序

对于分布式系统，或者说业界不断提及的互联系统的设计与构建，我与本书作者 Juval L 鼾y 可谓志同道合。我们经历了相似的技术历程，虽然我们效力于不同的公司，负责不同的项目，工作在不同的地方，但我们却有着共同的目标。

20 世纪 90 年代早期，我们开始了对一种新技术理念的探索，即实现计算机之间的通信与交互。这种被称为分布式系统应用程序的平台技术也逐渐为世人所了解。随着工作站与服务器硬件的逐渐普及，经济因素不再成为制约发展的瓶颈，构建不依赖于单事务网络中心的大型系统就成为了技术热点。对于大范围的数据交换系统而言，同样如此。在过去，我的电话公司如果要求每秒钟传递超过 1200 位的数据几乎是不可能的，而在如今看来连这都达不到简直不可思议。在同样的线路上，今天的传输速度已经达到了 6Mbit/s。这真是一个激动人心的时代啊。

随着分布式计算技术的逐渐成熟，在 90 年代早期分属两大阵营的大型分布式系统技术渐露峥嵘，即数字设备公司（最终被康柏兼并，并入惠普）主导的 DCE 技术，以及 OMG 组织（主要由 IBM 支持）倡导的 CORBA 技术。然而在 1996~1997 期间，所有这些杰出的工程学成果却突然停滞不前。因为此时是互联网的世界，整个世界都疯迷于 HTML、HTTP、风险投资以及 IPO（Initial Public Offerings，首次公开募股）。整个行业花费了整整 10 年的时间才逐渐从泡沫经济带来的崩溃中恢复过来。不仅是经济的复苏，技术的发展也重新走回正轨。随之获益的是分布式系统技术由此打破了过去由两大阵营各占半壁江山的局面，多达数十种新的分布式技术如雨后春笋一般展现在人们眼前，使我们拥有了更多的抉择权。

直到 2007 年，整个行业仍然在为分布式系统的正确编码方式争论不休。Sun 公司或者 BEA 力主 Java；而我在微软的同事（包括我）则坚定地主张 C#或者 Visual Basic 才是最佳的实现方式。无论是 Sun、BEA、IBM 还是微软，都希望机器之间的通信标准能够达成一致。试想昔日的 DCE 与 CORBA 之争，正是因为达成了一致的标准规范才为如今的 SOAP 1.1 奠定了基础，从而开创了分布式技术的盛大场面。

自从 SOAP 1.1 作为技术说明（Technical Note）被提交给 W3C，到现在已有超过 6 年的历史。期间，多家行业合作商共同开发与协定了众多基于 SOAP 的规范，从包括寻址以及众多安全选项的基础规范，到诸如原子事务协作的企业协议。我在微软的团队，仍然非正式地称呼我们的产品为“Indigo”，它代表了整个开发与协商过程中耗费的所有心血。如果没有 IBM、微软以及其他合作伙伴对创建通用标准集的大力支持，在竞争如此激烈的企业领域几乎不可能存在开放标准的框架，更不可能具有支持多个开发商以及各种平台的多种实现。

诚然，WCF 的实现超出了预计需要花费的时间。标准的协定耗费了大量时间，毕竟我们不能只顾着发布自己的软件（Windows Communication Foundation, WCF），而不考虑它与我们的行业合作伙伴以及竞争者之间的互操作性。设计同样如此，对于那些具有分布式系统开发经验的客户而言，他们花费了大量时间学习以及掌握了我们之前提供的分布式系统技术，包括 ASP.NET 服务、Web 服务增强（WSE）、.NET Remoting、消息传输/MSMQ 以及企业服务/COM+，我们在发布软件的同时必须考虑这些客户。

在我刚才引用的技术清单中，包含了五种技术。如果使用非托管代码，则还有更多的技术平台。WCF 的其中一个最重要的设计目标就是通过简单的方式将这些技术集合起来，以一种方式进行编程。不管是构建一个队列应用程序、事务型的 N 层应用程序、P2P 客户端、RSS 种子服务器，还是构建自己的企业服务总线，都不再需要掌握那些只能解决部分问题的多种技术。我们只需要学习和使用 WCF 即可。这就是以一种方式编程的魅力所在。

本书展示了大量微软已经构建好的技术细节，它们可以作为您的应用程序与服务的基础。在本书中，作者以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。作为微软互联框架团队成员的我们，也为自己构建的这一产品深感自豪。我们为开发者提供了一个统一的分布式技术体系架构，它具有广泛的互操作性，全面提升了面向服务的特性。同时它还是易于学习的，有利于提高构建面向服务应用程序的生产力。作为当今最杰出的分布式系统专家之一，Juval 愿意倾尽心血全力介绍 WCF，我们不禁深感荣幸。我们有足够的信心相信，Juval 的著作能够帮助您理解人们为什么会对这一产品的问世以及它将创造的新的机遇而激动不已。这些人也包括我们、Juval 以及早期的用户社区。享受本书，开始构建您的第一个 WCF 服务吧。

前言

2001 年 8 月，我在微软首次了解到使用托管代码重写 COM+的技术细节。随后一切如常，直到 2002 年 7 月，在对 C#2.0 作战略性设计评审期间，负责 Remoting 的程序经理提出了一个宏伟的计划，试图将 Remoting 重写为开发者真正能够使用的技术。同时，微软也在寻求合作，共同为 ASMX 中的 Web 服务制定全新的安全规范，起草一系列附加的 Web 服务规格说明书。

