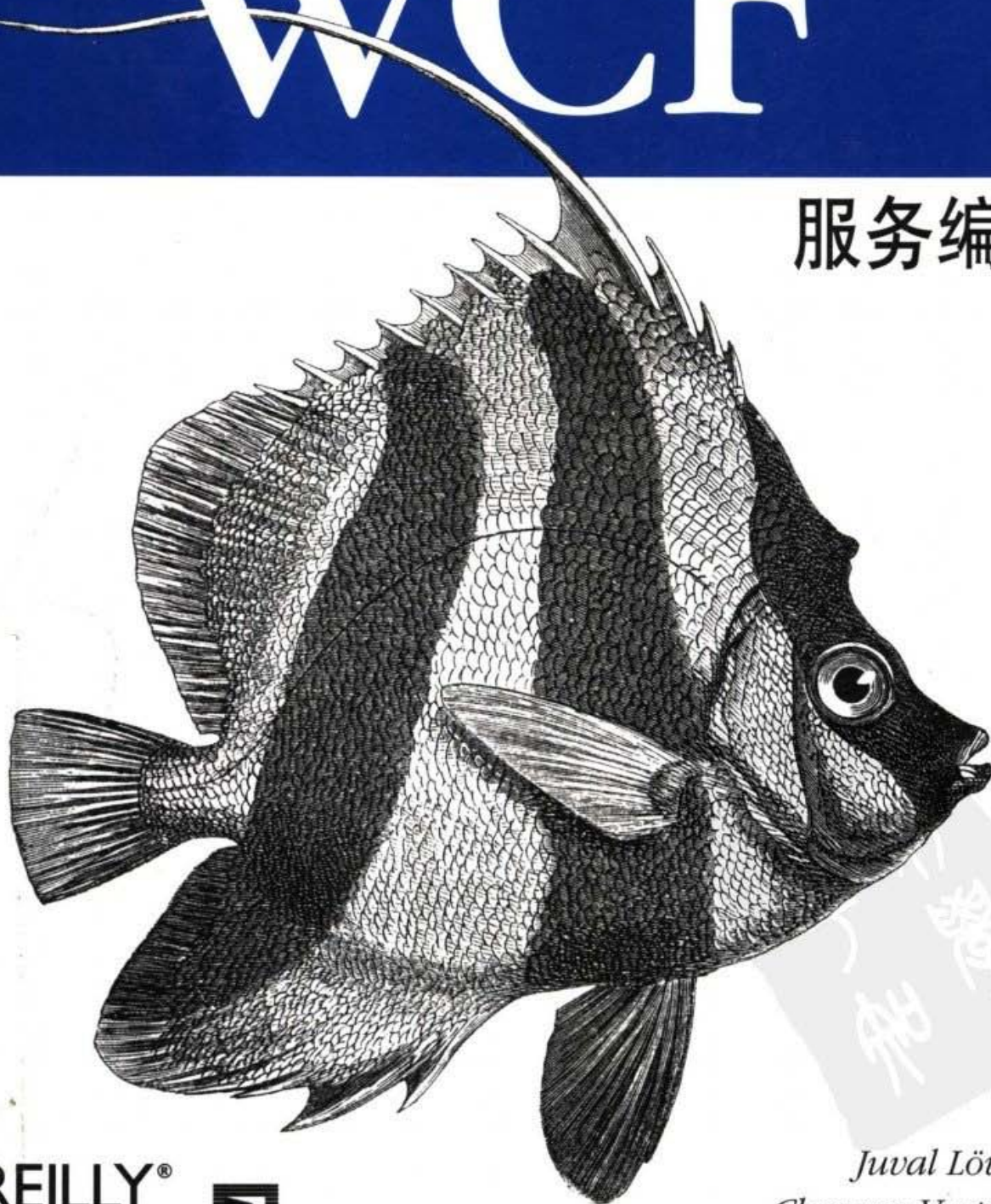


Programming WCF Services

# WCF

服务编程



O'REILLY®



机械工业出版社  
China Machine Press



Juval Löwy 著  
Clemens Vasters 序  
张逸 徐宁 译

## WCF服务编程



“Juval Löwy是当今最杰出的分布式系统专家之一。在本书中，Juval以他享有盛誉的写作技巧，深入浅出而又准确细致地介绍了WCF的体系架构。”

—— Clemens Vasters, 程序经理  
互联框架团队, Microsoft Corporation

本书是WCF的权威指南。WCF是Microsoft推出的在Windows操作系统下开发面向服务应用程序的统一平台，是革命性的技术平台。作为微软授予的“软件英杰”，本书作者Juval Löwy曾经参与了WCF的战略设计，并与WCF的开发团队一同合作，因此本书具有很高的实用价值，它对WCF进行了深入的技术剖析，而非死板的文档化描述。

本书关注隐藏在特殊设计决策之后的根本原理，这些原理包括SOA开发中极度匮乏的文档，以及难以理解的特性等。通过本书，开发者与架构师不仅能够了解如何进行WCF程序设计，还能够掌握相关的设计准则、最佳实践以及WCF存在的缺陷。

本书包括50多个工具和辅助类，以及70多个设计图，它们都设计用来提高我们的生产力，保障WCF服务的质量，同时能弥补WCF自身存在的一些缺陷，并帮助简化和自动化特定任务。

除了介绍面向服务的基础知识外，本书涵盖了以下内容：

- 服务契约分解。
- 数据契约版本控制与封送。
- 可伸缩性策略。
- 操作、调用与事件。
- 事务与错误处理。
- 并发管理。
- 队列服务。
- 面向服务安全性。

本书从软件工程的角度出发，深入探索了WCF的每个主题。本书能够使您如虎添翼，从而设计出可维护的、可扩展的、可重用的SOA应用程序。

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzsj@hzbook.com

华章网站：<http://www.hzbook.com>

网上购书：[www.china-pub.com](http://www.china-pub.com)



[www.oreilly.com](http://www.oreilly.com)

O'Reilly Media, Inc. 授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、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)

ISBN 978-7-111-22778-6



ISBN 978-7-111-22778-6

定价：75.00元



## O'Reilly Media, Inc. 介绍

为了满足读者对网络 and 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. 授权机械工业出版社，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，同时也是联机出版的先锋。

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

软件开发技术始终处于变迁之中,更新速度有时候甚至超过了定义硬件发展的摩尔定律值。诚然,软件设计思想的发展略微滞后,然而在这过往的几十年来,设计思想却仍然经历了数次重大的变迁。每一次变迁都会给整个软件开发行业带来翻天覆地的变化。在最近十余年,就在面向对象设计与面向组件设计已经成为软件开发的主流开发方法之时,软件设计又开始踏上面向服务设计的崭新之路。

从面向组件设计到面向服务设计无疑是软件设计的又一次重大飞跃,它摆脱了组件设计固有的约束与桎梏,不再依赖于实现的技术与平台,以“服务”为核心的实现原则,可以极大程度地解除组件之间的依赖关系,而面向服务框架在事务处理、安全控制、消息传输等方面提供的公共基础功能模块,又使得开发者可以从实现基础功能的工作中解放出来,服务的设计者可以只关注于与企业应用密切相关的业务逻辑。可以说,面向服务设计在互操作性、可扩展性、可重用性以及可伸缩性等诸多方面有着得天独厚的优势与巨大潜能。

是的,SOA (Service Oriented Application, 面向服务应用程序)的时代已经到来!“弄潮儿向涛头立,手把红旗旗不湿”,软件开发人员从来都不缺乏弄潮儿迎接怒涛的勇气,谁能够坚定地走在更前面,谁就能够具有更广阔的视野。

WCF无疑为.NET开发者提供了决战SOA的制胜利剑。WCF是基于Windows平台下开发和部署服务的SDK,为服务提供了运行时环境,以便于开发者设计服务、部署服务与运行服务。WCF是.NET Framework 3.0的一个重要组成部分,它集成了.NET Remoting、Web服务、WSE以及MSMQ的所有特性,使得开发者能够以统一的方式开发面向服务的应用程序。

在WCF还未推出正式版本(当时被命名为Indigo)的时候,我就开始关注它的技术元素与技术发展。出于对SOA的认知,我能够预见到WCF的远大前程。无疑,WCF是微软软件产品战略中针对企业级应用的重要产品。以WCF为基础框架搭建面向服务的企业级应用程序,以WF工作流引擎支撑企业应用中业务流程的传递与控制,以Cardspace和WCF固有的安全策略保证企业信息的安全,最后以ASP.NET AJAX、WPF和Silverlight技术丰富客户端界面的绚丽表现,从而改善企业客户对应用程序的体验,这就是微软实现企业级应用的霸业宏图。WCF在其中的地位举足轻重。

本书可以称得上是介绍WCF技术的开山之作。它全面准确地为我们描绘了一幅WCF画卷的清明上河图。Juval Löwy作为全世界享有盛誉的分布式系统专家,一位循循



善诱的技术讲师与作家，不仅具有高屋建瓴的体系架构知识，同时又能够细致入微地观察技术细节，然后用深入浅出的语言打造成通俗易懂的著作。就像清明上河图一般，巨细靡遗，浑然天成。

我很荣幸自己能够翻译 Juval Löwy 的这本巨著。然而，在兴奋之余，也难免感到诚惶诚恐。对于一本书而言，译者的翻译可以称得上是赋予书籍第二次生命，“信”乃其骨，“达”乃其肉，“雅”则为其貌。我担心自己拙劣的翻译会让这本书的第二次生命成为一次苦难的历程。翻译自古难为，译者就像走钢丝绳的杂耍者，如果不能准确、优雅地表达原作者的含义，就会失去平衡，从高达数丈的钢丝绳上摔下来。然而，这份惶恐与踌躇，终究敌不过本书给我带来的诱惑，何况，我自有作为一名技术人员的几分自信。同时，在我的诚邀之下，徐宁先生的“加盟”无疑又为本书的翻译加重了成功的砝码。作为微软的 MVP，徐宁先生曾经参与了 Patrick Smacchia 著作《Practical .NET 2 and C# 2》的翻译，拥有非常丰富的翻译经验。

本书的第7章、第9章以及第10章由徐宁先生翻译，而我则负责翻译除这几章之外的所有章节，同时包括附录、序、前言以及第10章的部分章节，并负责全书的技术审校。由于译者技术水平有限，疏漏在所难免，敬请广大读者指正。

我要感谢机械工业出版社引进这样一本介绍 WCF 技术的杰作，它必然会在面向服务设计技术领域的众多书籍中占据重要的地位，并为推动国内的 WCF 技术作出卓越贡献。感谢机械工业出版社华章分社的编辑们，正是你们的工作促成了本书中文版的顺利出版。

感谢我的合作者徐宁先生，你的加入无异于雪中送炭。否则，我一个人无法在如此短的时间内翻译完成这本厚达 600 多页的巨著。

最后，我要感谢我的父母，我的爱妻漆茜，感谢你们的宽容、体谅以及默默的支持。

—— 张逸

Microsoft Windows Server System-Connected  
System Developer, MVP

2007 年 10 月 30 日于重庆高新园

## 译者简介

---

张逸，正大软件集团高级技术管理人员、系统架构师。先后在中兴通讯、HP等国内外大型软件企业任职，参与了AAA、BOE-CIMS、NCIC-CRM和EAS等项目的管理、设计与开发。他主要专注的技术领域为.NET，他熟悉C#、ASP.NET、Web Service、.NET Remoting和WCF等技术，参与了大型项目的分布式架构设计以及企业信息化解决方案的实施。他是《软件设计精要与模式》一书的作者，在面向对象领域具有一定造诣，精通设计模式、测试驱动开发、极限编程与UML等技术或思想的运用。此外，他还担任公司的软件项目管理工作与技术团队建设，具有较为丰富的项目管理与开发经验。他是两任微软Windows Server System-Connected System Developer MVP得主，你可以通过他的个人主页 <http://www.brucezhong.com> 阅读他的技术文章。

徐宁，从2001年开始接触.NET开发，于2007年7月获得C#方向的微软MVP。对软件设计开发以及分布式技术有较大兴趣。现任职于道富信息科技(浙江)有限公司，平时喜欢阅读技术Blog和编程书籍，在博客园曾发表多篇技术文章。你可以在 <http://idior.cnblogs.com/> 阅读到他的技术文章，也可以通过 [xuning.net@gmail.com](mailto:xuning.net@gmail.com) 与他联系。





## 作者简介

---

**Juval Löwy** 是 IDesign (<http://www.idesign.net>) 的一名软件架构师和主要负责人。他致力于 WCF 体系架构设计的咨询与高级培训。Juval 是微软在硅谷的区域总监，帮助业界采用 WCF 技术。他是 O'Reilly 畅销书《Programming .NET Components》(.NET 组件程序设计) 的作者，此书被普遍认为是基于 .NET 技术开发的最优秀的一本书籍。Juval 参与了微软对于 WCF 以及相关技术的内部设计评审。他发表了大量文章，内容涵盖了 .NET 开发的方方面面，同时也是开发者大会上活跃的讲师。作为一名全球顶级的 .NET 专家和业界领袖，微软授予他软件英杰 (Software Legend) 的荣誉称号。

## 封面介绍

---

本书封面动物是一只天使鱼 (Angelfish)。在热带海洋与亚热带海洋的珊瑚礁之间常常能寻觅到它们的踪迹。天使鱼种类繁多，至少不低于 86 种。天使鱼的平均体长 7~12 英寸 (20~30 厘米)，幼鱼与成鱼的体长差异很大，色彩也会随着个体的逐渐成熟而变化。天使鱼喜食藻类、虫类以及各种贝壳和小的海洋生物。天使鱼的背鳍和臀鳍的形状与同样色彩斑斓的蝴蝶鱼不同，鳍条向后延长，侧斜舒展，状若天使的翅膀。天使鱼的繁殖习性因种类而异。大多数天使鱼会选择自然配对，相互厮守一生。一些雄鱼则会划定自己的领地，与众多雌鱼共同组成一个群体。所有天使鱼均为雌性先熟的雌雄同体动物，这意味着一旦统治领地的雄鱼死亡或者脱离了群体，为了繁殖的需要，一只雌鱼可能会转变为雄性。

少数人会将天使鱼作为日常的食品，但大多数人还是把它作为观赏动物。一些珍稀品种的天使鱼价格高达数百甚至数千美元。除了人们的捕捉，天使鱼的生存还受到了珊瑚礁的破坏以及环境持续恶化的威胁。

封面图片撷取自 Wood 关于爬行动物、鱼类、昆虫等物种的出版物。

## 图书在版编目 (CIP) 数据

WCF 服务编程 / (美) 罗威 (Löwy, J.) 著; 张逸, 徐宁译. — 北京: 机械工业出版社, 2008.1

书名原文: Programming WCF Services

ISBN 978-7-111-22778-6

I. W… II. ①罗… ②张… ③徐… III. 互联网络—网络服务器—程序设计  
IV. TP368.5

中国版本图书馆 CIP 数据核字 (2007) 第 175364 号

北京市版权局著作权合同登记

图字: 01-2007-1833 号

Copyright ©2007 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2007. 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.

英文原版由 O'Reilly Media, Inc. 出版 2007。

简体中文版由机械工业出版社出版 2007。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

本书法律顾问 北京市展达律师事务所

书 名 / WCF 服务编程

书 号 / ISBN 978-7-111-22778-6

责任编辑 / 王春华

封面设计 / Karen Montgomery, 张健

出版发行 / 机械工业出版社

地 址 / 北京市西城区百万庄大街 22 号 (邮政编码 100037)

经 销 / 新华书店北京发行所发行

印 刷 / 北京京北制版厂印刷

开 本 / 178 毫米 × 233 毫米 16 开本 37.75 印张

版 次 / 2008 年 1 月第 1 版 2008 年 1 月第 1 次印刷

印 数 / 0001-4000 册

定 价 / 75.00 元 (册)

凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换  
本社购书热线电话 (010) 68326294



# 目录

序 .....	1
前言 .....	5
第 1 章 WCF 基础 .....	13
什么是 WCF .....	13
服务 .....	14
地址 .....	16
契约 .....	19
托管 .....	23
绑定 .....	30
终结点 .....	34
元数据交换 .....	39
客户端编程 .....	47
编程方式配置与管理方式配置 .....	56
WCF 体系架构 .....	57
使用通道 .....	59
可靠性 .....	63

<b>第 2 章 服务契约 .....</b>	<b>68</b>
操作重载 .....	68
契约的继承 .....	70
服务契约的分解与设计 .....	75
契约查询 .....	79
 <b>第 3 章 数据契约 .....</b>	 <b>87</b>
序列化 .....	87
数据契约特性 .....	95
数据契约层级 .....	106
数据契约等效性 .....	114
版本控制 .....	117
枚举 .....	125
委托与数据契约 .....	127
数据集与数据表 .....	128
泛型 .....	132
集合 .....	136
 <b>第 4 章 实例管理 .....</b>	 <b>146</b>
行为 .....	146
单调服务 .....	147
会话服务 .....	153
单例服务 .....	163
分步操作 .....	168
实例停用 .....	171
限流 .....	176
 <b>第 5 章 操作 .....</b>	 <b>184</b>
请求-应答操作 .....	184
单向操作 .....	185



回调操作 .....	189
事件 .....	211
流操作 .....	215
<b>第6章 错误 .....</b>	<b>219</b>
错误与异常 .....	219
错误契约 .....	222
错误处理扩展 .....	234
<b>第7章 事务 .....</b>	<b>251</b>
恢复的挑战 .....	251
事务 .....	252
事务传播 .....	258
事务协议和管理器 .....	262
事务类 .....	267
事务型服务编程 .....	270
显式事务编程 .....	287
服务状态管理 .....	298
实例管理与事务 .....	301
回调 .....	321
<b>第8章 并发管理 .....</b>	<b>328</b>
实例管理与并发 .....	328
服务并发模式 .....	329
实例与并发访问 .....	336
资源与服务 .....	338
资源同步上下文 .....	340
服务同步上下文 .....	348
定制服务同步上下文 .....	359
回调与客户端安全 .....	366

回调与同步上下文 .....	368
异步调用 .....	376
<b>第 9 章 队列服务 .....</b>	<b>391</b>
离线服务与客户端 .....	391
队列调用 .....	392
事务 .....	400
实例管理 .....	406
并发管理 .....	413
传递故障 .....	415
回放失败 .....	423
队列调用与联机调用 .....	428
响应服务 .....	431
HTTP 桥 .....	450
<b>第 10 章 安全 .....</b>	<b>456</b>
身份验证 .....	456
授权 .....	457
传输安全 .....	458
身份管理 .....	465
总体策略 .....	465
场景驱动方式 .....	466
局域网应用程序 .....	467
互联网应用程序 .....	497
B2B 应用程序 .....	519
匿名应用程序 .....	525
无安全 .....	527
场景总结 .....	529
声明式安全框架 .....	530
安全审核 .....	547

附录 A 面向服务概述 .....	553
附录 B 发布－订阅服务 .....	562
附录 C WCF 编码规范 .....	580



## 第 1 章

# WCF 基础

本章主要介绍 WCF 的基本概念、构建模块以及 WCF 体系架构，以指导读者构建一个简单的 WCF 服务。从本章的内容中，我们可以了解到 WCF 的基本术语，包括地址 (Address)、绑定 (Binding)、契约 (Contract) 和终结点 (Endpoint, 译注 1)；了解如何托管服务，如何编写客户端代码；了解 WCF 的相关主题，诸如进程内托管 (In-Proc Hosting) 以及可靠性的实现。即使你已经熟知 WCF 的基本概念，仍然建议你快速浏览本章的内容，它不仅能够巩固你的已有知识，而且本章介绍的一些辅助类与技术术语也将有助于你阅读全书。

## 什么是 WCF

Windows 通信基础 (Windows Communication Foundation, WCF) 是基于 Windows 平台下开发和部署服务的软件开发包 (Software Development Kit, SDK)。WCF 为服务提供了运行时环境 (Runtime Environment)，使得开发者能够将 CLR 类型公开为服务，又能够以 CLR 类型的方式使用服务。理论上讲，创建服务并不一定需要 WCF，但实际上，使用 WCF 却可以使得创建服务的任务事半功倍。WCF 是微软对一系列产业标准定义的实现，包括服务交互、类型转换、封送 (Marshaling) 以及各种协议的管理。正因为如此，WCF 才能够提供服务之间的互操作性。WCF 还为开发者提供了大多数应用程序都需要的基础功能模块，提高了开发者的效率。WCF 的第一个版本为服务开发提供了许多有用的功能，包括托管 (Hosting)、服务实例管理 (Service Instance Management)、异步调用、可靠性、事务管理、离线队列调用 (Disconnected Queued Call) 以及安全

译注 1: Endpoint, 终结点, 有时候又被译为“端点”。我们踌躇了许久, 最后决定沿袭 MSDN 常用的翻译习惯。实际上, 这也是本书的约定, 对于多数关于 WCF 的技术术语, 我们尽可能地沿用 MSDN 的翻译, 除非它的翻译完全是错误的。

性。同时，WCF 还提供了设计优雅的可扩展模型，使开发人员能够丰富它的基础功能。事实上，WCF 自身的实现正是利用了这样一种可扩展模型。本书的其余章节会专注于介绍这诸多方面的内容与特征。WCF 的大部分功能都包含在一个单独的程序集 *System.ServiceModel.dll* 中，命名空间为 *System.ServiceModel*。

WCF 是 .NET 3.0 的一部分，同时需要 .NET 2.0 的支持，因此它只能运行在支持它的操作系统上。目前，这些操作系统包括 Windows Vista（客户端和服务端）、Windows XP SP2 和 Windows Server 2003 SP1 以及更新的版本。

## 服务

服务（Services）是公开的一组功能的集合。从软件设计的角度考虑，软件设计思想经历了从函数发展到对象，从对象发展到组件，再从组件发展到服务的几次变迁。在这样一个漫长的发展旅程中，最后发展到服务的一步可以说是最具革新意义的一次飞跃。面向服务（Service-Oriented, SO）是一组原则的抽象，是创建面向服务应用程序的最佳实践。如果你不熟悉面向服务的原则，可以参见附录 A，它介绍了使用面向服务的概况与目的。本书假定你对这些原则已经了然于胸。一个面向服务应用程序（SOA）将众多服务聚集到单个逻辑的应用程序中，这就类似于面向组件的应用程序聚合组件，或者面向对象的应用程序聚合对象，如图 1-1 所示。

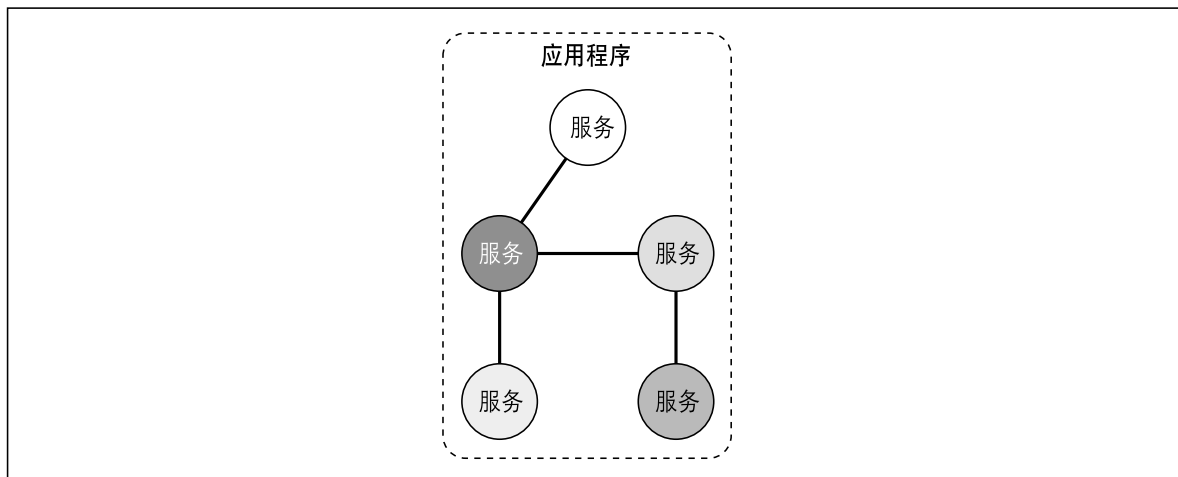


图 1-1：面向服务应用程序

服务可以是本地的，也可以是远程的，可以由多个参与方使用任意技术进行开发。服务与版本无关，甚至可以在不同的时区同时执行。服务内部包含了诸如语言、技术、平台、版本与框架等诸多概念，而服务之间的交互，则只允许指定的通信模式。



服务的客户端只是使用服务功能的一方。理论上讲,客户端可以是任意的Windows窗体类、ASP.NET 页面或其他服务。

客户端与服务通过消息的发送与接收进行交互。消息可以直接在客户端与服务之间进行传递,也可以通过中间方进行传递。WCF 中的所有消息均为 SOAP 消息。注意 WCF 的消息与传输协议无关,这与 Web 服务不同。因此, WCF 服务可以在不同的协议之间传输,而不仅限于 HTTP。WCF 客户端可以与非 WCF 服务完成互操作,而 WCF 服务也可以与非 WCF 客户端交互。不过,如果需要同时开发客户端与服务,则创建的应用程序两端都要求支持 WCF,这样才能利用 WCF 的特定优势。

因为服务的创建对于外界而言是不透明的,所以 WCF 服务通常通过公开元数据(Metadata)的方式描述可用的功能以及服务可能采用的通信方式。元数据的发布可以预先定义,它与具体的技术无关(Technology-Neutral),例如采用基于 HTTP-GET 方式的 WSDL,或者符合元数据交换的行业标准。一个非 WCF 客户端可以将元数据作为本地类型导入到本地环境中。相似的, WCF 客户端也可以导入非 WCF 服务的元数据,然后以本地 CLR 类与接口的方式进行调用。

## 服务的执行边界

WCF 不允许客户端直接与服务交互,即使它调用的是本地机器内存中的服务。相反,客户端总是使用代理(Proxy)将调用转发给服务。代理公开的操作与服务相同,同时还增加了一些管理代理的方法。

WCF 允许客户端跨越执行边界与服务通信。在同一台机器中(参见图 1-2),客户端可以调用同一个应用程序域中的服务,也可以在同一进程中跨应用程序域调用,甚至跨进程调用。

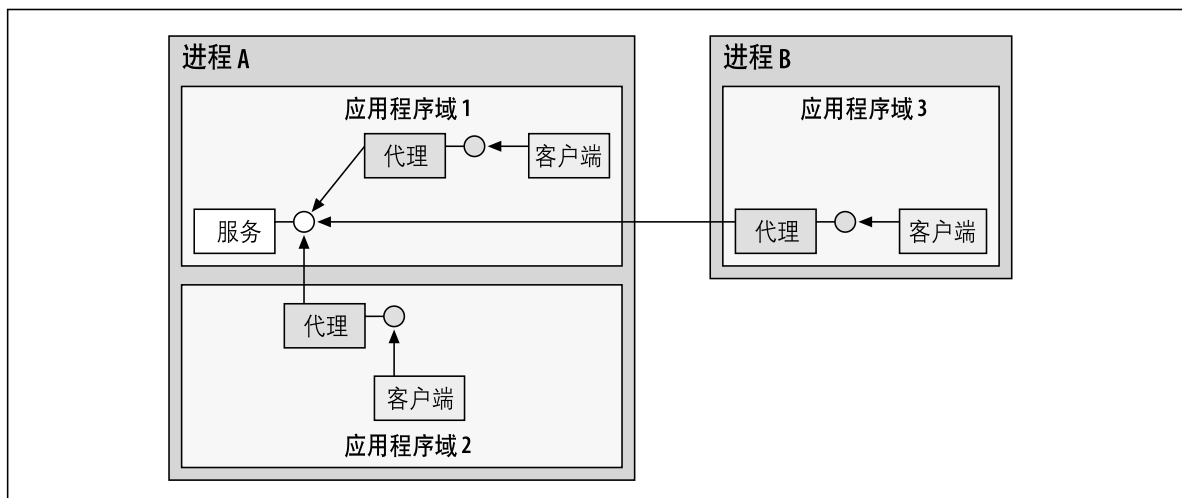


图 1-2: 使用 WCF 实现相同机器通信

图 1-3 则展示了跨机器边界的通信方式，客户端可以跨越 Intranet 或 Internet 的边界与服务交互。

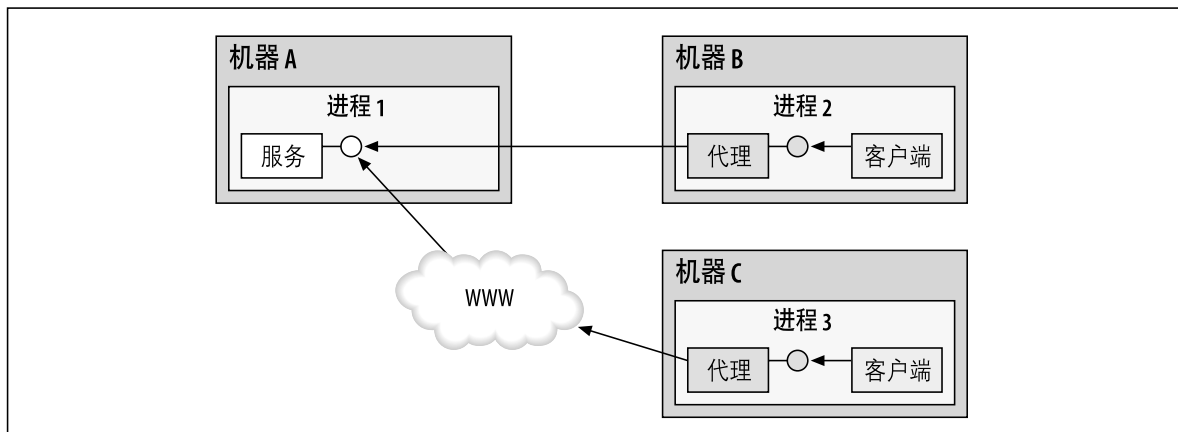


图 1-3：使用 WCF 实现不同机器通信

## WCF 与位置透明度

过去，诸如 DCOM 或 .NET Remoting 等分布式计算技术，不管对象是本地还是远程，都期望为客户端提供相同的编程模型。本地调用时，客户端使用直接引用；处理远程对象时，则使用代理。因为位置的不同而采用两种不同的编程模型会导致一个问题，就是远程调用远比本地调用复杂。复杂度体现在生命周期管理、可靠性、状态管理、可伸缩性 (scalability) 以及安全性等诸多方面。由于远程对象并不具备本地对象的特征，而编程模型却力图让它成为本地对象，反而使得远程编程模型过于复杂。WCF 同样要求客户端保持一致的编程模型，而不用考虑服务的位置。但它的实现途径却大相径庭：即使对象是本地的，WCF 仍然使用远程编程模型的实例化方式，并使用代理。由于所有的交互操作都经由代理完成，要求相同的配置与托管方式，因而对于本地和远程方式而言，WCF 都只需要维持相同的编程模型。这就使得开发者不会因为服务位置的改变影响客户端，同时还大大地简化了应用程序的编程模型。

## 地址

WCF 的每一个服务都具有一个唯一的地址 (Addresses)。地址包含两个重要元素：服务位置与传输协议 (Transport Protocol)，或者是用于服务通信的传输样式 (Transport Schema)。服务位置包括目标机器名、站点或网络、通信端口、管道或队列，以及一个可选的特定路径或者 URI。URI 即统一资源标识 (Universal Resource Identifier)，它可以是任意的唯一标识的字符串，例如服务名称或 GUID。

WCF 1.0 支持下列传输样式：

- HTTP
- TCP
- Peer network (对等网)
- IPC (基于命名管道的内部进程通信)
- MSMQ

地址通常采用如下格式:

[基地址] / [可选的 URI]

基地址 (Base Address) 通常的格式如下:

[传输协议] : // [机器名或域名] [: 可选端口]

下面是一些地址的示例:

```
http://localhost:8001
http://localhost:8001/MyService
net.tcp://localhost:8002/MyService
net.pipe://localhost/MyPipe
net.msmq://localhost/private/MyService
net.msmq://localhost/MyService
```

可以将地址 `http://localhost:8001` 读作: “采用 HTTP 协议访问 `localhost` 机器, 并在 8001 端口等待用户的调用。”

如果 URI 为 `http://localhost:8001/MyService`, 则读作: “采用 HTTP 协议访问 `localhost` 机器, `MyService` 服务在 8001 端口处等待用户的调用。”

## TCP 地址

TCP 地址采用 `net.tcp` 协议进行传输, 通常它还包括端口号, 例如:

```
net.tcp://localhost:8002/MyService
```

如果没有指定端口号, 则 TCP 地址的默认端口号为 808:

```
net.tcp://localhost/MyService
```

两个 TCP 地址 (来自于相同的宿主, 具体内容将在本章后面介绍) 可以共享一个端口:

```
net.tcp://localhost:8002/MyService
net.tcp://localhost:8002/MyOtherService
```

本书广泛地使用了基于 TCP 协议的地址。

---

注意：我们可以将不同宿主的 TCP 地址配置为共享一个端口。

---

## HTTP 地址

HTTP 地址使用 `http` 协议进行传输，也可以利用 `https` 进行安全传输。HTTP 地址通常会被用作对外的基于 Internet 的服务，并为其指定端口号，例如：

```
http://localhost:8001
```

如果没有指定端口号，则默认为 80。与 TCP 地址相似，两个相同宿主的 HTTP 地址可以共享一个端口，甚至相同的机器。

本书广泛地使用了基于 HTTP 协议的地址。

## IPC 地址

IPC 地址使用 `net.pipe` 进行传输，这意味着它将使用 Windows 的命名管道机制。在 WCF 中，使用命名管道的服务只能接收来自同一台机器的调用。因此，在使用时必须指定明确的本地机器名或者直接命名为 `localhost`，为管道名提供一个唯一的标识字符串：

```
net.pipe://localhost/MyPipe
```

每台机器只能打开一个命名管道，因此，两个命名管道地址在同一台机器上不能共享一个管道名。

本书广泛地使用了基于 IPC 的地址。

## MSMQ 地址

MSMQ 地址使用 `net.msmq` 进行传输，即使用了微软消息队列（Microsoft Message Queue，MSMQ）机制。使用时必须为 MSMQ 地址指定队列名。如果是处理私有队列，则必须指定队列类型，但对于公有队列而言，队列类型可以省略：

```
net.msmq://localhost/private/MyService  
net.msmq://localhost/MyService
```

本书第 9 章将专门介绍队列调用。

## 对等网地址

对等网地址（Peer Network Address）使用 `net.p2p` 进行传输，它使用了 Windows 的对等网传输机制。如果没有使用解析器（Resolver），我们就必须为对等网地址指定对等

网名、唯一的路径以及端口。对等网的使用与配置超出了本书范围，但在本书的后续章节中会简略地介绍对等网。

## 契约

WCF 的所有服务都会公开为契约 (Contract)。契约与平台无关，是描述服务功能的标准方式。WCF 定义了四种类型的契约。

### 服务契约 (Service Contract)

服务契约描述了客户端能够执行的服务操作。服务契约是下一章的主题内容，但书中的每一章都会广泛使用服务契约。

### 数据契约 (Data Contract)

数据契约定义了与服务交互的数据类型。WCF 为内建类型如 `int` 和 `string` 隐式地定义了契约；我们也可以非常便捷地将定制类型定义为数据契约。本书第 3 章专门介绍了数据契约的定义与使用，在后续章节中也会根据需要使用数据契约。

### 错误契约 (Fault Contract)

错误契约定义了服务抛出的错误，以及服务处理错误和传递错误到客户端的方式。第 6 章专门介绍了错误契约的定义与使用。

### 消息契约 (Message Contract)

消息契约允许服务直接与消息交互。消息契约可以是类型化的，也可以是非类型化的。如果系统要求互操作性，或者遵循已有消息格式，那么消息契约会非常有用。由于 WCF 开发者极少使用消息契约，因此本书不会介绍它。

## 服务契约

`ServiceContractAttribute` 的定义如下：

```
[AttributeUsage(AttributeTargets.Interface|AttributeTargets.Class,
    Inherited = false)]
public sealed class ServiceContractAttribute : Attribute
{
    public string Name
    {get;set;}
    public string Namespace
    {get;set;}
    // 更多成员
}
```

这个特性允许开发者定义一个服务契约。我们可以将该特性应用到接口或者类类型上，如例 1-1 所示。



## 例 1-1: 定义和实现服务契约

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    string MyMethod(string text);

    // 不会成为契约的一部分
    string MyOtherMethod(string text);
}
class MyService : IMyContract
{
    public string MyMethod(string text)
    {
        return "Hello " + text;
    }
    public string MyOtherMethod(string text)
    {
        return "Cannot call this method over WCF";
    }
}
```

`ServiceContract` 特性可以将一个 CLR 接口（或者通过推断获得的接口，后面将详细介绍）映射为与技术无关的服务契约。`ServiceContract` 特性公开了 CLR 接口（或者类）作为 WCF 契约。WCF 契约与类型的访问限定（译注 2）无关，因为类型的访问限定属于 CLR 的概念。即使将 `ServiceContract` 特性应用在内部（`Internal`）接口上，该接口同样会公开为公有服务契约，以便于跨越服务边界实现服务的调用。如果接口没有标记 `ServiceContract` 特性，WCF 客户端则无法访问它（即使接口是公有的）。这一特点遵循了面向服务的一个原则，即明确的服务边界。为满足这一原则，所有契约必须明确要求：只有接口（或者类）可以被标记为 `ServiceContract` 特性，从而被定义为 WCF 服务，其他类型都不允许。

即使应用了 `ServiceContract` 特性，类型的所有成员也不一定就是契约中的一部分。我们必须使用 `OperationContractAttribute` 特性显式地标明哪些方法需要暴露为 WCF 契约中的一部分。`OperationContractAttribute` 的定义如下：

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public string Name
    {
        get; set;
    }
    // 更多成员
}
```

---

译注 2： 类型的访问限定指 `private`、`protected`、`internal`、`protected internal` 和 `public`。

WCF 只允许将 `OperationContract` 特性应用到方法上, 而不允许应用到同样属于 CLR 概念的属性、索引器和事件上。WCF 只能识别作为逻辑功能的操作 (`Operation`)。通过应用 `OperationContract` 特性, 可以将契约方法暴露为逻辑操作, 使其成为服务契约的一部分。接口 (或类) 中的其他方法如果没有应用 `OperationContract` 特性, 则与契约无关。这有利于确保明确的服务边界, 为操作自身维护一个明确参与 (`Opt-In`) 的模型。此外, 契约操作不能使用引用对象作为参数, 只允许使用基本类型或数据契约。

### 应用 `ServiceContract` 特性

WCF 允许将 `ServiceContract` 特性应用到接口或类上。当接口应用了 `ServiceContract` 特性后, 需要定义类实现该接口。总的来讲, 我们可以使用 C# 或 VB 去实现接口, 服务类的代码无需修改, 自然而然成为一个 WCF 服务:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    string MyMethod();
}
class MyService : IMyContract
{
    public string MyMethod()
    {
        return "Hello WCF";
    }
}
```

我们可以隐式或显式实现接口:

```
class MyService : IMyContract
{
    string IMyContract.MyMethod()
    {
        return "Hello WCF";
    }
}
```

一个单独的类通过继承和实现多个标记了 `ServiceContract` 特性的接口, 可以支持多个契约。

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    string MyMethod();
}
[ServiceContract]
```

```
interface IMyOtherContract
{
    [OperationContract]
    void MyOtherMethod();
}

class MyService : IMyContract, IMyOtherContract
{
    public string MyMethod()
    { ... }
    public void MyOtherMethod()
    { ... }
}
```

然而，服务类还有一些实现上的约束。我们要避免使用带参构造函数，因为 WCF 只能使用默认构造函数。同样，虽然类可以使用内部（`internal`）（译注 3）的属性、索引器以及静态成员，但 WCF 客户端却无法访问它们。

WCF 允许我们直接将 `ServiceContract` 特性应用到服务类上，而不需要首先定义一个单独的契约：

```
// 避免
[ServiceContract]
class MyService
{
    [OperationContract]
    string MyMethod()
    {
        return "Hello WCF";
    }
}
```

通过服务类的定义，WCF 能够推断出契约的定义。至于 `OperationContract` 特性，则可以应用到类的任何一个方法（译注 4）上，不管它是私有方法，还是公有方法。

---

警告：应尽量避免将 `ServiceContract` 特性直接应用到服务类上，而应该定义一个单独的契约，这有利于在不同场景下使用契约。

---

## 名称与命名空间

可以为契约定义命名空间。契约的命名空间具有与 .NET 编程相同的目的：确定契约的

---

译注 3：这里所谓的 `internal`，并非访问限定符 `internal` 的含义，而是指未曾暴露到契约接口的相关成员。

译注 4：这里所指的方法应该为实例方法，不包括静态方法。

类型范围, 以降低类型的冲突几率。可以使用 `ServiceContract` 类型的 `Namespace` 属性设置命名空间:

```
[ServiceContract(Namespace = "MyNamespace")]
interface IMyContract
{...}
```

若非特别指定, 契约的默认命名空间为 <http://tempuri.org>。对外服务的命名空间通常使用公司的 URL; 至于企业网 (Intranet) 内部服务的命名空间, 则可以定义有意义的唯一名称, 例如 `MyApplication`。

在默认情况下, 契约公开的名称就是接口名。但是也可以使用 `ServiceContract` 特性的 `Name` 属性为契约定义别名, 从而在客户端的元数据 (Metadata) 中公开不同的名称:

```
[ServiceContract(Name = "IMyContract")]
interface IMyOtherContract
{...}
```

相似的, 操作公开的名称默认为方法名, 但我们同样可以使用 `OperationContract` 特性的 `Name` 属性设置别名, 从而公开不同的操作名:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract(Name = "SomeOperation")]
    void MyMethod(string text);
}
```

我们将在下一章介绍这些属性的使用。

## 托管

WCF 服务类不能凭空存在。每个 WCF 服务都必须托管 (Hosting) 在 Windows 进程中, 该进程被称为宿主进程 (Host Process)。单个宿主进程可以托管多个服务, 而相同的服务类型也能够托管在多个宿主进程中。WCF 没有要求宿主进程是否同时又是客户端进程。显然, 一个独立的进程有利于错误与安全的隔离。谁提供进程或是提供何种类型的进程并不重要。宿主可以由 IIS 提供, 也可以由 Windows Vista 的 Windows 激活服务 (Windows Activation Service, WAS) 提供, 或者开发者直接将它作为应用程序的一部分。

---

注意: 一种特殊的托管方式称为进程内托管 (In-Process Hosting), 简称 *in-proc*。服务与客户端驻留在相同的进程中。通过定义, 开发者能够提供进程内托管。

---

## IIS 托管

在微软的 Internet 信息服务器 (Internet Information Server, IIS) 中托管服务, 主要的优势是宿主进程可以在客户端提交第一次请求的时候自动启动, 还可以借助 IIS 管理宿主进程的生命周期。IIS 托管的主要缺点在于只能使用 HTTP 协议。如果是 IIS 5, 还要受端口限制, 要求所有服务必须使用相同的端口号。

在 IIS 中托管服务与经典的 ASMX Web 服务托管相似, 需要在 IIS 下创建虚拟目录, 并提供一个 .svc 文件。 .svc 文件的功能与 .asmx 文件相似, 主要用于识别隐藏在文件和类后面的服务代码。例 1-2 展示了 .svc 文件的语法结构。

例 1-2: .svc 文件

```
<%@ ServiceHost
    Language      = "C#"
    Debug         = "true"
    CodeBehind    = "~/App_Code/MyService.cs"
    Service       = "MyService"
%>
```

---

注意: 我们甚至可以将服务代码注入到 .svc 文件中, 但这样的做法并不明智。这与 ASMX Web 服务的要求相同。

---

使用 IIS 托管, 服务的基地址必需与 .svc 文件的地址保持一致。

## 使用 Visual Studio 2005

使用 Visual Studio 2005, 可以生成 IIS 托管服务的模版文件。选择 File 菜单的 New Website 菜单项, 然后从 New Web Site 对话框中选择 WCF Service。通过这种方式可以让 Visual Studio 2005 创建一个新的 Web 站点, 以及服务代码和对应的 .svc 文件。之后, 我们还可以通过 Add New Item 对话框添加另外的服务。

## Web.Config 文件

Web 站点的配置文件 (Web.Config) 必须列出需要公开为服务的类型。类型使用类型全名, 如果服务类型来自于一个没有被引用的程序集, 则还要包括程序集名:

```
<system.serviceModel>
  <services>
    <service name = "MyNamespace.MyService">
      ...
    </service>
  </services>
</system.serviceModel>
```



## 自托管

所谓自托管 (Self-Hosting), 就是由开发者提供和管理宿主进程的生命周期。自托管方式适用于如下场景: 需要确定客户端与服务之间的进程 (或机器) 边界时; 使用进程内托管, 即服务与客户端处于相同的进程中时。进程可以是任意的 Windows 进程, 例如 Windows 窗体应用程序、控制台应用程序或 Windows NT 服务。注意, 进程必须在客户端调用服务之前运行, 这意味着通常必须预先启动进程。但 NT 服务或进程内托管不受此限制。宿主程序的实现只需要简单的几行代码, 就能够实现 IIS 托管的一部分特性。

与 IIS 托管相似, 托管应用程序的配置文件 (*App.Config*) 必须列出所有希望托管和公开的服务类型:

```
<system.serviceModel>
  <services>
    <service name = "MyNamespace.MyService">
      ...
    </service>
  </services>
</system.serviceModel>
```

此外, 宿主进程必须在运行时显式地注册服务类型, 同时为客户端的调用打开宿主, 因此, 我们才要求宿主进程必须在客户端调用到达之前运行。创建宿主的方法通常是在 `Main()` 方法中调用 `ServiceHost` 类。`ServiceHost` 类的定义如例 1-3 所示。

### 例 1-3: `ServiceHost` 类

```
public interface ICommunicationObject
{
    void Open();
    void Close();
    // 更多成员
}

public abstract class CommunicationObject : ICommunicationObject
{...}

public abstract class ServiceHostBase : CommunicationObject, IDisposable, ...
{...}

public class ServiceHost : ServiceHostBase, ...
{
    public ServiceHost(Type serviceType, params Uri[] baseAddresses);
    // 更多成员
}
```

创建 `ServiceHost` 对象时, 需要为 `ServiceHost` 的构造函数提供服务类型, 至于默认的基地址则是可选的。可以将基地址集合设置为空。如果提供了多个基地址, 也可以将服务配置为使用不同的基地址。`ServiceHost` 拥有基地址集合可以使得服务能够接收来自于多个地址和协议的调用, 同时只需要使用相对的 URI。注意, 每个 `ServiceHost` 实例都与特定的服务类型相关, 如果宿主进程需要运行多个服务类型, 则必须创建与之

匹配的多个 `ServiceHost` 实例。在宿主程序中，通过调用 `Open()` 方法，可以允许调用传入；通过调用 `Close()` 方法终结宿主实例，完成进程中的调用。此时，即使宿主进程还在运行，仍然会拒绝客户端的调用。而在通常情况下，执行关闭操作会停止宿主进程。例如，在 **Windows** 窗体应用程序中托管服务：

```
[ServiceContract]
interface IMyContract
{...}
class MyService : IMyContract
{...}
```

我们可以编写如下的托管代码：

```
public static void Main()
{
    Uri baseAddress = new Uri("http://localhost:8000/");
    ServiceHost host = new ServiceHost(typeof(MyService),baseAddress);

    host.Open();

    // 可以执行用于阻塞的调用：
    Application.Run(new MyForm());

    host.Close();
}
```

打开宿主时，将装载 **WCF** 运行时 (**WCF runtime**)，启动工作线程监控传入的请求。由于引入了工作线程，因此可以在打开宿主之后执行阻塞 (**blocking**) 操作。通过显式控制宿主的打开与关闭，提供了 **IIS** 托管难以实现的特征，即能够创建定制的应用程序控制模块，管理者可以随意地打开和关闭宿主，而不用每次停止宿主的运行。

## 使用 Visual Studio 2005

**Visual Studio 2005** 允许开发者为任意的应用程序项目添加 **WCF** 服务，方法是在 **Add New Item** 对话框中选择 **WCF Service** 选项。当然，这种方式添加的服务，对于宿主进程而言属于进程内托管方式，但进程外的客户端仍然可以访问它。

## 自托管与基地址

启动服务宿主时，无需提供任何基地址：

```
public static void Main()
{
    ServiceHost host = new ServiceHost(typeof(MyService));

    host.Open();

    Application.Run(new MyForm());
}
```

```
        host.Close();  
    }
```

警告：但是我们不能向空列表传递 `null` 值，这会导致抛出异常：

```
serviceHost host;  
host = new ServiceHost(typeof(MyService), null);
```

只要这些地址没有使用相同的传输样式 (Transport Schema)，我们也可以注册多个基地址，并以逗号作为地址之间的分隔符。代码实现如下所示（注意例 1-3 中 `params` 限定符的使用）：

```
Uri tcpBaseAddress = new Uri("net.tcp://localhost:8001/");  
Uri httpBaseAddress = new Uri("http://localhost:8002/");  
  
ServiceHost host = new ServiceHost(typeof(MyService),  
                                     tcpBaseAddress, httpBaseAddress);
```

WCF 也允许开发者在宿主配置文件中列出基地址内容：

```
<system.serviceModel>  
  <services>  
    <service name = "MyNamespace.MyService">  
      <host>  
        <baseAddresses>  
          <add baseAddress = "net.tcp://localhost:8001/" />  
          <add baseAddress = "http://localhost:8002/" />  
        </baseAddresses>  
      </host>  
      ...  
    </service>  
  </services>  
</system.serviceModel>
```

创建宿主时，无论在配置文件中找到哪一个基地址，宿主都会使用它，同时还要加上以编程方式提供的基地址。需要特别注意，我们必须确保配置的基地址的样式不能与代码中的基地址的样式重叠。

我们甚至可以针对相同的类型注册多个宿主，只要这些宿主使用了不同的基地址：

```
Uri baseAddress1 = new Uri("net.tcp://localhost:8001/");  
ServiceHost host1 = new ServiceHost(typeof(MyService), baseAddress1);  
host1.Open();  
  
Uri baseAddress2 = new Uri("net.tcp://localhost:8002/");  
ServiceHost host2 = new ServiceHost(typeof(MyService), baseAddress2);  
host2.Open();
```

然而，这并不包括第 8 章介绍的使用线程的情况，以这种方式打开多个宿主并无优势可

言。此外,如果基地址是配置文件提供的,那么就需要使用 `ServiceHost` 的构造函数为相同的类型打开多个宿主。

## 托管的高级特性

`ServiceHost` 实现的 `ICommunicationObject` 接口定义了一些高级特性,如例 1-4 所示。

### 例 1-4: `ICommunicationObject` 接口

```
public interface ICommunicationObject
{
    void Open();
    void Close();
    void Abort();

    event EventHandler Closed;
    event EventHandler Closing;
    event EventHandler Faulted;
    event EventHandler Opened;
    event EventHandler Opening;

    IAsyncResult BeginClose(AsyncCallback callback, object state);
    IAsyncResult BeginOpen(AsyncCallback callback, object state);
    void EndClose(IAsyncResult result);
    void EndOpen(IAsyncResult result);

    CommunicationState State
    {get;}
    // 更多成员
}
public enum CommunicationState
{
    Created,
    Opening,
    Opened,
    Closing,
    Closed,
    Faulted
}
```

如果打开或关闭宿主的操作耗时较长,可以采用异步方式调用 `BeginOpen()` 和 `BeginClose()` 方法。我们可以订阅诸如状态改变或错误发生等宿主事件,通过调用 `State` 属性查询当前的宿主状态。`ServiceHost` 类同样实现了 `Abort()` 方法。该方法提供强行退出功能,能够及时中断进程中的所有服务调用,然后关闭宿主。此时,活动的客户端会获得一个异常。

## `ServiceHost<T>` 类

`ServiceHost<T>` 类能够改进 WCF 提供的 `ServiceHost` 类,它的定义如例 1-5 所示。

## 例 1-5: ServiceHost&lt;T&gt; 类

```
public class ServiceHost<T> : ServiceHost
{
    public ServiceHost() : base(typeof(T))
    {}
    public ServiceHost(params string[] baseAddresses) :
        base(typeof(T), Convert(baseAddresses))
    {}
    public ServiceHost(params Uri[] baseAddresses) :
        base(typeof(T), baseAddresses)
    {}
    static Uri[] Convert(string[] baseAddresses)
    {
        Converter<string,Uri> convert = delegate(string address)
        {
            return new Uri(address);
        };
        return Array.ConvertAll(baseAddresses, convert); (译注 5)
    }
}
```

ServiceHost<T> 简化了构造函数，它不需要传递服务类型作为构造函数的参数，还能够直接处理字符串而不是处理令人生厌的 Uri 值。在本书余下的内容中，对 ServiceHost<T> 进行了扩展，增加了一些特性，提高了它的性能。

## WAS 托管

Windows 激活服务 (WAS) 是一个系统服务，只适用于 Windows Vista。WAS 是 IIS 7 的一部分，但也可以独立地安装与配置。若要使用 WAS 托管 WCF 服务，必须提供一个 .svc 文件，这与 IIS 托管一样。IIS 与 WAS 的主要区别在于 WAS 并不局限于使用 HTTP，它支持所有可用的 WCF 传输协议、端口与队列。

WAS 提供了大量基于自托管的强大功能，包括应用程序池、回收机制、空闲时间管理 (Idle Time Management)、身份管理 (Identity Management) 以及隔离 (Isolation)；宿主进程可以根据情况选择使用这些功能。若需考虑可扩展性，就应该使用 Vista 服务器作为目标机器；如果只有少数客户端，则可以将 Vista 客户机作为服务器。

---

译注 5: Array.ConvertAll() 方法属于 .NET 2.0 新增的方法，其方法签名为：

```
public static TOutput[] ConvertAll<TInput,TOutput> (
    TInput[] array,
    Converter<TInput,TOutput> converter
)
```

它能够根据 Converter 对象将一种类型的数组转换为另一种类型的数组。



当然，自托管进程还提供了许多卓越特性，例如进程内宿主、匿名用户环境的处理，同时还为之前介绍的高级宿主特性提供了便捷地编程访问方式。

## 绑定

服务之间的通信方式是多种多样的，有多种可能的通信模式。包括：同步的请求/应答（Request/Reply）消息，或者异步的“即发即弃（Fire-and-Forget）”消息；双向（Bidirectional）消息；即时消息或队列消息；以及持久（Durable）队列或者可变（Volatile）队列。传递消息的传输协议包括：HTTP（或HTTPS）、TCP、P2P（对等网）、IPC（命名管道）以及 MSMQ。消息编码格式包括：保证互操作性的纯文本编码格式；优化性能的二进制编码格式；提供有效负载的 MTOM（消息传输优化机制，Message Transport Optimization Mechanism）编码格式。消息的安全保障也有多种策略，包括：不实施任何安全策略；只提供传输层的安全策略；消息层的隐私保护与安全策略。当然，WCF 还包括多种对客户端认证与授权的安全策略。消息传递（Message Delivery）可能是不可靠的，也可能是可靠的端对端跨越中间方，然后断开连接的方式。消息传递可能按照发送消息的顺序处理，也可能按照接收消息的顺序处理。服务可能需要与其他服务或客户端交互，这些服务或客户端或者只支持基本的 Web 服务协议，或者使用了流行的 WS-\* 协议，例如 WS-Security 或者 WS-Atomic Transaction。服务可能会基于原来的 MSMQ 消息与旧的客户端（Legacy Client）交互，或者限制服务只能与其他的 WCF 服务或客户端交互。

若要计算所有可能的通信模式与交互方式之间的组合，数量可能达到上千万。在这些组合选项中，有的可能是互斥的，有的则彼此约束。显然，客户端与服务必须合理地组合这些选项，才能保证通信的顺畅。对于大多数应用程序而言，管理如此程度的复杂度并无业务价值。然而，一旦因此作出错误决定，就会影响系统的效率与质量，造成严重的后果。

为了简化这些选项，使它们易于管理，WCF 引入了绑定（Binding）技术将这些通信特征组合在一起。一个绑定封装了诸如传输协议、消息编码、通信模式、可靠性、安全性、事务传播以及互操作性等相关选项的集合，使得它们保持一致。理想状态下，我们希望将所有繁杂的基础功能模块从服务代码中解放出来，允许服务只需要关注业务逻辑的实现。绑定使得开发者能够基于不同的基础功能模块使用相同的服务逻辑。

在使用 WCF 提供的绑定时，可以调整绑定的属性，也可以从零开始定制自己的绑定。服务在元数据中发布绑定的选项，由于客户端使用的绑定必须与服务的绑定完全相同，因此客户端能够查询绑定的类型与特定属性。单个服务能够支持各个地址上的多个绑定。

## 标准绑定

WCF 定义了 9 种标准绑定：

### 基本绑定 (*Basic Binding*)

由 `BasicHttpBinding` 类提供。基本绑定能够将 WCF 服务公开为旧的 ASMX Web 服务，使得旧的客户端能够与新的服务协作。如果客户端使用了基本绑定，那么新的 WCF 客户端就能够与旧的 ASMX 服务协作。

### TCP 绑定

由 `NetTcpBinding` 类提供。TCP 绑定使用 TCP 协议实现在 Intranet 中跨机器的通信。TCP 绑定支持多种特性，包括可靠性、事务性、安全性以及 WCF 之间通信的优化。前提是，它要求客户端与服务都必须使用 WCF。

### 对等网绑定

由 `NetPeerTcpBinding` 类提供。它使用对等网进行传输。对等网允许客户端与服务订阅相同的网格 (Grid)，实现广播消息。因为对等网需要网格拓扑 (Grid Topology) 与网状计算策略 (Mesh Computing Strategies) 方面的知识，故而不在于本书讨论范围之内。

### IPC 绑定

由 `NetNamedPipeBinding` 类提供。它使用命名管道为同一机器的通信进行传输。这种绑定方式最安全，因为它不能接收来自机器外部的调用。IPC 绑定支持的特性与 TCP 绑定相似。

### Web 服务 (WS) 绑定

由 `WSHttpBinding` 类提供。WS 绑定使用 HTTP 或 HTTPS 进行传输，为基于 Internet 的通信提供了诸如可靠性、事务性与安全性等特性。

### WS 联邦绑定 (译注 6) (*Federated WS Binding*)

由 `WSFederationHttpBinding` 类提供。WS 联邦绑定是一种特殊的 WS 绑定，提供对联邦安全 (Federated Security) 的支持。联邦安全不在于本书讨论范围之内。

### WS 双向绑定 (*Duplex WS Binding*)

由 `WSDualHttpBinding` 类提供。WS 双向绑定与 WS 绑定相似，但它还支持从服务到客户端的双向通信，相关内容在第 5 章介绍。

---

译注 6： `WSFederationHttpBinding` 支持 WS-Federation 安全通信协议。WS-Federation 是 WS 联盟协议的一部分，定义了如何创建跨越多个安全区域的联邦会话，以便在经过单次身份认证后即可使用部署在多个安全区域内的 Web 服务。它是一个联邦中多个实体相互信任的安全机制。对于该安全机制，我们可以想象一下联邦国家中各个成员体之间的安全关系。追本溯源，这也正是它命名为 WS-Federation 的主要原因。

### MSMQ 绑定

由 `NetMsmqBinding` 类提供。它使用 MSMQ 进行传输，用以提供对断开的队列调用的支持。相关内容在第 9 章介绍。

### MSMQ 集成绑定 (*MSMQ Integration Binding*)

由 `MsmqIntegrationBinding` 类提供。它实现了 WCF 消息与 MSMQ 消息之间的转换，用以支持与旧的 MSMQ 客户端之间的互操作。MSMQ 集成绑定不在本书讨论范围之内。

## 格式与编码

每种标准绑定使用的传输协议与编码格式都不相同，如表 1-1 所示。

表 1-1: 标准绑定的传输协议与编码格式 (默认的编码格式为黑体)

名字	传输协议	编码格式	互操作性
<code>BasicHttpBinding</code>	HTTP/HTTPS	<b>Text</b> , MTOM	Yes
<code>NetTcpBinding</code>	TCP	Binary	No
<code>NetPeerTcpBinding</code>	P2P	Binary	No
<code>NetNamedPipeBinding</code>	IPC	Binary	No
<code>WSHttpBinding</code>	HTTP/HTTPS	<b>Text</b> , MTOM	Yes
<code>WSFederationHttpBinding</code>	HTTP/HTTPS	<b>Text</b> , MTOM	Yes
<code>WSDualHttpBinding</code>	HTTP	<b>Text</b> , MTOM	Yes
<code>NetMsmqBinding</code>	MSMQ	Binary	No
<code>MsmqIntegrationBinding</code>	MSMQ	Binary	Yes

文本编码格式允许 WCF 服务 (或客户端) 能够通过 HTTP 协议与其他服务 (或客户端) 通信，而不用考虑它使用的技术。二进制编码格式通过 TCP 或 IPC 协议通信，它所获得的最佳性能是以牺牲互操作性为代价的，它只支持 WCF 到 WCF 的通信。

## 选择绑定

为服务选择绑定应该遵循图 1-4 所示的决策活动图表。

首先需要确认服务是否需要与非 WCF 的客户端交互。如果是，同时客户端又是旧的 MSMQ 客户端，选择 `MsmqIntegrationBinding` 绑定就能够使得服务通过 MSMQ 与该客户端实现互操作。如果服务需要与非 WCF 客户端交互，并且该客户端期望调用基本的 Web 服务协议 (ASMX Web 服务)，那么选择 `BasicHttpBinding` 绑定就能够模拟

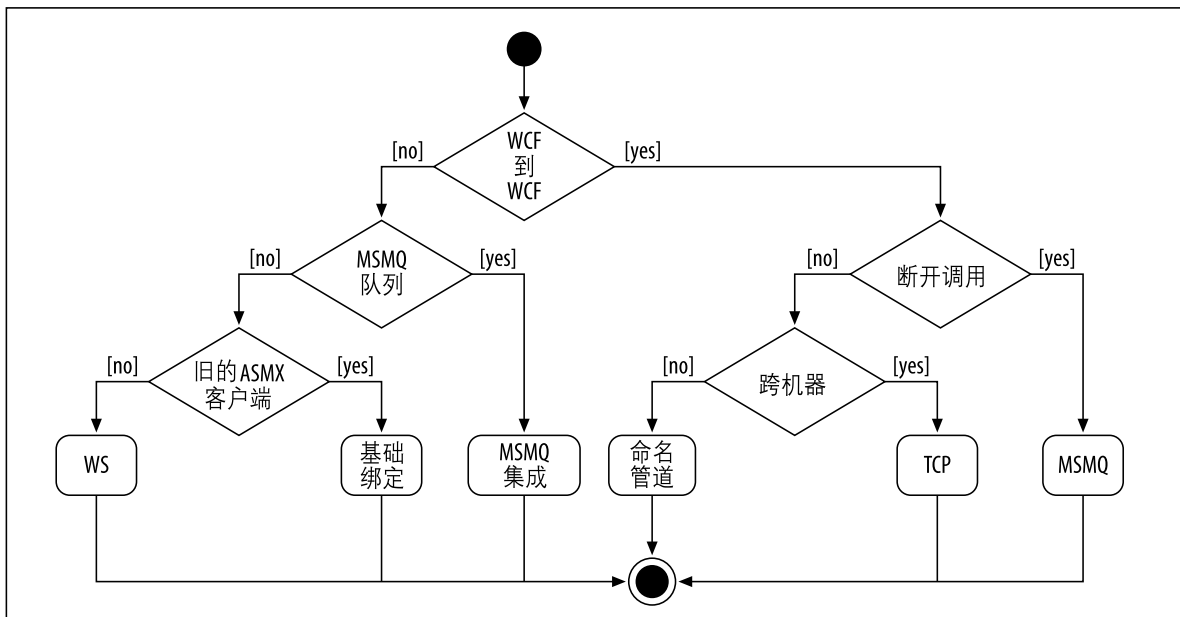


图 1-4：选择绑定

ASMX Web 服务（即 WSI-Basic Profile）公开 WCF 服务。缺点是我们无法使用大多数最新的 WS-\* 协议的优势。但是，如果非 WCF 客户端能够识别这些标准，就应该选择其中一种 WS 绑定，例如 `WSHttpBinding`、`WSFederationBinding` 或者 `WSDualHttpBinding`。如果假定客户端为 WCF 客户端，同时需要支持脱机或断开状态下的交互，则可以选择 `NetMsmqBinding` 使用 MSMQ 传输消息。如果客户端需要联机通信，但是需要跨机器边界调用，则应该选择 `NetTcpBinding` 通过 TCP 协议进行通信。如果相同机器上的客户端同时又是服务，选择 `NetNamePipeBinding` 使用命名管道可以使性能达到最优化。如果基于额外的标准，例如回调（选择 `WSDualHttpBinding`）或者联邦安全（选择 `WSFederationBinding`），则应对选择的绑定进行微调。

注意：即使超出了使用的目标场景，大多数绑定工作仍然良好。例如，我们可以使用 TCP 绑定实现相同机器甚至进程内的通信；我们也可以使用基本绑定实现 Intranet 中 WCF 对 WCF 的通信。然而，我们还是应尽量按照图 1-4 选择绑定。

## 使用绑定

每种绑定都提供了多种可配置的属性。绑定有三种工作模式。如果内建绑定符合开发者的需求，就可以直接使用它们。我们也可以对绑定的某些属性如事务传播、可靠性和安全性进行调整与配置，还可以定制自己的绑定。最常见的情况是使用已有的绑定，然后只对绑定的几个方面进行配置。应用程序开发者几乎不需要编写定制绑定，但这却是框架开发者可能需要做的工作。

## 终结点

服务与地址、绑定以及契约有关。其中，地址定义了服务的位置，绑定定义了服务通信的方式，契约则定义了服务的内容。为便于记忆，我们可以将这种类似于“三权分立”一般管理服务的方式简称为服务的 *ABC*。WCF 用终结点表示这样一种组成关系。终结点就是地址、契约与绑定的混成品（参见图 1-5）。

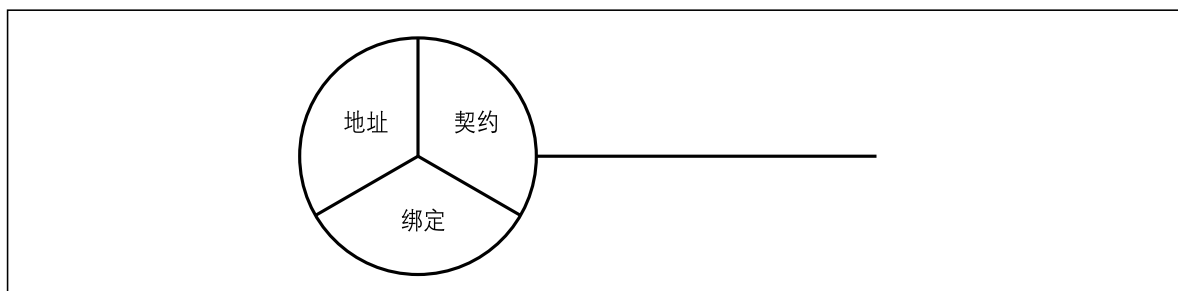


图 1-5：终结点

每一个终结点都包含了三个元素，而宿主则负责公开终结点。从逻辑上讲，终结点相当于服务的接口，就像 CLR 或者 COM 接口一样。注意，图 1-5 使用了传统的“棒棒糖”形式展示了一个终结点的构成。

---

注意：从概念上讲，不管是 C# 还是 VB，一个接口就相当于一个终结点：地址就是类型虚拟表的内存地址，绑定则是 CLR 的 JIT (Just-In-Time) 编译，而契约则代表接口本身。由于经典的 .NET 编程模式不需要处理地址或绑定，你可能认为它们是理所当然存在的。而 WCF 并未规定地址与绑定，因而必须对它们进行配置。

---

每个服务至少必须公开一个业务终结点，每个终结点有且只能拥有一个契约。服务上的所有终结点都包含了唯一的地址，而一个单独的服务则可以公开多个终结点。这些终结点可以使用相同或不同的绑定，公开相同或不同的契约。每个服务提供的不同终结点之间绝对没有任何关联。

重要的一点是，服务代码并没有包含它的终结点，它们通常放在服务代码之外。我们可以通过管理方式 (Administratively) 使用配置文件或者通过编程方式 (Programmatically) 配置终结点。

## 管理方式配置终结点

以管理方式配置一个终结点需要将终结点放到托管进程的配置文件中，如下的服务定义：



```
namespace MyNamespace
{
    [ServiceContract]
    interface IMyContract
    { ... }
    class MyService : IMyContract
    { ... }
}
```

例 1-6 演示了配置文件要求的配置入口。在每个服务类型下列出它的终结点。

#### 例 1-6: 管理方式配置终结点

```
<system.serviceModel>
  <services>
    <service name = "MyNamespace.MyService">
      <endpoint
        address = "http://localhost:8000/MyService/"
        binding = "wsHttpBinding"
        contract = "MyNamespace.IMyContract"
      />
    </service>
  </services>
</system.serviceModel>
```

当我们指定服务和契约类型时,必须使用类型全名。在本书的其余例子中,为简略起见,我省略了类型的命名空间,但在实际应用中,命名空间是必备的。注意,如果终结点已经提供了基地址,则地址的样式必须与绑定一致,例如 HTTP 对应 WSHttpBinding。如果两者不匹配,就会在装载服务时导致异常。

例 1-7 的配置文件为一个单独的服务公开了多个终结点。多个终结点可以配置相同的基地址,前提是 URI 互不相同。

#### 例 1-7: 相同服务的多个终结点

```
<service name = "MyService">
  <endpoint
    address = "http://localhost:8000/MyService/"
    binding = "wsHttpBinding"
    contract = "IMyContract"
  />
  <endpoint
    address = "net.tcp://localhost:8001/MyService/"
    binding = "netTcpBinding"
    contract = "IMyContract"
  />
  <endpoint
    address = "net.tcp://localhost:8002/MyService/"
    binding = "netTcpBinding"
    contract = "IMyOtherContract"
  />
</service>
```

大多数情况下，我们的首选是管理的配置方式，因为它非常灵活，即使修改了服务的地址、绑定和契约，也不需要重新编译服务和重新部署服务。

## 使用基地址

例 1-7 中的每个终结点都提供了自己独有的基地址。如果我们提供了显式的基地址，它会重写宿主提供的所有基地址。

我们也可以让多个终结点使用相同的基地址，只要终结点地址中的 URI 不同：

```
<service name = "MyService">
  <endpoint
    address = "net.tcp://localhost:8001/MyService/"
    binding = "netTcpBinding"
    contract = "IMyContract"
  />
  <endpoint
    address = "net.tcp://localhost:8001/MyOtherService/"
    binding = "netTcpBinding"
    contract = "IMyContract"
  />
</service>
```

反之，如果宿主提供了与传输样式匹配的基地址，则可以省略地址项。此时，终结点地址与该基地址完全相同：

```
<endpoint
  binding = "wsHttpBinding"
  contract = "IMyContract"
/>
```

如果宿主没有提供匹配的基地址，则在装载服务宿主时会抛出异常。

配置终结点地址时，可以为基地址添加相对 URI：

```
<endpoint
  address = "SubAddress"
  binding = "wsHttpBinding"
  contract = "IMyContract"
/>
```

此时，终结点地址等于它所匹配的基地址加上 URI。当然，前提是宿主必须提供匹配的基地址。

## 绑定配置

使用配置文件可以为终结点使用的绑定进行定制。为此，需要在 <endpoint> 节中添加 bindingConfiguration 标志，它的值应该与 <bindings> 配置节中定制的绑定名一

致。例 1-8 介绍了使用这种技术启用事务传播的方法。其中的 `transactionFlow` 标志会在第 7 章详细介绍。

#### 例 1-8: 服务端绑定的配置

```
<system.serviceModel>
  <services>
    <service name = "MyService">
      <endpoint
        address = "net.tcp://localhost:8000/MyService/"
        bindingConfiguration = "TransactionalTCP"
        binding = "netTcpBinding"
        contract = "IMyContract"
      />
      <endpoint
        address = "net.tcp://localhost:8001/MyService/"
        bindingConfiguration = "TransactionalTCP"
        binding = "netTcpBinding"
        contract = "IMyOtherContract"
      />
    </service>
  </services>
  <bindings>
    <netTcpBinding>
      <binding name = "TransactionalTCP"
        transactionFlow = "true"
      />
    </netTcpBinding>
  </bindings>
</system.serviceModel>
```

如例 1-8 所示，我们可以在多个终结点中通过指向定制绑定的方式，重用已命名的绑定配置。

### 编程方式配置终结点

编程方式配置终结点与管理方式配置终结点等效。但它不需要配置文件，而是通过编程调用将终结点添加到 `ServiceHost` 实例中。这些调用不属于服务代码的范围。`ServiceHost` 定义了重载版本的 `AddServiceEndpoint()` 方法：

```
public class ServiceHost : ServiceHostBase
{
    public ServiceEndpoint AddServiceEndpoint(Type implementedContract,
                                              Binding binding,
                                              string address);

    // 其他成员
}
```

传入 `AddServiceEndpoint()` 方法的地址可以是相对地址，也可以是绝对地址，这与

使用配置文件的方式相似。例 1-9 演示了编程配置的方法，它配置的终结点与例 1-7 的终结点相同。

#### 例 1-9：服务端编程配置终结点

```
ServiceHost host = new ServiceHost(typeof(MyService));

Binding wsBinding = new WSHttpBinding();
Binding tcpBinding = new NetTcpBinding();

host.AddServiceEndpoint(typeof(IMyContract), wsBinding,
    "http://localhost:8000/MyService");
host.AddServiceEndpoint(typeof(IMyContract), tcpBinding,
    "net.tcp://localhost:8001/MyService");
host.AddServiceEndpoint(typeof(IMyOtherContract), tcpBinding,
    "net.tcp://localhost:8002/MyService");

host.Open();
```

以编程方式添加终结点时，`address` 参数为 `string` 类型，`contract` 参数为 `Type` 类型，而 `binding` 参数的类型则是 `Binding` 抽象类的其中一个子类，例如：

```
public class NetTcpBinding : Binding, ...
{ ... }
```

由于宿主提供了基地址，因此若要使用基地址，可以将空字符串赋给 `address` 参数，或者只设置 `URI` 值，此时使用的地址就应该是基地址加上 `URI`：

```
Uri tcpBaseAddress = new Uri("net.tcp://localhost:8000/");

ServiceHost host = new ServiceHost(typeof(MyService), tcpBaseAddress);

Binding tcpBinding = new NetTcpBinding();

// 使用基地址作为地址
host.AddServiceEndpoint(typeof(IMyContract), tcpBinding, "");
// 添加相对地址
host.AddServiceEndpoint(typeof(IMyContract), tcpBinding, "MyService");
// 忽略基地址
host.AddServiceEndpoint(typeof(IMyContract), tcpBinding,
    "net.tcp://localhost:8001/MyService");

host.Open();
```

使用配置文件进行管理方式的配置，宿主必须提供一个匹配的基地址，否则会引发异常。事实上，编程方式配置与管理方式配置并没有任何区别。使用配置文件时，WCF 会解析文件，然后执行对应的编程调用。

#### 绑定配置

我们可以通过编程方式设置绑定的属性。例如，以下代码就实现了与例 1-8 相似的功能，启用事务传播：

```
ServiceHost host = new ServiceHost(typeof(MyService));  
  
NetTcpBinding tcpBinding = new NetTcpBinding();  
  
tcpBinding.TransactionFlow = true;  
  
host.AddServiceEndpoint(typeof(IMyContract), tcpBinding,  
                           "net.tcp://localhost:8000/MyService");  
  
host.Open();
```

注意, 在处理特定的绑定属性时, 通常应该与具体的绑定子类如 `NetTcpBinding` 交互, 而不是使用抽象类 `Binding`, 正如例 1-9 所示。

## 元数据交换

服务有两种方案可以发布自己的元数据。一种是基于 HTTP-GET 协议提供元数据, 另一种则是后面将要讨论的使用专门的终结点的方式。WCF 能够为服务自动提供基于 HTTP-GET 的元数据, 但需要显式地添加服务行为 (Behavior) 以支持这一功能。本书后面的章节会介绍行为的相关知识。现在, 我们只需要知道行为属于服务的本地特性, 例如是否需要基于 HTTP-GET 交换元数据, 就是一种服务行为。我们可以通过编程方式或管理方式添加行为。在例 1-10 演示的宿主应用程序的配置文件中, 所有引用了定制 `<behavior>` 配置节的托管服务都支持基于 HTTP-GET 协议实现元数据交换。为了使用 HTTP-GET, 客户端使用的地址需要注册服务的 HTTP 基地址。我们也可以在行为中指定一个外部 URL 以达到同样的目的。

例 1-10: 使用配制文件启用元数据交换行为

```
<system.serviceModel>  
  <services>  
    <service name = "MyService" behaviorConfiguration = "MEXGET">  
      <host>  
        <baseAddresses>  
          <add baseAddress = "http://localhost:8000/" />  
        </baseAddresses>  
      </host>  
      ...  
    </service>  
    <service name = "MyOtherService" behaviorConfiguration = "MEXGET">  
      <host>  
        <baseAddresses>  
          <add baseAddress = "http://localhost:8001/" />  
        </baseAddresses>  
      </host>  
      ...  
    </service>  
  </services>  
</behaviors>
```

```
<serviceBehaviors>
  <behavior name = "MEXGET">
    <serviceMetadata httpGetEnabled = "true"/>
  </behavior>
</serviceBehaviors>
</behaviors>
</system.serviceModel>
```

一旦启用了基于 HTTP-GET 的元数据交换,在浏览器中就可以通过 HTTP 基地址(如果存在)进行访问。如果一切正确,就会获得一个确认页面,如图 1-6 所示,告知开发者已经成功托管了服务。确认页面与 IIS 托管无关,即使使用自托管,我们也可以使用浏览器定位服务地址。

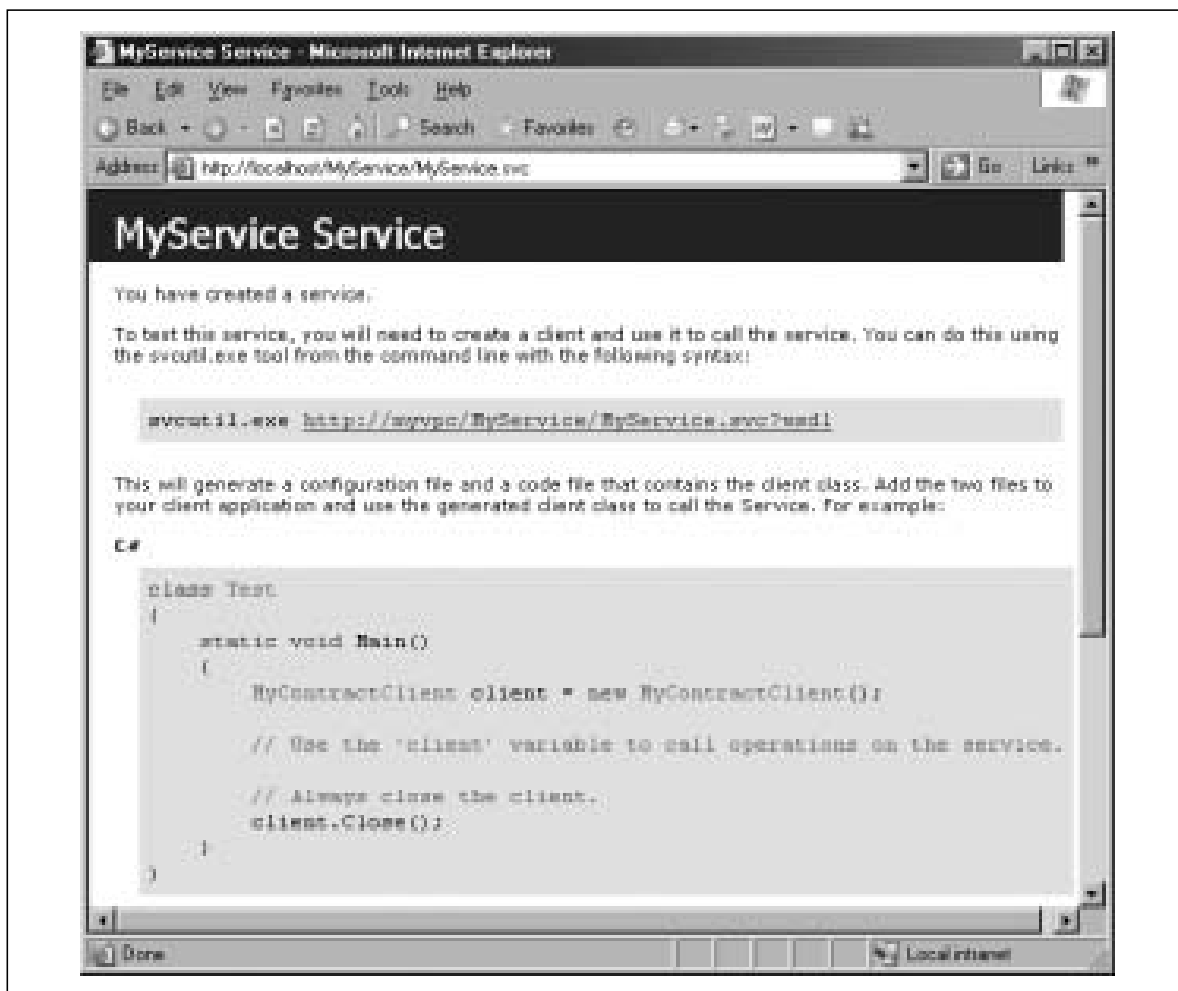


图 1-6: 服务的确认页面

## 编程方式启用元数据交换

若要以编程方式启用基于 HTTP-GET 的元数据交换,首先需要将行为添加到行为集合



中,该行为集合是宿主针对服务类型而维持的。`ServiceHostBase`类定义了 `Description` 属性,类型为 `ServiceDescription`:

```
public abstract class ServiceHostBase : ...
{
    public ServiceDescription Description
    {get;}
    // 更多成员
}
```

顾名思义, `ServiceDescription` 就是对服务各个方面与行为的描述。`ServiceDescription`类定义了类型为 `KeyedByTypeCollection<I>` 的属性 `Behaviors`。其中,类型 `KeyedByTypeCollection<I>` 的泛型参数类型为 `IServiceBehavior` 接口:

```
public class KeyedByTypeCollection<I> : KeyedCollection<Type,I>
{
    public T Find<T>();
    public T Remove<T>();
    // 更多成员
}
public class ServiceDescription
{
    public KeyedByTypeCollection<IServiceBehavior> Behaviors
    {get;}
}
```

所有行为的类与特性均实现了 `IServiceBehavior` 接口。`KeyedByTypeCollection<I>` 定义了泛型方法 `Find<T>()`,它能够返回包含在集合中的请求行为,如果在集合中没有找到,则返回 `null`。查询集合时,最多只能有一个返回的符合条件的行为类型。例 1-11 演示了如何通过编程方式启用行为。

#### 例 1-11: 编程方式启用元数据交换行为

```
ServiceHost host = new ServiceHost(typeof(MyService));

ServiceMetadataBehavior metadataBehavior;
metadataBehavior = host.Description.Behaviors.Find<ServiceMetadataBehavior>();
if(metadataBehavior == null)
{
    metadataBehavior = new ServiceMetadataBehavior();
    metadataBehavior.HttpGetEnabled = true;
    host.Description.Behaviors.Add(metadataBehavior);
}

host.Open();
```

首先,托管代码调用 `KeyedByTypeCollection<I>` 的 `Find<T>()` 方法,它负责判断配置文件是否提供 MEX 终结点行为。`Find<T>` 方法的类型参数为 `ServiceMetadata-`

Behavior 类型。ServiceMetadataBehavior 类定义在 System.ServiceModel.Description 命名空间下：

```
public class ServiceMetadataBehavior : IServiceBehavior
{
    public bool HttpGetEnabled
    {get;set;}
    // 更多成员
}
```

如果返回的行为为 null，托管代码就会创建一个新的 ServiceMetadataBehavior 对象，并将 HttpGetEnabled 属性值设为 true，然后将它添加到服务描述的 behaviors 属性中。

## 元数据交换终结点

元数据交换终结点是一种特殊的终结点，有时候又被称为 MEX 终结点。服务能够根据元数据交换终结点发布自己的元数据。图 1-7 展示了一个具有业务终结点和元数据交换终结点的服务。不过，在通常情况下并不需要在设计图中显示元数据交换终结点。

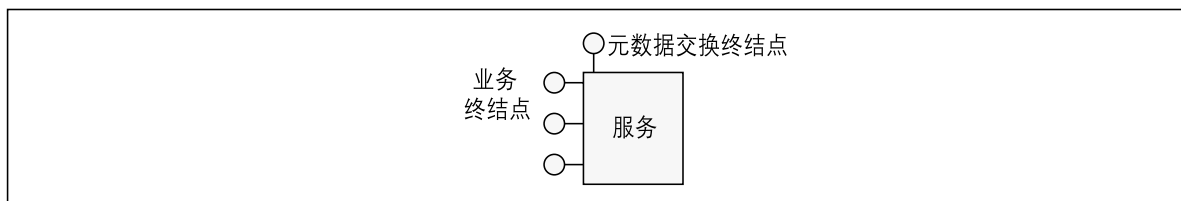


图 1-7：元数据交换终结点

元数据交换终结点支持元数据交换的行业标准，在 WCF 中表现为 IMetadataExchange 接口：

```
[ServiceContract(...)]
public interface IMetadataExchange
{
    [OperationContract(...)]
    Message Get(Message request);
    // 更多成员
}
```

IMetadataExchange 接口定义的细节并不合理。它与多数行业标准相似，都存在难以实现的问题。所幸，WCF 自动地为服务宿主提供了 IMetadataExchange 接口的实现，公开元数据交换终结点。我们只需要指定使用的地址和绑定，以及添加服务元数据行为。对于绑定，WCF 提供了专门的基于 HTTP、HTTPS、TCP 和 IPC 协议的绑定传输元素。对于地址，我们可以提供完整的地址，或者使用任意一个注册了的基地址。没有必要启

用 HTTP-GET 选项，但是即使启用了也不会造成影响。例 1-12 演示的服务公开了三个 MEX 终结点，分别基于 HTTP、TCP 和 IPC。出于演示的目的，TCP 和 IPC 的 MEX 终结点使用了相对地址，HTTP 则使用了绝对地址。

#### 例 1-12: 添加 MEX 终结点

```
<services>
<service name = "MyService" behaviorConfiguration = "MEX">
  <host>
    <baseAddresses>
      <add baseAddress = "net.tcp://localhost:8001/" />
      <add baseAddress = "net.pipe://localhost/" />
    </baseAddresses>
  </host>
  <endpoint
    address = "MEX"
    binding = "mexTcpBinding"
    contract = "IMetadataExchange"
  />
  <endpoint
    address = "MEX"
    binding = "mexNamedPipeBinding"
    contract = "IMetadataExchange"
  />
  <endpoint
    address = "http://localhost:8000/MEX"
    binding = "mexHttpBinding"
    contract = "IMetadataExchange"
  />
</service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name = "MEX">
      <serviceMetadata />
    </behavior>
  </serviceBehaviors>
</behaviors>
```

#### 编程方式添加 MEX 终结点

与其他终结点相似，我们只能在打开宿主之前通过编码方式添加元数据交换终结点。WCF 并没有为元数据交换终结点提供专门的绑定类型。为此，我们需要创建定制绑定。定制绑定使用了与之匹配的传输绑定元素（译注 7），然后将绑定元素对象作为构造函数的参数，传递给定制绑定实例。最后，调用宿主的 `AddServiceEndpoint()` 方法，参数值分别为地址、定制绑定与 `IMetadataExchange` 契约类型。例 1-13 的代码添加了基于 TCP 的 MEX 终结点。注意，在添加终结点之前，必须校验元数据行为是否存在。

---

译注 7：即 `BindingElement` 类型。

## 例 1-13: 编程方式添加 TCP MEX 终结点

```
BindingElement bindingElement = new TcpTransportBindingElement();
CustomBinding binding = new CustomBinding(bindingElement);

Uri tcpBaseAddress = new Uri("net.tcp://localhost:9000/");
ServiceHost host = new ServiceHost(typeof(MyService), tcpBaseAddress);

ServiceMetadataBehavior metadataBehavior;
metadataBehavior = host.Description.Behaviors.Find<ServiceMetadataBehavior>();
if (metadataBehavior == null)
{
    metadataBehavior = new ServiceMetadataBehavior();
    host.Description.Behaviors.Add(metadataBehavior);
}
host.AddServiceEndpoint(typeof(IMetadataExchange), binding, "MEX");
host.Open();
```

简化 `ServiceHost<T>` 类

我们可以扩展 `ServiceHost<T>` 类, 从而自动实现例 1-11 和例 1-13 中的代码。`ServiceHost<T>` 定义了 `Boolean` 型属性 `EnableMetadataExchange`, 通过调用该属性添加 HTTP-GET 元数据行为和 MEX 终结点:

```
public class ServiceHost<T> : ServiceHost
{
    public bool EnableMetadataExchange
    {get;set;}
    public bool HasMexEndpoint
    {get;}
    public void AddAllMexEndPoints();
    // 更多成员
}
```

如果 `EnableMetadataExchange` 属性设置为 `true`, 就会添加元数据交换行为。如果没有可用的 MEX 终结点, 它就会为每个已注册的基地址样式添加一个 MEX 终结点。使用 `ServiceHost<T>`, 例 1-11 和例 1-13 就可以简化为:

```
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.EnableMetadataExchange = true;
host.Open();
```

`ServiceHost<T>` 还定义了 `Boolean` 属性 `HasMexEndpoint`。如果服务包含了任意一个 MEX 终结点 (与传输协议无关), 则返回 `true`。`ServiceHost<T>` 定义的 `AddAllMexEndPoints()` 方法可以为每个已注册的基地址添加一个 MEX 终结点, 这些基地址的样式类型包括 HTTP、TCP 或 IPC。例 1-14 介绍了这些方法的实现。

例 1-14: 实现 EnableMetadataExchange 以及它支持的方法

```
public class ServiceHost<T> : ServiceHost
{
    public bool EnableMetadataExchange
    {
        set
        {
            if(State == CommunicationState.Opened)
            {
                throw new InvalidOperationException("Host is already opened");
            }
            ServiceMetadataBehavior metadataBehavior;
            metadataBehavior = Description.Behaviors.Find<ServiceMetadataBehavior>();
            if(metadataBehavior == null)
            {
                metadataBehavior = new ServiceMetadataBehavior();
                metadataBehavior.HttpGetEnabled = value;
                Description.Behaviors.Add(metadataBehavior);
            }
            if(value == true)
            {
                if(HasMexEndpoint == false)
                {
                    AddAllMexEndPoints();
                }
            }
        }
        get
        {
            ServiceMetadataBehavior metadataBehavior;
            metadataBehavior = Description.Behaviors.Find<ServiceMetadataBehavior>();
            if(metadataBehavior == null)
            {
                return false;
            }
            return metadataBehavior.HttpGetEnabled;
        }
    }
    public bool HasMexEndpoint
    {
        get
        {
            Predicate<ServiceEndpoint> (译注 8) mexEndPoint= delegate(ServiceEndpoint endpoint)
            {
```

译注 8: 例 1-14 中的 Predicate<T> 为 .NET Framework 2.0 新增的类型, 定义为:

```
public delegate bool Predicate<T>(T obj)
```

Predicate 泛型委托可以定义一组条件, 以确定指定对象是否符合这些条件。类型参数 T 为要比较对象的类型, 在例中为 ServiceEndpoint 类型。根据匿名方法的实现, 泛型委托会判断传入的 ServiceEndpoint 的契约类型是否为 IMetadataExchange 接口类型。

```
        return endpoint.Contract.ContractType == typeof(IMetadataExchange);
    };
    return Collection.Exists(Description.Endpoints,mexEndPoint);
}
}
public void AddAllMexEndpoints()
{
    Debug.Assert(HasMexEndpoint == false);

    foreach(Uri baseAddress in BaseAddresses)
    {
        BindingElement bindingElement = null;
        switch(baseAddress.Scheme)
        {
            case "net.tcp":
            {
                bindingElement = new TcpTransportBindingElement();
                break;
            }
            case "net.pipe":
            {...}
            case "http":
            {...}
            case "https":
            {...}
        }
        if(bindingElement != null)
        {
            Binding binding = new CustomBinding(bindingElement);
            AddServiceEndpoint(typeof(IMetadataExchange),binding,"MEX");
        }
    }
}
```

EnableMetadataExchange通过判断CommunicationObject基类的State属性值,确保宿主没有被打开。如果在配置文件中没有找到元数据行为,EnableMetadataExchange不会重写配置文件中的配置值,而只是将value赋给新建的元数据行为对象metadata behavior的HttpGetEnabled属性。读取EnableMetadataExchange的值时,属性首先会检查值是否已经配置。如果没有配置元数据行为,则返回false;否则返回它的HttpGetEnabled值。HasMexEndpoint属性将匿名方法(注1)赋给Predicate泛型委托。匿名方法负责检查给定终结点的契约是否属于IMetadataExchange类型。然后,调用Collection静态类的Exists()方法,方法的参数值为服务宿主中可用的终结点集合。Exists()方法将遍历集合中的每个元素并调用Predicate泛型委托对象

---

注1: 如果你不了解匿名方法,请参见作者在MSDN杂志2004年5月刊中发表的文章《Create Elegant Code with Anonymous Methods, Iterators, and Partial Classes》。



`mexEndPoint`, 如果集合中的任意一个元素符合 `Predicate` 指定的比较条件 (也就是说, 如果匿名方法的调用返回 `true`), 则返回 `true`, 否则返回 `false`。`AddAllMexEndpoints()` 方法会遍历 `BaseAddresses` 集合。根据基地址的样式, 创建匹配的 MEX 传输绑定元素, 然后再创建一个定制绑定, 并将它传入到 `AddServiceEndpoint()` 中, 就像例 1-13 那样添加终结点。

## 元数据浏览器

元数据交换终结点提供的元数据不仅描述了契约与操作, 还包括关于数据契约、安全性、事务性、可靠性以及错误的信息。为了可视化表示正在运行的服务的元数据, 我们开发了元数据浏览器工具, 它的实现包含在本书附带的源代码中。图 1-8 显示了使用元数据浏览器获得的例 1-7 定义的终结点。若要使用元数据浏览器, 只需要提供 HTTP-GET 地址或者正在运行的服务的元数据交换终结点, 就能获取返回的元数据。

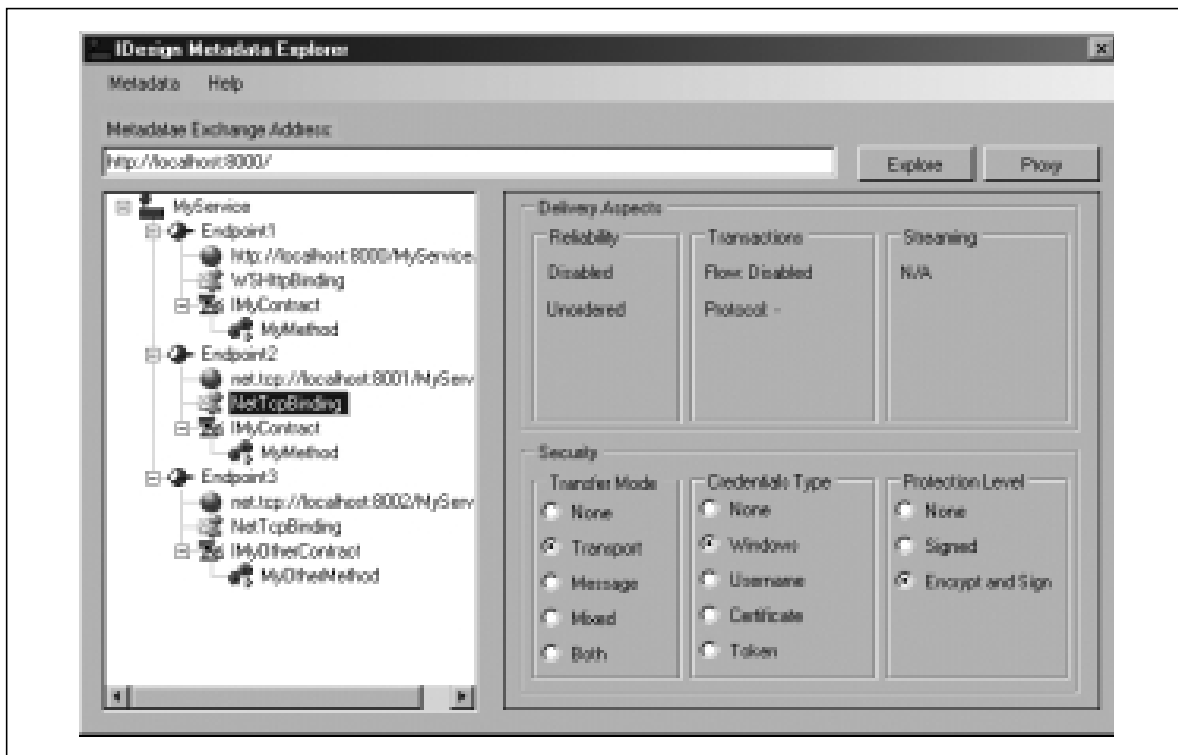


图 1-8: 元数据浏览器

## 客户端编程

若要调用服务的操作, 客户端首先需要导入服务契约到客户端的本地描述 (`Native Representation`) 中。如果客户端使用了 WCF, 调用操作的常见做法是使用代理。代理

是一个 CLR 类，它公开了一个单独的 CLR 接口用以表示服务契约。注意，如果服务支持多个契约（至少是多个终结点），客户端则需要一个代理对应每个契约类型。代理不仅提供了与服务契约相同的操作，同时还包括管理代理对象生命周期的方法，以及管理服务连接的方法。代理完全封装了服务的每个方面：服务的位置、实现技术、运行时平台以及通信传输。

## 生成代理

我们可以使用 Visual Studio 2005 导入服务的元数据，然后生成代理。如果服务是自托管的，则首先需要启动服务，然后从客户端项目的快捷菜单中选择“Add Service Reference...”。如果服务托管在 IIS 或 WAS 中，则无需预先启动服务。值得注意的是，如果服务同时又作为客户端项目自托管在相同解决方案的另一个项目中，则可以在 Visual Studio 2005 中启动宿主，并添加引用。不同于大多数项目的设置，这一选项在调试状态下并没有被禁用（参见图 1-9）。

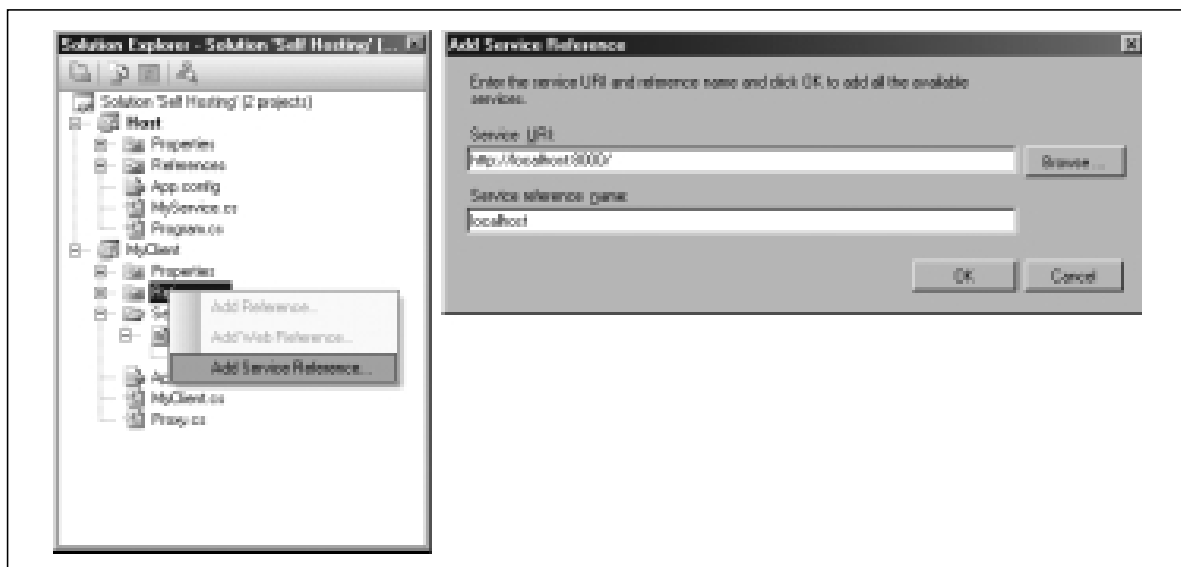


图 1-9：使用 Visual Studio 2005 生成代理

上述操作会弹出 Add Service Reference 对话框，然后开发者需要提供服务的基地址（或者基地址加上一个 MEX 的 URI）以及包含了代理的命名空间。

如果不使用 Visual Studio 2005，也可以使用命令行工具 *SvcUtil.exe*。我们需要为 *SvcUtil* 工具提供 HTTP-GET 地址或者元数据交换终结点地址，而代理文件名则作为可选项。默认的代理文件名为 *output.cs*，但是我们也可以使用 */out* 开关指定不同的名字。

例如，如果服务 *MyService* 托管在 IIS 或 WAS 上，同时拥有可用的基于 HTTP-GET 的元数据共享，则只需要运行下列命令行：

```
SvcUtil http://localhost/MyService/MyService.svc /out:Proxy.cs
```

如果服务的宿主为 IIS, 并且选择的端口号不是 80 (例如端口号 81), 则必须将端口号提供给基地址:

```
SvcUtil http://localhost:81/MyService/MyService.svc /out:Proxy.cs
```

如果是自托管服务, 假定它启用了基于 HTTP-GET 的元数据发布方式, 则可以注册这些基地址, 然后公开包含了一个 MEX 相对地址的与之匹配的元数据交换终结点:

```
http://localhost:8002/  
net.tcp://localhost:8003  
net.pipe://localhost/MyPipe
```

启动宿主后, 可以使用如下命令生成代理:

```
SvcUtil http://localhost:8002/MEX /out:Proxy.cs  
SvcUtil http://localhost:8002/ /out:Proxy.cs  
SvcUtil net.tcp://localhost:8003/MEX /out:Proxy.cs  
SvcUtil net.pipe://localhost/MyPipe/MEX /out:Proxy.cs
```

---

注意: SvcUtil 工具优于 Visual Studio 2005 之处, 在于它提供了大量的选项, 通过开关控制生成的代理, 正如我们在本书后面将要看到的那样。

---

针对服务的定义如下:

```
[ServiceContract(Namespace = "MyNamespace")]  
interface IMyContract  
{  
    [OperationContract]  
    void MyMethod();  
}  
class MyService : IMyContract  
{  
    public void MyMethod()  
    {...}  
}
```

SvcUtil 生成的代理如例 1-15 所示。在大多数情况下, 我们完全可以删除 Action 和 ReplyAction 的设置, 因为默认使用方法名的设置已经足够。

例 1-15: 客户端代理文件

```
[ServiceContract(Namespace = "MyNamespace")]  
public interface IMyContract  
{  
    [OperationContract(Action = "MyNamespace/IMyContract/MyMethod",  
        ReplyAction = "MyNamespace/IMyContract/MyMethodResponse")]
```

```
void MyMethod();  
}  
  
public partial class MyContractClient : ClientBase<IMyContract>,IMyContract  
{  
    public MyContractClient()  
    {}  
    public MyContractClient(string endpointName) : base(endpointName)  
    {}  
    public MyContractClient(Binding binding,EndpointAddress remoteAddress) :  
                                base(binding,remoteAddress)  
    {}  
    /* 其他构造函数 */  
  
    public void MyMethod()  
    {  
        Channel.MyMethod();  
    }  
}
```

代理类的闪光之处在于它可以只包含服务公开的契约,而不需要添加对服务实现类的引用。我们可以通过提供地址和绑定的客户端配置文件使用代理,也可以不通过配置文件直接使用。注意,每个代理的实例都确切地指向了一个终结点。创建代理时需要与终结点交互。正如前文提及,如果服务端契约没有提供命名空间,则默认的命名空间为 *http://tempuri.org*。

## 管理方式配置客户端

客户端需要知道服务的位置,需要使用与服务相同的绑定,当然,客户端还要导入服务契约的定义。本质上讲,它的信息与从服务的终结点获取的信息完全相同。为了体现这些信息,客户端配置文件需要包含目标终结点的信息,甚至使用与宿主完全相同的终结点配置样式。

例1-16演示了与一个服务交互时必需的客户端配置文件,其中,服务宿主的配置参见例1-6。

### 例 1-16: 客户端配置文件

```
<system.serviceModel>  
  <client>  
    <endpoint name = "MyEndpoint"  
      address = "http://localhost:8000/MyService/"  
      binding = "wsHttpBinding"  
      contract = "IMyContract"  
    />  
  </client>  
</system.serviceModel>
```

客户端配置文件可以列出同样多的对应服务支持的终结点,客户端能够使用这些终结点中的任意一个。例 1-17 展示的客户端配置文件,与例 1-7 中的宿主配置文件相匹配。注意,客户端配置文件中的每个终结点都有一个唯一的名称。

例 1-17: 包含了多个目标终结点的客户端配置文件

```
<system.serviceModel>
  <client>
    <endpoint name = "FirstEndpoint"
      address = "http://localhost:8000/MyService/"
      binding = "wsHttpBinding"
      contract = "IMyContract"
    />
    <endpoint name = "SecondEndpoint"
      address = "net.tcp://localhost:8001/MyService/"
      binding = "netTcpBinding"
      contract = "IMyContract"
    />
    <endpoint name = "ThirdEndpoint"
      address = "net.tcp://localhost:8002/MyService/"
      binding = "netTcpBinding"
      contract = "IMyOtherContract"
    />
  </client>
</system.serviceModel>
```

## 绑定配置

我们可以使用与服务配置相同的风格定制匹配服务绑定的客户端标准绑定,如例 1-18 所示。

例 1-18: 客户端绑定配置

```
<system.serviceModel>
  <client>
    <endpoint name = "MyEndpoint"
      address = "net.tcp://localhost:8000/MyService/"
      bindingConfiguration = "TransactionalTCP"
      binding = "netTcpBinding"
      contract = "IMyContract"
    />
  </client>
  <bindings>
    <netTcpBinding>
      <binding name = "TransactionalTCP"
        transactionFlow = "true"
      />
    </netTcpBinding>
  </bindings>
</system.serviceModel>
```

## 生成客户端配置文件

在默认情况下, `SvcUtil` 也可以自动生成客户端配置文件 `output.config`。同时, 可以使用 `/config` 开关指定配置文件名:

```
SvcUtil http://localhost:8002/MyService/ /out:Proxy.cs /config:App.Config
```

也可以使用 `/noconfig` 开关生成精简的配置文件:

```
SvcUtil http://localhost:8002/MyService/ /out:Proxy.cs /noconfig
```

建议永远不要使用 `SvcUtil` 工具生成配置文件。原因在于它会自动地为关键的绑定节设置默认值, 反而导致了整个配置文件的混乱。

## 进程内托管配置

对于进程内托管, 客户端配置文件就是服务宿主的配置文件。同一个文件既包含了服务配置入口, 也包含了客户端的配置入口, 如例 1-19 所示。

例 1-19: 进程内托管的配置文件

```
<system.serviceModel>
  <services>
    <service name = "MyService">
      <endpoint
        address = "net.pipe://localhost/MyPipe"
        binding = "netNamedPipeBinding"
        contract = "IMyContract"
      />
    </service>
  </services>
  <client>
    <endpoint name = "MyEndpoint"
      address = "net.pipe://localhost/MyPipe"
      binding = "netNamedPipeBinding"
      contract = "IMyContract"
    />
  </client>
</system.serviceModel>
```

注意, 进程内宿主使用了命名管道绑定 (译注 9)。

## SvcConfigEditor 编辑器

WCF 提供了配置文件的编辑器 `SvcConfigEditor.exe`, 它既能编辑宿主配置文件, 又能编

---

译注 9: 对于进程内托管而言, 由于服务与客户端处于相同的机器上, 因而只能使用命名管道绑定。



辑客户端配置文件（参见图 1-10）。启动编辑器的方法是在 Visual Studio 中，右键单击配置文件（客户端和宿主文件），然后选择“Edit WCF Configuration”。

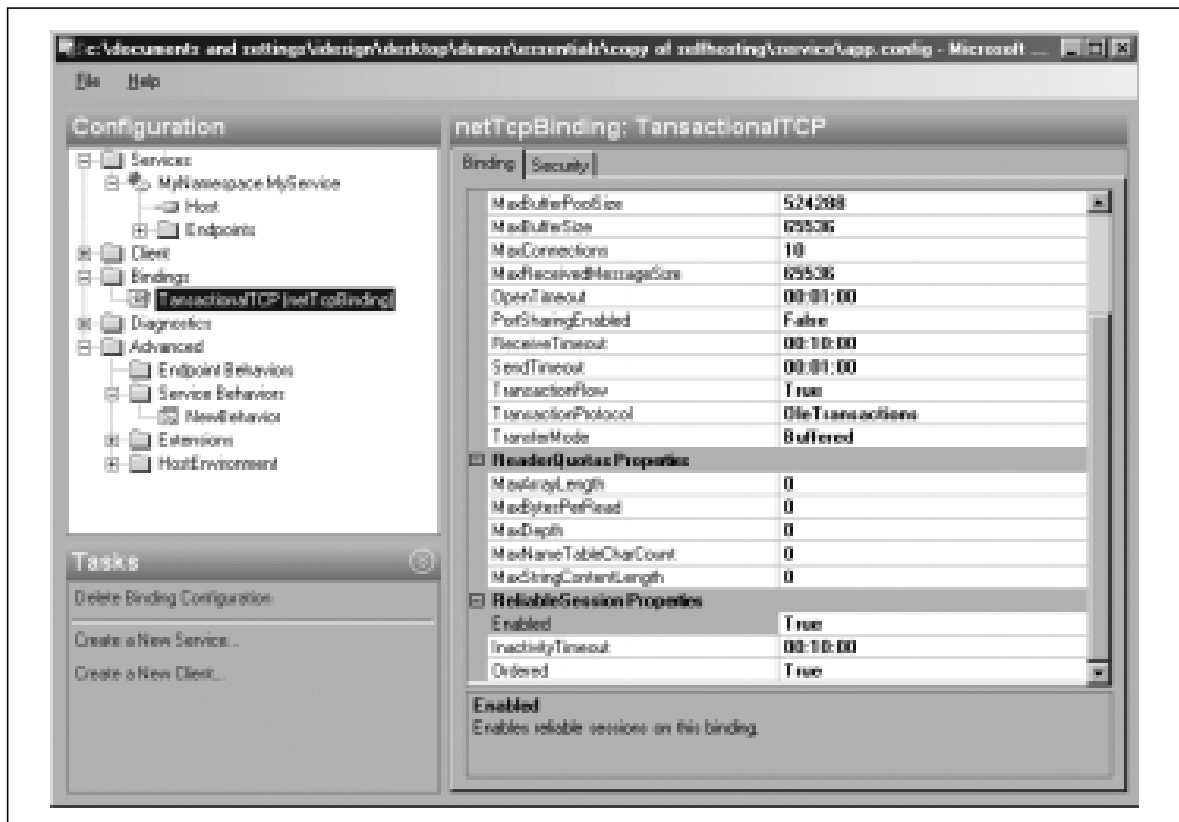


图 1-10：用于编辑宿主与客户端配置文件的 SvcConfigEditor

对于使用 SvcConfigEditor，优劣参半。一方面，它可以帮助开发者轻松快捷地编辑配置文件，从而节约了掌握配置样式的时间。另一方面，它却不利于开发者对 WCF 配置的整体理解。多数情况下，采用手工方式对配置文件进行细微的修改，要比使用 Visual Studio 2005 更加快速。

## 创建和使用代理

代理类派生自 ClientBase<T> 类，定义如下：

```
public abstract class ClientBase<T> : ICommunicationObject, IDisposable
{
    protected ClientBase(string endpointName);
    protected ClientBase(Binding binding, EndpointAddress remoteAddress);
    public void Open();
    public void Close();
    protected T Channel
    {get;}
}
```

```
// 其他成员  
}
```

`ClientBase<T>` 类通过泛型类型参数识别代理封装的服务契约。`ClientBase<T>` 的 `Channel` 属性类型就是泛型参数的类型。`ClientBase<T>` 的子类通过 `Channel` (译注 10) 调用它指向的服务契约的方法 (参见例 1-15)。

若要使用代理, 客户端首先需要实例化代理对象, 并为构造函数提供终结点信息, 即配置文件中的终结点名称。如果没有使用配置文件, 则为终结点地址和绑定对象。然后, 客户端使用代理类的方法调用服务。一旦客户端调用完毕, 就会关闭代理实例。以例 1-15 和例 1-16 的定义为例, 客户端创建代理, 然后通过配置文件识别使用的终结点, 再调用代理的方法, 最后关闭代理:

```
MyContractClient proxy = new MyContractClient("MyEndpoint");  
proxy.MyMethod();  
proxy.Close();
```

如果在客户端配置文件中, 只为代理正在使用的契约类型定义了一个终结点, 则客户端可以省略构造函数中的终结点名称:

```
MyContractClient proxy = new MyContractClient();  
proxy.MyMethod();  
proxy.Close();
```

然而, 如果相同的契约类型包含了多个可用的终结点, 则代理会抛出异常。

## 关闭代理

最佳的做法是在客户端调用代理完毕之后要关闭代理。第 4 章会详细解释为何在正确情况下客户端需要关闭代理, 因为关闭代理会终止与服务的会话, 关闭连接。

使用代理的 `Dispose()` 方法同样可以关闭代理。这种方式的优势在于它支持 `using` 语句的使用, 即使出现异常, 仍然能够调用:

```
using(MyContractClient proxy = new MyContractClient())  
{  
    proxy.MyMethod();  
}
```

如果客户端直接声明了契约, 而不是具体的代理类, 则客户端可以首先判断代理对象是否实现了 `IDisposable` 接口:

---

译注 10: 这里的 `Channel` 对象就相当于服务契约的代理对象, 通过它, 就可以调用服务契约的相关方法。在 WCF 的早期版本如 *Indigo* 中, 就直接使用了 `proxy` 对象。

```
IMyContract proxy = new MyContractClient();
proxy.MyMethod();
IDisposable disposable = proxy as IDisposable;
if(disposable != null)
{
    disposable.Dispose();
}
```

或者使用 using 语句，省略对类型的判断：

```
IMyContract proxy = new MyContractClient();
using(proxy as IDisposable)
{
    proxy.MyMethod();
}
```

### 调用超时

WCF 客户端的每次调用都必须在配置的超时值内完成。无论何种原因，一旦调用时间超出该时限，调用就会被取消，客户端会收到一个 `TimeoutException` 异常。绑定的一个属性用于设定超时的确切值，默认的超时值为 `1min`。若要设置不同的超时值，可以设置 `Binding` 抽象基类的 `SendTimeout` 属性：

```
public abstract class Binding : ...
{
    public TimeSpan SendTimeout
    {get;set;}
    // 更多成员
}
```

例如，使用 `WSHttpBinding` 时：

```
<client>
  <endpoint
    ...
    binding = "wsHttpBinding"
    bindingConfiguration = "LongTimeout"
    ...
  />
</client>
<bindings>
  <wsHttpBinding>
    <binding name = "LongTimeout" sendTimeout = "00:05:00"/>
  </wsHttpBinding>
</bindings>
```

### 编程方式配置客户端

如果不借助于配置文件，客户端也可以通过编程方式创建匹配服务终结点的地址与绑定对象，并将它们传递给代理类的构造函数。既然代理的泛型类型参数就是契约，因此不

必为构造函数提供契约。为了表示地址，客户端需要实例化 `EndpointAddress` 类，定义如下：

```
public class EndpointAddress
{
    public EndpointAddress(string uri);
    // 更多成员
}
```

例 1-20 演示了编程方式配置客户端的技术，所示代码的功能与例 1-16 等价，它们使用的目标服务则为例 1-9 的定义。

#### 例 1-20：编程方式配置客户端

```
Binding wsBinding = new WSHttpBinding();
EndpointAddress endpointAddress = new
    EndpointAddress("http://localhost:8000/MyService/");

MyContractClient proxy = new MyContractClient(wsBinding, endpointAddress);

proxy.MyMethod();
proxy.Close();
```

与在配置文件中使用绑定节的方法相似，客户端可以通过编程方式配置绑定属性：

```
WSHttpBinding wsBinding = new WSHttpBinding();
wsBinding.SendTimeout = TimeSpan.FromMinutes(5);
wsBinding.TransactionFlow = true;

EndpointAddress endpointAddress = new
    EndpointAddress("http://localhost:8000/MyService/");

MyContractClient proxy = new MyContractClient(wsBinding, endpointAddress);
proxy.MyMethod();
proxy.Close();
```

注意，使用 `Binding` 类的具体子类，是为了访问与绑定相关的属性，例如事务流。

## 编程方式配置与管理方式配置

目前介绍的配置客户端与服务的技术各有所长，相辅相成。管理配置方式允许开发者在部署服务之后，修改服务与客户端的主要特性，而不需要重新编译或重新部署。主要缺陷则是不具备类型安全，只有在运行时才能发现配置的错误。

如果配置的决策完全是动态的，那么编程配置方式就体现了它的价值，它可以在运行时基于当前的输入或条件对服务的配置进行处理。如果判断条件是静态的，而且是恒定不变的，就可以采取硬编码方式。例如，如果我们只关注于进程内托管的调用，就可以采

取硬编码方式,使用`NetNamePipeBinding`以及它的配置(译注11)。不过,大体而言,大多数客户端和服务都会使用配置文件。

## WCF 体系架构

本章内容全面地介绍了建立和使用简单的 WCF 服务所需要的知识。然而,正如本书其余章节将要描述的那样,WCF 提供了对可靠性、事务性、并发管理、安全性以及实例激活等技术的有力支持,它们均依赖于基于拦截机制的 WCF 体系架构(WCF Architecture)。通过代理与客户端的交互意味着 WCF 总是处于服务与客户端之间,拦截所有的调用,执行调用前和调用后的处理。当代理将调用栈帧(Stack Frame)序列化到消息中,并将消息通过通道链向下传递时,WCF 就开始执行拦截。通道相当于一个拦截器,目的在于执行一个特定的任务。每个客户端通道都会执行消息的调用前处理。链的组成与结构主要依赖于绑定。例如,一个通道对消息编码(二进制格式、文本格式或者 MTOM),另一个通道传递安全的调用上下文;还有一个通道传播客户端的事务,一个通道管理可靠会话,另一个通道对消息正文(Message Body)加密(如果进行了配置),诸如此类。客户端的最后一个通道是传输通道,根据配置的传输方式发送消息给宿主。

在宿主端,消息同样通过通道链进行传输,它会对消息执行宿主端的调用前处理。宿主端的第一个通道是传输通道,接收传输过来的消息。随后的通道执行不同的任务,例如消息正文的解密、消息的解码、参与传播事务、设置安全准则、管理会话、激活服务实例。宿主端的最后一个通道负责将消息传递给分发器(Dispatcher)。分发器将消息转换到一个栈帧,并调用服务实例。执行顺序如图 1-11 所示。

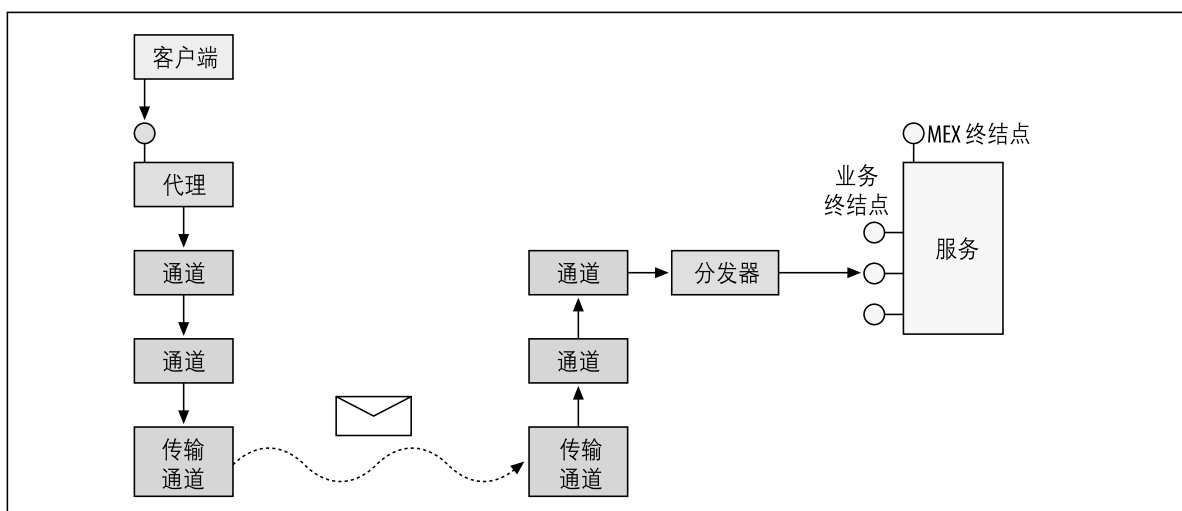


图 1-11: WCF 体系架构

译注 11: 由于进程内托管只能使用 `NetNamedPipeBinding`, 因此, 在部署服务之后不会出现修改绑定类型的情况, 此时采用硬编码方式, 不会造成服务配置的僵化。

服务并不知道它是否被本地客户端调用。事实上, 服务会被本地客户端——分发器调用。客户端与服务端的拦截器确保了它们能够获得运行时环境, 以便于它们执行正确的操作。服务实例会执行调用, 然后将控制权 (**Control**) 返回给分发器。分发器负责将返回值以及错误信息 (如果存在) 转换为一条返回消息。分发器获得控制权, 执行的过程则刚好相反: 分发器通过宿主端通道传递消息, 执行调用后的处理, 例如管理事务、停用实例、回复消息的编码与加密等。为了执行客户端调用后的处理, 包括解密、解码、提交或取消事务等任务, 传输通道会将返回消息发送到客户端通道。最后一个通道将消息传递给代理。代理将返回消息转化到栈帧, 然后将控制权返回给客户端。

特别值得注意的是, 体系架构中的所有要点均与可扩展性息息相关。我们可以为专有交互定制通道, 为实例管理定制行为, 以及定制安全行为等。事实上, **WCF** 提供的标准功能都能够通过相同的可扩展模式实现。本书介绍了许多针对可扩展性的实例与应用。

## 宿主体系架构

如何将与技术无关的面向服务交互转换为 **CLR** 接口与类, 对这一技术的探索无疑充满了趣味。宿主消除了两者之间的鸿沟, 搭建了相互之间转换的桥梁。每个 **.NET** 宿主进程都包含了多个应用程序域。每个应用程序域则包含了零到多个宿主实例。每个服务宿主实例专门对应于一个特殊的服务类型。创建一个宿主实例, 实际上就是为对应于基地址的宿主机器的类型, 注册一个包含了所有的终结点的服务宿主实例。每个服务宿主实例拥有零到多个上下文 (**Context**)。上下文是服务实例最核心的执行范围。一个上下文最多只能与一个服务实例关联, 这意味着上下文可能为空, 不包含任何服务实例。体系架构如图 1-12 所示。

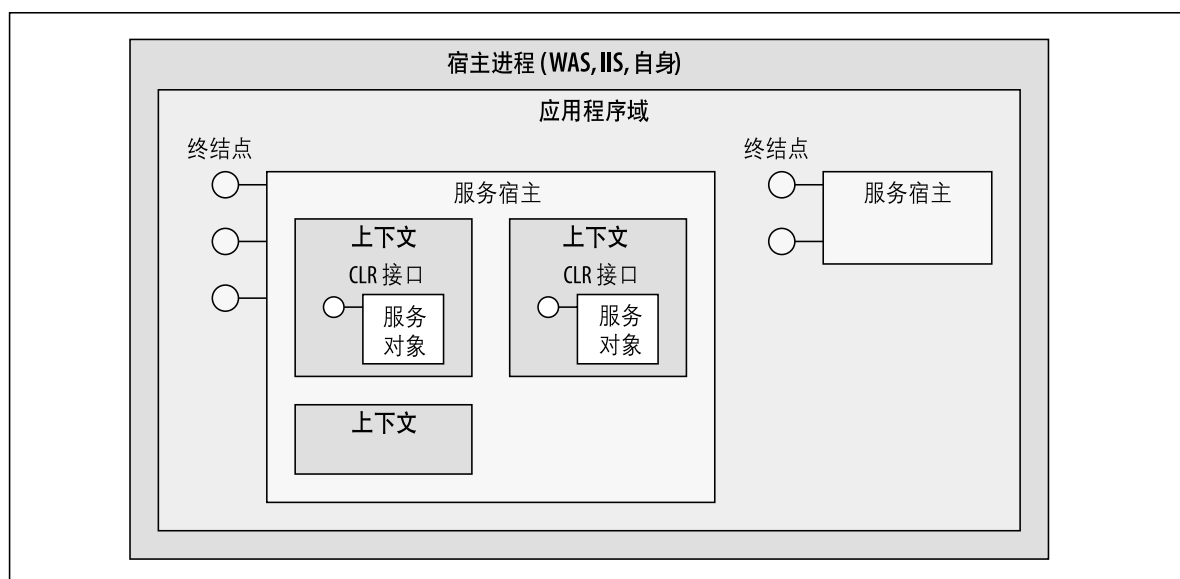


图 1-12: WCF 宿主体系架构



---

注意：WCF 上下文的概念与企业服务上下文（Enterprise Services Context）或者 .NET 上下文绑定对象（Context-Bound Object）的上下文概念相似。

---

WCF 上下文将服务宿主与公开本地 CLR 类型为服务的上下文组合在一起。当消息经由通道进行传递时，宿主会将消息映射到新的或者已有的上下文（内部包含对象实例），然后通过上下文处理调用。

## 使用通道

我们可以直接使用通道调用服务的操作，而无须借助于代理类。ChannelFactory<T> 类（以及它所支持的类型）有助于我们轻松地创建代理，如例 1-21 所示。

例 1-21: ChannelFactory<T> 类

```
public class ContractDescription
{
    public Type ContractType
    {get;set;}
    // 更多成员
}

public class ServiceEndpoint
{
    public ServiceEndpoint(ContractDescription contract, Binding binding,
                           EndpointAddress address);

    public EndpointAddress Address
    {get;set;}
    public Binding Binding
    {get;set;}
    public ContractDescription Contract
    {get;}
    // 更多成员
}

public abstract class ChannelFactory : ...
{
    public ServiceEndpoint Endpoint
    {get;}
    // 更多成员
}

public class ChannelFactory<T> : ChannelFactory, ...
{
    public ChannelFactory(ServiceEndpoint endpoint);
    public ChannelFactory(string configurationName);
    public ChannelFactory(Binding binding, EndpointAddress endpointAddress);
    public static T CreateChannel(Binding binding, EndpointAddress endpointAddress);
    public T CreateChannel();
}
```

```
// 更多成员  
}
```

我们需要向 `ChannelFactory<T>` 类的构造函数传递一个终结点对象, 终结点名称可以从客户端配置文件中获取; 或者传递绑定与地址对象, 或者传递 `ServiceEndpoint` 对象。接着, 调用 `CreateChannel()` 方法获得代理的引用, 然后使用代理的方法。最后, 通过将代理强制转换为 `IDisposable` 类型, 调用 `Dispose()` 方法关闭代理。当然, 也可以将代理强制转换为 `ICommunicationObject` 类型, 通过调用 `Close()` 方法关闭代理:

```
ChannelFactory<IMyContract> factory = new ChannelFactory<IMyContract>();  
  
IMyContract proxy1 = factory.CreateChannel();  
using(proxy1 as IDisposable)  
{  
    proxy1.MyMethod();  
}  
  
IMyContract proxy2 = factory.CreateChannel();  
proxy2.MyMethod();  
ICommunicationObject channel = proxy2 as ICommunicationObject;  
Debug.Assert(channel != null);  
channel.Close();
```

我们还可以直接调用 `CreateChannel()` 静态方法, 根据给定的绑定和地址值创建代理, 这样就不需要创建 `ChannelFactory<T>` 类的实例了。

```
Binding binding = new NetTcpBinding();  
EndpointAddress address = new EndpointAddress("net.tcp://localhost:8000");  
  
IMyContract proxy = ChannelFactory<IMyContract>.CreateChannel(binding, address);  
using(proxy as IDisposable)  
{  
    proxy1.MyMethod();  
}
```

## InProcFactory 类

为了演示 `ChannelFactory<T>` 类的强大功能, 考虑静态辅助类 `InProcFactory`, 定义如下:

```
public static class InProcFactory  
{  
    public static I CreateInstance<S,I>() where I : class  
                                   where S : I;  
    public static void CloseProxy<I>(I instance) where I : class;  
    // 更多成员  
}
```

设计 `InProcFactory` 类的目的在于简化并自动实现进程内托管。`CreateInstance()` 方法定义了两个泛型类型参数：服务类型 `S` 以及服务支持的契约类型 `I`。`CreateInstance()` 要求 `S` 必须派生自 `I`。可以直接使用 `InProcFactory` 类：

```
IMyContract proxy = InProcFactory.CreateInstance<MyService, IMyContract>();  
proxy.MyMethod();  
InProcFactory.CloseProxy(proxy);
```

`InProcFactory` 能够接收一个服务类，并将它升级为 WCF 服务。这就好比我们可以在 WCF 中调用旧的 Win32 函数 `LoadLibrary()`。

### `InProcFactory<T>` 的实现

所有的进程内调用都应该使用命名管道，同时还应该传递所有事务。我们可以使用编程配置自动地配置客户端和服务，同时使用 `ChannelFactory<T>`，以避免对代理对象的使用。例 1-22 演示了 `InProcFactory` 的实现。为简便起见，省略了一部分与此无关的代码。

#### 例 1-22: `InProcFactory` 类

```
public static class InProcFactory  
{  
    struct HostRecord  
    {  
        public HostRecord(ServiceHost host, string address)  
        {  
            Host = host;  
            Address = address;  
        }  
        public readonly ServiceHost Host;  
        public readonly string Address;  
    }  
    static readonly Uri BaseAddress = new Uri("net.pipe://localhost/" + Guid.NewGuid().ToString());  
    static readonly Binding NamedPipeBinding;  
    static Dictionary<Type, HostRecord> m_Hosts = new Dictionary<Type, HostRecord>();  
  
    static InProcFactory()  
    {  
        NetNamedPipeBinding binding = new NetNamedPipeBinding();  
        binding.TransactionFlow = true;  
        NamedPipeBinding = binding;  
        AppDomain.CurrentDomain.ProcessExit += delegate  
        {  
            foreach(KeyValuePair<Type, HostRecord> pair in m_Hosts)  
            {  
                pair.Value.Host.Close();  
            }  
        };  
    }  
}
```

```
}
public static I CreateInstance<S,I>() where I : class
    where S : I
{
    HostRecord hostRecord = GetHostRecord<S,I>();
    return ChannelFactory<I>.CreateChannel(NamedPipeBinding,
                                           new EndpointAddress(hostRecord.Address));
}
static HostRecord GetHostRecord<S,I>() where I : class
    where S : I
{
    HostRecord hostRecord;
    if(m_Hosts.ContainsKey(typeof(S)))
    {
        hostRecord = m_Hosts[typeof(S)];
    }
    else
    {
        ServiceHost host = new ServiceHost(typeof(S), BaseAddress);
        string address = BaseAddress.ToString() + Guid.NewGuid().ToString();
        hostRecord = new HostRecord(host, address);
        m_Hosts.Add(typeof(S), hostRecord);
        host.AddServiceEndpoint(typeof(I), NamedPipeBinding, address);
        host.Open();
    }
    return hostRecord;
}
public static void CloseProxy<I>(I instance) where I : class
{
    ICommunicationObject proxy = instance as ICommunicationObject;
    Debug.Assert(proxy != null);
    proxy.Close();
}
}
```

实现 `InProcFactory` 所面临的重大挑战, 就是如何实现 `CreateInstance()` 方法, 使得它能够为每个类型创建服务实例。由于每个服务类型都必须有一个对应的宿主 (`ServiceHost` 的一个实例), 如果为每次调用都分配一个宿主实例, 算不上是一个好的办法。问题在于, 如果需要为相同的类型实例化第二个对象, `CreateInstance()` 方法应该怎么做:

```
IMyContract proxy1 = InProcFactory.CreateInstance<MyService,IMyContract>();
IMyContract proxy2 = InProcFactory.CreateInstance<MyService,IMyContract>();
```

解决办法是在内部管理一个字典 (`Dictionary`) 集合, 以建立服务类型与特定的宿主实例之间的映射。调用 `CreateInstance()` 方法创建指定类型的实例时, 通过调用辅助方法 `GetHostRecord()` 查找字典集合, 只有不存在该服务类型, 才会创建宿主实例。如果需要创建宿主, `GetHostRecord()` 方法会以编程方式为宿主添加一个终结点, 并生成 GUID 作为唯一标识的管道名。`GetHostRecord()` 方法会返回一个 `HostRecord` 对

象。HostRecord 是一个结构类型，定义了 ServiceHost 实例与地址值。然后，CreateInstance() 根据 HostRecord 对象获得终结点地址，调用 ChannelFactory<T> 类的方法创建代理。在 InProcFactory 类的静态构造函数（静态构造函数会在初始化类时调用）中，InProcFactory 订阅了 ProcessExit 事件，使用了匿名方法在停止进程时关闭所有宿主。最后，为了方便客户端关闭代理，InProcFactory 定义了 CloseProxy() 方法，它将代理强制转换为 ICommunicationObject 类型以关闭代理。

## 可靠性

WCF 与其他面向服务技术之间最大的区别在于传输可靠性（Transport Reliability）与消息可靠性（Message Reliability）。传输可靠性（例如通过 TCP 传输）在网络数据包层提供了点对点保证传递（Point-to-Point Guaranteed Delivery），以确保数据包的顺序无误。传输可靠性不会受到网络连接的中断或其他通信问题的影响。

顾名思义，消息可靠性负责处理消息层的可靠性，它与传递消息的数据包数量无关。消息可靠性提供了端对端保证传递（End-to-End Guaranteed Delivery），确保消息的顺序无误。消息可靠性与引入的中间方的数量无关，与网络跳数（Network Hops）的数量也没有关联（译注 12）。消息可靠性基于一个行业标准。该行业标准为可靠的基于消息的通信维持了一个在传输层的会话。如果传输失败，例如无线连接中断，消息可靠性就会提供重试（Retries）功能。它还能够自动处理网络阻塞（Congestion）、消息缓存（Message Buffering）以及流控制（Flow Control），根据具体情况适时调整发送的消息数。消息可靠性还能够通过对连接的验证管理连接自身，并在不需要连接时清除它们。

## 绑定与可靠性

WCF 的可靠性是在绑定中控制与配置的。一个特定的绑定可以支持可靠消息传输（Reliable Messaging），也可以不支持它。如果支持，也可以通过设置为启用或禁用。何种绑定支持何种可靠性值，要根据绑定的目标场景而定。表 1-2 总结了绑定、可靠性、有序传递（Ordered Delivery）以及它们各自的默认值之间的关系。

表 1-2：可靠性与绑定

名称	支持可靠性	默认可靠性	支持有序传递	默认有序传递
BasicHttpBinding	No	N/A	No	N/A
NetTcpBinding	Yes	Off	Yes	On
NetPeerTcpBinding	No	N/A	No	N/A

译注 12：网络跳数是客户端传递消息到服务端所必须的。

表 1-2: 可靠性与绑定 (续)

名称	支持可靠性	默认可靠性	支持有序传递	默认有序传递
NetNamedPipeBinding	No	N/A (On)	Yes	N/A (On)
WSHttpBinding	Yes	Off	Yes	On
WSFederationHttpBinding	Yes	Off	Yes	On
WSDualHttpBinding	Yes	On	Yes	On
NetMsmqBinding	No	N/A	No	N/A
MsmqIntegrationBinding	No	N/A	No	N/A

BasicHttpBinding、NetPeerTcpBinding以及两种MSMQ绑定(NetMsmqBinding和MsmqIntegrationBinding)不支持可靠性。因为BasicHttpBinding面向旧的ASMX Web服务,是不具有可靠性的。NetPeerTcpBinding则为广播场景设计。MSMQ绑定针对断开调用,在任何情况下都不存在传输会话。

WSDualHttpBinding总是支持可靠性的,它能够保持回调通道,确保基于HTTP协议的客户端存在。

NetTcpBinding绑定以及各种WS绑定,默认情况下并不支持可靠性,但是允许启用对它的支持。由于NetNamedPipeBinding绑定总是拥有一个确定的从客户端到服务的跳数,因而它的可靠性是绑定固有的。

## 有序消息

消息可靠性确保了消息的有序传递,允许消息按照发送顺序而非接收顺序执行。此外,它保证了消息只会被传递一次。

WCF允许开发者只启用可靠性,而不启用有序传递。此时,消息按照接收它们的顺序进行传递。如果同时启用了可靠性与有序传递,则所有绑定的默认值均支持可靠性。

## 配置可靠性

通过编程方式或管理方式都可以配置可靠性(以及有序传递)。如果我们启用了可靠性,则客户端与服务宿主端必须保持一致,否则客户端无法与服务通信。我们可以只对支持它的绑定配置可靠性。例 1-23 所示的服务端配置文件,使用了绑定配置节,启用了TCP绑定的可靠性。



## 例 1-23: 启用 TCP 绑定的可靠性

```
<system.serviceModel>
  <services>
    <service name = "MyService">
      <endpoint
        address = "net.tcp://localhost:8000/MyService"
        binding = "netTcpBinding"
        bindingConfiguration = "ReliableTCP"
        contract = "IMyContract"
      />
    </service>
  </services>
  <bindings>
    <netTcpBinding>
      <binding name = "ReliableTCP">
        <reliableSession enabled = "true"/>
      </binding>
    </netTcpBinding>
  </bindings>
</system.serviceModel>
```

至于编程配置方式, TCP 绑定和 WS 绑定提供了略微不同的属性来配置可靠性。例如, NetTcpBinding 绑定接受一个 Boolean 型的构造函数参数, 用来启动可靠性:

```
public class NetTcpBinding : Binding,...
{
    public NetTcpBinding(...,bool reliableSessionEnabled);
    // 更多成员
}
```

我们只能在对象的构造期间启用可靠性。如果通过编程方式设置可靠性, 需要创建支持可靠性的绑定对象:

```
Binding reliableTcpBinding = new NetTcpBinding(...,true);
```

NetTcpBinding 定义了只读 (译注 13) 的 ReliableSession 类, 通过它获取可靠性的状态:

```
public class ReliableSession
{
    public TimeSpan InactivityTimeout
    {get;set;}
    public bool Ordered
    {get;set;}
    // 更多成员
}
public class OptionalReliableSession : ReliableSession
{
}
```

---

译注 13: 在 NetTcpBinding 类的定义中, 将 ReliableSession 属性定义为只有 get 访问器。

```
        public bool Enabled
        {get;set;}
        // 更多成员
    }
    public class NetTcpBinding : Binding,...
    {
        public OptionalReliableSession ReliableSession
        {get;}
        // 更多成员
    }
```

## 必备有序传递

理论上, 服务代码和契约定义应该与它使用的绑定及属性无关。服务不应该考虑绑定, 在服务代码中也不应该包含它所使用的绑定。不管配置的绑定是哪一种, 服务都应该能够正常工作。然而实际上, 服务的实现或者契约本身都会依赖于消息的有序传递 (Ordered Delivery)。为了帮助契约或服务的开发者能够约束支持的绑定, WCF 定义了 `DeliveryRequirementsAttribute` 特性:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface
                AllowMultiple = true)]
public sealed class DeliveryRequirementsAttribute : Attribute,...
{
    public Type TargetContract
    {get;set;}
    public bool RequireOrderedDelivery
    {get;set;}

    // 更多成员
}
```

`DeliveryRequirements` 特性可以应用到服务一级, 对服务的所有终结点施加影响, 或者只对公开了特定契约的终结点施加影响; 如果应用到服务一级, 则意味着选用有序传递是根据具体实现作出的决策。`DeliveryRequirements` 特性也可以应用到契约一级, 它会对所有支持该契约的服务施加影响。这样一种在契约一级的应用, 体现了对有序传递的要求是根据设计作出的决策。这一约束会在装载服务时得到执行与验证。如果一个终结点包含的绑定并不支持可靠性; 或者支持可靠性, 却被禁用了; 或者虽然启用了可靠性, 但却禁用了有序传递, 那么装载服务就会失败, 抛出 `InvalidOperationException` 异常。

---

注意: 命名管道绑定符合有序传递的约束。

---

举例来说, 如果不考虑契约, 要求服务的所有终结点都启用有序传递, 则可以将 `DeliveryRequirements` 特性直接应用到服务类上:

```
[DeliveryRequirements(RequireOrderedDelivery = true)]  
class MyService : IMyContract,IMyOtherContract  
{...}
```

通过设置TargetContract属性,只有支持目标契约的服务终结点才需要遵循可靠的有序传递的约束:

```
[DeliveryRequirements(TargetContract = typeof(IMyContract),  
    RequireOrderedDelivery = true)]  
class MyService : IMyContract,IMyOtherContract  
{...}
```

如果将DeliveryRequirements特性应用到契约接口上,则支持该契约的所有服务都必须遵循这一约束:

```
[DeliveryRequirements(RequireOrderedDelivery = true)]  
[ServiceContract]  
interface IMyContract  
{...}  
  
class MyService : IMyContract  
{...}  
  
class MyOtherService : IMyContract  
{...}
```

RequireOrderedDelivery的默认值为false,如果只应用了DeliveryRequirements特性,没有设置RequireOrderedDelivery的值,则是无效的。例如,如下语句是等效的:

```
[ServiceContract]  
interface IMyContract  
{...}  
  
[DeliveryRequirements]  
[ServiceContract]  
interface IMyContract  
{...}  
  
[DeliveryRequirements(RequireOrderedDelivery = false)]  
[ServiceContract]  
interface IMyContract  
{...}
```

## 第2章

# 服务契约

通过前一章的介绍，我们知道 `ServiceContract` 特性能够将接口（或者类）公开为面向服务的契约，允许开发者使用诸如 C# 语言进行编程，把类似于接口这样的语法结构公开为 WCF 契约和服务。本章首先会讨论如何通过操作重载与契约层级，为两种迥然不同的编程模型建立关联。然后会介绍一些简单而又强大的设计和分离服务契约的技术与指导原则。在本章末尾介绍了如何通过编程方式在运行时实现与契约元数据的交互。

## 操作重载

诸如 C++ 和 C# 等编程语言都支持方法重载，即允许具有相同名称的两个方法可以定义不同的参数。例如，如下的 C# 接口就是有效的定义：

```
interface ICalculator
{
    int Add(int arg1,int arg2);
    double Add(double arg1,double arg2);
}
```

然而，基于 WSDL 的操作却不支持操作重载。因此，在编译如下的契约定义时，装载服务宿主就会抛出 `InvalidOperationException` 异常：

```
// 无效的契约定义：
[ServiceContract]
interface ICalculator
{
    [OperationContract]
    int Add(int arg1,int arg2);

    [OperationContract]
    double Add(double arg1,double arg2);
}
```

但是，我们可以手动地启用操作重载。实现的窍门就是使用 `OperationContract` 特性的 `Name` 属性，为操作指定别名：

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public string Name
    {get;set;}
    // 更多成员
}
```

我们需要同时为服务与客户端的操作指定别名。在服务端，要为重载的操作提供唯一的标识名，如例 2-1 所示。

#### 例 2-1：服务端的操作重载

```
[ServiceContract]
interface ICalculator
{
    [OperationContract(Name = "AddInt")]
    int Add(int arg1,int arg2);

    [OperationContract(Name = "AddDouble")]
    double Add(double arg1,double arg2);
}
```

当客户端导入契约并生成代理时，导入的操作就会包含定义的别名：

```
[ServiceContract]
public interface ICalculator
{
    [OperationContract]
    int AddInt(int arg1,int arg2);

    [OperationContract]
    double AddDouble(double arg1,double arg2);
}
public partial class CalculatorClient : ClientBase<ICalculator>,ICalculator
{
    public int AddInt(int arg1,int arg2)
    {
        return Channel.AddInt(arg1,arg2);
    }
    public double AddDouble(double arg1,double arg2)
    {
        return Channel.AddDouble(arg1,arg2);
    }
    // 代理的其余内容
}
```

客户端虽然可以使用生成的代理和契约，但我们还需要进行修改，使客户端代码支持操作重载。方法是将导入的代理与契约的方法名修改为重载的名称，并确保代理类能够使用重载方法调用内部代理，例如：

```
public int Add(int arg1,int arg2)
{
    return Channel.Add(arg1,arg2);
}
```

最后,在客户端使用导入契约的 `Name` 属性,指定别名并重载方法,使它与导入的操作名保持一致,如例 2-2 所示。

### 例 2-2: 客户端操作重载

```
[ServiceContract]
public interface ICalculator
{
    [OperationContract(Name = "AddInt")]
    int Add(int arg1,int arg2);

    [OperationContract(Name = "AddDouble")]
    double Add(double arg1,double arg2);
}

public partial class CalculatorClient : ClientBase<ICalculator>,ICalculator
{
    public int Add(int arg1,int arg2)
    {
        return Channel.Add(arg1,arg2);
    }
    public double Add(double arg,double arg2)
    {
        return Channel.Add(arg1,arg2);
    }
    // 代理的其余内容
}
```

现在,通过操作重载,客户端就能够提供更加自然与优雅的编程模型,具有良好的可读性:

```
CalculatorClient proxy = new CalculatorClient();

int result1 = proxy.Add(1,2);
double result2 = proxy.Add(1.0,2.0);

proxy.Close();
```

## 契约的继承 (译注 1)

服务契约接口支持继承功能,我们可以定义一个契约层级。但是, `ServiceContract` 特性却是不能继承的:

---

译注 1: 要实现在客户端契约的继承,主要是通过手工修改客户端代码的方式。



```
[AttributeUsage(Inherited = false,...)]  
public sealed class ServiceContractAttribute : Attribute  
{...}
```

因此, 接口层级中的每级接口都必须显式的标记 `ServiceContract` 特性, 如例 2-3 所示。

### 例 2-3: 服务端契约层级

```
[ServiceContract]  
interface ISimpleCalculator  
{  
    [OperationContract]  
    int Add(int arg1,int arg2);  
}  
[ServiceContract]  
interface IScientificCalculator : ISimpleCalculator  
{  
    [OperationContract]  
    int Multiply(int arg1,int arg2);  
}
```

至于一个契约层级的实现, 一个单独的服务类能够实现整个契约层级, 这与经典的 C# 编程完全一致:

```
class MyCalculator : IScientificCalculator  
{  
    public int Add(int arg1,int arg2)  
    {  
        return arg1 + arg2;  
    }  
    public int Multiply(int arg1,int arg2)  
    {  
        return arg1 * arg2;  
    }  
}
```

宿主可以为契约层级最底层的接口公开一个单独的终结点:

```
<service name = "MyCalculator">  
  <endpoint  
    address = "http://localhost:8001/MyCalculator/"  
    binding = "basicHttpBinding"  
    contract = "IScientificCalculator"  
  />  
</service>
```

## 客户端契约层级

当客户端导入一个服务终结点的元数据时, 如果该终结点的契约属于接口层级的一部分, 则生成的客户端契约将不再维持原来的层级关系。相反, 它会取消层级, 组成一个单独

的契约, 名称为终结点的契约名。这个单独的契约包含了层级中从上至下所有接口定义的操作。然而, 如果使用 `OperationContract` 特性中的 `Action` 与 `ReplyAction` 属性, 那么导入的接口定义仍然可以保留原来定义每个操作的契约名。

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public string Action
    {get;set;}
    public string ReplyAction
    {get;set;}
    // 更多成员
}
```

最后, 一个单独的代理类可以实现导入契约的所有方法。如果给定例 2-3 的定义, 导入的契约以及生成的代理类如例 2-4 所示。

#### 例 2-4: 取消层级关系的客户端定义

```
[ServiceContract]
public interface IScientificCalculator
{
    [OperationContract(Action = ".../ISimpleCalculator/Add",
        ReplyAction = ".../ISimpleCalculator/AddResponse")]
    int Add(int arg1,int arg2);

    [OperationContract(Action = ".../IScientificCalculator/Multiply",
        ReplyAction = ".../IScientificCalculator/MultiplyResponse")]
    int Multiply(int arg1,int arg2);
}

public partial class ScientificCalculatorClient :
    ClientBase<IScientificCalculator>, IScientificCalculator
{
    public int Add(int arg1,int arg2)
    {...}
    public int Multiply(int arg1,int arg2)
    {...}
    // 代理的其余内容
}
```

#### 恢复客户端层级

客户端可以手工修改代理以及导入契约的定义, 恢复契约层级, 如例 2-5 所示。

#### 例 2-5: 客户端契约层级

```
[ServiceContract]
public interface ISimpleCalculator
{
    [OperationContract]
```

```

        int Add(int arg1,int arg2);
    }
    public partial class SimpleCalculatorClient : ClientBase<ISimpleCalculator>,
                                                ISimpleCalculator
    {
        public int Add(int arg1,int arg2)
        {
            return Channel.Add(arg1,arg2);
        }
        // 代理的其余内容
    }

    [ServiceContract]
    public interface IScientificCalculator : ISimpleCalculator
    {
        [OperationContract]
        int Multiply(int arg1,int arg2);
    }
    public partial class ScientificCalculatorClient :
                                                ClientBase<IScientificCalculator>,IScientificCalculator
    {
        public int Add(int arg1,int arg2)
        {
            return Channel.Add(arg1,arg2);
        }
        public int Multiply(int arg1,int arg2)
        {
            return Channel.Multiply(arg1,arg2);
        }
        // 代理的其余内容
    }

```

在不同的操作上使用Action属性值,客户端可以分解服务契约层级中合成契约的定义,提供接口与代理的定义。如例2-5中的ISimpleCalculator和SimpleCalculatorClient。在该例中,并不需要设置Action和ResponseAction属性值,我们完全可以移除它们。然后,手动地将接口添加到客户端所需要的接口链中:

```

[ServiceContract]
public interface IScientificCalculator : ISimpleCalculator
{...}

```

尽管服务可能已经为层级中最底层的接口公开了一个单独的终结点,客户端仍然可以将它看作是相同地址的不同终结点,每个终结点对应契约层级的不同层:

```

<client>
  <endpoint name = "SimpleEndpoint"
    address = "http://localhost:8001/MyCalculator/"
    binding = "basicHttpBinding"
    contract = "ISimpleCalculator"
  />
  <endpoint name = "ScientificEndpoint"
    address = "http://localhost:8001/MyCalculator/"

```

```
        binding = "basicHttpBinding"
        contract = "IScientificCalculator"
    />
</client>
```

现在，客户端可以编写如下代理，充分地利用契约层级的优势：

```
SimpleCalculatorClient proxy1 = new SimpleCalculatorClient();
proxy1.Add(1,2);
proxy1.Close();

ScientificCalculatorClient proxy2 = new ScientificCalculatorClient();
proxy2.Add(3,4);
proxy2.Multiply(5,6);
proxy2.Close();
```

例 2-5 对代理的分解，解除了契约中每一层级之间的依赖，实现了契约层级的解耦。在客户端代码中，凡是希望使用 `ISimpleCalculator` 引用的，都可以指派为 `IScientificCalculator` 类型的引用（译注 2）：

```
void UseCalculator(ISimpleCalculator calculator)
{...}

ISimpleCalculator proxy1 = new SimpleCalculatorClient();
ISimpleCalculator proxy2 = new ScientificCalculatorClient();
IScientificCalculator proxy3 = new ScientificCalculatorClient();
SimpleCalculatorClient proxy4 = new SimpleCalculatorClient();
ScientificCalculatorClient proxy5 = new ScientificCalculatorClient();

UseCalculator(proxy1);
UseCalculator(proxy2);
UseCalculator(proxy3);
UseCalculator(proxy4);
UseCalculator(proxy5);
```

但是，代理之间并不存在 IS-A 关系。即使 `IScientificCalculator` 接口派生自 `ISimpleCalculator` 接口，也不能认为代理类 `ScientificCalculatorClient` 就是 `SimpleCalculatorClient` 类型。此外，我们必须为子契约重复实现代理中的基契约。调整的办法是使用所谓的代理链（Proxy Chaining）技术，如例 2-6 所示。

#### 例 2-6：代理链

```
public partial class SimpleCalculatorClient : ClientBase<IScientificCalculator>,
                                             ISimpleCalculator
{
    public int Add(int arg1,int arg2)
    {
        return Channel.Add(arg1,arg2);
    }
}
```

译注 2： 使用了面向对象思想的多态原理。

```
// 代理的其余内容
}

public partial class ScientificCalculatorClient : SimpleCalculatorClient,
                                                IScientificCalculator
{
    public int Multiply(int arg1,int arg2)
    {
        return Channel.Multiply(arg1,arg2);
    }
    // 代理的其余内容
}
```

只有实现了最顶层的基契约的代理直接继承于 `ClientBase<T>`, 提供的类型参数则为最底层的子接口类型。所有的其他代理则直接继承于它们的上一级代理, 同时实现各自的契约接口。

代理链为代理建立了 IS-A 关系, 保证了代码的重用。在客户端代码中, 凡是希望使用 `SimpleCalculatorClient` 引用的, 都可以指派为 `ScientificCalculatorClient` 类型的引用:

```
void UseCalculator(SimpleCalculatorClient calculator)
{...}

SimpleCalculatorClient proxy1 = new SimpleCalculatorClient();
SimpleCalculatorClient proxy2 = new ScientificCalculatorClient();
ScientificCalculatorClient proxy3 = new ScientificCalculatorClient();

UseCalculator(proxy1);
UseCalculator(proxy2);
UseCalculator(proxy3);
```

## 服务契约的分解与设计

如果不考虑语法因素, 我们应该如何设计服务契约? 如何知道服务契约中应该定义哪些操作? 每个契约又应该包含多少操作? 解决这些问题与 WCF 技术并无太大关系, 更多地属于抽象的面向服务分析与设计的范畴。如何将系统分解为服务, 以及如何剖析契约方法, 并不在本书讨论范围之内。不过, 本节仍然给出了一些建议, 以指导开发者更好地设计服务契约。

### 契约分解

一个服务契约是逻辑相关的操作的组合。所谓的“逻辑相关”通常指特定的领域逻辑。我们可以将服务契约想象成实体的不同表现。一旦识别 (在需求分析之后) 出实体支持的所有操作, 就需要将它们分配给契约。这称为服务契约的分解 (Service Contract

Factoring)。分解服务契约时,通常需要考虑可重用元素(Reusable Element)。在面向服务的应用程序中,一个可重用的基本单元就是服务契约。那么,系统的其他实体能否重用这些被分解出的服务契约?实体对象的哪些职责能够被分解出来,哪些职责又能被其他实体所调用?

