在分布式环境中，断言可以被管理和交换。例如，协议允许一个身份供应者标识主体、四处传递断言、在不同身份间进行映射，以及执行分布式退出登录（也称为单点退出或

SLO）。结果就是，你可以在分布式流程上使用（不同的）身份供应商，管理和结合不同的用户配置文件。这包括对单点登录的支持（SSO）。

细节见[SAML]。

14.4.2 XML 和 Web Services 安全标准

XML and Web Services Security Standards

有几个特定的标准用于 XML 和 Web Services。这些标准并没有引入新的安全技术或过程，而是定义了在通过 Web Services，用 XML 文件交换数据时，如何应用现有的技术和过程。

例如，对 XML 来说存在着如下标准：

XML 签名（*XML DSig*）

允许 XML 文档被签名，保证其数据的完整性（和真实性）。这和 PGP 相似，然而，数据采用 XML 格式。细节见[XMLSignature]。

XML 加密（*XML Enc*）

允许 XML 文档被（也许是部分地）加密。例如，你可以加密整个 XML 文档，或仅仅加密文档中的一些数据，如信用卡号码或“除了用户名的所有东西”。细节见[XMLEncryption]。

XML 密钥管理

针对 XML 签名标准支持密钥管理。细节见[XMLKeyMan]。

对 Web Services 来说，最重要的标准是：

WS-安全

针对 Web Services（SOAP）的授权、完整性和私密性，定义了如何应用不同的安全技术。它描述了一个标准的方法，用于将类似于令牌、加密、签名、SAML、Kerberos 等安全信息嵌入 SOAP 头部的标准方法。细节见[WS-Security]。

WS-安全政策

使你能够找出一个服务供应者要求和/或支持哪个安全标准。一个供应者可以在 WSDL 或 UDDI 中，指定接受 X.509 和 Kerberos 协议（推荐使用 Kerberos 协议），同时要求有某些特定的签名，这些都是 Web Services 定义的一部分。细节见[WS-SecurityPolciy]。

WS-信任

使你能发布、更新和检验安全令牌。细节见[WS-Trust]。

WS-安全对话

在多个消息交换间建立一个共享的安全上下文环境。这样做的主要动机是，通过减少向 IdP 发送耗时的请求的数量，从而改善性能。细节见[WS-SecureConversation]。

WS-联盟

定义了若干机制，通过允许身份、属性和认证的托拉斯，并且在这些托拉斯之间进行中转，使得在不同的安全领域进行集成和建立联盟成为可能。细节见[WS-Federation]。

正如 Web Services 的情况一样，这些标准并不一定带来互操作性，因为有太多的版本，太多的方法来诠释这些标准。因此，和 WS-I 基本配套资料（见 16.3.1 节）相似，有一个 WS-I 基本安全配套资料，它将使用限制在某些安全标准版本上，从而保证了互操作性。细节见[WSI-SecurityProfile]。

要想获得更多信息，请见[KannegantiChodavarapu07]和[NewcomerLomow05]。

14.4.3 XML 和 Web Services 攻击
XML and Web Services Attacks

在所有的分布式系统中都存在来自外部攻击的风险，比如洪水（DoS）攻击。非常多的、合法的或非法的请求，能够使一个服务供应者崩溃。当恶意攻击发动时，你需要能察觉这些攻击并作出相应反应。然而，针对 XML 和 Web Services 还可能有其他类型的攻击（注 3）。让我们来看些例子（这些例子都非常具体，为的是给你这样的印象：这些事情是多么容易发生，在你的日常工作中，这些问题是多么的现实）：

XML 炸弹

XML 是一种原始递归语言，在某些情况下，XML 解析器甚至可能把非常小的 XML 文档扩展为巨大的文档。考虑如下的例子：

```
<?xml version="1.0"?>
<!DOCTYPE xmlbomb [
<!ELEMENT data (#PCDATA)>
<!ENTITY a "&b;&b;&b; ">
<!ENTITY b "&c;&c;&c; ">
<!ENTITY c "&d;&d;&d; ">
<!ENTITY d "foo ">
]>
<data>&a;</data>
```

问题在于，这儿被引用的实体 a 扩展成了许多其他的实体。在此，这导致了 3^3（一个实体变成 3 个实体后，每个再扩展为 3 个其他实体），在浏览器中看起来可能像下面一样：

```
<data>foo foo foo foo foo foo foo foo foo foo foo foo foo foo foo foo foo foo foo
foo foo foo foo foo foo foo foo</data>
```

现在，假设你有 10 个扩展，每个都扩展为 10 个实体。这并不需要多少代码，但是，

注 3：感谢 Bruce Sams 提供这些例子。

它导致 XML 解析器试图将其扩展为包含 10 000 000 000 个 foo 的序列，这会占用大量的时间和内存。通常情况下，碰到这个问题的进程直接就崩溃了。

注意，这是一类无法由防火墙察觉的拒绝服务攻击，因为问题不是出在消息的数量上，而是出在内容上。

如果对你的应用来说这是个风险，而且可用性对你来说很重要，你可能需要一些办法来限制 XML 解析器，或者在解析之前就发现这类情形。

XPath 注入

XPath 注入是一种类型的代码注入。正如 SQL 注入一样，这儿的手段是对“直接将用户输入映射到代码”这种实现技巧进行不当运用，如此一来就改变了代码最初的意思。

让我们先看一个展示了代码注入概念的例子。该例中，用户的输入直接映射给一个 SQL 请求，SQL 的定义如下：

```
SELECT city, zip, street FROM customers WHERE ID=input
```

现在，如果用户的输入（这也可能是一个服务请求的输入数据）是 42，则生成 SQL 请求如下：

```
SELECT city, zip, street FROM customers WHERE ID=42
```

这很好——对所有 ID 为 42 的数据库记录，返回城市、邮编和街道。

现在，考虑如下的输入：

```
42 UNION SELECT login, password, 'x' FROM user
```

如果你简单地把这个输入映射到 SQL 请求上，它就成了：

```
SELECT city, zip, street FROM customers WHERE ID=42 UNION
SELECT login, password, 'x' FROM user
```

结果是，你附带地从叫做“user”的表中取得了所有登录和密码字段（‘x’是必要的，它补充满 3 列，否则不可能做联合）。

使用 XML 时，同样的技巧也是可能的。用 XPath，你可以在 XML 文档间导航。如果输入直接映射到 XPath 的一个路径上，你就能操作路径，让它指向别处的 XML 文档，使你可以获取或操纵其他数据。

例如，考虑某些使用了如下 XPath 表达式的源代码：

```
string(//user[name/text()='user' and password/text()='pw']/account/text())
```

该代码的目的是在 XML 数据库中，或如下的 XML 文件中找出账户数据：

```
<user>
  <name>josuttis</name>
    <password>secret77</password>
    <account>admin</account>
  </name>
  <name>
    ...
  </name>
```

自由联盟计划

这儿提到的 Web Services 标准，并非是唯一与 SOA 协同工作的身份管理标准。2001 年，自由联盟计划（LAP，见[ProjectLiberty]）由超过 30 家组织共同创设，其目的是为联合身份管理建立一个公共标准。现在，LAP 由超过 150 个成员支持，其中包括主要的软件供应商，面向消费者的公司（比如电信运营商），以及政府和公共部门组织。

“身份联盟框架”（ID-FF）是 LAP 发布的第一个文件，它使用户可以参与一个互联网范围的单点登录（SSO），该登录基于联合的身份。之后，这些规范贡献给了 OASIS，形成了 SAML2.0 规范的基础。在 2003 年实施了下一个重要步骤，出版了“身份 Web Services 框架”（ID-WSF），一个开放式框架，支持将基于身份的 Web Services 作为 SOA 的一部分。ID-WSF 支持身份传播协议，以及在 Web Services 的供应者和客户端之间的服务发现，所有这些东西都基于标准协议。就 SOA 来说，ID-WSF 可以被认为是个长期解决方案，因为它把 SSO 从浏览器提高到了 Web Services 和后端系统的水平。

LAP 当前的焦点是为和身份管理相关的 Web Services 应用建立标准（ID-SIS）。地理定位（Geo-Location）、在场（Presence）、自由人群服务（Liberty People Service），以及其他规范都被证明是像博客、书签和即时通信应用这样的社区应用的核心组件。

自由联盟采纳了许多公共 Web Services 标准（比如，SOAP、XML 加密、XML 签名，以及 WS-安全），它也对现存的及新的 OASIS 标准（如 SAML 和元数据）作出了贡献。现在，有的 LAP 标准成了 Web Services 专用标准——如 WS-联盟——的竞争性方法。

为了处理互操作性，异质环境要求对自由联盟的标准、协议，以及这些东西的实现招数有深刻的认识。有鉴于此，自由联盟推出了证书计划——在正式的、有组织的互操作性测试中，各供应商和工具要证明自己在大规模异质环境下的互操作性。

通过商业产品和开源实现（比如 Sun 的 OpenSSO），自由联盟协议获得了重要软件供应商的支持。为了简化将自由联盟协议采纳入应用和安全基础设施的过程，2007 年初设立了被称为 openLiberty.org 的开源计划。不幸的是，到目前为止，还没有任何可公开获得的实现。

自由联盟的总体架构和协议都比较复杂，最好由大型组织来处理。对小一些的场景来说，当前我们看到的趋势是向即将出现的轻量级标准集中，如 OpenID 和 XRI，这减少了对 Web 应用进行简单集成的障碍。

——Jochen Hiller（*http://www.jochen-hiller.org*）

现在，如果用户名如下：

```
' or 1=1 or '='
```

XPath 表达式变成（插入了新行以使意思更加清楚）

```
string(//user[name/text()='
              or 1=1
              or '=' and password/text()='pw']/account/text())
```

因为由 or 连接的三个表达式中的第二个永远都为真，处于“[”和“]”中的整个限制就毫无意义了。结果是，这个表达式返回了第一个用户的账户。

对此要了解更多可见[Sanctum04]。

SOAP 附件

因为 SOAP 和电子邮件一样，允许任意类型的附件，所以，这就可能成为病毒、蠕虫等的另一个大门。因此，你可能必须集成通用的反病毒软件，将其作为你 SOA 基础设施的一部分。

14.5 安全问题何时凸现

When Security Comes into Play

最后讲几句有关安全问题何时凸现的话：SOA 是个相当复杂的战略，只有按部就班地引入，SOA 才能发挥作用。之所以如此，一个原因是你无法事先预测所有需求和某些实现的影响（不管是技术上还是流程上）。如此说来，你也无法在真正开始经历 SOA 之前解决所有的安全问题。这意味着你将随着 SOA 景观的演化而引入某些安全考量，改变某些安全状况。

安全是整个系统的基础部分，这意味着它对“所有的”服务都有影响。鉴于此，除了在一开始就应将其建造进来之外，你还必须能够改善现存服务的安全状况。

你应从一开始就有安全意识。你不必努力提供能适应所有可能的需求和情况的技术，但是，你应该提供供今后引入安全考量之用的扩展点。这意味着你应该引入执行安全的挂钩，在服务头中引入通用字段，从而使未来的字段不会破坏 API 的兼容性，如此等等。

从大处考虑，自小处着手。

14.6 总结

Summary

- 在 SOA 中，可以运用分布式系统常用的安全考量和技术，然后加上一些补充的措施：

  —SOA 强制要求高互操作性，因而降低了默认的安全性。

  —SOA 必须处理现有系统的、异质的安全概念。

—分布式流程通过多个服务传输数据，因此点到点解决方案不满足端到端的安全要求。

—多客户端能力迫使进行运行时安全检查。

- 你必须定义和实现一个覆盖基础设施、架构及应用的，战略性的安全方略。最好的办法是将安全当作服务引入。
- 安全影响性能，它也使无状态服务不得不具备处理状态的能力。
- XML 和 Web Services 可能引入特殊的安全问题，比如 XML 炸弹和 XPATH 注入之类的危险。
- 从一开始就保持安全意识（这并非意味着你需要立即着手处理安全的所有方面）。

一分钟之流量通过多个服务代理数据库，即此法测定响应方案不能足满到端的安全要求。

一客户端能力直接进行运行时安全检查。

- 必须定义和实现一个强壮基础设施，策略及应用的，能解决安全方略。此外的动态是指安全当作服务引入。
- 安全影响性能，它也使方法监控不得不具备处理状态的能力。
- XML 和 Web Services 可能引入特殊的安全问题，比如 XML 炸弹和 XPATH 注入之类的攻击。
- 从一开始就保持安全意识（这并非意味着你需要立即着手处理安全的最有方面）。

第 15 章

技术细节

Technical Details

每一个 SOA 都有一些需要实现（或至少要考虑）的技术考量。本章展示和讨论那些足够基本的、你应该熟悉的东西，因为迟早有一天，你非得和它们打交道不可。

15.1 服务和状态

Services and State

通常，对 SOA 的描述宣称服务应该（在理想情况）是无状态的。然而，在服务的上下文环境中，“无状态”或“有状态”到底指什么？这两个词可能是让人产生迷惑的源头。一个原因是，其实永远都有状态被牵扯进来，即使无状态服务也是如此。关键的问题是，这个状态在哪儿，能持续多久？除此之外，服务是否为有状态是个立场问题。从业务的角度来看，服务可能是无状态的；然而，从技术的角度来看，服务却是有状态的，反之亦然。本节在服务的上下文环境中仔细考察“状态”问题，澄清概念，讨论实践中使用的各种方法。

15.1.1 无状态服务
Stateless Services

从概念上，一个无状态服务是在不同服务调用之间不维持任何状态的服务。这就是说，在服务调用结束后，要销毁所有为运行服务而临时创建的局部变量和对象。请注意，我们谈论的是服务本身的数据，既不是调用服务的进程或应用的数据，也不是服务运行于其上的后端系统的数据。在调用之后，如果执行调用的服务实例（进程或线程）的所有数据都被销毁，服务就是无状态的。换句话说，下一个服务调用是用同一个还是不同的服务线程或服务进程执行，必须是完全无所谓的。

根据这个定义，甚至无状态服务也能改变状态。然而，这只是改变了后端或前端的状态。例如，如果一个服务向一个银行账户中增加些钱，然后返回新的账户余额，那么，这是个无状态服务。如果你两次增加 30 美元，你可以用两个不同的服务线程或进程来做。假设最初余额是 100 美元，第一次调用将增加 30 美元，返回新的 130 美元的余额；而第二次调用将增加另一个 30 美元，返回新的 160 美元的余额（见图 15-1）。

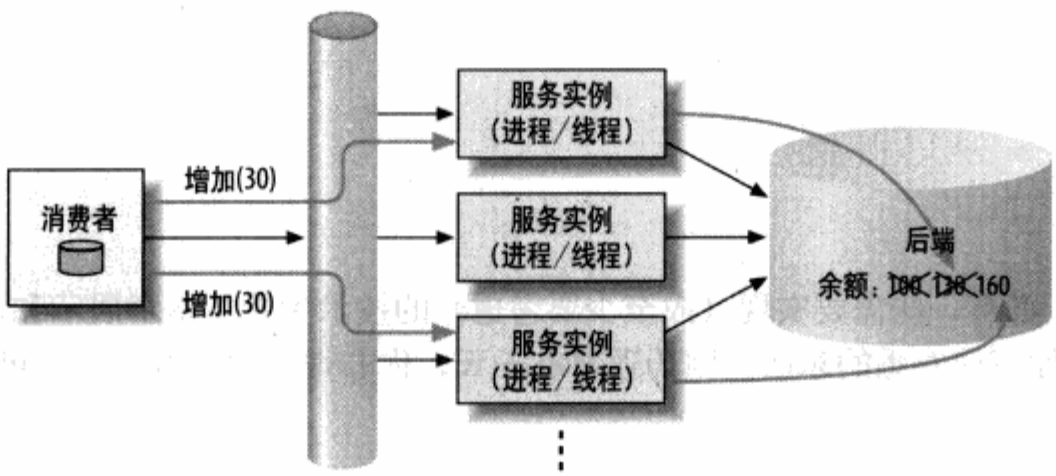


图 15-1：通过两个不同的无状态服务调用增加 30 美元

你可能会说，在这个例子中，很显然没有包括什么服务状态，因为该服务的运行甚至不需要在服务的实现中保留任何数据。但是，让我们看另一个需要在内部保持一个状态的无状态服务。假设你有一个在两个不同后端上修改地址的服务，对服务的一种松耦合实现是用补偿而非事务上下文环境（见 4.2.8 节），用伪代码书写如下：

```
string changeAddress (newAddress)
{
    oldAddress = backend_1.getAddress();
    status = backend_1.changeAddress(newAddress);
```

```
        if (status == failure) {
            return "couldn't modify address in backend 1";
        }

        status = backend_2.changeAddress(newAddress);
        if (status == failure) {
            // compenstation: restore old address to keep consistency
            status = backend_1.changeAddress(oldAddress);
            if (status == failure) {
                fatalError(...);
            }
            return "couldn't modify address in backend 2";
        }
        return "OK";
    }
```

如你所见，这个实现需要某些状态信息，比如 oldAddress。在第一个后端修改地址之前，这个后端的老地址被存储在 oldAddress 中。之所以这样做，是因为一旦后端 1 修改成功而后端 2 失败，你就能对后端 1 恢复老地址，从而维持总体一致性。在运行时，这个服务有一个状态，当服务结束时，相关后端的状态可能会被修改（见图 15-2）。

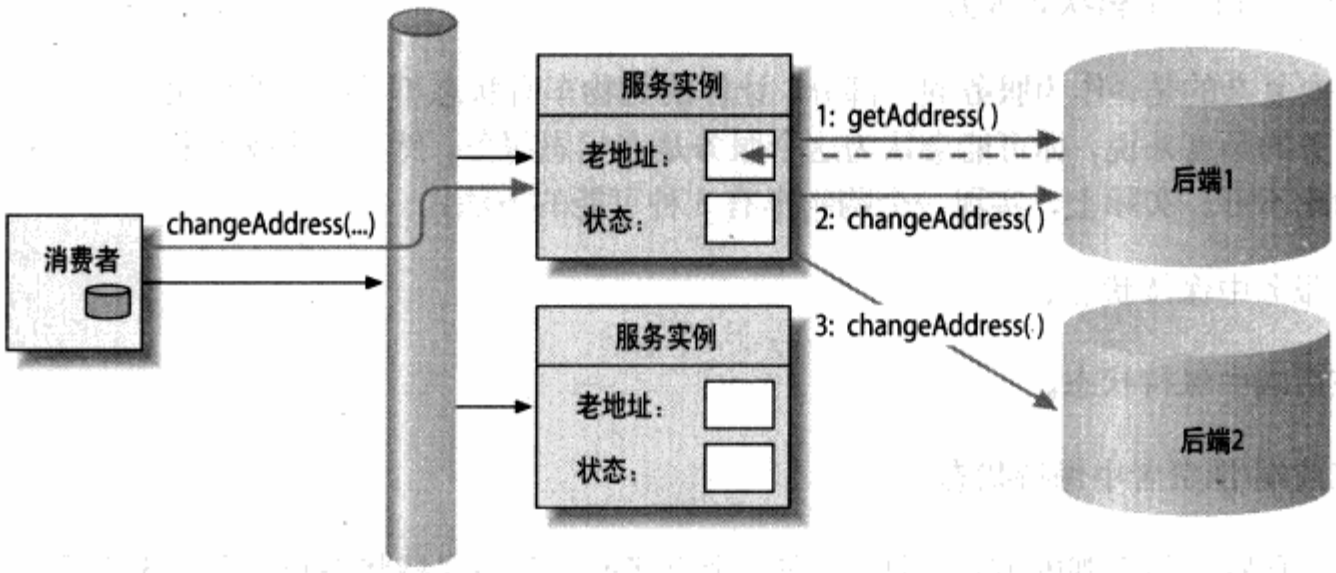


图 15-2：运行有一个临时内部状态的无状态服务

然而，该服务就整体而言是无状态的，因为在调用结束时，你可以销毁所有内部数据。对该服务的第二次调用可以运行在完全不同的服务实例上。

15.1.2 有状态服务
Stateful Services

那么，什么是“**有状态**”的服务？一个有状态的服务，指的是在多个服务调用之间，可以保持状态的服务。

有状态服务的典型例子是购物车。如果你在网上购物，比如在 Amazon.com，每当你挑了一本书，把它添加到你的购物车中时，购物车中商品项的数量就会增长。

并且，如果你对购物车中的商品下订单并付款，这些商品就会被发送给你（见图 15-3）。

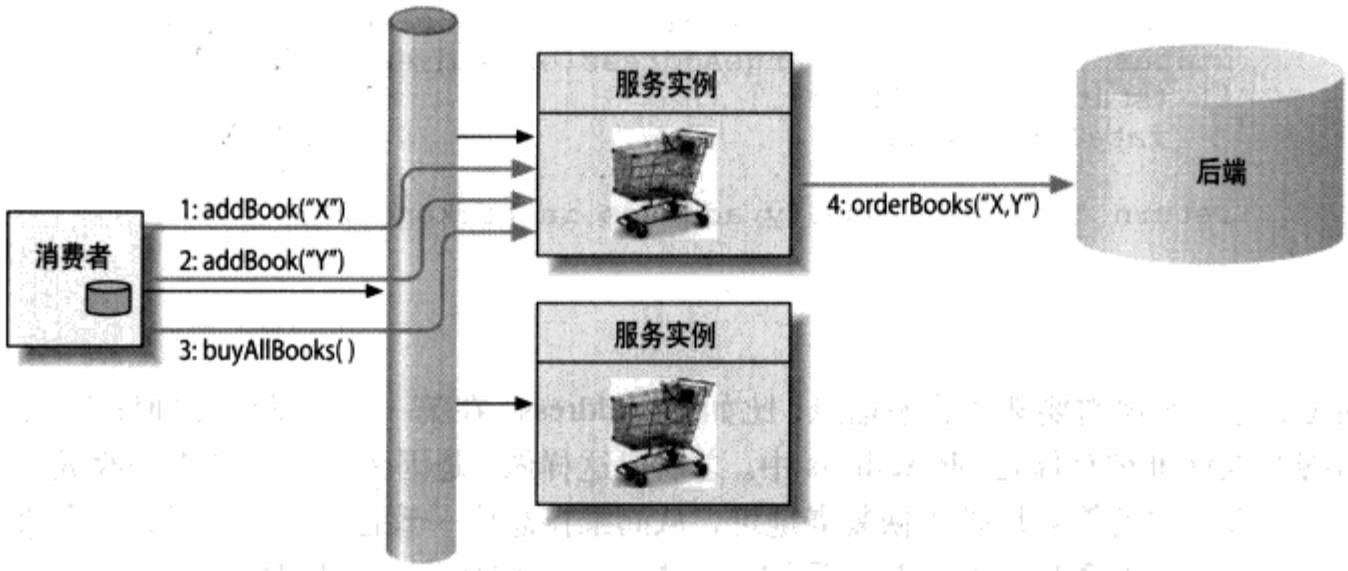


图 15-3：运行一个有状态服务

使事情变复杂的是，作为服务的一部分，让你的购物车有状态不是唯一的解决方案。虽然从业务的角度来说，你可能会认为这个服务是有状态服务，然而，从技术上来看却未必非如此不可。实际上，实现一个购物车有 3 种可能的方法：

- 在服务中保持状态。
- 在后端中保持状态。
- 在前端/消费者中保持状态。

实现一个购物车最直观的方式也许是，在一个有状态服务中保持状态（如图 15-3 所示）。这意味着直到用户对购物车中的商品下订单之前，后端不知道关于购物车的任何情况。要向购物车中装东西，你必须保证后续服务调用被路由到相同的服务实例。这就是说，你用第一个服务建立了一个“**会话**”。通常说来，每个服务调用都返回一个会话 ID，消费者将这个 ID 与后续请求一起发送，这样 ESB 就能将请求路由到相应的服务实例。当你使用一个浏览器在 *Amazon.com* 上购物时，这个状态通常以 cookie 或是 URL 一部分的形式保存下来。

依赖于你的浏览器政策，当你的 cookie 被删除时，或者当你使用另外的万维网浏览器访问该站点时（例如，如果你从 Firefox 切换到 Internet Explorer），你的购物车将被清空。

现在，让我们看看实现购物车应用的另一个方法。第二个解决方案是在后端保持状态。又一次，你需要一个方法在后端标识用户的购物车（或会话）。与上一个例子相同，后端可以创建一个会话 ID，服务把该 ID 传输给消费者，然后消费者再把 ID 作为 cookie 或 URL 的一部分保存下来。现在，当执行一个新服务调用时，消费者将会话数据发送给一个无状态服务，服务把数据传输给后端，后端将数据用于识别会话。

两个解决方案都需要某种方法来定位当前会话。对有状态服务来说，这是 ESB 的任务；当服务在后端保持时，这是后端的任务。

第 3 个解决方案是在前端或消费者中保存状态。此时，服务向消费者不仅仅返回会话 ID，而是返回完整的状态（其形式消费者也许能读取，也许不能读取）。状态还是保存为 cookie 或 URL 的一部分。当另一个商品添加到购物车时，消费者把整个状态发送给服务。因为完整的状态保存在客户端，所以，服务可以是无状态的。并且，再一次，直到用户对商品下订单，或者将购物车持久化供今后所用时，后端不知道关于购物车的任何情况。

注意：所有这 3 种选择（表 15-1 作了总结）有不同的优点和缺点：

- 将状态放入前端/消费者意味着，服务可以是无状态的，并且可以不牵涉后端。然而，当会话在运行时，你无法改变前端（如果要求支持多前端，或者如果你希望能够将会话转移到另一个桌面，这就是个缺点）。
- 将状态放入有状态服务需要在服务层上使用更多的资源，并且，要有办法将请求路由到同一个服务实例。然而，你可以改变前端，后端也不被牵扯进来。
- 将状态放入后端使你能使用无状态服务，前端也不必提供存储。然而，每一个会话都牵涉后端。

表 15-1：处理状态的方法的对比

	状态在前端	状态在服务	状态在后端
在前端存储	所有状态	ID	ID
服务	无状态	有状态	无状态
在后端存储	/	/	所有状态
支持多通道	否	是	是

注意：从消费者的角度来看，所有这 3 个政策导致相同的接口。服务都返回某些必须随着下一个请求发送的数据。唯一的区别是前端存储的数据大小不同、意义各异。因此，你的服务接口应该使用通用数据类型（二进制或字符串类型），为选择使用哪个政策留下活动余地。你甚至可以随着所考虑事情优先级的变化而改变政策。

15.1.3 无状态服务为什么更好
Why Stateless Services Are Better

虽然我们讨论了无状态和有状态服务之间的不同点，然而，我们没有讨论是否一个政策比另一个更好些。有一些建议更偏爱无状态服务，但是，为什么会这样？

首先，对无状态服务来说，在服务层做到负载平衡和失效备援相当简单。ESB 仅需实现能使用下一个最容易获得的服务实例（某些硬件上的进程或线程）既可。它甚至可以是其他消费者以前用过的线程（如果实现了线程池机制的话）。这意味着如果一个系统失败了，其他系统可以轻易地接手，继续工作。如果服务层的通量不够高，你可以将服务实例的数量翻倍，从而使通量翻倍。这意味着解决方案能线性伸缩，并且，如果一个服务实例意外死亡，其他服务实例能投入进来。

而有状态服务的缺点就是，它们被绑定在会话上。这样一来，直到消费者会话结束（或连接失效）之前，为它们分配的资源都绑定在一个特定的消费者会话上。除此之外，你必须能让 ESB 把连续的服务调用路由到同样的服务实例上。

然而，要注意的是，有状态服务仍然可以线性伸缩。使用一种被称为**“粘性路由”**的政策，根据可用的资源，你在第一个服务调用被执行时找到你的服务实例。这一来，对服务实例数量的翻倍就又能将可能的会话数量翻倍了。然而，此时不具备服务资源可以在不同消费者之间共享的优点。并且，如果一个服务实例意外死亡，会话状态就丢失了。

同时也要注意，有状态服务可以有失效备援机制（并且仍然可以线性伸缩）。此时，每个服务器不但返回它自己的会话 ID，还返回失效备援会话的 ID。只要这个失效备援会话不在一个中央服务器上，而是在邻近的服务器上（并且，每个服务器的邻近服务器各不相同），这就不会造成瓶颈。

有的应用服务器支持这个特性。然而，在实践中，如果服务器出故障，丢失服务器会话数据通常是可接受的（前提是这种场景相对罕见）。如果数据比较重要，或甚至对业务来说是“法律相关”的，通常建议使用后端也在用的、坚固的备份策略。如果你把这样的数据放到了有状态服务中，你的服务层就（静悄悄地）获得了后端系统的角色，这通常导致让人困惑的架构（角色和职责）。对这个话题的更多讨论请见 6.4.2 节。

15.1.4 实践中的无状态服务
Stateless Services in Practice

在实践中，由于各种各样、非常不同的原因，服务可能变成有状态的：

实现购物车和支持带多通道的流程

正如以前讨论的，可能有一个业务需求是，针对多个服务调用的联合状态，在对状态做进一步处理或将其持久化到后端之前，需要保存此状态。此时，可以用一个有状态服务来避免把状态保持在客户端，这样你才能在服务运行的同时改变前端。

批量服务

有时一个服务会处理大量数据和/或调用成百的其他服务。例如，一个 CRM 服务可能对某类型的所有客户（为同一家公司工作、现有的合同相同，等等）给予一个新合同或选项。通常，你不可能保证所有这些成百上千的修改都能成功。但是，你也无法事先预测会出现什么问题，所以，你无法提供解决问题的原则和实现。你如何处理错误依赖于错误的类型，以及错误发生得有多频繁，并且，你也不能等到服务结束了，才决定如何处理错误。在这种情况下，你需要一些和这种“长期运行”的服务打交道的能力，这意味着你必须能访问服务的状态。

性能

性能可能是引入有状态服务的非常重要的原因。一个典型的例子是允许你在结果列表中进行遍历的服务。处理结果可能要花相当多的时间。如果你返回了前 100 个结果项，然后丢弃了状态，那么，当消费者请求第 101 个到 200 个结果项时，你将不得不处理 200 个结果（返回前 100 个结果之所以必要，是因为要找出后 100 个结果项从哪儿开始）。用数据库术语来说，你需要一个存放完整的结果和相应的游标的地方，这样就可以对其进行遍历。然而，在这儿还是有别的选择：你可以把结果保存在后端，用无状态服务将游标/遍历器的状态传给消费者。

一般说来，当在服务层内进行缓存是有用的或必需的时，有状态服务总是有用的。从消费者的角度来看，这是个实现细节，所以，服务接口不应该变化。然而，你需要一些再次连接到相同服务实例的方法。如果这不是在 ESB 内部处理，你就可能需要一个通用的属性（二进制或字符串），第一个服务调用可以将其作为某种会话 ID 返回，后续服务调用可以发送它。

安全

安全是切换到有状态服务的另一个重要理由。也许有必要在一个服务内部保存会话状态，这样你就不必在每次服务调用时都要识别消费者，加载它的用户配置文件。细节见 14.3.4 节。

因为有状态服务是有用的，所以，提供某种实现和处理有状态服务的方法是个好主意。如前所述，有状态服务可能要求更多的资源，但是，它们不妨碍线性伸缩性（前提是没有一个会话或有状态服务发出的服务调用数量会占所有服务调用的 30%）。

当然，从概念上来讲，建议服务应该是无状态的仍然是个好想法。从业务的角度来看，服务应该是自足的、原子的。然而，正如你刚刚看到的，让服务有状态存在许多原因。

15.2 幂等性
Idempotency

幂等性是服务的一种可能的属性（见 3.3.5 节）。在服务的上下文环境中，它意味着对同一服务调用的多次递交/处理不会引发问题。假设你有一个向银行账户中增加钱的服务。如果一个消费者调用这个服务（例如，为了处理银行转账）并且没有得到应答，消费者不会知道是服务的请求还是服务的应答失败了（见图 15-4）。

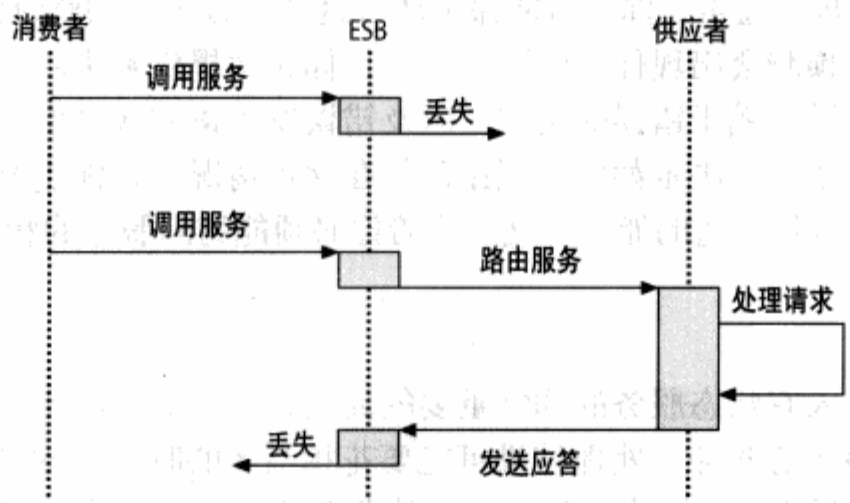


图 15-4：服务消息可能在请求被处理之前或之后丢失

在后一种情况下，从供应者的角度来看，服务成功了，指定数量的钱被增加到银行账户上。在前一种情况下，服务没有影响（即没有增加钱）。

当服务的消费者没有得到应答时，它不知道服务是否成功了。为求确保，它可能重试服务调用，再次发出同样的请求。结果是，到达供应者的服务请求可能有一次或两次。如果服务是幂等的，那么，服务请求到达多少次都没有关系：同一服务调用的多次请求只产生一次的效果。

刚刚描述的银行服务不是幂等的，因为，它依赖于第一个调用失败发生在何处，客户的余额可能最终被增加了两次，而不是消费者想要的一次。为了避免这样的问题，通常说来，只要有可能就使服务幂等是个好主意。

15.2.1 幂等的服务
Idempotent Services

所有读取服务都是幂等的，因为一个服务供应者无论多频繁地执行返回数据的请求都没关系。然而，要注意，一旦做了类似于每个请求都写一个备忘录项这种事，读取服务就变成了写入服务。如果这个备忘录和业务有某种关联，这就可能成为问题（例如，对同样的事项，最终可能有多个备忘录项）。

写入服务可能是幂等的，也可能不是。当一个服务调用的影响依赖于后端现有的状态时，写入服务不是幂等的。举个例子，在前面的例子中，银行账户的最终余额依赖于最初的余额。如果开始的余额是 500 美元，服务调用增加 100 美元，结果将是 600 美元。如果因为最初的应答丢失了，另一个请求到达并且得到处理，最后的余额将是 700 美元。

一个幂等的写入服务的例子是，一个通过发送所有地址数据来设定客户地址的服务。供应者处理多少次请求都没有关系，结果将是一样的。

注意：你总是可以避免让服务不幂等。例如，我们可以修改银行服务的语义，发送新值而不是应该增加的值，从而使服务幂等。这就是说，不是调用：

```
addToBalance(100)
```

我们可以调用：

```
setBalance(600);
```

这个调用可以被反复地发送和处理，余额的结果都会是 600。当然，如果其他服务也能在相同的账户中增加或减少资金，这就带来了问题。你如何才知道最终余额应该是多少呢？正如你看到的，从业务的角度来看，让服务幂等可能变得复杂。

除此之外，从业务的角度来看，有的服务内生就是不幂等的。所有创建某些东西的服务都没法和一个状态去比较（除非你知道创建新资源的工厂的状态）。例如，如果你有一个创建新银行账户的服务，你就是在创建账户，仅此而已。

15.2.2 实现幂等性

Implementing Idempotency

如果从业务的角度来看，引入幂等性是个难题的话，那么，你需要一些对幂等性的技术支持。实现幂等性的普通方法相当简单，消费者发送数据的方式必须能让供应者看得出两个技术性请求其实是同一个。为此，消费者可以随着每次新请求发送一个唯一的 ID。如果消费者没有得到应答，它的每次重试都用同样的 ID。图 15-5 是对此的图解。

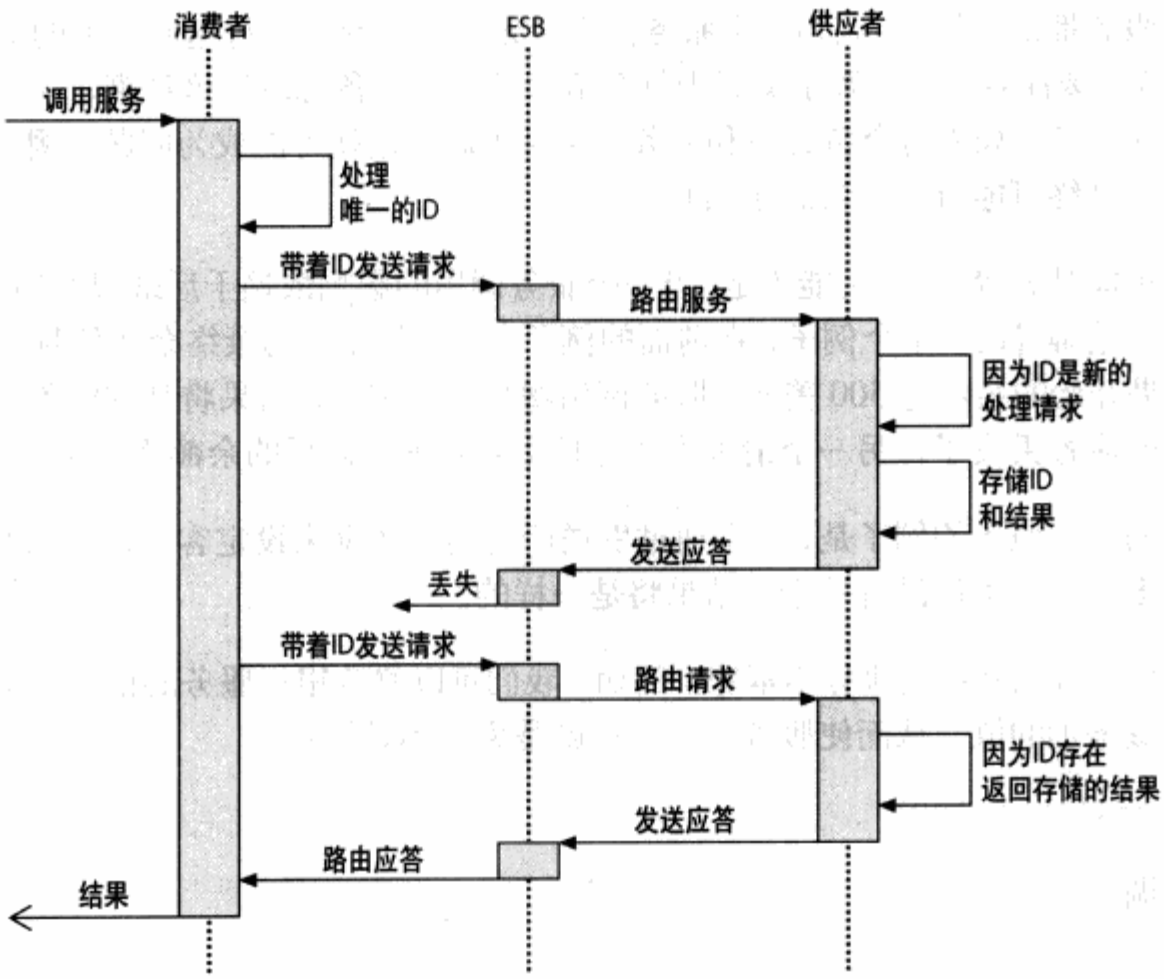


图 15-5：让服务在技术上幂等

在处理每个进入的请求之前，供应者应将请求及与其相关联的唯一 ID 存储在一个中央存储库中。如果供应者收到了一个 ID 已经在存储库中的请求，它就知道该请求已经被处理过了。于是，供应者不会再处理它，而是向消费者重新发送最初的应答（应答在第一次被送出去之前，也保存了下来）。如果在第二个请求到达时，服务实现正处于运行之中，则供应者可以回复一个应答，说服务调用在运行中，或者供应者可以等到第二次请求处理完毕后，再次把应答发送出去。

作为一项优化，消费者能和请求一起发送一个标志，这样供应者就知道此请求是否是一个重试。这样做能够改善性能，因为如果调用没有被标记为重试的话，供应者就不必验证请求 ID 是否已经在其存储库中了。当然，幂等性并不只与请求消息有关，也不是只

影响服务供应者。如果消费者因为应答经过太长时间没有到达，从而重发请求的话，就有可能最终接收了多个应答。然而，对消费者来说这种情形比较容易处理，因为它知道自己曾发出了两个请求。

要注意通信协议——包括著名的、用于 Web Services 的协议 HTTP——通常都是不可靠的。也就是说，消息有可能丢失。在这种情况下，幂等性是获得可靠性的常见要求：要变得更可靠，如果任何组件不能确定消息是否成功递交，则组件都应该重新发送消息。然而，这意味着消息的许多拷贝可能被递送出去，所以，不要造成预计外的效果这一点很重要。实际中，协议只保证消息将**至少**被递交一次。要保证消息将被“**递交一次并只有一次**”需要付出甚至更多的努力（即实现我们刚刚讨论的，作为一种一般方法的，实现于基础设施内部的，技术上的权宜之计）。第 10 章提供了对消息交换模式的全面讨论。

15.3 测试和调试
Testing and Debugging

分布式流程是业务的基础部分。因此，在把它们带入生产阶段之前，你必须保证它们工作正常。除此之外，你必须让跟踪问题，以及当问题发生时有修复缺陷的机制存在。

15.3.1 单元测试
Unit Tests

服务是单元测试的天然盟友。对每一个单独的服务来说，实现单元测试应该都相对容易。因为服务应该是自足的，所以测试接口的单元测试通常也会是自足的。唯一的问题是，提供的测试数据要足以对运行在测试模式下的服务进行语义上的验证。