到了 2003 年 7 月，我有机会体验了一个全新的事务型体系架构，它能够改善.NET 编程中关于事务处理的相关缺陷。当时，并没有一个稳定的编程模型能够统一那些独立的技术。直到 2003 年末，我有幸获邀参加一个由同行专家组成的小型团队，对代号为 Indigo 的开发平台进行战略性的设计评审。就我所知，这个开发团队可谓人才济济，汇聚了许多世界上最优秀的天才。在接下来的 2~3 年时间内，Indigo 一共经历了三代编程模型版本的演变。就在 2005 年早期发布了基于终结点驱动对象模型的版本之后，终于在当年 8 月逐渐稳定为一个固定的版本，同时更名为 Windows Communication Foundation（WCF）。要想得到开发者的众口称赞，可谓难于上青天，然而 WCF 却给了我们不同的诠释。对于 Web 服务的开发者而言，WCF 就是最终的应对互操作性的解决方案，实现了大多数行业标准。分布式应用程序的开发者则认为它简化了远程调用以及队列调用。系统开发者认为它具备下一代面向产品的特征，诸如事务与宿主，为应用程序提供了现成的基础功能模块。至于应用程序的开发者，WCF 则为他们构建应用程序提供了声明式的编程模型。而对于架构师，WCF 则是构建面向服务应用程序的最终选择。一言以敝之，WCF 涵盖了以上所有的一切，因为设计 WCF 的目的就是为了能够统一微软的下一代全新的技术。

对我而言，WCF 就是下一代开发者平台，它在很大程度上包容了最初的.NET 编程理念。任何.NET 开发者都可以使用 WCF，而不用考虑应用程序的类型、规模或者行业领域。WCF 是一门基础技术，它提供了生成服务与应用程序的“终南捷径”，完全符合我所认同的良好的设计准则。WCF 从一开始就是工程化的，能够简化应用程序的开发与部署，降低开发成本。WCF 服务用于构建面向服务的应用程序，不管这些程序是独立的桌面应用程序，还是 Web 应用程序和服务，还是高端的企业应用程序。

本书的结构

本书涵盖了所有设计开发基于 WCF 的面向服务应用程序所需的知识与技能。通过本书，你可以看到如何利用 WCF 内建的特性，例如服务托管、实例管理、并发管理、事务、离线队列调用以及安全。本书会为你展示如何使用这些特性并探究它们在这种特定的设计思路下的实现原理。你不仅能够了解到 WCF 编程技术，以及相关的系统知识，同时还包括了相应的设计方案、诀窍、最佳实践以及存在的缺陷。我之所以站在软件工程的立场阐述本书的每个主题与特征，是因为我期望它能够帮助读者不仅要成为一名 WCF 专家，而且还要成为一名优秀的软件工程师。本书带给您的这种认知能够使你如虎添翼，让你的应用程序在可维护性、可扩展性、可重用性以及高效性方面，更加符合软件工程的理念。

本书回避了许多 WCF 的实现细节，更多的是注重使用 WCF 的实用性与可行性：如何应用 WCF 技术？如何选择可行的设计原则与编程模型？本书大量使用了.NET 2.0 技术，从某种角度来说，本书也可以算是一本高级的 C#技术书籍。除此之外，本书包含了大量我所编写的套件类、工具类以及辅助类。这些内容可以提高你的开发效率，保障开发的 WCF 服务的质量。我还开发了一个基于 WCF 技术的小型框架，用以弥补一些设计缺陷，或者简化确切

的任务，使其能够自动化实现。在书中，我像介绍 WCF 技术那样，详细地介绍了这些工具、理念与技术。同时，我开发的框架则为你演示了如何对 WCF 进行扩展。

在过去的两年中，我在 MSDN 杂志上发表了大量关于 WCF 的文章。目前，我还在为杂志的基础专栏（Foundations Column）撰写 WCF 技术文章。我要感谢杂志社能够允许我将这些文章收录到本书中。如果你曾经阅读过这些文章，或许能够从本书的相关章节中发现它们的影子。比较而言，本书的章节更加全面，提供了 WCF 的多种视角、技术与实例，而且这些主题也与书中的其他章节紧密相连。

我在每一章中都系统地讲解了一个专题，深入探讨了这些专题的内容。然而，每一章又都依赖于前一章的内容，因此，我建议你最好按照先后顺序阅读本书。

以下是书中各章节以及附录的摘要。

第 1 章，WCF 基础

该章首先阐释了 WCF 的技术原理，并描述了 WCF 的基础概念和构建模块，例如地址、契约、绑定、终结点、托管以及客户端。在该章最后还讨论了 WCF 体系架构，它将是帮助我们理解后面章节的关键。该章假定读者已经了解面向服务的思想与优势。如果你不具备这方面的知识，可以首先阅读附录 A 的内容。即使你已经熟悉了 WCF 的基础概念，我仍然建议你至少能够快速地浏览该章的内容，它不仅能够巩固你已有的知识，更在于该章介绍的一些辅助类与技术术语有助于阅读全书。

第 2 章，服务契约

该章致力于介绍服务契约的设计与开发。首先，你会了解到一些有用的技术，包括服务契约的重载与继承以及其他高级技术。然后，该章深入探讨了如何设计以及分解契约，以利于服

务的重用、可维护性以及可扩展性。最后，展示如何通过公开契约元数据完成运行时的交互编程。

第 3 章，数据契约

如果没有实际存在的可共享的数据类型本身，如果没有使用相同的开发技术，应该如何处理客户端与服务之间的数据交换？在该章，你可以看到如何处理某些有趣的现实问题，例如数据版本控制，以及传递元素项集合的方式。

第 4 章，实例管理

究竟是哪一种服务实例处理哪一种客户端的请求？该章给出了问题之钥。WCF 支持多种服务实例管理、激活以及生命周期管理技术，这些技术与系统规模和性能息息相关。该章给出了每一种实例管理模式之间的关系，指导读者何时以及如何有效地使用它们。该章还介绍与之相关的主题，例如限流。

第 5 章，操作

随着对各种类型操作的处理，客户端能够调用服务，遵循相关的设计原则，例如如何改善和扩展基础功能，以支持回调的安装与销毁，管理回调端口与通道，提供类型安全的双向代理。