让我们考虑一个具体而又简单的实例。假定我们希望对一个狗的服务建模。需求说明狗能叫能吃,拥有一个兽医诊所的注册号,可以对它注射疫苗。我们可以定义一个 IDog 服务契约,并让不同的服务如 PoodleService (狮子狗) 和 GermanShepherdService (德国牧羊犬) 实现 IDog 契约:

```
[ServiceContract]
interface IDog
{
    [OperationContract]
    void Fetch();

    [OperationContract]
    void Bark();

    [OperationContract]
    long GetVetClinicNumber();

    [OperationContract]
    void Vaccinate();
}
class PoodleService : IDog
{...}
class GermanShepherdService : IDog
{...}
```

然而, IDog 服务契约的定义并没有体现职责分离的原则。虽然这些操作都是狗所应具有的,但是 Fetch() 和 Bark() 方法与 IDog 服务契约的逻辑关联性,远远强于 GetVetClinicNumber() 和 Vaccinate() 方法。Fetch() 和 Bark() 体现了狗的本性,与它的日常生活有关,属于实例化的犬类实体的职责。GetVetClinicNumber() 和 Vaccinate() 则体现了不同的特性,它们与兽医诊所的宠物记录有关。一个最佳方案是将 GetVetClinicNumber() 和 Vaccinate() 操作分解出来,形成一个单独的 IPet 契约:

```
[ServiceContract]
interface IPet
{
    [OperationContract]
    long GetVetClinicNumber();

    [OperationContract]
    void Vaccinate();
}

[ServiceContract]
```



```
interface IDog
{
    [OperationContract]
    void Fetch();

    [OperationContract]
    void Bark();
}
```

由于宠物的职责不依赖于犬类实体，因此其他实体（例如猫）可以重用以及实现 `IPet` 服务契约：

```
[ServiceContract]
interface ICat
{
    [OperationContract]
    void Purr();

    [OperationContract]
    void CatchMouse();
}

class PoodleService : IDog, IPet
{...}

class SiameseService : ICat, IPet
{...}
```

契约的分解实现了应用程序中诊所管理职责与实际服务（狗或者猫）之间的解耦。将操作分解为单独的接口，是服务设计中常见的做法，它能够降低操作之间的逻辑关系。但是，有时候在几个不相关的契约中会找到相同的操作，这些操作与它们各自的契约存在一定的逻辑关系。例如，猫和狗这两种动物都会脱毛，都能够哺育后代。从逻辑上讲，脱毛与犬吠一样，都属于狗的服务操作；同时它又与猫叫一样，属于猫的服务操作。

此时，我们将服务契约分解为契约层级的方式，而不是单独的契约：

```
[ServiceContract]
interface IMammal
{
    [OperationContract]
    void ShedFur();

    [OperationContract]
    void Lactate();
}

[ServiceContract]
interface IDog : IMammal
{...}

[ServiceContract]
interface ICat : IMammal
{...}
```

## 分解准则

显而易见，合理的契约分解（译注3）可以实现深度特化、松散耦合、精细调整以及契约的重用。这些优势有助于改善整个系统。总的来说，契约分解的目的就是使契约包含的操作尽可能少。

设计面向服务的系统时，需要平衡两个影响系统的因素（参见图2-1）。一个是实现服务契约的代价，一个则是将服务契约合并或集成为一个高内聚应用程序的代价。

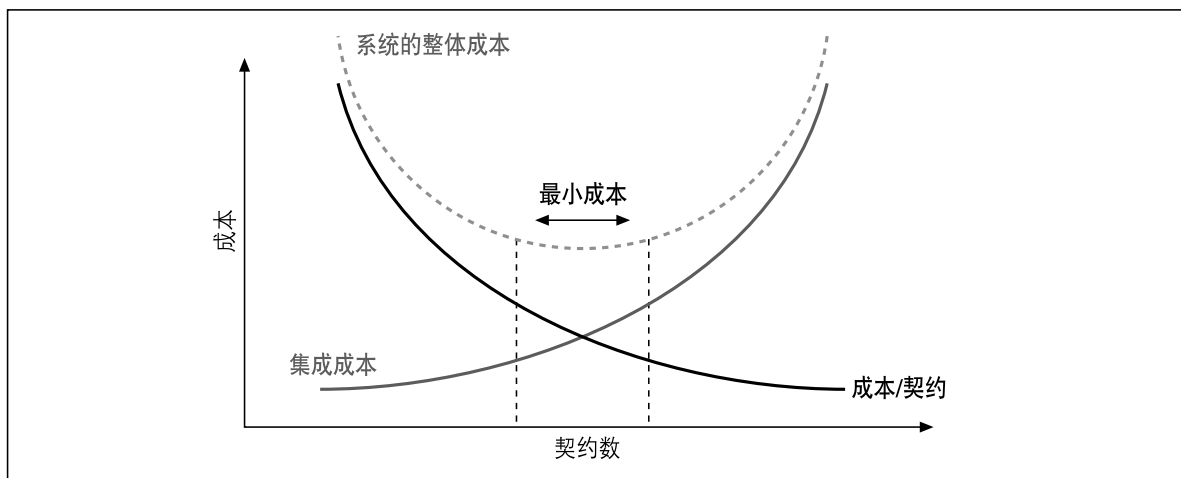


图2-1：平衡服务的个数与规模

如果我们定义了太多的细粒度服务契约，虽然它们易于实现，但集成它们的代价未免太高。另一方面，如果我们仅定义了几个复杂而又庞大的服务契约，虽然集成的代价可能会降低，但却制约了契约的实现。

实现契约的代价与服务契约的规模并非线性的关系，当契约的规模增加两倍时，复杂度会陡增至四到六倍。与之相似，集成契约的代价与服务契约的数量同样不是线性关系，因为参与的服务数与它们之间关联点的数目不是线形的。

对于任何一个系统，实现契约所付出的代价，包括设计服务以及维护服务的代价，等于上述两个因素的总和（实现的代价与集成的代价）。图2-1的一个区域显示了最小代价与服务契约规模和数量之间的关系。一个设计良好的系统，服务的个数与规模应该恰如其分，遵循平衡的“中庸之道”，力求达到“增之一分则太多（大），减之一分则太少（小）”的标准。

译注3： 契约分离与接口隔离原则（ISP，Interface Segregation Principle）的基本精神是一致的。

ISP原则建议使用多个专门的接口，而不是使用单个接口，这样可以防止接口污染，有利于接口重用。契约分解同样如此，但它还要受到实现契约代价的约束。

由于契约分解与使用的服务技术无关,对于职责分离以及大规模应用程序的架构设计,我们只能根据自己或他人的经验,总结出关于服务契约分解的规则和方法,与读者分享。

首先,我们应该避免设计只具有一个操作的服务契约。一个服务契约体现了实体的特征,如果服务只有一个操作,则过于单调,没有实际的意义。此时,就应该检查它是否使用了太多的参数?它的粒度是否过粗,因此需要分解为多个操作?是否需要将该操作转移到已有的服务契约中?

服务契约成员的最佳数量(根据经验总结,仅代表本人观点)应介于3到5之间。如果设计的服务契约包含了多个操作,例如6到9个,仍然可能工作良好。但是,我们需要判断这些操作会否因为过度分解而需要合并。如果服务契约定义了12个甚至更多的操作,毫无疑问,我们需要将这些操作分解到单独的服务契约中,或者为它们建立契约层级。开发者在制订WCF编码规范时,应该指定一个上限值(例如20)。无论在何种情况,都不能超过该值。

另一个原则是关于准属性操作(Property-Like Operation)的使用,例如:

```
[OperationContract]  
long GetVetClinicNumber();
```

我们应该避免定义这样的操作。服务契约允许客户端在调用抽象操作时,不用关心具体的实现细节。准属性操作由于无法封装状态的管理,因此在封装性的表现上差强人意。在服务端,我们可以封装读写变量值的业务逻辑,但在理想状态下,我们却不应该干涉客户端对属性的使用。客户端应该只负责调用操作,而由服务去管理服务对象的状态。这种交互方式应该被表示为DoSomething()样式,例如Vaccinate()方法。服务如何实现该方法,是否需要设置诊所号,都不是客户端需要考虑的内容。

需要注意的是,这些分解原则,包括经验法则与通用规律,只能作为帮助开发者核算和评估特定设计的工具。它不能替代领域专家的意见与经验。“实践出真知”,应用这些指导原则时,需要做出合理的判断,甚至提出质问。

## 契约查询

有时候,客户端需要通过编程方式验证一个特定的终结点(通过地址进行识别)是否支持一个特定的契约。设想有这样一个应用程序,终端用户在安装时(甚至在运行时)指定或配置应用程序,用以使用服务并与服务交互。如果服务不支持所需的契约,应用程序就会向用户发出警告,提示配置的地址是无效的,询问是否更正地址或替换地址。例如,第10章使用的证书管理器应用程序(Credentials Manager Application)就具备这样的特征:用户需要为应用程序提供管理账户成员与角色的安全证书服务的地址。在验证了地址支持所需的服务契约之后,证书管理器只允许用户选择有效的地址。

## 编程处理元数据

为了支持这一功能,应用程序需要获取服务终结点的元数据,查看是否存在至少一个终结点支持请求的契约。正如第 1 章阐释的那样,如果元数据交换终结点是服务支持的,或者基于 HTTP-GET 协议,那么元数据在这个终结点中就是可用的。当我们使用 HTTP-GET 协议时,元数据交换的地址就是 HTTP-GET 地址(通常,服务的基地址以?wsdl 为后缀)。为了简化对返回元数据的解析工作,WCF 提供了几个辅助类,位于 System.ServiceModel.Description 命名空间,如例 2-7 所示。

### 例 2-7: 支持元数据处理的类型

```
public enum MetadataExchangeClientMode
{
    MetadataExchange,
    HttpGet
}
class MetadataSet : ...
{...}
public class ServiceEndpointCollection : Collection<ServiceEndpoint>
{...}

public class MetadataExchangeClient (译注 4)
{
    public MetadataExchangeClient();
    public MetadataExchangeClient(Binding mexBinding);
    public MetadataSet GetMetadata(Uri address, MetadataExchangeClientMode mode);
    // 更多成员
}
public abstract class MetadataImporter
{
    public abstract ServiceEndpointCollection ImportAllEndpoints();
    // 更多成员
}
public class WsdlImporter : MetadataImporter
{
    public WsdlImporter(MetadataSet metadata);
    // 更多成员
}
```

译注 4: 这里列举的 MetadataExchangeClient 成员并不全面,以下是在例 2-8 中会使用到的几个成员:

一个构造函数版本:

```
public MetadataExchangeClient(Uri address, MetadataExchangeClientMode mode);
```

还有两个 GetMetadata() 方法的重载版本:

```
public MetadataSet GetMetadata();
public MetadataSet GetMetadata(EndpointAddress address);
```

```
public class ServiceEndpoint
{
    public EndpointAddress Address
    {get;set;}
    public Binding Binding
    {get;set;}
    public ContractDescription Contract
    {get;}
    // 更多成员
}
public class ContractDescription
{
    public string Name
    {get;set;}
    public string Namespace
    {get;set;}
    // 更多成员
}
```

MetadataExchangeClient能够使用与元数据交换关联的绑定,该元数据交换保存在应用程序的配置文件中。我们也可以将初始化后的绑定实例传递给MetadataExchangeClient的构造函数。传递的绑定实例包含一些自定义值,例如容量。如果返回的元数据超过默认接收消息大小时,为了接收更大的消息,就可以设置容量值。MetadataExchangeClient的GetMetadata()方法接收一个终结点地址实例,它封装了元数据交换地址以及一个枚举值,指定了访问的方式。方法返回的元数据放在一个MetadataSet实例中。我们不需要直接操作MetadataSet类型,而是创建MetadataImporter类的子类实例,例如WsdlImporter,将原来的元数据(译注5)传递给它的构造函数,然后调用ImportAllEndpoints()方法,获取在元数据中查找到的所有终结点的集合。终结点以ServiceEndpoint类型方式表示。

ServiceEndpoint定义了ContractDescription类型属性Contract。ContractDescription类定义了契约的名称与命名空间。

使用HTTP-GET时,为了判断配置的基地址是否支持特定的契约,通过刚才描述的步骤就能够生成终结点的集合。遍历集合的每一个终结点,比较请求契约中ContractDescription的Name和Namespace属性值,如例2-8所示。

#### 例 2-8: 查询契约的地址

```
bool contractSupported = false;

string mexAddress = "...?WSDL";

MetadataExchangeClient MEXClient = new MetadataExchangeClient(new Uri(mexAddress),
                                                                MetadataExchangeClientMode.HttpGet);
```

---

译注 5: 即 MetadataSet 实例。

```
MetadataSet metadata = MEXClient.GetMetadata();
MetadataImporter importer = new WsdlImporter(metadata);
ServiceEndpointCollection endpoints = importer.ImportAllEndpoints();

foreach(ServiceEndpoint endpoint in endpoints)
{
    if(endpoint.Contract.Namespace == "MyNamespace" &&
        endpoint.Contract.Name == "IMyContract")
    {
        contractSupported = true;
        break;
    }
}
```

注意：第 1 章提到的元数据浏览器工具采用的步骤与例 2-8 获取服务终结点的步骤相似。如果给定一个基于 HTTP 的地址，工具会同时尝试使用 HTTP-GET 和基于 HTTP 的元数据交换终结点。元数据浏览器也能够使用基于 TCP 或 IPC 的元数据交换终结点获取元数据。工具的大量实现都是用于处理元数据，以及显示元数据的内容，毕竟，WCF 提供的类很难获取和解析元数据。

## MetadataHelper 类

我们将例 2-8 所示的步骤封装到了设计的通用静态工具类 `MetadataHelper()` 的 `QueryContract()` 方法中：

```
public static class MetadataHelper
{
    public static bool QueryContract(string mexAddress, Type contractType);
    public static bool QueryContract(string mexAddress, string contractNamespace,
                                    string contractName);

    // 更多成员
}
```

可以为 `MetadataHelper` 类提供我们希望查询的契约类型，或者提供该契约的名称与命名空间：

```
string address = "...";
bool contractSupported = MetadataHelper.QueryContract(address, typeof(IMyContract));
```

至于 `QueryContract()` 方法中的元数据交换地址 `mexAddress`，我们可以提供带 HTTP-GET 地址的 `MetadataHelper` 类，也可以提供基于 HTTP、HTTPS、TCP 或者 IPC 的元数据交换终结点地址。例 2-9 演示了 `MetadataHelper.QueryContract()` 方法的实现，它省略了错误处理的代码。



## 例 2-9: MetadataHelper.QueryContract()的实现

```
public static class MetadataHelper
{
    const int MessageMultiplier = 5;

    static ServiceEndpointCollection QueryMexEndpoint(string mexAddress,
                                                       BindingElement bindingElement)
    {
        CustomBinding binding = new CustomBinding(bindingElement);

        MetadataExchangeClient MEXClient = new MetadataExchangeClient(binding);
        MetadataSet metadata = MEXClient.GetMetadata
                               (new EndpointAddress(mexAddress));
        MetadataImporter importer = new WsdlImporter(metadata);
        return importer.ImportAllEndpoints();
    }

    public static ServiceEndpoint[] GetEndpoints(string mexAddress)
    {
        /* 一些错误处理 */
        Uri address = new Uri(mexAddress);
        ServiceEndpointCollection endpoints = null;

        if(address.Scheme == "net.tcp")
        {
            TcpTransportBindingElement tcpBindingElement =
                new TcpTransportBindingElement();
            tcpBindingElement.MaxReceivedMessageSize *= MessageMultiplier;
            endpoints = QueryMexEndpoint(mexAddress, tcpBindingElement);
        }
        if(address.Scheme == "net.pipe")
        {
            ...
        }
        if(address.Scheme == "http") // 判断是否为 HTTP-GET
        {
            ...
        }
        if(address.Scheme == "https") // 判断是否为 HTTPS-GET
        {
            ...
        }
        return Collection.ToArray(endpoints);
    }

    public static bool QueryContract(string mexAddress, Type contractType)
    {
        if(contractType.IsInterface == false)
        {
            Debug.Assert(false, contractType + " is not an interface");
            return false;
        }
        object[] attributes = contractType.GetCustomAttributes(
                                         typeof(ServiceContractAttribute), false);
        if(attributes.Length == 0)
        {
            Debug.Assert(false, "Interface " + contractType +
                               " does not have the ServiceContractAttribute");
            return false;
        }
    }
}
```

```
ServiceContractAttribute attribute = attributes[0] as
ServiceContractAttribute;
if(attribute.Name == null)
{
    attribute.Name = contractType.ToString();
}
if(attribute.Namespace == null)
{
    attribute.Namespace = "http://tempuri.org/";
}
return QueryContract(mexAddress,attribute.Namespace,attribute.Name);
}
public static bool QueryContract(string mexAddress,string contractNamespace,
string contractName)
{
    if(String.IsNullOrEmpty(contractNamespace))
    {
        Debug.Assert(false,"Empty namespace");
        return false;
    }
    if(String.IsNullOrEmpty(contractName))
    {
        Debug.Assert(false,"Empty name");
        return false;
    }
    try
    {
        ServiceEndpoint[] endpoints = GetEndpoints(mexAddress);
        foreach(ServiceEndpoint endpoint in endpoints)
        {
            if(endpoint.Contract.Namespace == contractNamespace &&
                endpoint.Contract.Name == contractName)
            {
                return true;
            }
        }
    }
    catch
    {
        return false;
    }
}
```

在例2-9中，GetEndpoints()方法对元数据交换地址的样式进行了解析。根据找到的传输样式（例如TCP），GetEndpoints()方法创建了一个需要使用的绑定元素，这样就可以设置它的MaxReceivedMessageSize属性值：

```
public abstract class TransportBindingElement : BindingElement
{
    public virtual long MaxReceivedMessageSize
    {get;set;}
}
```

```
public abstract class ConnectionOrientedTransportBindingElement :  
    TransportBindingElement, ...  
{...}  
public class TcpTransportBindingElement : ConnectionOrientedTransportBindingElement  
{...}
```

MaxReceiveMessageSize的默认值为64K。它适用于简单的服务。如果服务包含多个终结点，终结点又使用了复杂类型，就会生成更大的消息。此时，调用MetadataExchangeClient.GetMetadata()方法就会失败。根据经验，大多数情况下最合适的倍数因子是5。接着，GetEndpoints()调用了QueryMexEndpoint()私有方法，以获取元数据。QueryMexEndpoint()接收元数据交换终结点的地址以及要使用的绑定元素。使用绑定元素是为了创建定制绑定，并将它提供给MetadataExchangeClient实例。MetadataExchangeClient实例能够获取元数据，返回终结点集合（译注6）。但是，GetEndpoints()方法返回的终结点集合不是ServiceEndpointCollection类型，而是使用了我们设计的Collection辅助类，返回了一个终结点数组。

接收Type参数的QueryContract()方法首先会验证传入的Type类型是否是接口类型，如果是，则判断该接口是否标记了ServiceContract特性。因为ServiceContract特性可以为契约的请求类型指定名称和命名空间的别名，QueryContract()使用这些值查询符合条件的契约。如果没有指定别名，QueryContract()方法则使用类型的名字与默认的命名空间http://tempuri.org，然后调用另一个重载版本的QueryContract()方法，它能够操作契约的名称和命名空间。该版本的QueryContract()方法调用了GetEndpoints()方法，以获得终结点数组，然后遍历该数组。如果找到至少一个终结点支持该契约，则返回true。不管出现何种错误，QueryContract()方法都会返回false。

例2-10介绍了MetadataHelper类定义的额外的查询元数据的方法。

#### 例2-10: MetadataHelper 类

```
public static class MetadataHelper  
{  
    public static ServiceEndpoint[] GetEndpoints(string mexAddress);  
    public static string[] GetAddresses(Type bindingType, string mexAddress,  
                                        Type contractType);  
    public static string[] GetAddresses(string mexAddress, Type contractType);  
    public static string[] GetAddresses(Type bindingType, string mexAddress,  
                                        string contractNamespace, string contractName)  
        where B: Binding;
```

---

译注6: QueryMexEndpoint()方法会将MetadataExchangeClient实例获得的元数据传递给WsdImporter的ImportAllEndpoints()方法，获得终结点集合。

```
public static string[] GetAddresses(string mexAddress,string contractNamespace,
                                   string contractName);
public static string[] GetContracts(Type bindingType,string mexAddress);
public static string[] GetContracts(string mexAddress);
public static string[] GetOperations(string mexAddress,Type contractType);
public static string[] GetOperations(string mexAddress,
                                   string contractNamespace,
                                   string contractName);
public static bool QueryContract(string mexAddress,Type contractType);
public static bool QueryContract(string mexAddress,
                                   string contractNamespace,string contractName);
// 更多成员
}
```

在管理程序和管理工具中,或者在对契约进行设置时,往往需要这些强大而有效的功能。它们的实现都是基于对终结点数组的处理,终结点数组则是通过`GetEndpoints()`方法返回的。

`GetAddresses()`方法返回的终结点地址,要么支持一个特定的契约,要么就是使用特定绑定的终结点地址。

相似的,`GetContracts()`方法返回的所有契约,要么被所有终结点支持,要么就是使用了特定绑定的所有终结点支持的契约。最后,`GetOperations()`方法会返回一个特定契约的所有操作。

---

注意: 第 10 章的证书管理器应用程序使用了 `MetadataHelper` 类,附录 B 则使用它来管理持久订阅者。

---

## 第3章

# 数据契约

WCF能够托管CLR类型（接口和类）并将它们公开为服务，也能够以本地CLR接口和类的方式使用服务。WCF服务的操作接收和返回类似于`int`和`string`的CLR类型，WCF客户端则传递和处理返回的CLR类型。然而，CLR类型却属于.NET的特定技术。这就带来一个问题，由于面向服务的一个核心原则就是在跨越服务边界时，服务不能够暴露它们的实现技术。因此，不管客户端采用了何种技术，它都能够与服务交互。显然，这意味着WCF不允许在跨越服务边界时公开CLR数据类型。我们需要找到一种办法，实现CLR数据类型与标准的平台无关的表示形式之间的转换。这样的表示形式就是基于XML的样式，也可以称之为信息集（`Infoset`）。此外，服务需要一种正式的规格说明来声明两者之间的转换。这个方法就是本章所要介绍的主题——数据契约。本章的第一部分介绍了数据契约支持类型编组（`Type Marshaling`）与转换的方法，以及如何通过基础架构处理类的层级与数据契约的版本控制。第二部分则介绍了如何将不同的.NET类型，例如枚举、委托、数据表以及集合，作为数据契约使用。

## 序列化

数据契约是服务支持的契约职责的一部分，就像服务契约是组成契约的一部分一样。数据契约发布于服务元数据中，服务元数据允许客户端将与平台、技术无关的数据类型表示形式转换为客户端本地的表示形式。由于对象与本地引用属于CLR的概念，因此无法实现与WCF服务操作之间的传递。如果允许传递CLR对象和引用，不仅会与之前讨论的面向服务的核心原则产生冲突，而且也是不现实的做法，因为对象是由操作它的状态和代码构成的。既然无法发送代码以及C#或者Visual Basic的方法调用逻辑，则唯一的解决之道就是将对象编组（`Marshaling`）到其他的平台和技术。事实上，将对象（或者值类型）作为操作参数进行传递时，真正需要发送的是对象的状态，然后，接收端再将它转换为本地的表示形式。这种传递状态的方式称为按值编组（`Marshaling By Value`）。执行按值编组的最简单

办法是利用大多数平台（包括.NET）自身提供的序列化技术。按值编组的实现非常简单，如图3-1所示。

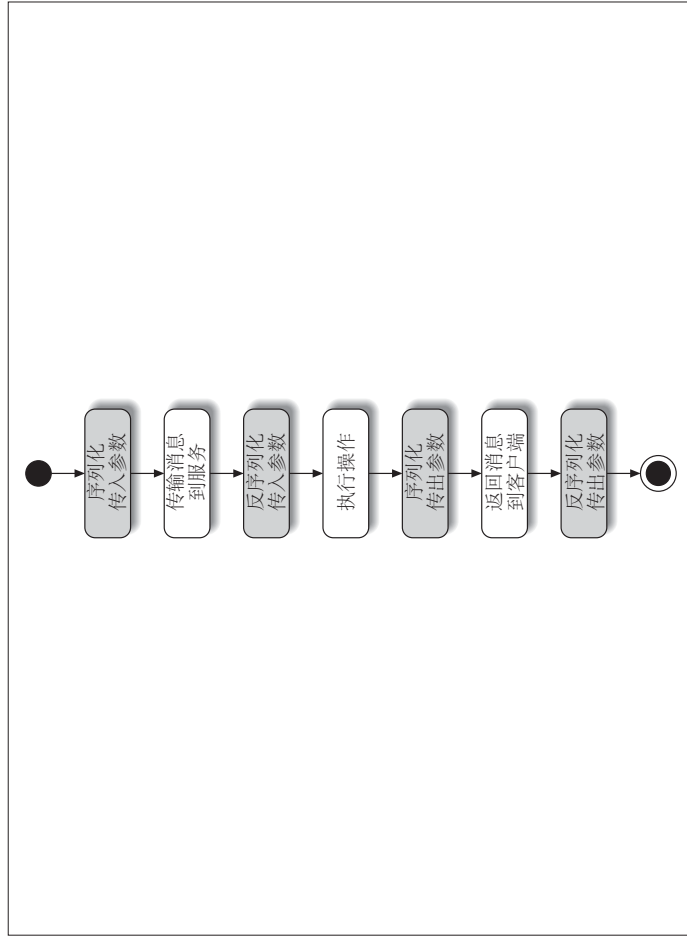


图3-1：操作调用期间的序列化与反序列化

在客户端，WCF会对传入参数执行序列化操作，将CLR本地表示形式转换为XML信息集，并将它们捆绑在客户端的输出消息中。一旦服务端接收到消息，WCF就会在将调用分发到服务之前执行反序列化，将与平台无关的XML信息集转换为对应的CLR表示形式。然后，服务会处理这些本地的CLR参数。一旦服务完成对操作的执行，WCF会序列化输出参数以及返回值，将它们转换为与平台无关的XML信息集，捆绑在一起放到返回消息中，传递给客户端。最后，WCF会在客户端对返回值执行反序列化，将它们转换为本地的CLR类型后返回到客户端。

对每次调用执行两次序列化和反序列化是WCF的性能瓶颈。与序列化相比，通过拦截器在客户端和服务端运行消息的负荷就显得微不足道了。





## .NET序列化

WCF能够利用现有的.NET技术实现序列化。.NET通过反射技术自动地实现了对对象的序列化与反序列化。.NET能够捕获对象的每个字段的值,并将它序列化到内存、文件或网络连接中。至于反序列化,.NET使用反射创建一个对应类型的新对象,读取它的持久化字段值,然后设置字段的值。由于反射能够访问私有字段,甚至包括基类的字段,因此.NET在序列化期间能够完整地反映出对象的状态,从而保证了反序列化可以完整地重建对象状态。.NET将对象状态序列化到流(Stream)中。流是字节的逻辑序列,与特定的介质如文件、内存、通信端口或其他资源无关。

### Serializable特性

默认情况下,用户自定义的类型(类和结构)并不支持序列化。因为.NET无法判断对象状态是否需要反射到流。或许对象成员只有一些临时值或临时状态(例如打开的数据库连接或者通信端口)。如果.NET对这样的对象状态执行了序列化,那么在反序列化时,根据流的内容创建的新对象,就可能存在缺陷。因此,是否执行序列化应该由类的开发者决定。

.NET中,若要指定类的实例支持序列化,需要在类或结构的定义中添加Serializable-Attribute特性:

```
[AttributeUsage(AttributeTargets.Delegate |
    AttributeTargets.Enum |
    AttributeTargets.Struct |
    AttributeTargets.Class,
    Inherited=false)]
public sealed class SerializableAttribute : Attribute
{
}
```

例如:

```
[Serializable]
class MyClass
{...}
```

### NonSerialized特性

一个类如果是可序列化的,则.NET要求它的所有成员变量都要支持序列化,如果发现存在不支持序列化的成员,就会抛出异常。然而,当存在一个希望序列化的类或者结构包含一个无法序列化的成员时,我们应该怎样设计?由于该成员的类型并不具有Serializable特性,在序列化时它会被排除在序列化的类型之外。通常情况下,非序列化成员就是那些需要执行特殊的初始化操作的引用类型。解决这个问题的办法是为这样的成员标记NonSerialized特性,然后在反序列化时,采取定制步骤对它进行初始化。

若要允许一个可序列化类型包含非序列化的成员变量,可以为该成员标记NonSerialized

字段特性，例如：

```
class MyOtherClass
{...}

[Serializable]
class MyClass
{
    [NonSerialized]
    MyOtherClass m_OtherClass;
    /*方法与属性*/
}
```

当.NET序列化成员变量时，首先通过反射确定它是否具有`NonSerialized`特性。如果有，.NET就会忽略该变量，直接跳过它。

你甚至可以采用这种技术将那些可正常序列化的类型例如`string`，排除在序列化之外：

```
[Serializable]
class MyClass
{
    [NonSerialized]
    string m_Name;
}
```

## .NET 格式器 (.NET Formatter)

.NET为类型的序列化和反序列化提供了两种格式器。`BinaryFormatter`会将类型序列化为二进制格式。这种格式器能够快速执行序列化和反序列化操作。`SoapFormatter`则使用了.NET特定的SOAP XML格式。

两种格式器都实现了`IFormatter`接口，定义如下：

```
public interface IFormatter
{
    object Deserialize(Stream serializationStream);
    void Serialize(Stream serializationStream, object graph);
    // 更多成员
}

public sealed class BinaryFormatter : IFormatter, ...
{...}
public sealed class SoapFormatter : IFormatter, ...
{...}
```

除了要持久化对象的状态，两种格式器都要将类型的程序集以及版本控制信息持久化到流中，这样才能保证序列化的对象能够被反序列化为正确的类型。然而，这种呈现（`Render`）类型的方式却无法完全满足面向服务的交互，因为面向服务的交互方式要求其他的参与方不仅拥有类型程序集，还要使用.NET。由于它要求客户端和服务能够共享流，因此，使用流也不过是无奈之举。

## WCF格式器

由于经典的.NET格式器功能上的不足，WCF不得不提供自己的面向服务格式器。WCF格式器 `DataContractSerializer` 能够共享数据契约，而不是基本的类型信息。`DataContractSerializer` 定义在 `System.Runtime.Serialization` 命名空间中，例3-1展示了它的一部分定义。

例3-1: `DataContractSerializer` 格式器

```
public abstract class XmlObjectSerializer
{
    public virtual object ReadObject(Stream stream);
    public virtual object ReadObject(XmlReader reader);
    public virtual void WriteObject(XmlWriter writer, object graph);
    public void WriteObject(Stream stream, object graph);
    // 更多成员
}

public sealed class DataContractSerializer : XmlObjectSerializer
{
    public DataContractSerializer(Type type);
    // 更多成员
}
```

通过序列化或者数据契约样式，`DataContractSerializer` 只能捕获对象的状态。注意，`DataContractSerializer` 并没有实现 `IFormatter` 接口。

一般而言，WCF会自动地选择使用 `DataContractSerializer`，不需要开发者直接调用。但是，我们可以使用 `DataContractSerializer` 实现类型与.NET流之间的序列化操作，使用方式与传统的格式器相似。但不同于使用二进制格式器或SOAP格式器，我们需要为 `DataContractSerializer` 的构造函数提供它要操作的类型，因为流没有包含类型的信息：

```
MyClass obj1 = new MyClass();
DataContractSerializer formatter = new DataContractSerializer(typeof(MyClass));

using(Stream stream = new MemoryStream())
{
    formatter.WriteObject(stream, obj1);
    stream.Seek(0, SeekOrigin.Begin);
    MyClass obj2 = (MyClass)formatter.ReadObject(stream);
}
```

使用 `DataContractSerializer` 处理.NET流时，如果输入值的形式为XML，那么我们也可以将XML读取器（XML Reader）和XML编写器联合起来使用。如果是其他类型的介质，例如文件或内存，处理方式则完全不同。

注意在例3-1中，`DataContractSerializer` 的定义使用了无法判断类型的 `object` 对象。这意味着它不具备编译时类型安全，因为构造函数可以接收一种类型，而 `WriteObject()` 方法又会接收另一种类型，然后在 `ReadObject()` 方法中，又将它转换为第三种类型。

要弥补这一缺陷，我们可以定义自己的泛型包装类DataContractSerializer，如例3-2所示。

例3-2：泛型类DataContractSerializer<T>

```
public class DataContractSerializer<T> : XmlObjectSerializer
{
    DataContractSerializer m_DataContractSerializer;

    public DataContractSerializer( )
    {
        m_DataContractSerializer = new DataContractSerializer(typeof(T));
    }
    public new T ReadObject(Stream stream)
    {
        return (T)m_DataContractSerializer.ReadObject(stream);
    }
    public new T ReadObject(XmlReader reader)
    {
        return (T)m_DataContractSerializer.ReadObject(reader);
    }
    public void WriteObject(Stream stream, T graph)
    {
        m_DataContractSerializer.WriteObject(stream, graph);
    }
    public void WriteObject(XmlWriter writer, T graph)
    {
        m_DataContractSerializer.WriteObject(writer, graph);
    }
    //更多成员
}
```

泛型类DataContractSerializer<T>比基于object的DataContractSerializer类更

加安全：

```
MyClass obj1 = new MyClass( );
DataContractSerializer<MyClass> formatter = new
    DataContractSerializer<MyClass>( );

using(Stream stream = new MemoryStream( ))
{
    formatter.WriteObject(stream, obj1);
    stream.Seek(0, SeekOrigin.Begin);
    MyClass obj2 = formatter.ReadObject(stream);
}
```

WCF还提供了NetDataContractSerializer格式器，它是IFormatter接口的多态实现：

```
public sealed class NetDataContractSerializer : IFormatter,...
{...}
```

既然NetDataContractSerializer（译注1）实现了IFormatter接口，因而它与传统

译注1：NetDataContractSerializer类型名并非一个好的定义，由于它实现了IFormatter接口，它更近似于传统的.NET格式器，例如SoapFormatter。如果定义为NetDataContract-Formatter会更好，可以避免与DataContractSerializer混淆。

的.NET格式器相似,除了可以捕获对象状态,还包括类型信息。使用它的方法也和传统的格式器相似:

```
MyClass obj1 = new MyClass( );
IFormatter formatter = new NetDataContractSerializer( );

using(Stream stream = new MemoryStream( ))
{
    formatter.Serialize(stream,obj1);
    stream.Seek(0,SeekOrigin.Begin);
    MyClass obj2 = (MyClass)formatter.Deserialize(stream);
}
```

设计NetDataContractSerializer的目的在于弥补DataContractSerializer的不足。我们可以使用NetDataContractSerializer序列化一个类型,然后使用DataContractSerializer执行反序列化:

```
MyClass obj1 = new MyClass( );
Stream stream = new MemoryStream( );

IFormatter formatter1 = new NetDataContractSerializer( );
formatter1.Serialize(stream,obj1);

stream.Seek(0,SeekOrigin.Begin);

DataContractSerializer formatter2 = new DataContractSerializer(typeof(MyClass));
MyClass obj2 = (MyClass)formatter2.ReadObject(stream);
stream.Close( );
```

NetDataContractSerializer的这种特性有利于版本的兼容,同时也有利于将那些共享了类型信息的旧有代码,迁移到更加具有面向服务特征的程序中。这样,我们就只需要维持类型的数据样式。

## 序列化数据契约

当一个服务操作接收和返回任意类型或参数时,WCF会使用DataContractSerializer对参数进行序列化与反序列化。这意味着只要其他参与方拥有数据样式或数据契约的定义,我们就可以将可序列化类型作为参数或者契约操作的返回值进行传递。所有.NET内建的基本类型都是可序列化的,例如int和string的定义:

```
[Serializable]
public struct Int32 : ...
{...}

[Serializable]
public sealed class String : ...
{...}
```

上一章演示的服务契约之所以能够正常工作,原因正在于此。基本类型在WCF中默认具有数据契约的特性,因为它们的样式都符合同一个行业标准。

为了能够在操作中使用定制类型参数，必须符合两个条件：首先，类型必须是可序列化的；其次，客户端与服务都拥有该类型的本地定义，而且该类型应该具有相同的数据样式。

考虑服务契约 `IServiceContractManager`，它被用来管理一个联系人列表：

```
[Serializable]
struct Contact
{
    public string FirstName;
    public string LastName;
}

[ServiceContract]
interface IServiceContractManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts();
}
```

如果客户端使用了 `Contact` 结构的等效定义，就能够将 `contact` 对象传递给服务。所谓“等效定义”，可以在序列化中能够生成相同数据样式的任何类型定义。例如，客户端就可以使用这样的定义：

```
[Serializable]
struct Contact
{
    public string FirstName;
    public string LastName;

    [NonSerialized]
    public string Address;
}
```

## 数据契约特性

`Serializable` 特性或许已经足够使用了，但用在客户端与服务之间面向服务的交互却还不够理想。不同于 `Serializable` 指明类型的所有成员都是可序列化的，并作为组成类型数据样式的一部分；更好的方式是能够提供一种明确参与（`Opt-In`）的途径，只有那些契约的开发者明确包含的成员才应该放到数据契约中。`Serializable` 特性强制要求数据类型是可序列化的，从而使类型可以被用作契约操作的参数，但它却无法实现将类型作为 WCF 操作参数（类型的“服务”特性）与序列化能力之间的职责分离。`Serializable` 特性不支持类型名和成员名的别名，也无法将一个新类型映射为预定义的数据契约。由于 `Serializable` 特性可以直接操作成员字段，使得封装了字段访问的属性形同虚设。访问字段的最好办法是通过属性添加它们的值，而 `Serializable` 却破坏了属性的封装性。最



后，`Serializable`特性并没有直接支持版本控制（Versioning），因为格式器期望获取版本控制的所有信息。无疑，它导致了版本控制的处理变得举步维艰。

解决之道是为WCF提供新的面向服务特性，这些特性中的首要特性就是定义在`System.Runtime.Serialization`命名空间中的`DataContractAttribute`特性：

```
[AttributeUsage(AttributeTargets.Enum |
    AttributeTargets.Struct |
    AttributeTargets.Class,
    Inherited = false,
    AllowMultiple = false)]
public sealed class DataContractAttribute : Attribute
{
    public string Name
    {get;set;}
    public string Namespace
    {get;set;}
}
```

如果只是在类或结构类型上应用`DataContract`特性，WCF不会序列化类型的成员：

```
[DataContract]
struct Contact
{
    //不会成为数据契约的一部分
    public string FirstName;
    public string LastName;
}
```

`DataContract`特性只能做到将类型参与数据契约中，以指示类型可以被按值编组（Marshal）。如果要序列化类型的成员，必须应用`DataMemberAttribute`特性，定义如下：

```
[AttributeUsage(AttributeTargets.Field|AttributeTargets.Property,
    Inherited = false,AllowMultiple = false)]
public sealed class DataMemberAttribute : Attribute
{
    public bool IsRequired
    {get;set;}
    public string Name
    {get;set;}
    public int Order
    {get;set;}
}
```

我们可以直接将`DataMember`特性应用到字段上：

```
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
```

```
        public string LastName;  
    }  
}
```

或者应用到属性上（可以是显式属性，提供了属性的实现；也可以是自动属性，编译器自动生成内部成员以及访问实现）：

```
[DataContract]  
struct Contact  
{  
    string m_FirstName;  
  
    [DataMember]  
    public string FirstName  
    {  
        get  
        {  
            return m_FirstName;  
        }  
        set  
        {  
            m_FirstName = value;  
        }  
    }  
  
    [DataMember]  
    public string LastName  
    {get;set;}  
}
```

与服务契约相似，数据成员或数据契约的访问限定与WCF之间并没有因果关系。数据契约完全可以包含私有数据成员等内部类型：

```
[DataContract]  
struct Contact  
{  
    [DataMember]  
    string m_FirstName;  
  
    [DataMember]  
    string m_LastName;  
}
```

为简便起见，本章的某些代码都是将DataMember特性直接应用到公有数据成员上。但在实际应用中，我们应该使用属性而不是公有成员。



数据契约的类型与成员均区分大小写。

## 导入数据契约

如果在契约操作中使用数据契约，它就会被发布在服务的元数据中。当客户端使用诸如

Visual Studio 2008之类的工具导入数据契约的定义时，使用的是与它等效的定义，但并非必须是同一个数据契约。区别在于工具的功能，而不是发布的元数据。使用Visual Studio 2008，导入的定义会保留类或者结构原有的类型名，以及原类型的命名空间。如果使用SvcUtil，则只有数据契约会保留命名空间。例如这样的服务端定义：

```
namespace MyNamespace
{
    [DataContract]
    struct Contact
    {...}

    [ServiceContract]
    interface IContactManager
    {
        [OperationContract]
        void AddContact(Contact contact);

        [OperationContract]
        Contact[] GetContacts( );
    }
}
```

导入的定义为：

```
namespace MyNamespace
{
    [DataContract]
    struct Contact
    {...}
}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts( );
}
```

若要重写默认实现，为它指定另外的命名空间，可以通过DataContract特性的Namespace属性设置命名空间的值。工具会区别对待提供的命名空间。例如这样的服务端定义：

```
namespace MyNamespace
{
    [DataContract (Namespace = "MyOtherNamespace")]
    struct Contact
    {...}
}
```

Visual Studio 2008导入的内容完全与定义相同，而SvcUtil导入的定义则发布为：

```
namespace MyOtherNamespace
{
```

```
[DataContract]
struct Contact
{...}
}
```

使用Visual Studio 2008，即使原来的服务端类型并没有定义任何属性，在导入的定义中，仍然会以标记DataMember特性的属性来表示。如果原来的服务端定义直接将DataMember特性应用在字段上，那么在导入的类型定义中，就会添加访问该字段的属性，同时为原来的字段名添加后缀Field。例如这样的服务端定义：

```
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}
```

导入的客户端定义为：

```
[DataContract]
public partial struct Contact
{
    string FirstNameField;
    string LastNameField;

    [DataMember]
    public string FirstName
    {
        get
        {
            return FirstNameField;
        }
        set
        {
            FirstNameField = value;
        }
    }

    [DataMember]
    public string LastName
    {
        get
        {
            return LastNameField;
        }
        set
        {
            LastNameField = value;
        }
    }
}
```

当然，客户端也可以手工修改导入的定义，使它与服务端的定义一致。



即使服务端将DataMember特性应用到私有字段或私有属性上，如下所示：

```
[DataContract]
struct Contact
{
    [DataMember]
    string FirstName
    {get;set;}

    [DataMember]
    string LastName;
}
```

在导入的定义中，原来的私有字段或私有属性都被定义成了公有属性。

如果将DataMember特性应用到属性上，并将它作为服务端数据契约的一部分，那么导入的定义则包含了相同的属性，字段的名称则为属性名加上Field后缀。同时，客户端属性的访问器包装了对字段的访问。例如这样的服务端数据契约：

```
[DataContract]
public partial struct Contact
{
    string m_FirstName;
    string m_LastName;

    [DataMember]
    public string FirstName
    {
        get
        {
            return m_FirstName;
        }
        set
        {
            m_FirstName = value;
        }
    }

    [DataMember]
    public string LastName
    {get; set;}
}
```

导入的定义为：

```
[DataContract]
public partial struct Contact
{
    string FirstNameField;
    string LastNameField;
}
```

```
[DataMember]
public string FirstName
{
    get
    {
        return FirstNameField;
    }
    set
    {
        FirstNameField = value;
    }
}

[DataMember]
public string LastName
{
    get
    {
        return LastNameField;
    }
    set
    {
        LastNameField = value;
    }
}
```

当DataMember特性应用到属性上时（不管是服务还是客户端），该属性必须具有get和set访问器。如果没有，在调用时就会抛出InvalidDataContractException异常。因为当属性自身就是数据成员时，WCF会在序列化和反序列化时使用该属性，使开发者能够将定制逻辑应用到属性中。



不要将DataMember特性既应用到属性上，又应用到相对应的字段上，这会导致导入的成员定义重复。

重要的是，我们要认识到迄今为止介绍的方法都是为了能在服务和客户端上同时应用DataMember特性。当客户端使用DataMember特性（或者本章其他地方介绍的与它相关的特性）时，它会对数据契约产生影响。数据契约既可以通过序列化操作发送参数到服务；又可以通过反序列化使用服务返回的值。两端使用的数据契约可以是等效的数据契约，而不是同一个数据契约，我们甚至可以使用不等效的数据契约，这一点会在后面介绍。客户端对它的数据契约的控制和配置，与服务没有任何关系。

## 数据契约与Serializable特性

服务仍然能够使用只标记了Serializable特性的类型：



```
[Serializable]
struct Contact
{
    string m_FirstName;
    public string LastName;
}
```

在导入如上类型的元数据时，导入的定义会使用DataContract特性。此外，既然Serializable特性只能影响字段，因而每一个可序列化的成员（无论公有还是私有）都是数据成员。所以，在导入定义中包装了的属性名就会沿袭原来的字段名：

```
[DataContract]
public partial struct Contact
{
    string LastNameField;
    string m_FirstNameField;

    [DataMember(...)]
    public string LastName
    {
        ... //访问LastNameField
    }

    [DataMember(...)]
    public string m_FirstName
    {
        ... //访问m_FirstNameField
    }
}
```

客户端同样可以在它的数据契约中使用Serializable特性，从而具有传输型表示形式（Wire Representation）。也即是说，它是可编组的，这与前面的描述相同。



传统的格式器不能序列化只标记了DataContract特性的类型。要序列化这样的类型，必须同时应用DataContract特性和Serializable特性。对于该类型生成的数据契约，效果与只应用了DataContract特性相同，同时，我们仍然需要为需要序列化的成员添加DataMember特性。

## 数据契约与XML序列化

.NET还提供了另外一种序列化机制：XML序列化，它使用了一系列专门的特性。如果我们正在处理的数据类型需要显式地控制XML序列化，那么就应该将XmlSerializer、FormatAttribute特性应用到契约定义中单独的操作上，以指导WCF在运行时使用XML序列化。如果契约的所有操作都需要采用这种序列化形式，我们就可以使用XmlUtil（如第1章所述）的/serializer:XmlSerializer开关，指导它自动地将XmlSerializerFormat特性应用到所有导入的契约中的所有操作上。使用这一开关时需要谨慎，因为它会影响所有的数据契约，包括那些不需要显式控制XML序列化的数据契约。

## 数据契约推断

.NET 3.5的服务包I引入了对数据契约进行推断的支持。如果编组类型是公共类型，且未曾标记DataContract特性，WCF就会自动推断，认为DataContract特性被应用到该类型上，且它的所有公有成员（字段或属性）均被应用了DataMember特性。

例如，给定如下的服务契约定义：

```
public struct Contact
{
    public string FirstName
    {get;set;}

    public string LastName;

    internal string PhoneNumber;

    string Address;
}
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);
    ...
}
```

WCF会推断出一个数据契约，就好似服务契约的开发者定义了一般：

```
[DataContract]
public class Contact
{
    [DataMember]
    public string FirstName
    {get;set;}

    [DataMember]
    public string LastName;
}
```

推断的数据契约会在服务元数据中发布。

如果类型已经包含了DataMember特性（但未包含DataContract特性），则DataMember特性就会被忽略，就像它们不存在一般。如果类型包含了DataContract特性，就没有数据契约会被推断。同样，如果类型是内部的（internal），也不会有数据契约被推断。而且，如果一个类是推断的数据契约，则它的子类自身也必须是推断的，即它们必须是公有的类，且不包含DataContract特性。



推断的数据契约有时候会被称之为POCO，即普通的旧的CLR对象（Plain Old CLR Object）。

在我看来，依赖于数据契约的推断是一种草率的破解方法，它与WCF的大多数内容背道而驰。既然WCF无法从仅有的接口定义中推断出服务契约，或者默认允许事务或可靠性，它就不应该推断数据契约。面向服务在很大程度上偏向于默认“否决”的方式（安全是一个例外），正因为如此，它需要最大限度的封装与解耦。对数据契约显式地使用DataContract特性，就使得你可以利用数据契约的功能，例如版本控制。本书的其余部分都不会使用或者依赖于数据契约的推断。

## 合成的数据契约

定义数据契约时，对于那些本身就是数据契约的成员，我们仍然可以为它们标记DataMember特性，如例3-3所示。

例3-3：一个合成的数据契约

```
[DataContract]
class Address
{
    [DataMember]
    public string Street;

    [DataMember]
    public string City;

    [DataMember]
    public string State;

    [DataMember]
    public string Zip;
}
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;

    [DataMember]
    public Address Address;
}
```

能够将其他数据契约聚合在一起，说明数据契约具有递归的性质。当我们序列化一个合成的数据契约时，DataContractSerializer会跟踪对象图（object graph）中的所有可用的引用，捕获它们的状态。当我们发布一个合成的数据契约时，它所包含的所有数据契约都会被发布。例如，使用与例3-3相同的定义，则服务契约的元数据为：

```
[ServiceContract]
```

```
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts();
}
```

同时，服务契约的元数据还包含了Address结构的定义。

## 数据契约事件

.NET 2.0为可序列化类型引入了对序列化事件的支持，WCF沿袭了这一技术，对数据契约提供了同样的支持。执行序列化和反序列化时，WCF会调用数据契约的指定方法。WCF一共定义了四个序列化与反序列化事件。serializing事件发生在序列化之前，而serialized事件则发生在序列化之后。相似地，deserializing事件发生在反序列化之前，而deserialized事件则发生在反序列化之后。我们可以使用方法特性，将指定的方法当作序列化事件的处理器（Handler），如例3-4所示。

例3-4：应用序列化事件特性

```
[DataContract]
class MyDataContract
{
    [OnSerializing]
    void OnSerializing(StreamingContext context)
    {...}

    [OnSerialized]
    void OnSerialized(StreamingContext context)
    {...}

    [OnDeserializing]
    void OnDeserializing(StreamingContext context)
    {...}

    [OnDeserialized]
    void OnDeserialized(StreamingContext context)
    {...}
    //数据成员
}
```

每个序列化事件的处理方法都必须遵循如下的方法签名：

```
void <MethodName>(StreamingContext context);
```

这是必需的，因为WCF会在内部使用反射与委托去订阅和调用事件处理方法。如果将序列化事件特性（定义在System.Runtime.Serialization命名空间中）应用在与方法签名不符的方法上，WCF就会抛出异常。

StreamingContext属于结构类型，用于告知类型被序列化的原因，但它也可以被WCF的数据契约忽略。

与特性名的含义一致，OnSerializing特性指定方法处理serializing事件，OnSerialized特性则指定方法处理serialized事件。相似的，OnDeserializing特性指定方法处理deserializing事件，而OnDeserialized特性则指定方法处理deserialized事件。

图3-2所示的活动图描述了在序列化期间事件触发的顺序。

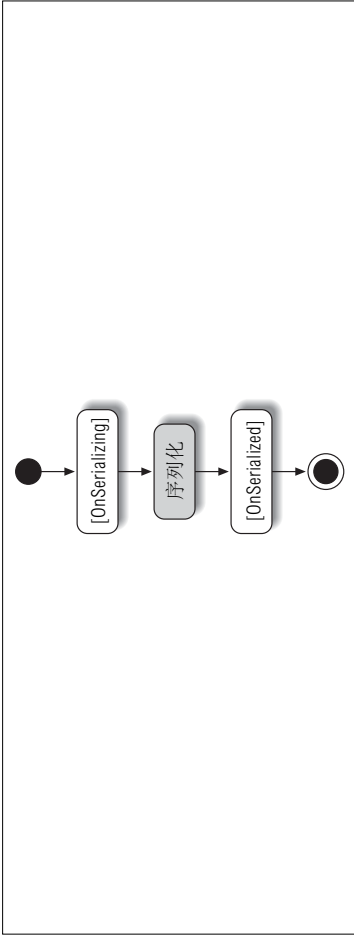


图3-2: 序列化期间的事件

WCF首先触发serializing事件，它会调用相关的事件处理器。接着，WCF序列化对象，最后触发serialized事件，调用它的事件处理器。

图3-3所示的活动图描述了反序列化事件触发的顺序。WCF首先触发deserializing事件，它会调用相应的事件处理器。接着，WCF反序列化对象，最后触发deserialized事件，并调用它的事件处理器。

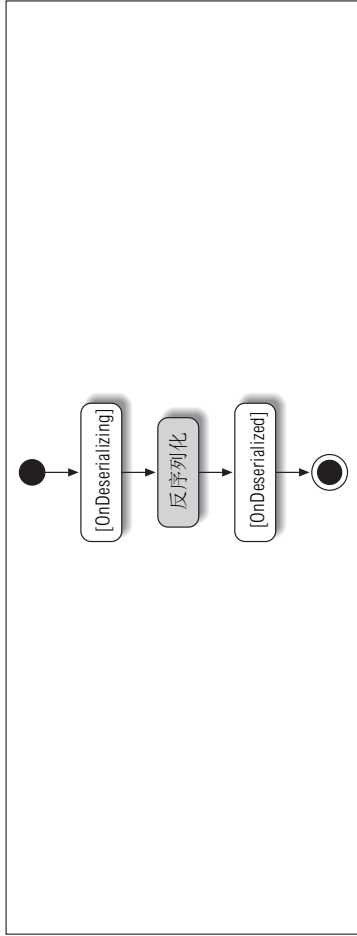


图3-3: 反序列化期间的事件

注意，为了调用 `Deserialize` 事件的处理方法，WCF 必须首先创建一个对象；但是它不会调用数据契约类的默认构造函数。



WCF 禁止将相同的序列化事件特性应用到数据契约类型的多个方法上。遗憾的是，这样就无疑将 `partial` 类型排除在外了，因为 `partial` 类型的每一部分都有可能处理自己的序列化事件。

### 使用 `Deserialize` 事件

在反序列化期间，没有构造函数会被调用，`Deserialize` 事件的处理方法在逻辑上就是反序列化的构造函数。这样做的目的在于它能够在反序列化之前执行某些定制步骤。通常，用于初始化的类成员并不需要被标记为数据成员。那些被标记为数据成员的类成员，在初始化时设置的任何值都是无效的，因为 WCF 会在反序列化期间根据从消息获得的值，对这些成员进行重新设置。此外，反序列化事件的处理方法还能够设置特定的环境变量（如线程本地存储），执行诊断操作，或者通知某些全局的同步事件。如果提供了这样的 `Deserialize` 事件处理方法，则可以让默认的构造函数和事件处理器调用同一个辅助方法，这样在使用常规 .NET 方式实例化类型时，完全可以执行相同的步骤，并在一个单独的地方维护这些代码：

```
[DataContract]
class MyClass
{
    public MyClass( )
    {
        OnDeserializing( );
    }
    [OnDeserializing]
    void OnDeserializing(StreamingContext context)
    {
        OnDeserializing( );
    }
    void OnDeserializing( )
    { ... }
}
```

### 使用 `Deserialize` 事件

在使用已经反序列化的值时，`Deserialize` 事件允许初始化数据契约或者重新声明非数据成员。例 3-5 演示了 `Deserialize` 事件的功能。它使用事件初始化数据库连接。如果没有事件的支持，数据契约是无法实现这样的功能的，因为构造函数永远都不会被调用，则 `connection` 对象将为空。

例 3-5：使用 `Deserialize` 事件初始化不可序列化的资源

[DataContract]



```
class MyDataContract
{
    IDbConnection m_Connection;

    [OnDeserialized]
    void OnDeserialized(StreamingContext context)
    {
        m_Connection = new SqlConnection(...);
    }
    /* 数据成员 */
}
```

## 共享数据契约

在Visual Studio 2008添加一个服务引用时，你必须为每个服务引用提供唯一的新命名空间。导入的类型会定义在这个新的命名空间中。如果为共享了相同数据契约的两个不同服务添加引用，就会出现这个问题，因为你得到了两个不同的类型，在两个不同的命名空间，表示的却是相同的数据契约。然而，默认情况下，如果被客户端引用的任意一个程序集包含的数据契约，与已经暴露在服务元数据的数据契约类型匹配，Visual Studio 2008就不会再次导入。需要再次强调的是，已有的数据契约引用必须是在另一个引用程序集中，而不是在客户端项目自身。这一限制会在未来的Visual Studio版本中提供，而目前最方便的弥补措施与最佳实践则为：将所有共享的数据契约分解到指定的类库中，并让所有的客户端引用该程序集。然后，通过服务引用的高级设置对话框（参见图1-10），可以控制和配置引用程序集（如果存在与有关的共享数据契约进行协调。“Reuse types in referenced assemblies”检查框默认是被选中的，但如果你需要也可以关闭这一功能。顾名思义，你只能共享数据契约，却不能共享服务契约。使用里面的单选按钮，可以让Visual Studio 2008跨所有的引用程序集重用数据契约，或者通过选择列表项限制对特定程序集的共享。

## 数据契约层级

数据契约类可能是另一个数据契约类的子类。WCF要求类层级的每一级数据契约都必须标记DataContract特性，因为该特性是不可继承的：

```
[DataContract]
class Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}

[DataContract]
class Customer : Contact
```

```
{
    [DataMember]
    public int OrderNumber;
}
```

如果类层级中的每个层级没有指定为可序列化或者数据契约，就会在服务装载时引发 `InvalidDataContractException` 异常。WCF 允许开发者在类层级中混合使用 `Serializable` 和 `DataContract` 特性：

```
[Serializable]
class Contact
{...}

[DataContract]
class Customer : Contact
{..}
```

但是，`Serializable` 特性通常会被应用到类层级的基类上，而对于新增的类，则应该使用 `DataContract` 特性。导出一个数据契约层级时，元数据会维持它的层级体系。在使用服务契约的子类时，类层级的每级数据契约定义都会被导出：

```
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddCustomer (Customer customer); //Contact 同时也将被导出
    ...
}
```

## 已知类型

在传统的面向对象编程中，对子类的引用同样也是其基类的引用，因此子类维持了与基类之间的 `Is-A` 关系。任何期待基类引用的方法也可以接收其子类的引用。这种方法导致的直接结果是编译器通过在基类部分之后添加子类的方式，在内存中涵盖子类的状态。

例如 C# 这样的语言允许开发者采取这种方式用子类替换基类（译注 2），但这却不适用于 WCF 操作。默认情况下，我们不能用数据契约的子类去替换基类。考虑如下的服务契约：

```
[ServiceContract]
interface IContactManager
{
    //这里不能接收Customer对象：
    [OperationContract]
    void AddContact (Contact contact);

    //这里不能返回Customer对象：
    [OperationContract]
}
```

译注 2：也就是所谓的 Liskov 替换原则（LSP），它指的是子类型必须能够完全替换其父类型，指对象的继承或者接口的实现。

```
Contact[] GetContacts ( );
}
```

假定客户端同时定义了一个Customer类:

```
[DataContract]
class Customer : Contact
{
    [DataMember]
    public int OrderNumber;
}
```

以下代码能够成功通过编译,但在运行时却会失败:

```
Contact contact = new Customer ( )
{
    ....
};
```

```
ContactManagerClient proxy = new ContactManagerClient ( );
//服务调用失败:
proxy.AddContact (contact);
proxy.Close ( );
```

原因在于我们并没有实际传递对象的引用,而是传递了对象的状态。在前面的例子中,当我们传递Customer对象而不是Contact对象时,服务并不知道它应该反序列化状态的Customer部分。

同样,如果操作返回Customer对象而不是Contact对象,客户端也不知道该如何反序列化它,因为它能识别的类型是Contact,而不是Customer:

```
//////////////////// 服务端 //////////////////////////////////
[DataContract]
class Customer : Contact
{
    [DataMember]
    public int OrderNumber;
}
class CustomerManager : IContactManager
{
    List<Customer> m_Customers = new List<Customer>( );

    public Contact[] GetContacts ( )
    {
        return m_Customers.ToArray ( );
    }
    //其余实现
}
//////////////////// 客户端 //////////////////////////////////
ContactManagerClient proxy = new ContactManagerClient ( );
//调用会失败:
Contact[] contacts = proxy.GetContacts ( );
proxy.Close ( );
```

解决之道是使用KnownTypeAttribute特性（译注3）明确地告知WCF关于Customer类的信息。KnownTypeAttribute特性的定义如下：

```
[AttributeUsage(AttributeTargets.Struct|AttributeTargets.Class,
    AllowMultiple = true)]
public sealed class KnownTypeAttribute : Attribute
{
    public KnownTypeAttribute(Type type);
    //更多成员
}
```

KnownType特性允许开发者指定数据契约能够接收的子类：

```
[DataContract]
[KnownType(typeof(Customer))]
class Contact
{...}
```

```
[DataContract]
class Customer : Contact
{...}
```

在宿主端，如果将KnownType特性应用在基类上，会影响所有的契约与操作。它能跨越所有的服务和终结点，允许服务接收子类，而不是基类。此外，应用KnownType特性就可以在元数据中包含子类，这样，客户端就具有了子类的定义，能够传递子类而不是基类。如果客户端同样将KnownType特性应用在基类的副本对象上，那么它就能够依次接收服务返回的已知子类对象。

## 服务已知类型

使用KnownType特性的缺陷在于它使得范围过于宽泛了。因此，WCF还提供了ServiceKnownTypeAttribute特性，定义为：

```
[AttributeUsage(AttributeTargets.Interface|
    AttributeTargets.Method|
    AttributeTargets.Class,
    AllowMultiple = true)]
public sealed class ServiceKnownTypeAttribute : Attribute
{
    public ServiceKnownTypeAttribute(Type type);
    //更多成员
}
```

译注3：使用KnownTypeAttribute并非一种好的解决方案，因为需要在父类中指定子类，无疑会引入两者之间的耦合度。后文介绍的ServiceKnownTypeAttribute相比较而言，有所改善，它可以应用在服务操作上。但不可避免的，它们都要求父类的开发者都必须事先知道子类的定义，这无疑违背了面向对象设计思想。虽然WCF还提供了配置方式配置已知类型，但这种方式又加大了开发者的负担，尤其是在配置文件中需要指定子类的类型全名，无疑增加了管理配置文件的工作量，并且易于出现错误。总之，在WCF中要实现面向对象的多态，还未能做到最佳。

如果不在基类数据契约使用KnownType特性，也可以将ServiceKnownType特性应用在服务端的指定操作上。而且，只有这样的操作（包括所有支持该契约的服务）才能够接收已知的子类：

```
[DataContract]
class Contact
{...}

[DataContract]
class Customer : Contact
{...}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    [ServiceKnownType(typeof(Customer))]
    void AddContact(Contact contact);


    [OperationContract]
    Contact[] GetContacts();
}
```

其他操作不能接收子类。

当ServiceKnownType特性被应用到契约级时，该契约以及实现该契约的所有服务包含的所有操作都能够接收已知的子类：

```
[ServiceContract]
[ServiceKnownType(typeof(Customer))]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts();
}
```

 不要将ServiceKnownType特性应用到服务类自身。虽然代码可以通过编译，但它只有在该服务契约没有被定义为接口时才有效（我们应尽量避免这样的用法）。如果服务拥有单独的契约定义，那么ServiceKnownType特性对于该服务无效。

无论ServiceKnownType特性是被应用到操作上，还是契约一级，导出的元数据和生成的代理都不会包含它的内容，而只会包含应用KnownType特性的基类（译注4）。例如，这样的服务端定义：

译注4：这意味着，自动生成的元数据与代理的定义是使用KnownType特性来表现已知类型。

```
[ServiceContract]
[ServiceKnownType(typeof(Customer))]  
interface IContactManager  
{...}
```

导入的定义如下：

```
[DataContract]  
[KnownType(typeof(Customer))]  
class Contact  
{...}  
[DataContract]  
class Customer : Contact  
{...}  
[ServiceContract]  
interface IContactManager  
{...}
```

我们可以手动修改客户端的代理类，通过删去基类的 `KnownType` 特性，然后将 `ServiceKnownType` 特性应用到契约的相应层级上，从而准确的反映服务端的语义。

## 多个已知类型

通过同时应用 `KnownType` 和 `ServiceKnownType` 特性，可以重复通知 WCF 需要多个已知类型：

```
[DataContract]  
class Contact  
{...}  
[DataContract]  
class Customer : Contact  
{...}  
[DataContract]  
class Person : Contact  
{...}  
[ServiceContract]  
[ServiceKnownType(typeof(Customer))]  
[ServiceKnownType(typeof(Person))]  
interface IContactManager  
{...}
```

WCF 格式器使用反射收集数据契约的所有已知类型，然后检查提供的参数以查看它是否具有任意一种已知类型。

注意，我们必须明确地将数据契约的所有类层级添加到 `ServiceKnownType` 中，但基类并不包括在内：

```
[DataContract]  
class Contact  
{...}
```



```
[DataContract]
class Customer : Contact
{...}

[DataContract]
class Person : Customer
{...}

[ServiceContract]
[ServiceKnownType(typeof (Customer))]
[ServiceKnownType(typeof (Person))]
interface IContactManager
{...}
```

## 配置已知类型

已知类型的相关特性存在的主要缺陷是它们要求服务或者客户端必须事先知道哪些子类可能会被其他调用方调用。添加一个新的子类的子类必然要求修改代码、重新编译和重新部署。要解决这一问题，WCF允许开发者在服务或客户端的配置文件中配置已知类型，如例3-6所示。我们不仅需要提供类型名，还要提供所在程序集名。

### 例3-6：配置文件中的已知类型

```
<system.runtime.serialization>
  <dataContractSerializer>
    <declaredTypes>
      <add type = "Contact,Host,Version=1.0.0.0,Culture=neutral,
        PublicKeyToken=null">
        <knownType type = "Customer,MyClassLibrary,Version=1.0.0.0,
          Culture=neutral,PublicKeyToken=null"/>
      </add>
    </declaredTypes>
  </dataContractSerializer>
</system.runtime.serialization>
```

若是不依赖于程序集名或版本，则可以使用程序集的友好名称（friendly name）：

```
<add type = "Contact,Host">
  <knownType type = "Customer,MyClassLibrary"/>
</add>
```

在配置文件中包含已知类型，将与KnownType特性应用在数据契约的作用相同，发布的元数据将包含已知类型的定义。

值得注意的是，如果已知类型对于另一个程序集而言是内部（internal）类型，要添加一个已知类型，只有使用配置文件声明它。

## Object与接口

数据契约类或数据契约结构的基类型可以是接口：

```
interface IContact
{
    string FirstName
    {get;set;}
    string LastName
    {get;set;}
}
[DataContract]
class Contact : IContact
{...}
```

只要我们使用ServiceKnownType特性指定了确切的数据类型，就可以在服务契约中使用基接口类型，或者在数据契约中定义基接口类型的数据成员：

```
[ServiceContract]
[ServiceKnownType(typeof(Contact))]
interface IContactManager
{
    [OperationContract]
    void AddContact(IContact contact);

    [OperationContract]
    IContact[] GetContacts();
}
```

不能将KnownType特性应用到基接口上，因为导出的元数据无法包含接口本身。相反，导入的服务契约则是基于object的，同时还包括了数据契约子类或者没有继承关系的结构。

```
//导入的定义：
[DataContract]
class Contact
{...}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    [ServiceKnownType(typeof(object[])]
    void AddContact(object contact);

    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    [ServiceKnownType(typeof(object[])]
    object[] GetContacts();
}
```

即使ServiceKnownType特性最初的定义范围是契约，导入的定义仍然可以将它应用到操作层级上。而且，每个操作都会包括一组所有操作都需要的ServiceKnownType特性，包括针对数组的冗长的服务已知类型的特性。在WCF的预先发布版本中需要这些定义，如今仍然被保留了下来。

我们可以手动修改导入的定义，让它只包含必需的ServiceKnownType特性：

```
[DataContract]
class Contact
{...}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    void AddContact(object contact);

    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    object[] GetContacts();
}
```

或者更好的做法是，如果客户端包含了基接口的定义，又或者对定义进行了分解，就可以使用基接口，而不是object类型。只要添加了从接口到数据契约的继承关系，就会给开发者带来一定程度的类型安全：

```
[DataContract]
class Contact : IContact
{...}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    void AddContact(IContact contact);

    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    IContact[] GetContacts();
}
```

但是，我们不能将导入契约中的object类型替换为具体的数据契约类型，因为两者是不兼容的：

```
//无效的客户端契约
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts();
}
```

## 数据契约等效性

如果两个数据契约具有相同的传输型表示形式（也就是说它们具有相同的信息集样式），就可以认为是等效的。等效的数据契约包括：类型定义相同（但并不必然代表类型的版本相同），或者指向两个不同类型的数据契约，它们的契约名以及成员的名称完全相同。等效的数据契约可以互换，因为WCF允许服务操作与它的数据契约等效的数据契约。

如果要定义一个等效的数据契约，最常用的方法是使用DataContract或者DataMember的Name属性，将数据契约映射为另一个数据契约。在DataContract特性中，Name属性的默认值为类型名，因此如下的两个定义是完全等同的：

```
[DataContract]
struct Contact
{...}

[DataContract(Name = "Contact")]
struct Contact
{...}
```

事实上，数据契约的全名还会包含它的命名空间。如前所述，我们可以分配不同的命名空间。

使用DataMember特性时，Name属性的默认值为成员名，因此如下的两个定义是完全等同的：

```
[DataMember]
string FirstName;

[DataMember(Name = "FirstName")]
string FirstName;
```

通过为数据契约与数据成员分配不同的名称，可以为不同的类型生成等效的数据契约。例如，如下的两个数据契约就是等效的：

```
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}

[DataContract(Name = "Contact")]
struct Person
{
    [DataMember(Name = "FirstName")]
    public string Name;

    [DataMember(Name = "LastName")]
    public string Surname;
}
```

除了具有等同的名称，数据成员的类型也必须匹配。

支持相同的数据契约的类与结构可以互换。



## 序列化顺序

在传统的.NET程序中，子类定义的成员的名称与类型，可以与其基类的私有成员相同。反之，它的子类同样可以如此：

```
class A
{
    string Name;
}
class B : A
{
    string Name;
}
class C : B
{
    string Name;
}
```

如果这样的类层级同时又是数据契约，那么在将子类的实例序列化为消息的时候，就会出现问题，因为消息会包含多份相同名称和类型的数据成员。为了辨别它们，WCF以特定的顺序将数据成员放到消息中。

在类型内部，默认的序列化顺序是简单地按照字母排序，至于整个类层级的顺序，则是自上而下的。在序列化顺序不匹配的情况下，成员则以它们的默认值进行初始化。例如，序列化的一个Customer实例，它的定义如下：

```
[DataContract]
class Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}
[DataContract]
class Customer : Contact
{
    [DataMember]
    public int CustomerNumber;
}
```

成员会按照如下的顺序进行序列化：FirstName、LastName、CustomerNumber。

当然，等效的数据契约必须以相同的顺序序列化和反序列化它们的成员。现在的问题是，如果为契约和成员指定了别名，结合数据契约层级的因素，就有可能破坏序列化的顺序。例如，如下的数据契约与Customer数据契约就是不等效的：

```
[DataContract(Name = "Customer")]
public class Person
{
    [DataMember(Name = "FirstName")]
    public string Name;

    [DataMember(Name = "LastName")]
    public string Surname;

    [DataMember]
    public int CustomerNumber;
}
```

因为它的序列化顺序为CustomerNumber、FirstName、LastName。要解决这一冲突，可以通过设置DataMember特性的Order属性值为WCF提供序列化顺序。Order属性的默认值为-1，这是它默认的WCF顺序，但是，我们也可以为它分配值来指定要求的顺序：

```
[DataContract(Name = "Customer")]
public class Person
{
    [DataMember(Name = "FirstName", Order = 1)]
    public string Name;

    [DataMember(Name = "LastName", Order = 2)]
    public string Surname;

    [DataMember, Order = 3]
    public int CustomerNumber;
}
```

重命名数据成员时，我们必须考虑手动地修改它们的顺序。即使没有重命名，若存在大量的数据成员，这种排序方式就会很快地变得难以控制。所幸，一旦另一个成员的order属性设置了相同的值，WCF会按照成员的字母顺序排序。利用这一原理，我们可以为类层级中同一层级的所有成员设置相同的数字值，或者，最好是将Order值设置为它们在类层级中的层数：

```
[DataContract(Name = "Customer")]
public class Person
{
    [DataMember(Name = "FirstName", Order = 1)]
    public string Name;

    [DataMember(Name = "LastName", Order = 1)]
    public string Surname;

    [DataMember, Order = 2]
}
```



```
public int CustomerNumber;  
}
```

## 版本控制

服务应尽可能地与它们的客户端解耦，特别是涉及到版本控制与技术的问题。任意一个版本的客户端都应该能够调用所有版本的服务。WCF的版本判断与.NET不同，并不借助于版本号（例如程序集中的版本号）。当服务和客户端共享一个数据契约时，一个重要目标是允许服务和客户端各自演化数据契约的版本。要支持这样的解耦，WCF需要支持版本的向前与向后兼容性，而不会共享类型或版本的信息。有三种主要的版本控制场景：

- 新增成员
- 缺失成员
- 双向传递，即新的数据契约与旧版本的数据契约之间相互传递，它同时需要向后与向前的兼容性

默认情况下，版本不会影响数据契约，我们几乎可以忽略版本的兼容性问题。

### 新增成员（译注5）

数据契约最常见的变化是在其中一端添加了新的成员，然后将新的契约发送到旧的客户端或服务。在反序列化这样的数据契约类型时，DataContractSerializer会忽略新增成员。这样，服务和客户端就能够接收包含新增成员的数据，且该新增成员并非原有契约的一部分。例如，可以构建基于如下数据类型的服务：

```
[DataContract]  
struct Contact  
{  
    [DataMember]  
    public string FirstName;  
  
    [DataMember]  
    public string LastName;  
}
```

而客户端却可能发送如下的数据契约：

```
[DataContract]  
struct Contact  
{  
    [DataMember]  
    public string FirstName;  
  
    [DataMember]  
}
```

译注5：指发送方包含了新增成员，默认处理方式忽略新增成员。

```
public string LastName;  
  
[DataMember]  
public string Address;  
}
```

注意，添加新成员然后再忽略这些新成员，会破坏数据契约样式的兼容性。因为，一个服务（或客户端）如果与一个样式兼容，则必须与新的样式兼容。

## 缺失成员 （译注6）

默认情况下，WCF允许服务与客户端双方都可以移除数据契约的成员。即是说，我们能够序列化没有确定成员的类型，然后将它发送到期望获取缺失成员的另一方。虽然在正常情况下，我们可能不会有意地移除成员，但更常见的一种情形是：客户端针对旧的数据契约的定义编写，而与之交互的服务则根据定义了新成员的契约定义编写。当接收成员的DataContractSerializer在消息中无法找到所需信息去反序列化这些成员，则根据成员的默认值进行反序列化。也就是说，将引用类型设置为null，值类型设置为0。实际上，这就像发送方从来没有初始化这些成员一样。这种默认策略允许服务接收包含缺失成员的数据，或者能够返回包含了缺失成员的数据到客户端。如例3-7所示。

例3-7：缺失成员以默认值进行初始化

```
//////////////////////////////////// 服务端 //////////////////////////////////////  
[DataContract]  
struct Contact  
{  
    [DataMember]  
    public string FirstName;  
  
    [DataMember]  
    public string LastName;  
  
    [DataMember]  
    public string Address;  
}  
  
[ServiceContract]  
interface IContactManager  
{  
    [OperationContract]  
    void AddContact(Contact contact);  
    ...  
}  
  
class ContactManager : IContactManager  
{
```

译注6：指发送方缺少成员，默认处理方式是缺失成员赋予其默认值。

```
public void AddContact(Contact contact)
{
    Trace.WriteLine("First name = " + contact.FirstName);
    Trace.WriteLine("Last name = " + contact.LastName);
    Trace.WriteLine("Address = " + (contact.Address ?? "Missing"));
    ...
}
...
////////// 客户端 //////////
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}

Contact contact = new Contact ( )
{
    FirstName = "Juval",
    LastName = "Lowy"
};

ContactManagerClient proxy = new ContactManagerClient ( );
proxy.AddContact(contact);

proxy.Close ( );
```

例3-7的输出结果如下：

```
First name = Juval
Last name = Lowy
Address = Missing
```

因为服务接收到的数据成员Address的值为null，故而调试的跟踪结果为Missing。例3-7的问题是，你不得采取这种方式在每一处手动地弥补使用该数据契约的服务（或其他服务，或其他客户端）。

## 使用OnDeserializing事件

当你希望对使用该数据契约的各个参与方共享弥补的逻辑时，最好是使用OnDeserializing事件，根据本地的实际情形对潜在的缺失数据成员进行初始化。如果消息包含了这些成员的值，就会重写在OnDeserializing事件中的设置值；如果没有，事件处理方法就会提供某些非默认的值（译注7）：

译注7：因为OnDeserializing事件被定义在服务端的数据契约中，事件在客户端设置值之前触发，触发时会的成员设置值。

这一技术的使用如下所示：

```
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;

    [DataMember]
    public string Address;

    [OnDeserializing]
    void OnDeserializing(StreamingContext context)
    {
        Address = "Some default address";
    }
}
```

例3-7的输出结果为：

```
First name = Juval
Last name = Lowy
Address = Some default address
```

## 必备成员

与新增成员采取忽略新成员的方式不同，缺失成员的默认处理有可能导致接收端调用链的操作失败，从而导致大体运行良好的系统产生灾难性的后果。原因在于缺失的成员有可能是正确执行操作的必要条件。如果成员是缺失的，可以通过将 `DataMember` 特性的 `IsRequired` 属性值设置为 `true`，避免 WCF 调用这些操作以至于调用失败。

```
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;

    [DataMember(IsRequired = true)]
    public string Address;
}
```

`IsRequired` 的默认值为 `false`，即忽略缺失成员。如果消息中的成员被标记为必备成员，当接收端的 `DataContractSerializer` 无法找到所需的信息进行反序列化时，就会取消这次调用，发送端会引发 `NetDispatcherFaultException` 异常。例如，如果服务端的数  
据契约为例3-7中的定义，同时 `Address` 成员又被标记为必备，则调用就不会到达服务。事

实上，一个必备的特定成员会被发布到服务元数据中，当它被导入到客户端时，生成的代理定义就会包含所需的成员。

客户端和服务都能够将它们的数据契约中的部分或所有数据成员标记为必备，彼此之间是完全独立的。被标记为必备的成员越多，则与服务或客户端之间的交互就越安全，但这却是以牺牲灵活性与版本兼容性为代价的。

当拥有一个必备新成员的数据契约被发送到接收方时，即使接收方无法识别该成员，这样的调用实际上仍然是有效的，同时也会允许这样的传递。换句话说，如果在版本2（简称为V2）的数据契约中，新增成员的IsRequired值被设置为true，那么即使接收端期望获得版本1（简称V1）的数据契约，并且V1并不包含这些新增成员，我们仍然可以将V2发送给它，且该新增成员会被忽略。IsRequired只能影响到包含了缺失成员的V2数据契约一方。

假定V1不知道V2新增的成员，表3-1列出了不同版本号的数据契约之间的交互，在对应的IsRequired属性值的前提下，是允许还是禁止。

表3-1：必备成员的版本兼容性（译注8）

是否必备	V1到V2	V2到V1
False	Yes	Yes
True	No	Yes

V1到V2：代表了缺失成员的情况。如果IsRequired为false，则交互正常，对于缺失成员则设置为默认值。如果IsRequired为true，就会抛出异常，消息不能正常发送。

V2到V1：代表了新增成员的情况。不管IsRequired的值为true还是false，WCF均以忽略新成员的方式进行交互，交互正常。

值得关注的是，若依赖于必备成员则必须处理可序列化类型。虽然在默认情况下，可序列化类型与缺失成员存在冲突，那么在导出它们的时候，生成的数据契约所包含的数据成员都是必备的。例如Contact定义：

```
[Serializable]
struct Contact
{
    public string FirstName;
    public string LastName;
}
```

导出的元数据表示形式为：

```
[DataContract]
```

译注8：假定V2包含了V1的所有数据成员，同时还定义了新增成员。则表3-1涵盖了版本的几种情况。

```
struct Contact
{
    [DataMember(IsRequired = true)]
    public string FirstName
    {get;set;}

    [DataMember(IsRequired = true)]
    public string LastName
    {get;set;}
}
```

为了保证标记了Serializable特性和DataContract特性的数据契约具有相同的版本兼容性，可以将OptionalField特性应用到可选成员上。例如这样的Contact定义：

```
[Serializable]
struct Contact
{
    public string FirstName;

    [OptionalField]
    public string LastName;
}
```

导出的元数据表示形式为：

```
[DataContract]
struct Contact
{
    [DataMember(IsRequired = true)]
    public string FirstName
    {get;set;}

    [DataMember]
    public string LastName
    {get;set;}
}
```

## 版本控制的双向传递

目前我们所讨论的版本兼容技术，包括忽略新增成员以及默认的缺失成员处理，都不是最理想的：虽然它们支持点对点客户端到服务的调用，但却不能广泛地支持交互的场景。考虑图3-4所示的两种交互方式：

在第一种交互方式中，客户端基于一个新的数据契约构建，该数据契约包含了新增成员。客户端将数据契约传递到服务A，而且服务A并不知道新增成员。然后，服务A再将数据传递给服务B，并且服务B能够识别新的数据契约。但是，从服务A传递到服务B的数据并不会包含新增成员，因为它们并不属于服务A的数据契约，因此客户端在进行反序列化时，这些成员会被阻止。相似的情形发生在第二种交互方式中，客户端能够识别包含新增成员的新数据契约，而服务C却只能识别没有包含新增成员的旧数据契约。服务C返回给客户端



的数据并不包含新增成员。

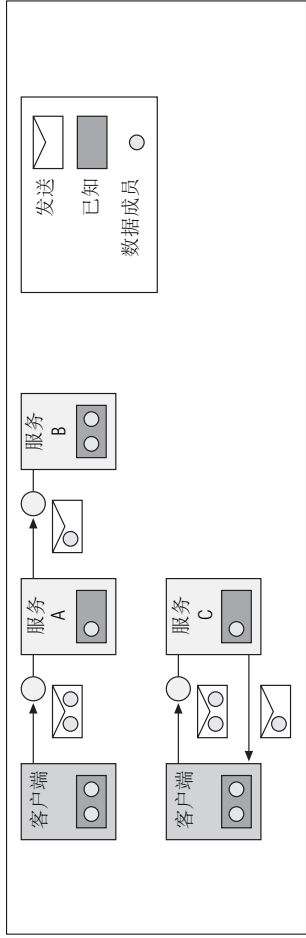


图3-4：版本的双向传递可能会影响整体的交互

这种从新到旧再到新的交互方式称为版本的双向传递。WCF支持对这种方式的处理，方法是允许一个仅能识别旧契约的服务（或客户端）只传递定义在新契约的成員的状态，而不会阻止它们。问题是如何支持服务/客户端在不知道新增成員的情况下，序列化和反序列化这些未知的没有样式的成員？以及应该将它们存储在调用之间的哪一个地方？WCF的解决方案是让数据契约类型实现IExtensibleDataContract接口，它的定义如下所示：

```
public interface IExtensibleDataContract
{
    ExtensionDataContract ExtensionData
    {get;set;}
}
```

IExtensibleDataContract接口定义了一个单独的属性，类型为ExtensionDataContract。无需考虑ExtensionDataContract的确切定义，因为开发者绝不会直接使用该类型。ExtensionDataContract包含了一个内部链表，用以存储对象引用与类型信息，同时也用来存储未知的数据成員。换句话说，如果数据契约类型支持IExtensibleDataContract，那么当消息中无法识别的新增成員可以使用时，它们就会被反序列化，并存储于链表中。当服务（或者客户端）发出向外的调用时，传递旧的数据契约类型，现在，该类型会在ExtensionDataContract属性中包含未知的数据成員，这些未知成員将被按序序列化到消息中。如果接收端能够识别新的数据契约，就会获得一个有效的不包含任何缺失成員的新数据契约。例3-8演示了实现IExtensibleDataContract接口的方法。我们可以看到，实现的方法非常简单：我们只需要为它添加一个ExtensionDataContract自动属性，并进行显式接口实现。

例3-8：实现IExtensibleDataContract

```
[DataContract]
class Contact : IExtensibleDataContract
```

```
{
    ExtensionDataObject IExtensibleDataObject.ExtensionData
    {get;set;}

    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}
```

### 样式兼容性

实现IExtensibleDataObject接口可以支持双向传递。然而，如果两个服务支持的数据契约存储在版本的差异，若要成功实现这两个服务之间的交互就会存在一定的弊端。因此，在某些复杂的应用场景中，服务可能需要禁用双向传递，并将自己的数据契约版本强加于它的下游服务。使用ServiceBehavior特性（在下一章将深入讨论），服务要求WCF通过IExtensibleDataObject接口重写对未知成员的处理，并忽略它们，即使数据契约支持IExtensibleDataObject接口。ServiceBehavior特性提供了Boolean类型的属性IgnoreExtensionDataObject，定义如下：

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute,...
{
    public bool IgnoreExtensionDataObject
    {get;set;}
    //更多成员
}
```

IgnoreExtensionDataObject的默认值为false。设置为true则可以保证服务使用的所有数据契约包含的所有未知数据成员都会被忽略：

```
[ServiceBehavior(IgnoreExtensionDataObject = true)]
class ContactManager : IContactManager
{...}
```

如果使用Visual Studio 2008导入数据契约，生成的数据契约类型总是支持IExtensibleDataObject接口，即使原来的数据契约并不具备这样的特性。定义数据契约的最佳实践是让数据契约实现IExtensibleDataObject接口，同时避免将IgnoreExtensionDataObject值设置为false。IExtensibleDataObject实现了服务与它的下游服务之间的解耦，允许它们完成单独演变。



在处理已知类型时，不必实现IExtensibleDataObject，因为子类在反序列化时不会导致数据缺失（译注9）。

译注9：意即只要父类实现了IExtensibleDataObject接口，子类就不必实现它。

## 枚举

枚举类型的定义总是支持序列化的。当我们定义一个新的枚举时，不必应用DataContract特性，就可以在数据契约中使用它，如例3-9所示。数据契约隐式地包含了枚举对象的所有值。

例3-9：在数据契约中使用枚举

```
enum ContactType
{
    Customer,
    Vendor,
    Partner
}

[DataContract]
struct Contact
{
    [DataMember]
    public ContactType ContactType;

    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}
```

如果要将确定的枚举值排除于数据契约之外，首先需要为枚举类型标记DataContract特性。然后，再将那些我们希望包含在枚举数据契约的枚举值，明确地标记为EnumMemberAttribute特性。EnumMember特性的定义如下：

```
[AttributeUsage(AttributeTargets.Field, Inherited = false)]
public sealed class EnumMemberAttribute : Attribute
{
    public string Value
    {
        get; set;
    }
}
```

没有标记EnumMember特性的枚举值不属于该枚举的数据契约。例如这样的枚举类型：

```
[DataContract]
enum ContactType
{
    [EnumMember]
    Customer,

    [EnumMember]
    Vendor,

    //不会成为数据契约的一部分
    Partner
}
```

生成的传输型表示形式为：

```
enum ContactType
{
    Customer,
    Vendor
}
```

EnumMember特性还有另外一个用途，就是通过它的Value属性，为现有的枚举数据契约的枚举值设置确切的别名。例如这样的枚举：

```
[DataContract]
enum ContactType
{
    [EnumMember(Value = "MyCustomer")]
    Customer,

    [EnumMember]
    Vendor,

    [EnumMember]
    Partner
}
```

生成的传输型表示形式如下：

```
enum ContactType
{
    MyCustomer,
    Vendor,
    Partner
}
```

EnumMember特性只对本地定义有效。发布元数据时（或在客户端定义它时），生成的数据契约不会包含EnumMember特性，而只会使用最终生成的枚举定义。

## 委托与数据契约

所有的委托定义都被编译到可序列化的类中，理论上，委托相当于数据契约类型的成员变量：

```
[DataContract]
class MyDataContract
{
    [DataMember]
    public EventHandler MyEvent;
}
```

事件同样如此（注意field限定符的使用）：

```
[DataContract]
class MyDataContract
{

```

```
[field:DataMember]
public event EventHandler MyEvent;
}
```

然而，实际上当数据契约引用一个自定义委托时，导入的数据契约将包含一个无效的委托定义。虽然可以手动修改定义，但更大的问题是在序列化一个包含了委托成员变量的对象时，委托的内部调用列表也会被序列化。在大多数情况下，对于服务和客户端而言，这并非理想的结果，因为列表具体的结构是本地的，客户端或服务无法跨服务边界共享委托列表的结构。此外，我们不能保证内部列表中的目标对象都是可序列化的，或者都是有效的数据契约。这会导致序列化的操作时而成功，时而失败。

要解决这一缺陷，最简单的办法是不要将 `DataMember` 特性应用到委托上。如果数据契约是可序列化类型，我们应该显式地将委托排除在数据契约之外：

```
[Serializable]
public class MyClass
{
    [NonSerialized]
    public EventHandler MyEvent;
}
```

## 数据集与数据表

在客户端与服务之间交换的数据契约中，最常见的一种数据契约类型是数据库的数据。在 .NET 中，与数据库交互的常见方式是通过 ADO.NET 的数据集或者数据表类型。应用程序能够使用 `DataSet` 和 `DataTable` 类型，或者使用 Visual Studio 2008 的数据访问管理工具生成类型安全的派生对象。

`DataSet` 和 `DataTable` 类型是可序列化的（标记了 `Serializable` 特性）：

```
[Serializable]
public class DataSet : ...
{...}

[Serializable]
public class DataTable : ...
{...}
```

这意味着我们能够定义有效的服务契约，接收或返回数据表或数据集：

```
[DataContract]
struct Contact
{...}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
```

```
void AddContact(Contact contact);

[OperationContract]
void AddContacts(DataTable contacts);

[OperationContract]
DataTable GetContacts();
}
```

我们还可以在契约中使用DataSet和DataTable的类型安全的子类。例如，假定数据库中有一个数据表ContactsDataTable，它包含了联系人的数据，数据列包括FirstName和LastName。使用Visual Studio 2008，可以生成一个类型安全的数据集MyDataSet，它包含了一个嵌套类ContactsDataTable，以及一个类型安全的行对象和数据适配器对象，如例3-10所示：

例3-10：类型安全的数据集和数据表

```
[Serializable]
public partial class MyDataSet : DataSet
{
    public ContactsDataTable Contacts
    {get;}

    [Serializable]
    public partial class ContactsDataTable : ...
    {
        public void AddContactsRow(ContactsRow row);
        public ContactsRow AddContactsRow(string FirstName, string LastName);
        //更多成员
    }

    public partial class ContactsRow : DataRow
    {
        public string FirstName
        {get;set;}

        public string LastName
        {get;set;}
        //更多成员
    }
    //更多成员
}

public partial class ContactstblAdapter : Component
{
    public virtual MyDataSet.ContactsDataTable GetData();
    //更多成员
}

在服务契约中，可以使用类型安全的数据表：
```

```
[DataContract]
struct Contact
```



```
{...}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    void AddContacts(MyDataSet.ContactsDataTable contacts);

    [OperationContract]
    MyDataSet.ContactsDataTable GetContacts()
}

//无效的定义
[OperationContract]
void AddContact(MyDataSet.ContactsRow contact);
```



数据行自身不支持序列化，因此，我们不能在操作中使用它以及它的类型安全的子类：

服务发布的元数据包含了类型安全的数据表。将它导入到客户端时，*SvcUtil*和Visual Studio能够智能地重新生成类型安全的数据表。代理文件不仅包含了数据契约，还包含了相关的代码。如果客户端已经包含了类型安全数据表的本地定义，也可以删去代理文件中的定义。

## 使用数组，而不是数据表

对于WCF的客户端与服务而言，虽然可以通过ADO.NET和Visual Studio工具使用DataSet、DataTable以及它们的类型安全的派生对象，但这种方式过于繁琐。而且，这些数据访问类型都是特定的.NET类型。在序列化时，它们生成的数据契约样式过于复杂，很难与其他平台进行交互。在服务契约中使用数据表或者数据集还存在一个缺陷，那就是它可能暴露内部的数据结构，而客户端却未必关心内部数据存储的要件，例如ID、主键以及外键关系。客户端可能只关心表中信息的子集。同时，将来对数据库样式的修改会影响到客户端，因而定义了服务契约（以内在的数据表方式）的任意服务必须持有不变的数据表样式（这很难做到）。除非正在设计一个数据访问服务，通常暴露数据的操作比暴露数据本身要好得多。简而言之，跨越服务边界发送数据表并非一个好主意。

如果需要传递数据本身，最好的方法是使用与数据结构无关的类型，例如数组类型，并将单个的列转变为某些封装了原有样式的数据契约。

## 转变旧的数据表

从最早的Visual Studio到Visual Studio 2008生成的类型安全的数据表都派生自DataTable类，该类并不具备遍历表的类型安全方式，也不能将其转换为数组：

```
[Serializable]
public partial class ContactsDataTable : DataTable, IEnumerable
{...}

public static class DataTableHelper
{
    public static T[] ToArray<R,T> (this DataTable table, Func<R,T> converter)
        where R : DataRow;
}
```

为了简化将数据表转换为数组的操作，可以使用我设计的DataTableHelper类，定义如下：

DataTableHelper为DataTable定义了一个扩展方法ToArray()。

DataTableHelper类需要一个转换器Converter，它能够将表中的数据列转换为数据契约。

DataTableHelper还添加了编译时和运行时的类型安全验证。例3-11演示了DataTableHelper的用法。

例3-11：使用DataTableHelper

```
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    Contact[] GetContacts( );
    ...
}

class ContactManager : IContactManager
{
    public Contact[] GetContacts( )
    {
        ContactsTableAdapter adapter = new ContactsTableAdapter( );
        MyDataSet.ContactsDataTable contactsTable = adapter.GetData( );

        Func<MyDataSet.ContactsRow, Contact> converter = (row) =>
        {
            return new Contact( )
            {
                FirstName = row.FirstName,
                LastName = row.LastName
            };
        };

        return contactsTable.ToArray(converter);
    }
}
```

```
        // 其余实现  
    }
```

在例3-11中, `GetContacts()` 方法使用了类型安全的数据表适配器 `ContractsTableAdapter` (参见例3-10), 通过它获取数据库的记录。该数据库是由类型安全的数据表 `MyDataSet.ContractsDataTable` 构成的。然后, `GetContacts()` 方法定义了一个 `lambda` 表达式, 通过它将类型安全的数据行 `MyDataSet.ContractsRow` 实例转换为一个 `Contact` 实例。最后, `GetContacts()` 调用 `ToArray()` 扩展方法, 传递参数为数据表与 `converter`。

例3-12给出了 `DataTableHelper.ToArray()` 方法的实现。

例3-12: `DataTableHelper` 类

```
public static class DataTableHelper  
{  
    public static T[] ToArray<R,T>(this DataTable table, Func<R,T> converter)  
        where R : DataRow  
    {  
        // 校验T是否标记了 [DataContract] 或者 [Serializable]  
        Debug.Assert(IsDataContract(typeof(T)) || typeof(T).IsSerializable);  
  
        if (table.Rows.Count == 0)  
        {  
            return new T[]{};  
        }  
  
        // 检验表是否包含了正确的行  
        Debug.Assert(MatchingTableRow<R>(table));  
  
        return table.Rows.Cast<R>().Select(converter).ToArray();  
    }  
    static bool IsDataContract(Type type)  
    {  
        object[] attributes =  
            type.GetCustomAttributes(typeof(DataContractAttribute), false);  
        return attributes.Length == 1;  
    }  
    static bool MatchingTableRow<R>(DataTable table)  
    {  
        if (table.Rows.Count == 0)  
        {  
            return true;  
        }  
        return table.Rows[0] is R;  
    }  
}
```

`DataTableHelper.ToArray()` 方法首先使用了LINQ扩展方法 `Cast()` 将每一行 (相当于一个对象) 转换为 `R` 对象, 然后使用 `Select()` 查询将行对象的集合转换 `T` 对象集合, 最后, 再将集合转换为数组。

DataTableHelper类增加了类型安全。在编译时，它将类型参数R限制为DataRow类型。在运行时，DataTableHelper会验证类型参数T是否标记了DataContract特性或Serializable特性。通过辅助方法IsDataContract()可以检验DataContract特性，它利用了反射去寻找相关的特性。验证Serializable特性，则可以检查类型是否设置了IsSerializable位。如果数据表为空，则ToArray()方法返回一个空数组。最后，ToArray()还要确保提供的数据表包含的行是否属于R类型。这一操作通过MatchingTableRow()方法获得表的第一行数据，以检验它的类型。

### 转换新的数据表

Visual Studio 2008引入了强类型查询与LINQ的支持。使用Visual Studio 2008生成一个数据表时，数据表继承自TypedTableBase<T>:

```
[Serializable]
public partial class ContactsDataTable : TypedTableBase<ContactsRow>
{...}
```

如此，使用LINQ直接将表转换为数组就变得轻而易举。使用与例3-11相同的定义，我们可以这样编程:

```
public Contact[] GetContacts()
{
    Func<CustomersDataSet.ContactsRow, Contact> converter = (row) =>
    {
        return new Contact()
        {
            FirstName = row.FirstName,
            LastName = row.LastName
        };
    };

    return contactsTable.Select(converter).ToArray();
}

甚至:

public Contact[] GetContacts()
{
    return contactsTable.Select(row => new Contact() {
        FirstName = row.FirstName, LastName = row.LastName}).ToArray();
}
```

## 使用LINQ to SQL

若不使用表，则可以使用LINQ to SQL与Visual Studio 2008生成强类型的数据上下文 (data context)。但是，上下文中的强类型对象集合既没有标记Serializable，也没有标记DataContract特性，这就迫使你不得使用数据契约推断，如例3-13所示。

例3-13：使用LINQ to SQL与数据契约推断

```
public partial class ContactsDataContext : DataContext
{
    public Table<Contact> Contacts
    {
        get;
    }
    // 更多成员
}

public partial class Contact
{
    public string FirstName
    {
        get; set;
    }

    public string LastName
    {
        get; set;
    }
}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    Contact[] GetContacts( );
    ...
}

class ContactManager : IContactManager
{
    public Contact[] GetContacts( )
    {
        using (ContactsDataContext context = new ContactsDataContext( ))
        {
            return context.Contacts.ToArray( );
        }
    }
}
```

如果不使用数据契约推断（或者不改变机器生成的代码），我们则可以将强类型的机器生成的代码转换为管理的数据契约，如例3-14所示，它使用了例3-13相同的定义。

例3-14：使用LINQ to SQL，但不使用数据契约推断

```
[DataContract(Name = "Contact")]
class MyContact
{
    [DataMember]
    public string FirstName
    {
        get; set;
    }

    [DataMember]
    public string LastName
    {
        get; set;
    }
}

[ServiceContract]
```

```
interface IContactManager
{
    [OperationContract]
    MyContact[] GetContacts ( );
    ...
}

class ContactManager : IContactManager
{
    public MyContact[] GetContacts ( )
    {
        using (ContactsDataContext context = new ContactsDataContext ( ))
        {
            return context.Contacts.Select(row => new MyContact ( ){
                FirstName = row.FirstName, LastName = row.LastName}).ToArray ( );
        }
    }
}
```

## 泛型

我们不能定义包含了泛型类型参数的数据契约。泛型是.NET的特定技术，使用它们可能会与WCF的面向服务本质发生冲突。但是，我们仍然可以在数据契约中使用限定的泛型类型，只要在服务契约中指定了类型参数，并且指定的类型参数具有有效的数据契约，如例3-15所示。

例3-15：使用限定的泛型类型

```
[DataContract]
class MyClass<T>
{
    [DataMember]
    public T MyMember;
}

[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod(MyClass<int> obj);
}
```

在导入如例3-15所示的数据契约的元数据时，导入类型会将所有的类型参数替换为指定的类型，数据契约则被重命名为如下格式：

<原名>Of<类型参数名><散列值>

使用与例3-15相同的定义，则导入的数据契约与服务契约如下所示：

```
[DataContract]
class MyClassOfint
{
}
```



```
int MyMemberField;

[DataContract]
public int MyMember
{
    get
    {
        return MyMemberField;
    }
    set
    {
        MyMemberField = value;
    }
}

[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod(MyClassOfint obj);
}
```

如果服务契约不是使用int类型，而是自定义类型如SomeClass：

```
[DataContract]
class SomeClass
{...}

[DataContract]
class MyClass<T>
{...}

[OperationContract]
void MyMethod(MyClass<SomeClass> obj);

导出的数据契约应该是这样：

[DataContract]
class SomeClass
{...}

[DataContract]
class MyClassOfSomeClassMTRdqN6P
{...}

[OperationContract(...)]
void MyMethod(MyClassOfSomeClassMTRdqN6P obj);
```

MTRdqN6是根据泛型类型参数与命名空间生成的散列值，它几乎是一个唯一标识的值。不同的数据契约和命名空间会生成不同的散列值。引入散列值能够降低潜在的冲突，因为其他的契约可能会使用同名的类型参数。如果泛型类型参数是基本类型，则不必创建散列值，因为int类型是保留关键字，MyClassOfint的定义是唯一的。

在大多数情况下，引入散列值未免显得过于谨慎。我们完全可以通过数据契约的Name属性为导出的数据契约指定不同的名字。例如，如下的服务端数据契约：

```
[DataContract]
class SomeClass
{...}

[DataContract(Name = "MyClass")]
class MyClass<T>
{...}

[OperationContract]
void MyMethod(MyClass<SomeClass> obj);
```

导出的数据契约为：

```
[DataContract]
class SomeClass
{...}

[DataContract]
class MyClass
{...}

[OperationContract]
void MyMethod(MyClass obj);
```

然而，这种做法可能会带来意义上的模糊不清，因为两个不同的自定义泛型类型可能会生成相同的类型名。

如果要建立泛型类型参数名与数据契约名之间的关联，可以使用{<数字>}标识符，表示其中的数字就是类型参数的序号。例如，给定服务端的定义：

```
[DataContract]
class SomeClass
{...}

[DataContract(Name = "MyClassOf{0}{1}")]
class MyClass<T, U>
{...}

[OperationContract]
void MyMethod(MyClass<SomeClass, int> obj);
```

导出的定义为：

```
[DataContract]
class SomeClass
{...}

[DataContract]
class MyClassOfSomeClassint
{...}
```

```
[OperationContract(...)]  
void MyMethod(MyClassOfSomeClassint obj);
```



编译时不会检验类型参数指定的数字。如果不匹配会产生运行时异常。

最后，我们可以在数字后添加#符号，生成唯一的散列值。例如，这样的数据契约定义：

```
[DataContract]  
class SomeClass  
{...}  
  
[DataContract(Name = "MyClassOf{0}{#}")]  
class MyClass<T>  
{...}  
  
[OperationContract]  
void MyMethod(MyClass<SomeClass> obj);
```

导出的定义为：

```
[DataContract]  
class SomeClass  
{...}  
  
[DataContract]  
class MyClassOfSomeClassWTRdqN6P  
{...}  
  
[OperationContract]  
void MyMethod(MyClassOfSomeClassWTRdqN6P obj);
```

## 集合

在.NET中，各种类型的集合均实现了IEnumerable或IEnumerable<T>接口。所有内建的.NET集合，例如数组、列表和栈都实现了这些接口。一个数据契约的数据成员可以是一个集合类型，服务契约也可以定义直接与集合交互的操作。因为.NET集合是.NET特有的，WCF不能在服务元数据中公开它们；但考虑到集合类型非常有用，WCF专门为集合提供了编组原则。

定义服务操作时，不管我们使用下列哪一种集合接口：IEnumerable<T>、IList<T>和ICollection<T>（译注10），它们的传输型表示形式都使用了数组。例如，如下的服务契约定义与实现：

```
[ServiceContract]  
interface IContactManager  
{  
    [OperationContract]
```

译注10：支持的接口均为泛型集合。只有如此，才能够转换为类型参数指代的数组类型。

```
IEnumerable<Contact> GetContacts ( );
...
}
class ContactManager : IContactManager
{
    List<Contact> m_Contacts = new List<Contact>( );

    public IEnumerable<Contact> GetContacts ( )
    {
        return m_Contacts;
    }
    ...
}
```

导出结果为：

```
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    Contact[] GetContacts ( );
}
```

## 具体集合类型

如果契约中的集合为具体的集合类型（不是接口），而且属于可序列化集合（标记为 `Serializable` 特性而不是 `DataContract` 特性），那么，WCF 就能够自动地将集合规范为数组类型，只要提供的集合包含了 `Add()` 方法，并符合如下的其中一种方法签名：

```
public void Add(object obj); //使用 IEnumerable 的集合
public void Add(T item);    //使用 IEnumerable<T> 的集合
```

例如，考虑这样的契约定义：

```
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    List<Contact> GetContacts ( );
}
```

List 类的定义为：

```
public interface ICollection<T> : IEnumerable<T>
{...}
public interface IList<T> : ICollection<T>
{...}
[Serializable]
public class List<T> : IList<T>
{
    public void Add(T item);
}
```

```
//更多成员
}

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts();
}
```

因为它是一个有效的集合，并且包含了Add()方法，则契约生成的表示形式为：

List<Contacts>会被编组为Contact[]数组类型。服务仍然可以返回List<Contacts>，但客户端只会与数组交互，如例3-16所示。

例3-16：将列表编组为一个数组

```
//////////////////// 服务端 //////////////////////////////////////
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    List<Contact> GetContacts();
}
//服务实现
class ContactManager : IContactManager
{
    List<Contact> m_Contacts = new List<Contact>();

    public void AddContact(Contact contact)
    {
        m_Contacts.Add(contact);
    }

    public List<Contact> GetContacts()
    {
        return m_Contacts;
    }
}
//////////////////// 客户端 //////////////////////////////////////
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts();
}
```

```
class ContactManagerClient : ClientBase<IContactManager>, IContactManager
{
    public Contact[] GetContacts( )
    {
        return Channel.GetContacts( );
    }
    //更多成员
}
//客户端代码
ContactManagerClient proxy = new ContactManagerClient( );
Contact[] contacts = proxy.GetContacts( );
proxy.Close( );
```

注意，为了能够将它编组为数组，集合必须包含Add()方法，但并不需要实现Add()方法。

## 自定义集合

并非只有内建的集合类型才具有自动编组为数组的能力，任何自定义集合只要符合相同的先决条件，都可以被编组为数组，如例3-17所示。例中，集合MyCollection<string>被编组为string[]。

例3-17：将自定义集合编组为数组

```
//////////////////////////////////// 服务端 //////////////////////////////////////
[Serializable]
public class MyCollection<T> : IEnumerable<T>
{
    public void Add(T item)
    {}

    IEnumerator<T> IEnumerable<T>.GetEnumerator( )
    {
        ...
        //其余实现
    }
    [ServiceContract]
    interface IMyContract
    {
        [OperationContract]
        MyCollection<string> GetCollection( );
    }
    ////////////////////////////////////// 客户端 //////////////////////////////////////
    [ServiceContract]
    interface IMyContract
    {
        [OperationContract]
        string[] GetCollection( );
    }
}
```

## CollectionDataContract特性

前面所示的编组为具体集合类型的机制并不理想。首先，它要求集合必须是可序列化的，

而不是使用面向服务的DataContract特性。而且，当服务的一方处理集合类型时，另一方若处理数组类型，就会导致双方语义的不对称性。集合具有数组所不具备的得天独厚的优势，否则我们不会优先考虑使用它。而且，对于集合是否包含Add()方法，或者集合是否支持IEnumerable和IEnumerable<T>接口，并没有编译时或运行时的有效验证。如果集合不符合条件，就会导致数据契约无法工作。WCF的解决方案是使用另外一个专门的特性CollectionDataContractAttribute，定义如下：

```
[AttributeUsage(AttributeTargets.Struct|AttributeTargets.Class,Inherited = false)]
public sealed class CollectionDataContractAttribute : Attribute
{
    public string Name
    {get;set;}
    public string Namespace
    {get;set;}
    //更多成员
}
```

CollectionDataContract特性与DataContract特性相似，它不能序列化集合。将它应用到集合上时，CollectionDataContract会将集合当作一个泛型的链表类型公开给客户端。链表可能不会对原来的集合执行任何操作，但它会提供一个类似接口的集合类型，而不是数组。

例如这样的集合定义：

```
[CollectionDataContract(Name = "MyCollectionOf{0}")]
public class MyCollection<T> : IEnumerable<T>
{
    public void Add(T item)
    {}

    IEnumerator<T> IEnumerable<T>.GetEnumerator( )
    {...}
    //其余实现
}
```

则服务端的契约定义为：

```
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    MyCollection<Contact> GetContacts( );
}
```

在导入元数据后，客户端的定义则为：

```
[CollectionDataContract]
public class MyCollectionOfContact : List<Contact>
```



```
{ }

[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    MyCollectionOfContact GetContacts( );
}
```

此外，在装载服务时，CollectionDataContract特性会检验Add()方法以及IEnumeralbe或者IEnumerable<T>接口是否存在。如果不存在，就会导致InvalidDataContract-Exception异常。

注意，我们不能同时将DataContract特性与CollectionDataContract特性应用到集合类型上，在装载服务时同样会检查这一点。

## 引用集合

WCF甚至允许开发者在客户端预留与服务端相同的集合。在服务引用的高级设置对话框（参见图1-10），包含了“Collection Type”组合框，允许开发者指定如何展现给客户端以确一切类型的集合和数组，且能够在服务元数据中找到。例如，如果服务操作返回IEnumerable<T>、IList<T>或ICollection<T>集合的其中一种，则代理默认表示为数组（组合框中的默认项）。但是，我们可以要求Visual Studio 2008使用另一种集合，例如针对数据绑定的BindingList、List<T>、Collection或者LinkedList<T>等。如果这种转换是允许的，代理就会使用要求的集合类型，而不是数组。例如：

```
[OperationContract]
List<int> GetNumbers( );
```

当集合在客户端项目引用的另一个程序集中被定义时（我们已经讨论过），集合就会保留原样而被导入。在与其中一种.NET内建集合例如定义在System.dll（该程序集实际上会被所有的.NET项目引用）中的Stack<T>进行交互时，这一功能就显得格外有用。

## 字典集合

字典集合是一个特殊的集合类型，它可以为两个数据契约类型建立映射关系。因此，字典集合不会被编组为数组或列表。不用感到惊讶，字典集合在WCF中有它自己的表现形式。

如果字典集合是一个实现了IDictionary接口的可序列化集合，那么它会被公开为Dictionary<object,object>。例如，这样的服务契约定义：

```
[Serializable]
```

```
public class MyDictionary : IDictionary
{...}

[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary GetContacts( );
}
```

公开的定义为:

```
[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    Dictionary<object,object> GetContacts( );
}
```

顺便说说,这同样适用于HashTable集合类型。

如果可序列化的集合实现了IDictionary<K,T>接口,例如:

```
[Serializable]
public class MyDictionary<K,T> : IDictionary<K,T>
{...}

[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary<int,Contact> GetContacts( );
}
```

Dictionary<K,T>导出的表示形式为:

```
[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    Dictionary<int,Contact> GetContacts( );
}
```

在最初的定义中,直接使用了Dictionary<K,T>类型,而不是MyDictionary<K,T>。

如果字典集合并不支持可序列化,而是标记了CollectionDataContract特性,那么它会被编组为各自表示形式的一个子类。例如,这样的服务契约定义:

```
[CollectionDataContract]
public class MyDictionary : IDictionary
{...}
```

```
[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary GetContacts( );
}
```

则它的表示形式为:

```
[CollectionDataContract]
public class MyDictionary : Dictionary<object,object>
{
}

[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary GetContacts( );
}
```

如果是泛型集合:

```
[CollectionDataContract(Name = "MyDictionary")]
public class MyDictionary<K,T> : IDictionary<K,T>
{...}

[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary<int,Contact> GetContacts( );
}
```

在元数据中就会被发布为:

```
[CollectionDataContract]
public class MyDictionary : Dictionary<int,Contact>
{
}

[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary GetContacts( );
}
```

至于集合, 在服务引用的高级设置对话框 (参见图1-10) 中, 我们可以要求其他的字典类型, 例如SortedDictionary<T,K>、HashTable或ListDictionary类型, 如果可行, 代理反过来会使用字典集合。

## 第 4 章

# 实例管理

实例管理是 WCF 使用的一系列技术的总称，通过它可以将客户端的请求绑定到服务实例上，并根据客户端请求的类型以确定服务实例的管理方式。由于应用程序在可扩展性、性能、吞吐量、事务与队列调用等方面存在巨大的差异，因而需要开发者对实例进行管理。要满足上述的各种要求，并没有一个放之四海而皆准的解决方案。不过，仍然可以将一些规范的实例管理技术应用到不同范围的应用程序上，从而衍生出众多应用场景与编程模型。本章所要探讨的主题正是这些实例管理技术，深入理解它们有助于我们开发出可扩展的、稳定的面向服务应用程序。WCF 支持三种实例激活的类型：单调服务 (Per-Call Service) 会为每次的客户端请求分配（销毁）一个新的服务实例。会话服务 (Sessionful Service) 则为每次客户端连接分配一个服务实例。最后一种是单例服务 (Singleton Service)，所有的客户端会为所有的连接和激活对象共享一个相同的服务实例。本章介绍了每种实例管理模式的基本原理，提供了设计指南用以判断何时以及如何有效地使用它们。同时，本章还将介绍与之相关的主题，例如行为 (Behavior)、上下文 (Context)、分步操作 (Demarcating Operations)、实例停用 (Instance Deactivation) 与限流 (Throttling，注 1)。

## 行为

大体而言，服务实例模式只是服务端的实现细节，在任何情况下，它们都不应该出现在客户端。为了支持服务端的其他本地特性，WCF 定义了行为的概念。一个行为就是服务的本地特性，它不会影响服务的通信模式。客户端并不知道行为，行为也不会出现在服

注 1： 本章引用了作者在 MSDN 杂志 2006 年 6 月刊中发表的文章《WCF Essentials: Discover Mighty Instance Management Techniques for Developing WCF Apps》。

务的绑定或发布的元数据中。WCF 定义了两端类型的服务端行为，分别体现为两个特性：ServiceBehaviorAttribute 特性用于配置服务行为，它能够影响服务的所有终结点（包括所有的契约与操作）。ServiceBehavior 特性可以直接应用到服务的实现类上。在本章的内容中，ServiceBehavior 特性用于配置服务的实例模式。正如例 4-1 所示，该特性定义了 InstanceContextMode 属性，它的类型为枚举类型 InstanceContextMode。InstanceContextMode 枚举的值用于控制应用在服务上的实例模式的类型。

例 4-1：用于配置实例上下文模式的 ServiceBehaviorAttribute 特性

```
public enum InstanceContextMode
{
    PerCall,
    PerSession,
    Single
}
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, ...
{
    public InstanceContextMode InstanceContextMode
    {get;set;}
    // 更多成员
}
```

OperationBehaviorAttribute 特性则用于配置操作的行为，它只能影响一个特定的操作实现。OperationBehavior 特性只能应用到实现了契约操作的方法上，而不能应用到契约自身的操作定义上。在本章后面的内容以及后续章节中，可以看到 OperationBehavior 的用法。

## 单调服务

当服务类型被配置为单调激活方式时，服务实例（CLR 对象）只存在于客户端的调用过程中。每次客户端发出请求（即 WCF 契约的方法调用），就会获得一个新建的专门的服务实例。以下列出了单调激活的执行细节，步骤如图 4-1 所示。

1. 客户端调用代理，代理将调用转发给服务。
2. WCF 创建一个服务实例，然后调用服务实例的方法。
3. 当方法调用返回时，如果对象实现了 IDisposable 接口，WCF 将调用 IDisposable.Dispose() 方法。
4. 客户端调用代理，代理将调用转发给服务。
5. WCF 创建一个对象，然后调用对象的方法。

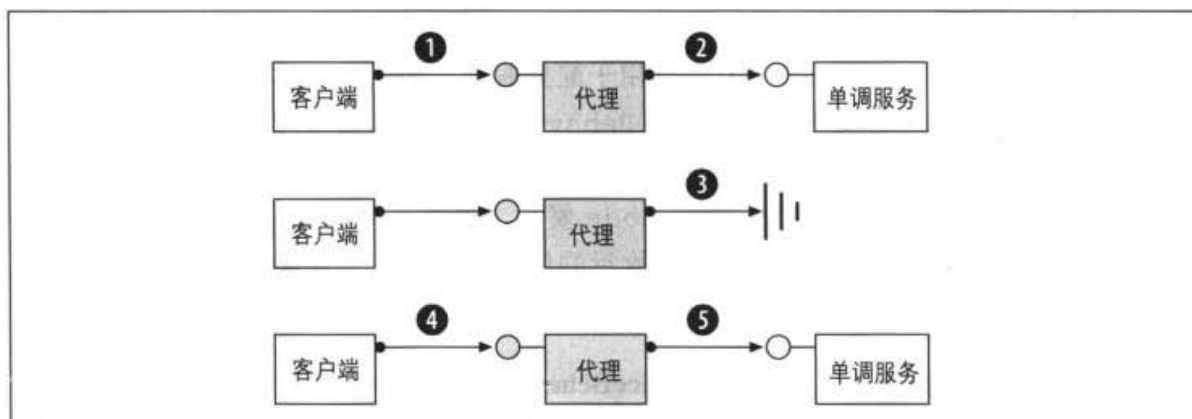


图 4-1：单调实例模式

需要关注的是服务实例的销毁。正如之前提到的，如果服务实现了 `IDisposable` 接口，WCF 会自动调用 `Dispose()` 方法，允许服务执行所有必要的清除工作。注意，调用 `Dispose()` 方法的线程与分发原有方法调用的线程是相同的，同时，`Dispose()` 还包含了一个操作上下文（后面会提及）。一旦调用了 `Dispose()` 方法，WCF 就会断开实例与其他 WCF 基础构架的连接，然后将它放到垃圾回收器中等待回收。

## 单调服务的优势

在经典的客户端/服务器编程模型中，使用诸如 C++ 或者 C# 语言，每个客户端都拥有自己的专门的服务器对象。这种方式的根本问题是它不能很好地应对系统规模的变化。服务器对象可能持有昂贵的或者稀缺的资源，例如数据库连接、通信端口或者文件。假定该应用程序需要为多个客户端提供服务。在通常情况下，这些客户端会在启动时创建它们需要的对象，而在关闭时释放这些对象。客户端/服务器模式之所以不支持系统的可伸缩性，是因为不管使用对象的时间长短，应用程序都会长时间的占用对象。如果我们为每个客户端分配对象，就会长期占用这些紧缺而有限的资源，直到将资源消耗完毕。

更好的激活方式是在客户端调用服务的进程中，只为该客户端分配一个对象。只有在发生并发调用时，我们才需要在内存中创建和维护多个对象，而不应该为多个客户端分配多个对象。在一个典型的企业系统中，只有 1% 的客户端可能执行并发调用（高负荷的企业系统可能存在 3% 的并发调用）。如果系统只能并发地支持 100 个昂贵的服务实例，则意味着它同时需要为 10000 个独立的客户端提供服务。这正是单调实例激活模式所能提供的，因为在调用期间，客户端仅仅持有一个代理的引用，而不是实际的对象。这样做的明显好处是在客户端释放代理对象之前，可以释放被服务实例长期占用的昂贵资源。同时，只有在客户端实际需要的时候，才会获取这些资源。

务必谨记,即使服务端重复地创建和销毁服务实例,也不会断开与客户端(通过客户端的代理)的连接,这比创建实例与连接所消耗的资源要少得多。第二个优势在于,如果每次调用都需要事务资源和事务编程(在第7章讨论),那么单调服务会强行要求服务实例重新分配或连接资源,从而减轻同步实例状态的任务。单调服务的第三个优势是它可以用于队列离线调用(在第9章讨论),因为它能够建立服务实例与离散队列消息之间的简单映射。

## 配置单调服务

要将服务类型配置为单调服务,可以应用ServiceBehavior特性,并将它的InstanceContextMode 属性设置为 InstanceContextMode.PerCall:

```
[ServiceContract]
interface IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{...}
```

例4-2定义了一个简单的单调服务,以及它的客户端。通过程序输出可以看到,单调服务会为每次客户端方法调用创建一个新的服务实例。

### 例4-2: 单调服务与客户端

```
////////// 服务代码 //////////
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract, IDisposable
{
    int m_Counter = 0;

    MyService()
    {
        Trace.WriteLine("MyService.MyService()");
    }
    public void MyMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + m_Counter);
    }
    public void Dispose()
    {
        Trace.WriteLine("MyService.Dispose()");
    }
}
```



```
    }  
}  
////////// 客户端代码 ///////////  
MyContractClient proxy = new MyContractClient();  
  
proxy.MyMethod();  
proxy.MyMethod();  
  
proxy.Close();  
  
// 可能的输出  
MyService.MyService()  
Counter = 1  
MyService.Dispose()  
MyService.MyService()  
Counter = 1  
MyService.Dispose()
```

## 设计单调服务

虽然理论上我们可以将单调实例激活模式应用到任意一个服务类型上,但实际上,我们需要从一开始就要将服务以及服务的契约设计为支持单调激活模式。主要的问题是客户端并不知道是否需要每次获取一个新的实例。单调服务必须是状态相关的,也就是说,它必须积极地管理自己的状态,并在持续的会话过程中给客户端一个提示。一个状态相关的服务与状态无关的服务迥然不同。事实上,如果单调服务真的与状态无关,就根本不需要单调激活模式。准确地讲,正是因为状态,特别是代价昂贵的状态,才需要使用单调模式。单调服务的一个实例创建于每个方法调用之前,调用完成后会立即销毁该服务实例。因此,在每次调用之初,对象需要获取存储在某处的值用来初始化它的状态。通常,使用的存储体为数据库或文件系统,但也可能是一个不稳定的存储体,例如静态变量。

并非所有的对象状态都能够存储。例如,如果状态包含了一个数据库连接,对象就必须在创建时或者在每次调用之初(或者在构造函数中)重新确认连接,并在调用完成后释放连接,或者在IDisposable.Dispose()方法的实现中释放连接。使用单调实例模式,对于操作的设计而言是一个重要的启示,那就是每个操作都必须定义一个参数,用来识别哪些是需要重新获取状态的服务实例。实例通过该参数从存储体中获得它自身的状态,而不是相同类型的另一个实例的状态。例如,银行账户服务中的账号参数,订单处理服务的订单号等。例4-3演示了实现单调服务的一个范例。

### 例4-3: 实现单调服务

```
[DataContract]  
class Param  
{...}
```

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod(Param stateIdentifier);
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyPerCallService : IMyContract, IDisposable
{
    public void MyMethod(Param stateIdentifier)
    {
        GetState(stateIdentifier);
        DoWork();
        SaveState(stateIdentifier);
    }
    void GetState(Param stateIdentifier)
    { ... }
    void DoWork()
    { ... }
    void SaveState(Param stateIdentifier)
    { ... }
    public void Dispose()
    { ... }
}
```

MyPerCallService 类实现了 MyMethod() 操作，操作接收了一个 Param 类型的参数（在本例中，是一个假设的类型），用来识别实例：

```
public void MyMethod(Param stateIdentifier);
```

然后，实例使用 stateIdentifier 获取它的状态，并在方法调用结束时保存状态。每一份状态对于所有客户端都是公共的，都能够在构造函数中被分配，在执行 Dispose() 方法时被释放。

同样，如果每次方法调用要执行的任务是限定的，单调激活模式就是我们的最佳选择。一旦方法返回，后台不会有太多的活动需要完成。为此，我们不应该拆分后台线程，也不应该将异步调用分发给实例，因为一旦方法返回，对象就会被丢弃。由于单调服务会从每次方法调用中的存储体获取状态，只要存储的状态是所有机器都能够访问的全局资源，那么单调服务完全适用于负载均衡（Load-Balancing）的系统环境。负载均衡器能够根据情况将调用重定位到不同的机器，而这些机器在获取它的状态后，就可以知道每个单调服务是否能够执行调用。

## 单调服务与性能

单调服务会为每次方法调用重建实例状态，这必然影响系统的性能，但换来的却是良好的可伸缩性，因为它持有了服务所依赖的状态与资源。选择性能，还是选择可伸缩性，正如“鱼与熊掌”，不可兼得。选择二者的方式与时机，并没有一个硬性的规定。我们

需要分析整个系统，最终做出设计上的权衡，从而判断哪些服务应该使用单调激活，哪些服务则不必要。

## Cleanup 操作

无论服务类型是否支持 `IDisposable`，它都与客户端无关，这仅仅是一个实现细节。实际上，客户端在任何情况下都不会调用 `Dispose()` 方法。当我们将一个契约设计为单调服务时，应该避免定义清除状态与资源的操作（译注1），如下所示：

```
// 避免
[ServiceContract]
interface IMyContract
{
    void DoSomething();
    void Cleanup();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyPerCallService : IMyContract, IDisposable
{
    public void DoSomething()
    { ... }
    public void Cleanup()
    { ... }
    public void Dispose()
    {
        Cleanup();
    }
}
```

显然，这样的设计无异于画蛇添足。假定存在 `Cleanup` 方法，客户端在调用 `Cleanup()` 方法之后，WCF 会紧接着执行 `Dispose()` 方法，这就导致了清除操作的重复执行。

## 选择单调服务

在客户端/服务器开发者的眼中，似乎单调服务的编程模型看起来有几分离经叛道，但对于 WCF 服务而言，单调服务才是最佳的实例管理模式。一个有力的论据是单调服务更利于系统的可伸缩性。为了更好的支持可伸缩性，服务设计有一个黄金法则是 10X，即设计出的每个服务应该能够处理至少多于需求一个量级以上的负载。这是一个工程学准则，工程师在设计系统时，绝不能够“鼠目寸光”，只考虑当前指定负载的处理。如果一幢大楼，只能够支撑当前需求确定的承重，还会有人胆敢居住吗？如果一座电梯，只能够承受规定的六位乘客的重量，还会有人愿意乘坐吗？软件系统同样如此。为什么

---

译注1：一旦调用结束，服务实例会自动被释放，因此在服务契约中没有必要公开 `Cleanup()` 方法，只需作为一个非服务操作，放在 `Dispose()` 方法中被调用。

不能针对当前指定的负载进行系统设计？假设采用这样的设计方式，一旦系统的每位用户增加了业务量，那么系统就会变得岌岌可危。设计良好的系统必须能够经久不衰，经得起时间的考验。为了实现这一目的，就需要应用10X的黄金法则，有效地利用单调服务所能提供的可伸缩性。采用单调服务的另一个有力论据是关于事务的处理。正如第7章介绍的那样，事务绝对是每个系统所必需的，单调服务有利于实现事务编程模型，而不用考虑系统的负载。

## 会话服务

WCF能够维持客户端与特定的服务实例之间的会话。如果服务被配置为会话服务（Per-Session Service，译注2），那么当客户端为该服务创建一个新的代理时，就会获得一个新建的专有的服务实例，它与相同服务的所有其他实例无关。该实例将一直保留在服务中直到客户端不再需要它。这种激活模式非常像经典的客户端-服务器模型。有时候，这种模式也会使用私有会话。每个私有会话单独地将代理与它的客户端集合，以及服务端通道绑定到特定的服务实例上（实际上是绑定到它的上下文，在后面将要讨论）。