对一个基本服务来说，你只须以某些方式为后端提供测试数据，这相对容易（你可能已经有这样的数据可用了）。然而，正如你将在下一节看到的，对更复杂的服务来说，测试工作就会更困难。

注意：如果你生成服务代码，那也要生成测试代码。这将减小编写单元测试的工作量（见 18.1 节）。

15.3.2 分布式测试
Distributed Testing

测试组合或流程服务会成为真正的难题。因为这些服务使用多个后端，所以，你需要分布到所有后端的、一致的测试数据。在所有者各异、利益不同的系统中，这可能成为一个噩梦。如果不可能这样做，你将不得不模拟使用到的其他服务（这降低了测试的质量）。

我曾见过的一个方法是，用本地哑元实现替换对其他系统的调用。然而，如此一来，单元测试就只能测试服务的“某些”行为，而不是全部的分布式行为。那家碰到问题的公司的情况是，这个方法确实有助于本地测试，但是，直到涉及所有后端的集成测试开始时，有许多问题都没有被发现。

另一个可选方法是引入给服务调用打上“在测试模式中”的标签的能力。因为提供分布式测试数据可能非常复杂，所以，你迟早会在生产环境中测试。如果只涉及一个后端，你也许还能回滚一些修改。如果分布开始发挥作用，那么，能将服务调用标记为测试，从而写入服务不会把修改提交给后端就很有用了。否则，测试数据将一遍遍地被写到后端上，后端必须定期恢复。如果 ESB 能将服务调用标记为“在测试模式中”，这将方便测试生产模式下的连接性和流程的流动情况。

然而，对复杂的分布式业务流程，我还从没看到过无须手工测试来验证流程在分布式系统上的确切行为的情况（包括使用恰当的方式来处理错误）。

15.3.3 关联 ID
Correlation IDs

在分布式环境中，监测流程和数据的流动是非常重要的，这样你才能理解系统的行为和分析缺陷。将业务逻辑进行分布没能使逻辑更简单，正如本地处理一样，你需要跟踪和监测分布式处理的能力。问题在于，在分布式环境中，你不是在单个进程中或单个系统上，你没法用自己最喜欢的调试器。要进行调试，就需要基础设施的支持。换句话说，对分布式流程来说，你的基础设施给了你调试器。

实际上，就工具支持和付出的努力来说，在本地调试和分布式调试之间的差异巨大，尤其是如果异质也在起作用的话。这可能是分布式处理的主要缺点。当所有东西运转顺利时，分布看起来很棒，但是，当出现问题时，分布就可能变成噩梦。

因此，对你的基础设施（ESB）有一个至关重要的要求：能将构成服务调用或完整业务流程的消息关联起来。另外，你还必须能将 ESB 外部的数据流和 ESB 内部的数据流关联起来。因此，通常你需要能对不同系统和流程的数据用一个或多个 ID 打标签。这样的 ID 被称为“**关联 ID**”。

关联 ID 既可以被用来关联所有属于“**一个服务调用**”（包括在服务调用中发起的服务调用）的数据和处理过程，也可以被用来关联所有属于“**一个业务流程**”（包括所有服务调用）的数据和处理过程。

使用多少关联 ID 由你自己（或你的基础设施供应者）决定。ESB 通常有其内部（消息）ID，但一般说来这不够。对每个服务调用，你最终可能需要用多达 4 个 ID：

- 一个用于消息的本地 ID，由你的基础设施（工具）提供。
- 一个 ID 将消息与所有其他属于同一个服务调用的消息关联起来。
- 一个 ID 将消息或服务与所有其他属于同一个业务流程的消息和服务调用关联起来。
- 一个技术性 ID 用于处理幂等性（见本章早些时候的 15.2.1 节）。

在实践中，ID 可能服务于不同的目的，也存在各种方法来关联不同的 ID。不过，你应该把不同 ID 的语义考虑进来。

你的关联 ID 政策可能随着 ESB 的成长而演进，但是，你应该在最开始决定一件事情：ID 的数据类型。毫无疑问，ID 应该是可读的字符串，这样它才容易被跟踪、过滤、查看和处理，而且，它应该没有固定的大小（为了适应未来的需求）。

注意

我的一个客户最初把关联 ID 定义为两个整数的组合（一个用于系统，一个是由系统计算出的唯一的 ID）。追踪这些 ID 变成了一场噩梦，因为整数的字符串表示各不相同（某些系统将它们解释为有符号的值，其他的则认为是无符号的值，一个系统甚至把两个 ID 并成了一个值）。除此之外，因为 ID 很难读，所以查看日志文件，试图过滤出某个特定时期内的日志记录需要巨大的工作量。不幸的是，我们无法采用更好的格式，因为这是每个服务、每个系统都使用的二进制数据，我们没法在所有系统上同时替换格式。最后，我们决定“滥用”另一个所有服务都有，但已经不再使用的字符串属性。

15.4 处理技术性数据（头数据）

Dealing with Technical Data (Header Data)

关联 ID 只是 ESB 内部在运行时使用技术性数据处理消息的一个例子。除业务数据（“有效载荷”）之外，你还需要数据用于路由消息、检查安全考量、跟踪和监测消息，如此等等。所有这些如何处理依赖于你的中间件和 ESB 技术，然而，这些信息通常都存放在某种类型的头部（例如一个 SOAP 头，见 16.2.3 节）中。关于这样的头数据，我想给出一个通用的提示：永远不要假设你事先就了解所有的技术特性。

当你的 SOA 系统景观处于运转中时，你会发现当自己从基础设施的角度来看，用了其他的或新的方法来处理消息时，不可避免地会需要某些额外的数据。因此，你必须提供一些方法，既不破坏为服务生成的接口，又可以添加新的技术性特性。这意味着二进制兼容性是必须满足的条件。改变消息的基础技术格式，以致需要所有相关系统都再编译和同时重新部署，这几乎是不可能的。正因如此，对你的技术性头部来说，推荐的方法是包含一个可以随着时间增长的通用链表（比如一个键/值链表）。

15.5 数据类型
Data Types

另一个你必须决定的基础技术细节是，通过服务交换数据时，使用哪种数据类型和编程范式。正如第 4 章所讨论的，有关松耦合的考虑会影响这个问题。你所引入的数据类型和编程模型越复杂，系统将耦合得越紧密。

在分布式系统上，一致化数据类型是个常见而自然的目标，因为这简化了数据共享过程。然而，要小心——一旦引入一个新的基础类型，你很可能再也无法去掉它。

注意

在一个项目中，我们为未来的技术性数据定义了包含一个键和一个值的通用链表，还定义了值的类型信息，以及一些能够对如何处理该值给出更多提示信息的额外的属性（例如，一个属性允许作出这样的定义：当服务供应者处理一个服务调用时，如果要调用其他服务，应该将这个数据传递下去）。

15.5.1 基础数据类型
Fundamental Data Types

你应该清楚地区分基础数据类型（FDT）和公共数据类型：

- 基础数据类型简单、稳定，所以不需要为其提供版本控制。代码生成器和适配器对这些类型具备内生的行为和实现。
- 公共数据类型可能随着时间变化，所以需要版本控制。

在实践中，对指定基础类型采取保守态度通常会获得良好回报。然而，不要走极端到只提供一个字符串类型。我建议从以下这些基础数据类型开始：

- 字符串。
- 数值类型（整数、浮点数，等等）。
- 布尔值。

- 一些基础的、通用的业务类型，比如日期/时间
- 一些特定领域的基础类型，比如电信公司的电话号码，金融领域的银行号码
- 一个用于交互二进制数据的类型

正如 4.2.2 节所建议的，要小心特定领域的基础类型。因为消费者对地址类型或银行账户类型之类东西的需求可能差异巨大，而且会随时间变化，所以，将这样的类型指定为基础数据类型就太冒风险了。除此之外，在所有的系统之间统一这样一个类型通常被证明是过于复杂的事情。如果你能针对这样的类型达成一致，则可以将其定义为具备不同版本的“公共”数据类型，就像任何其他特定领域的数据类型一样（见 12.3 节）。

注意，你也必须定义基本数据类型的确切语义。一个重要的问题是，这些数据类型是否提供引用语义（在下节讨论）。除此之外，你应该明确字符串是否能包含二进制和/或宽字符，数值类型的范围，等等。

基本数据类型应该是你的元模型的一部分（见 18.3 节）。

15.5.2 引用语义
Reference Semantics

实现引用语义（有时也称为“空值语义”）意味着存在被定义为代表“未设置”（在编程语言中，诸如 null 或 NIL 这样的值就是用于这个目的）状态的值。这对服务接口有重大影响。

例如，使用引用语义，你可以提供具备如下签名的接口：

```
changeContactData (string customerID,      // customer ID
                   string phoneNumber, // null: no change, empty: delete
                   string faxNumber, // null: no change, empty: delete
                   string email);    // null: no change, empty: delete
```

对所有这 3 个参数——phoneNumber、faxNumber 和 email——你可以传递一个包含新值的字符串。然而，传递 null 意味着没有变化，现有客户属性继续有效。这与传递空字符串不同，后者将导致现有的值被删除。没有引用语义，你没法讲明“不变化”和“删除”之间的区别，这意味着消费者要么总是必须传递所有的 3 个值（因为它们替换现有存在的值），要么意味着传递一个空字符串没变化，从而需要另一个额外的接口来删除现有的值。

当然，如果你要实现引用语义，那么你使用的基础设施协议及编程语言必须支持它。虽然 Web Services 确实支持引用语义（用 nillable 属性），Java 也支持（使用类似于 Integer 而不是 int 这样的类型），然而，对其他编程语言来说，你可能需要变通方案。例如，

在 C++中，你不能使用标准库字符串类型实现引用语义，因为它只有值语义。因此，当绑定在这种特殊的语言中时，你需要使用一个特殊的字符串类型，否则就不能处理“未设置”和空值之间的差异。

15.5.3 更高阶的数据类型
Higher Data Types

你可以利用基本数据类型来创建新类型。要注意，为了遵守松耦合原则，每个供应者通常有自己的类型（见 4.2.2 节），为避免冲突，可以采取某种命名空间约定。例如，正如以前提到过的，针对所有系统共用的、带版本的数据类型，你可以有一个公共命名空间。

原则上，先者推荐提供定义数据类型的能力，类似于结构化编程语言中可以进行的那样。这意味着：

- 你可以组合出包括许多属性或不同类型的成员的结构或记录。
- 你可以定义出特定数据类型的链表/数组。

除此之外，你可能还想定义类型别名（typedefs），这使你能在语义上使用一个类似于 CustomerNumber 的类型名，然而在技术上，它只是被作为字符串来处理。

你可能还想对值进行限制，比如数值的特定取值范围，或者字符串的格式。然而，不要掉进期望接口可以规定所有语义限制的陷阱。对大型系统来说，永远都存在某种局限性，拥有简单的接口总是更好些。对其他所有的东西，可以使用运行时的异常和出错消息。

对其他任何类型都要非常小心谨慎，你不应该使用枚举类型（代表一系列可能值的整数）。字符串（加上了对值的限制）更容易被读取，被更多的平台支持。另外，要避免继承和多态，否则，当为非面向对象的语言提供恰当的接口时，你可能会碰到麻烦。

注意

Web Services 提供了很多笔者建议应该小心应对或不要使用的功能：继承（称为扩展）、复杂的值限制（称为限制），以及选择（某种类型的联合）。Web Services 使用的是 XSD 类型，而 XSD 服务的对象不仅是 Web Services，这也许是用 Web Services 提供互操作性如此困难的原因之一（见 16.3.1 节）。虽然笔者的建议对 Web Services 提供的功能施加了限制，然而，遵循这些建议能使你的日子过得更轻松一些。

当定义你的基本数据类型时，你必须在强大、复杂性和可移植性之间找到正确的平衡。正如阿尔伯特•爱因斯坦所建议的："让任何事情都尽可能地简单，但不要比这更简单。"请牢记：要处理所有这些特性，你往往需要自己写的代码，并且所有的复杂性对维护和调试都有影响。

15.6 错误处理
Error Handling

不幸的是，事情可能出错。因此，你需要一些处理错误的方法。异常是通常使用的方法，它被用来处理未曾料想到的行为，同时又不必在正常的接口内部处理任何错误。于是，返回某些东西的服务（比如第 10 章讨论的，使用请求/应答消息交换模式的服务）不仅要有一个返回类型，还应该有一个或多个用于错误和异常的类型（请看 18.3 节的元模型）。例如，用 Web Services 时，你能指定用于此目的的错误消息。

虽然你能对每个服务定义特定的异常类型，然而，你应该考虑对所有服务引入一个公共的类型（注 1）。这样做的优点是：能够以一种通用的方式来处理异常。否则，诸如一个通用的路由器或网关，当必须处理它不知道的服务时，就无法返回有效的错误消息。

当然，公共错误或异常类型应该提供足够的信息，使服务仍然可以进行单个的错误处理。为此，它应该包含一个独特的错误 ID，以及一个有特定细节的键/值链表。表 15-2 给出了一个例子，展示了通用异常类型可能具备的属性。

表 15-2：通用的服务异常类型的属性

属性	数据类型	意义
消息	字符串	一个（默认的）错误消息
错误代码	字符串	一个 ID，它使你可以编程处理这类型的错误
错误分类	字符串	错误的类型（致命、技术、逻辑）
时间	日期/时间	错误发生的日期/时间
来源	字符串	发现错误的系统
上下文环境	字符串	某些上下文环境信息（比如堆栈跟踪）
参数	键/值链表	额外的特定错误的信息
关联 ID	字符串	关联 ID（指明了流程）
内部异常	服务异常类型	引起这个异常的异常（使你能处理被包装的，或者被映射到其他异常的异常）

注意，警告并非错误，警告必须是服务返回类型的一部分。

注 1：有可能将公共异常类型作为所有异常类型的基本类型，但是，记住笔者刚刚推荐的——在服务接口中避免继承。

15.7 总结
Summary

- 引入有状态服务存在好几个理由，然而，要注意，根据看问题的不同角度，处理状态有不同的方法，处理状态有可能相当复杂和让人困惑。
- 尽量用无状态服务，在必要时使用有状态服务。注意，有状态服务需要更多资源，但通常不会妨碍线性伸缩性。
- 处理状态的方法应该对服务消费者透明。
- 努力使你的服务幂等。如果这不可能，考虑一个处理不幂等服务的公共方法（政策）。
- 支持服务的单元测试。然而，要注意，对非基本服务而言，由于提供一致的分布式测试数据任务艰巨，单元测试可能变得非常复杂。
- 当所有事情运转顺利时，分布看起来很棒；但是，当出现问题时，分布就是个噩梦。对调试和监测提供支持是重要的。
- 使用关联 ID，并且对这些 ID 使用字符串数据类型。
- 要提供既不打破接口的二进制兼容性，又可以在今后扩展消息头的功能（通过键和值的链表）。
- 为你的服务基础设施（ESB）定义基本数据类型和编程模型时要小心，笔者建议使用公共的结构化数据类型。
- 考虑定义一个为所有服务使用的公共异常类型，这使你能用一种通用的方法来处理错误。

第 16 章
Web Services

如今，大多数分析家、供应商和作家都只推荐一种实现 SOA 景观的方法：使用 Web Services。

本章中，我将从概念角度讨论 Web Services，提供一个 Web Services 当前状况的概览，讨论如何在实践中运用 Web Services。笔者的目标是提高你对 Web Services 的理解程度，讲清楚使用 Web Services 时要完成的任务及需要付出的努力。

16.1 使用 Web Services 的动机
Motivation for Using Web Services

人们普遍认为，Web Services 应该是实践中实现 SOA 的“唯一”方法。例如，[Erl05]阐述：“当代 SOA 体现了一个……架构，该架构促进面向服务，由……服务构成，用 Web Services 实现。”（完整的引用见第 2 章“Thomas Erl 在“面向服务的架构”中对 SOA 的定义”）并且，[Gartner05]预测：

> 到 2006 年，金额为 5270 亿美元的 IT 专业服务市场中，60%以上将基于对 Web Services 标准和技术的开拓。

当然，正如 SOA 通常的情况那样，所有这些意味着什么不是特别明朗。在实践中，什么是 Web Services？让我们看看在一个访谈中给出的定义。采访人为 Kirk McKusick（KM），加州大学伯克利分校计算机系统研究组的前领导。采访对象为 Adam Bosworth（AB），20 世纪 90 年代晚期微软高级经理，定义 Web Services 工作的最核心的人物之一。访谈发表在 2003 年 3 月第一期 *ACM Quene* 上（见[McKusickBosworth03]）：

> KM：很显然，人们谈了很多关于 Web Services 的事情，但是，还是不清楚他们谈的是不是同一件事情。你如何定义"Web Services"？
>
> AB：Web Services 术语指的是允许应用程序相互通话的架构。
>
> KM：了解。那你认为 Web Services 不是什么呢？
>
> AB：呃，它们不是超级有效率。但是，那没什么大不了的，因为我们谈的是自我描述的，沿途都易于路由、控制和人工操纵的消息。在以前的消息基础设施中，情况可不是这样。

对读者来讲，上面这些可能还不够。那么，让我们深入看看 Web Services 的历史和术语的细节吧。

16.1.1 什么是 Web Services？
What Are Web Services?

Web Services 指的是一个覆盖了互操作性的标准集合。实际上，这些标准既定义了用来通信的协议，也定义了用来阐明服务和服务契约的接口的格式。

16.1.2 Web Services 的历史
History of Web Services

微软在 2000 年创造了 Web Services 这个词，它描述的是允许计算机通过网络（互联网是其典型，但并不局限于此）相互通信的一套标准，核心标准之一是可扩展标记语言（XML），另一个是 HTTP（互联网的核心协议）。

微软早就和别的一些公司一起致力于一个被称为 SOAP 的协议的开发，该协议利用 XML，基于 HTTP 和 TCP/IP，通过本地连接交换数据。然后 IBM 参加了进来，在 2000 年秋宣布发表了另外两个标准：Web Services 描述语言（WSDL），以及通用描述、发现和集成协议（UDDI）。2000 年末，Oracle、HP 和 Sun 都宣布要在自己的产品中支持和部署 Web Services（对 Web Services 历史的更多细节请见[Levitt01]）。

如今，基于几个标准，为许多公司和标准化组织所推动，Web Services 已经成为一个主要的潮流了，标准的名字通常以 WS 开头，基于大约 10 个底层标准，如 XML 和 HTTP。

Web Services 的标准超过了 50 个，并且还加上了大约 10 个由不同的标准化组织——如万维网联盟（W3C）、OASIS 和 Web Services 互操作性组织（WS-I）——阐述的配套资料。这些 Web Services 标准覆盖了几乎分布式计算和远程过程/函数/服务调用的所有领域，如安全（见 14.4 节）、事务性、可靠性、流程建模（如 BPEL，见 7.4.2 节），以及服务管理（见第 17 章），还有几个规定诸如错误处理、寻址等技术细节的小标准。

16.2 基础 Web Services 标准

Fundamental Web Services Standards

Web Services 有 5 个基础标准，其中两个是早就有了的通用标准，它们被用来实现 Web Services 方法：

- *XML* 是用来描述模型、格式和数据类型的通用格式，其他大多数标准都是用 XML 来表达的标准。实际上，Web Services 所有标准都基于 XML1.0、XSD（XML 大纲定义），以及 XML 命名空间。
- *HTTP*（包括 HTTPS）是互联网使用的底层协议。HTTP（S）是运用了互联网技术，通过网络发送 Web Services 的可能的协议之一。

其他 3 个基础标准是专用于 Web Services 的，也是最早规定的标准：

- *WSDL* 用来定义服务接口。实际上，它能描述服务的两个不同方面：服务的签名（名字和参数），以及服务的绑定和部署细节（协议和位置）。
- *SOAP* 是定义 Web Services 协议的标准。HTTP 是一个同时用于互联网的底层协议，而 SOAP 是 Web Services 的专用格式，通过此协议交换 Web Services 数据。
- *UDDI* 是管理 Web Services 的标准（即注册和找到服务）。

通常说来，使用 WSDL 标准是 Web Services 的关键特征，其他所有东西都是可选的。例如，你不一定非得使用 SOAP 和 HTTP 来发送服务请求，你可以使用其他协议而仍然算在用 Web Services。另外，UDDI 只扮演着补充的角色，既非必需，而且通常在实践中也不用（细节见 16.2.4 节和第 17 章）。

让我们再看看这 3 个标准更多一点的细节。

16.2.1 WSDL

为了帮助你理解 WSDL 文件都讲了些什么，我们从观察 WSDL 文件的一般结构开始。注意，WSDL 标准有不同的版本，写本书的时候，即 2007 年，WSDL1.1 是普遍使用的版本，但是，WSDL2.0 已经起步了（其状态为候选推荐）。

图 16-1 展示了两个版本的一般结构，它们是相似的，虽然并不完全相同。

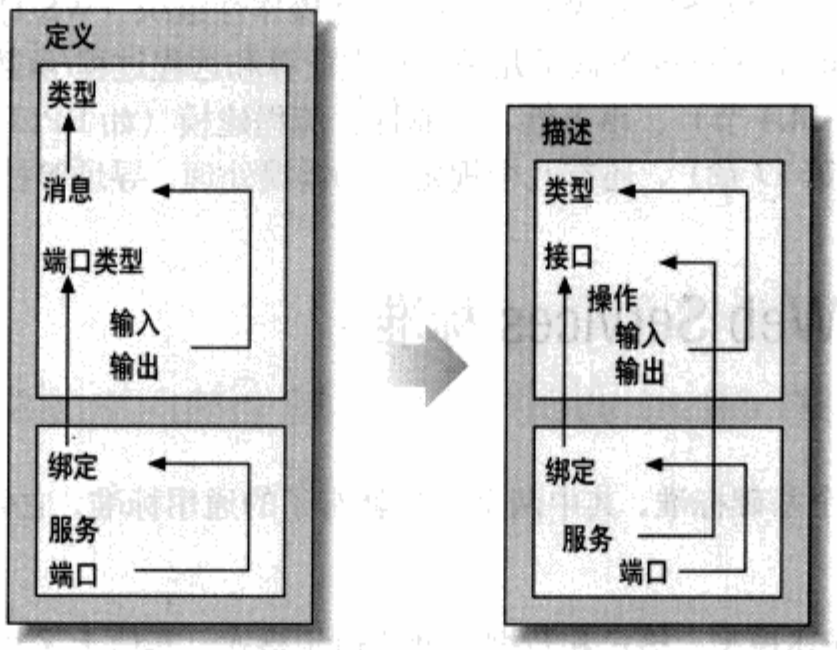


图 16-1：WSDL1.1 和 WSDL2.0 文件的一般结构

WSDL 文件自底向上描述服务，这就是说，它们从用到的数据类型开始，到服务的位置（地址）结束。另外，它们有 3 个描述的层次：

- 第一个层次描述服务的接口。这个接口（在 WSDL1.1 中称为“端口类型”）可以包含一个或多个有输入参数和输出参数的操作，参数使用在 WSDL 文件第一节中指定的类型。在 WSDL1.1 中，服务参数在<message>节中定义，而在 WSDL2.0 中，它们和任何其他类型一样，定义在<types>节中。
- 第二个层次定义 Web Services 的“绑定”，即为 Web Services 操作提供的协议和格式。
- 第三个层次定义可以找到 Web Services 的物理位置（地址、URL）。

要注意，由于 WSDL 文件内部有不同的层次，因此，不同的团队可在不同的时候修改 WSDL 文件的内容。细节见 16.3.3 节。

16.2.2 WSDL 示例
WSDL by Example

让我们看一个 WSDL 文件的简单示例。这个 WSDL 文件定义了一个被称为 CustomerService 的服务，该服务提供了一个被称为 getCustomerAddress()的操作。操作的输入参数是一个类型为 long 的客户 ID，输出的是一个包含 3 个 string 属性——街道、城市和邮编——的结构。

如下为 WSDL1.1 版本：

```
1   <?xml version="1.0" encoding="utf-8" ?>
2   <definitions name="CustomerService"
```

```
3        targetNamespace="http://soa-in-practice.com/wsdl"
4        xmlns:tns="http://soa-in-practice.com/wsdl"
5        xmlns:xsd1="http://soa-in-practice.com/xsd"
6        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8        xmlns="http://schemas.xmlsoap.org/wsdl/">
9
10     <types>
11       <xsd:schema
12          targetNamespace="http://soa-in-practice.com/xsd"
13          xmlns="http://soa-in-practice.com/xsd">
14
15         <xsd:element name="getCustomerAddress">
16           <xsd:complexType>
17             <xsd:sequence>
18               <xsd:element name="customerID" type="xsd:long"/>
19             </xsd:sequence>
20           </xsd:complexType>
21         </xsd:element>
22
23         <xsd:element name="getCustomerAddressResponse" type="Address"/>
24         <xsd:complexType name="Address">
25           <xsd:sequence>
26             <xsd:element name="street" type="xsd:string"/>
27             <xsd:element name="city" type="xsd:string"/>
28             <xsd:element name="zipCode" type="xsd:string"/>
29           </xsd:sequence>
30         </xsd:complexType>
31
32       </xsd:schema>
33     </types>
34
35     <message name="getCustomerAddressInput">
36       <part name="params" element="xsd1:getCustomerAddress"/>
37     </message>
38     <message name="getCustomerAddressOutput">
39       <part name="params" element="xsd1:getCustomerAddressResponse"/>
40     </message>
41
42     <portType name="CustomerInterface" >
43       <operation name="getCustomerAddress">
44         <input message="tns:getCustomerAddressInput" />
45         <output message="tns:getCustomerAddressOutput" />
46       </operation>
47     </portType>
48
49     <binding name="CustomerSOAPBinding"
50              type="tns:CustomerInterface" >
51       <soap:binding style="document"
52          transport="http://schemas.xmlsoap.org/soap/http" />
53       <operation name="getCustomerAddress">
54         <soap:operation
55           soapAction="http://soa-in-practice.com/getCustomerAddress" />
56         <input>
57           <soap:body use="literal" />
58         </input>
```

```
59        <output>
60          <soap:body use="literal" />
61        </output>
62      </operation>
63    </binding>
64
65    <service name="CustomerService" >
66      <port name="CustomerPort"
67           binding="tns:CustomerSOAPBinding">
68        <soap:address
69           location="http://soa-in-practice.com/customer11"/>
70      </port>
71    </service>
72
73  </definitions>
```

让我们从后向前过一遍：

- 最后的<service>节定义了一个被命名为 CustomerService 的服务。该服务可以从 *http://soa-in-practice.com/customer11* 得到，与其一同提供的有被称为 CustomerSOAPBinding 的绑定。前缀 tns:是可以找到这个绑定标识符细节的命名空间，被定义在文档的最开始，与目标命名空间一样，而目标命名空间是本文件中所有能识别的标识符所属的命名空间（见第 3 行和第 4 行）。这就意味着 CustomerSOAPBinding 是在本文件中定义的一个标识符。

- <binding>节定义了用来提供服务的协议和格式。在这里，我们看到了 CustomerSOAPBinding 的定义，此定义从指明该绑定用于哪个接口类型（CustomerInterface）开始，这个绑定是一个基于底层 HTTP 协议的 SOAP 绑定（见第 51 行和第 52 行），在此我们不详细讲述。本节也定义了使用 SOAP 的风格（是的，有不止一种 SOAP 绑定可用！）。

- <portType>节定义了接口 CustomerInterface。它包含了一个叫 getCustomerAddress 的操作，指明了当操作被调用时通过服务总线发送的消息。这个服务发送一个请求，收到一个应答（对消息交换模式的讨论见第 10 章），这是通过指定一个输入消息（getCustomerAddressInput）和一个输出消息（getCustomerAddressOutput）来定义的。其他的消息是错误消息。

- <message>节定义了各个消息，使用的是在<portType>节引用的标识符。不管是 getCustomerAddressInput 还是 getCustomerAddressOutput 消息，都使用了在<types>节中定义的数据类型。

- <types>节定义了将会使用到的数据类型：输入参数 customerID 的类型是 long，输出参数 address 的类型是有 3 个字符串属性的结构/记录。所有类型在自己的命名空间 xsd1 中。

<binding>节指出这里使用的 SOAP 风格为 document/literal，这是第 51 行 style 属性与第 57 行、第 60 行 body 定义的结合，实际的风格应为 document/literal wrapped，因为它使

用了额外的约定（输入消息有一个单独的部分引用了一个元素，该元素与操作同名，并且该元素的复杂类型没有属性）。经过这些年的发展，这已经演化成了流行的 SOAP 绑定方式。

注意

WSDL1.1 的细节见[WSDL1.1]，不同 SOAP 风格的细节见[Butek03]。

版本 2.0 对应的 WSDL 文件（使用 SOAP1.2）如下所示：

```
1   <?xml version="1.0" encoding="utf-8" ?>
2   <description
3       targetNamespace="http://soa-in-practice.com/wsdl"
4       xmlns:tns="http://soa-in-practice.com/wsdl"
5       xmlns:xsd1="http://soa-in-practice.com/xsd"
6       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7       xmlns:wsoap="http://www.w3.org/2006/01/wsdl/soap"
8       xmlns:wsdlx="http://www.w3.org/2006/01/wsdl-extensions"
9       xmlns="http://www.w3.org/2006/01/wsdl">
10
11    <types>
12      <xsd:schema
13          targetNamespace="http://soa-in-practice.com/xsd"
14          xmlns="http://soa-in-practice.com/xsd">
15
16        <xsd:element name="getCustomerAddress">
17          <xsd:complexType>
18            <xsd:sequence>
19              <xsd:element name="customerID" type="xsd:long"/>
20            </xsd:sequence>
21          </xsd:complexType>
22        </xsd:element>
23
24        <xsd:element name="getCustomerAddressResponse" type="Address"/>
25        <xsd:complexType name="Address">
26          <xsd:sequence>
27            <xsd:element name="street" type="xsd:string"/>
28            <xsd:element name="city" type="xsd:string"/>
29            <xsd:element name="zipCode" type="xsd:string"/>
30          </xsd:sequence>
31        </xsd:complexType>
32
33      </xsd:schema>
34    </types>
35
36    <interface name = "CustomerInterface" >
37
38      <operation name="getCustomerAddress"
39                 pattern="http://www.w3.org/2006/01/wsdl/in-out"
40                 style="http://www.w3.org/2006/01/wsdl/style/iri"
41                 wsdlx:safe = "true">
42        <input messageLabel="In"
43               element="xsd1:getCustomerAddress" />
```

```
44        <output messageLabel="Out"
45                element="xsd1:getCustomerAddressResponse" />
46      </operation>
47
48    </interface>
49
50    <binding name="CustomerSOAPBinding"
51            interface="tns:CustomerInterface"
52            type="http://www.w3.org/2006/01/wsdl/soap"
53            wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">
54
55      <operation ref="tns:getCustomerAddress"
56                wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
57
58    </binding>
59
60    <service name="CustomerService"
61            interface="tns:CustomerInterface">
62
63      <endpoint name="CustomerEndpoint"
64                binding="tns:CustomerSOAPBinding"
65                address="http://soa-in-practice.com/customer20"/>
66
67    </service>
68
69  </description>
```

与 WSDL1.1 主要的区别是：

- XML 根元素的名字是<description>而非<definitions>。
- <service>节使用的是“endpoint”而非“port”。
- 在<binding>节内，第 52 行、第 53 行和第 56 行定义了特定的协议（SOAP1.2、基于 HTTP，使用 HTTP GET）。
- <portType>节被<interface>节所取代，使用了更具体的、定义了底层消息顺序的消息交换模式（见第 10 章）。
- <message>节没有了，操作直接引用数据类型（定义在<types>节中），发送哪个消息遵循<interface>节中指定的类型和消息交换模式（MEP）。
- 用不同的命名空间来标识 WSDL2.0 和 SOAP1.2。

注意，细节见[WSDL2.0]。

第一层（WSDL1.1 的类型、消息和端口类型，WSDL2.0 的类型和接口）是 Web Services 的语法接口，指定了服务的签名。其他的层定义技术细节，如服务的绑定和位置。

这些例子忽略了错误处理（通常的做法是提供和输入、输出消息类似的报错消息）。

16.2.3 SOAP

如前所述，SOA 是第一个开发出来的真实的 Web Services 标准。SOAP 最初是“简单对象访问协议”的首字母缩写词，然而，后来开发者发现 SOAP 既不简单，也不处理对象访问。所以，从 1.2 版开始，这个缩写词就被当作一个词使用，不再代表一种缩写了。

当前 SOAP 版本为 1.1 和 1.2，但是，它们之间的差异不像 WSDL 版本之间的那样大。该规范主要与 SOAP 适配器的编写相关，Web Services 程序员一般不容易见到它。

根据前面列出的 WSDL 文件（使用 document/literal wrapped 风格），是其 SOAP 请求的一个例子，如下：

```
<?xml version='1.0' ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    ...
  </soap:Header>
  <soap:Body>
    <getCustomerAddress xmlns="http://soa-in-practice.com/xsd">
      <customerID>12345678</customerID>
    </getCustomerAddress >
  </soap:Body>
</soap:Envelope>
```

一个相应的应答消息如下：

```
<?xml version='1.0' ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    ...
  </soap:Header>
  <soap:Body>
    <getCustomerAddressResponse xmlns="http://soa-in-practice.com/xsd">
      <address>
        <street>Gaussstr. 29</street>
        <city>Braunschweig</city>
        <zipCode>D-38106</zipCode>
      </address>
    </getCustomerAddressResponse>
  </soap:Body>
</soap:Envelope>
```

可以看到，SOAP 消息格式为 XML，包含一个被称为<Envelope>的根元素。根元素包含一个可选的<Header>和必备的<Body>元素，Body 元素包含有效载荷（请求、回复，或者错误数据），Header 包含额外的、帮助基础设施处理消息的信息（如路由提示、安全提示等）。

注意，这里所记录的风格（document/literal wrapped）用服务名作为文档体的开始（对应答消息用“Response”字符串进行扩展），而且，传输的数据没有任何格式信息。也

有一些其他的 SOAP 风格允许你随着数据传送数据类型。没有类型编码信息的好处是，就（向后兼容）修改接口类型来说，这个方法更具通用性，有助于处理服务的不同版本。

当通过 HTTP 发送 SOAP 请求时，使用了通常的 HTTP 头部。例如，请求可以如下（SOAP1.1）：

```
POST /customer11 HTTP/1.1
Host: soa-in-practice.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: " http://soa-in-practice.com/getCustomerAddress"

<?xml version='1.0' ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  ...
</soap:Envelope>
```