第 6 章，错误

该章全面介绍了服务将错误与异常返回给客户端的方式，毕竟，诸如异常与异常处理的构建都是一门特定的技术，无法穿越服务边界。该章介绍了错误处理的最佳实践，使开发者能够解除客户端错误处理与服务的耦合度。该章还演示了如何扩展以及改善 WCF 基础的错误处理机制。

第 7 章，事务

首先，该章从整体上介绍了使用事务的目的，然后讨论了事务服务的众多特征：事务管理架构、事务传播配置、WCF 提供的声明性事务支持，以及客户端创建事务的方式。最后，该

章讨论了与事务相关的设计原则，例如事务服务状态管理与实例模式。

第 8 章，并发管理

WCF 提供了一种强大而简单的声明方式，用来管理客户端与服务的并发与同步。该章展现了诸多高级技术，例如回调、重入、线程关联度、同步上下文以及避免死锁的最佳实践与原则。

第 9 章，队列服务

该章描述了客户端如何通过队列调用服务，从而支持异步与离线工作。该章首先介绍如何创建与配置队列服务，然后，重点讲解诸如事务、实例管理、故障以及它们对服务业务模型与实现造成的影响。

第 10 章，安全

通过将多个方面的任务分解为一些基本的要素，如消息传递、认证和授权，就可以揭开面向服务安全神秘的面纱。该章演示了如何为局域网和互联网应用程序等关键场景提供安全保障。最后，你可以看到我为声明式的 WCF 安全所编写的框架，设计为自动实现安全的设置，从而极大地简化对安全的管理。

附录 A，面向服务概述

附录 A 为那些希望了解面向服务的读者提供，介绍了我在面向服务的具体应用。附录定义了面向服务应用程序(而非通常所谓的架构)以及服务自身，检验了它在方法学方面的优势。附录还给出了面向服务的原则，通过大多数应用程序所需要的实用要点，强化了面向服务的抽象原则。

附录 B，服务发布与订阅

附录 B 展现了我定义的框架，它实现了发布-订阅事件管理的解决方案。框架可以使你只需要编写一两行代码就能发布和订阅服务。发布-订阅模式属于第 5 章的内容，之所以将它放

入到附录中，是因为它使用了其他章节的内容，例如事务与队列调用。

附录 C，WCF 编码规范

基本上，附录 C 涵盖了全书提及的甚至于没有提及的最佳实践。规范在于阐释应该“如何做”以及“怎么做”，而不阐明其原因。隐藏在规范之中的基础原理可以在本书的其余部分找到。该规范同时还使用了本书讨论的辅助类。

对于读者的要求

本书假定读者是一名经验丰富的开发者，熟悉诸如封装与继承等面向对象的概念。我会利用读者现有的对对象和组件技术以及术语的认知，巩固对 WCF 知识的了解。读者应该对于.NET 以及 C# 2.0 的基础知识（包括泛型与匿名方法）有着清晰的了解。虽然本书大部分内容使用的是 C#语言，然而对于 Visual Basic 2005 的开发者而言，仍然具有参考价值。

怎样使用本书

若要使用本书，需要安装.NET 2.0、Visual Studio 2005、.NET 3.0 的发布组件，以及.NET 3.0 开发的 SDK 和 Visual Studio 2005 的.NET 3.0 扩展版。除非特别提示，本书适用的操作系统包括 Windows XP、Windows Server 2003 和 Windows Vista。同时，你还需要安装一些附加的 Windows 组件，如 MSMQ 和 IIS。

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）；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（客户端和服务器）、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-Orientation，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。