因为在整个会话期间，服务实例都保留了内存空间，并用它来维持会话的状态，这就使得这种编程模型更近似于经典的客户端/服务器模式。因此，它与客户端/服务器模式一样，仍然存在可伸缩性以及事务处理的问题。一个配置了私有会话的服务通常无法支持多达几十个（或者上百个）独立的客户端，因为创建专门的服务实例的代价太大。

---

**注意：** 客户端会话是一个代理对应一个服务终结点。如果客户端为相同或不同的终结点创建了另外的代理，则新建的代理就会与新的实例和会话建立关联。

---

## 配置私有会话

若要坚持会话，需要三个元素：行为、绑定和契约。行为元素是必备的，因为它能够确保服务实例不会在会话期间被销毁，同时还能够指示客户端发送消息给它。要实现这样的本地行为，可以将ServiceBehavior特性的InstanceContextMode属性设置为InstanceContextMode.PerSession：

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]  
class MyService : IMyContract  
{...}
```

---

译注2： 实际上，会话服务相当于.NET Remoting中的客户端激活模式。

既然 `InstanceContextMode` 属性的默认值为 `InstanceContextMode.PerSession`, 因此下列定义是等效的:

```
class MyService : IMyContract
{...}

[ServiceBehavior]
class MyService : IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
class MyService : IMyContract
{...}
```

通常, 客户端在关闭代理时会终止会话。这会触发代理通知服务会话已经终止(译注3)。如果服务支持 `IDisposable` 接口, 则 `Dispose()` 方法则以异步方式被客户端调用。如果没有操作上下文 (Operation Context), `Disposed()` 方法则会在工作线程中调用。

为了关联所有从特定的客户端发送到特定实例的消息, WCF 需要具有识别客户端的能力。一种办法是依靠传输层会话 (Transport-Level Session), 即传输层的持久连接, 例如 TCP 和 IPC 协议所维持的连接。因此, 在使用 `NetTcpBinding` 或者 `NetNamedPipeBinding` 时, WCF 会关联客户端的连接。如果是与连接无关的 HTTP 协议, 情况就要复杂得多。从概念上讲, 每个基于 HTTP 的消息都会通过一个新的连接到达服务。因此, 我们无法通过 `BasicHttpBinding` 来维持传输层的会话。而对于 WS 绑定而言, 它能够模拟传输层会话, 包括消息头中的逻辑会话 ID, 通过该 ID 能够唯一地识别客户端。事实上, 只要启用了安全或可靠的消息传输, `WSHttpBinding` 就会模拟一个传输会话。

若要跨越服务边界, 则契约元素就是必须的, 因为客户端的 WCF 运行时需要知道服务是否使用了会话。 `ServiceContract` 特性提供了 `SessionMode` 属性, 它属于 `SessionMode` 枚举类型:

```
public enum SessionMode
{
    Allowed,
    Required,
    NotAllowed
}
[AttributeUsage(AttributeTargets.Interface|AttributeTargets.Class,
    Inherited=false)]
public sealed class ServiceContractAttribute : Attribute
{
    public SessionMode SessionMode
    {get;set;}
}
```

译注3: 会话服务中的会话与代理有关, 而不是真正对应于一个客户端。

```
// 更多成员  
}
```

SessionMode 的默认值为 SessionMode.Allowed。当客户端导入契约元数据时，服务元数据将包含 SessionMode 值，并会如实地反映它的内容。

## SessionMode.Allowed

SessionMode.Allowed 是 SessionMode 属性的默认值，因此下列定义是等效的：

```
[ServiceContract]  
interface IMyContract  
{...}  
  
[ServiceContract(SessionMode = SessionMode.Allowed)]  
interface IMyContract  
{...}
```

所有的绑定都允许将终结点上的契约配置为 SessionMode.Allowed。SessionMode 属性不仅与实例模式有关，而且也代表了传输层会话（或者是在 WS 绑定下模拟的传输层会话）。顾名思义，当 SessionMode 属性被设置为 SessionMode.Allowed 时，只能代表它允许传输会话，但并不代表它会强制要求。生成的服务行为取决于服务配置与使用的绑定。如果服务被配置为单调服务，就会采用单调服务的执行方式，如例 4-2 所示。如果服务被配置为会话服务，那么只有当它使用的绑定维持了一个传输层会话时，才会采用会话服务的执行方式。例如 BasicHttpBinding 绑定，由于 HTTP 协议本质上无连接的，因此它永远不可能拥有传输层会话。没有包含安全与可靠消息传输的 WSHttpBinding 绑定同样不会维持一个传输层会话。在这几种情况下，即使服务被配置为 InstanceContextMode.PerSession，并且契约的 SessionMode 值为 SessionMode.Allowed，服务仍然会采用单调服务的执行方式，并以异步方式调用 Dispose() 方法。也就是说，在 WCF 执行调用释放实例后，客户端不会被阻塞。

然而，如果使用的 WSHttpBinding 绑定包含了安全（为默认配置）或者可靠的消息传输，或者使用 NetTcpBinding 绑定、NetNamedPipeBinding 绑定，则采用会话服务的执行方式。例如，假定使用了 NetTcpBinding 绑定，该服务就是会话服务：

```
[ServiceContract]  
interface IMyContract  
{...}  
  
class MyService : IMyContract  
{...}
```

注意，以上的代码段均采用了 SessionMode 和 InstanceContextMode 属性的默认值。



## SessionMode.Required

SessionMode.Required值要求必须使用传输层会话,但应用层会话却不是必要的。如果服务终结点的绑定没有维持一个传输层会话,就不能为这样的服务契约配置SessionMode.Required。这一约束条件会在装载服务时进行验证。然而,我们仍然可以将服务配置为单调服务,服务实例会在每次客户端调用期间创建与销毁实例。只有当服务被配置为会话服务时,服务实例才会存活于整个客户端会话中:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{...}

class MyService : IMyContract
{...}
```

---

**注意:** 若要设计一个会话契约,推荐使用SessionMode.Required,而非SessionMode.Allowed默认值。在本书的其余代码示例中,只要是会话契约,均应用了SessionMode.Required值。

---

例4-4列出了与例4-2相同的服务与客户端定义,唯一不同的是契约和服务根据需要被配置为私有会话。从输出结果可以看出,客户端获得了一个专门的实例。

### 例4-4: 会话服务与客户端

```
//////////////////// 服务代码 //////////////////////
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
class MyService : IMyContract, IDisposable
{
    int m_Counter = 0;
    MyService()
    {
        Trace.WriteLine("MyService.MyService()");
    }
    public void MyMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + m_Counter);
    }
    public void Dispose()
    {
        Trace.WriteLine("MyService.Dispose()");
    }
}
```



```
////////// 客户端代码 //////////  
MyContractClient proxy = new MyContractClient();  
  
proxy.MyMethod();  
proxy.MyMethod();  
  
proxy.Close();  
  
// 输出  
MyService.MyService()  
Counter = 1  
Counter = 2  
MyService.Dispose()
```

## SessionMode.NotAllowed

SessionMode.NotAllowed禁止使用传输层会话,也不允许使用应用层会话。不管服务配置如何,它总是采用单调服务方式。如果服务实现了IDisposable接口,则客户端将以异步方式调用Dispose()方法,即控制权将返回给客户端,而WCF则在后台的传入调用线程(Incoming Call Thread)中调用Dispose()方法。既然TCP和IPC协议均在传输层维持了一个会话,如果契约使用了NetTcpBinding或NetNamedPipeBinding绑定,此时我们就不能配置一个服务终结点去公开标记为SessionMode.NotAllowed的契约,服务装载时会对此进行校验。但如果使用WSHttpBinding模拟传输会话,则是允许的。考虑到代码的可读性,我的建议是在选择使用SessionMode.NotAllowed的同时,总是将服务配置为单调服务:

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]  
interface IMyContract  
{...}  
  
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]  
class MyService : IMyContract  
{...}
```

既然BasicHttpBinding不能拥有传输层会话,那么使用它的终结点的执行方式就与配置为SessionMode.NotAllowed相似,总是异步调用Dispose()方法。

## 绑定、契约与服务行为

表4-1对实例模式与服务使用的绑定、契约的会话模式以及在服务行为中配置的实例上下文模式之间的关系进行了总结。表中并没有列出无效的配置情况,例如使用BasicHttpBinding时配置SessionMode.Required。

表 4-1: 绑定、契约配置和服务行为与它们对应的实例模式 (译注 4)

绑定	会话模式	上下文模式	异步调用 Dispose()	实例模式
Basic	Allowed/ NotAllowed	PerCall/ PerSession	Yes	PerCall
TCP, IPC	Allowed/Required	PerCall	No	PerCall
TCP, IPC	Allowed/Required	PerSession	Yes	PerSession
WS (no security, no reliability)	NotAllowed/ Allowed	PerCall/ PerSession	Yes	PerCall
WS (with security or reliability)	Allowed/Required	PerSession	Yes	PerSession
WS (with security or reliability)	NotAllowed	PerCall/ PerSession	Yes	PerCall

## 一致性配置

如果服务实现的其中一个契约是会话契约,那么我们强烈推荐将所有契约都配置为会话契约,而且应该避免将单调服务与会话契约混合定义在相同的会话服务类型中,即使 WCF 允许这样的配置:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{...}

[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyOtherContract
{...}
```

译注 4: 几种特殊情况:

1. 只要 InstanceContextMode 被配置为 PerCall, 则不管 SessionMode 的值, 均为单调服务。
2. 如果使用了没有传输会话的绑定, 即使 InstanceContextMode 被配置为 PerSession, 并且 SessionMode 被配置为 Allowed, 仍然为单调服务; 如果此时的 SessionMode 被配置为 Required, 则会抛出异常。例如使用 BasicHttpBinding, InstanceContextMode 为 PerSession, SessionMode 为 Required, 就会出现运行时错误。
3. 不管 InstanceContextMode 的值, 只要 SessionMode 的值为 NotAllowed, 就不可能为会话服务。

```
// 避免
class MyService : IMyContract, IMyOtherContract
{...}
```

原因显而易见：单调服务需要预先管理它们的状态，而会话服务则不需要。如果两个契约暴露了两个不同的终结点，同时又分别被两个不同的客户端调用，就会为服务类的实现带来麻烦。

## 会话与可靠性

客户端与服务实例之间唯一可靠的会话就是基本的传输会话。因此，一个实现了会话契约的服务，它包含的所有终结点所公开的契约都应该使用支持可靠传输会话的绑定。必须确保使用的绑定是可靠的，如此才可以利用编程方式或管理方式，将客户端和服务配置为支持可靠会话，如例 4-5 所示。

### 例 4-5：启用单调会话服务的可靠性

```
<!-- 宿主配置:-->
<system.serviceModel>
  <services>
    <service name = "MyPerSessionService">
      <endpoint
        address = "net.tcp://localhost:8000/MyPerSessionService"
        binding = "netTcpBinding"
        bindingConfiguration = "TCPSession"
        contract = "IMyContract"
      />
    </service>
  </services>
  <bindings>
    <netTcpBinding>
      <binding name = "TCPSession">
        <reliableSession enabled = "true"/>
      </binding>
    </netTcpBinding>
  </bindings>
</system.serviceModel>

<!-- 客户端配置:-->
<system.serviceModel>
  <client>
    <endpoint
      address = "net.tcp://localhost:8000/MyPerSessionService/"
      binding = "netTcpBinding"
      bindingConfiguration = "TCPSession"
      contract = "IMyContract"
    />
  </client>
  <bindings>
    <netTcpBinding>
```

```
<binding name = "TCPSession">
    <reliableSession enabled = "true"/>
</binding>
</netTcpBinding>
</bindings>
</system.serviceModel>
```

这一原则唯一的例外是命名管道的绑定。该绑定不需要可靠的消息传输协议（因为所有的调用都发生在同一台机器上），可靠传输被认为是它与生俱来的本质特征。可靠的传输会话是可选的，消息的有序传递（Ordered Delivery）同样如此。如果禁用了有序传递方式，WCF 仍然会为会话提供消息的有序传递。显然，根据应用程序会话的本质，与会话服务交互的客户端希望所有消息都按照发送的顺序进行传递。幸运的是，当启用可靠传输会话时，默认为启用有序传递，并不需要额外进行设置。

## 会话 ID

每个会话都拥有一个客户端和服务都能获得的唯一 ID。会话 ID 是 GUID 的形式，用于日志记录与分析。服务可以通过操作调用上下文（Operation Call Context）访问会话 ID。每个服务操作都拥有一个操作调用上下文，它是一个属性集，可以用于除了回调、事务管理、安全、宿主访问以及访问定义在执行上下文中的对象之外的多种情形。OperationContext 类提供了对操作上下文以及会话 ID 的访问，服务可以通过 OperationContext 类的 Current 静态方法（译注 5）获取当前方法操作上下文的一个引用：

```
public sealed class OperationContext : ...
{
    public static OperationContext Current
    {get; set;}
    public string SessionId
    {get;}
}
```

为了访问会话 ID，服务需要读取 SessionId 属性的值，它会返回一个字符串类型的 GUID：

```
string sessionId = OperationContext.Current.SessionId;
Trace.WriteLine(sessionId);
// 跟踪结果：
//urn:uuid:c8141f66-51a6-4c66-9e03-927d5ca97153
```

如果会话服务没有传输会话（例如使用 BasicHttpBinding 绑定或者服务被设置为 SessionMode.NotAllowed），那么它在访问 SessionId 属性时，会返回 null。正所谓“皮之不存，毛将焉附”，没有会话，自然就没有 ID 了。

译注 5：原文为 static method，实际上是静态属性。

**注意：**在 `IDisposable.Dispose()` 方法中，服务没有包含操作上下文，因此无法访问会话 ID。

客户端能够通过代理访问会话 ID。正如第 1 章所述，`ClientBase<T>` 是通过 Visual Studio 2005 或者 `SvcUtil` 工具生成的代理的基类。`ClientBase<T>` 提供了 `IClientChannel` 类型的只读属性 `InnerChannel`。`IClientChannel` 接口继承了 `IContextChannel` 接口，它提供了 `SessionId` 属性，以返回一个 `string` 类型的会话 ID：

```
public interface IContextChannel : ...
{
    string SessionId
    {get;}
    // 更多成员
}
public interface IClientChannel : IContextChannel,...
{...}
public abstract class ClientBase<T> : ...
{
    public IClientChannel InnerChannel
    {get;}
    // 更多成员
}
```

在例 4-4 的定义中，通过客户端获取会话 ID 的方法可能是这样：

```
MyContractClient proxy = new MyContractClient();
proxy.MyMethod();

string sessionID = proxy.InnerChannel.SessionId;
Trace.WriteLine(sessionID);
// 跟踪结果:
//urn:uuid:c8141f66-51a6-4c66-9e03-927d5ca97153
```

至于客户端会话 ID 与服务的匹配究竟应该达到哪种程度，以及何时允许客户端访问 `SessionId` 属性，则与使用的绑定和配置有关。传输层的可靠会话负责建立客户端与服务端会话 ID 之间的关联。如果使用 TCP 绑定，同时允许可靠会话，那么在向服务发出第一次方法调用以建立会话之后（或者在显式打开代理之后），客户端只能获取一个有效的会话 ID。如果会话 ID 是在第一次调用之前获得的，`SessionId` 属性将被设置为 `null`。客户端获得的会话 ID 要匹配服务的会话 ID。如果使用了 TCP 绑定，但却禁用了可靠会话，客户端能够在发出第一次调用之前获取会话 ID，但获得的 ID 却不同于服务获得的 ID。无论是哪一种 WS 绑定方式，只要包含了可靠消息传输，那么在执行第一次调用之前（或者在打开代理之后），会话 ID 均为 `null`。但在调用之后，客户端与服务总是拥有相同的会话 ID。如果没有可靠消息传输，那么在访问会话 ID 之前，我们必须首先使用代理（或者打开代理），否则就有引发 `InvalidOperationException` 异常的风险。在打开代理之后，客户端与服务会拥有一个相关联的会话 ID。如果使用命名管

道绑定, 客户端能够在执行第一次调用之前访问SessionId属性, 但是客户端获得的会话ID总是不同于服务的会话ID。因此, 在使用命名管道绑定时, 双方最好同时忽略会话ID。

## 会话终止

通常, 一旦客户端关闭了代理, 会话就会终止。但是, 客户端也可以强行终止会话, 也可能因为通信故障而终止会话。每个会话还包含了一个空闲超时值, 默认为10min。如果客户端在10min内没有任何操作, 那么即使客户端期望继续使用该会话, 会话仍然会自动终止。会话如果是因为空闲超时的原因被终止, 那么当客户端试图使用它的代理时, 会获得一个CommunicationObjectFaultedException异常。在绑定中通过配置不同的值, 可以为客户端和服务配置不同的超时值。支持可靠传输层会话的绑定提供了ReliableSession属性, 类型为ReliableSession或者OptionalReliableSession。ReliableSession类定义了InactivityTimeout属性(译注6), 属于TimeSpan类型, 通过它可以配置一个新的空闲超时值:

```
public class ReliableSession
{
    public TimeSpan InactivityTimeout
    {get;set;}
    // 更多成员
}
public class OptionalReliableSession : ReliableSession
{
    public bool Enabled
    {get;set;}
    // 更多成员
}
public class NetTcpBinding : Binding,...
{
    public OptionalReliableSession ReliableSession
    {get;}
    // 更多成员
}
public abstract class WSHttpBindingBase : ...
{
    public OptionalReliableSession ReliableSession
    {get;}
    // 更多成员
}
public class WSHttpBinding : WSHttpBindingBase,...
{...}
public class WSDualHttpBinding : Binding,...
```

译注6: InactivityTimeout属性的默认值为10min。不能将该值设置为小于或等于0的值, 否则抛出ArgumentOutOfRangeException异常。



```
{
    public ReliableSession ReliableSession
    {get;}
    // 更多成员
}
```

例如, 下面的代码利用编程方式将 TCP 绑定的空闲超时值配置为 25 分钟:

```
NetTcpBinding tcpSessionBinding = new NetTcpBinding();
tcpSessionBinding.ReliableSession.Enabled = true;
tcpSessionBinding.ReliableSession.InactivityTimeout = TimeSpan.FromMinutes(25);
```

这等同于配置 config 文件:

```
<netTcpBinding>
  <binding name = "TCPSession">
    <reliableSession enabled = "true" inactivityTimeout = "00:25:00"/>
  </binding>
</netTcpBinding>
```

如果客户端与服务都配置了超时值, 则以短的超时值为准。

---

**注意:** 还有另外一种高级的服务端配置方式可以配置会话终止。ServiceBehavior特性提供了一种高级方案, 通过 AutomaticSessionShutdown 属性管理会话终止。该属性的目的在于优化明确的回调场景, 但在多数情况下完全可以忽略该属性。AutomaticSessionShutdown 的默认值为 true, 当客户端关闭代理时, 会终止会话。如果设置为 false, 会话会一直继续, 直到服务明确地关闭了它的发送通道。设置为 false 时, 具有双向会话 (Duplex Session, 在第 5 章介绍) 的客户端必须手动地关闭双向客户端通道上的输出会话, 否则, 客户端会一直挂起等待会话终止。

---

## 单例服务

单例服务 (Singleton Service) 是一种极端的共享式服务。当一个服务被配置为单例 (Singleton) 时, 所有客户端都将独自连接相同的单个知名 (Well-Known) 实例, 而不考虑它们连接的是服务的哪一个终结点。单例服务的生存期是无限的, 只有在关闭宿主时, 才会被释放。创建宿主时, 单例服务会被创建, 并且只能被创建一次。

使用单例服务不需要客户端维护单例实例的会话, 也不需要支持传输层会话的绑定。如果客户端调用的契约拥有一个会话, 那么在调用期间, 单例服务将拥有一个与客户端相同的会话 ID (在绑定允许的前提下)。关闭客户端代理只能终止会话, 不能终止单例实例。单例服务的会话永远都不会过期。如果单例服务支持的契约不包含会话, 这些契约将不会是单调方式, 它们会连接相同的实例。本质上讲, 单例实例是共享的, 每个客户端都可以创建自己的代理指向它。



可以通过 `InstanceContextMode.Single` 的 `InstanceContextMode` 属性配置单例服务：

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class MySingleton : ...
{...}
```

例4-6演示的单例服务同时实现了两个契约，其中一个契约需要会话，另一个契约则不需要。从客户端的调用可以看出，在两个终结点上的调用都经由相同的实例进行传递，即使关闭了代理，仍然不会终止单例服务。

#### 例4-6：单例服务与客户端

```
////////// 服务代码 //////////
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyOtherContract
{
    [OperationContract]
    void MyOtherMethod();
}
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
class MySingleton : IMyContract, IMyOtherContract, IDisposable
{
    int m_Counter = 0;

    public MySingleton()
    {
        Trace.WriteLine("MySingleton.MySingleton()");
    }
    public void MyMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + m_Counter);
    }
    public void MyOtherMethod()
    {
        m_Counter++;
        Trace.WriteLine("Counter = " + m_Counter);
    }
    public void Dispose()
    {
        Trace.WriteLine("Singleton.Dispose()");
    }
}
////////// 客户端代码 //////////
MyContractClient proxy1 = new MyContractClient();
```

```
proxy1.MyMethod();
proxy1.Close();

MyOtherContractClient proxy2 = new MyOtherContractClient();
proxy2.MyOtherMethod();
proxy2.Close();

// 输出
MySingleton.MySingleton()
Counter = 1
Counter = 2
```

## 初始化单例服务

有时，我们不能只使用默认的构造函数创建和初始化单例服务。或许初始化状态需要执行一些定制步骤，或者需要某些特定的技能，客户端无法使用这些步骤或技能，但它们也不会影响客户端。要支持这样的实现，WCF 允许开发者直接创建单例实例，通过使用 CLR 的初始化机制对它进行初始化。然后，以单例服务方式打开包含该实例的宿主。ServiceHost 类提供了专门的构造函数，可以接收一个 object 对象：

```
public class ServiceHost : ServiceHostBase, ...
{
    public ServiceHost(object singletonInstance,
        params Uri[] baseAddresses);
    public virtual object SingletonInstance
    {get;}
    // 更多成员
}
```

注意，object 参数对象必须被配置为单例方式。考虑例 4-7 的实现，类 MySingleton 首先初始化，然后以单例方式托管服务。

### 例 4-7：初始化并托管单例服务

```
// 服务代码
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class MySingleton : IMyContract
{
    int m_Counter = 0;

    public int Counter
    {
        get
        {
            return m_Counter;
        }
    }
}
```

```

    }
    set
    {
        m_Counter = value;
    }
}
public void MyMethod()
{
    m_Counter++;
    Trace.WriteLine("Counter = " + Counter);
}
}
// 宿主代码
MySingleton singleton = new MySingleton();
singleton.Counter = 42;

ServiceHost host = new ServiceHost(singleton);
host.Open();
// 然后执行某些阻塞调用
host.Close();

// 客户端代码
MyContractClient proxy = new MyContractClient();
proxy.MyMethod();
proxy.Close();

// 输出:
Counter = 43

```

如果采用这种方式初始化并托管单例服务,我们可能还需要具有在宿主端直接访问它的能力。通过使用 `ServiceHost` 的 `SingletonInstance` 属性, WCF 允许下游对象(译注 7)直接返回单例对象。调用链维持了一个单例服务的操作调用,它的任何一方都可以通过操作上下文的 `Host` 只读属性访问宿主:

```

public sealed class OperationContext : ...
{
    public ServiceHostBase Host
    {get;}
    // 更多成员
}

```

一旦拥有单例服务的引用,就能够直接与它交互:

```

ServiceHost host = OperationContext.Current.Host as ServiceHost;
Debug.Assert(host != null);
MySingleton singleton = host.SingletonInstance as MySingleton;
Debug.Assert(singleton != null);
singleton.Counter = 388;

```

如果没有单例实例提供给宿主, `SingletonInstance` 则返回 `null`。

译注 7: 指获得托管的单例服务实例的对象。

## 简化 ServiceHost<T>

我们可以扩展第1章介绍的 ServiceHost<T>类, 提供类型安全的单例对象的初始化与访问方式:

```
public class ServiceHost<T> : ServiceHost
{
    public ServiceHost(T singleton,params Uri[] baseAddresses)
        : base(singleton,baseAddresses)
    {}
    public virtual T Singleton
    {
        get
        {
            if(SingletonInstance == null)
            {
                return default(T);
            }
            return (T)SingletonInstance;
        }
    }
    // 更多成员
}
```

类型参数为对象提供了类型安全的绑定, 用于对象的创建:

```
MySingleton singleton = new MySingleton();
singleton.Counter = 42;

ServiceHost<MySingleton> host = new ServiceHost<MySingleton>(singleton);
host.Open();
```

对象通过 Singleton 属性返回。

```
ServiceHost<MySingleton> host = OperationContext.Current.Host
    as ServiceHost<MySingleton>;

Debug.Assert(host != null);
host.Singleton.Counter = 388;
```

---

**注意:** 相似地, 我们还可以扩展第1章介绍的 InProcFactory<T>类, 使它能够初始化单例实例。

---

## 选择单例服务

单例服务与可伸缩性之间的关系可谓“水火不容”。原因就在于单例服务的状态是同步的。对象是单例的就意味着它包含了一些有价值的状态, 开发者希望这些状态能够被多个客户端共享。问题是当多个客户端连接单例服务时, 它们可能需要并发处理, 而且传入的客户端调用也可能会基于多个工作线程。单例服务必须同步访问它的状态以避免状

态的冲突。这意味着在同一时间只能有一个客户端能够访问单例服务。这可能会降低系统的吞吐量、响应速度以及系统可用性,因而单例方式不能用于规模较大的系统。例如,如果单例服务的操作耗时十分之一秒,那么在一秒钟之内单例服务就只为10个客户端提供服务。如果客户端的数量更多(例如20或100),系统的性能就很难达到要求。

总的来讲,只要应用程序域符合单例模式场景,就可以使用单例对象。一个纯粹的单例对象是一种性质单一且唯一的资源。典型的例子是全局日志,所有服务都将它们的活动记录到日志中。一个单独的通信端口或者单独的机械电机,也可以看作是纯粹的单例对象。如果业务逻辑在将来发生变更,需要增加多个服务,例如添加新的电机或者第二个通信端口,即使这种变更的可能性非常低,我们也应该避免使用单例服务。原因不言而喻:如果依赖的客户端被隐式地连接到知名实例,同时有超过一个以上的服务是可用的,客户端就需要找到一种方法来绑定正确的实例。这可能会严重地影响应用程序的编程模型。既然存在这样的限制,因此我的建议在通常情况下应该尽量避免使用单例服务,通过寻求另外的方法来取代单例实例实现对象状态的共享。不过,对于前面提到的场景,使用单例服务仍然是可以接受的。

## 分步操作

一个会话契约的操作有时会隐含了操作调用的顺序。有的操作不能被最先调用,而有的操作则必须被最后调用。例如,考虑这样一个用于管理顾客订单的契约:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IOrderManager
{
    [OperationContract]
    void SetCustomerId(int customerId);

    [OperationContract]
    void AddItem(int itemId);

    [OperationContract]
    decimal GetTotal();

    [OperationContract]
    bool ProcessOrders();
}
```

IOrderManager 契约存在如下约束:客户端必须为会话过程中的第一个操作提供顾客ID,否则无法调用其他操作;商品的添加与金额的合计与调用顺序无关,这与客户端所希望的一致;至于订单的处理,由于它会终止会话,因此必须在最后调用。

WCF 允许契约设计者指定契约操作为启动或终止会话的操作,方法是使用 OperationContract 特性的 IsInitiating 和 IsTerminating 属性:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public bool IsInitiating
    {get;set;}
    public bool IsTerminating
    {get;set;}
    // 更多成员
}
```

使用这些属性可以划分会话的执行步骤，因此我们将这种技术称之为分步操作 (Demarcating Operation, 译注8)。在装载服务时 (或者在客户端使用代理时)，如果修改了这些属性的默认值，WCF就会验证分步操作是否属于会话契约 (SessionMode被设置为 SessionMode.Required) 的一部分；如果不是，就会抛出 Invalid-OperationException异常。如果一个契约使用分步操作管理它们的客户端会话，则会话服务与单例服务都能够实现这样的契约。

IsInitiating 属性的默认值为 true，IsTerminating 属性的默认值则为 false，因此如下的两个定义是等效的：

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract(IsInitiating = true, IsTerminating = false)]
    void MyMethod();
}
```

如例中所示，我们可以将两个属性同时设置到相同的方法上。此外，在默认情况下，操作不能划分会话的执行顺序。调用操作的顺序可以是最先、最后，也可以在会话中的其他操作之间被调用。使用属性的非默认值可以指定一个方法不被最先调用，或者被最后调用，或者同时符合两个约束条件：

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    void StartSession();
}
```

译注8：Demarcating Operation 翻译为分界操作更符合原文的含义，然而，事实上该操作并非划分会话的边界，而是将整个会话划分为多个执行步骤，因而译为“分步操作”。

```
[OperationContract(IsInitiating = false)]
void CannotStart();

[OperationContract(IsTerminating = true)]
void EndSession();

[OperationContract(IsInitiating = false, IsTerminating = true)]
void CannotStartCanEndSession();
}
```

回到订单管理契约的例子，我们就可以使用分步操作强制约束操作的交互：

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IOrderManager
{
    [OperationContract]
    void SetCustomerId(int customerId);

    [OperationContract(IsInitiating = false)]
    void AddItem(int itemId);

    [OperationContract(IsInitiating = false)]
    decimal GetTotal();

    [OperationContract(IsInitiating = false, IsTerminating = true)]
    bool ProcessOrders();
}

// 客户端代码
OrderManagerClient proxy = new OrderManagerClient();

proxy.SetCustomerId(123);
proxy.AddItem(4);
proxy.AddItem(5);
proxy.AddItem(6);
proxy.ProcessOrders();

proxy.Close();
```

当 `IsInitiating` 的值被设置为 `true`（属性的默认值）时，如果它是客户端最先调用的方法，则意味着操作将启动一个新的会话；但是，如果最先调用的是其他操作，那么它就会成为持续会话（On-Going Session）中的一部分。当 `IsInitiating` 值被设置为 `false` 时，那么在新的会话中，该操作绝对不能作为第一个操作被客户端调用，同时，该方法只能成为持续会话中的一部分。

当 `IsTerminating` 值被设置为 `false` 时（属性的默认值），意味着会话将持续至该操作返回之后。当它被设置为 `true` 时，则意味着一旦方法返回，会话就会被终止，然后 WCF 以异步方式释放服务实例。此时，客户端不能通过代理发出另外的调用。注意，客户端仍然应该关闭代理。



**注意：**如果服务使用了分步操作，那么在为它生成代理时，导入的契约定义会包含这些属性的设置。此外，WCF强制要求客户端与服务端必须分别实现分步操作。只有这样，我们才能单独地使用它们（译注9）。

## 实例停用（译注10）

迄今为止描述的会话服务的实例管理技术，能够将一个客户端（或多个客户端）连接到一个服务实例上。然而，真正的情形却要复杂得多。回想在第1章中，曾经提到每个服务实例都被托管在上下文中，如图4-2所示。

会话实际要做的不仅是关联客户端消息，同时还要关联托管了服务的上下文。启动会话时，宿主会创建一个新的上下文。会话终止时，上下文也随之而终止。在默认情况下，上下文的生命周期与发布的服务实例的生命周期相同。然而，出于优化的目的，WCF为服务设计者提供了一个分离两种生命周期的选项，该选项允许WCF独立地停用实例，而不必依赖于它的上下文。实际上，WCF还允许不包含实例的上下文存在，如图4-2所示。我将这种实例管理技术称为上下文停用（Context Deactivation）。控制上下文停用的最常见办法是通过 `OperationBehavior` 特性的 `ReleaseInstanceMode` 属性：

**译注9：**如果 `IsInitiating` 值为 `true`，并不代表该操作必然是启动会话的第一个操作。如果其他相同设置的操作首先被调用，就会启动一个会话，而原操作则在调用时被加入会话，成为会话的一部分。但如果 `IsTermination` 的值为 `true`，则代表该操作必须是终止会话的操作。虽然在服务契约定义时，允许将多个操作的 `IsTerminating` 值设置为 `true`，但一旦调用了 `IsTerminating` 值为 `true` 的方法，就不能再调用服务实例的其他方法，除非在客户端重新创建一个代理对象。此外，即使操作的 `IsTermination` 值为 `true`，它也可以是启动会话的第一个操作，但在操作执行后它会终止会话。因此，如下的两个操作定义是等效的：

```
[OperationContract(IsTerminating = true)]
void StartAndEndSession();

[OperationContract(IsInitiating=true, IsTerminating = true)]
void StartAndEndSession();
```

然而，如下的两个操作则是不等效的，因为后者要求该操作不能为启动会话的第一个操作：

```
[OperationContract(IsTerminating = true)]
void StartAndEndSession();

[OperationContract(IsInitiating=false, IsTerminating = true)]
void StartAndEndSession();
```

**译注10：**只针对会话服务而言。单例服务虽然也可以应用，但却无效。

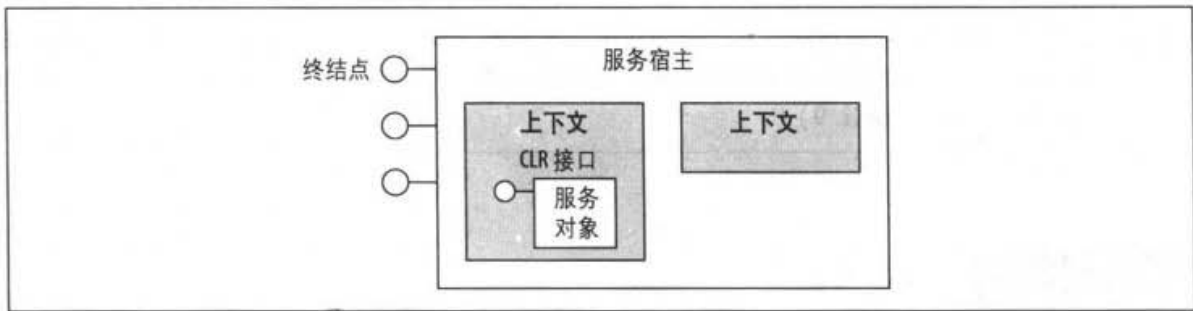


图 4-2: 上下文与实例

```

public enum ReleaseInstanceMode
{
    None,
    BeforeCall,
    AfterCall,
    BeforeAndAfterCall,
}
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute, ...
{
    public ReleaseInstanceMode ReleaseInstanceMode
    {get;set;}
    // 更多成员
}

```

`ReleaseInstanceMode` 属性属于 `ReleaseInstanceMode` 枚举类型。`ReleaseInstanceMode` 控制的不同值代表了释放与方法调用相关的实例的时间：调用前、调用后、调用前与调用后，或者调用中。释放实例时，如果服务支持 `IDisposable` 接口，则还要调用 `Dispose()` 方法，并且 `Dispose()` 方法会拥有一个操作上下文。

通常，我们只需要将实例停用应用到部分服务方法上，而不是全部方法；或者为不同的方法设置不同的值：

```

[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();

    [OperationContract]
    void MyOtherMethod();
}
class MyService : IMyContract, IDisposable
{
    [OperationBehavior(ReleaseInstanceMode = ReleaseInstanceMode.AfterCall)]
    public void MyMethod()
    { ... }
    public void MyOtherMethod()
    { ... }
}

```

```
    public void Dispose()  
    {...}  
}
```

之所以只对服务的部分操作应用实例停用,原因在于如果我们统一地应用它,服务实例的释放方式就与单调服务类似,此时还不如直接将服务配置为单调服务。如果依赖于实例停用的方法需要按照确定的调用顺序,我们可以考虑使用分步操作要求它必须遵循规定的顺序。

## 配置为 ReleaseInstanceMode.None

ReleaseInstanceMode属性的默认值为ReleaseInstanceMode.None,因此如下的两个定义是等效的:

```
[OperationBehavior(RelaseInstanceMode = ReleaseInstanceMode.None)]  
public void MyMethod()  
{...}  
  
public void MyMethod()  
{...}
```

ReleaseInstanceMode.None意味着实例的生命周期不受调用的影响,如图4-3所示。

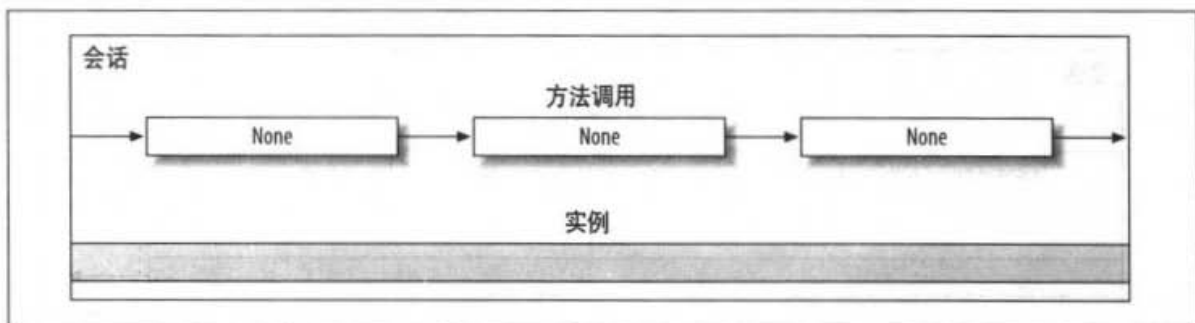


图 4-3: 方法被配置为 ReleaseInstanceMode.None 的实例的生命周期

## 配置为 ReleaseInstanceMode.BeforeCall

当方法被配置为ReleaseInstanceMode.BeforeCall时,如果会话已经存在一个实例,那么在转发调用之前 WCF 会停止实例,创建一个新的实例取代它,并让这个新的实例维持所要转发的调用,如图4-4所示。

在传入调用线程执行调用之前,当客户端被阻塞时,WCF会停止实例并调用Dispose()方法。这样就能确保实例真正可以在调用之前被停止,而不必进行并发处理。ReleaseInstanceMode.BeforeCall的设计可以优化诸如Open()这样的方法,这些

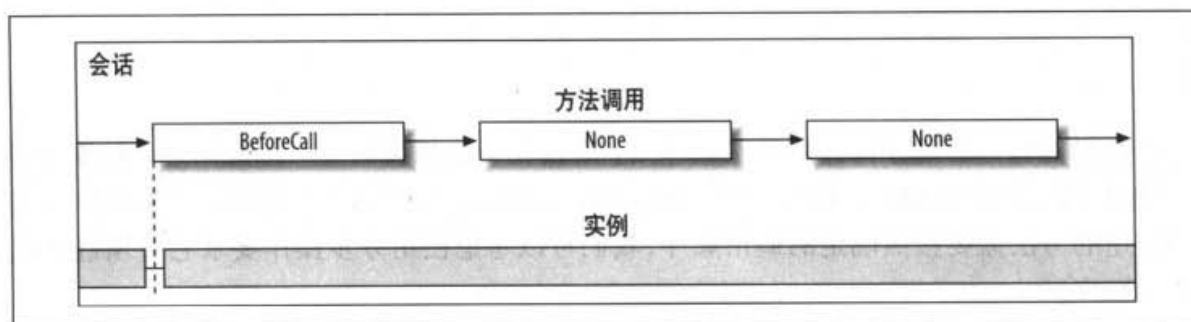


图 4-4：方法被配置为 `ReleaseInstanceMode.BeforeCall` 的实例的生命周期

方法能够获取一些有价值的资源，并有望释放原先分配的资源。启动会话时不需要获取资源，而应该等待，直到调用 `Open()` 方法，然后释放原来分配的资源，同时分配新的资源。在调用 `Open()` 方法之后，我们应准备启动对实例上的其他被配置为 `ReleaseInstanceMode.None` 的方法的调用。

## 配置为 `ReleaseInstanceMode.AfterCall`

当方法被配置为 `ReleaseInstanceMode.AfterCall` 时，WCF 会在调用方法之后停止实例，如图 4-5 所示。

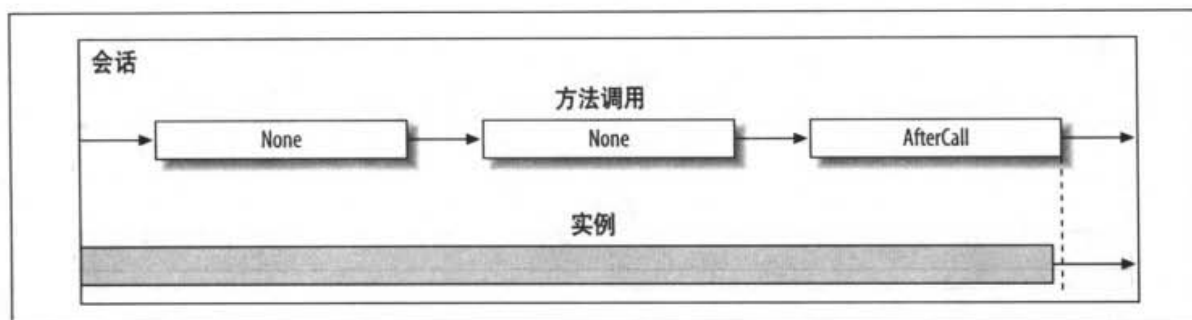


图 4-5：方法被配置为 `ReleaseInstanceMode.AfterCall` 的实例的生命周期

这样的设计可以优化诸如 `Close()` 这样的方法，该方法能够清除实例所持有的有价值的资源，而不需要等到会话结束。应用 `ReleaseInstanceMode.AfterCall` 的方法通常都会在配置为 `ReleaseInstanceMode.None` 的方法之后被调用。

## 配置为 `ReleaseInstanceMode.BeforeAndAfterCall`

当一个方法被配置为 `ReleaseInstanceMode.BeforeAndAfterCall` 时，顾名思义，它结合了 `ReleaseInstanceMode.BeforeCall` 和 `ReleaseInstanceMode.AfterCall`

的功能。在执行调用之前，如果上下文包含了一个实例，那么 WCF 就会在调用前停止实例，并创建一个新的实例去维持调用，然后在调用后停止新建的实例，如图 4-6 所示。

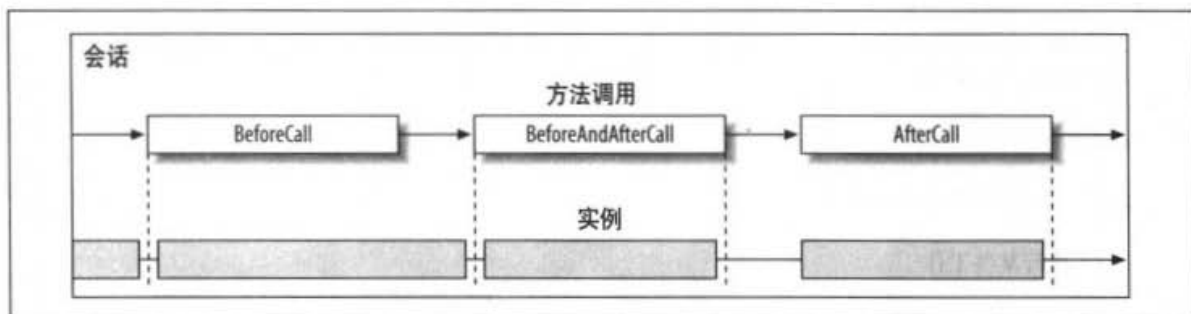


图 4-6: 方法被配置为 `ReleaseInstanceMode.BeforeAndAfterCall` 的实例的生命周期

`ReleaseInstanceMode.BeforeAndAfterCall` 给我们的第一印象似乎有些画蛇添足，但实际上它却弥补了其他设置的不足。它所应用的方法可以是在标记为 `ReleaseInstanceMode.BeforeCall` 或 `ReleaseInstanceMode.Null` 的方法之后调用，也可以是在标记为 `ReleaseInstanceMode.AfterCall` 或 `ReleaseInstanceMode.Null` 的方法之前调用。考虑这样一种情形，会话服务需要获益于状态相关的行为（类似于单调服务），在持有资源时，需要优化资源的分配与安全性查询。如果只能选择使用 `ReleaseInstanceMode.BeforeCall`，那么在调用之后的一段时间内，资源可能会被分配给对象，但对象却没有使用它。相似的，如果只能选择使用 `ReleaseInstanceMode.AfterCall`，在方法调用之前的一段时间内可能存在资源的浪费。

## 显式停止

在设计时如果无法判断方法是否需要停止实例，我们可以在方法返回之后，在运行中停止实例。我们可以通过调用实例上下文的 `ReleaseServiceInstance()` 方法来实现，至于实例上下文，则可以通过操作上下文的 `InstanceContext` 属性获得：

```
public sealed class InstanceContext : CommunicationObject, ...
{
    public void ReleaseServiceInstance();
    // 更多成员
}
public sealed class OperationContext : ...
{
    public InstanceContext InstanceContext
    {get;}
    // 更多成员
}
```

例 4-8 演示了这样的一种技术。

**例 4-8: 使用 ReleaseServiceInstance() 方法**

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
class MyService : IMyContract, IDisposable
{
    public void MyMethod()
    {
        // 执行某些工作
        OperationContext.Current.InstanceContext.ReleaseServiceInstance();
    }
    public void Dispose()
    { ... }
}
```

调用 `ReleaseServiceInstance()` 方法与使用 `ReleaseInstanceMode.AfterCall` 的效果相似。如果是在标记为 `ReleaseInstanceMode.BeforeCall` 的方法中调用 `ReleaseServiceInstance()` 方法, 则类似于使用 `ReleaseInstanceMode.BeforeAndAfterCall`。

---

**注意:** 实例停止同样能够影响单例服务, 虽然将两者结合起来并没有什么意义。服务允许这样的定义, 但试图停止单例对象的方式却是不足取的。

---

## 使用实例停止

实例停止是一种优化技术, 与所有的优化技术相似, 在通常情况下, 我们应避免使用它。只有在无法同时满足系统的性能要求与可伸缩性要求时, 才应该考虑使用实例停止。然而我们需要经过审慎的检查, 在证明信息没有疑义时, 使用实例停止技术才能够改善现状。如果我们需要关注可伸缩性与吞吐量, 就应该避免使用实例停止技术, 而是采取简化单调实例模式的方法。

## 限流

限流 (Throttling) 并非直接的实例管理技术, 它允许开发者限制客户端连接数以及服务的负荷。限流可以避免服务的最大化, 以及分配与使用重要资源的最大化。引入限流技术后, 一旦超出配置的设置值, WCF 就会自动地将等待处理的调用者放入到队列中, 然后依次从队列中取出。在队列中等待处理调用时, 如果客户端的调用超时, 客户端就会获得一个 `TimeoutException` 异常。每个服务类型都可以应用限流技术, 也就是说, 它

会影响到服务的所有实例以及服务类型的所有终结点。实现方式是为限流与服务使用的每个通道分发器建立关联。

WCF 允许开发者控制下列的某些或所有服务消费参数：

- 并发会话最大数：指包含了在服务传输层的一个会话的所有独立的客户端数。简明扼要地说，该数值就等于使用 TCP、IPC 或者任何一种包含会话的 WS 绑定的独立客户端的最大数。在使用基本绑定或者任何一种不包含传输会话的 WS 绑定时，这一数字是无效的，因为基本的 HTTP 连接本质上是无连接的。默认值为 10。
- 并发调用最大数：指所有服务实例当前正在执行的调用总数。通常，该数值应该保持为并发会话最大值的 1% 到 3%。默认值为 16（译注 11）。
- 并发实例最大数。它实际上代表了存活的并发上下文总数。默认值是无限的。实例上下文管理模式以及上下文与实例停止技术决定了实例与上下文映射的方式。对于会话服务而言，实例的最大值等于并发活动实例与并发会话的总和。引入实例停止技术后，实例的数量远远少于上下文的数量。如果上下文的数量达到并发实例最大值，客户端就会被阻塞。对于单调服务而言，实例数实际上与并发调用的数量相同。因此，单调服务实例的最大值要小于并发实例最大值与并发调用最大值的和。如果是单例服务，由于在任和时候都只有一个实例，因而可以忽略并发实例最大值。

---

**警告：**限流体现了服务托管与部署的特征。设计一个服务时，不需要配置限流，因为我们总是认为服务完全能够承受客户端的负荷。这就是为什么限流行为特性虽然易于实现，WCF 却没有提供的主要原因。

---

## 配置限流

通常，限流是管理员在配置文件中配置的。它允许我们能够在不同时间或者横跨不同的部署站点限制相同的服务代码。宿主同样能够基于某种运行时决策，以编程方式配置限流。

### 管理配置限流

例 4-9 演示了如何在宿主配置文件中配置限流。使用 `behaviorConfiguration` 标签，我们可以为服务添加设置限流值的定制行为。

---

译注 11：在 WCF 的早期版本中，并发调用最大数的默认值为 64，但在 2006 年 6 月的 CIP 版本中则被修改为 16。



#### 例 4-9: 管理配置限流

```
<system.serviceModel>
  <services>
    <service name = "MyService" behaviorConfiguration = "ThrottledBehavior">
      ...
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name = "ThrottledBehavior">
        <serviceThrottling
          maxConcurrentCalls      = "12"
          maxConcurrentSessions   = "34"
          maxConcurrentInstances = "56"
        />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

#### 编程配置限流

宿主进程基于某些运行时参数,能够以编程方式限制服务。我们只能在宿主打开之前执行。宿主可以通过移除限流行为,然后再添加限流行为的方式,重写在配置文件中找到的限流行为。如果配置文件没有包含限流行为,则通常需要编程提供限流行为。

ServiceHostBase 类定义了 Description 属性,类型为 ServiceDescription:

```
public abstract class ServiceHostBase : ...
{
    public ServiceDescription Description
    {get;}
    // 更多成员
}
```

顾名思义,服务描述就是对服务所有特征与行为的描述。ServiceDescription 包含了一个属性 Behaviors, 它的类型为 KeyedByTypeCollection<I>, 其中,泛型参数为 IServiceBehavior 类型。

例 4-10 演示了如何通过编程方式设置限流行为。

#### 例 4-10: 编程配置限流

```
ServiceHost host = new ServiceHost(typeof(MyService));

ServiceThrottlingBehavior throttle;
throttle = host.Description.Behaviors.Find<ServiceThrottlingBehavior>();
if(throttle == null)
{
    throttle = new ServiceThrottlingBehavior();
}
```

```
throttle.MaxConcurrentCalls    = 12;
throttle.MaxConcurrentSessions = 34;
throttle.MaxConcurrentInstances = 56;
host.Description.Behaviors.Add(throttle);
}

host.Open();
```

首先, 宿主代码调用 `KeyedByTypeCollection<I>` 的 `Find<T>()` 方法, 验证了配置文件是否提供了服务限流行为。`KeyedByTypeCollection<I>` 使用 `ServiceThrottlingBehavior` 作为类型参数。

`ServiceThrottlingBehavior` 类定义在 `System.ServiceModel.Design` 命名空间中:

```
public class ServiceThrottlingBehavior : IServiceBehavior
{
    public int MaxConcurrentCalls
    {get;set;}
    public int MaxConcurrentSessions
    {get;set;}
    public int MaxConcurrentInstances
    {get;set;}
    // 更多成员
}
```

如果返回的 `throttle` 对象为 `null`, 宿主代码会创建一个新的 `ServiceThrottlingBehavior` 对象, 并设置它的值, 然后将它添加到服务描述的行为中。

## 简化 `ServiceHost<T>`

我们可以扩展 `ServiceHost<T>` 的功能, 自动实现例 4-10 中的代码, 如例 4-11 所示。

### 例 4-11: 扩展 `ServiceHost<T>` 以处理限流

```
public class ServiceHost<T> : ServiceHost
{
    public void SetThrottle(int maxCalls, int maxSessions, int maxInstances)
    {
        ServiceThrottlingBehavior throttle = new ServiceThrottlingBehavior();
        throttle.MaxConcurrentCalls = maxCalls;
        throttle.MaxConcurrentSessions = maxSessions;
        throttle.MaxConcurrentInstances = maxInstances;
        SetThrottle(throttle);
    }
    public void SetThrottle(ServiceThrottlingBehavior serviceThrottle)
    {
        SetThrottle(serviceThrottle, false);
    }
    public void SetThrottle(ServiceThrottlingBehavior serviceThrottle,
                            bool overrideConfig)
    {
```

```

if(State == CommunicationState.Opened)
{
    throw new InvalidOperationException("Host is already opened");
}
ServiceThrottlingBehavior throttle =
    Description.Behaviors.Find<ServiceThrottlingBehavior>();
if(throttle == null)
{
    Description.Behaviors.Add(serviceThrottle);
    return;
}
if(overrideConfig == false)
{
    return;
}
Description.Behaviors.Remove(throttle);
Description.Behaviors.Add(serviceThrottle);
}
public ServiceThrottlingBehavior ThrottleBehavior
{
    get
    {
        return Description.Behaviors.Find<ServiceThrottlingBehavior>();
    }
}
// 更多成员
}

```

`ServiceHost<T>` 定义了 `SetThrottle()` 方法, 能够接收要使用的限流对象, 以及 `Boolean` 类型的标志, 用来指示在已有配置值的情况下是否需要重写配置值。默认值(使用 `SetThrottle()` 方法的其中一个重载版本)为 `false`。`SetThrottle()` 方法使用 `CommunicationObject` 基类的 `State` 属性检验宿主是否已经打开。如果需要重写限流的配置值, `SetThrottle()` 方法会将它从服务描述中移除。例 4-11 中的其余代码与例 4-10 相似。以下代码演示了如何使用 `ServiceHost<T>` 类编程设置限流:

```

ServiceHost<MyService> host = new ServiceHost<MyService>();
host.SetThrottle(12, 34, 56);
host.Open();

```

---

**注意:** 相似的, 我们也可以扩展第 1 章介绍的 `InProcFactory<T>` 类, 简化对限流的处理。

---

## 读取限流值

出于诊断与分析的目的, 服务开发者可以在运行时读取限流值。服务实例在运行时能够从它的分发器中访问服务的限流属性。首先, 从操作上下文中获取宿主的一个引用。宿主的基类 `ServiceHostBase` 定义了只读属性 `ChannelDispatchers`:

```
public abstract class ServiceHostBase : CommunicationObject, ...
{
    public ChannelDispatcherCollection ChannelDispatchers
    {get;}
    // 更多成员
}
```

ChannelDispatchers 是 ChannelDispatcherBase 对象的强类型集合：

```
public class ChannelDispatcherCollection :
    SynchronizedCollection<ChannelDispatcherBase>
{...}
```

集合中的每个元素均为 ChannelDispatcher 类型。ChannelDispatcher 定义了 ServiceThrottle 属性：

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    public ServiceThrottle ServiceThrottle
    {get;set;}
    // 更多成员
}
public sealed class ServiceThrottle
{
    public int MaxConcurrentCalls
    {get;set;}
    public int MaxConcurrentSessions
    {get;set;}
    public int MaxConcurrentInstances
    {get;set;}
}
```

ServiceThrottle 包含了配置的限流值：

```
class MyService : ...
{
    public void MyMethod() // 契约操作
    {
        ChannelDispatcher dispatcher = OperationContext.Current.
            Host.ChannelDispatchers[0] as ChannelDispatcher;

        ServiceThrottle serviceThrottle = dispatcher.ServiceThrottle;

        Trace.WriteLine("Max Calls = " + serviceThrottle.MaxConcurrentCalls);
        Trace.WriteLine("Max Sessions = " + serviceThrottle.MaxConcurrentSessions);
        Trace.WriteLine("Max Instances = " + serviceThrottle.MaxConcurrentInstances);
    }
}
```

注意，服务只能够读取限流值，而不能改变它们。如果服务试图设置限流值，就会抛出一个 InvalidOperationException 异常。

此外，我们可以通过 `ServiceHost<T>` 简化对限流的查找。首先，添加一个 `ServiceThrottle` 属性：

```
public class ServiceHost<T> : ServiceHost
{
    public ServiceThrottle Throttle
    {
        get
        {
            if(State == CommunicationState.Created)
            {
                throw new InvalidOperationException("Host is not opened");
            }
            ChannelDispatcher dispatcher = OperationContext.Current.
                Host.ChannelDispatchers[0] as ChannelDispatcher;
            return dispatcher.ServiceThrottle;
        }
    }
    // 更多成员
}
```

然后，使用 `ServiceHost<T>` 托管服务，使用 `ServiceThrottle` 属性访问配置的限流值：

```
// 宿主代码
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.Open();

class MyService : ...
{
    public void MyMethod()// 契约操作
    {
        ServiceHost<MyService> host = OperationContext.Current.
            Host as ServiceHost<MyService>;

        ServiceThrottle serviceThrottle = host.Throttle;
        ...
    }
}
```

---

**注意：**我们只能在打开宿主之后访问 `ServiceHost<T>` 的 `Throttle` 属性。因为分发器集合只能在打开宿主之后初始化。

---

## 绑定中的限流连接

在使用TCP和命名管道绑定时，我们也可以在绑定中为一个特定的终结点配置最大连接数。`NetTcpBinding` 与 `NetNamedPipeBinding` 都提供了 `MaxConnections` 属性：

```
public class NetTcpBinding : Binding,...
{
    public int MaxConnections
    {get;set;}
}
public class NetNamedPipeBinding : Binding,...
{
    public int MaxConnections
    {get;set;}
}
```

在宿主端，我们既可以通过编程方式，也可以使用配置文件设置属性值：

```
<bindings>
  <netTcpBinding>
    <binding name = "TCPThrottle" maxConnections = "25"/>
  </netTcpBinding>
</bindings>
```

MaxConnections的默认值为10。如果绑定层的限流与服务行为的限流都设置了最大连接值，WCF 则选择两者中较小的一个。



## 第5章

# 操作

经典的面向对象或面向组件编程模型只为客户端提供了一种调用方法的方式：客户端发出调用，然后在调用过程中阻塞客户端，直到方法返回客户端再继续执行。其他的调用模式则必须进行手动调整，从而影响了系统的开发效率与质量。WCF支持这种经典的调用模式，同时还提供了对其他操作类型的内建支持，包括：对即发即弃(Fire-and-Forget)操作的单向调用；允许服务将调用返回给客户端的双向回调；允许客户端或服务处理大量负荷的流操作。总体而言，它所使用的操作类型属于服务契约的一部分，同时也是服务设计的内在组成部分。操作类型甚至包含了对绑定的某些约束。因此，在设计客户端与服务时，必须从一开始就要考虑操作类型，因为我们很难在设计之后改变操作的类型。本章专门讲解了调用WCF操作的各种方式，以及相关的设计指南。至于异步处理与队列调用这两种调用操作的方式，则在后面的章节中介绍（注1）。

## 请求-应答操作

在前面章节介绍的所有例子中，契约中的操作均为请求-应答(Request-Reply)类型。客户端以消息形式发出请求，它会阻塞客户端直到收到应答消息。应答的默认超时值为1min，如果超过这一时间服务仍然没有应答，客户端就会获得一个TimeoutException异常。请求-应答是默认的操作模式。针对请求-应答操作的编程非常简单，它与经典的客户端/服务器编程模型相似。返回的应答消息包含了返回的结果，或者它会将返回值转换为一般方法的返回值。此外，如果存在通信异常或服务端异常，代理会在客户端抛出一个异常。除了NetPeerTcpBinding和NetMsmqBinding绑定，所有的绑定均支持请求-应答操作。

注1： 本章引用了作者在MSDN杂志2006年10月刊中发表的文章《WCF Essentials: What You Need to Know About One\_Way Calls, Callbacks, and Events》。



## 单向操作

一种情形是操作没有返回值，客户端也不会关心调用成功与否。为支持这种即发即弃的调用方式，WCF提供了单向操作。一旦客户端发出调用，WCF会生成一个请求消息，但却没有相关的应答消息返回给客户端。因此，单向操作不能返回值，服务端抛出的任何异常都不会传递给客户端。理想状态下，一旦客户端调用了一个单向方法，它只会在分发调用的一瞬间被阻塞。但事实上，单向调用并不等同于异步调用。当单向调用到达服务端时，不会立即分发这些调用，而是可能放入到服务端的队列中，并在某个时间被分发。这一过程要根据服务配置的并发模式行为（Concurrency Mode Behavior）而定。（第8章将深入探讨并发管理与单向调用的相关知识。）服务要放入到队列中的消息个数（单向操作或请求-应答操作的消息），与配置的通道及可靠性模式有关。如果队列消息的数量超出了队列容量，就会阻塞客户端，即使发出的只是单向调用。然而，一旦调用被放入队列（这是最常见的方式），就会取消对客户端的阻塞，继续执行。同时，服务会在后台处理这一操作。所有的 WCF 绑定均支持单向操作。

## 配置单向操作

OperationContract 特性定义了 Boolean 类型的 IsOneWay 属性：

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public bool IsOneWay
    {
        get; set;
    }
    // 更多成员
}
```

IsOneWay 的默认值为 false，即默认操作为请求-应答操作（这也是 WCF 默认的操作类型）。如果将 IsOneWay 属性设置为 true，方法就会成为单向操作：

```
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}
```

调用单向操作时，客户端并无任何特别或不同之处。IsOneWay 属性的值会包含在服务元数据中。注意服务契约定义与客户端导入定义中的 IsOneWay 值是相同的。

由于单向操作没有应答消息，因此它不能包含返回值或返回结果。例如，下面的单向操作定义就是无效的，因为它返回了值：

```
// 无效契约
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    int MyMethod();
}
```

事实上,在装载宿主时,WCF会强制要求验证方法的签名。如果不匹配,就会抛出一个 `InvalidOperationException` 异常。

## 单向操作与可靠性

客户端不关心调用的结果,并不意味着它不关心调用是否发生。总而言之,即使采用单向调用,我们也必须保障服务的可靠性。它能够确保请求正确地传递到服务。不过对于单向调用而言,客户端不会考虑单向操作的调用顺序。WCF之所以允许开发者将有效的可靠传递从有效的有序传递与执行中分离出来,这是其中一个主要原因。显然,客户端与服务必须就这些细节在之前达成一致,否则绑定配置会不匹配。

## 单向操作与会话服务

WCF 允许开发者设计一个具有单向操作的会话契约:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod()
}
```

如果客户端发出一个单向调用,在执行方法的同时会关闭代理,然后阻塞客户端直到操作完成。

然而,在总体上我们认为,在一个会话契约中包含一个单向操作,无疑是一种糟糕的设计。因为拥有一个会话往往意味着服务需要管理代表了客户端的状态。任何异常的发生都可能破坏这个状态,而客户端却无法觉察。此外,客户端(或者服务)之所以选择一个会话交互,是因为它使用的契约需要通过某个状态机(State Machine)完成锁步(Lock-Step)执行。单向调用无法满足这种模式的要求。因此,我的建议是我们只能将单向操作应用到单调服务或单例服务中。

如果在会话契约中定义了单向操作,就必须保证单向操作是终止会话的最后一个操作(该操作必须遵循单向操作的规定,例如返回 `void` 类型值)。这可以通过分步操作来实现:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IOrderManager
{
    [OperationContract]
    void SetCustomerId(int customerId);

    [OperationContract(IsInitiating = false)]
    void AddItem(int itemId);

    [OperationContract(IsInitiating = false)]
    decimal GetTotal();

    [OperationContract(IsOneWay = true, IsInitiating = false,
                      IsTerminating = true)]
    void ProcessOrders();
}
```

## 单向操作与异常

单向操作不是汽车行驶的单行道，也不像“黑洞”那样有去无回。如要这样认为，无疑会“差之毫厘，谬以千里”。首先，在分发一个单向操作时，因为通信问题（例如地址错误或宿主不可用）导致在分发调用时出现的错误，在试图调用操作时，会在客户端抛出一个异常。其次，根据使用的服务实例模式与绑定方式，客户端可能会受到服务端异常的影响。以下内容假定服务不会抛出第6章介绍的 `FaultException` 异常或者它的子类。

## 单调服务与单向异常

在单调服务中，如果没有传输会话（例如使用 `BasicHttpBinding` 绑定或者不包含可靠消息传输与安全的 `WSHttpBinding` 绑定），当调用一个单向操作时，如果发生异常，客户端并不会受到影响，继续发出对相同代理实例的调用：

```
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MethodWithError();
    [OperationContract]
    void MethodWithoutError();
}

class MyService : IMyContract
{
    public void MethodWithError()
    {
        throw new Exception();
    }
    public void MethodWithoutError()
    {}
}
```

```
}  
// 使用基础绑定时的客户端:  
MyContractClient proxy = new MyContractClient();  
proxy.MethodWithError();  
proxy.MethodWithoutError();  
proxy.Close();
```

但是, 如果使用具有安全性的WSHttpBinding绑定, 或者使用不包含可靠消息传输的NetTcpBinding或NetNamedPipeBinding绑定, 那么一个服务端的异常, 包括单向操作抛出的异常, 就会导致通道错误。此时, 客户端不能发出任何一个使用相同代理实例的新的调用:

```
[ServiceContract]  
interface IMyContract  
{  
    [OperationContract(IsOneWay = true)]  
    void MethodWithError();  
  
    [OperationContract]  
    void MethodWithoutError();  
}  
  
class MyService : IMyContract  
{  
    public void MethodWithError()  
    {  
        throw new Exception();  
    }  
    public void MethodWithoutError()  
    {}  
}  
  
// 使用TCP 或者 IPC 绑定时的客户端:  
MyContractClient proxy = new MyContractClient();  
proxy.MethodWithError();  
try  
{  
    proxy.MethodWithoutError(); // 因为通道错误, 从而被抛出  
    proxy.Close();  
}  
catch  
{}
```

客户端甚至不能安全地关闭代理。

在使用具有可靠消息传输的WSHttpBinding绑定或NetTcpBinding绑定时, 异常不会导致通道错误。此时, 客户端能够继续发出调用。

我们发现彼此之间存在的矛盾至少影响了开发者的判断。首先, 绑定的选择不仅会影响客户端的代码, 而且由于单向操作存在语义上的冲突, 使得调用者能够发现在单向调用期间服务出现的某些错误。

---

注意：从这个角度来看，无会话的单例服务与单调服务相似。

---

## 会话服务与单向异常

论及会话服务在单向方法中抛出的异常，情形就变得更加复杂了。如果服务使用的绑定为 `NetTcpBinding` 或 `NetNamedPipeBinding`，则异常会终止会话，WCF 将释放服务实例，通道也将产生错误。后续的操作调用如果使用相同的代理，会引发 `CommunicationException` 异常（如果 TCP 和 WS 绑定启用了可靠性，则引发 `CommunicationObjectFaultedException` 异常），因为它不再包含会话与服务实例。在抛出异常之后，如果只能调用代理的 `Close()` 方法，则 `Close()` 会抛出 `CommunicationException` 异常（或 `CommunicationObjectFaultException` 异常）。如果在错误发生之前客户端关闭了代理，`Close()` 方法就会被阻塞直到错误发生，然后 `Close()` 会抛出异常。上述复杂过程足以解释为什么要避免在会话服务中使用单向调用。

如果使用包含传输会话的 WS 绑定，异常会导致通道错误，客户端也不能通过代理发出新的调用。在调用抛出异常后，会立即关闭代理，这与其他绑定相似。

---

注意：从这一角度看，具有会话的单例服务与会话服务相似。

---

## 回调操作

WCF 支持服务将调用返回给它的客户端。在回调期间，表中列出的许多方面都将颠倒过来：服务成为客户端，客户端成为服务（参见图 5-1）。

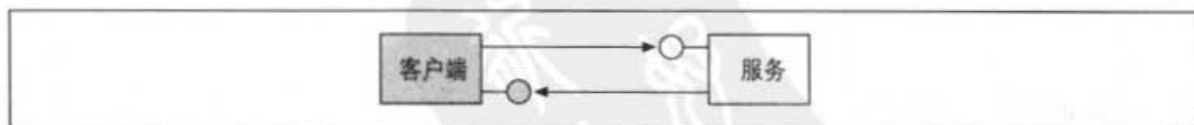


图 5-1：回调允许服务将调用返回给客户端

客户端必须为回调对象的托管提供易于实现的机制。回调操作适用于各种场景与应用程序，尤其适用于事件，在服务端触发事件时使用回调操作通知客户端。并非所有的绑定都支持回调操作，只有具有双向能力的绑定才能够用于回调。例如，HTTP 本质上是与连接无关的，所以它不能用于回调，因此，我们不能基于 `BasicHttpBinding` 或 `WSHttpBinding` 绑定使用回调。WCF 为 `NetTcpBinding` 和 `NetNamedPipeBinding` 提

供了对回调的支持，因为从本质上讲，TCP 和 IPC 协议均支持双向通信。为了让 HTTP 协议支持回调，WCF 提供了 `WSDualHttpBinding` 绑定，它实际上设置了两个 HTTP 通道：一个用于从客户端到服务的调用，另一个则用于服务到客户端的调用。

## 回调契约

回调操作是服务契约的一部分，它取决于服务契约对回调契约的定义。一个服务契约最多只能包含一个回调契约。一旦定义了回调契约，就需要客户端支持回调，并在每次调用中提供指向服务的回调终结点。若要定义回调契约，`ServiceContract` 特性提供了 `Type` 类型的属性 `CallbackContract`：

```
[AttributeUsage(AttributeTargets.Interface|AttributeTargets.Class)]
public sealed class ServiceContractAttribute : Attribute
{
    public Type CallbackContract
    {get;set;}
    // 更多成员
}
```

在定义包含回调契约的服务契约时，需要为 `ServiceContract` 特性提供回调契约的类型，以及回调契约的定义，如例 5-1 所示。

例 5-1：定义和配置回调契约

```
interface ISomeCallbackContract
{
    [OperationContract]
    void OnCallback();
}

[ServiceContract(CallbackContract = typeof(ISomeCallbackContract))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();
}
```

注意，回调契约不必标记 `ServiceContract` 特性，因为类型只要被定义为回调契约，就意味着它具有了 `ServiceContract` 特性，并且在服务元数据中也将包含该特性。当然，我们仍然需要为所有的回调接口方法标记 `OperationContract` 特性。

一旦客户端导入了回调契约的元数据，导入的回调接口与原来的服务端定义的名字是不相同的，它会被修改为服务契约接口名加后缀 `Callback`。例如，如果客户端导入了例 5-1 的定义，客户端会获得到这样的定义：

```
interface IMyContractCallback
{

```

```
[OperationContract]
void OnCallback();
}
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();
}
```

---

注意：为简便起见，建议在服务端为回调契约命名为：服务契约接口名称加上 Callback 后缀。

---

## 客户端回调设置

客户端负责托管回调对象以及公开回调终结点。回想第 1 章的内容，服务实例最内层的执行范围是实例上下文。InstanceContext 类定义的构造函数，能够将服务实例传递给宿主：

```
public sealed class InstanceContext : CommunicationObject, ...
{
    public InstanceContext(object implementation);
    public object GetServiceInstance();
    // 更多成员
}
```

为了托管一个回调对象，客户端需要实例化回调对象，然后通过它创建一个上下文对象：

```
class MyCallback : IMyContractCallback
{
    public void OnCallback()
    { ... }
}
IMyContractCallback callback = new MyCallback();
InstanceContext context = new InstanceContext(callback);
```

值得一提的是虽然回调方法是在客户端，但它们仍然属于 WCF 操作，因此它们都包含了一个操作调用上下文中，通过 OperationContext.Current 可以访问。

## 双向代理

无论何时，只要服务终结点的契约定义了一个回调契约，在与它进行交互时，客户端都必须使用代理创建双向通信，并将回调终结点的引用传递给服务。为此，客户端使用的代理必须继承一个专门的代理类 DuplexClientBase<T>，如例 5-2 所示。



## 例 5-2: DuplexClientBase&lt;T&gt; 类

```

public interface IDuplexContextChannel : IContextChannel
{
    InstanceContext CallbackInstance
    {get;set;}
    // 更多成员
}

public abstract class DuplexClientBase<T> : ClientBase<T> where T : class
{
    protected DuplexClientBase(InstanceContext callbackContext);
    protected DuplexClientBase(InstanceContext callbackContext,
                                string endpointName);
    protected DuplexClientBase(InstanceContext callbackContext,
                                Binding binding,
                                EndpointAddress remoteAddress);
    protected DuplexClientBase(object callbackInstance);
    protected DuplexClientBase(object callbackInstance,
                                string endpointConfigurationName);
    protected DuplexClientBase(object callbackInstance, Binding binding,
                                EndpointAddress remoteAddress);

    public IDuplexContextChannel InnerDuplexChannel
    {get;}
    // 更多成员
}

```

客户端需要为 DuplexClientBase<T> 的构造函数提供托管回调对象（以及具有一个常规代理的服务终结点信息）的实例上下文。代理会围绕回调上下文创建一个终结点，同时根据服务终结点的配置，可以推断出回调终结点的细节：服务契约的回调类型定义了回调终结点的契约。回调终结点会使用相同的绑定和传输方式作为传出调用。WCF 使用客户端机器名称作为地址，如果使用 HTTP，还会选择一个端口号。即使只是将实例上下文传递给双向代理，以及使用代理调用服务都会公开客户端的回调终结点。为了简化这一过程，DuplexClientBase<T> 提供了专门的构造函数，可以直接接收回调对象，并在内部为它包裹一个上下文对象。不管出于何种原因，客户端都需要访问上下文，因此，DuplexClientBase<T> 为客户端提供了类型为 IDuplexContextChannel 的 InnerDuplexChannel 属性，该属性通过 CallbackInstance 属性提供上下文。

使用 SvcUtil 或者 Visual Studio 2005，能够为包含了回调契约的目标服务生成代理类，生成的类派生自 DuplexClientBase<T>，如例 5-3 所示。

## 例 5-3: 工具生成的双向代理

```

partial class MyContractClient : DuplexClientBase<IMyContract>,IMyContract
{
    public MyContractClient(InstanceContext callbackContext) : base(callbackContext)
    {}

    public MyContractClient(InstanceContext callbackContext,string endpointName) :
        base(callbackContext,endpointName)
    {}
}

```

```
{  
public MyContractClient(InstanceContext callbackContext, Binding binding,  
                        EndpointAddress remoteAddress) :  
                        base(callbackContext, binding, remoteAddress)  
{  
    // 更多成员  
  
    public void DoSomething()  
    {  
        Channel.DoSomething();  
    }  
}
```

客户端可以使用派生的代理类创建回调实例，并将上下文作为它的宿主，创建代理，调用服务，这样就可以传递回调终结点的引用：

```
class MyCallback : IMyContractCallback  
{  
    public void OnCallback()  
    {...}  
}  
IMyContractCallback callback = new MyCallback();  
InstanceContext context = new InstanceContext(callback);  
  
MyContractClient proxy = new MyContractClient(context);  
proxy.DoSomething();
```

注意，只要客户端正在等待回调，就不能关闭代理。如果关闭回调终结点，当服务试图将调用返回时，就会导致服务端产生错误。

最常见的实现方式是由客户端自身实现回调契约，此时，客户端通常可以将代理定义为成员变量，并在释放客户端时关闭它，如例 5-4 所示。

#### 例 5-4：客户端实现回调契约

```
class MyClient : IMyContractCallback, IDisposable  
{  
    MyContractClient m_Proxy;  
  
    public void CallService()  
    {  
        InstanceContext context = new InstanceContext(this);  
        m_Proxy = new MyContractClient(context);  
        m_Proxy.DoSomething();  
    }  
    public void OnCallback()  
    {...}  
  
    public void Dispose()  
    {  
        m_Proxy.Close();  
    }  
}
```

需要注意的是，生成的代理无法利用 `DuplexClientBase<T>` 类的简化构造函数，即使它能够直接接收回调对象。但是，我们可以手动修改代理类，添加对它的支持，如例 5-5 所示。

例 5-5：使用修改后的基于对象的代理

```
partial class MyContractClient : DuplexClientBase<IMyContract>,IMyContract
{
    public MyContractClient(object callbackInstance) : base(callbackInstance)
    {
        // 更多构造函数
    }
    public void DoSomething()
    {
        Channel.DoSomething();
    }
}
class MyClient : IMyContractCallback,IDisposable
{
    MyContractClient m_Proxy;

    public void CallService()
    {
        m_Proxy = new MyContractClient(this);
        m_Proxy.DoSomething();
    }
    public void OnCallback()
    { ... }
    public void Dispose()
    {
        m_Proxy.Close();
    }
}
```

## 服务端回调调用

随着客户端的每一次调用，客户端的回调终结点引用都会被传递到服务，组成传入消息的一部分。`OperationContext` 类为服务提供了方便访问回调引用的途径，即调用泛型方法 `GetCallbackChannel<T>()`：

```
public sealed class OperationContext : ...
{
    public T GetCallbackChannel<T>();
    // 更多成员
}
```

服务如何处理回调引用以及何时决定使用它，完全由服务抉择，这一点毋庸置疑。服务能够从操作上下文中提取出回调引用，然后将它保存起来以待今后的使用；或者可以在服务运行期间使用它将调用返回给客户端。例 5-6 演示了第一种情形。

**例 5-6：存储回调引用以待今后使用**

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    static List<ISomeCallbackContract> m_Callbacks =
        new List<ISomeCallbackContract>();

    public void DoSomething()
    {
        ISomeCallbackContract callback = OperationContext.Current.
            GetCallbackChannel<ISomeCallbackContract>();

        if(m_Callbacks.Contains(callback) == false)
        {
            m_Callbacks.Add(callback);
        }
    }

    public static void CallClients()
    {
        Action<ISomeCallbackContract> invoke = delegate(ISomeCallbackContract callback)
        {
            callback.OnCallback();
        };

        m_Callbacks.ForEach(invoke);
    }
}
```

使用与例5-1相同的定义，服务使用了静态的、泛型的链表存储ISomeCallbackContract接口类型的引用。因为服务并不知道哪一个客户端会调用它，也不知道客户端是否已经调用，所以在每次调用时，服务都会检查链表是否已经包含了该回调引用。如果链表没有包含该引用，服务就会将它添加到链表中。服务类同时还提供了静态方法CallClients()。宿主端的任何一方都能够使用该方法将调用返回给客户端：

```
MyService.CallClients();
```

以这种方式进行调用，调用方需要使用某个宿主端线程执行回调调用。该线程与执行传入服务调用的线程无关。

---

**注意：**例5-6（以及在本章中相似的例子）并没有通过同步访问回调列表。显然，真实的应用程序代码正需要如此。并发管理（特别的，还包括同步访问共享资源）的内容将在第8章讨论。

---

**回调重入**

服务可能需要调用回调引用，调用的回调引用可能通过传递，也可能是在契约操作执行期间保存的副本。然而，这样的调用在默认情况下却是禁止的，因为它受到默认的服务

并发管理的限制(译注1)。在默认情况下,服务类被配置为单线程访问:服务实例与锁关联,在任何时刻都只能有一个线程拥有锁,也只能有一个线程能够访问服务实例。在操作调用期间,向客户端发出的调用需要阻塞服务线程,并调用回调。问题是一旦回调要返回它所需要的同一个锁的所有权,则处理从客户端返回的应答消息就会导致死锁(译注2)。注意,服务仍然可能调用到其他客户端的回调,或者调用其他服务。这属于正在调用的会导致死锁的客户端的回调。

如果单线程的服务实例试图将调用返回给它的客户端,为了避免死锁,WCF会抛出一个 `InvalidOperationException` 异常。这里有三种可能的解决方案。第一种方案是配置服务允许多线程访问,由于它与锁无关,因此允许回调。但是,也可能会增加服务开发者的负担,因为它需要为服务提供同步。第二种方案是将服务配置为重入(*Reentrancy*)。一旦配置为重入,则服务实例仍然与锁关联,同时只允许单线程访问。然而,如果服务正在回调它的客户端,WCF就会首先释放锁。第8章专门讲解了同步模式以及同步模式编程模型。就目前而言,如果服务需要回调它的客户端,可以使用 `ServiceBehavior` 特性的 `ConcurrencyMode` 属性,将服务的并发行为配置为多线程或者重入:

```
public enum ConcurrencyMode
{
    Single, // 默认值
    Reentrant,
    Multiple
}

[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : ...
{
    public ConcurrencyMode ConcurrencyMode
    {get;set;}
    // 更多成员
}
```

例5-7 演示了一个配置为重入的服务。在操作执行期间,服务访问操作上下文,获取并

---

译注1: 默认的服务并发管理为 `ConcurrencyMode.Single`, 也就是说禁止并发调用的。

译注2: 简单的说,就是指当回调的应答消息也需要获得与服务实例关联的相同的锁时,就会导致死锁。因为此时服务线程已经被阻塞,服务操作正在等待回调操作执行完毕,而回调操作却又在等待服务释放锁,自然会产生锁的争用。这种情况正可以解释后面介绍的第三种方案的可行性。第三种方案是将回调操作设置为单向操作,此时,回调调用不会产生应答消息,服务操作一旦执行了回调操作,就会继续执行,回调对象不会争用与服务实例关联的锁,从而解决了死锁问题。

调用回调引用。一旦回调返回，控制权只会返回到服务，而服务自身的线程则需要重新确认锁（译注3）。

#### 例 5-7：配置重入以支持回调

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();
}
interface IMyContractCallback
{
    [OperationContract]
    void OnCallback();
}
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
class MyService : IMyContract
{
    public void DoSomething()
    {
        IMyContractCallback callback = OperationContext.Current.
            GetCallbackChannel<IMyContractCallback>();
        callback.OnCallback(); (译注4)
    }
}
```

第三种解决方案是将回调契约操作配置为单向操作，这样服务就能够安全地将调用返回给客户端。因为没有任何应答消息会竞用锁，即使并发被设置为单线程，服务也能够支持回调。例 5-8 演示了这样的配置。注意，服务默认为单线程并发模式。

#### 例 5-8：默认支持单向回调

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();
}
```

译注3：所谓“重入”，是指对同步域拥有独占访问权的线程A调用了同步域之外对象的方法，此时，另外的线程B若要访问该同步域，则线程A将释放对同步域的锁，允许线程B进入。直到线程B执行完毕并释放对同步域的锁后，线程A将重新进入该同步域。配置回调为重入时，因为服务对象是与线程关联的，属于同步域的对象，而回调对象则属于同步域之外的对象。由于服务被配置为重入，则服务调用回调引用时会释放锁。然后将回调返回给客户端，控制权则返回给服务，服务会重入并重新获取锁。

译注4：执行回调，此时调用会返回给客户端，而控制权则返回给服务。

```
interface IMyContractCallback
{
    [OperationContract(IsOneWay = true)]
    void OnCallback();
}
class MyService : IMyContract
{
    public void DoSomething()
    {
        IMyContractCallback callback = OperationContext.Current.
            GetCallbackChannel<IMyContractCallback>();
        callback.OnCallback();
    }
}
```

## 回调连接管理

回调机制并不提供管理服务与回调终结点之间连接的高层协议。它取决于开发者提供的应用层协议，或者遵循管理连接的生命周期的一致模式。正如之前提到的那样，如果客户端通道仍然保持打开状态，服务只能将调用返回给客户端，在通常情况下，这建立在代理没有关闭的前提之上。保证代理处于打开状态同样可以避免回调对象被垃圾回收器回收。如果服务维持了一个在回调终结点上的引用，而且客户端代理是关闭的，或者客户端应用程序已经退出，那么当服务调用回调时，就会从服务通道处获得一个 `ObjectDisposedException` 异常。因此，对于客户端而言，当它不再需要接收回调或者客户端应用程序已经关闭时，最好能够通知服务。为此，我们可以在服务契约中显式地添加一个 `Disconnect()` 方法。既然每个方法调用都带有回调引用，那么服务就能够在 `Disconnect()` 方法中将回调引用从内部存储结构中移除。

另外，出于对称的目的，建议开发者同时在服务契约中显式添加 `Connect()` 方法。定义 `Connect()` 方法，能够使客户端重复地连接或断开，同时还提供了一个明确的时间点，以判断何时需要一个回调，因为回调只能够发生在调用 `Connect()` 方法之后。例 5-9 演示了这样的技术。在 `Connect()` 和 `Disconnect()` 方法中，服务都需要获取回调引用。在 `Connect()` 方法中，服务在将回调引用添加到列表（列表保证了客户端能够多次调用 `Connect()` 方法）之前，需要验证回调列表是否已经包含了该回调引用。在 `Disconnect()` 方法中，服务会验证列表是否包含该回调引用，如果没有则抛出异常。

### 例 5-9：显式的回调连接管理

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();

    [OperationContract]
    void Connect();
}
```



```
[OperationContract]
void Disconnect();
}
interface IMyContractCallback
{
    [OperationContract]
    void OnCallback();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    static List<IMyContractCallback> m_Callbacks = new List<IMyContractCallback>();
    public void Connect()
    {
        IMyContractCallback callback = OperationContext.Current.
            GetCallbackChannel<IMyContractCallback>();
        if(m_Callbacks.Contains(callback) == false)
        {
            m_Callbacks.Add(callback);
        }
    }
    public void Disconnect()
    {
        IMyContractCallback callback = OperationContext.Current.
            GetCallbackChannel<IMyContractCallback>();
        if(m_Callbacks.Contains(callback) == true)
        {
            m_Callbacks.Remove(callback);
        }
        else
        {
            throw new InvalidOperationException("Cannot find callback");
        }
    }
    public static void CallClients()
    {
        Action<IMyContractCallback> invoke = delegate(IMyContractCallback callback)
        {
            callback.OnCallback();
        };
        m_Callbacks.ForEach(invoke);
    }
    public void DoSomething()
    {
        ...
    }
}
```

### 连接管理与实例模式

只有在操作调用回调引用对象自身，或者将它存储在某些全局变量例如静态变量时，单调服务才能够使用回调引用，方法如前面介绍的实例所示。原因显而易见，当操作返回时服务可能使用的存储在引用对象中的任何实例状态均会被丢失。因而，单调服务特别

需要类似于 `Disconnect()` 的方法 (译注 5)。单例服务同样存在类似的情况。由于单例对象的生命周期不会结束, 因而它会累积回调引用的数目, 随着时间推移, 回调的客户端不复存在, 大多数回调引用也将随之失效。定义 `Disconnect()` 方法可以保证单例服务只连接到对应的活动客户端。

需要注意的是, 会话服务即使没有 `Disconnect()` 方法也能够获得回调引用, 只要它将回调引用保留在某个实例成员变量中。原因在于当会话结束时 (客户端关闭代理或者超时), 服务实例会被自动释放, 整个会话不存在保持引用的危险。这就能够保证回调引用总是有效的。但是, 如果会话服务为了让其他宿主端或跨会话访问, 而将它的回调引用保存在某个全局变量中, 就必须添加 `Disconnect()` 方法, 以达到移除回调引用的目的, 因为在调用 `Dispose()` 期间不能使用回调引用。

最后, 考虑到将来可能的扩展, 我们需要为会话服务添加配对的 `Connect()` 和 `Disconnect()` 方法, 因为它允许客户端在会话期间决定何时启动或停止接收回调。

## 双向代理与类型安全

WCF 提供的 `DuplexClientBase<T>` 没有为使用的回调接口提供强类型。编译器允许开发者传递任何对象, 甚至是无效的回调接口。编译器甚至允许开发者将那些没有回调契约定义的服务契约类型作为 `T` 参数类型。运行时, 我们能够成功地初始化代理。只有当我们试图使用代理时, 才会发现两者之间互不兼容, 从而引发 `InvalidOperationException` 异常。大致相同的是, `InstanceContext` 是基于 `object` 对象类型的, 在编译时同样不会检验它是否实际包含了一个有效的回调契约实例。将它作为构造函数参数传递给双向代理时, 并没有对 `InstanceContext` 与双向代理所期望的回调实例之间的相关性进行编译时验证。当我们试图使用代理时, 就会发现错误。在实例化代理对象时, 使用泛型能够在一定程度上弥补运行时的疏漏以及发现运行时的错误。

首先, 定义类型安全的 `InstanceContext<T>` 泛型类, 如例 5-10 所示。

例 5-10: `InstanceContext<T>` 类

```
public class InstanceContext<T>
{
    InstanceContext m_InstanceContext;

    public InstanceContext(T implementation)
    {
        m_InstanceContext = new InstanceContext(implementation);
    }
}
```

译注 5: 前提是单调服务保存了状态, 如第 4 章所述。

```

public InstanceContext Context
{
    get
    {
        return m_InstanceContext;
    }
}
public T ServiceInstance
{
    get
    {
        return (T)m_InstanceContext.GetServiceInstance();
    }
}
}

```

使用泛型，我们就可以为托管的回调对象提供类型安全的访问，获得需要的回调类型。

接下来定义一个新类，它是DuplexClientBase<T>类的泛型子类，能够保证类型安全，如例 5-11 所示。

#### 例 5-11: DuplexClientBase<T,C> 类

//T 为服务契约，C 为回调契约

```

public abstract class DuplexClientBase<T,C> : DuplexClientBase<T> where T :
class
{
    protected DuplexClientBase(InstanceContext<C> context) : base(context.Context)
    {}
    protected DuplexClientBase(InstanceContext<C> context,string endpointName) :
        base(context.Context,endpointName)
    {}
    protected DuplexClientBase(InstanceContext<C> context,Binding binding,
        EndpointAddress remoteAddress) :
        base(context.Context,binding,remoteAddress)
    {}
    protected DuplexClientBase(C callback) : base(callback)
    {}
    protected DuplexClientBase(C callback,string endpointName) :
        base(callback,endpointName)
    {}
    protected DuplexClientBase(C callback,Binding binding,
        EndpointAddress remoteAddress) :
        base(callback,binding,remoteAddress)
    {}

    /* 更多构造函数 */

    static DuplexClientBase()
    {
        VerifyCallback();
    }
    internal static void VerifyCallback()
    {

```

```

Type contractType = typeof(T);
Type callbackType = typeof(C);

object[] attributes = contractType.GetCustomAttributes(
    typeof(ServiceContractAttribute), false);
if(attributes.Length != 1)
{
    throw new InvalidOperationException("Type of " + contractType +
        " is not a service contract");
}
ServiceContractAttribute serviceContractAttribute;
serviceContractAttribute = attributes[0] as ServiceContractAttribute;
if(callbackType != serviceContractAttribute.CallbackContract)
{
    throw new InvalidOperationException("Type of " + callbackType +
        " is not configured as callback contract for " + contractType);
}
}
}

```

DuplexClientBase<T,C> 类使用了两个类型参数：T 为服务契约类型参数，C 为回调契约类型参数。DuplexClientBase<T,C> 的构造函数既能够接收一个 C 实例，也能够接收一个包裹了 C 实例的 InstanceContext<C> 实例。这就使得编译器能够确保使用的上下文是兼容的。然而，C# 2.0 无法约束 T 与 C 之间的声明关系。在使用 DuplexClientBase<T,C> 之前，需要执行单独的运行时检查，并在可能发生错误之前，立即取消对错误类型的使用。实现技巧是将运行时验证放到 C# 的静态构造函数中。DuplexClientBase<T,C> 的静态构造函数调用静态的辅助方法 VerifyCallback()。VerifyCallback() 方法利用了反射技术，首先验证 T 是否标记了 ServiceContract 特性，然后验证它所包含的回调契约类型是否为类型参数 C。通过在静态构造函数中抛出异常的方式，可以在运行时尽可能早地发现错误。

---

**注意：**在静态构造函数中执行回调契约的验证是可行的。对于那些无法在编译时验证的约束，可以通过这样的编程方法对它进行运行时的判断和约束。

---

接下来，我们需要修改机器自动生成的客户端代理类，让它继承类型安全的 DuplexClientBase<T,C> 类：

```

partial class MyContractClient : DuplexClientBase<IMyContract,IMyContractCallback>,
    IMyContract
{
    public MyContractClient(InstanceContext<IMyContractCallback> context)
        : base(context)
    {}
    public MyContractClient(IMyContractCallback callback) : base(callback)
    {}
}

```

```
/* 其余构造函数 */  
  
public void DoSomething()  
{  
    Channel.DoSomething();  
}  
}
```

我们既可以为修改后的代理提供类型安全的实例上下文，也可以直接提供回调实例：

```
// 客户端代码  
class MyClient : IMyContractCallback  
{...}  
  
IMyContractCallback callback = new MyClient();  
MyContractClient proxy1 = new MyContractClient(callback);  
  
InstanceContext<IMyContractCallback> context = new  
    InstanceContext<IMyContractCallback>(callback);  
MyContractClient proxy2 = new MyContractClient(context);
```

不管怎样，编译器都会验证提供给代理的类型参数是否匹配上下文类型参数或回调实例，静态构造函数则会在实例化时验证服务契约与回调实例之间的关系。

## 双向通道工厂

与 `ChannelFactory<T>` 类相似，WCF 同样提供了 `DuplexChannelFactory<T>` 类，它被用于通过编程方式设置双向代理：

```
public class DuplexChannelFactory<T> : ChannelFactory<T>  
{  
    public DuplexChannelFactory(object callback);  
    public DuplexChannelFactory(object callback, string endpointName);  
    public DuplexChannelFactory(InstanceContext context, string endpointName);  
  
    public T CreateChannel(InstanceContext context);  
    public static T CreateChannel(object callback, string endpointName);  
    public static T CreateChannel(InstanceContext context, string endpointName);  
    public static T CreateChannel(object callback, Binding binding,  
        EndpointAddress endpointAddress);  
    public static T CreateChannel(InstanceContext context, Binding binding,  
        EndpointAddress endpointAddress);  
  
    // 更多成员  
}
```

`DuplexChannelFactory<T>` 的用法类似与基类 `ChannelFactory<T>`，不同的是，它的构造函数既能接收回调实例，又能接收回调上下文。需要注意 `DuplexChannelFactory<T>` 将 `object` 类型作为回调实例，因而缺乏类型安全。类似于对 `DuplexClientBase<T>` 类的修改，例 5-12 演示了修改过的 `DuplexChannelFactory<T, C>` 类，它同时提供了编译时与运行时的类型安全。

## 例 5-12: DuplexChannelFactory&lt;T,C&gt; 类

```

public class DuplexChannelFactory<T,C> : DuplexChannelFactory<T> where T : class
{
    static DuplexChannelFactory()
    {
        DuplexClientBase<T,C>.VerifyCallback(); (译注 6)
    }
    public static T CreateChannel(C callback,string endpointName)
    {
        return DuplexChannelFactory<T>.CreateChannel(callback,endpointName);
    }
    public static T CreateChannel(InstanceContext<C> context,string endpointName)
    {
        return DuplexChannelFactory<T>.CreateChannel(context.Context,endpointName);
    }
    public static T CreateChannel(C callback,Binding binding,
                                   EndpointAddress endpointAddress)
    {
        return DuplexChannelFactory<T>.CreateChannel(callback,binding,
                                                         endpointAddress);
    }
    public static T CreateChannel(InstanceContext<C> context,Binding binding,
                                   EndpointAddress endpointAddress)
    {
        return DuplexChannelFactory<T>.CreateChannel(context,binding,
                                                         endpointAddress);
    }
    public DuplexChannelFactory(C callback) : base(callback)
    {}
    public DuplexChannelFactory(C callback,string endpointName):
                                   base(callback,endpointName)
    {}
    public DuplexChannelFactory(InstanceContext<C> context,string endpointName) :
                                   base(context.Context,endpointName)
    {}
    // 更多构造函数
}

```

作为使用双向通道工厂的一个示例，考虑例 5-13 所示的代码，它为第 1 章介绍的 InProcFactory 静态辅助类添加了回调功能。

## 例 5-13: 为 InProcFactory 添加双向支持 (译注 7)

```

public static class InProcFactory
{
    public static I CreateInstance<S,I,C>(C callback) where I : class
                                                where S : class,I
    {

```

译注 6: VerifyCallback() 方法是 internal 的，因而 DuplexChannelFactory<T,C> 类与 DuplexClientBase<T,C> 必须放在一个程序集中，否则无法调用。

译注 7: 类型参数 S 为服务，I 为契约，C 为回调。

```

        InstanceContext<C> context = new InstanceContext<C>(callback);
        return CreateInstance<S,I,C>(context);
    }
    public static I CreateInstance<S,I,C>(InstanceContext<C> context)
                                                where I : class
                                                where S : class,I
    {
        HostRecord hostRecord = GetHostRecord<S,I>();
        return DuplexChannelFactory<I,C>.CreateChannel(
            context,NamedPipeBinding,new EndpointAddress(hostRecord.Address));
    }
    // 更多成员
}
// 客户端实例
IMyContractCallback callback = new MyClient();

IMyContract proxy = InProcFactory.CreateInstance
                    <MyService,IMyContract,IMyContractCallback>(callback);
proxy.DoSomething();
InProcFactory.CloseProxy(proxy);

```

## 回调契约层级 (译注 8)

设计回调契约时,需要注意设计上的约束。如果一个服务契约的基契约定义了回调接口,则该服务契约定义的回调接口就必须是它的基契约定义的所有回调接口的子接口。例如,以下的回调契约就是无效的:

```

interface ICallbackContract1
{...}

interface ICallbackContract2
{...}

[ServiceContract(CallbackContract = typeof(ICallbackContract1))]
interface IMyBaseContract
{...}

// 无效
[ServiceContract(CallbackContract = typeof(ICallbackContract2))]
interface IMySubContract : IMyBaseContract
{...}

```

IMySubContract 不能指定 ICallbackContract2 为它的回调契约,因为 ICallbackContract2 不是 ICallbackContract1 的子接口,而 IMyBaseContract (IMySubContract 的基接口)又定义了 ICallbackContract1 作为它自身的回调契约。之所以要如此约束,原因非常明显:如果客户端将一个终结点引用传递给实现了 IMySubContract 接口的服务,那么回调引用就必须符合 IMyBaseContract 接口所期

译注 8: 简单说来,回调契约的层级应与服务契约的层级保持一致。



望的回调类型。WCF 在服务装载时会验证回调契约的层级，如果存在冲突，会抛出 `InvalidOperationException` 异常。

满足约束的最直接办法是在回调契约层级中反映服务契约的层级：

```
interface ICallbackContract1
{...}

interface ICallbackContract2 : ICallbackContract1
{...}

[ServiceContract(CallbackContract = typeof(ICallbackContract1))]
interface IMyBaseContract
{...}

[ServiceContract(CallbackContract = typeof(ICallbackContract2))]
interface IMySubContract : IMyBaseContract
{...}
```

但是，我们也可以让单个回调契约继承多个接口，以避免模仿服务契约层级：

```
interface ICallbackContract1
{...}
interface ICallbackContract2
{...}
interface ICallbackContract3 : ICallbackContract2, ICallbackContract1
{...}

[ServiceContract(CallbackContract = typeof(ICallbackContract1))]
interface IMyBaseContract1
{...}
[ServiceContract(CallbackContract = typeof(ICallbackContract2))]
interface IMyBaseContract2
{...}
[ServiceContract(CallbackContract = typeof(ICallbackContract3))]
interface IMySubContract : IMyBaseContract1, IMyBaseContract2
{...}
```

注意，服务同样能够实现自身的回调契约：

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{...}
[ServiceContract]
interface IMyContractCallback
{...}
class MyService : IMyContract, IMyContractCallback
{...}
```

服务甚至可以将一个引用存储到自身的某些回调存储结构中（如果服务希望像客户端那样被回调）。

## 回调、端口与通道

当我们使用 `NetTcpBinding` 或者 `NetNamedPipeBinding` 绑定时，回调通过绑定所维护的输出通道进入客户端，然后到达服务。它不需要为回调打开一个新的端口或管道。当我们使用 `WSDualHttpBinding` 时，WCF 专门为回调维护了一个单独的 HTTP 通道，因为 HTTP 自身属于单向协议。WCF 为回调通道选择了默认的 80 端口，并将使用 HTTP 的回调地址、客户端机器名称以及端口号 80 传递给服务。

通常，基于 Internet 的服务会使用 80 端口，但对于基于 Intranet 的服务而言，端口值就太小了。此外，如果客户端机器碰巧也运行了 IIS，由于 80 端口已经被占用，客户端将无法托管回调终结点。如果机器上安装了 IIS，并且指定了回调端口，那么在测试与调试期间，IIS 可能会导致冲突。如果是开发基于 Internet 的应用程序，这种冲突会频繁发生。但如果是要求使用 `WSDualHttpBinding` 的 Intranet 应用程序，发生冲突的可能性则比较低。

### 分配回调地址

幸运的是，`WSDualHttpBinding` 绑定提供了 `ClientBaseAddress` 属性，通过它可以为客户端配置不同的回调 URI：

```
public class WSDualHttpBinding : Binding,...
{
    public Uri ClientBaseAddress
    {get;set;}
    // 更多成员
}
```

下面的例子演示了在客户端配置文件中配置基地址的方法：

```
<system.serviceModel>
  <client>
    <endpoint
      address = "http://localhost:8008/MyService"
      binding = "wsDualHttpBinding"
      bindingConfiguration = "ClientCallback"
      contract = "IMyContract"
    />
  </client>
  <bindings>
    <wsDualHttpBinding>
      <binding name = "ClientCallback"
        clientBaseAddress = "http://localhost:8009/"
      />
    </wsDualHttpBinding>
  </bindings>
</system.serviceModel>
```

然而,既然服务不必预先知道回调端口,实际上所有可用的端口都能够作为基地址的端口。因此,最好以编程方式将客户端基地址设置为任意的可用端口。使用例5-14中的WsDualProxyHelper静态辅助类,我们能够自动实现这一步骤。

#### 例 5-14: WsDualProxyHelper 类

```
public static class WsDualProxyHelper
{
    public static void SetClientBaseAddress<T>(DuplexClientBase<T> proxy, int port)
        where T : class
    {
        WSDualHttpBinding binding = proxy.Endpoint.Binding as WSDualHttpBinding;
        Debug.Assert(binding != null);
        binding.ClientBaseAddress = new Uri("http://localhost:" + port + "/");
    }
    public static void SetClientBaseAddress<T>(DuplexClientBase<T> proxy)
        where T : class
    {
        lock(typeof(WsDualProxyHelper))
        {
            int portNumber = FindPort();
            SetClientBaseAddress(proxy, portNumber);
            proxy.Open();
        }
    }
    internal static int FindPort()
    {
        IPEndPoint endPoint = new IPEndPoint(IPAddress.Any, 0);
        using(Socket socket = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream,
            ProtocolType.Tcp))
        {
            socket.Bind(endPoint);
            IPEndPoint local = (IPEndPoint)socket.LocalEndPoint;
            return local.Port;
        }
    }
}
```

WsDualProxyHelper类定义了两个重载版本的SetClientBaseAddress()方法。第一个重载方法仅仅接收代理实例与端口号。它验证了代理是否使用了WSDualHttpBinding绑定,然后使用方法参数传入的端口号设置客户端的基地址。第二个版本的SetClientBaseAddress()方法能够自动选择一个可用的端口,得到的端口值再作为参数值去调用第一个版本的SetClientBaseAddress()方法。为避免在相同的应用程序域中与SetClientBaseAddress()方法的其他并发调用产生争用,在查找可用端口以及设置基地址期间,需要锁定类型本身,然后它会打开代理,锁定端口。注意,如果是相同机器上不同的进程或应用程序域,仍然可能具有这样的争用条件。

WsDualProxyHelper类的使用非常简单:

```
// 实例客户端代码:
class MyClient : IMyContractCallback
{...}

IMyContractCallback callback = new MyClient();
InstanceContext context = new InstanceContext(callback);

MyContractClient proxy = new MyContractClient(context);

WsDualProxyHelper.SetClientBaseAddress(proxy);
```

编程设置回调地址（相对于配置文件中的硬编码方式）的另一个好处是，它支持测试期间在相同机器上启动几个客户端。

### 以声明方式分配回调地址

我们还可以进一步改善这一过程，使用定制特性以声明方式分配回调端口，即使用 `CallbackBaseAddressBehaviorAttribute` 契约行为特性。它只能应用于使用 `WSDualHttpBinding` 绑定的回调终结点。`CallbackBaseAddressBehaviorAttribute` 特性定义了一个整数类型的属性 `CallbackPort`：

```
[AttributeUsage(AttributeTargets.Class)]
public class CallbackBaseAddressBehaviorAttribute : Attribute, IEndpointBehavior
{
    public int CallbackPort
    {get;set;}
}
```

`CallbackPort` 属性的默认值为 80。不需要设置，应用 `CallbackBaseAddressBehavior` 特性就能够为 `WSDualHttpBinding` 生成默认的行为，因此下面的两个定义是等效的：

```
class MyClient : IMyContractCallback
{...}

[CallbackBaseAddressBehavior]
class MyClient : IMyContractCallback
{...}
```

我们可以显式地指定一个回调端口：

```
[CallbackBaseAddressBehavior(CallbackPort = 8009)]
class MyClient : IMyContractCallback
{...}
```

但是，如果将 `CallbackPort` 值设置为 0，`CallbackBaseAddressBehavior` 会为回调自动选择任意的可用端口：

```
[CallbackBaseAddressBehavior(CallbackPort = 0)]
class MyClient : IMyContractCallback
{...}
```

例 5-15 列出了 CallbackBaseAddressBehaviorAttribute 特性的实现代码。

例 5-15: CallbackBaseAddressBehaviorAttribute 特性代码

```
[AttributeUsage(AttributeTargets.Class)]
public class CallbackBaseAddressBehaviorAttribute : Attribute, IEndpointBehavior
{
    int m_CallbackPort = 80;

    public int CallbackPort // 访问 m_CallbackPort
    {get;set;}
    void IEndpointBehavior.AddBindingParameters(ServiceEndpoint endpoint,
                                                BindingParameterCollection bindingParameters)
    {
        if(CallbackPort == 80)
        {
            return;
        }
        lock(typeof(WsDualProxyHelper))
        {
            if(CallbackPort == 0)
            {
                CallbackPort = WsDualProxyHelper.FindPort();
            }
            WSDualHttpBinding binding = endpoint.Binding as WSDualHttpBinding;
            if(binding != null)
            {
                binding.ClientBaseAddress = new Uri(
                    "http://localhost:" + CallbackPort + "/");
            }
        }
    }
    // IEndpointBehavior 的方法并不执行任何工作
}
```

CallbackBaseAddressBehaviorAttribute 属于一个终结点行为特性,它允许开发者截取终结点的配置信息(既可以在客户端截取,也可以在服务端截取)。该特性实现了 IEndpointBehavior 接口:

```
public interface IEndpointBehavior
{
    void AddBindingParameters(ServiceEndpoint endpoint,
                             BindingParameterCollection bindingParameters);
    // 更多成员
}
```

在第一次使用代理访问服务之前, WCF 在客户端调用了 AddBindingParameters() 方法, 它允许特性为回调配置使用的绑定。AddBindingParameters() 方法会检查 CallbackPort 的值。如果端口号为 80, 则什么都不需要做。如果为 0, AddBindingParameters() 方法就会查找一个可用端口, 并将它分配给 CallbackPort。然后,

AddBindingParameters()方法会为调用服务查找使用的绑定。如果使用了 WSDualHttpBinding 绑定, AddBindingParameters()方法就会使用回调端口设置客户端的基地址。

**警告:** 即使是在相同的应用程序域, 如果应用了 CallbackBaseAddressBehaviorAttribute, 仍然可能存在与其他使用相同端口的回调对象之间的争用。

## 事件

基础的 WCF 回调机制并不能阐明客户端与服务之间交互的本质。它们可以是等价交换的交互双方, 既可以发出调用, 也可以接收对方的调用。

然而, 双向回调的一种规范使用则是通过事件。服务端上发生的相关事项都可以通过事件通知客户端或多个客户端。事件可能来源于直接的客户端调用, 也可能来源于服务监听器。激活事件的服务被称为发布者 (Publisher), 而接收事件的客户端则被称为订阅者 (Subscriber)。事件是大多数应用程序所必备的特征, 如图 5-2 所示。

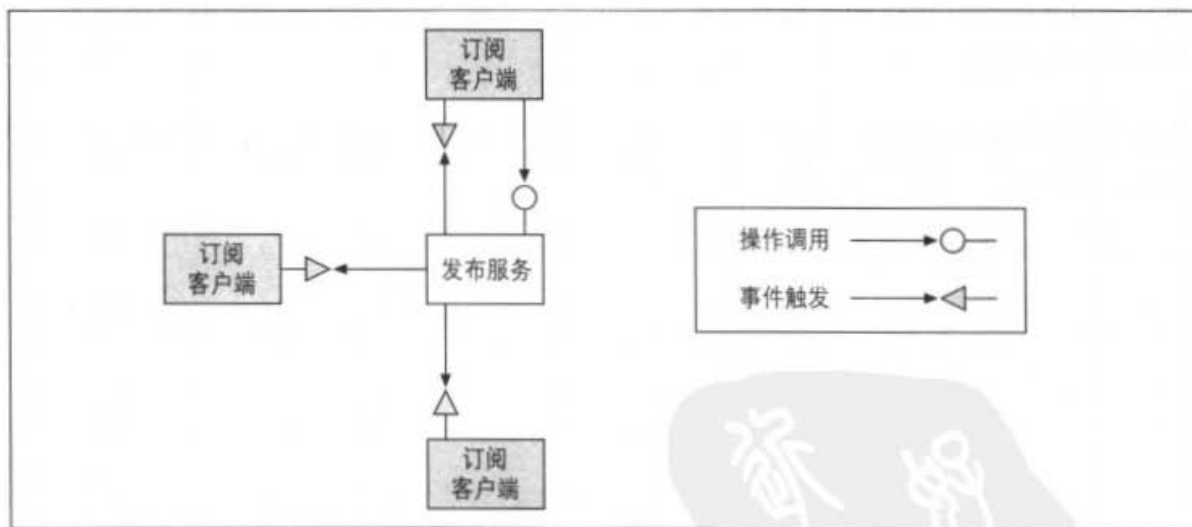


图 5-2: 发布服务能够针对多个订阅客户端触发事件

相比较于回调操作, WCF 更重视对事件的运用。从本质上讲, 事件代表了发布者与订阅者之间更加松散耦合的关系, 它优于客户端和服务之间的关系。处理事件时, 通常服务会为多个订阅客户端发布相同的事件。发布者一般不会考虑订阅者的调用顺序, 也不会考虑订阅者在处理事件时可能出现的错误。所有的发布者都知道它应该将事件传递给订阅者。如果事件存在问题, 服务对此也束手无策。此外, 服务并不关心订阅者返回的结果。因此, 事件处理操作的返回类型应该为 void, 它也不需要任何传出参数, 因而它应

该被标记为单向操作。建议将事件分解为单独的回调契约，而不要在相同的契约中将事件与常规的回调混合在一起。

```
interface IMyEvents
{
    [OperationContract(IsOneWay = true)]
    void OnEvent1();

    [OperationContract(IsOneWay = true)]
    void OnEvent2(int number);

    [OperationContract(IsOneWay = true)]
    void OnEvent3(int number, string text);
}
```

在订阅者端，即使使用了单向回调操作，事件处理方法的实现也应该是执行周期较短的操作。如果我们需要发布大量的事件，以至于超出了订阅者的能力，由于队列正在处理前一个事件，无法将回调放入到队列中，就会导致发布者被阻塞。阻塞发布者能够及时阻止事件到达其他的订阅者。发布者可以为它的契约添加专门的操作，允许客户端显式地订阅事件或取消对事件的订阅。如果发布者支持多个事件类型，也可以允许订阅者选择它们希望订阅或取消订阅的事件。

服务如何从内部管理订阅者列表以及它们的参数选择，完全属于服务端的实现细节，不会影响到客户端。

发布者甚至可以使用 .NET 委托管理订阅者列表，发布自身的行为。例 5-16 演示了这种技术，以及其他之前讨论过的设计要素。

#### 例 5-16：使用委托管理事件

```
enum EventType
{
    Event1 = 1,
    Event2 = 2,
    Event3 = 4,
    AllEvents = Event1 | Event2 | Event3
}
[ServiceContract(CallbackContract = typeof(IMyEvents))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();

    [OperationContract]
    void Subscribe(EventType mask);

    [OperationContract]
    void Unsubscribe(EventType mask);
}
```



```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyPublisher : IMyContract
{
    static GenericEventHandler m_Event1          = delegate();
    static GenericEventHandler<int> m_Event2      = delegate();
    static GenericEventHandler<int,string> m_Event3 = delegate();

    public void Subscribe(EventType mask)
    {
        IMyEvents subscriber = OperationContext.Current.
                                GetCallbackChannel<IMyEvents>();

        if((mask & EventType.Event1) == EventType.Event1)
        {
            m_Event1 += subscriber.OnEvent1;
        }
        if((mask & EventType.Event2) == EventType.Event2)
        {
            m_Event2 += subscriber.OnEvent2;
        }
        if((mask & EventType.Event3) == EventType.Event3)
        {
            m_Event3 += subscriber.OnEvent3;
        }
    }
    public void Unsubscribe(EventType mask)
    {
        // 与 Subscribe() 相似但是使用 -=
    }
    public static void FireEvent(EventType eventType)
    {
        switch(eventType)
        {
            case EventType.Event1:
            {
                m_Event1();
                return;
            }
            case EventType.Event2:
            {
                m_Event2(42);
                return;
            }
            case EventType.Event3:
            {
                m_Event3(42,"Hello");
                return;
            }
            default:
            {
                throw new InvalidOperationException("Unknown event type");
            }
        }
    }
}
```



```

    public void DoSomething()
    { ... }
}

```

服务契约 `IMyContract` 定义了 `Subscribe()` 和 `Unsubscribe()` 方法。这些方法接受一个枚举类型 `EventType`，枚举类型的字段均被设置为2的倍数。这就使得订阅客户端能够通过枚举值的掩码，标识事件类型是属于订阅还是取消订阅。例如，订阅 `Event1` 和 `Event3`，但不订阅 `Event2`，则订阅者的调用如下所示：

```

class MySubscriber : IMyEvents
{
    void OnEvent1()
    { ... }
    void OnEvent2(int number)
    { ... }
    void OnEvent2(int number, string text)
    { ... }
}
IMyEvents subscriber = new MySubscriber();
InstanceContext context = new InstanceContext(subscriber);
MyContractClient proxy = new MyContractClient(context);
proxy.Subscribe(EventType.Event1 | EventType.Event3);

```

`MyPublisher` 内部维持了三个静态委托，每一个委托对应一个事件类型。所有委托均属于泛型委托类型 `GenericDelegate`：

```

public delegate void GenericEventHandler();
public delegate void GenericEventHandler<T>(T t);
public delegate void GenericEventHandler<T,U>(T t,U u);
public delegate void GenericEventHandler<T,U,V>(T t,U u,V v);
public delegate void GenericEventHandler<T,U,V,W>(T t,U u,V v,W w);
public delegate void GenericEventHandler<T,U,V,W,X>(T t,U u,V v,W w,X x);
public delegate void GenericEventHandler<T,U,V,W,X,Y>(T t,U u,V v,W w,X x,Y y);

```

从字面上讲，`GenericDelegate` 允许开发者表示所有事件处理的方法签名。

---

**注意：**要了解更多关于委托与 `GenericDelegate` 的知识，可以参见《.NET 组件编程》(O'Reilly) 第6章的内容。

---

`Subscribe()` 与 `Unsubscribe()` 方法都检查了传入参数 `EventType` 值，从对应的委托中添加或移除订阅者的回调。为了触发事件，`MyPublisher` 提供了静态方法 `FireEvent()`。它根据 `FireEvent()` 值判断应该触发哪一个事件，然后调用对应的委托。

事实上，通过使用委托，`MyPublisher` 服务完全简化了查找事件的实现细节。服务还可以使用一个链表，尽管代码会因此变得更加复杂。

---

注意：附录B展现了一个发布-订阅 (publish-subscriber) 框架，它能够为事件提供更好的设计支持。

---

## 流操作 (译注9)

在默认情况下，当客户端与服务交换消息时，这些消息会放入接收端的缓存中，一旦接收到了完整的消息，就立即会被传递。无论是客户端发送消息到服务，还是服务返回消息给客户端，都是如此。当客户端调用服务时，只要接收到了完整的消息，服务就会被调用。当包含了调用结果的返回消息被客户端完整接收时，才会解除对客户端的阻塞。对于数据量小的消息，这种交换模式提供了简单的编程模型，因为接收消息的耗时较之处理消息本身而言，是微不足道的。然而，一旦需要处理数据量更大的消息，例如包含了多媒体内容、大文件或数据块，如果每次都要等到完整地接收到消息之后才能解除阻塞，则未免不太现实。为了解决这样的问题，WCF允许接收端（可以是客户端，也可以是服务端）通过通道接收消息的同时，启动对消息数据的处理。这样的处理过程被称为流传输模型。对于具有大量负载的消息而言，流操作改善了系统的吞吐量与响应速度，因为在发送或接收消息的同时，不管是发送端还是接收端都不会被阻塞。

## I/O 流

若要处理消息流，WCF需要使用.NET的Stream类。事实上，契约操作对流的使用看起来与传统的I/O方法并无区别。在.NET中Stream类是所有I/O流的基类（例如FileStream、NetworkStream、MemoryStream类），它允许我们将I/O资源中的内容转换为流。我们需要做的就是返回或接收一个Stream类型的操作参数，如例5-17所示。

例 5-17：流操作

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    Stream StreamReply1();

    [OperationContract]
    void StreamReply2(out Stream stream);

    [OperationContract]
    void StreamRequest(Stream stream);
}
```

---

译注9：流操作主要针对大容量的消息对象。

```
[OperationContract(IsOneWay = true)]  
void OneWayStream(Stream stream);  
}
```

注意,我们只能将抽象类Stream类型,或者特定的可序列化的子类,例如MemoryStream作为操作的参数。诸如FileStream这样的子类都不支持序列化,因而我们使用基类Stream实为无奈之举。

WCF允许服务使用单向流操作(One-Way Streaming)将请求、应答、或者请求与应答消息转换为流。

要将应答消息转换为流,只需要操作返回Stream对象即可:

```
[OperationContract]  
Stream GetStream1();
```

或者将Stream对象作为输出参数:

```
[OperationContract]  
void GetStream2(out Stream stream);
```

要将请求消息转换为流,需要将Stream对象定义为方法参数:

```
[OperationContract]  
void SetStream1(Stream stream);
```

最后,我们甚至可以将单向操作中的请求消息转换为流:

```
// 单向流操作  
[OperationContract(IsOneWay = true)]  
void SetStream2(Stream stream);
```

## 流操作与绑定

只有BasicHttpBinding、NetTcpBinding和NetNamedPipeBinding支持流操作。但是在默认情况下这些绑定是禁止流操作的,即使使用Stream对象,绑定仍然会将消息整体放入到缓存中。我们需要根据所需的流模式,通过设置Boolean类型的属性TransferMode启用流操作。例如,使用BasicHttpBinding时:

```
public enum TransferMode  
{  
    Buffered, // 默认值  
    Streamed,  
    StreamedRequest,  
    StreamedResponse  
}  
public class BasicHttpBinding : Binding, ...  
{
```

```

    public TransferMode TransferMode
    {get;set;}
    // 更多成员
}

```

TransferMode.Streamed支持所有的流模式，也是唯一能够支持例5-17定义的所有操作的传输模式。但是，如果契约只包含了特定的流类型，例如转换为流的应答消息：

```

[ServiceContract]
interface IMyContract
{
    // 转换为流的应答消息
    [OperationContract]
    Stream GetStream1();

    [OperationContract]
    int MyMethod();
}

```

我们可以包含一个设置为 Buffered 的请求消息，也可以通过选择 TransferMode.StreamedResponse 将应答消息转换为流。

在客户端或者服务端（或者同时在两端），我们需要为每个所需的流模式配置绑定：

```

<configuration>
  <system.serviceModel>
    <client>
      <endpoint
        binding = "basicHttpBinding"
        bindingConfiguration = "StreamedHTTP"
        ...
      />
    </client>
    <bindings>
      <basicHttpBinding>
        <binding name = "StreamedHTTP" transferMode = "Streamed">
        </binding>
      </basicHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>

```

## 流操作与传输

重要的是，我们必须认识到 WCF 的流操作只不过是一种良好的编程模型。从根本上讲，传输自身（例如 HTTP）并不支持流操作，它默认的最大消息长度为 64KB。这样的数据可能存在问题。此时，开发者可能希望使用流操作，因为转换为流的消息往往会非常大（这正是选择使用流操作的目的）。如果开发者发现需要增加接收端消息的最大长

度，以适应大数据量的消息，则可以设置 `MaxReceivedMessageSize` 属性，以获得期望的最大消息长度：

```
public class BasicHttpBinding : Binding,...
{
    public long MaxReceivedMessageSize
    {get;set;}
    // 更多成员
}
```

通常，我们应避免使用编程方式，而是将一段配置信息放到配置文件中，因为消息大小的改变通常会发生在部署阶段：

```
<bindings>
  <basicHttpBinding>
    <binding name = "StreamedHTTP" transferMode = "Streamed"
      maxReceivedMessageSize = "120000">
    </binding>
  </basicHttpBinding>
</bindings>
```

## 流操作的管理

当客户端将请求消息流传递给服务时，服务会在客户端停止调用之后读取流的长度。客户端无法知道服务何时使用流进行操作。这就导致客户端无法关闭流。一旦服务执行完流操作，WCF 就应该自动关闭客户端的流。

当客户端通过应答消息流进行交互时，存在相似的问题。流是在服务端产生的，但服务端却不知道客户端何时使用流进行操作。WCF 没有提供解决办法，因为它根本无法获知客户端使用流执行了什么操作。因此，总是客户端负责关闭应答消息流。

---

**注意：**当我们使用流操作时，不能使用消息层的传输安全。关于安全的更多知识可以参见本书第 10 章的内容。在使用 TCP 绑定执行流操作时，我们同样不能启用可靠消息传输。

---

对于流消息而言，还有一些额外的要求：首先，开发者需要同步访问流的内容。例如，通过只读方式打开文件流，以允许支持其他参与方访问文件；如果需要，也可以采用排他方式打开流，以防止其他参与方访问它。此外，我们不能对会话服务使用流操作。流操作可能会持续相当长一段时间，而会话则不同，它意味着一个锁步（Lock-Step）执行，同时具有一个良好设计的会话分界。



## 第 6 章

# 错误

不管是哪一种服务操作，在任意时刻都可能遭遇一些不可预期的错误。问题在于如何将错误报告给客户端。异常与异常处理机制是与特定的技术紧密结合的，不能够跨越服务边界。此外，错误处理通常都属于本地的实现细节，不会影响到客户端。这样设计的原因在于客户端不需要关心错误的细节（以及发生错误的事实），但最主要的还是因为设计良好的应用程序中，服务是被封装的，因此客户端无法知道有关错误的信息。设计良好的服务应尽可能自治的，不能依赖于客户端去处理或恢复错误。任何非空的错误通知都应该是客户端与服务之间契约交互的一部份。本章介绍了服务与客户端如何处理这些声明的错误，以及如何扩展与改善这样一种基础的实现机制。

## 错误与异常

在传统的 .NET 编程中，任何未经处理的异常都会立刻终止抛出异常的进程。但 WCF 却与之大相径庭。如果代表某个客户端的服务调用导致异常，并不会结束宿主进程，其他客户端仍然可以访问该服务，托管在相同进程中的其他服务也不会受到影响。因此，当一个未经处理的异常离开服务的范围时，分发器会捕获它，并将它序列化到返回消息中传递给客户端。当返回消息到达代理时，代理会在客户端抛出一个异常。

当客户端试图调用服务时，实际上可能会遭遇三种错误类型。第一种错误类型为通信错误，例如网络故障、地址错误、宿主进程没有运行等。客户端的通信错误表现为 `CommunicationException` 异常。

客户端可能遇到的第二种错误类型与代理和通道的状态有关，例如试图访问已经关闭的代理，就会导致 `ObjectDisposedException` 异常。或者契约与绑定的安全保护级别不匹配，也会出现错误。



第三种错误类型源于服务调用。这种错误既可能是服务抛出的异常,也可能是服务在调用其他对象或资源时,通过内部调用抛出的异常。这些错误正是本章所要讲述的主题。

出于封装与解耦的目的,在默认情况下,所有服务端抛出的异常总是以 `FaultException` 类型到达客户端:

```
public class FaultException : CommunicationException  
{...}
```

如果要解耦客户端与服务,就应该对所有的服务异常一视同仁。客户端知道服务端发生的内容越少,则两者之间关系的解耦才越彻底。

## 异常与实例管理

当服务实例出现异常时, `WCF` 并不会关闭宿主进程,但错误可能会影响服务实例,同时还会影响到客户端继续使用代理(事实上是通道)访问服务的能力。准确的说,异常对于客户端与服务实例的影响与服务的实例模式有关。

### 单调服务与异常

如果调用引发异常,那么紧跟在异常之后,服务实例会被释放,代理将在客户端抛出 `FaultException` 异常。在默认情况下,所有服务抛出的异常(包括 `FaultException` 的派生类)会使得通道发生错误。即使客户端捕获了异常,它也不能发出随后的调用,因为它们会引发一个 `CommunicationObjectFaultedException` 异常。此时,客户端只能关闭代理。

### 会话服务与异常

无论使用何种 `WCF` 会话绑定,在默认情况下,所有异常(包括 `FaultException` 的派生类)都会终止会话。`WCF` 将会释放实例,而客户端则获得一个 `FaultException` 异常。即使客户端捕获了该异常,也不能继续使用代理,因为随后的调用会引发一个 `CommunicationObjectFaultedException` 异常。客户端唯一可以安全执行的就是关闭代理,因为一旦参与会话的服务实例遇到了错误,会话就不能再使用了。

### 单例服务与异常

当我们调用单例服务时,如果遇到异常,单例实例并不会终止,而是继续运行。在默认情况下,所有异常(包括 `FaultException` 的派生类)都会导致通道发生错误,客户端无法发出随后的调用,只能关闭代理。如果客户端包含了一个单例实例的会话,那么会话会终止。

## 错误

异常的根本问题在于它们与特定的技术紧密结合,因此无法跨越服务边界被调用两端共享。若要考虑良好的互操作性,我们就需要将基于特定技术的异常映射为某种与平台无关的错误信息。这种表现形式就是所谓的 SOAP 错误 (SOAP Fault)。SOAP 错误基于一种行业标准,它不依赖于任何一种诸如 CLR 异常、Java 异常或 C++ 异常之类的特定技术的异常。若要为了抛出一个 SOAP 错误 (或者简称错误),服务就不能抛出一个传统的 CLR 异常,而是抛出一个 `FaultException<T>` 类的实例,它的定义如例 6-1 所示:

例 6-1: `FaultException<T>` 类

```
[Serializable] //更多特性
public class FaultException : CommunicationException
{
    public FaultException();
    public FaultException(string reason);
    public FaultException(FaultReason reason);
    public virtual MessageFault CreateMessageFault();
    //更多成员
}

[Serializable]
public class FaultException<T> : FaultException
{
    public FaultException(T detail);
    public FaultException(T detail,string reason);
    public FaultException(T detail,FaultReason reason);
    //更多成员
}
```

`FaultException<T>` 是 `FaultException` 的特化,因此任何针对 `FaultException` 异常进行编程的客户端都能够处理 `FaultException<T>` 类型。

`FaultException<T>` 的类型参数 `T` 负责传递错误细节 (Error Detail)。没有要求错误细节必须为 `Exception` 的派生类,它可以是任何类型。唯一的约束是该类型必须支持序列化,或者必须是数据契约。

例 6-2 演示了一个简单的计算器服务,在实现 `Divide()` 方法时,如果除数为 0,则抛出一个 `FaultException<DivideByZeroException>` 异常。

例 6-2: 抛出 `FaultException<T>` 异常

```
[ServiceContract]
interface ICalculator
{
    [OperationContract]
    double Divide(double number1,double number2);
    //更多方法
}
```

```
class Calculator : ICalculator
{
    public double Divide(double number1, double number2)
    {
        if (number2 == 0)
        {
            DivideByZeroException exception = new DivideByZeroException();
            throw new FaultException<DivideByZeroException>(exception);
        }
        return number1 / number2;
    }
    // 其余的实现
}
```

除了 `FaultException<DivideByZeroException>` 异常, 服务还能够抛出参数不是 `Exception` 派生类类型的异常:

```
throw new FaultException<double>();
```

但是, 将 `Exception` 派生类作为错误细节类型更加符合传统的 .NET 编程实践, 代码也具有更强的可读性。此外, 它允许实现后面将要讨论的异常提升 (`Exception Promotion`) 功能。

传递给 `FaultException<T>` 构造函数的 `reason` 参数作为异常消息, 因此我们可以传递纯粹的字符串作为 `reason` 参数的值:

```
DivideByZeroException exception = new DivideByZeroException();
throw new FaultException<DivideByZeroException>(exception, "number2 is 0");
```

如果要求本地化, 更有效的方式是传递 `FaultReason` 对象。

## 错误契约

在默认情况下, 服务抛出的异常均以 `FaultException` 类型传递到客户端。原因在于任何服务希望与客户端共享的基于通信错误之上的任何异常, 都必须属于服务契约行为的一部分。为此, WCF 提供了错误契约, 通过它列出服务能够抛出的错误类型。这些错误类型应该与 `FaultException<T>` 使用的类型参数的类型相同。只要它们在错误契约中列出, WCF 客户端就能够分辨契约错误与其他错误之间的区别。

服务可以使用 `FaultContractAttribute` 特性定义它的错误契约:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true, Inherited = false)]
public sealed class FaultContractAttribute : Attribute
{
    public FaultContractAttribute(Type detailType);
    // 更多成员
}
```

我们可以将 `FaultContract` 特性直接应用到契约操作上，指定错误细节类型，如例 6-3 所示。

### 例 6-3：定义错误契约

```
[ServiceContract]
interface ICalculator
{
    [OperationContract]
    double Add(double number1, double number2);

    [OperationContract]
    [FaultContract(typeof(DivideByZeroException))]
    double Divide(double number1, double number2);
    // 更多方法
}
```

`FaultContract` 特性只对标记了它的方法有效。只有这样的方法才能抛出错误，并将它传递给客户端。此外，如果操作抛出的异常没有包含在契约中，则以普通的 `FaultException` 形式传递给客户端。为了传递异常，服务必须抛出与错误契约所列完全相同的细节类型。例如，若要满足如下的错误契约定义：

```
[FaultContract(typeof(DivideByZeroException))]
```

服务必须抛出 `FaultException<DivideByZeroException>` 异常。服务甚至不能抛出错误契约的细节类型的子类，因为它要求异常要满足契约的定义：

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [FaultContract(typeof(Exception))]
    void MyMethod();
}

class MyService : IMyContract
{
    public void MyMethod()
    {
        // 不满足契约的要求
        throw new FaultException<DivideByZeroException>(new DivideByZeroException());
    }
}
```

`FaultContract` 特性支持重复配置，可以在单个操作中列出多个错误契约：

```
[ServiceContract]
interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(InvalidOperationException))]
```

```
[FaultContract(typeof(string))]  
double Add(double number1,double number2);  
  
[OperationContract]  
[FaultContract(typeof(DivideByZeroException))]  
double Divide(double number1,double number2);  
// 更多方法  
}
```

如上的代码允许服务抛出契约定义中的任何一种异常，并将它们传递给客户端。

---

警告：我们不能为单向操作提供错误契约，因为从理论上讲，单向操作是没有返回值的：

```
// 无效定义  
[ServiceContract]  
interface IMyContract  
{  
    [OperationContract(IsOneWay = true)]  
    [FaultContract(...)]  
    void MyMethod();  
}
```

如果这样做，就会在装载服务时引发 `InvalidOperationException` 异常。

---

## 错误处理

错误契约与其他服务元数据一同发布。当 WCF 客户端导入该元数据时，契约定义包含了错误契约，以及错误细节类型的定义。错误细节类型的定义包含了相关的数据契约。如果细节类型是某个包含了各种专门字段的定制异常，那么错误细节类型对数据契约的支持就显得格外重要了。

客户端期望能够捕获和处理导入的错误类型。例如，在针对例 6-3 所示的契约编写客户端时，客户端能够捕获 `FaultException<DivideByZeroException>` 异常：

```
CalculatorClient proxy = new CalculatorClient();  
try  
{  
    proxy.Divide(2,0);  
    proxy.Close();  
}  
  
catch(FaultException<DivideByZeroException> exception)  
{...}  
  
catch(CommunicationException exception)  
{...}
```

注意，客户端仍然可能引发通信异常。

客户端可以采用处理 `FaultException` 基类异常的方式, 统一地处理所有与通信无关的服务端异常:

```
CalculatorClient proxy = new CalculatorClient();
try
{
    proxy.Divide(2,0);
    proxy.Close();
}

catch(FaultException exception)
{...}

catch(CommunicationException exception)
{...}
```

---

注意: 当客户端的开发者通过在客户端移除错误契约, 手动修改导入契约的定义时, 情况就变得复杂了。此时, 如果服务抛出的异常是在服务端错误契约的列表之中, 该异常在客户端会被表示为 `FaultException`, 而不是契约错误。

---

当服务抛出的异常属于服务端错误契约中列举的异常时, 异常不会导致通信通道出现错误。客户端能够捕获该异常, 继续使用代理, 或者安全地关闭代理。

## 未知错误

`FaultException<T>` 类继承自 `FaultException` 类。服务 (或者服务使用的所有下游对象) 可以直接抛出 `FaultException` 实例:

```
throw new FaultException("Some Reason");
```

`FaultException` 是一种特殊的异常类型, 称之为未知错误 (Unknown Fault)。一个未知错误以 `FaultException` 类型传递到客户端。它不会使通信通道发生错误, 因此客户端能够继续使用代理, 就好像该异常属于错误契约中的一部分那样。

---

注意: 服务抛出的 `FaultException<T>` 异常总是以 `FaultException<T>` 或者 `FaultExcetion` 类型到达客户端。如果没有错误契约 (或者 `T` 没有包含在契约中), 则服务抛出的 `FaultExcetion` 和 `FaultException<T>` 异常则以 `FaultException` 类型到达客户端。

---

客户端异常对象的 `Message` 属性可以被设置为 `FaultException` 的 `reason` 构造参数。`FaultException` 对象主要被服务的下游对象使用, 这些对象并不知道它们正在调用的服务所使用的错误契约。为避免这些下游对象与顶层服务之间的耦合, 同时又不希望通



道发生错误,就应该抛出 `FaultException` 异常。如果这些下游对象希望客户端能够处理独立于其他通信错误的异常,同样应该抛出 `FaultException` 异常。

## 调试错误

如果服务已经部署,那么最佳方案就是解除该服务与调用它的客户端之间的耦合,在服务的错误契约中,声明最少的异常类型,提供最少的错误信息。但如果是在测试与调试期间,用途更大的却是在返回给客户端的信息中包含所有的异常。它可以使得开发者使用一个测试客户端与调试器分析错误源,而不必处理完全封装的不透明的 `FaultException`。为此,我们应使用 `ExceptionDetail` 类,它的定义如下:

```
[DataContract]
public class ExceptionDetail
{
    public ExceptionDetail(Exception exception);

    [DataMember]
    public string HelpLink
    {get;private set;}

    [DataMember]
    public ExceptionDetail InnerException
    {get;private set;}

    [DataMember]
    public string Message
    {get;private set;}

    [DataMember]
    public string StackTrace
    {get;private set;}

    [DataMember]
    public string Type
    {get;private set;}
}
```

我们需要创建一个 `ExceptionDetail` 实例,然后通过要传递给客户端的异常对它进行初始化。接着,我们将 `ExceptionDetail` 的实例作为构造参数,同时提供一个最初的异常消息作为错误原因,抛出一个 `FaultException<ExceptionDetail>` 异常对象,而不能抛出不规则的异常。这一过程如例 6-4 所示。

### 例 6-4: 在错误消息中包含服务异常

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
```



```
void MethodWithError();  
}  
class MyService : IMyContract  
{  
    public void MethodWithError()  
    {  
        InvalidOperationException exception =  
            new InvalidOperationException("Some error");  
        ExceptionDetail detail = new ExceptionDetail(exception);  
        throw new FaultException<ExceptionDetail>(detail, exception.Message);  
    }  
}
```

如此做法可以使客户端能够发现最初的异常类型和消息。客户端的错误对象定义了 `Detail.Type` 属性，它包含了最初的服务异常名，而 `Message` 属性则包含了最初的异常消息。例 6-5 演示的客户端代码对例 6-4 抛出的异常进行了处理。

#### 例 6-5：处理包含的异常

```
MyContractClient proxy = new MyContractClient(endpointName);  
try  
{  
    proxy.MethodWithError();  
}  
catch(FaultException<ExceptionDetail> exception)  
{  
    Debug.Assert(exception.Detail.Type ==  
        typeof(InvalidOperationException).ToString());  
    Debug.Assert(exception.Message == "Some error");  
}
```

#### 以声明方式包含异常

`ServiceBehavior` 特性定义了 `Boolean` 类型的属性 `IncludeExceptionDetailInFaults`，如下所示：

```
[AttributeUsage(AttributeTargets.Class)]  
public sealed class ServiceBehaviorAttribute : Attribute, ...  
{  
    public bool IncludeExceptionDetailInFaults  
    {get;set;}  
    // 更多成员  
}
```

`IncludeExceptionDetailInFaults` 属性的默认值为 `false`。如下的代码段将它的值设置为 `true`：

```
[ServiceBehavior(IncludeExceptionDetailInFaults = true)]  
class MyService : IMyContract  
{...}
```

它的功能与例 6-4 相同，但它却能够自动包含异常：为了客户端程序能够处理它们，服务或服务的下游对象抛出的所有非契约型错误与异常，都将传递给客户端，在返回的错误消息中包含这些异常，正如例 6-5 所演示的那样：

```
[ServiceBehavior(IncludeExceptionDetailInFaults = true)]
class MyService : IMyContract
{
    public void MethodWithError()
    {
        throw new InvalidOperationException("Some error");
    }
}
```

服务（或服务的下游对象）抛出的任何错误，只要是在错误契约中列出的，都不会受到影响，并被传递给客户端。

包含所有的异常有利于调试，但必须避免发布和部署 `IncludeExceptionDetailInFaults` 属性值为 `true` 的服务。若要自动实现这一步骤，以避免潜在的缺陷，可以使用条件编译，如例 6-6 所示。

例 6-6：调试状态（译注 1）下设置 `IncludeExceptionDetailInFaults` 的值为 `true`

```
public static class DebugHelper
{
    public const bool IncludeExceptionDetailInFaults =
#if DEBUG
        true;
#else
        false;
#endif
}

[ServiceBehavior(IncludeExceptionDetailInFaults =
    DebugHelper.IncludeExceptionDetailInFaults)]
class MyService : IMyContract
{...}
```

---

警告：当 `IncludeExceptionDetailInFaults` 属性值为 `true` 时，异常实际上会导致通道发生错误，因而客户端不能发出随后的调用。

---

## 宿主与异常诊断

显然，在错误消息中包含所有异常有助于调试，同时也可以用于分析已部署服务存在的问题。值得庆幸的是，WCF 允许我们选择编程方式或管理方式设置宿主配置文件，通过

---

译注 1：条件编译的最佳实践是使用 `Conditional` 特性。不过，由于 `Conditional` 特性只能应用在方法上，因而并不适用于例 6-6 的场景。在第 9 章中，作者也表明了使用条件编译的态度。

宿主将 `IncludeExceptionDetailInFaults` 的值设置为 `true`。如果以编程方式设置,就需要在打开宿主之前查找服务描述中的服务行为,然后设置 `IncludeExceptionDetailInFaults` 属性值:

```
ServiceHost host = new ServiceHost(typeof(MyService));

ServiceBehaviorAttribute debuggingBehavior =
    host.Description.Behaviors.Find<ServiceBehaviorAttribute>();

debuggingBehavior.IncludeExceptionDetailInFaults = true;

host.Open();
```

可以在 `ServiceHost<T>` 中封装这一过程,实现简化,如例 6-7 所示。

#### 例 6-7: `ServiceHost<T>` 与返回的未知异常

```
public class ServiceHost<T> : ServiceHost
{
    public bool IncludeExceptionDetailInFaults
    {
        set
        {
            if (State == CommunicationState.Opened)
            {
                throw new InvalidOperationException("Host is already opened");
            }
            ServiceBehaviorAttribute debuggingBehavior =
                Description.Behaviors.Find<ServiceBehaviorAttribute>();
            debuggingBehavior.IncludeExceptionDetailInFaults = value;
        }
        get
        {
            ServiceBehaviorAttribute debuggingBehavior =
                Description.Behaviors.Find<ServiceBehaviorAttribute>();
            return debuggingBehavior.IncludeExceptionDetailInFaults;
        }
    }
}
```

`ServiceHost<T>` 的用法简单、易读:

```
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.IncludeExceptionDetailInFaults = true;
host.Open();
```

若要通过管理方式应用这一行为,可以在宿主配置文件中添加定制行为节,然后在服务定义中引用它,如例 6-8 所示。

#### 例 6-8: 通过管理方式在错误消息中包含异常