一个相应的应答如下：

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

<?xml version='1.0' ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  ...
</soap:Envelope>
```

16.2.4 UDDI

“通用描述、发现和集成（UDDI）”最初是更宽泛的术语“通用描述、发现和集成业务注册中心”（或短些的“UDDI 业务注册中心”，或者更短的 UBR）的一部分。最初的想法是，引入一个有效的市场所包含的全部 3 个角色：提供服务的供应者、需要服务的消费者，以及通过广告和定位服务把两方撮合到一起的中间人。这就产生了解释 Web Services 通常都要展示的著名的三角形（见图 16-2）。

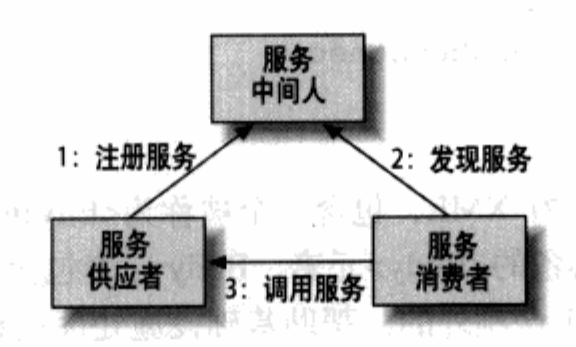


图 16-2：供应者、消费者和中间人

UBR 的意图是，它将成为 Web Services 世界市场上位于中央的中间人（就像 Web Services 的世界黄页一样），供应者把新 Web Services 服务放到 UBR 中，消费者可以搜索UBR，找到它们想要的任何服务。由此UDDI打算成为用来注册和找到Web Services的协议。

然而，后来开发者发现 UBR 没法工作，于是他们于 2006 年 1 月关闭了 UBR。官方发表的原因如下（见[UBR]）：

> IBM、微软和 SAP 评估了 UDDI 业务注册中心的状况，结论是该项目的目标已经达到。鉴于此，UDDI 业务注册中心将于 2006 年 1 月 12 日停止运行。

这真是篡改历史的一个绝佳例子。UBR 的供应者们不是承认“做 Web Services 的全球黄页”的想法行不通，而是反过来重新定义了 UBR 背后最初的动机，宣称 UBR 的目的就是作为一个原型，用来学习如何为真实的、位于中央的服务中间人设计一个合适的接口。

UBR 的失败并不一定意味着 UDDI 也失败了，但是，对于 UDDI 在实践中的重要性，人们存在怀疑，部分原因是服务管理要求有大量的、有实用价值的服务信息详细目录，然而，另一部分原因仅仅是因为 UDDI 自己的功课做得不足，引用[NewcomerLomow05]如下：

> 然而，由于许多原因，其中包括很难给互联网创建有意义的分类信息，UDDI 没有达到其目标，即成为所有 Web Services 元数据的注册中心。对通过互联网进行的大多数 Web Services 交互来说，UDDI 也没有成为有用的东西。

但是，由于技术方面的原因，Web Services 通常确实需要一个中间人。Web Services 内生是点到点的连接，需要某种机制和/或过程来避免在 Web Services 消费者里面进行硬编码寻址（见 5.3 节）。于是，出现了其他解决方案。再次引用[NewcomerLomow05]如下：

> 一般说来，看起来有的公司在使用 UDDI，但是，其他公司在扩展既存的注册中心解决方案，如 LDAP、Java 命名与目录接口（JNDI）、关系型数据库，或者 CORBA 交易者，还有另一个用得少些的选择是 ebXML 注册中心。

UDDI 现在有多个版本，并且，其他即将出现的 Web Services 标准也与其部分重叠。[NewcomerLomow05]的说法如下：

> 这些规范间的重叠清楚地展示出 Web Services 标准化过程之所以需要领导和架构的又一个原因。

总的说来，UDDI 的整个情形看起来一片混乱，时间会告诉我们事情将如何发展。

16.3 实践中的 Web Services
Web Services in Practice

笔者已经介绍了使 Web Services 比它们初看起来更复杂的某些方面，在本节中，笔者将详细讨论它们。

16.3.1 开放的标准化
Open Standardization

Web Services 的第一个问题是，它不是一个由单个标准化组织定义的标准。如前所述，有超过 50 个不同的 Web Services 规范，分别由三家不同的标准化组织制定（W3C、OASIS 和 WS-I）。图 16-3 给出了一个简要的概览（注 1）。

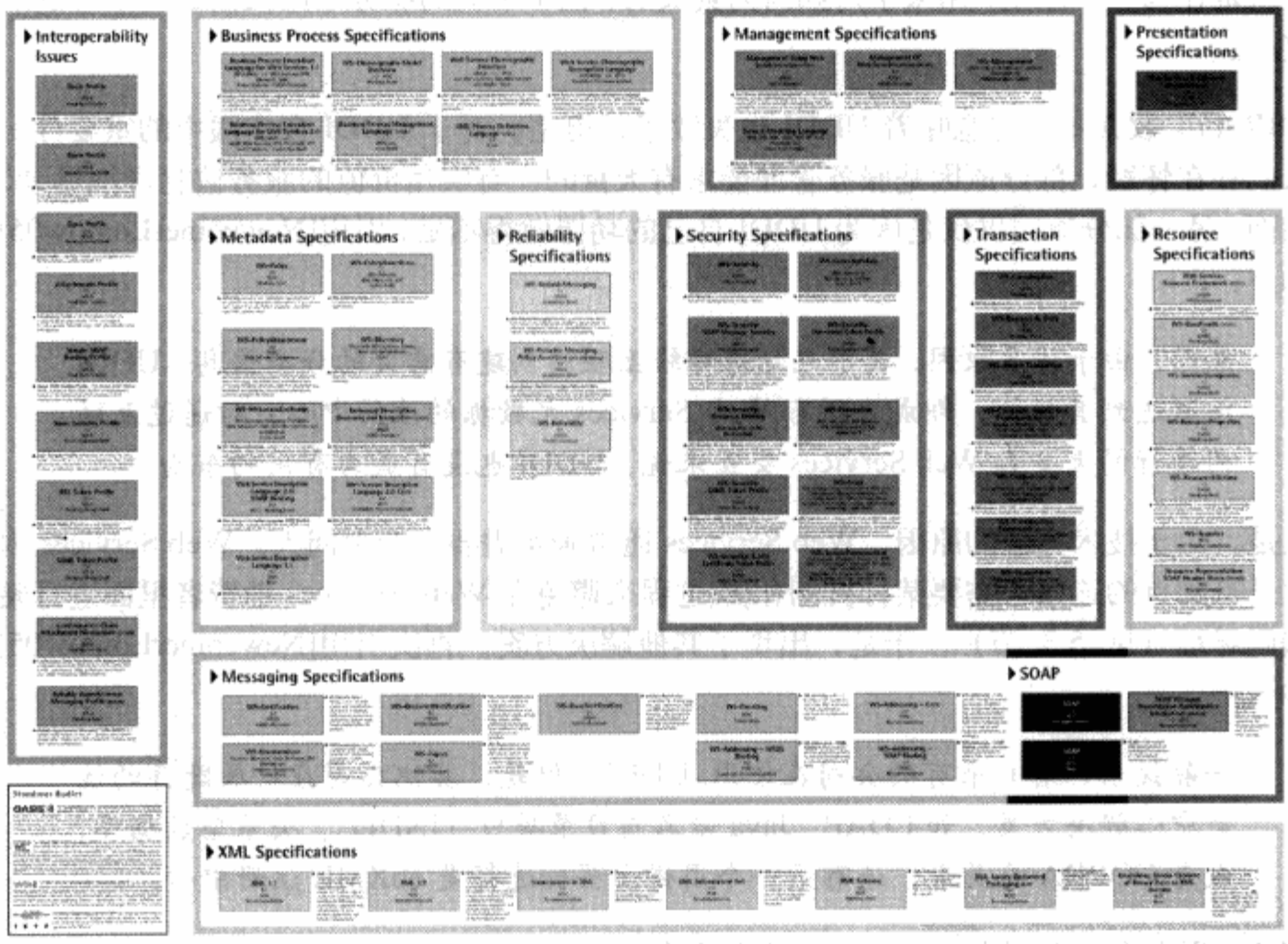


图 16-3：Web Services 标准概览

假设你是一家工具供应商，想为 Web Services 提供工具，你应该遵循这些标准中的哪一个？并且，一旦你决定了遵循的标准，对该标准来说，你应该遵循哪个版本？这简直就是一团糟，互操作性仍然是个现实的问题。

对于前面提到的 5 个基础标准（XML、HTTP、WSDL、SOAP 和 UDDI）来说，一直以来，达到互操作性的目标也很困难。即使你选择了这些标准的同一版本，规范也太粗

注 1：感谢 innoQ 允许我复制这幅概览图。你可以在[WSStandards]找到原图和一份对 Web Services 的描述。

糙、太宽泛。因此，发生了一些有趣的事情：2002 年，几家供应商（Accenture、BEA、Fujitsu、HP、IBM、Intel、微软、Oracle 和 SAP）聚到一起，成立了一家对这些标准进行标准化的标准化组织。这个组织被称为 Web Services 互操作性组织（WS-I）。在 2006 年，WS-I 已经有超过 130 个会员，包括 Sun（注 2）（WS-I 的更多细节，见[WS-I]）。

WS-I 创造的一个重要东西是“配套资料”，它是一整套已经定义了的标准，每一个标准都指定了特定的版本，再加上使用这些标准的指南和约定，最后以保证互操作性的方式把所有东西结合起来。2004 年，WS-I 基本配套资料 1.1 得以确定（针对 5 个基础 Web Services 标准：XML1.0、WSDL1.1、SOAP1.1、HTTP1.1，以及 UDDI 版本 2），为调用 Web Services 提供互操作性。这个基本配套资料包含 150 个额外的一致性要求，以及 24 个扩展点，从而对基础的 5 个标准提供额外的限制和扩展。细节见[WSI-BasicProfile]。

要注意的是，基本配套资料 1.1 仅仅覆盖了超过 50 个标准中的 5 个，所以，对其他标准而言（这些标准覆盖了诸如安全性、可靠性、流程建模和配制、处理服务等级协议，以及服务管理等方面），互操作性仍然是个问题。覆盖更多标准的基本配套资料的未来版本将接踵而至，并且，其他配套资料会覆盖诸如安全和可靠性这样的主题。实际上，已经有了一个 WS-I 基本安全配套资料（见 14.4.2 节）。在实践中，只要有配套资料可用，你都应该永远遵守它们。但是，就 Web Services 的一般情况而言，互操作性仍然是一个必须要考虑的问题。

16.3.2 协议考量

Protocol Aspects

我想要提的下一个关于 Web Services 的考量基于如下事实：Web Services 内生地是点到点的，并且是协议驱动的。

正如 5.3.3 节解释过的，一个协议驱动的 ESB 有其优点：每个消费者和供应者都可以使用各自的工具来调用和实现 Web Services。唯一必须保证的是协议的要求被适当地满足了。不幸的是，正如刚刚讨论的，对 Web Services 来说互操作性还是个问题。结果就是，仅仅因为消费者 A 和消费者 B 能与供应者 P 互操作，以及消费者 A 能与供应者 Q 互操作，不能由此推出消费者 B 也能与供应者 Q 互操作。换句话说，你必须就每对供应者和消费者的组合进行验证。这意味着你不是去验证每个系统的一个接口，而是你必须去验证接口的 $n \times (n-1)$的组合。

点到点协议的一个重要考量是你需要确保服务供应者的地址没有被硬编码在消费者代码中（见 5.3.1 节）。为此，你可以使用 UDDI（它的缺点我们在前面已经讨论过），或者你也可以使用其他解决方案，但是，为了可伸缩性和容错，你需要“某种”解决方案。

注 2：WS-I 的创立者名单中没有 Sun，这是个有意的选择，因为微软参加的前提是“Sun 不能作为提议者/宣告者/创立成员”之一（见[Wong02]）。

本质上，我强烈建议当你在建立 Web Services SOA 景观时，就在其中植入对这些问题的适当的解决方案。最好的解决方案是建立这样的事实：ESB 将 Web Services 用作基础设施只是个实现细节，并且，ESB 给出了消费者和供应者调用和实现服务所需要的一切。这就是说，从供应者和消费者的角度来看，ESB 应该是 API 驱动的，任何修改，比如引入拦截器、改变 WSDL 或 SOAP 的版本等，就都成为透明的了。（对 API 驱动的 ESB 和拦截器的细节见 5.3 节）

Web Services 一个典型的可伸缩的方法看起来可能如图 16-4 所示：

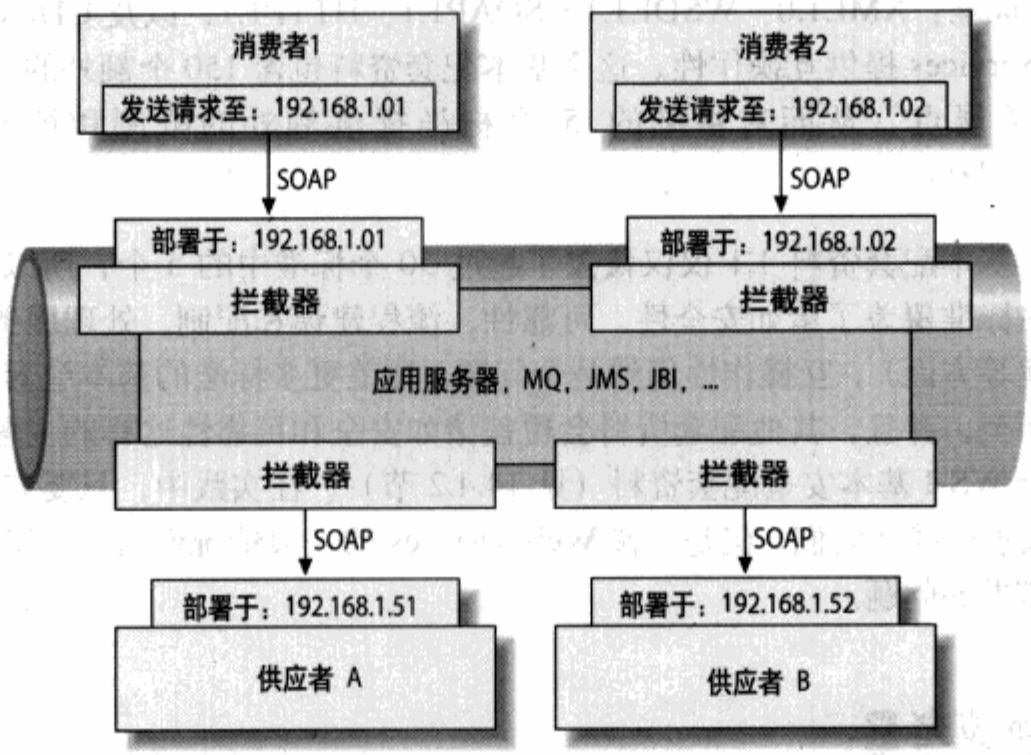


图 16-4：一个 Web Services ESB 在内部可能使用不同的协议

在 ESB 之外，拦截器（见 5.3.2 节）提供了一个 Web Services 的接口（使用 SOAP 协议），然而，在 ESB 内部，用某种其他标准或专有技术来实现。例如，一系列的应用服务器可能被当作星形拓扑的中央服务器，通过一个特定的 JMS 协议连接。每个应用服务器都提供了对特定供应者集合和消费者集合的连通性，并且，或者向内部，或者向其他服务器路由消息（例如，每个应用服务器可能负责一个物理地址）。

用这种方法也可以顺便解决内生的安全（见第 14 章）和可靠性（见第 10 章）问题，每个供应者和消费者都使用一个特定的 HTTPS 通道，并且在总线内部，比起 HTTP 所提供的可靠性来说，所使用的其他协议的可靠性可能更有保证。

要注意的是，你可以采取一种渐进的方法；当开始 Web Services 时，你只要求有某种间接性，这样消费者就不必使用某个具体的供应者指定的端点。为达到此目的，你应该反过来定义一个位置，消费者可以在此找到要使用的实际端点。最开始时，你可以把供应者的端点放到这个位置（这样一来服务调用就直接连到供应者那里），之后当你引入拦

截器时，你可以把拦截器的端点放到这个位置，如此一来拦截器就能相应地调用路由服务。随着时间的推移，你可以指定关于如何实现 ESB 的更多细节，提供越来越多的增值 ESB 服务（见 5.4 节）。渐进地提供增值 ESB 服务使每个人都能从经验中学习到什么是必需的，什么是恰当的。

16.3.3 WSDL 生命周期
The WSDL Lifecycle

正如 15 章所述，WSDL 文件内部有不同的层，所以不同的团队可以在不同的时间对这些文件的内容进行修改。这就意味着，从开发的角度来看，没有描述 Web Services 的“单个”WSDL 文件，从消费者的角度来看，你可能看到的是 WSDL 文件的最终版本，处于生产状况中，包含了调用一个 Web Services 所需要的所有信息（但是，即使在那时，当使用不同拦截器时，正如刚刚讨论的，端点也可能不同）。如此说来，至少从开发的角度来看，WSDL 文件有生命周期，其包含的步骤如表 16-1 所示。这个生命周期与第 11 章讨论的服务生命周期相对应。

表 16-1：WSDL 生命周期

层	WSDL1.1	WSDL2.0	内容明确于何时	由谁定义
服务接口	<types>, <message>, <portType>	<types> <interface>	设计解决方案时	解决方案经理，业务流程经理，服务信息项目经理……
有命名空间和其他细节的服务接口	<types>, <message>, <portType>	<types> <interface>	设计和/或实现服务时	服务设计师和/或实现者
协议	<binding>	<binding>	配置时	基础设施(ESB)团队
位置	<service>	<service>	部署时，运行时	操作者

各阶段详述如下：

- 最初，你使用来指定服务接口的 WSDL 文件可能只用了文件的第一层（<types>节和<message>/<portType>或<interface>节），当解决方案经理或其他负责服务设计的高层设计师将服务识别为一个业务流程或解决方案的一部分时，典型情况下是这样做的（见第 7 章和 11.1.2 节）。然而，要注意 WSDL 也许没有覆盖全部服务契约（见 16.3.4 节，“WSDL 的缺陷”）。

- 要把服务集成为 SOA 景观的一部分，你必须处理应用到特定基础设施上的惯例和要求，需要引入命名空间，使用特定的数据类型，等等。通常说来，对最初的、第一份设计的精化由服务设计师和/或服务实现者在设计和/或实现服务时完成（见 11.1.1 节）。

- 被用来提供和调用服务的协议依赖于使用的基础设施，因此，当服务的基础设施细节明朗时，<binding>节就发挥作用了。一般来说，这发生在实现和配置时（具体何时发生依赖于是由实现者填充相应数据，还是由一个配置工具恰好在部署前做这事）。

- 最后，你必须定义在哪里可以获得服务。这通常在部署时和/或运行时指定（即使用一个测试环境一个生产环境已经明朗时）。

因为用于服务的 WSDL 文件经历了不同的阶段，所以，你应该设立自己的过程、文档，以及配置管理过程，这样才不容易混淆。这些可以由生成器提供部分支持，并且可能是模型驱动开发过程的一个步骤（见第 18 章）。

图 16-5 给出了一个例子，说明了实践中的 WSDL 生命周期可能是怎样的。在设计时，某人定义了描述一个新服务的 WSDL 文件的第一个版本。虽然内部团队应该遵守通常的命名约定，然而，外部团队可能不会遵守这些约定，因此，你需要某些映射才能把外部创建的文件集成到你的 SOA 基础设施中，这可能也是你根据基础设施和政策定义绑定的时间点，然后，不同的团队才能从这些与 SOA 景观一致的 WSDL 文件生成出代码，并且根据确定出的 API 进行实现。然而，在部署时（这可能在集成测试期间，或者当服务进入生产阶段时），使用的是补充了部署细节的 WSDL 文件来提供可连接性，因此要注意的是，如果你使用了拦截器，那么，不同的消费者可能使用不同的 WSDL 文件来调用服务。同时，将内部技术要求和约定映射到不遵守这些东西的外部服务供应者上，可能也会产生一些工作量。这件事应该不是太难。

16.3.4 WSDL 的缺陷
WSDL Deficits

你还应该理解的是，WSDL 文件没有包含一个服务的完整接口，即它没有包含任何语义信息，这就意味着 WSDL 文件不是被提供来管理整个服务契约的，它仅仅在调用 Web Services 时，对技术给予明确的格式，因此，将这个格式用于服务业务库毫无意义，WSDL 只是被提供来用于注册中心的（业务库和注册中心的更多信息请见第 17 章）。

令人惊讶的是，就服务的技术特性而言，WSDL 也有缺陷，一个严重的缺陷（至少到 WSDL1.1 都是如此）是缺乏非功能性特性和服务等级协议特性。一个 WSDL 文件没有指明服务通常运行多长时间，谁能调用服务，服务调用的成本有多大，等等。通常说来，要在一个大型 SOA 景观中管理服务，必须知道这些方面。

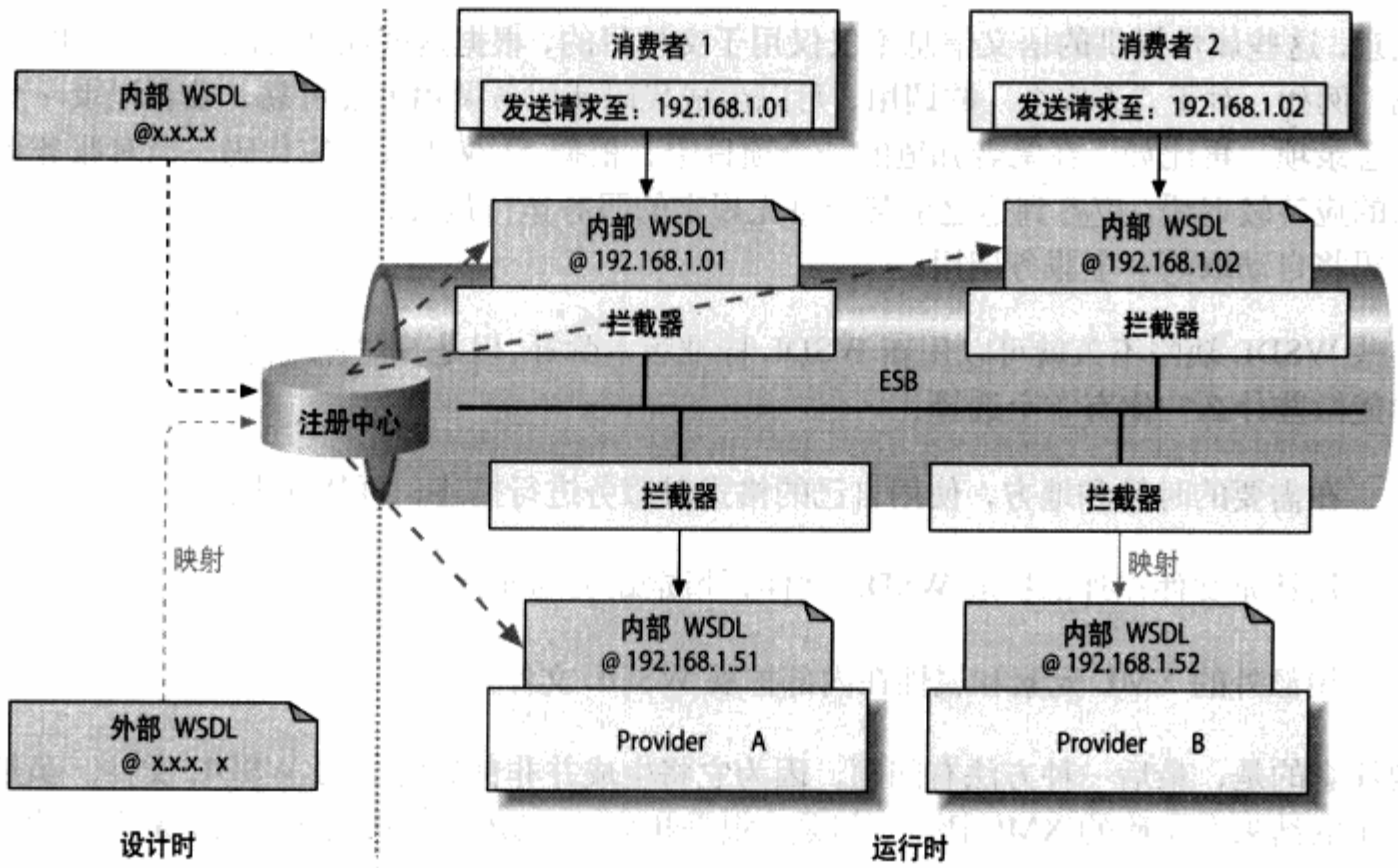


图 16-5：WSDL 生命周期示例

在 WSDL 的未来版本中，这种情况可能会发生变化。例如，WSDL2.0 提供了扩展 WSDL 文件的功能。我们在本章前面看过的 WSDL2.0 文件已经展示了一个可能的扩展，WSDL 扩展“safe”定义了一个服务调用对消费者来说是否是“安全”的，“安全”的意义是消费者除了与供应者发生交互外，不会承担任何别的负担：

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    ...
    xmlns:wsdlx="http://www.w3.org/2006/01/wsdl-extensions"
    ...>
  ...
  <interface name = "CustomerInterface" >

    <operation name="getCustomerAddress"
              pattern="http://www.w3.org/2006/01/wsdl/in-out"
              style="http://www.w3.org/2006/01/wsdl/style/iri"
              wsdlx:safe = "true">
        <input messageLabel="In"
              element="xsd1:getCustomerAddress" />
        <output messageLabel="Out"
               element="xsd1:getCustomerAddressResponse" />
    </operation>

  </interface>
```

一个典型的“不安全”的服务调用是一个订单或一个预约。

这只是个例子，但是，它展示了这个概念。

注意，这些属性提供的语义信息不仅仅用于文档目的，根据这些属性可以生成不同的代码。例如，对于“不安全”的调用，可以生成代码使服务调用更加可靠，或者记录一个备忘录项。再比如，在笔者知道的一个项目中，根据 SLA 生成相应代码，当对服务请求的应答较迟缓，应答到达之前某个事先规定的服务运行最大时间已经用完时，生成的代码将自动取消一个服务调用。

某些 WSDL 缺陷不久就可以用新 WSDL 标准版本弥补，但是，那些不能弥补的怎么办？你能做些什么？你有 3 个选择：

- 在需要的时候和地方，使用自己的格式对服务进行描述，再从这里面生成 WSDL。
- 用补充文件在外部扩展 WSDL 文件，指定缺失的属性。
- 用额外的 XML 元素和属性在内部扩展 WSDL 文件。

要注意的是，最后一种方法有问题，因为它将生成并非良好定义的 WSDL 文件。虽然产生的结果有正确的 XML 语法，但是，根据相应的 WSDL 格式描述，它不是有效的文件（一般说来，增加新的属性会破坏被指定了格式的 XML 文件的有效性）。

原则上，笔者推荐第一种方法，它的优点是：特定于基础设施的考量不会太早掺和进来。而且，当 Web Services 标准改善时，它也使你能够根据需要，为 WSDL 文件生成额外的信息。关于定义你自己的服务模型和符号，更多的内容请见 18.3 节。

16.3.5 那么，你应该使用 Web Services 吗？

So, Should You Use Web Services?

现在，对应用 Web Services 涉及的几个问题（或“挑战”），我们已经读过如下这些：

- WSDL 文件有生命周期，所以你需要一个过程。
- WSDL 文件缺乏一些属性。
- 过多的标准阻碍了互操作性。
- 缺乏互操作性使得针对每对供应者和消费者的组合进行验证成为必要。
- Web Services 协议内生的点到点性质使得拦截器成为必要。
- 分别验证和拦截器所要求花费的工作量导致一个基于 API 的 ESB。

那么，这是否意味着你应该放弃使用 Web Services 呢？

好的，让我们来看看其他选择：

- 你可以使用一个专有的 SOA 基础设施解决方案，它更一致、更成熟，并且解决了 Web Services 固有的问题。但是，和所有专有解决方案一样，它可能使你失去控制，无法进行自由选择，变得依赖于特定的供应商。
- 你可以靠自己从头创造一切，这意味着重新发明轮子，以及再次犯别人犯过的错误。

当然，你的决策将取决于你自己的特定情况，但是，对即将出现的标准给予较高的优先级不是个坏主意。同时请注意，要靠你自己决定在多深的层次上接纳 Web Services 标准。例如：

- 你可以只把 WSDL 用作服务接口的格式（带有或不带额外的属性），使用你自己特定的协议。
- 你可以使用一个运用了 BPEL 的流程引擎，使用你自己的 ESB 协议。
- 你可以用 SLA 属性扩展 Web Services 文件。
- 你可以在 Web Services 上加以更多的限制，以提供更好的互操作性。

本章重要的教训是，使用 Web Services 不会自动解决你所有的问题。你将需要提供人、时间和资金来使 Web Services 成为自己 SOA 景观中有效的基础设施。

请做好你的功课，如果基础设施团队提供了一个半成熟的解决方案，它带来的只会是挫折和无知。如果大家不想使用提供的基础设施，这永远都是基础设施团队的错。你应该使人们对你的 SOA 基础设施感到兴奋（见 19.2.2 节）。

本章的焦点是使用 Web Services 作为你的技术基础设施，但是，当然地，这只是一种选择。让整个 SOA 战略屈从在 Web Services 标准之下是非常冒险的、短视的方法，Web Services 只是一个可能的基础设施，并且，即使你没有从一开始就为自己的 SOA 准备多个基础设施（见 5.2 节），你也不能指望 Web Services 将是自己实现过的唯一基础设施（标准）。

因此，在独立于基础设施的层上使用 Web Services 标准时要小心了，实际上，服务契约（对服务的整体描述）应该以独立于任何特定基础设施的格式来描述和管理，这就意味着服务业务库（与注册中心不同，见 17.2 节）不应只是管理 WSDL 文件的工具。

同时注意，使用 Web Services 没有自动意味着你在实现 SOA，它可能只是让某些系统交互的方法，没有提供和要求 SOA 的其他所有方面。最近，针对像这样应用的 Web Services，演化出了一个很棒的缩写词：JaBoWS（译注 1），其意思是“仅仅是另一捆 Web Services”。

译注 1：JaBoWS，Just another Bunch of Web Services 的缩写。

16.4 结束按语
Closing Notes

经历了所有这些后，让我们回到 Web Services 到底是什么这个问题上。在这儿，笔者想指出的重要的一点是：Web Services 标准扮演了两个不同的角色。

- 一方面，它们为服务的抽象描述指定了标准。这些标准可能是接口描述，如 WSDL 文件中的第一层；或者是服务配制和工作流，如 BPEL 中的第一层（见第 7 章）。
- 另一方面，它们为 SOA 基础设施（ESB）的具体实现指定协议。

一定要把这两个角色分开。你可以使用 BPEL 来建模工作流而不使用任何特定的 Web Services 协议，你也可以用 WSDL 指定服务接口（有前面讨论过的局限性）而不涉及有关如何调用这些服务的任何内容。

就一般情况而言，笔者再一次建议直到基础设施的细节真正发挥作用之前，不要运用任何特定于 Web Services 基础设施的考量。换句话说，对于 SOA 景观的技术基础设施的变化，你应该永远都清楚这种变化意味着什么，并且努力将变化涉及的工作量最小化。

表现状态传输（REST）

最近，一个被称为“表现状态传输”（REST）的 Web Services 替代品引起了公众的注意，它指的是一个网络架构原则的集合，聚焦于对资源进行简单和无状态访问（这可以被认为是帮助互联网伸缩和成长的关键设计原则）。

RESTful HTTP（这通常被用作 REST 的同义语）使用 4 个基础 HTTP 方法：GET、PUT、POST 和 DELETE，对由 URL 标识出的资源进行无状态的读、写、创建/执行和删除。这种对 HTTP 协议的自然运用既简单又迅速，并且包含了对诸如缓存这样的典型需求的内生技术支持，所以，对访问由（万维网）服务器提供的数据或资源来说，这可能是一个很好的方法。

然而，它缺乏对安全性考量、非功能性特性、可组合性等的支持，再加上全面铺开的、复杂的 SOA 基础设施有流程服务、配制，某些场景下读取与写入服务还有不同的粒度，所以，它很难被用作 SOA 的核心协议。

如果只是想向外部系统提供对数据和资源的访问，REST 绝对是应该考虑的原则或技术，但是，如果你计划建立一个运行分布式业务流程的 SOA 景观，REST（还）不是一个可选项。

16.5 总结
Summary

- Web Services 是（正在演化为）SOA 基础设施事实上的标准。
- 基于 XML 和 HTTP 这两个核心协议，最重要的协议是用于描述 Web Services 接口的 WSDL，以及用于规定 Web Services 通信协议格式的 SOAP。
- WSDL 文件将不同的层用于服务接口、绑定和部署细节，因此，通常需要适当的过程来处理 WSDL 文件的生命周期。
- WSDL 并没有绑定在 SOAP 上，你可以使用其他绑定，所以接口描述（在原则上）能和任何技术一起使用。
- 作为一个标准，WSDL 只定义了互操作性的核心特性，没有提供安全性、SLA 等附加特性，这些特性可以由其他标准提供。
- 不同的标准化组织规定了许多不同的 Web Services 标准，各标准也有许多不同的版本，因此，互操作成为一个问题。Web Services 互操作性组织（WS-I）试图解决这个问题，其方法为针对某些标准的集合提供各种配套资料。
- 在实践中，为了可互操作，你通常应该使用 WS-I 基本配套资料和 SOAP document/literal wrapped 绑定。
- Web Services 是协议驱动的 ESB 标准，内生为点到点通信，因此，你有可能要引入拦截器，这也给了你在 ESB 内部使用不同协议或技术的机会。
- 为了更好地协同，你可以考虑实现一个 API 驱动的 ESB。

16.5 总结

- Web Services 是（正在演化为）SOA 基础设施事实上的标准。
- 基于 XML 和 HTTP 这两个核心协议，最重要的协议是用于描述 Web Services 接口的 WSDL，以及用于规定 Web Services 通信协议格式的 SOAP。
- WSDL 文件将不同的显用于服务接口、绑定和部署细节，因此，通常需要适当的过程来处理 WSDL 文件的生命周期。
- WSDL 并没有规定在 SOAP 上，你可以使用其他绑定，所以接口描述（在语则上）能和任何技术一起使用。
- 作为一个标准，WSDL 只定义了互操作性的核心特性，没有提供安全性、SLA 等附加特性，这些特性可以由其他标准提供。
- 不同的标准在组织规定了许多不同的 Web Services 标准，各标准也有许多不同的版本，因此，互操作成为一个问题。Web Services 互动性组织（WS-I）试图解决这个问题，其方法为针对某些标准的集合提供各种配置资料。
- 在实践中，为了可互操作，你通常应该使用 WS-I 基本配置资料和 SOAP document/literal wrapped 绑定。
- Web Services 是抽象层新的 ESB 标准，内生为点到点通信，因此，你有可能要引入些特殊，这也给予你在 ESB 内部使用不同协议的机会。
- 为了更轻量级的，你可以考虑实现一个 API 驱动的 ESB。

第 17 章

服务管理

Service Management

当一个 SOA 成长时，如何管理所有现存服务的问题迟早会被提出来。消费者如何知道服务是否存在，如何知道服务到底在哪儿？哪儿能找到关于服务的更多信息，让消费者能够理解服务和正确使用服务？处理这样的问题通常带来服务业务库和/或服务注册中心的概念。本章将介绍这些概念，讨论何时，以及如何管理服务。

17.1 服务中间人的历史

The History of Service Brokers

几乎所有对 Web Services 的介绍都会引入 3 个主要角色：服务供应者、服务消费者（请求者），以及服务中间人。它们之间的交互关系，见图 17-1。

通过该三角可以认识到，需要某种方法把供应者和消费者联结到一起，当供应者提供一个服务时，它就把服务注册到一个人所共知的地方（即使用服务中间人），当消费者需要一个服务时，消费者就可以使用中间人来找到该服务的供应者。

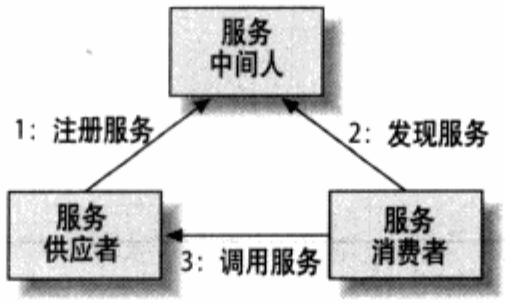


图 17-1：供应者、消费者和中间人

2000 年，当 Web Services 开始起步时，一个被称为“通用描述、发现和集成业务注册中心”（也被称为 UDDI 业务注册中心，或者 UBR）的中间人出现了，并且宣布要成为 Web Services 的世界黄页。UBR 打算扮演全球中间人的角色，让全世界供应者向自己注册它们提供的 Web Services，让消费者搜寻 UBR 就可以找到所需的任何服务。

正如 16.2.4 节提到的，这个概念最终失败了。那么，这件事证明了什么？它仅仅证明了当时那几家支持 UBR 的公司所采取的方法没有起作用，也许是时机不对、也许是概念有问题，或者，也许这些公司的功课做得还不足。

UBR 失败的一个重要原因可能是，建立一个位于中央的中间人并非实现 SOA 的第一步，正如笔者曾经在一个会议上听过的，“那些有朋友的人不需要黄页。”这就是说，当在黄页中寻找一个牙医之前，我们会向自己的朋友征求建议。同样地，当服务被提供出来，作为使用分布式业务流程的解决方案的一部分时，供应者和消费者往往已经彼此了解，此，在早期（还）不需要一个中间人。

当 SOA 成长时，这个话题慢慢变得有必要了，在一个地方列出所有现存服务的需求可能会出现，这样消费者才能通过询问一个中间人（或使用当前的术语，通过浏览一个服务“**业务库**”）轻松地找到所需服务。换句话说，只有当 SOA 景观扩展到一定规模时，才会出现对帮助消费者找到合适服务的中间人的需求。

然而，引入 UDDI 和 UBR 还有另一个原因：Web Services 调用内生地是点到点的调用（这是指调用者必须将消息发送到一个物理地址，16.3.2 节），所以，在这里引入某种形式的松耦合通常说来是有意义的，是把接收系统的地址硬编码到调用系统中，反之，可以在运行时从一个中间人那里请求得到供应者系统的地址。

这种方法使得负载平衡和失效备援成为可能，因为能够给调用者不同的地址。然而，要注意的是，这不是引入松耦合来处理 Web Services 调用点到点性质的唯一方法，一个方法是使用代理或拦截器，根据它们的路由政策（细节见 5.3.2 节），把每个调用委派给特定的供应者系统。

换句话说，一个中间人可能被用于不同的目的，比如在设计时找到一个服务，以及在运行时找到正确的供应者。这意味着我们可以对中间人能扮演的角色进行区分。接下来的一节对两个主要角色进行讨论。

17.2 业务库和注册中心
Repositories and Registries

这些日子以来，对中间人最常用的术语就是“业务库”和“注册中心”。这些术语经常可以当作通用术语互换使用，指的是从所有角度来看的服务中间人概念。然而，它们现在被用来指截然不同的角色，区别如下：

- 业务库从业务的角度来管理服务及其制品。这就是说，它们管理的是接口、契约、SLA、依赖关系等，其目的是帮助识别、设计和开发服务。从业务的观点来看，一个业务库应该包含有关服务的接口和行为的所有信息。这些信息应该独立于技术细节和基础设施考量，当公司切换到新的基础设施（ESB）时，不必改变业务库。
- 注册中心从技术的角度来管理服务。这是指它们管理服务，但不管理契约的所有细节，它们管理的是在运行时使用服务需要的所有技术细节（即部署信息），可以被用来将服务调用路由到提供相应服务的各个系统。如果必要的话，一个注册中心可以是基础设施（ESB）的一部分。

业务库和注册中心可以同时在 SOA 景观中存在，甚至可能用一个系统达成这两个目的。然而，此时会产生使业务库依赖于基础设施的危险，这意味着改变基础设施不会像业务库独立于基础设施时那样容易。

如果业务库和注册中心是不同的实体，那么，在注册中心中有一个对业务库的引用，通常说来是个好主意。如此一来，当人们想要了解运行时系统的语义细节时，他们就知道上哪去看。一般来说，注册中心是从业务库中生成的（用部署信息进行了增强），因此，从注册中心引用一个业务库应该不是什么问题。

你可能还想从业务库引用一个注册中心，这样做的方向与正常数据（或生成器）的流向相反，因此可能会产生一些多余的依赖。一般说来，类似的引用应该是间接的（比如 URL）。

17.2.1 设计业务库和注册中心
Designing Repositories and Registries

如前所述，业务库和注册中心可能被用于不同的目的。因此，你应该清楚一个业务库和/或注册中心在你具体的 SOA 景观中所扮演的角色和承担的职责，不同的角色和职责导致处理这些实体时有不同类型的解决方案，遵循的过程也不尽相同。

以下是一些要考虑的重要主题和问题：

内部协调或市场

业务库/注册中心是提供给内部使用的，目的是优化流程和强制重用？或者，它是提供出来支持市场的想法的？后者导致的需求是由多个供应者提供同样的服务（在内部使用时情况通常不是这样，你会努力避免冗余）。

信息量

业务库/注册中心是管理服务的所有信息(即契约、所有语义信息和非功能性特性)，还是只管理与技术相关的信息？

在开发过程中的角色

提供业务库/注册中心是用来帮助解决方案经理和设计师在设计新业务流程和解决方案时找到现有的服务，还是用来找到所需信息以提供运行时的互操作性？也就是说，它包含特定基础设施的技术数据和/或部署信息吗？

用户接口

业务库/注册中心对设计或建模新服务提供（图形化）支持吗？或者说它“仅仅”提供了接口用以插入、更新和获取服务，而服务由别的工具设计？

集中程度

只有一个中央的业务库/注册中心吗？或者说整个业务库/注册中心是若干分散的信息块的集合？如果是前者，你如何防止它在开发过程中变成瓶颈？如果是后者，你如何处理和避免冗余问题？

离线能力

在开发环境内部，不在线是否也可以使用业务库/注册中心进行工作？

版本划分和配置管理支持

业务库如何处理一个服务有不同的版本，以及处于开发中的服务有不同配置的问题？（比如，因为最近有些事情发生了变化，一个处于测试中的服务接口可能与处于开发中的服务接口不同。细节见第 12 章）

在实践中，对这些问题的回答可能有非常大的差异，因此业务库也可能看起来差异巨大。图 17-2 展示了处理不同业务库和注册中心的一个可能的例子。在这里，每个部门或业务单位都有自己设计和建模服务的方法。一个普遍的要求是，这些模型被转换为中央注册中心的输入格式，注册中心为消费者和供应者提供接口，把服务信息部署到 ESB 中。因此，在注册中心里面，部署细节要发挥作用，业务库也被要求提供对服务进行描述的

HTML 文件，在用来向中央注册中心传输数据的中间格式里，相应的 URL 也是其一部分，这样注册中心就可以显示出 URL，详细的服务规范可以通过这个 URL 得到。

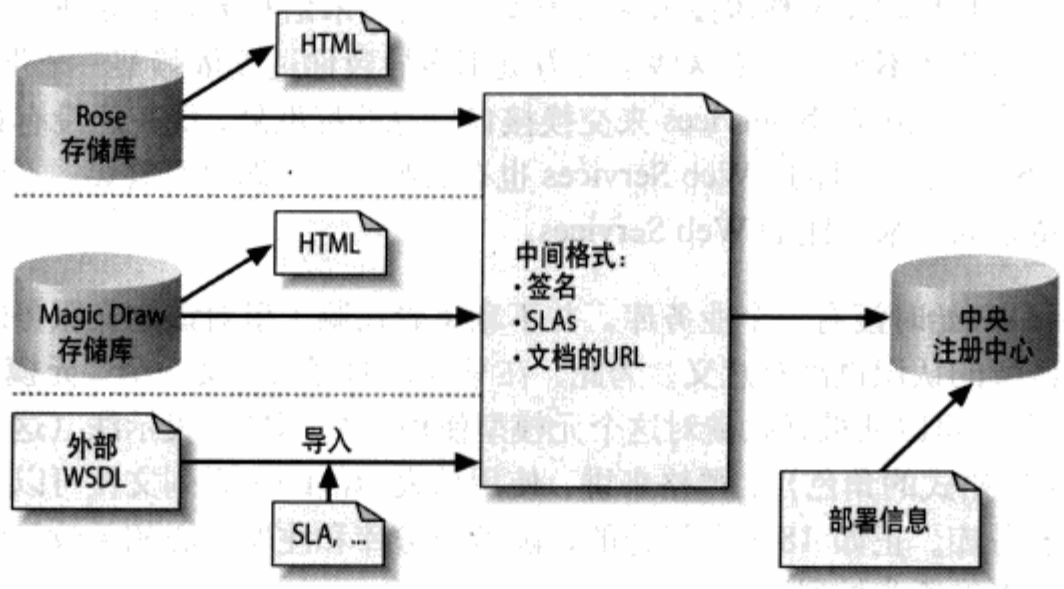


图 17-2：一种业务库和中央注册中心可能的结合

对于由外部系统(代表了 B2B 连接)提供的服务来说，有一个将服务描述(通常是 WSDL 格式)转换到中间格式的重要机制，它在服务文档中添加了缺失的信息，如 SLA 和 URL。

17.2.2 建立业务库和注册中心

Establishing Repositories and Registries

业务库和注册中心通常随时间流逝而演化。正如本章前面提到过的，除非你的技术基础设施要求你有一个注册中心以提供松耦合，否则你并不需要从一开始就有它们。因此，开始时可以没有业务库，或者仅仅用一个简单的 Excel 电子表格来描述你最初的服务。

当你的 SOA 成长时，你可能会考虑更通用的设计流程和建模服务的方法。然而，发生的情况也可能是：不同的团队、部门或业务单位会偏向于使用不同的设计和建模服务的方法。当所有人使用同样的工具时，协同效应会更好些，但需要付出的组织结构方面的努力有可能太高，并且，使用半成熟的解决方案做事很容易让业务团队受挫。因此，很可能会出现像图 17-2 所示的情形。

这个方法没什么不好，晚些时候才集成服务也许是个问题，但是，别忘了就服务设计而言，总是存在某种类型的异质性，因为你可能和别的公司合并，从别的公司引入新的 B2B 连接，而这些都可能使用了不同的工具和流程。

只要你的服务描述以格式良好的风格（比如用 XML 文件）存储，你就总是应该能够把这个信息转换为所要求或引入的任何其他格式。

要注意，工具供应商可以提供业务库和注册中心的解决方案，这些解决方案可能集成了业务库和注册中心，或者将业务库与 ESB 的技术组件集成。实际上，在工具供应商提供的包括业务库的基础设施整体技术解决方案中，所用术语和笔者在此使用的可能不太一样。术语不同关系不大，只要这些解决方案不会导致你过度依赖某一个供应商就行。除了即将出现的、使用 Web Services 来交换接口信息的标准外，现在还没有真正的标准（并且正如 16.3 节讨论过的，Web Services 也有其缺陷）。你也要警惕，不要让自己过度依赖某个特定的技术，比如 Web Services。

在 SOA 项目最开始时没有一个业务库，并不意味着在脑子里对该定义服务的那些方面和特性建立清醒的认识也没有意义。为此，在早期，你可以定义一个**“元模型”**（见第 18 章），你甚至可以从最开始就对这个元模型使用一个 XML 表示法（这可以扮演图 17－2 里的中间格式的角色）。严格来讲，使用该表示法的一系列文件可以被认为是你最初的服务业务库。正如 18.3 节讨论的，随着业务库和注册中心相关政策的发展和成熟，这种方法能够逐步扩展。

17.3 总结
Summary

- 不管从组织的角度还是从技术的角度来说，业务库和注册中心都有助于管理服务。
- 业务库从业务角度管理服务，处理整个契约，但是不包含特定基础设施的信息。注册中心从技术角度管理服务，包含了在特定基础设施中，部署和运行服务所需要的所有信息。
- 在定义业务库和注册中心的角色和职责时，需要考虑许多方面的内容。
- 如果你改变自己的基础设施，应该做到能保留自己的业务库。
- 当你开始建立 SOA 时，一般来说，并不需要业务库和注册中心，当业务库和注册中心变得有用或有必要时，再将其引入是一种不错的做法。

第 18 章

模型驱动的服务开发

Model-Driven Service Development

实现在异质环境中的服务接口需要大量代码，这些代码中的大多数经常结构相同，只能根据不同的参数、异常和其他配置数据来区分，这是应用代码生成器，或者用一个更加被热炒的术语——**“模型驱动的服务开发”**（*MDSD*）——的理想情况（MDSD 中的 S 更多时候代表“软件”，但是我们在这里讨论的是服务，所以，笔者将之替换为“服务”）。模型驱动的服务开发覆盖了针对供应者和针对消费者这两个方面。

18.1 生成的服务代码

Generated Service Code

SOA 是用于异质环境的一个概念，所以，你可能需要很多不同的代码来实现和调用一个服务。然而，除了实现细节之外，对于给定的平台，代码的结构通常非常相似。如此一来，两个不同服务的代码，其区别只是在传输的特定数据上，包括诸如消息交换模式（见第 10 章）这些方面。