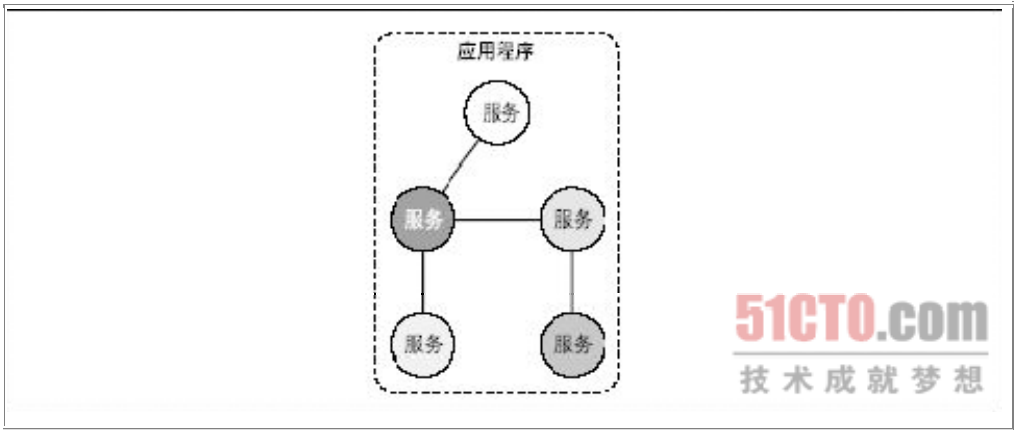

图 1-1：面向服务应用程序

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。

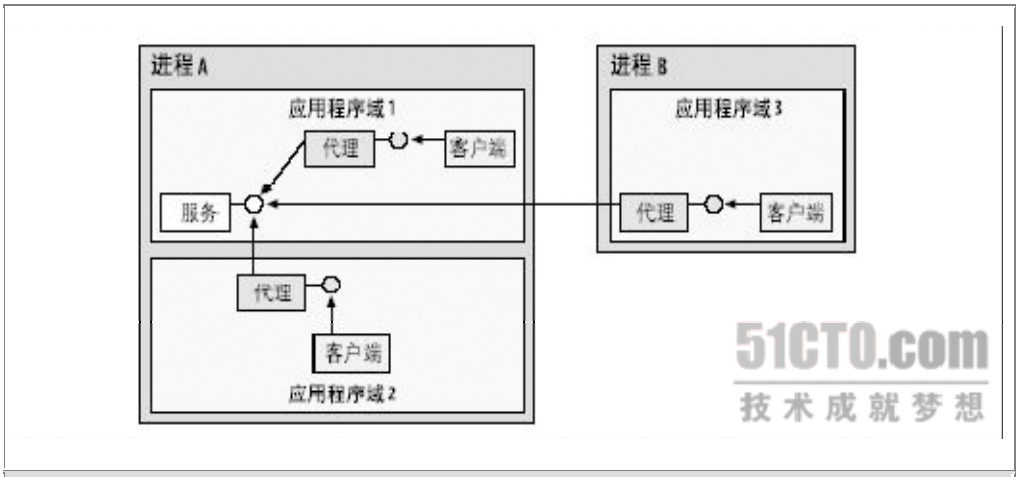

图 1-2：使用 WCF 实现相同机器通信

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。

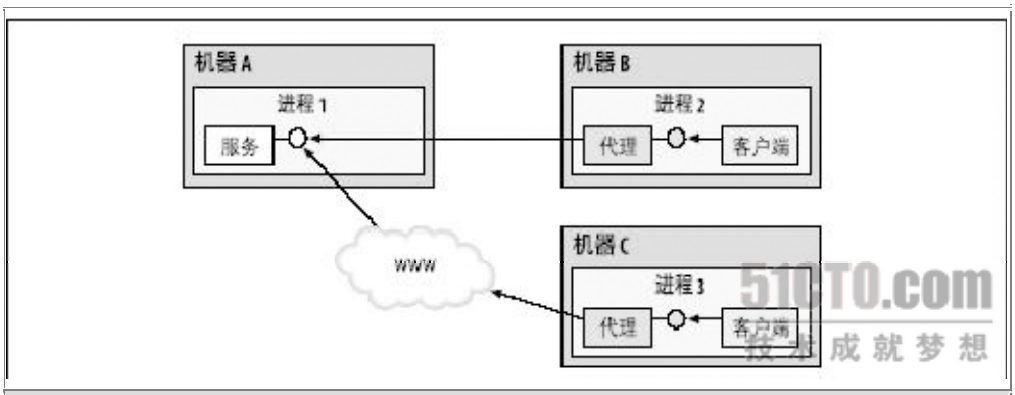

图 1-3：使用 WCF 实现不同机器通信

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
}
```

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 Service-Contract 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）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。

Win98 系统：c:\Windows c:\Windows\system

Winnt 和 Win2000 系统：

c:\Winntc:\Winnt\system32

Winxp 系统：c:\Windows

c:\Windows\system32

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：

Win98 系统：c:\Windows c:\Windows\system

Winnt 和 Win2000 系统：

c:\Winntc:\Winnt\system32

Winxp 系统：c:\Windows

c:\Windows\system32