```
<system.serviceModel>
  <services>
```

```
<service name = "MyService" behaviorConfiguration = "Debugging">
    ...
</service>
</services>
<behaviors>
    <serviceBehaviors>
        <behavior name = "Debugging">
            <serviceDebug includeExceptionDetailInFaults = "true"/>
        </behavior>
    </serviceBehaviors>
</behaviors>
</system.serviceModel>
```

管理方式配置的优势在于它能够绑定已部署服务的行为，而不会影响服务代码。

## 错误与回调

由于通信异常或者回调自身抛出了异常，到客户端的回调自然就会失败。与服务契约操作相似，回调契约操作同样可以定义错误契约，如例 6-9 所示。

### 例 6-9：包含错误契约的回调契约

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();
}
interface IMyContractCallback
{
    [OperationContract]
    [FaultContract(typeof(InvalidOperationException))]
    void OnCallBack();
}
```

---

注意：WCF 中的回调通常被配置为单向调用，因而无法定义自己的错误契约。

---

然而，不同于通常一般的服务调用，传递给服务的内容以及错误展现自身的方式，与以下内容相关：

- 调用回调的时间。则意味着，或者回调在它正在调用的客户端发出服务调用期间被调用，或者是在宿主端的某个参与方对回调执行带外 (Out-Of-Band) 调用 (译注 2)。

---

译注 2：所谓带外调用，就是不被获得回调对象的服务直接调用，而是被参与的其他方调用。此时，回调对象会被存放在集合中，以待今后的调用。

- 使用的绑定。
- 抛出的异常类型。

如果回调属于带外调用,也就是说,在服务操作期间它被除了服务之外的其他参与方调用,那么回调的执行方式则与通常的 WCF 操作调用相似。例 6-10 演示了回调契约的带外调用,其中,回调契约的定义请参见例 6-9。

例 6-10: 带外调用中的错误处理

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    static List<IMyContractCallback> m_Callbacks =
        new List<IMyContractCallback>();

    public void DoSomething()
    {
        IMyContractCallback callback =
            OperationContext.Current.GetCallbackChannel<IMyContractCallback>();

        if(m_Callbacks.Contains(callback) == false)
        {
            m_Callbacks.Add(callback);
        }
    }

    public static void CallClients()
    {
        Action<IMyContractCallback> invoke = delegate(IMyContractCallback callback)
        {
            try
            {
                callback.OnCallback();
            }
            catch(FaultException<InvalidOperationException> exception)
            {
                ...
            }
            catch(FaultException exception)
            {
                ...
            }
            catch(CommunicationException exception)
            {
                ...
            }
        };

        m_Callbacks.ForEach(invoke);
    }
}
```

正如例6-10所示,由于通过回调契约可以将错误传递到宿主端,因而能够处理回调错误契约。如果客户端回调抛出的异常属于回调错误契约列出的异常,或者回调抛出一个 `FaultException` 异常,那么异常并不会导致回调通道发生错误,我们能够捕获异常,继续使用回调通道。然而,如果其中一个异常不属于错误契约的一部分,那么在抛出该异常之后,服务调用应会避免使用回调通道。

如果在服务操作期间,服务直接调用回调,并且抛出的异常被定义在错误契约的列表中,或者客户端回调抛出了一个 `FaultException` 异常,则回调错误的表现与在带外调用中的表现相同执行方式就是带外调用:

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
class MyService : IMyContract
{
    public void DoSomething()
    {
        IMyContractCallback callback =
            OperationContext.Current.GetCallbackChannel<IMyContractCallback>();
        try
        {
            callback.OnCallBack();
        }
        catch(FaultException<int> exception)
        { ... }
    }
}
```

注意,服务必须被配置为重入,才能避免死锁,正如第 5 章所阐释的那样(译注 3)。因为回调操作定义了一个错误契约,同时又必须保证它不能为单向方法,因此需要定义为重入。

无论是带外调用还是服务回调,都是为预期的行为提供的。

当服务调用回调时,如果回调操作抛出的异常不在错误契约之列(或者不是 `FaultException`),情况就变得异常复杂了。

如果服务使用的绑定为 `TCP` 或 `IPC` 绑定,当回调抛出的异常不在契约之中时,即使服务捕获了异常,第一个调用服务的客户端仍然会立即收到一个 `CommunicationException` 异常。然后,服务会获得一个 `FaultException` 异常。服务能够捕获和处理该异常,但它却会导致通道出现错误,因此服务无法重用它:

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
class MyService : IMyContract
{
    public void DoSomething()
    {
        IMyContractCallback callback =
            OperationContext.Current.GetCallbackChannel<IMyContractCallback>();
        try
        {
            callback.OnCallBack();
        }
        catch(FaultException exception)
        { ... }
    }
}
```

译注 3: 参见第 5 章例 5-7, 第 197 页。

```
        {...}  
    }  
}
```

如果服务使用双向WS绑定,当回调抛出的异常不在契约之中时,即使服务捕获了异常,第一个调用服务的客户端仍然会立即收到一个CommunicationException异常。其间,服务会被阻塞,直到抛出超时异常才会解除:

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]  
class MyService : IMyContract  
{  
    public void DoSomething()  
    {  
        IMyContractCallback callback =  
            OperationContext.Current.GetCallbackChannel<IMyContractCallback>();  
        try  
        {  
            callback.OnCallBack();  
        }  
        catch(TimeoutException exception)  
        {...}  
    }  
}
```

服务不能重用回调通道。

---

注意: 前面描述的各种回调行为,存在着巨大的差异,这是WCF的一个设计瑕疵。但它并非一个缺陷,或许在未来的版本中能够得到修正。

---

## 回调的调试

回调能够使用例6-4所示的技术,手动地将异常包含在错误消息中。CallbackBehavior特性提供了Boolean类型的属性 IncludeExceptionDetailInFaults,用来在消息中包含所有非错误类型的契约异常 (FaultException 除外):

```
[AttributeUsage(AttributeTargets.Class)]  
public sealed class CallbackBehaviorAttribute : Attribute,...  
{  
    public bool IncludeExceptionDetailInFaults  
    {get;set;}  
    // 更多成员  
}
```

与服务相似,引入异常有助于调试:

```
[CallbackBehavior(IncludeExceptionDetailInFaults = true)]  
class MyClient : IMyContractCallback  
{
```



```
public void OnCallBack()  
{  
    ...  
    throw new InvalidOperationException();  
}  
}
```

同样，我们可以在客户端配置文件中以管理方式配置这一行为：

```
<client>  
  <endpoint ... behaviorConfiguration = "Debug"  
    ...  
  />  
</client>  
<behaviors>  
  <endpointBehaviors>  
    <behavior name = "Debug">  
      <callbackDebug includeExceptionDetailInFaults = "true"/>  
    </behavior>  
  </endpointBehaviors>  
</behaviors>
```

注意，`endpointBehaviors` 标签的使用会影响到客户端的回调终结点。

## 错误处理扩展

WCF 允许开发者定制默认的异常报告与异常传递，它甚至为定制日志提供了一个钩子 (hook)。每个通道分发器都支持这种可扩展性，虽然在大多数情况下，我们可能只是简单地利用分发器的可扩展性。

若要安装自己的错误处理扩展，需要提供实现了 `IErrorHandler` 接口的分发器。`IErrorHandler` 接口的定义如下：

```
public interface IErrorHandler  
{  
    bool HandleError(Exception error);  
    void ProvideFault(Exception error, MessageVersion version, ref Message fault);  
}
```

虽然说任意一个参与方都可以提供这样的实现，但通常是由服务自身或通过宿主提供接口的实现。实际上，我们能够提供多个错误处理扩展，并将其链接在一起。本节后面会介绍如何安装这样的扩展。

## 提供错误

服务或服务操作的调用链中的任何一个对象在抛出异常后，会立即调用扩展对象的 `ProvideFault()` 方法。在将控制权返回给客户端之前调用 `ProvideFault()` 方法。如

果存在会话，则在终止会话之前，WCF 会调用 `ProvideFault()` 方法。如有必要，在释放服务实例之前，WCF 仍然会调用 `ProvideFault()` 方法。当客户端仍然被阻塞以等待操作完成时，由于传入调用线程调用了 `ProvideFault()` 方法，因此我们应该避免在该方法中执行耗时过长的操作。

### 使用 `ProvideFault()` 方法

调用 `ProvideFault()` 方法时，并不需要考虑异常抛出的类型。抛出的异常可以是常规的 CLR 异常，也可以是错误或错误契约中的错误。`error` 参数是抛出的异常的一个引用。如果 `ProvideFault()` 什么也没做，则客户端会通过错误契约（如果存在的话）获得一个异常，同时，异常的类型也会被抛出，正如本章前面所述：

```
class MyErrorHandler : IErrorHandler
{
    public bool HandleError(Exception error)
    { ... }

    public void ProvideFault(Exception error, MessageVersion version,
                             ref Message fault)
    {
        // 什么都不作异常会照常抛出
    }
}
```

`ProvideFault()` 方法会检查 `error` 参数，然后将它返回给客户端，或者提供一个替换错误。这种替换行为甚至会影响错误契约中的异常。若要提供一个替换错误，需要调用 `FaultException` 异常的 `CreateMessageFault()` 方法创建一个替换的错误消息。如果提供了一个新的错误契约消息，则必须创建一个新的错误细节对象，而不能重用原来的错误引用。然后，我们再将创建好的错误消息提供给 `Message` 类的静态方法 `CreateMessage()`：

```
public abstract class Message
{
    public static Message CreateMessage(MessageVersion version,
                                         MessageFault fault, string action);
    // 更多成员
}
```

注意，我们需要为 `CreateMessage()` 方法提供它所使用的错误消息的动作（`action`）。例 6-11 演示了这样一个复杂的步骤。

#### 例 6-11：创建一个替换错误

```
class MyErrorHandler : IErrorHandler
{
    public bool HandleError(Exception error)
    { ... }
```

```
public void ProvideFault(Exception error, MessageVersion version,
                        ref Message fault)
{
    FaultException<int> faultException = new FaultException<int>(3);
    MessageFault messageFault = faultException.CreateMessageFault();
    fault = Message.CreateMessage(version, messageFault, faultException.Action);
}
}
```

在例 6-11 中, ProvideFault() 方法将值 3 传递给 FaultException<int> 类, 以此来创建一个对象模拟服务实际抛出的异常。

实现 ProvideFault() 方法时, 也可以将 fault 参数值设置为 null:

```
class MyErrorHandler : IErrorHandler
{
    public bool HandleError(Exception error)
    { ... }
    public void ProvideFault(Exception error, MessageVersion version,
                            ref Message fault)
    {
        fault = null; // 在契约中禁止任何错误
    }
}
```

这样做的结果是将所有异常当作为 FaultException 类型传递给客户端, 即使这些异常与错误契约有关。将错误设置为 null 能够有效地约束所有的错误契约出现在正确的位置。

## 异常提升

最大可能使用 ProvideFault() 的是在一种我称之为异常提升 (Exception Promotion, 译注 4) 的技巧中。服务可以使用下游对象, 这些对象也可以被各种服务调用。考虑到系统的松散耦合, 服务的下游对象不应该依赖于调用它们的服务的特定错误契约。出现错误时, 对象只需要抛出常规的 CLR 异常。服务要做的是使用错误处理扩展检查抛出的异常。如果异常属于 FaultException<T> 中的 T 类型, 同时 FaultException<T> 又属于错误契约操作的一部分, 那么服务就能够将该异常提升为完整的 FaultException<T> 类型。例如, 给定这样的服务契约:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [FaultContract(typeof(InvalidOperationException))]
    void MyMethod();
}
```

译注 4: 异常提升能够将与服务异常无关的异常提升为服务异常, 从而使得服务使用的对象可以不用考虑服务使用的异常类型, 实现两者之间的解耦。

如果下游对象抛出一个 `InvalidOperationException` 异常, `ProvideFault()` 方法就会将它提升为 `FaultException<InvalidOperationException>` 类型, 如例 6-12 所示。

#### 例 6-12: 异常提升

```
class MyErrorHandler : IErrorHandler
{
    public bool HandleError(Exception error)
    { ... }
    public void ProvideFault(Exception error, MessageVersion version,
                             ref Message fault)
    {
        if (error is InvalidOperationException)
        {
            FaultException<InvalidOperationException> faultException =
                new FaultException<InvalidOperationException>(
                    new InvalidOperationException(error.Message));
            MessageFault messageFault = faultException.CreateMessageFault();
            fault = Message.CreateMessage(version, messageFault, faultException.Action);
        }
    }
}
```

例 6-12 存在的问题是代码与特定的错误契约是强耦合的, 需要对所有服务执行大量重复乏味的工作, 才能实现异常的提升。而且, 不管错误契约发生任何改变, 都必须修改错误扩展。

使用我们编写的 `ErrorHandlerHelper` 静态类, 可以自动地实现异常的提升:

```
public static class ErrorHandlerHelper
{
    public static void PromoteException(Type serviceType,
                                        Exception error,
                                        MessageVersion version,
                                        ref Message fault);

    // 更多成员
}
```

`ErrorHandlerHelper.PromoteException()` 方法将服务类型作为参数, 然后使用反射技术检查该服务类型的所有接口和操作, 并为特定的操作查找错误契约。通过解析 `error` 对象, 可以获得引发错误的操作。如果在该操作的错误契约中, 具有任何一个错误细节类型与异常类型匹配, `PromoteException()` 就会将 CLR 异常提升为契约的错误类型。

使用 `ErrorHandlerHelper` 类, 只需要一到两行代码就能够实现例 6-12 的功能:

```
class MyErrorHandler : IErrorHandler
{
```

```
public bool HandleError(Exception error)
{...}
public void ProvideFault(Exception error, MessageVersion version,
                        ref Message fault)
{
    Type serviceType = ...;
    ErrorHandlerHelper.PromoteException(serviceType, error, version, ref fault);
}
}
```

PromoteException()方法的实现与WCF技术无关,因此书中没有给出它的实现代码。但本书附带的源代码却包含了它的完整实现(译注5)。该方法的实现使用了C#的一些高级编程技术,例如泛型、反射、字符串解析、匿名方法与迟绑定。

## 处理错误

IErrHandler接口的HandleError()方法定义如下:

```
bool HandleError(Exception error);
```

在控制权被返回给客户端后,WCF会调用HandleError()方法。HandleError()只能在服务端使用,它在任何情况下都不会影响到客户端。调用HandleError()方法的线程是一个单独的工作线程,而不是用来处理服务请求(同时调用ProvideFault()方法)的线程。在后台使用一个单独的线程,可以使得开发者执行复杂耗时的过程,例如在不影响客户端的情况下将日志记录到数据库中。

因为我们能够将多个错误处理扩展安装到一个列表中,故而WCF允许开发者控制是否允许使用列表中的错误处理扩展。如果HandleError()方法返回false,WCF会继续调用其余已安装的扩展对象的HandleError()方法。如果返回true,WCF会停止调用错误处理扩展。显然,大多数错误处理扩展对象都应该返回false(译注6)。

HandleError()方法的error参数就是原来抛出的异常。HandleError()方法主要用于日志记录与错误跟踪,如例6-13所示。

### 例 6-13: 通过 Logbook 服务记录错误日志

```
class MyErrorHandler : IErrHandler
{
    public bool HandleError(Exception error)
    {
        try
        {
```

---

译注5: ErrorHandlerHelper类的实现在本书附带源代码的ErrorHandlerAttribute实例中。

译注6: 只有这样,才能够遍历列表中的所有处理对象。

```
        LogbookServiceClient proxy = new LogbookServiceClient();
        proxy.Log(...);
        proxy.Close();
    }
    catch
    {
    }
    finally
    {
        return false;
    }
}
public void ProvideFault(Exception error, MessageVersion version,
                        ref Message fault)
{
    ...
}
```

## Logbook 服务

本书附带的源代码提供了一个独立的服务 LogbookService (译注7), 专门用于记录错误日志。它能够将错误日志记录到 SQL Server 数据库中。服务契约同时还提供了获取日志内容与清除日志的操作。源代码中还包含了一个功能简单的日志查看器与管理工具。LogbookService 服务除了具有记录错误日志的功能之外, 还允许开发者记录与异常无关的内容。Logbook 服务的框架架构如图 6-1 所示。

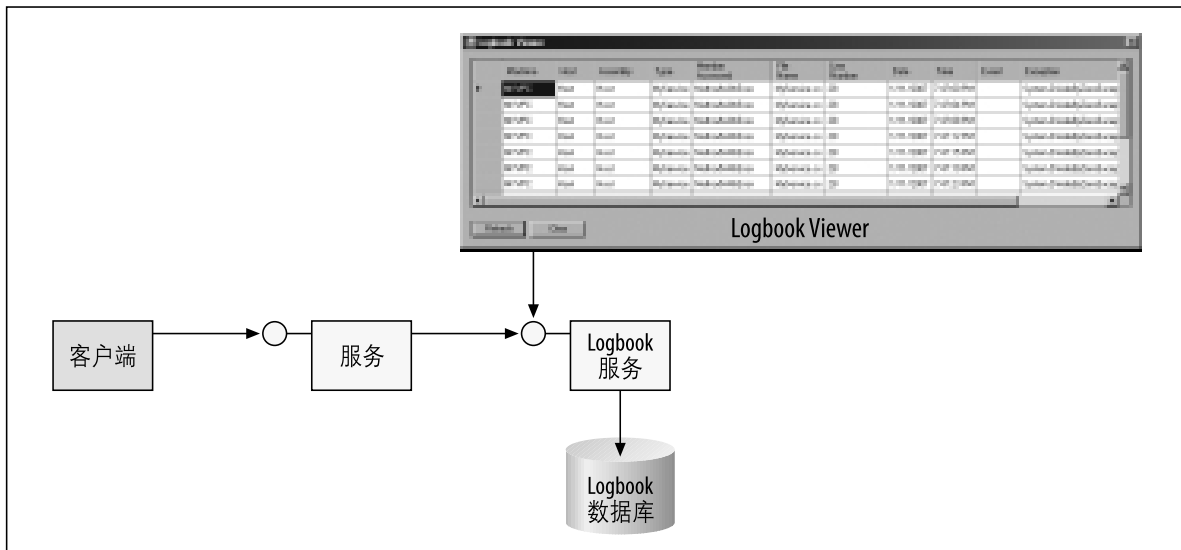


图 6-1: LogbookService 服务与日志查看器

使用编写的 ErrorHandlerHelper 静态类的 LogError() 方法, 可以自动调用 LogbookService 记录错误日志。

译注 7: LogbookService 服务在本书附带源代码的 ErrorHandlerAttribute 实例中。

```
public static class ErrorHandlerHelper
{
    public static void LogError(Exception error);
    // 更多成员
}
```

`error` 参数就是我们希望记录的异常对象。`LogError()` 方法封装了对 `LogbookService` 的调用。因此, 我们只需要编写如下几行代码就可以实现例 6-13 的功能:

```
class MyErrorHandler : IErrorHandler
{
    public bool HandleError(Exception error)
    {
        ErrorHandlerHelper.LogError(error);
        return false;
    }
    public void ProvideFault(Exception error, MessageVersion version,
                             ref Message fault)
    {...}
}
```

除了原来的异常信息, `LogError()` 方法还扩展了异常的解析功能, 以及其他记录错误与相关信息的环境变量。

特别的, `LogError()` 方法能够捕获如下信息:

- 异常产生的位置 (机器名和宿主进程名)。
- 异常产生的代码行 (程序集名、文件名和行号)。
- 抛出异常的类型以及被访问的成员。
- 异常的日期和时间。
- 异常名和异常消息。

`LogError()` 方法的实现与 WCF 无关, 因此本章没有给出它的实现。但是, 它的实现大量地运用了 .NET 编程技术, 例如字符串与异常解析、获取环境信息等。错误信息通过专门的数据契约传递给 `LogbookService` 服务。

## 安装错误处理扩展

WCF 的每个通道分发器都提供了一个错误扩展对象的集合:

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    public Collection<IErrorHandler> ErrorHandlers
    {get;}
    // 更多成员
}
```



若要安装自己定制的 `ErrorHandler` 实现，只需要将它添加到它所需的分发器（通常是全部）中即可。

我们必须在第一个调用到达服务之前，宿主创建分发器集合之后添加错误扩展。这一时机稍纵即逝，恰恰处于宿主被初始化却还没有被打开的一瞬间。要抓住这一时机，最好的解决方案就是将错误扩展看作是定制的服务行为，因为只有行为才能够在准确的时刻把握与分发器交互的时机。第 4 章曾经提及，所有的服务行为均实现了 `IServiceBehavior` 接口。`IServiceBehavior` 接口的定义如下：

```
public interface IServiceBehavior
{
    void AddBindingParameters(ServiceDescription description,
                              ServiceHostBase host,
                              Collection<ServiceEndpoint> endpoints,
                              BindingParameterCollection parameters);

    void ApplyDispatchBehavior(ServiceDescription description,
                               ServiceHostBase host);

    void Validate(ServiceDescription description, ServiceHostBase host);
}
```

`ApplyDispatchBehavior()` 方法提示我们可以将错误扩展添加到分发器中。我们完全可以忽略 `IServiceBehavior` 接口的其他所有方法，只为它们提供空的实现。

`ApplyDispatchBehavior()` 方法需要使用 `ServiceHostBase` 的 `ChannelDispatchers` 属性访问可用的分发器集合：

```
public class ChannelDispatcherCollection :
    SynchronizedCollection<ChannelDispatcherBase>
{
}
public abstract class ServiceHostBase : ...
{
    public ChannelDispatcherCollection ChannelDispatchers
    {get;}
    // 更多成员
}
```

`ChannelDispatchers` 中的每一项元素均为 `ChannelDispatcher` 类型。我们可以将 `ErrorHandler` 接口的实现添加到所有的分发器中，或者只将它添加到与特定绑定有关的分发器中。例 6-14 演示了如何将 `ErrorHandler` 实现添加到服务的所有分发器中。

例 6-14：添加一个错误扩展对象

```
class MyErrorHandler : ErrorHandler
{...}

class MyService : IMyContract, IServiceBehavior
{
```

[illegible]

```
{
    protected Type ServiceType
    {get;set;}
}
```

可以直接应用 ErrorHandlerBehavior 特性：

```
[ErrorHandlerBehavior]
class MyService : IMyContract
{...}
```

特性将自身作为错误处理扩展进行安装。它使用 ErrorHandlerHelper 类实现了自动将异常提升为所需错误契约的功能，同时调用 LogbookService 服务自动记录异常。例 6-16 列出了 ErrorHandlerBehavior 特性的实现代码。

#### 例 6-16: ErrorHandlerBehavior 特性

```
[AttributeUsage(AttributeTargets.Class)]
public class ErrorHandlerBehaviorAttribute : Attribute, IServiceBehavior,
                                           IErrorHandler
{
    protected Type ServiceType
    {get;set;}

    void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription description,
                                                ServiceHostBase host)
    {
        ServiceType = description.ServiceType;
        foreach(ChannelDispatcher dispatcher in host.ChannelDispatchers)
        {
            dispatcher.ErrorHandlers.Add(this);
        }
    }
    bool IErrorHandler.HandleError(Exception error)
    {
        ErrorHandlerHelper.LogError(error);
        return false;
    }
    void IErrorHandler.ProvideFault(Exception error, MessageVersion version,
                                    ref Message fault)
    {
        ErrorHandlerHelper.PromoteException(ServiceType, error, version, ref fault);
    }
    void IServiceBehavior.Validate(...)
    {}
    void IServiceBehavior.AddBindingParameters(...)
    {}
}
```

注意，例 6-16 中的 ApplyDispatchBehavior() 方法将服务类型保存在一个 protected 属性中。因为在 ProvideFault() 方法中，调用的 ErrorHandlerHelper.PromoteException() 方法需要服务类型。

## 宿主与错误扩展

`ErrorHandlerBehavior` 特性可以极大地简化安装错误扩展的过程, 服务开发者只需要应用该特性即可。如果宿主在添加错误扩展对象时, 可以不用考虑该错误扩展是否由服务提供, 那么这种实现方式无疑是很好的。但是, 由于安装扩展对象的时间非常短, 而宿主添加这样的扩展对象却需要执行多个步骤。首先, 需要提供一个支持 `IServiceBehavior` 和 `IErrorHandler` 接口的错误处理扩展类型。如前所示, `IServiceBehavior` 的实现会将错误扩展添加到分发器中。接下来, 需要定义一个宿主类, 继承 `ServiceHost` 并重写 `CommunicationObject` 基类定义的 `OnOpening()` 方法:

```
public abstract class CommunicationObject : ICommunicationObject
{
    protected virtual void OnOpening();
    // 更多成员
}
public abstract class ServiceHostBase : CommunicationObject ,...
{...}
public class ServiceHost : ServiceHostBase,...
{...}
```

在 `OnOpening()` 方法中, 我们需要将定制的错误处理类型添加到服务描述中的服务行为集合中。第 1 章和第 4 章曾经介绍过这一行为集合:

```
public class Collection<T> : IList<T>,...
{
    public void Add(T item);
    // 更多成员
}
public abstract class KeyedCollection<K,T> : Collection<T>
{...}
public class KeyedByTypeCollection<I> : KeyedCollection<Type,I>
{...}
public class ServiceDescription
{
    public KeyedByTypeCollection<IServiceBehavior> Behaviors
    {get;}
}
public abstract class ServiceHostBase : ...
{
    public ServiceDescription Description
    {get;}
    // 更多成员
}
```

步骤的执行顺序已被封装到 `ServiceHost<T>` 中, 并实现了执行的自动化:

```
public class ServiceHost<T> : ServiceHost
{
    public void AddErrorHandler(IErrorHandler errorHandler);
    public void AddErrorHandler();
}
```

```
        // 更多成员  
    }
```

`ServiceHost<T>` 定义了两个重载版本的 `AddErrorHandler()` 方法。接收参数 `errorHandler` 的方法建立了 `ErrorHandler` 对象与行为对象之间的内在关联, 因而方法参数类型只能是实现 `ErrorHandler` 接口的类, 而不是实现 `IServiceBehavior` 接口的类:

```
class MyService : IMyContract  
{...}  
  
class MyErrorHandler : ErrorHandler  
{...}  
  
ServiceHost<MyService> host = new ServiceHost<MyService>();  
host.AddErrorHandler(new MyErrorHandler());  
host.Open();
```

无参方法 `AddErrorHandler()` 使用 `ErrorHandlerHelper` 类安装一个错误处理扩展对象, 就好像服务类被标记了 `ErrorHandlerBehavior` 特性一样:

```
class MyService : IMyContract  
{...}  
  
ServiceHost<MyService> host = new ServiceHost<MyService>();  
host.AddErrorHandler();  
host.Open();
```

实际上, 在后一个例子中, `ServiceHost<T>` 的内部使用了一个 `ErrorHandlerBehaviorAttribute` 实例。

例 6-17 演示了 `AddErrorHandler()` 方法的实现。

例 6-17: 实现 `AddErrorHandler()` 方法 (译注 8)

```
public class ServiceHost<T> : ServiceHost  
{  
    class ErrorHandlerBehavior : IServiceBehavior, ErrorHandler  
    {  
        ErrorHandler m_ErrorHandler;
```

---

译注 8: `AddErrorHandler()` 的带参方法使用了一个嵌套类 `ErrorHandlerBehavior`; 而无参方法则使用了 `ErrorHandlerBehaviorAttribute` 特性。注意两者之间的区别, 因为前者是通过外部传入的 `ErrorHandler` 对象, 从而完成 `ErrorHandler` 接口方法的实现; 而后者则通过调用 `ErrorHandlerHelper` 的相关方法, 完成 `ErrorHandler` 接口方法的实现。因而, 前一个方法需要传递一个 `ErrorHandler` 对象, 而后者则不必。

```
public ErrorHandlerBehavior(ILogger errorHandler)
{
    m_ErrorHandler = errorHandler;
}

void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription description,
                                           ServiceHostBase host)
{
    foreach(ChannelDispatcher dispatcher in host.ChannelDispatchers)
    {
        dispatcher.ErrorHandlers.Add(this);
    }
}

bool ILogger.HandleError(Exception error)
{
    return m_ErrorHandler.HandleError(error);
}

void ILogger.ProvideFault(Exception error, MessageVersion version,
                           ref Message fault)
{
    m_ErrorHandler.ProvideFault(error, version, ref fault);
}

// 其余实现
}

List<IServiceBehavior> m_ErrorHandlers = new List<IServiceBehavior>();

public void AddErrorHandler(ILogger errorHandler)
{
    if(State == CommunicationState.Opened)
    {
        throw new InvalidOperationException("Host is already opened");
    }
    IServiceBehavior errorHandler = new ErrorHandlerBehavior(errorHandler);
    m_ErrorHandlers.Add(errorHandlerBehavior);
}

public void AddErrorHandler()
{
    if(State == CommunicationState.Opened)
    {
        throw new InvalidOperationException("Host is already opened");
    }
    IServiceBehavior errorHandler = new ErrorHandlerBehaviorAttribute();
    m_ErrorHandlers.Add(errorHandlerBehavior);
}

protected override void OnOpening()
{
    foreach(IServiceBehavior behavior in m_ErrorHandlers)
    {
        Description.Behaviors.Add(behavior);
    }
    base.OnOpening();
}

// 其余实现
}
```

为了避免强制要求提供的 `ErrorHandler` 引用对象同时支持 `IServiceBehavior` 接口, `ServiceHost<T>` 定义了一个私有嵌套类 `ErrorHandlerBehavior`。它同时实现了 `ErrorHandler` 和 `IServiceBehavior` 接口。要创建 `ErrorHandlerBehavior` 对象, 我们需要为它提供一个 `ErrorHandler` 的实现, 同时保存这一实现供以后使用。`IServiceBehavior` 的实现将实例自身添加到所有分发器的错误处理集合中。`ErrorHandler` 接口的实现只不过是将引用指向之前保存的构造参数 `ErrorHandler` 对象。`ServiceHost<T>` 定义了一个 `IServiceBehavior` 引用对象的链表 `m_ErrorHandlers`, 作为类的成员变量。`AddErrorHandler()` 方法接收一个 `ErrorHandler` 类型的参数, 用它来构建一个 `ErrorHandlerBehavior` 实例, 添加到 `m_ErrorHandlers` 中。无参方法 `AddErrorHandler()` 则创建了一个 `ErrorHandlerBehaviorAttribute` 实例, 因为该特性实际上就是一个同时支持 `ErrorHandler` 和 `IServiceBehavior` 的类。创建的特性实例也被添加到 `m_ErrorHandlers` 中。最后, `OnOpening()` 方法遍历整个 `m_ErrorHandlers` 链表对象, 将每个行为添加到行为集合中。

## 回调与错误扩展

客户端的回调对象同样能够为错误处理提供 `ErrorHandler` 的一个实现。与服务错误扩展相比, 主要的区别在于它安装回调扩展的方法, 需要使用 `IEndpointBehavior` 接口, 定义如下:

```
public interface IEndpointBehavior
{
    void AddBindingParameters(ServiceEndpoint serviceEndpoint,
                             BindingParameterCollection bindingParameters);
    void ApplyClientBehavior(ServiceEndpoint serviceEndpoint,
                             ClientRuntime behavior);
    void ApplyDispatchBehavior(ServiceEndpoint serviceEndpoint,
                               EndpointDispatcher endpointDispatcher);
    void Validate(ServiceEndpoint serviceEndpoint);
}
```

所有回调行为均支持 `IEndpointBehavior` 接口。接口中只有 `ApplyClientBehavior()` 方法与错误扩展的安装有关, 它允许开发者将错误扩展与回调的单个分发器关联起来。`behavior` 参数为 `ClientRuntime` 类型, 它定义了 `DispatchRuntime` 类型的属性 `CallbackDispatchRuntime`。`DispatchRuntime` 类则定义了 `ChannelDispatcher` 属性, 它包含了一个错误处理器集合:

```
public sealed class ClientRuntime
{
    public DispatchRuntime CallbackDispatchRuntime
    {get;}
}
```



```
// 更多成员
}
public sealed class DispatchRuntime
{
    public ChannelDispatcher ChannelDispatcher
    {get;}
    // 更多成员
}
```

作为服务端的错误处理扩展，我们需要将 `ErrorHandler` 的定制错误处理的实现添加到集合中。

回调对象自身能够实现 `IEndpointBehavior` 接口，如例 6-18 所示。

#### 例 6-18：实现 `IEndpointBehavior`

```
class MyErrorHandler : ErrorHandler
{...}

class MyClient : IMyContractCallback, IEndpointBehavior
{
    public void OnCallBack()
    {...}

    void IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint serviceEndpoint,
                                                ClientRuntime behavior)
    {
        ErrorHandler handler = new MyErrorHandler();

        behavior.CallbackDispatchRuntime.ChannelDispatcher.
            ErrorHandlers.Add(handler);
    }

    void IEndpointBehavior.AddBindingParameters(...)
    {}
    void IEndpointBehavior.ApplyDispatchBehavior(...)
    {}
    void IEndpointBehavior.Validate(...)
    {}
}
```

回调类自身可以直接实现 `ErrorHandler`，而不必通过外部类去实现 `ErrorHandler`：

```
class MyClient : IMyContractCallback, IEndpointBehavior, ErrorHandler
{
    public void OnCallBack()
    {...}

    void IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint serviceEndpoint,
                                                ClientRuntime behavior)
    {
        behavior.CallbackDispatchRuntime.ChannelDispatcher.ErrorHandlers.Add(this);
    }
}
```

```
public bool HandleError(Exception error)
{...}
public void ProvideFault(Exception error, MessageVersion version,
                        ref Message fault)
{...}
}
```

### 回调错误处理特性

如果要自动实现例6-18所示的代码,可以使用 `CallbackErrorHandlerBehaviorAttribute` 特性,它的定义如下:

```
public class CallbackErrorHandlerBehaviorAttribute : ErrorHandlerBehaviorAttribute,
                                                    IEndpointBehavior
{
    public CallbackErrorHandlerBehaviorAttribute(Type clientType);
}
```

`CallbackErrorHandlerBehavior` 特性继承了服务端的 `ErrorHandlerBehavior` 特性,同时还显式实现了 `IEndpointBehavior` 接口。该特性使用了 `ErrorHandlerHelper` 类提升异常类型,同时以日志方式记录异常。

此外,特性需要传递一个构造参数,它的类型就是应用该特性的回调的类型:

```
[CallbackErrorHandlerBehavior(typeof(MyClient))]
class MyClient : IMyContractCallback
{
    public void OnCallBack()
    {...}
}
```

类型参数是必须的,因为 `ErrorHandlerHelper.PromoteException()` 方法需要使用该类型,而特性却没有办法获取它。

`CallbackErrorHandlerBehaviorAttribute` 特性的实现如例 6-19 所示。

#### 例 6-19: 实现 `CallbackErrorHandlerBehavior` 特性

```
public class CallbackErrorHandlerBehaviorAttribute : ErrorHandlerBehaviorAttribute,
                                                    IEndpointBehavior
{
    public CallbackErrorHandlerBehaviorAttribute(Type clientType)
    {
        ServiceType = clientType;
    }
    void IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint serviceEndpoint,
                                                ClientRuntime behavior)
    {
        behavior.CallbackDispatchRuntime.ChannelDispatcher.ErrorHandlers.Add(this);
    }
}
```

```
void IEndpointBehavior.AddBindingParameters(...)  
{  
}  
void IEndpointBehavior.ApplyDispatchBehavior(...)  
{  
}  
void IEndpointBehavior.Validate(...)  
{  
}  
}
```

注意在例 6-19 中，提供的回调客户端类型被保存在 `protected` 属性 `ServiceType` 中，`ServiceType` 属性定义参见例 6-16。

事务是构建健壮的、高质量的面向服务应用程序的关键。WCF为服务开发人员提供了简单的、声明式的事务支持，使开发者可以配置如资源登记、投票之类的参数，这些参数的配置都是在服务范围之外完成的。此外，WCF允许客户端应用程序创建事务并跨越服务边界传播事务。本章首先介绍了事务试图解决的问题背景和基本的事务术语，接着探讨了WCF对事务的支持以及它所提供的事务管理，余下的部分则专注于客户端与服务间的事务性编程模型，以及事务如何关联WCF的其他领域，诸如实例管理和回调等。

## 恢复的挑战

错误处理与恢复机制是众多应用程序的“阿喀琉斯之踵”。一旦应用无法执行某个特定的操作，就应该设法将它从错误状态中恢复，并使系统（即参与交互的客户端与服务的集合）还原到一个满足一致性的状态，通常是系统在执行导致错误发生的操作之前的状态。可能失败的操作通常由多个潜在地并发执行的小步骤构成。这些步骤中有的可能失败，有的则可能执行成功。恢复面临的问题是我们必须为成功部分与失败部分的所有排列组合编写防御代码。例如，由10个小的并发步骤组成的一个操作大概有三百萬种恢复机制，因为对于恢复逻辑而言，操作失败的顺序同样存在影响，而10的阶乘差不多就是三百万。

试图采用手工方式为一个像样的应用程序编写恢复代码无异于“缘木求鱼”，结果只会造成脆弱的代码。这些代码很容易受到应用程序执行过程或业务用例改变的影响，从而损害系统的开发效率与性能。简单地将所有工作都放到手工编写的恢复逻辑中损害了开发效率，而性能的损害则是因为在每个操作后都需要执行大量的代码以验证一切正常。在现实中，开发人员倾向于只处理那些容易的恢复事例，也就是那些不仅能察觉到而且知道如何处理的事例。更复杂的错误状况，如中间网络的失效或磁盘损伤则是无法处理

的。此外，由于恢复就是将系统还原到一个一致状态（通常是操作前的状态），因而，关键问题在于那些成功执行的操作，而不是那些失败的。原因在于失败的操作无法影响系统。这里的挑战在于需要撤销成功的步骤，比如从表中删除一行，或从链表中删除一个节点，又或调用一个远程服务。牵涉到的场景可能非常复杂，而手工实现的恢复逻辑几乎必然会漏掉一些成功的子操作。

恢复逻辑越复杂，自身也就越容易出现错误。假如在恢复机制中存在一个错误，我们应该如何将它恢复？开发者如何对复杂的恢复逻辑进行设计、测试和调试？我们如何模拟这些无尽的错误和可能的失败？不仅如此，如果在操作失败前，也就是正在成功地执行一步步子操作时，其他参与方又访问了我们的应用程序，并作用于我们准备在恢复中回滚的系统状态，我们又该如何处理？此时，其余的参与方正作用于处在非一致状态的信息，而且，根据定义它们同样处于错误之中。此外，我们的操作也可能是另一个范围更大的跨越多个服务的操作中的一个步骤，这些服务可能分别由多个厂商提供，并部署在不同的机器上。在这种情况下我们应该如何将系统当成一个整体进行恢复？即便存在一种神奇的恢复方式可以将服务恢复，我们又该如何将这个恢复逻辑插入到跨服务的恢复中？

## 事务

使用事务是维持系统一致性并合理实现错误恢复的最好方式。事务往往是一个复杂操作的集合，这个操作集就如同一个原子操作（Atomic Operation）一样，其中任何一个单独操作的失败都会导致整个集合的失败。如图 7-1 所示，当事务正在进行时，允许系统处于短暂的不一致状态，一旦事务结束，则必须保证系统的一致状态，或者是新的一致状态（B），或者是事务开始前系统的原始一致状态（A）。

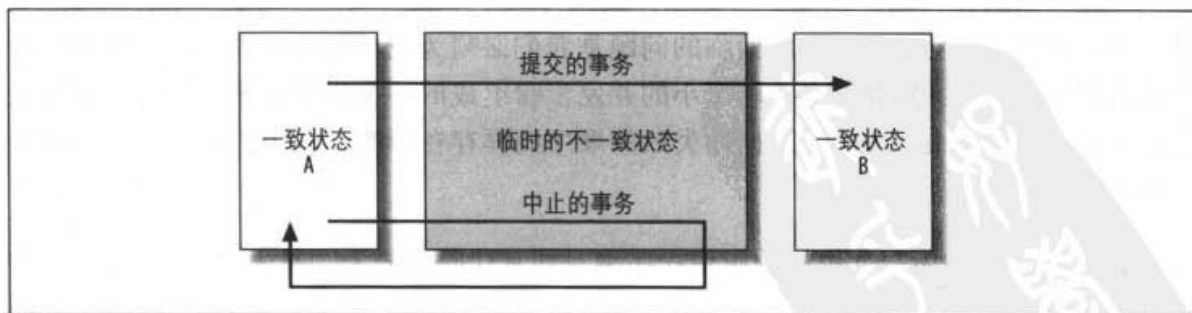


图 7-1：事务对系统在一致状态之间的迁移

如果事务执行顺利，并且成功地将系统由一致性状态 A 迁移到一致性状态 B，我们把它称为提交的事务（Committed Transaction）。假如事务在执行过程中遇到了错误，并且回滚了之前已经成功完成的中间步骤，我们则把它称作中止的事务（Aborted Transaction）。

假如事务既没能提交也没能中止，则将它称为悬疑的事务（In-Doubt Transaction）。悬疑的事务通常需要管理人员或用户协助解决，这已经超出了本书的范围。

## 事务型资源

事务型编程需要与诸如数据库或消息队列这类资源一起协作，它们能够参与事务并提交或回滚在事务中发生变化的内容。这类资源从诞生之初就以一种或多种形式存在数十年了。如果要沿袭传统，我们就必须通知这些我们希望执行事务性工作的资源。这一过程称为在事务中登记资源（Enlisting the Resource）。一些资源可能支持自动登记，也就是说，它们能检测自身是否被事务访问，并完成自动登记。一旦资源被登记，我们就可以对它执行操作。假如没有错误发生，则要求资源提交对它状态所做的修改；反之则要求资源回滚所有的变化。在执行事务期间，必须确保不会访问任何非事务型资源（如 Windows XP 上的文件系统），因为一旦事务被中止，针对那些资源所做的修改将无法回滚。

## 事务属性

在面向服务应用程序中使用事务时，必须遵守四个关键属性，也就是众所周知的 ACID：原子性（Atomic）、一致性（Consistent）、隔离性（Isolated）和持久性（Durable）。设计事务性服务时，我们必须符合 ACID 的要求，它们都是必备的。正如你将在本章看到的，WCF 对事务属性有极其严格的要求。

### 原子性

原子性（注 1）意味着当一个事务结束时，它对资源状态的所有改变都应该被视为一个不可分割的原子操作。改变资源状态时，就好似整个宇宙停止了运转，万事万物都失去了生机；而当我们做出修改后，宇宙的万物又重新活转过来。事务之外的实体如果只想观察事务涉及的发生部分改变的资源，而非全部改变的资源，则是不可能的（译注 1）。事务一经完成就不应该残留有任务在后台继续运行，因为这样的操作违反了事务的原子性。所有事务产生的操作都必须包含在该事务内。

---

注 1：“atom”一词来源于希腊语“atomos”，意思是不可分割。古代的希腊人认为如果对物质进行分割，并不断的分割下去，最终将得到一些不可分的部件，他们把这些部件称为“atomos”。当然，希腊人对原子的认知是错误的，原子还可以继续被分为诸如电子、质子和中子这些比原子更细的粒子。然而，事务却完全符合 atom 最初的语义，它是无法分割的。

译注 1：换言之，就是事务之外的实体只能观察到发生全部改变的资源。



事务具有的原子性属性使得我们更易于开发客户端应用程序。客户端无需管理部分失败的请求，也不必编写复杂的恢复逻辑。客户端清楚地知道事务将作为一个整体，只能有成功或者失败两种选择。失败时，客户端可以选择发出一个新的请求（开始一个新的事务），或做一些其他的事，例如向用户发出警报。重要的是客户端不必自己实现系统的恢复。

### 一致性

一致性意味着事务必须让系统保留一致的状态。要注意一致性与原子性的不同。就算所有的修改都能像一个原子操作一样被提交，事务还必须保证所有改变都有意义，即保持一致。通常情况下是由开发者来确保操作在语义上的一致性。事务所要的就是将系统从一个一致的状态迁移到另一个一致的状态。

### 隔离性

隔离性意味着其他实体（事务型或非事务型的）无法看到事务过程中资源的中间状态，因为这个中间状态可能是不一致的。事实上，就算它是一致的，事务仍然可以中止，而且之前的修改也可以被回滚。隔离性对系统总体的一致性具有决定性的作用。设想事务A允许事务B访问自己的中间状态。事务A中止了，而事务B决定提交，就会出现問題，因为事务B把它的执行建立在已经回滚的系统状态的基础之上，这样事务B就停留在一种未知的不一致状态下。管理隔离性并不是一件可有可无的事。参与事务的资源必须对数据加锁，以防止其他事务对数据的访问，还必须在提交或中止事务时解除对数据访问的锁定。

### 持久性

传统上，资源对事务的支持不仅意味着它属于事务相关的资源，而且还表示该资源具有持久性。因为应用程序在任何时候都有可能崩溃，至于正在使用的内存中的内容则可能会被擦除。如果对系统状态的改变处于内存改变阶段，它们就很可能丢失，而系统则可能处于一种不一致的状态。无论如何，持久性实际上包含了一个选择范围。资源应该如何适应这种灾难是一个公开的问题，它取决于数据的性质及其敏感性、预算、可用时间以及现有系统的管理者等等。如果持久性实际上意味着属于不同程度的持久性范围，那么我们同样可以考虑保存在内存中的易失性资源。易失型资源的优势在于，它们提供了比持久型资源更好的性能。更重要的，它们允许我们在使用支持错误恢复的事务同时，更加符合传统的编程模型。在本章后面我们将看到服务在何时以及如何从易失性资源管理器（Volatile Resource Manager, VRM）中获益。



## 事务管理

WCF服务可以直接作用于事务型资源,并通过显式地使用类似ADO.NET提供的编程模型来管理事务。在使用该模型时,我们需要显式地启动以及管理事务,如例7-1所示。

例7-1: 显式地管理事务

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}

class MyService : IMyContract
{
    public void MyMethod()
    {
        // 避免这样的编程模型:

        string connectionString = "...";
        IDbConnection connection = new SqlConnection(connectionString);
        connection.Open();
        IDbCommand command = new SqlCommand();
        command.Connection = connection;
        IDbTransaction transaction = connection.BeginTransaction();// 登记
        command.Transaction = transaction;
        try
        {
            /* 此处与数据库交互, 然后提交事务 */
            transaction.Commit();
        }
        catch
        {
            transaction.Rollback(); // 中止事务
        }
        finally
        {
            connection.Close();
            command.Dispose();
            transaction.Dispose();
        }
    }
}
```

通过调用 connection 对象的 BeginTransaction() 方法, 可以获得代表底层数据库事务的一个对象。函数 BeginTransaction() 返回了一个实现了 IDbTransaction 接口的对象, 通过它管理事务。当数据库被登记到事务中时, 并不会真正执行任何请求, 而仅仅是将针对事务的请求记录下来。假如对数据库的所有更新或其他改变都满足一致性要求, 并且没有发生错误, 那么只要简单地调用事务对象的 Commit() 方法, 就将指示

数据库把所有的更改作为一个原子操作提交出去。如果有异常发生, `Commit()` 方法将被跳过, `catch` 语句则通过调用 `Rollback()` 方法中止事务。中止事务要求数据库丢弃目前记录的所有变更。

### 事务管理的挑战

显式编程模型简单而又直接, 它对执行事务的服务没有任何要求, 适用于一个客户端调用一个单独的服务, 并且该服务只与一个单独的数据库(或一个单独的事务性资源)交互, 服务通过它启动以及管理事务, 如图 7-2 所示。

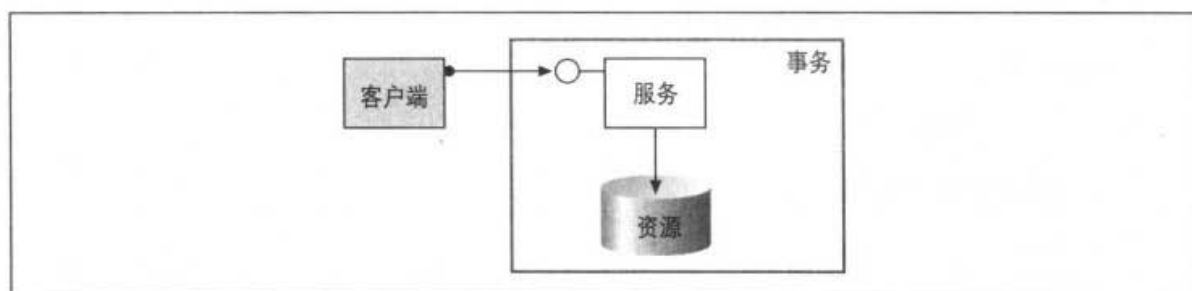


图 7-2: 单服务（单资源）事务

事务管理的障碍要归结于事务的协调问题。考虑如下的实例, 在一个面向服务应用程序中, 客户端将与多个服务发生交互, 而这些服务会轮流与彼此及多个资源交互, 如图 7-3 所示。

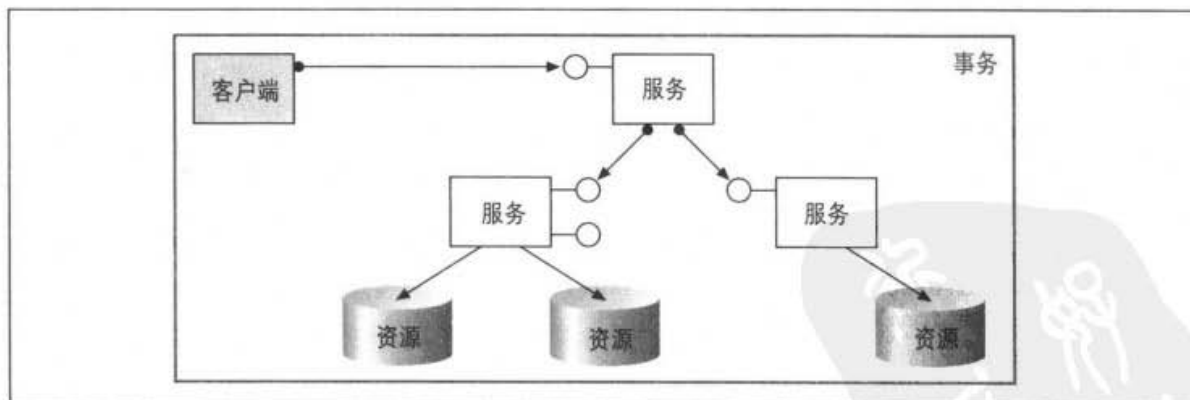


图 7-3: 分布式的事务型面向服务应用程序

问题在于, 究竟是哪一个参与的服务会负责启动事务并登记每一个资源? 如果它们都这样做, 最终我们将得到多个事务。把资源登记的逻辑放到服务代码中会导致服务与资源之间的高度耦合。更重要的是由哪一个服务负责提交或回滚事务? 一个服务如何获知其他服务对该事务的了解程度? 管理事务的服务又如何向其他服务通知该事务的结果? 试图将事务对象或某个标识符当作一个操作参数来传递, 显然违背了面向服务的原则, 因

为客户端和服务都可能使用任何一种实现平台与技术。服务当然可以被部署在不同的进程中,甚至跨不同机器或站点,并产生诸如网络故障或机器崩溃等问题,这些都将增加事务管理的复杂性。因为某个服务可能崩溃,而与此同时其他服务可能继续处理该事务。通过在客户端与服务间加入事务协调的逻辑,使得两者产生耦合是一种可行的解决方案,但这种方法十分脆弱,并且甚至可能无法承受对业务流程或参与服务的数目的微小改变。此外,图 7-3 中的服务可能由不同的供应商开发,而这种情况可能会妨碍到事务的协调工作。即使在服务层找到了一种解决协调问题的方法,当涉及多个资源时,由于每个资源都可能独立地造成服务的失败,我们会得到多个彼此独立的失败点。

## 分布式事务

之前提及的尴尬处境就是所谓的分布式事务 (Distributed Transactions)。一个分布式事务包含两个或两个以上独立的服务 (通常在不同的执行上下文中),或者具有一个单独的服务,它包含了两个或两个以上的事务型资源。如果希望显式地管理分布式事务存在的各种潜在错误场景,显然是不切实际的。对于分布式事务,我们需要依靠两阶段提交协议 (Two-Phase Commit Protocol), 以及一个专门的事务管理器。将事务管理的逻辑放置到服务代码中显然非我们所愿,而事务管理器正是能够帮助我们管理事务的一个第三方工具。

## 两阶段提交协议

为了克服分布式事务的复杂度,事务管理器使用了一种叫做两阶段提交协议的事务管理协议,以决定事务的结果以及提交或回滚对系统状态的改变。两阶段提交协议实现了分布式系统的原子性和一致性。协议使得 WCF 能够支持涉及多个客户端、服务和资源的事务。在本章后面我们可以看到事务启动的方式,以及它们流经服务边界的方式。现在,我们需要着重注意的是当一个事务正在进行时,事务管理器基本上不加干涉。新的服务可能加入到事务中,并且每一个被访问的资源都会被登记到事务中。服务执行业务逻辑,而资源则记录在事务范围内产生的变化。在事务进行期间,所有服务 (以及参与到事务中的客户端) 都必须投票表决它们是希望提交它们所执行的修改,还是以某种理由中止事务。

当事务结束时 (在后面我们会看到事务结束的时间), 事务管理器会检查参与服务的综合投票结果。如果有任何服务或客户端投票要求中止,事务将被销毁。所有参与的资源都将被要求放弃事务过程中所做的改变。如果事务中的所有服务都投票要求提交,则启动两阶段提交协议。在第一阶段,事务管理器询问所有参与事务的资源,了解它们在事务过程中提交修改记录时是否有所保留,也就是说,当被要求提交修改时,它们是否能如实地完成修改? 值得注意的是,此时事务管理器并没有指示资源去提交修改,它仅仅是咨询该资源在此问题上的投票结果。在第一阶段末期,事务管理器已经组合了各种资

源的投票结果。协议的第二阶段则基于这样的综合结果开展工作。如果在第一阶段中所有的资源投票都要求提交事务，那么事务管理器将通知所有的资源提交修改。哪怕只有一个资源在第一阶段表明自己无法提交修改，那么在第二阶段内，事务管理器都会通知所有的资源回滚所做的修改，从而中止事务并将系统恢复到发生事务前的状态。

必须强调的是，一个资源如果在被询问后，投出了提交票，那么这一票有其特殊含义：它是一种不可违背的承诺。如果一个资源投票要求提交事务，就意味着当它在随后的第二阶段被要求提交修改时，绝不能出现任何失败。资源必须在投票要求提交前核实所有的修改是否一致，是否合法。资源的投票决定不可撤销，这是实现分布式事务的基础，而各个资源供应商都已竭尽全力准确地实现这一行为。

## WCF 资源管理器

WCF 资源管理器 (RM) 可以是任何一种支持自动注册以及两阶段提交协议的资源，这里的两阶段提交协议将由一个 WCF 的事务管理器进行管理。该资源必须检测自身是否被事务访问，并自动登记到事务中。RM 既可以是一种持久性资源也可以是一种易失性资源，如一个事务型整数、字符串或集合。虽然 RM 必须支持两阶段提交协议，但是它也可以有选择地实现一个经过优化的协议，该协议用于事务中仅有一个 RM 的情况。该优化协议被称为一阶段提交协议，RM 只需使用一个步骤通知事务管理器对事务提交的尝试是成功还是失败。

## 事务传播

WCF 可以跨越服务边界传播事务。这就使得服务能够参与到客户端的事务中，客户端也可以在相同事务中包含多个服务的操作。客户端自身既可以是一个 WCF 服务，也可以不是。客户端的事务是否向服务传播由绑定和操作契约的配置共同决定。通过配置使得绑定具有将客户端事务传播到服务的能力，称为事务相关的绑定 (Transaction-Aware Binding)。并非所有的绑定都是事务相关的，只有与 TCP-、IPC- 及 WS- 有关的绑定才与事务相关 (它们分别对应了 NetTcpBinding、NetNamedPipeBinding、WSHttpBinding、WSDualHttpBinding 和 WSFederationHttpBinding)。

## 事务流与绑定

事务相关的绑定在默认情况下并不会传播事务。与 WCF 的大多数设置相同，它被设置为一个参与 (Opt-In) 选项。服务的宿主或管理者必须显式的允许接受传入的事务，这些事务大多跨越了组织或业务边界。为了实现事务的传播，我们必须显式地在服务宿主和客户端两端的绑定中都启用该功能。所有事务相关的绑定都提供了 Boolean 类型的属性 TransactionFlow，例如：

```
public class NetTcpBinding : Binding, ...
{
    public bool TransactionFlow
    {get;set;}
    // 更多成员
}
```

TransactionFlow的默认值为 false。

若要启用事务传播，只需在代码或宿主的配置文件中将该属性设置为 true 即可。例如在 TCP 绑定中：

```
NetTcpBinding tcpBinding = new NetTcpBinding();
tcpBinding.TransactionFlow = true;
```

或采用配置文件：

```
<bindings>
  <netTcpBinding>
    <binding name = "TransactionalTCP"
      transactionFlow = "true"
    />
  </netTcpBinding>
</bindings>
```

注意，属性 TransactionFlow 的值并没有发布在服务的元数据里。如果使用 VS2005 或 SvcUtil 生成客户端的配置文件，我们仍需要按照需求手动地启用或禁用事务流的功能。

## 事务与可靠性

严格说来，事务并不要求可靠的消息传输，原因在于当消息的可靠性被禁用时，如果 WCF 的消息丢失或客户端与服务某一方的连接断开，事务都会被中止。由于我们可以确保客户端在执行事务型操作时或者完全成功，或者完全失败，因此事务本身就是可靠的。但是，启用消息的可靠性选项可以降低中止事务的几率，因为它保证了通信是可靠的，从而使事务不会由于通信的问题被中止。因此，在 NetTcpBinding 和 WSHttpBinding 绑定中启动事务时，我们强烈推荐大家同时启用消息的可靠性选项：

```
<netTcpBinding>
  <binding name = "TransactionalTCP"
    transactionFlow = "true">
    <reliableSession enabled = "true"/>
  </binding>
</netTcpBinding>
```

对于 NetNamedPipeBinding 和 WSDualHttpBinding 绑定则无需启用可靠性，正如我们在第 1 章所讨论的，这两种绑定总是可靠的。

## 事务流与操作契约

使用一个事务相关的绑定乃至启用事务流,并不意味着服务希望在每个操作中使用客户端的事务,或代表客户端的事务在事务传播中占有优先权。这样的服务级决定应该是客户端与服务在契约中达成一致的部分。为此,WCF提供了TransactionFlowAttribute方法特性,以控制客户端的事务是否传播到服务以及传播的时机:

```
public enum TransactionFlowOption
{
    Allowed,
    NotAllowed,
    Mandatory
}

[AttributeUsage(AttributeTargets.Method)]
public sealed class TransactionFlowAttribute : Attribute, IOperationBehavior
{
    public TransactionFlowAttribute(TransactionFlowOption flowOption);
}
```

注意 TransactionFlowAttribute 是一个方法级的特性,因为 WCF 一贯认为关于事务流的决议应该应用在每一个操作的级别上,而不是服务的级别上。

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void MyMethod(...);
}
```

这么做是经过深思熟虑的,因为它可以增强我们对事务流的控制粒度,选择在某些操作上使用客户端事务,而在某些操作上不使用客户端事务。

TransactionFlowAttribute 特性的值包含在服务发布的元数据中,因此在导入一个契约定义时,导入的定义将包含配置的值。WCF 同样允许将 TransactionFlowAttribute 特性直接应用到实现契约的服务类的操作:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod(...);
}

class MyService : IMyContract
{
    [TransactionFlow(TransactionFlowOption.Allowed)]
    public void MyMethod(...)
    { ... }
}
```



但是,这样的用法不值得提倡,因为它会混淆服务契约中的定义,而发布的服务契约则体现了服务逻辑。

### TransactionFlowOption.NotAllowed

当操作被设置为禁止事务流时,客户端无法向服务传播自己的事务。即便在绑定中启用了事务流并且客户端确实拥有一个事务,该事务仍然会被忽略,而不会被传播到服务中。因此,服务将永远无法使用客户端的事务,服务与客户端可以选择任意配置的任意绑定。TransactionFlowOption特性的默认值为TransactionFlowOption.NotAllowed,因此以下两个定义是等效的:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod(...);
}

[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.NotAllowed)]
    void MyMethod(...);
}
```

### TransactionFlowOption.Allowed

当将操作的TransactionFlowOption特性赋予TransactionFlowOption.Allowed值时,它将被设置为允许事务流。此时,如果客户端拥有一个事务,服务将允许客户端的事务流经服务的边界。不过,即使客户端的事务被传播了,服务也不一定会使用它。当我们选择TransactionFlowOption.Allowed时,服务仍然可以被设置为使用任意一种绑定,服务可能与事务相关,也可能无关。但是,客户端与服务的绑定配置必须兼容。对事务流而言,所谓的兼容是指如果服务操作允许事务流,而绑定禁止事务流,那么客户端也应该在自己的绑定中禁用事务流。试图传播客户端的事务会导致错误发生,因为服务无法识别消息中的事务信息。然而,当服务端的绑定配置被设置为允许事务流时,客户端可以自由选择是否启用事务传播,因此,即使服务将TransactionFlow特性设置为True,客户端仍然可以选择将TransactionFlow设置为false。

### TransactionFlowOption.Mandatory

当操作被设置为TransactionFlowOption.Mandatory时,服务与客户端都必须使用启用了事务流的与事务相关的绑定。WCF会在加载服务时验证此项要求,只要服务包含



了一个不兼容的终结点，就会抛出 `InvalidOperationException` 异常。`TransactionFlowOption.Mandatory` 意味着客户端必须具有一个事务以传播给服务。如果客户端没有这样的事务，则调用服务的操作就会抛出一个 `FaultException` 异常，以表示该服务需要一个事务。使用强制事务流选项，客户端的事务将总是传播给服务。但是，服务不一定会使用客户端传播过来的事务。

## 单向调用

从本质上讲，向服务传播客户端事务要求服务能够选择是否中止客户端的事务。这就意味着我们不能通过一个单向操作从客户端事务向服务传播事务，因为单向操作是没有应答信息的。在服务加载阶段，WCF 会对此进行验证，一旦我们为一个单向操作设置了除 `TransactionFlowOption.NotAllowed` 特性在外的其他特性，就会抛出一个异常。

```
// 无效定义：
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void MyMethod(...);
}
```

## 事务协议和管理器

根据事务参与方的执行范围，WCF 会使用各种不同的事务管理协议。协议一词用在这里会有一些误导，因为从理论上讲，这里使用的协议是指两阶段提交协议。而事务管理协议的差异则在于远程调用的类型、通信所用的协议及所跨越的边界种类。

### 轻量级协议 (Lightweight Protocol)

该协议仅用于管理本地环境中的事务，即处在同一应用程序域中的事务。它无法跨越应用程序域的边界传播事务（更不用说跨进程或机器边界），也无法跨越服务边界传播事务（即事务从客户端传播到服务）。轻量级协议只适用于某个服务的内部或外部。但相对于其他协议而言，轻量级协议的性能是最好的。

### OleTx 协议

该协议用于跨越应用程序域、进程和机器边界传播事务并管理两阶段提交协议。协议使用远程过程调用 (RPC)，并且调用的二进制形式是基于 Windows 的。由于同时使用了远程过程调用 (RPC) 和基于 Windows 的格式，该协议无法穿越防火墙或者与非 Windows 方协作。通常情况下，这并不存在问题，因为 OleTx 协议主要用于管理在内网中的 Windows 环境中的事务，同时还会引入一个单独的事务管理器。

### WS 原子性事务 (WSAT) 协议

该协议与OleTx协议非常相似, 同样允许事务穿越应用程序域、进程和机器边界传播以及管理两阶段提交协议。但是, 不同于OleTx协议的是, WSAT协议基于一种行业标准, 并且当它使用HTTP协议时, 编码形式为文本格式, 因而可以穿越防火墙。虽然我们可以在内网中使用WSAT协议, 但它主要还是用于跨Internet的事务管理, 通过Internet引入了多个事务管理器。

## 协议与绑定

因为轻量级协议不能跨越服务边界传播事务, 所以没有绑定支持轻量级协议。各种事务相关的绑定在支持另外两种事务管理协议时也存在差异。TCP和IPC绑定可以被配置为与OleTx和WSAT协议中的一个或全部一起工作。两种绑定都默认采用OleTx协议, 但可以根据需要转换为WSAT协议。与其他绑定属性相似, 对于这两种内网绑定, 都可以通过配置文件或以编码方式配置协议。

WCF 提供的 TransactionProtocol 抽象类定义如下:

```
public abstract class TransactionProtocol
{
    public static TransactionProtocol Default
    {get;}
    public static TransactionProtocol OleTransactions
    {get;}
    public static TransactionProtocol WSAtomicTransactionOctober2004
    {get;}
}
```

NetTcpBinding和NetNamedPipeBinding都提供了匹配类型的TransactionProtocol属性, 例如:

```
public class NetTcpBinding : Binding,...
{
    TransactionProtocol TransactionProtocol
    {get;set;}
    // 更多成员
}
```

若要以编码方式设置协议, 首先要编写特定的绑定类型, 然后再通过调用一个静态方法设置属性:

```
NetTcpBinding tcpBinding = new NetTcpBinding();
// 与传输相关的协议
tcpBinding.TransactionFlow = true;
tcpBinding.TransactionProtocol = TransactionProtocol.WSAtomicTransactionOctober2004;
```

需要注意的是, 事务协议的配置只有在允许事务传播的情况下才有意义。

要在配置文件中设置协议，可以像平常一样定义绑定节：

```
<bindings>
  <netTcpBinding>
    <binding name = "TransactionalTCP"
      transactionFlow = "true"
      transactionProtocol = "WSAtomicTransactionOctober2004"
    />
  </netTcpBinding>
</bindings>
```

当我们为 TCP 或 IPC 绑定设置协议时，必须保证服务与客户端使用的协议是相同的。

由于 TCP 和 IPC 绑定只能在内网使用，将它们设置为 WSAT 协议并无实际意义，并且 WSAT 协议的功能主要是针对完整性。

WS 绑定（如 WSHttpBinding、WSDualHttpBinding 和 WSFederationHttpBinding）的设计目的在于当涉及多个使用 WSAT 协议的事务管理器时，能够跨越 Internet。但是，如果 Internet 只涉及一个单独的事务管理器，OleTx 协议将是默认的绑定，不必也不能为它配置一个特殊的协议。

## 事务管理器

回顾本章开始的论述，曾经提及依靠自身管理事务实属无奈之举，最好的解决办法是找一个第三方工具如事务管理器，为我们的客户端和服务管理两阶段提交协议。如图 7-4 所示，WCF 在一个提供者模型中可以与三个不同的事务管理器一起工作，而不只限于一个。

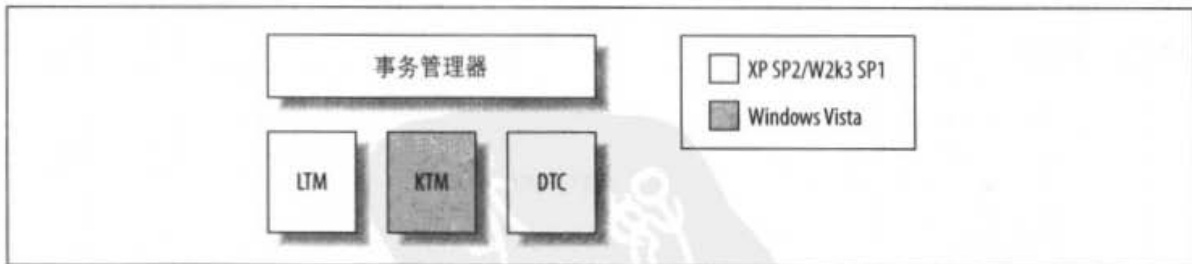


图 7-4：WCF 事务管理器

这三个事务管理器分别是轻量级事务管理器（LTM）、核心事务管理器（KTM）和分布式事务协调器（DTC）。WCF 根据平台使用的功能，应用程序的事执行任务、调用的服务以及所消耗的资源分配合适的事务管理器。通过自动地分配事务管理器，WCF 将事务管理从服务代码和用到的事务协议中解耦出来，开发者不必再为事务管理器而苦恼了。接下来，我们的关注目标是如何将开发者从提高系统性能与效率的繁重工作中解脱出来。

### 轻量级事务管理器 LTM

LTM 只能管理一个本地事务，即在一个单独应用程序域内的事务。LTM 使用轻量级事务协议来管理两阶段提交协议。它所管理的事务，最多只能引入一个单独的持久化资源管理器，但对于易失型资源管理器而言，则不存在这样的限制。如果当前只有一个单独的资源管理器，且该资源支持单一阶段提交，那么 LTM 将采用优化后的协议。最重要的是，LTM 只能管理一个在单独服务内的事务，而且该服务不能将事务传递给其他服务。LTM 在所有的事务管理器中，性能最佳。

### 核心事务管理器 KTM

KTM 可被用于在 Windows Vista 操作系统上管理事务型的核心资源管理器 (KRM)，特别是事务型文件系统 (TXF) 与事务型注册表 (TXR)。KTM 使用基于直接内存与核心调用的轻量级事务协议。KTM 管理的事务，最多只能引入一个单独的持久化 KRM，但如果事务包含的是易失型资源管理器，则对管理器的数目没有限制。与 LTM 一样，KTM 管理的事务最多只能引入一个服务，并且该服务不得向其他服务传播事务。

### 分布式事务协调器 DTC

DTC 有能力管理跨越任意执行边界的事务，从本地（例如相同的应用程序域）跨越所有的边界，诸如进程、机器或站点的边界。DTC 既可以使用 OleTx 协议，也可以使用 WSAT 协议。DTC 是在事务流经服务边界时所用的事务管理器，它可以便捷地管理引入任意期望数目的服务和资源管理器的事务。

DTC 与 WCF 紧密的集成在一起，它是每个运行 WCF 的机器上默认可用的一项系统服务。DTC 可以创建新的事务，跨机器传播事务，收集资源管理器的投票并通知资源管理器进行回滚或提交。例如，考虑如图 7-5 所示的面向服务应用程序，非事务型的客户端访问了机器 A 的一个服务，而机器 A 的服务被设置为使用事务。该服务将成为事务的根，它不仅能够启动事务，还能在事务完成后发出指示。

---

**注意：**WCF 中的每个事务最多只能有一个根服务，因为一个非服务型的客户端也可以成为事务的根。在任何情况下，事务的开始和结束都是由根来完成的。

---

如果一个服务属于机器 A 中某个服务的一部分，当它试图访问机器 B 的另一个服务或资源时，实际上它会包含一个到远程服务或资源的代理。这个代理向机器 B 传播事务的 ID。机器 B 上的拦截器会联系本地的 DTC，把事务 ID 传递给它，通知它启动对机器 B 上事务的管理。由于事务 ID 被传递到机器 B，因而机器 B 上的资源管理器可以自动对它进行登记。同理，可以推断事务 ID 被传播到机器 C 的情况。

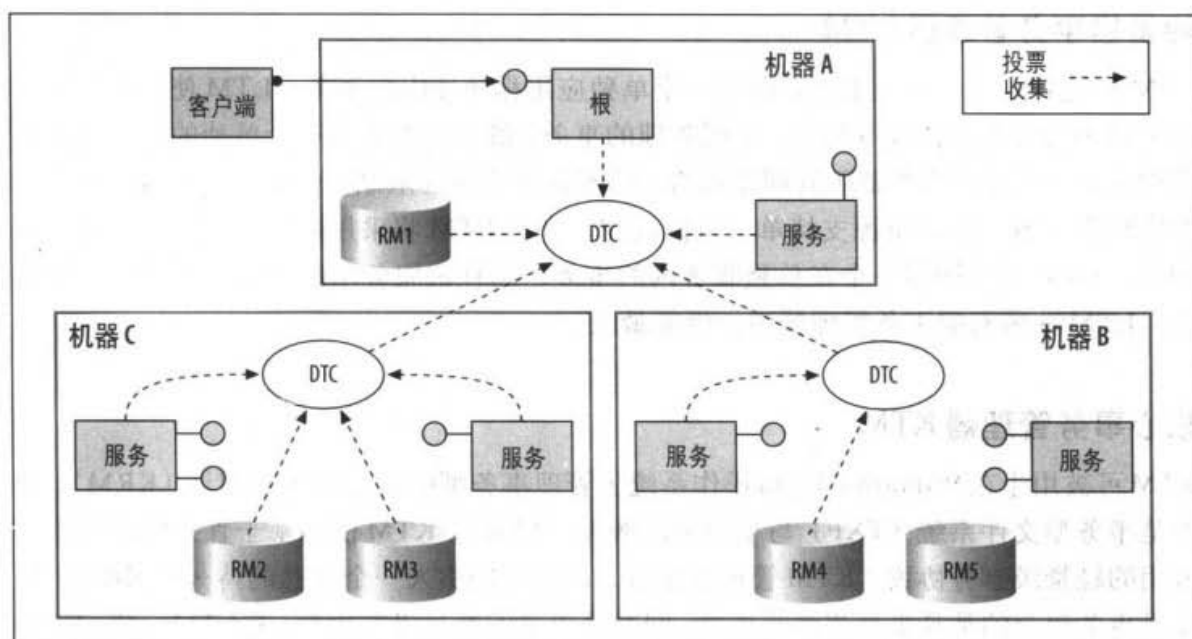


图 7-5: DTC 管理事务

当事务完成后，如果综合的服务投票结果为提交该事务，就应该启动两阶段提交协议。根机器上的DTC收集它的资源管理器的投票，并联系每一个参与事务的机器上的DTC，通知它们在自己的机器上进行协议的第一阶段。在远程机器上的DTC收集本机上资源管理器的投票，并把结果转交回根机器上的DTC。根机器上的DTC收到来自所有远程DTC的结果之后，它会综合资源管理器的投票。如果所有的资源管理器都投票要求提交，那么在根机器上的DTC会再次联系在远程机器上的所有DTC，通知它们在各自的机器上进行协议的第二阶段并提交事务。即使只有一个资源管理器投票要求中止事务，根机器上的DTC都会通知正在各自机器上执行第二阶段的远程机器上的所有DTC，以中止事务。值得注意的是，只有在根机器上的DTC才能综合第一阶段的投票结果，也只有它才能指定最终是中止还是提交。

## 事务管理器的提升

WCF能够为事务动态地分配合适的事务管理器。如果某个事务管理器不能满足事务的要求，WCF将提升事务，即请求上一级的事务管理器来处理该事务。一个事务可被提升多次。一旦事务被提升，它将保持被提升的状态而无法降级，原先用于管理该事务的事务管理器也将被直接忽略。在事务动态提升的支持下，开发者可以避免直接与事务管理器进行交互，因为这么做很难实现事务提升。我们之所以不应该编写如例7-1所示的代码，其中一个原因在于事务提升，因为例7-1会阻止任何提升事务的可能。

## LTM 的提升

WCF中的每一个事务通常都从被LTM管理开始。只要该事务仅与一个持久化资源交互，并且不会将事务传播给一个WCF服务，LTM就可以管理该事务，并获得最好的吞吐量及性能。但是，如果该事务试图登记第二个持久化资源或要传播事务到某个服务，WCF将把它从LTM提升为DTC。如果事务访问的第一个资源是一个KTM资源，此时，WCF将执行另一种类型的事务提升，将它从LTM提升为KTM。

## KTM 的提升

只要事务只与一个KRM交互，并且该事务是在本地运行的，就可以使用KTM进行管理。KTM可以管理多个易失型资源管理器。当事务被传播到另一个服务，或者第二个持久化资源（内核资源或一般资源）被登记时，KTM事务将被提升为DTC。

## 资源与升级

在创作本书时，能够参与LTM事务的资源只包括易失型资源管理器以及SQL Server 2005的各种版本。旧有的资源管理器如SQL Server 2000、Oracle、DB2和MSMQ只能参与到DTC事务中。因此，当一个LTM事务访问某个旧有资源时，就算它是事务中唯一的一个资源，该事务也会被自动提升为DTC。表7-1概括了资源与事务管理器之间的关系。

表 7-1: 资源与事务管理器

资源	LTM	KTM	DTC
易失型资源	Yes	Yes	Yes
SQL Server 2005	Yes	No	Yes
核心资源	No	Yes	Yes
其他 RM	No	No	Yes

## 事务类

定义在System.Transactions命名空间中的Transaction类，是.Net 2.0新引入的一个类，它代表了所有WCF事务管理器能够管理的事务：

```
[Serializable]
public class Transaction : IDisposable, ISerializable
{
    public static Transaction Current
    {get;set;}
}
```



```
public void Rollback(); // 中止事务  
public void Dispose();
```

```
// 更多成员
```

```
}
```

开发者很少需要直接使用 `Transaction` 类。`Transaction` 类的主要用途是通过调用它的 `Rollback()` 方法手动中止事务。`Transaction` 类还可用于登记资源管理器、设定事务隔离级别、订阅事务事件、为并发线程克隆事务，以及获取事务的状态和信息。

## 环境事务

.NET 2.0 定义了环境事务 (Ambient Transaction) 的概念。环境事务就是就是执行在代码中的事务。调用 `Transaction` 类的静态属性 `Current` 可以获得一个指向环境事务的引用：

```
Transaction ambientTransaction = Transaction.Current;
```

如果环境事务不存在，则 `Current` 属性将返回 `null`。每一段代码，无论它是属于客户端还是服务，总是能够获得自己的环境事务。环境事务对象保存在线程本地存储 (TLS) 中。因此，当线程跨越相同调用链中的多个对象和方法时，所有的对象和方法都能访问它们的环境事务。

在 WCF 中，环境事务极为重要。只要它存在，任何一个 WCF 资源管理器都会在环境事务中自动登记。当客户端调用一个 WCF 服务时，如果客户端存在一个环境事务，并且相关的绑定和契约被设置为允许事务传播，那么该环境事务将被传播到服务中。

---

**警告：** 客户端不能向服务传播一个已经被中止的事务。这样做将引发一个异常。

---

## 本地事务与分布式事务

`Transaction` 类适用于本地事务与分布式事务。每个事务都有两个标识符用来识别本地事务和分布式事务。我们可以通过访问 `Transaction` 类中的 `TransactionInformation` 属性获得事务的标识符：

```
[Serializable]  
public class Transaction : IDisposable, ISerializable  
{  
    public TransactionInformation TransactionInformation  
    {get;}  
    // 更多成员  
}
```



TransactionInformation属性的类型为TransactionInformation, 它的定义如下:

```
public class TransactionInformation
{
    public Guid DistributedIdentifier
    {get;}
    public string LocalIdentifier
    {get;}
    // 更多成员
}
```

TransactionInformation类提供了对两个标识符的访问。它们主要用于记录、跟踪以及分析。在本章, 我们将使用标识符这种便捷的办法演示如何在代码中配置事务流。

### 本地事务标识符

本地事务标识符(本地ID)包含了一个当前应用程序域中LTM的标识符, 以及一个用来列举事务数量的整数。我们可以通过TransactionInformation类的LocalIdentifier属性访问本地ID。环境事务总是可以获得本地ID, 并且它的值永远不会为null。只要环境事务存在, 那它必然有一个有效的本地ID。本地ID的值包含两部分: 一部份为GUID常量, 它在每个应用程序中是唯一的, 代表了为该应用程序域分配的LTM; 另一部分则为一个递增的整数, 它用来列举到目前为止被该LTM管理的事务的数目。

例如, 某一个服务参与了三个连续的事务, 从第一个调用开始, 我们将得到如下的结果:

```
8947aec9-1fac-42bb-8de7-60df836e00d6:1
8947aec9-1fac-42bb-8de7-60df836e00d6:2
8947aec9-1fac-42bb-8de7-60df836e00d6:3
```

GUID对于每个应用程序域而言是恒定不变的。如果客户端与服务处在相同的应用程序域中, 它们将拥有相同的GUID; 如果客户端发送了一个跨应用程序域的调用, 它将得到一个属于自己的独一无二的GUID标识符, 该标识符代表了它自己的本地LTM。

### 分布式事务标识符

无论何时, 如果一个由LTM或KTM管理的事务被提升为由DTC管理(如环境事务被传播给另一个服务时), 系统就会自动生成一个分布式事务标识符(分布式ID)。我们可以通过TransactionInformation类的DistributedIdentifier属性访问分布式ID。分布式ID对于每个事务而言都是唯一的, 两个事务永远都不可能拥有相同的分布式ID。最重要的, 在跨越服务边界期间, 以及在跨越整个调用链从最顶端的客户端贯穿到调用链中每个服务与对象期间, 分布式ID都是统一的。这对于记录及跟踪分布式事务非常有用。值得注意的是, 在事务还没有被提升时, 分布式ID的值可能为Guid.Empty。当客户端作为事务的根且该客户端尚未调用服务时, 分布式ID在客户端一端的值通常为

Guid.Empty, 而在服务一端, 如果服务并没有使用客户端的事务, 而是转而启动了自己的本地事务时, 该值为空。

## 事务型服务编程

WCF为服务提供了一种简单优雅的声明式的编程模型。但是, 这种模型并不适用于被服务调用的非服务型代码以及非服务型的 WCF 客户端。

### 环境事务的设置

在默认情况下, 服务类以及它的所有操作都不包含环境事务。即使是当客户端事务被传播到服务时同样如此。考虑下列服务:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    void MyMethod(...);
}
class MyService : IMyContract
{
    public void MyMethod(...)
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction == null);
    }
}
```

尽管强制的事务流确保了客户端事务的传播, 但是服务的环境事务仍将为null。为了拥有一个环境事务, 服务必须标识每个契约方法, 向 WCF 标明该方法需要将方法体包含在一个事务中。为此, WCF在OperationBehaviorAttribute中提供了TransactionScopeRequired 属性:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute, ...
{
    public bool TransactionScopeRequired
    {get;set;}
    // 更多成员
}
```

TransactionScopeRequired的默认值为false, 因此在默认情况下服务不拥有环境事务。将TransactionScopeRequired设置为true, 就可以为相应的服务操作提供一个环境事务:

```
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction != null);
    }
}
```

如果客户端事务被传播给服务，WCF 会把客户端事务设置为服务操作的环境事务。否则，WCF 会为该操作创建一个新的环境事务并把它设置为服务操作的环境事务。

**警告：** 服务类的构造函数无法包含事务：它无法参与到客户端的事务中去，我们也无法要求 WCF 将其包含到一个事务中。除了手动创建一个新的环境事务（稍后展示），我们不能在服务的构造函数中执行与事务有关的工作。

图 7-6 展示了在绑定配置、契约操作以及本地操作（即契约操作的具体实现）的行为特性影响下，一个 WCF 服务应该使用哪个事务。

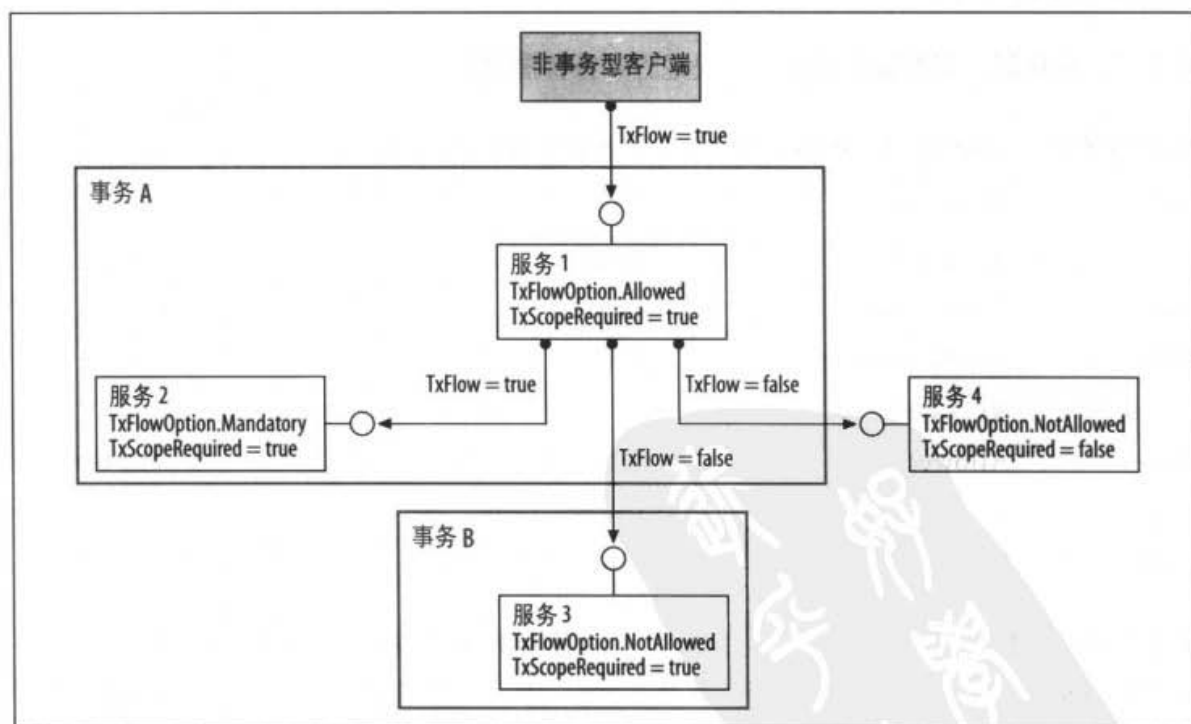


图 7-6：契约、绑定以及操作行为对事务传播的影响

图 7-6 中，一个不使用事务的客户端调用了服务 1。服务 1 的操作契约被设定为 `TransactionFlowOption.Allowed`。尽管在绑定中已启用了事务流，但是由于客户

端没有使用事务，因此不会有事务被传播给服务 1。服务 1 的行为特性被设置为需要一个事务范围 (Transaction Scope)，因此，WCF 为服务 1 创建了一个新的事务 A (如图 7-6 所示)。然后，服务 1 调用了其他三个服务，而这些服务的配置各不相同。服务 2 所用的绑定也启用了事务流，并且在它的操作契约中对事务流做了强制要求。同时，服务 2 的行为特性又被设置为需要事务范围，因此 WCF 将把事务 A 设为服务 2 的环境事务。服务 3 的绑定和操作契约都不支持事务流。但是由于服务 3 具有的操作行为需要一个事务范围，因此 WCF 会为服务 3 创建了一个新的事务 (事务 B)，并将其设置为服务 3 的环境事务。与服务 3 类似，服务 4 的绑定和操作契约也不支持事务流，但因为服务 4 对事务范围没有要求，所以它没有环境事务。

## 事务传播模式

服务使用哪个事务由绑定的事务流属性 (两个值)、操作契约中的事务流选项 (三个值) 以及操作行为特性中的事务范围属性 (两个值) 共同决定。一共有 12 种可能的配置设定。在这些配置设定中，有 4 种不满足一致性要求 (例如在绑定中禁用事务流，却在操作契约中却强制要求事务流)，或者因为根本不可行而被 WCF 排除在外。表 7-2 列出了剩下的 8 种情况 (注 2)。

表 7-2: 由绑定、契约以及操作行为决定的事务传播模式

绑定事务流	TransactionFlowOption	TransactionScopeRequired	事务模式
False	Allowed	False	None
False	Allowed	True	Service
False	NotAllowed	False	None
False	NotAllowed	True	Service
True	Allowed	False	None
True	Allowed	True	Client/Service
True	Mandatory	False	None
True	Mandatory	True	Client

表 7-2 的 8 种排列组合实际上最终只产生了四种事务传播模式。我们把这 4 种方式分别称为 Client/Service、Client、Service 及 None。表 7-2 用黑体字标明了针对各个模式推荐的配置方法。在设计应用程序时，每种方式都有它自己的适用场景，因此，如能了解选取正确模式的方式，将有利于我们选择事务支持，简化对事务支持的配置。

注 2: 在 MSDN 杂志 2007 年 5 月刊中，作者首次提出了事务传播模式的概念。

## Client/Service 事务

顾名思义, Client/Service 模式能够确保服务使用客户端的事务, 或在客户端没有事务时使用服务的事务。要配置为该模式, 需要完成以下步骤:

1. 选择一个事务相关的绑定, 并通过将 `TransactionFlow` 设置为 `True` 启用事务流选项。
2. 将操作契约中的事务流选项设置为 `TransactionFlowOption.Allowed`。
3. 将操作行为的 `TransactionScopeRequired` 属性设置为 `true`。

Client/Service 模式是耦合度最低的配置, 因为服务对客户端的功能所知最少。如果客户端存在一个可供传播的事务, 服务将加入到该客户端事务中。通常, 服务加入客户端事务有利于整个系统的一致性。如果服务使用一个与客户端不同的事务, 那么就可能出现其中一个事务被提交而另一个事务被中止的情况, 最终使系统处于不一致的状态。当服务加入到客户端事务时, 客户端与服务 (以及客户端可能调用的其他服务) 所做的全部工作会被当作一个原子操作被一起提交或中止。假如客户端没有使用事务, 而服务仍然要求得到事务的保护, 那么在这种模式下, 系统会将服务作为一个新事务的根节点, 从而为它提供一个附加的事务。例 7-2 向我们展示了一个被配置为 Client/Service 事务模式的服务。

例 7-2: Client/Service 事物模式的配置

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void MyMethod(...);
}

class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction != null);
    }
}
```

注意, 在例 7-2 中, 服务会断言它总有一个事务, 但服务无法假设或者断言该事务是否是一个客户端的事务或本地创建的事务。当服务被单独使用或被用作一个更大的事务中的一部分时, Client/Service 模式都是不错的选择。但是, 当我们选择该模式时, 如果生成的事务是服务端事务, 我们应特别注意可能发生的死锁现象。由于资源隔离了每个事

务对它的访问,且服务端事务又将是一个新的事务,那么当其他事务试图访问相同的资源时,就可能会导致死锁。当我们使用 Client/Service 模式时,服务不一定是事务的根,但不管服务是根,还是加入客户端事务,其执行方式都必须保持一致。

## 要求事务流

Client/Service 模式需要使用启用了事务流功能的与事务相关的绑定,不过, WCF 在加载服务时并没有强制要求。为了强化这方面的要求,可以使用我们编写的 BindingRequirementAttribute 类:

```
[AttributeUsage(AttributeTargets.Class)]
public class BindingRequirementAttribute : Attribute, IServiceBehavior
{
    public bool TransactionFlowEnabled // 默认值为 false
    {get;set;}
    // 更多成员
}
```

该特性可以直接应用于服务类。TransactionFlowEnabled 的默认值为 false。然而对于每个终结点而言,当我们把它设置为 true 时,只要终结点的契约包含了至少一个将 TransactionFlow 属性设置为 TransactionFlowOption.Allowed 的操作, BindingRequirement 特性就将强制要求该终结点使用一个 TransactionFlow 属性值为 true 的事务相关的绑定:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void MyMethod(...);
}

[BindingRequirement(TransactionFlowEnabled = true)]
class MyService : IMyContract
{...}
```

通过在启动宿主时抛出一个 InvalidOperationException 异常,可以强制绑定满足这一要求。例 7-3 演示了 BindingRequirementAttribute 类的某些简单实现。

### 例 7-3: BindingRequirementAttribute

```
[AttributeUsage(AttributeTargets.Class)]
public class BindingRequirementAttribute : Attribute, IServiceBehavior
{
    public bool TransactionFlowEnabled
    {get;set;}
}
```



```

void IServiceBehavior.Validate(ServiceDescription description,
                               ServiceHostBase host)
{
    if(TransactionFlowEnabled == false)
    {
        return;
    }
    foreach(ServiceEndpoint endpoint in description.Endpoints)
    {
        Exception exception = new InvalidOperationException(...);

        foreach(OperationDescription operation in endpoint.Contract.Operations)
        {
            foreach(IOperationBehavior behavior in operation.Behaviors)
            {
                if(behavior is TransactionFlowAttribute)
                {
                    TransactionFlowAttribute attribute =
                        behavior as TransactionFlowAttribute;
                    if(attribute.Transactions == TransactionFlowOption.Allowed)
                    {
                        if(endpoint.Binding is NetTcpBinding)
                        {
                            NetTcpBinding tcpBinding =
                                endpoint.Binding as NetTcpBinding;
                            if(tcpBinding.TransactionFlow == false)
                            {
                                throw exception;
                            }
                            break;
                        }
                        ... // 检查余下的与事务相关的绑定, 与前面相似
                        throw new InvalidOperationException(...);
                    }
                }
            }
        }
    }
}

void IServiceBehavior.AddBindingParameters(...)
{}

void IServiceBehavior.ApplyDispatchBehavior(...)
{}
}

```

BindingRequirementAttribute 类是一个服务行为, 因此它支持第 6 章介绍的 IServiceBehavior 接口。在启动宿主时, 会调用 IServiceBehavior 接口的 Validate() 方法, 通过该方法我们能够中止服务的装载过程。Validate() 首先会检查 TransactionFlowEnabled 属性是否为 false。如果是, Validate() 方法则立即返回, 什么也不做; 如果 TransactionFlowEnabled 为 true, Validate() 将遍历在服



务描述中可用的服务终结点的集合。它会通过每个终结点获取操作的集合,进一步通过每个操作访问操作的行为集合。所有的操作行为,包括 `TransactionFlowAttribute` 都实现了 `IOperationBehavior` 接口。如果行为是 `TransactionFlowAttribute` 特性, `Validate()` 会检查该值是否被设置为 `transactionFlowOption.Allowed`; 如果是, `Validate()` 将进一步检查服务的绑定。对于每种事务相关的绑定,它将检验该绑定是否将 `TransactionFlow` 属性设置为 `true`; 如果没有,将抛出一个 `InvalidOperationException` 异常。假如该终结点使用了一个非事务性绑定, `Validate()` 同样会抛出一个 `InvalidOperationException` 异常。

**注意:** 例 7-3 中所演示的用于实现 `BindingRequirementAttribute` 的技术具有一定的通用性,我们可以使用类似的方法强制要求绑定的需求。例如, `BindingRequirementAttribute` 定义了另外一个属性 `WCFOnly`,用以强制 WCF 到 WCF 绑定的使用。而定义的 `ReliabilityRequired` 属性,则用来强制使用支持可靠性的可靠绑定:

```
[AttributeUsage(AttributeTargets.Class)]
public class BindingRequirementAttribute :
    Attribute, IServiceBehavior
{
    public bool ReliabilityRequired
    {get;set;}
    public bool TransactionFlowEnabled
    {get;set;}
    public bool WCFOnly
    {get;set;}
}
```

## Client 事务

Client 模式确保服务只会使用客户端的事务,可以采用以下步骤配置该模式:

1. 选取一个事务相关的绑定并通过将 `TransactionFlow` 设置为 `true` 启用事务流功能。
2. 将操作契约中的事务流选项设置为 `TransactionFlowOption.Mandatory`。
3. 将操作行为的 `TransactionScopeRequired` 属性设置为 `true`。

如果服务必须使用它的客户端的事务,而且不会被单独使用,我们在设计该服务时应该选择 Client 事务模式。这样做的主要目的是避免死锁,并最大限度地满足系统的总体一致性。让服务共享客户端的事务,可以降低发生死锁的可能性,因为所有被访问的资源都将被登记在相同的事务中,从而避免多个事务相互竞争对相同资源及其底层的锁的访问权。只使用一个事务可以最大限度地保证系统的一致性,因为该事务将作为一个原子操作提交或中止。例 7-4 演示了一个被设置为 Client 事务模式的服务。

例 7-4: 配置为 Client 事务模式

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    void MyMethod(...);
}
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction.TransactionInformation.
            DistributedIdentifier != Guid.Empty);
    }
}
```

注意, 例 7-4 的方法包含了判断事务为环境事务, 还是分布式事务的断言, 这意味着此处的环境事务来源于客户端。

## Service 事务

Service 模式确保服务总是拥有一个事务, 而不管它的客户端是否使用事务。该服务是一个新事务的根。我们可以采用以下步骤配置该模式:

1. 选取任一绑定。如果选择了一个事务相关的绑定, 保留其 TransactionFlow 属性的默认值, 或显式地将它设置为 false。
2. 不要应用 TransactionFlow 特性, 或者将其设置为 TransactionFlowOption.NotAllowed (译注 2)。
3. 把操作行为的 TransactionScopeRequired 属性设置为 true。

当服务需要在客户端事务的范围外执行事务型工作时, 我们会选择 Service 事务模式。例如, 当我们需要执行一些记录或审核的操作时; 或者当我们需要客户端发布事件供其订阅时, 不用考虑客户端事务是提交还是中止。考虑 Logbook 服务的实例, 它能够将错误记录到数据库。如果客户端发生了一个错误, 它会调用 Logbook 服务或其他手段记录这个错误。而在错误被记录后, 客户端的错误会中止客户端事务。如果服务使用客户端事务, 那么一旦客户端事务被中止, 被记录的错误就因为事务的回滚而从数据库中消失, 我们将无法查看该错误记录, 从而失去了之前记录它的意义。如果将服务配置为拥有属

---

译注 2: 如果绑定不支持事务流, 可以将 TransactionFlowOption 设置为 TransactionFlowOption.Allowed。

于自己的事务，那么即便客户端事务中止，服务方对错误的记录仍会被提交。不过，这样做显然会对系统的一致性产生潜在的危害，因为在客户端事务提交的时候服务方事务可能会被中止。

这种模式无疑更适合于服务事务成功并提交的可能性远远大于客户端事务的情况。日志服务的例子就是一种常见的情况，因为一旦确定的日志存在，相关操作一般都能成功，而与之相反，业务事务却可能由于各种各样的原因而失败。通常，我们应谨慎使用 Service 事务模式，要注意避免因为两个事务（客户端的事务以及服务的事务）一个中止而另一个提交，从而对系统一致性造成的危害。日志服务与审核服务都是选择这种模式的典型代表。

例 7-5 演示了一个被配置为 Service 事务模式的服务。

例 7-5：配置为 Service 事务模式

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod(...);
}
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction.TransactionInformation.
            DistributedIdentifier == Guid.Empty);
    }
}
```

注意，在例 7-5 中服务能够断言自己实际拥有一个本地事务。

## None 事务

None 事务模式是指服务永远不使用事务。可以采用以下步骤配置该模式：

1. 选取任一绑定。如果选择了一个事务相关的绑定，保留其 TransactionFlow 属性的默认值，或显式地将它设置为 false。
2. 不要应用 TransactionFlow 特性，或将其设置为 transactionFlowOption.NotAllowed。
3. 无需设置操作行为的 TransactionScopeRequired 属性，如果要设置，则设置为 false。

当需要服务执行某些操作,而这些操作如果并不是非常重要,并且不会因为它们的失败而导致客户端事务中止时,可以采用该模式。例如,一个打印转账收据的服务不能因为打印机缺纸而中止客户端事务。None事务模式还有一种应用场景:当我们需要提供一些定制行为,以及自己提供对事务编程的支持,或者手动地登记资源,正如例7-1所示的调用旧有的代码。显然, None 事务模式存在一定的风险,因为它可能损害系统的一致性:假如调用的客户端有一个事务,而它所调用的服务则被配置为 None 事务模式,然后客户端中止了它的事务,那么服务对系统状态所做的改变将不会回滚。这种方式的另一个隐患则在于一个被配置为 None 事务模式的服务,在调用另一个被配置为 Client 事务的服务时,调用会失败,因为调用服务没有事务可供传播。

例7-6 演示了一个被配置为 None 事务模式的服务。

例7-6: 配置为 None 事务模式

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
class MyService : IMyContract
{
    public void MyMethod()
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction == null);
    }
}
```

注意例7-6中的服务能够断言自己没有环境事务。

None 事务模式允许我们在事务性客户端中调用非事务型服务。如前所述, None 事务模式主要适用于那些通过它可以得到改善的操作。这样带来的问题是, None 事务模式的服务所抛出的任何异常都将中止调用客户端的事务,因此在这种操作中应该避免出现这样的情况。我们可以通过让客户端捕获 None 事务模式的服务抛出的所有异常,以避免破坏客户端的事务。例如,以如下方式调用例7-6中的服务:

```
MyContractClient proxy = new MyContractClient();
try
{
    proxy.MyMethod();
    proxy.Close();
}
catch
{
}
```

**注意：**即使将 None 事务服务的操作配置为单向操作，我们仍然需要将对它的调用包装在一个 catch 语句中，因为单向操作仍然可以抛出消息传输方面的异常。

## 选择一个服务的事务模式

在以上四种模式中，Service 事务模式与 None 事务模式无疑比较特殊。它们适用于之前提及的特殊场景中的环境，但在其他环境下却会对系统的一致性产生潜在的威胁。我们应该尽量使用 Client/Service 或 Client 事务模式，选择的基准则在于服务能否被单独使用，同时根据潜在的死锁与一致性问题作出判断。务必谨记避免使用 Service 事务模式或 None 事务模式。

## 投票与完成

尽管 WCF 会负责事务传播的各个方面，并全面管理跨越多个资源管理器的两阶段提交协议，但是它并不知道事务是应该提交还是中止。WCF 无法知晓对系统状态所做的改变是否满足一致性要求，即不能获知它们是否有意义。每个参与其中的服务都必须对事务的结果进行投票，并对提交或中止事务发表自己的意见。此外，WCF 并不知道何时开始两阶段提交协议，即事务结束的时间以及所有服务完成自己工作的时间。这也是服务（事实上，只是根服务）需要告诉 WCF 的。针对服务对事务的结果进行投票，WCF 提供了两种编程模型，包括声明式模型与显式模型。正如我们所看到的，事务的完成与结束和投票密切相关。

### 声明式投票

WCF 能自动代表服务投票以决定提交或中止事务。自动投票由 OperationBehavior 特性的 Boolean 型变量 TransactionAutoComplete 所控制：

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute, ...
{
    public bool TransactionAutoComplete
    {get;set;}
    // 更多成员
}
```

TransactionAutoComplete 变量的默认值为 true，因此以下两种定义是等效的：

```
[OperationBehavior(TransactionScopeRequired = true, TransactionAutoComplete = true)]
public void MyMethod(...)
{...}
```

```
[OperationBehavior (TransactionScopeRequired = true)]  
public void MyMethod(...)  
{...}
```

该属性值被设置为true时,如果操作不存在未被处理的异常,WCF将自动投票提交该事务;如果存在未被处理的异常,WCF将投票中止该事务。值得注意的是,尽管WCF为了中止事务必须捕捉异常,但是它也可以重新抛出异常,允许其沿着调用链继续传播。为了使用自动投票的功能,服务中的方法必须将TransactionScopeRequired设为true,这是因为自动投票功能只有在WCF为服务设置了环境事务的情况下才能生效。

当TransactionScopeRequired被设置为true时,我们必须避免捕获和处理异常,显式地让中止事务的操作为空:

```
// 避免  
[OperationBehavior(TransactionScopeRequired = true)]  
public void MyMethod(...)  
{  
    try  
    {  
        ...  
    }  
    catch  
    {  
        Transaction.Current.Rollback();  
    }  
}
```

原因在于我们的服务可能是某个跨越多个服务、机器和站点的更大的事务的一部分,该事务的其他参与者正在执行相关的操作,消耗了系统资源,然而由于我们的服务投票中止了事务,其他参与者却无从知晓,从而使得之前的操作徒劳无功。如果允许异常沿着调用链传播,它将中止在它的传播路径上的所有对象,并最终到达根服务或客户端,从而终止该事务。忽略异常可以提高系统的吞吐量及性能。如果我们想要捕获异常,以便加入一些诸如记录日志的本地处理,则必须重新抛出异常:

```
[OperationBehavior(TransactionScopeRequired = true)]  
public void MyMethod(...)  
{  
    try  
    {  
        ...  
    }  
    catch  
    {  
        /* 此处执行某些本地的处理 */  
        throw;  
    }  
}
```



## 显式投票

当 `TransactionAutoComplete` 被设置为 `false` 时,则需要显式投票。只有在 `TransactionScopeRequired` 为 `true` 时,才能将 `TransactionAutoComplete` 设置为 `false`。

当声明式投票被禁用时,WCF将默认投票中止所有事务,而不管是否发生异常。我们必须使用操作上下文中的 `SetTransactionComplete()` 方法显式地投票:

```
public sealed class OperationContext : ...
{
    public void SetTransactionComplete();
    // 更多成员
}
```

必须确保在调用 `SetTransactionComplete()` 之后没有执行任何工作,尤其是事务型的工作(译注3)。调用 `SetTransactionComplete()` 方法应该是服务操作返回结果前的最后一行代码:

```
[OperationBehavior(TransactionScopeRequired = true,
                    TransactionAutoComplete = false)]
public void MyMethod(...)
{
    /* 此处执行事务型工作: */
    OperationContext.Current.SetTransactionComplete();
}
```

如果我们在调用 `SetTransactionComplete()` 后试图执行任何事务型工作(包括访问 `Transaction.Current` 属性),WCF会抛出 `InvalidOperationException` 异常并中止该事务。

如果在 `SetTransactionComplete()` 之后没有执行任何工作,那么在调用 `SetTransactionComplete()` 之前发生的异常都会被忽略,WCF则根据默认情况中止该事务。因此,除非我们想对异常做一些本地处理,否则没有必要捕获它。就像声明式投票那样,如果我们要捕获异常,必须确保重新抛出该异常,以便加速事务的中止。

```
[OperationBehavior(TransactionScopeRequired = true,
                    TransactionAutoComplete = false)]
public void MyMethod(...)
{
    try
    {
        /* 此处执行事务型工作: */
    }
}
```

译注3: 如果在调用 `SetTransactionComplete()` 之后执行任何工作,则执行的工作会被忽略;如果执行的工作是事务型工作,则会抛出异常。



```
        OperationContext.Current.SetTransactionComplete();
    }
    catch
    {
        /* 执行错误处理 */
        throw;
    }
}
```

当投票依赖于通过事务获得的除异常和错误之外的信息时，适合使用显式投票。但是，对于大多数应用程序与服务而言，我们更偏向于声明式投票的简洁性。

---

**警告：** 不要轻易地将 `TransactionAutoComplete` 设为 `false`，事实上，只有在会话服务中才允许这样设置，因为它会严重地影响服务实例与某个事务的关联度。为了在事务过程中获取用于投票的信息，必须使用相同的事务和相同的实例。在本章后面的内容中，我们会看到设置 `TransactionAutoComplete` 为 `false` 的原因、时机与方式。

---

## 终止事务

事务何时结束是事务发起者的责任。考虑这样一个客户端，它既没有使用事务，也没有将它的事务传播给服务，同时，它却调用了将 `TransactionScopeRequired` 设置为 `true` 的服务操作。该服务操作会变成一个新事务的根。根服务可以调用其他服务并把事务传播给它们。一旦根操作完成了事务，该事务就会立即结束。通过将 `TransactionAutoComplete` 设置为 `true`，根操作可以以声明方式完成事务；也可以通过将该属性值设置为 `false`，然后显式地调用 `SetTransactionComplete()` 方法完成事务。这正是 `TransactionAutoComplete` 和 `SetTransactionComplete()` 如此命名的部分原因，因为它们所做的不仅仅是投票，它们还可以为一个根服务完成并终止事务。需要注意的是，任何被根操作调用的下游服务都只能对事务进行投票，而不能完成它。只有根操作既能投票也能完成事务。

当一个非服务型的客户端启动事务时，事务会随着客户端销毁事务对象而结束。我们将在显式事务编程一节中看到更多关于这方面的内容。

## 事务隔离性

总体而言，事务的隔离性越好，它们结果的一致性也就越好。最高级别的隔离性被称为串行化 (Serializable)，意思是一组并发事务的结果与它们串行运行所得到的结果完全相同。为了实现串行化，一个事务需要对它所访问的所有资源加锁，以隔离其他事务对该资源的访问。如果其他事务试图访问这些资源，它们会被阻塞直到原先的事务提交或

者中止后才能继续执行。隔离级别由System.Transactions中的枚举值IsolationLevel定义。

```
public enum IsolationLevel
{
    Unspecified,
    ReadUncommitted,
    ReadCommitted,
    RepeatableRead,
    Serializable,
    Chaos,    // 无论如何都不会隔离
    Snapshot // SQL 2005 支持的 ReadCommitted 的特殊形式
}
```

四种隔离级别 (ReadUncommitted、ReadCommitted、RepeatableRead 和 Serializable) 之间的差异在于它们对不同级别的读、写锁的使用方式。事务可以只在访问受资源管理器管理的数据的时候才持有一个锁,也可以一直持有该锁直到事务被提交或中止。前者能获得更好的吞吐量,后者则更有利于维持系统的一致性。这两种锁与两种操作 (读/写) 的组合构成了四种基本的隔离级别。此外,并非所有的资源管理器都支持四种级别的隔离性,并且当它们参与到事务中时,可能会选择比配置时更高的隔离等级。当多个事务访问相同资源时,除了串行化之外的其他隔离级别都会在系统的一致性方面受到某种程度的影响。

选择除串行化之外的隔离级别,通常用于强调读操作的系统,它需要对事务处理原理、事务自身的语义、相关的并发问题以及系统一致性的因果关系有深刻的理解。之所以需要对事务的隔离性进行配置,是因为高等级的隔离性是以牺牲整个系统的吞吐量为代价的,因为只要事务在进行中,它所涉及的资源管理器就必须保持读写锁,同时阻塞其他所有的事务。然而,在某些情况下我们愿意以降低隔离级别的方式,以系统的一致性换取好的吞吐量。以一个银行系统为例。需求之一是获取所有客户账户的总金额。尽管可以采用串行化隔离级别来执行该事务,但如果银行有数十万计的账户,这个计算过程将花费很长的时间。而且该事务很可能因为会超时而被中止,因为其中的某些账户可能同时正在被其他事务访问。但是账户的数目也不一定就全是负面的影响。根据统计,一般说来,假如允许事务在一个较低的隔离级别上运行,的确可能会在某些账户上得到错误的金额,但这些错误的金额很可能会彼此抵消,实际产生的误差将维持在一个可以被银行所接受的程度。

在 WCF 中,事务隔离性是一个服务行为,因此服务的所有方法都将使用相同配置的隔离级别。隔离级别是通过 ServiceBehavior 特性的 TransactionIsolationLevel 属性配置的:

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, ...
{
```

```
public IsolationLevel TransactionIsolationLevel
{get;set;}
// 更多成员
}
```

我们无法在宿主的配置文件中设置隔离级别。只有当服务含有至少一个将TransactionScopeRequired设为true的操作时,我们才能为该服务设置TransactionIsolationLevel 属性。

## 隔离性与事务流

TransactionIsolationLevel的默认值是 IsolationLevel.Unspecified, 因此以下两个声明是等效的:

```
class MyService : IMyContract
{...}

[ServiceBehavior(TransactionIsolationLevel = IsolationLevel.Unspecified)]
class MyService : IMyContract
{...}
```

当服务参与了客户端的事务并被配置为 IsolationLevel.Unspecified时, 该服务将使用客户端的事务隔离级别。

但是, 如果服务指定了一种不同于 IsolationLevel.Unspecified的隔离级别, 客户端就必须与该级别相匹配, 否则会在客户端抛出一个 FaultException 异常。

当服务是事务的根且服务被配置为 IsolationLevel.Unspecified时, WCF 会将隔离级别设置为 IsolationLevel.Serializable。如果根服务提供了一种不同于 IsolationLevel.Unspecified的级别, WCF 会使用该指定的级别。

## 事务超时

由于使用了事务隔离, 当一个事务试图访问被另一个事务拥有的资源管理器时, 引入隔离锁会增加产生死锁的几率。如果某个事务执行了很长时间, 则很有可能发生一个事务型死锁。为了应对这种情况, 当事务的执行时间长于预先设定的超时时间 (默认为60s) 时, 该事务将自动中止。一旦事务被中止, 任何将该事务向服务传播的尝试都会导致一个异常。即使没有异常发生, 该事务最终也会中止。参与事务的所有客户端与服务所要做的就是完成该事务。事务超时既可以以编程方式配置, 也可以以管理方式在配置文件中配置。

超时是一个服务行为属性, 服务的所有终结点中的所有操作都使用相同的超时值。我们可以通过将 ServiceBehaviorAttribute 的 String 类型变量 TransactionTimeout 设置为 time-span 格式, 以配置超时值:

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, ...
{
    public string TransactionTimeout
    {get;set;}
    // 更多成员
}
```

例如，使用下列语句配置一个 30s 的超时：

```
[ServiceBehavior(TransactionTimeout = "00:00:30")]
class MyService : ...
{...}
```

我们也可以通过在宿主的配置文件中设置事务的超时值，只需在配置文件中创建一个定制的行为配置节，并在服务配置节中引用它：

```
<services>
  <service name = "MyService" behaviorConfiguration = "ShortTransactionBehavior">
    ...
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name = "ShortTransactionBehavior"
      transactionTimeout = "00:00:30"
    />
  </serviceBehaviors>
</behaviors>
```

事务超时允许的最大值为 10min，即使指定了更大的值，超时值仍然为 10min。如果我们想要改写默认的 10min 超时最大值，比如设为 30min，请在 *machine.config* 中添加如下内容：

```
<configuration>
  <system.transactions>
    <machineSettings maxTimeout = "00:30:00"/>
  </system.transactions>
</configuration>
```

---

**注意：**在 *machine.config* 中设定任何值都会影响到在机器上的所有应用程序。

---

配置较长的超时值主要用于调试。例如，当我们试图通过单步调试自己的代码以隔离业务逻辑中的某个问题时，并不希望在我们解决问题的时候出现事务超时的情况。如果在其他情况下设置较长时间的超时值，我们需要特别小心，因为这通常意味着会出现事务死锁。

通常，我们会在两种情况下将超时值设置为小于默认值。第一种是在开发期间，当我们希望测试自己的应用程序处理事务中止的方式时。通过将超时值设置为一个较小的值（例如1毫秒），可以使得事务尽快失败，以便观察自己的错误处理代码。

第二种情况是当我们确信引入某个服务会过久占用资源，从而造成死锁的时候。此时，我们希望尽早的中止该事务，而不是等待默认的超时值到期。

### 事务流与超时

当事务传递给服务时，如果服务配置的超时值小于传入事务的超时值，则事务将转而采用服务的超时值，服务就会使用较短的超时值。如此设计的初衷在于解决刚刚讨论的所谓问题服务可能出现的死锁。如果服务配置的超时值大于传入事务的超时值，服务设置的超时值无效。

## 显式事务编程

目前所描述的事务编程模型只适用于声明方式的事务性服务。非服务型的客户端，非事务型的服务或被服务调用的普通.NET对象都无法利用它。针对这些情况，WCF借助了.NET2.0的System.Transactions命名空间中提供的事务型基础结构。此外，我们也可以在事务型服务中使用System.Transactions发挥一些高级功能，例如，事务事件、事务克隆、异步提交与手动事务。在本书作者撰写的MSDN白皮书《简介.NET Framework 2.0中的System.Transactions》（2005年4月发布，2005年12月更新）一文中，描述了System.Transactions的功能。以下内容摘录了文中对如何在WCF环境中使用System.Transactions的核心内容的描述。如要了解其余功能的详细介绍，可以参阅该白皮书的内容。

### TransactionScope 类

显式使用事务最常见的方法是通过 TransactionScope 类：

```
public class TransactionScope : IDisposable
{
    public TransactionScope();
    // 其余的构造函数

    public void Complete();
    public void Dispose();
}
```

一如名称所示，TransactionScope类用于将一段代码加入到一个事务的范围中，如例7-7中所示。

### 例 7-7: 使用 TransactionScope 类

```
using(TransactionScope scope = new TransactionScope())
{
    /* 在此执行业务型工作 */

    // 没有错误 - 提交事务
    scope.Complete();
}
```

TransactionScope的构造函数可以创建一个新的LTM事务,并通过设置Transaction.Current将它设置为环境事务,或加入一个已有的环境事务。TransactionScope是一个可以销毁的对象,如果它创建了一个新的事务,那么一旦Dispose()方法被调用(在例7-7中using语句的末尾),该事务就会结束。Dispose()方法还会将环境事务恢复到初态(在例7-7的情况中为空)。

最后,假如不是在一个using语句中使用TransactionScope对象,那么只有在事务超时并中止的情况下,它才会被垃圾回收。

### TransactionScope 投票

TransactionScope对象无法获知事务是否应该提交还是中止。因此,每个TransactionScope对象都有一个一致性校验位,它的默认值为false。我们可以通过调用Complete()方法将一致性校验位设为true。注意,我们只能调用一次Complete()方法,如果再次调用Complete()会引发一个InvalidOperationException异常。这样的设计是经过深思熟虑的,它可以提醒开发者在调用Complete()方法后,不要再编写与事务有关的代码。

如果事务结束后(因为调用了Dispose()方法或被垃圾回收),它的一致性校验位被设置为false,那么该事务将被中止。例如,下面的TransactionScope对象将中止其事务,因为一致性校验位一直维持在它的默认值从未改变过:

```
using(TransactionScope scope = new TransactionScope())
{
}
```

将调用Complete()方法作为事务范围的最后一个动作,我们就可以在错误发生时自动投票中止事务。原因在于在事务范围中抛出的任何异常都会跳过对Complete()方法的调用,using语句的finally语句会销毁TransactionScope对象,同时事务也会被中止。另一方面,假如我们确实调用了Complete()方法,并且在事务结束时一致性校验位被设置为true,如例7-7所示,则事务将试图提交。注意,在调用了Complete()方法以后,我们就不能访问环境事务了,否则会引发InvalidOperationException异常。一旦TransactionScope对象被销毁,我们又可以再次访问环境事务(通过Transaction.Current)。



事实上,在事务范围中调用 Complete() 方法并不能保证事务被提交。即使我们调用了 Complete() 方法,同时事务范围被销毁了,它所做的一切也仅仅是尝试提交该事务。这个尝试最终是成功还是失败将由两阶段提交协议决定,其中可能涉及多个资源与服务,而这些是我们在代码中无法了解的。因此,如果提交事务失败,Dispose() 方法将抛出 TransactionAbortedException 异常。我们可以捕获并处理该异常,或者向用户告警,如例 7-8 所示。

例 7-8: TransactionScope 与错误处理

```
try
{
    using(TransactionScope scope = new TransactionScope())
    {
        /* 在此执行事务型工作 */
        // 没有错误 - 提交事务
        scope.Complete();
    }
}
catch(TransactionAbortedException e)
{
    Trace.WriteLine(e.Message);
}
catch // 发生其余的异常
{
    Trace.WriteLine("Cannot complete transaction");
    throw;
}
```

## 事务流管理

事务范围可以直接或间接嵌套。在例 7-9 中,scope2 就嵌套在 scope1 的内部。

例 7-9: 直接事务范围嵌套

```
using(TransactionScope scope1 = new TransactionScope())
{
    using(TransactionScope scope2 = new TransactionScope())
    {
        scope2.Complete();
    }
    scope1.Complete();
}
```

在一个方法的 TransactionScope 内调用另一个方法,而被调用的方法同样使用了 TransactionScope,则属于 TransactionScope 的间接嵌套,如例 7-10 中的 RootMethod() 方法。



## 例 7-10: 间接事务范围嵌套

```

void RootMethod()
{
    using(TransactionScope scope = new TransactionScope())
    {
        /* Perform transactional work here */
        SomeMethod();
        scope.Complete();
    }
}

void SomeMethod()
{
    using(TransactionScope scope = new TransactionScope())
    {
        /* Perform transactional work here */
        scope.Complete();
    }
}

```

事务范围也可以嵌套在一个服务方法中, 如例 7-11 所示。此时, 服务方法可以为事务型的方法, 也可以不是。

## 例 7-11: 在服务方法中的事务范围嵌套

```

class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        using(TransactionScope scope = new TransactionScope())
        {
            scope.Complete();
        }
    }
}

```

如果事务范围创建了一个它要使用的新的事务, 则被称为根事务范围。一个事务范围是否能够成为一个根事务范围, 取决于事务范围的配置情况, 以及当前是否存在一个环境事务。一旦建立了一个根事务范围, 就存在它与所有它嵌套的事务范围或调用的下游服务之间隐式的关系。TransactionScope 类定义了多个重载版本的构造函数, 它们都接收一个 TransactionScopeOption 类型的枚举值:

```

public enum TransactionScopeOption
{
    Required,
    RequiresNew,
    Suppress
}

public class TransactionScope : IDisposable
{
    public TransactionScope(TransactionScopeOption scopeOption);
}

```

```
public TransactionScope(TransactionScopeOption scopeOption,  
                        TransactionOptions transactionOptions);  
public TransactionScope(TransactionScopeOption scopeOption,  
                        TimeSpan scopeTimeout);  
// 另外的构造函数与成员  
}
```

我们可以通过TransactionScopeOption的值决定一个事务范围是否参与某个事务,如果是,我们还可以进一步决定它是加入当前的环境事务,还是新建一个事务并成为该事务的根事务范围。

例如,如下的代码演示了如何在事务范围的构造函数中指定TransactionScopeOption的值:

```
using(TransactionScope scope  
        = new TransactionScope(TransactionScopeOption.Required))  
{...}
```

事务范围选项的默认值为TransactionScopeOption.Required,这意味着在无参构造函数中,会使用该值,因此,以下两个定义是等效的:

```
using(TransactionScope scope = new TransactionScope())  
{...}  
using(TransactionScope scope  
        = new TransactionScope(TransactionScopeOption.Required))  
{...}
```

在创建TransactionScope对象时,将决定它属于哪一个事务。一旦确定,该事务范围将一直属于那个事务。TransactionScope基于以下两个因素做出决定:是否存在一个环境事务,以及TransactionScopeOption参数的值。

一个TransactionScope对象有三种选择:

- 加入环境事务
- 成为一个新的根事务范围,即启动一个新的事务并使之成为自己事务范围内的新的环境事务
- 不参与任何事务

如果事务范围被设置为TransactionScopeOption.Required,并且存在一个环境事务,事务范围将加入该事务。反之,如果不存在环境事务,该事务范围将创建一个新的事务并成为根事务范围。

如果事务范围被设置为TransactionScopeOption.RequiresNew,那么它总会成为一个根事务范围。它将启动一个新事务,并且该事务将成为内部的新的环境事务。

如果范围被配置为 `TransactionScopeOption.Suppress`, 那么无论是否存在一个环境事务, 它都不会成为事务的一部分。一个被设置为 `TransactionScopeOption.Suppress` 的事务范围的环境事务总是为 `null`。

### 在嵌套事务范围内投票

尽管一个嵌套的事务范围可以加入其父事务范围的环境事务, 但我们仍然需要注意到两个事务范围对象具有不同的一致性校验位。在嵌套的子事务范围中调用 `Complete()` 方法对其父事务范围没有任何影响:

```
using(TransactionScope scope1 = new TransactionScope())
{
    using(TransactionScope scope2 = new TransactionScope())
    {
        scope2.Complete();
    }
    //scope1 的持久位仍然为 false
}
```

只有从根事务范围向下直到最内层的所有嵌套事务范围都投票提交事务的时候, 该事务才会被提交。此外, 只有根事务范围可以决定事务的生存周期。当一个 `TransactionScope` 对象加入一个环境事务的时候, 销毁该事务范围不会结束事务。事务仅在根事务范围被销毁或启动事务的服务方法返回的时候结束。

### TransactionScopeOption.Required

`TransactionScopeOption.Required` 不仅是最常用的事务范围选项值, 它还具有最好的解耦性。如果事务范围具有一个环境事务, 它将加入该环境事务以提高系统的一致性。即便无法加入, 该事务范围也能够为代码提供一个新的环境事务。如果使用 `TransactionScopeOption.Required`, 当事务范围为根事务范围, 或者事务范围刚刚加入环境事务时, 在 `TransactionScope` 内代码的表现必须保持一致, 操作也必须相同。在服务端, `TransactionScopeOption.Required` 常用于被服务调用的非服务下游对象类, 如例 7-12 所示。

例 7-12: 在一个下游类中使用 `TransactionScopeOption.Required`

```
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        MyClass obj = new MyClass();
        obj.SomeMethod();
    }
}
```

```
class MyClass
{
    public void SomeMethod()
    {
        using(TransactionScope scope = new TransactionScope())
        {
            // 执行某些工作
            scope.Complete();
        }
    }
}
```

尽管服务自身可以直接使用 `TransactionScopeOption.Required`, 但这种做法并没有实际意义:

```
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        // 只有一个事务
        using(TransactionScope scope = new TransactionScope())
        {
            // 执行某些工作
            scope.Complete();
        }
    }
}
```

原因显而易见: 服务只要将 `TransactionScopeRequired` 设置为 `true`, 就可以要求 WCF 将一个服务操作包含在一个事务范围中 (这同样也是该属性名称的来源)。注意, 尽管服务可以使用声明式投票, 但是下级范围的 (或直接嵌套的) 事务范围必须显式地调用 `Complete()` 函数以实现事务的提交。

服务方法使用显式投票时遵循同样的原则:

```
[OperationBehavior(TransactionScopeRequired = true,
                    TransactionAutoComplete = false)]
public void MyMethod(...)
{
    using(TransactionScope scope = new TransactionScope())
    {
        // 执行某些工作
        scope.Complete();
    }
    /* 在此处执行事务型工作: */
    OperationContext.Current.SetTransactionComplete();
}
```

简言之, 在一个服务调用内的嵌套事务范围中, 并且该事务范围被设置为 `TransactionScopeRequired`, 如果投票中止事务, 不管是否出现异常, 或者使用声明式投票 (通过

TransactionAutoComplete), 亦或显式投票 (通过 SetTransactionComplete() 方法), 都会中止服务事务 (译注 4)。

### TransactionScopeOption.RequiresNew

当我们想要在环境事务的范围外执行事务型工作时, 将事务范围配置为 TransactionScopeOption.RequiresNew 就显得十分有用。例如, 当我们想要执行一些日志记录或审核操作时, 或当我们想要向订阅事件的用户发布事件时, 环境事务的提交或中止对我们的工作都不会产生影响:

```
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        // 两个独立的事务
        using(TransactionScope scope =
            new TransactionScope(TransactionScopeOption.RequiresNew))
        {
            // 执行某些工作
            scope.Complete();
        }
    }
}
```

注意, 我们必须在事务范围内调用 Complete () 方法以提交新事务。我们也可以考虑在一个 try、catch 语句中包含一个事务范围, 并将其设置为 TransactionScopeOption.RequiresNew, 从而使得它与服务的环境事务隔离。

在使用 TransactionScopeOption.RequiresNew 时, 我们需要特别当心, 必须确保两个事务 (环境事务以及事务范围创建的事务) 不会因为其中一个事务提交、另一个事务中止而影响系统的一致性。

### TransactionScopeOption.Suppress

当配置了 TransactionScopeOption.Suppress 的代码使得操作能够更好的执行, 并且在操作失败时, 不会中止环境事务, 那么 TransactionScopeOption.Suppress 就是最佳选择, 这一设置同时适用于客户端与服务。TransactionScopeOption.Suppress 允许我们在一个事务范围或事务型服务操作中包含一段不使用事务的代码, 如例 7-13 所示。

---

译注 4: 由于嵌套的事务加入到服务的环境事务中, 因此两者是密切相关的。终止嵌套事务的同时也会终止服务的事务, 这与 RequiresNew 不同。

**例 7-13: 使用 TransactionScopeOption.Suppress**

```
[OperationBehavior(TransactionScopeRequired = true)]
public void MyMethod(...)
{
    try
    {
        // 非事务型节的开始
        using(TransactionScope scope = new
            TransactionScope(TransactionScopeOption.Suppress))
        {
            // 此处执行非事务型工作
        } // 此处恢复环境事务
    }
    catch
    {}
}
```

注意, 在例 7-13 中不需要在被标记了 Suppress 的事务范围内调用 Complete() 函数。TransactionScopeOption.Suppress 的另一个用途是, 我们可以在它所包含的代码段中提供一些定制行为, 例如需要编程实现事务支持, 或者手动地将资源登记到事务中。

这即是说, 当我们将非事务型范围与事务型范围或服务方法混合使用时, 必须谨慎小心, 避免它们损害系统的隔离性和一致性, 因为在标记为 Suppress 的事务范围内, 对系统状态所做的改变不会随着包含它的环境事务一起回滚。此外, 非事务型范围可能会出现错误, 但这些错误不应该影响环境事务的结果。这就是在例 7-13 中为什么要将被标记为 Suppress 的事务范围放入到 try、catch 语句中的原因。

---

**警告:** 不要在一个标记为 Suppress 的事务范围内调用配置为 Client 事务的服务 (通常为强制性事务流), 因为这种调用必定会失败。

---

**TransactionScope 超时**

如果在事务范围内的代码运行了很长时间, 则意味着可能发生了事务死锁。为了应对这种情况, 当事务执行超过了某个预定的超时值 (默认为 60s) 时, 它将自动中止。我们可以在应用程序的配置文件里设置默认的超时值。例如, 把下面这段代码加入到配置文件中, 就可以将默认超时值设置为 30s:

```
<system.transactions>
  <defaultSettings timeout = "00:00:30"/>
</system.transactions>
```

在应用程序的配置文件里设置新的默认超时值, 会影响到在该应用程序内被所有客户端和服务使用的所有事务范围。我们也可以为某个特定的事务范围专门配置一个超时值。

TransactionScope构造函数的某些重载版本接收一个TimeSpan类型的值,该值可以用来控制事务的超时值,例如:

```
public TransactionScope(TransactionScopeOption scopeOption,
                        TimeSpan scopeTimeout);
```

要想指定一个不同于默认的 60s 的超时值,只需简单地传入期望值:

```
TimeSpan timeout = TimeSpan.FromSeconds(30);
using(TransactionScope scope
        = new TransactionScope(TransactionScopeOption.Required, timeout))
{ ... }
```

如果一个TransactionScope在加入环境事务时指定了一个比环境事务更短的超时值,那么环境事务会转而使用这个更短的时间。环境事务必须在该值设定的期间内结束,否则会自动被中止。如果 TransactionScope 的超时值比环境事务的要长,则无效。

### TransactionScope 的隔离等级

如果事务范围是一个根事务范围,该事务在运行时会默认采用串行化的隔离级别。TransactionScope 构造函数的某些重载版本可以接收一个 TransactionOptions 类型的结构,该结构的定义如下:

```
public struct TransactionOptions
{
    public IsolationLevel IsolationLevel
    { get; set; }
    public TimeSpan Timeout
    { get; set; }
    // 其他成员
}
```

尽管我们可以使用 TransactionOptions 结构中的 Timeout 属性指定一个超时值,但 TransactionOptions 主要还是用于设定事务的隔离级别。我们可以为 TransactionOptions 中的 IsolationLevel 属性赋予一个前面提及的 IsolationLevel 类型的枚举值:

```
TransactionOptions options = new TransactionOptions();
options.IsolationLevel = IsolationLevel.ReadCommitted;
options.Timeout = TransactionManager.DefaultTimeout;

using(TransactionScope scope
        = new TransactionScope(TransactionScopeOption.Required, options))
{ ... }
```

一个事务范围加入环境事务时,必须被设置为与环境事务完全相同的隔离级别,否则会抛出一个 ArgumentException 异常。



## 非服务型客户端

尽管服务可以使用 `TransactionScope`, 但目前为止事务范围主要用于非服务型客户端。一个非服务型的客户端要想将多个服务调用组成一个单独事务, 只能通过 `TransactionScope` 这一途径, 如图 7-7 所示。

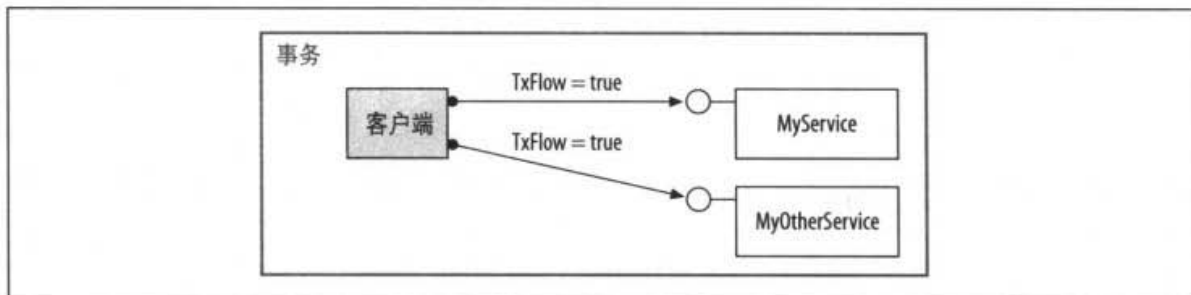


图 7-7: 一个非服务型客户端使用一个单独事务调用多个服务

创建根事务范围的选项允许客户端将自己的事务传播给服务, 并根据服务的综合结果管理并提交该事务, 如例 7-14 所示。

例 7-14: 使用 `TransactionScope` 在一个事务中调用多个服务

```
////////// 服务端 //////////////////////////////////////
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void MyMethod(...);
}
[ServiceContract]
interface IMyOtherContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    void MyOtherMethod(...);
}
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    { ... }
}
class MyOtherService : IMyOtherContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyOtherMethod(...)
    { ... }
}
```

```
////////// 客户端 ////////////
using(TransactionScope scope = new TransactionScope())
{
    MyContractClient proxy1 = new MyContractClient();
    proxy1.MyMethod(...);
    proxy1.Close();

    MyOtherContractClient proxy2 = new MyOtherContractClient();
    proxy2.MyOtherMethod(...);
    proxy2.Close();

    scope.Complete();
}

// 可以联合在一个 using 语句块中:
using(MyContractClient proxy3 = new MyContractClient())
using(MyOtherContractClient proxy4 = new MyOtherContractClient())
using(TransactionScope scope = new TransactionScope())
{
    proxy3.MyMethod(...);
    proxy4.MyOtherMethod(...);
    scope.Complete();
}
```

## 服务状态管理

事务编程的目的只是为了能够在系统出错的时候，将系统的状态恢复到出错前的状态，因此，我们需要始终将系统维持在一致的状态下。系统的状态包括在事务中涉及的所有资源以及客户端及服务在内存中的实例。WCF资源管理器除了具有诸如自动登记以及参与两阶段提交协议的优势之外，使用资源管理器的一个最基本也最明显的优势在于，一旦事务中止，在事务执行期间对其状态所做的一切改变都将自动回滚。但是，对于那些将参与事务的服务保存在内存中的实例成员和静态成员而言，却并非如此。因此，系统在事务完成后可能处于不一致的状态。如果考虑到服务参与的事务还可能跨越多个服务、机器甚至站点，问题就变得更加复杂。即使服务实例没有遇到错误并投票提交事务，该事务也可能最终被服务边界外的其他参与方中止。但是，如果服务只是简单地把它的状态保存在内存中，它将如何获知事务的结果，从而手动回滚对其状态所做的改变？

针对服务实例状态管理的问题，一个解决方案是将服务作为一个与状态相关的服务进行开发，并能够主动管理它的状态。正如在第4章所阐释的那样，一个状态相关的服务不同于无状态的服务。如果服务是无状态的，那么实例状态的回滚就不存在任何问题。实际上，只要事务在进行当中，服务实例就可以在内存中保存状态。为了维持系统的一致性，在两个不同的事务之间，服务应该将它的状态保存在一个资源管理器中。该状态资源管理器可能与事务执行期间被业务逻辑访问的资源相关，也可能无关。在事务的开始阶段，服务应该从资源中取回自己的状态，同时将资源登记到事务中。在事务的结束阶段，服务应该保存自己的状态，并返回给资源管理器。这一技术的好处在于它提供了状

态的自动恢复。对实例状态所做的任何改变都将作为该事务的一部分被提交或回滚。如果事务提交了，服务下一次获取的状态将会是新的状态；如果事务中止了，那么它将会获得自己在执行事务前的状态。不管是哪种情况，服务都保持一致的状态下，以待下一个新事务的访问。为了强制服务实例以这种方式将它保存在内存中的状态完全清除，在默认情况下，一旦事务完成，WCF将销毁该服务的实例，以保证在内存中不存在可能危害到系统一致性的残留物。

## 事务边界

编写状态相关的事务型服务时还会遇到两个问题。第一个问题是服务如何知道事务开始与结束的时间？只有知道准确的时间才能获取与保存自己的状态。服务可能是某个更大的跨越多个服务和机器的事务的一部分。事务在服务调用过程中随时都可能结束，那么应该由谁来通知服务保存其状态？第二个问题则关于隔离，不同的客户端可能使用不同的事务并发地访问该服务，那么，服务应该如何将一个事务对自己状态所做的改变与另一个事务相隔离？服务禁止跨事务的调用，因为这么做会损害隔离性。假如其他事务访问了它的状态，并根据其值进行操作，那么一旦原始的事务中止且回滚，该事务将会被脏状态（Foul State）所污染。

解决这两个问题的方法是统一服务的方法边界与事务边界。服务应该在每个方法的开始处读取它的状态，然后在每个方法的结束处将自己的状态保存到资源管理器中。当事务在方法调用之间结束时，通过这种方式，服务将确保其状态随着事务的成功与否保存或回滚。因为服务把方法的边界与事务的边界等同起来，因此服务的实例同样必须在每个方法结束处对该事务的结果进行投票。从服务的角度来看，方法返回后事务就立即结束。这也是为什么使用TransactionAutoComplete属性而不是类似TransactionAutoVote的根本原因。因为就服务而言，此时的事务已经完成了。如果服务同时也是事务的根，那么方法的结束实际上也就代表了事务的终结。

此外，在每个方法调用中读取并保存资源管理器中的状态，还可以解决事务的隔离性问题，因为服务只需要使用资源管理器隔离并发事务对状态的访问。

## 状态标示符

因为可能会有相同服务类型的多个实例访问相同的资源管理器，每个操作都必须包含一些参数，以供服务实例在资源管理器中找到自己的状态并绑定。最好的方法是让每个操作包含某些键，作为标识状态的参数，该参数被称为状态标示符（State Identifier）。客户端必须提供状态标识符。通常，状态标识符就是账号、订单号之类的数据。例如，客户端创建了一个新的事务相关的订单处理对象，而在调用每一个方法时，客户端除了提供其他参数之外，还必须提供一个订单号作为输入参数。

例 7-15 演示了用于实现事务型单调服务的一个模板。

例 7-15: 实现一个事务型服务

```
[DataContract]
class Param
{...}

[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod();
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract, IDisposable
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(Param stateIdentifier)
    {
        GetState(stateIdentifier);
        DoWork();
        SaveState(stateIdentifier);
    }
    void GetState(Param stateIdentifier)
    {...}
    void DoWork()
    {...}
    void SaveState(Param stateIdentifier)
    {...}
    public void Dispose()
    {...}
}
```

MyMethod()的方法签名中包含了一个Param类型(在本例中,它是一个伪类型)的状态标识符参数,GetState()辅助方法通过它从资源管理器中获取服务的状态。然后,服务实例使用DoWork()辅助方法执行相关工作。紧接着,服务实例为SaveState()方法指定自己的状态标识符,从而将自己的状态存回资源管理器。

---

**注意:** 在单调服务中,用来储存服务状态的资源管理器,还可以是被静态成员变量访问的易失型资源管理器。

---

注意,并非所有的服务实例状态都可以保存到资源管理器中。如果状态中包含对其他对象的引用,GetState()方法就需要创建出这些对象,而SaveState()(或Dispose())方法)则负责销毁它们。

## 实例管理与事务

既然在每次方法调用时，服务实例获取以及保存自己的状态都是如此艰难，那么为何还要等到事务结束才销毁该对象呢？从本质上讲，单调服务是最适合于事务型 WCF 服务的编程模型。此外，状态相关的事务对象与单调对象的行为要求也是一样的，它们都会在方法边界上取出并保存自己的状态（比较例4-3与例7-15）。尽管单调实例模式最适合于事务，但 WCF 还是支持会话服务，甚至是单例服务，只不过它们的编程模型更加复杂。

### 单调事务型服务

就调用单调服务而言，事务编程几乎是附带的。每次调用服务都会得到一个新的实例，而该调用可能与前一个调用处在相同或不同的事务中（参见图 7-8）。

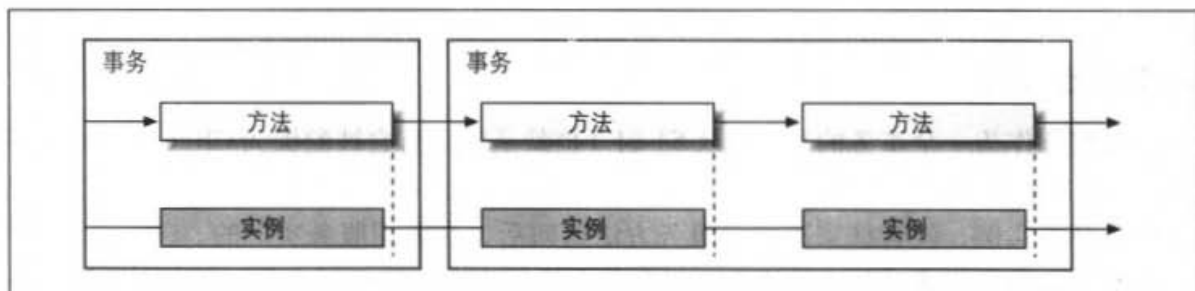


图 7-8：单调服务与事务

无论是否与事务相关，服务在每次调用时都会从资源管理器中取出并保存自己的状态，从而保证方法总是在前一个事务的一致性状态上进行操作，或是在当前正在运行的事务的临时的但具有良好隔离性的状态上进行操作。一个单调服务必须在每次方法调用中投票并完成它的事务。事实上，单调服务一般情况下都必须使用自动完成（AutoComplete）选项（将 TransactionAutoComplete 设置为 true，即它的默认值）。