从概念上，处理这些数据的代码（将中间件协议映射到 API 的代码，处理安全性和可靠性的代码等）将非常相似，你可以使用程序库和框架来处理这些共同的性质。但是，这个方法有局限性。在某些地方，一个特定服务的特定行为一定会凸现，除非你提供了完全通用的解决方案，否则，你会需要操作在不同属性和不同数据类型之上的代码。基于给定的服务描述生成这样的代码是有意义的。

为服务生成的代码可能包括（但不仅限于）：

- 为服务的实现细节提供框架的代码。它的目的是为服务供应者提供一个 API，如此一来，服务实现的所有公共方面都得以解决，服务实现者能集中在特定的业务功能上。这样的代码被称为“**框架**”（*skeleton*），因为被生成的代码是能够用特定的实现代码进行手工扩展的框架。
- 提供对服务的调用功能的代码。它的目的是为服务消费者提供一个 API，如此一来，从业务的角度来看，调用服务就容易了。它应该覆盖调用一个服务的所有技术细节（这是 ESB 所要求的），这样的代码经常被叫做“**存根**”（*stub*）或“**代理**”（*proxy*）。
- 使你能在基础设施内部处理服务的代码，比如进行业务活动监测的代码（见 5.4.7 节）。
- 提供测试框架的代码。例如，代码可能提供了非常简单和直观的方法——仅仅使用描述性配置文件——来指定不同的测试用例。
- 文档。严格意义上这不是代码，而是生成器工作在服务描述上的结果。例如，你可能生成 HTML 页面，或者是普通的，描述一个或所有系统服务的 Word 文件。
- 在构造和部署期间，对生成的制品进行处理的脚本。

图 18-1 展示了一种可能的 MDSD 方法。有的团队用 Rational Rose 存储库设计其服务，而另一些团队使用 Web Services 工具创建最初的 WSDL 文件。Rose 的存储库包含了所有契约，有一个生成器从中创建 Word 文档。Rose 和 WSDL 都可以被转换到一个中间格式上，这个格式可以用作特定平台（Java、C、C++）代码生成器的基础。除此之外，还可以生成一些针对 ESB 的注册中心代码，以及通过 Java 代码，生成针对特定基础设施的 WSDL 文件（注 1）。

注 1：那些疑惑为什么 WSDL 又是输入又是输出的人，请记住 WSDL 有其生命周期。所以，即使是针对相同的服务，输入和输出的 WSDL 文件也可能不同。细节见 16.3.3 节。

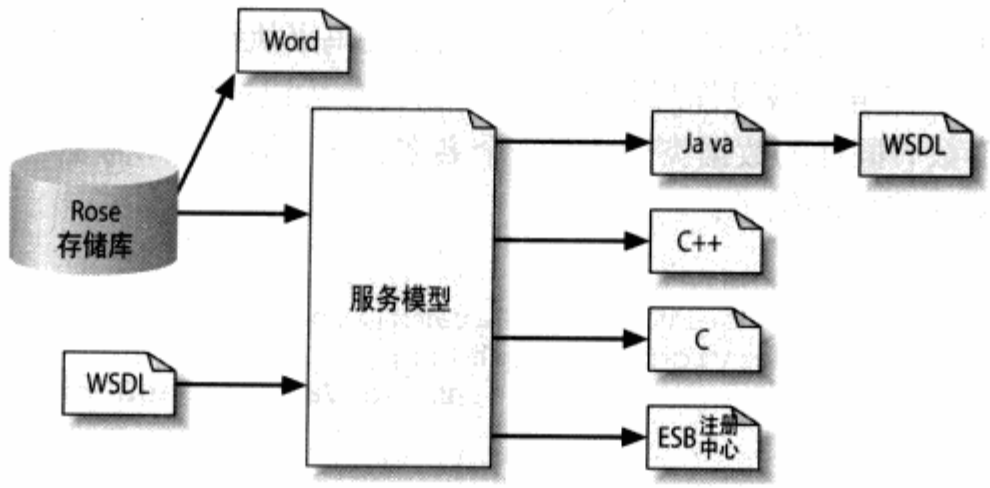


图 18-1：MDSD 方法的范例

笔者并不宣称这是个正确的过程，这只是个例子，展示出了具备历史成长轨迹的代码生成过程看起来会是怎样的。在实践中，笔者看到过比这复杂得多的例子。例如，特定平台的代码生成有不同的模式，用于控制生成哪种类型的代码（框架、存根、数据类型、测试代码，等等）；如果中间模型指定了用哪个平台来实现服务，则将只生成针对那个平台的框架代码。

18.2 建模服务
Modeling Services

生成代码这种想法由来已久，遵循“连来三次就要自动化”的法则（见[ThreeStrikes]），代码生成是避免一遍又一遍做同样事情的常用机制。然而，为了更具体一点，让我们更深入地讨论代码生成的一般性概念，以及用于服务的模型驱动的软件开发。（对这个话题的更深入讨论可以见[StahlVölter06]）

18.2.1 术语
Terminology

如果比较两个硬编码的服务实现（或是对服务的调用），你会发现 3 种类型的代码：

- **“普适代码”**，这是对每个实现都一样的代码。例如，对服务而言，这可能是对服务名字的记录日志，以及处理其他常见技术数据的代码。如果有可能以普适的方式处理参数（例如，使用 Java 的反射），这样的代码也归于本类别。
- **“纲要代码”**，这是结构相同，然而根据具体情况多少有些区别的代码。例如，为了将服务的所有参数映射到一个有特定数据类型的 API 上，纲要代码将使用特定名字和类型逐个地处理参数，而普适代码则以共通的方式对所有参数进行循环处理。
- **“没有共同性质的代码”**。对服务而言，这是指一个服务特定功能的具体实现。

针对同样的任务，可以频繁地调用普适代码或纲要代码来执行。下面的例子显示了这两种实现方式之间的差异。假设你想将服务名字和其参数记录到日志中，使用普适方法时，所有输入参数有同样的数据类型（有一个名字、一个值），你需要一个循环来记录数据。这里是 C++代码的例子：

```
logName(service.getName());
for (int i = 0; i < service.getParams().getSize(); ++i) {
    Param param = service.getParam(i);
    logParam (param.getName(), param.getValue().asString());
}
```

使用纲要代码方法时，你会逐个处理参数，参数可能是直接访问，也可能是作为数据结构的成员访问：

```
logName("getContractData");
logParam("contractId", contractId.asString());
logParam("firstName", firstName);
logParam("lastName", lastName);
logParam("isActive", isActive ? "true" : "false");
…
```

两种形式都有利有弊。例如，普适代码在属性数量变化时不需要修改，而纲要代码的运行速度可能更快些，因为它在编译时处理类型差异，阅读和调试也方便些。哪个解决方案更好，依赖于当时确切的环境，没有什么一成不变的规则。

普适代码通常是框架的一部分，因为它不必随着每个服务实例而变化。对比起来，纲要代码是特定于服务实例的，必须从服务的正式描述（定义了服务名字、参数名字和参数类型）中生成。源的格式可以变化，例如，它可以是一个 UML 类（见图 18-2）或一个 XML 文件（见图 18-3）。

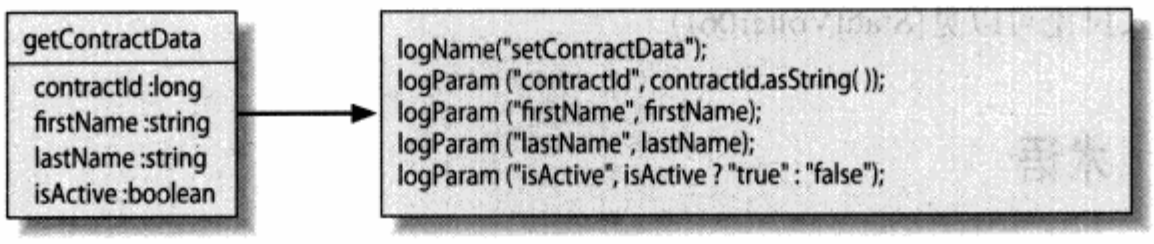


图 18-2：从 UML 源生成代码

不管格式如何，这样的一个源被称为“**模型**”（因为它抽象了实现细节）。

除了特定的表示法外，一个模型通常有特定的结构。例如，前面的两幅图包含了一个模型的两种不同的表示法，使用这些表示法，你可以指明服务的名字，以及不同参数（输入和输出）的名字和类型。模型的结构被称为“**元模型**”，换句话说，一个元模型定义

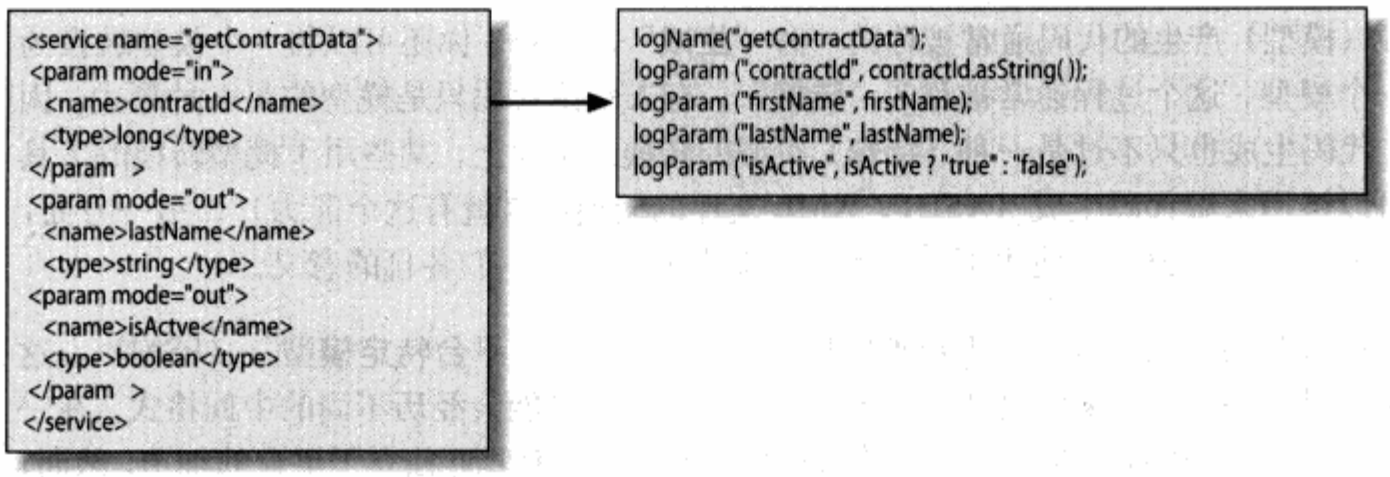


图 18-3：从 XML 源生成代码

了模型的正式结构和元素。它还可以包括正式的限制，比如值域，但是它不包含语义，你可以仅仅基于元模型来判断一个模型是否格式良好，可要从元模型生成代码（或做任何其他事情），你必须了解语义。

针对前面两幅图的一个元模型看起来可能像图 18-4。使用 UML 表示法时，这个元模型的定义是：一个服务有一个名字和许多参数，每个参数有一个模式（“入”或“出”）、一个名字，以及一个它所用的类型。注意，只有图 18-3 的 XML 表示法定义了模式，图 18-2 的 UML 表示法没有指明参数的模式，在这种情况下，要了解一个参数是输入还是输出，就要求有附加的信息。

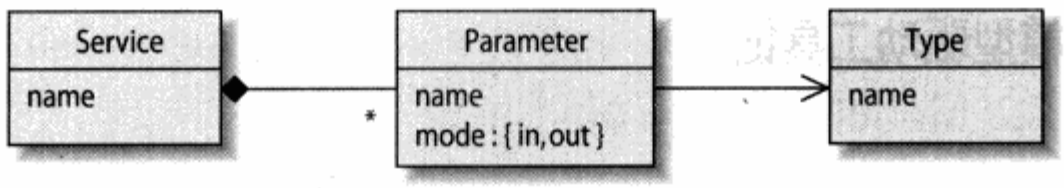


图 18-4：一个纯示意、不实用的元模型

正如你所见，表达一个模型的方法非常不同，用来规定一个模型的具体表示法被称为“**建模语言**”或“**领域特定语言**”（*DSL*）。后一个术语表明，针对模型所采用的表示法是领域特定的，所以，对模型的描述能集中在关键信息上。一个 DSL 应该精确、简练、可读和完整。正如前面的图所展示的，DSL 可以使用图形表示法（如图 18-4）或文字表示法（如图 18-3）。

注意

你可以把 Web Services 描述语言（WSDL）当作 Web Services 的一种 DSL。然而，正如第 16 章讨论的，WSDL 并不完整，只覆盖了服务的某些方面，因此，你不应该用它来指定服务的所有特性。

从源（模型）产生的代码通常被称为一种“**生成**”。另外，你还可以将一个模型转换为另一个模型，这个过程通常被称为“**转换**”。有人争论代码只是模型的另一种形式，因此，代码生成也只不过是一种（特殊）类型的转换。实际上，某些用于模型转换的工具就可以被用来做代码生成（XSLT，XML 文件的转换语言就有这个能力），另一方面，对模型转换和代码生成使用不同的术语可能有助于澄清它们各自的意义。

有的模型驱动方法区分了“**平台独立模型**”（PIM）与“**平台特定模型**”（PSM）。这样做的目的是，在最终代码被生成出来之前，生成有可能会经历不同的中间格式。在一定程度上，比起由模型转换产生出的目标模型来说，源模型更独立于平台的细节。然而，在实践中，很难对 PIM 和 PSM 作出泾渭分明、非此即彼的区分，因为模型往往都特定于某些平台考量，而独立于别的平台考量。

同时注意，转换不一定是双向的。通常你无法从转换后的目标模型产生出源模型，因为目标模型虽然往往在整体上包含更多细节，但是语义细节更少。不同的源模型可能产生同样的目标结果，仅仅根据目标模型这个结果，无法指出在源里用了怎样的概念。因此，MDSD 通常不能执行往返式的处理，这意味着你不能通过操纵已经生成的、针对特定平台的源代码，达到改变服务参数的目的。你必须修改源模型，再次生成代码。正由于此，生成过程及其相关工具的易用和快捷是关键所在。其于同样的原因，对于（纲要）代码，你应该只生成绝对需要的代码，对其他所有东西都提供为程序库和框架。

18.2.2 模型驱动工具链

The Model-Driven Tool Chain

模型使用的表示法（DSL）将影响你对工具的选择，因为特定的工具会使你能直接在特定表示法这种抽象层上进行操作。

例如，对 WSDL（Web Services 的建模语言，在第 16 章讨论过）来说，特殊的工具使你能用特殊的（甚至是图形的）IDE 来创建 WSDL 文件。虽然 WSDL 本身是文本的 DSL（使用特定的 XML 大纲），然而，这些工具使你能通过一个可视化的界面而非难读和难维护的 DSL 表示法，在相应的服务设计抽象层上进行操作。如果你愿意，你也可以使用自己喜欢的文本编辑器或 XML 编辑器。然而，使用特殊编辑器和 IDE 的优点是，

工具将帮你保持结果模型的格式良好（即根据元模型来看是有效的）。当你使用自己喜欢的文本编辑器时，则由你自己来保证有效性。

一般说来，对服务而言，差别巨大的工具和视图都可能是合适的。例如，虽然一个图形化模型有助于显示结构、依赖性和数据流，但对于定义签名来说，它往往不合适。因此，一个模型不应该依赖于一个特定的工具，理想的情况是有选择的自由，开发者可以使用他们喜欢的文本编辑器、电子表格工具、UML 工具、XML 编辑器，以及 IDE，这有助于增强可接受性，提高开发性能。

要注意，工具操作于模型之上，而此处的模型将产生你主要的源代码，这些是定义你系统行为的代码。要应用有关源代码配置管理的通常原则如下：

- 工具的启动时间应该非常短，这样，用户就可以用工具快速执行小的修改。如果启动工具需要较长时间，那么工具至少需要能够运行一整天。
- 工具应该具备一个恰当的人机界面（HMI），这样用户得到一个想要的行为就只需要最少量的操作（点击、文本输入等）。
- 模型必须能被版本化，这样你才能维护和区分开发中、生产中的不同版本（即同一代码/模块的不同版本，不要将其与从业务的角度来看的服务的不同版本相混淆）。有关版本划分的更多信息请见第 12 章。
- 模型应该支持团队开发，这意味着你要能合并对同一模型的多个修改，显示不同模型和版本间的差异，以及直接导航到模型的特定部分。

理想中，模型应该有一个让你用普通的文本编辑器也能进行编辑的表示法，这是指模型应该包含可读的字符，有一个在不同文本行上分割不同信息项的结构，包含不同层次的缩进等。遵循这个指导方针的好处是，所有的配置管理工具都可以管理这种模型，甚至可以执行合并和比较。

建模的表示法越二进制化、越专有，开发过程就有越多的限制。例如，如果过程中的某部分有这样一个步骤——所有的服务描述都存储在中央业务库（见第 17 章）中，并且，其内容存储在数据库中——则合并和比较可能会变得非常复杂（如果这些操作不被你的工具支持的话）。源代码格式带来局限性的另一个例子见选读栏“用 Rational Rose 进行服务建模”。

用 Rational Rose 进行服务建模

一个相当古老的建模工具是 IBM 的 Rational Rose。这个工具使用图形化界面，使你能工作在所有不同种类的 UML 图表上，其中一些图表可以用于定义服务。

然而，Rational Rose 有一些局限性。首先，它启动要花太长时间，其次则是人机界面很有限，这就很浪费开发时间，导致严重的可接受性问题（在笔者工作过的一个项目上，加载所有服务模型要花 10 到 30 分钟）。除此之外，对合并多个修改，或者显示模型之间的差异，它都没有真正的支持。结果就是，两个不同的人不可能在同一时间工作在同样的模型上。如果把模型分割为不同的 CAT 文件，这个局限性可以被绕过，然而，主要的问题仍然存在。例如，在之前提过的项目中，使用不同的工作区对即将发布的新服务建模，当开发开始时，新服务模型被手动加入到主模型中。这就意味着在所有不同的发布中，模型中的缺陷都必须修复完毕。

另一个问题是，图形图表并不适用于以紧凑的形式列出不同服务和服务版本的接口，这使人很难察觉不同服务版本间的不一致和差异之处，因此，模型存在比通常预计的多得多的设计错误。

为了应对某些局限性，做这个项目的公司写了特殊的脚本（使用内置的 Visual Basic 脚本语言），设计了特殊的 GUI 元素。但是，当然地，这要求付出大量的时间和精力。

不过，最后的结果是，该公司最终得到了一个列出了几百个服务的、能工作的业务库，服务排列方式相当一致、完整且平台独立。这就意味着，靠修改或增加相应的代码生成器，这家公司能够轻易地满足特定平台和基础设施的所有新需求。

18.3 实践中的元模型
Meta Models in Practice

在实践中，很早开始就定义一个最初的元模型是个好主意。这个元模型应该定义出在你具体的 SOA 中，特定服务的不同能力和特性。

图 18-4 显示了一个非常不实用的元模型，更实用的元模型可能还要做以下这些：

- 提供对服务和服务操作进行分组的能力。实际上，将一个服务定义为由不同服务操作组成（见 18.4.8 节），并且根据不同的逻辑或物理类别对不同服务进行分组，都是典型的行为。
- 定义一个类型系统，它定义了基本数据类型，以及组合基本数据类型的方式（用于结构和链表）。
- 提供管理服务所需的所有信息（例如，指明供应者和消费者、契约、SLA，等等）。
- 区分某类信息的抽象态和实例态。

让我们看几个（仍然是简化的）例子，看看 SOA 元模型在实践中可能是怎样的。图 18-5 主要处理类型系统，它指出，一个服务接口有一个名字、一个或多个操作。一个操作可能包含几个参数、一个返回类型，以及多个异常（这是个特殊种类的类型）。每个类型至少有一个名字，但是，存在一些更复杂的类型：链表（数组）有同样元素类型的多个值，而结构和异常有属性，属性也依次有名字和类型。（关于服务数据类型的更多内容，见 15.5 节）

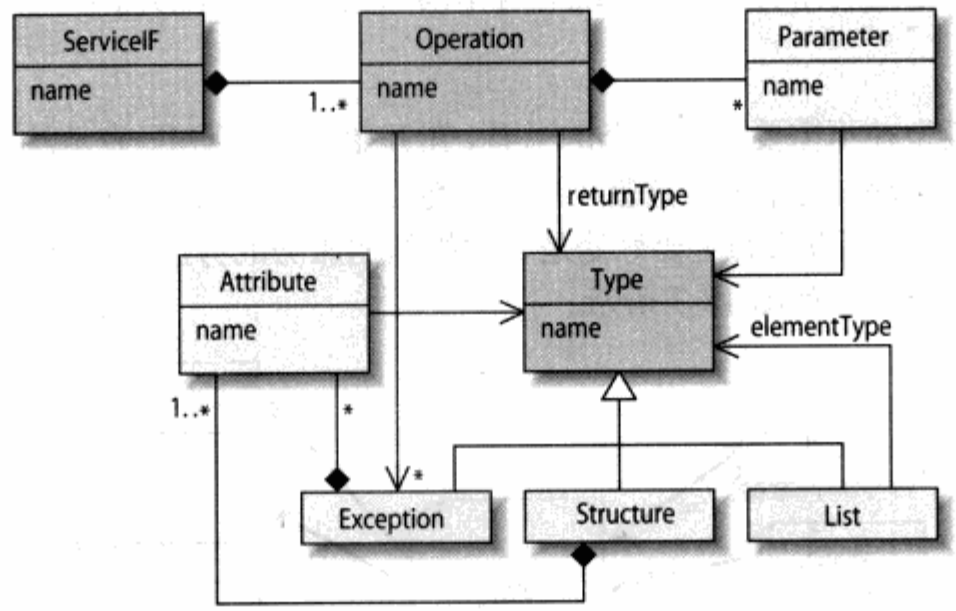


图 18-5：元模型的一个例子，第一部分

和图 18-4 相对比，本模型定义的 parameter 永远是输入参数，只有一个输出参数，它被指定为 returnType。

举个例子，一个具体的模型可能定义了一个名为 CustomerService 的服务，提供了一个被称为 getCustomerData 的操作，操作有一个输入参数 contractId，其类型为 long，返回值（输出参数）是客户数据的某种结构体，包括若干属性，比如 firstName 和 lastName，这两个属性的类型为 string。

图 18-6 显示了一个展现服务和系统间联系的元模型。首先，它在服务接口和服务实现之间进行了区分，因为对一个服务接口可能有多个实现。每个服务实现都由一个特定系统提供，该系统有一个名字和一些联络数据。除此之外，你还能定义 SLA，它被规定为特定服务实现的特定供应者和特定消费者之间的契约的一部分。

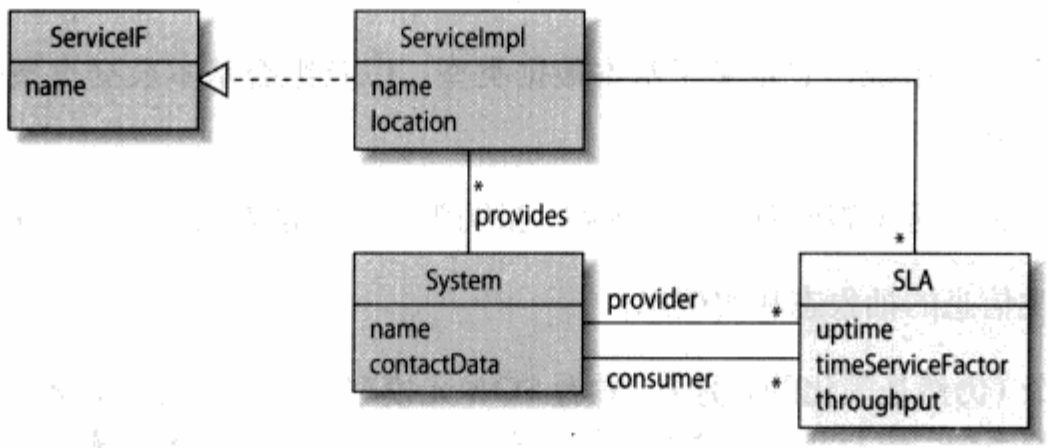


图 18-6：元模型的一个例子，第二部分

在实践中，元模型看起来可能和笔者展示的例子非常不同。例如，你可能使用其他术语，如服务、服务实例，或者端口。你也可能将系统划分为不同的组件，并且如果你有类似于“服务能扮演不同的角色”、“系统能有不同的实例”这样的需求，则事情会变得非常地复杂。

设计最初的元模型并不需要非常复杂的工具，在开始时，将 XML 表示法用于整个模型，从中生成相应的代码，通常说来这就足够了，然后，可以将其他阐明模型的方法转换到 XML 表示法上。这个方法的主要好处是，它把阐明服务的方法从转换模型和代码生成的方法中解耦了出来，如图 18-7 所示。

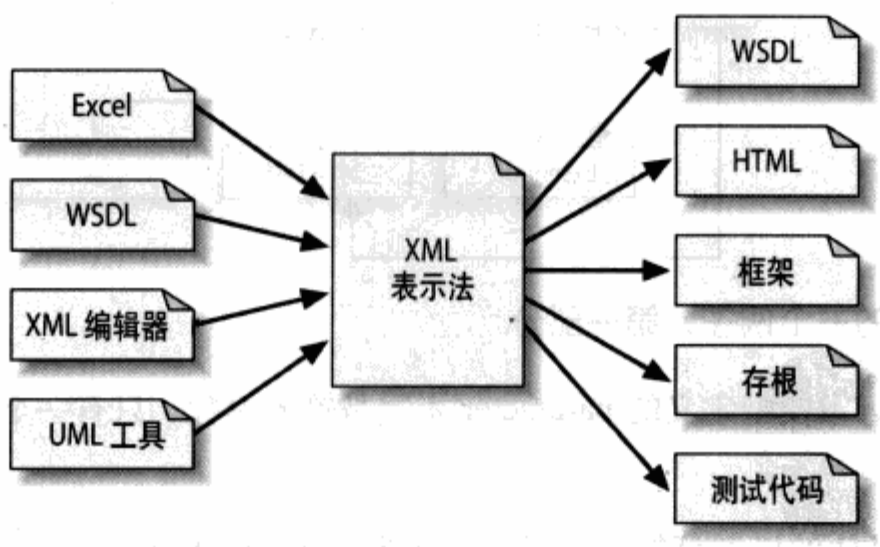


图 18-7：元模型表示法将规范从生成方式中解耦了出来

如果你后来引入了不同种类的服务规范，则只需提供一个方法把这些规范转换到你的模型表示法上。如果你对基础设施进行了修改，则只需定义新的，或者修改现有的代码生成器。如果你引入了通用的业务库和注册中心（见第 17 章），则只需实现一个将其读出和写入模型表示法的方法，就能将其集成到现有的系统开发景观中（见图 18-8）。

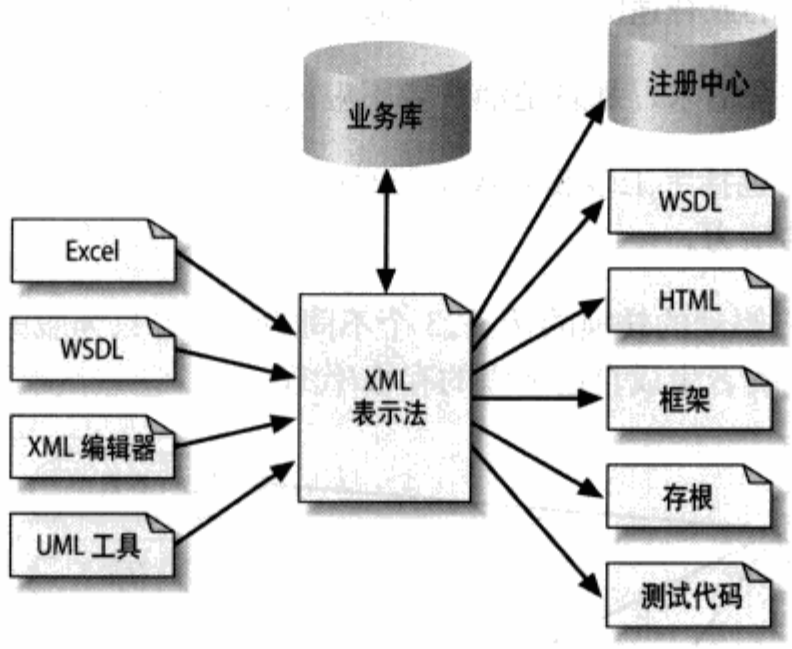


图 18-8：将业务库和注册中心集成到现有的系统开发景观

18.4 建立 MDSD 过程

Setting Up MDSD Processes

一旦你定义了哪些代码由哪个模型生成，你可能认为自己的 MDSD 工作已经完成了。然而，还有一个重要的问题没回答：谁，以及什么时候执行转换和代码生成？从概念上，可能有 3 种不同的方法（为了简化，在接下来的一览表和后续讨论中，我使用“转换”作为“模型转换或代码生成”的缩写）：

消费者驱动的转换

需要转换后结果的人执行转换。

供应者驱动的转换

定义转换源（源模型）的人执行转换。

第三方驱动的转换

一个单独的团队执行转换。

让我们讨论这些方法的更多细节。

18.4.1 消费者驱动的转换

Consumer-Driven Transformations

消费者驱动的转换（需要转换后结果的人执行转换）通常最容易被建立为公司范围的政策。消费者只需要能访问源（模型）和生成器。一个额外的好处是，消费者有足够的理

解力和基础设施来验证生成的结果，因为它们执行转换的原因是需要结果用于自身更进一步的处理。然而，这个方法有一些局限性：

- 消费者必须知道在何时需要新的转换。这就是说，元模型的供应者或生成器在一个新转换是必须的或合适的时候，必须通知消费者。
- 可能会有一个安全需求，那就是消费者不应该看到元模型。
- 如果转换复杂（包括手工步骤）或花费时间较多，每个消费者可能都必须投入同样多的、大量的精力。

图 18-9 展示了消费者驱动的转换的例子。3 个不同的供应者只为服务实现生成它们各自特定的框架，每个消费者生成自己所需的存根/代理。

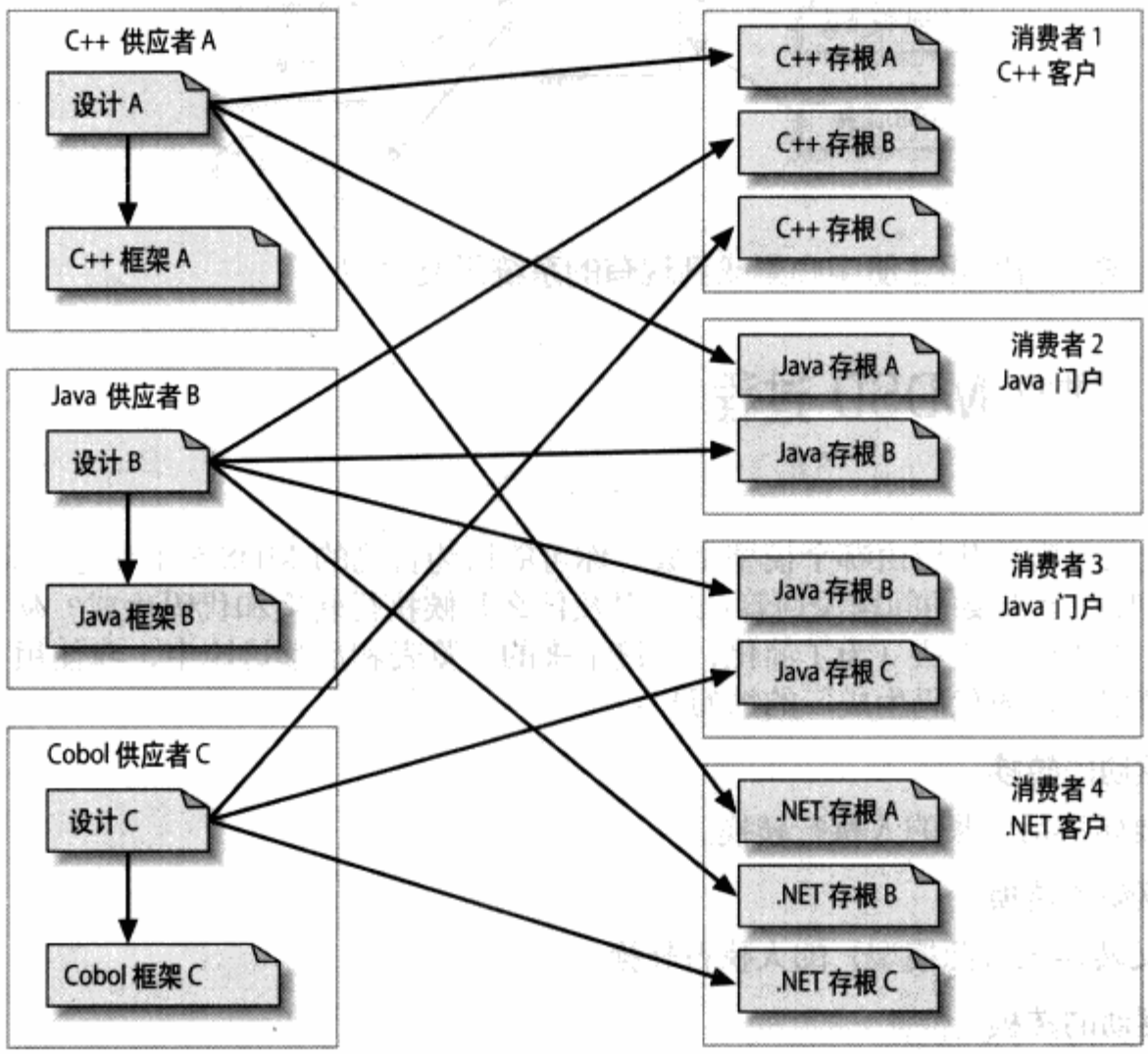


图 18-9：消费者驱动的转换

注意，在实践中要生成更多的东西，并且，有的系统可能既是供应者又是消费者，其结果就是，你甚至可能有循环依赖（例如，如果系统 A 调用了系统 B 的一个服务，并且系统 B 调用了系统 A 的一个服务，那么后一个调用可能是，也可能不是一个逻辑应答）。

18.4.2 供应者驱动的转换
Provider-Driven Transformations

供应者驱动的转换（定义转换源的人执行转换）解决了每个消费者有多个转换的问题。然而，它也有一些缺点：

- 难以（如果并非不可能的话）验证生成的代码，因为供应者也为其他平台生成代码，然而，供应者可能只熟悉自己的平台。
- 可能出现非必需的转换。
- 如果修改了生成器，供应者必须知道需要一个新的转换。
- 如果服务接口变化，消费者必须执行相应的修改，以避免破坏下一次代码构造。

图 18-10 展示了如果转换由供应者驱动，图 18-9 的例子看起来会是怎样。所有 3 个供应者既为它们的服务实现生成特定的框架，又为可能的消费者平台生成存根/代理。

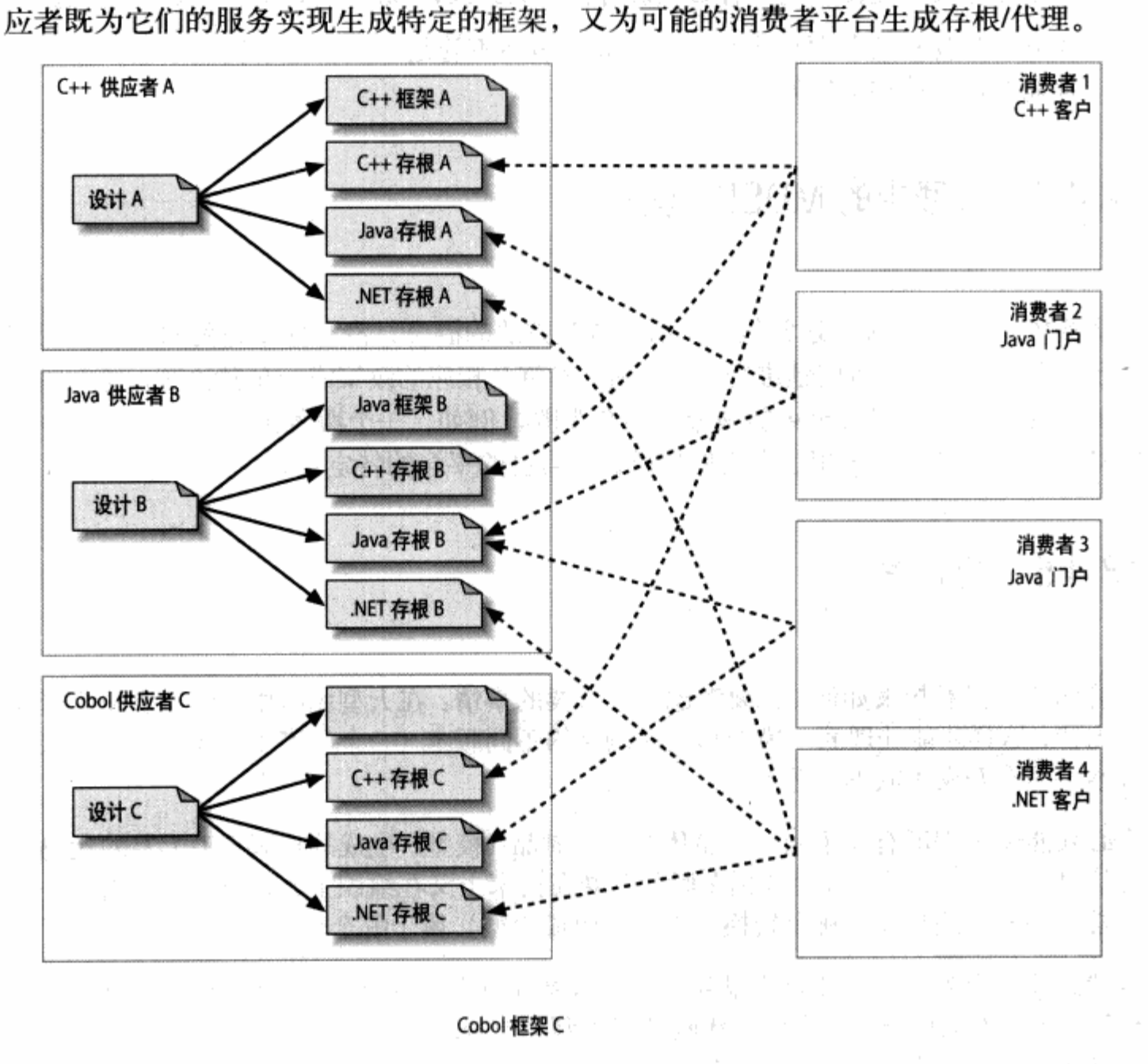


图 18-10：供应者驱动的转换

注意，在实践中，可能会生成更多的东西，而且，有的系统可能既是供应者又是消费者（有可能产生循环依赖）。

18.4.3 第三方驱动的转换
Third-Party-Driven Transformations

当有一个构造团队专门集成来自不同部门的源代码，并将此作为迭代软件开发过程的一部分时，第三方驱动的转换（一个单独的团队执行转换）通常是合适的。“每晚构造”可以包括必需的代码生成。这个方法的优点是，不管是源模型还是生成器的修改，都会产生相应的新代码。除此之外，构造团队应该能够验证结果，因为它通常也构造使用生成代码的消费客户端。然而，这个方法也有一些缺点：

- 消费系统必须集成到统一的构造过程中。
- 如果服务接口变化，消费者必须执行相应的修改，以避免破坏下一次代码构造。
- 建立“每晚构造”要付出精力和时间（虽然不管是否使用这个转换策略，你可能都需要执行该构造任务）。

18.4.4 实践中的 MDSD 过程
MDSD Processes in Practice

在实践中，MDSD 过程通常包含了所有这些方法的混合，没有什么完美的转换政策。相反，你必须针对具体问题进行具体分析，对每个相应的政策进行单独决策，建立出所需的过程。转换过程甚至可能被分成两个步骤，例如，一个服务供应者生成一个中间 WSDL 文件，消费者再用这个文件生成针对自己的平台的特定代码。

18.4.5 打标签
Labeling

不管你的过程看起来如何，都要考虑一个重要的事情：在大型团队中，你需要配置管理的支持，这样才能处理同一模型和生成制品的不同版本（一个生产中版本、一个测试中版本、一个开发中版本，等等）。

你必须能够识别所有属于同一个整体的不同制品（模型、生成的代码、手工写的代码部署信息，等等）。转换和生成需要时间（既在技术上又在组织上），所以，你需要一个非常清楚的方法来识别哪个转换结果（代码或模型）属于哪个转换源（模型）。

这就是说，作为模型一部分的服务描述，需要有与为其生成的代码相同的（或相应的）标签。这两者的标签，也应该与代码被部署到测试或生产环境中时，加入的部署信息的标签相同。理想情况下，你应该能够用一个标签，或者一套对应的标签来识别所有互相关联的制品。

要注意，在实践中，打标签也应该包括转换的结果。你可能认为这不必要，因为你可以再次从打好标签的源模型中生成结果。然而，生成要花时间，并且可能发生在与源模型修改不同的时候，所以，你需要一个方法来快速取得生成的制品，同时知道哪个生成代码属于哪个源模型。

作为可选方法，你可以不对结果打标签，而对生成器打标签，从而提供对所有制品的可追溯性。如果生成器是一段简短的脚本，或者是一堆配置文件，这可能是个好主意。然而，对复杂的生成器打标签和维护不同的版本有其局限性。严格来讲，你可能需要对生成器使用的操作系统打标签，因为它也可能影响生成器的行为。对结果打标签要容易得多（并且，因为生成结果是进一步处理所需配置的一部分，所以经常也要求这样做）。

18.4.6 调试
Debugging

当你生成代码时，通常会改变抽象层，不像在模型内部对每个参数用一行或一个实体来体现，你生成的代码在逻辑上对每个参数用多行来处理。现在，当调试系统时，你不会一步步地单步执行过去，因此，生成格式良好、可读性好的代码就是关键所在，这包括生成解释干了什么的注释。没有这些东西，则将只有生成器的实现者能够调试生成的代码。