```
<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 SeriviceHost 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>       <a
dd 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<T>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);
}
}
```

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 Service-Host<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。

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（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。

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
BasicHttpBinding	HTTP/HTTPS	Text, MTOM	Yes
NetTcpBinding	TCP	Binary	No
NetPeerTcpBinding	P2P	Binary	No
NetNamedPipeBinding	IPC	Binary	No
WSHttpBinding	HTTP/HTTPS	Text, MTOM	Yes
WSFederationHttpBinding	HTTP/HTTPS	Text, MTOM	Yes
WSDualHttpBinding	HTTP	Text, MTOM	Yes
NetMsmqBinding	MSMQ	Binary	No
MsmqIntegrationBinding	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

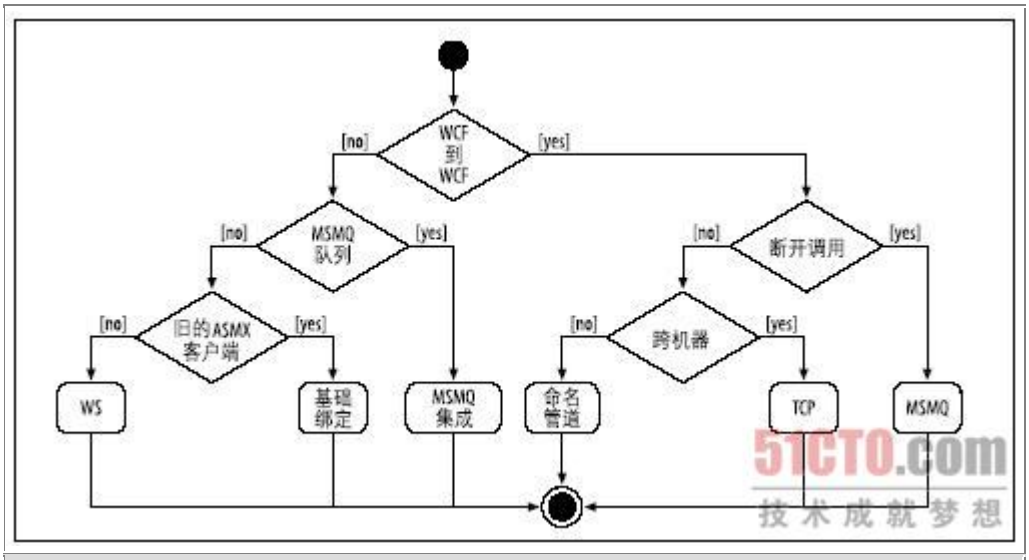

图 1-4：选择绑定

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）。

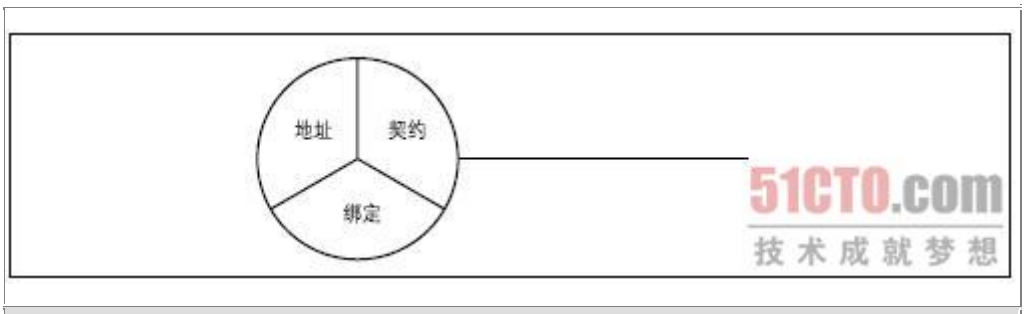

图 1-5：终结点

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 互不不同。

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。

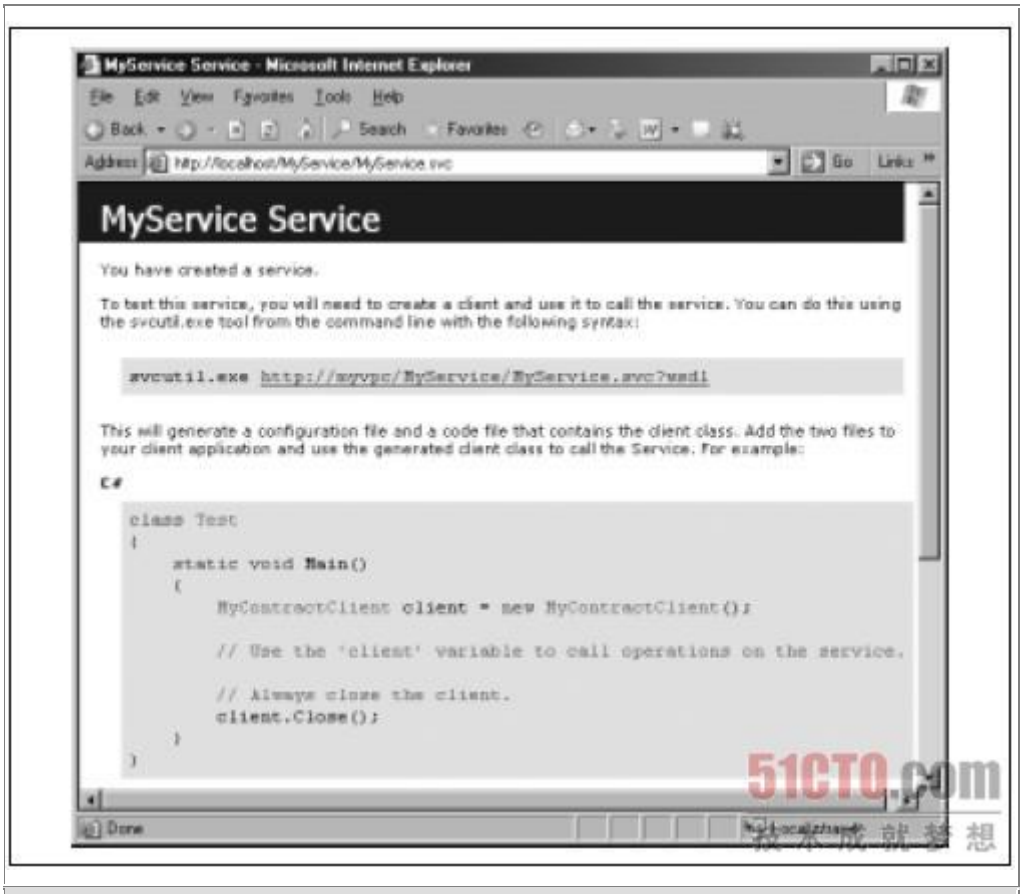

图 1-6：服务的确认页面

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。Service-Description 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。

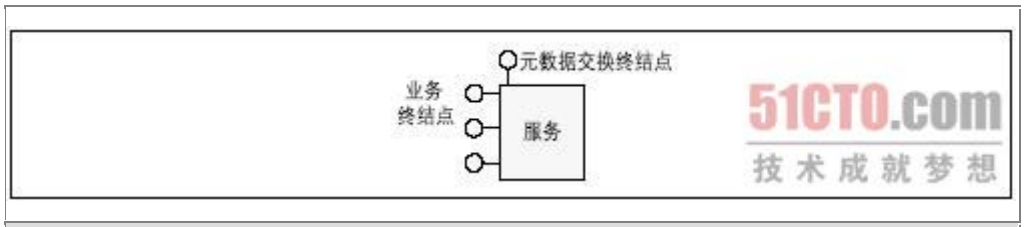

图 1-7：元数据交换终结点

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，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。

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> mexEndPoint= delegate(ServiceEndpoint endpoint)
{
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 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。AddAllMexEnd-Points()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）。

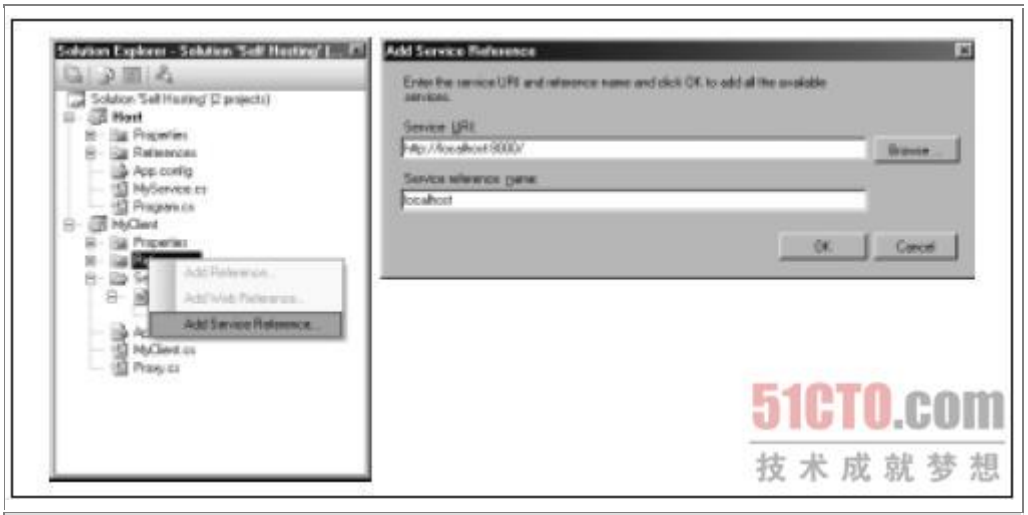

图 1-9：使用 Visual Studio 2005 生成代理

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
{
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