从客户端的角度来看，相同的服务代理可以参与到一个或多个事务中。例如，在下面代码段中，每个调用都将处于不同的事务中：

```
MyContractClient proxy = new MyContractClient();

using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod(...);
    scope.Complete();
}
using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod(...);
}
```

```
        scope.Complete();  
    }  
  
    proxy.Close();
```

客户端也可以在相同事务中重复使用相同的代理，甚至可以独立于任何事务关闭代理：

```
MyContractClient proxy = new MyContractClient();  
using(TransactionScope scope = new TransactionScope())  
{  
    proxy.MyMethod(...);  
    proxy.MyMethod(...);  
    scope.Complete();  
}  
proxy.Close();
```

---

**注意：**调用单调服务的 `Dispose()` 方法时，不存在环境事务。

---

## 事务生命周期

当单调服务作为一个事务的根（译注5）时（也就是说，当它被配置为 Client/Service 事务，并且不存在客户端事务的时候，或者当它被配置为 Service 事务的时候），一旦停止了该服务的实例，就会结束事务。服务方法返回后，在调用服务实例的 `Dispose()` 方法之前，WCF 会完成并结束该事务。而当客户端作为事务的根时（或者当客户端的事务流向服务，同时服务加入该事务时），事务的结束则以客户端的事务为准。

## 会话事务型服务

WCF 默认的事务配置会将任何服务都转换为单调服务，而不会考虑它所采取的实例模式。这是为了满足服务状态管理中的一致性要求，它需要服务是状态相关的。然而，采用一个状态相关的服务会首先否定那些针对会话服务的需求。不过，WCF 的确允许我们可以在一个事务型服务中保留会话的语义。一个会话事务型服务的实例可以被多个事务访问，该实例也可以与某个特定的事务建立一个关联，在这种情况下，直到该事务完成，其他事务都无法访问该实例。但是，正如我们所见，这种支持会给编程模型带来不相称的复杂性。

## 释放服务实例

任何非单调模式的事务型服务的生命周期都是由 `ServiceBehavior` 特性的 `Boolean` 属性 `ReleaseServiceInstanceOnTransactionComplete` 控制的：

---

译注5：此时客户端没有环境事务。



```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, ...
{
    public bool ReleaseServiceInstanceOnTransactionComplete
    {get;set;}
    // 更多成员
}
```

当ReleaseServiceInstanceOnTransactionComplete被设置为true(默认值)时,一旦实例完成事务,就会立即销毁服务实例。注意,释放发生在实例完成事务的瞬间,而不一定要等到事务真正完成(这可能是在很久以后)。当ReleaseServiceInstanceOnTransactionComplete的值为true时,实例有两种完成事务并被释放的方式:如果方法将TransactionAutoComplete设置为true,那么在方法的边界处就会完成事务并释放实例;如果方法将TransactionAutoComplete设置为false时,则在调用SetTransactionComplete()方法的时候,完成事务并释放实例。

ReleaseServiceInstanceOnTransactionComplete会与Service以及Operation的另外两个行为属性产生交互。第一,只有当Service中包含至少一个将TransactionScopeRequired设置为true的操作时,才能设置ReleaseServiceInstanceOnTransactionComplete的值(不管是true还是false)。这一点会在Service加载时由ReleaseServiceInstanceOnTransactionComplete属性的set访问器进行验证。

例如,如下代码就是一个合法的配置:

```
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = true)]
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }

    [OperationBehavior(...)]
    public void MyOtherMethod()
    { ... }
}
```

这一约束条件意味着即使ReleaseServiceInstanceOnTransactionComplete的默认值为true,下面两个定义在语义上也是不等效的,因为第二个在Service加载时会抛出一个异常:

```
class MyService : IMyContract
{
    public void MyMethod()
    { ... }
}
```



```
// 无效定义
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = true)]
class MyService : IMyContract
{
    public void MyMethod()
    { ... }
}
```

当我们在多线程并发访问服务实例时使用ReleaseServiceInstanceOnTransactionComplete, 则会遇到第二个约束条件。

并发管理是下一章的主题。现在我们只需要知道ServiceBehavior特性的ConcurrencyMode 属性可以控制对服务实例的并发访问。

```
public enum ConcurrencyMode
{
    Single,
    Reentrant,
    Multiple
}

[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : ...
{
    public ConcurrencyMode ConcurrencyMode
    { get; set; }
    // 更多成员
}
```

ConcurrencyMode 的默认值为 ConcurrencyMode.Single。

WCF 会在加载服务时检验服务是否符合如下要求：当 ReleaseServiceInstanceOnTransactionComplete 被设置为 true (默认或显式地设置) 时, 服务应该至少包含一个操作, 它的 TransactionScopeRequired 被设置为 true。同时, 服务的并发模式必须被设置为 ConcurrencyMode.Single。

例如, 给定如下的契约:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod();

    [OperationContract]
    [TransactionFlow(...)]
    void MyOtherMethod();
}
```

下面两个定义是合法且等效的：

```
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }

    public void MyOtherMethod()
    { ... }
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single,
    ReleaseServiceInstanceOnTransactionComplete = true)]
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }

    public void MyOtherMethod()
    { ... }
}
```

在如下定义中，尽管 `ReleaseServiceInstanceOnTransactionComplete` 被设置为 `true`，但由于没有方法需要一个事务范围，所以它也是合法的：

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyService : IMyContract
{
    public void MyMethod()
    { ... }

    public void MyOtherMethod()
    { ... }
}
```

相反，下面的定义则是非法的，因为此时 `ReleaseServiceInstanceOnTransactionComplete` 的值为 `true`，同时又有一个方法要求事务范围，而服务的并发模式并非 `ConcurrencyMode.Single`。

```
// 无效配置：
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }

    public void MyOtherMethod()
    { ... }
}
```

注意：并发的约束条件适用于所有的实例模式。

`ReleaseServiceInstanceOnTransactionComplete`属性可以在客户端与服务之间启用一个事务型的会话交互。它的默认值为`true`，这意味着一旦服务实例完成事务（或者是声明式的或者是显式的），在方法返回时就会停止服务实例，这与单调服务相似。

例如，例 7-16 中的服务具有与单调服务相似的表现。

例 7-16：表现如单调服务的会话事务型服务

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod();
}
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }
}
```

每当客户端调用 `MyMethod()` 方法时，客户端就会获得一个新的服务实例。新的客户端调用也可能会使用一个新的事务，服务实例与任何事务都不存在关联。服务实例与事务之间的关系正如图 7-8 所示（译注 6）。

## 禁止释放服务实例

显然，例 7-16 的配置毫无价值。服务若要展现会话模式，可以将 `ReleaseServiceInstanceOnTransactionComplete` 设置为 `false`，如例 7-17 所示。

例 7-17：会话事务型服务

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod();
}
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = false)]
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
```

译注 6：由于服务实例与事务之间没有关联，实例被停止时，事务不一定会结束。

```
public void MyMethod()  
{...}  
}
```

当 `ReleaseServiceInstanceOnTransactionComplete` 的值为 `false` 时，服务实例不会在事务完成后立即被销毁，如图 7-9 所示。

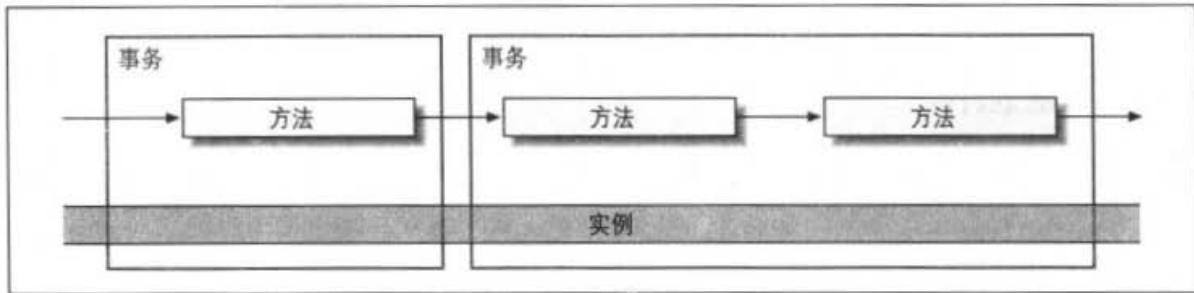


图 7-9：支持会话的事务型实例与事务

例如，图 7-9 的交互可能是如下客户端代码的结果，所有的调用都针对相同的服务实例。

```
MyContractClient proxy = new MyContractClient();  
using(TransactionScope scope = new TransactionScope())  
{  
    proxy.MyMethod();  
    scope.Complete();  
}  
  
using(TransactionScope scope = new TransactionScope())  
{  
    proxy.MyMethod();  
    proxy.MyMethod();  
    scope.Complete();  
}  
proxy.Close();
```

## 状态相关的会话服务

当 `ReleaseServiceInstanceOnTransactionComplete` 的值为 `false` 时，WCF 在处理事务时将不再询问服务实例的状态，而是交由服务的开发者自己管理。显然，我们必须采取一些措施来监控事务，并在事务中止的情况下，回滚对实例状态所做的任何改变。会话服务必须对方法的边界与事务的边界一视同仁，因为每个方法都可能处在不同的事务中。这里存在两种可能的编程模型。第一种是与状态相关的，但却使用会话 ID 作为状态标识符。在每个方法的开始部分，服务都会使用会话 ID 作为一个键，从资源管理器中获取它的状态，而在每个方法的结束部分，该服务实例又会把状态保存回资源管理器中，如例 7-18 所示。

## 例 7-18: 状态相关的事务型会话服务

```
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = false)]
class MyService : IMyContract, IDisposable
{
    readonly string m_StateIdentifier;

    public MyService()
    {
        InitializeState();
        m_StateIdentifier = OperationContext.Current.SessionId;
        SaveState();
    }
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    {
        GetState();
        DoWork();
        SaveState();
    }
    public void Dispose()
    {
        RemoveState();
    }

    // 辅助方法

    void InitializeState()
    { ... }
    void GetState()
    {
        // 使用 m_StateIdentifier 获取状态
        ...
    }
    void DoWork()
    { ... }
    void SaveState()
    {
        // 使用 m_StateIdentifier 保存状态
        ...
    }
    void RemoveState()
    {
        // 使用 m_StateIdentifier 将状态从 RM 中移除
        ...
    }
}
```

在例 7-18 中，构造函数首先初始化对象的状态，然后把该状态保存到一个资源管理器中，从而使得任何方法都能够获取该实例的状态。注意，由于客户端不需要显式地传递一个状态标识符（译注 7），会话服务对象会造成一种客户端与服务之间的交互是有状态、有会话的假象。在这种情况下，服务的实现必须严格遵循规定，它必须在调用每个

译注 7： 因为此时的状态标识符即为会话 ID。

操作时都取出并保存状态。当会话结束时,服务实例会在Dispose()方法中把自己的状态从资源管理器里清除掉。

## 易失性资源管理器

在文章《Volatile Resource Managers in .NET Bring Transactions to the Common Type》(发表在2005年5月的MSDN杂志)中,作者提出了一种实现通用型易失性资源管理器的方法,该资源管理器名为Transactional<T>。

```
public class Transactional<T> : ...
{
    public Transactional(T value);
    public Transactional();
    public T Value
    {get;set;}
    /* Conversion operators to and from T */
}
```

通过为Transactional<T>指定任意一个可序列化的类型参数(如int型或string型),我们可以将该类型转化为一个功能完备的易失型资源管理器,它可以将指定类型的对象自动登记到环境事务中,参与两阶段提交协议,并使用原先基于事务的锁,将当前对类型实例所做的修改与其他事务隔离。

例如,在如下的代码段中,事务范围没有完成事务。因此,事务将被中止,而number和city的值也将恢复到它们执行事务前的状态。

```
Transactional<int> number = new Transactional<int>(3);
Transactional<string> city
    = new Transactional<string>("New York");

using(TransactionScope scope = new TransactionScope())
{
    city.Value = "London";
    number.Value = 4;
    number.Value++;
    Debug.Assert(number.Value == 5);
    Debug.Assert(number == 5);
}
Debug.Assert(number == 3);
Debug.Assert(city == "New York");
```

除了Transactional<T>之外,我们还提供了一个名为TransactionalArray<T>的事务型数组,以及针对System.Collections.Generic命名空间下的所有集合的事务版本,例如TransactionalDictionary<K,T>和TransactionalList<T>。易失性资源管理器的实现与WCF没有任何关系,因此在本书中并没有将它包含在内。尽管如此,这个实现充分运用了C# 2.0、System.Transactions以及.NET系统编程的高级功能,因此它本身也具有相当的价值。

### 有状态的会话服务 (译注 8)

第二种编程模型是目前常用的编程模型,也就是为服务成员使用易失性资源管理器(参见补充资料“易失性资源管理器”),如例 7-19 所示。

例 7-19: 使用易失性资源管理器实现有状态的会话事务型服务

```
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = false)]
class MyService : IMyContract
{
    Transactional<string> m_Text =
        new Transactional<string>("Some initial value");

    TransactionalArray<int> m_Numbers = new TransactionalArray<int>(3);

    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    {
        m_Text.Value = "This value will roll back if the transaction aborts";

        // 如果事务被中止,下列内容会回滚
        m_Numbers[0] = 11;
        m_Numbers[1] = 22;
        m_Numbers[2] = 33;
    }
}
```

例 7-19 使用了我开发的 `Transactional<T>` 与 `TransactionalArray<T>` 易失性资源管理器,在本书源代码中可以获得它们的实现。通过使用泛型, `Transactional<T>` 可以随意选取一个可序列化的类型,并提供针对它的事务性访问。因此,会话服务完全可以将 `ReleaseServiceInstanceOnTransactionComplete` 设置为 `false`, 成员的访问也没有任何限制。易失性资源管理器的使用能够支持有状态的编程模型,服务实例可以便捷地访问其状态,就像没有使用事务一样。易失性资源管理器在事务中会自动登记资源,并将该事务与其他所有的事务隔离开来。对状态所做的任何改变都将随着事务一起提交或回滚。

### 事务生命周期

当会话服务作为事务的根时,一旦服务完成事务,即方法返回时,会立即结束该事务。当客户端作为事务的根时(或当一个事务流向服务的时候),服务事务会随着客户端事务的结束而结束。如果会话服务实现了 `IDisposable` 接口,则无论事务的根是谁, `Dispose()` 方法中都不会包含任何事务。

---

译注 8: 有状态与状态相关的区别在于使用的资源管理器。前者存储在持久性资源中,后者则存储在易失性资源中。



## 并发事务

因为一个会话服务可以使用相同的服务实例服务于多个客户端调用,所以它同样可以支持多个并发事务。根据例 7-17 的服务定义,例 7-20 演示了用于在同一个服务实例上启动多个并发事务的部分客户端代码。scope2 将使用一个与 scope1 不同的事务,但是它们将在同一个会话中访问同一个服务实例。

### 例 7-20: 启动并发事务

```
using(TransactionScope scope1 = new TransactionScope())
{
    MyContractClient proxy = new MyContractClient();
    proxy.MyMethod();

    using(TransactionScope scope2
           = new TransactionScope(TransactionScopeOption.RequiresNew))
    {
        proxy.MyMethod();
        scope2.Complete();
    }
    proxy.MyMethod();

    proxy.Close();
    scope1.Complete();
}
```

例 7-20 中产生的事务如图 7-10 所示。

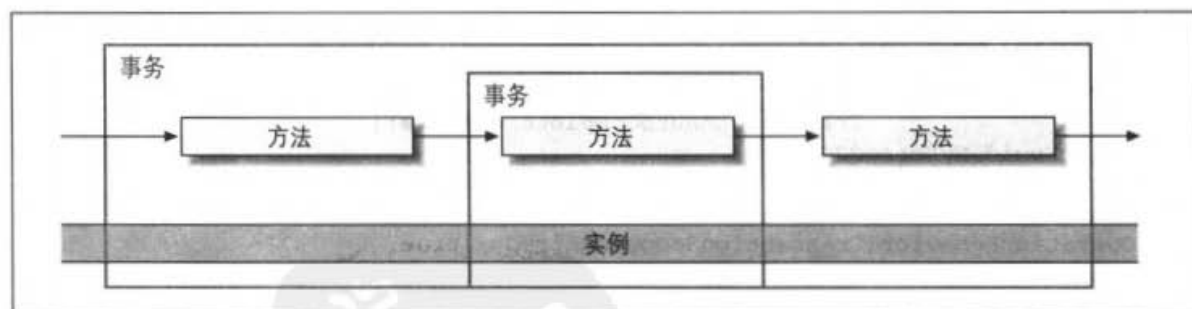


图 7-10: 并发事务

**警告:** 诸如例 7-20 中的代码,在服务访问的底层资源上几乎必然会造成事务死锁。第一个事务首先会获得资源锁,而第二个事务又必须等待它释放资源锁,与此同时,第一个事务会等待第二个事务完成之后,才能够继续执行。

## 会话结束时完成事务

WCF 为事务型的会话服务提供了另外一种编程模型,该模型与 ReleaseService-

InstanceOnTransactionComplete完全无关。当会话的生存期仅仅是某个事务生存期的一部分时,也就是说整个会话属于某个单独的事务时,可以采用该模型。它的主要思想是服务不应该在会话过程中完成事务,而应等到会话结束再完成事务,因为过早的完成事务将导致 WCF 释放服务实例。为了避免完成事务,会话服务可以将 TransactionAutoComplete 设置为 false,如例 7-21 所示。

例 7-21: 将 TransactionAutoComplete 设置为 false

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod1();

    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod2();

    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod3();
}
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void MyMethod1()
    { ... }

    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void MyMethod2()
    { ... }

    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void MyMethod3()
    { ... }
}
```

注意,只有会话服务可以将 TransactionAutoComplete 设置为 false,在服务加载时会检验这一约束。例 7-21 的问题在于由于服务没有投票提交事务,服务参与的事务总是会被中止。如果会话的生存期被完全地包含在一个单独的事务中,那么一旦会话结束服务,就应该立即投票。为此,ServiceBehavior 特性提供了 Boolean 属性 TransactionAutoCompleteOnSessionClose,该属性定义如下:

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, ...
{
```

```
public bool TransactionAutoCompleteOnSessionClose  
{get;set;}  
// 更多成员  
}
```

TransactionAutoCompleteOnSessionClose的默认值为false,然而当它被设置为true时,它将自动完成在会话中所有未完成的方法。如果在会话期间没有发生异常,而TransactionAutoCompleteOnSessionClose又被设置为true,服务将在会话结束时投票提交事务。以下代码对例7-21做了改进,图7-11则演示了它所产生的实例与会话:

```
[ServiceBehavior(TransactionAutoCompleteOnSessionClose = true)]  
class MyService : IMyContract  
{...}
```

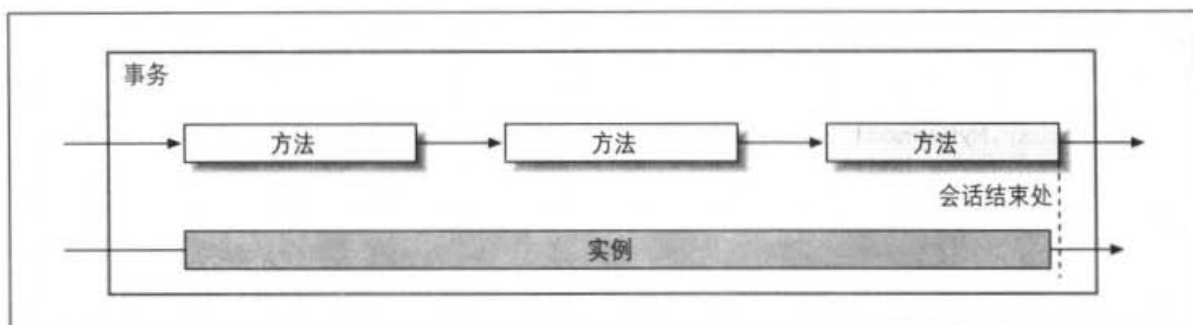


图 7-11: 将 TransactionAutoCompleteOnSessionClose 设置为 true

在会话期间,实例可以维护并访问保存在成员变量中的状态,无需使用状态相关的服务或易失性资源管理器。

**警告:** 当服务加入客户端事务并依靠会话结束时的自动完成功能完成事务时,服务必须避免在 Dispose() 方法中做过多的处理,我们甚至不应该实现 IDisposable 接口。原因参见如下的竞态条件 (Race Condition)。回顾第 4 章中的内容,在会话结束时,Dispose() 方法是以异步方式调用的,而会话结束时的事务自动完成则发生在实例销毁之后。如果客户端在事务被销毁前取得控制权,由于服务尚未完成事务,事务会被中止。

注意,使用 TransactionAutoCompleteOnSessionClose 是具有一定风险的,因为它总是受制于事务的超时。会话本质上应该是长期存在的实体,而设计良好的事务却是短期存在的。只有当事务的投票决议需要用到后续调用通过会话获得的信息时,才适合选用这种编程模型。

由于将 TransactionAutoCompleteOnSessionClose 设为 true,会把会话的结束与事务的结束等同起来,因此,在使用客户端事务时,要求客户端在事务完成前终止会话:

```
using(TransactionScope scope = new TransactionScope())
{
    MyContractClient proxy = new MyContractClient();
    proxy.MyMethod();
    proxy.MyMethod();
    proxy.Close();

    scope.Complete();
}
```

如果不这样做，就会中止事务。产生的副作用就是客户端无法便捷地将事务范围的using语句与代理的using语句叠加在一起，因为这样可能会导致代理在事务结束后才被销毁。

```
// 这总是会被中止：
using(MyContractClient proxy = new MyContractClient())
using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod();
    proxy.MyMethod();

    scope.Complete();
}
```

此外，由于代理更适用于一次使用，因此在成员变量中保存代理就显得过于累赘了。

## 事务关联度

设置 TransactionAutoComplete 为 false，在 WCF 中具有独一无二的效果：它在服务实例与事务之间创建了一个关联关系，使得只有某个特定事务可以访问该服务实例。当第一个事务访问服务实例时，会立即建立关联，一旦建立，就会持续到该实例被销毁（直到会话结束）。事务关联度（Transaction Affinity）仅适用于会话服务，这是因为只有会话服务才能将 TransactionAutoComplete 设置为 false。关联度非常关键，因为服务不是状态相关的，它使用了普通成员，因而必须隔离其他事务对成员的访问，以防止与它有关联的事务被中止。因此，关联度提供了一种简陋的事务加锁功能。如果具有事务关联度，那么诸如例 7-20 那样的代码就必定会造成死锁（并最终由于超时而中止），因为第二个事务被阻塞了（与服务访问的资源无关），它在等待第一个事务的完成，与此同时，第一个事务也因为等待第二个事务而被阻塞。

## 混合模式的状态管理

WCF 还支持前两种编程模型的混合（Hybrid）模式，它可以将状态相关的事务型会话服务与有状态的事务型会话服务结合起来。混合模式允许服务实例维护其内存中的状态，直到它完成事务然后使用 ReleaseServiceInstanceOnTransactionComplete 回收状态。考虑例 7-22 中的服务，它实现了例 7-21 中定义的契约。

## 例 7-22: 采用混合模式的会话服务

```
[ServiceBehavior(TransactionAutoCompleteOnSessionClose = true)](译注 9)
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void MyMethod1()
    { ... }
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void MyMethod2()
    { ... }
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod3()
    { ... }
}
```

服务使用 `TransactionAutoCompleteOnSessionClose` 的默认值 (`false`), 而且它的两个方法 (`MyMethod1()` 和 `MyMethod2()`) 将 `TransactionAutoComplete` 设置为 `false`, 也就是说它们不会自动完成事务, 因此会与某个特定事务建立关联关系。这种关联关系可以隔离其他事务对其成员的访问, 以防止与自己有关联关系的事务被中止。现在的问题在于由于服务没有完成事务, 它总是会被中止。为了最终能够完成事务, 服务提供了可完成事务的 `MyMethod3()` 方法。由于服务使用了 `ReleaseServiceInstanceOnTransactionComplete` 的默认值 (`true`), 所以在调用 `MyMethod3()` 方法后, 将完成事务并销毁服务实例, 如图 7-12 所示。注意, `MyMethod3()` 方法也可以调用 `SetTransactionComplete()` 方法显式投票。不过, 重要的是它完成了事务。

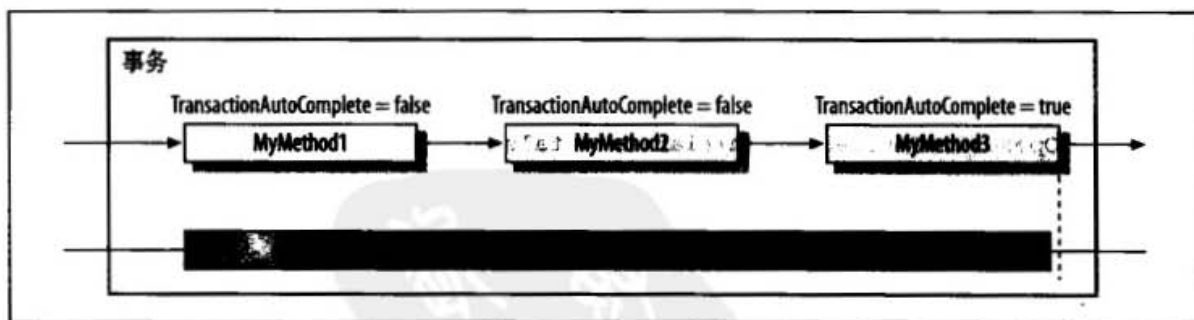


图 7-12: 混合模式的状态管理

混合模式从本质上讲极易出错。首先, 服务实例必须在事务超时之前完成该事务。由于不知道客户端何时才能调用完成方法 (Completing Method), 在此之前我们一直面临着超时的风险。此外, 服务还延长了自己持有对资源管理器锁定的时间, 而资源管理器在

译注 9: 错误, 此处应该将 `TransactionAutoCompleteOnSessionClose` 设置为 `false`, 或者不设置该值, 使用默认值。

会话期间会访问资源。持有锁的时间越长，其他事务发生超时或与该服务的事务产生死锁的可能性也就越高。最终，服务将受控于客户端，因为必须由客户端调用完成方法来结束会话。我们可以使用分步操作 (Demarcating Operations)，对客户端强制实施上述约束：

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(...)]
    void MyMethod1();

    [OperationContract(IsInitiating = false)]
    [TransactionFlow(...)]
    void MyMethod2();

    [OperationContract(IsInitiating = false, IsTerminating = true)]
    [TransactionFlow(...)]
    void MyMethod3();
}
```

## 选择会话事务型服务

当事务的执行以及随后的投票决议需要用到来自会话中的信息时，可以使用事务型会话。例如，考虑下面这个用于订单处理的契约：

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IOrderManager
{
    [OperationContract]
    [TransactionFlow(...)]
    void SetCustomerId(int customerId);

    [OperationContract(IsInitiating = false)]
    [TransactionFlow(...)]
    void AddItem(int itemId);

    [OperationContract(IsInitiating = false, IsTerminating = true)]
    [TransactionFlow(...)]
    bool ProcessOrders();
}
```

实现的服务只有在获得了客户ID以及所有订货之后才能处理订单。不过，依赖于事务型会话的服务通常暗示着拙劣的设计，因为它会影响到系统的吞吐量以及可伸缩性。服务必须较长时间的持有资源锁，从而会面临死锁的风险。我们认为，事务型会话即使是在最好的情况下，也只会是一个畸形的设计；如果情况糟糕，对它的引入无疑会是一个诅咒。在事务型会话中，由状态管理、事务关联度以及事务完成所带来的复杂性，远远超过了从会话中所能获取的好处。通常情况下，最好能够分解契约，以避免使用会话：

```
[ServiceContract]
interface IOrderManager
{
    [OperationContract]
    [TransactionFlow(...)]
    bool ProcessOrders(int customerId,int[] itemIds);
}
```

我们更喜欢简洁的单调服务，并会避免使用事务型会话。

## 事务型单例服务

在默认情况下，一个事务型单例服务的表现与单调服务类似。因为 `ReleaseServiceInstanceOnTransactionComplete` 的默认值为 `true`，所以在单例服务自动完成事务之后，WCF 为了实现状态管理以及满足系统一致性的要求会将服务实例销毁。反过来这也暗示着，单例服务必须是状态相关的，而且会通过资源管理器在每个方法调用中积极地管理其状态。它与单调服务的最大不同在于，WCF 会强制单一实例的语义，以此来保证在任何时间都只有一个单独的实例在运行。WCF 通过并发管理与停止实例来确保这一原则。我们记得，当 `ReleaseServiceInstanceOnTransactionComplete` 的值为 `true` 时，并发模式必须设为 `ConcurrencyMode.Single`，才能够阻止并发调用。此时，WCF 会保持单例上下文，只是停止了托管在上下文中的实例。这意味着尽管单例服务需要与状态相关，但它并不需要客户端在每次调用时都提供一个显式的状态标识符。单例服务可以使用任何一个类型级别的常量，在状态资源管理器中标识其状态，如例 7-23 所示。

例 7-23：状态相关的单例服务

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class MySingleton : IMyContract
{
    readonly static string m_StateIdentifier = typeof(MySingleton).GUID.ToString();

    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    {
        GetState();
        DoWork();
        SaveState();
    }

    // 辅助方法
    void GetState()
    {
        // 使用 m_StateIdentifier 获取状态
    }
    void DoWork()
    {}
    public void SaveState()
```



```

    {
        // 使用 m_StateIdentifier 保存状态
    }
    public void RemoveState()
    {
        // 使用 m_StateIdentifier 从资源管理器中移除
    }
}
// 宿主代码
MySingleton singleton = new MySingleton();
singleton.SaveState(); // 在资源管理器中创建初始化状态

ServiceHost host = new ServiceHost(singleton);
host.Open();

/* 某些用于阻塞的调用 */

host.Close();
singleton.RemoveState();

```

例7-23中，单例服务使用每个类型所独有的GUID作为一个状态标识符。在每个方法的开始部分，服务实例会读取它的状态，并在每个方法的结束部分，将状态存回到资源管理器中去。但是，在第一次调用服务实例时，就必须能够绑定状态，因此我们必须事先向资源管理器中填充服务实例的状态。为此，我们需要在启动宿主前创建服务实例，并将它的状态保存到资源管理器中，然后像第4章介绍的那样，将这个单例实例托管到ServiceHost中。而在关闭宿主后，也一定要从资源管理器中删除实例的状态，如例7-23所示。注意，我们无法在单例服务的构造函数中创建初始状态，因为每次调用单例实例的操作时，都会执行构造函数，这样就会将之前操作中保存的状态覆盖掉。尽管一个状态相关的单例服务确定是可行的（例7-23已经得以证明），但是随之而来的复杂性却使得我们不得不避免使用这一技术。下面介绍的有状态的事务型单例服务会是一种更好的选择。

### 有状态的单例服务

将ReleaseServiceInstanceOnTransactionComplete设置为false后，我们又重新获得了单例的语义。单例对象会在启动宿主之后被立即创建，而该实例又将被所有的客户端与事务所共享。现在的问题在于如何管理单例服务的状态。单例服务必须有状态，否则就失去了存在价值。与之前的有状态会话服务类似，我们可以使用易失性资源管理器作为成员变量来解决这一问题，如例7-24所示。

#### 例7-24：实现有状态的事务型单例服务

```

//////////////////// 服务代码 //////////////////////
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                 ReleaseServiceInstanceOnTransactionComplete = false)]
class MySingleton : IMYContract

```

```
{
    Transactional<int> m_Counter = new Transactional<int>();

    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    {
        m_Counter.Value++;
        Trace.WriteLine("Counter: " + m_Counter.Value);
    }
}

////////// 客户端代码 //////////
using(TransactionScope scope1 = new TransactionScope())
{
    MyContractClient proxy = new MyContractClient();
    proxy.MyMethod();
    proxy.Close();
    scope1.Complete();
}
using(TransactionScope scope2 = new TransactionScope())
{
    MyContractClient proxy = new MyContractClient();
    proxy.MyMethod();
    proxy.Close();
}
using(TransactionScope scope3 = new TransactionScope())
{
    MyContractClient proxy = new MyContractClient();
    proxy.MyMethod();
    proxy.Close();
    scope3.Complete();
}

////////// 输出 //////////
Counter: 1
Counter: 2
Counter: 2
```

在例7-24中，客户端创建了三个事务范围，每个范围都拥有一个指向单例服务的新的代理对象。在每次调用中，服务实例中的计数器成员的值都会被递增，而该成员是由一个 `Transactional<int>` 易失性资源管理器管理的。`scope1` 完成了事务，并提交了 `count` 的新值（此时该值为1）。在 `scope2` 中，客户端调用单例对象，然后暂时将计数器的值增加到2。但是，`scope2` 没有完成它的事务，因此易失性资源管理器拒绝增加计数器的值，并将其恢复到之前的值1。此后，在 `scope3` 中的调用再次将计数器的值由1增加到2，如输出的记录中所示。

注意，在设定 `ReleaseServiceInstanceOnTransactionComplete` 时，单例服务必须至少包含一个将 `TransacionScopeRequired` 设置为 `true` 的方法。

此外，单例服务必须将每个方法的 `TransactionAutoComplete` 属性设为 `true`，从而避免服务实例与某个事务产生关联，以实现并发事务。所有调用和所有事务都指向相同

的实例。例如，下面的客户端代码会形成如图 7-13 所示的事务图。using (MyContractClient proxy = new MyContractClient())。

```
using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod();
    scope.Complete();
}

using(MyContractClient proxy = new MyContractClient())
using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod();
    proxy.MyMethod();
    scope.Complete();
}
```

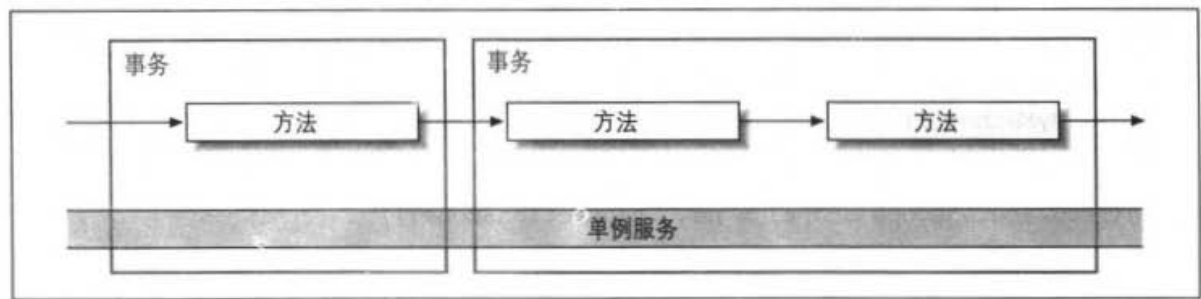


图 7-13：有状态的事务型单例服务

### 实例模式与事务

到目前为止我们可以看到，除非将事务型服务的实例模式设为单调模式，其他实例模式都会给编程模型带来与其优点不相称的复杂性。虽然 WCF 功能强大，并且提供了可扩展性以支持范围最广的配置组合，但我仍然推荐大家尽可能使用单调服务。如果你能够接受单例模式带来的性能损失，那么使用易失性资源管理器的事务型单例服务也是可以接受的。表 7-3 对服务的实例管理模式与事务的关系做了一个总结，列出了到目前为止所讨论过的所有配置选项。注意，表 7-3 只是列出了可行的配置以及它们所产生的效果，其他一些组合在技术上或许是允许的，但是却没有意义。最后，还有一些组合则是 WCF 明确禁止的。

表 7-3：实例模式、配置及事务

配置的实例模式	自动完成	完成时释放	会话结束时完成	最终的实例模式	状态管理	事务关联度
Per call	True	True/false	True/false	Per call	State-aware	Call
Session	False	True/false	True	Session	Stateful	Instance

表 7-3: 实例模式、配置及事务 (续)

配置的实例模式	自动完成	完成时释放	会话结束时完成	最终的实例模式	状态管理	事务关联度
Session	True	False	True/false	Session	Stateful (State-aware, VRM)	Call
Session	True	True	True/false	Per call	State-aware	Call
Session	Hybrid	True	True/false	Hybrid	Hybrid	Instance
Singleton	True	True	True/false	Per call	State-aware	Call
Singleton	True	False	True/false	Singleton	Stateful (State-aware, VRM)	Call

## 回调

回调契约与服务契约一样，也可以向回调客户端传播服务的事务。我们可以像服务契约那样应用 TransactionFlow 特性，例如：

```
interface IMyContractCallback
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void OnCallback();
}
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    ...
}
```

回调方法的实现可以像一个服务操作那样使用 OperationBehavior 特性，并指定对事务范围和事务自动完成功能的要求：

```
class MyClient : IMyContractCallback
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void OnCallback()
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction != null);
    }
}
```

## 回调事务模式

回调客户端具有四种配置模式：Service、Service/Callback、Callback 以及 None，它们

类似于服务的事务模式，只是现在服务扮演了客户端的角色，而回调则扮演了之前服务的角色。例如，要想将回调配置为 Service 事务模式（即回调总是使用服务的事务），应遵循以下步骤：

1. 使用一个与事务相关的双向绑定，并开启事务流。
2. 将回调操作的事务流选项设成强制要求。
3. 将回调操作配置成需要一个事务范围。

例 7-25 演示了一个配置为使用 Service 事务的回调客户端。

例 7-25：将回调配置为 Service 事务模式

```
interface IMyContractCallback
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    void OnCallback();
}

class MyClient : IMyContractCallback
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void OnCallback()
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction.TransactionInformation.
            DistributedIdentifier != Guid.Empty);
    }
}
```

当回调操作被配置为强制要求事务流时，WCF 将强制使用一个事务相关的绑定并启用事务流功能。

如果配置为 Service/Callback 事务传播模式，WCF 不会强制使用一个事务相关的绑定或要求启用事务流功能。开发者可以使用我们定义的 BindingRequirement 特性来验证这一点：

```
interface IMyContractCallback
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void OnCallback();
}

[BindingRequirement(TransactionFlowEnabled = true)]
class MyClient : IMyContractCallback
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void OnCallback()
```

```
    {...}  
}
```

通过实现 `IEndpointBehavior` 接口，我们扩展了 `BindingRequirement` 特性，从而实现检验回调绑定的功能：

```
public interface IEndpointBehavior  
{  
    void AddBindingParameters(ServiceEndpoint endpoint,  
                              BindingParameterCollection bindingParameters);  
    void ApplyClientBehavior(ServiceEndpoint serviceEndpoint,  
                              ClientRuntime behavior);  
    void ApplyDispatchBehavior(ServiceEndpoint endpoint,  
                               EndpointDispatcher endpointDispatcher);  
    void Validate(ServiceEndpoint serviceEndpoint);  
}
```

正如第 6 章所介绍的那样，`IEndpointBehavior` 接口使得我们可以配置被服务用作回调的客户端的终结点。`BindingRequirement` 特性使用了 `IEndpointBehavior.Validate()` 方法，方法的实现与例 7-3 几乎相同。

## 隔离与超时

与服务相似，`CallbackBehavior` 特性使得一个回调类型能够控制其事务的超时与隔离级别：

```
[AttributeUsage(AttributeTargets.Class)]  
public sealed class CallbackBehaviorAttribute : Attribute, IEndpointBehavior  
{  
    public IsolationLevel TransactionIsolationLevel  
    {get;set;}  
    public string TransactionTimeout  
    {get;set;}  
    // 更多成员  
}
```

这些属性就像它们在服务中一样，接受相同的值。同理，它们会选择一个特定的值。

## 回调投票

在默认情况下，WCF 会为回调操作使用自动投票，就好似针对一个服务操作那样。在回调中的任何异常都将投票中止事务，如果没有错误，WCF 将投票提交事务，如例 7-25 所示。不过，回调实例的生命周期是由客户端管理的，它不像服务实例那样有多种实例化模式。任何回调实例都可以将 `TransactionAutoComplete` 设置为 `false`，然后再使用 `SetTransactionComplete()` 方法显式地进行投票：

```
class MyClient : IMyContractCallback
{
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void OnCallback()
    {
        /* Do some transactional work then */

        OperationContext.Current.SetTransactionComplete();
    }
}
```

与会话服务一样，只有当投票依赖于除异常之外的其他原因时，才适用于显式投票。在调用SetTransactionComplete()方法之后，不能执行任何操作，尤其是事务型工作。调用SetTransactionComplete()方法应该是回调操作在方法返回前的最后一行代码。如果我们在调用SetTransactionComplete()方法之后，试图执行任何事务型工作(包括访问Transaction.Current)，WCF就会抛出一个InvalidOperationException异常并中止该事务。

## 使用事务型回调

虽然WCF提供了将服务事务传播到客户端回调方法中的功能，不过事实上回调无法很好地使用服务的事务。首先，回调通常都是单向操作，这样的操作是无法传播事务的。其次，为了让服务能调用客户端的回调操作，服务不能设置为ConcurrencyMode.Single，否则，WCF将中止该调用以避免死锁。服务通常被设置为Client/Service或Client事务传播模式。理想情况下，服务应该能够将原先调用它的客户端事务回传给它要调用的回调操作。然而，服务为了使用客户端事务，必须将TransactionScopeRequired的值设置为true。但是由于ReleaseServiceInstanceOnTransactionComplete的默认值为true，这就要求服务必须使用ConcurrencyMode.Single并发模式，这就阻止了回调。

## 带外事务型回调

实施事务型回调有两种方式。第一种是带外(Out-of-Band)回调，由在宿主端的非服务方使用被服务保存下来的回调引用。由于此时不存在死锁的风险，因此非服务方可以轻松地将它们的事务(通常在一个事务范围中)传播给回调，如例7-26所示。

例 7-26：带外回调

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    static List<IMyContractCallback> m_Callbacks = new List<IMyContractCallback>();
```



```

public void MyMethod()
{
    IMyContractCallback callback = OperationContext.Current.
        GetCallbackChannel<IMyContractCallback>();

    if (m_Callbacks.Contains(callback) == false)
    {
        m_Callbacks.Add(callback);
    }
}
public static void CallClients()
{
    Action<IMyContractCallback> invoke = delegate(IMyContractCallback callback)
    {
        using (TransactionScope scope
            = new TransactionScope())
        {
            callback.OnCallback();
            scope.Complete();
        }
    };

    m_Callbacks.ForEach(invoke);
}
// 带外回调:
MyService.CallClients();

```

## 服务事务型回调

第二种方式是谨慎地配置事务型服务，使得它能够将调用返回给它正在调用的客户端。为此，我们需要将服务设置为 `ConcurrencyMode.Reentrant`，同时将 `ReleaseServiceInstanceOnTransactionComplete` 属性设置为 `false`，并确保至少有一个操作的 `TransactionScopeRequired` 属性值为 `true`，如例 7-27 所示。

### 例 7-27：配置为事务型回调

```

[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void MyMethod(...);
}
interface IMyContractCallback
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void OnCallback();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
    ConcurrencyMode = ConcurrencyMode.Reentrant,

```

```

        ReleaseServiceInstanceOnTransactionComplete = false)]
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(...)
    {
        Trace.WriteLine("Service ID: " +
            Transaction.Current.TransactionInformation.DistributedIdentifier);

        IMyContractCallback callback =
            OperationContext.Current.GetCallbackChannel<IMyContractCallback>();
        callback.OnCallback();
    }
}

```

这一约束的基本原理将在下一章阐释。

根据例 7-27 的定义，同时在绑定中开启事务流，则客户端代码如下所示：

```

class MyClient : IMyContractCallback
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void OnCallback()
    {
        Trace.WriteLine("OnCallback ID: " +
            Transaction.Current.TransactionInformation.DistributedIdentifier);
    }
}

MyClient client = new MyClient();
InstanceContext context = new InstanceContext(client);
MyContractClient proxy = new MyContractClient(context);

using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod();
    Trace.WriteLine("Client ID: " +
        Transaction.Current.TransactionInformation.DistributedIdentifier);
    scope.Complete();
}
proxy.Close();

```

输出的内容与如下结果相似：

```

Service ID: 23627e82-507a-45d5-933c-05e5e5a1ae78
OnCallback ID: 23627e82-507a-45d5-933c-05e5e5a1ae78
Client ID: 23627e82-507a-45d5-933c-05e5e5a1ae78

```

这就表明客户端事务被传播给了服务，并且进入了回调。

显然，将 `ReleaseServiceInstanceOnTransactionComplete` 设置为 `false` 意味着 WCF 不会在事务完成后立即回收该实例。对此最好的补救措施是尽量为事务型回调选择

单调服务（如例 7-27 所示），因为无论如何它们都会在方法返回后被销毁，而它们的状态相关的编程模型是与 `ReleaseServiceInstanceOnTransactionComplete` 属性无关的。

如果我们使用的是一个会话服务，就必须遵循此前提到的准则，我们必须在 `ReleaseServiceInstanceOnTransactionComplete` 为 `false` 时，管理会话服务的状态，即采用状态相关的编程模型，或者使用易失性资源管理器（译注 10）。

---

译注 10：即采用有状态的编程模型。

## 第8章

# 并发管理

传入的客户端调用会被分发给线程池中线程上的服务。既然多个客户端可以发出多个并发调用,那么服务自身也能够维持多线程的多个调用。如果这些调用被分发给相同的实例,我们就必须提供线程安全的访问,以访问服务内存状态或者危机状态的崩溃与错误。在回调期间,同样需要提供对客户端内存状态的线程安全访问,因而回调也会被线程池中的线程分发。如果实例状态可用,除了要同步访问这些实例状态,所有服务都需要同步访问实例之间共享的资源,例如静态变量或者UI(用户接口)控件。从另一个层面上讲,如果需要的话,并发管理能够确保服务(或者服务访问的资源)执行在特定的线程上。

WCF 提供了两种同步模式。自动同步(Automatic Synchronization)要求 WCF 同步访问服务实例。它的实现非常简单,易于使用,但只能用于服务与回调类。手动同步(Manual Synchronization)增加了开发者实现同步的负担,要求实现特定的应用程序集成。开发者必须使用 .NET 同步锁,这也是目前实现同步的一条专门的原则。手动同步的优势在于它可以用于服务和类似于服务的非服务类,允许开发者优化吞吐量与可伸缩性。本章首先介绍了基本的并发模式,然后深入探讨了关于并发管理更多的高级特性,如资源的安全与同步、线程关联度(Thread Affinity)、定制同步上下文、回调以及异步调用。至于并发管理技术的最佳实践与设计指南,则贯穿于本章始终。

## 实例管理与并发

服务实例的线程安全与服务实例模式紧密相关。单调服务实例从定义上讲是线程安全的,因为每次调用都获取了自己专有的实例。该实例仅仅对于它分配的工作线程才是可用的,因为没有其他线程能够访问它,自然就不必进行同步。然而,单调服务通常都是与状态相关的。状态的存储可以是内存资源,例如静态变量。由于服务能够维持同步调用,因此存储的状态允许被多线程访问。因而,我们必须以同步方式访问存储的状态。

会话服务需要并发管理与同步。原因在于客户端可以使用相同的代理,也可以将多个客户端线程的调用分发给服务。一个单例服务更需要并发访问,并且必须执行同步访问。单例服务拥有的内存状态自动地被所有客户端所共享,而不需要明确的指定。会话服务通过多线程分发调用的可能性非常高,而一个单例对象则可以在不同的执行上下文中拥有多个客户端,每个客户端都使用了自己的线程去调用服务。所有的这些调用会进入线程池中不同线程的单例对象,因此需要同步处理。

## 服务并发模式

ServiceBehavior 特性的 ConcurrencyMode 属性负责管理服务实例的并发访问:

```
public enum ConcurrencyMode
{
    Single,
    Reentrant,
    Multiple
}

[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : ...
{
    public ConcurrencyMode ConcurrencyMode
    {get;set;}
    // 更多成员
}
```

ConcurrencyMode 枚举值负责控制并发调用是否允许以及何时应用到服务实例上。

### ConcurrencyMode.Single

如果服务被设置为 ConcurrencyMode.Single, WCF 会为服务实例提供自动同步,并通过关联服务实例与同步锁以禁止并发调用。每次进入服务的调用首先必须获取锁。如果锁是无主的,那么调用者就会锁住该同步锁,并被获准进入。一旦操作返回, WCF 就会将锁解除,以允许其他调用者进入。重要的是在同一时刻只能允许一个调用者。在锁定同步锁时,如果有多个并发调用者,则这些调用者就会被放入到队列中,然后按顺序在队列中等待使用。如果调用在阻塞期间超时了, WCF 就会从队列中移除该调用者,客户端则获得一个 TimeoutException 异常。ConcurrencyMode.Single 是 WCF 的默认设置,因此如下的定义是等效的:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
    ConcurrencyMode = ConcurrencyMode.Single)]
```

```
class MyService : IMyContract  
{...}
```

因为默认的并发模式为同步访问,所以会话服务与单例服务默认的实例模式都是同步的:

```
[ServiceContract(SessionMode = SessionMode.Required)]  
interface IMyContract  
{...}  
  
// 这些有状态的服务是线程安全的  
  
class MyService1 : IMyContract  
{...}  
  
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]  
class MyService2 : IMyContract  
{...}  
  
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]  
class MySingleton : IMyContract  
{...}
```

---

**注意:** 在会话服务或单例服务中,应尽量定义执行周期短的操作,以避免阻塞客户端的时间过长。因为服务实例是同步的,如果操作耗费的时间太长,就会导致挂起的调用者超时。

---

## 同步访问与事务

正如第7章所阐释的那样,在服务装载时WCF会校验服务的操作,以判断是否包含了一个或多个操作,它们的 `TransactionScopeRequired` 属性值以及 `ReleaseServiceInstanceOnTransactionComplete` 属性值为 `true`。同时,确保服务的并发模式必须是 `ConcurrencyMode.Single`。只有如此,才能保证服务实例在事务结束时被回收,而不会让其他线程访问已经释放的实例。

## ConcurrencyMode.Multiple

如果服务被设置为 `ConcurrencyMode.Multiple`, WCF 不会管理服务,而且,不管在任何情况都不会同步访问服务实例。`ConcurrencyMode.Multiple` 代表了服务实例是与同步锁无关的,因此,服务实例支持并发调用。换言之,如果服务实例被配置为 `ConcurrencyMode.Multiple`, WCF 就不会将客户端消息放入队列中,一旦它们到达,就立即将它们分发给服务实例。

---

**注意:** 大量的并发客户端调用不会在服务上产生数量匹配的并发执行调用。分发给服务的并发调用最大值与配置的最大并发调用的限流值 (Throttled Value) 有关。正如第4章所述,默认的最大并发调用值为 16。

---

显然,我们必须重视会话服务与单例服务,但有时也需要关注单调服务,我们会在后面看到这方面的内容。这些服务必须手动执行对服务状态的同步访问。通常的做法是利用.NET锁,例如Monitor或者WaitHandle的继承子类。手动同步方式非常复杂,在拙著《Programming .NET Components》(O'Reilly)的第8章中对其有深入地介绍。手动同步允许服务开发者优化服务实例上客户端调用的吞吐量,不管调用发生的时机与位置,只要同步是必要的,我们就能够锁定服务实例,从而允许相同服务实例的其他客户端处于同步的范围之内。实现手动同步的服务如例8-1所示。

例8-1: 使用局部锁 (Fragmented Locking) 实现手动同步

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    void MyMethod();
}
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyService : IMyContract
{
    int[] m_Numbers;
    List<string> m_Names;

    public void MyMethod()
    {
        lock(m_Numbers)
        {
            ...
        }

        /* '此处不要访问成员 */

        lock(m_Names)
        {
            ...
        }
    }
}
```

例8-1所示的服务支持并发访问。既然需要同步操作的关键节可以是任意的成员变量访问,因此服务在访问对象之前使用了Monitor(封装在lock语句中)锁定对象。本地变量不需要同步,因为只有它们在它们自己的调用栈上创建它们的线程才可以访问它们。例8-1使用的技术可能会导致死锁,也易于出错。它只提供了线程安全访问,要求服务的其他操作必须遵循一个原则,就是在访问这些操作之前需要为成员加锁。但是,即使所有的操作锁定了所有的成员,仍然存在死锁的危险。例如,如果线程A的一个操作试图访问成员M2,而它又锁定了成员M1。同时,线程B又在并发执行另外的操作,这个操作试图访问成员M1,而它又锁定了成员M2。此时就会陷入死锁的僵局。



**注意：**通过促使调用超时并抛出 `TimeoutException` 异常的方式，WCF 可以解决服务调用的死锁问题。我们应避免设置过长的超时值，因为它会减弱 WCF 为解决死锁做出及时反应的能力。

因此建议避免使用局部锁。最好的办法是锁定整个服务实例：

```
public void MyMethod()
{
    lock(this)
    {
        ...
    }

    /* 在此不访问成员 */

    lock(this)
    {
        ...
    }
}
```

问题依然存在，因为锁仍然是局部的，容易出现错误。如果在将来某个开发者在某个时候为方法调用添加了非同步的代码，这些代码又会访问成员，则整个方法就不再是同步访问的了。更好的办法是对整个方法的主体加锁：

```
public void MyMethod()
{
    lock(this)
    {
        ...
    }
}
```

我们甚至可以使用带有 `MethodImplOptions.Synchronized` 标志的 `MethodImpl` 特性，通过 .NET 自动注入调用，对实例加锁：

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyService : IMyContract
{
    int[] m_Numbers;
    List<string> m_Names;

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void MyMethod()
    {
        ...
    }
}
```

我们需要为所有的服务操作实现重复地分配 `MethodImpl` 特性。

现在的问题是，代码虽然是线程安全的，然而在此时使用 `ConcurrencyMode.Multiple` 却毫无意义（译注1）。因为从同步的角度来说，目前的定义与使用 `ConcurrencyMode.Single` 的效果几乎相同，反而还增加了整个代码的复杂度，为开发者增加了实现的障碍。在这种情况下，我们应该使用配置方式，特别是在引入回调的时候更是如此。之后我们会看到这一内容的介绍。

## 非同步访问与事务

当服务被配置为 `ConcurrencyMode.Multiple` 时，如果存在至少一个操作的 `TransactionScopeRequired` 属性值被设置为 `true`，那么服务行为的 `ReleaseServiceInstanceOnTransactionComplete` 属性就必须设置为 `false`。例如，下面的代码就是有效的定义，因为在默认的 `ReleaseServiceInstanceOnTransactionComplete` 属性值为 `true` 的情况下，没有一个方法的 `TransactionScopeRequired` 值被设置为 `true`：

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyService : IMyContract
{
    public void MyMethod()
    { ... }
    public void MyOtherMethod()
    { ... }
}
```

下面的定义则是无效的，因为至少有一个方法的 `TransactionScopeRequired` 属性值被设置成了 `true`：

```
// 无效配置：
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }
    public void MyOtherMethod()
    { ... }
}
```

具有事务功能的非同步服务必须明确地将 `ReleaseServiceInstanceOnTransactionComplete` 属性的值设置为 `false`：

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple,
    ReleaseServiceInstanceOnTransactionComplete = false)]
class MyService : IMyContract
```

---

译注1：锁定了整个方法，实际上就相当于禁止了方法的并发调用。

```
{  
    [OperationBehavior(TransactionScopeRequired = true)]  
    public void MyMethod()  
    {...}  
    public void MyOtherMethod()  
    {...}  
}
```

这一约束的背后所隐含的原理就是：只有会话服务或单例服务采用非同步访问才具有价值。因此，在执行事务访问时，WCF 要求实例模式的配置在语义上要符合要求。此外，当实例被其他调用者使用时，这样的配置能够避免调用者访问实例、完成事务以及发布实例。

## ConcurrencyMode.Reentrant

ConcurrencyMode.Reentrant 值相当于简化了的 ConcurrencyMode.Single。与 ConcurrencyMode.Single 相似，ConcurrencyMode.Reentrant 关联了服务实例与同步锁，因而禁止在相同的实例上执行并发调用。但是，如果重入服务（Reentrant Service）执行了向其他服务或回调的向外的调用，则调用链（或调用的结果）就会间接地返回如图 8-1 所示的服务实例，然后允许调用重新进入服务实例。

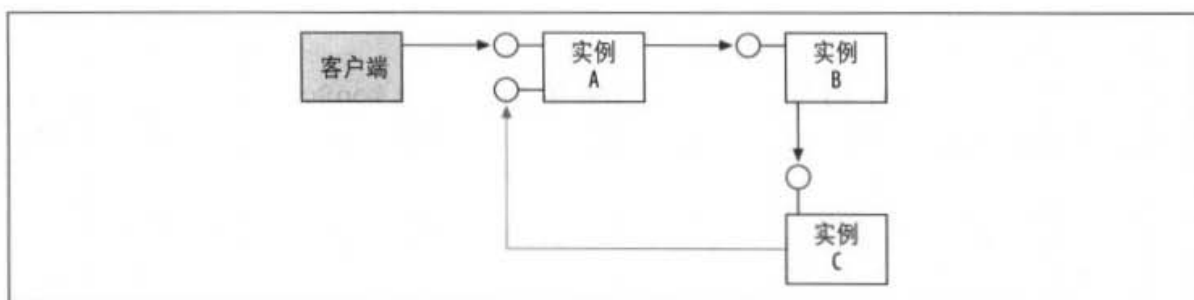


图 8-1：调用重入

ConcurrencyMode.Reentrant 的实现非常简单。当重入服务通过 WCF 执行向外的调用时，WCF 会发布一个与实例关联的同步锁。设计 ConcurrencyMode.Reentrant 的目的在于避免重入存在的潜在死锁。在执行向外的调用时，如果服务要继续维持对锁的占用，那么当调用的结果试图进入相同的实例时，就会导致死锁。

如下几种场景支持重入：

- 如果单例服务调用的任何一个下游服务（Downstream Service）试图将调用返回给单例对象，则单例服务在执行向外的调用时就存在死锁的危险。
- 在相同的应用程序域中，如果客户端将代理引用存储在某些可用的全局静态变量中，

则代理引用的服务所调用的某些下游服务就会使用代理引用将调用返回给原来的服务。

- 非单向操作上的回调必须允许正在调用的服务能够重入。
- 如果服务执行向外的调用周期过长，甚至没有重入，我们可能需要通过支持其他客户端使用相同的服务实例以优化吞吐量。

---

**注意：**配置为 `ConcurrencyMode.Multiple` 的服务也可以定义为重入，因为在调用期间锁没有被占用。但是，与重入服务本质上是线程安全的不同，配置为 `ConcurrencyMode.Multiple` 的服务必须通过诸如在每次调用期间锁定实例的方式，支持服务自身的同步，正如之前所阐释的那样。服务的开发者负责决定在执行向外的调用之前，是否需要释放实例的锁，以避免重入死锁。

---

## 设计重入

我们务必要认识到与重入相关联的职责问题。首先，当重入服务执行向外的调用时，它必须保持服务状态为可工作的一致状态，因为服务在执行向外调用时，允许其他服务进入服务实例。一个一致的线程安全状态意味着重入服务与自己的成员或其他本地对象、静态变量没有太多的交互。当向外的调用返回时，重入服务可能只是将控制权返回给它的客户端。例如，假定重入服务修改了某个链表中的状态。如果链表缺少一个头节点，而状态又需要从另外的服务中获取新的头节点的值，此时就会存在不一致的状态。然后，重入服务调用其他服务，但是在此刻它却会影响其他客户端的稳定性。因为如果这些客户端调用了重入服务，并访问了链表，就会出现错误。

同样，当重入服务从它的调出端返回时，必须刷新所有的本地方法状态。例如，如果服务拥有的本地变量包含了成员变量状态的一个副本，那么本地变量的现有值可能就是错误的，因为在调出期间，其他参与方可能已经进入重入服务，修改了成员变量的值。

## 重入与事务

关于重入服务的事务处理面临与 `ConcurrencyMode.Multiple` 服务相同的设计约束，也就是说，如果至少有一个操作的 `TransactionScopeRequired` 属性值被设置为 `true`，则 `ReleaseServiceInstanceOnTransactionComplete` 的属性值必须设置为 `false`。

## 回调与重入

回调是使用重入的主要原因。正如第5章所述，如果一个WCF服务需要调用双向回调到正在调用的客户端，服务需要重入（或者通过 `ConcurrencyMode.Multiple` 执行局部

同步)。因为一旦回调返回实例锁需要的所属权,就必须处理从客户端返回的应答消息。如果配置为 `ConcurrencyMode.Single` 的服务允许调用返回给它的客户端,就会导致死锁。若要支持回调,服务必须配置为 `ConcurrencyMode.Multiple`, 更好的方式是配置为 `ConcurrencyMode.Reentrant`。即使对于只需要配置 `ConcurrencyMode.Single` 的单调服务而言,如果需要在支持回调,仍然需要配置为重入。注意,服务可能会调用回调到其他客户端,或者调用其他服务。回调到正在调用的客户端则是被禁止的。

如果一个服务被配置为 `ConcurrencyMode.Single`, 当它试图调用一个双向回调时, WCF 会抛出一个 `InvalidOperationException` 异常。例 8-2 演示了如何将一个服务配置为重入的方法。在操作执行期间,服务将调用返回给它的客户端。一旦回调返回,控制权就只会返回给服务,服务自身的线程需要重新确认锁。

例 8-2: 配置重入以支持回调

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();
}
interface IMyContractCallback
{
    [OperationContract]
    void OnCallback();
}
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
class MyService : IMyContract
{
    public void DoSomething()
    {
        IMyContractCallback callback = OperationContext.Current.
            GetCallbackChannel<IMyContractCallback>();
        callback.OnCallback();
    }
}
```

如同在第 5 章中提到的那样,当服务被配置为 `ConcurrencyMode.Single` 时,如果要将调用返回给它的客户端,只有一种方式可行,就是将回调契约的操作配置为单向,因为它没有任何应答消息与锁产生争用。

## 实例与并发访问

使用相同的代理,单个客户端能够发出多个并发调用到服务。客户端能够使用多线程调用服务的调用,客户端也可以快速地连续发出多个单向调用。不管是何种情况,对相同

客户端调用的并发处理,与服务配置的实例模式、服务的并发模式以及配置的传递方式,即服务的绑定类型与会话模式有关。

## 单调服务

单调服务如果没有传输层会话,也就是说,如果服务的绑定为BasicHttpBinding绑定,或者服务使用任何一种WS绑定,同时契约被配置为SessionMode.NotAllowed或SessionMode.Allowed,并且没有安全与可靠消息传输,则单调服务允许对调用进行并发处理。一旦调用到达服务,它们就会被分发,每一个调用对应一个新的实例,并被并发执行。此时,我们不需要考虑服务的并发模式。我认为这种行为应该是合理的。

如果单调服务拥有传输层会话,也就是说,服务使用TCP绑定或IPC绑定,或者使用WS绑定,但契约的SessionMode属性被设置为SessionMode.Allowed,并且包含了安全或可靠消息传输,或者使用WS绑定,但契约被设置为SessionMode.Required。此时,调用的并发处理与服务的并发模式有关。如果服务被配置为ConcurrencyMode.Single,则禁止对未决调用执行并发处理。由于这一约束归因于通道架构的直接影响,因此我个人认为这属于WCF的一个设计缺陷,因为锁应该与实例相关,而不是与类型相关。如果服务被配置为ConcurrencyMode.Multiple,并发处理是允许的。一旦调用到达客户端,它们就会被分发。每一个调用对应一个新的实例,并被并发执行。当服务被配置为ConcurrencyMode.Reentrant时,如果服务没有执行向外的调用,则它的执行方式与ConcurrencyMode.Single相似。如果服务执行了向外的调用,那么下一个调用就会允许进入,返回的调用则必须协调对锁的占用,就像对其他未决调用的处理方式一样。

## 会话服务与单例服务

对于会话服务或单例服务,单独配置的并发模式负责管理未决调用的并发执行。如果服务被配置为ConcurrencyMode.Single,则调用每次都会被放到服务实例上,未决调用(Pending Call)则被放入队列中。我们应该避免耗时过长的调用处理,因为它存在调用超时的危险。

如果服务实例被配置为ConcurrencyMode.Multiple,WCF允许对调用执行并发处理。服务实例执行调用的速度与它们回到通道的速度一致(这取决于限流的限制)。对于有状态的非同步服务实例,我们必须同步访问服务实例,否则存在状态被破坏的风险。

如果服务实例被配置为ConcurrencyMode.Reentrant,则它的执行方式与ConcurrencyMode.Single相似。但是,如果服务执行向外的调用,则允许执行下一个调用(可能是单向,也可能不是)。我们必须遵循之前讨论过的有关重入环境中的编程准则。



**注意：**对于配置为 `ConcurrencyMode.Multiple` 的会话服务，若要采用并发调用，客户端必须使用多个工作线程去访问相同的代理实例。但是，如果客户端线程依赖于代理的自动打开特性（即在调用方法时，如果代理没有打开，调用会打开代理），同时以并发方式调用代理，那么实际上调用会被序列化，直到打开代理，并在之后执行并发处理。如果我们不考虑代理的状态就分发并发调用，则客户端需要在发出任何基于工作线程的调用之前，显式地打开代理（通过调用 `Open()` 方法）。

## 资源与服务

使用 `ConcurrencyMode.Single` 或者一个显式的同步锁同步访问服务实例时，只能管理服务实例状态自身的同步访问。它无法为服务可能正在使用的重要资源提供安全访问。这些资源必须是线程安全的。例如，考虑图 8-2 所示的应用程序。

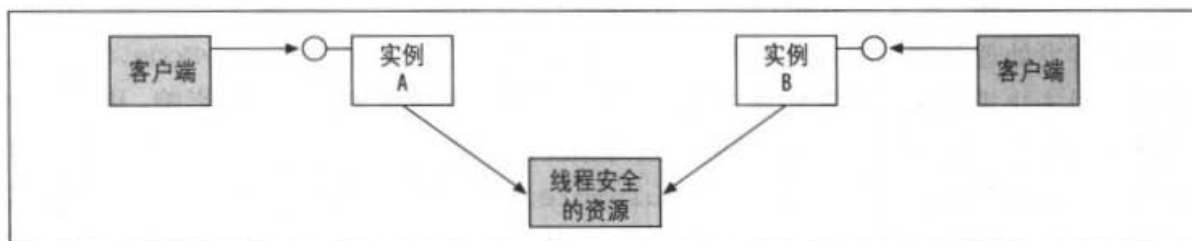


图 8-2：应用程序必须同步访问资源

即使服务实例是线程安全的，如果两个实例试图并发访问相同的资源（例如一个静态变量、静态辅助类或文件），则必须同步访问资源自身，而与服务的实例模式无关。甚至于单调服务也可能出现图 8-2 所示的情形。

## 访问死锁

为了提供对资源的线程安全访问，一个简单的解决方案是为每个资源提供它自身拥有的锁，潜在的含义就是将锁封装在资源之中。当服务访问资源时，要求资源锁定锁；当服务处理资源时，则解除锁。但是，这种方式却容易导致死锁。考虑图 8-3 所示的情形。

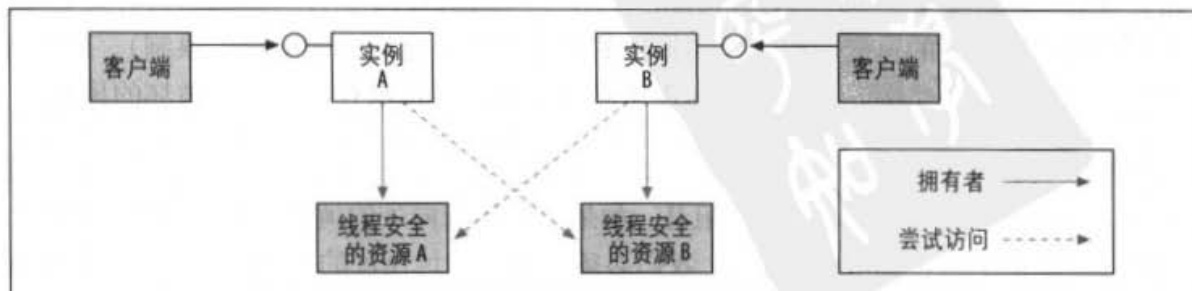


图 8-3：资源访问造成的死锁



图中，服务实例 A 访问线程安全的资源 A。资源 A 拥有自己的同步锁，而实例 A 会获取该锁。同样的，实例 B 访问资源 B，并获取资源 B 的锁。当实例 A 试图访问资源 B，而实例 B 同时又试图访问资源 A 时，就会导致死锁，此时，每个实例都会等待对方释放彼此的锁。

服务的并发模式以及实例模式与死锁没有必然联系。避免死锁的唯一途径是将服务同时配置为 `InstanceContextMode.Single` 和 `ConcurrencyMode.Single`，因为一个同步的单例服务，在同一时间只能有一个客户端。在访问资源时，没有其他的实例处于死锁状态。并发模式与实例模式的其他组合都不能解决死锁的问题。例如，一个同步的会话服务可能拥有两个独立的线程安全实例，分别与两个不同的客户端相关联。在访问资源时，这两个实例仍然会导致死锁。

## 避免死锁

有几种可能的方案避免死锁。如果服务的所有实例都严格按照相同顺序访问所有的资源，例如总是首先获取资源 A 的锁，然后再获取资源 B 的锁，就不会导致死锁。然而，我们很难在服务的整个生命周期中始终如一地执行这样的顺序访问。随着时间的推移，在维护代码期间，可能会逐渐偏离这条严格的规定（甚至在调用辅助类的方法时，会不经意地破坏规定），从而触发死锁。

另一种解决方案是让所有资源使用一个相同的共享锁。为了降低死锁的几率，我们需要减少系统中锁的数量，让服务自身也使用相同的锁。要达到这样的目的，我们可以将服务配置为 `ConcurrencyMode.Multiple`（即使是单调服务），从而避免使用 WCF 提供的锁。获取了共享锁的第一个服务实例会锁定所有的其他实例，并拥有所有的重要资源。使用一个共享锁，最简单的技巧是锁定服务类型，如例 8-3 所示。

例 8-3：使用服务类型作为共享锁

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
                 ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyService : IMyContract
{
    public void MyMethod()
    {
        lock(typeof(MyService))
        {
            ...
            MyResource.DoWork();
            ...
        }
    }
}
static class MyResource
```

```
{  
    public static void DoWork()  
    {  
        lock(typeof(MyService))  
        {  
            ...  
        }  
    }  
}
```

资源自身必须同时锁定服务类型（或者其他事先协定的共享类型）。使用共享锁的方法存在两个问题。首先，它会引入资源与服务之间的耦合，因为资源的开发者必须知道服务的类型或者用于同步的类型。虽然我们可以通过将类型作为资源对象的构造参数，以此降低两者之间的耦合度，但这种方式却不适用于第三方提供的资源。第二个问题是在执行服务实例时，会阻塞所有的其他实例（以及它们对应的客户端）。如果系统要求吞吐量与响应速度，那么在使用共享锁时，应避免耗时过长的操作。

---

**警告：** 服务绝对不能共享资源。不管是否并发管理，资源作为本地的实现细节，不应被多个服务共享。最重要的是，跨越服务边界去共享资源很容易导致死锁。这样的共享资源难以跨技术与组织实现锁的共享，而服务也需要协调锁的顺序。这就意味着服务之间的强耦合，是与面向服务的最佳实践与原则是相悖的。

---

## 资源同步上下文

传入的服务调用在工作线程上执行。WCF会管理这些线程，它们与所有的服务或资源线程无关。这意味着在默认状态下服务不能依赖于各种线程关联度（Thread Affinity），它总是被相同的线程访问。大致相同的是，在默认状态下，如果宿主或服务的开发者创建了一些宿主端定制线程，那么服务并不能依赖于这些线程的执行。因而这样的解决方案就会导致某些资源会依赖于线程关联度，例如被服务更新的用户界面资源只能在用户界面（UI）线程中执行与访问。另一个例子是使用了线程局部存储（Thread Local Storage, TLS）的资源存储了相同线程的所有参与者全局共享的带外（Out-Of-Band）信息。TLS要求使用相同的线程。此外，考虑到系统的可伸缩性与吞吐量，可能需要通过自己的线程池访问某些资源或框架。

无论何时，只要服务与一个特定的线程或多个线程存在关联关系，服务就不能只在传入的 WCF 工作线程上执行调用。相反，服务必须将调用封送到服务访问的资源所获取的正确线程。

## .NET 2.0 的同步上下文

.NET 2.0 引入了同步上下文的概念。任何参与方都可以提供执行上下文，并让其余参与方封送调用到该上下文。同步上下文可以是单线程的，也可以是任意数量的指定线程。通常情况下，同步上下文都是单个而特定的线程。所有同步上下文都要确保调用执行在正确的线程上。注意，术语“上下文”代表了多重含义。它与本书前面介绍的服务实例上下文或操作上下文没有半点联系。

从概念上讲，同步上下文仅仅是一种可供使用的设计模式。同步上下文的实现异常复杂，开发者通常并不愿意面对它。

### SynchronizationContext 类

System.Threading 命名空间中定义了 SynchronizationContext 类，用以代表同步上下文：

```
public delegate void SendOrPostCallback(object state);

public class SynchronizationContext
{
    public virtual void Post(SendOrPostCallback callback, object state);
    public virtual void Send(SendOrPostCallback callback, object state);
    public static void SetSynchronizationContext(SynchronizationContext context);
    public static SynchronizationContext Current
    {get;}
    // 更多成员
}
```

.NET 2.0 中的每个线程都有一个与之关联的同步上下文。通过访问 SynchronizationContext 类的静态属性 Current，可以获取线程的同步上下文。如果线程没有包含同步上下文，则 Current 返回 null。我们也可以在线程之间传递同步上下文的引用，如此，一个线程就能够封送调用到另一个线程。

为了表示同步上下文中的调用，我们可以在方法中包装一个 SendOrPostCallback 类型的委托。注意委托的签名使用了 object 类型。如果要传递多个参数，可以将这些参数组成一个结构类型，并将结构当作 object 进行传递。

---

**警告：**同步上下文使用了没有确定类型的 object，因而缺乏编译时的类型安全。当我们使用同步上下文时，应该加倍小心。

---

## 使用同步上下文

有两种办法可以将调用封送到同步上下文：分别以同步和异步发送或传递一个工作项。`Send()`方法会阻塞调用者，直到调用在另外的同步上下文中完成。`Post()`方法只是将调用分发给同步上下文，然后将控制权返回给它的调用者。

例如，若要以同步方式将调用封送给特定的同步上下文，首先需要获得该同步上下文的引用，然后调用`Send()`方法：

```
// 获取同步上下文
SynchronizationContext context = ...

SendOrPostCallback doWork = delegate(object arg)
{
    // 此处代码用于保证执行在正确的线程上
};

context.Send(doWork, "Some argument");
```

例 8-4 演示了一个更具有实际意义的例子。

例 8-4：在正确的同步上下文中调用资源

```
class MyResource
{
    public int DoWork()
    {...}
    public SynchronizationContext MySynchronizationContext
    {get;}
}

class MyService : IMyContract
{
    MyResource GetResource()
    {...}

    public void MyMethod()
    {
        MyResource resource = GetResource();
        SynchronizationContext context = resource.MySynchronizationContext;

        int result = 0;
        SendOrPostCallback doWork = delegate
        {
            result = resource.DoWork();
        };

        context.Send(doWork, null);
    }
}
```

例中，`MyService` 服务需要与 `MyResource` 资源交互，并通过执行 `DoWork()` 方法完成某些工作，并返回结果。但是，`MyResource` 要求所有的调用都要在它的特定同步

上下文中执行。MyResource通过MySynchronizationContext属性使得执行上下文可以被使用。服务操作MyMethod()在一个WCF工作线程上执行。它首先会获取资源及它的同步上下文。然后,定义一个匿名方法调用DoWork方法,并将该匿名方法分配给SendOrPostCallback类型的doWork委托。最后,MyMethod()方法调用Send()方法,并将null值作为方法的参数值,因为资源的DoWork()方法不需要参数值。注意例8-4是从调用中获取返回值。既然Send()返回void值,那么匿名方法就会将DoWork()的返回值分配给输出变量。如果没有匿名方法,要完成这一任务就需要使用复杂的同步成员变量。

例8-4会导致服务与实例之间的强耦合。服务需要知道资源能够识别它的同步上下文,获取上下文,并且管理执行。因而,更好的办法是将这一需求封装到资源自身的定义中,如例8-5所示。

#### 例8-5: 封装同步上下文

```
class MyResource
{
    public int DoWork()
    {
        int result = 0;
        SendOrPostCallback doWork = delegate
        {
            result = DoWorkInternal();
        };
        MySynchronizationContext.Send(doWork,null);
        return result;
    }
    SynchronizationContext MySynchronizationContext
    {get;}
    int DoWorkInternal()
    {...}
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    MyResource GetResource()
    {...}
    public void MyMethod()
    {
        MyResource resource = GetResource();
        int result = resource.DoWork();
    }
}
```

对比例8-5与例8-4。例8-5中的所有服务都必须访问资源,因为服务要从内部将调用封装给它的同步上下文。

## UI 同步上下文

使用同步上下文的一个标准场景是Windows用户界面框架,如Windows窗体或Windows Presentation Foundation (WPF)。为简便起见,本章余下讨论的内容只涉及Windows窗体,即使它的应用与WPF等同。Windows UI应用程序依靠基本的Windows消息与消息处理循环(消息通道)处理同步上下文。消息循环必须具有线程关联度,因为消息到窗体的传递只能发生在创建它的线程上。一言以蔽之,我们必须将消息封送到任何试图访问窗体控件或窗体的UI线程,否则就会导致错误或失败发生。如果服务需要更新某些用户界面,客户端调用或相关事件的结果则可能存在问题。所幸,Windows窗体支持同步上下文模式。传递消息的线程包含了一个同步上下文,即WindowsFormsSynchronizationContext类:

```
public sealed class WindowsFormsSynchronizationContext : SynchronizationContext, ...
{ ... }
```

无论何时,只要调用Windows窗体的Application.Run()方法,应用程序就会生成一个主窗体。它不仅启动了对窗体消息的处理,还会安装WindowsFormsSynchronizationContext对象作为当前线程的同步上下文。

WindowsFormsSynchronizationContext要做的是将Send()或Post()的调用转换为定制的窗体消息,然后传递窗体消息到UI线程的消息队列中。每一个Windows窗体UI类都派生自Control类,它定义了一个特殊方法,能够通过调用SendOrPostCallback委托处理定制消息。在某个时刻,UI线程会处理定制的窗体消息,以及调用委托。

因为在正确的同步上下文中已经调用了窗体或控件,当调用Send()方法时,为了避免死锁,Windows窗体同步上下文的实现会校验调用的封送是否真正需要。如果不需要封送,则直接在调用的线程上调用。

## UI 访问与更新

当一个服务需要更新某些用户界面时,首先必须通过某些专有机制查找需要更新的窗体。一旦服务获得正确的窗体,就必须持有窗体的同步上下文,并将调用封送给服务。例8-6演示了一种可能的交互形式。

例8-6: 使用窗体同步上下文

```
partial class MyForm : Form
{
    Label m_CounterLabel;
    SynchronizationContext m_SynchronizationContext;

    public MyForm()
    {
        InitializeComponent();
```

```

        m_SynchronizationContext = SynchronizationContext.Current;
        Debug.Assert(m_SynchronizationContext != null);
    }
    public SynchronizationContext MySynchronizationContext
    {
        get
        {
            return m_SynchronizationContext;
        }
    }
    public int Counter
    {
        get
        {
            return Convert.ToInt32(m_CounterLabel.Text);
        }
        set
        {
            m_CounterLabel.Text = value.ToString();
        }
    }
}
[ServiceContract]
interface IFormManager
{
    [OperationContract]
    void IncrementLabel();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IFormManager
{
    public void IncrementLabel()
    {
        MyForm form = Application.OpenForms[0] as MyForm;
        Debug.Assert(form != null);

        SendOrPostCallback callback = delegate
        {
            form.Counter++;
        };
        form.MySynchronizationContext.Send(callback, null);
    }
}
static class Program
{
    static void Main()
    {
        ServiceHost host = new ServiceHost(typeof(MyService));
        host.Open();

        Application.Run(new MyForm());

        host.Close();
    }
}

```



例8-6演示的MyForm窗体类定义了MySynchronizationContext属性,通过它客户端能够获取窗体的同步上下文。MyForm在构造函数中通过获取当前线程的同步上下文,初始化MySynchronizationContext。由于Main()方法调用了Application.Run(),它会触发消息通道,因此线程就拥有了一个同步上下文。MyForm还定义了Counter属性,负责计算Windows窗体标签,然后更新其值。Counter属性必须被拥有窗体的线程访问。MyService服务实现了IncrementLabel()操作。在该操作中,服务通过Application类的静态成员OpenForms集合获取窗体的引用:

```
public class FormCollection : ReadOnlyCollectionBase
{
    public virtual Form this[int index]
    {get;}
    public virtual Form this[string name]
    {get;}
}

public sealed class Application
{
    public static FormCollection OpenForms
    {get;}
    // 其余成员
}
```

一旦IncrementLabel()方法需要更新窗体,它就会通过MySynchronizationContext访问同步上下文,并调用Send()方法。更新Counter值的匿名方法会被提供给Send()方法,作为它的参数值。例8-6是例8-4所示的编程模型的一个具体实例。与之相同的是,例8-6使用的技术具有同样的缺陷,即服务与窗体之间的强耦合。如果服务需要更新多个控件,就会导致编程模型趋向复杂。用户界面布局、窗体控件以及所需的行为发生的任何改变,都会导致服务代码的重大改变。

## 安全控件

更好的办法是将Windows窗体同步上下文的交互封装到安全控件中,或者封装到窗体的安全方法中,实现它们与服务之间的解耦,从而简化整个编程模型。例8-7给出了SafeLabel控件的实现代码,它继承了Label类,并提供了对它的Text属性的线程安全访问。SafeLabel继承了Label控件具有的所有设计时可视化功能,并与Visual Studio集成。我们只修改了它需要安全访问的属性。

例8-7: 封装同步上下文

```
public class SafeLabel : Label
{
    SynchronizationContext m_SynchronizationContext =
        SynchronizationContext.Current;
    override public string Text
    {
```

```
set
{
    SendOrPostCallback setText = delegate(object text)
    {
        base.Text = text as string;
    };
    m_SynchronizationContext.Send(setText,value);
}
get
{
    string text = String.Empty;
    SendOrPostCallback getText = delegate
    {
        text = base.Text;
    };
    m_SynchronizationContext.Send(getText,null);
    return text;
}
}
```

通过创建 SafeLabel 对象，保存了它的同步上下文。SafeLabel 重写 (Override) 了基类的 Text 属性，并在 get 和 set 访问器中使用了一个匿名方法，将调用发送到正确的 UI 线程。注意，在 get 访问器中，正如之前讨论的那样，它使用了一个外部变量，用来返回 Send() 方法的值。使用 SafeLabel 可以简化例 8-6 的代码，如例 8-8 所示。

#### 例 8-8：使用安全控件

```
class MyForm : Form
{
    Label m_CounterLabel;

    public MyForm()
    {
        InitializeComponent();
    }
    void InitializeComponent()
    {
        ...
        m_CounterLabel = new SafeLabel();
        ...
    }
    public int Counter
    {
        get
        {
            return Convert.ToInt32(m_CounterLabel.Text);
        }
        set
        {
            m_CounterLabel.Text = value.ToString();
        }
    }
}
```

```
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IFormManager
{
    public void IncrementLabel()
    {
        MyForm form = Application.OpenForms[0] as MyForm;
        Debug.Assert(form != null);

        form.Counter++;
    }
}
```

注意，在例 8-8 中服务可以直接访问窗体：

```
form.Counter++;
```

创建的 form 对象是一个普通的窗体对象。例 8-8 是例 8-5 所示的编程模型的一个具体实例。

---

**注意：**在本书附带的源代码中，程序集 *ServiceModelEx.dll* 不仅实现了 *SafeLabel* 控件，还包括其他常用控件，例如 *SafeButton*、*SafeListBox*、*SafeProgressBar*、*SafeStatusBar* 和 *SafeTextBox*。

---

## 服务同步上下文

目前提及的编程技术主要指服务或资源的开发者在正确的线程上访问资源。如果服务能够建立自身与专门的同步上下文之间的关联，无疑更佳。因为 WCF 就能据此检测上下文，自动将工作线程的调用封送到服务同步上下文。WCF 支持这样的实现。我们可以要求 WCF 维护所有特定宿主的服务实例与同步上下文之间的关联关系。*ServiceBehavior* 特性提供了 *Boolean* 属性 *UseSynchronizationContext*，如下所示：

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : ...
{
    public bool UseSynchronizationContext
    {get;set;}
    // 更多成员
}
```

当宿主被打开时，服务的类型、服务的宿主以及同步上下文之间的关联关系是被锁定的。如果线程打开的宿主拥有一个同步上下文，且 *UseSynchronizationContext* 属性的值被设置为 *true*，WCF 就会为同步上下文以及托管在宿主上的服务的所有实例创建一个关联关系。WCF 会自动将所有传入的调用封送到服务的同步上下文。所有线程的特定信

息都存储在 TLS 中，例如客户端的事务信息或安全信息（第 10 章讨论），都会被准确地封送到同步上下文中。

如果 `UseSynchronizationContext` 属性的值被设置为 `false`，那么不管打开的线程是否拥有同步上下文，服务与同步上下文之间都不会存在关联关系。与之相似，即使 `UseSynchronizationContext` 属性的值被设置为 `true`，如果打开的线程没有同步上下文，服务与同步上下文之间也不会存在关联关系。

`UseSynchronizationContext` 属性的默认值为 `true`，因此下列的定义是等效的：

```
[ServiceContract]
interface IMyContract
{...}

class MyService : IMyContract
{...}

[ServiceBehavior(UseSynchronizationContext = true)]
class MyService : IMyContract
{...}
```

## 在 UI 线程上托管服务

`UseSynchronizationContext` 的一个经典用法是允许服务直接更新用户界面控件以及窗体，而无需借助如例 8-6 和 8-7 所示的技术。如果提供了特定的宿主与 UI 线程的所有服务实例之间的关联关系，WCF 就能够大大地简化 UI 的更新。为此，我们应该在 UI 线程上托管服务，因为 UI 线程创建了服务需要交互的窗体或控件。既然窗体的同步上下文是在 `Application.Run()` 方法初始化消息通道期间建立的，同时 `Application.Run()` 方法又是用于阻塞的操作（它还负责创建窗体），因此不管宿主是在 `Application.Run()` 运行前还是运行后被打开，都无关紧要。如果是在 `Application.Run()` 运行前打开宿主，仍然不会包含同步上下文；如果在之后运行，则通常会关闭应用程序。一个简便办法是在装载窗体之前，让服务需要交互的窗体去打开宿主，如例 8-9 所示。

### 例 8-9：通过窗体托管服务

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{...}

partial class HostForm : Form
{
    ServiceHost m_Host;

    public HostForm()
    {
        InitializeComponent();
```

```

        m_Host = new ServiceHost(typeof(MyService));

        m_Host.Open();
    }
    void OnFormClosed(object sender, EventArgs e)
    {
        m_Host.Close();
    }
}
static class Program
{
    static void Main()
    {
        Application.Run(new HostForm());
    }
}

```

例8-9的服务默认使用宿主获得的任意一个同步上下文。窗体 HostForm 在成员变量中存储了服务宿主，因此窗体能够在关闭自身的同时关闭服务。HostForm 的构造函数已经拥有了一个同步上下文，当它打开宿主时，同时会建立与同步上下文之间的关联关系。

## 访问窗体

即使例8-9中的服务通过窗体托管，服务实例仍然需要应用程序的专有机制去访问窗体。如果服务实例需要更新多个窗体，可以使用 Application.OpenForms 集合（参见例8-6）去查找正确的窗体。一旦服务拥有了 form 对象，就可以随心所欲地直接访问它，而不用像例8-6那样使用封送的方式：

```

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IFormManager
{
    public void IncrementLabel()
    {
        HostForm form = Application.OpenForms[0] as HostForm;
        Debug.Assert(form != null);
        form.Counter++;
    }
}

```

我们也可以在静态变量中存储要使用的窗体引用。使用全局变量带来的问题是，如果使用多个UI线程将消息传递给相同窗体类型的不同实例，就不能针对每个窗体类型使用单个的静态变量。我们需要为使用的每个线程分配一个静态变量，从而导致事情趋于复杂。

相反，窗体（或多个窗体）能够将自身的引用存储在 TLS 中，使得服务实例能够访问和获取存储的引用。然而，使用 TLS 会使得编程模型不具备类型安全。改善的途径是使用线程相关的静态变量。默认情况下，应用程序域中的所有线程都可以访问静态变量。通过线程相关的静态变量，在应用程序域中的每个线程都能获得静态变量存储的自身的副

本。我们可以使用 `ThreadStaticAttribute` 将静态变量标记为线程相关。线程相关的静态变量总是线程安全的，因为它们只能够被单线程访问，而且每个线程获取的只是静态变量自身的副本。线程相关的静态变量被存储在 TLS 中，它们提供了类型安全的基于 TLS 的简化的编程模型。例 8-10 演示了这一技术。

例 8-10：在线程相关的静态变量中存储窗体的引用

```
partial class HostForm : Form
{
    Label m_CounterLabel;
    ServiceHost m_Host;

    [ThreadStatic]
    static HostForm m_CurrentForm;

    public static HostForm CurrentForm
    {
        get
        {
            return m_CurrentForm;
        }
        set
        {
            m_CurrentForm = value;
        }
    }

    public int Counter
    {
        get
        {
            return Convert.ToInt32(m_CounterLabel.Text);
        }
        set
        {
            m_CounterLabel.Text = value.ToString();
        }
    }

    public HostForm()
    {
        InitializeComponent();

        CurrentForm = this;

        m_Host = new ServiceHost(typeof(MyService));
        m_Host.Open();
    }

    void OnFormClosed(object sender, EventArgs e)
    {
        m_Host.Close();
    }
}

[ServiceContract]
interface IFormManager
```

```
{
    [OperationContract]
    void IncrementLabel();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IFormManager
{
    public void IncrementLabel()
    {
        HostForm form = HostForm.CurrentForm;
        form.Counter++;
    }
}
static class Program
{
    static void Main()
    {
        Application.Run(new HostForm());
    }
}
```

窗体 HostForm 将自身的引用存储在线程相关的静态变量 `m_CurrentForm` 中。服务访问静态属性 `CurrentForm`，可以获取在 UI 线程中 HostForm 实例的一个引用。

## 多个 UI 线程

服务宿主进程实际上能够拥有多个 UI 线程，每个线程都会将消息传递到它自身的窗体集中。例如，像安装程序这样与 UI 紧密相关的应用程序，会竭力避免让多个窗体共享一个单独的 UI 线程，同时也避免让多个窗体托管服务。因为当 UI 线程正在处理一个服务调用（或复杂的 UI 更新）时，并非所有的窗体都要响应。既然每个宿主都建立了一个服务同步上下文，如果拥有多个 UI 线程，我们就需要为每个 UI 线程上的相同服务类型打开一个服务宿主实例。因此，对于宿主的服务实例，每个服务宿主都会拥有一个不同的同步上下文。如第 1 章所述，为了相同的服务类型能够拥有多个宿主，我们必须提供每一个具有不同基地址的宿主。最简单的实现办法就是将基地址作为窗体构造函数的参数。在这种情况下，我的建议是让服务终结点使用基地址的相对地址。客户端会调用不同服务终结点的调用，每个终结点根据基地址的样式以及使用的绑定对应不同的宿主。例 8-11 演示了这样的配置。

例 8-11：在多个 UI 线程中托管服务

```
partial class HostForm : Form
{
    public HostForm(string baseAddress)
    {
        InitializeComponent();

        CurrentForm = this;
    }
}
```



```

        m_Host = new ServiceHost(typeof(MyService), new Uri(baseAddress));
        m_Host.Open();
    }
    // 其余代码与例 8-10 相同
}
static class Program
{
    static void Main()
    {
        ParameterizedThreadStart threadMethod = delegate(object baseAddress)
        {
            string address = baseAddress as string;
            Application.Run(new HostForm(address));
        };

        Thread thread1 = new Thread(threadMethod);
        thread1.Start("http://localhost:8001/");

        Thread thread2 = new Thread(threadMethod);
        thread2.Start("http://localhost:8002/");
    }
}
/* MyService 与例 8-10 相同 */

//////////////////// 宿主配置文件 //////////////////////
<services>
  <service name = "MyNamespace.MyService">
    <endpoint
      address = "MyService"
      binding = "basicHttpBinding"
      contract = "IFormManager"
    />
  </service>
</services>
//////////////////// 客户端配置文件 //////////////////////
<client>
  <endpoint name = "Form A"
    address = "http://localhost:8001/MyService/"
    binding = "basicHttpBinding"
    contract = "IFormManager"
  />
  <endpoint name = "Form B"
    Address = "http://localhost:8002/MyService/"
    binding = "basicHttpBinding"
    contract = "IFormManager"
  />
</client>

```

例 8-11 中的 Main() 方法启用了两个 UI 线程，每个线程都包含了 HostForm 实例。每个窗体实例都接收了一个基地址作为构造函数的参数，并轮流提供给窗体自身的宿主实例。一旦宿主被打开，就建立了与 UI 线程同步上下文之间的关联关系。从客户端到对应基地址的调用会被发送到各自的 UI 线程中。注意，服务使用 BasicHttpBinding 绑定公

开了基于HTTP的终结点。如果服务公开的第二个终结点是基于TCP绑定的，则应该为宿主提供TCP基地址。

## 将窗体定义为一个服务

在UI线程上运行WCF服务的主要目的在于服务需要更新UI或者窗体。问题是服务应该怎样获得窗体的引用呢？前面的示例演示的技术与实现原理虽然能够达成这一目的，但是如果我们将窗体自身当成服务实现托管，无疑会更加简单。为此，窗体必须被定义为单例服务。原因在于单例是唯一支持开发者为WCF提供一个活动实例并托管在宿主中的实例模式。此外，开发者既不需要一个只存在于客户端调用期间的窗体（通常这样会更简单），也不需要一个只有单个客户端才能够建立会话以及更新的窗体。当一个窗体同时又是服务时，将它定义为单例服务才是实例模式的最佳选择。例8-12定义了这样的服务。

例8-12：将窗体定义为单例服务

```
[ServiceContract]
interface IFormManager
{
    [OperationContract]
    void IncrementLabel();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
partial class MyForm : Form, IFormManager
{
    Label m_CounterLabel;
    ServiceHost m_Host;

    public MyForm()
    {
        InitializeComponent();
        m_Host = new ServiceHost(this);
        m_Host.Open();
    }
    void OnFormClosed(object sender, EventArgs args)
    {
        m_Host.Close();
    }
    public void IncrementLabel()
    {
        Counter++;
    }
    public int Counter
    {
        get
        {
            return Convert.ToInt32(m_CounterLabel.Text);
        }
        set
        {

```

```
        m_CounterLabel.Text = value.ToString();
    }
}
```

MyForm 实现了 IFormManager 契约，并被配置为 WCF 单例服务。与前一致，MyForm 同样将 ServiceHost 对象定义为成员变量。在 MyForm 创建宿主对象时，使用了宿主的构造函数，接收了一个 object 引用，参见第 4 章的示例。MyForm 将自身作为 object 对象进行传递。它在创建窗体时打开宿主，而在关闭窗体时关闭宿主。作为客户端调用的结果，窗体控件的更新可以直接通过访问，因为窗体运行在它自身的同步上下文中。

## FormHost<F> 类

可以使用我定义的 FormHost<F> 类简化例 8-12 中的代码，实现过程的自动化。FormHost<F> 类的定义如下：

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
public abstract class FormHost<F> : Form where F : Form
{
    public FormHost(params string[] baseAddresses);

    protected ServiceHost<F> Host
    {get;set;}
}
```

使用 FormHost<F>，例 8-12 可以简化为：

```
partial class MyForm : FormHost<MyForm>, IFormManager
{
    Label m_CounterLabel;

    public MyForm()
    {
        InitializeComponent();
    }
    public void IncrementLabel()
    {
        Counter++;
    }
    public int Counter
    {
        get
        {
            return Convert.ToInt32(m_CounterLabel.Text);
        }
        set
        {
            m_CounterLabel.Text = value.ToString();
        }
    }
}
```

**注意：**Windows窗体设计器不能呈现(Rendering)一个具有抽象基类的窗体，更不能呈现使用泛型的窗体类。我们必须将基类修改为支持可视化编辑的Form类，同时为了便于调试再将其转换为FormHost<F>类型。这些类型之间的差异给开发者带来极大的不便，希望在将来的版本能够解决这一问题。

例8-13演示了FormHost<F>类的实现。

**例8-13：实现FormHost<F>**

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
public abstract class FormHost<F> : Form where F : Form
{
    ServiceHost<F> m_Host;

    protected ServiceHost<F> Host
    {
        get
        {
            return m_Host;
        }
        set
        {
            m_Host = value;
        }
    }

    public FormHost(params string[] baseAddresses)
    {
        m_Host = new ServiceHost<F>(this as F, baseAddresses);

        Load += delegate
        {
            if(Host.State == CommunicationState.Created)
            {
                Host.Open();
            }
        };

        FormClosed += delegate
        {
            if(Host.State == CommunicationState.Opened)
            {
                Host.Close();
            }
        };
    }
}
```

FormHost<F>类是一个被配置为单例服务的抽象类。它是一个泛型类，接受单个类型参数F。F被约束为Windows窗体类类型Form。FormHost<F>使用了我实现的ServiceHost<T>作为成员变量，指定F为宿主的类型参数。FormHost<F>为派生的窗体类提供了对宿主的访问，主要用于高级配置，因此Host属性被标记为protected。

FormHost<F>的构造函数创建了宿主,但没有打开它。原因在于子窗体可能需要执行某些初始化宿主的操作,例如配置限流(Throttle)。只能在打开宿主之前执行初始化。子类应该将初始化放到它自己的构造函数中:

```
public MyForm()
{
    InitializeComponent();
    Host.SetThrottle(10,20,1);
}
```

为了支持这样的操作,构造函数使用了一个匿名方法订阅窗体的Load事件。事件首先通过子窗体确认宿主是否打开,如果没有,则打开宿主。相似的,构造函数还订阅了窗体的FormClosed事件,通过它关闭宿主。

## UI 线程与并发管理

无论何时使用UI线程(或者使用一个单线程关联的同步上下文)托管服务,都有可能导致死锁。例如,下面的设置肯定会造成死锁:一个Windows Forms应用程序托管了一个UseSynchronizationContext设置为true的服务,同时它还建立了UI线程关联度。然后,Windows Forms应用程序基于它的其中一个终结点调用进程内服务。当WCF将消息传递给UI线程去调用服务时,对服务的调用就会阻塞UI线程。由于UI线程被阻塞,因而无法处理消息,从而导致死锁。

另外一种可能发生死锁的情形是Windows Forms应用程序托管了一个UseSynchronizationContext设置为true的服务,同时它还建立了UI线程关联度。服务接收了一个来自远程客户端的调用。调用会被封送给UI线程,并在该线程中执行这一调用。如果允许服务执行传出到其他服务的调用,而这一向外的调用又试图更新UI或者将调用返回给服务的终结点,由于所有与终结点关联的服务(不考虑服务的实例模式)共享了相同的UI线程,从而可能会导致死锁。相似的,如果服务被配置为重入,并将调用返回给它的客户端,也会造成死锁。如果回调链试图更新UI或者进入服务,由于重入必须被封送给被阻塞的UI线程,因而也会导致死锁。

### UI 响应速度

每次客户端调用托管在UI线程中的服务时,调用消息都会被转换为Windows消息。实际上,这些调用会执行在UI线程上。相同的线程既负责更新UI并持续响应用户的输入,同时又负责更新与应用程序状态有关的UI和用户。UI线程在处理服务调用时,不会处理UI消息。因此,我们应避免服务操作的执行耗时过长,因为它会严重地降低UI的响应速度。开发者可以将服务操作中的Windows消息放入通道,从而在一定程度上减轻它带来的影响。方法是显式调用静态方法Application.DoEvents(),处理所有放入队列

中的 Windows 消息，或者通过调用方法 `MessageBox.Show()` 传递一部分消息，而不是所有的队列消息。刷新 UI 的缺点是它可能会将客户端调用分发给放入到队列中的服务实例，从而导致不必要的重入或者死锁。

由于 UI 的响应速度与服务的并发模式（接下来会讨论）有关，即使服务调用的时间非常短暂，但如果要求客户端立即把它们中的一部分分发给服务，事情会否变得更加糟糕？将这些调用依次放入 Windows 消息队列，然后处理它们，目的是为了节省时间。此时，所有调用都不会更新 UI。无论何时，在 UI 线程中托管服务都会仔细检查调用的持续时间与频率，以了解降低 UI 响应速度的结果是否可以接受。能否接受的标准与具体的应用程序有关。通常情况下，大多数用户不会介意低于半秒钟的延迟；一旦超过四分之三秒，用户就会注意到 UI 的延迟。如果延迟超过了一秒，就会给用户带来不愉快的体验。如果是这样的情况，我们就需要考虑在多个 UI 线程上托管一部分 UI（以及关联服务），正如前面所阐释的那样。由于具有多个 UI 线程，就可以最大化地提高响应速度，因为当一个线程正忙于响应一个客户端调用时，其余线程仍然可以更新它们的窗体与控件。如果应用程序不能使用多个 UI 线程，那么当服务调用的处理导致过低的 UI 响应速度时，我们就应该检查服务的操作到底做了什么，又是什么操作导致了延迟。通常情况下，更新 UI 不是导致延迟的唯一原因，如果执行的操作耗时过长，例如调用其他服务，或者诸如图形处理这样的密集型计算操作，都可能是导致延迟的罪魁祸首。因为 UI 线程托管了服务，WCF 在 UI 线程上会执行所有的操作，而不仅仅是与 UI 直接交互的关键部分。如果情形确实如此，就应该将 `UseSynchronizationContext` 属性值设置为 `false`，从而禁用 UI 的线程关联度。

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
                  UseSynchronizationContext = false)]
class MyService : IMyContract
{
    public void MyMethod()
    {
        Debug.Assert(Application.MessageLoop == false);
        // 其余实现
    }
}
```

（我们甚至可以设置断言，以判断线程在执行服务调用时是否具有消息循环。）在传入的工作线程上执行耗时较长的操作时，只有在需要时才应该使用安全控件（例如 `SafeLabel`），通过它将调用封送给 UI 线程。这一做法的缺点在于它是一种复杂的编程模型：服务不能是窗口或窗体自身（利用简化的 `FormHost<F>` 类），我们需要一种绑定窗体的方法，服务的开发者必须与 UI 开发者协作，才能确保它们使用安全控件，或者提供对窗体同步上下文的访问。



## UI 线程与并发模式

拥有 UI 线程关联度的服务本身是线程安全的，因为只有 UI 线程能够调用它的实例。既然只有一个单线程能够访问实例，因而定义的实例就是线程安全的。配置为 `ConcurrencyMode.Single` 的服务不需要额外添加安全性，因为服务在任何时候都是单线程的。如果将服务配置为 `ConcurrencyMode.Single`，首先会通过实例锁将并发的客户端调用放入到队列中，同时依次将它们分发给服务的消息循环。这些客户端调用可能会与其他 UI 的 Windows 消息交叉存取，UI 线程又能在客户端调用处理与用户交互之间交替转换，因而，`ConcurrencyMode.Single` 能够最大程度地改善 UI 的响应速度。如果将服务配置为 `ConcurrencyMode.Multiple`，一旦客户端的调用到达通道，就会被分发给服务的消息循环，然后依次调用。这就可能导致成批量的客户端调用被组合在一起，放入到 Windows 消息队列中。当 UI 线程处理这些调用时，UI 响应速度会非常缓慢。因此，`ConcurrencyMode.Multiple` 会严重影响 UI 响应速度。如果配置为 `ConcurrencyMode.Reentrant`，服务不一定会重入，死锁仍然可能发生，正如本节一开始所阐述的那样。显然，UI 线程关联度的一种最佳实践就是将服务配置为 `ConcurrencyMode.Single`。之所以避免使用 `ConcurrencyMode.Multiple`，是因为它会严重地影响了响应速度。至于避免使用 `ConcurrencyMode.Reentrant` 的原因，则是因为它不具备安全性。

## 定制服务同步上下文

同步上下文是一种通用的模式，.NET 2.0 与 WCF 只实现了它单独发生的情况，即 Windows 窗体同步上下文。开发一个定制的服务同步上下文包含两个方面。首先是实现定制的同步上下文，然后再利用声明方式将它安装或应用到服务上。实现定制同步上下文的方法与 WCF 无关，因此不在本书讨论范围之内。在本节中，我使用了现有的 `AffinitySynchronizer` 类，它的定义如下：

```
public class AffinitySynchronizer : SynchronizationContext, IDisposable
{
    public AffinitySynchronizer();
    public AffinitySynchronizer(string threadName);
    public void Dispose();
}
```

`AffinitySynchronizer` 在相同的私有工作线程上执行所有被封送的调用。当隶属的线程打开一个宿主时，服务的所有实例都会在工作线程上执行，而与它们的实例模式、并发模式、终结点以及契约无关。`AffinitySynchronizer` 的实现包含在本书附带的源代码中。概括地讲，`AffinitySynchronizer` 类创建了一个工作线程，并维持一个包含工作项的同步队列。出于调试和日志记录的目的，我们甚至可以为



AffinitySynchronizer类提供一个线程名,作为构造函数的参数。如果不分配该参数值,则工作线程名默认为“AffinitySynchronizer Worker Thread”。队列中的每个工作项都包含了一个 SendOrPostCallback 类型的委托。调用 Post() 或 Send() 方法时, AffinitySynchronizer 类会将委托对象传递给工作项的构造函数,然后在创建工作项对象之后,将它传递给队列。工作线程会监控该队列。只要队列中存在工作项,工作线程就会从队列中取出第一个工作项,并调用委托。若要终止工作线程,可以释放 AffinitySynchronizer 实例。相似地,我们可以开发一个定制同步上下文,负责将所有传入的调用封送到私有的、空间小的线程池中。只有线程池中的线程才允许执行调用。

实现定制同步上下文的第二方面是安装同步上下文。作为 WCF 可扩展性的重要体现,实现它的一个关键点是考虑安装这种扩展的方式与时机。

## 具有线程关联度的服务

如果 WCF 服务创建了需要线程关联度的资源,并与这样的资源交互,例如 TLS,则 AffinitySynchronizer 类就有了用武之地。使用 AffinitySynchronizer 对象的服务总是线程安全的,因为只有特定的 AffinitySynchronizer 对象的内部工作线程能够调用它。如果服务被配置为 ConcurrencyMode.Single,由于它在任何情况下都是单线程的,所以服务无需增加额外的线程安全。开发者能够获取并发调用的双向队列:所有到服务的并发调用首先进入锁定的队列,然后在某个时刻被分发给工作线程。如果服务被配置为 ConcurrencyMode.Multiple,则分发调用到工作线程的速度,与它们到达的速度保持一致,然后放入队列依次调用,而不是并发调用。注意,如果我们使用一个定制同步上下文将它的调用封送给线程池,而不是单个线程,那么 ConcurrencyMode.Multiple 就能提供最佳的系统性能。如果服务被配置为 ConcurrencyMode.Reentrant,服务并不会重入,因为传入的重入调用会被放入到队列中,并且会导致死锁。如果要使用 AffinitySynchronizer,那么推荐的并发模式为 ConcurrencyMode.Single。

## 安装服务同步上下文

例 8-14 演示了如何在打开宿主之前安装 AffinitySynchronizer,这样,服务实例就能够运行在相同的线程中。

例 8-14: 安装 AffinitySynchronizer

```
SynchronizationContext synchronizationContext = new AffinitySynchronizer();
SynchronizationContext.SetSynchronizationContext(synchronizationContext);

using(synchronizationContext as IDisposable)
{
```

```
ServiceHost host = new ServiceHost(typeof(MyService));
host.Open();
/* 执行某些阻塞操作 */
host.Close();
}
```

要将同步上下文添加到当前线程中，应该调用 `SynchronizationContext` 的静态方法 `SetSynchronizationContext()`。一旦宿主被打开，宿主会使用提供的同步上下文。注意在例 8-14 中，实例在关闭宿主之后释放了 `AffinitySynchronizer` 对象，从而关闭了当前使用的工作线程。

我们可以将安装 `AffinitySynchronizer` 的过程封装到定制宿主 `ServiceHost<T>` 中，以此来简化例 8-14 中的代码：

```
public class ServiceHost<T> : ServiceHost
{
    public void SetThreadAffinity(string threadName);
    public void SetThreadAffinity();
    // 更多成员
}
```

使用 `SetThreadAffinity()` 方法可以直接添加 `AffinitySynchronizer`：

```
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.SetThreadAffinity();

host.Open();

/* 执行某些阻塞操作 */

host.Close();
```

例 8-15 给出了 `SetThreadAffinity()` 方法的实现。

例 8-15：添加线程关联度以支持 `ServiceHost<T>`

```
public class ServiceHost<T> : ServiceHost
{
    AffinitySynchronizer m_AffinitySynchronizer;

    public void SetThreadAffinity(string threadName)
    {
        if (State == CommunicationState.Opened)
        {
            throw new InvalidOperationException("Host is already opened");
        }
        m_AffinitySynchronizer = new AffinitySynchronizer(threadName);
        SynchronizationContext.SetSynchronizationContext(m_AffinitySynchronizer);
    }
    public void SetThreadAffinity()
    {
        SetThreadAffinity("Executing all endpoints of " + typeof(T));
    }
}
```

```

    }
    protected override void OnClosing()
    {
        using (m_AffinitySynchronizer)
        {}
        base.OnClosing();
    }
    // 更多成员
}

```

`ServiceHost<T>` 定义了一个类型为 `AffinitySynchronizer` 的成员变量，同时还提供了两个版本的 `SetThreadAffinity()` 方法。带参方法接收一个线程名，赋给 `AffinitySynchronizer` 的工作线程；无参方法 `SetThreadAffinity()` 则调用了带参的 `SetThreadAffinity()` 方法，线程名则是根据宿主服务类型推断获得的值，例如“Executing all endpoints of MyService.” `SetThreadAffinity()` 方法首先检查宿主是否还未打开，因为我们只能在宿主打开之前添加同步上下文。如果宿主还未打开，`SetThreadAffinity()` 方法就会创建一个新的 `AffinitySynchronizer` 对象，构造参数则是它使用的线程名，并将新创建的对象添加到当前线程中。最后，`ServiceHost<T>` 重写了基类的方法 `OnClosing()`，目的在于调用 `AffinitySynchronizer` 成员的释放操作，关闭它的工作线程。如果没有调用 `SetThreadAffinity()` 方法，`AffinitySynchronizer` 成员值则为 `null`，因而 `OnClosing()` 方法使用了 `using` 语句，在调用 `Dispose()` 方法之前从内部检查是否分配了 `null` 值。

## ThreadAffinityBehavior 特性

前一节介绍了在不考虑服务配置的情况下，如何通过宿主安装 `AffinitySynchronizer`。然而，如果设计的服务总是需要在相同的线程上执行，那么最好是不要影响宿主以及打开服务的线程。可以使用我们定义的 `ThreadAffinityBehaviorAttribute` 特性：

```

[AttributeUsage(AttributeTargets.Class)]
public class ThreadAffinityBehaviorAttribute : Attribute,
                                             IContractBehavior, IServiceBehavior
{
    public ThreadAffinityBehaviorAttribute(Type serviceType);
    public ThreadAffinityBehaviorAttribute(Type serviceType, string threadName);
    public string ThreadName
    { get; set; }
}

```

顾名思义，`ThreadAffinityBehavior` 特性提供了一个本地行为，要求所有的服务实例总是运行在相同的线程上。`ThreadAffinityBehavior` 特性的内部使用了我定义的 `AffinitySynchronizer` 类。应用该特性时，需要提供服务类型以及可选的线程名：

```

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
[ThreadAffinityBehavior(typeof(MyService))]

```

```
class MyService : IMyContract
{...}
```

默认的线程名为“Executing all endpoints of <service type>.”

`ThreadAffinityBehavior` 特性是一个定制的契约行为，因为它实现了第 5 章介绍的 `ISyncBehavior` 接口。`ISyncBehavior` 定义了 `ApplyDispatchBehavior()` 方法，允许开发者对单独的终结点分发器的运行时（Runtime）施加影响，并设置它的同步上下文：

```
public interface IContractBehavior
{
    void ApplyDispatchBehavior(ContractDescription description,
                               ServiceEndpoint endpoint,
                               DispatchRuntime dispatch);

    // 更多成员
}
```

每个终结点都拥有自己的分发器，而每个分发器又都拥有自己的同步上下文，因此特性会被实例化，每个终结点也会调用 `ApplyDispatchBehavior()` 方法。例 8-16 演示了 `ThreadAffinityBehavior` 特性的主要实现。

### 例 8-16: 实现 ThreadAffinityBehaviorAttribute

[illegible]

```

void IContractBehavior.Validate(...)
{
}
void IContractBehavior.AddBindingParameters(...)
{
}
void IContractBehavior.ApplyClientBehavior(...)
{
}
void IServiceBehavior.Validate(ServiceDescription description,
                               ServiceHostBase serviceHostBase)
{
    serviceHostBase.Closed += delegate
    {
        ThreadAffinityHelper.CloseThread(m_ServiceType);
    };
}
void IServiceBehavior.AddBindingParameters(...)
{
}
void IServiceBehavior.ApplyDispatchBehavior(...)
{
}
}

```

ThreadAffinityBehavior 特性的构造函数存储了提供的服务类型与线程名。

**注意：**例 8-16 中的 ApplyDispatchBehavior() 方法使用了空合并运算符 ?? (C# 2.0 中引入)。如果线程名为空，就会分配一个线程名，表达式为：

```

m_ThreadName = m_ThreadName ?? "Executing endpoints
                               of " + m_ServiceType;

```

它相当于以下内容的缩写：

```

if(m_ThreadName == null)
{
    m_ThreadName = "Executing endpoints of " +
                  m_ServiceType;
}

```

ThreadAffinityBehavior 特性作为一个高层的协调者，会在 ApplyDispatchBehavior() 方法的实际实现中，调用静态辅助类 ThreadAffinityHelper 的 ApplyDispatchBehavior() 方法：

```

public static class ThreadAffinityHelper
{
    internal static void ApplyDispatchBehavior(Type type, string threadName,
                                              DispatchRuntime dispatch)
    public static void CloseThread(Type type);
}

```

ThreadAffinityHelper 类同样定义了静态方法 CloseThread()，通过该方法关闭与服务类型同步上下文关联的工作线程。ThreadAffinityBehavior 同时还属于服务行为。它实现了 IServiceBehavior 接口，因此在 Validate() 方法的实现中，它能

够获取服务宿主的引用，并利用匿名方法订阅 `Closed` 事件。匿名方法通过调用 `ThreadAffinityHelper.CloseThread()` 方法以及提供的服务类型关闭工作线程。例 8-17 演示了 `ThreadAffinityHelper` 类的实现。

#### 例 8-17：实现 `ThreadAffinityHelper`

```
public static class ThreadAffinityHelper
{
    static Dictionary<Type, AffinitySynchronizer> m_Contexts =
        new Dictionary<Type, AffinitySynchronizer>();

    [MethodImpl(MethodImplOptions.Synchronized)]
    internal static void ApplyDispatchBehavior(Type type, string threadName,
        DispatcherRuntime dispatch)
    {
        Debug.Assert(dispatch.SynchronizationContext == null);

        if(m_Contexts.ContainsKey(type) == false)
        {
            m_Contexts[type] = new AffinitySynchronizer(threadName);
        }
        dispatch.SynchronizationContext = m_Contexts[type];
    }

    [MethodImpl(MethodImplOptions.Synchronized)]
    public static void CloseThread(Type type)
    {
        if(m_Contexts.ContainsKey(type))
        {
            m_Contexts[type].Dispose();
            m_Contexts.Remove(type);
        }
    }
}
```

`DispatcherRuntime` 类提供了 `SynchronizationContext` 类型的属性 `ThreadAffinityHelper`，用来为分发器分配一个同步上下文：

```
public sealed class DispatcherRuntime
{
    public SynchronizationContext SynchronizationContext
    {get;set;}
    // 更多成员
}
```

在分配之前，`ThreadAffinityHelper` 验证了分发器是否包含其他同步上下文，因为它可能会导致一些无法解决的冲突。对于那些为服务类型提供了同步上下文的相同实例的所有终结点而言，`ThreadAffinityHelper` 的功能与它们的所有分发器有关。为保证同步上下文与类型之间的一对一关联，`ThreadAffinityHelper` 在内部定义了一个静态的字典集合，建立了同步上下文与类型之间的映射关系。在 `ApplyDispatchBehavior()`



方法中, ThreadAffinityHelper会检查字典集合中是否已经包含了现有类型对应的同步上下文。如果没有找到匹配值, ThreadAffinityHelper则根据线程名创建一个新的同步上下文, 并将它添加到字典集合中。然后, 它会在字典集合中根据类型查找同步上下文, 并将它分配给分发器。CloseThread()方法将参数传递的类型值作为关键字查找与同步上下文关联的字典集合, 然后释放它, 从而关闭线程。所有访问静态字典集合的操作都是同步的, 我们使用了MethodImpl特性的MethodImplOptions.Synchronized标志, 以声明方式对方法进行设置。

## 回调与客户端安全

客户端接收并发调用时, 会发生如下几种情形。如果客户端为多个服务提供了一个回调的引用, 这些服务就能够以并发的方式将调用返回给客户端。但是, 即使是单个的回调引用, 服务也有可能启动多线程, 并使用这些线程去调用该单一引用。双向回调在工作线程中进入客户端, 如果不以同步方式执行并发, 就有可能影响客户端的状态。客户端必须同步访问它自身在内存中的状态, 而且还要同步访问回调线程可能访问的所有资源。与服务相似, 回调客户端既可以使用手动方式, 也可以使用声明方式实现同步。第6章介绍的 CallbackBehavior 特性提供了 ConcurrencyMode 和 UseSynchronizationContext 属性:

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class CallbackBehaviorAttribute : Attribute, ...
{
    public ConcurrencyMode ConcurrencyMode
    {get;set;}
    public bool UseSynchronizationContext
    {get;set;}
}
```

这些属性的默认值与 ServiceBehavior 特性的值相同, 执行方式也是相似的。例如, ConcurrencyMode 属性的默认值为 ConcurrencyMode.Single, 因此以下的两个定义是等效的:

```
class MyClient : IMyContractCallback
{...}

[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Single)]
class MyClient : IMyContractCallback
{...}
```

## ConcurrencyMode.Single 回调

如果回调被配置为 ConcurrencyMode.Single (默认值), 那么在一段时间内只允许一



个回调进入回调对象。不同于服务的是回调对象通常依赖于 WCF。当服务实例被 WCF 拥有时，只有工作线程会被访问服务实例的 WCF 分发，回调对象同时会与本地的客户端线程交互。使用 `ConcurrencyMode.Single` 时，这些客户端线程并不知道与回调对象关联的同步锁。`ConcurrencyMode.Single` 针对回调对象所要做的就是通过 WCF 线程进行串行访问。因此，我们必须手动地实现对回调状态以及其他被回调方法访问的资源的同步访问，如例 8-18 所示。

例 8-18：手动实现 `ConcurrencyMode.Single` 回调的同步

```
interface IMyContractCallback
{
    [OperationContract]
    void OnCallback();
}
class MyClient : IMyContractCallback, IDisposable
{
    MyContractClient m_Proxy;

    public void CallService()
    {
        InstanceContext callbackContext = new InstanceContext(this);
        m_Proxy = new MyContractClient(callbackContext);
        m_Proxy.DoSomething();
    }
    // 该方法会在某个时刻被一个回调对象和客户端线程调用
    public void OnCallback()
    {
        // 以手动同步的方式访问状态与资源
        lock(this)
        { ... }
    }
    public void Dispose()
    {
        m_Proxy.Close();
    }
}
```

## ConcurrencyMode.Multiple 回调

当回调被配置为 `ConcurrencyMode.Multiple` 时，WCF 允许在回调实例上执行并发调用。这意味着我们需要在回调操作中实现同步访问，因为它们可能会被 WCF 工作线程与客户端线程并发调用，如例 8-19 所示。

例 8-19：手动实现 `ConcurrencyMode.Multiple` 回调的同步

```
[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
class MyClient : IMyContractCallback, IDisposable
{
    MyContractClient m_Proxy;
```

```
public void CallService()
{
    InstanceContext callbackContext = new InstanceContext(this);
    m_Proxy = new MyContractClient(callbackContext);
    m_Proxy.DoSomething();
}
// 该方法会被回调对象和客户端线程并发调用
public void OnCallback()
{
    // 以手动同步的方式访问状态与资源
    lock(this)
    { ... }
}
public void Dispose()
{
    m_Proxy.Close();
}
}
```

## ConcurrencyMode.Reentrant 回调

既然回调对象能够通过 WCF 执行传出的调用，这些调用实际上可能会重新进入回调对象。使用 `ConcurrencyMode.Single` 时，为避免死锁，可以根据需要将回调配置为 `ConcurrencyMode.Reentrant`：

```
[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
class MyClient : IMyContractCallback
{ ... }
```

如果回调对象属于 WCF 传出调用的一部分，那么，将回调配置为重入就可以使得其他服务能够调用回调。

## 回调与同步上下文

与服务调用相似，回调可能需要访问依赖于某个线程或多个线程关联度的资源。此外，为了使用 TLS 或者与 UI 线程交互，回调实例自身也可能需要线程关联度。当回调使用例 8-4 和例 8-5 所示的技术将交互封送到资源的同步上下文时，我们也可以要求 WCF 将 `UseSynchronizationContext` 属性值设置为 `true`，从而关联回调与特定的同步上下文。然而，不同于服务的是，客户端不会使用任何宿主去公开终结点。如果 `UseSynchronizationContext` 属性为 `true`，使用的同步上下文就会在打开代理时被锁定。或者，更常见的是，当客户端发出对使用代理的服务的第一次调用时，如果没有显式地调用 `Open()` 方法，使用的同步上下文同样会被锁定。如果正在调用的客户端线程拥有一个同步上下文，则 WCF 的所有回调都会使用该同步上下文，目的是为了关联客户端终结点与代理。注意，只有在代理上执行的第一次调用（或者调用 `Open()` 方法）

才有可能确定同步上下文。随后的调用即使采用同样的方式，也无法确定同步上下文。如果正在调用的客户端线程没有同步上下文，那么即使将 `UseSynchronizationContext` 属性设置为 `true`，回调仍然没有可用的同步上下文。

## 回调与 UI 同步上下文

如果回调对象运行在 Windows 窗体同步上下文中，或者它需要更新某些 UI，那么我们就必须将回调或更新封送到 UI 线程中（译注 2）。我们可以使用例 8-6 或 8-8 的技术。但是，对于基于回调的 UI 更新而言，更常见的做法是让窗体自身实现回调契约以及更新 UI，如例 8-20 所示。

例 8-20：依赖于 UI 同步上下文实现回调

```
partial class MyForm : Form, IMyContractCallback
{
    MyContractClient m_Proxy;

    public MyForm()
    {
        InitializeComponent();
        InstanceContext callbackContext = new InstanceContext(this);
        m_Proxy = new MyContractClient(callbackContext);
    }
    // 作为 UI 事件进行调用
    public void OnCallService(object sender, EventArgs args)
    {
        m_Proxy.DoSomething(); // 建立同步上下文与代理之间的关联关系
    }
    // 该方法总是运行在线程上
    public void OnCallback()
    {
        // 无需同步和封送
        Text = "Some Callback";
    }
    public void OnClose(object sender, EventArgs args)
    {
        m_Proxy.Close();
    }
}
```

在例 8-20 中，作为 UI 事件而被 UI 线程调用的 `OnCallService()` 方法首先使用了代理。在 UI 同步上下文中调用代理，可以建立同步上下文与代理之间的关联关系，因此回调不需要封送任何调用，就能直接访问和更新 UI。此外，既然只有一个线程（以及与它相同的线程）会执行在同步上下文中，因而可以保证回调能够被同步。

译注 2：如果建立了同步上下文与回调对象之间的关联关系，则不必将回调或更新封送到 UI 线程中。

我们也可以不调用任何操作,直接在窗体的构造函数中通过打开代理显式地建立UI同步上下文的关联关系。如果我们需要将调用分发给工作线程的服务(或者可能采用异步方式,这将在本章末讨论),同时还要让回调进入UI同步上下文,正如例8-21所示,那么这种方式就非常有用。

例 8-21: 显式打开代理以建立同步上下文

```
partial class MyForm : Form, IMyContractCallback
{
    MyContractClient m_Proxy;

    public MyForm()
    {
        InitializeComponent();
        InstanceContext callbackContext = new InstanceContext(this);
        m_Proxy = new MyContractClient(callbackContext);

        // 建立代理与UI同步上下文的关联关系
        m_Proxy.Open();
    }
    // 作为UI事件进行调用
    public void CallService(object sender, EventArgs args)
    {
        ThreadStart invoke = delegate
        {
            m_Proxy.DoSomething();
        };
        Thread thread = new Thread(invoke);
        thread.Start();
    }
    // 该方法总是运行在UI线程上
    public void OnCallback()
    {
        // 无需同步和封送
        Text = "Some Callback";
    }
    public void OnClose(object sender, EventArgs args)
    {
        m_Proxy.Close();
    }
}
```

## UI 线程回调与响应速度

在UI线程上处理回调时,UI自身并不会响应。如果回调被配置为ConcurrencyMode.Multiple,即使开发者执行的是耗时相对较短的回调,仍然会将多个回调依次放入到UI消息队列中,然后立即处理它们,从而导致了响应速度的降低。我们应该避免在UI线程上执行耗时过长的回调处理,并选择将回调配置为ConcurrencyMode.Single,这样,回调对象锁就能够将多个回调放入队列,然后在一段时间内将它们分发给回调对象,启用它们与UI消息的交错处理。

## UI 线程回调与并发管理

配置回调支持 UI 线程的关联关系，可能触发死锁。考虑如下步骤：一个 Windows 窗体客户端建立了一个回调对象（或者客户端自身）与 UI 同步上下文之间的关联关系。然后，客户端调用服务传递回调引用。服务被配置为重入模式，它会调用客户端。由于到达客户端的回调需要在 UI 线程上执行，而该线程又为了等待服务调用的返回而被阻塞，此时就会发生死锁（译注 3）。即使将回调配置为单向操作，仍然无法解决问题，因为单向调用仍然需要首先被封送给 UI 线程。在这种情况下，解决死锁的唯一办法就是关闭回调对 UI 同步上下文的使用，然后使用它的同步上下文手动地以异步方式将更新封送到窗体。例 8-22 演示了这种技术的使用。

### 例 8-22：避免在 UI 线程上的回调死锁

```
//////////////////// 客户端 //////////////////////////////////////
[CallbackBehavior(UseSynchronizationContext = false)]
partial class MyForm : Form, IMyContractCallback
{
    SynchronizationContext m_Context;
    MyContractClient m_Proxy;

    public MyForm()
    {
        InitializeComponent();
        m_Context = SynchronizationContext.Current;
        InstanceContext callbackContext = new InstanceContext(this);
        m_Proxy = new MyContractClient(callbackContext);
    }

    public void CallService(object sender, EventArgs args)
    {
        m_Proxy.DoSomething();
    }

    // 回调运行在工作线程上
    public void OnCallback()
    {
        SendOrPostCallback setText = delegate
        {
            Text = "Manually marshaling to UI thread";
        };
        m_Context.Post(setText, null);
    }

    public void OnClose(object sender, EventArgs args)
```

译注 3：客户端调用服务并传递回调引用。然后服务执行回调以调用客户端。由于调用发生在线程上，线程此时会被阻塞，等待调用返回。此时的调用即执行回调，因为回调对象与线程之间存在关联关系，因此回调对象的执行必须执行在线程上。但是，线程此时已经被阻塞，因而回调对象无法执行。而正是因为回调对象无法执行，导致服务的调用无法返回，则线程将无法解除阻塞，从而导致死锁。

```

    {
        m_Proxy.Close();
    }
}
////////// 服务端 //////////
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void DoSomething();
}
interface IMyContractCallback
{
    [OperationContract]
    void OnCallback();
}
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
class MyService : IMyContract
{
    public void DoSomething()
    {
        IMyContractCallback callback = OperationContext.Current.
            GetCallbackChannel<IMyContractCallback>();
        callback.OnCallback();
    }
}

```

正如例8-22所示，开发者必须使用同步上下文的 `Post()` 方法。而且，无论在任何情况下都不能使用 `Send()` 方法，即使工作线程正在执行回调，而 UI 线程又被传出的调用阻塞。调用 `Send()` 方法会触发死锁，因为它会阻塞线程，直到 UI 线程能够处理请求（译注4）。相似的，例8-22的回调不能使用任何一种安全控件（例如 `SafeLabel`），因为它们都使用了 `Send()` 方法。

## 回调定制同步上下文

与服务定制同步上下文相似，我们可以为回调的使用安装一个定制同步上下文。打开代理的线程（或第一次调用代理的线程）需要将定制同步上下文添加到线程中。例8-23演示了如何在使用代理之前将 `AffinitySynchronizer` 类添加到回调对象中。

例8-23：为回调设置定制同步上下文

```

interface IMyContractCallback
{
    [OperationContract]
    void OnCallback();
}
class MyClient : IMyContractCallback

```

译注4：参见341页的描述。

```

{
    // 该方法总是被相同的线程调用
    public void OnCallback()
    {....}
}

MyClient client = new MyClient();
InstanceContext callbackContext = new InstanceContext(client);
MyContractClient proxy = new MyContractClient(callbackContext);

SynchronizationContext synchronizationContext = new AffinitySynchronizer();
SynchronizationContext.SetSynchronizationContext(synchronizationContext);

using(synchronizationContext as IDisposable)
{
    proxy.DoSomething();
    /* 在回调之后执行某些阻塞操作 */
    proxy.Close();
}

```

### CallbackThreadAffinityBehaviorAttribute 特性

与服务相似，我们同样可以使用特性为回调终结点分配一个定制同步上下文。例如我定义的 `CallbackThreadAffinityBehaviorAttribute` 特性：

```

[AttributeUsage(AttributeTargets.Class)]
public class CallbackThreadAffinityBehaviorAttribute : Attribute, IEndpointBehavior
{
    public CallbackThreadAffinityBehaviorAttribute(Type callbackType);
    public CallbackThreadAffinityBehaviorAttribute(Type callbackType,
                                                    string threadName);

    public string ThreadName
    {get;set;}
}

```

`CallbackThreadAffinityBehavior` 特性可以在相同线程上通过客户端支持的回调契约，执行所有的回调。既然特性需要影响回调终结点，因此它可以像第6章介绍的那样实现 `IEndpointBehavior` 接口。我们可以直接将特性应用到回调类型上，如例8-24所示。

#### 例8-24：应用 `CallbackThreadAffinityBehavior` 特性

```

[CallbackThreadAffinityBehavior(typeof(MyClient))]
class MyClient : IMyContractCallback, IDisposable
{
    MyContractClient m_Proxy;

    public void CallService()
    {
        InstanceContext callbackContext = new InstanceContext(this);
        m_Proxy = new MyContractClient(callbackContext);
    }
}

```



```

        m_Proxy.DoSomething();
    }
    // 该方法总是被相同的回调线程以及客户端线程调用
    public void OnCallback()
    {
        // 以手动的同步方式访问状态与资源
    }
    public void Dispose()
    {
        m_Proxy.Close();
    }
}

```

该特性需要一个构造参数，类型为它所标记的回调类型。注意，虽然回调总是在相同的线程中被 WCF 调用，但如果其他客户端线程也要访问该方法，我们仍然需要对它执行同步访问。

如果使用例 8-24 所示的 `CallbackThreadAffinityBehavior` 特性，则例 8-23 可以简化为：

```

MyClient client = new MyClient();
InstanceContext callbackContext = new InstanceContext(client);
MyContractClient proxy = new MyContractClient(callbackContext);

proxy.DoSomething();
/* 在回调之后执行某些阻塞操作 */
proxy.Close();

```

例 8-25 给出了 `CallbackThreadAffinityBehavior` 特性的实现。

#### 例 8-25：实现 `CallbackThreadAffinityBehaviorAttribute` 特性

```

[AttributeUsage(AttributeTargets.Class)]
public class CallbackThreadAffinityBehaviorAttribute :
    Attribute, IEndpointBehavior
{
    string m_ThreadName;
    Type m_CallbackType;

    public string ThreadName // 访问 m_ThreadName
    {get;set;}

    public CallbackThreadAffinityBehaviorAttribute(Type callbackType)
        : this(callbackType,null)
    {}
    public CallbackThreadAffinityBehaviorAttribute(Type callbackType,
        string threadName)
    {
        m_ThreadName = threadName;
        m_CallbackType = callbackType;
        AppDomain.CurrentDomain.ProcessExit += delegate
        {

```

```

        ThreadAffinityHelper.CloseThread(m_CallbackType);
    };
}
void IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint serviceEndpoint,
                                           ClientRuntime clientRuntime)
{
    m_ThreadName = m_ThreadName ?? "Executing callbacks of " + m_CallbackType;
    ThreadAffinityHelper.ApplyDispatchBehavior(m_CallbackType, m_ThreadName,
                                              clientRuntime.CallbackDispatchRuntime);
}
void IEndpointBehavior.AddBindingParameters(...)
{}
void IEndpointBehavior.ApplyDispatchBehavior(...)
{}
void IEndpointBehavior.Validate(...)
{}
}

```

CallbackThreadAffinityBehavior特性的构造函数通过成员变量保存了它所支持的回调类型，如果有线程名，则还要保存线程名。特性使用了之前介绍的ThreadAffinityHelper类，通过它将AffinitySynchronizer添加到分发器中。与服务不同，由于没有引入宿主，因而没有Closing事件可供订阅，缺少该事件，也就无法将它作为关闭AffinitySynchronizer的工作线程的信号。事实上，构造函数通过一个匿名方法订阅了当前应用程序域的ProcessExit事件，当客户端应用程序关闭时，匿名方法会使用ThreadAffinityHelper来关闭工作线程。

CallbackThreadAffinity特性实现了IEndpointBehavior接口，目前，我们只需要关注一个ApplyClientBehavior()方法，它能够对回调终节点的分发器施加影响：

```

public interface IEndpointBehavior
{
    void ApplyClientBehavior(ServiceEndpoint serviceEndpoint,
                           ClientRuntime clientRuntime);
    // 更多成员
}

```

在ApplyClientBehavior()方法中，特性通过ClientRuntime类型参数的CallbackDispatchRuntime属性获取它的分发器：

```

public sealed class ClientRuntime
{
    public DispatchRuntime CallbackDispatchRuntime
    {get;}
    // 更多成员
}

```

然后将它传递给ThreadAffinityHelper，用来添加AffinitySynchronizer。

一旦关闭了客户端应用程序, AffinitySynchronizer使用的客户端工作线程也会随之自动关闭。客户端可能需要加快处理线程, 然后尽快关闭它。为了实现这一目的, 客户端应该显式地调用 ThreadAffinityHelper 类的 CloseThread() 方法:

```
MyClient client = new MyClient();
InstanceContext callbackContext = new InstanceContext(client);
MyContractClient proxy = new MyContractClient(callbackContext);
proxy.DoSomething();
/* 在回调之后执行某些阻塞操作
proxy.Close();
ThreadAffinityHelper.CloseThread(typeof(MyClient));
```

## 异步调用

当一个客户端调用服务时, 在服务执行调用并将控制权返回给客户端期间, 客户端通常会被阻塞, 直到操作完成执行并返回。但是, 在某些情况下, 我们也可以采用异步方式调用操作, 即服务在后台执行操作, 然后以某种方式让客户端知道方法已经执行完毕, 并将调用的结果传递给客户端, 这样, 就能够将控制权立即返回给客户端。这种执行模式被称为异步操作调用, 也就是所谓的异步调用活动。异步调用能够改善客户端的响应速度与可用性 (Availability)。

**注意:** 单向操作不适用于异步调用。首先, 单向调用无法保证异步的执行方式。如果服务的传入调用队列的容量能够满足调用数, WCF 就会阻塞单向调用的调用者, 直到它能够将调用放入到队列中。此外, 当单向操作完成时, 没有一种便捷的方法通知客户端调用的结果与错误。虽然以手工方式可以定制一种机制, 为每个单向调用传递方法 ID, 然后使用回调将方法执行完毕的消息、方法的结果以及错误报告给客户端, 然而, 这种解决方案太过于复杂, 难以普及。当通信错误无法到达客户端时, 单向操作仍然要求服务捕获所有的异常。它同时还强制使用双向绑定, 这就决定了我们无法同时以同步方式与异步方式调用服务的操作。

## 异步机制的必备条件

要在各种不同的情况下使用 WCF 异步调用, 最好是首先列出面向服务的异步调用所支持的通用的必备条件, 它们包括:

- 同步调用与异步调用应该使用相同的服务代码。这样就使得服务开发者可以只关注业务逻辑, 又能同时满足同步与异步客户端的要求。
- 第一个必备条件的一个推论是, 应该由客户端决定调用服务的方式是同步还是异步。反之而言, 就意味着在不同的情况下 (或者是同步调用, 或者是异步调用), 客户端具有不同的实现代码。

- 客户端能够发出多个异步调用，在这个过程中，它将拥有多个异步调用。客户端能够分辨多个方法执行完成的情况。
- 如果服务操作定义了输出参数或返回值，那么当控制权返回给客户端时，这些参数是不可用的。在操作执行完成后，客户端应该想办法获取这些结果值。
- 类似的，通信错误或服务错误应该返回给客户端。在操作执行期间抛出的异常应该在随后返回给客户端。
- 异步机制的实现应该与使用的绑定和传输技术无关。任何绑定都应该支持异步调用。
- 不应使用特定的技术创建异步机制，例如使用 .NET 异常或委托。
- 最后一点与其说是异步机制的必备条件，还不如说是一条设计指南：异步调用机制应该直接、简单，易于使用。例如，异步调用机制应该尽可能隐藏它的实现细节，例如用于分发调用的工作线程。

客户端可以有多种方式处理操作完成。客户端发出一个异步调用，然后选择：

- 在调用过程中执行某些工作，阻塞它直到操作完成。
- 在调用过程中执行某些工作，然后轮询访问操作完成的情况。
- 方法完成后接收到一个通知。通知由客户端提供的方法的回调组成。回调包含的信息应该能够识别是哪一个操作执行完成，同时还要包含它的返回值。
- 在调用过程中执行某些工作，然后在预先设定的时间值内等待。即使操作没有完成，一旦超过该时间值，仍然会终止等待。
- 同时等待多个操作完成。客户端可以选择等待所有或者任意的正在处理的调用执行完成。

WCF 为客户端提供了以上所有的选项。WCF 完全支持客户端的易用性 (Facility)，事实上，服务并不关心客户端是否被异步调用。这意味着任何服务在本质上都是支持异步调用的，我们可以同时采用同步方式和异步方式调用相同的服务。此外，由于客户端支持所有的异步调用方式，而不用考虑服务，因此异步调用能够使用任何一种绑定方式。

---

**注意：**本节介绍的 WCF 对异步调用的支持，与 .NET 的常规 CLR 类型支持的基于委托的异步调用相似，但两者并不完全一致。

---

## 基于代理的异步调用

由于客户端决定了调用的方式是同步还是异步，因此我们需要为异步方式创建不同的代

理。使用 SvcUtil 的 /async 开关, 可以生成一个代理, 它除了包含同步方法外, 还包含了异步方法。对于原来的契约中的每个操作, 异步代理与契约还定义了两个额外的方法, 形式如下:

```
[OperationContract(AsyncPattern = true,
    Action = "<original action name>",
    ReplyAction = "<original response name>")]
IAsyncResult Begin<Operation>(<in arguments>,
    AsyncCallback callback, object asyncState);
<returned type> End<Operation>(<out arguments>, IAsyncResult result);
```

OperationContract 特性定义了 Boolean 类型的属性 AsyncPattern:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public bool AsyncPattern
    {get;set;}
    // 更多成员
}
```

AsyncPattern 属性的默认值为 false。它只对于契约的客户端副本有意义。可以只设置 Begin<Operation>() 方法的 AsyncPattern 的值为 true, 同时, 定义的契约必须包含一个与 End<Operation>() 相匹配的方法。装载代理时会验证这些约束条件。AsyncPattern 负责绑定同步方法与 Begin/End 方法对, 关联同步执行与异步执行。简要地讲, 当客户端调用 AsyncPattern 值为 true 的 Begin<Operation>() 方法时, 它会告知 WCF 不要通过服务定义中的方法名直接调用异步方法, 而是使用线程池的线程同步调用原来的方法 (通过 Action 名称识别)。同步调用会阻塞线程池的线程, 但不会阻塞正在调用的客户端。只有在将调用请求分发到线程池的一瞬间, 客户端才会被阻塞。同步调用的应答方法对应于 End<Operation>() 方法。