具体的代码生成政策经常是有效情况的反面：生成器创建非常简短的代码，不用空行，不写注释，因为可读性不被认为是个问题（此处作出的假设是，一旦代码被生成，没人应该需要去读它或者维护它）。但是，即使是实现生成器的人也会犯错误，能够追踪所有软件的数据流很重要（除非代码生成包括了某种形式的加密）。

不提供结构良好的生成代码的唯一借口是，调试器运作在和建模语言相同的抽象层次上。BPEL 引擎（见 7.4.2 节）是这种方法的好例子。大多数 BPEL IDE 都允许你图形化地创建高层服务和业务流程，同时也提供有图形界面的 BPEL 引擎，这样，你就可以在和 IDE 相同的抽象层次上看到运行时流程或服务的状况。

18.4.7 手工修改生成代码
Manually Modified Generated Code

有时需要手工修改生成的代码。例如，一个框架将包含生成的函数或方法，而函数体或方法体必须由一个程序员来实现，其结果就是文件将既包含生成代码，也包含手工创建的代码。

最初这不会有问题。一般来说，你只生成框架代码一次，所以，不存在手写的修改会被未来的生成所覆盖的风险。但是，当由于源模型的修改，生成代码必须被改变时，情况

会怎样？你必须重新生成框架代码，当然，你不想丢失自己手工编码写出的扩展。

处理这个问题有许多方法。

使用不同的文件

最好的办法是努力将生成的代码以如下的方式组织起来：用于手工修改的代码，与完全生成的代码，这两者在物理上互相隔离。例如，你可以生成一个基类，包含所有不允许改变的代码，或者包含默认行为，然后，派生一个类，在其中加入手工编码的扩展或修改。

另一种常用的方法是使用“被保护区域”（有时也称为“被保护地区”或“被保护地带”）。在生成的代码的内部，你标记出每个被保护区域的开始点和结束点，表明哪儿允许进行手工修改：

```
class
package ...;
public interface CustomerService extends ... {
    public static public java.lang.String getServiceName() {
        return "CustomerService";
    }

    public void changeAddress (Address newAddress) {
        // ==> Begin Protected Area CustomerService.changeAddress

        // ==> End Protected Area CustomerService.changeAddress
    }
    ...
}
```

未来生成的框架代码将只更新这些被保护区域之外的代码，这些区域内部手工书写的代码将保持不变。当然，进行复杂的修改时，这个方法有其局限性，但是，对小修改来说，它能工作得很好，因为你通常用一个独特的 ID 来标识每个被保护区域，而这个 ID 与源模型中的特定项相匹配（当然，这意味着你的模型必须支持这些 ID）。

生成一次

第 3 种方法是避免重新生成框架代码。用这种方法时，你只生成第一个版本的代码，在这之上进行手工修改，未来对生成代码的任何修改（与相应的模型修改相一致）都必须手工进行。理想情况下，所需的修改是必须的，如果不修改，代码将不能工作或编译。然而，你必须理解所有的模型修改对生成代码的影响。

在某种意义上，这种方法与第一种方法相似，用第一种方法，你也是分离出需要手工更新的、只生成一次（或从未生成）的代码。

合并生成代码与手工代码

笔者曾经见过一个方法，它试图使用配置管理工具的合并工具将框架代码的修改和手工扩展合并起来。

例如，你可能有两个不同的版本分支，一个服务于生成的代码，另一个服务于生成代码的手工扩展。当生成了框架代码的一个新版本时，你将其加到框架代码分支上，再将这个修改与生成代码的手工扩展分支进行合并。对自动化合并和手工合并你都可以使用这个政策。当然，这个方法能工作得多好依赖于合并工具的质量。

这些方法没有一个是完美的，你必须找出哪个方法对自己的特定情况来说是最佳选择。注意，所有方法共享同一个重要的目标：尽可能最小化和隔离手工修改。这就是说，尽可能地将生成代码和手写代码分离到不同文件中，每个这样的分离都将增强可维护性。

18.4.8 代码生成粒度的两难困境
The Code-Generation Granularity Dilemma

服务应该是自足的，因此，对每个服务用不同的文件通常说来是个好主意。另外，对从这些文件生成的制品也作相应的分离可能也会有所帮助。当源文件变化时，这有助于隔离必需的生成。

例如，如果针对两个不同的 WSDL 文件所指定的两个不同的服务，代码生成的结果是“一个”Java 或 C++文件，在此情况下，两个服务中任一个服务的任何变化都将创建出被生成文件的新版本。这就造成了一种风险：由一个服务变化引起的错误生成将使两个服务都不可用。除此之外，要删除一个服务，你必须将其代码从生成文件中删除出去，而不是仅仅删除相应的生成文件。最后，你无法简单地只对其中一个服务执行更进一步的处理。

分割不同服务的代码很显然是个好主意。除此之外，你可能还想分割：

- 不同平台的代码
- 框架代码和存根代码
- 声明、接口和实现

遵循这个建议，模型驱动过程很容易生成出成百上千的文件。笔者所知道的一家公司对每个服务创建了多达 80 个文件：

- 对每个平台创建框架、存根、测试代码，以及针对上述某些文件的哑元实现和公共接口文件。
- 对服务中使用的每种数据类型都创建一个文件。

- 构造文件（比如 makefile）也是生成的。
- 针对 C++平台可能创建额外的文件，因为代码被分割到头文件和定义文件中。

这种产生大量文件的情况当然有缺点：理解代码的组织很难，在创建、部署和运行代码时也要处理大量的文件，这都要花费大量的时间（既有编译时间又有运行时间）和空间。

找出正确的粒度并非易事。代码的良好组织——让你的目录结构与流程结构一致——对此有帮助，这是将转换的源和结果分离到不同目录中的原因之一。（该分离的另一个更重要的原因是，你应该能够看到源文件和目标文件的差异，这样才能通过删除相应的目标目录来删除特定的生成结果。）

18.4.9 部署和健壮性
Deployment and Robustness

正如笔者在本书中通篇都在阐述的，SOA 这个概念支持所有各异的系统。通常情况下，你无法保证所有服务参与者（供应者和消费者）都同时部署修改，每个系统可能有自己的（模型驱动的）软件开发过程，所以，进行修改要花的时间可能各不相同（见图 18-11）。

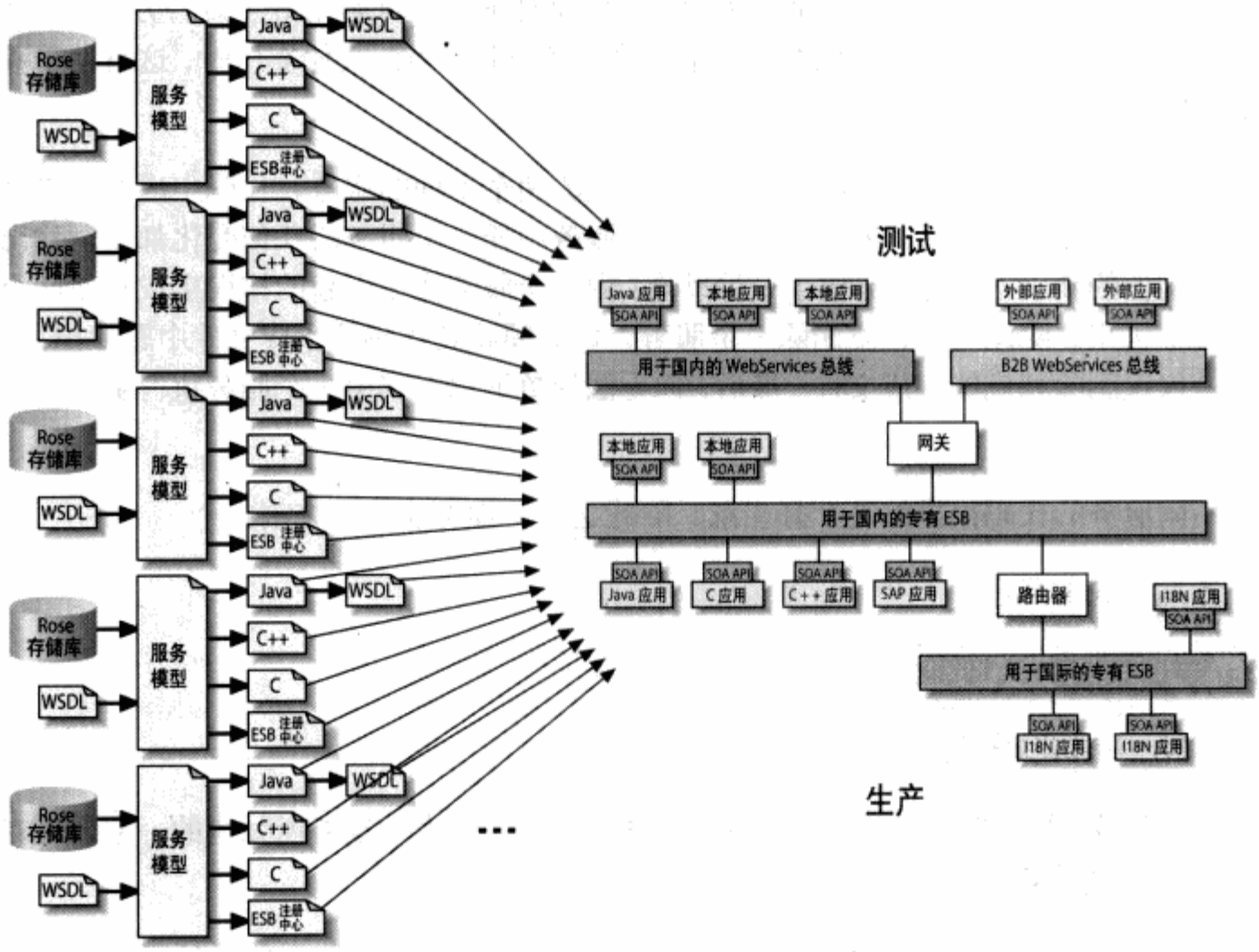


图 18-11：多个不同的系统部署相应的数据

这个考虑有两个重要的结果：

- 一个企业服务总线至少在测试环境中必须能处理不一致性。如果供应者在开发时修改了一个服务，然后部署了它，这不应该导致该供应者的所有服务再也无法工作，或者一个使用该服务的消费者无法调用接口依然匹配的其他服务的情况出现。这就是说，ESB 的接口一定要能够暂时不匹配。当然，当一个接口不匹配的服务被调用时，会导致运行时异常。
- 做向后兼容的修改而影响了所有服务的情况应该几乎不可能发生，这包括对元模型或基本数据类型（它可能是也可能不是元模型的一部分）做向后兼容的修改。由于这个原因，基本数据类型应该非常稳定（关于该话题的更多讨论见 15.5.1 节）。除了诸如字符串、整数、布尔值和浮点数这些类型外，将其他类型用作常用基本数据类型时，你要非常谨慎。注意，类似于将时区语义引入进来用作基本数据类型的修改，可能会破坏整个系统。

18.5 工具
Tools

到目前为止，笔者还没有讨论模型驱动开发的工具。在当代，几乎所有建模语言都使用基于 XML 的格式，所以，一个用于转换和/或代码生成的工具很可能是：**扩展样式表转换语言**，或者 XSLT。然而，XSLT 有其局限性，这点笔者将接着讨论，然后指出另一个选择。

18.5.1 XSLT

一个 XSLT 处理器可以将一个或多个 XML 文件转换为一个或多个结果文件，其格式可能是 XML，也可能是任何其他文本形式，转换方式由一个或多个 XSLT 样式表文件所定义。这些文件提供了结果代码的模板，模板中点缀着可用于对输入记号进行循环的控制结构，以及可以用从源文件继承的特定数据进行填充的占位符（见图 18-12）。

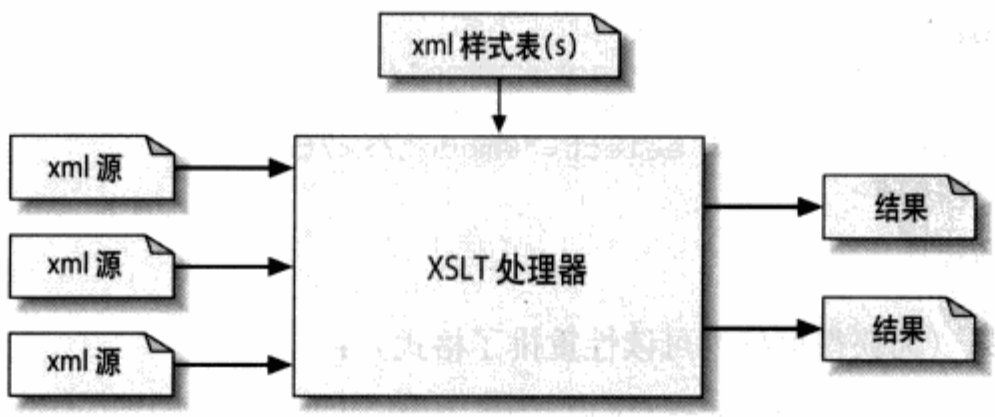


图 18-12：XSLT 转换

例如，假设你有如下的服务描述：

```
<service name="getContractData">
  <param mode="in">
    <name>contractId</name>
    <type>long</type>
  </param>
  <param mode="out">
    <name>firstName</name>
    <type>string</type>
  </param>
  <param mode="out">
    <name>lastName</name>
    <type>string</type>
  </param>
  <param mode="out">
    <name>isActive</name>
    <type>boolean</type>
  </param>
</service>
```

依据下列样式表进行转换：

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="/service">
  <html xmlns="http://www.w3.org/1999/xhtml">
  <head> <title><xsl:value-of select="@name"/>(  )</title> </head>
  <body>
    <h1>Service: <xsl:value-of select="@name"/>(  )</h1>
    Parameters (sorted):
      <table border="1">
        <tr><td><b>Name</b></td><td><b>Type</b></td>
            <td><b>Mode</b></td></tr>
        <xsl:apply-templates select="param">
          <xsl:sort select="name" />
        </xsl:apply-templates>
      </table>
  </body>
  </html>
</xsl:template>

<xsl:template match="param">
  <tr><td><xsl:value-of select="name"/></td>
      <td><xsl:value-of select="type"/></td
      <td><xsl:value-of select="@mode"/></td></tr>
</xsl:template>

</xsl:stylesheet>
```

将产生如下结果（为获得更好的可读性重排了格式）：

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head> <title>getContractData(  )</title> </head>
<body>
```

```
    <h1>Service: getContractData(  )</h1>
    Parameters (sorted):
    <table border="1">
      <tr><td><b>Name</b></td><td><b>Type</b></td>
          <td><b>Mode</b></td></tr>
      <tr><td>contractId</td><td>long</td><td>in</td></tr>
      <tr><td>firstName</td><td>string</td><td>out</td></tr>
      <tr><td>isActive</td><td>boolean</td><td>out</td></tr>
      <tr><td>lastName</td><td>string</td><td>out</td></tr>
    </table>
  </body>
  </html>
```

针对源模型中所描述服务的一个 HTML 页面，一个可能的屏幕打印见图 18-13。

服务：getContractData( )

参数（排名后）：

名字	类型	模式
contractId	long	in
firstName	string	out
isAcrive	boolean	out
lastName	string	out

图 18-13：所生成 HTML 输出的可能的屏幕打印

XSLT 转换对简单的代码生成和代码转换工作得很好。然而，经验显示，对更复杂的转换来说（比如模型转换和许多文件的代码转换），它并不理想，其原因如下：

- 带着转换规则的样式表模板很难阅读。
- XSLT 不是设计用来从许多文件中读取复杂数据模型，再生成几百个不同结果的。
- XSLT 在读取输入和生成输出之间没有清晰的划分，于是，对每个源模型和目标模型（或生成代码）的组合，通常都需要进行 XSLT 转换。
- XSLT 难以对发生在许多地方的特殊处理进行扩展。
- XSLT 相对缓慢。
- XSLT 只能处理 XML 输入。

然而，XSLT 确实能够工作，笔者见过使用 XSLT 处理成千的文件的实例（尽管经历了一些我们刚刚提过的问题所造成的挫折）。

18.5.2 MDSD 工具
MDSD Tools

更好的选择是使用专用于代码生成和/或模型转换的特殊工具，其中某些工具是开源的。例如，在一个大项目中，我们从 XSLT 切换到一个被称为 openArchitectureWare（见[oAW]）的工具，在代码生成上有了可观的改进，生成器的代码有了好得多的可维护性，生成耗费的时间也少了很多。

正如笔者在 18.3 节中所阐述的，当针对模型驱动的服务开发建立基础设施时，理想情况下，你应该从一个最初的、平台独立的元模型开始，该模型使用至少一种领域特定的语言，比如一种 XML 表示法。但是，你如何才能针对服务创建一个这样的模型表示法？你如何才能将你的模型转换为其他的模型或代码？为了使用 XML 模型表示法来创建一个服务描述，你要么使用一个内建在建模工具中的脚本语言（如针对 Rational Rose 或 Excel 的 Visual Basic）；要么使用像 openArchitectureWare 这样的工具，依靠读入特定工具的输出（如 XMI）或特定的输入格式（如 WSDL）来读入模型，再用 XML 表示法将其写出。此后，如果要进行代码生成，你可以读入 XML 表示法，创建你所需的任何制品。

18.6 避免瓶颈
Avoiding Bottlenecks

再次提请注意，为了伸缩性，大型系统必须避免技术上和组织上的瓶颈。这意味着当建立自己的过程时，你应该避免独身（译注 1）（即对整个系统只存在一份）的工具、过程步骤和制品，不要引入一个模型控制者，以免每个变化在被处理前都要得到其许可（虽然这并不意味着在开发过程中逐个地检查修改没有意义）。同样，当没有与整个 SOA 系统连接时，也要能离线地执行所有转换。

18.7 总结
Summary

- 模型驱动的服务开发（MDSD）是生成所有必需的服务代码（包括框架的实现、存根/代理的调用、测试代码和文档）的完美的方法。
- 源模型应该满足源代码所有常见的要求。
- 建模工具应该满足源代码编辑器所有常见的要求。
- 你应该独立于特定基础设施来定义自己的元模型，并且，通过使用相应的 XML 表示法，将服务规范与生成特定于基础设施的代码和其他制品的过程解耦。
- 提供所有必需的生成器还不够，建立合适的过程（定义谁生成什么，以及何时进行）也是 MDSD 的一部分。

译注 1：Singleton，独身，一种设计模式，它使用一个有全局可见性的单独实例，提供唯一的访问点。

- 能够匹配属于同一软件版本的制品非常重要，即使这些制品是在不同的时间，由不同的人所生成（或创建）的。
- 一个企业服务总线应该有能力处理这样的事实：不是所有相应的更新都在同一时间得到部署。
- XSLT 是生成代码的方法之一，但并不一定是最好的。虽然它在将小 XML 文件转换为其他格式时工作得很好，然而，当处理复杂模型、复杂转换和许多文件时，XSLT 的局限性就变得明显了。

- 能这样配置于同一软件版本的制品非常重要，即使这些制品是在不同的时间，由不同的人所生成（或创建）的。
- 一个企业应该应该有能力实现这样的需求：不是所有相应的更新都在同一时间得到解决。
- XSLT 是主版代码的方法之一，但并不一定是最好的，虽然它在将 XML 文件转换为其他格式时工作得很好，然而，当处理复杂模型、复杂转换和许多文件时，XSLT 的局限性就变得明显了。

第 19 章

建立 SOA 和 SOA 监管

Establishing SOA and SOA Governance

SOA 是个战略，你必须逐步地把 SOA 引入到自己的企业和组织中，这需要时间，并且完全成熟的 SOA 需要大量的修改和扩展（不管是在技术上还是在组织上），所以，你不可能一下子得到所有东西。那么，你应该如何开始？你如何保证方向正确？

本章讨论建立/实施 SOA，其中包括“SOA 监管”这个话题。“监管”这个术语通常用来描述为保证 SOA 战略成功，你必须做的一切事情。

19.1 引入 SOA

Introducing SOA

正如你在本书中通篇看到的，SOA 是既包括技术考量又包括组织考量的战略。在技术上，你需要某些基础设施（ESB）以提供互操作性，在组织上，你需要恰当的过程和恰当的组织结构（见第 8 章）。恰当的过程用于明晰如何设计新的解决方案和识别新服务（见第 7 章）、如何经历服务的各个生命周期（见第 11 章）及如何建立相应的软件开发过程（例如，在第 18 章讨论过的模型驱动的开发）。

你也必须作出大量的架构决策，如定义你的服务数据、编程模型和元模型，选择消息交换模式，对服务分类，决定 ESB 是协议驱动还是 API 驱动的，等等。

所有这些方面都相互影响，并且被你的业务和组织所影响。因此，有两件事情是清楚的：

- 你不可能买来 SOA。这就是说，你无法买来某个特定的产品（它可能在自身内部作了大量的架构决策，要求采用某些特定的流程），然后把它用到你的组织中。
- 你不可能预先作出所有决策。

引入 SOA 唯一正确的方法是让它自己一步步成长，正如在[Spanyi03]中所写：

> 业务流程的性能越来越依赖于技术的运用和流程自动化的实现。把这条原则贯彻到实践中的过程仍然要按迭代的方式进行。

19.2 SOA 监管
SOA Governance

在对 SOA 的热炒中，最近有大量关于引入 SOA 时需要监管的文章和建议。例如，Gartner 的研发副总裁 Paolo Malinverno 在[Gartner06]中如此写道：

> 面向服务的架构的监管不是可选项，它绝对必要。SOA 越大，就越需要更多的监管，监管的角色和机制也势必更复杂。监管方面的安排要花长时间来设计和安装，并且难以执行，但是，如果没有它们的话，所有超过了先驱阶段的 SOA 项目都将面临风险。

但是，到底“SOA 监管”意味着什么？谁对它负责？笔者将在接下来的各节中讨论这些问题。

19.2.1 监管的定义
Definition of Governance

关于在一般的意义上监管是什么，以及专门的 SOA 监管又是什么的问题，笔者喜欢 Anne Thomas Manes 在 OOP2007 上给出的简单定义（见[Manes07]）。根据她的定义，监管是：

> 确保人们做“正确”的事情。

对于监管意味着什么，她还给出了补充的、稍微具体点的提示：

> 控制软件的开发和运行。

从这个观点来看，进行监管永远都是绝对必要的。否则，陷入混乱和/或浪费资金、资源的风险就太高了。

至于监管 SOA 意味着什么，让我们努力找点更详细的答案，即你如何确保就 SOA 而言，人们“做‘正确’的事情”？

在[Manes07]中，Manes 定义了如下的监管基础：

- “政策”定义了什么是正确的。
- “过程”执行政策。
- “度量标准”提供了可见性，对政策的执行进行验证。
- “组织”必须建立支持监管过程的文化。

当然，还有很多其他的监管定义，试图形成一个相关问题的完整列表几乎是不可能的。然而，毫无疑问的是，SOA 监管主要集中在非技术性考量上。对基础设施（ESB）和诸如业务库这样的技术细节做出限制是必要的，然而，如果你不知道为什么使用它们，何时使用它们，以及谁应该使用它们，这样的细节限制就毫无价值。

根据自己的经验，笔者通常将 SOA 监管定义为处理如下非技术性问题的任务：

愿景、目标、业务案例和出资模式

为了回答为什么引入 SOA 这个问题，重要的是讲清楚愿景、目标和业务案例。决定出资模式也是很重要的，它决定了如何为最初引入 SOA 和新服务发生的开销付费。

参考架构

你需要一个参照物来展示自己的基础架构决策，包括偏好的技术、消息交换模式、元模型，等等。

角色和职责

你需要定义角色和职责，从而明确由谁来关注问题、推动问题。例如，你必须明白地指出谁来做 SOA 的架构决策，ESB 的职责到哪儿为止，以及谁来负责做识别新服务的高层解决方案设计。

政策、标准和格式

架构决策、制品和技术都带来政策和需求，这些都应该使用标准集和专有的格式进行定义。

过程和生命周期

你必须随着角色、职责和政策一起定义解决方案、服务等的过程和生命周期。

你应该按何种顺序处理这些问题？虽然你应该总是用一个愿景和目标来开始一项新事

物，然而，继承自第一个试点项目的参考架构是引入 SOA 的一个不错的出发点，角色和职责也该一早就阐述明白，将这些同目标一起展示出来将有助于带来先前提过的，能起支持作用的组织文化。政策和过程随着时间演化，往往最后才成形（这并不意味着对成功实施一个企业范围的 SOA 战略来说，其他因素比政策和过程更重要些）。

总有不止一个过程

很多组织花费了大量的金钱和精力来定义企业范围的过程，但是，到目前为止，在笔者见过的所有公司中，这些过程基本上都没被实现。笔者看见的情况是——而且对处于完全不同领域的组织来说情况也是如此——实际上存在着 3 种过程：

- 首先，有一个“正式的过程”，经常也被称为企业范围的过程，它往往都没有被实现。大多数时候，开发人员知道这个过程，但是他们也承认，自己从来没有把它实际用到项目上过。有时情况甚至更糟，开发人员甚至不知道有这个过程。
- 其次，有一个“认为的过程”，它是每个人都相信在使用的过程。不时地也出现认为的过程就是正式的过程的情况，但比较罕见。最经常的情况是，认为的过程是由一厢情愿的想法和误解所“创造”出来的。
- 再次，有一个“实际的过程”，它是实际应用的过程，也是实际用于完成任务的过程。

正如笔者在[Eckstain04]中指出的，“重要的是要理解，只有实际的过程才在实践中扮演了一个角色。其他所有过程只在理论上存在。”

因此，找出实际是哪个过程在产生价值就非常重要，然后，要把那个过程显性化。将实际过程显性化在两个方面会颇有裨益：一方面，它降低了新人融入团队的难度，另一方面，它使实际过程能按一种受控的方式改善。然而，就算已经将实际过程显性化了，你也必须明白：每个过程的生命期都有限，如果大家学会了一种能更好地支持工作的新方法，他们会切换到该方法上。这就（再次）让你面对着这样的情况：出现了未使用的过程，然而团队却更为高效。所以，正如将实际过程显性化一样，你应该明确地规定，团队要不时地检视过程，反映可能的改进。通过这种周期性的检视和反映（在敏捷编程世界中，这被称为“回顾”），团队将频繁地更新实际（现在希望它也是正式的）过程。这当然也意味着正式的过程必须足够灵活，允许进行频繁的更新。

——Jutta Eckstein

除了非技术考量外，实践中对监管的技术考量也有助于支持非技术性问题。监管的技术考量包括：

文档

对“透明性”来说，文档扮演着重要的角色。这是指文档有助于支持所有的非技术性监管问题（过程、职责、政策，等等）。

服务管理

对服务和服务契约的管理来说，类似于业务库和注册中心这样的工具可能会有帮助。

监测

监测是验证规则、政策、契约等的关键任务。例如，你可以监测违反 SLA 的情况，或者监测其使用情况，这样就可以撤销不再使用的服务。

变化和配置管理

通常的配置管理工具也能有助于管理 SOA 软件、SOA 制品和 SOA 文档。

对这些监管的技术考量来说，诸如业务库（见第 17 章）这样的特殊 SOA 工具，以及诸如配置管理和监测工具这样的普通工具都能有所帮助。因此，工具供应商往往把注意力集中在 SOA 监管的这些技术考量上，这种现象并不让人感到奇怪。然而，要注意，SOA 监管包含比上述项目多得多的内容。

19.2.2 中央 SOA 团队
Central SOA Teams

监管需要人来执行，这是指你需要一群人来实施控制，协调实施 SOA 景观和战略的所有步骤。通常有一个中央团队来负责建立和监管 SOA，这样的一个团队可能被称为“**SOA 能力中心**”（SCC）、“**SOA 卓越中心**”（SCoE）、或者“**集成能力中心**”（ICC），并且，它可能在一个“**SOA 委员会**”或“**SOA 筹划指导委员会**”的指导下，从整体角度建立、控制和决策 SOA 战略，这也是“建立 SOA”和“经历 SOA”这些运作方面的东西得以交汇的地方。

在本章的剩余部分，笔者将使用 SCC 这个缩写词来指监管 SOA 的团队。

SCC 协调、巩固、标准化，甚至在一定程度上控制着整个 SOA 战略，并且，执行这些任务要求本着实用主义精神和具备专门的技能。从概念上讲，SCC 是集中运作的，然而，不要掉进建立一个新的集中控制象牙塔的陷阱。

SCC 应该是一个由前述的不同的驱动者——开发者、业务专家和 IT 专家——组成的虚拟团队。然而，记住：建立 SOA 是一个大任务。一旦你把这些专家汇聚到一起，你就必须确保他们有足够的时间和其他所需资源，能够完成建立和监管 SOA 时涉及到的任务。

从字里行间，你可能已经读出了笔者在这里想传达的主要信息：引入和建立 SOA 是一个必须被正确处理的任务。笔者经常看到这个任务被分配给错误的人员。引入服务要求相关负责人具备面向服务的思维方式，对建立和监管 SOA 的人员也有同样的要求。

为了支持面向服务的思维方式，中央团队必须有足够的时间到各个业务团队中去，主动帮助他们使用和体会 SOA。当 SOA 战略成长时，执行第一个试点项目的人往往会成为业务团队的“养父母”。对这样的事情笔者经验颇丰。这是为什么需要充足的资源的一个主要理由。

注意，对在 SCC 中的人来说，一个有趣的心理体验是，在某种意义上，他们必须以“让自己不再必需”为目标。为什么会这样？可伸缩性是 SOA 的关键，集中化导致瓶颈，所以必须避免这样的瓶颈。如果一个供应者和一个消费者想交换数据，必然是个双边或多边的过程。虽然某些集中决策确有必要，然而，集中化必须在日常工作中被最小化。这就意味着，在理论上，如果 SCC 将自己的工作做好了，它自己也就没必要存在了。在现实中，因为需求和上下文环境在变化，而完美性又代价高昂，所以这种情况永远不会发生。然而，端正自己的思想态度仍然是一个重要的事情。

19.3 按部就班 SOA
SOA Step-by-Step

现在，让我们再深入一点，讨论引入 SOA 的一种可能的方法。要注意，该方法不是“唯一”的可能，与其说它是一个如何采纳 SOA 的指南，不如说它是一个迭代的建立过程的范例。然而，在如何找到适合你自己的过程这个问题上，它也许能够给你一些启发。

19.3.1 步骤一：理解 SOA
Step 1: Understand SOA

为了引入 SOA，第一个必要的步骤是理解它。因为 SOA 是股热潮，所以有不少人针对 SOA 说了很多愚蠢的话。不要因为你主要的业务分析师推荐了 SOA，所以你就引进 SOA，要因为你“看到”了好处才这样做。不幸的是，这个好处通常不太容易看到，学习小系统和大系统的区别、中央控制和分布控制的区别是关键所在（例如，见第 1 和第 4 章）。

除此之外，在你开始着手实干之前，你的管理层要接受 SOA 是一个战略而不是一个现成的解决方案的想法，要了解实施这个战略的可能后果（尤其是分布式的处理意味着什么，见第 8 章），这所有的一切都至关重要。当然，在管理层次上，你没必要讨论版本划分和幂等性这些细节，重要的是，管理层要理解松耦合的概念及其后果。

另一个重要的信息是，SOA 不是针对任意类型的系统集成的银弹，确切说来，它是处理分布式业务流程的概念，数据库复制和解耦前端与后端是完全不同的任务（虽然它们可能使用同样的模式或技术）。

19.3.2 步骤二：SOA 试点项目
Step 2: The SOA Pilot

引入 SOA 的第 2 个步骤通常是试用 SOA。这里的意思是，开展一个试验项目，目标是将某种功能——该功能包含影响到两三个不同系统的业务流程——带进生产阶段。要注意，这不仅是个原型而已。SOA 原型往往被如下问题困扰：到一定的复杂度之前，所需要的工作量都比较小，事情看起来不错，但是，当超过这个复杂度后，所需要的工作量突然急剧上涨，其上涨速度远远快于从中获益的增长速度，到最后，所有东西都崩溃掉（这通常以“曲棍球方程”为人所知，我们在第 1 章一开始作了介绍）。

在这个最初的项目中，我们主要集中于技术和架构决策。然而，它应该由业务需求驱动。这意味着实现这个试点项目的团队应该混合了来自 SCC 的 IT 人员与来自业务团队的业务人员。

在扩展你的 SOA 景观的最初阶段，要小心谨慎。下面是应该遵循的一些一般性指导方针：

- 只使用基本服务，这意味着从基础的 SOA 开始（见 6.2 节）。
- 只使用一个或两个消息交换模式（MEP）。笔者建议从同步或异步的请求/应答模式开始（见 10.2.1 和 10.3.1 节）。
- 定义一个基础数据类型的最小集合，以及聚合这些数据类型的语言结构的最小集合（见 15.5 节）。
- 选择用于 ESB 的中间件技术（例如，Web Services，见第 16 章）。
- 决定松耦合的程度（见第 4 章）。对松耦合要保守些，小心 YAGNI（“You Ain't Gonna Need It”，你不会需要它）原则。
- 考虑你现在和今后想要提供的安全性和可维护性的程度。注意我说的是“**考虑**”，这意味着即使第一个试点项目不必实现它们，你也应该把这些主题装在脑子里（“从大处考虑，自小处着手”）。

最重要的着眼点（除让软件能运转之外）应该是在 ESB 和业务之间提供优良的接口。在此你没必要对过程作出决策，首先你需要能工作的软件，然后你可以看看如何在一个分布式软件开发过程中创建、生成或实现它。

试点项目的开发本身当然也应该是一个迭代和渐进的过程，它可能包含一个原型、一个最初的解决方案，以及一个最终确定的解决方案。

从业务的观点来看，你不应从一个非常复杂的组合或流程服务开始，试点项目的目标是用新 SOA 基础设施提供某些典型的后端功能——也许你将把某些现有的远程数据库访问替换为服务。被实现的功能应该和业务有某种联系，虽然选择公司的关键业务功能可能风险太高，然而，实现一些有用的、实际所需的功能将有助于让 SOA 方法集中在实用性和可用性上。

要注意的是，第一个解决方案即使实现了，它也不会稳定。在执行后续步骤时，你必须对其进行重构。因此，这个试点项目不应该包含作为固定价格合同的一部分的代码，对这种代码，每做一点修改都必须支付酬金。

19.3.3 步骤三：第二个与第三个 SOA 项目
Step 3: The Second and Third SOA Projects

在引入 SOA 时，第二个、第三个项目是非常重要的阶段，此时，你将开始考虑所有能带来协同效应的重要事项。考虑服务的可重用性，为不同的团队建立过程，在集中和分散之间找到正确的平衡，在理论、概念和实践之间找到恰当的妥协点，所有这些都变得重要了。

当越来越多的项目被引入进来时，多项目管理的通用方法也开始发挥作用。图 19-1 通过基于 3 个试点项目的公共基础设施展示出了这一点（注 1）。第一个试点项目随着基础设施成长，第二和第三个试点项目基于以前在基础设施上的经验、决策和实现，但它们需要，同时也增加了新的、对整体基础设施有影响的特性，其结果是最初的基础设施发生了变化。

要注意，这些变化并不仅仅是增加新特性。变化的主体是升级，这往往基于合并和通用化的考虑。随着你的不断使用和经验增长，有用的接口、流程和政策会逐步演化出来。

在这里可以应用极限编程社区所说的“连来三次”模式（见[ThreeStrikes]）：任何时候，只要第 3 次复制了某工作，你就应该做些什么来将过程自动化。然而要注意，有两种类型的复制：

- 如果你在“**三个不同的时间**”做同样的事情，应将其自动化。这作为“连来三次就要自动化”模式为人所知。这种自动化带来更多、更好的模型驱动的开发（见第 18 章）。

注 1：本图基于[KrafzigBankeSlama04]的图 12-7，对其的使用和修改获得了作者慷慨的许可。

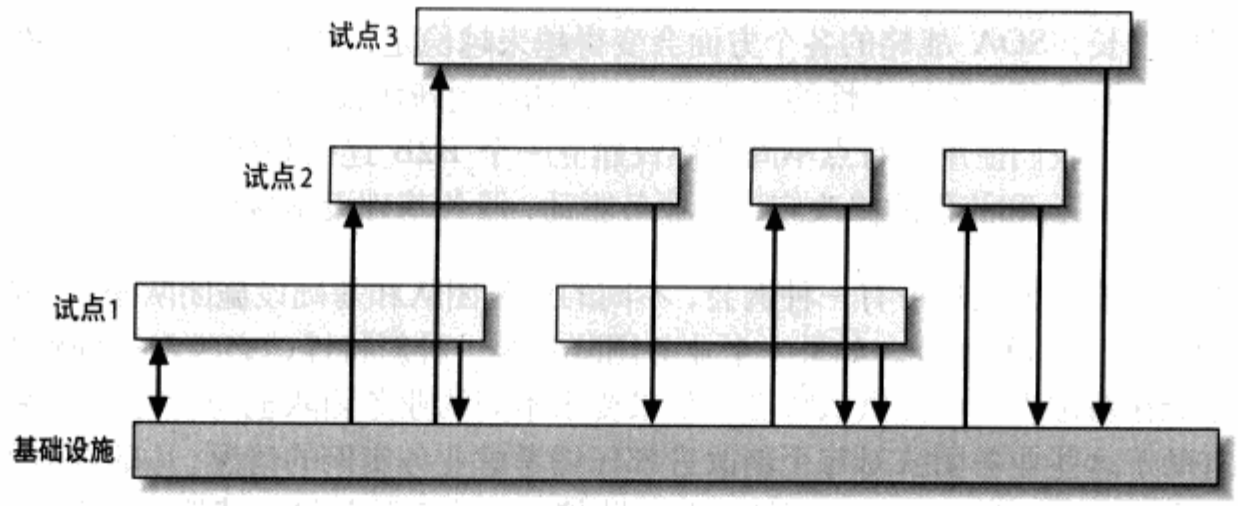


图 19-1：基础设施和多个项目之间的依赖关系

- 如果你在“三个不同的地方”做同样的事情，应重构自己的代码。这作为“连来三次就要重构”模式为人所知。这些重构带来变化的接口、政策和流程。

进行优化，避免重复做事是达成优秀的最佳途径。这样做的缺点是你并非从一开始就能做到优秀。然而，要注意，大型项目的编程模型通常基于“拷贝和粘贴”方法。因此，要从改进中受益，你必须对已经完成的试点项目进行重构，否则，人们将从老的、过时的代码和 API 中拷贝。重构已经完成的项目需要资源，但是，因为实施 SOA 是一种投资，所以，相应的资源应该作为 SOA 战略的一部分得到保障。

“连来三次”模式不光可以运用到基础设施考量上，你可以把图 19-1 中的“基础设施”一词换成“过程”、“生命周期”、“政策”、“架构”等等。重要的是，SOA 各方面都随着时间流逝而不断演进，这意味着为得到更好的一致性、可维护性，以及（最后但并非最不重要的）更好的主拷贝，你最好具备修改现有实践和代码的资源。

19.3.4 步骤四：成长和成为总体战略

Step 4: Grow and Become a General Strategy

随着时间的推移，SOA 将成为你企业的一个总体战略。迟早有一天，你将不得不开始管理你的 SOA，业务库（见第 17 章）将有助于管理服务和服务契约，业务活动监测（见 5.4.7 节）将使你的公司能快速察觉业务机会和挑战。

在此关键是，建立新业务功能的常用方法做到了完全的分散化和尽可能的自动化。为达到这个目标，中央服务团队必须做足功课。这需要资源，不要在此时停止对 SOA 的战略性支持。

随着规模的增长，SOA 战略的各个方面会变得越来越稳定，然而，总会有需要处理的新需求。这些需求可能是技术性的（“我们应该切换到新协议标准上吗？”），或者是业务驱动的（“我们使用稍有点不同的协议建立一个 B2B 连接如何？”）。这里的关键是，对 SOA 平台和流程的修改应该由业务驱动，服务基础设施应该服务于业务团队。

在笔者工作过的一个项目中有一种例会，不同的业务团队和基础设施团队坐在一起，讨论对 SOA 基础设施的修改。如果一个业务团队需要一种新特性，必须在得到别人的同意后，才能在基础设施中提供该特性。如果要求不同的业务团队也为新特性做些工作，该机制有助于这些业务团队对每个新改进都仔细考虑业务案例的情况。基础设施团队自己也可以提出改进建议，但是必须被业务团队接受，从而避免象牙塔效应。当然，这个方法要求合作，也要求业务团队即使自己没有得到直接利益，也能够而且愿意接受改进工作。正如 8.1.3 节所讨论的，利润中心方法和封地在这里起的是反作用。在一定程度上，所有部门必须都能从公司整体出发来考虑问题。

19.3.5 对所有步骤都必要的任务
Tasks Necessary over All Steps

有一些必要的任务覆盖了这里提到的所有步骤，并随着这些步骤成长，这些任务中最重要的是文档和评审。

对当前的 SOA 景观应该总有某种类型的文档描述，这应该包括“五页最重要的幻灯片”，它解释了概念和主要的决策（就像笔者在 1.5 节展示出的那样，但通常会更具体些）。除此之外，应该有文档和幻灯片用来给项目涉及到的每个人介绍 SOA，为他们解释应该了解的东西，从而让他们理解自己的角色和任务。这个文档应该包括攻略和菜谱。然而，要注意文档不应篇幅过大，没人会去读 1000 页的文档，这里的重要任务是（为此需要付出些努力）提供清晰、准确、简练和完整的文档。

另外，笔者建议为评审和回顾留出时间。评审检查的是你有什么，它是否合适，你需要什么，以及接下来你该做什么。评审可以在内部或外部做。内部评审不一定非得是专家来评审新手或普通的程序员，颠倒普通情况下的评审上下级关系也很重要，因为这能保证专家提供了合适的、可理解的概念。当然，外部评审有助于保证不会重犯前一个项目所犯的错误。除此之外，即使是专家也难免掉入犯错而不自知的陷阱，所以，外部评审可以应对这种风险。