```
    address  = "http://localhost:8000/MyService/"
binding  = "wsHttpBinding"
contract = "IMyContract"
/>
```

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

```
      address   = http://localhost:8000/MyService/
binding   = "wsHttpBinding"
contract = "IMyContract"
/>

       address   =
"net.tcp://localhost:8001/MyService/"
binding   = "netTcpBinding"
contract = "IMyContract"
/>

       address   =
"net.tcp://localhost:8002/MyService/"
binding   = "netTcpBinding"
contract = "IMyOtherContract"
/>
```

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

```
   address   =
"net.tcp://localhost:8000/MyService/"
bindingConfiguration = "TransactionalTCP"
binding   = "netTcpBinding"
contract = "IMyContract"
/>









          transactionFlow = "true"
/>
```

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

```
  address  = "net.pipe://localhost/MyPipe"
binding  = "netNamedPipeBinding"
contract = "IMyContract"
/>




      address  = "net.pipe://localhost/MyPipe"
```

```
binding   = "netNamedPipeBinding"
contract = "IMyContract"
/>
```

注意，进程内宿主使用了命名管道绑定。

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”。

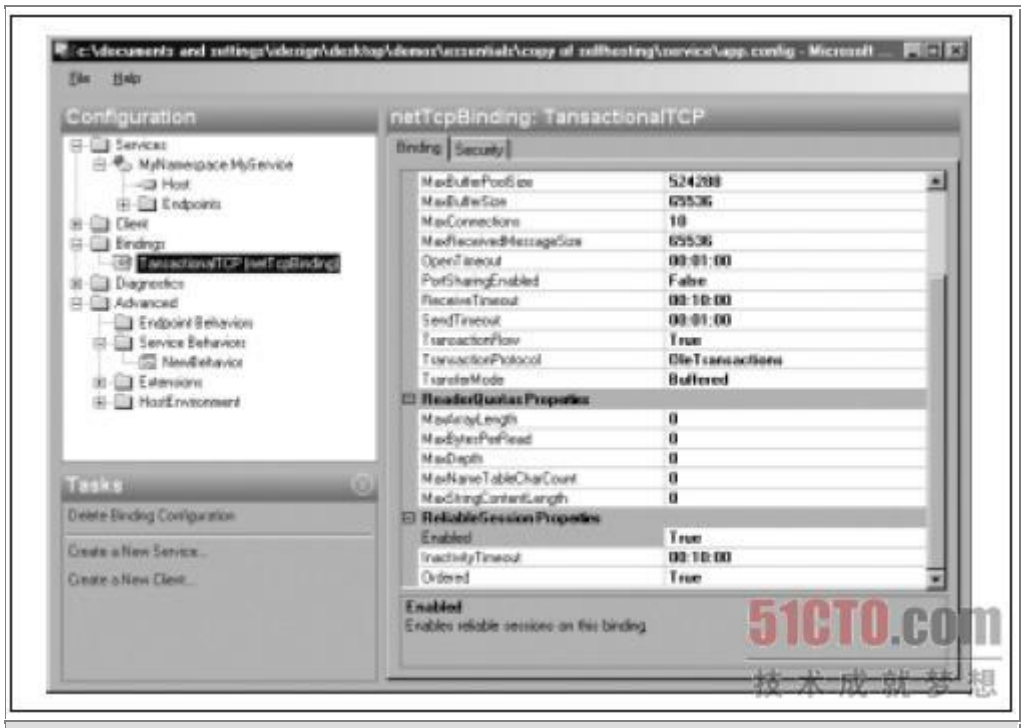

图 1-10：用于编辑宿主与客户端配置文件的 SvcConfigEditor

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 类，定义如下：

```
public abstract class ClientBase :
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 类通过泛型类型参数识别代理封装的服务契约。ClientBase 的 Channel 属性类型就是泛型参数的类型。ClientBase 的子类通过 Channel 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节名。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：

```
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：