例 8-26 定义了一个 Calculator 契约, 以及实现它的服务, 和通过 /async 开关生成的代理类。

例 8-26: 异步契约与代理

```
//////////////////// 服务端 //////////////////////////////////////
[ServiceContract]
interface ICalculator
{
    [OperationContract]
    int Add(int number1, int number2);
    // 更多操作
}
class Calculator : ICalculator
{
    public int Add(int number1, int number2)
    {
```

```

        return number1 + number2;
    }
    // 其余实现
}
////////// 客户端 //////////
[ServiceContract]
public interface ICalculator
{
    [OperationContract]
    int Add(int number1,int number2);

    [OperationContract(AsyncPattern = true,
        Action = ".../ICalculator/Add",
        ReplyAction = ".../ICalculator/AddResponse")]
    IAsyncResult BeginAdd(int number1,int number2,AsyncCallback callback,
        object asyncState);
    int EndAdd(IAsyncResult result);

    // 其余方法
}
public partial class CalculatorClient : ClientBase<ICalculator>,ICalculator
{
    public int Add(int number1,int number2)
    {
        return Channel.Add(number1,number2);
    }
    public IAsyncResult BeginAdd(int number1,int number2,
        AsyncCallback callback,object asyncState)
    {
        return Channel.BeginAdd(number1,number2,callback,asyncState);
    }
    public int EndAdd(IAsyncResult result)
    {
        return Channel.EndAdd(result);
    }
    // 其余方法与构造函数
}

```

注意，契约中的 BeginAdd() 操作仍然保留了原来定义的动作与 ReplyAction，但我们完全可以删去它们：

```

[OperationContract(AsyncPattern = true)]
IAsyncResult BeginAdd(int number1,int number2,AsyncCallback callback,
    object asyncState);

```

## 异步调用

Begin<Operation>() 方法接收对应的同步操作的人参。人参可以是按值传递的数据契约，也可以是按引用传递（使用 ref 修饰符）的数据契约。它对应的原来方法的返回值以及显式的输出参数（使用 out 和 ref 修饰符）则成为 End<Operation>() 方法的一部分。例如，如下的操作定义：



```
[ServiceOperation]
string MyMethod(int number1,out int number2,ref int number3);
```

对应的 `Begin<Operation>()` 和 `End<Operation>()` 方法如下所示:

```
[ServiceOperation(...)]
IAsyncResult BeginMyMethod(int number1,ref int number3,
                           AsyncCallback callback,object asyncState);
string EndMyMethod(out int number2,ref int number3,IAsyncResult asyncResult);
```

`Begin<Operation>()` 接收了两个额外的输入参数 `callback` 和 `asyncState`, 在对应的同步操作的方法签名中并没有定义它们。`Callback` 参数是一个委托类型, 指向的事件能够在客户端方法完成后发出通知。`asyncState` 是 `object` 对象, 负责传递处理方法完成一方所需的状态信息。两个参数都是可选的: 调用者可以将它们两个的值同时设置为 `null`。例如, 使用异步代理以异步方式调用例 8-26 中 `Calculator` 服务的 `Add()` 方法, 如果开发者不需要调用的结果或产生的错误, 就可以传递 `null` 值:

```
CalculatorClient proxy = new CalculatorClient();
proxy.BeginAdd(2,3,null,null);//Dispatched asynchronously
proxy.Close();
```

只要客户端具有异步契约的定义, 我们还可以使用通道工厂异步调用操作:

```
ChannelFactory<ICalculator> factory = new ChannelFactory<ICalculator>();
ICalculator proxy = factory.CreateChannel();
proxy.BeginAdd(2,3,null,null);
ICommunicationObject channel = proxy as ICommunicationObject;
channel.Close();
```

然而, 这样的调用会导致客户端无法获取执行的结果。

## IAsyncResult 接口

每个 `Begin<Operation>()` 方法都会返回一个实现了 `IAsyncResult` 接口的对象, 该接口被定义在 `System.Runtime.Remoting.Messaging` 命名空间中:

```
public interface IAsyncResult
{
    object AsyncState
    {get;}
    WaitHandle AsyncWaitHandle
    {get;}
    bool CompletedSynchronously
    {get;}
    bool IsCompleted
    {get;}
}
```



返回的 `IAsyncResult` 对象使用 `Begin<Operation>()` 对被调用的方法进行唯一性验证。开发者可以将 `IAsyncResult` 对象传递给 `End<Operation>()` 方法，然后从期望获取的结果中识别指定的异步方法执行。`End<Operation>()` 方法会阻塞它的调用者，直到它等待的操作完成（通过传递的 `IAsyncResult` 对象进行识别），它能够返回执行的结果或产生的错误。调用 `End<Operation>()` 时，如果方法已经完成，则 `End<Operation>()` 方法不会阻塞调用者，同时返回执行结果。例 8-27 演示了整个执行步骤。

例 8-27：简单的异步执行步骤

```
CalculatorClient proxy = new CalculatorClient();
IAsyncResult asyncResult1 = proxy.BeginAdd(2,3,null,null);
IAsyncResult asyncResult2 = proxy.BeginAdd(4,5,null,null);
proxy.Close();

/* 执行某些工作 */

int sum;

sum = proxy.EndAdd(asyncResult1); // This may block
Debug.Assert(sum == 5);

sum = proxy.EndAdd(asyncResult2); // This may block
Debug.Assert(sum == 9);
```

虽然例 8-27 非常简单，但它却蕴含了几个关键点。第一个关键点是相同的代理实例能够执行多个异步调用。调用者使用 `Begin<Operation>()` 方法返回的唯一的 `IAsyncResult` 对象辨别不同的正在处理的调用。事实上，当调用者发出异步调用时，如例 8-27 所示，调用者必须保存 `IAsyncResult` 对象。此外，调用者不应该假定调用完成的顺序。第二个调用完全有可能在第一个调用之前完成。最后，如果不再使用代理，我们可以在分发异步调用之后立即关闭代理，此时，它仍然能够调用 `End<Operation>()` 方法。

虽然在例 8-27 中没有体现出来，但对于异步调用，还有两个重要的编程要点：

- 每个异步操作只能调用一次 `End<Operation>()` 方法。如果多次调用，会导致 `InvalidOperationException` 异常。
- 只能在用于分发调用的相同代理对象上，将 `IAsyncResult` 对象传递给 `End<Operation>()` 方法。传递 `IAsyncResult` 对象给不同的代理实例，会导致 `AsyncCallbackException` 异常。

## 轮询或等待完成

客户端在调用 `End<Operation>()` 方法时会被阻塞，直到异步方法返回。如果客户端在调用过程中只有一定数量的工作要做，而且一旦工作执行完毕，客户端就不能继续执行

没有返回值与输出参数的操作，或者客户端正好获知操作已经完成，那么采用这种方式还算不错。但是，如果客户端只是希望获知操作是否已经完成，则应该采用哪种方式呢？如果客户端在固定时间内等待操作完成后，执行了一些额外的限定过程，然后继续等待，又应该采用哪种方式呢？WCF提供了几种调用 `End<Operation>()` 方法的替代编程模型。

`Begin<Operation>()` 方法返回的 `IAsyncResult` 接口对象定义了 `AsyncWaitHandle` 属性，类型为 `WaitHandler`：

```
public abstract class WaitHandle : ...
{
    public static bool WaitAll(WaitHandle[] waitHandles);
    public static int WaitAny(WaitHandle[] waitHandles);
    public virtual void Close();
    public virtual bool WaitOne();
    // 更多成员
}
```

`WaitHandle` 的 `WaitOne()` 方法只有在处理对象发出信号后才会返回。例 8-28 演示了 `WaitOne()` 方法的使用方式。

#### 例 8-28：使用 `IAsyncResult.AsyncWaitHandle` 阻塞直到操作完成

```
CalculatorClient proxy = new CalculatorClient();
IAsyncResult asyncResult = proxy.BeginAdd(2,3,null,null);
proxy.Close();

/* 执行某些工作 */

asyncResult.AsyncWaitHandle.WaitOne(); //This may block
int sum = proxy.EndAdd(asyncResult); //This will not block
Debug.Assert(sum == 5);
```

从逻辑上讲，例 8-28 与只调用了 `End<Operation>()` 方法的例 8-27 是等同的。如果操作仍然在执行过程中，`WaitOne()` 就会阻塞。在调用 `WaitOne()` 方法的同时，如果方法执行完成，就会解除 `WaitOne()` 方法的阻塞，客户端继续调用 `End<Operation>()` 方法以获取返回值。例 8-28 与例 8-27 之间的最大区别在于，例 8-28 在调用 `End<Operation>()` 方法时，可以保证调用者不会被阻塞。

例 8-29 演示的示例更加贴近实际的应用，它通过指定超时值（在本例中被设置为 10 毫秒）调用 `WaitOne()` 方法。如果指定了超时值，不管方法是执行完成，还是达到了超时值，只要满足其中一个条件，`WaitOne()` 方法就会返回。

#### 例 8-29：指定等待的超时值使用 `WaitOne()` 方法

```
CalculatorClient proxy = new CalculatorClient();
IAsyncResult asyncResult = proxy.BeginAdd(2,3,null,null);
```

```
while(asyncResult.IsCompleted == false)
{
    asyncResult.AsyncWaitHandle.WaitOne(10, false); // 这样可能会阻塞
    /* 执行某些工作 */
}
int sum = proxy.EndAdd(asyncResult); // 这样不会阻塞
```

例8-29使用了 `IAsyncResult` 的另一个现有的属性 `IsCompleted`。`IsCompleted` 属性允许我们无需等待或通过阻塞就能获得调用的状态。我们甚至可以在严格的轮询模式下使用 `IsCompleted` 属性：

```
CalculatorClient proxy = new CalculatorClient();
IAsyncResult asyncResult = proxy.BeginAdd(2, 3, null, null);
proxy.Close();

// 在后面的某个时候：
if(asyncResult.IsCompleted)
{
    int sum = proxy.EndAdd(asyncResult); // 这样不会阻塞
    Debug.Assert(sum == 5);
}
else
{
    // 同时执行某些工作
}
```

使用 `AsyncWaitHandle` 属性管理执行过程中的多个并发的异步方法，足以体现这一属性眩目的价值。我们可以使用 `WaitHandle` 的静态方法 `WaitAll()` 等待多个异步方法执行完成，如例8-30所示。

#### 例8-30：等待多个方法执行完成

```
CalculatorClient proxy = new CalculatorClient();
IAsyncResult asyncResult1 = proxy.BeginAdd(2, 3, null, null);
IAsyncResult asyncResult2 = proxy.BeginAdd(4, 5, null, null);
proxy.Close();

WaitHandle[] handleArray = {asyncResult1.AsyncWaitHandle,
                             asyncResult2.AsyncWaitHandle};

WaitHandle.WaitAll(handleArray);

int sum;
// 下列对 EndAdd() 方法的调用不会导致阻塞

sum = proxy.EndAdd(asyncResult1);
Debug.Assert(sum == 5);

sum = proxy.EndAdd(asyncResult2);
Debug.Assert(sum == 9);
```

若要使用 `WaitAll()` 方法，需要构建一个 `WaitHandle` 类型元素的数组。注意，我们

仍然需要调用 `End<Operation>()` 方法去访问返回的值。相反, 如果使用 `WaitHandle` 类的 `WaitAny()` 静态方法, 则不必等待所有的方法返回, 而只要其中任意一个方法返回即可。与 `WaitOne()` 方法非常相似, `WaitAll()` 与 `WaitAny()` 方法都具有多个重载版本, 允许我们指定等待的超时值, 而不是无限期地等待。

## 完成回调

异步调用的完成并不需要使用阻塞、等待或轮询机制, WCF 提供了另一种编程模型, 称为完成回调 (Completion Callback), 它集成了上述机制的功能。客户端为 WCF 提供了一种方法, 要求 WCF 在异步方法完成时调用返回的方法。客户端能够提供一个回调的实例方法或静态方法, 让相同的回调方法处理多个异步调用的执行完成。当异步方法执行完成时, 不需要返回到线程池, 工作线程会调用完成回调。若要指明完成回调方法, 客户端需要为 `Begin<Operation>()` 提供一个 `AsyncCallback` 类型的委托, 定义为:

```
public delegate void AsyncCallback(IAsyncResult asyncResult);
```

该委托作为 `Begin<Operation>()` 方法的倒数第二个参数。

例 8-31 演示了使用完成回调管理异步调用的方法。

例 8-31: 通过完成回调管理异步调用

```
class MyClient : IDisposable
{
    CalculatorClient m_Proxy = new CalculatorClient();

    public void CallAsync()
    {
        m_Proxy.BeginAdd(2, 3, OnCompletion, null);
    }
    void OnCompletion(IAsyncResult result)
    {
        int sum = m_Proxy.EndAdd(result);
        Debug.Assert(sum == 5);
    }
    void Dispose()
    {
        m_Proxy.Close();
    }
}
```

与之前描述的编程模型不同, 在使用完成回调方法时, 不需要保存 `Begin<Operation>()` 方法返回的 `IAsyncResult` 对象, 因为 WCF 在调用完成回调时, 会把 `IAsyncResult` 对象作为参数。由于 WCF 为每个异步方法提供了唯一的 `IAsyncResult` 对象, 因而我们可以在相同的回调方法中使用多个异步方法完成:

```
m_Proxy.BeginAdd(2,3,OnCompletion,null);  
m_Proxy.BeginAdd(4,5,OnCompletion,null);
```

无需将类的方法作为完成回调，我们可以便捷地使用本地匿名方法：

```
CalculatorClient proxy = new CalculatorClient();  
int sum;  
AsyncCallback completion = delegate(IAsyncResult result)  
{  
    sum = proxy.EndAdd(result);  
    Debug.Assert(sum == 5);  
};  
proxy.BeginAdd(2,3,completion,null);  
proxy.Close();
```

注意，匿名方法设置了一个外部变量（sum），用来保存 Add() 操作的结果。

回调完成方法是事件驱动的应用程序首选的模式。一个事件驱动的应用程序包含了触发事件（或请求）的方法，这些方法负责处理事件，最后触发这些事件。如果将应用程序设计为事件驱动模式，可以使应用程序更易于管理多线程、事件与回调，从而提高系统的可扩展性、响应速度与性能。通过回调完成方法管理异步调用，完全符合事件驱动的架构，正如手套之于手一般完全吻合。其他实现方式（等待、阻塞和轮询）则主要用于那些要求严格的、可预见的以及确定了执行流程的应用程序。建议只要有可能，最好都使用完成回调方法。

## 完成回调与线程安全

因为回调方法在线程池的线程上执行，我们必须在回调方法以及提供该方法的对象中提供线程安全。这意味着我们必须使用同步对象，锁定对客户端成员变量的访问。我们需要考虑客户端线程与线程池中工作线程之间的同步，潜在含义就是要考虑多个工作线程之间的同步，这些工作线程会将所有的调用并发地放入完成回调方法中，以处理各自的异步调用完成。我们需要确保完成回调方法是重入的，线程安全的。

## 传递状态信息

Begin<Operation>()方法的最后一个参数是 asyncState。asyncState 对象为状态对象，会在适合的时候提供给一个可选的容器。处理方法完成的一方可以通过 IAsyncResult 的 AsyncState 属性访问这样的容器对象。虽然我们完全可以使用其他任何一个异步调用编程模型（阻塞、等待或者轮询）的状态对象，但最有力的方式还是与完成回调进行联合。原因很简单：在其他的编程模型中，都需要开发者自己管理 IAsyncResult 对象，管理一个额外的容器并非它需要增加的职责。当我们使用完成回调时，容器对象提供了将额外参数传递给回调方法的唯一途径，只要这些方法的签名是预先确定的。

例8-32演示了如何使用状态对象将整数值当作一个额外的参数传递给完成回调。注意，回调必须将 `AsyncState` 属性的类型向下转换为实际的类型。

例8-32：使用状态对象传递一个额外参数

```
class MyClient : IDisposable
{
    CalculatorClient m_Proxy = new CalculatorClient();

    public void CallAsync()
    {
        int asyncState = 4; // 整型，仅为举例
        m_Proxy.BeginAdd(2, 3, OnCompletion, asyncState);
    }
    void OnCompletion(IAsyncResult result)
    {
        int asyncState = (int)result.AsyncState;
        Debug.Assert(asyncState == 4);

        int sum = m_Proxy.EndAdd(result);
    }
    void Dispose()
    {
        m_Proxy.Close();
    }
}
```

状态对象的常见用法是向 `Begin<Operation>()` 方法传递它要使用的代理，而不是以成员变量的方式保存代理。

```
class MyClient
{
    public void CallAsync()
    {
        CalculatorClient proxy = new CalculatorClient();
        proxy.BeginAdd(2, 3, OnCompletion, proxy);
        proxy.Close();
    }
    void OnCompletion(IAsyncResult result)
    {
        CalculatorClient proxy = result.AsyncState as CalculatorClient;
        Debug.Assert(proxy != null);

        int sum = proxy.EndAdd(result);
        Debug.Assert(sum == 5);
    }
}
```

## 完成回调的同步上下文

完成回调可能需要线程关联度，运行在特定的同步上下文中。一种特殊的情形是完成回调需要更新与异步调用结果相关的用户界面。遗憾的是，我们必须使用之前介绍的技术，



手动地将完成回调的调用封装送给正确的同步上下文。例8-33演示了与容器窗体直接交互的完成回调，它能够确保 UI 更新是在 UI 同步上下文中。

例 8-33：依赖于完成回调的同步上下文

```
partial class CalculatorForm : Form
{
    CalculatorClient m_Proxy;
    SynchronizationContext m_SynchronizationContext;

    public MyClient()
    {
        InitializeComponent();
        m_Proxy = new CalculatorClient();
        m_SynchronizationContext = SynchronizationContext.Current;
    }

    public void CallAsync(object sender, EventArgs args)
    {
        m_Proxy.BeginAdd(2, 3, OnCompletion, null);
    }

    void OnCompletion(IAsyncResult result)
    {
        SendOrPostCallback callback = delegate
        {
            Text = "Sum = " + m_Proxy.EndAdd(result);
        };
        m_SynchronizationContext.Send(callback, null);
    }

    public void OnClose(object sender, EventArgs args)
    {
        m_Proxy.Close();
    }
}
```

## 单向异步操作

异步调用单向操作并无多大意义，因为异步调用的一个主要特征就是获取以及关联响应消息，而单向调用却没有这样可用的消息。如果执意要以异步方式调用单向操作，则 `End<Operation>()` 方法永远都不会阻塞，也不会抛出异常。如果为单向操作的异步调用提供一个完成回调，那么在 `Begin<Operation>()` 方法返回值后，会立即调用回调。只有一种情形需要异步调用单向操作，就是为了避免单向调用可能造成的阻塞。此时，我们应该为状态对象以及完成回调传递一个 `null` 值。

## 异步错误处理

在分发异步调用时，输出参数与返回值并非唯一不能使用的元素。实际上在这种情况下，异常同样会丢失。在调用 `Begin<Operation>()` 方法之后，控制权返回给客户端，但



是在异步方法出现错误以及抛出异常之前,可能还需要一段时间;而在客户端实际调用 `End<Operation>()` 方法之前,同样需要一段时间。因此,WCF必须为客户端提供某种方法,以获知抛出的异常,并允许客户端处理它。当异步方法抛出一个异常时,代理会捕获异常,然后在客户端调用 `End<Operation>()` 方法时,代理重新抛出该异常对象,并让客户端处理该异常。如果提供了完成回调,WCF会在接收到异常之后立即调用回调方法。抛出的异常应遵循错误契约与异常类型的规定,正如第6章所述。

---

**注意:** 如果服务操作契约定义了错误契约,则 `FaultContract` 特性只能应用到同步操作上。

---

## End<Operation> 之后资源的清除

无论何时调用 `Begin<Operation>()` 方法,返回的 `IAsyncResult` 对象都持有一个单独的 `WaitHandle` 对象的引用,通过 `AsyncWaitHandle` 属性可以访问。调用对象的 `End<Operation>()` 方法不会关闭处理对象。相反,只有实现的对象支持垃圾回收时,处理对象才会被关闭。使用非托管资源时,我们必须考虑应用程序关于析构的要求。应用程序分发异步调用的速度可能会比 .NET 回收处理对象的速度要快(至少在理论上存在),这就可能导致资源泄露。为了弥补这一缺陷,可以在调用 `End<Operation>()` 方法之后显式地关闭处理对象。例如,使用与例 8-31 相同的定义:

```
void OnCompletion(IAsyncResult result)
{
    int sum = m_Proxy.EndAdd(result);
    Debug.Assert(sum == 5);
    result.AsyncWaitHandle.Close();
}
```

## 异步调用与事务

事务与异步调用是相互冲突的。首先,设计良好的事务一般周期较短,而使用异步调用的主要目的则是为操作提供一定的延迟。其次,客户端的环境事务(Ambient Transaction)在默认情况下不会流动到服务中,因为异步操作是在工作线程上调用的,而不是在客户端线程。开发一个使用克隆事务(Cloned Transaction)的专有机制是可能的,但这样的实现过于复杂,我们应尽量避免。最后,当事务完成时,在提交或取消与事务无关的内容的后台中,不应该有剩余的活动还在执行,而这些却是在事务范围内异步操作调用产生的。因而,绝对不要混合使用事务与异步调用。

## 同步调用与异步调用

虽然从技术上讲，可以同时以同步方式和异步方式调用相同的服务，但这种可能性却非常低。

原因在于异步调用服务必然会改变客户端的工作流程，从而导致客户端无法使用相同的逻辑执行顺序来实现同步访问。例如，考虑一个在线商店的应用程序。假定客户端（执行客户请求的服务端对象）访问了一个Store（服务），该服务包含了顾客的订单细节。Store服务使用了三个相对独立的服务帮助处理订单：Order、Shipment和Billing。在同步场景下，Store服务调用了Order服务完成订单的提交。如果Order服务成功地处理了订单（例如库存包含了订单中的商品），则Store服务就会调用Shipment服务。只有Shipment服务执行成功，Store服务才会访问Billing服务，执行顾客的出账操作。执行步骤如图8-4所示。

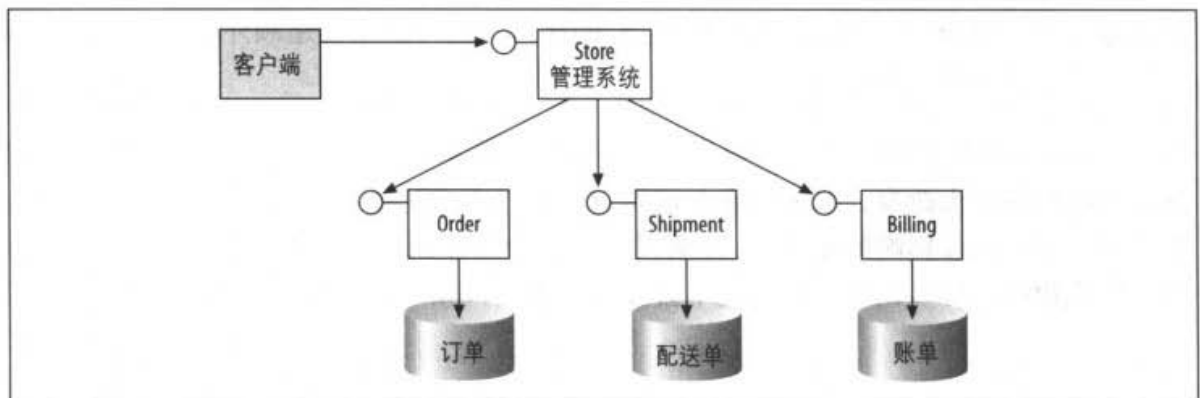


图 8-4：同步处理订单

图8-4中的工作流程存在一个致命的缺点，就是商店必须同步地、连续地处理订单。表面上看，似乎Store服务也可以异步调用它的辅助对象，因为在顾客提交订单后，如果能尽快地处理传入的订单，就可以提高系统的性能。然而，如果分开调用Order、Shipment和Billing服务，就有可能导致操作失败。这样的实现方式无疑为系统打开了通往地狱的大门。例如，Order服务可能会发现在库存中没有顾客需要的商品，而Shipment服务却试图运送一个并不存在的商品，Billing服务则已经对顾客完成了出账操作。

对一系列相互影响的服务使用异步调用，要求开发者修改代码与工作流程。要异步调用辅助服务，则Store服务只能调用Order服务。因为只有成功地处理了订单（参见图8-5），才会调用Shipment服务，从而避免刚才提及的潜在的 inconsistency。相似的，只有发货操作成功，Shipment服务才能异步调用Billing服务。

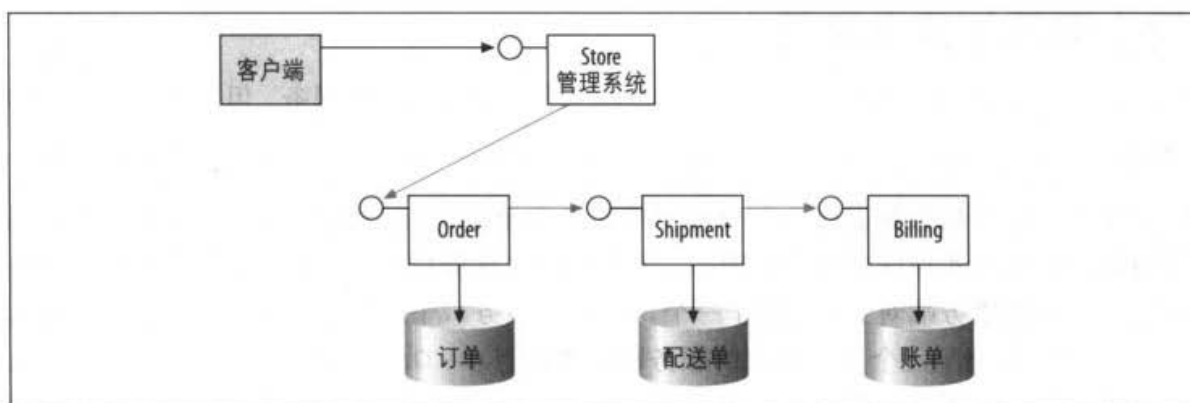


图 8-5: 针对异步处理订单修改工作流程

总体而言，如果异步工作流程包含了超过一个的服务，就应该让每个服务按照逻辑执行的顺序调用下一个服务。不言而喻，这样的编程模型会使得服务之间的耦合度增强（因为服务之间互相依赖），同时也会修改它们的接口（因为必须传递额外的参数，该参数是下一个服务调用所必需的）。

综上所述，如果调用服务时使用异步方式，而不是同步方式，就会导致服务接口与客户端工作流程的巨大改变。服务的异步调用应该建立在相互独立的同步执行的基础之上。在处理一系列相互影响的服务时，最好是分开一个工作线程去调用它们，同时使用另外的工作线程执行异步调用。这样就不会影响服务接口和原来的客户端执行顺序。



# 队列服务

WCF支持客户端和服务之间的离线工作。客户端将消息发送到一个队列中,再由服务对它们进行处理。这种交互方式增加了不同的可能性,从而形成了一种此前尚未出现过的编程模型。本章首先会向读者展示如何建立并配置简单的队列服务,然后重点关注诸如事务、实例管理、故障以及它们对服务的业务模型以及实现的影响。最后,本章提供了一个我们设计的框架,用于一个响应服务以及在 Internet 上实现队列调用的 HTTP 桥(HTTP Bridge)。

## 离线服务与客户端

前面章节的前提都是以客户与服务之间互联交互作为基础,它要求服务与客户端两端都必须启动并且运行,从而实现彼此间的交互。然而,还有相当多的情况(以及整体业务模型的调整)希望在一个面向服务的应用中拥有离线交互的能力。

### 可用性 (Availability)

即使客户端处于离线状态,它也可能需要用到服务,例如在使用一个移动设备的时候。解决办法是在一个本地队列中对请求排队,并在客户端联机之后,将这些请求发送给服务。同样的,服务也有可能离线,例如因为网络故障。即使在这种情况下,我们也希望客户端能够继续使用服务。当服务再次联机时,它可以从一个队列中取回那些等待处理的调用。即便客户端与服务两者都是有效的,并且正在运行,网络连接也可能发生中断,此时我们也希望客户端和服务两者能继续它们各自的工作。在两端同时使用队列将帮助我们实现这样的需求。

### 分解 (Disjoint) 工作

无论何时,只要可以将一个业务流分解为多个以时间分隔的操作(甚至可能是事务型的),而且这些操作必须执行,但却不一定立即执行,或者以某个特定的顺序执

行,此时使用队列无疑是最佳选择,因为它将改善系统的可用性与吞吐量。我们可以将操作放入队列中,并使它们彼此独立地执行。

#### 补偿 (Compensating) 工作

当我们的业务事务可能需要几个小时或几天来完成的时候,我们通常将它分为至少两个事务。第一个事务将需要立即完成的工作放入队列,而第二个事务用于验证第一个事务是否成功,并在必要的情况下对它的失败进行补偿。

#### 负载均衡 (Load Leveling)

负载通常是指独立的客户端的数目,而压力则是并发调用的数目。如果我们有一个不受约束的大量负载,它将转化成对我们系统的直接压力。大多数系统没有一个恒定的负载或压力,如图9-1所示。如果我们以最大负载来设计系统,我们将在大多数的负载周期中浪费系统资源;如果我们以处理平均服务来设计系统,我们将无法处理峰值时的情况。类似的,我们不仅需要检查绝对负载水平,还需要检查负载增长的速率(它的一阶导数)。一个负载中突然的峰值,即使它并没有超出我们系统在绝对概念中的处理能力,也可能仍然超过了我们系统可以及时处理的能力,因而系统的响应能力将显得不足。有了队列调用,服务就可以简单地把超出负载的部分加入队列,并在空闲的时候处理它。这使得我们能针对期望吞吐量的额定平均值来设计系统,而不是针对最大负载。

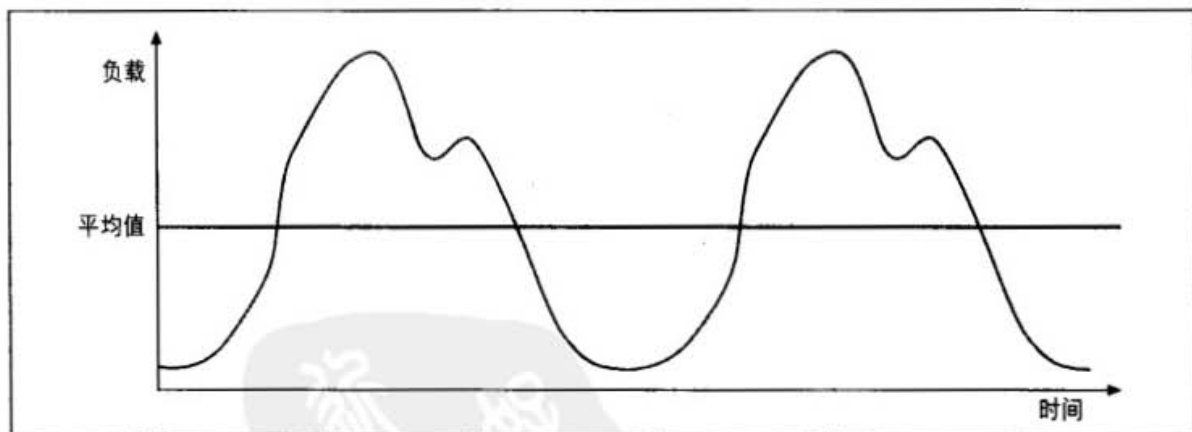


图 9-1: 波动的负载

## 队列调用

WCF 使用 `NetMsmqBinding` 以支持队列调用,并使用 MSMQ 代替 TCP、HTTP 或 IPC 传输消息。WCF 将 SOAP 消息打包到一个 MSMQ 消息中,并把它发送到一个指定的队列中。注意,就像不存在 WCF 消息到 TCP 包的直接映射一样, WCF 消息到 MSMQ 消息的直接映射也不存在。根据契约的会话模式,一个单独的 MSMQ 消息可以包含多个

WCF消息，也可以只是单独的一个，稍后我们将详细讨论。客户端通过将消息发送到一个MSMQ队列来代替向一个活动的服务发送WCF消息。客户端所能看到并与之交互的就是该队列，而不是一个服务的终结点。因此，调用实质上是异步、延迟的。当服务处理消息时，调用才会被执行（这使得调用是异步的），而且服务或客户端可能会与本地的队列进行交互（这使得调用是离线的）。

## 队列调用架构

和每个WCF服务一样，客户端都使用一个代理与服务进行交互，如图9-2中所示。

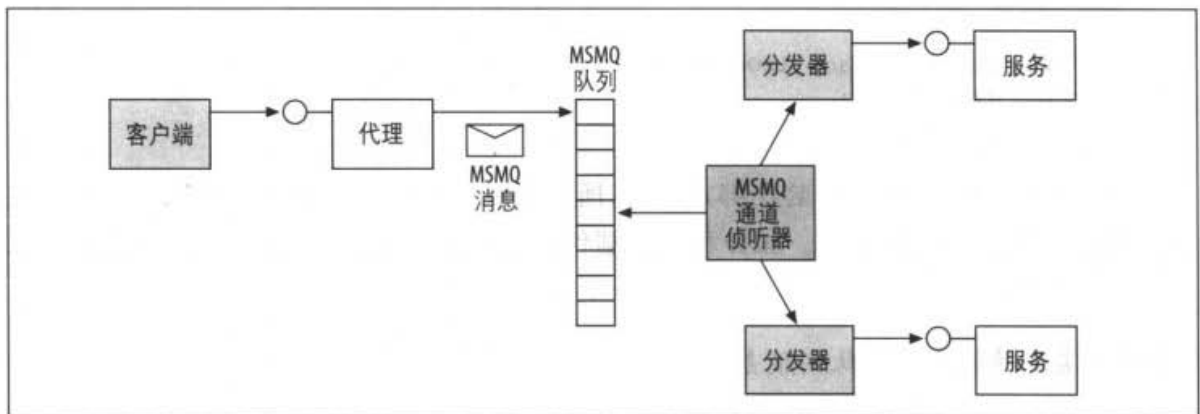


图 9-2：队列调用结构

但是，既然代理被配置为使用MSMQ绑定，它就不会向任何特定的服务发送WCF消息。相反，它会将对一个服务的调用（或多个调用）转化为一个MSMQ消息（或多个消息），并把它发送到在终结点地址中指定的队列。在服务端，当一个拥有队列终结点的服务宿主启动时，该宿主就建立了一个队列侦听器，它在概念上类似于使用TCP或HTTP时与端口关联的侦听器。队列侦听器在队列中检测到消息时，会从队列中取出该消息，然后创建宿主一端的拦截器链，并以一个分发器结束。分发器会像平常那样调用服务实例。假如有多个信息被发送到队列，侦听器会在取出消息后立即创建新的实例，从而实现异步的、离线的且并发的调用。

如果宿主是离线的，消息就会被放入到队列中。当宿主下一次联机时，消息将传递给服务。显而易见，假如客户端与服务两者都是活动的，而且正在运行并且是联机的，那么宿主将立即处理这些调用。

## 队列契约

一个针对某个队列所做的离线调用有可能无法返回任何结果，因为在消息被发送到队列的时候并没有调用任何服务逻辑。不仅如此，该调用还有可能在客户端应用程序已经关



闭后才被发送到服务并被处理，而此时已没有可用的客户端来处理返回值了。同理，调用也无法向客户端返回任何服务端异常，因为没有客户端可以捕获并处理该异常。事实上，WCF不允许在队列操作上使用错误契约。既然客户端不能被操作的调用所阻塞，更确切地说，客户端只会在消息被加入队列时发生短暂的阻塞，那么从客户端的角度来看队列调用本质上是异步的。而所有这些都是单向调用的典型特征。因此，任何一个契约，只要是使用了NetMsmqBinding绑定的终结点所公开的契约，就只能包含单向操作，并且WCF会在服务（及代理）加载的时候对此进行验证：

```
// 队列契约中只能是单向调用
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}
```

因为与MSMQ的交互是被封装在绑定中的，所以在服务或客户端调用代码中没有任何迹象表明该调用属于队列调用。服务和客户端代码看起来就像其他WCF客户端和服务代码一样，如例9-1所示。

例9-1：实现并使用一个队列服务

```
//////////////////// 服务端 //////////////////////////////////////
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}
class MyService : IMyContract
{
    public void MyMethod()
    { ... }
}
//////////////////// 客户端 //////////////////////////////////////
MyContractClient proxy = new MyContractClient();
proxy.MyMethod();
proxy.Close();
```

## 配置与安装

当我们为某个队列服务定义一个终结点时，该终结点地址必须包含队列的名称与标志，即队列的类型。MSMQ定义了两类类型的队列：公有队列和私有队列。公有（Public）队列需要安装一个MSMQ域控制器（MSMQ Domain Controller），并且可以被跨越机器边界访问。由于公有队列的安全性以及离线工作的特性，作为产品发布的应用程序往往需要使用公有队列。私有（Private）队列与其托管的机器同属本地，且不需要域管理



器。这样一种 MSMQ 的部署被称为工作组安装 (Workgroup Installation)。在开发过程中, 开发人员通常利用工作组安装为他们安装并管理私有队列。我们可以指定队列类型 (private 或 public), 作为队列终结点地址的一部分:

```
<endpoint
  address = "net.msmq://localhost/private/MyServiceQueue"
  binding = "netMsmqBinding"
  ...
/>
```

在公有队列情况下, 我们可以省略 public 标识符, 让 WCF 推断队列的类型。对于私有队列, 我们则必须包含标识符。另外还需要注意的是, 在队列的类型中没有 \$ 符号。

## 工作组安装及安全

当我们在一个工作组安装中使用私有队列时, 必须在客户端和服务两端禁用 MSMQ 安全。第 10 章讨论了如何保证包括队列调用在内的 WCF 调用的安全。简而言之, 默认的 MSMQ 安全配置希望用户能呈现验证的证书, 而且基于证书安全的 MSMQ 需要一个 MSMQ 域管理器。另一种方式, 则是为 MSMQ 的传输安全选择 Windows 安全, 需要活动目录集成, 这对于 MSMQ 工作组安装而言是不可能的。此处的例 9-2 只演示了禁用 MSMQ 安全的方法。

### 例 9-2: 禁用 MSMQ 安全

```
<system.serviceModel>
...
  <endpoint name = ...
    address = "net.msmq://localhost/private/MyServiceQueue"
    binding = "netMsmqBinding"
    bindingConfiguration = "NoMSMQSecurity"
    contract = "IMyContract"
  />
...
  <bindings>
    <netMsmqBinding>
      <binding name = "NoMSMQSecurity">
        <security mode = "None">
      </security>
    </binding>
  </netMsmqBinding>
</bindings>
</system.serviceModel>
```

## 创建队列

在服务和客户端两端, 队列应该在客户端调用被放入队列之前而存在。有几种创建队列的方案。管理者 (如果是开发过程中, 则是开发者) 可以使用 MSMQ 控制面板程序创建

队列,但这是一个应该实现自动化的手动步骤。宿主的进程可以使用System.Messaging的API验证在打开宿主之前队列是否存在。MessageQueue类提供了Exists()方法,用以验证一个队列是否被创建,同时还定义了创建一个队列的Create()方法:

```
public class MessageQueue : ...
{
    public static MessageQueue Create(string path); // 非事务型
    public static MessageQueue Create(string path, bool transactional);
    public static bool Exists(string path);
    public void Purge();
    // 更多成员
}
```

如果当前没有队列,宿主进程可以首先创建队列,然后再打开宿主。例9-3演示了这一执行步骤。

#### 例9-3: 在宿主端验证队列

```
ServiceHost host = new ServiceHost(typeof(MyService));

if(MessageQueue.Exists(@".\private$\MyServiceQueue") == false)
{
    MessageQueue.Create(@".\private$\MyServiceQueue", true);
}
host.Open();
```

例9-3中,宿主针对自身机器的MSMQ安装,验证了在打开宿主之前是否存在队列。如果需要,宿主代码会创建一个队列。这里没有使用设置值为true的事务队列,这一内容会在后面介绍。同样,我们需要注意在队列标识中关于\$符号的用法。显然,例9-3中存在的问题是它对队列名称进行了硬编码。更好的途径是通过将它保存在一个应用程序的设置中,然后从应用程序的配置文件中读出队列的名称。但是,即使使用这种方法还是有一些额外的问题。首先,我们必须不断地在应用程序的设置和终结点地址中保持队列名称的同步。其次,我们必须在每个队列服务的情况中重复这段代码。所幸,我们可以将例9-3的代码封装到我编写的ServiceHost<T>中,以自动化实现这一过程,如例9-4所示。

#### 例9-4: 在ServiceHost<T>中创建队列

```
public class ServiceHost<T> : ServiceHost
{
    protected override void OnOpening()
    {
        foreach(ServiceEndpoint endpoint in Description.Endpoints)
        {
            QueuedServiceHelper.VerifyQueue(endpoint);
        }
        base.OnOpening();
    }
}
```

```
// 更多成员
}
public static class QueuedServiceHelper
{
    public static void VerifyQueue(ServiceEndpoint endpoint)
    {
        if(endpoint.Binding is NetMsmqBinding)
        {
            string queue = GetQueueFromUri(endpoint.Address.Uri);
            if(MessageQueue.Exists(queue) == false)
            {
                MessageQueue.Create(queue,true);
            }
        }
    }
    // 通过地址解析队列名
    static string GetQueueFromUri(Uri uri)
    {...}
}
```

在例 9-4 中, `ServiceHost<T>` 重写了基类的 `OnOpening()` 方法。在打开宿主之前, 调用 `Open()` 方法之后, 会调用该方法。`ServiceHost<T>` 遍历了配置终结点的整个集合。对于每个终结点, 如果使用的是 `NetMsmqBinding` 绑定, 即队列调用所期望使用的绑定, 则 `ServiceHost<T>` 会调用静态辅助类 `QueuedServiceHelper`, 通过传入终结点对队列进行验证。`QueuedServiceHelper` 类的 `VerifyQueue()` 方法会从终结点的地址中解析出队列的名称, 如果需要, 则使用类似于例 9-3 的代码创建队列。

使用 `ServiceHost<T>`, 例 9-3 可以被简化为:

```
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.Open();
```

在客户端, 同样必须在向队列发出调用之前验证它是否已经存在。例 9-5 演示了在客户端所需的步骤。

#### 例 9-5: 由客户端验证队列

```
if(MessageQueue.Exists(@".\private$\MyServiceQueue") == false)
{
    MessageQueue.Create(@".\private$\MyServiceQueue",true);
}
MyContractClient proxy = new MyContractClient();
proxy.MyMethod();
proxy.Close();
```

但是同样的, 我们不应硬编码队列的名称, 而是应该将它保存在应用程序的设置里, 然后从应用程序的配置文件中读出队列的名称。同样, 我们还要面对一个挑战, 就是需要持续不断地在应用程序的设置和终结点的地址里保持队列名称的同步。我们可以直接在代理之后的终结点上使用 `QueuedServiceHelper` 类, 但是那会强制我们创建代理 (或

一个ServiceEndpoint实例),用来验证队列。我们可以通过扩展QueuedServiceHelper以简化并支持客户端的队列验证,如例9-6中所示。

例9-6: 扩展 QueuedServiceHelper 用以验证的客户端的队列

```
public static class QueuedServiceHelper
{
    public static void VerifyQueue<T>(string endpointName) where T : class
    {
        ChannelFactory<T> factory = new ChannelFactory<T>(endpointName);
        VerifyQueue(factory.Endpoint);
    }
    public static void VerifyQueue<T>() where T : class
    {
        VerifyQueue<T>("");
    }
    // 与例9-4 相同
    public static void VerifyQueue(ServiceEndpoint endpoint)
    { ... }

    // 更多成员
}
```

例9-6为QueuedServiceHelper类增加了两个方法。一个方法为带参的VerifyQueue<T>()方法,它接收一个终结点名,并使用通道工厂从配置文件中获取终结点,然后调用例9-4中的VerifyQueue()方法。另一个方法则为无参的VerifyQueue<T>()方法,它使用配置文件里指定契约类型的默认终结点。使用QueuedServiceHelper类,例9-5可以被简化为:

```
QueuedServiceHelper.VerifyQueue<IMyContract>();

MyContractClient proxy = new MyContractClient();

proxy.MyMethod();
proxy.Close();
```

注意,客户端需要为每个队列契约验证队列。

## 清除队列

启动宿主时,在队列中可能已经有消息存在了,这些消息是在宿主离线时由MSMQ接收到的,然后,宿主会处理这些消息。队列服务的核心特征之一就是处理这样的情况,从而使得我们能够提供离线服务。当我们在部署队列服务时,这确实是我们所期待的一种方式,但在调试中它却通常会成为一种障碍。设想队列服务的调试会话场景。客户端发出了一些调用,服务开始处理第一个调用,而在单步调试代码的时候,我们注意到存在一个缺陷。我们停止调试,修改服务代码,然后重新启动宿主,但是,它只会处理上次调试会话后余留在队列中的消息,哪怕这些消息会破坏新的服务代码。通常,来自某个

调试会话的消息不应该遗留到下一个会话中。解决的办法是在调试模式下, 关闭宿主时以编程方式清除队列。我们可以使用 `ServiceHost<T>` 对其进行简化, 如例 9-7 所示。

例 9-7: 调试期间在关闭宿主时清除队列

```
public static class QueuedServiceHelper
{
    public static void PurgeQueue(ServiceEndpoint endpoint)
    {
        if(endpoint.Binding is NetMsmqBinding)
        {
            string queueName = GetQueueFromUri(endpoint.Address.Uri);
            if(MessageQueue.Exists(queueName) == true)
            {
                MessageQueue queue = new MessageQueue(queueName);
                queue.Purge();
            }
        }
    }
    // 更多成员
}

public class ServiceHost<T> : ServiceHost
{
    protected override void OnClosing()
    {
        PurgeQueues();
        // 如有必要执行更多清除的操作
        base.OnClosing();
    }
    [Conditional("DEBUG")]
    void PurgeQueues()
    {
        foreach(ServiceEndpoint endpoint in Description.Endpoints)
        {
            QueuedServiceHelper.PurgeQueue(endpoint);
        }
    }
    // 更多成员
}
```

例 9-7 中, `QueuedServiceHelper` 类提供了静态方法 `PurgeQueue()`, 它接收了一个服务的终结点。如果该终结点使用的绑定为 `NetMsmqBinding`, `PurgeQueue()` 方法将从终结点地址中提取出队列的名称, 创建一个新的 `MessageQueue` 对象, 并清除它。`ServiceHost<T>` 重写了 `OnClosing()` 方法, 该方法是在正常关闭宿主时所调用的。然后它将调用私有的 `PurgeQueues()` 方法。`PurgeQueues()` 方法标记了 `Conditional` 属性, 使用 `DEBUG` 作为条件。这意味着尽管 `PurgeQueues()` 的代码段总会进行编译, 但它的调用点却根据 `DEBUG` 符号而定。只有在调试模式下, `OnClosing()` 方法才会真正地调用 `PurgeQueues()` 方法。`PurgeQueues()` 方法会遍历宿主所有的终结点, 为每个终结点调用 `QueuedServiceHelper.PurgeQueue()`。

**注意：**在.NET中使用条件编译时，应该优先选择 Conditional 特性，因为它可以避免使用 #if 显式条件编译带来的缺陷。

## 队列、服务及终结点

WCF 要求我们总是为每个服务的每个终结点专门提供一个队列，也就是说，一个有两个契约的服务需要为两个终结点提供两个对应的队列：

```
<service name = "MyService">
  <endpoint
    address = "net.msmq://localhost/private/MyServiceQueue1"
    binding = "netMsmqBinding"
    contract = "IMyContract"
  />
  <endpoint
    address = "net.msmq://localhost/private/MyServiceQueue2"
    binding = "netMsmqBinding"
    contract = "IMyOtherContract"
  />
</service>
```

理由是客户端事实上是与一个队列，而不是与一个服务终结点进行交互。事实上，甚至可能没有服务，只有队列。两个不同的终结点不能共享一个队列，因为它们会得到彼此的消息。既然在 MSMQ 消息中，WCF 消息是不匹配的，WCF 会主动丢弃那些它认为无效地消息，从而导致丢失这些调用。同理，两个服务的两个多态的终结点也不能共享一个队列，因为它们将得到彼此的消息。

## 公开元数据

WCF 不能在 MSMQ 上交换元数据。因此，对于一个只有队列调用的服务而言，同样可以公开一个 MEX 终结点，或许允许基于 HTTP-GET 进行元数据交换，因为服务的客户端仍然需要一种方法获取服务的描述，并通过它实现绑定。

## 事务

MSMQ 是一种 WCF 事务型资源管理器。当我们创建一个队列时（无论是编码方式还是管理方式），可以将该队列创建成一个事务型的队列。如果队列是事务型的，那么该队列就是持久性的，并且消息会持久化保存到磁盘里。更重要的是，向队列中发送及移除消息将总是会在一个事务中执行。如果试图与队列交互的代码中存在一个环境事务，该队列就会加入到事务中；如果不存在环境事务，MSMQ 将为交互启动一个新的事务。这就好像队列代码被包含在一个设置为 `TransactionScopeOption.Required` 的



TransactionScope 范围中。队列一旦处于某个事务之中，它将与访问它的事务一起提交或回滚。例如，访问队列的事务在向队列发送一个消息后被中止了，那么队列将丢弃该消息。

## 传递及回放

当一个非事务型客户端调用一个队列服务时，如果客户端在调用服务后出错，发送到队列中的消息将无法回滚，而且加入队列中的调用将被分发给服务。然而，如果客户端正在调用的队列服务是在一个事务中，如图 9-3 所示。

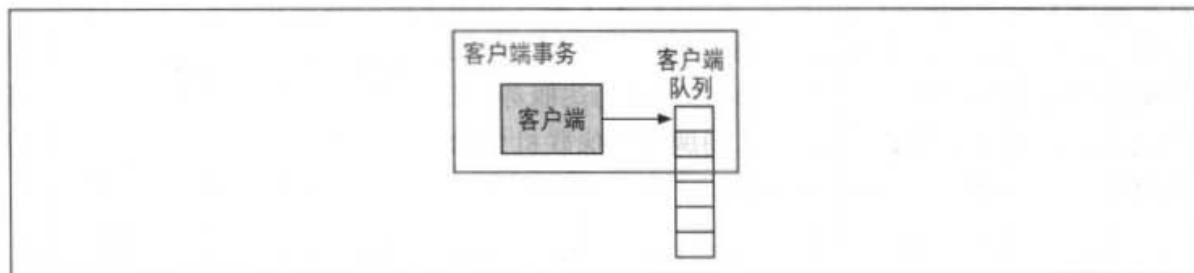


图 9-3：发送消息到客户端队列

客户端调用会被转化为一个 WCF 消息（或多个消息），然后封装在一个 MSMQ 消息（或多个消息）中。如果提交了客户端事务，这些 MSMQ 消息将被发送给队列并保存。假如客户端事务中止了，队列将丢弃这些 MSMQ 消息。实际上，WCF 向队列服务的客户端提供了一套自动取消机制，以应对可能的异步方式的离线调用。正常的联机异步调用无法轻易地，或根本就不可能与事务结合。因为一旦调用被分发，就无法在原来的事务被中止时完成对调用的恢复。与联机异步调用不同，队列服务调用则是专门针对这种事务场景而设计的。不仅如此，客户端还可以在相同事务中与多个队列服务交互。无论何种原因取消了客户端事务，都会自动取消所有到这些队列服务的调用。

## 传递事务

由于客户端可能与服务处在不同的机器上，而且客户端、服务或它们两者都可能是离线的，因此 MSMQ 在客户端也维护了一个队列。这个客户端队列被当作服务队列的一个“代理”使用。在一个远程队列调用的情况中，客户端首先将消息发送给客户端队列，如果客户端连接到网络，MSMQ 将把队列消息从客户端队列传递给服务队列，如图 9-4 所示。

由于 MSMQ 是一个资源管理器，从客户端队列删除消息将创建一个事务（如果队列确实是事务型的）。无论何种原因（诸如，网络故障或服务崩溃），如果 MSMQ 未能将消息传递给服务端队列，从客户端队列移除的消息将被回滚，发往服务端队列的消息则被



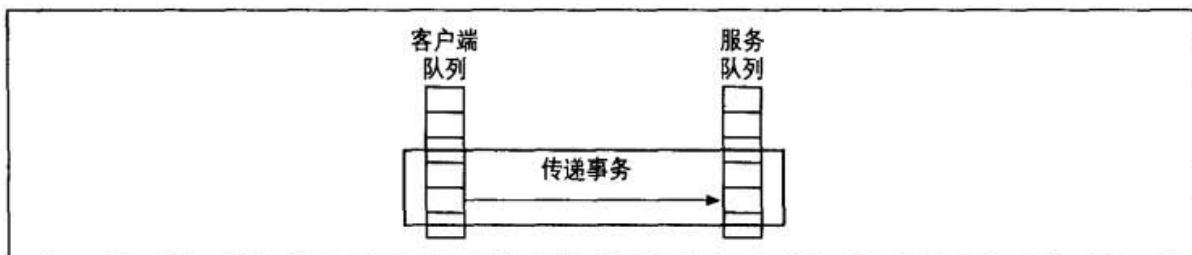


图 9-4：传递事务

取消，最终消息回退到客户端队列，MSMQ可以尝试再次传递消息。当我们能够配置与控制故障处理机制时（稍后将会看到），除了那些无法解决的致命错误，只要传递消息在技术上是可能的（在故障处理机制的范围内），队列服务实际上会乐于获得一种可靠的消息传递机制，保证消息能够从客户端到达服务端。这正是 WCF 为队列服务提供可靠消息传输的有效措施。当然，队列调用不像联机调用那样存在对可靠消息传输协议的直接支持，仅仅是一种模拟机制。

### 回放事务

为了将消息回放给服务，从队列中移除消息时，WCF 会启动一个新的事务（假设该队列是事务型的），如图 9-5 所示。

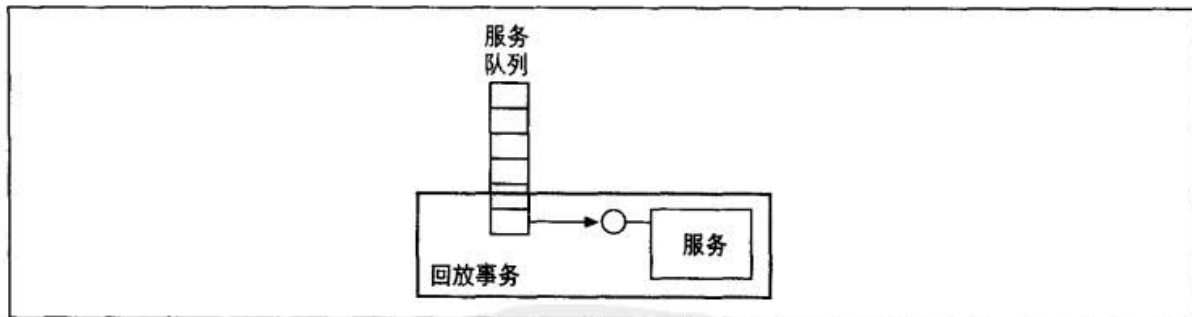


图 9-5：回放事务

在通常情况下，服务被设为参与回放事务。如果中止了回放事务（一般是由于服务端的异常），消息会回滚到队列中，随后 WCF 会在队列中检测到它并将它再一次发送给服务。这实际上构成了一种自动重试机制。因此，我们应该保证服务对队列调用的处理相对的短，否则就会面临回放事务被中止的风险。

一个重要的关注点是不能将队列调用与耗时长的异步调用等同起来。如果调用以异步方式分发，服务可以从容地处理接收到的调用，通常不会引入任何客户端事务。对于队列调用而言，由于服务具有一个传入事务，服务处理就不应该使得回放事务超时，因为那样会触发 WCF 不断重试该调用。

## 服务事务配置

正如刚刚所演示的那样，事实上，在每个队列调用中（假定是事务型队列）会涉及三种事务：客户端、传递与回放，如图 9-6 所示。

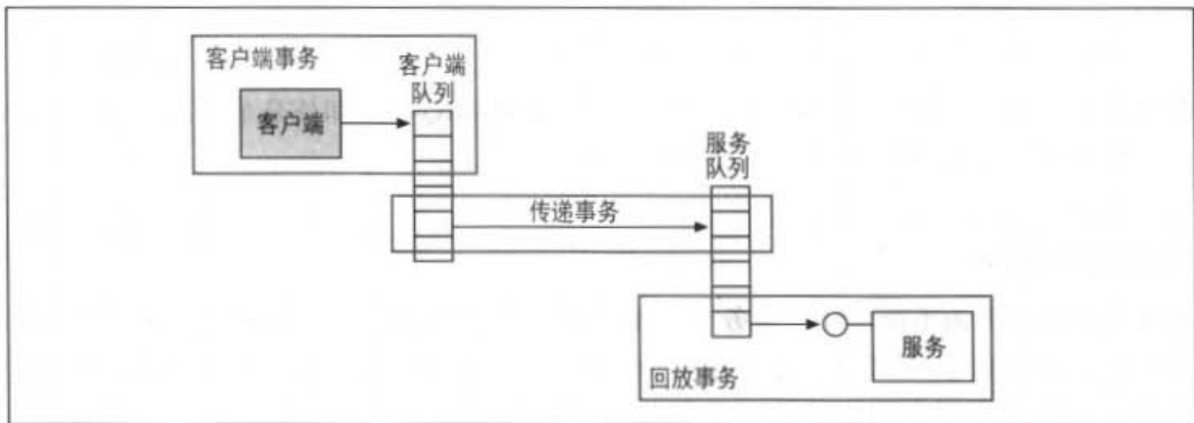


图 9-6：队列调用与事务

从设计角度看，如果存在事务，我们很少在设计图中表示传递事务，而仅仅是将它认为是理所当然。此外，服务不会参与到客户端事务中，因此，我们在第 7 章中提到的四种逻辑上的事务型模式（Client、Client/Service、Service、None）实际上无法适用此处。若服务契约操作被配置为 `TransactionFlowOption.Allowed` 或 `TransactionFlowOption.NotAllowed`，两者不会存在差异，客户端事务不会传播到服务中。不仅如此，队列契约甚至禁止被设置为 `TransactionFlowOption.Mandatory`，在服务加载时会对此进行验证。因此，目前真正的问题在于回放事务与服务事务配置之间的关系。

## 参与回放事务

从 WCF 的角度看，它将回放事务当作是服务的传入事务。为了参与回放事务，服务需要将操作行为特性中的 `TransactionScopeRequired` 属性设置为 `true`，如例 9-8 以及图 9-5 所示。

### 例 9-8：参与回放事务

```
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    {
```

```
[OperationBehavior(TransactionScopeRequired = true)]
public void MyMethod()
{
    Transaction transaction = Transaction.Current;
    Debug.Assert(transaction.TransactionInformation.
        DistributedIdentifier != Guid.Empty);
}
}
```

值得注意的是，在例9-8中不管是 MSMQ3.0 还是 MSMQ4.0，即使是单个服务与单个回放，每个事务总是使用 DTC 管理事务。

### 忽略回放事务

如果服务被配置为不使用任何事务（如例9-9和图9-7所示），WCF 还是会使用一个事务从队列中读取消息，只不过该事务总是会成功的（除非 MSMQ 本身出现某种无法预见的故障）。服务自身的异常和失败不会中止回放服务。

#### 例 9-9：忽略回放事务

```
[ServiceContract]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    public void MyMethod()
    {
        Transaction transaction = Transaction.Current;
        Debug.Assert(transaction == null);
    }
}
```

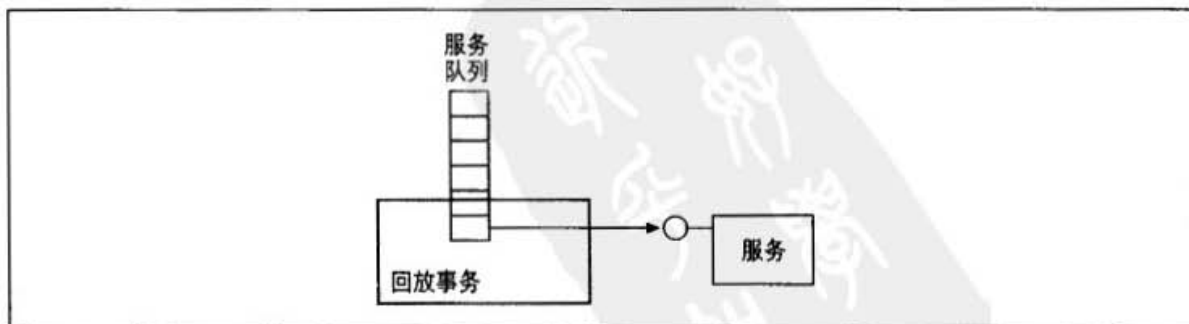


图 9-7：忽略回放事务

在回放失败的情况下，没有参与回放事务的服务无法执行 WCF 的自动重试功能，而且在出队事务（De-Queued Transaction）提交时，回放调用可能会失败。之所以让服务放

弃自动重试的优势，主要目的是为了耗时较长的处理。如果不参与回放事务，服务调用就不必考虑处理耗时的长短。

### 使用一个独立事务

我们也可以在编写服务的时候手动使用一个新的事务，如例 9-10 和图 9-8 所示。

例 9-10：使用一个新的事务

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    public void MyMethod()
    {
        using(TransactionScope scope =
            new TransactionScope(TransactionScopeOption.RequiresNew))
        {
            ...
            scope.Complete();
        }
    }
}
```

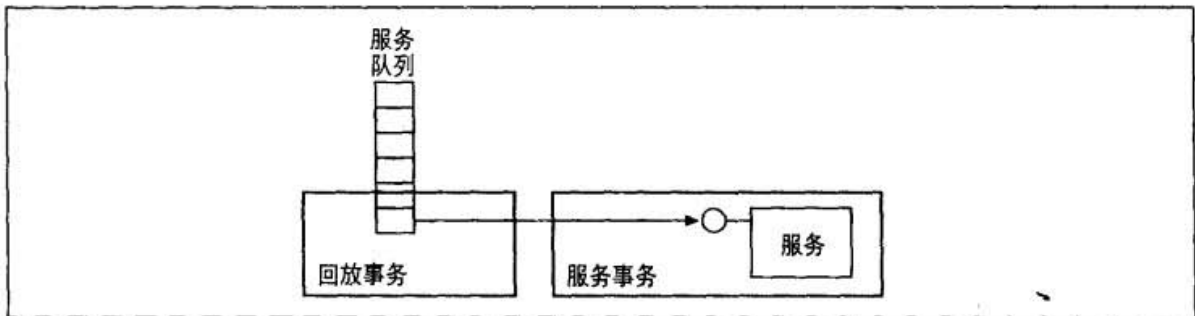


图 9-8：使用一个新的事务

当服务使用它自己的新事务时，它仍需避免使用回放事务（将 `TransactionScopeRequired` 的值默认设为 `false`），这样就不会影响到任何形式的回放事务。毫无疑问，这就否定了自动重试机制的优势。通过使用一个独立于回放事务的新事务，给服务提供一个执行它拥有的事务型工作的机会。通常，在事务的保护下调用以及执行队列操作时，我们应该将服务配置为使用它自己拥有的事务，即使失败了，也无需重试。

### 临时队列

到目前为止所描述的 MSMQ 队列既是持久性的，也是事务型的。消息被持久化到磁盘上，向队列发送消息与从队列中读取消息都属于事务型的操作。MSMQ 也支持非事务型的易失性队列。这样的队列被保存在内存里，并且不会使用事务。由于消息被保存在内

存中,如果机器关闭或者崩溃,在队列中的所有消息都会丢失。当我们创建一个队列(或者使用MSMQ管理工具,或者通过编程)时,我们可以将它配置为事务型的或者非事务型的,一旦作出选择,那么在队列的整个生命周期内都将无法改变。临时队列无法提供任何事务型消息系统所具有的优点,例如,自动取消、可靠的消息传递以及自动重试。尽管如此,WCF仍然可以使用临时队列。MsmqBindingBase (NetMsmqBinding类的基类)定义了Boolean属性Durable,它的默认值为true。

```
public abstract class MsmqBindingBase : Binding,...
{
    public bool Durable
    {get;set;}
    public bool ExactlyOnce
    {get;set;}
    // 更多成员
}
public class NetMsmqBinding : MsmqBindingBase
{...}
```

当该属性被设置为false时,WCF将不会在访问队列时使用事务。客户端与服务使用的Durable属性必须是相同的。队列本身应该被配置为易失性的。使用一个易失性队列时,如果客户端事务被中止,一个或多个消息会驻留在队列中,然后交付给服务。如果回放事务被中止了,消息则会丢失。

由于不具备可靠的消息传递机制,在使用一个易失性队列时,WCF还要求我们将绑定的ExactlyOnce属性设置为false(默认值是true),否则,WCF在加载服务时会抛出一个InvalidOperationException异常。因此,下面是一个针对临时队列的固定配置:

```
<netMsmqBinding>
  <binding name="VolatileQueue" durable="false" exactlyOnce="false">
  </binding>
</netMsmqBinding>
```

## 实例管理

契约会话模式与服务实例模式对于队列调用的行为、调用回放到服务的方式、程序的整个 workflow 以及调用队列服务时的假设都有着极其深远的影响。

### 单调队列服务

对于单调服务,客户端无法预知到是否最终会调用一个放入队列的单调服务。客户端所能看到的只有契约上的会话模式。假如契约上的SessionMode是SessionMode.Allowed或SessionMode.NotAllowed中的任意一个,队列调用(使用NetMsmqBinding)就

会被认为是无会话的。诸如这种无会话的终结点，对它们的所有调用都被放置在单独的 MSMQ 消息中，也就是说使用这种终结点的代理在每次调用方法时，都会产生一个单独的 MSMQ 消息。

### 非事务型客户端

当一个没有环境事务的客户端访问一个不支持会话的队列终结点时（如例 9-11 所示），为每次调用生成的 MSMQ 消息都会在方法被调用后被立即发送给队列。如果客户端在方法调用后出现异常，那之前发送的消息也会被传递给服务，不会被服务拒绝。

例 9-11：一个无会话的队列终结点的非事务型客户端

```
using(TransactionScope scope =
    new TransactionScope(TransactionScopeOption.Suppress))
{
    MyContractClient proxy = new MyContractClient();

    proxy.MyMethod();// 此处会将消息传递给队列
    proxy.MyMethod();// 此处会将消息传递给队列

    proxy.Close();
}
```

### 事务型客户端

当一个使用事务的客户端（即一段存在环境事务的客户端代码）访问一个不支持会话的队列终结点时，如例 9-12 所示，只有在提交客户端的事务时，所有（与每个调用相对应的）消息才会被发送给队列。如果客户端事务中止，所有的消息都将被队列拒绝，并且所有的调用都会被取消。

例 9-12：一个无会话的队列终结点的事务型客户端

```
using(TransactionScope scope = new TransactionScope())
{
    MyContractClient proxy = new MyContractClient();

    proxy.MyMethod();// 消息写入队列
    proxy.MyMethod();// 消息写入队列

    proxy.Close();
    scope.Complete();
} // 此处会将消息提交到队列
```

代理与环境事务之间没有任何联系。假如客户端使用了一个事务范围（如例 9-12 所示），那么客户端既可以在范围中，也可以在范围外关闭代理，甚至可以在事务完成之后或一个新的事务中继续使用代理。客户端同样可以在调用 Complete() 方法之前或之后关闭代理。

## 单调处理

在宿主端, 队列调用是被分别分发给服务的, 而且每个调用都会被分发给不同的服务实例。即使在服务实例模式为 per-session 的情况下也是如此。因此考虑到易读性, 我建议使用一个无会话的队列契约时, 总是显式地将服务配置为单调方式, 并将契约配置为禁止会话:

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyContract
{...}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{...}
```

如同联机单调服务那样, 服务实例在每次调用之后都会被销毁。单调服务是否为事务型并不确定, 如果是事务型, 而且中止了回滚事务, 那么只有特定的调用为了重试会被回滚到队列中。在后面我们会看到, 由于并发回放以及 WCF 的故障处理行为, 对一个单调队列服务的调用可以以任意的顺序执行并完成, 而且客户端也不能假定调用的顺序。注意, 哪怕是由一个事务型客户端发送的每个调用, 也可能某些失败而某些成功。绝对不要对某个单调队列服务的调用顺序做出任何假设。

## 会话型队列服务

若要开发并配置一个支持会话的队列服务, 服务契约必须被配置为 SessionMode.Required:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{...}

class MyService : IMyContract
{...}
```

当客户端针对一个会话型队列终结点将调用加入队列时, 整个会话产生的所有调用都会被分组放入一个单独的 MSMQ 信息中。一旦这条单独的消息被分发给服务处理时, WCF 就会创建一个新的服务实例, 专门用于处理在消息中的所有调用。消息中的所有调用将按原来的调用顺序交由该实例分别处理。由于让客户端来关闭代理并没有什么意义, 所以在最后一个调用之后, 实例会被自动销毁。在客户端与服务两端, WCF 将会为它们各自提供一个唯一的会话 ID。不过, 这两个 ID 完全不相关。为了更接近会话的语义, 对宿主端相同实例的所有调用都将共享相同的会话 ID。



## 客户端与事务

为了调用一个会话型队列终结点的代理，客户端必须有一个环境事务，否则在调用时将引发一个 `InvalidOperationException` 异常：

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}

using(TransactionScope scope =
    new TransactionScope(TransactionScopeOption.Suppress))
{
    MyContractClient proxy = new MyContractClient();
    proxy.MyMethod();// 抛出 InvalidOperationException 异常
    proxy.MyMethod();

    proxy.Close();
}
```

当一个事务型客户端使用一个会话型队列终结点时，WCF会在事务提交的时候向队列发送一条单独的消息，如果事务中止，则该消息将被队列丢弃：

```
using(TransactionScope scope = new TransactionScope())
{
    MyContractClient proxy = new MyContractClient();

    proxy.MyMethod();
    proxy.MyMethod();

    proxy.Close();// 完成对消息的组成，并将其写入到队列中

    scope.Complete();
} // 在此处单独消息会被提交给队列
```

注意，殊为重要的是代理准备的单独消息必须在相同的客户端事务中发送给队列，而且客户端必须在事务内部结束会话。如果客户端在事务完成之前没有关闭代理，那么该事务将总是被中止：

```
MyContractClient proxy = new MyContractClient();
using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod();
    proxy.MyMethod();

    scope.Complete();
} // 中止事务
proxy.Close();
```

这一行为还有一个值得关注的副作用，如果我们将代理保存在成员变量中的一个队列会

话终结点中,是没有任何意义的。因为该代理只能在一个单独的事务中使用一次,而无法跨越多个客户端事务重用。客户端只能在一个单独的事务中使用这个代理,并在事务范围内关闭它。

客户端不仅必须在事务结束之前关闭代理,而且在使用一个事务范围的时候,客户端还必须在完成事务(调用Complete方法)以前就关闭代理。因为在关闭一个指向某个队列会话终结点的代理时,需要访问当前的环境事务,如果在调用Complete()方法之后执行这一过程,是无法实现的。如果试图采用如下方式执行,就会造成InvalidOperationException异常:

```
MyContractClient proxy = new MyContractClient();
using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod();
    proxy.MyMethod();

    scope.Complete();
    proxy.Close(); // 事务中止
}
```

根据这一要求,我们可以得出一个推论,就是我们不能简单地将using语句叠加在一起使用,因为这样可能会造成Dispose()方法的调用顺序出错,使得首先销毁的是事务范围,而不是代理对象:

```
using(MyContractClient proxy = new MyContractClient())
using(TransactionScope scope = new TransactionScope())
{
    proxy.MyMethod();
    proxy.MyMethod();

    scope.Complete();
} // 事务中止
```

## 服务与事务

一个会话型队列服务必须被配置为在所有操作中使用事务,方法是将TransactionScopeRequired属性设置为true。如果无法做到,就会导致所有的回放事务被中止。此外,服务必须为实例提供事务关联度,方法是将会话中除最后一个操作之外的所有操作的TransactionAutoComplete属性设置为false。由于WCF的一个设计缺陷,服务不能只依赖于将TransactionAutoCompleteOnSessionClose设置为true,为实现这一目的,必须要求会话中的最后一个方法调用以自动或手动方式完成事务。例9-13是一个用于实现会话型队列服务的模板,它假定MyMethod3()方法是会话中的最后一个操作调用。

## 例 9-13: 实现一个会话型队列服务

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod1();

    [OperationContract(IsOneWay = true)]
    void MyMethod2();

    [OperationContract(IsOneWay = true)]
    void MyMethod3();
}

class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true,
                       TransactionAutoComplete = false)]
    public void MyMethod1()
    { ... }

    [OperationBehavior(TransactionScopeRequired = true,
                       TransactionAutoComplete = false)]
    public void MyMethod2()
    { ... }

    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod3()
    { ... }
}
```

在服务代码中假定某个特定的方法为会话中的最后一个方法, 显然不切实际。不过我们可以使用分步操作, 指定用于结束会话的方法, 从而在一定程度上缓解这一问题。使用例 9-13 相同的定义, 可以编写如下代码:

```
[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract(IsOneWay = true, IsTerminating = false)]
    void MyMethod1();

    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = false)]
    void MyMethod2();

    [OperationContract(IsOneWay = true, IsInitiating = false, IsTerminating = true)]
    void MyMethod3();
}
```

即便如此, 这样的解决方案仍非万无一失。如果客户端始终不调用终止操作, 此时会话中止事务。还有一种方案就是为契约添加一个显式的 CompleteTransaction() 操作, 该契约的唯一目的就是完成事务并结束会话。我们需要在会话结束时调用该方法, 对于这样的需求, 我们应该明确地对其进行编档:

```

[ServiceContract(SessionMode = SessionMode.Required)]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod1();

    [OperationContract(IsOneWay = true)]
    void MyMethod2();

    [OperationContract(IsOneWay = true)]
    void CompleteTransaction();
}

class MyService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void MyMethod1()
    { ... }

    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void MyMethod2()
    { ... }

    [OperationBehavior(TransactionScopeRequired = true)]
    public void CompleteTransaction()
    { ... }
}

```

综上所述, 我们应避免使用会话型队列服务, 因为它十分脆弱, 并且引入了紧密的耦合。

## 单例服务

一个事务型单例队列服务永远无法使用会话, 它只能实现无会话的契约。将 `SessionMode` 配置为 `SessionMode.Allowed` 或 `SessionMode.NotAllowed`, 都会得到相同的结果: 一个无会话的交互。因此, 最好总是显式地将一个单例队列服务的契约配置为无会话形式:

```

[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyContract
{ ... }

[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
class MyService : IMyContract
{ ... }

```

一个非事务型单例队列服务在实例化方面与一个普通的 WCF 单例服务相似。我们不用考虑客户端和代理, 代理的各个调用都会被打包到不同的 MSMQ 信息中去, 并各自分发给单例服务, 这与单调服务相似。不过, 与单调服务不同的是, 所有的这些调用都将回放给同一个服务实例。

对于一个事务型单例队列服务而言,默认情况下它的执行方式与单调服务相似,因为在每个完成事务的调用之后,WCF都将释放服务的唯一实例。一个单例服务与一个真正的单调服务之间唯一的区别在于:不管队列消息的数目如何,WCF最多只允许单例服务拥有一个实例。应用第7章对于状态相关的事务型单例服务所使用的技术,我们同样可以通过将`ReleaseServiceInstanceOnTransactionComplete`属性设置为`false`,以恢复单例语义。回顾第7章所述,对于单例服务的约束就是至少有一个操作的`TransactionScopeRequired`属性值被设置成`true`。例9-14演示了一个用于实现事务型单例队列服务的模板。

例 9-14: 事务型单例队列服务

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]
interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}

[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single,
                 ReleaseServiceInstanceOnTransactionComplete = false)]
class MySingleton : IMyContract, IDisposable
{
    [OperationContract(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }
    // 更多成员
}
```

---

**注意:** 一个事务型单例服务无法实现会话契约,除非所有的客户端在每个会话中只调用一次服务,然而这却使得会话失去意义。因为只有会话服务能够关闭相同会话中的多个方法的自动完成选项。我们应避免使用一个具有会话的事务型单例队列服务。

---

## 调用与顺序

因为多个调用被打包成各自独立的MSMQ消息,所以各个调用被分派到服务实例上的顺序也可能因为消息的重试或事务而打乱。此外,即便一个事务客户端分发的调用既可能失败,也可能成功,调用也可以以任意的顺序完成。绝对不要对单例服务的调用顺序做任何假设。

## 并发管理

与联机服务一样,我们使用`ConcurrencyMode`属性管理队列消息的并发回放。在单调服务中,所有的队列消息在离开队列后就立即分发给不同的实例,直到达到了预先配置

的限流上限。此时无需将服务配置为重入 (Reentrancy) 以支持回调, 因为操作的上下文不存在回调的引用。同样也没有必要配置为多个并发访问, 因为不会出现两个消息共享一个实例的情况。总之, 对于一个单调队列服务而言, 是无需考虑并发模式的。

对于一个会话队列服务, 我们必须将服务配置为 `ConcurrencyMode.Single` 模式。因为只有该模式允许我们关闭事务的自动完成功能, 而这对于维护会话的语义是至关重要的。尽管消息中包含了多个调用, 但是服务实例在同一时刻只会执行消息中的一个调用。

只有单例队列服务才在并发模式的配置上有多种选择。如果单例服务被配置为 `ConcurrencyMode.Single`, WCF 会立即从队列中取出所有的消息 (直至达到线程池与限流的上限), 并把它们加入到由实例锁维护的内部队列中。服务实例在同一时刻只会被分发给一个调用。如果单例被配置为 `ConcurrencyMode.Multiple`, WCF 会立即从队列中取出所有的消息 (直至达到线程池与限流的上限), 并将它们并发地分发给服务的唯一实例。显而易见, 服务的唯一实例必须提供对其状态的同步访问。如果单例服务同时也是事务型的, 那么提高每个事务的隔离级别, 会很容易引发事务型死锁。不仅如此, 一个配置为 `ConcurrencyMode.Multiple` 的事务单例服务还必须将 `ReleaseServiceInstanceOnTransactionComplete` 设置为 `false`, 而且必须包含至少一个将 `TransactionScopeRequired` 设置为 `true` 的操作:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                 ConcurrencyMode = ConcurrencyMode.Multiple,
                 ReleaseServiceInstanceOnTransactionComplete = false)]
class MySingleton : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }
}
```

## 限流

我们可以通过队列服务的限流来控制服务要承受的压力, 以此避免将服务的负载全部转化为压力。限流中的一个重要数值是并发回放的数目。限流是一种抑制执行消息数目的非常有效的途径, 因为一旦超出并发调用的数目 (总体压力), 那么超出数目后的消息将保留在队列中。举例来说, 如果一个服务在队列中有 50 条消息, 但它可能并不希望有 50 个并发的实例与工作线程。对单调服务而言, 限流控制了允许并发运行的实例的总数目 (以及它们隐含的资源消耗)。对会话服务而言, 限流则控制了允许发生的会话数目。对于一个队列单例服务, 我们可以将限流与 `ConcurrencyMode.Multiple` 结合起来, 控制并发执行的消息数 (压力) 以及保持在队列中的消息数 (缓冲负载)。



## 传递故障

正如在第6章中讨论的那样，一个联机调用可能因为通信故障或服务端的错误而导致失败。同样，一个队列调用也可能因为传递故障 (Delivery Failure) 或服务端的回放错误而失败。WCF为两种类型的错误都提供了专门的错误处理机制，理解它们并将错误处理逻辑整合其中是使用队列服务的一项重要内容。

虽然从技术上讲，MSMQ能够确保消息的传递，但是仍然存在多个无法传递消息的情形。以下内容列出了部分传递故障的情形：

### 超时与过期 (Time Out and Expiration)

正如我们即将看到的那样，每个消息都有一个时间戳，消息必须在超时值内被传递并处理，否则将导致传递失败。

### 安全性不匹配 (Security Mismatch)

如果消息中的安全凭据 (或被选中的身份验证机制本身) 与服务所期望的不相匹配，服务将拒绝该消息。

### 事务不匹配 (Transactional Mismatch)

客户端不能使用一个本地的非事务型队列向一个服务端的事务型队列发送消息。

### 网络问题 (Network Problems)

如果底层网络出现了故障或者不可靠，消息就可能永远也无法到达服务。

### 机器崩溃 (Machine Crashes)

服务机器可能因为软件或硬件的故障而崩溃，使得队列无法接收消息。

### 清除 (Purges)

就算消息被成功交付，管理者 (或任何应用程序通过编码方式) 可以清除队列中的消息，以避免服务处理这些消息。

### 超出配额 (Quota Breach)

每个队列都有一个配额，用于控制它可以持有的数据的最大容量。如果超过了配额，后来的消息将会被拒绝。

每次发生传递故障后，消息都会返回到客户端队列，之后MSMQ会不断地重新传递该消息。尽管在某些情况下，例如中间网络故障或队列配额的问题，重新传递可能会最终成功，但是在很多情况下MSMQ永远都不能成功地传递消息。事实上，就实际情况而言，大量的重试是无法接受的，它可能会对系统带来严重的危机。传递故障处理能够解决下列问题：MSMQ如何获知自己是否应该继续重试？需要经历多少次重试才应该放弃？需要经历多长时间才应该放弃？又应该如何处理失败的消息？



MsmqBindingBase 提供了一系列属性，可以用来控制对传递故障的处理：

```
public abstract class MsmqBindingBase : Binding,...
{
    public TimeSpan TimeToLive
    {get;set;}

    //DLQ 设置
    public Uri CustomDeadLetterQueue
    {get;set;}
    public DeadLetterQueue DeadLetterQueue
    {get;set;}

    // 更多成员
}
```

## 死信队列

在消息传输系统中，发生一个明显的传递故障之后，消息会进入一个被称为死信队列 (Deat-Letter Queue, DLQ) 的特殊队列中。DLQ 有点类似于传统的邮局里的死信邮箱。目前所讨论的传递故障，不仅代表消息无法到达服务端，还代表了无法提交回放事务。注意，服务即使无法处理回放，但它仍旧可以提交回放事务。MSMQ 在客户端与服务端之间不断地确认彼此接收以及处理的消息。如果服务端的 MSMQ 成功地从服务端队列中接收并获取消息（即回放事务被提交），它会向客户端 MSMQ 发送一个肯定的确认通知 (ACK)。服务端 MSMQ 也可以向客户端 MSMQ 发送一个否定的确认通知 (NACK)。当客户端 MSMQ 收到一个 NACK 的时候，它会把消息发送到 DLQ 中。如果客户端 MSMQ 收到的既不是 ACK 也不是 NACK，该消息被认为处在不确定 (In-Doubt) 状态。

在 MSMQ 3.0 中（即在 Windows XP 和 Windows Server 2003 平台上），死信队列是在操作系统范围内的一个队列。来自任何应用程序的所有传递失败的消息都将到达这个单独的消息库中。在 MSMQ 4.0（即在 Windows Vista 平台上）中，我们可以配置一个和特定应用程序相关的 DLQ，只有属于指定服务的消息才能到达。这样的死信队列大大地简化了管理者与开发者的工作。

---

**注意：** 在处理一个非持久化队列时，传递失败的非事务型消息将到达一个特殊系统范围内的持久化 DLQ。

---

## 存活时间

在 MSMQ 中，每一个消息都携带一个时间戳，它在消息第一次被发送到客户端队列时被初始化。此外，每个 WCF 队列消息都有一个超时值，该值由 MsmqBindingBase 类的

TimeToLive 属性控制。WCF 在发送某个消息到客户端队列之后，会强制要求消息必须在预先设好的超时值内被传递并处理。注意，此时消息不仅需要被成功地传递到服务端队列，消息对应的调用还必须得到处理。因此，TimeToLive 属性有点类似于联机服务绑定中的 SendTimeout 属性。TimeToLive 属性仅与发送消息的客户端有关，而对服务端没有任何影响，服务也不能改变它的值。TimeToLive 属性的值默认为一天。若在 TimeToLive 允许的最长时间内不断地重发但仍无法将消息传递（和处理），MSMQ 将停止重发并将该消息移到预先设定的 DLQ 中去。

我们既可以通过编程方式也可以通过管理方式设定存活时间的值。例如，如下的例子演示了如何通过配置文件将存活时间设置为 5min：

```
<bindings>
  <netMsmqBinding>
    <binding name = "ShortTimeout" timeToLive = "00:05:00">
    </binding>
  </netMsmqBinding>
</bindings>
```

之所以会设置一个较短的超时值，主要是针对那些对时间比较敏感的方法调用，此类调用必须被及时处理。但是，对时间敏感的队列调用总的来说违背了离线队列调用的精神，因为调用对时间的敏感程度越高，使用队列服务就越成问题。把存活时间当作让管理者注意到消息没有被传递的最后一招，而不是把它当作一种强调消息敏感度的业务手段，才是看待存活时间的正确观点。

## 配置死信队列

MsmqBindingBase 提供了枚举类型 DeadLetterQueue 的 DeadLetterQueue 属性：

```
public enum DeadLetterQueue
{
    None,
    System,
    Custom
}
```

当 DeadLetterQueue 属性被设置为 DeadLetterQueue.None 时，WCF 不会使用死信队列。出现传递故障后，WCF 将自动丢弃该消息，就好像调用从来没有发生过那样。DeadLetterQueue.System 是该属性的默认值。顾名思义，它使用了一个系统范围内的 DLQ，在出现传递故障后，WCF 会把该消息从客户端队列移动到系统范围内的 DLQ 中去。

当 DeadLetterQueue 属性被设置为 DeadLetterQueue.Custom 时，应用程序可以使用一个专用的 DLQ。DeadLetterQueue.Custom 需要使用 MSMQ 4.0，WCF 在调用时会

验证这一点。此外，WCF还要求应用程序在绑定的CustomDeadLetterQueue属性中指定专用的DLQ名称。CustomDeadLetterQueue的默认值是null，但在使用DeadLetterQueue.Custom时，CustomDeadLetterQueue的值却不能为null：

```
<netMsmqBinding>
  <binding name = "CustomDLQ"
    deadLetterQueue = "Custom"
    customDeadLetterQueue = "net.msmq://localhost/private/MyCustomDLQ">
  </binding>
</netMsmqBinding>
```

反之，当DeadLetterQueue被设置为除DeadLetterQueue.Custom之外的其他任何值时，CustomDeadLetterQueue必须为null。

我们必须认识到定制DLQ只不过是另一个MSMQ队列。客户端的开发人员完全可以部署一个能够处理内部消息的DLQ服务。WCF在MSMQ 4.0上所做的一切仅仅是为了在检测到某个故障时，能将消息自动移至DLQ中。

## 检验定制DLQ

如果需要一个定制DLQ，那么就像其他任何队列一样，客户端需要在运行期间，在发出队列调用之前检验定制DLQ是否存在，如果不存在，则需要负责它的创建。按照之前介绍的模式，我们可以用QueuedServiceHelper.VerifyQueue()方法完成封装，从而实现这一过程的自动化，如例9-15所示。

例9-15：检验一个定制DLQ

```
public static class QueuedServiceHelper
{
    public static void VerifyQueue(ServiceEndpoint endpoint)
    {
        if(endpoint.Binding is NetMsmqBinding)
        {
            string queue = GetQueueFromUri(endpoint.Address.Uri);
            if(MessageQueue.Exists(queue) == false)
            {
                MessageQueue.Create(queue, true);
            }
            NetMsmqBinding binding = endpoint.Binding as NetMsmqBinding;
            if(binding.DeadLetterQueue == DeadLetterQueue.Custom)
            {
                Debug.Assert(binding.CustomDeadLetterQueue != null);
                string DLQ = GetQueueFromUri(binding.CustomDeadLetterQueue);
                if(MessageQueue.Exists(DLQ) == false)
                {
                    MessageQueue.Create(DLQ, true);
                }
            }
        }
    }
}
```

```
    }  
  }  
  // 更多成员  
}
```

## 处理死信队列

客户端需要处理 DLQ 中累积的消息。对于系统范围的 DLQ 来说, 客户端可以提供一个能够支持系统上所有队列终结点中的所有契约的大型服务 (mega-service), 使用该服务能够处理所有的失败消息。这显然是一个不切实际的想法, 因为这样的服务不可能知晓所有的队列契约, 更不用说针对所有应用程序执行有意义的处理。唯一可行的办法是, 限制客户端在每个系统中最多只使用一个单独的队列服务。或者, 我们也可以使用 `System.Messaging` 编写一个定制的应用程序, 直接管理并操作系统 DLQ。该应用程序将解析并提取相关的消息, 再对它们进行处理。这种方法带来的问题 (除了涉及大量的工作) 是, 假如消息是受保护的并被加密了 (它们本应如此), 应用程序将很难处理并区分它们。在现实情况下, 唯一可行的解决方案是使用 MSMQ 4.0 提供的一个通用的客户端环境, 也就是一个定制 DLQ。使用定制 DLQ 时, 我们还要提供一个客户端服务, 该服务的队列就是为应用程序定制的 DLQ, 服务将根据应用程序指定的要求处理失败的消息。

## 定义 DLQ 服务

实现 DLQ 服务的做法与其他任何队列服务没有什么区别, 唯一的要求是 DLQ 服务必须是原来的服务契约的多态实现。如果引入了多个队列终结点, 则需要为每个终结点提供一个 DLQ。例 9-16 演示了一种可能的配置。

例 9-16: DLQ 服务配置文件

```
//////////////////// 客户端 //////////////////////////////////////  
<system.serviceModel>  
  <client>  
    <endpoint  
      address = "net.msmq://localhost/private/MyServiceQueue"  
      binding = "netMsmqBinding"  
      bindingConfiguration = "MyCustomDLQ"  
      contract = "IMyContract"  
    />  
  </client>  
  <bindings>  
    <netMsmqBinding>  
      <binding name = "MyCustomDLQ" deadLetterQueue = "Custom"  
        customDeadLetterQueue = "net.msmq://localhost/private/MyCustomDLQ">  
      </binding>  
    </netMsmqBinding>  
  </bindings>  
</system.serviceModel>
```

```

////////// DLQ 服务端 //////////
<system.serviceModel>
  <services>
    <service name = "MyDLQService">
      <endpoint
        address = "net.msmq://localhost/private/MyCustomDLQ"
        binding = "netMsmqBinding"
        contract = "IMyContract"
      />
    </service>
  </services>
</system.serviceModel>

```

上述的客户端配置文件定义了一个IMyContract契约的队列终结点。客户端使用了一个定制绑定节以实现对定制DLQ地址的定义。DLQ队列服务（可能处在另一台独立的机器上）同样实现了IMyContract契约，并使用客户端定义的DLQ作为自己的地址。

## 故障属性

通常，DLQ服务需要知道队列调用出现传递故障的原因。为此，WCF提供了MsmqMessageProperty类用于找出故障原因以及消息当前的状态。MsmqMessageProperty被定义在System.ServiceModel.Channels命名空间中：

```

public sealed class MsmqMessageProperty
{
    public const string Name = "MsmqMessageProperty";

    public int AbortCount
    {get;internal set;}
    public DeliveryFailure? DeliveryFailure
    {get;}
    public DeliveryStatus? DeliveryStatus
    {get;}
    public int MoveCount
    {get;internal set;}
    // 更多成员
}

```

DLQ服务需要从操作上下文的传入消息属性中获得MsmqMessageProperty：

```

public sealed class OperationContext : ...
{
    public MessageProperties IncomingMessageProperties
    {get;}
    // 更多成员
}
public sealed class MessageProperties : IDictionary<string,object>,...
{
    public object this[string name]
    {get;set;}
}

```

```
// 更多成员  
}
```

当一个消息传入到DLQ时，WCF将向消息的属性集合添加一个MsmqMessageProperty的实例，用以说明消息传递故障的细节。MessageProperties只是一个消息属性的集合，我们可以使用字符串作为键访问其中的值。若要获得MsmqMessageProperty，可以使用常量MsmqMessageProperty.Name作为键，如例9-17所示。

#### 例9-17：获得MsmqMessageProperty

```
[ServiceContract(SessionMode = SessionMode.NotAllowed)]  
interface IMyContract  
{  
    [OperationContract(IsOneWay = true)]  
    void MyMethod();  
}  
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]  
class MyDLQService : IMyContract  
{  
    [OperationBehavior(TransactionScopeRequired = true)]  
    public void MyMethod()  
    {  
        MsmqMessageProperty msmqProperty = OperationContext.Current.  
            IncomingMessageProperties[MsmqMessageProperty.Name] as MsmqMessageProperty;  
  
        Debug.Assert(msmqProperty != null);  
        // 处理 msmqProperty  
    }  
}
```

注意，尽管在例9-17中实际运用了迄今为止讨论过的会话模式、实例管理以及事务，然而归根结底，DLQ服务仍然只是另一个队列服务而已。

MsmqMessageProperty的属性包含了失败原因的细节信息，同时提供了一些上下文信息。MoveCount记录了试图将消息发送到服务的次数。AbortCount记录了试图从队列中读取消息的次数。AbortCount与企图恢复系统的重试无关，因为它的失败是由于MSMQ的原因，通常不需要考虑。DeliveryStatus是一个可以为空的DeliveryStatus枚举类型，它的定义为：

```
public enum DeliveryStatus  
{  
    InDoubt,  
    NotDelivered  
}
```

除非确定消息不能被传递（接收到一个NACK），否则DeliveryStatus会被设置为DeliveryStatus.InDoubt。例如，过期的消息会被认为是不确定的（In-Doubt），因为服务在向客户端确认收到该消息之前，已经超过了它们的存活时间。

DeliveryFailure属性是一个可以为空的DeliveryFailure枚举类型，它的定义（未包含特殊的数字值）如下：

```
public enum DeliveryFailure
{
    AccessDenied,
    NotTransactionalMessage,
    Purged,
    QueueExceedMaximumSize,
    ReachQueueTimeout,
    ReceiveTimeout,
    Unknown
    // 更多成员
}
```

### 实现一个 DLQ 服务

DLQ服务无法影响消息的属性，例如延长它的存活时间。处理传递故障时一般会引入一些补偿类型的事务：通知管理员；或者尝试重新发送一个新的消息；或者重新发送包含了延长的超时值的新的请求；记录错误日志；或者可能什么都不做，在处理失败的调用时直接返回，从而丢弃该消息。

例9-18演示了这样的实现。

#### 例9-18：实现一个 DLQ 服务

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyDLQService : IMyContract
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod(string someValue)
    {
        MsmqMessageProperty msmqProperty = OperationContext.Current.
            IncomingMessageProperties[MsmqMessageProperty.Name] as MsmqMessageProperty;
        // 如果重试超过 25 次，则丢弃该消息
        if (msmqProperty.MoveCount >= 25)
        {
            return;
        }
        // 如果超时：重试
        if (msmqProperty.DeliveryStatus == DeliveryStatus.InDoubt)
        {
            if (msmqProperty.DeliveryFailure == DeliveryFailure.ReceiveTimeout)
            {
                MyContractClient proxy = new MyContractClient();
                proxy.MyMethod(someValue);
                proxy.Close();
            }
            return;
        }
    }
}
```



```
        if(msmqProperty.DeliveryStatus == DeliveryStatus.InDoubt ||  
           msmqProperty.DeliveryFailure == DeliveryFailure.Unknown)  
        {  
            NotifyAdmin();  
        }  
    }  
    void NotifyAdmin()  
    {...}  
}
```

例 9-18 中的 DLQ 服务检查了失败的原因。如果 WCF 在传递消息时重试了 25 次以上，DLQ 将直接丢弃该消息并放弃进一步的处理。如果失败的原因是超时，DLQ 服务会创建一个指向队列服务的代理，并再次尝试调用它，传入的参数（即 DLQ 服务操作的入参）与原来的调用相同。如果该消息的状态是不确定的或发生了未知的故障，服务将通知应用程序的管理员。

## 回放失败

即使消息被成功传递，仍然可能会在服务回放的过程中失败。这种失败通常会中止回放事务，这时消息会被送回服务端的消息队列。WCF 将在队列中重新检测该消息并进行重试。如果下一个调用也失败了，消息将再次回到队列中去，如此反复。显然，我们无法接受这种不断重试的方式。如果使用队列服务主要为了提高负载，那么自动重试机制将对服务产生相当大的压力。我们需要一个智能的失败处理机制来应对调用永远不会成功（当然，“永远不会”只是一个相对的概念）的情况。该失败处理机制将决定在经过多少次重试后放弃，或经过多久的重试后放弃，甚至在重试的频率超过多少时放弃。不同的系统需要不同的重试策略，对于重试所带来的额外影响以及它的成功几率，也会具有不同的敏感程度。例如，同样重试 10 次，每小时重试一次的策略肯定不同于每隔 1min 重试一次的策略，更不同于将这 10 次重试分成两批，每隔一天连续执行 5 次重试的策略。此外，一旦我们放弃了重试，我们应该如何处理失败的消息，又该向消息的发送者返回什么样的确认标志？

## 有害消息

事务型消息系统本质上容易受到不断重复的失败的影响，因为不断的重试会将系统的性能降到最低。回放时不断失败的消息被称为有害消息（Poison Messages），因为它们使得系统不断地进行无意义的重试。事务型消息系统必须积极地检测并消除有害消息。既然我们无法获知下一次重试是否会成功，因而可以采用如下简单的启发式算法：消息失败的次数越多，再次失败的可能性也就越高。如果消息只是失败了一次，那么再重试一次就显得很合理；如果消息已经失败了 1000 次，那么它在第 1001 次失败的可能性将非常大，此时再重试就是无意义的。导致无意义（或只是浪费）的操作显然与具体的应用

程序有关，但重试的次数应该是一个可配置的决策。MsmqBindingBase 提供了许多属性，用于处理消息的回放失败：

```
public abstract class MsmqBindingBase : Binding, ...
{
    // 有害消息处理
    public int ReceiveRetryCount
    {get;set;}

    public int MaxRetryCycles
    {get;set;}

    public TimeSpan RetryCycleDelay
    {get;set;}

    public ReceiveErrorHandling ReceiveErrorHandling
    {get;set;}
    // 更多成员
}
```

## MSMQ 4.0 中对有害消息的处理

如果使用 MSMQ 4.0（只在 Windows Vista 上可用），WCF 会成批量地重试回放一条失败的消息。WCF 为每个队列终结点提供了一个重试队列和一个有害消息队列。当一批重试中的所有调用都失败之后，消息不会返回到终结点的队列中，相反，它将被送入重试队列中。一旦该消息被认为是有害的，我们就可以让 WCF 将该消息转移到有害队列中。

### 重试批处理

在一批重试处理中，WCF 会在第一次调用失败之后，立即进行 `ReceiveRetryCount` 次的重试。`ReceiveRetryCount` 默认为 5 次重试，换言之，就是包括第一次尝试在内共有 6 次尝试。在第一批重试失败之后，消息会被送到重试队列。在等待了 `RetryCycleDelay` 属性所规定的时间后，消息将从重试队列返回终结点队列，以等待下一次批重试，其中的等待延迟默认为 30min。如果这批重试再次失败，消息又将回到重试队列中，它会在那儿它等待延迟期满之后被再次重试。显然，我们不可能一直循环上述过程。`MaxRetryCycles` 属性规定了循环可以发生的最多次数。`MaxRetryCycles` 的默认值只有两个循环。在重试了 `MaxRetryCycles` 次的批重试以后，消息将被认为是一个有害消息。若要修改 `MaxRetryCycles` 的默认值，我的建议是设置与 `RetryCycleDelay` 成正比的值。因为延迟时间越长，增加的重试批处理对系统负载的影响就越小。把集中在一小段时间内的重试分散到较长的一段时间中，系统的总体压力总会得到一些缓解。如果 `RetryCycleDelay` 属性设置的时间较短，我们应该尽量减少重试批处理循环的次数，以避免出现连续的影响。

最后, `ReceiveErrorHandling` 属性负责管理在最后一次重试失败, 并且该消息被认为是有害消息之后所要采取的措施。该属性的类型为 `ReceiveErrorHandling` 枚举, 定义如下:

```
public enum ReceiveErrorHandling
{
    Fault,
    Drop,
    Reject,
    Move
}
```

### `ReceiveErrorHandling.Fault`

`Fault` 选项将有害消息看作是一个灾难性的故障, 该消息会造成 MSMQ 通道与服务宿主的错误。这样一来, 服务将不能处理其他任何消息, 不论该消息是来自一个使用队列的客户端, 还是来自一个标准的联机客户端。有害消息将遗留在终结点的队列中, 必须通过管理员或某些补偿式逻辑显式地将它移出。为了继续处理各种客户端调用, 我们必须回收宿主进程或打开一个新的宿主 (在我们从队列中删除了有害消息以后)。尽管我们可以安装一个错误处理的扩展 (如第 6 章所讨论的) 用于完成这样的工作, 但在现实中几乎无可避免地需要管理员的参与。`ReceiveErrorHandling.Fault` 是 `ReceiveErrorHandling` 属性的默认值, 使用该选项将不会有任何形式的确认信息返回给消息的发送方。

### `ReceiveErrorHandling.Drop`

顾名思义, `Drop` 选项会丢弃有害消息, 并让服务继续处理其他消息, 就像没有收到有害消息一样。如果系统对于错误及重试的忍耐程度比较高, 应该使用该选项。假如该消息不是很关键, 或者是一个可有可无的服务, 那么丢弃它并继续处理其他消息是可以接受的。此外, 丢弃消息并不意味着不需要重试, 不过从理论上讲, 不应该设置过多的重试次数, 因为如果我们非常重视这条消息, 就不应该在最后一次失败之后选择丢弃它。设置 `ReceiveErrorHandling.Drop` 选项也会向消息的发送方返回一个 ACK 消息, 因此, 从发送方的角度来看, 消息是被成功地传递并处理了。

### `ReceiveErrorHandling.Reject`

`Reject` 选项将主动拒绝有害消息, 同时拒绝对它的所有处理。与 `ReceiveErrorHandling.Drop` 类似, 使用该选项将丢弃有害消息, 不过它会返回一个 NACK 消息给发送方, 以此表明消息的传递及处理最终失败了。发送方收到 NACK 消息后, 会把原来的消息移入到自己的死信队列中。

## ReceiveErrorHandling.Move

在所有选项中，Move 选项可能是最好的，也是最具有实用价值的。它把消息移入指定的有害消息队列中，而且它不会向发送方返回一个 ACK 或 NACK 消息，最终的确认信息将在处理完有害消息队列之后返回。

### 配置示例

例 9-19 演示了宿主配置文件中的一个配置节，设置了 MSMQ 4.0 对有害消息的处理方式，而图 9-9 则以图形方式说明了以下配置在处理有害消息的表现。

例 9-19：在 MSMQ4.0 上处理有害消息

```
<bindings>
  <netMsmqBinding>
    <binding name = "PoisonMessageHandling"
      receiveRetryCount      = "2"
      retryCycleDelay        = "00:05:00"
      maxRetryCycles         = "3"
      receiveErrorHandling   = "Move"
    />
  </netMsmqBinding>
</bindings>
```

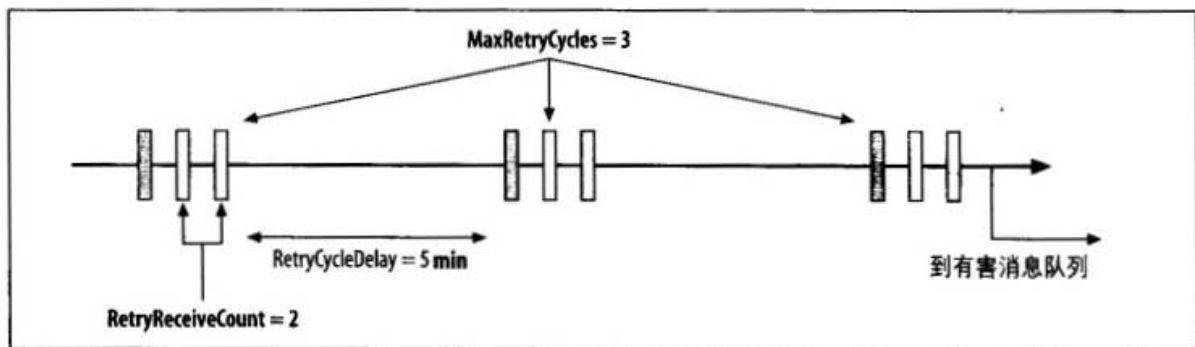


图 9-9：例 9-19 中对有害消息的处理

### 有害消息服务

当绑定被配置为 ReceiveErrorHandling.Move 选项时，服务可以提供一个专门的有害消息处理的服务，用来处理有害消息队列中的消息。有害消息服务必须是服务队列终结点契约的多态实现。WCF 从有害消息队列中取出有害消息，并将它回放给有害消息服务。因此，有害消息服务应该尽可能避免抛出未经处理的异常或中止回放事务。（最好将它配置为例 9-9 那样忽略回放事务，或使用一个像例 9-10 那样的新的事务）。这样的有害消息服务一般负责一些与失败消息有关的补偿工作，例如，如果库存没有客户订购的商品，则向客户退款。此外，一个有害消息服务还可以做许多事，包括通知管理

员,记录错误日志,或简单的返回方法以忽略该消息。有害消息服务的开发和配置与其他队列服务相同。唯一的区别是,它的终结点地址应该是原来的终结点地址加上“;poison”后缀。例9-20演示了一个服务以及它的有害消息服务的配置。在例9-20中,服务与它的有害消息服务共享同一个宿主进程,当然,这不是必选项。

例 9-20: 配置一个有害消息服务

```
<system.serviceModel>
  <services>
    <service name = "MyService">
      <endpoint
        address = "net.msmq://localhost/private/MyServiceQueue"
        binding = "netMsmqBinding"
        bindingConfiguration = "PoisonMesssageSettings"
        contract = "IMyContract"
      />
    </service>
    <service name = "MyPoisonServiceMessageHandler">
      <endpoint
        address = "net.msmq://localhost/private/MyServiceQueue;poison"
        binding = "netMsmqBinding"
        contract = "IMyContract"
      />
    </service>
  </services>
  <bindings>
    <netMsmqBinding>
      <binding name = "PoisonMesssageSettings"
        receiveRetryCount = "2"
        retryCycleDelay = "00:05:00"
        maxRetryCycles = "3"
        receiveErrorHandling = "Move"
      />
    </netMsmqBinding>
  </bindings>
</system.serviceModel>
```

## 在 MSMQ 3.0 上的有害消息处理

在 MSMQ 3.0 (在 Windows XP 和 Windows Server 2003 上可用) 中,不存在重试队列或一个专门的自动化的有害消息队列。因此,在原来的终结点队列中,WCF 最多只支持一次重试批处理。执行完一次批处理后,如果最后一次仍然失败,该消息将被认为是有害的。因此,这样的设定等同于 WCF 将 MaxRetryCycles 属性的值设置为 1,同时忽略 RetryCycleDelay 属性。ReceiveErrorHandling 属性只能使用 ReceiveErrorHandling.Fault 或 ReceiveErrorHandling.Drop 选项。使用其他选项会在服务加载时抛出一个 InvalidOperationException 异常。

**注意：**无论是 `ReceiveErrorHandling.Fault`，还是 `ReceiveErrorHandling.Drop`，都不是理想的选项。在 MSMQ 3.0 中，服务端处理回放故障（即一个由服务业务逻辑直接产生的，而不是某些通信问题造成的故障）的最佳方案是使用一个响应服务（response service），这将在后面的章节中讨论。

## 队列调用与联机调用

尽管使用相同的服务代码同时实现队列调用与联机调用，从技术上是可行的（只需要一些简单的更改，如将操作配置为单向，或为单向操作添加额外的契约），但是实际上，我们不可能使用相同的服务实现这两种方式。原因与第8章中关于异步调用的讨论一样。针对相同业务场景的同步调用和异步调用，使用了不同的工作流，而这些差异将迫使我们不得不更改服务代码以适应不同的情况。现在，队列调用的使用又为服务代码的重用（无论联机还是离线）带来了另一个障碍：服务的事务型语义发生了变化。

考虑如下的实例，在图9-10所示的应用程序中，描述了一个只使用联机调用的网上商店应用程序。

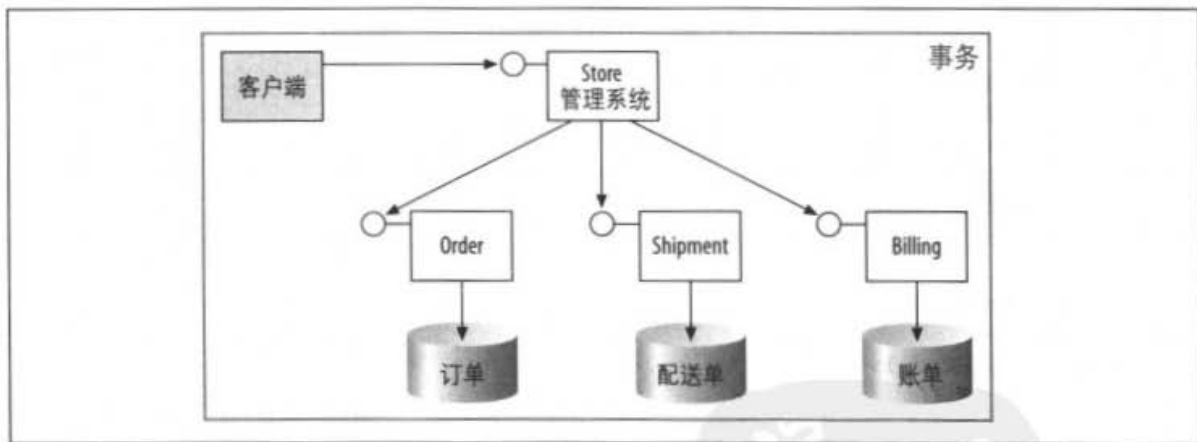


图 9-10：一个依赖单独事务的联机应用程序

Store 服务使用了三个职责分离良好的辅助服务来处理订单：Order（订货）、Shipment（配送）及 Billing（出账）。在联机调用场景中，Store 服务调用 Order 服务订货。只有当 Order 服务成功处理了该订单后（即库存中有所需的商品），Store 服务才能调用 Shipment 服务，而只有当 Shipment 服务成功执行后，Store 服务才调用 Billing 服务为客户开出账单。联机调用只需使用一个由客户端创建的事务。所有操作都被当作一个原子操作提交或中止。现在，假设 Billing 服务还暴露了一个队列终结点供 Store 服务使用，如图 9-11 所示。



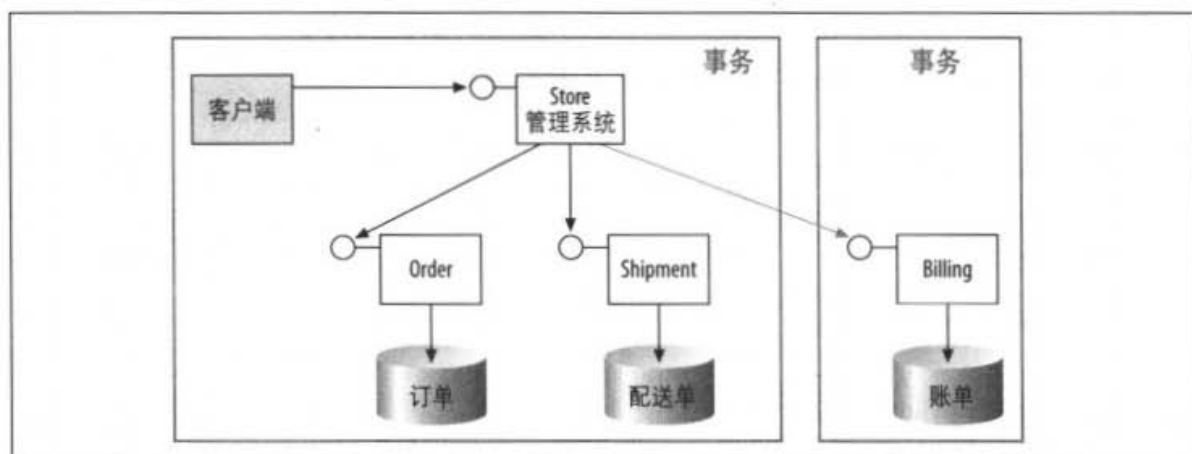


图 9-11：一个依赖多个事务的联机应用程序

Billing 服务在执行队列调用时，将使用一个与 Store 服务的其他事务不同的独立事务，该事务的提交与中止也会独立于 Order 和 Shipment 服务中的事务。这样的实现反过来就会危害到系统的一致性。因此，我们必须在 Billing 服务中包含一些逻辑来检测其他服务是否失败，并在它自己失败时启动一些相应的补偿逻辑。因此，此时的 Billing 服务与联机情况下的 Billing 服务是不相同的。

## 要求使用队列

既然并非每个服务都是联机调用和队列调用，而且一些服务还可能被设计为针对某个特定的调用，并且只适用于这种调用，因此，WCF 允许我们限定这些服务的通信模式。使用第 1 章提到的 `DeliveryRequirements` 特性，我们可以指定服务的消息传递方式。

```
public enum QueuedDeliveryRequirementsMode
{
    Allowed,
    Required,
    NotAllowed
}
[AttributeUsage(AttributeTargets.Interface|AttributeTargets.Class,
    AllowMultiple = true)]
public sealed class DeliveryRequirementsAttribute : Attribute, ...
{
    public QueuedDeliveryRequirementsMode QueuedDeliveryRequirements
    {get;set;}
    public bool RequireOrderedDelivery
    {get;set;}
    public Type TargetContract
    {get;set;}
}
```

该特性可以被用来限制一个契约（以及所有它所支持的终结点），或者一个特定的服务



类型。QueuedDeliveryRequirements 属性的默认值为 QueuedDeliveryRequirementsMode.Allowed, 因此, 以下定义都是等效的:

```
[ServiceContract]
interface IMyContract
{...}

[ServiceContract]
[DeliveryRequirements]
interface IMyContract
{...}

[ServiceContract]
[DeliveryRequirements(QueuedDeliveryRequirements =
                      QueuedDeliveryRequirementsMode.Allowed)]
interface IMyContract
{...}
```

QueuedDeliveryRequirementsMode.Allowed表示我们可以对契约或服务同时使用联机调用或队列调用。QueuedDeliveryRequirementsMode.NotAllowed显式地禁止我们使用MSMQ绑定, 此时, 所有针对终结点的调用都必须是联机调用。只有在契约或服务被显式地设计为联机方式, 才使用该值。QueuedDeliveryRequirementsMode.Required正好与之相反, 它强制要求在终结点上使用MSMQ绑定, 我们应该在契约或服务被设计为队列调用时, 才使用它。

注意, 尽管DeliveryRequirements特性提供了RequireOrderedDelivery属性(参见第1章), 但是如果使用了QueuedDeliveryRequirementsMode.Required选项, 则RequireOrderedDelivery属性必须为false, 因为队列调用本质上是无序的, 而且消息可以以任何顺序进行回放。

当DeliveryRequirements特性被应用于一个接口时, 它会影响到所有在终结点中公开该契约的服务:

```
[ServiceContract]
[DeliveryRequirements(QueuedDeliveryRequirements =
                      QueuedDeliveryRequirementsMode.Required)]
interface IMyQueuedContract
{...}
```

客户端同样可以在服务契约的副本上应用DeliveryRequirements特性。

当DeliveryRequirements特性被应用到服务类时, 它会影响到该服务的所有终结点:

```
[DeliveryRequirements(QueuedDeliveryRequirements =
                      QueuedDeliveryRequirementsMode.Required)]
class MyQueuedService : IMyQueuedContract, IMyOtherContract
{...}
```

如果设定了TargetContract属性的DeliveryRequirements特性被应用到服务类上, 则会影响服务中所有公开指定契约的终结点:

```
[DeliveryRequirements(TargetContract = typeof(IMyQueuedContract),  
    QueuedDeliveryRequirements =  
        QueuedDeliveryRequirementsMode.Required)]  
class MyService : IMyQueuedContract, IMyOtherContract  
{...}
```

## 响应服务

迄今描述的队列调用的编程模型都是单边的: 客户端向某个队列发送一个单向消息, 而服务对该消息进行处理。当队列操作被设计为单向调用时, 这样的模型足以应付了。然而, 根据调用后的结果, 队列服务可能需要向它的客户端返回一些信息, 比如返回处理结果甚至一些错误信息。但是在默认情况下这是不可能的。WCF把队列调用与单向调用等同起来, 而单向调用本质上是禁止这样的响应的。此外, 队列服务(以及它们的客户端)可能是断开的。如果客户端向某个断开的服务发送一个队列调用, 当服务最终获得这些消息并处理它们时, 可能已经无法向客户端返回信息了, 因为客户端可能已不存在。解决方案是让服务将消息返回给一个客户端提供的队列服务。我把这样一个服务称为响应服务(注1)。图9-12展示了这个解决方案的体系架构。

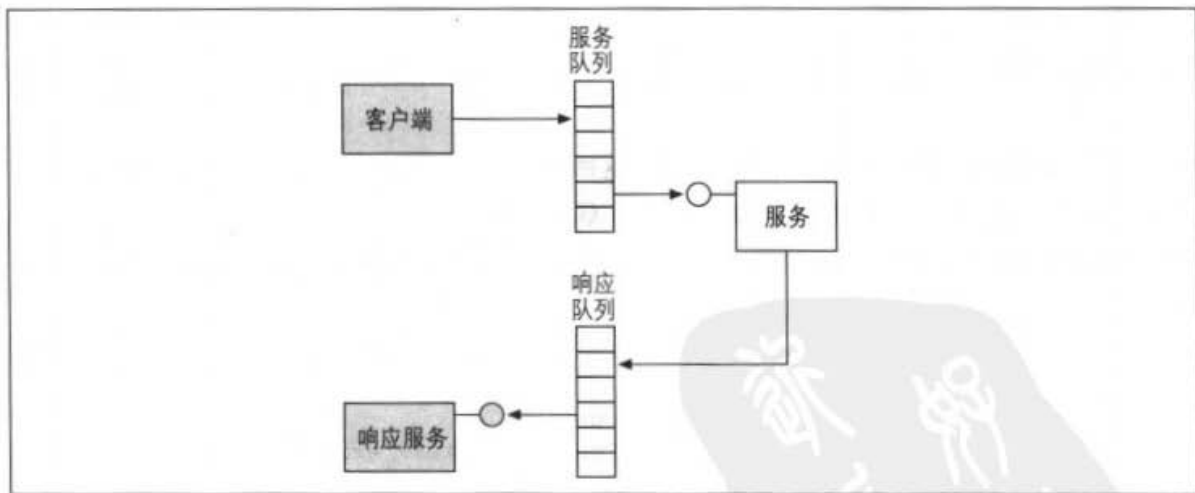


图 9-12: 一个响应服务

响应服务只是系统中的另一个队列服务。响应服务同样可以断开客户端, 并托管到一个单独的进程或一台单独的机器上, 或者它也可以共享客户端进程。如果响应服务共享客户端的进程, 那么在客户端启动后, 响应程序就开始处理放入队列中的响应消息。把响

注1: 作者在MSDN杂志2007年2月刊中首次提出了响应服务技术。

应用程序放在一个与客户端进程分开的进程（甚至机器）中，有助于对响应服务与客户端或使用它的客户端进一步解耦。

**注意：**并非所有的队列服务都需要响应服务，正如我们将要看到的，响应服务增加了系统的复杂性。务实的做法是只在合适的地方使用响应服务，也就是说，把它用在能提供最大价值的地方。

## 定义一个响应服务契约

与所有 WCF 服务的用法相同，客户端与服务需要预先就响应契约以及它的用途达成共识，例如我们是只返回值还是在返回值时还包括错误信息。注意，我们也可以将响应服务分为两个服务，并让其中一个响应服务返回结果，另一个返回错误信息。例如，考虑下面这个由队列服务 `MyCalculator` 实现的 `ICalculator` 契约：

```
[ServiceContract]
interface ICalculator
{
    [OperationContract(IsOneWay = true)]
    void Add(int number1, int number2);
    // 更多操作
}
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculator : ICalculator
{...}
```

`MyCalculator` 服务被要求向它的客户端返回计算的结果，以及可能出现的任何错误。计算结果是一个整数，而错误则表现为第 6 章提到的 `ExceptionDetail` 类型的数据契约。对于响应服务，`ICalculatorResponse` 契约可以被定义如下：

```
[ServiceContract]
interface ICalculatorResponse
{
    [OperationContract(IsOneWay = true)]
    void OnAddCompleted(int result, ExceptionDetail error);
}
```

支持 `ICalculatorResponse` 的响应服务需要检查返回的错误信息，在方法完成时通知客户端应用程序、用户或应用程序管理员，并将计算结果提供给对它感兴趣的实体。例 9-21 展示了一个支持 `ICalculatorResponse` 的简单的响应服务。

### 例 9-21：一个简单的响应服务

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculatorResponse : ICalculatorResponse
```

```
{
    public void OnAddCompleted(int result, ExceptionDetail error)
    {
        MessageBox.Show("result = " + result, "MyCalculatorResponse");

        if(error != null)
        {
            // 处理错误
        }
    }
}
```

## 响应地址与方法 ID

MyCalculator和MyCalculatorResponse两者的实现都存在两个明显的问题。第一个问题在于同样的响应服务可用于处理对多个队列服务的多个调用的响应（或完成），但是，正如在例9-21中列举的那样，MyCalculatorResponse（以及更重要的是它所服务的客户端）无法对各个调用的响应进行区分。为了解决这一问题，我们让发出最初队列调用的客户端向该调用分配某个唯一（至少在客户端应用程序中是唯一的）的 ID 作为标记。队列服务 MyCalculator 需要将该 ID 传递给 MyCalculatorResponse，使其能够应用与该 ID 相关的某种自定义逻辑。第二个问题在于队列服务如何发现响应服务的地址。与双向回调不同，WCF 并不直接支持向队列服务传递响应服务的引用。因此，队列服务需要构建一个响应服务的代理，再调用响应服务的契约的操作。由于响应服务的契约是在设计时就被决定的，又总是使用 NetMsmqBinding 绑定（或根据配置形成的变种），队列服务无法获得响应服务的地址，因而无法响应。尽管我们可以将该地址放置在服务宿主的配置文件中（位于一个客户端配置节中），但这并非明智之举。因为相同的队列服务可能会被多个客户端调用，而每个客户端都有其自身专用的响应服务及其地址。一种可能的解决方案是将客户端管理的 ID 和服务所需的响应服务地址作为参数，显式地传递给队列服务契约上的每个操作：

```
[ServiceContract]
interface ICalculator
{
    [OperationContract(IsOneWay = true)]
    void Add(int number1, int number2, string responseAddress, string methodId);
}
```

与之类似，队列服务也可以将方法 ID 作为一个参数，显式地传递给在响应服务契约上的每个操作：

```
[ServiceContract]
interface ICalculatorResponse
{
    [OperationContract(IsOneWay = true)]
    void OnAddCompleted(int result, ExceptionDetail error, string methodId);
}
```

## 使用消息标头

尽管将地址和ID作为参数显式地传递能够解决以上问题,但这种方案改变了原先的契约,并且在同一个操作中同时引入了系统层面与业务层面的参数。因此,更好的解决方案是让客户端将响应地址和操作ID存储在调用的输出消息标头中。使用消息标头的方式是我们向服务传递带外信息时经常采取的一种技术,采用这种技术,将不会使带外信息出现在服务契约中。

操作上下文提供了传入和传出消息标头的集合,这些集合可以通过 `IncomingMessageHeaders` 和 `OutgoingMessageHeaders` 属性获取:

```
public sealed class OperationContext : ...
{
    public MessageHeaders IncomingMessageHeaders
    {get;}
    public MessageHeaders OutgoingMessageHeaders
    {get;}
    // 更多成员
}
```

每个集合均为 `MessageHeaders` 类型,代表 `MessageHeader` 对象的集合:

```
public sealed class MessageHeaders : IEnumerable<...>,...
{
    public void Add(MessageHeader header);
    public T GetHeader<T>(int index);
    public T GetHeader<T>(string name,string ns);
    // 更多成员
}
```

应用程序开发者不适合直接使用 `MessageHeader` 类,相反,应该使用 `MessageHeader<T>` 类,它提供了类型安全性,并能轻松地将 CLR 类型转换为一条消息标头:

```
public abstract class MessageHeader : ...
{...}

public class MessageHeader<T>
{
    public MessageHeader();
    public MessageHeader(T content);
    public T Content
    {get;set;}
    public MessageHeader GetUntypedHeader(string name,string ns);
    // 更多成员
}
```

我们可以使用任何可序列化类型或数据契约类型作为 `MessageHeader<T>` 的类型参数。根据 CLR 类型构造相应的 `MessageHeader<T>`,然后使用 `GetUntypedHeader()` 方法将其转换为 `MessageHeader`,并存储在输出消息标头中。`GetUntypedHeader()` 要求

开发者提供通用类型的参数名称与命名空间。名称与命名空间将用于从消息标头集合中查找标头。可以通过 `MessageHeaders` 类的 `GetHeader<T>()` 方法执行查询操作。调用 `GetHeader<T>()` 方法可获取使用的 `MessageHeader<T>` 类型参数的值。

## ResponseContext 类

由于客户端需要在消息标头中同时传递地址和方法ID,因此一个基本类型参数无法满足这样的要求。作为替代,我编写了 `ResponseContext` 类,它的定义如例 9-22 所示。

例 9-22: `ResponseContext` 类

```
namespace ServiceModelEx
{
    [DataContract]
    public class ResponseContext
    {
        [DataMember]
        public readonly string ResponseAddress;

        [DataMember]
        public readonly string FaultAddress;

        [DataMember]
        public readonly string MethodId;

        public ResponseContext(string responseAddress, string methodId) :
            this(responseAddress, methodId, null)
        {}

        public ResponseContext(ResponseContext responseContext) :
            this(responseContext.ResponseAddress, responseContext.MethodId,
                responseContext.FaultAddress)
        {}

        public ResponseContext(string responseAddress) : this(responseAddress,
            Guid.NewGuid().ToString())
        {}

        public ResponseContext(string responseAddress, string methodId,
            string faultAddress)
        {
            ResponseAddress = responseAddress;
            MethodId = methodId;
            FaultAddress = faultAddress;
        }
        // 更多成员
    }
}
```

`ResponseContext` 类提供了一个位置用以存储响应地址和ID。此外,如果客户端要使用单独的错误响应服务, `ResponseContext` 类还提供了一个字段用以存储错误响应服务的地址。本章并没有用到这一特性。客户端负责构造具有唯一ID的 `ResponseContext`



实例。客户端既可以提供该ID作为一个构造参数,也可以使用 `ResponseContext` 的另一个构造函数,它能够接收响应地址作为参数,并让构造函数生成一个GUID作为ID。

## 客户端编程

客户端可以为每次方法调用提供一个ID,甚至在处理一个会话队列服务时,通过为每个调用使用不同的 `ResponseContext` 实例,为每次方法调用提供ID。客户端需要将 `ResponseContext` 实例保存在输出消息标头中。此外,客户端还必须在新的操作上下文中执行该操作,而不能使用现有的操作上下文。这一要求对于服务和非服务型客户端同样适用。WCF允许客户端使用如下定义的 `OperationContextScope` 类,在当前线程中采用新的操作上下文:

```
public sealed class OperationContextScope : IDisposable
{
    public OperationContextScope(IContextChannel channel);
    public OperationContextScope(OperationContext context);
    public void Dispose();
}
```

当现有上下文不适用时,可以使用 `OperationContextScope` 技术切换为新的上下文。`OperationContextScope` 的构造函数可以将当前线程的操作上下文替换为新的上下文。在 `OperationContextScope` 实例上调用 `Dispose()` 方法可以恢复原来的上下文(即使它为 `null`)。如果不调用 `Dispose()`,可能会破坏同一线程中需要使用先前上下文的其他对象。因此, `OperationContextScope` 可以用在 `using` 语句中,只为一个代码域提供新的操作上下文,即使面对异常:

```
using(OperationContextScope scope = new OperationContextScope(...))
{
    // 使用新的上下文进行工作
    ...
} // 这里存储之前的上下文
```

构造新的 `OperationContextScope` 实例时,需要为构造函数提供调用所用代理的内部通道。客户端需要使用响应地址及ID创建一个新的 `OperationContextScope`,将该 `ResponseContext` 添加到一个消息标头中,并将它添加到新的上下文的传出消息标头的集合中,然后调用代理。例9-23演示了这些步骤。

### 例9-23: 针对响应服务的客户端编程

```
string methodId = GenerateMethodId();
string responseQueue = "net.msmq://localhost/private/MyCalculatorResponseQueue";
ResponseContext responseContext = new ResponseContext(responseQueue, methodId);

MessageHeader<ResponseContext> responseHeader =
    new MessageHeader<ResponseContext>(responseContext);
```



```
CalculatorClient proxy = new CalculatorClient();
using(OperationContextScope contextScope =
    new OperationContextScope(proxy.InnerChannel))
{
    OperationContext.Current.OutgoingMessageHeaders.Add(
        responseHeader.GetUntypedHeader("ResponseContext", "ServiceModelEx"));
    proxy.Add(2, 3);
}
proxy.Close();

// 辅助方法
string GenerateMethodId()
{...}
```

在例9-23中，客户端使用辅助方法 `GenerateMethodId()` 生成一个独一无二的ID。然后，它使用响应地址及ID作为构造参数创建一个新的 `ResponseContext` 实例。客户端使用了 `MessageHeader<ResponseContext>`，将 `ResponseContext` 实例包装在一个消息标头中。客户端构建了一个队列服务的新代理，并将它的内部通道当作一个构造参数以建立一个新的 `OperationContextScope`。在 `using` 语句范围域中自动恢复到之前的操作上下文。在新的上下文范围中，客户端访问它的新的操作上下文，并从中获得 `OutgoingMessageHeaders` 集合。客户端使用响应上下文的名称及命名空间作为参数，调用强类型消息标头中的 `GetUntypedHeader()` 方法，将它转换为一个原始的 `MessageHeader`，并将该消息标头添加到传出消息标头的集合中。客户端调用代理，然后释放操作上下文的作用域，并关闭代理。现在，队列消息包含带外响应地址及方法ID。在例9-23中，我们可以使用一个通道工厂来代替代理类，此时我们可以将工厂返回的代理转换为一个上下文通道，从而创建出一个操作上下文作用域：

```
ChannelFactory<ICalculator> factory = new ChannelFactory<ICalculator>("");
ICalculator proxy = factory.CreateChannel();
using(OperationContextScope contextScope =
    new OperationContextScope(proxy as IContextChannel))
{...}
```

## 服务端编程

队列服务通过 `ResponseContext.Current` 访问它的传入消息标头，并从中读取响应地址和方法ID。服务需要使用该地址构造响应服务的代理，并将方法ID回传给响应服务。服务通常不会直接使用该ID。服务可以使用与客户端相同的技术将方法ID传递给响应服务；也就是说，不使用显式参数，而是用传出消息标头在带外将ID传递给响应服务。与客户端一样，服务也必须通过 `OperationContextScope` 使用新的操作上下文，以便能够修改传出消息标头的集合。服务能够以编程方式构造响应服务的代理，并将响应地址和 `NetMsmqBinding` 实例提供给该代理。服务甚至还可以从配置文件中读取绑定设置。具体步骤如例9-24所示。

例 9-24: 响应服务的服务端编程

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculator : ICalculator
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void Add(int number1, int number2)
    {
        int result = 0;
        ExceptionDetail error = null;
        try
        {
            result = number1 + number2;
        }
        // 不要重新抛出异常
        catch(Exception exception)
        {
            error = new ExceptionDetail(exception);
        }
        finally
        {
            ResponseContext responseContext =
                OperationContext.Current.IncomingMessageHeaders.
                    GetHeader<ResponseContext>("ResponseContext", "ServiceModelEx");

            EndpointAddress responseAddress =
                new EndpointAddress(responseContext.ResponseAddress);

            MessageHeader<ResponseContext> responseHeader =
                new MessageHeader<ResponseContext>(responseContext);

            NetMsmqBinding binding = new NetMsmqBinding();
            CalculatorResponseClient proxy =
                new CalculatorResponseClient(binding, responseAddress);

            using(OperationContextScope scope =
                new OperationContextScope(proxy.InnerChannel))
            {
                OperationContext.Current.OutgoingMessageHeaders.Add(
                    responseHeader.GetUntypedHeader("ResponseContext", "ServiceModelEx"));

                proxy.OnAddCompleted(result, error);
            }
            proxy.Close();
        }
    }
}
```

在例9-24中, 服务会捕获由业务逻辑操作引发的所有异常, 并使用 `ExceptionDetail` 对象包装各个异常。服务不会重新抛出异常。正如我们会在后面看到的那样, 在事务和响应服务的上下文中, 重新抛出异常会取消响应。而且, 使用响应服务返回出错消息远比依赖于 WCF 的回放错误处理机制好的多。

在 `finally` 语句中, 无论是否出现异常, 服务都会作出响应。它使用来自响应上下文中的地址和一个 `NetMsmqBinding` 的新实例构造响应服务的代理。服务访问它自己的操作上下文中的传入消息标头集合, 并使用 `GetHeader<T>()` 方法从中提取出客户端提供的响应上下文。注意, 在这里我们使用命名空间加名称的方法从传入消息标头的集合中查询响应上下文。然后, 服务使用代理的内部通道生成了一个新的 `OperationContextScope`。在新的作用域中, 服务会将原来的响应上下文 (尽管可能只是加入 ID) 添加到新的上下文的传出消息标头中, 然后调用响应服务代理, 实际上就是将响应消息加入队列中。之后, 服务会释放上下文作用域并关闭代理。如果响应服务已经使用了错误响应的地址, 那么最好将整个响应上下文 (而不仅仅是 ID) 发送给响应服务。

## 响应服务端编程

响应服务会访问它的传入消息标头集合, 从中读取方法 ID 并作出相应的响应。例 9-25 演示了这样一个响应服务可能的实现方式。

例 9-25: 实现一个响应服务

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculatorResponse : ICalculatorResponse
{
    public static event GenericEventHandler<string,int> AddCompleted = delegate{};
    public static event GenericEventHandler<string> AddError = delegate{};

    [OperationBehavior(TransactionScopeRequired = true)]
    public void OnAddCompleted(int result, ExceptionDetail error)
    {
        ResponseContext responseContext =
            OperationContext.Current.IncomingMessageHeaders.
                GetHeader<ResponseContext>("ResponseContext", "ServiceModelEx");

        string methodId = responseContext.MethodId;

        if(error == null)
        {
            AddCompleted(methodId, result);
        }
        else
        {
            AddError(methodId);
        }
    }
}
```

例 9-25 中的响应服务定义了两个公共的静态事件, 一个用于完成通知, 而另一个用于错误通知。响应服务调用这些委托, 通知完成事件与出错事件的订阅者。响应服务会像队列服务那样访问操作上下文中的传入消息标头集合, 并从中提取出响应服务的上下文以

及其中包含的方法ID。然后,响应服务将调用完成委托,提供操作的结果及它的ID。如果队列服务出现异常,响应服务会调用错误委托,此时只需传入方法ID。

---

**注意:** 一个队列响应服务不只限于配合队列服务使用。我们可以运用相同的技术向一个联机服务传递地址及方法ID,并让该服务将响应消息发送到一个由客户端提供的队列中。

---

## 简化响应服务

尽管目前实现及使用一个响应服务的技术是可行的,但它仍然存在很大的问题。在例9-23、例9-24和例9-25中,那些原本应该体现业务逻辑的服务端与客户端代码被底层的WCF编码所淹没。幸运的是,该技术能够被封装到辅助类中,从而简化响应服务,这样,每个参与方就能够方便地进行交互,而无需将底层操作暴露在外。

第一步是向ResponseContext添加Current静态属性,封装交互过程中的原始消息标头:

```
[DataContract]
public class ResponseContext
{
    public static ResponseContext Current
    {get;set;}
    // 其余成员
}
```

例9-26列出了Current属性。

例9-26: ResponseContext.Current 属性

```
[DataContract]
public class ResponseContext
{
    public static ResponseContext Current
    {
        get
        {
            OperationContext context = OperationContext.Current;
            if(context == null)
            {
                return null;
            }
            try
            {
                return context.IncomingMessageHeaders.
                    GetHeader<ResponseContext>("ResponseContext", "ServiceModelEx");
            }
            catch
            {
            }
        }
    }
}
```

```
{
    return null;
}
}
set
{
    OperationContext context = OperationContext.Current;
    Debug.Assert(context != null);
    // 具有多个 ResponseContext 标头就是一个错误
    bool headerExists = false;
    try
    {
        context.OutgoingMessageHeaders.GetHeader<ResponseContext>
            ("ResponseContext", "ServiceModelEx");

        headerExists = true;
    }
    catch (MessageHeaderException exception)
    {
        Debug.Assert(exception.Message == "There is not a header with name
            ResponseContext and namespace
            ServiceModelEx in the message.");
    }
    if (headerExists)
    {
        throw new InvalidOperationException("A header with name ResponseContext
            and namespace ServiceModelEx
            already exists in the message.");
    }
    MessageHeader<ResponseContext> responseHeader
        = new MessageHeader<ResponseContext>(value);
    context.OutgoingMessageHeaders.Add
        (responseHeader.GetUntypedHeader("ResponseContext", "ServiceModelEx"));
}
}
// 其余代码与例 9-22 相同
}
```

如果传入消息标头中不存在响应上下文，则 `Current` 属性的 `get` 访问器将返回 `null`。此时，队列服务会检验是否存在一个响应对象：

```
if (ResponseContext.Current != null)
{
    // 在此处执行响应的内容
}
```

注意，`Current` 属性的 `set` 访问器在设值前，会检验当前操作上下文的传出消息标头集合中是否存在其他的 `ResponseContext`。

### 简化客户端

若要简化客户端，实现上述步骤地自动化，需要使用一个代理基类将例 9-23 所示的配置

响应服务的步骤封装起来。不同于双向回调，WCF并没有直接提供这么一个代理类，因此我们需要手工定制。为了减轻工作量，我提供了 `ResponseClientBase<T>` 类，定义如下：

```
public abstract class ResponseClientBase<T> : ClientBase<T> where T : class
{
    public readonly string ResponseAddress;

    public ResponseClientBase(string responseAddress);
    public ResponseClientBase(string responseAddress, string endpointName);
    public ResponseClientBase(string responseAddress,
                               NetMsmqBinding binding, EndpointAddress remoteAddress);
    /* 更多的构造函数 */
    protected string Enqueue(string operation, params object[] args);
    protected virtual string GenerateMethodId();
}
```

要使用 `ResponseClientBase<T>` 类，需要从它派生出一个具体类，并将队列服务的契约类型作为类型参数。不同于普通代理，这个具体类不用继承契约。相反，它提供了一套与契约相似的方法，只不过这些方法的返回值都是一个代表方法ID的字符串，而不是契约中的 `void` 类型（之所以不能继承契约，是因为契约的操作都是单向操作，没有返回值）。例如，使用如下的队列服务契约：

```
[ServiceContract]
interface ICalculator
{
    [OperationContract(IsOneWay = true)]
    void Add(int number1, int number2);
    // 更多操作
}
```

例9-27 演示了与之对应的支持响应服务的代理。

#### 例9-27：派生于 `ResponseClientBase<T>`

```
class CalculatorClient : ResponseClientBase<ICalculator>
{
    public CalculatorClient(string responseAddress) : base(responseAddress)
    {}
    public CalculatorClient(string responseAddress, string endpointName) :
        base(responseAddress, endpointName)
    {}
    public CalculatorClient(string responseAddress,
                           NetMsmqBinding binding, EndpointAddress remoteAddress),
        : base(responseAddress, binding, remoteAddress)
    {}
    /* 更多构造函数 */

    public string Add(int number1, int number2)
    {
        return Enqueue("Add", number1, number2);
    }
}
```

```

    }
}

```

使用 `ResponseClientBase<T>`，例 9-23 可以简化为：

```

string responseAddress = "net.msmq://localhost/private/MyCalculatorResponseQueue";

CalculatorClient proxy = new CalculatorClient(responseAddress);
string methodId = proxy.Add(2,3);
proxy.Close();

```

`ResponseClientBase<T>` 类的虚方法 `GenerateMethodId()` 使用一个 GUID 作为方法 ID。`ResponseClientBase<T>` 的子类可以重写它，并提供另外的唯一标识的字符串，例如一个递增的静态整数：

```

class CalculatorClient : ResponseClientBase<ICalculator>
{
    static int m_MethodId = 123;
    protected override string GenerateMethodId()
    {
        lock(typeof(CalculatorClient))
        {
            int id = ++m_MethodId;
            return id.ToString();
        }
    }
    // 其余实现
}

```

例 9-28 演示了 `ResponseClientBase<T>` 类的实现。

#### 例 9-28：实现 `ResponseClientBase<T>` 类