回顾是元级别的评审即它们检查完成任务时涉及的人和过程。例如，一个评审检查是否使用了正确的技术，而一个回顾检查的是选择正确技术的方法是否恰当。

对战略项目来说，评审和回顾非常重要，作出错误的决策、选择不恰当的过程来建立新技术和概念，会花费数额惊人的金钱并带来巨大的风险。这意味着参与到评审和回顾中来的不应仅仅是系统架构师和程序员，SOA 是一个对组织有整体影响的战略，所以，评审和回顾的参与者应该包括所有人和牵涉到的“利益攸关方”。

19.4 其他 SOA 方法
Other SOA Approaches

我们刚刚看了引入 SOA 的一种可能的方法，但现在，笔者想要提提从其他来源找到的另一些方法和建议。当然，这个列表既不完整，也决非完美，你必须找到适合自己的建立 SOA 的方法，但是，下面的例子和观点对你也许会有所帮助。

19.4.1 Pulier 和 Taylor 的“四 P”法
The “Four Ps,” by Pulier and Taylor

针对最初的步骤，[PulierTaylor06]建议“四 P”最佳实践：人员（People）、先驱（Pilot）、计划（Plan）和动手（Proceed）。笔者喜欢这个想法，对其元素的意义稍做了些变化：

- 所有“人员”都必须学习 SOA 的原则及其相关技术。
- 一个“先驱”项目有助于验证概念。
- 一个灵活的、多阶段的“**计划**”有助于将 SOA 战略集成到日常业务中，并用日常业务进行验证。
- 基于前 3 个 P，你通过实际的开发“**动手**”建立 SOA。

19.4.2 Krafzig、Banke 和 Slama 的架构路线图
Architectural Roadmap, by Krafzig, Banke, and Slama

在[KrafzigBankeSlama04]中，作者定义了有 3 个扩展阶段的架构路线图：

- 基础的 SOA（只有基本服务的 SOA）
- 联合的 SOA（有组合/配制服务的 SOA）
- 流程使能的 SOA（有流程服务的 SOA）

细节见第 6 章。

19.4.3 Krafzig 和 Slama 论述的自顶向下或自底向上

Top-Down or Bottom-Up, by Krafzig and Slama

在[KrafzigSlama07]中，作者讨论了实施和监管 SOA 的不同方法（注 2）。一个自底向上的方法（见图 19-2）从一小群 SOA 热心者开始，开发者使用它开发了越来越多的成功的项目，使其最终扩展到整个企业范围。

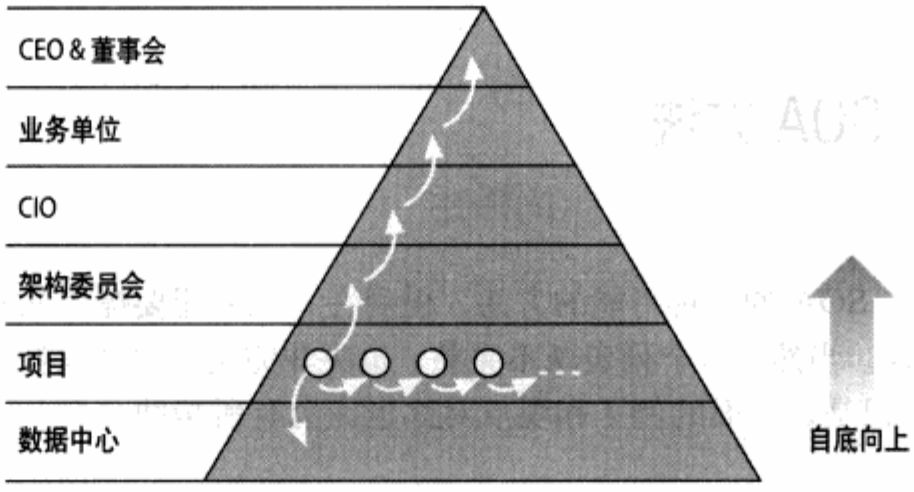


图 19-2：实施 SOA 的自底向上法

一个（理想中的）自顶向下方法（见图 19-3）由最高管理层发起，定义目标、考虑业务案例、落实出资，然后建立 SOA 战略（包括建立架构、组织结构和基础设施），最后开始 SOA 项目。

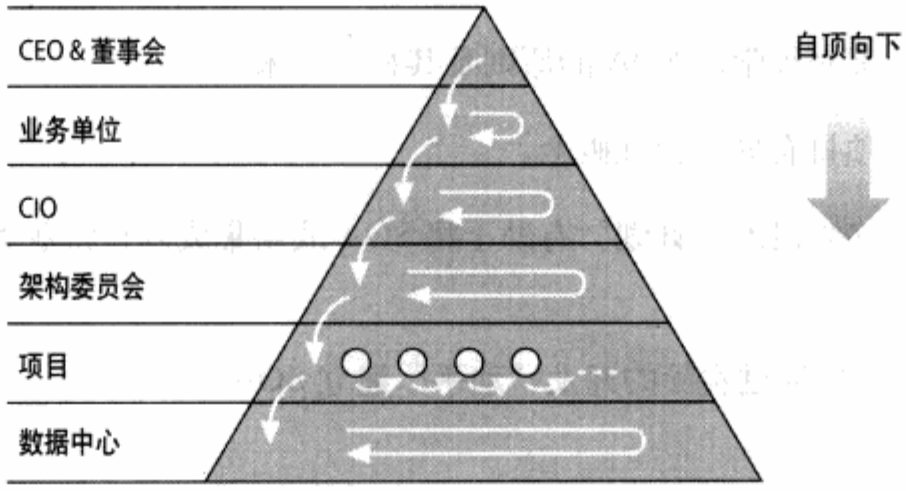


图 19-3：实施 SOA 的自顶向下法

“理想中”这个限制条件暗示着，在实践中这个方法几乎从来不起作用（分散和所有者各异是大型分布式系统的特征）。在一定程度上，你既需要中央资源，又需要来自高层管理的支持，只有这样，类似于“支持性的组织结构重组”才能展开。

注 2：供职于德国 BusinessGlue GmbH 公司的 Dirk Krafzig 和 Dirk Slama 慷慨地允许我使用实施 SOA 的自底向上图和自顶向下图。

因此，通常情况下，一种结合了自顶向下和自底向上的方法（这可以被称为“中间立场”方法）才是合适的。

毫无疑问，你应该按部就班地建立 SOA，但你迟早会需要管理部门和业务部门的支持。哪件事情先做并不那么重要，在一定意义上，这只决定了诸如 SCC 这样的中央服务团队和必需的资源是由官方来建立，还是以非正式的方式，多多少少在 CIO 和 CEO 的非官方支持下逐步推进。

19.4.4 Anne Thomas Manes 论述的 SOA 驱动者

SOA Drivers, by Anne Thomas Manes

Anne Thomas Manes 得出了类似的结论。根据[Manes07]的说法，在 SOA 背后有 4 种典型的驱动力量。SOA 可能是：

开发者驱动（草根）

该方法能增长经验和培养技术专家，但是包含着不能协调一致及合作性不佳的风险。

业务项目驱动

该方法通常的结果是证明了某些概念，或者开展了一个试点项目。这种做法一旦成功，将鼓励对 SOA 更多的接纳。然而，第一个孤立实施的项目可能好处有限，所以你需要采取措施将 SOA 方法扩展到跨组织的合作上。

IT 驱动

该方法针对基础设施方面有效，但隐含着只集中注意力于基础设施方面，从而走进象牙塔的危险。

高层管理驱动（巨型项目）

该方法由业务优先驱动，通过高度协调的努力，力图自顶向下贯彻 SOA。然而，基于从一开始就做对所有事这种理念产生的方法，通常是昂贵、有破坏性，并且风险巨大的。

这些方法中通常没有一个可以单独成功，实际上，你需要所有这些方法的混合。SOA 从一开始就要求合作。换句话说，最好的方法是结合了开发者驱动的、业务项目驱动的，以及 IT 驱动的方法，同时牢记这是一个战略的第一步（从总体上来看这是个巨型项目，但不应该从一开始就作为巨型项目来处理）。

19.4.5 不同 SOA 方法的本质

Essence of Different SOA Approaches

如果说所有这些建立 SOA 的不同方法有潜在的相同本质，那就是如下这些：

- 你必须用迭代和渐进的方式，按部就班建立 SOA。
- 迟早有一天你需要管理层的支持。
- 当建立一个新战略时，你需要强有力的、能发挥作用的领导力。

- 对最初的试点项目要当心。它们必须成功，还要和一定的业务有关联，但它们不应该牵涉关键业务。
- 成功的关键因素是合作。

19.5 其他建议
Additional Recommendations

针对引入 SOA 这个话题，在本节笔者想提出其他一些考量。

19.5.1 政策
Policies

SOA 政策是建立、经历 SOA 所必需的规则和指导方针，这些政策从技术和组织的角度定义了什么是正确的东西。

如前所述，政策应该随着时间流逝不断演进，因为只有在它们被证明是恰当的时候，政策才有用，没有人能预见到所有形形色色的需求、上下文环境细节及 SOA 战略风格。

要注意有两类政策：强制性的和作为目标的。正如 Anne Thomas Manes 在[Manes07]中描述的，一个政策可能是“规定”，也可能是“指导方针”：

- 一个“**规定**”是你必须遵守的东西。规定应该包括对 SOA 目标而言所有关键的需求。例如，你可能要求每个服务供应者都必须使用一个特定的协议（如带着 document/literal wrapped 绑定的 Web Services 协议）。如果所有基础设施组件在实现时都遵循这条限制，那这就应该是个规定。
- 一个“**指导方针**”是你建议的某种东西，但是也可以不遵守。例如，一个指导方针可能是，服务应该是无状态的或幂等的。

讨论政策是强制性的或仅为建议性的，通常导致过程和工具在多处得以澄清。规定带来更简单的解决方案，因为你可以忽略其他的可能性。另一方面，规定潜在的缺点是，它们使开发更复杂，而且可能对某些需求并不合适。

例如，考虑第 12 章讨论的服务版本划分政策。一个服务最多应该或必须有多少同时可用的版本？如果你对此作出规定，就需要相应的过程来处理在运行中出现了版本数量超过允许值的情况。如果你对此作出的是指导方针，则应该监测违规情况，但不必在违规时坚持那个最大值。因为在实践中，撤销一个老服务版本虽然没有直接的业务价值，却是对可维护性的一种投资。通常对本例的情况处理为指导方针，而不是规定。

再举一个例子，考虑如何处理这样一个政策：所有服务调用都必须使用一个标准的日志 API 来记录日志。如果一个服务消费者不遵守这个政策会如何？它会被禁止使用服务

吗？或者使用异常？当然，最好的政策是一旦实现，就无需任何工作量的政策。例如，如果一个基础设施提供了记录服务调用日志的标准机制，这就可以轻易地定为规定而非指导方针。

政策必须针对不同的主题来定义，这些政策带来不同的结果。在笔者工作过的一个项目上，我们有一个“政策表”，它的各列列出了如下信息：

- 主题
- 处理这个主题的政策（方法）
- 政策是否为强制的（是一个规定还是指导方针）
- 对基础设施/ESB 的影响结果
- 对服务供应者的影响结果
- 对服务消费者的影响结果

表 19-1 给出了一个示例，从中可以看到政策表的某些项目是怎样的（不要把这张表的内容当作建议，该表只包含为展示概念而虚构的例子）。

表 19-1：一个政策表可能的格式

主题	政策	强制性	基础设施	供应者	消费者
私密性	DMZ。消费者必须保证不可能有误用	是	提供审计（监测）	无关	必须确保服务调用和处理结果是被允许的
MEP	只允许同步的请求/应答和非同步的订阅/发布模式	是	只对这两个 MEP 提供 API	必须对每个服务指定 MEP	必须使用相应的 API
版本数量	生产中只应有三个服务的版本	否	无关（从技术上来说服务版本是不同的服务）	必须将老服务版本打上过时标记	只要可能，必须升级到新的服务版本
幂等性	服务应该是幂等的	否	对使服务幂等提供为可选的支持项	必须标记应成为幂等的服务，并且提供存储数据的方法以应对重入	对幂等的服务调用必须使用特殊 API/程序库

对给定的主题而言政策是什么，以及对基础设施、供应者和消费者来说该政策意味着什么，只要你能用一种形式使得这些东西是透明的，那么，政策描述的确切形式无所谓。

19.5.2 与人打交道
Dealing with People

引入诸如 SOA 这样的新战略是一种影响到组织中几乎每个人的变化，产生的后果和引发的反应将会多种多样，可能有人对此热情高涨，可能有人对此害怕和抗拒，可能有人心存怀疑，也可能有人想要保持或获得控制，因此，变化往往难以发起，要维持变化甚至更难。

Mary Lynn Manns 和 Linda Rising 所写的《无须畏惧的变化》（Fearless Change，Addison-Wesley，见[MannsRising04]）一书，针对在所有规模的组织或团队中实施变化，提供了几个被证实为有用的、被系统地表述成了模式的技巧。笔者强烈建议阅读此书。

19.5.3 SOA 成熟度模型
SOA Maturity Models

实施 SOA 有不同方法和不同的扩展阶段（见第 6 章），所以，成熟度模型的概念，比如能力成熟度集成模型（CMMI）和软件能力成熟度模型（SW-CMM；以前的 CMM）被转换到 SOA 上就并不令人感到惊讶了。

CMMI（见[CMMI]）处理 5 个成熟度级别。

- 第一级：初始
- 第二级：受管理
- 第三级：已定义
- 第四级：量化管理
- 第五级：持续优化

相似地，SOA 成熟度模型处理 SOA 概念的不同成熟度级别。

例如，2005 年出现的 Sonic SOA 成熟度模型（细节见[SOAMM]）列举了如下成熟度级别（见图 19-4）（注 3）：

- 第一级，当你提供的服务“体现了采纳 SOA 时最初的学习和最开始的项目阶段”时。
- 第二级，当你提供的服务使用了“被设定来对 SOA 的实施进行技术监管”的标准时。

注 3：使用于此的图基于 Sonic 软件公司、AmberPoint 公司、BearingPoint 公司和 Systinet 公司发布的新 SOA 成熟度模型©2005，感谢它们的慷慨许可。上述公司保留一切权利。

- 第三级，当你提供的服务“源于技术组织和业务组织间的合作，目的是保证 SOA 的使用提供了明确的业务响应能力”时。
- 第四级，当你集中于“实施内部和外部业务流程”时。
- 第五级，当你拥有“优化后的业务流程 SOA”，以致“SOA 信息系统已经成了‘企业神经系统’”时。

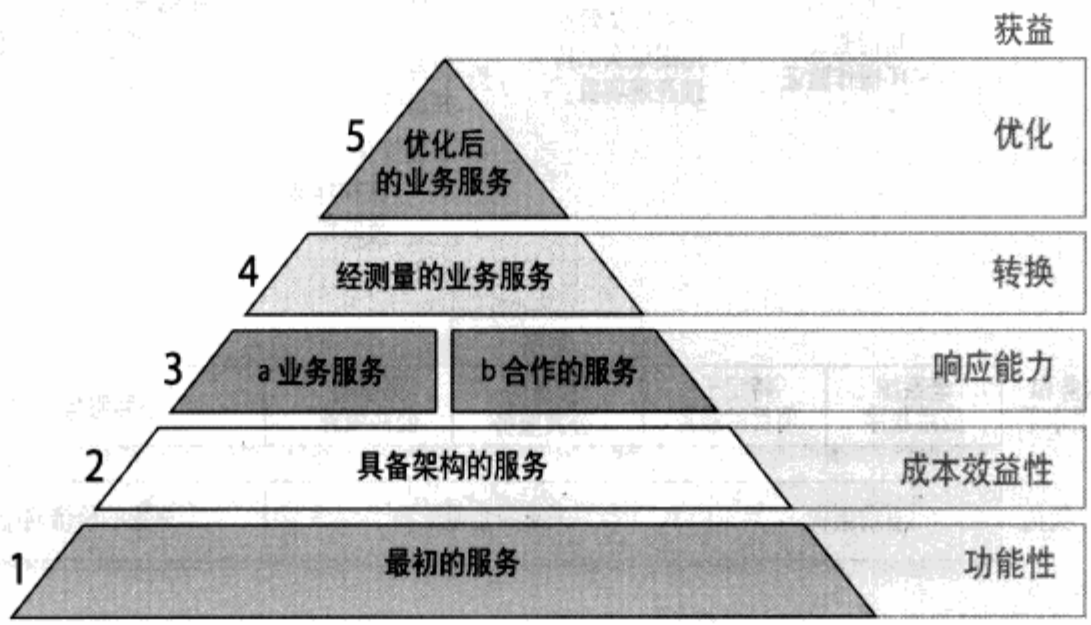


图 19-4：Sonic 软件的 SOA 成熟度模型

IBM 开发了另一个模型。以下内容来自其网站（见[SIMM]）：

> 在 IBM，我们致力于研究一个成熟度模型，以及为达到理想的成熟度阶段所需的过程。这个模型被称为“服务集成成熟度模型”（SIMM）。根据每个成熟度阶段所达到的松耦合级别及灵活性程度，我们划分出 7 个成熟度级别。
>
> 1. 单个软件（数据集成）
> 2. 已集成（应用集成）
> 3. 已组件化（功能集成）
> 4. 简单服务（流程集成）
> 5. 组合服务（供应链集成）
> 6. 虚拟服务（虚拟基础设施）
> 7. 可动态配置服务（生态系统集成）

基于此，IBM 引入了一个“SOA 采纳过程的渐进领域”模型（见图 19-5）（注 4）。

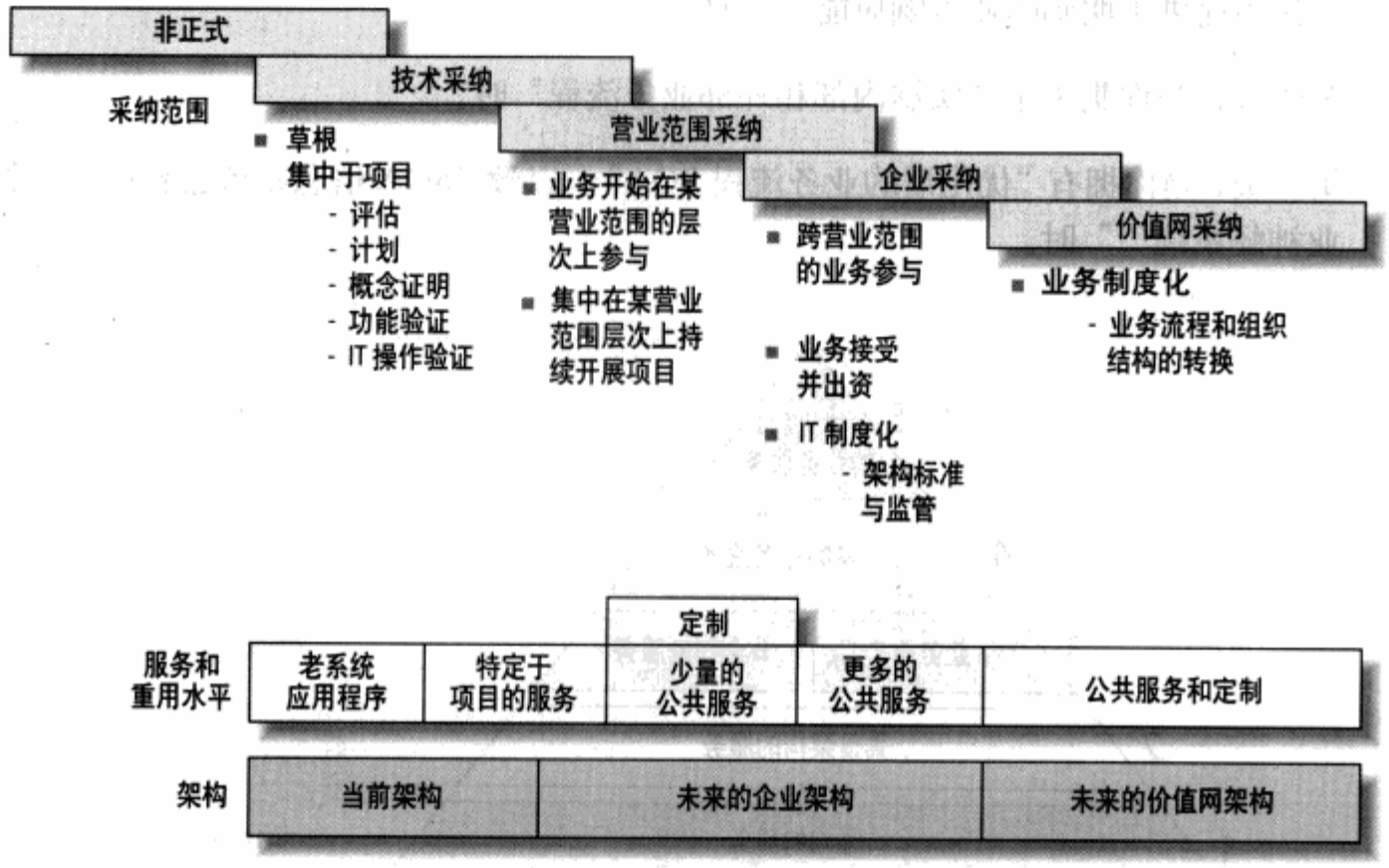


图 19-5：IBM 定义的 SOA 采纳领域

你必须自行决定 SOA 成熟度模型这个思想是否有用。成熟度模型的好处是再一次展示了建立 SOA 是一个渐进的过程，但它的问题是可能导致这样的印象产生——达到最高级别的成熟度很重要，并且在任何情况下都是如此。然而，灵活性、松耦合和分布有其缺陷，所以，对每一级别成熟度，业务案例都应被仔细监测。

19.6 总结
Summary

- 你买不来 SOA，你只能买到某些帮助你实施 SOA 战略的工具。
- 为避免混乱和浪费资金、资源，有必要进行监测。
- 你需要一个中央 SOA 团队来监管 SOA 的建立和经历过程。然而，这个团队必须将自身理解为业务团队的服务供应者，并且在理想中，应该以将自身变得不再必要作为目标。
- 极强的领导力加上管理层支持，对 SOA 的成功至关重要。
- 合作是成功的关键要素。

注 4：此图的使用感谢 IBM 的 Ali Arsanjani 给予的慷慨许可。

- 从大处考虑，自小处着手。谨慎处理最初的试点项目，要认识到 SOA 将会随着你对它经验的增长而演化。
- 使用评审和回顾（包括内部的和外部的）来保证你有正确的决策、解决方案、优先级和过程。

- 从大处考虑，自小处着手。清洁生产是新的研究点而且，要从削减 2CA 将会影响等对它经济的潜在影响。
- 使用环境问题（污染内部的冲洗的）来保证非有正确的步骤，解决方案，优先和过程。

第 20 章

后记

Epilogue

现在笔者已经深入展示和讨论了 SOA，对 SOA 常常被问的几个问题，是回头想想答案的时候了。

20.1 SOA 是新东西吗？

Is SOA Something New?

SOA 没有引入任何新发明的概念，它是个把现有概念和实践放到一起，用于特定需求集的范式。你甚至可以说 SOA 别的什么都不是，就是将实用主义和头脑风暴运用到分布式系统上而已。看看现有的成功的、已集成的大型系统景观，你会找到在本书中提到的所有概念。

SOA 的改进之一可能体现在如下事实上：Web Services（不谈其所有的缺陷）引入了一个互操作性的新标准。

SOA 另一个重要的方面是对异质的接受，这一点体现了 SOA 是一个革命性的方法，不同于我们以前经常看到的东西。过去，相当多的解决方案都基于同质化想法，然而，只要系统超过一定的规模，同质化就不可能。同质化也无法伸缩，这意味着任何要求同质

化的方法迟早会失败（注 1）。接受异质化改变了我们设计大型系统景观的方法，这个思维方式的转换也许是一小步，但它能产生戏剧性的效果（类似于敏捷编程接受需求不断变更的事实，而不是力图与这个事实抗争）。

20.2 SOA 增加了复杂性吗？
Does SOA Increase Complexity?

毫无疑问，分布式处理比非分布式处理更复杂。要处理大型系统和不同的所有者，需要合作与松耦合。但是，正如笔者已经指出的，所有形式的松耦合都增加复杂性。除此之外，比起所有事情都由中央控制的环境来说，分布式调试和测试都要求付出更多的努力。

正由于此，SOA 自身从来都不应是目的，如果你能避免分布式处理，就尽量避免它。

然而，如果你被要求将业务流程分布到在不同的所有者控制下的系统上，那么，SOA 能帮你处理这个难题所涉及到的复杂性（从而在某种意义上减少了复杂性）。

20.3 SOA 关键的成功要素是什么？
What Are the Key Success Factors for SOA?

如同任何新战略一样，要使 SOA 成功，有很多重要的方面需要注意。你必须一开始就有恰当的理解、现实的预期、正确的人选，你还必须努力工作，找出和规划适当的技术和过程。因此，就这个问题而言，笔者通常的回答是包含了如下要素：

理解

你必须事先就清楚 SOA 是个远远超越基础设施范畴的概念。

合作

你必须努力创建一个人人都愿意帮助他人、为他人服务的组织文化氛围。

管理支持

管理层必须被说服以提供足够的支持（时间和资金），并且在需要时能作出恰当的组织决策。

谨慎的引入

对 SOA 的引入，应该允许用三年或更多的时间，而且你必须培养根据最初的经验进行调整的能力。

钻研

对达成卓越应足够用心，如此才能保证所有团队都热心于建造（你的）SOA。

注 1：失败并不一定意味着作为结果的系统不能运行（虽然这是通常发生的情况）。正如 Bjarne Stroustrup 在[Stroustrup00]中所写的："基本上来说，不管一个方法被认为有多糟糕，也不管被牵涉的个人有多痛苦，在大型项目中，该方法看起来总是可以工作的，其前提是你愿意为它造成的问题投入多得过分的时间和金钱。"

所有这些可以浓缩为如下几个字：你买不来 SOA。

20.4 SOA 在哪些地方不合适？
Where Is SOA Not Appropriate?

你可能有这样的印象或期望：作为分布式流程的解决方案，SOA 对不同系统之间任何形式的通信都是合适的。但是，SOA 存在局限性，此需求有可能过于苛刻，或者 SOA 引入的额外开销带来的问题可能比它解决的问题还要多。

数据库复制、海量数据处理和本地客户端尤其不是 SOA 的强项。你有可能从基于 SOA 方法的概念和解决方案中获益，然而，不要忘记，像任何其他概念、技术或范式一样，SOA 也不是银弹。

要注意的是，该提的问题从来都不是“某事用 SOA 是否合适”，而应该是“你正在考虑的解决方案对你的问题和需求是否合适”。

笔者感觉在未来几年内会出现 SOA 热潮的消退。当我们发现分布和松耦合的代价太高之后，可能会退回去使用单个软件（当然，要使用一个新的、积极的术语，比如“总承包商驱动的架构”）。

20.5 SOA 替代了 OOP 吗？
Does SOA Replace OOP?

另一个常见的问题与 SOA 是否替代了面向对象的编程（*OOP*）有关，或者说这两者比起来谁更好些。

Don Woods 和 Thomas Mattern 所著的《企业 SOA》（*O'Reilly*）一书给出了一个有趣的答案。在题为“为什么面向服务比面向对象要好？”的节下有两个小自然段。第一段解释了 OOP 有如下局限性：“调用程序必须用与自己正试图访问的对象相同的语言编写。”第二段仅仅包含了一句话：“Web Services 提供了服务间通信的标准方法，服务可以用不同语言编写。”

这整个讨论当然都是废话。比较 OOP 和 SOA 根本就毫无意义，因为它们服务于不同的目的。

不管是 SOA 还是 OOP 都不比对方强，也无法相互替代。OOP 是一个应用程序的“**编程**”范式，而 SOA 则是一个系统景观的“**架构**”范式。你只能在一定的规模内控制和管理自己的对象，所以 OOP 无法伸缩。SOA 是用来连接系统的方法，这些系统用面向对象或其他范式编写。换句话说，除非你只需管理单个程序，否则这两者你都需要。

所有这些可以概括为如下几个字：脱不了来 SOA。

20.4 SOA 在哪些地方不合适？

你可能有这样的印象或期望：作为分布式应用的解决方案，SOA 对不同系统之间任何形式的通信都是合适的，但是，SOA 并非如此，也绝不可能过于普遍。毫无疑问，要将 SOA 引入的额外开销带来的问题可能比它解决的问题还要多。

数据库复制：有些数据处理和本地客户端程序不是 SOA 的强项。你有可能从基于 SOA 方法的概念和解决方案中获益，然而，不要草率，将任何事情概括，技术就绝对一样，SOA 也不是例外。

要注意的是，这些问题从来都不是"某种用 SOA 是否合适"，而应该是"脱机在本地的解决方案可以解决的问题和需求是否合适"。

毫无疑问在未来几年内会出现 SOA 成熟的消息，当我们度过分布和整合的代价大幅之后，可能会回归使用单个软件（当然，是使用一个新的、智能的系统，比如"总线包商提供的服务"）。

20.5 SOA 替代了 OOP 吗？

另一个常见的问题与 SOA 是否替代了面向对象的编程（OOP）有关，我将尝试阐明此问题的要点。

Don Woods 和 Thomas Mulcern 所著的《企业 SOA》（O'Reilly）一书给出了一个有趣的答案。在题为"为什么面向服务比面向对象更好？"的章节中，两个小自然段，逐一剖析了 OOP 有如下局限性："用用户必须用与自己正在试图协同的对象相同的语言编写"，"要求所有的对象使用相同的语言"，"Web Services 提供了服务间通信的标准方法，服务可以用不同语言编写"。

这些个问题当然是真实的，但将 OOP 和 SOA 做本质上的对比是无意义的，因为它们服务于不同的目的。

不管是 SOA 还是 OOP 都不能对立起来，也不能相互替代。OOP 是一个应用程序的"编程"范式，而 SOA 则是一个系统层级的"编排"范式。你只能在一定的规模内经验和掌握自己的对象，所以 OOP 无法伸缩。SOA 是用来连接系统的方法，这些系统用面向对象或其他范式编写，彼此独立。你非常只需管理单个程序，代码这两者都需要。

参考文献
References

此处的附录列出了本书提到或引用的所有参考文献。当然，我并非宣称这是个全面的SOA 参考文献列表。但是，这些参考文献对我本人和本书而言都非常重要。

通常说来，由于网站比纸质的书和文章更容易变化，所以，列出的因特网链接可能会在未来失效。因此，我将在本书的网站（希望这个站点能一直存在）上维护该参考文献列表。网站的地址是：

http://www.soa-in-practice.com

[ActiveBPEL]
ActiveBPEL Open Source Engine Project. *http://www.activebpel.org*.

[Allen06]
Allen, Paul. 2006. *Service Orientation: Winning Strategies and Best Practices*. Cambridge, UK: Cambridge University Press.

[Bartonitz05]
Bartonitz, Martin. 2005. "BPMS, BPML, BPEL, BPMN, WMS, XPDL,...'Alles so schön bunt hier'" (in German). *http://www.bpm-guide.de/articles/17.*

[Bartonitz06]
Bartonitz, Martin. 2006. "Wachsen die BPM- und Workflow-Lager zusammen?" (in German). *http://www.bpm-guide.de/articles/66.*

[BassClementsKazman03]
Bass, Len, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice*, Second Edition. Boston, MA: Addison-Wesley.

[BloombergSchmelzer06]
Bloomberg, Jason and Ronald Schmelzer. 2006. *Service Orient or Be Doomed!: How Service Orientation Will Change Your Business*. Hoboken, NJ: John Wiley & Sons.

[Booch06]
Booch, Grady. 2006. Blog for March 2006, "SOA Best Practices," March 11, 2006. *http://www.booch.com/architecture/blog.jsp?archive=2006-03.html.*

[BPDM]
Business Process Definition Metamodel (BPDM). *http://www.omg.org/bpdm/.*

[BPMN]
Business Process Modeling Notation. *http://www.bpmn.org.*

[BrownEllis04]
Brown, Kyle and Michael Ellis. 2004. "Best practices for Web services versioning." *http://www-128.ibm.com/developerworks/webservices/library/ws-version/.*

[Butek03]
Butek, Russell. 2003. "Which style of WSDL should I use?" *http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/.*

[Chappel04]
Chappel, David A. 2004. *Enterprise Service Bus*. Sebastopol, CA: O'Reilly Media.

[CMMI]
Software Engineering Institute of the Carnegie Mellon University, The Capability Maturity Model Integration Web Site. *http://www.sei.cmu.edu/cmmi/cmmi.html.*

[DesignByContract]
"Building bug-free O-O software: An introduction to Design by Contract." *http://archive.eiffel.com/doc/manuals/technology/contract/.*

[Dreyfus06]
Dreyfus, Paul et al. 2006. "What's the best software to implement as a service if you're just starting SOA?" *http://www-128.ibm.com/developerworks/library/ar-itio4/.*

[ebXML]
Electronic Business using eXtensible Markup Language. *http://www.ebxml.org.*

[Eckstein04]
Eckstein, Jutta. 2004. *Agile Software Development in the Large*. New York: Dorset House.

[Erl05]
Erl, Thomas. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ: Prentice Hall.

[ESB Inventor]
"'ESB Inventor' Riddle Solved?" *http://www.businessreviewonline.com/blog/archives/2005/08/esb_inventor_ri.html* and *http://www.looselycoupled.com/blog/lc00aa00109.html.*

[Gartner96]
Gartner (Roy W. Schulte, Yefim V. Natis). 1996. "'Service Oriented' Architectures," Parts 1 and 2. *http://www.gartner.com/DisplayDocument?id=302868* and *http://www.gartner.com/DisplayDocument?id=302869.*

[Gartner03]
Gartner (Yefim V. Natis). 2003. "Service-Oriented Architecture Scenario." *http://www.gartner.com/DisplayDocument?doc_cd=114358.*

[Gartner05]
Gartner (D. Cearley, J. Fenn, and D. Plummer). 2005. "Gartner's Positions on the Five Hottest IT Topics and Trends in 2005." *http://www.gartner.com/DisplayDocument?id=480912.*

[Gartner06]
Gartner (Paolo Malinverno). 2006. "Service-Oriented Architecture Craves Governance." *http://www.gartner.com/DisplayDocument?id=488180.*

[GartnerEDA06]
Gartner (Yefim V. Natis and Roy W. Schulte). 2006. "Advanced SOA for Advanced Enterprise Projects." *http://mediaproducts.gartner.com/reprints/oracle/141940.html.*

[GoF95]
Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

[HintonHondoHutchison05]
Hinton, Heather, Maryann Hondo, and Dr. Beth Hutchinson. 2005. "Security Patterns within a Service-Oriented Architecture" (Parts I and II). *http://www.ebizq.net/topics/soa/features/6535.html* and *http://www.ebizq.net/topics/soa/features/6554.html.*

[HohpeWoolf04]
Hohpe, Gregor and Bobby Woolf. 2004. *Enterprise Integration Patterns*. Boston, MA: Addison-Wesley.

[ISO10181-3]
ISO/IEC 10181-3:1996 – "Security frameworks for open systems: Access control framework." *http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=18199.*

[ITSecCity02]
"Konzeption und Umsetzung von Disaster Recovery-Strategien: Stillstand ist tödlich" (in German). *http://www.itseccity.de/?url=/content/fachbeitraege/grundlagen/020315_fac_gru_veritas.html.*

[KanneganticHodavarapu07]
Kanneganti, Ramarao and Prasad A. Chodavarapu. 2007. *SOA Security in Action*. Greenwich, CT: Manning.

[KrafzigBankeSlama04]
Krafzig, Dirk, Karl Banke, and Dirk Slama. 2004. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Upper Saddle River, NJ: Prentice Hall.

[KrafzigSlama07]
Krafzig, Dirk and Dirk Slama. 2007. Handouts of a talk at the 6th OMG Information Days in Germany. SIGS Datacom.

[Levitt01]
Levitt, Jason. "From EDI To XML And UDDI: A Brief History of Web Services." *InformationWeek*, Oct. 1, 2001. *http://www.informationweek.com/shared/printableArticle.jhtml?articleID=6506480.*

[MagicBus]
Lyrics of the song "Magic Bus" by The Who. *http://www.lyricsdir.com/the-who-magic-bus-lyrics.html* and *http://petetownshend.lyrics.info/themagicbus.html.*

[Manes03]
Manes, Anne Thomas. 2003. *Web Services: A Manager's Guide*. Boston, MA: Addison-Wesley.

[Manes07]
Manes, Anne Thomas. 2007. Handouts of a talk at OOP 2007. SIGS Datacom.

[MannsRising04]
Manns, Mary Lynn and Linda Rising. 2004. *Fearless Change: Patterns for Introducing New Ideas*. Boston, MA: Addison-Wesley.

[McKusickBosworth03]
McKusick, Kirk and Adam Bosworth. "Building Web Services: A Conversation with Adam Bosworth." *ACM Queue* Vol. 1 No. 1, March 2003. *http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=29.*

[NewcomerLomow05]
Newcomer, Eric and Greg Lomow. 2005. *Understanding SOA with Web Services*. Boston, MA: Addison-Wesley.

[OasisSoaRM06]
"Reference Model for Service Oriented Architectures," Committee Draft 1.0, February 7, 2006. *http://www.oasis-open.org/committees/download.php/16587/wd-soa-rm-cd1ED.pdf.*

[oAW]
openArchitectureWare. *http://www.openarchitectureware.org.*

[ProjectLiberty]
Liberty Alliance Project. *http://www.projectliberty.org.*

[PulierTaylor06]
Pulier, Eric and Hugh Taylor. 2006. *Understanding Enterprise SOA*. Greenwich, CT: Manning.

[SAML]
The Security Assertion Markup Language. *http://www.oasis-open.org/committees/security/.*

[Sanctum04]
Blind XPath Injection. *http://www.packetstormsecurity.org/papers/bypass/Blind_XPath_Injection_20040518.pdf.*

[SIMM]
Arsanjani, Ali and Kerrie Holley. 2005. "Increase flexibility with the Service Integration Maturity Model (SIMM)." *http://www-128.ibm.com/developerworks/webservices/library/ws-soa-simm.*

[SOAMM]
Sonic Software (Jon Bachman). 2005. "SOA Maturity Model" and "Movin' SOA On Up." *http://www.sonicsoftware.com/soamm* and *http://www.sonicsoftware.com/solutions/learning_center/soa_insights/movin_soa_on_up.*

[Spanyi03]
Spanyi, Andrew. 2003. *Business Process Management is a Team Sport: Play It to Win!* Tampa, FL: Anclote Press.

[StahlVölter06]
Stahl, Thomas and Markus Völter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ: John Wiley & Sons.

[Stroustrup00]
Stroustrup, Bjarne. 2000. *The C++ Programming Language*, Special Third Edition. Boston, MA: Addison-Wesley.

[ThreeStrikes]
Web Services Federation Language. *http://c2.com/cgi/wiki?ThreeStrikes.*

[UBR]
UDDI Business Registry. *http://www.uddi.org/find.html.*

[WfMC]
Workflow Management Coalition. *http://www.wfmc.org.*

[Wong02]
Wong, Wylie. 2002. "Microsoft ploy to block Sun exposed." *http://news.com.com/2100-1001-912906.html.*

[WS-BPEL]
OASIS Web Services Business Process Execution Language. *http://www.oasis-open.org/committees/wsbpel.*

[WS-BPEL4people]
WS-BPEL Extension for People. *http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/.*

[WSDL 1.1]
Web Services Description Language (WSDL) 1.1. *http://www.w3.org/TR/wsdl.*

[WSDL 2.0]
Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. *http://www.w3.org/TR/wsdl20-primer/.*

[WS-Federation]
Web Services Federation Language. *http://www.128.ibm.com/developerworks/library/specification/ws-fed.*

[WSFL]
Web Services Flow Language. *http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.*

[WS-Glossary]
W3C (H. Hass and A. Brown). "Web Services Glossary." *http://www.w3.org/TR/ws-gloss.*

[WS-I]
Web Services Interoperability Organization. *http://www.ws-i.org* and *http://www.ws-i.org/press/pressrelease.aspx?pr=20020206a.*

[WSI-BasicProfile]
WS-I Basic Profile, Version 1.1. *http://www.ws-i.org/Profiles/BasicProfile-1.1.html.*

[WSI-SecurityProfile]
WS-I Basic Security Profile, Version 1.0. *http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html.*

[WS-SecureConversation]
Web Services Secure Conversation Language. *http://www-128.ibm.com/developerworks/library/specification/ws-secon/.*

[WS-Security]
Web Services Security v1.1. *http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.*

[WS-SecurityPolicy]
Web Services Security Policy Language. *http://www-128.ibm.com/developerworks/webservices/library/specification/ws-secpol/.*

[WSStandards]
Web Services Standards Overview. *http://www.innoq.com/resources/ws-standards-poster/* and *http://www.innoq.com/soa/ws-standards/*.

[WS-Trust]
Web Services Trust Language. *http://www-128.ibm.com/developerworks/library/specification/ws-trust*.

[XMLEncryption]
W3C (D. Eastlake et al.). XML Encryption Syntax and Processing, World Wide Web Consortium Recommendation, December 2002. *http://www.w3.org/TR/2002/REC-xmlenc-core-20021210*.

[XMLKeyMan]
W3C (Phillip Hallam-Baker, Shivaram H. Mysore). XML Key Management Specification (XKMS 2.0), World Wide Web Consortium Recommendation, June 2005. *http://www.w3.org/TR/xkms2/*.

[XMLSignature]
W3C (D. Eastlake et al.). XML-Signature Syntax and Processing, World Wide Web Consortium Recommendation, February 2002. *http://www.w3.org/TR/xmldsig-core*.

[WSStandards]
Web Services Standards Overview, http://www.ibm.com/software/ws-standards-poster/ and http://www.ibm.com/software/ws-standards/.

[WS-Trust]
Web Services Trust Language, http://www-128.ibm.com/developerworks/library/specification/ws-trust/.