```
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：

<ENDPOINT
...
binding = "wsHttpBinding"
bindingConfiguration = "LongTimeout"
```

```
...
/>
```

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两种技术各有所长，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。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。

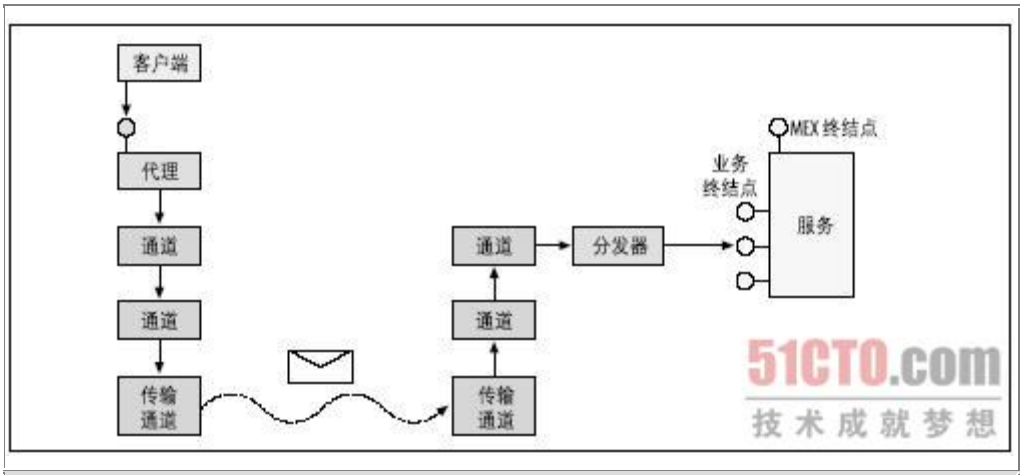

图 1-11：WCF 体系架构

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。

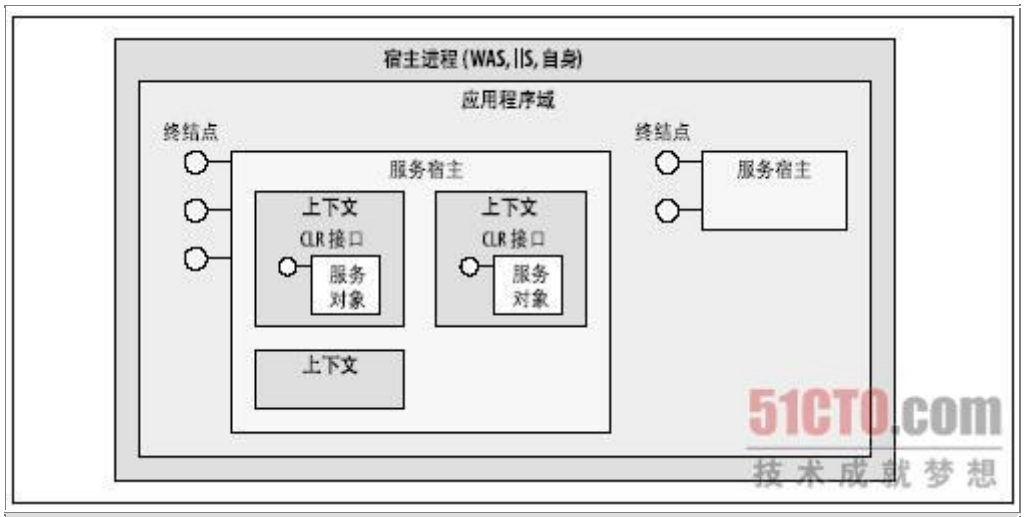

图 1-12：WCF 宿主体系架构

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

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 所面临的最大挑战，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。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
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
{
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

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：

```
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 ResponseAction 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：

```
 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);
}
//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。

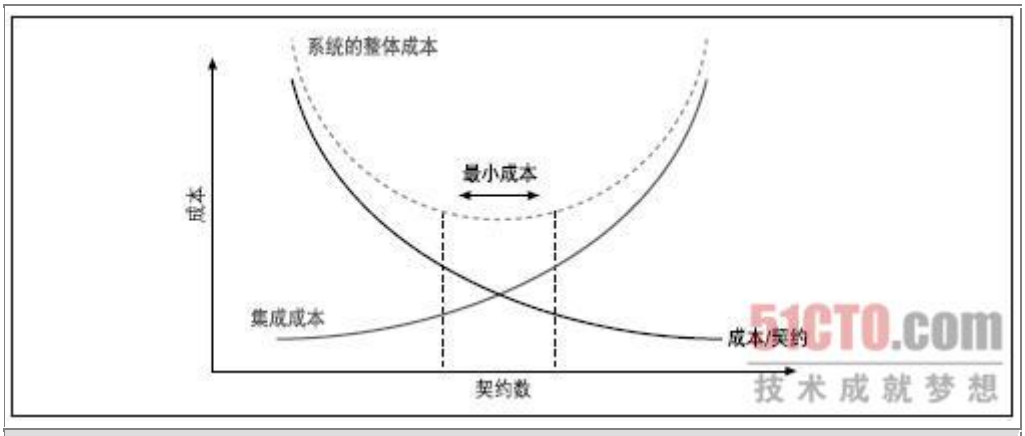

图 2-1：平衡服务的个数与规模

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。

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
{
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
}
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 MetadataExchange-Client 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的接收消息大小时，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。Contract-Description 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 Contract-Description 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);
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 MetadataExchange-Client 实例。MetadataExchangeClient 实例能够获取元数据，返回终结点集合。但是，GetEndpoints()方法返回的终结点集合不是 ServiceEndpoint-Collection 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;
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 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在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是自治的，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相匹配，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 类的实例，它的定义如例 6-1 所示：

例 6-1：FaultException 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 : FaultException
{
public FaultException(T detail);
public FaultException(T detail,string reason);
public FaultException(T detail,FaultReason reason);
//更多成员
}
```

FaultException 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 类型。

FaultException 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 异常。

例 6-2：抛出 FaultException 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(exception);
}
return number1 / number2;
}
//其余的实现
}
```

除了 FaultException 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();
```

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 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(exception,"number2 is
0");
```

如果要求本地化，更有效的方式是传递 FaultReason 对象。

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在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是自治的，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相匹配，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 类的实例，它的定义如例 6-1 所示：

例 6-1：FaultException 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 : FaultException
{
public FaultException(T detail);
public FaultException(T detail,string reason);
public FaultException(T detail,FaultReason reason);
//更多成员
}
```

FaultException 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 类型。

FaultException 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 异常。

例 6-2：抛出 FaultException 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(exception);
}
return number1 / number2;
}
//其余的实现
}
```

除了 FaultException 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();
```

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 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(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 IncludeException-DetailInFaults 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）。

•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)
{...}
}
}
```

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 faultException = new FaultException(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 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 falut 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 中的 T 类型，同时 FaultException 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 类型。例如，给定这样的服务契约：