```

public class ResponseClientBase<T> : ClientBase<T> where T : class
{
    public readonly string ResponseAddress;

    public ResponseClientBase(string responseAddress)
    {
        ResponseAddress = responseAddress;
        QueuedServiceHelper.VerifyQueue(Endpoint);
        Debug.Assert(Endpoint.Binding is NetMsmqBinding);
    }

    public ResponseClientBase(string responseAddress, string endpointName)
        : base(endpointName)
    {
        ResponseAddress = responseAddress;
        QueuedServiceHelper.VerifyQueue(Endpoint);
        Debug.Assert(Endpoint.Binding is NetMsmqBinding);
    }

    public ResponseClientBase(string responseAddress,
        NetMsmqBinding binding, EndpointAddress remoteAddress)
        : base(binding, remoteAddress)
    {
        ResponseAddress = responseAddress;
        QueuedServiceHelper.VerifyQueue(Endpoint);
        Debug.Assert(Endpoint.Binding is NetMsmqBinding);
    }
}

```



```

    {
        ResponseAddress = responseAddress;
        QueuedServiceHelper.VerifyQueue(Endpoint);
    }
    protected string Enqueue(string operation, params object[] args)
    {
        using(OperationContextScope contextScope =
            new OperationContextScope(InnerChannel))
        {
            string methodId = GenerateMethodId();
            ResponseContext responseContext =
                new ResponseContext(ResponseAddress, methodId);
            ResponseContext.Current = responseContext;

            Type contract = typeof(T);
            // 不支持契约层级或者重载
            MethodInfo methodInfo = contract.GetMethod(operation);
            methodInfo.Invoke(Channel, args);

            return responseContext.MethodId;
        }
    }
    protected virtual string GenerateMethodId()
    {
        return Guid.NewGuid().ToString();
    }
}

```

ResponseClientBase<T> 类的构造函数接收响应地址, 以及诸如终结点名称、地址和绑定之类的标准代理参数作为构造参数。构造函数将响应地址保存在一个只读的公共字段中。ResponseClientBase<T> 派生自一个标准的 ClientBase<T> 类, 因此所有的构造函数都会将工作委派给它们各自的基类构造函数。此外, 构造函数会使用 QueuedServiceHelper.VerifyQueue() 方法验证队列 (以及 DLQ) 是否存在, 并在需要的情况下创建它。构造函数还将验证用于服务终结点的绑定是否为 NetMsmqBinding 绑定, 因为该代理只能配合队列调用一起使用。ResponseClientBase<T> 类的核心是 Enqueue() 方法。Enqueue() 方法接收一个需要调用的操作名 (事实上是将消息放入队列) 以及操作的参数。Enqueue() 方法创建了一个新的操作上下文作用域, 生成了一个新的方法 ID, 并把该 ID 和响应地址保存在一个 ResponseContext 中。接着, 通过设置 ResponseContext.Current 属性, 将 ResponseContext 分配给传出消息的标头。然后, Enqueue() 方法使用反射去调用提供的操作名。由于使用了反射以及延迟绑定, ResponseClientBase<T> 无法支持契约层级或重载操作。针对这两种情况, 我们需要手工编码, 如例 9-23 所示。

---

**注意:** 如果我们希望联机服务使用一个队列响应服务 (不会使用 NetMsmqBinding), 就需要重写 ResponseClientBase<T> 类, 在类的定义中只使用 Binding 类型, 并将 Enqueue() 方法重命名为 Invoke(), 同时还要避免对队列调用的使用下断言。

---

## 简化队列服务

为了简化服务从消息标头集合以及响应中提取响应参数的工作,并实现这一步骤的自动化,我创建了 `ResponseScope<T>` 类:

```
public class ResponseScope<T> : IDisposable where T : class
{
    public readonly T Response;
    public ResponseScope();
    public ResponseScope(string bindingConfiguration);
    public ResponseScope(NetMsmqBinding binding);

    public void Dispose();
}
```

`ResponseScope<T>` 是一个可销毁的对象。它可以装载一个新的操作上下文,而当它被销毁时又会将当前的操作上下文恢复为旧的操作上下文。为了在发生异常时能够自动完成以上工作,我们应该在一个 `using` 语句中使用 `ResponseScope<T>`。`ResponseScope<T>` 以响应契约的类型作为类型参数,并且提供了相同类型的公有只读字段 `Response`。`Response` 是一个响应服务的代理,且 `ResponseScope<T>` 的使用者会通过 `Response` 来调用响应服务上的操作。我们不需要销毁 `Response`,因为 `ResponseScope<T>` 会在 `Dispose()` 方法中自动将它销毁。`ResponseScope<T>` 会根据配置文件中默认的终结点,或者使用绑定信息(配置节名称或实际的绑定实例)提供给构造函数对 `Response` 进行初始化。使用 `ResponseScope<T>`,例 9-24 中的 `finally` 语句可以被简化为:

```
finally
{
    using(ResponseScope<ICalculatorResponse> scope
        = new ResponseScope<ICalculatorResponse>())
    {
        scope.Response.OnAddCompleted(result,error);
    }
}
```

例 9-29 演示了 `ResponseScope<T>` 类的实现。

### 例 9-29: 实现 `ResponseScope<T>`

```
public class ResponseScope<T> : IDisposable where T : class
{
    OperationContextScope m_Scope;

    public readonly T Response;

    public ResponseScope() : this(new NetMsmqBinding())
    {}
    public ResponseScope(string bindingConfiguration) :
        this(new NetMsmqBinding(bindingConfiguration))
    {}
}
```

```

public ResponseScope(NetMsmqBinding binding)
{
    ResponseContext responseContext = ResponseContext.Current;

    EndpointAddress address
        = new EndpointAddress(responseContext.ResponseAddress);

    ChannelFactory<T> factory = new ChannelFactory<T>(binding, address);
    QueuedServiceHelper.VerifyQueue(factory.Endpoint);
    Response = factory.CreateChannel();

    // 现在转换上下文
    m_Scope = new OperationContextScope(Response as IContextChannel);

    ResponseContext.Current = responseContext;
}
public void Dispose()
{
    IDisposable disposable = Response as IDisposable;
    disposable.Dispose();
    m_Scope.Dispose();
}
}

```

`ResponseScope<T>`的构造函数使用`ResponseContext.Current`提取输入的响应上下文。然后,它使用一个通道工厂验证响应队列是否存在,并创建一个响应服务的代理对`Response`进行初始化。实现`ResponseScope<T>`的技巧是不在`using`语句中使用`OperationContextScope`。通过构造一个新的`OperationContextScope`,`ResponseScope<T>`建立了一个新的操作上下文。`ResponseScope<T>`封装了一个`OperationContextScope`对象,通过它保存新建的`OperationContextScope`。一旦`ResponseScope<T>`在`using`语句中被销毁,旧的上下文就将被恢复。之后,`ResponseScope<T>`使用`ResponseContext.Current`向新的操作上下文的传出消息标头集合添加响应上下文,然后返回。

## 简化服务端响应

响应服务需要从传入消息标头集合中提取方法ID。我们只需要简单地使用`ResponseContext.Current`就可以自动完成上述工作。一旦采用这种方法,例9-25中的`OnAddComplete()`方法就可以被简化为:

```

public void OnAddCompleted(int result, ExceptionDetail error)
{
    string methodId = ResponseContext.Current.MethodId;

    if(error == null)
    {
        AddCompleted(methodId, result);
    }
    else

```

```
{  
    AddError(methodId);  
}  
}
```

## 事务

队列服务通常会将响应消息作为传入回放事务的一部分放入响应队列中。根据例9-30定义的队列服务，图 9-13 描述了最终的事务范围以及参与的动作。

例 9-30：将响应消息作为回放事务的一部分放入队列

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]  
class MyCalculator : ICalculator  
{  
    [OperationBehavior(TransactionScopeRequired = true)]  
    public void Add(int number1, int number2)  
    {  
        ...  
        try  
        {  
            ...  
        }  
        catch// 不要重抛异常  
        {  
            ...  
        }  
        finally  
        {  
            using(ResponseScope<ICalculatorResponse> scope  
                    = new ResponseScope<ICalculatorResponse>())  
            {  
                scope.Response.OnAddCompleted(...);  
            }  
        }  
    }  
}
```

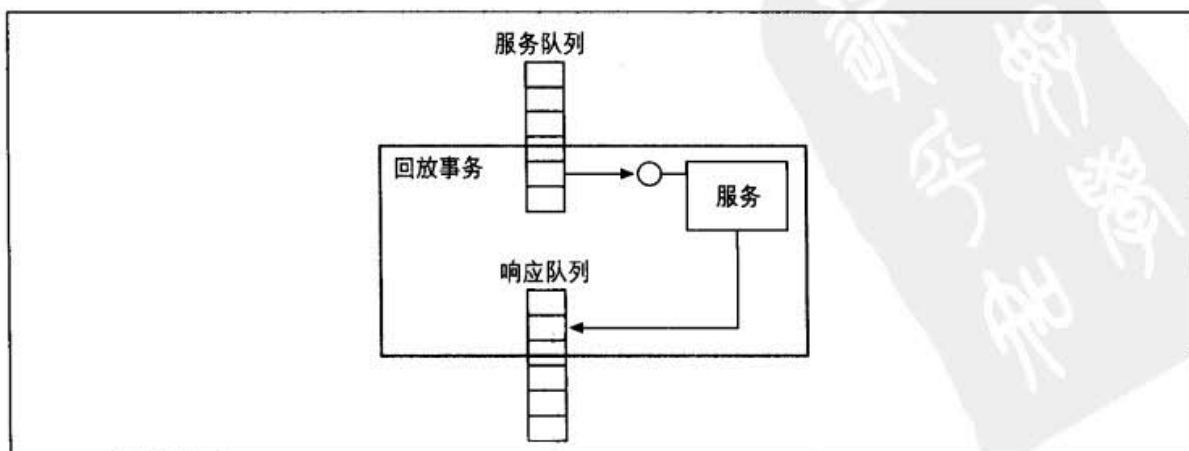


图 9-13：在回放事务中将响应消息放入队列

将队列调用回放以及队列响应放在同一个事务中的设计，其优势在于不管什么原因，只要中止了回放事务（即使归咎于事务中的其他服务），服务的响应都会被自动取消。这也是大多数应用程序常见的选择。注意，在例9-30中，服务捕获了所有异常，但并不会将它们重新抛出。这一点很重要，因为任何未被处理的异常（或重新抛出的异常）都将中止响应，这会使得服务之前所做的响应失去意义。使用一个响应服务在本质上意味着服务不会依赖于WCF的自动重试机制，而且服务会对自身业务逻辑上的错误做出相应的处理，因为客户端期望服务能以一种预先规定好的方式进行响应。

## 使用一个新的事务

除了将响应作为回放事务的一部分，服务也可以在一个新的事务中进行响应，方法是将响应包含在一个新的事务范围中，如例9-31与图9-14所示。

例9-31：在一个新的事务中进行响应

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculator : ICalculator
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void Add(int number1, int number2)
    {
        ...
        finally
        {
            using (TransactionScope transactionScope =
                new TransactionScope(TransactionScopeOption.RequiresNew))
            using (ResponseScope<ICalculatorResponse> responseScope
                = new ResponseScope<ICalculatorResponse>())
            {
                scope.Response.OnAddCompleted(...);
            }
        }
    }
}
```

在一个新的事务中，有两种情形需要进行响应。第一种情形是，无论回放事务的结果怎样，即便它有可能被其他下游服务中止，服务都要做出响应。第二种情形是，响应可有可无，这时服务不希望由于响应的中止而导致回放事务回滚。

## 响应服务与事务

由于响应服务只是另一种形式的队列服务，因此管理它的方式以及它参与到一个事务中的方式，均与其他队列服务十分相似。但是，在它所处的特定背景下，仍有一些要点值得提出。响应服务可以在传入消息的回放事务中处理响应消息：

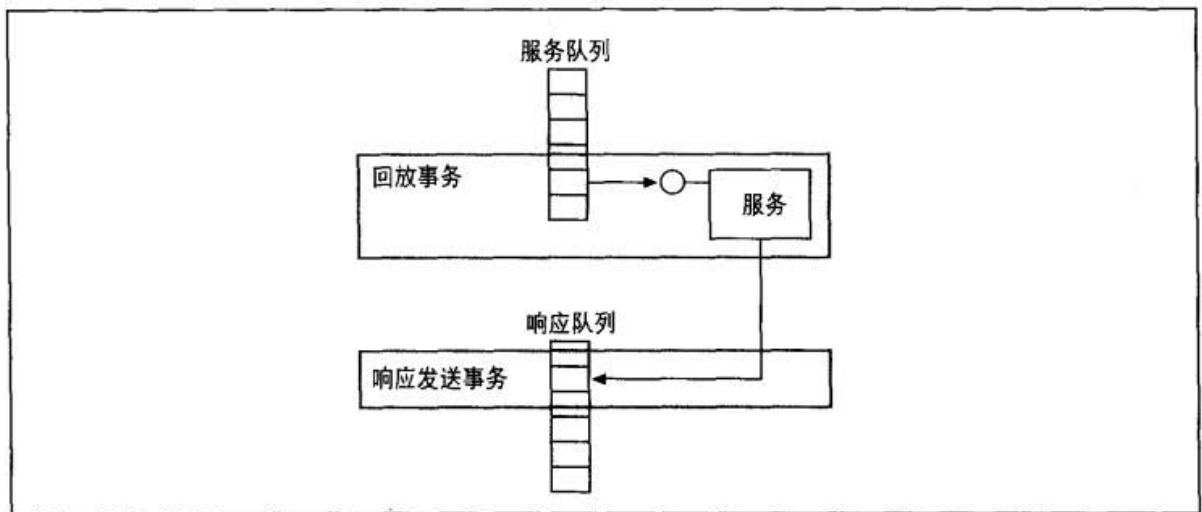


图 9-14：在一个新的事务中进行响应

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculatorResponse : ICalculatorResponse
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void OnAddCompleted(...)
    { ... }
}
```

以上是当前最常用的配置选项，因为这种配置允许重试。这就意味着，响应服务应该避免处理来自队列的响应消息的时间过长，因为这面临着中止回放事务的风险。考虑到响应服务的提供者，如果响应对于服务而言是“有胜于无，但非必要”，那么响应服务就应该在一个单独的事务中处理响应消息：

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculatorResponse : ICalculatorResponse
{
    public void OnAddCompleted(int result, ExceptionDetail error)
    {
        using(TransactionScope scope =
            new TransactionScope(TransactionScopeOption.RequiresNew))
        { ... }
    }
}
```

如果在一个新的事务中处理响应，即使该事务被中止，WCF 也不再会重试该响应。最后，对于耗时较长的响应处理而言，我们可以将响应服务配置为不使用事务（包括回放事务在内）：

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyCalculatorResponse : ICalculatorResponse
{
```

```
public void OnAddCompleted(...)  
{...}  
}
```

## HTTP 桥

MSMQ 绑定被设计用于局域网。默认情况下它是无法穿越防火墙的,更重要的是,它使用了微软特有的编码与消息格式。就算能够成功地穿过防火墙,还需要另一方也同样使用 WCF。尽管在局域网中要求服务双方都使用 WCF 是一个合理的假设,但是对于面向互联网的客户端及服务来说,这样的要求就显得不切实际了,而且它违反了面向服务的一个核心原则——服务的边界是显式的,它表示在实现服务时所采用的技术不应该影响到客户端。也就是说,互联网服务可以像局域网中的客户端与服务一样受益于队列调用,但由于在业界缺乏针对队列调用的互操作标准(WCF 中也没有提供支持),从而阻止了这种交互。不过针对这一问题,我们可以采用一项被称之为 HTTP 桥的技术。与本书中的其他技术不同,HTTP 桥不是一套在一个小框架下的辅助类,而更像是一种配置模式。HTTP 桥,正如它名字中所隐含的意义,它的设计就是用于在跨越互联网的服务与客户端之间提供队列调用支持。HTTP 桥需要用到 `WSHttpBinding`,因为它是一个事务型的绑定。HTTP 桥包含两部分,它允许 WCF 客户端将针对服务的调用放入队列,这个服务是一个使用了 WS 绑定的互联网服务;它同样允许一个使用 WS 绑定,并通过该绑定暴露了一个 HTTP 终结点的 WCF 服务,将来自互联网上的客户端调用放入队列。我们可以独立地使用桥的每一部分,当然也可以将它们联合起来使用。HTTP 桥只能在远程服务契约支持队列调用的情况下(即契约只包含单向操作)使用,但这是一种常见的情形。否则,客户端首先不应考虑 HTTP 桥。

## 设计 HTTP 桥

由于我们无法使用 WS 绑定实现队列调用,因此需要借助一个中间方将客户端与服务桥接起来。如果客户端希望在调用一个基于互联网的服务时,也能使用队列调用,就可以先调用一个被称为 `MyClientHttpBridge` 的本地(即基于局域网的)队列服务,并将调用消息放入到本地队列中。而这个客户端的队列桥接服务会在处理队列调用的过程中,使用 WS 绑定来调用远程的互联网服务。另一方面,当一个互联网服务希望接受队列调用时,它也会在本地使用一个队列。不过由于该队列无法被互联网上的非 WCF 客户端访问,队列服务将使用一个被称为 `MyServiceHttpBridge` 的联机服务作为其门面(Facade),并使用该联机服务公开一个使用 WS 绑定的终结点。`MyServiceHttpBridge` 在处理互联网调用的过程中,只需要调用相应的本地队列服务即可。图 9-15 演示了 HTTP 桥的体系架构。



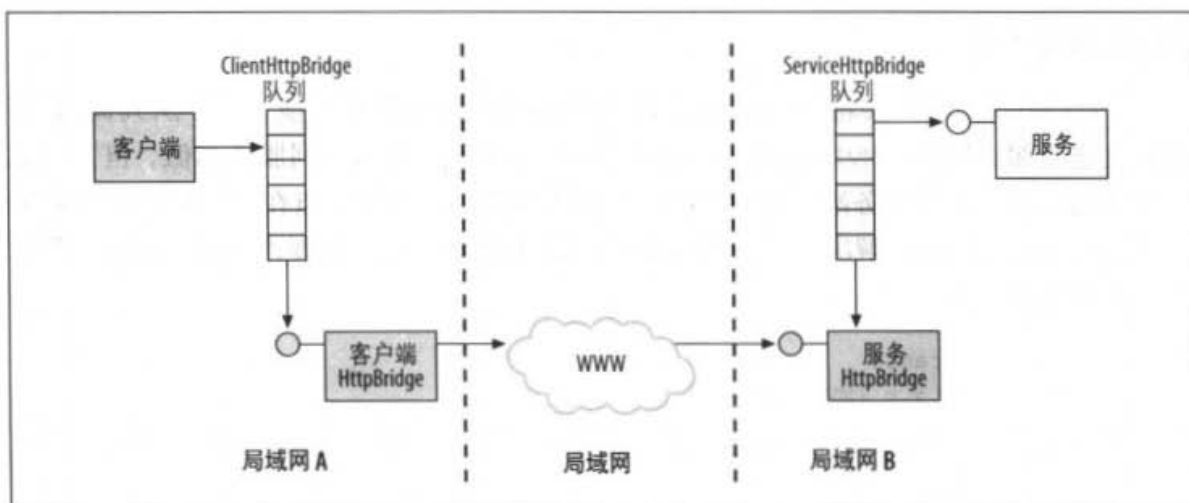


图 9-15: HTTP 桥

## 事务配置

使用HTTP桥时,我们必须在客户端与MyClientHttpBridge之间,以及MyClientHttpBridge与远程服务之间使用事务,而且服务端的HTTP桥(MyServiceHttpBridge)必须被配置为第7章中的Client事务模式,因为在向MyClientHttpBridge以及MyServiceHttpBridge(如果存在的话)回放客户端调用时,如果能够使用相同的事务,将更加符合正常队列调用的语义,如图9-16所示。

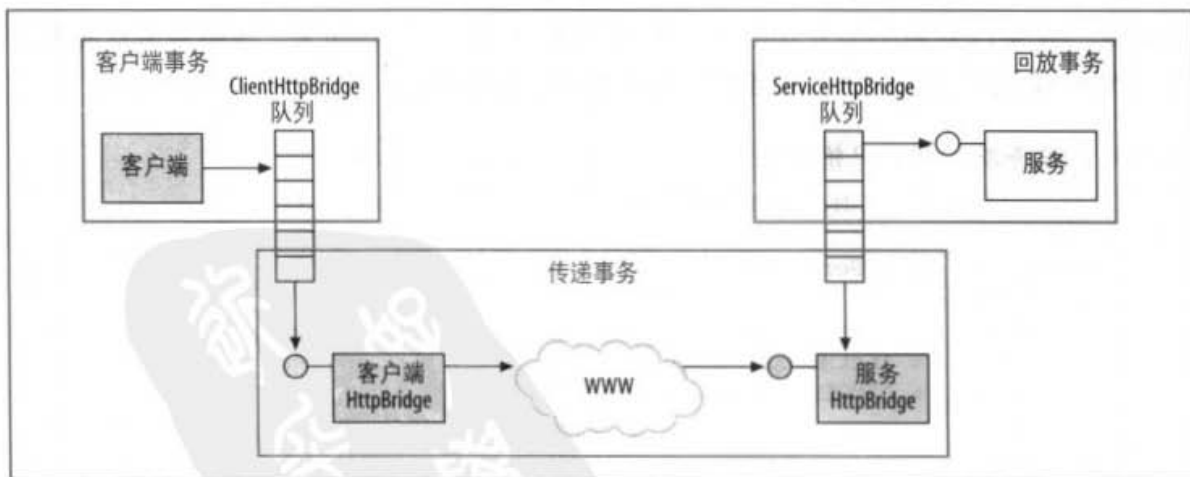


图 9-16: HTTP 桥与事务

比较图 9-16 与图 9-6。如果由于某种原因传递事务在 HTTP 桥中止了,消息将回滚到 MyClientHttpBridge 队列中,准备下一次重试。为了增加消息传递成功的可能性,我们应该为 MyClientHttpBridge 与远程服务之间的调用启用可靠性。

## 服务端配置

MyServiceHttpBridge将一个标准的使用WS绑定的联机调用转换为一个队列调用,并把它发送给服务队列。MyServiceHttpBridge实现了一个与队列服务相似,但又不完全相同的契约,因为服务端的HTTP桥虽然能够参与传入事务,但在单向操作中却是无法传播事务的。因而,解决之道就是修改契约以支持甚至是强制要求事务。例如,如果最初的服务契约如下所示:

```
[ServiceContract]
public interface IMyContract
{
    [OperationContract(IsOneWay = true)]
    void MyMethod();
}
```

那么, MyServiceHttpBridge 应该暴露以下的契约作为替代:

```
[ServiceContract]
public interface IMyContractHttpBridge
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    void MyMethod();
}
```

事实上,我们需要将 IsOneWay 设置为 false, 并使用 TransactionFlowOption.Mandatory。考虑到代码的可读性,我的建议是对接口重命名,为它加上 HttpBridge 的后缀。MyServiceHttpBridge 可以托管在服务内网的任何位置,包括服务自身的进程。例 9-32 演示了服务以及服务的 HTTP 桥所需的配置。

### 例 9-32: 服务端的 HTTP 桥配置

```
//////////////////// MyService 配置文件 //////////////////////
<services>
  <service name = "MyService">
    <endpoint
      address = "net.msmq://localhost/private/MyServiceQueue"
      binding = "netMsmqBinding"
      contract = "IMyContract"
    />
  </service>
</services>
//////////////////// MyServiceHttpBridge 配置文件 //////////////////////
<services>
  <service name = "MyServiceHttpBridge">
    <endpoint
      address = "http://localhost:8001/MyServiceHttpBridge"
      binding = "wsHttpBinding"
      bindingConfiguration = "ReliableTransactedHTTP"
    />
  </service>
</services>
```

```

        contract = "IMyContractHttpBridge"
    />
</service>
</services>

<client>
    <endpoint
        address = "net.msmq://localhost/private/MyServiceQueue"
        binding = "netMsmqBinding"
        contract = "IMyContract"
    />
</client>

<bindings>
    <wsHttpBinding>
        <binding name = "ReliableTransactedHTTP" transactionFlow = "true">
            <reliableSession enabled = "true"/>
        </binding>
    </wsHttpBinding>
</bindings>

```

MyService 服务公开了一个实现 IMyContract 契约的队列终结点。MyServiceHttpBridge 服务则公开了一个实现 IMyContractHttpBridge 契约并使用 WSHttpBinding 绑定的终结点。MyServiceHttpBridge 同时还是一个队列终结点的客户端，该队列终结点是由队列服务定义的。例9-33演示了相应的实现。注意，MyServiceHttpBridge 被配置为 Client 事务模式。

### 例 9-33: HTTP 桥在服务端的实现

```

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IMyContract
{
    // 该调用将通过 MSMQ 传入
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    { ... }
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyServiceHttpBridge : IMyContractHttpBridge
{
    // 该调用通过 HTTP 传入
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    {
        MyContractClient proxy = new MyContractClient();

        // 该调用通过 MSMQ 传出
        proxy.MyMethod();

        proxy.Close();
    }
}

```

## 客户端配置

客户端对本地的 MyClientHttpBridge 服务使用队列调用。MyClientHttpBridge 甚至可以托管在客户端进程中，当然它也可以托管在客户端内网中一个独立的机器上。MyClientHttpBridge 使用 WSHttpBinding 调用远程服务。客户端需要获得远程互联网服务的元数据（例如 IMyContractHttpBridge 的定义），并将它转化为一个队列契约（如 IMyContract）。例 9-34 演示了客户端以及它的 HTTP 桥所需的配置。

例 9-34: 客户端的 HTTP 桥配置

```

//////////////////// 客户端配置文件 //////////////////////
<client>
  <endpoint>
    address = "net.msmq://localhost/private/MyClientHttpBridgeQueue"
    binding = "netMsmqBinding"
    contract = "IMyContract"
  />
</client>
//////////////////// MyClientHttpBridge 配置文件 //////////////////////
<services>
  <service name = "MyClientHttpBridge">
    <endpoint>
      address = "net.msmq://localhost/private/MyClientHttpBridgeQueue"
      binding = "netMsmqBinding"
      contract = "IMyContract"
    />
  </service>
</services>
<client>
  <endpoint>
    address = "http://localhost:8001/MyServiceHttpBridge"
    binding = "wsHttpBinding"
    bindingConfiguration = "ReliableTransactedHTTP"
    contract = "IMyContractHttpBridge"
  />
</client>
<bindings>
  <wsHttpBinding>
    <binding name = "ReliableTransactedHTTP" transactionFlow = "true">
      <reliableSession enabled = "true"/>
    </binding>
  </wsHttpBinding>
</bindings>

```

MyClientHttpBridge 服务公开了一个实现 IMyContract 契约的队列终结点。MyClientHttpBridge 同时还是一个使用了 WS-binding 绑定的联机终结点的客户端，该联机终结点则是由联机服务 MyServiceHttpBridge 定义的。例 9-33 演示了相应的实现。

## 例 9-35: HTTP 桥在客户端的实现

```
MyContractClient proxy = new MyContractClient();

// 该调用通过 MSMQ 传出
proxy.MyMethod();

proxy.Close();

////////// 客户端桥的实现 //////////
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyClientHttpBridge : IMyContract
{
    // 该调用通过 MSMQ 传入
    [OperationBehavior(TransactionScopeRequired = true)]
    public void MyMethod()
    {
        MyContractHttpBridgeClient proxy = new MyContractHttpBridgeClient();

        // 该调用通过 HTTP 传出
        proxy.MyMethod();

        proxy.Close();
    }
}
```



## 第 10 章

# 安全

客户端与服务之间的安全交互与多个方面有关。类似于传统的客户端-服务器以及面向组件应用程序，服务需要对它的调用者进行身份验证，而且在调用者执行某些敏感的操作之前，通常还需要为它授权。此外，当我们试图在分布式系统中保护一个服务（以及它的客户端）时，无论采取何种技术，我们都需要对客户端到服务的消息加以保护。一旦消息安全到达服务，并通过验证和授权，服务可以采用多种身份验证方式执行操作。除了传统的验证、授权、传输安全以及身份管理等安全特性，我们还增加了一些更加抽象的概念，称之为总体安全策略（Overall Security Policy），它代表了个人与公司（或客户）在面对安全问题时的解决方法与思路。本章首先定义了WCF环境中的安全特性，以及在利用WCF与.NET安全机制时，可供开发者选择的解决方案。接着，我们可以看到保障各种主流应用程序安全的方式。最后，还将介绍声明式安全框架，它可以极大地简化我们对安全的实现，忽略WCF安全中的技术细节。

## 身份验证

身份验证（Authentication）代表了一种特定的动作，在该动作中我们将检验服务的调用者是否确实符合它自己所声称的身份。虽然我们通常所指的身份验证是针对服务调用者而言，但客户端同样需要服务的身份验证，它能够确保客户端调用的服务正是它希望调用的服务。当客户端的调用需要跨越互联网的时候，这一点显得尤为重要，因为如果一个怀有恶意的第三方暗中破坏了客户端的DNS服务，它就能够截获客户端的调用。WCF提供了多种身份验证机制：

### 无身份验证

服务不对调用者做身份验证，也就意味着所有的调用者都能够访问服务。

### Windows 身份验证

当 Windows 域服务器有效的时候, 服务会采用 Kerberos 协议做身份验证, 如果服务部署在工作组配置下, 则采用 NTLM。服务调用者向服务提供它的 Windows 凭证 (如一个票据或一个口令), 然后服务通过 Windows 系统对它进行身份验证。

### 用户名与密码

调用者向服务提供一个用户名和一个密码。服务通过某种类型的凭证库 (如 Windows 账号或定制凭证库, 如数据库中特定的表) 对调用者提供的凭证加以验证。

### X509 证书

客户端使用一个证书标识自己。通常服务会预先知道这些标识客户端身份的证书。这样, 在验证客户端身份时, 服务在宿主端查找客户端的证书, 并验证其合法性, 从而实现对客户端身份的验证。或者, 服务也可以隐式地信任证书的发布者, 这样客户端只需直接向服务提供证书即可。

### 定制机制

WCF 允许开发者使用任意一种协议与凭证类型, 如使用生物识别技术 (Biometric) 替代原有的身份验证机制。

### 发布口令 (Issued Token)

调用者与服务可以同时依赖于一个安全口令服务, 通过它我们可以向客户端发布一个能够被服务认可并加以信任的口令。这样的安全口令服务通常以联邦形式 (Federated) 存在, 并封装了对调用的身份验证以及对调用的保护。Windows CardSpace 就是这样一种安全口令服务。不过, 联邦安全以及 CardSpace 已经超出了本书所讨论的范围。

## 授权

授权 (Authorization) 用于处理允许服务调用者能够执行的内容, 通常是允许客户能够调用服务的某种操作。对调用者的授权建立在调用者确实符合它所声称的身份的假设之上。换句话说, 如果没有经过身份验证, 则没有必要讨论授权。服务通常会依赖于某种类型的凭证库, 在那里调用者会被映射为相应的逻辑角色。当我们向客户端授权调用某个操作时, 该操作会声明或显式地要求只有某些角色能够访问它。服务需要从凭证库中查询调用者的一个或多个角色, 从而确认它是操作所要求角色中的一员。WCF 本身支持两种类型的凭证库: 服务可以使用 Windows 组 (及其账号) 实现授权, 或者使用一个 ASP.NET Provider (如 SQL Server Provider) 保存用户账号与角色。WCF 还支持自定义的角色库, 不过迄今为止我所发现的用于实现自定义库的最简单办法, 还是实现一个自定义的 ASP.NET Provider。因此, 在本章后面会致力于对 ASP.NET Provider 的讲解。



**注意：**WCF能够根据消息中携带的一系列声明对服务的调用者进行身份验证与授权，为此它提供了一个精妙的且具有扩展性的基础架构。不过，对该机制的讨论并不在本书范围之内。

## 传输安全

身份验证与授权涉及到本地安全的两个方面——一旦消息被服务收到，如何批准调用者对服务的访问，以及在何种情况下允许调用者对服务的访问。在这一方面，WCF服务与传统的客户端-服务器模式并无太大区别。不过，身份验证与授权的前提是消息传输本身是安全的，也就是说从客户端到服务的消息传输必须是安全的，如果没有这一保证，身份验证与授权就失去意义了。传输安全涉及三个要素，必须满足这三个要素才能保证服务的安全。消息完整性关注于如何确保消息本身不会在传输途中被篡改。一个恶意的攻击者或一个传输中介都可能会截获消息并修改它的内容，例如在银行服务的转账操作中提供一个错误的账号。消息机密性关注于确保消息的机密性，从而保证第三方无法读取消息中的内容。机密性还对完整性做了补充。缺少了它，就算有恶意的一方没有篡改消息，但它仍然可以通过收集敏感的信息造成危害，例如从消息中获取账号。最后，传输安全必须提供相互验证的能力，即向客户端保证只有满足条件的服务才能够读取消息的内容，换言之，客户端连接的必须是正确的服务。一旦接收到了消息中的凭证，服务必须在本地验证凭证。互相验证同样需要检测并消除重放攻击（Replay Attacks）与拒绝服务攻击（DOS）。在这两种情况下，一个恶意的攻击者可能向服务反复重放某个合法的消息，甚至是假的非法消息，但它的重复频率足以降低服务的可用性。

## 传输安全模式

WCF支持五种不同的模式以保障传输安全的三要素。选择正确的传输安全模式，可能是在确保服务安全时应该首要考虑的事宜。这五种传输安全模式分别是 None、Transport Security、Message Security、Mixed 及 Both。

### None 传输安全模式

顾名思义，None 模式完全关闭了传输安全的功能。客户端无需向服务提供任何凭证，消息本身对于任何一个恶意攻击方都是公开的，它们甚至可以任意妄为。显然，将传输安全模式设置为 None 是十分不可取的。

### Transport Security 模式

当传输安全模式被设置为 Transport Security 时，WCF 将采用一种安全的传输通道。可用的安全通道包括 HTTP、TCP、IPC 和 MSMQ。Transport Security 对通道上的所有通

信加密,因而提供了完整性、机密性及相互验证的功能。如果不知道加密的密钥,任何试图对消息的修改都将破坏消息,使它变得毫无用处,这样就实现了完整性;而除了接收者之外,没有任何一方可以看见消息的内容,因而保障了机密性;互相验证被部分实现是因为客户凭证与消息的其余部分被一起加密,并且只有指定的消息接收者才可以读取,所以客户端无需担心消息被转发到有恶意的终结点,因为它们将无法使用该消息。一旦消息被解密,服务就可以读取客户的安全凭证并对客户端进行身份验证。

Transport Security模式要求客户端与服务就加密的细节进行协商,不过这会作为通信协议的一部分在双方各自的绑定中自动完成。Transport Security模式可以从网卡上的硬件加速功能中获益,从而避免因消息的加密、解密给宿主机器的CPU造成负担。显而易见,硬件加速有助于实现系统的高吞吐量,它甚至可以忽略实现安全时所增加的负担。Transport Security模式是获得消息最简单的模式,同时也是性能最好的。它最主要的缺点在于,它只能保证点对点的传输安全,也就是说客户端必须与服务直接连接。若在客户端与服务之间存在多个中间方,就会使得Transport Security模式出现问题,因为这些中间方可能是不安全的。因此,Transport Security模式通常只用于局域网应用程序,在那里我们可以在一个受控环境中确保客户端与服务之间只有单独一个跳转。

---

**注意:** 当我们为任意一个HTTP绑定配置传输安全时,WCF会在加载服务时验证终结点上相应的服务地址是否使用HTTPS协议,而不是简单的HTTP协议。

---

## Message Security 模式

Message Security模式只是简单地为消息本身进行加密。通过加密消息,我们将获得完整性、机密性的保障,并实现相互验证,原因等与Transport Security模式在加密通道时所能提供的上述功能相同。然而,通过对消息而不是对传输进行加密,可以使得服务能够安全地在无安全保证的传输方式(如HTTP)进行通信。因此,Message Security模式实现了端对端的安全,而不用考虑传送消息过程中所涉及的中间方的数目,也不用考虑传输是否安全。此外,Message Security模式是建立在一套行业标准的基础之上的,这些标准为互操作性以及防止诸如重放与DOS等一般性攻击而设计,WCF对这套标准所提供的支持不仅丰富,而且便于扩展。Message Security模式的缺点在于,由于实现过程的复杂性,它可能会带来调用延迟。Message Security模式一般用于互联网应用,在这样的场景下,调用模式不太频繁,而且传输也不一定安全。

## Mixed 传输安全模式

Mixed传输安全模式在实现消息的完整性和机密性及服务端身份验证时采用了Transport Security模式,而在保护客户端安全凭证的安全时又采用了Message Security模式。

Mixed 模式试图结合 Transport Security 模式与 Message Security 模式的优点，一方面获益于 Transport Security 模式的安全传输，以及它所提供的实现高吞吐量的硬件加速功能；同时又能使用 Message Security 模式提供的类型丰富且可扩展的客户安全凭证。Mixed 模式的不足之处在于，由于使用了 Transport Security 模式，它只能保证点对点的安全。应用程序的开发者很少需要使用 Mixed 模式，但它可以应用于某些高级场景。

### Both 传输安全模式

顾名思义，Both 传输安全模式同时使用了 Transport Security 模式与 Message Security 模式。消息本身使用 Message Security 模式保证安全，然后又通过安全传输将其传送给服务。Both 模式提供了最好的安全性，虽然对于大多数应用程序而言，它可能有些过分安全了，除非是那种不在乎调用延迟的断开应用程序。

## 传输安全模式的配置

传输安全模式的配置是在绑定中完成的，并且客户端与服务配置的传输安全模式必须相同，当然也必须满足它的要求。与其他任何绑定配置一样，我们既可以通过编程方式配置传输安全模式，也可以直接修改配置文件达到同样的目的。所有的绑定（除对等网绑定以外）都提供了一个指明传输安全模式的构造参数，而且所有的绑定都提供了一个使用特定枚举量标识配置模式的 Mode 属性，该属性包含在绑定的 Security 属性中。如表 10-1 所示，并非所有的绑定都支持所有的传输安全模式，并且每种绑定所支持的模式都是由针对该绑定的目标场景所驱动的。

表 10-1：绑定及传输安全模式

名称	None 模式	Transport Security 模式	Message Security 模式	Mixed 模式	Both 模式
BasicHttpBinding	Yes (默认值)	Yes	Yes	Yes	No
NetTcpBinding	Yes	Yes (默认值)	Yes	Yes	No
NetPeerTcpBinding	Yes	Yes (默认值)	Yes	Yes	No
NetNamedPipeBinding	Yes	Yes (默认值)	No	No	No
WSHttpBinding	Yes	Yes	Yes (默认值)	Yes	No
WSFederationHttpBinding	Yes	No	Yes (默认值)	Yes	No
WSDualHttpBinding	Yes	No	Yes (默认值)	No	No
NetMsmqBinding	Yes	Yes (默认值)	Yes	No	Yes

所有的局域网绑定 (NetTcpBinding、NetNamedPipeBinding 和 NetMsmqBinding) 都默认使用 Transport Security 模式。原因在于相对互联网而言, 局域网是一种相对较为安全的环境, 而且使用 Transport Security 模式可以获得更好性能。值得注意的是, 三种传输协议 (TCP、IPC 和 MSMQ) 本质上都支持传输安全, 因此对于服务或客户端的开发人员来说无需编写特别的代码以实现传输安全。不过, 局域网绑定同样可以配置为 None 模式, 也就是它们可以使用相同的没有安全的传输方式。注意, NetNamedPipeBinding 只支持 None 模式与 Transport Security 模式, 因为在 IPC 上使用 Message Security 模式是没有意义的, 这是由于在 IPC 上, 从客户端到服务只存在一个跳转。同样需要注意的是, 只有 NetMsmqBinding 支持 Both 模式。

所有的互联网绑定都默认使用 Message Security 模式, 从而使得它们能够基于不安全的传输方式传递消息, 即 HTTP。注意, 虽然 WSHttpBinding 可以使用 Transport Security 模式, 但 WSDualHttpBinding 却不支持。这是因为该绑定采用了一个单独的 HTTP 通道从服务连接回调客户端, 而这个通道是不便于使用 HTTPS 协议的, 因为回调客户端通常不像一个服务那样托管在一个真实的 Web 服务器上。

所有 WCF 绑定都将被配置为使用某种类型的传输安全模式, 默认是安全的, 只有 BasicHttpBinding 默认是非安全的。因为基本绑定的设计目的就是使得一个 WCF 服务看起来像一个旧的 ASMX 服务, 而 ASMX 默认是非安全的。也就是说, 我们可以, 也应该将 BasicHttpBinding 配置为使用诸如 Message Security 之类的传输安全模式。

## 专用的绑定配置

BasicHttpBinding 使用 BasicHttpSecurityMode 枚举类型配置传输模式。该枚举类型可以通过绑定的 Security 属性中的 Mode 属性访问:

```
public enum BasicHttpSecurityMode
{
    None,
    Transport,
    Message,
    TransportWithMessageCredential,
    TransportCredentialOnly
}
public sealed class BasicHttpSecurity
{
    public BasicHttpSecurityMode Mode
    {get;set;}
    // 更多成员
}
public class BasicHttpBinding : Binding,...
{
    public BasicHttpBinding();
    public BasicHttpBinding(BasicHttpSecurityMode SecurityMode);
```

```

    public BasicHttpSecurity Security
    {get;}
    // 更多成员
}

```

其中, `Security` 属性的类型为 `BasicHttpSecurity`。 `BasicHttpBinding` 类型的一个构造函数使用 `BasicHttpSecurityMode` 枚举类型作为参数。我们可以在构造 `BasicHttpBinding` 的时候指定消息的传输安全模式采用 `Message Security` 模式, 不过, 在创建 `BasicHttpBinding` 之后也可以通过设置 `Mode` 属性达到同样的目的。因此, 例 10-1 中的 `binding1` 与 `binding2` 是等效的。

#### 例 10-1: 以编码方式保障 `BasicHttpBinding` 的安全

```

BasicHttpBinding binding1 = new BasicHttpBinding(BasicHttpSecurityMode.Message);

BasicHttpBinding binding2 = new BasicHttpBinding();
binding2.Security.Mode = BasicHttpSecurityMode.Message;

```

如果不采用编码方式的设置, 我们也可以使用配置文件, 如例 10-2 所示。

#### 例 10-2: 以管理方式保障 `BasicHttpBinding` 的安全

```

<bindings>
  <basicHttpBinding>
    <binding name = "SecuredBasic">
      <Security mode = "Message">
      </Security>
    </binding>
  </basicHttpBinding>
</bindings>

```

其余的绑定都使用它们各自的枚举类型以及专用的安全类, 但是它们的配置方式与例 10-1 或例 10-2 十分相似。例如, `NetTcpBinding`、`NetPeerTcpBinding` 以及 `WSHttpBinding` 都使用 `SecurityMode` 枚举类型, 该类型定义如下:

```

public enum SecurityMode
{
    None,
    Transport,
    Message,
    TransportWithMessageCredential // Mixed 模式
}

```

虽然不是所有的绑定都会提供一个匹配的构造参数, 不过它们都提供了一个 `Security` 属性。 `NetNamedPipeBinding` 采用 `NetNamedPipeSecurityMode` 枚举类型, 该枚举类型只支持 `None` 模式与 `Transport Security` 模式:

```

public enum NetNamedPipeSecurityMode
{
    None,

```



```
    Transport  
}
```

WSDualHttpBinding使用的是WSDualHttpSecurityMode枚举类型，它只支持None模式与和Message Security模式：

```
public enum WSDualHttpSecurityMode  
{  
    None,  
    Message  
}
```

NetMsmqBinding采用了NetMsmqSecurityMode枚举类型：

```
public enum NetMsmqSecurityMode  
{  
    None,  
    Transport,  
    Message,  
    Both  
}
```

NetMsmqSecurityMode是唯一支持Both传输安全模式的枚举类型。

几乎每个绑定都有其专用的安全模式枚举值，原因在于WCF安全的设计者在安全与系统复杂性之间选择了安全。设计者完全可以只定义一个包含所有传输安全模式的枚举类型，然而，这样就有可能使得用户会在编译期为绑定分配一个非法的传输安全模式，例如将NetNamedPipeBinding配置为Message Security模式，或将WSDualHttpBinding配置为Transport Security模式。采用各个绑定专用的枚举类型，就避免在配置传输安全模式时出现错误，随之带来的还有更多令人心动的特性。

## Transport Security 模式与凭证

WCF提供了多种客户端凭证类型供用户选择。客户端既可以使用经典的用户名加密码的方式标识自己，也可以使用一个Windows安全口令。如果NTLM协议或Kerbores协议可用，则会通过它们验证Windows凭证。客户端还可以使用一个X509证书，或者干脆不提供任何凭证而采取匿名模式。当我们采用Transport Security模式作为传输安全模式时，并不是所有绑定都支持所有的客户端凭证类型，如表10-2所示。

表 10-2：绑定与Transport Security模式的客户端凭证

名称	None	Windows	UserName	Certificate
BasicHttpBinding	Yes（默认值）	Yes	Yes	Yes
NetTcpBinding	Yes	Yes（默认值）	No	Yes

表 10-2: 绑定与 Transport Security 模式的客户端凭证 (续)

名称	None	Windows	UserName	Certificate
NetPeerTcpBinding	No	No	Yes (默认值)	Yes
NetNamedPipeBinding	No	Yes (默认值)	No	No
WSHttpBinding	Yes	Yes (默认值)	Yes	Yes
WSFederationHttpBinding	N/A	N/A	N/A	N/A
WSDualHttpBinding	N/A	N/A	N/A	N/A
NetMsmqBinding	Yes	Yes (默认值)	No	Yes

何种绑定支持何种凭证类型在很大程度上由设计绑定的目标场景决定。例如,所有的局域网绑定都默认使用 Windows 凭证,因为它们是在一个 Windows 环境中使用的,而 BasicHttpBinding 默认不使用凭证,就像一个经典的 ASMX Web 服务那样。因为 WSFederationHttpBinding 与 WSDualHttpBinding 根本无法使用 Transport Security 模式,自然就不支持任何客户端凭证类型。

## Message Security 模式与凭证

使用 Message Security 传输安全模式时, WCF 允许应用程序采用与 Transport Security 模式相同的凭证类型,此外,还增加了一个发布令牌凭证类型。同样,在使用 Message Security 模式时,并非所有绑定都支持所有的客户端凭证类型,如表 10-3 所示。

表 10-3: 绑定与 Message Security 模式的客户端凭证

名称	None	Windows	UserName	Certificate	Issued token
BasicHttpBinding	No	No	No	Yes	No
NetTcpBinding	Yes	Yes (默认值)	Yes	Yes	Yes
NetPeerTcpBinding	N/A	N/A	N/A	N/A	N/A
NetNamedPipeBinding	N/A	N/A	N/A	N/A	N/A
WSHttpBinding	Yes	Yes (默认值)	Yes	Yes	Yes
WSFederationHttpBinding	N/A	N/A	N/A	N/A	N/A
WSDualHttpBinding	Yes	Yes (默认值)	Yes	Yes	Yes
NetMsmqBinding	Yes	Yes (默认值)	Yes	Yes	Yes

虽然让所有支持 Message Security 模式的局域网绑定均默认使用 Windows 凭证是有根据的,但值得注意的是,尽管(将在稍后介绍)互联网应用很少在 HTTP 上使用 Windows



凭证,但是互联网绑定 (WSHttpBinding 与 WSDualHttpBinding) 同样默认使用了 Windows 凭证。之所以如此,是为了让用户安全地使用 WCF 直接提供的各种绑定,而不是首先求助于定制凭证库。

---

**警告:** BasicHttpBinding 只在 Mixed 模式下支持使用用户名客户端凭证,以保障消息的安全。这里可能会导致一个运行时的验证错误,因为 BasicHttpMessageCredentialType 枚举类型中含有 BasicHttpMessageCredentialType.UserName 这个值。

---

## 身份管理

身份管理作为服务安全的一方面,主要用于处理客户端应该向服务发送何种安全身份,以及反过来服务又能够对客户端的身份做些什么。不仅如此,在设计一个服务时,我们还需要预先确定服务应以何种身份执行。服务既可以以它自己的身份执行,也可以以客户端(在适当的时候)的身份执行,甚至还可以采用一种混合的形式,也就是说它可以在一个操作中交替使用它自己的身份或客户端的身份,甚至是一个第三方的身份。选取正确的身份对于应用程序的可伸缩性及管理成本会产生重大影响。在 WCF 中,当启用身份管理时,安全身份将沿着调用链传播,无论服务以何种身份执行,它总能查出它的调用者是谁。

## 总体策略

在包括验证、授权、传输安全及身份管理这些传统安全所考虑方面之外,我们希望加入一个新的元素,尽管它不符合常规,技术性也不是很强,但对于我们来说同样重要:针对安全问题,系统的业务乃至我们个人将采取什么样的解决方案?也就是说我们的安全策略是什么?我们相信在绝大多数的情况下,我们的应用程序是离不开安全保障的。而且,尽管伴随着安全而来的是性能及吞吐量上的损失,但这些都不影响我们对安全的需求。简单地说,它们是系统正常运转的必然代价。为实现安全而付出的代价是在设计与管理现代联机应用程序时不可避免的。那种开发者不用考虑安全问题就直接部署应用程序,只依赖目标环境的外围安全措施(如员工的门禁卡或防火墙之类的物理安全)来确保系统安全的日子已经一去不复返了。

由于大多数开发者不可能成为一名全职的安全专家(他们也不应该这样做),因而我们所倡导的总体安全策略将采用一种简单的解决方案:启用最高等级的安全保障直到有人抱怨为止。如果在最高安全等级下,应用程序的性能及吞吐量仍然够用,那么就保持现状;如果性能不理想,也只有在此时,我们才应该对潜在的威胁进行细致分析,以找出

可以牺牲安全来换取性能的地方。从我们的经验来看，只有在极少情况下真正需要采取这种手段，而且，大多数的开发者可能永远也不需要牺牲系统的安全性。本章所描述的安全策略遵循了总体安全策略。WCF在安全问题上的总体方案在很大程度上与我们的方案是一致的，在本章后面，我还会明确指出两者的不同之处，以及调整的方法。除了 BasicHttpBinding 这个显而易见的例外情形，WCF 的其他绑定都默认有安全保障，即使是 BasicHttpBinding 也可以很容易地保证其安全。WCF 的所有其他绑定均依赖于传输安全，并默认地对所有服务的调用者进行身份验证。

## 场景驱动方式

到目前为止，安全是 WCF 中最复杂的部分。举个例子，下面的列表展示了在每个 WCF 操作的调用中管理安全的元素：

- 服务契约
- 操作契约
- 错误契约
- 服务行为
- 操作行为
- 宿主配置
- 方法的配置及代码
- 客户端行为
- 代理配置
- 绑定配置

以上列表的每一个条目都可能具有大量的与安全相关的属性。显然，由它们构成的可能的排列组合的数目是惊人的。然而，并非所有的组合都被允许或被支持，而且，也不是所有允许的组都是有意义的或是一致的。例如，尽管在技术上可行，但在某个同构的 Windows 局域网中使用一个证书作为客户凭证的意义并不大，同样地，在一个互联网应用程序中使用 Windows 账户也没有什么意义。本书所选取的解决方案主要关注于几个关键场景，它们基本上能够满足当前大多数应用程序在安全方面的需求。

这些场景包括：

- 局域网应用程序
- 互联网应用程序

- B2B 应用程序
- 匿名应用程序
- 无安全需求

下面将演示如何保持这些场景的一致性与安全性。在每个场景中，将讨论如何对传输安全、身份验证、授权及身份管理这些安全特性提供支持。如果我们面临其他的场景，也可以按照我们的分析方法派生出所需的安全特性及设置。

## 局域网应用程序

客户端与服务两者都使用 WCF 是局域网应用的特征之一，并且客户端与服务都部署在同一个局域网中。客户端位于防火墙之后，我们可以采用 Windows 提供的安全机制实现传输安全、身份验证及授权，使用 Windows 账户与 Windows 组保存客户凭证。局域网的应用场景涵盖了大范围的业务应用，从金融业到制造业再到机构内部的 IT 应用。局域网场景同样也是所有场景中能够为开发者提供安全配置选项最丰富的一个场景。接下来有关局域网应用场景的章节也同样会介绍到其他应用场景中的术语、技术以及用到的类型。

## 保证局域网绑定的安全

在局域网场景中，我们应该采用局域网绑定，即 `NetTcpBinding`、`NetNamedPipeBinding` 与 `NetMsmqBinding`。因为对服务的调用总是点对点的，所以我们可以依赖 Transport Security 模式实现传输安全。自然，Transport Security 模式也就成为了局域网绑定默认的传输安全模式（见表 10-1）。同样地，Windows 类型也成为了局域网场景中默认的客户凭证类型（见表 10-2）。需要注意的是，客户端与服务必须采用上述相同的配置。

### Transport Security 的保护级别

三种局域网绑定都需要配置为 Transport Security 传输安全模式，其中每个绑定都有一个可配置的保护级别，它是 Transport Security 模式的主开关。这三种保护级别分别为：

#### *None*

在消息从客户端传输到服务的过程中，WCF 不做任何保护。任何有恶意的一方都可以读取消息的内容甚至改写它。

#### *Sign*

当配置为这种保护级别时，WCF 将为消息附加一个加密的校验和 (Checksum)，以

确保该消息只可能出自一个经过验证的发送方。一旦接收到消息,服务会根据消息内容重新计算出校验和,并将它与附加在消息中的校验和进行比较。如果两者不一致,消息将被拒绝。因而消息就无法被篡改了。不过,消息的内容在传输过程中还是可见的。

### *EncryptAndSign*

当配置为这种保护级别时,WCF不仅对消息进行签名,还会为它的内容进行加密。加密与签名提供了完整性、机密性以及真实性的保护。

Sign保护级别很明显是为了在安全性与性能方面取得平衡的结果。但是我却认为我们应该避免这种平衡方式,而尽量选用EncryptAndSign这种保护级别。WCF用ProtectionLevel枚举类型表示保护级别,定义如下:

```
public enum ProtectionLevel
{
    None,
    Sign,
    EncryptAndSign
}
```

并非所有的局域网绑定都默认为相同的保护级别。NetTcpBinding与NetNamedPipeBinding默认为EncryptAndSign,而NetMsmqBinding默认为Sign。

## NetTcpBinding 的配置

NetTcpBinding接收一个用于指定传输安全模式的构造参数:

```
public class NetTcpBinding : ...
{
    public NetTcpBinding(SecurityMode SecurityMode);
    public NetTcpSecurity Security
    {get;}
    // 更多成员
}
```

NetTcpSecurity类型的Security属性包含了传输安全模式Transport与Message,以及对应的两个属性:

```
public sealed class NetTcpSecurity
{
    public SecurityMode Mode
    {get;set;}
    public MessageSecurityOverTcp Message
    {get;}
    public TcpTransportSecurity Transport
    {get;}
}
```

在局域网安全场景中，我们应该选择 Transport Security 传输安全模式，并为类型为 TcpTransportSecurity 的 Transport 属性赋值：

```
public sealed class TcpTransportSecurity
{
    public TcpClientCredentialType ClientCredentialType
    {get;set;}

    public ProtectionLevel ProtectionLevel
    {get;set;}
}
```

Transport 属性应该在初始化的时候使用 TcpClientCredentialType 枚举类型将客户凭证类型设为 Window 类型。TcpClientCredentialType 枚举类型定义如下：

```
public enum TcpClientCredentialType
{
    None,
    Windows,
    Certificate
}
```

Transport 属性应将保护级别设置为 ProtectionLevel.EncryptAndSign。由于以上两项设置都是 NetTcpBinding 的默认配置，因此，以下两个声明是等效的：

```
NetTcpBinding binding1 = new NetTcpBinding();

NetTcpBinding binding2 = new NetTcpBinding(SecurityMode.Transport);
binding2.Security.Transport.ClientCredentialType = TcpClientCredentialType.Windows;
binding2.Security.Transport.ProtectionLevel = ProtectionLevel.EncryptAndSign;
```

或改用以下配置文件：

```
<bindings>
  <netTcpBinding>
    <binding name = "TCPWindowsSecurity">
      <Security mode = "Transport">
        <transport
          clientCredentialType = "Windows"
          protectionLevel = "EncryptAndSign"
        />
      </Security>
    </binding>
  </netTcpBinding>
</bindings>
```

出于完整性的考虑，尽管局域网场景可能并不需要用到，下面仍然演示了如何将 NetTcpBinding 配置为使用 Message Security 传输安全模式，并使用 UserName 类型的客户凭证：

```
public enum MessageCredentialType
{
    None,
    Windows,
    UserName,
    Certificate,
    IssuedToken
}
public sealed class MessageSecurityOverTcp
{
    public MessageCredentialType ClientCredentialType
    {get;set;}
    // 更多成员
}
NetTcpBinding binding = new NetTcpBinding(SecurityMode.Message);
binding.Security.Message.ClientCredentialType = MessageCredentialType.UserName;
```

NetTcpSecurity 提供了类型为 MessageSecurityOverTcp 的 Message 属性。我们需要使用 MessageCredentialType 枚举类型设置凭证类型。大多数绑定会使用 MessageCredentialType 枚举类型来表示 Message Security 模式下的客户凭证。

图 10-1 展示了 NetTcpBinding 中与安全相关的元素。

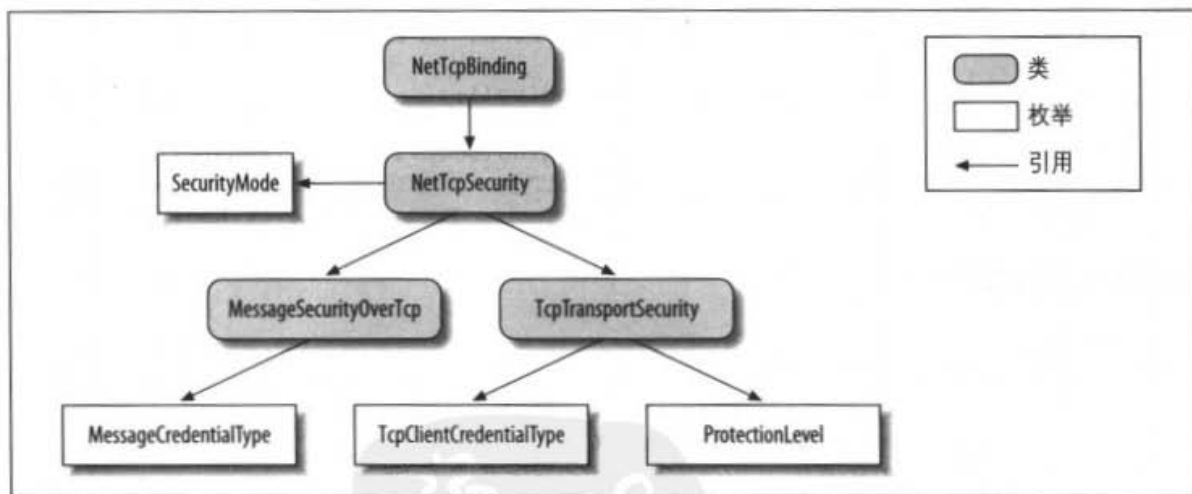


图 10-1: NetTcpBinding 及安全

NetTcpBinding 包含了一个指向 NetTcpSecurity 的引用, 该引用使用 SecurityMode 枚举类型来指定传输安全模式。使用 Transport Security 模式时, NetTcpSecurity 会使用一个 TcpTransportSecurity 类型的实例, 其中包含了用于定义客户凭证类型的 TcpClientCredentialType 枚举类型, 以及用于定义保护级别的 ProtectionLevel 枚举类型; 使用 Message Security 模式时, NetTcpSecurity 会使用一个 MessageSecurityOverTcp 类型的实例, 其中包含了用于定义客户凭证类型的 MessageCredentialType 枚举类型。

## NetNamedPipeBinding 的配置

NetNamedPipeBinding 接收一个用于指定传输安全模式的构造参数：

```
public class NetNamedPipeBinding : Binding,...
{
    public NetNamedPipeBinding(NetNamedPipeSecurityMode SecurityMode);

    public NetNamedPipeSecurity Security
    {get;}
    // 更多成员
}
```

NetNamedPipeSecurity 类型的 Security 属性包含了传输模式 Transport 与 None，以及一个对应 Transport 模式的属性：

```
public sealed class NetNamedPipeSecurity
{
    public NetNamedPipeSecurityMode Mode
    {get;set;}
    public NamedPipeTransportSecurity Transport
    {get;}
}
```

在局域网安全场景中，我们应该选择 Transport Security 传输安全模式，并为类型为 NamedPipeTransportSecurity 的 Transport 属性赋值：

```
public sealed class NamedPipeTransportSecurity
{
    public ProtectionLevel ProtectionLevel
    {get;set;}
}
```

Transfer 属性应在初始化时将保护级别设置为 ProtectionLevel.EncryptAndSign。由于以上两项设置都是 NetNamedPipeBinding 的默认配置，因此，以下两个声明是等效的：

```
NetNamedPipeBinding binding1 = new NetNamedPipeBinding();

NetNamedPipeBinding binding2 = new NetNamedPipeBinding(
    NetNamedPipeSecurityMode.Transport);
binding2.Security.Transport.ProtectionLevel = ProtectionLevel.EncryptAndSign;
```

或改用以下配置文件：

```
<bindings>
  <netNamedPipeBinding>
    <binding name = "IPCWindowsSecurity">
      <Security mode = "Transport">
        <transport protectionLevel = "EncryptAndSign"/>
      </Security>
    </binding>
  </netNamedPipeBinding>
</bindings>
```



```
</netNamedPipeBinding>
</bindings>
```

由于 `NetNamedPipeBinding` 只支持 Windows 凭证（见表 10-2），因此 `NetNamedPipeBinding` 中无需（也没有选择）设定客户凭证的类型。图 10-2 展示了 `NetNamedPipeBinding` 中与安全相关的元素。

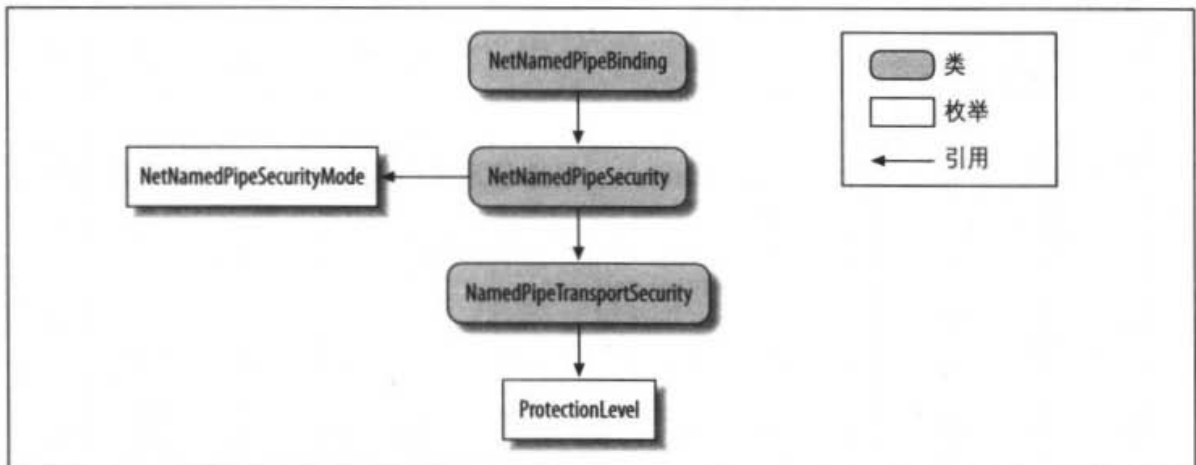


图 10-2: `NetNamedPipeBinding` 及安全

`NetNamedPipeBinding` 包含了一个指向 `NetNamedPipeSecurity` 的引用，该引用使用 `NetNamedPipeSecurityMode` 枚举类型指定传输安全模式。使用 `Transport Security` 模式时，`NetNamedPipeSecurity` 会使用一个 `NamedPipeTransportSecurity` 类型的实例，其中包含了用于定义保护级别的 `ProtectionLevel` 枚举类型。

### NetMsmqBinding 的配置

`NetMsmqBinding` 提供了一个用于指定传输安全模式的构造参数以及一个 `Security` 属性：

```
public class NetMsmqBinding : MsmqBindingBase,...
{
    public NetMsmqBinding(NetMsmqSecurityMode SecurityMode);
    public NetMsmqSecurity Security
    {get;}
    // 更多成员
}
```

`NetMsmqSecurity` 类型的 `Security` 属性包含了传输安全模式 `Transport` 与 `Message`，以及对应的两个属性：

```
public sealed class NetMsmqSecurity
{
    public NetMsmqSecurityMode Mode
```

```

    {get;set;}
    public MsmqTransportSecurity Transport
    {get;}
    public MessageSecurityOverMsmq Message
    {get;}
}

```

在局域网安全场景中，我们应该选择 Transport Security 传输安全模式，并为类型为 MsmqTransportSecurity 的 Transport 属性赋值：

```

public sealed class MsmqTransportSecurity
{
    public MsmqAuthenticationMode MsmqAuthenticationMode
    {get;set;}
    public ProtectionLevel MsmqProtectionLevel
    {get;set;}
    // 更多成员
}

```

Transport 属性应该在初始化的时候使用 MsmqAuthenticationMode 枚举类型将客户凭证类型设置为 Windows 域。MsmqAuthenticationMode 枚举类型定义如下：

```

public enum MsmqAuthenticationMode
{
    None,
    WindowsDomain,
    Certificate
}

```

Windows 域是默认的凭证类型。此外，由于 MSMQ 绑定默认的保护级别为 ProtectionLevel.Signed，因此需要将保护级别设置为 ProtectionLevel.EncryptAndSign。以下两种定义是等效的：

```

NetMsmqBinding binding1 = new NetMsmqBinding();
binding1.Security.Transport.MsmqProtectionLevel = ProtectionLevel.EncryptAndSign;

NetMsmqBinding binding2 = new NetMsmqBinding();
binding2.Security.Mode = NetMsmqSecurityMode.Transport;
binding2.Security.Transport.MsmqAuthenticationMode =
    MsmqAuthenticationMode.WindowsDomain;
binding2.Security.Transport.MsmqProtectionLevel = ProtectionLevel.EncryptAndSign;

```

或改用以下配置文件：

```

<bindings>
  <netMsmqBinding>
    <binding name = "MSMQWindowsSecurity">
      <Security mode = "Transport">
        <transport
          msmqAuthenticationMode = "WindowsDomain"
          msmqProtectionLevel = "EncryptAndSign"
        />

```

```
        </Security>
      </binding>
    </netMsmqBinding>
  </bindings>
```

图 10-3 展示了 NetMsmqBinding 中与安全相关的元素。

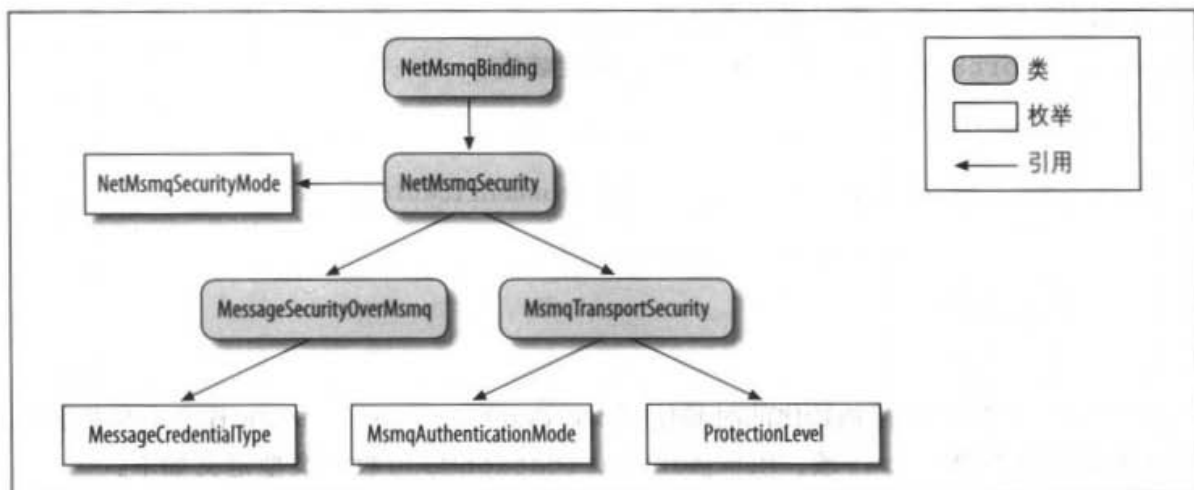


图 10-3: NetMsmqBinding 及安全

NetMsmqPipeBinding 包含了一个指向 NetMsmqSecurity 的引用，该引用使用 NetMsmqSecurityMode 枚举类型指定传输安全模式。使用 Transport Security 模式时，NetMsmqSecurity 会使用一个 MsmqTransportSecurity 类型的实例，其中包含了用于定义客户凭证类型的 MsmqAuthenticationMode 枚举类型，以及用于定义保护级别的 ProtectionLevel 枚举类型；与 NetTcpBinding 类似，NetMsmqPipeBinding 中也含有那些用于控制 Message Security 传输安全模式的类型引用。

## 消息保护

尽管一个服务应该采用安全的最高级别，然而实际上，服务完全是由它的宿主决定的，因为服务的绑定是由宿主配置的。当服务被部署在处于未知环境中的一个宿主时，这个问题显得尤为严重。为了弥补这一隐患，WCF 允许服务的开发者对保护级别做出强制要求，即对他们开发的服务能够工作的最小保护级别做出限定。服务和客户端两者都可以相互独立地限定各自的保护级别。我们一共可以在三个地方限定保护级别。当我们在服务契约上限定保护级别时，该契约上所有的操作都将被认为是敏感的并受保护的。当我们在具体的操作契约上做限定时，只有该操作是受保护的，而相同契约的其他操作则不受保护。最后，我们也可以在错误契约上限定保护级别，因为有时候返回给客户端的错误信息也是敏感的，它可能含有了参数值、异常消息及调用堆栈的信息。以上各个契约的特性中都定义了 ProtectionLevel 枚举类型的 ProtectionLevel 属性：

```
[AttributeUsage(AttributeTargets.Interface|AttributeTargets.Class,
    Inherited = false)]
public sealed class ServiceContractAttribute : Attribute
{
    public ProtectionLevel ProtectionLevel
    {get;set;}
    // 更多成员
}
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
    public ProtectionLevel ProtectionLevel
    {get;set;}
    // 更多成员
}
[AttributeUsage(AttributeTargets.Method,AllowMultiple = true,
    Inherited = false)]
public sealed class FaultContractAttribute : Attribute
{
    public ProtectionLevel ProtectionLevel
    {get;set;}
    // 更多成员
}
```

下面演示了如何在一个服务契约上设置保护级别：

```
[ServiceContract(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
interface IMyContract
{...}
```

在契约特性上设置 `ProtectionLevel` 属性，仅仅设置了服务的最低警戒线，即该契约能够接受的最低的保护级别。如果绑定被配置为一个更低保护级别，那么加载服务或打开代理时会引发一个 `InvalidOperationException` 异常；假如绑定被配置为一个更高的保护级别，契约将欣然接受。契约特性上 `ProtectionLevel` 属性的默认值为 `ProtectionLevel.None`，也就是说它不起任何作用。

服务所期望的保护级别被认为是服务在本地的实现细节，所以它不会随着服务的元数据一同导出。正因为如此，客户端可以要求一个不同的保护级别，并独立于服务做出强制限定。

---

**注意：** 尽管非局域网的绑定没有提供一个保护级别的属性，在使用 `Transport Security` 或 `Message Security` 传输安全模式时，必须满足应用于服务、操作或错误契约上的保护级别约束条件。不过在关闭安全的 `None` 模式中，则无法满足约束条件。

---

## 身份验证

默认地，当某个客户端调用一个代理，而该代理的目标是一个绑定被设置为使用 Transport Security 传输安全模式以及 Windows 凭证的终结点时，客户端无需显式地传递它的凭证。WCF 将自动地向服务传递客户端进程的 Windows 身份标识。

```
class MyContractClient : ClientBase<IMyContract>,IMyContract
{...}

MyContractClient proxy = new MyContractClient();
proxy.MyMethod(); // 在此处传递客户端身份
proxy.Close();
```

当服务接收到客户端的调用时，如果客户凭证代表了一个合法的 Windows 账户，则服务端的 WCF 就会鉴别调用者的身份，以允许它访问服务上的操作。

## 提供另一种 Windows 凭证

客户端可以传递另一种 Windows 凭证代替其所处进程中的身份标识。ClientBase<T> 基类提供了类型为 ClientCredentials 的 ClientCredentials 属性：

```
public abstract class ClientBase<T> : ...
{
    public ClientCredentials ClientCredentials
    {get;}
}
public class ClientCredentials : ...,IEndpointBehavior
{
    public WindowsClientCredential Windows
    {get;}
    // 更多成员
}
```

ClientCredentials 包含了 WindowsClientCredential 类型的 Windows 属性，定义如下：

```
public sealed class WindowsClientCredential
{
    public NetworkCredential ClientCredential
    {get;set;}
    // 更多成员
}
```

WindowsClientCredential 中还包含了 NetworkCredential 类型的 ClientCredential 属性，客户端可以通过该属性设置另一种安全凭证：

```
public class NetworkCredential : ...
{
```

```

public NetworkCredential();
public NetworkCredential(string userName,string password);
public NetworkCredential(string userName,string password,string domain);

public string Domain
{get;set;}
public string UserName
{get;set;}
public string Password
{get;set;}
}

```

例 10-3 演示了如何使用这些类与属性，向服务提供不同于进程标识的另一种 Windows 凭证。

### 例 10-3：提供另一种 Windows 凭证

```

NetworkCredential credentials = new NetworkCredential();
credentials.Domain = "MyDomain";
credentials.UserName = "MyUsername";
credentials.Password = "MyPassword";

MyContractClient proxy = new MyContractClient();
proxy.ClientCredentials.Windows.ClientCredential = credentials;

proxy.MyMethod();
proxy.Close();

```

注意，在例 10-3 中，客户端必须实例化一个类型为 NetworkCredential 的新对象，而且不能简单地在已有的 WindowsClientCredential 类型的 ClientCredential 属性中指定该凭证。一旦使用了一个指定的标识，代理就不能在以后使用任何其他的标识。当客户端向服务提供的凭证需要在程序运行中动态收集时，我们会使用例 10-3 中的技术，比如我们在客户端使用了一个登录对话框。另一方面，如果作为替换的凭证是一个静态值，客户端的开发者可以将它们封装在代理的构造函数中：

```

public partial class MyContractClient: ClientBase<IMyContract>,IMyContract
{
    public MyContractClient()
    {
        SetCredentials();
    }
    /* 更多构造函数 */
    void SetCredentials()
    {
        ClientCredentials.Windows.ClientCredential =
            new NetworkCredential("MyClient", "MyPassword", "MyDomain");
    }
    public void MyMethod()
    {
        Channel.MyMethod();
    }
}

```

如果使用通道工厂而不是代理类，`ChannelFactory` 基类则提供了类型为 `ClientCredentials` 的 `Credential` 属性：

```
public abstract class ChannelFactory : ...
{
    public ClientCredentials Credentials
    {get;}
    // 更多成员
}
public class ChannelFactory<T> : ChannelFactory, ...
{
    public T CreateChannel();
    // 更多成员
}
```

就像例 10-3 那样在 `Credentials` 属性中简单地设置替换的凭证：

```
ChannelFactory<IMyContract> factory = new ChannelFactory<IMyContract>("");
factory.Credentials.Windows.ClientCredential = new NetworkCredential(...);
IMyContract proxy = factory.CreateChannel();
using(proxy as IDisposable)
{
    proxy.MyMethod();
}
```

注意，我们不能使用 `ChannelFactory<T>` 类中的 `CreateChannel()` 静态方法，因为我们必须首先实例化一个工厂以访问其 `Credentials` 属性。

## 身份

所有 Windows 进程在运行时都有一个通过验证的安全身份，作为 WCF 服务宿主的进程当然也不例外。身份事实上是一个 Windows 账户，该账户的安全口令与进程关联，并且在默认情况下口令也会与进程中的所有线程关联。然而，使用哪个身份则是由应用程序的管理者所决定的。我们可以让宿主以交互式用户（Interactive User）的身份运行，所谓交互式用户也就是启动宿主进程的用户。如果简单地由某个用户启动宿主进程，那么宿主进程将使用该用户的账户作为其身份标识。交互式身份（Interactive Identity）一般用于自托管（Self-Hosting）场景，它也比较适合在调试时使用，因为当我们从 Visual Studio 中启动调试器时，调试器会自动地将自身附加到宿主进程中。不过，当服务被部署到一台服务器上时，再使用交互式身份就有点不切实际了，因此服务器并不一定存在登录用户，而且登录用户也不一定有启动服务所必需的凭证。在产品部署阶段，我们通常会依赖于一个指定的账户，该账户是一个主要为我们的服务所用的一个预先设定的 Windows 账户。若想以一个指定账户启动服务，我们可以使用 `Run as ...` 的 `Shell` 选项来启动该服务。不过，`Run as ...` 选项仅适用于简单的测试。在实际应用中可以使用一



个NT服务作为宿主,并通过控制面板中的服务管理程序为宿主分配一个指定的身份。如果服务寄宿在IIS6或WAS中,则可以使用那些环境配置工具为它分配一个指定的身份。

## IIdentity 接口

在.NET中, System.Security.Principal命名空间下的IIdentity接口代表了一个安全身份:

```
public interface IIdentity
{
    string AuthenticationType
    {get;}
    bool IsAuthenticated
    {get;}
    string Name
    {get;}
}
```

我们可以通过IIdentity接口获知它代表的身份是否通过了验证(以及采用了何种验证机制)以及身份的名称。WCF直接使用了由.NET提供的IIdentity的三种实现。WindowsIdentity类代表了一个Windows账户。GenericIdentity类则是一个通用类,我们主要利用它将一个身份的名称用一个IIdentity接口包装起来。对于GenericIdentity和WindowsIdentity两者而言,假如身份名称是一个空的字符串,那么该身份将被认为是未通过验证的,而任何其他长度大于0的名称都被认为是通过验证的。最后还有一个内部类X509Identity,它代表了一个使用X509证书通过验证的身份。X509Identity代表的身份总是通过验证的。

## 使用 WindowsIdentity

WindowsIdentity类在实现IIdentity接口的基础上提供了一些更有用的方法:

```
public class WindowsIdentity : IIdentity,...
{
    public WindowsIdentity(string sUserPrincipalName);
    public static WindowsIdentity GetAnonymous();
    public static WindowsIdentity GetCurrent();
    public virtual bool IsAnonymous
    {get;}
    public virtual bool IsAuthenticated
    {get;}
    public virtual string Name
    {get;}
    // 更多成员
}
```

Boolean属性IsAnonymous表明接口代表的身份是否为匿名的,而GetAnonymous()方

法则用于返回一个匿名的 Windows 身份，该身份一般用于模拟场景，以掩饰其真实身份：

```
WindowsIdentity identity = WindowsIdentity.GetAnonymous();
Debug.Assert(identity.Name == "");
Debug.Assert(identity.IsAuthenticated == false);
Debug.Assert(identity.IsAnonymous == true);
```

静态方法 `GetCurrent()` 返回其调用所处进程的身份。在默认情况下，该身份不是匿名的，而且还是通过验证的：

```
WindowsIdentity.currentIdentity = WindowsIdentity.GetCurrent();
Debug.Assert(currentIdentity.Name != "");
Debug.Assert(currentIdentity.IsAuthenticated == true);
Debug.Assert(currentIdentity.IsAnonymous == false);
```

## 安全调用上下文

在施加安全保护的 WCF 服务上的每个操作都有一个安全调用上下文。`ServiceSecurityContext` 类用于表示该上下文，定义如下：

```
public class ServiceSecurityContext
{
    public static ServiceSecurityContext Current
    {get;}
    public bool IsAnonymous
    {get;}
    public IIdentity PrimaryIdentity
    {get;}
    public WindowsIdentity WindowsIdentity
    {get;}
    // 更多成员
}
```

安全调用上下文主要用于实现定制的安全保护机制，以及对服务的分析与监管。尽管这里提到的只是安全调用上下文在局域网环境中的应用，但是在其他场景中也会用到它。

注意，从安全调用上下文的名称中我们就可以看出，它的作用对象是每个调用，而不是每个服务。安全调用上下文保存在 TLS 中，因此沿着服务的调用链向下，每个对象上的每个方法都可以访问安全调用上下文，包括服务的构造函数。只需访问 `Current` 静态属性，我们就可以获得当前的安全调用上下文。通过 `OperationContext` 类中的 `ServiceSecurityContext` 属性，我们也能便捷地获得当前的安全调用上下文：

```
public sealed class OperationContext : ...
{
    public ServiceSecurityContext ServiceSecurityContext
    {get;}
```

```
        // 更多成员  
    }
```

不管我们使用了哪种方法，都将得到相同的对象：

```
ServiceSecurityContext context1 = ServiceSecurityContext.Current;  
ServiceSecurityContext context2 = OperationContext.Current.ServiceSecurityContext;  
Debug.Assert(context1 == context2);
```

---

**警告：**只有在启动安全保护后，服务才会获得一个安全调用上下文。禁用安全保护时，`ServiceSecurityContext.Current`将返回 `null`。

---

`ServiceSecurityContext` 类中的 `PrimaryIdentity` 属性包含了调用链上最终客户端的身份。如果客户端未通过验证，`PrimaryIdentity` 将引用一个使用空白身份的 `IIdentity` 的实现。采用 Windows 验证时，`PrimaryIdentity` 属性将被设置为一个 `WindowsIdentity` 类型的实例。

`WindowsIdentity` 属性只在使用 Windows 验证时才有意义，而且该属性总是 `WindowsIdentity` 类型。当客户端提供了合法的 Windows 凭证时，`WindowsIdentity` 属性将包含相应的客户端身份，并与 `PrimaryIdentity` 属性中的值保持一致。

---

**注意：**单例服务的构造函数中不会包含一个安全调用上下文，因为它是在服务宿主启动时被调用的，而不是由客户端调用触发的。

---

## 模拟

一些资源，比如文件系统、SQL Server、网络套接字，甚至是 DCOM 对象，是根据调用者的安全口令来决定是否授权调用者访问的。通常，服务的宿主进程会被分配一个身份，在这个身份中会包含一个被升级的安全凭证，服务需要使用该凭证访问那些特殊资源以确保服务的正常运行。但是，相对于这些服务而言，客户端的安全凭证通常受到很大的限制。在传统方法中，开发人员采用模拟 (Impersonation) 技术弥补凭证上的差异。模拟允许服务首先假设采用客户端的身份，从而验证客户端是否有权执行它要求服务所做的工作。模拟对于在本节后面讨论的应用程序有许多关键的负面影响。我们应该应用基于角色的安全机制对调用者进行授权以代替使用模拟技术。不过，由于很多开发者习惯于采用模拟技术来设计系统，所以 .NET 与 WCF 都考虑到了对这种需求的支持。

## 手动模拟

服务可以通过调用 `WindowsIdentity` 类中的 `Impersonate()` 方法模拟调用它的客户端：

```
public class WindowsIdentity : IIdentity,...
{
    public virtual WindowsImpersonationContext Impersonate();
    // 更多成员
}
public class WindowsImpersonationContext : IDisposable
{
    public void Dispose();
    public void Undo();
}
```

`Impersonate()` 方法返回了一个 `WindowsImpersonationContext` 类型的实例，该实例包含了服务以前的身份。服务可以调用 `Undo()` 方法恢复以前的身份。为了进行模拟，服务需要在调用者的身份上调用 `Impersonate()` 方法，该身份可通过其安全调用上下文中的 `WindowsIdentity` 属性获得，如例 10-4 所示。

### 例 10-4：显式地模拟及恢复

```
class MyService : IMyContract
{
    public void MyMethod()
    {
        WindowsImpersonationContext impersonationContext =
            ServiceSecurityContext.Current.WindowsIdentity.Impersonate();
        try
        {
            /* 模拟执行客户端的工作 */
        }
        finally
        {
            impersonationContext.Undo();
        }
    }
}
```

注意，在例 10-4 中，服务将 `Undo()` 方法的调用放置在 `finally` 语句中，即便服务出现异常，它也能恢复原先的身份。为了简化恢复身份的工作，`WindowsImpersonationContext` 类在 `Dispose()` 方法的实现中也调用了 `Undo()` 方法，这使得我们可以在一个 `using` 语句块中方便地调用 `Impersonate()` 方法：

```
public void MyMethod()
{
    using (ServiceSecurityContext.Current.WindowsIdentity.Impersonate())
    {
        /* 模拟执行客户端的工作 */
    }
}
```

```
    }  
}
```

## 声明式模拟

我们也可以让 WCF 自动模拟方法的调用者以代替手动模拟。OperationBehavior 特性提供了 ImpersonationOption 枚举类型的 Impersonation 属性：

```
public enum ImpersonationOption  
{  
    NotAllowed,  
    Allowed,  
    Required  
}  
[AttributeUsage(AttributeTargets.Method)]  
public sealed class OperationBehaviorAttribute :  
    Attribute, IOperationBehavior  
{  
    public ImpersonationOption Impersonation  
    {get;set;}  
    // 更多成员  
}
```

ImpersonationOption.NowAllowed 是该属性的默认值。它表明 WCF 在默认情况下不会自动模拟的，但是，我们可以通过编写代码的形式显式地进行模拟（如例 10-4 所示）。

ImpersonationOption.Allowed 指示 WCF 在使用 Windows 验证时，自动地模拟调用者，但它对于其他的验证机制无效。在 WCF 采用自动模拟时，一旦方法返回，它也将自动恢复到服务之前的身份。

ImpersonationOption.Required 值强制要求使用 Windows 验证，否则会抛出一个异常。正如其名所示，WCF 在每个操作被调用时都会自动地进行身份模拟（并自动恢复）：

```
class MyService : IMyContract  
{  
    [OperationBehavior(Impersonation = ImpersonationOption.Required)]  
    public void MyMethod()  
    {  
        /* 模拟客户端执行工作 */  
    }  
}
```

注意，我们无法对服务的构造函数使用声明式模拟，因为我们无法在一个构造函数上应用 OperationBehavior 特性。构造函数只能采用手动模拟。如果我们在构造函数中确实需要用到模拟，那么总是在构造函数中同时恢复身份。这么做是为了避免对服务中的其他操作，甚至是相同宿主中的其他服务产生无法预料的影响。

## 模拟所有操作

假如我们需要在服务的所有操作中启用模拟技术, `ServiceHostBase` 类提供了一个 `ServiceAuthorizationBehavior` 类型的 `Authorization` 属性:

```
public abstract class ServiceHostBase : ...
{
    public ServiceAuthorizationBehavior Authorization
    {get;}
    // 更多成员
}
public sealed class ServiceAuthorizationBehavior : IServiceBehavior
{
    public bool ImpersonateCallerForAllOperations
    {get;set;}
    // 更多成员
}
```

`ServiceAuthorizationBehavior` 提供了 **Boolean** 属性 `ImpersonateCallerForAllOperations`, 它的默认值为 `false`。与其名称的含义相反, 当它被设置为 `true` 时, 该属性仅仅用于确保服务上没有操作被配置为 `ImpersonationOption.NotAllowed`。这个约束条件是在服务加载的时候进行验证的, 一旦条件不满足, 就会引发一个 `InvalidOperationException` 异常。

实际上, 在采用 Windows 验证时, WCF 就会在所有的操作中自动地进行模拟, 不过我们必须显式地将所有操作都设置为 `ImpersonationOption.Allowed` 或 `ImpersonationOption.Required`。`ImpersonateCallerForAllOperations` 对于构造函数而言是无效的。

我们可以通过编码方式或在配置文件中设置 `ImpersonateCallForAllOperations` 属性。如果采用编码方式, 则必须在打开宿主之前采用如下做法:

```
ServiceHost host = new ServiceHost(typeof(MyService));
host.Authorization.ImpersonateCallerForAllOperations = true;
host.Open();
```

当使用配置文件进行设置时, 我们需要在服务的声明中引用相应的服务行为:

```
<services>
  <service name = "MyService" behaviorConfiguration= "ImpersonateAll">
    ...
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name = "ImpersonateAll">
      <serviceAuthorization impersonateCallerForAllOperations = "true"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

```
</serviceBehaviors>
</behaviors>
```

为了省去在每个方法上应用 `OperationBehavior` 特性,可以在所有操作中自动进行模拟。在我编写的静态类 `SecurityHelper` 中,定义了 `ImpersonateAll()` 方法:

```
public static class SecurityHelper
{
    public static void ImpersonateAll(ServiceHostBase host);
    public static void ImpersonateAll(ServiceDescription description);
    // 更多成员
}
```

我们必须在打开宿主前调用 `ImpersonateAll()` 方法:

```
// 将模拟所有的操作
class MyService : IMyContract
{
    public void MyMethod()
    {...}
}
ServiceHost host = new ServiceHost(typeof(MyService));
SecurityHelper.ImpersonateAll(host);
host.Open();
```

例 10-5 中演示了 `ImpersonateAll()` 方法的实现。

#### 例 10-5: 实现 `SecurityHelper.ImpersonateAll()` 方法

```
public static class SecurityHelper
{
    public static void ImpersonateAll(ServiceHostBase host)
    {
        host.Authorization.ImpersonateCallerForAllOperations = true;
        ServiceDescription description = host.Description;
        ImpersonateAll(description);
    }
    public static void ImpersonateAll(ServiceDescription description)
    {
        foreach(ServiceEndpoint endpoint in description.Endpoints)
        {
            foreach(OperationDescription operation in endpoint.Contract.Operations)
            {
                foreach(IOperationBehavior behavior in operation.Behaviors)
                {
                    if(behavior is OperationBehaviorAttribute)
                    {
                        OperationBehaviorAttribute attribute =
                            behavior as OperationBehaviorAttribute;
                        attribute.Impersonation = ImpersonationOption.Required;
                        break;
                    }
                }
            }
        }
    }
}
```



```

    }
}
// 更多成员
}

```

在例 10-5 中, `ImpersonateAll()` 方法将参数提供的宿主的 `ImpersonateCaller-ForAllOperations` 属性设为 `true`, 然后它再从宿主中获得包含所有终结点的服务描述。该描述将被传递给一个重载版本的 `ImpersonateAll()` 方法, 该方法通过遍历服务描述中的终结点集合, 把所有的操作配置为 `ImpersonateOption.Required`。对于每个终结点而言, `ImpersonateAll()` 方法会访问每个终结点中的所有契约的操作集合。每个操作都可能有一个或多个实现 `IOperationBehavior` 接口的操作行为 (即使我们没有显式地提供一个操作行为, 也至少存在一个 WCF 提供的操作行为 `OperationBehaviorAttribute`)。每个操作行为都会被检查, 直至找到 `OperationBehaviorAttribute`。一旦找到该特性, 它的 `Impersonate` 属性就会被设置为 `Impersonate.Required`。

我们可以进一步在 `ServiceHost<T>` 中封装以及简化对 `SecurityHelper.ImpersonateAll()` 方法的使用:

```

public class ServiceHost<T> : ServiceHost
{
    public void ImpersonateAll()
    {
        if(State == CommunicationState.Opened)
        {
            throw new InvalidOperationException("Host is already opened");
        }

        SecurityHelper.ImpersonateAll(this);
    }
    // 更多成员
}

```

使用 `ServiceHost<T>` 就可以自动实现对所有调用者的模拟:

```

ServiceHost<MyService> host = new ServiceHost<MyService>();
host.ImpersonateAll();
host.Open();

```

## 限制模拟

授权与验证保护了服务免受那些来自未经授权和未经验证的可能有恶意的客户端的访问所带来的损害。但是, 应该如何让客户端避免有恶意的服务的危害呢? 恶意服务滥用客户端的其中一种方式就是通过假冒客户端的身份及凭证的方式, 对系统造成一定的损害。

在某些情况下,客户端甚至有可能根本不希望服务能够获得它的身份。WCF允许客户端指定在什么模拟级别上服务可以获得客户端的身份并使用它。模拟事实上包含了一个选择范围,该范围指定了客户端与服务间的信任等级。WindowsClientCredential类提供了TokenImpersonationLevel枚举类型的AllowedImpersonationLevel属性,该枚举类型位于System.Security.Principal命名空间:

```
public enum TokenImpersonationLevel
{
    None,
    Anonymous,
    Identification,
    Impersonation,
    Delegation
}
public sealed class WindowsClientCredential
{
    public TokenImpersonationLevel AllowedImpersonationLevel
    {get;set;}
    // 更多成员
}
```

客户端既可以通过编码方式,也可以通过管理方式限制允许的模拟级别。例如,要想采用编码方式将模拟级别限制为TokenImpersonationLevel.Identification,我们可以在打开代理之前在客户端编写如下代码:

```
MyContractClient proxy = new MyContractClient();
proxy.ClientCredentials.Windows.AllowedImpersonationLevel =
    TokenImpersonationLevel.Identification;
proxy.MyMethod();
proxy.Close();
```

使用配置文件时,管理者应该将允许的模拟等级定义为一个定制的终结点行为,并在相关的终结点配置节中引用它:

```
<client>
  <endpoint behaviorConfiguration = "ImpersonationBehavior"
    ...
  />
</client>
<behaviors>
  <endpointBehaviors>
    <behavior name = "ImpersonationBehavior">
      <clientCredentials>
        <windows allowedImpersonationLevel = "Identification"/>
      </clientCredentials>
    </behavior>
  </endpointBehaviors>
</behaviors>
```

`TokenImpersonationLevel.None`简单地表示没有指定模拟的级别,同时也意味着客户端没有使用安全凭证。当模拟级别被设置为`TokenImpersonationLevel.Anonymous`时,客户端也不会提供任何凭证。在实际应用中,`TokenImpersonationLevel.None`和`TokenImpersonationLevel.Anonymous`拥有同样的表现,即客户端不提供身份信息。从客户端的角度来看,这两种值无疑是最安全的,但是从应用程序的角度来看,它们是实用性最差的选项,因为服务无法对客户端做出有效的验证或授权。只有在服务被配置为匿名访问或没有安全要求时,才可能不使用安全凭证,而这显然并不适用于局域网场景。如果服务被配置为Windows安全模式,客户端使用这两种值会引发`ArgumentOutOfRangeException`异常。

服务使用`TokenImpersonationLevel.Identification`级别可以识别出客户端的身份,即获取发起调用的客户端的安全身份。不过,此时服务是不允许模拟客户端的,服务所做的每件事都必须使用自己的身份完成。任何模拟的尝试都会在服务端抛出一个`ArgumentOutOfRangeException`异常。`TokenImpersonationLevel.Identification`是Windows安全模式所使用的默认值,同时,它也是局域网场景中推荐使用的模拟级别。注意,假如服务与客户端处在同一台机器上,那么就算配置为`TokenImpersonationLevel.Identification`,服务还是可以模拟客户端。

`TokenImpersonationLevel.Impersonation`不仅允许服务获得客户端的身份,同时还准许它模拟客户端。模拟表明了客户端与服务之间彼此信任的程度相当高,因为即使服务宿主被配置为使用一个权限较小的身份,服务也可以做任何客户端可以做的事。客户端与模拟它的服务之间唯一的区别在于,当服务与客户端不在同一台机器上时,它将无法像客户端一样访问位于另一台机器上的资源或对象,因为服务所在的机器无法真的获得客户端的密码。当服务与客户端处在同一台机器上时,模拟客户端的服务则可以访问另一台只需要一个网络跳转的机器,因为服务所在的机器能够验证并模拟客户端身份。

最后,`TokenImpersonationLevel.Delegation`为服务提供了客户端的Kerberos票据。此时,服务可以像客户端那样自由访问任何机器上的资源。如果服务同时被设置为使用委托(Delegation),那么当它调用其他下游服务时,客户端身份将沿着调用链不断地传播。需要委托的Kerberos验证无法应用于Windows工作组安装模式。由于会在客户端与服务间建立起非常高的信任关系(以及因此存在的安全风险),客户端与服务两者的用户账户都必须在活动目录(Active Directory)中被正确地配置以支持委托。默认情况下,委托会使用另一种名为Cloaking的安全服务,该服务将沿着调用链传播客户端的身份。

从客户端的角度来看,使用委托极其危险,因为客户端无法控制它的身份被谁或在什么地方使用。当模拟级别被设置为`TokenImpersonationLevel.Impersonation`时,客户端承担的风险是可控的,因为它知道自己正在访问哪个服务,并且,如果这些服务位

于不同的机器上，客户端身份是不会跨越网络传播的。我们认为应该把委托当作一种使得服务可以充当客户端替代者的手段，而不仅仅是对客户端进行模拟。

## 使用模拟

我们应该将服务设计为不依赖模拟，此时，客户端应该使用 `TokenImpersonation-Level.Identification` 选项。

客户端的身份离客户端越远，则它的相关度也就越低，这是一条通用的设计规范。如果我们在自己的系统设计中使用了某种分层的方法，那么每层都应该以它自己的身份运行，对它的直接调用者进行验证，并隐式地信任调用它的上层对初始调用者所做的验证，以此建立起一个值得信任的并经过验证的调用者链。而模拟却要求我们不断地沿着调用链传播客户端的身份，直到底层的资源。这么做会降低系统的可伸缩性，因为此时会针对每个身份分配很多资源（例如SQL SERVER连接）。使用模拟时，我们需要的资源会随着客户端的增多而不断增加，这样也就无法从资源池（比如连接池）中获益。模拟还增加了管理资源的复杂性，由于我们需要将资源的访问权授予最初的客户端身份，从而导致大量的身份需要管理。不管访问服务的各种身份有多少，一个总是以它自己的身份运行的服务是不存在以上问题的。我们应该对资源的访问控制使用授权（Authorization）技术，这会在接下来的内容中介绍。最后，使用模拟技术并不适合使用Windows验证之外的验证模式。如果我们决定使用模拟技术，就应该正确的使用它，并且仅在我们没有其他或更好的设计方法时才使用它。

---

**注意：**模拟不能用于队列服务。

---

## 授权

我们使用身份验证（Authentication）技术确认客户端的身份是否与它声称的一致，而大多数应用程序还需要确认客户端（更准确地说是它指向的身份）是否有权执行相应的操作。为每个单独的身份设计访问权限是不切实际的，更好的做法是将权限授予客户端在应用程序域中所扮演的角色。一个角色就是一类身份的象征，这些身份拥有相同的安全权限。当我们为某个应用程序资源分配了一个角色时，我们就向角色中的所有成员授予了访问该资源的权限。就像分解服务与接口那样，在分析与设计应用程序需求时，也应该辨别客户端在业务领域中所扮演的角色。通过与角色而不是特定的身份进行交互，我们就可以使得应用程序不受到现实生活中变化的影响。例如加入新用户，交换现有用户的职务，用户升职或离职。如果确认用户角色是一个动态过程，.NET允许我们以声明或编码两种方式应用基于角色的安全技术。

## 安全主体

出于安全考虑,将身份与角色成员的信息放在一起是比较合适的一种选择。这种身份表现形式被称为安全主体。

.NET 中的安全主体可以是任何实现了 `IPrincipal` 接口的对象,该接口被定义在 `System.Security.Principal` 命名空间中:

```
public interface IPrincipal
{
    IIdentity Identity
    {get;}
    bool IsInRole(string role);
}
```

如果与该安全主体关联的身份是某个指定角色的成员, `IsInRole()` 方法将返回 `true`, 否则返回 `false`。只读类型的 `Identity` 属性作为一个实现 `IIdentity` 接口的对象,提供了与身份相关的只读信息的访问。.NET 已经提供了多个 `IPrincipal` 的实现。`GenericPrincipal` 是一个一般主体,使用它时必须预先配置相关的角色信息。它通常在不需要用到授权技术的时候使用,在这种情况下, `GenericPrincipal` 包装了一个空白身份。`WindowsPrincipal` 类则会在 Windows NT 组中查询角色成员的信息。

每个 .NET 的线程都有一个与之相关的安全主体对象,该对象可以通过 `Thread` 类的 `CurrentPrincipal` 静态属性获得:

```
public sealed class Thread
{
    public static IPrincipal CurrentPrincipal
    {get;set;}
    // 更多成员
}
```

例如,下面的代码演示了如何获得用户名,并确认该调用者是否通过验证:

```
IPrincipal principal = Thread.CurrentPrincipal;
string userName = principal.Identity.Name;
bool isAuthenticated = principal.Identity.IsAuthenticated;
```

## 选择授权模式

正如之前所提到的, `ServiceHostBase` 类提供了 `ServiceAuthorizationBehavior` 类型的 `Authorization` 属性。`ServiceAuthorizationBehavior` 类型中又定义了一个 `PrincipalPermissionMode` 枚举类型的 `PrincipalPermissionMode` 属性,定义如下:

```
public enum PrincipalPermissionMode
{
```

```
        None,  
        UseWindowsGroups,  
        UseAspNetRoles,  
        Custom  
    }  
    public sealed class ServiceAuthorizationBehavior : IServiceBehavior  
    {  
        public PrincipalPermissionMode PrincipalPermissionMode  
        {get;set;}  
        // 更多成员  
    }  
}
```

在打开宿主前，我们可以通过 `PrincipalPermissionMode` 属性选择主体模式，也就是说，安装某种类型的安全主体向调用者进行授权。

如果 `PrincipalPermissionMode` 被设置为 `PrincipalPermissionMode.None`，那么基于安全主体的授权就是可行的。在对调用者进行验证之后（如果需要验证），WCF 会为 `GenericPrincipal` 配置一个空白的身份，并把它附加到调用服务操作的线程上。该安全主体可以通过 `Thread.CurrentPrincipal` 获得。

当 `PrincipalPermissionMode` 被设置为 `PrincipalPermissionMode.UseWindowsGroups` 时，WCF 将会为 `WindowsPrincipal` 配置一个与提供的凭证相匹配的身份。如果没有发生任何 Windows 验证（因为服务没有要求这么做），那么 WCF 将为 `WindowsPrincipal` 配置一个空白的身份。

`PrincipalPermissionMode.UseWindowsGroups` 是 `PrincipalPermissionMode` 属性的默认值。因此，以下两种定义是等效的：

```
ServiceHost host1 = new ServiceHost(typeof(MyService));  
  
ServiceHost host2 = new ServiceHost(typeof(MyService));  
host2.Authorization.PrincipalPermissionMode =  
    PrincipalPermissionMode.UseWindowsGroups;
```

在使用配置文件时，我们需要引用一个定制行为配置节用于指定安全主体模式：

```
<services>  
  <service name = "MyService" behaviorConfiguration = "WindowsGroups">  
    ...  
  </service>  
</services>  
<behaviors>  
  <serviceBehaviors>  
    <behavior name = "WindowsGroups">  
      <serviceAuthorization principalPermissionMode = "UseWindowsGroups"/>  
    </behavior>  
  </serviceBehaviors>  
</behaviors>
```



## 声明方式实现基于角色的安全

我们使用 `PrincipalPermissionAttribute` 特性在服务端应用声明式的基于角色的安全，该特性定义在 `System.Security.Permissions` 命名空间中：

```
public enum SecurityAction
{
    Demand,
    // 更多成员
}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public sealed class PrincipalPermissionAttribute : CodeAccessSecurityAttribute
{
    public PrincipalPermissionAttribute(SecurityAction action);

    public bool Authenticated
    {get;set;}
    public string Name
    {get;set;}
    public string Role
    {get;set;}
    // 更多成员
}
```

`PrincipalPermission` 特性用于声明所需的角色成员。对于局域网场景而言，当我们指定某个 Windows NT 组作为一个角色时，我们必须以域名或本地机器名（假如该角色被定义为只在本地使用）作为其前缀。在例 10-6 中，`PrincipalPermission` 特性的声明仅授权管理员用户组中的调用者访问 `MyMethod()` 方法。

### 例 10-6：局域网中的声明式基于角色的安全

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}

class MyService : IMyContract
{
    [PrincipalPermission(SecurityAction.Demand, Role = @"<domain>\Managers")]
    public void MyMethod()
    {...}
}
```

如果用户不是该角色中的成员，.NET 将抛出一个 `SecurityException` 类型的异常。

---

**注意：**对 Windows 基于角色的安全进行试验时，我们常常会向用户组中添加或删除用户。由于在登录时，Windows 对用户组的信息做了缓存，我们所做的改变只有在下次登录时才能体现出来。

---



如果允许多个角色访问某个方法，我们可以多次应用 `PrincipalPermission` 特性：

```
[PrincipalPermission(SecurityAction.Demand,Role=@"<domain>\Managers")]
[PrincipalPermission(SecurityAction.Demand,Role=@"<domain>\Customers")]
public void MyMethod()
{...}
```

在使用了多个 `PrincipalPermission` 特性时，.NET 将验证调用者至少是某个所要求的角色中的一个成员。如果我们希望验证该用户同时是两个角色中的成员，则需要编写代码完成检查，这种方法将在稍后介绍。

尽管根据它的定义，该特性完全可以被应用于方法及类上，但是在一个 WCF 服务类中，我们只能将它应用在方法上。因为在 WCF 中（不同于一般的类），无论采用何种验证机制，服务类的构造函数总是在一个使用空白身份的 `GenericPrincipal` 主体下执行。因此，构造函数运行时所用的身份是未经验证的，任何类型的授权尝试都将失败（就算没有使用 Windows NT 工作组）：

```
// 总是会失败
[PrincipalPermission(SecurityAction.Demand,Role = "...")]
class MyService : IMyContract
{...}
```

---

**警告：** 在服务的构造函数中，应尽量避免用到那些需要授权的敏感内容。在会话服务中，可以在操作中执行这些工作，如果要初始化实例以及对调用者授权，可以定义一个专门的 `Initialize()` 操作。

---

通过设置 `PrincipalPermission` 特性的 `Name` 属性，我们甚至可以仅授权某个特定用户访问方法。然而，这种做法并不可取，因为对用户名进行硬编码并不是好的做法：

```
[PrincipalPermission(SecurityAction.Demand,Name = "John")]
```

我们还可以在指定某个特定用户的基础上，进一步限定该用户是某个角色的成员：

```
[PrincipalPermission(SecurityAction.Demand,Name = "John",
    Role = @"<domain>\Managers")]
```

---

**注意：** 声明式基于角色的安全使用硬编码引用角色的名称。如果应用程序需要动态地查找角色的名称，我们就必须采用下面将要介绍的编码方式实现角色验证。

---

## 编码方式实现基于角色的安全

有些时候我们需要用编码方式来验证角色成员。通常，当我们在决定是否授权访问时，

需要同时判断角色成员以及某些在运行时才能获得的值（例如参数值、一天中的时间或地点），此时，我们会采用这种方式。当我们处理用户组本地化时，也会采用这种方式。下面举例说明第一种情况。试想某个银行服务，该服务允许客户端在两个指定的账户之间转移一定量的货币。只有客户和出纳是被允许调用 `TransferMoney()` 操作的，而且必须遵守以下的业务规则：如果转账的数额大于 5000，则只有出纳才有资格执行该项交易。如果采用声明方式实现基于角色的安全，可以验证调用者是出纳还是客户，但它无法约束附加的业务规则。为此，我们需要使用 `IPrincipal` 中 `IsInRole()` 方法，如例 10-7 所示。

例 10-7：编码方式实现基于角色的安全

```
[ServiceContract]
interface IBankAccounts
{
    [OperationContract]
    void TransferMoney(double sum, long sourceAccount, long destinationAccount);
}
static class AppRoles
{
    public const string Customers = @"<domain>\Customers";
    public const string Tellers = @"<domain>\Tellers";
}
class BankService : IBankAccounts
{
    [PrincipalPermission(SecurityAction.Demand, Role = AppRoles.Customers)]
    [PrincipalPermission(SecurityAction.Demand, Role = AppRoles.Tellers)]
    public void TransferMoney(double sum, long sourceAccount, long destinationAccount)
    {
        IPrincipal principal = Thread.CurrentPrincipal;
        Debug.Assert(principal.Identity.IsAuthenticated);

        bool isCustomer = principal.IsInRole(AppRoles.Customers);
        bool isTeller = principal.IsInRole(AppRoles.Tellers);

        if(isCustomer && ! isTeller)
        {
            if(sum > 5000)
            {
                string message = "Caller does not have sufficient authority to" +
                                "transfer this sum";
                throw(new UnauthorizedAccessException(message));
            }
        }
        DoTransfer(sum, sourceAccount, destinationAccount);
    }
    // 辅助方法
    void DoTransfer(double sum, long sourceAccount, long destinationAccount)
    { ... }
}
```

例 10-7 还说明了许多其他的问题。首先，尽管它使用 `sum` 变量的值实现编码方式的角

色成员验证,但它仍然将声明方式实现基于角色的安全作为第一道防线,这样,就只有那些客户或出纳角色中的成员才能够访问服务。然后,我们可以通过编码方式,使用 `IIIdentity` 的 `IsAuthenticated` 属性确保服务的调用者已通过身份验证。最后,注意 `AppRoles` 静态类的使用,它封装了代表角色的实际字符串,这样就可以避免在多个地方硬编码角色字符串。与此类似,我们需要采用编码方式实现基于角色的安全,以此来约束其他附加的业务规则,如转账操作只允许发生在正常的营业时间内。

**注意:** 基于角色的安全不会与具体的安全主体类型交互。当 `PrincipalPermission` 特性用于验证角色成员时,它只是简单获取当时在线程中以 `IPrincipal` 形式出现的安全主体,并调用它的 `IsInRole()` 方法。同时,以编码方式使用 `IPrincipal` 实现角色成员的验证,也采用了相同的做法,如例 10-7 所示。将 `IPrincipal` 接口从它的实现中分离出来,对于提供除 Windows NT 工作组以外的其他基于角色的安全机制是十分关键的,我们将会在其他场景中看到这一点。

## Windows 角色的本地化

以声明方式实现基于角色的安全,需要对角色名称硬编码。假如我们的应用程序是面向国际市场的,并且使用 Windows 工作组作为角色,那么角色的名称很有可能不匹配。在局域网场景中,附加到访问服务的线程上的安全主体对象是 `WindowsPrincipal` 类型的:

```
public class WindowsPrincipal : IPrincipal
{
    public WindowsPrincipal(WindowsIdentity ntIdentity);

    //IPrincipal 实现
    public virtual IIIdentity Identity
    {get;}
    public virtual bool IsInRole(string role);

    // 另外的方法:
    public virtual bool IsInRole(int rid);
    public virtual bool IsInRole(WindowsBuiltInRole role);
}
```

`WindowsPrincipal` 类提供了两个额外的 `IsInRole()` 方法,以简化 Windows NT 工作组的本地化任务。我们可以向其中一个 `IsInRole()` 方法提供一个 `WindowsBuiltInRole` 类型的枚举值,该值将与内建的 NT 角色(如 `WindowsBuiltInRole.Administrator` 或 `WindowsBuiltInRole.User`)匹配。另一个 `IsInRole()` 方法则接收一个用于表明指定角色的整数索引。例如,512 号角色索引将映射为 `Administrators` 组。MSDN 库中包含了一个预先定义好的索引列表,并演示了如何为用户组指定属于自己的别名与索引。

## 身份管理

在局域网场景中，成功完成验证之后，WCF 将向操作线程附加一个 `WindowsIdentity` 类型的安全主体身份，客户端提供的用户名（或 Windows 账户）将被设为该身份的 `Name` 属性值。因为客户端提供了有效的安全凭证，安全调用上下文的两种身份——基本身份（Primary Identity）及 Windows 身份将被设为同一个身份——安全主体身份。所有的三种身份都将被认为是通过验证的。表 10-4 演示了这些身份以及它们的值。

表 10-4：局域网场景中的身份管理

身份	类型	值	是否验证
Thread principal	<code>WindowsIdentity</code>	<code>Username</code>	Yes
Security context primary	<code>WindowsIdentity</code>	<code>Username</code>	Yes
Security context Windows	<code>WindowsIdentity</code>	<code>Username</code>	Yes

需要注意的是，尽管宿主进程（以及线程令牌 `Thread Token`）保留着它们最初被指定的身份，然而安全主体身份其实是调用者的身份。我们将这个行为称为软模拟（Soft Impersonation），而且在与基于角色的安全一起使用时，我们基本上不再需要用到真正的模拟，因为它会使用客户端令牌代替宿主的安全令牌。

## 回调

涉及到局域网中的安全时，回调与正常的服务操作相比，有几个显著的区别。首先，对于一个回调契约，我们只能在操作层面设定保护级别，而无法在整个回调契约的层面上设定保护级别。例如，下面这个对保护级别的约束将会被忽略：

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{...}
// 要求保护级别的命令会被忽略
[ServiceContract(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
interface IMyContractCallback
{...}
```

我们只能在操作层面设定保护级别：

```
[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{...}

interface IMyContractCallback
{
    [OperationContract(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
    void OnCallback();
}
```

所有发生在回调对象上的调用都伴随着一个未经验证的安全主体,即使所有情况下都使用 Windows 安全来调用服务也是如此。因此,安全主体身份将被设为带有一个空白身份的 Windows 身份,这就使得授权以及基于角色的安全都无法发挥作用。

虽然回调的确有一个安全调用上下文,不过由于 Windows 身份被设为带有一个空白身份的 `WindowsIdentity` 实例,模拟技术也就失效了。唯一有意义的信息在基本身份 (Primary Identity) 中,它将被设置为服务宿主进程的身份与机器名:

```
class MyClient : IMyContractCallback
{
    public void OnCallback()
    {
        IPrincipal principal = Thread.CurrentPrincipal;
        Debug.Assert(principal.Identity.IsAuthenticated == false);

        ServiceSecurityContext context = ServiceSecurityContext.Current;
        Debug.Assert(context.PrimaryIdentity.Name == "MyHost/localhost");

        Debug.Assert(context.IsAnonymous == false);
    }
}
```

由于无法便捷地运用基于角色的安全技术,因此我建议尽量避免在回调中执行任何敏感的工作。

## 互联网应用程序

在互联网场景中,客户端或服务可能不会使用 WCF,甚至不使用 Windows 平台。如果我们正在编写一个服务或者一个客户端,并不能假设另一端也使用了 WCF。此外,在一个互联网应用中通常都会有相对较多的客户端调用服务。这些客户端调用来源于防火墙之外。我们需要依靠 HTTP 进行传输,而且可能会有多个中继 (Intermediarie)。在一个互联网应用程序中,我们一般不希望使用 Windows 账户与工作组充当凭证,而是让应用程序访问一些定制的凭证库 (Credential Store)。也就是说,我们仍然可以使用 Windows 安全,在后面会介绍到这一内容。

## 保证互联网绑定的安全

在互联网应用程序中,我们必须使用 Message Security 模式实现传输安全,以提供穿越所有中继的端到端的安全。客户端应该以用户名和密码的形式提供凭证。我们应该在互联网场景中使用 `WSHttpBinding` 与 `WSDualHttpBinding`。此外,如果我们有一个采用 `NetTcpBinding` 的局域网应用程序,但又不希望使用 Windows 安全管理用户账户及工作组,那么我们就应该遵循与基于 WS 的绑定相同的配置。在 Message Security 传输

安全模式下, 选择 `MessageCredentialType.UserName` 类型的客户凭证就可以在所有相关绑定中完成安全配置。我们需要同时在客户端和服务两端对绑定进行配制。

## 配置 WSHttpBinding

`WSHttpBinding` 定义了 `WSHttpSecurity` 类型的 `Security` 属性:

```
public class WSHttpBinding : WSHttpBindingBase
{
    public WSHttpBinding();
    public WSHttpBinding(SecurityMode SecurityMode);
    public WSHttpSecurity Security
    {get;}
    // 更多成员
}
```

对于 `WSHttpSecurity`, 我们需要将 `SecurityMode` 类型的 `Mode` 属性设置为 `SecurityMode.Message`。如此, `WSHttpSecurity` 的 `Message` 属性才能够生效:

```
public sealed class WSHttpSecurity
{
    public SecurityMode Mode
    {get;set;}
    public NonDualMessageSecurityOverHttp Message
    {get;}
    public HttpTransportSecurity Transport
    {get;}
}
```

`Message` 属性的类型为 `NonDualMessageSecurityOverHttp`, 该类型继承自 `MessageSecurityOverHttp`:

```
public class MessageSecurityOverHttp
{
    public MessageCredentialType ClientCredentialType
    {get;set;}
    // 更多成员
}
public sealed class NonDualMessageSecurityOverHttp :
    MessageSecurityOverHttp
{...}
```

我们需要将 `MessageSecurityOverHttp` 类型中的 `ClientCredentialType` 属性设置为 `MessageCredentialType.Username`。我们可以回想一下, `WSHttpBinding` 默认的消息凭证为 `Windows` 类型 (见表 10-3)。

由于 `WSHttpBinding` 默认的传输安全模式是 `Message Security` (见表 10-1), 所以以下三种定义是等效的:



```
WSHttpBinding binding1 = new WSHttpBinding();
binding1.Security.Message.ClientCredentialType = MessageCredentialType.UserName;

WSHttpBinding binding2 = new WSHttpBinding(SecurityMode.Message);
binding2.Security.Message.ClientCredentialType = MessageCredentialType.UserName;

WSHttpBinding binding3 = new WSHttpBinding();
binding3.Security.Mode = SecurityMode.Message;
binding3.Security.Message.ClientCredentialType =
MessageCredentialType.UserName;
```

或采用以下配置文件：

```
<bindings>
  <wsHttpBinding>
    <binding name = "UserNameWS">
      <Security mode = "Message">
        <message clientCredentialType = "UserName"/>
      </Security>
    </binding>
  </wsHttpBinding>
</bindings>
```

又因为 Message Security 模式是传输安全模式的默认值，我们在配置文件中可以略去对传输安全模式的设定：

```
<bindings>
  <wsHttpBinding>
    <binding name = "UserNameWS">
      <Security>
        <message clientCredentialType = "UserName"/>
      </Security>
    </binding>
  </wsHttpBinding>
</bindings>
```

图 10-4 演示了 WSHttpBinding 中与安全相关的元素。

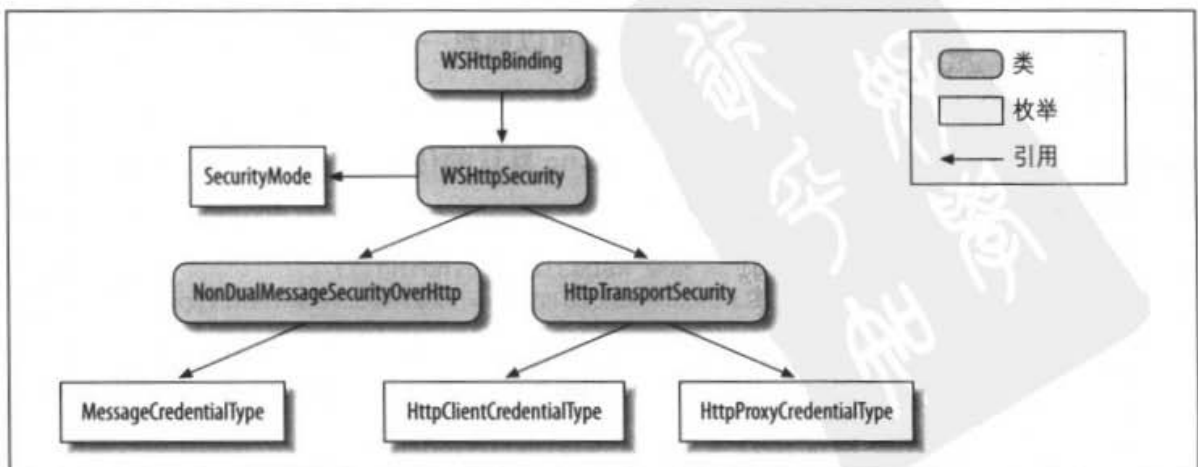


图 10-4：WSHttpBinding 与安全



WSHttpBinding 包含了一个指向 WSHttpSecurity 的引用, 该引用使用 SecurityMode 枚举类型表示传输安全模式。使用 Transport Security 模式时, WSHttpSecurity 会使用一个 HttpTransportSecurity 类型的实例; 使用 Message Security 模式时, WSHttpSecurity 会用到一个 NonDualMessageSecurityOverHttp 类型的实例, 该实例通过 MessageCredentialType 枚举类型指定了客户凭证类型。

## 配置 WSDualHttpBinding

WSDualHttpBinding 定义了 WSDualHttpSecurity 类型的 Security 属性:

```
public class WSDualHttpBinding : Binding,...
{
    public WSDualHttpBinding();
    public WSDualHttpBinding(WSDualHttpSecurityMode SecurityMode);
    public WSDualHttpSecurity Security
    {get;}
    // 更多成员
}
```

对于 WSDualHttpSecurity, 我们需要将 WSDualHttpSecurityMode 类型中的 Mode 属性设置为 WSDualHttpSecurityMode.Message。之后, WSDualHttpSecurity 的 Message 属性将开始生效:

```
public sealed class WSDualHttpSecurity
{
    public MessageSecurityOverHttp Message
    {get;}
    public WSDualHttpSecurityMode Mode
    {set;set;}
}
```

Message 属性的类型为前面提到的 MessageSecurityOverHttp。

我们需要将 MessageSecurityOverHttp 类型中的 ClientCredentialType 属性设为 MessageCredentialType.Username。我们可以回想一下, WSDualHttpBinding 默认的消息凭证类型为 Windows (参见表 10-3)。

由于 Message Security 是 WSDualHttpBinding 默认的传输安全模式 (见表 10-1), 所以以下三种定义是等效的:

```
WSDualHttpBinding binding1 = new WSDualHttpBinding();
binding1.Security.Message.ClientCredentialType = MessageCredentialType.Username;

WSDualHttpBinding binding2 = new WSDualHttpBinding(WSDualHttpSecurityMode.Message);
binding2.Security.Message.ClientCredentialType = MessageCredentialType.Username;

WSDualHttpBinding binding3 = new WSDualHttpBinding();
binding3.Security.Mode = WSDualHttpSecurityMode.Message;
binding3.Security.Message.ClientCredentialType = MessageCredentialType.Username;
```

或采用以下配置文件：

```
<bindings>
  <wsDualHttpBinding>
    <binding name = "WSDualWindowsSecurity">
      <Security mode = "Message">
        <message clientCredentialType = "UserName"/>
      </Security>
    </binding>
  </wsDualHttpBinding>
</bindings>
```

又因为 Message Security 模式是传输安全模式的默认值，我们在配置文件中可以略去对传输安全模式的设定：

```
<bindings>
  <wsDualHttpBinding>
    <binding name = "WSDualWindowsSecurity">
      <Security>
        <message clientCredentialType = "UserName"/>
      </Security>
    </binding>
  </wsDualHttpBinding>
</bindings>
```

图 10-5 演示了 WSDualHttpBinding 中与安全相关的元素。

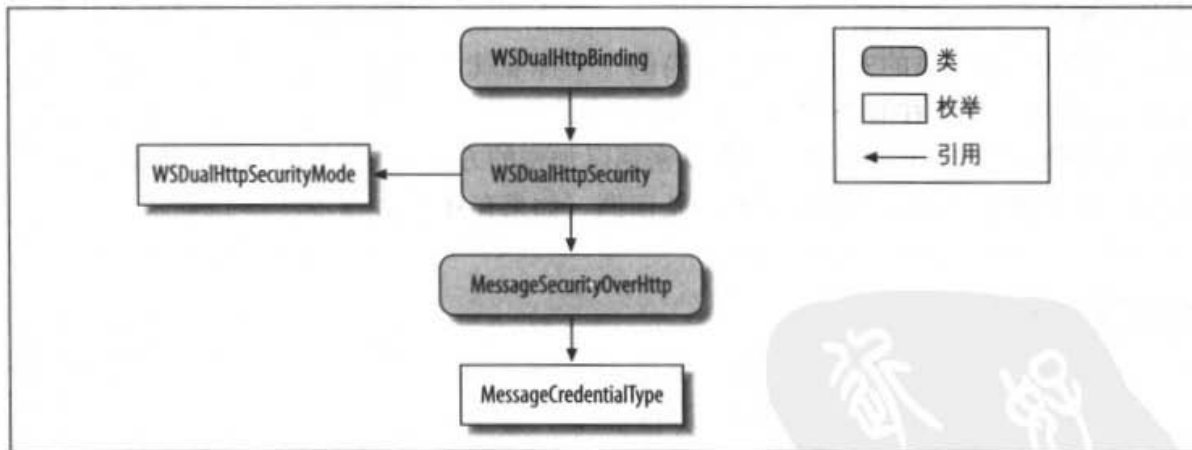


图 10-5: WSDualHttpBinding 与安全

WSDualHttpBinding 包含了一个指向 WSDualHttpSecurity 的引用，该引用使用 WSDualHttpSecurityMode 枚举类型表示传输安全模式：Message 或 None。当使用 Message Security 模式时，WSDualHttpSecurity 将使用一个 MessageSecurityOverHttp 类型的实例，该实例通过 MessageCredentialType 枚举类型指定了客户凭证类型。

## 消息保护

由于在互联网场景中,发送到服务的客户端消息是通过普通的HTTP协议传输的,因此,为它的内容(同时包括客户凭证与消息体)提供加密保护就显得非常重要。加密能够保证消息的完整性与机密性。使用客户端的密码对消息内容加密是一种选择。不过 WCF 从不使用客户端密码加密消息,原因如下:第一,无法保证该密码的强度足够高,以使任何监听该通信的人都无法破解加密。第二,它要求服务(更确切的说是服务的宿主)必须能够获得客户端的密码,这样就使得宿主与凭证库产生不必要的联系。最后,尽管密码可以对消息加以保护,但是它无法向客户端证明服务的真实性。

WCF 使用 X509 证书保护消息。该证书提供了高强度的保护,并向客户端证明它独一无二的真实性。证书采用两种密钥以及一个公用名(Common Name, CN, 如 MyServiceCert)实现上述功能,这两种密钥分别被称为公钥和私钥。它们的关键特性在于,以公钥加密的任何内容都只能由私钥解密。证书包含了公钥和公用名,而私钥则被保存在宿主机上的某个宿主能够访问到的安全区域。宿主将证书设为可被公共访问(以及它的公钥),这样,任何客户端都可以访问宿主的终结点以获取公钥。

简而言之,在一个调用过程中,位于客户端的 WCF 框架将使用公钥加密发往服务的所有消息。在接收到经过加密的消息后,处于宿主一端的 WCF 框架将使用私钥解密消息。一旦消息被解密, WCF 将从消息中读取客户凭证,验证客户端身份,从而判断是否允许它访问服务。真实的场景可能会更复杂些,因为 WCF 还需要确保回复消息以及由服务到客户端的回调消息的安全。WCF 支持的一个标准就是用于建立这样一个安全会话的。事实上,在客户端发往服务的第一个请求消息前就已经完成了多个调用。在这些调用中,客户端会将一个 WCF 生成的一个共享密钥以加密的方式(采用服务证书)传递给服务。此后,服务将用它来加密回复消息以及回调(如果存在)。

## 配置宿主证书

ServiceHostBase 类提供了类型为 ServiceCredentials 的 Credentials 属性, ServiceCredentials 是一个服务行为:

```
public abstract class ServiceHostBase : ...
{
    public ServiceCredentials Credentials
    {get;}
    // 更多成员
}
public class ServiceCredentials : ..., IServiceBehavior
{
    public X509CertificateRecipientServiceCredential ServiceCertificate
    {get;}
    // 更多成员
}
```

ServiceCredentials定义了类型为X509CertificateRecipientServiceCredential的ServiceCertificate属性:

```
public sealed class X509CertificateRecipientServiceCredential
{
    public void SetCertificate(StoreLocation storeLocation,
                             StoreName storeName,
                             X509FindType findType,
                             object findValue);

    // 更多成员
}
```

我们可以使用 SetCertificate() 方法告知 WCF 加载服务证书的位置与方式。通常, 我们在宿主配置文件的 serviceCredentials 配置节下, 使用一个定制行为向 WCF 提供以上信息, 如例 10-8 所示。

#### 例 10-8: 配置服务证书

```
<services>
  <service name = "MyService" behaviorConfiguration = "Internet">
    ...
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name = "Internet">
      <serviceCredentials>
        <serviceCertificate
          findValue      = "MyServiceCert"
          storeLocation  = "LocalMachine"
          storeName      = "My"
          x509FindType   = "FindBySubjectName"
        />
      </serviceCredentials>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

#### 使用宿主证书

客户端的开发人员可以使用任意一种带外机制(例如一个电子邮件或者通过一个公共的网页)获得服务证书。此后, 客户端可以将有关服务证书的详细信息包含在配置文件中的终结点行为配置节里, 比如证书在客户端保存地址以及如何找到它。从客户端的角度来看, 这是迄今为止最为安全的选择, 由于其他服务无法拥有正确的证书(确切的说是证书对应的私钥), 任何试图通过搅乱客户端的地址解析, 将客户端调用重定向到一个有恶意的服务的举动都将被挫败。但同时这也是灵活性最差的选项, 因为每当客户端需要与另一个服务交互时, 客户端管理员都不得不重写客户端的配置文件。

如果要显式引用客户端可能与之交互的所有服务的证书,一个可行的替代方案是将这些证书保存在客户端的 Trusted People 证书库中,并指示 WCF 只允许调用其证书位于该文件夹下的服务。在这种情况下,客户端需要获取组成初始预调用协商 (Initial Pre-Call Negotiation) 的运行时的服务证书,并检查该证书是否位于 Trusted People 证书库中,如果是,则继续使用它以保护消息。事实上,这种证书协商形式是 WCF 的默认行为,不过我们可以在 WSHttpBinding 与 WSDualHttpBinding 绑定中禁用该功能 (使用一个硬配置 (Hard-Configured) 证书)。在互联网场景中,我们建议将服务证书保存在客户端的 Trusted People 证书库中,并配合使用证书协商机制。

## 服务证书验证

为了通知 WCF 采用何种程度去验证与信任服务的证书,我们需要在客户端的配置文件中添加一个定制的终结点行为。该行为应该使用 ClientCredentials 配置节。ClientCredentials 是一个终结点行为,它提供了类型为 X509CertificateRecipientClientCredential 的 ServiceCertificate:

```
public class ClientCredentials : ..., IEndpointBehavior
{
    public X509CertificateRecipientClientCredential ServiceCertificate
    {get;}
    // 更多成员
}
```

X509CertificateRecipientClientCredential 提供了类型为 X509CertificateRecipientClientCredential 的 Authentication 属性:

```
public sealed class X509CertificateRecipientClientCredential
{
    public X509ServiceCertificateAuthentication Authentication
    {get;}
    // 更多成员
}
```

X509CertificateRecipientClientCredential 提供了 X509CertificateValidationMode 枚举类型的 CertificateValidationMode 属性:

```
public enum X509CertificateValidationMode
{
    None,
    PeerTrust,
    ChainTrust,
    PeerOrChainTrust,
    Custom
}

public class X509ServiceCertificateAuthentication
```

```
{  
    public X509CertificateValidationMode CertificateValidationMode  
    {get;set;}  
    // 更多成员  
}
```

例 10-9 演示了如何在客户端的配置文件中设置服务证书的验证模式。

#### 例 10-9: 验证服务证书

```
<client>  
  <endpoint behaviorConfiguration = "ServiceCertificate"  
    ...  
  </endpoint>  
</client>  
<behaviors>  
  <endpointBehaviors>  
    <behavior name = "ServiceCertificate">  
      <clientCredentials>  
        <serviceCertificate>  
          <authentication certificateValidationMode = "PeerTrust"/>  
        </serviceCertificate>  
      </clientCredentials>  
    </behavior>  
  </endpointBehaviors>  
</behaviors>
```

X509CertificateValidationMode.PeerTrust 模式表示, WCF 可以信任位于客户端的 Trusted People 证书库中的服务证书。X509CertificateValidationMode.PeerTrust 模式则表示, WCF 可以信任由可信任的根权威 (Root Authority, 如 VeriSign 或 Thwart) 发布的服务证书, 而根权威的证书又可以在 Trusted Root Authority 证书库中找到。WCF 使用的默认值为 X509CertificateValidationMode.ChainTrust。X509CertificateValidationMode.PeerOrChainTrust 模式允许使用以上两个选项中的任意一个。

### 使用测试证书

开发者通常没有使用其所在组织的证书的权利, 因此, 我们要求助于一个测试证书, 比如由 *MakeCert.exe* 命令行工具所产生的证书。测试证书存在两个问题。第一个问题是, 当客户端采用默认的证书验证模式时, 它们无法通过验证, 因为客户端会默认使用 X509CertificateValidationMode.ChainTrust 模式。这个问题比较容易解决, 我们可以在客户端的 Trusted People 证书库中安装测试证书, 并采用 X509CertificateValidationMode.PeerTrust 或 X509CertificateValidationMode.PeerOrChainTrust 模式。第二个问题是, 在默认情况下 WCF 期望服务的证书名与服务宿主的域名 (或机器名) 一致。为了解决该问题, 客户端必须显式地在终结点的身份 DNS 配置节中指定测试证书的名称:

```
<client>
  <endpoint
    address = "http://localhost:8001/MyService"
    binding = "wsHttpBinding"
    contract = "IMyContract">
    <identity>
      <dns value = "MyServiceCert"/>
    </identity>
  </endpoint>
</client>
```

## 身份验证

客户端需要向代理提供它的凭证。ClientBase<T> 基类中的 ClientCredentials 属性（前面曾经提及）含有类型为 UserNamePasswordClientCredential 的 UserName 属性：

```
public class ClientCredentials : ..., IEndpointBehavior
{
    public UserNamePasswordClientCredential UserName
    {get;}
    // 更多成员
}

public sealed class UserNamePasswordClientCredential
{
    public string UserName
    {get;set;}
    public string Password
    {get;set;}
}
```

客户端使用 UserNamePasswordClientCredential 向服务传递它的用户名和密码，如例 10-10 所示。

---

**注意：**客户端不需要提供域名（如果使用 Windows 安全）或应用程序名（如果使用 ASP.NET Provider）。宿主会根据情况使用它自己的服务域名或一个配置好的应用程序名。

---

### 例 10-10：提供用户名及密码的凭证

```
MyContractClient proxy = new MyContractClient();

proxy.ClientCredentials.UserName.UserName = "MyUsername";
proxy.ClientCredentials.UserName.Password = "MyPassword";

proxy.MyMethod();
proxy.Close();
```



在局域网场景（见例10-3）中，客户端必须实例化一个新的 `NetworkCredential` 对象，互联网场景则不相同，我们无需（也没有办法）指定一个新的 `UserNamePasswordClientCredential` 对象。

当使用一个通道工厂而不是一个代理类时，我们需要用凭证对工厂的 `Credentials` 属性进行设定：

```
ChannelFactory<IMyContract> factory = new ChannelFactory<IMyContract>("");

factory.Credentials.UserName.UserName = "MyUsername";
factory.Credentials.UserName.Password = "MyPassword";

IMyContract proxy = factory.CreateChannel();
using(proxy as IDisposable)
{
    proxy.MyMethod();
}
```

注意，由于我们必须实例化一个工厂去访问 `Credentials` 属性，因此将不能使用 `ChannelFactory<T>` 类的 `CreateChannel()` 静态方法。

一旦在服务端的 WCF 框架接收到由用户名与密码构成的客户凭证，宿主就可以有选择地将它们作为 Windows 凭证、ASP.NET Membership Provider 的凭证，甚至是定制凭证加以验证。不管我们做出何种选择，都必须确保该凭证与基于角色的策略配置相匹配。

`ServiceCredentials` 类（`ServiceHostBase` 类中的 `Credentials` 属性）提供了类型为 `UserNamePasswordServiceCredentials` 的 `UserNameAuthentication` 属性：

```
public class ServiceCredentials : ..., IServiceBehavior
{
    public UserNamePasswordServiceCredential UserNameAuthentication
    {get;}
    // 更多成员
}
```

`UserNamePasswordServiceCredential` 类中含有一个对应的枚举类型的 `UserNamePasswordValidationMode` 属性：

```
public enum UserNamePasswordValidationMode
{
    Windows,
    MembershipProvider,
    Custom
}

public sealed class UserNamePasswordServiceCredential
{
    public MembershipProvider MembershipProvider
    {get;set;}
}
```

```

public UserNamePasswordValidationMode UserNamePasswordValidationMode
{get; set;}
// More members
}

```

通过设定 `UserNamePasswordValidationMode` 属性, 宿主可以选择如何对由用户名与密码构成的客户凭证进行身份验证。

## 使用 Windows 凭证

虽然并非常用, WCF 仍然允许面向互联网的服务将客户凭证作为 Windows 凭证加以验证。为了将客户端的用户名及密码作为 Windows 凭证验证, 需要将 `UserNamePasswordValidationMode` 设为 `UserNamePasswordValidationMode.Windows`。由于 `UserNamePasswordValidationMode.Windows` 是 `UserNamePasswordValidationMode` 属性的默认值, 因此以下两种定义是等效的:

```

ServiceHost host1 = new ServiceHost(typeof(MyService));

ServiceHost host2 = new ServiceHost(typeof(MyService));
host2.Credentials.UserNameAuthentication.UserNamePasswordValidationMode =
    UserNamePasswordValidationMode.Windows;

```

使用配置文件时, 我们需要增加一个定制行为, 该行为指定了用户名和密码的验证模式以及服务证书的信息, 如例 10-11 所示。

例 10-11: 互联网安全与 Windows 凭证

```

<services>
  <service name = "MyService" behaviorConfiguration = "UsernameWindows">
    ...
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name = "UsernameWindows">
      <serviceCredentials>
        <userNameAuthentication userNamePasswordValidationMode = "Windows"/>
        <serviceCertificate
          ...
        />
      </serviceCredentials>
    </behavior>
  </serviceBehaviors>
</behaviors>

```

与编程方式一样, 下面这行内容不是必须的, 因为它是默认设置。

```

<userNameAuthentication userNamePasswordValidationMode = "Windows"/>

```

## 授权

假如ServiceAuthorizationBehavior类中的PrincipalPermissionMode属性被设置为它的默认值PrincipalPermissionMode.UseWindowsGroups,那么一旦用户名和密码通过Windows身份验证,WCF将会创建一个Windows安全主体的对象,并把它附加到线程上。这就使得服务可以像在局域网场景中那样自由地(声明式或编码式)使用Windows NT工作组完成授权工作。

## 身份管理

只要安全主体的权限模式被设置为PrincipalPermissionMode.UseWindowsGroups,那么,在身份管理方面就与局域网场景一样,包括安全调用上下文的身份,如表10-4所示。一个局域网应用与一个使用Windows凭证的互联网应用之间主要区别在于,客户端不能指定所允许的模拟级别,这样宿主就可以任意地模拟客户端的身份。这是因为WCF会把安全调用上下文的Windows身份设置为TokenImpersonationLevel.Impersonation。

## 使用ASP.NET Provider

WCF基于角色的安全默认使用Windows用户组作为角色,并使用Windows账户作为安全标识。这一默认策略存在以下几个缺陷。首先,我们可能不希望为互联网应用的每个客户端都分配一个Windows账户。其次,该安全策略的控制粒度仅能达到宿主域中的用户组级别。通常我们无法控制最终客户的IT部门,而如果我们将自己的应用程序部署在一个粗的用户组粒度,或是用户组与用户在我们的应用程序里所扮演的真实角色未能实现良好的映射,又或是组名略有不同的环境中时,Windows基于角色的安全就无法发挥作用。角色的本地化问题也是阻碍我们应用Windows基于角色安全技术的另一个重要原因,因为角色的名称在不同区域的客户站点之间势必存在差异。因此,互联网应用极少使用Windows账户和用户组。.NET2.0在框架的实现中直接提供了被称为ASP.NET Provider模型的定制凭证管理机制,虽然名称中含有ASP.NET,但如果不是ASP.NET应用程序(如WCF应用程序),同样可以便捷地使用它验证用户身份并向它们授权,而无需再借助于Windows账户。

使用SQL Server作为用户凭证库是该体系结构的一种具体实现办法。在互联网应用中经常使用SQL Server作为存储信息的库,所以在这个场景中我们也可以发挥它的作用。运行位于%Windir%\Microsoft.NET\Framework\<Version>\目录下的安装文件aspnet\_regsql.exe,即可使用SQL Server Provider。安装程序将创建一个新的名为aspnetdb的数据库,其中包含了管理凭证所要用的表与存储过程。

SQL Server的凭证库是经过精心设计的,它采用了最新最好的技术管理用户凭证,如密码加盐(Password Salting)、存储过程等等。该基础设施有助于提高生产力,节约了开

发者宝贵的时间和精力，却提供了一个高质量的、安全的解决方案。又因为凭证管理架构采用了 Provider 模型，这就使得我们可以很容易地添加其他存储库实现，如一个 Access 数据库。