[XMLEncryption]
W3C (D. Eastlake et al.), XML Encryption Syntax and Processing, World Wide Web Consortium Recommendation, December 2002, http://www.w3.org/TR/2002/REC-xenc-core-20021210/.

[XMLKeyMan]
W3C (Phillip Hallam-Baker, Shivaram H. Mysore), XML Key Management Specification (XKMS 2.0), World Wide Web Consortium Recommendation, June 2005, http://www.w3.org/TR/xkms2/.

[XMLSignature]
W3C (D. Eastlake et al.), XML-Signature Syntax and Processing, World Wide Web Consortium Recommendation, February 2002, http://www.w3.org/TR/xmldsig-core/.

词汇表
Glossary

本词汇表列出了本书使用的特定 SOA 术语中最重要的部分。有时 SOA 的术语没有常用定义，所以这个词汇表也体现了笔者自己对本书使用术语所持的观点。在定义中使用的术语如果本身也在这个词汇表中，则用下划线（译注 1）表示。

Web Sevices 社区有另一个词汇表。见 [WS-Glossary]。

注意，这个词汇表将在本书的网站上维护。

http://www.soa-in-prctice.com

2PC（两阶段提交） 在多个系统间保持一致性的方法。在第一个阶段，所有后端都被要求对所请求的改变做确认，这样在第二个阶段提交更新时通常就能成功。为与**松耦合**的原则一致，在 SOA 中，通常使用**补偿**而不是 2PC。

活动 可以用来指一个**业务流程**中的一个步骤的术语。在 SOA 上下文环境中，一般而言，一个活动由一个**服务**来实现。

代理 在 *Web Services* 术语中指参与者，是既可指消费者又可指供应者的通用术语。

译注 1：原作为“斜体”，但汉字斜体不显著，所以改用**下划线**。

架构 根据[BassClementsKazman03]，计算机系统的架构是指结构或系统的结构，由软件（和硬件）组件、这些组件可为外部所见的特性，以及组件之间的联系组成。

异步通信 一种通信形式，在任何**消息**内容的发送和接收之间都有相当的时间间隔。**面向消息的中间件**通常基于这个想法实现，通过引入消息队列，面向消息的中间件对系统发展的信息进行排队（持久化）处理，直到它们被接收系统接受为止。

异步通信是**松耦合**的一种形式，因为它避免了一个消息的发送者和接收者必须同时可用的要求。

异步请求/应答 **请求／回调消息交换模式**的另一个名字。

后端 为特定**领域**保持数据和业务规则的系统。通常说来，在一个系统景观中，它提供了一个特定的角色或具有特定责任。在 SOA 中，一个后端通常被某种**基本服务**所包装。

基本服务 **服务**的常用术语，提供了单个**后端**的基本业务功能。这些服务通常是最初的服务集的一部分，包装或隐藏了特定后端实现细节。基本服务可能是数据驱动的或逻辑驱动的，它们是组装更高层服务——如**组合服务**和**流程服务**——的基础。

BPEL 业务流程执行语言。一个用来**配制**服务，使其成为**组合服务**或**流程服务**的基于 XML 的语言，产生的服务是 *Web Services*。

IDE 使你能用图形用户界面来创建 BPEL 文件，引擎使你能运行（和调试）用 BPEL 实现的服务。

BPM 见**业务流程管理**和**业务流程建模**。

BPMN 业务流程建模符号，由 *OMG* 维护的业务流程的图形符号。

总线 用于在多个系统间传输数据的抽象软件模式。与**星型拓扑**模式不同，总线使用一个遵循着公共政策或协议的组件联盟来发送、路由和接收**消息**。

业务流程 对为满足某种业务需要，不得不去做的**活动**或**任务**的结构化描述，活动和任务可以是手工步骤（人机交互）或自动化步骤（IT 步骤）。

业务流程可能是被管理的（见**业务流程管理**）和使用诸如 *BPMN*、*EPC* 之类的建模符号，或者诸如 *BPEL* 之类的执行语言实现的。

有人在**工作流**与业务流程之间作出了区分，说业务流程一般描述了必须做什么，而工作流则描述了应该如何执行活动或任务。

业务流程执行语言 见 *BPEL*。

业务流程管理（BPM） 一个通用术语，指为管理（即计划、实现、写文档、观察和改进）**业务流程**所执行的所有活动。

业务流程建模（BPM） 根据[Bloomberg-Schmelzer06]的说法："一系列公司能采取的，描绘或描述**业务流程**各个方面（包括业务流程的流动、控制和决策点、**活动**执行的触发器和条件、活动执行于其中的上下文环境，以及关联的资源）的行动或**任务**。"

业务流程建模符号 见 *BPMN*。

编排 将**服务**聚合为**业务流程**的一种方法。与**配制**不同，编排并不将服务组合为对整个流程实施中央控制的新服务，相反，它定义规则和政策，使不同的服务能够相互协作，形成一个业务流程。每个被牵涉进来的服务都只能看到部分流程，也只对这部分有贡献。

CMMI 能力成熟度模型集成，一个分类和改进组织中产品和软件开发过程的方法。CMMI 是 SW-CMM（以前仅仅称为 CMM）的扩展，CMM 只处理软件开发方面的内容。CMMI 的一部分是用不同级别分类的一个组织的成熟度（"初始"、"受管理"、"已定义"、"量化管理"和"持续优化"）。

补偿 一个在许多系统间保持一致性的方法。与**两阶段提交**相反，补偿并不同步更新所有的后端，相反，它定义了当针对不同系统的相应更新没有成功时（不管更新是顺序执行还是平行执行），应该执行的补偿活动。由此，这个方法带来了系统间的**松耦合**，然而，它的实现可能需要更多工作量。BPEL 对补偿有直接的支持。

组合服务 对于由**基本服务**和其他组合服务组合而成的**服务**的常用术语。

消费者 通用术语，指扮演着调用（"消费"）一个**服务**（这由一个服务**供应者**提供）的角色的系统。另一个用于该角色的术语是（服务）**请求者**。

契约 对一个**消费者**和一个**供应者**之间的**服务**接口的完整描述，它包括技术性接口（签名）、语义，以及非功能性考量，如**服务等级协议**。

有时一个契约也被称为一个"定义良好的接口"。

CORBA 公共对象请求代理架构，一个允许远程访问不同平台上对象的 *OMG* 标准。虽然它最初的目的是提供一个访问分布式对象的基础设施，然而，通过集中于它的传值对象（OBV）上，CORBA 可以被用作 SOA 的基础设施。

领域 一个可定义的、扮演了特定角色和有特定职责的（业务）区域或范围。在 SOA 中，这可能是一家公司、一个分区、一个业务单位、一个部门、一个团队，或者一个系统。

领域特定语言（DSL） 一个用于**元模型**的特定的图形或文字符号，它使你能以精确、简练、可读及完整的形式指定**模型**的具体行为。

EAI 企业应用集成（有时也仅称为企业集成，或 EI），一个让分布式系统使用公共的基础设施（中间件和/或协议），从而对其进行集成的方法。使用这种方法，对每个系统只需提供和维护一个针对基础设施的适配器，不用针对要与之通信的每个系统都提供一个特定的适配器。

基础设施可以使用**总线**或**星型拓扑**方法。

SOA 通常可以被描述为提供了**互操作性**考量的 EAI 扩展，因此，EAI 概念可以被认为是**企业服务总线**的主要部分，甚至就等同于企业服务总线。

EDA 见**事件驱动的架构**。

企业应用集成 见 *EAI*。

企业服务总线（ESB） 实现服务**互操作性**的 SOA 景观基础设施。它的核心任务是提供可连接性、数据传输及（智能）路由，这样系统就能通过**服务**来通信。ESB 可以提供额外的能力来处理安全、可靠性、服务管理，甚至包括流程合成，然而，一个组合服务的工具是否是 ESB 的一部分，或者仅仅是 ESB 之外的、额外的实现组合与流程服务的平台，对此存在不同的观点。

除此之外，工具供应商倾向于将 ESB 定义为某种可以买的东西。你也可以认为一个诸如 *Web Services* 之类的标准是 ESB，因为从概念上来说，这些标准定义了在系统间提供互操作性所必需的东西（不需要买某些特定的硬件或软件）。

一个 ESB 也可能是异质的，使用着各种各样的中间件和通信技术。

你可以将 *EAI* 解决方案当作（部分）ESB。

EPC 见**事件驱动的流程链**。

ESB 见**企业服务总线**。

事件 一个发送给若干人所共知的接收者（**消费者**）集合的通知。通常情况下，一个事件的接收者必须订阅特定的事件类型（由特定系统或组件发送）。依赖于不同的编程模型或系统模型，发送事件的系统（**供应者**）可能知道也可能不知道订阅的接收者，可能同意也可能不同意向其发送事件。

你可以把事件当作**发布/订阅消息交换模式**的一部分。

事件驱动的架构（EDA） 一种鼓励对事件的生产、发现、消费和响应的软件架构模式。有人认为 EDA 是对 SOA 的扩展或补充，另一些人认为 EDA 是 SOA 方法的一部分（一种特殊的**消息交换模式**，在此服务**供应者**向多个**消费者**发送消息）。

事件驱动的流程链（EPC） 一种用于**业务流程**的图形符号，主要由 SAP 推广。

发射后不管 **单程**消息的另一个名字（一种**消息交换模式**，在此一个服务发送消息，但不期望有一个应答）。

前端 一个通过调用必要的**服务**发起和控制业务流程的系统，即它像一个服务**消费者**那样操作。一个前端可能是一个有人机交互的系统，也可能是一个批处理系统。

监管 一般而言，本术语描述了“确保人们做正确的事情”这个任务。在 SOA 中，监管是关于架构决策、**流程**、工具和**政策**的。

HTTP 超文本传输协议，万维网的基础协议。在其安全形式下（使用 *SSL* 传输层安全），它被称为 HTTPS。

星型拓扑 用于在多个系统间传输数据的抽象软件模式。与**总线**模式不同，它使用一个中央组件来协调所有发送者和接收者之间的通信。

IDE 集成开发环境，一个（通常是图形化的）用于特定软件开发的、面向项目的环境。

幂等性 **服务**处理被多次递送的**消息**的能力，如此一来重复递送就不会产生不希望的效果。

在一个不可靠的网络上，如果你没有收到一个确认，你就不知道消息是否被递送了（有可能接收者处理了消息，然后应答丢失了）。如果你再次发送消息（为了确保消息被递送），接收者应该能够正确处理消息，不产生与只接收了一次消息不一样的效果。例如，如果消息是请求向一个银行账户上增加钱，即使由于可靠性的原因，消息被发送了两次，接收者也应该只增加一次钱。

互操作性 不同系统彼此通信的能力。不同平台和编程语言间的互操作性是 SOA 的一个基础目标。

要注意的是，标准不一定保证了互操作性，因此，在 *Web Services* 的世界中，一个被称为 *WS-I* 的特殊组织提供了使标准可以互操作的**配套资料**。

JMS Java 消息服务，用于**面向消息的中间件**（MOM）的标准 Java API。因为它只是个 API 标准，所以它提供了可移植性（允许你在保留接口的同时改变中间件）而不是**互操作性**（允许你使用不同的 MOM 实现）。

松耦合 减少不同系统间依赖性的概念。

减少耦合系统间的耦合紧密程度有不同的方法，比如使用不同的对象模型，使用**异步通信**，或者使用**补偿**而非 2PC 来维护一致性。一般而言，松耦合导致更高的复杂度。因此，在一个特定的 SOA 中，你必须找到恰当的松耦合程度。

成熟度模型 用不同的级别对组织的成熟度进行分类的模型。最著名的是能力成熟度模型（CMM）及其后继者——能力成熟度模型集成（*CMMI*）。遵循这个方法，许多组织开发了 SOA 成熟度模型。

MDSD 模型驱动的软件开发。在此方法中，从一个抽象的**模型**生成出相当数量的纲要代码，这些代码结构相同，但又根据具体情况有所变化。在 SOA 和本书的上下文环境中，MDSD 又可能代表模型驱动的**服务**开发。

消息 作为一个**服务**调用的一部分被到处传送的一个数据。**消息交换模式**定义了执行服务调用的典型消息序列。

消息交换模式（MEP） 对一个服务调用或**服务**操作中**消息**序列的定义，这个序列包括直到一个特定的服务操作完成为止，发送的消息的顺序、方向和基数。

最重要的消息交换模式是**单程**、**请求/应答**和**请求/回调**（异步请求/应答）。

举个例子，请求/应答 MEP 的定义是：一个**消费者**发送请求消息，然后等待由**供应者**以应答消息形式发送的回答。

面向消息的中间件（MOM） 基于**异步通信**概念的中间件。例如 IBM 的 WebSphere MQ（以前的 MQ Series）、微软的 MSMQ、Tibco 的 Rendezvous，以及 SonicMQ。

元模型 对**模型**的描述。一个元模型是一些规则，定义了模型能具备怎样的结构。换句话说，一个元模型定义了模型的正规结构和元素。

模型 一种抽象。在 SOA 中，一般而言，模型被用来阐明**服务**。在 *MDSD* 的帮助下，你能从模型中生成不同的代码和其他制品。模型的结构通常由**元模型**来描述。对模型来说，一般情况下，有一种或多种特定的图形或文本表示法（有时称为**领域特定语言**，或者 DSL）使你能以精确、简练、可读和完整的形式来指定具体的行为。

模型驱动的软件/服务开发 见 *MDSD*。

OASIS 结构化信息标准推进组织，一个致力于电子商务和 *Web Services* 标准的国际非盈利计算机工业联盟。见 *http://www.oasis-open.org*。

OMG 对象管理组织，一个致力于企业集成标准的国际非盈利计算机工业联盟。OMG 的标准包括 UML、MDA 和 BPMN。见 *http://www.omg.org*。

单程 一种**消息交换模式**，在此服务发送一条消息而不期望收到应答。这个模式的另一个名字是**发射后不管**。

配制 一种聚合服务形成**业务流程**的方法。与**编排**不同，配制将服务组合成对整个流程具备中央控制的新服务。

对 *Web Services* 而言，*BPEL* 是配制的标准，而且有开发工具和引擎可用于 BPEL。

参与者 对**消费者**和**供应者**的通称。在 *Web Services* 术语中，还可以选择使用**代理**这个术语。

政策 一个通用的规则或指导方针。在 SOA 中，政策对基础设施（*ESB*）、**供应者**和**消费者**都有影响。一个政策可能是强

制性的规定（比如一种要求遵守的命名约定）或是一个目标（比如运行中服务的最大版本数量）。

过程 执行起来满足某个需要或达到某个目标的一系列结构化的步骤（**活动**或**任务**）。

在 SOA 中涉及不同的过程，其目标是实现**业务流程**。为此，你必须有建立和管理解决方案和**服务**的过程（解决方案生命周期、服务生命周期，等等），在元级别上你也有建立 SOA 和进行 SOA **监管**的过程。

流程服务 一个代表了长期**工作流**或**业务流程**的服务。从业务的观点来看，这类服务代表了一个宏流，它是可以中断（由人工干预）的、长期运转的**活动**（服务）流。

与**基本服务**和**组合服务**不同，本类服务通常有在多个服务调用之间保持稳定的状态。

配套资料 在 SOA 上下文环境中，尤其是在 *Web Services* 中，一个配套资料是一套标准，每个标准都有特定的版本，由指导方针和约定结合起来，目的是以保证**互操作性**的方式来一起使用这些标准。

供应者 通用术语，指扮演着贡献（“提供”）**服务**的角色的（部分）系统。这些服务可以被不同的**消费者**使用/调用。

发布/订阅 一种**消息交换模式**，在此一个**消费者**向一个服务**供应者**订阅，以达到在某种条件满足，或者某种状况发生或变化时，从供应者那里收到通知消息的目的。

订阅可能发生在设计时或运行时。如果供应者不知道消费者，则这个模式是**事件驱动的架构**的基础，在那里通知是一个**事件**。

注册中心 与从业务角度来管理服务的业务库不同，注册中心从技术角度来管理服务。注册中心管理在运行时使用服务所必需的所有技术细节（签名、部署信息，等等）。通常情况下，注册中心被认为是基础设施（ESB）的一部分。

业务库 业务库从业务的角度来管理服务及其制品，即它管理接口、**契约**、**服务等级协议**、依赖性等，目的是帮助识别、设计和开发服务。与**注册中心**不同，服务描述应该独立于技术细节和基础设施考量，这就是说，当一家公司切换到新的基础设施（*ESB*）时，应该不必改变业务库。

请求 在大多数**消息交换模式**（见**请求/应答**和**请求/回调**）中，一条由**消费者**发出的，作为最初的消息的**消息**。

有时这个术语也被用作**服务**调用的同义词。

请求者 **消费者**的另一个可选术语（主要用于 *Web Services* 上下文环境中）。

请求/回调 一种**消息交换模式**，在此一个服务的**消费者**发出一个**请求**消息，但并不阻塞和等待答复，相反，它定义了一个当服务**供应者**发出的**应答**消息到达后才调用的回调函数。

有时请求/回调也被叫做**异步请求/应答**。

请求/应答 一种**消息交换模式**，在此一个服务**消费者**发送一个请求消息并期望得到一个回答。通常情况下，消费者会阻塞，直到从服务**供应者**发出的**应答**消息到达为止。然而，有时并不要求阻塞，此时，在同步和**异步请求/应答**之间有一个区分，后者通常也被称为**请求/回调**消息交换模式。

应答 **供应者**发出的，作为对服务**请求**（见**请求/应答**）进行回答的**消息**。

服务 某种自足的业务功能的 IT 实现。

在技术上，一个服务是对一个或多个使用（多个）**消息**在**供应者**和**消费者**之间交换数据的操作的描述。服务调用的典型效果是消费者从供应系统或组件得到了某些信息，或者修改了的供应系统或组件的状态。

服务有不同的属性，可以归入不同的类别，最有名的分类法区分出**基本服务**、**组合服务**以及**流程服务**。

一个服务通常为一个接口所描述。从消费者的观点（签名和语义）来看，对服务的完整描述也被叫做一个“定义良好的接口”或契约。

服务等级协议（SLA） 两方之间经谈判后的正式协议，在 SOA 上下文环境中，这两方通常指一个服务**供应者**和一个服务**消费者**。对一个特定主题而言，一个 SLA 通常记录了对优先级、职责及保证的共同理解，其主要目的是对服务质量达成一致理解。例如，它可能指定了服务的可用性、适用性、性能、操作或其他属性（如在违反了 SLA 的情况下进行计费甚至采取惩罚措施）。

面向服务的架构（SOA） 对 SOA 有各种各样的定义。有的仅仅指出了它是种“接口即服务”的架构方法。然而，从更特定的意义（并且根据笔者的理解）来看，SOA 是一个处理分布式**业务流程**的架构范式，业务流程分布在处于不同所有者控制之下，由现有系统和新系统构成的、大型的和异质的系统景观上。

SOA 的关键概念是**服务**、**互操作性**和**松耦合**，关键部分是基础设施（*ESB*）、**架构**和**流程**，成功的关键要素是理解、监管、管理支持和钻研。

要注意，*Web Services* 不是 SOA 的同义词，它是实现 SOA **基础设施**的一种可能的方法。

SOAP *Web Services* 的基本协议。作为一种基于 *XML* 的格式，它定义了 Web Services **消息**的头和体的格式。

以前，这个缩写代表着“简单对象访问协议”，但是 SOAP 既不简单，也不用于对象访问，所以这个词现在就只代表自己，不是缩写词了。

该协议仍然允许不同类型的消息交换，最常用的是 document/literal wrapped 模式。

软件架构 见**架构**。

SSL 安全套接字层，一个在因特网 *HTTP* 协议之上提供安全通信的加密协议（加密后的 HTTP 协议往往被叫做 HTTPS）。

任务 可以用于指**业务流程**的一个步骤的术语。在 SOA 上下文环境中，一个任务一般由一个**服务**来实现。

两阶段提交 见 *2PC*。

UBR *UDDI* 业务注册中心，建立于 2000 年，目的是成为公共 Web Services 的全球**注册中心**。然而，这个想法未能成功，UBR 于 2006 年关闭。

UDDI 通用描述、发现和集成，一个用于**注册中心**的 *Web Services* 标准。最初为 UDDI 业务注册中心（UBR）设计，现在它被用作技术管理和 Web Services 中介的标准。

W3C 万维网联盟，一个负责万维网标准开发的国际联盟，同时也开发了诸如 *XML* 和 *SOAP* 这样的 SOA 标准。见 *http://www.w3.org*。

Web Services 实现 SOA 基础设施的一种可能方法的一套标准，它最初从一些核心标准开始，如 *XML*、*HTTP*、*WSDL*、*SOAP* 和 *UDDI*，现在，它包含超过 60 个标准和**配套资料**，由不同标准化组织，如 *W3C*、*OASIS* 和 *WS-I* 开发和维护。

工作流 与**业务流程**类似，对为了满足某个业务需要，必须完成的**活动**或**任务**的一个描述。

有人在工作流和业务流程之间作出了区分，认为业务流程一般描述必须做什么，而工作流则描述应该如何执行活动或任务。

WS Web Services 的通用缩写，也是 Web Services 标准的常用前缀。

WSDL Web Services 描述语言，一种基于 *XML* 的语言，从技术角度描述了**服务**接口。虽然它是一个 *Web Services* 标准，然而，WSDL 也能用于其他基础设施。

WS-I Web Services 互操作性组织，一个开放的工业组织，将 *Web Services* 标准进一步标准化为**配套资料**，使它们支持互操作。见 *http://www.ws-i.org*。

XML 可扩展标记语言，一个广泛用于描述和交换数据的，人类可读的通用表示法，可以根据 *XML* **大纲定义**来定义和验证特定的 XML 格式。

XML 大纲定义（XSD） 一种语言，用来描述一套规则，这套规则是相应的 *XML* 文档如果想要被认为有效，就必须遵守的，它包括一系列基本数据类型。

SSL 安全套接字层，一个在因特网 HTTP 协议之上提供安全通信的加密协议（加密后的 HTTP 协议在此被叫做 HTTPS）。

任务 可以用于指业务流程的一个步骤的术语，在 SOA 上下文环境中，一个任务一般由一个服务来实现。

两阶段提交 见 2PC。

UBR UDDI 业务注册中心，建立于 2000 年，目的是成为公共 Web Services 的全球注册中心。然而，这个想法未能成功，UBR 于 2006 年关闭。

UDDI 通用描述、发现和集成，一个用于注册中心的 Web Services 标准，最初为 UDDI 业务注册中心（UBR）设计，现在它被用作技术管理和 Web Services 中分类标准。

W3C 万维网联盟，一个负责万维网标准开发的国际联盟，同时也开发了诸如 XML 和 SOAP 这样的 SOA 标准。见 http://www.w3.org。

Web Services 实现 SOA 基础设施的一种可能方法的一套标准，它最初从一些核心标准开始，如 XML、HTTP、WSDL、SOAP 和 UDDI，现在，它包含超过 60 个标准和规范资料，由不同标准化组织，如 W3C、OASIS 和 WS-I 开发和维护。

工作流 与业务流程类似，对为了满足某个业务需要，必须完成的所有动作任务的一个描述。

有人在工作流和业务流程之间作出了区分，认为业务流程一般描述业务流程什么，而工作流则描述应该如何执行活动或任务。

WS Web Services 的缩写，也是 Web Services 标准的常用前缀。

WSDL Web Services 描述语言，一种基于 XML 的语言，从技术角度描述了服务接口，虽然它是一个 Web Services 标准，然而，WSDL 也能用于其他基础设施。

WS-I Web Services 互操作组织，一个开源的工业组织，将 Web Services 标准进一步标准化和提供资料，使它们互操作更稳健。见 http://www.ws-i.org。

XML 可扩展标记语言，一个广泛用于描述和交换数据的、人类可读的通用表示法，可以根据 XML 大纲定义来定义和验证特定的 XML 格式。

XML 大纲定义（XSD），一种语言，用来描述一套规则，这套规则是相应的 XML 文档必须满足要被认为有效，就必须遵守的，它包括一系列基本数据类型。

索引
INDEX

Number

2PC (two-phase commit), 293
  compensation, 44, 70
  large systems, rarely used, 44
  problems with, 44
  transaction safety, 70

A

AAA, 174
access control
  decision function (ADF), 180
  enforcement function (AEF), 180
accounting, 174
ACID properties, 64
activation services, 76
ActiveBPEL, 88
activities, 67, 83
activity, 293
actual process, 264
adapter services, 68
addresses (see endpoints)
ADF (access control decision function), 180
advanced ESB (enterprise service bus), 113
AEF (access control enforcement function), 180
AES, 183
agent, 24, 293
agile programming, 282
alignment
  business IT, 2, 83
  domain/IT, 110
analysis paralysis, 39
API-driven
  ESB, 54
  with Web Services, 222
APIs, versus protocols, 133
application services, 76
approaches, to SOA, 281

architecture, 19, 294
bottom-up approach, 272
BPEL (Business Process Execution Language), 89
defined, 21
designs
issues, 73
solutions, 103
domain-driven, 111
EDA (event-driven architecture), 134
decoupling, 135
flexibility, 87
IDE, 86
iterative service development, 138–139
logical models, 109
mixed models, 111
models, 109
modifications, 87
overall, 109
Plug and Play, 88
Rational Rose Repository, 238
reference, 263
technical models, 112
tools, 86
top-down approach, 272
useful assumptions, 96
working systems and, 19
arrays
higher data types, 206
meta models and, 245
artifacts, 155
asynchronous communication, 37, 128, 294
error handling, 131
reply problems, 37
asynchronous request/response, 128, 294
at least once
messages, 201
reliability, 57
atomic, 64, 66
attributes
integers, 169
nillable, 205
required, 164
SAML (Security Assertion Markup Language), 183
of services, 29
skipContractData flag, 166
string, 166
auditing, 174
authentication, 173
SOA and, 174
authorization
SOA and, 174
availability, 174

B

backends, 294
ACID properties, 64
composed services, 68
consistency, 68
corruption of, 66
data access, 114
data modification, 66
defined, 65
frontends and, 114
inconsistencies with, 64, 69
large systems and, 66
multiple, 68
operations of, 64
read or write data, 63
SOA and, 65, 110, 114–117
SOA infrastructures, 63, 110
state and, 71
stateful services, 195
user input, 115
validation, 117
multilayer, 117
prevalidation, 118
workflow, 117
backplane, 47
backward compatibility, 70, 148, 169
backward-incompatibility, overall modifications, 255
BAM (business activity monitoring)
business opportunities, detecting, 58
code generation, 238
detection of unusual behavior, 58
ESB and, 58
message layers, 58
messages, 58
basic data services, 62
basic services, 62, 294
architecture models, 110
performance and, 162
SOA establishment, 267
binary data
exchange, 205
types, 205
binding, 43, 212
business process modeling, 90
infrastructure, 224
layers and, 212
loose coupling, 43
SOAP (Simple Object Access Protocol), 214
WSDL, 211
WSDL lifecycles, 224
blocking request/response, 124
BOM (business object model), 38

Booleans, 204
bottlenecks
large distributed systems, 4
MDSD and, 258
BPEL (Business Process Execution Language), 294
batch language, 114
BPEL4People, 117
BPM, 83
debugging, 251
ESB and, 48, 58
frontends, 114
human interaction, 114
models, 89
nonfunctional attributes and, 172
notation, 89
performance and, 168, 171
variables section, 90
Web Service operations, 90
Web Services and, 90
WS-BPEL (BPEL4WS), 90
WSFL (Web Services Flow Language), 89
XLANG, 89
XML language, 89
BPEL4people, 117
BPEL4WS, 90
BPM (business process management), 294
bottom-up approach, 84
BPEL, 83, 89
defined, 82
engines, 87
flexibility improvement, 13
goals of SOA, 12
layers, 82
mixed approaches, 84
models, 82
Plug and Play, 88
service example, 84–87
service identification, 139
SOA and, 83
system identification, 85
terminology, 82
top-down approach, 84
versus workflow management, 82
(see also business process modeling; business process)
BPMN (Business Process Modeling Notation), 92, 294
brainpower, 281
brokers
history of, 231
indirection and, 52
Web Services and, 219
bug fixes, 141, 147
build team, 250
bulk services, 197
bus, 6, 47, 294
business activity monitoring, 58
business case, 263
business categorization of services, 75
business data, 203
business functionality, 25
business ID, 119
business IT gap, 2
business object model (BOM), 38
business process, 294
backend workflow, 117
BAM and, 58
BOM (business object model), 38
BPM (Business process modeling), 19
business/IT gap, 2
categorization, 76–77
decomposition, 84
departments or units, defined, 58
distribution
multisystem, 9
objects, 23
engine protocol, 48
frontend workflow, 117
functionality, 2, 17, 26, 54
future management, 120
IDs, 202
incident management procedures, 104
interfaces
business-driven, 27
technically driven, 27
interrupting, 115
logging and monitoring of, 57
macro flow, 71
maintenance, 8
mapping business and protocol APIs, 56
MDSD (model-driven service development), 19
micro flow, 67
modeling standard, 91
platform-specific interfaces, 55
process chain, 135
protocol code layers, 55
routing messages, 56
services
lifecycles, 19
management, 57
SOA and, 8
statelessness, 32
teams and, 55
value-added, 3

Business Process Execution Language, 294
business process management (BPM), 295
business process modeling, 295
binding, 90
BPM, 82
composition, 88
defined, 82, 87
funding for, 106
graphic models, 86
IDEs, 86
MDSD (model-driven software development), 19
Oasis SOA Reference Model, 13, 20
Plug and Play, 88
rule engines, 86
services and, 86
standards
BPDM (Business Process Definition Metamodel), 92
BPEL (Business Process Execution Language), 89, 92
BPMN (Business Process Modeling Notation), 92
BPSS (Business Process Specification Schema), 92
Business Process Management Initiation, 92
EPC (Event-driven Process Chain), 92
EPML (EPC Markup Language), 92
implementation problems, 92
OMG (Object Management Group), 92
UML (Unified Modeling Language), 92
WfMC (Workflow Management Coalition), 91
Wf-XML, 92
WPDL (Workflow Process Definition Language), 91
WS-CDL (Choreography Definition Language), 92
XPDL (XML Process Definition Language), 91
structure, 90
tools, 86, 88
Business Process Modeling Notation (BPMN), 92, 295
business rules
basic data services, 63
basic logic services, 110
recommendations, 274
redundancy, 116
business services, 76
business strategy of SOA, 266, 269
business unit, as domain, 111
business-project-driven SOA, 273
businesss/IT gap, 88

C

caching
on consumer side, 120
provider side, 167
stateful services, 197
call center, portal and, 119
call constraints, 166
versioning, 156
callback function, 128
calling services
attributes, 157
automatically canceling, 226
call constraints, 156, 166
multiple consumers, 156
EDA (event-driven architecture), 134
ESB responsibility for, 47
flags, 156
interoperability, 48
mapping, 54
routing, 48
runtimes, 156
safe calls, 225
special contexts, 156
unsafe calls, 226
validation, 116
capability, 13
Capability Maturity Model Integration (CMMI), 276
cardinality of messages, 123
categorization of services, 61
CBR (content-based routing), 56
centralization, 18
central teams, 265
versus commonality, 114
repositories, 234
certificates, 174
change-management
governance and, 265
system maintenance, 143
choices, 206
choreography, 295
EDA and, 136
Choreography Definition Language (WS-CDL), 92
Churchill, Winston, 4
classification, of services, 61
client, 7, 24
CMMI (Capability Maturity Model Integration), 276, 295
coarse-grained, 30, 163
code
debugging, 251
functionality, without commonalities, 239

generation
manual modifications, 251
service implementation and calling, 237
source models and, 242
generic, 239
granularity problems, 253
injections, 186
manually modified, 251
schematic, 239
service implementation, 237
transformation, 242
(see also MDSD, model-driven service development)
collaboration
distributed systems and, 104
key success factors, 274
key success factors and, 282
withdrawing services, 143
commodity services, 77
common data types, 204
commonality versus centralization, 113
company
as domain, 109
effectiveness of, 102
compensation, 44, 70, 295
loose coupling, 44
complexity
asynchronous communication, 128
distributed processing and, 282
loose coupling, 9
SOA and, 282
composability
business functionality, 32
of services, 88
composed services, 295
backends
and inconsistencies, 69
using a single, 68
using multiple, 68
business process chain versus, 135
compensation, 70
domains, 111
examples of, 68
federated SOA, 69
frontends, 67
micro flows, 67
orchestration, 67
security, 70
concepts of SOA (Service-oriented architecture), 16
confidentiality, 174
SOA and, 175
configuration management
governance and, 265
labeling, 251
versioning, 155
conformance, 36
Conformance Requirements, 221
connectivity, 48
consistency
backends, 68
error handling, 69, 131
real-world scenarios, 4
consistent, 64
consumer-driven transformations, 247
consumers, 295
basic services, 67
brokers and, 231
defined, 24
differentiation of, 74
international services, 74
role systems, 24
security issues, 74
services
categories, 74
implementation, 74
internal versus public, 74
specific services, 74
stability, 74
versus provider role of a system, 125
Web Services and, 218
WSDL and, 225
consumer-specific services, 167
content-based routing, 56
context of SOA (service-oriented architecture), 109
contracts, 26, 140, 157, 295
modification, 138
controller services, 76
cookie, 195
coordination of services, 234
coordinator services, 76
copy-and-paste programming, 269
CORBA, 23, 219, 295
correlation IDs, 128, 202
counter-hype of SOA, 262
cross-domain departments, 103, 111
customized services, SOA and, 167

D

Darwinism, 1
data mapping (see data types, mapping)
data transformation (see data types, mapping)
data types
aliases, 206
APIs and, 152, 154
arrays, 206
compatibility, 152
defined, 206
definition, 206
balance of design, 207

data types *(continued)*
domain-specific, 205
enumerations, 206
fundamental, 204
fundamental versus common, 204
generic, 154
higher, 206
IDs and, 202
in meta models, 245
Integer, 205
libraries, make consistent, 153
lists, 206
mapping, 40, 48, 56
meta model example, 244
modified types, 150
null semantics, 205
numeric, 204
options for, 150
paradigm, 43
parameters, 154
pros and cons, 154
records, 206
reference semantics, 205
service providers, consequence of using, 151
upgrading services, 152
structures, 206
use of different types in different versions, 150
use of the same type in different versions, 152
validation, and attributes, 152
versioning policy, 149
WSDL, 214
XSD, 206
database access, 162
data-centric services, 76
date/time, 205
debugging, 57
code generation and, 251
complexity and, 282
decentralization, 1, 10, 269
decision points, 118
decision service, 181
decomposition, 84
decoupling (see loose coupling)
definitions of SOA (Service-oriented architecture), 11
demilitarized zone (DMZ), 182
denial-of-service attack, 186
departments
as domain, 111
cross-domain, 103
orientation, 102
SOA and, 101
dependencies
between services, 29
events and, 135
Magic Bus and, 6
from Web Services, 224
deployment
different owners and, 254
registries and, 234
system updates, 45
WSDL lifecycles, 224
deprecated, 143
DES, 183
deserializing, 160
design
repositories and registries, 233
solutions, 85, 103
developer-driven SOA, 273
development
iterative and incremental, 87
versus maintenance, 137
development teams, 54
different owners, 2, 13
service implementation and, 171
service modification, 140
digital signatures, 175
discoverability, of services, 31, 140
distributed systems
abstraction layers, 175
analysis paralysis, 39
attacks on, 185
backends, 176
BOM (business object model), 39
communications
mapping layers, 55
protocol-driven approach, 55
confidentiality, 176
data transference security, 177
drivers, 13
forms, 36
frontends, 176
heterogeneous security concepts, 175
identities (profiles), 177
layer problems, 176
objects versus SOA, 23
processes, defining, 9

security requirements, 173
SOA drivers, 13
SLO (single logouts), 184
SSL (Secure Sockets Layer), 177
system withdrawal, correct procedure for, 143
users
authentication, 176
consumer cooperation, 143
identity context, 175
multiclient capabilities, 177
versioning, 146
distributed testing, 201
distributed, objects, 23
distribution
organizations and, 102
security and, 175
DMZ (demilitarized zone), 182
document/literal wrapped, 214
documentation
generation of, 238
tasks, necessary steps, 270
transparency, 265
domain/IT alignment, 110
domain-driven
architecture, 111
versioning, 146
domains, 109, 295
harmonization, 111
heterogeneity, 111
hierarchies, 111
subdomains, 111
domain-specific
data types, 205
language, 241
domain-specific language (DSL), 296
DoS (denial of service), 174
double confirmation, error handling, 130
drivers
distributed systems, 13
domains, ownership of, 13
heterogeneity, 14
and politics, 14
SOA and, 13
DSL (domain-specific language), 241
durable, 64

E

EAI (enterprise application integration), 175, 296
bus, 6
EDA (event-driven architecture), 134, 296
employee management, and using, 276
encapsulation, 63
encryption, 175
endpoints
definition, 50
mediators and service calls, 41
processing at runtime, 52
WSDL and, 212
end-to-end
encryption, 179
security, 177
enforcement points, 118
engines
business processes, 89, 111
rules engine, 111
enterprise application integration (EAI), 6, 175
enterprise business performance, 102
enterprise service bus (see ESB, enterprise service bus)
enterprise service bus, ESB (see ESB, enterprise service bus)
enumerations, 206
EPC, 296
error handling
consistency and, 69
data types, 207
error IDs, 207
exception type, 207
frontends, 115
key/value list, 207
MEP (message exchange patterns) and, 129
routing and, 207
services, 207
solution management and, 104
ESB (enterprise service bus), 296
advanced, 113
API driven, 222
API responsibility and, 54
BPEL and, 48, 58
caching, 120
differences between, 50
API-driven approach, 54
indirect approach configuration, 51
interceptors (proxies), 52
point-to-point connections, 50
protocol-driven approach, 53
gateways, 50
generic ESB, 42
heterogeneous, 49, 161
hub-and-spoke servers, 222
implementation, 48, 54

ESB *(continued)*
inconsistencies, 255
infrastructure teams, 54
code generation, 55
development teams and, 54
mapping between business and protocol APIs, 56
tasks of, 54
interoperability, 9, 48, 55, 174
JMS protocol, 222
libraries, 55
location, defining, 222
Magic Bus and, 6
mediator role, 51
mixed architecture models, 111
dynamic modifications, 51
protocol-driven, 54
reliability, 132
repositories, 233
responsibilities, 47–49
routing, 48
security and, 179
stateful services, 194
type checking, 42
value-added services, 56
BAM (business activity monitoring), 58
CBR (content-based routing), 56
data mapping, 54, 56
implementation, 58
logging, 57
management, 57
monitoring, 57
reliability issues and, 57
routing, 56
security, 57
versioning and, 155
WSDL (Web Services Description Language), 226
XML schema files, 42
establishment
architectural roadmap, 271
bottom-up approach, 272
different approaches, 273
drivers, four typical, 273
four Ps, 271
MDSD processes, 247
overall tasks, 269
repositories, 235
SOA, 261
top-down approach, 272
event, 296
event manager, 135
event-driven architecture (EDA), 296
event-driven process chain (EPC), 296
events
architecture and, 134
publish/subscribe and, 129
exception in meta models, 255
expansion stage, first fundamental, 65
Extensibility Points, 221
extensions, 206
external reviews, 270
external services, 74

F

failover
load balancing and, 51
process services and, 71
statefullness and, 196
Web Services and, 53
fault messages
generic, 207
MEP and, 130
fault tolerance, 17, 35
FDTs (fundamental data types), 204
federated SOA, 69, 73
federation, 181
fiefdoms, 102
fire and forget, 125, 297
firewalls, 175
five slides approach, 8, 270
flexibility, 11, 17, 35
flooding, 174, 185
formats, governance and, 263
foster parents, 266
frontends, 297
backend consistency, 66
composed services, 67
multiapplication, 119
multichannel support, 119
multichannels, 119
multiple applications, 119
prevalidation, 117
responsibility of, 114
role of, 114
SOA and, 114–117
stateful services, 195
user input, 115
validation, 118
multilayer, 117
prevalidation, 118
workflow, 117
fundamental data types (FDTs), 204
fundamental SOA (service-oriented architecture), 65
funding models
implementation, 106
organizations and, 106

SOA and, 263
withdrawing services and, 143
future management, 120

G

generating service code, 237
generation, 242
generic
code, 239
data types, 154
interfaces, 42
services, 166
GetCustomerData( ), 147
governance, 19, 297
basics of, 263
central SOA teams, 265
decentralization goals, 10
defined, 262
definition, 262
employees, selection of, 10
enterprise-wide processes, 264
executive support of SOA, 10
nontechnical issues, 263
nontechnical tasks, 263
SOA and, 19, 261
technical tasks, 265
granularity
basic services and, 63
code generation, 253
events and, 134
performance and, 164
of services, 30
services and, 166
graphical user interfaces (see user interfaces)
grass roots, 273
guidelines, SOA and, 274

H

harmonization
versus centralization, 114
data types and, 38, 148
domains and, 111
versus heterogeneity, 14
large distributed systems, 3
tools and, 113
header data (see data types)
heterogeneity, 3, 14
data types, 38
domains, 111
ESBs and, 49
repositories, 235
security and, 175
tools, 113
heterogeneous platforms, 17
high interoperability
securty versus, 175
SOA concepts, 16
high-level design, 85, 103
history of SOA
Booch, Grady, 8
critics of SOA, 8
Natis, Yefim V., 7
Pasik, Alexander, 7
Schute, Roy, 7
server orientation, 7
strategy of, 8
terminology, 7
Web Services, 7
history, of service brokers, 231
history, of Web Services, 210
hockey stick function, 2, 267
homework, 227, 282
HTTP, 297
SOAP and, 216
Web Services standards, 211
WS-I Basic Profile, 221
HTTP RESTful, 228
hub-and-spoke, 297
connections, 222
integration, 58
JMS protocol, 222
human interaction, 71, 114
hybrid services, 76
hype of SOA, 262