```
[ServiceContract]
interface IMyContract
{
[OperationContract]
[FaultContract(typeof(InvalidOperationException))]
void MyMethod();
}
```

如果下游对象抛出一个 InvalidOperationException 异常，ProvideFault()方法就会将它提升为 FaultException 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 faultException =
new FaultException(
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

IErrorHandler 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 : IErrorHandler
{
public bool HandleError(Exception error)
{
try
{
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。

```
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 ErrorHandlers
{get;}
//更多成员
}
```

若要安装自己定制的 IErrorHandler 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 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
{}
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 IErrorHandler 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 IErrorHandler 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 : IErrorHandler
{...}
class MyService : IMyContract,IServiceBehavior
{
public void
ApplyDispatchBehavior(ServiceDescription
description,
ServiceHostBase host)
{
IErrorHandler handler = new MyErrorHandler();
```

```
foreach(ChannelDispatcher dispatcher in
host.ChannelDispatchers)
{
dispatcher.ErrorHandlers.Add(handler);
}
}
public void Validate(...)
{}
public void AddBindingParameters(...)
{}
}
```

例 6-14 定义的服务自身实现了 IServiceBehavior 接口。在 ApplyDispatchBehavior()方法中，服务获取了分发器集合，并将 MyErrorHandler 类的实例添加到每个分发器中。事实上，我们不必依靠一个外部类去实现 IErrorHandler 接口，服务类自身就可以直接实现 IErrorHandler，如例 6-15 所示。

例 6-15：实现 IErrorHandler 的服务类

```
class MyService :
IMyContract,IServiceBehavior,IErrorHandler
{
public void
```

```
ApplyDispatchBehavior(ServiceDescription
description,
ServiceHostBase host)
{
foreach(ChannelDispatcher dispatcher in
host.ChannelDispatchers)
{
dispatcher.ErrorHandlers.Add(this);
}
}
public bool HandleError(Exception error)
{...}
public void ProvideFault(Exception
error,MessageVersion version,
ref Message fault)
{...}
//更多成员
}
```

错误处理特性

例 6-14 和例 6-15 存在的共同问题是它们会污染包含了 WCF 基础功能模块的服务类代码。

与只关注于业务逻辑的服务不同，这些服务同时还包含了错误扩展逻辑。不过，开发者可以使用编写的 ErrorHandlerBehaviorAttribute 特性，以声明方式提供相同的功能实现。

ErrorHandlerBehaviorAttribute 特性的定义如下：

```
public class ErrorHandlerBehaviorAttribute :
Attribute,IErrorHandler,
IServiceBehavior
{
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，而宿主添加这样的一个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 : IList,...
{
public void Add(T item);
//更多成员
}
```

```
public abstract class KeyedCollection : Collection
{...}
public class KeyedByTypeCollection :
KeyedCollection
{...}
public class ServiceDescription
{
public KeyedByTypeCollection 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 中，并实现了执行的自动化：

```
public class ServiceHost : ServiceHost
{
public void AddErrorHandler(IErrorHandler
```

```
errorHandler);
public void AddErrorHandler();
//更多成员
}
```

ServiceHost 定义了两个重载版本的 AddErrorHandler()方法。接收参数 errorHandler 的方法建立了 IErrorHandler 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 IErrorHandler 接口的类，而不是实现 IServiceBehavior 接口的类：

```
class MyService : IMyContract
{...}
class MyErrorHandler : IErrorHandler
{...}
ServiceHost host = new ServiceHost();
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 host = new ServiceHost();
host.AddErrorHandler();
host.Open();
```

实际上，在后一个例子中，ServiceHost 的内部使用了一个

ErrorHandler-BehaviorAttribute 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 : ServiceHost
{
class ErrorHandlerBehavior : IServiceBehavior,IErrorHandler
{
IErrorHandler m_ErrorHandler;
public ErrorHandlerBehavior(IErrorHandler errorHandler)
{
m_ErrorHandler = errorHandler;
}
void
IServiceBehavior.ApplyDispatchBehavior(ServiceDescription
description,
ServiceHostBase host)
```

```
{
foreach(ChannelDispatcher dispatcher in
host.ChannelDispatchers)
{
dispatcher.ErrorHandlers.Add(this);
}
}
bool IErrorHandler.HandleError(Exception error)
{
return m_ErrorHandler.HandleError(error);
}
void IErrorHandler.ProvideFault(Exception
error,MessageVersion version,
ref Message fault)
{
m_ErrorHandler.ProvideFault(error,version,ref fault);
}
//其余实现
}
List m_ErrorHandlers = new List();
public void AddErrorHandler(IErrorHandler 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 IErrorHandler 引用对象同时支持 IServiceBehavior 接口，ServiceHost 定义了一个私有嵌套类 ErrorHandlerBehavior。它同时实现了 IErrorHandler 和 IServiceBehavior 接口。要创建 ErrorHandlerBehavior 对象，我们需要为它提供一个 IErrorHandler 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。IErrorHandler 接口的实现只不过是将引用指向之前保存的构造参数 IErrorHandler 对象。ServiceHost 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 IErrorHandler 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 IErrorHandler 和 IServiceBehaivor 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 IErrorHandler 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 IErrorHandler 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 : IErrorHandler
{...}
class MyClient : IMyContractCallback,IEndpointBehavior
{
public void OnCallBack()
{...}
void
IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint
serviceEndpoint,
ClientRuntime behavior)
{
IErrorHandler 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 IErrorHandler，而不必通过外部类去实现 IErrorHandler：

```
class MyClient : IMyContractCallback,IEndpointBehavior,IErrorHandler
{
public void OnCallBack()
{...}
void IEndpointBehavior.ApplyClientBehavior(ServiceEndpoint
serviceEndpoint,
ClientRuntime behavior)
{
behavior.CallbackDispatchRuntime.ChannelDispatcher.ErrorHandlers.Ad
d(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.Ad
d(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。