## 凭证 Provider

图 10-6 演示了 ASP.NET 2.0 凭证 Provider 的架构。

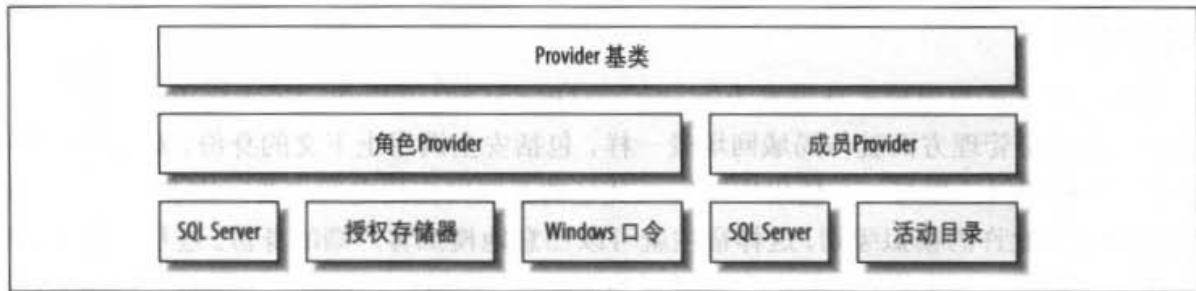


图 10-6: ASP.NET Provider 模型

成员 Provider 用于管理用户（用户名及密码），而角色 Provider 用于管理角色。ASP.NET 在框架的实现中支持直接将成员保存在 SQL Server 或活动目录中，而角色则可以被保存在 SQL Server、文件（授权库）或 NT 工作组（Windows 口令）中。

我们使用位于 System.Web.Security 命名空间下的一个名为 MembershipProvider 的类验证用户名和密码，类的定义如下：

```
public abstract class MembershipProvider : ProviderBase
{
    public abstract string ApplicationName
    {get;set;}
    public abstract bool ValidateUser(string name,string password);
    // 其他成员
}
```

MembershipProvider 类的目的是为了封装实际用到的 Provider 以及数据访问的细节，这样就可以在不影响应用程序本身的情况下更换成员 Provider (Membership Provider)。根据在宿主配置文件中配置的安全 Provider，WCF 使用了一个具体的数据访问类，例如针对 SQL Server 或 SQL Server Express 的 SqlMembershipProvider 类：

```
public class SqlMembershipProvider : MembershipProvider
{...}
```

然而，WCF 只会使用 MembershipProvider 基类中的功能。WCF 通过访问 Membership 类的 Provider 静态属性获得所需的成员 Provider。Membership 类的定义如下：

```
public sealed class Membership
{
    public static string ApplicationName
    {get;set;}
    public static MembershipProvider Provider
    {get;}
    public static bool ValidateUser(string username,string password);
    // 其他成员
}
```

Membership类定义了许多成员,用于支持用户管理的诸多方面。Membership.Provider 可用获取配制的 Provider 的类型,该 Provider 被放在宿主配置文件的 System.Web 配置节中。若未特别指明,角色 Provider 默认为 SqlMembershipProvider。

---

**注意:** 由于所有的成员 Provider 都继承于 MembershipProvider 抽象类,假如我们需要编写定制的凭证 Provider,同样应该派生自 MembershipProvider 类。

---

因为一个凭证库可以服务于多个应用程序,所以多个应用程序可能会定义相同的用户名。为了不出现冲突,凭证库中的每个记录都使用应用程序名称做进一步的限定(与 Windows 中用户名使用域名或机器名加以限定的方法类似)。

Membership 类的 ApplicationName 属性用于读取与设定应用程序的名称,ValidateUser() 方法使用凭证库验证指定的凭证,如果匹配则返回 true,否则返回 false。使用 Membership.ValidateUser 属性可方便地获得预先配置的 Provider。

如果我们为了实现授权,将应用程序配置为使用 ASP.NET 凭证库,并且启用了角色支持,那么在验证用户身份后,WCF 会创建一个 RoleProviderPrincipal 内部类的实例,并把它附加到调用操作的线程上:

```
sealed class RoleProviderPrincipal : IPrincipal
{...}
```

RoleProviderPrincipal 使用了 RoleProvider 抽象类实现授权:

```
public abstract class RoleProvider : ProviderBase
{
    public abstract string ApplicationName
    {get;set;}
    public abstract bool IsUserInRole(string username,string roleName);
    // 其他成员
}
```

RoleProvider 类中的 ApplicationName 属性将角色 Provider 绑定到特定的应用程序。IsUserInRole() 方法用于验证用户的角色成员。与成员 Provider 一样,所有的角色 Provider (包括定制的角色 Provider) 都必须继承于 RoleProvider 类。

RoleProvider类封装了实际用到的Provider,而实际用到的角色Provider会在宿主的配置文件中指定。根据配置的角色Provider,RoleProviderPrincipal会使用一个对应的数据访问类对调用者进行授权,例如SqlRoleProvider:

```
public class SqlRoleProvider : RoleProvider
{...}
```

我们可以通过访问Roles类的Provider静态属性获得所需的角色Provider。Roles类的定义如下:

```
public sealed class Roles
{
    public static string ApplicationName
    {get;set;}
    public static bool IsUserInRole(string username,string roleName);
    public static RoleProvider Provider
    {get;}
    // 其他成员
}
```

Roles.IsUserInRole()方法提供了访问Roles.Provider属性,并调用它的IsUserInRole()方法的简便途径。Roles.Provider从宿主的配置文件中取回预先配置好的Provider。若未特别指定,角色Provider将默认为SqlRolerProvider。

## 凭证管理

如果我们选择了Windows或活动目录保存自己应用程序中的用户与角色,那么就需要使用针对它们的指定工具管理用户凭证,如控制面板中的计算机管理工具或活动目录工具。

如果我们使用的是SQL SERVER,.NET 2.0 会在\inetpub\wwwroot\aspnet\_webadmin\<version number>目录下安装网站管理的页面。开发人员可以直接在Visual Studio 2005中配置应用程序。当我们从网站菜单中选择ASP.NET配置时,Visual Studio 2005将会转到ASP.NET管理页面,通过它设置包括安全配置在内的各种参数。我们可以为自己的应用程序配置以下方面的内容:

- 创建新的用户及删除已有的用户
- 创建新的角色及删除已有的角色
- 为用户分配角色
- 获取用户的详细信息
- 设置用户的状态
- 使用与本章无关的额外功能



## Visual Studio 2005 的缺点

在 Visual Studio 2005 驱动的管理页面中存在许多显著的缺陷。首先，我们需要用到 Visual Studio 2005。但是，应用程序或系统管理员未必会有 Visual Studio 2005，更不用说知道如何使用它了。管理页面默认地使用“/”作为应用程序的名称，而且没有提供任何可视化的方法可以修改它。我们必须创建一个 web 应用程序去激活该管理页面却又无法实现远程访问：应用程序和 Visual Studio 2005 必须处于同一台机器，这样 Visual Studio 2005 才可以访问应用程序的配置文件。此外，基于浏览器的用户界面有点令人厌恶（我们需要频繁地点击返回按钮），而且相对较为死板。更严重的是管理员希望用到的很多功能并不能通过这个管理页面访问到，即使底层的 Provider 类支持这些功能。Visual Studio 2005 驱动的管理页面缺少的功能包括：

- 更新用户账户中大多数或所有详细信息
- 获取用户密码
- 更改用户密码
- 重新设置用户密码
- 获得当前在线用户的数目
- 在一个操作中删除某个角色下的所有用户
- 获取有关密码管理策略的信息（如长度、重设策略、密码类型等等）
- 测试用户的凭证
- 验证用户的角色成员

此外，仍有一些管理员可能希望用到，但是目前尚未支持（甚至 Provider 类也不支持）的额外功能。这些功能包括获取数据库中所有应用程序列表的能力，删除一个应用程序中所有用户的能力，删除一个应用程序的能力，删除某个应用程序（以及与它相关的所有用户和角色），还有删除所有应用程序的能力。

## 凭证管理器

这个工具存在的诸多缺陷促使我自己开发了一个凭证管理器应用程序，它是一个智能客户端（Smart Client）应用程序，能够解决以上所有问题。图 10-7 为凭证管理器的截图（注 1）。

---

注 1： 在 CoDe 杂志 2005 年 11 月刊中，作者的文章《Manage Custom Security Credentials the Smart (Client) Way》首次发布了一个凭证管理器的早期版本。此后，凭证管理器成为了在实际应用中管理 ASP.NET SQL SERVER 凭证库的标准工具。



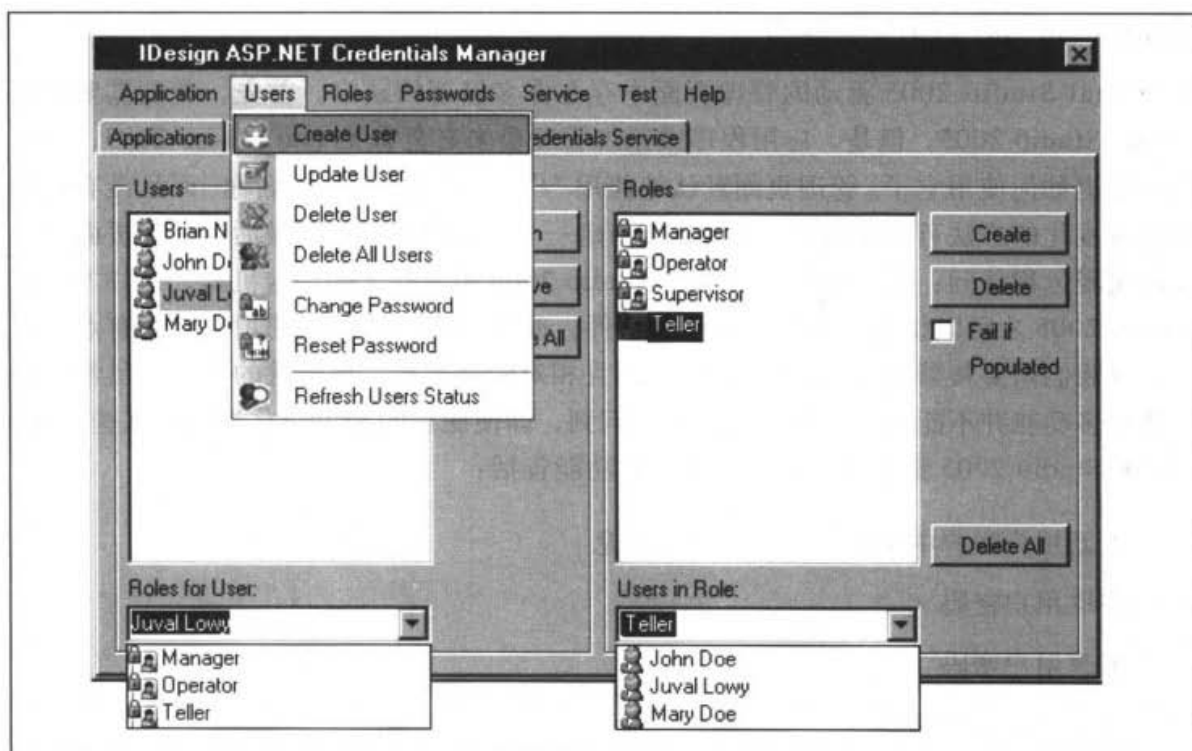


图 10-7：凭证管理器工具

凭证管理器连同本书中的源代码均可以通过<http://www.oreilly.com/catalog/9780596526993>获得。

我们使用了一个 WCF 服务（可以是自托管或托管在 IIS 中）包装 ASP.NET Provider，并增加了一些缺乏的功能，如删除一个应用程序。

凭证管理器使用这些终结点提供的功能管理凭证库。此外，它还允许管理者在运行时选取凭证服务的地址，并使用在第 2 章中提到的 MetadataHelper 验证所提供的地址是否真的支持需要用到的契约。

## 验证

为了使用了 ASP.NET Provider 验证客户端的用户名与密码，我们需要将 `UserNamePasswordValidationMode` 属性设置为 `UserNamePasswordValidationMode.MembershipProvider`：

```
ServiceHost host = new ServiceHost(typeof(MyService));  
host.Credentials.UserNameAuthentication.UserNamePasswordValidationMode =  
    UserNamePasswordValidationMode.MembershipProvider;
```

宿主的配置文件决定了使用何种 Provider。此外，宿主的配置文件还必须包括诸如 SQL Server 连接字符串在类的所有与 Provider 相关的设置，如例 10-12 所示。

## 例 10-12: 在互联网安全中使用一个 ASP.NET SQL Server Provider

```

<connectionStrings>
  <add name= "AspNetDb" connectionString = "data source=(local);
                                             Integrated Security=SSPI;Initial Catalog=aspnetdb"/>
</connectionStrings>

<system.serviceModel>
  <services>
    <service name = "MyService" behaviorConfiguration = "ASPNETProviders">
      <endpoint
        ...
      />
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name = "ASPNETProviders">
        <serviceCredentials>
          <userNameAuthentication
            userNamePasswordValidationMode = "MembershipProvider"/>
          <serviceCertificate
            ...
          />
        </serviceCredentials>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>

```

默认的应用程序名是一个没有意义的“/”符号,因此我们必须为自己的应用程序指定一个名称。一旦 ASP.NET Provider 配置好后, WCF 将为 UserNamePasswordServiceCredential 类中的 MembershipProvider 属性初始化一个配置好的成员 Provider 的实例。我们可以通过编程方式访问该成员 Provider, 并通过它设置应用程序的名称:

```

ServiceHost host = new ServiceHost(typeof(MyService));
Debug.Assert(host.Credentials.UserNameAuthentication.MembershipProvider != null);
Membership.ApplicationName = "MyApplication";

```

我们也可以在配置文件中设置应用程序的名称,只不过我们需要为此定义一个定制的 ASP.NET 成员 Provider, 如例 10-13 所示。

## 例 10-13: 为成员 Provider 设置应用程序名称

```

<system.web>
  <membership defaultProvider = "MySqlMembershipProvider">
    <providers>
      <add name = "MySqlMembershipProvider"
        type = "System.Web.Security.SqlMembershipProvider"
        connectionStringName = "AspNetDb"
        applicationName = "MyApplication"
      >
    </providers>
  </membership>

```

```

        />
    </providers>
</membership>
</system.web>
<connectionStrings>
    <add name = "AspNetDb"
        ...
    />
</connectionStrings>

```

在例 10-13 中，我们在 `system.web` 配置节中增加了一个 `Providers` 配置节，在其中添加了一个定制成员 `Provider`，并将它设为新的默认成员 `Provider`。接下来，我们需要列出新的 `Provider` 完整的类型名称。我们完全可以引用一个成员 `Provider` 已有的任何实现，如 `SqlMembershipProvider`，正如例 10-13 所示。使用 `SQLProvider` 时，我们还必须列出需要用到的连接字符串，而不能依赖于 `machine.config` 文件的默认连接字符串。最后，千万别忘记在 `ApplicationName` 标签中设置我们所期望的应用程序的名称。

## 授权

为了支持对用户的授权，我们必须在宿主配置文件中添加以下片断以启用基于角色的安全：

```

<system.web>
    <roleManager enabled = "true"/>
</system.web>

```

---

**注意：** 如果希望在代码中启用角色管理，需要使用反射技术。

---

采用这种方式启用角色管理时，会初始化一个 `Roles` 类，并将它的 `Provider` 属性设为之前配置的 `Provider`。如果想使用 ASP.NET 角色 `Provider`，需要将 `PrincipalPermissionMode` 属性设置为 `PrincipalPermissionMode.UseAspNetRoles`：

```

ServiceHost host = new ServiceHost(typeof(MyService));
host.Authorization.PrincipalPermissionMode=PrincipalPermissionMode.UseAspNetRoles;

```

或使用配置文件，为它增加一个定制行为以达到同样目的：

```

<services>
    <service name = "MyService" behaviorConfiguration = "ASPNETProviders">
        ...
    </service>
</services>
<behaviors>
    <serviceBehaviors>
        <behavior name = "ASPNETProviders">

```

```

        <serviceAuthorization principalPermissionMode = "UseAspNetRoles"
        ...
    </behavior>
</serviceBehaviors>
</behaviors>

```

验证完客户端份之后, ServiceAuthorizationBehavior 类中的 RoleProvider 属性将被设置为之前配置的角色 Provider:

```

public sealed class ServiceAuthorizationBehavior : IServiceBehavior
{
    public RoleProvider RoleProvider
    {get;set;}
    // 更多成员
}

```

默认的应用程序名是一个没有意义的“/”符号, 因此我们必须为自己的应用程序指定一个名称。不同于成员 Provider, 我们无法通过访问 ServiceAuthorizationBehavior 类中的 RoleProvider 属性来设定应用程序的名称, 因为在通过身份验证前该值始终为 null。因此, 我们使用静态辅助类 Roles 作为替代的解决方案:

```

ServiceHost host = new ServiceHost(typeof(MyService));
Debug.Assert(host.Credentials.UserNameAuthentication.MembershipProvider != null);
Roles.ApplicationName = "MyApplication";

```

我们也可以在配置文件中设置应用程序的名称, 只不过我们需要为此定义一个定制的 ASP.NET 角色 Provider, 如例 10-14 所示。

#### 例 10-14: 为角色 Provider 设置应用程序的名称

```

<system.web>
  <roleManager enabled = "true" defaultProvider = "MySqlRoleManager">
    <providers>
      <add name = "MySqlRoleManager"
        type = "System.Web.Security.SqlRoleProvider"
        connectionStringName = "AspNetDb"
        applicationName = "MyApplication"
      />
    </providers>
  </roleManager>
</system.web>
<connectionStrings>
  <add name = "AspNetDb"
    ...
  />
</connectionStrings>

```

和成员 Provider 一样, 我们在 system.web 配置节中增加了一个 providers 配置节, 在其中添加了一个定制的角色 Provider, 并将它设置为新的默认的角色 Provider。接下来, 我们需要列出新的 Provider 完整的类型名称。和成员 Provider 一样, 我们完全可以

引用一个角色 Provider 已有的任何实现，例如 SqlRoleProvider。同样，我们必须列出要用到的连接字符串。最后，在 ApplicationName 标签中设置我们所期望的应用程序的名称。

## 声明方式实现基于角色的安全

我们可以像在局域网场景中一样使用 PrincipalPermission 特性验证角色成员，因为该特性所做的工作就是访问附属于线程的安全主体对象，它已经被 WCF 设置为 RoleProviderPrincipal。例 10-15 演示了使用 ASP.NET Provider 以声明方式实现基于角色的安全。

例 10-15：使用 ASP.NET Provider 以声明方式实现基于角色的安全

```
class MyService : IMyContract
{
    [PrincipalPermission(SecurityAction.Demand, Role = "Manager")]
    public void MyMethod()
    { ... }
}
```

局域网场景与互联网场景之间唯一显著的差异在于，我们无需在互联网场景中为角色名称加上应用程序名或域名作为其前缀。

## 身份管理

当我们在互联网场景中使用 ASP.NET Provider 时，与安全主体对象相关的身份其实是一个 GenericIdentity，它包装了客户端所提供的用户名。该身份被认为是通过验证的。此时，安全调用上下文的基本身份与安全主体的身份是一致的。而另一方面，Windows 身份将被设为一个带有空白用户名的 Windows 身份，也就是说，它是未经验证的。表 10-5 演示了在这个场景中的各种身份。

表 10-5：在互联网场景中使用 ASP.NET Provider 进行身份管理

身份	类型	值	是否验证
Thread Principal	GenericIdentity	Username	Yes
Security Context Primary	GenericIdentity	Username	Yes
Security Context Windows	WindowsIdentity	-	No

## 模拟

由于缺乏有效的 Windows 凭证，服务无法模拟它的任何客户端。

## 回调

当我们使用 ASP.NET Provider 时，尽管回调消息施加了保护，但所有针对回调对象的调用都使用了一个未经验证的安全主体。因此，安全主体身份将被设为一个带有空白用户名的 Windows 身份，从而无法对它进行授权或采用基于角色的安全技术，因为它被视为一个匿名身份。虽然回调调用中确实包含一个安全调用上下文，但 Windows 身份还是会被设为一个表示空白身份的 `WindowsIdentity` 实例，这就使得我们无法使用模拟技术。此时，唯一有意义的信息保存在基本身份中，该身份被设为一个 `X509Identity` 类型的实例，它的名称被设为服务宿主证书的公用名，并以证书的指纹（Thumbprint，一种哈希码）作为后缀。

```
class MyClient : IMyContractCallback
{
    public void OnCallback()
    {
        IPrincipal principal = Thread.CurrentPrincipal;
        Debug.Assert(principal.Identity.IsAuthenticated == false);

        ServiceSecurityContext context = ServiceSecurityContext.Current;
        Debug.Assert(context.PrimaryIdentity.Name ==
            "CN=MyServiceCert; D6E33B50BCF6D9609E68762F2C6A14F65679268B");
        Debug.Assert(context.IsAnonymous == false);
    }
}
```

建议避免在回调过程中执行任何敏感性的工作，因为我们无法方便地使用基于角色的安全技术。

## B2B 应用程序

在 B2B 场景中，服务与客户端是两个完全不同的商业实体。它们不会共享凭证或者账户，而且它们之间的通信通常是不对外公开的。与服务交互的客户端的数量相对较少，而且只有当它们签订了一个复杂的商业合约，且相关条件都得到满足后才会产生交互。客户端使用 X509 证书向服务表明自己的身份，而不是采用 Windows 账户或用户名。这些证书被认为是访问服务的先决条件。客户端或服务可能都不使用 WCF，甚至没有使用 Windows。如果我们正在编写一个服务或者一个客户端，我们不能假设在另一端也使用了 WCF。客户端的调用起始于防火墙之外，在传输过程中依赖于 HTTP，并且可能会经过多个中间方。

## 确保 B2B 绑定的安全

对于 B2B 场景而言，我们应该采用互联网绑定，即 `BasicHttpBinding`、`WSHttpBinding`

与 `WSDualHttpBinding`。我们必须采用 `Message Security` 作为传输安全模式以提供能够跨越所有中介的端对端的安全。就像在互联网场景中一样，我们会使用服务端的证书确保消息的安全。不过，与互联网场景中不同的是，客户端提供的凭证是以证书形式表现的。只要我们选择 `MessageCredentialType.Certificate` 作为 `Message Security` 传输安全模式的客户端凭证类型，就可以指定这些绑定都使用证书作为凭证。而且，我们需要同时在客户端和服务端对此进行配置。例如，我们可以编写如下代码完成 `WSHttpBinding` 的配置：

```
WSHttpBinding binding = new WSHttpBinding();
binding.Security.Message.ClientCredentialType = MessageCredentialType.Certificate;
```

或者采用配置文件：

```
<bindings>
  <wsHttpBinding>
    <binding name = "WSCertificateSecurity">
      <Security mode = "Message">
        <message clientCredentialType = "Certificate"/>
      </Security>
    </binding>
  </wsHttpBinding>
</bindings>
```

## 身份验证

服务管理员有多种方式验证客户端发送的证书。如果证书通过验证，那么该客户端也随之通过验证。如果不对证书加以验证，客户端就会随便发送一个证书。如果我们采用信任链作为验证方法，那么只要客户端的证书是由一个值得信任的根权威发布的，客户端就可以通过验证。尽管如此，对等信任（Peer Trust）才是验证客户端证书的常用方法。服务管理员应该安装所有被允许与服务进行交互的客户端的证书，这些证书将保存在服务本地机器的 `Trusted People` 证书库中。当服务接收到客户端的证书时，如果该证书可以在信任证书库中找到，则客户端可以通过验证。因此，在 B2B 场景中我们应采用对等信任方式。

`ServiceCredentials` 类定义了类型为 `X509CertificateInitiatorServiceCredential` 的 `ClientCertificate` 属性：

```
public class ServiceCredentials : ..., IServiceBehavior
{
    public X509CertificateInitiatorServiceCredential ClientCertificate
    {get;}
    // 更多成员
}
```



X509CertificateInitiatorServiceCredential 类定义了类型为 X509ClientCertificateAuthentication 的 Authentication 属性, 我们可以通过该属性设置证书的验证方法:

```
public sealed class X509CertificateInitiatorServiceCredential
{
    public X509ClientCertificateAuthentication Authentication
    {get;}
    // 更多成员
}
public class X509ClientCertificateAuthentication
{
    public X509CertificateValidationMode CertificateValidationMode
    {get;set;} // 更多成员
}
```

例 10-16 演示了在 B2B 场景中设置宿主配置文件的方式。注意, 在例 10-16 中宿主仍然需要向客户端提供自己的证书, 以实现 Message Security 传输安全模式。

#### 例 10-16: 为 B2B 安全配置宿主

```
<services>
  <service name = "MyService" behaviorConfiguration = "BusinessToBusiness">
    ...
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name = "BusinessToBusiness">
      <serviceCredentials>
        <serviceCertificate
          ...
        />
        <clientCertificate>
          <authentication certificateValidationMode = "PeerTrust"/>
        </clientCertificate>
      </serviceCredentials>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

客户端需要指定证书的位置、名称及检索方法, 以引用将要使用的证书。我们可以通过访问代理的 ClientCredentials 属性实现这点。该属性类型中又包含了类型为 X509CertificateInitiatorClientCredential 的 ClientCertificate 属性:

```
public class ClientCredentials : ..., IEndpointBehavior
{
    public X509CertificateInitiatorClientCredential ClientCertificate
    {get;}
    // 更多成员
}
```

```

public sealed class X509CertificateInitiatorClientCredential
{
    public void SetCertificate(StoreLocation storeLocation,
                              StoreName storeName,
                              X509FindType findType,
                              object findValue);

    // 更多成员
}

```

不过，客户端通常是在它的配置文件中设定这些值的，如例 10-17 所示。

例 10-17：设置客户端的证书

```

<client>
  <endpoint behaviorConfiguration = "BusinessToBusiness"
    ...
  />
</client>
...
<behaviors>
  <endpointBehaviors>
    <behavior name = "BusinessToBusiness">
      <clientCredentials>
        <clientCertificate
          findValue      = "MyClientCert"
          storeLocation  = "LocalMachine"
          storeName      = "My"
          x509FindType   = "FindBySubjectName"
        />
        ...
      </clientCredentials>
    </behavior>
  </endpointBehaviors>
</behaviors>

```

配置文件还需要指定验证服务证书的方法。如果采用 BasicHttpBinding，由于该绑定不能通过协商获得服务证书，客户端的配置文件需要在终结点行为的服务证书配置节中包含将要用到的服务证书所处的位置。注意，与互联网场景一样，当我们使用一个服务测试证书时，客户端的配置文件仍然必须包含与终结点身份相关的信息。

一旦配置好客户端证书，我们也就无需再对代理类做特别处理了：

```

MyContractClient proxy = new MyContractClient();
proxy.MyMethod();
proxy.Close();

```

## 授权

默认地，服务不能使用基于安全主体的或基于角色的安全。理由是提供的凭证，即客户端的证书，无法与 Windows 或者 ASP.NET 的用户账户匹配。由于 B2B 终结点和服务通

常专用于一个小的客户端的集合，甚至是一个特殊的客户端，因此，这种缺少对授权的支持不会造成太大的问题。如果实际情况正是如此，就应该将 `PrincipalPermissionMode` 属性设置为 `PrincipalPermissionMode.None`，如此一来，WCF 将向一个一般主体附加一个空白的身份，而不是向一个 Windows 身份附加一个空白的身份。

另一方面，如果我们仍然希望对客户端进行授权，那么事实上我们也可以做到这一点。本质上我们所需要做的只是部署一些凭证声明库 (Credentials Claim Store)，并添加客户端的证书名称，即对应于该声明库的证书的通用名称以及证书指纹，然后在需要的时候对该库执行访问检查。

实际上，没有什么能够阻止我们利用 ASP.NET 角色 Provider 来进行授权，尽管在验证中并不使用成员 Provider。这种单独使用 Provider 的能力是针对 ASP.NET Provider 模型的一个核心的设计目标。

首先，我们需要在宿主的配置文件中启动角色 Provider，并像例 10-14 中那样配置应用程序的名称，或者我们也可以通过编程方式提供应用程序的名称。

接下来，把客户端的证书和指纹当作一个用户添加到成员库中，并为它分配角色。例如，如果使用一个通用名为 `MyClientCert` 的证书，我们需要通过“`CN=MyClientCert; 12A06153D25E94902F50971F68D86DCDE2A00756`”的名称向成员库中添加一个用户，并为它提供密码。当然，这个密码是不相关的，而且将不会被使用。一旦我们已经创建好了该用户，就应该在应用程序中为它分配合适的角色。

其中最重要的是将 `PrincipalPermissionMode` 属性设置为 `PrincipalPermissionMode.UseAspNetRoles`。例 10-18 列出了在宿主配置文件中所要求的设置。

例 10-18：针对 B2B 场景的 ASP.NET 基于角色的安全

```
<system.web>
  <roleManager enabled = "true" defaultProvider = "MySQLRoleManager">
    <Providers>
      <add name = "MySQLRoleManager"
        ...
        applicationName = "MyApplication"
      />
    </Providers>
  </roleManager>
</system.web>
<system.serviceModel>
  <services>
    <service name = "MyService" behaviorConfiguration = "BusinessToBusiness">
      ...
    </service>
  </services>
  <behaviors>
```

```

<serviceBehaviors>
  <behavior name = "BusinessToBusiness">
    <serviceCredentials>
      <serviceCertificate
        ...
      />
      <clientCertificate>
        <authentication certificateValidationMode = "PeerTrust"/>
      </clientCertificate>
    </serviceCredentials>
    <serviceAuthorization principalPermissionMode = "UseAspNetRoles"/>
  </behavior>
</serviceBehaviors>
</behaviors>
<bindings>
  ...
</bindings>
</system.serviceModel>

```

现在我们可以向例 10-15 中那样使用基于角色的安全了。

## 身份管理

如果 `PrincipalPermissionMode` 属性被设置为 `PrincipalPermissionMode.None`, 那么一个带有空白用户名的 `GenericIdentity` 将成为安全主体的身份。包含客户证书公用名与指纹的一个 `X509Identity` 类型的实例, 将成为安全调用上下文的基本身份。由于缺乏有效的 Windows 凭证, 安全调用上下文的 Windows 身份的用户名将为空。如果 `PrincipalPermissionMode` 属性被设置为 `PrincipalPermissionMode.UseAspNetRoles`, 那么安全主体的身份以及安全调用上下文的基本身份都会使用一个包含客户证书公用名与指纹的 `X509Identity` 类型的实例。而安全调用上下文的 Windows 身份中的用户名仍将为空。表 10-6 列出了上述配置。

表 10-6: 在 B2B 场景中使用 ASP.NET 角色 Provider 的身份管理

身份	类型	值	是否验证
Thread Principal	X509Identity	Client cert name	Yes
Security Context Primary	X509Identity	Client cert name	Yes
Security Context Windows	WindowsIdentity	-	No

## 模拟

由于缺乏有效的 Windows 凭证, 服务无法模拟它的任何客户端。

## 回调

由于B2B场景和互联网场景使用了相同的传输安全模式，并且服务都使用证书标识自己的身份，因此在两个场景中回调的表现也是一致的。和互联网的回调场景一样，应当尽量避免在回调过程中执行敏感性工作，因为我们不能使用基于角色的安全，而且就安全主体而言，调用者是未经验证的。

## 宿主的安全配置

尽管图 10-8 不是专门针对 B2B 场景的，但它也涵盖了这个场景。本图是我们在本章中第一次向读者完整地展示服务宿主中与安全有关的所有元素。

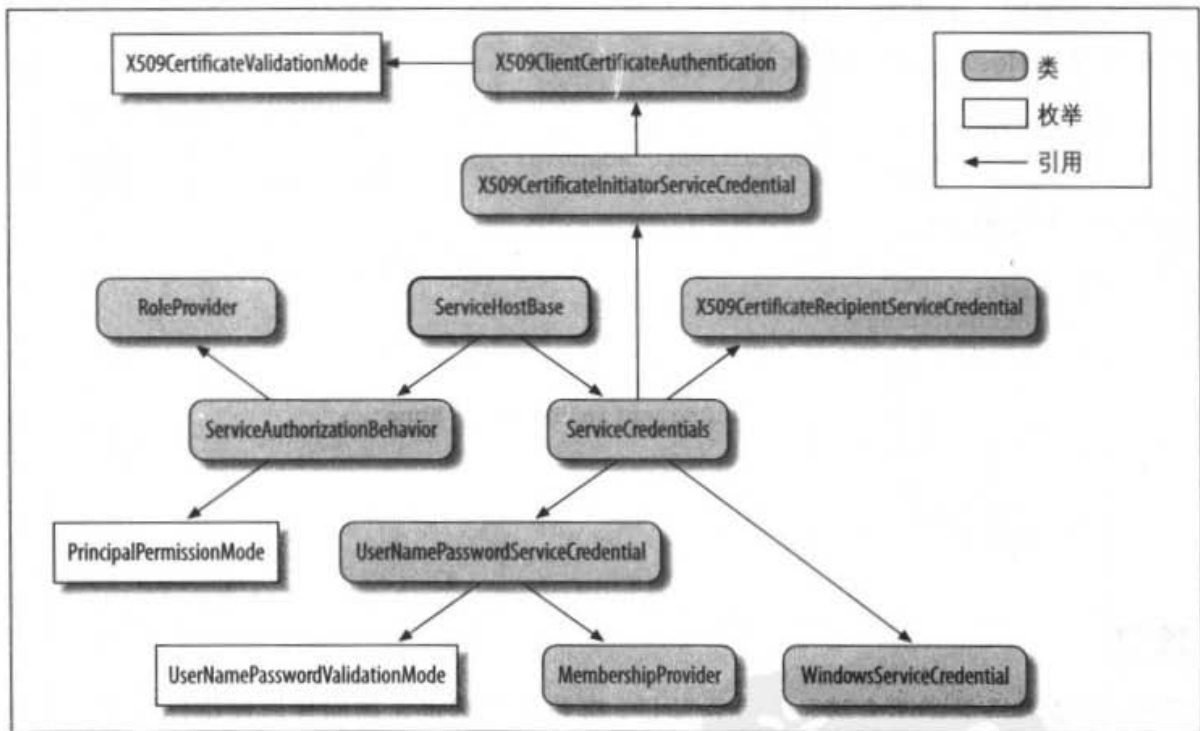


图 10-8: ServiceHostBase 中的安全元素

## 匿名应用程序

在匿名场景中，客户端无需提交任何凭证就可访问服务，此时它们都是匿名的。而另一方面，客户端和服务确实也需要保证消息传输的安全，以防消息被篡改或被嗅探 (Sniffing)。面向互联网与基于局域网的应用程序可能都需要提供匿名的，同时又能保证端到端安全的访问。匿名场景中的客户端数量根据实际应用可大可小。客户端可以通过 HTTP 或者 TCP 连接服务。

## 确保匿名绑定的安全

由于客户端访问服务时可能经过多个中间方，同时还要保证消息的安全，这意味着在匿名场景中我们应该使用 Message Security 传输安全模式，因为只要将消息凭证设为不使用凭证就可以轻松地实现这两个需求。不过服务本身需要被配置为使用证书，以确保消息的安全。在匿名场景中，我们只能使用 NetTcpBinding、WSHttpBinding、WSDualHttpBinding 与 NetMsmqBinding，可以看出，在这些绑定中，既包含互联网绑定也包含了局域网绑定，正如之前所要求的那样。注意，我们不能使用 BasicHttpBinding、NetNamedPipeBinding、NetPeerTcpBinding 或 WSFederationHttpBinding，因为这些绑定既不支持 Message Security 模式，也不允许在消息中不包含任何凭证（见表 10-1 和表 10-3）。对允许的绑定进行配置的方式与之前的场景类似，主要差异体现在这里不使用客户凭证，例如在 WSHttpBinding 中将凭证类型设为 MessageCredentialType.None：

```
WSHttpBinding binding = new WSHttpBinding();  
binding.Security.Message.ClientCredentialType = MessageCredentialType.None;
```

或使用一个配置文件：

```
<bindings>  
  <wsHttpBinding>  
    <binding name = "WSAnonymous">  
      <Security mode = "Message">  
        <message clientCredentialType = "None"/>  
      </Security>  
    </binding>  
  </wsHttpBinding>  
</bindings>
```

## 验证

显然，在匿名场景中不会涉及到身份验证，客户端当然也无需向代理提供任何安全凭证之类的信息。不过，为了向客户端证明自己的身份，并对消息加以保护，服务需要提供自己的证书，如例 10-8 所示。

## 授权

由于客户端是匿名的（而且未经验证），我们也就无法使用授权以及基于角色的安全技术。服务宿主应该将 PrincipalPermissionMode 属性设置为 PrincipalPermissionMode.None，以便让 WCF 建立一个具有空白身份的一般安全主体，而不是一个具有空白身份的 Windows 安全主体。

## 身份管理

在使用 `PrincipalPermissionMode.None` 选项时，与安全主体关联的身份将是一个带有空白用户名的 `GenericIdentity`。这个身份被认为是未经验证的。安全调用上下文的基本身份与安全主体的身份在这里是一致的。而另一方面，Windows 身份也会被设 为一个带有空白用户名的 Windows 身份，它同样表示未经验证。表 10-7 展示了在这个场景中的各种身份。

表 10-7: 匿名场景中的身份管理

身份	类型	值	是否验证
Thread Principal	<code>GenericIdentity</code>	-	No
Security Context Primary	<code>GenericIdentity</code>	-	No
Security Context Windows	<code>WindowsIdentity</code>	-	No

## 模拟

由于客户端都是匿名的，服务无法模拟它的任何客户端。

## 回调

尽管客户端对服务的调用是匿名的，但服务却向客户端表明了它的身份。安全调用上下文的基本身份将被设置为 `X509Identity` 类的一个实例，该实例的名称被设为服务证书的公共名并以证书的指纹作为后缀。其他信息则被屏蔽。安全主体的身份将被设置为一个带有空白用户名的 Windows 身份，这使得我们无法应用授权和基于角色的安全技术，就像被认为是匿名的那样。安全调用上下文的 Windows 身份将被设为一个带有空白身份的 `WindowsIdentity` 的实例，这也阻碍了模拟技术的应用。由于我们无法使用基于角色的安全技术，而且就安全主体而言调用者是未经验证的，所以应当尽量避免在回调过程执行敏感性的工作。

## 无安全

在最后一个场景中，我们的应用程序将完全关闭安全保护。服务不依赖于任何传输安全模式，并且它也不会对自己的调用者加以验证或授权。显而易见，这样一个服务是完全暴露的，而且通常是由于业务的需要，我们才会放弃对服务的安全保护。放弃安全保护，服务就可以接受任意数目的客户端，而且无论互联网服务或局域网服务都可以被配置为无安全保护。



## 不保证绑定的安全

为了关闭安全选项，需要将传输安全模式设为 `None`。这也使得消息中不再包含客户凭证。所有的绑定都支持 `None` 选项（见表 10-1），尽管如此，我们不应该在使用 `WSFederalHttpBinding` 时采用这种模式，因为选择该绑定的唯一原因就是出于对联邦安全的考虑。

除了安全传输模式应该设为 `None`，对绑定的其他配置都与此前场景类似。例如，下面对 `NetTcpBinding` 采用 `MessageCredentialType.None`：

```
NetTcpBinding binding = new NetTcpBinding(SecurityMode.None);
```

或使用一个配置文件：

```
<bindings>
  <netTcpBinding>
    <binding name = "NoSecurity">
      <Security mode = "None"/>
    </binding>
  </netTcpBinding>
</bindings>
```

## 身份验证

显然，在这个场景中不会执行任何身份验证的工作，而且客户端也无需向代理提供任何凭证。同样地，客户端也不会验证服务的身份。

## 授权

由于客户端是匿名的（而且未经验证），我们也就无法使用授权以及基于角色的安全技术。`WCF` 会自动将 `PrincipalPermissionMode` 属性设置为 `PrincipalPermissionMode.None`，此时，`WCF` 会安装一个带有空白身份的一般主体。

## 身份管理

与安全主体对象关联的身份是一个带有空白用户名的 `GenericIdentity`。该身份被认为是未经验证的。与此前的所有场景不同，在无安全场景中，操作并没有安全调用上下文，`ServiceSecurityContext.Current` 将返回 `null`。表 10-8 演示了在无安全场景中的身份。

表 10-8：在无安全场景中的身份管理

身份	类型	值	是否验证
Thread Principal	GenericIdentity	-	No
Security Context Primary	-	-	-
Security Context Windows	-	-	-

## 模拟

由于客户端都是匿名的，服务无法模拟它的任何客户端。

## 回调

不同于此前所有的场景，在没有传输安全保障的情况下，回调是以客户端本身的身份执行。安全主体的身份将被设置为一个带有客户端用户名的WindowsIdentity的实例。回调将会被验证，但由于仅会出现客户端对其自身进行授权的情况，所以无论使用模拟还是基于角色的安全技术都是没有意义的。此外，回调的安全调用上下文将被设为null。

## 场景总结

现在我们已经了解了五种关键场景，表10-9和表10-10将作为其中关键元素的一个总结。表10-9列举了在每个场景中用到的绑定。再次需要注意的是，尽管就技术而言几乎在每一个场景中都可以使用其他的绑定，但所选择的绑定应该与它被用到的场景的上下文相一致。

表 10-9：绑定与安全场景

绑定	局域网场景	互联网	B2B 场景	匿名场景	无安全场景
BasicHttpBinding	No	No	Yes	No	Yes
NetTcpBinding	Yes	Yes	No	Yes	Yes
NetPeerTcpBinding	No	No	No	No	Yes
NetNamedPipeBinding	Yes	No	No	No	Yes
WSHttpBinding	No	Yes	Yes	Yes	Yes
WSFederationHttpBinding	No	No	No	No	No
WSDualHttpBinding	No	Yes	Yes	Yes	Yes
NetMsmqBinding	Yes	No	No	Yes	Yes

表 10-10 演示了在本章开始所定义的每一个安全特性（传输安全、服务及客户端的验证、授权还有模拟）如何与每个场景关联。

表 10-10：安全场景特性

特性	局域网场景	互联网	B2B 场景	匿名场景	无安全场景
Transport	Yes	No	No	No	No
Message	No	Yes	Yes	Yes	No
Service authentication	Windows	Certificate	Certificate	Certificate	No
Client authentication	Windows	ASP.NET	Certificate	No	No
Authorization	Windows	ASP.NET	No/ASP.NET	No	No
Impersonation	Yes	No	No	No	No

## 声明式安全框架

WCF 安全确实是一个很大的课题。有太多令人生畏的细节需要掌握，彼此之间又存在着错综复杂的关系。它的编程模型非常复杂，给人一种仿佛陷入迷宫一般无法抗拒的感觉。而使得情形更加糟糕的是，为了保障安全性，它会对应用程序以及业务层带来严重的影响。为了解决这一问题，我们提出了一个针对 WCF 的声明式安全框架。针对服务，我们提供了一个安全特性（以及它对宿主对应的支持）；针对客户端，我们则提供了几个辅助类和安全代理类。我们所实现的声明式框架极大地简化了对 WCF 安全的应用，它使得 WCF 对安全的配置可以如同其他 WCF 配置，例如对事务或同步的配置那样，采取同样的方式实现。我们所期望的声明式模型是便于使用的，能够最小程度地忽略对安全的众多细节的理解。作为开发者，你所需要的做就是选择正确的场景（不仅仅是本章所讨论的 5 种常见场景），那么我们实现的框架就能够自动完成配置。不仅如此，框架还会强制要求正确的选项，遵循建议方案。同时，我们所期望的模型还能够随着需求的增加，能够维持对基本配置的粒度与控制。

## SecurityBehaviorAttribute 特性

例 10-19 列出了 SecurityBehaviorAttribute 以及 ServiceSecurity 枚举的定义。ServiceSecurity 定义了框架支持的 5 种场景。

例 10-19：SecurityBehaviorAttribute 特性

```
public enum ServiceSecurity
{
    None,
    Anonymous,
```

```
        BusinessToBusiness,
        Internet,
        Intranet
    }
    [AttributeUsage(AttributeTargets.Class)]
    public class SecurityBehaviorAttribute : Attribute, IServiceBehavior
    {
        public SecurityBehaviorAttribute(ServiceSecurity mode);
        public SecurityBehaviorAttribute(ServiceSecurity mode,
                                         string serviceCertificateName);
        public SecurityBehaviorAttribute(ServiceSecurity mode,
                                         StoreLocation storeLocation,
                                         StoreName storeName,
                                         X509FindType findType,
                                         string serviceCertificateName);

        public bool ImpersonateAll
        {get;set;}
        public string ApplicationName
        {get;set;}
        public bool UseAspNetProviders
        {get;set;}
    }
}
```

应用 SecurityBehavior 特性时, 我们需要为其提供一个 ServiceSecurity 枚举值形式的目标场景。我们可以使用 SecurityBehavior 特性的构造函数, 或者通过设置属性值。如果没有指定, 则所有属性的默认值则为适合目标场景的合理的值。选择场景时, 配置的行为要遵循我之前描述的每个场景对应的文字。SecurityBehavior 特性会产生一个可组合的安全模式, 允许进行修改或配置子场景。使用该特性时, 我们甚至可以使用不考虑安全性的宿主配置文件, 或者通过特性组合配置文件中值的设置。同样的, 我们的宿主代码也可以不考虑安全, 或者通过特性组合编程方式的宿主安全。

## 配置局域网服务

若要配置一个服务适用于局域网安全场景, 可以应用具有 ServiceSecurity.Intranet 值的 SecurityBehavior:

```
[ServiceContract]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}

[SecurityBehavior(ServiceSecurity.Intranet)]
class MyService : IMyContract
{
    public void MyMethod()
    {...}
}
```

即使服务契约没有限制保护级别,特性也可以通过编程方式添加对强制消息保护的需求。我们可以为基于角色安全使用 Windows NT 组:

```
[SecurityBehavior(ServiceSecurity.Intranet)]
class MyService : IMyContract
{
    [PrincipalPermission(SecurityAction.Demand, Role = @"<Domain>\Customer")]
    public void MyMethod()
    { ... }
}
```

服务能够通过编程方式模拟调用者,或者为单个方法的模拟使用操作行为特性。同样,我们也可以使用 ImpersonateAll 属性配置服务,使其能够自动模拟所有方法的所有调用者。ImpersonateAll 的默认值为 false,但是如果设置为 true,则 SecurityBehavior 特性直接就可以模拟所有操作的所有调用者,并不需要应用任何操作行为特性或者配置宿主:

```
[SecurityBehavior(ServiceSecurity.Intranet, ImpersonateAll = true)]
class MyService : IMyContract
{ ... }
```

### 配置互联网服务

在互联网场景下,我们需要在针对该场景进行配置的同时,选择要使用的服务证书。注意在例 10-19 中,ServiceBehavior 特性的构造函数可以接收服务证书的名称。如果没有指定,则服务证书从例 10-8 所示的宿主配置文件中装载:

```
[SecurityBehavior(ServiceSecurity.Internet)]
class MyService : IMyContract
{ ... }
```

我们也可以指定服务证书名称,此时,指定的证书根据名称从存储在 My 文件夹下的 LocalMachine 中装载:

```
[SecurityBehavior(ServiceSecurity.Internet, "MyServiceCert")]
class MyService : IMyContract
{ ... }
```

如果证书名称被设置为空字符串,则特性会使用托管机器名(或域)作为证书名称,然后通过该名称从存储在 My 文件夹下的 LocalMachine 中装载:

```
[SecurityBehavior(ServiceSecurity.Internet, "")]
class MyService : IMyContract
{ ... }
```

最后,特性允许开发者显式地指定存储的位置,存储名以及查找的方法:

NetTcpBinding, 以及在配置为 ASP.NET Providers 时允许局域网应用程序, 以避免使用 Windows 账户与组, 正如之前所阐释的那样。

接下来的问题是为 ASP.NET Providers 提供应用程序名, 它由 ApplicationName 属性负责。如果未分配应用程序名, 特性会从配置文件中查找应用程序名, 配制文件如例 10-13 和 10-14 所示。如果在宿主配置文件中没有找到值, 特性并不会将 *machine.config* 文件中没有丝毫意义的 / 作为默认值。相反, 它会默认地将宿主的程序集名作为应用程序名。如果 ApplicationName 属性分配了值, 就会覆盖宿主配置文件中配置的应用程序名:

```
[SecurityBehavior(ServiceSecurity.Internet,"MyServiceCert",
                  UseAspNetProviders = true,ApplicationName = "MyApplication")]
class MyService : IMyContract
{...}
```

## 配置 B2B 服务

为 B2B 场景进行配置需要将 ServiceSecurity 设置为 ServiceSecurity.BusinessToBusiness。特性为使用对等信任验证客户端证书。对服务证书的配置则与 ServiceSecurity.Internet 相同, 例如:

```
[SecurityBehavior(ServiceSecurity.BusinessToBusiness)]
class MyService : IMyContract
{...}

[SecurityBehavior(ServiceSecurity.BusinessToBusiness,"")]
class MyService : IMyContract
{...}

[SecurityBehavior(ServiceSecurity.BusinessToBusiness,"MyServiceCert")]
class MyService : IMyContract
{...}
```

在默认情况下, 如果为 ServiceSecurity.BusinessToBusiness, 特性会将宿主的 PrincipalPermissionMode 属性设置为 PrincipalPermissionMode.None, 此时服务无法授权它的调用者。但是, 将 UseAspNetProviders 属性设置为 true, 则允许使用 ASP.NET 角色 Providers, 就像例 10-18 那样:

```
[SecurityBehavior(ServiceSecurity.BusinessToBusiness,UseAspNetProviders = true)]
class MyService : IMyContract
{...}
```

如果使用 ASP.NET 角色 Provider, 则提供应用程序名以进行查询与判断, 就像使用 ServiceSecurity.Internet 时那样:

```
[SecurityBehavior(ServiceSecurity.BusinessToBusiness,"MyServiceCert",
                  UseAspNetProviders = true,ApplicationName = "MyApplication")]
```

```
class MyService : IMyContract
{...}
```

## 配置匿名服务

要允许匿名场景的调用者，我们需要将特性配置为 `ServiceSecurity.Anonymous`，对服务证书配置则与 `ServiceSecurity.Internet` 的配置相同；例如：

```
[SecurityBehavior(ServiceSecurity.Anonymous)]
class MyService : IMyContract
{...}

[SecurityBehavior(ServiceSecurity.Anonymous, "")]
class MyService : IMyContract
{...}

[SecurityBehavior(ServiceSecurity.Anonymous, "MyServiceCert")]
class MyService : IMyContract
{...}
```

## 配置无安全服务

要完全关闭安全，可以为特性配置 `ServiceSecurity.None`：

```
[SecurityBehavior(ServiceSecurity.None)]
class MyService : IMyContract
{...}
```

## 实现 SecurityBehaviorAttribute

例 10-20 部分展示了 `SecurityBehaviorAttribute` 的实现。

### 例 10-20：实现 SecurityBehaviorAttribute

```
[AttributeUsage(AttributeTargets.Class)]
class SecurityBehaviorAttribute : Attribute, IServiceBehavior
{
    SecurityBehavior m_SecurityBehavior;
    string m_ApplicationName = String.Empty;
    bool m_UseAspNetProviders;
    bool m_ImpersonateAll;

    public SecurityBehaviorAttribute(ServiceSecurity mode)
    {
        m_SecurityBehavior = new SecurityBehavior(mode);
    }
    public SecurityBehaviorAttribute(ServiceSecurity mode,
                                     string serviceCertificateName)
    {
        m_SecurityBehavior = new SecurityBehavior(mode, serviceCertificateName);
    }
}
```



```

public bool ImpersonateAll      // 访问 m_ImpersonateAll
{get;set;}
public string ApplicationName  // 访问 m_ApplicationName
{get;set;}
public bool UseAspNetProviders // 访问 m_UseAspNetProviders
{get;set;}

void IServiceBehavior.ApplyDispatchBehavior(...)
{}

void IServiceBehavior.AddBindingParameters(ServiceDescription description,
                                           ServiceHostBase serviceHostBase,
                                           Collection<ServiceEndpoint> endpoints,
                                           BindingParameterCollection parameters)
{
    m_SecurityBehavior.AddBindingParameters(description, serviceHostBase,
                                           endpoints, parameters);
}

void IServiceBehavior.Validate(ServiceDescription description,
                               ServiceHostBase serviceHostBase)
{
    m_SecurityBehavior.UseAspNetProviders = UseAspNetProviders;
    m_SecurityBehavior.ApplicationName = ApplicationName;
    m_SecurityBehavior.ImpersonateAll = ImpersonateAll;
    m_SecurityBehavior.Validate(description, serviceHostBase);
}
// 其余实现
}

```

特性有三个公有属性对应于保存配置值的三个私有成员。

SecurityBehaviorAttribute 是一个服务行为特性,因此我们可以直接将它应用到服务类上。当 IServiceBehavior 接口的 AddBindingParameters() 方法被调用时, SecurityBehaviorAttribute 会强制要求绑定的配置匹配请求的场景。IServiceBehavior 的 Validate() 方法指定了 SecurityBehaviorAttribute 配置宿主的位置。除此之外,特性还会对配置的整个顺序进行排序,以上功能基本上涵盖了该特性的所有工作。实际的配置会使用 SecurityBehavior 辅助类。特性构造了一个 SecurityBehavior 实例,提供的参数值为场景 (mode 参数) 以及与构造函数匹配的证书名。SecurityBehavior 通过编程调用提供了符合所有安全场景语义的设置。它封装了之前介绍的每个场景的确切步骤。SecurityBehavior 本身就具有服务行为的特征,因此它的设计是可以单独使用的,而不用依赖于 SecurityBehaviorAttribute 特性。例 10-21 列举了 SecurityBehavior 的部分实现,并演示了它的执行原理。

#### 例 10-21: 实现 SecurityBehavior (部分)

```

class SecurityBehavior : IServiceBehavior
{
    ServiceSecurity m_Mode;
    StoreLocation m_StoreLocation;
}

```

```

StoreName m_StoreName;
X509FindType m_FindType;
string m_SubjectName;
bool m_UseAspNetProviders;
string m_ApplicationName = String.Empty;
bool m_ImpersonateAll;

public SecurityBehavior(ServiceSecurity mode) :
    this(mode, StoreLocation.LocalMachine, StoreName.My,
        X509FindType.FindBySubjectName, null)
{
}

public SecurityBehavior(ServiceSecurity mode, StoreLocation storeLocation,
    StoreName storeName, X509FindType findType, string subjectName)
{ ... } // 设置对应的成员值

public bool ImpersonateAll      // 访问 m_ImpersonateAll
{get;set;}
public bool UseAspNetProviders // 访问 UseAspNetProviders
{get;set;}
public string ApplicationName  // 访问 m_ApplicationName
{get;set;}

public void Validate(ServiceDescription description,
    ServiceHostBase serviceHostBase)
{
    if(m_SubjectName != null)
    {
        switch(m_Mode)
        {
            case ServiceSecurity.Anonymous:
            case ServiceSecurity.BusinessToBusiness:
            case ServiceSecurity.Internet:
            {
                string subjectName;
                if(m_SubjectName != String.Empty)
                {
                    subjectName = m_SubjectName;
                }
                else
                {
                    subjectName = description.Endpoints[0].Address.Uri.Host;
                }
                serviceHostBase.Credentials.ServiceCertificate.
                    SetCertificate(m_StoreLocation, m_StoreName, m_FindType, subjectName);
                break;
            }
        }
    }
    .
    .
    .
}

public void AddBindingParameters(ServiceDescription description,
    ServiceHostBase serviceHostBase,

```

```

        Collection<ServiceEndpoint> endpoints,
        BindingParameterCollection parameters)
    {
        .
        .
        .
        switch(m_Mode)
        {
            case ServiceSecurity.Intranet:
            {
                ConfigureIntranet(endpoints);
                break;
            }
            case ServiceSecurity.Internet:
            {
                ConfigureInternet(endpoints, UseAspNetProviders);
                break;
            }
            .
            .
            .
        }
    }
}
internal static void ConfigureInternet(Collection<ServiceEndpoint> endpoints)
{
    foreach(ServiceEndpoint endpoint in endpoints)
    {
        Binding binding = endpoint.Binding;
        if(binding is WSHttpBinding)
        {
            WSHttpBinding wsBinding = (WSHttpBinding)binding;
            wsBinding.Security.Mode = SecurityMode.Message;
            wsBinding.Security.Message.ClientCredentialType =
                MessageCredentialType.UserName;
            continue;
        }
        .
        .
        .
        throw new InvalidOperationException(binding.GetType() +
            "is unsupported with ServiceSecurity.Internet");
    }
}
// 其余实现
}

```

SecurityBehavior的构造函数通过成员变量保存了构造参数,例如安全模式与证书细节。Validate()方法是一个决策树(Decision Tree),它根据场景与提供的信息对宿主进行配置,同时支持之前描述的SecurityBehaviorAttribute行为。AddBindingParameters()方法为每个场景调用了专门的辅助方法,用来配置宿主公开的终结点集

合。每个辅助方法（例如 `ConfigureInternet()`）遍历了服务终结点集合。对于每个终结点而言，它会验证使用的绑定是否匹配场景，然后根据场景对绑定进行配置。

## 宿主与声明式安全

通过 `SecurityBehavior` 特性，声明式安全的实现就变得唾手可得，它通常由宿主决定对安全的配置，而服务则专注于业务逻辑。此外，可能还需要托管我们还没有开发的服务，这些服务并不会使用我实现的声明式安全。接下来的步骤自然是将声明式安全添加到具有一系列 `SetSecurityBehavior()` 方法的 `ServiceHost<T>` 中：

```
public class ServiceHost<T> : ServiceHost
{
    public void SetSecurityBehavior(ServiceSecurity mode,
                                   bool useAspNetProviders,
                                   string applicationName,
                                   bool impersonateAll);

    public void SetSecurityBehavior(ServiceSecurity mode,
                                   string serviceCertificateName,
                                   bool useAspNetProviders,
                                   string applicationName,
                                   bool impersonateAll);

    // 更多成员
}
```

通过宿主使用声明式安全必须遵循与 `SecurityBehavior` 特性相同的一致性原则。例如，以下演示了如何为具有 ASP.NET Providers 的互联网安全配置宿主：

```
ServiceHost<MyService> host = new ServiceHost<MyService>();
host.SetSecurityBehavior(ServiceSecurity.Internet,
                        "MyServiceCert", true, "MyApplication", false);
host.Open();
```

例 10-22 演示了在 `ServiceHost<T>` 中声明式安全的部分实现。

例 10-22：为 `ServiceHost<T>` 添加声明式安全

```
public class ServiceHost<T> : ServiceHost
{
    public void SetSecurityBehavior(ServiceSecurity mode,
                                   string serviceCertificateName,
                                   bool useAspNetProviders,
                                   string applicationName,
                                   bool impersonateAll)
    {
        if (State == CommunicationState.Opened)
        {
            throw new InvalidOperationException("Host is already opened");
        }
    }
}
```

```

        SecurityBehavior securityBehavior = new
            SecurityBehavior(mode, serviceCertificateName);
        securityBehavior.UseAspNetProviders = useAspNetProviders;
        securityBehavior.ApplicationName = applicationName;
        securityBehavior.ImpersonateAll = impersonateAll;

        Description.Behaviors.Add(securityBehavior);
    }
    // 更多成员
}

```

实现 `SetSecurityBehavior()` 的前提是 `SecurityBehavior` 类支持 `IServiceBehavior` 接口。`SetSecurityBehavior()` 方法利用提供的参数初始化了 `SecurityBehavior` 类型的实例，然后将它添加到服务描述中的行为集合中，就好像服务被标记了 `SecurityBehaviorAttribute` 特性一样。

## 客户端声明式安全

WCF 不允许将特性应用到代理上，如果存在一个契约级的特性，客户端可能需要在运行时提供它的证书以及其他设置。要在客户端支持声明式安全，首先需要使用我编写的 `SecurityHelper` 静态辅助类，定义如例 10-23 所示。

例 10-23: `SecurityHelper` 辅助类

```

public static class SecurityHelper
{
    public static void UnsecuredProxy<T>(ClientBase<T> proxy) where T : class;
    public static void AnonymousProxy<T>(ClientBase<T> proxy) where T : class;
    public static void SecureProxy<T>(ClientBase<T> proxy,
        string userName, string password) where T : class;
    public static void SecureProxy<T>(ClientBase<T> proxy,
        string domain, string userName, string password) where T : class;
    public static void SecureProxy<T>(ClientBase<T> proxy, string domain,
        string userName, string password, TokenImpersonationLevel impersonationLevel)
        where T : class;
    public static void SecureProxy<T>(ClientBase<T> proxy,
        string clientCertificateName) where T : class;
    public static void SecureProxy<T>(ClientBase<T> proxy,
        StoreLocation storeLocation, StoreName storeName,
        X509FindType findType, string clientCertificateName) where T : class;
    // 更多成员
}

```

我们可以根据需要的安全场景与行为，通过使用 `SecurityHelper` 提供的专门的静态方法，配置一个普通的代理类。我们只能在打开代理之前配置代理。在客户端配置文件或者客户端代码的任何位置，都不需要设置任何安全。

SecurityHelper是智能的，它可以基于提供的参数以及调用的方法选择正确的安全行为。我们不需要显式地使用 ServiceSecurity 枚举。

例如，如下代码演示了如何为互联网场景保障代理的安全，以及为它提供客户端的 Windows 证书：

```
MyContractClient proxy = new MyContractClient();
SecurityHelper.SecureProxy(proxy, "MyDomain", "MyUsername", "MyPassword");
proxy.MyMethod();
proxy.Close();
```

对于互联网场景，客户端只需要提供用户名与密码（谨记，究竟是使用 Windows 证书还是 ASP.NET Provider 证书，应根据服务端而定）：

```
MyContractClient proxy = new MyContractClient();
SecurityHelper.SecureProxy(proxy, "MyUsername", "MyPassword");
proxy.MyMethod();
proxy.Close();
```

对于 B2B 场景，如果希望使用配置文件中的证书，客户端可以为客户端证书名指定一个 null 值或者一个空字符串，我们也可以显式地列出证书的名称：

```
MyContractClient proxy = new MyContractClient();
SecurityHelper.SecureProxy(proxy, "MyClientCert");
proxy.MyMethod();
proxy.Close();
```

SecurityHelper 会根据名称从存储在 My 文件夹中的客户端 LocalMachine 中，装载证书。客户端也可以指定它需要查找和装载证书的所有信息。在设计 SecurityHelper 时，为简便起见，如果在 B2B 场景中使用 BasicHttpBinding，客户端必须在配置文件中或者以编程方式显式地指定服务证书的位置。

对于一个匿名客户端，可以使用 AnonymousProxy() 方法：

```
MyContractClient proxy = new MyContractClient();
SecurityHelper.AnonymousProxy(proxy);
proxy.MyMethod();
proxy.Close();
```

如果完全不应用安全性，则调用 UnsecuredProxy() 方法：

```
MyContractClient proxy = new MyContractClient();
SecurityHelper.UnsecuredProxy(proxy);
proxy.MyMethod();
proxy.Close();
```

## 实现 SecurityHelper

在 SecurityHelper 内部使用了 SecurityBehavior 配置代理的终结点以及设置证书, 如例 10-24 所示。

例 10-24: 实现 SecurityHelper (部分)

```
public static class SecurityHelper
{
    public static void SecureProxy<T>(ClientBase<T> proxy,
                                     string userName, string password) where T : class
    {
        if(proxy.State == CommunicationState.Opened)
        {
            throw new InvalidOperationException("Proxy channel is already opened");
        }
        Collection<ServiceEndpoint> endpoints = new Collection<ServiceEndpoint>();
        endpoints.Add(proxy.Endpoint);

        SecurityBehavior.ConfigureInternet(endpoints);

        proxy.ClientCredentials.UserName.UserName = userName;
        proxy.ClientCredentials.UserName.Password = password;
        proxy.ClientCredentials.ServiceCertificate.Authentication.
            CertificateValidationMode = X509CertificateValidationMode.PeerTrust;
    }
    // 其余实现
}
```

## SecureClientBase<T> 类

使用 SecurityHelper 的优势在于它能够操作所有的代理, 即使该代理不由客户端开发者负责创建。缺陷在于客户端必须执行这一步骤。如果我们负责生成代理, 就应该利用我设计的 SecureClientBase<T> 类, 定义如例 10-25 所示。

例 10-25: SecureClientBase<T> 类

```
public abstract class SecureClientBase<T> : ClientBase<T> where T : class
{
    // 这些构造函数都使用默认的终结点
    protected SecureClientBase();
    protected SecureClientBase(ServiceSecurity mode);
    protected SecureClientBase(string userName, string password);
    protected SecureClientBase(string domain, string userName, string password,
                               TokenImpersonationLevel impersonationLevel);
    protected SecureClientBase(string domain, string userName, string password);
    protected SecureClientBase(string clientCertificateName);
    protected SecureClientBase(StoreLocation storeLocation,
                               StoreName storeName, X509FindType findType,
                               string clientCertificateName);
    // 更多的构造函数是针对其他类型的终结点
}
```



SecureClientBase<T>继承自传统的ClientBase<T>类,同时添加了对声明式安全的支持。我们需要将代理继承自 SecureClientBase<T>,而不是 ClientBase<T>,同时提供一个能够匹配安全场景的构造函数。在构造函数中通过提供的证书与终结点信息调用 SecureClientBase<T> 基类的构造函数:

```
class MyContractClient : SecureClientBase<IMyContract>,IMyContract
{
    public MyContractClient(ServiceSecurity mode) : base(mode)
    {}
    public MyContractClient(string userName,string password) :
                                                base(userName,password)
    {}

    /* 更多构造函数 */

    public void MyMethod()
    {
        Channel.MyMethod();
    }
}
```

可以直接使用继承的代理类。例如,对于互联网场景而言:

```
MyContractClient proxy = new MyContractClient("MyUsername","MyPassword");
proxy.MyMethod();
proxy.Close();
```

或者对于匿名场景:

```
MyContractClient proxy = new MyContractClient(ServiceSecurity.Anonymous);
proxy.MyMethod();
proxy.Close();
```

SecureClientBase<T>的实现只是使用了SecurityHelper(参见例10-26),因此 SecureClientBase<T> 遵循相同的行为,例如相关的客户端证书。

#### 例 10-26: 实现 SecureClientBase<T> (部分)

```
public class SecureClientBase<T> : ClientBase<T> where T : class
{
    protected SecureClientBase(ServiceSecurity mode)
    {
        switch(mode)
        {
            case ServiceSecurity.None:
            {
                SecurityHelper.UnsecuredProxy(this);
                break;
            }
            case ServiceSecurity.Anonymous:
            {
```

```

        SecurityHelper.AnonymousProxy(this);
        break;
    }
    ...
}
}
protected SecureClientBase(string userName, string password)
{
    SecurityHelper.SecureProxy(this, userName, password);
}
// 更多构造函数
}

```

## 安全通道工厂

如果不使用代理, `SecurityHelper` 与 `SecureClientBase<T>` 就没有什么用处了。为此, 我们编写了 `SecureChannelFactory<T>` 类, 定义如例 10-27 所示。

例 10-27: `SecureChannelFactory<T>` 类

```

public class SecureChannelFactory<T> : ChannelFactory<T>
{
    public SecureChannelFactory()
    {
    }
    public SecureChannelFactory(string endpointName) : base(endpointName)
    {
    }
    // 更多构造函数
    public void SetSecurityMode(ServiceSecurity mode);
    public void SetCredentials(string userName, string password);
    public void SetCredentials(string domain, string userName, string password,
                               TokenImpersonationLevel impersonationLevel);
    public void SetCredentials(string domain, string userName, string password);
    public void SetCredentials(string clientCertificateName);
    public void SetCredentials(StoreLocation storeLocation, StoreName storeName,
                               X509FindType findType, string clientCertificateName);
}

```

我们可以像 WCF 提供的通道工厂那样使用 `SecureChannelFactory<T>`, 不同的是我们还能够使用声明式安全。我们需要在打开通道之前调用 `SetSecurityMode()` 方法或者调用其中一个符合目标场景的 `SetCredentials()` 方法。例如, 将代理对象指向一个基于互联网安全的服务:

```

SecureChannelFactory<IMyContract> factory =
    new SecureChannelFactory<IMyContract>("");

factory.SetCredentials("MyUsername", "MyPassword");

IMyContract proxy = factory.CreateChannel();

using(proxy as IDisposable)
{

```

```

        proxy.MyMethod();
    }

```

SecureChannelFactory<T>的实现与SecurityHelper的实现相似,因此我们省略了它的实现细节。

## 双向客户端与声明式安全

我同样提供了SecureDuplexClientBase<T,C>类(与SecureClientBase<T>相似),定义如例 10-28。

例 10-28: SecureDuplexClientBase<T,C> 类

```

public abstract class SecureDuplexClientBase<T,C> : DuplexClientBase<T,C>
    where T : class
{
    protected SecureDuplexClientBase(C callback);
    protected SecureDuplexClientBase(ServiceSecurity mode,C callback);
    protected SecureDuplexClientBase(string userName,string password,C callback);
    protected SecureDuplexClientBase(string domain,string userName,string password,
        TokenImpersonationLevel impersonationLevel,C callback);
    protected SecureDuplexClientBase(string domain,string userName,string password,
        C callback);
    protected SecureDuplexClientBase(string clientCertificateName,C callback);
    protected SecureDuplexClientBase(StoreLocation storeLocation,
        StoreName storeName,X509FindType findType,
        string clientCertificateName,C callback);

    /* 更多使用 InstanceContext<C> 的构造函数以及
       使用配置的终结点和编程方式终结点的构造函数 */
}

```

SecureDuplexClientBase<T,C>派生自第5章介绍的类型安全的DuplexClientBase<T,C>类,它添加了对声明式基于场景的安全的支持。对于DuplexClientBase<T,C>类而言,我们需要让代理类继承SecureDuplexClientBase<T,C>,这样才能使用回调参数或者类型安全的上下文InstanceContext<C>。例如,给定这样的服务契约与回调契约的定义:

```

[ServiceContract(CallbackContract = typeof(IMyContractCallback))]
interface IMyContract
{
    [OperationContract]
    void MyMethod();
}
interface IMyContractCallback
{
    [OperationContract]
    void OnCallback();
}

```

继承的代理类就应该如下所示：

```
class MyContractClient :
    SecureDuplexClientBase<IMyContract,IMyContractCallback>,IMyContract
{
    public MyContractClient(IMyContractCallback callback) : base(callback)
    {}
    public MyContractClient(ServiceSecurity mode,IMyContractCallback callback)
                                                : base(mode,callback)
    {}
    /* 更多构造函数 */

    public void MyMethod()
    {
        Channel.MyMethod();
    }
}
```

使用代理类时，应提供安全场景或证书，以及回调对象与终结点信息。例如，当目标场景为匿名场景时：

```
class MyClient : IMyContractCallback
{...}

IMyContractCallback callback = new MyClient();

MyContractClient proxy = new MyContractClient(ServiceSecurity.Anonymous,callback);
proxy.MyMethod();

proxy.Close();
```

SecureDuplexClientBase<T,C>的实现几乎与SecureClientBase<T>相同，最主要的区别则在于它们的基类不同。

## SecureDuplexChannelFactory<T,C> 类

如果不使用继承自SecureDuplexClientBase<T,C>的代理去创建双向通信，则可以使用我们定义的SecureDuplexChannelFactory<T,C>通道工厂，定义如例10-29所示。

### 例 10-29: SecureDuplexChannelFactory<T,C> 类

```
public class SecureDuplexChannelFactory<T,C> : DuplexChannelFactory<T,C>
{
    where T : class

    public SecureDuplexChannelFactory(C callback)
    public SecureDuplexChannelFactory(InstanceContext<C> context,Binding binding) :
                                                base(context,binding)

    // 更多构造函数

    public void SetSecurityMode(ServiceSecurity mode);
    public void SetCredentials(string userName,string password);
```

```

    public void SetCredentials(string domain,string userName,string password,
                               TokenImpersonationLevel impersonationLevel);
    public void SetCredentials(string domain,string userName,string password);
    public void SetCredentials(string clientCertificateName);
    public void SetCredentials(StoreLocation storeLocation,StoreName storeName,
                               X509FindType findType,string clientCertificateName);
}

```

`SecureDuplexChannelFactory<T,C>` 派生自第 5 章定义的类型安全的 `DuplexChannelFactory<T,C>` 类。我们可以通过针对回调的泛型类型参数的实例（或者类型安全的 `InstanceContext<C>` 实例），创建 `SecureDuplexChannelFactory<T,C>` 对象。我们需要在打开通道之前调用 `SetSecurityMode()` 方法，或者调用其中一个符合目标场景的 `SetCredentials()` 方法。例如，当目标场景为互联网场景时：

```

class MyClient : IMyContractCallback
{
    ...

    IMyContractCallback callback = new MyClient();

    SecureDuplexChannelFactory<IMyContract,IMyContractCallback> factory =
        new SecureDuplexChannelFactory<IMyContract,IMyContractCallback>(callback,"");

    factory.SetCredentials("MyUsername","MyPassword");

    IMyContract proxy = factory.CreateChannel();
    using(proxy as IDisposable)
    {
        proxy.MyMethod();
    }
}

```

`SecureDuplexChannelFactory<T,C>` 的实现与 `SecureChannelFactory<T>` 的实现非常相似，主要的区别则在于它们的基工厂类。

## 安全审核

我将以介绍 WCF 提供的另一项有用的功能——安全审核作为本章的结束。正如它的名称所示，安全审核负责记录服务中与安全相关的事件。WCF 能够记录身份验证与授权的每一次尝试，它们发生的时间和位置，以及客户端的身份。`ServiceSecurityAuditBehavior` 类用于管理审核，我在例 10-30 中列出了它的定义以及它所支持的枚举类型。

例 10-30: `ServiceSecurityAuditBehavior` 类

```

public enum AuditLogLocation
{
    Default, // 根据操作系统而定
    Application,
    Security
}

```

```

public enum AuditLevel
{
    None,
    Success,
    Failure,
    SuccessOrFailure
}
public sealed class ServiceSecurityAuditBehavior : IServiceBehavior
{
    public AuditLogLocation AuditLogLocation
    {get;set;}
    public AuditLevel MessageAuthenticationAuditLevel
    {get;set;}
    public AuditLevel ServiceAuthorizationAuditLevel
    {get;set;}
    // 更多成员
}

```

ServiceSecurityAuditBehavior 是一个服务行为。AuditLogLocation 属性用于指定保存日志条目的保存地址，是保存在应用程序的日志文件中还是在安全日志中，它们两者都在宿主机上的事件日志里。MessageAuthenticationAuditLevel 属性用于设定验证审核的级别。出于对性能的考虑，我们可能希望只对失败事件进行审核，当然也可能同时审核成功事件及失败事件。为了实现诊断，我们也可以对成功的身份验证加以审核。MessageAuthenticationAuditLevel 的默认值为 AuditLevel.None。类似的，我们使用 ServiceAuthorizationAuditLevel 属性管理授权审计的级别，而且它也是默认被禁用的。

## 配置安全审核

通常我们通过修改宿主配置文件启用安全审核，我们只需要在文件中添加一个定制行为的配置节，并在服务的声明中引用它，如例 10-31 所示。

例 10-31：配置一个安全审核

```

<system.serviceModel>
  <services>
    <service name = "MyService" behaviorConfiguration = "MySecurityAudit" >
      ...
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name = "MySecurityAudit">
        <serviceSecurityAudit
          auditLogLocation = "Default"
          serviceAuthorizationAuditLevel = "SuccessOrFailure"
          messageAuthenticationAuditLevel = "SuccessOrFailure"
        />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>

```

```

        </serviceBehaviors>
    </behaviors>
</system.serviceModel>

```

我们同样可以在运行时向宿主（在打开宿主之前）添加该行为，从而以编码的方式实现对安全审核的配置。和通过编码方式添加其他行为类似，我们可以先检查宿主是否已经存在一个审核行为，以避免重写配置文件中的设置，如例 10-32 所示。

#### 例 10-32：以编码方式启用安全审核

```

ServiceHost host = new ServiceHost(typeof(MyService));

ServiceSecurityAuditBehavior securityAudit =
    host.Description.Behaviors.Find<ServiceSecurityAuditBehavior>();
if(securityAudit == null)
{
    securityAudit = new ServiceSecurityAuditBehavior();

    securityAudit.MessageAuthenticationAuditLevel = AuditLevel.SuccessOrFailure;
    securityAudit.ServiceAuthorizationAuditLevel = AuditLevel.SuccessOrFailure;
    host.Description.Behaviors.Add(securityAudit);
}
host.Open();

```

我们可以通过向 `ServiceHost<T>` 类添加一个 `Boolean` 类型的 `SecurityAuditEnabled` 属性简化例 10-32 的代码：

```

public class ServiceHost<T> : ServiceHost
{
    public bool SecurityAuditEnabled
    {get;set;}
    // 更多成员
}

```

在使用 `ServiceHost<T>` 类之后，例 10-32 则被简化为：

```

ServiceHost<MyService> host = new ServiceHost<MyService>();
host.SecurityAuditEnabled = true;
host.Open();

```

例 10-33 演示了 `SecurityAuditEnabled` 属性的实现。

#### 例 10-33：实现 `SecurityAuditEnabled` 属性

```

public class ServiceHost<T> : ServiceHost
{
    public bool SecurityAuditEnabled
    {
        get
        {
            ServiceSecurityAuditBehavior securityAudit =
                Description.Behaviors.Find<ServiceSecurityAuditBehavior>();
            if(securityAudit != null)

```



```

    {
        return securityAudit.MessageAuthenticationAuditLevel ==
                                   AuditLevel.SuccessOrFailure
            &&
            securityAudit.ServiceAuthorizationAuditLevel ==
                                   AuditLevel.SuccessOrFailure;
    }
    else
    {
        return false;
    }
}
set
{
    if(State == CommunicationState.Opened)
    {
        throw new InvalidOperationException("Host is already opened");
    }
    ServiceSecurityAuditBehavior securityAudit =
        Description.Behaviors.Find<ServiceSecurityAuditBehavior>();
    if(securityAudit == null && value == true)
    {
        securityAudit = new ServiceSecurityAuditBehavior();
        securityAudit.MessageAuthenticationAuditLevel =
                                   AuditLevel.SuccessOrFailure;
        securityAudit.ServiceAuthorizationAuditLevel =
                                   AuditLevel.SuccessOrFailure;
        Description.Behaviors.Add(securityAudit);
    }
}
}
// 更多成员
}

```

在 `SecurityAuditEnabled` 属性的 `get` 访问器中, 通过访问服务的 `description` 属性查找一个 `ServiceSecurityAuditBehavior` 类的实例。如果找到了这样的一个实例, 并且身份验证及授权的审核级别都被设置为了 `AuditLevel.SuccessOrFailure`, 那么 `SecurityAuditEnabled` 将返回 `true`, 反之返回 `false`。在 `SecurityAuditEnabled` 属性的 `set` 访问器中, 只有当服务描述中不存在一个由配置文件设置的行为时, 才会启用安全审核。如果没有发现优先级更高的行为, `SecurityAuditEnabled` 属性会将身份验证与授权两者的审核级别全部设为 `AuditLevel.SuccessOrFailure`。

## 声明式安全审核

我们还可以编写一个 .NET 特性将安全审核选项置于服务级别。我选择以一个 `Boolean` 类型属性的方式将这种支持添加到 `SecurityBehavior` 特性中, 该属性名为 `SecurityAuditEnabled`:

```
[AttributeUsage(AttributeTargets.Class)]
public class SecurityBehaviorAttribute : Attribute, IServiceBehavior
{
    public bool SecurityAuditEnabled
    {get;set;}
    // 更多成员
}
```

SecurityAuditEnabled的默认值为 false，即无安全审核。使用该属性使得声明式安全模型的功能更加完善，例如：

```
[SecurityBehavior(ServiceSecurity.Internet, UseAspNetProviders = true,
                  SecurityAuditEnabled = true)]
class MyService : IMyContract
{...}
```

例 10-34 演示了如何将这一支持添加到 SecurityBehavior 特性中。

#### 例 10-34：实现一个声明性安全审核

```
[AttributeUsage(AttributeTargets.Class)]
public class SecurityBehaviorAttribute : Attribute, IServiceBehavior
{
    bool m_SecurityAuditEnabled;

    public bool SecurityAuditEnabled // 访问 m_SecurityAuditEnabled
    {get;set;}

    void IServiceBehavior.Validate(ServiceDescription description,
                                   ServiceHostBase serviceHostBase)
    {
        if (SecurityAuditEnabled)
        {
            ServiceSecurityAuditBehavior securityAudit = serviceHostBase.Description.
                Behaviors.Find<ServiceSecurityAuditBehavior>();
            if (securityAudit == null)
            {
                securityAudit = new ServiceSecurityAuditBehavior();
                securityAudit.MessageAuthenticationAuditLevel =
                    AuditLevel.SuccessOrFailure;
                securityAudit.ServiceAuthorizationAuditLevel =
                    AuditLevel.SuccessOrFailure;
                serviceHostBase.Description.Behaviors.Add(securityAudit);
            }
            // 其余内容与例 10-20 相同
        }
    }
    // 其余实现
}
```

IServiceBehavior中的Validate()方法使用与ServiceHost<T>相同的审核等级，同时也避免了重写配置文件中的设置。



# 面向服务概述

本书全面介绍了使用 WCF 设计与开发面向服务应用程序的相关知识。附录 A 则展示了我对面向服务的理解，以及面向服务的具体应用场景。但是，如果要了解面向服务的发展方向以及它在软件行业所占的地位，首先就要了解它的起源与发展，因为没有任何一种新的方法学是一蹴而就的，而应该是经历了数十年渐进的演化历程。在简要地介绍了软件工程的发展历程以及发展趋势之后，附录给出了面向服务应用程序的定义（不仅仅是指纯粹的架构），阐释了服务的本质，以及从方法学角度所体现出来的价值。接着，附录还给出了面向服务的设计原则，其中增加的抽象原则对于大多数应用程序而言，具有更强的实践意义与具体应用。

## 软件工程简史

20 世纪 20 年代末期，世界上第一台现代计算机诞生于波兰，这是一台电子机械，大约与打字机一般大小，主要用于消息的加密。后来，这台设备被卖给了德国商业部。30 年代，它被德国军方用来实现通信加密，也就是闻名遐迩的“英格玛 (Enigma)”。Enigma 使用机械转子 (Mechanical Rotors) 根据不同的密码字母改变从键到灯板的电流路径。Enigma 并非通常意义上的计算机：它只能实现加密 (Enciphering) 与解密 (Deciphering) (现在，我们将它们称之为 Encryption 与 Decryption)。如果操作者要改变加密算法，必须通过改变转子的顺序、初始位置，以及连接键盘到灯板的线缆，改变机器的机械结构。这就导致“程序”与它要解决的问题（加密）紧密地耦合在一起，是采用机械设计的计算机。

在上个世纪 40 年代末期到 50 年代期间，终于诞生了世界上第一台真正意义上的电子计算机，它主要用于国防。这些机器能够运行代码解决问题，但无法执行预先制定的任务。这类计算机执行的代码存在的硬伤，则在于“语言”是面向机器的，因而程序完全依赖

于硬件。针对一台机器编写的代码无法运行在另外一台机器上。最初，这个缺陷并没有引起足够的重视，因为在当世只有屈指可数的几台计算机。随着计算机的大量生产，在60年代初出现了汇编语言，它使得代码能够不依赖于特定的机器，可以运行在多种机器上。但是，代码却与机器的体系架构密切相关。针对8位机编写的代码不能运行在16位机上，更不用说寄存器或内存以及内存地址之前的区别。因此，维护程序的代价开始逐步增加。随着计算机被广泛的应用在民用以及政府部门，为满足有限的资源与预算，需要提供更好的解决方案。

60年代，诸如COBOL和FORTRAN等高级语言引入了编译器的概念。开发者可以在抽象层面上编写机器编程语言，编译器能够将它编译为实际的汇编代码。编译器开启了将代码与硬件以及硬件架构解耦的先河。第一代语言存在的问题是代码是非结构化编程的，代码内部通过使用跳转指令或go-to语句依赖于它自身的结构。即使代码结构发生微小的改变，也可能对程序的多个地方产生灾难性的影响。

在七十年代，结构化编程语言例如C和Pascal占据了统治地位。它通过函数与结构，完全解除了代码与内部地址及内部结构的依赖。正是在七十年代，开发者与研究者首次开始将软件作为工程科学进行研究。在所属权利益的驱动下，许多公司开始考虑软件的重用，即代码段能够重用在不同的程序上下文中。例如C语言，基本的重用单元就是函数。基于函数重用的问题是函数依赖于它操作的数据，如果数据是全局的，在重用上下文中改变一个函数，就会影响不同地方使用的其他函数。

## 面向对象

上述问题的解决方案是引入面向对象，例如Smalltalk，以及之后产生的C++。面向对象语言将函数和函数操作的数据包裹在一起，放到一个对象中。函数（现在则称为方法）封装逻辑，对象则封装数据。面向对象通过类层级的形式以支持领域建模（Domain Modeling）。重用机制是基于类的，允许直接重用，或者通过继承（Inheritance）进行特化（Specialization）。但是，面向对象仍然存在自身的问题。首先，生成的应用程序（或伪代码）是单一的应用程序。类似C++的编程语言并不能识别二进制形式的生成代码。即使只是针对细微的修改，开发者每一次都必须重新部署大量的代码。这对开发过程、质量、发布时间以及成本都会产生负面影响。由于类作为重用的基本单元，这些单元会在源代码中被定义为类的格式。因此，应用程序会依赖于它使用的语言。我们不能让一个Smalltalk编写的客户端去调用或继承C++的类。而且，继承实际上是一种糟糕的重用机制，大多数情况下，它都是弊大于利，因为派生类与基类的实现密切相关，从而在类层级中引入了垂直的依赖关系。面向对象忽略了许多现实问题，例如部署与版本控制。序列化与持久化则是存在的另一个问题。大多数应用程序都无法凭空获取对象。对象包含了某些持久化状态，这些状态需要组合为对象，但却无法保证持久状态与可能的新的

对象代码的兼容性。如果对象被跨进程或跨机器分发,就无法使用C++的调用方式,因为C++需要直接内存引用,并不支持分布式调用。开发者必须编写宿主进程,使用一些远程调用技术例如TCP套接字执行远程调用,但这样的调用迥异于通常的C++调用方式,从而抵消了C++语言的优势。

## 面向组件

随着时间的推移,相继产生了一些新的技术,例如静态库(.lib)与动态库(.dll),它们能够解决面向对象存在的问题。终于,在1994年人们首次提出面向组件技术,称为COM(组件对象模型)。面向组件提供了可交换的、可互操作的二进制组件。与共享源代码文件不同,客户端与服务器都支持二进制类型系统(例如IDL),以元数据的表示方式放入到封装的二进制组件中。组件在运行时被发现以及装载,例如拖动一个控件到窗体上,则该控件会在客户端机器的运行时自动被装载。客户端程序仅仅是服务的抽象与契约,称为接口。只要接口不变,服务就能够任意扩展。代理能够实现相同的接口,通过为远程调用封装底层机制实现无缝的远程调用。公共二进制类型系统的可用性使得跨语言的互操作性成为可能,这样,Visual Basic的客户端就能够调用C++的COM组件。重用的基本单元是接口,而不是组件,多态的实现是可互换的。通过为每个接口、COM对象以及类型库分配唯一的标识符可以解决版本冲突的问题。然而,作为现代软件工程学的一个根本性突破,COM在大多数开发者的眼中却如鸡肋一般,食之无味,弃之可惜。COM未必是丑陋的实现,因为它能够与操作系统顶层结合在一起,而操作系统却不用考虑COM的实现。编写COM组件所使用的最佳语言(例如C++和Visual Basic)是面向对象的,而不是面向组件的。因为面向组件语言的编程模型过于复杂,需要框架(如ATL)来消除两种模型之间的鸿沟。正是认识到这一问题,微软于2002年发布了.NET 1.0。.NET对比COM、C++以及Windows,不仅更加简洁,而且还能够无缝地与单独的、新的面向组件运行时集成。.NET支持COM的所有优势,并实现了许多技术要素,例如类型元数据共享、序列化以及版本的统一与标准化。.NET具有更强的功能与COM协作,但COM与.NET又都存在相似的问题:

### 技术与平台

应用程序与代码依赖于技术与平台。COM与.NET都只能应用于Windows平台。它们要求客户端以及服务也应该是COM或者.NET,而无法支持与其他技术的互操作,不管它们是在Windows平台下,还是其他操作系统。虽然利用Web服务使得技术之间的互操作成为可能,但它却要求开发者放弃使用本地框架进行实现的大部分优势,从而引入了复杂性。

### 并发管理

当开发商(Vendor)发布一个组件时,并不能假定该组件不会被它的客户端多线程的并发访问。事实上,唯一安全的假设就是开发商要求组件支持多线程访问。因此,



组件必须是线程安全的,同时必须包含一个同步锁。如果应用程序的开发者在构建应用程序时,聚集了多个开发商开发的多个组件,则多个锁的引入就会使得应用程序易于死锁。必须避免死锁与应用程序和组件之间的依赖。

#### 事务

如果应用程序希望组件只参与到一个单独的事务中,则需要运行组件的应用程序协调事务以及组件之间的事务流,这是一个严格的编程要求。它同样会引入应用程序与组件之间关于事务协调的耦合。

#### 通信协议

如果组件被跨进程或跨机器边界部署,则组件将依赖于远程调用、传输协议以及编程模型要素(例如可靠性与安全性)的实现细节。

#### 通信模式

组件可以被同步或异步调用,也能够联机或离线调用。一个组件能够以上述的各种方式调用,但应用程序必须知道准确的选项。

#### 版本控制

在编写应用程序时,可能会使用一个版本的组件,而在发布产品时使用另一个版本的组件。处理版本冲突的问题会导致应用程序依赖于它所使用的组件。

#### 安全

组件需要对它们的调用者进行验证与授权。那么,组件如何才能知道它所使用的安全权限,以及用户所对应的角色?不仅如此,组件还需要确保来自客户端的通信是安全的,而对客户端施加确定的限制会反过来增加组件与组件安全之间的耦合度。

COM 与 .NET 都试图采用一些技术解决上述提及的部分(不是全部)问题,例如 COM+ 以及企业服务(相似的,Java 使用 J2EE),但事实上,这些应用程序都被淹没在大量的公共基础功能实现中。在正常规模的应用程序中,大量的工作、开发以及调试时间都花费在实现这样的公共基础功能上,而不是关注业务逻辑与特性。事情更加糟糕的是,终端用户(或者开发经理)很少去关注这些公共基础功能(与业务特性相对),而开发者却没有足够的时间去开发健壮的公共基础功能。而且,大多数公共基础功能的解决方案都是专有的(这意味着无法重用、迁移与租赁)、低劣的,因为大部分开发者都不是安全专家,也不是同步处理专家,开发者也没有时间与资源专门开发这些公共基础功能。

## 面向服务

如果我们仔细阅读了刚才概述的软件工程发展简史,就会注意到这样一个模式:每个新的方法学与技术都会融合前一代技术的优点,并致力于改善前一代技术的缺陷。然而,



每个新产生的技术又会面临新的挑战。我这里所谓的现代软件工程，就是对过去技术的去芜存菁，降低耦合程度。

不同的是，耦合虽然糟糕，它却是不可避免的。一个绝对解耦的应用程序毫无用处，因为它不具有任何价值。开发者只能通过耦合其他内容，才能为系统添加职责。编写代码的行为就是将两个内容关联起来。真正的问题是耦合的范围究竟有多宽。我相信世上只存在两种类型的耦合。好的耦合仅限于业务层的耦合。开发者通过实现系统用例或特性，将软件的功能结合起来，完成对职责的添加。坏的耦合则将所有的内容都集成在一起。.NET与COM存在的问题，不是概念上的错误，而是基于开发者必须编写大量的公共基础功能这一事实。

正是认识到过去的问题，在2000年末，面向服务方法学作为应对面向对象以及面向组件缺陷的解决方案呈现在人们眼前。在面向服务的应用程序中，开发者只需要关注于业务逻辑的编写，以及通过可交换的、可互操作的服务终结点暴露业务逻辑。客户端调用这些终结点，而不是服务代码或者它的实现包。客户端与服务终结点的交互基于标准的消息交换，服务发布各种标准元数据，描述服务的功能，以及客户端调用服务操作的方式。元数据就是服务，相当于C++的头文件，COM的类型库，或者.NET程序集的元数据。服务的终结点是可重用的，在交互的约束（例如同步、事务以及安全通信）下，服务是与客户端兼容的，而与客户端的实现技术无关。

从多个角度来看，服务都是组件的一个本质上的飞跃，就像组件是对象的一个本质上的飞跃一样。在软件行业中，面向服务是我们目前所知的构建可维护的、健壮的以及安全的应用程序的最佳方案，也是最可行的方案。

在开发面向服务应用程序时，我们能够实现服务代码与客户端使用的技术与平台的解耦，也与并发管理、事务传播和管理以及通信可靠性、协议和模式无关。总的来讲，实现从客户端到服务的消息传递的安全，就是对调用者的认证，它属于服务范围之外。服务根据需求仍然要实现服务自身的本地授权。在大多数情况下，客户端并不知道服务的版本：只要终结点支持客户端期望访问的契约，客户端就不用考虑服务的版本。为了处理客户端与服务之间传递数据的版本兼容，面向服务同时还构建了版本兼容的标准。

## 面向服务的价值

由于客户端与服务之间的交互是基于行业标准的，这个行业标准包括了保障调用安全的方式、传播事务流的方式以及管理可靠性的方式等等。我们也可以使用现有的这些公共基础功能的实现。这就保证了应用程序的可维护性，因为应用程序在正确性方面是完全脱耦的。即使公共基础功能发生演化，应用程序也不会受到影响。面向服务的应用程序是健壮的，因为开发者能够使用可用的、已验证的、通过测试的公共基础功能。同时也

提高了开发者的效率，因为他们可以将更多的时间投入到功能特性的实现，而不是这些公共基础功能。面向服务的真正价值就是：允许开发者从代码中抽取出公共基础功能的实现，更多地关注业务逻辑和需要的功能特性。

面向服务还包括许多广受欢迎的价值，例如跨技术的互操作性，就是核心价值的体现。虽然不借助于服务，我们也能够实现互操作性，但直到面向服务的诞生，才能够应用到实践中。两者的区别在于后者能够通过已有的公共基础功能为开发者提供互操作性。编写服务时，通常不用考虑客户端执行在什么平台上，因为面向服务完全实现了无缝的互操作性。面向服务应用程序所能提供的不仅仅是互操作性，它还允许系统跨越边界。其中一种边界就是技术与平台的边界，跨越这样的边界则完全体现了互操作性。但是，边界可能还存在于客户端与服务之间，例如安全与信任边界、地域边界、组织边界、时区边界、事务边界，甚至是业务模型边界。无缝地跨越这些边界是可能的，原因在于基于消息的交互标准。例如，保障消息安全的标准，建立客户端与服务安全交互的标准，即使交互双方存在于不具有直接信赖关系的域（或站点）中。事务标准允许客户端的事务管理器将事务传递到服务端的事务管理器，并让服务参与到事务中，即使两个事务管理器从来没有直接登记彼此的事务。

## 面向服务应用程序

一个面向服务应用程序只是简单地将服务组合到一个单一逻辑的、整体的应用程序（参见图 A-1）中，这类似于聚合了对象的面向对象应用程序。

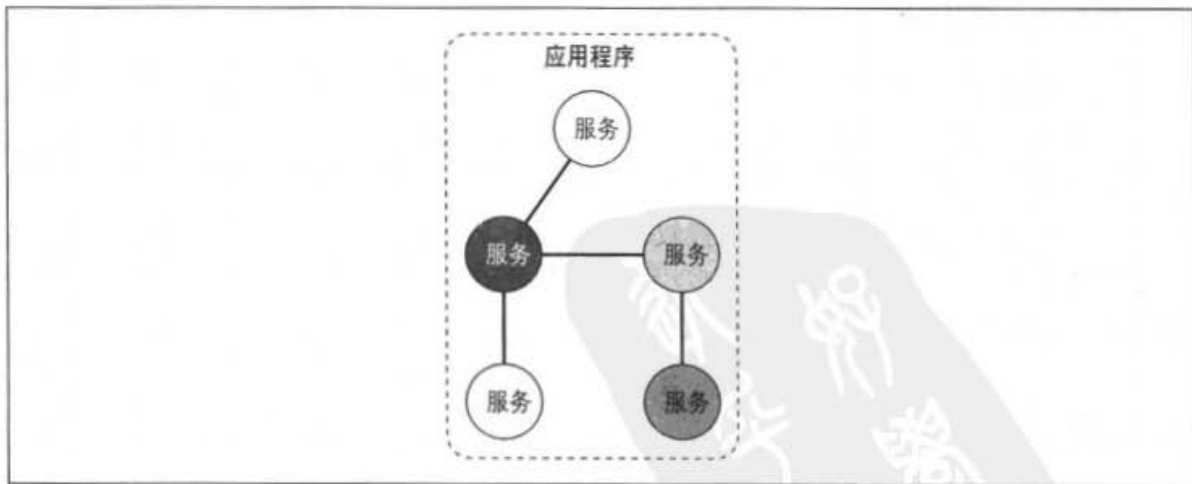


图 A-1：面向服务应用程序

应用程序自身可以将组合服务公开为新的服务，就好像一个对象可以由多个小的对象组成一样。

在服务内部，开发者仍然使用传统编程的概念，例如特定的编程语言，版本，技术与框架，操作系统，API等。但是，服务之间则必须使用标准的消息与协议、契约以及元数据交换。

应用程序中的不同服务全部可以放到相同的位置上，或者分布放到局域网或互联网上。它们也可以来自于多个开发商，使用各种不同的技术与平台进行开发，版本独立，甚至执行在不同的时区。所有的这些公共基础功能特性对于在应用程序中与服务交互的客户端而言，都是隐藏的。客户端发送标准消息到服务，两端的公共基础功能通过消息以及与平台无关的传输型表示形式进行转换，并对客户端与服务之间存在的区别实现封送(Marshal)。

既然面向服务框架为了将服务连接在一起，提供了现有的公共基础功能，那么服务的粒度越小，就越能够有助于应用程序对这些基础设施的使用，开发者所要编写的公共基础功能就越少。在极端的情况下，每一个基本的类都应该是服务，以便于最大程度地利用现有的互联性，避免编码实现公共基础功能。理论上讲，它能够轻松地实现事务型整数，安全字符串以及可靠的类。然而实际上，粒度过于细化会影响到使用的框架(例如WCF)的性能。我相信，面向服务技术会随着时日的发展而逐步演进，服务边界的内聚性会越来越强，服务的粒度会越来越小，甚至于每个基本的构建模块都可以成为服务。显然，这是历史的发展趋势，那就是通过方法学的改进以及抽象，以性能换取效率的提高。

## 要素与原则

面向服务方法学负责管理服务之间所发生的内容(参见图A-1)。它有一套设计原则与最佳实践，用于构建面向服务应用程序，称为面向服务架构原则：

### 服务边界是明确的

任何服务总是被限制在边界例如实现技术和分布位置之后。服务公开的契约与数据类型不会将它的实现技术与分布位置透露给客户端，从而隐藏了这些边界的本质。坚持这一原则可以使得服务与位置和技术无关。不管是以何种方式思考这一原则，它所表达的思想就是客户端知道服务的实现越多，则客户端与服务的耦合度就越高。要减小潜在的耦合度，服务就必须明确地公开它的功能，而且只有操作(或数据契约)才会被明确地被公开给客户端共享。服务的其余内容会被封装起来。面向服务技术采用了默认为“否决(Opt-Out)”的编程模型，公开的内容则被明确标记为参与(Opt-In，译注1)。

译注1：Opt-Out与Opt-In本身属于发送广告中的两种不同行为与授权方式。在这里，Opt-Out指的是如果服务的成员没有明确地进行设置，则默认是不暴露的，即否决机制。Opt-In指的是只有明确标记了需要暴露的成员，则该成员才会参与到服务中，能够被跨越服务边界调用。

### 服务是自治的

服务无需获取它的客户端或其他服务的内容。服务的运行与版本应该与客户端无关。这一原则允许服务脱离客户端单独演化。服务的安全也是独立的,它能够保护服务自身以及传递的消息,而不用考虑客户端使用的安全级别。这样做(除了尽人皆知的常识之外)同时也能够解除客户端与服务安全之间的耦合。

### 服务共享操作契约与数据样式,而不是类型与特定技术的元数据

服务要做的就是决定公开在服务边界之外的内容应与技术无关。服务能够将本地的数据类型转换为某种与技术无关的表示形式,而不是共享本地的、特定技术的内容,例如程序集版本号或者它的类型。此外,服务应该禁止客户端知道本地的实现细节,例如实例管理模式或并发管理模式。服务只公开逻辑操作。服务对于操作的实现方式以及执行方式,对于客户端而言是不透明的。

### 服务与策略保持一致

服务应该发布一种策略,指示它所能完成的内容以及客户端与服务交互的方式。策略所体现的访问约束(例如可靠通信)不应依赖于服务的实现细节。并非所有的客户端都能与所有的服务交互。这种不兼容性是完全有效的,它能够防止特殊的客户端访问服务。发布的策略是客户端决定它们能否与服务交互的唯一方法,同时不应有任何的带外机制让客户端做出这样的决策。不同的是,服务必须能够在策略的标准表达形式中,表示服务能够执行的内容以及客户端能够与之通信的方式。如果无法表示,就意味着服务的设计是拙劣的。注意,如果服务是私有的(即不是公有服务),那么实际上它可能不会发布任何策略。这一原则暗示如果服务需要,就应该能够发布策略。

## 实用原则

前面列举的原则是非常抽象的,对它们的支持主要体现在开发、调用以及设计服务的技术方面。因此,应用程序可能会不同程度地遵循这些原则,正如开发者可以在C++中编写非面向对象的代码那样。然而,精心设计的应用程序应该力图坚持这些原则。因此,我又补充了一些更加实用的原则:

### 服务是安全的

服务与它的客户端必须使用安全通信。至少,从客户端传递到服务的消息必须是安全的,客户端必须具备验证服务的方法。同时,客户端可能会在消息中提供它们的安全证书,这样服务才能够对它们进行授权与认证。

### 服务在系统中应保持一致的状态

执行客户端请求时,禁止进行部分替换的条件。服务访问的所有资源在客户端调用之后必须是一致的。服务不能有任何剩余内容作为错误的结果,例如部分地影响系

统状态。服务不应寻求它的客户端的帮助，在发生错误后，服务会将系统恢复为一致的状态。

#### 服务是线程安全的

服务必须设计为线程安全，才能够维持多线程的并发访问。服务同样能够处理因果关系或逻辑线程的重入。

#### 服务是可靠的

如果客户端调用服务，客户端总是能够以确定的方式获知消息是否被服务接收。消息应该按照发送的顺序处理，而不是接收的顺序。

#### 服务是健壮的

服务与它的错误分离能够防止错误影响服务本身或其他服务。服务不能要求客户端根据服务遇到的错误类型改变它们的行为。这能够有助于客户端与错误处理层面上的服务解耦。

## 可选原则

我们可以将实用原则看作是强制原则，同时还有一套可选原则，这些原则并非所有应用程序所必需的，虽然坚持这些原则通常是一个不错的主意：

#### 服务是互操作的

设计的服务应该能够被任意的客户端调用，而不用考虑客户端的技术。

#### 服务的规模是不变的

不管客户端有多少，也不管服务的承载是多少，服务代码都应该相同。随着系统的发展，这样的设计才能够极大地降低维护服务的成本，服务也能够支持不同的部署场景。

#### 服务是可用的

服务总是能够接收客户端的请求，而不会因此停止。如果服务不可用，则意味着客户端需要解决服务的问题，反过来就会引入耦合。

#### 服务是及时响应的

服务开始处理客户端的请求时，不能让客户端等待太久。如果服务不能及时响应，则意味着客户端需要解决服务的问题，反过来就会引入耦合。

#### 服务是受限的

服务执行的任意操作应尽可能短，不能消耗太多时间去处理客户端的请求。长时间的处理过程意味着客户端需要解决服务的问题，反过来就会引入耦合。



# 发布-订阅服务

针对事件使用原来的双向回调通常会引入发布者与订阅者的高度耦合。订阅者必须知道所有发布的服务在应用程序的位置，并连接它们。订阅者不能识别的发布者无法通知事件的订阅者。如果增加新的订阅者（或者移除已经存在的订阅者）就会给已经部署了的应用程序带来困难。无论什么时候，对于应用程序的任何人发出的事件的一个特定类型，订阅者都无法要求获得通知。此外，订阅者必须为每个发布者发出多个昂贵的调用，不管是订阅还是取消订阅。不同的发布者可能会触发相同的事件，但却为订阅者和取消订阅提供了略微不同的方法，自然而然给订阅者与相关方法带来耦合。

大致相同的是，发布者只能通知它知道的订阅者。无论是谁，如果希望接收事件，发布者都无法将消息传递给它，也没有能力广播事件。此外，所有的发布者都必须包含必要的代码，管理订阅者列表以及自身的发布行为。这些代码几乎与服务要解决的业务问题无关，如果还要提供一些高级特性，例如并发发布，就会增加相当大的复杂度。

而且，基于双向的回调同样会引入发布者与订阅者生命周期的耦合度。为了订阅和接收事件，必须运行订阅者。

订阅者无法询问事件是否被触发，而应用程序则需要创建一个订阅者的实例，让它处理该事件。

安全性则代表了另外一种耦合：订阅者需要通过各种安全模式以及使用的证书，以具备验证所有发布者的能力。同时，发布者也需要具有足够的安全证书，从而允许触发事件，不同的发布者可能具有不同的角色成员机制。

最后，必须以编程方式设置订阅信息。我们很难通过管理方式在应用程序中配置订阅信息，或者在系统运行时，改变订阅者的选项。

这些问题实际上不是 WCF 双向调用所特有的，过去的技术例如 COM 连接点或者 .NET 委托同样具有这样的特性。所有这些都属于紧密耦合的事件管理机制。

## 发布-订阅设计模式

若要解决以上提及的问题，可以使用已知的发布-订阅设计模式对它们进行设计。该模式所隐藏的含义很简单：通过引入一个专门的订阅服务，以及一个专门的发布服务，解除发布者与订阅者之间的耦合，如图 B-1 所示。

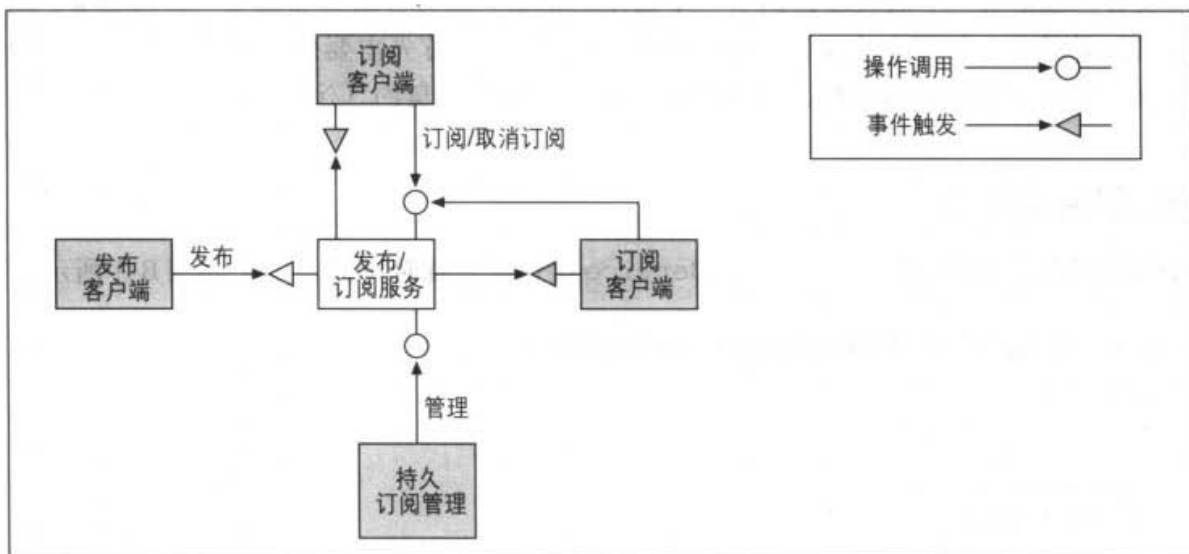


图 B-1：一个发布-订阅系统

需要订阅事件的订阅者注册订阅服务，该服务负责管理订阅者列表，同时为取消订阅提供了相似的功能。同样，所有发布者均使用发布服务触发它们的事件，避免将事件直接传递给订阅者。订阅和发布服务提供了一个间接层，从而解除了与系统之间的耦合。订阅者不需要了解发布者的身份。它们能够订阅事件类型，以及接收任何发布者的事件，并且订阅机制对于所有订阅者都是统一的。事实上，发布者不需要管理任何订阅列表，也不用关心订阅者是谁。它们会将事件传递给发布服务，然后再传递给需要事件的订阅者。

### 订阅者类型

我们甚至可以定义两种类型的订阅者：临时订阅者是在内存中运行的订阅者；持久订阅者则是持久化到磁盘的订阅者，它们代表了服务对发生事件的调用时间。对于临时订阅者，可以使用双向回调机制，通过它将回调引用传递到正在运行的服务。持久订阅者则需要将订阅者地址当作引用进行记录。当事件发生时，发布服务会调用持久订阅者的地



址，然后将事件传递给它。两种订阅类型还有另外一个显著区别，就是我们可以将持久订阅者存储在磁盘或数据库中。这样就能够在关闭应用程序或机器崩溃或重启的时候，持久化订阅者。这一过程允许以管理方式对订阅进行配置。显然，在关闭应用程序时，我们不能存储临时订阅，而需要在每次应用程序启动时，明确地创建临时订阅。

## 发布-订阅框架

本书附带的源代码中包含了完整的发布-订阅实例。我并不只是提供了发布-订阅服务以及客户端的实例，而且还提供了一个通用的框架，能够自动实现发布-订阅服务，以及增加对所有应用程序的支持。若要构建这样的框架，首先需要分解管理发布-订阅的接口，然后为临时订阅和持久订阅以及发布提供单独的契约（注1）。

### 管理临时订阅

可以使用我定义的 `ISubscriptionService` 接口管理临时订阅，定义如例 B-1 所示。

例 B-1：管理临时订阅者的 `ISubscriptionService` 接口

```
[ServiceContract]
public interface ISubscriptionService
{
    [OperationContract]
    void Subscribe(string eventOperation);

    [OperationContract]
    void Unsubscribe(string eventOperation);
}
```

注意，`ISubscriptionService` 接口无法识别实现了它所期待的终结点的回调契约。作为一个通用的接口，它与特定的回调契约无关。定义这些回调契约取决于如何使用应用程序。通过继承 `ISubscriptionService` 接口，可以在应用程序中提供回调接口，并指定所需的回调契约：

```
interface IMyEvents
{
    [OperationContract(IsOneWay = true)]
    void OnEvent1();

    [OperationContract(IsOneWay = true)]
    void OnEvent2(int number);

    [OperationContract(IsOneWay = true)]
    void OnEvent3();
}
```

注1：在 MSDN 杂志 2006 年 10 月刊中，作者在《WCF Essentials: What You Need to Know About One-Way Calls, Callbacks, and Events》这篇文章中首次提出发布-订阅框架。

```

        void OnEvent3(int number,string text);
    }

    [ServiceContract(CallbackContract = typeof(IMyEvents))]
    interface IMySubscriptionService : ISubscriptionService
    {}

```

通常,回调契约的每个操作都对应于特定的事件。ISubscriptionService的子接口(在本例中为IMySubscriptionService接口)不需要添加操作。ISubscriptionService接口提供了临时订阅的管理功能。每次调用Subscribe()或Unsubscribe()方法时,订阅者需要提供它需要订阅或取消订阅的操作名(以及事件名)。如果调用者希望订阅所有的事件,则传递一个空字符串,或者null值。

我设计的框架提供了ISubscriptionService接口方法的实现,形式为SubscriptionManager<T>泛型抽象类:

```

public abstract class SubscriptionManager<T> where T : class
{
    public void Subscribe(string eventOperation);
    public void Unsubscribe(string eventOperation);
    // 更多成员
}

```

SubscriptionManager<T>的泛型类型参数为事件契约。注意,SubscriptionManager<T>并没有实现ISubscriptionService接口。

应用程序需要以终结点形式暴露自己的临时订阅服务,该终结点需要支持ISubscriptionService接口的特定子接口。为此,应用程序需要提供派生自SubscriptionManager<T>的服务类,并将回调契约指定为类型参数,同时还要实现ISubscriptionService接口的子接口。例如,使用IMyEvent回调接口实现一个临时订阅服务:

```

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MySubscriptionService : SubscriptionManager<IMyEvents>, IMySubscriptionService
{}

```

MySubscriptionService的实现无需任何代码,因为IMySubscriptionService接口不会添加任何新的操作,而SubscriptionManager<T>已经实现了ISubscriptionService接口的方法。

注意,仅仅继承SubscriptionManager<IMyEvents>是不够的,因为它没有派生自契约接口。我们必须添加对IMySubscriptionService的继承,才可以支持临时订阅。

最后,应用程序需要定义IMySubscriptionService的终结点:

```

<services>
  <service name = "MySubscriptionService">
    <endpoint
      address = "..."/>
      binding = "..."/>
      contract = "IMySubscriptionService"/>
    </service>
  </services>

```

例 B-2 演示了 SubscriptionManager<T> 管理临时订阅的方式。

例 B-2: 在 SubscriptionManager<T> 中管理临时订阅者

```

public abstract class SubscriptionManager<T> where T : class
{
    static Dictionary<string, List<T>> m_TransientStore;

    static SubscriptionManager()
    {
        m_TransientStore = new Dictionary<string, List<T>>();
        string[] methods = GetOperations();
        Action<string> insert = delegate(string methodName)
        {
            m_TransientStore.Add(methodName, new List<T>());
        };
        Array.ForEach(methods, insert);
    }
    static string[] GetOperations()
    {
        MethodInfo[] methods = typeof(T).GetMethods(BindingFlags.Public |
                                                    BindingFlags.FlattenHierarchy |
                                                    BindingFlags.Instance);
        List<string> operations = new List<string>(methods.Length);

        Action<MethodInfo> add = delegate(MethodInfo method)
        {
            Debug.Assert(!operations.Contains(method.Name));
            operations.Add(method.Name);
        };
        Array.ForEach(methods, add);
        return operations.ToArray();
    }
    static void AddTransient(T subscriber, string eventOperation)
    {
        lock(typeof(SubscriptionManager<T>))
        {
            List<T> list = m_TransientStore[eventOperation];
            if(list.Contains(subscriber))
            {
                return;
            }
            list.Add(subscriber);
        }
    }
}

```

```
static void RemoveTransient(T subscriber, string eventOperation)
{
    lock(typeof(SubscriptionManager<T>))
    {
        List<T> list = m_TransientStore[eventOperation];
        list.Remove(subscriber);
    }
}

public void Subscribe(string eventOperation)
{
    lock(typeof(SubscriptionManager<T>))
    {
        T subscriber = OperationContext.Current.GetCallbackChannel<T>();
        if(String.IsNullOrEmpty(eventOperation) == false)
        {
            AddTransient(subscriber, eventOperation);
        }
        else
        {
            string[] methods = GetOperations();
            Action<string> addTransient = delegate(string methodName)
            {
                AddTransient(subscriber, methodName);
            };
            Array.ForEach(methods, addTransient);
        }
    }
}

public void Unsubscribe(string eventOperation)
{
    lock(typeof(SubscriptionManager<T>))
    {
        T subscriber = OperationContext.Current.GetCallbackChannel<T>();
        if(String.IsNullOrEmpty(eventOperation) == false)
        {
            RemoveTransient(subscriber, eventOperation);
        }
        else
        {
            string[] methods = GetOperations();
            Action<string> removeTransient = delegate(string methodName)
            {
                RemoveTransient(subscriber, methodName);
            };
            Array.ForEach(methods, removeTransient);
        }
    }
}

// 更多成员
}
```

SubscriptionManager<T>在泛型的静态字典对象m\_TransientStore中存储了临时订阅者:

```
static Dictionary<string,List<T>> m_TransientStore;
```

字典中包含了事件操作名以及以链表形式组成的所有订阅者。SubscriptionManager<T>的静态构造函数使用反射获得了回调契约(SubscriptionManager<T>的类型参数)的所有操作,并初始化了字典对象,让所有的操作都包含了一个空的链表。Subscribe()方法从操作调用的上下文中抽取出了回调引用。如果调用指定了一个操作名,Subscribe()方法会调用辅助方法AddTransient()。AddTransient()从存储中获取了事件的订阅者列表。如果列表没有包含该订阅者,则添加它。

如果调用者为操作名指定了空字符串或者null,Subscribe()方法则在回调契约中为每个操作调用AddTransient()方法。

Unsubscribe()方法的执行方式相似。注意,调用者能够订阅所有事件,然后取消其中一个的订阅。

## 管理持久订阅者

我所定义的IPersistentSubscriptionService接口可以管理持久订阅者,定义如例B-3所示。

例 B-3: 管理持久订阅者的 IPersistentSubscriptionService 接口

```
[ServiceContract]
public interface IPersistentSubscriptionService
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void Subscribe(string address,string eventsContract,string eventOperation);

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void Unsubscribe(string address,string eventsContract,string eventOperation);
    // 更多成员
}
```

调用者若要添加一个持久订阅者,需要调用Subscribe()方法,调用时需要提供订阅者的地址、事件的契约名以及指定的事件操作自身。若要取消订阅,则可以提供相同的信息,然后调用Unsubscribe()方法。注意,IPersistentSubscriptionService接口并没有指定订阅者持久化在服务端的哪个地方,因为这属于实现细节。

之前介绍的类SubscriptionManager<T>同样可以实现IPersistentSubscriptionService接口的方法:

```
[BindingRequirement(TransactionFlowEnabled = true)]
public abstract class SubscriptionManager<T> where T : class
{
    public void Unsubscribe(string address,string eventsContract,
                                                                    string eventOperation);
    public void Subscribe(string address,string eventsContract,
                                                                    string eventOperation);
    // 更多成员
}
```

SubscriptionManager<T>在SQL Server中存储了持久订阅者。它的配置使用了Client/Service事务模式(参见第7章的内容),它要求该模式使用我编写的BindingRequirement特性。

SubscriptionManager<T>的泛型类型参数为事件契约。注意,SubscriptionManager<T>并没有继承IPersistentSubscriptionService接口。应用程序需要公开它自己的持久订阅服务,但是不需要继承IPersistentSubscriptionService的新契约,因为它不需要回调引用。应用程序只需要继承SubscriptionManager<T>,并将事件契约指定为类型参数,同时继承IPersistentSubscriptionService接口;例如:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MySubscriptionService : SubscriptionManager<IMyEvents>,
                            IPersistentSubscriptionService
{
}
```

MySubscriptionService类的实现无需编写任何代码,因为SubscriptionManager<T>已经实现了IPersistentSubscriptionService接口的方法。

注意,仅仅继承SubscriptionManager<IMyEvents>是不够的,因为它没有继承一个契约接口。我们必须添加对IPersistentSubscriptionService的实现,才可以支持持久订阅。

最后,应用程序需要定义一个IPersistentSubscriptionService的终结点:

```
<services>
  <service name = "MySubscriptionService">
    <endpoint
      address = "... "
      binding = "... "
      contract = "IPersistentSubscriptionService"
    />
  </service>
</services>
```

IPersistentSubscriptionService接口的方法通过SubscriptionManager<T>实现,如例B-4所示。例B-4与例B-2非常相似,但订阅者是被存储到SQL Server中,而不是存储在内存的字典对象中。

## 例 B-4: 在 SubscriptionManager&lt;T&gt; 中管理持久订阅者

```

public abstract class SubscriptionManager<T> where T : class
{
    static void AddPersistent(string address,string eventsContract,
                                string eventOperation)
    {
        // 使用 ADO.NET 在 SQL Server 中存储订阅
    }

    static void RemovePersistent(string address,string eventsContract,
                                string eventOperation)
    {
        // 使用 ADO.NET 将订阅从 SQL Server 中移除
    }

    [OperationBehavior(TransactionScopeRequired = true)]
    public void Unsubscribe(string address,string eventsContract,
                                string eventOperation)
    {
        if(String.IsNullOrEmpty(eventOperation) == false)
        {
            RemovePersistent(address,eventsContract,eventOperation);
        }
        else
        {
            string[] methods = GetOperations();
            Action<string> removePersistent = delegate(string methodName)
            {
                RemovePersistent(address,eventsContract,methodName);
            };
            Array.ForEach(methods,removePersistent);
        }
    }

    [OperationBehavior(TransactionScopeRequired = true)]
    public void Subscribe(string address,string eventsContract,
                                string eventOperation)
    {
        if(String.IsNullOrEmpty(eventOperation) == false)
        {
            AddPersistent(address,eventsContract,eventOperation);
        }
        else
        {
            string[] methods = GetOperations();
            Action<string> addPersistent = delegate(string methodName)
            {
                AddPersistent(address,eventsContract,methodName);
            };
            Array.ForEach(methods,addPersistent);
        }
    }
    // 更多成员
}

```



如果针对相同的事件契约，应用程序需要同时支持临时订阅者和持久订阅者，可以让订阅服务类直接继承 `ISubscriptionService` 接口的子接口，以及 `IPersistentSubscriptionService` 接口：

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MySubscriptionService : SubscriptionManager<IMyEvents>,
                             IMySubscriptionService, IPersistentSubscriptionService
{ }
```

同时公开两个与之匹配的终结点：

```
<services>
  <service name = "MySubscriptionService">
    <endpoint
      address = "..."/>
    </service>
  </services>
```

## 事件发布

目前介绍的发布-订阅框架只包含处理订阅管理的一部分。该框架同样可以简化发布服务的实现。发布服务必须支持与订阅者相同的事件契约，这是获知应用程序的发布者的唯一连接点。因为发布服务在端点中暴露了事件契约，我们需要将事件契约标记为服务契约。假定我们只是通过双向回调使用临时订阅者：

```
[ServiceContract]
interface IMyEvents
{
    [OperationContract(IsOneWay = true)]
    void OnEvent1();

    [OperationContract(IsOneWay = true)]
    void OnEvent2(int number);

    [OperationContract(IsOneWay = true)]
    void OnEvent3(int number, string text);
}
```

发布-订阅框架包含了辅助类 `PublishService<T>`，定义如下：

```
public abstract class PublishService<T> where T : class
{ }
```

```
protected static void FireEvent(params object[] args);
}
```

PublishService<T> 定义了一个类型参数，它的类型为事件契约。若要提供开发者自己的发布服务，可以继承 PublishService<T>，然后使用 FireEvent() 方法将事件传递给所有订阅者，不管它们是临时订阅者还是持久订阅者，如例 B-5 所示。

#### 例 B-5：实现一个事件发布服务

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyPublishService : PublishService<IMyEvents>, IMyEvents
{
    public void OnEvent1()
    {
        FireEvent();
    }
    public void OnEvent2(int number)
    {
        FireEvent(number);
    }
    public void OnEvent3(int number, string text)
    {
        FireEvent(number, text);
    }
}
```

注意，我们可以使用 FireEvent() 触发任意类型的事件，而不用考虑参数的个数，因为我们使用了 params object 数组。

最后，应用程序需要为包含了事件契约的发布服务公开一个终结点：

```
<services>
  <service name = "MyPublishService">
    <endpoint
      address = "..."
      binding = "..."
      contract = "IMyEvents"
    />
  </service>
</services>
```

例 B-6 演示了 PublishService<T> 的实现。

#### 例 B-6：实现 PublishService<T>

```
public abstract class PublishService<T> where T : class
{
    protected static void FireEvent(params object[] args)
    {
        StackFrame stackFrame = new StackFrame(1);
        string methodName = stackFrame.GetMethod().Name;
        // 解析显式接口实现
        if (methodName.Contains("."))
```

```

    {
        string[] parts = methodName.Split('.');
        methodName = parts[parts.Length-1];
    }
    FireEvent(methodName, args);
}
static void FireEvent(string methodName, params object[] args)
{
    PublishPersistent(methodName, args);
    PublishTransient(methodName, args);
}
static void PublishPersistent(string methodName, params object[] args)
{
    T[] subscribers = SubscriptionManager<T>.GetPersistentList(methodName);
    Publish(subscribers, true, methodName, args);
}
static void PublishTransient(string methodName, params object[] args)
{
    T[] subscribers = SubscriptionManager<T>.GetTransientList(methodName);
    Publish(subscribers, false, methodName, args);
}
static void Publish(T[] subscribers, bool closeSubscribers, string methodName,
                    params object[] args)
{
    WaitCallback fire = delegate(object subscriber)
    {
        Invoke(subscriber as T, methodName, args);
        if(closeSubscribers)
        {
            using(subscriber as IDisposable)
            {
            }
        }
    };
    Action<T> queueUp = delegate(T subscriber)
    {
        ThreadPool.QueueUserWorkItem(fire, subscriber);
    };
    Array.ForEach(subscribers, queueUp);
}
static void Invoke(T subscriber, string methodName, object[] args)
{
    Debug.Assert(subscriber != null);
    Type type = typeof(T);
    MethodInfo methodInfo = type.GetMethod(methodName);
    try
    {
        {
            methodInfo.Invoke(subscriber, args);
        }
    }
    catch(Exception e)
    {
        Trace.WriteLine(e.Message);
    }
}
}

```

若要通过发布服务简化事件的触发过程, `FireEvent()` 方法需要接受多个参数, 并将它们传递给订阅者。而且, 发布服务的调用者并不提供订阅者调用的操作名。要实现这一目的, `FireEvent()` 方法会访问它的堆栈帧, 以获取它正在调用的方法名。然后使用重载版本的 `FireEvent()`, 它会接收方法名。该方法紧接着会调用辅助方法 `PublishPersistent()`, 发布给所有的持久订阅者, 以及调用 `PublishTransient()` 辅助方法, 发布给所有的临时订阅者。两种发布方法的实现几乎完全相同: 它们通过访问 `SubscriptionManager<T>` 以获取各自的订阅者列表, 然后使用 `Publish()` 方法触发事件。订阅者以订阅者代理数组的形式返回。该数据会被传递给 `Publish()` 方法。

采用这种办法, `Publish()` 可以简单地调用订阅者。但是, 我还需要支持事件的并发发布, 如果能够这样, 即使订阅者在处理事件时耗费时间过长, 仍然不会影响其他订阅者能够及时地接收事件。注意, 将事件操作标记为单向并不能保证异步调用。此外, 在事件操作没有被标记为单向操作时, 我还需要支持并发发布。`Publish()` 定义了两个匿名方法。第一个匿名方法调用了 `Invoke()` 辅助方法, 通过 `Invoke()` 方法触发事件, 并传递到提供的单独的订阅者。如果 `SubscriptionManager<T>` 指定要求关闭代理, 则匿名方法还会关闭代理。由于 `Invoke()` 方法不可能调用编译后的指定的订阅者类型, 因此需要使用反射对调用进行迟绑定。同时, `Invoke()` 方法还会禁止调用抛出的任何异常, 因为这些都与发布方所需要关注的。第二个匿名方法则对第一个匿名方法委托对象进行排队, 以便于线程池中的线程执行。最后, `Publish()` 对方法提供的 `subscribers` 数组中的每个订阅者执行第二个匿名方法的操作。

注意, `PublishService<T>` 对待订阅者是一视同仁的, 它并没有区分订阅者是临时的, 还是持久的。唯一的区别是在发布到持久订阅者之后, 需要关闭代理。当然不一致的地方还包括获取订阅列表的方法, 分别为 `GetTransientList()` 和 `GetPersistentList()`。两者之中, `GetTransientList()` 更简单:

```
public abstract class SubscriptionManager<T> where T : class
{
    internal static T[] GetTransientList(string eventOperation)
    {
        lock(typeof(SubscriptionManager<T>))
        {
            if(m_TransientStore.ContainsKey(eventOperation))
            {
                List<T> list = m_TransientStore[eventOperation];
                return list.ToArray();
            }
            return new T[]{};
        }
    }
    // 更多成员
}
```

GetTransientList()根据指定的操作查找临时存储的所有订阅者,然后以数组形式返回。GetPersistentList()面临更大的挑战:框架没有现成的持久订阅者的代理列表;我们所能获知的只是它们的地址。因此,GetPersistentList()需要实例化持久订阅者的代理,如例B-7所示。

#### 例B-7: 创建持久订阅者的代理列表

```
public abstract class SubscriptionManager<T> where T : class
{
    internal static T[] GetPersistentList(string eventOperation)
    {
        string[] addresses = GetSubscribersToContractEventOperation(
            typeof(T).ToString(), eventOperation);

        List<T> subscribers = new List<T>(addresses.Length);

        foreach(string address in addresses)
        {
            Binding binding = GetBindingFromAddress(address);
            T proxy = ChannelFactory<T>.CreateChannel(binding,
                new EndpointAddress(address));
            subscribers.Add(proxy);
        }
        return subscribers.ToArray();
    }
    static string[] GetSubscribersToContractEventOperation(string eventsContract,
        string eventOperation)
    {
        // 使用ADO.NET在SQL Server中查询事件的订阅者
    }
    static Binding GetBindingFromAddress(string address)
    {
        if(address.StartsWith("http:") || address.StartsWith("https:"))
        {
            WSHttpBinding binding = new WSHttpBinding(SecurityMode.Message,true);
            binding.ReliableSession.Enabled = true;
            binding.TransactionFlow = true;
            return binding;
        }
        if(address.StartsWith("net.tcp:"))
        {
            NetTcpBinding binding = new NetTcpBinding(SecurityMode.Message,true);
            binding.ReliableSession.Enabled = true;
            binding.TransactionFlow = true;
            return binding;
        }
        /* 命名管道绑定与MSMQ绑定与上面的代码相似 */
        Debug.Assert(false,"Unsupported protocol specified");
        return null;
    }
    // 更多成员
}
```

若要为每个订阅者创建代理，GetPersistentList()方法需要订阅者的地址、绑定和契约。当然，契约正好就是SubscriptionManager<T>的类型参数。为获取地址，GetPersistentList()方法调用了GetSubscribersToContractEventOperation()查询数据库，然后返回那些订阅了特定事件的持久订阅者的所有地址的数组。现在，GetPersistentList()方法所需要的就是每个订阅者的绑定。为此，GetPersistentList()调用了辅助方法GetBindingFromAddress()，它可以根据地址样式推断出使用的绑定。GetBindingFromAddress()将所有的HTTP地址当作WSHttpBinding。此外，GetBindingFromAddress()还为每个绑定实现了可靠性以及事务传播。如果没有使用单向操作，则允许在发布者的事务中包含事件，例如这样的事件契约：

```
[ServiceContract]
interface IMyEvents
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void OnEvent1();

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void OnEvent2(int number);

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void OnEvent3(int number, string text);
}
```

## 管理持久订阅者

如果我们要在运行时添加和移除持久订阅，可以使用例B-3所示的IPersistentSubscriptionService接口的方法，因为它们持久的本质，管理订阅最合适的方式就是通过某种管理工具。为了这一目的，IPersistentSubscriptionService定义了额外的操作，以支持对订阅者存储的各种查询。

### 例 B-8: IPersistentSubscriptionService 接口

```
[DataContract]
public struct PersistentSubscription
{
    [DataMember]
    public string Address;

    [DataMember]
    public string EventsContract;

    [DataMember]
    public string EventOperation;
}
```



```
[ServiceContract]
public interface IPersistentSubscriptionService
{
    // 管理操作
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    PersistentSubscription[] GetAllSubscribers();

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    PersistentSubscription[] GetSubscribersToContract(string eventsContract);

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    string[] GetSubscribersToContractEventType(string eventsContract,
                                                string eventOperation);

    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    PersistentSubscription[] GetAllSubscribersFromAddress(string address);
    // 更多成员
}
```

所有的这些管理操作都使用了一个简单的数据结构PersistentSubscription,它包含了订阅者的地址、契约和事件。GetAllSubscribers()方法只是返回所有订阅者的列表。GetSubscribersToContract()方法则返回对应于特定契约的所有订阅者,而GetSubscribersToContractEventType()方法则返回对应于特定契约的特定事件操作的所有订阅者。最后,出于完整性的考虑,GetAllSubscribersFromAddress()根据提供的指定地址,返回符合条件的所有订阅者。我设计的发布-订阅框架包含了一个简单的持久订阅管理工具,名为持久订阅管理器(Persistent Subscription Manager),如图B-2所示。

管理工具使用IPersistentSubscriptionService添加和移除订阅。要添加新的订阅,需要为它提供事件契约定义的元数据交换地址。我们可以使用持久订阅者自己的元数据交换地址,也可以使用发布服务的元数据交换地址(例如例B-5中定义的发布服务),因为它们是多态的。在MEX地址文本框中输入元数据交换基地址,然后单击Lookup按钮。工具会以编码方式获取事件服务的元数据,生成Contract和Event组合框。我们可以使用第2章介绍的MetadataHelper获取元数据,解析它的内容。

如果要订阅,则提供持久订阅者的地址,然后单击Subscribe按钮。持久订阅管理器会调用订阅服务(在实例中为MySubscriptionService服务)添加订阅。订阅服务的地址由持久订阅管理器的配置文件维护。



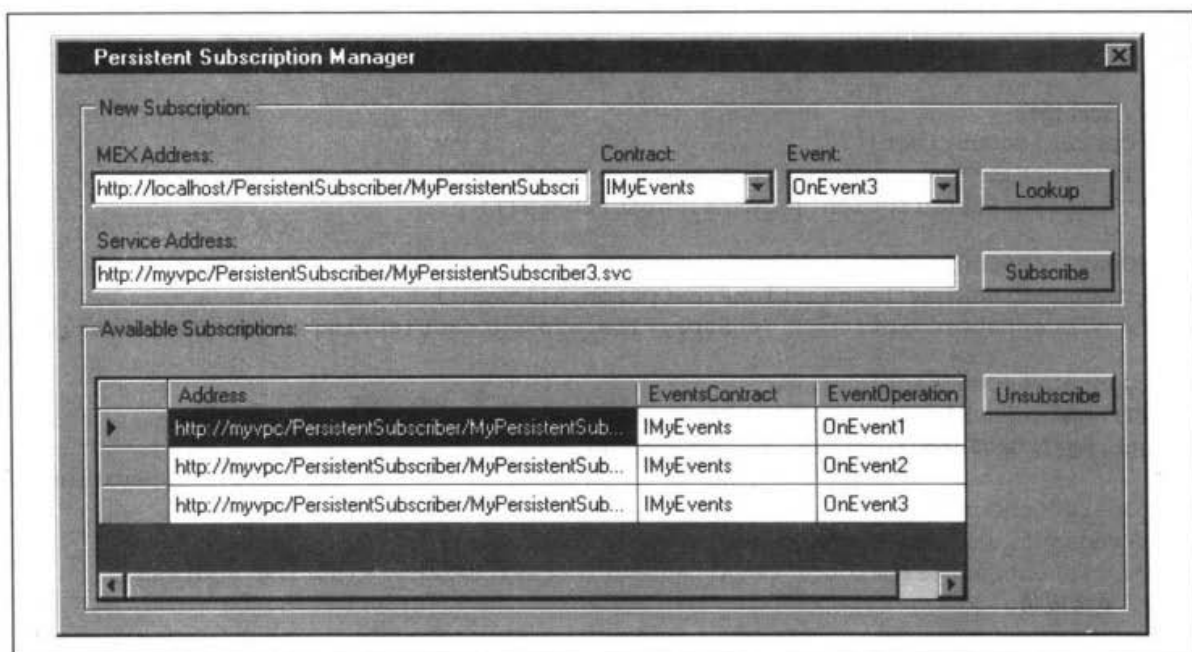


图 B-2：持久订阅管理器应用程序

**注意：**发布-订阅模式同样可以解除系统安全的耦合。相对于多个订阅者以及多个潜在的安全机制，所有的发布者都需要针对一个单独的发布者认证它们自身。接着，订阅者只需要允许发布服务将它们传递给事件，而不是系统的所有发布者，因为发布者信任发布服务会对发布者进行妥善地授权与认证。对于发布服务而言，如果应用基于角色的安全机制，可以允许开发者在一个地方轻易地应用多个角色，以授权支持跨系统发布一个事件。

## 队列发布者与队列订阅者

如果发布事件或订阅事件时不使用同步绑定，可以使用 `NetMsmqBinding`。一个队列发布-订阅服务组合了松散耦合系统的优势，以及离线执行 (`Disconnected Execution`) 的灵活性。当然，使用队列事件时，契约的所有事件需要被标记为单向操作。如图 B-3 所示，我们可以让队列的两端保持独立。

我们可以使用一个队列发布者和联机的同步订阅者，也可以让一个联机的发布者发布给队列订阅者，或者同时为队列发布者与队列订阅者。然而需要注意的是，我们不能使用队列的临时订阅，因为它不支持 MSMQ 绑定下的双向回调，从而无法使用通信的断开特性。如前所示，我们也可以使用管理工具管理订阅者，管理的操作仍然是联机的，以及同步的。

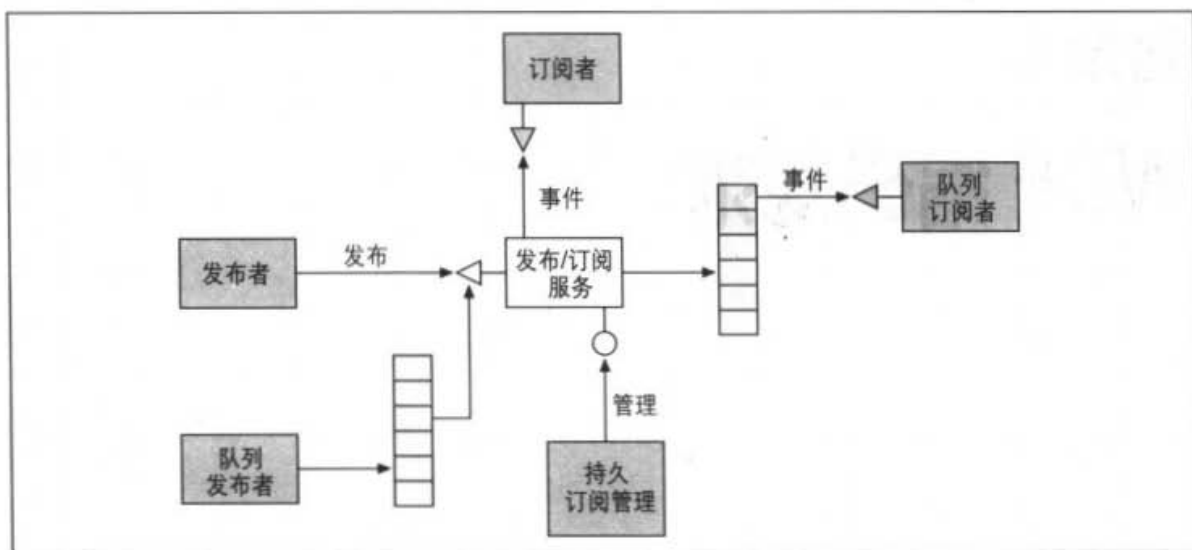


图 B-3: 队列发布-订阅

### 队列发布者

如果要使用一个队列发布者，那么在发布服务时，需要使用MSMQ绑定暴露一个队列终结点。当队列发布者触发事件时，发布服务可以为离线状态，或者发布的客户端自身可以是断开的。注意，当发布两个事件到队列发布服务中时，并不能保证它们传递的顺序，以及终端订阅者对这些事件的处理。只有当事件契约被配置为一个会话服务时，并且只有在处理一个单独的发布服务时，我们才能够假定发布的顺序。

### 队列订阅者

要部署一个队列订阅者，持久订阅服务需要暴露一个队列终结点。即使发布者为联机状态，这样做也能够使得它处于离线状态。当订阅者再次联机时，就会收到所有排队等候的事件。此外，当发布服务自身为断开状态时，队列订阅者也能够处理，因为并没有事件被丢失。在一个单独的队列订阅者中，如果触发多个事件，无法保证事件传递的顺序。如果事件契约拥有一个会话，则订阅者只能假定发布的顺序。当然，如果同时为队列发布者与队列订阅者，则允许它们在同一时间可以脱机工作。

## 附录 C

# WCF 编码规范

一个全面完整的编码规范是成功交付产品的根本。规范有助于推行最佳实践以及避免缺陷，可以让团队成员更容易分享知识与技能。传统的编码规范是一本浩如烟海一般的高文大册，厚达数百页，详尽了每个指南的基本原理。制定这样的规范固然是有胜于无，然而，如此努力却很难被大多数开发者所接受。比较而言，本书介绍的 WCF 编码规范却只有薄薄的几篇，主要介绍了 WCF 编码细节、编码内容以及设计目的。我相信，若要充分理解各种编程决策，可能需要阅读大量书籍，积累数年的经验，然而如果要实施编码规范则不必如此。当吸收一名新兵加入你的团队时，你可以交给他（她）这份规范，告诉他（她）：“先看看这个。”若要完全理解以及认识到规范的价值，或许需要时间与经验，然而，如果只是需要在此之前就能够遵循规范的约定，却是一蹴而就的事情。编码规范详细介绍了编码要求、缺陷、指南以及建议。编码规范同时还使用了本书介绍的最佳实践以及辅助类。

## 通用设计指南

1. 所有的服务必须遵循以下原则：
  - a. 服务是安全的。
  - b. 服务在系统中应保持状态一致。
  - c. 服务是线程安全的。
  - d. 服务可以被并发客户端访问。
  - e. 服务是可靠的。
  - f. 服务是健壮的。