I

IBM model, 277
ICC (Integration Competency Center), 265
IDE, 297
BPEL and, 251
BPM and, 86
models and, 242
idempotency, 297
IDS and, 200
service attribute, 198
idempotent
MEPs and, 130
services and, 32
identification
of services, 85, 139
WSDL lifecycles, 224
identification federation, 181
Identity Federation Framework (ID-FF), 187
identity management, 187
identity provider (IdP), 175, 181
Identity Web Services Framework
(ID-WSF), 187
ID-FF (Identity Federation Framework), 187

IdP (identity provider), 181
ID-WSF (Identity Web Services Framework), 187
imperfectionism, 4
implementation
caching, 167
funding models, 106
meta models, 246
pilot (step 2), 267
process automation, 268
projects (step 3), 268
services, 58, 137
SOA, 261
strategy (step 4), 269
tasks, using all 4 steps, 270
teams and, 171
understanding (step 1), 266
inconsistencies, infrastructures and, 255
incremental development, 87, 94
infrastructure
binding, 224
consumers and, 222
ESB and, 47
inconsistencies, 255
insufficient security, 178
interoperability, 18
layers and independent, 227
mapping, 224
mediator role, 51
protocols and, 224
reliability, 133
repositories, 233
SOA
presentation services, 117
solutions, 63
SOA and, 18
solutions, 227
teams, 54, 227
versioning, 156
Web Services, 227
WSDL and, 226
(see also ESB, enterprise service bus)
infrastructure services, 77
infrastructure teams, 54, 133
integration, 102, 139
Integration Competency Center (ICC), 265
integration services, 76
integration tests, 202
Integrity, 174
interaction patterns, 43
interactive SOA, 134
interceptors, 52, 223
stub, 238
Web Services and, 224
interfaces
business-driven, 27
EBS (enterprise service bus), 42
generic functionality, 156
granularity (operations), 27
meta models, 246
modification, 138
parameters, 27
signatures, 27
technically driven, 27
three strikes pattern, 269
versioning policy, 156
WSDL and, 212
(see also services; versioning policy)
intermediate services, 67
internal reviews, 270
internal services, 74
international services, 74
internationalization, 74
interoperability, 297
ESB and, 175
natural firewalls, 175
high and SOA, 33
interfaces, 221
Liberty Alliance Project (LAP), 187
protocol aspects, 221
protocol modification, 55
protocol-driven ESBs, 221
security, 70
security versus, 175
SOA and, 17
SOA concepts, 16
interrupting business processes, 117
introducing SOA, 261, 282
inventions of SOA, 281
isolated, 64
IT-driven SOA, 273
iterative
business process performance, 262
development, 87, 138
incremental approach, 273

J

JaBoWS (Just another Bunch of Web Services), 227
Java Naming and Directory Interface (JNDI), 219
JMS, 127, 222, 297
JNDI (Java Naming and Directory Interface), 219
juristic data, 66, 72, 196

K

Kerberos, 174, 183
key success factors of SOA, 282

L

labeling, 250
landscape, 47
large distributed systems, 3
  bottlenecks, 4
  design of, 170
  heterogeneous systems, 3
  imperfections of, 4
  landscape maintenance, 3
  redundancy, 4
  tale of the Magic Bus, 4–6
  types of owners, 4
layers
  abstraction and distributed processes, 175
  binding, 212
  business layer, 179
  description of primary, 212
  ESB, 179
  infrastructure-independent, and Web Services standards, 227
  mapping, 55
  MEP (message exchange patterns), 132–133
  MEPs and, 131
  messages, and constraints, 58, 179
  multiple users, 179
  problems with, 176
  process, 73
  security and, 175
  SOA and, 62
  special, 73
  SSL (Secure Sockets Layer), 177
  transport, 130
  transport-layer
    browsers, 55
    implementation tools, 55
    protocol HTTPs, 55
    security, 179
  validation, multilayers, 117
  workflow layers, 82
  WSDL and, 212
LDAP, 219
leadership, 273
legacy, 3
Liberty Alliance Project (LAP), 181, 187
lifecycles
  establishing, 269
  governance and, 263
  services and, 137
  WSDL and, 223
lifetime of data, 3
limit of SOA, 283
limits
  collaboration, 171
  orchestration and, 168
  signatures, 206
lists
  data types, 206
  meta models, 244
load balancing
  endpoints and, 52
  failover, 51
  statefulness and, 196
local services, 74
logging, 57
logic services, 64
logical architecture models, 109
loose coupling, 16, 30, 297
  2PC (two-phase commit), 44
  asynchronous communication
    advantages of using, 38
    choosing whether to use, 128
    correlation ID, 38
    drawbacks of using, 38
    race conditions, debugging, 38
    reply problems, 37
  binding, 43
  BOM (business object model), 38
  common object models, disabling effect of, 56
  compensation, 44
  complexity and, 282
  complexity of, 36
  data types
    advantages of using, 40
    disadvantages of using, 40
    heterogeneous, 38–41
    mappings, 40
    multiple address types, 41
    type checking, 42
    versioning, 45
  deployment, 45, 255
  ESB (enterprise service bus), 42
  fault tolerance, 35
  flexibility, 36
  forms, 36, 46
  mediators, 41
  point-to-point approach, 41
  process logic, 45
  repositories, 232
  scalability, 36

loose coupling *(continued)*
SOA and, 17
SOA establishment, 267
SOA versus EDA, 135
system dependencies, reducing, 9
transaction safety, 44
versioning policy, 45

M

macro flow, 71
Magic Bus, 4
maintenance versus development, 137
management
of services, 231, 265
of solutions, 103
management support, 103, 106, 266, 273, 282
management-driven SOA, 273
manual modification of generated code, 251
mapping
data types, 40, 56
external to internal, 155
between protocol and API, 54
protocols, 161
services interfaces, 68
marketplace
internal coordination, 234
UDDI, 219
master data, 67, 119
master data management (MDM), 56
maturity models, 297
maturity models, SOA and, 276
MDM (master data management), 56
MDSD (model-driven service development), 19, 298
approaches for, 238
code
debugging, 251
generation, 237
granularity problems, 253
manually modified, 252
uses for, 238
DSL (domain-specific language), 241
labeling, 250
languages, 241
manual modifications, 252
model, defined, 240
modifications, 254
PIMS (platform-independent models), 242
platform-independent models, 242
platform-specific models, 242
PSMs (platform-specific models), 242
software development, 239
terminology, 239
three strikes pattern, 269
tools, 255
configuration management rules, 243
design implementation, 244
model-driven tools, 242
oAW (openArchitectureWare), 258
Rational Rose Repository, limitations of, 244
XML notation, 246
XSLT, 255–257
transformation
code generation, 247
policy, 250
provider-driven, 249
third-party-driven, 250
mediation, 41, 50
mega project, 273
MEP (message exchange patterns), 298
architecture
EDA (event-driven architecture), 135
basic, 124
categories
MOM (message-oriented middleware), 127
one-way, 125–127
pros and cons, 128
publish/subscribe, 129
request/callback, 124–128
consumer process, 126
defined, 29, 123
EDA (event-driven architecture), 134
errors, 129
fault messages, 130
message, molder, 131
technical, 130
layers, 131–133
notifications (events), 129
publish subscribe, 129
request/callback, 128
request/response, 124
return addresses, 128
merging generated code, 253
message, 298
message exchange patterns (see MEP, message exchange patterns)
message header
routing of, 57
security and, 179
message-layer security, 177, 179
message-oriented middleware (see MOM, message-oriented middleware)
messages, 26
faults, 130
IDs, 202
meta model, 240, 298
metrics, governance and, 263
micro flow, 67
mixed architecture models, 111

model, 240, 298
model transformation
consumer-driven, 247
(see also MDSD, transformation)
model-driven service development (MDSD), 237
model-driven software/service development, 298
modeling language, 241
modeling services, 81, 239
models
choice of, 113
cross-department domains, 111
domain, 111
logical, 110
mixed, 111
technical models, 112
modification
versus bug fixes, 147
of services, 140
MOM (message-oriented middleware), 298
defined, 127
MEP (message exchange patterns) and, 127
message exchange, 124
one-way messages, 127
request/response, 127
service calls, 124
monitoring, 143
BAM and, 58
ESB and, 57
governance and, 265
monolithic systems, 101
MQ, 127
multiapplication support, 119
multichannel support, 119, 195
stateful services, 197
multiclient capabilities, 177
multiple requests, 198
multiproject management, 268

N

name server, 52
namespaces
higher data types, 206
WSDL, 214
WSDL 2.0 and SOAP 1.2, 216
national services, 74
network (see infrastructure; ESB)
networked SOA (see federated SOA)
new aspects of SOA, 281
nightly builds, 250
NIL, 205
nillable, 205
nonblocking request/response, 136
nonfunctional attributes
backward compatibility, 148
BPEL and, 172
REST and, 228
nontechnical governance tasks, 263
not invented here syndrome, 104
notation
BPEL and, 89
services and, 246
notifications
event-driven architecture, 134
publish/subscribe and, 129
null semantics (see data types, reference semantics)

O

OASIS, 298
Oasis SOA Reference Model
definition, 20, 31
distributed systems, 13
granularity, 30
loose coupling, 30
oAW (openArchitectureWare), 258
objectives, 263
object-oriented programming (see OOP, object-oriented programming)
objects
distributed, 23
models, disabling effect of, 56
OBV (Objects by Value), 23
versus SOA, 23
(see also SOAP, Simple Object Access Protocol)
observer pattern, 129
OBV (Objects by Value), 23
official process, 264
offline ability, of repositories, 234
OMG, 298
once and only once
messages, 201
reliability, 57
one-way, 298
fire and forget, 125
MEP (message exchange patterns), 125
request/response, 125
OOP (object-oriented programming), SOA and, 283
open standardization, 220
openArchitectureWare (see oAW, openArchitectureWare)
operating system interdependence, 16
operations, in meta model, 245

orchestrated services, 67
orchestration, 298
choreography
and SOA, 97
process chains, 97
versus, 96–97
customized service, 168
design process drawbacks, 96
EDA versus, 134
limits of, 167
REST and, 228
of services, 67, 88
order management, 120
order of messages, 123, 128
organizational structures
backends, 65
profit centers and, 105
organizations
collaboration, 104
distribution, 102
governance, 263
incident management procedure, 104
management support, 106
model funding, 106
monolithic systems, 102
SOA and, 101
solutions for, 103–105, 111
ownership
different owners, 2
domains, 13
service modification and, 140

P

paradigm, 11, 21
parameters, in meta model, 245
participant, 24, 298
passwords, 174
payload, 203
PDP (policy decision point)
policy enforcement point versus, 118
security, 180
validation, 118
PEP (policy enforcement point), 180
perceived process, 264
perfectionism
expense of, 87
large systems and, 4
versioning and, 145
performance, 159
security and, 181
service quality and, 164
physical location of a service, 212
pilots
copy-and-paste, 269
first, 267
infrastructure, establishing, 268
project approaches, 274
second and third, 268
PIMS (platform-independent models), 242
platform dependencies
fundamental data types, 44
independence of, 43
interaction patterns, 43
platform independence, 16, 63
platform-independent models (PIMS), 242
Plug and Play, 88
point-to-point
connections, 50, 221
encryption, 179
security, 177
Web Services and, 221
policies
decision points, 118
enforcement points, 118
governance and, 263
law versus guideline, 274
three strikes pattern, 269
policy, 298
policy decision point (see PDP, policy decision point)
policy enforcement point (PEP), 118, 180
policy enforcement point (see PEP, policy enforcement point
port type, 212
portal, call center and, 119
portfolio management
reusability and, 169
services, 94
(see also services, lifecycles)
post-conditions, 33
practice versus theory, ix
preconditions, 33
presentation of backend data, 114
presentation services, 117
prevalidation, user input, 117
process, 299
process logic control, 45
process services, 299
business process chain versus, 135
domains, 111
human interaction, 114
orchestration platform, 76
third stage expansion and SOA, 71
process steps, 82
process-enabled SOA (service-oriented architecture), 71
processes
governance, 263
governance and, 263
large systems and, 19

MDSD, 247
three strikes pattern, 269
processing, of service calls, 160
profiles, 299
Basic profiles, 221
standards and, 221
of users, 181
profit centers, 105
programming language independence, 16
programming paradigm, 43
project orientation, 102
protected areas, 252
protocol-driven ESB
versus API-driven ESB, 53
Web Services and, 222
protocols
APIs, 56
APIs versus, 131
aspects, 221
code layers, 55
defined, 54
driven approach, 55, 221
ESB and, 53
HTTPs, 55
infrastructure and, 224
intermediate, 160
interoperability, 221
mapping, 161
mapping to API, 54
point-to-point, 221
process engines, 48
reliability, 132
SOA (service-oriented architecture), 160
Web Services, 228
security, 178
WSDL, 224
XML-based, 160
(see also SOAP, Simple Object Access Protocol)
provider-driven transformations, 247
providers, 24, 299
brokers and, 231
caching, 167
consumer role of a system versus, 125
Web Services and, 218
proxies (see interceptors)
public services, 74
publish/subscribe, 129, 299

Q

QoS (Quality of Service), 28, 33
quality assurance, 140

R

race conditions, debugging, 38
Rational Rose Repository, 238, 244
reading services, 75, 110
receive activity, 90
records, 206
red zone, 66
redundancy
input validation, 116, 118
Magic Bus and, 4
multiple backends and, 66
reference architecture
copy-and-paste, 269
fundamental decisions for, 263
reference semantics, signatures and, 205
registration services, 76
registries, 233
designing, 233
establishment of, 235
services and, 233
Web Services and, 219
registry, 299
reliability, 131
at least once, 57
error handling, 131
once and only once, 57
remote procedure call (RPC), 124
remote stored procedures, 162
repositories, 233, 299
architecture models, 113
designing, 233
establishment of, 235
meta models and, 247
services and, 233
versioning, 155
WSDL and, 224
request, 299
request/callback, 128, 299
request/response, 300
consumers and, 124
one-way, 125
requestor, 24, 299
requests
IDs and, 200
multiple, 198
response, 300
responsibility
defining roles of, 263
between infrastructure and domain teams, 117
services and, 111
REST (Representational State Transfer), 228
RESTful HTTP, 228
restrictions, 206

retries, 132, 198
retrospectives, 270
reusability
consumer-specific services versus, 168
performance and, 164
of services, 32, 77
reviews, 270
revision, 146
revolutionary aspects of SOA, 281
risks
reviews and, 270
services and, 77
roadmap for SOA, 261
robustness, 254
roles
defining, 263
of services, 76
round-trip processing and MDSD, 242
routing
content-based, 56
error handling, 207
intelligent, 48, 56
internationalization and, 74
messages, 56
response messages, 127
stateful services, 194
sticky, 196
value-added services and, 56
RPC (remote procedure call), 124
RSA, 183
rule services, 110, 113, 115
rules (see business rules)
rules engine, 113, 118
decision points, 118
mixed architecture models, 111
runtime, 159

S

safe extension, 225
SAML (Security Assertion Markup Language)
assertions, managed, 183
attributes, 183
distributed logouts, 184
security data, management and exchange, 183
SSO (single sign-on), 184
subjects, 183
scalability, 17, 36, 113
statefulness and, 196
SCC (SOA Competency Center), 265
schematic code, 239
SCoE (SOA Center of Excellence), 265
Secure Sockets Layer (SSL), 177
security
abstraction layers, 176
accounting, 174
ADF (access control decision function), 180
authentication, 173, 180
authorization, 174, 180
availability, 174
confidentiality, 174, 179
data integrity, 179
digital signatures, 175
DMZ (demilitarized zone), 182
encryption, 175
end-to-end, 177
establishment, 188
heterogeneity, 175
IdP (identity provider), 181
implementation of, 188
infrastructures, and insufficient, 178
Integrity, 174
interoperability, 70
interoperability versus, 175
lower default, 188
message-layer, 177, 179
natural firewalls, 175
passwords, 174
PDP (policy decision point), 180
PEP (policy enforcement point), 180
performance and state, 181
point-to-point, 177
reasons for neglected, 178
requirements, 188
REST (Representational State Transfer), 228
services
decision, 180
enforcement, 180
SOA, 178
SOA establishment, 267
SOA overview, 173
SOAP attachments, attacks on, 188
SSL (Secure Sockets Layer), 177
standards
assertions, managed, 183
general, 183
SAML (Security Assertion Markup Language), 183
Web Services, 183
WS-Federation, 185
WS-SecureConversation, 184
WS-Security, 183, 184
WS-SecurityPolicy, 184
WS-Trust, 184
XML DSig (XML Signature), 184
XML Enc (XML Encryption), 184

XML Key Management, 184
XML-based, 183
statelessness and, 181
traffic, auditing, 174, 182
transport-layer, 177, 179
user IDs, 174
Web Services, 182, 221
Web Services, attacks on, 185
XML, 182
attacks on, 185
bombs, 185
XPath injections, 186
self-contained, 29
serialization, 160
server, 7, 24
service agent, 24
Service Integration Maturity Model (SIMM), 277
service-oriented teams, 266
services, 16, 24, 300
ACID properties, 64
activation, 76
additional attributes, 29
agent, 24
applications, 76
as adapters, 68
backends, 65, 66
basic, 62
data, 63
functionality, 63
logic, 64
bottom-up approach, 94
BPM, 84
brokers, 219
central services interface, 219
history of, 231
purposes for, 231
business, 76
business event tracing, 94
categories, 74
commodity, 77
compensation context code, 192
composed, 67
consumer-specific, 28, 167
contracts
definition, 28
interfaces, 28
nonfunctional attributes, 33
pre- and post-conditions, 33
resources, required, 28
control of, 114
controller, 76
coordinator, 76
correlation IDs, 202
drawbacks, 203
one business process, 202
one service call, 202
customized, 167
data access, 62
slow data access, 30
data types, 29
data-centric, 76
defined, 25, 26–34, 61
development, 137
distributed system landscapes, 78
distributed testing, 201
domains
cross-domains, 111
decomposition, 94
expansion stages, 62
external, 74
external interfaces, 30
functionality, 8, 26
generic, 42, 166
goals, 26
granularity, 30, 253
hybrid, 76
idempotent, 32, 198
identification, 85, 139
identification of, 94
implementation, 58, 137
infrastructure, 77, 78
integration, 76
interaction patterns, 43
intermediate, 67
internal, 74
international, 74
layers, 62
lifecycles, 19
design phase, 138–139
discovery, 140
identification, 139
iterative development, 138–139
portfolio management
local, 74
logic service (rule service), 112
logic services, 64
maintenance, 140
management, 231
MDSD, 250
message exchange patterns, 29
meta models, 246
models, 86
modification, 87, 138, 140
modifications, 147
national, 74
number of service versions, 155
orchestrated, 67

services *(continued)*
portfolio management, 94
presentation services, 117
remote stored procedures, 162
process
decomposition, 94
enabled SOA and, 72
examples, 71
failover mechanisms, 71
juristic data, 72
persistent data, 72
processing, 160
production, 140
bug fixes, 141
modification of, 141
run phase, 141
dead services, 142
introduction of new services, 140
software, 140
public, 74
public-enterprise, 74
QoS (Quality of Service), 28
reading, 75
registration, 76
registries, 233–236
remote stored procedures versus, 162
repositories, 233–236
responsibility for, 111
reusability, 168
reusability of, 32
roles of, 76
searches for, 31
security, 180
service state versus backend state, 72
signatures, 28, 43
SLA (service-level agreement), 28
fundamental SOA, 65
software, removing existing, 142
stateful, 193
backends, 195
definition, 194
drawbacks, 196
failover mechanisms, 196
frontends, 195
options for, 195
shopping cart application, 194
stateless, 192
advantages, 196
backends, 195
bulk services, 197
defined, 32
definition, 192
modification of, 192
performance, 197
process services, multichannel support, 197
security, 197
task-centric, 76
technical, 32, 77
territory, 77
time service factor, 161
topology of interfaces, 26
unit tests, 201
value-added, 77
versioning (see versioning policy)
views, 165
withdrawal of, 142
withdrawing, 142
wrapper, 76
writing, 75
sessions, 71
shopping cart application, 194, 197
signature
reference semantics and, 205
services and, 28
signatures
input parameters, 28
silos (see monolithic systems)
SIMM (Service Integration Maturity Model), 277
simplicity, 206
single logouts (SLO), 183
single sign-on (SSO), 184
skeletons
API and, 238
manual modifications, 252
SLA (service-level agreement), 28, 300
and quality of service, 33
backward compatibility, 148
meta models, 245
performance and, 161
time service factor, 161
WSDL (Web Services Description Language), 226
SLO (single logouts), 183
SOA (service-oriented architecture), 300
and Web Services, 7, 33
attachments, attacks on, 188
attributes, 166
bandwidth issues, 161
business strategies, 266
call constraints, 166

CIT (telephony integration), 170
concepts of, 16
customized services
loading data, 167
orchestrating limits, 168
defined, 8, 12
definition, 11
deserialization, 161
distributed systems, 170
infrastructure, generic, 156
multiclient capabilities, 177
versus OOP, 283
OOP (object-oriented programming), 283
performance, 159
backward compatibility, 169
recommendations, 164
reusability, 168
solutions, 170
critical steps, 160
remote stored procedures, 162–163
intermediate protocol, 160
reusability, 164
skipContractData flag, 166
SLA (service-level agreement), 161
specialOptimizationForSystemX flag, 166
time service factor, 161
Web Services and, 21, 209, 222
when not to use, 283
XML-based protocol, 160
SOA 2.0, 134
SOA Center of Excellence (SCoE), 265
SOA Competency Center (SCC), 265
SOAP (Simple Object Access Protocol), 300
attachments, 188
binding, 214
data exchange, 211
defined, 217
document/literal wrapped, 214
header data, 217
HTTP headings, 218
messages, 217
modifications and, 218
request code example, 217
styles, 214
versions, 217
Web Services and, 210
software architecture, 300
Software Capability Maturity Model (SW-CMM), 276
software, development versus maintenance, 137
solution management, 103
solution manager, 103
solutions
design, 85
first step of SOA project, 268
specialOptimizationForSystemX flag, 166
SSL (Secure Sockets Layer), 177, 183, 301
SSO (single sign-on), 184
stability
of first pilot, 268
of services, 146
standards
basic, 211
Basic Profile, 221
for BPM, 91
defined, 210, 220
governance and, 263
HTTP, 211
interoperability, 220
open standardization, 220
profiles, 221
purpose for, 228
security, 184
SOA and, 82
SOAP (Simple Object Access Protocol), 217
UDDI (Universal Description, Discovery, and Integration), 218
Web Services and, 211
WS-1 (Web Services Interoperability Organization), 221
WSDL (Web Services Description Language), 211
XML, 211
state, 191
security and, 181
services versus backend, 71
stateful services
versus backend state, 71
technical details, 191
stateless services, 191
statelessness, 32
step, 82
sticky routing, 196
strings, 204
formats, 206
reference semantics and, 205
structures
data types, 206
meta models, 247
modifications, 247
stub, 238
subdomains, 111

subjects, 183
subprocess, 82
success factors, of SOA, 282
suites, SOA, 113
supply chain, 135
SW-CMM (Software Capability Maturity Model), 276
systems
  architecture, 19
  boundaries, 110
  consumer versus provider role, 125
  meta models, 243
  monolithic, 101
  (see also backends)

T

task, 82, 301
task-centric business, 76
teams
  central SOA teams, 265
  as domain, 111
  infrastructure versus development, 54, 117
  structures of, 101
technical architecture models, 112
technical data, 203
technical details, 191
technical governance tasks, 265
technical services, 32, 77
terminology
  coarse-grained concepts, 20
  consumer, 24
  loose coupling concepts, 20
  MDSD, 239
  participant, 24
  provider, 24
  requestor, 24
  SOA (Service Oriented Architecture), 20
  visibility, 20
  Web Services, 20
territory services, 77
test mode, 202
testing, 140
  code generation, 238
  complexity and, 282
  debugging and, 201
theory versus practice, ix
third-party-driven transformations, 247
three strikes pattern, 239, 268
time service factor, 161
  meta model, 251
time service factor (TSF), 161
tokens, 174
tool vendors
  architecture models, 113
  repositories, 236
  Web Services, 220
tools
  busines process modeling, 87
  MDSD and, 242, 255
traffic, auditing, 182
transaction context, 70
transaction safety, 44, 70
transformation, 247
transparency, 265
transport-layer
  reliability and, 130
  security, 177, 179
two-phase commit
  (see 2PC, two-phase commit)
type aliases, 206
type checking, 42
typedefs, 206
types (see data types)

U

UBR (UDDI Business Registry), 218, 232, 301
UDDI (Universal Description, Discovery, and Integration), 301
  central service brokers, 219
  defined, 218
  Web Services, managing, 211
  WS-I Basic Profile, 221
UML (Unified Modeling Language), 92, 240
understanding SOA, 266, 282
unit tests, 201
usage factor, 169
user interfaces, 114
users
  IDs, 174
  input validation, 116
  interfaces, 116
  prevalidation of input, 116
  profiles, 181
  repository interfaces, 234

V

validation
  backends, 117
  decision points, 118
  frontends, 118
  prevalidation, 118
  redundancy, 118
  SOA and, 117
  user input, 115, 117

value ranges, 206
value-added ESB services, 56, 77, 223
vendor diversity, 33
versioning policy, 45
call constraints, 156, 166
configuration-management-driven, 155
artifacts, 155
tools, 155
consumers, and changes, 147
data types, 149–155
data types and, 148
data types, fundamental, 204
domain-driven, 146
backward-compatible modifications, 148
consequences of, 147
nontrivial policy, 149
trivial policy, 147
generic functionality, 156
labeling, 250
migrations, 146
effects of modifications, 156
namespaces, 149
number of service versions, 148, 155
repositories and, 234
requirements, 145
multiple revisions, 146
running multiple versions, 146
runtime, 148
new service introduction, 147
service name conventions, 147
service withdrawal, 148
UDDI registry, 149
too many versions, avoid, 148
versions, versus revisions, 146
view concept, 165
visibility of services, 31
visions, 263

W

W3C (World Wide Web Consortium), 301
warnings, 207
waterfall approach, 138
Web Services, 301
activity sequences, 90
alternatives, 227
application problems, 226
assigning, 90
attacks on, 185
BPEL, 90
central service brokers, 219
fault messages, 130
flow, 90
history of, 7, 210
infrastructure teams, 227
inheritance, avoid, 206
invoke, 90
nillable, 205
operations, 90
priorities for using, 227
problems with, 9
process engine, 227
proprietary SOA infrastructure solutions, 227
protocols, 221
point-to-point, 52
security and, 178
receive activity, 90
registries, 219
reply, 90
restrictions (complex value constraints), avoid, 206
revolution, 281
roles, 228
security, 179, 182, 184
service interface format, 227
SLA attributes, 227
SOA, 22
SOA (service-oriented architecture), 222
SOA and, 7, 21, 33, 209
standardization, 220
standards for, 210, 228
variables section, 90
XSD types, 206
(see also protocols)
Web Services Interoperability Organization (see WS-I, Web Services Interoperability Organization)
web site for the book, xiii
well-defined, 28
WfMC (Workflow Management Coalition), 91
Wf-XML, 92
withdrawing of services, 142
workflow, 301
(see BPM, business process management)
frontend versus backend, 117
macro flow, 71
micro flow, 67
WPDL (Workflow Process Definition Language), 91
wrapper services, 76
writing services, 75, 110
idempotent, 199
WS (Web Ser vices), 301
WS-BPEL, 89
WS-CDL (Choreography Definition Language), 92

WSDL (Web Services Description Language), 301
attributes
code generation, 226
nonfunctional, 224
service-level agreements and, 224
binding, 224
code example, 212–216
consumer's perspective, 223
deficits, solutions for, 224
extensions, 225
files
defined, 224
deployment details, 224
lifecycle of, 223
various stages, 224
interceptors, 223
layers of description, 212
lifecycles, 223
limits of, 224
modeling language, 241
protocol, 224
repositories, 224
safe extension, 225
service
availability, 224
identification, 223
service interface definitions, 211
SOA landscape integration, 223
structure of, 211
teams, 224
WS-I Basic Profile, 221
WSDL lifecycles, 211
WSFL (Web Services Flow Language), 89
WS-I (Web Services Interoperability Organization), 301
WS-Security, 179, 183
security techniques, 184
SOAP and, 184
Web Services encryption, 179
WS-I (Web Services Interoperability Organization), 221
Basic Security Profile, 185, 221
Conformance Requirements, 221
Extensibility Points, 221
profiles, 221
WS-Federation
trust brokering, 185
WS-SecureConversation
shared security context, 184
requests, reducing time-consuming, 184
WS-SecurityPolicy
Kerberos, 184
X.509, 184
WS-Trust
security token validation, 184

X

X.509, 184
XLANG, 89
XML, 211, 301
and SAML, 183
models and, 240
Open Standards, 178
overview, 183
performance and, 160
security, 182
data management and exchange, 183
documents, 186
standards, 183
Web Services, 184
XML bombs, 185
XML DSig (XML Signature), 184
XML Enc (XML Encryption), 184
XML Key Management, 184
XML language, 89
XPath, 186
XSD (XML Schema Definition), 301
XPath injections, 186
XPDL (XML Process Definition Language), 91
XSD, 211
XSLT (eXtensible Stylesheet Language Transformations)
bottlenecks, avoiding, 258
code generation, 242
oAW (openArchitectureWare), 258
problems with, 257
processor, defined, 255
stylesheet files, 255
templates, 255

Y

YAGNI (You Ain't Gonna Need It), 95, 169, 267
yellow pages
friends and, 232
Web Services, 219

关于作者

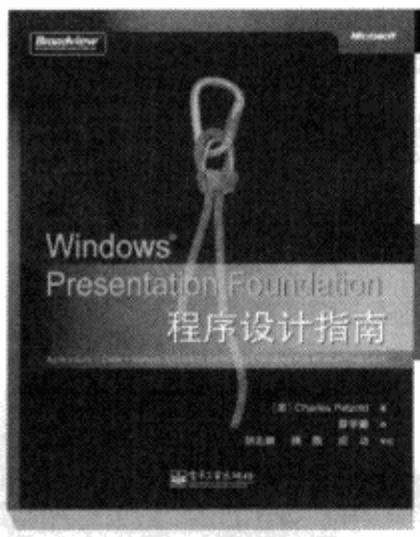
Nicolai M. Josuttis 是一位经验丰富的系统架构师，同时也是一位成功负责实现了好几个大型分布式系统的 IT 经理人。最近三年，他把时间花在了为一家行业领先的国际电话公司全面推广 SOA 上。对 IT 业界来说，他是一位知名的、经验丰富的演讲者，同时也是《C++模板库》和《C++模板》两书（均由 Addison-Wesley 出版）的作者。他总是对以符合实际、恰当的方式应用技术和原则充满兴趣。

封面介绍

封面插图出自 Mike Kobke 之手，他在本书的照片上加上了一些艺术图。

关于作者

Nicolai M. Josuttis 大型分布式系统的 IT 咨询人，最近三年，他专门研究在了为一家行业领先的国际电信公司全面推广 SOA 上。对 IT 业界来说，他是一位知名的、经验丰富的演讲者，同时也是《C++模板库》和《C++模板》两书（均由 Addison-Wesley 出版）的作者。他总是对以符合实际、恰当的方式应用技术和原则充满兴趣。



《Windows Presentation Foundation程序设计指南》

[美] Charles Petzold 著
蔡学镛 译
胡志鹏 魏颢 成功 审校

- Windows大师Charles Petzold倾力打造
- 助您迈进.NET 3.0开发殿堂
- 用简洁的代码描述复杂的问题

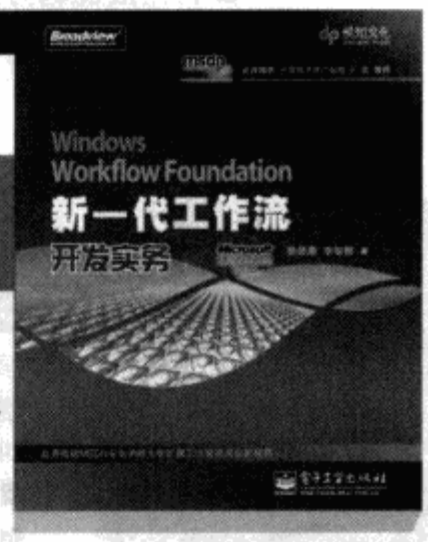
如果你想了解隐藏在WPF华丽外表之下的奇妙世界，如果你有“千淘万漉虽辛苦，吹尽狂沙始到金”的那份执着和耐心，那么就请泡上一杯茶，打开你的计算机，准备好WPF的编程环境，开始细细品味这本充满乐趣的书吧。

——胡志鹏

《Windows Workflow Foundation新一代工作流开发实务》

彭靖灏 李智桦 著

- 深入探讨了Workflow Foundation与Web Service及ASP.NET应用程序的结合
- 充分展现Windows Workflow Foundation与外界互动的惊人实力。



我相信，以两位作者平常在研讨会上说故事的能力，一定可让读者在看完这本书之后，将这个软件开发必备的技能发挥得淋漓尽致。

——王 森

《学习WCF：从原理到实践》

[美] Michele Leroux Bustamante 著
庞引明 侯伯薇 译

- 囊括100多个代码示例、20多个操作实例
- 助您加深WCF各个重要方面的相关知识

当你完成的时候，你将成为WCF专业开发者群体中的一员，这些人能够以自己喜欢的方式来使用WCF。

——Ted Neward

博文视点Inside Microsoft SQL Server系列

SQL Server 2005微软官方权威参考手册

全球公认SQL Server 2005经典著作

数据库“铁人”、微软MVP胡百敬先生鼎力推荐

微软SQL Server总部Principal Group项目经理朱凌志鼎力推荐

《Microsoft SQL Server 2005技术内幕：T-SQL查询》

[美] Itzik Ben-Gan, Lubor Kollar, Dejan Sarka 著

赵立东 唐灿 刘波 译

赵立东 审校

■囊括最优的方法、务实的建议和简短有力的实例

《Microsoft SQL Server 2005技术内幕：存储引擎》

[美] Kalen Delaney 著

聂伟 方磊 揭磊骏 译

刘莹 审校

■Kalen Delaney亲自执笔撰写

■全面、详细地阐述SQL Server 2005存储引擎方面的知识

《Microsoft SQL Server 2005技术内幕：T-SQL程序设计》

[美] Itzik Ben-Gan, Dejan Sarka, Roger Wolter 著

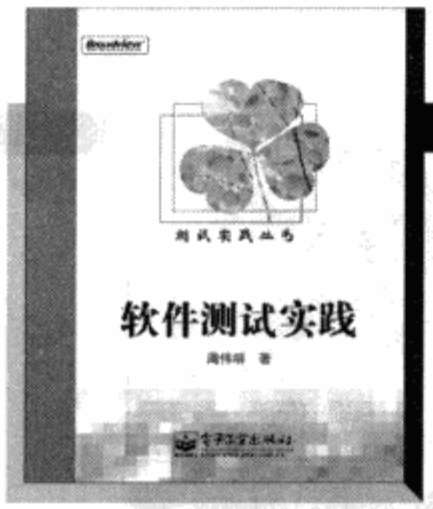
赵立东 译

■详细讲解T-SQL的内部构造

■彻底掌握T-SQL的高级用法

■深入讨论T-SQL的最新特性

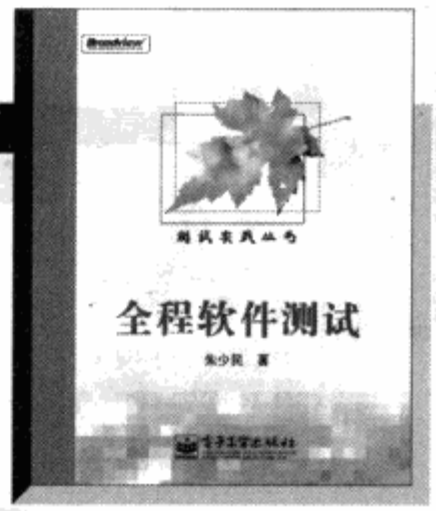
博文视点测试实践丛书



《软件测试实践》

周伟明　著

- 资深软件测试专家周伟明倾力打造
- 从实践出发，深入测试的各个细节
- 助您切实提高代码和软件的质量



《全程软件测试》

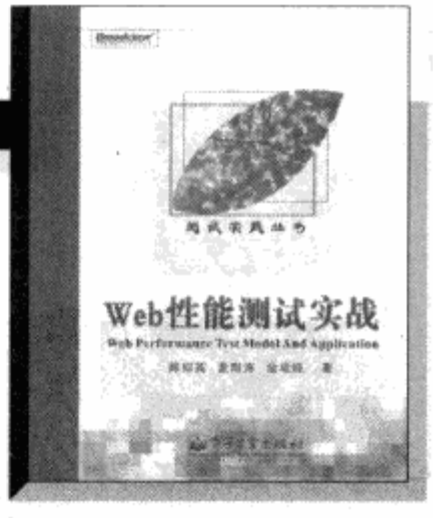
朱少民　著

- 深入国际一流软件企业的软件测试流程和实践
- 深入各项测试活动，演绎测试方法和实践

《LoadRunner性能测试实战》

陈绍英　刘建华　金成姬　著

- 第一本原创LoadRunner性能测试图书
- 深入脚本开发与性能测试分析
- 囊括大量一线实践案例



《Web性能测试实战》

陈绍英　夏海涛　金成姬　著

- 总结多年实践经验和成果
- 理论与实践紧密结合

《国际化软件测试》

崔启亮　胡一鸣　编著

- 国内全球软件工程书籍的开山之作
- 阐述国际化软件测试的根本问题
- 分享业界同行的最佳实践

博文视点原创精品大系

《Windows用户态程序高效排错》

熊力　著

- 深入剖析数十个微软企业客户的真实案例
- 用缜密的思考，恰当的工具Debug
- 让您成为福尔摩斯一样的排错高手

《圣殿祭司的ASP.NET 2.0开发详解——使用C#（第2版）》

《圣殿祭司的ASP.NET 2.0开发详解——使用VB》

奚江华　著

- 台湾微软开发技术推广经理王森先生诚挚推荐
- ASP.NET 2.0 账号权限系统彻底解析
- 全新 ASP.NET 2.0 主题
- 百分百正宗 ASP.NET 2.0 语法

《软件架构设计》

温昱　著

- 架构设计 贵在务实
- 畅销书《应用框架的设计与实现——.NET平台》译者温昱先生最新力作
- 《程序员》杂志技术主编孟岩鼎力推荐
- 包含大量来自开发一线的实际案例
- 理论与实践高度而巧妙地结合

《Framework的设计与应用——基于Windows Forms的应用开发实践》

黄忠成　著

- 剖析Framework原理
- 实践Framework应用
- 《深入剖析ASP.NET组件设计》作者黄忠成最新力作

《ASP.NET AJAX 应用剖析立即上手》

董大伟　著

- 台湾微软ASP.NET AJAX Developer Day研讨会讲师亲自撰写
- 功力尽出，技术毫无保留

Broadview®

博文视点资讯有限公司（BROADVIEW Information Co.,Ltd.）是信息产业部直属的中央一级科技与教育出版社——电子工业出版社（PHEI）与国内最大的IT技术网站CSDN.NET和最具专业水准的IT杂志社《程序员》合资成立的以IT图书出版为主业、开展相关信息和知识增值服务的资讯公司。

我们的理念是：创新专业出版体制；培养职业出版队伍；打造精品出版品牌；完善全面出版服务。

秉承博文视点的理念，博文视点的产品线为面向IT专业人员的出版物和相关服务。博文视点将重点做好以下工作：

（1）在技术领域开发专业作（译）者群体和高质量的原创图书

（2）在图书领域建立专业的选题策划和审读机制

（3）在市场领域开创有效的宣传手段和营销渠道

博文视点有效地综合了电子工业出版社、《程序员》杂志社和CSDN.NET的资源和人才，建立全新专业的立体出版机制，确立独特的出版特色和优势，将打造IT出版领域的著名品牌，并力争成为中国最具影响力的专业IT出版和服务提供商。

作为合资公司，博文视点的团队融合了各方面的精英力量：原电子工业出版社IT图书专业出版实力的代表部门——计算机图书事业部的团队；《程序员》杂志社和CSDN网站的主创人员；著名IT专业图书策划人周筠女士及其创作群。这是一个整合专业技术人员和专业出版人员的团队；这是一个充满创新意识和创作激情的团队；这是一个不断进取、追求卓越的团队。

电子工业出版社与《程序员》杂志和CSDN网站的合作以最有效率的方式形成了出版资源、媒体资源、网络资源的整合和互动，成为2003年IT出版界备受瞩目的事件。

“技术凝聚实力，专业创新出版”，BROADVIEW与您携手共迎信息时代的机遇与挑战！

博文视点

地址：北京市万寿路173信箱电子工业出版社博文视点资讯有限公司　邮 编：100036

总 机：010-88254356　传 真：010-88254356-802

作译者服务热线(Fax)：027-66012955

读者服务热线：027-66012959　读者信箱: reader@broadview.com.cn

武汉分部地址：武汉市洪山区吴家湾湖北信息产业科技大厦1402室　邮编：430074

欢迎访问以下资源网站：

博文官方博客：http://blog.csdn.net/bvbook/

博文官方网站：http://www.broadview.com.cn/

在线有奖读者调查表

Broadview®
www.broadview.com.cn

《SOA实践指南
——分布式系统设计的艺术》

http://bv.csdn.net

登录以上网站告诉我们您关于这本书的建议、意见

就有机会获赠博文视点的『**新书一本**』

并参加年终大抽奖活动

您的支持就是我们创造精品动力的源泉!

欢迎投稿: bvtougao@gmail.com

读者信箱: reader@broadview.com.cn

博文视点更多资源网站:

- VSTS虚拟社区:
  http://yishan.cc/
- 《代码大全》资源网站:
  http://www.cc2e.com.cn/
- 博文视点官方博客:
  http://blog.csdn.net/bvbook/